FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

Neuronové sítě — porovnání výkonnosti knihovny založené na PyTorch v Pythonu a C++

Matěj Černý

PLZEŇ

2023

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

Neuronové sítě — porovnání výkonnosti knihovny založené na PyTorch v Pythonu a C++

Matěj Černý

Vedoucí práce
Ing. Martin Prantl, Ph.D.

PLZEŇ 2023

Citace v seznamu literatury:
ČERNÝ, Matěj. *`Neuronové sítě` — porovnání výkonnosti knihovny založené na PyTorch v Pythonu a C++*. Plzeň, 2023. Bakalářská práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Vedoucí práce Ing. Martin Prantl, Ph.D.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Matěj ČERNÝ**
Osobní číslo:	**A20B0074P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Neuronové sítě – porovnání výkonnosti knihovny založené na PyTorch v Pythonu a C++**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s knihovnou PyTorch a neuronovými sítěmi. Seznamte se s existující knihovnou vytvořenou vedoucím práce.
2. Implementujte (a modifikujte) danou Python knihovnu v C++.
3. Ověřte funkčnost implementované C++ knihovny na vhodně zvoleném modelu.
4. Proveďte sadu měření výkonu na tomto modelu v obou knihovnách a zjištěné výsledky zhodnoťte.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce

Vedoucí bakalářské práce: **Ing. Martin Prantl, Ph.D.**
Nové technologie pro informační společnost

Datum zadání bakalářské práce: **3. října 2022**
Termín odevzdání bakalářské práce: **4. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 25. října 2022

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného akademického titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Západočeská univerzita v Plzni má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Plzni dne 2. května 2023

. . . . . .
Matěj Černý

V textu jsou použity názvy produktů, technologií, služeb, aplikací, společností apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Abstrakt

Neuronové sítě jsou v současné době velmi populární. Naprostá většina kódů je nicméně psaná v jazyce Python, například s použitím knihovny PyTorch. Jádro této knihovny je ovšem nativně psáno v C++ a umožňuje použití přímo z C++. Cílem této bakalářské práce je konverze knihovny vyvinuté vedoucím práce založené na PyTorch z Pythonu do C++ a následné porovnání výkonu obou řešení na jednoduchém modelu. Výsledky naznačují zanedbatelný vliv výběru jazyka na výkon.

Abstract

Neural networks are currently very popular. However, the vast majority of code is written in the Python language, for example using the PyTorch library. The core of the library is, however, natively written in C++ and allows for direct use from C++. The aim of this bachelor's thesis is to convert the PyTorch-based library developed by the thesis supervisor from Python to C++, and subsequently compare the performance of both solutions on a simple model. The results suggest negligible impact of language choice on performance.

Klíčová slova

strojové učení • frameworky pro strojové učení • PyTorch • LibTorch • Python • C++

Poděkování

Na tomto místě bych rád poděkoval Ing. Martinu Prantlovi, Ph.D. za odborné vedení, cenné rady a bezpodmínečnou podporu 24/7. Také děkuji Ing. Kamilu Ekšteinovi, Ph.D. za skvělou LaTeXovou šablonu.

Obsah

Úvod 1

V současnosti jsou metody strojového učení využívány napříč mnoha odvětvími. S rostoucím využitím těchto metod vzniká také tlak na usnadnění návrhu a práce s neuronovými sítěmi. Tento tlak zapříčinil vznik mnoha frameworků urychlujících přechod od návrhu jednotlivých modelů k jejich produkčnímu nasazení. Tyto frameworky mj. nabízí mnoho základních stavebních bloků pro modelování neuronových sítí a poskytují abstrakci od implementačních detailů jako je například výpočet gradientů, či paralelizace výpočtu pomocí jazyka CUDA. Mezi hlavní představitele lze zařadit PyTorch a TensorFlow. Typickým jazykem pro práci je u obou frameworků Python.

Využití Pythonu pro návrh modelů přináší řadu výhod pro vývojáře: nabízí plně objektový přístup, rozsáhlý ekosystém populárních nástrojů pro vizualizaci a práci s daty (např. knihovny Matplotlib, Numpy) a mimo to je také díky své jasné a stručné syntaxi ceněn i mezi začátečníky. Na druhou stranu je tento jazyk interpretovaný a vyžaduje často dynamickou alokaci paměti, což se může negativně projevit na celkové rychlosti běhu.

Jazyk C++ stejně jako Python podporuje objektový i procedurální přístup. Na rozdíl od Pythonu je však C++ kompilovaný jazyk, který nabízí vyšší kontrolu nad CPU a správou paměti. Díky tomu dosahuje výrazně vyšší rychlosti běhu. V tomto jazyce jsou také napsána jádra již zmíněných neuronových frameworků.

Jazyk Python tvoří hlavní rozhraní frameworku PyTorch. Nicméně tento framework nabízí i C++ frontend. Stejně jako Python frontend, i C++ frontend rozšiřuje rozsáhlé C++ jádro. Toto jádro nabízí základní datové struktury a funkčnosti, jako jsou například tensory a automatický výpočet gradientů. Python frontend i C++ frontend mají vestavěnou kolekci běžných komponent pro modelování neuronových sítí, rozhraní pro přidání vlastních modulů a mj. také knihovnu populárních optimalizačních algoritmů.

Předmětem této bakalářské práce je převod knihovny vyvinuté vedoucím práce založené na PyTorch z Pythonu do C++ a následné testování, zdali využití jiného jazyka vede ke zvýšení výkonu. Toto testování bude provedeno porovnáním obou řešení (tedy Python verze a C++ verze) na jednoduchém modelu.

Neuronové sítě 2

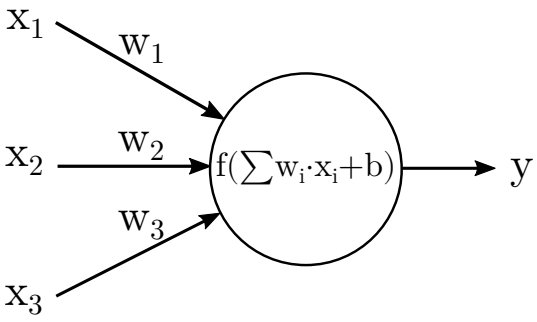
Umělé neuronové sítě jsou prostředkem strojového učení založeného na biologických neuronových sítích. Tyto sítě se snaží napodobit chování skutečných biologických sítí při řešení různých úloh, jako je např. zpracovávání jazyka, rozpoznávání obrazu a dalších komplexních problémů, pro které neexistuje vhodné analytické řešení. Umělá neuronová síť je systémem se vstupy a výstupy, který je tvořen větším počtem podobných, vzájemně propojených jednotek. Tyto jednotky mají větší počet vnitřních parametrů nazývaných váhy. Změnou těchto parametrů lze ovlivnit chování jednotky, a, z důvodu vzájemného propojení, také chování celé sítě. Cílem je zvolit parametry tak, aby bylo dosaženo požadované konverze vstupu na výstup. Tento proces změny parametrů se nazývá učení. [1]

První kroky k umělým neuronovým sítím (AANs - Artificial Neural Networks) byly podniknuty již v roce 1943, když Warren McCulloch a Walter Pitts vyvinuli první modely neuronových sítí. V následujících letech rostla jejich komplexita a začalo se experimentovat s vícevrstvými modely. Mezi roky 2009 a 2012 vyvinula výzkumná skupina Juergena Schmidhubera rekurentní a hluboké neuronové sítě. V současné době se nejen díky dostupnosti vyššího výpočetního výkonu staly neuronové sítě významným nástrojem napříč mnoha odvětvími. [2]

2.1 Struktura neuronu

Základní jednotkou biologických neuronových sítí je neuron. Jedná se o specializovanou buňku schopnou přenášet elektrické a chemické signály. Lidský mozek např. obsahuje miliardy neuronů vzájemně propojených pomocí synapsí. [3]

Umělý neuron je matematicky zjednodušeným modelem biologického neuronu. Tento neuron je reprezentován váhami (citlivostmi) jednotlivých vstupů a přechodovou funkcí, přičemž výstup této funkce je také výstupem samotného neuronu (obr. 2.1). Výstup umělého neuronu se může větvit a vést jako vstup do více dalších neuronů. Model neuronu lze popsat následující funkcí:



Obrázek 2.1: Model neuronu

$$y = f(\sum_{i=1}^{n} w_i x_i + b) \tag{2.1}$$

V této rovnici y reprezentuje výstup neuronu, f je aktivační funkce, w_i jsou váhy vstupů x_i, b je bias a n je počet vstupů.

2.2 Typy neuronových sítí

Jeden rozdíl, kterým se umělé neuronové sítě liší od svých biologických protějšků, je způsob propojení. Ve srovnání s komplexní strukturou biologických sítí jsou umělé neuronové sítě obvykle uspořádány do vrstev. V závislosti na konkrétním zapojení lze rozlišovat mnoho typů, každý s jiným případem využití.

2.2.1 Feedforward sítě

Ve Feedforward sítích informace směřují ze vstupní vrstvy, skrze jednu či více skrytých vrstev až k výstupní vrstvě, bez smyček/zpětné vazby. Typickým zástupcem této třídy je Multi-Layer Perceptron (MLP). U tohoto typu platí, že je každý neuron z vrstvy propojen se všemi neurony vrstvy předchozí, tedy model je tzv. Fully-Connected. Feedforward sítě lze použít pro širokou škálu úloh včetně klasifikace a regrese. Slouží také jako základ složitějších architektur, jako jsou např. rekurentní (RNN) a konvoluční (CNN) neuronové sítě.

Frameworky strojového učení

3

3.1 PyTorch

PyTorch [4] je open-source framework pro strojové učení nabízející tenzorové výpočty s podporou hlubokých neuronových sítí s GPU akcelerací. Původně byl vyvinut společností Facebook v roce 2017, v současnosti již však patří pod Linux Foundation. Je založen na dnes již zastaralé knihovně Torch, která jako primární rozhraní používala jazyk Lua. PyTorch si získal popularitu díky své flexibilitě, jednoduchosti použití a funkci dynamického výpočetního grafu, vedoucí k efektivnímu debugování a vývoji. Mimo to disponuje PyTorch rozsáhlou a rozrůstající se komunitou.

3.1.1 Popis PyTorch

PyTorch umožňuje snadný návrh a trénování neuronových sítí. Jednou z předností je funkce automatického výpočtu gradientů (autograd) při procesu učení. To znamená, že na základě výsledku ztrátové (loss) funkce jsou spočítány gradienty pro jednotlivé naučitelné parametry sítě, které jsou následně s využitím optimalizátoru aktualizovány. Tento proces je automatizován díky využití dynamických výpočetních grafů (Dynamic Computational Graphs). PyTorch si při dopředném šíření (Forward Propagation) automaticky vytváří graf operací s tensory, na jehož základě je poté schopen vypočítat gradienty jednotlivých parametrů (Backpropagation algoritmus). Pro získání této funkcionality stačí pouze označit tensory příznakem `requires_grad`. Výhoda dynamických výpočetních grafů oproti statickým spočívá ve vytvoření nového grafu při každém výpočtu gradientů, to umožňuje měnit síť za běhu, čehož může být využito u dynamických architektur jako jsou např. rekurentní neuronové sítě.

Jádro PyTorch je napsáno v C++, primárním rozhraním je však jazyk Python. Pro snazší integraci s již existujícím C++ ekosystémem nabízí PyTorch také nové C++ rozhraní: Libtorch. Obě tyto rozhraní (frontendy) disponují kolekcí běžných komponent pro modelování neuronových sítí a dalšími pomocnými funkcemi.

3.1.2 Omezení LibTorch

Jelikož je LibTorch sekundárním rozhraním pro práci s PyTorch, není mu věnována ve srovnání s Python frontendem taková pozornost. Určité funkce, které Python frontend nabízí, v LibTorch nenalezneme. Jednou funkcí, kterou LibTorch frontend postrádá, je Automatic Mixed Precision (AMP), resp. postrádá její část.

Automatic Mixed Precision je technika, při které se využívají pro určité operace desetinná čísla s nižší přesností, například 16bitová čísla místo standardních 32bitových. Tato technika, která se standardně aplikuje při trénování, obvykle vede ke zvýšení výkonu a snížení paměťové náročnosti. V PyTorch tuto funkci nabízí balíček `amp`. Ten obsahuje manažer zvaný autocast, zodpovědný za samotné AMP, a GradScaler, který se stará o správné škálování gradientů při zpětném šíření (učení). LibTorch v minulých verzích neoficiálně disponoval alespoň funkcí autocast, v nejnovější verzi PyTorch 2.0 však nenabízí autocast ani ekvivalent objektu GradScaler. V současnosti tedy Automatic Mixed Precision techniku pro trénování nelze v LibTorch použít.

Další chybějící funkcí je `torch.utils.tensorboard`, což je balíček umožňující ukládání trénovacích a testovacích metrik v nativním tensorboard formátu pro snadnější vizualizaci později.

Nakonec také systém profilování[1] je v LibTorch omezený. Zatímco Python verze používá objekt PyTorch Profiler, LibTorch obsahuje profilery Legacy a Kineto, které jsou ovšem špatně zdokumentované. Pro obě dvě verze lze však použít externí nástroj NVIDIA Nsight Systems [5].

3.2 TensorFlow

TensorFlow je konkurentem frameworku PyTorch. Byl vyvinut společností Google v roce 2015. Stejně jako u PyTorch se jedná o open-source framework s rozsáhlým ekosystémem a širokou komunitou. Také disponuje funkcí automatického výpočtu gradientu, narozdíl však od PyTorch, který používá dynamické výpočetní grafy pro následné provedení Backpropagation algoritmu, jsou výpočetní grafy u TensorFlow statické. To nutí uživatele k explicitní specifikaci grafu před samotným výpočtem s využitím placeholder proměnných[2]. Následně může být graf proveden vícekrát s rozdílnými daty. Tato zdánlivá nevýhoda umožňuje frameworku optimalizovat výpočet grafu, což vede ke zvýšení efektivity, která je na druhou stranu vykoupena snížením flexibility, jelikož není možno graf za běhu měnit. Pro vývoj dynamických

[1] Profilování je proces měření a analýzy výkonu programu za účelem identifikace úseků kódu s nejvyšší časovou náročností.

[2] Placeholder proměnné představují symbolické proměnné, které slouží jako zástupné prvky pro skutečné hodnoty vstupních dat při výpočtu.

architektur, kdy je nutno grafy za běhu měnit, lze využít separátní knihovnu TensorFlow Fold. Další nadstavbu tvoří Keras, vysokoúrovňové open-source API běžící nad TensorFlow usnadňující vývoj a trénování neuronových modelů. Jádro TensorFlow je stejně jako PyTorch napsáno v C++. Narozdíl od PyTorch TensorFlow nabízí více rozhraní: Python, C++, Java, Javascript. Python však zůstává primárním rozhraním u obou frameworků.

3.3 Využití PyTorch a TensorFlow

PyTorch i TensorFlow disponují rozsáhlými komunitami, které se ovšem liší přístupem a zaměřením. PyTorch má více akademicky zaměřenou komunitu, která si cení flexibility, jednoduchosti návrhu a intuitivity API (například výhoda dynamických výpočetních grafů). Z toho také plyne vyšší dostupnost modelů, a to jak v podobě výzkumných prací [6], tak v repozitáři předtrénovaných modelů HuggingFace [7]. TensorFlow je na druhou stranu užíván spíše v komerční sféře. Komunita se zaměřuje především na nasazení a škálování na produkční úrovni. Nutno ovšem zmínit, že toto rozdělení komunit není výlučné, a oba frameworky nyní nabízí rozsáhlý ekosystém včetně nástrojů pro nasazení. [8]

Python a C++ 4

4.1 Rozdíly Pythonu a C++ a jejich interoperabilita

Jeden podstatný rozdíl plynoucí z odlišnosti jazyků je typování. Python patří k dynamicky typovaným jazykům, které umožňují definovat proměnné bez nutnosti určení jejich datového typu. Datový typ se stanovuje až při vykonávání programu. Dynamické typování také umožňuje snadnou refaktorizaci kódu, protože kód nemusí být aktualizován, pokud se změní datový typ proměnné. C++ je oproti Pythonu staticky typované, kde se typy proměnných určují během kompilace a nelze je za běhu měnit. Toto omezení umožňuje předcházet mnoha chybám v programu již při překladu, což vede k větší spolehlivosti a bezpečnosti programů.

Dalším rozdílem je přístup ke správě paměti. Python patří ke Garbage Collected jazykům, kde jsou nepoužívané objekty, tedy objekty, na které neukazují žádné reference, automaticky uvolňovány. V C++ je na druhou stranu nutno tyto objekty uvolňovat explicitně, případně automaticky s využitím chytrých (smart) ukazatelů.[1]

Jednoduchost a vyšší flexibilita Pythonu v porovnání s C++ je ovšem vykoupena obecně nižším výkonem. První možností, jak tento problém řešit, je implementovat výkonný kód v C++ a následně jej navázat do Pythonu např. s využitím nástroje Cython [9] nebo Nanobind [10]. Výsledkem je možnost volání nativního C/C++ kódu přímo z Pythonu. Tento proces lze do jisté míry automatizovat využitím automatických wrapper generátorů (např. SWIG), které vygenerují Python wrapper moduly kolem C/C++ kódu. Tímto způsobem lze zpřístupnit z Pythonu celé C/C++ knihovny. Nevýhodou této metody je ovšem stálá závislost na Python prostředí.

Druhou možností je využít kompilátory (např. Cython), které Python kód na ekvivalentní C/C++ kód převedou. Tím sice odpadá závislost na Python prostředí, na druhou stranu může být vygenerovaný kód hůře čitelný a náročný na modifikaci.

[1] C++ umožňuje definovat u objektů destruktory, které jsou automaticky volány při jejich destrukci a mohou dealokovat další paměť, která je s objektem spjata. Moderní C++ práci s paměti velmi usnadňuje.

4.2 Adaptace Python konvencí v C++

Přestože jsou Python i C++ objektově orientované, konverze mezi nimi není zcela triviální. Jedním z hlavních úskalí při převodu libovolné Python knihovny do C++, pokud chceme zachovat její API, je adaptace typového systému. Pythonské knihovny disponují obvykle metodami jak s parametry s proměnlivými typy, tak proměnlivým počtem pojmenovaných parametrů. Jako příklad je uvedena deklarace Python metody `modify_data`:

Zdrojový kód 4.1: Python, Metoda s proměnlivými typy parametrů

```
def modify_data(data:Union[Tensor,np.array], name=None)
```

V této deklaraci lze identifikovat parametr s proměnlivým typem, *data*, který může pomocí konstrukce Union nabývat typu `Tensor` nebo `np.array`. Následuje parametr s výchozí hodnotou: *name*, který může být jakéhokoliv typu.

4.2.1 Rozdílné typy

Zatímco Python poskytuje dynamické typování bez potřeby dalších úprav, C++, jakožto staticky typovaný jazyk, vyžaduje specifické techniky pro práci s více typy. V C++ existují různé způsoby, jak se s různými typy vypořádat, jako jsou použití ukazatelů a dynamický polymorfismus, standardní knihovna s std::variant a std::any nebo použití templatů a statického polymorfismu.

4.2.1.1 Použití ukazatelů

Jedním tradičním přístupem jak dosáhnout dynamického typování je užití ukazatelů. První možností jsou void ukazatele, tedy ukazatele obsahující pouze adresu objektu bez jakékoli typové specifikace. Tento přístup, jakkoli flexibilní, umožňuje psát kód vysoce náchylný k chybám. Kompilátor nemůže otestovat typovou kompatibilitu a chyba se může projevit za běhu a velice špatně se identifikuje. Kromě toho vyžaduje také manuální správu paměti. Jako příklad je uvedena C++ verze ukázky 4.1:

Zdrojový kód 4.2: C++, Dynamické typování pomocí void ukazatele

```
// při použití void ukazatele je nutno typ
// předat jiným způsobem
void modify_data(
    void* data, bool tensor_or_numpy,
    void* name=null, int name_type_id=0)
```

Další možností jsou raw ukazatele, které již informaci o typu obsahují. Ukazatele na instance základní třídy mohou odkazovat i na instance tříd odvozených, což umožňuje polymorfní chování za běhu programu. Toto řešení však vyžaduje, aby

všechny odvozené třídy dědily ze třídy základní a obsahovaly příslušné virtuální metody. Pro využití v příkladu výše je tedy toto řešení nevhodné. Mimo to může být toto řešení neefektivní kvůli přídavné režii volání virtuálních funkcí.

Stejně jako u void ukazatelů i u tohoto typu ukazatelů je nutná manuální správa paměti. Tento problém lze v moderním C++ eliminovat pomocí chytrých (smart) ukazatelů, které spravují celý životní cyklus objektů pomocí metody zvané počítání referencí (reference counting). Následují C++ verze metody z ukázky 4.1:

Zdrojový kód 4.3: C++, Dynamické typování pomocí raw a smart ukazatele

```
// AbstractTensor je uživatelská třída,
// od které dědí Tensor i np::array
void modify_data(
    AbstractTensor* data,
    /* name - zde nelze polymorfismus využít vůbec */);

// využití smart ukazatele pro automatickou správu paměti
// AbstractTensor objektu
void modify_data(
    std::shared_ptr<AbstractTensor> data);
```

4.2.1.2 std::variant, std::any

Dalším přístupem je použití tříd/wrapperů std::variant a std::any ze standardní knihovny, které byly představeny v C++17. std::variant je bezpečný typový svazek (nástupce struktury union), který může obsahovat jeden z několika alternativních typů, zatímco std::any je kontejner, který může obsahovat jakýkoli typ objektu. Oba typy poskytují způsob, jak pracovat s více typy obecně, aniž by bylo nutno sáhnout k dynamickému polymorfismu. Také poskytují ověřování typů za běhu a v případě variant i při překladu, což umožňuje efektivnější kód a snižuje riziko chyb za běhu programu. std::variant také nealokuje další dynamickou paměť a daný objekt konstruuje v sobě. std::any nealokuje další dynamickou paměť, pokud je objekt dostatečně malý. Syntaxe při použití těchto wrapperů je z uvedených přístupů syntaxi v Pythonu asi nejbližší, na druhou stranu je zde stále přítomna runtime režie při zjišťování typu:

Zdrojový kód 4.4: C++, Dynamické typování s využitím knihovních struktur

```
// stále je nutno předávat data jako ukazatele,
// aby se zabránilo kopírování
void modify_data(
    std::variant<Tensor*, np::array*> data,
    std::any name)
```

4.2.1.3 Templaty

V C++ je templating mechanismus implementován pomocí generického programování, tedy programování, kdy je algoritmus napsán s typy, které budou definovány později. Koncept templatingu umožňuje definovat funkce a třídy, které nejsou vázány na konkrétní typy, ale na parametry templatu (template parameters). Tyto parametry jsou nahrazeny konkrétními typy a hodnotami při překladu programu. Stejný kód lze použít pro různé datové typy, což vede ke snadnější údržbě a rozšiřování kódu. Použití templatů může vést také ke zvýšení efektivity, jelikož odpadá režie spojená se zjišťováním typů za běhu. Na druhou stranu to ovšem vyžaduje, aby byl typ objektu znám již v době překladu. Kromě toho, použití templatů může zvýšit čas překladu a template chybové zprávy bývají náročné na pochopení:

Zdrojový kód 4.5: C++, Statické typování pomocí templatů

```
// TensorType je placeholder typ pro Tensor a np::array
template<typename TensorType>
void modify_data(
    TensorType& data,
    std::any name)
```

4.2.2 Pojmenované argumenty

Dalším úskalím po typové adaptaci Pythonu do C++ je imitace "pojmenovaných argumentů" (keyword arguments). V Pythonu lze předávat argumenty funkcím dle jejich pozice nebo specifikováním jejich jmen. Tato syntaxe mimo jiné umožňuje definovat funkce s mnoha argumenty s výchozími hodnotami a následně při jejich volání specifikovat pouze ty argumenty, jejichž hodnoty se od výchozích liší. V následujícím příkladě je volána funkce conv s výchozími argumenty, kromě argumentu bias, který je nastaven na True:

Zdrojový kód 4.6: Python, Volání metody s výchozími parametry

```
def conv(stride=2, padding=1, bias=False)
  pass

conv(bias=True) # upravujeme pouze hodnotu argumentu bias
```

C++, na druhou stranu, argumenty rozlišuje pouze na základě jejich pozice a typu, a pojmenované argumenty nenabízí. To vede k nutnosti specifikace všech argumentů předcházejících upravovanému argumentu, včetně těch s výchozí hodnotou:

Zdrojový kód 4.7: C++, Volání metody s výchozími parametry

```
void conv(int stride=2, int padding=1, bool bias=false);

// nutno specifikovat všechny parametry před bias,
// přestože mají definovanou výchozí hodnotu
conv(2, 1, true);
```

V C++ se pro tento účel používá paradigmatu předávání struktur argumentů. Místo předávání každého argumentu do metody zvlášť se vytvoří struktura se všemi argumenty s výchozími hodnotami, některé z nich se modifikují a následně se předá dané metodě ukazatel na vytvořenou strukturu. Tento postup má řadu výhod. Metody s jedním argumentem jsou jednodušší, lze definovat více metod se stejným jménem, ale jinou sadou argumentů, což je užitečné zvláště u rozsáhlých API, jako je např. Vulkan [11], a nakonec, definováním výchozích hodnot atributů struktury lze replikovat funkci pojmenovaných argumentů v Pythonu:

Zdrojový kód 4.8: C++, Předávání struktury jako parametru

```
struct ConvOptions{
  int stride = 2;
  int padding = 1;
  bool bias = false;
}
// ...
ConvOptions c;
c.bias = true;
conv(c);
```

Následně lze využít tzv. zřetězení metod (method chaining) pro plnou replikaci Python syntaxe na jednom řádku. Stejné paradigma používá i C++ frontend PyTorche LibTorch:

Zdrojový kód 4.9: Srovnání volání metody s výchozími parametry v Pythonu a C++

```
# Python
conv(bias=True, padding=4)

// C++
conv(ConvOptions().bias(true).padding(4));
```

5 nnframework

Jedním z cílů této bakalářské práce je převedení resp. adaptace Python knihovny *nnframework* do C++. Tato knihovna obsahuje mnoho částí jako např. příprava dat, jejich zpracování, návrh modelu, implementace trénovacího cyklu, ukládání modelu i loggování a metrika. Kromě toho patří k *nnframeworku* rozsáhlá Model Zoo[1] s mnoha již konkrétními implementovanými modely. V rámci této práce bude však převedena/adaptována pouze základní část *nnframeworku* s podmnožinou modelů umožňujících otestování správnosti její implementace. Mimo to je *nnframework* neustále vyvíjen a určité informace zde zmíněné již nemusí být zcela aktuální.

5.1 Popis nnframeworku

Projekt nnframework byl založen v roce 2021 vedoucím práce s cílem usnadnit a zrychlit práci a experimentování s různými modely se širokým zaměřením zahrnujícím např. zpracování obrazu (UNET), zpracování přirozeného jazyka (BERT) a generování obrazu (DCGAN). nnframework poskytuje abstrakci nad PyTorch objekty a funkcemi. Nabízí již hotové trénovací procedury, automatické ukládání a načítání modelu a logging. Dále disponuje metodami pro snadné načítání a preprocessing trénovacích a testovacích dat. Díky tomu lze mnohé části implementovat jednou a využít u více modelů.

5.2 Struktura nnframeworku

Knihovna se skládá z několika základních tříd, které jsou vesměs určené k dědění a následnému doimplementování funkcionalit. Nutno podotknout, že cílem následujícího popisu není poskytnout podrobné pojednání o funkcích nnframeworku, ale pouze nastínit jeho strukturu, která bude do jisté míry C++ verzí převzata.

[1] Model Zoo je termín obecně používaný pro sbírku předtrénovaných modelů pro strojové učení.

- **AbstractTorchModel**

  Základní třída dědící z PyTorch třídy Module. Kromě základní funkcionality plynoucí z třídy torch.Module, která představuje jednu či více neuronových vrstev, nabízí AbstractTorchModel také metody pro načítání a ukládání modelu. Mimo to obsahuje metody `runForward()` a `runTrain()`, které jsou určené k přetížení.

  Přesněji AbstractTorchModel dědí ze tříd torch.Module a AbstractModel, přičemž AbstractModel nabízí stejné rozhraní jako AbstractTorchModel, ale bez funkcionalit specifických pro PyTorch. Třída AbstractModel byla teoreticky určena k použití v rámci vícero rozhraní: PyTorch i TensorFlow, ale v praxi se nnframework zaměřil pouze na PyTorch.

- **InputLoader**

  Neformální interface[2] poskytující trénovací/testovací data v raw podobě. Tato třída řeší problém s vysokou rozmanitostí vstupních dat a poskytuje jednotné rozhraní pro přístup k nim. Potomek této třídy je tedy zodpovědný za načtení dat z disku, případně jejich augmentaci (rozšíření) či vygenerování dat nových.

- **DefaultDataset**

  Potomek třídy torch.Dataset zodpovědný za přípravu dat pro konkrétní model. DefaultDataset využívá InputLoader pro získání raw dat, které následně konvertuje obvykle na torch.Tensory. Typickým příkladem raw dat může být obrázek, text či štítek (label). Pro poslední jmenované se užívá další třídy ClassConvertor, která převádí obvykle textový štítek na integer.

- **DefaultTorchEvaluator**

  Třída zajišťující samotné trénování i testování. Obsahuje smyčku s epochami a vnitřní smyčku iterující přes vzorky. K tomu také vypisuje průběh do konzole. Obsahuje základní implementaci pro jeden input, target a loss. Pro implementaci složitějších architektur, jako jsou např. GANs (General Adversarial Networks), je nutno vytvořit vlastního potomka třídy DefaultTorchEvaluator a přetížit metodu `runLoop()`.

- **AbstractTask**

  AbstractTask je třída zastřešující celý proces trénování a inference[3]. Je vytvořena s důrazem na flexibilitu, a mnoho metod lze přetížit, ale zároveň obsahuje

[2] Neformální interface v Pythonu je třída s definovanými metodami s prázdným tělem. Potomek této třídy pak může dané metody přetížit.

[3] Inference je proces výpočtu výstupní hodnoty neuronového modelu v závislosti na vstupních datech. Síť se v tomto případě neučí.

výchozí implementaci, což umožňuje snadné použití s minimálními úpravami. Základní metodou, jejíž tělo je třeba doplnit, je `initModel()`, která je určena pro vytvoření konkrétního AbstractModelu. V této metodě je možno také nastavit další parametry, např. ztrátovou funkci či Evaluator.

- **Settings**

  Třída Settings působí jako 'pojidlo' všech komponent. Obsahuje atributy jako jsou např. počet epoch (`epochCount`), velikost dávky (`batchSize`), ztrátová funkce (`lossFn`) a separátní input loadery (`inputs`) pro trénování, testování a validaci. Typicky se instanci Settings nejprve nastaví základní parametry (`batchCount`, `epochCount`), následně input loadery a nakonec se instance předá do konstruktoru třídy AbstractTask. Mimo to lze nastavit v Settings také PretrainedManager, který se stará o automatické ukládání a načítání modelu i mezi epochami.

5.3 Příklad užití nnframeworku

Typické použití nnframeworku pro načtení vah existujícího modelu, jeho natrénování a následné otestování může vypadat následovně:

Zdrojový kód 5.1: nnframework, Příklad typického trénování modelu

```
# Nastavení základních trénovacích parametrů
settings = Settings()
settings.type = AbstractModel.ModelType.SEGMENTATION
settings.batchSize = 40
settings.epochCount = 10
settings.numWorkers = 0

# Nastavení input loaderů
settings.input.train = SkyImageInputLoader("train/.")
settings.input.dev = SkyImageInputLoader("dev/.")
settings.input.test = SkyImageInputLoader("test/.")

# Nastavení automatického načítání vah (weights) modelu
settings.pretrainManager = PretrainedManager()
settings.pretrainManager.enableLoading(True)

# Vytvoření potomka AbstractTasku
ss = SkyDetectionTask(settings)

# Trénování a testování
ss.train()
ss.test()
```

Návrh a Implementace C++ knihovny

6

Prvním důležitým bodem je rozsah konverze. Originální verze knihovny obsahuje mnoho rozšiřujících funkcionalit, které nejsou nezbytné z hlediska jejího použití (např. sada univerzálních metrik či více před-připravených PyTorch modulů). Nakonec bylo vybrána podmnožina funkcionalit, které budou nezbytné pro implementaci testovacího modelu a práci s ním. Mimo to byly pro lepší návrh knihovny zvoleny ještě další dva modely: Feedforward model pro klasifikaci obrázků a DCGAN (Deep Convolutional Generative Adversarial Network) pro jejich generování.

Dalším bodem je samotná konverze kódu. Vzhledem k významným rozdílům mezi jazyky Python a C++ není možné provést přímé mapování nnframeworku do C++ a je nutné se uchýlit ke kompromisům, viz Kapitola 4.2. Jako hlavní vodítko v tomto procesu poslouží samotný LibTorch [12]. Tento C++ frontend PyTorche nabízí velmi podobnou syntaxi Python frontendu, ale pro zajištění typové flexibility používá templaty. [13]

Každý model může mít jiné typy a počty vstupů a výstupů. Např. zatímco v případě GAN sítí se může jednat o jediný obrázek, Unet vyžaduje k vstupnímu obrázku také vzorové (target) mapy. Tato data vážící se k jednomu vzorku je obvykle nutno dávkovat (batching)[1], a bylo by tedy výhodné je uložit do jedné struktury. Pro univerzální přístup k této struktuře by bylo naivně možné definovat virtuální třídu, jejíž potomci by příslušné formáty vstupů a výstupů implementovali, to by ovšem vedlo ke zbytečným alokacím a voláním virtuálních funkcí. Z tohoto důvodu využívá LibTorch templaty. Příkladem může být `torch::data_loader`, který přijímá jako template parametr typ struktury obsahující příslušná data.

Modely, resp. PyTorch Moduly, na druhou stranu templatované nejsou a uplatňuje se zde dynamický polymorfismus. Lze je tedy bez obtíží uložit jako ukazatel na svého předka a volat příslušné virtuální metody. V tomto případě problém s vy-

[1]Velikost dávky určuje počet vzorků, které jsou zpracovávány modelem najednou. Při procesu učení se jedná o minimální počet vzorků, po kterých se aktualizují váhy modelu.

sokým počtem alokací nehrozí, protože model obvykle žije po celou dobu běhu programu.

Ve skutečnosti jsou LibTorch Moduly pouze kontejnery pro shared_ptr, které ukazují na skutečné implementace modulů. V tomto případě se jedná pouze o syntax sugar[2]. LibTorch pro vytvoření kontejneru pro příslušnou implementaci nabízí dokonce macro.

Kvůli snadnější implementaci budou výše zmíněné konvence využity i v navrhované C++ verzi nnframeworku. Např. třída AbstractModel nebude po vzoru `torch::Module` templatovaná a bude využívat kontejneru, a InputLoader, včetně všech tříd s ním souvisejících, bude templatovaný. To umožní flexibilní přístup k datům bez negativního vlivu na dobu běhu. Na druhou stranu třída Settings, která v originální knihovně obsahovala InputLoadery, v C++ verzi InputLoadery vlastnit nemůže, protože ty jsou templatované a templatovanou by se pak musela stát i třída Settings, což by výrazně zvýšilo složitost C++ knihovny.

Celkově je tedy cílem konverze se co nejvíce přiblížit struktuře originální knihovny, současně je však vhodné dodržovat konvence nastavené knihovnou LibTorch. Hlavní komponenty knihovny a jejich funkce však budou moci být s menšími úpravami zachovány. Kromě toho by měla být knihovna snadno rozšiřitelná a umožňovat přidání dalších funkcí.

6.1 Struktura projektu

Projekt samotný je psán ve standardu C++17. Tento standard již zdaleka není nejnovější verzí C++ standardu, a je tedy široce přijat a podporován všemi hlavními kompilátory, což z něj činí stabilní a spolehlivou volbu pro vývoj. Mimo to C++17 představil řadu užitečných jazykových funkcí, jako je např. `if constexpr` který nalezl využití právě u templatů. Mezi další novinky patří také upravení `std::filesystem`u pro snadnější práci se soubory. Přestože je knihovna primárně určena pro platformu Windows, je kompatibilní i s Linuxem. Jako systém správy verzí (Version Control System) je použit git. [14]

Pro správu projektu byl vybrán nástroj CMake [15]. Přestože jeho zápis není tak přehledný jako např. novější Premake [16], CMake je de facto standardem pro C++ projekty a pro snadnější integraci knihoven třetích stran, které obvykle CMake také využívají, je jasnou volbou. Kromě toho je CMake doporučován i LibTorchem [17].

6.1.1 Závislosti projektu

Projekt závisí na následujících knihovnách:

[2] Syntax sugar je pojem označující jazykovou konstrukci nebo zkratku, která sice není nezbytná pro funkčnost programu, ale značně usnadňuje jeho psaní a čtení.

- **LibTorch (verze 2.0)**

  C++ frontend frameworku PyTorch. Pro využití GPU akcelerace je možné nainstalovat také CUDA, resp. CuDNN. Pro platformu Windows existují dvě separátní verze LibTorch v závislosti na konfiguraci: Debug a Release.

- **OpenCV (verze 4.6.0)**

  Open-source knihovna zaměřená na počítačové vidění a strojové učení [18]. Knihovna obsahuje algoritmy pro zpracování obrazu, detekci a rozpoznávání objektů, sledování pohybu, rekonstrukci 3D modelů a další. Tato knihovna byla vybrána jako náhrada za Python knihovnu torchvision, která je často používána v kombinaci s PyTorch. OpenCV umožňuje snadnou integraci, je stabilní a nabízí širší spektrum funkcí.

- **SPDLOG**

  Rychlý logovací nástroj [19]. Knihovna je navržena tak, aby poskytovala efektivní a flexibilní logování s minimálním dopadem na výkon. SPDLOG podporuje různé formáty zpráv, ukládání do souborů, a další výstupy. Navíc je k dispozici také ve verzi header-only.[3] Tato knihovna byla vybrána pro replikaci nativní Python knihovny `logging`.

- **tensorboard_logger**

  TensorBoard_logger [20] je C++ knihovna pro záznam metrik během trénování neuronových modelů ve formátu TensorBoard. Umožňuje ukládat skalární hodnoty, histogramy, obrázky a další informace, které lze následně vizualizovat pomocí TensorBoard webového rozhraní. Díky tomuto nástroji je sledování vývoje modelů a optimalizace jejich hyperparametrů jednodušší. Tato knihovna nahrazuje funkcionalitu PyTorch modulu torch.utils.tensorboard. Je důležité poznamenat, že se jedná o samostatný neoficiální projekt, který nesouvisí s LibTorch ani s TensorBoard.

Závislé knihovny se nacházejí v hlavní složce projektu `vendor/`. Tato složka obsahuje knihovny SPDLOG a LibTorch. Naopak knihovna OpenCV je externí a je nutno její lokaci zahrnout do systémové proměnné `PATH`. Nakonec knihovna tensorboard_logger využívá CMake modulu FetchContent, který umožňuje stáhnout projekt automaticky přímo z git repositářů.

[3] Header-only knihovna znamená, že veškerá implementace je obsažena v hlavičkových souborech, což zjednodušuje integraci do projektu, protože není třeba kompilovat a linkovat zdrojové soubory.

6.1.2 Zdrojový kód

Zdrojový kód je členěn do 4 složek. Ve složce `core` se nachází hlavní hlavičkový soubor `nn.h`, který importuje základní hlavičkové soubory, které budou s vysokou pravděpodobností relevantní v rámci celého projektu.

Tento soubor je rovněž označen jako předkompilovaný hlavičkový soubor (PCH, precompiled header). PCH obsahuje předem přeložený kód knihoven a hlavičkových souborů, které jsou v projektu často využívány. Bez PCH by se tyto soubory překládaly v rámci každé překladové jednotky zvlášť, s PCH se tyto soubory přeloží pouze jednou do binárního souboru, který je poté využíván při každém dalším spuštění překladu projektu. To obvykle vede ke zrychlení překladu, zvláště u větších projektů.

Dále složka `core` obsahuje:

- **string utilities** (sil.h) nabízejí funkce pro snadnou práci s řetězci.
- **io utilities** (nn_io.h) umožňují snadné načítání/ukládání obrázků rovnou do/z LibTorch tensorů. Je podporováno také ukládání celé dávky obrázků do mřížky jednoho obrázku, což může být užitečné pro následnou analýzu dat.
- **Log** umožňuje jednotný přístup ke globálnímu SPDLOG Loggeru pomocí maker, např. `NN_INFO("Hello")` či `NN_ERROR("Bad")`. Využití těchto maker umožňuje vypnutí veškerého konzolového výstupu včetně režie s ním spojené pomocí redefinice těchto maker jako prázdných.
- **Settings** obsahuje parametry modelu a testování
- **Rozšíření** (extensions.h) umožňují tvořit templatované data-processing pipeliny pro trénovací a testovací data.
- **Core** (nn_core.h) zajišťuje základní funkcionality jako výpis progress bar do konzole a malý wrapper pro profilování. [4]

Následují složky, které není nutno podrobněji popisovat, jelikož obsahují C++ verze již zmíněných objektů originálního nnframeworku. Složka `model` obsahuje třídy zajišťující práci s modely, jako jsou např. AbstractModel, AbstractTask a DefaultEvaluator. Ve složce `data` se zase nachází třídy zodpovědné za načítání a přípravu datasetu: InputLoader, ClassConvertor a DefaultDataset. Nakonec složka `examples` obsahuje již implementované modely: DCGAN, Unet, feedforward klasifikátor datasetu MNIST. Kompilaci obsahu této složky lze vypnout CMake přepínačem `NN_EXAMPLES`.

[4] Profilování je proces sběru, analýzy a interpretace dat o běhu programu s cílem zjistit, které části kódu jsou nejčastěji používány a kde se tráví nejvíce času.

6.2 Práce s projektem

Při práci s projektem se předpokládá využití Visual Studia 2022 IDE (integrovaného vývojového prostředí, Integrated Development Environment) [21]. Projekt obsahuje soubor `MANUAL.md`, který podrobně popisuje princip práce s frameworkem znázorněném na jednoduchém modelu. Pro snadnou přípravu byl projekt vybaven několika skripty.

Prvním skriptem je CMake skript `DownloadLibTorch.cmake`, který je zodpovědný za stáhnutí a extrakci správné verze knihovny LibTorch. Při práci s knihovnou se předpokládá nutnost využití dvou projektových konfigurací: Debug, obsahující debug symboly pro krokování běhu programu a snadné nalezení případných chyb, a Release, který je vhodný pro následné náročné výpočetní úkony jako jsou inference či trénování. Obvykle lze pro knihovny zvolit příslušnou konfiguraci při sestavování projektu. Knihovna LibTorch však na platformě Windows vyžaduje užití jedné ze dvou verzí, kterou skript v závislosti na zvolené konfiguraci připraví. Tento skript je volán automaticky při zpracování CMakeLists.txt.

Dalšími dvěma skripty jsou `MakeBuildDebug.bat` a `MakeBuildRelease.bat`, které zavolají CMake pro vygenerování nativního Visual Studio projektu pro konfiguraci Debug, resp. Release. Pro usnadnění práce s příklady modelů je složka `data/` vybavena skripty pro stáhnutí a extrakci příslušných datasetů.

S projektem lze pracovat dvěma způsoby. Prvním způsobem je vygenerování projektového souboru `.sln` prostřednictvím CMake a následné otevření jako nativního projektu ve Visual Studiu. Projekt může být ovšem otevřen také jako CMake projekt. V tomto případě odpadá nutnost mít dva separátní Visual Studio projekty v závislosti na konfiguraci. Podrobný popis přípravy projektu a sestavení jednoduchého příkladu je uveden v příloze A.

Testování knihovny 7

Pro otestování funkčnosti knihovny byl zvolen model Unet [22]. Unet byl vybrán také jako testovací model pro porovnání C++ a Python verze z důvodu jeho snadné implementace a také možnosti exaktního stanovení úspěšnosti modelu pomocí chybové funkce.

7.1 Unet

Unet je typ konvoluční sítě počátečně vytvořený pro úlohy segmentace biomedicínského obrazu, konkrétně pro segmentaci obrázků buněk a tkání. Název "Unet" je odvozen podle tvaru neuronové sítě, která připomíná písmeno U. Od svého vzniku v roce 2015 našel Unet uplatnění i v mnoha dalších odvětvích jako je např. obnova obrazu, odstranění šumu, zvyšování rozlišení, i ve zpracování přirozeného jazyka.

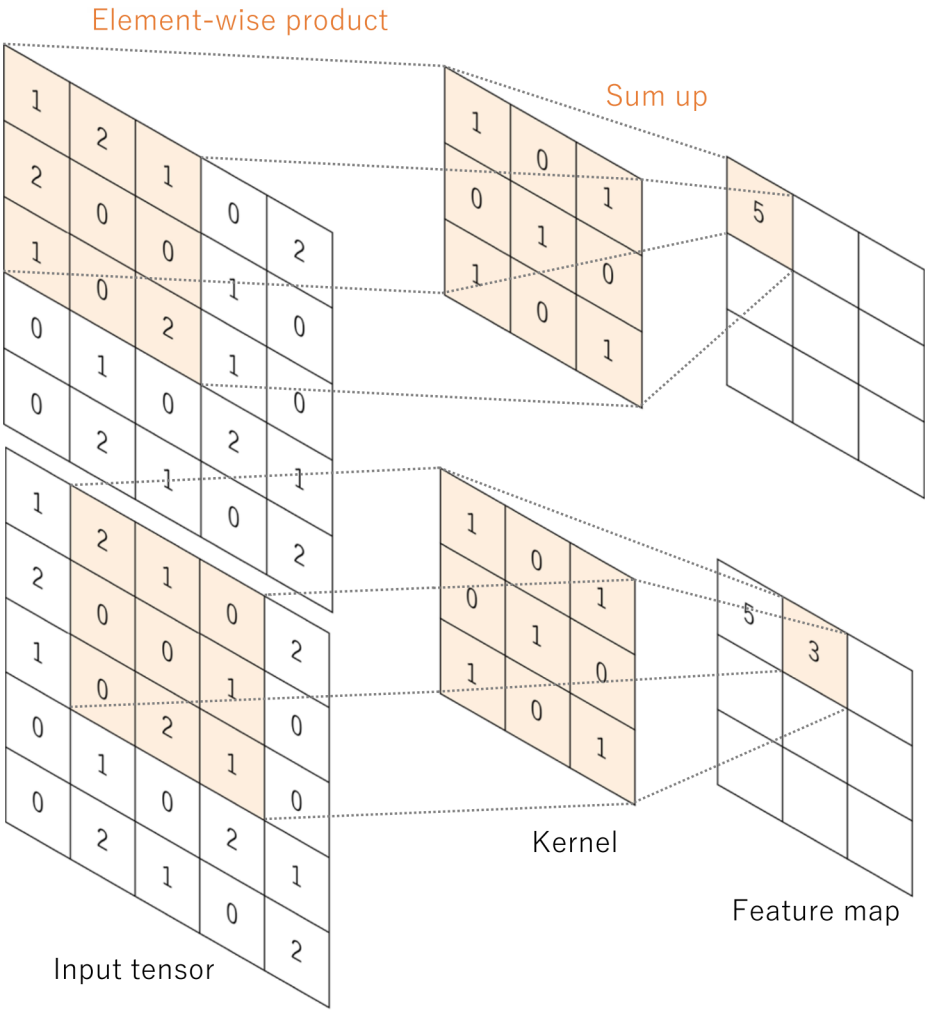
Segmentace obrazu spočívá v rozhodování, zdali daný pixel v obraze patří do určité třídy, či nikoliv. Obecně tedy rozlišení vstupního obrazu odpovídá rozlišení výstupní masky. Výstupní maska ovšem může obsahovat více kanálů, každý odpovídající určitému typu segmentovaných objektů. (Viz obr. 7.4 na str. 27.)

7.1.1 Konvoluční vrstva

Jednou ze základních vrstev modelu Unet je konvoluční vrstva [23]. Narozdíl od Fully-Connected vrstev, kde jsou všechny neurony ve vrstvě propojené s všemi neurony z vrstvy předchozí, v konvoluční vrstvě se aplikuje filtr/kernel (který se skládá z několika neuronů) na menší části vstupního obrazu nebo mapy předchozí vrstvy, viz obr. 7.1. Tímto způsobem se redukuje počet propojení mezi neurony, což umožňuje efektivnější výpočet a snižuje riziko přeučení[1]. Další důležitou vlastností konvoluční vrstvy je prostorová nezávislost (Spatial Invariation). Jedná se o schopnost sítě rozpoznat vzory nebo rysy ve vstupním obrázku bez ohledu na jejich pozici.

[1] Přeučení (Overfitting) je jev, který nastává, když se model příliš přizpůsobí trénovacím datům, což vede ke špatné generalizaci na nových, neviděných datech. Model se v tomto případě naučí i šum a náhodné fluktuace trénovacích dat, místo aby se zaměřil na skutečné vzory a závislosti.

Tato vlastnost vyplývá z faktu, že kernely jsou posouvány napříč celým obrázkem a pokud se kernel naučí určitý znak, může pak tento znak rozeznat kdekoliv. Tato vlastnost činí konvoluční vrstvy zvláště vhodnými pro úlohy s obrázky nebo jinými prostorovými daty.



Obrázek 7.1: Příklad výpočtu dvou hodnot při aplikaci kernelu/filtru [24]

Vstupem konvoluční vrstvy je obvykle 3 dimenzionální tensor. V případě obrázku to může být výška, šířka a počet kanálů. Počet dimenzí však může být i vyšší, např. v případě volumetrických dat. Obecně se tedy jedná o vstupy, mezi kterými existuje prostorová závislost, tedy např. dva sousední pixely budou s vysokou pravděpodobností patřit jednomu objektu.

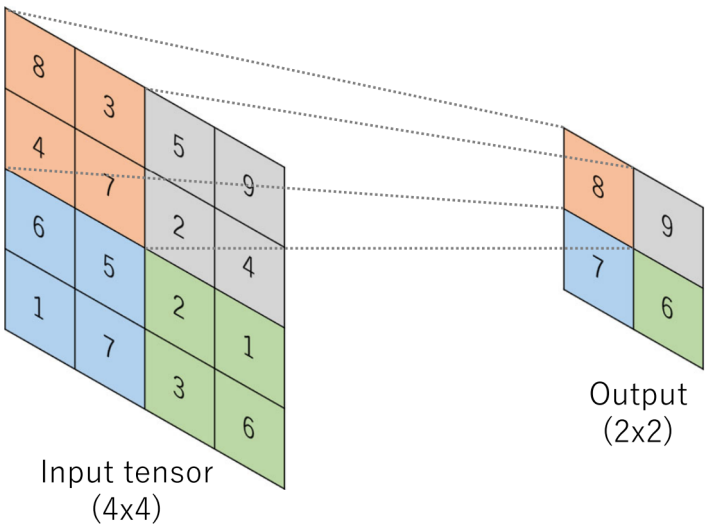
Konvoluční vrstva se skládá z více filtrů - kernelů provádějících konvoluci nad vstupními daty. Pro případ vstupu s jedním kanálem[2], tedy například černobílým

[2]V případě vstupu s vyšším počtem kanálů bude mít kernel také hloubku odpovídající počtu

obrázkem, se kernelem rozumí čtvercová "matice" s lichou délkou hrany, např. 3x3. Výstup je získán postupným posouváním kernelu přes obrázek a aplikací skalárního součinu mezi vstupními daty a kernelem, viz obr. 7.1. Pro zjednodušení lze chápat každý kernel jako detektor určitého znaku (např. vertikální hrana). Každý kernel určuje jednu výstupní mapu příznaků, a tedy počet kernelů udává počet výstupních kanálů.

7.1.2 Pooling vrstva

Pooling vrstva [25] obvykle následuje za konvoluční. Narozdíl od konvoluční vrstvy však neobsahuje žádné naučitelné parametry. Funkcí této vrstvy je snížení prostorové dimenze map příznaků. Slučuje tedy sadu hodnot do menšího počtu hodnot. Existuje více typů jako např. MaxPooling (obr. 7.2), který vybere ze sady hodnot jejich maximum, či AveragePooling, kde je vybrán jejich průměr.

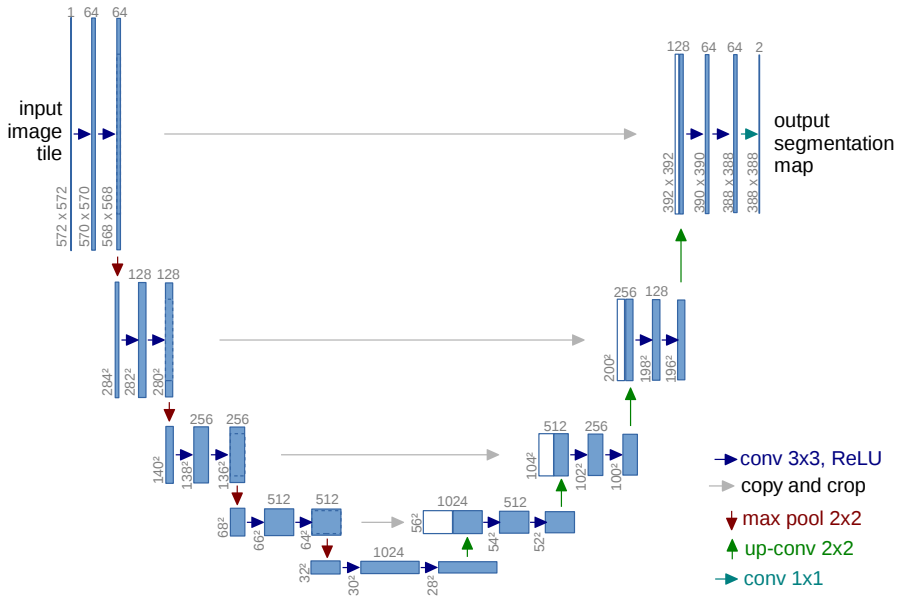


Obrázek 7.2: Aplikace 2x2 MaxPooling vrstvy [24]

7.1.3 Struktura Unet

Struktura Unet se skládá ze dvou částí: Enkodér a Dekodér. Encoder obsahuje více konvolučních (3x3) a na ně navazujících MaxPooling (2x2) vrstev, které postupně snižují prostorovou dimenzi obrazu a zároveň zvyšují počet kanálů, tedy map příznaků. Na druhou stranu Dekodér je složen z více upsampling [26] a konvolučních vrstev, které naopak zvyšují prostorovou dimenzi a zároveň snižují počet kanálů (obr. 7.3).

vstupních kanálů. Tedy například pro obrázek RGB (se třemi kanály) by mohl mít kernel rozměr 3x3x3.



Obrázek 7.3: Struktura sítě Unet [22]

Jednou z významných vlastností Unetu jsou přeskakující spojení (Skip Connections, na obr. 7.3 znázorněny jako dlouhé horizontální šipky), které propojují vrstvy Enkodéru s korespondujícími vrstvami Dekodéru. Tato propojení umožňují Dekodéru přístup k informacím z dřívějších fází Enkodéru, a díky tomu jsou zachovány prostorové informace, které by byly jinak opakovanou aplikací poolovací vrstvy ztraceny.

Další výhodou této architektury, která plyne z faktu, že jsou všechny hlavní vrstvy konvoluční a síť neobsahuje žádné Fully Connected (FC) vrstvy, je nezávislost na velikosti vstupního obrazu. Tedy velikost výstupní masky je dána velikostí vstupního obrazu. Nutno ovšem zmínit, že paměťová náročnost s rostoucím rozlišením stoupá, a v praxi se velké vstupní obrazy rozdělují na dlaždice s přesahem. Pro zajištění plynulého přechodu při využití dlaždic se volí velikost vstupního obrazu jako násobek dvou. (Každý vstup MaxPooling vrstev bude mít sudé rozměry.)

7.2 Implementace testovacího modelu

Cílem této práce bylo vytvořit C++ knihovnu, která by se přibližovala stylem zápisu originálnímu Python nnframeworku. To by mělo umožnit snadný přechod z originální knihovny na tuto knihovnu a zároveň usnadnit případnou konverzi rozsáhlé kolekce implementovaných modelů (včetně modelu Unet), kterou nnframework disponuje. Záměrem při implementaci modelu Unet v C++ nebyla tedy pouze implementace modelu samotná, ale také refaktorování celé C++ verze nnframeworku, která by měla implementaci a následnou práci s modelem co nejvíce usnadnit.

Při konverzi Unet testovacího modelu byl také vyzkoušen a použit nástroj ChatGPT společnosti OpenAI [27], který byl s menšími zásahy schopen převést PyTorch model na LibTorch. Samotnou integraci modelu do knihovny bylo ovšem nutno provést ručně. Obecně je tento nástroj zatím vhodný na samostatné struktury či části kódu, které nevyžadují rozsáhlý kontext (např. PretrainedManager).

Na příkladu zdrojového kódu 7.1 (řádek 12, str. 26) je patrné, že se InputLoader neukládá do struktury Settings, jako je tomu v originálním Python nnframeworku (zdrojový kód 5.1, str. 16). Namísto toho je nutné ho předat do UnetTask, což je potomek AbstractTask, separátně. Kromě tohoto detailu je však inicializace trénování v C++ knihovně blízká originální Python verzi.

Zdrojový kód 7.1: C++, Příklad inicializace a zahájení trénování Unet modelu

```
1  Log::init();
2  auto settings = std::make_shared<nn::Settings>();
3  settings->device = torch::kCUDA;
4  settings->batch_size = 8;
5  settings->epoch_count = 1;
6  settings->num_workers = 4;
7  settings->loss_fn = &bceDiceLoss;
8
9  // {{width, height, channels}, outputMaps}
10 UnetSettings s = { {256, 256, 3},1 };
11
12 auto input = nn::InputLoaders<TargetImageInputLoader>();
13 input.train = std::make_shared<nn::SegmentationInputLoader(
14     "unet_augmented/data",
15     "unet_augmented/target",
16     s.dims, true,
17     settings->input_settings,
18     cv::IMREAD_COLOR, cv::IMREAD_GRAYSCALE);
19
20 auto model = SimpleUNetModel(
21     std::make_tuple(dims.chan_count,dims.height, dims.width),
22     std::make_tuple(outputMaps, dims.height, dims.width)
23 );
24
25 settings->pretrained_manager = PretrainedManager("latest",
26     "C:/model_folder", "");
27 settings->pretrained_manager.enableSaving(false);
28 settings->pretrained_manager.enableLoading(true);
29
30 auto ss = UNetTask(settings, input);
31 ss.train();
```

7.3 Způsob testování

Testovací zařízení bylo vybaveno mobilní verzi grafické karty s podporou CUDA od společnosti NVIDIA - GeForce RTX 2060, a procesorem Intel Core i7-10750H s operačním systémem Windows 10, M.2 NVMe SSD diskem a 32GB RAM. Verze Python i C++ frontendu byla PyTorch 2.0. Následně bylo zvažováno použití matematických knihoven pro optimalizovaný výpočet na konkrétních zařízeních: knihovna cuDNN [28] pro karty NVIDIA a knihovna MKL-DNN [29] pro procesory Intel. Nakonec byla vybrána pouze knihovna cuDNN, jelikož grafické karty jsou očekávaným zařízením pro výpočet v neuronových sítích.

Pro trénování segmentačního modelu Unet byl využit dataset obsahující různé komiksové obrázky opatřené černobílou mapou příznaků, která identifikuje textové bubliny (obr. 7.4). Dataset obsahuje 100 obrázků s vyšším rozlišením, model však očekává vstupy o rozlišení 256x256. Tento problém ovšem knihovna řeší automaticky aplikací škálovací transformace. Transformaci obrázku v tomto případě zajišťuje konkrétní InputLoader pomocí zabudované transform pipeline. Z důvodu nízkého počtu vstupních dat bylo využito augmentace, což je technika umělého rozšíření trénovacích dat pomocí různých transformací. V rámci augmentace, jejíž základní a neoptimalizovaná verze je zahrnuta v knihovně, byl vygenerován nový dataset s náhodnými změnami jasu, kontrastu, pozice a rotace a nakonec byla také na podmnožinu dat aplikována elastická tranformace. V rámci tohoto procesu byl počet vzorků (sample count) navýšen na více než 3200, což dále vedlo ke zvýšení diverzity dat a také ke snížení rizika overfittingu.

Jako velikost dávky (batch size) byla pro všechny testy zvolena hodnota 8 odpovídající maximální kapacitě paměti GPU.

Obrázek 7.4: Příklad dat v použitém datasetu, nalevo vstupní obrázek, napravo mapa textových bublin

7.4 Výsledky testování

Následující testy inference i trénování byly provedeny vícekrát (v závislosti na jejich trvání) a jejich výsledky zprůměrovány, aby se zamezilo výkyvům způsobeným náhodnými faktory.

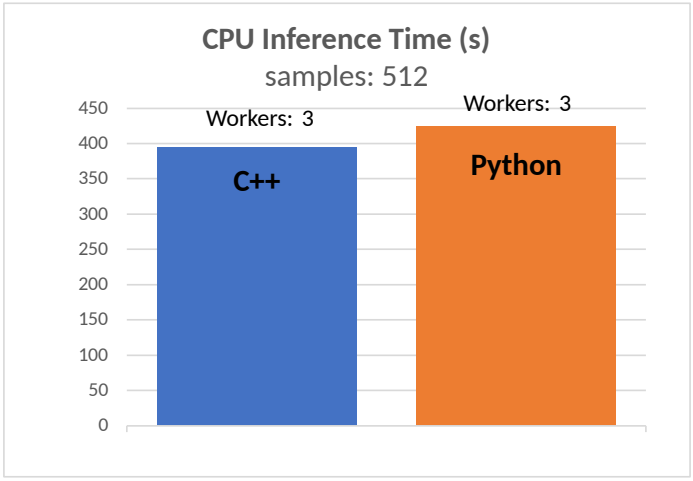
7.4.1 Inference

Prvním testem bylo měření doby inference modelu v závislosti na více proměnných. První testovanou proměnnou byl počet pracovníků/subprocesů/vláken načítajících data (atribut `num_workers` v PyTorch). Ve výchozím nastavení je hodnota této proměnné rovna 0, což znamená, že data budou načítána hlavním procesem. V tomto případě může jakékoli načítání dat pozastavit výpočet. Teoreticky by měl vyšší počet pracovníků vést k efektivnějšímu načítání dat. Obecně se doporučuje, aby byl počet pracovníků roven 4násobku počtu GPU, záleží ovšem na konkrétním hardware. Další testovanou proměnnou byl caching, kdy byl celý dataset načten do RAM před začátkem testu. Tento krok však nejevil vliv na výkon. Test byl proveden samostatně pro GPU i CPU. Nakonec byla také u Python frontendu otestována v rámci GPU funkcionalita AMP (Automatic Mixed Precision), která v LibTorch stále chybí.

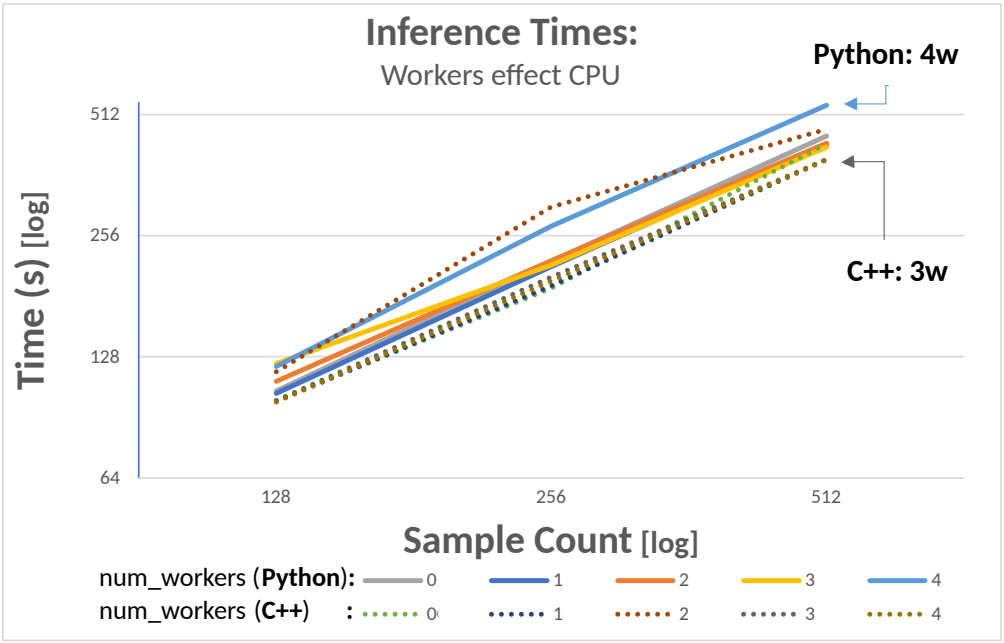
Při testování inference na CPU byly zaznamenané rozdíly v čase mezi oběma frontendy minimální. Při srovnání nejlepších výsledků napříč pracovníky dosahuje LibTorch zhruba 5% urychlení oproti Python frontendu (obr. 7.5 a 7.6). Nejlepších výsledků dosahovaly oba frameworky při použití 3 pracovníků, zatímco nejhorších při použití 4 prac. pro Python frontend a 2 prac. pro LibTorch. Je nutno ovšem podotknout, že pracovníci v tomto případě data pouze načítají z disku a neaplikují žádné náročné transformace, které by mohly rozdíly umocnit.

V rámci druhého testu bylo využito CUDA rozhraní GPU (obr. 7.7 a 7.8). Stejně jako v případě CPU jsou rozdíly v dobách běhu zanedbatelné, přičemž vliv počtu pracovníků se v tomto případě s přibývajícím počtem zpracovávaných vzorků vytrácí. Nakonec, aktivace funkce AMP u Python frontendu vedla ještě k přibližně 4% prodloužení doby výpočtu oproti Python frontendu bez AMP. To lze vysvětlit např. tím, že náklady na konverzi typů převažují nad výhodami plynoucími z výpočtů s menšími čísly.

Přestože LibTorch dosahuje konzistentně lehce příznivějších výsledků, výkony obou frontendů jsou v kategorii inference velmi podobné.



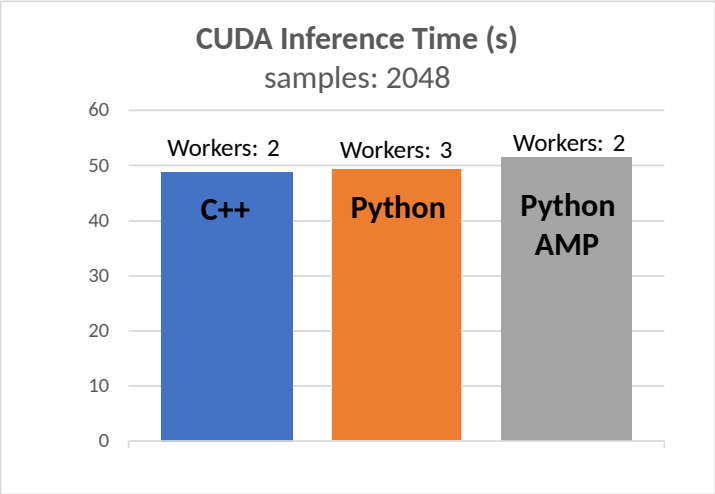
Obrázek 7.5: Srovnání inferenčních časů pro CPU
(byl vybrán nejnižší čas napříč všemi počty pracovníků)



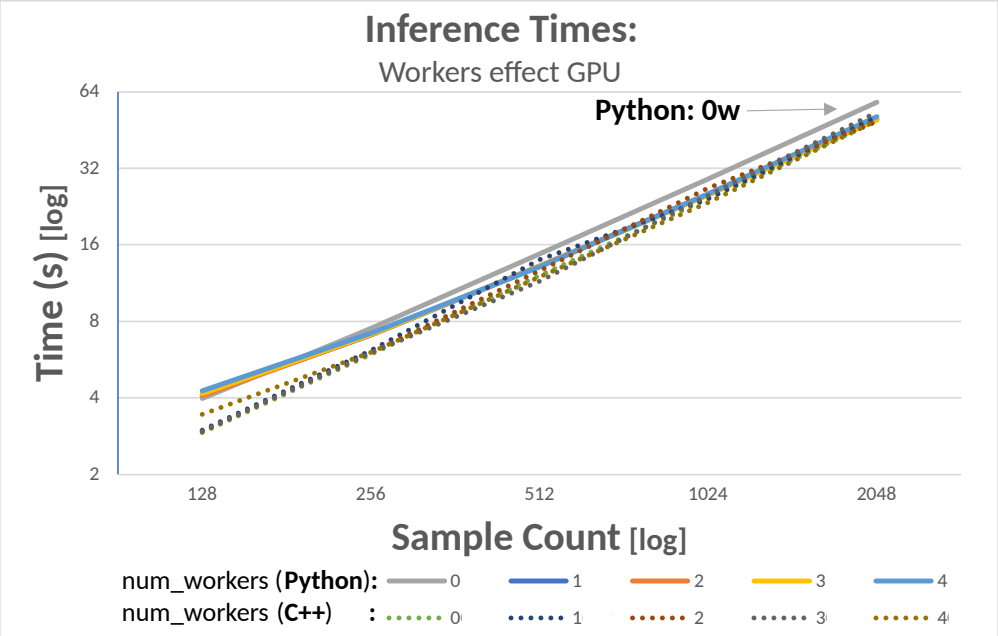
Obrázek 7.6: Inferenční čas v závislosti na počtu pracovníků pro CPU

7.4.2 Trénování

Následný test porovnával u obou frontendů doby trénování modelu a také přesnost výsledného natrénovaného modelu. Pro určení optimálního počtu pracovníků byla nejprve provedena krátká trénování (bez ukládání modelu) s různými počty pracovníků. Na základě těchto výsledků byl vybrán počet pracovníků, který odpovídal nejkratší době běhu. Pro trénování se obecně předpokládá využití GPU, v tomto



Obrázek 7.7: Srovnání inferenčních časů pro CUDA
(byl vybrán nejnižší čas napříč všemi počty pracovníků)



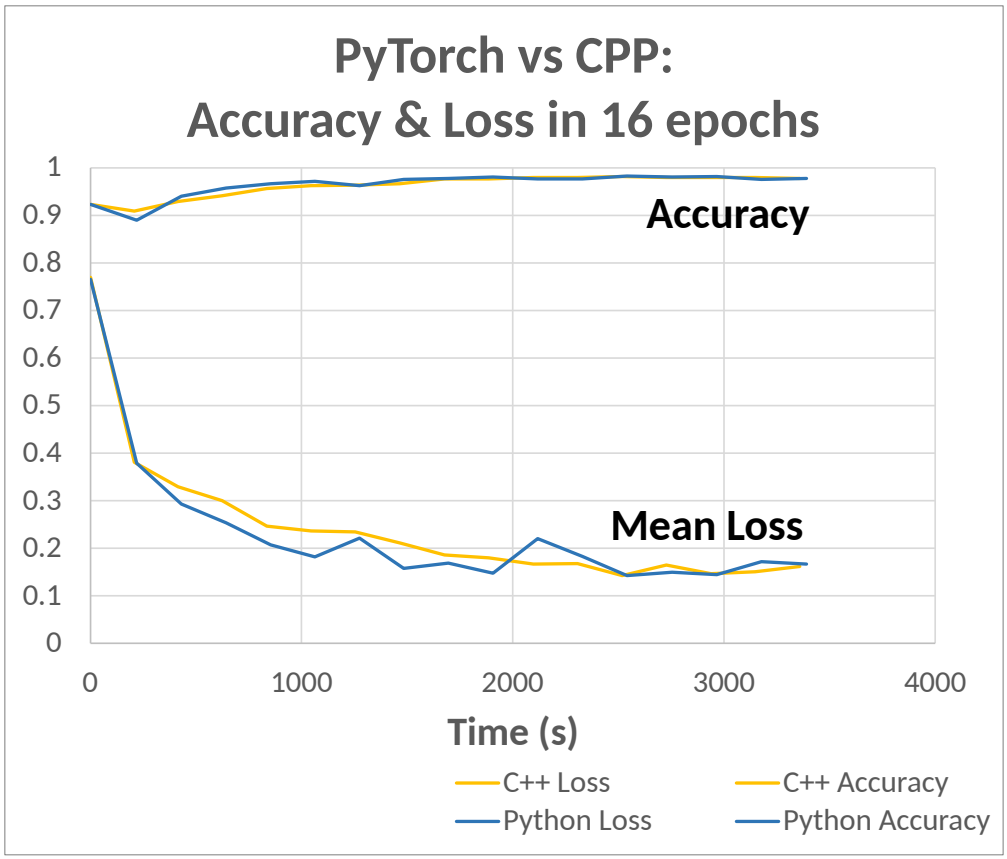
Obrázek 7.8: Inferenční čas v závislosti na počtu pracovníků pro CUDA

testu byl však měřen i výkon na CPU, což může být relevantní pro uživatele, kteří k výkonným grafickým kartám nemají přístup. Na obou verzích byl model na GPU trénován na 16 epochách. I v tomto případě byla otestována funkcionalita AMP.

Nejprve byl spuštěn test na CPU. Z časových důvodů byla provedena pouze 1 epocha při 5% počtu vzorků oproti GPU verzi. Podobně jako u inference byly v případě trénování nejlepší výsledky napříč pracovníky prakticky shodné (rozdíly menší než 1 %).

Následoval test trénování na GPU. I v tomto případě však nebyl zaznamenán větší rozdíl (obr. 7.10). Model sice již od 8. epochy jevil znaky overfittingu[3] (viz obr. 7.9), ovšem na měření rychlosti trénování neměl tento fakt vliv. Na obr. 7.9 stojí za zmínku poměrně vysoká počáteční hodnota přesnosti (accuracy). Tento jev je však způsoben povahou map textových bublin v datasetu, kdy je velká část mapy černá, a zároveň shodou okolností je počáteční výstup nenatrénovaného modelu plně černá mapa. [4]

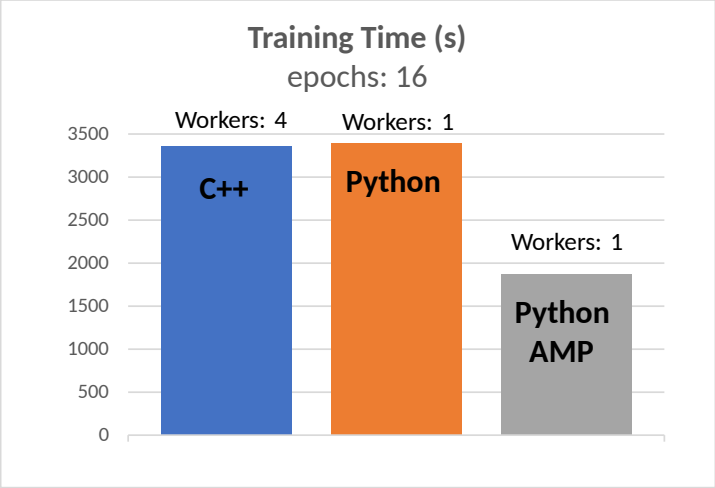
Výrazného rozdílu bylo dosaženo aktivováním funkce AMP u Python frontendu. To vedlo ke 44% poklesu trénovacího času oproti LibTorch bez ovlivnění přesnosti výsledného modelu (viz tab. 7.1, obr. 7.10).



Obrázek 7.9: Trénování v průběhu 16 epoch
(naměřené metriky se vztahují k testovacímu datasetu)

[3] Na trénovacím datasetu ztráta (loss) v průběhu epoch stále klesala, ale u testovacího datasetu po přibližně 8. epoše stagnovala.

[4] Jako funkce přesnosti byl zvolen poměr správných pixelů oproti všem.



Obrázek 7.10: Délka trénování GPU
(byl vybrán nejnižší čas napříč všemi počty pracovníků)

	Time (min)	Time (rel)	Loss	Accuracy
Python	56.50	1.01	0.142	0.982
C++	55.99	1.00	0.142	0.982
PythonAMP	31.14	0.56	0.116	0.984

Tabulka 7.1: Časy a výsledky trénování GPU

Závěr 8

Cílem této práce bylo implementovat C++ verzi Python knihovny nnframework a následně porovnat výkon obou verzí na zvoleném modelu. V první části byla navrhnuta a implementována základní verze knihovny. Následně byl zvolen vhodný model pro porovnání: Unet, který byl již v originální knihovně implementován. V druhé části byl testován výkon obou verzí nnframeworku. Nejprve byl vybrán vhodný dataset, který byl pro potřeby testování augmentován. S využitím datasetu byla realizována řada testů porovnávajících výkon obou verzí knihovny v závislosti na několika parametrech: počet pracovníků, typ zařízení, cache, AMP.

C++ verzi nnframeworku se podařilo úspěšně implementovat se základní sadou funkcí pro snadnou implementaci modelů a práci s nimi. Následně byl realizován model Unet, na kterém byla také funkčnost C++ knihovny otestována. Implementace modelu přispěla k dalšímu rozšíření funkcionality a snazšímu použití C++ knihovny.

Z naměřených výsledků vyplývá, že obě verze nnframeworku (Python i C++) dosahují velmi podobného výkonu s výjimkou GPU trénování s funkcí AMP, která je k dispozici pouze u Python frontendu. Její použití vedlo k více než dvojnásobnému zrychlení oproti C++ verzi bez AMP. Pro vývoj lze tedy doporučit spíše Python frontend. Python frontend jakožto primární rozhraní PyTorch disponuje větší komunitní podporou, obsahuje větší počet funkcí a mimo to nabízí rozsáhlejší dokumentaci. Při použití Pythonu také odpadá nutnost kompilace, což vývoj značně urychluje. C++ frontend naopak postrádá významné funkce (AMP) a dokumentace určitých části je neexistující (Kineto profiler). Na druhou stranu C++ frontend dosahuje v inferenci lehce příznivějších výsledků a nevyžaduje Python prostředí, což ho umožňuje nasadit jako samostatnou aplikaci bez dalších závislostí. Lze ho tedy doporučit pro nasazení v produkčním prostředí.

Mezi budoucí vylepšení C++ knihovny by mohly patřit další funkce originální knihovny, jako např. podpora jazykových modelů, rozšíření palety výchozích metrik či implementace více PyTorch Modulů. Pro obsáhlejší testování na CPU se nabízí použití knihovny MKL-DNN, která optimalizuje výpočet na procesorech Intel. Nakonec pro získání obecnějších výsledků by bylo vhodné testování provést na více rozdílných zařízeních.

Základní uživatelská příručka

A

Pro bližší seznámení s knihovnou lze doporučit textové soubory zahrnuté v projektovém adresáři: `README.md` a `MANUAL.md`. `README` poskytuje základní informace o projektu, zatímco `MANUAL` podrobně popisuje strukturu knihovny na jednoduchém MNIST modelu, který je také zahrnut ve zdrojovém kódu. Následující návod popisuje proces konfigurace projektu na Windows a následně také sestavení programu pro trénování a testování modelu MNIST.

A.1 Sestavení a spuštění

A.1.1 OpenCV

Nejprve je nutno nainstalovat knihovnu OpenCV. Knihovna musí být dohledatelná CMake příkazem `findPackage(OpenCV)`. Obvykle stačí stáhnout a extrahovat oficiální verzi binární distribuci (verze 4.6.0) do libovolné složky, např. `C:\opencv`. Následně je nutno přidat systémovou proměnnou `OpenCV_DIR`, která ukazuje do `C:\opencv\build` a dále přidat cestu `C:\opencv\build\x64\vc15` do systémové proměnné `PATH`. Korektní instalaci lze ověřit příkazem `opencv_version`.

A.1.2 CUDA

Pro využití GPU verze LibTorch (která je ve výchozím nastavení stažena a všechny příklady ji používají) je nutno nainstalovat také NVIDIA CUDA Toolkit dostupný na adrese `https://developer.nvidia.com/cuda-toolkit`.

Pokud chce uživatel používat pouze CPU verzi LibTorch, není třeba CUDA instalovat, ale je nutno upravit URL adresy pro stažení CPU verzí LibTorch v souboru `DownloadLibTorch.cmake`, například CPU Release by mohla vypadat následovně:

```
set(LIBTORCH_LINK_Release
"https://download.pytorch.org/libtorch/cpu/libtorch-win-
    shared-with-deps-2.0.0%2Bcpu.zip")
```

A.1.3 Generování projektového souboru

Po úspěšné instalaci knihoven je nutno konfigurovat a sestavit projekt. Tento proces je automatizován pomocí skriptů `MakeBuild<Config>.bat` v závislosti na požadované konfiguraci. Tento skript vytvoří příslušnou build složku a spustí generování CMake projektu. Vytváření projektu poprvé může trvat déle, jelikož CMake automaticky stáhne a extrahuje LibTorch knihovnu do složky `vendor`. Následně stačí otevřít vygenerovaný soubor `NNCProj.sln` Visual Studiem 2022. Alternativně je také možné otevřít kořenovou složku projektu jako nativní CMake projekt ve Visual Studiu.

A.1.4 Sestavení programu

Ve zdrojové složce projektu se nachází složka `examples`, která obsahuje více spustitelných příkladů neuronových modelů. Pro vybrání požadovaného příkladu stačí odkomentovat příslušnou řádku v metodě `main()` souboru `src/examples/example_entry_point.cpp`:

```
int main(){
    nn::Log::init();

    // mnist example is chosen
    NN_RUN_EXAMPLE(main_mnist);

    return 0;
}
```

Ve výchozím nastavení projektu je tento příklad již vybrán. (Pro vyřazení složky `examples` z překladu stačí nastavit CMake přepínač `NN_EXAMPLES`.) Nyní stačí program sestavit. Do složce se spustitelným souborem se při sestavení automaticky nakopírují také všechny nutné `.dll` knihovny.

A.1.5 Stažení datasetu MNIST

Před spuštěním MNIST příkladu je nutno stáhnout MNIST dataset do složky `data`. Tento proces je rovněž automatizován pomocí skriptu `data\DownloadMNIST.bat`.

A.1.6 Spuštění

MNIST příklad by nyní měl být spustitelný. Program nalezne MNIST dataset, natrénuje model na 4 epochách a nakonec provede 10 testů rozpoznávání a vypíše přesnost modelu. Pokud je program spouštěn samostatně, je nutné, aby byla složka `data` ve stejné složce jako spustitelný soubor.

Struktura přiloženého souboru

- `Aplikace_a_knihovny`
  - `NNC` ..... Repositář NNC frameworku
    - `data` . pro ukládání trénovacích dat, natrénovaných modelů, obsahuje skripty pro stažení datasetů
    - `src` ..... C++ zdrojový kód frameworku
    - `vendor` ..... Knihovny třetích stran: spdlog, LibTorch (bude stáhnut automaticky)
    - `MakeBuild<Config>.bat` ... Generuje build složku s Visual Studio projektem
    - `CMakeLists.txt` .....Popisuje strukturu CMAKE projektu
    - `README.md` .....Základní informace o projektu
    - `MANUAL.md` ..... Podrobný návod, vysvětluje koncepty a strukturu frameworku na modelu MNIST
- `Text_prace`
  - `Latex` ..... Latex projekt, šablona Ing. Kamila Ekšteina, Ph.D
    - `img` ..... Obrázky a grafy
    - `texmf` ..... Fonty
    - `manual.tex` ..... Zdrojový kód bakalářské práce
    - `output.pdf` ..... Elektronická verze bakalářské práce
- `Vysledky`
  - `graphs.xlsx` ... Naměřené metriky modelu Unet, srovnávací grafy pro inferenci a trénink

Bibliografie

1. HAYKIN, Simon. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1998. ISBN 978-0132733502.
2. MACUKOW, Bohdan. Neural Networks – State of Art, Brief History, Basic Models and Architecture. In: SAEED, Khalid; HOMENDA, Władysław (ed.). *Computer Information Systems and Industrial Management*. Cham: Springer International Publishing, 2016, s. 3–14. ISBN 978-3-319-45378-1. Dostupné také z: https://link.springer.com/chapter/10.1007/978-3-319-45378-1_1.
3. NIELSEN, Michael A. *Neural Networks and Deep Learning*. Determination Press, 2015. Dostupné také z: http://neuralnetworksanddeeplearning.com.
4. *PyTorch*. PyTorch, 2023. Dostupné také z: https://pytorch.org. Version 2.0.
5. *NVIDIA Nsight Systems*. NVIDIA Corporation, 2023. Dostupné také z: https://developer.nvidia.com/nsight-systems.
6. *Papers With Code: Trends* [online]. MetaAI, 2023. [cit. 2023-03-30]. Dostupné z: https://paperswithcode.com/trends.
7. *Hugging Face*. Hugging Face, 2023. Dostupné také z: https://huggingface.co.
8. O'CONNOR, Ryan. *PyTorch vs TensorFlow in 2023* [online]. [cit. 2023-03-30]. Dostupné z: https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/.
9. BEHNEL, Stefan; BRADSHAW, Robert; CITRO, Craig; DALCIN, Lisandro; SELJEBOTN, Dag Sverre. *The Cython Programming Language*. Cython, 2023. Dostupné také z: https://cython.org.
10. JAKOB, Wenzel. *nanobind: tiny and efficient C++/Python bindings*. 2022. Dostupné také z: https://github.com/wjakob/nanobind.
11. *Vulkan: High-Efficiency Graphics and Compute API*. Khronos Group, 2023. Dostupné také z: https://www.khronos.org/vulkan.
12. *PyTorch C++ API* [online]. [cit. 2023-03-30]. Dostupné z: https://pytorch.org/cppdocs.

13. *PyTorch C++ Frontend* [online]. [cit. 2023-03-30]. Dostupné z: https://pytorch.org/cppdocs/frontend.html.
14. *Git Version Control System*. San Francisco, CA: Git, 2023. Dostupné také z: https://git-scm.com.
15. *CMake*. Kitware, 2023. Dostupné také z: https://cmake.org. Version 3.25.0.
16. *Premake*. Premake, 2023. Dostupné také z: https://premake.github.io. Version v5.0.0-beta2.
17. *Installing C++ Distributions of PyTorch* [online]. [cit. 2023-04-07]. Dostupné z: https://pytorch.org/cppdocs/installing.html.
18. *OpenCV*. OpenCV Team, 2023. Dostupné také z: https://opencv.org.
19. MELMAN, Gabi. *spdlog*. 2023. Dostupné také z: https://github.com/gabime/spdlog. Fast C++ logging library.
20. RUSTINGSWORD. *TensorBoard Logger*. 2022. Dostupné také z: https://www.github.com/RustingSword/tensorboard_logger.
21. *Visual Studio 2022*. Microsoft Corporation, 2023. Dostupné také z: https://www.visualstudio.microsoft.com/vs. Version 17.5.
22. RONNEBERGER, Olaf; FISCHER, Philipp; BROX, Thomas. *U-Net: Convolutional Networks for Biomedical Image Segmentation*. 2015. Dostupné z arXiv: 1505.04597 [cs.CV].
23. O'SHEA, Keiron; NASH, Ryan. *An Introduction to Convolutional Neural Networks*. 2015. Dostupné z arXiv: 1511.08458 [cs.NE].
24. YAMASHITA, Rikiya; NISHIO, Mizuho; DO, Richard Kinh Gian; TOGASHI, Kaori. Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*. 2018, roč. 9, č. 4, s. 611–629. ISSN 1869-4101. Dostupné z DOI: 10.1007/s13244-018-0639-9.
25. GHOLAMALINEZHAD, Hossein; KHOSRAVI, Hossein. *Pooling Methods in Deep Neural Networks, a Review*. 2020. Dostupné z arXiv: 2009.07485 [cs.CV].
26. *Upsample - PyTorch 2.0 Documentation* [online]. [cit. 2023-03-30]. Dostupné z: https://pytorch.org/docs/stable/generated/torch.nn.Upsample.html.
27. *OpenAI*. OpenAI, 2023. Dostupné také z: https://openai.com.
28. *cuDNN: NVIDIA CUDA Deep Neural Network Library*. NVIDIA Corporation, 2023. Dostupné také z: https://developer.nvidia.com/cudnn.
29. *Intel(R) Math Kernel Library for Deep Neural Networks*. Intel Corporation, 2023. Dostupné také z: https://oneapi-src.github.io/oneDNN/v0.

Seznam obrázků

Seznam tabulek

Seznam výpisů