SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Optimizacija procesa učenja dubokih modela

Dominik Stipić
Voditelj: *prof. dr. sc. Siniša Šegvić*

Zagreb, svibanj 2021.

SADRŽAJ

1. Uvod

Duboko učenje je trenutno najzastupljeniji pristup rješavanja problema na području umjetne inteligencije. Trenutni, *state of the art*, modeli posjeduju nekoliko milijuna parametara što otežava njihovo učenje i rad u praksi. Zbog toga su razvijene brojne optimizacijske metode s ciljem ubrzanja vremena učenja i predviđanja.

Cilj ovog seminara je dati kratki pregled metoda za skraćivanje vremena učenja koje ne smanjuju performanse modela na neviđenim primjerima. Metode za skraćivanje vremena učenja koje su obrađene u ovom seminaru, možemo ugrubo podijeliti na metode koje nastoje **ubrzati protočnost cjevovoda** za dostavljanje podataka (ulazni cjevovod) i na one koje **ubrzavaju izvođenje aritmetičkih operacija na akceleratoru**.

U eksperimentalnom dijelu dan je izvještaj o učinkovitosti korištenih metoda. Većina je eksperimenata napravljena u TensorFlowu, ali je također dio eksperimenta proveden i u PyTorchu radi mogućnosti usporedbe i radi dobivanja što boljih zaključaka. Tamo gdje radimo usporedbe, eksperimenti u različitim programskim okvirima provedeni su s istim modelima i sa istim podacima.

Pri eksperimentiranju s metodama koje nastoje ubrzati ulazni cjevovod koristimo ugrađene funkcionalnosti radnih okvira TensorFlow i PyTorch, specifično radi se o metodama modula `tensorflow.data` i o atributima razreda `torch.utils.data.DataLoader`. Kod PyTorcha testirali smo kako drugačija organizacija podataka na disku utječe na brzinu učitavanja. Inspiracija za drugačiju organizaciju podataka na disku preuzeta je iz članka Aizman et al. (2020).

Kod eksperimenata za ubrzanje postupka računanja na akceleratoru isprobano je učenje s miješanom preciznošću i učenje s grupiranim CUDA jezgrama uz pomoć XLA prevodioca.

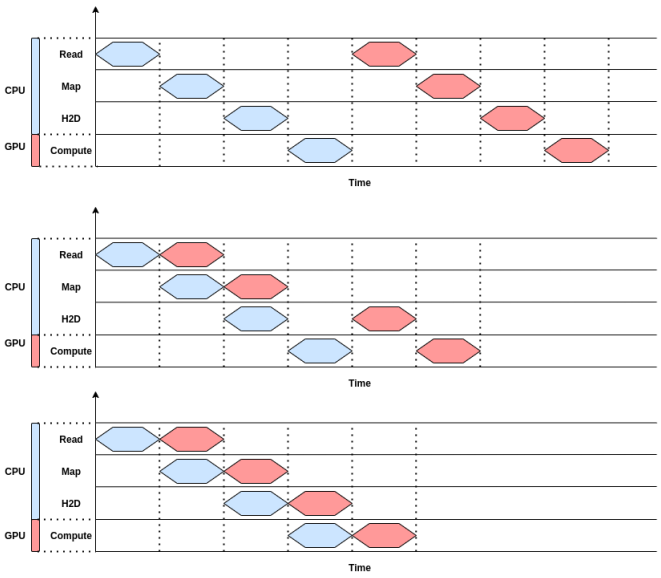
2. Optimizacijske Metode

2.1. Učitavanje i obrada podataka

Bitan dio procesa učenja je dostaviti podatke akceleratoru i učitati ih u radnu memoriju na što učinkovitiji način. Ako radimo na problemima koji koriste jako velike podatke za učenje, najčešće će nam se desiti da faza učitavanja i procesuiranja podataka postane usko grlo procesa, što će razultirati s nezaposlenim akceleratorom. Kompoziciju niza transformacija podatka i proces učitavanja podataka iz diska nazivamo ulaznim cjevovodom našeg procesa učenja. U načelu, optimizacijske metode s kojima eksperimentiramo iskorištavaju ugrađeni paralelizam u postupku učitavanja podataka i efikasnije koriste memoriju. Na slici 2.1 prikazane su 3 razine optimiziranosti cjevovoda na temelju broja faza koje se mogu izvoditi paralelno. Slijede opisi metoda s kojima eksperimentiramo.

Vektorizacija transformacija Efikasnije je izvoditi transformacije nad grupom podataka nego nad svakim podatkom pojedinačno. U Tensorflowovu modulu `tf.data` ovu metodu izvodimo odabirom pametnog redoslijeda faza transformacija i faze stvaranja mini-grupa, npr. ulazni cjevovod `dataset.map(transformation).batch(batch_size)` pretvaramo u `dataset.batch(batch_size).map(transformation)`.

Pretpribavljanje (engl. prefetch) Omogućava nam paralelno izvršavanje nekoliko faza u cjevovodu. U osnovnoj inačici cjevovoda ne postoje preklapanja između faza, nego se sve dešavaju slijedno u jednoj dretvi. Pretpribavljanje je implementiranano na način da pozadinske dretve pune interni međuspremnik (*engl. buffer*) s podacima, tako da ti podaci dočekuju spremni glavnu dretvu, koja istovremeno može raditi augmentaciju podataka ili neke druge zadatke ulaznog cjevovoda. U Tensorflowu ovu funkcionalnost implementiramo pomoću poziva `prefetch` metode s argumentom `buffer_size` koji određuje duljinu internog međuspremnika. Pythorchov razred `DataLoader` sadrži atribut `num_workers` kojim određujemo broj podprocesa



Slika 2.1: Različite razine optimiziranosti cjevovoda. Prva slika prikazuje neoptimiziranu verziju cjevovoda gdje ne postoji preklapanja među fazama. Druga slika prikazuje cjevovod u kojem se paralelno učitavaju podaci, pa zbog toga dolazi do preklapanja faza na procesoru. Posljednja slika prikazuje optimalnu verziju cjevovoda u kojoj ne postoje čekanja i u kojoj je akcelerator uvijek zaposlen.

koji paralelno učitavaju podatke iz diska. Pretpribavljanje nam omogućuje da prijeđemo iz prve razine na drugu razinu optimiziranosti cjevovoda sa slike 2.1.

Međuspremnik (engl. cache) U većini zadataka dubokog učenja postoje dvije vrste transformacija u ulaznom cjevovodu, one s determinističkim izlazima i one s nedeterminističkim izlazima. Korištenjem ove metode determinstički transformirane podatke možemo spremiti u međuspremnik tako da takve transformacije računamo samo jednom. Nakon što prvi put učitamo i obradimo cijeli skup podataka iz diska, svaka sljedeća epoha iskorištava već obrađene podatke i jedan dio cjevovoda se više ne koristi. U Tensorflowu to izgleda ovako:

`pipeline.cache().map(augmentation))`,

transformacije unutar varijable `pipeline` izvode se samo u prvoj epohi, te se zatim spremaju u radnu memoriju. Sljedeće faze cjevovoda, poput augmentacije podataka, iskorištavaju podatke iz međuspremnika. Podatke nastale augmentacijom ne spremamo u memoriju jer želimo zadržati svojstvo nasumičnosti.

Paralelno pretprocesiranje slika (engl. parallelized map) Omogućuje nam da iskoristimo potencijalni paralelizam unutar faza transformacija u cjevovodu. Ova funkcionalnost se u Tensorflowu omogućuje na način da postavimo argument `num_parallel_calls`

metode `map`

Zamrzivanje memorije (engl. pinned memory) Kada korisnik naredi Pythonu da prebaci podatke iz diska u radnu memoriju, ti podaci se u stvarnosti neće u potpunosti prebaciti iz diska u radnu memoriju zbog procesa virtualizacije memorije. Prebaciti će se tek nekoliko stranica npr. slike, a ne svi bajtovi jedne slike. Kod prijenosa podataka iz radne memorije na akcelerator svi bajtovi jedne slike moraju se nalaziti u radnoj memoriji. U Pytorchu možemo narediti operacijskom sustavu da prebaci sve stranice jedne slike u RAM, tako da namjestimo atribut `pin_memory` razreda `torch.utils.data.DataLoader`. Time operacijski sustav u trenutku prijenosa na akcelerator neće trebati čekati da se učitaju sve stranice slika iz diska nego će sve stranice slika već nalaziti u RAM-u.

Grupna organizacija podataka na disku Ako radimo na zadacima poput semantičke segmentacije na raspolaganju su nam slike i maske tih slika. Ti podaci su najčešće organizirani na način da su slike i maske smještene u zasebne direktorije. Takva organizacija podataka nije dobra jer u strojnom učenju uvijek čitamo podatke i njihove oznake zajedno, a oni u ovom slučaju nisu spremljeni slijedno na disku, nego su razbacani po različitim blokovima diska. Da bismo podatke spremili slijedno na disk moramo koristiti serijalizacijske protokole koji će npr. zapis $(slika, maska, metapodaci)$ binarizirati i formatirati prema pravilima protokola. Prednosti koje imamo s ovakvom organizacijom podataka je ta da koristimo manje prostora na disku zbog smanjenja efekta fragmentacije i brže čitamo podatke jer glava tvrdog diska ne treba posjećivati nasumične blokove nego tada može slijedno čitati blokove diska.

Tensorflowova implementacija TFRecord koristi Googlov *Protocol buffer* serijalizacijski protol za stvaranje tfrecord datoteke koja podatke organizira u zapise i sprema ih slijedno na disk. Prema Aizman et al. (2020), Stvaranje grupe zapisa koji su spremljeni slijedno može se implementirati uz pomoć GNU tar naradbe za arhiviranje podataka.

2.2. Računanje na akceleratoru

Unaprijedni i unazadni prolaz podataka kroz model vremenski i memorijski su najzahtjevnije faze u procesu učenja, zato su nam metode za ubrzanje tih faza posebno interesantne.

Učenje s miješanom preciznošću (engl. mixed precision) je popularna i jednostavna metoda za ubrzanje procesa računanja prvi put spomenuta u Micikevicius et al.

(2017). Pokazuje se da se većina operacija u modelima dubokog učenja može izvoditi podjednako točno s FP16 tenzorima, kao i s uobičajenom FP32 reprezentacijom. Teorijski, prebacivanje svih operacija u modelu daje nam ubrzanje od 2x jer imamo duplo manji zapis brojeva. Učenje s miješanom preciznosti koristi 2 tehnike kako bi se spriječili neki problemi koje proizlaze korištenjem ove metode, a to su skaliranje gubitka i spremanje težina u FP32 zapisu.

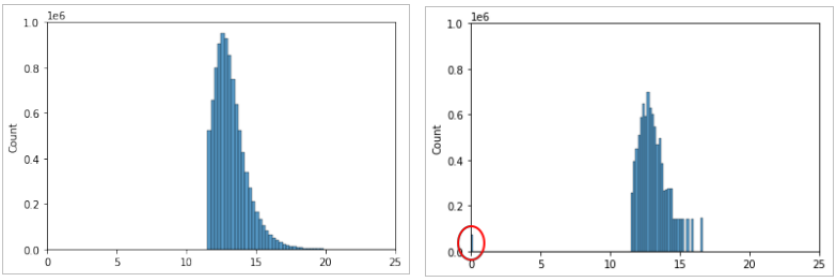
Prvi problem se događa ako naivno pretvorimo sve tenzore u FP16 zapis, desiti će nam se nešto slično kao i na slici 2.2. Vrijednosti manje od 2^{-24} postaju 0 u FP16 zapisu, što rezultira time da dobijemo jako velik broj nula. Ovaj problem rješavamo uporabom tehnike skaliranja gubitka. Cilj ove tehnike je da pomaknemo gradijente, $\frac{\partial L}{\partial W}$, u područje koje je pogodnije za prikazivanje u FP16 zapisu. To radimo na način da nakon što izračunamo gubitak s FP16 aktivacijama, taj gubitak skaliramo sa skalom S i nad tim skaliranim gubitkom radimo unazadni prolaz. Unazadnim prolazom dobivamo skalirane gradijente u FP16 zapisu, koje zatim pretvaramo u punu preciznost, množimo sa $\frac{1}{S}$ i s tim gradijentima radimo ažuiranje glavnih težina.

Pokazuje se da nije dobro prebaciti težine modela u FP16 zapis nego se ipak moramo zadovoljiti time da operacije radimo s FP16, a težine održavamo u FP32. Gradijente dobivamo u FP16 preciznosti i nakon što bi pomnožili te gradijente sa stopom učenja α, svi pomaci, ΔW, postali bi 0 i učenja ne bi bilo. Zato moramo prije množenja sa stopom učenja, pretvoriti gradijente u FP32 zapis i s takvim formatom raditi učenje. To implicira da težine modela moramo održavati u punoj preciznosti.

Ova metoda je posebno efikasna na Volta i Turing arhitekturama grafičkih kartica, jer te generacije arhitektura posjeduju posebne instrukcije i resurse (tenzorske izvršne jedinice) za izvođenje aritmetičkih operacija $D = AB + C$, gdje su A i B matrice s FP16 preciznošću, a C i D matrice s FP32 preciznošću. Zbog toga se većina operacija u modelima koji uče s miješanom preciznosti može izvršiti pomoću jedne instrukcije. Prema NVIDIA (2020), empirijski se pokazalo da učenje s miješanom preciznošću može ubrzati proces učenja do 3x za aritmetički zahtjevnije modele bez korištenja tenzorskih izvršnih jedinica.

XLA (accelerated linear algebra) je optimizacijski prevodioc (*engl. compiler*) koji za graf izvođenja daje optimizirani binarni kod tog grafa. Graf izvođenja u dubokom učenju je struktura podataka koja čvorove modelira kao operacije, a ulazi i izlazi tih operacija su tenzori. Postoje dvije faze u životnom ciklusu grafova, a to je proces građenja grafa ili prevođenja i proces izvršavanja grafa. Ovakav životni ciklus je posebno pogodan za optimizaciju jer je moguće u procesu prevođenja raditi razne transformacije grafa prije njegova izvođenja. Upravo je za tu svrhu dizajniran XLA

prevodioc. Grupirane operacije smanjuju korištenje glavne memorije od akceleratora jer takve operacije ne trebaju komunicirati preko te memorije.



Slika 2.2: Na obje slike je prikazan histogram apsolutnih vrijednosti logoritma gradijenta ili $|log(grad)|$. Na lijevoj slici se nalazi histogram u FP32 zapisu, a na desnoj slici histogram u FP16 zapisu. Crvenim ovalom na desnoj slici zaokružen je stupac koji odgovara težinama koje su postale 0 u FP16 zapisu. Broj gradijenta koji su postali 0 neprihvatljivo je velik za praktične svrhe.

3. Eksperimenti

Cilj sljedećih eksperimenta je značajno smanjiti vrijeme učenje uz pomoć opisanih metoda. Poglavlje je podijeljeno na dijelove u kojima eksperimente radimo u Tensorflowu i u PyTorchu. Verzija Tensorflowa bila je 2.4, a PyTorcha 1.8. Za svaki radni okvir testirane su metode za optimizaciju vremena učitavanja i ubrzanje vremena računanja na akceleratoru.

Kod eksperimenata u kojima se nastoji ubrzati rad akceleratora korištena je NVIDIA Tesla T4 grafička kartica čije specifikacije se nalaze u tablici 3.1. NVIDIA za svaku grafičku karticu specificira računsku sposobnost (*engl. compute capability*). Taj podatak korisno je znati jer svaka nova generacija grafičkih kartica podržava drugačiju instrukcijsku arhitekturu što znači da novije verzije grafičkih kartica imaju veće mogućnosti nego prijašnje generacije. Računska sposobnost NVIDIA Tesla T4 kartice je 7.4, što znači da sadrži tenzorske izvršne jedinice (*engl. tensor cores*) koje imaju sve kartice iznad verzije 7.0. Grafičke kartice koje sadrže tenzorske izvršne jedinice su nam posebno interesantne jer još više ubrzavaju proces učenja s miješanom preciznošću.

GPU arhitektura	Turing
Tenzor jezgre	320
Cuda jezgre	2560
fp32 propusnost.	8.1 TFLOPS
Mješana preciznost (fp16/fp32) propusnost.	65 TFLOPS
Gpu memorija	16 GB
Propusnost memorije	300 GB/s

Tablica 3.1: Performanse grafičke kartice NVIDIA Tesla T4. Posebno je zanimljivi podatak o broju tenzorskih izvršnih jedinica i propusnosti kod operacija s miješanom preciznošću. Grafičke kartice koje imaju tenzorske izvršne jedinice dobri su kandidati za učenje s miješanom preciznošću.

3.1. Eksperimenti u Tensorflowu

Za mjerenje vremena korišten je Tensorflowov *profiler* koji je integriran s Tensorboardom i koji uz vremenske podatke daje dodatne vizualno prikazane statistike o korištenju računalnih resursa.

U dijelu **Učitavanje i obrada podataka**, radi se vremenska analiza i optimizacija ulaznog cjevovoda za jednostavni klasifikatorski model. Model se u eksperimentima uči na slikama rezolucije 500 x 500. Isprobavamo i opisujemo 5 različitih metoda za ubrzanje cjevovoda, a to su: vektorizacija transformacija, pretpribavljanje, korištenje međuspremnika i paralelelno procesuiranje slika.

U dijelu **Ubrzanje računanja na akceleratoru**, radimo s metodama koje nastoje smanjiti vrijeme računanja na akceleratoru. U tom dijelu opisani su rezultati kad učimo s miješanom preciznošću (*engl. mixed precision*) i kad učimo s grupiranim operacijama (XLA).

3.1.1. Učitavanje i obrada podataka

U ovom dijelu, eksperimenti su izrađeni s različitim vrstama cjevovoda. Za implementaciju ulaznog cjevovoda korišteni su razredi modula `tensorflow.data`. Kao model koristi se jednostavna verzija klasifikatora sa slike 3.1 s glavnom motivacijom da se što manje operacija izvodi na GPU-u, a da se većina vremena provede u obradi podataka u ulaznom cjevovodu.

Slika 3.1: Jednostavni klasifikator za testiranje ulaznog cjevovoda. Tamno plavi kvadrat predstavlja grupu Conv -> ReLu -> MaxPool, svijetlo plavi kvadrat odgovara skupini Dense -> ReLu. Na izlazu modela dobivamo distribuciju po 3 razreda.

Za početak je odabrana osnovna, neoptimizirana inačica cjevovoda koja se nastoji ubrzati pomoću 4 različite metode za ubrzanje protočnosti podataka kroz cjevovod. Transformacije koje radimo na slikama rezolucije 500 x 500 su:

– Normalizacija

– Stvaranje mini-grupa podataka (*engl. Batch*)

– Slučajni isječci rezolucije 224 x 224

– Slučajna rotacija slika s kutom između [-30, 30].

– Promjena svjetline slike.

Programski kod osnovne inačice cjevovoda iz koje se gradi optimiziranija verzija cjevovoda može se vidjeti na slici 3.2.

```
# baseline
ds_base1 = tfds.load('beans', split='train', shuffle_files=True). \
                take(300). \
                map(standardize). \
                map(lambda x, y: (tf.image.per_image_standardization(x), y)). \
                map(random_crop). \
                batch(batch_size, drop_remainder=True). \
                map(tf_random_rotate_image). \
                map(brightness)
```

Slika 3.2: Osnovna inačica ulaznog cjevovoda. Na početku cjevovoda učitavamo slike čija je rezolucija 500 x 500, uzimamo 300 slika, pretvaramo slike u tenzore, normaliziramo slike, uzimamo nasumični isječak rezolucije 224 x 224, stvaramo mini-grupu od 12 primjera, radimo nasumičnu rotaciju za kut između [-30, 30] stupnjeva i mijenjamo svjetlinu slike.

Izgradnja poboljšane verzije cjevovoda napravljena je na način da su se funkcionalnosti dodavale postepeno. Model je treniran s osnovnom verzijom cjevovoda 5 epoha i u svakoj epohi analizirani su koraci od 15 do 18. Kao procjena efikasnosti cjevovoda korišteno je prosječno vrijeme koraka. Prosječna vremena su znala dosta varirati pa je za svaki cjevovod napravljeno 10 takvih procedura, što daje 10 prosječnih vremena. Za prosjek prosječnih vremena napravljen je 95 postotni obostrani interval povjerenja, pomoću formule:

$$CI = \hat{X} \pm \frac{t_\alpha S}{\sqrt{n}}$$

Rezultati ove analize nalaze su u tablici 3.2.

Inačica	Vectorized Random Crop	Prefetch	Cache	Parallel Normalization map	Parallel Random Crop map	Parallel Random Rotation map	Parallel Brightness map	95% interval povjerenja za prosječno vrijeme koraka učenja (ms)
1	NE	NE	NE	NE	NE	NE	NE	1095.8 +/- 129.82
2	DA	NE	NE	NE	NE	NE	NE	1102.54 +/- 125.85
3	DA	DA	NE	NE	NE	NE	NE	1119.38 +/- 124.29
4	DA	DA	DA	NE	NE	NE	NE	1111.28 +/- 132.38
5	DA	DA	DA	DA	NE	NE	NE	1103.84 +/- 135.92
6	DA	DA	DA	NE	DA	NE	NE	1100.88 +/- 131.80
7	DA	DA	DA	NE	NE	DA	NE	502.28 +/- 427.09
8	DA	DA	DA	NE	NE	NE	DA	1086.52 +/- 419.66
9	DA	DA	DA	DA	DA	NE	NE	1102.98 +/- 142.67
10	DA	DA	DA	DA	NE	DA	NE	555.98 +/- 380.39
11	DA	DA	DA	DA	NE	NE	DA	1083.48 +/- 366.74
12	DA	DA	DA	NE	DA	DA	NE	642.14 +/- 346.32
13	DA	DA	DA	NE	DA	NE	DA	1093.98 +/- 350.92
14	DA	DA	DA	NE	NE	DA	DA	476.44 +/- 422.87
15	DA	DA	DA	DA	DA	DA	NE	643.5 +/- 182.75
16	DA	DA	DA	DA	DA	NE	DA	1098.8 +/- 333.61
17	DA	DA	DA	DA	NE	DA	DA	594.96 +/- 375.69
18	DA	DA	DA	NE	DA	DA	DA	500.18 +/- 228.04
19	DA	DA	DA	DA	DA	DA	DA	548.72 +/- 169.40

Tablica 3.2: Analiza efikasnosti različitih inačica cjevovoda. Za svaku vrstu cjevovoda prikupljeno je 10 uzoraka prosječnog vremena koraka učenja iz kojih je izračunat 95% interval povjerenja.

Iz dane analiza može se primijetiti da je korak u kojem se izvodi rotacija grupe slika usko grlo, tj. faza cjevovoda koja značajno smanjuje protočnost. Tensorflow posjeduje dvije opcije kako može izvoditi operacije, a to su dinamičko *(engl. eager)* izvršavanje i izvršavanje na grafu. Kod izvršavanja na grafu korisnik izgrađuje graf tako da povezuje operacije i daje im podatke, a tijekom sesije, operacije u grafu se izvode. Takav način izvođenja još se naziva i lijenim izvođenjem, jer se operacije ne izvode u trenutku stvaranja, već kasnije. Prije izvršavanja grafa, Tensorflowov optimizacijski sustav Grappler razmješta čvorove i pojednostavljuje reprezentaciju grafa s ciljem poboljšanja performansi. Kod dinamičkog izvršavanja, Tensorflow izvodi operacije kao i kod normalnog Python programa, operacije se izvode jedna po jedna te ne postoji prilika da se operacije grupiraju i zbog toga efikasnije izvedu.

U svim verzijama cjevovoda implementacija rotacije se izvodi dinamički, a ne lijeno što predstavlja problem s gledišta brzine učenja. Funkcija `tf.py_function(func)` omogućava nam da umotanu funkciju `func` izvršavamo kao običnu Python funkciju, ali koja se nalazi unutar Tensorflowovog grafa. Isječak koda koji implementira spomenutu funkcionalnost može se vidjeti na 3.1.

```python
import numpy as np
import tensorflow as tf
import scipy.ndimage as ndimage

def py_random_rotate_image(images):
  images = [ndimage.rotate(image, np.random.uniform(-30, 30),
   reshape=False) for image in images]
  images = tf.stack(images)
  return images

def tf_random_rotate_image(images, labels):
  [images] = tf.py_function(py_random_rotate_image,
                            [images],
                            [tf.float32])
  return images, labels
```

Listing 3.1: Implementacija rotacije grupe slika. Za razliku od Tensorflowovih funkcija za obradu podataka, rotacija se ne izvodi na grafu nego dinamički. Funkcija tf.py function omogućava dinamičko izvršavanje (*engl. eager execution*) proizvoljnih Python funkcija unutar TensorFlowovog grafa.

Za novu implementaciju rotacije korištena je Tensorflowova funkcija `tensorflow.keras.layers.experimental.preprocessing.RandomRotation(factor=[-0.3, 0.3])` koja se izvršava unutar grafa. Prikaz usporedbe vremena koraka za prijašnju implementaciju i za implementaciju nakon promjene možemo vidjeti u tablici 3.3.

Vrsta izvšavanja rotacije	**95% Interval povjerenja**
Dinamičko izvršavanje	548.72 ± 169.40
Izvršavanje na grafu	$39{,}56 \pm 7{,}53$

Tablica 3.3: Usporedba intervala povjerenja za vrijeme prosječnog koraka za dinamičko izvršavanje rotacije i izvršavanje rotacije na grafu. Podatak o vremenu za dinamičko izvršavanje preuzet je iz tablice 3.2 za 19-tu inačicu cjevovoda.

3.1.2. Ubrzanje računanja na akceleratoru

Poslije optimizacije cjevovoda napravljena je analiza tehnika koje ubrzavaju računanje na grafičkim karticama. Posebno nam je zanimljiv podatak o vremenu izvođenja operacija na akceleratoru (*engl. device time*).

U ovom djelu učenje je napravljeno s kompleksnijom verzijom modela, zbog što realističnije simulacije. Problem s početnom vrstom modela je to što se velika većina operacija izvodila na procesoru, a ne na GPU-u. U početku se otprilike 90 % operacija izvodilo na procesoru što u praksi baš i nije čest slučaj. Za novu vrstu modela izabran je ResNet-50, čime je udio operacija na GPU-u porastao na 77.8 %. Rezultati eksperimenata mogu se naći u tablici 3.4.

Vrsta metode	**GPU vrijeme**
Početna verzija	130.05 ± 1.24
XLA	102.92 ± 0.71
Mješana preciznost	66.28 ± 0.63
XLA + Mještana preciznosti	37.85 ± 0.38

Tablica 3.4: Tablica prikazuje 95%-tne intervale povjerenja za GPU vrijeme. GPU vrijeme je prosječno vrijeme izvođenja CUDA jezgri na GPU-u.

3.2. Eksperimenti u PyTorchu

U ovoj sekciji nalaze se rezultati eksperimenta u PyTorchu. U prvoj verziji eksperimenata testiran je utjecaj organizacije podataka na brzinu učitavanja slika za zadatak semantičke segmentacije. U drugoj verziji eksperimenata implementirano je učenje s miješanom preciznošću. Postava eksperimenta je jednaka postavi kod eksperimenta u Tensorflowu tako da se ti eksperimenti mogu usporediti.

```
def start_timer():
    global start_time
    start_timer = 0
    gc.collect()
    torch.cuda.empty_cache()
    torch.cuda.synchronize()
    start_time = time.time()

def end_timer():
    torch.cuda.synchronize()
    end_time = time.time()
```

```
12      return end_time - start_time
```

Listing 3.2: Kod za mjerenje vremena u PyTorchu. Funkcija Start timer osim što započinje mjerenje vremena poziva čišćenje nereferencirane memorije (*engl. garbage collector*), čisti alociranu memoriju na GPU-u (torch.cuda.empty cache) i čeka da sve pokrenute CUDA jezgre završe (funkcija torch.cuda.synchronoze). Druga fukcija se blokira sve dok ne završe sve CUDA jezgre pa tek bilježi završno vrijeme.

3.2.1. Učitavanje i obrada podataka

Eksperimenti su napravljeni sa satelitskim slikama za zadatak semantičke segmentacije, rezolucije 5000 x 5000 piksela od kojih svaka ima 75 MB. Eksperimentirano je s dvije različite inačice organizacije podataka na tvrdom disku, tj. s organizacijom u kojoj su podaci **organizirani u grupe** i s **raspršenom organizacijom** podataka. Kod raspršene organizacije podataka slike i maske tih slika smještene su u zasebne direktorije, a kod grupne organizacije podatka grupa zapisa, gdje se zapis definira kao par (slika, maska), arhivirani su pomoću linuxove tar naredbe. U obje inačice dajemo izvještaj o prosječnom vremenu učitavanja podataka.

Za raspršenu organizaciju podataka testiramo kako broj potprocesa *(engl. workers)*, veličina mini-grupe *(engl. batch size)* i zamrzivanje memorije *(engl. pined memory)* utječe na prosječno vrijeme učitavanja slike i maske. Za grupnu organizaciju također testiramo utjecaj veličine grupe na prosječno vrijeme učitavanje podataka.

U svakom eksperimentu dobivena je jedna vremenska serija, duljine m, u kojoj je svaki uzorak vrijeme trajanja učitavanja podataka s diska. Radi otklanjanja efekta nasumičnosti i dobivanja što stvarnijih vremena, stvoreno je N=20 takvih vremenskih serija za svaki eksperiment. Na kolekciju od N vremenskih serija primjenjujemo usrednjavanje i računamo standardnu devijaciju po uzorcima, čime dobivamo dvije serije duljine m. Iz te dvije agregirane vremenske serije računamo srednju vrijednost i standardnu devijaciju koje koristimo za računanje 95 postotnih intervala povjerenja, pomoću koeficijenta $t^m_{\alpha=0.05}$=2.093. Intervali povjerenja za eksperimente kod raspršene organizacije prikazani su u tablici 3.5, a kod grupne u tablici 3.6. Sva vremena iskazana su u sekundama.

Iz tablice 3.5 možemo vidjeti da u suprotnosti s očekivanjem, rast broja potprocesa ne mora nužno smanjivati prosječno vrijeme učitavanja podataka. Povećanjem broja potprocesa dolazi do povećanja korištenja memorije, te bi se stoga taj parametar trebao namjestiti u ovisnosti o zadatku na kojem radimo. Ista stvar vrijedi za zamrzivanje memorije. Povećanjem broja mini-grupe možemo očekivati linearni porast vremena

Eksperiment	Mini-grupa	Potprocesi	Zamrznuta memorija	95% CI (sec)
1	1	0	NE	1.283 +/- 0.009
2	1	1	NE	1.543 +/- 0.013
3	1	2	NE	1.248 +/- 0.04
4	1	0	DA	1.696 +/- 0.036
5	1	1	DA	1.603 +/- 0.017
6	1	2	DA	1.636 +/- 0.372
7	1	0	NE	1.283 +/- 0.009
8	2	0	NE	3.152 +/- 0.01
9	3	0	NE	5.29 +/- 0.037

Tablica 3.5: Tablica prikazuje 95% intervale povjerenja za **raspršenu organizaciju podataka** na disku. Radi lakšeg snalaženja, različitim bojama označene su ćelije stupaca varijabli eksperimenta koje mijenjamo i testiramo. Plavom bojom označeni su eksperimenti u kojima promatramo utjecaj broja potprocesa koji paralelno učitavaju podatke, crvenom bojom oni u kojima gledamo utjecaj zamrznute memorije, a žutom promatramo utjecaj veličine mini-grupe.

trajanja.

Iz tablice 3.6 možemo primijetiti da povećanjem zapisa u grupi dobivamo ubrzanje od pedesetak milisekundi. Ako je moguće veličinu mini-grupe i veličinu zapisa bilo bi dobro izjednačiti. Kada je veličina mini grupe veća od broja zapisa značajno se povećava vrijeme učitavanja podataka jer moramo učitati više arhiva za stvaranje jedne mini-grupe.

Pod pretpostavkom da je organizacija podataka takva da je broj zapisa u arhivi jednak mini-grupi s kojom učimo i ako zanemarimo statističke greške, možemo usporediti prosječna vremena učitavanja iz obje tablice. Za mini-grupu od 2 primjera dobivamo ubrzanje od 430 ms, a za mini grupu od 3 primjera dobivamo ubrzanje od 602 ms. Ta ubrzanja možda djeluju mala, ali zbog učestalosti učitavanja podataka učinak ubrzanja je dosta velik, npr. ako učimo model 100 epoha i ako svaka epoha ima 100 podataka, učenje će nam se smanjiti za 1 sat i 42 minute ako koristimo raspršenu organizaciju podataka.

Eksperiment	Mini-grupa	Zapisi u grupi (arhivi)	95% CI (sec)
1	1	2	1.392 +/- 0.019
2	1	3	1.409 +/- 0.014
3	2	3	3.031 +/- 0.123
4	3	3	4.09 +/- 0.022
5	2	2	2.72 +/- 0.069
6	3	2	4.688 +/- 0.265

Tablica 3.6: Tablica prikazuje 95% intervale povjerenja za **grupnu organizaciju podataka** na disku. Učitavanje se radi bez potprocesa i bez zamrznute memorije. Plavom bojom označeni su ekperimenti u kojima promatramo utjecaj broja zapia, crvenom bojom oni u kojima mijenjamo veličinu mini grupe uz 3 zapisa u tar arhivi, a žutom oni u kojima mijenjamo veličinu mini-grupe uz 2 zapisa.

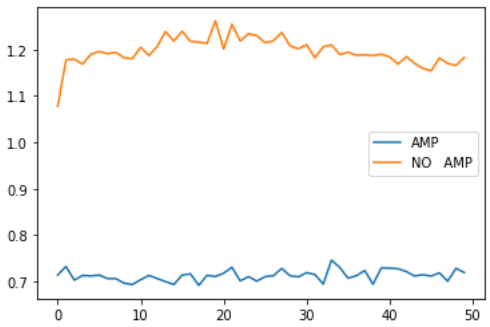
3.2.2. Ubrzanje računanja na akceleratoru

Eksperimentu su napravljeni s Resnet-50 arhitekturom na slikama dimenzija 500 x 500. Za implementaciju učenja s miješanom preciznosti korišteni su `torch.cuda.amp.GradScaler` i `torch.cuda.amp.autocast` moduli. Modul GradScaler zadužen je za dinamičko skaliranje gubitka sa skalarom S, tako da gradijenti budu pomaknuti u raspon koji je pogodniji za prikaz s FP16 preciznosti. Prije ažuiranja parametara, gradijenti se ponovno vraćaju u FP32 reprezentaciju i množe se s inverzom 1/S.

Argumenti aritmetičkih operacija unutar `torch.cuda.amp.autocast` konteksta automatski se pretvaraju u preciznost koja je najpogodnija za takvu vrstu operacija. Zbog toga Pytorchova dokumentacija zajedničko korištenje modula `torch.cuda.amp.GradScaler` i `torch.cuda.amp.autocast` naziva učenjem s **automatskom miješanom preciznosti**.

```
model = ResNet50().cuda()
optimizer = optim.Adam(model.parameters())
scaler = torch.cuda.amp.GradScaler()
for input, target in data:
    input, target = input.cuda(), target.cuda()
    with torch.cuda.amp.autocast():
        output = model(input)
        loss = loss_fn(output, target)
    scaler.scale(loss).backward()
    scaler.step(optimizer)
    scaler.update()
    optimizer.zero_grad()
```

Listing 3.3: Demonstracija učenja s automatskom miješanom preciznosti pomoću PyTorchovih torch.cuda.amp.GradScaler i torch.cuda.amp.autocast modula



Slika 3.3: Slika prikazuje vremensku seriju trajanja jednog koraka učenja. Vremenska serija sadrži 50 uzoraka. Narančastom linijom prikazana je vremenska serija za učenje bez miješane preciznosti, a s plavom vremenska serija koja se dobiva učenjem s miješanom preciznošću.

	AMP	**NO AMP**
Srednja vrijednost vremena	0.713 s	1.196 s
Standardna devijacija vremena	0.012 s	0.029 s

Tablica 3.7: Usporedba rezultata za procese učenja sa i bez miješane preciznosti. AMP je kratica za *automatic mixed precision*

4. Zaključak

Vremenskom analizom trajanja koraka učenja za svaku optimizacijsku metodu i ablacijskim testovima možemo dobiti procjenu koliko pojedina metoda skraćuje vrijeme učenja.

Kod eksperimenta u TensorFlowu pokazali smo kako faze procesuiranja koje se ne izvode na Tensorflowovu grafu, vrlo lako mogu postati usko grlo cjevovoda. Uklanjanjem uskih grla cjevovoda dobili smo smanjenje koraka učenja za 15x u odnosu na optimiziranu verziju cjevovoda koja sadrži usko grlo. Iskorištavanjem paralelizma u cjevovodu i smanjivanjem zastoja u toku operacija povećali smo performansu cjevovoda za otprilike 2x. Što se tiče ubrzanja operacija na akceleratoru, pokazalo se da učenje s grupiranim jezgrama (XLA) može smanjiti vrijeme računanja na GPU-u za otprilike 21% u odnosu na početnu verziju. Učenjem s miješanom preciznošću dobili smo ubrzanje od 2x u odnosu na početnu verziju. Kombinacija obje metode daje dodatno ubrzanje od 4x u odnosu na početnu verziju.

Kod eksperimenta u PyTorchu korištenjem učenja s miješanom preciznošću smanjili smo vrijeme učenja za otprilike 40%. Organizacijom podataka u zapise koji se spremaju u tar arhive i izjednačavanjem veličine mini-grupe s veličinom zapisa u arhivi smanjili smo vrijeme učitavanja podataka u jednom slučaju za 14%, a u drugom 23%. Prikaz spomenutih poboljšanja nalazi se u 4.1.

Eksperimenti	**Ubrzanje**	**Smanjenjeno vrijeme izvođenja**
Optimizacija cjevovoda	2x	50%
Grupna organizacija podataka	1.2x	18%
Učenje s mješanom preciznošću	2x	50%
Učenje s grupiranim jezgrama	1.3x	21%

Tablica 4.1: Sažetak svih poboljšanja koji su zabilježeni u eksperimentima. Poboljšanja performansi iskazana su kvalitativnim putem.

5. Literatura

Alex Aizman, Gavin Maltby, i Thomas Breuel. High performance I/O for large scale deep learning. *CoRR*, abs/2001.01858, 2020. URL http://arxiv.org/abs/2001.01858.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory F. Diamos, Erich Elsen, David García, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, i Hao Wu. Mixed precision training. *CoRR*, abs/1710.03740, 2017. URL http://arxiv.org/abs/1710.03740.

NVIDIA. Nvidia deep learning performance, 2020. URL https://docs.nvidia.com/deeplearning/performance/index.html.

6. Sažetak

Efikasno korištenje i implementacija algoritma dubokog učenja nosi svoj infrastrukturni teret, koji iz godine u godinu sve više raste. Zbog toga su u ovom seminaru opisane optimizacijske metode koje nastoje skratiti vrijeme učenja, a koje pri tome ne oštećuju performanse modela na neviđenim podacima.

Ovaj seminar predlaže optimizacijske metoda koje značajno smanjuju vrijeme učenja, a koje je jednostavno ugraditi u postojeće projekte. Za svaku predloženu metodu napravljena je statistička analiza vremena trajanja simuliranih faza i rezultati tih eksperimenta su zabilježeni u zasebne tablice.

Optimizacijske metode koje razmatramo su metode za ubrzanje protočnosti ulaznog cjevovoda i one koje ubrzavaju računanje aritmetičkih operacija na akceleratoru. Metode za ubrzavanje protočnosti ulaznog cjevovoda koriste implicitni paralelizam koji je sadržan u fazama obrade podataka, efikasnije koriste memoriju i efikasnije skladište podatke na disku.Metode za ubrzanje računanje aritmetičkih operacija na akceleratoru s kojima radimo su učenje s miješanom preciznošću i učenje s grupiranim operacijama pomoću XLA prevodioca.