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

Petar Poljarević

Tehnike dubinskog učenja za klasifikaciju peludi

Završni praktični projekt - dokumentacija

Osijek, 2021.

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

Petar Poljarević

Tehnike dubinskog učenja za klasifikaciju peludi

Završni praktični projekt - dokumentacija

Mentor: doc. dr. sc. Slobodan Jelić

Osijek, 2021.

Sadržaj

1 Uvod

1.1 O problemu

Ovaj projekt bavi se klasifikacijom peludnih čestica na osnovu scattering signala i signala fluorescencije izmjerenih koristeći RapidE uređaj. Budući da je ovo još relativno novo i neistraženo područje strojnog učenja, može se reći kako je zapravo cilj projekta bio provjeriti mogu li se peludne čestice uopće klasificirati na osnovu spomenutih signala.

Primjenom raznih modela konvolucijskih neuronskih mreža uočeno je da se neke tipove peludi može klasificirati s točnošću od čak 80-90%, dok se neke tipove može klasificirati s točnošću manjom od 40%. Naravno, to ne znači da ih se ne može klasificirati bolje, nego možda treba pronaći bolju arhitekturu i/ili hiperparametre, a možda jednostavno trebamo više reprezentanata tih klasa.

1.2 Tehnologije

Projekt je napisan u programskom jeziku Python, uz korištenje sljedećih biblioteka:

- Pandas[1] - za učitavanje .csv datoteka
- PyTorch[2] - za rad s tenzorima i neuronskim mrežama
- Matplotlib[3] - za vizualizaciju točnosti i funkcije cilja po epohama treninga
- time[4] - za mjerenje brzine treniranja modela
- sklearn[5] - za skaliranje ulaznih podataka
- os[6] - za kreiranje datoteka i mapa (ukoliko se koristi *autosave* opcija)
- json[7] - za spremanje informacija o treningu modela (ukoliko se koristi *autosave* opcija)
- random[8] - za generiranje uniformne distribucije prilikom traženja optimalnih hiperparametara

[1] `https://pandas.pydata.org/`
[2] `https://pytorch.org/`
[3] `https://matplotlib.org/`
[4] `https://docs.python.org/3/library/time.html`
[5] `https://scikit-learn.org/stable/`
[6] `https://docs.python.org/3/library/os.html`
[7] `https://docs.python.org/3/library/json.html`
[8] `https://docs.python.org/3/library/random.html`

1.3 Organizacija koda

Kod se nalazi u Jupyter bilježnici `Pollen Classifier` na GitHub repozitoriju projekta[9] i podijeljen je na poglavlja čiji nazivi ukratko opisuju sadržaj tog poglavlja ili potpoglavlja. Većina funkcija i metoda je također ukratko opisana koristeći komentare.

Projekt je zamišljen tako da se ćelije koda pokreću onim redom kojim su poslagane. Vrijedi napomenuti kako je većina funkcija i klasa neovisna o preostalima, što znači da se projekt može vrlo lako i uz minimalne modifikacije iskoristiti i ako npr. imamo drugačiji skup podataka ili želimo definirati neku svoju arhitekturu neuronske mreže.

[9] `https://github.com/ppoljare/Pollen-classifier`

2 Učitavanje i obrada podataka

Kao što je već spomenuto u uvodu, podaci koji su korišteni u projektu dolaze s RapidE uređaja za mjerenje signala peludnih čestica. Podijeljeni su na 8 tipova peludi, pri čemu svaki tip ima dva podtipa, ali tretiramo ih kao jednu klasu. Svaki podatak (redak) ima 608 vrijednosti koje označavaju neke određene karakteristike te skupine peludnih čestica.

2.1 Klasa `PollenDataset`

Ova klasa služi za učitavanje podataka i nasljeđuje klasu `torch.utils.data.Dataset` (nadalje u tekstu: `Torch Dataset`). Treba napomenuti kako ova klasa učitava cijeli skup podataka u radnu memoriju, što znači da je važno imati je puno više nego što vam obično treba (ovaj projekt je izrađen na računalu sa 16 GB radne memorije). Detaljnije objašnjenje kako definirati ovakvu klasu može se pronaći na [1].

2.1.1 Metoda `__init__()`

Konstruktor klase. Metoda prima dvije .csv datoteke: jednu koja sadrži skup podataka te jednu koja svakom tipu peludi dodjeljuje cjelobrojnu vrijednost (primjere `data_file` i `labels_file` datoteka može se pronaći na GitHub repozitoriju projekta[10]). Osim spomenutih datoteka, metoda prima opcionalni parametar koji pretvara učitani skup podataka u skup podataka s dvije klase. Nakon što su podaci učitani u radnu memoriju, na njih se primjenjuje `scikit.StandardScaler()` i pretvaraju se u `torch.Tensor` tipa `Float`. Također se učitavaju i oznake za svaku klasu te se pretvaraju u `torch.Tensor` tipa `int`.

Povratni tip: `PollenDataset`

Parametri:

- `data_file (str)` - putanja do .csv datoteke u kojoj se nalaze podaci
- `labels_file (str)` - putanja do .csv datoteke u kojoj se nalaze imena klasa i njihovi pripadni redni brojevi
- `main_class (int)` - redni broj klase koju želimo klasificirati u odnosu na sve ostale. Ukoliko ne proslijedimo ništa, ostat će nam oznake iz `labels_file`. Ukoliko proslijedimo valjan broj, ta klasa će imati oznaku 0, dok će sve ostale klase imati oznaku 1 `(default: None)`

Primjer:

```
dataset = PollenDataset(
    data_file='./data.csv',
    labels_file='./labels.csv'
)
```

[10] `https://github.com/ppoljare/Pollen-classifier`

```
dataset_binary = PollenDataset(
    data_file='./data.csv',
    labels_file='./labels.csv',
    main_class=6
)
```

2.1.2 Metoda `__len__()`

Vraća veličinu (broj podataka) *dataset*-a.

Povratni tip: `int`

Primjer:

```
total_items = len(dataset)
```

2.1.3 Metoda `__getitem__()`

Vraća podatak i njegov *label* koji se nalazi u traženom retku. Obično se koristi pomoću `for` petlje kao u donjem primjeru.

Povratni tip: `(torch.Tensor<Float>, torch.Tensor<int>)`

Parametri:

- `idx (int)` - indeks traženog retka

Primjer:

```
for data in dataset:
    # do something with data...
#end for data
```

2.2 Funkcija `splitData()`

Funkcija prima skup podataka te ih dijeli na skup za treniranje, testiranje i validaciju. Zadani omjer je 70:15:15.

Povratni tip: `tuple <Torch Dataset>`

Parametri:

- `dataset (Dataset)` - skup podataka kojeg želimo podijeliti
- `test_percent (float)` - postotak od ukupnih podataka koji će postati skup za testiranje `(default: 0.15)`
- `valid_percent (float)` - postotak od ukupnih podataka koji će postati skup za validaciju `(default: 0.15)`

Primjer:

```
(train_data, test_data, valid_data) = splitData(
    dataset,
    test_percent = 0.2
    valid_percent = 0.15
)
```

2.3 Funkcija `createDataLoaders()`

Funkcija prima tri skupa podataka te ih učitava u varijable tipa `DataLoader` iz biblioteke `torch.utils.data` (nadalje u tekstu: `DataLoader`). Također moramo proslijediti veličinu *batch*-a kao jedan od parametara. Ovaj parametar značajno utječe na brzinu treniranja mreže jer rad s velikim *batch*-evima ubrzava paralelno računanje pomoću grafičke kartice.

Povratni tip: `(DataLoader, DataLoader, DataLoader)`

Parametri:

- `train_data (Dataset)` - skup podataka za treniranje
- `test_data (Dataset)` - skup podataka za testiranje
- `valid_data (Dataset)` - skup podataka za validaciju
- `batch_size (int)` - veličina grupe u kojoj će se podaci učitavati. Veće vrijednosti ubrzavaju treniranje, ali manje vrijednosti mogu povećati točnost modela
- `shuffle (bool, bool, bool)` - uređena trojka koja nam govori želimo li nasumično razbacati podatke za treniranje, odnosno testiranje i validaciju `(default: (True, False, False))`

Primjer:

```
(trainloader, testloader, validloader) = createDataLoaders(
    train_data,
    test_data,
    valid_data,
    batch_size=32,
    shuffle=(True,False,True)
)
```

2.4 Funkcija `getClassWeights()`

Funkcija prima skup podataka i broj klasa te računa težinu pojedine klase. Veći broj članova neke klase daje manju težinu toj klasi. Kada radimo s neujednačenim skupovima podataka, važno je dati odgovarajuće težine svakoj klasi. Na ovaj način smo se osigurali da mreža neće sve označavati kao pripadnike najbrojnije klase jer se tako dobije velika točnost, ali nam ta točnost ne znači ništa ako takvoj mreži probamo dati podatke s drugačijim rasporedom klasa.

Povratni tip: `torch.Tensor<Float>`

Parametri:

- `dataset (Dataset)` - skup podataka na kojem računamo težine
- `no_of_classes (int)` - broj klasa u skupu podataka
- `power (float)` - potencija na koju želimo podići težinu svakog skupa `(default: 1)`
- `force_cpu (bool)` - želimo li "prisiliti" slanje težina na CPU, čak i ako nam je GPU dostupan? `(default: False)`

Primjer:

```
class_weights = getClassWeights(
    train_data,
    no_of_classes=8,
    power=2
)
```

3 Konvolucijska neuronska mreža

Konvolucijska neuronska mreža (CNN) je vrsta neuronske mreže koja obično služi za klasifikaciju slika ili detekciju objekata na slikama. Sastoji se od:

- konvolucijskih slojeva, koji traže određene informacije na ulaznim podacima pomoću nekog zadanog broja filtera (najčešće je to potencija broja 2)
- *pooling* slojeva, koji služe za sažimanje informacija dobivenih konvolucijskim slojevima u podatke manjih dimenzija
- *fully connected* (FC) slojeva, od kojih su izgrađene standarne neuronske mreže

Osim navedenog, gotovo uvijek se na neke od slojeva primjenjuje jedna od tzv. aktivacijskih funkcija (vidi [2] i [3]) koje su, u neku ruku, mali klasifikatori sami po sebi. Konvolucijske neuronske mreže najčešće koriste funkciju pod nazivom ReLU (vidi [4]).

Više o konvolucijskim neuronskim mrežama na [5] i [6].

3.1 Arhitekture korištene u projektu

Prilikom izrade projekta korištene su mnoge arhitekture, ali u konačnu verziju projekta dospjele su tri: `CNN_Odin`, `CNN_Odinson` i `CNN_Ymir`. Sve tri su međusobno vrlo slične, a razlika je u broju konvolucijskih i/ili FC slojeva. U projektu je primarno korištena arhitektura `CNN_Odin`, a preostale dvije služe za dobivanje perspektive, tj. koliko na točnost utječe broj određenih slojeva.

3.1.1 Arhitektura `CNN_Odin`

Ova arhitektura je inspirirana poznatom arhitekturom AlexNet (vidi [7]).

Konvolucijski dio					
Tip sloja	Ime sloja	Broj filtera	Jezgra	Padding	Stride
`Conv1d + ReLU`	`Conv1`	96	5	2	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv2`	256	5	2	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv3`	384	5	2	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv4`	384	3	1	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv5`	256	3	1	1
`MaxPool1d (kernel_size=2)`					
`Flatten`					

Fully connected dio		
Tip sloja	Ime sloja	Broj neurona
`Fully connected`	`FC1`	4096
`Fully connected`	`FC2`	2048
`Fully connected`	`FC_final`	`no_of_classes`

3.1.2 Arhitektura `CNN_Odinson`

Konvolucijski dio					
Tip sloja	Ime sloja	Broj filtera	Jezgra	Padding	Stride
`Conv1d + ReLU`	`Conv1`	96	5	2	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv2`	256	5	2	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv3`	384	5	2	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv4`	384	3	1	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv5`	256	3	1	1
`MaxPool1d (kernel_size=2)`					
`Flatten`					

Fully connected dio		
Tip sloja	Ime sloja	Broj neurona
`Fully connected`	`FC1`	2048
`Fully connected`	`FC_final`	`no_of_classes`

3.1.3 Arhitektura `CNN_Ymir`

Konvolucijski dio					
Tip sloja	Ime sloja	Broj filtera	Jezgra	Padding	Stride
`Conv1d + ReLU`	`Conv1`	128	5	2	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv2`	256	5	2	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv3`	512	5	2	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv4`	512	3	1	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv5`	256	3	1	1
`MaxPool1d (kernel_size=2)`					
`Conv1d + ReLU`	`Conv6`	128	3	1	1
`MaxPool1d (kernel_size=2)`					
`Flatten`					

Fully connected dio		
Tip sloja	Ime sloja	Broj neurona
`Fully connected`	`FC1`	2048
`Fully connected`	`FC2`	2048
`Fully connected`	`FC_final`	`no_of_classes`

3.2 Klasa `PollenClassifier`

Ovo je "glavna" klasa projekta. Služi za treniranje, testiranje, spremanje i učitavanje modela, kao i za vizualizaciju točnosti i funkcije cilja kroz epohe treniranja.

3.2.1 Metoda `__init__()`

Konstruktor klase.

Povratni tip: `PollenClassifier`

Parametri:

- `model (torch.nn.Module)` - arhitektura neuronske mreže koju želimo trenirati
- `force_cpu (bool)` - želimo li "prisiliti" slanje modela na CPU, čak i ako nam je GPU dostupan? `(default: False)`

Primjer:

```
classifier = PollenClassifier(model, force_cpu=False)
```

3.2.2 Metoda `train()`

Ova metoda radi sljedeće:

1. Inicijalizira liste u koje će spremati točnost i vrijednost funkcije cilja kroz epohe.
2. Šalje funkciju cilja na uređaj mreže (CPU ili GPU).
3. Ispisuje informacije o mreži: naziv arhitekture, optimizator, stopu učenja, regularizaciju, broj epoha i uređaj na kojem se mreža trenira.
4. Inicijalizira mapu za *autosave* (za `autosave=True`).
5. Ispisuje trenutno vrijeme (za `print_all=True`).
6. Pokreće "štopericu".
7. Trenira neuronsku mrežu:
   (a) Ispisuje broj epohe (za `print_all=True`).
   (b) Inicijalizira vrijednosti (broj točno klasificiranih, broj ukupno klasificiranih i vrijednost funkcije cilja).
   (c) Iterira kroz training podatke i radi propagaciju unaprijed i unatrag.
   (d) Računa točnost i vrijednost funkcije cilja za training podatke.
   (e) Iterira kroz validacijske podatke i radi propagaciju unaprijed.
   (f) Računa točnost i vrijednost funkcije cilja za validacijske podatke.

   (g) Ispisuje stanje na štoperici te izračunate vrijednosti (za `print_all=True`) ili broj epohe (za `print_all=False`).
   (h) Sprema trenutne parametre modela (za `autosave=True`).

8. Zaustavlja štopericu i ispisuje ukupno vrijeme.
9. Ispisuje trenutno vrijeme (za `print_all=True`) ili točnost na validacijskom setu (za `print_all=False`).

Povratni tip: `None`

Parametri:

- `trainloader (DataLoader)` - `DataLoader` koji učitava skup podataka za treniranje
- `validloader (DataLoader)` - `DataLoader` koji učitava skup podataka za validaciju
- `loss_fn` (a `torch.nn` loss function) - funkcija cilja
- `optimizer` (a `torch.optim` optimizer) - algoritam za optimizaciju treniranja
- `epochs (int)` - broj epoha, određuje koliko puta želimo proći kroz podatke prilikom treniranja `(default: 10)`
- `print_all (bool)` - želimo li ispisati detaljne informacije o modelu nakon svake završene epohe? `(default: False)`
- `autosave (bool)` - želimo li spremiti parametre modela u vanjsku datoteku nakon svake završene epohe? `(default: True)`
- `save_path (str)` - putanja do mape u koju želimo spremati parametre modela koristeći *autosave*. Ukoliko lokacija već postoji ili predamo `None`, putanja će biti nazvana po datumu i vremenu početka treninga i bit će ispisana prije početka treninga `(default: None)`

Primjer:

```
model = CNN_Odin(no_of_classes=8, picture_size=608)
classifier = PollenClassifier(model)

crossEntropy = nn.CrossEntropyLoss(weight=class_weights)
adam = optim.Adam(model.parameters(), lr=1e-4, weight_decay=1e-4)

classifier.train(
    trainloader,
    validloader,
    crossEntropy,
    adam,
    epochs=10,
    print_all=True
)
```

3.2.3 Metoda `test()`

Testira model na danim podacima i vraća točnost, pri čemu će za npr. točnost od 72.3% metoda vratiti rezultat 72.3.

Povratni tip: `float`

Parametri:

- `testloader (DataLoader)` - `DataLoader` koji učitava skup podataka za testiranje

Primjer:

```
test_acc = classifier.test(testloader)
print('Test accuracy: %.3f %%' % test_acc)
```

3.2.4 Metoda `testPerClass()`

Radi isto što i `test()`, samo što provjerava točnost za svaku klasu posebno te odmah ispisuje rezultate. Npr. ako klasa ima 100 pripadnika, a mreža je od tih 100 točno klasificirala njih 62, onda će točnost za tu klasu biti 62%.

Povratni tip: `None`

Parametri:

- `testloader (DataLoader)` - `DataLoader` koji učitava skup podataka za testiranje
- `classes (tuple<str>)` - uređena n-torka koja sadrži naziv svake od klasa

Primjer:

```
classes = ('Prva', 'Druga', 'Treca', 'Cetvrta')
classifier.testPerClass(testloader, classes)
```

3.2.5 Metoda `save()`

Sprema trenutno stanje (parametre) modela na danoj lokaciji.

Povratni tip: None

Parametri:

- `path (str)` - putanja do .pth datoteke na koju želimo spremiti stanje modela (ukoliko neka od mapa ne postoji, ista neće biti stvorena automatski, nego ćemo dobiti error)

Primjer:

```
classifier.save('./path/to/folder/File_name.pth')
```

3.2.6 Metoda `load()`

Učitava stanje (parametre) modela sa dane lokacije.

Povratni tip: None

Parametri:

- `path (str)` - putanja do .pth datoteke s koje želimo učitati stanje modela (ukoliko datoteka ne postoji, dobit ćemo error)

Primjer:

```
classifier.load('./path/to/folder/File_name.pth')
```

3.2.7 Privatna metoda `__autosave()`

Radi isto što i `save()`, ali uz predefinirane nazive putanje i datoteke. Poziva se nakon svake epohe treninga, ukoliko metodi `train()` proslijedimo parametar `autosave=True`. Putanja na koju se parametri spremaju je ili proslijeđena kroz metodu `train()` ili je dobivena koristeći trenutni datum i vrijeme.

Povratni tip: None

Parametri:

- `epoch (int)` - broj trenutne epohe

Primjer:

```
self.__autosave(4)
```

3.2.8 Metode `display_acc()` i `display_loss()`

Metoda iscrtava graf ovisnosti točnosti (odnosno funkcije cilja) o broju epohe za trening set i validacijski set. Preporučeno je pozvati obje metode odmah nakon pozivanja metode `train()`.

Povratni tip: None

Parametri: None

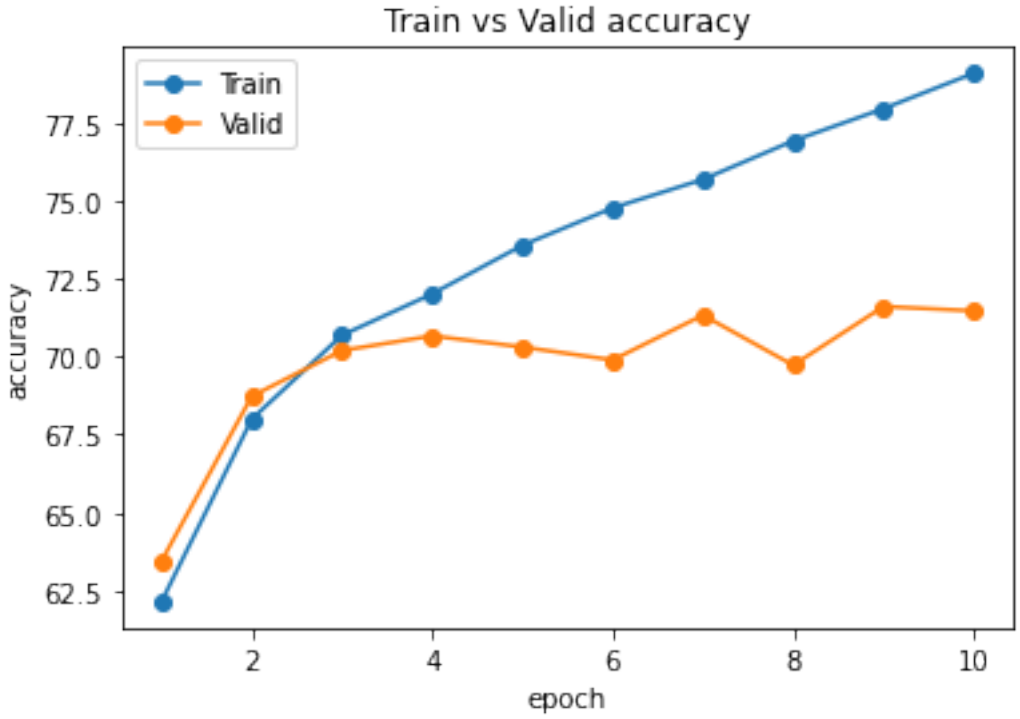
Primjer:

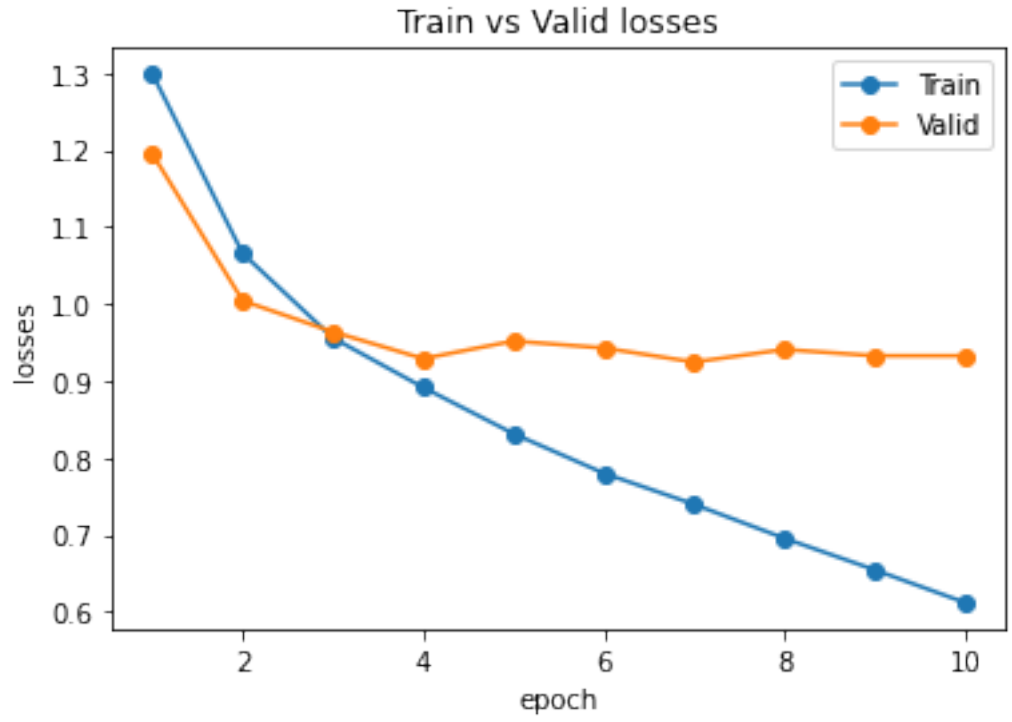
```
classifier.display_acc()
classifier.display_loss()
```

4 Rezultati projekta

4.1 Klasificiranje svih klasa odjednom

Rezultati klasifikacije na testnom skupu kreću se između 60% i 75%, ovisno o arhitekturi mreže, veličini *batch*-a i hiperparametrima. Najbolji rezultat (od 72%) dobije se kada koristimo arhitekturu `CNN_Odin` i *batch* veličine 1, no takav rezultat nije baš najtočniji, budući da mreža dobro klasificira samo neke klase, dok se za ostale čini da ih ignorira.





4.2 Binarni klasifikatori

Rezultati klasifikacije pomoću binarnih klasifikatora su druga priča. Dvije vrste, *Artemisia Vulgaris* i *Cedrus*, mogu se klasificirati s ujednačenom točnošću od 80%, no to su ujedno i najbrojnije vrste u skupu podataka, tako da ovakav rezultat nije začuđujuć.

Treća najbrojnija vrsta, *Quercus*, uspješno je klasificirana s ujednačenom točnošću od čak 90%. Ova vrsta zauzima tek 7.8% skupa podataka, što znači da *Quercus* posjeduje svojstva koja su potpuno drugačija od drugih vrsta ili skup podataka jednostavno sadrži jako dobre reprezentante *Quercusa*.

Preostalih 5 vrsta bilo je gotovo nemoguće klasificirati s ujednačenom točnošću. "Krivac" tome može biti činjenica da skup podataka jednostavno sadrži premalo njihovih reprezentanata. Također, moguće je da ti reprezentanti nisu dovoljno dobri da bi jednoznačno određivali svoju vrstu.

4.3 Klasificiranje preostalih vrsta

Nakon prilično uspješnog pokušaja da se izgrade binarni klasifikatori za gore navedene vrste, uslijedio je ne baš toliko uspješan pokušaj da se klasificiraju preostale, ali nakon što smo iz skupa podataka izbacili one vrste koje smo uspjeli klasificirati pomoću binarnih klasifikatora.

Rezultati ove klasifikacije (na testnom skupu) kreću se između 40% i 50%, pri čemu se rezultati klasifikacije po vrstama kreću od 30% do 60%. Budući da najbrojnije klase sada više "ne smetaju", očekivali bismo veću točnost, no problem vjerojatno leži u premalenom broju reprezentanata ovih klasa.

Literatura

[1] https://pytorch.org/tutorials/beginner/basics/data_tutorial.html

[2] https://en.wikipedia.org/wiki/Activation_function

[3] https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

[4] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

[5] https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[6] https://www.youtube.com/watch?v=bNb2fEVKeEo&ab_channel=StanfordUniversitySchoolofEngineering

[7] https://www.analyticsvidhya.com/blog/2021/03/introduction-to-the-architecture-of-alexnet/