Program za klasifikaciju znamenki iz skupa MNIST
Tehnička dokumentacija
Verzija 1.3

Studentski tim: Jelena Bratulić
Lea Ming Li
Hrvoje Maligec
Klara Marijan
Pavo Matanović
Iva Panić
Leo Pleše
Karlo Puh
Frano Rajič
Tin Reiter

Nastavnik: Siniša Šegvić

Sadržaj

1. Uvod

U današnjem svijetu gdje se većina podataka obrađuje digitalnim računalom, a ne umom, neuronske mreže igraju veliku ulogu. Općenita ideja o neuronskim mrežama potiče još iz 1940. godine kada je objavljen matematički model neuronske mreže u okviru teorije automata koji su, istražujući neurofizičke karakteristike živih bića, objavili McCulloch i Pitts. Računala tog doba nisu imala procesnu moć da bi implementirala neuronske mreže no danas je situacija sasvim drugačija te su neuronske mreže postale nezaobilazan koncept u razvoju inteligentnih sustava. Umjetna neuronska mreža je umjetna replika ljudskog mozga kojom se želi simulirati postupak učenja, tj, skup međusobno povezanih procesnih elemenata čija se funkcionalnost temelji na biološkom neuronu. Svoju primjenu našle su u tehničkim znanostima, društvenim znanostima, ekonomiji te mnogim drugim područjima. Jedna od čestih primjena neuronskih mreža je raspoznavanje uzoraka što je i glavni zadatak našeg programa. Kod raspoznavanja uzoraka potrebno je realizirati klasifikator koji za ulazne podatke (vektore) određuje klasu kojoj taj vektor pripada. Neuronska mreža je zapravo nelinearni klasifikator koja dijeli ulazni vektorski prostor uzoraka u klase koje imaju nelinearne granice. Naš zadatak je napraviti program za klasifikaciju rukom pisanih znamenki (skup MNIST koji predstavlja slike znamenki od 0 do 9). Za programsko rješenje ovog zadatka korišten je Python uz uporabu biblioteka Matplotlib i Numpy. Matplotlib biblioteka služi za izradu grafova u Pythonu, a Numpy omogućuje svođenje iterativnih postupaka na matrične i vektorske izraze. Sljedeća poglavlja sadrže opisani rad na projektu: u poglavlju 2. dana je arhitektura korištene mreže dok su u 3. i 4. redom opisana funkcija gubitka te gradijentni spust. U poglavlju 5. je opisana numerička provjera gradijenta. Poglavlje 6. objašnjava regularizaciju, a u 7. se poglavlju opisuje izvedba u PyTorchu. Osmo poglavlje daje opis proizvoda dok u 9. poglavlju sadrži tehničke značajke proizvoda. u desetom poglavlju su upute za korištenje dok su zadnja dva poglavlja iskorištena za zaključak projekta i popis literature.

2. Arhitektura mreže

Arhitektura ili topologija mreže određuje način na koji su neuroni međusobno povezani. Postoje četiri osnovne vrste mreža: unaprijedne mreže, mreže s povratnom vezom, lateralno povezane mreže i hibridne mreže. U našem projektu koristimo unaprijednu (feedforward) potpuno povezanu mrežu. U našoj mreži koristimo dva sloja neurona. U prvom sloju korištena prijelazna funkcija je zglobnica (ReLu) dok je u drugom sloju prijelazna funkcija softmax. Mreža može koristiti razne postupke optimizacije (SGD, SGD s momentom, ADAM), regularizacije (L1, L2, rano zaustavljanje) te različite funkcije gubitka (L1, glatki L1, L2, Hinge) koji se odabiru pomoću komandne linije. Mreža također implementira numeričku provjeru gradijenta za dodatnu točnost.

3. Funkcija gubitka

Funkcija gubitka je funkcija koja mapira vrijednosti kao realne brojeve koji predstavljaju neki "trošak", a to radi tako da računa razliku između očekivanog i stvarnog izlaza mreže. Tipičan primjer funkcije gubitka (1) je prosjek negativne log-izglednosti modela preko svih podataka:

$$L(\mathbf{W_1}, \mathbf{b_1}, \mathbf{W_2}, \mathbf{b_2}|\mathbf{X}, \mathbf{y}) = -\frac{1}{N}\sum_i \log P(Y = y_i|\mathbf{x}_i) \quad \textbf{(1)}$$

Gubitak ***L*** ovisi o vjerojatnostima ***P*** koje ovise o linearnoj klasifikacijskoj mjeri drugog sloja **s2** koja ovisi o prvom skrivenom sloju **h1** te parametrima **W2** i **b2**. Nadalje, prvi skriveni sloj **h1** ovisi o **s1** koja ovisi o parametrima **W1** i **b1** te podacima **x.** Iz toga razloga gradijente gubitka obzirom na parametre određujemo ulančavanjem. Cij optimizacije je dovesti funkciju gubitka na što manju vrijednost te sam optimizacijski postupak definira na koji način funkcija gubitka utječe na napredovanje parametara modela.
Kao funkcije gubitka implementirali smo module **negativne log-izglednosti** (CrossEntropyLoss), **L1 gubitka** (L1 Loss), **L2 gubitka** (L2 Loss), **glatki L1 gubitak** (L1 smooth Loss) i **zglobnicu** (Hinge Loss). L1 i L2 gubitak su tipične funkcije gubitka korištene u strojnom učenju, glatki L1 gubitak je neprekidna diferencijabilna verzija L1 gubitka s L2 gubitkom za apsolutnu vrijednost argumenta manju od 1, dok je zglobnica važna u klasifikaciji za stroj s potpornim vektorima (SVM). Svaka od funkcija gubitka ima za cilj minimizirati grešku pri učenju modela.

4. Gradijentni spust

Gradijentni spust je najjednostavnija metoda gradijentne optimizacije koja koristeći iterativno pomicanje u smjeru negativnog gradijenta može dovesti do globalnog minimuma. U svakom se koraku mijenja vektor težina u smjeru najvećeg spusta niz plohu pogreške i tako sve dok ne dostigne minimum. Promatramo li gradijent kao stupčani vektor, gradijent će odgovarati transponiranoj Jakobijevoj matrici. Aproksimiramo li funkciju koristeći Taylorov razvoj prvog reda dobijemo sljedeće (2) :

$$f(\mathbf{x} + \Delta\mathbf{x}) = f(\mathbf{x}) + \frac{df(\mathbf{x})}{d\mathbf{x}}\Delta\mathbf{x} = f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \Delta\mathbf{x}. \quad \textbf{(2)}$$

Uvrstimo li pomak u smjeru negativnog gradijenta (3) u formulu (2) vidimo da bi vrijednost funkcije ***f*** trebala padati (4) :

$$\Delta\mathbf{x} = -\epsilon \cdot \mathbf{g},\ \mathbf{g} = \nabla f(\mathbf{x}) \quad \textbf{(3)}$$

$$f(\mathbf{x} - \epsilon \cdot \mathbf{g}) = f(\mathbf{x}) - \epsilon \cdot \mathbf{g}^\top \mathbf{g}\ . \quad \textbf{(4)}$$

Točnost aproksimacije u praksi ovisi o vrijednosti hiper-parametra epsilon, što je on manji,

aproksimacija je točnija. Glavni problemi gradijentnog spusta su spora konvergencija u minimum te to što ne garantira pronalazak globalnog minimuma u slučaju više lokalnih minimuma.

5. Numerička provjera gradijenata

Implementacija postupka analitičkog određivanja gradijenata u neuronskoj mreži nije uvijek jednostavna i lako je pogriješiti. S druge strane, numeričko računanje gradijenta je vrlo jednostavno za implementirati i pretpostavlja samo točno implementiran unaprijedni prolaz. Jednom implementiran, postupak numeričkog računanja se vrlo lako može koristiti neovisno o modelu neuronske mreže. Kako izračunati gradijenti predstavljaju aproksimaciju stvarnog gradijenta tako se mogu usporediti s rezultatom algoritma za njihovo analitičko određivanje. Ako je razlika između vektora analitičkog i numeričkog pristupa unutar definirane tolerancije, analitički algoritam se može smatrati uspješnim.

6. Regularizacija

Da bi poboljšali model, uz funkcije gubitka L1 i L2 implementirali smo i odgovarajuće regularizatore L1 odnosno L2 regularizator. Osim prethodno spomenutih, implementirali smo regularizator ranog zaustavljanja (early stopping) koji računa optimalan broj iteracija učenja. Algoritam se zaustavlja u trenutku kad greška na skupu za testiranje počinje rasti odnosno na skupu za učenje padati. Regularizatori se koriste kako bi smanjili problem prenaučenosti modela (overfitting). Oni povećavaju grešku na skupu za učenje u korist smanjenja greška na skupu za testiranje te poboljšavaju sposobnost generalizacije.

7. Optimizacija

Optimizacijske se metode koriste kako bismo minimalizirali očekivanu pogrešku generalizacije. Najjednostavnija metoda optimizacije je gradijentni spust, no ona može često naići na probleme tijekom treniranja mreže koji mogu usporiti tijek učenja, ili još gore - spriječiti pronalazak optimalnih težina. Neki od tih problema su lokalni minimumi i sedla u funkciji čiji su gradijenti jednaki ili teže nuli, a mogu se izbjeći korištenjem sofisticiranijih algoritama poput stohastičkog gradijentnog spusta, stohastičkog gradijentnog spusta s momentom i ADAMA koji su korišteni i u ovom programu. Stohastički gradijentni spust u svakom koraku optimizacije procjenjuje iznos gradijenta zbog čega uvijek postoji šum pa u lokalnom/globalno minimumu procijenjeni gradijent ne pada na nulu. Uporaba momenta koristi se kao metoda za ubrzavanje učenja, dok ADAM prati uprosječeno kretanje gradijenta i kvadrata gradijenta uz eksponencijalno zaboravljanje što ga čini dobrim izborom za rad sa širokim skupom parametara.

8. Izvedba u PyTorchu i izvođenje na NCS2

Drugi pristup zadatku klasifikacije znamenki iz skupa MNIST je korištenje biblioteke PyTorch. PyTorch je biblioteka strojnog učenja temeljena na biblioteci Torch. PyTorch je razvijen od strane Facebookova laboratorija za istraživanje umjetne inteligencije (FAIR), a iako je osnovni fokus temeljen na Pythonu, postoji i C++ sučelje. Pytorch pruža dvije karakteristične značajke, a to su tenzorsko računanje s velikim ubrzanjem putem grafičkih procesorskih jedinica (GPU) te mogućnost korištenja i modeliranja neuronskih mreža.

U prvoj fazi našeg rada, pomoću Pytorcha, implementirani su različiti modeli neuronskih mreža pomoću kojih će biti testirane i klasificirane znamenke iz MNIST baze podataka. Prva četiri radna paketa u svome radu su se dotakli i implemenirali sve funkcije koje ćemo mi ostvariti pomoću PyTorcha. Pytorch, budući da je zasnovan na biblioteci Torch, nudi mogućnost korištenja već implementiranih funkcija za optimizaciju, regularizaciju, itd.

U sljedećoj fazi rada, kako bismo pripremili istrenirane modele mreže za evaluaciju na Intelovom Neural Compute Stick-u, morali smo modele mreža spremiti u ONNX format budući da NCS2 ne podržava PyTorch modele.

ONNX(Open Neural Network Exchange) je otvoreni ekosustav koji omogućuje odabir pravih alata pri razvoju projekta. ONNX nudi otvoreni izvorni format za modele neuronskih mreža te podržava duboko i strojno učenje. ONNX je široko podržan i može ga se naći u mnogim okvirima, alatima i hardveru. Omogućivanje interoperabilnost između različitih okvira i pojednostavljuje put od istraživanja do proizvodnje što ubrzava pojavu inovacija.

Kako bi se moglo evaluirati modele mreža na NCS2 potrebno je postaviti odgovarajuće okruženje. Intel OpenVINO Tools nudi niz mogućnosti i operacija za lakše baratanje s modelima neuronskih mreža. Zanimljiva stvar kod OpenVINO-a je što su pojedine testne skripte napisane na način kako je potrebno minimalno izmjena kako bi se one mogle pokretati i služiti za provjeru ili testiranje odgovarajućeg modela mreže. Upravo jedna od već implementiranih skripti je korištena, a riječ je o skripti classification_sync.py koja je dodatno konfigurirana i prilagođena modelu te ulazima naše mreže.

9. Opis razvijenog proizvoda

Program koristi dvoslojnu unaprijednu potpuno povezanu neuronsku mrežu za klasifikaciju

znamenki iz skupa MINST. Program također koristi regularizaciju, optimizaciju i usporedbu analitički izračunatog gradijenta s numeričkim za dodatnu točnost. Isto tako je omogučen odabir različitih optimizacija, regularizacija i funkcija gubitka.

Kod implementacije u PyTorchu, korištena su 3 različita modela mreže - potpuno povezani, konvolucijski i resnet18. Potpuno povezani i konvolucijski model samostalno su implementirani korištenjem klase Net, dok je implementacija modela resnet18 napravljena po uputama službene dokumentacije PyTorcha uz male korekcije kako bi model mreže mogao čitati slike MNIST skupa. Nadalje, za svaki model je provedena validacija na 2000 slika za različiti broj parametara kako bismo za konačni model mogli uzeti odgovarajuće parametre koji daju najbolji rezultat. Za izvednu na OpenVINO-u korištene su testne već implementirane skripte koje su konfigurirane da odgovaraju našim modelima i njihovim ulaznim parametrima.

10. Tehničke značajke

Funkcije gubitka modelirane su apstraktnom klasom `LossFunction` i smještene u paket `losses`. Dvije metode koje svaki gubitak standardno ima su prolaz unaprijed i prolaz unatrag. Tijekom prolaza unaprijed (forward) računa se podatkovni gubitak modela u koji je uključen i gubitak zbog regularizacije, ako postoji. Prolazom unatrag računaju se gradijenti ulaza iz prethodnog sloja (`backward_inputs`) i parametara (`backward_params`). Gradijenti parametara računaju se ukoliko postoje parametri kao što je slučaj s regularizacijom (regulariziramo težine modela).

Regularizatori su modelirani apstraktnom klasom `Regularizer` i smješteni su u paket `regularizers`. Dvije metode koje ima regularizator imaju isti naziv kao i metode funkcija gubitka, a to su metode za prolaz unaprijed (`forward`) i unazad (`backward_params`). Njih pozivamo u istoimenim metodama funkcija gubitka ukoliko je regularizacija uključena. Ukoliko želimo primijeniti jedan ili više regularizatora uz zadanu funkciju gubitka, u prolazu unaprijed ukupnom gubitku pribrajamo regularizacijski gubitak, a u prolazu unazad računamo gradijent parametara - težina (`weights`), dok je pristranost (bias) zanemariva pa se na nju ne primjenjuje regularizacija.

Regularizacija ranim zaustavljanjem implementirana je kao posebna metoda treniranja zbog drugačije prirode. Ona optimizira broj iteracija potrebnih za treniranje modela tako da trenira model u više manjih iteracija. Nakon svake grupe iteracija provjerava uspješnost modela na validacijskom skupu te uspoređuje razliku gubitaka prije i poslije trenutne grupe iteracija.

Modeli mreža u PyTorchu implementirani su korištenjem klase Net te funkcija gubitka (Cross Entropy Loss i Negative log-likelihood loss) i optimizacije (SGD) iz biblioteke Torch. Također, parametri koji su korišteni za same modele jesu learning rate u vrijednosti 0.001 te momentum u vrijednosti 0.9. Za ostvarenje samog treniranja, validacije i testiranja korištene su funkcije koje su prikazane na dolje priloženom screenshotu.

```
def train(epoch):
    network.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = network(data)
        loss = F.cross_entropy(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % print_interval == 0: #ispisujemo samo neke da ne bude nakracan ispis
              print('Train| Epoch {}| {}/{} ({:.0f}%)| Loss: {:.6f}'.format(
                    epoch, batch_idx * len(data), len(train_loader.dataset),
                    100. * batch_idx / len(train_loader), loss.item()))
```

Slika 10.1: Funkcija za treniranje modela

```
def valid(epoch):
    network.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in valid_loader:
            data, target = data.to(device), target.to(device)
            output = network(data)
            test_loss += F.cross_entropy(output, target, size_average=False).item()
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()
    test_loss /= len(valid_loader.dataset)
    print('Valid| Epoch {}| Avg. loss: {:.4f}| Accuracy: {}/{} ({:.2f}%)'.format(epoch,
    test_loss, correct, len(valid_loader.dataset),
    100.*correct / len(valid_loader.dataset)))
    accuracy_list.append(100.*correct / len(valid_loader.dataset))
```

Slika 10.2: Funkcija za validaciju modela

```
def test():
    network.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = network(data)
            test_loss += F.cross_entropy(output, target, size_average=False).item()
            pred = output.data.max(1, keepdim=True)[1]
            correct += pred.eq(target.data.view_as(pred)).sum()
    test_loss /= len(test_loader.dataset)
    print('Tests| Avg. loss: {:.4f}| Accuracy: {}/{} ({:.2f}%)'.format(test_loss, correct,
            len(test_loader.dataset),100.*correct / len(test_loader.dataset)))
```

Slika 10.3: Funkcija za testiranje modela

U drugoj fazi radnog paketa 5, prilikom rada s NCS2 i alatom OpenVINO korištene su već implementirane skripte za konverziju ONNX modela u IR (skripta mo_onnx.py) te skripta classification_sync.py za provedbu testiranja slika.

Nadalje, na samom NCS2 provedena je usporedba performansi mreža u odnosu na CPU (Intel i5 6300hq) i GPU (Nvidia Geforce GTX 950M). Za usporedbu je korišten model mreže resnet18, a kod koji se koristio za testiranje performansi za CPU i GPU je prikazan na slici 10.4.

```
batch_size = 1
q = 28 # 128, 256, 512, 1024
device = torch.device('cuda')
network.eval()
network.to(device)

data = torch.randn(batch_size,3,q,q)

with torch.no_grad():
    data_network = torch.randn(batch_size,3,q,q).cuda()
    logits = network.forward(dataa)
    torch.cuda.synchronize()
    time0 = 1000*time.perf_counter()
    for _ in range (n):
        data_network = torch.randn(batch_size,3,q,q).cuda()
        logits = network.forward(dataa)
        _, pred = logits.max(1)
        out = pred.data.byte().cpu()
        torch.cuda.synchronize()
    time1 = 1000*time.perf_counter()
fps = (1000*n)/(time1-time0)
```

Slika 10.4: Kod za testiranje performansi za CPU i GPU

Za testiranje performansi za NCS2 se koristio već implementirani benchmark_app.py. Modeli su testirani za ulazne random tenzore veličine (q*q*3), gdje je q = 28, 128, 256, 512 ili 1024 te je dobiveno kako su najbolje performanse na GPU.

Numerička provjera gradijenta je implementirana kao funkcija u zasebnom modulu grad_check. Funkcija uspoređuje predani gradijent s numerički izračunatim gradijentom. Za numeričko računanje gradijenta se koristi centralna diferencija s korakom h. Taj se parametar može podesiti zajedno s preostalim parametrima mreže. U praksi se najčešće koristi 10^{-4}, 10^{-5}. Tolerancija razlike između vektora gradijenata dobivenih analitički odnosno numerički se također može podesiti u parametrima.

Optimizacijski algoritmi implementirani su na svoje uobičajene načine: dok su prva dva, SGD i SGDM, česti, uobičajeni i jednostavni te ne postoje brojni različiti načini implementacije, Adam je implementiran prema uputama iz znanstvenog rada u kojem je originalno predstavljen. Parametri, koje može zadati i korisnik, unaprijed su zadani i uzeti iz različitih izvora. Zadani parametri su sljedeći:

- SGD:

  learning_rate = 0.01 (brzina konvergencije)

- SGDM:

  learning_rate = 0.01 (brzina konvergencije)

  momentum = 0.9 (iznos koji označava koliku će ulogu zamah imati pri konvergenciji)

- Adam:

  learning_rate = 0.1 (brzina konvergencije)

  beta1 = 0.9 (iznos eksponencijalnog raspada za prvi moment)

  beta2 = 0.999 (iznos eksponencijalnog raspada za drugi moment)

Umjesto nule, u implementaciji algoritma Adama, dijeli se s 10^(-8), što je također moguće mijenjati.

11. Upute za korištenje

11.1 Upute za glavni program

Programu se preko argumenata naredbenog sučelja (CMD-a) pri pokretanju zadaju različiti moduli i opcije za parametre, regularizaciju, funkciju gubitka, optimizaciju i odabir korištenja ranog zaustavljanja. Ti se argumenti zadaju sljedećim redoslijedom:

[-h] [--params PARAMS] [--loss LOSS] [--optimizer OPTIMIZER] [--early_stopping]

gdje PARAMS i LOSS su imena modula sa parametrima i funkcijom gubitka bez .py nastavka, a OPTIMIZER ime algoritma optimizacije. Pokretanje programa sa argumentom -h će ispisati pomoč pri korištenju i ugasiti program. Program u slučaju izostanka bilo kojih od navedenih argumenata koristi pretpostavljenu vrijednost (za PARAMS parameters, za LOSS CrossEntropyLoss, za OPTIMIZER SGD, rano zaustavljanje isključeno). Ako korisnik želi koristit svoje module za parametre, funkcije gubitka i optimizator mora slijediti sljedeća pravila. Modul sa parametrima mora imati parametre s sljedećim imenima: niter, learning_rate_bias, learning_rate_sgd, learning_rate_sgdm, learning_rate_adam, momentum, beta_1, beta_2, epsilon, valid_set_factor, n_eval, patience, weight_decay, hinge_margin, tolerance, h. Modul sa funkcijom gubitka mora sadržavati klasu Loss koja ima konstruktor koji prima model mreže, modul s parametrima, regularizacijsku klasu te vektor točnih klasa podataka. Modul sa optimizacijskim algoritmima se mora zvati optimizator te mora sadržavati klasu koja se zove Optimizator i u konstruktoru prima model mreže, modul s parametrima i string koji sadrži ime algoritma koji se želi koristiti. Ta klasa također treba implementirati mogućnost odabira algoritma preko stringa koji se dobije u konstruktoru. Nakon što se program pokrene automatski se započinje treniranje mreže te se svakih 10 iteracija treniranja ispisuje gubitak, a svakih 100 razlika između analitičkog i numeričkog gradijenta. Pri završetku treniranja se ispisuje za set za učenje ukupna točnost klasifikacije, matrica zabune te vektori preciznosti i odziva pojedinih razreda. Nakon toga se mreža testira na setu za provjeru te se ponovno ispisuju ranije navedeni parametri i time završava program.

11.2 Upute za korištenje sučelja za optimizaciju

Algoritam optimizacije poziva se u dva osnovna koraka: stvaranje instance razreda Optimizator koji sadrži algoritme te pozivanje metode. Na slici 11.2.1. prikazano je tijelo konstruktora razreda Optimizator.

```
def __init__(self, model, params, algorithm):
    self.algorithm = algorithm
    self.model = model
    self.params = params
    self.iterations = 0
    self.domain_point_derivative_1 = 0
    self.domain_point_derivative_2 = 0
    self.first_moment_vector_1 = 0
    self.first_moment_vector_2 = 0
    self.second_moment_vector = 0
```

Slika 11.2.1. Inicijalizacija modela, parametara i ključa

Instanca razreda Optimizator stvara se pozivanjem konstruktora s idućim potpisom: (self, model, params, algorithm) predavši mu model kojim se bavimo, ime modula koji sadrži parametre te ključ u stringu kojim biramo korišteni algoritam.

Sam korak algoritma poziva se zvanjem metode inačica_optimizatora(grad_W1, grad_W2), odnosno predavši vektore gradijenata težina. Ukoliko želite krenuti s optimizacijom ispočetka, to se može učiniti stvaranjem nove inačice optimizatora.

```
def __call__(self, grad_W1, grad_W2):
    self.iterations += 1
    if self.algorithm.upper() == "SGD":
        return self.__sgd(grad_W1, grad_W2)
    elif self.algorithm.upper() == "SGDM":
        return self.__sgdm(grad_W1, grad_W2)
    elif self.algorithm.upper() == "ADAM":
        return self.__adam(grad_W1, grad_W2)
```

Slika 11.2.2. Pozivanje algoritma

Na slici 11.2.2. prikazano je tijelo metode koja, ovisno o dobivenom ključu, poziva metodu definicije __inačica_optimizatora(self, grad_W1, grad_W2) i vraća povratnu vrijednost odabranog algoritma.

```
def __sgd(self, grad_W1, grad_W2):
    w1 = np.transpose(self.model.W1) - self.params.learning_rate_sgd * grad_W1
    w2 = np.transpose(self.model.W2) - self.params.learning_rate_sgd * grad_W2

    return w1, w2
```

Slika 11.2.3. Tijelo funkcije SGD algoritma

Slika 11.2.3. pokazuje da optimizacijski algoritmi kao povratnu vrijednost vraćaju optimizirane vektore težina zapisane kao n-torka.

11.3 Upute za pokretanje na NCS2

Za izvedbu na NCS2 potrebno je modele mreža trenirati te prebaciti u ONNX format. Nakon uspješnog formatirnaja, potrebno je pomoću Intelovog OpenVINO alata Model Optimizer stvoreni ONNX format prebaciti u xml i bin datoteku. U xml datototeci zapisane su sve informacije o slojevima mreže, dok su u bin datoteci pohranjeni podaci o parametrima i težinama mreže. Poziv Model Optimizera prikazan je na dolje priloženom screenshotu. Nužno je navesti lokaciju ONNX modela, no sam Model Optimizer nudi veliki broj dodatnih opcija poput tipa podataka, imena novogeneriranih datoteka i slično.

```
C:\Program Files (x86)\IntelSWTools\openvino\deployment_tools\model_optimizer>python mo_onnx.py -m C:\Users\Jelena\Documents\Jelena\FER\ppp\CNN_softmax_CrossEntropyLoss2.onnx -n novi_model -o C:\Users\Jelena\Doc
uments\Jelena\FER\ppp\ --data_type FP16
Model Optimizer arguments:
Common parameters:
        - Path to the Input Model:      C:\Users\Jelena\Documents\Jelena\FER\ppp\CNN_softmax_CrossEntropyLoss2.onnx
        - Path for generated IR:        C:\Users\Jelena\Documents\Jelena\FER\ppp\
        - IR output name:       novi_model
        - Log level:    ERROR
        - Batch:        Not specified, inherited from the model
        - Input layers:         Not specified, inherited from the model
        - Output layers:        Not specified, inherited from the model
        - Input shapes:         Not specified, inherited from the model
        - Mean values:  Not specified
        - Scale values:         Not specified
        - Scale factor:         Not specified
        - Precision of IR:      FP16
        - Enable fusing:        True
        - Enable grouped convolutions fusing:   True
        - Move mean values to preprocess section:       False
        - Reverse input channels:       False
ONNX specific parameters:
Model Optimizer version:        2019.3.0-408-gac8584cb7

[ SUCCESS ] Generated IR model.
[ SUCCESS ] XML file: C:\Users\Jelena\Documents\Jelena\FER\ppp\novi_model.xml
[ SUCCESS ] BIN file: C:\Users\Jelena\Documents\Jelena\FER\ppp\novi_model.bin
[ SUCCESS ] Total execution time: 6.11 seconds.
```

Slika 11.3.1: Poziv Model Optimizera

Nakon uspješno generiranih xml i bin datoteka, moguće je testirati model pomoću već raznih skripti. U našem slučaju, koristili smo skriptu classification_syn.py čije su opcije prikazane na donjem screenshotu.

```
C:\Program Files (x86)\IntelSWTools\openvino_2019.3.379\inference_engine\samples\python_samples\classification_sample>python classification_sample.py -h
usage: classification_sample.py [-h] -m MODEL -i INPUT [INPUT ...]
                                [-l CPU_EXTENSION] [-d DEVICE]
                                [--labels LABELS] [-nt NUMBER_TOP]

Options:
  -h, --help            Show this help message and exit.
  -m MODEL, --model MODEL
                        Required. Path to an .xml file with a trained model.
  -i INPUT [INPUT ...], --input INPUT [INPUT ...]
                        Required. Path to a folder with images or path to an
                        image files
  -l CPU_EXTENSION, --cpu_extension CPU_EXTENSION
                        Optional. Required for CPU custom layers. MKLDNN
                        (CPU)-targeted custom layers. Absolute path to a
                        shared library with the kernels implementations.
  -d DEVICE, --device DEVICE
                        Optional. Specify the target device to infer on; CPU,
                        GPU, FPGA, HDDL, MYRIAD or HETERO: is acceptable. The
                        sample will look for a suitable plugin for device
                        specified. Default value is CPU
  --labels LABELS       Optional. Path to a labels mapping file
  -nt NUMBER_TOP, --number_top NUMBER_TOP
                        Optional. Number of top results
```

Slika 11.3.2: Opcije skripte classification_syn.py

Nužno je navesti put do modela te slike koju se testira, a ukoliko se ne navede opcija -d (DEVICE) podrazumijeva se izvođenje na CPU. Kako bi se skripta pokrenula na NCS2 potrebno je, uz put do modela i slike, navesti i zastavicu -d MYRIAD.

11.4 Upute programa za regularizaciju i funkcije gubitka

Algoritam rada funkcije gubitka s omogućenom regularizacijom odvija se u tri osnovna koraka. Prije početka treniranja instancira se razred željenog gubitka (CrossEntropyLoss, L1Loss, L2Loss, L1SmoothLoss ili HingeLoss) konstruktorom potpisa: __init__(self, Y_, regularizers) gdje Y_ označava indekse točnih klasa, a regularizers je lista regularizatora. Željeni regularizatori (L1Regularizer, L2Regularizer) se dodaju u listu regularizatora te se instanciraju konstruktorom potpisa: __init__(self, weights, weight_decay, name), weights je referenca na tenzor težina koje se regulariziraju, weight_decay je faktor regularizacije lambda (reda veličine exp(-3), exp(-2), exp(-1)), a name je ime regularizatora.

Tijekom treniranja, poziva se funkcija forward(self, scores) gdje su scores izlazi iz zadnjeg sloja modela, tj. klasifikacijske mjere klasa koje pokazuju koje su klase vjerojatnije u odnosu na druge. Unutar funkcije forward se za svaki od regularizatora (specificiranih prije treniranja) poziva metoda forward(self) samog regularizatora koja vraća gubitak zbog regularizacije. On se pribroja gubitku zbog pogreške predviđanja modela i tako se dobije ukupni gubitak modela koji funkcija forward(self, scores) funkcije gubitka vraća.

Na kraju se pozivaju metode backward_inputs(self, previous_input) i backward_params(self) koje vraćaju podatkovni, odnosno regularizacijski gubitak. Regularizacijski gubitak se vraća u obliku liste u kojoj se nalaze parovi [weights, grad_weights], weights je referenca na tenzor za koji su izračunati gradijenti grad_weights.

Regularizacija ranim zaustavljanjem koristi se tako da se prije treniranja skup za treniranje dodatno podijeli, te dobijemo skup za validaciju. Metodi early_stopping predamo te skupove te dodatno predamo n - broj iteracija nakon kojih se model validira i p (patience) - broj koraka koliko dopuštamo da se validacijska greška povećava. Metoda vraća optimalni broj iteracija, naučene parametre modela i postignutu minimalnu grešku.

12. Zaključak

Glavni cilj projekta bio je razviti program koji klasificira rukom pisane znamenke. To smo postigli koristeći unaprijednu dvoslojnu neuronsku mrežu te smo upotrijebili znamenke iz skupa MINST. Nakon učenja na skupu od 60000 uzoraka, program se testirao na 10000 uzoraka i dao zadovoljavajuće rezultate klasifikacije. Program sadrži razne opcije poput više vrsta optimizatora (SGD, SGD s momentom te ADAM), regularizatora(L1, L2, rano zaustavljanje) te funkcija gubitaka(L1,L1 glatki, L2, Hinge, CrossEntropy). Ovisno o odabiru prethodno navedenih opcija dobivamo bolje ili lošije rezultate. Konkretno, najbolji rezultat je zapažen u slučaju korištenja gubitka L1, SGDM optimizatora te L1 regularizatora. U navedenom slučaju točnost na testiranom uzorku bila je 90.84%, a gubitak 11208.28. U daljnjem ispitivanju pronađen je i najlošiji slučaj u kojemu je točnost na niskih 9.08%, a gubitak 63.11. Za taj je slučaj korišten HingeLoss te SGD optimizator. Prilikom izrade ovog programa korišten je i drugi pristup, tj. izvedba na NCS2(Neural Compute Stick). Za rad na prethodno navedenom Intelovom sticku, što

je nedvojbeno najzanimljiviji dio projekta, bilo se potrebno spremiti modele mreža u ONNX(Open Neural Network Exchange) format. Primjena ONNX-a je velika i može ga se naći u mnogim alatima te hardverima jer upravo on omogućuje interoperabilnost između različitih okvira. Uzevši u obzir karakteristike navedenog formata lako se vidi zašto isti ubrzava pojavu inovacija. Iako je primarna svrha ovoga projekta bila edukacija, jasno je kako ovakav program lako nalazi uporabu u širokom spektru industrija te općeniti interes za ovakvu vrstu programa svakodnevno raste.

13. Literatura

Casper Hansen, Optimizers explained, 16.10.2019., https://mlfromscratch.com/optimizers-explained/, 15.11.2019.

Kingma, D. P., Lei Ba, J., Adam: A Method for Stochastic Optimisation, ICLR 2015 : International Conference on Learning Representations 2015, San Diego, Kalifornija, SAD (2015.)

I. Krešo, Github projekt fer-deep-learning, 13.2.2017., https://githuhttps://github.com/ivankreso/fer-deep-learning/b.com/ivankreso/fer-deep-learning/, pristup: 17.1.2020.

J. Krapac, S. Šegvić, Regularizacija dubokih modela, http://www.zemris.fer.hr/~ssegvic/du/du4regularization.pdf, pristup: 12.1.2020.

Eli Bendersky. The Softmax function and its derivative, https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/, pristup: 12.1.2020.

Stanford edu, CS231n Convolutional Neural Networks for Visual Recognition, http://cs231n.github.io/neural-networks-case-study/#grad, pristup: 10.1.2020.

M. Čupić, Optimizacija parametara modela - Prezentacija za kolegij Duboko učenje, FER, http://www.zemris.fer.hr/~ssegvic/du/du3optimization.pdf, pristup: 15.1.2020.

Ravindra Parmar, Towards Data Science: Demystifying optimizations for machine learning, https://towardsdatascience.com/demystifying-optimizations-for-machine-learning-c6c6405d3eea, pristup: 15.1.2020.