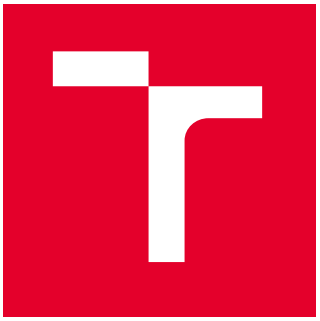


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

NÁSTROJ PRO PODPORU PŘÍPRAVNÉ FÁZE PENETRAČNÍHO TESTOVÁNÍ

SUPPORT TOOL FOR INITIAL PHASE OF PENETRATION TESTING

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Dominik Žáček**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Pavel Sikora**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Bc. Dominik Žáček ***ID:*** 221587

Ročník: 2 ***Akademický rok:*** 2023/24

NÁZEV TÉMATU:

Nástroj pro podporu přípravné fáze penetračního testování

POKYNY PRO VYPRACOVÁNÍ:

Hlavním cílem diplomové práce je navrhnout a implementovat pokročilý nástroj pro zvýšení efektivity přípravné fáze penetračního testování. Nástroj bude automaticky přiřazovat penetrační testery k dílčím úkolům, přičemž bude zohledňovat jejich znalosti a zkušenosti, aby byl výběr co nejvhodnější. Po provedení penetračního testu bude každý tester individuálně hodnocen pro budoucí optimalizaci algoritmu. V teoretické části navrhněte analyzujte současný stav problematiky se zaměřením na modely strojového učení, v praktické části implementujte program a ověřte jeho funkčnost. Přístup programu bude využívat definované API a výstup bude ve formátu JSON. Vlastní implementaci realizujte v jazyku Python.

DOPORUČENÁ LITERATURA:

[1] GANGUPANTULU, R.; CODY, T.; PARK, P.; RAHMAN, A.; EISENBEISER, L.; RADKE, D.; CLARK, R.; REDINO, C. Using Cyber Terrain in Reinforcement Learning for Pene- tration Testing. 2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS), 2022, s. 1–8. ISBN 978-1-6654-8356-8.

[2] STONE, G. B.; TALBERT, D. A.; EBERLE, W.; Utilizing Real-Time Strategy for Penetration Testing. International Journal of Chaotic Computing (IJCC), 2022, roč. 8, č. 1, s. 196–203. ISSN 2046-3359.

Termín zadání: 5.2.2024 ***Termín odevzdání:*** 21.5.2024

Vedoucí práce: Ing. Pavel Sikora

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Tato práce se zabývá vývojem pokročilého nástroje určeného pro zefektivnění týmového penetračního testování. Nástroj pracuje tak, že automaticky přiřazuje úkoly penetračním testerům na základě schopností a historického výkonu. Teoretická část práce podrobně analyzuje různé metody řešení problému přiřazení, zejména Maďarskou metodu a lineárním programováním. V teoretické části následuje návrh dvou-krokového algoritmu pro přiřazení úkolů. Dále je detailně popsán princip fungování neuronových sítí, které jsou základem druhého kroku přiřazení. V rámci práce byly také vytvořeny unikátní metody pro generování dvou datových sad. Bylo implementováno rozhraní pro přiřazení úkolů a byly navrženy metriky určující kvalitu přiřazení. Výsledkem je nástroj, který výrazně zefektivňuje přidělení úkolů penetračním testerům a zvyšuje celkovou efektivitu penetračního testování v týmech.

KLÍČOVÁ SLOVA

lineární programování, maďarská metoda, neuronové sítě, penetrační testování, problém přiřazení, přiřazení na základě schopností, strojové učení

ABSTRACT

This thesis deals with the development of an advanced tool designed to make team penetration testing more efficient. The tool works by automatically assigning tasks to penetration testers based on skills and historical performance. The theoretical part of the thesis analyzes in detail various methods for solving the assignment problem, in particular the Hungarian method and linear programming. The theoretical part continues with the design of a two-step algorithm for task assignment. Then, the principle of the neural networks underlying the second step of the assignment is described in detail. Unique methods for generating two datasets have also been developed as part of the work. An interface for task assignment has been implemented and metrics to determine the quality of the assignment have been proposed. The result is a tool that significantly streamlines the assignment of tasks to penetration testers and increases the overall efficiency of penetration testing teams.

KEYWORDS

assignment problem, Hungarian method, linear programming, machine learning, penetration testing, skill-based assignment

ŽÁČEK, Dominik. *Nástroj pro podporu přípravné fáze penetračního testování* . Diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2024. Vedoucí práce: Ing. Pavel Sikora

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Dominik Žáček
VUT ID autora:	221587
Typ práce:	Diplomová práce
Akademický rok:	2023/24
Téma závěrečné práce:	Nástroj pro podporu přípravné fáze penetračního testování

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing.Pavlu Sikorovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Obsah

Seznam obrázků

Úvod

V rychle vyvíjejícím se odvětví kybernetické bezpečnosti se simulace kybernetických útoků, takzvané penetrační testování, ukázaly jako dobrá strategie pro zajištění bezpečnosti informačních systémů. Penetračním testováním mohou organizace aktivně identifikovat zranitelná místa a díky této znalosti významně zdokonalit bezpečnost jejich systémů. Důležité a rozsáhlé testy systémů jsou zpravidla realizovány v týmu penetračních testerů. Účinnost těchto testů potom závisí na efektivním a kvalitním přidělováním úkolů penetračním testerům. Přidělené úkoly musí co nejlépe odpovídat jejich schopnostem, zároveň je žádoucí, aby každý tester dostal přiřazeno tolik úkolů, kolik je schopen splnit v určitém časovém úseku.

Tato práce si klade za cíl navrhnout a implementovat pokročilý nástroj pro zefektivnění zmíněné přípravné fáze penetračního testování, kdy dochází k přiřazení úkolů jednotlivým testerům. S využitím optimalizačních algoritmů a strojového učení nástroj automaticky přiřadí penetrační testery k dílčím úkolům. Nejprve na základě schopností, jenž testeři uvedou v dotazníku. Ve druhé fázi z historických záznamů o odvedené práci testerů.

První kapitola práce analyzuje optimalizační algoritmy pro problematiku přiřazení. Druhá kapitola navrhuje algoritmus přiřazení, který zahrnuje i záznamy o odvedené práci. Ve třetí kapitole je popsána teorie neuronových sítí, na kterých je algoritmus založen. Implementaci samotného nástroje je věnována kapitola čtvrtá, v rámci níž byly také vytvořeny unikátní metody pro generování dvou datových sad. První datová sada je tvořena úkoly a penetračními testery, zatímco druhá sada obsahuje záznamy o provedené práci. Dále bylo vytvořeno rozhraní pro přiřazení úkolů založené na dvou nejlepších algoritmech. Byly vytvořeny metriky pro měření kvality přiřazení. Proběhla i implementace algoritmu pro přiřazení na základě záznamů o práci. V kapitole porovnání jsou na vygenerované datové sadě porovnány algoritmy přiřazení a různé modely neuronových sítí využité pro přiřazení dle záznamů o práci.

1 Problém přiřazení

Obsah této kapitoly se zabývá optimálním přiřazením pracovníků k úkolů. Úloha optimálního přiřazení je dobře známý a relativně starý matematický problém optimalizace. Ve vojenství se výzkum této problematiky nazývá operační analýza [3]. Existují hlavní dva druhy problémů optimálního přiřazení:

- Problém lineárního přiřazení,
- problém kvadratického přiřazení.

Problém kvadratického přiřazení je situace, kdy jsou například rozmisťovány budovy na stavebních parcelách. Mezi budovami má být transportováno zboží v určitých počtech. Řešením problému je nejefektivnější transport zboží. Z podstaty problému je zřejmé, že má dvě takzvané kvalifikační matice, vzdálenost mezi budovami a počet transportovaného zboží mezi nimi. Problém kvadratického přiřazení o kvalifikačních maticích velikosti $n \cdot n$ má shodnou složitost jako problém lineárního přiřazení o kvalifikačních maticích velikosti $n^2 \cdot n^2$. [2]

Tato práce se dále zabývá pouze problémem lineárního přiřazení (dále jen problém přiřazení), protože problém kvadratického přiřazení se pro ni nejeví příliš relevantní. O problému přiřazení říkáme, že je vyvážený, pokud je počet úkolů a pracovníků stejný. Naopak pokud se počet pracovníků a úkolů liší, o problému můžeme prohlásit, že je nevyvážený.

Při opuštění myšlenky matematické optimalizace se na problém lze také dívat alternativním pohledem, kdy cílem je propojit pár *pracovník* a *úkol*. Řešením tohoto problému je algoritmus Gale-Shapley [6]. Dalšími vhodnými řešeními jsou jeho variace například *Hospitals/Residents problem* [7]. Nevýhodou tohoto přístupu je neschopnost pracovat s kvalitou přiřazení tj. spojitou hodnotou. Algoritmus pracuje pouze s hodnotou binární (vhodná či nevhodná dvojice).

1.1 Optimalizační problém

Na problém přiřazení se lze dívat jako na specifickou úlohu lineárního programování [1], kterému je věnována podkapitola 1.1.1. Nejčastějším řešením problému přiřazení je ovšem *Maďarská metoda* nazývaná také jako *Maďarský algoritmus* či po svých autorech jako *Kuhn-Munkres algoritmus* [4], viz podkapitola 1.1.2. Mezi další řešení se řadí *Aukční algoritmus* [8]. Ten již v dalších kapitolách není rozebírán protože v [5] autoři prokázali, že Maďarský algoritmus je pro tento typ problému efektivnější.

1.1.1 Lineární programování

Lineární programování je matematická metoda používaná k nalezení optimálního způsobu využití omezených zdrojů k dosažení daného cíle. Zahrnuje vytvoření matematického modelu problému definovaného lineární účelovou funkcí, která má být maximalizována nebo minimalizována například:

$$min \quad 5x_1 + 10x_2,$$

a také sadou lineárních (ne)rovnic:

$$x_1 + x_2 \geq 10,$$

$$x_1, x_2 \geq 0,$$

jenž se nazývají omezení. [10]

Nejjednodušší metodou pro řešení problémů lineárního programování je Simplexová metoda. Naproti tomu mezi nejpoužívanější a nejefektivnější metody patří metody založené na nalezení vnitřních bodů, například *neproveditelný primální-duální algoritmus* [11]. Pro řešení problémů lineárního programování je k dispozici také několik softwarových nástrojů a knihoven. Tato práce využívá Python knihovnu `scipy` a její funkci `optimize.linprog`.

Sestavení problému přiřazení

V následujícím textu je sestaven problém přiřazení jako lineární program. Jsou uvažovány stejné podmínky pro přiřazování jako u Maďarské metody 1.1.2, pro možnost srovnání obou metod. Obecně, mimo toto srovnání, je ale možné úlohu lineárního programování mnohem více zobecnit. Nemusí být dodržena například podmínka Maďarského algoritmu, kdy nesmí být přiřazení úkolů u_i nebo u_{i+1} pro pracovníka p_j povinné. Dále je možné, opět na rozdíl od Maďarkého algoritmu, nastavit maximální počet úkolů přiřazených pracovníkovi p_j. To může být v praxi velice vhodné například pro pracovníky se zkráceným úvazkem. Úloha jako taková nepřináší žádná omezení počtu pracovníků vůči počtu úkolů, nemusí se tedy jednat o vyvážený problém přiřazení.

Např. nechť jsou čtyři pracovníci a čtyři úkoly s kvalifikací danou na Obr. 1.1. Uvažovanému problému přiřazení bude odpovídat účelová funkce [12]:

$$\begin{aligned} min \quad & 8x_{1,1} + 4x_{1,2} + 10x_{1,3} + 14x_{1,4} + 16x_{2,1} + 6x_{2,2} + 20x_{2,3} + 16x_{2,4} + 24x_{3,1} + \\ & 10x_{3,2} + 8x_{3,3} + 10x_{3,4} + 12x_{4,1} + 6x_{4,2} + 14x_{4,3} + 28x_{4,4} \ , \end{aligned}$$

kde $x_{i,j}$ reprezentuje binární hodnotu zda-li má dojít k přiřazení úkolu u_i pro pracovníka p_j. Dále je nutné sestavit omezení. Ke každému úkolu může být přiřazen

	P_1	P_2	P_3	P_4	
U_1	8	4	10	14	$mr_1 = 4$
U_2	16	6	20	16	$mr_2 = 6$
U_3	24	10	8	10	$mr_3 = 8$
U_4	12	6	14	28	$mr_4 = 6$

Obr. 1.1: Kvalifikační matice s minimálními hodnotami každého řádku.

právě jeden pracovník:

$$x_{1,1} + x_{1,2} + x_{1,3} + x_{1,4} = 1$$
$$x_{2,1} + x_{2,2} + x_{2,3} + x_{2,4} = 1$$
$$x_{3,1} + x_{3,2} + x_{3,3} + x_{3,4} = 1$$
$$x_{4,1} + x_{4,2} + x_{4,3} + x_{4,4} = 1$$

A také každému pracovníkovi může být přiřazen právě jeden úkol:

$$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} = 1$$
$$x_{1,2} + x_{2,2} + x_{3,2} + x_{4,2} = 1$$
$$x_{1,3} + x_{2,3} + x_{3,3} + x_{4,3} = 1$$
$$x_{1,4} + x_{2,4} + x_{3,4} + x_{4,4} = 1$$

Nakonec jsou nastaveny meze ve kterých jsou proměnné x definovány:

$$\forall i, j \in \{1, 2, \ldots, 4\}, \quad 0 \leq x_{i,j} \leq 1$$

Výstupem bude vektor s optimálními hodnotami promněnných $(x_{1,1}, x_{1,2} \ \ldots \ x_{4,4})$. Výsledkem této úlohy, získaným za použití Python knihovny `scipy` konkrétně její funkce `optimize.linprog`, je vektor $\mathbf{v}^T = (1,0,0,0,0,0,0,1,0,0,1,0,0,1,0,0)$, který znázorňuje optimální přiřazení nacházející se na pozici každé jedničky. Jako optimální tedy vzešlo přiřazení dvojic úkolů u a pracovníků p následující: (u_1, p_1), (u_2, p_4), (u_3, p_3), (u_4, p_2). Minimalizovaná celková veličina tedy dle kvalifikace dané Obr. 1.1 je $8 + 16 + 8 + 6 = \mathbf{38}$.

Jak lze z příkladu pozorovat, náročnost problému s počtem úkolů pu a počtem pracovníků pp roste lineárně. Konkrétně je potřeba $pu \cdot pp$ proměnných v účelové funkci a $pu + pp$ základních omezení. Navíc musí být každá proměnná omezena v intervalu $0 \leq x_i \leq 1$ pro kompletní sestavení problému.

1.1.2 Maďarská metoda

Oproti lineárnímu programování je algoritmus mnohem méně obecný a potřebuje specifičtější vstup. Díky tomu je rychlejší a jeho časová složitost dosahuje $\phi(n^3)$. Jako vstup algoritmu je nutné mít n pracovníků a n úkolů (počet pracovníků tedy musí být stejný jako počet úkolů). Algoritmus tudíž pracuje pouze s vyváženým problémem přiřazení. Nesmí existovat kritérium nutnosti přiřadit úkol u_i pro pracovníka p_j, pro $i, j \in \{x | 1 \leq x \leq n\}$. Pro každou dvojici úkol u_i a pracovník p_j musí být známa kvalifikace $q_{i,j}$ pracovníka p_j vykonat úkol u_i, tvořící takzvanou kvalifikační matici Q. Z kvalifikační matice Q je možné vybrat každý řádek i a každý sloupec j právě jednou tj. každý úkol u_i je přiřazen právě jednomu pracovníkovi p_j. [9]

Při splnění dříve definovaných podmínek algoritmus zaručeně najde takové řešení, že

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} q_{i,j} \ ,$$

kde $q_{i,j}$ je kvalifikace pracovníka p_j pro algoritmem přiřazený úkol u_i. Pokud převrátíme hodnoty $q_{i,j} \in Q, \forall i, j$ například jako $q_{i,j} = max(Q) - q_{i,j}$ výsledkem algoritmu vůči původnímu $q_{i,j}$ potom bude naopak

$$max \sum_{i=1}^{n} \sum_{j=1}^{n} q_{i,j} \ ,$$

kde $max(Q)$ je maximální hodnota z kvalifikační matice Q.

Postup

Algoritmus Maďarské metody obsahuje následující postup [1]:

1. Pro řádek $i \in N$, kde $N = \{x | 1 \leq x \leq n\}$ je nalezena minimální hodnota řádku $mr_i = min(q_{i,j})$ viz Obr. 1.1
2. Pro řádek $i \in N$ je od $q_{i,j}$, kde $j \in N$ odečtena hodnota mr_i viz Obr. 1.2
3. Pro sloupec $j \in N$ je nalezena minimální hodnota sloupce $ms_j = min(q_{i,j})$ viz Obr. 1.3
4. Pro sloupec $j \in N$ je od $q_{i,j}$, kde $i \in N$ odečtena hodnota ms_j viz Obr. 1.4
5. Najít minimální velikost množin $|R + S|$, kde R je množina řádků $R \subseteq N$ a S je množina sloupců $S \subseteq N$. Množiny $R + S$ dohromady obsahují $\{\forall q_{i,j} \in Q : q_{i,j} = 0\}$ viz Obr. 1.5
6. Pokud $|R + S| = n$, potom je nalezeným řešením $\{\forall q_{i,j} \in Q : q_{i,j} = 0\}$, kde $q_{i,j}$ je optimální přiřazení úkolu u_i pro pracovníka p_j. Pokud $|R + S| \neq n$ algoritmus pokračuje krokem 7.
7. Nalézt mt odpovídající nejmenšímu $q_{i,j} \in Q$ takovému, že $i \notin R$ a zároveň $j \notin S$ viz Obr. 1.6

8. Odečíst mt od všech $q_{i,j} \in Q$, kde $i \notin R$ a zároveň $j \notin S$ viz Obr. 1.7
9. Přičíst mt ke všem $q_{i,j} \in Q$, kde $i \in R$ a zároveň $j \notin S$ viz Obr. 1.8
10. Opakovat algoritmus od kroku 5 viz Obr. 1.9

Konečný stav algoritmu po další iteraci je možné vidět na Obr. 1.10. Velikost množin $|R + S| = n$, což znamená, že máme optimální přiřazení. V případě zobrazeném na obrázku je optimální přiřadit dvojice (u_1, p_1), (u_1, p_3), (u_2, p_2), (u_2, p_4), (u_3, p_3), (u_3, p_4), (u_4, p_1), (u_4, p_2), (u_4, p_3).

Přiřazení může být tedy realizováno více způsoby, protože jak lze vidět například u_1 může být přiřazen k p_1 nebo p_3. V takovém případě je nejlepší začít přiřazovat pracovníky k úkolům, kterým může být přiřazeno nejméně pracovníků. Výsledkem tohoto příkladného přiřazení tedy mohou být dvojice: (u_1, p_1), (u_2, p_2), (u_3, p_4), (u_4, p_3).

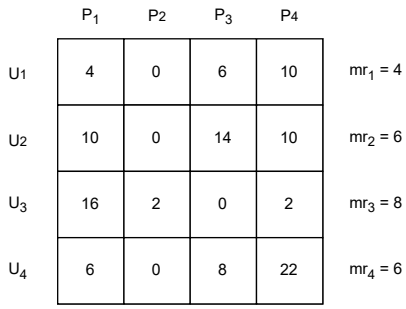
Celková minimalizovaná kvalifikace tedy bude $q_{1,1} + q_{2,2} + q_{3,4} + q_{4,3} = 8 + 6 + 10 + 14 = \mathbf{38}$. Znamená to, že bylo dosaženo stejně dobrého řešení jako u lineárního programování v podkapitole 1.1.1.

1.2 Řešení pomocí strojového učení

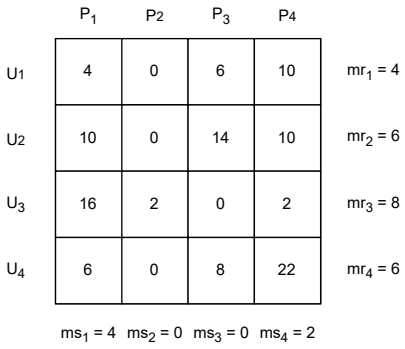
Pro řešení problému přiřazení je možné použít také strojové učení. Při takové metodě řešení problému není naprogramován algoritmus jak dosáhnout výsledku, ale stroj se algoritmus učí. Čím více unikátních dat je k dispozici tím je zpravidla lepší dosažený výsledek. Učení může probíhat na vstupech u kterých známe požadovaný výstup tzv. učení s učitelem. Příkladem může být úloha rozpoznání zvířete na obrázku, kde jako data pro trénování (učení) používáme obrázky, ke kterým je napsáno, jaké zvíře se na nich vyskytuje. Naproti tomu učení bez učitele probíhá na vstupech, u kterých výstup není znám. Může jít například o algoritmus učení pro co nejvyšší možné skóre Atari her. [14]

1.2.1 Neuronové sítě

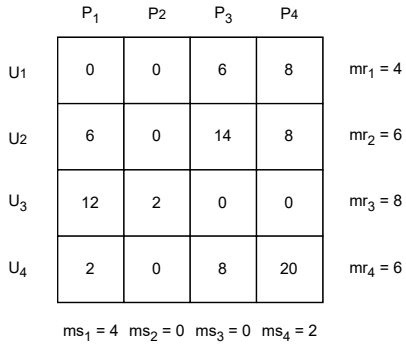
Neuronové sítě spadají do kategorie strojového učení s učitelem. K jejich učení jsou tedy potřeba správné výstupy při daných vstupech. Pro problém přiřazení to znamená, že pro danou kvalifikační matici musíme znát optimální přiřazení úkolů k pracovníkům. Optimální přiřazení lze získat dříve popsanými algoritmickými metodami buď Maďarskou metodou 1.1.2 nebo lineárním programováním 1.1.1. Po časově náročném natrénování, dokáže neuronová síť aproximovat původní algoritmickou metodu. Výhoda je ovšem ta, že predikce již natrénované neuronové sítě má asympto-



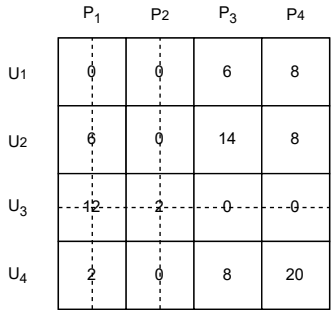
Obr. 1.2: Odečtení mr_i od každého řádku.



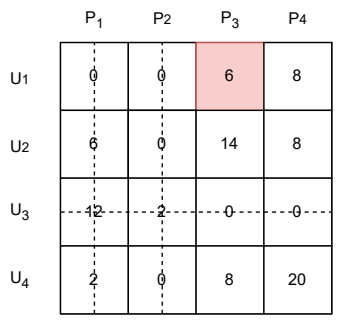
Obr. 1.3: Nalezení minimální hodnoty každého sloupce.



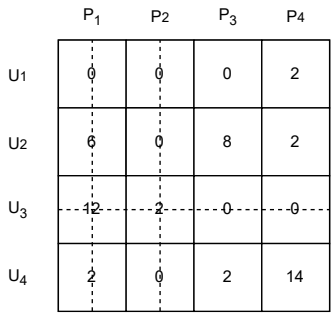
Obr. 1.4: Odečtení ms_j od každého sloupce.



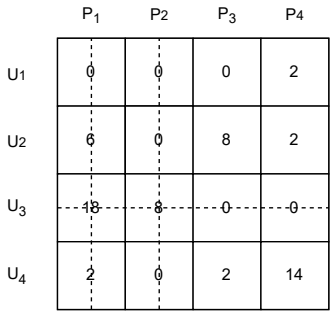
Obr. 1.5: Výběr množin $R = \{3\}$ a $S = \{1, 2\}$.



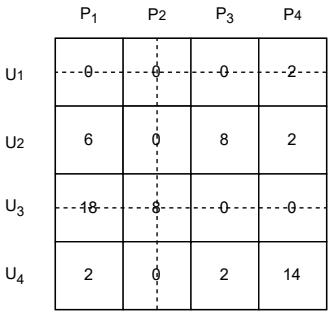
Obr. 1.6: Nalezeno nepřeškrtnuté minimum $mt = 6$.



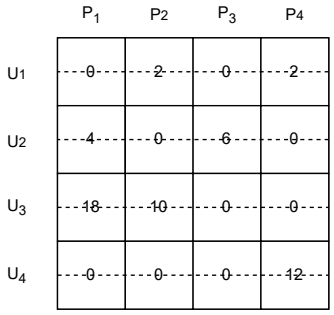
Obr. 1.7: Odečteno mt od $q_{1,3}, q_{1,4}, q_{2,3}, q_{2,4}, q_{4,3}, q_{4,4}$.



Obr. 1.8: Připočteno mt ke $q_{3,1}$ a $q_{3,2}$.



Obr. 1.9: Výběr množin $R = \{1, 3\}$ a $S = \{2\}$.



Obr. 1.10: Konečný stav přiřazení.

tickou složitost:

$$\phi(\sum_{l=1}^{f} D \cdot W \cdot H \cdot N),$$

kde l udává index vrstvy, D dimenzi vstupních neuronů, W šířku vstupu, H výšku vstupu, N počet výstupních neuronů. [15] Díky tomu, že predikce spočívá zejména v násobení matic je výhodná vlastnost ještě znásobena možností akcelerovat výpočetně nejnáročnější operaci na grafické kartě (GPU). Pokud je síť natrénována, je možné snížit složitost například u Maďarského algoritmu $\phi(n^3)$ na uvedenou složitost predikce neuronové sítě.

Konkrétní návrh pro účely aproximace Maďarské metody je možné nalézt v publikaci autorů LEE, M. et al. [16]. Jako první krok je uváděno vytvoření p neuronových sítí, kde p je počet pracovníků. Vstupem každé sítě je kvalifikační matice a výstupem je vektor znázorňující optimální přiřazení. Navrhované jsou dva modely. Prvním je dopředná neuronová síť se třemi skrytými vrstvami obsahujícími 32, 64 a 256 neuronů, respektive. Tato síť používá aktivační funkci sigmoid a křížovou entropii jako ztrátovou funkci. Druhým navrhovaným modelem je neuronová síť se dvěma konvolučními vrstvami obsahujícími dva a čtyři konvoluční filtry. Jednou skrytou vrstvou obsahující 256 neuronů, aktivační funkce pro tento model je tzv. REctified Linear Unit(ReLu). Oba modely využívají L2 regularizace.[1]

Pro získání finální aproximace maďarského algoritmu jsou výsledky z jednotlivých neuronových sítí spojeny. V případě kolize (úkol je přiřazen dvěma pracovníkům) je úkol přiřazen pracovníkovi s vyšší kvalifikací. Autoři testovali řešení na problémech s kvalifikačními maticemi $n \cdot n$ pro $n = 4, 8, 16$.

1.2.2 Q-Učení

Na rozdíl od neuronových sítí, Q-Učení je algoritmus učení spadající do kategorie bez učitele [17]. Algoritmus optimalizuje danou ziskovou funkci. Učí se na základě prozkoumávání možných akcí a hodnot ziskové funkce, kterou jednotlivé akce přinesou. Zároveň bere také v potaz možné budoucí hodnoty ziskové funkce získané akcí. Tyto veličiny ukládá do takzvané Q-tabulky. Sloupce tabulky reprezentují stavy a řádky tabulky reprezentují akce. Hodnota buňky je počítána iterativně dle vzorce:

$$Q^{nové}(s_t, a_t) = Q^{staré}(s_t, a_t) + \alpha(r_t + \gamma max_a Q^{staré}(s_{t+1}, a) - Q^{staré}(s_t, a_t)),$$

kde $Q^{nové}(s_t, a_t)$ je nová hodnota Q-tabulky pro stav s_t a akci a_t, $Q^{staré}(s_t, a_t)$ je stará hodnota Q-tabulky pro stav s_t a akci a_t, α je volený parametr $\alpha \in (0, 1>$ rychlosti učení, r_t představuje odměnu v čase t, γ je volený parametr $\gamma \in (0, 1>$ udávající váhu budoucích odměn plynoucích z dané akce. [18].

[1] Problematika neuronových sítí a jejich komponent je detailně rozebrána v kapitole 3.

V rámci učení je v každém stavu dle definované šance zvolena buď akce s největší Q hodnotou pro daný stav a nebo je zvolena náhodná akce. Zvolení vhodné šance závisí vždy na konkrétním problému a souvisí s dilematem průzkumu proti vykořisťování. Při zvolení vysoké šance pro náhodnou akci bude prozkoumávaný stavový prostor moc velký a algoritmus se nebude příliš učit. Při zvolení příliš malé šance na náhodnou akci algoritmus uvízne v tzv. lokálním minimu. Doporučuje se vzrůstajícím počtem iterací snižovat šanci na náhodnou akci. Obecný cíl algoritmu je tedy naučit se co největší a nejpřesnější Q-tabulku. Podle ní lze postupně iterovat k nejlepšímu nalezenému řešení tak, že bude pro každý stav vybrána akce s největší Q hodnotou.

Q-učení lze také aplikovat na problém přiřazení. Zisková funkce je definována jako metrika měřící kvalitu přiřazení viz podkapitola 4.2. Akce je chápána jako přiřazení pracovníka k úkolu. Stav je dán množinou přiřazených úkolů k pracovníkům. Akcí je tedy vždy tolik, kolik je pracovníků. Stavů je tolik, kolik je permutací přiřazení pracovníků k úkolům. Pro 10 úkolů a 10 pracovníků je $10! = 3628800$ stavů. Počet stavů tedy roste faktoriálně s počtem úkolů a pracovníků. To je velká nevýhoda tohoto algoritmu, kdy stavový prostor je pro větší počty úkolů a pracovníků se současnými výpočetními prostředky neprozkoumatelný a Q-tabulka může narůst do obrovských paměťových rozměrů.

1.2.3 Hluboké Q-Učení

Hluboké Q-učení je vylepšením algoritmu Q-učení. Problematická Q-tabulka je odstraněna, s ní také největší problém Q-učení. Na její místo je dosazena neuronová síť, která Q-tabulku aproximuje. Neuronová síť má počet vstupních neuronů roven počtu stavů. Počet neuronů výstupní vrstvy je roven počtu akcí. Výstupem neuronu reprezentujícího danou akci je pravděpodobnost, že daná akce má největší Q-hodnotu.

Pro problém přiřazení je možné hluboké Q-učení aplikovat stejně jako Q-učení. Již přiřazené úkoly k testerům jsou přivedeny na vstup neuronové sítě. Po natrénování neuronové sítě je výstupem pravděpodobnost, kterého testera bude nejlepší k úkolu přiřadit v zájmu maximalizace dané metriky.

2 Zpětná vazba

Pro sestavení problému přiřazení je třeba definovat kvalifikační matici, vyobrazenou například na Obr. 1.1. Tato kvalifikační matice by měla být v ideálním případě sestavena z reálných časů, které pracovníkům jednotlivé úkoly zaberou. Tohoto ideálu ale není možné dosáhnout v případě, pokud dopředu nevíme přesný čas, jenž bude pracovník potřebovat k dokončení úkolu.

Penetrační testování je právě tento případ. Dopředu není přesně známo kolik času zabere penetračnímu testerovi vykonat daný úkol. Proto pro prvotní rozřazení úkolů penetračního testování k penetračním testerů bude kvalifikační matice sestavena na základě schopností, které uvedou samotní testeři v dotazníku. Aby tyto údaje bylo možné ověřit a kvalifikační matici sestavit na základě schopností odpovídajících realitě, je nutné implementovat algoritmus zpětné vazby. V rámci zpětné vazby je potřeba odvodit kolik času tester na daném úkolu strávil. Tento údaj je poté využit pro zlepšení budoucích přiřazení. Návrhem algoritmu zpětné vazby se zabývají v [19, 20, 21, 22, 23].

V [19] se autoři zaměřují na vylepšení crowdsourcingových platforem tím, že efektivně zobrazí pro individuální pracovníky poptávající práci nejrelevantnější úkoly na předních místech seznamu inzerátů úkolů. Pro vstup do algoritmu využívá hodnoty jako míra přijetí odevzdaných úkolů pracovníka, preferovaný typ úkolů pracovníka, preferovaná odměna, preferovaná doba trvání úkolu. Výstupem je pořadí úkolu v seznamu úkolů.

Článek [20] se zabývá efektivním přiřazením pracovníků k úkolům nacházejícím se v určité lokaci. Může se jednat například o platformu pro roznášku jídla. Jádrem práce je výhodné spojení problému přiřazení a problému plánování. Ovšem vzhledem k zahrnutým lokalizačním údajům není tento mechanismus použitelný pro přiřazení penetračních testerů k úkolům.

Zdroj [21] se snaží nalézt stabilní přiřazení pro účely davového mobilního snímání. Zde se jedná spíše o pohled na přiřazení jehož řešením je stabilita nikoliv optimalizace přiřazení. Jádrem rešení problému je tedy již popisovaný Gale-Shapley algoritmus.

Zpětná vazba, kdy jsou na základě ní přiřazovány úkoly pracovníkům je rozebírána v [22]. Zde se ovšem článek zaměřuje na úkoly, které jsou řešeny ve více pracovnících tzv. kolaborativní crowdsourcing. Zpětná vazba je zde získávána od zadavatele úkolu v podobě dotazníku.

Strategie přiřazení pro co největší zlepšení schopností pracovníka v crowdsourcingu byla popsána v [23]. Bylo zde nahlíženo na problém přiřazení jako na optimalizační problém batohu. Kdy za určitý čas pracovníka je snaha dosáhnout co největší hodnoty zlepšení jeho schopností. Na samotnou kvalitu přiřazení bylo také

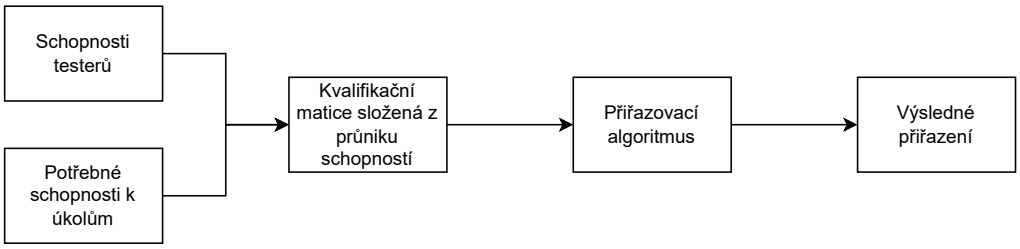
pohlíženo jako na problém batohu v [24], kde je cílem dosáhnout co nejvíce odvedené práce za daný čas. Problém byl modelován na grafu, na základě kterého byl navržen i algoritmus řešení.

Byly nalezeny také zdroje [25, 26] s podobnými klíčovými slovy. Oba zdroje se ovšem zabývají produktivitou a efektivitou přiřazení z pohledu managementu a ekonomie.

Nebyla tedy nalezena žádná literatura, která by problém přiřazení pracovníků k úkolům, kde není znám čas plnění úkolu, řešila. Uvedená literatura se vždy zaobírala pouze podobnými problémy, které kvůli svým odlišnostem nelze jako celek aplikovat pro tento problém přiřazení. Z literatury byla pouze převzata inspirace v podobě vstupních parametrů do algoritmů přiřazení.

2.1 Návrh algoritmu

Byl navržen dvou-krokový algoritmus přiřazení. První krok spočívá v sestavení kvalifikační matice z průniku schopností penetračních testerů a ze schopností požadovaných daným úkolem. Průnik schopností je spočítán dle popsané metriky v podkapitole 4.2.1. Kvalifikační matice je přivedena na vstup přiřazovacího algoritmu. Ten může být realizován buď Maďarskou metodou nebo lineárním programováním, viz podkapitola 1.1. Výstupem algoritmu je již optimální přiřazení vzhledem k zadaným schopnostem testerů. Schéma je zobrazeno na Obr. 2.1. První krok je vykonán

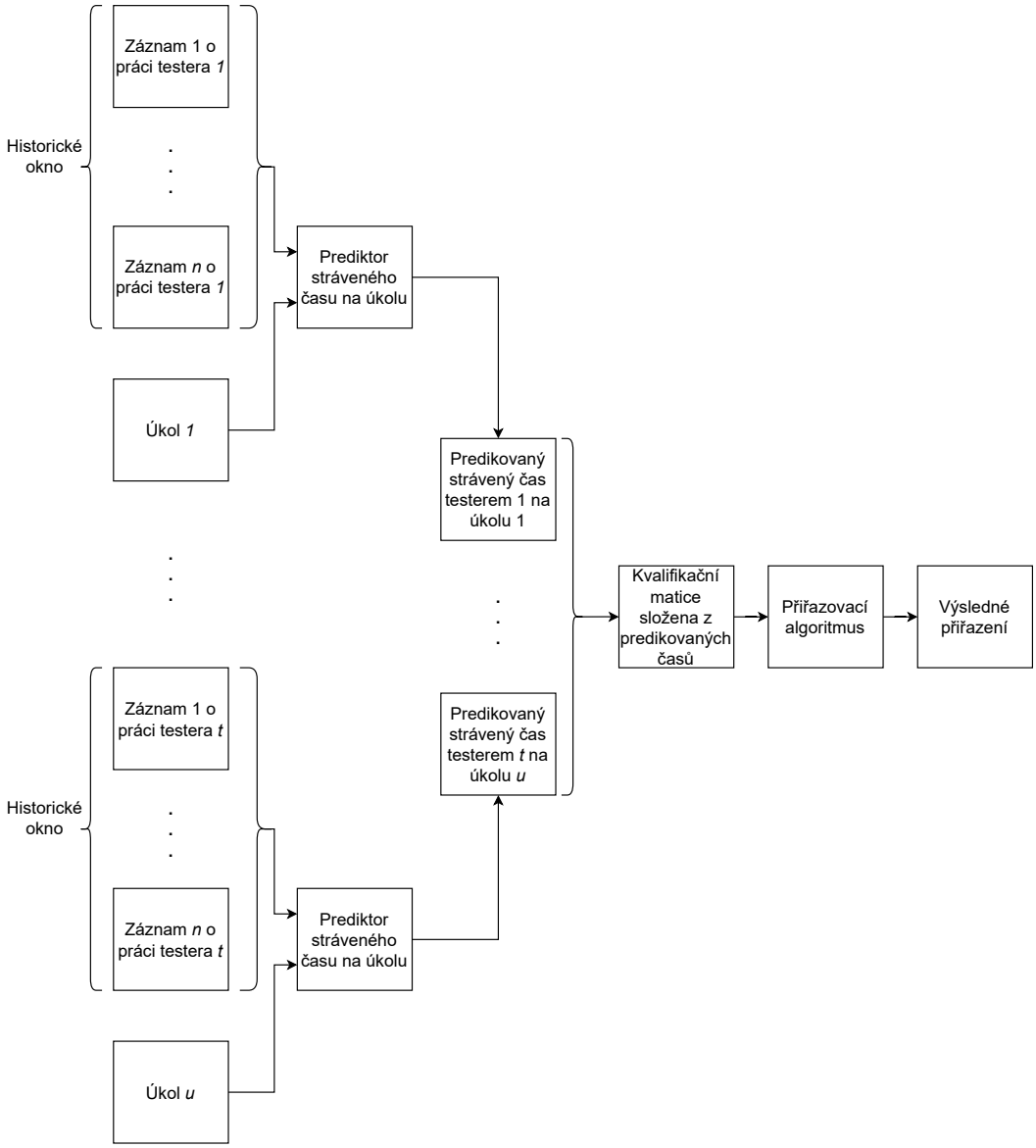


Obr. 2.1: Prvotní přiřazení úkolů k penetračním testerům.

ve fázi, kdy nejsou ještě dostupné údaje o reálných časech, které byly potřeba pro vykonání zadaných úkolů přiřazenými testery. Výsledné přiřazení je sice globálně optimální, ovšem pouze vzhledem k schopnostem testerů, které testeři sami zadali v dotazníku. Přiřazení bude tedy tak přesné, jak přesně testeři dokázali odhadnout svoje schopnosti.

Druhý krok může být vykonán jakmile je dostupných n historických záznamů o splněném úkolu pro každého testera, kde n je počet záznamů práce daného testera, které mají být v rámci algoritmu uvažovány tzv. historické okno. Pro každý

úkol u určený k přiřazení testerům je nejprve pro každého testera vyhledáno n nejpodobnějších úkolů k úkolu u, které v minulosti tester řešil. Tato množina úkolů s úkolem u poté slouží jako vstup pro prediktor času, který by tester měl strávit na úkolu. Z predikovaných časů každého úkolu pro všechny testery je sestrojena kvalifikační matice. Pro t testerů a u úkolů je sestrojena kvalifikační matice o $t \cdot u$ členech. Tato kvalifikační matice je nakonec předána přiřazovacímu algoritmu, jehož výsledkem je finální přiřazení úkolů k testerům. Proces je znázorněn na Obr. 2.2. Přiřazení je v tomto případě optimální pro zadanou kvalifikační matici. Což znamená, že přiřazení bude tak dobré, jak budou přesné predikce času, který by měl tester na úkolu strávit. Princip prediktoru je založen na neuronové síti. Predikce tedy jsou tak kvalitní, jak dobře je natrénována neuronová síť realizující predikce.



Obr. 2.2: Přiřazení úkolů k penetračním testerům na základě zpětné vazby.

3 Neuronová síť

Neuronová síť (dále jen NN) je algoritmus spadající do kategorie strojového učení. Učení probíhá na vstupech, u kterých je znám požadovaný výstup, takzvané učení s učitelem. Na vstup NN jsou přivedena data, predikce NN je porovnána s požadovaným výstupem. Chyba mezi požadovaným výstupem a predikcí je použita pro zlepšení budoucích predikcí díky algoritmu zpětného šíření chyby.

Architektura základní dopředné NN je realizována vrstvami takzvaných neuronů. Neurony v jednotlivých vrstvách jsou mezi sebou propojeny. Tyto vrstvy se dělí na vstupní, skryté a výstupní. Historicky měly neurony napodobovat proces zpracování informací v lidském mozku. V počítačové vědě je jádrem neuronu aktivační funkce, která společně s jeho váhami určuje výstupní hodnotu. Vstup neuronu je vážený součet výstupních hodnot neuronů předchozí vrstvy. Pro vstupní vrstvu platí, že vstupní hodnoty NN přímo tvoří hodnotu neuronu. [27] Výstup neuronu j je možné spočítat dle vzorce [28]:

$$o_j = f(a_j),$$

kde f je aktivační funkce a a_j lze vyjádřit jako:

$$a_j = \sum_i x_i \cdot w_i,$$

kde x_i je výstup neuronu i předchozí vrstvy a w_i je váha pro neuron i. Váhy jsou z počátku nejčastěji inicializovány na náhodné hodnoty. Postupně procesem tréninku sítě se váhy upravují. Trénink spočívá v úpravě vah díky algoritmu zpětného šíření chyby (error backpropagation).

3.1 Ztrátová funkce

Úzce s algoritmem zpětného šíření chyb souvisí ztrátová funkce. Její výstupní hodnota je totiž algoritmu předána. Ztrátová funkce udává rozdíl mezi predikovanou a požadovanou hodnotou výstupu NN. Ztrátových funkcí je několik a vždy záleží na konkrétním případu, která funkce bude vhodná. Dle řešeného problému se funkce rozdělují, zda-li jsou využity za účelem klasifikace nebo regrese. [29]

Pro řešení problému klasifikace se běžně používá binární křížová entropie, jež lze vypočítat jako [30]:

$$-\frac{1}{m}\sum_{i=1}^{m} y_i \cdot log(\hat{y}_i),$$

kde m je počet výstupních neuronů, y_i je požadovaná výstupní hodnota neuronu i a $\hat{y}_i$ je výstupní (predikovaná) hodnota neuronu i. Další možnou funkcí pro klasifikační problémy může být závěsová ztráta (hinge loss), definována jako [30]:

$$max(0, 1 - \sum_{i=1}^{m} \frac{1}{2} - y_i \cdot \hat{y}_i),$$

značení proměnných je stejné jako v předchozím vzorci, pouze y_i je zde zakódováno v kódu 1 z n.

Pro regresní problémy jsou doporučovány a běžně používány jiné ztrátové funkce. Často používanou je střední kvadratická chyba, vypočtena dle vzorce [30]:

$$\frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2,$$

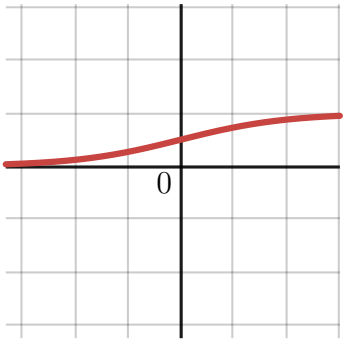
proměnné opět mají stejný význam jako u předchozích vzorců. Jelikož se jedná o regresi y_i není nijak kódováno. Druhou velice často používanou funkcí je střední absolutní chyba, spočtena jako [30]:

$$\frac{1}{m} \sum_{i=1}^{m} |\hat{y}_i - y_i|,$$

přičemž význam promněných je stejný jako u předchozích ztrátových funkcí.

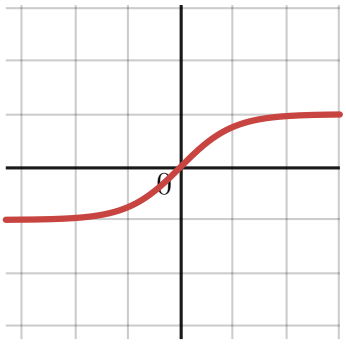
3.2 Aktivační funkce

Aktivační funkce jsou rozhodovací jednotkou každého neuronu. Každá aktivační funkce má jiné parametry, s čímž přichází i různé výhody a nevýhody. Nejznámější a historicky nejpoužívanější aktivační funkcí je sigmoida zobrazena na Obr. 3.1. Je



Obr. 3.1: Sigmoidní aktivační funkce.

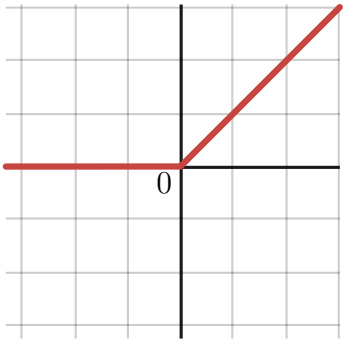
definována jako $\frac{1}{1+e^x}$. Obor hodnot funkce je z intervalu $(0, 1)$. Podobnou aktivační funkcí je hyperbolický tangens (tanh) spočten jako $\frac{sinh(x)}{cosh(x)}$, zobrazen na Obr. 3.2. Liší se zejména oborem hodnot, který je definován v intervalu $(-1, 1)$. Obě akti-



Obr. 3.2: Hyperbolický tangens aktivační funkce.

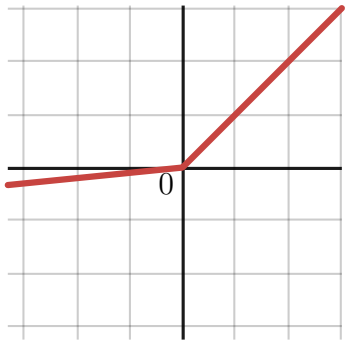
vační funkce jsou vhodné pro NN s menším počtem skrytých vrstev, kde mohou produkovat velice dobré výsledky. V sítích, které jsou hluboké, nemusí tyto aktivační funkce pracovat optimálně. Tyto funkce trpí známým problémem mizejícího gradientu, který brání síti ve zlepšování při tréninku. Problém vzniká tak, že sigmoida a hyperbolický tangens mají malé gradienty pro velké kladné či záporné hodnoty. Během algoritmu zpětného šíření chyb se malé gradienty chybové funkce skrze vrstvy NN násobí a tím se exponenciálně zmenšují.

Pro sítě s větším počtem skrytých vrstev je doporučeno používat novější aktivační funkci známou jako ReLU (zkratka z anglického REctified Linear Unit) a její variace. Základní relu je spočtena jako $max(0, x)$. Obor hodnot je z intervalu $(0, x)$. Graf funkce je vykreslen na Obr. 3.3. Tato funkce má velkou výhodu, že problémem



Obr. 3.3: ReLU aktivační funkce.

mizejícího gradientu netrpí a v některých případech síť konverguje k řešení rychleji než s klasickou sigmoidou či tanh. ReLU ovšem trpí problémem zvaným umírající ReLU. Stav vznikne tak, že neuron učením dostane pracovní bod na velkou zápornou hodnotu, což způsobí, že jeho výstup bude vždy 0. Jako snaha tento problém vyřešit vznikla aktivační funkce zvaná děravá ReLU definovaná jako x pro $x \geq 0$ a jako $a \cdot x$ pro $x < 0$, kde a je volitený parametr například 0,01. Funkce je zobrazena na Obr. 3.4. Bylo ovšem dokázáno, že rozdíl funkce oproti standardní ReLU je



Obr. 3.4: Děravá ReLU aktivační funkce, $a = 0,1$.

zanedbatelný. [31]

3.3 Prevence nadměrného přizpůsobení

Čím více vrstev a neuronů síť má, tím komplexnější vzorce se má potenciál naučit. Z toho plyne problém zvaný nadměrné přizpůsobení neboli také overfitting. Tato situace nastává ve chvílí, kdy si NN vytvoří příliš komplexní model funkce, kterou má aproximovat. Učením síť potom zlepšuje svoje predikce na trénovací množině, ale zhoršuje predikce obecně, například na testovací množině. Stává se tak postupně horším aproximátorem dané funkce. Tento problém řeší zejména techniky regularizace a redukce počtu neuronů, blíže popsané dále v textu.

3.3.1 Regularizace

Hlavní myšlenkou regularizace L2 nebo také váhového úbytku je penalizovat neurony za nastavování vysokých vah. Toho lze docílit tak, že ke ztrátové funkci je přičten regularizační výraz. Ztrátová funkce s regularizací lze potom zapsat jako:

$$L_\lambda(w) = L(w) + \lambda||w||_2^2,$$

kde w je vektor vah, $L(w)$ je hodnota ztrátové funkce pro váhy w a λ je regularizační parametr. Čím vyšší regularizační parametr je nastaven, tím více bude model přinucen váhy jednotlivých neuronů přibližovat nule. To způsobí zjednodušení modelu a může předejít problému nadměrného přizpůsobení. Běžné hodnoty λ jsou v intervalu $<0.1, 0.001>$. Pro optimální výkon sítě je nutné s těmito hodnotami experimentovat (tzv. ladění hyperparametrů) a zjistit, jaká je nejvhodnější pro daný problém a architekturu NN. [32]

3.3.2 Redukce počtu neuronů

Další technikou k potlačení nadměrného přizpůsobení sítě je redukce počtu neuronů (dropout). To je realizováno vynecháním určitého procenta náhodně zvolených neuronů během každého tréninku sítě. Přičemž vynechané neuronu jsou vždy náhodně zvoleny znovu po každé tréninkové dávce. Efektivně to vede k tomu, že jeden neuron bude vystaven menšímu množství tréninkových dat a také predikce výsledného modelu bude průměrem zmenšených modelů s redukovanými neurony. To zabrání přílišné adaptaci na tréninková data. Technika je velice populární v oblasti strojového vidění, kde zajišťuje určitým způsobem i redundanci modelu. [33]

3.4 Normalizace

Normalizace je transformace dat, nejčastěji na stejné měřítko. V NN nalézá uplatnění, protože může zlepšit generalizaci a zrychlit dobu tréninku NN. Z toho důvodu se vstupní data NN často normalizují. [34]

Min-max normalizace převádí data na hodnoty od nuly do jedné. Výslednou hodnotu lze vyjádřit následovně:

$$x' = \frac{x - n}{r},$$

kde n je minimální hodnota z datové množiny X a:

$$r = m - n,$$

kde m je maximální hodnota z datové množiny X, přičemž $x \in X$.

Dalším typem normalizace je Z-normalizace. Ta převádí data tak, aby hodnoty odpovídaly standardizovanému normálnímu rozdělení. Výpočet je proveden jako:

$$x' = \frac{x - \mu}{\sigma},$$

kde μ je průměr hodnot X a σ je směrodatná odchylka hodnot X.

3.5 Konvoluční vrstva

Kromě již popisovaných vrstev, které jsou plně propojeny, může mít NN také speciální vrstvy provádějící operaci diskrétní konvoluce (dále jen konvoluce). Konvoluční vrstvy se umisťují před plně propojené vrstvy. Účelem této vrstvy je provádět konvoluci mezi vstupem a konvolučním filtrem. Jednodimenzionální konvoluce je definována jako:

$$(f * h)[i] = \sum_{m \in M} f[m] \cdot h[i - m],$$

-1	-1	-1
-1	8	-1
-1	-1	-1

Obr. 3.5: Konvoluční filtr velikost 3x3 pro detekci hran.

dvou dimenzionální konvoluce je definována jako:

$$(f * h)[i, j] = \sum_{m \in M} \sum_{n \in N} f[m, n] \cdot h[i - m, j - n],$$

kde $*$ je operace konvoluce, i a j reprezentují řádek a sloupec výstupní matice respektive, m a n jsou souřadnice vstupní matice f a filtru h. Intuitivně si lze operaci představit jako posouvání filtru nad vstupní maticí po jednotlivých sloupcích a řádcích.

Filtry mohou mít libovolnou velikost, například pro konvoluci dvou-dimenzionálního vstupu může být rozměr 3x3. Každá konvoluční vrstva může mít libovolný počet filtrů. Filtr je v takovém případě matice 3x3, jejíž členy jsou naučeny během procesu učení NN. Filtry mohou například detekovat hrany, křivky nebo také rozostřovat či zaostřovat obraz. Příklad filtru, který umí detekovat hrany je na Obr. 3.5.

Nejčastější použití konvolučních vrstev je tedy v oblasti počítačového vidění, nicméně se prosadily například i v oblasti zpracování přirozeného jazyka. [35]

4 Popis programu

Implementace byla provedena v programovacím jazyce *Python 3*, konkrétně testováno ve verzi 3.10.11 na operačním systému *Microsoft Windows 11*. První část se zabývá vygenerováním datové sady (dále jen dataset), protože nebyl nalezen žádný veřejný dataset obsahující pracovníky, jejich schopnosti a úkoly vyžadující určité schopnosti. Další část se věnuje metrikám, které byli vymyšleny a implementovány pro vyhodnocení kvality přiřazení penetračních testerů k daným úkolům. Po metrikách následuje část popisující několik otestovaných algoritmů pro řešení problému přiřazení. Následující část se zaměřuje na generování datasetu obsahujícího záznamy vykonané práce. Poslední kapitola řeší problém přiřazení na základě těchto záznamů.

4.1 Generování datasetu

Dataset penetračních testerů a úkolů je generován ve formátu JSON (JavaScript Object Notation). Výsledkem je jeden JSON soubor s vlastnostmi `workers` typu `array` a `tasks` typu `array`.

Pro vytvoření datasetu slouží soubor `create_json_dataset.py`. Spustitelný soubor přijímá argumenty, z nichž některé jsou povinné:

- `-w` povinný argument udávající požadovaný počet penetračních testerů v datasetu,
- `-t` povinný argument udávající požadovaný počet úkolů v datasetu,
- `-tt` povinný argument udávající požadovaný celkový počet typů schopností,
- `-o` volitelný argument s výchozí hodnotou `dataset.json` udávající název výstupního JSON souboru.

4.1.1 Penetrační testeři

Penetrační testeři se nacházejí v JSON jako `workers` a mají `id` a `skills`. `id` je unikátní číslo penetračního testera, které začíná od jedné a pro každého dalšího testera je zvýšeno o jedna. Vlastnost `skills` udává, jaké schopnosti tester má.

Tyto schopnosti jsou uloženy jako páry `typ schopnosti: úroveň schopnosti`, jejichž počet je vybrán náhodně. Typ schopnosti je udáván libovolným číslem a v praxi může být využit jako identifikátor různých typů schopností a znalostí penetračního testování jako například *testování webových aplikací* nebo *testování mobilních aplikací*. Úroveň schopnosti je udávána jako desetinné číslo od 0,1 do 1; které je během generování v tomto rozsahu vybráno náhodně a následně zaokrouhleno na jedno desetinné místo. Příklad objektu testera je ve výpise 4.1.

Výpis 4.1: Příklad objektu `worker`

```
{
    "id": 1,
    "skills": {
        "3": 0.1,
        "2": 0.6,
        "1": 0.7
    }
}
```

4.1.2 Úkoly

Úkoly nacházející se v JSON jako `tasks` mají vlastnosti `id` a `required_skills`. Vlastnost `id` je stejně jako u penetračních testerů unikátní identifikátor. Vlastnost `required_skills` je typ `array` uvnitř jsou čísla vyjadřující požadovaný typ schopnosti. Požadované schopnosti jsou vybrány náhodně v náhodném počtu od 1 do celkového počtu typů schopností. Úkol tedy může například vyžadovat schopnost *testování webových aplikací* a *testování bezpečnosti síťových zařízení* označené jako kategorie s číslem 3 a 6. Příkladný úkol ve formátu JSON je zobrazen ve výpise 4.2.

Výpis 4.2: Příklad objektu `task`

```
{
    "id": 1,
    "required_skills": [
        3,
        1,
        2
    ]
}
```

4.2 Metriky

Bylo navrženo a implementováno několik metrik měřících kvalitu přiřazení penetračních testerů k úkolům. Tato kvalita je jednak měřena ve smyslu jak dobře je skupina testerů schopna řešit přiřazené úkoly, na základě uvedených vlastních schopností. Dalším měřením kvality přiřazení je měření rovnoměrnosti přiřazení tj. testeři co

jsou nejvíce schopní nedostanou nejvíce úkolů a naopak testeři s nejnižšími schopnostmi dostanou dostatečný počet úkolů.

Metriky je možné použít jako porovnání jednotlivých algoritmů přiřazení. Dalším možným využitím metrik je jejich použití jako ztrátové/ziskové funkce pro zpětnovazební učení.

4.2.1 Schopnostní metriky

Implementovaná a navržená metrika *společných schopností* měří průnik schopností testera a úkolu. Opakem je metrika *chybějících schopností*. Metrika *chybějících/nadbytečných schopností* měří velikost chybějících a nadbytečných schopností testera plnit úkoly.

Společné schopnosti

Cílem metriky je spočítat průměrnou schopnost testerů plnit přiřazené úkoly. Výpočet je implementován jako:

$$\frac{1}{|P|}\sum_{p\in P}\frac{1}{|D_p|}\sum_{d\in D_p} s_{d,p}\cdot n_{d,p},$$

kde P je množina přiřazení, D_p je množina typů schopností úkolu v přiřazení p, $s_{d,p}$ je schopnost typu d testera v přiřazení p, $n_{d,p}$ je požadovaná schopnost typu d úkolu v přiřazení p. Pokud úkol v přiřazení p vyžaduje schopnost typu d potom $n_{d,p} = 1$ jinak $n_{d,p} = 0$.

Výstupem metriky je desetinné číslo od 0 do 1. Nula značí, že žádný tester nemá přiřazen úkol, ke kterému by měl alespoň nějaké schopnosti. Jedna značí, že všichni testeři mají přiřazené úkoly, které vyžadují typy schopností, které všichni přiřazení testeři mají na 100%. Na reálném datasetu je velice nepravděpodobné, že by bylo možné této hodnoty dosáhnout i pokud bychom znali globálně optimální přiřazení.

Chybějící schopnosti

Cílem metriky je spočítat chybějící schopnost testerů plnit přiřazené úkoly. Výpočet je implementován jako:

$$\frac{1}{|P|}\sum_{p\in P}\frac{1}{|D_p|}\sum_{d\in D_p} max(s_{d,p} - n_{d,p}, 0),$$

definice proměnných je stejná jako pro společné schopnosti. Výstupem je také desetinné číslo od 0 do 1. Hodnoty mají přesně opačný význam jako u předchozí podkapitoly, tedy značí chybějící schopnosti jednotlivých testerů plnit zadané úkoly.

Chybějící/nadbytečné schopnosti

Cílem metriky je spočítat chybějící schopnost a překvalifikovanost testerů plnit přiřazené úkoly. Výpočet je implementován jako:

$$\frac{1}{|P|}\sum_{p\in P}\frac{1}{|D|}\sum_{d\in D}|s_{d,p}-n_{d,p}|,$$

definice proměnných je stejná jako u předchozích výpočtů schopností, pouze s tím rozdílem, že je použita množina všech typů schopností D z celého datasetu namísto pouze množiny typů schopností D_p požadovaných úkolem v přiřazení p.

4.2.2 Rovnoměrnostní metriky

Rovnoměrnostní metriky byly vyvinuty za účelem měření rovnoměrnosti přiřazení. Přiřazení je dokonale rovnoměrné, pokud má každý tester přiřazen stejný počet úkolů. Veličina je měřena metrikou *Nerovnoměrné pracovní rozložení*. Byla také implementována metrika *Chybně přiřazené úkoly* měřící úkoly přiřazené více testerům či nepřiřazené vůbec.

Nerovnoměrné pracovní rozložení

Cílem metriky je určit míru nadprůměrného vytížení testerů. Výpočet je implementován jako:

$$\frac{1}{n_{max}}\sum_{t\in T}|u_t-\frac{|U|}{|T|}|,$$

kde T je množina testerů v celém datasetu, u_t je počet úkolů testera t, U je množina úkolů v datasetu, n_{max} je maximální nerovnoměrné rozložení datasetu dle vzorce:

$$n_{max}=(|T|-1)\cdot\frac{|U|}{|T|}+|U|-\frac{|U|}{|T|}$$

Výstupem metriky je opět desetinné číslo od 0 do 1. Oba extrémy jsou v případě této metriky dosažitelné v každém datasetu. Hodnota 0 značí, že každý tester má přiřazený přesně průměrný počet úkolů připadajících na jednoho testera. Hodnota 1 značí, že všechny úkoly jsou přiřazeny jednomu testerovi.

Chybně přiřazené úkoly

Cílem metriky je zjistit zda-li nebyl nějaký úkol přiřazen vícekrát či nepřiřazen ani jednou. Výpočet je implementován jako:

$$||U|-|P||,$$

kde $|U|$ je velikost množiny úkolů a $|P|$ je velikost množiny přiřazení.

Výstupem metriky je číslo od 0 do ∞. 0 značí, že všechny úkoly byli přiřazeny právě jednou. Číslo větší než jedna značí počet úkolů, které byly přiřazeny více než jednou či ani jednou.

4.3 Prvotní přiřazení

Pro prvotní přiřazení (viz schéma na Obr. 2.1) úkolů k penetračním testerům bylo implementováno a na definovaných metrikách otestováno celkem devět algoritmů. Dva nejvhodnější byly vybrány pro aplikaci ve finálním rozhraní. Všechny algoritmy jsou přiložené v podobě zdrojových kódů v příloze práce.

4.3.1 Algoritmy

Algoritmy ve skupině *Přiřazení dle metriky* pouze procházejí seznam testerů s úkoly a hledají maximum stejnojmenné metriky. Další algoritmy *Lineární programování* a *Maďarská metoda* jsou založeny na optimalizačních algoritmech. Algoritmy *Q-učení* a *Hluboké Q-učení* jsou založeny na zpětnovazebním učení. Algoritmus *Náhodné přiřazení* je založen na náhodě a byl implementován čistě pro účely porovnání. Všechny algoritmy v této kapitole se spouští pro dataset s umístěním v `dataset/dataset_1000t`. Umístění je psáno vždy relativně ke kořenovému adresáři projektu.

Přiřazení dle metriky

Přiřazení dle metrik realizují metody `max_common_skills`, `min_insufficient_skills` a `min_insufficient_skills_or_overqualified` v souboru `assignment/algorithmic/bruteforce_classifier.py`. Vyhodnocení metod lze z kořenového adresáře spustit ze souboru `assignment/evaluation/matching_comparison.py`.

Metody fungují na stejném principu, jen pracují s odlišnou metrikou. Prochází kvalifikační tabulkou testerů k jednotlivým úkolům a během toho ukládají maximální hodnoty, na které narazí. Hodnoty jsou ukládány do Python slovníku, typ `dict`. Klíčem slovníku je identifikátor úkolu. Hodnotou slovníku je dvojice *identifikátor testera*, *hodnota metriky*. Pokaždé, když je během procházení datasetu nalezen pro úkol tester s lepší metrikou je hodnota slovníku aktualizována na hodnotu nově nalezenou.

Metody využívají metrik definovaných v podkapitole 4.2.1. Metoda `max_common_skills` využívá metriku společných schopností, metoda `min_insufficient_skills` využívá metriku chybějících schopností a metoda `min_insufficient_skills_or_overqualified` využívá metriku chybějících/nadbytečných schopností.

Lineární programování

Lineárního programování bylo implementováno ve dvou variantách. Obě varianty mají stejnou účelovou funkci, ale liší se ve způsobu jakým definují omezení.

První algoritmus je možné spustit ze souboru `assignment/algorithmic/linear_programming_unbound.py`. Prvně je načten dataset z JSON souboru do paměti. Následně je vytvořena kvalifikační matice. Matice je převedena do datového typu `list[int]`, který reprezentuje účelovou funkci. Hodnoty (koeficienty) reprezentující účelovou funkci jsou převráceny způsobem popsaným v podkapitole 1.1.2. Převrácení je nutné, protože funkce řešící lineární programování umí pracovat pouze s minimalizací účelové funkce. Dále jsou vytvořeny meze způsobem popsaným v podkapitole 1.1.1, jen s malým rozdílem. Jelikož se pracuje primárně s nevyváženým problémem přiřazení a meze by byly nastaveny tak, že k jednomu testerovi je možné přiřadit jen jeden úkol, tak nebudou všechny úkoly přiřazeny. Znamenalo by to, že program bude muset přiřazovat úkoly v několika iteracích. Algoritmus tedy problém řeší tak, že podmínku přiřazení jednoho úkolu k jednomu testerovi vynechává. Jeden tester tak může mít přiřazeno tímto algoritmem neomezený počet úkolů.

Druhý algoritmus se nachází v souboru `assignment/algorithmic/linear_programming.py`. Pracuje stejně jako první algoritmus jen řeší nevyvážený problém přiřazení jinak. Upravuje podmínku stanovení počtu úkolů jednomu testerovi. Pro každého testera je stanoven limit maxima přiřazený úkolů. Maximum přiřazených úkolů je spočteno jako $\frac{u}{t}$, kde u je počet úkolů a t je počet testerů. Tento způsob řešení je srovnatelný s maďarskou metodou.

Maďarská metoda

Implementována v souboru `assignment/algorithmic/hungarian_method.py`. Po spuštění nejprve načte dataset z JSON souboru do paměti. Poté je z datasetu vytvořená kvalifikační matice. Kvalifikační matice je převedena do třídy `DataFrame` z `pandas` balíčku pro Python. Nad celým objektem `DataFrame` je volána funkce `linear_sum_assignment` z balíčku `scipy.optimize`, která provede přiřazení. Metoda ale není schopna řešit nevyvážený problém přiřazení, a tak se musí postupovat v iteracích. Každá iterace přiřadí k n testerům pouze n úkolů. Pokud je tedy úkolů například $10n$, musí proběhnout 10 iterací. Hlavním výstupem po poslední iteraci je typ `list[Assignment]`, kde jsou uloženy optimální hodnoty přiřazení. V objektu `Assignment` je uložen vždy identifikátor testera `worker_id` a identifikátor úkolu `task_id`.

Q-učení

Lze spustit pomocí souboru `assignment/machine_learning/reinforcement_learning/q_learning.py`. Nejprve je načten soubor s datasetem do paměti. Poté je nastaveno několik parametrů pro Q-učení.

Parametr `epsilon` určuje, z kolika procent bude pro aktuální stav zvolena akce s nejvyšší Q-hodnotou vůči náhodné akci, hodnota je nastavena na 0,9. Parametr `discount_factor` udává váhu budoucích odměn, je nastaven na hodnotu 0,9. Parametr `learning_rate` udává míru učení a je nastaven na hodnotu 0,9. Parametr `episodes` značí kolik trénovacích epizod má program vykonat před vyhodnocením, je nastaven na hodnotu 10000. Dále je nastavena váha pro ziskovou funkci. Parametr `common_skills_weight` je nastaven na hodnotu 1, `workload_imbalance_weight` na hodnotu 0,25; což značí že ve finální ziskové funkci záleží čtyřikrát více na společných schopnostech než na rovnoměrném přiřazení testerů.

Následně je definováno několik metod realizujících různé části Q-učení. Je definována Q-tabulka v podobě stromové struktury. Stromová tabulka byla zvolena z důvodu optimalizace rychlosti a paměťové velikosti. Jednotlivé listy představují individuální stavy. Při změně stavu se vždy změní aktuální list, tak aby odpovídal aktuálnímu stavu. Listy obsahují Q-hodnoty pro jednotlivé pracovníky a pracovníka s nejvyšší Q-hodnotou pro stav daný listem. Přístup ke Q-hodnotám a pracovníkovi s nejvyšší Q-hodnotou probíhá v čase $O(1)$, protože se v obou případech jedná o pole. Výhoda oproti předchozímu přístupu (objekt `QTable`), kdy byly údaje uloženy v hashovací tabulce, spočívá v tom, že je ušetřen čas za hashování a díky použití pole z knihovny `numpy` je ušetřena i paměť. Knihovna totiž umožňuje ukládat desetinná čísla jako `float16` narozdíl od Pythonu, kde jsou ukládána (na 64 bitových procesorech) jako `float64`.

Před vstupem do smyčky pro epizody je inicializováno pole, do kterého se ukládají hodnoty ziskové funkce pro každou epizodu. V každé epizodě je nastaven ukazatel aktuálního listy na kořen stromu Q-tabulky, tím je nastaven stav nula. Také je inicializován objekt `DifferentialEvaluator` provádějící měření společných schopností a rovnoměrnosti přiřazení. Tento objekt byl vysoce optimalizován. Namísto měření společných schopností a rovnoměrnosti přiřazení pro celý stav vyhodnocuje ziskovou funkci na základě změny stavu. Výpočet společných schopností je realizován násobením matic, jež proces rapidně zrychluje. V objektu je udržována hashovací tabulka pro pracovníky, úkoly a počet úkolů přiřazených jednotlivým pracovníkům.

Další, vnořená smyčka, pokračuje po jednotlivých úkolech. Pro každý úkol vybere s pomocí metody `get_next_worker` pracovníka. Pracovník je vybrán buď dle nejvyšší Q-hodnoty pro daný úkol a nebo náhodně, závisle na parametru `epsilon`. Následně je aktualizován stav na vybraného pracovníka. Je spočtena hodnota ziskové

funkce metodou `get_reward_differential`. Následně je spočtena nová Q-hodnota pro vybraného pracovníka (akci) dle vzorce uvedeného v podkapitole 1.2.2. Nakonec je hodnota propsána do Q-tabulky a cyklus pokračuje.

Jakmile jsou cykly dokončeny je zobrazena maximální dosažená hodnota ziskové funkce. Všechny hodnoty ziskové funkce jsou také uloženy jako graf do složky `plots`.

Hluboké Q-učení

Spustitelný soubor pro přiřazení pomocí hlubokého Q-učení je v adresáři `assignment/machine_learning/reinforcement_learning/keras_dqn.py`. Spuštění vyžaduje knihovnu `keras-rl2`. Knihovna je kompatibilní pouze s verzí knihovny `tensorflow` 2.1.0.

Knihovna obsahuje třídu `Env`, která tvoří základní prostředí pro trénování algoritmů zpětnovazebního učení. Pro realizaci přiřazení pomocí hlubokého Q-učení je vytvořeno vlastní prostředí tréninku odvozující se z třídy `Env`. Prostředí obsahuje dataset, třídu vyhodnocují ziskovou funkci, stav tvořený seznamem přiřazených testerů a velikostí akčního prostoru, což je pro problém přiřazení počet pracovníků. Metoda obsluhující každý tah v tréninkovém prostředí aktualizuje stav o nově vykonanou akci, počítá hodnotu ziskové funkce plynoucí z akce a kontroluje zda-li už nejsou přiřazeny všechny úkoly. Prostředí obsahuje také metodu pro restart prostředí mezi epizodami.

Další krok je inicializace prostředí a nastavení počtu tréninkových epizod. Poté je sestaven model plně propojené dopředné neuronové sítě. Ten tvoří dvě skryté vrstvy o 500 a 250 neuronech respektive a výstupní vrstva s w neurony, kde w je počet pracovníků. Jako aktivační funkce je použita ReLU. Rychlost učení je nastavena na 0,001. Následně je sestaven agent hlubokého strojového učení, který využívá definovanou NN, sekvenční paměť a Boltzmannovu Q-proceduru.

Nakonec je agent sestaven a trénink spuštěn pro definovaný počet epizod. Jakmile trénink skončí je vytvořen graf ve složce `plots`, zobrazující vývoj ziskové funkce v jednotlivých epizodách.

Zrychlení pomocí neuronových sítí

Jak bylo popsáno v podkapitole 1.2.1, přiřazovací algoritmy je možné zrychlit pomocí NN. Experiment je možné spustit ze souboru `assignment/machine_learning/neural_networks/neural_network_speedup.py`.

Na začátku jsou nastaveny parametry trénování a algoritmu, který má být natrénován. Je nastaven počet pracovníků, počet úkolů, počet typů schopností, počet tréninkových epizod, velikost testovacího datasetu vůči trénovacímu datasetu a pracovníku, pro kterého se budou dělat predikce. Je inicializován objekt `DatasetCreator`,

který vytváří dataset dle specifikovaných parametrů.

Unitř smyčky vykonávané tolikrát, kolik je počet datasetů je volána metoda `create` dříve inicializovaného objektu `DatasetCreator`. Dataset je přiřazen Maďarskou metodou. Následně jsou jak dataset, tak i výsledné přiřazení uloženy jako součást trénovacího datasetu.

Po vytvoření datasetu je vytvořen model NN. Síť tvoří pět plně propojených vrstev o 256, 196, 128, 64, 32 neuronech respektive. Všechny vrstvy používají aktivační funkci ReLU. Výstupní vrstva má t neuronů, kde t je počet úkolů v datasetu. Jako ztrátová funkce je použita kategorická ohnisková křížová entropie (categorical focal crossentropy).

Za účelem kontroly vývoje tréninku modelu je definováno několik vlastních metrik. Metriky jsou založeny na F1 skóre. To lze vypočíst jako:

$$\frac{2tp}{2tp + fp + fn},$$

kde tp je počet pravdivých pozitivních nálezů, fp je počet falešných pozitivních nálezů a fn je počet falešných negativních nálezů. [36]

Následně je model zkompilován. Dataset je rozdělen na trénovací a testovací. S trénovacím datasetem započne trénink. Jakmile trénink skončí jsou ve složce `plots` vytvořeny dva grafy. Jeden graf dokumentuje průběh ztrátové funkce a druhý graf zaznamenává hodnoty F1 na testovacím datasetu v průběhu tréninku.

Nakonec je vypsán vektor predikcí. Složka vektoru na pozici i vyjadřuje pravděpodobnost přiřazení úkolu i k definovanému pracovníkovi. Také je vypsáno jaké úkoly by měly být pracovníkovi přiřazeny dle maďarské metody.

Náhodné přiřazení

Lze spustit souborem `random_assignment.py`. Program nejprve načte dataset ze souboru do paměti. Zjistí unikátní identifikátory testerů. V cyklu projde všechny úkoly, kde vytvoří přiřazení. Přiřazení se vytvoří z aktuálně procházeného úkolu a náhodného testera. Náhodný tester je vybrán funkcí `random.choice` z dříve zjištěných unikátních identifikátorů.

4.3.2 Rozhraní

Jako nejlepší a prakticky nejpoužitelnější byly vybrány algoritmy maďarské metody a lineárního programování. Pro tyto algoritmy bylo naprogramováno rozhraní, jež je možné spustit ze souboru `initial_assignment.py`. Soubor přijímá argumenty:

- `-d` povinný argument, cesta k datasetu který nad kterým má být spuštěn přiřazovací algoritmus,

- `-o` jméno výstupního souboru, kam bude výsledné přiřazení zapsáno, pokud je argument nevyplněn, bude soubor pojmenován `assignment.json`,
- `-r` volitený argument, cesta k souboru s omezeními.

Soubor s omezeními musí mít standardní JSON formát ve tvaru, který je zobrazen ve výpise 4.3. V souboru je vždy pro pracovníka s identifikátorem `worker_id` nastaven

Výpis 4.3: Příklad souboru omezení

```
[
    {
        "worker_id": 1,
        "maximum_tasks": 3
    },
    {
        "worker_id": 2,
        "maximum_tasks": 1
    }
]
```

maximální počet úkolů `maximum_tasks`, který má pracovníkovi být přiřazen.

V případě, že soubor není přiložen, spustí se Maďarská metoda. V opačném případě bude použito lineární programování a omezení budou čerpána ze souboru s omezeními. Po dokončení algoritmu přiřazení je vytvořen JSON soubor s výsledným přiřazením, příkladný soubor je ve výpise 4.4. Ve výsledném souboru `task_id` iden-

Výpis 4.4: Příklad souboru přiřazení

```
[
    {
        "worker_id": 1,
        "task_id": 2
    },
    {
        "worker_id": 2,
        "task_id": 1
    }
]
```

tifikátor úkolu, který má být přiřazen k pracovníkovi s identifikátorem `worker_id`.

4.4 Generování zpětné vazby

Dataset zpětné vazby, nebo také historických záznamů o práci, lze generovat spuštěním souboru `create_json_feedbackdataset.py`. Soubor přijímá argumenty:

- `-d` povinný argument, udává cestu k datasetu obsahujícímu úkoly a pracovníky pro něž mají být záznamy o práci vygenerovány,
- `-o` povinný argument, název výstupního souboru zpětné vazby.

Samotný generátor je implementován ve třídě `FeedbackDatasetCreator`, které soubor pouze předává parametry a volá její metodu `create`.

V metodě `create` je nejprve dataset úkolů a pracovníků rozřazen Maďarskou metodou. Pracovníci tedy dostanou úkoly které by jim byly prvotně přiřazeny (bez zpětné vazby). Dále je inicializována hashovací tabulka. Klíčem tabulky je dvojice *pracovník*, *požadovaná schopnost* a hodnotou tabulky je desetinné číslo udávající rozdíl mezi skutečnou schopností a zadanou schopností v datasetu. Následuje cyklus pro každé vygenerované přiřazení.

Pro úkol jednotlivého přiřazení jsou zjištěny potřebné schopnosti. Pro každou potřebnou schopnost je sestavena dvojice *přiřazený pracovník*, *potřebná schopnost*. Dvojice je vyhledána v hashovací tabulce. Pokud se v tabulce nachází, je nalezená hodnota použita jako rozdíl mezi reálnou a zadanou testerovou schopností. Pokud se v tabulce žádná hodnota pro sestavenou dvojici nenachází, je vygenerována nová hodnota reprezentující rozdíl mezi reálnou a zadanou testerovou schopností. Hodnota je vygenerována náhodně z normálního rozdělení se směrodatnou odchylkou $\sigma = 0,1$ a střední hodnotou $\mu = 0$. Následně jsou rozdíly pro úkol sečteny a představují procentuální rozdíl mezi reálnou a očekávanou dobou trvání úkolu, označen jako td.

Očekávaná doba trvání úkolu v sekundách, značena ex, je vypočtena jako:

$$[[\frac{r}{300}] \cdot 300],$$

kde $[x]$ značí funkci zaokrouhlující x dolů na nejbližší celou hodnotu, $r \in <600, 7200)$ je náhodná hodnota z rovnoměrného rozdělení. Reálná doba trvání úkolu je vypočtena jako:

$$[ex \cdot (1 + td),]$$

počet nalezených špatných nálezů (falešných pozitiv) v úkolu je spočten jako:

$$[max([\frac{td \cdot 20}{5}], 0)],$$

nahlášená obtížnost úkolu pracovníkem je spočtena jako:

$$[min(max(10td + r, -5), 5)] + 5,$$

vypočtené údaje jsou uloženy jako historický záznam práce pracovníka přiřazeného k úkolu. Názorná ukázka výsledného souboru je ve výpisu 4.5. Hodnota `worker_id`

Výpis 4.5: Příklad souboru vzpětné vazby

```
{
    "feedbacks": [
        {
            "worker_id": 1,
            "task_id": 1,
            "duration": 1059,
            "expected_duration": 1200,
            "false_positives": 0,
            "reported_difficulty": 2,
            "created_timestamp": 1715274974
        },
        {
            "worker_id": 2,
            "task_id": 2,
            "duration": 3252,
            "expected_duration": 3600,
            "false_positives": 0,
            "reported_difficulty": 5,
            "created_timestamp": 1715274984
        }
    ]
}
```

udává identifikátor pracovníka, `task_id` identifikátor úkolu. Hodnota `duration` je reálná doba strávená pracovníkem na úkolu udávaná ve vteřinách, `expected_duration` je očekávaná doba strávená na úkolu také ve vteřinách, `false_positives` je počet chybných pozitivních nálezů testerem na úkolu, `reported_difficulty` je zadaná náročnost testerem na úkolu. Datum a čas vykonání práce je vyjádřeno `created_timestamp` jako počet vteřin od 1.1.1970 (Unixový čas).

4.5 Přiřazení dle zpětné vazby

Jak bylo popsáno v podkapitole 2.1, přiřazení na základě zpětné vazby lze realizovat jakmile pro každého pracovníka je alespoň n záznamů zpětné vazby, kde n je velikost historického okna. Přiřazení na základě těchto záznamů lze spustit ze souboru `feedback_based_assignment.py`. Pro spuštění lze použít použít argumenty:

- `-d` povinný argument, udává cestu k souboru s datasetem úkolů a pracovníků,
- `-o` nepovinný argument, název výstupního souboru přiřazení, standardně zvolen jako `assignment.json`,
- `-r` nepovinný argument, cesta k souboru s omezeními, struktura stejná jako ve výpise 4.3,
- `-f` povinný argument, soubor se zpětnou vazbou struktura ve výpise 4.5,
- `-w` nepovinný argument, cesta k souboru obsahující váhy NN realizující časovou predikci, výchozí hodnota je `feedback/predictor_weights.keras`.

Po spuštění je nejprve zkontrolováno, zda-li jsou soubory na uvedených cestách dostupné. Poté je vytvořen objekt `FeatureExtractor` realizující extrakci příznaků ze záznamů o práci. Objektu jsou předány dva parametry. Dataset s úkoly a pracovníky a dataset se zpětnou vazbou. Parametr udávající velikost historického okna je ponechán na hodnotě 19.

4.5.1 Extrakce příznaků

Extrakce příznaků je spuštěna metodou `get_new_features` objektu `FeatureExtractor`. Nejprve je sestaven seznam úkolů, pro které neexistuje záznam v datasetu záznamů o práci tj. úkoly co nebyly dosud vykonány. Seznam úkolů je předán metodě `get_predict_features` zároveň se všemi pracovníky v datasetu.

V metodě je pro každý nevykonaný úkol a každého pracovníka nalezeno n nejpodobnějších již vykonaných úkolů metodou `_get_most_similar_tasks`, n je velikost historického okna. Podobnost je posuzována na základě schopností, které úkoly vyžadují ke svému splnění. Nabývá hodnot od nuly do jedné a je spočtena jako $\frac{s}{c}$, kde s je počet společných schopností dvou úkolů a $c = max(s_1, s_2)$, přičemž s_i je počet schopností vyžadovaných úkolem i. Pro co nejrychlejší dobu trvání algoritmu je

využita datová struktura haldy, kde klíčem je právě podobnost. Pokud počet úkolů v haldě přesahuje historické okno, tak je z haldy vyjmut prvek s nejmenším klíčem tj. podobností.

Po získání nejpodobnějších dokončených úkolů pro nevykonaný úkol je z každého podobného úkolu sestrojen vektor obsahující čtyři příznaky. První příznak je *časový rozdíl* spočítaný jako:

$$\frac{rd - ex}{ex},$$

kde rd je doba trvání vykonání úkolu a ex je očekávaná doba trvání úkolu. Dalšími příznaky jsou počet *falešných pozitiv*, *nahlášená obtížnost* a *podobnost úkolu*.

Jakmile je operace dokončena pro všechny nevykonané úkoly a pracovníky, tak je matice obsahující uložené vektory s příznaky konvertována na datový typ `float16`. Konverze je provedena za účelem úspory paměti. Bylo ověřeno že všechny hodnoty jsou i po konverzi odlišné pouze v rámci tolerance $\pm 1 \cdot 10^{-3}$.

4.5.2 Predikce času

Se získanou maticí příznaků jako argumentem je zavolána metoda `predict` objektu `DurationPredictor`. Objekt je inicializován s NN a jejími váhami. Vrstvy NN jsou implementovány pomocí knihovny `keras`, jež tvoří nadstavbu populární knihovny `tensorflow`. Vstupní vrstva sítě má $n \cdot p$ neuronů, kde n je historické okno a p je počet parametrů. Standardně je nastaveno $n = 19$ a $p = 4$. Výstupní vrstva je tvořena jedním neuronem, jehož výstupem je predikovaný časový rozdíl. Metoda `predict` potom jen transformuje rozměry vstupní matice na rozměry požadované vstupní vrstvou a získá s pomocí modelu predikci.

4.5.3 Přiřazení

Poté, co jsou získány predikované časové rozdíly pro celou matici, program pokračuje algoritmem přiřazení. Algoritmus přiřazení postupuje stejně jako bylo popsáno v podkapitole 4.3.2.

Jediný rozdíl je v tom, že kvalifikační matice je sestavena z predikovaných časových rozdílů pro plnění úkolu. Opět je tedy možné použít soubor s restrikcemi a přiřazení realizovat pomocí lineárního programování. Výstupem celého algoritmu je tedy soubor přiřazení, strukturou stejný jako výpis 4.4.

5 Porovnání

Dvě nejdůležitější komponenty dvou-krokového návrhu přiřazení, popsaného v podkapitole 2.1, jsou přiřazovací algoritmus a prediktor času stráveného na úkolu. Možností realizace těchto komponent je ovšem několik. Z toho důvodu bylo provedeno porovnání různých druhů algoritmů přiřazení a architektur časového prediktoru.

5.1 Algorimy přiřazení

Algoritmy přiřazení byly porovnány na vygenerovaném datasetu pomocí souboru popsaného v podkapitole 4.1. Dataset byl vygenerován s 50 testery, 1000 úkoly a 10 typy úkolů (schopností). Porovnání algoritmů dle definovaných metrik z podkapitoly 4.2 je zapsáno do tabulky 5.1. Do tabulky byla navíc přidána doba průměrného trvání přiřazovacího algoritmu. Průměr byl spočten vždy z 10 běhů algoritmu.

5.1.1 Přiřazení na základě metrik

Přiřazení na základě metrik bylo realizováno algoritmy *Přiřazení na základě společných schopností* a *Přiřazení na základě chybějících/ přebytečných schopností*. Oba algoritmy maximalizují danou metriku, z čehož plyne, že v dosahují nejlepších hodnot. Velkou nevýhodou ovšem je, že algoritmy nepracují s konceptem rovnoměrnosti přiřazení. Tedy v zájmu co nejvyšší metriky algoritmus přiřadí stejnému pracovníkovi velký počet úkolů. To vede k situaci, kdy pár nejschopnějších pracovníků má přiřazenou velkou většinu úkolů. V praxi je nutné takové situaci předejít. Algoritmus *Přiřazení na základě chybějících/ přebytečných schopností* způsobuje nerovnoměrnost přiřazení podstatně nižší než algoritmus *Přiřazení na základě společných schopností*, protože metrika bere v potaz i přebývající schopnosti pracovníka. To přirozeně vede k přiřazení nejschopnějších testerů k nejkomplexnějším úkolům. Nicméně i přesto je přiřazení nevyvážené.

5.1.2 Maďarská metoda

Algoritmus Maďarské metody maximalizuje danou metriku v rámci přesně vyváženého přiřazení. Výsledkem tedy bude vždy přiřazení, kde každý pracovník má přiřazen průměrný počet úkolů. To z podstaty věci vede k horším výsledkům schopnostních metrik, než u algoritmů navržených danou metriku maximalizovat. Maďarská metoda je velice rychlá, protože k realizaci přiřazení je použita metoda `linear_sum_assignment` z knihovny `scipy`. Tato `Python` knihovna je velice optimalizována a její jádro je napsáno v programovacím jazyce `C`.

Tab. 5.1:

	Metriky					
Algoritmus	Společné schopnosti [%]	Chybějící schopnost [%]	Chybějící/ přebytečné schopnosti [%]	Nerovnoměrné pracovní rozložení [%]	Chybně přiřazené úkoly [-]	Doba trvání [s]
Lineární programování	49,0340	50,9659	29,4599	0	0	67,9903
Lineární programování (unbound)	78,7158	21,2841	36,6550	75,6122	0	3,3667
Maďarská metoda	48,4780	51,5219	35,7939	0	0	0,0060
Přiřazení podle společných schopností	78,7158	21,2841	36,5689	75,4081	0	18,1674
Přiřazení podle chybějících/ přebytečných schopností	60,2803	39,7196	25,2100	39,2857	0	18,2464
Q-Učení	33,7355	66,2644	52,1420	8,2653	0	93,1108
Hluboké Q-Učení	34,2028	65,7971	51,9240	16,3265	0	4006,237
Náhodné přiřazení	32,7766	67,2233	52,5266	9,0918	0	0,3633

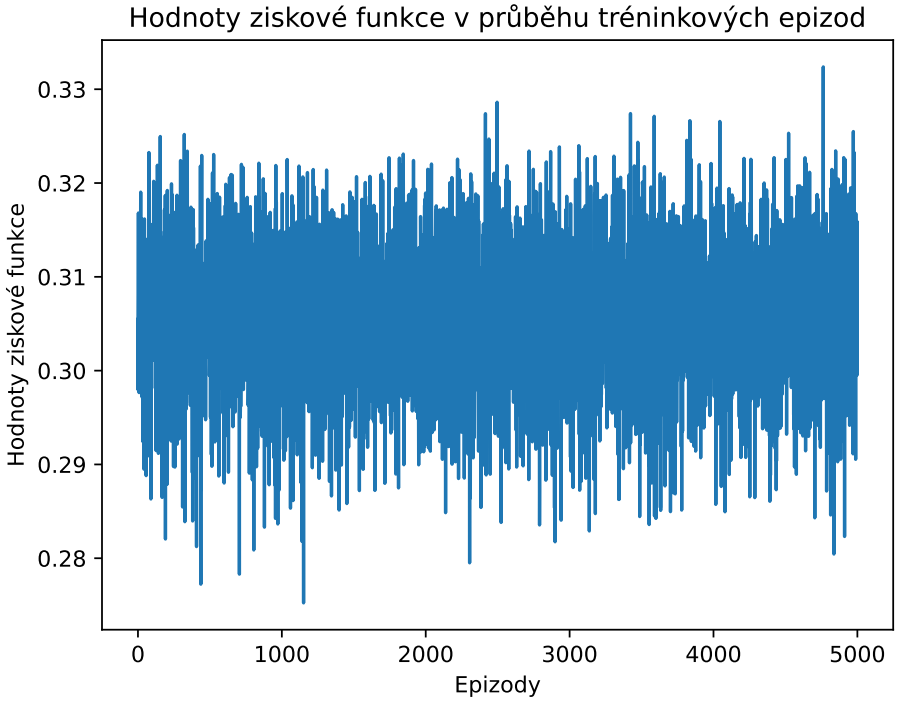
5.1.3 Lineární programování

Na zvolených metrikách byly porovnány dva algoritmy založené na lineárním programování, které se liší ve svých omezeních. Algoritmus označen v tabulce 5.1 jako *Lineární programování* obsahuje omezení na přiřazení průměrného počtu úkolů ke každému testerovi. Toto omezení v algoritmu *Lineární programování (unbound)* chybí. Výsledné přiřazení obou algoritmů je tedy výrazně odlišné. Zatímco *Lineární programování* je srovnatelné s Maďarskou metodou v absolutně rovnoměrném přiřazení, *Lineární programování (unbound)* maximalizuje metriku *Společné schopnosti*. Skrze omezení lineárního programování lze tedy docílit dramaticky jiný přiřazení. V praxi toho lze výhodně využít, kdy jednotlivým testerům lze přiřadit požadovaný počet úkolů, například na základě jejich pracovního úvazku. Nevýhodou algoritmu oproti

předchozím je malá rychlost. I přesto doba trvání v reálném prostředí není nijak problematická, protože na datasetu o kvalifikační matici $50 \cdot 1000$ trvá algoritmus v průměru něco málo přes jednu minutu. Pokud ovšem by se měl každému pracovníkovi přiřadit průměrný počet úkolů, je výhodnější z hlediska rychlosti použít Maďarskou metodu.

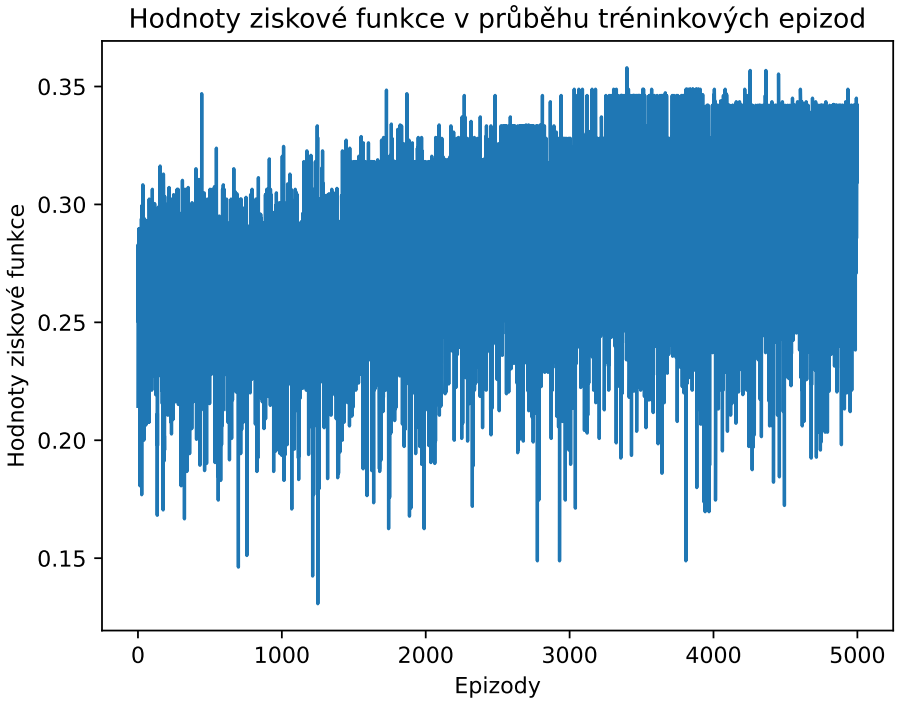
5.1.4 Q-Učení

Pro účely porovnání bylo Q-Učení spuštěno na 5000 tréninkových epizodách. Zisková funkce byla definována jako $c - 0,25w$, kde c jsou společné schopnosti a w je nerovnoměrnost přiřazení. Algoritmus dosáhl hodnot velice podobných náhodnému přiřazení, protože nebyl schopen se efektivně učit. Jak je patrné z grafu 5.1 zisková funkce v průběhu epizod příliš nestoupala. I přesto, že algoritmus by teoreticky



Obr. 5.1: Q-Učení na velkém datasetu.

mohl mít více tréninkových epizod, tak velikost Q-tabulky začne rychle narůstat. Po všech 5000 tréninkových episodách alokuje tabulka 4,04 GB paměti RAM. Q-Učení bylo otestováno i na menším datasetu s kvalifikační maticí $2 \cdot 46$. Výsledek testu je vyobrazen v grafu 5.2. Algoritmus je schopný se učit, pouze ale na mnohem menších akčních a stavových prostorech. Algoritmus tedy nemá příliš vhodné praktické využití.



Obr. 5.2: Q-Učení na malém datasetu.

5.1.5 Hluboké Q-Učení

Při zachování stejné definice ziskové funkce jako v předchozí podkapitole 5.1.4, dosahuje hluboké Q-Učení jen nepatrně lepších metrik než Q-Učení. Pro trénink bylo zvoleno 300 epizod, protože každá epizoda trvá déle než u algoritmu Q-Učení. I přesto celková doba trvání algoritmu přesáhla jednu hodinu, což je maximálního hodnota přijatelná pro praktickou aplikaci. Hodnota ziskové funkce začne velice rychle stoupat během prvních 10 tréninkových epizod. Nicméně během zbylých epizod hodnota ziskové funkce stagnuje, jak lze vidět na Obr. 5.3. Algoritmus tedy stejně jako Q-Učení nedokáže pracovat s tak velkým akčním a stavovým prostorem a není vhodný k použití pro problémy přiřazení s velkou kvalifikační maticí.

5.2 Modely prediktoru

Pro účely srovnání modelů prediktoru času stráveného na úkolu byl vytvořen dataset, který se skládá ze vstupní matice umístěné v `dataset/ml/x.npy` a výstupního vektoru `dataset/ml/y.npy`. Za pomocí knihovny `KerasTuner` bylo na datasetu otestováno několik modelů neuronových sítí pro prediktor času stráveného na úkolu. Detailní provedení jednotlivých modelů je zapsáno v tabulce 5.2. Sloupec tabulky

Obr. 5.3: Hluboké Q-Učení na velkém datasetu.

Počet neuronů udává počet neuronů ve skrytých vrstvách kromě poslední skryté vrstvy, která má vždy 19 neuronů. Redukce neuronů popsána v podkapitole 3.3.2 jejíž míra je udávána sloupcem *Míra redukce počtu neuronů* je aplikována pro každou v pořadí lichou skrytou vrstvu. Regularizace L2 je aplikována pro všechny skryté vrstvy, kromě poslední, s parametrem λ rovným hodnotě ve sloupci *Hodnota L2 regularizace*. Konvoluční vrstvy, jejichž počet je dán sloupcem *Počet konvolučních vrstev*, jsou vždy umístěny před plně propojená vrstvy.

Porovnání přesností modelů z tabulky je vyobrazeno v grafu 5.4. Přesnost je spočtena jako:

$$1 - mae(y, \hat{y}),$$

kde *mae* je střední absolutní chyba definována v podkapitole 3.1, y je vektor strávených časů na úkolech a $\hat{y}$ je vektor predikovaných strávených časů na úkolech. Všechny modely byly testovány pro rychlosti učení $0{,}001; 0{,}0005; 0{,}0001$. Nejvyšší přesnosti 98,07% dosahuje *model 3* s rychlostí učení 0,0005. Jeho architektura je zobrazena na Obr. 5.6.[1] Model obsahuje normalizační vrstvu (BatchNormalization), dvě konvoluční vrstvy (Conv1D) a dvě skryté vrstvy (Dense). Váhy tohoto modelu byly uloženy do souboru `feedback/predictor_weights.keras`. Učinit jednu predikci času stráveného na úkolu modelu trvá v průměru 3ms.

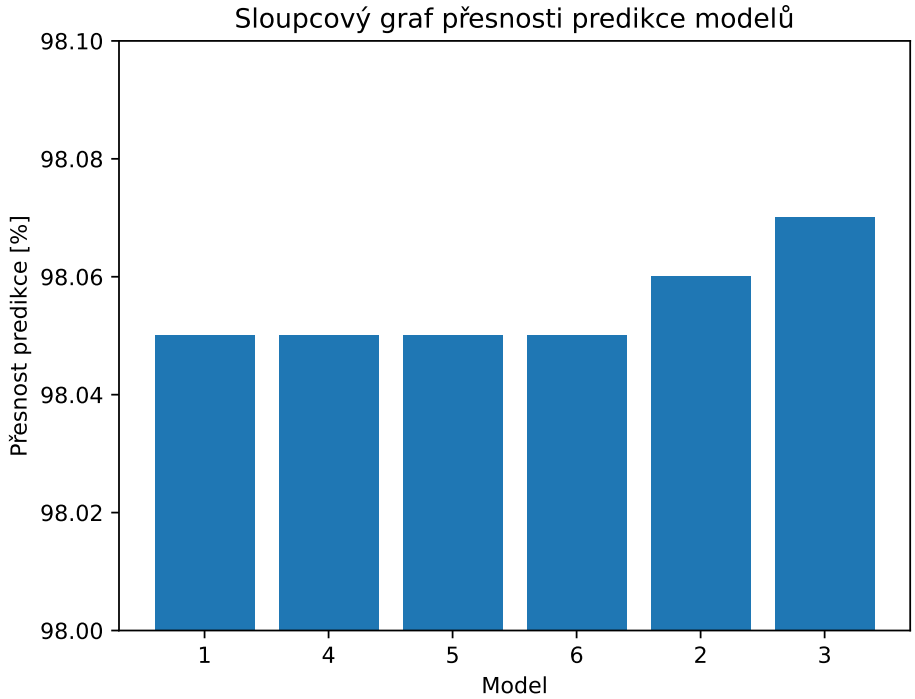
[1]Obrázek byl vygenerován s pomocí online nástroje `https://netron.app/`.

Tab. 5.2:

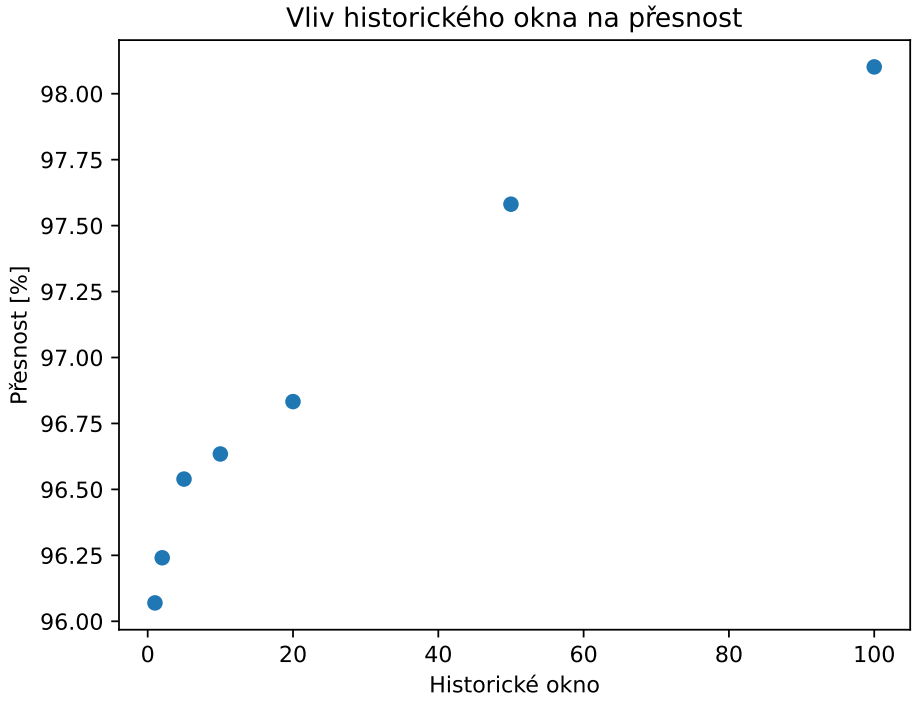
Model	Počet konvolučních vrstev	Počet konvolučních filtrů	Velikost konvolučního filtru	Počet skrytých vrstev	Počet neuronů	Míra redukce počtu neuronů	Hodnota L2 regularizace
1	0	0	0	5	152	0	$1 \cdot 10^{-5}$
2	1	64	2	3	304	0,2	$1 \cdot 10^{-5}$
3	2	32	4	2	152	0	$1 \cdot 10^{-3}$
4	1	32	4	4	152	0,5	$1 \cdot 10^{-4}$
5	1	32	2	5	152	0	$1 \cdot 10^{-4}$
6	0	0	0	1	152	0	$1 \cdot 10^{-5}$

5.2.1 Historická okna

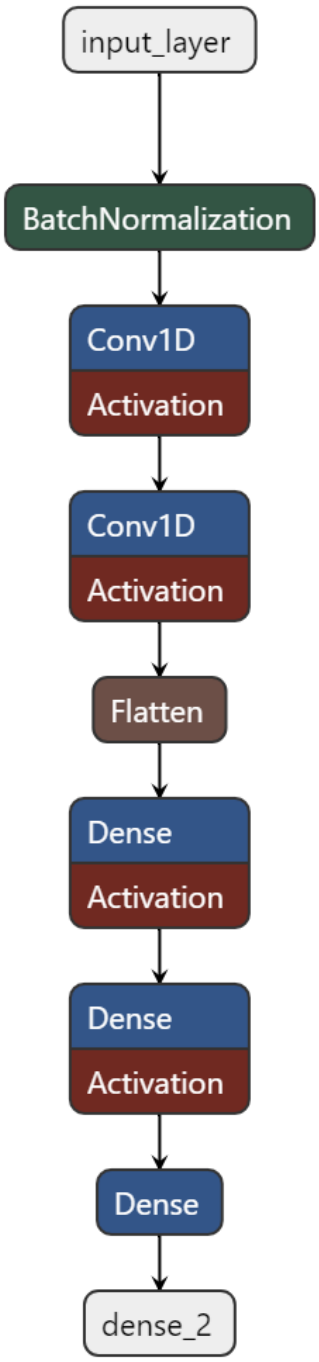
Na nejlepším modelu byla porovnána různá velikost historických oken. Porovnání probíhalo na vygenerovaném datasetu, který obsahoval 100 historických položek pro každého testera. Všechny modely byly trénovány 20 epizod a finální přesnost byla zanesena do grafu 5.5. Z grafu je patrné, že s rostoucím historickým oknem roste i přesnost modelu. Největší nárůst přesnosti je v rámci 5 historických položek. Nastavené velikost historického okna je tedy zejména omezena praktickou skutečností, že pro všechny penetrační testery nemusí být dostupný požadovaný počet historických záznamů.



Obr. 5.4: Přesnosti predikce modelů.



Obr. 5.5: Porovnání přesnosti modelu v závislosti na velikosti historického okna.



Obr. 5.6: Architektura modelu 3.

Závěr

Hlavním cílem práce bylo navrhnout a implementovat nástroj pro automatické přiřazení penetračních testerů k úkolům penetračního testování. Přičemž nástroj má také brát v úvahu historický výkon testerů na přiřazených úkolech.

V rámci teoretické práce nejprve proběhla analýza několika hlavních metod řešení problému přiřazení. Hlavními dvěma zanalyzovanými oblastmi jsou optimalizační problémy a strojové učení. Další část práce navrhuje algoritmus realizující přiřazení penetračních testerů k úkolům pomocí dvou kroků. Nejprve proběhne přiřazení na základě schopností, které testeři sami uvedli v dotazníku. Jakmile jsou k dispozici historické záznamy o práci penetračních testerů, je přiřazení realizováno na základě nich. Historické záznamy jsou přetvořeny neuronovou sítí na predikovaný čas strávený na úkolu, který je vstupem přiřazovacího algoritmu. Detailní teoretický popis neuronových sítí a souvisejících komponent je rozebrán ve třetí kapitole.

Praktická část popisuje implementaci samotného dvou-krokového algoritmu realizujícího přiřazení. Jsou popsány také generátory datových sad a implementované metriky měřící kvalitu přiřazení. Z porovnání se jako nejlepší algoritmus přiřazení jeví lineární programování a Maďarská metoda. Lineární programování je využito pokud každý penetrační tester má vypracovat jiné množství úkolů. Pokud má být každému testerovi přiřazeno stejné množství úkolů je využita Maďarská metoda. Navržené prediktory časů strávených na úkolu dosahují přesnosti přes 95%, kterou je možné ještě zvýšit zvětšením počtu historických záznamů na vstupu algoritmu. Výstupem je tedy nástroj který výrazně zefektivní přidělení úkolů penetračním testerům a tak celkově zvýší efektivitu penetračního testování v rámci týmů.

Literatura

[1] TULSIAN, P. C.; PANDEY V. *Quantitative techniques: theory and problems* 2006.

[2] LAWLER, E. L. *The quadratic assignment problem.* In: Management science. 1963, roč. 9, č. 4, s. 586–599.

[3] ŠMEREK, M. *Linear programming - Operations Research.* Online. UNOB. Dostupné z: `https://moodle.unob.cz/pluginfile.php/43732/mod_resource/content/1/AJ_OV_T11.pdf`. [cit. 2023-12-01].

[4] *Maďarská metoda - co to je, definice a koncept.* Online. Economy-Pedia. Dostupné z: `https://cs.economy-pedia.com/11033922-hungarian-method`. [cit. 2023-12-01].

[5] NARAYANAN, A. et al. *Experimental comparison of hungarian and auction algorithms to solve the assignment problem.* 2000.

[6] Gale, D.; Shapley, L. S. *College admissions and the stability of marriage.* In: The American Mathematical Monthly. 1962, roč. 69, č. 1, s. 9–15.

[7] MANLOVE, D. F. *Hospital/residents problem.* 2008.

[8] Bertsekas, D. P. *A distributed algorithm for the assignment problem* In: Lab. for Information and Decision Systems Working Paper, MIT. 1979.

[9] KUHN, H. W. *The Hungarian method for the assignment problem.* In: Naval research logistics quarterly. 1955, roč. 2, č. 1-2, s. 83–97.

[10] PECHERKOVÁ, P.; JOZOVÁ, Š.; NAGY, I. *Lineární programování I.* Online. ČVUT. Dostupné z: `https://www.fd.cvut.cz/personal/nagyivan/LinPrg1/LP1Skripta.pdf`. [cit. 2023-12-01].

[11] ANDERSEN, E. D. et al. *Implementation of interior point methods for large scale linear programming.* HEC/Université de Geneve, 1996.

[12] MARTELLO, S.; TOTH, P. *Linear assignment problems.* In: North-Holland Mathematics Studies. 1987, roč. 132, s. 259–282. ISSN 0304-0208.

[13] NAZEMI, A.; GHEZELSOFLA, O. *A dual neural network scheme for solving the assignment problem.* In: The Computer Journal. 2017, roč. 60, č. 3, s. 431—443.

[14] MNIH, V. et al. *Playing atari with deep reinforcement learning.* In: arXiv preprint arXiv:1312.5602. 2013.

[15] SHAH, B.; BHAVSAR, H. *Time complexity in deep learning models.* In: Procedia Computer Science. 2022, roč. 215, s. 202—210.

[16] LEE, M. et al. *Deep neural networks for linear sum assignment problems.* In: IEEE Wireless Communications Letters. 2018, roč. 7, č. 6, s. 962-–965.

[17] RAJAWAT, A. S. et al. *IoT in renewable energy generation for conservation of energy using artificial intelligence.* In: Applications of AI and IOT in Renewable Energy. 2022, s. 89-–105.

[18] LIU, H. *Chapter 5—single-point wind forecasting methods based on reinforcement learning.* In: Wind Forecasting in Railway Engineering; Liu, H., Ed. Elsevier: Amsterdam, The Netherlands. 2021, s. 177-–214.

[19] YUEN, M.; KING, I.; LEUNG, K. *Task matching in crowdsourcing.* In: 2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing. 2011, s. 409—412.

[20] DENG, D.; SHAHABI, C.; ZHU, L. *Task matching and scheduling for multiple workers in spatial crowdsourcing.* In: Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems. 2015, s. 1—10.

[21] YANG, G. et al. *Competition-congestion-aware stable worker-task matching in mobile crowd sensing.* In: IEEE Transactions on Network and Service Management. 2021, roč. 18, č. 3, s. 3719-–3732.

[22] QIAO, L.; TANG, F.; LIU, J. *Feedback based high-quality task assignment in collaborative crowdsourcing.* In: 2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA). 2018, s. 1139—1146.

[23] MATSUBARA, M. et al. *Task Assignment Strategies for Crowd Worker Ability Improvement.* In: Proceedings of the ACM on Human-Computer Interaction. 2021, roč. 5, č. CSCW2, s. 1-–20.

[24] GOEL, G.; NIKZAD, A.; SINGLA, A. *Matching workers expertise with tasks: incentives in heterogeneous crowdsourcing markets.* In: NIPS Workshop on Crowdsourcing. 2013.

[25] DAS, B.; SHIKDAR, A. A. *Participative versus assigned production standard setting in a repetitive industrial task: a strategy for improving worker productivity.* In: International journal of occupational Safety and Ergonomics. 1999, roč. 5, č. 3, s. 417—430.

[26] CHONG, V.; LEUNG, S. T. *The effect of feedback, assigned goal levels and compensation schemes on task performance.* In: Asian Review of Accounting. 2018, roč. 26, č. 3, s. 314—335.

[27] WU, Y.; FENG, J. *Development and application of artificial neural network.* In: Wireless Personal Communications. 2018, roč. 102, s. 1645-–1656.

[28] ABDI, H. *A neural network primer.* In: Journal of Biological Systems. 1994, roč. 2, č. 3, s. 247—281.

[29] JANOCHA, K.; CZARNECKI, W. M. *On loss functions for deep neural networks in classification.* In: arXiv preprint arXiv:1702.05659. 2017.

[30] GUPTA, S. *7 Most Common Machine Learning Loss Functions* Online. Built In. Dostupné z: `https://builtin.com/machine-learning/common-loss-functions`. [cit. 2024-05-01].

[31] APICELLA, A. et al. *A survey on modern trainable activation functions.* In: Neural Networks. 2021, roč. 138, s. 14–32.

[32] VAN LAARHOVEN, T. *L2 regularization versus batch and weight normalization.* In: arXiv preprint arXiv:1706.05350. 2017.

[33] SRIVASTAVA, N. et al. *Dropout: a simple way to prevent neural networks from overfitting.* In: The journal of machine learning research. 2014, roč. 15, č. 1, s. 1929-–1958.

[34] SHAO, J. et al. *Is normalization indispensable for training deep neural network?.* In: Advances in Neural Information Processing Systems. 2020, roč. 33, s. 13434-–13444.

[35] ALBAWI, S.; MOHAMMED, T. A.; AL-ZAWI, S. *Understanding of a convolutional neural network.* In: 2017 international conference on engineering and technology (ICET). 2017, s. 1-–6.

[36] LIPTON, Z. C.; ELKAN, C.; NARAYANASWAMY, B. *Thresholding classifiers to maximize F1 score.* In: arXiv preprint arXiv:1402.1892. 2014.