VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘEVOD MODELŮ MEZI NÁSTROJI STROJOVÉHO UČENÍ PRO MOBILNÍ PLATFORMY

CONVERSION BETWEEN MODELS OF MACHINE LEARNING FRAMEWORKS FOR MOBILE PLATFORMS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **MARTIN PAVELLA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. RADEK KOČÍ, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

145052

Ústav:	Ústav inteligentních systémů (UITS)
Student:	**Pavella Martin**
Program:	Informační technologie
Specializace:	Informační technologie
Název:	**Převod modelů mezi nástroji strojového učení pro mobilní platformy**
Kategorie:	Algoritmy a datové struktury
Akademický rok:	2022/23

Zadání:

1. Seznamte se s nástroji pro strojové učení na mobilních platformách, např. TensorFlow Lite a ONNX Runtime.
2. Nastudujte formáty pro ukládání modelů jednotlivých nástrojů. Zaměřte se na podporované operátory, způsob uložení parametrů a podporu kvantizace.
3. Navrhněte a implementujte konvertor natrénovaných modelů mezi různými formáty. Zaměřte se především na formáty nástrojů ONNX Runtime a TensorFlow Lite.
4. Funkčnost konvertoru ověřte na různých architekturách modelů pro úlohy klasifikace, detekce a segmentace obrazů a modelů pro analýzu akustických dat.
5. Zhodnoťte přínos své práce, diskutujte možná rozšíření, vylepšení a případné nedostatky.

Literatura:

- Open Neural Network Exchange Format, online [https://github.com/onnx/onnx]
- TensorFlow Lite Inference Framework for Mobile, Embedded and Edge Devices, online [https://www.tensorflow.org/lite/guide]
- Goodfellow I., Bengio Y., Courville A.: Deep Learning, 2016, MIT Press, www.deeplearningbook.org

Při obhajobě semestrální části projektu je požadováno:
První dva body zadání.

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

Vedoucí práce:	**Kočí Radek, Ing., Ph.D.**
Vedoucí ústavu:	Hanáček Petr, doc. Dr. Ing.
Datum zadání:	1.11.2022
Termín pro odevzdání:	10.5.2023
Datum schválení:	3.11.2022

Abstrakt

Nástroje pre strojové učenie používajú rôzne formáty pre reprezentáciu a uloženie modelov hlbokých neurónových sietí. Jedným z najpoužívanejších je formát *Open Neural Network Exchange (ONNX)*. Vývoj softwarovej podpory pre hardwarové akcelerátory na vstavaných systémoch je drahý, a *ONNX* je len výnimočne podporovaný. Potrebné ovládače sú typicky implementované iba pre formát *TensorFlow Lite (TFLite)*. Aktuálne možnosti pre konverziu netrénovaných *ONNX* modelov na *TFLite* sú nedostatočné, a produkujú neoptimálne modely. Táto práca sa zameriava na návrh a vývoj priameho konvertoru *ONNX* modelov na *TFLite*, ktorý produkuje čo najoptimálnejšie modely. Výsledný program bol v spolupráci so spoločnosťou NXP overený na reálnych modeloch. Tie po konverzii produkujú identické výstupy a rýchlosť ich inferencie na cieľových platformách je značne vyššia.

Abstract

Machine learning frameworks use various formats to represent and store models of deep neural networks (DNN). One of the most commonly used ones is *Open Neural Network Exchange (ONNX)*. Developing drivers for hardware accelerators on embedded systems is expensive, and *ONNX* is rarely supported. The necessary software support is typically only implemented for the *TensorFlow Lite (TFLite)* DNN model format. Currently, the options for conversion of pre-trained *ONNX* models to *TFLite* are inadequate and produce suboptimal models. This work focuses on designing and developing a direct converter of *ONNX* models to *TFLite*, which produces as optimal models as possible. The resulting program was verified on real models in collaboration with the NXP company. The models produce identical outputs after conversion and their inference speed on target platforms is significantly higher.

Klíčová slova

TFLite, ONNX, TensorFlow Lite, konverzia hlbokých neurónových sietí, konverzia modelov neurónových sietí, flatbuffer, protocol buffer, ONNX na TFLite

Keywords

TFLite, ONNX, TensorFlow Lite, deep neural network conversion, conversion of neural network models, flatbuffer, protocol buffer, ONNX to TFLite

Citace

PAVELLA, Martin. *Převod modelů mezi nástroji strojového učení pro mobilní platformy.* Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Radek Kočí, Ph.D.

Převod modelů mezi nástroji strojového učení pro mobilní platformy

Prohlášení

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Radka Kočího, Ph.D. Ďalšie informácie mi poskytol pán Ing. Róbert Kalmár z firmy NXP. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

.....
Martin Pavella
7. května 2023

Poděkování

Chcel by som sa poďakovať pánovi Ing. Radkovi Kočímu, Ph.D. za pomoc a rady pri vypracovávaní mojej bakalárskej práce. Taktiež ďakujem pánovi Ing. Róbertovi Kalmárovi z firmy NXP, za jeho čas, mnohé odborné rady z praxe a pomoc pri testovaní modelov na cieľových platformách. Veľmi si vážim aj zhrnutie motivácie celej práce, ktoré Ing. Kalmár vypracoval a poskytol ako podklad pre úvod.

Obsah

Kapitola 1

Úvod

V súčasnosti je k dispozícii viacero frameworkov pre tvorbu, trénovanie a inferenciu hlbokých neurónových sietí (*Deep Neural Networks*, skrátene *DNN*). Medzi najpopulárnejšie patria *TensorFlow* a *PyTorch*, za ktorými stoja spoločnosti *Google* a *Meta*. Oba definujú svoj vlastný formát súboru, v ktorom uchovávajú natrénované modely *DNN* a spravidla nepodporujú načítavanie modelov vo formátoch konkurenčných riešení.

Pri nasadzovaní *DNN* modelov na vstavaných a mobilných zariadeniach je primárnym zámerom inferencia. V prípade vstavaných systémov je potrebné počítať s limitmi ich výpočetnej kapacity. *DNN* modely sa vyznačujú relatívne veľkou výpočtovou náročnosťou, aj v prípade samotnej inferencie. Preto je trendom vývoj špecializovaných hardwarových akcelerátorov, tzv. *Neural Processing Unit* (*NPU*), ako napríklad *Neutron*[1] od firmy *NXP*, *NPU* od firmy *VeriSilicon*[2], prípadne produktová rada *Ethos*[3] od spoločnosti *Arm*.

NPU jednotky sú hardwarové akcelerátory, navrhnuté pre zrýchlenie výpočtovo náročných operácií používaných v *DNN*. Typicky sa jedná o tenzorové a maticové operácie ako napríklad násobenie matíc, konvolúcia či aktivačné funkcie. Akceleráciu dosahujú vysokou paralelizáciou, typicky rádovo tisícov *MAC* (*Multiply and Accumulate*) operácií v jednom takte. Avšak pre využitie *NPU* jednotiek je potrebné vyvinúť príslušnú softwarovú podporu v samotných inferenčných enginoch. Vývoj tejto podpory a ovládačov, čo i len pre hlavné enginy je nákladný. Často je dané *NPU* podporované len v jednom z frameworkov a typicky sa jedná o *TensorFlow Lite*. Chýbajúca podpora akcelerátorov v inferenčných enginoch limituje použitie modelov vytvorených a natrénovaných v iných frameworkoch. Konceptuálne sa však stále jedná len o spôsob uloženia výpočtových grafov a ich príslušných parametrov, ktoré reprezentujú dané neurónové siete a ich architektúry. Tieto výpočtové grafy sú často vyjadriteľné v každom z formátov.

Táto práca sa zameriava na prevod modelov z jedného frameworku do druhého. Konkrétne z formátu *ONNX* do formátu *TensorFlow Lite*. Ako zdrojový formát bol zvolený *ONNX* (*.onnx*) pre jeho otvorenosť a prebiehajúci proces štandardizácie. Je zároveň exportným formátom *PyTorchu*. Ako cieľový formát je zvolený formát inferenčného enginu *TensorFlow Lite* (*.tflite*), z dôvodu rozsiahlej podpory hardwarových akcelerátorov.

[1] https://www.nxp.com/company/blog/introducing-the-nxp-eiq-neutron-neural-processing-unit-npu:BL-INTRODUCING-THE-NXP-EIQ-NPU

[2] https://www.verisilicon.com/en/IPPortfolio/VivanteNPUIP

[3] https://www.arm.com/products/silicon-ip-cpu?families=ethos%20npus

Existujúce konvertory, napríklad *onnx-tensorflow*[4] či *onnx2tf*[5] sa primárne zameriavajú na prevod modelov z formátu *ONNX* do formátu *TensorFlow*. Pre následný prevod do formátu do formátu *TensorFlow Lite* je potrebné použiť *TensorFlow Lite Converter*, čo je oficiálny nástroj od *Google*, konvertujúci z *TensorFlow* do *TensorFlow Lite*. Táto kaskáda konvertorov prináša rôzne obmedzenia, napríklad na strane podpory kvantizovaných modelov. Zároveň generuje nežiadúce artefakty, ako napríklad rozloženie konvolúcie do hĺbky na štandardné operátory konvolúcie. To je síce matematicky správne, ale je to nevhodné z pohľadu generovania optimálneho exekučného plánu na *NPU*. Cieľom tejto práce je vyvinúť prototyp konvertoru, ktorý bude vykonávať priamu konverziu formátu *.onnx* na *.tflite*.

Podklady pre úvod mi poskytol Ing. Róbert Kalmár z firmy NXP. Jedná sa o jeho osobné postrehy a skúsenosti z praxe. Všetky doposiaľ uvedené informácie som prevzal výhradne z jeho rád a z žiadneho iného zdroju.

Samotný text práce je rozčlenený do nasledujúcich kapitol. V kapitole 2 sú opísané štandardy *ONNX* a TensorFlow Lite, ako aj formáty súborov s ich modelmi. V kapitole 3 sú definované významy niektorých často používaných *ONNX* operátorov a do detailu sú analyzované možnosti ich konverzie na *TFLite*. Kapitola 4 obsahuje jednoduchú ilustráciu konverzie časti modelu *Alexnet*. Slúži pre overenie navrhnutého postupu konverzie modelov a pre ucelenie problematiky. Kapitola 5 poskytuje zjednodušený náhľad do implementácie samotného konvertoru. V kapitole 6 sú zhodnotené dosiahnuté výsledky. Pôvodné *ONNX* modely sú porovnané s ich konvertovanými *TFLite* variantami pomocou rôznych experimentov.

[4] https://github.com/onnx/onnx-tensorflow
[5] https://github.com/PINTO0309/onnx2tf

Kapitola 2

Formáty modelov neurónových sietí

Existuje viacero bežne používaných formátov pre reprezentáciu a následné uloženie modelov neurónových sietí. Táto práca sa zameriava výhradne na *.tflite*, teda formát nástroja *TensorFlow Lite* (sekcia 2.2) a na formát štandardu *ONNX*, *.onnx* (sekcia 2.3). V sekcii 2.1 sú vysvetlené základné pojmy, ktoré sú podstatné pre nasledujúce sekcie a kapitoly.

2.1 Základné pojmy

Modely neurónových sietí prevažne pracujú s dátami v podobe tzv. tenzorov (sekcia 2.1.1). Formáty *ONNX* a *TFLite* používajú rozdielny spôsob ukladania tenzorov. To má značný vplyv na konverziu modelov, a niekedy spôsobuje nutnosť pridať pomocné operátory.

V sekcii 2.1.2 je opísaný princíp a využitie kvantizácie tenzorov. Tá sa často využíva v modeloch hlbokých neurónových sietí, keďže znižuje veľkosť modelov a zvyšuje rýchlosť ich inferencie. V sekciách 2.2 a 2.3 je okrem iného opísané, ako jednotlivé formáty reprezentujú kvantizované tenzory.

2.1.1 Tenzor

Pri práci s neurónovými sieťami je často potrebné použiť maticu, s viac než dvomi dimenziami. Vo všeobecnosti sa takéto usporiadanie čísel do pravidelnej mriežky s ľubovoľným počtom dimenzií nazýva tenzor. Často sa značí veľkými písmenami, napríklad $\mathbf{A}$. Prvok tenzoru $\mathbf{A}$ s koordinátami (i, j, k) sa značí $\mathbf{A}_{i,j,k}$. [3, s. 31]

Viacdimenzionálne tenzory sú však spravidla uložené v pamäti ako obyčajné 1D pole. Existuje viacero spôsobov, ako tenzor serializovať. Medzi najčastejšie používané patria tzv. formáty

- *NCHW*
- *NHWC*

kde

- N značí veľkosť dávky dát
- H je výška tenzoru

- W je šírka tenzoru
- C udáva počet kanálov.

Veľkými písmenami sú značené veľkosti jednotlivých dimenzií (napríklad N). Malými písmenami sú reprezentované indexy do príslušných dimenzií (napríklad n). Spomínané formáty definujú spôsob určenia pozície prvku na indexoch (n, c, h, w) resp. (n, h, w, c) v serializovanom 1D poli. [7]

V prípade formátu *NCHW*, je W najviac vnorená dimenzia. To znamená že susedné prvky v pamäti zdieľajú indexy n, c a h a ich index w sa líši o 1. Toto samozrejme platí iba pokiaľ prvky nie sú okrajové. Naopak, N je najviac vonkajšia dimenzia. Takže pre prístup k prvku na indexoch (c, h, w) ale na susedných tenzoroch, teda s indexom n rozdielnym o 1, je nutné skočiť na pozíciu v pamäti o $C \cdot H \cdot W$ ďalej. Postup výpočtu indexu prvku *NCHW* tenzoru v pamäti je v tabuľke 2.1. [7]

Pri formáte *NHWC* je najviac zanorenou dimenziou C, teda kanály. V prípade veľkosti dávky $N = 1$, je tento formát veľmi podobný formátu súboru BMP (bitmap), kde je obrázok uložený po pixeloch a každý pixel obsahuje informácie o svojich farbách, teda kanáloch. Postup výpočtu indexu prvku *NHWC* tenzoru v pamäti je v tabuľke 2.1. [7]

Formát tenzoru	Výpočet indexu prvku
NCHW	$offset\_nchw(n, c, h, w) = n \cdot CHW + c \cdot HW + h \cdot W + w$
NHWC	$offset\_nhwc(n, h, w, c) = n \cdot HWC + h \cdot WC + w \cdot C + c$

Tabulka 2.1: Matematické výrazy pre výpočet indexu prvku tenzoru v pamäti pre jednotlivé formáty. Obsah tabuľky bol prevzatý z [7].

2.1.2 Kvantizácia

Kvantizácia je jedným zo spôsobov redukovania veľkosti modelov neurónových sietí a zvyšovania rýchlosti ich inferencie. To všetko pri minimálnej strate presnosti výpočtu. Kvantizácia je teda proces mapovania celých čísel q na reálne čísla r. Značenie q je odvodené od quantized value (kvantizovaná hodnota) a značenie r znamená real value (reálna hodnota). Toto mapovanie je realizované matematickým výrazom 2.1

$$r = S(q - Z) \tag{2.1}$$

kde S a Z sú vhodné konštanty, resp. kvantizačné parametre. Pri 8-bitovej kvantizácii je q kvantizované ako 8-bitová celočíselná hodnota. Všeobecne pri B-bitovej kvantizácii je q kvantizované ako B-bitová celočíselná hodnota. Niektoré tenzory, typicky vektory *bias*, sú kvantizované na 32 bitov. Konštanta S znamená scale (škála). Je to ľubovoľné pozitívne reálne číslo. Parameter Z značí zero point (nulový bod). Je rovnakého dátového typu ako q a konkrétne reprezentuje kvantizovanú hodnotu q, ktorá prináleží reálnej hodnote 0. [9]

Použitím spomínaného mapovania je možné realizovať výpočty s reálnymi číslami pomocou celočíselnej aritmetiky. Napríklad násobenie dvoch štvorcových matíc r_1 a r_2 s reálnymi číslami o rozmeroch $N \times N$. Výsledok operácie je matica $r_3 = r_1 r_2$. Prvky matíc r_α ($\alpha =$ 1, 2 alebo 3) sa značia $r_\alpha^{(i,j)}$ pre $1 \leq i, j \leq N$. Kvantizačné parametre sú označené ako S_α a Z_α. Kvantizované prvky sa značia ako $q_\alpha^{(i,j)}$. Z rovnice 2.1 sa teda stane

$$r_\alpha^{(i,j)} = S_\alpha(q_\alpha^{(i,j)} - Z_\alpha). \tag{2.2}$$

Následne z definície násobenia matíc získame

$$S_3(q_3^{(i,k)} - Z_3) = \sum_{j=1}^{N} S_1(q_1^{(i,j)} - Z_1)S_2(q_2^{(j,k)} - Z_2), \tag{2.3}$$

čo je možné preformulovať na

$$q_3^{(i,k)} = Z_3 + M \sum_{j=1}^{N} (q_1^{(i,j)} - Z_1)(q_2^{(j,k)} - Z_2), \tag{2.4}$$

kde parameter M je daný ako

$$M = \frac{S_1 S_2}{S_3}. \tag{2.5}$$

V rovnici 2.4 je jediným neceločíselným činiteľom M. Ako konštanta závislá iba na kvantizačných parametroch S_1, S_2 a S_3 môže byť staticky predpočítaná. Následne za určitých okolností môže byť násobenie konštantou M aproximované násobením konštantou M_0 s pevnou rádovou čiarkou a bitovým posunom, pokiaľ $M = 2^{-n} M_0$. [9]

Podobným spôsobom je možné reprezentovať rôzne operácie s plávajúcou rádovou čiarkou za využitia výhradne celočíselnej aritmetiky. Dôsledkom tohto postupu je výrazné zníženie veľkosti modelov. Niekedy až štvornásobné. Taktiež efektivita inferencie je zlepšená. Tieto výsledky naznačujú, že inferencia založená na celočíselnej aritmetike môže byť kľúčová pri implementácii technológií rozpoznávania obrazu v reálnom čase na menej výkonných mobilných zariadeniach. [9]

2.2 TensorFlow Lite

TensorFlow Lite (alebo *TFLite*) je sada nástrojov, ktorá umožňuje realizovať strojové učenie na mobilných a vstavaných zariadeniach. Je odvodená od štandardu *TensorFlow* a je optimalizovaná na inferenciu na koncovom zariadení, tým že rieši niektoré obmedzujúce problémy. Inferencia modelu sa odohráva priamo na zariadení a nie na vzdialenom serveri. Ušetrí sa teda doba odozvy a nie je nutné aby zariadenie bolo neustále pripojené k internetu. Taktiež keďže dáta neopúšťajú zariadenie, znižuje sa riziko narušenia súkromia. Formát *.tflite* využíva technológiu *flatbuffer*, ktorá znižuje veľkosť súboru s modelom siete. V neposlednom rade sú inferenčné programy optimalizované pre zníženie spotreby zariadenia. Tento formát podporuje množstvo rozličných platforiem, ako napríklad *Android*, *iOS*, vstavaný *Linux* či rôzne mikrokontrolery. S *TFLite* modelmi je možné pracovať v jazykoch ako *Java*, *C++* alebo *Python*. [16]

Ako už bolo spomenuté, *TFLite* využíva formát súboru *flatbuffer*. Jedná sa o efektívny, prenosný serializačný formát. Ponúka mnohé výhody oproti formátu *protocol buffer*, ako napríklad zníženú veľkosť súboru alebo jednoduchý prístup k dátam. Môže sa k nim totiž pristupovať priamo, bez nutnosti dáta rozbaľovať alebo nejako prekladať. Spolu s ďalšími vlastnosťami majú za následok zvýšenie rýchlosti inferencie, ktorá je podstatnou najmä na zariadeniach s obmedzenými výpočtovými a pamäťovými zdrojmi. [16]

Formát *flatbuffer* je binárny, takže pohľad na surové dáta v tomto formáte nie je užitočný. Analýza obsahu *flatbuffer* súborov je možná po prvotnej konverzii na serializačný formát JSON. Ten je textový a jednotlivé prvky súboru sú v ňom jasne viditeľné. Existuje nástroj *flatc*, ktorý je schopný takúto konverziu realizovať. Na Linuxe je nástroj spustený príkazom

```
flatc -t --strict-json --defaults-json -o . <schema> -- <model> --raw-binary
```

kde <schema> je schémový súbor popisujúci štruktúru *flatbuffer* súboru. V prípade konverzie *TFLite* modelov na *JSON* sa použije *schema.fbs*[1]. <model> je *flatbuffer* súbor, ktorý bude preložený do formátu *JSON*. V tomto prípade sa jedná o *.tflite* model. [4]

TensorFlow Lite podporuje mnohé tzv. *delegáty*, ktoré umožňujú hardwarovú akceleráciu inferencie modelov. Robia tak využitím hardwarových prvkov zariadenia, ako napríklad grafickej karty alebo procesoru digitálnych dát. Každý *delegát* je optimalizovaný pre konkrétne platformy a pre konkrétne druhy modelov. Pri modelovom kritériu záleží, či je model kvantizovaný alebo nie. Prípadne záleží aj na type kvantizácie. *Delegáty* zvyčajne realizujú výpočty pri inej presnosti než keby je inferencia vykonávaná na procesore. Dôsledkom toho je malá odchýlka medzi výstupmi pri ich využití. Niekedy však môže byť presnosť dokonca vyššia. Napríklad grafické karty niekedy využívajú aritmetiku s plávajúcou rádovou čiarkou aj pri kvantizovaných modeloch, čo môže zvýšiť presnosť výpočtu. [15]

Dôležitou vlastnosťou *TFLite* modelov je formát tenzorov, ktoré sa v ňom vyskytujú. Podľa informácií uvedených v súbore so schémou *.tflite* súboru[1], všetky vstavané *TFLite* operátory pracujú s formátom *NHWC*, resp. *NHWDC* pri viac dimenzionálnych tenzoroch.

Formát súboru *.tflite*

TFLite využíva pre uloženie modelov a natrénovaných hodnôt formát súboru *.tflite* [16]. Jedná sa o serializačný formát *flatbuffer*, a štruktúra uložených dát je dostupná v oficiálnom súbore *schema.fbs*[1]. Všetky informácie v tejto sekcii som odvodil z metadát z tohto súboru, pokiaľ nie je uvedené inak.

V nasledujúcom texte je používaná terminológia spojenú s *flatbuffer* formátom. Objekty ktoré obsahujú atomické hodnoty a ďalšie objekty sa nazývajú *tabuľky*. Jedná sa o analógiu *štruktúr* formátu *JSON*. Ďalším typom objektu je *únia*, ktorá je podobná výčtovému typu *enum*. Jej hodnoty sú však *tabuľky* a objekt typu *únia* smie obsahovať práve jednu hodnotu naraz. Posledným typom, ktorý táto sekcia používa sú *vektory*. Jedná sa o analógiu *kolekcií* formátu *JSON*. [4]

```
table Model {
  version:uint;
  operator_codes:[OperatorCode];
  subgraphs:[SubGraph];
  description:string;
  buffers:[Buffer];
  metadata_buffer:[int];
  metadata:[Metadata];
  signature_defs:[SignatureDef];
}
```

Obrázek 2.1: Definícia tabuľky *Model* v súbore *schema.fbs*.

Schéma popisuje štruktúru *.tflite* súboru nasledovne. Koreňovým prvkom je práve jedna tabuľka *Model* (obrázok 2.1), ktorá obsahuje vektor `operator_codes` s popisom všetkých operátorov, ktoré sa v modeli vyskytujú. Ďalej obsahuje vektor `subgraphs` s tabuľkami reprezentujúcimi jednotlivé výpočtové grafy modelu. Model taktiež obsahuje vektor `buffers` ktorý pozostáva zo surových dát tenzorov, ktoré sa v modeli objavujú. Sú uložené po jednotlivých bytoch vo formáte *little endian*. *Buffer* na pozícii 0 by mal byť vždy prázdny,

[1] Dostupné z https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/schema/schema.fbs

pretože predstavuje výstup celého modelu. Okrem týchto troch vektorov sú v modeli ešte ďalšie, menej dôležité informácie.

```
table SubGraph {
  tensors:[Tensor];
  inputs:[int];
  outputs:[int];
  operators:[Operator];
  name:string;
}
```

Obrázek 2.2: Definícia tabuľky *SubGraph* v súbore *schema.fbs*, ktorá opisuje výpočtový graf reprezentujúci model neurónovej siete.

Tabuľka *SubGraph* reprezentuje výpočtový graf. *Model* ich môže obsahovať viac, ale typicky sa používa len jeden, ktorý reprezentuje celý model neurónovej siete. Každý *SubGraph* obsahuje vektor operátorov, ktoré reprezentujú rôzne, často matematické operácie. Musia byť uložené v poradí, v akom budú vyhodnocované. *SubGraph* taktiež vo vektore `tensors` uchováva definície tenzorov, ktoré jeho operátory používajú. Každý tenzor obsahuje index do globálneho vektoru `buffers`, kde sú uložené jeho dáta. Dôležitou súčasťou grafu je taktiež špecifikácia jeho vstupných a výstupných tenzorov. Na tento účel sa vo vektoroch `inputs` a `outputs` používajú indexy do vektoru `tensors`. Definícia tabuľky *SubGraph* je na obrázku 2.2.

```
table QuantizationParameters {
  min:[float];
  max:[float];
  scale:[float];
  zero_point:[long];
  details:QuantizationDetails;
  quantized_dimension:int;
}
```

Obrázek 2.3: Definícia tabuľky *QuantizationParameters* v súbore *schema.fbs*, ktorá definuje použitý spôsob kvantizácie hodnôt jednotlivých tenzorov.

Kvantizácia je zabezpečená tabuľkou *QuantizationParameters*, ktorú v sebe voliteľne môžu definovať jednotlivé tenzory. Samotná tabuľka je zobrazená na obrázku 2.3. Atribúty `min` a `max` definujú rozsah pôvodných hodnôt s plávajúcou rádovou čiarkou pre jednotlivé dimenzie tenzoru. Atribúty `scale` a `zero_point` sú štandardné kvantizačné parametre, ktoré umožňujú výpočet pôvodných hodnôt z kvantizovaných. *TFLite* nedefinuje žiadne špecializované operátory pre prácu s kvantizovanými dátami. Operátory implicitne podporujú tenzory s plávajúcou rádovou čiarkou, ako aj kvantizované tenzory.

```
table Operator {
  opcode_index:uint;
  inputs:[int];
  outputs:[int];
  builtin_options:BuiltinOptions;
  custom_options:[ubyte];
  custom_options_format:CustomOptionsFormat;
  mutating_variable_inputs:[bool];
  intermediates:[int];
}
```

Obrázek 2.4: Definícia tabuľky *Operator* v súbore *schema.fbs*.

Operátory sú definované tabuľkou na obrázku 2.4. Majú vektory vstupov a výstupov, ktoré slúžia ako indexy do vektoru `tensors` v nadradenom *SubGraph* objekte. Každý operátor taktiež obsahuje atribút `opcode_index`, teda index do vektoru `operator_codes` celého modelu. Tento index určuje o aký operátor sa jedná. Celkovo *TFLite* definuje až 158 rôznych operátorov. Jednou z najpodstatnejších položiek tabuľky *Operator* je `builtin_options`. Jedná sa o objekt *únie BuiltinOptions*, ktorý definuje parametre ovplyvňujúce chovanie uloženého operátoru. Väčšina operátorov má svoj vlastný *BuiltinOptions*, niektoré ho však zdieľajú. Napríklad operátory `MaxPool2D` a `AveragePool2D` oba používajú *Pool2DOptions*. *TFLite* schéma definuje 123 rôznych variánt *BuiltinOptions*. Ako príklad je uvedený často používaný operátor `Conv2D`. Tento operátor využíva parametre definované v *Conv2DOptions* (obrázok 2.5). Konkrétne sa jedná o dva celočíselné *dilatačné* faktory, dva atribúty `stride_h` a `stride_w`, objekt typu *Padding* definovaný v samostatnej tabuľke a posledný je atribút `fused_activation_function`, ktorý umožňuje spojiť operátor s aktivačnou funkciou. Operátor *Conv2D* a význam jeho parametrov sú opísané v sekcii 3.1.2.

```
table Conv2DOptions {
  padding:Padding;
  stride_w:int;
  stride_h:int;
  fused_activation_function:ActivationFunctionType;
  dilation_w_factor:int = 1;
  dilation_h_factor:int = 1;
}
```

Obrázek 2.5: Definícia tabuľky *Conv2DOptions* v súbore *schema.fbs*, ktorá určuje parametre *TFLite* operátoru `Conv2D`.

Sémantika jednotlivých operátorov a ich parametrov nie je v schéme uvedená. Súbor *schema.fbs* iba definuje syntax *.tflite* súboru. Zisťovanie ich významu je relatívne komplikované. Ani po diskusii s pánom Ing. Róbertom Kalmárom z firmy NXP sme nenašli nijakú dokumentáciu k *TFLite* operátorom. Jediným zdrojom informácií v tejto oblasti boli súbory s implementáciou inferencie jednotlivých operátorov. Sú verejne dostupné na oficiálnej *GitHub* stránke *TensorFlow Lite*[2].

2.3 ONNX

Open Neural Network Exchange alebo *ONNX* je otvorený ekosystém. ktorý umožňuje vývojárom umelých inteligencií používať rôzne nástroje pri práci na jednom projekte. *ONNX* poskytuje otvorený formát modelov umelej inteligencie. Definuje model založený na rozšíriteľnom výpočtovom grafe, veľké množstvo vstavaných operátorov a štandardné dátové typy. *ONNX* má širokú podporu medzi rôznymi nástrojmi, frameworkami a hardvérmi. Umožňuje teda interoperabilitu medzi nástrojmi, čím zvyšuje rýchlosť vývoja umelej inteligencie. [13]

Dôležitou vlastnosťou *.onnx* modelov sú formáty tenzorov, ktoré sa v nich vyskytujú. Narozdiel od *TFLite* som nenašiel všeobecné pravidlo pre všetky operátory. V oficiálnej dokumentácii je zvyčajne pri jednotlivých operátoroch uvedený formát vstupných tenzorov, ktorý operátor očakáva. V *ONNX* modeloch, ktorými sa táto práca zaoberá sú použité výhradne tenzory vo formáte *NCHW*, pokiaľ majú 4 a viac dimenzií.

[2]Dostupné z : https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/kernels/internal/reference

ONNX modely sú uložené v serializačnom formáte *protocol buffer*. Ten nie je textový a pohľad na jeho surové dáta nie je užitočný. Analýza obsahu *protocol buffer* súborov je možná po predošlej konverzii na formát *JSON*. Ten už textový je a jednotlivé prvky modelu sú v ňom jasne viditeľné. Tento preklad je možné realizovať pomocou oficiálnej python knižnice `protobuf`. Implementoval som krátky skript v súbore *test/protobufToJson.py*, ktorý preloží vstupný *protocol buffer* na formát *JSON*.

Formát súboru *.onnx*

ONNX využíva pre uloženie modelov a natrénovaných hodnôt formát súboru *.onnx*. Jedná sa o nadstavbu nad serializačným formátom *protocol buffer* so štruktúrou uložených dát popísanou v súboroch

- `onnx-ml.proto`
- `onnx-data.proto`
- `onnx-operators-ml.proto`

dostupných na *GitHub* stránke *ONNX*[3]. Tieto súbory teda popisujú spôsob ukladania a načítavania dát vo formáte *protocol buffer* Hlavný z nich je `onnx-ml.proto`. Definuje serializáciu samotného modelu, jeho výpočtového grafu, operátorov a tenzorov. Narozdiel od *.tflite* je *.onnx* súbor značne komplikovanejší a umožňuje reprezentáciu komplexnejších dátových štruktúr, než sú tenzory. Súbor definuje dátové štruktúry ako napríklad *Map*, *Sequence* alebo *Opaque*. Tie spadajú mimo rozsah tejto práce, nakoľko ich konverzia do *.tflite* nie je možná, pretože *TFLite* definuje iba tenzory a riedke tenzory.

V nasledujúcom texte je používaná terminológia spojenú s formátom *protocol buffer*. Ten definuje objekty pozostávajúce z atomických hodnôt a ďalších objektov ako *message*. Položky každého *message* objektu musia mať jedno z definovaných obmedzení. Pre účely tejto sekcie sú podstatné obmedzenia *optional*, ktoré stanovuje že položka môže a nemusí mať definovanú hodnotu, a obmedzenie *repeated*, čo znamená že položka nadobúda ľubovoľný počet prípustných hodnôt. Jedná sa teda o pole hodnôt.

```
message GraphProto {
  repeated NodeProto node = 1;
  optional string name = 2;
  repeated TensorProto initializer = 5;
  repeated SparseTensorProto sparse_initializer = 15;
  optional string doc_string = 10;
  repeated ValueInfoProto input = 11;
  repeated ValueInfoProto output = 12;
  repeated ValueInfoProto value_info = 13;
  repeated TensorAnnotation quantization_annotation = 14;
  reserved 3, 4, 6 to 9;
  reserved "ir_version", "producer_version", "producer_tag", "domain";
}
```

Obrázek 2.6: Definícia *message GraphProto*, teda výpočtového grafu v *.onnx* súbore.

Základ *.onnx* súboru pozostáva z *message ModelProto*. Ten okrem iného obsahuje práve jeden *message GraphProto*, ktorý reprezentuje acyklický výpočtový graf modelu. Jeho definícia je na obrázku 2.6. V rámci neho sú uložené uzly grafu ako *message NodeProto*, ktoré reprezentujú jednotlivé operátory. Ďalej graf obsahuje informácie o svojich tenzoroch, ktoré

[3]Dostupné z https://github.com/onnx/onnx/tree/main/onnx

sú rozdelené do dvoch *message* typov. *message ValueInfo* popisuje iba názov tenzoru, jeho tvar a dátový typ prvkov v ňom. Týmto spôsobom musia byť definované vstupy a výstupy celého grafu, pomocou *repeated* položiek `input` a `output`. Taktiež takto môžu byť definované vstupy a výstupy jednotlivých operátorov v grafe. Teda medzivýsledky, nedostupné mimo výpočtového grafu. Na tento účel slúži *repeated* položka `value_info`. Definícia týchto medzivýsledkov nie je povinná. Model ich môže definovať neúplne, napríklad neuvedie tvar tenzoru, alebo je ich definícia často vynechaná úplne.

```
message TensorProto {
  repeated int64 dims = 1;
  optional int32 data_type = 2;
  message Segment {
    optional int64 begin = 1;
    optional int64 end = 2;
  }
  optional Segment segment = 3;
  repeated float float_data = 4 [packed = true];
  repeated int32 int32_data = 5 [packed = true];
  repeated bytes string_data = 6;
  repeated int64 int64_data = 7 [packed = true];
  optional string name = 8;
  optional string doc_string = 12;
  optional bytes raw_data = 9;
  repeated StringStringEntryProto external_data = 13;
  enum DataLocation {
    DEFAULT = 0;
    EXTERNAL = 1;
  }
  optional DataLocation data_location = 14;
  repeated double double_data = 10 [packed = true];
  repeated uint64 uint64_data = 11 [packed = true];
}
```

Obrázek 2.7: Zjednodušená definícia *message TensorProto*, reprezentujúceho tenzor so statickými dátami v *.onnx* súbore.

Na druhej strane, tenzory ktoré obsahujú aj natrénované dáta, musia byť reprezentované pomocou *message TensorProto*, v opakovanom atribúte `initializer`. *Message TensorProto* je relatívne komplexný, preto len jeho zjednodušená verzia je na obrázku 2.7. Podobne ako *ValueInfo*, taktiež obsahuje názov, tvar a dátový typ tenzoru. Dopĺňa ich však o statické dáta. Tie môžu byť uložené práve jedným z celkových ôsmych spôsobov:

- pole `float_data` pre dáta typu *FLOAT* alebo *COMPLEX64*
- pole `double_data` pre dáta typu *DOUBLE* alebo *COMPLEX128*
- pole `int32_data` pre dáta typu *INT32*, *INT16*, *INT8*, *UINT16*, *UINT8*, *BOOL*, *FLOAT16* alebo *BFLOAT16*
- pole `int64_data` pre dáta typu *INT64*
- pole `uint64_data` pre dáta typu *UINT32* alebo *UINT64*
- pole `string_data` pre dáta typu *STRING* v kódovaní UTF-8
- pole `raw_data` pre ľubovoľný typ okrem *STRING*
- `external_data` pre dáta uložené v externom súbore

V prípade využitia poľa *raw_data* musí mať každý prvok konštantnú veľkosť a musí používať poradie bytov *little-endian*. Toto pole totiž ukladá dáta po bytoch, presne tak isto ako v prípade *TFLite*. Výhodou využitia jedného zo špecializovaných polí je, že v niektorých prípadoch dokáže *protocol buffer* skrátiť dĺžku niektorých prvkov a ušetriť tak celkovú veľkosť súboru. Toto je možné napríklad pri použití poľa s celočíselnými hodnotami.

ONNX pracuje s kvantizovanými modelmi dvomi spôsobmi. Prvým je použitie špecializovaných operátorov, ktoré sú schopné priamo pracovať s kvantizovanými tenzormi. Pre použitie ostatných operátorov v kvantizovaných modeloch je nutné aplikovať druhú možnosť. *Quantize and DeQuantize* (*QDQ*) je metodika vkladania operátorov `QuantizeLinear` a `DeQuantizeLinear` pred a za štandardné *ONNX* operátory. Tie dokážu prepočítať statické aj vypočítané kvantizované tenzory na tenzory s plávajúcou rádovou čiarkou. [11]

```
message StringStringEntryProto {
  optional string key = 1;
  optional string value = 2;
};

message TensorAnnotation {
  optional string tensor_name = 1;
  repeated StringStringEntryProto quant_parameter_tensor_names = 2;
}
```

Obrázek 2.8: Definícia *message TensorAnnotation* a *message StringStringEntryProto*, pomocou ktorých je reprezentovaná kvantizácia tenzorov v *.onnx* súbore.

Špecifikácia kvantizovaných tenzorov je zabezpečená pomocou *message TensorAnnotation* a *message StringStringEntryProto*, ktoré sú zobrazené na obrázku 2.8. *TensorAnnotation* sa viaže na ľubovoľný tenzor pomocou jeho názvu v položke `tensor_name`. Pomocou *repeated* položky `quant_parameter_tensor_names` k tenzoru priradí mapovania preddefinovaných kľúčových slov na kvantizačné parametre. Napríklad kľúčové slovo `SCALE_TENSOR` značí škálovací kvantizačný parameter a `ZERO_POINT_TENSOR` značí nulový bod kvantizácie.

```
message NodeProto {
  repeated string input = 1;
  repeated string output = 2;
  optional string name = 3;
  optional string op_type = 4;
  optional string domain = 7;
  repeated AttributeProto attribute = 5;
  optional string doc_string = 6;
}
```

Obrázek 2.9: Definícia *message NodeProto*, reprezentujúceho uzol výpočtového grafu v *.onnx* súbore.

Jednotlivé operátory sú v *.onnx* súbore definované pomocou *message NodeProto* z obrázku 2.9. Ten reprezentuje jeden uzol výpočtového grafu modelu neurónovej siete. Schéma definuje vstupy a výstupy uzlu pomocou názvov jednotlivých tenzorov v *repeated* položkách `input` a `output`. Konkrétny typ obaľovaného operátoru je identifikovaný jeho názvom v *optional* položke `op_type`. Veľkou odlišnosťou voči *TFLite* reprezentácii výpočtového grafu a jeho operátorov je definícia atribútov samotných operátorov. Kde *TFLite* definoval presnú *flatbuffer table* pre parametre každého operátoru, *ONNX* definuje atribúty všetkých operátorov pomocou *repeated* položky `attribute`, generického typu *AttributeProto*. Zjednodušená verzia komplexnej definície *message AttributeProto* je na obrázku 2.10. Tento prístup

umožňuje špecifikovať atribúty operátorov ako pole objektov identifikovaných svojim názvom, teda položkou `name`. Hodnota tohto atribútu môže nadobúdať jedného z veľkého množstva typov. Od atomických celočíselných či floatových typov, cez štruktúrované typy ako výpočtové grafy či tenzory až po celé kolekcie týchto typov v *repeated* položkách *AttributeProto*. Dokonca vďaka *repeated* položke `type_protos` typu *message TypeProto*, je možné tieto štruktúry neobmedzene do seba vnorovať.

```
message AttributeProto {
  optional string name = 1;
  optional string ref_attr_name = 21;
  optional string doc_string = 13;
  optional AttributeType type = 20;
  optional float f = 2;
  optional int64 i = 3;
  optional bytes s = 4;
  optional TensorProto t = 5;
  optional GraphProto g = 6;
  optional SparseTensorProto sparse_tensor = 22;
  optional TypeProto tp = 14;
  repeated float floats = 7;
  repeated int64 ints = 8;
  repeated bytes strings = 9;
  repeated TensorProto tensors = 10;
  repeated GraphProto graphs = 11;
  repeated SparseTensorProto sparse_tensors = 23;
  repeated TypeProto type_protos = 15;
}
```

Obrázek 2.10: Zjednodušená definícia *message AttributeProto*, reprezentujúceho generický atribút operátorov v *.onnx* súbore.

Všetky informácie ohľadom formátu *.onnx* súboru v tejto sekcii som sám odvodil z metadát v schémovom súbore *onnx-ml.proto*[4], pokiaľ nie je uvedené inak.

2.4 Zhrnutie

Súbor vo formáte *.onnx* má komplexnejšiu štruktúru než *.tflite*. Mnohé aspekty *ONNX* modelu nie je možné reprezentovať *TFLite* modelom. Tieto formáty teda nie sú ekvivalentné a líšia sa svojou vyjadrovacou silou. Súbory s *ONNX* modelmi zriedka kedy využívajú všetky možnosti svojho štandardu. Často používajú iba tenzory a základné dátové typy. V týchto prípadoch je možné *.onnx* súbor vyjadriť ekvivalentným *.tflite* súborom, za predpokladu, že štandard *TFLite* definuje operátory, pomocou ktorých sa dá reprezentovať chovanie operátorov v *ONNX* modeli. Možnosti konverzie niektorých *ONNX* operátorov na *TFLite* sú detailne analyzované v sekcii 3.1. Konkrétny príklad jednoduchých *.tflite* a *.onnx* súborov vizualizovaných vo formáte *json* je v kapitole 4.

[4]Dostupné z https://github.com/onnx/onnx/blob/main/onnx/onnx-ml.proto

Kapitola 3

Možnosti konverzie modelov

V tejto kapitole sú opísané možnosti a postup konverzie *ONNX* modelu na ekvivalentný *TFLite* model. V sekcii 3.1 je analyzovaný spôsob konverzie obmedzenej sady *ONNX* operátorov na *TFLite*. Detaily konverzie sú málokedy triviálne. Pri analýze nových modelov sa objavujú ďalšie hraničné situácie, kedy je nutné operátor konvertovať novým spôsobom. Z tohoto dôvodu nie je uvedený postup konverzie všetkých operátorov. Táto práca sa zameriava najmä na konvolučné siete a operátory, ktoré sa v nich často vyskytujú. Sekcia 3.2 sa venuje konverzii tenzorov *ONNX* modelov, a ich funkcii v ekvivalentných *TFLite* modeloch.

3.1 Operátory a ich konverzia

Pri analýze postupu konverzie operátorov som čerpal informácie z oficiálnej dokumentácie *ONNX*[1] a z referenčnej implementácie *TFLite* operátorov[2]. Na stránkach jednotlivých operátorov *ONNX* dokumentácie je spravidla popísaný význam jednotlivých atribútov a taktiež býva uvedený matematický výraz, ktorý operátor realizuje. *TFLite* v čase písania práce podobnú dokumentáciu nemá. Na GitHub stránke *TFLite* sú verejne dostupné súbory, v ktorých sú jednotlivé operátory implementované v jazyku *C++*. Z týchto súborov som odvodil detaily chovania daných operátorov a významy jednotlivých parametrov. Na základe získaných informácií som navrhol postup konverzie operátorov.

Pre parametre ovplyvňujúce chovanie operátorov sa v *TFLite* a *ONNX* využíva rozdielna terminológia. *TFLite* má parametre a *ONNX* využíva atribúty. Jedná sa však o rovnaký koncept, s iným názvom. Sú to hodnoty, ktoré ovplyvňujú chovanie operátoru. Často priamo vystupujú v matematickom výraze, ktorý operátor implementuje.

Náznak mapovania *ONNX* operátorov na *TFLite* operátory je zhrnutý v tabuľke 3.1. Ich konverzia je spravidla komplexnejšia, než obyčajné nahradenie ekvivalentnou verziou v *TFLite* modeli. V nasledujúcich pod-sekciách sú preto opísané významy niektorých operátorov, ich implementácie v *ONNX* a *TFLite*, a detaily postupu konverzie medzi nimi.

3.1.1 Operátor BatchNormalization

Trénovanie hlbokých neurónových sietí je skomplikované faktom, že distribúcia vstupov každej vrstvy sa mení počas učenia, keďže aj parametre predchádzajúcich vrstiev sa menia. Táto

[1] Dostupné z https://onnx.ai/onnx/operators/index.html

[2] Dostupné z https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/kernels/internal/reference

ONNX operátor	*TFLite* operátory	Detail
`Add`	`Add`	
`AveragePool`	`AveragePool2D`	
`BatchNormalization`	`Mul`, `Add`	Sekcia 3.1.1
`Conv`	`Conv2D` alebo `Conv3D`	Sekcia 3.1.2
`Constant`		Sekcia 3.1.3
`Dropout`		Sekcia 3.1.4
`Gemm`	`FullyConnected`, `Transpose`, `Mul`	Sekcia 3.1.5
`LeakyRelu`	`LeakyRelu`	
`LogSoftmax`	`LogSoftmax`	
`LRN`	`LocalResponseNormalization`	Sekcia 3.1.6
`MaxPool`	`MaxPool2D`	Sekcia 3.1.7
`MatMul`, `Add`	`FullyConnected`	
`Mul`	`Mul`	
`Relu`	`Relu`	
`Reshape`	`Reshape`, `Transpose`	Sekcia 3.1.8
`Softmax`	`Softmax`	Sekcia 3.1.9
`Sum`	`Add` alebo `AddN`	Sekcia 3.1.10
`Transpose`	`Transpose`	

Tabulka 3.1: Tabuľka mapovania *ONNX* operátorov na *TFLite* operátory.

skutočnosť spomaľuje učenie tým, že vyžaduje nižší *learning rate* a opatrnú inicializáciu parametrov. Tento problém je možné adresovať normalizáciou vstupov vrstiev. Metóda *batch normalization* je účinná vďaka tomu, že zakomponuje normalizáciu do architektúry modelu a aplikuje ju na každú trénovaciu dávku dát. Umožňuje používať o mnoho vyššie *learning rates* a znižuje mieru opatrnosti, nutnú pri inicializácii parametrov. Taktiež slúži ako určitý regularizátor a v niektorých prípadoch oslobodzuje od potreby použiť *dropout*. V praxi využitie *batch normalization* dosahuje vyššej presnosti pri klasifikácii so 14 krát menším počtom trénovacích krokov a celkovo prekoná pôvodný model so značným rozdielom. Matematické vyjadrenie operácie *batch normalization* je na obrázku 3.1. Počas učenia sa hodnoty $E[x]$ a $Var[x]$ pravidelne upravujú. V dobe inferencie už zostávajú konštantné. [8]

$$\widehat{x} = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}}$$

Obrázek 3.1: Matematický výraz operácie *batch normalization*. Prevzaté z [8].

.

Štandard *ONNX* definuje operátor `BatchNormalization`, ktorý realizuje vlastnú verziu tejto operácie. *TFLite* podobný operátor nemá, takže presná konverzia nie je možná. Táto práca sa zameriava na konverziu už natrénovaných modelov a predpokladá sa, že ďalej na nich bude vykonávaná iba inferencia. Funkcionalitu `BatchNormalization` výhradne v čase inferencie je možné reprezentovať v *TFLite*.

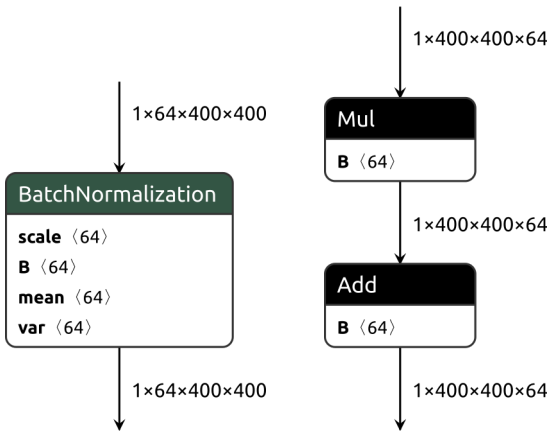
ONNX implementácia *batch normalization* počas inferencie jednoducho realizuje nasledujúcu operáciu.

$$Y = \frac{X - mean}{\sqrt{var + epsilon}} \cdot scale + B$$

Kde *mean*, *var*, *scale* a B sú tenzory rovnakých rozmerov a *epsilon* je skalárny parameter operátoru. Takže `BatchNormalization` je možné reprezentovať v *TFLite* ako sústavu viacerých operátorov, vykonávajúcich základné matematické operácie. Pokiaľ sú v modeli tenzory z rovnice staticky uložené, teda nie sú výsledkom výpočtu počas inferencie, je možné výraz prepísať do nasledujúceho tvaru.

$$Y = X \cdot \frac{scale}{\sqrt{var + epsilon}} - \frac{mean \cdot scale}{\sqrt{var + epsilon}} + B$$

V *TFLite* teda stačí vygenerovať operátor `Mul`, ktorého vstupmi budú pôvodný vstup `BatchNormalization` a staticky predpočítaný tenzor s hodnotou $\frac{scale}{\sqrt{var+epsilon}}$. Za ním bude nasledovať operátor `Add`, ktorého vstupmi budú výstup predchádzajúceho `Mul` operátoru a staticky vypočítaný tenzor s hodnotou $B - \frac{mean \cdot scale}{\sqrt{var+epsilon}}$. Nie je teda nutné používať jeden operátor pre každú matematickú operáciu. Príklad konverzie takejto situácie je zobrazený na obrázku 3.2.



(a) *ONNX* (b) *TFLite*

Obrázek 3.2: Výsledok konverzie *ONNX* `BatchNormalization` na *TFLite* `Mul` a `Add`. Grafická reprezentácia operátorov bola vygenerovaná nástrojom *Netron*.

V prípade že niektorý zo vstupov *ONNX* operátoru `BatchNormalization` nie je staticky daný by bolo nutné pri konverzii použiť viacero *TFLite* operátorov. Táto situácia sa prakticky nevyskytuje, keďže súčasťou myšlienky *batch normalization* je pravidelná úprava svojich parametrov počas učenia, vzhľadom na vstupné dáta.

Informácie ohľadom *ONNX* operátoru `BatchNormalization` som čerpal z oficiálnej dokumentácie[3]. Možnosti konverzie som sám odvodil na základe uvedených zdrojov.

3.1.2 Operátor Conv

Konvolučné siete, alebo *CNN*, sú typom neurónových sietí pre spracovanie dát, ktoré majú známu mriežkovú topológiu. Jedná sa napríklad o dáta obrázkov, ktoré sú dvoj dimenzio-

[3]Dostupné z: https://onnx.ai/onnx/operators/onnx__BatchNormalization.html

nálnym polom pixelov alebo o 1D dáta so vzorkami signálu. Konvolučné siete boli nesmierne úspešné v určitých aplikáciách. Ako názov napovedá, využívajú matematickú operáciu zvanú konvolúcia, čo je typ lineárnej operácie zobrazenej na obrázku 3.3. V *CNN* modeloch je konvolúcia v niektorých prípadoch používaná ako náhrada za generické násobenie matíc pri plne prepojených vrstvách. [3, s. 326]

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n).$$

Obrázek 3.3: Matematický výraz pre dvoj dimenzionálnu konvolúciu. Prevzatý z [3, s. 328].

Konvolúcia využíva štyri dôležité myšlienky, ktoré pomáhajú vylepšovať systémy strojového učenia.

- riedke prepojenie vrstiev
- zdieľanie parametrov
- ekvivariancia voči posunu
- podpora vstupov o rôznych veľkostiach

Plne prepojené vrstvy neurónovej siete využívajú násobenie matíc s parametrami pre spracovanie dát. To znamená, že každá vstupná jednotka interaguje s každou výstupnou. Konvolučné siete však využívajú riedke prepojenie vrstiev, vďaka tomu, že konvolučné jadro je menšie než vstupný tenzor. Takže napríklad ak má vstupný obrázok milióny pixelov, je možné odhaliť aj malé ale významné vlastnosti ako napríklad hrany, s pomocou jadra pozostávajúceho z len desiatok až stoviek pixelov. To znamená, že nie je potrebné ukladať také veľké množstvo parametrov. [3, s. 329-330]

Zdieľaním parametrov sa myslí používanie tých istých parametrov pre viaceré funkcie v modeli. V tradičných neurónových sieťach je každý prvok matice váh využitý práve raz, pri výpočte výstupu vrstvy. V *CNN* je každý prvok konvolučného jadra použitý pri spracovaní každého prvku vstupu. Toto zdieľanie parametrov znamená, že namiesto trénovania samostatnej sady parametrov pre každú časť vstupu, stačí natrénovať iba jednu sadu pre celý vstup. Tento prístup značne znižuje veľkosť modelu. [3, s. 331-333]

V prípade konvolúcie, spomínaná forma zdieľania parametrov spôsobuje vlastnosť zvanú ekvivariancia voči posunu. To znamená že ak sa posunie vstup konvolúcie, efekt bude rovnaký, ako keby bol posunutý iba jej výstup. Takže napríklad pri spracovaní signálu, konvolúcia produkuje určitú časovú os s rôznymi rysmi, ktoré sa vyskytli na vstupe. Pokiaľ sa nejaká vstupná udalosť presunie na neskorší čas, jej rovnaká reprezentácia sa objaví na výstupe, iba neskôr. Podobne pri obrázkoch, konvolúcia vytvára 2D mapu rysov, ktoré sa objavujú na vstupe. Pri posunutí vstupného objektu sa jeho reprezentácia na výstupe posunie o rovnaké množstvo. To je užitočné napríklad pri hľadaní hrán. Podobné hrany sa objavujú na rôznych miestach vo vstupe, takže je praktické využiť jednu sadu parametrov pre celý obrázok. [3, s. 334-335]

ONNX operátor `Conv` realizuje konvolúciu vstupu s jadrom o ľubovoľných dimenziách. Dimenzionalita vstupu je daná *ONNX* atribútom `kernel_shape`. *TFLite* podporuje špecializované operátory `Conv2D` a `Conv3D`. Konvolúciu dimenzionalít vyšších ako 3D nepodporuje. V prípade že `kernel_shape` má práve 2 alebo 3 dimenzie, je možné operátor priamo konvertovať na jeho príslušnú *TFLite* variantu. Mapovanie *ONNX* atribútov na parametre *TFLite* operátorov je špecifikované v tabuľke 3.2.

ONNX `Conv` atribút	*TFLite* `Conv2D` parametre	*TFLite* `Conv3D` parametre
`strides`	`stride_w`, `stride_h`	`stride_w`, `stride_h`, `stride_d`
`dilations`	`dilation_w_factor`, `dilation_h_factor`	`dilation_w_factor`, `dilation_h_factor`, `dilation_d_factor`
`pads`, `auto_pads`	Opísané v sekcii 3.1.11	
`kernel_shape`	Netreba konvertovať, implicitne dané operátorom	

Tabulka 3.2: Konverzia atribútov *ONNX* operátoru `Conv` na parametre *TFLite* operátorov `Conv2D` a `Conv3D`

V prípade konverzie na operátor `Conv2D` aj na `Conv3D` zostávajú vstupy operátoru rovnaké. Prvý vstup je tenzor, na ktorý je konvolúcia aplikovaná. Druhý vstup predstavuje tenzor váh konvolučného jadra, ktorý je typicky staticky uložený v modeli. V takom prípade je nutné ho konvertovať z formátu NCHW na NHWC. Tretí vstup je tenzor *bias*, pripočítaný ku výsledku. *ONNX* špecifikácia umožňuje aby bol tenzor *bias* vynechaný, pokiaľ sa k výsledku nemá nič pripočítať. *TFLite* toto neumožňuje. V takomto prípade je nutné vytvoriť nový statický tenzor, ktorý obsahuje iba hodnoty 0 a napojiť ho na tretí vstup operátoru.

ONNX umožňuje operátorom `Conv` reprezentovať aj konvolúciu s 1-dimenzionálnym jadrom. *TFLite* príslušný operátor nepodporuje, ale 1D konvolúciu je možné reprezentovať za použitia operátoru `Conv2D`, kedy jedna z dimenzií konvolučného jadra je rovná 1. Vstup vo formáte *NCH* je nutné rozšíriť o jednu dimenziu a transformovať na *NHWC*. Po konvolučnom operátore je zas potrebné výstupný tenzor navrátiť do pôvodného formátu. Tieto transformácie sú zabezpečené operátormi `Reshape` a `Transpose` s vhodnými parametrami. Výsledok konverzie tejto situácie je zobrazený na obrázku 3.4.

V prípade viacerých po sebe nasledujúcich operátorov, ktoré podobným spôsobom emulujú 1-dimenzionálne operácie, nie je nutné zakaždým transformovať výstupný tenzor na *NCH* a späť na *NHWC*. Stačí operátory `Reshape` a `Transpose` pridať na začiatku a na konci celej postupnosti týchto operátorov.

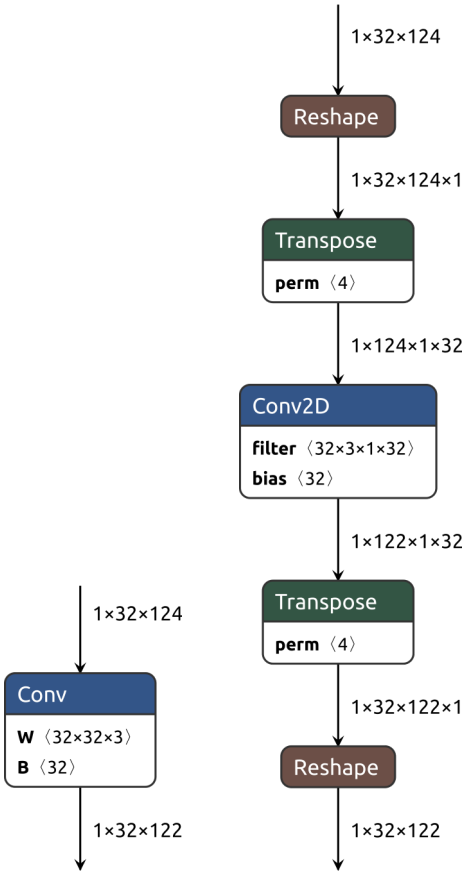
Informácie v tejto sekcii ohľadom *ONNX* operátoru `Conv` som čerpal z oficiálnej *ONNX* dokumentácie[4]. Poznatky ohľadom *TFLite* operátorov `Conv2D` a `Conv3D` som odvodil zo schémového súboru s definíciou *.tflite* súboru[5]. Detaily a možnosti konverzie som navrhol a odvodil sám.

3.1.3 Operátor Constant

`Constant` je *ONNX* operátor, ktorý produkuje konštantný tenzor. Tento operátor nemá žiadne vstupy a jeho výstup je vždy rovný hodnote jedného z jeho atribútov. Tieto atribúty sú nasledovné.

- `sparse_value` - riedky tenzor
- `value` - všeobecný tenzor

[4]Dostupné z: https://onnx.ai/onnx/operators/onnx__Conv.html
[5]Dostupné z: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/schema/schema.fbs



(a) *ONNX* (b) *TFLite*

Obrázek 3.4: Výsledok konverzie 1-dimenzionálneho *ONNX* operátoru `Conv` na *TFLite* `Conv2D` s pridanými `Transpose` a `Reshape` operátormi. Grafická reprezentácia operátorov bola vygenerovaná nástrojom *Netron*.

- `value_float` - 0-dimenzionálny tenzor s jednou hodnotou typu `float32`
- `value_floats` - 1-dimenzionálny tenzor s hodnotami typu `float32`
- `value_int` - 0-dimenzionálny tenzor s jednou hodnotou typu `int64`
- `value_ints` - 1-dimenzionálny tenzor s hodnotami typu `int64`
- `value_string` - 0-dimenzionálny tenzor s jednou hodnotou typu `UTF-8 string`
- `value_strings` - 1-dimenzionálny tenzor s hodnotami typu `UTF-8 string`

TFLite obdobný operátor nepodporuje. Konverziu je možné realizovať vytvorením statického tenzoru s hodnotou použitého atribútu operátoru `Constant`. Tento tenzor bude následne napojený na vstup operátorov, ktoré pracovali s výstupom operátoru `Constant`.

Informácie ohľadom *ONNX* operátoru `Constant` som čerpal z oficiálnej dokumentácie[6].

[6] https://onnx.ai/onnx/operators/onnx__Constant.html

3.1.4 Operátor Dropout

Hlboké neurónové siete pozostávajú z viacerých nelineárnych skrytých vrstiev, čo im umožňuje naučiť sa veľmi komplikované vzťahy medzi vstupmi a výstupmi modelu. Avšak s obmedzeným množstvom trénovacích dát, mnohé z týchto vzťahov vzniknú ako dôsledok vzorkovacieho šumu. Takže existujú iba v trénovacej sade, a už nie v reálnych testovacích dátach. Toto vedie k fenoménu *overfitting*. Jednou z metód regulácie *overfittingu* je kombinácia výstupov rozličných modelov s rozdielnou architektúrou. Naivná realizácia takéhoto prístupu je však výpočetne až príliš drahá. Technika *Dropout* rieši spomínaný problém a poskytuje spôsob ako efektívne kvázi kombinovať exponenciálne množstvo rôznych neurónových sietí s rôznymi architektúrami. Realizuje sa *vyhadzovaním* prvkov neurónovej siete, teda neurónov. *Vyhodením* prvku sa myslí jeho dočasné odstránenie zo siete, aj spolu s jeho vstupnými a výstupnými prepojeniami. Výber prvkov pre vyhodenie je náhodný. Takže pri každom kroku trénovania sa z pôvodnej siete zachová iná časť jej prvkov.[14]

Architektonicky je teda sieť pri každom kroku odlišná, ale keďže sa vždy jedná o podmnožinu jednej siete, trénované váhy sú rovnaké. Táto technika znižuje komplexné koadaptácie neurónov, keďže žiaden neurón sa nemôže spoliehať na prítomnosť iných konkrétnych neurónov. Z tohto dôvodu je nútený naučiť sa rozpoznávať robustnejšie vlastnosti vstupu a nie len šum.[10]

ONNX definuje operátor `Dropout`, ktorý realizuje práve túto operáciu. Umožňuje špecifikovať aké percento prvkov jeho prvého vstupu sa zahodí, pomocou druhého vstupného skaláru `ratio`. Operátor je určený na použitie výhradne počas doby učenia. Je aktívny jedine pokiaľ jeho tretí vstup `training_mode` je nastavený na *true*. V čase inferencie, keď je `training_mode` *false*, nemá operátor žiaden efekt na vstupné dáta.

TFLite nepodporuje operátor, ktorý by bol schopný realizovať ekvivalentné chovanie. Keďže účelom tejto práce je konverzia už natrénovaných modelov, na ktorých bude následne vykonávaná iba inferencia, je možné operátor vo výslednom modeli jednoducho vynechať.

Informácie ohľadom *ONNX* operátoru `Dropout` som čerpal z oficiálnej dokumentácie[7].

3.1.5 Operátor Gemm

`Gemm` (*General Matrix Multiplication*) je *ONNX* operátor, ktorý realizuje matematickú operáciu z rovnice 3.1.

$$Y = alpha * A' * B' + beta * C \tag{3.1}$$

kde A' je rovné buď vstupnému tenzoru A, alebo jeho transponovanej verzii ak je atribút `transA` nastavený na *true*. Podobne B' predstavuje buď vstupný tenzor B, alebo transponované B ak je nastavený atribút `transB`. *alpha* a *beta* sú skalárne hodnoty. [12]

Operátor `Gemm` je možné konvertovať na *TFLite* operátor `FullyConnected`. Ten realizuje prenásobenie dvoch tenzorov A a B, a pripočítanie tretieho tenzoru C k výsledku. Nepodporuje však násobenie jednotlivých tenzorov skalárnymi hodnotami, ani prípadné transponovanie vstupov. Túto funkcionalitu je možné zabezpečiť spôsobmi, opísanými v tabuľke konverzie atribútov 3.3.

V prípade že atribúty `alpha` a `beta` sú rovné 1.0, nie je nutné sa ich konverziou zaoberať.

Dôvodom pre rozdielnu konverziu atribútov `transA` a `transB` je rozdielna implementácia inferenčných programov *ONNX* a *TFLite*. Z *ONNX* dokumentácie a súboru s imple-

[7] Dostupné z: https://onnx.ai/onnx/operators/onnx__Dropout.html

ONNX atribút	*TFLite* parameter
`alpha`	Pokiaľ je aspoň jeden z tenzorov A a B staticky uložený v modeli, je možné jeho dáta staticky prenásobiť hodnotou `alpha` počas konverzie. Ak sú oba tenzory vypočítané v čase inferencie modelu, je nutné pred jeden z nich vložiť *TFLite* operátor `Mul` so vstupom `alpha`.
`beta`	Pokiaľ je vstupný tenzor C statický, môže byť prenásobený hodnotou `beta` počas konverzie. Pokiaľ je vypočítaný, je opäť nutné pred neho vložiť *TFLite* operátor `Mul` so vstupom `beta`.
`transA`	Pokiaľ má atribút hodnotu *true*, je nutné tenzor A transponovať. Pokiaľ je statický, jeho dáta sa transponujú počas konverzie. Pokiaľ je A vypočítaný teznor, je nutné pred neho vložiť *TFLite* operátor `Transpose`.
`transB`	Rovnaký postup ako pri atribúte `transA`, ale tenzor B musí byť transponovaný práve vtedy, keď je atribút `transB` nastavený na *false*.

Tabulka 3.3: Konverzia atribútov *ONNX* operátoru `Gemm`

mentáciou inferencie operátoru `FullyConnected`[8] som zistil, že *ONNX* vyžaduje, aby mali vstupné tenzory správny tvar z matematického pohľadu. Teda keď zjednodušíme situácie na násobenie dvoj-dimenzionálnych matíc, *ONNX* operátor `Gemm` požaduje, aby ich tvary boli $[a, b] \times [b, c]$. Takže počet stĺpcov prvej matice sa zhoduje s počtom riadkov druhej. Naopak TFLite operátor `FullyConnected` vyžaduje, aby sa ich prvé dimenzie zhodovali. Teda aby matice boli v tvare $[a, b] \times [a, c]$.

3.1.6 Operátor LRN

ONNX operátor `LRN` (*Local Response Normalization*) implementuje matematickú operáciu popísanú v štúdii k hlbokej konvolučnej sieti *Alexnet*. Táto normalizácia odozvy implementuje formu plošnej inhibície, inšpirovanej skutočnými neurónmi v ľudskom mozgu. Vytvára priestor pre určitú súťaž medzi neurónmi o čo najväčšiu aktiváciu. Dôsledkom je experimentálne podložené zníženie tzv. top-1 a top-5 chyby pri inferencii. Teda zvyšuje presnosť výsledkov neurónovej siete. [10] Definícia tejto operácie je na obrázku 3.5.

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0, i-n/2)}^{\min(N-1, i+n/2)} (a^j_{x,y})^2 \right)^{\beta}$$

Obrázek 3.5: Matematický výraz pre výpočet Local Response Normalization, kde $a^i_{x,y}$ je aktivita neurónu jadra i na pozícii (x, y), $b^i_{x,y}$ je jeho aktivita po výpočte. α, β k a n sú parametre a N je celkový počet jadier. Prevzaté z [10]

[8] Dostupné z https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/kernels/internal/reference/fully_connected.h

ONNX aj *TFLite* podporujú operátory, ktoré túto operáciu implementujú. *TFLite* verzia sa nazýva `LocalResponseNormalization`. Analýzou *ONNX* dokumentácie a súboru s implementáciou inferencie *TFLite* operátoru `LocalResponseNormalization`[9] som zistil, že ich algoritmy implementujú rovnicu z obrázku 3.5 rôznymi spôsobmi. *ONNX* využíva atribút `size` ktorý približne zodpovedá parametru n z rovnice. *TFLite* používa `radius`, ktorý zodpovedá $n/2$. Väčší rozdiel je pri atribúte `alpha`, ktorý sa v *ONNX* implementácii objavuje jedine predelený atribútom `size`. Mapovanie *ONNX* atribútov na *TFLite* parametre definované v tabuľke 3.4 zabezpečuje ekvivalentné výstupy operátorov.

TFLite parameter	*ONNX* atribút
`alpha`	`alpha`/`size`
`beta`	`beta`
`bias`	`bias`
`radius`	$\lfloor$`size`$/2\rfloor$ Pokiaľ je `size` párne, nebude *TFLite* verzia produkovať identické výstupy. Mali by však byť dostatočne podobné pre praktické použitie.

Tabulka 3.4: Konverzia atribútov *ONNX* operátoru `LRN` na parametre *TFLite* operátoru `LocalResponseNormalization`

3.1.7 Operátor MaxPool

Takzvané *pooling* operácie tvoria dôležitú časť konvolučných neurónových sietí. Tieto operácie redukujú veľkosť vstupných tenzorov tým, že využívajú nejakú funkciu, ktorá sumarizuje menšie regióny vstupu. Často sa jedná napríklad o priemer alebo výber maxima z každého regiónu. *Pooling* funguje priložením a posúvaním určitého okna cez vstupný tenzor a následné predávanie jeho obsahu zvolenej funkcii. V určitom zmysle *pooling* funguje podobne ako diskrétna konvolúcia, ale lineárna operácia charakterizovaná konvolučným jadrom je nahradená inou funkciou. V neurónovej sieti poskytujú *pooling* vrstvy invarianciu voči malému posunu vstupu. Najčastejšie sa vyskytujúcim typom *poolingu* je tzv. *max pooling*. [1]

ONNX a *TFLite* podporujú operátory `MaxPool` a `MaxPool2D`, ktoré implementujú spomínanú operáciu *max pooling*. Konverzia samotných atribútov *ONNX* operátoru na *TFLite* parametre je špecifikovaná v tabuľke 3.5.

ONNX atribút	*TFLite* parameter
`auto_pad`, `pads`	Opísané v sekcii 3.1.11
`strides`	`stride_w`, `stride_h`
`kernel_shape`	`filter_width`, `filter_height`
`dilations`	Neexistuje ekvivalentný parameter
`storage_order`	Neexistuje ekvivalentný parameter
`ceil_mode`	Neexistuje ekvivalentný parameter

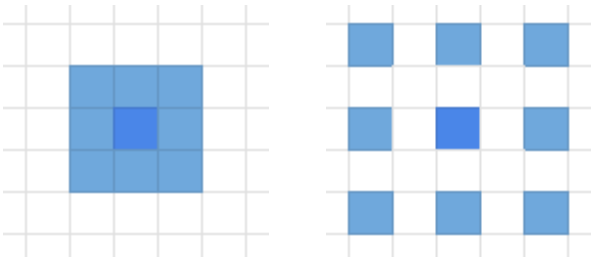
Tabulka 3.5: Konverzia atribútov *ONNX* operátoru `MaxPool` na parametre *TFLite* operátoru `MaxPool2D`.

[9] Dostupné z: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/kernels/internal/reference/reference_ops.h

Atribút `kernel_shape` udáva veľkosť *okna* v každej dimenzii, pričom nie je explicitne stanovený limit na ich počet. *ONNX* teda podporuje viac dimenzionálne okno, kdežto *TFLite* iba dvoj dimenzionálne. V prípade že by `kernel_shape` mal viac než dve dimenzie by konverzia operátorov nebola možná. V prípade 1-dimenzionálneho operátoru je možné využiť `MaxPool2D` s jednou dimenziou okna rovnou 1 a pridaním operátorov `Transpose` a `Reshape`. Jedná sa o podobnú situáciu ako pri operátore `Conv`, ktorá je detailnejšie opísaná v sekcii 3.1.2.

Atribút `storage_order` ovplyvňuje výhradne druhý výstup *ONNX* verzie operátoru `MaxPool`, ktorý obsahuje pôvodné indexy príslušných hodnôt v hlavnom výstupe operátoru. *TFLite* takýto druhý výstup nepodporuje, preto nie je konverzia atribútu možná.

Atribút `dilations` popisuje akým spôsobom sa má jadro operácie (spomínané *okno*) rozťahovať. Príklad vplyvu tohto atribútu na okno je zobrazený na obrázku 3.6. *TFLite* verzia operátoru takúto funkcionalitu nepodporuje, implicitne pracuje kvázi s hodnotami dilatácie 1 v každom smere. Pokiaľ by atribút mal inú hodnotu než (1.1), priama konverzia operátoru by nebola možná. Potenciálne možnosti konverzie pomocou kombinácie s inými operátormi som neskúmal.



(a) `dilations` = (1, 1) (b) `dilations` = (2, 2)

Obrázek 3.6: Vplyv *ONNX* atribútu `dilations` na tvar okna *pooling* operácie. Tmavomodrá bunka v strede predstavuje stred okna. Svetlomodré bunky predstavujú zbytok okna.

Atribút `ceil_mode` určuje spôsob zaokrúhľovania pri výpočte tvaru výstupného tenzoru. *TFLite* podobnú funkcionalitu nepodporuje. V modeloch, ktorými sa táto práca zaoberá, nespôsobuje atribút `ceil_mode` problémy pri konverzii.

Informácie v tejto sekcii ohľadom *ONNX* operátoru `MaxPool` a dilatácie som čerpal z oficiálnej *ONNX* dokumentácie[10]. Poznatky ohľadom *TFLite* operátoru `MaxPool2D` som odvodil zo súboru s implementáciou jeho inferencie[11].

3.1.8 Operátor Reshape

ONNX a *TFLite* operátory `Reshape` umožňujú zmeniť tvar ľubovoľného tenzoru. Počet prvkov vstupného a výstupného tenzoru operátorov musí byť však zachovaný. Teda súčin dimenzií nového tvaru sa musí rovnať súčinu dimenzií pôvodného. Operátory v oboch špecifikáciách očakávajú ako prvý vstup, tenzor ktorého tvar bude zmenený. Druhým vstupom je tenzor celočíselných hodnôt, ktorý špecifikuje nový tvar tenzoru.

[10] Dostupné z: https://onnx.ai/onnx/operators/onnx__MaxPool.html

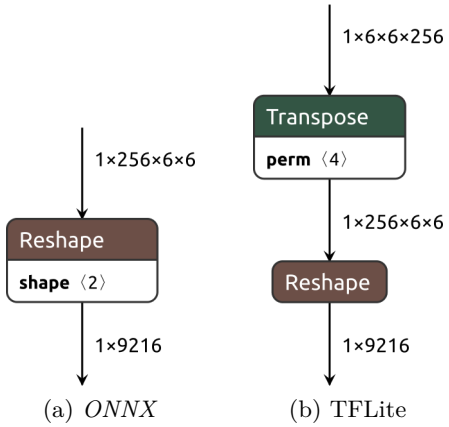
[11] Dostupné z: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/kernels/internal/reference/pooling.h

ONNX `Reshape` obsahuje jeden atribút `allowzero`, ktorý umožňuje zachovanie konkrétnych dimenzií vstupného tenzoru umiestnením hodnoty 0 na príslušné miesto v tenzore určujúcom nový tvar. *TFLite* takúto funkcionalitu nepodporuje.

TFLite `Reshape` umožňuje uložiť nový tvar tenzoru aj pomocou parametra `new_shape`, pokiaľ je tvar staticky daný. Toto jemne zmenší veľkosť modelu v pamäti.

Konverzia je teda relatívne priamočiara. Je akurát nutné konvertovať tenzor určujúci nový tvar vstupu, do formátu *NHWC*, v prípade že bol vo formáte *NCHW*. Ak je tenzor s novým tvarom vypočítaný, je nutné túto transformáciu realizovať dynamicky, pridaním operátoru `Transopse`.

Niekedy sa v modeloch vyskytuje situácia, kedy je operátor `Reshape` používaný za účelom tzv. *sploštenia* tenzoru. Teda 4-dimenzionálny tenzor je pretvarovaný na 2-dimenzionálny. Tu nastáva problém, že operátory pred `Reshape` pracovali v *ONNX* modeli s tenzorom vo formáte *NCHW* a v konvertovanom *TFLite* modeli, vo formáte *NHWC*. Po *sploštení* na dve dimenzie už však tieto formáty strácajú význam. Nasledujúce operátory sa týmito formátmi nezaoberajú, ale pracujú s dátami tak, ako sú. Jedná sa napríklad o operátory násobenia matíc, alebo samotný výstup siete. Po *sploštení* z oboch formátov majú výstupné tenzory síce rovnaký tvar, ale konkrétne dáta sú uložené na rôznych pozíciách v tom tenzore. Má to za následok, že váhy pri násobení tenzorov v konvertovanom *TFLite* modeli, by boli natrénované na *sploštené NCHW* dáta. Dostávali by však *sploštené NHWC* dáta, a teda výstupy by boli úplne odlišné. Podobne pokiaľ by napríklad bezprostredne za operátorom `Reshape` nasledoval výstup siete, najvyššia hodnota tenzoru by bola na inom indexe. Takže napríklad sieť pre klasifikáciu obrázkov by produkovala nezmyselné výsledky.



(a) *ONNX* (b) TFLite

Obrázek 3.7: Výsledok konverzie *ONNX* `Reshape` na *TFLite* `Reshape` s pridaným `Transpose` operátorom. Vstup *ONNX* operátoru je vo formáte *NCHW*, ale výstup už nie. Grafická reprezentácia operátorov bola vygenerovaná nástrojom *Netron*.

Jednoduchým riešením je pridanie *TFLite* operátoru `Transpose` pred operátor `Reshape`. Výstupné dáta budú identické s tými v *ONNX* modeli. Príklad takejto situácie je na obrázku 3.7.

V prípade že by v tejto situácii za operátorom `Reshape` nasledovali iba operátory násobenia tenzorov so staticky uloženými váhami, by bolo možné tieto statické tenzory transformovať v dobe konverzie tak, aby výstup mal požadované vlastnosti. Táto metóda by

vyprodukovala menší model, ktorého inferencia by bola samozrejme rýchlejšia, keďže by sa ušetril jeden operátor `Transpose`. Túto možnosť konverzie som detailnejšie neskúmal.

3.1.9 Operátor Softmax

Softmax je pomenovanie funkcie, ktorá sa často využíva na odhad pravdepodobností spojených s kategorickým rozdelením. Po aplikovaní *softmax* na tenzor je súčet jeho hodnôt rovný 1. Samotná funkcia je definovaná výrazom na obrázku 3.8. Pri implementácii tejto funkcie za pomoci čísel s plávajúcou rádovou čiarkou môžu nastať rôzne problémy. Za predpokladu že všetky vstupy x_i nadobúdajú približne rovnakú hodnotu c, je analyticky dokázateľné, že každý výstup by mal byť približne rovný $1/n$. Pri numerickom vyhodnotení funkcie toto však nemusí nastať. Pokiaľ by c bolo veľmi negatívne, $exp(c)$ *podtečie*, teda bude rovné 0. Takže menovateľ funkcie bude taktiež 0 a výstup nebude definovaný. Podobne ak c je veľké pozitívne číslo, $exp(c)$ pretečie a výsledok výrazu bude opäť nedefinovaný. Oba problémy môžu byť vyriešené vyhodnocovaním funkcie $softmax(\mathbf{z})$, kde $\mathbf{z} = \mathbf{x} - max_i x_i$. Algebraicky je možné dokázať, že výstup funkcie sa nezmení odčítaním skaláru od vstupného vektoru. [3, s. 78-79]

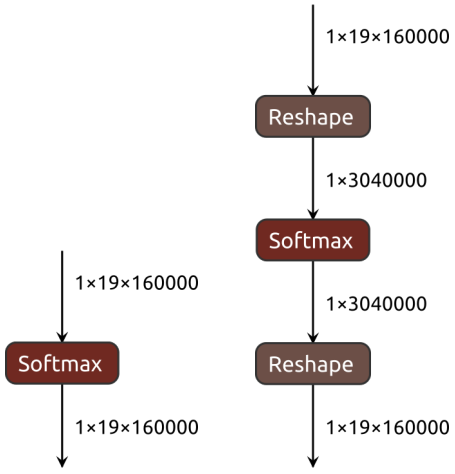
Zo súborov s implementáciami *TFLite* a *ONNX* operátorov `Softmax` som vydedukoval, že oba toto ošetrenie pretečenia implicitne implementujú.

$$\text{softmax}(\boldsymbol{x})_i = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)}.$$

Obrázek 3.8: Matematický výraz pre výpočet funkcie *softmax*. [3, s. 79]

ONNX aj *TFLite* implementujú svoju verziu operátoru `Softmax`. *ONNX* verzia používa jeden atribút `axis`. Ten určuje na ktoré dimenzie bude operácia *softmax* aplikovaná. Jeho hodnota udáva index prvej dimenzie, na ktorú sa operácia aplikuje. Situácia kedy napríklad na vstupe je tenzor s tvarom $[d0, d1, d2, d3]$ a atribút *axis* má hodnotu 2 je ekvivalentná prípadu, kedy by vstupný tenzor mal tvar $[d0 \cdot d1, d2 \cdot d3]$. Operácia *softmax* sa následne aplikuje na jeho druhú dimenziu. *TFLite* obdobnú funkcionalitu nepodporuje. Striktne vyžaduje, aby vstupný tenzor mal práve dve dimenzie a *softmax* aplikuje opäť na tú druhú. V prípade že *ONNX* operátor `Softmax` má hodnotu atribútu *axis* inú než -1 alebo s indexom poslednej dimenzie tenzoru, je jeho konverzia na *TFLite* možná kombináciou s dvomi operátormi `Reshape`. Jeden pred a druhý bezprostredne po operátore `Softmax`. Prvý `Reshape` predpripravý tvar tenzoru na dvoj dimenzionálny a to ekvivalentným spôsobom k tomu, ako to robí *ONNX*. Druhý `Reshape` za operátorom zas zmení tvar výstupného tenzoru naspäť, aby bol kompatibilný s ďalšími operátormi v modeli, či celkovým výstupom modelu. Príklad konverzie takejto situácie, kedy vstup *ONNX* operátoru `Softmax` má tvar $[1, 19, 160000]$ a jeho atribút *axis* má hodnotu 1, je zobrazený na obrázku 3.9.

TFLite `Softmax` podporuje jeden skalárny parameter `beta`, ktorý umožňuje vstupné hodnoty po odčítaní maxima, ešte prenásobiť, pred aplikáciou exponenciálnej funkcie. Teda namiesto každého výskytu výrazu $exp(x_i)$ vo vzorci 3.8, sa použije výraz $exp((x_i - max_i x_i) \cdot$ `beta`). Pri konverzii je tento parameter vždy nastavený na hodnotu 1.0, keďže *ONNX* takúto funkcionalitu nepodporuje.



(a) *ONNX* (b) *TFLite*

Obrázek 3.9: Výsledok konverzie *ONNX* `Softmax` na *TFLite* `Softmax` s pridanými `Reshape` operátormi. Grafická reprezentácia operátorov bola vygenerovaná nástrojom *Netron*.

Poznatky týkajúce sa *ONNX* operátoru `Softmax` som získal z oficiálnej dokumentácie[12] a zo súoru s referenčnou implementáciou inferencie operátoru[13]. Informácie týkajúce sa *TFLite* verzie operátoru som získal analýzou súboru s implementáciou jeho inferencie[14].

3.1.10 Operátor Sum

`Sum` je *ONNX* operátor, ktorý realizuje sčítanie ľubovoľného počtu vstupných tenzorov po prvkoch. V *TFLite* existuje operátor `AddN`, ktorý implementuje presne túto operáciu. V reálnych modeloch sa niekedy vyskytujú situácie, kedy operátor `Sum` má iba jeden vstup vypočítaný. Ostatné sú statické a obsahujú výhradne hodnoty 0. Za takýchto okolností je možné operátor pri konverzii úplne vynechať. Ďalšia situácia ktorá v modeloch často nastáva je ,že operátor `Sum` má práve dva netriviálne vstupy. V tejto situácii je možné `Sum` konvertovať aj na *TFLite* operátor *Add*, ktorý sčíta práve dva vstupné tenzory po prvkoch.

Výhodou oproti konverzii na `AddN` je fakt, že operátor `Add` je jedným z mála, ktoré podporujú parameter `fused_activation_function`. Ten umožňuje operátor spojiť dokopy s jednoduchou aktivačnou funkciou, ktorá je aplikovaná na jeho výstupy. `AddN` tento parameter nepodporuje. Spojením operátoru s aktivačnou funkciou sa zmenší veľkosť celkového modelu, keďže aktivačnú funkciu nie je nutné reprezentovať samostatným operátorom. Taktiež sa ušetrí nutnosť špecifikovať jeden vypočítaný tenzor, ktorý by predstavoval výstup operátoru `AddN` a zároveň vstup operátoru aktivačnej funkcie.

Nie všetky aktivačné funkcie môžu byť spojené s predchádzajúcim operátorom pomocou spomínaného parametra `fused_activation_function`. *TFLite* schéma definuje podporu iba pre nasledujúce aktivačné funkcie.

- `RELU`

[12] Dostupné z: https://onnx.ai/onnx/operators/onnx__Softmax.html
[13] Dostupné z: https://github.com/onnx/onnx/blob/main/onnx/reference/ops/op_softmax.py
[14] Dostupné z: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/kernels/internal/reference/softmax.h

- `RELU_N1_TO_1`
- `RELU6`
- `TANH`
- `SIGN_BIT`

Iné aktivačné funkcie, ako napríklad `LeakyRelu`, ktorý má narozdiel od podporovaných funkcií aj vlastný parameter `alpha`, musia byť reprezentované samostatným operátorom.

Túto možnosť konverzie je možné využiť iba v situácii, keď sa v *ONNX* modeli nachádza operátor `Sum` s dvomi vstupmi, za ním nasleduje operátor predstavujúci jednu z podporovaných aktivačných funkcií, a navyše žiaden ďalší operátor nepristupuje priamo na výstup operátoru `Sum`.

Informácie z tejto sekcie som odvodil z *ONNX* dokumentácie operátoru `Sum`[15], zo súborov s implementáciami inferencie *TFLite* operátorov `Add`[16] a `AddN`[17] a zo schémového súboru formátu *.tflite*.

3.1.11 Konverzia atribútov `auto_pad` a `pads`

Výrazom *padding* sa rozumie určitá výplň pri práci s tzv. jadrom operácie, ktoré je prikladané na vstupný tenzor a postupne posúvané. Udáva obmedzenia pri hraničných situáciách, kedy by toto jadro presahovalo za rozsah tenzoru, na ktorý je aplikované. Využívajú ho napríklad *ONNX* operátory `Conv`, `AveragePool`, `MaxPool`, `LpPool` a mnohé ďalšie. [12]

V *ONNX* modeloch je *výplň* reprezentovaná vždy dvojicou operátorov `auto_pad` a `pads`. `auto_pad` je obyčajný reťazec, ktorý môže nadobúdať jednu z nasledujúcich hodnôt:

- `"NOTSET"` znamená že sa použije atribút `pads`.
- `"SAME_UPPER"` znamená, že sa použije toľko výplne, koľko jadro vyžaduje. Pokiaľ by bolo nutné pridať rôzne veľkú výplň na začiatku a na konci, tá väčšia sa pridá na konci.
- `"SAME_LOWER"` znamená, že sa použije toľko výplne, koľko jadro vyžaduje. Pokiaľ by bolo nutné pridať rôzne veľkú výplň na začiatku a na konci, tá väčšia sa pridá na začiatku.
- `"VALID"` znamená že sa nepoužije žiadna výplň. Jadro nebude presahovať za hranice vstupného tenzoru.

Pokiaľ má `auto_pad` hodnotu *"NOTSET"*, použije sa pre určenie výplne atribút `pads`. Ten predstavuje zoznam nezáporných celých čísel, ktoré reprezentujú výplň na začiatku a na konci, pre každú dimenziu. Zoznam je vo formáte `[x1_začiatok, x2_začiatok, ..., x1_koniec, x2_koniec, ...]`, kde x1, x2, ... sú dimenzie vstupného tenzoru. [12]

TFLite reprezentuje výplň pomocou jedného parametra `padding`. Jedná sa o výčtový typ, s dvomi prípustnými hodnotami:

[15] Dostupné z: https://onnx.ai/onnx/operators/onnx__Sum.html

[16] Dostupné z: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/kernels/internal/reference/add.h

[17] Dostupné z: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/kernels/internal/reference/add_n.h

- `SAME` znamená, že sa použije toľko výplne, koľko jadro vyžaduje. Pokiaľ by bolo nutné pridať rôzne veľkú výplň na začiatku a na konci, tá väčšia sa pridá na konci.
- `VALID` znamená že sa nepoužije žiadna výplň. Jadro nebude presahovať za hranice vstupného tenzoru.

Možnosti konverzie medzi *ONNX* a *TFLite* reprezentáciami výplne sú opísané v tabuľke 3.6.

Hodnota *ONNX* `auto_pad`	Hodnota *TFLite* `padding`
`"SAME_UPPER"`	`SAME`
`"SAME_LOWER"`	`SAME`. Táto konverzia nie je identická. Rozdiel pri výstupe by mal byť ale zanedbateľný.
`"VALID"`	`VALID`

Tabulka 3.6: Konverzia hodnôt *ONNX* atribútu `auto_pad` na *TFLite* parameter `padding`

Pokiaľ je použitý *ONNX* atribút `pads`, je konverzia závislá na jeho konkrétnych hodnotách. Tieto hodnoty je často možné reprezentovať aj pomocou atribútu `auto_pad`, takže konverzia na *TFLite* `padding` je realizovateľná podobným postupom, ako v tabuľke 3.6. Je nutné akurát korektne identifikovať aká hodnota `auto_pad` je atribútom `pads` reprezentovaná. Na to je nutné poznať tvar vstupného tenzoru a jadra operácie, taktiež sú dôležité atribúty samotného operátoru, ako napríklad `strides` alebo `dilations`. Pokiaľ hodnoty atribútu `pads` nie je možné takýmto spôsobom namapovať na `TFLite padding`, presná konverzia nie je možná.

Poznatky v tejto sekcii týkajúce sa štandardu *ONNX* som čerpal výhradne z oficiálnej dokumentácie dostupnej z [12]. Informácie ohľadom *TFLite* operátorov som vydedukoval analýzou schémového súboru pre formát *TFLite*[18] a súboru s implementáciou funkcií pre prácu s parametrom `padding`[19].

3.2 Konverzia tenzorov

ONNX operátory spravidla využívajú formát tenzorov *NCHW*. Je to však individuálne a pre jednotlivé operátory je ich vstupný formát špecifikovaný v dokumentácii dostupnej z [12]. Naopak všetky vstavané *TFLite* operátory pracujú s tenzormi vo formáte *NHWC*. Je tak stanovené v schémovom súbore[18]. Konverzia medzi týmito formátmi je triviálna. Stačí presunúť druhú dimenziu statického tenzoru na koniec. Napríklad v jazyku *Python* knižnica *numpy* implementuje funkciu `moveaxis()`, ktorá dokáže túto operáciu vykonať. Čo už nie je triviálne, je identifikácia situácií kedy je nutné tenzory konvertovať.

Zo schémy[20] *.onnx* súboru vyplýva, že tenzory ktoré sú výstupmi operátorov a vstupmi ďalších operátorov, nemusia byť v modeli uložené. *ONNX* model používa názov tenzoru pre prepojenie operátorov. Samotný tenzor však nemusí byť definovaný. Inferenčný program si dokáže domyslieť jeho detaily ako sú tvar či dátový typ. Pri konverzii je nutné tieto tenzory vytvoriť a určiť ich vlastnosti na základe ich vzťahov s operátormi. Niektoré operátory totiž

[18] Dostupné z: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/schema/schema.fbs

[19] Dostupné z: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/kernels/padding.h

[20] Dostupné z: https://github.com/onnx/onnx/blob/main/onnx/onnx-ml.proto

potrebujú poznať presný tvar svojho vstupu, aby ich bolo možné korektne konvertovať. Jedným príkladom je operátor *Softmax*. Detaily jeho konverzie sú v sekcii 3.1.9.

Jedna zo špeciálnych situácií týkajúcich sa konverzie tenzorov je taká, že vstupom *ONNX* operátoru je tenzor vo formáte *NCHW* ale jeho výstupom je tenzor s menším počtom dimenzií než 4. Tento výstupný tenzor už nie je vo formáte *NCHW*. Význam jeho dimenzií je daný operátorom, ktorý s ním ďalej pracuje. Po konverzii tejto situácie do *TFLite* modelu bude na vstupe operátoru tenzor vo formáte *NHWC* a na výstupe bude opäť tenzor s menším počtom dimenzií. Tvary týchto výstupov budú často identické v *ONNX* aj *TFLite*, avšak samotné dáta v nich sa môžu líšiť. Napríklad pri operátore `Reshape`, výstupné tenzory v *ONNX* aj v *TFLite* obsahujú identické dáta, ale na rôznych indexoch, keďže vznikli z tenzorov s rovnakými dátami na rôznych pozíciách (*NCHW* a *NHWC*). V takýchto situáciách je nutné zabezpečiť korektnú konverziu, napríklad transformáciou vstupného tenzoru operátoru na formát *NCHW* aj v *TFLite* modeli. Táto situácia je detailnejšie analyzovaná aj sekcii 3.1.8.

Kapitola 4

Ilustrácia úlohy konverzie modelov

Táto kapitola slúži na objasnenie procesu konverzie *ONNX* modelu na *TFLite* a na overenie postupu konverzie definovaného v kapitole 3. Ako ukážkový model bol zvolený *Alexnet*[1]. Jedná sa o predtrénovaný model určený na klasifikáciu obrázkov. Tento model využíva relatívne malé množstvo operátorov, ktoré sa navyše objavujú často aj v iných modeloch. Tok dát v modeli sa nikdy nerozvetvuje, takže je jednoduchší a vhodný ako príklad. Keďže celý model je relatívne veľký, v nasledujúcom texte sa pracuje iba s jeho podmnožinou, čo je pre účely tejto kapitoly dostatočné.

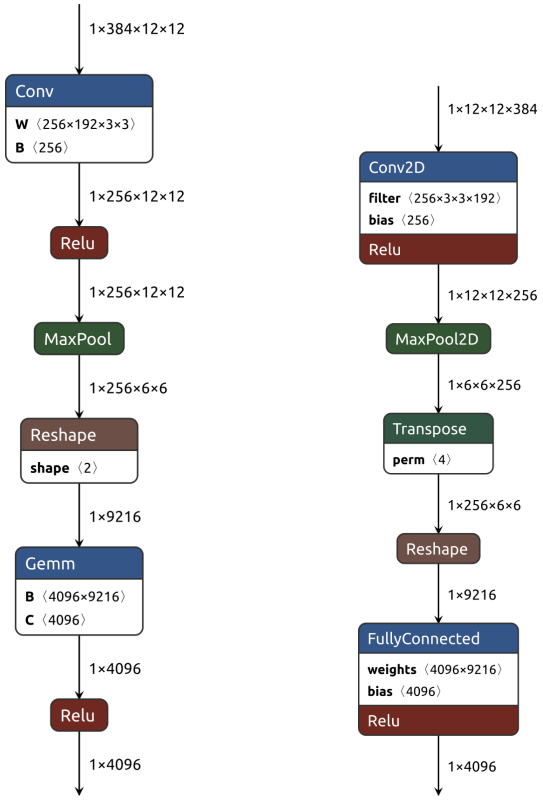
Na obrázku 4.1 je zobrazený spomínaný úsek pôvodného *.onnx* modelu, ako aj jeho ekvivalentná *.tflite* verzia. Zobrazené modely sa líšia v niekoľkých aspektoch. Tvary príslušných tenzorov v modeloch nie sú úplne rovnaké. Poradie dimenzií je odlišné. Ako je uvedené v kapitole 2, *TFLite* využíva formát NHWC a *ONNX* zas formát NCHW. Takže sa jedná o ekvivalentné tenzory, v iných formátoch. Ďalším významným rozdielom je menší počet operátorov v *TFLite* variante. *TFLite* umožňuje v niektorých prípadoch spojiť operátor s jednoduchou aktivačnou funkciou. V tomto prípade sú operátory `Conv2D` a `FullyConnected` spojené do jedného s operátormi `Relu`. Tým sa zmenší veľkosť modelu.

V *TFLite* verzii modelu sa vyskytuje operátor `Transpose`, ktorý v *ONNX* variante nie je. Jedná sa o okrajovú situáciu, kedy vstup operátoru `Reshape` má viac dimenzií, ako jeho výstup, ktorý už nie je vo formáte NCHW. V takejto situácii by po naivnej konverzii výstup operátoru `Reshape` mal síce rovnaký tvar, ale jeho hodnoty by boli na iných indexoch. Keďže výstupný tenzor predstavuje maticu, ktorá bude násobená, výstup nasledujúcich operátorov by bol nesprávny. Je nutné v *TFLite* modeli pred operátor `Reshape` vložiť operátor `Transpose`, ktorý transformuje NHWC tenzor na formát NCHW. Tým zaručí ekvivalentné chovanie konvertovaného modelu. Táto situácia je detailnejšie analyzovaná v sekcii 3.1.8.

Grafické zobrazenie výpočtových grafov oboch modelov pomocou nástroja Netron na obrázku 4.1, naznačuje požadovaný výsledok konverzie z pohľadu operátorov a tenzorov. Nezachytáva však detaily štruktúry súborov s modelmi, ako sú opísané v kapitole 2. Spomína sa v nej aj, že súbory *.onnx* a *.tflite* sú nadstavbami nad serializačnými formátmi protocol buffer a flatbuffer. Ani jeden z nich nie je textový formát, takže by nebolo užitočné zobraziť priamo ich surové dáta. Pre tento účel je vhodnejšie modely zobraziť vo formáte *JSON*.

Model z obrázku 4.1 je stále príliš veľký na to, aby sa sem jeho *JSON* reprezentácia rozumne vošla. Ďalej sa bude pracovať s jeho verziou zredukovanou na iba dva operátory. Konkrétne sú to prvé dva operátory `Conv` a `Reshape`, zobrazené na obrázku 4.2. Pre účely

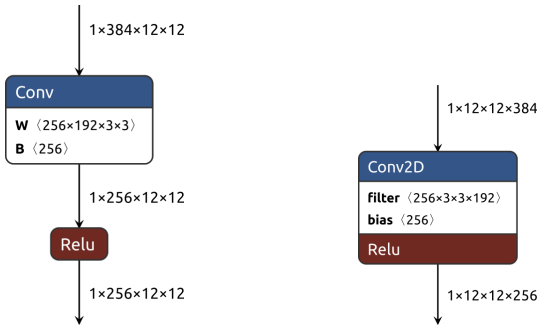
[1] Dostupné z : https://github.com/onnx/models/blob/main/vision/classification/alexnet/model/bvlcalexnet-12.onnx



(a) Redukovaný *ONNX* model

(b) Konvertovaný *TFLite* model

Obrázek 4.1: Výsledok konverzie časti *.onnx* modelu *Alexnet* na formát *.tflite*. Grafická reprezentácia modelov bola vygenerovaná nástrojom Netron.



(a) Redukovaný *ONNX* model

(b) Konvertovaný *TFLite* model

Obrázek 4.2: Výsledok konverzie dvoch operátorov *.onnx* modelu *Alexnet* na formát *.tflite*. Grafická reprezentácia modelov bola vygenerovaná nástrojom Netron.

vizualizácie boli *.onnx* a *.tflite* verzie tohto modelu následne konvertované do formátu *JSON*. Výsledky sú zdokumentované a okomentované v sekciách 4.1 a 4.2.

Poznatky ohľadom formátu *.onnx* súboru som odvodil z metadát v súbore *onnx-ml.proto* [2].

[2]Dostupné z: https://github.com/onnx/onnx/blob/main/onnx/onnx-ml.proto

4.1 *ONNX* model vizualizovaný pomocou formátu *JSON*

Súbor *.onnx* obsahujúci jednoduchý model s dvomi operátormi reprezentovaný pomocou *JSON* formátu vyzerá nasledovne.

```
1  {
2    "irVersion": "7",
3    "graph": {
4      "node": [
5        {
6          "input": ["conv4_2","conv5_w_0", "conv5_b_0"],
7          "output": ["conv5_1"],
8          "name": "",
9          "opType": "Conv",
10         "attribute": [
11           {"name": "group", "i": "2", "type": "INT"},
12           {"name": "strides", "ints": ["1","1"], "type": "INTS"},
13           {"name": "pads", "ints": ["1","1","1","1"], "type": "INTS"},
14           {"name": "kernel_shape", "ints": ["3","3"], "type": "INTS"}
15         ]
16       },
17       {
18         "input": ["conv5_1"],
19         "output": ["conv5_2"],
20         "name": "",
21         "opType": "Relu"
22       }
23     ],
```

Na riadkoch 4 až 23 sú definície operátorov modelu. Špecifický je spôsob reprezentácie atribútov operátoru `Conv`. Jedná sa o *repeated* položku `attribute`, ktorej prvky sú identifikované reťazcom `name` a typ ich hodnoty je daný položkou `type`. Vstupy a výstupy operátorov sú špecifikované reťazcami, ktoré predstavujú unikátne názvy jednotlivých tenzorov. Konkrétny typ operátoru je daný reťazcom v položke `opType`.

```
24     "initializer": [
25       {
26         "dims": ["256"],
27         "dataType": 1,
28         "name": "conv5_b_0",
29         "rawData": "W8WUPsGNZbvMz4E+peG5PpiCmz2Av2 ... "
30       },
31       {
32         "dims": ["256","192","3","3"],
33         "dataType": 1,
34         "name": "conv5_w_0",
35         "rawData": "O2l/vOfKtLwWJeS8fkYgvGseTLnHlj ... "
36       }
37     ],
```

Po operátoroch nasleduje *repeated* položka `initializer`, v ktorej sú uložené tenzory so statickými dátami. Ako je spomenuté v sekcii 2.3, existuje až 8 rôznych spôsobov uloženia

dát tenzorov. V tomto prípade je použité pole `raw_data`. Samotné dáta som iba naznačil a nezobrazil celé, pretože ich sú až jednotky MB.

```
38      "input": [
39        {
40          "name": "conv4_2",
41          "type": {
42            "tensorType": {
43              "elemType": 1,
44              "shape": {
45                "dim": [{"dimValue": "1"},{"dimValue": "384"},{"dimValue":
                      ↪ "12"},{"dimValue": "12"}]
46              }
47            }
48          }
49        }
50      ],
51      "output": [
52        {
53          "name": "conv5_2",
54          "type": {
55            "tensorType": {
56              "elemType": 1,
57              "shape": {
58                "dim": [{"dimValue": "1"},{"dimValue": "256"},{"dimValue":
                      ↪ "12"},{"dimValue": "12"}]
59              }
60            }
61          }
62        }
63      ],
64      "valueInfo": [
65        {
66          "name": "conv5_1",
67          "type": {
68            "tensorType": {
69              "elemType": 1,
70              "shape": {
71                "dim": [{"dimValue": "1"},{"dimValue": "256"},{"dimValue":
                      ↪ "12"},{"dimValue": "12"}]
72              }
73            }
74           }
75        }
76      ]
77    },
78    "opsetImport": [{"domain": "","version": "12"}]
79  }
```

Na riadkoch 38 až 63 sú špecifikované vstupné a výstupné tenzory celého grafu. Za nimi nasleduje *repeated* položka `value_info`, ktorá obsahuje definície vypočítaných tenzorov. V tomto prípade pozostáva iba z tenzoru *conv5_1*, ktorý slúži ako výstup operátoru `Conv` a zároveň ako vstup operátoru `Relu`. Model je zakončený položkou `opsetImport`, ktorá

definuje verziu *ONNX* štandardu, v ktorej bol model vytvorený. Konkrétne sa jedná o verziu *ONNX* 1.12.

4.2 *TFLite* model vizualizovaný pomocou formátu *JSON*

Pre porovnanie, *JSON* reprezentácia *TFLite* verzie modelu je nasledovná.

```
1  {
2    "version": 3,
3    "operator_codes": [
4      {
5        "deprecated_builtin_code": 3,
6        "version": 1,
7        "builtin_code": "CONV_2D"
8      }
9    ],
10   "subgraphs": [
11     {
12       "operators": [
13         {
14           "opcode_index": 0,
15           "inputs": [1,3,2],
16           "outputs": [0],
17           "builtin_options_type": "Conv2DOptions",
18           "builtin_options": {
19             "padding": "SAME",
20             "stride_w": 1,
21             "stride_h": 1,
22             "fused_activation_function": "RELU",
23             "dilation_w_factor": 1,
24             "dilation_h_factor": 1
25           },
26           "custom_options_format": "FLEXBUFFERS",
27           "mutating_variable_inputs": []
28         }
29       ],
```

Na začiatku je položka `version`, ktorá značí verziu *TFLite* schémy, použitej pre zakódovanie dát v súbore. Na riadku 3 je vektor `operator_codes`, ktorý obsahuje špecifikáciu verzie každého operátoru, ktorý sa v modeli vyskytuje. V tomto prípade je použitý iba operátor `Conv2D`. Ďalej nasleduje vektor `subgraphs`, ktorý ako je zvykom obsahuje práve jeden objekt, a to hlavný výpočtový graf modelu. Na riadkoch 14 až 28 je definícia jediného operátoru modelu, `Conv2D`. Narozdiel od *ONNX* varianty, parametre operátoru sú jasne určené v tabuľke `builtin_options`, ktorej štruktúra je jednoznačne daná hodnotou položky `builtin_options_type`. V tomto prípade sa jedná o parametre operátoru `Conv2D`, takže `Conv2DOptions`.

```
30       "tensors": [
31         {
32           "shape": [1,12,12,256],
33           "type": "FLOAT32",
34           "buffer": 0,
```

```
35          "name": "conv5_2",
36          "is_variable": false,
37          "has_rank": false
38        },
39        {
40          "shape": [1,12,12,384],
41          "type": "FLOAT32",
42          "buffer": 0,
43          "name": "conv4_2",
44          "is_variable": false,
45          "has_rank": false
46        },
47        {
48          "shape": [256],
49          "type": "FLOAT32",
50          "buffer": 1,
51          "name": "conv5_b_0",
52          "is_variable": false,
53          "has_rank": false
54        },
55        {
56          "shape": [256,3,3,192],
57          "type": "FLOAT32",
58          "buffer": 2,
59          "name": "conv5_w_0",
60          "is_variable": false,
61          "has_rank": false
62        }
63      ],
64      "inputs": [1],
65      "outputs": [0]
66    }
67  ],
```

Na riadkoch 30 až 63 je znázornená definícia všetkých tenzorov modelu vo vektore `tensors`. Oproti *ONNX* modelu, *TFLite* nerozlišuje medzi vypočítanými tenzormi a tenzormi so statickými dátami. Každý tenzor má položku `buffer`, s celočíselnou hodnotou slúžiacou ako index to vektoru `buffers` s dátami tenzoru. Vypočítané tenzory majú index na položku, ktorá má prázdne dáta. V tomto prípade je to index 0 pri tenzoroch *conv5_2* a *conv4_2*. Položky `inputs` a `outputs` na riadkoch 64 a 65 sú vektory indexov do vektoru `tensors`. Špecifikujú vstupy a výstupy výpočtového grafu.

```
68  "buffers": [
69    {},
70    {
71      "data": [
72        91,
73        197,
74        148,
75        62,
76        ...
77      ]
78    }
```

```
79   ]
80 }
```

Na konci je zobrazený vektor `buffers`, obsahujúci surové dáta tenzorov, uložené po jednotlivých bytoch v poradí *little endian*, ako sa uvádza v kapitole 2.2. V príklade zobrazujem iba prvé 4 byty, teda jednu 32-bitovú *float* hodnotu. Kompletné dáta tenzorov by totiž zaberali milióny riadkov. Z vektoru `tensors`, konkrétne z riadkov 50 a 51 vyplýva, že tieto štyri byty predstavujú prvú hodnotu tenzoru *conv5_b_0*.

Informácie ohľadom formátu *.tflite* súboru som odvodil z metadát v súbore *schema.fbs* [3].

4.3 Zhrnutie

Táto kapitola demonštruje dva problémy konverzie modelov. Prvým je hľadanie čo najefektívnejšieho spôsobu reprezentácie chovania *ONNX* modelu pomocou vstavaných *TFLite* operátorov. Táto problematika je riešená v kapitole 3. Druhým aspektom konverzie je korektná reprezentácia operátorov a tenzorov v *.tflite* súbore. Dôležité je nie len vyprodukovať syntakticky a sémanticky správny flatbuffer súbor. Taktiež je podstatné zaručiť jeho efektivitu a snažiť sa aby výsledný súbor bol pokiaľ možno čo najoptimálnejší.

Ukážkový model s dvomi operátormi z predchádzajúcich sekcií, má v oboch formátoch približne 1.8 MB. V praxi sa používajú výrazne komplikovanejšie modely. Napríklad model *Dense Upsampling Convolution*, ktorého *ONNX* verzia má 355 operátorov a približne 260.7 MB. Ručná konverzia modelov, napríklad pomocou formátu *JSON*, by bola neúnosne pracná a nepripadá do úvahy. Je nutné implementovať program, ktorý vykoná preklad medzi formátmi automaticky. Týmto sa zaoberá kapitola 5.

[3] Dostupné z: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/schema/schema.fbs

Kapitola 5

Implementácia konvertoru

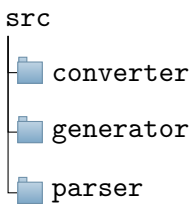
Celý program som písal v jazyku *Python*. Jazyk bol vybraný na základe jednoduchosti použitia a rýchlosti vývoja aplikácie. Dôležitým faktorom výberu bol aj fakt, že *Python* je často využívaný pre prácu s neurónovými sieťami. Existujú rôzne nástroje s rozsiahlou komunitou a veľkou podporov pre tento jazyk.

Návod na použite konvertoru je v súbore `README.md`. Repozitár s kompletnou implementáciou je verejne dostupný na *GitHube*[1]. Archív na priloženom médiu obsahuje všetky knižnice, súbory a modely potrebné pre konverziu modelov a testovanie.

Program pre konverziu medzi formátmi *ONNX* a *TensorFlow Lite* pozostáva z troch hlavných častí.

- Načítanie, kontrola, a interná reprezentácia súboru s modelom vo formáte *.onnx*.
- Interná reprezentácia *TFLite* modelu, následná kontrola a generovanie výstupného *.tflite* súboru so skonvertovaným modelom.
- Prekladač medzi internými reprezentáciami modelu.

Implementácia je rozdelená do troch hlavných adresárov v priečinku `src`, prislúchajúcich spomínaným častiam programu. Táto štruktúra je zobrazená na obrázku 5.1. Triedy implementujúce načítanie a reprezentáciu *ONNX* modelu sú v adresári `parser`. Definície tried pre reprezentáciu *TFLite* modelu a generovanie *.tflite* súboru sú uložené v priečinku `generator`. Moduly zabezpečujúce konverziu medzi internými reprezentáciami modelov sú implementované v adresári `converter`.

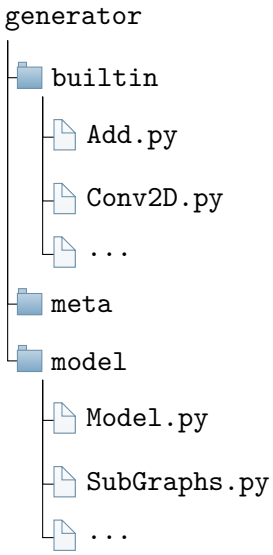


Obrázek 5.1: Rozdelenie implementácie programu do troch hlavných adresárov.

[1] Dostupné z: https://github.com/Pop-korn/ONNX2TFLite

5.1 Generovanie *.tflite* súboru

Implementácie tried umožňujúcich reprezentáciu *TFLite* modelu a jeho následné generovanie sú rozdelené do adresárovej štruktúry v priečinku `generator` (obrázok 5.2). Triedy, ktoré reprezentujú jednotlivé vstavané operátory sú v adresári `builtin`. Priečinok `meta` obsahuje definície abstraktných tried, z ktorých všetky ostatné triedy dedia. Priečinok `Model` pozostáva zo súborov s implementáciami tried, ktoré predstavujú základné prvky štruktúry *TFLite* modelu.



Obrázek 5.2: Štruktúra implementácie internej reprezentácie *TFLite* modelu a generovania výstupného súboru.

5.1.1 Práca s *flatbuffer* súborom

Ako bolo spomenuté v kapitole 2.2, formát *.tflite* je postavený na serializačnom formáte *flatbuffer* s definovanou štruktúrou uložených dát. Pre prácu s *flatbuffer* súborom program využíva *Python* knižnicu *flatbuffers*, ktorá umožňuje taký súbor generovať. S touto knižnicou úzko spolupracuje knižnica *tflite*. Tá musí byť vygenerovaná pomocou nástroja *flatc* použitím príkazu `flatc --python schema.fbs`. Knižnica *tflite* implementuje jednoduché funkcie pre tvorbu vektorov, tabuliek a štruktúr, a pre pridávanie prvkov do nich, pre konkrétnu schému. Tieto funkcie iba obaľujú funkcie knižnice *flatbuffers*. Umožňujú tak vytvárať a pridávať prvky do výsledného súboru podľa definícií v schéme, iba na základe názvu objektu. Vďaka tomu programátor nemusí riešiť detaily ako veľkosť dátových typov alebo ich zarovnanie.

Pri práci som odhalil nedostatok knižnice generovanej nástrojom *flatc* pre Python. Súbor so schémou obsahuje okrem iného aj položku `file_identifier`. Tá špecifikuje 4 byty, ktoré sa musia vyskytnúť vo výslednom *flatbuffer* súbore na bytoch 4 až 7 [4]. Knižnica by mala poskytnúť funkciu pre ukončenie tvorby modelu, ktorá obaľuje metódu *Builder.Finish()*, pričom implicitne sama špecifikuje korektný `file_identifier`. Táto funkcia však vygenerovaná nebola. Výsledný model teda nebol interpretovateľný, ani zobraziteľný v nástroji *Netron*. Reťazec `file_identifier` totiž určuje aká schéma bola použitá pre tvorbu súboru, teda aj aká schéma je potrebná pre jeho čítanie. Bez nej nie je možné identifikovať

dáta v súbore. Odhalenie tejto chyby nebolo jednoduché. Riešením je pridanie reťazca `file_identifier` ručne, pomocou metódy *Builder.Finish()*.

5.1.2 Interná reprezentácia

Pri práci s *TFLite* modelom je potrebné uchovávať nie len dáta jednotlivých prvkov, ale aj ich hierarchickú štruktúru. Je teda logické a praktické model reprezentovať objektovo. Všetky tabuľky, vektory a únie definovanú v súbore *schema.fbs* sú reprezentované práve jednou triedou. Atribúty tried predstavujú parametre týchto objektov alebo ich samotné pod-objekty. Každá trieda dedí z abstraktnej triedy *TFLObject*, ktorá deklaruje metódu *genTFLite()*. Každá trieda v nej popisuje spôsob generovania svojej reprezentácie v *.tflite* formáte. Rekurzívne pri tom volá metódu *genTFLite()* svojich pod objektov. Generovanie výstupného súboru prebehne naraz, volaním *genTFLite()* metódy koreňového objektu triedy *Model*.

Generovanie *TFLite* súboru spomínaným postupom poskytuje výhody oproti pridávaniu dát do *flatbuffer* súboru priebežne počas konverzie. Umožňuje počas analýzy a konverzie medzi modelmi postupne vytvárať objekty, zaraďovať ich do hierarchie a priebežne do nich ukladať dáta. Je možné jednoducho pristupovať k dátam modelu, či ich modifikovať.

Ďalšou výhodou spomínaného prístupu je zjednotenie a skrytie samotného generovania *flatbuffer* súboru. Programátor iba pridáva objekty do modelu a ich následným prevodom na *.tflite* súbor sa už nezaoberá. Zistil som, že poradie pridávania prvkov do *flatbuffer* súboru dokáže značne ovplyvniť veľkosť výstupného súboru. Napríklad v prípade tabuľky `Tensor`, pridanie jeho názvu skôr než jeho tvaru spôsobí zväčšenie súboru s jedným tenzorom o 12 bytov. Detaily príčiny týchto rozdielov som neskúmal. Zameral som sa iba na to, aby bol výsledný súbor čo najmenší. Experimentálne som zistil, že nárast veľkosti súboru nastáva vtedy, keď sú dáta pridávané do *flatbufferu* v inom poradí než definuje *TFLite* schéma. Skrytie pridávania prvkov do *flatbuffer* súboru garantuje optimálnu veľkosť výstupného modelu.

Nevýhodou tohto prístupu je, že model je držaný v operačnej pamäti počas celej doby chodu programu. Knižnica *flatbuffers* si počas tvorby *flatbuffer* súboru taktiež udržuje všetky dáta v pamäti, a vypisuje ich do súboru až na koniec a všetky naraz. Zároveň od načítania *.onnx* súboru je podobným spôsobom držaný v pamäti aj vstupný *ONNX* model. Vo výsledku je celý model v rôznych formách držaný v pamäti až trikrát. V prípade že by bol konvertovaný model veľmi veľký, mohlo by to spomaliť chod programu. Po diskusii s Ing. Róbertom Kalmárom z firmy NXP sme usúdili, že by to nemal byť problém.

Opisovaná interná reprezentácia *.tflite* súboru je v nasledujúcom texte nazývaná ako *objektový TFLite model*, alebo iba ako *objektový model*.

Spomínaný objektový model je izomorfný s modelom uloženým v *.tflite* súbore. Takže každý prvok v súbore je jednoznačne a rovnako reprezentovaný v objektovom modeli, a naopak. Jedná sa iba o prostriedok umožňujúci tvoriť model postupne a korektne generovať výstupný *.tflite* súbor.

5.1.3 Tvorba *TFLite* modelu

Objektový model popísaný v predchádzajúcej sekcii bol navrhnutý ako nadstavba nad štruktúrou *.tflite* súboru. Je teda určený na priamočiare generovanie výstupného súboru z jeho atribútov. To má za následok, že vytváranie tohoto objektového modelu v programe nie je úplne jednoduché. Napríklad všetky tenzory sú uložené v kolekcii *tensors* a každý operátor sa odkazuje na svoje tenzory výhradne indexom do tejto kolekcie. Podobne dáta tenzorov

sú uložené v kolekcii *buffers*, a samotný tenzor sa na svoje dáta odkazuje indexom do kolekcie. Takéto a ešte iné detaily značne znižujú použiteľnosť objektového modelu priamo, bez použitia nejakého nástroja, ktorý by model zostavoval.

Rozhodol som sa taký nástroj implementovať. Trieda `ModelBuilder` definovaná v súbore `converter/builder/ModelBuilder.py` obaľuje objektový *TFLite* model. Poskytuje metódy pre jednoduché pridávanie rôznych prvkov do modelu, pričom si sama udržuje informácie o jeho tvorbe, ktoré dokáže programátorovi správnym spôsobom sprístupniť. Pre riešenie spomínaného problému, kde *TFLite* model využíva pre referenciu indexy do kolekcií, som rozšíril objektový model o dočasné odkazy na konkrétne objekty. *ModelBuilder* pracuje počas tvorby modelu výhradne s nimi. Poskytuje metódu `finish()`, ktorá skompletizuje tvorený objektový model. Každú pomocnú referenciu na objekt nahradí indexom a nakoniec vráti kompletný objektový *TFLite* model.

Počas finalizácie modelu sú naň taktiež aplikované rôzne optimalizácie. Cieľom bolo aby sa počas konverzie programátor nemusel zaoberať efektivitou a čistotou tvoreného modelu. Z tohoto dôvodu sú v metóde `finish()` volané rôzne metódy, ktoré dokážu značne zefektívniť výsledný model. Napríklad sa eliminujú všetky nepoužívané tenzory, ktoré bolí prípadne zabudnuté počas konverzie, alebo pretrvali ešte z pôvodného *ONNX* modelu. Ďalej je všetkým tenzorom, ktoré nemajú žiadne dáta priradený jeden a ten istý prázdny *buffer*, pre úsporu veľkosti výsledného súboru. Niektoré *TFLite* operátory môžu byť priamo spojené s jednou aktivačnou funkciou. Tá je uložená ako atribút tohto operátoru a nevyžaduje samostatný operátor. Celý model je preto prehľadaný, a kde to je možné, operátory aktivačných funkcií sú týmto spôsobom odstránené. Trieda `ModelBuilder` implementuje množstvo ďalších podstatných optimalizácií.

Moduly implementujúce konverziu medzi *.onnx* a *.tflite* súbormi využívajú výhradne objekt typu *ModelBuilder*. Nikdy nepracujú s objektovým *TFLite* modelom priamo.

5.2 Načítavanie *.onnx* súboru

Ako bolo spomenuté v kapitole 2.3, formát *.onnx* je nadstavbou nad serializačným formátom *protocol buffer*. Pre načítanie *.onnx* súboru, teda *protocol bufferu* s definovanou štruktúrou dát, bola použitá špeciálna knižnica. Tá musí byť vygenerovaná za použitia schémových súborov dostupných na oficiálnej *GitHub* stránke *ONNX*[2]. Existuje viacero verzií spomínaných súborov. Pri generovaní knižnice boli použité schémy

- `onnx-ml.proto`
- `onnx-data.proto`
- `onnx-operators-ml.proto`

Pre vygenerovanie knižnice bol použitý príkaz `protoc --python_out=/lib/onnx onnx/onnx-ml.proto onnx-operators-ml.proto onnx-data.proto`. Ten vygeneruje tri súbory ktoré umožňujú prístup k dátam uloženým v súboroch, ktoré dodržujú predpisy definované v uvedených schémach.

V prípade jazyka *Python*, prístup k dátam *.onnx* súboru je zabezpečený cez takzvané *deskriptory*. Jedná sa o objekty, ktorých atribúty sú buď atomické dáta, alebo ďalšie *deskriptory*. Každý *deskriptor* reprezentuje jeden *message* alebo *enum* objekt, definovaný v schémovom súbore. Veľkou nevýhodou tohto prístupu je, že nevyužíva klasické triedy, ale *Python*

[2] https://github.com/onnx/onnx/tree/main/onnx

meta triedy. [5] To má za následok že pri programovaní v IDE, nie je *intellisense* schopný rozpoznať atribúty týchto objektov. Programovanie je značne náročnejšie a náchylnejšie na chyby, keďže programátor musí neustále hľadať v schémových súboroch, aké konkrétne atribúty objekt má a aké sú ich typy.

Táto objektová reprezentácia *ONNX* modelu je izomorfná z modelom uloženým vo formáte *Protocol Buffer*. Takže obsahuje presne tie isté *message* a *enum* objekty s tými istými atribútmi. Toto nie je ideálne pre ďalšie použitie. Napríklad v prípade tenzorov so statickými dátami, existuje až 8 rôznych spôsobov ich uloženia. Táto skutočnosť je detailne opísaná v sekcii 2.3. Pre účely konverzie medzi formátmi *ONNX* a *TFLite* je prakticky nevyhnutné reprezentovať dáta tenzoru jednotne. Taktiež v mnohých prípadoch musí mať objekt definovaný práve jeden atribút z viacerých. Pre zistenie, ktorý to je, je nutné pre každý z nich zavolať metódu deskriptoru `<deskriptor>.HasField("<názov atribútu>")`. Tento prístup nie je veľmi efektívny, je náročný na použitie a opäť vyžaduje aby programátor poznal presné názvy atribútov.

5.2.1 Interná reprezentácia

Pre skutočnosti popísané v predošlej sekcii som sa rozhodol vytvoriť vlastnú hierarchiu tried, ktoré reprezentujú *ONNX* model. Každý objekt týchto tried predstavuje jeden prvok *ONNX* modelu, obaľuje deskriptor z vygenerovanej knižnice a poskytuje rozhranie pre jednoduchý a konzistentný prístup k jeho dátam. Definície spomínaných tried sú v adresári `src/parser` (obrázok 5.3). Tento prístup rieši všetky spomínané nedostatky generovanej knižnice. Všetky atribúty objektov sú explicitne deklarované, vrátane ich typov za použitia *Python type hints*. Taktiež nezáleží na tom, akým spôsobom sú uložené dáta tenzorov. Vždy sú reprezentované ako *numpy array* v objekte typu *Tensor*.

Adresár `builtin` obsahuje definície tried, ktoré predstavujú jednotlivé *ONNX* operátory. V priečinku meta sú implementované triedy, z ktorých dedia všetky ostatné triedy. Základné prvky *ONNX* modelu sú reprezentované triedami z adresáru `model`.

Aktuálne nie je implementovaná podpora pre všetky aspekty *.onnx* súboru. Objektový *ONNX* model dokáže momentálne reprezentovať práve to, čo je konvertor schopný konvertovať. Pri rozširovaní konvertoru o nové operátory je nutné doimplementovať aj ich reprezentáciu v *ONNX* objektovom modeli. Tento proces je jednoduchý a priamočiary.

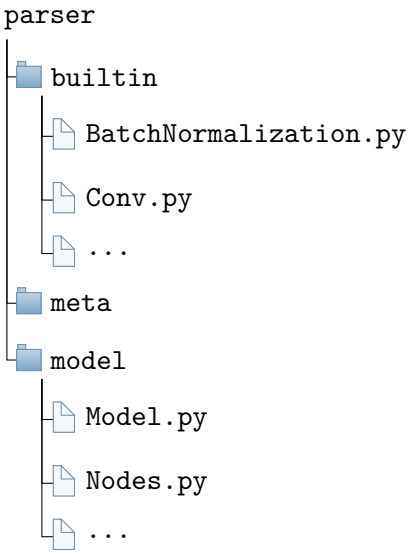
5.3 Konverzia

Úloha konverzie medzi súbormi vo formátoch *.onnx* na *.tflite* bola transformovaná na konverziu medzi objektovými modelmi popísanými v predošlých sekciách tejto kapitoly. Takže konvertor nemusí riešiť detaily vstupného a výstupného súboru, ich formáty či ich syntaktickú a sémantickú správnosť. Zaoberá sa výhradne konverziou operátorov, kvantizácie a tvaru a dát tenzorov.

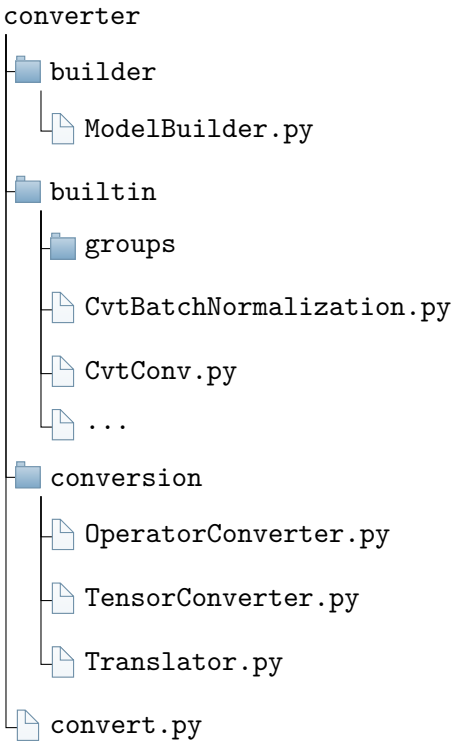
Moduly konvertoru sú rozdelené do troch častí, podľa charakteru úloh ktoré riešia.

- Analýza *ONNX* modelu a rozpoznávanie vzorov pre konverziu.
- Konverzia objektov *ONNX* modelu na *TFLite*
- Bezkontextový prekladač medzi jednoduchými a atomickými prvkami modelov

Moduly zabezpečujúce konverziu sú implementované v priečinku `converter`, zobrazeného na obrázku 5.4.



Obrázek 5.3: Štruktúra implementácie načítavania *.onnx* súboru a internej reprezentácie *ONNX* modelu.



Obrázek 5.4: Štruktúra modulov zabezpečujúcich konverziu internej reprezentácie *ONNX* modelu na *TFLite* objektový model.

5.3.1 Analýza *ONNX* modelu

Prvou časťou konverzie je rozpoznávanie prvkov v *ONNX* modeli a rozhodovanie postupu konverzie. Túto funkcionalitu zabezpečuje modul `converter/convert.py`. Jedná sa o naj-

vyššiu vrstvu konvertoru. Ako jediná pracuje s *ONNX* modelom, analyzuje ho, a na základe konkrétnych dát v ňom, volá funkcie pre konverziu a tvorbu *TFLite* modelu.

Dôležitou zodpovednosťou tejto časti je napríklad rozhodovanie, ktoré operátory sa majú konvertovať, a približne akým spôsobom. Množstvo operátorov je možné konvertovať priamo, pretože existuje ich *TFLite* verzia. Niekedy je však optimálne konvertovať celú postupnosť *ONNX* operátorov na jeden *TFLite* operátor.

Jednoduchým príkladom takéhoto postupu konverzie je postupnosť operátorov `MatMul` a `Add` v *ONNX* modeli. Tieto operátory realizujú štandardné násobenie matíc a následné pripočítanie vektoru *bias*. Napriek tomu že *ONNX* definuje operátor `Gemm`, ktorý realizuje obe operácie naraz, sa `MatMul` a `Add` v modeloch občas vyskytujú. Pri konverzii na *TFLite* môžu byť oba reprezentované jedným operátorom *FullyConnected*, čím sa zmenší veľkosť modelu.

5.3.2 Konverzia objektov

Po identifikácii objektov v *ONNX* modeli a rozhodnutí o postupe konverzie nasleduje samotná transformácia daných objektov na ich *TFLite* verziu. V programe je zodpovednosť rozdelená primárne do dvoch súborov v adresári `converter/conversion`:

- `TensorConverter.py`
- `OperatorConverter.py`

Tieto moduly predstavujú určitú strednú vrstvu konverzie. Nepracujú priamo ani so vstupným *ONNX* modelom, ani s vznikajúcim *TFLite* modelom. Na vstupe dostávajú iba útržky *ONNX* modelu, tie zanalyzujú a volajú funkcie iných modulov, ktoré zabezpečia ich konverziu na *TFLite*.

Modul `TensorConverter` implementuje funkcie, ktoré zo vstupných *ONNX* tenzorov vytvoria *TFLite* tenzory. Zachovávajú pri tom ich názov. Ich tvar, dátový typ a prípadne aj tvar uložených dát sú konvertované za použitia modulu `Translator`.

Modul `OperatorConverter` implementuje metódu `convertOperator()`, ktorá riadi konverziu jednotlivých operátorov a ich začlenenie do *TFLite* modelu. Pre každý *ONNX* operátor existuje súbor v adresári `converter/builtin`, obsahujúci implementáciu konverzie toho operátoru. `OperatorConverter` akurát určí, o aký operátor sa jedná a volá funkciu `convert()` príslušného modulu. Tá zabezpečí konverziu operátoru na ekvivalentný *TFLite* operátor, ktorý inicializuje. Často je nutné operátor konvertovať na celú skupinu nových operátorov. V takom prípade môžu tieto moduly riešiť pridávanie operátorov do *TFLite* modelu sami, za použitia objektu triedy `ModelBuilder`.

`OperatorConverter` taktiež riadi konverziu skupín operátorov, ako bolo opísané v predošlej sekcii. Volá pri tom funkcie modulov z adresáru `converter/builtin/groups`, ktoré implementujú túto konverziu. Vytvára pre nich vhodné prostredie a poskytuje týmto modulom potrebné informácie pre konverziu.

5.3.3 Bezkontextový prekladač

Na najnižšej úrovni konverzie medzi *ONNX* a *TFLite* modelmi je preklad atomických dát. Súbor `converter/conversion/Translator.py` rieši práve tento problém.

Jednou z úloh tohto modulu je preklad dátových typov. Oba formáty definujú vlastný verziu takmer všetkých štandardných typov. Jediný typ ktorý *TFLite* nepodporuje je *BFLOAT16*. Jedná sa číselný typ s plávajúcou rádovou čiarkou. Jeho názov znamená *Brain*

Floating Point Format a je často využívaný nástrojmi strojového učenia. Na exponent má vyhradených 8 bitov a na mantisu 7. Narozdiel od štandardného 16 bitového typu *FLOAT16*, ktorý má iba 5 bitový exponent a 10 bitovú mantisu. To znamená, že *BFLOAT16* má menšiu presnosť, ale zato značne väčší rozsah prípustných hodnôt. Jeho rozsah je takmer ekvivalentný s typom *FLOAT32*, keďže majú rovnako veľký exponent.[6] Aktuálna verzia konvertoru nepodporuje tenzory s dátovým typom *BFLOAT16*. V budúcnosti by o to mohol byť program rozšírený buď konverziou dát na typ *FLOAT16*, pokiaľ by boli všetky uložené hodnoty v dostatočne malom rozsahu. Realisticky však by dáta museli byť konvertované na typ *FLOAT32*. Táto konverzia by bola bezstratová a jednoduchá, ale veľkosť tenzorov v pamäti by sa zdvojnásobila.

Ďalšou úlohou bezkontextového prekladača je identifikácia formátu tenzorov a ich prípadná konverzia. Ako bolo spomínané v kapitole 2, *ONNX* model využíva prevažne tenzory, ktorých dáta sú uložené vo formáte *NCHW*. Naopak *TFLite* štandardne používa formát *NHWC*. Keďže dáta tenzorov sú reprezentované v objektových modeloch ako *numpy array*, je táto konverzia jednoduchá. Je realizovaná pomocou funkcie `moveaxis()` knižnice `numpy`.

5.4 Možnosti rozšírenia konvertoru

Ako bolo spomenuté v sekcii 3.1, konverzia operátorov je zriedka jednoduchá. Väčšinou nestačí nahradiť *ONNX* operátor ekvivalentnou *TFLite* verziou. Je nutné odhaliť mnohé okrajové situácie, ktoré si vyžadujú špecifický postup konverzie. Identifikovanie týchto situácii nie je jednoduché. Prakticky je nutné pokúšať sa konvertovať nové modely a kontrolovať validitu ich konverzie, pri čom sa odhalia nové nedostatky. Následná implementácia týchto špecifík konverzie je zvyčajne nenáročná, každopádne je ale nutná. Kvôli tejto skutočnosti nie je možné určiť, či program dokáže ľubovoľný model úspešne skonvertovať iba na základe operátorov, ktoré sa v ňom vyskytujú. Dôležité sú napríklad aj kombinácie atribútov operátorov. Taktiež je podstatné či sú vstupné tenzory niektorých operátorov statické alebo vypočítané. Prípadne aj tvary vstupných tenzorov môžu mať zásadný vplyv na postup konverzie. Keďže odhalenie všetkých takýchto situácií je prakticky nemožné, výsledný skonvertovaný model nemusí byť vždy validný. Aktuálny zoznam otestovaných modelov je k dispozícii na *GitHub* stránke projektu[3]. Z týchto dôvodov by nedávalo zmysel v rámci tejto bakalárskej práce implementovať konverziu všetkých operátorov, ktoré *ONNX* definuje. Zámer bol sústrediť sa na kvalitnú konverziu tých operátorov, ktoré sa pravidelne vyskytujú v niektorých reálnych modeloch. Program teda aktuálne podporuje konverziu približne 20 operátorov. Je to dostatočné pre konverziu mnohých skutočne používaných modelov. Jedným z postupov pre rozšírenie programu je implementácia konverzie ďalších operátorov na základe ich výskytu v reálnych modeloch.

Ako bolo spomenuté v predošlom odseku, tvar vypočítaných tenzorov môže mať zásadný vplyv na konverziu niektorých operátorov. Niektoré *.onnx* modely v sebe obsahujú definície vypočítaných tenzorov. Teda tenzorov, ktoré sú výstupmi operátorov a taktiež vstupmi ďalších operátorov. Nie je to ale povinné, a modely často tieto tenzory vôbec nedefinujú, alebo neuvedú ich očakávaný tvar. *ONNX* inferenčné programy sú schopné takýto neúplný model úspešne spustiť. *TFLite* taktiež dokáže vykonať inferenciu modelu bez špecifikovaných tvarov vypočítaných tenzorov. Avšak toto nie je ideálne pre rôzne akceleračné prvky, ktoré by vedeli tie tvary využiť. Hlavne však korektná konverzia modelov z *ONNX* na *TFLite* nie je niekedy možná, pokiaľ nie sú známe tvary vnútorných tenzorov. Príkladmi takejto situ-

[3] Dostupné z: https://github.com/Pop-korn/ONNX2TFLite

ácie sú konverzia operátorov `Reshape` (sekcia 3.1.8) alebo `Softmax` (sekcia 3.1.9). Existuje množina modelov bez špecifikovaných tvarov vnútorných tenzorov, ktoré môj program nie je momentálne schopný korektne konvertovať.

Jedným z možných riešení je implementácia odhadovania tvarov výstupných tenzorov operátorov. Operátory v *ONNX* modeli musia byť uložené v takom poradí, aby vstupy operátoru boli buď vstupmi celého grafu, alebo výstupmi predchádzajúcich operátorov. Inými slovami, operátory sú uložené v takom poradí, v akom sa budú aj vyhodnocovať. Konvertor konvertuje operátory presne v tomto poradí. Tvar výstupu operátoru je často závislý na detailoch konverzie. Tieto detaily sú v moduloch pre konverziu už odhalené a rozlíšené. Je teda možné implementovať odhadovanie tvarov výstupných tenzorov ako súčasť existujúceho kódu pre konverziu operátorov.

Druhým možným riešením je využitie existujúceho programu knižnice `onnx`[4], pre výpočet tvarov vnútorných tenzorov. Ten by musel byť spustený pred začiatkom konverzie. Vygeneroval by nový *.onnx* súbor, na ktorom by bola následne spustená konverzia. Toto riešenie je značne jednoduchšie na implementáciu a je preferovanou možnosťou rozšírenia. Určitým problémom je skutočnosť, že tento program zlyhá, pokiaľ vstupný model nie je korektne definovaný. Napríklad ak sú použité operátory z rôznych verzií *ONNX*. Tieto modely je však často možné spustiť a keby mali definované vnútorné tenzory, aj konverzia by bola možná.

Ďalšou možnosťou rozšírenia konvertoru je implementácia podpory *broadcastovania* tvarov *ONNX* tenzorov. Niektoré *ONNX* operátory môžu pracovať so vstupnými tenzormi o rôznych tvaroch, pokiaľ je možné tie vstupy *broadcastovať* na rovnaký tvar. *ONNX* podporuje dva typy *broadcastovania*, jednosmerné a viac-smerné. Pri viac-smernom *broadcastovaní* sa tvary všetkých tenzorov upravia na jeden rovnaký tvar. Napríklad ak vstupné tenzory majú tvary $(1, 4, 5)$ a $(2, 3, 1, 1)$, oba budú transformované na tvar $(2, 3, 4, 5)$. Pri jednosmernom *broadcastovaní* iba jeden tenzor prispôsobuje druhému. Napríklad pri vstupoch o tvaroch $(2, 3, 4, 5)$ a (5), sa druhý tenzor transformuje na tvar prvého. [2]

Aktuálne môj program podporuje iba základnú formu *broadcastovania* tvarov tenzorov. Keď sa v *.onnx* modeli objaví operátor, ktorý využíva pokročilejšie *broadcastovanie*, konverzia zlyhá, alebo nebude možné vykonať inferenciu výstupného modelu keďže nie je korektný.

[4]Dostupné z: https://github.com/onnx/onnx/tree/main/onnx/shape_inference

Kapitola 6

Testovanie

Testy sú implementované v module *conversion_test.py* v koreňovom adresári projektu. Tento súbor obsahuje iba funkcie, ktoré umožňujú spúšťať *ONNX* aj *TFLite* modely s náhodnými alebo predpripravenými vstupmi. Taktiež obsahuje implementácie funkcií, ktoré dokážu z reálneho *ONNX* modelu odstrániť nežiadúce operátory a tenzory, prestaviť vstupy a výstupy tohto okresaného modelu a uložiť ho. To umožňuje testovať iba určitú časť modelu, často iba jeden vybraný operátor. Bez tejto techniky by bolo ťažké odhaliť, kde presne je problém, pokiaľ celý model po konverzii neprodukuje rovnaké výsledky ako pôvodný model.

Testovanie validity konvertovaných modelov je zhrnuté v sekcii 6.1. Sú tu porovnávané výstupy modelov pred a po konverzii pri náhodných vstupných hodnotách. V sekcii 6.2 je porovnávaná doba inferencie modelov pred a po konverzii. Slúži pre testovanie efektivity konvertovaných modelov a pre overenie prínosu samotnej konverzie. Sekcia 6.3 zhŕňa výsledky testov s reálnymi dátami.

6.1 Porovnávanie výstupov konvertovaných modelov pri náhodných vstupoch

Jedným s vykonaných testov validity konvertoru je testovanie odlišnosti výstupu skonvertovaného modelu od výstupu pôvodného modelu, keď na vstupe sú identické náhodné dáta. Postup testov je nasledovný. *ONNX* model sa prekonvertuje na *TFLite*. Následne sa vygeneruje tenzor s náhodnými dátami, ktoré sú predané ako vstup jednotlivým modelom. Tento tenzor má tvar ktorý očakáva *ONNX* model a pre *TFLite* je príslušne transformovaný do formátu *NHWC*. Podobne výstupný tenzor *TFLite* modelu je transformovaný do formátu *NCHW*, aby bol jednotný s *ONNX*. Väčšinou je výstupný tenzor iba dvoj dimenzionálny, takže táto konverzia nie je nutná. Výstup by mal byť identický s výstupom *ONNX* modelu. Z týchto výstupných tenzorov sú následne automaticky vypočítané rôzne hodnoty a štatistiky pre porovnanie.

Tento test zabezpečuje funkcia `testConversion()`, ktorá je implementovaná v module `conversion_test.py`. Bola spustená na niekoľkých verejne dostupných natrénovaných modeloch. Konkrétne šlo o klasifikačný model *Alexnet*[1], model pre detekciu objektov *Tiny*

[1] Dostupné z: https://github.com/onnx/models/blob/main/vision/classification/alexnet/model/bvlcalexnet-12.onnx

YOLO v2[2], segmentačný model *Dense Upsampling Convolution (DUC)*[3] a model *Speech Classifier*[4] určený pre rozpoznávanie slov v hlasových nahrávkach, ktorý bol v spolupráci s firmou *NXP* vygenerovaný na základe online návodu[5]. Výsledky sú zhrnuté v tabuľkách 6.1, 6.2, 6.3 a 6.4. Zobrazené hodnoty sú priemerom zo 100 iterácií testov nad rôznymi náhodnými vstupmi.

Model	Najväčšia absolútna chyba	Najväčšia relatívna chyba
Alexnet	$1.173466 \cdot 10^{-8}$	$2.602906 \cdot 10^{-4}$ %
Tiny YOLO v2	$2.083778 \cdot 10^{-5}$	3.710334 %
DUC	$1.800101 \cdot 10^{-8}$	3.212610 %
Speech Classifier	$3.552436 \cdot 10^{-6}$	$1.308060 \cdot 10^{-4}$ %

Tabulka 6.1: Maximálny rozdiel medzi príslušnými prvkami výstupných tenzorov *ONNX* modelu a skonvertovaného *TFLite* modelu pre rovnaké vstupné dáta. Príslušnými prvkami sa rozumejú prvky vo výstupoch modelov na rovnakých indexoch. Implicitne sa predpokladá rovnaký formát výstupných tenzorov. Konkrétne je vyobrazený najväčší absolútny rozdiel príslušných prvkov a taktiež najväčší relatívny rozdiel medzi príslušnými prvkami.

Model	Priemerná absolútna chyba	Priemerná relatívna chyba
Alexnet	$4.642541 \cdot 10^{-10}$	$3.993724 \cdot 10^{-5}$ %
Tiny YOLO v2	$1.823265 \cdot 10^{-6}$	$1.063845 \cdot 10^{-3}$ %
DUC	$3.262347 \cdot 10^{-10}$	$9.271910 \cdot 10^{-2}$ %
Speech Classifier	$5.922135 \cdot 10^{-7}$	$1.503717 \cdot 10^{-5}$ %

Tabulka 6.2: Priemerná chyba medzi príslušnými prvkami výstupov modelov. Príslušnými prvkami sa rozumejú prvky vo výstupoch modelov na rovnakých indexoch. Vyobrazený je priemerný absolútny rozdiel a priemerný relatívny rozdiel príslušných prvkov výstupov modelov.

Z nameraných výsledkov vyplýva, že pôvodný *ONNX* model a skonvertovaný *TFLite* model neprodukujú 100% identický výstupy pri rovnakých vstupoch. Sú však takmer rovnaké, vzniknutá chyba je vždy nesmierne malá. Po konzultácii s pánom Ing. Róbertom Kalmárom z firmy NXP sme usúdili, že potencionálnym dôvodom pre rozdielne výstupy môžu byť akumulované chyby pri práci s hodnotami s plávajúcou rádovou čiarkou. Pri konverzii niektorých operátorov konvertor predpočíta niektoré statické hodnoty v dobe konverzie. Napríklad v prípade konverzie operátorov *LRN* (sekcia 3.1.6) alebo *BatchNormalization* (sekcia 3.1.1). Tieto predpočítané hodnoty sú často uložené ako dátový typ *float32*, kdežto v *ONNX* modeli, kde sa počítajú až v dobe inferencie sa potenciálne môžu medzivýsledky

[2]Dostupné z: https://github.com/onnx/models/blob/main/vision/object_detection_segmentation/tiny-yolov2/model/tinyyolov2-8.onnx

[3]Dostupné z: https://github.com/onnx/models/blob/main/vision/object_detection_segmentation/duc/model/ResNet101-DUC-12.onnx

[4]Dostupné z: https://github.com/Pop-korn/ONNX2TFLite/blob/main/data/onnx/speech_command_classifier_trained.onnx

[5]Dostupné z: https://pytorch.org/tutorials/intermediate/speech_command_classification_with_torchaudio_tutorial.html

Model	Maximálna chyba strednej hodnoty	%	Priemerná chyba strednej hodnoty	%
Alexnet	$3.492459 \cdot 10^{-10}$	$3.492459 \cdot 10^{-5}$ %	$2.328306 \cdot 10^{-10}$	$2.328306 \cdot 10^{-5}$ %
Tiny YOLO v2	$5.960464 \cdot 10^{-8}$	$1.996670 \cdot 10^{-5}$ %	$2.384185 \cdot 10^{-8}$	$7.985611 \cdot 10^{-6}$ %
DUC	$9.720225 \cdot 10^{-12}$	$2.954962 \cdot 10^{-3}$ %	$8.168399 \cdot 10^{-12}$	$2.4832028 \cdot 10^{-3}$ %
Speech Classifier	$9.536743 \cdot 10^{-7}$	$2.226101 \cdot 10^{-5}$ %	$9.536743 \cdot 10^{-8}$	$2.220202 \cdot 10^{-6}$ %

Tabulka 6.3: Maximálna a priemerná absolútna chyba medzi strednými hodnotami výstupov pôvodných a konvertovaných modelov. Taktiež vyjadrená v percentách.

Model	Maximálna chyba smerodajnej odchýlky	%	Priemerná chyba smerodajnej odchýlky	%
Alexnet	$4.074536 \cdot 10^{-10}$	$5.729276 \cdot 10^{-5}$%	$1.455191 \cdot 10^{-10}$	$2.031822 \cdot 10^{-5}$%
Tiny YOLO v2	$4.768371 \cdot 10^{-7}$	$2.346760 \cdot 10^{-5}$%	$2.384185 \cdot 10^{-7}$	$1.173152 \cdot 10^{-5}$%
DUC	$3.173284 \cdot 10^{-11}$	$1.840316 \cdot 10^{-2}$%	$2.036415 \cdot 10^{-11}$	$1.180983 \cdot 10^{-2}$%
Speech Classifier	$2.384185 \cdot 10^{-7}$	$1.909996 \cdot 10^{-5}$%	$5.960464 \cdot 10^{-8}$	$4.787680 \cdot 10^{-6}$%

Tabulka 6.4: Maximálna a priemerná absolútna chyba medzi smerodajnými odchýlkami výstupov pôvodných a konvertovaných modelov. Taktiež vyjadrená v percentách.

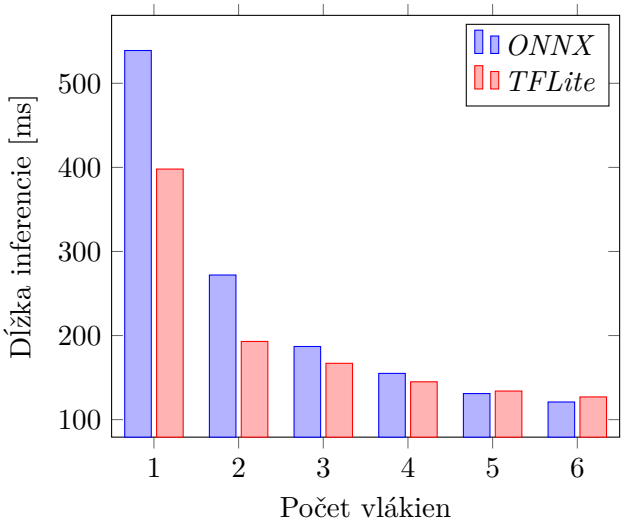
držať v registroch s väčším počtom bitov. Taktiež implementácie inferenčných programov *TFLite* a *ONNX* sa prirodzene odlišujú. Je možné že rôzne algoritmy a teda odlišné poradie operácií s plávajúcou rádovou čiarkou taktiež vnášajú chyby. Celkovo nepovažujeme namerané výsledky ako dôvod pochybovať o validite konverzie.

Možným dôvodom vyšších relatívnych chýb výstupov modelu *DUC* môže byť fakt, že pozostáva až z 355 operátorov. Pre porovnanie, *Alexnet* má 24, *Tiny YOLO v2* 32 a *Speech Classifier* má 22 operátorov. V modeli *DUC* existuje teda viac príležitostí pre aproximačné chyby, ktoré sa akumulujú.

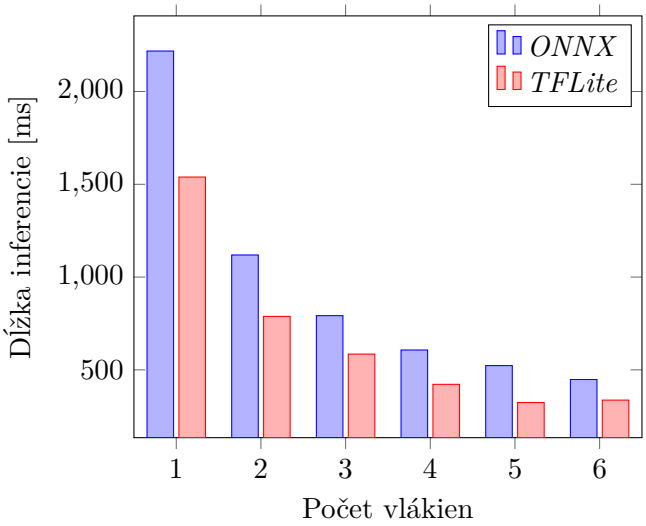
Dôvodom pre vyššie absolútne chyby modelov *Tiny YOLO v2* a *Speech Classifier* je skutočnosť, že ich posledným operátorom nie je `Softmax`. Ten normalizuje svoj vstupný tenzor do malého rozsahu a súčet hodnôt jeho výstupu je štandardne rovný 1.0. Väčšina ostatných operátorov svoj výstup nenormalizuje. Rozsah hodnôt ich výstupných tenzorov je teda značne väčší. Z tohto dôvodu sú absolútne chyby výstupov týchto modelov väčšie, pri porovnateľnej relatívnej chybe.

6.2 Experimenty na cieľovej platforme

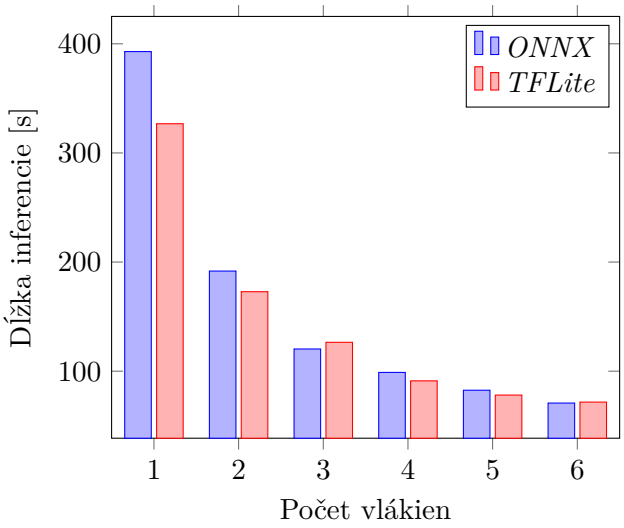
Pre testovanie vplyvu konverzie modelov na rýchlosť inferencie, boli v spolupráci s firmou NXP vykonané testy sa cieľovej platforme. Konkrétne bolo vybrané zariadenie s procesorom *i.MX 8QuadMax*. Experimenty boli vykonané na modeloch *Alexnet*, *Tiny YOLO v2*, *Dense Upsampling Convolution (DUC)* a *Speech Classifier*, rovnako ako v predošlej sekcii. *ONNX* modely boli skonvertované na *TFLite*. Pri každom experimente bol stanovený počet povolených vlákien, ktorý sa pohyboval v rozsahu 1 až 6. Jednotlivé verzie modelov boli spustené na cieľovom zariadení s meniacim sa počtom povolených vlákien a doba potrebná pre inferenciu modelov bola zaznamenaná. Výsledky experimentov sú zachytené v grafoch 6.1, 6.2, 6.3 a 6.4.



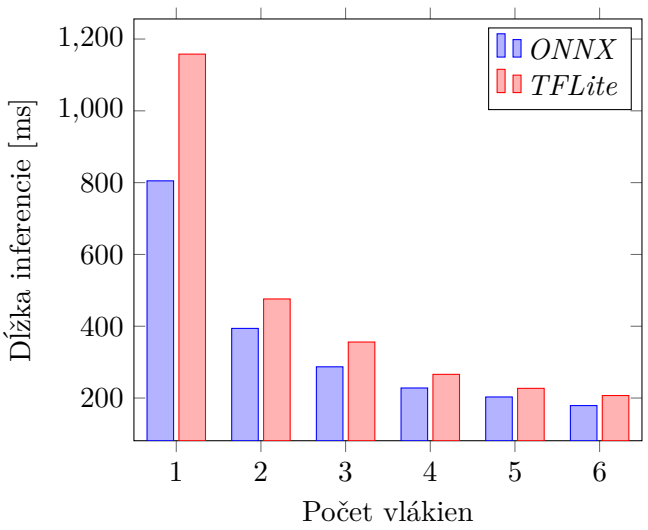
Obrázek 6.1: Porovnanie dĺžky doby inferencie modelu *Alexnet* pred a po konverzii. Experiment bol spustený na 1 až 6 vláknach naraz.



Obrázek 6.2: Porovnanie dĺžky doby inferencie modelu *Tiny YOLO v2* pred a po konverzii. Experiment bol spustený na 1 až 6 vláknach naraz.



Obrázek 6.3: Porovnanie dĺžky doby inferencie modelu *DUC* pred a po konverzii. Experiment bol spustený na 1 až 6 vláknach naraz.

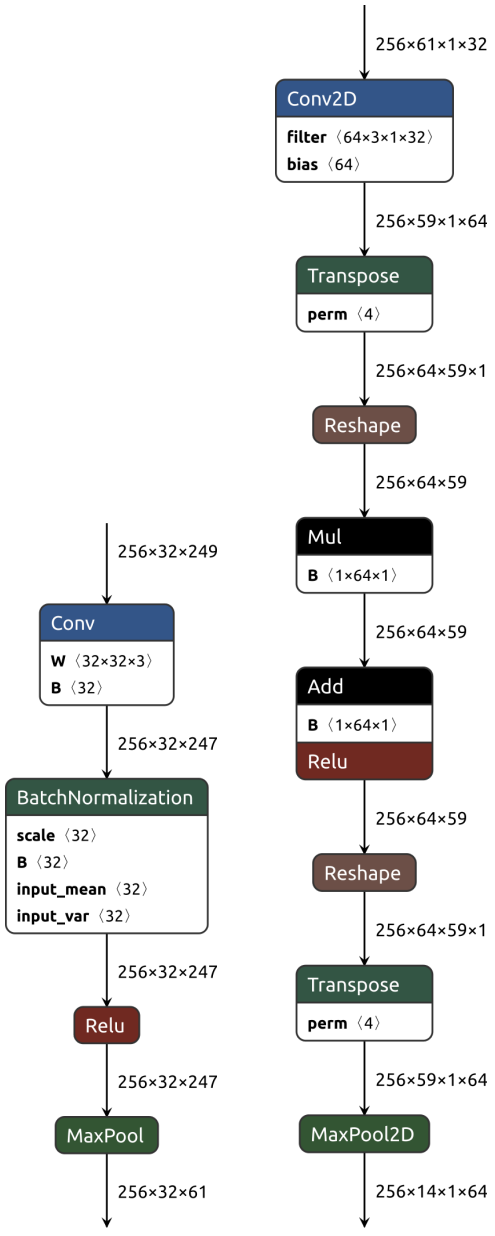


Obrázek 6.4: Porovnanie dĺžky doby inferencie modelu *Speech Classifier* pred a po konverzii. Experiment bol spustený na 1 až 6 vláknach naraz.

Z nameraných výsledkov vyplýva, že inferencia *TFLite* verzií modelov bola značne rýchlejšia, než pri pôvodných *ONNX* modeloch. Najväčší rozdiel je pozorovateľný pri menšom počte povolených vlákien. *TFLite* verzie modelov *Alexnet* a *Tiny YOLO v2* dosahovali pri použití 1 a 2 vlákien zrýchlenie približne 30%. Pri inferencii modelu *Tiny YOLO v2* s 5 vláknami bolo dokonca dosiahnuté zrýchlenie až 38%.

Výnimkou je model *Speech Classifier*, kde inferencia konvertovaného *TFLite* modelu bola pomalšia než pri pôvodnom *ONNX* modeli (obrázok 6.4). Dôvodom sú pridané operátory `Transpose` a `Reshape`. *ONNX* verzia modelu obsahuje operátory `Conv`, `MaxPool` a `AveragePool` s 1-dimenzionálnymi jadrami operácie. Ako je opísané v sekcii 3.1.7, konverzia je možná jedine rozšírením tenzorov o jednu dimenziu pomocou operátorov `Transpose` a `Reshape`, a následným použitím 2D verzie pôvodného operátoru. Model obsahuje sek-

vencie takýchto operátorov, ktoré sú prerušované operátormi `BatchNormalization` a `Relu` (`Add` a `Mul` v *TFLite* modeli). Príklad tejto situácie je na obrázku 6.5.



(a) *ONNX* (b) TFLite

Obrázek 6.5: Konverzia operátorov `Conv` a `MaxPool` s 1-dimenzionálnym jadrom, medzi ktorými sú operátory `BatchNormalization` a `Relu`. Grafická reprezentácia bola vygenerovaná nástrojom *Netron*.

Tento postup konverzie je korektný a výsledný model produkuje rovnaké výstupy. V tejto konkrétnej situácii je možné operátory `BatchNormalization` a `Relu` (`Add` a `Mul` v *TFLite* modeli) staticky rozšíriť na štyri dimenzie a odstrániť operátory `Transpose` a `Reshape`. Táto optimalizácia nie je momentálne implementovaná a je súčasťou budúceho rozšírenia práce.

6.3 Analýza použiteľnosti konvertovaných modelov s reálnymi dátami

Pre overenie použiteľnosti konvertovaných modelov v reálnych situáciách boli implementované testy, s reálnymi vstupnými dátami, pre ktoré boli pôvodné modely vyvinuté. Modul `conversion_test.py` definuje funkciu `testConversionWithInputs()`. V nej sa daný *ONNX* model skonvertuje na *TFLite* a obe verzie sú spustené s konkrétnymi vstupnými hodnotami. Následne sú porovnané výstupy oboch modelov. Funkcia umožňuje porovnávať indexy prvkov s najvyššími hodnotami, ako aj definovať maximálnu akceptovateľnú chybu výstupov. Na základe úspešnosti vykonaných testov je možné usúdiť, že modely po konverzii na formát *TFLite* produkujú rovnaké výstupy ako pôvodné *ONNX* modely. Takže konvertor je možné použiť pre reálne aplikácie.

Kapitola 7

Záver

Hlavným cieľom tejto práce bola analýza formátov pre reprezentáciu hlbokých neurónových sietí *ONNX* a *TensorFlow Lite (TFLite)*, a následný návrh a implementácia konvertoru medzi modelmi v týchto formátoch.

V texte práce je zdokumentovaná štruktúra *.onnx* a *.tflite* súborov. Jediným zdrojom v tejto oblasti sú schémové súbory s metadátami, ktoré som analyzoval a odvodil z nich hierarchiu modelov oboch formátov. *ONNX* operátory sú kvalitne zdokumentované na oficiálnej stránke. Na druhej strane, chovanie *TFLite* operátorov a význam ich parametrov som odvodzoval analýzou `c++` súborov s implementáciami inferencie jednotlivých operátorov. Nadobudnuté poznatky som detailne zhrnul v samostatnej kapitole a na ich základe som navrhol postup konverzie *ONNX* operátorov na *TFLite*. Následne som implementoval program v jazyku `Python`, ktorý načíta *.onnx* súbor a identifikuje v ňom jednotlivé objekty. Tie s využitím navrhnutého postupu konverzie transformuje na ekvivalentné objekty *TFLite* modelu. Tento model je ďalej optimalizovaný a serializovaný do výstupného *.tlite* súboru.

Implementovaný konvertor sa podarilo úspešne verifikovať na reálnych modeloch pre klasifikáciu obrázkov, segmentáciu, detekciu objektov a analýzu akustických dát. Tieto modely po konverzii produkujú rovnaké výstupy ako pôvodné *ONNX* varianty. Veľkým úspechom sú aj výsledky experimentov na cieľovej platforme. Tie vykazujú značné zvýšenie rýchlosti inferencie konvertovaných modelov voči pôvodným. Najväčšie zaznamenané zrýchlenie bolo až 38% v prípade inferencie modelu *Tiny YOLO v2* s piatimi vláknami. Taktiež veľkosť niektorých modelov sa po konverzii podarilo zredukovať. Napríklad *TFLite* varianta modelu *Dense Upsampling Convolution* je o 420kB menšia než pôvodný *ONNX* model.

Identifikácia a implementácia hraničných prípadov konverzie operátorov je náročná. Zámerom práce bolo sústrediť sa na kvalitnú konverziu menšej množiny operátorov, ktoré sa objavujú v mnohých modeloch. Hlavným priestorom pre rozšírenie programu je implementácia konverzie ďalších operátorov, ktoré sa vyskytujú v nových modeloch. Aktuálne taktiež nie je podporovaná konverzia kvantizovaných modelov. Prvotným cieľom bola konverzia modelov s plávajúcou rádovou čiarkou a overenie použiteľnosti navrhnutého postupu konverzie. Mám v pláne naďalej pracovať na projekte v spolupráci s firmou NXP a implementovať podporu kvantizovaných modelov.

Táto práca bola prijatá na konferenciu *Excel@FIT*. Plagát, pomocou ktorého bola práca prezentovaná je v prílohe A. Príloha B obsahuje stručný komentár k plagátu a abstrakt celej práce.

Literatura

[1] Dumoulin, V. a Visin, F. *A guide to convolution arithmetic for deep learning*. 2018. Dostupné z: https://arxiv.org/abs/1603.07285.

[2] Fang, L. *Broadcasting in ONNX*. Feb 2018. [Accessed 5-Apr-2023]. Dostupné z: https://github.com/onnx/onnx/blob/main/docs/Broadcasting.md.

[3] Goodfellow, I., Bengio, Y. a Courville, A. *Deep Learning*. Online. MIT Press, 2016. ISBN 9780262035613. http://www.deeplearningbook.org.

[4] Google. *FlatBuffers: Writing a schema*. Google, 2022. [Accessed 11-Feb-2023]. Dostupné z: https://google.github.io/flatbuffers/md__schemas.html.

[5] Google. *Protocol Buffer Basics: Python*. 2023. [Accessed 4-Mar-2023]. Dostupné z: https://protobuf.dev/getting-started/pythontutorial/.

[6] Higham, N. *What is bfloat16 arithmetic?* Jun 2020. [Accessed 4-Mar-2023]. Dostupné z: https://nhigham.com/2020/06/02/what-is-bfloat16-arithmetic/.

[7] Intel. *Understanding memory formats*. 2016. [Accessed 6-Apr-2023]. Dostupné z: https://www.intel.com/content/www/us/en/docs/onednn/developer-guide-reference/2023-0/understanding-memory-formats.html.

[8] Ioffe, S. a Szegedy, C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *CoRR*. 2015, abs/1502.03167. Dostupné z: http://arxiv.org/abs/1502.03167.

[9] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M. et al. *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*. 2017. Dostupné z: https://arxiv.org/abs/1712.05877.

[10] Krizhevsky, A., Sutskever, I. a Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F., Burges, C., Bottou, L. a Weinberger, K., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012, sv. 25. Dostupné z: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[11] ONNX. *Quantize ONNX models*. [Accessed 3-May-2023]. Dostupné z: https://onnxruntime.ai/docs/performance/model-optimizations/quantization.html.

[12] ONNX. *ONNX Operators*. 2023. [Accessed 17-Mar-2023]. Dostupné z: https://onnx.ai/onnx/operators/index.html.

[13] ONNX. *Onnx/onnx: Open standard for machine learning interoperability.* 2023. [Accessed 3-Apr-2023]. Dostupné z: https://github.com/onnx/onnx.

[14] SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I. a SALAKHUTDINOV, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research.* 2014, sv. 15, č. 56, s. 1929–1958. Dostupné z: http://jmlr.org/papers/v15/srivastava14a.html.

[15] TENSORFLOW. *Tensorflow Lite delegates.* Oct 2021. [Accessed 1-Apr-2023]. Dostupné z: https://www.tensorflow.org/lite/performance/delegates.

[16] TENSORFLOW. *TensorFlow Lite.* May 2022. [Accessed 11-Feb-2023]. Dostupné z: https://www.tensorflow.org/lite/guide.

Příloha A

Excel@FIT Plagát

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2023

NXP

Converter between formats of Deep Neural Network models on mobile platforms

Martin Pavella, xpavel39@stud.fit.vutbr.cz

supervisor: Ing. **Radek Kočí**, PhD.

consultant: Ing. **Róbert Kalmár** of NXP

1. The need for conversion

- High popularity and availability of Deep Neural Network (DNN) models in the **ONNX** format
- Superior HW accelerator support on mobile platforms for models in the **TFLite** format
- Existing converters produce sub-optimal models with unnecessary operators due to indirect approach

2. Proposed solution

A **direct** converter from ONNX to TFLite

- Represent an ONNX model using a hierarchy of objects
- Convert it to an equivalent TFLite object model (Fig.1)
- Serialize the model to the output TFLite file

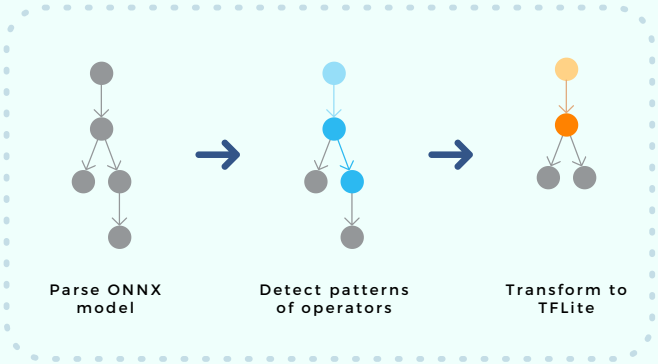
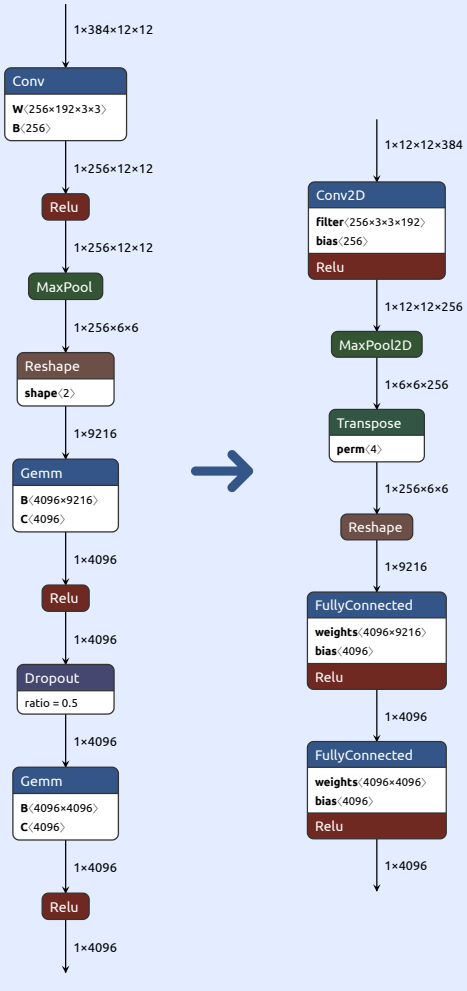


Fig. 1
Process of model conversion

3. Results

- Operator conversion is a complex and evolving problem
- Conversion of all operators is not feasible -> Focus on a subset of commonly used operators
- Successful conversion of models used for classification, object detection, segmentation and analysis of acoustic data
- Model **size reduction** by up to 420kB

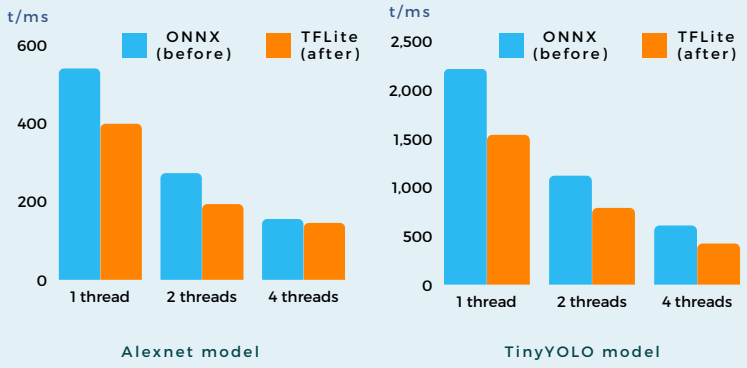


ONNX model — Equivalent **TFLite** model

Fig. 2
Example of a section of a convolutional Alexnet model before and after conversion

4. Impact on inference

- Converted models produce identical outputs as the original ones
- Experiments in collaboration with the NXP company show a significant improvement of inference speed on target platforms



Alexnet model — TinyYOLO model

Tab. 1
The time duration of DNN model inference on target platforms before and after conversion

5. Limitations

- Limited subset of supported operators
- Conversion is not always possible
- Some models can be converted, but are not efficient

Příloha B

Excel@FIT Abstrakt

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2023
http://excel.fit.vutbr.cz

Converter between formats of Deep Neural Network models on mobile platforms

Martin Pavella*

Abstract

One of the most popular formats for representing *Deep Neural Network* (DNN) models is *ONNX*. The development of drivers for HW accelerators on embeded systems is expensive and *ONNX* is rarely supported. The necessary SW support is typically only implemented for the *TensorFlow Lite* (*TFLite*) DNN model format. Currently the options for conversion of pre-trained *ONNX* models to *TFLite* are inadequate and produce sub-optimal models.

The result of my work is a direct converter of *ONNX* models to *TFLite* which focuses on producing as efficient models as possible. The program was verified on DNN models for image classification, object detection, segmentation and acoustic data analysis. The converted models produce identical outputs as the original ones and experiments show a significant improvement in inference speed.

*xpavel39@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

ONNX is a popular format for representing Deep Neural Network (DNN) models. HW accelerators for inference on mobile and embeded systems are however typically only supported by models in the *TensorFlow Lite* (*TFLite*) format.

A DNN model can often be represented in either format. Converting a pre-trained *ONNX* model to *TFLite* can provide a significant increase in inference speed as well as a reduction in model size.

Current solutions[1] for such conversion first convert the *ONNX* model to the *TensorFlow* format and then apply the *TensorFlow* to *TFLite* converter[2] by *Google*. This approach often introduces unwanted artifacts and restrictions for tensor quantisation.

My solution is a direct converter of *ONNX* models to *TFLite*. It bypasses the *TensorFlow* to *TFLite* converter, which allows for total control over the resulting model. This approach generates efficient models which produce identical outputs as the original ones. The improved utilization of HW accelerators has significantly increased the inference speed of models after conversion.

The program was developed and tested in collaboration with the *NXP* company.

2. The need for conversion

Many frameworks for training and inference of DNN models use the *ONNX* format, as it is widely supported and open source. Special hardware accelerators called *Neural Processing Units* (NPU) are being designed to accelerate costly operations such as matrix multiplication or convolution on mobile and embeded devices. The development of drivers and supporting SW for them is very expensive and typically a given NPU is only supported by *TFLite*. As these accelerators can significantly increase the speed of model inference, the motivation to convert *ONNX* models to the *TFLite* format is strong.

Current solutions take a shortcut by first converting to the *TensorFlow* format and then using the existing converter to *TFLite* by *Google*. This approach takes away the ability to precisely control the output model, which sometimes results in sub-optimal models. For instance *ONNX* uses two operators to represent depth-wise convolution while *TFLite* defines a single dedicated operator. The converters often

[1] For example: https://github.com/PINTO0309/onnx2tf or https://github.com/onnx/onnx-tensorflow

[2] https://www.tensorflow.org/lite/models/convert/convert_models

avoid this optimization. Sometimes extra operators are added to transform tensors dynamically during inference even when static transformation during conversion is possible.

3. Proposed solution

My approach was to design a direct converter from *ONNX* to *TFLite* which provides total control over the output model. It works by deserializing the input *ONNX* model and representing it as a hierarchy of objects. A part of this hierarchy is a directed computational graph, where each node represents an operator. As suggested on **Fig. 1**, this graph is analyzed for patterns of operators which are then converted and added to the newly forming *TFLite* model. *TFLite* uses a directed computational graph to represent its DNN as well, but it defines its own set of operators with different behavior. The conversion between operators of these formats is the main challenge of model conversion. The operators have complex behavior and edge cases are difficult to identify and implement. Operator conversion is a complicated and evolving problem as new operators are constantly updated and new ones are being introduced.

Both formats use different ways to represent tensors. *ONNX* uses NCHW while *TFLite* only supports NHWC. These formats define how tensor dimensions are stored in memory. Static tensors must be transformed during conversion and dynamic tensors sometimes need to be reshaped using operators.

4. Results

The converter has been validated on models used for classification, object detection, segmentation and analysis of acoustic data.

The example on **Fig. 2** shows a selected portion of the *Alexnet* model. It is a convolutional DNN for image classification [1] and can be relatively efficiently represented in *TFLite*. The *TFLite* model has a *Transpose* operator, which is not present in the original *ONNX* version. This is because the following *Reshape* operator flattens the tensor into a 2D matrix. In the *ONNX* model, the tensor is flattened from the NCHW format. Flattening a tensor from a different format would cause its values to be at different positions in the matrix, which would cause undefined behavior of the following operators.

The files with *TFLite* models use a binary *flatbuffer* format, which can result in smaller size then in the case of *ONNX*, which uses *protocol buffers*. However the need to add extra operators often cancels out the savings. In the case of an efficiently converted model, the file size has been reduced by up to 420 kB.

5. Impact on inference

The converted models produce nearly identical outputs as their *ONNX* versions. The absolute errors tend to be in the range of 10^{-6} to 10^{-12} depending on the model, which can be attributed to accumulated errors of floating point calculations.

With the help of the NXP company and their specialized HW, we were able to test the effects of model conversion on inference speeds on target platforms. **Tab. 1** shows a significant improvement in the time it takes to complete inference of selected models. The *TFLite* variants display up to 30% faster inference speeds and the effects are most noticeable when a smaller number of threads is used.

6. Limitations

ONNX supports over 140 operators. Implementing the conversion of all of them is far beyond the scope of this work. The aim was to focus on a smaller subset of operators which are used by many models. So the introduction of a new model will occasionally require implementation of the conversion of new operators.

ONNX is a more complex format which defines various data structures and powerful operators. As a result, not every model can be represented using *TFLite's* limited expressive options. In some cases it is possible to convert a model, but it requires the addition of extra operators. This can result in a suboptimal model, which sometimes defeats the purpose of conversion.

Acknowledgements

I would like to thank Ing. Róbert Kalmár of NXP for his professional experience which was the source for section 2.

References

[1] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.