doc. Ing. Pavel Šenovský, Ph.D.

Umělá inteligence v bezpečnosti

skripta
2. vydání

Ostrava, 2024

Umělá inteligence v bezpečnosti
tento text neprošel jazykovou úpravou

Vysoká škola báňská - Technická univerzita Ostrava, Fakulta bezpečnostního inženýrství

Obsah

Seznam obrázků **7**

Seznam tabulek **9**

Seznam výpisů kódu **12**

Úvod **13**

1 Umělá inteligence **17**
1.1 Strojové zpracování jazyka . . . 17
1.2 Expertní systémy . . . 20
1.3 Multiagentní systémy . . . 21
1.4 Big data a průmysl 4.0 . . . 22
1.5 Současný stav řešení umělé inteligence . . . 26

2 Úvod do R **33**
2.1 Instalace prostředí . . . 33
2.2 RStudio . . . 35
2.3 Základy jazyka R . . . 36

3 Neuronové sítě **47**
3.1 Neuron . . . 48
3.2 Neuronové sítě a jejich adaptace . . . 50
3.3 Aplikace neuronových sítí v bezpečnostních oborech . . . 57

4 Neuronové sítě - cesta k hloubkovému učení **61**
4.1 Knihovna neuralnet - analýza obrazové informace . . . 61
4.2 Knihovna MXNet . . . 65
4.2.1 Instalace MXNet na macOS . . . 65
4.2.2 Instalace MXNet na Linux . . . 66
4.2.3 Instalace MXNet na Windows . . . 67
4.3 Hloubkové učení - symbolické programování . . . 67
4.4 MNist dataset - MXNet klasická neuronová síť . . . 70
4.5 MNist dataset - MXNet a konvoluční síť . . . 73
4.6 Generativní AI . . . 75
4.7 Etika nasazení systémů na bázi umělé inteligence . . . 80
4.8 Bezpečnost AI . . . 82

5 Modelování znalostí **85**
5.1 Historie jazyka UML . . . 85
5.2 Diagramy jazyka UML ... ve Star UML . . . 87
5.3 Třídní diagram . . . 88
5.4 Model jednání (Use Case Diagram) . . . 91
5.5 Stavový diagram (State Diagram) . . . 93
5.6 Scénáře činností (sequence diagram, sekvenční diagram) . . . 95
5.7 Diagram spolupráce (collaboration diagram) . . . 96
5.8 Diagram činností (Activity Diagram) . . . 97

5.9 Diagram komponent (Component Diagram) . . . 97
5.10 Diagram nasazení (Deployment Diagram) . . . 98
5.11 UML závěr . . . 99

6 Expertní systémy **101**
6.1 Základy expertních systémů . . . 101
6.2 Expertní systém CLIPS . . . 103
6.2.1 Šablony CLIPS . . . 103
6.2.2 Definice faktů . . . 105
6.2.3 Pravidla . . . 105
6.3 Aplikace expertních systémů v bezpečnosti . . . 107

7 Celulární automaty **111**
7.1 Počátky buněčných automatů – Conwayova hra života . . . 111
7.2 Wolframův 1D buněčný automat . . . 114
7.3 Boidi – Reynoldsův model shlukování ptáků . . . 115
7.4 Lindenmayerovy systémy . . . 117
7.5 Programovací jazyk Logo . . . 119
7.6 Aplikace celulárních automatů v bezpečnosti . . . 122

8 Multiagentní systémy **127**
8.1 Inteligentní agent . . . 127
8.2 Multiagentní systémy . . . 130
8.3 Aplikace v bezpečnosti . . . 132

9 Genetické algoritmy **135**
9.1 Evoluční strategie . . . 136
9.2 Křížení a mutace . . . 136
9.3 Možnosti nasazení genetických algoritmů . . . 138

Literatura **144**

Seznam zkratek **146**

Rejstřík **147**

Seznam obrázků

Umělá inteligence **17**
1.1 MS Graph a jeho integrace s dalšími službami Microsoftu (převzato z [58]) 18
1.2 Analýza struktury chipu A11 Bionic (převzato z [72]) 19
1.3 Podoblasti umělé inteligence a jejich vzájemný vztah 23
1.4 Automatizovaná výrobní linka pro výrobu automobilů (převzato z [19]) 24
1.5 Transformace průmyslové výroby v čase (převzato z [70]) 25
1.6 Hype cyklus pro rok 2023 podle průzkumu společnosti Gardner (převzato z [21]) . . . 27
1.7 Point cloud vytvořený analýzou zachyceného obrazu (převzato z [60]) 28
1.8 Identifikace silnic v obrazu (převzato z [60]) 28
1.9 Model silnice (převzato z [60]) 29
1.10 Simulované prostředí křižovatky (převzato z [60]) 29
1.11 Situace (odlišná od předchozích) doplněná popisky objektů (převzato z [60]) 30

Úvod do R **33**
2.1 Rozhraní RStudio v OS X 35
2.2 Data Viewer - proměnná tornado_data 36
2.3 Párové srovnání vlastností kosatců v datové sadě IRIS 42
2.4 Délka vs šířka listu kosatců v datasetu IRIS (bodový graf) 43
2.5 Dostupné typy vykreslování ve funkci plot 44
2.6 Rozložení délky listu dataset IRIS (histogram) 44
2.7 Délka listu dataset iris (boxplot) 45
2.8 Rozložení počtu prodejů podle jejich úspěšnosti (pie) 45

Neuronové sítě **47**
3.1 Schéma neuronu (převzato z [23]) 48
3.2 Snímek neuronu pořízený mikroskopem (převzato z [22]) 49
3.3 Schématické znázornění modelu perceptronu 49
3.4 Jednoduchá neuronová síť pro OCR 50
3.5 Jednoduchá neuronová síť pro OCR 51
3.6 Schéma vícevrstevné sítě 51
3.7 Neuronová síť pro predikci hodnot funkce $y = x^2$ 53
3.8 Funkce $y = x^2$ - srovnání modelu neuronovou sítí a lineární regresí 54
3.9 Přeučený model 55
3.10 Klasifikace klientů banky neuronovou sítí 56

Neuronové sítě - cesta k hloubkovému učení **61**
4.1 Příklady jednotlivých cifer z datasetu MNIST (převzato z [76]) 62
4.2 Neuronová síť pro predikci v knihovně neuralnet pro rozlišení mezi číslicemi 3 a 8 . . . 64
4.3 Graf aktivační funkce ReLU 68
4.4 Konvoluční neuronová síť (převzato z [40]) 69
4.5 Demonstrace operace subsampling (pooling) 69
4.6 Vývoj přesnosti adaptace po iteracích 72

4.7 Vývoj přesnosti adaptace konvoluční sítě po iteracích . . . 74
4.8 MS Copilot (MS Bing Chat) . . . 76
4.9 Google Bard . . . 76
4.10 LM Studio - vyhledání LLM . . . 78
4.11 LM Studio - chat . . . 79
4.12 Typologie útoků na generativní AI (převzato z [78]) . . . 83

Modelování znalostí **85**
5.1 Rozhraní programu StarUML ve verzi 3.1 na MacOS . . . 87
5.2 Třídy realizované v CASE StarUML . . . 88
5.3 Typy asociací podporované UML diagramy . . . 89
5.4 Asociace mezi třídami . . . 90
5.5 Příklad použití kompozice v třídním diagramu . . . 90
5.6 Příklad použití agregace v třídním diagramu . . . 91
5.7 Příklad použití generalizace/dědičnosti v třídním diagramu . . . 91
5.8 Příklad použití modelu jednání - realizace elektronického obchodu . . . 92
5.9 Rozšíření modelu jednání . . . 92
5.10 Příklad modelu jednání pro zásah s přítomností nebezpečných látek . . . 93
5.11 Stavový diagram - struktura modelu . . . 94
5.12 Stavový diagram - schématický příklad zdolání mimořádné události . . . 95
5.13 Scénář činnosti - zjednodušený scénář zdolání požáru . . . 96
5.14 Diagram spolupráce - zjednodušený scénář zdolání požáru . . . 97
5.15 Diagram činností - zjednodušený průběh činností v jednom dni . . . 97
5.16 Diagram komponent - zjednodušená vizualizace závislostí komponent informačního systému . . . 98
5.17 Diagram nasazení - zjednodušený diagram nasazení informačního systému z předchozího příkladu . . . 99

Expertní systémy **101**
6.1 Struktura expertního systému . . . 102
6.2 Třídní diagram - třída nebezpečná látka . . . 104
6.3 Activity diagram procesu identifikace látky na základě UN kódu . . . 106

Celulární automaty **111**
7.1 Úplné okolí (vlevo) vs von Neumanovo okolí (vpravo) . . . 112
7.2 Conwayova hra života . . . 113
7.3 Conwayova hra života - stabilní (neměnné) struktury zleva: blok, úl, list, loď, vana . . . 114
7.4 Conwayova hra života - oscilující struktury . . . 114
7.5 Conwayova hra života - pohybující se struktury . . . 115
7.6 Wolframův 1D automat - vizualizace pravidla 30 . . . 116
7.7 Wolframovy 1D automaty - CA1 - CA4 . . . 116
7.8 Ulita mlže *Conus textile* (převzato z [52]) . . . 117
7.9 Hejno ptáků (převzato z [30]) . . . 117
7.10 Pravidla chování ptáků v hejnu [67] . . . 118
7.11 Simulace hejna - Flocking Perspective Demo (NetLogo) . . . 118
7.12 Sierpińského trojúhelník v NetLogu . . . 119
7.13 Modelování travin pomocí fraktálů (převzato z [74]) . . . 119
7.14 GUI NetLogo . . . 121
7.15 Model Fire po 27 iteracích (obrázek vygenerován pomocí appletu [73]) . . . 122
7.16 Struktura experimentálního modelu požáru v uzavřených prostorách (převzato z [88]) 123
7.17 Simulace rozvoje požáru v uzavřeném prostoru (převzato z [88]) . . . 123

Multiagentní systémy **127**
8.1 Vnitřní konstrukce inteligentního agenta 128
8.2 Reakce různých typů agentů na vnější podnět (zdroj světla v místnosti) 129
8.3 Rozdíl v pohybu agentů 130
8.4 Přímá komunikace mezi agenty 131
8.5 Komunikace agentů prostřednictvím facilitátora 131
8.6 Rámec multiagentního systému pro abstrakci modelování kritické infrastruktury (převzato z [64]) 132
8.7 Simulace evakuace budovy v balíku SIMULEX (převzato z [43]) 133
8.8 Simulace urbanEXODUS lesní požár-chodci-automobily (převzato z [33]) 134

Genetické algoritmy **135**
9.1 Scénáře generační obměny v generickém algoritmu 137
9.2 Možné řešení problému obchodního cestujícího (převzato z [8]) 139

Seznam tabulek

Neuronové sítě **47**
3.1 Trénovací množina funkce $y = x^2$ 52

Neuronové sítě - cesta k hloubkovému učení **61**
4.1 Confusion matice binární klasifikace čísel 3 a 8 adaptovaných knihovnou Neuralnet . . 64
4.2 Confusion matice pro OCR čísel - MXNet s plně propojenými vrstvami 72
4.3 Confusion matice pro OCR čísel - MXNet - konvoluční síť 74
4.4 Srovnání velikosti VRAM grafických karet NVidia RTX řady 4000 a AMD RX 7000 (pro desktop) 78

Modelování znalostí **85**
5.1 Přehled známějších UML CASE nástrojů 86

Listings

Úvod do R **33**
2.1 Instalace R a RStudio v distribucích Linux založených na Debian . . . 34
2.2 Základní operátory v R a jejich použití . . . 37
2.3 Základní funkce v R . . . 37
2.4 Manipulace s vektory v R . . . 37
2.5 Spojování vektorů v R . . . 38
2.6 Realizace Data framu spojením vektorů v R . . . 38
2.7 Manipulace se sloupci Data framu v R . . . 38
2.8 Aplikace filtrů na sloupce Data framu v R . . . 38
2.9 Řazení v Data framu . . . 39
2.10 Použití faktorů v R . . . 39
2.11 Vytvoření ordinální proměnné v R . . . 39
2.12 Načtení CSV souboru . . . 40
2.13 Zápis dat do textového souboru . . . 40
2.14 Základní statistické funkce v R . . . 41
2.15 Vykreslování základních typů grafů pro dataset IRIS . . . 41
2.16 Funkce *plot* - jednotlivé typy grafů . . . 42
2.17 Tvorba koláčových grafů v R . . . 43

Neuronové sítě **47**
3.1 Použití knihovky neural net pro natrénování funkce druhé mocniny . . . 52
3.2 Klasifikace klientů banky podle bonity . . . 55

Neuronové sítě - cesta k hloubkovému učení **61**
4.1 Pužití knihovny neural net pro natrénování a validování rozpoznávání čísel 3 a 8 z datasetu MNIST v R . . . 63
4.2 Instalace Homebrew ma macOS . . . 65
4.3 Aktualizace některých systémových utilit . . . 65
4.4 Aktualizace Homebrew a jím nainstalovaných aplikací a utilit . . . 66
4.5 Instalace OpenCV a OpenBLAS pro macOS pomocí Homebrew . . . 66
4.6 Instalace MXNet v R . . . 66
4.7 MXNet - řešení OCR ručně zadaných čísel s využitím plně propojených vrstev . . . 70
4.8 MXNet - řešení OCR ručně zadaných čísel s využitím konvoluční sítě . . . 73

Expertní systémy **101**
6.1 Šablona informace o nebezpečné látce . . . 104
6.2 Definice atributu v šabloně . . . 104
6.3 Definice faktu: základní údaje o nebezpečných látkách . . . 105
6.4 Pomocna fakta ovlivňující výběr otazky nebo pravidel . . . 106
6.5 Načtení UN kódu od uživatele . . . 106
6.6 Pravidlo pro vyhledání UN kódu v seznamu známých faktů v bázi znalostí . . . 107
6.7 Předpoklady pro porovnání se vzorem . . . 107
6.8 otestování faktů na shodu se zadaným UN kódem . . . 107

Celulární automaty **111**
7.1 Pravidlo 22 Wolframova 1D automatu - implementace v jazyku Logo 120
7.2 Model rozvoje požáru v uzavřeném prostoru realizovaný v NetLogo 123

Multiagentní systémy **127**
8.1 Komunikační akt v jazyku KQML 131

Úvod

Vážený studente, dostává se Vám do rukou učební text předmětu *Umělá inteligence v bezpečnosti*. Tento text navazuje filozoficky na starší texty určené pro studium předmětu *Expertní systémy* [89] a zaměřuje se na problematiku umělé inteligence a její dopady na fungování moderní společnosti stejně jako do bezpečnosti.

V posledních dvou letech jsme byli svědky masivního nástupu umělé inteligence do prakticky všech oblastí našeho života. velké jazykové modely (Large Language Model (LLM)) jako je např. Chat-GPT jednoznačně ukázaly, že umělá inteligence je zde a je dostupná ve formě, která sice na jedné straně není zcela bez problémů, na straně druhé je schopna efektivně řešit řadu úloh, se kterými se setkáváme dnes a denně.

Efektivní zvládnutí umělé inteligence, jako nástroje, bude proto zřejmě jednou ze základních dovedností, která takovému uživateli umožní výrazně zvýšit efektivitu řady pracovních činností, které vykonává a to bez ohledu na obor, ve kterém tyto činnosti vykonává.

K této změně došlo překvapivě rychle. V roce 2017 byl uveřejněn článek [79] popisující mechanismus pozornosti (attention) jako nejdůležitějším konstrukčním prvku, na kterém jsou současné LLM modely postaveny. během nějakých 5-ti let jsme se dostali od teoretického konceptu k prakticky nasazovaným modelům, trénovaných na podstatné části vědění, které lidstvo shromáždilo za dobu celé své existence. Tyto modely jsou navíc dostupny v řadě případů zdarma k všeobecnému použití.

Tyto změny si vyžádaly do jisté míry úpravu textu těchto skript pro jejich druhé vydání. Skripta jsou nově organizována do tří základních částí:

Začneme právě **velkými jazykovými modely**, jelikož se jedná v současnosti o nejprogresivnější oblast umělé inteligence a zároveň se jedná o nástroj, který potenciálně budeme používat neustále. V této sekci textu se nejprve podíváme na tuto problematiku uživatelsky a popíšeme kromě pozitivních stránek také negativní. Abychom lépe pochopili principy fungování vysvětlíme si funkci umělých neuronových sítí, nejprve obecně a následně také pro některé speciální typy sítí. Zaměříme se především na:

- konvoluční sítě - používané především pro zpracování obrazové informace
- transformátory - velmi efektivní pro jazykové modely

V druhé části se zaměříme na problematiku obecnější **analytiky**. Konkrétně si něco povíme o jazyku Unified Modeling Language (UML). Jedná se o obecný grafický modelovací jazyk, který umožňuje systematickou dekompozici problému. Ke ale také často používán pro návrh systémů a komunikaci tohoto návrhu s dalšími osobami, které na něm participují nebo třeba budoucími uživateli systému.

UML je ale mnohem univerzálnější - umožňuje mimo jiné také dokumentovat postupy nebo procesy, specifikovat datové struktury a řadu dalších aplikací, které můžete využít při práci na semestrálních projektech v dalších předmětech a po dokončení studia také v praxi.

V třetí a poslední části skript se zaměříme na další **metody** vycházející z umělé inteligence a jejich možné praktické aplikace.

Studium nevyžaduje nějaké speciální předchozí znalosti nebo schopnosti - vyžaduje ale ochotu naučit se něco nového, vyzkoušet si prakticky některé postupy, protože prostým přečtením skript a „naučením“ se teorie znalosti v potřebné míře prostě nenabudete.

Na druhou stranu v tomto ohledu není předmět *Umělá inteligence v bezpečnosti* nijak výjimečný, protože výše uvedené platí pro takřka všechny předměty, které jste v průběhu studia absolvovali nebo ještě absolvujete.

Organizace textu

Předtím, než přejdeme k vlastnímu výkladu si dovolím přidat několik poznámek k organizaci skript.

Text je organizován do kapitol, přičemž každá z nich má stanoveny určité didaktické cíle, které by prostudováním kapitoly měly být dosaženy.

Tyto cíle naleznete v náhledu kapitoly společně s krátkým zasazením problému do širšího kontextu. Náhled je doplněn odhadem času nutného pro prostudování problémové oblasti. Mějte Prosím na paměti, že tento časový údaj je pouze orientační, nebuďte proto prosím smutní nebo naštvaní, když ve skutečnosti budete kapitole věnovat o něco méně nebo více času.

Zároveň pokud Vás problém zaujme neváhejte a ponořte se hlouběji. Nástroje a postupy, se kterými se v průběhu studia seznámíte, jsou dostatečně robustní, aby Vám umožnily růst až na úroveň řešení problémů tak rozsáhlých, že vyžadují práci celých týmů a použití výkonu superpočítače. (Než se ale dostanete na takovou úroveň vystačíte si sami a s běžně dostupnými počítači a to i pro řešení problémů podstatně složitějších než ty představované v těchto skriptech.)

Za kapitolou pak následuje shrnutí, ve kterém budou zdůrazněny informace, které byste si rozhodně měli zapamatovat (určitě Vám ale neuškodí, pokud si jich zapamatujete více).

To, že jste správně pochopili probíranou látku, si budete moci ověřit pomocí kontrolních otázek a testů, které by Vám měly poskytnout dostatečnou zpětnou vazbu k rozhodnutí, zdali jít dále nebo si vyhradit delší čas na opakování.

Kontrolní otázky tak nepředstavují seznam otázek, které budou použity u zkoušky, ale jsou čistě Vaším zpětnovazebním prvkem.

Pro zjednodušení orientace je také v textu zaveden systém ikon:

Průvodce studiem
Slouží pro seznámení studentů s látkou, která bude v kapitole probírána.

Čas nutný ke studiu
Představuje odhad doby, který budete potřebovat k prostudování celé kapitoly. Jedná se pouze o orientační odhad, neznepokojujte se proto, pokud Vám studium bude trvat o něco déle nebo budete hotovi rychleji.

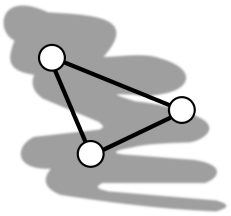
Vysvětlení, definice, poznámka
U této ikony najdete vysvětlující text, poznámku k probíranému tématu, která problém uvede do širších souvislostí, popřípadě důležitou definice.

Kontrolní otázky
Na závěr každé kapitoly je zařazeno několik otázek, které prověří, zda jste problematice kapitoly dostatečně porozuměli. Pokud nebudete vědět odpověď na některou otázku, je to signál pro Vás, abyste se ke kapitole vrátili.

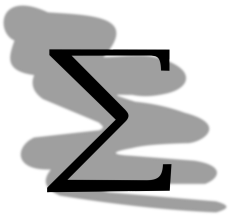
Skripta jsou zamýšlena jako pracovní nástroj pro Vás. Pokud tedy narazíte na nejasnosti, nebo dokonce na chybu, byl bych rád, abyste si takové poznatky nenechaly pro sebe, ale zaslaly je na můj e-mail: pavel.senovsky@vsb.cz, tak ať je příští verze skript ještě lepší než ta současná.

Závěrem mi dovolte popřát Vám příjemné čtení a pokud skripta studujete v rámci přípravy na zkoušku úspěch při jejím složení.

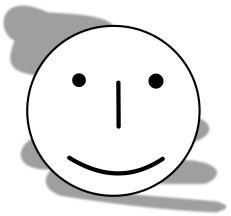
Příklad
Příklady obsahují praktické demonstrace diskutovaného problému.



Návaznosti
V tomto segmentu budou zmíněny další návaznosti probíraného tématu na další témata tohoto předmětu, ale také dalších předmětů.



Shrnutí
Obsahuje základní myšlenky kapitoly, kterým by měl být věnována zvláštní pozornost během studia.



Přestávka
Po obtížné části textu, nebo prostě občas jenom tak je nutné si udělat krátkou přestávku, načerpat síly k novému studiu.

doc. Ing. Pavel Šenovský, Ph.D.

Kapitola 1

Umělá inteligence

Průvodce studiem
V této kapitole se zamyslíme nad tím, proč se vůbec zabývat umělou inteligencí, jak ji chápat. Vytvoříme si také určitý rámec, do kterého postupně budeme přidávat vědomosti z různých oblastní umělé inteligence.

Po prostudování této kapitoly budete
vědět
- jakým způsobem dělíme umělou inteligenci
- důvody seřazení jednotlivých kapitol
- vzájemnou provázanost kapitol

znát - něco málo o historickém vývoji umělé inteligence.

Čas pro studium
Pro prostudování kapitoly budete potřebovat přibližně hodinu až dvě.

Svět kolem nás se mění, přesto některé věci zůstávají stále stejné. Dobrým příkladem je třeba tvrzení, že *umělá inteligence má potenciál změnit svět*. Toto tvrzení se přitom objevuje v literatuře ale také běžném hovoru již desítky let. Přesto univerzální umělá inteligence není stále dostupná (dokonce v současnosti není zcela jisté, zda takovou univerzální umělou inteligenci je vůbec možné sestavit).

V průběhu času se ale výrazným způsobem posunulo vnímání toho, co vlastně umělá inteligence je a jakým způsobem by mohla naši společnost změnit.

Základy oboru umělé inteligence položila trojice velikánů: Norbert Wiener (kybernetika), Claude Shannon (teorie informace) a Alan Turing (teorie výpočtů). V tomto smyslu jsou ale základy umělé inteligence totožné se základy informatiky jako takové. Konečně jsme se společně něco málo z oblasti teorie informace dozvěděli v předmětu *Bezpečnostní informatika* [90].

1.1 Strojové zpracování jazyka

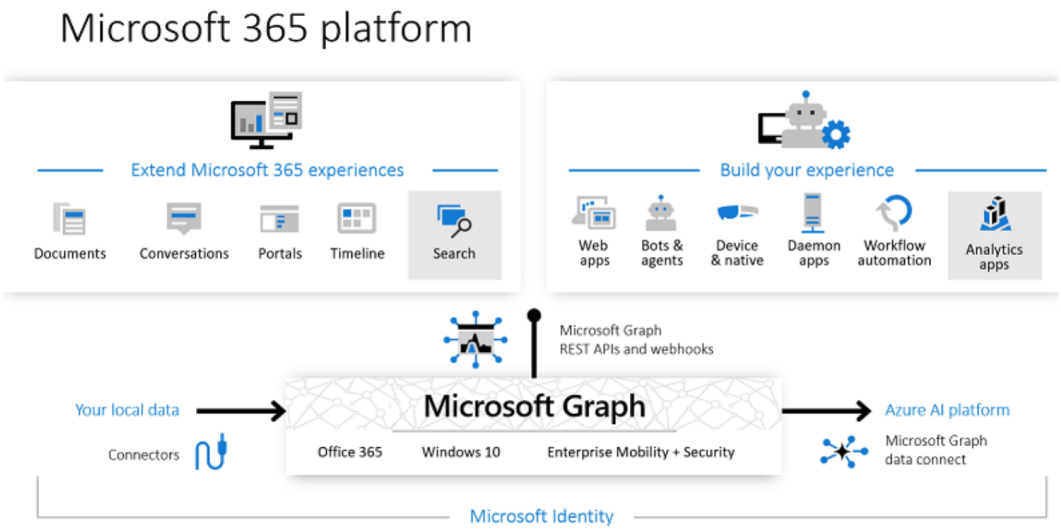
Výzkum umělé inteligence se v průběhu let postupně měnil a vyvíjel. Už v 60. letech minulého století byly realizovány první pokusy o strojové zpracování jazyka (viz např. ELIZA[1] [83]) a inteligentní prohledávání prostoru řešení (viz např. General Problem Solver [59]). První výpočetní model neuronu byl navržen již v roce 1957 (Rosenblatt [69]). Umělá inteligence (UI) jako nástroj se zdála blízko a úspěch se zdál neodvratný.

[1] Pro program ELISA existují simulátory, viz např. http://epanel.cz/eliza/eliza.php.

Jenomže se ukázalo, že tomu tak úplně není. Simulátor psychoanalytika ELISA fungující na principu zpracování *promluv* živého člověka, které automatizovaný systém zpracoval a reformuloval do podoby otázky ve snaze motivovat člověka k pokračování v konverzaci se ukázal být na dlouhou dobu maximem, kterého bylo možno dosáhnout, aniž by počítačový systém skutečně chápal, obsah zpracovávaných sdělení.

K pochopení textu je ale nutný podstatně vyšší výpočetní výkon než byl v dobách spuštění ELISy k dispozici. Takový systém navíc vyžaduje ke svému provozu vytvoření celých map různých konceptů a jejich vzájemných vazeb. Tyto komponenty začaly být dostupné teprve relativně nedávno (před pár lety), jeho dopady jsou ale již dnes citelné.

Jako příklad takového posunu můžeme použít např. službu *Miscrosoft Graph* [58]. Schéma zapojení služby do infrastruktury Microsoftu je dostupný na obr. 1.1.



Obrázek 1.1: MS Graph a jeho integrace s dalšími službami Microsoftu (převzato z [58])

Znalosti jsou tedy reprezentovány formou grafu, který pracuje v „*cloudu*“ a jsou přes standardizované rozhraní dostupné dalším aplikacím a službám společnosti Microsoft nebo popř. dalším subjektům za úplatu.

Obdobný přístup aplikují prakticky všichni velcí hráči např. pro provoz virtuálních asistentů jako je Siri (Apple), Alexa (Amazon), OK Google (Google) a řada dalších. Je potřeba také říci, že přes veškerou snahu zůstávají schopnosti těchto asistentů dosud velmi omezené a je otázkou zda jejich provoz svým tvůrcům přináší alespoň nějaký zisk.

Myšlenka strojového zpracování jazyka tak byla správná, její implementace, která je přímo nasaditelná v masivním měřítku v praxi si ale vyžádala několik revolucí ve výpočetní technice a desítky let času na vývoj. Musela vzniknout gigantická datová centra zpracovávající ohromné objemy dat a její využití pak vyžaduje připojení se k tomuto zdroji dat dálkově - tedy služby jsou primárně odebírány on-line.

Virtuální asistenti tedy pracují tak, že zachytávají promluvy ve svém okolí. (Všechny promluvy ve svém okolí.) Zachycení promluv se děje na koncovém zařízení uživatele, jako je mobilní telefon, tablet, chytrý reproduktor nebo jiné zařízení s podporou této technologie. Promluva je vyhodnocována na klíčové slovo indikující, že se jedná o pokyn k aktivitě asistenta (např. „OK Google“ nebo „Hej Siri“). V případě, že systém vyhodnotí, že promluva obsahuje požadavek na aktivitu asistenta zachycené promluvy zašle do datového centra svého provozovatele ke zpracování. Odesílání se děje po několikasekundových blocích. Informace o tom, co asistent má dělat pak po síti přijde zpět do zařízení, které pokyn zrealizuje.

Výše uvedené ale jasně demonstruje určité bezpečnostní problémy, které jsou s použitím takové technologie spojeny:

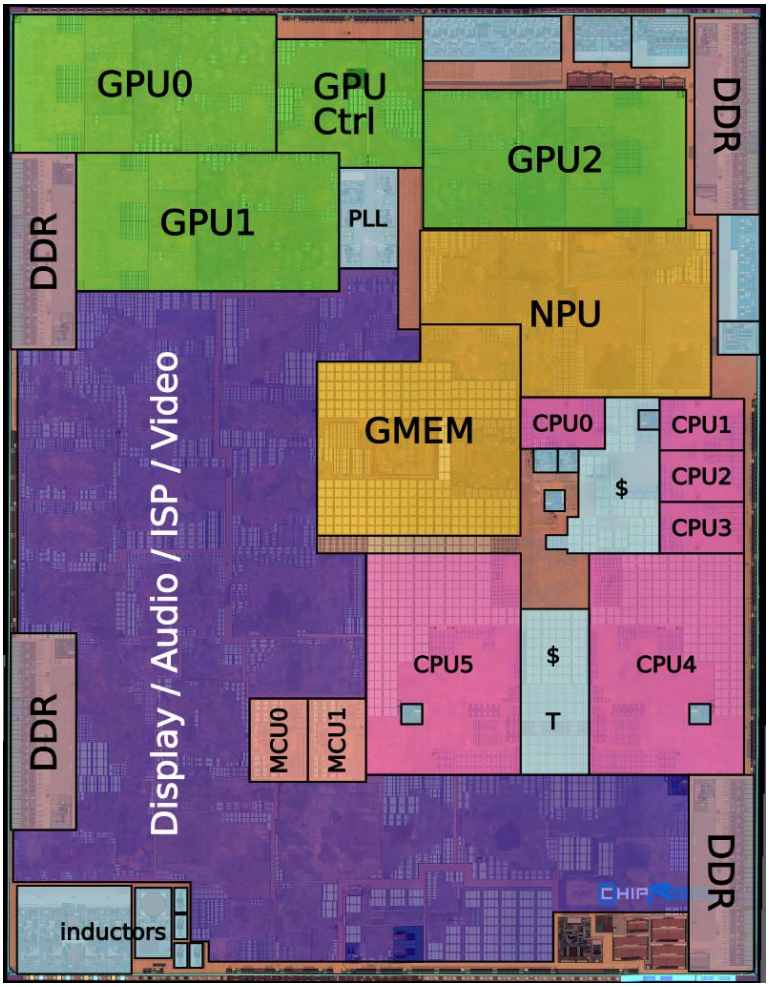
1. informace mohou být zachyceny cestou
2. promluvy mohou být (a také často jsou) ukládány pro další zpracování (tréning algoritmů)
3. uložené údaje mohou být zpracovávány také lidmi

4. ...

S tím jak se postupně začínají měnit používané technologie dochází k pozvolnému přesunu modelů, na kterých je výše uvedená funkcionalita založena na koncová zařízení. Komunikace s datovým centrem je sice z určitého pohledu pro provozovatele služby výhodná, protože získává jistou kontrolu nad zpracovávanými pokyny, na druhou stranu s provozem služby v datovém centru jsou spojeny také poměrně velké náklady, které nese provozovatel služby.

Je pak otázka, zda služba jako taková svému provozovateli přináší obchodovatelné nebo jinak využitelné informace, které tyto náklady pokryjí, popř. vyváží. Z praxe vyplývá, že schopnost provozovatelů generovat zisk právě z této služby je minimální a proto také dochází k postupnému přechodu těchto modelů z cloudu na koncová zařízení např. mobilní telefony.

Běžný procesor v počítači, mobilním telefonu nebo kterémkoliv jiném zařízení není optimalizován pro výpočty typické pro manipulaci s neuronovými sítěmi. Proto, aby bylo možné s tímto přechodem začít, musely se procesory změnit. Již v roce 2017 uvedl Apple jádra pro akceleraci výpočtů neuronových sítí v čipu A11 Bionic pro iPhone 8. Na obr. 1.2 je struktura tohoto chipu. Všimněte si Neural Processing Unit (NPU) bloku.



Obrázek 1.2: Analýza struktury chipu A11 Bionic (převzato z [72])

Tento blok, resp. jeho další iterace, Apple v současnosti používá na všech svých procesorech od mobilních telefonů až po notebooky a počítače a právě tato část čipu je odpovědná za akceleraci neuronových sítí.

Obdobný přístup volí také další výrobci čipů např. pro mobilní telefony s operačním systémem Android. Na veletrhu CES 2024 společnost AMD uvedla NPU jednotku v čipech určených do běžných počítačů (řada 8000G).

Na úrovni přenosných zařízení (např. mobilní telefony) v posledních letech proto sledujeme postupný přechod části služeb, které původně vyžadovaly spolupráci s cloudem na tato koncová zařízení. Podobný trend lze v budoucnu očekávat také u běžných počítačů.

Tento přechod bude obzvláště důležitý s očekávaným masivním nástupem LLM, které neúměrně jsou řádově větší než do té doby používané modely a tak zatěžují výrazně více datová centra. Zároveň pro tento typ modelů díky své jednoduché obsluze (ovládání přirozeným jazykem) a univerzálnosti nasazení pro každodenní práci (generování textu, překlady, reformulace textu, programování, apod.) lze předpokládat podstatně intenzivnější a častější nasazení, což problém výrazně zhoršuje.

LLM a obdobné modely jsou založeny na použití rozsáhlých neuronových sítích. Ty pracují na jiném principu než třeba původní jazykové modely, které se snažily modelovat strukturu jazyka, nebo třeba expertní systémy (viz dále), které se snaží popsat uvažování experta pomocí formálně specifikovaných pravidel práce s objektivně zjištěnými skutečnostmi - fakty.

Neuronové sítě oproti tomu šly jinou cestou - snaží se řešit problémy podobným způsobem, jakým problémy řeší mozek - tedy aktivací neuronů v neuronové síti v určitých vzorech, v závislosti na zachycených vstupech. Adaptovaná (naučená) neuronová síť si tak řešení vybaví podobně jako, se člověku vybaví znalost, když má odpovědět na nějakou otázku např. u zkoušky.

Neuronové sítě tak představují poměrně univerzální nástroj schopný řešit prakticky všechny typy problémů ... jenomže rozvoj neuronových sítí na poměrně dlouhou dobu přerušila zdrcující kritika nejpoužívanějšího modelu neuronu: *perceptron* Minského a Paperta [56] prokazující, že jeden Perceptron nelze použít k natrénování jednoduché matematické operace exklusive or (XOR) na jednom neuronu.

To bylo považováno za obrovský problém, protože podpora XOR je jednou z podmínek, kterou pro univerzální počítače (tzv. Turingovy stroje) formuloval matematik Alan Turing. Význam Turingova stroje je v tom, že se jedná o teoretický popis vlastností, které musí splňovat počítač, aby v něm bylo možné univerzálně řešit problémy. Neschopnost použít neuron pro XOR tak vlastně vyloučila neuron jako univerzálně použitelný blok pro řešení problémů ... a nastala první zima umělé inteligence (1974-1980), kdy výzkum v této oblasti téměř ustal.

Paradoxně z pohledu praktického užití neuronových sítí nemožnost natrénování XOR nepředstavuje problém - jedná se totiž o omezení vztaženo pouze na jeden neuron, nikoliv neuronové sítě - tedy spojení několika neuronů. V rámci více vrstevných sítí není problém operaci XOR adaptovat. Vlastně jsou k tomu potřeba pouze 2 neurony.

Dalším limitujícím faktorem praktického nasazení neuronových sítí byla relativní výpočetní náročnost na proces adaptace a pro opravdu velké sítě, také proces inference nebo vybavování. Výpočetní výkon potřebný pro řešení praktických problémů v době vydání knihy Perceptron nebyl k dispozici. Lze říci, že problém s výpočetním výkonem do jisté míry řešíme dosud, byť v současnosti je již k dispozici specializovaný hardware, který je pro tento typ úloh řádově efektivnější než běžně používané procesory.

K modernímu pojetí neuronových sítí se ale musíme historií propracovat.

Výzkum uměle inteligence totiž v průběhu první AI zimy neustal, pouze se zaměřil jiným směrem.

1.2 Expertní systémy

Další možností, jak vyřešit problém umělé inteligence poznat a napodobit způsob, kterým uvažuje člověk tak, že se formálně popíše způsob, jakým člověk uvažuje. Vzhledem k tomu, že lidské myšlení a konání je velmi komplexní, omezil se výzkum pouze na řešení určitých konkrétních problémů a napodobení způsobu myšlení vybraných expertů ve zvolené problémové oblasti. Odtud také pochází název tohoto druhu umělé inteligence - *expertní systémy*.

S touto problematikou je přímo spojen také obor *znalostní inženýrství*, které se zaměřuje na způsob jakým znalostní inženýr analyzuje problém a způsob jakým jej expert řeší. Tyto informace pak znalostní inženýr transformuje do podoby znalostní báze obsahující fakta a pravidla, jak s nimi manipulovat.

Zpočátku se tento směr zdál být velmi nadějným. Ukázalo se totiž že vytvořit pro jednoduché problémy je možné také jednoduše vytvořit znalostní bázi, na základě které bude expertní systém schopen konzistentně a efektivně dosahovat výsledků srovnatelných s expertem v dané problémové doméně. To vědce naplnilo optimismem, stran toho, že tyto systémy budou dobře škálovatelné také pro složitější problémy.

Tento optimismus spustil tzv. druhý boom UI, který se datuje někdy mezi léta 1980 – 1987.

Dobrým příkladem expertního systému je systém CLIPS [1] vyvinutý původně v NASA. Objevila se ale také celá řada jiných systému na komerční i nekomerční bázi.

S postupem času se ale ukázalo, že předpoklad jednoduché škálovatelnosti expertních systémů není platný. Čím je tedy problém složitější tím je mnohem, mnohem složitější zformulovat takovou znalostní bázi, která by expertnímu systému umožnila fungovat tak, jak bychom to od něj očekávali. Nejen, že pravidel musí být zpracováno mnohem více, samotní experti často neví jak přesně ke svým závěrům docházejí.

Expert se totiž kromě znalostí která jsou vyjádřitelná pomocí vzorců a formálně zapsatelných pravidel řídí také svými „měkkými“ dovednostmi, jako je empirická zkušenost s řešením obdobných problémů, intuice a podobně, které nejsou jednoduše zachytitelné ve znalostní bázi.

Dalším problémem expertních systémů je to, že jsou z pohledu schopnosti poskytnout řešení „křehké“. K rozbití systému pak může dojít velmi snadno tak, že jej vystavíme byť jen trochu odlišné situaci, než se kterou bylo počítáno při tvorbě báze znalostí. Expertní systém v takovém případě není schopen poskytnou potřebné odpovědi buďto vůbec nebo dokonce může poskytnout odpověď chybnou.

Vytvoření komplexní báze dat je tedy pro expertní systém naprostou nutností a to se pro většinu aplikací prostě nevyplatí.

Výše uvedené problémy vedly k tomu, že z hlediska výzkumu došlo k vystřízlivění a v důsledku toho také ke druhé AI zimě v letech 1987 - 1993. Vývoj v oblasti umělé inteligence ale neustává úplně, svůj comeback zažívají opět neuronové sítě, konkrétně vícevrstevné. Takové sítě netrpí podobnými omezeními jako sítě jednovrstevné, které stály na počátku výzkumu neuronových sítí. Obnovení výzkumu bylo také podpořeno výrazným nárůstem dostupných výpočetních kapacit, které jsou pro nasazování neuronových sítí vyžadovány.

Paradoxně nelze říci, že expertní systémy byly zcela neúspěšné. Vedly ke vzniku znalostního inženýrství. Posunuly výrazně dopředu schopnosti analytiky problémů. To je třeba oblast, které se v rámci tohoto předmětu a těchto budeme věnovat v rámci kapitoly věnované jazyku UML.

Jedná se o univerzální grafický modelovací jazyk, který nám umožňuje analyzovat datové struktury, časový průběh, popisovat propojení a komunikaci komponent systémů apod.

Pro někoho možná překvapivě vzhledem k popsaným omezením, i samotné expertní systémy jsou poměrně masivně prakticky nasazovány. Akorát jsme museli ustoupit od toho že by tento typ přístupu mohl vést ke skutečné umělé inteligenci nebo že by byl schopen pro složitější problémy nahradit experta.

Na druhou stranu tento typ systémů vykázal vysokou schopnost řešit jednoduché problémy a právě tato schopnost je z hlediska nasazení v jedné specifické oblasti vysoce přínosná. Konkrétně nám umožňuje ve složitých informačních systémech automatizovat částečně workflow dokumentů a údajů, které jsou v těchto systémech zpracovávány.

Takové systémy ale už neoznačujeme jako znalostní, ale spíše jako systémy pro podporu rozhodování (Decision Suport System (DSS)). Takové systémy jsou v současnosti již využívány rutinně.

1.3 Multiagentní systémy

Od roku 1993 až do dneška probíhá již nepřerušovaný intenzivní rozvoj metod UI, byť i tady lze vysledovat jisté etapy charakteristické na soustředění se na určité skupiny technik nebo nástrojů. Za přelomový z tohoto pohledu by mohl být považován rok 2010. V prvním období (před rokem 2010) se výzkum primárně zaměřil na tzv. *multiagentní systémy* (viz např. SWARM [6]).

Multiagentiní systémy reagují na problém expertních systémů, které byly koncipovány jako monolitické, složité systémy, které řeší problém s použitím ucelené, konsolidované báze znalostí. Už víme, že vytvoření a údržba takové znalostní báze je extrémně obtížná. Lze ale takové omezení obejít? Multiagentní systém problém řeší jeho rozložením do většího množství menších, zvládnutelných subproblémů. Jejich řešení pak svěřuje specializovaným agentům. Řešení problému jako celku je pak dosaženo vzájemnou spoluprací jednotlivých specializovaných agentů.

Komplexity je tedy dosahováno interakcí jednoduchých, „snadno“ řešitelných komponent. Interakce jednotlivých agentů je realizována pomocí předpřipravených komunikačních protokolů. Vzhledem k relativní složitosti realizace skutečně funkčního multiagentního systému budeme se v rámci výkladu této problematiky věnovat pouze základnímu úvodu s demonstracemi jednoduchých 2D multiagentních systémů a také *emergence efektu*.

Tento efekt se objevuje jak v naturogenních, tak antropogenních systémech pouze na v důsledku opakované vzájemné interakce drobných, samostatně nevýznamných efektů. Tyto efekty se objevují - odtud také název z angličtiny emerge, tedy objevit se. V praxi se s tímto efektem setkáváme třeba v systémech kritické infrastruktury.

Z pohledu praktické nasaditelnosti si multiagentní systémy, alespoň tedy jejich softwarová varianta nalezla pevné místo pro simulace celé řady problémů. V oblasti bezpečnosti se například tento typ simulací extenzivně využívá pro simulaci evakuace. Evakuace objektů je z tohoto pohledu považována za vyřešený problém a existují pro ni komerční nástroje, které tento způsob výpočtů využívají. Pro evakuaci území existují některé experimentální modely, s tím že výzkum stále ještě pokračuje.

Výzkum v této oblasti je komplikován faktem, že v případě evakuace území pracujeme s mnohem větším prostorem, který je potřeba simulovat, je potřeba také pracovat s mnohem vetším množstvím typů agentů, kteří se zásadně liší svými vlastnostmi. Simulovat je totiž potřeba jednotlivé chodce ale také různé typy vozidel, které pro evakuaci mohou být využívány. Rozložení populace je také velmi odlišné v čase a konečně je často nutné simulovat také postup fyzického děje, který evakuaci vyvolal (např. povodeň nebo lesní požár).

V obecné rovině se také využívá přístup pro řešení problémů preferovaný multiagentními systémy. Tedy, že problém je rozdělen na několik menších částí, které jsou řešeny samostatně a řešení komplexního problému je dosahováno řetězením takových modelů za sebou.

Technicky pak nezáleží na tom, na jakém principu nebo která metoda, byla zvolena pro řešení problému.

Do jisté míry si řešení lze pak představit jako blokový digram, kde vidíme pouze účel použití jednotlivých bloku, ale nikoliv jejich vnitřní způsob fungování a řešíme spíše způsob, jakým budou společně komunikovat. V IT systémech takové komunikační rozhraní obvykle označujeme jako Application Interface (API)

1.4 Big data a průmysl 4.0

Zpět k historii - ukazuje se totiž, že přístupy a technologie, které v minulosti byly vyvinuty a do určité míry nasazeny, ale ukázalo se že nemají očekávaný potenciál k řešení problémů nebo tato řešení trpí nějakým zásadním omezením, mají tendenci se v průběhu času vracet. Dobrým příkladem tohoto trendu jsou třeba neuronové sítě, které byly opuštěny počátkem 90. let minulého století, aby po 10-ti letech zažily svůj velký comeback.

Neuronové sítě se tak postupně začaly prosazovat v nástrojích pro datamining i ve specializovaných aplikacích pro zpracování textu, obrazu, hlasu a celé řadě dalších aplikací.

Tento návrat je poháněn ohromným rozvojem výpočetní techniky a s ním souvisejícím nárůstem výpočetního výkonu, který je dostupný pro adaptaci a inferenci (konzultaci) neuronových sítí.

Tento rozvoj umožnil nástup vícejádrových procesorů umožňujících paralelní zpracování dat a také nástup specializovaného hardware, který lze úspěšně nasadit pro řešení problémů UI včetně adaptace neuronových sítí.

Takovým specializovaným hardware jsou třeba moderní grafické karty. Oproti klasickým procesorům grafické karty obsahují velké množství malých a relativně jednoduchých jader umožňujících paralelní zpracování jednotlivých částí obrazu. Ukázalo se, že tato vlastnost činí grafické karty obzvláště vhodné pro manipulaci i s rozsáhlými neuronovými sítěmi.

V současnosti se také začínají prosazovat hardwarová řešení čistě pro řešení problémů neuronových sítí.

Lze přitom říci, že možnosti (schopnosti) dosažitelné nasazením neuronových sítí roste se složitostí používaných sítí exponenciálně. Nárůst dostupného výpočetního výkonu společně s vývojem nástrojů, které jej dokáží využít je proto klíčovým faktorem úspěchu nasazení neuronových sítí.

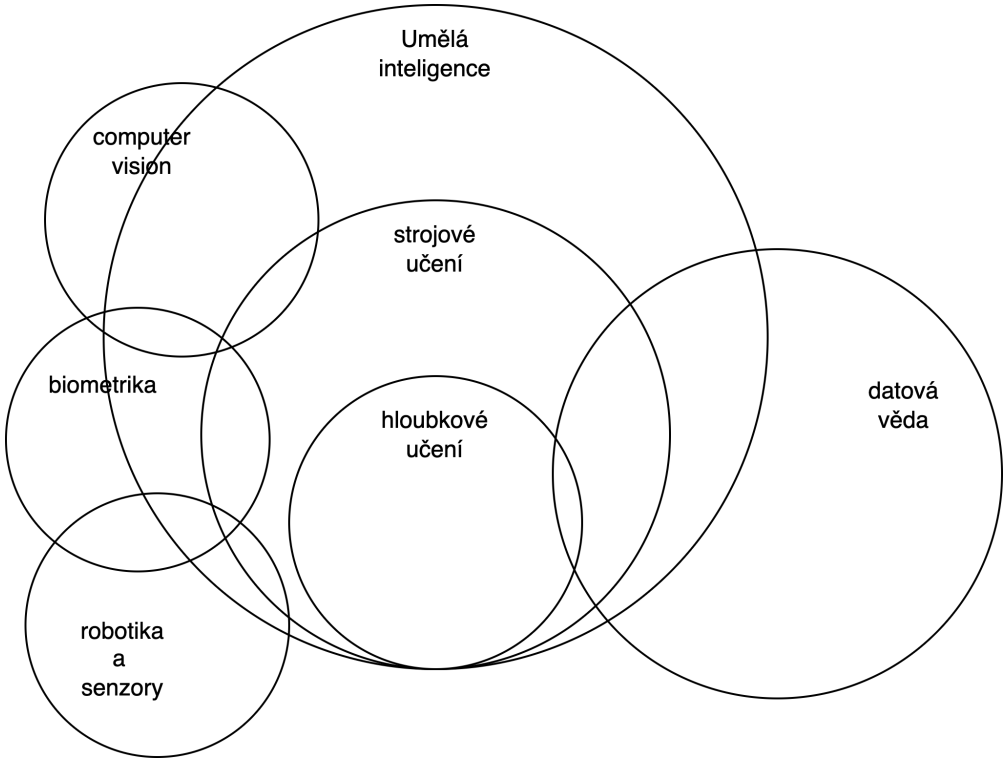
Dostáváme se tak téměř do současnosti. Hovoříme tedy o období někdy po roce 2010. Pro *velké* problémy je typické, že pro jejich charakterizaci a následné řešení je potřeba zpracovávat ohromné objemy dat. Jenomže velké objemy dat s sebou nesou také velkou náročnost jejich zpracování. Ukázalo se, že do té doby používané přístupy nejsou úplně efektivní pro zpracování takových objemů dat a vyžadují tedy specifický přístup k řešení.

Z tohoto důvodu označujeme takové objemy dat jako tzv *big data* a metody, kterými je zpracováváme pak označujeme často jako *metody strojového učení.*

„Velkými daty“ se obvykle rozumí objemy dat, které není možné zpracovat pomocí běžně dostupných databázových nástrojů na bázi relačních databází, s jakými jste se mohli v rámci studia setkat

např. v předmětu *Bezpečnostní informatika*. Přesná hranice, kdy data považujeme za velká přitom není nikde stanovena. Orientačně lze ale jako hranici stanovit datové objemy řádově ve stovkách gigabajtů až terabajtů, které pro řešení problému je potřeba zpracovat.

Vlastně pracujeme s velkým množstvím vzájemně propojených pojmů. Zkuste se podívat na obr. 1.3 pro některé z nich.



Obrázek 1.3: Podoblasti umělé inteligence a jejich vzájemný vztah

Umělá inteligence (Artificia Inteligence (AI)) je tedy velký obor, pod který se nachází nebo se kterým spolupracuje celá řada dalších oborů nebo skupin metod a technik. Obr. 1.3 také nelze považovat za vyčerpávající výčet suboborů umělé inteligence. Všimněte si např., že na obr. nenaleznete oblasti expertních systémů a multiagentní systémy, ačkoliv do této oblasti zcela jistě patří.

Účelem obr. je spíše naznačit složitost a propojenost celé problematiky než vyčerpávajícím způsobem „zakastlíkovat“ vše. Také vazby mězi jednotlivými podobory nebo problémovými oblastmi mohou být komplikovanější.

Z obr. lze říci, že metody *strojového učení* (Machine Learning (ML)) jsou považovány zcela za podobor umělé inteligence. Trošičku problematické je z tohoto pohledu postavení klasické statistiky (např. regresních modelů), které s problematikou strojového učení velmi úzce souvisí, ale zároveň není považováno za součást umělé inteligence nebo strojového učení.

My se v tomto předmětu statistice věnovat nebudeme, v průběhu Vašeho studia se ale později se statistikou podrobněji setkáte v předmětu *Statistika*. Věnujte při studiu pozornost zejména problematice regresních modelů a testování hypotéz, které je typické také pro metody strojového učení.

V tomto předmětu se z metod strojového učení budeme věnovat pouze metodám souvisejících s nasazením neuronových sítí. Pokud ale budete pokračovat v magisterském navazujícím studijním programu budete mít možnost studovat předmět Informatika v bezpečnosti, který Vám poskytne mnohem širší portfolio metod v této oblasti.

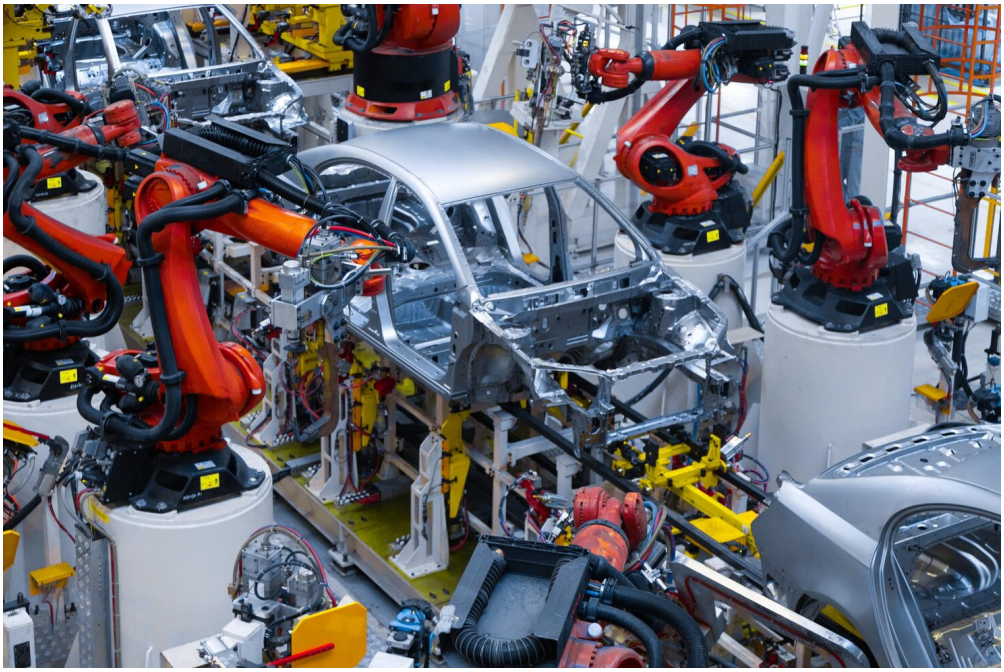
Hloubkové učení (deep learning) je založeno na použití mnohavrstevných neuronových sítí. Je proto očividně podmnožinou strojového učení.

Datová věda oproti tomu je obecnější a svým zaměřením přesahuje rámec, jak umělé inteligence, tak jejích podoborů. Zabývá se např. problematikou databází, je zde přesah na statistické metody apod.

V levé části obrázku jsou pak různé podobory/problémové oblasti, které mají úzkou vazbu na umělou inteligenci, ale jsou také vzájemně provázány. Zajímavá je např. oblast robotiky. Tato oblast

byla původně zcela samostatná a s oblastí umělé inteligence zcela nesouvisející.

To pro někoho může být překvapivé, protože slovo robot v literatuře má jednoznačně konotaci na umělou inteligenci. V praxi tomu tak ale nebylo. Robotika byla využívána primárně pro automatizaci převážně výrobních procesů v rámci výrobních linek. Typickým představitelem takového robota je robotické rameno jako na obr. 1.4.



Obrázek 1.4: Automatizovaná výrobní linka pro výrobu automobilů (převzato z [19])

Toto rameno původně bylo ale přesně naprogramováno tak aby vykonávalo přesně stanovenou sérii pohybů a interakce s objekty na výrobní lince. Výsledný robot tak nebyl ani inteligentní a už vůbec nebyl univerzální.

To se ale v posledních letech začíná měnit. Moderní robotická ramena jsou vybavována kamerovými systémy a jsou trénována ve virtuálním prostředí (k tomuto se ještě později v této kapitole vrátíme) tak, aby bylo samostatně schopno jednak vykonávat požadovanou činnost a také reagovat na své okolí.

Nová, umělou inteligencí vybavená, robotická ramena tak nemají naprogramovány své pohyby, ale odvozují je ze stavu sebe sama (robotického ramene) a okolí, jak je vnímáno kamerovým systémem. Robot je v takovém případě mnohem více univerzální a je schopen reagovat plynule na změny v prostředí, ve kterém funguje. Automatizovat tak lze mnohem více činností ve výrobním procesu než bylo dosud možné.

Jak je patrno z výše uvedeného. Souvisí schopnost inteligentního chování na náročné adaptaci a také schopnostech *počítačového vidění* a schopnosti získané informace interpretovat ke splnění úkolů robotovi zadaným. V praxi jsou již takové systémy celkem běžně nasazovány, byť v současnosti nemají schopnost být skutečně univerzálními a nahradit tak plně práci běžného pracovníka.

Z praktického pohledu se ale jedná o problém s iterativním řešením. To znamená, že s každou iterací modernizace výrobního procesu bude docházek k postupnému nahrazování pracovníků stroji vybavenými umělou inteligencí.

Může dojít k úplnému nahrazení práce člověka strojem? V současnosti je odpověď ne, zároveň ale víme, že do jisté míry k takovému nahrazování již dochází, není ale intenzivní a nelze nahradit všechny typy práce. Podíl automatizovatelných činností se ale v čase zvyšuje. Zatímco v současnosti jsou změny v zaměstnanosti plynule absorbovány jinými sektory ekonomiky (převážně v oblasti služeb) v budoucnu tomu tak být nemusí.

Vlastně se jedná o jedno poměrně palčivých témat, o kterém se vedou doslova nekonečné debaty na všech možných platformách. Jednou z pesimističtějších knih, které interpretují v současnosti pozorované trendy a extrapolují je do budoucna je kniha Jeremy Rifkina End of Work [68].

Existují také některé poměrně pesimistické předpovědi OECD a celá řada dalších knih a studií. Naštěstí se ukazuje, že tyto nejpesimističtější scénáře se nenaplňují. Proto, ačkoliv je změna, kterou

tyto procesy přinášejí již pozorovatelná není tak rychlá, jak se původně předpokládalo.

To ale nutně bohužel neznamená, že nástup takových systémů do budoucna zásadní změny na trhu práce skutečně nemá potenciál vyvolat. Nakonec to je také jedním z důvodů, proč se vůbec touto problematikou zabýváte v průběhu studia. Budoucnost tedy sice předpovídat stále ještě neumíme, zároveň ale jsme schopni vidět některé trendy. Což by nám mohlo umožnit se jim do jisté míry přizpůsobit nebo jich dokonce využít ve svůj prospěch.

Úspěch tohoto přechodu ale závisí na schopností společností efektivně shromažďovat, zpracovávat a rutinně využívat opravdu velké objemy dat. Už víme, že zpracování tradičními relačními databázemi pro taková data nepředstavuje schůdnou cestu. Ke zpracování takových objemů dat jsou pak vyžadovány specializované databáze označované souhrnně jako No SQL databáze, jako jsou např. Redis [71], MongoDB [12] nebo Cassandra [11]

Hloubkovým učením se rozumí použití gigantickým (i miliardy neuronů), mnohavrstevných neuronových sítí, které jsou schopny se adaptovat i na objemy dat poskytovanými big daty. Adaptace takto rozsáhlých neuronových sít ale vyžaduje použití specializovaného hardware, jako je např. Tensor Procesor Unit (TPU) v5 Googlu, akcelerátory NVidia H100 nebo AMD MI300X.

Zpracování velkých dat bylo v minulosti doménou velkých datových center a specializovaných nástrojů, jejichž nasazení vyžadovalo specialisty na datovou vědu (data science). To vše měly tradičně k dispozici zejména velké, často nadnárodní firmy, popř. výzkumné instituce popř. úřady.

S rozvojem výpočetní techniky se ale objevila řada nástrojů, které jsou cenově dostupné jako možnost si např. pronajmout dočasně výkon „v cloudu“, čímž odpadají náklady nutné pro pořízení vlastního datového centra. Z pohledu software jsou dnes dostupné jak komerční analytické nástroje za úplatu tak open source nástroje zadarmo. Dnes tak nic nebrání tomu, aby výhodu „velkých dat“ využívaly také střední nebo dokonce malé podniky a řada z nich tak skutečně činí.

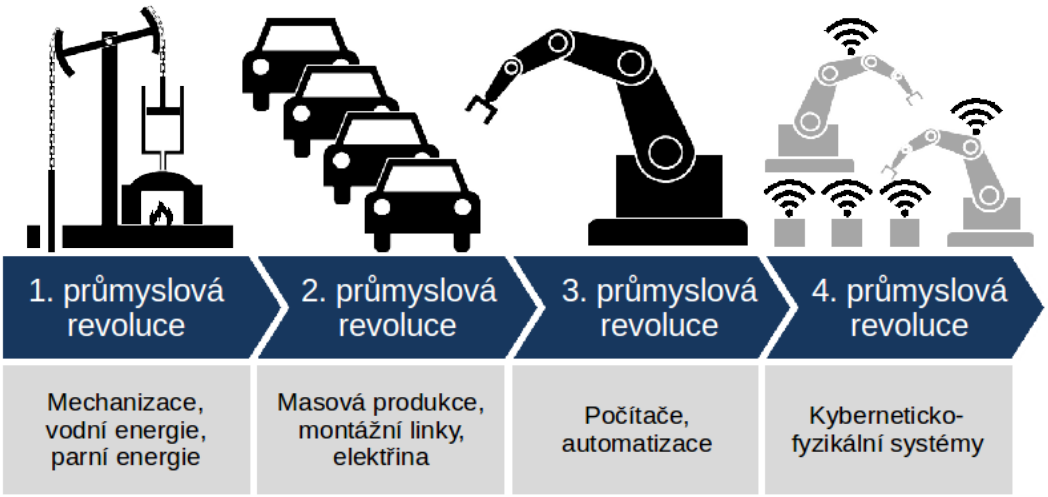
Prováděné analýzy jim pak přináší lepší vhled do fungování obchodních společností. To umožňuje transformovat a často také automatizovat jednotlivé procesy ve společnostech a dosahovat tak podstatně vyšší efektivity.

Průmysl 4.0

Velmi často jsou pak změny ve společnostech, zejména výrobních, masivnější. Myšlenka automatizace není nová - konečně průmyslová revoluce v 18. a 19. století umožnila rozložit výrobní proces na sled na sebe navazujících kroků. Tyto kroky pak byly realizovány pomocí specializovaných dělníků a jejich strojů - a tak vznikly první manufaktury a byla nastolena cesta k realizaci velkovýroby.

Stroje se postupně modernizovaly a umožnily tak relativně malém prostoru realizovat i složitější operace. Produktivita práce tak stoupala. S rozvojem výpočetní techniky pak přišla možnost další úspory lidské práce - některé úkoly v oblasti regulace převzaly systémy automatizovaného řízení.

Technologický pokrok je dobře viditelný na obr. 1.5.



Obrázek 1.5: Transformace průmyslové výroby v čase (převzato z [70])

Programmable Logic Controler (PLC) automaty tak na základě vstupů zachycených připojenými senzory a programu instalovaného v PLC rozhodnou samy o vhodném způsobu regulace řízeného stroje

bez účasti člověka. PLC automaty sice člověka nedokázaly plně nahradit, neboť stále nelze vyloučit, že regulovaná soustava se dostane do stavu, se kterým nebylo počítáno - automat na něj nebude schopen reagovat a v takovém případě je zásah lidského operátora nezbytný.

Automatizace tak znovu zvyšuje efektivitu celého procesu a člověku ponechává ty aktivity v rámci výrobního procesu, jejichž řešení není algoritmicky efektivní. Tolik tradiční automatizace - v současnosti totiž probíhá poměrně rychlý přechod na další stupeň automatizace, která už není omezená na postupy naprogramované do jednotlivých strojů, ale jsou vybaveny schopností adaptovat se do výrobního procesu a pružně reagovat na změny, které v něm probíhají.

Tuto změnu umožnil až nástup využití umělé inteligence. Obdobné změny probíhají také v nevýrobních organizacích, možná dokonce rychleji než v podnicích výrobních, protože toky informací lze softwarově řešit jednodušeji.

Tuto transformaci podniků někdy označujeme jako iniciativu *Průmysl 4.0*.

Definovat přesně, co Průmysl 4.0 je, není úplně snadné, protože primárně se jedná o marketingovou značku, zkratku, chcete-li, pro označení dlouhodobých, ve své podstatě však velmi odlišných, změn, které probíhají ve společnostech. Do této iniciativy obvykle zařazujeme následující směry transformace:

- Internet of Things (IoT) - Internet věcí
- mobilní zařízení (telefony, tablety, ...)
- 3D tisk
- chytré senzory
- big data
- cloud computing
- automatizace technologií a procesů
- a další.

Přestože výše uvedené směry jsou relativně různorodé, řada států se snaží nad procesem transformace získat kontrolu a dosáhnout tak pozitivních synergických efektů na ekonomiku státu a fungování společnosti jako takové.

ČR např. přijala *Strategii pro průmysl 4.0* [57]. Jejím gestorem je Ministerstvo průmyslu a obchodu.

V tomto předmětu se zaměříme primárně na problematiku *neuronových sítí* a možnosti jejich aplikace pro řešení různých problémů spojených s bezpečností.

1.5 Současný stav řešení umělé inteligence

Abychom lépe pochopili současný stav problematiky umělé inteligence podíváme se na tzv. hype graf AI, viz obr. 1.6.

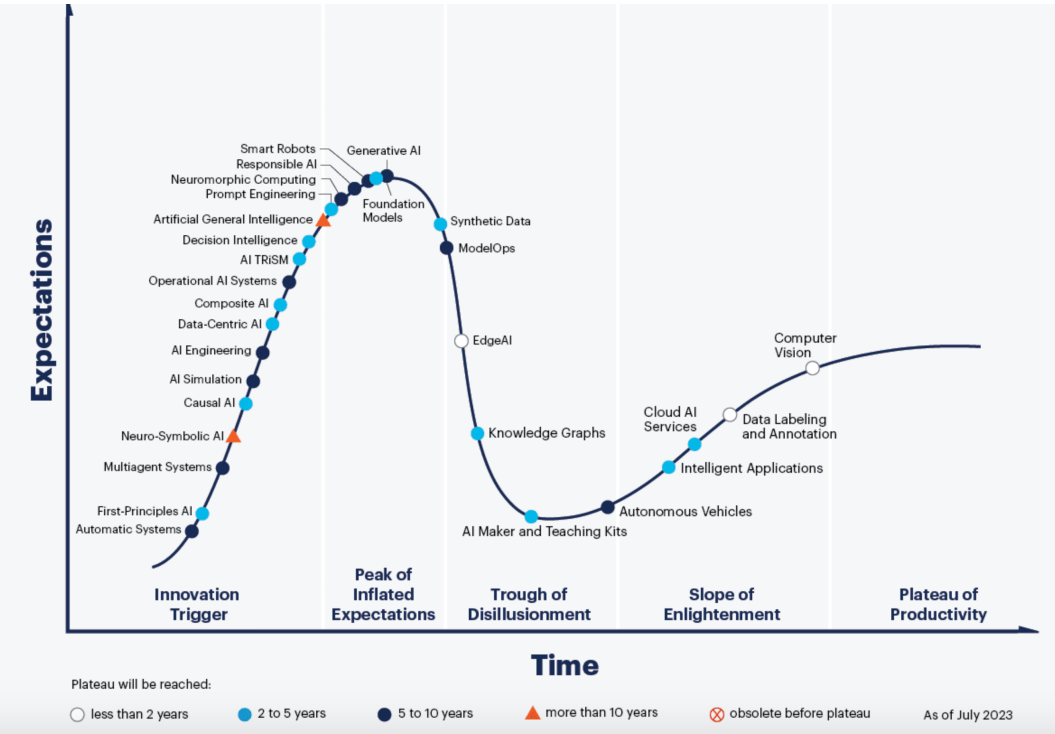
Společnost Gardner provádí řadu průzkumů a analýz pro technologie v AI, ale třeba také technologie obecně.

Hype cyklus má vždy stejnou strukturu:

- *uvedení inovace* (innovation tregger) - očekávání stran uvedené inovace rychle a razantně stoupají
- *vrchol očekávání* (peak of inflated expectations) - vlivem medializace jsou ale tato očekávání obvykle velmi nadsazená a nerealistická, proto v určitém momentu dosahují tato očekávání vrcholu aby následovala
- *deziluze/vystřízlivění* (through if disillusionment) kdy dochází k prudkému propadu očekávání, někdy až hluboko pod reálnou užitnost inovace
- *sklon narovnání* [očekávání] (slope of enlingment) - očekávání postupně stoupají s tím, jak se inovace iteračně zlepšuje a přibližuje k praktické nasaditelnosti.
- *rovina produktivity* (plateu of productivity) - inovace je již „dospělá“ a rutinně je nasazována v praxi

Z praktického pohledu je hype křivka na obr. 1.6 stylizovaná. Tedy každá inovace obdobnou křivkou musí projít, ale zároveň bude tato křivka pro každou inovaci odlišná a to v délce jednotlivých částí, ale také např. velikosti očekávání, která jsou s ní spojena.

Část roviny produktivity obvykle zůstává prázdná, jelikož rutinně nasazované inovace nejsou z pohledu studie zajímavé.



Obrázek 1.6: Hype cyklus pro rok 2023 podle průzkumu společnosti Gardner (převzato z [21])

Příkladem takových inovací je třeba Natural Language Processing (NLP). V roce 2022 byla tato technologie ještě ve studii Gardner [84] ve fázi deziluze, ale v důsledku průlomu v oblasti generativní AI je dnes už problematika NLP považována za vyřešenou. Podobný skok udělala celá oblast hloubkového učení.

Podobnou cestu urazily třeba také technologie zajišťující převod textu na mluvu nebo mluvy na text.

Všimněte si také, některých klíčových technologií a jejich pozici na křivce:

- generativní AI - na vrcholu očekávání
- Artificia General Inteligence (AGI) - blíží se k vrcholu hype křivky syntetická data - ve fázi vystřízlivění

Generativní umělá inteligence je něco co zažilo ohromný boom v roce 2023 a pro následující léta se očekává, že tento boom bude dále pokračovat. AI jako jsou ChatGPT, Google Bard a další jsou dostupné široké veřejnosti, a jsou schopny v řadě poskytovat již užitečné služby svým uživatelům. Zároveň ale v řadě ohledů nejsou lepší než člověk, nechápou text, který je jim předkládán nebo jej zpracovávají, ani ten který generují, trpí halucinacemi. S použitím jsou spojeny některé etické otázky a již dnes existuje celá řada útoků na ně s potenciálem posunout výsledek práce AI určitým nežádoucím směrem.

Žádný s výše uvedených problémů ve skutečnosti není jednoduše řešitelný a etický rozměr dokonce ani nutně nesouvisí s AI jako takovou, ale spíše s námi (lidmi) a způsoby, kterými sestavujeme obsáhlé datové sady používané pro tréning AI. Ukazuje se totiž, že způsob jakým naše, nebo jakákoliv jiná, společnost funguje ve výsledkem dlouhodobých procesů, stavu poznání, zvyků a kultury formovaných v průběhu stovek let. Tyto vlivy jsou hluboce zakořeněny ve společnosti a propisují se do všeho, co dělá bez ohledu na to, jestli při objektivním zkoumání považujeme výsledky těchto vlivů za žádoucí nebo dokonce legální.

Datasety jsou pak pouze obrazem této ne úplně žádoucí reality. Pokud AI byla na takových datech trénována bude udržovat přirozenou formou tento nežádoucí standard a to je problém. Někdy totiž máme tendenci chápat umělou inteligenci jako objektivní, nepodjatou. Jedná se přece jenom o algoritmus, který ze své podstaty nemůže být třeba rasistický. Ukazuje se ale, že pokud tyto vlastnosti datasetů neodhalíme předem, může umělá inteligence aplikovat třeba rasistické předsudky nebo jakoukoliv jinou formu biasu.

Pro odstranění tohoto problému proto nestačí data pořízená z fungování společnosti, tedy data generovaná člověkem. Řešením je použití tzv. *syntetických dat*, které jsou uměle vygenerované tak, aby poskytovaly vyvážený pohled na problém, který AI má řešit.

Bohužel z praktického pohledu není způsob tvorby syntetických dat dořešen. Proto se nachází v současnosti tato inovace v sestupné trajektorii (vystřízlivění) hype cyklu.

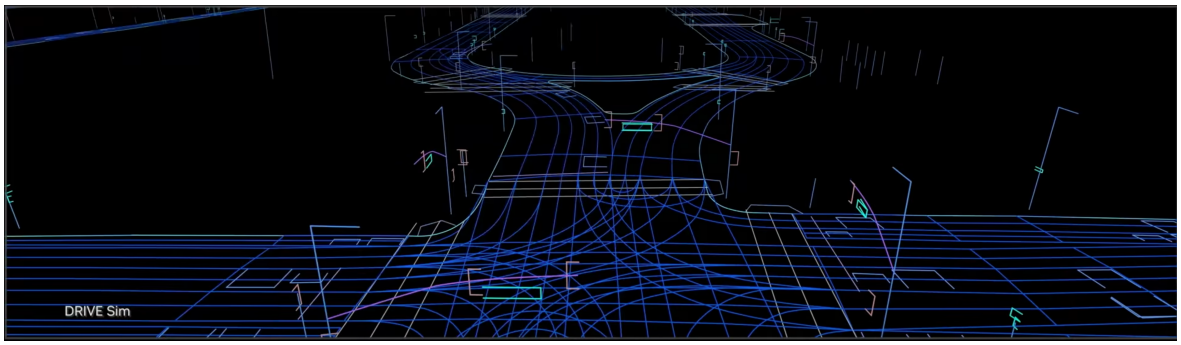
Problém je v tom, že neexistuje univerzální způsob, jak syntetická data připravit. Uveďme si dva příklady, jak by takové generování mohlo fungovat. V prvním příkladu se podíváme na samo-řiditelná auta, v druhém pak na hypotetickou AI vyhodnocující vhodnost žadatele o určitou práci.

K problematice konstrukce datasetů pro tréning řídících jednotek automobilů byl původně realizován záznamem běžného silničního provozu. Fungovalo to tak, že se vytvořila flotila vozidel vybavených senzory (kamery, lidary, apod.) které z běžného provozu pořizovaly záznamy z těchto senzorů. Problémem je, že takto získané údaje je nutno zpracovat - označit je popisky, aby se identifikovaly důležité momenty a údaje v datech, na které ma auto reagovat. Což není vůbec snadné.

Tradiční způsob zpracování totiž vyžaduje, aby toto opopiskování (labeling) provedl člověk, což ale vzhledem k rozsahu získaných dat není reálně možné - jelikož by to trvalo příliš dlouho a stálo to příliš mnoho peněz.

Proto se v současnosti používá spíše kombinace reálných dat, kde se o proces popisování stará umělá inteligence a syntetická data. Kde jsou jednotlivé situace v silničním provozu simulovány ve virtuálním prostředí a rovnou také popisovány. Jistou představu si o způsobu si můžete udělat z obr. 1.7-1.11.

Obrázek 1.7: Point cloud vytvořený analýzou zachyceného obrazu (převzato z [60])

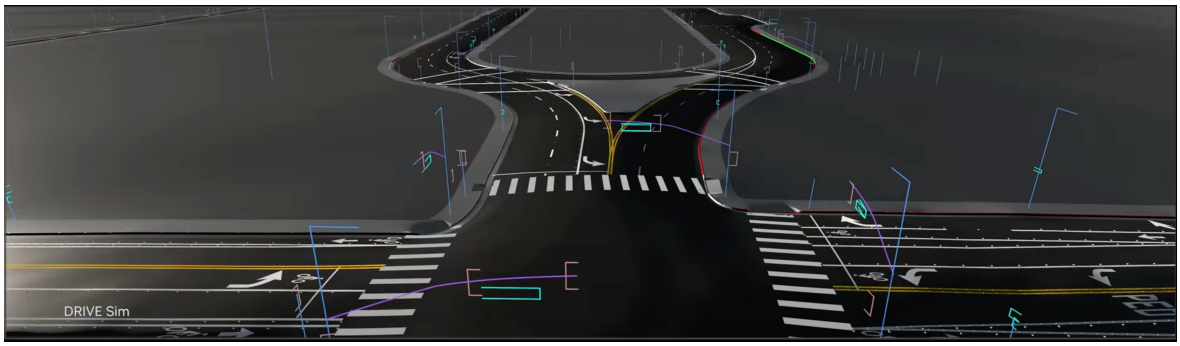


Obrázek 1.8: Identifikace silnic v obrazu (převzato z [60])

Jak je patrno z obr., použití syntetických dat není snadné, protože je vyžadováno realistické 3D prostředí, ve kterém se jednotlivé objekty budou pohybovat. Tento přístup je ale velmi flexibilní, protože lze simulovat naprosto cokoliv. Auta všech možných typů a barev a také mezní situace v dopravě, se kterými se v rámci běžného silničního provozu nesetkáváme příliš často.

Nevýhodou tohoto přístupu je, že jím vzniká gigantické množství dat, se kterými se musí budoucí AI vypořádat a to je také důvod, proč v současnosti na trhu dosud nejsou automobily, které by funkcí plnohodnotného automatického řízení disponovaly.

Všimněte si také, že pro účely adaptace modelu simulace jízdy se využívá digitální dvojče města i automobilu. Jedná se o koncept, který má mnohem větší uplatnění než jen uvedený příklad. Extenzivně



Obrázek 1.9: Model silnice (převzato z [60])

Obrázek 1.10: Simulované prostředí křižovatky (převzato z [60])

se např. používá pro průmysl 4.0, kde pro realizaci chytrého řízení výrobního procesu potřebujeme mít plnou kontrolu nad stavem technologie. Vytvoření digitálního dvojčete je přirozeným řešením tohoto problému.

Pro případ druhý budeme postupovat spíše formou úvah, než ilustračních obrázků. AI v tomto případě má za úkol vybrat nejvhodnějšího adepta pro pracovní pozici z řad žadatelů o práci. Známými slabinami procesu tohoto výběru je, že pro určité pozice máme tendenci preferovat určitý gender, ačkoliv proto neexistuje reálné opodstatnění.

Syntetická data je v takovém případě potřeba generovat tak aby v dostupných datech simulovala chybějící osoby, které splňují požadavky, ale zároveň jsou opačného genderu. Tímto způsobem se statisticky vyrovná identifikovaná nerovnováha a výsledný AI model by se měl chovat lépe.

Problémem tohoto přístupu je to, že identifikovat v čem přesně bias v datech spočívá a jak mají vypadat v jejich „čisté“ podobě, není snadné, což je hlavní překážkou tohoto procesu.

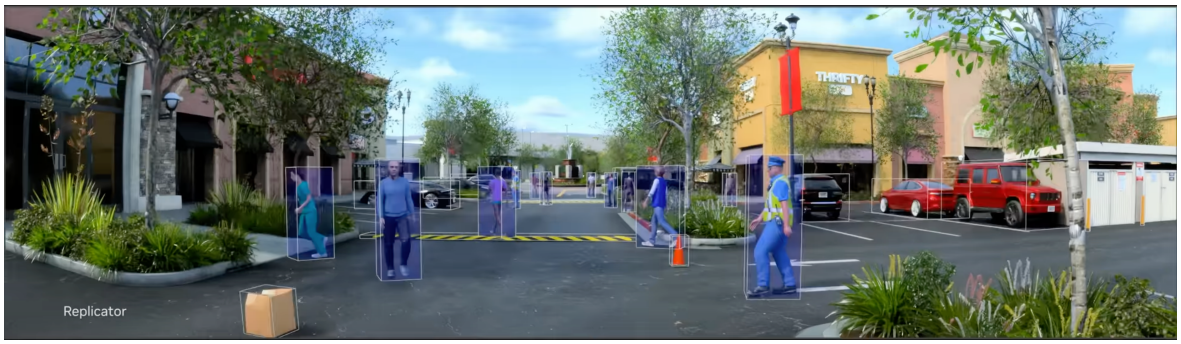
Určité řešení je v tomto ohledu spatřováno ve spolupráci několika umělých inteligencí. Pro každý problém je ale v takovém případě potřeba proces opakovat téměř od nuly. To je také důvod proč oblast syntetických dat v současnosti prochází fází vystřízlivění.

Konečné *halucinace* AI je situace, kdy AI poskytne sebejistě odpověď na otázku, ale ta je nesprávná. Tady narážíme na to, že AI současnosti jsou tzv. *úzké*, tedy zaměřené vždy na řešení určitého konkrétního problému. To také znamená, že AI ve skutečnosti nemá implementován koncept pravdy, nerozumí zpracovávanému textu a prostě svému uživateli poskytne odpověď, kterou vyhodnotí jako nejpravděpodobnější.

Informace o tom, nakolik si je AI jistá svou odpovědí, ale uživatel nezíská, protože v současnosti používané modely ji nejsou schopny poskytnout.

Halucinace je proto možno v současnosti pouze omezit, ale není jim možné zcela zabránit. Ověřenými strategiemi, které používají vývojáři AI jsou:

- použití obsáhlejších datasetů - halucinace mají tendenci vzniká především tam, kde AI nemá o určitém tématu dostatek informací.
- experimentování se strukturou použitého modelu - hledání modelů, které jsou méně náchylné k tomuto problému
- poskytování zpětné vazby k AI generovaným odpovědím
- syntetická data (pokud je možné získat je v dostatečné kvalitě)



Obrázek 1.11: Situace (odlišná od předchozích) doplněná popisky objektů (převzato z [60])

- manuální úprava obsahu, respektive odpovědi, které je správná/očekávaná u určitých typů otázek

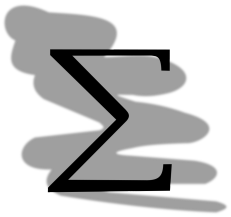
Výše uvedené lze také interpretovat tak, že halucinace a problémy s ní spojené tady s námi ještě nějakou, možná také hodně dlouhou, dobu pobudou.

Řešení by mohl přinést nástup AGI. AGI totiž již není úzce zaměřená, jedná se proto o umělou inteligenci v pravém slova smyslu, nikoliv „načančanou statistiku“ za jakou lze považovat v současnosti využívané systémy umělé inteligence.

Na hype křivce je AGI přesně na rozhraní mezi startem inovace a vrcholem očekávání. To znamená, že vrcholu hype křivky v tomto případě ještě nebylo dosaženo. Čistě z pohledu časového se použitelnost této technologie v příštích 10-ti letech neočekává.

Dosažení AGI by ale bylo skutečným milníkem ve vývoji umělé inteligence a služeb, které by mohla poskytovat a tak se do této oblasti výzkumu investují ohromné finanční prostředky. Problémem je, že v současnosti si nejsme jisti, jak by tohoto cíle mohlo být dosaženo.

Zásadním problémem je, že nerozumíme to, alespoň tedy technicky, jak funguje naše inteligence. Chápeme pouze, že jsme inteligentní a dopady tohoto stavu. Porozumění inteligenci je často považováno za podmínku nutnou k tomu, abychom mohli ji mohli začít replikovat uměle. Z tohoto pohledu může být hranice 10-ti let ještě optimistická. Je teoreticky také možné, že tohoto zlomového momentu nebude dosaženo nikdy.



Shrnutí

V této kapitole jsme se seznámili se základními vývojovými směry v oblasti umělé inteligence:

- strojové zpracování jazyka
- expertní systémy
- multiagentní systémy
- neuronové sítě

Tyto směry jsme zařadili do historického kontextu a souvislostí s dalšími iniciativami a postupy jako je např. iniciativa Průmysl 4.0, nebo tzv. big data.

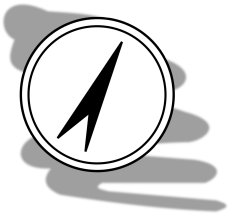
Podívali jsme se také lehce do budoucnosti k základním možnostem a omezením generativní AI a AGI.

Kontrolní otázky

1. Podívejte se na seznam směrů umělé inteligence a definujte každý z nich (jak funguje/co dělá).
2. Vyjmenujte alespoň tři oblasti, které řadíme do iniciativy Průmysl 4.0
3. Co jsou to big data?
4. Co je to promluva?
5. Jak funguje virtuální asistent?
6. Co je to halucinace (u generativní AI) a jak ji omezit?
7. Čím se liší AGI od běžné, v současnosti používané AI?

Kapitola 2

Úvod do R



Náhled kapitoly
Účelem této kapitoly je připravit si k použití nástroj (a základní znalosti jeho obsluhy), který budeme potřebovat pro hlavní předmět našeho zájmu v těchto skriptech - *neuronové sítě* a metodu hloubkového učení (deep learning). Tímto nástrojem bude R.
Pozor - tato kapitola je zaměřena prakticky. Předtím, než vůbec začnete studovat R jako nástroj - nainstalujte jej a jednotlivé probírané kroky rovnou zkoušejte na počítači. Zjistíte, že to, co ve skriptech není příliš jasné, se náhle vyjasní při praktické aplikaci.
Nebojte se experimentovat. Skripta mohou obsahovat vždy pouze základy. Ilustrační příklady tak lze jednoduše dále rozšiřovat. Experimentováním jednak utužíte znalosti, jednak lépe proniknete do probírané problematiky.

Po prostudování této kapitoly budete
mít

- k dispozici špičkové analytické prostředí
- základní znalosti o filozofii použití R

umět

- řešit některé základní problémy v R (načítat data, manipulovat s nimi, vykreslovat grafy)

Čas pro studium
Prostudování této kapitoly může proběhnout buďto velmi rychle nebo také velmi pomalu. Při seznamování se s programem Vám doporučuji, postupovat spíše pomalu a zaměřit se na pochopení filozofie fungování programu. Čas, který do nástroje investujete se Vám bohatě vrátí při zpracování dat např. pro bakalářskou nebo diplomovou práci a také v praxi při zpracování libovolných dat.

2.1 Instalace prostředí

Před započetím práce bude potřeba připravit si pracovní prostředí. Během výuky budeme používat systém R [14] a také editor RStudio [5]. V obou případech se jedná o open source nástroje, které jsou dostupné k použití zdarma, byť pro RStudio existuje, komerční placená varianta. Oproti open sorce verzi má verze placená některé pokročilejší funkce zejména pro podporu práce v týmech. A je určená zejména velkým výzkumným institucím, statistickým úřadům, pojišťovnám apod.

Nejprve provedeme instalaci samotného R. Prostředí je dostupné prakticky pro všechny používané operační systémy na https://www.r-project.org/. Stažení se provádí z CRAN. CRAN je síť serverů

(zrcadel) sloužících pro obsluhu základní instalace R a také jeho toolkitů.

Jednotlivé servery jsou rozmístěny po celém světě, čímž je zajištěno, že stahování probíhá z místa, které je nejblíže uživatelům (a také je tím řešena vzájemná zastupitelnost v případě výpadku).

Základní instalace prostředí je dostupná Windows, Linux a Mac.

MS Windows

Přímý odkaz je https://cran.r-project.org/bin/windows/base/. Pro instalaci je potřeba stáhnout a spustit instalátor zvolené verze. Pro instalaci doporučujeme vždy poslední stabilní dostupnou verzi (v době psaní těchto skript to byla verze 3.6.1).

Po dokončení průvodce, budete mít k dispozici základní instalaci R, včetně jednoduchého editoru. Prostředí bude mít schopnost instalovat rozšiřující balíky, ale pouze v případě, že jsou dostupné v binární podobě. Bohužel velká část toolkitů je dostupná pouze v podobě zdrojového kódu, který je před použitím potřeba zkompilovat.

R tuto činnost provádí automatizovaně (z pohledu uživatele je většinou jediným rozdílem trochu delší doba instalace toolkitu) - potřebuje ale mít naistalovaná navíc RTools - ty jsou dostupné ke stažení z https://cran.r-project.org/bin/windows/Rtools/.

Po dokončení instalace RTools je možno ještě pokračovat dále instalací MikTeX, InnoSetup a Perl - tyto balíky ale nejsou povinné a pro naši práci je potřebovat nebudeme.

Ze stránek https://posit.co/downloads/ stáhneme a nainstalujeme *RStudio Desktop* - z dostupných edicí zvolíme Open Source Licenci, která je dostupná bezplatně a pro naše účely plně dostačuje.

Po dokončení instalace RStudia máme k dispozici vše potřebné pro zahájení práce.

Mac (OS X)

Instalace pro Mac probíhá ve skutečnosti velmi podobně jako instalace pro Windows. Základní prostředí R instalujeme z balíku (.pkg souboru) staženého z https://cran.r-project.org/bin/macosx/ a RStudio z https://www.rstudio.com/.

Pro podporu kompilace je potřeba ale doinstalovat podporu pro Clang a Fortran, dostupnou z https://cran.r-project.org/bin/macosx/tools/. Podporu Tcl/Tk není potřeba moderních instalacích R řešit (je vestavěná). I tyto balíky stáhneme a nainstalujeme.

Linux

Verze R a RStudia je dostupná v repozitářích řady distribucí Linux. Tato verze ale nutně není nejaktuálnější. Pro naše účely by ale měla stačit.

Pro Ubuntu (a jiné distribuce založené na Debian) by mohla instalace z příkazové řádky vypadat následovně.

Listing 2.1: Instalace R a RStudio v distribucích Linux založených na Debian

```
sudo apt update
sudo apt-get install r-base
sudo apt-get install gdebi-core
cd ~/Downloads
wget https://download1.rstudio.org/rstudio-xenial-1.1.456-amd64.deb
sudo gdebi rstudio-xenial-1.1.456-amd64.deb
```

Prosím berte v úvahu, že Vaše instalace se může drobně lišit např. podle verze RStudia, kterou stáhnete. Místo instalace z repozitářů Vaší distribude Linux, lze také přidat oficiální repozitář R podle návodu [65] a R instalovat z něj.

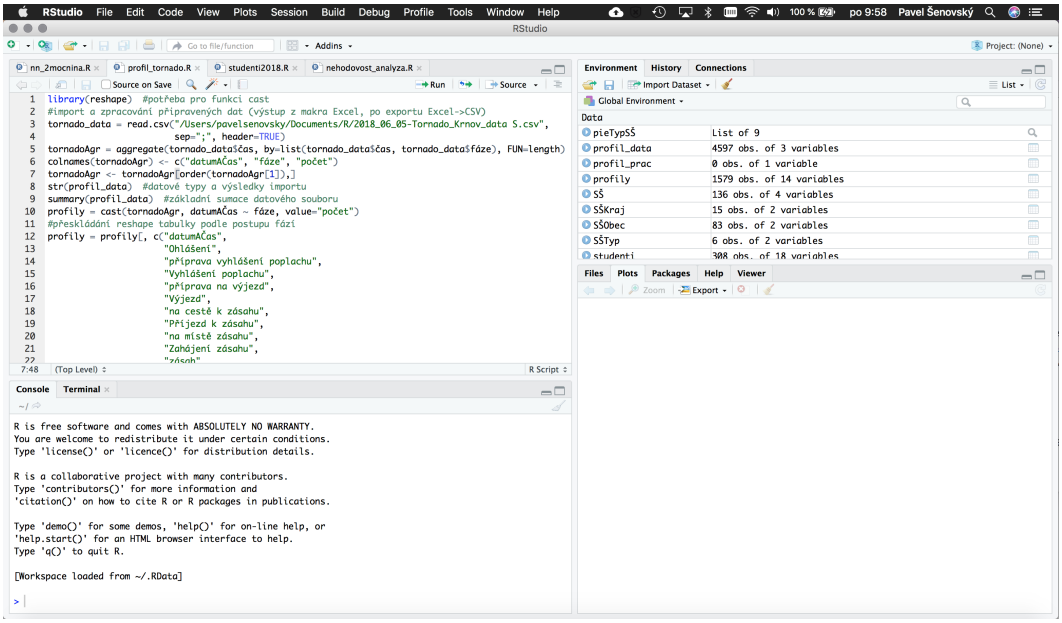
Podle stavu Vaší instalace Linuxu, mohou být vyžadovány také některé další balíky. Linux ale obvykle velmi dobře signalizuje, které balíky mu chybí. Takže odstranění těchto problémů by mělo být relativně bezbolestné.

Distribuce Linux navíc obsahují nástroje vyžadované pro kompilaci toolkitů, takže instalace dalších podpůrných balíků obvykle není vyžadována.

Oficiálně podporované jsou pak i distribuce RedHat (Fedora) a SuSe. Pokud používáte jinou distribuci Linux, bude nutné provést nejspíše kompilaci ze zdrojových kódů.

2.2 RStudio

Zkusme prozkoumat RStudio. Po spuštění se zobrazí rozhraní podobné obr. 2.1. V mém případě je RStudio provozováno na Macu (OS X), podle zvoleného operačního systému se proto vzhled může drobně lišit. Funkčně, ale mezi různými operačními systémy rozdíl není.



Obrázek 2.1: Rozhraní RStudio v OS X

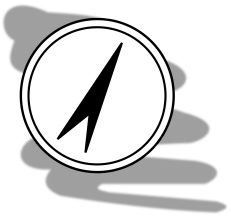
Rozhraní RStudio má čtyři základní části:

- zdrojové kódy našich analytických projektů, popř. prohlížeč dat (vlevo nahoře)
- konzole R (vlevo dole)
- prostředí (s jednotlivými načtenými datasety) - vpravo nahoře
- soubory, grafy, nápověda ... vpravo dole

Nejvíce času strávíme zápisem příkazů jazyka R - tedy v části vlevo nahoře. Zde můžeme mít otevřenu řadu záložek obsahující soubory .R, se kterými právě pracujeme.

Např. na obr. 2.1 je znázorněn soubor řešící vytvoření časového profilu výjezdů jednotek HZS k zásahům vyvolaným tornádem v Krnově (2018). Editor podporuje zvýrazňování syntaxe, poskytuje bublinkovou nápovědu k jednotlivým parametrům používaných funkcí a také kontextově nabízí jména funkcí samotných podle toho, co píšeme, jedná se tedy o obdobu intelisence známou z vývojového prostředí Visual Studio - napovídající se např. jména samotných funkcí.

Zapsané příkazy lze pak spouštět buďto jednotlivě po řádcích, postupným klikáním na tlačítko *Run*, nebo lze označit segmenty kódu, popř. celý obsah souboru a Run pak provede všechny takto označené kroky.

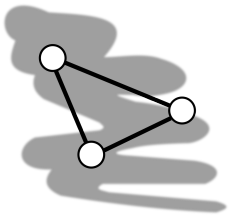


Kód R

Kód v R za určitých okolností dokáže být náročný. V tomto předmětu ale v R nebudeme programovat v tradičním smyslu tohoto slova. Kód je proto potřeba si představit spíše jako posloupnost funkcí tabulkového procesoru, ale bez nutnosti věci naklikat.

Okno konzole obsahuje interaktivní konzoli R, do které lze přímo zadávat příkazy. Většina příkazů se zadává prostřednictvím kódu v editoru vlevo nahoře. Spouštění kódu pak generuje výstupy v konzoli. V případě, že ale chceme realizovat jednorázovou akci - např. instalaci nového toolkitu, lze příkaz zadat do konzole přímo.

Při výkladu samotného jazyka tak budeme zkoumat jednak funkce R, jednak jejich výstup.



Proč R?
... a ne třeba Excel. MS Excel je skvělý produkt, ale v případě zpracování velkých objemů dat není právě nejpříjemnější. Excel také neobsahuje plnohodnotnou podporu statistických nástrojů, nebo nástrojů data miningu. K těmto účelům jej tak lze použít, ale uživateli s řešením těchto problémů příliš nepomůže. Také natrénování neuronové sítě není věc, se kterou by nám Excel pomohl.

Oproti tomu R prostřednictvím různých toolkitů má k dispozici celou škálu nástrojů pro řešení různých typů problémů, bez velikostních omezení. Produktivita práce je tak obvykle v případě R vyšší než u tabulkových procesorů. Cenou za rychlost je ale nutnost naučit se nové funkce (nebo se je naučit „vygooglovat“).

Vpravo nahoře je dostupný výpis prostředí R - zejména jaké proměnné jsou definovány se základními informacemi. V seznamu proměnných je vidět počet pozorování z počtu proměnných. To lze interpretovat jako počet řádků ve sloupcích - např. 4597 obs. of 3 variables znamená tabulka 4597 řádků a 3 sloupce.

Dvojitým kliknutím na název proměnné ji lze spustit v prohlížeči. Např. proměnná *tornado_ data* vypadá v prohlížeči jako na obr. 2.2.

Filter

	Název.JPO	fáze	čas
1	stanice Krnov – 811011	Ohlášení	18.6.2013 17:30
2	stanice Krnov – 811011	Vyhlášení poplachu	18.6.2013 17:30
3	stanice Krnov – 811011	Výjezd	18.6.2013 17:31
4	stanice Krnov – 811011	Příjezd k zásahu	18.6.2013 17:33
5	stanice Krnov – 811011	Zahájení zásahu	18.6.2013 18:05
6	stanice Krnov – 811011	Odjezd na základnu	19.6.2013 4:35
7	stanice Krnov – 811011	Příjezd na základnu	19.6.2013 4:58
8	stanice Krnov – 811011	na cestě k zásahu	18.6.2013 17:32
9	stanice Krnov – 811011	na místě zásahu	18.6.2013 17:34
10	stanice Krnov – 811011	na místě zásahu	18.6.2013 17:35
11	stanice Krnov – 811011	na místě zásahu	18.6.2013 17:36
12	stanice Krnov – 811011	na místě zásahu	18.6.2013 17:37
13	stanice Krnov – 811011	na místě zásahu	18.6.2013 17:38
14	stanice Krnov – 811011	na místě zásahu	18.6.2013 17:39
15	stanice Krnov – 811011	na místě zásahu	18.6.2013 17:40

Showing 1 to 15 of 27,361 entries

Obrázek 2.2: Data Viewer - proměnná tornado_data

Všimněte si, že do prohlížeče nejsou načtena všechna data, ale pouze 15 záznamů. Prohlížeč totiž primárně slouží pro náhled nebo průzkum načtených proměnných, nikoliv jejich editaci. R předpokládá, že data byla pořízena v jiném systému. Do R se tak pouze naimportují a pak zpracují.

V prohlížeči tak pohodlně zjistíme, jak data vypadají, což nám umožní se následně efektivně rozhodnout o tom, co s nimi budeme dělat dále. V prohlížeči lze data také filtrovat a vyzkoušet si tak např. různé typy výběrů dat.

Konečně vpravo dole se zobrazuje případná grafika, jako jsou grafy, apod. Lze zde také prohlížet vestavěnou nápovědu apod.

2.3 Základy jazyka R

Od této sekce budou přikládány funkční segmenty R kódu, jejichž funkci si můžete prakticky vyzkoušet ve vlastní instalaci R. Doporučujeme proto nečíst text „na sucho“ - ale prakticky s ním experimentovat.

Operátory a proměnné
R podporuje všechny běžné operátory: +, -, *, /, ^(umocnění), %% (zbytek po dělení). Pro přiřazení

lze použít buďto = nebo <- (lomená závorka následovaná pomlčkou). My v zápisech budeme používat rovnítko, v literatuře se ale <- používá velmi často.

Experimentovat můžeme v tomto případě přímo v konzoli.

Listing 2.2: Základní operátory v R a jejich použití

```
> 2 + 2
[1] 4
> 3 %% 2
[1] 1
> a = 5
> b = 3
> c = a * b
> a
[1] 5
> b
[1] 3
> c
[1] 15
```

v kódu výše rozlišujte mezi řádky začínající > tyto řádky odpovídají příkazům zadávaných do konzole. Řádky bez > jsou pak odpovědí systému na příkaz.

Tedy máme nějaké jednoduché sčítání (2 + 2), zbytek po dělení (3 %% 2) a do proměnné a přiřadíme hodnotu 5, do b hodnotu 3, obě proměnné sečteme a výsledek uložíme do proměnné c.

Obsah proměnné zobrazíme zavoláním jejího jména.

Pro úplnost uvádíme také operátory porovnání: rovno ==, není rovno !=, větší >, větší nebo rovno >=, menší <, menší nebo rovno <=.

Základní funkce

Komentáře začínají znakem #. Vše so po tomto znaku na daném řádku následuje, je považováno za komentář.

Listing 2.3: Základní funkce v R

```
sqrt(4) #√4 = 2
abs(-4) #|−4| = 4
round(x, 2) #zaokrouhlí x na 2 desetinná místa
log(x) #přirozený logaritmus
log10(x) #logaritmus x o základu 10
log2(x)  #logaritmus x o základu 2
trunc(x) #odebrání desetinných míst (zůstane celá část x)
ceiling(x) #zaokrouhlení x nahoru
floor(x) #zaokrouhlení x dolů
exp(x)  #e^x
sin(x), cosin(x), tan(x) #trigonometrické funkce
asin(x), acos(x), atan(x) #inverzní trigonometrické funkce
```

Použití základních funkcí je relativně jednoduché, podívejme se jak na vektory.

Vektory

Listing 2.4: Manipulace s vektory v R

```
a = c(1, 2, 3, 4) #vytvoří vektor 1, 2, 3, 4
b = c(TRUE, FALSE, TRUE, FALSE) # vytvoří vektor boolenovských hodnot
c = c("one", "two", "three") # vytvoří vektor textových řetězců
class(a) #slouží pro zjištění datového typu ( výsledek je numeric, character, integer,
complex, logical)
d = as.integer(3) #vytvoří proměnnou typu integer o hodnotě 3
```

```
e = rep(v, times=3) #vytvoří vektor tak, že zopakuje první parameter (v) times krát
f = 1:5 #vytvoří vektor na základě sekvence 1, 2, 3, 4, 5, tedy od 1 do 5
g = seq(100, 2000, by=300) #vytvoří vektor na základě sekvence od 100 do 2000 s krokem 300
```

Základní funkcí, pro vytvoření vektoru je *c()*. Do závorek - parametrů funkce c - se pak dávají jednotlivé prvky, ze kterých má být vektor složen. Tímto způsobem lze skládat také vektory. Zkusme vytvořit vektory a, b obsahující čísla a následně tyto vektory spojme funkcí c(). Výsledný vektor bude obsahovat všechny prvky obou spojovaných vektorů.

Listing 2.5: Spojování vektorů v R

```
> a = c(1, 2, 3)
> b = c(4, 5, 6)
> c = c(a, b)
> c
[1] 1 2 3 4 5 6
```

Data.Frame

Nejpoužívanější datovou strukturou v R je tzv. *Data.Frame*. Data.Frame si lze jednoduše představit jako tabulku. Zkusme udělat jednoduchý Data.Frame na základě vektorů obsahujících údaje o měsíčních prodejích:

Listing 2.6: Realizace Data framu spojením vektorů v R

```
> ID = c(1,2,3,4)
> mesic = c("Leden", "Unor", "Brezen", "Duben")
> prodeje = c(15000, 20000, 125000, 40000)
> region = c("vychod", "zapad", "jih", "sever")
> prodejData = data.frame(ID, mesic, prodeje, region)
> prodejData
ID mesic prodeje region
1 1 Leden 15000 vychod
2 2 Unor 20000 zapad
3 3 Brezen 125000 jih
4 4 Duben 40000 sever
```

My z praktických důvodů budeme načítat většinou Data.Frame z externích souborů.
S jednotlivými sloupci tabulky lze poměrně dobře manipulovat.

Listing 2.7: Manipulace se sloupci Data framu v R

```
prodejData[2] # vypíše sloupec mesic
prodejData[c("mesic", "region")] #vypíše sloupce měsíc a region
prodejData$prodeje  # vypíše sloupec prodeje
prodejData[1:2] #vypíše první 2 sloupce
prodejData[-c(2,3)] #odebere sloupce 2 a 3
prodejData[-3] #odebere sloupec 3
prodejData$ID = NULL #odebere sloupec ID
#přidá další řádek
prodejData[nrow(prodejData) + 1,] = c(5, "Unor", 25000, "vychod")
prodejData = prodejData[-nrow(prodejData)] #smaže poslední řádek
```

Všimněte si trošku odlišné práce buďto s označováním sloupců přes $ nebo přes [ ]. Zatímco notace $ umožňuje přistoupit pouze k jednomu (pojmenovanému sloupci), pak [ ] umožňuje filtrovat tabulku, např.

Listing 2.8: Aplikace filtrů na sloupce Data framu v R

```
> podminka = prodejData$prodeje > 20000 & prodejData$mesic == "Leden"
```

```
> prodejData[podminka,] #vybere řádky splňující podmínku
[1] ID mesic prodeje region
<0 rows> (or 0-length row.names)
> prodejData[!podminka,] #vybere řádky, které nesplňují podmínku
ID mesic prodeje region
1 1 Leden 15000 vychod
2 2 Unor 20000 zapad
3 3 Brezen 125000 jih
4 4 Duben 40000 sever
5 5 Unor 25000 vychod
```

Podmínka je vytvořena na prodeje v lednu větší než 20 000. V našich datech, ale řádky splňující tuto podmínku nejsou - R zahlásí <0 rows>. Alternativně můžeme explicitně vypsat řádky které podmínku nesplňují (!pominka). V tomto případě se nám zobrazí všechna data.

Tabulky je možno také řadit:

Listing 2.9: Řazení v Data framu

```
prodejData[order(prodejData$prodeje),] #seřadit řádky datasetu prodejData podle prodejů
#seřadí podle prodejů a tam kde jsou stejné podle měsíce
prodejData[order(prodejData$prodeje, prodejData$mesic)]
prodejData[, order(names(prodejData))] #seřadí abecedně sloupce datasetu
```

Všimněte si, že u řazení a filtrací rozhoduje o tom, na co se budou aplikovat čárka v [ ] - tedy [řádky, sloupce].

Faktory

Pokud se podíváme na sloupec *měsic* v předchozí tabulce - ve skutečnosti měsíc nutně nemusí být vyjádřen textovým řetězcem - mohli bychom jej vyjádřit také číselně: 1 - leden, 2 - únor, ...

Podobně když uvažujeme o známkách ve škole pak můžeme známku vyjádřit slovně nebo číselně: 1 - výborně, 2 - velmi dobře, ... V případě měsíců i známek nám přitom záleží na přesném mapování čísel a jejich textového označení. Každopádně se z analytického pohledu jedná o tzv. nominální popř. ordinální proměnné.

K nominálním proměnným nelze připojit „kvantitu“ - tedy, pokud hodnotám takových proměnných přiřadíme čísla, pak tato čísla jsou pouze rozlišujícím znakem, ale neumožňují porovnání. Známkování je pak dobrým příkladem ordinální proměnné - zde lze provést řazení, podle zvoleného klíče.

R tento typ transformací řeší přes faktory. Uvažujme o příkladu známkování:

Listing 2.10: Použití faktorů v R

```
znamky = c("vyborne", "velmi dobre", "dobre", "nedostatecne") #vektor typu character
znamky = factor(znamky)
znamky
```

Funkce *factor* přiřadí převáděným položkám integer hodnotu, tato hodnota, ale není viditelná. Jak to funguje můžeme zjistit porovnáním dvou prvků vektoru známky (po převodu na faktor). Můžeme zkusit např. min(znamky), ale R se bude bránit. Pomocí factor jsme v našem příkladu totiž převedli proměnnou typu character na nominální proměnnou. My už víme, že v takovém případě porovnávání nemá smysl.

Ordinální proměnnou uděláme z nominální specifikací hodnot použité škály v rámci funkce factor:

Listing 2.11: Vytvoření ordinální proměnné v R

```
> znamky = c("vyborne", "velmi dobre", "dobre", "nedostatecne")
> znamky = factor(znamky,
        levels=c("vyborne", "velmi dobre", "dobre", "nedostatecne"),
        ordered=TRUE)
> znamky[1] > znamky[2]
[1] FALSE
```

```
> min(znamky)
[1] vyborne
Levels: vyborne < velmi dobre < dobre < nedostatecne
> max(znamky)
[1] nedostatecne
Levels: vyborne < velmi dobre < dobre < nedostatecne
```

Nejen, že v tomto případě lze vzájemně porovnávat položky, lze používat i některé funkce (např. min a max).

Známkování - změna pořadí
Z čistě formálního pohledu by v příkladu výše mělo být pro známkování použito obrácené pořadí - použijte své nově nabyté znalosti a změňte pořadí tak, aby odpovídalo způsobu jak obvykle známkujeme.

Import a export dat v R

Jednu z nejčastějších úloh, kterou potřebujeme řešit prakticky v každém data miningovém projektu je import dat. Data lze importovat z různých datových zdrojů (různých databází, MS Excel, textových souborů apod.). Při importu je potřeba vždy zohlednit omezení, která nástroj pro import má.

Jako nejjednodušší a zároveň nejspolehlivější se většinou uvádí výměnný formát Comma Separated Values (CSV). Jedná se o čistě textový formát, kde řádek tabulky tvoří jeden řádek textového souboru a sloupce jsou oddělovány zvoleným oddělovačem - např. čárkou (může být však použit také jiný oddělovač).

Import z CSV souboru by mohl vypadat následovně:

Listing 2.12: Načtení CSV souboru

```
import_data = read.csv(cesta, sep=";", header=TRUE)
str(import_data)
```

V tomto případě se bude importovat soubor identifikovaný cestou k tomuto souboru. Jako oddělovač sloupců je použit středník. Importovaný soubor obsahuje v prvním řádku hlavičku. Za import samotný je zodpovědná funkce *read.csv*.

Kromě této funkce existují další, které lze k tomuto účelu použít:

- *read.table* - nejuniverzálnější funkce pro import textových datových souborů
- *read.delim* - pro textové soubory s tabulátorem jako oddělovačem sloupců
- *read.csv2* a *read.delim2* - jako oddělovač desetinných míst se používá čárka

Všimněte si také funkce str v příkladu výše - ta je velmi důležitá protože vypíše informaci o identifikovaných datových formátech jednotlivých sloupců a celkový počet řádek. Tady je potřeba dodat, že způsob, jakým jsou data importována často neodpovídá záměrům, které s daty máme - řádky je proto potřeba často upravovat, měnit jejich formáty apod.

Pro import z relačních databází je možno použít toolkity specializované na danou databázi, např. *RMySQL*, *ROracle*, *RPostgreSQL*, *RSQLite*, *RODBC*. Posledně zmíněný je možno považovat za do velké míry univerzální, protože umožňuje přistoupit k databázi pomocí Open Database Connect (ODBC) rozhraní. Přitom prakticky všechny relační databáze ODBC v nějaké formě podporují.

Analogickou operací k importu je export dat. V případě, že chceme použít CSV soubor, mohl by export vypadat následovně.

Listing 2.13: Zápis dat do textového souboru

```
write.table(data_k_exp, file = "cesta k souboru.csv", sep=";", row.names=FALSE,
        fileEncoding = "UTF-8")
```

Export je tedy relativně přímočarý. Data musí před voláním funkce write.table uložená jako tabulka nebo data.frame. V případě, že data obsahují text s interpunkčními znaménky je potřeba specifikovat

také parametr fileEncoding specifikující znakovou sadu, která má být použita během exportu. V našem případě je používáno UTF-8, což je více méně univerzální kódování.

Parametr row.names specifikuje, že se nemají při exportu používat popisky řádků. Popisky R přidává jinak k datům automaticky prostě tak, že k řádku přidá pořadové číslo tohoto řádku v datovém souboru.

Základní statistické funkce

Základní přehled funkcí:

Listing 2.14: Základní statistické funkce v R

```
mean(x) #průměr z x
length(x) #počet prvků v x
median(x) #medián z x
unique(dataset) #z předloženého datasetu vytvoří vektor obsahující pouze unikátní položky datasetu (tedy bez opakování)
range(x) #vrací rozsah hodnot v předloženém datasetu (nejmenší a největší), vrací jej jako dvouprvkový vektor
quantile(x, probs=seq(0, 1, 0.25)) #R nezná quartily, v tomto případě se to obchází jednoduchou sekcencí od 0 do 1 po 0.25. Místo sekvence lze zadat i jednu položku
sd(x) #standardní odchylka (standard deviation)
min(x), max(x) #minimum a maximum
var(x) #rozptyl
summary(x) # poskytuje přehled základních statistik (min, první kvartil, medián, průměr, třetí kvartil, max), pokud u výše uvedených funkcí místo parametru ve formátu vektoru použijeme data.frame, provede se výpočet pro všechny sloupce framu.
```

Vykreslování grafů

R obsahuje řadu režimů a knihoven pro vykreslování grafů. My budeme v tomto textu používat pouze tu nejjednodušší používající funkce:

- *plot* - běžné grafy
- *hist* - histogram
- *pie* - koláčový graf
- *boxplot* - graf typu box and whiskers

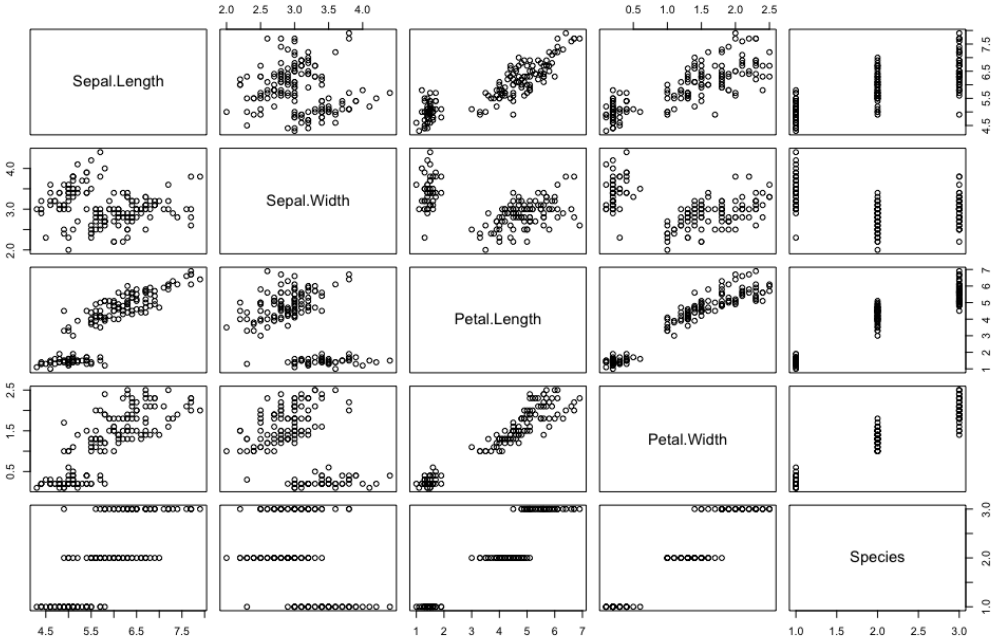
Bohužel tento typ grafů nevypadá vizuálně hezky. R poskytuje alternativu k tomuto prostředí pomocí knihovny *GGPlot2*, která umožňuje vytvářet graficky extrémně sofistikovanou grafiku. Cenou za větší možnosti knihovny je ale podstatně vyšší komplexita nastavení.

My v těchto skriptech nebudeme mít prostor pro zvládnutí této knihovny, případně se ale do studia můžete pustit sami. Grafické příklady (k motivaci) lze najít např. na Google při vyhledání „*image ggplot2*“.

Pro demonstraci můžeme použít vestavěný dataset IRIS, který obsahuje některé charakteristiky odrůd kosatců:

Listing 2.15: Vykreslování základních typů grafů pro dataset IRIS

```
summary(iris)
plot(iris)
plot(iris$Sepal.Length~iris$Sepal.Width, col="red",
       main="Delka vs sirka kosatcu", xlab="delka listu",
       ylab="sirka listu")
hist(iris$Sepal.Length, col="blue", main="Rozlozeni delky listu",
       xlab="delka listu", ylab="frekvence")
boxplot(iris, las=2,
main="Boxplot graf namerenych vlastnosti kosatcu", at=c(1,2,4,5,7),
names=c("Sep.length", "Sep.width", "Pet.length", "Pet.wight", "Species"),
col=c("red", "red", "blue", "blue", "green", "green"))
```



Obrázek 2.3: Párové srovnání vlastností kosatců v datové sadě IRIS

Datový soubor obsahuje popis vlastností vzorku rostlin - především pak délku a šířku listů a okvětních lístků. Funkce plot(iris) vykreslí párové srovnání sloupců datového souboru, viz obr. 2.3.

Názvy sloupců se vykreslují na diagonále grafu. Všimněte si, že grafy nad a pod diagonálou jsou podle diagonály zrcadlově obráceny.

Takový typ grafu se často používá pro rychlou vizualizaci hodnot v datovém souboru. Mnohem častěji, ale potřebujeme získat větší kontrolu nad vykreslováním grafu.

Delší forma zápisu specifikuje bodový graf, ale definuje pro něj řadu parametrů. Prvním parametrem jsou přitom data, která se mají vykreslit. Tato data se definují ve formátu osa_X ~osa_Y. Oddělovačem os je pak znak tilda (vlnovka). Podle toho, co potřebujeme nastavit můžeme použít jeden nebo více parametrů.

V našem případě jsou nastavovány barva bodů (col), popisky os (xlab, ylab) hlavní nadpis grafu (main). Graf je znázorněn na obr. 2.4.

V rámci funkce plot lze také změnit způsob vykreslování. IRIS dataset se ale k tomuto účelu nehodí. Pro demonstraci proto použiji námi v rámci skript vytvořený dataset prodejData a vizualizovat budeme prodeje. Všechny typy vykreslování (příkladu níže) jsou znázorněny na obr. 2.5.

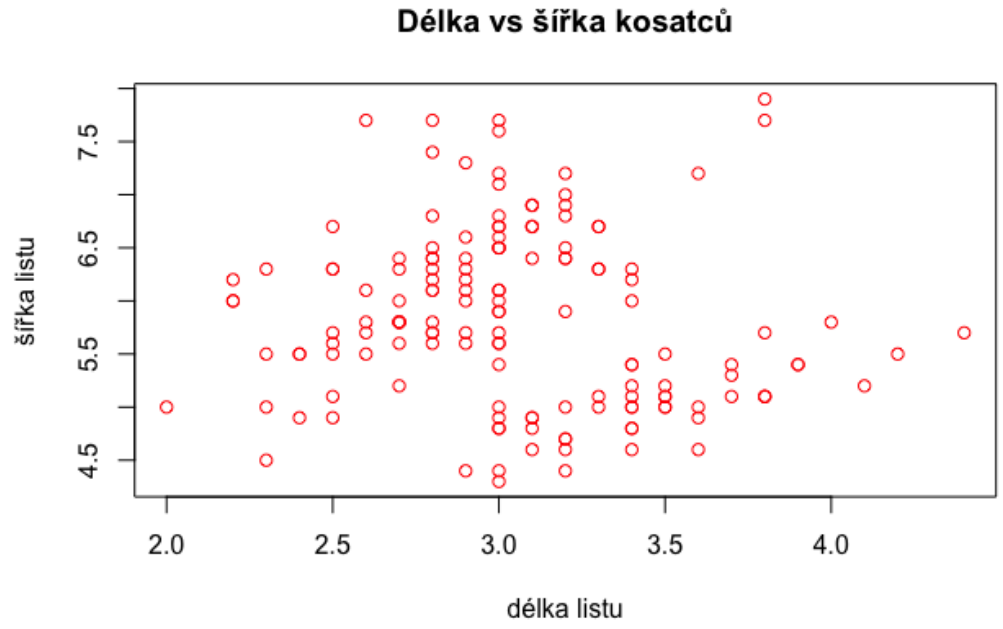
Listing 2.16: Funkce *plot* - jednotlivé typy grafů

```
1 plot(prodejData$prodeje, type="p")
2 plot(prodejData$prodeje, type="l")
3 plot(prodejData$prodeje, type="b")
4 plot(prodejData$prodeje, type="h")
```

Vraťme se ale k datasetu iris - příklad histogramu je na obr. 2.6. I u histogramu lze použít řadu parametrů. Pro obr. 2.6 byla pomocí parametru col nastavena barva na modrou a přidány popisky os a grafu jako takového (parametry xlab, ylab, main).

Dalším typem grafu, který graf typu box and whiskers - viz obr. 2.7. Tento typ grafu je velmi populární pro rychlou vizuální kontrolu rozložení hodnot jednotlivých proměnných. Struktura vykreslovaného boxu je následující (zhora dolů):

- horní outliners (hodnoty $>$ 1.5 * IQR, kde IQR (interquartile range) je oblast mezi 3. a 1. kvartilem)
- maximum



Obrázek 2.4: Délka vs šířka listu kosatců v datasetu IRIS (bodový graf)

- 3. kvartil
- medián
- 1. kvartil
- minimum
- dolní outliners

Také funkce boxplot má řadu parametrů, kromě těch, které jsme použili u ostatních grafů jsem použil v příkladu parametr las=2, který seskupuje položky po dvojicích. Tímto způsobem jsem zdůraznil souvislost mezi položkami. Patametr at se zadává jako vektor popisující umístění jednotlivých položek v grafu. Všimněte si že pozice 3 a 6 jsou vynechány čímž vznikají v grafu mezery.

Pomocí parametru names lze přepsat názvy polí. Pomocí parametru col jsou definovány barvy. V tomto případě jsou barvy zadávány vektorem.

Poslední typ grafu, který jsem ještě neprobrali, je graf koláčový vytvářený funkjcí *pie*. Tento graf je do určité míry specifický, protože vyžaduje určitou přípravu dat. Data, která jsou používána pro jeho vytvoření musí být agregovaná.

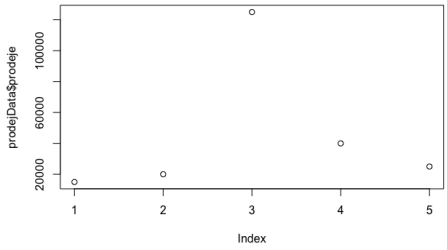
Funkce pie tak nemá agregaci zabudovanou přímo v sobě. Pro demonstraci použijeme náš upravený dataset prodejů. Jeho úpravu a vykreslení koláčového grafu budeme realizovat následujícím skriptem:

Listing 2.17: Tvorba koláčových grafů v R

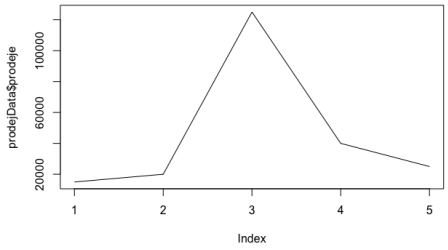
```
vykon <- c("mizerny", "mizerny", "skvely", "prumerny", "skvely")
vykon <- factor(vykon, levels=c("mizerny", "prumerny", "skvely"))
prodejData$vykon <- vykon
prodeje = table(prodejData$vykon)
prodeje
pie(prodeje, main="Rozlozeni prodeju")
```

Všimněte si způsobu, jakým došlo ke zpracování údajů v proměnné prodeje. Takové údaje je již možno zobrazit v koláčovém grafu, viz obr. 2.8.

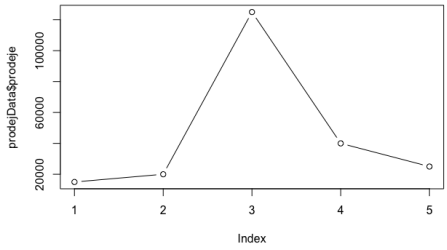
```
mizerný průměrný   skvělý
      2         1         2
```



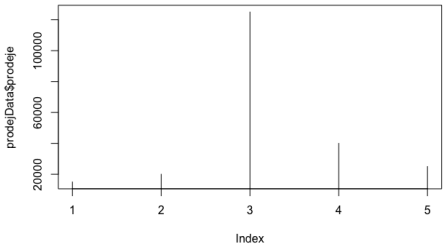
(a) type = „p“



(b) type = „l“



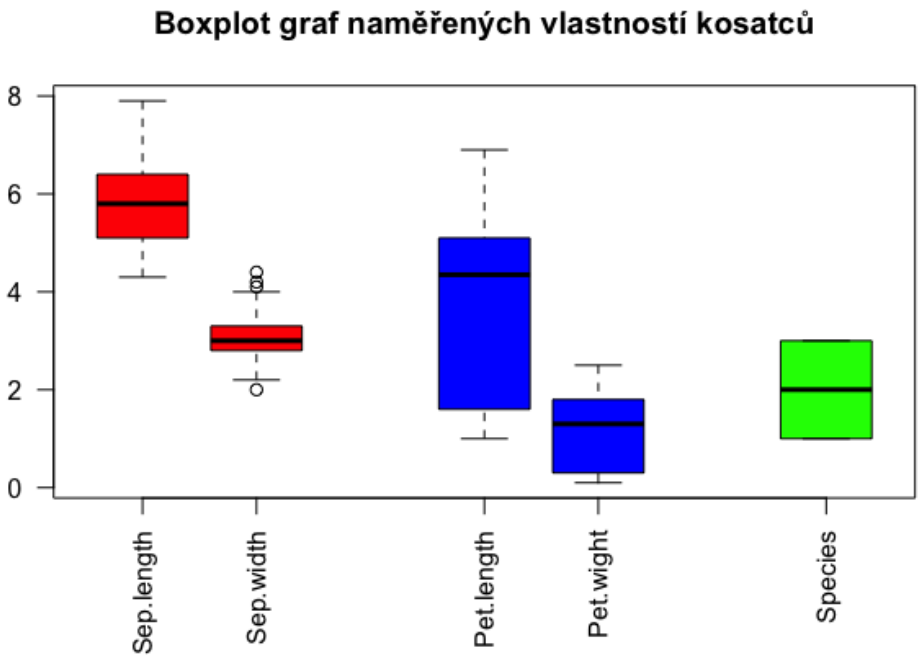
(c) type = „b“



(d) type = „h“

Obrázek 2.5: Dostupné typy vykreslování ve funkci plot

Obrázek 2.6: Rozložení délky listu dataset IRIS (histogram)



Obrázek 2.7: Délka listu dataset iris (boxplot)

Obrázek 2.8: Rozložení počtu prodejů podle jejich úspěšnosti (pie)

Místo shrnutí
V této kapitole jsme nainstalovali prostředí R a editor RStudio. Následně jsme prošli některé ez nejčastěji používaných postupů a funkcí.

Pokud jste pečlivě zkoušeli příklady uvedené v kapitole a chápete, jak systém funguje, gratuluji - zvládli jste základy R. Samozřejmě pro získáni jistoty v práci je potřeba prakticky realizovat v tomto prostředí nějakou práci.

Pokud jste jednotlivé příklady nerealizovali, tak teď je na to vhodná doba. Začněte procházet kapitolu, pěkně od začátku a zkuste, jak funguje každá funkce, o které jsme v kapitole psali.

V další kapitole totiž už budeme R potřebovat!

Kapitola 3

Neuronové sítě

Náhled kapitoly
Neuronové sítě jsou jednou z nejrychleji se rozvíjejících oblastí umělé inteligence. Zejména nástup moderních grafických karet a specializovaných Application Specific Integrated Circuit (ASIC) obvodů umožnila nasazení rozsáhlých mnohavrstevných neuronových sítí umožňujících v reálném čase řešit širokou škálu úloh od autonomního řízení automobilů, po automatizované překlady, systémů text-to-speech a dalších.

Po prostudování této kapitoly budete
vědět

- jak fungují vrstvené neuronové sítě

umět

- řešit jednoduché úlohy s použitím neuronových sítí

Čas pro studium
Pro prostudování budete potřebovat 2 - 4 hodiny.

Myšlenka umělých neuronových sítí není nikterak nová. Vlastně první model neuronu byl vytvořen podstatně dříve, než rozvoj výpočetní techniky umožnil takový model prakticky použít. Předtím než se ale vrhneme na podrobnosti, podívejme se na filozofické základy tohoto přístupu.

Pomocí neuronových sítí se snažíme napodobit způsob jakým živé organizmy myslí a to co možná nejblíže způsobu, jakým je organizován mozek - tedy pomocí neuronů a vazeb mezi nimi. Tím se přístup neuronových sítí liší od jiných oblastí uměle inteligence, např. expertních systémů, které se snaží formalizovat logiku myšlení exportů specifikací faktů a jejich použití v problémové doméně, nebo multiagentních systémů, kde jednotliví agenti mají implementovanou nějakou „malou“ funkcionalitu a pokročilé funkčnosti je dosahováno až vzájemnou interakcí jednotlivých agentů.

Neuronové sítě, které jsou v současnosti využívány, jsou ale relativně malé - obsahují tisíce až milióny neuronů podle problému a architektury neuronové sítě zvolené pro jeho řešení. Pokud to srovnáme s počty neuronů v mozcích, pak průměrný hlodavec má v mozku 12 a člověk 86 miliard neuronů [39].

U biologických organismů ale z hlediska myšlení nezáleží pouze na počtu neuronů. Např. mozek průměrného primáta obsahuje 93 miliard neuronů, což je více než kolik obsahuje mozek průměrného člověka (viz [39]). To, co považujeme za inteligenci je pak odvozováno také z toho, jak je mozek organizován, tedy k čemu jsou neurony používány.

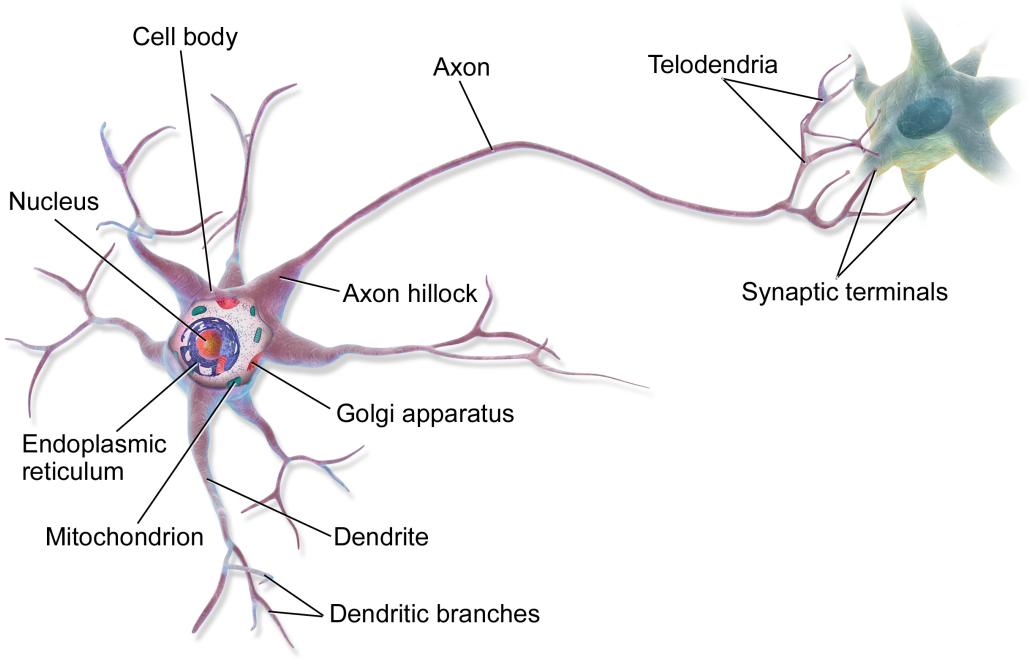
Umělé neuronové sítě, jak jsou používány dnes, ke svému tréningu obvykle vyžadují nasazení poměrně značného množství strojového času superpočítače. Takové neuronové sítě jsou ale vždy specializované na řešení určitého konkrétního úkolu. Náš mozek oproti tomu je univerzální.

Lze také říci, že v současnosti výzkum ani nesměřuje k vytvoření plnohodnotné umělé inteligence. Ne, že by výzkumníci dosáhnout tuto metu nechtěli, spíše není tak úplně jasné jakým způsobem bychom měli vůbec k řešení tohoto probléme přistoupit. Tyto plnohodnotné umělé inteligence označujeme jako AGI a v současnosti (počátek roku 2024) se jedná o čistě teoretický konstrukt, o kterém se ale hodně hovoří.

Vraťme se ale k neuronům jako takovým a jejich implementaci pomocí počítačových modelů.

3.1 Neuron

Schéma neuronu bychom mohli vizualizovat např. jako na obr. 3.1. Tato vizualizace ve skutečnosti vizuálně příliš neodpovídá realitě - srovnejte s obr. 3.2, který byl pořízen mikroskopem.



Obrázek 3.1: Schéma neuronu (převzato z [23])

Biologický neuron je buňka a proto jako každá buňka má jádro (nucleus), které pracuje jako „elektrárna“ buňky a umožňuje ji tak plnit svou funkci. Neurony mají řadu výběžků (dendritů) a jeden dlouhý výběžek označovaný jako axon (neurit). Pomocí axonu probíhá přenes tzv. nervových vzruchů. Výběžky axonu (telodendria) se pak připojují na další neurony.

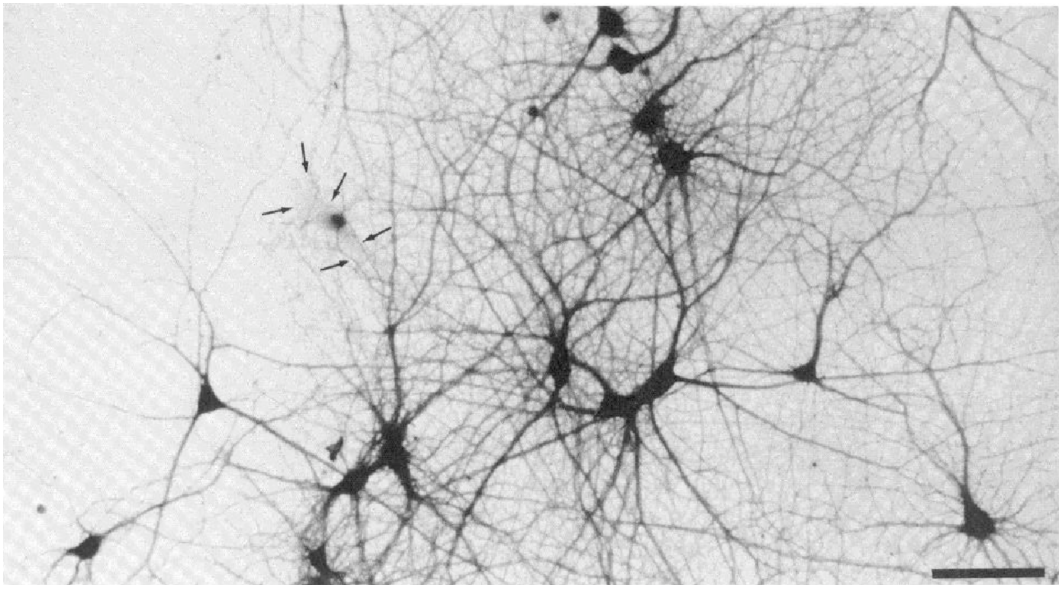
Přenos nervového vzruchu do dalšího neuronu je realizován prostřednictvím synapse. Synapse má velmi důležitou úlohu tzv. *prahování*. To si lze představit tak, že synapse přijme nervový vzruch - pokud intenzita tohoto vzruchu přesáhne hodnotu prahu, pak vzruch projde synapsí až do připojeného neuronu, čímž dojde k jeho excitaci a nervový vzruch se prostřednictvím jeho axonu šíří dále v neuronové síti.

Naopak pokud intenzita nervového vzruchů nepřesáhne hodnotu prahu, dojde k jeho utlumení - neuron ve svém původním stavu a nešíří vzruch dále.

Reakci neuronové sítě mozku si tak vizuálně můžeme představit tak, že v reakci na různé podněty přijímané všemi smysly se budou excitovat odlišné trajektorie spojení neuronů v mozku - a to nám umožňuje realizovat funkci myšlení.

Umělé neuronové sítě se pak snaží tyto funkce přírodních neuronů napodobit a využít k řešení různých problémů.

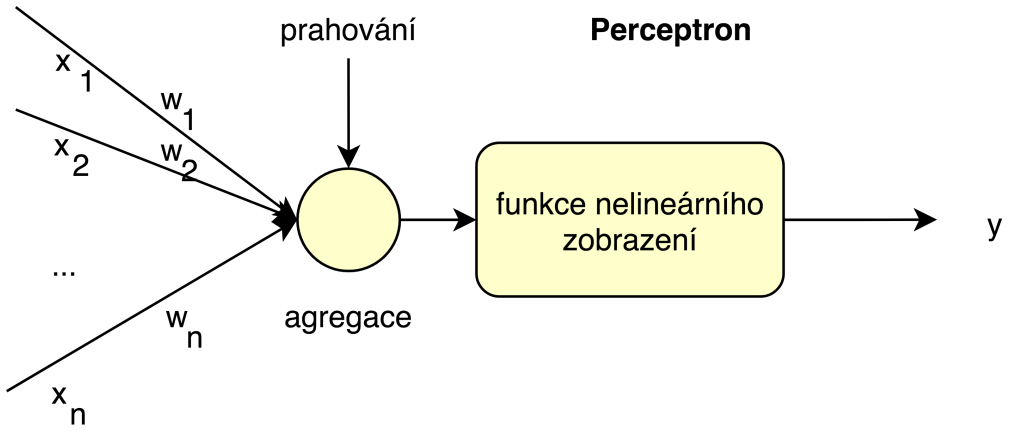
První model neuronu je byl vytvořen v roce 1943 Warrenem McCullochem a Walterem Pittsem [54] - autoři jej nazvali *perceptron*. Perceptron poměrně věrně kopíruje funkce reálného neuronu a přes své



Obrázek 3.2: Snímek neuronu pořízený mikroskopem (převzato z [22])

stáří je i dnes jedním z nejpopulárnějších modelů neuronů.

Funkci perceptronu si můžeme schematicky představit jako na obr. 3.3.



Obrázek 3.3: Schématické znázornění modelu perceptronu

Na normované vstupy x_1 až x_n jsou aplikovány váhy w a výsledek je agregován (sečten) a podroben procesu prahování. Na výslednou hodnotu je aplikována funkce nelineárního zobrazení, která transformuje vstupní hodnotu na výstupní hodnotu y.

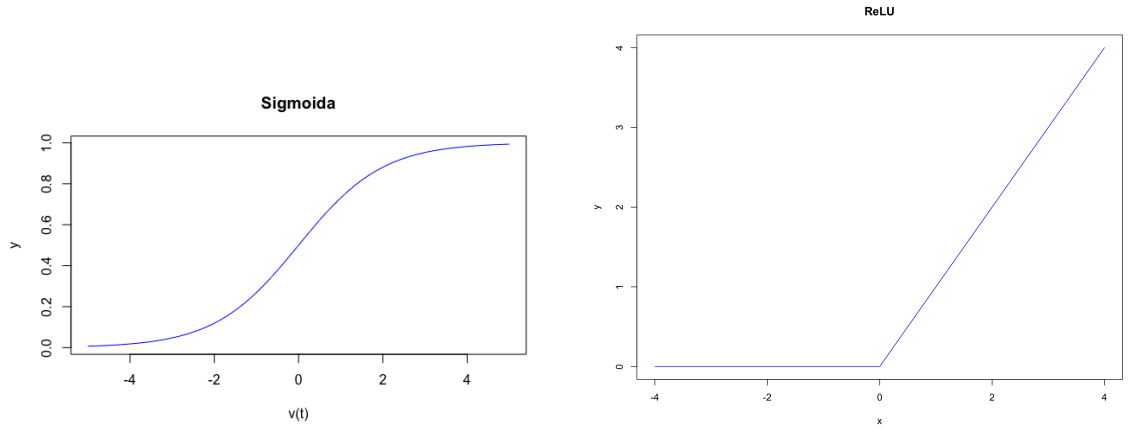
Matematicky lze perceptron vyjádřit následovně. Vážený vstupní signál $z(t)$ (3.1) je agregován do $u(t)$ (3.2).

$$z(t) = x(t) \cdot w(t) \tag{3.1}$$

kde z je vážený vstupní signál, x normovaný vstupní signál (obvykle v intervalu 0-1), w jsou pak váhy.

$$u(t) = \sum_{i=1}^{n} x_i(t) \cdot w_i(t) \tag{3.2}$$

Prahování se provádí tak, že se od agregovaných vstupních signálů $u(t)$ odečte hodnota prahu w_0, viz rovnice (3.3).



(a) sigmoida (SiLU) (b) ReLU

Obrázek 3.4: Jednoduchá neuronová síť pro OCR

$$v(t) = u(t) - w_0 \tag{3.3}$$

Funkci nelineárního zobrazení volíme podle požadavků, které klademe transformaci prahované hodnoty $v(t) \to y$ na hodnotu výstupní. Jednou z nejčastěji používaných funkcí tohoto typu je tzv. *sigmoida*, kterou je možno vypočítat dle vzorce (3.4). Graficky je pak sigmoida znázorněna na obr. 3.4a.

$$y(t) = \frac{1}{1 + e^{-v(t)}} \tag{3.4}$$

Všimněte si, že sigmoida je spojitou funkcí, ale lze uvažovat také o jiných modelech transformace - např. $y(t) = 0$ pro $v(t) < 0$, $y(t) = 1$ pro $v(t) \geq 0$. Taková funkce je nespojitá v $x = 0$ y skočí z 0 na 1.

Sigmoida není ale jedinou funkcí nelineárního zobrazení, existují možná desítky dalších. Sigmoida je totiž relativně výpočetně náročná. Pokud pracujeme pouze s velmi malou neuronovou sítí, pak cena, kterou platíme za její použití je taktéž velmi malá. Při hloubkovém učení ale pracujeme často s neuronovými sítěmi s miliardami parametrů. V takovém případě se každé dílčí zpomalení výpočtu na úrovni neuronů může kumulovat a výrazně prodloužit proces adaptace sítě.

Z tohoto důvodu se často používá funkce Rectified Linear Unit (ReLU). Tuto funkci můžeme matematicky vyjádřit jako (3.5):

$$y = \begin{cases} 0 & \text{pokud } v(t) < 0 \\ v(t) & \text{pokud } v(t) \geq 0 \end{cases} \tag{3.5}$$

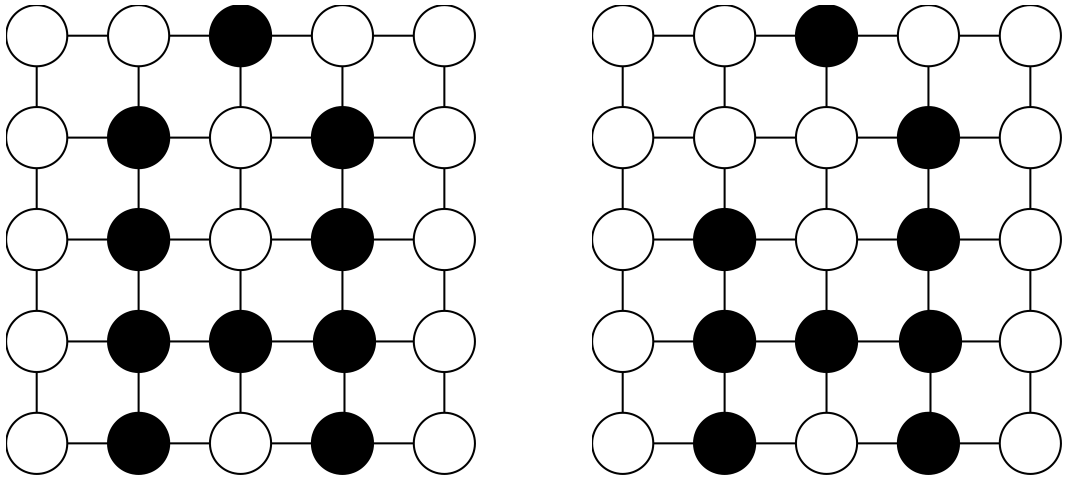
Graficky pak funkce vypadá jako na obr. 3.4b. Tím, že je výpočetně jednoduchá je velmi oblíbenou metodami hloubkového učení.

3.2 Neuronové sítě a jejich adaptace

V minulé podkapitole jsme se seznámili s modelem neuronu, jeden neuron nám ale problém nevyřeší. Řešení je možné pouze při zapojení, obvykle velkého množství, neuronů do sítě. Přitom platí, že čím je síť rozsáhlejší (s čím většího počtu neuronů se skládá) tím dokáže být při řešení problémů přesnější.

Toto je původní myšlenka neuronových sítí, která rozvíjí neuronové sítě tzv. „do šířky“. Neuronová síť je tak spíše plochá, ve smyslu skládající se pouze z několika málo vrstev. Hloubkové učení řetězí vrstvy různých typů za sebou, tedy „do hloubky“. Velikost jednotlivých vrstev je poplatná jejich účelu a může být velmi rozsáhlá.

V průběhu doby se objevovala celá řada typů sítí. V prvopočátcích se pracovalo pouze s jedinou vrstvou v síti. Uvažujme problém: *potřebujeme vyřešit problém rozpoznávání znaků v naskenovaném*



(a) znak A (b) poškozený znak A

Obrázek 3.5: Jednoduchá neuronová síť pro OCR

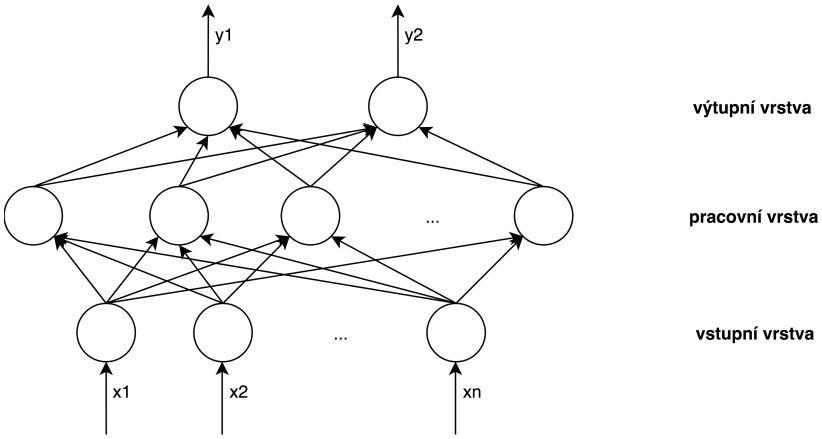
textu. jako vstup nám slouží stav pixelu obrázku rozlišovaného znaku. Uvažujeme přitom 2 stavy reprezentované bílou a černou barvou znaku - černá znamená součást písmene, bílá znamená papír.

Znak A pomocí jednoduché jednovrstevné sítě bychom mohli znázornit třeba jako na obr. 3.5a. Z hlediska vstupů do neuronové sítě lze místo neuronů (na obr. kolečka :-) dosadit hodnotu 0 (bílá) nebo 1 (černá). Výhodou použití neuronové sítě je pak to, že neuronová síť správně určí nejen když bude rozložení „aktivních“ neuronů přesně odpovídat písmenu, ale také případy kdy je reprezentace písmene do určité míry poškozená. Proto např. také 3.5b bude vyhodnoceno jako písmeno A.

Jednovrstevné sítě jsou právě relativně vhodné pro zpracování grafických informací. V moderních sítích, např. pro účely identifikace objektů v prostoru pro samořiditelná auta jsou ale využívány mnohem sofistikovanější sítě, obvykle označované jako *konvoluční*. Jejich výstup je pak integrován do komplexního modelu prostředí, který slouží jako podklad pro rozhodování řídící jednotky automobilu.

Podrobnosti lze nalézt např. v článku NVidie [24].

Více vrstevné neuronové sítě jsou z hlediska použití mnohem univerzálnější. Na obr. 3.6 je znázorněn zjednodušený příklad takové sítě.



Obrázek 3.6: Schéma vícevrstevné sítě

Všimněte se, že vazby mezi neurony v síti 3.6 jsou orientované. Vstupy ve *vstupní vrstvě* x_1 až x_n procházejí v tomto případě jednou *pracovní vrstvou*, výstupní vrstvu v tomto případě tvoří dva neurony y_1 a y_2.

Pracovní vrstvy se někdy označují také jako skryté. To je dáno tím, že vstupy zadáváme a výstupy y pak odečítáme, ale uvnitř neuronová síť funguje jako černá skříňka. Pracovních vrstev v síti může být libovolné množství. Přitom počet neuronů v takové síti není nijak stanoven, vetšinou jich ale

volíme více než je ve vstupní vrstvě. Předpokládáme přitom, že neuronů ve vstupní vrstvě je více než ve vrstvě vrstvě výstupní.

Původní neuronové sítě, pracovaly pouze s těmito typy vrstev. Moderní deep learning ale pracuje s mnohem větším počtem typů vrstev, viz kapitola *Hloubkové učení*.

Volba architektury sítě tak není exaktně daná ve smyslu počtu vrstev a počtu neuronů v nich obsažených. Dokonce je možno zvolit řadu přístupů pro kódování vstupních údajů na vstupní vrstvě a jejich odečítání na vrstvě výstupní. Není tak od věci experimentovat s různou velikostí sítě apod. a porovnávat jakých výsledků bude dosaženo.

Všimněte si také, že strukturálně je každý neuron předchozí vrstvy připojen na každý neuron vrstvy následující. Prakticky to znamená, že s rostoucím počtem neuronů v síti roste počet vazeb v sítích tohoto typů exponenciálně. To pak na nás může klást jistá omezení z hlediska „vypočítatelnosti/adaptovatelnosti“ takových sítí. Složitost roste exponenciálně, ale výkon výpočetní techniky jsme obvykle schopni škálovat pouze lineárně.

To je signifikantní, protože právě spojení mezi neurony představují *parametry* modelu, které upravujeme v rámci procesu adaptace. Konkrétně v procesu adaptace upravujeme váhové koeficienty přináležejícím k těmto spojením.

Počet parametrů modelu je základním ukazatelem, který se používá pro charakteristiku modelu. Mělo by platit, že modely s větším počtem parametrů mají lepší potenciál vyřešit problém. Ale pozor tento parametr sám o sobě nezaručuje úspěch. O úspěchu rozhoduje:

- celková architektura,
- počet parametrů a
- kvalita vstupních dat

Zkusme zrealizovat jednoduchou demonstrační síť pro predikci průběhu funkce, jako alternativu *lineární regrese*, se kterou jste se mohli setkat např. v předmětu *Statistika*.

Mějme jednoduchou funkci $y = x^2$. Řekněme, že trénovací množinu (viz tab. 3.1) budeme mít hodnoty $x \in < 0; 10 >$ po celých číslech.

Tabulka 3.1: Trénovací množina funkce $y = x^2$

input	output
0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

Listing 3.1: Použití knihovky neural net pro natrénování funkce druhé mocniny

```
library("neuralnet")
mydata=read.csv("cesta/nn_fce_2mocnina.csv", sep=",", header=TRUE)
#tréning neuronové sítě
model=neuralnet(formula=output~input, data=mydata, hidden=10, threshold=0.01)
model_lm = lm(output~input, data=mydata) #pro srovnání lineární regrese
plot(model) #vykreslení vrstev sítě v (síťový graf)
#porovnání dat - skutečná a predikce
final_output=cbind(mydata$input, mydata$output,
       as.data.frame(model$net.result),
       as.data.frame(model_lm$fitted.values))
colnames(final_output) = c("Input", "Expeceted output", "NN Output", "LM Output")
plot(final_output$‘Expeceted output‘, col="red", main="Predikce vs realita",
       xlab="cisla", ylab="druha mocnina")
```

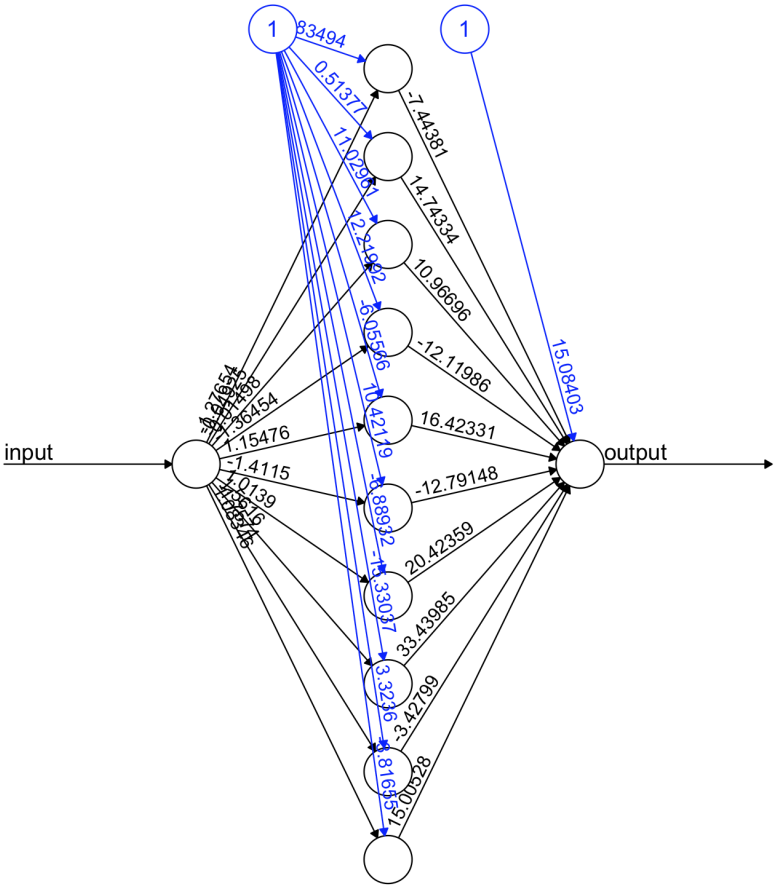
```
14 lines(final_output$‘NN Output‘, col="blue")
15 lines(final_output$‘LM Output‘, col="green")
16 legend(1,95, legend=c("skutecne hodnoty", "predikce NN", "predikce LM"),
17        col=c("red", "blue", "green"), lty=1:3, cex=0.8)
```

Pro demonstraci použijeme toolkit *neuralnet* [38]. Který umožňuje realizaci jednoduchých modelů neuronových sítí. (Pro pokročilejší práci je ale vhodnější použít některé pokročilejší toolkity.)

Model je načten ze souboru CSV, a natrénován pomocí funkce *neuralnet*. Počet neuronů vstupní a výstupní vrstvy je dán strukturou zvoleného modelu (parametr *formula*). V našem případě, bude mít tedy jede vstupní a jeden výstupní neuron. Počet neuronů pracovní vrstvy jsme nastavili parametrem *hidden* na 10. Pomocí parametru *threshold* nastavujeme požadovanou přesnost modelu.

Prosím všimněte si, že chyba nemůže být nikdy nastavena na hodnotu nula - důvodem je tzv. problém *přeučení* neuronové sítě. K problému přeučení se ale ještě vrátíme.

Struktura neuronové sítě je pak vykreslena pomocí funkce plot, viz obr. 3.7.



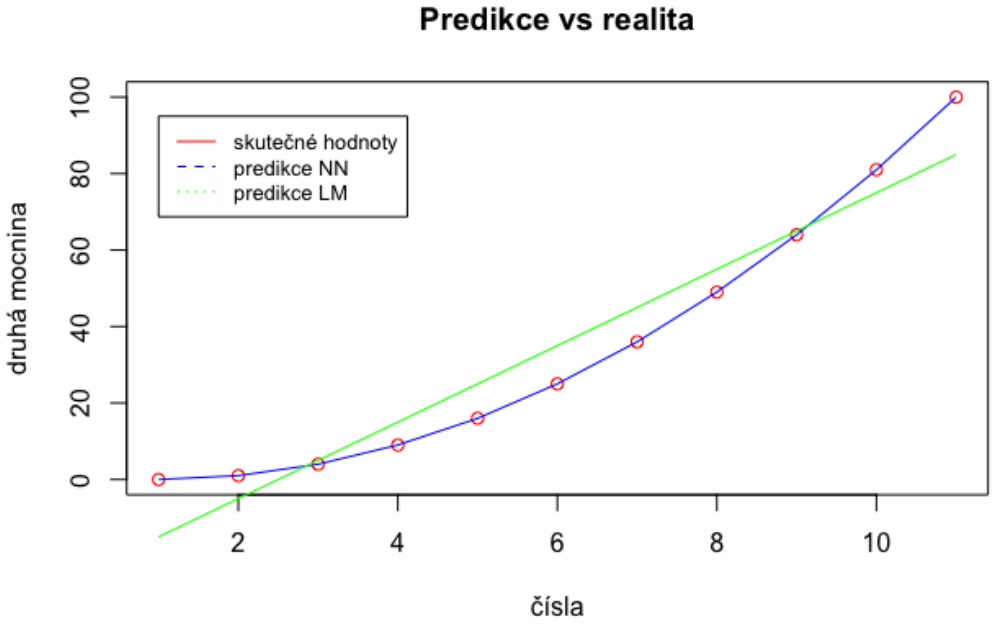
Obrázek 3.7: Neuronová síť pro predikci hodnot funkce $y = x^2$

Zbývající část skriptu R pak slouží pro přípravu porovnání modelu lineární regrese, neuronové sítě a skutečnosti a vykreslení do grafu na obr. 3.8.

Porovnání z obr. 3.8 jasně ukazuje, že model neuronovou sítí je přesnější než námi zvolený model lineární regrese. K tomu je potřeba dodat, že by tomu tak být nemuselo. V našem případě není důvod, abychom nebyli schopni odvodit regresní model s podobnou přesností.

Místo toho, ale tuto situaci použijeme pro zdůvodnění jedné z nejsilnějších stránek použití neuronové sítě. Lineární model totiž po nás požadoval přijmout jisté předpoklady o vzhledu modelu. V našem případě, že bude vypadat následovně: $y = a + bx$, kde koeficienty a a b budou odvozeny.

My samozřejmě víme, že správně model měl být $y = x^2$, ale informace tohoto typu obvykle není dostupná - konečně, pokud víme, jak vypadá funkce, není potřeba se snažit ji odvodit.



Obrázek 3.8: Funkce $y = x^2$ - srovnání modelu neuronovou sítí a lineární regresí

Neuronová síť proti tomu po nás nežádala přijmout žádný předpoklad o fungování modelu. Specifikovali jsme pouze základní strukturu neuronové sítě a model se odvodil sám. Ve skutečnosti mohla být struktura neuronové sítě podstatně jednodušší - mohlo stačit možná 5 možná ještě méně neuronů. Můžete sami vyzkoušet kolik neuronů z pracovní vrstvy můžete odebrat aniž by to zanechalo následky na přesnosti modelu.

Jak se tedy vůbec neuronová síť adaptuje?

Pro adaptaci v neuronových sítích, které jsme popisovali výše se většinou používá tzv. *učení s učitelem*. To znamená, že adaptačnímu algoritmu se předkládá trénovací množina ve formátu vstup → výstup, což umožňuje pro neuronovou síť porovnáním s jejím skutečným výstupem vypočítat chybu. Tato informace je pak použita pro další iteraci adaptace sítě.

Představit si to lze tak, že na začátku procesu adaptace je všem vazbám mezi neurony v síti přiřazena náhodná hodnota váhy w_n. Adaptací pak postupně tyto váhy měníme s cílem konvergovat postupně ke specifikované chybě.

Nejpoužívanější metodou pro tento účel je tzv. *backpropagation* - česky metoda *zpětného šíření chyb*. Matematicky je metoda založena na výpočtu globální chyby GE jako rozdílu očekávaného a skutečného výstupu sítě, viz (3.6).

$$GE = \sum_{i=1}^{n} (y_i - d_i)^2 \tag{3.6}$$

Kde GE je globální chyba (global error), y je očekávaný výsledek a d skutečný výsledek sítě.

Proč součet čtverců?

Podívejte se na vzorec 3.6 - proč myslíte, že je pro výpočet GE použit součet čtverců a ne např. $GE = \sum y_i - d_i$?

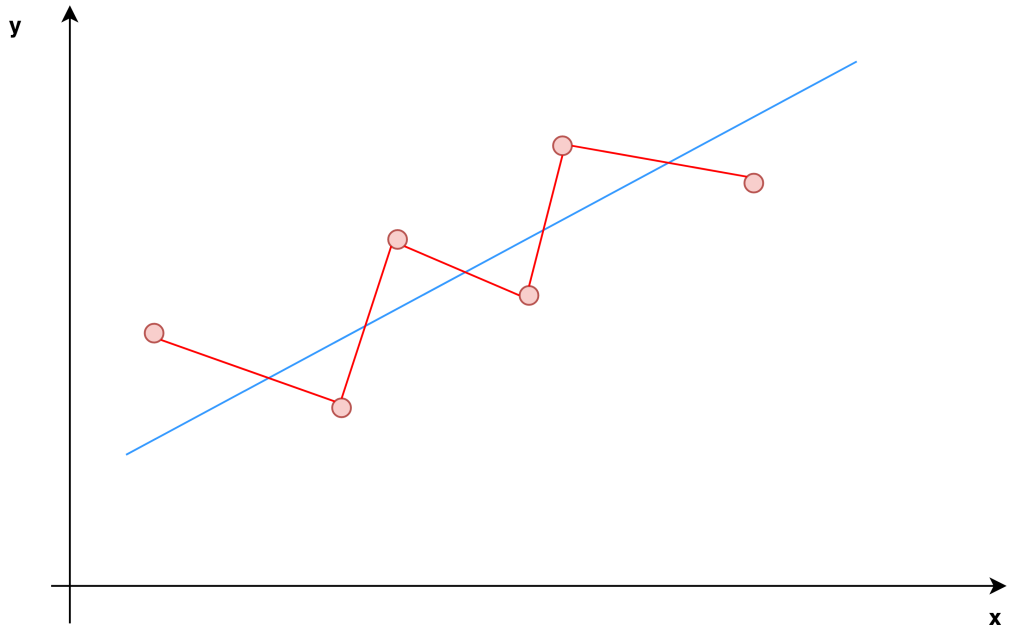
Z *Matematiky* víme :-), že k nalezení minima funkce potřebujeme tuto funkci derivovat (3.7).

$$\Delta w = -\eta \frac{\partial GE}{\partial w} \tag{3.7}$$

Úkolem hodnoty η je pak zajistit, aby Δw bylo velmi malé. Zdůvodnit si to můžeme tak, že při velkých změnách riskujeme, že budeme oscilovat okolo očekávané hodnoty, aniž bychom k ní konvergovali. Volbou $\eta = 0,1$ nebo $\eta = 0,01$ nebo dokonce menší zajistíme, že proces adaptace bude pomalu směřovat k cíli. To ... *pomalu* je bohužel pro rozsáhlejší sítě potřeba zdůraznit.

Minimalizace globální chyby má svá omezení. U neuronových sítí riskujeme u příliš malých chyb možnost *přeučení*. Přeučená neuronová síť u případů z trénovací množiny dosahuje minimální chyby, ale u příkladů, které nejsou v trénovací množině je chyba naopak výrazně vyšší.

Graficky si to můžeme představit podobně jako na obr. 3.9.



Obrázek 3.9: Přeučený model

Vstupem pro modely vizualizované na obr. 3.9 jsou body (na obr. červená kolečka). Pro tyto body byly vytvořeny dva modely reprezentované modrou a červenou křivkou. Všimněte si, že červená křivka přesně kopíruje zadané body, zatímco modrá jde mezi nimi.

V tomto případě je modrá křivka přesnější, protože rozmístění bodů okolo této linie bylo způsobeno náhodnou chybou měření. Taková chyba se v reálu objevuje prakticky pokaždé. Přílišnou snahou o přesnost, která vedla k odvození červeného modelu, jsme tak ve skutečnosti zvětšili výrazně chybu, s jakou model pracuje.

Z tohoto důvodu je potřeba se smířit s jistou nevyhnutelnou chybou, kterou model bude obsahovat vždy a která je vynucena fyzickými charakteristikami způsobu jakým byla data pořízena.

Problém přeučení není problém čistě jen neuronových sítí - podobný problém mají ale také další metody používané v dataminingu, např. rozhodovací stromy - i ty lze přeučit. Pro rozhodovací strom spočívá řešení v tzv. „prořezání stromu“. Postupně odebíráme listové uzly a sledujeme, jestli dochází ke zvyšování přesnosti řešení.

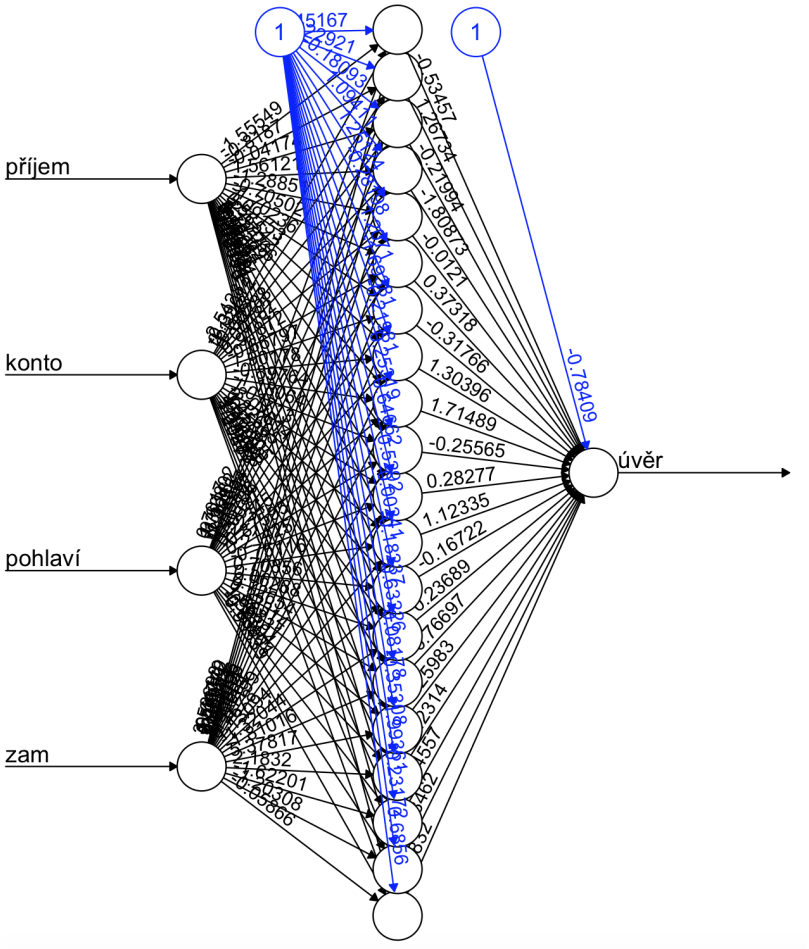
Řešení klasifikačního problému probíhá vlastně stejně jako probíhalo odvození modelu funkce $y = x^2$. Pro demonstraci můžeme použít opět příklad klientů banky, viz tab. **??**.

Listing 3.2: Klasifikace klientů banky podle bonity

```
1 library("neuralnet")
2 mydata=read.csv("cesta/klienti.csv", sep=",", header=TRUE)
3 model=neuralnet(formula=uver~prijem+konto+pohlavi+zam,
```

```
        data=mydata, hidden=20, threshold=0.01)
plot(model)
final_output=cbind(mydata$uver, as.data.frame(model$net.result))
colnames(final_output) = c("ocekavano", "spocteno")
final_output
```

Pouze v tomto případě jsem do pracovní vrstvy zvolil 20 neuronů. Vizuálně neuronová síť vypadá jako na obr. 3.10.



Obrázek 3.10: Klasifikace klientů banky neuronovou sítí

Modely tvořené desítkami neuronů je možno na moderních počítačích počítat prakticky v reálném čase. 20 neuronů skryté vrstvy je tak pravděpodobně nadsazený počet, ale nějaké výrazné úspory času bychom z hlediska výpočtu nedosáhli.

Výsledek sítě:

	očekáváno	spočteno
1	1	1.0019334196327
2	1	1.0069034997006
3	0	-0.0130942067801
4	1	0.9825741850726
5	1	0.9861262522598
6	0	-0.0033695338052
7	1	1.0086557396241
8	1	0.9899790546700
9	0	-0.0002532946061

```
10          1   0.9844730618735
11          0   0.0380269653583
12          1   1.0135488795748
```

Všimněte si, že neuronová sít nespočte nikdy číslo zcela přesně, takže 1 není nikdy zcela přesně 1, ale něco mezi 0,98 a 1,1 a podobně je to s nulou. Všimněte si, že přesto je výsledek poměrně přesný.

Realizace příkladů v R
Pokud jste dosud ještě nerealizovali příklady v R, teď je k tomu vhodná doba.

Úpravy skriptu modelu funkce $y = x^2$
Zajisté jste si všimli, že naše ověření kvality modelů není úplně korektní - pro ověření používáme trénovací množinu. Vzhledem k tomu, že známe funkci, kterou se modelem snažíme pokrýt můžeme jednoduše vygenerovat validační množinu novou, např. 0,5 → 0,25, 1,5 → 2,25, ... a provést ověření zcela konkrétní ... vzhůru do toho!

3.3 Aplikace neuronových sítí v bezpečnostních oborech

Aplikace neuronových sítí jsou extrémně široké. Kromě běžné analytiky dat, kde neuronové sítě mohou do určité míry nahradit regresní modely používané statistikou a které se používají prakticky ve všech oborech lidské činnosti, mají neuronové sítě zásadní význam pro řešení klasifikačních problémů.

Řada problémů v bezpečnostních oborech má právě charakter klasifikačního problému. Do této oblasti může patřit například zjišťování příčin chyb na základě dat charakterizujících vznik a projevy chyby.

Do této oblasti patří taktéž některé úlohy zpracování obrazu - např. ve smyslu identifikace předmětů na nich. Příklady problémů tohoto typu mohou být následující:

- identifikace osob na fotografii
- identifikace podezřelých předmětů na fotografii nebo video streamu z kamerového systému např. na letišti
- identifikace podezřelého chování
- detekce neoprávněného vstupu nebo pohybu neuatorizovaných osob v určitém prostoru
- identifikace nebezpečné látky pomocí zjištění UN kódu pro identifikace vozidel v tunelu převážející látky, které by tunelem neměly být transportovány
- a řada dalších

Neuronové sítě mají ale mnohá další obecná využití. Moderní Text To Speech (TTS) systémy jsou založeny na neuronových sítích. Schopnosti takových aplikaci si můžete konečně otestovat také v domácích podmínkách pomocí TTS společnosti Google založené na technologii WaveNet: https://cloud.google.com/text-to-speech/ [36].

Takové systémy jsou schopné generovat realisticky znějící hlas schopný přečíst zprávu v jakémkoliv podporovaném jazyce. Mimochodem čeština je alespoň v případě TTS od Google podporovaným jazykem.

Z hlediska bezpečnostních aplikací lze uvažovat o možnosti zpřístupnění bezpečnostně orientovaných informací v textové podobě zrakově postiženým osobám. Podobně by např. v budoucnu v systému jednotného varování by nemusela být použita předem připravená nahrávka (např. *zkouška sírén, zkouška sirén, ...*) nebo přímá řeč živého člověka. Automatizovaně generovaná mluva vyžaduje pouze textový vstup (informaci která má být přečtena) a výsledná mluva je pak obvykle jasná a

srozumitelná. Textový vstup je jednoduše přenositelný (vyžaduje minimální přenosovou kapacitu). Automat se pak neunaví, nepůsobí na něj stres apod.

Použití takových technologií pro varování by mělo samozřejmě i svá úskalí - v současnosti jsou např. tyto technologie závislá na infrastruktuře v „cloudu“, víme však, že při mimořádných událostech velkého rozsahu, jako jsou např. rozsáhlé blackouty, taková infrastruktura nemusí být dostupná.

S vývojem IT ale lze předpokládat, že výkon potřebný pro provoz takového systému nemusí nutně doménou velkých datových center napořád. V současnosti jsme např. svědky přenosu některých funkcí virtuálních asistentů na mobilní telefony nebo počítače, popř. notebooky, ačkoliv v minulosti pro svou činnost vyžadovali výhradně dostupnost cloudu.

Lze tedy předpokládat, že rozsah lokálně dostupných technologií tohoto typu se bude dále rozšiřovat a to možná dokonce rychle.

Uvažovat lze i o kombinaci více technologií dohromady. Uživatel tak může použití fotoaparát mobilního telefonu pro nasnímání textu v neznámém jazyce a použít umělou inteligenci pro detekci jazyka a jeho přeložení do jazyka, kterému uživatel rozumí. Ucelená řešení jdoucí tímto směrem představil např. Google v nových verzích svého operačního systému Android. Tyto schopnosti jsou dostupné také samostatně, např. v rámci aplikací Google Lens a Google Translate a existuje také celá řada řešení do dalších vývojářů.

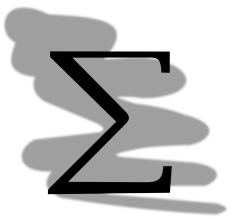
Výsledek je pak možné nechat si přečíst pomocí TTS, pokud je to tedy potřeba. Všechny výše uvedené aplikace a technologie přitom pro svou funkci extenzivně využívají neuronových sítí.

V neposlední řadě lze používat neuronovou síť pro transformaci vstupů. Transformací přitom rozumíme např. odstraňování šumu z fotografií nebo strojové zpracování obrazu - doplnění barvy, zvýšení rozlišení. Populární aplikace a filtry pro přidávání např. vousů, vlasů, vytvoření mladší nebo starší verze vzhledu osoby, ale tak tzv. zkrášlovací filtry bývají často založeny na neuronových sítích.

Pro bezpečnostní obory mohou být v takovém případě použití pro vytvoření lepších verzí identikitů (kriminalistika) nebo zlepšení obrazu pro účel identifikace příčiny/průběhu nějakého děje apod. Obdobné technologie jsou např. využívány v některých počítačových hrách pro zajištění plynulého hraní v rozlišení 4K i na relativně málo výkonných grafických kartách. Společnost NVidia takovou technologii nazývá Deep Learning Super-Sampling (DLSS) a podporuje ji na grafických kartách řad RTX.

Z aplikací výše je patrné, že možnosti použití jsou skutečně široké. V současnosti jsme přitom z hlediska našich znalostí o možnostech aplikace v expanzivní fázi. Možnosti nový aplikací se tak objevují takřka denně.v Limity přitom v současnosti nejsou známé.

To je také důvodem toho, že problematiku neuronových sítí zatím neopustíme a vyzkoušíme si také některé aplikace a to už v následující kapitole.



Shrnutí kapitoly

Neuronové sítě jsou v současnosti jednou z nejpopulárnějších a také nejuniverzálnějších metod pro provádění analýz všeho druhu. Neuronové sítě jsou tvořeny neurony a váženými vazbami mezi nimi. Tento typ modelů se tak snaží napodobovat funkci mozku vyšších organismů.

Univerzálnější vícevrstevné sítě obsahují, vstupní, výstupní a jednu nebo více pracovních vrstev. Síť pak funguje jako černá skříňka - hodnoty můžeme odečítat na vstupu a výstupu, ale dovnitř již „nevidíme“.

Pro správné fungování se musí síť tzv. adaptovat. Nejpoužívanější metodou adaptace je pak *backpropagation*, která využívá kvantifikace chyby sítě k odvození změny vah vazeb mezi neurony, které k této chybě vedly.

Neuronové sítě jsou primárně použitelné pro řešení problémů klasifikace, popř. predikce.

Kontrolní otázky

1. Vysvětlete funkci perceptronu.
2. kolik skrytých vrstev musí být ve vícevrstvé síti?
3. vysvětlete backpropagation

Správné odpovědi

1. Perceptron umožňuje prahované agregované vstupy x transformovat funkcí nelineárního zobrazení na výstup y.
2. minimálně jedna.
3. metoda učení s učitelem - založená na zpětné šíření chyb v síti. (váhy v síti se upravují směrem od výstupní vrstvy zpět ke vstupní).

Kapitola 4

Neuronové sítě - cesta k hloubkovému učení

Náhled kapitoly
Tato kapitola je postavena trochu jinak než ty ostatní. Její úspěšné zvládnutí předpokládá, že jste úspěšně nastudovali základy neuronových sítí z předchozí kapitoly. V této kapitole se zaměříme spíše na postupy a metody pro využití neuronových sítí pro řešení praktických problémů. Text je proto silně založen na příkladech a předkládá, že s nimi budete sami aktivně pracovat - *nestačí tedy kapitolu přečíst a jít dál.*

Po prostudování této kapitoly budete
vědět

- co je to hloubkové učení (deep learning) a jaká má úskalí
- jak řešit některé typy problémů pomocí neuronových sítí

Čas pro studium
Prostudování této kapitolu může být poměrně náročné - neomezujte se. Vyzkoušejte si příklady z této kapitoly a pokuste se je modifikovat pro řešení dalších problémů.
Celkový odhad délky studia této kapitoly jsme proto neprováděli.

Tip
Zprvu se Vám může zdát, že to, co se v této kapitole probírá (a co jste slyšeli na přednáškách a na cvičení) je naprosto nepochopitelné a nebudete schopni to sami použít. Ve skutečnosti tomu tak není. Ačkoliv je problematika hloubkového učení **jako celek** skutečně složitá a lze se jí zabývat celý život (a řada lidí to také dělá), poměrně velkých úspěchů lze dosáhnout v této oblasti s minimem úsilí.

Při studiu se proto zaměřte na pochopení jednotlivých kroků představovaných postupů - *samostatně* a dívejte se na ně jako na stavební kameny, popř. šablony budoucích řešení Vaši problémů (analytických potřeb). Vycházejte z předpřipravených postupů a příkladů a upravujte je pro své vlastní potřeby.

4.1 Knihovna neuralnet - analýza obrazové informace

Jako vhodný příklad zpracování obrazové informace je neuronová síť, která by nám umožnila rozlišit obrázky ručně psaných číslic. Do určité míry tak navazujeme na příklad teoreticky diskutovaný v

předchozí kapitole a graficky znázorněný na obr. 3.5. K tomuto účelu použijeme veřejně dostupný MNIST dataset obsahující okolo 40 000 ručně zapsaných cifer.

Příklad vzhledu zaznamenaných číslic je dostupný na obr. 4.1.

Obrázek 4.1: Příklady jednotlivých cifer z datasetu MNIST (převzato z [76])

Všimněte si, že jednotlivé cifry na obr. 4.1 se liší podle toho, kdo je zapsal. Tato variabilita je něco s čím si lidský intelekt bez problému poradí, ale počítač ve skutečnosti nechápe, na co se kouká a tak za normálních okolností s interpretací takových drobných rozdílů může mít problém.

Tento problém vyřešíme právě nasazením neuronových sítí.

Nejprve začneme tím, že stáhneme dataset. K tomuto účelu doporučujeme použít odkaz [50] `apache-mxnet.s3-accelerate.dualstack.amazonaws.com/R/data/mnist_csv.zip`. Jedná se o upravenou verzi datasetu distribuovanou jako součást open source analytického nástroje pro hloubkové učení *MXNet* [20]. Alternativně je možno stáhnout dataset ze stránek kurzu na Moodle VŠB `https://lms.vsb.cz`.

Dataset je rozdělen do dvou souborů a to test.csv a train.csv určených pro adaptaci (trénování, train.csv) a ověření validity odvozeného modelu (test.csv). Vzhledem k tomu, že budeme primárně pro tréning používat CPU počítače budeme muset stejně množství zpracovávaných údajů omezit a tak si ze stávající trénovací množiny vytvoříme menší trénovací a validační datové soubory.

Co do struktury oproti originální verzi [49] je datová sada připravena ve formátu CSV, kde každé číslo tvoří jeden řádek tabulky a každý pixel obrázku má vlastní sloupeček. Hodnota pixelu je vždy celé číslo v intervalu $< 0; 255 >$ kde 255 odpovídá černé barvě a hodnota 0 bílé barvě (obrázek je tak ve stupních šedi).

Každý obrázek tvoří rastr $28 \cdot 28px = 784px$. Formát dat je tak následující:

label; px_11; px_12; ...; px_2828

Label představuje číslo v intervalu $< 0; 9 >$ které odpovídá zaznamenanému obrázku, *px_11* odpovídá hodnotě pixelu na prvním řádku a v prvním sloupci.

Soubor je již připraven pro nasazení analytických metod. To úplně neodpovídá způsobu, který bychom museli použít vyřešili reálný problém. Dostat data do použitelné podoby, z hlediska nasazení metod je považováno často za obtížnější než samotný model neuronové sítě a její adaptace.

Uvažme proces přípravy čísel, se kterými pracujeme. Účelem, pro který dataset vznikl, bylo vytvořit OCR systém pro automatizované zpracování poštovních směrovacích čísel. Z tohoto pohledu pro získání finálního datového souboru museli analytici:

1. shromáždění obálek

2. naskenování obálek
3. řešení orientace obálky (např. vzhůru nohama)
4. identifikace PSČ
5. identifikace cifer PSČ
6. sjednocení podoby cifer (vycentrování cifry na obr., sjednocení rozměru, převod do stupňů šedi, ...)

Naštěstí pro nás, analytici výše uvedené kroky již zrealizovali za nás. Některé zdroje uvádějí, že příprava dat může zabrat i 80 % z celkového času analýzy.

Pro naše pokusy nejprve provedeme načtení datasetu do systému R. V našem případě je možno dataset ještě zjednodušit. Cifry na obrázcích v datovém souboru jsou totiž ohraničeny jedním volným pixelem. Jelikož tyto pixely ohraničení nepřispívají k „vysvětlení“ problému, lze je z datového souboru vyřadit.

Zjednodušení by v takovém případě bylo poměrně výrazné, protože každý pixel, kterého se budeme moci zbavit, představuje jeden sloupec datového souboru. Při velikosti obr. 28 x 28 se tak jedná o $28 \cdot 2 + 26 \cdot 2 = 108$ sloupců ze 784. V našem případě, ale soubor ponecháme tak jak je.

Náš první pokus bude realizován pomocí knihovny *neuralnet*, která není optimalizovaná pro dataminingové úlohy. V zásadě se tedy jedná o přímou aplikaci poznatků z předchozí kapitoly. Absence optimalizace by ale způsobila, že adaptace neuronové sítě by trvala neúměrně dlouho.

Z tohoto důvodu budeme redukovat problém do podoby tzv. *binární klasifikace*. Ze souboru vybereme pouze dvě čísla, řekněme, že 3 a 8, jelikož svým způsobem vypadají podobně (alternativní dobrou volbou by byl výběr čísel 5 a 6). Takto trénovací množinu silně zredukujeme a výsledek tak dostaneme velmi rychle, nikoliv však v reálném čase.

Jenom pro upřesnění - všimněte si ve výpisu níže použití výsledku funkce *sample*. Tato funkce pouze identifikuje čísla řádků tabulky výběru. Jedná se tedy o index vybraných řádků. Tento index (tr_idx) je pak použit pro odlišení řádků, které nebyly vybrány [-tr_idx,]

Jelikož se jedná o binární klasifikační model, hodnotu prediktoru (label) převedeme na 0 a 1, kde 0 reprezentuje 3 a 1 reprezentuje 8.

Listing 4.1: Pužití knihovny neural net pro natrénování a validování rozpoznávání čísel 3 a 8 z datasetu MNIST v R

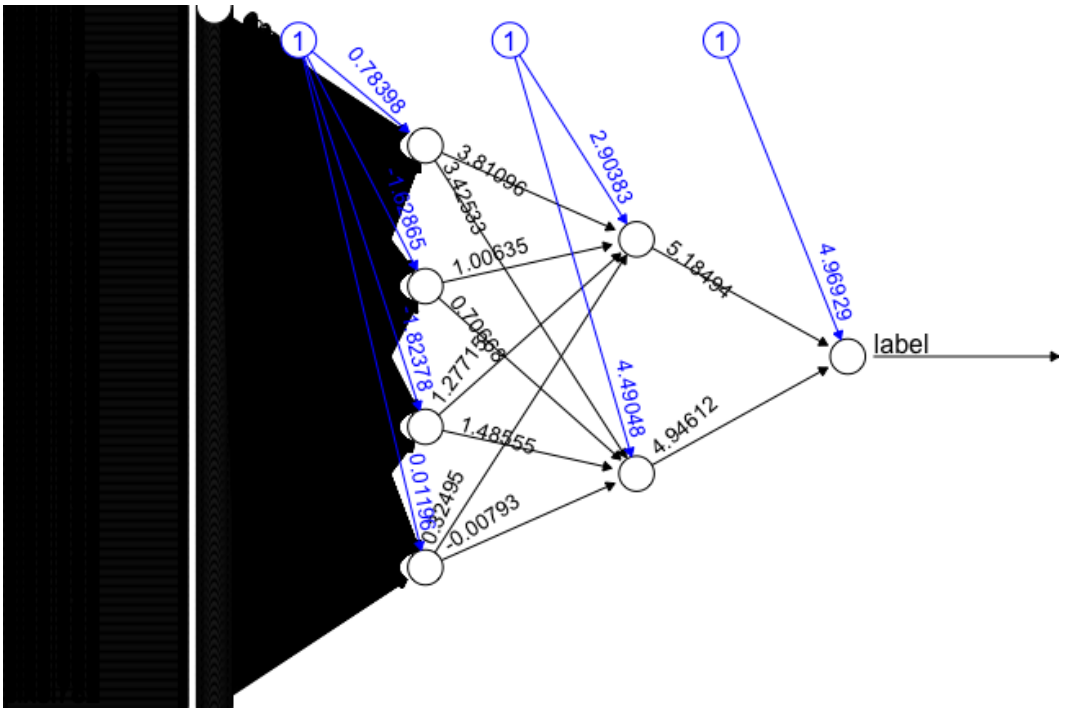
```
library(neuralnet)
cisla = read.csv("train.csv")
cisla38 = cisla[(cisla$label == 3) | (cisla$label == 8),]
cisla38$label[cisla38$label == 3] = 0
cisla38$label[cisla38$label == 8] = 1
tr_idx = sample(nrow(cisla38), nrow(cisla38) * 0.8)
train = cisla38[tr_idx, ]
valid = cisla38[-tr_idx, ]
jmena_sloupcu = names(cisla)
f = as.formula(paste("label~", paste(jmena_sloupcu[!jmena_sloupcu %in% "label"],
        collapse = "+")))
nn = neuralnet(f, data = train, hidden = c(4, 2), linear.output = F)
plot(nn)
valid = as.data.frame(valid)
pred = compute(nn, valid[,2:ncol(valid)])
pred = ifelse(pred$net.result > 0.5, "1", "0")
t = table(valid$label, pred)
acc = round(100 * sum(diag(t))/sum(t), 2)
print(sprintf("presnost: %1.2f", acc))
```

K tomu je však potřeba dodat, že výše uvedený model pozbyl univerzálnosti, bude schopen pouze rozpoznávat mezi čísly 3 a 8. Pokud takovému modelu předložíme jakákoliv jiná čísla bude se stále snažit identifikovat zda se jedná o číslo 3 nebo 8, ačkoliv to nemá smysl.

Trénovací množina proto vždy determinuje způsob, jakým bude možné výsledný model dále použít.

Zvolená architektura neuronové sítě je znázorněna na obr. 4.2. Architekruta sítě je 784 neuromů ve vstupní vrstvě (neuronů je tolik že na obr 4.2 splývají do jednotné černé barvy a jednotlivé neurony

tak na obr. není možno rozlišit), 4 a 2 neurony ve skrytých vrstvách a konečně jediný neuron ve vrstvě výstupní.



Obrázek 4.2: Neuronová síť pro predikci v knihovně neuralnet pro rozlišení mezi číslicemi 3 a 8

Použití neuronové sítě pro predikce je relativně jednoduché - budeme aplikovat funkci *compute* odvozeného modelu na validační množině. Výsledek výpočtu je pak dostupný ve sloupci net.result v proměnné, do které jsme výsledek přiřadili.

Přesnost predikce byla v mém případě 94,53 %, což není špatný výsledek. Vaše přesnost bude jiná, jelikož výběr dat do trénovací množiny je náhodný.

Alternativní pohled na přesnost poskytuje tzv. *confusion matice* (tabulka). V našem příkladě je tato matice dostupná v proměnné t a je také dostupná v tab. 4.1.

Tabulka 4.1: Confusion matice binární klasifikace čísel 3 a 8 adaptovaných knihovnou Neuralnet

	0	**1**
0	805	54
1	38	786

Tato matice porovnává predikci a skutečný stav. Hodnoty na diagonále pak reprezentují shodu - tedy např. predikce 0, skutečnost 0. Hodnoty mimo diagonálu pak ukazují chyby.

Jiné číslovky
Zkuste výše uvedený příklad binární klasifikace čísel 3 a 8 upravit tak, aby rozlišoval čísla 5 a 6. Jaké přesnosti se Vám podařilo dosáhnout?

Výše uvedený příklad nelze ve skutečnosti považovat za hloubkové učení - celková hloubka použité sítě je dána dvěma skrytými vrstvami. Hloubkové učení ale funguje jinak - zkusme se na něj proto podrobněji zaměřit. Demonstrace přitom budeme realizovat na stejné datové sadě. Z tohoto důvodu by bylo dobré, abyste příklad z této kapitoly prakticky vyzkoušeli a pochopili jeho funkci.

4.2 Knihovna MXNet

Příklad řešený pomocí knihovny Neuralnet byl řešený pomocí tzv. *imperativného programu*. To znamená, že jsme systému R řekli přesně jakým způsobem má realizovat výpočet. Výhodou tohoto přístupu je, že z výpisu získáme naprosto přesnou představu o způsobu výpočtu. Nevýhodou tohoto přístupu ale naopak je, že neposkytuje prakticky žádný prostor pro optimalizaci výpočtu. Výpočetní procedura tak prakticky není paralelizovaná a výpočet pak probíhá déle, než by odpovídalo požitému hardware.

To je problém, protože pomocí neuronových sítí můžeme chtít zpracovávat gigantické objemy dat a v takovém případě výše uvedené zásadní překážkou úspěchu. Z tohoto důvodu knihovny používané pro hloubkové učení imperativní programování opustily a nahradily jej tzv. *symbolickým programováním*.

Architekt neuronové sítě tuto síť specifikuje pomocí tzv. symbolů, ale vnitřní organizaci výpočtu si řídí systém sám včetně možností paralelního výpočtu sítě nebo její části. To otevírá řadu možností pro optimalizaci výpočtu, které jsou realizovány automatizovaně bez nutnosti naší aktivní účasti.

Odpovědnost za tyto optimalizace má tvůrce použité knihovny. My, jako uživatelé této knihovny, se pak můžeme soustředit na problém, který řešíme.

V současnosti existuje celá řada knihoven pro hloubkové učení. MY se zaměříme na knihovnu MXNet [20], která je jednou z nejpoužívanějších a zároveň relativní jednoduchost jejího použití ji činí vhodným kandidátem pro demonstraci postupů hloubkového učení.

Z tohoto pohledu je velmi populární taktéž knihovna Keras [17]. Tato knihovna ale funguje trošičku jinak - jedná se totiž o nadstavbu nad dalšími metodami hloubkového učení jako je třeba TensorFlow [37] společnosti Google, Microsoft Congnitive Toolkit [55] a řada dalších včetně knihovny MXNet.

Uživatel tak může specifikovat, která knihovna se má pro adaptaci a vybavovací fázi použití neuronové sítě použít a v některých případech mohou dokonce odvozené modely mezi těmito knihovnami migrovat. Keras tak představuje vlastně abstrakční vrstvu nad dalšími knihovnami.

Pro zjednodušení se ale v tomto textu zaměříme na přímé použití knihovny MXNet.

4.2.1 Instalace MXNet na macOS

MXNet je nejprve nutné na Vašem počítači nainstalovat. Knihovna tedy nefunguje čistě uvnitř výpočetního prostředí R. Postup instalace se liší podle operačního systému, který používáte.

Pro **macOS** je potřeba nejprve zprovoznit správce balíků Homebrew [2]. To můžete udělat jednoduše z příkazové řádky:

Listing 4.2: Instalace Homebrew ma macOS

```
1 /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/
2 Homebrew/install/master/install)"
```

Výše uvedený příkaz zadávejte prosím do terminálu na jednom řádku (mě se ve skriptech bohužel na jeden řádek nevlezl).

Příkaz spustí průvodce, který Vás provede instalací Homebrew na Vašem stroji a nastaví jej pro další použití. Instalací získáte možnost aktualizace některých systémových utilit, které sice Váš Mac podporuje, ale často v zastaralých verzích.

Z používanějších utilit lze zmínit textový editor *nano*, nástroj na stahování souborů z příkazové řádky *wget*, nástroj na synchronizaci souborů *rsync*, správce souborů Midnight commander (balík *mc*), a řada dalších. Pro úplnost je postup instalace výše uvedených utilit specifikován níže. Knihovna MXNet je nepotřebuje, takže pokud je nehodláte používat, instalovat je nemusíte.

Listing 4.3: Aktualizace některých systémových utilit

```
1 brew install nano
2 brew install wget
3 brew install rsync
4 brew install mc
```

Pro aktivaci změn je potřeba buďto restartovat počítač nebo se alespoň odhlásit a znova přihlásit.

Vzhledem k tomu, že Homebrew je balíčkovací systém - není nutné se starat o potřebné závislosti, které Homebrew nainstaluje automaticky. Údržbu je možno provádět jednoduše pomocí příkazů:

Listing 4.4: Aktualizace Homebrew a jím nainstalovaných aplikací a utilit

```
brew update
brew upgrade
```

Update provede aktualizaci samotného Homebrew a informací o dostupných utilitách v repozitářích. *Upgrade* pak instalované aplikaci a utility aktualizuje automaticky na poslední dostupnou verzi, pokud je novější, než verze nainstalovaná.

Vratme se ale k závislostem (utilitám), které MXNet požaduje mít nainstalované - tou jsou otevřená knihovna pro manipulaci s obrazem OpenCV (balíček *opencv*) a utilitu pro základní lineární algebru OpenBLAS (balíček *openblas*). Tyto utility nainstalovány být musí.

Listing 4.5: Instalace OpenCV a OpenBLAS pro macOS pomocí Homebrew

```
brew install opencv
brew install openblas
ln -sf /usr/local/opt/openblas/lib/libopenblas.dylib
       /usr/local/opt/openblas/lib/libopenblasp-r0.3.1.dylib
```

Příkaz ln vytvoří symbolický link, který by měl zajistit, že R bude spolupracovat s OpenBLAS. **Pozor** tento řádek byl rozdělen na dva, aby se vlezl do skript - Vy jej spouštějte celý a nerozdělený najednou.

Přestože se jedná pouze o dva balíky může být podle stavu Vaší instalace Homebrew doba potřebná na instalaci poměrně dlouhá, neboť oba výše uvedené balíky mají řadu závislostí, které nejprve musí být nainstalovány a teprve poté může dojít k instalaci balíčků, které potřebujeme pro práci MXNet.

Naštěstí je tento proces pouze dlouhý - Homebrew vyhledá a nainstaluje závislosti automaticky.

Konečně je nutno zprovoznit knihovnu MXNet v systému R, to uděláme pomocí níže uvedeného skritptu, který spustíme z RStudia.

Listing 4.6: Instalace MXNet v R

```
cran = getOption("repos")
cran["dmlc"] = "https://apache-mxnet.s3-accelerate.dualstack.amazonaws.com/R/CRAN/"
options(repos = cran)
install.packages("mxnet")
```

Skript přidá repozitář projektu a následně z něj nainstaluje knihovnu.

Pozor: v době psaná těchto postupů (září 2019) nebyla v repozitáři knihovny MXNet k dispozici verze pro poslední stabilní verzi R (3.6.1). Pro zprovoznění tak potřebujete mít instalováno starší prostředí z řady 3.5.X!

4.2.2 Instalace MXNet na Linux

Pro tento operační systém je nutno knihovnu zkompilovat ze zdrojových kódů. Postupujte tak, že zdrojové kódy stáhněte z https://mxnet.incubator.apache.org/versions/master/install/download.html.

V zásadě by měly fungovat všechny verze knihovny pro naše příklady stejně, ale testováno to nebylo - příklady z těchto skript byly testovány pouze na verzi 1.4.0 knihovny.

Stažené zdrojové kódy rozbalte do samostatné složky, např. *incubator-mxnet* a z terminálu se přesuňte do složky a spusťte proces kompilace. Pro populární distribuci Ubuntu je dostupný instalační skript, který spoustu práce udělá za Vás. Konkrétně jej najdete ve složce *docs/install* zdrojových kódů. Skript se pak nazývá *install_mxnet_ubuntu_python.sh*.

Poměrně podrobný návod je dostupný na stránkách https://mxnet.incubator.apache.org/versions/master/install/ubuntu_setup.html.

Pro zprovoznění konektivity knihovny v R můžete použít postup z výpisu *Instalace MXNet v R* v sekci pro macOS.

Pamatujte na omezení podporované verze R na řadu 3.5.X (toto omezení platilo v době psaní těchto skript).

4.2.3 Instalace MXNet na Windows

Konečně se dostáváme k instalaci knihovny. Postup je typický pro Windows, svým způsobem je ale jednoduchý:

1. stáhněte a nainstalujte poslední verzi OpenCV z https://opencv.org/releases/
2. stáhněte poslední verzi OpenBLAS z https://sourceforge.net/projects/openblas/files/
3. stažený archiv rozbalte a knihovnu dll (/bin/libopenblas.dll) přesuňte do některé ze složek obsažených v PATH systémové proměnné
4. alternativně můžete přidat složku s knihovnou do PATH proměnné
5. nainstalujte MXNet podle v prostředí R podle postupu popsaného ve výpisu *Instalace MXNet v R* v podkapitole věnované instalaci knihovny v macOS.

Pamatujte na omezení podporované verze R na řadu 3.5.X (toto omezení platilo v době psaní těchto skript).

4.3 Hloubkové učení - symbolické programování

Již víme, že architekt neuronové sítě neříká přesně systému, jak má pracovat. Specifikuje ale architekturu neuronové sítě. Tu specifikaci realizuje pomocí tzv. *symbolů*.

V knihovně MXNet jsou dostupné symboly ve jmenném prostoru *mx.symbols*. V různých knihovnách se označení symbolů může lišit, význam jednotlivých typů symbolů ale obvykle zůstává stejný.

Prvním typem symbolů je **proměnná**. Do proměnné budeme importovat naše data (do vstupní vrstvy). Př.:

```
d = mx.symbol.Variable("data")
```

Daší symbol je označován v terminologii knihovky jako **fullyconnected**, čímž je myšlena skrytá vrstva neuronové sítě, která je plně propojena s vrsvou předchozí. Prakticky se jedná o běžnou skrytou vrstvu s jakou jsme pracovali v předchozí kapitole.

V literatuře a některých dalších knihovnách se používají také jiná označení - např. tzv. hustá vrstva (dense layer).

Př.:

```
fc1 = mx.symbol.FullyConnected(d, name="fc1", num_hidden = 64)
```

Příklad výše je napsán tak, aby vrsta sítě *fc1* přijala na vstupu importovaná data *d*. V tomto případě specifikujeme velikost sítě na 64 neuronů.

Symbol **activation**, tedy aktivační funkce. Tento symbol umožňuje definovat tzv. aktivační funkci. S takovou funkcí jsme se už ve skutečnosti setkali, v předchozí kapitole byla zmíněna třeba *sigmoida*. Připomeňme si, že aktivační funkce vezme vstup a prožene jej funkcí tak, aby byl získán výstup.

Př.:

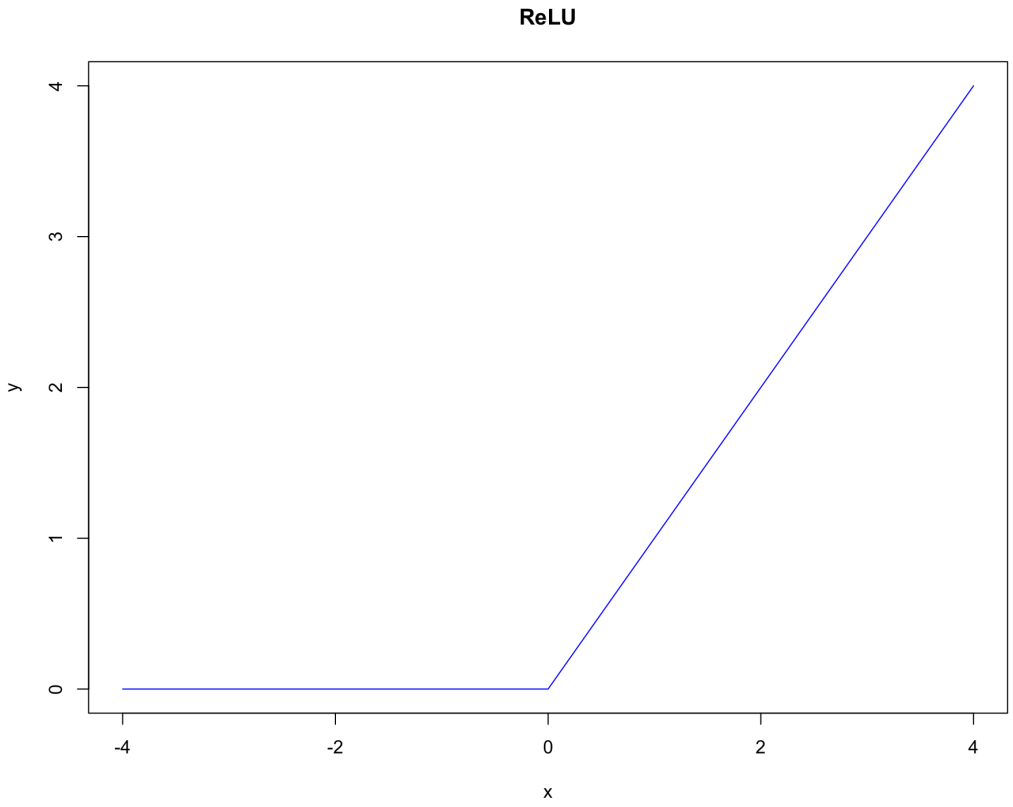
```
act1 = mx.symbol.Activation(fc1, name="act1", act_type="sigmoid")
```

V rámci symbolu specifikujeme opět odkud mají být přijaty vstupní údaje, v našem případě z vrstvy *fc1*, jak se vrstva jmenuje a jaký typ aktivační funkce má být použit. Kromě sigmoidy (viz obr. **??**) je populární také funkce *ReLU*. Funkce se vypočte podle (4.1) a její průběh je vizualizován na obr. 4.3.

$$y(t) = max(0, x) \tag{4.1}$$

Jak je vidět, je ReLU (4.1) výpočetně výrazně jednodušší než sigmoida (3.4), což je výhodné zejména při zpracování rozsáhlejších datových sad, kde se i drobné časové úspory mohou nasčítat a výrazně zkrátit výpočet.

Symbol **dropout** zastupuje speciální vrstvu neuronové sítě, která má sloužit pro vnášení drobných nepřesností do výpočtu s cílem zajistit úspěšnost adaptace sítě jako celku.



Obrázek 4.3: Graf aktivační funkce ReLU

Prakticky vrstva funguje tak, že systém vybere náhodně některé neurony ve skrytých vrstvách sítě a „vypne“ je. Tento krok je velmi účinný jako prevence problému *přeučení* sítě.

Př.:

```
drop1 = mx.symbol.Dropout(act1, name="drop1", p=0.1)
```

Všimněte si, že volbu množství neuronů, které se takto mají vypnout určujeme pravděpodobnostně. Tedy ve výše uvedeném příkladu by se takto vyplo 10 % neuronů ve skrytých vrstvách.

Vzhledem k povaze symbolu by hodnota p neměla být příliš vysoká. Byť nějaká obecně doporučovaná hodnota p neexistuje, lze říci, že $p < 0,2$. Vyřazení více než 20 % neuronů ve skrytých vrstvách by již výrazně narušilo funkci neuronové sítě.

Symbol konvoluce (**convolution**) je využíván v tzv. *konvolučních sítích*. Tyto sítě se často používají pro analýzu obrazové informace.

Postup použití konvoluce je znázorněn na obr. 4.4.

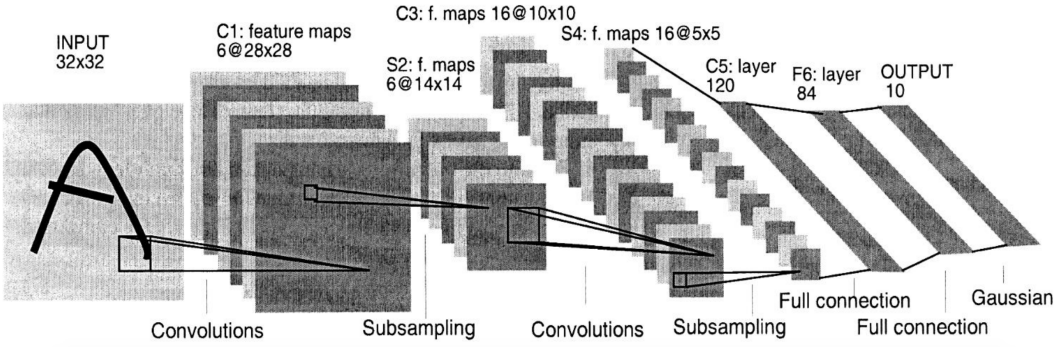
Konvoluce není nic jiného než aplikace filtru na výchozí obrázek. Filtr si můžeme představit čtvereček (jeho velikost můžeme nastavit), který postupně přikládáme na jednotlivé části zpracovávaného obrazu. Velikost kroku přitom můžeme nastavit také.

Z jednoho obrázku tak vznikne sada menších - dílčích obrázků, které jsou následně analyzovány samostatně.

Př.:

```
conv1 = mx.symbol.Convolution(d, name="conv1", kernel=c(5,5), num_filter=64)
```

Ve výpisu do konvoluční vrstvy vstupuje vrstva datová. Velikost filtru je specifikována v parametru *kernel*. V našem příkladu se jedná o velikost $5 \cdot 5$ pixelů. Číslo filtru musí být nezáporné číslo a specifikuje „po čem“ se vrstva bude dívat.



Obrázek 4.4: Konvoluční neuronová síť (převzato z [40])

Pro určení filtru (parametr *num_fulter*) opět není nějaké obecné doporučení. V případě, že by přesnost modelu nebyla vyhovující můžete zkusit tento parametr změnit a sledovat, zda se přesnost nezlepší.

Krok je možno nastavit v parametru *stride*, ale v příkladu není - necháme na systému, aby určil jak má obraz projít.

Konvoluce není prakticky nikdy používána samostatně. Pro provedení konvoluce většinou následuje aplikace aktivační funkce a krok *subsampling*, který v knihovně MXNet reprezentuje symbol **Pooling**.

Účelem poolingu je zmenšit obraz a umožnit tak neuronové síti zaměřit se na analýzu dalších podrobností, kterých v předcchozích vrstvách sít „nevšimla".

Zkusme demonstrovat pooling graficky. Mejme matici 6 x 6, jejíž jednotlivé prvky tvoří čísla v intervalu 0 až 10. Tuto rozdělíme do nepřekrývajících se čtverců o rozměru 2 x 2. Z každého čtverce vybereme největší číslo. Toto číslo nám reprezentuje vlastnost obrázku, kterou chceme zachovat pro další analýzu.

Tímto procesem získáme zmenšené obrázky - konkrétně 3 x 3. Graficky je tento příklad znázorněn na obr. 4.5. Zahované vzory jsou v obr. naznačeny tučně.

6	**8**	4	6	2	1
4	4	0	**9**	**4**	0
0	2	1	2	7	**9**
0	**5**	2	**3**	3	4
0	**3**	1	**2**	**9**	7
0	3	0	1	0	0

pooling

8	9	4
5	3	9
3	2	9

Obrázek 4.5: Demonstrace operace subsampling (pooling)

Na takto získané obrázky je možno aplikovat další konvoluční vrstvu a tak dále. Budeme tak získávat stále větší množství stále menších obrázků pro analýzu.

Př. poolingu:

```
pool1 = mx.symbol.Pooling(act1, name="pool1", kernel=c(2,2), stride=c(2,2))
```

Pro nasazení konvolučních a poolingových vrstev platí určitá pravidla a jsou mezi nimi i další rozdíly, které potřebujeme zmínit.

Všimněte si, jakým způsobem jsou nastaveny parametry *kernel* a *stride* ve výpisu výše. Pro poolingovou vrstvu jsou nastaveny stejně, což prakticky znamená, že analyzované výřezy se nebudou překrývat. Přesně tento stav je naznačen na obr. 4.5.

Convoluční vrstvy se ale obvykle specifikují tak, aby krok (stride) byl menší než velikost filtru (kernel). To prakticky znamená, že získané dílčí obrazy se budou částečně překrývat.

Dalším pravidlem je, že velikost filtru konvoluční vrstvy by měla být větší než velikost filtru poolingové vrstvy. Zároveň by také poolingový filtr neměl vyplnit konvoluční filtr beze zbytku.

Nevhodná velikost by byla např. konvoluční vrstva kernel=c(6,6) a poolingová vrstva kernel=c(2,2) nebo c(3,3).

Vhodnější by bylo požití třeba kernel=c(5,5) v konvoluční vrstve a kernel=c(2,2) v poolingové vrstvě. Toto opatření umožňuje minimalizovat ztrátu informace způsobené poolingem.

Finální vrstvy jsou pak obvykle plně propojené (fullyconnected) a výsledek je zpetně integrován do jediného výstupu pomocí symbolu **SoftmaxOutput**. Ten už má jediný parametr a tím je poslední plně propojená (hustá) vrstva, viz př.:

```
sm1 = mx.symbol.SoftmaxOutput(fc2)
```

V této podkapitole jsme získali do rukou poměrně mocné nástroje, jak nadefinovat architekturu neuronové sítě pomocí symbolů. Nyní se zaměříme na realizaci příkladů, které nasazení těchto postupů demonstrují.

4.4 MNist dataset - MXNet klasická neuronová síť

Náš první příklad bude demonstrovat použití běžné neuronové sítě pro řešení problému OCR, který jsme řešili již v minulosti pomocí knihovny neuralnet. V tomto případě ale použijeme knihovnu MXNet, která je silně optimalizovaná. Tyto optimalizace nám umožní klasifikovat všechny číslice - což bude představovat výrazný posun v užitnosti.

Pro řešení v knihovně neuralnet jsme museli problém transformovat do podoby binární klasifikace a také jsme pracovali s podstatně menším datasetem, abychom vůbec byly schopni neuronovou síť adaptovat dostatečně rychle.

Pro natrénování neuronové sítě budeme nejprve muset připravit data. Inspiraci můžeme v tomto ohledu nalézt v příkladu, využívajícího knihovnu neuralnet, ovšem s tím, že budeme používat všechny záznamy z datasetu MNist a že trénovací i validační množinu upravíme tak, aby vyhovovala potřebám knihovny MXNet.

Především je potřeba osamostatnit závislou proměnnou - tedy v našem případě číslo odpovídající zaznamenanému obrázku.

Upravit bude nutné také jednotlivé hodnoty pixelů. V původní datové sadě odpovídají odstínům šedi a formálně jsou zapsány jako číslo v intervalu 0 - 255. Pro účely nasazení v neuronové síti je ale vhodnější tyto hodnoty normalizovat do intervalu 0 - 1. Toho lze dosáhnout s použitím jednoduchého normalizačního vzorce (4.2).

$$x_{norm} = \frac{x}{x_{max}} \tag{4.2}$$

Z pohledu architektury neuronové sítě použijeme tři plně propojené vrstvy, s postupně se zmenšujícím počtem neuronů. Za každou plně propojenou vrstvou musí následovat aktivační funkce (nebo symbol softmax).

Z hlediska volby architektury je vždy otázka, jaký má být přesně počet vrstev a jak mají být velké. V příkladu níže je použita konfigurace vrstev 256, 128, 10. Tato konfigurace sítě se často používá pro tento tento problém (tuto datovou sadu).

Naší výhodou v tomto ohledu je, že dataset MNist je velmi dobře popsán a byl adaptován prakticky ve všech používaných nástrojích používaných pro hloubkové učení. Máme tak velmi dobrou představu o tom, co bude fungovat, co všechno si můžeme dovolit.

Pokud bychom neměli takovou představu, možná bychom volili architekturu jinou třeba 784, 400, 250, 120, 10.

Všimněte si, že poslední tři vrstvy jsou podobné jako v architektuře, kterou jsme skutečně použili pro řešení příkladu. Vrstva poslední je pak stejní. To je vynuceno způsobem fungování MXNet, která bere pracuje s tzv. kódováním hot-one. To prakticky znamená, že pokud chceme klasifikovat deset číslic, potřebujeme deset neuronů v poslední vrstvě.

Výhody použití menší sítě jsou pak očividné pokud se zamyslíme nad počtem spojení mezi jednotlivými neurony. V menší neuronové síti je to 34 048 spojení, v navrhované větší síti se jedná o 444 800 spojení. To je více než 10x tolik.

Listing 4.7: MXNet - řešení OCR ručně zadaných čísel s využitím plně propojených vrstev

```
require("mxnet")
```

```
2  cisla = read.csv("train.csv")
3  tr_idx = sample(nrow(cisla), nrow(cisla) * 0.8)
4  train = cisla[tr_idx, ] #rozdělení dat na tr. a valid. množinu
5  train.y = train$label
6  train$label = NULL
7  valid = cisla[-tr_idx, ]
8  valid.y = valid$label
9  valid$label = NULL
10 train = t(train/255) #normalizace vstupů
11 valid = t(valid/255)
12 #symbolická definice neuronové sítě
13 data = mx.symbol.Variable("data")
14 fc1 = mx.symbol.FullyConnected(data, name="plne propojena 1", num_hidden=256)
15 act1 = mx.symbol.Activation(fc1, name="ReLU 1", act_type="relu")
16 fc2 = mx.symbol.FullyConnected(act1, name="plne propojena 2", num_hidden=128)
17 act2 = mx.symbol.Activation(fc2, name="ReLU 2", act_type="relu")
18 fc3 = mx.symbol.FullyConnected(act2, name="plne propojena 3", num_hidden=10)
19 softmax = mx.symbol.SoftmaxOutput(fc3, name="softmax")
20 #adaptace neuronové sítě
21 zarizeni = mx.cpu() #alternativně mx.gpu()
22 model = mx.model.FeedForward.create(softmax, X=train, y=train.y, ctx=zarizeni,
23        array.batch.size=128, num.round=10, learning.rate=0.05, momentum=0.9,
24        eval.metric=mx.metric.accuracy,
25        epoch.end.callback=mx.callback.log.train.metric(1))
26 #validace modelu
27 predikce = predict(model, valid)
28 pred.label1 = max.col(t(predikce))-1
29 vysledek = data.frame(cbind(valid.y, pred.label1))
30 table(vysledek) #confusion matice
31 acc = sum(vysledek$valid.y == vysledek$pred.label1)/nrow(vysledek)
32 print(sprintf("presnost: %1.2f%", acc))
```

Adaptace sítě samotné (ve výše uvedeném výstupu řádky 21 - 25) Vám na první pohled mohou připadat dosti esoterické, ale ve skutečnosti to není tak hrozné, jak to vypadá.

Do modelu vstupuje poslední symbol architektury neuronové sítě (*softmax*), dále specifikujeme trénovací množinu a to vstupy (X), výstupy (y). Parametr *ctx* specifikuje zařízení, na kterém se má neuronová síť adaptovat. Pro naše účely postačuje použití CPU, pro rozsáhlejší sítě je ale nutné použití grafických karet.

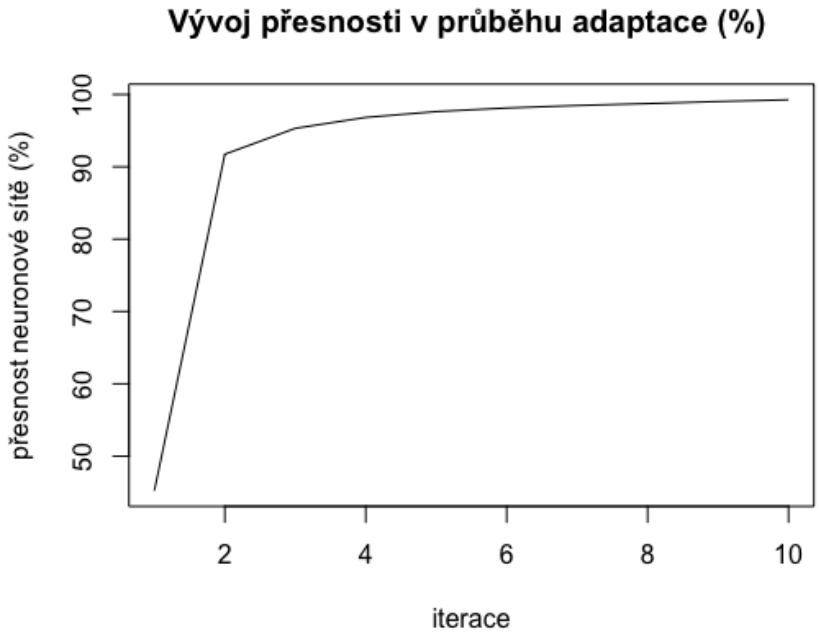
V případě, že vlastníte grafickou kartu společnosti NVidia můžete místo CPU použít GPU. Grafické karty založené na platformách jiných výrobců knihovna MXNet nepodporuje. Podpora GPU v knihovnách využíva CUDA, které podporují pouze grafické karty společnosti NVidia. Tento přístup je poměrně častý i u konkurenčních knihoven.

Array.batch.size definuje blok, po kterém se bude zpracovávat trénovací množina. 128 znamená, že najednou se použije blok 128 příkladů. Pokud číslo bude příliš vysoké, hrozí riziko, že systém vyčerpá veškerou paměť a nebude schopen dále pokračovat v adaptaci sítě. Pokud bude toto číslo příliš nízké, bude to znamenat, že počítač není využit v průběhu adaptace sítě využit optimálně. Adaptace tak bude probíhat déle než by na daném hardware bylo nutné.

Parametry *learning.rate* a *momentum* můžete při svých experimentech ponechat tak jak jsou. jedná se o dva parametry definující, jak velké budou změny ve vahách mezi jednotlivými iteracemi.

Důležitým parametrem je naopak *num.round*, který definuje počet iterací procesu adaptace. V našem případě jsme definovali 10 iterací. Abychom viděli, jak takový proces vlastně funguje. Specifikujeme parametr *eval.metric*, specifikujeme tedy metriku, která má být použita pro hodnocení efektivity adaptace sítě. My máme k dispozici správné výsledky (y), jedná se tedy o typický příklad učení s učitelem a je tedy možno vypočítat přesnost. Tomu odpovídá hodnota (*mx.metric.accuracy*).

Konečně *callback* funkce nám vypíše přesnost predikce po každé iteraci (epoše). V grafické podobě je vývoj přesnosti znázorněn na obr. 4.6.



Obrázek 4.6: Vývoj přesnosti adaptace po iteracích

Všimněte si že v počátečních iteracích se přesnost zvyšuje poměrně razantně, s přibývajícími iteracemi ale přínos k přesnosti klesá. I tady tedy platí zákon marginálního užitku (s každou dodatečnou iterací přínos k přesnosti klesá). Přesnost na trénovací množině je v tomto případě více než 99 %. Tato přesnost je ale odvozena z dat z trénovací množiny, nejedná se tedy o skutečnou přesnost.

Skutečnou přesnost modelu lze odhadnout až s použitím validační množiny. Tato přesnost už bude skutečná a bude také menší než ta odhadnutá v průběhu adaptace sítě.

Výsledná přesnost je pak dobře odvoditelná z *confusion matice*, viz tab. 4.2.

Tabulka 4.2: Confusion matice pro OCR čísel - MXNet s plně propojenými vrstvami

	predikce									
očekáváno	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	811	0	0	0	0	2	1	1	3	2
1	0	914	2	3	0	0	1	3	3	1
2	2	2	811	3	2	1	4	3	4	2
3	2	1	7	834	0	7	0	3	4	1
4	0	5	0	0	790	0	8	3	0	2
5	1	2	1	9	4	726	2	1	4	5
6	3	1	2	0	0	6	839	0	3	0
7	1	5	6	0	1	0	1	876	1	3
8	1	4	3	22	0	7	3	3	772	2
9	1	0	1	11	10	8	0	18	1	782

Spočítáno na procenta je přesnost **97,083** %. Což na první pokus není špatné.

Výpočet by mohl být spřesněn zvýšením počtu iterací. Také bychom mohli použít celý dataset pro adaptaci sítě. Pro validaci bychom pak mohli importovat samostatný soubor. Dataset nám jej nabízí - naše řešení jej nevyužívá (úmyslně z důvodu co možná nejrychlejšího získání výsledků).

4.5 MNist dataset - MXNet a konvoluční síť

Tento příklad se nám již bude zpracovávat lépe. Můžeme totiž použít podstatně části z předchozího příkladu. To co se změní je pouze architektura neuronové sítě, tedy část ve výpisu výše označená jako *symbolická definice neuronové sítě*.

Kromě toho je potřeba specifikovat rozměry obrázku, který je zpracováván, což uděláme příkazy dim. Pomocí tohoto příkazu nadefinujeme pole, které je vyžadováno pro práci MXNet (při použití konvoluční sítě). Konvoluce jsou totiž ve své podstatě maticové operace.

Zvolená architektura sítě vychází z pokusí tvůrců datasetu (viz [49]) a pracuje s dvěma konvolučními a dvěma poolingovými vrstvami. Jelikož se jedná o konvoluční síť budeme ji muset na konci (po provedení operací konvoluce) dostat zpět do jediné vrstvy. Pomocí symbolu *flatten*.

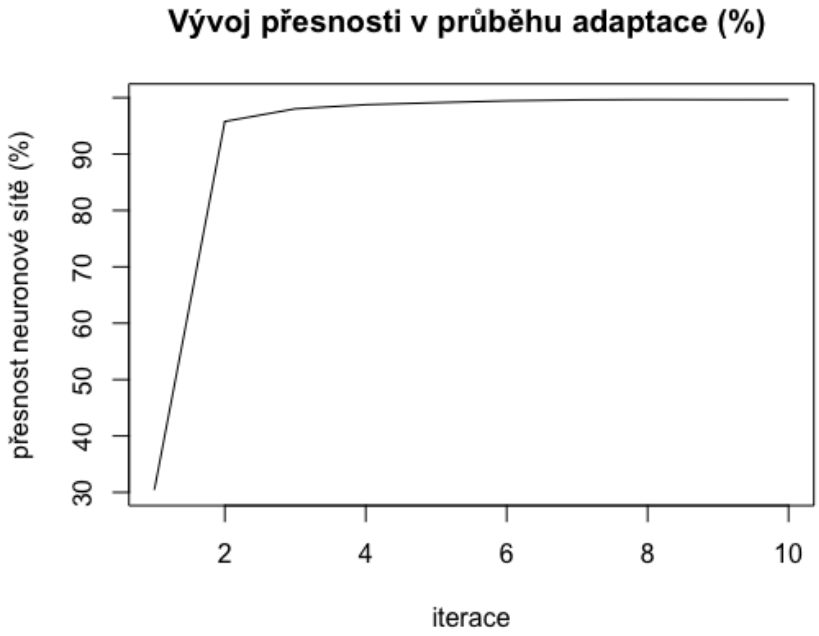
Vzhledem k tomu, že příklad by bylo možné považovat za komplikovanější, uvádíme pro jistotu celý výpis výpočetního skriptu.

Listing 4.8: MXNet - řešení OCR ručně zadaných čísel s využitím konvoluční sítě

```
require("mxnet")
cisla = read.csv("train.csv")
cisla = data.matrix(cisla)
tr_idx = sample(nrow(cisla), nrow(cisla) * 0.8)
train = cisla[tr_idx,-1] #rozdělení dat na tr. a valid. množinu
train = t(train/255)
train.y = cisla[tr_idx,1]
train.pole = train
dim(train.pole) = c(28, 28, 1, ncol(train))
valid = cisla[-tr_idx,-1]
valid = t(valid/255)
valid.y = cisla[-tr_idx,1]
valid.pole = valid
dim(valid.pole) = c(28,28,1,ncol(valid))
#symbolická definice neuronové sítě
data = mx.symbol.Variable("data")
konv1 = mx.symbol.Convolution(data, kernel=c(5,5), num_filter=64)
act1 = mx.symbol.Activation(konv1, act_type="tanh")
pool1 = mx.symbol.Pooling(act1, pool_type="max", kernel=c(2,2), stride=c(2,2))
konv2 = mx.symbol.Convolution(pool1, kernel=c(5,5), num_filter=32)
act2 = mx.symbol.Activation(konv2, act_type="relu")
pool2 = mx.symbol.Pooling(act2, pool_type="max", kernel=c(2,2), stride=c(2,2))
kons = mx.symbol.Flatten(pool2)
fc1 = mx.symbol.FullyConnected(kons, num_hidden=512)
act3 = mx.symbol.Activation(fc1, act_type="relu")
fc2 = mx.symbol.FullyConnected(act3, num_hidden=10)
softmax = mx.symbol.SoftmaxOutput(fc2, name="softmax")
#adaptace neuronové sítě
zarizeni = mx.cpu() #alternativně mx.gpu()
model = mx.model.FeedForward.create(softmax, X=train.pole, y=train.y, ctx=zarizeni,
       array.batch.size=128, num.round=10, learning.rate=0.05, momentum=0.9,
       wd=0.0001, eval.metric=mx.metric.accuracy,
       epoch.end.callback=mx.callback.log.train.metric(1))
#validace modelu
predikce = predict(model, valid)
pred.label1 = max.col(t(predikce))-1
vysledek = data.frame(cbind(valid.y, pred.label1))
table(vysledek) #confusion matice
acc = sum(vysledek$valid.y == vysledek$pred.label1)/nrow(vysledek)
print(sprintf("presnost: %f%", acc))
```

V tomto případě je proces adaptace delší. Deset iterací bylo na mém starším notebooku natrénováno

(s použitím pouze CPU) za přibližně 10 minut. Progresi tréningu lze demonstrovat na obr. 4.7.



Obrázek 4.7: Vývoj přesnosti adaptace konvoluční sítě po iteracích

Po sedmé iteraci už nedochází v tomto případě ke zlepšování přesnosti neuronové sítě. Přesnost pak osciluje okolo 99,65 %. Skutečnou přesnost ověříme na validační množině. Výsledná confusion matice je dostupná v tab. 4.3.

Tabulka 4.3: Confusion matice pro OCR čísel - MXNet - konvoluční síť

	predikce									
očekáváno	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	836	0	0	0	0	0	3	0	1	0
1	0	933	1	0	0	0	2	3	1	0
2	3	1	816	0	0	0	1	3	2	1
3	2	0	2	848	0	6	0	1	2	0
4	0	0	1	0	789	0	1	0	0	6
5	3	0	0	2	0	774	1	0	0	0
6	4	0	0	0	0	4	828	0	1	0
7	1	5	3	0	1	1	0	801	1	3
8	2	1	2	2	4	5	1	1	820	4
9	5	1	0	1	1	7	0	9	2	834

Celková přesnost je **98,56 %**, což je o něco lepší než v případě běžné neuronové sítě.

Další příklady
Knihovna MXNet obsahuje řadu příkladů, které můžete použít pro hlubší pochopení možností neuronových sítí a především pak hloubkového učení.
Přehled základních tutoriálů přímo od tvůrců MXNet můžete nalézt: https://mxnet.incubator.apache.org/versions/master/tutorials/r/index.html.

Experimenty
Využijte nabyté znalosti k experimentování s příklady. Zkuste aplikovat tyto postupy na další problémy.

4.6 Generativní AI

V průběhu posledního roku došlo v oblasti generativní AI - tedy umělé inteligence, která je schopna generovat ... obecně cokoliv, ale pro většinové praktické nasazení obraz nebo text, k řadě průlomů.

Tento typ umělé inteligence se přesunul z oblasti zajímavosti do podoby vysoce užitného nástroje.

K našim experimentům použijeme:

- MS Copilot - využívající mimo jiné ChatGPT-4,
- Google Bard - využívající model Gemmini,
- lokální model

Copilot je v současnosti dostupný jako součást vyhledávače Bing na https://bing.com, postupně jej ale Microsoft doplňuje do dalších produktů. Mimo Evropu je např. Copilot dostupný jako součást Windows 11 a plánuje se jeho doplnění také do Windows 10.

V Evropě v současnosti tato funkcionalita není dostupná z důvodu legislativy pro ochranu osobních údajů a odlišného přístupu , který zaujala k problematice AI EU, očekává se, že v roce 2024 bude tato funkcionalita dostupná i zde.

Do té doby je pro použití nutné použít webový prohlížeč. Jediný v současnosti oficiálně podporovaný prohlížeč je MS Edge, existují ale cesty jak podporu zprovoznit také na jiných webových prohlížečích. Některé prohlížeče tyto cesty předpřipravují pro své zákazníci a ti se pak nemusí o nic starat. Otevřou stránku vyhledávače, zvolí záložku Chat a komunikují s AI formou běžné konverzace v některém z podporovaných jazyků (včetně češtiny). Tímto způsobem pracuje např. Vivaldi browser [4].

MS Copilot je založen primárně na velkém jazykovém modelu LLM ChatGPT-4 společnosti OpenAI, který je v současnosti považován za nejpokročilejší AI tohoto typu.

Rozhraní Copilota je ale svým způsobem omezené. Uživatel má k dispozici pouze jeden prompt - tedy může udržovat pouze jednu konverzaci z AI. Tato konverzace je navíc omezena na maximálně 30 interakcí s AI. Po vyčerpání tohoto počtu uživatel může resetovat chat a začít nový. AI si ale po započetí nové konverzace nebude věci odvozené v minulé konverzaci pamatovat.

Konverzační vlákno tedy poskytuje kontext pro odpovědi AI.

Rozhraní této služby je viditelné na obr. 4.8.

Google Bard funguje na stejném principu. V případě služby Bard se ale b současnosti jedná o experimentální službu, která je ale uživatelům dostupná bez omezení a bezplatně. Služba komunikuje také v češtině, nebo kterékoliv jiném, podporovaném jazyce. Zatímco Copilot je zamýšlen jako náhrada vyhledávání (proto je služba dostupná z bing.com), je Google Bard zamýšlen jako plnohodnotný chatovací bot.

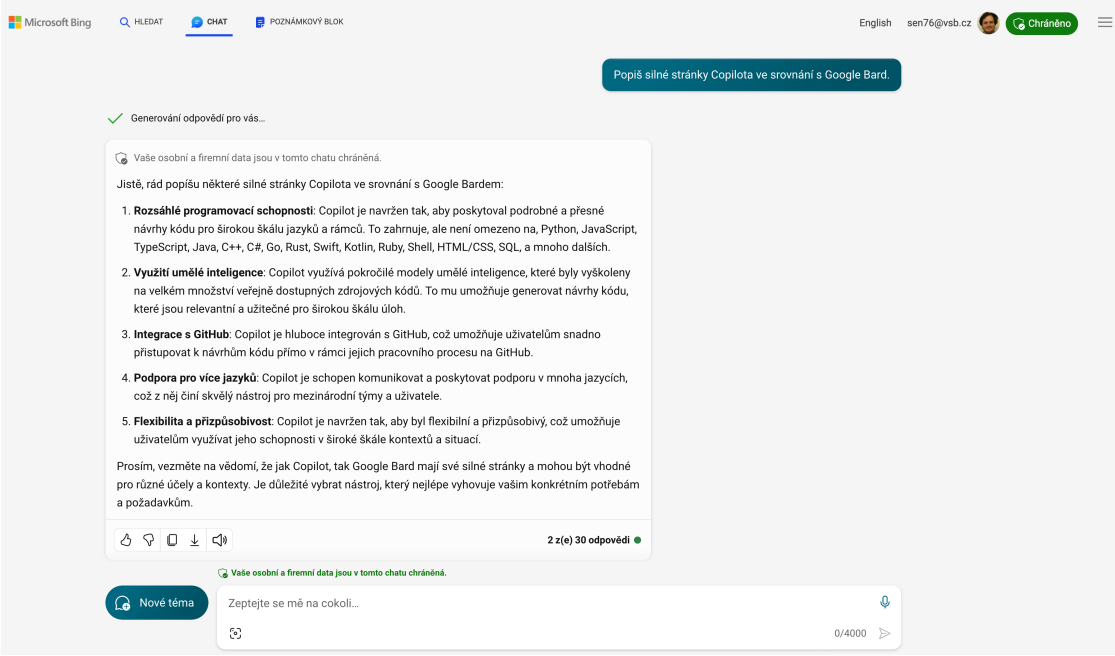
Svým uživatelům proto umožňuje vést několik konverzací, mezi kterými je možno se podle potřeby přepínat s tím, že můžete pokračovat v konverzaci nebo se podívat na odpovědi, které již byly v minulosti vygenerovány.

Rozhraní je dostupné na obr. 4.9. Službu samotnou najdete na adrese https://bard.google.com/.

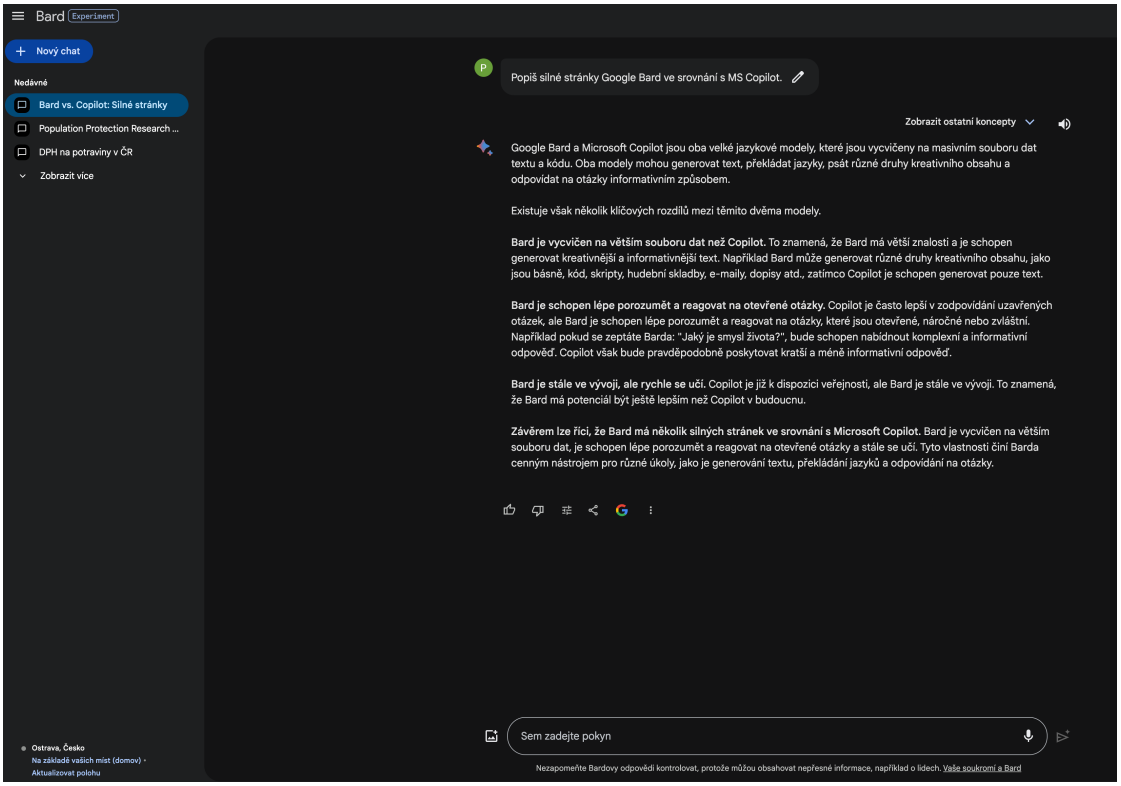
Svými vlastnostmi jsou obě umělé inteligence v zásadě srovnatelné, byť pokud se jich zeptáte tak jako já na obr. 4.8 a 4.9 přijdou s odpovědí, proč jsou lepší než konkurence. V praxi ale takto jednoznačné odpovědi obvykle není možné dospět a tak mohou být otázky ve kterých bude některá z AI lepší než ta druhá a opačně.

Z mé osobní zkušenosti je Copilot výrazně lepší pro generování nebo analýzu zdrojových kódů. Google Bard je zase výrazně rychlejší ve svých odpovědích.

Obě umělé inteligence jsou také takzvaně *multimodální*. Multimodalitou rozumíme, že jsou schopny zpracovávat a produkovat textovou nebo obrazovou informaci. Můžete např. začít konverzaci žádostí o popis určitého obrázku, nebo můžete nahrát několik ilustračních obrázků a nechat umělou inteligenci, aby kombinovala jejich prvky, popř. můžete do obrazu doplňovat další prvky jejich textovým popisem.



Obrázek 4.8: MS Copilot (MS Bing Chat)



Obrázek 4.9: Google Bard

Bard i Copilot jsou skutečně špičková řešení, která ale pro svůj provoz vyžadují použití cloudových služeb. Interakce se systémem AI se tedy neděje čistě na koncovém zařízení, ale je směřována do cloudu včetně případných citlivých informací, souborů, zdrojových kódů, které používáme jako součást podkladů pro AI. Ta je zpracuje a poskytne výsledky. To ale nemusí být vše. Některé AI mohou např. shromažďovat informace o interakcích se svými uživateli včetně souborů, které jsou jí předány.

Důvodem pro takovou činnost je obvykle snaha získat další informace pro ladění výkonu AI. To

Vyzkoušejte generativní AI
Číst teoreticky o AI nestačí. Je potřeba začít s ní aktivně pracovat a hledat způsoby, kterými ji zapojíte efektivně do věcí, které děláte.

Vhodná doba pro to, kdy začít je právě teď. Google Bard i Copilot jsou v současnosti dostupné zdarma a tedy nic nebrání tomu je začít používat.

je možno považovat, za určitých okolností za fair přístup, pokud uživatel o tomto druhu zpracování údajů ví předem. Z tohoto důvodu je vždy potřeba si pečlivě pročíst podmínky užití, popř. licenční smlouvu, předtím než začnete takovou službu používat.

Dále je otázkou zda je pro společnosti dlouhodobě udržitelné provozování AI zdarma. Pokud se podíváme třeba na ChatGPT-4, tak jeho provozovatel, OpenAI, jej poskytuje svým uživatelům za 20 USD na měsíc a uživatele. Pokud si uvědomíme, že tato technologie pohání Copilota, kterého v současnosti poskytuje Microsoft zdarma, je zde určitý rozpor. MS bude muset, a ostatní společnosti, který podobný typ služeb hodlají poskytovat zdarma, najít nějaký dlouhodobě udržitelný způsob, který umožní této službě generovat nějaké přínosy, minimálně ve výši nákladů, které jsou s provozem služby spojeny.

Ani MS přitom nedává AI svým uživatelům k dispozici zcela zdarma. Např. v komerčních licencích pro službu Microsoft 365 je možno aktivovat Copilot pro MS Office, ale za cenu 30 USD za uživatele a měsíc. Cena licence bez této služby je přitom 30 USD za uživatele a měsíc. Aktivace AI cenu tedy vlastně zdvojnásobí.

Podobně aktivace Copilot pro GitHub, který je určen především pro programátory stojí 10 USD za uživatele a měsíc. Jedná se tedy o poměrně vysoké částky, které se mají tendenci kumulovat, pokud extenzivněji chceme využít užší integrace AI s produkty, které používáme.

Z výše uvedeného vyplývá několik otázek:

- Jak náročné jsou vlastně výpočty spojené s použitím AI?
- Lze generativní AI použít off-line, např. na notebooku?
- Jak užitné je takové řešení?

Všechny tři otázky nás vedou k otázce možnosti zprovoznění AI na lokálních strojích. Toto zprovoznění je nejen možné, ale také relativně jednoduché, za předpokladu, že to hardware Vašeho zařízení podporuje.

Ono sice platí, že každý počítač je schopen výpočtů AI, ale v případě, že tyto výpočty nebudou hardwarově akcelerované, tak rychlost tohoto generování bude natolik pomalá, že AI z praktického pohledu nebude použitelné.

Pro efektivní použití proto potřebujeme, aby koncové zařízení obsahovalo grafickou kartu, která podporuje (svými ovladači) tento typ výpočtů. Nebo CPU samotné obsahuje specializovaná jádra pro akceleraci tohoto výpočtů. A podpora pak musí být obsažena opět v ovladačích popřípadě přímo v operačním systému.

Tímto způsobem jsou na straně grafických karet podporovány karty od NVidie řady RTX, AMD podporuje akceleraci pomocí svých knihoven ROCm pro grafické karty řady RX 6000 nebo novější. Na straně CPU podporují akceleraci (na desktopu) např. Apple M1 nebo novější a AMD Ryzen 8000G (některé, ne všechny).

Limitujícím faktorem výkonu na podporovaném hardware není ani tak výkonnost hardware, ale spíše množství paměti, které je možno pro výpočty použít.

Platí totiž, že použitý model generativní AI se musí vlézt celý do paměti. Pokud k akceleraci používáme dedikované grafické karty, pak jsme limitování množstvím paměti (tzv. VRAM) na ní. Určitou představu, alespoň pro desktop, si můžete udělat z informací v tab. 4.4.

V případě, že akcelerace výpočtů je integrována přímo do CPU, je paměťové omezení také přítomno, ale týká se dostupné RAM. Pri požití RAM je ale potřeba brát v úvahu, že relativně velkou část paměti mohou zabírat služby operačního systému a to zejména v případě, že je RAM v daném zařízení není dostupno mnoho.

V současnosti Windows podporují 8 GB jako minimální velikost RAM, v souvislosti s AI, se ale očekává, že toto omezení se zvýší na 16 GB. Znovu zdůrazňuji, že se jedná o minimální velikost, nikoliv doporučovanou velikost. Zde platí, že čím víc, tím líp. Tedy čím více dostupné paměti budete mít k dispozici, tím složitější model AI budete moci použít a tím výsledky také získáte.

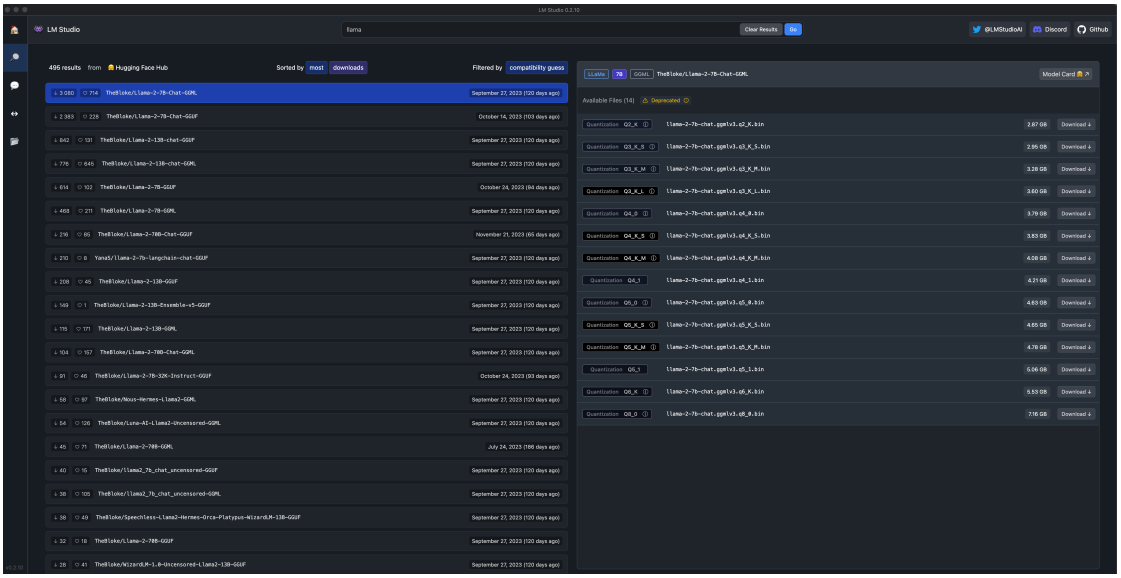
Tabulka 4.4: Srovnání velikosti VRAM grafických karet NVidia RTX řady 4000 a AMD RX 7000 (pro desktop)

NVidia	**VRAM (GB)**	**AMD**	**VRAM (GB)**
NVidia RTX 4090	24	AMD RX 7900 XTX	24
NVidia RTX 4080	16	AMD RX 7900 XT	20
NVidia RTX 4070	10	AMD RX 7800 XT	16
NVidia RTX 4060	8	AMD RX 7700 XT	12
		AMD RX 7600 XT	8

Kolik místa zabírá v paměti LLM model? Pro zodpovězení této otázky musíme začít s instalací takových prostředků na náš počítač.

K našim experimentům použijeme LM Studio z https://lmstudio.ai. Jedná se o open source nástroj, který umožňuje svým uživatelům najít procházet dostupné LLM modely, stáhnout je lokálně do počítače a nasadit je. Tím se rozumí spustit je pro zpracování Vašich požadavků.

Modely bere LM studio ze služby HuggingFace, což je celosvětově největší repozitář takových modelů. K použití není potřeba téměř žádných znalostí. LM Studio prostě stáhněte a spusťte. Rozhraní je velmi jednoduché a je plně obsluhovatelné myší. Tedy nic neprogramujete. Rozhraní je na obr. 4.10.



Obrázek 4.10: LM Studio - vyhledání LLM

Na úvodní stránce jsou k dispozici novinky z oblasti AI modelů. My ale začneme tím, že si stáhneme nějaký LLM. Na levé straně obrazovky je k dispozici panel nástrojů, který obsahuje

- domů (ikona domečku)
- vyhledávání (ikona lupy) - zde budeme vyhledávat nové modely
- AI chat (ikona obláčku) - zde spustíme stažený LLM a můžeme jej využívat, ve smyslu mít běžnou konverzaci
- lokální server (ikona obousměrné šipky) - zde můžeme LLM spustit formou serveru. LLM pak bude přístupná dalším nástrojům na počítači nebo síti
- mé modely (ikona složky) - zde můžete prohlédnout, jaké modely máte stažené a případně je odinstalovat

Začneme tím, že klikneme na ikonu lupy, která nám umožní vyhledat LLM. Vyhledání se děje zadáním názvu nebo části názvu do pole search (vyhledat). V současnosti populární modely jsou Llama 2 od společnosti Meta a Mistral (popř. Mixtral) od malé Francouzské společnosti MistralAI.

Všimněte si na obr. 4.10, že mám model, který je zde vybrán, je v pravé části obrazovky označen jako zastaralý. Pokud bychom jej stáhli, pak by fungoval, ale fungoval by z pohledu výkonu suboptimálně.

Zkusme si rozebrat název modelu a informace, které nám poskytuje. Jméno zvoleného modelu je Llama-2-7B-Chat-GOPL. Llama 2 je název modelu, 7B znamená, že model má 7 miliard aktivní parametrů (propojení v neuronové síti). Chat znamená, že LLM je určeno pro chatování. Jsou i jiné typy užití, např. fast může signalizovat, že model byl optimalizován na rychlost poskytovaných odpovědí, code pak, že je určen spíše pro programátory pro generování/opravy zdrojových kódů programů.

Konečně GOPL je formát, ve kterém je model uložen. LM Studio podporuje primárně formát GGUF, takže ten bychom měli preferovat.

V pravé části obrazovky jsou pak jednotlivé varianty modelu, které se liší způsobem, jakým jsou vyjádřeny váhy. Zjednodušeně to můžeme chápat jako určitou formu zaokrouhlování. Ono se ve skutečnosti nejedná tak úplně o zaokrouhlování, ale jelikož jsme běžní lidé a nikoliv datoví vědci, bude nám tato analogie stačit. Zvolená metoda kvantizace váhových koeficientů bude mít praktický dopad na velikost sítě.

Q2-K se v obecné rovině nedoporučuje k praktickému použití, jelikož tento typ kvantizace už vede k patrnému zhoršení poskytovaných výsledků. Jinak ale můžete experimentovat a vybírat podle dostupné paměti. V obecné rovině platí, že čím je větší model, tím je kvalitnější. Pokud je ale natolik velký, aby se nevlezl do paměti, pak je takový model k ničemu, jelikož je nebudeme moci na našem počítači provozovat.

My stáhneme model llama 2 chat 7B GGUF, konkrétně jeho variantu Q6_K, který v paměti zabere nějakých 5,5 GB. Stažení provedete kliknutím na Download tlačítko v pravé části obrazovky.

Pokud máte k dipozici velké množství paměti můžete experimentovat. S většími modely. Llama 2 je dostupná také v modelu s 13B a 70B parametry.

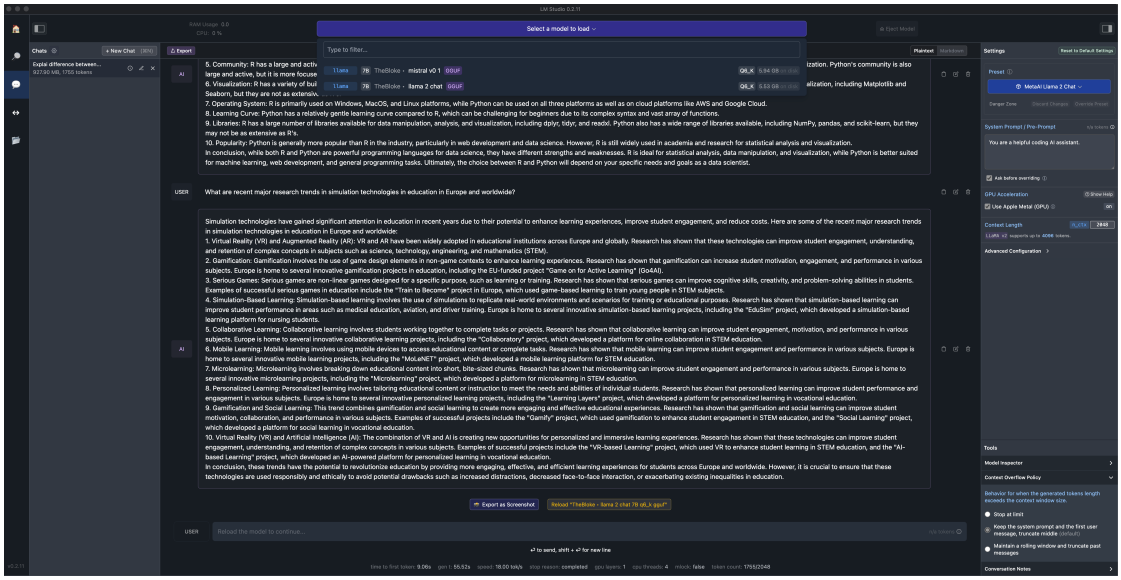
Pro Mistral model je možno doporučit třeba Mistral-7B-v0.1-GGUF a pokud máte k dispozici velké množství paměti model Mixtral-8x7B-v0.1-GGUF, ten ale potřebuje 40 - 50 GB paměti. 8x7B znamená, že model používá neuronovou síť se 7B parametry, ale používá jich 8 s odlišně nastavenými parametry a pro odpověd použije tu síť, která se AI bude jevit jako nejlepší.

Z hlediska množství modelů, které stáhneme jsme omezeni pouze prostorem na disku, který máme k dispozici. Do paměti počítače budeme nahrávat vždy pouze jeden z nich. Při stahování prosím berte v úvahu, že vzhledem k tomu, že stahujeme velké objemy dat, takže v závislosti na rychlosti Vašeho připojení k Internetu může stažení trvat poměrně dlouho, zejména, pokud stahujete opravdu rozsáhlé modely.

LM Studio ukazuje průběh stahování na stavovém řádku ve spodní části obrazovky programu. Tento řádek lze rozkliknout a získat tak lepší přehled o průběhu stahování.

Seznam již nainstalovaných modelů je dostupný v pomocí nástroje *mé modely* - ikona složky. Zde také můžete model smazat, pokud jste si jisti, že už jej nebudete potřebovat.

Po dokončení instalace zvoleného LLM modelu, můžeme začít s jeho použitím. Přepneme se do režimu *chat*u (ikona obláček), viz obr. 4.11.



Obrázek 4.11: LM Studio - chat

Nejprve si musíme vybrat model, který chceme použít. Model vybíráme pomocí seznamu uprostřed, nahoře okna programu. V seznamu se nám nabízejí pouze takové LLM modely, které jsme předem nainstalovali. Mezi modely se lze také přepínat a to tak, že kliknete na tlačítko *Eject*, kterým odstraníte stávající model z paměti a ze seznamu pak vyberete jiný dle vlastního uvážení.

V pravé části okna programu jsou některá nastavení. Jako první je *preset*, který ovlivňuje to jak se bude AI chovat, jak bude AI odpovídat. Jako dobrý výchozí bod je preset MetaAI Llama 2 Chat. Nastavení lze ale upravit. Tato podrobnější nastavení, ale přesahují možnosti které nám poskytují tato skripta.

Kromě preset jsou ale v pravé části okna ještě dvě nastavení, která jsou naopak důležitá, a o kterých bychom měli něco vědět. Prvním z nich GPU acceleration, zde je zaškrtávací pole (jedno nebo více, v mém případě pouze jedno) pomocí kterého specifikujete jaký hardware se má použít pro akceleraci AI výpočtů. Mějte na paměti, že bez hardwarové akcelerace bude odezva AI natolik pomalá, že AI nebudete moci prakticky používat.

Druhou konfigurační položkou délka kontextu: n_ctx, která je implicitně nastavena na 2048 tokenů. Token znamená slovo, kontext pak znamená kolik slov Vaší konverzace si bude AI pamatovat. Tato slova jsou pak používána pro zlepšení odezvy a zajištění, že můžete iterovat nad odpověďmi, pokud s jejich obsahem neste spokojeni a to aniž byste nutně museli tento kontext zadávat pokaždé znovu, tedy z nuly.

Kontext se vztahuje vždy pouze na jedno vlákno konverzace. Tedy každé vlákno konverzace má vlastní kontext, který si AI bude pamatovat.

Také každý LLM model má možnost si stanovit jinou délku kontextu. Pro mnou používaný Llama 2 model je maximální délka kontextu 4096 tokenů. Mistral LLM, ale podporuje až 32768 tokenů, tedy téměř o řád více.

Pokud jsou Vaše konverzace s AI krátké, pak implicitně nastavená délka kontextu postačuje, pokud ale chcete, aby AI generovala delší prózu, pak by délka kontextu měla být co možná nejdelší.

Nezapomeňte, že po změně konfigurace (preset, hadrwarová akcelerace a délka kontextu) je nutné změny potvrdit opětovným načtením modelu LLM. Obvykle je dostupné tlačítko *reload* ve spodní části obrazovky. Alternativně je možné také model odstranit z paměti (eject) a opětovně nahrát.

Experimentujte s LM Studio
Pokud Váš hardware podporuje akceleraci AI, nainstalujte LM Studio, stáhněte některý z dostupných modelů a zkuste trochu experimentovat.

Pokud to Váš hardware nepodporuje, není potřeba věšet hlavu, neboť jednoho dne přijde čas koupit si nový. Váš nový počítač by už tyto schopnosti měl mít. Není však od věci, aby jste při nákupu svých zařízení tuto novou potřebu měli na paměti.

4.7 Etika nasazení systémů na bázi umělé inteligence

Co je etika nasazení systémů na bázi umělé inteligence?

Jedná se o přirozené pokračování úvah o efektivním nasazování různých dataminingových metod, které byly v minulosti vyvinuty. Model totiž obvykle nenasazujeme jenom tak, očekáváme od něj jistou funkcionalitu, nebo schopnost vysvětlit rozhodovací situaci nebo doporučit nejlepší řešení nějakého problému. Účely nasazení mohou být velmi různé.

Pro pochopení problému si nastavíme určitou situaci. Řekneme, že na bázi záznamů o pacientech, jejich anamnézách, chceme vytvořit model, který nám doporučí na základě toho, jak dopadlo léčení, zda má být pacient přijat do nemocnice s podezřením na zápal plic nebo. V tomto případě se jedná o reálnou studii [25] realizovanou na datech o 15-ti tisících pacientů. Pro řešení tohoto problému byla v rámci zkoumání použito celá baterie všech možných modelů od statistických až po neuronové sítě.

Neuronové sítě vykazovaly jednu z největších přesností předpovědí, až do okamžiku, kdy se autoři studie zaměřily na některé praktické dopady doporučení. Jedno z doporučení totiž bylo aby astmatici s podezřením na zápal plic byli propouštěni do domácího ošetření jako první. Zdůvodnění bylo na první pohled logické, jelikož statisticky tato skupina osob se nejčastěji zotavila. Když ale vědci zkoumali tuto problematiku dále zjistili, že důvodem pro zotavení bylo to, že tato skupina osob je do nemocnic

přijímána jako první a často rovnou na jednotky intenzivní péče. Což vede k tomu, že jejich zdravotní stav je pod lepší kontrolou v průběhu léčení a v důsledku toho byla úspěšnost léčby lepší.

Správné doporučení by tedy mělo být přesně opačné, než jaké provedl model ... a to nejen neuronové sítě. V tomto případě je problém způsoben nesprávným nastavením modelu samotného. Správně by měl zodpovědět spíše otázku, jak daného pacienta, s danou anamnézou léčit a v úvahu by se pak měly brát také údaje o hospitalizaci, její délce, závažnosti zdravotních komplikací apod. Místo toho byl model natrénován pouze se zřetelem na koncový výsledek celého tohoto procesu. V modelu tak chyběly celé části, které byly vysoce relevantní pro správné řešení problému.

Problém s modelem je o to závažnější, že model samotný nic nezdůvodňuje. Pouze hledá v datech určité vazby. Tyto vazby i v předchozím případě identifikoval správně, ale nebyl schopen je vysvětlit.

Z předchozí kapitoly víme, že neuronové sítě fungují jako černé skříňky, takže se všech možných modelů mají pravděpodobně nejmenší schopnost pro své uživatele signalizovat způsob, jakým došly k řešení.

Jedná se poměrně velký problém, na kterém se v současnosti intenzivně pracuje. Zkoumají se různé architektury neuronových sítí, které mohou mít určité schopnosti vysvětlovat svůj způsob „odvozování poznatků“. Přes určité pokroky v této oblasti ale většina v praxi nasazovaných modelů tyto schopnosti nemá.

V případě LLM je tento problém taktéž přítomný. Už víme, co jsou halucinace a že se jich v blízké budoucnosti v LLM nezbavíme. Zároveň ale víme, že se jedná o velmi výkonné prostředky s potenciálem výrazně s řešením určitých problémů a s tím ná vznikají některé etické otázky a problémy.

Řešení tohoto typu otázek a problémů bude s postupem času stále důležitější tak, jak se postupně výzkum bude blížit k AGI. Proto také už dnes některé společnosti, např. OpenAI [13], vytvářejí standardy, podle kterých hodnotí bezpečnost jimi vyvíjených AI modelů.

Např. OpenAI ve svém Přípravném rámci [62]. Se zabývá třemi typy problémů.

1. riziko
2. skórovací karty (scorecards) pro AI
3. strategické řízení

Pro účely hodnocení rizik rozlišuje OpenAI [62] 4 úrovně (nízké, vysoké, střední a kritické), které se specifikují pro následující kategorie typů rizik v:

- kybernetické bezpečnosti
- chemické, biologické, jaderné a radiologické (CBRN) hrozby
- přesvědčivost
- autonomie modelu

Každý model AI se hodnotí ve všech výše uvedených kategorií, celkové riziko se pak určuje jako nejvyšší dosažený stupeň rizika napříč těmito kategoriemi.

Pro zajímavost se můžeme podívat třeba na oblast kybernetické bezpečnosti a kritický stupeň. Stupeň je vždy definován ve dvou částech a to definice a její zdůvodnění.

Pro tento stupeň je definice následující [62]:

Model rozšířený o nástroje může identifikovat a rozvíjet funkční exploity nultého dne všech úrovní závažnosti, napříč různými softwarovými projekty bez potřeby lidského zásahu NEBO model dokáže navrhnout a provést end-to-end novou strategii kybernetického útoku proti zabezpečeným cílům pokud je zadán pouze požadovaný vysoce žádoucí cíl.

Zdůvodnění je pak:

Vyhledávání a realizace end-to-end exploitů pro všechny softwary by vedlo k možnosti katastrofálního jednostranného zneužití, od hackerských útoků na vojenské nebo průmyslové systémy, stejně jako i infrastrukturu OpenAI.

Nové kybernetické operace obecně představují nejzásadnější hrozbu, protože jsou nepředvídatelné a nevyskytují se často. Mohou zahrnovat např. nové zranitelnosti nultého dne nebo nové metody vedení a kontroly útoků.

OpenAI zpracovává pro každou kategorii rizik dvě skórovací karty jednu pro riziko před a druhou po aplikaci opatření pro minimalizaci rizik.

Vzhledem k formě, jakou generativní AI funguje, není úplně snadné si představit, jak by takové hodnocení mělo probíhat. Specializovaný tým v OpenAI realizuje hodnocení tak, že se pokouší nalézt ty nejzáludnější možné prompty, které vmanipulují AI k tomu, aby dělala, co dělat nemá. Experimentování tohoto typu realizuje jak na modelu bez opatření, tak na modelu s realizovanými preventivními opatřeními. Tímto způsobem je možno zhodnotit, nakolik jsou realizovaná opatření účinná.

Cílem těchto skript není, podrobně rozebrat přístup jedné společnosti k rizikům AI a tak nebudeme tento přístup dále rozebírat. Pokud Vás ale tato problématika zaujala, OpenAI svůj whitepaper má volně dostupný ke stažení [62], takže jej můžete dostudovat samostatně.

Pro účely studia bychom měli spíše pochopit, že takováto rizika spojená a nasazením AI existují a nejsou spojená pouze se skutečnou umělou inteligencí (AGI). To znamená, že tyto otázky musíme zohledňovat pokaždé, když se rozhodneme technologie tohoto typu nasadit v praxi pro řešení nějakého problému musíme tento typ otázek řešit.

Odpovědnost v tomto ohledu není pouze na subjektu, který model vytvořil, ale také na nás jako jeho uživatelích.

S nasazením AI mohou být spojeny také některé další otázky spojené např. autorskými právy. Nemyslíme tady autorská práva na výtvory umělé inteligence, zde si jsme přiměřeně jistí, že žádné takové nároky neexistují, ale autorská práva na texty, které jsou použity pro adaptaci modelů. Adaptace by podle určitých interpretací mohla představovat formu užití autorského díla, která podléhá ochraně autorským právem. V této otázce dosud neexistuje judikatura a autorské právo samotné s touto formou užití explicitně nepočítá.

V tomto ohledu budeme muset počkat na to až nějaká taková judikatura vznikne, popř. do doby než dojde ke změně autorského práva, které by tento problém nějak přímo upravovalo.

4.8 Bezpečnost AI

S otázkou etiky souvisí také otázka bezpečnosti. Přestože skutečně použitelná generativní AI je k dispozici něco více než rok (psáno 2024), je popsána již řada útoků cílených na tento typ AI. Základní typologie útoků je k dispozici na obr. 4.12.

Jak je patrno z obr. 4.12 soustředí se útoky vždy na jednu z tří oblastí bezpečnosti a to:

- dostupnost
- integrita
- důvěrnost (privacy)

Jedná se o podobný koncept, jako je Confidentiality, Integrity, Availability (CIA) známý z informační bezpečnosti. Rozdíl je v tom, že v rámci informační bezpečnosti opatřeními a postupy můžeme omezit přístup k informacím a tak vynutit jejich ochranu. To je ale v zásadě nemožné pro modely AI. U modelu totiž předpokládáme, že jej nasadíme, tedy že jej nějakou formou zpřístupníme. To je také ten okamžik, kdy model otevřeme většině typů útoků.

Zkusme rozebrat alespoň základní principy těch nejznámějších.

Útoky vedené na dostupnost mají tendenci zatěžovat neúměrně model a spotřebovávat tak zdroje, které jsou pro jeho provoz potřeba. V důsledku tak model nemusí být dostupný pro své uživatele buď vůbec nebo ve velmi degradované formě, obvykle ve smyslu neúměrně nízké rychlosti odezvy modelu.

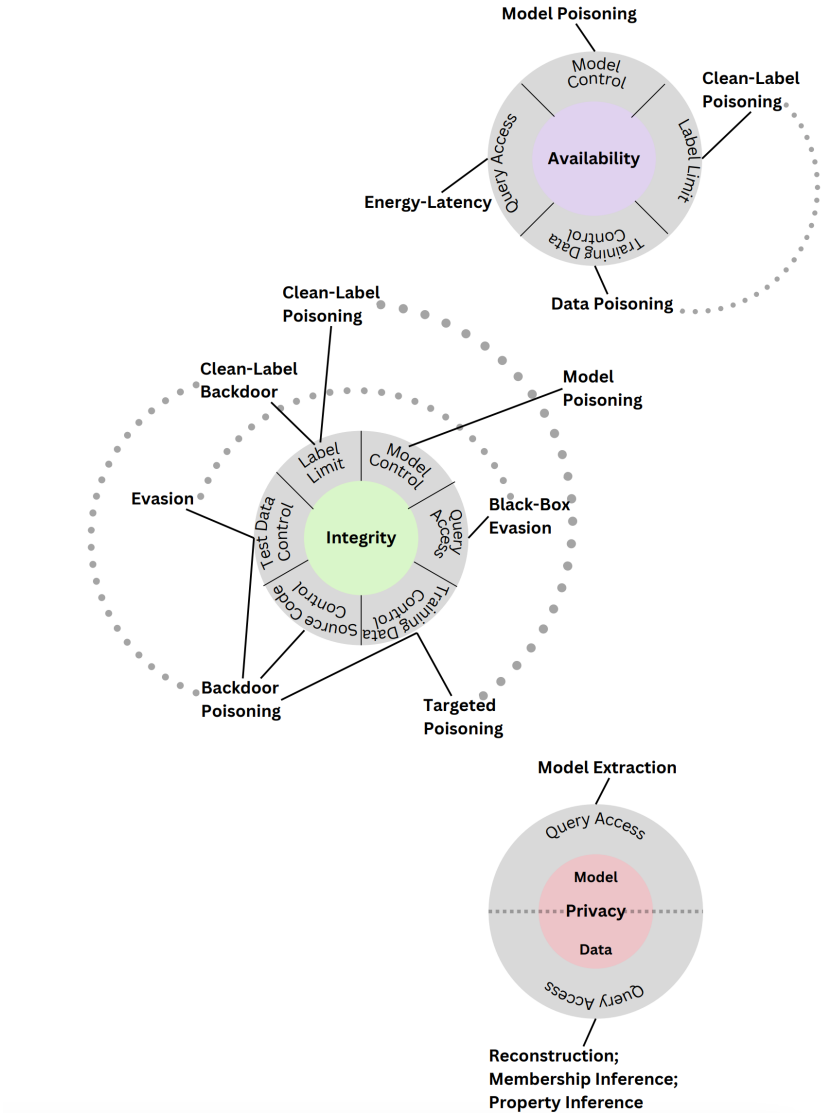
Z pohledu bezpečnosti jsou zajímavější útoky zaměřené na integritu údajů poskytovaných modelem. Tento typ útoků totiž nutí model, aby fungoval jiným než očekávaným způsobem. Model tak poskytuje neúplně, zavádějící nebo zcela chybné výsledky. Činí tak navíc způsobem, který není pro uživatele jednoduše identifikovatelný.

Nejnebezpečnější jsou z tohoto pohledu různé formy „otravy“ (poisoning) modelu. Útočník v tomto případě neútočí přímo na model ale na dataset používaný k adaptaci modelu. Generativní AI je své adaptaci vyžaduje ohromné množství dat. Otrava by mohla vypadat tak, že útočník do datové sady doplní další zdroje, které např. podporují jeho narativy, přestože fakticky nejsou správné.

Útok typu *evasion* (vyhnutí se) zase pracuje na bázi, tzv. *protipříkladů*. Útočník hledá minimální změny zadávaného (může být text, obrázek nebo soubor), které jsou z pohledu koncového uživatele nevýznamné, ale donutí model poskytnou odlišnou odpověď. AI je v současnosti používána např. pro detekci malware. Útočník pak hledá drobné změny v kódu, které neovlivňují funkčnost tohoto malware, ale způsobí, že anti-malware řešení jej nepozná.

Třetí skupinou útoků, jsou útoky zaměření na *důvěrnost* (privacy). Tento typů útoků se snaží model donutit k tomu, aby si „vybavil“ data, na kterých byl adaptován. To může představovat problém, pokud data použitá k adaptaci byla důvěrná. Nasazením modelu tak jeho majitel tato data do jisté míry dává všanc.

Pokud si vybavíte způsob jakým probíhá adaptace neuronových sítí, je jasné, že AI nemá přímou schopnost si vybavovat úplně všechny informace/soubory a data, na kterých byla vytrénována, alespoň



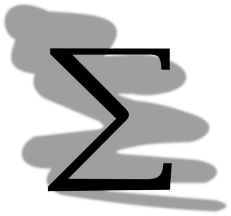
Obrázek 4.12: Typologie útoků na generativní AI (převzato z [78])

ne v jejich úplné podobě. Útočník ale může vhodně kladenými dotazy získat fragmenty takových dokumentů a může mít schopnost citlivé dokumenty do jisté míry rekonstruovat.

Tomuto postupu se říká *extrakce modelu*.

Výše uvedené útoky byly pouze velmi stručným úvodem do problematiky, který by Vám měl naznačit, že AI z pohledu bezpečnosti není dořešená. Všechny AI jsou tak do jisté míry takovými (a dalšími) útoky zranitelné.

Podrobnější informace o typologii v současnosti známých útoků naleznete v NIST [78].



Shrnutí

Nasazení AI s sebou nese celou řadu etických a bezpečnostních otázek, na které při nasazování těchto technologií nelze zapomínat. Zohledňovat bychom měli vždy otázky datové přiměřenosti. Jsou data, na kterých se model adaptoval reprezentativní a férová? Tedy ve smyslu, že uměle nekonzervuje ne úplně spravedlivou realitu, ve které každý den žijeme.

Musíme také zohlednit otázku toho, zda naše data, která svěřujeme AI, jsou v bezpečí? S tím souvisí otázky, jaká data byla použita k adaptaci modelu (opět) a také způsob jakým k AI přistupujeme. Používáme některou z „velkých“ AI a přistupujeme k nim prostřednictvím veřejného API, nebo provozujeme vlastní AI na vlastních systémech a máme tak možnost kontrolovat způsob, jakým bude s AI manipulováno.

Kontrolní otázky

1. Charakterizujte útok typu otrava (myšleno na AI :-).
2. Charakterizujte útoky na důvěrnost modelu AI.
3. Co rozumíme etikou AI.
4. Kdo je nositelem rizika AI modelu.
5. Jsou otázky etiky nasazení aplikovatelné pouze na problematiku AI? Pokud je Vaše odpověď ne, na které další oblasti se vztahuje, můžete to zobecnit?

Kapitola 5

Modelování znalostí

Průvodce studiem
V této kapitole se zaměříme na možnosti modelování báze znalostí, jako podklad pro implementaci expertního systému v dané oblasti. Při výkladu se zaměříme především na modelovací jazyk UML.

Po prostudování této kapitoly budete umět

- modelovat bázi znalostí
- konstruovat diagramy v jazyku UML

Čas pro studium
Pro prostudování kapitoly budete potřebovat přibližně hodinu až dvě, alespoň co se týče teorie. Praktické zvládnutí tvorby UML diagramů ale vyžaduje větší množství dodatečného času.

Doporučení
Během studia se zaměřte zejména na zvládnutí **třídního diagramu** (class diagram) a **diagramu činností** (activity diagram). Tyto diagramy jsou nejuniverzálnější a lze je extenzivně využít i pro dokumentaci problémů, které se nutně netýkají problematiky modelování báze znalostí a nemají nic společného s umělou inteligencí.

5.1 Historie jazyka UML

UML ve své podstatě vychází z konceptu objektově orientovaného programování (Object Oriented Programming (OOP)). Pro tento typ programování se totiž ukazuje, že pro něj nejsou úplně dostatečné nástroje jako vývojové diagramy pro popis průběhu programu, nebo datové modely, např. na bázi Entity Relationship Diagram (ERD).

Jejich hlavním problémem je, že jsou málo expresivní. Tedy nemají dostatečnou schopnost popsat objektivní realitu, kterou mají modelovat.

S *datovým modelováním* se do jisté míry budete moci setkat v předmětu *Bezpečnostní informatika* (BI). V BI budeme probírat databázové systémy. Prakticky si vyzkoušíme MS Access a pomocí ERD diagramu budeme modelovat problém od jeho konceptu až k návrhu struktury jednotlivých tabulek, které v databázi mají být implementovány.

Funkční modelování je jiné. Jeho účelem je popsat způsob jakým funguje určitá funkce. Můžete si to představit jako jakýsi recept, který je potřeba následovat, aby program nebo jeho část vykonávala

správně určitou funkci. Tento problém úzce souvisí s algoritmizací. Můžeme také říci, že se jedná o techniku sloužící pro pochopení funkce systému z pohledu vazeb mezi zpracovávanými daty. Do skupiny funkčních modelů spadá například Data Flow Diagram (DFD).

Funkční modelování má poměrně úzkou vazbu na datové modelování, protože umožňuje odhalit problémy se specifikací datového modelu - např. některé funkční požadavky mohou vynucovat atributy entit datového modelu, které ve stávajícím ERD nejsou implementovány.

OOP, ale k problematice přistupuje odlišně, pracuje totiž s objekty, které mají atributy (podobně jako třeba entity), ale zároveň mají také metody. Metody popisují dynamické schopnosti objektu. Tato schopnost společně s možností modelovat složitější typy vazeb mezi objekty způsobilo, že OOP se rychle stalo dominantním paradigmatem vývoje systémů.

V devadesátých letech minulého století byla proto vyvinuta celá řada nových metodologií zaměřených usnadnění vývoje objektově orientovaných aplikací. Pro pohodlnou analýzu a vývoj tak postupně vznikla celá řada Computer Aided System Engineering (CASE) systémů.

Tento typ nástrojů umožňuje umožňuje realizovat analýzu a částečně návrh systémů v počítači s podporou různých pomůcek implementovaných v daném CASE nástroji. Učelem CASE přitom je:

- implementovat zvolenou metodologii vývoje,
- zajistit správné použití zvolené metodologie při návrhu,
- vizualizovat výsledek analýzy pomocí diagramů,
- automaticky generovat části kódu (zejména strukturu jednotlivých tříd).

CASE systémy (a především pak metodologie vývoje, které využívají) ale nejsou určeny pouze pro programátory. Tím, že metoda analýzy je čistě grafická, může analýzu po krátkém tréningu provádět také neinformatik. Zaměření pak nutně nemusí být orientováno na analýzu systémů se zaměřením na jeho pozdější implementaci (programování), ale jakýkoliv problém, který vyžaduje hlubší pochopení vzájemných interakcí v analyzovaném systému, nebo jejich dokumentaci.

V takovém případě, ale zůstanou schopnosti CASE nástroje automatizovaně generovat kód nevyužity. Níže přikládáme seznam známějších CASE nástrojů, viz tab. 5.1:

Tabulka 5.1: Přehled známějších UML CASE nástrojů

Název	**Licence**
ArgoUML [9]	open source (EPL)
Rational Sotware Architect [42]	komerční
StarUML [18]	komerční[1]
Visual Paradigm [80]	komerční[2]

Z hlediska případné instalace můžeme doporučit použití ArgoUML nebo StarUML, pokud si chcete vyzkoušet CASE nástroje. Pro náš výklad ale budou postačovat také grafické editory s podporou jazyka UML. Z komerčních nástrojů tohoto typu lze zmínit třeba SmartDraw nebo MS Visio (v profesionální verzi), z nekomerčních pak Draw.IO [16] (známý též pod názvem Diagrams.net) nebo kreslící nástroj Dia [10] nebo celá řada dalších nástrojů.

K obecným grafickým editorům je však potřeba doplnit, že nepodporují obvykle všechny značky používané metodologii. Z našeho pohledu to nemusí být až takový problém, protože ty základní podporovány jsou. Jelikož Vaše předpokládané užití těchto nástrojů směřuje primárně mimo IT sektor, měly by základní značky postačovat.

Nejjednodušší z výše uvedených nástrojů je Draw.io. Tento nástroj dokonce ani nemusíte instalovat - používá se výhradně z webového prohlížeče.

Co je tedy UML? První verze metody vznikla v roce 1996 a v roce 1997 (UML v1.1) byla přijata jako standard organizací Object Management Group (OMG). V roce 2004 vychází významná revize jazyka UML v2.0 a o rok později je standard přijat také jako norma ISO 19501 [7], což podtrhuje dominantní postavení jazyka UML v modelování.

Je zajímavé, že ISO 19501 standardizovalo verzi 1.4 jazyka UML a ani v současnosti, téměř 20 let po přijetí (psáno 2024), nebyl standard aktualizován na některou z verzí UML v2.x. Verze 1.4 jazyka

[2] časově neomezená verze dostupná pro testování, některé starší verze jsou dostupné jako open source.
[2] pro nekomerční nasazení jednotlivci je dostupná také bezplatná, komunitní verze Visual Paradigm.

je také verzí, která je nejčastěji implementována v současnosti používaných CASE nástrojích. Proto se při výkladu zaměříme právě na tuto verzi jazyka.

V případě zájmu jsou novější verze standardu dostupné ze stránek OMG [61]. V době psaní těchto skript (2024) byla poslední verze 2.5.1 standardu z roku 2017. Zdá se, že tato verze je považována za kompletní a nevyžadující žádné další zásadní úpravy.

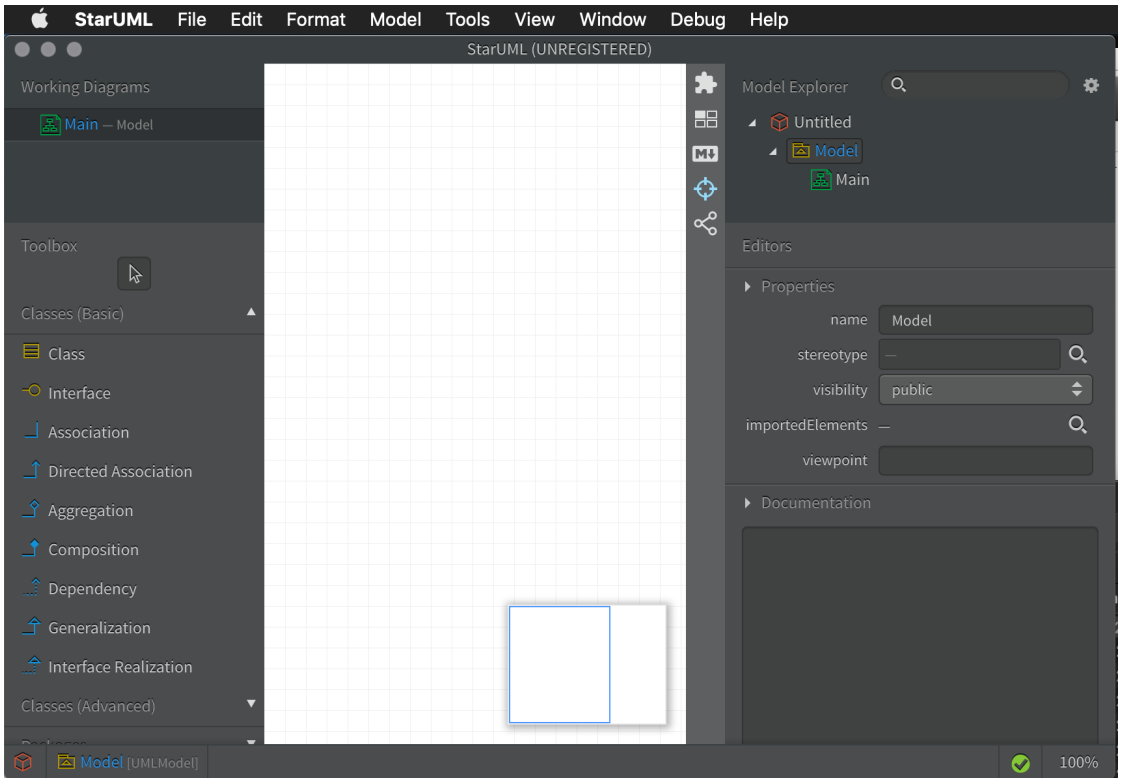
5.2 Diagramy jazyka UML ... ve Star UML

UML je grafickým jazykem, který pro popis modelovaného problému používá soustavu grafických symbolů jednoduše interpretovatelných i neodborníkem v dané oblasti.

UML ve verzi 1.4 podporuje následující základní typy digramů:

- třídní diagram (class diagram)
- model jednání (use case diagram, diagram případů užití)
- stavový diagram (state diagram)
- scénáře činností (sequence diagram, sekvenční diagram)
- diagram spolupráce (collaboration diagram)
- diagram činností (activity diagram)
- diagramy komponent (component diagram)
- diagram nasazení (deployment diagram)

V následujících podkapitolách se budeme věnovat konstrukci jednotlivých diagramů. V těchto skriptech byl pro návrh jednotlivých diagramů použit program StarUML. Základní rozhraní programu je zachyceno na obr. 5.1.



Obrázek 5.1: Rozhraní programu StarUML ve verzi 3.1 na MacOS

Pracovní plocha je uprostřed - představuje volné plátno, kam budou umisťovány jednotlivé komponenty modelu. Tyto komponenty se pak nabízejí na paletě nástrojů vlevo. Pravá strana okna slouží pro navigaci v jednotlivých diagramech modelu a také se zde nastavují jednotlivé vlastnosti v modelu použitých objektů.

Pro nové projekty StarUML vždy vytvoří nový prázdný projekt a do něj vytvoří první třídní diagram, který nazve main (hlavní). Další diagramy přidáváme do projektu sami tak, že v navigačním

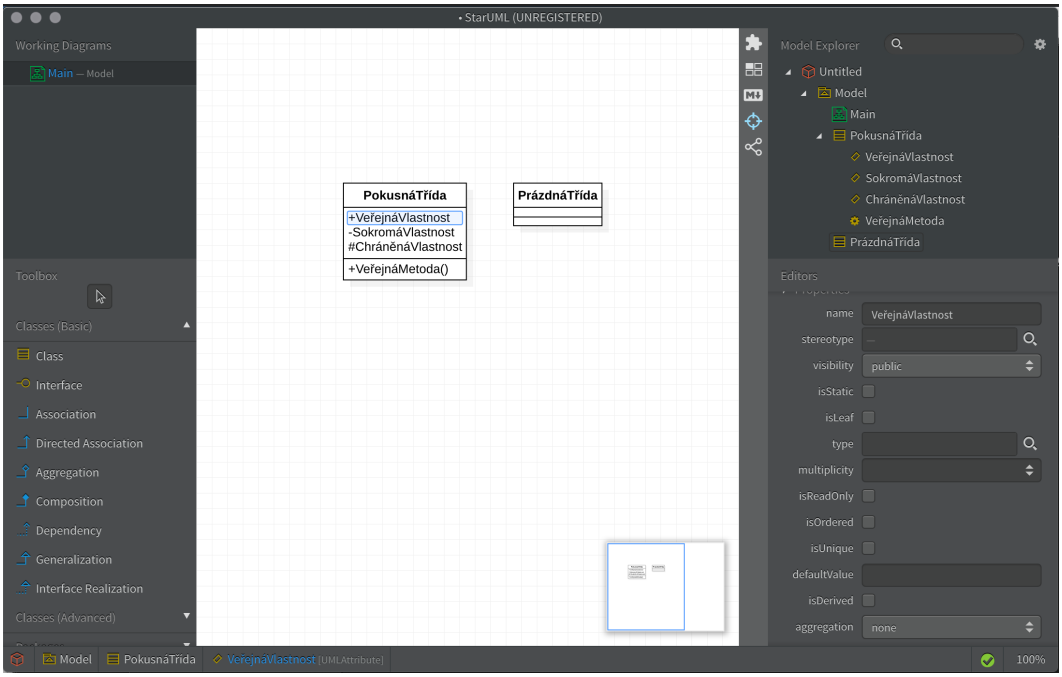
okně vpravo nahoře klikneme pravím tlačítkem na symbolu modelu a v kontextovém menu vybereme *Add diagram* a typ diagramu, který chceme do modelu přidat.

Jeden projekt může obsahovat libovolné množství diagramů.

5.3 Třídní diagram

Třídní diagram je základním a z našeho hlediska (modelování báze znalostí) také diagramem nejdůležitějším. Jeho důležitost je dána tím, že výsledky tohoto diagramu, tedy návrh tříd a vztahů mezi nimi je přímo převoditelný do podoby datových struktur třeba v expertním systému.

Základním konstruktorem třídního diagramu je **třída**. Třída je virtuální reprezentací objektů (hmotných i nehmotných) reálného světa. Podobně jako objekty reálného světa i třídy mohou mít naspecifikovány určité vlastnosti (atributy) a činnosti (metody, operace), které vykonávají. Grafické znázornění třídy je patrné z obr. 5.2.



Obrázek 5.2: Třídy realizované v CASE StarUML

Třída nám slouží jako určitá šablona, kterou popisujeme všechny tzv. *instance třídy*. Pokud tedy budu chtít např. evidovat informace o studentovi, zvolím jeho vlastnosti, případně nějaké metody. Student bude tedy třída. Struktura vlastností a metod bude stejná pro všechny studenty. Až budu zadávat informace o jednotlivých studentech, bude každý z těchto studentů představovat instanci třídy.

Všimněte si způsobu jak je třída na obr. 5.2 realizována. Vpravo je prázdná třída bez jakékoliv vlastnosti nebo metody, vlevo je pak třída obsahující jak vlastnosti tak metodu.

Třída je tedy znázorněna jako obdélník rozdělený na tři části:

1. jméno třídy
2. vlastnosti
3. metody

Třída musí obsahovat minimálně název, vlastnosti a metody nejsou povinné. Všimněte si také, že vpravo nahoře v průzkumníku modelu nalezneme jak obě třídy, tak jejich komponenty.

Třídu vytvoříme buďto tak, že klikneme na symbol třídy (class) v zásobníku objektů v okně vlevo dole, nebo klikneme pravým tlačítkem na třídním diagramu a třídu přidáme. Jednotlivé vlastnosti je pak možné přidat do vybrané třídy v jejím kontextovém menu. Vlastnosti jsou v StarUML označovány jako atributy (atributes), metody jsou pak označovány jako operace (operation).

Pro vlastnosti a metody lze nastavovat řada vlastností. Nastavení se provádí v Editoru (v okně vpravo dole). Nejdůležitější jsou následující vlastnosti:

- name - jméno vlastnosti nebo metody, jméno se nesmí v rámci jedné třídy opakovat
- streotype - datový typ vlastnosti (např. INT - celočíselný datový typ) - pokud je ale účelem tvorby UML pouze pochopení problému, je možno stereotyp vynechat
- visibility - viditelnost je ale naopak potřeba za každé okolnosti vyplnit. Podporovány jsou následující nastavení
  - public - veřejný, vlastnost nebo metoda je veřejně přístupná k užití dalšími třídami (symbol +)
  - private - soukromý - vlastnost nebo metody je dostupná pouze v metod uvnitř dané třídy, nikoliv tedy z vnějšku (symbol -)
  - protected - chráněný - vlastnost je přístupná z vnějšku, ale pouze pro metody tříd, které se nacházejí ve stejném jmenném prostoru (symbol #)

Symboly viditelnosti jsou zobrazovány také přímo v třídě před názvem vlastnosti, popř. metody. Nejčastěji přitom používáme typ veřejný nebo soukromý.

Metody mají pro náš účel spíše podpůrný charakter – nebudeme realizovat dynamické chování třídy v nějakém programovacím jazyce. Uvažovaní nad specifikací metod nám však může napomoci ve správném zachycení procesu řešení problému.

Rozlišujte pečlivě mezi třídou a instancí třídy
Třídu je potřeba chápat jako obecný pojem popisující určitou skupinu (třídu) objektů, instance třídy je popisem konkrétního výskytu třídy. Zkusme si to demonstrovat na příkladu: třída - pes, instance třídy - Váš Alík.

Třídy si tak lze představit jako šablony, které vyplňujeme jednotlivými instancemi třídy. Při tvorbě třídního digramu se obvykle soustředíme právě na třídy a instance tříd pak obvykle do třídního diagramu nedáváme. Třídní diagram tedy řešíme obvykle obecně, tedy nespecifikujeme instance třídy.

Za určitých okolností však může být výhodné toto pravidlo porušit a zachytit v tomto diagramu i instance třídy a to zejména v okamžiku, kdy popisujeme konkrétní situaci, s konkrétními objekty (resp. instancemi objektů). Získáme tak třídní diagram tvořený instancemi tříd, který následně můžeme zobecnit, pokud je to potřeba, do podoby třídního diagramu tvořeného třídami.

Kromě tříd v tomto druhu diagramů modelujeme i vazby mezi nimi. Tyto vazby nabývají podob asociací, generalizací a agregací. Pro grafické značení viz. obr. 5.3.

Obrázek 5.3: Typy asociací podporované UML diagramy

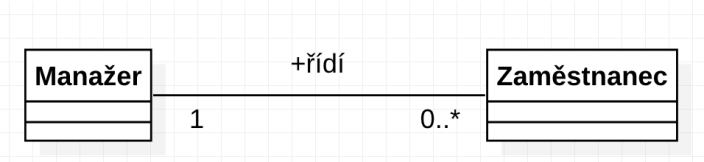
Běžné vazbě mezi třídami říkáme **asociace**. Asociaci je možno pojmenovat, abychom usnadnili orientaci v diagramu. Pokud asociaci pojmenujeme dáváme jméno přibližně doprostřed mezi propojované třídy.

U vazeb specifikujeme také četnost (kardinalitu). Kardinalita se specifikuje pro obě zakončení vazby. Možnosti kardinality:

- 1 (nebo vynecháno) – na takto označeném konci vazby musí existovat jedna instance třídy
- 1..* - na takto označeném konci musí existovat minimálně jedna instance třídy
- 0..3 – na takto označeném konci vazby musí existovat maximálně 3 instance třídy

Příklad použití naleznete na obr. 5.4. Na tomto obrázku je běžná asociace mezi třídou reprezentující managera a zaměstnance, které řídí. Číslo 1 představuje specifikaci počtu - manažer v našem diagramu je pouze jeden. Jednička se však vztahuje pouze k asociaci k zaměstnanci. To znamená, že zaměstnanec (jeho instance) může mít pouze jednoho nadřízeného.

V opačném směru je pak četnost nastavena na 0 .. *, což znamená, že pod managera nemusí nutně spadat žádný zaměstnanec (0), nebo může řídit více (libovolné množství) zaměstnanců (*).



Obrázek 5.4: Asociace mezi třídami

Specifickými případy asociace jsou *orientované asociace* a *závislosti*. Závislost je pravděpodobně nejabstraktnějším typem vazby, kterou v UML diagramu lze využít. Závislost symbolizuje, že závislý objekt využívá služeb objekty, na kterém je závislý. Jinými slovy závislá třída ke své práci potřebuje všechny třídy, na kterých závisí.

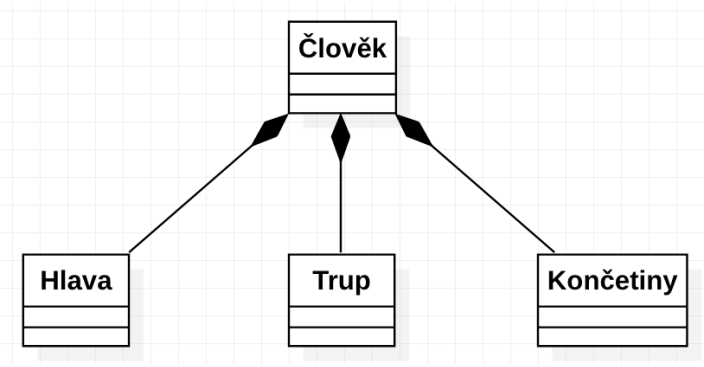
Závislost může být realizována tak, že závislá třída využívá metodu třídy, na které závisí.

Orientovanou asociaci oproti tomu vyjadřuje silnější vazbu. Orientace přitom může být jedním směrem nebo oběma sběry. Příkladem oboustranné orientované asociace je vazba mezi třídou „letová třída“ a letadlo.

Vzhledem k tomu, že nuance mezi různými typy vazeb z pohledu orientace jsou relativně malé, velmi často se používá běžná asociace (bez podrobnějšího rozlišení) pro zachycení všech vazeb tohoto typu. Někdy se dokonce běžná asociace používá jako substitut pro kompozici/agregaci, byť v takovém případě už jsou rozdíly ve významu vyšší.

Vazby typu agregace a kompozice zachycují situaci, kdy jeden objekt „vlastní“ jinou třídu. *Kompozice* je na ilustraci jednodušší. Mějme třídu člověk, která se skládá ze tříd hlava, tělo, ruce nohy. Pokud zrušíme třídu člověk nebudou mít ostatní třídy smysl. Podobná situace nastane u tříd budova a místnosti - pokud nebude existovat budova, nemá místnost smysl.

Příklad kompozice je dostupný na obr. 5.5.



Obrázek 5.5: Příklad použití kompozice v třídním diagramu

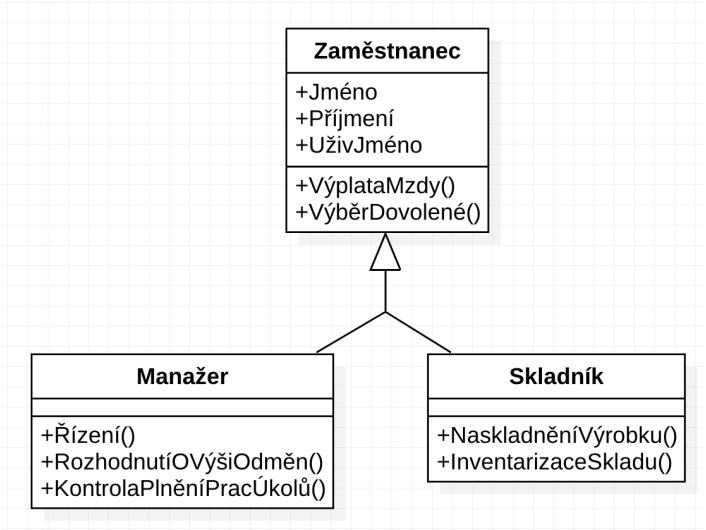
Agregace je z tohoto pohledu vazbou volnější. Uvažujme třídy výbor a členové výboru. Lze říci, že výbor se skládá ze členů výboru. Avšak členové výboru mají, nebo spíše mohou mít smysl i bez výboru. Příklad agregace je dostupný na obr. 5.6.

Mimochodem způsob implementace agregace a kompozice se v různých verzích jazyka UML liší. Např. verze 1.4 je chápe jako samostatné typy asociací. Verze 2.0 standardu ale bere v úvahu pouze základní typ asociace a agregaci resp. kompozici je možno nastavit jako vlastnost zakončení asociace - tedy specifikací způsobu napojení asociace na třídu.

V různých CASE nástrojích se tak může lišit způsob vytváření těchto asociací, byť z hlediska interpretace výsledného diagramu ke změně nedošlo.

Konečně *generalizace, resp. dědičnost* umožňuje definovat úzkou souvislost mezi dvěma třídami. Uvažujme situaci na obr. 5.7.

Obrázek 5.6: Příklad použití agregace v třídním diagramu



Obrázek 5.7: Příklad použití generalizace/dědičnosti v třídním diagramu

Jak manager tak skladník z obr. 5.7 jsou zaměstnanci, ale vzhledem k tomu, že jejich náplň práce je výrazně odlišná bude se do určité míry struktura tříd, které je vyjadřují lišit. Zároveň ale, jelikož obě osoby jsou stále zaměstnanci, budou mít také některé společné vlastnosti a metody.

Generalizace, česky zobecnění nám umožňuje tuto situaci řešit efektivně tak, že zobecníme konkrétní třídy a jejich části, které se opakují napříč třídami a osamostatníme je do samostatné třídy, kam vše opakující se napříč třídami přesuneme. V našem případě do třídy zaměstnanec. V samostatných třídách ponecháme pouze ty údaje, kterými se původní třídy před zobecněním lišily.

Lze také říci, že manažer i skladník (potomci) dědí metody a vlastnosti z třídy zaměstnanec (rodič). Z pohledu interpretace to znamená, že všechny veřejné nebo chráněné vlastnosti a metody se přenášení (jsou zděděny) do tříd potomků.

Generalizační vazby tedy mohou výrazným způsobem zjednodušit struktury tříd. V praxi se pak setkáváme s tím, že se vytvářejí celé hierarchie tříd. Tento typ vazby je tak v praxi velmi používaný.

5.4 Model jednání (Use Case Diagram)

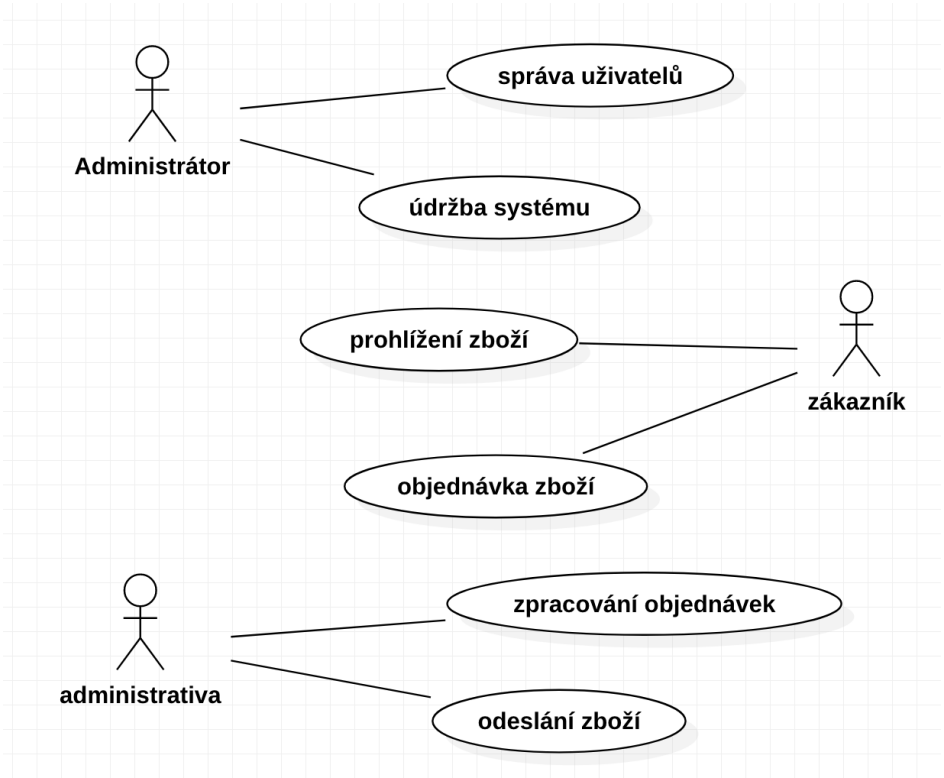
Úkolem diagramu je specifikovat modelové (nosné) způsoby využití analyzovaného systému. Diagram využívá tři základní konstrukty - uživatel (Actor), scénář/případ užití (Use Case) a vazby/asociace mezi nimi.

již na první pohled je tak účel tohoto diagramu jiný než v případě třídního diagramu. Zatímco u třídního diagramu jsme se zaměřili na hluboké pochopení struktury zájmových objektů (tříd), v případě modelu jednání se zaměřujeme spíše na pohled zvenčí a specifikaci různých scénářů, které považujeme z pohledu činnnosti analyzovaného systému za nosné.

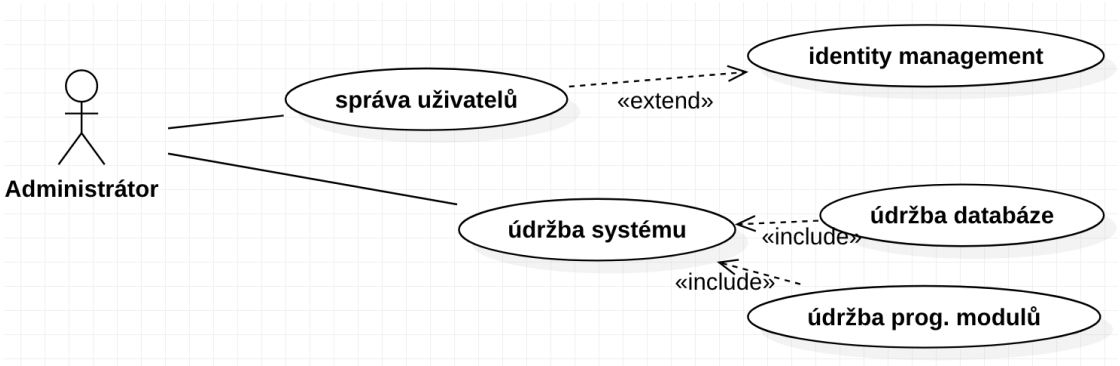
Na obr. 5.8 je dostupný zjednodušený příklad modelu jednání pro elektronický obchod.

Notace diagramu je tedy velmi jednoduchá - uživatel je znázorněn symbolem panáčka, ovál s popisem uprostřed je pak specifikovaný scénář (use case). Asociace je pak reprezentována prostou čarou mezi uživatelem a scénářem.

Vazba však nemusí být nutně takto jednoduchá. Podívejme se podrobněji na administrátora a jeho interakci s analyzovaným systémem e-shopu, viz obr. 5.9.



Obrázek 5.8: Příklad použití modelu jednání - realizace elektronického obchodu



Obrázek 5.9: Rozšíření modelu jednání

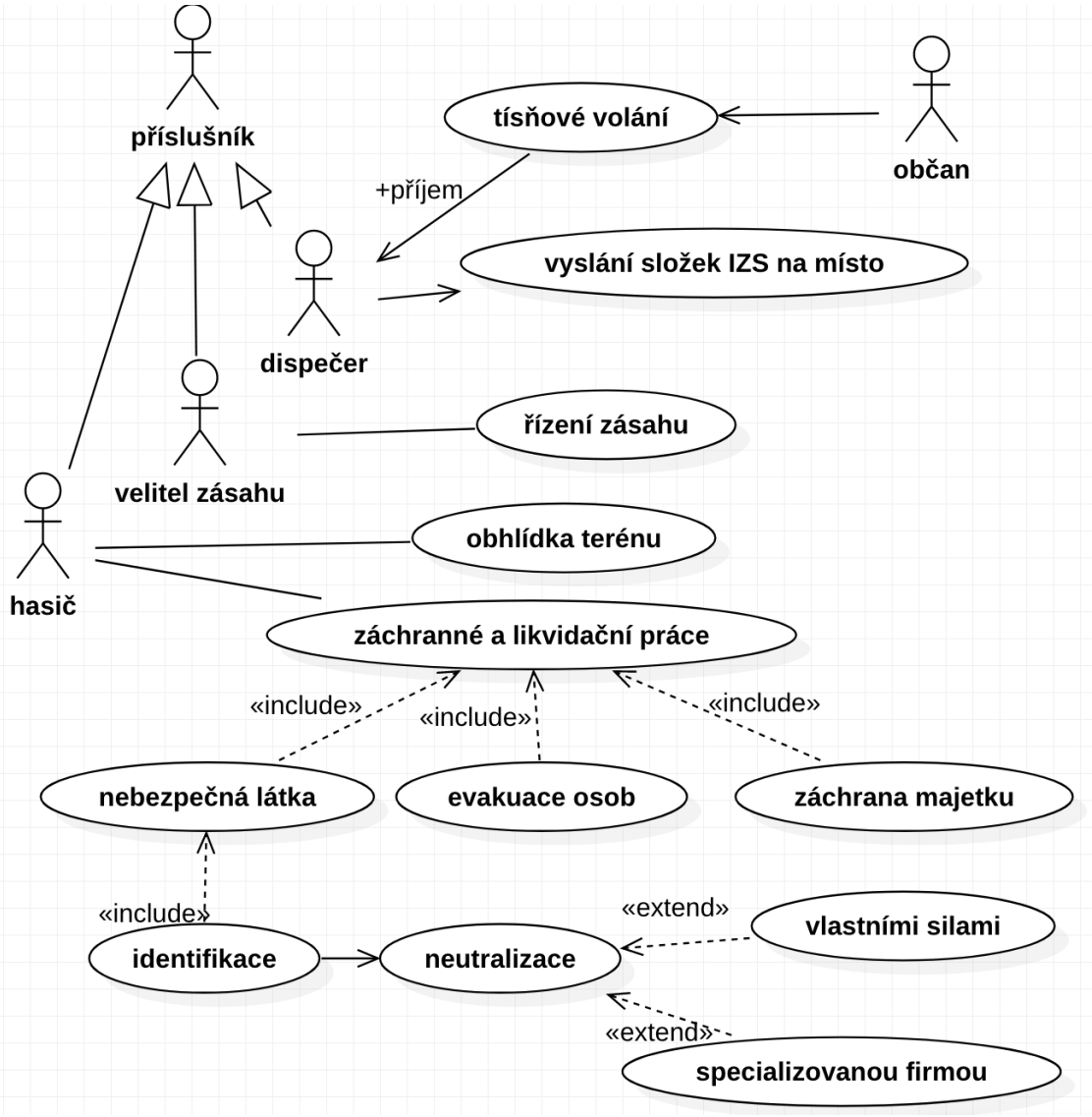
Všimněte si nových scénářů a způsobu, jak jsou navázány na scénáře původní. Např. původní scénář správy uživatelů rozšiřuje (extend) obecnější scénář identity managementu. Prakticky bychom to mohli chápat tak, že organizace má implementován obecný systém řízení identit a správa uživatelů pak tento systém rozšiřuje o některé specifické údaje o uživatelích, které jsou nutné pro provoz e-shopu.

V případě scénáře údržby systému je situace opačná. Pracujeme se dvěma scénáři a to údržbou databáze a údržbou programových modulů. Údržba jako celek zahrnuje oba tyto scénáře - proto je zde vazba typu vložit (include).

Kromě výše uvedených vazeb je možno také používat asociace typu závislost, orientovaná asociace a generalizace. Tyto typy vazeb používáme stejně jako v případě třídního diagramu, ovšem s tím, že pracujeme se scénáři místo tříd.

Zkusme použít získané znalosti u sestavit jednoduchý model jednání pro případ zásahu HZS s přítomností nebezpečných látek. Samozřejmě se bude jednat o zjednodušený pohled, viz obr. 5.10.

V diagramu máme celkem pět účastníků. Občan má asociovaný pouze jeden scénář a to tísňové volání, poté, co zjistí nehodu třeba cisterny převážející neznámou látku. Tísňové volání je přijato dispečerem, který získá potřebné informace od volajícího a vyšle složky IZS na místo.



Obrázek 5.10: Příklad modelu jednání pro zásah s přítomností nebezpečných látek

Dispečer společně s velitelem zásahu a běžným hasičem jsou příslušníky HZS, což je na diagramu znázorněno pomocí generalizačních vazeb.

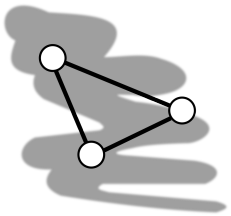
Činnosti velitele zásahu jsou zde pro zjednodušení pouze naznačeny. K veliteli je proto asociován pouze jeden scénář a to řízení zásahu.

Oproti tomu pro hasiče je vytvořen základní scénář záchranných a likvidačních prací, který obsahuje (include) další scénáře pro evakuaci osob, záchranu majetku a konečně vypořádání se se samotnou nebezpečnou látkou. Evakuace a záchrana majetku již dále nejsou rozpracovávány. Zlikvidování nebezpečné látky ale vyžaduje realizaci dalších podřízených scénářů.

Nejprve je nutno látku identifikovat, tak aby bylo možné navrhnout optimální postup neutralizace látky a to buďto vlastními silami nebo s použitím specializovaných firem. K tomuto účelu se v praxi využívají informace obsažené v systému Transportní informační a nehodový systém (TRINS). Tak hluboko ale v rámci představované analýzy nepůjdeme.

5.5 Stavový diagram (State Diagram)

Stavový diagram umožňuje zachytit časové změny v modelovaném systému. Model jednání i třídní diagram byly z hlediska notace čistě statickými nástroji. Tedy oba diagramy se dívají na určitý snímek



Návaznosti - model jednání
V diagramu identifikované scénáře nezůstávají obvykle pouze v tomto diagramu, ale jsou postupně rozpracovávány pomocí dalších diagramů tak, aby byl získán ucelený celkový obraz o analyzovaném systému.

Prostředkem, který k tomuto účelu často používáme jsou stavové diagramy, digramy činností nebo diagram scénářů činností, které mají schopnost podrobněji specifkovat postup činností uvnitř takového scénáře.

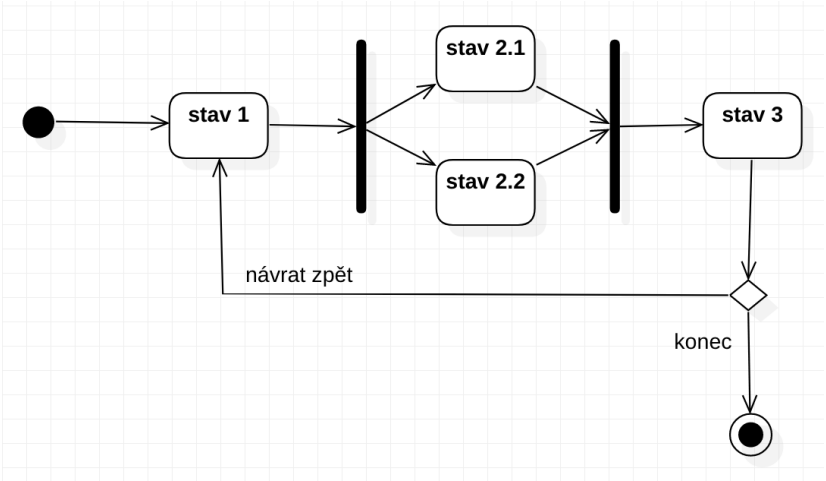
reality a na základě něj odvozují strukturu systému (třídní digram) nebo jednotlivé scénáře zájmových činností systému.

Jak jsme ale viděli v předchozí kapitole - model jednání se omezuje pouze na identifikaci scénáře, ale není schopen zachytit průběh scénáře samotného. Z tohoto důvodu potřebujeme další nástroje (diagramy) pro podrobnější specifikaci takových postupů.

Stavový diagram zavádí do analýzy časové hledisko pomocí sledování změn stavů v jednotlivých objektech a vzájemných vazeb mezi těmito stavy.

Diagram, vzhledem k tomu že je časově vymezen, musí obsahovat začátek a konec. Jednotlivé stavy mohou podobně jako třídy obsahovat stavové proměnné (odpovídá vlastnostem třídy) a činnosti (odpovídá metodám třídy). Podobně jako u tříd existuje i zjednodušená notace, která obsahuje pouze název stavu. Tento zkrácený zápis budeme nadále používat pro zápis stavových diagramů v těchto skriptech.

Grafickou notaci stavového diagramu si můžete prohlédnout na obr. 5.11.



Obrázek 5.11: Stavový diagram - struktura modelu

Všimněte si, že vzhledem k tomu, že činnosti zachycené stavovým diagramem jsou časově vymezeny, jsou spojení mezi stavy vždy orientovaná.

Všimněte si také symbolů přechodu (svislé čáry ze kterých nebo do kterých jde více toků.) Stavový diagram tak podporuje zachycení případného paralerního průběhu prováděných činností. Touto schopností se liší od běžných vývojových diagramů.

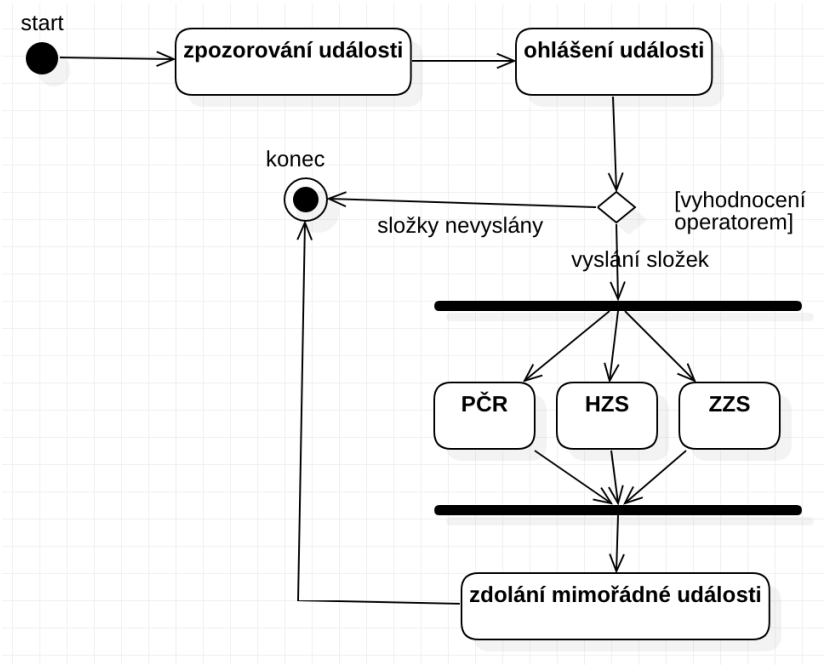
Symbol kosočtverce pak umožňuje zachytit bod rozhodování.

Funkci stavového digramu můžeme demonstrovat na jednoduchém příkladu, viz obr. 5.12.

Na příkladu máme znázorněn schématický postup od zpozorování události až do okamžiku jejího zdolání. Diagram začíná zpozorováním mimořádné události občanem. Ten pak událost ohlásí na tísňovou linku. Operátor rozhodne o tom, zda je volání oprávněné. Pokud ne pouze hlášení přijme, ale celý proces končí. V opačném případě na místo vyšle složky Integrovaného záchranného systému, o kterých předpokládá, že budou nutné pro zdolání události.

Složky pak na místě vykonávají záchranné práce až do okamžiku zdolání události, kdy celý proces končí.

Očividně se jedná o zjednodušený proces, ale měl by nám postačovat pro vytvoření představy o použití jednotlivých komponent diagramu.



Obrázek 5.12: Stavový diagram - schématický příklad zdolání mimořádné události

Elegantnější vývojový diagram
Alternativou stavového diagramu jsou diagramy vývojové. Tyto diagramy nejsou součástí UML (jsou starší), ale používají se v praxi i dnes a to i pro dokumentaci procesů. Krátký tutoriál k jejich použití naleznete např. https://popelka.ms.mff.cuni.cz/~lessner/mw/index.php/U%C4%8Debnice/Algoritmus/V%C3%BDvojov%C3%A9_diagramy. Diagramy činností ale proti vývojovým diagramům mají řadu výhod:

- mají menší množství konstruktorů (používaných symbolů), které jsou univerzálnější
- jsou proto jednodušší na naučení a rychlejší pro realizaci
- podporují paralelní zpracování, které se běžně používá - činnosti tak mohou běžet paralelně vedle sebe
- integrován do CASE nástrojů
- diagramy činností jsou proto mnohem šířeji uplatnitelné

5.6 Scénáře činností (sequence diagram, sekvenční diagram)

Diagram scénářů činností, někdy též označovaný jako sekvenční diagram nám umožňuje vnést do analýzy systému časové hledisko. Tuto možnost do určité míry poskytují také stavové diagramy. Stavový diagram umožňuje zachytit časovou posloupnost jednotlivých aktivit, to ale často nestačí pro účely analýzy, zejména v okamžiku, kdy v systému máme velké množství interakci mezi velkým množstvím objektů.

Sekvenční diagram nám umožňuje jednak dokumentovat velmi přehledně interakce mezi objekty v čase a to tak, že každý objekt má vlastní sloupeček (používají se také názvy jako life line nebo swim line). Časově přitom postupujeme směrem z hora dolů. Tím, že každý objekt má vlastní sloupec máme možnost jednoduše vizuálně identifikovat všechny aktivity, které se jej přímo týkají, což je hlavní výhodou tohoto diagramu.

Obvykle se předpokládá, že v okamžiku, kdy začneme tento diagram tvořit, jsou již k dispozici základní třídní diagramy, které nám všechny potřebné objekty definují. V tomto diagramu je proto obvykle pouze používáme, tedy nevymýšlíme nové. Může se samozřejmě stát, že v průběhu analýzy budeme identifikovat nové potřeby, které si vyžádají vznik nových objektů v naší analýze.

V takovém případě bychom se ale měli vrátit k již existujícím modelů a tyto nově identifikované objekty do nich doplnit.

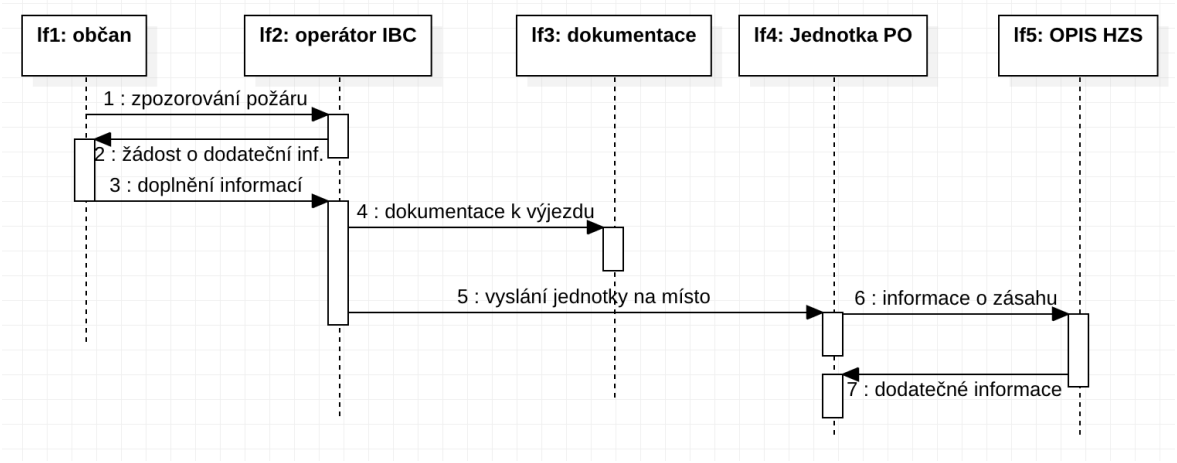
Systém diagramů v jazyku UML, tím že nám umožňuje pohlížet na modelovaný systém z různých pohledů vede k tomu, diagramy jsou velmi úzce propojené. Velmi často tak s nimi pracujeme jako s „živými“ grafy. V průběhu času je tedy postupně měníme, opravujeme, vylepšujeme a přizpůsobujeme našim analytickým potřebám a cílům, které modelováním sledujeme.

Jak přesně tedy sekvenční diagram funguje? Jednotlivé objekty si mezi sebou posílají zprávy. V diagramu lze toto zachytit šipkou. Pro účely programování se pracuje v tomto diagramu s různými typy vazeb umožňujících rozlišit, zda je zpráva zaslána synchronně nebo asynchronně (systém nečeká na odpověď a pokračuje se své činnosti dále). Pro naše účely ale postačuje tzv. *jednoduchá vazba* realizovaná symbolem běžné šipky zachycující běžné předání zprávy bez dalšího rozlišení.

Časově rozlišujeme posloupnost zpráv pomocí posunu šipek směrem dolů ve scénáři. Obvykle lze vysledovat také prvotního původce činnosti, ať už člověka nebo nějaký objekt systému. Pokud se jedná o člověka lze použít symbol aktora (panáčka), který jsme už používali v modelu jednání.

Pro demonstraci použití zkusmě přepracovat předchozí příklad zdolání mimořádné události (obr. 5.12) do podoby scénáře.

Pro tuto transformaci budeme ale muset vytipovat zájmové objekty scénáře. Pro zjednodušení omezme scénář na *požár*. Dále se omezme pouze na zachycení procesu předávání informací zahrnujících složky IZS. Řešení takového scénáře by mohlo vypadat podobně jako je znázorněno na obr. 5.13.



Obrázek 5.13: Scénář činnosti - zjednodušený scénář zdolání požáru

Jak je patrné z obr. 5.13, poskytuje nám scénář odlišnou perspektivu. Předávání zpráv je v tomto případě mnohem přehlednější. Scénář, byť jednoduchý však zabere relativně hodně místa a orientace v něm tak nemusí být úplně prostá. Pokud potřebujeme informaci o komunikaci mezi objekty konsolidovat na menším prostoru, můžeme použít *diagram spolupráce*, ovšem s tím, že se v něm do jisté míry ztrácí přehlednost, minimálně z pohledu časového.

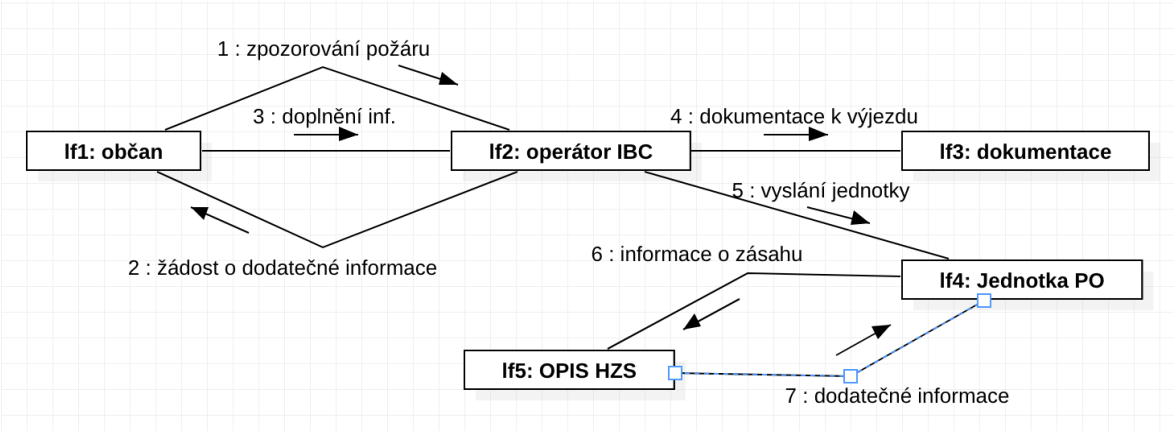
Mimochodem, různé CASE nástroje s podporou UML zachycují scénáře s drobnými odchylkami v notaci. Název *lf* s číslem je zkratkou pro tzv. *lifeline*. Jedná se o přerušovanou čáru vedoucí obr. 5.13 svisle dolů z objektu. Toto označení používá nástroj StarUML, ale řada jiných CASE nástrojů *lf* nebo obdobné označení vynechává a ponechává pouze název objektu jako takového.

Při použití CASE (oproti použití obecného grafického editoru) je výhodou provázanost jednotlivých diagramů. Pokud je v některém z diagramů již použita/definována třída lze tuto třídu přímo použít v diagramu scénáře přetažením z průzkumníka modelu. Tímto způsobem lze tvorbu modelu výrazně zrychlit a také v podstatě vynutit zajištění konzistence v pojmenovávání objektů napříč celým modelovaným systémem.

5.7 Diagram spolupráce (collaboration diagram)

Diagram spolupráce představuje alternativní způsob zápisu zasílání zpráv k sekvenčnímu diagramu. Tento diagram řeší problém velkého prostoru nutného pro zachycení komunikace mezi objekty. Zatímco ve scénáři činností byly objekty seřazeny vedle sebe na horním okraji diagramu a komunikace mezi nimi byla vedena mezi jejich „lifelines“, objekty v diagramu spolupráce mohou být umístěny libovolně. Lifelines ale jsou přímo součástí objektu - nemají vlastní linii a diagram je tak prostorově úspornější.

Sdělovací hodnota obou typů diagramů je stejná. Podívejte se na obr. 5.14 a porovnejte se sekvenčními diagramem na obr. 5.13. Oba diagramy zachycují stejnou situaci a poskytují stejnou informaci. Volba diagramu tak závisí na účelu, za kterým byl vytvářen.



Obrázek 5.14: Diagram spolupráce - zjednodušený scénář zdolání požáru

Podobně jako při tvorbě scénářů lze při použití CASE proces návrhu zrychlit použitím existujících objektů (jejich přetažením z průzkumníka modelu).

5.8 Diagram činností (Activity Diagram)

Diagram činností zachycuje návaznosti mezi jednotlivými činnostmi vykonávanými analyzovaným systémem. Koncepčně je diagram velmi podobný *stavovému diagramu*, ze kterého je také odvozen, poskytuje však trochu odlišný pohled na systém.

Stavový diagram chápal systém jako sled stavů a transformací těchto stavů (činností) ústících do nových stavů systému. Diagram činností se zaměřuje na modelování právě těchto transformací (činností) a zcela pomíjí stavy.

Používané konstruktory jsou stejné včetně použití symbolů pro větvení, nebo rozhodování.

Symboly oválu ale znamenají místo stavu činnost, kterou systém vykonává.

Jednoduchý příklad použití tohoto typu diagramu je na obr. 5.15.

Obrázek 5.15: Diagram činností - zjednodušený průběh činností v jednom dni

Z praktického pohledu, pokud neprovádíme analýzu za účelem přípravy podkladů pro programování rozlišovat příliš mezi stavovým diagramem a diagramem činností nemá smysl.

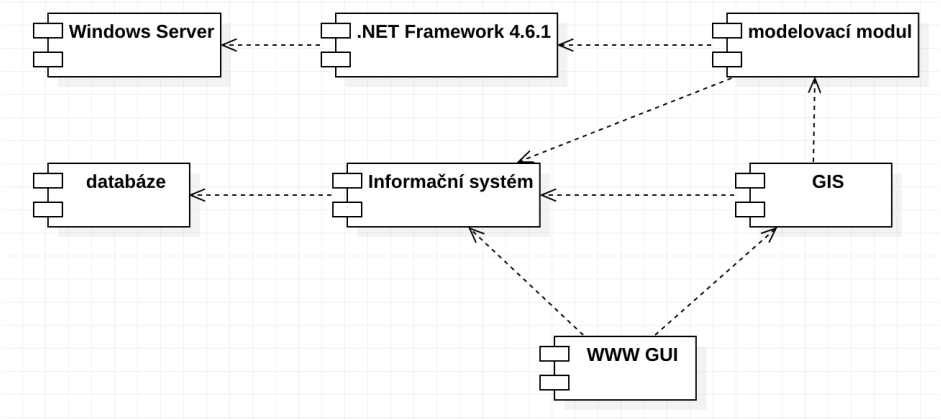
5.9 Diagram komponent (Component Diagram)

Diagram komponent má z hlediska modelování báze znalostí minimální význam. Tento diagram umožňuje modelovat vztahy mezi jednotlivými komponentami systému, ale i komponentami fungujícími

mimo systém, které s modelovaným systémem ale nějak komunikují.

Komponentou přitom rozumíme samostatný softwarový modul (knihovna, program, informační systém, databáze apod.).

Příklad použití diagramu komponent je na obr. 5.16.



Obrázek 5.16: Diagram komponent - zjednodušená vizualizace závislostí komponent informačního systému

Na obr. 5.16 provozujeme informační systém, který využívá databázový backend pro uchovávání zájmových informací a modelovací modul pro modelovaní např. následků mimořádných událostí.

Modelovací modul je realizován nad .NET Framework (např. naprogramovaný v jazyku C#). K provozu je nutný server provozující MS Windows Server, na kterém .NET Framework běží.

Grafické uživatelské rozhraní (GUI) je realizováno pomocí tenkého klienta, který bere informace z informačního systému a vizualizuje je pomocí GIS na mapách.

Příklad prosím berte pouze jako ilustrační.

Výše uvedený příklad řeší IT problém. Konečně UML bylo pro řešení právě takových problémů původně navrženo. To však neznamená, že se právě na tento typ komponent musíme omezit - právě naopak. Pomocí tohoto diagramu lze znázornit vazby mezi jakýmikoliv komponentami - ať už se jedná o softwarové komponenty, hardware nebo třeba součástky nějakého zařízení (např. motoru).

Tento typ diagramu lze použít i pro vizualizaci vztahů v rozsáhlejších systémech jako jsou např. vztahy ve výrobě a distribuci elektrické energie.

Na druhou stranu je potřeba poznamenat, že pokud něco je technicky možné realizovat, je otázka, zda je skutečně nutné to právě takto realizovat. Pokud tedy pro řešený problém (z pohledu komponent) existuje ustálená sada symbolů nebo speciální druhy diagramů - může být z hlediska celkového vzhledu a schopnosti pochopit problém výhodnější použít takový specializovaný diagram.

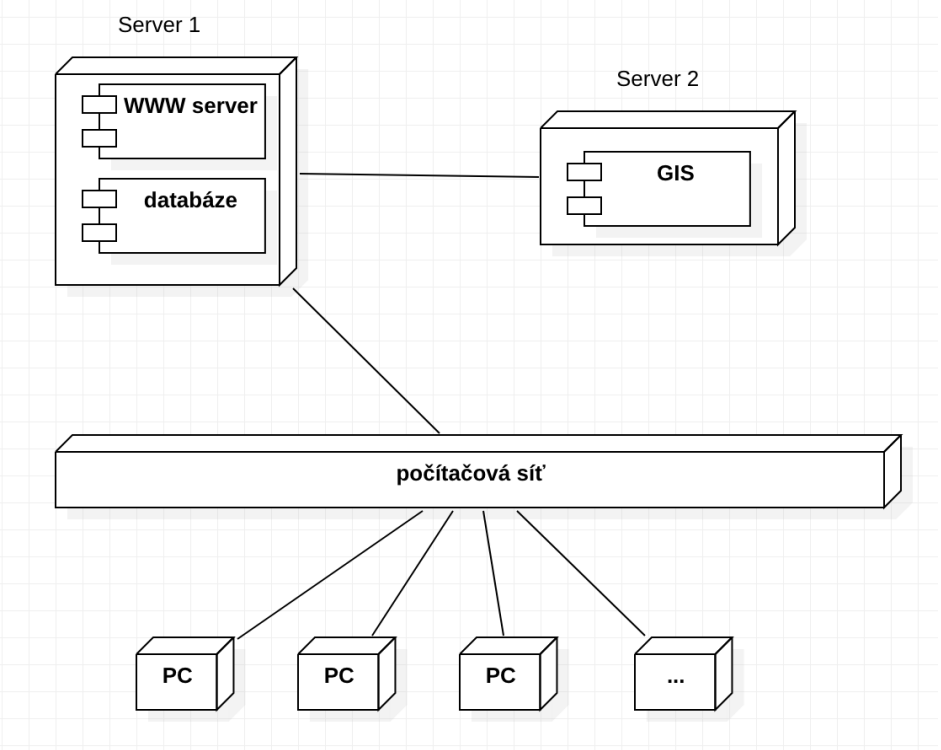
UML tedy není nutné používat za každou cenu. Zároveň je však potřeba poznamenat, že za každou cenu nemusíme znovu a znovu vymýšlet nové typy diagramů pro každý problém, který řešíme, když existující nástroje/diagramy jsou jej schopny vyřešit uspokojivě také. Každý nový diagram má s sebou spojenu určitou režii. Uživatel před použitím, musí pochopit, jak jednotlivé symboly diagramu mají fungovat a stejná znalost se očekává od osob, které budou diagram číst.

5.10 Diagram nasazení (Deployment Diagram)

Diagram nasazení zkoumá operační prostředí, ve kterém bude nasazen modelovaný systém. Umožňuje tak zkoumat které části systému budou na jakých místech a v jakém vzájemném vztahu budou.

K tomuto účelu využívá diagram symboliku diagramu komponent. Komponenty zde mají charakter samostatných systémů, které běží v jednotlivých uzlech obvykle síťového prostředí.

Demonstrujme tento postup na jednoduchém modelu počítačové sítě, viz obr. 5.17.



Obrázek 5.17: Diagram nasazení - zjednodušený diagram nasazení informačního systému z předchozího příkladu

5.11 UML závěr

Modelování pomocí jazyka UML probíhá obvykle iteračně. Různé pohledy na systém reprezentované různými diagramy UML nám umožňují získat komplexní pohled na systém. To nám také umožní, abychom v jednotlivých diagramech odhalili chybějící souvislosti načrtnuté v odlišných druzích UML diagramů (třeba na základě třídního diagramu zjistíme, že v diagramu případů užití chybí podstatný aktor).

Z hlediska procesu modelování se nedá říci, že by některý způsob byl optimální, vždy je nutné vycházet z dané situace, účelu za kterým modelování probíhá. Velmi často se ale během modelování začíná konstrukcí diagramu případů užití, protože tento diagram nám řekne co má modelovaný systém dělat z pohledu z vnějšku. Požadované funkce a jejich vzájemné vztahy můžeme zkoumat v třídních diagramech, diagramech činností apod.

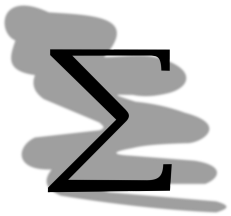
Diagramem nasazení obvykle končíme. Teprve až je celý systém hotov - budeme znát jaké jsou úplné požadavky na nasazení a tyto můžeme zachytit právě v tomto druhu diagramu.

Diagramy by měly být navrhovány tak, aby byly samonosné. Měly by v nich být jasně patrné návaznosti mezi jednotlivými typy diagramů. Zde je potřeba si dát pozor zejména na konzistenci v pojmenovávání objektů, scénářů, ale také diagramů.

Ačkoliv jsou diagramy UML obvykle velmi expresivní, bývá alespoň v některých případech nutné doplnit diagramy o komentáře, někdy i dosti dlouhé a to zejména v případě, že UML používáte pro jiné účely než projektování softwarového systému s jeho předpokládanou následnou implementací (zprogramováním).

Právě v takovém režimu (ne IT aplikace UML) nejspíše budete UML používat také Vy. Postup je pak jednoduchý. Diagramy exportujeme z nástroje, ve kterém vznikly buďto do podoby bitmapy nebo vektorové grafiky a importujeme je do zvoleného textového procesoru jako MS Word, Libre Office Writter nebo řada jiných, kde jednotlivé diagramy opatříme doprovodným vysvětlujícím textem.

V tomto režimu budete pravděpodobně zpracovávat svůj semestrální projekt.



Shrnutí

V této kapitole jsme se seznámili s jazykem UML. Tento grafický jazyk umožňuje přehlednou analýzu různých typů systémů. Různé typy diagramů jazyka jsou pak schopny poskytnout odlišnou perspektivu chování analyzovaného systému.

Jazyk UML je dobře podporován v grafických editorech. Pro serióznější práci ale obvykle preferujeme použití CASE nástrojů. Tyto nástroje jsou schopny zrychlit analýzu schopností znovu využít již definované komponenty analyzovaného systému (které jsme již do CASE zavedli) a obsahuje specializované nástroje pro generování dokumentace apod.

Kontrolní otázky

1. Vyjmenujte alespoň 3 druhy UML diagramů.
2. Jaký je vztah mezi třídním diagramem a diagramem případů užití?
3. Jaký je rozdíl mezi činnostmi a stavy?
4. Jaký je rozdíl mezi kreslícími nástroji (Dia, Visio) a CASE nástroji (StarUML, ArgoUML)?
5. Jaký je rozdíl mezi diagramem spolupráce a scénáři činností?

Kontrolní otázky

1. Zkuste vymyslet případ kdy by bylo vhodnější použít opačné pořadí použití diagramů než je popsáno v závěru této kapitoly.
2. Najděte softwarový produkt pro tvorbu UML diagramů a nainstalujte jej.
3. Zkuste napsat diagram činností popisující Vaše studium tohoto textu. Použijte k zápisu Vámi zvolený softwarový prostředek.

Kapitola 6

Expertní systémy

Náhled kapitoly
Expertní systémy jsou pravděpodobně první oblastí která se začala rozvíjet v oboru, který dnes nazýváme souhrnně *umělá inteligence*. Koncepčně vychází ze snahy formálně popsat způsob rozhodování experta a tento způsob uvažování zachytit a replikovat v uměle vytvořeném systému – *expertním systému*.

Po prostudování této kapitoly budete
vědět

- co je to expertní systém
- jak takový systém funguje
- jakým způsobem je konstruována báze znalostí

Čas nutný ke studiu
Na plné pochopení této kapitoly budete potřebovat přibližně dvě hodiny.

6.1 Základy expertních systémů

Expertním systémem rozumíme software obsahující bázi znalostí odpovídající znalostem experta v určité (specifické) problémové oblasti, který umožňuje konzultovat nebo poskytuje informace k rozhodnutí v problémové oblasti.

První expertní systém, MYCIN, byl vyvinut v roce 1974. Jeho základním úkolem bylo doporučovat vhodné léky na základě analýzy zdravotní karty pacienta, zejména s ohledem na vzájemné působení jednotlivých léků.

Mimochodem, snaha řešit tento problém trvá do dneška. Jedním z posledních pokusů tento problém vyřešit byl např. systém Watson společnosti IBM. Podle dostupných informací [41] sice systém Watson doporučoval stejnou medikaci jako lékaři, zároveň ale v malém procentu případů doporučovat buďto nebezpečné kombinace léků nebo jejich nepřiměřené množství, což bylo považováno za výraznou chybu. Zároveň se neprokázalo, že by doporučení systému byla alespoň v části případů signifikantně lepší nežli doporučení ošetřujícího lékaře. A tak Watsen z tohoto pohledu spíše selhal.

K výše uvedenému je potřeba dodat, že Watson už není klasický expertní systém, ale v určitém smyslu jeho evolucí. Watson je schopen přijímat otázky v přirozeném jazyce a odpovědi odvozuje z předpřipravené báze příkladů, které prohledává.

Vraťme se ale ke klasickým expertním systémům. Od uvedení systému MYCIN se objevila celá řada dalších expertních systémů zaměřených na jiné oblasti. Všem těmto expertním systémům říkáme tzv. *plné expertní systémy*.

Plností expertních systémů přitom rozumíme naplnění báze znalostí údaji pro řešení určitého problému. Práce s tímto systémem pak obvykle spočívá převážně v konzultaci se systémem. *Prázdné expertní systémy* obsahují pouze prázdnou bázi znalostí. Od uživatele se očekává, že tuto bázi před použitím naplní.

Kromě tohoto základního dělení můžeme nalézt i jiná dělení těchto systémů např. na *diagnostické, plánovací a hybridní.*

Diagnostické expertní systémy mají za úkol diagnostikovat, tedy na základě zjištěných údajů rozhodnout o nějaké variantě řešení z konečného počtu předem známých řešení.

Představme si to na příkladu systému pro stanovení diagnózy → choroby v okamžiku, kdy do něj vložíme jenom některé choroby – třeba viróza, chřipka a angína. Systém sám pak nebude moci diagnostikovat další choroby, protože je jednoduše nebude znát a není sám schopen odvodit nějaké nové diagnózy.

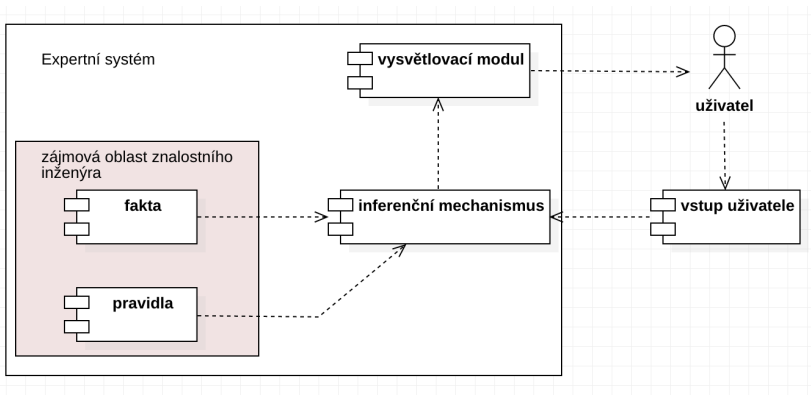
Zde zároveň narážíme na poměrně významné omezení expertních systémů. Pokud budeme diagnostikovat pacienta s jinou chorobou, expertní systém jej zaklasifikuje do jedné z existujících možností, které má. Expertní systém tak neumí dobře pracovat s nejistotami, neznámem apod. což výrazným způsobem omezuje jeho uplatnitelnost v praxi.

Úkolem *plánovacích expertních systémů* je plánování. Pomocí těchto systémů se řeší úlohy, u kterých je znám počáteční stav a cíl řešení. Expertní systém má za úkol naplánovat posloupnost kroků, které k tomuto cíli vedou. Úkolem znalostního inženýra je omezit možnou kombinaci těchto možných kroků a nějakým způsobem zabezpečit ohodnocení příspěvku těchto kroků k dosažení cíle.

Na základě tohoto hodnocení totiž systém může zkonstruovat účelovou funkci, tak aby vybral vhodné řešení. Tady je potřeba podotknout, že vyhledávané řešení není v případě expertních systémů (ale i některých jiných metod se kterými jste se mohli setkat třeba v předmětu Modelování rozhodovacích procesů) optimální, ale **suboptimální** – tedy nejlepší získatelné řešení při akceptaci omezení časem a výkonem výpočetní techniky.

Hybridní expertní systémy v sobě kombinují vlastnosti diagnostických i plánovacích expertních systémů, přičemž obvykle převažuje diagnostická složka.

Expertní systém se skládá z několika základních komponent, viz obr. 6.1. Požadované inteligentní chování vzniká až vzájemnou interakcí těchto modulů.



Obrázek 6.1: Struktura expertního systému

Inferenční mechanismus z obr. 6.1 tvoří jádro expertního systému. Jeho úkolem je zpracovávat údaje zadané uživatelem a porovnávat je s údaji zjištěnými z báze znalostí (tvořené pravidly a fakty). Výsledek této činnosti je předkládán uživateli ve formě vysvětlovacího modulu.

Inferenční mechanismus je v expertním systému zabudován napevno, základním problémem použití tak bude z našeho hlediska formulace báze znalostí, tedy naplnění expertního systému a realizace vysvětlovacího modulu. Oba tyto moduly jsou totiž úzce spojeny s řešeným problémem a musíme je proto pro každý takový problém navrhnout, často z nuly.

Na bázi znalostí máme přitom několik požadavků. Předně budeme potřebovat, aby tato báze byla rozšiřitelná, tedy abychom do ní podle potřeb mohli doplňovat nová fakta a pravidla. Jazyk, ve kterém budeme tyto pravidla musí být formální a musí nám umožnit popsat všechny objekty problémové oblasti a vztahy mezi nimi. Proces určování těchto objektů a vztahů mezi nimi nazýváme *konceptualizace.*

Pro grafický popis tohoto procesu můžeme použít zápis pomocí nám už teď známého jazyka UML. Z hlediska potřeb expertního systému je přímo použitelný třídní diagram.

Formální zápis univerza získaného konceptualizací problému nazýváme *ontologie*. Existuje celá řada jazyků, které nám umožní formálně vyjádřit ontologii jako rámec problému a znalosti, jako např. jazyky KIF [35], CycL [27], které se k tomuto účelu používaly tradičně. Z moderních jazyků pro zachycení ontologií lze doporučit např. OWL 2 [82].

Tyto jazyky ale nejsou obecně použitelné. Proto výběr výrazového prostředku úzce souvisí s volbou expertního systému a toho, jaké jazyky podporuje.

Z konstrukce báze dat nám také vyplývá omezení z hlediska problému, které jsou expertní systémy schopné řešit. Pokud problémovou oblast nejsme schopni formálně vyjádřit ať už vůbec nebo pouze obtížně, není tento problém řešitelný pomocí expertních systémů.

Pomocí expertních systémů také není vhodné řešit problémy které naopak velmi dobře známe a jsme je schopni explicitně zachytit prostřednictvím formálních modelů např. vzorcem, soustavou diferenciálních rovnic apod. V takovém případě je výhodnější řešit konvenčními metodami – poskytují rychleji přesnější výsledky.

Ontologie se obvykle nevytvářejí ručně, ale s pomocí specializovaného software. Jedním z nejpopulárnějších nástrojů pro tvorbu ontologií je Protégé [75].

Z hlediska tvorby rozsáhlejšího expertního systému se obvykle doporučuje začít buďto s existujícím (naplněným) expertním systémem nebo tvorbou ontologie řešeného problému.

My se z prostorových důvodů omezíme na zjednodušenou definici bázi znalostí ve smyslu jednotlivých komponent této báze a jejich použití v expertním systému CLIPS.

6.2 Expertní systém CLIPS

Vývoj tohoto expertního systému započal v roce 1985 v NASA nicméně záhy se výzkum v NASA přestal orientovat na vývoj expertních systémů a tak byl tento systém uvolněn včetně zdrojových kódů pro užití široké veřejnosti.

Vývoj CLIPS probíhá na domácích stránkách projektu: `http://www.clipsrules.net/` [1].

S postupem času se objevila řada navazujících projektů, které filozoficky vyšly z konceptu CLIPS a rozvinuly ho odlišným směrem. Příkladem takového rozvoje je třeba systém Drools vyvíjený společností Red Hat jako Business Rule Management System (BRMS). Účelem systémů BRMS je zjednodušit zavedení a údržbu automatizace komplexní „bussiness“ logiky automatizovanou obsluhou systémů v organizaci.

Ale zpět k systému CLIPS. Zkusme pro tento systém nadefinovat jednoduchou bázi znalostí umožňující identifikaci nebezpečné látky podle informace na kontejneru. Tento problém je efektivněji řešitelný bez použití expertního systému prostým použitím některé z existujících databází nebezpečných, jeho výhodou ale je to, že si jednotlivé komponenty problému dokážeme poměrně dobře představit, aniž bychom je museli extenzivně definovat v tomto textu. Můžeme se tak soustředit na bázi znalostí a způsob její tvorby.

V našem případě bude bázi znalostí tvořit:

- šablony,
- fakta a
- pravidla.

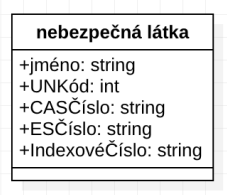
6.2.1 Šablony CLIPS

Šablony umožňují formálně zachytit strukturu faktů v bázi znalostí. To, co jsme definovali v předchozí kapitole v *třídním diagramu* v jazyku UML nebylo z tohoto pohledu nic jiného než šablona.

Uvažujme jednoduchý příklad: chceme vytvořit expertní systém schopný identifikace nebezpečné látky na základě zadání některých informací z jejího přepravního obalu.

Třídní diagram v tomto případě bude jednoduchý - bude tvořený pouze jedinou třídou, viz obr. 6.2.

Na obr. by samozřejmě mohlo být použito také podstatně větší množství informací o látce a mohly by být uvažovány další objekty, pro jednoduchost ale ponechme v budoucí bázi znalostí pouze tuto jedinou třídu.



Obrázek 6.2: Třídní diagram - třída nebezpečná látka

Šablona výše uvedené třídy v tzv. COOL syntaxy systému CLIPS vypadá následovně:

Listing 6.1: Šablona informace o nebezpečné látce

```
(deftemplate NebezpecnaLatka "informace o nebezpecne latce"
        (slot jmeno (type STRING))
        (slot UNKod (type NUMBER) (range 1 9999))
        (slot CASCislo (type STRING))
        (slot ESCislo (type STRING))
        (slot IndexoveCislo (type STRING))
        ;pripadne dalsi atrinbuty nebezpecnych latek
)
```

Několik pravidel práce s CLIPS. Existuje celá řada verzí CLIPS. Základní verze pracuje z příkazové řádky. Aby bylo možné odlišit, co je a co není součástí jednoho příkazu, je nutné všechny příkazy obalit do kulatých závorek.

Tyto příkazy je možné zadávat z příkazové řádky, ale to je samozřejmě poměrně nepohodlné, proto se velmi často šablony i pravidla definují mimo prostředí CLIPS v nějakém textovém editoru. Pro tyto soubory se často používá přípona .cls.

Příkazy jsou citlivé na velikost písmen. Jednotlivé příkazy jsou obvykle psány malými písmeny, pro datové typy se používá písmo velké.

Nyní k samotné šabloně, příkazem pro definici šablony je *deftemplate*, za kterým následuje jméno šablony, v uvozovkách je specifikován vysvětlující text, tak aby byla zjednodušena orientace ve zdrojovém kódu šablon.

Pozn. na okraj - CLIPS nevadí interpunkce, ale systému, ve kterém jsou sázena tato skripta ve výpisech kódu interpunkce vadí a tak ve výpisech zdrojových kódů v této kapitole interpunkce chybí.

Jednotlivé atributy se definují klíčovým slovem slot, za kterým následuje jméno atributu a naspecifikovaný datový typ atributu. Pro atributy, tam kde to má smysl, je možné definovat omezení, např. rozsahem. V našem případě je takovým způsobem nadefinováno omezení atributu UNKod, které může být pouze v rozmezí 1 – 9999.

Pro atributy je možné nadefinovat také předvolenou hodnotu, pomocí klíčového slova default. V našem případu to smysl moc nemá, proto příklad předvolby naleznete až zde.

Listing 6.2: Definice atributu v šabloně

```
        (slot ESCislo (type STRING) (default -)
```

V tomto případě je tím předvoleným znakem „-“. Toto samozřejmě nejsou všechny možnosti, které máme při definici šablon, pouze ty nejjednodušší.

Podívejme se na možnosti manipulace s šablonami.

- (list-deftemplates) - vypíše seznam aktivních šablon
- (undeftemplate jménoŠablony) - odebere šablonu z báze znalostí
- (clear) - smazání báze znalostí z paměti
- (save "c:\\cesta\\soubor.clp") - báze znalostí se uloží do souboru: *soubor.clp*
- (load "c:\\cesta\\soubor.clp") - načtení báze znalostí ze souboru: *soubor.clp*

Během načtení dojde k definici nedefinovaných šablon a redefinici (přepsání v paměti) již načtených šablon.

- (ppdeftemplate jménoŠablony) zobrazí na obrazovce obsah šablony, resp. její definici

6.2.2 Definice faktů

Fakta definujeme podobným způsobem jako šablony. Základním příkazem pro tuto editaci je *deffacts*. Můžeme zadávat libovolná fakta podle libovolné šablony. Podmínkou však je, aby tato šablona byla načtena v paměti pomocí *deftemplate*. Je jedno zda je šablona zadána ručně nebo načtena pomocí funkce *load*. (Z praktického pohledu můžeme jednoznačně doporučit, aby šablona byla předpřipravena.)

U faktů již zapisujeme konkrétní hodnoty. Podívejme se na příklad používající šablonu NebezpecneLatky definovanou výše.

Listing 6.3: Definice faktu: základní údaje o nebezpečných látkách

```
(deffacts fakta-nebezpecne-latky "zakladni udaje o neb. latkach"
        (NebezpecnaLatka
                (jmeno "Acefate")
                (UNKod 3018)
                (CASCislo "30516-19-1")
                (ESCislo "250-241-2")
                (IndexoveCislo "015-079-00-7")
        )
        (NebezpecnaLatka (jmeno "Acetaldehyde") (UNKod 1089)
                (CASCislo "75-07-0") (ESCislo "200-836-8")
                (IndexoveCislo "605-003-00-6")
        )
)
```

Ve výpisu jsou nadefinovány dvě látky acefát a acetaldehyd - všimněte si, že obě látky se z hlediska formy zápisu do faktů liší - acefát má na každém řádku vyplněn jeden atribut, pro acetaldehyd jsem ale zápis z důvodu úspory místa konsolidoval. Oba zápisy jsou funkčně ekvivalentní, avšak delší forma zápisu je mnohem lépe pochopitelná (přehledná).

Fakta jsou vždy založena na šabloně. Přitom platí, že šablona v bázi znalostí může být použita libovolně krát. Takže teoreticky naši bázi znalostí o nebezpečných látkách můžeme rozšířit o tisíce (desetitisíce nebo milióny :-) dalších látek, aniž by to expertnímu systému jakkoliv vadilo nebo překáželo.

Při specifikaci tedy nejprve odkážeme jméno šablony, na kterou se odkazujeme (v našem případě NebezpecnaLatka), a potom vypisujeme názvy atributů a k nim přináležející hodnoty.

Poznámka: zarovnání pomocí entrů a použití tabulátorů je pouze pro lepší čitelnost souboru. Teoreticky mi nic nebrání, abychom všechny tyto definice zapsali na jediný řádek. Orientace v takovém prostředí a případné dohledávání chyb by se samozřejmě velmi ztížilo.

Při manipulaci s fakty můžeme používat řadu dalších příkazů:

- (ppdeffacts jmenoFaktu) - zobrazení obsah daného faktu
- (reset) - resetování báze faktů
- (facts) - vypíše seznam všech faktů na obrazovku, tedy napříč všemi definicemi *deffacts*.
- (undeffacts jmenoFaktu) - odstranění definice daného faktu
- (list-deffacts) - vypíše seznam všech definic faktů

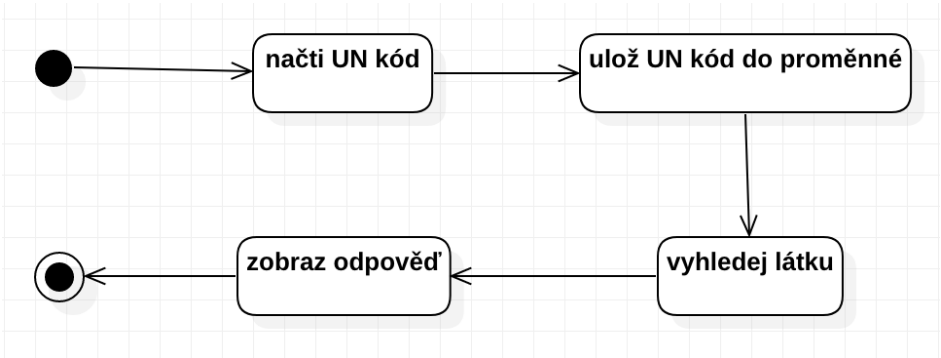
6.2.3 Pravidla

Pravidla obvykle formulujeme ve formě předpoklad → důsledek (příčina → následek). Od pravidel očekáváme, že provedou analýzu zadaných údajů a vyplivnou odpověď. Tomuto postupu (směru odvozování předpoklad → důsledek) říkáme *dopředné řetězení*.

Z hlediska konstrukce jsou pravidla složitější neboť v musí sobě zahrnovat prostor pro otázky – zjišťování údajů od uživatele, a zároveň obsahuje logiku pro zpracování těchto údajů, jejich porovnání s fakty a poskytnutí odpovědi.

Zkusme si nadefinovat jednoduché pravidlo pro identifikaci látky podle UN kódu. UN kód v reálu sice není unikátní identifikátor nebezpečné látky, ale pro naše účely jej za unikátní identifikátor považovat budeme. Proces identifikace látky, který pravidel ošetříme je zobrazen na obr. 6.3.

Proces identifikace je záměrně zjednodušen, pomíjíme možnost, že UN kód není k dispozici a spoustu dalších věcí, takže nám vznikl lineární proces identifikace.



Obrázek 6.3: Activity diagram procesu identifikace látky na základě UN kódu

Podívejme se na první krok tohoto procesu - *načtení UN kódu*. Pro tuto situaci totiž budeme už potřebovat nadefinovat pravidlo. Pro definici pravidla používáme příkaz *defrule*.

Předtím, ale vytvoříme ještě jednu sadu faktů, která nám bude kontrolovat jaké otázky se mají načíst.

Listing 6.4: Pomocna fakta ovlivňující výběr otazky nebo pravidel

```
(deffacts BehProgramu "Pomocna fakta, ktera ovlivni, ktere otazky nebo pravidla
se spusti a ktera ne"
        (otazka-na-unkod 1)
        ; pripadne dalsi otazky
)
```

Pravidlo pro situaci zobrazenou na obr. 6.3 bude vypadat následovně:

Listing 6.5: Načtení UN kódu od uživatele

```
(defrule NactiUNKod "provede nacteni UN kodu"
        ?f <- (otazka-na-unkod 1)
=>
        (printout t crlf "Zadejte UN kod: ")
        (bind ?odp_un (read))
        (assert (UzivUNKod ?odp_un))
        (retract ?f)
        (assert (otazka-na-unkod 2))
)
```

Co nám pravidlo říká? První řádek pravidla začínající ?f provede vyhodnocení údajů v závorce. Jinými slovy systém zkontroluje zda otazka-na-unkod = 1. Pokud pravidlo spouštíme poprvé pak tomu tak skutečně je, protože jsme to tak naspecifikovali ve faktech o pár řádků výše.

Znak => nám odděluje předpoklady od důsledků. Tedy v našem případě, pokud otazka-na-unkod = 1 pak se provedou činnosti specifikované za tímto znakem. Jak vidíte v důsledkové části pravidla máme 4 nové příkazy, podívejme se co znamenají.

Printout vytiskne textový řetězec v uvozovkách na zadaném umístnění, parametr t specifikuje, že tímto místem bude terminál (příkazová řádka). Tento příkaz používáme v okamžiku, kdy je potřeba uživateli něco sdělit. CRLF je substitut za znak konce řádku. Stisknutí enter v tomto případě nestačí, protože enter je v pravidlech považován za tzv. bílý znak a jako takový je ignorován.

Příkaz *bind* provede načtení hodnoty do proměnné. Proměnné v CLIPS se poznají tak, že začínají znakem "?". V našem případě do proměnné ?odp_un načítáme pomocí funkce *read* vstup uživatele z klávesnice. Proměnnou jsme si mohli pojmenovat libovolně, je zde pouze omezení, že jméno proměnné musí začínat písmenem.

Příkazem *assert* provedeme přiřazení hodnoty z proměnné ?odp_un do naší báze znalostí. *Retract* umožňuje odstranit určitý fakt. V našem případě odstraníme fakt že otázka na un kód = 1, protože ji přiřadíme hodnotu 2. Tímto způsobem zajistíme, že dané pravidlo se již při vyhodnocování nespustí.

Ke spuštění pravidel používáme obvykle 2 příkazy a to (*reset*) a (*run*). Příkaz *reset*, resetuje bázi znalostí na základě původních definic. Resetování je nutné pokud báze znalostí byla modifikována

aplikací pravidel. Absence *resetu* může významně ovlivnit způsob chování expertního systému.

Příkaz run provede spuštění pravidel nadefinovaných v systému. Pozor tato pravidla mohou ovlivňovat bázi znalostí. Třeba v našem případě bude vytvořen nový fakt, který bude obsahovat zadaný UN kód. Existenci tohoto nového faktu můžete jednoduše ověřit pomocí příkazu (*facts*) který vypíše všechna definovaná fakta.

Pokud jste náš společný výtvor zkusili spustit, CLIPS se Vás zeptal na UN kód a to je všechno, co systém dělal. Přiznejme si, je to málo. Z tohoto důvodu navrhneme ještě jedno pravidlo, které zpětně identifikuje látku podle UN kódu.

Listing 6.6: Pravidlo pro vyhledání UN kódu v seznamu známých faktů v bázi znalostí

```
(defrule HledejUNKod "Provede vyhledani UN kodu"
        (otazka-na-unkod 2)
        (UzivUNKod ?un)
        (NebezpecnaLatka
                (UNKod ?UNKod)
                (jmeno ?jmeno))
        (test (eq ?un ?UNKod))
=>
        (printout t crlf "Identifikovana latka: " ?jmeno crlf)
)
```

Jak si zajisté všimnete, tak předpokladová část pravidla je poněkud složitější. Prvním předpokladem (otazka-na-unkod 2) je, že proběhlo předcházející pravidlo, kterým jsme načetli od uživatele UN kód.

Další dva předpoklady:

Listing 6.7: Předpoklady pro porovnání se vzorem

```
(UzivUNKod ?un)
(NebezpecnaLatka
        (UNKod ?UNKod)
        (jmeno ?jmeno)
)
```

se starají o tzv. porovnání se vzorem. Toto porovnání probíhá tak, že se procházejí všechna dostupná fakta zavedená v bázi znalostí a porovnají se se vzorem. Hledáme výsledek předchozího pravidla, tedy uživatelem zadaný UN kód, tento výsledek se automaticky uloží do proměnné ?un.

Podobně to funguje s druhým předpokladem, ten načte a do paměti uloží všechny UN kódy a jména nebezpečných látek obsažených v bázi znalostí. Z seznamu těchto látek je však nutno odstranit látky, u kterých neodpovídá UN kód načtený od uživatele. O to se stará poslední předpoklad.

Listing 6.8: otestování faktů na shodu se zadaným UN kódem

```
(test (eq ?un ?UNKod))
```

Tento předpoklad provede srovnání UN kódu načteného od uživatele (?un) a načteného z báze znalostí (?UNKod).

Důsledková část pravidla pouze zobrazí název hledané(-ých) látek. Pokud předpokladům bude vyhovovat více látek provede se vypsání jména pro všechny tyto látky bez nutnosti něco dalšího doplňovat.

6.3 Aplikace expertních systémů v bezpečnosti

Nyní, když už máme určitou představu o způsobu jakým fungují expertní systémy - můžeme začít uvažovat o možných aplikacích do oblasti bezpečnosti.

Lze konstatovat, že expertní systémy jako takové pro řešení bezpečnostních problémů nejsou využívány, zároveň ale na obdobných filozofických principech jsou založeny systémy BRMS o kterých jsme se zmínili již v úvodní části této kapitoly.

Společným znakem expertních systémů a systémů BRMS je dopředné řetězení faktů a pravidla manipulace s nimi, byť systémy BRMS jsou technicky pokročilejší a často v sobě mají zakomponovány další techniky a algoritmy pro manipulaci s daty a odvozování znalostí. Takovými dodatečnými metodami jsou často metody strojového učení, neuronové sítě, fuzzy logika a další.

Tyto dodatečné metody přitom obvykle nejsou přítomny jako BRMS systému, ale jsou realizovány jako samostatné komponenty nebo programy, ke kterým BRMS přistupuje pomocí API. BRMS je v tomto ohledu velmi flexibilní, což ale takovému systému zároveň umožňuje, aby byl také velmi univerzální.

BRMS pak díky této schopnosti přistupovat a manipulovat s externími komponentami může fungovat jako jemné předivo, které integruje jednotlivé informační a analytické systémy v organizacích a umožňuje jim plnit poměrně složité cíle, včetně např. automatizace určitých workflow ve společnostech.

Z tohoto důvodu je BRMS integrováno velmi často do „velkých“ Enterprise Resource Planning (ERP) informačních systémů, jako je SAP a další. Přítomnost pravidlového enginu pak umožní přihnutí funkčnosti takového systému potřebám organizace. Tato možnost je přitom od takových systémů vyžadována, protože jednotlivé společnosti používají obvykle do jisté míry unikátní procesy. Bez přítomnosti BRMS by takový informační systém musel být implementován buďto téměř od nuly pro každou společnost, nebo by se systém procesů ve společnostech musel sjednotit.

První možnost je příliš drahá na to, aby byla realizovatelná a druhá předpokládá, že by se všechny společnosti s ambicí takový systém používat ve svých procesech sjednotily, což taktéž není realistické.

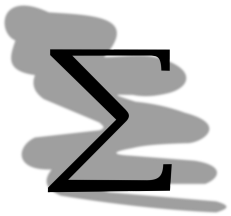
Z pohledu přímých bezpečnostních aplikacích lze uvažovat např. o nasazení takových systémů jako další vrstvy nad stávajícími bezpečnostními systémy jako jsou firewally, Intruder Detection System (IDS) a Intruder Prevention System (IPS), popřípadě firewally nové generace.

Takové řešení se nevyplatí realizovat všem organizacím, ale pouze těm největším, často nadnárodním. Vysoké uplatnění takové systémy nacházejí také u komerčních provozovatelů datových center.

Hlavní překážkou nasazení je především unikátnost každého řešení. Každé řešení je totiž potřeba realizovat na míru dané organizace, jejím procesům, infrastruktuře a také sadě událostí, které mají být detekovány a nastavení způsobu automatické reakce na ně.

Zavedení takového systému vyžaduje obvykle poměrně vysokou počáteční investici a systém samotný bývá zaváděn postupně a dlouhodobě.

V případě, že se ale implementace takového systému povede může takový systém poskytnout solidní základ pro řízení bezpečnosti počítačové sítě i např. procesu oběhu dokumentů.



Shrnutí

V této kapitole jsme se stručně seznámili s *expertními systémy*. Úkolem takových systémů je zpřístupnit v počítači znalosti z vybrané problémové domény v čase potřeby bez nutnosti přítomnosti experta.

Jedním z nejznámějších expertních systémů je CLIPS, ve kterém jsme zpracovali jednoduchý, plně funkční příklad. Pro realizaci báze znalostí o řešeném problému je potřeba připravit zejména:

- šablony,
- fakta a
- pravidla.

V současnosti se v praxi používají spíše BRMS systémy umožňující integraci pravidel do běžného fungování organizace. Finanční a časová náročnost implementace způsobuje, že systémy provozují zejména velké, nadnárodní organizace - popř. organizace poskytující komplexní služby většímu množství zákazníků (banky, provozovatelé datových center ale třeba také univerzity apod.).

Implementace příkladu v CLIPS
Stáhněte a nainstalujte expertní systém CLIPS a implementujte příklad z této kapitoly.

Do příkladu doplňte několik dalších nebezpečných látek.

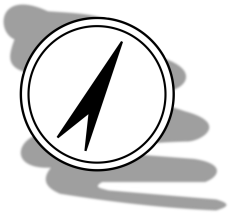
Zvažte jakým způsobem by expertní systém bylo možné dále rozšířit, aby poskytoval lepší výsledky a svůj nápad se pokuste v systému zrealizovat.

Kontrolní otázky

1. Definujte pojem expertní systém a jaký je jeho účel?
2. Co jsou fakta a jak je používáme?
3. Jak se liší šablony, fakta a pravidla?

Kapitola 7

Celulární automaty



Náhled kapitoly
V této kapitole se podíváme na možnosti použití metody umělého života na bázi celulárních („buněčných“) automatů pro modelování některých dějů. V rámci společného studia budeme experimentovat se jednoduchým programovacím jazykem Logo, pomocí kterého, lze tyto automaty ovládat.

Po prostudování této kapitoly budete
vědět

- co jsou to celulární (buněčné) automaty
- k modelování jakých problémů je možné je využít
- jakým způsobem probíhá programování automatu

Čas pro studium
Na prostudování této kapitoly budete potřebovat dvě hodiny.

7.1 Počátky buněčných automatů – Conwayova hra života

Celulární automaty (buněčné automaty, cellular automata) jsou jednou z aplikací umělého života. Jedná se o dynamické modely, které jsou diskrétní v čase a jejich prostor je rozděleno do samostatných částí, které označujeme jako *buňky*. Odtud také pochází název tohoto typu algoritmů.

Diskrétností v čase rozumíme to, že simulace probíhá po samostatných, v čase oddělených iteracích - krocích.

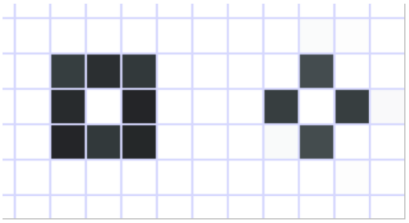
Jednotlivé buňky v modelu, jednak žijí vlastním životem v rámci nastavených pravidel, jednak působí na buňky ve svém okolí (opět podle určitých pravidel). Iterace pak probíhá na realizací tzv. *lokální přechodové funkce*. Iterace tedy probíhá tak, že se zpracovávají jednotlivé buňky samostatně. Lokální přechodová funkce vezme v úvahu stav buňky a jejího okolí a odvodí nový stav buňky.

Základy této oblasti umělé inteligence položil John von Neuman v 50. letech minulého století (viz [81]). Když vytvořil s použitím tužky a papíru první buněčný automat, který byl schopen samoreplikace. Tento automat byl sestaven z přibližně 200 000 buněk, přitom každá z těchto buněk mohla mít 29 stavů. Ve své době se jednalo o unikátní teoretické dílo, které bylo vzhledem k neexistenci počítačů v dnešním smysly toho slova neověřitelné. Práci na tomto díle přerušila až smrt von Neumana v roce 1957.

Složitost celé práce zapříčinila, že od prvních teoretických pokusů v letech 60. k praktičtějším pokusům uběhla poměrně dlouhá doba. Další podstatný průlom nastal až v roce 1970, kdy vyšel

článek [34] popisující práci matematika John Horton Conway v oblasti celulárních automatů, kterou dnes známe pod názvem *Conwayova hra života*.

Technicky je Conwayova hra života mnohem jednodušší než původní von Neumanův automat. Počet stavů je v tomto případě omezen pouze na dva (von Neuman uvažoval 29 stavů). Takovou konfiguraci můžeme interpretovat např. z pohledu biologie prostě tak že buňka na daném umístnění žije (je) a nebo nežije (není). Univerzum, ve kterém probíhá simulace také nemusí obsahovat sta tisíce buněk, stačí desítky až stovky buněk. Tento druh automatu je složitější pouze v jediném parametru a to uvažováním tzv. úplného okolí buněk, to tvoří 8 buněk oproti von Neumanovu okolí uvažujícímu 4 buňky. Rozdíl mezi oběma okolími je jasně patrný z obr. 7.1.



Obrázek 7.1: Úplné okolí (vlevo) vs von Neumanovo okolí (vpravo)

V Conwayově hře života existují tři možné výsledky iterace:
1. *narození buňky* - buňka se zrodí pokud v jejím okolí jsou právě tři žijící buňky,
2. *přežití buňky* - buňka přežije pokud v jejím okolí jsou 2 - 3 žijící buňky,
3. v ostatních případech buňka zahyne nebo se nezrodí (pokud je vyhodnocovaná buňka prázdná).

Výše uvedená pravidla tvoří lokální přechodovou funkci automatu. Každá buňka se tedy podle pravidel v rámci iterace vyhodnotí a z jejího stavu a stavu jejího okolí se odvodí hodnota (stav) buňky pro další iteraci. Výsledek hodnocení všech buněk se následně použije jako vstup pro další iteraci.

Při experimentování s modelem Conway zjistil, že výsledky simulací je možné zařadit do jedné ze čtyř skupin:
1. *totální zánik* - po uplynutí několika generací se celá plocha úplně vyčistí,
2. vytvoření *stabilního*, v dalších generacích neměnného obrazce,
3. *cyklicky se opakující* obrazce,
4. cyklicky se *opakující obrazce s posunem* (stejný tvar ale posunutí oproti původnímu umístnění).

Dost bylo teorie – podívejme se na vzhled Conwayovy hry života. Pro tuto simulaci jsem zvolil prostředí NetLogo [3], které je volně dostupné ke stažení i s řadou různých modelů včetně Conwayovy hry života. Výsledek simulace je dostupný ná obr. 7.2.

NetLogo budeme později používat pro některé pokročilejší modely, pro případ Conwayovy hry života a experimentování s ní je možno použít také webovou aplikaci `https://playgameoflife.com/` [53].

K průběhu simulace je potřeba doplnit, že hra jako taková je nekonečná, byť může dospět do stavu kdy se hra s přibývajícími iteracemi již nemění.

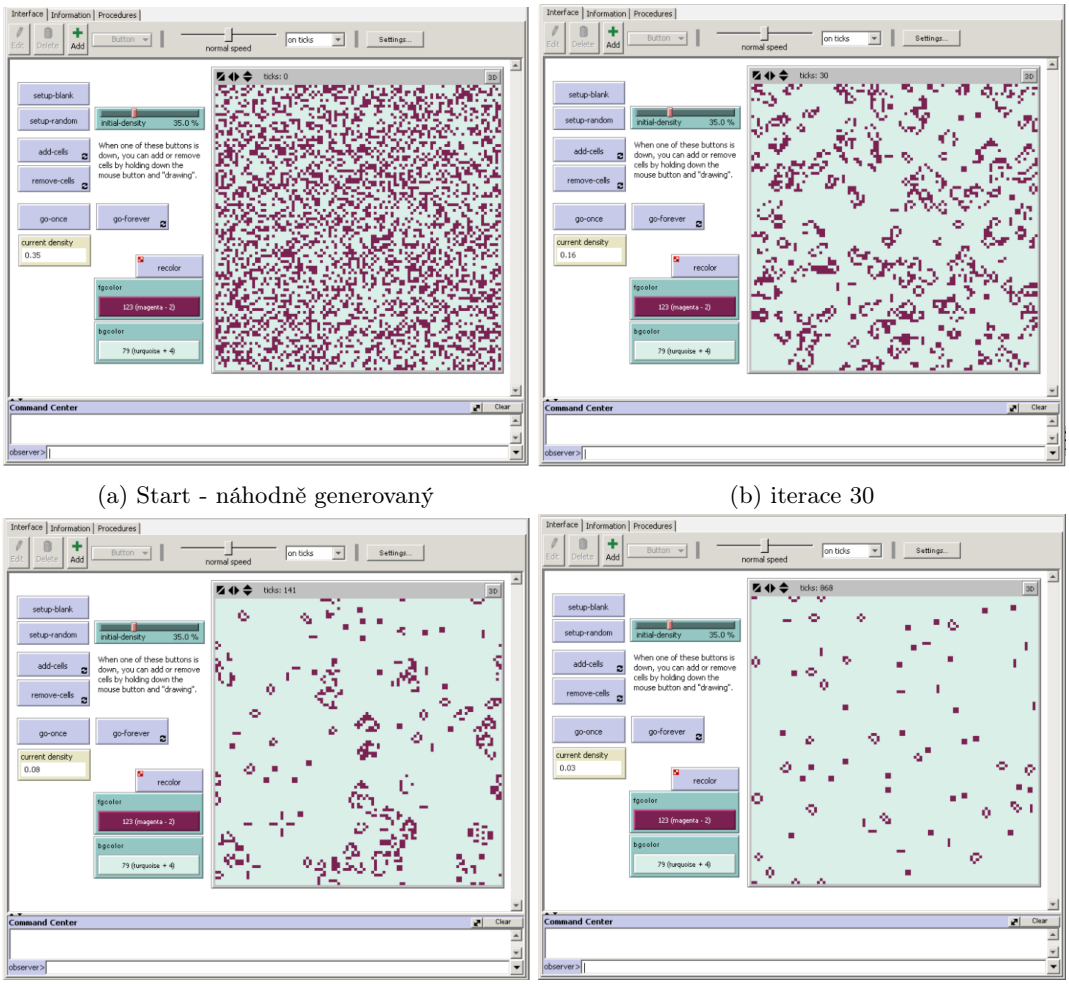
Tento model v NetLogu naleznete ve File → Models Library → Computer Science → Cellular Automata → Life.

Pro experimentování s modelem máme k dispozici jednoduché GUI. V prvním kroku nastavíme hustotu modelu (initial density). Hustotou rozumíme, kolik % univerza modelu zaberou živé buňky. Implicitní nastavení hustoty je na 35 %, které postačují pro demonstraci (ale klidně experimentujte s jiným nastavením).

Kliknutím na tlačítko setup-random si vygenerujeme počáteční stav našeho univerza. Simulaci spustíme kliknutím na go-once (každé kliknutí 1 iterace) nebo go-forever (automatické provádění iterací podle nastavené rychlosti).

Přestože jsou pravidla automatu velmi jednoduchá byla vypozorována řada zajímavých aspektů modelu - jako jsou například stabilní struktury, viz obr. 7.3, tzv. oscilátory (periodicky se měnící struktury) na obr. 7.4 a pohybující se struktury, viz obr. 7.5.

Všimněte si, že tzv. pentadecathlon na obr. 7.4e označuje pouze počáteční stav z patnácti. To znamená, že tato struktura se vrátí do svého počátečního stavu po 15-ti iteracích. Všech 15 stavů



(a) Start - náhodně generovaný (b) iterace 30

(c) iterace 141 (d) iterace 868

Obrázek 7.2: Conwayova hra života

nebylo na obr. zavedeno z prostorových důvodů - ale můžete si tuto strukturu naklikat do zvoleného simulátoru sami a životní cyklus struktury podrobně prozkoumat.

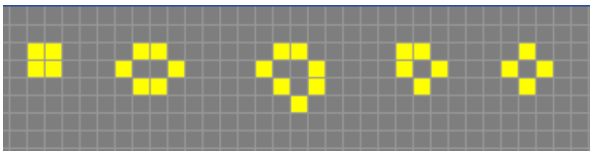
Pohybující se struktury prezentované na obr. 7.5 jsou znázorněny pouze ve svém počátečním stavu - opět je tomu tak z prostorových důvodů, jelikož každá z těchto struktur má 3 a více stavů (period). Pro celou škálu struktur lze použít libovolný simulátor.

Po čtení tohoto textu, prohlížení obrázků umístěných všude okolo Vás možná napadla otázka - k čemu je to dobré? Proč byste zrovna Vy měli uvažovat o takovýchhle věcech? K odpovědi lze přistoupit z několika různých pohledů, já si je dovolím trošku zjednodušeně označit jako pragmatický a akademický.

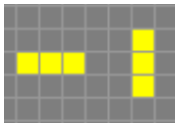
Z pragmatického pohledu, abychom pochopili složitější modely založené na celulárních automatech, potřebujeme pochopit modely jednodušší. V podkapitole věnované bezpečnostním aplikacím celulárních automatům je pak takových příkladů celá řada. Přesto svou aplikovatelností je tento typ modelů značné omezen - postupem času se ale z něj vyčlenila v současnosti samostatná disciplína které říkáme *multiagentní systémy*. Ty jsou složitější, ale na druhou stranu s obrovskou aplikovatelností na řadu problémů bezpečnosti, např. pro simulaci evakuace ať už objektové nebo plošné.

Z akademického pohledu nám studium celulárních automatů umožňuje lépe pochopit fungování dynamických systémů a zejména pak upravit naše očekávání stran toho, jakým způsobem v systémech vznikají stabilní, silně organizované struktury. Při pohledu na strukturu např. sítí (silniční, železniční, jakékoliv jiné) máme tendenci předpokládat, že za vysokou úrovní organizace stojí člověk a do určité míry tomu tak je.

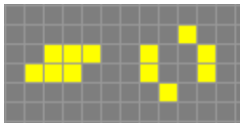
Budování infrastruktur je však ve skutečnosti motivováno zespoda. Pokud např. na zelené louce



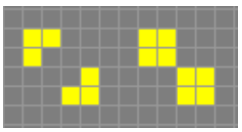
Obrázek 7.3: Conwayova hra života - stabilní (neměnné) struktury zleva: blok, úl, list, loď, vana



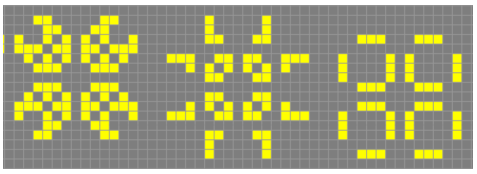
(a) Blinker - perioda 1 a 2



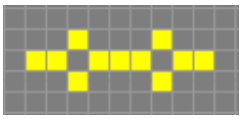
(b) Žába (toad) - perioda 1 a 2



(c) Maják (beacon) - perioda 1 a 2



(d) Pulsar - perioda 3, 2, 1



(e) Pentadecathlon - perioda 1 z 15

Obrázek 7.4: Conwayova hra života - oscilující struktury

chce developer postavit řadu rodinných domů musí po vyřízení veškeré administrativy zajistit tzv. zasíťování pozemků, tedy vybudování přístupové cesty, přivedení elektřiny, vodovodu, kanalizace apod. Zvýšení odběru elektrické energie obyvateli nových domů si může ale vyžádat další investice v rozvodové soustavě města atd.

Vysoce organizovaná struktura tak v řadě případů mohla vzniknout „samovolně“, v průběhu dlouhého časového období, realizací opakovaných drobných/lokálních investic do infrastruktury. Takové „vynoření“ vysoce organizovaných struktur často označujeme jako tzv. *emergence efekt* a procesu kterým takové struktury vznikají *samoorganizace*.

Takové úvahy by nám mohly později napomoci při lepším chápání zákonitostí fungování, rozvoje a bezpečnosti třeba kritické infrastruktury.

7.2 Wolframův 1D buněčný automat

americký matematik Stephen Wolfram začal s buněčnými automaty experimentovat na počátku 80. let. Na rozdíl od předešlých typů automatů však pracuje pouze s jednoduššími tzv. „jednorozměrnými“ automaty.

Graficky takový automat lze zobrazit prostřednictvím jediné řádky. Výhodou také je, že průběh generací můžeme vizualizovat najednou na další řádky, aniž by došlo k přepsání výsledku minulých iterací. Při takovém způsobu vizualizace představuje každý řádek jednu iteraci. Průběhem simulace tak vzniká překvapivě komplexní 2D struktura znázorňující trajektorii vývoje pravidla.

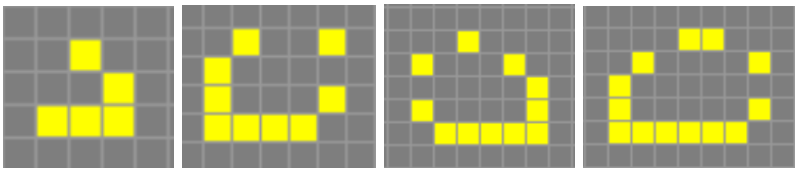
Pokud budeme zkoumat nejjednodušší případ automatů tohoto typ, pak okolí posuzované buňky budou tvořit právě dvě buňky (buňka vpravo a vlevo), se dvěma možnými stavy každé z nich tato trojice může nabýt celkem $2^3 = 8$ podob. Na tuto osmici může být aplikováno celkem $28 = 256$ pravidel.

Tato pravidla označuje Wolfram číslem podle dekadického zápisu pravidla. Například pravidlo 30 (dekadické označení) znamená v binárním zápisu 00011110.

Můžeme zapsat ve formě transformačního pravidla:

okolí	111	110	101	100	011	010	001	000
výsledek	0	0	0	1	1	1	1	0

Podívejme se jakým způsobem toto pravidlo vypadá vizuálně (viz. obr. 7.6).



(a) Kluzák (glider) (b) Lehká raketa (LWSS) (c) Střední raketa (MWSS) (d) Těžká raketa (HWSS)

Obrázek 7.5: Conwayova hra života - pohybující se struktury

Prozkoumejte předpřipravené modely NetLogo
Modely založené na podobných principech jsou schopny zachytit dobře některé dynamické děje jako může být třeba vztah predátor – kořist nebo simulace šíření lesního požáru apod. Podívejte se na další modely z knihovny modelů NetLogo. Objevily se některé další aplikace buněčných automatů jako je třeba *Coddův automat* (1968) [26] nebo *Langtonovy Q-smyčky* (1984) [48]. Novým impulzem pro výzkum v této oblasti však přinesl až *Wolframův 1D buněčný automat* [85].

K obr. 7.6 připomínám, že každý řádek odpovídá jedné iteraci. Každý žlutý bod pak představuje buňku ve stavu „1“ a každý černý body pak buňku ve stavu „0“. NetLogo poskytuje několik dalších předpřipravených pravidel.

Tím, že možných pravidel pro tento typ buněčných automatů je pouze 256, bylo možné provést analýzu výsledků všech těchto pravidel. Wolfram zjistil, že všechny výsledky lze zařadit do jedné ze čtyř kategorií (všimněte si souvislosti s pozorovaným chováním u Conwayovy hry života).

- *CA1* - všechny buňky nabudou stejné hodnoty (0 nebo 1) - např. pravidlo 40
- *CA2* - postupný přechod na opakující se tvary - např. pravidlo 228
- *CA3* - chaotický vývin (náhodný šum) - např. pravidlo 22
- *CA4* - složitá pravidelnost - např. pravidlo 110

Výsledky těchto čtyř příkladů máte možnost vidět na obr 7.7.

Bylo vysledováno, že podobné vzory se vyskytují také v přírodě. Klasická statistika přitom nebyla schopna vysledovat pravidelnosti (vzory) v těchto tvarech. Jak jsme si ukázali výše ve skutečnosti pravidlo může být velmi jednoduché. Příklad takového vzoru na ulitách mlžů naleznete na obr. 7.8.

Z praktického pohledu se nejedná o nic jiného nežli další příklad samoorganizace.

7.3 Boidi – Reynoldsův model shlukování ptáků

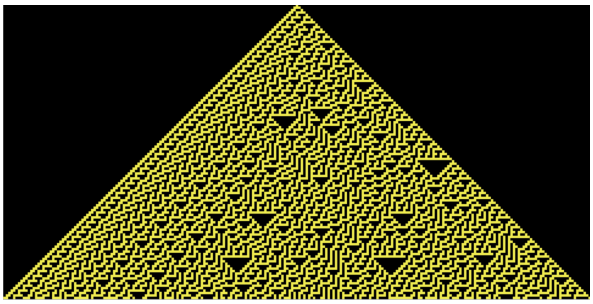
Reynolds [66] se při vytváření svého modelu nechal inspirovat hejny tažných ptáků. Tažní práci se pohybují v hejnech, která jako celek vykazují poměrně složité chování. Hejna ptáků svým elegantním pohybem fascinovala řadu lidí (viz obr. 7.9 pro inspiraci) a dlouhou dobu bylo záhadou, jaká je vlastně role jedince v takovém hejnu - např. otázka jak se má jedinec chovat tak aby neohrožoval sebe ani své okolí.

Modely Reynoldse ale ukázaly, že toto chování je ve skutečnosti výsledkem aplikace pouze velmi jednoduchých pravidel, a že s použitím těchto pravidel je možné vytvářet modely, které věrně simulují v počítači chování jedinců v těchto hejnech.

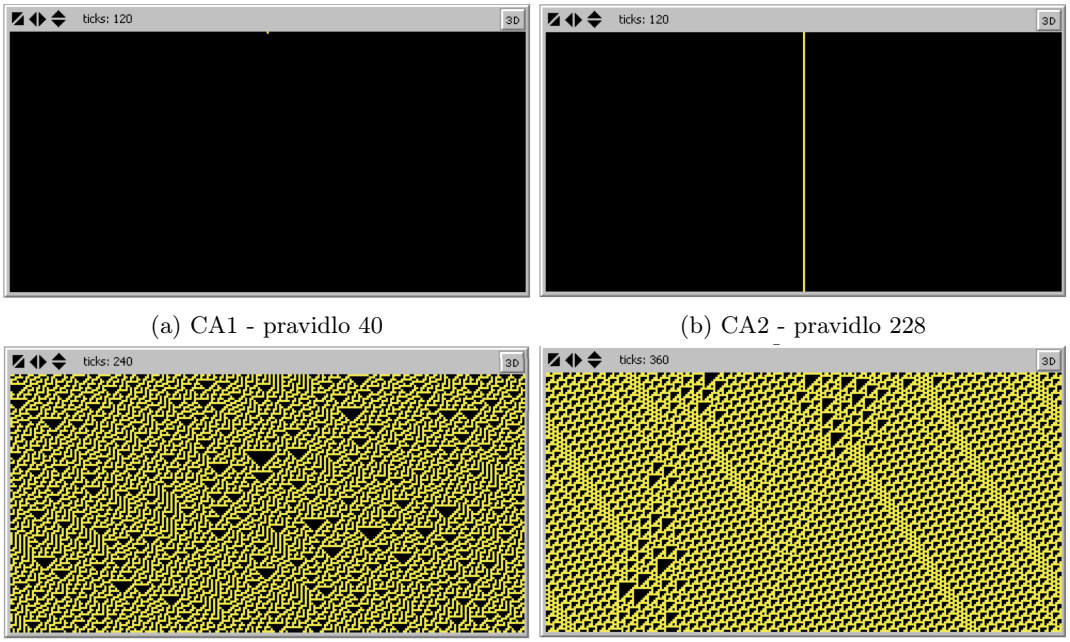
Tato pravidla jsou:

1. *separace* - jedinec se snaží přesunout k těžišti svých sousedů tak, aby se vyhnul potenciální kolize s nimi
2. *sladění rychlosti a směru* - jedinec se pohybuje přibližně stejnou rychlostí a stejným směrem jako jeho sousedé
3. *koheze* - jedinec přizpůsobí posici hejnu tak aby se postupně přibližoval průměrné poloze (těžišti) ostatních jedinců ve svém okolí.

Lepší představu o pravidlech si lze udělat z jejich grafického vyjádření viz obr. 7.10.



Obrázek 7.6: Wolframův 1D automat - vizualizace pravidla 30



(a) CA1 - pravidlo 40

(b) CA2 - pravidlo 228

(c) CA3 - pravidlo 22

(d) CA4 - pravidlo 110

Obrázek 7.7: Wolframovy 1D automaty - CA1 - CA4

Celé komplexní chování je tedy založeno na reakci jedince na chování jedinců v jeho nejbližším okolí. Výše uvedená pravidla tak tvoří přechodovou funkci s výchozími stavy - pozicí nejbližších jedinců v hejnu a rychlost a směru jejich letu. Funkce je aplikována lokálně na jednotlivé členy hejna.

Výše uvedený popis by Vám už měl něco říkat - porovnejte jej s postupy, které jsme aplikovali na Wolframovy 1D automaty a Conwayovu hru života.

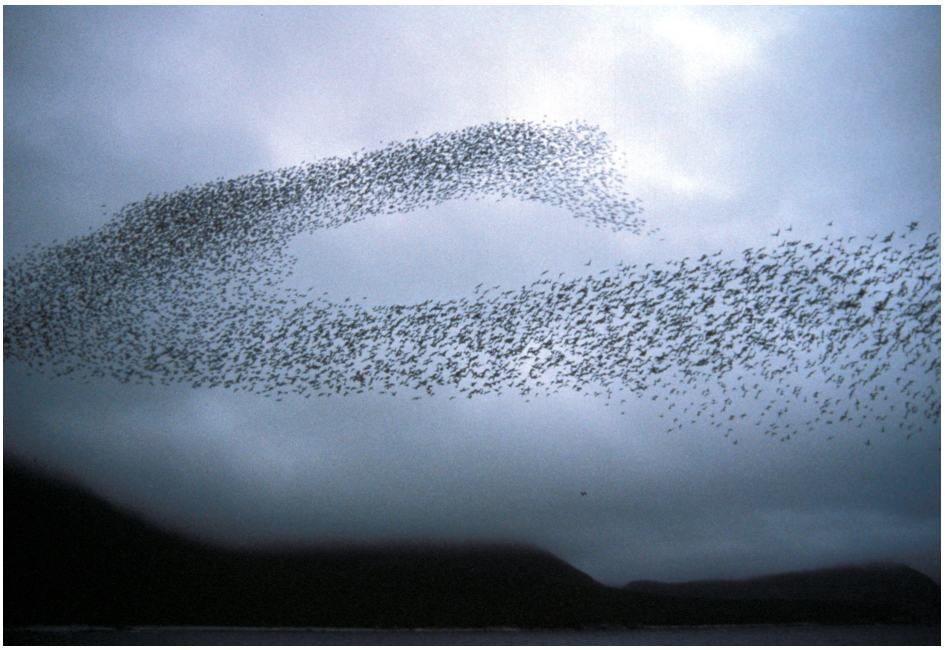
Podobná pravidla lze aplikovat i na jiné druhy zvířat žijících v hejnech jako jsou třeba některé druhy ryb.

Na obr. 7.11 je výsledek jednoduchého modelu boidů. Počáteční rozložení a směrování boidů v modelu je čistě náhodné, ale postupem času aplikací jednoduchých pravidel se boidi začínají shlukovat do malých hejn a taktéž se upravuje směr pohybu.

Tento model si můžete vyzkoušet sami v NetLogu z knihovny modelů → Perspective demos → Flocking (Perspective Demo).

Z bezpečnostně zajímavých oblastí se prakticky obdobné postupy používají např. pro zjednodušení pilotáže hejna dronů. V takovém případě nemusí být pilotovány všechny drony v hejnu ale pouze jeden nebo několik málo (podle účelu nasazení hejna). Ostatní nepilotované drony se přizpůsobí. V dopravě se o podobném modelu uvažuje v souvislosti s pozvolným nástupem samořiditelných aut. U takových vozidel se předpokládá, že do budoucna budou vybavena signalizací polohy dostupné pro ostatní podobně vybavená auta. Obdobný model pak může být využit např. v kolonách ve smyslu automatizované koordinace řízení vozidel v koloně.

Obrázek 7.8: Ulita mlže *Conus textile* (převzato z [52])



Obrázek 7.9: Hejno ptáků (převzato z [30])

7.4 Lindenmayerovy systémy

Lindenmayerovy systémy [51], někdy také nazývané zkráceně L-systémy, jsou pojmenovány po Arisitdu Lindenmayerovi, který modifikoval formální logiku pro účely modelování vývoje jednoduchých organismů. L-systémy spadají do oblasti počítačově generované grafiky - tzv. *fraktálů*.

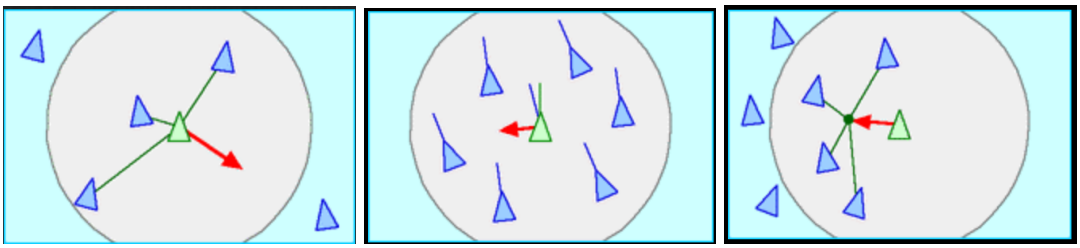
L-systém je potřeba chápat jako řetězec 7.1:

$$G = \{V, \omega, P\} \tag{7.1}$$

Kde V jsou sada symbolů (proměnných) používaných L-systémem, ω popisuje počáteční stav, někdy je tento prvek označován též jako *axiom* nebo *iniciátor* a konečně P je sada pravidel používaných v L-systému.

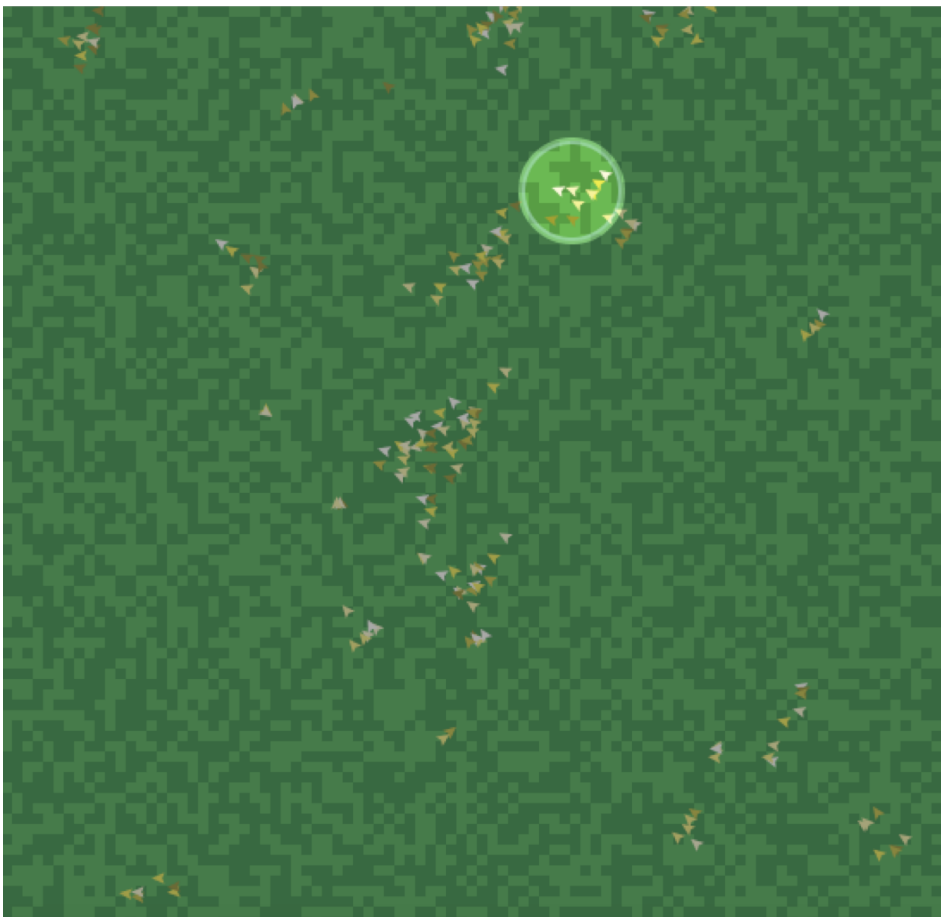
Můžeme demonstrovat na jednoduchém případě Sierpińského trojúhelníku.

- proměnné: A B
- konstanty: + -
- počáteční stav: A



(a) Separace (b) Sladění rychlosti a směru (c) Koheze

Obrázek 7.10: Pravidla chování ptáků v hejnu [67]



Obrázek 7.11: Simulace hejna - Flocking Perspective Demo (NetLogo)

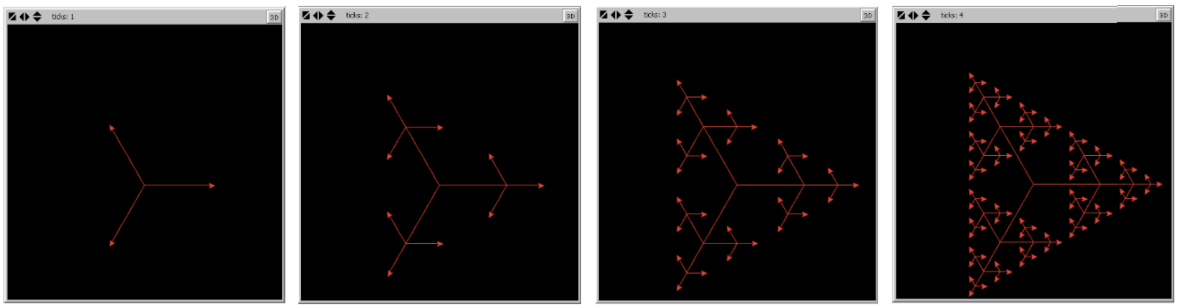
- pravidla: (A → B - A – B), (B → A + B + A)
- úhel: 60°

Pravidlo tvoří předpis provádějící transformaci proměnné. Proměnná A se tedy transformuje podle předpisu $B - A - B$. Tento předpis je možné interpretovat pomocí pravidel programovacího jazyka Logo, kde operátor + znamená otočení orientace v našem případě o 60° doleva, a operátor – otočení o 60° doprava.

Tedy $B - A - B$ provede vykreslení B, otočí pero o 60° doprava, vykreslí A, otočí o 60° doprava a vykreslí B.

L-systémy podobně jako běžně buněčné automaty pracují v iteracích. Podívejme se na obrázek 7.12, který lépe demonstruje způsob konstrukce tohoto fraktálního obrazce.

Na obr. 7.12 jsou znázorněny 4 iterace L-systému podle výše uvedených pravidel. Simulace však může probíhat neomezeně dlouho. V každé další iteraci se trojúhelník zahustí a výpočet bude trvat



Obrázek 7.12: Sierpińského trojúhelník v NetLogu

podstatně déle. Ostatně si to můžete vyzkoušet sami v NetLogu - knihovna modelů → Mathematics → Fractals → Sierpinski Simple.

Zajímavá je i konstrukce vločky Kochové, kterou tady ovšem uvádět nebudu, máte možnost si ji spustit z téhož umístnění (pod názvem Koch Curve).

Svůj věhlas získaly však L-systémy především svou schopností modelovat růst rostlin nebo jejich částí (např. listů). Výsledkem takového modelu může být třeba obr. 7.13.

Obrázek 7.13: Modelování travin pomocí fraktálů (převzato z [74])

Výhody využití tohoto postupu jsou jasné. Tvary popsatelné pomocí L-systémů není potřeba uchovávat ve formě 3D modelu, stačí pouze specifikovat předpis, podle kterého se vytvoří, jejich vytvoření může proběhnout za běhu daného programu. K těmto činnostem jsou dnešní výkonné grafické karty jako dělané.

7.5 Programovací jazyk Logo

Zatím jsme pracovali v prostředí NetLogo [3] uživatelsky, tedy používali jsme rozhraní, které pro nás vytvořil někdo jiný. Na tomto přístupu není samo o sobě nic špatného, pouze jsme odkázáni na někoho jiného, který s největší pravděpodobností nevytvoří takový model jaký bychom potřebovali.

Takže několik málo slov o programovacím jazyku Logo. Jedná se o programovací jazyk (počátek 1967) určený pro výuku programátorských konceptů pro malé děti. Funguje to tak, že děti dostanou k dispozici kurzor (ve tvaru želvy), kterou může dítě ovládat a pomocí ní kreslit.

V ČR se používal jiný koncept pro výuku programování a to programovací jazyk Karel, kde uživatelé ovládali robotka Karla [63], který mohl různě přenášet kostičky apod.

Podívejme se na program pravidla 22 Wolframova 1D automatu, který jsme popisovaly dříve.

Listing 7.1: Pravidlo 22 Wolframova 1D automatu - implementace v jazyku Logo

```
globals [row]
patchess-own [left-pcolor center-pcolor right-pcolor]

to setup
        ca
        set row max-pycor
        ask patch 0 max-pycor [ set pcolor yellow ]
end

to go
        if (row = min-pycor)
                [ stop ]
        ask patches with [pycor = row]
                [ do-rule ]
        set row (row - 1)
        tick
end

to do-rule
        set left-pcolor [pcolor] of patch-at -1 0
        set center-pcolor pcolor
        set right-pcolor [pcolor] of patch-ar 1 0
        if((left-pcolor = yellow and center-pcolor = black and right-color = black)
            or
        (left-pcolor = black and center-pcolor = yellow and right-pcolor = black) or
        (left-pcolor = black and center-pcolor = black and right-pcolor = yellow) or
        (left-pcolor = black and center-pcolor = yellow and right-pcolor = yellow))
                [ set [pcolor] of patch-at 0 -1 yellow ]
                [ set [pcolor] of patch-at 0 -1 black ]
end

to setup-continue
        ask patches with [pycor = max-pycor]
                [ set pcolor ([pcolor] of patch pxcor min-pycor) ]
        ask patches with [pycor != max-pycor]
                [ set pcolor black ]
        set row max-pycor
end
```

Jak vidíte samotný program není dlouhý (pouze jedna stránka) a vyskytuje se v něm pouze minimum příkazů. První dva řádky mají deklarační charakter. První z nich definuje *row* jako globální proměnnou dostupnou ve všech procedurách programu.

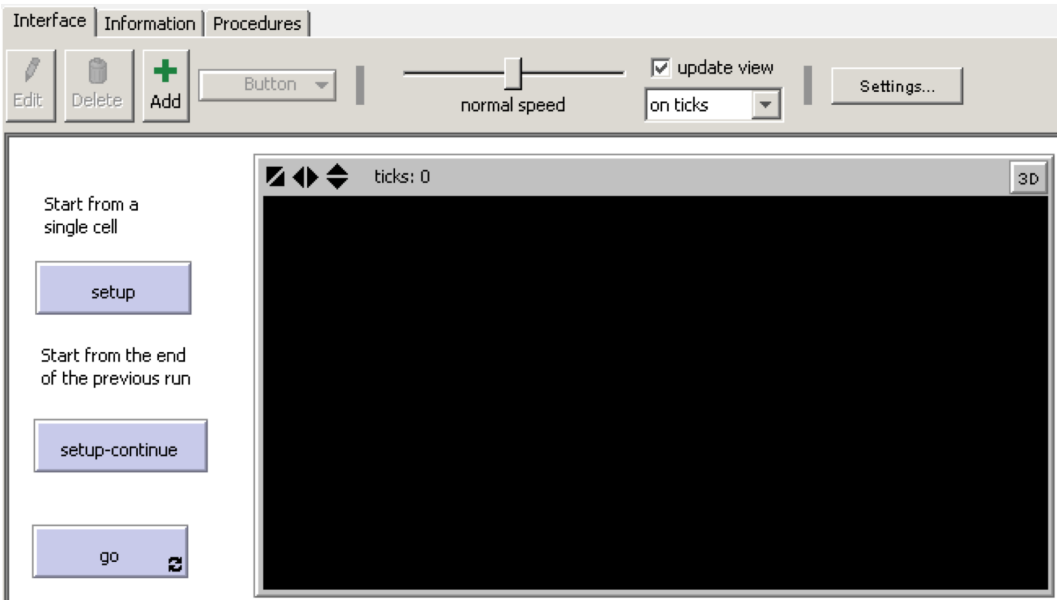
Proměnné *left-pcolor, center-pcolor a right-pcolor* jsou dostupné pouze buňkám (*patches-own*). Při zkoumání okolí buňky v 1D hodnotíme buňku samotnou a její okolí, tedy buňky vpravo a vlevo. Stav buňky můžeme vyjádřit graficky změnou barvy. V tomto programu budeme využívat barvy 2 a to černou pro „mrtvou“ buňku a žlutou pro „živou“ buňku.

Předtím něž se podíváme na zbytek programu, podívejme se na uživatelské rozhraní na obr. 7.14.

Jednotlivá tlačítka, nebo i jiné prvky GUI se přidávají velmi jednoduše kliknutím na tlačítko velkého plus s popiskou add (tedy přidat) a výběrem typu prvku GUI, který je potřeba přidat. Na pracovní ploše potom pak tento prvek vytvoříme kliknutím a tažením.

Po vytvoření prvku je nutno nastavit základní vlastnost, především jméno tohoto prvku, ale i některé další vlastnosti.

V našem případě jsou již prvky GUI předpřipraveny. Všimněte si nápisu na tlačítcích *setup, setup-continue* a *go*. Pokud se podíváte do zdrojového kódu zjistíte, že jsou v něm obsaženy procedury to



Obrázek 7.14: GUI NetLogo

setup, *setup-continue* a *go*. Logo totiž obsahuje velmi jednoduchý model událostí, zejména pro tlačítka. Po kliknutí na tlačítko se provede procedura nazvaná stejně.

Kromě těchto tří procedur napojených na tlačítka je v programu ještě procedura *do-rule*, která se stará o aplikaci pravidla. Podívejme se na jednotlivé procedury.

Procedura *setup* se stará o úvodní nastavení modelu. Provede se resetování univerza modelů do prázdné (černé) podoby a do prvního řádku se přidá jediná aktivní (žlutá) buňka. Proberme proceduru příkaz za příkazem

ca je synonymum pro příkaz *clear-all*, který provede smazání celého obsahu univerza.

Příkaz *set* nastaví proměnnou *row* na hodnotu *max-pycor*, tedy maximální hodnotu Y-nové koordináty buňky dostupné v našem univerzu.

Konečně poslední řádek procedury *setup* můžeme do běžné řeči přeložit, že požadujeme (*ask*) po buňce (*patch*) uprostřed (0) v prvním řádku (*max-pycor*), aby nastavil (*set*) svou barvu (*pcolor*) na žlutou (yellow). Protože bychom po buňkách mohli chtít více operací oddělujeme to, co mají buňky vykonat, od všeho ostatního pomocí hranatých závorek.

Jako alternativou k proceduře *setup* je procedura *setup-continue*. Tato procedura opět provede reset univerza, jako počáteční stav se ale použije poslední iterace předchozího běhu modelu. Nezačínáme tak s jednou buňkou ale větším množství buněk.

Procedura *go* se stará procházení univerza v rámci jednotlivých iterací modelu. Činnost probíhá tak, že se zkontroluje zda jsme náhodou nedošli na poslední řádek univerza (*if (row = min-pycor)* a v takovém případě se přeruší (*stop*) běh programu, v opačném případě se ale požádají buňky na řádku (*ask patches with [pycor = row]*) aby na nich bylo aplikování pravidlo obsažené v proceduře do-rule. Nakonec dojde k přesunu na další řádek (*set row (row – 1)*) a provedení další iterace (*tick*).

Pravidlo samotné je aplikováno procedurou *do-rule*. Tato procedura nejprve načte do proměnných stav posuzované buňky a jejích okolí a následně rozhodne, zda výsledkem bude „živá“ nebo „mrtvá“ buňka.

Pravidlo 22 vypadá následovně:

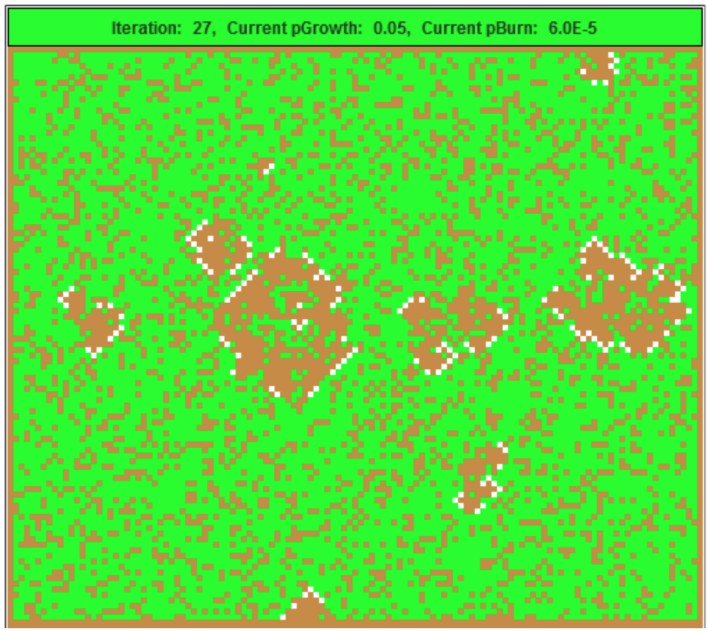
```
okolí      111 110 101 100 011 010 001 000
výsledek     0   0   0   1   0   1   1   1
```

Pro aplikaci pravidla musíme otestovat zda konfigurace buněk odpovídá některé ze čtyř konfigurací vedoucích k výsledku 1 (život). Všechny ostatní logicky musí vést k výsledku 0. Tuto funkčnost zajistíme použitím podmínky, kde jako predikát použijeme deklarativní vyjádření výše uvedeného pravidla, např. 100 zapíšeme left-pcolor = yellow and center-pcolor = black and right-pcolor = black.

7.6 Aplikace celulárních automatů v bezpečnosti

V předchozích podkapitolách jsme popisovali některé menší bezpečnostní aplikace buďto přímo jednotlivých typů automatů nebo principů, které používají. V této podkapitole se ale podíváme na využití vlastností automatů pro účely simulace zájmového děje.

Zkusme vytvořit jednoduchý simulátor rozvoje požáru v uzavřeném prostoru. Myšlenka modelovat požáry pomocí celulárních automatů není nová. První experimenty byly s modelováním lesních požárů, viz obr. 7.15.



Obrázek 7.15: Model Fire po 27 iteracích (obrázek vygenerován pomocí appletu [73])

Model používá tři stavy buněk - zelená barva reprezentuje lesní porost, hnědá barva pak zem. Konečně bílá barva reprezentuje čelo požáru. Model se iniciuje jedním nebo více zdroji požáru a následně v každé iteraci se propočítává jeho šíření. Požár se může šířit pouze do sousedních polí obsahujících porost. To, zda požár do sousedního pole přeskočí rozhodne nastavení pravděpodobnosti šíření.

Tento model předpokládá, že porost během jedné iterace shoří. Tento model je jednoduchý - ve skutečnosti je natolik jednoduchý, že by neměl být pro Vás neřešitelný problém upravit příklad z předchozí podkapitoly (s pomocí manuálu NetLogo a trochou kreativního Googlování) do funkční podoby odpovídající obr. 7.15.

Takový model je ale samozřejmě silně zjednodušující. Existují ale pokročilejší modely, které jsou schopny brát v úvahu např. hustotu zalesnění, vliv topologie terénu na šíření požáru a především pak vliv působení větru. Takový pokročilejší model je demonstrován např. na videu [44], model samotný vznikl v rámci výzkumného projektu P. Jakubase [45].

Model požáru v uzavřených prostorách popisuje Černý a Šenovský [88] odlišně. Univerzum simulace zachycuje uzavřený prostor v půdoryse. V modelu se vyskytují výkonní agenti „želvy“[1] a dlaždice (buňky), jejichž implementace je podobná jako v modelu *Fire*, ovšem je k nim přiřazeno více vlastností, v souladu s větší složitostí modelu.

Dlaždice představují hořlavý prostor, dále zdi a jiné požárně-dělicí konstrukce, zároveň jsou nositeli plamenů. Vlastnosti dlaždic reprezentují následující vlastnosti:

- *Ignition-temperature* [°C]: představuje parametr hořlavosti – jak snadno je hořlavina zapalitelná
- *Fire-load* [kg na dlaždici]: představuje množství hořlaviny na dlaždici
- *Temperature* [°C]: udává teplotu okolí dlaždice
- *Fire-resistance* [min]: reprezentuje požární odolnost dlaždice. Její hodnota představuje čas v minutách, po který je konstrukce (reprezentovaná dlaždicí) odolná vůči účinkům požáru

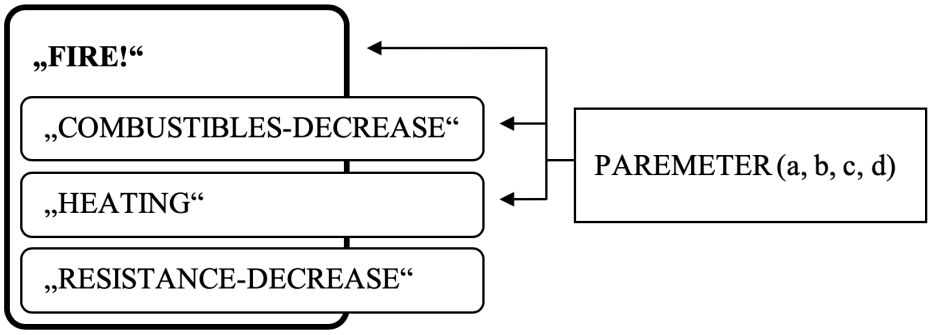
[1] V angličtině se grafika Logo prostředí často označuje jako tzv. *turtle grafic*, tedy želví grafika podle tvaru symbolu který označoval polohu „kreslícího pera“ na obrazovce

- *Patch-time* [min]: časový úhrn v minutách, po který byl na dlaždici požár

Výše uvedené vlastnosti zároveň vystupují jako parametry výpočtu.

Želvy: zastupují přítomnost plamenů. Nepohybují se, vznikají na dlaždicích, na kterých jsou podmínky pro požár, a ve chvíli, kdy podmínky přestanou být pro požár příznivé, zanikají. Na jedné dlaždici může být maximálně jediná „želva“.

Struktura modelu je znázorněna na obr. 7.16.



Obrázek 7.16: Struktura experimentálního modelu požáru v uzavřených prostorách (převzato z [88])

Model je tvořen hlavní procedurou FIRE, která obsahuje další při procedury ovládající vlastnosti dlaždic:

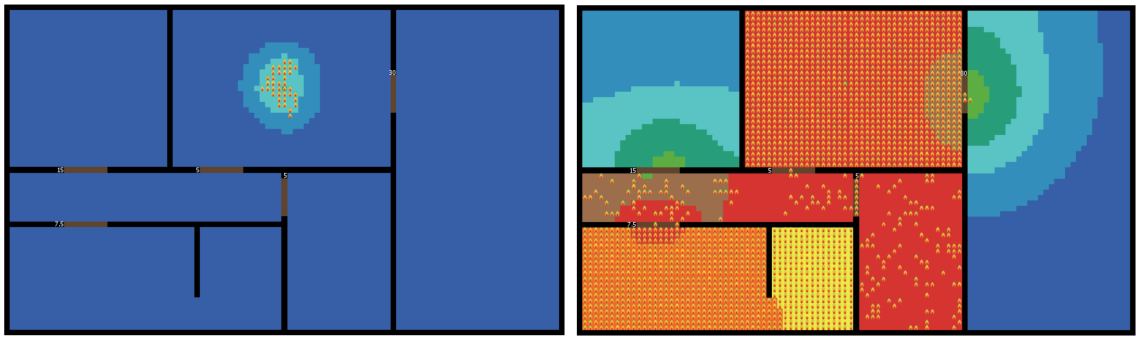
- COMBUSTIBLES-DECREASE - realizuje úbytek hořlaviny,
- HEATING - realizuje sdílení tepla,
- RESISTANCE-DECREASE - realizuje pokles požární odolnosti konstrukce v důsledku působení požáru.

Funkce PARAMETER představuje v procedurách míru, jakou se budou provádět změny vlastností jednotlivých dlaždic.

Pro samotné odladění modelu jsme zvolili simulaci rozvoje požáru v prostředí s půdorysem jako na obr. 7.17a. Zvolená konfigurace modelu představuje půdorys bytového prostoru. Stěny jsou černě, volný prostor je modře. Prostupy jsou řešeny hořlavými dlaždicemi s přidělenou požární odolností (hnědě). Každý prostor má přidělenou různou hodnotu množství hořlaviny a její hořlavosti.

Na obr. 7.17a je zároveň dobře patrný zdroj požáru, jehož rozvoj se snažíme simulovat.

Oblasti zasažené plamenem jsou symbolizovány ikonou plamene. V prostoru s původně konstantní teplotou se v důsledku požáru postupně zvyšuje teplota. Stav simulace v čase 34:03 je znázorněn na obr. 7.17b.



(a) čas 0:46 (b) čas 34:03

Obrázek 7.17: Simulace rozvoje požáru v uzavřeném prostoru (převzato z [88])

Zdrojový kód simulace by mohl vypadat následovně (převzato [87]):

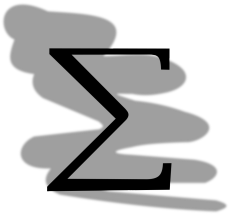
Listing 7.2: Model rozvoje požáru v uzavřeném prostoru realizovaný v NetLogo

```
breed [cursors cursor]
breed [flames flame]
globals [blue? start-x start-y end-xend-yG-X G-Y B-X B-Y]
patches-own [ignition-temperature fire-load temperature
       type-of fire-resistance fire-here? patch-time]

; HLAVNÍ, ŘIDÍCÍ PROCEDURY
to-report PARAMETER [a b c d]
       report((a * count neighbors with [any? flames-here]
              / count neighbors with [type-of ="combustibles"] * 2)
              + (b * sum([temperature] of neighbors with [type-of ="combustibles"])
              / (count neighbors with [type-of ="combustibles"]) /500)
              + (c * fire-load /120) + (d * 300/ignition-temperature))
              / (a + b + c + d)
end

to FIRE!
       tick
       set minutes int (ticks * sec-per-tick / 60)
       set seconds int (ticks * sec-per-tick) mod 60 ;časomíra
       ask patches [setfire-here? false]
       ask patches with [type-of = "combustibles" and not any? flames-here
                     and fire-load >1 and any? neighbors with [any? flames-here]]
              [set fire-here? true]
       ask patches with [fire-here? = true] ;zapálení dlaždice
              ;fire-resistance: požární odolnost
              [if(PARAMETER 3 9 3 3) * 0.75 + 0.25 > (random-float 20 +
                  fire-resistance)
                     [sprout-flames 1]
              ]
       COMBUSTIBLES-DECREASE ; úbytek hořlaviny
       HEATING ; změny teploty
       RESISTANCE-DECREASE ; snížení požární odolnosti
end

to HEATING
       ask patches with [any? flames-here] ;ohřev dlaždice
              [set temperature temperature + (345
                     * log((8 * (patch-time + sec-per-tick / 60) + 1)
                     / (8 * patch-time +1)) 10) * (PARAMETER 0 0 3 3 * 0.75 + 0.5)
              set patch-time patch-time + (sec-per-tick / 60)]
        ;rozptyl tepla
        askpatches with [type-of = "combustibles"
              and fire-resistance < random-float 5]
                     [settemperature ((temperature +sum [temperature] of neighbors
                            with [type-of = "combustibles"
                            and fire-resistance < 5])
                            / (1 + count neighbors with [type-of = "combustibles"
                            and fire-resistance < 5]))]
        ask patches with [type-of = "combustibles"] ;ochlazení dlaždic
              [if temperature > 20 + random-float 50
                     [set temperature temperature - PARAMETER 0 1 1 0 * 0.1
                            * sec-per-tick]]
end

to COMBUSTIBLES-DECREASE
       ask flames
```

```
57              [if fire-load < random-float 5[die] ;podmínka uhašení požáru
58              set fire-load fire-load - ((random-float 0.001) * fire-load
59                      * PARAMETER 1 5 1 5)] ;úbytek hořlaviny
60  end
61
62  to RESISTANCE-DECREASE ;snížení požární odolnosti
63          ask patches with [fire-resistance > sec-per-tick / 60
64                  and any? neighbors with [any? flames-here]]
65                          [setfire-resistance fire-resistance - sec-per-tick / 60]
66  end
```

Jak je vidno z výpisu výše, není kód pro simulátor příliš obsáhlý, přesto je pomocí něj dosahováno poměrně komplexní funkčnosti. Model samotný je ale potřeba chápat spíše jako experimentální, byť prošel určitou validací, viz [87].

Plný text práce Černého [87] včetně zdrojových kódů simulátoru i validačních modelů lze dohledat na https://dspace.vsb.cz.



Shrnutí kapitoly

V této kapitole jsme se seznámili s řadou modelů a postupů které společně označujeme jako celulární (buněčné) automaty. Jedná se o skupinu nástrojů umožňujících dynamické modelování formálně specifikovaného univerza - nejčastěji představitelného jako mřížka, jejíž jednotlivá políčka (buňky) nastavujeme pomocí pravidel.

Pravidla jsou aplikována najednou (na všechny buňky zároveň) v rámci iterace pomocí tzv. *lokální přechodové funkce*.

Celulární automaty mají poměrně široké spektrum aplikací od rozšiřování lidského poznání, např. o chování některých komplexních systémů (viz např. *Conwayova hra života*), po poznání tvarů, které se objevují v přírodě (*Wolframův 1D automat*, *L-systémy*) až po model chování jedinců v hejnu (*boidi*).

V oblasti bezpečnosti je tento typ modelů poměrně extenzivně používán pro simulace průběhu lesních požárů. Z pohledu možných nasazení se ale jedná o poměrně univerzální nástroj aplikovatelných pro modelování různých vlastností řady dynamických systémů.

Kontrolní otázky

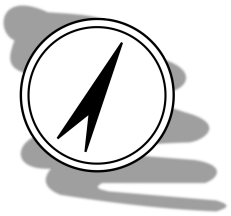
1. Jaké je použití L-systémů?
2. Řadíme buněčné automaty mezi statické nebo dynamické modely?
3. Vysvětlete způsob chování ryb v hejnu.
4. Jaký je rozdíl mezi von Neumanovým a tzv. úplným okolím?

Úkoly k zamyšlení

1. Prozkoumejte další předpřipravené modely v NetLogu.
2. Zkuste modifikovat existující pravidlo 22 1D Wolframova buněčného automatu na pravidlo 222.

Kapitola 8

Multiagentní systémy



Náhled kapitoly
V této kapitole se seznámíme s možnostmi využití jednodušších samostatně fungujících komponent na bázi umělé inteligence (agentů), pro získání komplexního - inteligentního chování.

Po prostudování této kapitoly budete
vědět

- co je to agent
- jak takový agent funguje
- jakým způsobem dochází k výměně informací mezi agenty

Čas pro studium
Na prostudování této kapitoly budete potřebovat dvě hodiny.

Multiagentní systémy pro nás ve skutečnosti nejsou nové - již jsme se s nimi setkali i v tomto předmětu. V předchozí kapitole jsme pracovali s *celulárními automaty*, které ve skutečnosti spadají pod širší definici multiagentních systémů. Oproti systémům, na které se zaměříme v této kapitole ale mají pevně stanovenou strukturu univerza (buňky) a omezený stavový prostor.

Multiagentní systémy popisované v této kapitole však taková omezení nemají.

Zkusme definovat probrat problematiku multiagentních systémů postupně. Začněme samostatnými agenty.

8.1 Inteligentní agent

Multiagentní systémy jsou oblastí umělé inteligence, která zejména v poslední době nabývá na významu. Multiagentní systémy se zásadně liší od aplikací umělé inteligence, které jsme probírali dosud. Všechny ostatní přístupy se totiž vesměs (s různou mírou úspěšnosti) snažily napodobit co možná nejlépe systém uvažování myslící bytosti. Expertní systémy svou pokročilou funkčností v oblasti odvozování poznatků, neuronové sítě se snažily podchytit samotnou fyzickou strukturu mozku.

Agentní systémy na takové komplexní chování úmyslně rezignují.

Úkolem agenta tedy není přiblížit se co možná nejvíce myslící bytosti, ale vykonávat co nejlépe úkol, pro který byl navržen. Předpokládá se přitom, že agent nebude ve finále pracovat sám, ale v kooperaci s dalšími agenty s odlišnými schopnostmi. Teprve vzájemnou interakci agentů vzniká komplexita v chování celého systému a do jisté míry se v něm objevují[1] prvky inteligentního chování.

[1] objevují ve smyslu emerge, tedy bavíme se zde o emergentním chování kdy systém získává pokročilejší/nečekané vlastnosti na základě interakci a fungování jeho jednoduchých komponent.

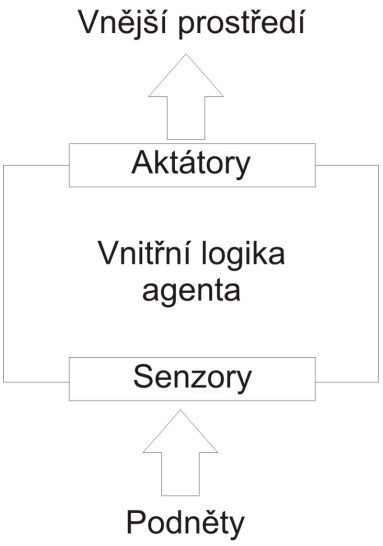
Z výše uvedeného vyplývá, že agenti jsou:
- specializovaní na řešení určeného (konkrétního) úkolu
- jednotliví agenti jsou strukturálně jednoduší
- pro řešení úkolu je obvykle nutno nasadit řadu agentů s různými specializacemi
- mezi agenty musí existovat čilá komunikace pomocí předem definovaného komunikačního protokolu, aby mohli sdílet výsledky své práce

Vnitřní implementace jednotlivých agentů může být různá. Agenti mohou být naprogramováni klasicky - algoritmizací činnosti, nebo mohou implementovat různé nástroje strojového učení, jako třeba neuronové sítě. Z pohledu multiagentního systému je vnitřní organizace jednotlivých agentů irelevantní jelikož z pohledu ostatních agentů a okolního prostředí fungují jako černé skříňky a vůči svému okolí vystavují pouze části svého chování realizovanými do vnějšího prostředí a to buďto aktátory nebo vysíláním, popř. vyžádáním nebo předáním dalších informací agentům v systému (komunikace agentů).

Komplexnosti chování multiagentního systému jako celku se dosáhne pomocí výměny informací a interakcí mezi nimi. Cílem takové informační výměny je koordinovat činnost agentů s různými schopnostmi tak, aby vzájemně plnili své potřeby a získali tak schopnost dosáhnout zadaných cílů, které by bez této spolupráce nebyly dosažitelné.

Na základě výše uvedeného můžeme již přistoupit k formální definici agenta v takovém systému. Kubík [46] definuje *inteligentního agenta* následovně: *Agent je entita zkonstruovaná za účelem kontinuálně a do jisté míry autonomně plnit své cíle v adekvátním prostředí na základě vnímání prostřednictvím senzorů a prováděním akcí prostřednictvím aktátorů.* Agent přímo ovlivňuje podmínky prostředí tak, aby se přibližoval k plnění cílů.

Podívejme se na vnitřní konstrukci agenta (viz obr. 8.1).



Obrázek 8.1: Vnitřní konstrukce inteligentního agenta

Agent získává podněty z okolí prostřednictvím *senzorů*, kterými je vybaven. Za senzory přitom považujeme vše, co je schopno přijímat informace z vnějšku, v případě softwarových agentů se tak může jednat o nějakou službu nebo interface, který zachytává určité údaje.

Údaje zachycené senzory jsou zpracovávány vnitřní logikou agenta. Jak jsme již zmínili výše, může být tato vnitřní logika rozdílná a proto se k ní ještě vrátíme podrobněji později. Každopádně na základě těchto podnětů začne agent konat prostřednictvím mechanismů k tomu určeným tzv. *aktátorů*.

Aktátory rozumíme všechny mechanismy, kterými agent působí na své okolí. Může se jednat o kola pro přesun agenta, může se jednat o manipulační „ruku“, nebo o nějaké jiné nástroje.

Z hlediska vnitřní logiky agenta je možné rozdělit agenty do dvou skupin:
- reaktivní agenty
- deliberativní agenty

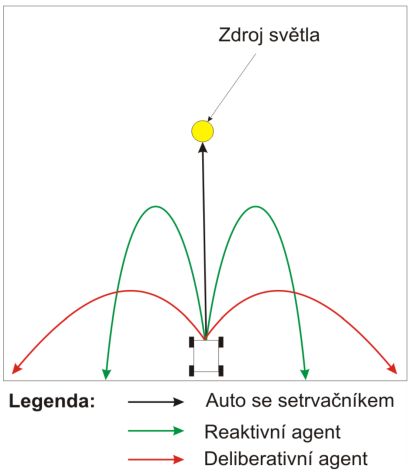
Reaktivní agenti svou činností pouze reagují na změny v okolním prostředí. V této koncepci se vychází z toho, že živé organismy vykonávají řadu funkcí automaticky, aniž by musely nad těmito úkoly přemýšlet, prostě jako reakce na různé události, se kterými se setká. Reaktivní agenti jsou založení právě na tomto principu.

Deliberativní agenti oproti agentům reaktivním vytvářejí vnitřní reprezentaci (model) problémové situace, mají určité cíle a těchto cílů se snaží optimálně dosáhnout. Agent v tomto případě má k dispozici konečnou sadu akcí, které může použít pro řešení situace. Agent nejprve zkusí jaký následek bude mít použití těchto akcí ve vnitřním modelu a určí optimální akci nebo posloupnost akcí které povedou k jeho cíli.

Teprve v tomto okamžiku se akce považované za optimální začnou provádět, agent zároveň kontroluje prostřednictvím senzorů, zda prováděné akce skutečně vedou k očekávaným výsledkům a pokud ne provádí opětovné přehodnocení.

Demonstrujme si jednoduchý příklad reakce agenta. Náš agent bude mít podobu autíčka, jehož úkolem bude dostat se do místa co nejdále od zdroje světla.

Reakce různých typů agentů jsou zobrazeny na obr. 8.2.



Obrázek 8.2: Reakce různých typů agentů na vnější podnět (zdroj světla v místnosti)

Auto se setrvačníkem samozřejmě není agentem do obrázku bylo přidáno jednom z důvodu demonstrace směru, kterým se agent v počátku simulace ubírá. Autíčko se setrvačníkem prostě pokračuje kupředu bez ohledu na své okolí, protože není ani trošku inteligentní a jede tak dopředu ve směru, ve kterém bylo vypuštěno.

Reaktivní agent již vykazuje inteligentnější chování. Jeho pouť začíná stejným způsobem jako pouť autíčka setrvačníkem, tedy kupředu. Reaktivní agent ale záhy zaregistruje prostřednictvím svých senzorů, že intenzita světla, kterou má minimalizovat, roste a je nucen na to reagovat. Reakce spočívá v tom, že může obrátit směr a to doleva nebo doprava (rozhodnutí může být náhodné) tak dlouho, než nabere směr od zdroje světla.

Zakřivení křivky pohybu (prudkost reakce) agenta je dána nastavením parametrů agenta. Reaktivní agent tak úplně klidně může skončit ve stejném umístnění jako deliberativní agent, dá se ale očekávat, že mu to bude trvat déle, protože jeho křivka pohybu nebude optimální.

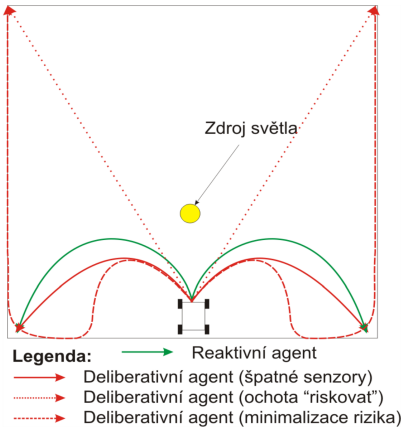
Konečně deliberativní agent pomocí senzorů „pozná“ prostředí, ve kterém se pohybuje, na jeho základě si vytvoří model (vnitřní reprezentaci) prostředí a naplánuje optimální trasu pohybu tak, aby se dostal do nejvzdálenějších rohů, co možná nejdále od zdroje světla, kde je intenzita světla nejmenší.

Zakřivení křivky pohybu bude jen tak velké, aby se agent otočil do požadovaného směru.

Rozdíl mezi oběma druhy agentů bude ještě markantnější pokud předchozí modifikujeme tak, aby zdroj světla byl na stejné půlce jako hrací plochy (univerza agentů), viz obr. 8.3.

Reaktivní agent se bude chovat přibližně stejně. Zásadní rozdíl v chování ale může nastat u deliberativního agenta, protože v závislosti na kvalitě senzorů a vnitřního modelu může odhalit, že ve vzdálenější části univerza se „*za světlem*“ nacházejí oblasti s nižší intenzitou světla než se nachází v jeho bezprostředním okolí.

Pokud agent disponuje dostatečnými senzory a vnitřním modelem o cestě do vytouženého místa rozhodne jeho tolerance k dočasnému „nepohodlí“, tedy dočasně se zvyšující intenzitě světla. Agent si



Obrázek 8.3: Rozdíl v pohybu agentů

tak může optimalizovat své nepohodlí plánováním trasy na základě pravidel daných tvůrcem agenta.

8.2 Multiagentní systémy

Multiagentní systémy jsou systémy složené z inteligentních agentů, kteří vzájemně spolupracují nebo na sebe jinak působí.

Z tohoto pohledu za nejjednodušší multiagentní systém může být považován jakýkoliv celulární automat. Jednotlivé buňky by pak byly agenty a působily by na sebe svým stavem. Takovýto multiagentní systém by nám však příliš užitku (ve srovnání s teorií celulárních automatů) nepřinesl.

Podívejme se na složitější příklad. Pro demonstraci možností multiagentního simulačního prostředí se obvykle používá simulace *heatbugs*, která zachycuje život brouků, kteří žijí ve světě s místy o různé teplotě. Brouk, pokud má správnou teplotu, je šťastný a nepohybuje se, pokud je mu ale zima nebo horko, snaží se dostat na teplejší nebo chladnější místo.

Tuto simulaci si můžete prohlédnout v NetLogu, ale také v jiných simulačních balících jako je Repast [15], Swarm [6] a řada dalších. Takové simulace jsou však obvykle založeny na zkoumání emergentního chování agentů. Tedy jako takové spadají někde mezi umělý život a agenty. Tyto nástroje je možné použít i pro simulace skutečných agentů, je to pouze o hodně složitější.

Heatbugs na sebe opět působili svou přítomností, co když potřebujeme zajistit, aby se agenti přímo domluvili. V takovém případě je nutné implementovat společný komunikační protokol, který všichni agenti dodrží. Přitom je potřeba si uvědomit, že vnitřní logika agentů může být velmi odlišná od operačního systému, ve kterém jsou implementováni až po různé programovací jazyky.

Samozřejmě je možné vyvinout komunikační protokol přímo pro naši aplikaci multiagentní technologie, nicméně by se jednalo o proces svým významem odpovídajícímu opětovnému vynálezu kola. Existuje řada protokolů, které pro tento účel lze použít. Z této řady jsou nejrozšířenější dva a to konkrétně Knowledge Query and Manipulation Language (KQML) [31] a FIPA-ACL [32].

Význam těchto protokolů spočívá především v tom, že standardizují tzv. *komunikační akty* mezi agenty. Při návrhu komunikačního protokolu se vycházelo z lingvistické analýzy běžného jazyka. Teorie komunikačního aktu vychází z hypotézy, že řečový akt (to co říkáme) souvisí s tím co je nebo bude provedeno.

Příklad vypíši zkušební termín na středu druhý týden v únoru. Tento akt deklaruje co bude provedeno v budoucnu.

Komunikační akty podle obsahu lze rozdělit [46] do několika kategorií:

- *oznamovací* - tento akt sděluje dosažení nebo nedosažení nějakého cíle nebo třeba zjištění důležité okolnosti. Př.: Dorazil jsem na místo. Baterie je plně nabitá. Diagnostika systému proběhla v pořádku.
- *zavazující akt* - zavazuje účastníka komunikace provést nějakou činnost. Př.: za 5 minut provedu odečet hodnoty.
- *přikazující akt* - přikazuje příjemci vykonat nějakou činnost. Př. Proveď odečet hodnoty. Zavři ventil apod.

Komunikace mezi agenty má formu obvykle odpovídající výše uvedeným kategoriím.

V KQML by samotný komunikační akt mohl vypadat následovně:

Listing 8.1: Komunikační akt v jazyku KQML

```
(ask-one
        :sender Agent002
        :reciever Agent007
        :language prolog
        :ontology NL
        :content "identifyNL(0055)"
)
```

Tento komunikační akt vyžaduje odpověď agenta 007 agentovi 002 na identifikaci nebezpečné látky z ontologie NL (nebezpečné látky).

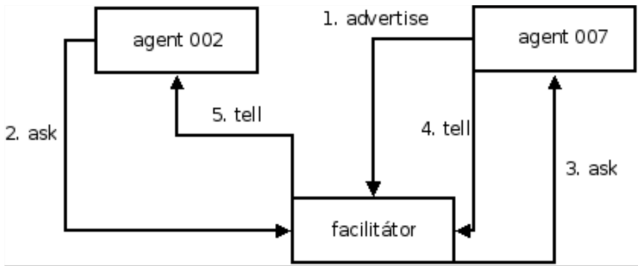
Každý protokol obsahuje odlišnou sadou dostupných komunikačních aktů. Tyto akty, jelikož vyžadují na druhé straně nějakou činnost nazýváme *performativy*.

Agenti mohou komunikovat přímo (viz. obr. 8.4). Například na výše uvedenou otázku by mohl agent 007 přímo odpovědět jménem identifikované nebezpečné látky (otázka - odpověď).

Obrázek 8.4: Přímá komunikace mezi agenty

Taková komunikace však vyžaduje splnění několika podmínek. Agenti o sobě musí vědět nejenom kteří agenti jsou v prostoru přítomni, ale také jaké služby jsou schopni poskytovat. V případě, že toto nejsme schopni nebo ochotni zajistit, je potřeba použít jiný typ scénáře a to s použitím prostředníka – *facilitátora*.

Úkolem facilitátora je „seznámit“ agenty v případě potřeby a nebo zprostředkovat poskytnutí služby (viz. obr. 8.5).



Obrázek 8.5: Komunikace agentů prostřednictvím facilitátora

V prvním kroku zaregistruje své schopnosti agent 007 u facilitátora. Podobným způsobem by měly postupovat všichni agenti, aby pak v okamžiku, kdy vznikne nějaký požadavek (2) byl facilitátor schopen najít a zkontaktovat (3) vhodného agenta, získat od něj odpověď (4) a tuto odpověď sdělit původnímu agentovi.

V takovém komunikačním scénáři agenti nemusí mít povědomí o počtu a schopnostech dalších agentů. Stačí jim znát facilitátora, který jim zprostředkuje veškeré odpovědi.

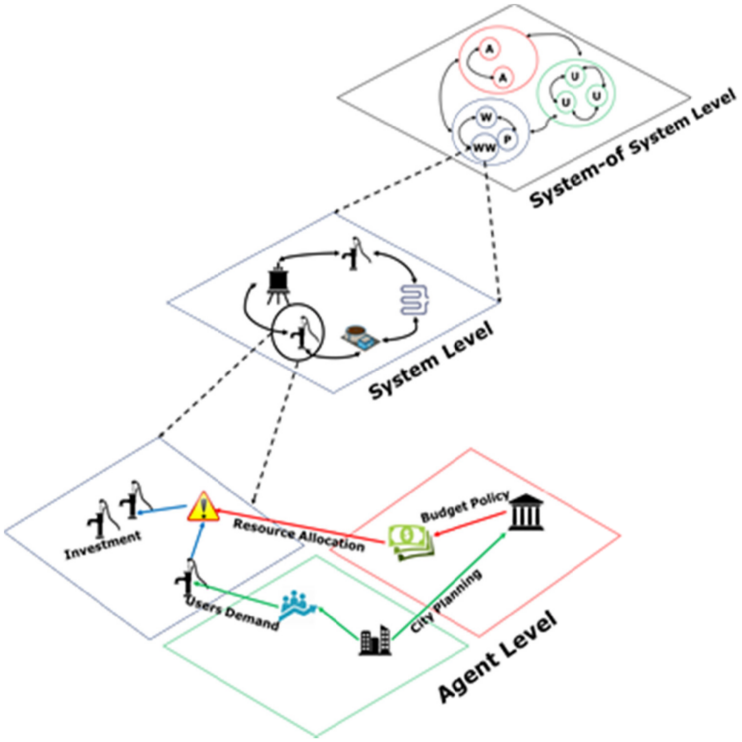
Facilitátora je možné použít i distribuovanější formě. V takovém případě facilitátor přímo nezprostředkovává poskytování služeb, ale pouze poskytuje kontaktní informace na vhodného agenta a očekává, že tito agenti se domluví sami na přímo.

8.3 Aplikace v bezpečnosti

Předchozí podkapitoly ve Vás mohly vzbudit pocit, že multiagentní systémy jsou čistě robotické systémy, ale není tomu tak. Agenty (a jejich systémy) je možno implementovat také čistě softwarově a vytvořit tak simulační prostředí, které pro nás řeší zvolený problém.

Právě v bezpečnosti se tento typ modelů využívá velmi extenzivně a to zejména pro svou univerzálnost, široké možnosti nastavení a schopnosti nechat se propojit s dalšími modely a systémy pro získání kvalitnějších výstupů o složitějších problémech.

Např. v oblasti ochrany kritické infrastruktury se takové systémy často využívají pro simulaci interakcí jednotlivých komponent takových systémů. Uvažujme např. systém na obr. 8.6.



Obrázek 8.6: Rámec multiagentního systému pro abstrakci modelování kritické infrastruktury (převzato z [64])

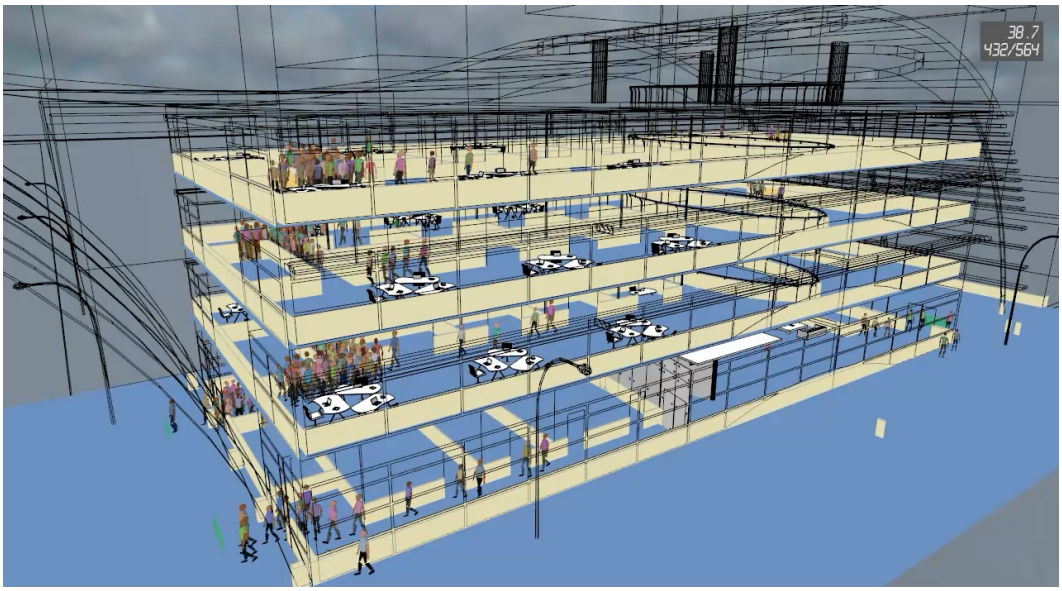
Jednotliví agenti v takové simulaci by hráli roli jednak odběratelů služeb infrastruktury, jednak by hráli roli také prvků KI samotných, které tyto služby poskytují. Tímto způsobem by mohly být simulovány různé situace, kterým infrastruktura bude vystavena a hledány problémy a jejich optimální řešení.

Existuje celá řada studií, které takto postupují [29, 64] a řada dalších, byť se v současnosti nezdá, že by tento přístup z hlediska pokroku ve výzkumu kritické infrastruktury byl nejperspektivnějším.

Kromě výzkumu se ale multiagentní systémy rutinně používají pro simulace evakuace z uzavřených prostor, popř. celých budov. Pro tyto účely jsou dokonce dostupné komerční nástroje, umožňující tyto systémy použít i osobám, které o umělé inteligenci vědí velmi málo. V dostupné literatuře jsou takové modely označovány jako behaviorální. Porovnání vlastnosti různých evakuačních modelů je dostupné např. v [47].

Z komerčně dostupných softwarových prostředků lze zmínit např. SIMULEX [43], viz obr. 8.7 nebo Pathfinder [77].

Tyto modely jsou validovány, tedy je ověřeno, že jejich výsledky poskytují dostatečně dobrou shodu se skutečnou evakuací z prostoru. Výhodou těchto modelů je schopnost využití 2D plánů budov, nebo dokonce 3D modelů prostor, do kterých jsou doplňování v měřítku jednotliví agenti reprezentující evakuované osoby. Agenti mohou mít odlišné vlastnosti odpovídající např. různým věkovým skupinám, nebo zdravotnímu stavu populace, která bude v dané budově.



Obrázek 8.7: Simulace evakuace budovy v balíku SIMULEX (převzato z [43])

Koncept multiagentních systémů je možno využít také pro simulaci plošné evakuace. Nasazení pro tento problém je však v experimentální fázi. Hlavní překážkou pro nasazení je obtížná validace takových řešení. Bavíme se zde totiž o jevech, které v praxi nenastávají příliš často, jelikož jsou spojovány s mimořádnými událostmi jako jsou extrémní záplavy, velké průmyslové havárie apod.

Dalším komplikujícím faktorem je, že většina osob v území se evakuuje vlastními silami (po vlastní ose). Z pohledu rychlosti evakuace a náročnosti její organizace je to jednoznačně pozitivum. Z pohledu modelování to ale představuje výzvu, protože nejsou k dispozici údaje o průběhu této evakuace. Pro simulaci evakuace území je tak nutné provést určité předpoklady o vybavenosti obyvatelstva vozidly a také čase, který zvolí pro svou evakuaci.

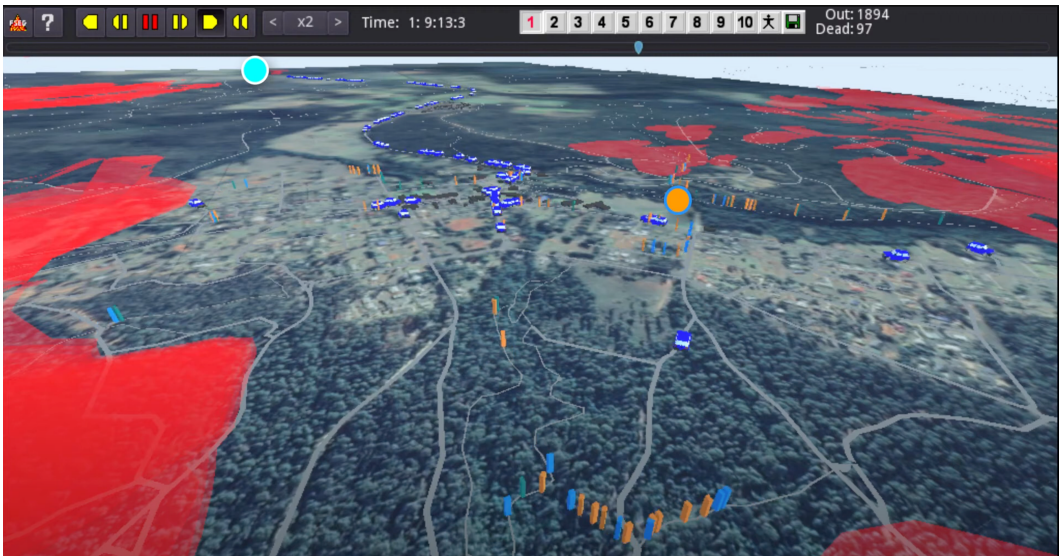
V rámci simulace evakuace uzavřeného prostoru nebo budovy odpovídá jeden agent jednomu evakuovanému. V případě simulace plošné evakuace tato paralela nemusí platit. Agent může reprezentovat evakuační vozidlo, ať už osobní nebo třeba autobus se schopností převést větší množství osob.

Pro realizaci simulace je pak potřeba zachytit modelem dynamiku běžného chování v modelované obci (přesuny do z práce, obchůzky, ...) až do okamžiku vyvolaní potřeby evakuace. Modelování běžných vzorů chování je přitom nutné pro realistické rozložení obyvatelstva v prostoru obce. Výrazné rozdíly budou např. mezi nocí, kdy lze předpokládat, že většina obyvatel bude doma a dopoledne v pracovní den, kdy naopak lze předpokládat přesun a pobyt obyvatel v práci.

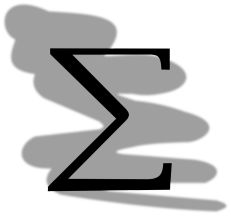
Plošná evakuace tak má neúměrně vysoké požadavky na data a simulace i kdyby se nám tyto údaje podařilo získat bude výpočetně náročná. Z tohoto důvodu se tento typ simulací v praxi dosud příliš nevyužívá. Představu o možnostech si lze udělat z experimentálních nasazení modelů jako je urbanEXODUS [33], viz obr. 8.8.

V tomto případě je evakuační model kombinován s modelem rozvoje lesního požáru, který ovlivňuje jednak, které oblasti jsou přímo zasaženy požárem a je nutné je tedy evakuovat, jednak vyvíjí tlak na systém přepravy, kdy v důsledku postupu požáru mohou přestat být dostupné některé evakuační trasy.

Celkově vzato je tento typ modelů velmi univerzální a je schopen poskytnout poměrně diverzifikované odpovědi na některé otázky, které by nasazení takových modelů pravděpodobně musely buďto zůstat nezodpovězené, nebo by byla použita pouze hrubá aproximace názorem experta na danou problémovou doménu.



Obrázek 8.8: Simulace urbanEXODUS lesní požár-chodci-automobily (převzato z [33])



Srhnutí

Multiagentní systémy lze považovat za zobecnění přístupu aplikovaného v celulárních automatech. Fungování systému je založeno na vzájemné spolupráci specializovaných agentů v systémů. Specializace umožňuje relativní jednoduchost implementace agenta, komunikace pak umožňuje propojení schopností jednotlivých agentů pro řešení složitých problémů, které by jinak bylo obtížné řešit.

V bezpečnostních oborech se multiagentní systémy využívají ve výzkumu, masivně jsou také nasazovány pro modelování evakuace z uzavřených prostor nebo budov. Pro tento typ modelování jsou dostupné také předpřipravené komerční modelovací nástroje.

Kontrolní otázky

1. Co je to komunikační akt?
2. Jaký je rozdíl mezí inteligentním agentem a multiagentním systémem.
3. Co je to performativ?
4. Jakou úlohu plní facilitátor?
5. Vzpomněli byste si na nějaké kategorie komunikačních aktů (vyjmenujte alespoň dva)?

Úkoly k zamyšlení

1. Prozkoumejte model heatbugs v NetLogu nebo jiném simulátoru multiagentních systémů.

Kapitola 9

Genetické algoritmy

Náhled kapitoly
V této kapitole se podíváme na použití evolučních strategií pro řešení některých zejména optimalizačních problémů. Tato kapitola pokrývá spíše doplňkové téma vzhledem k náplni předmětu. Proto věnujte pozornost spíše základní filozofii použití a návaznostem na další témata probíraná předmětem

Po prostudování této kapitoly budete
vědět

- jak funguje genetický algoritmus
- jakým způsobem se používají mutace, křížení a přirozený výběr
- jaká jsou úskalí použití genetických algoritmů

Čas pro studium
Na prostudování této kapitoly budete potřebovat dvě hodiny.

Genetické algoritmy vycházejí z pojetí evoluce tak jak jej popsal Charles Darwin ve své knizi *O původu druhů* [28]. Evoluce má dva hybné mechanismy – *geny*, jako základní prostředek kterým se předávají informace z předchozích generací a potomky a přirozený výběr, který preferuje jedince, jejichž genetická výbava je činí maximálně přizpůsobenými prostředí, ve kterém žijí.

Genetické algoritmy vycházejí přesně z tohoto konceptu a používají jej pro řešení především optimalizačních problémů. Podobně jako řada dalších používaných postupů ani genetické algoritmy nejsou schopny nalézt optimální řešení - tedy nejlepší možné řešení problému. Místo toho umožňuje nalézt pouze sub-optimální řešení - tedy nejlepší řešení dostupné ve vymezeném čase a stanovenými prostředky. V tomto smyslu tedy tyto algoritmy fungují podobně jako třeba *neuronové sítě*.

Jelikož u genetických algoritmů vycházíme z biologie používáme i podobnou terminologii. Kupříkladu pracujeme s *populací* – tedy souhrnem jedinců různých vlastností, kteří mající jinou „genetickou“ výbavu. Genetická výbava je v úvozovkách zo toho důvodu, že se jedná o užitečnou analogii popisující filozofii fungování. Jelikož výpočet probíhá zcela v počítači, jedinci ve skutečnosti nemají žádnou genetickou výbavu, pouze specifickým způsobem stanovené vlastnosti, se kterými je následně manipulováno podobně jako dědičnou informací živých organizmů.

Základním mechanismem, kterým genetické algoritmy postupují směrem k optimálnímu řešení, je kombinace vlastností jedinců v populaci pro odvození/změnu populace. Základní, nikoliv však jediným nástrojem k provedení takové optimalizace je *křížení*. V rámci křížení kombinujeme náhodně vlastnosti dvou nebo více jedinců vstupujících do procesu křížení. Křížením pak vznikne jedinec nový - *potomek*.

Genetické algoritmy jsou pouze inspirované ději v živé přírodě, lze tak v rámci jejich použití experimentovat s řadou postupů, které v přírodě buďto nejsou obvyklé nebo dokonce se v přírodě

nevyskytují. Cílem takových modifikací je pak obvykle zrychlit průběh výpočtu nebo zvýšit kvalitu dosaženého řešení.

Některé z takových postupů zmíníme ještě později v této kapitole.

Aby celý algoritmus směřoval k nějakému řešení je nutné použít nějaké optimalizační pravidlo. V přírodě tuto funkci plní schopnost reprodukce. Schopní jedinci mají větší šanci na reprodukci než ti méně úspěšní. V počítači musíme tuto funkci nahradit funkcí umělou, která umožní exaktně změřit úspěšnost jedinců v řešení právě našeho problému. Tuto funkci můžeme nazývat *fitness funkcí*. Bez přítomnosti fitness funkce je sice možné genetický algoritmus spustit, ale není možné dosáhnout optimálního řešení, protože v populaci by nedocházelo k optimalizaci, resp. i kdyby k optimalizaci docházelo, nebyli bychom schopni to zjistit.

9.1 Evoluční strategie

Křížení probíhá v iteracích. Iterace fungují jako diskrétní časové kroky. Filozoficky se iterace genericko algoritmu nejvíce blíží generační obměně v rámci populace rostlin nebo živočichů. Vzhledem k této analogii iterace obvykle nazýváme *generacemi*. Jednou z poměrně významných úloh analytika je pak rozhodnutí, jak přesně by generační obměna měla proběhnout. Uvažujme tři scénáře:

- úplná obměna
- částečná obměna s náhradou rodičů
- částečná obměna s koexistencí potomků a rodičů

Rozdíly v přístupu by měly být lépe patrné při grafickém vyjádření výše uvedených scénářů, viz obr. 9.1.

Ve všech třech případech nová generace do určité míry nahradí generaci původní. Technicky toto není úplně striktní požadavek, protože technicky lze zachovat každého jednoho jedince, ale z praktického hlediska to však nepřináší žádné výhody ba právě naopak. Ze zvětšující se populací narůstá výpočetní čas pro každou iteraci výpočtu. V populaci se pak mohou vyskytovat „lepší jedinci“, ale jejich šance na výběr do křížení bude s každou iterací stále menší a menší.

Generační obměna tak sleduje dva zásadní cíle - udržet populaci ve výpočetně zvládnutelném stavu a zlepši kvalitu jedinců v populaci měřenou fitness funkcí.

Mimochodem, každý jedinec představuje možné řešení problému. Fitness funkce nám pak umožňuje rozlišit kvalitu takového řešení.

V případě úplné obměny (viz obr. 9.1a) budou rodiče zcela nahrazení svými potomky. V populaci se tak v žádné iteraci nebudou vyskytovat zároveň rodiče a potomci. Takový typ generační obměny označujeme někdy jako *generační obměnu*. Na obr. 9.1a je taková strategie aplikována na celou populaci. Na obr. 9.1b se ale v rámci generační obměny změní pouze vybraná část populace. I v tomto případě se jedná o generační obměnu, jen se neděje na celé populaci. Zbytek populace v tomto případě přechází nezměněn do další iterace.

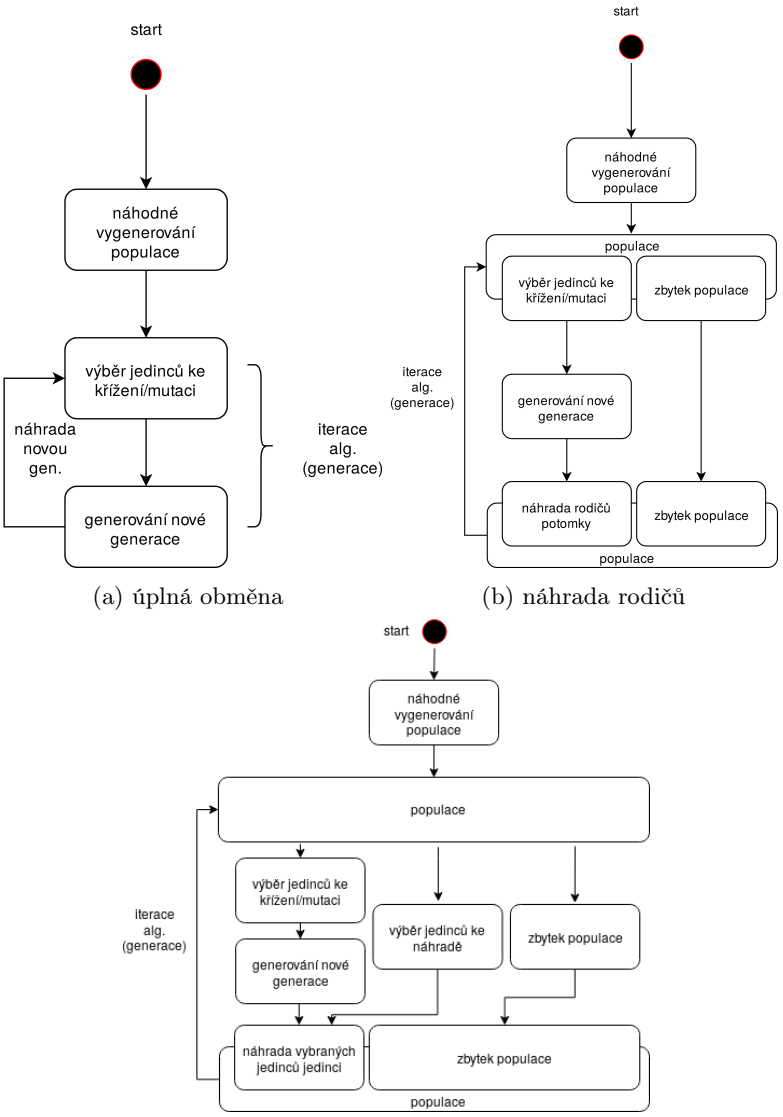
Na obr. 9.1c je znázorněna odlišná situace. Proces výběru ke křížení a výběru k náhradě jsou v tomto případě odděleny. Prakticky to znamená, že rodiče a potomci mohou koexistovat v jedné generaci. Takovou situaci označujeme jako *stálou evoluci*.

Jelikož genetické algoritmy jsou umělé můžeme si typ evoluce volit sami, všechny přístupy přitom mají určité výhody i nevýhody. Generační evoluce je většinou mnohem dynamičtější, změny ve vlastnostech populace jsou tak velmi rychlé. Nevýhodou tohoto přístupu však je, že v populaci mohou relativně jednoduše zaniknout z hlediska řešení problému zajímavé vlastnosti, které obvykle v populaci nejsou tak rozšířené.

Pokud nahradíme pouze část populace, dává nám to do rukou poměrně mocné nástroje. Evoluce tohoto typu je podstatně pomalejší - celá populace se obnoví/změní teprve po několika generacích. Zároveň lze uvažovat o jedincích, které budeme vybírat pro křížení. Můžeme třeba vybírat ty nejvhodnější jedince, identifikované fitness funkcí. Pokud umožníme koeexistovat rodičům a potomkům otevíráme se další možnosti pro vynucení zachování některých důležitých vlastností v populaci.

9.2 Křížení a mutace

Křížení jako takové umožňuje, aby jedinci v nové generací dědily vlastnosti svých předků. Problémem tohoto přístupu je, že různorodost těchto vlastností je konečná a daná iniciací (způsobem vygenerování)



(a) úplná obměna (b) náhrada rodičů

(c) koexistence s rodiči

Obrázek 9.1: Scénáře generační obměny v generickém algoritmu

počáteční populace. Nikde přitom není zajištěno, že požadované vlastnosti, tedy vlastnosti které by se měly objevit ve výsledku evoluce, jsou v populaci vůbec obsaženy nebo že je v rámci selekce nevyřadím.

Aby se předešlo stagnaci v evoluci, využívají v menší míře také metodu *mutace* jedinců. Zatímco v případě křížení dvou jedinců vytvoříme nové jedince kombinací, přibližně 1:1 vlastností rodičů, v případě mutace vybereme náhodně jednu vlastnost a náhodně ji změníme.

Mutací podstupuje ve srovnání s křížením méně jedinců. Cílem mutace je pomalé vnášení drobných změn do populace tak, aby její vývoj nestagnoval. Přesto není mutace schopna sama o sobě zajistit, aby v populaci byly zachovány žádoucí znaky. Z tohoto důvodu se někdy používá Best so Far (BSF) strategie. Strategie spočívá v tom, že jedinec nebo jedinci s nejlepšími vlastnostmi automaticky zůstává přítomni v populaci. Část populace tak vyjímáme z procesu generační obměny. Mohou tedy vstoupit do procesu křížení, ale zároveň nemohou být nahrazení novými jedinci až do doby, kdy se objeví nový jedinec nebo jedinci s lepšími vlastnostmi.

Předtím než se začneme zabývat selekčními strategiemi, podívejme se na matematické vyjádření toho, co jsme si v předchozích odstavcích pověděli.

Populace G v generaci t je tvořena jedinci $x_{t,i}$.

$$G(t) = \{x_{t,1}, x_{t,2}, ..., x_{t,N}\} \tag{9.1}$$

Schopnosti jednotlivců v populaci hodnotíme pomocí fitness funkce. Pro fitness funkci $f(x)$ by mělo platit, že je nezáporná. V takovém případě lze implementovat mechanismus tzv. *ruletového kola*. To znamená, že výběr např. pro křížení je zcela náhodný.

Pro vzorec (9.2) genetický algoritmus řeší maximalizaci fitness funkce.

$$P(i) = \frac{f(x_i)}{\sum_{j=1}^{N} f(x_j)}, i = 1, ..., N \tag{9.2}$$

Účelem selekce je výběr nebo chcete-li preference jedinců, kteří mají z hlediska řešení problému dobré vlastnosti. Ve vzorci (9.2) bude pravděpodobnost výběru jedince i $P(i)$ vyšší pro vyšší hodnoty fitness funkce. To je také důvod proč tento vzorec nelze použít pokud by fitness funkce nabývala záporných hodnot. Pokud je tedy fitness funkce nezáporná spočívá optimalizace v maximalizaci této funkce.

Je však potřeba dodat, že řadu problémů ať už vedoucích k minimalizaci funkce nebo na záporné hodnoty fitness funkce lze formálně upravit tak, aby podmínka nezápornosti byla splněna.

Výsledkem selekce podle vzorce (9.2) je preference jedinců s lepšími (nadprůměrnými) vlastnostmi na úkor jedinců s vlastnostmi horšími (podprůměrnými). Počet průměrných jedinců zůstává přibližně stejný.

Nevýhodou tohoto přístupu je nutnost dostatečně velké populace, se kterou pracujeme.

Pro připomenutí ze statistiky – při dostatečném počtu nezávislých pokusů lze předpokládat, že relativní četnosti těchto pokusů se budou blížit teoretické hodnotě pravděpodobnosti. Matematicky, lze toto tvrzení vyjádřit následovně (13):

$$\lim_{n\to\infty} P(|\frac{1}{n}\sum_{i=1}^{n} X_i - \frac{1}{n}\sum_{l=1}^{n} E(X_i)| < \epsilon) = 1 \tag{9.3}$$

Tuto náhodnou selekci je možné dále modifikovat, zejména s ohledem na to, že obvykle máme k dispozici pouze omezenou populaci, v takovém případě bohužel nemůžeme použít vzorec (9.2) determinističtější způsob, lze použít např. *zbytkový stochastický výběr* nebo řadu dalších strategií.

Výběr v takovém případě už není zcela náhodný - resp uměle upravujeme populaci tak, aby lépe odpovídala našim záměrům. To může znamenat, že jedinci, kteří jsou v populaci minoritní, avšak zároveň mají z hlediska řešení zajímavé vlastnosti mohou být do populace „naklonováni“ tak, aby se zvýšila uměle pravděpodobnost jejich výběru pro křížení a také pravděpodobnost toho, že cenná vlastnost z populace nevymizí.

Možných strategií se nabízí více, ale pro naše účely není potřeba tuto problematiku rozvíjet příliš do hloubky.

9.3 Možnosti nasazení genetických algoritmů

Podívejme se na praktickou ukázku použití genetických algoritmů na řešení problému obchodního cestujícího. Hezké řešení poskytoval dnes již nefunkční *FGA: Graphical TSP solver* [8]. Jak víme, problém obchodního cestujícího je tzv. *NP kompletní problém*, který není možné pro velký počet měst řešit standardními cestami, tedy numerickým výpočtem.

Řešení problému pomocí genetického algoritmu je znázorněno na obr. 9.2.

Na obr. 9.2 jsou znázorněny dvě různé konfigurace destinací obchodního cestujícího, v každém řádku jeden. Jednotlivá zastavení obchodního cestujícího jsou znázorněna pomocí teček (na obr. 9.2 vlevo). Uprostřed je pak náhodná volba cesty. Postupnou optimalizací se pak dostaneme k řešení prezentovanému na obr. 9.2 vpravo.

Jako fitness funkce v tomto případě slouží délka cesty, kterou obchodní cestující musí urazit, aby prošel všechny plánované destinace.

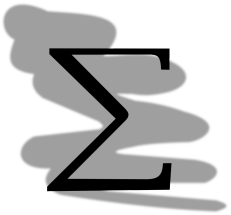
Současný stav řešení problému obchodního cestujícího pomoci genetických algoritmů lze najít v literatuře, např. [86].

Obrázek 9.2: Možné řešení problému obchodního cestujícího (převzato z [8])

Z hlediska dalšího nasazení lze genetické algoritmy nasazovat tam, kde si např. nejsme jisti možnou výchozí konfigurací nastavení algoritmů - např. neuronové sítě, resp. nastavení konfigurace sítě jako je počet neuronů ve skrytých vrstvách, popř. nastavení dalších parametrů adaptace sítě. Fitness funkce je pak odvozena z kvality predikce testovací množiny.

K výše uvedenému je ale potřeba dodat, že existují také jednodušší cesty, jak dosáhnout podobného výsledku a rychleji.

Genetické algoritmy tak sice mají své využití v praxi, ale jejich význam je spíše okrajový, popř. mohou posloužit pro řešení problémů kde metody klasické statistiky i ostatní metody strojového učení selhaly.



Shrnutí

Genetické algoritmy jsou optimalizační metodou založenou na evolučních principech. Možná řešení problému jsou představována jedinci. V běhu algoritmu dochází ke kombinaci vlastností jedinců v populaci (dvou nebo více) - *křížení* a nebo k jejich drobným náhodným změnám - *mutacím*.

Kvalita každého jedince je měřena pomocí jeho *fitness funkce*. Pro řešení vybíráme takového jedince, jehož hodnota fitness funkce je nejlepší.

Pro posílení schopností algoritmu dosáhnout výsledku jsou někdy využívány metody jako BSF nebo oslabení mechanismu *ruletového kola* pro zvýšení pravděpodobnosti zachování pozitivních nebo jinak zajímavých znaků v populaci.

Z hlediska nasazení se genetické algoritmy používají spíše jako doplňkový nástroj pro posílení některých vlastností dalších metod, jako jsou např. neuronové sítě. Samostatně pak mohou být používány pro řešení problémů, které pomocí jiných metod by nebyly řešitelné.

Kontrolní otázky

1. Jaký je rozdíl mezi křížením a mutací?
2. Vysvětlete princip ruletového kola pro selekci jedinců?
3. V čem spočívá přínos BSF?

Literatura

[1] CLIPS: A Tool for Building Expert Systems.

[2] The missing package manager for macOS (or Linux) — Homebrew.

[3] NetLogo.

[4] Prohlížeč Vivaldi | Výkonný, osobní a soukromý webový prohlížeč.

[5] RStudio.

[6] Swarm.

[7] ISO/IEC 19501:2005 (OMG-UML VER 1.3) Information technology – Open Distributed Processing – Unified Modeling Language (UML) Version 1.4.2, 2005.

[8] FGA: Graphical TSP solver, 2007.

[9] ArgoUML, 2011.

[10] Dia draws your structured diagrams: Free Windows, Mac OS X and Linux version of the popular open source program, 2014.

[11] Apache Cassandra, June 2018. original-date: 2009-05-21T02:10:09Z.

[12] MongoDB, June 2018. original-date: 2009-01-15T16:15:18Z.

[13] OpenAI, 2018.

[14] R: The R Project for Statistical Computing, 2018.

[15] Repast Suite Documentation, 2018.

[16] draw.io, 2019.

[17] Home - Keras Documentation, 2019.

[18] StarUML, 2019.

[19] Free Photo | Assembly line production of new car automated welding of car body on production line robotic arm on car production line is working, 2024.

[20] Apache. Apache MXNet (Incubating) - A flexible and efficient library for deep learning., 2019.

[21] Nisha Arya. Gartner Hype Cycle for AI in 2023, 2023. Section: KDnuggets Originals.

[22] Wp Bartlett and Ga Banker. An electron microscopic study of the development of axons and dendrites by hippocampal neurons in culture. II. Synaptic relationships. *The Journal of Neuroscience*, 4(8):1954–1965, August 1984.

[23] Bruce Blaus. Neuron, 2013. Page Version ID: 1197710164.

[24] Mariusz Bojarski, Ben Firner, Beat Flepp, Larry Jacker, Urs Muller, Karol Zieba, and Davide Del Tesla. End-to-End Deep Learning for Self-Driving Cars, August 2016.

[25] Brian Christian. *The Alignment Problem*. Atlantic Books, London, 2021.

[26] E. F. Codd. *Cellular Automata*. Academic Press, 1968.

[27] Cycorp. The Syntax of CycL.

[28] Charles Darwin. *On the Origin of Species*. Pan Macmillan, Londýn, 2017 edition, 1859.

[29] Enrique de la Hoz, Jose Manuel Gimenez-Guzman, Ivan Marsa-Maestre, Luis Cruz-Piris, and David Orden. A Distributed, Multi-Agent Approach to Reactive Network Resilience. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, pages 1044–1053, Richland, SC, 2017. International Foundation for Autonomous Agents and Multiagent Systems. event-place: São Paulo, Brazil.

[30] D. Dibenski. Auklet flock, Shumagins 1986, 2006.

[31] Tim Finin, Jay Weber, Gio Wiederhold, Michael Gensereth, and Richard Fritzson. *DRAFT Specification of the KQML Agent-Communication Language*. DARPA, 1993.

[32] FIPA. Agent Communication Language Specifications, 2002.

[33] FSEGresearch. urbanEXODUS wildfire-pedestrian-vehicle coupled evacuation scenario, 2020.

[34] M. Gardner. Mathematical Games - The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, 1970.

[35] Michael R. Genesereth and Richard E. Fik. *Knowledge Interchange Format version 3.0 Reference Manual*. Stanford University, Stanford, 1992.

[36] Google. Cloud Text-to-Speech - Speech Synthesis, 2019.

[37] Google. TensorFlow, September 2019. original-date: 2015-11-07T01:19:20Z.

[38] Frauke Guenther. *Package 'neuralnet'*. 2016.

[39] Suzana Herculano-Houzel. The Human Brain in Numbers: A Linearly Scaled-up Primate Brain. *Frontiers in Human Neuroscience*, 3, November 2009.

[40] Michal Hradiš. Konvoluční neuronové sítě, 2015.

[41] Joel Hruska. IBM Watson Recommends Unsafe Cancer Treatments, July 2018.

[42] IBM. IBM Developer : Download : IBM Rational Software Architect Designer, June 2014.

[43] IESVE. SIMULEX - homepage, 2019.

[44] Piotr Jakubas. Forest Fire - Simulation on Cellular Automata _3, 2008.

[45] Piotr Jakubas. ForestFire – forest fire simulation, 2010.

[46] A. Kubík. *Inteligentní agenty - tvorba aplikačního software na bázi multiagentních systémů*. Computer Press, Brno, 2004.

[47] Erica D. Kuligowski, Richard D. Peacock, and Bryan L. Hoskins. *A Review of Building Evacuation Models*. NIST, 2 edition, 2010.

[48] Christopher G. Langton. Self-reproduction in cellular automata. *Physica D: Nonlinear Phenomena*, 10(1):135–144, 1984.

[49] Yann LeCun, Corinna Cortes, and Christopher J. C. Burges. MNIST handwritten digit database, 1998.

[50] Yann LeCunn, Corinna Cortes, and Christopher J. C. Burges. MNIST Dataset v CSV formátu, 1998.

[51] Aristid Lindenmayer. Mathematical models for cellular interaction in development. *Journal of Theoretical Biology*, 18(3):280–299, 1968.

[52] Richard Ling. Conus textile exhibits a cellular automaton pattern on its shell, 2005.

[53] Edwin Martin. John Conway's Game of Life.

[54] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, December 1943.

[55] Microsoft. The Microsoft Cognitive Toolkit - Cognitive Toolkit - CNTK, 2019.

[56] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry, Expanded Edition.* The MIT Press, Cambridge, Mass, 3 edition, 1987.

[57] MPO. *Iniciativa Průmysl 4.0.* MPO, Praha, 2016.

[58] MS. Overview of Microsoft Graph, 2019.

[59] A. Newell, J. C. Shaw, and H. A. Simon. Report on a general problem-solving program. In *Proceedings of the International Conference on Information Processing*, pages 256–264, 1959.

[60] NVIDIA. NVIDIA Keynote at SIGGRAPH 2023, 2023.

[61] OMG. About the Unified Modeling Language Specification Version 2.5.1, 2017.

[62] OpenAI. *Preparedness Framework (beta).* OpenAI, 2023.

[63] Richard E. Pattis. *Karel The Robot: A Gentle Introduction to the Art of Programming.* John Wiley & Sons, 2 edition, 1995.

[64] Pereyra Jose, He Xudong, and Mostafavi Ali. Multi-Agent Framework for the Complex Adaptive Modeling of Interdependent Critical Infrastructure Systems. In *Construction Research Congress 2016*, pages 1556–1566.

[65] R Project. Ubuntu Packages for R, 2018.

[66] Craig Reynolds. Flocks, herds and schools: A distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987.

[67] Craig Reynolds. Boids - Background and Update, 2001.

[68] Jeremy Rifkin. *The End of Work: The Decline of the Global Labor Force and the Dawn of the Post-Market Era.* Tarcher, New York, 1996.

[69] Frank Rosenblatt. *The Perceptron - A Perceiving and Recognizing Automaton.* Cornell Aeronautical Laboratory, Inc., Buffalo, N.Y., 1957.

[70] Christoph Roser. Industry 4.0 (cs), 2017.

[71] Salvatore Sanfilippo. Redis, June 2018. original-date: 2009-03-21T22:32:25Z.

[72] David Schor. A11 Bionic dice schema, 2018.

[73] A. Shueller. Cellular Automata Fire Model, 2010.

[74] Solkoll. Fractal weeds, 2005.

[75] Stanford University. Protégé 5, 2019.

[76] Joseph Steppan. MNIST Dataset Examples, 2017.

[77] ThunderHead Engineering. Parthfinder, 2019.

[78] Apostol Vassilev, Alina Oprea, Alie Fordyce, and Hyrum Anderson. *NIST AI 100-2e2023 Adversarial Machine Learning: A Taxonomy and Terminology of Attacks and Mitigations.* NIST, Gaithersburg, 2024.

[79] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention Is All You Need, August 2023. arXiv:1706.03762 [cs].

[80] Visual Paradigm International Ltd. Ideal Modeling & Diagramming Tool for Agile Team Collaboration, 2019.

[81] John von Neuman and Arthur W. Burks. *Theory of Self-Reproducing Automata.* University of Illinois Press, 1966.

[82] W3C. OWL 2 Web Ontology Language Document Overview (Second Edition), 2012.

[83] Joseph Weizenbaum. *Computer Power and Human Reason: From Judgment to Calculation.* W. H. Freeman & Co., San Francisco, 1976.

[84] Jackie Wiles. What's New in Artificial Intelligence from the 2022 Gartner Hype Cycle, 2022.

[85] Stephen Wolfram. *A New Kind of Science.* Wolfram Media, Champaign, IL, 2002.

[86] Ai-Hua Zhou, Li-Peng Zhu, Bin Hu, Song Deng, Yan Song, Hongbin Qiu, and Sen Pan. Traveling-Salesman-Problem Algorithm Based on Simulated Annealing and Gene-Expression Programming. *Information*, 10(1):7, January 2019.

[87] Ondřej Černý. *Modelování rozvoje požáru s využitím prostředků umělého života.* VŠB-TU Ostrava, Fakulta bezpečnostního inženýrství, Ostrava, 2010.

[88] Ondřej Černý and Pavel Šenovský. Nekonvenční metody modelování rozvoje požáru. *Sborník vědeckých prací, řada Bezpečnostní inženýrství*, 5(2):23–30, 2010.

[89] Pavel Šenovský. *Expertní systémy.* VŠB-TU Ostrava, Fakulta bezpečnostního inženýrství, Ostrava, 2007.

[90] Pavel Šenovský. *Bezpečnostní informatika 1.* VŠB-TU Ostrava, Fakulta bezpečnostního inženýrství, Ostrava, 8 edition, 2017.

Slovník

AGI Artificia General Inteligence.

AI Artificia Inteligence.

API Application Interface.

ASIC Application Specific Integrated Circuit.

BRMS Business Rule Management System.

BSF Best so Far.

CASE Computer Aided System Engineering.

CIA Confidentiality, Integrity, Availability.

CSV Comma Separated Values.

DFD Data Flow Diagram.

DLSS Deep Learning Super-Sampling.

DSS Decision Suport System.

ERD Entity Relationship Diagram.

ERP Enterprise Resource Planning.

GUI Grafické uživatelské rozhraní.

IDS Intruder Detection System.

IoT Internet of Things.

IPS Intruder Prevention System.

KQML Knowledge Query and Manipulation Language.

LLM Large Language Model.

ML Machine Learning.

NLP Natural Language Processing.

NPU Neural Processing Unit.

ODBC Open Database Connect.

OOP Object Oriented Programming.

PLC Programmable Logic Controler.

ReLU Rectified Linear Unit.

TPU Tensor Procesor Unit.

TRINS Transportní informační a nehodový systém.

TTS Text To Speech.

UI Umělá inteligence.

UML Unified Modeling Language.

[title=Seznam zkratek]

Rejstřík

, 27

agent
    aktátor, 128
    deliberativní, 129
    reaktivní, 128
    senzor, 128
AGI, 30
aktivační funkce, 67
    ReLU, 50
    sigmoida, 50
automatizace, 25

backpropagation, 54
big data, 22
Boidi, 115
báze znalostí, 102

CASE, 86
cellular automata, 111
celulární automat, 127, 130
celulární automaty, 111
ChatGPT-4, 75
CIA, 82
CLIPS, 103
    fakta, 105
    pravidla, 105
    šablona, 103
computer vision, 24
confusion matice, 64
Conwayova hra života, 112
Copilot, 75

data science, 23
datová věda, 23
deep learning, 23
diagramy UML, 87
DSS, 21
důvěrnost, 82

emergence, 130
emergence efekt, 21, 114, 127
ERD, 85
evasion, 82
expertní systém, 101
    diagnostický, 102
    hybridní, 102
    plný, 101
    plánovací, 102
expertní systémy
    prázdné, 101
extrakce modelu, 83

facilitátor, 131
fitness funkce, 136
fraktály, 117

generace, 136
generativní AI, 27
generační obměna, 136
Google Bard, 75

halucinace, 29
hloubkové učení, 23

instance třídy, 88, 89
inteligentní agent, 127, 128
Internet věcí, 26
IoT, 26

komunikační akt, 130
konceptualizace, 102
konvoluční sítě, 68
KQML, 130
kritická infrastruktura, 114
křížení, 135, 136

Lindenmayerovy systémy, 117
LLM, 20
LM Studio, 78
Logo, 119
lokální přechodová funkce, 111

machime learning, 23
metoda, 86
ML, 23
multiagentní systém, 130
multiagentní systémy, 127
multimodalita, 75
mutace, 137
MYCIN, 101

nervový vzruch, 48
neuronová síť, 47
neuronové sítě, 20, 135
NLP, 26
NP kompletní problém, 138
NPU, 19

objektově orientované programování, 85
ontologie, 103
OOP, 85

parametr modelu, 52
Perceptron, 20
perceptron, 48
performativ, 131
PLC, 25
poisonig, 82
populace, 135
potomek, 135
počítačové vidění, 24
prahování, 48, 49
privacy, 82
program
    imperativní, 65
    symbolické, 65
promluva, 17
proměnná
    nominální, 39
    ordinální, 39
průmysl 4.0, 26
přeučení, 55
přeučení sítě, 67

R, 33
    Data.Frame, 38
    export dat, 40
    faktor, 39
    import dat, 40
    operátory, 36
    vektory, 37
    vykreslování grafů, 41
    základní funkce, 37
    základní statistické funkce, 41
Reynoldsův model shlukování ptáků, 115
rozhodovací strom, 55
ruletové kolo, 138

samoorganizace, 114
strojové učení, 22, 23
stálá evoluce, 136
symbol
    aktivační funkce, 67
    convolution, 68
    dropout, 67
    flatten, 73
    fullyconnected, 67
    pooling, 69
    proměnná, 67
    SoftmaxOutput, 70
syntetická data, 27
systémy pro podporu rozhodování, 21

text to speech, 57
TTS, 57
Turing, Alan, 20
Turingův stroj, 20
třída, 88, 89

UML, 21, 85
    Activity Diagram, 97
    collaboration diagram, 96
    Component Diagram, 97
    Deployment Diagram, 98
    Diagram komponent, 97
    diagram nasazení, 98
    diagram spolupráce, 96
    diagram činností, 97
    Model jednání, 91
    scénáře činností, 95
    sekvenční diagram, 95
    Sequence Diagram, 95
    State Diagram, 93
    stavový diagram, 93
    třídní diagram, 88
    Use Case Diagram, 91
UML vazby
    agregace, 90
    asociace, 89
    generalizace, 90
    kompozice, 90
    orientovaná asociace, 90
    závislost, 90
umělá inteligence, 17
    expertní systém, 20
    genetické algoritmy, 135
    multiagentní systémy, 21
    neuronové sítě, 22
    strojové zpracování jazyka, 17
Use Case
    Actor, 91
    extend, 91
    include, 92
učení s učitelem, 54

velká data, 22
virtuální asistent, 18
vrstva
    pracovní, 51
    skrytá, 51
    vstupní, 51
    výstupní, 51

Watson, 101
Wolframův 1D automat, 114, 119

zbytkový stochastický výběr, 138
zpětné šíření chyb, 54

úzká AI, 29