



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

**VELKÉ JAZYKOVÉ MODELY PRO GENEROVÁNÍ KÓDU
SE ZAMĚŘENÍM NA VESTAVĚNÉ SYSTÉMY**

LARGE LANGUAGE MODELS FOR GENERATING CODE FOCUSING ON EMBEDDED SYSTEMS

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

MATEJ VADOVIČ

VEDOUcí PRÁCE

SUPERVISOR

doc. RNDr. PAVEL SMRŽ, Ph.D.

BRNO 2024

Zadání bakalářské práce



154306

Ústav: Ústav počítačové grafiky a multimédií (UPGM)
Student: **Vadovič Matej**
Program: Informační technologie
Název: **Velké jazykové modely pro generování kódu se zaměřením na vestavěné systémy**
Kategorie: Informační systémy
Akademický rok: 2023/24

Zadání:

1. Seznamte se se způsoby využití rozsáhlých jazykových modelů typu GPT-3 a jejich adaptace na doménových datech, např. archivu GitHub a StackExchange.
2. Zpracujte dostupná data komentovaných kódů pro vestavěné systémy tak, aby bylo možné průběžně vyhodnocovat výsledky vytvářených modelů.
3. Na základě získaných poznatků navrhnete a implementujete systém, který dokáže na základě hlaviček funkcí a prvotních komentářů navrhovat strukturu následného kódu, se zaměřením na vybrané vestavěné systémy.
4. Vyhodnoťte výsledky systému na reprezentativním vzorku dat, diskutujte závislost výsledků na intuitivní složitosti popisu a adaptaci pro specifické platformy.
5. Vytvořte stručný plakát prezentující práci, její cíle a výsledky.

Literatura:

- dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- funkční prototyp řešení

Podrobné závazné pokyny pro vypracování práce viz <https://www.fit.vut.cz/study/theses/>

Vedoucí práce: **Smrž Pavel, doc. RNDr., Ph.D.**
Vedoucí ústavu: Černocký Jan, prof. Dr. Ing.
Datum zadání: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 21.12.2023

Abstrakt

Cielom tejto práce bola adaptácia predtrénovaného jazykového modelu pre účely generovania kódu v oblasti vstavaných systémov. V práci je predstavená nová dátová sada pre ladenie modelov generovania kódu, ktorá obsahuje 50 tisíc dvojíc zdrojového kódu a komentárov zameraných na oblasť programovania vstavaných systémov. Táto sada je zložená zo zozbieraného zdrojového kódu z platformy GitHub. Na dátach nového korpusu boli ladené dva nové jazykové modely pre generovanie kódu založené na predtrénovaných modeloch s architektúrou transformer. Model MICROCODER je založený na modeli CODELLAMA-INSTRUCT 7B a pri jeho ladení bola využitá technika QLoRA pre minimalizáciu výpočtových nárokov ladenia. Druhý model, MICROCODERFIM, je založený na modeli STARCODERBASE 1B a podporuje vyplňovanie kódu na základe okolia (fill-in-the-middle). Jednotlivé modely boli porovnávané na základe metrík BLEU, CodeBLEU, ChrF++ a ROUGE-L. Model MICROCODERFIM dosahuje najlepšie výsledky adaptácie na novú úlohu, pričom zaznamenal viac ako 120 % zlepšenie vo všetkých meraných metrikách. Váhy modelov spolu s novou dátovou sadou sú voľne prístupné na verejnom úložisku.

Abstract

The goal of this work was to adapt a pre-trained language model for the purpose of generating code in the field of embedded systems. The work introduces a new dataset for fine-tuning code generation models, consisting of 50,000 pairs of source code and comments focused on embedded systems programming. This dataset is composed of collected source code from the GitHub platform. Two new language models for code generation, based on transformer architecture pre-trained models, were fine-tuned on the data of the new corpus. Model MICROCODER is based on the CODELLAMA-INSTRUCT 7B model, and during its fine-tuning, the QLoRA technique was used to minimize computational requirements. The second model, MICROCODERFIM, is based on the STARCODERBASE 1B model and supports code infilling. The individual models were compared based on BLEU, CodeBLEU, ChrF++, and ROUGE-L metrics. Model MICROCODERFIM achieves the best adaptation results to the new task, with over 120 % improvement in all measured metrics. The weights of the models along with the new dataset are freely accessible on a public repository.

Klíčové slová

velké jazykové modely, generovanie kódu, programovanie vstavaných systémov, transformer, dátová sada, CodeLlama, StarCoderBase

Keywords

large language models, code generation, embedded code, transformer, dataset, CodeLlama, StarCoderBase

Citácia

VADOVIČ, Matej. *Velké jazykové modely pro generování kódu se zaměřením na vestavěné systémy*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. RNDr. Pavel Smrž, Ph.D.

Velké jazykové modely pro generování kódu se zaměřením na vestavěné systémy

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána doc. RNDr. Pavla Smrža, Ph.D. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Matej Vadovič
8. mája 2024

Podakovanie

Chcel by som vyjadriť vďaku môjmu vedúcemu práce, doc. RNDr. Pavlovi Smržovi, Ph.D., za jeho odbornú pomoc a vedenie pri príprave tejto práce.

Obsah

1	Úvod	5
2	Základy neurónových sietí	7
2.1	Model neurónu	7
2.1.1	Aktivačná funkcia	8
2.2	Hlboké neurónové siete	9
2.2.1	Spätne šírenie	10
3	Transformer – model hlbokého učenia	12
3.1	Tokenizácia	12
3.1.1	BPE tokenizácia	13
3.2	Mechanizmus pozornosti	14
3.2.1	Samopornosť	14
3.2.2	Multi-headornosť	15
3.3	Architektúra transformer	16
3.3.1	Kodér	17
3.3.2	Dekodér	17
4	Spracovanie prirodzeného jazyka	19
4.1	Veľké jazykové modely	19
4.2	Generatívny predtrénovaný transformer	20
4.3	Generovanie zdrojového kódu	20
5	Trénovanie veľkých jazykových modelov	23
5.1	Predtrénovanie	23
5.2	Ladenie	25
5.2.1	Efektívne ladenie parametrov	25
5.2.2	Kvantizácia váh modelu	25
5.2.3	Adaptácia nízkorozmerných reprezentácií	26
5.2.4	QLoRA	26
5.3	Evaluačné metriky	27
5.3.1	BLEU	27
5.3.2	CodeBLEU	28
5.3.3	ROUGE	29
5.3.4	ChrF	29
5.3.5	Pass@k	29
6	Prehľad súčasných riešení	31

6.1	Dátové sady pre generovanie kódu	31
6.2	Prehľad modelov pre generovanie kódu	31
6.2.1	Codex	31
6.2.2	CodeLLaMA	32
6.2.3	StarCoderBase	33
6.2.4	DeepSeek-Coder	33
7	Návrh a implementácia	34
7.1	Dátová sada	35
7.2	Analýza dátovej sady	38
7.2.1	Možné rozšírenia dátovej sady	39
7.3	Ladenie modelov	40
7.3.1	MicroCoder	41
7.3.2	MicroCoderFIM	42
7.3.3	Emisie CO2	43
7.4	Evaluácia modelov	43
7.4.1	MicroCoder	44
7.4.2	MicroCoderFIM	45
7.4.3	Rýchlosť inferencie	47
8	Záver	48
	Literatúra	49
A	Ďalšie výsledky inferencie modelov	54
A.1	MicroCoder	54
A.2	Github Copilot	56
A.3	MicoCoderFIM	57
B	Výsledky modelu MicroCoderFIM pri rôznych nastaveniach	59

Zoznam obrázkov

2.1	Schéma biologického neurónu. Prebrané z [48].	7
2.2	Perceptrón – základný model umelého neurónu s n vstupmi a váhami, jedným výstupom a prahom citlivosti.	8
2.3	Zjednodušená ilustrácia hlbokej neurónovej siete s tromi skrytými vrstvami	10
3.1	Proces rozkladu vstupného zdrojového kódu na tokeny. Tokeny môžu byť napríklad kľúčové slová, identifikátory, operátory alebo zátvorky.	13
3.2	Vizualizácia naučených váh hláv jednej vrstvy mechanizmu pozornosti modelu BERT s využitím nástroja [44].	14
3.3	Pozornosť skalárneho súčinu (vľavo) a multi-head pozornosť (vpravo). Prebrané z [43].	15
3.4	Architektúra transformer s N kodér a dekodér blokmi. Prebrané z [43]. . . .	16
4.1	Architektúra GPT. Prebrané z [25].	21
5.1	Trénovanie modelu prebieha v niekoľkých fázach. Obvykle začína predtrénovanie, nasledované jemným ladením a posilňovaným učením modelu. Vo fázach sú potrebné rôzne dáta, z ktorým sa model učí.	24
5.2	Váhy matice A sú inicializované náhodne Gaussovým rozdelením. Matica B je inicializovaná nulami, takže matica $\Delta W = BA$ je na začiatku tréningu nulová. Počas tréningu sa A a B upravujú pomocou gradientných metód, čím sa model adaptuje na špecifickú úlohu. Prebrané z [17].	26
5.3	Porovnanie prístupu plného ladenia všetkých váh, techniky LoRA a techniky QLoRA. Prebrané z [10].	27
6.1	Prehľad metód pozornosti. MHA obsahuje rovnaký počet matíc Q , K a V . GQA zdieľa jednu maticu K a V pre skupinu matíc Q , zatiaľ čo MQA zdieľa jedinú maticu K a V pre všetky Q . Prebrané z [2].	33
7.1	Schéma návrhu riešenia práce.	34
7.2	Proces tvorby dátovej sady.	35
7.3	Štruktúra vzorky dátovej sady.	35
7.4	Ukážka príkladu z dátovej sady.	38
7.5	Distribúcia počtu tokenov v popise a tele funkcie.	38
7.6	Distribúcia počtu funkcií extrahovaných z jednotlivých repozitárov.	39
7.7	Distribúcia kľúčových slov v kóde funkcie	39
7.8	Graf zobrazujúci tréningovú a validačnú stratu modelu MICROCODER počas viacerých epoch, ilustrujúci pokrok v učení a hodnotenie výkonnosti modelu.	42

7.9	Graf trénovacej a validačnej straty modelu MICROCODERFIM. Trénovacia strata postupne klesá, zatiaľ čo validačná strata spočiatku jemne klesá, ale následne stúpa. Tento jav naznačuje možný výskyt preučenia. Model dobre funguje na trénovacích dátach, ale jeho schopnosť generalizovať na nové, nevidené dáta, je obmedzená.	43
7.10	Formát promptu použitý pre model GPT-3.5 TURBO.	44
7.11	Výstup modelu MICROCODER (vľavo) a výstup nástroja Github Copilot očistený o sprievodné komentáre (vpravo) pre rovnaký príklad.	45
7.12	Výstup modelu MICROCODER.	45
7.13	Model MICROCODERFIM má problém korektne naviazať na nasledujúci kód funkcie, ale je vidieť, že dobre zachytil kontext pred miestom vloženia nového kódu.	46
7.14	Výstup modelu MICROCODERFIM na zadanie z dátovej sady HumanEval v jazyku Python.	47
A.1	Dva rôzne výstupy modelu MICROCODER na rovnaký dotaz s použitím rovnakých parametrov.	54
A.2	Dva rôzne výstupy modelu MICROCODER na rovnaký dotaz s použitím rovnakých parametrov. Na prvý pohľad je vidno, že dané výstupy sú veľmi odlišné. Prvý výstup (hore) je jednoduchší a priamočiarejší. Definuje dva hlavné príkazy pre nastavenie registra. Druhý výstup (dolu) je zložitejší, obsahuje viac vetvení a kontrolných štruktúr.	55
A.3	Výstup modelu MICROCODER.	55
A.4	Kód vygenerovaný nástrojom GitHub Copilot.	56
A.5	Kód vygenerovaný nástrojom GitHub Copilot. Kód nebol vygenerovaný naraz, ale postupne na tri rôzne nápovedy.	56
A.6	Kód vygenerovaný nástrojom GitHub Copilot. Model bol inštruovaný, aby nakonfiguroval rôzne parametre, ako sú Break feature, dead time a Lock level. Výstup efektívne odráža tieto požiadavky.	56
A.7	Výstup modelu MICROCODERFIM.	57
A.8	Výstup modelu MICROCODERFIM.	57
A.9	Návrh kódu vygenerovaný modelom MICROCODERFIM (vľavo) a referenčné riešenie (vpravo).	58
A.10	Návrh kódu vygenerovaný modelom MICROCODERFIM (vľavo) a referenčné riešenie (vpravo).	58
B.1	Výsledky inferencie modelu MICROCODERFIM pri rôznych nastaveniach teploty, top_p a maximálnej dĺžky generovanej sekvencie. Model dosiahol najlepšie výsledky pri teplote 0.2 a maximálnej dĺžke 128 tokenov.	60
B.2	Zmena výsledkov inferencie modelu MICROCODERFIM v porovnaní so základným nastavením (128 tokenov) pri rôznych hodnotách maximálnej dĺžky generovanej sekvencie. Platí, že pri dĺžke 128 tokenov sa model správa lepšie v troch metrikách. Metrika CodeBLEU vykazuje lepšie výsledky pri dĺžke 256 tokenov.	61

Kapitola 1

Úvod

Vstavané systémy sú neoddeliteľnou súčasťou pokroku a všadeprítomných technológií. Integrované systémy sú obvykle vyvíjané a optimalizované pre predom definovaný účel použitia a platformu. Ich vývoj je často náročný. Vyžaduje pochopenie hardvéru na nízkej úrovni, prácu s referenčnými manuálmi alebo porozumenie dôležitých aspektov operačného systému. Využívanie neurónových sietí prináša vývojárom radu výhod. Od využitia pri vývoji kódu, cez automatizáciu úloh (generovanie dokumentácie, testovanie), až po zabezpečenie určitých bezpečnostných štandardov a kapacitných nárokov (optimalizácia). Výber oblasti integrovaných systémov pre generovanie kódu je motivovaný ich rozsiahlou aplikáciou v rôznych oblastiach, ako sú automobilový priemysel, zdravotníctvo, priemyselná automatizácia alebo IoT zariadenia. To poskytuje široké spektrum možností pre aplikáciu nástrojov, ktoré budú schopné pomáhať s vývojom kódu pre integrované systémy.

Súčasný pokrok v trénovaní veľkých jazykových modelov a v oblasti generovania kódu umožnil vznik nových nástrojov a techník pre automatizáciu programovania. V oblasti generovania kódu sa využívajú jazykové modely, ktoré sú trénované na veľkých dátových sadách a sú schopné generovať kód vysokej kvality. Nástroje ako GitHub Copilot alebo Tabnine, ktoré využívajú jazykové modely na generovanie kódu, sú schopné dopĺňovať kód, navrhovať funkcie, názvy premenných alebo písať popisy funkcií. Tieto nástroje sa tešia veľkej obľube medzi vývojármi, o čom svedčia aj správy o ich využívaní v praxi. GitHub Copilot totiž v súčasnosti stojí za 46% nového kódu na platforme GitHub [51].

Vývoj takto komplexných nástrojov je náročný proces, ktorý si vyžaduje odborné znalosti v oblasti strojového učenia, veľké množstvo dát a výpočtovú silu. Počet hodín, ktoré je potrebné investovať do trénovania modelu, sa pohybuje v státisícoch až miliónoch [22, 49]. Ďalšie zdroje sú vynaložené na vytváranie a správu rozsiahlych dátových sád, na ktorých sa tieto modely učia. Vývoj modelov strojového učenia je preto náročný proces, ktorý si vyžaduje nielen odborné znalosti a skúsenosti v oblasti strojového učenia, ale aj značné zdroje.

Rozvoj v tejto oblasti je aj napriek týmto náročnostiam z veľkej časti otvorený a orientovaný na komunitu. Vývojári z celého sveta prispievajú k vývoju nových modelov, techník a nástrojov, ktoré sú zdieľané s komunitou. Tento prístup umožňuje rýchlejší vývoj nových technológií a nástrojov, ktoré sú následne dostupné pre širokú verejnosť. Vývojári preto nemusia vytvárať vlastné modely, ale môžu využiť existujúce modely a techniky, ktoré sú dostupné ako open-source projekty. Tento prístup umožňuje adaptovať predtrénované modely (pretrained language model, PLM) pre špecifické účely a využiť ich v praxi.

V tejto práci popisujem modely MICROCODER a MICROCODERFIM a dátovú sadu, ktorú som pripravil. Modely MICROCODER a MICROCODERFIM sú zamerané na genero-

vania kódu pre integrované systémy a sú trénované na dátach špecifických pre túto oblasť. V práci popisujem zbieranie dát pre nový korpus zdrojových kódov, proces a techniky tréovania modelov. Analyzoval som výsledky modelov s cieľom poskytnúť prehľad o ich schopnostiach.

V práci sa zameriavam na nasledujúce ciele:

- Zbieranie dát z verejne dostupných repozitárov na platforme GitHub.
- Príprava dátových sád pre tréovanie jazykových modelov.
- Tréovanie jazykových modelov na dátach z oblasti integrovaných systémov.
- Vyhodnotenie výsledkov tréovania modelov.

V kapitole 2 popisujem základné princípy neurónových sietí a ich využitie v oblasti strojového učenia. Začínam základným modelom neurónu a pokračujem hlbokými neurónovými sieťami. Tieto koncepty sú kľúčové pre pochopenie architektúry transformer, ktorá je hlavným objektom záujmu tejto práce. Popisujem ju v kapitole 3. V kapitole 4 sa zameriavam na úlohu spracovania prirodzeného jazyka, generovanie kódu a rozdiely medzi prirodzeným a programovacím jazykom. V kapitole 5 popisujem základné princípy tréovania jazykových modelov a techniky, ktoré sa používajú pri tréovaní a evaluácií modelov. V kapitole 6 poskytujem prehľad súčasných riešení v danej oblasti a diskutujem o ich výhodách a nevýhodách. V kapitole 7 popisujem proces zbierania dát, prípravu dátovej sady, tréovanie modelov MICROCODER a MICROCODERFIM a evaluáciu výsledkov. Kapitola 8 uzaviera prácu sumarizáciou dosiahnutých výsledkov. Zamýšľa sa nad možnosťami rozšírenia dátovej sady a navrhuje smer, ktorým by sa mohlo uberať ďalšie výskumné úsilie.

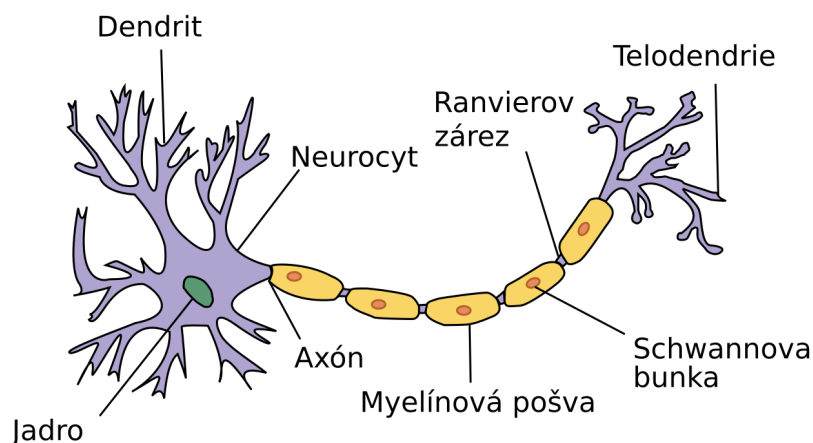
Kapitola 2

Základy neurónových sietí

Prvá kapitola sa zameriava na základné princípy neurónových sietí a ich využitie v oblasti strojového učenia. Začína popisom základného modelu neurónu, ktorý je základnou stavebnou jednotkou neurónových sietí. Nasledujú hlboké neurónové siete, ktoré predstavujú rozšírenie základného modelu a umožňujú modelom učiť sa zložené vzory v dátach. Tieto koncepty sú kľúčové pre pochopenie architektúry transformera, ktorý je objektom záujmu tejto práce.

2.1 Model neurónu

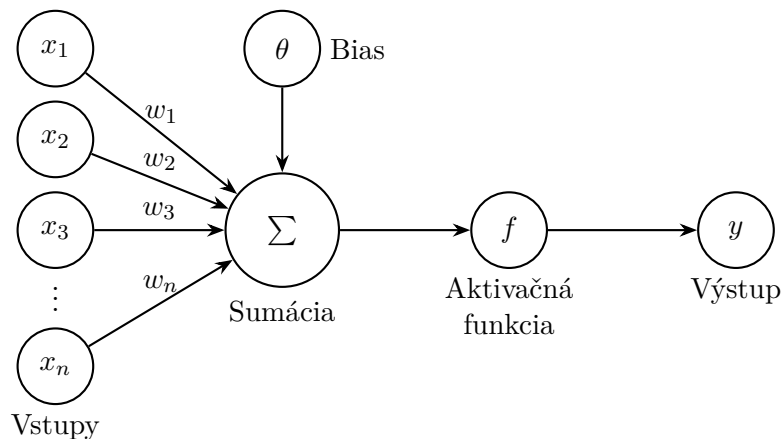
Základnou funkčnou jednotkou ľudskej nervovej sústavy je bunka nazývaná neurón. Neurón, ktorý je na obrázku 2.1, sa skladá z tela bunky (soma), vodivých výbežkov (niekoľko vstupných – dendridy a jeden výstupný – axón, ktoré slúžia na prijímanie, resp. vysielanie elektrických vzruchov) a synapsí (majú pamäťovú funkciu a vyjadrujú priechodnosť pre vstupné vzruchy).



Obr. 2.1: Schéma biologického neurónu. Prebrané z [48].

Prvý formálny model neurónu predstavili v r. 1943 W. S. McCulloch a W. Pitts [27]. Tento prvotný McCullochov-Pittsov neurón nebol schopný učiť sa a fungoval ako binárna prahová funkcia, ktorá berie vstupy, aplikuje váhy a vytvára binárny výstup založený na prahovej hodnote. Dôležitý prelom dosiahol F. Rosenblatt v roku 1962 s jeho modelom neurónu, ktorý nazval PERCEPTRÓN. Rosenblatt navrhol algoritmus učenia, ktorý umožnil

perceptrónu prispôsobovať svoje váhy na základe chýb v predikciách. Týmto spôsobom vznikli prvé jednovrstvové učiace sa neurónové siete.



Obr. 2.2: Perceptrón – základný model umelého neurónu s n vstupmi a váhami, jedným výstupom a prahom citlivosti.

Operácie vykonávané perceptrónom, podľa knihy [28], možno rozdeliť na synaptické operácie (konfluencia) a somatické operácie (agregácia, prahovanie a nelineárne zobrazenie), ktoré spoločne poskytujú základný pohľad na fungovanie perceptrónu.

Synaptické operácie

Do perceptrónu vstupuje n -rozmerný vektor vstupných signálov $x \in R^n$. Operácia konfluencie (splnutia) kombinuje vstupné signály s uloženými synaptickými váhami, ktoré sú reprezentované váhovým vektorom $w \in R^n$. Tieto váhy odrážajú dôležitosť jednotlivých vstupov pre výsledný výstup neurónu.

Somatické operácie

Následne sa výsledok konfluencie agreguje na skalárnu hodnotu. V základnom modeli je možné túto agregáciu nahradiť sumáciou zložiek výsledku konfluencie. Ďalej nasleduje prahovanie a nelineárne zobrazenie. Ak je výsledok sumácie menší ako hodnota prahu (bias) θ , na výstupe sa zobrazí hodnota neaktívneho stavu. V opačnom prípade, ak výsledok prekročí hodnotu prahu, aktivuje sa neurón a výstup sa získa prostredníctvom nelineárnej aktivačnej funkcie f . Tento mechanizmus umožňuje perceptrónu realizovať nelineárne transformácie vstupných signálov, čím zvyšuje jeho schopnosť modelovať komplexné vzťahy. Výstup perceptrónu možno matematicky vyjadriť ako:

$$y = f\left(\theta + \sum_{i=1}^n x_i \cdot w_i\right) \quad (2.1)$$

2.1.1 Aktivačná funkcia

Aktivačná funkcia je nelineárna funkcia, ktorá sa aplikuje na výsledok sumácie vstupov a váh. Oproti binárnej prahovej funkcií, ktorá bola pôvodne používaná, obor hodnôt aktivačnej funkcie je spojitý a umožňuje modelu aproximovať zložitejšie vzory v dátach. Spojitý

obor hodnôt umožňuje modelu odpovedať na malé zmeny vstupov malými zmenami výstupov, čo je dôležité pre učenie modelu. Výber vhodnej aktivačnej funkcie závisí od konkrétnej úlohy, ktorú model rieši. Medzi najpoužívanjšie aktivačné funkcie patria:

- **Sigmoidálna funkcia** – Sigmoidálna aktivačná funkcia je vyhladená verzia binárnej prahovej funkcie, ktorá transformuje vstup na hodnotu medzi 0 a 1. Táto funkcia sa často používa na binárne klasifikačné úlohy, kde model predikuje pravdepodobnosť príslušnosti k jednej z dvoch tried. Sigmoidálna funkcia zohrala dôležitú úlohu pri vývoji algoritmu spätného šírenia, ktorý je základným algoritmom pre tréning hlbokých neurónových sietí, ale v súčasnosti sa používa menej kvôli problémom s gradientom pri tréningu hlbokých sietí.

$$f(x) = \frac{1}{1 + e^{-x}}$$

- **Hyperbolický tangens (tanh)** – Hyperbolický tangens je podobný sigmoidálnej funkcii, ale obor hodnôt funkcie je medzi -1 a 1. Táto funkcia sa často používa na regresné úlohy, kde model predikuje hodnotu v určitom rozsahu. Hyperbolický tangens má výhodu oproti sigmoidálnej funkcii v posunutí, stredná hodnota je 0, čo zjednodušuje učenie modelu.

$$f(x) = \tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

- **Rectified Linear Unit (ReLU)** – Jedná sa o najčastejšie využívanú aktivačnú funkciu implementovanú v skrytých vrstvách neurónovej siete. Na rozdiel od tanh alebo sigmoidálnej funkcie je rýchlejšia, pretože implementuje jednoduchšie operácie.

$$f(x) = \max(0, x) = \frac{x + |x|}{2}$$

Ďalšou výhodou je, že spĺňa podmienku $f(\delta z) = \delta f(z)$ a patrí do skupiny aktivačných funkcií, ktoré nie sú závislé od škálovania vstupu.

Jednou z nevýhod ReLU je, že sa neuróny môžu dostať do stavu, v ktorom sa stanú neaktívne pre „všetky“ vstupy – varianta miznúceho gradientu. ReLU má niekoľko variant, ktoré ponúkajú riešenia – GELU, leaky ReLU, SiLU, Softplus a iné.

- **Softmax** – Softmax je aktivačná funkcia, ktorá sa používa na klasifikačné úlohy s viacerými triedami. Funkcia transformuje vstup na pravdepodobnostné rozdelenie, ktoré vyjadruje pravdepodobnosť príslušnosti k jednej z n tried.

$$f(x) = \frac{e^{x_i}}{\sum_{j=1}^n e^{x_j}}$$

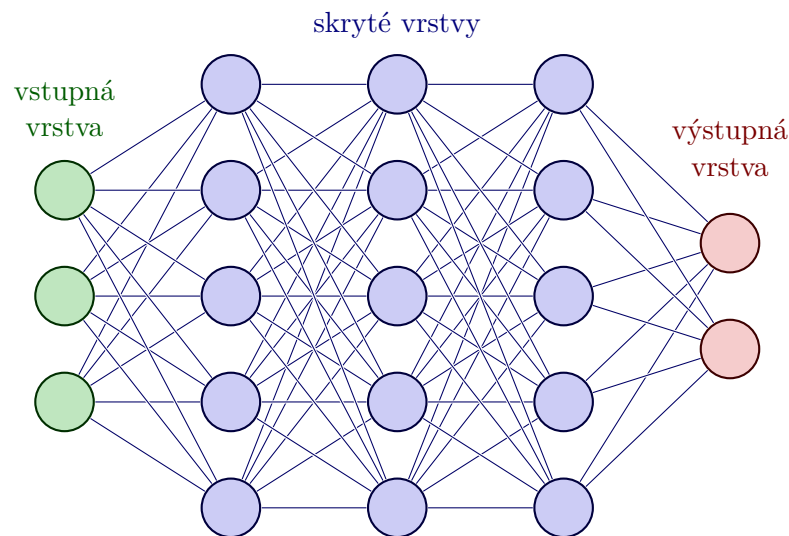
2.2 Hlboké neurónové siete

V tejto sekcii čerpám zo zdrojov [45, 46]. Schopnosti perceptrónu s jednou vrstvou sú obmedzené na riešenie lineárne separovateľných problémov. To znamená, že dokáže korektné klasifikovať dáta len v prípadoch, kde existuje lineárna hranica medzi rôznymi triedami

alebo kategóriami. Pre riešenie zložitejších problémov, kde dáta nie sú lineárne separovateľné alebo kde vzťahy medzi vstupmi a výstupmi sú komplexnejšie, je potrebné použiť sofistikovanejšie modely. Teorém univerzálnej aproximácie [47] je teoretický základom toho, ako fungujú hlboké neurónové siete. Tvrdí, že neurónová sieť s aspoň jednou skrytou vrstvou (angl. hidden layer), ktorá sa skladá z dostatočného množstva umelých neurónov s nelineárnymi aktivačnými funkciami, je schopná aproximovať ľubovoľnú spojitú funkciu s ľubovoľnou presnosťou. Hlboká neurónová sieť (Deep Neural Network, DNN) predstavuje rozšírenie perceptrónu s viacerými vrstvami, ktoré umožňujú modelu učiť sa komplexné vzorce a vytvárať abstraktné reprezentácie vstupov. Hlboké neurónové siete sú schopné reprezentovať nelineárne vzťahy medzi vstupmi a výstupmi, a tým sa stali kľúčovým nástrojom pre riešenie širokej škály problémov v oblasti strojového učenia, presnejšie hlbokého učenia (Deep Learning, DL).

V súčasnej dobe sú hlboké neurónové siete základom pre pokročilé aplikácie v mnohých oblastiach vrátane spracovania prirodzeného jazyka, počítačového videnia, rozpoznávania reči, a ďalších. Pri implementácii hlbokých neurónových sietí sú kľúčové techniky ako napríklad spätné šírenie chyby (backpropagation), ktoré umožňuje efektívne aktualizovať váhy siete na základe chyby výstupu.

Okrem toho, hlboké učenie využíva rôzne typy architektúr sietí, ako sú konvolučné neurónové siete (Convolutional neural network, CNN) pre vizuálne údaje, rekurentné neurónové siete (Recurrent neural network, RNN) pre sekvenčné dáta, a transformer, ktorý sa čoraz viac používa v mnohých oblastiach spracovania prirodzeného jazyka.



Obr. 2.3: Zjednodušená ilustrácia hlbokéj neurónovej siete s tromi skrytými vrstvami.¹

2.2.1 Spätné šírenie

Spätné šírenie (angl. backpropagation) predstavuje kľúčový algoritmus používaný na efektívne tréningovanie hlbokých neurónových sietí. Jeho základ tvorí metóda gradientného zostupu, ktorá sa systematicky aplikuje na aktualizáciu váh v sieti s cieľom minimalizovať chybovosť modelu porovnaním jeho predikcií s referenčnými výstupnými hodnotami. Spätné

¹Vizualizácia neurónovej siete adaptovaná z https://tikz.net/neural_networks/.

šírenie využíva reťazové pravidlo derivácií, ktoré umožňuje efektívne propagovať chyby späť cez sieť a vypočítať gradienty pre každú váhu, čím sa poskytuje presný smer a veľkosť potrebných úprav váh.

Tento proces je výpočtovo intenzívny, pretože vyžaduje výpočet gradientov pre všetky váhy v sieti, čo v prípade veľkých modelov predstavuje obrovské množstvo operácií. Preto je spätné šírenie často implementované s využitím grafických procesorov (GPU), ktoré dokážu výrazne zrýchliť tréning vďaka ich schopnosti paralelne spracovávať veľké množstvá dát. Vďaka tejto schopnosti a pokrokom v oblasti výpočtovej techniky sa spätné šírenie stalo fundamentálnou technikou pre tréning hlbokých neurónových sietí, umožňujúc modelom učiť sa komplexné vzorce a funkcie z rozsiahlych dátových sád.

Kapitola 3

Transformer – model hlbokého učenia

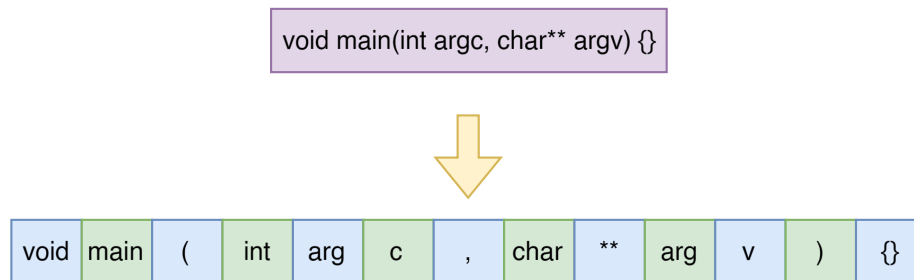
Ako bolo možné vidieť v predchádzajúcich sekciách, základy neurónových sietí a ich schopnosti učiť sa z komplexných dátových súborov formujú elementárne kamene moderného strojového učenia. Postupy ako spätné šírenie a aktivačné funkcie, ktoré umožňujú neurónovým sieťam efektívne sa učiť a adaptovať, sú neoceniteľné pre riešenie rozmanitých problémov v oblasti umelej inteligencie. Hlboké neurónové siete, založené na týchto princípoch, dokázali dosiahnuť významný pokrok v mnohých aplikáciách, vrátane spracovania prirodzeného jazyka.

V oblasti neurónových sietí transformery predstavujú revolučnú metódu spracovania prirodzeného jazyka. Na rozdiel od tradičných architektúr, ako sú RNN alebo CNN, transformery využívajú mechanizmy pozornosti (attention) na efektívne zachytenie komplexných vzťahov v sekvencných dátach. Možnosť každého slova v sekvencii vplývať na ostatné slová v sekvencii umožňuje transformerom excelovať vo vytváraní modelov dlhodobých závislostí, ktoré sú kľúčové pre úlohy ako strojový preklad a generovanie textu. Architektúra kodéru (encoder) a dekodéru (decoder), charakteristická pre transformery, umožňuje spracovanie vstupných sekvencií do abstraktných reprezentácií prostredníctvom kodéru, nasledované generovaním výstupných sekvencií dekodérom. Transformery umožnili vznik rozsiahlych predtrénovaných jazykových modelov, ako je napríklad séria GPT od firmy OpenAI alebo LLAMA od firmy Meta, ktoré preukázali prispôsobivosť pre rôzne oblasti a schopnosť generalizovať naučené vzorce. Schopnosťou prenosu učenia transformery zmenili paradigmu v porozumení prirodzenému jazyku, otvárajúc cestu k inovatívnym aplikáciám v oblasti výskumu a vývoja umelej inteligencie.

V tejto kapitole sa podrobnejšie pozriem na to, ako transformer využíva tieto koncepty a ako sa architektúra transformera stala kľúčovou technológiou pre generatívne modely v oblasti spracovania prirodzeného jazyka (NLP). V tejto kapitole popíšem transformáciu textu, mechanizmus pozornosti, zameriam sa na základné stavebné bloky transformera, ako sú kodér a dekodér, a diskutujem o ich funkcii a prínosoch.

3.1 Tokenizácia

Tokenizácia je proces transformácie textu na sekvenciu tokenov. Tokeny môžu byť jednotlivé slová, písmená alebo podreťazce, ktoré sú použité na reprezentáciu vstupných dát. Dôležitou črtou tokenizácie je schopnosť rozdeliť text na časti, ktoré majú pre model zmysel. To zahŕňa



Obr. 3.1: Proces rozkladu vstupného zdrojového kódu na tokeny. Tokeny môžu byť napríklad kľúčové slová, identifikátory, operátory alebo zátvorky.

rozdelenie textu na jeho zložky tak, aby boli zachované sémantické a syntaktické vlastnosti originálneho textu. Existuje niekoľko prístupov k tokenizácii, ktoré sa odlišujú granularitou a metodikou tokenizácie. Medzi najpoužívanejšie prístupy patria:

- **Tokenizácia na úrovni slov** – Text je rozdelený na jednotlivé slová, ktoré sú tokenmi. Tento prístup je najpoužívanejší v prípade, že slová jazyka sú efektívne separovateľné medzerami.
- **Tokenizácia na úrovni znakov** – Text je rozdelený na jednotlivé znaky, ktoré sú tokenmi. Tento prístup je vhodný pre jazyky, ktoré nemajú jasne definované separátory slov, alebo napríklad pre úlohy, ktoré analyzujú štruktúru textu na úrovni znakov.
- **Tokenizácia na podreťazce** – Spojenie predchádzajúcich prístupov, ktoré rozdeľuje text na podreťazce, ktoré sú často používané v texte. Tento prístup umožňuje modelu pracovať s neznámymi slovami alebo slovami, ktoré nie sú obsiahnuté v slovníku.

3.1.1 BPE tokenizácia

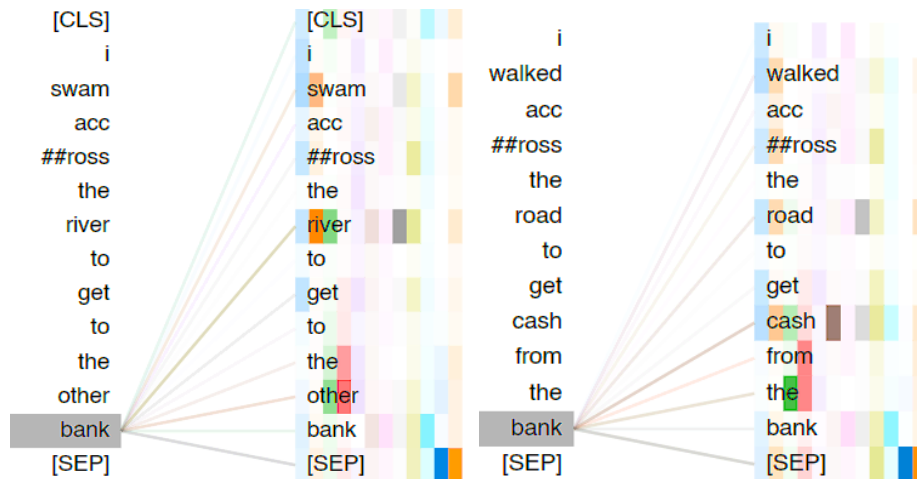
Byte-Pair Encoding (BPE) je pokročilá metóda tokenizácie na podreťazce, ktorá zohráva kľúčovú úlohu NLP, najmä v aplikáciách ako strojový preklad a generovanie textu. BPE funguje na princípe postupného spájania znakov alebo reťazcov znakov na základe frekvencie ich výskytu v dátovej sade, čím sa postupne vytvára optimalizovaný slovník tokenov. Tento proces sa opakuje, kým nie je dosiahnutá požadovaná veľkosť slovníka. Nevýhodou je, že takto naučený model môže mať problém s neznámymi znakmi, ktoré nie sú obsiahnuté v slovníku. Tento problém rieši BYTE-LEVEL BPE tokenizácia, ktorá pracuje na úrovni bytov, na nižšej úrovni reprezentácie textu. Počiatočný slovník tak má vždy veľkosť 256 tokenov, ktoré postupne spája do zložitejších tokenov. Tento prístup umožňuje modelu pracovať s neznámymi znakmi a zároveň zachováva výhody BPE tokenizácie.

Rovnako ako existujú rozdiely medzi prirodzenými a programovacími jazykmi, diskutované v sekcii 4.3, tokenizácia pre programovacie jazyky vyžaduje špecifické úpravy. Je štandardné, keď je tokenizér trénovaný spolu s jazykovým modelom na tréningových dátach z oblasti, v ktorej bude model nasadený. Tento prístup umožňuje tokenizáciu, ktorá je špecifická pre konkrétnu doménu, čo môže viesť k lepším výsledkom.

3.2 Mechanizmus pozornosti

Mechanizmus pozornosti [43] predstavuje fundamentálny koncept v architektúre transformerov a umožňuje efektívne zachytávať komplexné vzájomné väzby v sekvenčných dátach. Neurónové siete typu transformer neimplementujú rekurentné prepojenia, ktoré sú charakteristické pre RNN, ale namiesto toho sa spoliehajú výlučne na mechanizmus pozornosti na zachytenie vzájomných vzťahov dát v sekvencii [26].

Pozornosť umožňuje pre každú pozíciu v sekvencii, token, vypočítať váhy pre ostatné pozície. Tieto váhy kvantifikujú mieru, do akej daná pozícia pri generovaní reprezentácie vplýva na ostatné pozície v sekvencii. Pre ilustráciu toho, na čo slúži pozornosť, uvádzam príklad z knihy [26]. Slovo „bank“, má v anglickom jazyku viacero významov. Jedným môže byť breh rieky, druhým je banka, ako finančná inštitúcia, pričom záleží na kontexte, v ktorom je slovo použité. Na obrázku 3.2 je vidieť, ako vplývajú jednotlivé slová na ostatné v sekvencii. Váhy pozornosti pre slovo "bank" v kontexte „*I swam accross the river to get to the other bank.*“ sú zamerané na slovo „*river*“, zatiaľ čo v kontexte „*I walked accross the road to get cash from the bank.*“ sú zamerané na slovo „*cash*“.



Obr. 3.2: Vizualizácia naučených váh hláv jednej vrstvy mechanizmu pozornosti modelu BERT s využitím nástroja [44].

3.2.1 Samopozornosť

Na výpočet váh pozornosti sa v praxi používa mechanizmus nazývaný samopozornosť (self-attention). Samopozornosť umožňuje modelu vypočítať váhy pozornosti pre každú pozíciu v sekvencii na základe vstupných reprezentácií. V kontexte samopozornosti sa mechanizmus pozornosti aplikuje na tú istú sekvenciu.

Fungovanie mechanizmu samopozornosti v architektúre transformer:

1. Každý vstupný vektor x_i je lineárne transformovaný na tri vektory q_i , k_i a v_i pomocou naučených váhových matic W_Q , W_K a W_V .
2. Pre každú pozíciu x_i v sekvencii sú vypočítané váhy pozornosti a_{ij} pre ostatné pozície x_j v sekvencii na základe vektorov q_i a k_j . Skalárny súčin $q_i \cdot k_j$ je následne upravený

faktorom $\frac{1}{\sqrt{d_k}}$ a normalizovaný pomocou funkcie softmax, ktorá zabezpečuje, že váhy pozornosti sú nezáporné a ich súčet je rovný jednej.

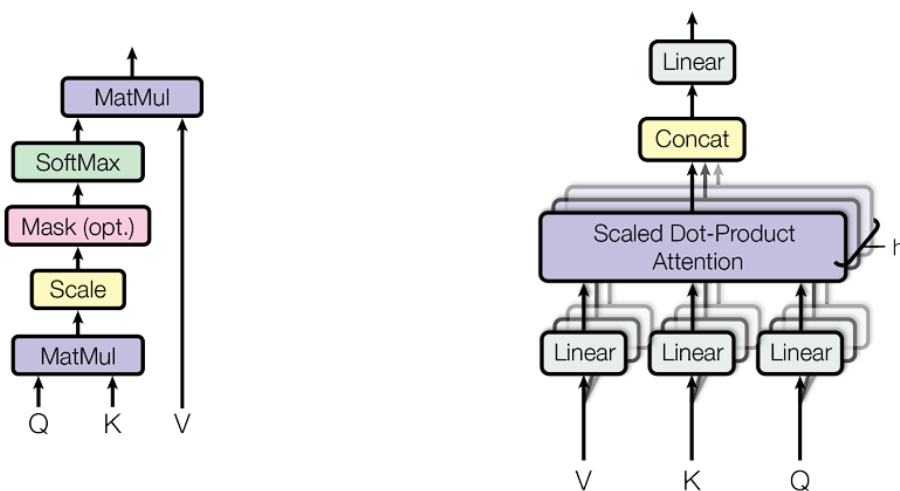
$$a_{ij} = \text{softmax} \left(\frac{q_i \cdot k_j}{\sqrt{d_k}} \right) \quad (3.1)$$

3. Výsledná reprezentácia y_i pre pozíciu x_i je vypočítaná ako vážená suma vstupných reprezentácií x_j na základe váh pozornosti a_{ij} .

$$y_i = \sum_{j=1}^n a_{ij} v_j \quad (3.2)$$

Rovnica 3.2 reprezentuje výpočet výslednej reprezentácie y_i pre pozíciu x_i ako váženú sumu vstupných reprezentácií x_j , kde váhy pozornosti a_{ij} určujú príspevok každej vstupnej reprezentácie x_j k výslednej reprezentácii y_i . Takto spočítaná pozornosť sa nazýva pozornosť skalárneho súčinu (scaled dot-product attention), pretože váhy pozornosti sú vypočítané ako skalárny súčin vektorov q_i a k_j upravený faktorom $\frac{1}{\sqrt{d_k}}$.

Jedným z významných prínosov pozornosti je možnosť interpretovať rozhodnutia modelu týkajúce sa dôležitosti rôznych častí vstupných dát. To znamená, že model môže poskytnúť vysvetlenia, prečo určité časti vstupných dát ovplyvňujú jeho rozhodnutia, čo je kritické pre aplikácie, ktoré vyžadujú transparentnosť a interpretovateľnosť. V praxi sa pozornosť v transformeroch používa v rôznych častiach modelu a hrá kľúčovú úlohu vo vytváraní presných a výkonných modelov spracovania prirodzeného jazyka. Mechanizmus samopozornosti sa tak stáva revolučným nástrojom v oblasti spracovania prirodzeného jazyka a umožňuje dosahovať mimoriadne výsledky v rôznych úlohách.



Obr. 3.3: Pozornosť skalárneho súčinu (vľavo) a multi-head pozornosť (vpravo). Prebrané z [43].

3.2.2 Multi-head pozornosť

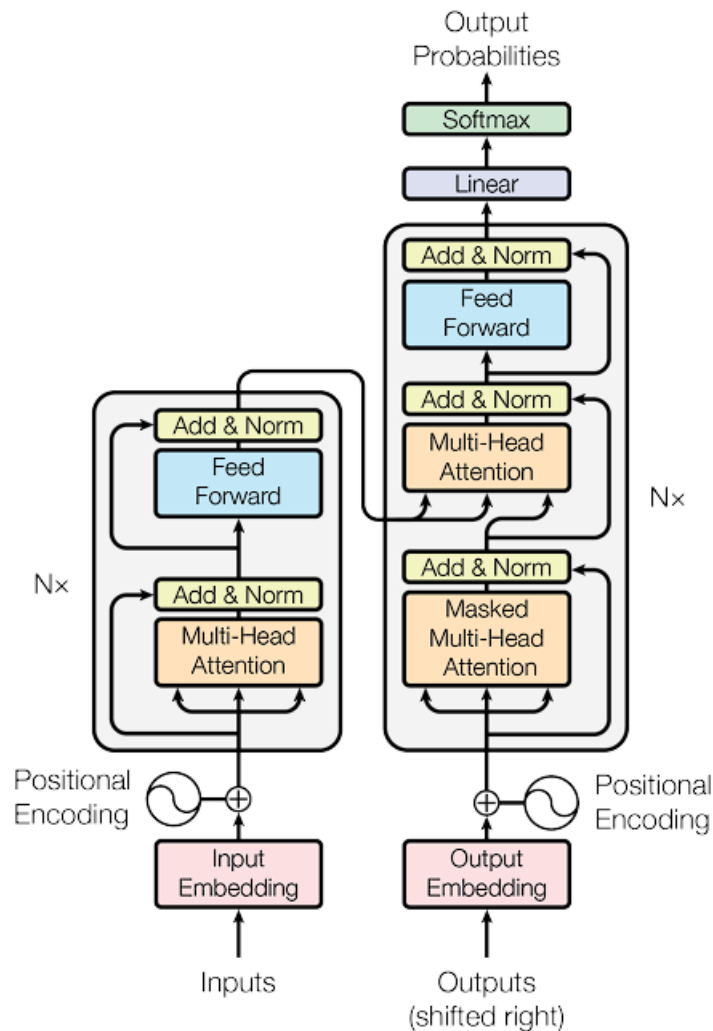
Na obrázku 3.3 vidieť, že takto navrhnutých „hláv“ pozornosti sa využíva niekoľko paralelne poskladaných. Tento prístup, nazývaný multi-head pozornosť (multi-head attention, MHA),

umožňuje modelu zachytiť rôzne aspekty vstupných dát a vytvoriť bohatšie reprezentácie, ktoré sú schopné zachytiť zložité vzťahy v dátach. Každá hlava pozornosti sa naučí vytvárať rôzne reprezentácie vstupných dát a následne sú tieto reprezentácie spojené do jednej výslednej reprezentácie. Výsledná rovnica pre Multi-head pozornosť je definovaná ako:

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O \quad (3.3)$$

3.3 Architektúra transformer

V tejto podkapitole je popísaná architektúra transformer, pričom sa čerpá z práce [43]. Architektúra transformer je založená na princípe kodéru a dekodéru. Kodér aj dekodér sa skladajú z niekoľkých na sebe naskladaných identických vrstiev. Kodér je zodpovedný za reprezentáciu vstupných dát, zatiaľ čo dekodér generuje výstupné dáta na základe reprezentácií vytvorených kodérom a výstupov z predchádzajúcich vrstiev dekodéra. Architektúra transformer je zobrazená na obrázku 3.4.



Obr. 3.4: Architektúra transformer s N kodér a dekodér blokmi. Prebrané z [43].

3.3.1 Kodér

Kodér je navrhnutý tak, aby spracoval vstupnú sekvenciu a transformoval ju na sériu vektorových reprezentácií, ktoré obsahujú kontextové informácie z celého vstupu. Kontextové vektory sú následne použité dekodérom na generovanie výstupnej sekvencie.

Popis fungovania kodéru:

1. Vstupná sekvencia tokenov do kodéru je najprv prevedená na spojité vektorové reprezentácie, vrstva vnorenia slov (input embedding), následne sú pripočítané informácie o poradí jednotlivých tokenov v sekvencii, pozičné kódovanie (positional encoding). Po inicializácii vstupných vektorových reprezentácií nasleduje spracovávanie sériou N vrstiev kodéru.
2. Každá vrstva kodéru sa skladá z dvoch podvrstiev. Prvou je MHA a druhou je plne pozičná plne prepojená dopredná sieť (position-wise fully connected feed-forward network, FFN), ktorá pozostáva z dvoch lineárnych transformácií, aplikovaných na každú pozíciu sekvencie, a nelineárnej ReLU aktivačnej funkcie.

$$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2 \quad (3.4)$$

MHA umožňuje kodéru analyzovať vstupné dáta z rôznych kontextových perspektív, zatiaľ čo dopredná sieť ďalej spracováva výstup mechanizmu pozornosti na generovanie výslednej reprezentácie pre každý token v sekvencii.

3. Po každej podvrstve MHA a FFN sú pridané reziduálne spojenia, nasledované normalizačnou vrstvou¹. Tieto komponenty pomáhajú zabrániť problémom s miznúcim alebo explodujúcim gradientom a zlepšujú stabilitu učenia.

3.3.2 Dekodér

Dekodér sa skladá zo série identických vrstiev, kde každá vrstva prijíma vstupy z predchádzajúcej vrstvy a poskytuje vstupy pre nasledujúcu vrstvu. Tento proces sa opakuje, až kým nie je dosiahnutý požadovaný výstup alebo dosiahnutá maximálna dĺžka sekvencie.

Dekodér má podobnú štruktúru ako kodér, ale obsahuje aj zásadné zmeny:

1. Podobne ako pre kodér platí, že vstup do dekodéra je najprv transformovaný do vektorového priestoru a skombinovaný s pozičným kódovaním. Vstupy do kodéru sú obvykle výstupné tokeny, ktoré model generoval v predchádzajúcich krokoch (autoregresívny prístup).
2. Prvá MHA vrstva je upravená, aby maskovala budúce informácie, čo znamená, že pri generovaní reprezentácie pre pozíciu x_i v sekvencii dekodér nemá prístup k informáciám z pozícií x_j pre $j > i$. Táto úprava spolu s posunutím vstupných reprezentácií do dekodéra zabezpečujú, že dekodér generuje reprezentácie iba na základe informácií na pozíciách menších ako i .
3. Stavba dekodéra je rozšírená o ďalší mechanizmus pozornosti, ktorý umožňuje dekodéru pozeráť sa na výstupné reprezentácie vytvorené kodérom. Tento mechanizmus,

¹Na rozdiel od pôvodnej architektúry platí, že v súčasných modeloch sa vykonáva normalizácia pred vrstvami MHA a FFN. Zistilo sa totiž, že pre modely, ktoré majú 10 a viac vrstiev, sa stáva učenie nestabilné [40].

nazývaný kodér-dekodér pozornosť, umožňuje dekodéru prístup ku kompletnej reprezentácii vstupných dát z kodéru a ich využitie pri generovaní výstupných reprezentácií.

4. Výstupy sú nakoniec prevedené cez lineárnu vrstvu a softmax funkciu, ktorá generuje pravdepodobnosti nasledujúcich tokenov v sekvencii.

Kapitola 4

Spracovanie prirodzeného jazyka

Spracovanie prirodzeného jazyka (Natural Language Processing, NLP) je disciplína umelej inteligencie a lingvistiky, ktorá sa zaoberá spracovaním a analýzou ľudského (prirodzeného) jazyka. Zahŕňa vývoj algoritmov a modelov s cieľom umožniť výpočtovým systémom porozumieť, interpretovať a generovať ľudský jazyk. Tieto systémy môžu byť aplikované na rôzne úlohy, ako sú rozpoznávanie reči, strojový preklad, analýza sentimentu, generovanie textu a extrakcia informácií.

NLP nie je novým konceptom; jeho korene siahajú už do 40. rokov 20. storočia. V roku 1950 publikoval Alan Turing vedecký článok, ktorý sa zaoberal otázkou, či stroj môže myslieť [42]. Tento článok je považovaný za jeden z prvých príspevkov k oblasti umelej inteligencie a NLP. Mimo iného sa v článku zaoberal aj otázkou, či by mohol stroj porozumieť a generovať ľudský jazyk, respektíve navrhol hru, ktorá sa stala známou ako Turingov test. Tento test sa stal kľúčovým v oblasti NLP a umelej inteligencie, pretože sa zaoberá otázkou, či je počítač schopný komunikovať s človekom tak, aby človek nebol schopný rozlíšiť, či komunikuje s počítačom alebo s iným človekom. Turingov test je dnes už všeobecne považovaný za nedostatočný pre posúdenie inteligencie, ale výzvy spojené s nejednoznačnosťou a zložitosťou prirodzeného jazyka pretrvávajú.

Slová a frázy často majú viacero významov v závislosti od kontextu, čo robí ťažké pre počítače presne porozumieť a spracovať jazykové údaje. Okrem toho jazyky prejavujú syntaktické a sémantické varianty, čo viac komplikuje úlohu NLP. S nástupom strojového učenia a neskôr hlbokého učenia (deep learning) došlo k dramatickému posunu v schopnostiach NLP. Modely založené na strojovom učení sa učia zo vzorov v dátach, čo umožňuje adaptáciu na nové jazykové konštrukcie bez potreby explicitného programovania každej možnej situácie. Hlboké učenie, a najmä neurónové siete, posunuli tieto schopnosti ďalej, umožňujúc modelom učiť sa komplexné vzorce a nuansy jazyka z obrovských množstiev textových dát. Na riešenie týchto výziev vznikli sofistikované modely ako rekurentné neurónové siete, siete s dlhou krátkodobou pamäťou (long short-term memory network, LSTMs [16]) a v posledných rokoch veľmi úspešné modely založené na architektúre transformer. Tieto modely využívajú veľké dátové sady a pokročilé tréningové techniky na naučenie sa komplexných vzorov v jazykových údajoch.

4.1 Veľké jazykové modely

Veľký jazykový model (large language model, LLM) je označenie pre model hlbokého učenia, ktorého počet parametrov, resp. váh, sa pohybuje v rádoch miliárd, pričom počet pa-

rametrov modelu je kľúčovým faktorom výkonu. Po predtrénovaní pod vlastným dohľadom (self-supervised learning) na rozsiahlom korpuse textu jazykové modely dosahujú nevídané schopnosti zovšeobecnenia pri úlohách porozumenia a generovania textu. Pri prispôbení na nadväzujúce úlohy predtrénované jazykové modely prekonávajú modely, ktoré sú od začiatku tréované pre danú doménu. Okrem toho, veľké jazykové modely vykazujú schopnosť zapamätať si a využívať informácie načerpané počas fázy učenia. Toto zahŕňa nielen jazykové štruktúry a gramatiku, ale aj obecné informácie a znalosti o svete zahrnuté v tréovacích dátach. Tieto vlastnosti jazykových modelov podnietili výskum a tréovanie väčších modelov na väčších súboroch dát. To malo za následok zistenie, že zväčšovanie veľkosti modelu a súboru údajov zvyšuje schopnosť modelov generalizovať [18] a vznikajú modely o veľkosti desiatok až stoviek miliárd parametrov. Modelovanie jazyka výrazne pokročilo, keďže predtým menšie modely nemohli dosiahnuť takéto zovšeobecnenie. V dôsledku záujmu a nadšenia vedeckej komunity o zdokonaľovanie návrhov a metód tréovania LLM bolo v posledných rokoch vyvinutých veľké množstvo jazykových modelov [41, 8, 6, 49, 4].

Tieto veľké jazykové modely majú spoločnú charakteristiku v tom, že na rozdiel od starších modelov založených na konvolučných alebo rekurentných kodér-dekodér sieťach sú založené výlučne na dekodér architektúre transformer.

4.2 Generatívny predtrénovaný transformer

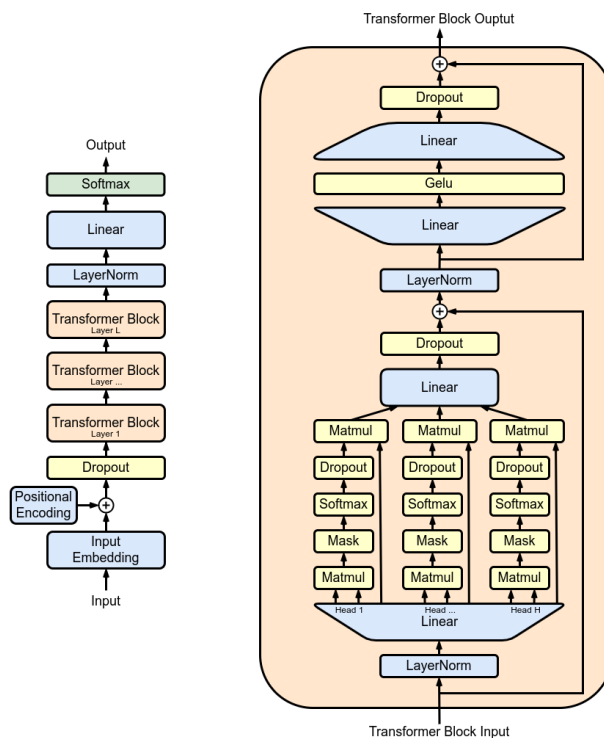
Aj keď pôvodnú kodér-dekodér architektúru transformera adaptovalo množstvo jazykových modelov [34, 21] a stále nájde svoje uplatnenie, v posledných rokoch sa pozornosť výrazne presunula smerom k modelom, ktoré využívajú výlučne dekodér bloky. Táto modifikácia bola prvýkrát predstavená v práci [24].

Generatívny predtrénovaný transformer (Generative Pre-trained Transformer, GPT) je označenie pre celú sériu jazykových modelov, z ktorých prvý bol GPT [33] od firmy OpenAI. Tieto modely sú tréované na veľkých množstvách textových dát pomocou metódy učenia pod vlastným dohľadom, kedy sa model učí predikovať nasledujúce tokeny v sekvencii na základe predchádzajúcich tokenov. Ich výhodou v porovnaní s kodér-dekodér modelmi je jednoduchosť architektúry, keďže dekodér blok spája porozumenie jazyka s generovaním textu do jedného modulu, a flexibilita a generalizácia na rôzne úlohy spracovania prirodzeného jazyka.

Model GPT-4 je významným krokom vpred v oblasti generatívnych jazykových modelov, ktorý posúva hranice použiteľnosti. S potenciálnou veľkosťou až 1,76 bilióna parametrov [38], čo predstavuje desaťnásobok veľkosti jeho predchodcu GPT-3 so 175 miliardami parametrov [6], GPT-4 predstavuje exponenciálny nárast v kapacite učenia a spracovania jazyka. Tento nárast v počte parametrov umožňuje GPT-4 lepšie zachytiť nuansy ľudského jazyka a zlepšiť pochopenie kontextu. Vďaka tomu dokáže generovať presnejšie a relevantnejšie odpovede na zložité otázky. Taktiež to prinieslo pokrok v schopnosti poskytnúť vysvetlenia pre svoje rozhodnutia.

4.3 Generovanie zdrojového kódu

V oblasti informačných technológií predstavuje generovanie zdrojového kódu spojenie ľudskej kreativity s presnosťou a automatizáciou, ktorú poskytujú stroje. Proces transformácie myšlienok vyjadrených v prirodzenom jazyku do presných, vykonateľných programovacích inštrukcií otvára nové možnosti vo vývoji softvéru.



Obr. 4.1: Architektúra GPT. Prebrané z [25].

Schopnosť generovať kód možno chápať ako druh generovania prirodzeného jazyka. Tento proces možno konkrétne popísať ako preklad z prirodzeného jazyka do programovacieho jazyka (Natural Language- Programming Language, NL- PL) a naopak. Generovanie kódu, ako forma prekladu, nie je iba jednoduché mapovanie slov medzi dvoma jazykmi; ide o premostenie rozdielu v logike, štruktúre a význame, ktoré tieto dva jazyky odlišujú.

Konceptuálny skok od prirodzeného jazyka k jazyku programovania a naopak je zložitý, pretože oba jazyky slúžia veľmi odlišným účelom. Prirodzený jazyk je flexibilný, bohatý a umožňuje veľkú variabilitu. Naproti tomu programovací jazyk je definovaný svojou štruktúrou a predvídateľnosťou.

Podľa práce [35] sú hlavné rozdiely medzi prirodzeným jazykom a programovacím jazykom nasledovné:

- **Obmedzené kľúčové slová vs. milióny slov:** Na rozdiel od prirodzených jazykov s veľkou slovnou zásobou, ktoré sa vyvíjali dlhú dobu a menili sa v priebehu rokov, sú programovacie jazyky navrhnuté ľuďmi a používajú malý počet kľúčových slov. Tieto kľúčové slová sú pre programovacie jazyky kritické, definujú ich syntax a sémantiku, a preto by im mala byť venovaná zvýšená pozornosť.
- **Stromová štruktúra vs. sekvenčná štruktúra:** Prirodzené jazyky sú zvyčajne písané a čítané sekvenčne, zatiaľ čo programovacie jazyky sú reprezentované stromovou štruktúrou. Táto štruktúra je dôležitá pre správne vykonanie programu, a preto je dôležité, aby generovaný kód bol syntakticky správny.
- **Unikátne inštrukcie vs. nejednoznačná sémantika:** Prirodzené jazyky majú často nejednoznačnú a premennú sémantiku. Naproti tomu programovací jazyk vyža-

duje štandardizáciu a systematickosť v definícii inštrukcií. Každý výraz alebo príkaz v programovacom jazyku má presne definovanú funkcionálnosť, čo eliminuje nejednoznačnosť a zaisťuje, že program bude vykonaný tak, ako bol zamýšľaný. Tento prístup znižuje možnosť výkladových chýb, ktoré sú v prirodzených jazykoch pomerne bežné v dôsledku ich inherentnej nejednoznačnosti.

Pokiaľ ide o výzvy a príležitosti, ktoré generovanie zdrojového kódu prináša, existujú rôzne aspekty, na ktoré sa treba sústrediť. Jedným z hlavných aspektov je zlepšenie presnosti a účinnosti tohto procesu. Jazykové modely musia byť schopné pochopiť nielen textový opis funkcie alebo požiadavky, ale aj kontext, v ktorom bude kód použitý. To vyžaduje pokročilé metódy spracovania prirodzeného jazyka a schopnosť analyzovať a interpretovať veľké množstvo dát.

Kapitola 5

Trénovanie veľkých jazykových modelov

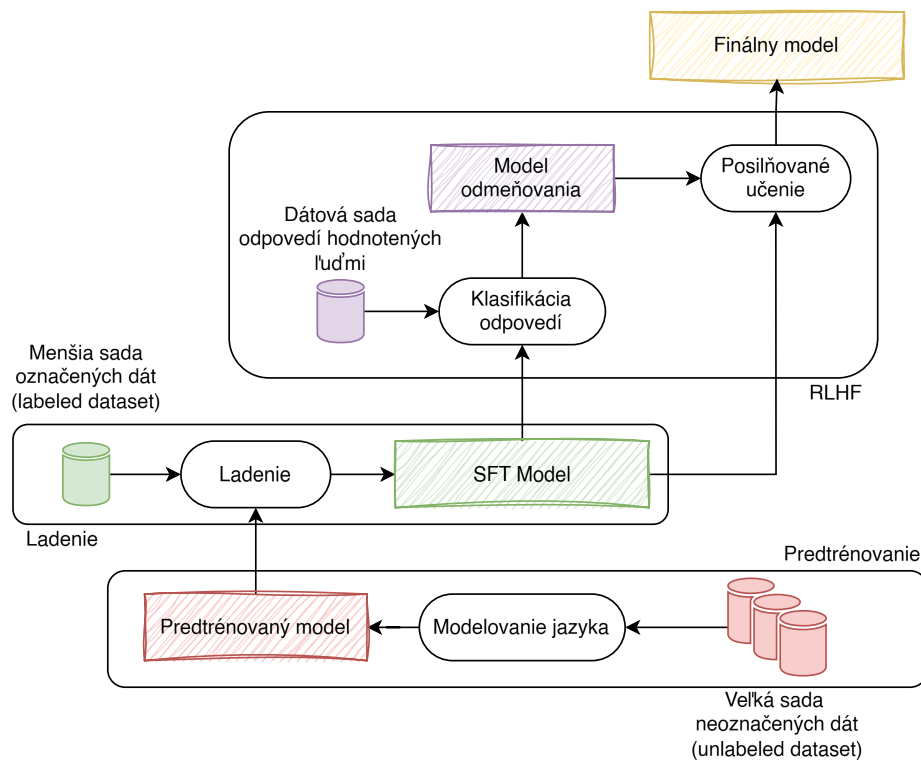
Trénovanie jazykových modelov môže byť založené na rôznych prístupoch a zahŕňať niekoľko fáz, tak ako je vidieť na obrázku 5.1. Medzi prístupy učenia patrí učenie pod dohľadom (angl. supervised learning), učenie bez dohľadu (angl. unsupervised learning) a posilňované učenie (angl. reinforcement learning). Pri učení pod dohľadom model využíva označené dáta, kde každý vstup je priradený k preddefinovanému výstupu. Učenie bez dohľadu pracuje s neoznačenými dátami a model sa snaží nájsť štruktúru alebo vzory v dátach bez vonkajších označení. Posilňované učenie, ktoré je často využívané pri veľkých konverzačných modeloch, umožňuje modelu učiť sa z vlastných chýb prostredníctvom systému odmien a trestov. Model je pri tomto spôsobe odmeňovaný za správne reakcie a trestaný za chyby, čo ho motivuje k neustálemu zlepšovaniu.

Trénovanie modelov, ktoré obsahujú veľké množstvo parametrov, vyžaduje rozsiahle dátové sady a významné výpočtové zdroje, najmä v podobe výkonných grafických kariet. To predstavuje jednu z hlavných výziev v oblasti vývoja umelej inteligencie, keďže náklady na výpočtové zdroje môžu byť značné.

Techniky efektívneho ladenia parametrov (Parameter-Efficient Fine-Tuning, PEFT) poskytujú riešenie týchto výziev tým, že umožňujú trénovanie veľkých jazykových modelov s menšími výpočtovými nárokmi. Tento prístup dosahuje úspory zmenšením počtu priamo trénovaných parametrov alebo aplikovaním techník, ako je kvantizácia váh modelu. V tejto kapitole sa venujem technikám trénovania veľkých jazykových modelov. Rozoberiem jemné ladenie a ďalšie metódy, ktoré umožňujú efektívne trénovanie modelov pri nižších výpočtových nárokoch.

5.1 Predtrénovanie

Modely, ako je GPT, sú predtrénované na rozsiahlych sadách, ktoré zahŕňajú knihy, články a webové stránky, aby poskytli komplexný jazykový základ. Architektúra modelu, najmä mechanizmus samopozornosti, o ktorom sa hovorilo v sekcii 3.2.1, umožňuje modelom zachytiť zložité vzory v jazyku, čo umožňuje vytvárať presné a sémanticky bohaté reprezentácie textu.



Obr. 5.1: Trénovanie modelu prebieha v niekoľkých fázach. Obvykle začína predtrénovanie, nasledované jemným ladením a posilňovaným učením modelu. Vo fázach sú potrebné rôzne dáta, z ktorým sa model učí.

Príčinné modelovanie jazyka

Modely založené na architektúre dekodéru, ako GPT alebo LLAMA, používajú metódu autoregresie. Inými slovami, model predikuje ďalší token v sekvencii na základe predchádzajúcich stavov, smerov zľava-doprava, a porovnáva ho s referenčným riešením.

Maskované modelovanie jazyka

Táto techniku bola použitá pri trévaní kodér modelu BERT [11]. Jej podstatou je náhodné maskovanie časti vstupu a následné generovanie tokenov. Za správne riešenie sú považované maskované časti. Tento prístup umožňuje modelu učiť sa kontextovú reprezentáciu slov a ich vzťahy v sekvencii.

Vyplňovanie na základe okolia

Autoregresívne dekodér modely, ktoré sú v súčasnosti prevažujúcou technológiou v modelovaní jazyka pre generovanie ďalšieho slova v sekvencii, pracujú výlučne s kontextom na ľavej strane (prefix). Úloha vyplňovania na základe okolia (Fill in the Middle, FIM) [13, 50, 5] predstavuje špecifický druh modelovania jazyka, ktorý umožňuje dekodér modelom generovať sekvencie so zreteľom na kontext umiestnený na oboch stranách. Prístup FIM dovoľuje modelom efektívne využívať informácie z oboch strán sekvencie tým, že vykonáva permutáciu podsekvencií, čím model dostane informácie z okolia miesta a následne generuje strednú časť. Umožňuje tak jazykovým modelom pre generovanie kódu napríklad predikovať názvy

funkcií na základe kontextu v tele funkcie, písať komentáre na základe kódu alebo doplniť chýbajúce časti kódu na základe kontextu v okolí.

5.2 Ladenie

Po predtrénovaní sú modely ladené na menšej úlohe, špecifickejšej kolekcii dát. Táto fáza prispôsobuje široko naučené vzorce špecifickým aplikáciám. Ladenie vyžaduje označené údaje a prebieha pod dohľadom (supervised learning), čo znamená, že vstupný text je spojený so správnymi výstupmi alebo označeniami klasifikácie. Počas tejto fázy dochádza ku gradientovým úpravám všetkých váh modelu za účelom minimalizácie rozdielu medzi predpoveďami a skutočnými označeniami, čím sa zlepšuje presnosť na špecifickú úlohu.

Ladenie umožňuje prispôbienie všeobecne predtrénovaného modelu širokej škále aplikácií, využívajúc komplexné porozumenie získané počas predtrénovania a aplikujúc ho na konkrétne úlohy. Tento prístup výrazne znižuje čas a zdroje potrebné na vývoj vysoko výkonných modelov pre špecifické aplikácie.

5.2.1 Efektívne ladenie parametrov

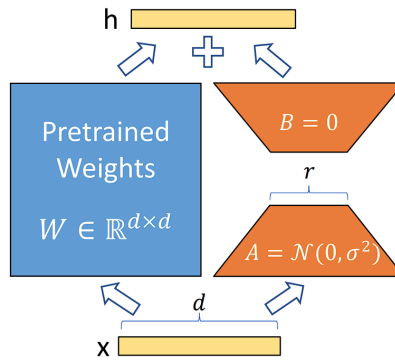
Jazykové modely sa v čase zlepšujú, ale to prichádza aj so stále sa zväčšujúcim počtom trénovateľných váh, ktoré sa v súčasnosti pohybujú v desiatkach až stovkách miliárd, GPT-3 má 175 miliárd parametrov [6], PaLM model až 540 miliárd [8]. Úplné ladenie váh predtrénovaných modelov sa so zväčšujúcimi modelmi stáva čoraz náročnejším. Z tohto dôvodu bol vyvinutý koncept PEFT, ktorý umožňuje efektívnejšie využitie počtu parametrov. Táto technika sa sústreďuje na optimalizáciu malého počtu kľúčových parametrov namiesto upravovania celého modelu, čím sa výrazne znižujú nároky na výpočtové zdroje a čas potrebný pre ladenie.

PEFT využíva metódy ako napríklad selektívne trénovanie iba určitých vrstiev modelu alebo zavedenie nových, špecificky nastaviteľných parametrov, ktoré sa dajú efektívnejšie trénovať v kontexte konkrétnej úlohy. Nasadenie techniky PEFT tak prináša značné výhody vo flexibilitate, efektivitave a škálovateľnosti ladenia jazykových modelov, čo je kľúčové pre ich široké využitie v rôznych aplikáciách a pri rýchlo meniacich sa požiadavkách.

5.2.2 Kvantizácia váh modelu

Kvantizácia váh modelu [36] je sofistikovaný proces, ktorý umožňuje preklenúť medzeru medzi presnosťou spojitého priestoru a praktickou potrebou kompaktnej reprezentácie. Prostredníctvom kvantizácie je možné váhy neurónových sietí, pôvodne reprezentované s vysokou presnosťou v spojitom rozsahu, transformovať na hodnoty s nižšou bitovou hĺbkou reprezentované v diskretnom priestore. Tento proces sa vykonáva aplikáciou kvantizačnej funkcie, ktorá priradí každej pôvodnej hodnote najbližšiu hodnotu z preddefinovaného súboru kvantizačných úrovní. Napríklad pri kvantizácii váh modelu z pôvodnej reprezentácie 32-bitového čísla s pohyblivou desatinnou čiarkou na 8-bitové celé čísla je prechod z rozsahu $[-3.4 \times 10^{38}, 3.4 \times 10^{38}]$ na rozsah $[-128, 127]$. Tento proces umožňuje zmenšiť veľkosť modelu a zvýšiť efektivitu pri výpočtoch, ale za cenu straty presnosti.

V kontexte trénovania jazykových modelov sa kvantizácia využíva na zmenšenie počtu parametrov modelu, čím sa znižujú výpočtové a priestorové nároky potrebné na trénovanie a nasadenie modelu. Tento prístup umožňuje efektívne trénovanie veľkých modelov na za-



Obr. 5.2: Váhy matice A sú inicializované náhodne Gaussovým rozdelením. Matica B je inicializovaná nulami, takže matica $\Delta W = BA$ je na začiatku tréningu nulová. Počas tréningu sa A a B upravujú pomocou gradientných metód, čím sa model adaptuje na špecifickú úlohu. Prebrané z [17].

riadeniach s obmedzenými výpočtovými zdrojmi, čo je kľúčové pre ich nasadenie v reálnom svete.

5.2.3 Adaptácia nízkorozmerných reprezentácií

Adaptácia nízkorozmerných reprezentácií (Low-Rank Adaptation, LoRA) čerpá inšpiráciu z prác, ktoré ukázali, že modely s nadmerným počtom parametrov (over-parametrized models) v skutočnosti existujú v priestore s nízkou vnútornou dimenziou. To znamená, že aj keď modely majú veľký počet parametrov, ich efektívne správanie a schopnosť generalizácie môže byť opísané podstatne menším počtom relevantných dimenzií [17].

Na základe tohto pozorovania autori formulovali hypotézu, že zmeny vo váhach modelu počas jeho adaptácie (napríklad pri ladení na špecifickú úlohu) majú nízky vnútorný rozmer alebo „intrinsic rank“. Inak povedané, predpokladali, že aj keď sa počas adaptácie modelu upravuje veľký počet váh, skutočná efektívna zmena, ktorá sa deje v modeli, môže byť opísaná menším počtom kľúčových dimenzií alebo parametrov.

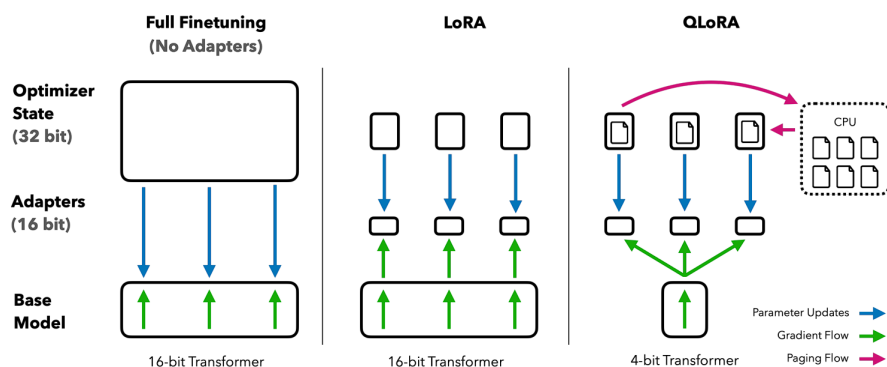
Ako je znázornené na obrázku 5.2, predtrénovanú váhová matica ($W_0 \in \mathbb{R}^{d \times k}$) sa aktualizuje pomocou nízkorozmernej dekompozície $W_0 + \Delta W = W_0 + BA$, kde $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, a hodnota r je menšia ako $\min(d, k)$. Týmto prístupom sa udržiavajú pôvodné váhy W_0 fixné, zatiaľ čo A a B sú trénovateľné. To umožňuje modelu adaptovať sa na nové úlohy bez potreby zásadných zmien v jeho štruktúre.

Pri tomto procese W_0 a $\Delta W = BA$ pracujú na rovnakom vstupe a ich výsledky sú sčítané podľa súradníc. Takto upravený dopredný prechod pre $h = W_0x$ poskytuje výsledok:

$$h = W_0x + \Delta Wx = W_0x + BAx$$

5.2.4 QLoRA

Technika QLoRA (Quantized Low-Rank Adaptation) [10] kombinuje kvantizáciu váh modelu s LoRA, čím umožňuje efektívne tréningovanie rozsiahlych modelov. Najprv sa váhy modelu kvantizujú na nižšiu bitovú hĺbku, po ktorej sa pridávajú nízkorozmerné dekompozície A a B . Váhy v maticiach A a B sú v 16-bitovej presnosti. Tieto reprezentácie sú následne



Obr. 5.3: Porovnanie prístupu plného ladenia všetkých váh, techniky LoRA a techniky QLoRA. Prebrané z [10].

trénované pomocou spätného šírenia chýb cez kvantizované váhy modelu. Autori práce predstavili stránkovaný optimalizátor (Paged Optimizer), ktorý využíva stránkovanie CPU a aktivuje sa, keď sa vyčerpá pamäť GPU. Keď sa tak stane, pamäť sa presúva stránka po stránke z GPU na CPU, čo zabezpečuje efektívnu správu pamäťových požiadavkov. Týmto spôsobom sa zmierni vplyv rýchlych pamäťových nárastov pri ukladaní kontrolných bodov gradientov (angl. gradient checkpointing), ktoré sa vyskytujú pri spracovaní dávky s väčšou dĺžkou sekvencie.

5.3 Evaluačné metriky

Pri vývoji modelov pre generovanie kódu je kľúčovým aspektom vývoja ich evaluácia. Evaluácia modelov v tejto oblasti sa líši od tradičného hodnotenia modelov strojového učenia. Kód, na rozdiel od prirodzeného jazyka, vyžaduje syntaktickú presnosť, sémantickú správnosť a funkčnú vykonateľnosť. Bežné metriky z oblasti prirodzeného jazyka zanedbávajú dôležité sémantické a syntaktické vlastnosti kódu alebo podceňujú syntakticky odlišné výstupy s rovnakou sémantickou logikou.

V nasledujúcich sekciách predstavím 3 rôzne prístupy pre evaluáciu modelov pre generovanie kódu. Prvý prístup predstavujú metriky, ktoré boli pôvodne navrhnuté pre hodnotenie strojového prekladu, druhý prístup predstavuje metriky, ktoré boli navrhnuté špecificky pre hodnotenie generovania kódu. Tieto dva prístupy porovnávajú generovaný kód s referenčným kódom, prípadne viacerými variantami, a hodnotia syntaktickú a sémantickú podobnosť medzi nimi. Tretí prístup predstavujú metriky založené na vykonávaní generovaného kódu a hodnotení jeho správnosti sadou preddefinovaných testov.

5.3.1 BLEU

BLEU (Bilingual Evaluation Understudy) [29] predstavuje algoritmus pre evaluáciu kvality strojovo preložených textov. Kritériom kvality je presnosť (angl. precision¹), alebo miera zhody, medzi výstupom strojového prekladu a prekladom vykonaným človekom.

¹V slovenčine je termín „presnosť“ používaný aj pre anglický termín „accuracy“, čo môže viesť k nejasnostiam. Pre „accuracy“ je vhodnejší preklad „správnosť“.

Myšlienka BLEU je založená na výpočte presnosti n-gramov². Rovnica 5.1 definuje presnosť, ktorá sa počíta ako podiel počtu zhodných slov v preklade a referenčnom preklade (True Positive, TP) a počtu všetkých slov v strojovom preklade, ktoré ale môžu obsahovať aj slová nevyskytujúce sa v referenčnom preklade (False Positive, FP).

$$Precision = \frac{TP}{TP + FP} \quad (5.1)$$

Tu treba poznamenať, že napríklad pre strojový preklad `sum sum sum sum sum` a referenčný preklad `sum = a + b`, by bol rovný 1. Tento prístup je zjavne nedostačujúci, a tak BLEU zavádza modifikovanú presnosť n-gramov. Modifikácia je transformovaná do obmedzenia počtu možných zhôd daného slova podľa toho, koľkokrát sa vyskytuje v referenčnom preklade. Inými slovami, ak sa slovo `sum` vyskytuje v referenčnom preklade iba raz, tak počet zhôd tohto slova môže byť najviac 1. Pre predošlý príklad je výsledok modifikovanej presnosti unigramov rovný 1/5. Skóre BLEU je váženým súčtom týchto modifikovaných zhôd pre n-gramy. Dlhšie n-gramy sú zvyčajne presnejšie, pretože sú menej pravdepodobné, že sa vyskytnú náhodne. Preto BLEU zavádza váhy pre n-gramy, ktoré zohľadňujú ich dĺžku.

Dlhšie vygenerované sekvencie ako referenčné sú z podstaty výpočtu penalizované. To neplatí pre sekvencie, ktoré sú kratšie ako referenčné. Preto BLEU zavádza penalizáciu krátkych sekvencií (brevity penalty). Výsledné skóre BLEU je definované v rovnici 5.2.

$$BLEU = BP \cdot \exp \left(\sum_{n=1}^N w_n \log p_n \right), \quad (5.2)$$

kde BP je brevity penalty, w_n je váha n-gramu, p_n je modifikovaná presnosť n-gramu a N je maximálna dĺžka n-gramu.

5.3.2 CodeBLEU

Tak ako bolo spomenuté v podkapitole 4.3, medzi prirodzenými jazykmi a programovacími jazykmi existujú zásadné rozdiely, ktoré znamenajú, že metriky navrhnuté pre hodnotenie kvality prekladu medzi prirodzenými jazykmi nemusia byť vhodné pre hodnotenie generovania kódu. Preto boli navrhnuté metriky, ktoré sú špecificky prispôbené pre hodnotenie generovania kódu. Jednou z týchto metrick je CodeBLEU [35], ktorá je založená na metrike BLEU, ale bola upravená tak, aby zohľadňovala špecifiká programovacích jazykov.

Metrika CodeBLEU pridáva do výpočtu tri nové faktory a to:

1. výpočet presnosti n-gramov s vyššími váhami pre kľúčové slová programovacieho jazyka,
2. abstraktné syntaktické stromy porovnávaných kódov,
3. Orientované grafy pre reprezentáciu toku hodnôt premenných v kóde.

Výsledné skóre je váženým súčtom týchto faktorov a pôvodného skóre BLEU :

$$CodeBLEU = \alpha \cdot BLEU + \beta \cdot BLEU_{weight} + \gamma \cdot Match_{ast} + \delta \cdot Match_{af} \quad (5.3)$$

²N-gram je označenie pre skupinu po sebe idúcich slov. Teda napríklad unigramy budú slová samotné a bigramy dvojice za sebou idúcich slov.

5.3.3 ROUGE

ROUGE (Recall-Oriented Understudy for Gisting Evaluation) [23] je metrika používaná na evaluáciu automatického generovania zhrnutí alebo iných textových úloh, kde je dôležitá miera zhody medzi generovaným výstupom a referenčným textom. Na rozdiel od BLEU, ktoré je zamerané na presnosť, ROUGE posudzuje n-gramovú zhodu z hľadiska úplnosti (angl. recall), teda podiel počtu zhodných slov v generovanom texte a počtu slov v referenčnom texte.

ROUGE je sada viacerých variant, ako napríklad ROUGE-N (n-gramová úplnosť), ROUGE-L (najdlhšia spoločná sekvencia), a ROUGE-S (zhoda založená na preskakovaní n-gramov). Tieto metriky umožňujú posúdiť text z rôznych perspektív, čo je dôležité pri komplexných úlohách, ako je zhrnutie textu, kde môže byť mnoho platných formulácií.

$$ROUGE - N = \frac{\sum_{s \in Reference} \sum_{gram_n \in s} Count_{match}(gram_n)}{\sum_{s \in Reference} \sum_{gram_n \in s} Count(gram_n)} \quad (5.4)$$

5.3.4 ChrF

ChrF (Character n-gram F-score) [30] je metrika, ktorá hodnotí kvalitu prekladu alebo textového generovania na základe zhody znakových n-gramov medzi referenčným a generovaným textom. Na rozdiel od slovne orientovaných metrík, ako BLEU alebo ROUGE, chrF skúma text na úrovni znakov, čo umožňuje zachytiť jemnejšie nuansy jazyka, ako sú koncovky slov a morfológické štruktúry. Výpočet metriky chrF je založený na výpočte F-skóre, ktoré je váženým súčtom presnosti a úplnosti:

$$ngrF_{\beta} = (1 + \beta^2) \cdot \frac{ngrP \cdot ngrR}{\beta^2 \cdot ngrP + ngrR}, \quad (5.5)$$

kde β je parameter, ktorý ovplyvňuje vzájomnú váhu presnosti a úplnosti, $ngrP$ je presnosť znakových n-gramov a $ngrR$ je úplnosť znakových n-gramov.

Verzia chrF++ [31], ktorá je rozšírením pôvodnej metriky, zahŕňa do výpočtu okrem znakových n-gramov aj slovné unigramy a bigramy. Vďaka tomu chrF++ kombinuje výhody znakových a slovných metrík, čo umožňuje lepšiu koreláciu s referenčným textom a presnejšie hodnotenie kvality generovaného textu. ChrF++ vypočítame ako aritmetický priemer F-skóre pre znakové n-gramy a slovné k-gramy.

5.3.5 Pass@k

Metrika pass@k [19] sa používa pre hodnotenie funkčnej správnosti kódu vygenerovaného modelom. Základná myšlienka spočíva v nedostatku metrík založených na porovnávaní s referenčným výstupom, ktoré nemôžu zachytiť komplexný priestor ekvivalentných riešení a navyše majú problém zachytiť sémantické špecifiká kódu. Preto sa pass@k zameriava na vykonanie generovaného kódu a vyhodnotenie jeho správnosti pomocou sady vopred definovaných testovacích príkladov. Inšpiráciu nachádza v testovaní riadenom vývoji softvéru (Test-Driven Development, TDD), ktoré je štandardnou praxou v softvérovom inžinierstve. Metrika sa používa ako hodnotiace kritérium toho, či je kód správne implementovaný. Práca [7] definuje pass@k ako:

$$pass@k := E \left[1 - \left(\frac{\binom{n-c}{k}}{\binom{n}{k}} \right) \right],$$

kde k je počet zvolených riešení z $n \geq k$ vygenerovaných vzoriek pre daný príklad. Premenná $c \leq n$ je počet vzoriek, ktoré úspešne prešli testovaním.

Zjavnou nevýhodou tejto metriky je, že vyžaduje existenciu sady testovacích príkladov pre konkrétnu úlohu. Tieto príklady musia byť navyše vysokej kvality a musia pokrývať širokú škálu možných riešení. Vykonávanie strojovo generované kódu môže predstavovať bezpečnostné riziká. Je preto potrebné mať k dispozícii izolované testovacie prostredie, ktoré umožní bezpečné vykonávanie kódu. V oblasti programovania vstavaných systémov a mikrokontrolérov je toto obzvlášť dôležité, pretože chyby v kóde môžu mať vážne následky na fyzické zariadenie. Preto je v tejto oblasti na zváženie použitie nástrojov na emuláciu hardvéru.

Dátová sada HumanEval

Zlatý štandard v oblasti funkčného testovania modelov je dátová sada HumanEval [7]. HumanEval je ručne vytvorená dátová sada, ktorá je špeciálne navrhnutá na evaluáciu schopností generovania kódu umelou inteligenciou v jazyku Python a obsahuje 164 príkladov. Úlohy v sade HumanEval sú formulované ako funkcie, ktoré vyžadujú naprogramovanie správneho riešenia na základe špecifikovaných požiadaviek. Každá úloha obsahuje textuálny popis, signatúru, telo funkcie a súbor testovacích prípadov, ktoré overujú funkčnosť riešenia. Tieto testy zahŕňajú vstupy a očakávané výstupy, čím poskytujú objektívne meradlo úspešnosti riešenia.

Kapitola 6

Prehľad súčasných riešení

V tejto kapitole predstavím pokročilé modely pre generovanie zdrojového kódu a ich vlastnosti. Poviem, v čom sa do seba odlišujú, a zhrniem výhody a nevýhody použitia daného modelu. V súčasnosti prebieha vývoj veľkých jazykových modelov rýchlym tempom. Nové modely vychádzajú na mesačnej, ak nie na týždennej báze a je náročné sledovať vývoj v tejto oblasti. Preto vyberám iba niektoré modely, ktoré sú zaujímavé z hľadiska práce. Pred tým, ako sa dostanem k samotným modelom, je potrebné si povedať niečo o dátových sádach, ktoré sú používané na tréning týchto modelov, a to predovšetkým v oblasti vstavaných systémov a mikrokontrolérov.

6.1 Dátové sady pre generovanie kódu

V oblasti generovania kódu existuje množstvo dátových sád, ktoré sú špeciálne navrhnuté na tréning modelov pre generovanie kódu. Tieto dátové sady zahŕňajú rôzne typy kódov, ako sú napríklad kódy z webových stránok, kódy z verejných repozitárov alebo z programovacích súťaží. V súčasnosti existuje množstvo verejne dostupných dátových sád, ktoré sú špeciálne zamerané na generovanie kódu v jazykoch ako Python, Java, C++ alebo JavaScript.

V oblasti programovania vstavaných systémov a mikrokontrolérov som pri hľadaní narazil iba na jednu dátovú sadu [53]. Táto dátová sada bohužiaľ nie je dostupná pre širokú verejnosť. Navyše je pomerne malá (1000 vzoriek) a nie je zameraná na generovanie kódu, ale na jeho klasifikáciu. Myslím si, že výskum v danej problematike má zmysel. Je preto vhodné vytvoriť nové dátové sady zamerané na generovanie kódu pre vstavané systémy a mikrokontroléry. Štruktúra a proces tvorby vzoriek spomínanej dátovej sady poskytuje vhodný príklad potenciálnej stavby vzoriek novej dátovej sady.

6.2 Prehľad modelov pre generovanie kódu

6.2.1 Codex

CODEx [7] je séria modelov, ktoré vznikli ladením GPT-3 na rozsiahlej kolekcii dát kódu a textov prirodzeného jazyka. Dátová sada pre tréning Codex modelov vznikla zozbieraním 54 miliónov verejne dostupných repozitárov na platforme GitHub a má celkovú veľkosť 159 GB. Codex, presnejšie verzia CODEx 12B s dvanástimi miliardami parametrov, tvoril jadro nástroja GitHub Copilot. Je schopný generovať kód v rôznych jazykoch, vrátane jazyka Python, JavaScript, Go, PHP, Ruby, Swift alebo TypeScript.

Značnou nevýhodou je jeho uzavretosť a nedostupnosť váh. To nedovoľuje trénovať model na vlastných dátach, a tým prispôbiť ho na konkrétnu úlohu. Na druhej strane tak, ako je možno vidieť v tabuľke 6.1, CODEX 12B dosahuje vysoké skóre a je schopný generovať kód na základe krátkych popisov a požiadaviek. Môže tak slúžiť ako referenčný model pri vývoji iných modelov.

Model	pass@1	pass@10	pass@100
CODEX 12B	28,81 %	46,81 %	72,31 %
CODELLAMA-INSTRUCT 7B	34,8 %	64,3 %	88,1 %
STARCODERBASE 1B	15,17 %	—	—

Tabuľka 6.1: Reportované pass@k skóre jednotlivých modelov na sade príkladov HumanEval.

6.2.2 CodeLLaMA

CODELLAMA-INSTRUCT 7B je súčasťou širšej rodiny voľne prístupných veľkých jazykových modelov pre generovanie a dopĺňovanie kódu CODE LLAMA [37]. Označenie 7B v názve značí počet váh. Konkrétne sa skladá z 32 dekodér blokov spolu s takmer siedmimi miliardami parametrov, čo ho stavia do pozície modelu so značným výpočtovým výkonom a inteligenciou. Označenie INSTRUCT hovorí, že tento model bol ladený tak, aby odpovede lepšie zodpovedali zadaným pokynom, a to posilňovaným učením s využitím špeciálnej dátovej sady.

Modely CODE LLAMA vznikli trénovaním modelov LLAMA 2 [41], ktoré sú pôvodne určené pre modelovanie prirodzeného jazyka. Tieto modely následne ďalej trénovali na sade veľkého množstva zdrojových súborov a vzoriek prirodzeného jazyka súvisiacich s kódom. Čo sa týka vzoriek zameraných na prirodzený jazyk, tak aj tieto súvisia s kódom a ide napríklad o príspevky na diskusných fórach, ktoré diskutujú rôzne programovacie problémy, ich riešenia, a obsahujú úryvky kódu. Model, ktorý som si zvolil, bol navyše ladený na troch dátových sadách s celkovou veľkosťou 5 miliard tokenov, z ktorých nás zaujímajú hlavne tieto:

1. Kolekcia dát zameraná na užitočnosť a bezpečnosť, predstavená v práci [41] – približuje formát a štruktúru odpovede tak, aby bola bezpečná a obsahovala čo najviac užitočných informácií.
2. Samoinštruovaná sada – Približne 14 000 trojíc (zadanie, testy a riešenie), zozbierané s využitím modelov LLAMA 2 70B (generovanie zadania programovacieho problému) a CODE LLAMA 7B (generovanie testov a správnych riešení problémov). Je zameraná na generovanie kódu na základe zadaného problému a testov, ktoré by mali byť splnené. Ide o samoinštruovanú kolekciu, ktorá vznikla dopytovaním iných modelov namiesto použitia anotácie ľudí. Konkrétna sada je zaujímavá z hľadiska interpretovateľnosti. Výsledky generovania testov a riešení poskytujú informácie pre analýzu schopností modelov. Na druhej strane, táto kolekcia dát môže obsahovať chyby, ktoré boli prenesené z dotazovaných modelov.

6.2.3 StarcoderBase

STARCODERBASE [22] je séria modelov predtrénovaných na podmnožine rozsiahlej kolekcie dát THE STACK (v 1.2), ktorá obsahovala 1 bilión tokenov a kódy v 80 programovacích jazykoch. Tieto modely s veľkosťou od 1 do 15,5 miliardy parametrov využívajú upravenú Multi-Head pozornosť, nazvanú Multi-Query pozornosť [39] a kontextové okno viac ako 8 tisíc tokenov.

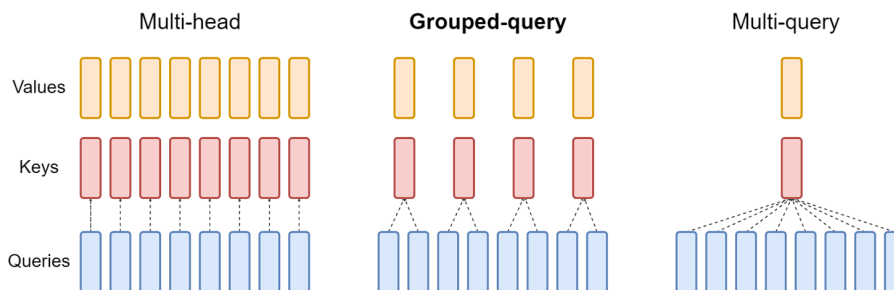
Veľkou výhodou je ich rýchlosť inferencie, dosiahnutá použitím Multi-Query pozornosti (MQA), kde sa využívajú rovnaké matice K a V pre všetky paralelne pracujúce hlavy pozornosti, čo zrýchľuje inferenciu. Sú vhodné pre použitie v reálnom čase, kde je dôležitá rýchlosť generovania kódu, a majú schopnosť generovať kód na základe kontextu, čo je užitočné pri nasadení vo vývojových prostrediach.

Nevýhodou je ich menšia veľkosť, čo môže ovplyvniť kvalitu generovaného kódu. Na rozdiel od modelov CODE LLAMA nie sú ladené na inštruované odpovedanie, ale sú zamerané na rýchle generovanie kódu v kontexte.

6.2.4 DeepSeek-Coder

DEEPSEEK-CODER [14] je séria modelov s veľkosťou od 1,3 do 33 miliárd parametrov. Modely boli predtrénované na 2 biliónoch tokenov z 87 programovacích jazykov. Využívajú prístup FIM s kontextovým oknom 16K na zlepšenie generovania a dopĺňania kódu na úrovni projektu. Hlavnou prednosťou modelov DEEPSEEK-CODER je vysoký výkon a schopnosť prekonať aj existujúce uzavreté modely ako CODEX alebo GPT-3.5-TURBO.

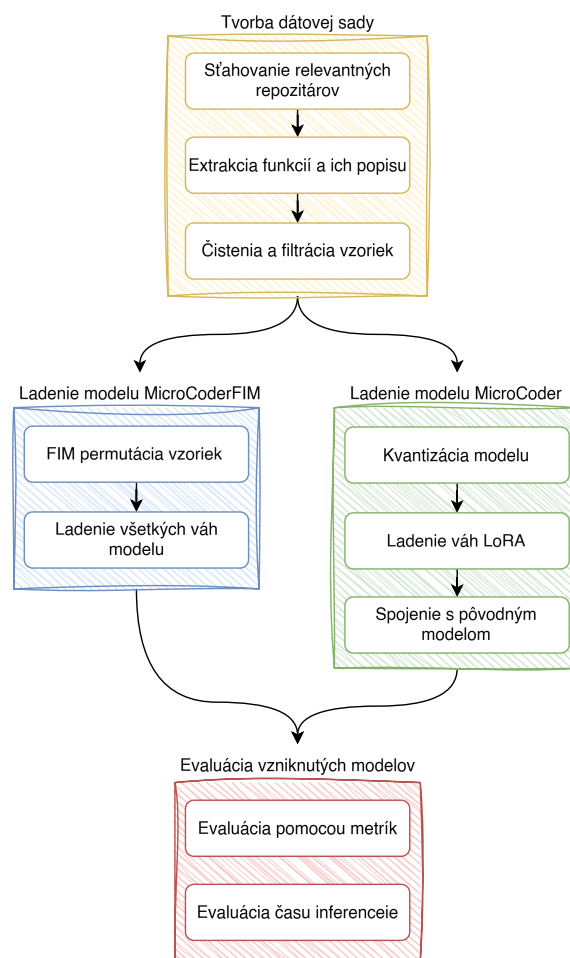
DEEPSEEK-CODER sú modely s dekodér transformer architektúrou postavené na rovnakom základe ako DEEPSEEK LLM [9]. Využívajú SWIGLU aktivačnú funkciu, a najväčší model implementuje Grouped-Query pozornosť (GQA). GQA je ďalší typ MHA, ktorá rozdeľuje matice váh W_Q do skupín tak, že skupina zdieľa jednu maticu W_K a W_V . Tento prístup syntetizuje prednosti MHA a MQA, čím dosahuje vyššiu výpočtovú rýchlosť v porovnaní s MHA a zároveň prekonáva MQA v kvalitatívnom hodnotení výstupov [2].



Obr. 6.1: Prehľad metód pozornosti. MHA obsahuje rovnaký počet matic Q , K a V . GQA zdieľa jednu maticu K a V pre skupinu matic Q , zatiaľ čo MQA zdieľa jedinou maticu K a V pre všetky Q . Prebrané z [2].

Kapitola 7

Návrh a implementácia



Obr. 7.1: Schéma návrhu riešenia práce.

V tejto kapitole sa venujem popisu procesu tvorby dátovej sady z existujúcich zdrojových súborov, následnému ladeniu predtrénovaných modelov na tejto dátovej sade a evaluácii vyvinutých modelov. Sekcia 7.1 poskytuje detailný popis metodiky vytvárania dátovej sady. V sekcii 7.2 následne na novej dátovej sade vykonávam stručnú analýzu niektorých jej vlastností. V sekcii 7.3 sa zaoberám kvantizáciou a tréňovaním rozsiahlejšieho modelu

CODELLAMA-INSTRUCT 7B a trénovaním kompaktnejšieho STARCODERBASE 1B na vytvorenej dátovej sade. V záverečnej sekcii 7.4 je predstavená evaluácia modelov na súbore testovacích príkladov.

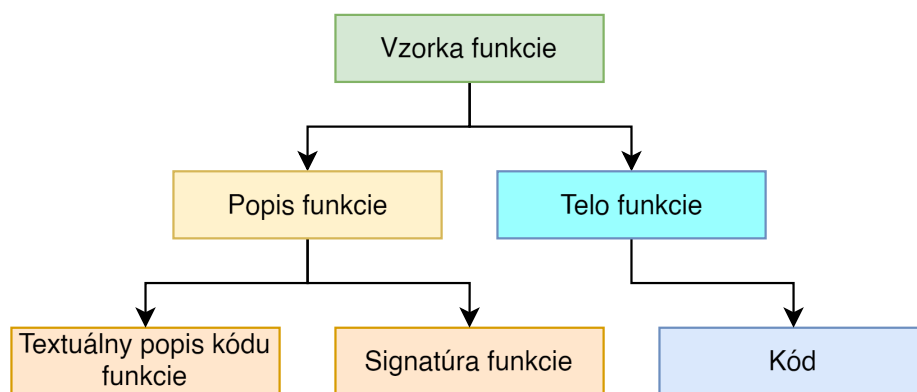
7.1 Dátová sada

Na zostavenie dátovej sady som využil zdrojové súbory z platformy GitHub. Táto platforma umožňuje ukladať a spravovať rôzne súbory, ako sú predovšetkým zdrojové kódy, a umožňuje ich verejné zdieľanie s ostatnými užívateľmi. Zdrojové súbory sú uložené v gitových repozitároch, ktoré poskytujú priestor pre ukladanie súborov a zaznamenávanie histórie zmien v projektoch. Pre účely práce som sa zameriaval na zdrojové súbory napísané v jazykoch C++ alebo C, pretože tieto jazyky sú najčastejšie používané v oblasti programovania vstavaných systémov a mikrokontrolérov.



Obr. 7.2: Proces tvorby dátovej sady.

Výsledok krokov tvorby dátovej sady, ktoré popisujem nižšie, je dátová množina s približne 50 tisíc príkladmi. Dátovú sadu som rozdelil v bežne používanom pomere 80 : 10 : 10 na trénovaciu, validačnú a testovaciu podmnožinu. Každá vzorka obsahuje zadanie (textový popis a signatúra funkcie), a referenčné riešenie (vnútorný kód funkcie). Dátovú sadu som zverejnil a je voľne dostupná na platforme Hugging Face¹. Povzbudzujem všetkých záujemcov o danú oblasť generovania kódu, aby prispeli k rozšíreniu dátovej sady alebo použili dáta na vytvorenie nových modelov.



Obr. 7.3: Štruktúra vzorky dátovej sady.

¹https://huggingface.co/datasets/xvadov01/cpp_emb_n12p1

Zber zdrojových súborov

Prvým krokom pri zbere dát bolo vyhľadávanie vhodných zdrojových súborov. Použil som dva prístupy:

1. Vyhľadávanie projektov na základe témy, ktorú danému repozitáru prideli jeho autor. V tomto kroku som vyhľadával repozitáre s témou FIRMWARE, MICROCONTROLLER a EMBEDDED-SYSTEMS. Z každej témy bolo zvolených 50 repozitárov na základe kritéria udelených hviezd².
2. Vyhľadávanie na základe kľúčových slov v metadátach repozitáru (názov, obsah súboru README, popis, ...). Zameral som sa na vyhľadávanie kľúčových slov CORTEX-M0, CORTEX-M0+, CORTEX-M1, CORTEX-M3, CORTEX-M4, CORTEX-A, STM32, ESP32 a ESP8266.

Filtrovanie súborov a predspracovanie zdrojového kódu

Z každého stiahnutého repozitára boli vybrané iba zdrojové súbory v jazyku C/C++. Súbory, ktoré neobsahovali zdrojový kód, ako napríklad dokumentácia alebo konfiguračné súbory, boli z repozitárov odstránené. Tiež boli odstránené všetky priečinky a súbory, ktorých názov obsahoval slovo `test`.

Štýl písania popisov funkcií v jednotlivých repozitároch sa líši. Snahou bolo upraviť čo najviac komentárov tak, aby vyhovovali štýlu JAVADOC. Tento štýl je štandardným spôsobom písania popisov funkcií v jazykoch C/C++ a je podporovaný mnohými nástrojmi na generovanie dokumentácie. Často sa vyskytoval formát, ktorý uvádzal komentár znakmi `/*\n`³, ten bol nahradený tak, aby komentár začínal `/**\n`.

Extrakcia funkcií

Z každého zdrojového súboru boli extrahované individuálne funkcie, pretože cieľom je trénovať model na funkciách, nie na celých súboroch. Pre extrahovanie jednotlivých funkcií s ich popisom som využil nástroj Doxygen, respektíve syntaktický analyzátor (parser), ktorý Doxygen poskytuje. Tento parser umožňuje spracovanie zdrojového súboru a generovanie jeho reprezentácie vo formáte XML. Parser identifikuje v zdrojovom súbore popisy funkcií, ktoré programátor napísal v určitom formáte. Blok komentára je následne spracovaný parserom a z neho je vygenerovaná XML reprezentácia, ktorá obsahuje informácie o funkcii, ako sú jej názov, popis, parametre a návratová hodnota. Podobne sú identifikované aj bloky funkcií, ktoré sú následne extrahované priamo zo zdrojového súboru.

Syntaktický analyzátor Doxygen je schopný identifikovať popisy funkcií, ktoré sú napísané v rôznych štýloch. Štandardným spôsobom písania popisov funkcií v jazykoch C/C++ je štýl JAVADOC, ktorý je podporovaný mnohými nástrojmi na generovanie dokumentácie.

Použil som prepínač `JAVADOC_BLOCK`, ktorý umožní, aby parser identifikoval bloky komentárov typu "banner", ktorý nie je štandardným štýlom, ale často sa v praxi využíva.

Týmto spôsobom som extrahoval 425 090 jednotlivých funkcií z celkového počtu 196 916 súborov rozdelených do 421 GitHub repozitárov.

²Hviezdy udeľujú užívatelia. Počet hviezd, ktoré repozitár získal, je dobrým ukazovateľom užitočnosti a popularity. Vysoký počet hviezd naznačuje, že daný repozitár je v komunite uznávaný.

³Označením nového riadku zdôrazňujem, že ide o viacriadkový komentár.

Predspracovanie a tokenizácia

Pred ďalšou prácou s dátovou sadou som vykonal úpravy na extrahovaných textoch s cieľom ich optimalizácie. Postupoval som systematicky, začínajúc zjednotením popisu funkcií. Tento proces zahŕňal vytvorenie jednotného opisu funkcií z krátkych a podrobných popisov automaticky extrahovaných použitým parserom. Ďalej som aplikoval filtrovanie vzoriek na základe identifikovaných kľúčových slov, ktoré som považoval za nežiaduce v kóde alebo popise funkcií. Po filtrácii funkcií som ďalej upravil kódy funkcií a ich popisy odstránením identifikovaných vzorcov, ktoré som získal prostredníctvom systematického preskúmania náhodných príkladov.

Hlavným dôvodom bolo odstrániť chyby automatickej extrakcie funkcií zo zdrojových súborov. Často bol text funkcie extrahovaný čiastočne alebo obsahoval znaky zo signatúry, prípadne zblúdilé znaky z priestoru za/pred funkciou, ktoré pravdepodobne patrili inej časti kódu. Bolo preto najprv potrebné upraviť telo funkcie tak, aby začínalo znakom „{“ a končilo znakom „}“. Ďalšie čistenie zahŕňovalo odstraňovanie vzorcov ako napríklad:

- Viacnásobné riadkové skoky – skresľovali počet riadkov v tele funkcie.
- Komentáre v tele funkcie – je nežiaduce, aby sa model priveľa zameriaval na komentáre v kóde, preto boli komentáre odstránené využitím regulárnych výrazov.

Textový popis funkcie bol očistený o nežiaduce vzorce a formátovaný podľa vzoru dátovej sady HUMAN-EVAL-X [52] pre jazyk C++.

Následne bol takto vyčistený kód a popis funkcie tokenizovaný. Pre tokenizáciu som zvolil tokenizér modelu STARCODERBASE 1B. Táto voľba bola založená na preskúmaní výstupov tokenizérov oboch modelov a spôsobu, akým rozkladali časti kódu, predovšetkým premenné a výrazy, na jednotlivé tokeny.

Filtrácia funkcií na základe tokenov

Z extrahovaných funkcií som odstránil všetky, ktoré mali menej ako 20 tokenov v tele funkcie. Takéto krátke funkcie sú zvyčajne nepodstatné a neprinášajú žiadnu hodnotu pre tréning. Taktiež som odstránil funkcie, ktoré obsahovali viac ako 256 tokenov, pretože tieto funkcie môžu obsahovať priveľa zložitých vzťahov, ktoré by spôsobili problémy pri tréningu. Obmedzenie pre počet riadkov v tele funkcie som nastavil v rozmedzí $3 \leq n \leq 30$. Ďalším kritériom bol počet tokenov v popise funkcie. Tento parameter bol obmedzený spodnou hranicou 10 a hornou hranicou 100 tokenov. Toto obmedzenie bolo nastavené tak, aby som získal popisy funkcií, ktoré sú dostatočne podrobné na to, aby z nich model mohol extrahovať podstatné informácie pre generovanie kódu.

Odstránenie duplicitných vzoriek a repozitárov s malým počtom vzoriek

Posledný krok spočíval v odstránení duplicitných vzoriek, ktoré vznikli pri úprave dát. Odstránil som aj vzorky z repozitárov s menej ako 100 vzorkami. Nízke zastúpenie v repozitároch by neumožnilo modelu získať potrebné informácie pre efektívne tréningu a naznačuje nedostatočnú dokumentáciu repozitára.

```

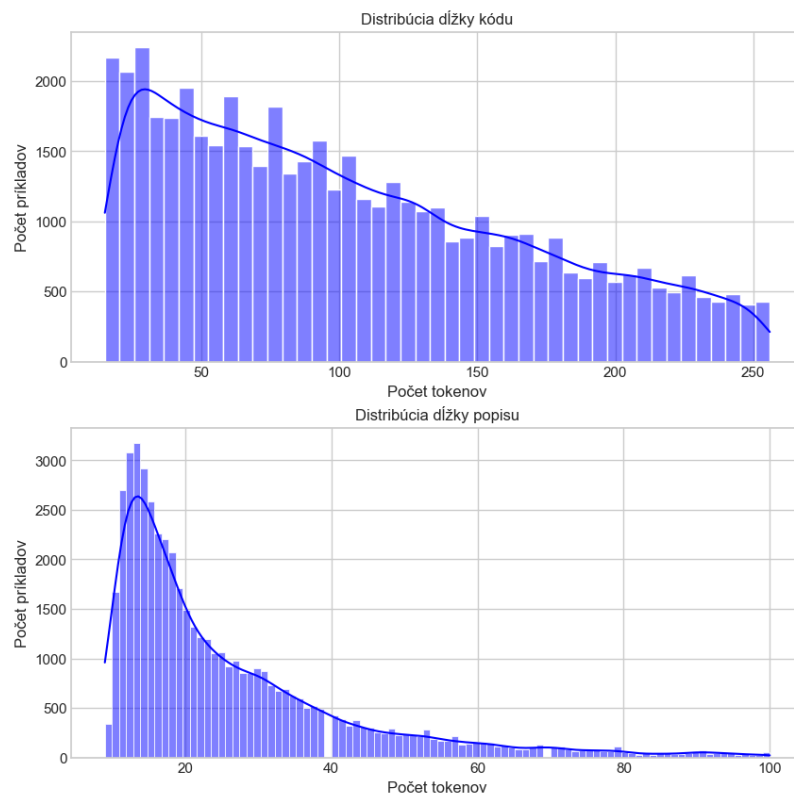
1 /* This function waits until the SDIO DMA data transfer is finished. This function should be called after
   SDIO_WriteBlock() and SDIO_WriteMultiBlocks() function to insure that all data sent by the card are already
   transferred by the DMA controller. */
2 SD_Error SD_WaitWriteOperation(void)
3 {
4     SD_Error errorstatus = SD_OK;
5     while ((SD_DMAEndOfTransferStatus() == RESET) && (TransferEnd == 0) && (TransferError == SD_OK))
6     {}
7     if (TransferError != SD_OK)
8     {
9         return(TransferError);
10    }
11    SDIO_ClearFlag(SDIO_STATIC_FLAGS);
12    return(errorstatus);
13 }

```

Obr. 7.4: Ukážka príkladu z dátovej sady.

7.2 Analýza dátovej sady

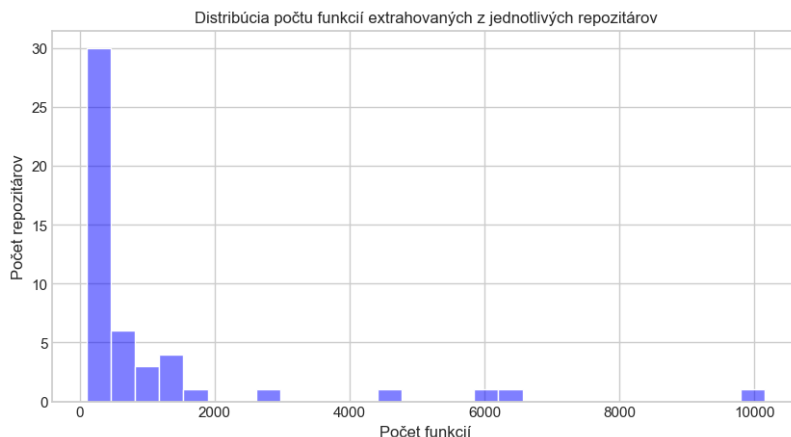
Analýza dát bola vykonaná s cieľom získať lepšie porozumenie pre štruktúru a charakteristiku dátovej sady. Analýza zahŕňala výpočet štatistík, ako sú priemerná dĺžka distribúcia tokenov v tele a popise funkcie a výskyt kľúčových slov jazyka v kóde.



Obr. 7.5: Distribúcia počtu tokenov v popise a tele funkcie.

Dátová sada obsahuje celkom 6,5 milióna tokenov, z ktorých 5,23 milióna tokenov je v tele funkcie a 1,3 milióna tokenov je v popise funkcie. Priemerná dĺžka popisu funkcie je 25 tokenov a priemerná dĺžka tela funkcie je 104 tokenov. Distribúcia dĺžky popisu a tela funkcie v počte tokenov je zobrazená na obrázku 7.5. Z grafu je možno určiť, že väčšina

popisov funkcií má dĺžku do 20 tokenov, zatiaľ čo telá funkcií majú menšiu variabilitu a väčšina z nich má dĺžku v rozmedzí 50 až 150 tokenov.



Obr. 7.6: Distribúcia počtu funkcií extrahovaných z jednotlivých repozitárov.

Na obrázku 7.6 je zobrazený počet vzoriek z jednotlivých repozitárov v dátovej sade. Z grafu je vidieť, že väčšina vzoriek pochádza z repozitárov, ktoré majú zastúpenie v rozmedzí do 1000 vzoriek. Najväčšie zastúpenie vzoriek v sade má repozitár `robotest/uclinux` s počtom viac ako 10 000 príkladov.



Obr. 7.7: Distribúcia kľúčových slov v kóde funkcie

Extrakcia kľúčových slov z fragmentov zdrojového kódu umožňuje identifikovať opakujúce sa vzorce a posúdiť zložitost programovania. Analýza štatistik kľúčových slov z dátovej sady poskytuje hlbší pohľad na zdrojový kód, ktorý často závisí od riadenia toku, je presný pri manipulácii s dátami a je systematicky štrukturovaný pri riadení systémových stavov a pamäťových operácií.

7.2.1 Možné rozšírenia dátovej sady

Pri extrakcii funkcií som zvažoval, ako pridať k jednotlivým príkladom aj informácie o linkovaní knižníc a ich funkcií, ktoré sú využívané v kóde. Tieto informácie by mohli byť následne využité pri generovaní kódu, pretože by poskytli informácie o tom, aké knižnice

sú potrebné pre správne fungovanie vygenerovaného kódu. V zozbieraných repozitároch sa nachádzalo veľké množstvo hlavičkových súborov.

Prekážkou bolo, že išlo iba o hlavičkové súbory, ktoré obsahovali informácie o linkovaní ďalších knižníc a vzťahy medzi jednotlivými súbormi v projektoch boli komplikované. Rozhodol som sa túto informáciu neextrahovať, z dôvodu nutnosti zaviesť zložitejší mechanizmus na identifikáciu a spracovanie hlavičkových súborov. Verím, že táto informácia by mohla byť doplnená v budúcnosti, a vylepšiť tak kvalitu generovaného kódu.

Ďalším rozšírením by bola integrácia prvkov zo štandardov MISRA-C, CERT a podobných. Tieto smernice majú za cieľ uľahčiť vývoj bezpečného a spoľahlivého softvéru, a predchádzať tak možným chybám v kriticky dôležitých systémoch. Splňanie týchto štandardov je kľúčové v oblasti vstavaných systémov, kde bezpečnosť a spoľahlivosť majú najvyšší význam.

Integrácia týchto smerníc by umožnila zlepšiť kvalitu dátovej sady a zabezpečila by, že vygenerovaný kód bude spĺňať prísne požiadavky na bezpečnosť a spoľahlivosť. Konkrétne by sa mohli zahrnúť pravidlá MISRA-C do procesu filtrácie vzoriek, aby sa zaistilo, že sú konformné s týmito smernicami.

7.3 Ladenie modelov

Pre ladenie, evaluáciu a inú prácu s modelmi som vytvoril niekoľko skriptov v jazyku Python, v ktorých využívam rôzne knižnice, ktoré mi uľahčili prácu s modelmi v Jupyter zápisníkoch alebo priamo v Python skriptoch. Platforma Hugging Face poskytuje rôzne nástroje pre prácu s modelmi, ako sú napríklad knižnice `transformers`, `datasets` a `tokenizers`. Knižnice `transformers` a `tokenizers` poskytujú jednoduché rozhranie pre načítanie a použitie predtrénovaných modelov a tokenizérov, ktorých váhy a konfigurácie sú uložené v repozitári Hugging Face.

Pri samotnom tréningu som využil knižnicu `trl` (Transformer Reinforcement Learning), ktorá poskytuje jednoduché API rozhranie `SFTTrainer`. Supervised Fine-tuning Trainer, tak ako názov napovedá, je nástroj optimalizovaný pre ladenie technikou tréningu pod dohľadom. Táto knižnica natívne podporuje metódy PEFT. Okrem toho som do svojej práce začlenil `BitsAndBytesConfig` z knižnice `transformers` pre kvantizáciu, ako aj `LoraConfig` pre nastavenia LoRA z knižnice `peft`.

Váhy modelov `MICROCODERFIM`⁴ a `MICROCODER`⁵ sú voľne prístupné na verejnom úložisku.

Parameter	Hodnota	
	MicroCoder	MicroCoderFIM
počet epoch	3	5
miera učenia	1×10^{-4}	3×10^{-5}
lr_scheduler_type	constant	cosine
per_device_batch_size	2	8
gradient_accumulation_steps	1	8
warmup_ratio	5%	5%
optimalizátor	paged 32bit AdamW	paged 32bit AdamW

Tabuľka 7.1: Prehľad nastavených parametrov počas ladenia modelov.

⁴<https://huggingface.co/xvadov01/microcodertifim-1B>

⁵<https://huggingface.co/xvadov01/microcoder-7B-q4>

7.3.1 MicroCoder

Proces optimalizácie modelu MICROCODER prebiehal pomocou ladenia, kde základný model CODELLAMA-INSTRUCT 7B, ktorý som opísal v sekcii 6.2.2, podstúpil ďalšiu fázu tréovania. Tento postup viedol k vylepšeniu výkonnosti v oblasti programovania vstavaných systémov.

V tabuľke 7.2 je prehľad veľkosti modelu pred a po kvantizácii a aplikácii LoRA. Je zjavné, že kvantizácia a aplikácia LoRA viedli k významnému zmenšeniu veľkosti. Kvantizácia umožňuje redukciu rozsahu modelu, avšak nedovoľuje tréovanie kvantizovaných váh. Pri tréovaní kvantizovaných modelov sa preto upravujú iba váhy LoRA. Tieto váhy je následne možné spojiť s kvantizovanými, alebo pôvodnými váhami. Tento postup umožňuje načítať zmenšený model, optimalizovať vybrané projekcie a následne spojiť váhy LoRA s váhami modelu tak, aby bolo možné využiť vzniknutý model. Takýto prístup reprezentuje efektívnu metódu ladenia, čo je zásadné v prípade práce s obmedzenými výpočtovými prostriedkami. Voľba QLoRA bola motivovaná analýzou vplyvu kvantizácie na výkon veľkého modelu, ktorý disponuje takmer siedmimi miliardami parametrov. Využitie kvantizačných techník mi umožnilo tréovať model na jednej grafickej karte NVIDIA GEFORCE RTX 3090 s 24 GB pamäťou. Parametre a ich nastavenia pre QLoRA adaptáciu sú v tabuľke 7.3.

	Počet trénovateľných parametrov	Veľkosť (GiB)
Základný model	6 788 878 336	25,60
16bit	6 788 878 336	12,80
8bit	6 788 878 336	6,77
4bit	6 788 878 336	3,75
LoRA $r = 64$	33 554 432	0,064

Tabuľka 7.2: Porovnanie veľkosti modelu MICROCODER pred a po kvantizácii a aplikácii LoRA.

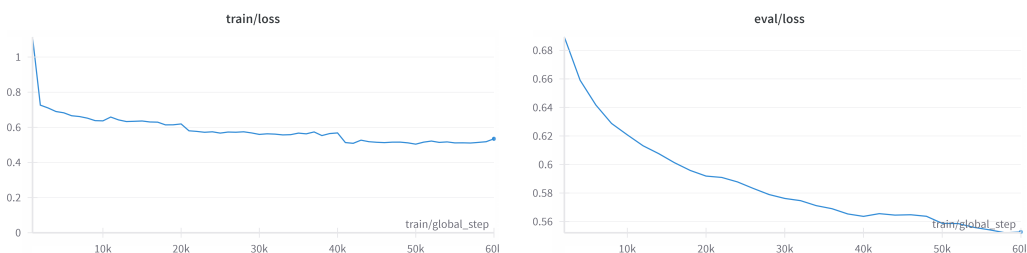
Parameter	Hodnota	
r	32	LoRA rank, rozmer adaptácie
lora_alpha	32	Škálovací faktor, zvyšuje vplyv nízkorozmerných matíc
target_modules	Q, K, V	Trénované projekcie dotazu, kľúča a hodnoty
lora_dropout	0,05	Podiel váh LoRA nastavených na nulu
Kvantizácia	4 bity	Bitová hĺbka váh
Kompresný typ	nf4	Formát pre ukladanie kvantizovaných váh
Výpočtový typ	bfloat16	Formát pre vykonávanie výpočtov

Tabuľka 7.3: Nastavenia metódy QLoRA pre tréovanie modelu MICROCODER.

Pri tréovaní bola použitá konštantná miera učenia 1×10^{-4} , čo umožnilo slabú, ale stabilnú konvergenciu počas celého procesu tréovania. Veľkosť dávky bola nastavená na 2, čo optimalizovalo využitie pamäte GPU a zároveň udržiavalo efektívny tok. Nakoľko GPU GEFORCE RTX 3090 má obmedzenú kapacitu pamäte, akumulácia gradientov bola nastavená na 1, čo umožnilo pravidelné aktualizácie váh bez nadmerného zaťaženia pamäte.

Celkový počet epoch bol určený na 3 a proces zabil 15 h a 28 m. Zahrievací pomer bol nastavený na 5%, čo zabezpečuje plynulé zvyšovanie miery učenia na začiatku tréovania a predchádza príliš rýchlym a nestabilným zmenám vo váhach.

V tabuľke 7.1 je uvedený prehľad nastavených hyper-parametrov ladenia modelu MICROCODER.



Obr. 7.8: Graf zobrazujúci tréovaciu a validačnú stratu modelu MICROCODER počas viacerých epoch, ilustrujúci pokrok v učení a hodnotenie výkonnosti modelu.

7.3.2 MicroCoderFIM

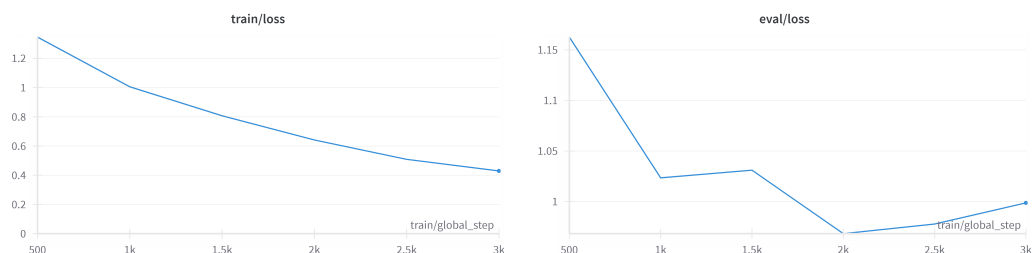
Pre druhú fázu ladenia som vybral model STARCODERBASE 1B, podrobne opísaný v sekcii 6.2.3. Výber tohto modelu bol motivovaný jeho menšou komplexitou a relatívne jednoduchšou tréovateľnosťou. Táto vlastnosť umožnila experimentovanie s tréovaním všetkých váh pri zachovaní 32-bitovej presnosti. Proces ladenia modelu bol realizovaný na grafickej karte NVIDIA GEFORCE RTX 3090 po dobu 5 hodín a 25 minút. Pri ladení bola nastavená miera učenia 3×10^{-5} a ďalšie parametre, ktoré sú podrobným spôsobom uvedené v tabuľke 7.1.

Príprava vzoriek do formátu pre tréovanie

Na tréovanie modelu som použil rovnakú dátovú sadu ako pri tréovaní modelu MICROCODER, ale s iným prístupom k príprave vzoriek. Model bol tréovaný na úlohe FIM a bolo potrebné pripraviť vzorky do formátu SPM (Suffix-Prefix-Middle) alebo PSM (Prefix-Suffix-Middle). Pre tieto účely som adaptoval funkciu `permute`, ktorú som prebral z [3]. Vstup do funkcie je kód spolu s textovým popisom funkcie. Výsledok je text obohatený o tokeny, ktoré označujú oddelenie jednotlivých častí. Používam pravdepodobnosť transformácie FIM 100%, čo znamená, že všetky vzorky sú transformované do formátu FIM. Dĺžka jednotlivých častí je $\frac{1}{3}$ s náhodným posunom o maximálne 10% dĺžky vzorky. Pre inferenciu som využil len časť po token `<MID>` tak, aby model dostal informáciu, že má nasledovať stredná časť, ktorú následne generuje.

Tokenizovaný vstup pre tréovanie potom nadobúda jeden z formátov:

`<PRE>(prefix)<SUF>(suffix)<MID>(middle)`
`<PRE><SUF>(suffix)<MID>(prefix)(middle)`



Obr. 7.9: Graf trénovacej a validačnej straty modelu MICROCODERFIM. Trénovacia strata postupne klesá, zatiaľ čo validačná strata spočiatku jemne klesá, ale následne stúpa. Tento jav naznačuje možný výskyt preučenia. Model dobre funguje na trénovacích dátach, ale jeho schopnosť generalizovať na nové, nevidené dáta, je obmedzená.

7.3.3 Emisie CO₂

Uvádžam uhlíkovú stopu [20] modelov MICROCODER a MICROCODERFIM, s cieľom porovnať ich efektivitu a environmentálny dopad. Obidva modely boli trénované na grafickej karte NVIDIA GEFORCE RTX 3090, čo umožňuje priame porovnanie ich energetických nárokov a súvisiacich emisií CO₂. Jednotka CO₂eq (ekvivalent oxidu uhličitého) predstavuje štandardný spôsob vyjadrenia rôznych typov skleníkových. Zohľadňuje relatívny vplyv na globálne otepľovanie konvertovaním množstva plynov na ekvivalentné množstvo oxidu uhličitého s rovnakým potenciálom globálneho otepľovania [1].

Model MICROCODER bol trénovaný po dobu 15,5 hodiny, počas ktorej vyprodukoval celkovo 2,34 kg CO₂eq. Model MICROCODERFIM vyžadoval menej trénovacieho času, konkrétne 5,5 hodiny, a jeho celkové emisie CO₂eq dosiahli 0,83 kg. Tieto údaje sú základom pre ďalšie diskusie o efektivite a environmentálnom vplyve pri vývoji jazykových modelov. Rozdiel v emisiách je značný a poukazuje na potenciálnu úsporu energie pri optimalizácii menších jazykových modelov.

7.4 Evaluácia modelov

Ladené modely som vyhodnotil na testovacej sade, ktorá pozostávala z 10% vzoriek z pôvodnej dátovej sady. Pre evaluáciu kvality modelov som využil metriky BLEU [32], CodeBLEU⁶, chrF++ [32] a ROUGE-L⁷. Tieto metriky som zvolil na základe zistení práce [12], ktorá zároveň tvrdí, že metrika chrF++ vykazuje najvyššiu koreláciu s ľudským hodnotením. Skóre modelu MICROCODER som vypočítal zo vzorky 1000 príkladov kvôli dlhšiemu času inferencie. Pre MICROCODERFIM som použil všetkých 5025 príkladov z testovacej sady. Výsledky evaluácie sú zhrnuté v Tab. 7.4.

Detaily k použitej implementácii a parametrom použitých metrik:

CodeBLEU Nastavenie váh $\alpha = \beta = \gamma = \delta = 0.25$, špecifikovaný programovací jazyk C++.

ChrF Parametre boli zvolené tak, aby zodpovedali metrike ChrF++ (nastavené bigramy slov a 6-gramy znakov).

⁶<https://github.com/k4black/codebleu>

⁷<https://github.com/google-research/google-research/tree/master/rouge>

Úloha	Model	Metriky			
		BLEU	CodeBLEU	CHRF++	ROUGE-L
NL-PL	GPT-3.5 TURBO 175B	15,21	15,00	25,92	27,84
	CODELLAMA-INSTRUCT 7B	4,67	18,40	19,50	16,10
	DEEPSEEK-CODER 6.7B	9,63	17,18	25,79	18,45
	MICROCODER 7B	10,43	16,85	25,33	20,84
	MICROCODER ⁸ 7B	8,24	17,49	24,92	17,55
FIM	STARCODERBASE 1B	11,85	11,58	19,54	19,18
	MICROCODERFIM 1B	31,74	40,53	51,54	43,31

Tabuľka 7.4: Tabuľka výsledkov inferencie modelov na testovacej sade.

Pre porovnanie som vyhodnotil aj model CHATGPT. Pri vygenerovaní výsledkov bolo použité OpenAI API a model GPT-3.5 TURBO. Model dostal zadanie, aby generoval iba telo funkcie, bez dodatočných komentárov, ktoré tento model generuje kvôli technike posilňovaného učenia. Keďže ako referenčný výstup používam iba kód funkcie, bolo potrebné očistiť výstup tak, aby bola dosiahnutá najvyššia miera objektivity a až následne vypočítať skóre. Model bol vyhodnotený iba na vzorke 2000 testovacích príkladov z testovacej sady.

```

1 Complete the implementation of the function with the following docstring and signature:
2 {task}
3 Remember, you only need to fill in the body of the function, without altering the provided signature or adding
any additional comments.

```

Obr. 7.10: Formát promptu použitý pre model GPT-3.5 TURBO.

7.4.1 MicroCoder

Tento model si aj napriek nízkym výsledkom základného modelu polepšil iba o niekoľko percent vo všetkých metrikách okrem CodeBLEU. Aj keď výsledky ukazujú, že výstupy modelu sa vzdialene približujú referenčným výstupom, ukážky kódu 7.11 a 7.12 ukazujú, že model je schopný generovať syntakticky korektný kód, ktorý do určitej miery rešpektuje popis funkcie.

Adaptácia modelu na novú úlohu je v porovnaní s modelom MICROCODERFIM podstatne slabšia, ako ukazuje tabuľka 7.4. Tento fakt môže byť spôsobený tým, že model bol trénovaný po menší počet epoch. Ďalším možným faktorom je použitie kvantizácie a nízkorozmerných matic LoRA, ktoré mohli znížiť schopnosť modelu efektívne sa prispôbiť novej úlohe v porovnaní s modelom MICROCODERFIM, ktorý prešiel trénovaním všetkých váh bez kvantizácie.

Bežný postup je ladenie LoRA matic s využitím kvantizovaných váh modelu a následné spojenie 16-bitového adaptéru s modelom, ktorý je v 32 alebo 16-bitovej hĺbke. Pri testovaní modelov po zlúčení vykazovalo spojenie adaptéru so 16-bitovými váhami modelu v priemere o 2% horšie výsledky, ako zlúčenie s kvantizovanými váhami. Rozhodol som sa preto zlúčiť LoRA adaptér s kvantizovaným modelom, čím vznikol MICROCODER.

Všimol som si, že pri práci s kvantizovaným modelom sú rozdiely medzi rôznymi inicializáciami váh modelu výraznejšie a model sa môže ľahšie uväzniť v lokálnom optimálnom bode. Pri inicializácii váh následného modelu totiž dochádza k zníženiu presnosti reprezentácie váh. Malé zmeny vo váhach môžu spustiť reťazovú reakciu, ktorá vedie k zhoršeniu

⁸Model vzniknutý zlúčením LoRA váh do 16-bitového modelu.


```

1 /* Unregisters an interrupt handler for the timer
   interrupt. */
2 void TimerIntUnregister(unsigned long ulBase,
   unsigned long ulTimer)
3 {
4     ASSERT((ulBase == TIMER0_BASE) || (ulBase ==
   TIMER1_BASE));
5     ASSERT((ulTimer == 0) || (ulTimer == 1));
6
7     if (ulTimer == 0)
8     {
9         IntDisable(INT_TIMER0A);
10        IntUnregister(INT_TIMER0A);
11    }
12    else
13    {
14        IntDisable(INT_TIMER0B);
15        IntUnregister(INT_TIMER0B);
16    }
17 }

```

```

1 /* Unregisters an interrupt handler for the timer
   interrupt. */
2 void TimerIntUnregister(unsigned long ulBase,
   unsigned long ulTimer)
3 {
4     ASSERT(ulBase == TIMER0_BASE);
5     ASSERT((ulTimer == TIMER_A) || (ulTimer ==
   TIMER_B));
6
7     if(ulTimer == TIMER_A)
8     {
9         HWREG(ulBase + TIMER_0_IMR) &=
~TIMER_TIMA_TIMEOUT;
10    }
11    else
12    {
13        HWREG(ulBase + TIMER_0_IMR) &=
~TIMER_TIMB_TIMEOUT;
14    }
15 }

```

Obr. 7.11: Výstup modelu MICROCODER (vľavo) a výstup nástroja Github Copilot očistený o sprievodné komentáre (vpravo) pre rovnaký príklad.

výsledkov. V prílohe A sú detailnejšie popísané niektoré výstupy a problémy kvantizovaného modelu.

```

1 /* This routine will start the transmit DMA running if it is not already running. This routine ned only be
   called when queueing a new packet to an empty tx queue. Writes the current tx chain head as start address of
   the tx descriptor chain and enables the transmission DMA engine. */
2 static void spider_net_kick_tx_dma(struct spider_net_card *card) {
3     if (spider_netdev[0]->transmit_queue == card->active_queue && !list_empty(&card->sendq)) {
4         u32 val;
5         val = readl(card->iaddr + GLOBAL_CTRL);
6         writel((val | TXDMAEN), card->iaddr + GLOBAL_CTRL);
7     } else {
8         printk("Trying to kick off non-currently active send queue\n");
9     }
10 }
11 /* Sets up all necessary fields in the next available tx description, then kicks on the dma controller for this
   device's currently selected queue. Returns error code or zero upon success. If there are no free descriptors
   left we return -EBUSY */
12 int spider_net_start_xmit(struct sk_buff* skb, struct net_device *ndev){
13     unsigned long flags=0;
14     struct spider_net_card *card = dev_to_card(ndev);
15     local_irq_save(flags);
16     if (!skb || list_empty(&card->freeq)){
17         return NETDEV_TX_BUSY;
18     }else{
19         writew((__u16 *)&skb->data, &card->desc_head[card->next_out].address);
20         writew(sizeof(skb->len)+4+8, &card->desc_head[card->next_out++].length);
21         card->next_out %= SPIDER_NET_MAXFRAMES;
22         spider_net_kick_tx_dma(card);
23         atomic_dec(&card->pendingframes);
24     }
25     local_irq_restore(flags);
26     return NETDEV_TX_OK;
27 }

```

Obr. 7.12: Výstup modelu MICROCODER.

Výstup modelu na obrázku 7.12 ilustruje jeho schopnosť manipulovať s generickými hardvérovými funkciami, ako sú `local_irq_save` a `writel`. Zároveň je pozoruhodný plynulý prechod medzi funkciami, ktoré model vygeneroval a prepojil dva logické celky kódu.

7.4.2 MicroCoderFIM

Výsledky modelu MICROCODERFIM ukazujú výrazné zlepšenie vo všetkých metrikách v porovnaní so základným modelom. Tento model dosiahol najlepšie výsledky vo všetkých metrikách, čo naznačuje, že sa dokázal efektívne adaptovať na novú úlohu. Ukážky kódu 7.13 naznačujú schopnosť efektívne dopĺňať kód v kontexte a zachovávať jeho funkcionality.

Proces tréovania trval dlhšie ako pri skôr spomínanom modeli. Platí, že miera učenia bola nižšia a nebola konštantná, ale bola nastavená na kosínusovú mieru učenia, čo znamená, že dochádzalo k jej postupnému znižovaniu počas tréovania.

V rámci experimentálnej časti práce som vykonal rozsiahle testovanie modelu, aby som identifikoval optimálne parametre pre generovanie textu. Podrobnosti experimentov a ich výsledky sú detailne popísané v prílohe B, ktorá obsahuje analýzu vplyvu rôznych nastavení parametrov, ako sú teplota (temperature), maximálna dĺžka generovanej sekvencie (max_new_tokens) a pravdepodobnosť výberu (top_p), na výkonnosť modelu pri generovaní textu.

```

1 /* This function is registered to the SPI framework for this SPI master controller. It will
   queue the spi_message in the queue of driver if the queue is not stopped and return. */
2 static int pl022_transfer(struct spi_device *spi, struct spi_message *msg){
3     struct pl022 *pl022 = spi_master_get_devdata(spi->master);
4     unsigned long flags;
5     spin_lock_irqsave(&pl022->queue_lock, -EBUSY);
6     if (pl022->run == QUEUE_HALTED)
7         return -ESHUTDOWN;
8     if (pl022->stopping) {
9         spin_unlock_irqrestore(&pl022->queue_lock, flags);
10        return -ESHUTDOWN;
11    }
12    if (!list_empty(&msg->queue))
13        pl022->busy = 1;
14    else
15        pl022->busy = 0;
16    pl022->msg = msg;
17    msg->actual_length = msg->status = -EINPROGRESS;
18    msg->state = STATE_START;
19    list_add_tail(&msg->queue, &pl022->queue);
20    if (pl022->run == QUEUE_RUNNING &&!pl022->busy)
21        queue_work(pl022->workqueue, &pl022->pump_messages);
22    spin_unlock_irqrestore(&pl022->queue_lock, flags);
23    return 0;
24 }

```

Obr. 7.13: Model MICROCODERFIM má problém korektne naviazať na nasledujúci kód funkcie, ale je vidieť, že dobre zachytil kontext pred miestom vloženia nového kódu.

Výsledky hodnotenia pass@k, uvedené v tabuľke 7.5 na dátovej sade HumanEval, poukazujú na to, že malý model po jeho ladení výrazne stratil schopnosť efektívne pracovať s jazykom Python. V porovnaní s pôvodným modelom STARCODERBASE 1B sú výsledky o 48% nižšie. Ako ilustruje obrázok 7.14, aj napriek poklesu výkonu model stále dokáže generovať korektný kód. Tento vývoj bol očakávaný vzhľadom na veľkosť modelu a je kľúčový pre posúdenie schopnosti modelu generalizovať na nové úlohy bez straty predtým nadobudnutých znalostí. Použil som $n = 200$ výstupov na jeden príklad a parametre generovania sú temperature = 0,2 a max_new_tokens = 128. Pre inferenciu modelu som použil PSM formát techniky FIM. Ako prefix slúžilo zadanie z dátovej sady. Sufix som nechal prázdny.

Model	pass@1	pass@10	pass@100
MICROCODERFIM	7,83 %	12,73 %	19,07 %

Tabuľka 7.5: Výsledky metrick pass@1, pass@10, pass@100 modelu MICROCODERFIM na dátovej sade HumanEval.

```

1 from typing import List
2 def has_close_elements(numbers: List[float], threshold: float) -> bool:
3     """ Check if in given list of numbers, are any two numbers closer to each other
4     than given threshold.
5     >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
6     False
7     >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
8     True
9     """
10    if len(numbers) < 2:
11        return False
12    previous = numbers[0]
13    for number in numbers[1:]:
14        if abs(previous - number) < threshold:
15            return True
16        previous = number
17    return False

```

Obr. 7.14: Výstup modelu MICROCODERFIM na zadanie z dátovej sady HumanEval v jazyku Python.

7.4.3 Rýchlosť inferencie

Rýchlosť, akou model generuje kód, je dôležitým faktorom pri jeho použití v reálnom prostredí. Preto som vyhodnotil rýchlosť inferencie modelov na testovacej sade. Každý model prešiel najprv fázou „zohriatia“ na jednom príklade a následne bol zmeraný čas generovania kódu 10-krát pre jednu vybranú vzorku. Výsledky sú zhrnuté v tabuľke 7.6.

Model	Hĺbka [<i>bit</i>]	Ø rýchlosť inferencie [<i>token · s⁻¹</i>]	Ø čas inferencie [<i>s</i>]	Max. nových tokenov
CODELLAMA-	16	2,18	92,3	512
INSTRUCT 7B	8	10,98	38,30	512
	4	32,61	12,10	512
MICROCODER	4	32,82	10,99	512
STARCODERBASE	32	128	±2,1	256
1B				
MICROCODERFIM	32	128	±2,1	256

Tabuľka 7.6: Tabuľka rýchlosti inferencie modelov. Výsledky sú priemerné hodnoty desiatich generácií výstupu pre jeden príklad. Porovnávané metriky sú priemerný čas inferencie a rýchlosť inferencie, ako počet tokenov vygenerovaný za jednotku času.

Z výsledkov je zrejmé, že kvantizácia modelu zlepšila rýchlosť inferencie modelov. Nevýhodou je potenciálne zníženie kvality generovaného kódu. Rýchlosť inferencie modelu MICROCODERFIM je výrazne vyššia ako modelu MICROCODER, čo je spôsobené menším počtom parametrov a rozdielnou implementáciou hláv pozornosti. Aj tieto výsledky ukazujú, že model MICROCODERFIM je vhodný pre použitie v reálnom prostredí pri generovaní kódu a poskytovaní rýchlych nápoved programátorom.

Kapitola 8

Záver

V tejto práci som sa venoval adaptácii predtrénovaných jazykových modelov na generovanie zdrojového kódu pre aplikácie vstavaných systémov. Práca sa sústredila na vytvorenie vhodnej dátovej sady zo zdrojových súborov v jazykoch C a C++ a na následné ladenie modelov na novej dátovej sade s cieľom optimalizovať ich výkonnosť.

V úvodných kapitolách som predstavil základné princípy umelých neurónových sietí a architektúru transformer. Následne som sa venoval špecifikám použitia modelov hlbokého učenia v oblasti NLP a generovania kódu. Zdôraznil som rozdiely medzi prirodzenými jazykmi a programovacími jazykmi, pričom som rozoberal ich syntaktické a sémantické osobitosti. Následne som sa zaoberal metodikou učenia veľkých jazykových modelov, metódami evaluácie ich generatívnych schopností a popísal som niekoľko existujúcich modelov.

V implementačnej časti práce som popísal proces zberu dát a tvorbu novej dátovej sady. Vytvorená dátová sada obsahuje výber funkcií, z rôznych projektov v jazykoch C a C++, pre tréovanie jazykových modelov v oblasti vstavaných systémov. Vykonal som analýzu vzoriek z hľadiska ich veľkosti, štruktúry a navrhol spôsoby rozšírenia dátovej sady. Na vytvorenom korpuse dát som ladením vytvoril dva nové modely: MICROCODER a MICROCODERFIM, určené na generovanie kódu špecificky pre oblasť vstavaných systémov. Popísal som proces ich tréovania, nastavené parametre a uhlíkovú stopu. Pri tréovaní modelu MICROCODER som využil techniku kvantizácie váh a tréovania nízkorozmerných reprezentácií, za účelom optimalizácie výpočtových nárokov. Model MICROCODERFIM prešiel ladením všetkých váh modelu bez kvantizácie na upravenej dátovej sade technikou dopĺňania kódu v kontexte.

Výsledky evaluácie modelov ukázali, že oba modely sú schopné generovať syntakticky správny a funkčne relevantný kód. Model MICROCODERFIM exceloval v adaptácii na novú úlohu, čo dokázal výrazným zlepšením metrik po ladení. Tento model dokázal viac ako zdvojnásobiť efektívnosť pri generovaní kódu na dátovej sade po procese ladenia. Výsledky na metrike pass@k ukazujú, že MICROCODERFIM znížil schopnosť generovať kód v jazyku Python a dosiahol skóre 7.83 % v metrike pass@1 na sade HumanEval. Vyhodnotil som tiež rýchlosť inferencie modelov, pričom MICROCODERFIM dosahuje násobne väčšiu rýchlosť generovania kódu vďaka implementácií MQA.

Na konci možno konštatovať, že ciele práce boli dosiahnuté. Zoznámil som sa s využitím a adaptáciou jazykových modelov pre generovanie kódu, vytvoril som reprezentatívnu dátovú sadu komentovaných kódov a dva nové modely pre generovanie kódu. Budúce práce by mohli pokračovať v rozširovaní dátového súboru použitím navrhovaných metód. Jednou z možností je využitie smerníc MISRA-C na výber kvalitných tréovacích vzoriek.

Literatúra

- [1] AGENCY, E. E. *Glossary: Carbon dioxide equivalent* online. August 2023. Dostupné z: https://ec.europa.eu/eurostat/statistics-explained/index.php?title=Glossary:Carbon_dioxide_equivalent. [cit. 03.05.2024]. Založené na: IPCC Third Assessment Report, 2001.
- [2] AINSLIE, J.; LEE THORP, J.; JONG, M. de; ZEMLYANSKIY, Y.; LEBRÓN, F. et al. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023. Dostupné z: <https://doi.org/10.48550/arXiv.2305.13245>.
- [3] ALLAL, L. B. *Santacoder-finetuning* online. GitHub, 2023. Dostupné z: <https://github.com/loubnabnl/santacoder-finetuning>.
- [4] ALMAZROUEI, E.; ALOBEIDLI, H.; ALSHAMSI, A.; CAPPELLI, A.; COJOCARU, R. et al. *The Falcon Series of Open Language Models*. 2023. Dostupné z: <https://doi.org/10.48550/arXiv.2311.16867>.
- [5] BAVARIAN, M.; JUN, H.; TEZAK, N.; SCHULMAN, J.; MCLEAVEY, C. et al. *Efficient Training of Language Models to Fill in the Middle*. 2022.
- [6] BROWN, T. B.; MANN, B.; RYDER, N.; SUBBIAH, M.; KAPLAN, J. et al. *Language Models are Few-Shot Learners*. 2020. Dostupné z: <https://doi.org/10.48550/arXiv.2005.14165>.
- [7] CHEN, M.; TWOREK, J.; JUN, H.; YUAN, Q.; OLIVEIRA PINTO, H. P. de et al. *Evaluating Large Language Models Trained on Code*. 2021. Dostupné z: <https://doi.org/10.48550/arXiv.2107.03374>.
- [8] CHOWDHERY, A.; NARANG, S.; DEVLIN, J.; BOSMA, M.; MISHRA, G. et al. *PaLM: Scaling Language Modeling with Pathways*. 2022.
- [9] DEEPSEEK AI; ; BI, X.; CHEN, D.; CHEN, G. et al. *DeepSeek LLM: Scaling Open-Source Language Models with Longtermism*. 2024. Dostupné z: <https://doi.org/10.48550/arXiv.2401.02954>.
- [10] DETTMERS, T.; PAGNONI, A.; HOLTZMAN, A. a ZETTLEMOYER, L. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. Dostupné z: <https://doi.org/10.48550/arXiv.2305.14314>.
- [11] DEVLIN, J.; CHANG, M.-W.; LEE, K. a TOUTANOVA, K. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. Dostupné z: <https://doi.org/10.48550/arXiv.1810.0480>.

- [12] EVTIKHIEV, M.; BOGOMOLOV, E.; SOKOLOV, Y. a BRYKSIN, T. *Out of the BLEU: how should we assess quality of the Code Generation models?* 2022. Dostupné z: <https://doi.org/10.48550/arXiv.2208.03133>.
- [13] FEDUS, W.; GOODFELLOW, I. a DAI, A. M. *MaskGAN: Better Text Generation via Filling in the_____*. 2018. Dostupné z: <https://doi.org/10.48550/arXiv.1801.07736>.
- [14] GUO, D.; ZHU, Q.; YANG, D.; XIE, Z.; DONG, K. et al. *DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence*. 2024. Dostupné z: <https://doi.org/10.48550/arXiv.2401.14196>.
- [15] GÄSSLER, J. *Perplexity* online. GitHub, 2024. Dostupné z: <https://github.com/ggerganov/llama.cpp/tree/master/examples/perplexity>. [cit. 01.05.2024].
- [16] HOCHREITER, S. a SCHMIDHUBER, J. Long Short-Term Memory. *Neural Comput.* Cambridge, MA, USA: MIT Press, nov 1997, zv. 9, č. 8, s. 1735–1780. ISSN 0899-7667. Dostupné z: <https://doi.org/10.1162/neco.1997.9.8.1735>.
- [17] HU, E. J.; SHEN, Y.; WALLIS, P.; ALLEN ZHU, Z.; LI, Y. et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. Dostupné z: <https://doi.org/10.48550/arXiv.2106.09685>.
- [18] KAPLAN, J.; MCCANDLISH, S.; HENIGHAN, T.; BROWN, T. B.; CHESSE, B. et al. *Scaling Laws for Neural Language Models*. 2020. Dostupné z: <https://doi.org/10.48550/arXiv.2001.08361>.
- [19] KULAL, S.; PASUPAT, P.; CHANDRA, K.; LEE, M.; PADON, O. et al. *SPoC: Search-based Pseudocode to Code*. 2019. Dostupné z: <https://doi.org/10.48550/arXiv.1906.04908>.
- [20] LACOSTE, A.; LUCCIONI, A.; SCHMIDT, V. a DANDRES, T. Quantifying the Carbon Emissions of Machine Learning. *ArXiv preprint arXiv:1910.09700*, 2019.
- [21] LEWIS, M.; LIU, Y.; GOYAL, N.; GHAZVININEJAD, M.; MOHAMED, A. et al. *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. 2019. Dostupné z: <https://doi.org/10.48550/arXiv.1910.13461>.
- [22] LI, R.; ALLAL, L. B.; ZI, Y.; MUENNIGHOFF, N.; KOCETKOV, D. et al. *StarCoder: may the source be with you!* 2023. Dostupné z: <https://doi.org/10.48550/arXiv.2305.06161>.
- [23] LIN, C.-Y. ROUGE: A Package for Automatic Evaluation of Summaries. In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, Júl 2004, s. 74–81. Dostupné z: <https://aclanthology.org/W04-1013>.
- [24] LIU, P. J.; SALEH, M.; POT, E.; GOODRICH, B.; SEPASSI, R. et al. *Generating Wikipedia by Summarizing Long Sequences*. 2018. Dostupné z: <https://doi.org/10.48550/arXiv.1801.10198>.

- [25] MARXAV. *Full GPT architecture.png*. 2022. Dostupné z: https://commons.wikimedia.org/wiki/File:Full_GPT_architecture.png.
- [26] M.BISHOP, C. a BISHOP, H. *Deep learning : foundations and concepts*. Cham: Springer, 2024. ISBN 978-3-031-45467-7. Dostupné z: <https://doi.org/10.1007/978-3-031-45468-4>.
- [27] MCCULLOCH, W. S. a PITTS, W. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology*, 1990, zv. 52, č. 1, s. 99–115. ISSN 0092-8240.
- [28] NOVÁK, M. et al. *Umělé neuronové sítě. Teorie a aplikace*. 1. vyd. Praha: C. H. BECK, 1998. ISBN 80-7179-132-6.
- [29] PAPINENI, K.; ROUKOS, S.; WARD, T. a ZHU, W.-J. Bleu: a Method for Automatic Evaluation of Machine Translation. In: ISABELLE, P.; CHARNIAK, E. a LIN, D., ed. *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Júl 2002, s. 311–318. Dostupné z: <https://aclanthology.org/P02-1040>.
- [30] POPOVIĆ, M. ChrF: character n-gram F-score for automatic MT evaluation. In: BOJAR, O.; CHATTERJEE, R.; FEDERMANN, C.; HADDOW, B.; HOKAMP, C. et al., ed. *Proceedings of the Tenth Workshop on Statistical Machine Translation*. Lisbon, Portugal: Association for Computational Linguistics, September 2015, s. 392–395. Dostupné z: <https://aclanthology.org/W15-3049>.
- [31] POPOVIĆ, M. ChrF++: words helping character n-grams. In: BOJAR, O.; BUCK, C.; CHATTERJEE, R.; FEDERMANN, C.; GRAHAM, Y. et al., ed. *Proceedings of the Second Conference on Machine Translation*. Copenhagen, Denmark: Association for Computational Linguistics, September 2017, s. 612–618. Dostupné z: <https://aclanthology.org/W17-4770>.
- [32] POST, M. A Call for Clarity in Reporting BLEU Scores. In: BOJAR, O.; CHATTERJEE, R.; FEDERMANN, C.; FISHEL, M.; GRAHAM, Y. et al., ed. *Proceedings of the Third Conference on Machine Translation: Research Papers*. Brussels, Belgium: Association for Computational Linguistics, Október 2018, s. 186–191. Dostupné z: <https://aclanthology.org/W18-6319>.
- [33] RADFORD, A.; NARASIMHAN, K.; SALIMANS, T. a SUTSKEVER, I. *Improving Language Understanding by Generative Pre-Training*. 2018.
- [34] RAFFEL, C.; SHAZEER, N.; ROBERTS, A.; LEE, K.; NARANG, S. et al. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. 2023. Dostupné z: <https://doi.org/10.48550/arXiv.1910.10683>.
- [35] REN, S.; GUO, D.; LU, S.; ZHOU, L.; LIU, S. et al. *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*. 2020. Dostupné z: <https://doi.org/10.48550/arXiv.2009.10297>.
- [36] ROKH, B.; AZARPEYVAND, A. a KHANTEYMOORI, A. *A Comprehensive Survey on Model Quantization for Deep Neural Networks in Image Classification*. 2022. Dostupné z: <https://doi.org/10.1145/3623402>.

- [37] ROZIÈRE, B.; GEHRING, J.; GLOECKLE, F.; SOOTLA, S.; GAT, I. et al. *Code Llama: Open Foundation Models for Code*. 2024. Dostupné z: <https://doi.org/10.48550/arXiv.2308.12950>.
- [38] SCHREINER, M. *GPT-4 architecture, datasets, costs and more leaked* online. THE DECODER, 2020. Dostupné z: <https://the-decoder.com/gpt-4-architecture-datasets-costs-and-more-leaked/>. [cit. 03.08.2024].
- [39] SHAZEER, N. *Fast Transformer Decoding: One Write-Head is All You Need*. 2019. Dostupné z: <https://doi.org/10.48550/arXiv.1911.02150>.
- [40] TAKASE, S.; KIYONO, S.; KOBAYASHI, S. a SUZUKI, J. *On Layer Normalizations and Residual Connections in Transformers*. 2023.
- [41] TOUVRON, H.; MARTIN, L.; STONE, K.; ALBERT, P.; ALMAHAIRI, A. et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. Dostupné z: <https://doi.org/10.48550/arXiv.2307.09288>.
- [42] TURING, A. M. I.-COMPUTING MACHINERY AND INTELLIGENCE. *Mind*, Október 1950, LIX, č. 236, s. 433–460. ISSN 0026-4423. Dostupné z: <https://doi.org/10.1093/mind/LIX.236.433>.
- [43] VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L. et al. *Attention Is All You Need*. 2017.
- [44] VIG, J. A Multiscale Visualization of Attention in the Transformer Model. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*. Florence, Italy: Association for Computational Linguistics, Júl 2019, s. 37–42. Dostupné z: <https://www.aclweb.org/anthology/P19-3007>.
- [45] WIKIPEDIA CONTRIBUTORS. *Deep learning — Wikipedia, The Free Encyclopedia*. 2024. Dostupné z: https://en.wikipedia.org/w/index.php?title=Deep_learning&oldid=1222136552. [Online; accessed 2-May-2024].
- [46] WIKIPEDIA CONTRIBUTORS. *Perceptron — Wikipedia, The Free Encyclopedia*. 2024. Dostupné z: <https://en.wikipedia.org/w/index.php?title=Perceptron&oldid=1222140497>. [Online; accessed 2-May-2024].
- [47] WIKIPEDIA CONTRIBUTORS. *Universal approximation theorem — Wikipedia, The Free Encyclopedia*. 2024. Dostupné z: https://en.wikipedia.org/w/index.php?title=Universal_approximation_theorem&oldid=1221184532. [Online; prístup 5-máj-2024].
- [48] WIKIPÉDIA. *Neurón — Wikipédia, Slobodná encyklopédia*. 2022. Dostupné z: <https://sk.wikipedia.org/w/index.php?title=Neur%C3%B3n&oldid=7345982>. [Online; prístup 4-február-2024].
- [49] WORKSHOP, B.; ; SCAO, T. L.; FAN, A.; AKIKI, C. et al. *BLOOM: A 176B-Parameter Open-Access Multilingual Language Model*. 2023. Dostupné z: <https://doi.org/10.48550/arXiv.2211.05100>.

- [50] YANG, Z.; DAI, Z.; YANG, Y.; CARBONELL, J.; SALAKHUTDINOV, R. et al. *XLNet: Generalized Autoregressive Pretraining for Language Understanding*. 2020. Dostupné z: <https://doi.org/10.48550/arXiv.1906.08237>.
- [51] ZHAO, S. *GitHub Copilot now has a better AI model and new capabilities* online. The GitHub Blog, 14. februára 2023. Dostupné z: <https://github.blog/2023-02-14-github-copilot-now-has-a-better-ai-model-and-new-capabilities/>. [cit. 2024-19-04].
- [52] ZHENG, Q.; XIA, X.; ZOU, X.; DONG, Y.; WANG, S. et al. *CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Evaluations on HumanEval-X*. 2023.
- [53] ZHOU, Y.; CUI, S. a WANG, Y. Machine Learning Based Embedded Code Multi-Label Classification. *IEEE Access*, 2021, zv. 9, s. 150187–150200.

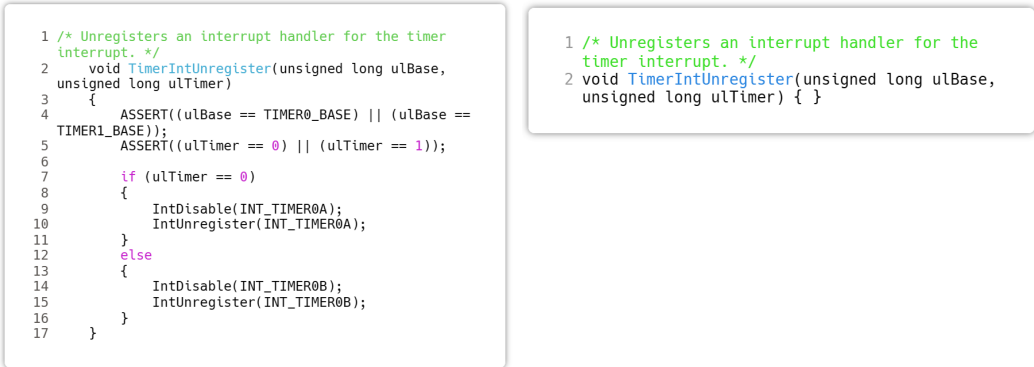
Príloha A

Ďalšie výsledky inferencie modelov

V tejto prílohe sú zobrazené ďalšie výsledky inferencie modelov MICROCODER, MICROCODERFIM a nástroja GitHub Copilot na základe zvolených príkladov. Výsledky sú zobrazené vo forme obrázkov, ktoré zobrazujú zdrojový kód vygenerovaný modelmi pre zvolené príklady. Niektoré výsledky sú zobrazené v porovnaní s referenčným zdrojovým kódom, ktorý bol použitý na tréning modelov, prípadne šedou farbou sú zvýraznené časti kódu, ktoré boli modelom vygenerované. V prípade MICROCODER a GitHub Copilot dostal model iba textové zadanie a signatúru funkcie, zatiaľ čo vstup pre MICROCODERFIM je upravený spôsobom, ktorý som opísal v sekcii 7.3.2.

A.1 MicroCoder

Na obrázku A.2 je vidieť, že výsledky modelu MICROCODER sa pre rovnaký vstup významne líšia. Kvantizácia váh má vplyv na „zmätenosť“ (angl. perplexity) modelu a tým aj na jeho schopnosť generovať zmysluplné výstupy. Niekedy model nevygeneruje nič zmysluplné, inokedy je výstup prínosný – obrázok A.1. Miera zmätenosti modelu sa zvyšuje s klesajúcou bitovou hĺbkou modelu [15]. Na základe toho, že kvantizácia ovplyvňuje zmätenosť modelu, je dôležité posúdiť, či takáto optimalizácia je vhodná pre špecifické použitie modelu. Pri aplikáciách, kde je kritická vysoká presnosť a komplexnosť generovaného kódu, môže byť kvantizácia kontraproduktívna.



```
1 /* Unregisters an interrupt handler for the timer
interrupt. */
2 void TimerIntUnregister(unsigned long ulBase,
unsigned long ulTimer)
3 {
4     ASSERT((ulBase == TIMER0_BASE) || (ulBase ==
TIMER1_BASE));
5     ASSERT((ulTimer == 0) || (ulTimer == 1));
6
7     if (ulTimer == 0)
8     {
9         IntDisable(INT_TIMER0A);
10        IntUnregister(INT_TIMER0A);
11    }
12    else
13    {
14        IntDisable(INT_TIMER0B);
15        IntUnregister(INT_TIMER0B);
16    }
17 }
```

```
1 /* Unregisters an interrupt handler for the
timer interrupt. */
2 void TimerIntUnregister(unsigned long ulBase,
unsigned long ulTimer) { }
```

Obr. A.1: Dva rôzne výstupy modelu MICROCODER na rovnaký dotaz s použitím rovnakých parametrov.

```

1 /* Configure the Break feature, dead time, Lock level, the IMOS. */
2 void TMR_ConfigBDT(TMR_T *tmr, TMR_BDTConfig_T *BDTConfig)
3 {
4     tmr->BCR = BDTConfig->BreakCmd;
5     tmr->BDTR = (u32)(BDTConfig->DeadTime | \
6                   ((u32)BDTConfig->LockLevel << 8U) | \
7                   BDTConfig->IM);
8 }

```

```

1 /* Configure the Break feature, dead time, Lock level, the IMOS. */
2 void TMR_ConfigBDT(TMR_T *tmr, TMR_BDTConfig_T *BDTConfig)
3 {
4     u32 tmpVal = (tmr->CTRL1 & ~0x7F);
5     tmpVal |= BDTConfig->BreakState;
6     if((BDTConfig->DeadTimeEn == ENABLE)) {
7         tmr->BDTR &= ~(TMROCOMP_DEADTIMEENA|TMCNT_RELOADVALU);
8         if ((tmr==TIMER4)&(BDTConfig->BreakState!=DISABLE)){
9             while(!(READ_BITS(SYSREG->SOFTRESINDICATOR[5], SYSREG_SOFTRESINDI_PLCLKSEL)));
10        }else{
11            WRITE_PERI_REG(&tmr->CNTVALUE, 0UL);
12        }
13        WRITE_PERI_REG(&tmr->CNTOUTPUT, RESET_VALUE);
14        tmr->BDTR |= (BDTConfig->DutyValue<<8)|(BDTConfig->CompareMatchValue << 16);
15        SET_FIELD(tmpVal, TMROCOMP_COMPARATORENABLE, DISABLE);
16        SET_FIELD(tmpVal, TMRACTIONTYPE, ACTION_NOACTION);
17        CLEAR_PERI_REG_MASK(tmr->INTSTS, TMRIE_OUTPUTCHANNELCOMPIE);
18    } else {
19        tmr->BDTR &= ~(TMROCOMP_DEADTIMEENA|TMCNT_RELOADVALU);
20        tmr->BDTR |= BDTConfig->CompareMatchValue << 16 ;
21        SET_FIELD(tmpVal, TMROCOMP_COMPARATORENABLE, ENABLE);
22        SET_FIELD(tmpVal, TMRACTIONTYPE, BDTConfig->ActionType);
23        SET_PERI_REG_MASK(tmr->INTSTS, TMRIE_OUTPUTCHANNELCOMPIE);
24    }
25    WRITE_PERI_REG(tmr->CTRL1, tmpVal);
26 }

```

Obr. A.2: Dva rôzne výstupy modelu MICROCODER na rovnaký dotaz s použitím rovnakých parametrov. Na prvý pohľad je vidno, že dané výstupy sú veľmi odlišné. Prvý výstup (hore) je jednoduchší a priamočiarejší. Definuje dva hlavné príkazy pre nastavenie registra. Druhý výstup (dolu) je zložitejší, obsahuje viac vetvení a kontrolných štruktúr.

```

1 /* Computes a cycle count for a given time in nanoseconds. */
2 static uint32_t LPI2C_GetCyclesForWidth(uint32_t sourceClock_Hz, uint32_t width_ns, uint32_t maxCycles, uint32_t prescaler) {
3     assert(width_ns != 0U);
4     if (prescaler == 1675UL || ((sourceClock_Hz / prescaler) <= kMaxFreq))
5         return (((float)(sourceClock_Hz * width_ns) + (kLpi2cClksPerMicroSec - 1ULL))) / kLpi2cClksPerMicroSec;
6     else
7         return (((float)(sourceClock_Hz >> prescaler) * width_ns) + (kLpi2cClksPerMicroSec - 1ULL)) / kLpi2cClksPerMicroSec;
8 }
9 /*! @brief Get the I2CSCL divisor to set frequency of SDA clock.*/
10 void LPI2C_MasterSetBaudRate(I2C_Type *base, const lpi2c_master_config_t *cfg, uint32_t srcClock_Hz) {
11     base->SCFSR = 0xFFu << LPI2C_SCFSR_SCLSMFREQSEL_SHIFT | cfg->baudrate
12     while (!(base->SCFSR & LPI2C_SCFSR_VLD_MASK)) {}
13 }

```

Obr. A.3: Výstup modelu MICROCODER.

A.2 Github Copilot

```
1 /* This routine will start the transmit DMA running if it is not already running. This routine ned only be called
   when queueing a new packet to an empty tx queue. Writes the current tx chain head as start address of the tx
   descriptor chain and enables the transmission DMA engine. */
2 static void spider_net_kick_tx_dma(struct spider_net_card *card){
3     struct spider_net_descr_chain *chain = card->tx_chain;
4     struct spider_net_descr *descr = chain->head;
5     u32 status;
6
7     /* If the DMA is already running, we don't need to do anything */
8     if (chain->running)
9         return;
10
11    /* If the tx chain is empty, we don't need to do anything */
12    if (descr == NULL)
13        return;
14
15    /* Set the start address of the tx descriptor chain */
16    writel((u32)descr->dma_addr, card->regs + SPIDER_NET_TX_DESCR_LIST);
17
18    /* Enable the transmission DMA engine */
19    status = readl(card->regs + SPIDER_NET_DMA_STATUS);
20    status |= SPIDER_NET_DMA_TX_ENABLE;
21    writel(status, card->regs + SPIDER_NET_DMA_STATUS);
22
23    chain->running = 1;
24 }
```

Obr. A.4: Kód vygenerovaný nástrojom GitHub Copilot.

```
1 /* Write a "service pulse" to the AD5755 watchdog timer when enabled. */
2 void ad5755_feed_watch_dog_timer(struct ad5755_dev *dev) {
3     uint16_t data = 0;
4     data = AD5755_CTRL_REG_PD | AD5755_CTRL_REG_PD | AD5755_CTRL_REG_PD | AD5755_CTRL_REG_PD;
5     ad5755_write(dev, AD5755_CTRL_REG, data);
6 }
```

Obr. A.5: Kód vygenerovaný nástrojom GitHub Copilot. Kód nebol vygenerovaný naraz, ale postupne na tri rôzne nápovedy.

```
1 /* Configure the Break feature, dead time, Lock level, the IMOS. */
2 void TMR_ConfigBDT(TMR_T *tmr, TMR_BDTConfig_T *BDTConfig) {
3     /* Configure the Break feature, dead time, Lock level, the OSSSI, the OSSR State and the BKE, AOE and
   BKP. */
4     tmr->BDTR = (BDTConfig->Break << 10) | (BDTConfig->DeadTime << 8) | (BDTConfig->LockLevel << 7) |
   (BDTConfig->OSSSI << 5) | (BDTConfig->OSSR << 2) | (BDTConfig->BKE << 10) | (BDTConfig->AOE << 8) |
   (BDTConfig->BKP << 1);
5     /* Configure the Break polarity, dead time, Lock level, the IMOS. */
6     tmr->BDTR |= (BDTConfig->BreakPolarity << 13) | (BDTConfig->IMOS << 11);
7 }
```

Obr. A.6: Kód vygenerovaný nástrojom GitHub Copilot. Model bol inštruovaný, aby nakonfiguroval rôzne parametre, ako sú Break feature, dead time a Lock level. Výstup efektívne odráža tieto požiadavky.

A.3 MicoCoderFIM

```
1 /* Enables or disables the SS output for the selected SPI port. */
2 void SPISSOutputConfig(unsigned long ulBase, unsigned long ulSSOutput)
3 {
4     xASSERT((ulBase == SPI3_BASE) || (ulBase == SPI1_BASE) ||
5             (ulBase == SPI2_BASE));
6     xASSERT((ulSSOutput == SPI_SSOE_ENABLE) ||
7             (ulSSOutput == SPI_SSOE_DISABLE));
8     if ((ulSSOutput != SPI_SSOE_ENABLE))
9     {
10        return ;
11    }
12    while (!(xHWREG(ulBase + SPI_SR) & SPI_SR_TXE))
13    {
14    };
15    xHWREG(ulBase + SPI_DR) = SPI_SS_ACTIVE;
16    xHWREG(ulBase + SSOutput == SPI_SSOE_DISABLE));
17    xHWREG(ulBase + SPI_CR2) &= ~SPI_CR2_SSOE;
18    xHWREG(ulBase + SPI_CR2) |= ulSSOutput;
19 }
```

Obr. A.7: Výstup modelu MICROCODERFIM.

```
1 /* This routine is a wrapper for lower-level driver's revalidate_disk call-backs. It is
2 used to do common pre and post operations needed for all revalidate_disk operations. */
3 int revalidate_disk(struct gendisk *disk){
4     struct block_device *bdev = disk->private_data;
5     int ret;
6     ret = __revalidate_disk(disk);
7     if (unlikely(!ret))
8         return 0;
9     WARN_ONCE(1, KERN_ERR "Block device %llu has failed sanity checks.\n",
10              ((unsigned long long)bdev->bd_dev << 32));
11     bdget(bdev);
12     INIT_LIST_HEAD(&bdev->queue);
13     init_waitqueue_head(&bdev->wait);
14     mutex_lock(&bdev->bd_mutex);
15     check_disk_size_change(disk, bdev);
16     mutex_unlock(&bdev->bd_mutex);
17     bdpup(bdev);
18     return ret;
19 }
```

Obr. A.8: Výstup modelu MICROCODERFIM.

```

1 /* Sets the size of the internal buffer. See
2 G_SEEKABLE_BUFFERED_INPUT_STREAM_GET_BUFFER(). */
3 void g_seekable_set_buffer_size(GSeekable *stream, goffset
4 size){
5     GBufferedInputStreamPrivate *priv;
6     g_return_if_fail (G_IS_SEEKABLE (stream));
7     if (size == stream->priv->cur)
8         return;
9     priv = stream->priv;
10    if (!priv->use_buffering || !priv->buffer)
11    {
12        priv->pos = 0;
13        priv->len = size;
14    }
15    else
16    {
17        priv->buffer = g_malloc (size);
18        priv->len = size;
19        priv->pos = 0;
20    }
21    g_object_notify (G_OBJECT (stream), "buffer-size");

```

```

1 /* Sets the size of the internal buffer to @size. */
2 void
3 g_buffered_output_stream_set_buffer_size(GBufferedOutp
4 utStream *stream, gsize size)
5 {
6     GBufferedOutputStreamPrivate *priv;
7     guint8 *buffer;
8     g_return_if_fail (G_IS_BUFFERED_OUTPUT_STREAM
9 (stream));
10    priv = stream->priv;
11    if (size == priv->len)
12        return;
13    if (priv->buffer)
14    {
15        size = MAX (size, priv->pos);
16        buffer = g_malloc (size);
17        memcpy (buffer, priv->buffer, priv->pos);
18        g_free (priv->buffer);
19        priv->buffer = buffer;
20        priv->len = size;
21    }
22    else
23    {
24        priv->buffer = g_malloc (size);
25        priv->len = size;
26        priv->pos = 0;
27    }
28    g_object_notify (G_OBJECT (stream), "buffer-size");

```

Obr. A.9: Návrh kódu vygenerovaný modelom MICROCODERFIM (vľavo) a referenčné riešenie (vpravo).

```

1 /* If this interface is not supported, then return FALSE. */
2 BOOLEAN EFIAPI Sha256HashAll(IN CONST VOID *Data, IN UINTN
3 DataSize, OUT UINT8 *HashValue)
4 {
5     SHA256_CTX Context;
6     if (HashValue == NULL) {
7         return FALSE;
8     }
9     if ((DataSize > INT_MAX) ||
10        (Data == NULL && DataSize != 0))
11    {
12        return FALSE;
13    }
14    if (!SHA256_CreateCtx (NULL, &Context)) {
15        return FALSE;
16    }
17    if (!SHA256_Update (&Context, Data, DataSize)) {
18        return FALSE;
19    }
20    if (!SHA256_Final (HashValue, &Context)) {
21        return FALSE;
22    }
23    CopyMem (HashValue + sizeof (UINT32), &EfiCrc32_Init
24 (&Context)) {
25        return FALSE;
26    }
27    if (!SHA256_Update (&Context, Data, DataSize)) {
28        return FALSE;
29    }
30    if (!SHA256_Final (HashValue, &Context)) {
31        return FALSE;
32    }
33    return TRUE;
34 }

```

```

1 /* If this interface is not supported, then return
2 FALSE. */
3 BOOLEAN EFIAPI Sha256HashAll(IN CONST VOID *Data,
4 IN UINTN DataSize, OUT UINT8 *HashValue)
5 {
6     SHA256_CTX Context;
7     if (HashValue == NULL) {
8         return FALSE;
9     }
10    if ((Data == NULL) && (DataSize != 0)) {
11        return FALSE;
12    }
13    if (!SHA256_Init (&Context)) {
14        return FALSE;
15    }
16    if (!SHA256_Update (&Context, Data, DataSize)) {
17        return FALSE;
18    }
19    if (!SHA256_Final (HashValue, &Context)) {
20        return FALSE;
21    }
22    return TRUE;
23 }

```

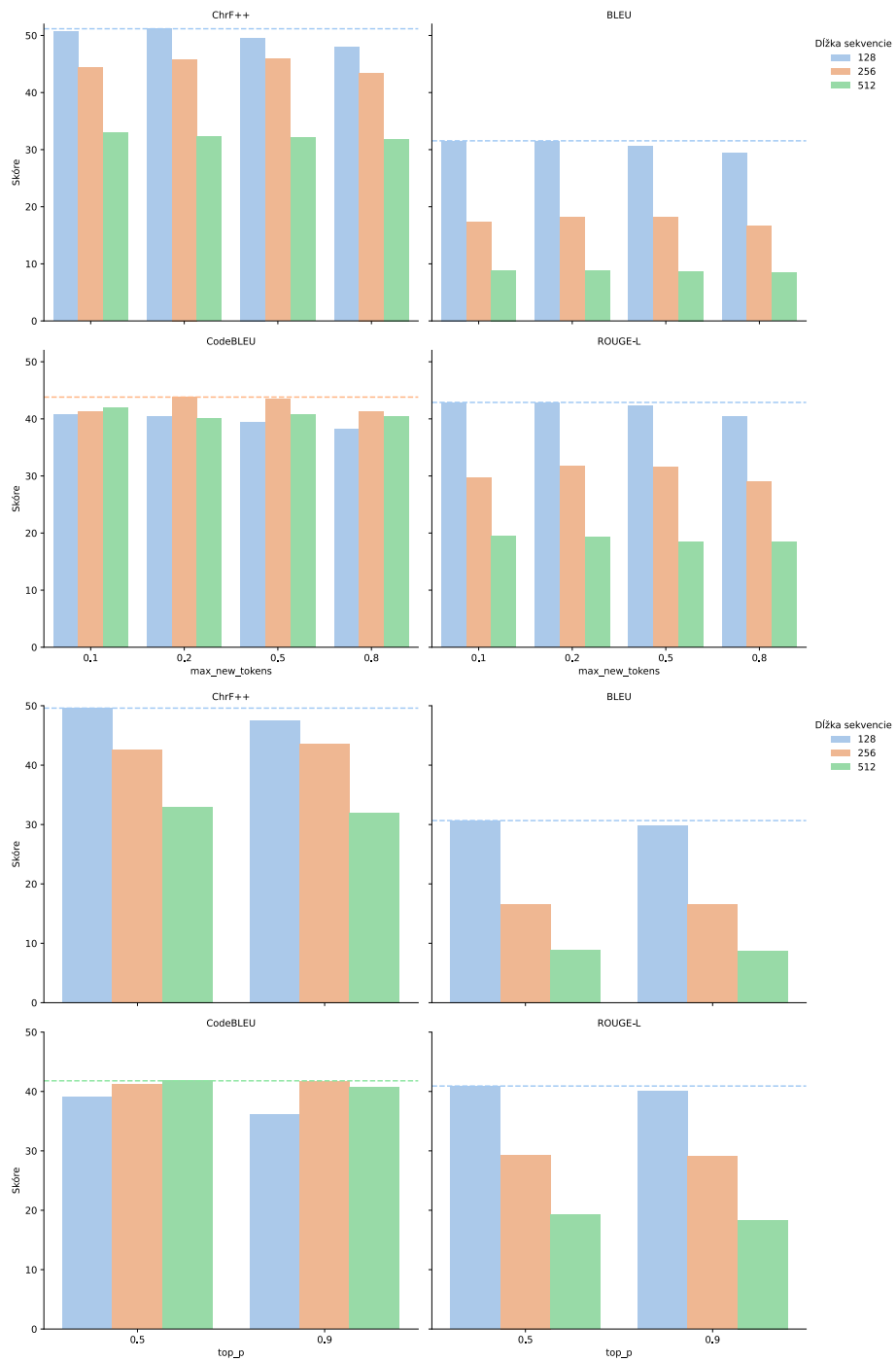
Obr. A.10: Návrh kódu vygenerovaný modelom MICROCODERFIM (vľavo) a referenčné riešenie (vpravo).

Príloha B

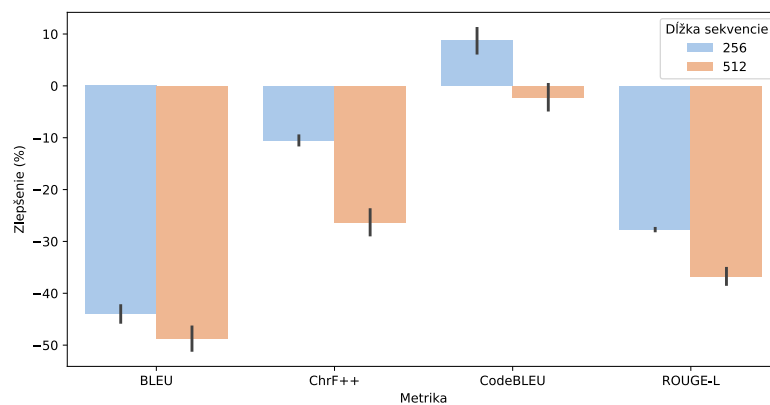
Výsledky modelu MicroCoderFIM pri rôznych nastaveniach

Pre zistenie optimálnych nastavení parametrov teploty (temperature), maximálnej dĺžky generovanej sekvencie (max_new_tokens), a pravdepodobnosti výberu (top_p), bola vykonaná sada experimentov na vzorke 250 príkladov. Analýza ukázala, že model dosahuje najlepšie celkové výsledky pri nastavení nižšej teploty (0.2 a 0.1) a maximálnej dĺžke sekvencie 128 tokenov. Pre metriku CodeBLEU však model vykazoval lepšie výsledky pri dlhších sekvenciách (256 tokenov), čo naznačuje odlišnú dynamiku optimalizácie pre túto špecifickú metriku. Výsledky pre rôzne nastavenia teploty a maximálnej dĺžky generovanej sekvencie sú podrobne dokumentované v grafoch, ktoré ilustrujú závislosti medzi meniacimi sa hodnotami týchto parametrov a výkonomi modelu. Hodnotenie parametru top_p ukazuje, že model mal lepšie výsledky pri hodnote 0.5 ako pri 0.9, ale pri teplote 0.2 vykazuje najlepšie skóre. Podrobné výsledky sú na obrázku [B.1](#).

Obrázok [B.2](#) ukazuje, že obidva testy modelujú rovnaké chovanie nastavenia maximálnej dĺžky generovania. Pri nižších sekvenciách model vykazuje lepšie výsledky. Zároveň sa ukázalo, že pri metrike CodeBLEU dosahuje model o takmer 10% lepšie výsledky pri nastavení sekvencie 256 tokenov.



Obr. B.1: Výsledky inferencie modelu MICROCODERFIM pri rôznych nastaveniach teploty, top_p a maximálnej dĺžky generovanej sekvencie. Model dosiahol najlepšie výsledky pri teplote 0.2 a maximálnej dĺžke 128 tokenov.



Obr. B.2: Zmena výsledkov inferencie modelu MICROCODERFIM v porovnaní so základným nastavením (128 tokenov) pri rôznych hodnotách maximálnej dĺžky generovanej sekvencie. Platí, že pri dĺžke 128 tokenov sa model správa lepšie v troch metrikách. Metrika CodeBLEU vykazuje lepšie výsledky pri dĺžke 256 tokenov.