FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Generátor modelu neuronové sítě do VHDL
Student:	Jan Medek
Vedoucí:	Ing. Miroslav Skrbek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Proveďte rešerši generátorů modelů neuronových sítí do jazyka VHDL a C.
Na základě této rešerše navrhněte a implementujte generátor z neuronového modelu z Tensorflow případně obdobných frameworků do VHDL.
Zaměřte se hlavně na generování integrujících částí VHDL popisu za předpokladu existujících funkčních bloků ve formě operátorů, neuronů, vrstev apod. Generátor musí umět extrahovat topologii a parametry z modelu, parametry konvertovat do datových typů požadovaných implementací neuronové sítě a jejich uložení ve formě kompatibilní s cílovou platformou.
U generátoru umožněte dostatečnou míru konfigurace tak, aby bylo možné zohlednit specifika a potřeby výzkumu na katedře. Generované VHDL musí být syntetizovatelné vývojovým systémem Vivado.
Generátor implementujte v jazyce Python, zdrojový kód řádně zdokumentujte a okomentujte.
Rozsah práce upřesněte po dohodě s vedoucím práce.

Elektronicky schválil/a prof. Ing. Hana Kubátová, CSc. dne 27. prosince 2022 v Praze.

Bakalářská práce

GENERÁTOR MODELU NEURONOVÉ SÍTĚ DO VHDL

Jan Medek

Fakulta informačních technologií
Katedra číslicového návrhu
Vedoucí: Ing. Miroslav Skrbek, Ph.D.
11. května 2023

Odkaz na tuto práci: Medek Jan. *Generátor modelu neuronové sítě do VHDL*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Seznam obrázků

Seznam výpisů kódu

Chtěl bych poděkovat především vedoucímu mé práce, panu Ing. Miroslavu Skrbkovi, Ph.D., za příjemnou spolupráci, skvělý přístup a ochotu se vším pomoct. Dále chci poděkovat panu Ing. Pavlu Kubalíkovi, Ph.D. za cenné konzultace ohledně VHDL kódů. V závěru chci poděkovat mé rodině, přítelkyni a dalším známým, kteří mě podporovali nejenom při tvorbě této práce, ale i v průběhu celého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 11. května 2023 .....

Abstrakt

Tato práce se zabývá návrhem a implementací generátoru hardwarového popisu v jazyce VHDL ze softwarového modelu neuronové sítě, popsaného v Keras API. Důraz je také kladen na konfiguraci datových typů parametrů sítě a jejich export. Práce analyzuje Keras API a existující řešení generování VHDL ze softwarových modelů. V rámci práce bylo navrženo řešení, pomocí kterého lze generovat VHDL přímo z Keras modelu bez dalších mezistupňů. Výsledkem práce je funkční aplikace implementovaná v jazyce Python, která je schopná z modelu, popsaného v Kerasu, a předložených VHDL stavebních bloků vygenerovat syntetizovatelné VHDL a exportovat parametry sítě konvertované do datových typů specifikovaných konfigurací.

Klíčová slova neuronové sítě, hardwarová akcelerace, generátor VHDL, Keras API, Python

Abstract

This thesis covers design and implementation of a generator, which generates hardware description in VHDL language from software model of a neural network designed in Keras API. It also covers a configuration of network parameters in a way, that allows to convert them to data types specified by the created configuration and is able to export them. The thesis analyzes Keras API, existing implementations which generate VHDL from the software models and proposes a solution, which achieves generating the VHDL straight from the Keras model without any stages in between. The result is a working application implemented in Python that is able to take a model implemented in Keras with given building blocks written in VHDL, generate synthesizable VHDL and export network's parameters converted to data types specified by the configuration.

Keywords neural networks, hardware acceleration, VHDL generator, Keras API, Python

Seznam zkratek

API	Application Programming Interface
FPGA	Field Programmable Gate Array
HLS	High Level Synthesis
HLS4ML	High Level Synthesis for Machine Learning
JSON	JavaScript Object Notation
ReLU	Rectified Linear Unit
RGB	Red Green Blue
VHDL	VHSIC Hardware Description Language
YAML	Yet Another Markup Language

Úvod

Umělá inteligence nás v dnešní době provází napříč mnoha odvětvími. Rozpoznávání obrazu, robotika, autonomní vozidla ale i předpovídání počasí, lidských chorob a dalších událostí na základě opakujících se vzorů — to všechno jsou některé z mnoha oblastí, kde se tato věda uplatňuje. Základem umělé inteligence jsou lidskými mozky inspirované *neuronové sítě*, které jsou schopny řešit matematické a výpočetně složité problémy.

Umělá inteligence a její neuronové sítě jsou často simulovány na počítačích, tedy implementovány *softwarově*. To znamená, že model neuronové sítě je popsán programem a stavební bloky sítě jsou reprezentovány běžnými datovými strukturami v paměti počítače. Tento přístup je sice funkční, ale pro složitější sítě je výpočet příliš pomalý.

Problém pomalého výpočtu lze vyřešit akcelerací algoritmů na GPU nebo tak, že se pro stavební bloky neuronové sítě navrhne *specializovaný hardware*, který od základu funguje paralelně a lze ho maximálně optimalizovat. Avšak oproti počítačům je specializovaný hardware méně flexibilní. Výhodou specializovaného hardwaru je možnost implementace formou zákaznických integrovaných obvodů nebo na FPGA. FPGA jsou sice o něco pomalejší, ale mají řadu výhod. Návrh na nich lze jednoduše testovat, verifikovat a vidět výsledky provedených změn návrhu v rámci několika okamžiků. V obou případech se hardware navrhuje v jazyce pro popis hardwaru, například ve VHDL nebo Verilogu.

Pokud se implementují neuronové sítě přímo v hardwaru a v nějakém z uvedených jazyků, tak je vhodné zajistit, aby byly jednotlivé stavební bloky parametrizovány pomocí nějakých konstant, například pro počty vrstev či neuronů. Při vývoji nových neuronových sítí je totiž potřeba experimentovat s počty stavebních bloků a kdyby se musel návrh pokaždé předělávat pro konkrétní verzi modelu, byl by vývoj velmi zdlouhavý a neefektivní. Když budou všechny bloky parametrizovány zmíněnými konstantami, tak se nabízí možnost nastavovat je automatizovaně, pokud budou dostupné informace o tom, jaké stavební bloky sítě použít a také kolik jich použít. Pokud by byl model sítě popsaný softwarově, tak by se z něj mohla zjistit topologie a podle ní se vygenerovaly příslušné hardwarové bloky, dohromady tvořící celou síť. Výsledkem těchto úvah by byl generátor, který je schopen ze softwarového modelu vygenerovat návrh hardwaru v jednom z jazyků pro popis hardware. Výhodou tohoto postupu by také bylo, že poskytuje vysokou úroveň abstrakce — uživatel nemusí znát implementační detaily hardwaru. Pracoval by pouze s modelem neuronové sítě na vyšší úrovni, podobně jako tomu je u moderních jazyků, kde překlad do strojového kódu zajišťuje kompilátor.

Cíle práce

Cílem této práce je nejprve nastudovat knihovnu Keras, pomocí které lze popisovat softwarové modely neuronových sítí. Dále je cílem analyzovat existující řešení generování VHDL ze softwarového modelu sítě a navrhnout funkční generátor syntetizovatelného VHDL, který je schopen z modelu, popsaného v Kerasu, extrahovat topologii, parametry a další potřebné informace k tomu, aby bylo možné vygenerovat VHDL pro tento model. Také je potřeba zajistit dostatečnou míru konfigurace parametrů sítě a jejich převedení do specifikovaných datových typů. Požadavkem také je, aby byl generátor implementován v jazyce Python a řádně zdokumentován. Předpokladem pro generování jsou hotové stavební bloky sítě popsané ve VHDL, které mi byly poskytnuty vedoucím práce.

Struktura práce

V Kapitole 1 se zabývám stručným úvodem do teorie neuronových sítí. Vysvětluji, co je to neuronová síť a z čeho se skládá. Zmiňuji dvě možné struktury neuronu a dále vysvětluji váhy, biasy a některé vybrané aktivační funkce. V závěru uvádím některé ze základních typů vrstev, včetně jejich stručného vysvětlení.

V Kapitole 2 provádím analýzu použitých technologií, existujících řešení generování VHDL a v závěru popisuji návrh mého řešení. Z použitých technologií popisuji Keras API a použité Python knihovny v rámce mého řešení. U existujících řešení zmiňuji přednosti a nedostatky a věnuji se hlavně řešení hls4ml. V závěru kapitoly se zabývám návrhem mého řešení a popisu jednotlivých částí.

V Kapitole 3 se zabývám implementací navrženého řešení z předešlé kapitoly. Popisuji rozhraní vrstev, konfigurace, transformovaného modelu a generování VHDL. Ve všech těchto částech uvádím vybrané části kódu a principy znázorňuji sekvenčními diagramy.

V Kapitole 4 popisuji způsob, jakým jsem aplikaci testoval a poté uvádím možná budoucí rozšíření implementované aplikace. Testování jsem rozdělil do dvou částí.

V závěrečné kapitole poté už jen shrnuji dosažené výsledky.

Součástí práce jsou také dvě přílohy. V Příloze A v jednotlivých krocích popisuji, jak vytvořenou aplikaci nainstalovat. V Příloze B popisuji základní použití aplikace a odkazuji v ní na přiloženou dokumentaci.

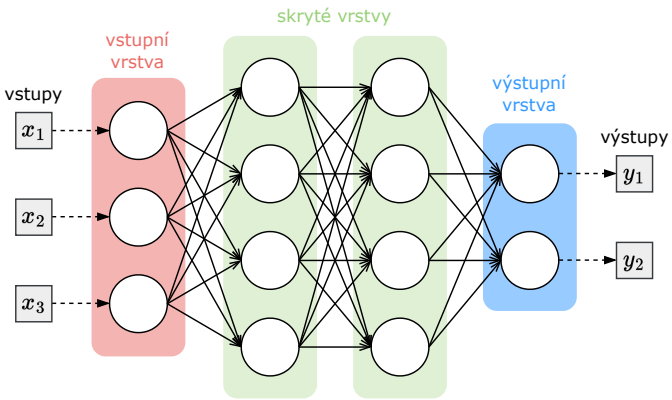
Kapitola 1
Úvod do neuronových sítí

V této kapitole se zabývám základní teorií neuronových sítí. Nejprve v sekci 1.1 vysvětluji, co je to neuronová síť a dále v sekci 1.2 představuji strukturu neuronu a popisuji jeho jednotlivé části. Ke konci kapitoly v sekcích 1.3 až 1.7 zmiňuji některé typy vrstev a stručně vysvětluji, jaké jsou jejich vlastnosti.

1.1 Neuronová síť

Neuronová síť[1] je matematický výpočetní model, který pracuje paralelně. Skládá se ze vzájemně propojených *umělých neuronů*, jejichž propojení představuje tok dat s příslušnými *váhami*. Váhy jsou upravovány v procesu *učení*. Vlastnosti sítě lze upravovat i jinými způsoby, například změnou počtu neuronů nebo vrstev.

Jednou ze základních sítí je *dopředná*[2] neuronová síť, která se skládá ze *vstupní* vrstvy (viz sekce 1.3), jedné nebo více *skrytých* vrstev a *výstupní* vrstvy. Příklad topologie takovéto sítě lze vidět na obrázku 1.1. Dopředné sítě jsou dle [1] *univerzálními aproximátory funkcí* — navržením vhodné topologie a parametrů lze pomocí nich reprezentovat jakoukoliv funkci.



Obrázek 1.1 Příklad topologie dopředné neuronové sítě

Dalším příkladem dopředné neuronové sítě je *konvoluční* síť, která se využívá v hlubokých neuronových sítích pro rozpoznávaní obrazu. Rozpoznávaný obrázek vstupuje do sítě přes *vstupní*

[1] tento pojem bude dále v textu vždy označovat umělé neuronové sítě
[2] feed-forward

vrstvu, ze které se data propagují skrz sadu *konvolučních* (viz sekce 1.5) a dle [2] několik pomocných vrstev — *pooling* (viz sekce 1.6) či *flatten* (viz sekce 1.7), extrahující příznaky z obrázku. Následují klasifikační vrstvy, skládající se většinou z *dense* vrstev (viz sekce 1.4). Poslední vrstva v síti je výstupní.

1.2 Neuron

Historie umělých neuronů sahá až do roku 1943, kdy byl zveřejněn článek [3], představující matematické popisy neuronů a jejich sítí včetně důkazu, že mohou implementovat jakoukoliv myšlenku vyjádřenou jazykem matematické logiky. Na to navazoval text [4] z roku 1958, představující *perceptron*, ze kterého dle [2] dědí dnešní umělé neurony některé vlastnosti. V tomto textu se dále budu zabývat pouze moderními umělými neurony používanými dnes.

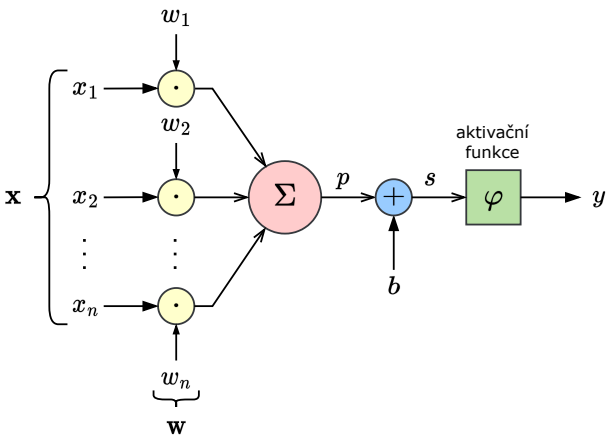
Neuron je základní výpočetní jednotkou v neuronové síti. Obrázek 1.2 znázorňuje nejběžnější strukturu neuronu, který se nachází ve většině moderních sítí. Do neuronu vstupuje vektor $\mathbf{x} = (x_1, x_2, \ldots, x_n)$. Vstupnímu vektoru přísluší vektor vah $\mathbf{w} = (w_1, w_2, \ldots, w_n)$, který je součástí neuronu. Uvnitř neuronu vzniká suma s, která se spočte jako skalární součin p vstupního a váhového vektoru, ke kterému je přičteno b, neboli *bias*. Suma je tedy definována jako

$$s = p + b = (\mathbf{w} \cdot \mathbf{x}) + b = \sum_{i=1}^{n} w_i x_i + b. \tag{1.1}$$

Na tento součet je typicky aplikována *aktivační funkce* φ, která může do systému vnášet nelinearitu. Výstup y neuronu lze popsat vztahem

$$y = \varphi(s), \tag{1.2}$$

kde φ může být ve specifických případech identické zobrazení, pokud se nepoužije žádná aktivační funkce.

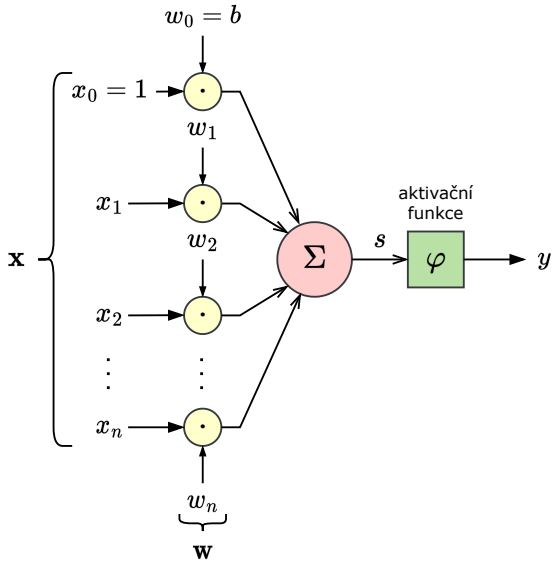


Obrázek 1.2 Schéma umělého neuronu

Obrázek 1.3 znázorňuje alternativní strukturu neuronu, kde se bias zařazuje jako další váha, u které se odpovídající vstupní hodnota nastavuje na 1. Suma s je potom definována jako

$$s = \mathbf{w} \cdot \mathbf{x} = \sum_{i=0}^{n} w_i x_i, \tag{1.3}$$

kde $\mathbf{x}$ má navíc složku $x_0 = 1$ a $\mathbf{w}$ složku $w_0 = b$. Výstup y to nemění, protože obě definice sumy s jsou matematicky ekvivalentní. Tato modifikace se ale používá z praktických důvodů, potencionálně zjednodušující některé implementace.



Obrázek 1.3 Alternativní schéma umělého neuronu

1.2.1 Váhy

Váhy jsou jedny z parametrů neuronové sítě, které určují její chování. Neuronové sítě se pomocí nich jsou schopné *učit* rozeznávat informace na jejich vstupu [2]. Hodnoty vah se nastavují v procesu *učení* pomocí dostupných algoritmů — nejčastějším z nich je algoritmus *zpětného šíření*[3], což je gradientní optimalizační algoritmus. Tento algoritmus minimalizuje chybu mezi výstupem neuronové sítě a požadovaným výstupem z trénovací množiny pomocí *gradientního sestupu*[4] [5]. Proces učení často bývá náročný, a proto se akceleruje na grafických kartách.

Jednotlivé váhy konkrétního neuronu dohromady tvoří vektor $\mathbf{w}$, jak lze vidět na obrázku 1.2. Při modelování konkrétní neuronové sítě máme požadavek na to, jakou funkci má síť reprezentovat. Cílem je najít takové hodnoty vah, aby funkce, kterou síť představuje, aproximovala požadovanou funkci jak nejlépe je možné. Hledání těchto vah se nazývá proces učení. V procesu učení hraje důležitou roli *trénovací množina*, která se skládá z párů vstupních a výstupních vektorů. Vstupní vektory trénovací množiny se pošlou do sítě a cílem je, aby se výstupy sítě co nejvíce blížily hodnotám výstupních vektorů v trénovací množině [6]. Váhy se upravují pomocí zmíněných algoritmů tak dlouho, dokud není dosažena požadovaná úspěšnost správného výstupu sítě.

1.2.2 Bias

Bias je dalším parametrem neuronové sítě, který určuje její chování. Je to skalární veličina, jejímž použitím lze provést *afinní transformaci* nad p z rovnice 1.1, což má za následek posunutí aktivační funkce φ po ose x [7]. Podobně jako váhy, bias je parametrem pro každý neuron a jeho hodnota se vypočítává v procesu učení.

[3] backpropagation algorithm
[4] gradient descent

1.2.3 Aktivační funkce

V rovnicích 1.1 a 1.3 lze pozorovat, že veškeré operace jsou lineární (skalární součin je lineárním zobrazením). Aby neuronová síť splňovala vlastnost být univerzálním aproximátorem funkcí, musí být schopna zpracovávat i nelineární problémy. K tomu slouží *aktivační funkce*, které jsou důležitou součástí neuronových sítí, přinášející do systému zmíněnou nelinearitu, pokud jsou samy nelineární. Nutnou podmínkou pro použití konkrétní aktivační funkce je, aby byla *diferencovatelná*. To je důležité v procesu učení, zejména při použití metody zpětného šíření [8]. V následujících řádcích stručně popíšu některé z nejpoužívanějších aktivačních funkcí.

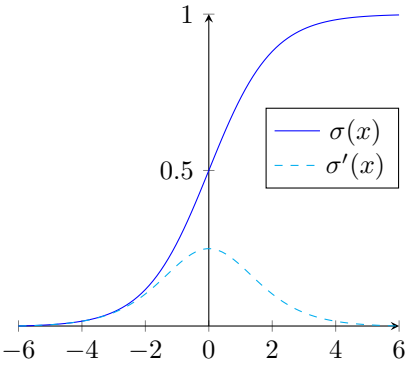
1.2.3.1 Sigmoida

Sigmoida je jedna z nejpoužívanějších aktivačních funkcí díky své nelinearitě. Její předpis je definován jako

$$\sigma(x) = \frac{1}{1 + e^{-x}}, \tag{1.4}$$

kde $\sigma(x) \in (0, 1)$ pro $x \in (-\infty, +\infty)$. Protože je spojitá na celém definičním oboru, je diferencovatelná, čímž splňuje požadavek na aktivační funkce.

Graf sigmoidy a její derivace znázorňuje obrázek 1.4. Díky tomu, že je $\sigma'(x)$ jednoduchá na výpočet, se sigmoida používá v sítích s malým počtem vrstev [8]. Avšak dle [9], sítě s více jak pěti vrstvami trpí problémem zvaným *vanishing gradient problem*, jehož příčinou je průběh $\sigma(x)$. Problém spočívá v tom, že pro velká kladná a záporná x se derivace této funkce blíží k nule a při zpětné propagaci gradientu neuronovou sítí se gradient derivací sigmoidy násobí.



Obrázek 1.4 Graf sigmoidy a její derivace

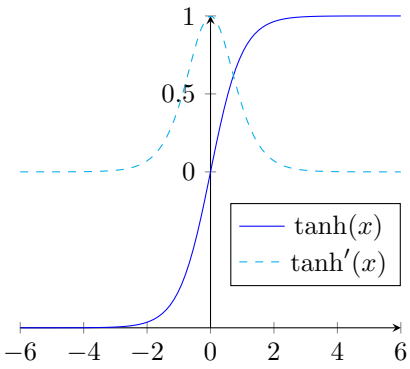
1.2.3.2 Hyperbolický tangens

Hyperbolický tangens je podobný sigmoidě, ale je symetrický podle svých os. Lze ho definovat několika způsoby, například jako

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1, \tag{1.5}$$

kde $\tanh(x) \in (-1, 1)$ pro $x \in (-\infty, +\infty)$ a $\sigma(x)$ je definovaná v 1.4. Tato funkce je stejně jako $\sigma(x)$ spojitá na celém definičním oboru, tedy diferencovatelná.

Graf $\tanh(x)$ a jeho derivaci znázorňuje obrázek 1.5. Derivace této aktivační funkce je oproti sigmoidě sice strmější, díky čemuž má větší obor hodnot [10], ale je složitější na výpočet. Stejně jako sigmoida také trpí problémem *vanishing gradient*, viz sekce 1.2.3.1.



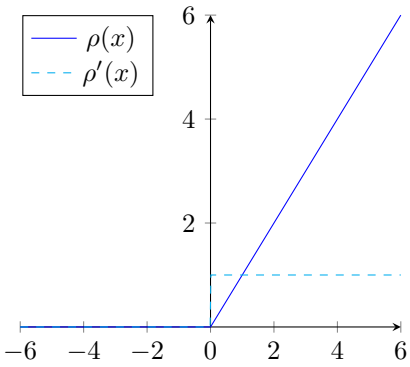
Obrázek 1.5 Graf hyperbolického tangensu a jeho derivace

1.2.3.3 ReLU

Rectified Linear Unit, zkráceně ReLU, je dle [9] nejpopulárnější aktivační funkce. Je to nelineární funkce definována jako

$$\rho(x) = \max(0, x). \tag{1.6}$$

Její graf i derivaci lze vidět na obrázku 1.6. Výpočet ReLU funkce, včetně derivace, je jednodušší než u $\sigma(x)$ a $\tanh(x)$, protože se zde nevyužívá exponciální funkce. Další výhodou je, že saturuje pouze pro $x < 0$. Kvůli tomuto se zde ale objevuje speciální případ *vanishing gradient* problému nazývaný jako „dying ReLU“ [10], kdy je každý záporný výstup neuronu nastaven na hodnotu 0, kvůli čemuž nebudou nadále váhy neuronu aktualizovány a neuron nebude aktivován [9].



Obrázek 1.6 Graf ReLU a její derivace

1.3 Vstupní vrstva

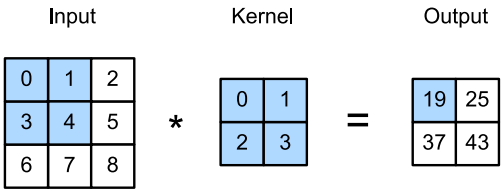
Vstupní vrstva slouží k propagaci vstupních dat do první skryté vrstvy. Je nutné, aby tato vrstva měla m neuronů, kde m je počet vstupních hodnot. V této vrstvě se data žádným způsobem neupravují, a proto musí platit $y_n = x_n$ pro všechna $n \in \{1, \ldots, m\}$, kde x_n je vstup a y_n výstup m-tého neuronu. Toto chování lze matematicky popsat tak, že všechny váhy jsou nastaveny na hodnotu 1, biasy na hodnotu 0 a aktivační funkce jsou identitou.

1.4 Dense vrstva

Dense vrstva, neboli „plně propojená“ vrstva, se vyznačuje úplným propojením s vrstvou předchozí — do každého neuronu dense vrstvy vstupují výstupy všech neuronů z předchozí vrstvy, viz obrázek 1.1. Je to jedna z nejpoužívanějších vrstev a je typická pro mělké neuronové sítě. Najdeme ji také na výstupech hlubokých neuronových sítí.

1.5 Konvoluční vrstva

Konvoluční vrstva je základním stavebním blokem v konvolučních sítích. Tato vrstva provádí *konvoluci* obrázku s *konvolučním jádrem*, neboli *kernelem*, kterým se filtrují hledané příznaky z obrázku. Zpracovávaný obrázek je vždy výrazně větší než konvoluční jádro, které se přes něj posouvá. Jádro obsahuje koeficienty, nad kterými se s odpovídající částí obrázku počítá skalární součin, viz obrázek 1.7, převzat z [11]. Tento proces lze převést do kontextu neuronu. Zavádí se pojem *filtr*, představující neuron konvoluční vrstvy, který je zodpovědný za hledání určitých vzorů v obrázku. Koeficienty v kernelu jsou reprezentovány jednotlivými váhami neuronu. V neuronu se většinou jako aktivační funkce používá ReLU, viz sekce 1.2.3.3.



Obrázek 1.7 Princip konvoluce

Dalším pojmem je *stride*, udávající počet pixelů, o který se kernel posouvá. Při posouvání kernelu se ale může stát, že přeteče za hranici obrázku, kvůli čemuž nelze spočítat hodnotu na kraji. Pro tento problém se zavádí pojem *padding* — ten je buď *valid* (výstup je menší) nebo *same* (přidáváme okraj obrázku s nulami).

Důležitými údaji jsou také rozměry obrázku a počet *kanálů*, ze kterých se skládá. Obrázek s rozměry $m \times n$ ve stupních šedi bude mít jeden kanál, podobně jako na obrázku 1.7. Pokud budeme mít obrázek stejných rozměrů, ale s hodnotami RGB pro každý pixel, bude mít tři kanály. Matematicky se potom obrázky interpretují jako *tensory*[5]. Data na vstupu ale i výstupu mohou mít několik kanálů.

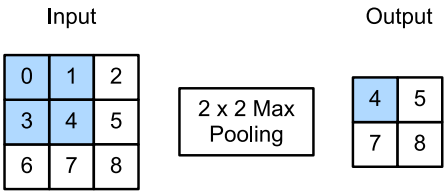
1.6 Pooling vrstva

Pooling vrstvy se v konvolučních sítích vkládají mezi konvoluční vrstvy a slouží ke zmenšení rozměrů dat na vstupu. Tato vrstva, podobně jako konvoluční vrstva, má *okénko*, které se posouvá po obrázku. V rámci okénka se ze vstupních hodnot počítá maximum, což se nazývá jako *max pooling*, nebo se počítá průměr vstupů, což se nazývá jako *average pooling*. Obrázek 1.8, převzat z [12], znázorňuje princip počítání max pooling s okénkem rozměrů 2×2.

1.7 Flatten vrstva

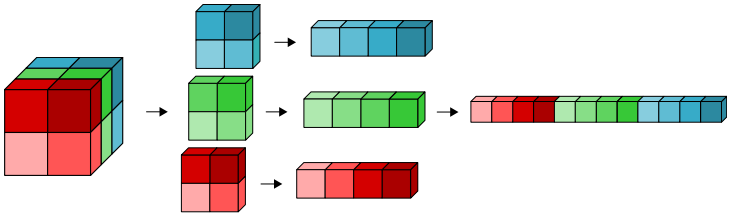
Vrstva flatten se standardně používá v konvoluční síti jako poslední vrstva před sadou dense vrstev a slouží k překonvertování 3D tensoru do 1D tensoru. Tento proces lze vidět na obrázku

[5] jedná se o zobecnění vektorů



Obrázek 1.8 Max pooling s okénkem rozměrů 2×2

1.9 (převzat z [2] a upraven), kde se kanály vstupního tensoru ukládají za sebe do jednoho pole.



Obrázek 1.9 Princip flatten vrstvy

Kapitola 2
Analýza a návrh

V této kapitole je mým cílem nejprve v části 2.1 popsat knihovnu Keras, implementovanou v jazyce Python, kterou využívám pro popis neuronové sítě. Dále v sekci 2.2 popisuji některá existující řešení, generující VHDL z popisů neuronových sítí různých formátů. Poté následuje popis použitých Python knihoven v sekci 2.3 a v závěru kapitoly se v sekci 2.4 věnuji návrhu mého generátoru VHDL.

2.1 Keras API

Keras je API napsané v jazyce Python, které poskytuje prostředky k vysokoúrovňovému popisu hlubokých neuronových sítí a jejich učení. Využívá open-source platformy TensorFlow, napsané v jazycích Python a C++, která se zaměřuje se na oblast strojového učení. Hlavními cíli Kerasu je poskytnout vývojářům prostředky pro experimentování a rychlou implementaci modelů neuronových sítí, včetně jejich učení a souvisejících výpočtů.

V této kapitole se budu zabývat základním použitím funkčností, které Keras nabízí pro formální popis modelu neuronové sítě. Veškeré informace a zdrojové kódy jsou inspirovány Keras dokumentací, viz [13], popřípadě Models API dokumentací, viz [14].

2.1.1 Sekvenční model

Pomocí rozhraní sekvenčního modelu lze popsat nejzákladnější topologii modelů, kde každá vrstva má právě jeden vstupní a výstupní tensor. Slouží k tomu třída `Sequential`, jejíž použití není vhodné, pokud mají vrstvy více vstupních či výstupních tensorů, je potřeba sdílení vrstev nebo je požadovaná jiná než sekvenční topologie. Příklad vytvoření sekvenčního modelu je ve výpisu 2.1. Zavoláním `model.summary()` lze na terminál vytisknout popis topologie modelu s počty parametrů.

2.1.2 Funkční model

Rozhraní funkčního modelu nabízí vytvoření flexibilnějšího modelu, než je model sekvenční. Podporuje totiž více vstupních a výstupních tensorů, nelineární topologii a sdílené vrstvy. Avšak nelze pomocí něj popsat neuronové sítě s cykly, například *rekurzivní*. Příklad vytvoření funkčního modelu je ve výpisu 2.2. Pro přidání vrstvy se nepoužívá metoda `add()`, jako u sekvenčního modelu, ale na instanci třídy `Layer` lze zavolat předchozí vrstvu. Model se vytváří ze třídy `Model`.

```python
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential(name="my_sequential_model") # vraci instanci Model
model.add(layers.Dense(units=2, activation="relu"))
model.add(layers.Dense(units=3, activation="relu"))
model.add(layers.Dense(units=4))
```

Výpis kódu 2.1 Příklad tvorby sekvenčního modelu

```python
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.utils import plot_model

inp = keras.Input(shape=(28, 28, 1), name="img")
x = layers.Conv2D(filters=16, kernel_size=3, activation="relu")(inp)
x = layers.MaxPooling2D(pool_size=3)(x)
x = layers.Conv2D(filters=16, kernel_size=3, activation="relu")(x)
output = layers.GlobalMaxPooling2D()(x)

model = keras.Model(inp, output, name="my_functional_model")
plot_model(model, "schema.png", show_shapes=True) # export topologie do png
```

Výpis kódu 2.2 Příklad tvorby funkčního modelu

2.1.3 Ukládání, načítání a informace o modelu

Instanci třídy `Model` lze uložit zavoláním její metody `save()` s povinným parametrem `filepath` nebo zavoláním funkce `save_model()` s povinnými parametry `model` a `filepath`. Obě funkce mají parametr `save_format`, díky kterému lze specifikovat formát, do kterého se má model uložit. Pro načítání modelu ze souboru slouží funkce `load_model()`, která má povinný argument `filepath`, specifikující cestu k souboru s modelem.

Často je potřeba získat informace o konkrétním modelu. Pro tyto účely má třída `Model` metodu `get_config()`, která vrací *slovník*[1] se všemi parametry. Další možností je zavolat metodu `to_json()`, která navíc uloží data ve slovníku do formátu JSON.

2.1.4 Parametry modelu

Společné rozhraní pro vrstvy nabízí třída `Layer`, ze které všechny ostatní typy vrstev dědí. Pro získání vah lze zavolat `@property weights`, které vrátí pole se všemi váhami a biasy. Protože se v Kerasu se využívá princip znázorněný na obrázku 1.3 v Kapitole 1, znamená to, že toto pole obsahuje dvě `numpy` pole. Jedno z nich je pojmenováno jako *kernel*, obsahující váhy, a druhé jako *bias*, obsahující biasy. V kerasu tedy neexistuje žádná metoda pouze pro biasy.

[1] datová struktura, v Pythonu nazvaná `dict`

2.2 Existující řešení

V této části zmíním některé z existujících řešení, pomocí kterých lze generovat VHDL ze softwarového popisu neuronové sítě. Některé z nich, kromě projektu hls4ml, podporují pouze malou množinu modelů neuronových sítí a frameworků pro tvorbu modelů nebo jsou uživatelsky nepřívětivé. Projekt hls4ml podporuje velké množství modelů, což byl jeden z mých cílů, a proto pro mě byl jediným zdrojem inspirace.

2.2.1 CNN VHDL Generator

Toto řešení [15], vytvořené v rámci diplomové práce, poskytuje rozhraní pro generování VHDL konvolučních neuronových sítí a je distribuováno jako Java aplikace. Výhodou řešení je, že poskytuje grafické rozhraní pro konfiguraci a sestavení modelu sítě. Dále lze načíst konfiguraci z textového souboru.

Nevýhodou je, že jsou podporovány pouze konvoluční sítě. Také zde není podpora frameworků, poskytující rozhraní pro tvorbu modelů neuronových sítí, jako je například Keras. Uživatel je nucen modelovat neuronové sítě přímo v grafickém rozhraní a nebo je popisovat ve speciálním formátu, daným touto aplikací.

2.2.2 NNGen

Toto řešení, viz [16], je prezentováno jako plně přizpůsobitelný HW kompiler pro neuronové sítě, napsaný v jazyce Python. Na rozdíl od předešlého řešení nenabízí grafické rozhraní, ale vše se nastavuje prostřednictím Python souborů. Také nabízí lepší dokumentaci, která je generována pomocí Sphinx. Lze si také stáhnout dockerfile, který ulehčuje instalaci aplikace. Pro popis modelu lze použít framework ONNX.

Nevýhodou je ne příliš uživatelsky přívětivé rozhraní. V [16] jsou příklady použití, které jsou velmi rozsáhlé a nepřehledné — proces generování není příliš automatizovaný, protože se musí VHDL návrh popisovat pomocí jazyka Python, což značí, že návrh HW není předem definovaný pomocí šablon. Také není moc patrné, co vše tento projekt doopravdy poskytuje.

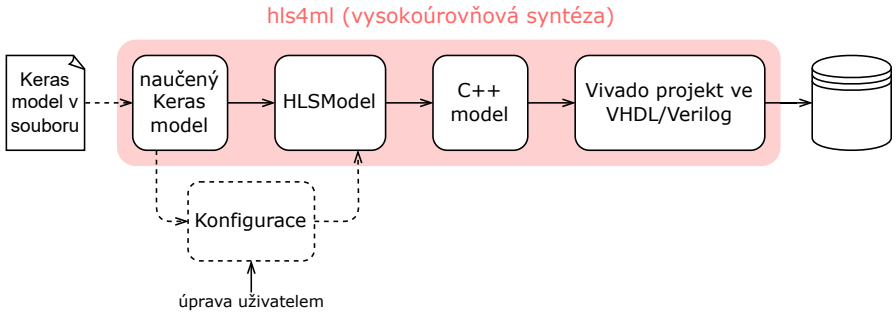
2.2.3 High Level Synthesis for Machine Learning

High Level Synthesis for Machine Learning, neboli *hls4ml* [17], je open-source projekt, který má za cíl poskytnout vývojářům rozhraní pro generování syntetizovatelného popisu hardwaru ze softwarového modelu neuronové sítě vytvořeného ve vyšších programovacích jazycích, zejména v Pythonu. Model sítě lze popsat v Kerasu (viz sekce 2.1), ONNX či v PyTorch. V tomto textu se dále budu zabývat pouze syntézou z Keras modelu.

Princip fungování platformy hls4ml lze vidět na obrázku 2.1. Nejprve je potřeba navrhnout a popsat model v Kerasu — může být načten ze souboru nebo vytvořen v rámci programu. Dále je pomocí hls4ml překonvertován do třídy `HLSModel`. Parametry tohoto modelu lze upravovat, viz sekce 2.2.3.1. `HLSModel` je v další fázi překonvertován do modelu v jazyce C++, uzpůsobeného tak, aby byl kompatibilní s platformou Vitis HLS, kde je možnost ho upravit a i simulovat. Následuje vygenerování projektu pro simulátor Vivado, obsahující přeložený popis hardwaru v jazyce VHDL nebo Verilog. Ve Vivadu je také možné vygenerovaný návrh upravovat a simulovat.

Proces od zpracování Keras modelu až do získání hardwarového popisu je *vysokoúrovňová syntéza*, která se vyznačuje vygenerováním popisu hardware z vyšších programovacích jazyků a je výrazně složitější než *běžná syntéza*, která se provádí z HDL[2] jazyků do bitstreamu. Zde nastává

[2]Hardware Description Language



Obrázek 2.1 Princip platformy hls4ml

problém — popis hardwaru by měl mít čitelnou strukturu, protože v něm nezáleží na pořadí příkazů, narozdíl od běžných programovacích jazyků. Protože je tento popis generován přes dvě úrovně abstrakce, tedy z Pythonu a poté z C++, není již naplno dodržena přehlednost a popis hardwaru není pro člověka příliš čitelný. Kvůli zmíněné abstrakci je těžké si pod jednotlivými částmi popisu představit bloky hardwaru, který popisují, což znemožňuje upravování popisu a hlavně hledání chyb v něm.

2.2.3.1 Konfigurace Keras modelu

Ještě než se spustí vysokoúrovňová syntéza z třídy `HLSModel` do jazyka C++, lze pomocí hls4ml vygenerovat slovník s konfigurací celého modelu, obsahující například informace o datových typech všech parametrů sítě a plno dalších. Tuto konfiguraci lze upravit dle vlastních požadavků a přiložit jako parametr při tvorbě třídy `HLSModel`.

Ve výpisu 2.3 lze vidět vytvoření konfigurace modelu z Kerasu. Mezi nejužitečnější argumenty patří povinný argument `model`, vyžadující instanci třídy `Model` z Kerasu, volitelný argument `granularity`, specifikující podrobnost konfigurace pomocí hodnot

- `"model"`, určující konfiguraci nad celým modelem,
- `"type"`, určující konfiguraci nad stejnými typy vrstev,
- `"name"`, určující konfiguraci nad jednotlivými vrstvami,

a volitelný argument `default_precision`, nastavující výchozí datový typ pro parametry sítě. Vytvořená konfigurace má slovníkovou strukturu, tedy lze k hodnotám přistupovat přes jejich klíče.

```
1 from hls4ml.utils import config_from_keras_model
2
3 model_config = config_from_keras_model(
4         keras_model,                # model z Kerasu
5         granularity="name",         # podrobnost konfigurace
6         default_precision="float") # datove typy
```

Výpis kódu 2.3 Příklad vytvoření konfigurace modelu

Uvnitř funkce `config_from_keras_model()` se děje několik věcí. Nejdříve se načtou informace o modelu (viz 2.1.3), na základě kterých se konfigurace vytváří. Dále se zjišťuje, zda je model z Kerasu *sekvenční* nebo *funkční* — to je důležité pro parsování. Následuje větvení, které vytvoří příslušnou konfiguraci podle toho, jakou hodnotu má argument `granularity`. Pokud

je jeho hodnota `"type"` či `"model"`, tak se volá funkce `make_layer_config()` s argumentem `layer`, který je jednou z hodnot slovníku popsaného v sekci 2.1.3. V této funkci se znovu větví, tentokrát podle typu vrstvy a nastavuje se výchozí konfigurace jejíž parametrů. Na konci se vrací vytvořená konfigurace modelu v podobě slovníku.

Dle mého názoru je zde plno příležitostí použít techniky objektově orientovaného programování, protože není kód příliš čitelný a těžce by se rozšiřoval. Šlo by například vytvořit třídy pro různé typy konfigurací a hlavně třídy pro vrstvy, jejichž konfigurace by se poté dala vytvářet polymorfně.

2.2.3.2 Vytvoření a syntéza HLS modelu

Konfiguraci ze sekce 2.2.3.1 lze nyní uplatnit při vytvoření instance třídy `HLSModel`, což je uvedeno ve výpisu 2.4 na řádcích 1-5. Stejně jako u konfigurace je zde argument `model` a dále volitelný argument `hls_config`, kterému se může předat buď vytvořená konfigurace a nebo žádná — vytvoří se výchozí konfigurace. Pro specifikaci FPGA slouží argumenty `board` a `part`. Na řádku 7 lze vidět uložení modelu a na řádku 8 spuštění syntézy do HDL včetně zabalení do projektu pro Vivado.

```
1 from hls4ml.converters import convert_from_keras_model
2
3 hls_model = convert_from_keras_model(
4         keras_model,             # model z Kerasu
5         hls_config=model_config) # konfigurace modelu
6
7 hls_model.write() # ulozeni HLS modelu
8 hls_model.build(export=True) # synteza a ulozeni Vivado projektu
```

Výpis kódu 2.4 Příklad vytvoření a syntézy HLS modelu

Ve funkci `convert_from_keras_model()` se nejdříve validuje přiložená konfigurace a pokud není žádná, tak se vytváří konfigurace s granularitou `"model"`. Do konfigurace se pod klíč `"KerasModel"` uloží instance modelu z Kerasu. V poslední řadě se volá funkce `keras_to_hls()` s argumentem konfigurace. V této funkci se načítá[3] a validuje model z Kerasu, aby se zjistilo, zda jsou platformou hls4ml podporované všechny vrstvy v něm. Pokud je vrstva podporovaná, tak jsou jí nastaveny všechny parametry dle konfigurace. Dále se model parsuje a upravuje dle specifik hls4ml. Na konci se vytvoří instance třídy `HLSModel`, předají se jí potřebné parametry a vrátí se. Ani zde se nepoužívají techniky objektového programování.

▶ Poznámka 2.1 (Aktivační vrstvy v HLS modelu). Je důležité podotknout, že HLS model může v procesu parsování vytvářet takzvané *aktivační vrstvy*. Ty se vytvářejí za vrstvami z Kerasu, které mají aktivační funkci. V hls4ml jsou tyto vrstvy potřeba kvůli specifické hardwarové implementaci.

Metodami `write()` a `build()` se v tomto textu zabývat dále nebudu, jelikož nejsou důležité pro moje řešení a jsou odlišné od způsobu, jakým generuji popis hardwaru.

2.3 Python knihovny

V následujících sekcích budu popisovat použité knihovny. Nejsou zde ale uvedeny všechny použité knihovny — standardně ty, z kterých jsem použil 1-2 funkce, které jsem stručně popsal přímo v místě použití. Následující knihovny ale potřebují více vysvětlení.

[3] pokud je model předán jako řetězec, tak se načítá ze souboru, viz sekce 2.1.3

2.3.1 abc

Pro podporu abstraktního rozhraní existuje v jazyce Python knihovna `abc`, viz [18]. Nutno podotknout, že tato knihovna ve starších verzích Pythonu neexistovala, proto doporučuji nahlédnout do příloh, kde je zmíněná použitá verze Pythonu.

Aby se třída stala abstraktní, je potřeba, aby dědila z třídy `abc.ABC`. Dále je potřeba nové třídě definovat abstraktní rozhraní. To lze popsat pomocí následujících *dekorátorů*[4]:

- `@abc.abstractmethod` — označuje abstraktní metodu,
- `@abc.abstractclassmethod` — označuje abstraktní třídní metodu,
- `@abc.abstractstaticmethod` — označuje abstraktní statickou metodu,
- `@abc.abstractproperty` — označuje abstraktní `@property`.

Je také možnost všechny zmíněné dekorátory popsat pouze pomocí dekorátoru pro abstraktní metodu — přidat před něj dekorátor pro třídní metodu (`@classmethod`), statickou metodu (`@staticmethod`) či property (`@property`).

2.3.2 re

Pro podporu regulárních výrazů v jazyce Python slouží knihovna `re`, viz [19]. Tuto knihovnu jsem použil pro parsování veškerých VHDL souborů. Výhodou je, že lze používat syntaxi regulárních výrazů definovanou ve standardu POSIX.

Pro vytvoření regulárního výrazu slouží funkce `re.compile(regex)`, vracející objekt `Pattern`, který zapouzdřuje regulární výraz `regex`. Řetězec `regex` je doporučeno uvádět ve tvaru *raw*, který lze vytvořit zápisem `r'retezec'`. Bez tohoto tvaru by se muselo každé zpětné lomítko escapovat pomocí dalšího zpětného lomítka. Dále tato metoda nabízí volitelný argument `flags`, pomocí kterého lze specifikovat chování regulárního výrazu. Pokud je flagů více, stačí je spojit operací `or`. Využil jsem zejména tyto flagy:

- `re.IGNORECASE` — ignoruje velikost písma,
- `re.MULTILINE` — upřesňuje chování speciálních výrazů `"^"` a `"$"` tak, že fungují na jednotlivé řetězce a nezachytávají pouze začátek řetězce a konec řetězce za posledním výskytem znaku `"\n"`, ale za každým výskytem,
- `re.DOTALL` — upřesňuje chování speciálního výrazu `"."` tak, že zachytává úplně všechny znaky, včetně `"\n"`,
- `re.ASCII` — upřesňuje chování speciálních výrazů, pracující se slovy, tak, že zachytávají pouze ASCII znaky.

Pro práci s objektem `Pattern` jsem použil následující metody:

- `search(string)` — najde první výskyt regulárního výrazu v řetězci `string` a vrátí `Match` objekt v případě úspěchu, jinak hodnotu `None`,
- `finditer(string)` — vrací iterátor se všemi shodami v podobě objektů `Match`,
- `sub(replace, string)` — vrací řetězec `string`, ve kterém jsou všechny shody s regulárním výrazem nahrazeny za řetězec `replace`.

[4] jedná se o Python funkcionalitu, pomocí které lze funkci dodat nějaké vlastnosti bez jejího předefinování

Dále je v knihovně definována výše zmíněná třída `Match`, reprezentující shodu nějakého řetězce s regulárním výrazem. Využil jsem její metodu `group(group)`, vracející podvýraz, který je součástí shody. Argumentem `group` lze říct, jaký podvýraz se má vrátit. Může to být buď číslo (0 udává celý výraz, 1 udává první podvýraz, 2 druhý podvýraz a tak dále) nebo řetězec s identifikátorem podvýrazu. V obou případech je nutné podvýrazy v regulárním výrazu označit následujícími způsoby:

- v případě, kdy je argument `group` číslo, se předpokládá, že podvýrazy v regulárním výrazu se nachází v kulatých závorkách, například `r'neco1(podvyraz)neco2'`
- v případě, kdy je argument `group` řetězec, se předpokládá, že podvýrazy v regulárním výrazu jsou označeny ve tvaru `r'neco1(?P<id_podvyrazu>podvyraz)neco2'`.

V případě, kdy by bylo potřeba podvýraz označit závorkami, ale nebylo by požadováno, aby se dal získat pomocí metody `group()`, tak ho lze označit jako `r'neco1(?:podvyraz)neco2'`.

2.3.3 cppyy

Tato knihovna poskytuje rozhraní pro volání C++ kódu přímo v Pythonu. Nejedná se o built-in knihovnu, ale o knihovnu třetí strany. Její dokumentaci lze najít na [20]. Knihovnu používám pro volání tříd specializovaných datových typů z C++ knihovny `laflib`, jejíž autorem je vedoucí mé práce, pan Ing. Miroslav Skrbek, Ph.D.

Předpokladem pro fungování je fungující C++ projekt, který je zkompilovaný a slinkovaný do sdílené knihovny s příponou `.so`. Po importování knihovny příkazem `import cppyy` lze volat funkci `cppyy.include(header)`, kde `header` je název hlavičkového souboru, který chceme importovat. Dále je potřeba zavolat funkci `cppyy.load_library(name)`, kde `name` je název sdílené knihovny. Po těchto krocích už lze volat C++ metody ve tvaru `cppyy.gbl.metoda()`.

2.4 Návrh řešení

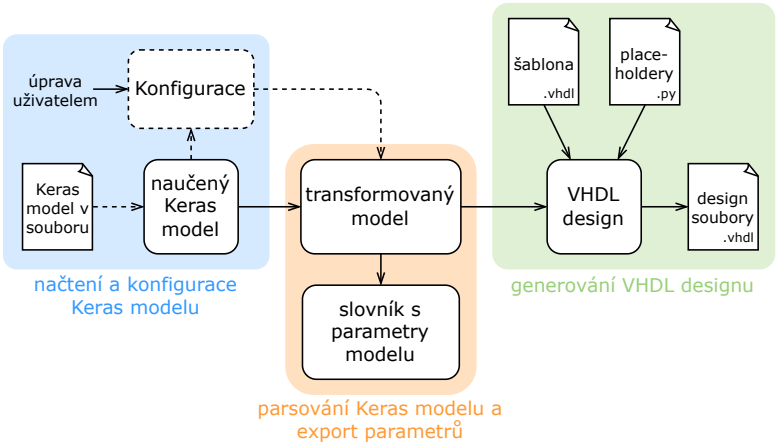
Rozdíl mého generátoru a řešení hls4ml spočívá v tom, že generuji VHDL přímo z naučeného Keras modelu. Tím lze docílit toho, že návrh ve VHDL je dobře čitelný (oproti hls4ml), protože vychází z designů, které napsal člověk a celkový proces generování probíhá sekvenčně tak, jak by ho dělal člověk.

Architekturu mého generátoru, znázorňující schéma na obrázku 2.2, lze rozdělit do tří částí. V první části se z naučeného Keras modelu, který lze stejně jako v hls4ml načíst ze souboru, vytváří nepovinná *konfigurace*, díky které lze konfigurovat parametry modelu. Ve druhé části vzniká z Keras modelu a případné konfigurace *transformovaný model*, ze kterého lze exportovat slovník s parametry sítě, překonvertovaných do specializovaných datových typů, daných konfigurací. V poslední části se z transformovaného modelu vytváří *VHDL design*, vycházející ze speciálních VHDL souborů — VHDL šablon — které nejsou kompletní a části, které se do nich doplňují v rámci generování, jsou reprezentovány jednoduchými komentáři v daném formátu. Tyto komentáře jsou uloženy jako konstantní proměnné definované v Python souboru se *zástupnými symboly* (dále placeholdery).

V následujících řádcích se budu podrobněji věnovat principům všech tří částí výše nastíněné architektury, zejména z hlediska uživatele, aby byla vybudována dostatečná představa o fungování generátoru a také bylo snažší pochopit implementaci popsanou v Kapitole 3.

2.4.1 Konfigurace Keras modelu

Podobně jako v hls4ml, v mém generátoru lze ještě před transformováním modelu vytvořit konfiguraci, díky které se dají upravovat parametry modelu, zejména datové typy vah či biasů. Dále jsem se z hls4ml inspiroval myšlenkou *granularity* konfigurace (viz sekce 2.2.3.1) a vytvořil třídy



Obrázek 2.2 Architektura mého generátoru VHDL

1. `ModelConfig`, určující konfiguraci nad celým modelem,
2. `LayerTypeConfig`, určující konfiguraci nad stejnými typy vrstev,
3. `LayerNameConfig`, určující konfiguraci nad jednotlivými vrstvami,

které mají konfigurační data uložené ve slovníku. V hls4ml je možnost konfiguraci upravovat za běhu programu pouze prostřednictvím slovníku, ve kterém jsou položky uložené. Tuto možnost jsem zanechal a data ve slovníku všech tří tříd zpřehlednil, ale přidal jsem možnost upravit konfiguraci pomocí vyskakovacího okna, kde může uživatel přímo upravit jednotlivé položky podobně, jako by je měl otevřené v textovém editoru. Konfigurace ve vyskakovacím okně je ve formátu YAML, který jsem zvolil z důvodu dobré čitelnosti a také proto, že jej lze jednoduše získat z formátu JSON, který lze získat právě z Python slovníku. Diagram celého procesu je vidět na obrázku 2.3.

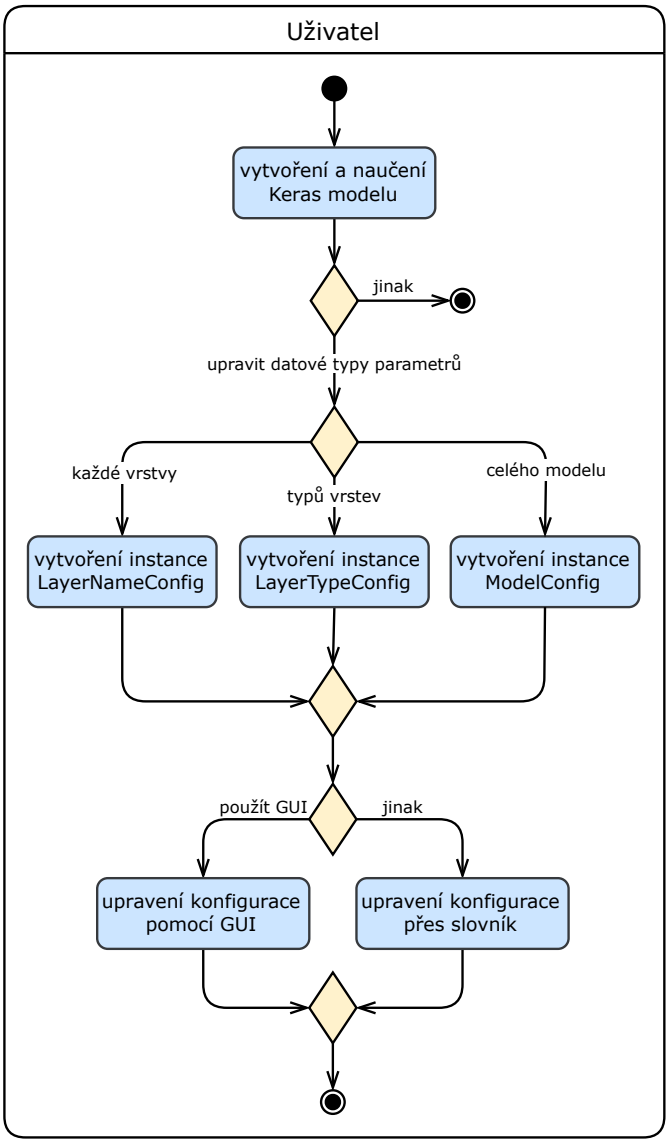
2.4.2 Transformace Keras modelu a export parametrů

V dalším kroku je potřeba naučený Keras model parsovat a tato data uložit do vhodné struktury — k tomu jsem navrhl třídu `Model`. Této třídě musí uživatel předat naučený Keras model a případně jednu z konfigurací popsaných v předešlé sekci. Pokud uživatel nepředá žádnou konfiguraci, bude při parsování vytvořena konfigurace `ModelConfig` s výchozím datovým typem pro parametry sítě. V procesu parsování se vytvářejí třídy pro každou vrstvu, naplňují se potřebnými informacemi a aplikuje se na ně konfigurace.

Po parsování lze na třídu `Model` zavolat metodu, která uloží všechny parametry sítě do slovníku, který může uživatel exportovat do potřebného formátu. Všechny parametry jsou převedeny do datových typů, které jsou specifikovány vytvořenou konfigurací či výchozí konfigurací pro celý model s výchozími datovými typy. Celý proces znázorňuje diagram na obrázku 2.4.

2.4.3 Generování VHDL designu

Vytvoření konkrétního VHDL designu zajišťuje hierarchie tříd vycházející z abstraktní třídy `VHDLDesign`. Na nejvyšší úrovni s touto třídou ale uživatel neinteraguje — k tomu slouží potomci třídy `VHDLTemplateDesign`, která dědí z `VHDLDesign`. Pro jednoduchost bude v této kapitole každý potomek třídy `VHDLTemplateDesign`, s kterým uživatel interaguje, nazýván pouze jako *design* a detailní vysvětlení celé hierarchie bude popsáno v Kapitole 3.

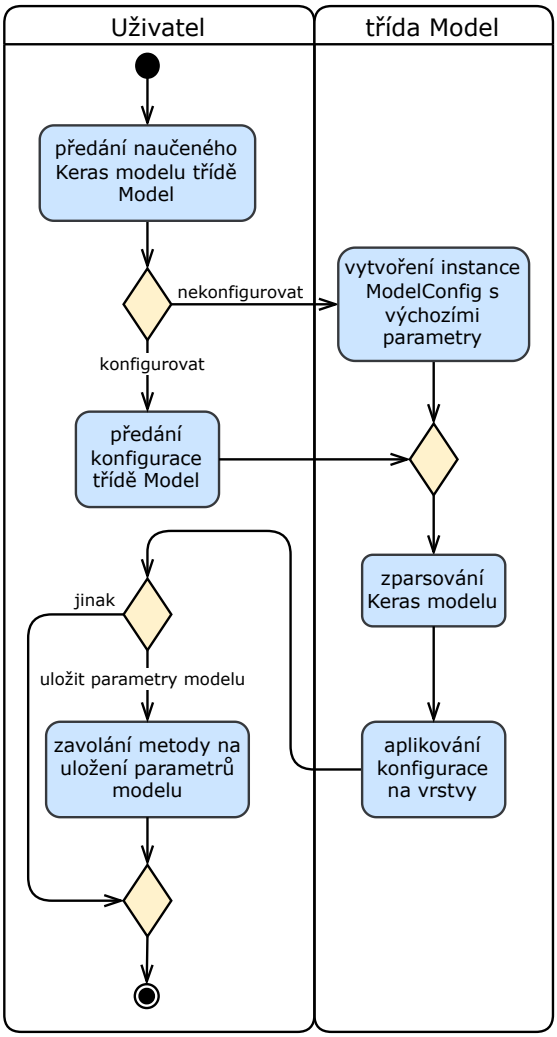


Obrázek 2.3 Princip konfigurace Keras modelu

Každý design je svázán s VHDL šablonou (viz obrázek 2.5), ze které vychází a doplňuje do ní části, které vyplývají z topologie modelu a je potřeba je přizpůsobit. Často platí, že neměnnou částí je entita, která je v šabloně již definovaná, ale i tak jí lze upravovat. Předem dané části může obsahovat i architektura — například kód v deklarační části, ale i kód za klíčovým slovem *begin*. Zbytek architektury je ale reprezentován komentáři ve speciálním formátu, které jsou svázány s částmi designu, které se za ně mají nahrazovat.

Kromě VHDL šablony je ještě design svázán s Python souborem, obsahující placeholdery, viz obrázek 2.5. Jedná se o řetězce výše zmíněných VHDL komentářů, za které se substituují měnící se části designu na základě topologie modelu. Řetězce jsou uloženy v konstantních proměnných, aby se v kódu nemusely jednotlivě kopírovat. Detailnějšímu popisu šablon i placeholderů se zabývám v Kapitole 3.

Pro vytvoření konkrétního designu je nejprve potřeba zkonstruovat jeho instanci. Ta načte ze své VHDL šablony knihovny, entitu a architekturu, naparsuje je a uloží do vnitřních datových



Obrázek 2.4 Princip vytvoření transformovaného modelu a jeho parametrů

VHDL šablona

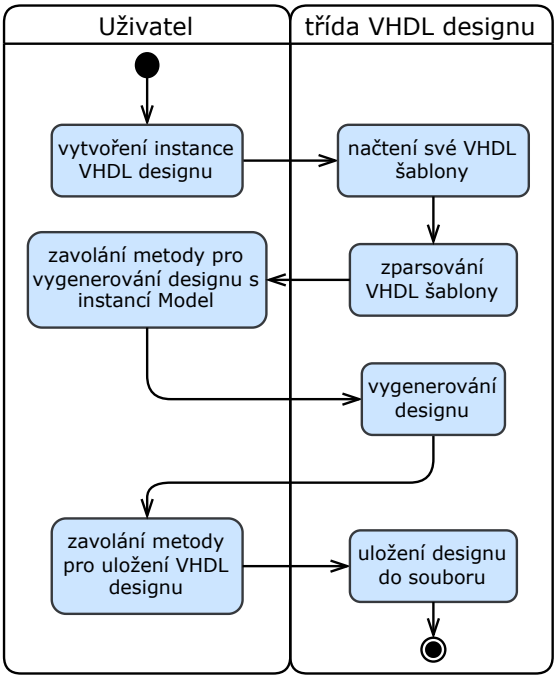
```
if (rising_edge(clk)) then
-- <PLACEHOLDER1>
status <= (others => '0');
ack     <= '0';
done    <= '0';
-- <PLACEHOLDER2>
end if;
```

Python soubor s placeholdery

```
PLACEHOLDER1 = "-- <PLACEHOLDER1>"
PLACEHOLDER2 = "-- <PLACEHOLDER2>"
```

Obrázek 2.5 Příklad VHDL šablony a Python souboru s placeholdery

struktur. Dále je potřeba její metodě předat instanci třídy `Model` ze sekce 2.4.2, čímž začne proces generování. V závěru stačí zavolat metodu pro uložení designu do souboru, který ponese stejné jméno, jako má jeho entita. Celý proces znázorňuje diagram na obrázku 2.6.



Obrázek 2.6 Princip vytvoření VHDL designu

Kapitola 3
Implementace

V této kapitole se zabývám implementací mého řešení, jehož návrh byl popsán v sekci 2.4. Nejprve se v sekci 3.1 zabývám rozhraním vrstev, poté v sekci 3.2 rozhraním konfigurace a v sekci 3.3 rozhraním transformovaného modelu. V závěru kapitoly se v sekci 3.4 věnuji generování VHDL. Ve všech zmíněných částech se snažím uplatňovat techniky objektově orientovaného návrhu, čímž lze docílit jednoduchou rozšiřitelnost a podpořit přehlednost kódu.

Celý projekt je zabalen do balíčku `nntovhdl`, jehož strukturu lze vidět na obrázku 3.1. Pro každý stavební blok generátoru jsem vytvořil další balíčky, ve kterých se nachází zdrojové soubory s třídami. Každá ze tříd má vlastní soubor, takže názvy souboru z velké většiny nesou název třídy, která je uvnitř. Soubory, které nemají vlastní balíček, jsou:

- `nntovhdl/constants.py`, obsahující konstanty s cestami k VHDL šablonám, designům a balíčkům,
- `nntovhdl/laf.py`, ve kterém se volají funkce z knihovny `cppyy` pro importování hlavičkových souborů C++ knihovny `laflib` a tříd specializovaných datových typů z ní,
- `nntovhdl/model.py`, obsahující implementaci třídy `Model`,
- `nntovhdl/regexps.py`, obsahující konstanty a funkce, zapouzdřující regulární výrazy pro parsování VHDL souborů,
- `nntovhdl/utils.py`, obsahující pomocné funkce,
- `nntovhdl/vhdl_file.py`, obsahující implementaci třídy `VHDLFile`.

Detailnější popisy jednotlivých balíčků lze nalézt v přiložené dokumentaci. Za zmínku stojí adresář `nntovhdl/sources`. v něm se nachází dva adresáře:

- `cpp`, obsahující zdrojové kódy knihovny `laflib` včetně souboru utility CMake, pomocí které lze vygenerovat Makefile,
- `vhdl`, obsahující podadresáře `designs`, `packages` a `templates`, viz dále.

Zmíněné podadresáře v adresáři `vhdl` mají následující význam:

- `designs` — obsahuje hotové VHDL designy, které neobsahují placeholdery a berou se pouze jako hotové komponenty,
- `packages` — obsahuje hotové VHDL balíčky (packages), které se mohou používat při generování, ale balíčky samy o sobě generované nejsou,
- `templates` — obsahuje VHDL soubory, které se berou jako šablony, protože jsou v nich placeholdery.

```
__init__.py ..... konstruktor balíčku nntovhdl
configs ..... balíček s třídami konfigurace
managers ..... balíček s managery
layers ..... balíček s třídami vrstev
    __init__.py ..... konstruktor balíčku
    layer.py ..... implementace třídy Layer
    conv_layers ..... balíček s konvolučními vrstvami
    core_layers ..... balíček se základními vrstvami
    dense_layers ..... balíček s dense vrstvami
    pooling_layers ..... balíček s pooling vrstvami
vhdl_designs ..... balíček s třídami zapouzdřující VHDL designy
constants.py ..... konstanty
laf.py ..... import C++ metod z knihovny laflib do Pythonu
model.py ..... implementace třídy Model
regexps.py ..... regulární výrazy pro parsování VHDL souborů
utils.py ..... pomocné funkce
vhdl_file.py ..... implementace třídy VHDLFile
sources ..... zdrojové kódy v C++ a VHDL
    cpp ..... zdrojové kódy C++ knihovny laflib
        include ..... hlavičkové soubory
        src ..... implementační soubory
        CMakeLists.txt ..... soubor utility CMake pro generování Makefile
    vhdl ..... VHDL zdrojové kódy
        designs ..... VHDL designy
        packages ..... VHDL packages
        templates ..... VHDL šablony
```

Obrázek 3.1 Struktura mého projektu `nntovhdl`

3.1 Rozhraní vrstev

V sekcích 2.2.3.1 a 2.2.3.2 jsem zmiňoval, že je plno příležitostí použít objektový přístup včetně polymorfismu, který by napomohl čitelnosti a rozšiřitelnosti a eliminoval větvení dle typů vrstev. Proto jsem se pro vrstvy rozhodl vytvořit hierarchii tříd se společným rozhraním, definovaným abstraktní třídou `Layer`, ze které dědí ostatní typy vrstev a každá z nich do rozhraní přidává to, co vyžaduje. Příklad hierarchie tříd znázorňuje obrázek 3.2, kde jsem z důvodu přehlednosti uvedl jen některé třídy (kurzíva v názvu značí abstraktní třídu).

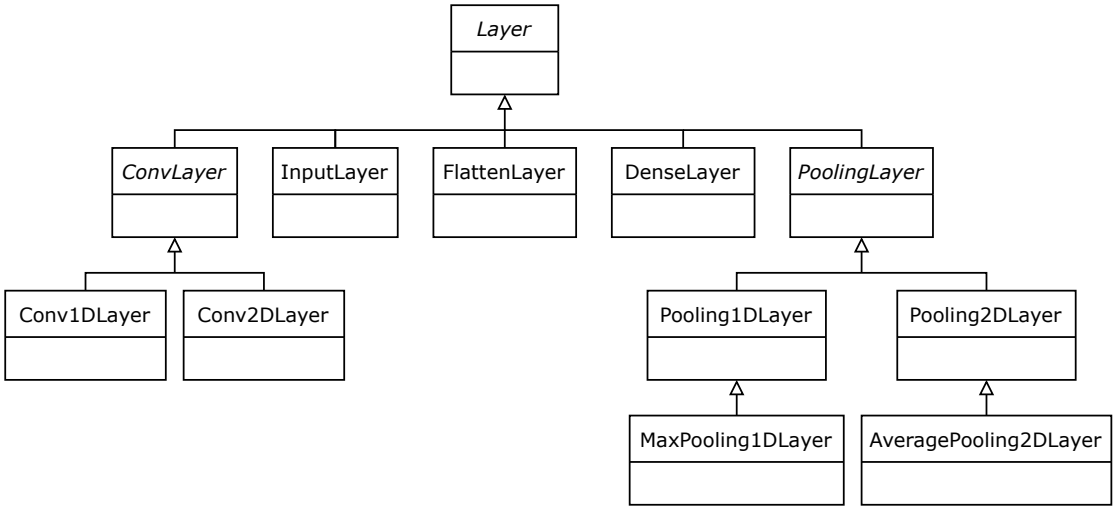
3.1.1 Třída Layer

Konstruktor všech tříd implementuje základní abstraktní třída `Layer`, který přijímá:

- povinný argument `keras_layer_dict: dict` — slovník s informacemi o vrstvě z Kerasu,
- povinný argument `keras_layer: keras.Layer` — instance vrstvy z Kerasu.

Tyto argumenty se ukládají do třídy včetně dalších jednoduchých třídních proměnných — `name`, pole `prev_layers` a `next_layers`, do kterých se ukládají instance třídy `Layer`, a aktivační funkce `activation`, pro jejíž reprezentaci postačuje řetězec. Za zmínku stojí zejména uložení vah do `weights` a biasů do `biases`, které je potřeba oddělit (více v sekci 2.1.4). Pokud jsou použity datové typy z knihovny `laflib`[1], tak se navíc počítají *koeficienty škálování* a ukládají se do `scales`, kterými se v závěru škálují váhy a biasy.

[1] C++ knihovna se specializovanými datovými typy; autorem je pan Ing. Miroslav Skrbek, Ph.D.



Obrázek 3.2 Diagram tříd vrstev

Kromě konstruktoru definuje třída `Layer` abstraktní metody:

- `parse() -> None`, která parsuje vrstvu z Kerasu a vytváří další potřebné třídní proměnné,
- `output_shape() -> list`, vracející pole, které specifikuje tvar výstupu z vrstvy[2].

Tyto metody je potřeba implementovat v případě přidání třídy pro novou vrstvu. Jejich implementace je specifická pro každou zděděnou vrstvu, které zde všechny popisovat nebudu, a proto si dovolím čtenáře odkázat do kódu či vygenerované dokumentace. Dále zde uvedu některé z nejdůležitějších metod, které tato třída implementuje:

- `make_config(layer: dict, precision) -> dict`, která je statická, vrací slovník s konfigurací každé vrstvy a používá se v metodě `create_config()` třídy `Config`,
- `apply_config(config: Config) -> None`, aplikující konfiguraci na vrstvu a používaná v metodě `create_layers()` třídy `Model`,
- `scale_parameters() -> None`, která počítá koeficienty škálování a aplikuje je na váhy a biasy,
- `is_laflib_type(type) -> bool`, zjišťující, zda `type` je z knihovny `laflib`,
- `neurons_count() -> int`, vracející počet neuronů ve vrstvě,
- `inputs_count() -> int`, vracející počet vstupů do vrstvy.

▶ Poznámka 3.1 (Předefinování metod). Výše popsané metody je možné ve zděděných třídách předefinovat (některé třídy to už dělají) a pokud by bylo potřeba měnit složení argumentů, doporučuji je změnit u všech definicí upravované metody — jazyk Python by jiné složení argumentů dovolil, ale narušilo by to přehlednost objektového návrhu.

Z výše popsaných metod jsou `scale_parameters()`, `is_laflib_type(type)` a `inputs_count()` jediné, které nejsou předefinované ve zděděných třídách. Metoda pro počítání koeficientů škálování se počítá ve vrstvách, které s tím souvisí. Počítají se pouze pokud jsou ve vrstvě použity datové typy z knihovny `laflib`.

[2] pořadí hodnot určuje třídní proměnná `data_format`

Ve výpisu 3.1 uvedená implementace metody počítající počet vstupů. Nejdříve se pomocnou metodou `get_weights_shape()` získá `numpy` pole s vahami[3], které se uloží do proměnné `tmp`. Může se stát, že vrstva nemá váhy, což znamená, že počet vstupů je dán počtem neuronů, který lze spočítat pomocí metody `neurons_count()`. V opačném případě se spočítají shapes pomocí metody `np.prod()`.

```
import numpy as np
def inputs_count ( self ) -> int:
    tmp = self.get_weights_shape("kernel") # "kernel" gets weights
    if not tmp:  # No weights
        return self.neurons_count()
    w_shapes = [ 1 if s is None else s for s in tmp ]
    return np.prod(w_shapes)
```

Výpis kódu 3.1 Implementace metody `inputs_count()` třídy `Layer`

V metodě `neurons_count()` se iteruje přes výstupní tvary získané metodou `output_shape()` a postupně se mezi sebou násobí. Metoda není nijak zajímavá, proto jí zde neuvedu a v případě zájmu lze nahlédnout do kódu či přiložené dokumentace.

3.1.2 Třída LayerManager

Polymorfní přístup podporuje i třída `LayerManager`, která zapouzdřuje slovník s objekty všech potomků třídy `Layer`. Konkrétní třídu potomka lze vyzvednout pomocí metody `get_layer(layer_name: str)`, kde `layer_name` je název třídy vrstvy z Kerasu.

Pro fungování slouží instance `layer_manager` z výpisu 3.2, kde jsem uvedl jen pár základních tříd. Pro lepší představu je možné nahlédnout do implementací metod třídy `Config` (výpisy 3.5 a 3.7) či `Model` (výpis 3.10). Zmíněná instance je automaticky vytvořena při jejím importování z balíčku `managers`. Při přidání nové vrstvy je potřeba do instance přidat třídu této vrstvy včetně klíče, odpovídající Keras vrstvě, na kterou se má nová třída mapovat.

```
import nntovhdl.layers as layers
layer_manager = LayerManager (
    {
        "Dense": layers.dense_layers.DenseLayer,
        "Conv2D": layers.conv_layers.Conv2DLayer,
        "MaxPooling2D": layers.pooling_layers.MaxPooling2DLayer,
    }
)
```

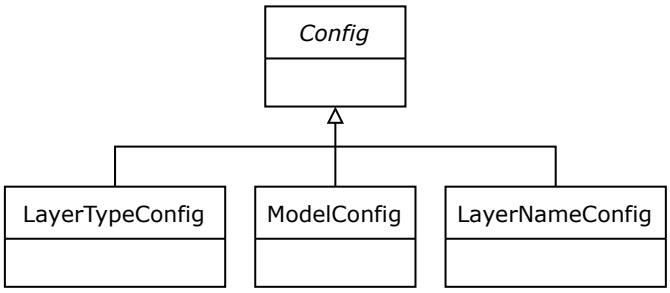
Výpis kódu 3.2 Vytvoření instance třídy `LayerManager`

3.2 Rozhraní konfigurace

V projektu hls4ml jsou konfigurace reprezentovány pouze slovníkovou strukturou a pro každý typ konfigurace se v kódu musí dělat zvlášť větvení. Moje snaha byla k tomuto přistoupit objektově,

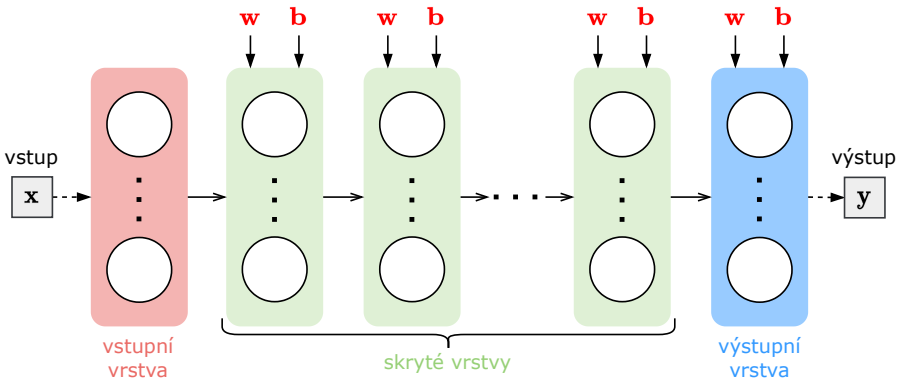
[3] detaily o vahách v Kerasu jsem sepsal v části 2.1.4

a proto jsem zapouzdřil slovník s konfigurací do abstraktní třídy `Config`, definující základní rozhraní všech typů konfigurací, tedy `ModelConfig`, `LayerTypeConfig` a `LayerNameConfig`. Hierarchii dědění a rozhraní těchto tříd lze vidět na obrázku 3.3.



Obrázek 3.3 Diagram tříd konfigurace

Pomocí konfigurace lze konfigurovat datové typy vah a biasů, ale pouze nad jednotlivými vrstvami, což znázorňuje obrázek 3.4 s obecnou strukturou modelu.



Obrázek 3.4 Konfigurace datových typů v modelu

3.2.1 Třída Config

Konstruktor, společný pro všechny typy konfigurací, definuje třída `Config`. Předává se do něj:

- povinný argument `keras_model` — třída Model z Kerasu,
- nepovinný argument `precision` — třída s datovým typem výchozím pro konfiguraci (například třída `float`),
- nepovinný argument `gui: bool` — udávající, zda se má pro upravení konfigurace použít grafické rozhraní (více v sekci 3.2.5).

▶ Poznámka 3.2. Datovými typy v argumentu `precision` mohou být i specializované typy z C++ knihovny `laflib`. Datový typ `lv20` by se například volal jako `nntovhdl.lv20`, což je možné díky souboru `nntovhdl/laf.py`, který je importován v konstruktoru balíčku `nntovhdl`.

V konstruktoru se nejdříve iniciují třídní proměnné `keras_model`, `default_precision` a `gui` hodnotami, které jsou předány skrz argumenty. Dále se Keras model převede do slovníkové struktury pomocí funkce ve výpisu 3.3, aby se z něj dobře získávaly informace. Používá se k tomu

funkce `loads()` z knihovny `json` v kombinaci s metodou `to_json()` pro Keras model, případně se používá metoda `load_model()`, pokud je model uložen v souboru (obě varianty jsou popsány v sekci 2.1.3). Protože se umístění vrstev v tomto slovníku liší podle toho, zda je model z Kerasu sekvenční nebo funkční, tak se musí větvit dle těchto dvou typů modelu, aby se do třídní proměnné `layers_dict` správně uložil slovník s vrstvami. Na konci konstruktoru se už jen zavolá metoda `create_config()`, kterou sice třída `Config` neimplementuje, ale knihovna `abc` (viz sekce 2.3.1) umožňuje abstraktní metodu volat, pokud existuje alespoň jedna třída, která jí implementuje[4], jinak by knihovna vyvolala chybovou hlášku.

```
import json
import tensorflow as tf
def keras_model_to_dict ( keras_model ) -> dict:
    if isinstance(keras_model, dict):
        return keras_model
    if isinstance(keras_model, str):  # From file
        return json.loads(tf.keras.models.load_model(keras_model).to_json())
    return json.loads(keras_model.to_json())
```

Výpis kódu 3.3 Funkce `keras_model_to_dict()` převádějící Keras model do slovníku

Třída `Config` dále definuje abstraktní metody:

- `create_config() -> None`, volaná z konstruktoru a sloužící k vytvoření výchozí konfigurace, kterou ukládá do slovníku ve třídní proměnné `data`,
- `get_values(layer: Layer, keys: list[str]) -> tuple`, sloužící k získání dat konfigurace vrstvy `layer` ze slovníku `data` pod klíči `keys`.

Tyto metody musí implementovat každý typ konfigurace.

▶ Poznámka 3.3 (Uživatelská interakce s třídou konfigurace). Zvenku uživatel nevolá žádnou z popsaných metod, ale pouze konstruktor a popřípadě upravuje hodnoty ve slovníku `data`, pokud nastavil `gui` na hodnotu `False`, jinak se mu zobrazí okno s konfigurací ve formátu YAML (viz sekce 3.2.5). Příklad konfigurace přes slovník `data` jsem uvedl v Příloze B a také v přiložené dokumentaci.

Protože se slovník `data` skládá z více vnořených slovníků, tak je potřeba mít klíče `keys` v metodě `get_values(layer, keys)` ve speciálním formátu — řetězce klíčů, oddělené tečkou, které vedou k cílovému slovníku — například `"Precision.weight"`. K získání hodnot ze slovníku v takovémto tvaru jsem vytvořil statickou metodu `deep_get`, popsanou ve výpisu 3.4 a inspirovanou [21].

```
from functools import reduce
@staticmethod
def deep_get ( data: dict, nested_key: str, default = None ):
    return reduce ( lambda d, k: d.get(k, default) if isinstance (d, dict)
                    else default, nested_key.split("."), data )
```

Výpis kódu 3.4 Statická metoda `deep_get()` pro získání hodnoty ze vnořeného slovníku

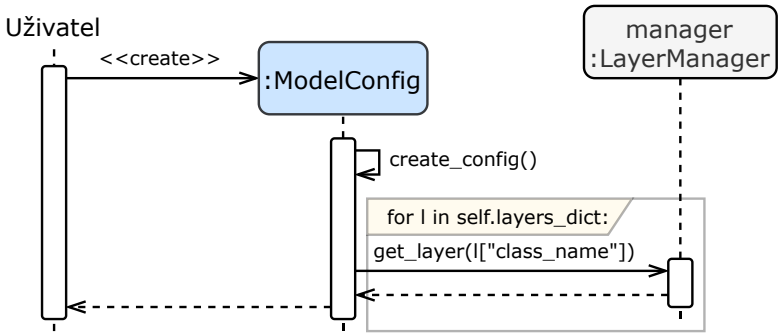
[4] implementují jí zděděné typy konfigurací

3.2.2 Třída ModelConfig

Tato třída zapouzdřuje konfiguraci nad celým modelem bez specifikací pro vrstvy. Implementuje metodu `create_config()`, kteoru lze vidět ve výpisu 3.5. Iteruje se v ní přes třídní proměnnou `layers_dict`, kde se v `try-except` bloku z instance `layer_manager` vyzvedává potomek třídy `Layer`, indexovaný názvem jeho třídy. Komunikaci těchto tříd znázorňuje obrázek 3.5. Pokud třída vrstvy neexistuje, je vyvolána výjimka. Protože třída `ModelConfig` ve své konfiguraci nezohledňuje parametry vrstev, nedělá se v tomto bloku žádná další operace a po konci cyklu se pod klíč `"Precision"` ukládá objekt výchozího datového typu.

```
1 from nntovhdl.managers import layer_manager
2 def create_config ( self ) -> None:
3     for l in self.layers_dict:  # Check layer support
4         try:
5             layer_manager.get_layer(l["class_name"])
6         except KeyError as ex:
7             print(ex)
8             raise TypeError("Can't initialize ModelConfig.")
9     self.data["Precision"] = self.default_precision
```

Výpis kódu 3.5 implementace metody `create_config()` třídy `ModelConfig`



Obrázek 3.5 Komunikace třídy `ModelConfig` s třídou `LayerManager`

Dále je implementována metoda `get_values()`, kterou lze vidět ve výpisu 3.6. Iteruje se v ní přes klíče `keys` a pro každý se získá hodnota `val`, která pod ním je. Protože je tato metoda *polymorfní*, tak volající přímo neví, jaké všechny klíče třída `ModelConfig` zapouzdřuje, a proto pokud klíč `k` obsahuje alespoň podřetězec `"Precision"`, je mu vrácena hodnota pod tímto klíčem. Příkladem může být situace, kdy bude volající požadovat hodnotu pod klíčem `"Precision.weight"`, což by při kontrole přesné shody s klíčem `"Precision"` vedlo k chybě — proto jsem shodu ošetřil tímto způsobem. Pokud nenastane tato možnost a klíč neexistuje, je vyvolána výjimka. Každá úspěšně získaná hodnota `val` se ukládá do pole a to je v závěru vráceno jako `tuple`.

3.2.3 Třída LayerTypeConfig

Tato třída zapouzdřuje konfiguraci o jednotlivých typech vrstev. Implementuje metodu `create_config()`, kterou lze vidět ve výpisu 3.7. Princip je podobný implementaci třídy `ModelConfig`, ale na potomka třídy `Layer`, získaného z manageru, je volána jeho statická metoda `make_config()`, vracející slovník s konfigurací této vrstvy, který se uloží do proměnné `l_config`. Komunikaci

```
def get_values ( self, layer: Layer, keys: list[str] ) -> tuple:
    values = []
    for k in keys:
        val = self.deep_get(self.data, k)
        if val is None:
            if "Precision" in k:
                val = self.data["Precision"]
            else:
                raise KeyError(f"Key '{k}' was not found.")
        values.append(val)
    return tuple(values)
```

Výpis kódu 3.6 implementace metody `get_values()` třídy `ModelConfig`

těchto tříd znázorňuje obrázek 3.6. Slovník `l_config` se přidává do slovníku `type_config`, a to pod klíč, který nese název třídy související vrstvy. Ještě před získáním třídy vrstvy se zjišťuje, zda už byla vrstva zpracována — jinak by se konfigurace stejných typů vrstev pokaždé přepisovala. Na konci se slovník `type_config` ukládá do třídní proměnné `data`.

```
from nntovhdl.managers import layer_manager
def create_config ( self ) -> None:
    type_config = {}
    for l in self.layers_dict:
        if l["class_name"] in type_config: continue
        try:
            l_config = layer_manager.get_layer(l["class_name"]).make_config(l,
            ↪  self.default_precision)
        except KeyError as ex:
            print(ex)
            raise TypeError("Can't initialize LayerTypeConfig.")
        type_config[l["class_name"]] = l_config
    self.data = type_config
```

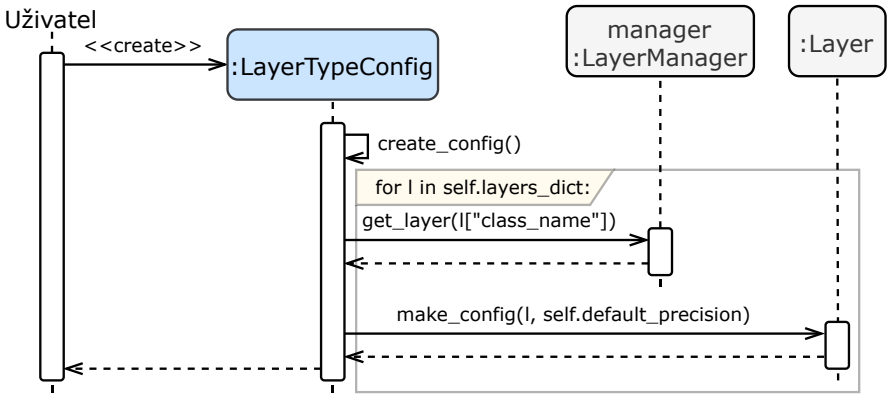
Výpis kódu 3.7 implementace metody `create_config()` třídy `LayerTypeConfig`

Dále je implementována metoda `get_values()`, znázorněna ve výpisu 3.8, ve které se nejprve získá slovník s daty konfigurace volající vrstvy `layer` uloží se do proměnné `layer_data`. Následně se opět iteruje přes klíče `keys`. Na rozdíl od implementace třídy `ModelConfig` se zde navíc získá hodnota `val` pod klíčem `k` a zbytek je totožný.

3.2.4 Třída LayerNameConfig

Tato třída zapouzdřuje konfiguraci každé vrstvy, která se v modelu nachází. Také implementuje metodu `create_config()`, ale nebudu jí zde uvádět, protože je skoro totožná s implementací třídy `LayerTypeConfig` ve výpisu 3.7 — neobsahuje větvení na řádku 5 a místo ukládání slovníku s konfigurací konkrétní vrstvy do proměnné `type_config`, ho ukládá do proměnné `name_config` pod klíč nesoucí název vrstvy.

V poslední řadě implementuje metodu `get_values()`, totožnou s implementací ve výpisu 3.8, až na řádek 3, který v tomto případě není potřeba, a proto jí zde také nebudu uvádět.



Obrázek 3.6 Komunikace třídy `LayerTypeConfig` s rozhraním vrstev

```
def get_values ( self, layer: Layer, keys: list[str] ) -> tuple:
    values = []
    layer_type = layer.keras_layer_json["class_name"]
    layer_data = self.data.get(layer_type)
    if layer_data is None:
        raise KeyError(f"Layer type '{layer_type}' was not found in config.")
    for k in keys:
        val = self.deep_get(layer_data, k)
        if val is None:
            raise KeyError(f"Key '{k}' was not found.")
        values.append(val)
    return tuple(values)
```

Výpis kódu 3.8 implementace metody `get_values()` třídy `LayerTypeConfig`

3.2.5 Úprava konfigurace pomocí grafického rozhraní

Jednou z možností, jak z uživatelského hlediska upravit konfiguraci, je pomocí grafického rozhraní. To zajišťuje funkce `edit_with_gui()` z balíčku `utils`, která je inspirovaná návodem [22] pro vytvoření textového editoru pomocí knihovny `tkinter`. Protože má tato funkce 40 řádků, tak zde uvedu pouze zjednodušenou verzi, znázorňující převod dat konfigurace do formátu YAML a zpět, viz výpis 3.9.

Nejprve je potřeba uložit slovník `config.data` do JSON formátu. Protože tento slovník obsahuje objekty datových typů, tak je potřeba zaručit, že je bude možné zpětně zkonstruovat. K tomu je potřeba použít knihovnu `jsonpickle`, ukládající objekty ve slovníku do speciálního formátu. Slouží k tomu funkce `jsonpickle.encode()`, která převede slovník do JSON řetězce se speciálním formátem pro objekty. Teprve po tomto kroku je možné pomocí funkce `json.loads()` převést `config.data` do slovníku, kompatibilního s JSON. Dále se použije funkce `yaml.dump()`, která vrátí slovník v YAML formátu. Ten už je připraven pro vytisknutí do grafického okna.

Po editaci konfigurace se získá upravený YAML text a načte se pomocí funkce `yaml.safe_load()` do slovníku který je kompatibilní s JSON. Ten ale ještě neobsahuje zkonstruované objekty — je potřeba zavolat funkci `jsonpickle.decode()`, která převede speciálně formátované řetězce zpět do Python objektů.

▶ Poznámka 3.4 (zápis objektů v JSON formátu). Nevýhodou převodu slovníku s objekty do JSON formátu je, že řetězce objektů obsahují speciální předponu — u vestavěných Python ob-

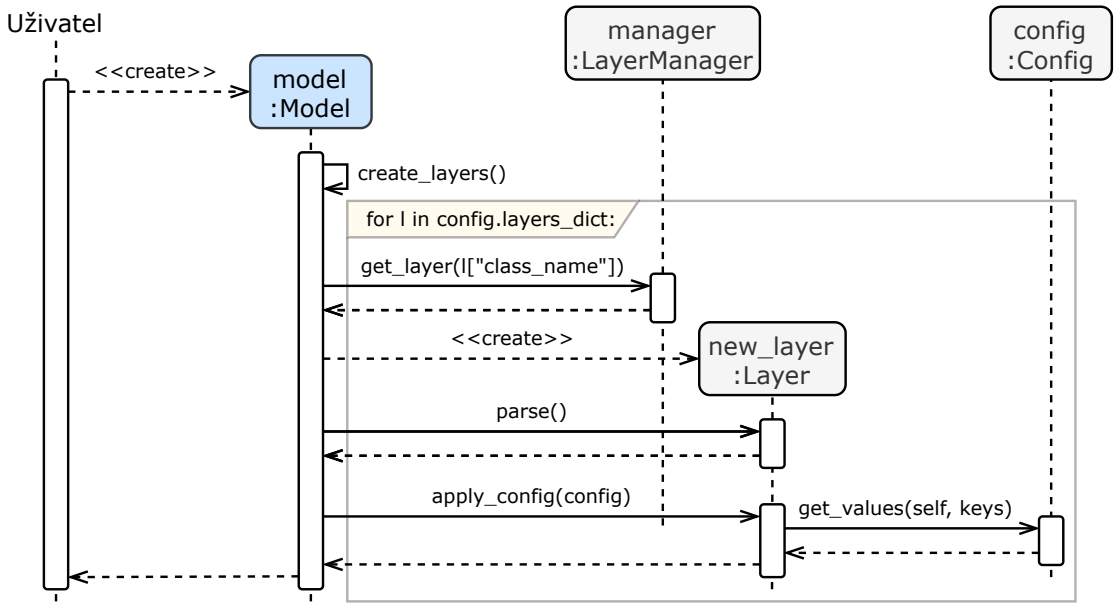
```python
import json, jsonpickle, yaml
def edit_with_gui ( config: Config ) -> None:
    # Pickle config dict to JSON string
    config_data_pickled = jsonpickle.encode(config.data)
    # Convert JSON string to dict (required by yaml.dump)
    config_dict = json.loads(config_data_pickled)
    # Convert dict to YAML format
    config_yaml = yaml.dump(config_dict, sort_keys=False, indent=4)
    # ... create window, insert YAML text, get text back after edit ...
    # Get YAML from text field
    config_yaml = txt.get("1.0", tk.END)
    # Convert YAML to dict
    config_dict = yaml.safe_load(config_yaml)
    # Convert dict to JSON string, depickle and save to config dict
    config.data = jsonpickle.decode(json.dumps(config_dict))
```

Výpis kódu 3.9 Princip implementace funkce `edit_with_gui()`

jektů je to předpona `builtins`, takže by se například typ `float` zapsal jako `builtins.float`. U ostatních objektů je předponou knihovna, ze které pochází. Bez použití předpony se objekt nezkonstruuje.

3.3 Rozhraní transformovaného modelu

Podobně jako hls4ml implementuje třídu `HLSModel`, v mém řešení implementuji třídu `Model`. Jedná se o strukturu, zapouzdřující parsovaná data, získaná z Keras modelu a konfigurace. Také se zde propojují všechny předešlé části, tedy rozhraní vrstev a rozhraní konfigurace, což znázorňuje obrázek 3.7, kde lze vidět jejich komunikaci.



Obrázek 3.7 Komunikace třídy `Model` s rozhraním vrstev a konfigurace

▶ Poznámka 3.5 (Aktivační funkce a vrstvy). V poznámce 2.1 jsem zmiňoval, že se v HLS modelu generují aktivační vrstvy kvůli hardwarové implementaci. To v mém modelu není potřeba, protože hardware, který je generovaný, to nepotřebuje. Stačí, aby každá vrstva měla atribut, udávající, jaká aktivační funkce je použita. Dokonce stačí, aby tento atribut byl řetězec se jménem aktivační funkce.

3.3.1 Třída Model

Konstruktoru třídy `Model` je potřeba předat:

- povinný argument `keras_model`, reprezentující model z Kerasu,
- nepovinný argument `config: Config`, reprezentující potomka třídy `Config`.

Kontroluje se, zda je hodnota argumentu `config` rovna `None`, což je jeho výchozí hodnota. Pokud ano, tak se vytvoří instance třídy `ModelConfig`, do které se předá argument s modelem z Kerasu. Protože si konfigurace ukládá model z Kerasu, ze kterého vychází, tak se kontroluje, zda se shoduje s modelem z argumentu. Pokud ne, vyvolá se výjimka. Dále se argument `config` uloží do stejnojmenné třídní proměnné a větví se podle booleovské hodnoty `config.gui`. Ta říká, zda se má konfigurace upravit pomocí grafického rozhraní. V závěru se `keras_model` uloží do stejnojmenné třídní proměnné, vytvoří se prázdný slovník `layers`, kde klíčem je název vrstvy a hodnotou potomek třídy `Layer`, a zavolá se metoda `create_layers()`.

Třída `Model` dále implementuje metody:

- `__iter__() -> Model` — automaticky volaná při iteraci přes `Model`,
- `__next__() -> Layer` — automaticky volaná pro vrácení další vrstvy v rámci iterace,
- `get_parameters(nested_lists: bool = True) -> dict` — vrací slovník s parametry, překonvertovanými do typů specifikovaných konfigurací,
- `create_layers() -> None` — vytváří jednotlivé potomky třídy `Layer`, které přidává do výše zmíněného slovníku `layers`.

▶ Poznámka 3.6 (Iterování přes `Model`). Někdy je potřeba iterovat přes všechny vrstvy třídy `Model`. Pokud by se ale `for` cyklus zapsal ve tvaru `for l in model`, kde `model` je instance `Model`, tak by nefungoval správně nebo vyvolal chybovou hlášku. Pro podporu iterování nabízí Python vestavěné funkce `__iter__()` a `__next__()`, zajišťující správnou iteraci přes slovník `layers`, které jsem implementoval. Díky tomu lze používat `for` cyklus ve výše zmíněném tvaru.

Metoda `get_parameters()` má jediný argument `nested_lists` booleovského typu, kterým lze říct, zda se mají výsledné parametry nechat v podobě n dimenzionálních polí, jako jsou `numpy` pole, ze kterých parametry vychází, nebo je uložit do pole s jednou dimenzí. Uvnitř metody se na datové typy, do kterých se konvertují parametry a jsou definovány konfigurací, volá jejich konstruktor s příslušnou hodnotou parametru a následně built-in funkce `str()`.

Implementaci metody `create_layers()` lze vidět ve výpisu 3.10. Nejdříve se získají pole s jmény vstupních a výstupních vrstev. Dále se iteruje přes slovník `layers_dict` a vytvářejí se instance tříd jednotlivých vrstev pomocí manageru vrstev (třída `LayerManager` ze sekce 3.1.2). Po vytvoření nové vrstvy se větví dle toho, jestli je model z Kerasu sekvenční či funkční, protože jsou vrstvy v Keras modelu uloženy odlišnými způsoby. Poté jsou na instance `Layer` volány jejich metody `parse()` a `apply_config()`. V závěru je vrstva uložena do slovníku `layers` pod klíč se svým jménem.

▶ Pozorování 3.7 (Polymorfismus v metodě `create_layers()`). Tato metoda je jedním z příkladů využití navrženého polymorfismu — pracuje se zde s abstraktním rozhraním třídy `Layer`, díky čemuž se nemusí větvit dle typu vrstvy. Souběžně s tím se využívá i abstraktního rozhraní třídy `Config`, jehož potomek je aplikován na vrstvu a nastavuje se zcela bez potřeby větvení dle typu konfigurace.

```
def create_layers ( self ) -> None:
    self.input_names: list[str] = self.keras_model.input_names.copy()
    self.output_names: list[str] = self.keras_model.output_names.copy()
    prev_layer: Layer = None  # Used only if model is Sequential
    for l in self.config.layers_dict:
        new_layer: Layer = None  # For created layer
        if l["class_name"] == "InputLayer":
            new_layer = layer_manager.get_layer(l["class_name"])(l,
            ↪  self.keras_model.input.node.layer)
        else:
            new_layer = layer_manager.get_layer(l["class_name"])(l,
            ↪  self.keras_model.get_layer(l["config"]["name"]))

        if "inbound_nodes" in l:  # Functional model
            if len(l["inbound_nodes"]) > 0:  # If not input layer
                for inbound_node in l["inbound_nodes"]:
                    prev_l_name = inbound_node[0][0]
                    new_layer.prev_layers.append(self.layers[prev_l_name])
                    self.layers[prev_l_name].next_layers.append(new_layer)
        else:  # Sequential model
            if prev_layer is not None:  # If not input layer
                new_layer.prev_layers.append(prev_layer)
                self.layers[prev_layer.name].next_layers.append(new_layer)
            prev_layer = new_layer
        new_layer.parse()
        new_layer.apply_config(self.config)
        self.layers[new_layer.name] = new_layer
```

Výpis kódu 3.10 implementace metody `create_layers()` třídy `Model`

3.4 Generování VHDL

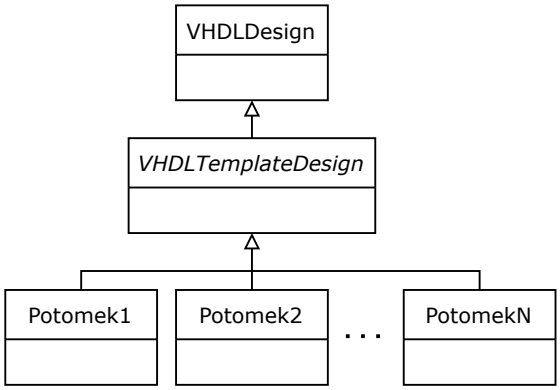
Pro generování VHDL slouží hiearchie tříd, která je znázorněna na obrázku 3.8. Na nejnižší úrovni je třída `VHDLDesign`, která poskytuje rozhraní pro práci s VHDL designy. Z ní dědí abstraktní třída `VHDLTemplateDesign`, která rozhraní rozšiřuje a je svázaná se svou šablonou, ze které při generování vychází. Na nejvyšší úrovni jsou potomci třídy `VHDLTemplateDesign`, které vytváří a implementuje uživatel. Pro generování designů z výzkumu mého vedoucího jsem vytvořil potomky `VHDLNeuralNetwork` a `VHDLNeuralNetworkTop`.

Nutno podotknout, že implementovaný generátor není schopen navrhovat VHDL za běhu. Funguje tak, že se mu předá již hotové VHDL, ve kterém jsou zvýrazněné části[5], za které se má doplňovat VHDL, specifikované kódem generující třídy, například `VHDLNeuralNetwork`. K tomu, aby se dalo generování naprogramovat, je poskytnuto vysokoúrovňové rozhraní. Uživatel, který implementuje generování nového designu, se potom nemusí starat o parsování souborů a pro práci s jednotlivými VHDL strukturami používá jednoduché rozhraní tříd, do kterých jsou struktury zapouzdřeny. Více informací bude poskytnuto dále v textu.

▶ Poznámka 3.8 (význam pojmu VHDL design). Pod tímto pojmem si lze představit několik věcí, avšak v mém návrhu je tento pojem definován jako struktura, která má vždy *entitu* a *architekturu*.

Před vytvářením designů a jejich generováním je nejdříve potřeba umět zpracovat VHDL

[5] princip bude vysvětlen dále v textu



Obrázek 3.8 Diagram tříd VHDL designů

soubor, pro jehož zapouzdření jsem vytvořil třídu `VHDLFile`. Protože je tyto soubory potřeba parsovat, tak jsem využil rozhraní regulárních výrazů, které poskytuje knihovna `re` (viz sekce 2.3.2), a vytvořil soubor `nntovhdl/regexps.py`. V něm jsem vytvořil konstanty a také funkce, protože je občas potřeba podvýrazy parametrizovat, což lze pomocí argumentů funkce. Pro příklad jsem do výpisu 3.11 uvedl konstantu s regulárním výrazem pro hlavičku entity a funkci s regulárním výrazem pro konec entity, kde je potřeba parametrizovat název entity.

```
1 import re
2 ENTITY_START_RE = re.compile(r'^entity\s+(?P<name>\w+)\s+is', re.MULTILINE |
  ↪  re.IGNORECASE)
3 def ENTITY_END_RE ( entity_name: str = r'(?P<name>\w+)' ):
4     return re.compile(r'^end(?:\s*(?P<keyword>entity))?(?:\s*' + entity_name +
      ↪  r')?\s*;', re.MULTILINE | re.IGNORECASE)
```

Výpis kódu 3.11 Příklad konstanty a funkce s regulárním výrazem pro parsování VHDL souborů

V následujících sekcích se budu nejprve zabývat abstraktní třídou `VHDLObject`, zapouzdřující jednotlivé VHDL struktury v souboru, a jejími potomky. Poté popíšu třídu `VHDLFile` a dále třídy `VHDLDesign`, `VHDLTemplateDesign` a její potomky, konkrétně `VHDLNeuralNetwork` a `VHDLNeuralNetworkTop`, včetně tipů, jak vytvořit další potomky. V závěru kapitoly popíšu třídu `BlkRamLayerManager`, zapouzdřující názvy VHDL souborů, ze kterých se generují dostupné designy.

▶ Poznámka 3.9 (Syntaxe VHDL). Znalosti VHDL syntaxe jsem sice získal v rámci předmětů mého oboru, ale přesto bylo občas potřeba čerpat z dokumentace, zabývající se syntaxí tohoto jazyka. Touto dokumentací byla knížka *VHDL handbook*, viz [23].

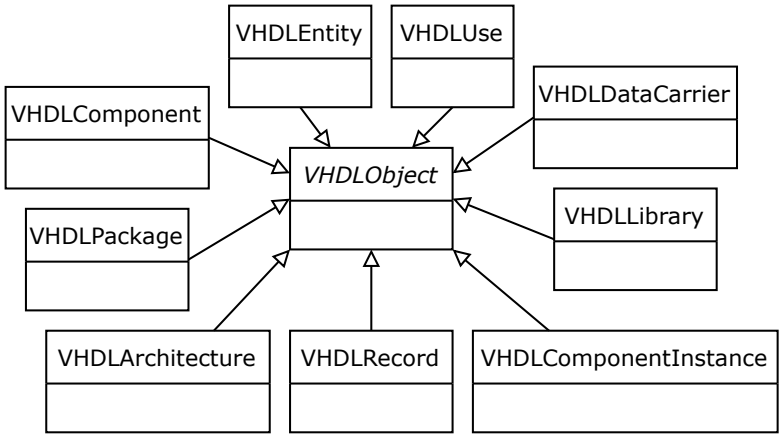
3.4.1 Třída VHDLObject

Tato třída zapouzdřuje a definuje rozhraní následujících VHDL struktur, pro které jsem vytvořil třídy:

- `VHDLLibrary` — knihovna,
- `VHDLUse` — použití knihovny (dále uses),
- `VHDLDataCarrier` — nosič dat či parametr VHDL struktury (`port`, `signal`, apod.),

- `VHDLEntity` — entita,
- `VHDLComponent` — komponenta,
- `VHDLComponentInstance` — instance komponenty,
- `VHDLArchitecture` — architektura,
- `VHDLRecord` — datový typ `record`,
- `VHDLPackage` — balíček (dále `package`).

Hiearchii tříd znázorňuje obrázek 3.9.



Obrázek 3.9 Diagram tříd VHDL struktur

Třída `VHDLObject` sice definuje abstraktní konstruktor, aby jej musel uživatel ve zděděných třídách implementovat, ale nespecifikuje žádné argumenty. Kromě toho definuje následující dvě metody:

- `string() -> str` — abstraktní metoda, která má vracet řetězec VHDL struktury, aby se mohla zapsat do souboru,
- `__str__() -> str` — přetížená built-in metoda, která volá zmíněnou abstraktní metodu `string()`.

Dále budu popisovat jednotlivé třídy výše zmíněných struktur.

3.4.2 Třídy VHDLLibrary a VHDLUse

Obě třídy jsou skoro totožné — liší se jen v řetězci, který zapouzdřují. Třída `VHDLLibrary` zapouzdřuje řetězec `library <nazev_knihovny>;` a třída `VHDLUse` zapouzdřuje řetězec `use <nazev_knihovny>.<package>;`. Obě třídy implementují konstruktor, přijímající argument `lib_name`, respektive `use_name`, který specifikuje název knihovny, respektive řetězec s názvem knihovny a balíčkem z ní. V obou případech se argument ukládá do třídní proměnné `name`. Třídy také implementují metodu `string()`, viz výpis 3.12. Dále třídy implementují následující metody, díky kterým je lze ukládat do množiny (`set`):

- `__eq__(other) -> bool` — operátor rovnosti dvou instancí,
- `__ne__(other) -> bool` — operátor nerovnosti dvou instancí,

- `__hash__(other) -> int` — výpočet hashe instance,
- `__lt__(other) -> bool` — operátor porovnání (<) dvou instancí.

```
# Implementace tridy VHDLLibrary
def string ( self ) -> str:
    return "library " + self.name + ";"
# Implementace tridy VHDLUse
def string ( self ) -> str:
    return "use " + self.name + ";"
```

Výpis kódu 3.12 Implementace metody `string()` tříd `VHDLLibrary` a `VHDLUse`

3.4.3 Třída VHDLDataCarrier

Tato třída má za úkol zapouzdřovat většinu nosičů dat — například `port`, parametr `generic`, `signal`, `variable` a další. Protože se velmi často s těmito nosiči pracuje obdobně, nevytvářel jsem pro ně separátní třídy. Konstruktor jsem tomu uzpůsobil tak, že přijímá povinný argument `name: str`, protože název má každý nosič dat. Konstruktor dále přijímá následující nepovinné argumenty, které jsou defaultně nastaveny na hodnotu `None`:

- `type: str` — jedná se o typ nosiče, například `signal`,
- `direction: str` — směr toku dat, který je specifický pro `port`,
- `data_type: str` — datový typ nosiče, například `std_logic`,
- `default_value` — výchozí hodnota nosiče, která může být číslo i řetězec.

Tyto argumenty se v závěru ukládají do stejnojmenných třídních proměnných.

Dále třída implementuje metodu `string()`, kterou lze vidět ve výpisu 3.13. Metoda je udělána tak, že do výsledného řetězce přidává jen ty proměnné, které byly inicializovány, tedy nemají výše zmíněnou hodnotu `None`.

```
def string ( self ) -> str:
    result = ""
    result += self.type + " " if self.type is not None else ""
    result += self.name + " :"
    result += " " + self.direction if self.direction is not None else ""
    result += " " + self.data_type if self.data_type is not None else ""
    result += " := " + str(self.default_value) if self.default_value is not
    ↪  None else ""
    result += ";"
    return result
```

Výpis kódu 3.13 Implementace metody `string()` třídy `VHDLDataCarrier`

3.4.4 Třída VHDLEntity

Tato třída zapouzdřuje entitu. Její konstruktor přijímá povinné argumenty:

- `entity: str` — řetězec s entitou,
- `entity_name: str` — název entity, aby se nemusel znovu parsovat.

Tyto argumenty se ukládají do třídních proměnných a dále se parsují generické parametry a porty, které se zapouzdřují do třídy `VHDLDataCarrier` a ukládají do slovníku `generics`, respektive `ports`, kde klíčem je název generického parametru, respektive portu.

Dále je implementována metoda `string()`, viz výpis 3.14. Nejdříve se pomocí konstanty `ENTITY_START_RE` a funkce `re.sub()` nahradí název entity, protože její jméno mohlo být změněno. Pomocí funkcí `ENTITY_END_RE()` a `re.sub()` se název nahradí i na konci entity, v případě, že tam bylo původně — zakončovací řádek entity totiž nemusí jméno obsahovat. Poté se pomocí funkcí `GENERIC_RE()` a `re.sub()` aktualizují hodnoty generických parametrů.

```
1  import re
2  def string ( self ) -> str:
3      # Replace name in entity start statement (in case it was changed)
4      result = ENTITY_START_RE.sub(f'entity {self.name} is', self._entity_str)
5      # Search for name in entity end
6      ent_end = ENTITY_END_RE().search(result)
7      if ent_end.group("name"):  # If entity end contains name, replace it
8          result = ent_end.re.sub(f'end {ent_end.group("keyword") + " " if
           ↪  ent_end.group("keyword") else ""}{self.name};', result)
9      # Update generic default values
10     for g in self.generics.values():
11         tmp = GENERIC_RE(g.name).sub(r'\g<before_value>' +
           ↪  str(g.default_value) + r'\g<after_value>', result)
12         result = tmp
13     return result
```

Výpis kódu 3.14 Implementace metody `string()` třídy `VHDLEntity`

3.4.5 Třída VHDLComponent

Tato třída zapouzdřuje komponentu. Implementuje jednoduchý konstruktor, přijímající jediný povinný argument `entity`, který se ukládá do třídní proměnné `_entity`.

Dále je implementována metoda `string()`, viz výpis 3.15. Nejdříve se do výsledného řetězce přidá komentář, který uvádí tělo komponenty. Tělo komponenty je přidáno v dalším kroku a vychází ze své entity, která se musí nejdříve upravit. Pomocí konstanty `ENTITY_START_RE` a funkce `re.sub()` se v hlavičce nahrazuje klíčové slovo `entity` za `component` a na návratovou hodnotu se znovu volá funkce `re.sub()`, tentokrát s funkcí `ENTITY_END_RE()`, pomocí které se nahradí klíčové slovo i na konci a přidá se název komponenty.

Dále jsou implementovány následující `@property`:

- `name() -> str` — vrací hodnotu `self._entity.name`,
- `generics() -> dict[str, VHDLDataCarrier]` — vrací hodnotu `self._entity.generics`,
- `ports() -> dict[str, VHDLDataCarrier]` — vrací hodnotu `self._entity.ports`.

```
def string ( self ) -> str:
    # Add component comment
    result = f'--{"="*15} COMPONENT {self.name} {"="*15}\n'
    # Modify entity body
    result += ENTITY_END_RE(self.name).sub(r'end component ' + self.name +
    ↪  r';', ENTITY_START_RE.sub(r'component \g<name> is',
    ↪  str(self._entity)))
    return result
```

Výpis kódu 3.15 Implementace metody `string()` třídy `VHDLComponent`

Tyto `@property` záměrně odkazují na entitu, protože z hlediska VHDL musí mít název komponenty stejné jméno, generické parametry a porty, jako její entita.

3.4.6 Třída VHDLComponentInstance

Tato třída zapouzdřuje instanci komponenty. Její konstruktor přijímá povinné argumenty:

- `component: VHDLComponent` — komponenta, z které se instance vytváří,
- `name_prefix: str` — předpona jména instance,
- `name_suffix: str` — přípona jména instance.

▶ Poznámka 3.10 (Jméno instance komponenty). Předpona i přípona jména instance komponenty je povinná z toho důvodu, aby si volající uvědomil, že instance nemohou mít stejné jméno a byl donucen tyto jména upravovat. Celé jméno lze poté získat pod `@property name() -> str`, které ke jménu komponenty připojuje předponu a příponu.

Dále se v konstruktoru vytváří třídní proměnné `generics` a `port_maps`. Do `generics` se ukládají generiky komponenty v podobě slovníku, kde klíčem je *název generiky* a hodnotou je *hodnota generiky*. Do `port_maps` se ukládají porty komponenty také v podobě slovníku, kde klíčem i hodnotou je *název portu*[6]. Oba slovníky slouží k modifikaci těchto parametrů — u generik je potřeba nastavit požadovanou hodnotu a porty je potřeba namapovat na požadované nosiče dat.

Dále třída implementuje metodu `string()`, kterou zde z důvodu přehlednosti ukazovat nebudu. Jediné, co dělá, je vrácení řetězce s tělem instance komponenty a částmi `generic map` a `port map` s odpovídajícím odsazením řádků pro lepší přehlednost.

3.4.7 Třída VHDLArchitecture

Tato třída zapouzdřuje architekturu. Její konstruktor přijímá povinné argumenty:

- `arch: str` — řetězec s architekturou,
- `arch_name: str` — název architektury, aby se nemusel znovu parsovat,
- `entity_name: str` — název entity, aby se také nemusel parsovat.

Tyto argumenty se ukládají do třídních proměnných a vytváří se slovník `code_to_add: dict[str, list]`. Ten je důležitý pro třídu `VHDLTemplateDesign`, která do něj ukládá části generovaného VHDL kódu. Klíč slovníku je místo v architektuře (reprezentované placeholderem), kam

[6] nemá to žádný hlubší smysl, jde jen o to nastavit nějakou výchozí hodnotu

se má kód vložit, a hodnotou je pole s kódy, které se na toto místo mají přidat. Detailnější popis této myšlenky je vysvětlen v popisu zmíněné třídy `VHDLTemplateDesign` — sekce 3.4.12.

Dále je implementována metoda `string()`, viz výpis 3.16. Nejdříve se pomocí konstanty `ARCH_START_RE` a funkce `re.sub()` nahradí název architektury a entity, protože mohly být změněny. Pomocí funkcí `ARCH_END_RE()` a `re.search()` se nahradí název i na konci architektury, v případě, že tam původně bylo. Počínaje řádkem 9 se v těle architektury, vycházející ze šablony, nahrazují klíče ze slovníku `code_to_add` za prvky pole, které je související hodnotou ve slovníku. To se děje za pomoci funkcí `PLACEHOLDER_RE()` a `re.search()`. Také se zde využívá pomocná funkce `indent()` z mého balíčku `utils`, která řetězec v prvním argumentu odsadí řetězcem v druhém argumentu.

```
1  from nntovhdl.utils import indent
2  def string ( self ) -> str:
3      # Replace name in architecture start statement (in case it was changed)
4      result =
       ↪  ARCH_START_RE.sub(f'architecture {self.name} of {self.entity_name} is',
       ↪  self._arch_str)
5      # Search for name in architecture end
6      arch_end = ARCH_END_RE().search(result)
7      if arch_end.group("name"):  # If arch end contains name, replace it
8          result = arch_end.re.sub(f'end {arch_end.group("keyword") + " " if
           ↪  arch_end.group("keyword") else ""}{self.name};', result)
9      # Replace all placeholders with corresponding code
10     for placeholder, replacements in self.code_to_add.items():
11         # Get indentation
12         ind = PLACEHOLDER_RE(placeholder).search(result).group("indentation")
13         # Join all replacements into one string and indent them
14         repl_str = utils.indent ( '\n'.join ( map(str, replacements) ), ind )
15         # Replace placeholder with replacement
16         tmp = PLACEHOLDER_RE(placeholder).sub(repl_str, result)
17         result = tmp
18     return result
```

Výpis kódu 3.16 Implementace metody `string()` třídy `VHDLArchitecture`

3.4.8 Třída VHDLRecord

Tato třída zapouzdřuje datový typ `record`. Její konstruktor přijímá povinné argumenty:

- `record: str` — řetězec s tělem typu `record`,
- `record_name: str` — název typu `record`,
- `elements: dict[str, VHDLDataCarrier]` — slovník s parametry, které typ `record` zapouzdřuje.

Implementace metody `string()` je jednoduchá, protože vrací přímo řetězec s typem `record`, který byl předán jako argument v konstruktoru.

3.4.9 Třída VHDLPackage

Tato třída zapouzdřuje `package`. Tuto třídu jsem vytvořil pro podporu datového typu `record`, který je v generovaných VHDL designech využíván. Další struktury, které v `package` mohou být, nejsou v tuto chvíli zohledněny.

Konstruktor této třídy přijímá povinné argumenty:

- `pkg: str` — řetězec s `package`,
- `pkg_name: str` — název `package`, aby se nemusel znovu parsovat.

Tyto argumenty jsou následně uložené do třídních proměnných a dále se pomocí konstanty `RECORD_RE` a funkce `re.finditer()` parsují datové typy `record`, které se v této `package` nachází. Ty jsou ukládány do slovníku `records`, kde klíčem je *název* tohoto datového typu a hodnotou je *instance* třídy `VHDLRecord`.

Implementace metody `string()` je jednoduchá, protože vrací přímo řetězec s `package`, který byl předán jako argument v konstruktoru.

3.4.10 Třída VHDLFile

Tato třída zapouzdřuje VHDL soubor a poskytuje rozhraní, které ulehčuje práci se souborem. Volající pak nemusí používat souborové operace pro čtení, ale jsou mu poskytnuty metody, vracející potomky třídy `VHDLObject`. Vysokoúrovňové rozhraní je poskytnuto i pro zápis do souboru. Veškeré parsování je prováděno pomocí mnou navržených konstant a funkcí, které obsahují regulární výrazy.

Konstruktor třídy přijímá povinné argumenty:

- `file_name: str` — cesta k VHDL souboru,
- `file_mode: str` — režim souboru.

Kontroluje se, zda soubor má příponu `.vhd` — pokud ne, je vyvolána výjimka. V dalším kroku se kontroluje, zda je režim souboru čtení (`r`) či zápis (`w`), protože tyto režimy pro práci s VHDL souborem postačují. V případě neúspěchu je opět vyvolána výjimka. V závěru se inicializují třídní proměnné, do kterých se ukládají názvy jednotlivých VHDL struktur a řetězce jejich těl. Toto ukládání sice není nutné, ale při opakovaném vyzvedávání některé z VHDL struktur by se musel soubor pokaždé parsovat, což není efektivní.

Dále jsou implementovány následující metody:

- `__enter__() -> VHDLFile` — built-in funkce pro podporu context manageru,
- `__exit__() -> None` — built-in funkce pro podporu context manageru,
- `write(design: VHDLDesign) -> None` — zápis VHDL designu do souboru,
- `libraries() -> set[VHDLLibrary]` — získává knihovny ze souboru a zapouzdřuje je do `VHDLLibrary`,
- `uses() -> set[VHDLUse]` — získává `use` ze souboru a zapouzdřuje je do `VHDLUse`,
- `package() -> VHDLPackage` — získává `package` ze souboru a zapouzdřuje jej do `VHDLPackage`,
- `entity() -> VHDLEntity` — získává entitu ze souboru a zapouzdřuje jí do `VHDLEntity`,
- `architecture() -> VHDLArchitecture` — získává architekturu ze souboru a zapouzdřuje jí do `VHDLArchitecture`.

První dvě built-in metody slouží pro použití context manageru, což je Python konstrukce se slovem `with`. Příklad takového použití lze vidět například v konstruktoru třídy `VHDLDesign`, viz výpis 3.19. V hlavičce této konstrukce se vytvoří instance třídy `VHDLFile`, se kterou lze v následujícím bloku pracovat. Volání se slovem `with` zaručí, že se zavolá funkce `__enter__()`, která otevře soubor `file_name` a po konci bloku se automaticky zavolá funkce `__exit__()`, která tento soubor zavře.

Metoda `write()` je jednoduchá, protože se v ní na soubor pouze volá funkce `write()` s argumentem `str(design)`, což lze, protože třída `VHDLDesign` přetěžuje built-in funkci `__str__()` (více v sekci 3.4.11).

Implementaci metody `libraries()` lze vidět ve výpisu 3.17. Nejdříve se zjistí, zda už byly knihovny v tomto souboru parsovány — pokud ne, provede se první parsování. V cyklu se čtou jednotlivé řádky souboru a pomocí mých konstant s regulárními výrazy se v řádku hledají knihovny a uses. Pokud se v řádku vyskytuje hlavička package nebo entity, znamená to, že jsou již přečteny všechny knihovny a uses. V závěru se vrací množina s instancemi `VHDLLibrary`.

▶ Poznámka 3.11 (Čtení uses v metodě `library()`). Přestože se v této metodě mají číst knihovny, čtou se souběžně s nimi i uses. Je to z toho důvodu, že pokud volající chce získat knihovny, je velmi pravděpodobné, že jeho další volání bude pro uses. Voláním pro uses se potom nemusí soubor znovu parsovat.

```
1  def libraries ( self ) -> set[VHDLLibrary]:
2      if not self._lib_names:  # First libs parsing
3          self._file.seek(0)
4          for line in self._file:  # Read over lines
5              # Check line for entity or package start
6              if (ENTITY_START_RE.search(line) != None) or
                   (PACKAGE_START_RE.search(line) != None):
7                  break
8              lib_name = LIBRARY_RE.search(line)  # Search for lib in line
9              use = USE_RE.search(line)  # Search for use in line
10             if lib_name:  # Lib on this line
11                 self._lib_names.append(lib_name.group("name"))
12             elif use:  # Use on this line
13                 self._use_names.append(use.group("name"))
14     return { VHDLLibrary(name) for name in self._lib_names }
```

Výpis kódu 3.17 Implementace metody `libraries()` třídy `VHDLFile`

Implementace metody `uses()` je velmi jednoduchá, protože se v ní podobně jako v metodě `libraries()` zjišťuje, zda byly uses již načítané. Pokud ne, zavolá se metoda `libraries()`, která souběžně s knihovnami čte i uses. V závěru se vrací množina s instancemi `VHDLUse`.

Výpis s implementací metody `package()` zde uvádět nebudu, protože je velmi podobná metodě `entity()` (viz výpis 3.18), akorát se používají regulární výrazy pro package. Opět se zjišťuje, zda byla tato struktura již parsována. Pokud ne, provede se první parsování. Nejdříve se najde hlavička package — pokud v souboru není, je vyvolána výjimka. Dále se ukládají všechny řádky, dokud není nalezena uzavírací řádka package. V závěru se vrátí instance `VHDLPackage`.

Následuje implementace metody `entity()`, viz výpis 3.18. Stejně jako v předešlých metodách se kontroluje, zda byla tato struktura již parsovaná a pokud ne, provede se první parsování. Nejdříve se hledá hlavička entity — pokud v souboru není, je vyvolána výjimka. Dále se ukládají všechny řádky, dokud není nalezena uzavírací řádka entity. V závěru se vrátí instance `VHDLEntity`.

```
if not self._entity_str:  # First entity parsing
    self._file.seek(0)
    for line in self._file:  # Find entity start
        entity_start = ENTITY_START_RE.search(line)
        if entity_start is not None:  # Line where entity starts
            self._entity_name = entity_start.group("name")
            self._entity_str += line
            break
    if entity_start is None:  # Entity was not found
        raise RuntimeError(f"Entity was not found in {self._file_name}.")
    for line in self._file:  # Save the rest of entity
        self._entity_str += line
        if ENTITY_END_RE(self._entity_name).search(line) is not None:
            break
    return VHDLEntity(self._entity_str, self._entity_name)
```

Výpis kódu 3.18 Implementace metody `entity()` třídy `VHDLFile`

Poslední metodou je `architecture()`, jejíž implementace je velmi podobná metodě `entity()` (viz výpis 3.18), a proto zde její výpis také nebudu uvádět. Rozdíl je v podstatě jen v použití jiných regulárních výrazů, a to pro architekturu.

3.4.11 Třída VHDLDesign

Tato třída zapouzdřuje VHDL designy a poskytuje rozhraní pro jejich vytváření a ukládání do souvisejícího VHDL souboru.

Její konstruktor přijímá jeden povinný argument `file_name`, reprezentující název souboru, ze kterého design pochází. V konstruktoru se využívá rozhraní třídy `VHDLFile`, viz výpis 3.19. Pomocí něho se načtou knihovny, uses, entita a architektura. Dále je vytvořen slovník `sub_designs`, významný zejména ve třídě `VHDLTemplateDesign` (viz sekce 3.4.12), který slouží k ukládání ostatních designů, vázajících se k tomuto designu.

```
def __init__ ( self, file_name: str ) -> None:
    with VHDLFile(file_name, "r") as f:
        self.libs = f.libraries()
        self.uses = f.uses()
        self.entity = f.entity()
        self.architecture = f.architecture()
    self.sub_designs: dict[str, VHDLDesign] = {}  # Designs created along the
    ↪  way; used in save()
```

Výpis kódu 3.19 Konstruktor třídy `VHDLDesign`

Dále jsou definovány a implementovány následující metody:

- `name() -> str` — `@property`, jejíž volání vrací název entity,
- `name(value: str) -> None` — `@name.setter`, který nastaví nové jméno entitě a poté v hlavičce architektury jméno implementované entity,

- `save() -> None` — ukládá všechny designy v `sub_designs` a poté i instanci `self`,
- `string() -> str` — vrací řetězec s celým designem,
- `__str__() -> str` — built-in funkce, která volá metodu `string()`.

Metoda `save()` je rekurzivní, viz výpis 3.20. Nejdříve se zavolá na všechny designy v `sub_designs` a poté vytvoří pomocí context manageru instanci `VHDLFile`, na kterou zavolá metodu `write()`. Rekurze se zastaví, jakmile se dojde k designu, který má slovník `sub_designs` prázdný.

```
def save ( self ) -> None:
    for sub_d in self.sub_designs.values():  # Recursively save all subdesigns
        sub_d.save()
    with VHDLFile(self.entity.name + ".vhd", "w") as f:  # Save self
        f.write(self)
```

Výpis kódu 3.20 Implementace metody `save()` třídy `VHDLDesign`

Dále je implementována metoda `string()`, viz výpis 3.21. Nejprve se abecedně seřadí knihovny a uses, které se přidají do výsledku. Poté se už jen přidají řetězce s entitou a architekturou a výsledný řetězec se vrátí.

```
def string ( self ) -> str:
    result = ""
    for lib in sorted(self.libs):
        result += str(lib) + "\n"
    result += "\n"
    for use in sorted(self.uses):
        result += str(use) + "\n"
    result += "\n"
    result += str(self.entity) + "\n"
    result += str(self.architecture) + "\n"
    return result
```

Výpis kódu 3.21 Implementace metody `string()` třídy `VHDLDesign`

3.4.12 Třída VHDLTemplateDesign

Tato abstraktní třída dědí z třídy `VHDLDesign` a poskytuje rozhraní pro generování designů, na rozdíl od předešlých tříd. Pro generování je potřeba vytvořit potomka této třídy společně dvěma soubory — VHDL soubor, reprezentující šablonu, a Python soubor s konstantami. Více informací o generování lze nalézt v číslovaném seznamu níže.

Konstruktor je v tomto případě jednoduchý, protože se volá konstruktor rodičovské třídy `VHDLDesign` s argumentem `TEMPLATE_PATH`, specifikující cestu k šabloně.

Třída definuje následující abstraktní metody:

- `TEMPLATE_PATH() -> str` — `@property`, která má za úkol vracet cestu k šabloně, ze které se při generování vychází,
- `generate() -> None` — má za úkol implementovat generování designu.

Pro přidání nového designu, který se bude generovat, je potřeba učinit několik kroků. Postup by se měl skládat z následujících kroků:

1. V adresáři `nntovhdl/sources/vhdl/templates` vytvořit nový adresář s libovolným, ale unikátním názvem.
2. Ve vytvořeném adresáři vytvořit VHDL soubor s unikátním názvem, který bude šablonou pro generovaný design. Měl by obsahovat potřebné knihovny a uses, mít definované rozhraní (tedy kompletní entitu[7]) a mít tělo architektury, která bude v libovolné fázi implementace. V architektuře budou jistě místa, kde se má generovat VHDL — tato místa je potřeba označit řetězcem s formátem `-- <IDENTIFIKATOR_CASTI>`, který bude přesně na tom místě, kam se má generovat kód, tedy včetně odpovídajícího odsazení od okraje.
3. Ve vytvořeném adresáři vytvořit Python soubor s unikátním názvem, který obsahuje takzvané *placeholdery*. Jedná se o obyčejné konstanty, ve kterých je uložen řetězec s identifikátory částí, zmíněných v předešlém kroku. Za každý identifikátor v šabloně je tedy potřeba vytvořit tuto konstantu. Konstanty mohou mít libovolné jméno, ale musí být unikátní v rámci tohoto souboru.
4. V adresáři `nntovhdl/vhdl_designs` vytvořit Python soubor s třídou, která bude dědit ze třídy `VHDLTemplateDesign` a implementovat všechny její abstraktní metody, viz výše.
5. Do souboru `nntovhdl/vhdl_designs/__init__.py` vložit řádek s ve tvaru `from nntovhdl.vhdl_designs.<balicek_s_tridou> import <nazev_tridy>`, kde `<balicek_s_tridou>` je název adresáře s třídou. Tento krok je nutný z toho důvodu, aby se po importování balíčku `nntovhdl` zvenku dala třída volat pomocí `nntovhdl.<nazev_tridy>`.

▶ Poznámka 3.12 (Cesta k šabloně). Pro specifikování cesty k šabloně v `@property TEMPLATE_PATH` lze využít konstanty `VHDL_TEMPLATES_DIR`, definované v souboru `nntovhdl/constants.py`, a přidat k ní relativní cestu k šabloně.

Tento postup bude prakticky vysvětlen v popisu následujících tříd, tedy `VHDLNeuralNetwork` a `VHDLNeuralNetworkTop`.

3.4.13 Třída VHDLNeuralNetwork

Tato třída je potomkem abstraktní třídy `VHDLTemplateDesign` a je příkladem třídy, která generuje VHDL. V následujících řádcích budou ukázky některých částí implementace metody `generate()` a postupu, který byl popsán u třídy `VHDLTemplateDesign` v sekci 3.4.12. Části implementace jsou oproti nasazené implementaci zjednodušené pro účely textu.

Do adresáře `nntovhdl/sources/vhdl/templates/neural_network` bylo nejdříve potřeba vytvořit soubor `neural_network_template.vhd`, který reprezentuje šablonu, a soubor `neural_network_placeholders.py` s placeholdery. Výpis 3.22 znázorňuje deklarační část architektury s některými speciálními komentáři a výpis 3.23 znázorňuje konstanty s těmito komentáři. Z důvodu přehlednosti jsou uvedené výpisy nekompletní — kompletní soubory lze nalézt ve zmíněném adresáři.

▶ Poznámka 3.13 (Volání třídy `VHDLNeuralNetwork` zvenku). Dle popisu v sekci 3.4.12 by se do souboru `nntovhdl/vhdl_designs/__init__.py` měla přidat řádka s importováním této třídy, aby byla přístupná zvenku. V tomto případě to ale není potřeba, protože se tato třída zvenku nevolá. Je volána při generování top-level designu, který zapouzdřuje třída `VHDLNeuralNetworkTop`.

Konstruktor, který je znázorněn ve výpisu 3.24, volá konstruktor nadřazené třídy a do slovníku `code_to_add`, náležící architektuře designu, přidává vybrané placeholdery s prázdnými poli.

[7] není nutné, aby měl soubor stejný název, jako jeho entita — název vygenerovaného souboru bude již shodný s názvem entity

```
architecture arch of neural_network is

    -- <LAYER_COMPONENTS_DECLARATION>

    -- <LAYER_SIGNALS_DECLARATION>

begin
```

Výpis kódu 3.22 Příklad struktury z šablony třídy `VHDLNeuralNetwork`

```
LAYER_COMPONENTS_DECLARATION = "-- <LAYER_COMPONENTS_DECLARATION>"
LAYER_SIGNALS_DECLARATION    = "-- <LAYER_SIGNALS_DECLARATION>"
```

Výpis kódu 3.23 Příklad placeholderů k šabloně třídy `VHDLNeuralNetwork`

```
def __init__ ( self ) -> None:
    super().__init__()
    self.architecture.code_to_add[LAYER_COMPONENTS_DECLARATION] = []
    self.architecture.code_to_add[LAYER_SIGNALS_DECLARATION] = []
```

Výpis kódu 3.24 Příklad konstruktoru třídy `VHDLNeuralNetwork`

Dále třída implementuje metodu `generate(model: Model)`, která se od abstraktní metody třídy `VHDLTemplateDesign` liší v tom, že přidává argument s instancí mé třídy `Model`. V následujících sekcích budou stručně popsány jednotlivé části implementace této metody.

3.4.13.1 Načtení package a souvisejícího typu record

Nejdříve se načítá `package` ze souboru `nntovhdl/sources/vhdl/packages/layer_pack.vhd`, který obsahuje `record type_layer` s porty entity vrstvy, uložené v souboru `nntovhdl/sources/vhdl/designs/neuron_npu_layer_rblk.vhd`. Protože se k designu přidává `package`, je potřeba do designu přidat knihovnu `work` a související `use`. Z načtené `package` se načítá zmíněný typ `record`. Další interakce s `record` je ukázána v následujících sekcích. Výše popsaný postup znázorňuje výpis 3.25. Používá se zde konstanta `VHDL_PACKAGES_DIR` ze souboru `nntovhdl/constants.py`.

```
# Load layer record from package
with VHDLFile(VHDL_PACKAGES_DIR + "layer_pack.vhd", "r") as pkg_f:
    pkg = pkg_f.package()
# Add work lib and corresponding use
self.libs.add(VHDLLibrary("work"))
self.uses.add(VHDLUse("work." + pkg.name + ".all"))
# Get layer record
l_record = pkg.records["type_layer"]
```

Výpis kódu 3.25 Interakce s package v metodě `generate()` třídy `VHDLNeuralNetwork`

3.4.13.2 Vytvoření komponent vrstev

Dále je potřeba načíst design zmíněné vrstvy. K tomu slouží instance `blk_ram_layer_manager` třídy `BlkRamLayerManager`, popsané v sekci 3.4.15. V cyklu přes vrstvy modelu se design vrstvy v proměnné `l_design` uloží do proměnné `sub_designs` (v architektuře) a je vytvořena komponenta `l_comp`, která je přidána do slovníku `components`. V závěru je slovník `components` uložen jako pole do slovníku `code_to_add` pod související placeholder. Tento proces znázorňuje výpis 3.26.

```
components: dict[str, VHDLComponent] = {}  # Components to add to architecture
for n, l in enumerate(model, start=0):
    # Get layer specific design and create component from it's entity
    l_design_file_name = blk_ram_layer_manager.get_design(l.activation)
    l_design = VHDLDesign(l_design_file_name)
    l_comp = VHDLComponent(l_design.entity)
    # Add component to architecture and subdesigns dict
    if l_comp.name not in components:
        components[l_comp.name] = l_comp
        self.sub_designs[l_design.name] = l_design
# Add components to architecture
self.architecture.code_to_add[LAYER_COMPONENTS_DECLARATION].extend(
    components.values())
```

Výpis kódu 3.26 Načtení komponent vrstev v metodě `generate()` třídy `VHDLNeuralNetwork`

3.4.13.3 Vytvoření instancí komponent vrstev

K vytvořeným komponentám v předešlé sekci je potřeba vytvořit jejich instance, což znázorňuje výpis 3.27. V cyklu přes vrstvy modelu se nejdříve vytváří `signal` vrstvy s datovým typem `record` z předešlých sekcí. Poté se vytváří jednotlivé instance komponent, které se ukládají do proměnné `l_comp_inst`. Následně se generické parametry `NUMOFX` a `NUMOFN` této instance nastaví na požadované hodnoty. Poté se namapují porty instance na `l_signal` a v závěru se vytvořená instance přidá do slovníku `code_to_add`, náležící architektuře, pod odpovídající placeholder.

3.4.14 Třída VHDLNeuralNetworkTop

Tato třída je dalším potomkem abstraktní třídy `VHDLTemplateDesign`. Postup pro vytvoření tohoto generovaného designu je podobný, jako u třídy `VHDLNeuralNetwork`, a proto zde nebudou detailně popsány všechny části — pro detailnější náhled si dovolím čtenáře odkázat do kódu či přiložené dokumentace. Šablonu a soubor s placeholdery lze najít v adresáři `nntovhdl/sources/vhdl/templates/neural_network_top`.

V poznámce 3.13 jsem zmiňoval, že řádek s importováním se přidává pouze v této třídě — ve výpisu 3.28 lze vidět příklad importování této třídy.

3.4.15 Třída BlkRamLayerManager

Tato třída zapouzdřuje názvy VHDL souborů s designy vrstev. Její princip je podobný třídě `LayerManager` ze sekce 3.1.2. VHDL soubory vrstev s konkrétní aktivační funkcí jsou ukládány jako hodnoty do slovníku, kde klíči jsou názvy těchto aktivačních funkcí. Pro vyzvednutí

```
for n, l in enumerate(model, start=0):
    # Layer signal declaration
    l_signal = VHDLDataCarrier(name=f"layer{n}", type="signal",
    ↪  data_type=l_record.name)
    # Create layer component instance
    l_comp_inst = VHDLComponentInstance(l_comp, name_prefix="i_",
    ↪  name_suffix=f'_n{l.neurons_count()}w{l.inputs_count()}_L{n}')
    # Map generics
    l_comp_inst.generics["NUMOFX"] = l.inputs_count()
    l_comp_inst.generics["NUMOFN"] = l.neurons_count()
    # Map ports
    for elem in l_record.elements.values():
        l_comp_inst.port_maps[elem.name] = f'{l_signal.name}.{elem.name}'
    # Add layer component instance to architecture
    self.architecture.code_to_\\
        add[LAYER_COMPONENTS_INSTANTIATION_CONNECTION].append(l_comp_inst)
```

Výpis kódu 3.27 Načtení instancí vrstev v metodě `generate()` třídy `VHDLNeuralNetwork`

```
from nntovhdl.vhdl_designs.vhdl_neural_network_top import VHDLNeuralNetworkTop
```

Výpis kódu 3.28 Import třídy `VHDLNeuralNetworkTop` v konstruktoru balíčku `vhdl_designs`

konkrétního souboru s vrstvou tedy stačí na instanci manageru zavolat metodu `get_design(activation)`. Výpis 3.29 znázorňuje instanci tohoto manageru.

```
blk_ram_layer_manager = BlkRamLayerManager (
    {
        "sigmoid": VHDL_DESIGNS_DIR + "neuron_npu_layer_rblk_sigmoid.vhd",
        "relu": VHDL_DESIGNS_DIR + "neuron_npu_layer_rblk_relu.vhd",
        "softmax": VHDL_DESIGNS_DIR + "neuron_npu_layer_rblk_softmax.vhd",
        "tanh": VHDL_DESIGNS_DIR + "neuron_npu_layer_rblk_tanh.vhd",
        "none": VHDL_DESIGNS_DIR + "neuron_npu_layer_rblk.vhd"
    }
)
```

Výpis kódu 3.29 Instance třídy `BlkRamLayerManager`

Kapitola 4
Testování

V této kapitole popisuji způsob testování, který jsem nad aplikací prováděl. Dále uvádím seznam požadavků, které by mohly být předmětem navazující práce na tomto projektu.

Testování jsem rozdělil do dvou částí. V první části jsem testoval navrženou třídu `Model`, která je zodpovědná za transformování Keras modelu. S touto třídou souvisí konfigurace a i export parametrů, takže jsem do testování této třídy zahrnul i tyto části. V druhé části bylo potřeba otestovat, zda se správně generuje VHDL. To spočívalo hlavně v otestování parsování, které se provádí pomocí regulárních výrazů, a poté jsem zkoumal vygenerované designy, zda splňují požadovanou strukturu.

Pro účely testování jsem nepoužíval žádné skripty, ale každou nově implementovanou část jsem prohlížel v debugovacím režimu programu Visual Studio Code, který pro tyto účely poskytuje velice dobré debugovací rozhraní. To má v podstatě všechny funkcionality známé z nástroje `gdb`, akorát v grafickém rozhraní. Skripty jsem nepoužil z toho důvodu, že bych ztratil hodně času jejich implementováním, protože jsem potřeboval ručně prolézat jednotlivé objekty, vytvořené za běhu programu, a dívat se, co se v nich ukládá a jakým způsobem.

4.1 Testování transformovaného modelu

Protože Keras API poskytuje i hotové a naučené modely, nemusel jsem si vytvářet vlastní a funkčnost transformovaného modelu jsem ověřoval na nich. Testoval jsem jak sekvenční modely, tak i funkční. Netestoval jsem ale funkční modely, ve kterých vstupem do jedné vrstvy jsou výstupy z několika předešlých vrstev. Třída `Model` je sice na tyto modely připravena, ale v aktuální verzi nejsou podporovány. Lze tedy pouze používat modely, kde každá vrstva má před a za sebou právě jednu vrstvu. Testování poté spočívalo v tom, že jsem si vybral některý z modelů v Kerasu, k němu jsem vygeneroval instanci své třídy `Model`, spustil si debugovací režim a procházel vrstvu po vrstvě v obou modelech, abych zjistil, zda vše sedí. Cílem bylo, aby v mém modelu nebyla ztracena žádná důležitá informace z Keras modelu. Pochopitelně mi stačilo ukládat méně informací, než obsahuje Keras model, ale bylo důležité, aby byla zachována topologie modelu.

Dále bylo potřeba otestovat konfigurace vytvořené ze tříd `ModelConfig`, `LayerTypeConfig` či `LayerNameConfig`. Zde již porovnání s Keras modelem nebylo relevantní. Protože jsem se z hls4ml inspiroval myšlenkou granularity konfigurace (viz sekce 2.4.1), tak šlo konfigurace porovnávat s těmi, které používá hls4ml. Použil jsem opět metodu krokování a dívání se do vnitřků vytvořených objektů. Vytvoření konfigurace bylo na otestování velmi jednoduché — stačilo kontrolovat vytvořený slovník `data`, který je v mých třídách konfigurací zapouzdřen. O něco složitější bylo ale otestovat, zda se konfigurace správně aplikuje na mojí instanci třídy `Model`. Opět mi velice pomohlo si program pozastavit a podívat se přímo do vnitřku vytvořených instancí.

Následně zbývalo otestovat export parametrů, jejichž typy byly nastaveny prostřednictvím konfigurace. Nejprve bylo potřeba parametry z Keras modelu získat. K tomu mi vedoucí práce poskytl zdrojový kód ze svého projektu, který ukládal `numpy` pole s váhami a biasy z Keras modelu do polí. Zde se nejednalo o nic složitého, takže jsem pouze ověřil, zda tato implementace funguje v mé třídě `Model`. Protože jsem ale používal i specializované datové typy z knihovny `laflib`, tak bylo potřeba implementovat počítání koeficientů škálování a pomocí nic naškálovat váhy i biasy. Tuto implementaci mi také poskytl vedoucí mé práce, takže stačilo opět ověřit, zda to funguje i v mé implementaci. Vše fungovalo, a proto jsem přešel k testování konverze parametrů modelu do datových typů, daných konfigurací. Testoval jsem standardní built-in datové typy jazyka Python a knihovny `numpy`, a poté již zmíněné specializované typy. Testování těch specializovaných bylo o něco složitější, protože bylo potřeba otestovat, zda se správně volají metody z C++, ve kterém je knihovna `laflib` implementována. Jakmile jsem odladil volání C++ kódu, prozkoumal jsem konvertované hodnoty parametrů. U těch standardních bylo možné použít rozhraní `numpy` polí, pomocí kterého jsem převedl hodnoty v něm do požadovaného typu a programově v cyklu porovnal s parametry z mé implementace. Knihovna `numpy` ale pochopitelně nepodporuje specializované `laflib` typy, a proto je nešlo testovat stejně, jako standardní typy. Použil jsem tedy experimentální přístup — náhodně jsem vybral menší množinu původních parametrů a uložil si jejich pozice v `numpy` poli, převedl je pomocí metod z `laflib` do specializovaných datových typů a porovnal s těmi, které byly přetypovány v rámci mé implementace v exportu.

4.2 Testování generace VHDL

Kontrolu vygenerovaného VHDL jsem prováděl naprosto odlišně od způsobu, který jsem popsal v předešlé sekci. Od vedoucího práce a jeho kolegy, pana Ing. Pavla Kubalíka, Ph.D., jsem dostal hotové VHDL designy vrstev a top-level entit. Nejdříve jsem dostal designy, které měly být kompatibilní s AXI protokolem. Na nich jsem testoval implementace tříd, které parsují VHDL soubory a zapouzdřují je do specifických tříd. Díky nim se mi z velké části povedlo odladit regulární výrazy pro parsování a prokázat, že navržená hiearchie tříd pro zapouzdření VHDL struktur je funkční. Design pro AXI sběrnici jsem ale nedotáhnul do konce — nebyl úplně vhodně navržen, a proto bylo některé části těžké generovat. Naštěstí byly později navrženy další VHDL designy, které již nebyly vázané na AXI a jejich rozhraní bylo daleko čistší a srozumitelnější. Také bylo hodně věcí parametrizováno pomocí generických parametrů, což zjednodušovalo generování. Musel jsem ale přidat například generický parametr pro počet vrstev a velikosti některých portů s typem `std_logic_vector` jsem musel také parametrizovat. Tyto úpravy byly ale minimální. Na této druhé verzi VHDL designů jsem doladil regulární výrazy do stádia, kdy fungují dobře a predikovatelně. Dodělal jsem také třídy, zapouzdřující VHDL struktury. Po dokončení generujících tříd `VHDLNeuralNetwork` a `VHDLNeuralNetworkTop` jsem pomocí nich vygeneroval VHDL designy a šel je konzultovat s panem Kubalíkem. To zaručilo, že byly vygenerované designy správně z logického hlediska. Chyby ale ještě mohly být v syntaxi, které se dají okem lehko přehlédnout, a proto jsem soubory přenesl do Vivada, kde je možné zkontrolovat syntaxi pomocí Tcl příkazu `check_syntax`. Vše bylo v pořádku.

4.3 Navazující práce

Do budoucna by bylo dobré rozšířit aplikaci následujícím způsobem:

1. Rozšířit množinu podporovaných vrstev. V tuto chvíli jsou implementovány pouze dense vrstvy, konvoluční vrstvy, flatten vrstvy a pooling vrstvy.

2. Softwarová i hardwarová podpora modelů, kde do jedné vrstvy mohou vést vstupy z více vrstev.

3. Grafické rozhraní pro každou z částí rozhraní — tvorba modelu, konfigurace parametrů modelu a výběr VHDL designu, který se má generovat, včetně zobrazení topologie modelu v každé této fázi.
4. Automatizace tvorby VHDL šablon a souvisejících Python souborů s placeholdery.
5. Generování testbenchů pro souběžně vygenerované VHDL designy.

Závěr

Cílem této práce bylo navrhnout aplikaci, která ze softwarového modelu neuronové sítě, popsaného v Kerasu, extrahuje topologii a parametry sítě, z nich vygeneruje syntetizovatelné VHDL, skládající se z předložených stavebních bloků sítě, popsaných taktéž ve VHDL, a exportuje parametry sítě v datových typech, určených vytvořenou konfigurací. Požadavkem bylo, aby byla aplikace napsaná v jazyce Python, byla řádně zdokumentovaná a umožňovala budoucí rozšiřování.

Nejdříve jsem musel nastudovat základní teorii neuronových sítí, protože jsem se s nimi v minulosti nesetkal. Zjistil jsem, co je to neuronová síť, jak funguje a z čeho se skládá. Zkoumal jsem dvě možné struktury neuronu a jejich váhy, biasy a také nejčastěji používané aktivační funkce. Nastudoval jsem také několik základních typů vrstev neuronových sítí.

Následně jsem se začal zabývat jazykem Python, ve kterém jsem nikdy žádný program nenapsal. Prozkoumal jsem, jaké rozhraní nabízí pro objektové programování, jak spravuje paměť a používá garbage collector pro její úklid. Seznámil jsem se s metodami, jak správně psát Python kód, jak využívat anotace typů a jak strukturovat projekt do modulů a balíčků.

Po získání základních znalostí o jazyku Python jsem se přesunul k analýze Keras API. Zkoumal jsem dva druhy modelů neuronové sítě, které se v Kerasu dají popsat. Musel jsem zjistit, jak vytvořený model uložit do souboru a načíst ze souboru, jak získat informace o modelu a také jak jsou v něm uloženy parametry.

Poté jsem provedl analýzu existujících řešení, která generují VHDL. V rámci této rešerše jsem zkoumal tři existující řešení. Zanalyzoval jsem, co každé z nich nabízí, ale i nenabízí. Nejvíce jsem zkoumal řešení hls4ml, které bylo mou jedinou inspirací v tvorbě vlastního řešení. Důkladně jsem prostudoval etapy, ve kterých toto řešení převádí Keras model až do Vivado projektu, připraveného pro syntézu a souběžně s tím jsem přemýšlel nad řešením, které by bylo schopné Keras model převést přímo do VHDL bez jakýchkoliv mezistupňů.

Přesunul jsem se k návrhu a implementaci vlastního řešení. Důraz jsem kladl na jednoduchost a rozšiřitelnost řešení. Navrhl jsem architekturu, pomocí které lze jednoduše konfigurovat datové typy parametrů modelu, transformovat Keras model do navržených datových struktur a exportovat konfigurované parametry do slovníkové struktury. Vymyslel jsem také způsob, jak generovat VHDL přímo ze softwarového modelu bez jakýchkoliv mezistupňů, jako tomu je u hls4ml. Pro parsování VHDL souborů jsem využil znalosti regulárních výrazů, díky které se mi povedlo navrhnout sadu regulárních výrazů, které stabilně parsují jednotlivé části VHDL souboru. Jednotlivé VHDL struktury jsem zapouzdřil do tříd a designy, které bylo potřeba generovat, jsem proložil řetězci ve speciálním formátu, za které se nahrazují požadované části VHDL kódu. Díky tomuto rozhraní je možné generovat VHDL designy z vyšší úrovně, která uživatele osvobozuje od ručního parsování souborů a nabízí jednodušší práci s VHDL strukturami. Řešení jsem průběžně testoval pomocí nástrojů, které nabízí editor Visual Studio Code. Aplikaci jsem také zdokumentoval pomocí platformy Sphinx.

Výsledkem této práce je funkční generátor syntetizovatelného VHDL, které je generováno ze softwarového modelu neuronové sítě, popsaného v Kerasu. Generátor umí exportovat parametry v datových typech, které jsou specifikovány vytvořenou konfigurací. Celá aplikace je navržena dle technik objektově orientovaného programování a umožňuje budoucí rozšiřitelnost.

Příloha A

Instalace

Instalace Pythonu a externích knihoven

Celý generátor je implementován a testován na operačním systému Linux, konkrétně na školním serveru Laboratoře Inteligentních Vestavných Systémů, kde je nainstalována distribuce Debian. Všechny knihovny jsou instalovány pomocí správce balíčku `pip`, a proto by neměl být problém spustit generátor na většině Linuxových distribucí. Instalaci na systému Windows jsem netestoval, a proto za tento postup neručím.

Generátor je implementován v Pythonu verze 3.7. Zjistil jsem ale, že na distribucích, vycházejících z Ubuntu, nelze verzi 3.7 stáhnout. Otestoval jsem tedy i novější verze 3.8, 3.9 a 3.11, a vše fungovalo v pořádku. Lze tedy zvolit některou z těchto verzí, ale doporučil bych zvolit nejnižší dostupnou verzi z uvedených. Následující příkazy budu předvádět s verzí 3.7. Pokud byla použita jiná verze, stačí všechna volání `python3.7` nahradit za volání Pythonu s použitou verzí.

Za předpokladu, že systém obsahuje správce balíčku `apt`, lze nainstalovat Python pomocí příkazu:

```
sudo apt install python3.7-venv
```

Doporučuji stáhnout `venv` verzi, která zajišťuje podporu virtuálního prostředí. Python lze pochopitelně instalovat i pomocí jiných správců balíčků, ale uvedu pouze příklad s `apt`.

Veškeré následující kroky budou prováděny ve virtuálním prostředí. Výhoda tohoto je, že veškeré instalace knihoven se provádí na čistou instalaci Pythonu. Eliminují se tedy konflikty se systémovými balíčky a je daleko větší záruka, že všechny následující kroky budou fungovat. Virtuální prostředí lze na jednoduše smazat, čímž se smažou i provedené instalace.

Pro vytvoření virtuálního prostředí je potřeba zavolat následující příkazy:

```
mkdir <nazev_adresare>  # vytvoření adresáře pro virtuální prostředí
cd <nazev_adresare>     # přepnutí do vytvořeného adresáře
python3.7 -m venv .     # vytvoření virtuálního prostředí
source bin/activate     # přepnutí do virtuálního prostředí
deactivate              # volat, až bude potřeba prostředí ukončit
```

Pro instalaci veškerých knihoven doporučuji použít správce balíčků `pip`, ale lze použít i jiné.

Správce `pip` by měl být automaticky nainstalován při instalaci Pythonu, ale hodí se ho upgradovat:

```
pip install --upgrade pip
```

Instalaci, deinstalaci a zjištění aktuální verze libovolné knihovny lze provést příkazy:

```
pip install <nazev_knihovny>                  # instalace knihovny
pip install <nazev_knihovny>==<cislo_verze>   # instalace s konkrétní verzí
pip uninstall <nazev_knihovny>                # deinstalace knihovny
pip show <nazev_knihovny>                     # info o verzi knihovny
```

Dále zde uvedu všechny externí knihovny, které jsou importované v rámci mého kódu. Uvedl jsem je i s použitými verzemi. Verze není potřeba dodržovat kromě knihoven `hls4ml` a `tensorflow`, kde doporučuji stáhnout uvedenou verzi.

- `hls4ml` **verze 0.6.0**,
- `tensorflow` **verze 2.10.0** (obsahuje Keras API),
- `pyparsing` (vyžadován volanými knihovnami),
- `cppyy` verze 3.0.0,
- `jsonpickle` verze 3.0.1,
- `tkinter` verze 8.6 — neinstaluje se přes `pip`, viz dále.

Pokud se při instalaci některé z uvedených knihoven vyskytne chyba ohledně nenalezeného modulu, znamená to, že instalovaná knihovna vyžaduje instalaci knihovny, uvedné v chybové hlášce. Je potřeba doinstalovat i tento modul pomocí zmíněných příkazů pro instalaci z `pip`.

Knihovna `tkinter` se neinstaluje přes `pip`, ale pomocí `apt`:

```
sudo apt install python3.7-tk
```

Spuštění generátoru

Nyní je potřeba se přepnout z adresáře virtuálního prostředí do nadřazeného adresáře a přejít do adresáře s implementací:

```
cd ..                       # přepnutí z adresáře virtuálního prostředí
cd nn-model-to-vhdl-master # přepnutí do adresáře projektu
```

Následně je potřeba zkompilovat knihovnu `laflib`. Stačí se přepnout do adresáře s jejím zdrojovým kódem a pomocí utility `cmake` vytvořit Makefile. Ten zajistí vytvoření knihovny `liflab.so`:

```
cd nntovhdl/sources/cpp/  # přepnutí do adresáře s knihovnou laflib
cmake . -B build          # Výstup CMake do adresáře build
cd build                  # přepnutí do adresáře build
make                      # kompilace knihovny liflab.so
# přepnout zpět do adresáře nn-model-to-vhdl-master/
```

Vše je připraveno. Bližší info k používání je sepsáno v příloze B. Pro spouštění stačí volat příkaz:

```
python example.py
```

▶ Poznámka A.1 (Zdrojový kód na Gitlabu). Celý projekt jsem verzoval v systému Git na školním Gitlabu, takže je možné získat zdrojový kód i tam. Pro stahování verze je klíčová větev `master`. V repozitáři jsou i další větve, ale ty jsem používal při implementaci. Repozitář lze nalézt na adrese `https://gitlab.fit.cvut.cz/medekja5/nn-model-to-vhdl`.

Příloha B

Návod k použití

V příloze A jsem popsal instalaci Pythonu, externích knihoven a první spuštění generátoru. V této příloze budu nejdříve ukazovat jednoduché příklady použití generátoru a poté odkážu na přiloženou dokumentaci, generovanou ve Sphinx, včetně návodu, jak jí vygenerovat.

Základní použití generátoru

Celý generátor je pojat jako běžná Python knihovna až na to, že jej v aktuální verzi nelze instalovat přes `pip`. Pro osobní vyzkoušení slouží soubor `example.py`, který obsahuje příklad volání rozhraní generátoru. V následujících řádcích popíšu jeho jednotlivé části a u některých tříd ukážu alternativní použití.

Generátor je zabalen do balíčku `nntovhdl`, který stačí ve spouštěném souboru importovat pomocí příkazu `import nntovhdl`. Dále jsou v přiloženém souboru vytvořeny dva modelu z Kerasu. Prvním z nich je Iris model, který se nachází v adresáři `iris-model` a je již naučený, takže je pouze načten do proměnné `iris_model`. Druhým modelem je VGG16, který je načten přímo z Kerasu do proměnné `vgg16_model`. Dále následuje volání metod generátoru.

Pro vytvoření konfigurace bez grafického rozhraní lze použít jedno z následujících volání:

```
# Konfigurace nad celým modelem
my_config = nntovhdl.configs.ModelConfig(iris_model, gui=False)

# Konfigurace nad stejnými typy vrstev
my_config = nntovhdl.configs.LayerTypeConfig(iris_model, gui=False)

# Konfigurace každé vrstvy
my_config = nntovhdl.configs.LayerNameConfig(iris_model, gui=False)
```

Dále je možné vytvořenou konfiguraci upravit. V případě vytvoření konfigurace nad stejnými typy vrstev lze například nastavit u všech dense vrstev konkrétní datové typy vah a biasů — k vrstvám se přistupuje přes název jejich třídy:

```
my_config.data["Dense"]["Precision"]["weight"] = nntovhdl.lv20
my_config.data["Dense"]["Precision"]["bias"] = nntovhdl.sv24
```

V případě vytvoření konfigurace nad každou vrstvou lze konfigurovat každou vrstvu zvlášť — k vrstvám se přistupuje přes jejich konkrétní názvy:

```
import numpy as np
my_config.data["dense"]["Precision"]["weight"] = float
my_config.data["dense"]["Precision"]["bias"] = float
my_config.data["dense_1"]["Precision"]["weight"] = np.int32
my_config.data["dense_1"]["Precision"]["bias"] = np.int32
```

Dále lze z upravené konfigurace vytvořit model, ze kterého se bude generovat VHDL:

```
my_model = nntovhdl.Model(iris_model, my_config)
```

a také z něj vygenerovat slovník s parametry, který lze uložit do souboru libovolného formátu, například JSON:

```
import json
params = my_model.get_parameters()
with open("params.json", "w") as f:
    json.dump(params, f, indent=2)
```

Nyní ukážu, jak vygenerovat VHDL z návrhů, které mi poskytl vedoucí práce. Nutno podotknout, že generátor dá na výstup pouze soubory, které se generují — komponenty v designu vrstvy již nejsou součástí generování, a proto je potřeba si je opatřit externě. Také nejsou na výstup poslány použité `package` — ty je možné vyzvednout v adresáři `nntovhdl/sources/vhdl/packages`. V tomto případě je pro signály vrstev použita `package layer_pack.vhd`

V závěru lze z vytvořeného modelu vygenerovat design `VHDLNeuralNetworkTop`:

```
my_model_design = nntovhdl.vhdl_designs.VHDLNeuralNetworkTop()
my_model_design.generate(my_model)
```

a uložit jej, včetně použitých designů, do VHDL souborů:

```
my_model_design.save()
```

Nyní stačí vygenerované soubory společně s ostatními přenést například do Vivada a provést syntézu.

Dokumentace a její vygenerování

Dokumentace byla vytvořena prostřednictvím Sphinx. Pro její náhled je nejdříve potřeba jí nainstalovat, což lze pomocí správce balíčku `pip`, o kterém jsem sepsal důležité informace v Příloze A. Instalaci ve vytvořeném virtuálním prostředí včetně potřebné theme lze provést příkazem:

```
pip install sphinx
pip install sphinx_rtd_theme
```

Nyní stačí přejít do adresáře `docs`, ve kterém se nacházejí zdrojové kódy pro dokumentaci. Důležitý je Makefile, který vygeneruje html stránky s dokumentací. To lze příkazem:

```
make html
```

Nyní stačí už jen otevřít `docs/_build/html/index.html` v libovolném prohlížeči.

Bibliografie

1. HORNIK, Kurt; STINCHCOMBE, Maxwell; WHITE, Halbert. Multilayer feedforward networks are universal approximators. *Neural Networks* [online]. 1989, roč. 2, č. 5, s. 359–366 [cit. 2023-04-10]. ISSN 0893-6080. Dostupné z DOI: 10.1016/0893-6080(89)90020-8.
2. GLASSNER, Andrew S. *Deep learning: A visual approach*. San Francisco, CA, USA: No Starch Press, Inc., 2021. ISBN 978-1-7185-0073-0.
3. MCCULLOCH, Warren S; PITTS, Walter. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics 5* [online]. 1943, roč. 5, s. 115–133 [cit. 2023-04-08]. Dostupné z DOI: 10.1007/BF02478259.
4. ROSENBLATT, Frank. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review* [online]. 1958, roč. 65 6, s. 386–408 [cit. 2023-04-08]. Dostupné z DOI: 10.1037/h0042519.
5. PATTERSON, Josh; GIBSON, Adam. *Deep Learning: A Practitioner's approach*. Sebastopol, CA, USA: O'Reilly, 2017. ISBN 978-1-491-91425-0.
6. FELDMAN, J.; ROJAS, Raul. *Neural networks: a systematic introduction*. Springer-Verlag Berlin Heidelberg, 1996. ISBN 978-3-642-61068-4. Dostupné také z: https://page.mi.fu-berlin.de/rojas/neural/chapter/K7.pdf.
7. HAYKIN, Simon S. *Neural networks and learning machines*. 3rd. New Jersey, USA: Pearson Education, 2009. ISBN 978-0-13-147139-9.
8. SHARMA, Sagar; SHARMA, Simone; ATHAIYA, Anidhya. Activation functions in neural networks. *International Journal of Engineering Applied Sciences and Technology* [online]. 2020, roč. 4, č. 12, s. 310–316 [cit. 2023-04-10]. ISSN 2455-2143. Dostupné z: https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf.
9. DING, Bin; QIAN, Huimin; ZHOU, Jun. Activation functions and their characteristics in deep neural networks. In: *2018 Chinese Control And Decision Conference (CCDC)* [online]. 2018, s. 1836–1841 [cit. 2023-04-10]. Dostupné z DOI: 10.1109/CCDC.2018.8407425.
10. RASAMOELINA, Andrinandrasana David; ADJAILIA, Fouzia; SINČÁK, Peter. A Review of Activation Function for Artificial Neural Network. In: *2020 IEEE 18th World Symposium on Applied Machine Intelligence and Informatics (SAMI)* [online]. 2020, s. 281–286 [cit. 2023-04-10]. Dostupné z DOI: 10.1109/SAMI48414.2020.9108717.
11. ZHANG, Aston; LIPTON, Zack C.; LI, Mu; SMOLA, Alex J. *Convolutions for images* [online]. 2021. [cit. 2023-04-13]. Dostupné z: https://d2l.ai/chapter_convolutional-neural-networks/conv-layer.html.

12. ZHANG, Aston; LIPTON, Zack C.; LI, Mu; SMOLA, Alex J. *Maximum Pooling and Average Pooling* [online]. 2021. [cit. 2023-05-09]. Dostupné z: https://d2l.ai/chapter_convolutional-neural-networks/pooling.html.
13. *Keras API reference* [online]. 2023. [cit. 2023-04-14]. Dostupné z: https://keras.io/api/.
14. *Keras*. Models API [online]. 2023. [cit. 2023-04-14]. Dostupné z: https://keras.io/api/models/.
15. HAMDAN, Muhammad K; ROVER, Diane T. Vhdl generator for a high performance convolutional neural network fpga-based accelerator. In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 2017, s. 1–6.
16. TAKAMAEDA-YAMAZAKI, Shinya. *NNGen: A Fully-Customizable Hardware Synthesis Compiler for Deep Neural Network* [online]. 2022. Ver. v1.3.3 [cit. 2023-05-07]. Dostupné z: https://github.com/NNgen/nngen.
17. TEAM, FastML. *hls4ml* [online]. 2023. Ver. v0.7.0 [cit. 2023-04-14]. Dostupné z DOI: 10.5281/zenodo.1201549.
18. *abc — Abstract Base Classes* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://docs.python.org/3.7/library/abc.html.
19. *re — Regular expression operations* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://docs.python.org/3.7/library/re.html.
20. LAVRIJSEN, Wim. *cppyy: Automatic Python-C++ bindings* [online]. 2023. [cit. 2023-04-29]. Dostupné z: https://cppyy.readthedocs.io/en/latest/index.html.
21. Safe method to get value of nested dictionary. In: *Stack Overflow* [online]. 2023 [cit. 2023-04-29]. Dostupné z: https://stackoverflow.com/questions/25833613/safe-method-to-get-value-of-nested-dictionary.
22. Building a Text Editor (Example App). In: *Real Python* [online]. 2020 [cit. 2023-05-03]. Dostupné z: https://realpython.com/python-gui-tkinter/#building-a-text-editor-example-app.
23. HARDI ELECTRONICS AB. *VHDL Handbook* [online]. HARDI Electronics AB, 2007 [cit. 2023-05-04]. Dostupné z: https://redirect.cs.umbc.edu/portal/help/VHDL/VHDL-Handbook.pdf.

Obsah přiloženého média

- `nn-model-to-vhdl-master` ... implementace včetně dokumentace a příkladu
  - `nntovhdl` ... implementace aplikace
  - `example.py` ... příklad použití
  - `iris-model` ... Iris model použitý v příkladu
  - `docs` ... dokumentace
  - `LICENSE.md` ... prohlášení v angličtině
- `thesis` ... zdrojový kód práce v LaTeX
  - `img` ... obrázky
  - `text` ... text práce
  - `ctufit-thesis.cls` ... šablona
  - `ctufit-thesis.tex` ... hlavní zdrojový soubor
  - `medekja5-assignment.pdf` ... zadání práce