Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

Matija Patajac

Problem klasifikacije programskog koda

Završni praktični projekt - dokumentacija

Osijek, 2021.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

Matija Patajac

Problem klasifikacije programskog koda

Završni praktični projekt - dokumentacija

Mentor: izv. prof. dr. sc. Domagoj Matijević

Sumentor: Antonio Jovanović

Osijek, 2021.

Sadržaj

1 Uvod

1.1 O problemu

Ovaj projekt bavi se problemom klasifikacije programskog koda - za dani isječak koda, potrebno je odrediti kojoj kategoriji on pripada, odnosno što taj isječak koda radi.
Konkretno, koristimo POJ-104 set podataka koji se sastoji od 104 različita problema i 500 pripadnih rješenja za svaki od tih problema. U sklopu projekta razvijen je model koji s velikom točnošću (oko 94%) klasificira ovaj skup podataka.

1.2 Tehnologije

Projekt je napisan koristeći programski jezik Python i sljedeće Python module:

- pytorch - rad s modelom
- pickle - spremanje i učitavanje obrađenih podataka
- matplotlib - vizualizacija funkcije troška pri učenju modela
- sklearn - određivanje točnosti (engl. *accuracy*) naučenog modela
- argparse - parsiranje argumenata proslijeđenih u naredbenom retku pri pokretanju programa

Uređaj
Budući da strojno učenje zahtjeva vrlo velik broj matematičkih operacija koje se često mogu izvoditi paralelno, nerijetko se u tu svrhu koristi grafička kartica. Takav je slučaj i u ovom projektu - program će tokom rada s modelima umjesto procesora koristiti CUDA jezgre[1] s NVIDIA grafičke kartice ukoliko je ona dostupna.

1.3 Organizacija koda

Moduli

Ovaj projekt sastavljen je od 4 modula:

- `main` - *omotač* oko čitavog programa; pruža jednostavno sučelje za pokretanje i korištenje programa
- `data_handler` - sadrži logiku za rad s podacima (obrađivanje, spremanje, učitavanje)
- `model` - sadrži sav kod vezan za stvaranje te korištenje modela
- `utility` - modul s izvdojenim podacima i funkcijama koji su potrebni unutar više drugih modula (kako bi se izbjegla duplikacija koda)

[1] https://developer.nvidia.com/cuda-zone

Konvencije imenovanja

Projekt prati sljedeću konvenciju imenovanja u kodu:

- imena klasa koriste Pascal Case[2]
- ostatak koda (varijable, funkcije, metode, moduli) u svome imenu koristi Snake Case[3]

Github repozitorij

Kroz ovu tehničku dokumentaciju se na mjestima pojavljuju isječci koda kao dodatak tekstualnom objašnjenju, no kod u tim isječcima je najčešće izoliran od svog okruženja i/ili ne slijedi nužno poredak iz samog programa. Za pristup izvornom kodu u cjelosti, posjetite pripadni Github repozitorij[4] ovog projekta.

[2] https://www.theserverside.com/definition/Pascal-case
[3] https://en.wikipedia.org/wiki/Snake_case
[4] https://github.com/mpatajac/algorithm-classification

2 Korištenje

2.1 `main`

Modul `main` je pokretač (engl. *driver*) modul - njime pokrećemo čitav program. Možemo ga koristiti na dva načina:

- zadani - pokreće učenje novog modela s proslijeđenim hiperparametrima, potom ga testira
- evaluacijski - samo testira već postojeći model

Pri pokretanju programa prosljeđujemo *argumente naredbenog retka* (engl. *command line arguments*) kojima upravljamo izvođenjem samog programa. Slijedi njihov popis.

Argument	Opis	Tip	Zadana vrijednost
`--dropout`	vjerojatnost korištena u `dropout` komponenti	`float`	0.2
`--layers`	broj naslaganih (engl. *stacked*) rekurentnih (LSTM) slojeva	`int`	1
`--embedded`	dimenzija prostora u kojeg se *ugrađuju* (engl. *embed*) ulazne vrijednosti	`int`	100
`--hidden`	dimenzija vektora *skrivenog stanja* (engl. *hidden state*) u LSTM-u	`int`	200
`--batch-size`	broj ulaznih vrijednosti u svakoj *hrpi* (engl. *batch*)	`int`	64
`--epochs`	broj *epoha* (koliko puta model uči skup podataka za učenje)	`int`	3
`--save-name`	pod kojim nazivom će se model spremiti	`str`	`model`
`--attention`	na koji će se način `attention` vrijednosti korisiti	`str`	`cat`
`-v`, `--verbose`	koristi li se verbozan način (ispis)	`bool`	`False`
`--bidirectional`	koristi li se dvosmjerni LSTM	`bool`	`False`
`--evaluate`	koristi li se način za evaluaciju	`bool`	`False`

Tablica 1: Popis *argumenata naredbenog retka* koje program podržava.

Zadani način

Zadani način rada `main` modula učita sva tri skupa podataka (za učenje, validaciju i testiranje) te ih razdvoji na *hrpe* (engl. *batch*) veličine `--batch-size`, potom napravi novi model koristeći proslijeđene hiperparametre (`--embedded`, `--hidden`, `--layers`, `--bidirectional`, `--dropout` i `--attention`), taj model uči `--epochs` epoha, spremi ga pod nazivom `--save-name` te ga u konačnici testira na testnom skupu.

Primjer

Novi model, koji ima 2 sloja, učimo 5 epoha i pritom koristimo verbozan ispis:

```
python main.py -v --layers=2 --epochs=5
```

Evaluacijski način

U evaluacijskom načinu rada program učita model spremljen pod nazivom danim u `--save-name` argumentu te ga testira na skupu podataka za testiranje. Za korištenje evaluacijskog načina potrebno je pri pokretanju proslijediti `--evaluate` zastavicu.

Primjer

Testiramo model spremljen u datoteci `moj_model.pt`:

```
python main.py --evaluate --save-name=moj_model
```

2.2 Drugi moduli

Iako se ne preporučuje, zasebno pokretanje pojedinačnih modula (npr. u svrhu provjere ispravnosti koda) moguće je kroz `if __name__ == "__main__"` blok. U nastavku navodimo nekoliko primjera takve upotrebe.

`data_handler`

Indeksirani skup podataka pretvaramo u kompaktan oblik.

```
if __name__ == "__main__":
    prepare("test")
```

Učitavamo skup podataka i ispisujemo prvi (indeksirani) podatak i njemu pripadnu oznaku (kategoriju) iz prve *hrpe*.

```
if __name__ == "__main__":
    loader = get("test")
    for batch in loader:
        indices, categories, _ = batch
        print(indices[0], categories[0])
        break
```

`model`

Učitavamo naučeni model pod nazivom `moj_model` i testiramo ga na testnom skupu podataka.

```
if __name__ == "__main__":
    import data_handler

    test_loader = data_handler.get("test")
    classifier = ReviewClassifier.load("moj_model")
    test(classifier, test_loader, "cpu")
```

3 Implementacija

3.1 Rad s podacima

Za početak, pogledajmo kako izgledaju podaci u svom izvornom obliku, kako ih preprocesiramo u komprimirani oblik i kako ih u konačnici koristimo s našim modelom.

O podacima
Kao što smo spomenuli u uvodnom dijelu, koristimo POJ-104[5] set podataka koji se sastoji od 104 problema i 500 rješenja svakog problema napisanih u C/C++ programskom jeziku. Budući da je C/C++ (kao i svi drugi "standardni" programski jezici) stvoren tako da bude jasan programerima, a ne računalima, umjesto samog izvornog koda koristili smo *srednju reprezentaciju* (engl. *intermediate representation*) koda za LLVM[6] platformu (*LLVM IR*). Na taj smo način osigurali 3 stvari:

1. prilagođenost računalu - LLVM IR svojom je sintaksom bliži asemblerskom jeziku nego jeziku više razine, stoga mu je skup naredbi jednostavniji, što je pogodno za učenje modela

2. neovisnost o platformi - budući da LLVM podržava gotovo sve procesorske arhitekture[7], nemamo ograničenje da sav kod mora biti kompajliran na istoj procesorskoj arhitekturi, kao što bi bio slučaj da smo se odlučili za specifičnu asemblersku sintaksu

3. neovisnost o programskom jeziku - iako je u ovom projektu korišten isključivo jezik C/C++, velika podržanost LLVM platforme kao kompilacijskog odredišta (engl. *compilation target*) od strane programskih jezika[8] omogućuje široku primjenu modela

Kod kompajliranja koda koristili smo postupak predložen u radu [1] - set podijelili na 3 dijela (učenje, validacija, testiranje) u omjeru $3:1:1$ te kod u setu za učenje kompajlirali 8 puta koristeći različite *zastavice* (engl. *flags*): `-O{0-3}` te `-ffast-math`.

3.1.1 Obrada

Obrada podataka sastoji se od dva dijela:

1. mapiranje LLVM IR instrukcija u pripadne indekse

2. povezivanje i komprimiranje čitavog (pod)skupa podataka u jednu datoteku

Kao rezultat obrade dobivamo 3 datoteke (svaka odgovara po jednom podskupu početnoga skupa podataka) s pripremljenim podacima koje potom možemo koristiti u procesu učenja i testiranja našeg modela. Na ovaj način smanjujemo potreban skladišni prostor (`97.2MB` umjesto `3.37GB` koje zauzimaju originalne LLVM IR instrukcije) čime odmah olakšavamo i prijenos podataka, te izbjegavamo potrebu za dugotrajnom obradom podataka pri svakom pokretanju postupka učenja modela. Iz tih je razloga postupak obrade potrebno izvršiti samo ukoliko obrađeni podaci nedostaju, oštećeni su ili je došlo do promjene u načinu obrade pa su postojeći podaci nevažeći.

[5] Poveznica na podatke: https://sites.google.com/site/treebasedcnn/
[6] https://llvm.org/docs/LangRef.html
[7] Popis arhitektura koje LLVM podržava: https://en.wikipedia.org/wiki/LLVM#Back_ends
[8] Popis programskih jezika koje podržavaju LLVM: https://en.wikipedia.org/wiki/LLVM#Front_ends

Mapiranje
Postupak mapiranja LLVM IR instrukcija u indekse proveden je koristeći pripremljeni riječnik i `llvm_ir_to_trainable` funkciju iz skripte `task_utils.py` (uz njene *ovisnosti* (engl. *dependency*) `inst2vec_preprocess.py`, `inst2vec_utils.py` i `rgx_utils.py`)[9] iz [1]. U njima se uz pomoć *regularnih izraza* dodatno smanjuje broj mogućih instrukcija (npr. uklanjanjem identifikatora i numeričkih vrijednosti, *umetanjem* struktura (engl. *inline*) itd.), nakon čega se te pojednostavljene instrukcije mapiraju u pripadni indeks iz riječnika (ili u indeks `!UNK` tokena, ukoliko se instrukcija ne nalazi u riječniku) te spremaju u pripadnu datoteku.

Prije korištenja funkcije napravili smo dvije manje izmjene:

1. bile su nam potrebne samo `.csv` datoteke, stoga smo maknuli dio s `.rec` datotekama
2. budući da riječnik koristi 0-indeksiranje, a nulu koristimo kao *punjenje* (engl. *padding*) u našem modelu, mapirane indekse smo uvećali za 1 prije zapisivanja u datoteku

Promjene su vidljive u sljedećim isječcima koda.

Original:

```
# . . .

# Write to csv
file_name_csv = os.path.join(
    seq_folder, file_names[file_counter][:-3] + '_seq.csv'
)
file_name_rec = os.path.join(
    seq_folder, file_names[file_counter][:-3] + '_seq.rec'
)
with open(file_name_csv, 'w') as csv, open(file_name_rec, 'wb') as rec:
    for ind in stmt_indexed:
        csv.write(str(ind) + '\n')
        rec.write(struct.pack('I', int(ind)))
print('\tPrinted data pairs to file', file_name_csv)
print('\tPrinted data pairs to file', file_name_rec)
print('\t#UNKS', unknown_counter)

# . . .
```

[9]Skripte možete pronaći u ovom repozitoriju: https://github.com/spcl/ncc

Izmijenjena verzija:

```
# . . .

# Write to csv
file_name_csv = os.path.join(
    seq_folder, file_names[file_counter][:-3] + '_seq.csv'
)
with open(file_name_csv, 'w') as csv:
    for ind in stmt_indexed:
        # shift indices by 1 to leave `0` for padding
        csv.write(str(ind + 1) + '\n')
print('\tPrinted data pairs to file', file_name_csv)
print('\t#UNKS', unknown_counter)

# . . .
```

Povezivanje i komprimiranje

Povezivanje i komprimiranje mapiranih indeksa u jednu datoteku radimo koristeći funkciju `prepare` iz modula `data_handler.py`. Ona kao ulaz prima naziv (pod)skupa kojeg želimo pripremiti, a kao rezultat stvara datoteku `{set_name}_set.pt`, gdje je `set_name` proslijeđeni naziv skupa.

Funkcija za početak provjerava je li traženi skup ispravan i postojeći.

```
assert set in ["train", "test", "val"], "Invalid data set."
assert os.path.exists(f".{base_path}/data/seq_{set}"), \
    f"Can't find 'seq_{set}', please create it using \
    `task_utils.llvm_ir_to_trainable(set)`."
```

Ukoliko je provjera uspješno prošla, prolazimo po svim direktorijima (kategorijama) u tom skupu. Broj datoteka unutar svakog direktorija spremimo - njih ćemo kasnije koristiti kako bismo izgenerirali pripadne oznake. Datoteke redom čitamo, niz indeksa pretvaramo u Pythonovu listu te njih spremamo u listu pridruženu toj kategoriji, koju potom dodajemo u listu pridruženom čitavom skupu.

```
all_indices = [[] for _ in range(104)]
category_count = [0 for _ in range(104)]
root_directory_name = f".{base_path}/data/seq_{set}"

# collect
directories = os.listdir(root_directory_name)
for directory in directories:
    # use 0-index
    category = int(directory) - 1

    files = os.listdir(f"{root_directory_name}/{directory}")
    files_in_category = len(files)
    category_count[category] = files_in_category
```

```
    directory_indices = []
    for file in files:
        with open(f"{root_directory_name}/{directory}/{file}", 'r') as f:
            indices = [int(line.strip()) for line in f.readlines()]
            directory_indices.append(indices)

    all_indices[category] = directory_indices
```

Važno je napomenuti da red kojim se čitaju direktoriji nije `1, 2, 3, 4, ...`, već `1, 10, 100, 11, 12, ...` (budući da ih program vidi kao tekst, sortira ih leksikografski), stoga je nužno indekse prvo grupirati prema kategorijama, a potom ih umetnuti na pravo mjesto u listi svih indeksa, umjesto da indekse direktno nadodajemo na sveukupnu listu, čime bi došlo do pogrešnog označavanja (indeksi iz kategorije `10` bi imali oznaku `2`, `100` bi imali `3` itd.). Primjetimo također da oznaku kategorije (odnosno naziv direktorija) smanjujemo za 1 kako bismo mogli koristiti 0-indeksiranje, koje je u skladu s Pythonovim listama. Od ovog trenutka nadalje, kategorije označavamo sa `{0, 1, 2, ..., 103}`, umjesto `{1, 2, 3, ..., 104}`.

Nakon što prikupimo sve indekse, listu svih indeksa *izravnamo* (engl. *flatten*) kako indeksi više ne bi bili razdvojeni po kategorijama, nego bi činili jednu dugačku listu lista.
Dobivenu listu indeksa i broj datoteka po kategoriji spremimo u komprimirani *pickle*[10] format kao `.pt` datoteku, što je često korištena datotečna ekstenzija u projektima koji koriste *PyTorch* ([6]) biblioteku.

```
# flatten
all_indices = list(chain.from_iterable(all_indices))

# save
with open(f".{base_path}/data/{set}_data.pt", "wb") as f:
    pickle.dump({
        "indices": all_indices,
        "category_count": category_count
    }, f, -1)
```

3.1.2 Korištenje

Jednom pripremljene podatke možemo po potrebi učitati koristeći `get` funkciju iz `data_handler.py` modula. Pri pozivu joj prosljeđujemo naziv skupa podataka koji želimo te željenu veličinu *hrpe* (`--batch-size` argument naredbenog retka), a kao rezultat vraća `DataLoader` objekt iz *PyTorch* ([6]) biblioteke kojeg koristimo za lakše organiziranje podataka u *hrpe* te prosljeđivanje modelu.
Nakon inicijalne provjere valjanosti traženog skupa, `get` funkcija učitava podatke te ih prosljeđuje funkciji `to_loader`.

[10] https://docs.python.org/3/library/pickle.html

```
def get(set, batch_size=64):
    assert set in ["train", "test", "val"]
    assert os.path.exists(f".{base_path}/data/{set}_data.pt"), \
        f"Can't find '{set}_data.pt', please prepare it \
        using function `prepare(set)`."

    with open(f".{base_path}/data/{set}_data.pt", "rb") as f:
        stored_data = pickle.load(f)
        indices = stored_data["indices"]
        category_count = stored_data["category_count"]

    return to_loader(indices, category_count, batch_size)
```

Funkcija `to_loader` kreće generiranjem oznaka za učitane podatke. Tokom obrade smo prebrojali koliko podataka (datoteka) ima u pojedinoj kategoriji i taj broj spremili. Zbog načina na koji smo povezivali podatke, imamo garanciju da su oni poredani - prvi su podaci iz kategorije `0`, drugi iz kategorije `1` itd. Upravo zbog toga nam je dovoljno znati samo *koliko* podataka u kojoj kategoriji ima - oznake možemo generirati tako da ponovimo oznaku onoliko puta koliko je datoteka bilo u pripadnom direktoriju i na kraju ih samo redom pospajamo:

```
nested_categories = [
    [category for _ in range(count)]
    for (category, count) in enumerate(category_count)
]
# flatten the nested categories
return list(chain.from_iterable(nested_categories))
```

Sve što nam preostaje nakon toga je podatke i oznake pretvoriti u `tensore` ([6]), od njih napraviti `Dataset` ([6]) i u konačnici od toga `Dataset`-a napraviti `DataLoader`.

Bitan dodatak u stvaranju `DataLoader`-a je funkcija `pad_collate` - budući da `DataLoader` očekuje da svi podaci unutar jedne *hrpe* budu jednake duljine (što kod nas općenito nije slučaj jer su isječci koda različitih duljina), ona će odrediti koji je podatak u *hrpi* najdulji te će ostale *napuniti* (engl. *pad*) nulama tako da budu jednake duljine. Pored toga vraća i originalne duljine svakog podatka, kako bi ih kasnije mogli vratiti u originalno stanje.

```
def pad_collate(batch):
    (algorithms, categories) = zip(*batch)

    # get original lengths for packing
    algorithm_lengths = [len(algorithm) for algorithm in algorithms]

    # pad sequences to longest in batch
    padded_algorithms = pad_sequence(
        algorithms, batch_first=True, padding_value=0
    )

    return padded_algorithms, categories, algorithm_lengths
```

3.2 Arhitektura

Nakon što smo proučili kako se priređuju i koriste podaci, pogledajmo kako izgleda rad sa samim modelom. U ovom poglavlju proučit ćemo *arhitekturu* modela - koje komponente sadrži i na koji su način povezane. Arhitekturu ćemo prvo proučiti na konceptualnoj razini, a potom ćemo pokazati kako je ona implementirana u našem programu.
Nakon što proučimo arhitekturu modela, pogledat ćemo kako ga naučiti, testirati te spremiti za kasniju upotrebu.

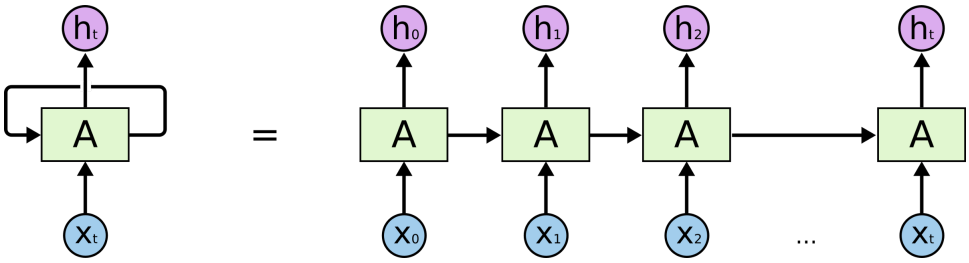
3.2.1 Konceptualni pregled

Analizu arhitekture našeg modela započet ćemo LSTM-om i *attention* mehanizmom - ključnim komponentama koje čine središnji dio našeg modela. Zatim ćemo reći nešto o *dropoutu*, *embeddinzima* i *linearnom sloju*, pomoćnim komponentama koje transformiraju ulaz i izlaz modela tako da čini smislenu cjelinu, te ćemo za kraj pogledati kako su sve te komponente međusobno povezane.

LSTM
Jezgru našeg modela čini LSTM (skraćeno od *long short-term memory*) - vrsta *rekurentne* neuronske mreže. Rekurentne neuronske mreže (RNN) podatak primaju kao niz *vremenskih koraka* (engl. *time steps*), što ih čini sjajnim kandidatom za naš problem - nama će jedan korak činiti jedna LLVM IR instrukcija. Način na koji rekurentne mreže rade je da koriste *skrivena stanja* (engl. *hidden states*). Skriveno stanje sadrži prikupljene informacije o dosadašnjim elementima ulaznog niza te se, uz sljedeći korak ulaza, koristi za stvaranje novog skrivenog stanja:

$$h_t = rnn(x_t,\ h_{t-1})$$

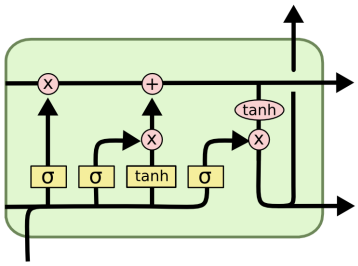
gdje je h_t skriveno stanje u koraku t, x_t ulaz u koraku t te h_{t-1} skriveno stanje u koraku $t-1$. Kao početno skriveno stanje (h_0) često se koristi nulvektor - tu ćemo praksu pratiti i mi.



Slika 1: Lijevo: *skupljena* RNN. Desno: *razmotana* (engl. *unfolded*) RNN[11]

[11] Izvor: https://towardsdatascience.com/introduction-to-recurrent-neural-network-27202c3945f3

Obične RNN pate od problema znanih kao *nestajući*, odnosno *eksplodirajući* gradijent (engl. *vanishing and exploding gradient*) - gradijenti u procesu minimizacije funkcije troška počnu težiti u 0 ili u ∞. Način na koji to tumačimo je da RNN najbolje "pamti" one korake koji su se dogodili nedavno, dok one koji su se dogodili nešto ranije zaboravlja, što je veliki problem kod podataka s velikim brojem koraka kod kojih može postojati povezanost između koraka s velikim vremenskim razmakom.
LSTM je osmišljen upravo kako bi spriječio taj problem. To postiže korištenjem *stanja ćelije*[12] - dodatnog skupa parametara čiji je cilj "dugoročno pamćenje". Stanje ćelije kontroliraju troja *vrata* (engl. *gates*): ulazna, izlazna i *zaboravna* (engl. *forget*).



Slika 2: Unutrašnja struktura jedne LSTM *ćelije*.[13]

Za svaki korak u ulaznom nizu, stanja se računaju na sljedeći način:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

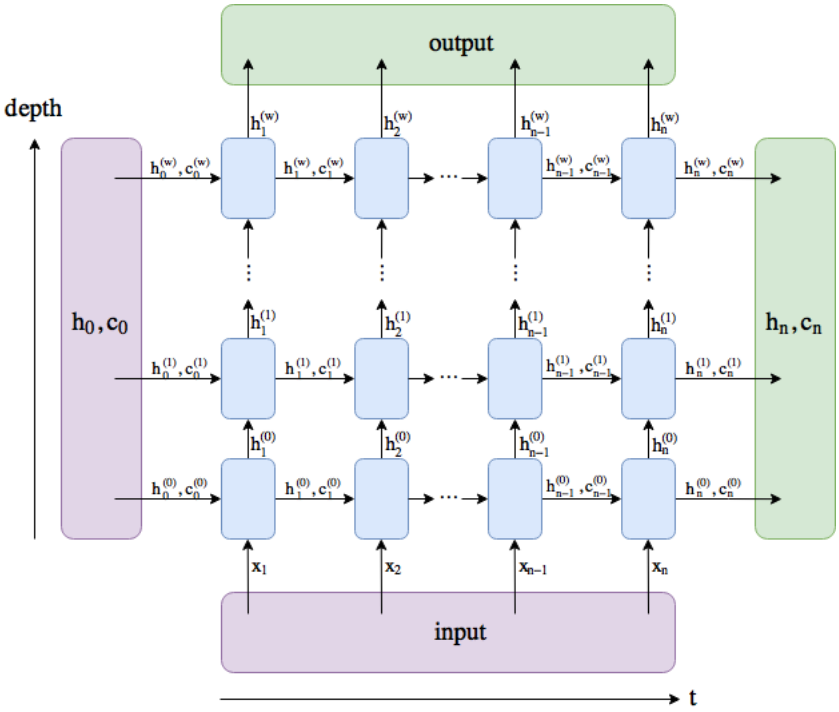
gdje je h_t skriveno stanje u koraku t, c_t stanje ćelije u koraku t, x_t ulaz u koraku t, h_{t-1} skriveno stanje u koraku $t-1$ (ili početno stanje, ako smo u nultom koraku), i_t, f_t, g_t, o_t su ulazna, zaboravna, ćelijska i izlazna *vrata*, σ je `sigmoid`[14] funkcija, a $\odot$ Hadamardov produkt (produkt "po elementima").

Osim osnovnog oblika LSTM-a, koriste se i neke njegove složenije verzije[15].

[12] Ćelijom nazivamo skup operacija kojim rekurentna mreža kombinira korak ulaza te prijašnje skriveno stanje kako bi stvorila novo skriveno stanje.
[13] Izvor: http://colah.github.io/images/post-covers/lstm.png
[14] https://en.wikipedia.org/wiki/Sigmoid_function
[15] Ekvivalentne verzije prisutne su i kod drugih vrsta rekurentnih mreža.

Višeslojni LSTM

Često je praksa umjesto korištenja samo jednog LSTM-a *naslagati* (engl. *stack*) njih više uzastopno - u tom slučaju svaki od njih nazivamo *slojem*, odakle dolazi naziv *višeslojni* LSTM (engl. *multi-layer LSTM*).



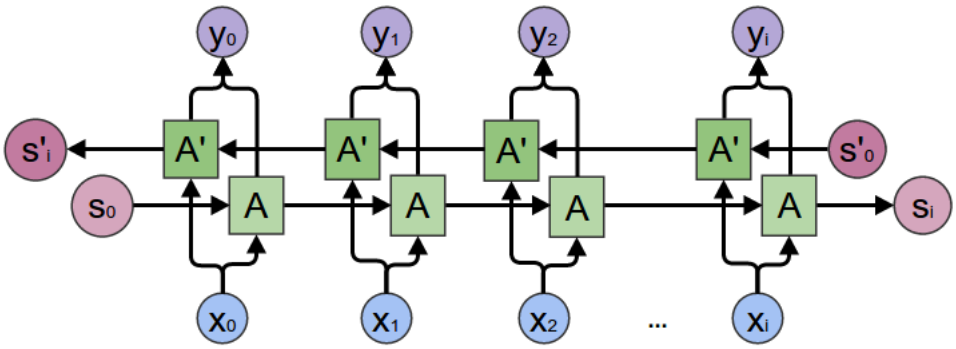
Slika 3: Struktura višeslojnog LSTM-a.[16]

Višeslojni LSTM radi tako što prvi sloj prima ulazni niz i obrađuje ga kao što bi i inače. Tokom obrade ulaza, prvi sloj LSTM-a stvara svoje *izlaze* (skrivena stanja na svakom koraku ulaznog niza). Kod jednoslojnog LSTM-a bismo ovdje stali, uzeli zadnje skriveno stanje (stanje nakon zadnjeg koraka ulaznog niza) te ga dalje koristili u svom modelu. No, kod višeslojnog LSTM-a postupak je sljedeći: izlaze koje je generirao prvi sloj sada tretiramo kao novi ulazni niz te ga dajemo drugom sloju na obradu. Drugi sloj tokom obrade novog niza generira svoje izlaze, koje opet dajemo trećem sloju kao ulaz. Taj postupak ponavljamo sve dok ne dođemo do zadnjeg sloja - tek tada uzimamo zadnje skriveno stanje i njega dalje koristimo u modelu.

[16]Izvor: https://i.stack.imgur.com/SjnTl.png

Dvosmjerni LSTM

Budući da LSTM podatke prima linearno (korak po korak, od početka do kraja), u svakom trenutku "zna" samo ono što je dosad "vidio", odnosno korake koji su došli *prije* trenutnog. Nekad je to dovoljno, no nekad je za bolje razumijevanje potrebno i ono što dolazi *nakon* trenutnog koraka. Iz tog se razloga ponekad koristi *dvosmjerni* LSTM (engl. *bidirectional LSTM*).



Slika 4: Prikaz rada dvosmjernog LSTM-a.[17]

Dvosmjerni LSTM možemo promatrati kao dva zasebna LSTM-a od kojih jedan podatak prima normalno (kao što bi običan LSTM), dok drugi prima isti taj podatak, samo u obratnom poretku (od zadnjeg koraka prema prvom). Zadnje skriveno stanje oba LSTM-a se najčešće konkatenira te tretira kao skriveno stanje čitavog (dvosmjernog) LSTM-a. Time se postiže efekt da LSTM "vidi" čitavo okruženje - ono što se nalazi i prije, a i poslije trenutnog koraka.

`attention`

LSTM je značajan napredak od obične rekurentne mreže, ali svejedno s njom dijeli veliko ograničenje - svu informaciju o ulaznom nizu mora pohraniti u jedno (zadnje) skriveno stanje. Kako bismo izašli na kraj s tim ograničenjem, predložen je `attention` mehanizam.
Ideja `attention` mehanizma je da radi sa svim skrivenim stanjima LSTM-a i svakom pridruži određenu *pažnju*, odnosno odredi koje stanje (pa onda i pripadna instrukcija u nizu) koliko utječe na donošenje odluke. S vremenom su se razvile razne verzije `attention` mehanizma - mi koristimo modificiranu[18] verziju Loungova mehanizma (vidi [5]) s *generalnim* načinom računanja *alignment scorea*[19].

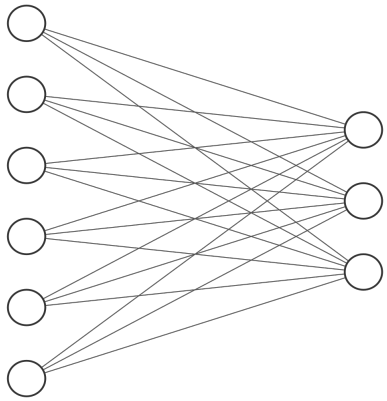
Linearni sloj

Linearni (također znan kao *potpuno povezani* (engl. *fully connected*) ili *gusti* (engl. *dense*)) sloj ulazni vektor dimenzije l mapira u izlazni vektor dimenzije k na način da svaki element izlaznog vektora izračuna kao linearnu kombinaciju ulaznog vektora i pripadnog vektora težina.

[17] Izvor: http://colah.github.io/posts/2015-09-NN-Types-FP/img/RNN-bidirectional.png

[18] Verzija predložena u radu namijenjena je modelima koji niz pretvaraju u niz (*seq2seq*), stoga su bile potrebne izmjene kako bi mehanizam radio na klasifikacijskom modelu.

[19] Uz pomoć *alignment scorea* određuje koliko će *pažnje* dati kojem koraku (instrukciji).



Slika 5: Primjer linearnog sloja koji mapira ulaz dimenzije 6 u izlaz dimenzije 3.

Ako s $\mathbf{u} = (u_1, u_2, \ldots, u_l) \in \mathbb{R}^l$ označimo ulazni vektor, s $\mathbf{v} = (v_1, v_2, \ldots, v_k) \in \mathbb{R}^k$ izlazni vektor i s $\mathbf{w_i} = (w_{i1}, w_{i2}, \ldots, w_{il}) \in \mathbb{R}^l$ vektor težina pridružen i-tom elementu izlaznog vektora v, element v_i računamo na sljedeći način:

$$v_i = \Sigma_{j=1}^{l} w_{ij} u_j$$

za $i = 1, 2, \ldots, k$.

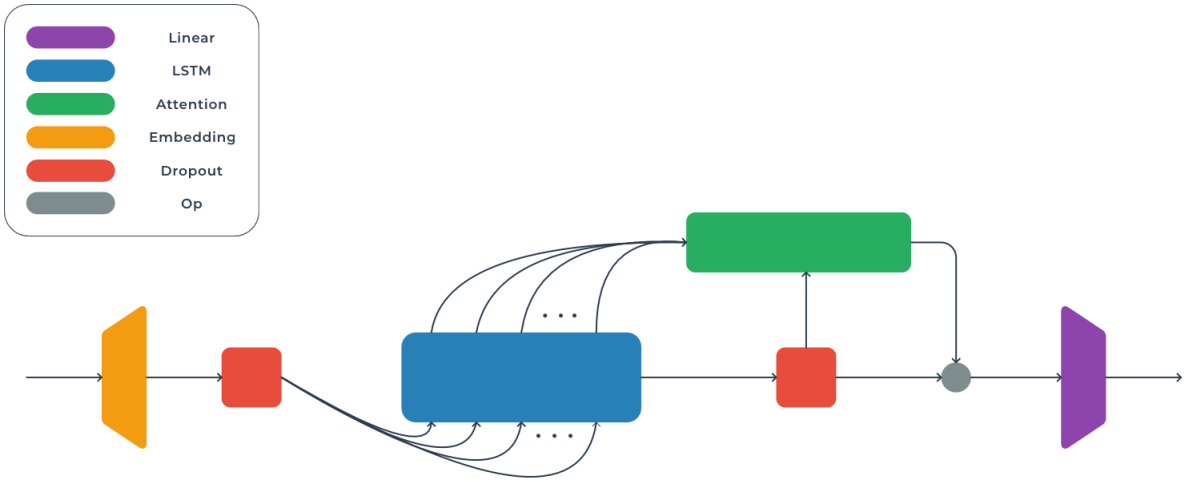
Ukoliko na izlazni vektor primjenimo neku nelinearnu funkciju (npr. `ReLU`[20]) te dobiveni rezultat proslijedimo nekom drugom linearnom sloju kao ulaz, možemo izgraditi samostalnu neuronsku mrežu znanu kao *feedforward*[21] neuronska mreža.
Za naše je potrebe dovoljan samo jedan linearan sloj.

Ugrađivanje
Komponenta ugrađivanja (engl. *embedding*) ulazne podatke ugrađuje u d-dimenzionalni vektorski prostor. Drugim riječima, naše LLVM IR instrukcije opisuje pomoću d svojstava te svakoj instrukciji iz našeg riječnika pridružuje d vrijednosti koje označavaju izraženost pojedinog svojstva. Koristeći ta svojstva možemo odrediti sličnost, odnosno povezanost dvaju instrukcija.
Iako je ručni odabir svojstava i određivanje pripadnih vrijednosti za svaku instrukciju moguće, bilo bi vrlo zahtjevno i neefikasno, stoga to ostavljamo modelu kao dio procesa učenja, odnosno dodatan set parametara.
U praksi se nerijetko koriste komponente ugrađivanja s unaprijed naučenim parametrima koje su već prije procesa učenja modela bile izložene iznimno velikom broj podataka, no u ovom projektu koristimo parametre koji su nasumično odabrani iz $\mathcal{N}(0, 1)$ (standardne normalne distribucije).

`dropout`
Posljednja komponenta koju koristimo je `dropout` - ona s vjerojatnošću p neke od elemenata ulaznog vektora postavi na 0. To se pokazalo kao efektivna tehnika *regularizacije* (vidi [2]) (tj. sprječavanja prenaučenosti modela).

[20] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
[21] https://en.wikipedia.org/wiki/Feedforward_neural_network

Model



Slika 6: Konačna arhitektura modela

Model započinjemo ugrađivanjem ulaznog podatka u višedimenzionalni vektorski prostor - pridruživanjem svojstava kojima opisujemo instrukcije. Neke od elemenata dobivenih vektora postavimo na 0 s ciljem regularizacije.
Transformirani ulaz dajemo LSTM-u, koji za svaku instrukciju u nizu vraća odgovarajuće skriveno stanje. U slučaju dvosmjernog LSTM-a, skrivena stanja koja je vratio prolaz u jednom smjeru konkateniramo sa skrivenim stanjima dobivenim prolaskom u drugom smjeru, čime se broj skrivenih stanja udvostručuje. Ukoliko koristimo višeslojni LSTM, stanja koja je vratio prvi sloj koristimo kao novi niz kojeg dajemo drugom sloju, izlaz drugog sloja kao ulaz trećem itd. Kao konačan rezultat LSTM-a koristimo skrivena stanja iz zadnjeg sloja. Neke elemente konačnog skrivenog stanja postavimo na 0 s ciljem regularizacije.
Sva skrivena stanja (*izlaz* LSTM-a) kao i skriveno stanje vraćeno nakon zadnje instrukcije u nizu (*zadnje* skriveno stanje) prosljeđujemo `attention` mehanizmu - on nam vraća novi vektor parametara, dimenzije jednake onoj zadnjeg skrivenog stanja LSTM-a. Ovisno o zadanim hiperparametrima, parametre `attention` mehanizma povezujemo sa zadnjim skrivenim stanjem LSTM-a tako da ih

- zbrojimo (čime dimenzija ostaje ista) ili
- konkateniramo (čime se dimenzija udvostručuje).

Za kraj, dobiveni vektor parametara koristeći linearni sloj mapiramo u 104 parametra koje tumačimo kao odabir kategorije.

Optimalni hiperparametri
Najbolji rezultati dobiveni su korištenjem sljedećih hiperparametara:

- instrukcije su ugrađivane u 100-dimenzionalni vektorski prostor
- korišten je 2-slojni, jednosmjerni LSTM s 200 skrivenih stanja
- `attention` parametri konkatenirani su na skrivena stanja dobivena iz LSTM-a
- `dropout` je korišten s vjerojatnošću $p = 0.2$

Učenjem modela inicijaliziranog s navedenim hiperparametrima 7 epoha dobili smo model čija je točnost bila 94.67%

3.2.2 Implementacija

Sad kad znamo kako izgleda arhitektura našeg modela, pogledajmo kako je ona implementirana. Implementaciju pronalazimo u klasi `AlgorithmClassifier` koja nasljeđuje klasu `torch.nn.Module` - baznu klasu za sve *PyTorch* ([6]) modele. Kako bi imali funkcionalan model, nužno je preopteretiti metode `__init__` i `forward`.

`__init__`
Kako joj i sam naziv govori, `__init__` metoda služi inicijalizaciji objekta, odnosno u našem slučaju modela. Izuzev parametara `vocab_size` (kojeg čitamo iz modula `data_handler`) i `categories` (kojih u našem skupu podataka ima 104), svim hiperparametrima moguće je upravljati kroz argumente naredbenog retka te na taj način isprobavati razne konfiguracije modela bez direktnog mijenjanja programskog koda.

Prvi dio metode čine provjere i postavljanje potrebnih varijabli:

- osiguravamo da je `attention` način jedan od definiranih
- `dropout` koristimo samo ako smo se odlučili za višeslojni LSTM
- određujemo kolika će biti dimenzija konačnog skrivenog stanja (imamo li dvosmjerni LSTM te je li odabran `cat attention` način)

```
assert attention in ["sum", "cat"]

if layers == 1:
    dropout = 0

# int(b: bool) --> map {False, True} to {0, 1}
directions = 1 + int(bidirectional)
attention_multiplier = 1 + int(attention == "cat")
```

U ovom se dijelu također spremaju prosljeđeni hiperparametri. Informacija o hiperparametrima nam je bitna kako bismo mogli uspješno učitati spremljene modele. Naime, kao što ćemo kasnije vidjeti, model ćemo spremiti tako što ćemo spremiti njegove parametre i hiperparametre, a učitati tako da napravimo novi model i njegove parametre postavimo na one koje smo spremili. Kada ne bi imali informaciju o tome koji su hiperparametri korišteni pri stvaranju modela, morali bismo ih nagađati, čime bismo riskirali inkompatibilnost modela i grešku pri učitavanju parametara.

```
self._hyperparameters = {
    "vocab_size": vocab_size,
    "categories": categories,
    "embedding_size": embedding_size,
    "hidden_size": hidden_size,
    "layers": layers,
    "bidirectional": bidirectional,
    "dropout": dropout,
    "attention": attention
}
```

Ostatak metode čini inicijalizacija komponenti koristeći prosljeđene hiperparametre. `attention` komponenta prati postojeću implementaciju[22], dok ostale komponente koriste implementacije iz *PyTorchova* ([6]) `nn` modula.

```
self.attention = Attention(directions * hidden_size)
self.dropout   = nn.Dropout(p=dropout)
self.embedding = nn.Embedding(vocab_size, embedding_size)
self.decode    = nn.Linear(
    attention_multiplier * directions * hidden_size, categories
)
self.recurrent = nn.LSTM(
    embedding_size, hidden_size, num_layers=layers, batch_first=True,
    dropout=dropout, bidirectional=bidirectional
)
```

[22] Originalnu implementaciju `attention` mehanizma možete pronaći ovdje:
https://github.com/chrisvdweth/ml-toolkit/blob/master/pytorch/models/text/classifier/rnn.py

forward
Metoda `forward` odgovara prolasku podatka kroz model. U njoj definiramo kako su komponente koje smo definirali u `__init__` metodi međusobno povezane.

Krećemo *ugrađivanjem* podataka u višedimenzionalni prostor i poništavanjem nekih od elemenata s ciljem regularizacije (tj. sprječavanja *prenaučenosti* modela).

```
embedded = self.embedding(input)
embedded = self.dropout(embedded)
```

Ugrađene podatke *pakiramo* (uklanjamo *punjenje* koje je zahtjevao `DataLoader`), prosljeđujemo LSTM-u te dobivene izlaze *otpakiravamo* kako bismo ih mogli dalje koristiti.

```
packed = pack_padded_sequence(
    embedded, input_lengths, batch_first=True, enforce_sorted=False
)
output, (state, _) = self.recurrent(packed)
output, _ = pad_packed_sequence(
    output, batch_first=True, padding_value=0
)
```

Od dobivenih skrivenih stanja LSTM-a uzimamo ona iz zadnjeg sloja i ponovno poništavamo elemente.

```
if self._hyperparameters["bidirectional"]:
    state = torch.cat((state[-1], state[-2]), dim=1)
else:
    state = state[-1]

state = self.dropout(state)
```

Određujemo `attention` te ga pridružujemo dosadašnjem skrivenom stanju na zadan način.

```
attention_output, _ = self.attention(output, state)

if self._hyperparameters["attention"] == "cat":
    state = torch.cat((state, attention_output), dim=1)
else:
    state = state + attention_output
```

Za kraj, linearnim slojem mapiramo konačno skriveno stanje u 104 vrijednosti koje predstavljaju odabir kategorije.

```
decoded = self.decode(state)
return decoded
```

3.3 Učenje

Nakon inicijalnog stvaranja modela, njegovi parametri (koje koristi za predviđanje kategorije) postavljeni su na nasumične vrijednosti, zbog čega je najbolji rezultat kojem se možemo nadati to da će kategorije pogađati uniformno (kao da odluku donosi bacanjem kockice sa 104 stranice), što znači da vjerojatnost pogotka točne kategorije nije niti 1%. Kako bismo postigli veću točnost, potrebno je *naučiti* model - dati mu podatke i pokazati točne kategorije kako bi mogao prilagoditi parametre zadanom zadatku.

Funkcija troška
Kao funkciju troška (engl. *loss function*) koristimo `CrossEntropyLoss`, funkciju iz *PyTorch* ([6]) biblioteke koja kombinira `softmax` funkciju (nešto više o njoj reći ćemo u dijelu vezanom za testiranje modela) te *funkciju maksimalne vjerodostojnosti* (engl. *maximum likelihood function*)[23], odnosno njenu negativnu, logaritamsku verziju. Minimiziranjem te funkcije želimo postići da za svaki podatak model točnoj kategoriji (kategoriji kojoj pripada dani podatak) pridruži najveću vjerojatnost (tj. da model *odabere* tu kategoriju).

Optimizator
Za optimizator odabrali smo Adam (vidi [4]), odnosno njegovu *PyTorch* ([6]) implementaciju sa zadanim hiperparametrima, koji se u praksi pokazao kao dobar odabir. Njegov zadatak je da ažurira parametere modela tako da funkcija troška što prije dostigne lokalni minimum.

Implementacija
Proces učenja unutar `train` funkcije provodimo na sljedeći način: prolazimo kroz podatke dane u obliku `DataLoader`-a, te podatke proslijedimo modelu kako bi napravio predikciju kategorije (implicitno pozivamo `forward` metodu). Rezultat modela i originalnu oznaku dajemo funkciji troška kako bi odredila koliku je pogrešku naš model napravio. Računamo gradijent funkcije troška, kojeg potom optimizator koristi kako bi ažurirao parametre modela.

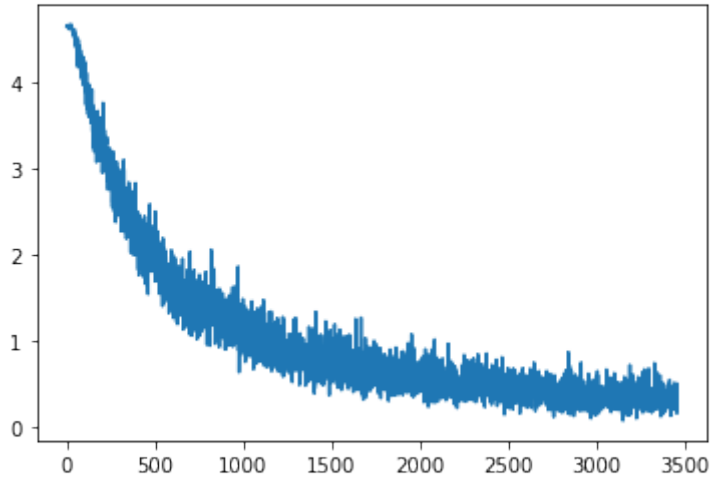
```
for (algorithms, categories, algorithm_sizes) in train_loader:
    predictions = model(algorithms, algorithm_sizes)
    loss = loss_fn(predictions, categories)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Tokom procesa učenja važno je pratiti stanje modela - koliko je dobro *naučio* kategorizirati podatke. To možemo učiniti tako da nakon svake *epohe* (prolaska kroz cijeli skup podataka) ispišemo kolika je bila prosječna vrijednost funkcije troška u toj epohi

```
average_loss = sum(loss_values)/len(loss_values)
print(f"\nEpoch #{epoch + 1}: {average_loss}")
```

te da vrijednosti funkcije troška kroz epohu grafički prikažemo koristeći biblioteku *Matplotlib* ([3]).

```
plt.plot(range(1, len(loss_values)+1), loss_values)
```



Slika 7: Graf kretanja funkcije troška kroz prvu epohu učenja modela.

Predugo učenje modela može dovesti do problema znanog kao *prenaučenost* (engl. *overfitting*) - model umjesto učenja općenitih svojstava problema počinje učiti specifičnosti skupa podataka na kojem uči, što kao posljedicu ima da mu se točnost na općenitim, neviđenim podacima počne smanjivati.
Kako bismo spriječili prenaučenost, nakon svake epohe učenja model kojeg trenutno učimo pomoću funkcije `compare_to_saved` usporedimo s modelom spremljenim pod istim nazivom (`--save-name` argument naredbenog retka), bilo da se radi o starijoj verziji istog modela (iz prijašnje epohe) ili o nekom sasvim drugom modelu.

```
current_model_accuracy = test(model, loader, device, output=False)

saved_model = AlgorithmClassifier.load(name)
saved_model_accuracy = test(saved_model, loader, device, output=False)
```

Tek kada novi model nadmaši stari po pitanju točnosti na skupu podataka za validaciju se novi model prihvaća i sprema.

```
if current_model_accuracy > saved_model_accuracy:
    AlgorithmClassifier.save(model, name)
```

Postupak učenja modela može trajati vrlo dugo, stoga ćemo ga nekad htjeti ranije zaustaviti. To možemo napraviti tako da u okruženju u kojem smo pokrenuli program prekinemo izvršavanje (npr. kombinacijom tipki `Ctrl + C`) - program će stati s učenjem, o tome obavijestiti korisnika porukom te nastaviti s daljnim izvršavanjem programa (npr. testiranjem dobivenog modela). U suprotnom, ranije navedeni proces će se ponoviti zadani broj epoha (`--epochs` argument naredbenog retka).

```
try:
    # . . .

# enable early exit
except KeyboardInterrupt:
    print(f"Stopped training (during epoch {epoch + 1} of {epochs}).")
```

[23] https://towardsdatascience.com/probability-concepts-explained-maximum-likelihood-estimation-c7b4342fdbb1

3.4 Testiranje

Cilj je testiranja modela odrediti koliko točno za dani podatak može odrediti kojoj kategoriji taj podatak pripada. Testirati možemo na skupu podataka kojeg je model već vidio (skup za učenje), ali najčešće testiramo na podacima koje model dosad nije vidio - na validacijskom skupu (tokom učenja, kako bismo se uvjerili da model i dalje uči tj. raste točnost kojom određuje kategorije) te na testnom skupu (kako bismo utvrdili točnost konačnog modela).

Sam proces testiranja je relativno jednostavan - evaluiramo dobiveni model na podacima koje smo dobili u obliku `DataLoader`-a, te rezultate i originalne oznake stavljmo u odgovarajuće liste sve dok nismo prošli kroz cijeli skup podataka.

```
all_predictions = torch.tensor([])
all_categories = []

for (algorithms, categories, algorithm_sizes) in test_loader:
    all_categories.extend(categories)

    algorithms = algorithms.to(device)
    predictions = model(algorithms, algorithm_sizes)
    all_predictions = torch.cat((
        all_predictions, predictions.cpu().detach()
    ))
```

Budući da je svaka kategorija `tensor`, moramo izvući vrijednost kako bismo s njom mogli raditi,

```
def _extract_categories(categories):
    return [category.item() for category in categories]
```

dok na rezultate trebamo primjeniti funkciju `softmax`. Funkcija `softmax` definirana je na sljedeći način:

$$\sigma : \mathbb{R}^k \to [0,1]^k, \quad \sigma(\mathbf{z})_i = \frac{e^{\mathbf{z}_i}}{\sum_{j=1}^{k} e^{\mathbf{z}_j}}$$

gdje je $i = 1, 2, \ldots, k$ i $\mathbf{z} = (z_1, z_2, \ldots, z_k) \in \mathbb{R}^k$.

Njena važnost je u tome što normalizira vrijednosti u vektoru tj. sve vrijednosti mapira u raspon [0, 1] te u sumi daju 1, zbog čega ih možemo promatrati kao vjerojatnosti da dani podatak pripada nekoj kategoriji.

Nakon što smo na rezultate modela primjenili `softmax` funkciju, kategoriju s najvećom pripadnom vjerojatnošću odabiremo kao predikciju te tu kategoriju i pripadnu vjerojatnost vraćamo.

```
def _extract_predictions(outputs):
    outputs = softmax(outputs, dim=1)

    predictions = outputs.max(dim=1)
    predicted_categories = predictions.indices
    prediction_confidence = predictions.values

    return predicted_categories, prediction_confidence
```

Sada kada imamo oznake i predikcije, možemo izračunati *točnost* (engl. *accuracy*) našeg modela koristeći funkciju `accuracy_score` iz biblioteke *scikit-learn* ([7]).

```
predictions, confidence = _extract_predictions(all_predictions)
categories = _extract_categories(all_categories)
accuracy = accuracy_score(categories, predictions)
```

Ukoliko smo pokrenuli testiranje s uključenim ispisom (argument `output`), nakon testiranja ispisat će nam se dobivena *točnost* kao i prosječna vjerojatnost odabrane kategorije, formatirane u obliku postotka s dvije decimale točnosti. Funkcija kao rezultat vraća izračunatu *točnost*.

```
if output:
    print(f"Accuracy: {_format_percentage(accuracy)}")

    average_confidence = (sum(confidence) / len(confidence)).item()
    print(f"Average confidence: {_format_percentage(average_confidence)}")

return accuracy
```

3.5 Spremanje i učitavanje

Ponovno učenje modela svaki put kada nam je on potreban je neisplativo - morali bismo čekati barem nekoliko desetaka minuta (ovisno o hardverskim mogućnostima), a i krajnja točnosti modela varirala bi zbog nasumičnih početnih stanja parametara u modelu. Iz tog razloga naš naučeni model želimo moći spremiti te ga kasnije ponovno učitati i koristiti.

Spremanje
Za spremanje modela zadužena je statička metoda `save` klase `AlgorithmClassifier`. Ona kao argumente prima model koji treba spremiti te naziv pod kojim će ga spremiti (`--save-name` argument naredbenog retka) te ga sprema pod tim nazivom u direktorij `models`.
Metoda započinje na način da provjeri postoji li direktorij `models` (te ga napravi ukoliko ne postoji),

```
if not os.path.exists(f".{base_path}/models"):
    os.mkdir(f".{base_path}/models")
```

zatim napravi *duboku kopiju* (engl. *deep copy*) parametara modela

```
model_state = deepcopy(model.state_dict())
```

te u konačnici spremi parametre modela te hiperparametre koji su prosljeđeni pri stvaranju modela.

```
torch.save(
    {
        "state": model_state,
        "hyperparameters": model._hyperparameters
    },
    f".{base_path}/models/{name}.pt"
)
```

Učitavanje
Ukoliko želimo učitati (npr. u svrhe testiranja) postojeći spremljeni model, učinit ćemo to tako da statičkoj metodi `load` klase `AlgorithmClassifier` proslijedimo naziv pod kojim je model spremljen.
Metoda `load` prvo će provjeriti da taj model uistinu postoji te ga učitati,

```
assert os.path.exists(f".{base_path}/models/{name}.pt")
model_data = torch.load(f".{base_path}/models/{name}.pt")
```

potom će iskoristiti spremljene hiperparametre kako bi stvorila novi model i kao njegovo *stanje* (parametre koje učimo) postaviti spremljene parametre

```
model = AlgorithmClassifier(**model_data["hyperparameters"])
model.load_state_dict(model_data["state"])
```

i u konačnici model postaviti na evaluacijski način rada (preporučeni način rada za testiranje modela) i vratiti model.

```
model.eval()
return model
```

4 Dodaci

`measure_time` dekorator
Služi praćenju vremena izvršavanja funkcije na način da prije početka i nakon završetka izvođenja funkcije spremi trenutno vrijeme, oduzme krajnje od početnog te ispiše ime funkcije zajedno s izmjerenim vremenom u formatu `mm:ss`.
Budući da je primarna namjena ovog dekoratora mjerenje vremena potrebnog za učenje i testiranje modela (koje je najčešće u minutama), vrijeme se mjeri s malom preciznošću pa se njegovo korištenje ne preporučuje u svrhe *benchmarkinga*.

Implementacija

```python
import time

def fix_zeros(x):
    return f"{'0' if len(str(x)) == 1 else ''}{x}"

def display_time(t):
    t = int(t)
    return f"{fix_zeros(t // 60)}:{fix_zeros(t % 60)}"

def measure_time(func):
    def inner(*args, **kwargs):
        start = time.time()

        return_value = func(*args, **kwargs)

        end = time.time()
        print(f"Finished `{func.__name__}` in {display_time(end - start)}.")

        return return_value

    return inner
```

Primjer
Ukoliko pokrenemo funkciju `func` na koju je primijenjen dekorator

```python
@measure_time
def func():
        # . . .
```

i njeno izvršavanje traje 7 minuta i 39 sekundi, nakon njenog izvršavanja na zaslonu ćemo dobiti sljedeći ispis:

```
Finished `func` in 07:39.
```

`base_path`

`base_path` je konstanta koja je stavljena ispred svake putanje korištene u programu. Služi kako bi se program jednostavno (bez dupliciranja) prilagodio slučaju kada ga ne pokrećemo s istog mjesta na kojem je pohranjen.
Primjer takvog okruženja je Google Colab[24], gdje se Python bilježnica unutar koje pokrećemo program ne nalazi gdje i spremljen kod, stoga je potrebno `base_path` prilagoditi da upućuje do lokacije pohrane (npr. `"/drive/MyDrive/algorithm-classification"`).
Zadana vrijednost ove konstante je `""` (prazan string) i za lokalno korištenje ju nije potrebno mijenjati.

[24]https://colab.research.google.com/notebooks/intro.ipynb

Literatura

[1] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 3588–3600. Curran Associates, Inc., 2018.

[2] Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors, 2012.

[3] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[4] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[5] Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation, 2015.

[6] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[7] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.