Predmet: "Vještačka inteligencija"
Laboratorijska vježba 3: Neuronske mreže (1/4)
Odgovorna nastavnica: Vanr. prof. dr Amila Akagić

Sadržaj vježbe:

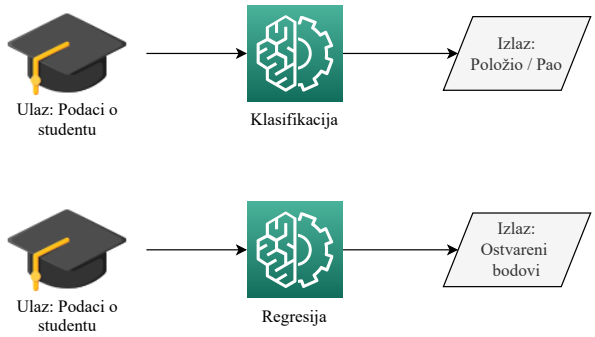
1 Cilj vježbe

Na prvoj laboratorijskoj vježbi iz neuronskih mreža, studenti se upoznaju sa problemom klasifikacije, kao jednim od najčešćih problema iz prakse. Kroz zadatke, studenti rješavaju jednostavne probleme klasifikacije nad tekstualnim ulaznim podacima, koristeći Keras razvojni okvir.

2 Neuronske mreže u klasifikaciji

2.1 Problem klasifikacije

Čest zadatak algoritama vještačke inteligencije je da prepoznaju objekte kao pripadnike neke kategorije, odnosno klase. Ovaj proces je poznat pod nazivom klasifikacija i on omogućava razvrstavanje ulaznih instanci podataka u unaprijed definisane diskretne vrijednosti (klase). Klasifikacija je jedan od dva tipa nadziranog učenja (eng. *supervised learning*). Drugi je regresija, i ona će biti fokus naredne vježbe. Razlika između klasifikacije i regresije je primjerom objašnjena na slici 1.
Kada ljudi uče, prvo se oslanjaju na poznata rješenja, kako bi znali da li uče ispravno. Nakon što čovjek svoju vještinu istrenira na riješenim primjerima, svoje znanje može primijeniti (testirati) nad novim problemima, čija rješenja nisu poznata. Mogu se, dakle, formirati dva osnovna koraka u procesu nadziranog učenja, i to: treniranje, u kojem se uči na osnovu riješenih primjera, i testiranje, gdje se primijeni naučeno iz faze treniranja nad neriješenim primjerima. Opcionalno može se uvesti i međukorak koji se zove validacija. On služi kako bi se provjeravalo da model ne bude previše prialogođen na trening skup podataka i onda počne loše raditi na svim novim podacima koje do tada nije vidio. Ovaj problem se zove problem *overfitting*-a.



Slika 1: Razlika između klasifikacije i regresije. Klasifikacija ulazne podatke pridruži nekoj predefinisanoj klasi, dok regresija pokušava odrediti neku kontinualnu vrijednost na osnovu ulaza.

Postoje tri glavna tipa klasifikacije:

1. Binarna klasifikacija (eng. *binary classification*);
2. Višeklasna klasifikacija (eng. *multiclass classification*);
3. Klasifikacija sa više labela pripadnosti (eng. *multilabel classification*).

2.2 Dizajn neuronske mreže za klasifikaciju kroz Python i Keras

Neuronske mreže se sastoje od jednostavnih elemenata koji se nazivaju neuroni. U najjednostavnijem obliku, oni na ulazu primaju realne vrijednosti, množe ih sa nekim težinskim koeficijentom i šalju ih kroz neku aktivacijsku funkciju. Neuronska mreža u opštem slučaju sastoji od više slojeva, svaki sloj se sastoji od više pomenutih neurona i podaci sa ulaza prolaze kroz povezane neurone i dolaze do krajnjeg sloja koji se naziva izlaznim. Prednosti neuronskih mreža ogledaju se u tome što su vrlo efikasne za probleme visoke dimenzionalnosti i, bar teoretski uz dovoljno računarske moći, može naučiti proizvoljan oblik funkcije. Međutim, za neke jednostavnije probleme nema potrebe za neuronskim mrežama te i jednostavniji algoritmi se mogu pokazati sasvim dovoljnim. Također, neuronske mreže često zahtijevaju velik skup podataka za treniranje kako bi se postigli zadovoljavajući rezultati.

Prije nego se krene sa dizajniranjem modela neuronske mreže, prvo se podaci moraju pripremiti i svesti na odgovarajući oblik. Ovaj korak se naziva preprocesiranje podataka.

2.2.1 Predprocesiranje podataka

U okviru ovog koraka prvo je potrebno učitati skup podataka. Ukoliko se radi o nekom skupu podataka koji se nalazi u sklopu Kerasa to se može jednostavno učiniti kao što je učinjeno u prethodnoj vježbi. Ukoliko to nije slučaj, potrebno je prilagoditi se skupu podataka i, u najčešćem slučaju, učitati ga kao CSV datoteku. Zatim je potrebno razdvojiti labele od podataka, te podijeliti skup podataka u skup za treniranje i skup za testiranje.

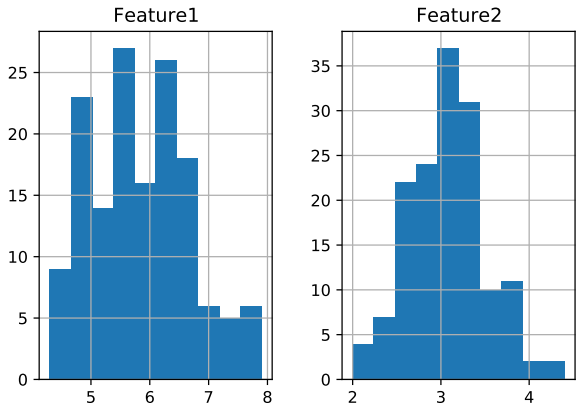
Često podaci iz skupa podataka imaju više atributa/karakteristika (eng. *feature*). Prije nego se predprocesiraju podaci, treba se prvo upoznati sa njima. Biblioteka `Pandas` olakšava ovaj korak te omogućava jednostavan prikaz osnovnih informacija o skupu podataka poput minimalne vrijednosti, maksimalne vrijednosti, srednje vrijednosti, standardne devijacije, i tako dalje. Ovo se ostvaruje sljedećim kodom:

```
data = pd.read_csv('dataset.csv')
data.describe()
```

Također, `Pandas` modul omogućava i crtanje histograma za svaki od atributa. To se može uraditi sljedećim kodom:

```
data = pd.read_csv('dataset.csv')
data.hist()
```

Prikaz karakteristika podataka, kao i histograma, je dat na slici 2.



(a) Prikaz histograma.

	Feature1	Feature2
count	150.000000	150.000000
mean	5.843333	3.057333
std	0.828066	0.435866
min	4.300000	2.000000
25%	5.100000	2.800000
50%	5.800000	3.000000
75%	6.400000	3.300000
max	7.900000	4.400000

(b) Prikaz osonvnih informacija o skupu podataka.

Slika 2: Prikaz osnovnih informacija o podacima.

U slučaju da se podaci značajno razlikuju, bilo unutar istog atributa ili opsezi koje atributi mogu zauzimati, često se pokazuje da je potrebno vršiti neki vid normalizacije. Na prethodnoj vježbi su spomenuti *MinMax* i *Z-score* normalizacija. Često se, kada se kaže samo normalizacija, misli na *MinMax*, a za *Z-score* se često koristi naziv standardizacija. Normalizacija svodi sve vrijednosti na opseg od 0 do 1, dok standardizacija transformiše podatke tako da oni imaju srednju vrijednost 0, a standardnu devijaciju 1. U prethodnoj vježbi je pokazano kako se ove tranfsormacije mogu vršiti. Tada se "učenje" *scaler*-a, i vršenje samog skaliranja radilo u jednom koraku. Međutim, u nekim slučajevima, kao što je pri radu sa neuronskim mrežama, potrebno je ta dva koraka razdvojiti. S obzirom da pri pripremi podataka za treniranje neuronske mreže treba koristiti isključivo podatke za treniranje, to je *scaler*-e potrebno prilagoditi samo tom skupu, te ga primijeniti kao takvog i na skup za treniranje i na skup za testiranje. Razlog za to je što skup za tesiranje se treba tretirati kao nepoznat, jer uistinu, u stvarnoj primjeni modela neće biti poznati podaci jer će oni stalno biti novi. Prema tome sljedeći kodovi omogućavaju opisano razdvajanje koraka pri skaliranju podataka:

```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler().fit(x_train)
x_train = scaler.fit_transform(x_train)
x_test = scaler.fit_transform(x_test)
```

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler().fit(x_train)
x_train = scaler.transform(x_train)
x_test = scaler.transform(x_test)
```

2.2.2 Odabir arhitekture mreže

Dizajniranje arhitekture neuronske mreže nije nimalo jednostavan zadatak. Ne postoji fiksan skup pravila koji se može slijediti kojim će nam biti zagarantovana uspješnost modela. Dizajniranje je iterativan proces i dobrim dijelom se bazira na metodi pokušaja i pogrešaka (eng. *trial and error*).

Sam problem definišu ulazni i izlazni sloj. Dimenzije podataka definišu koliko neurona će biti u ulaznom sloju. Ukoliko nakon predprocesiranja skup podataka ima 100 atributa, tada ulazni sloj mora imati isto toliko neurona. Slično, broj izlaznih neurona je određen onime što se želi postići. Ako se radi o binarnoj klasifikaciji, tada je dovoljan jedan neuron koji će davati informaciju da li je izlaz klasa 1 ili klasa 0. Ukoliko je riječ o višeklasnoj klasifikaciji sa 10 klasa, tada je potrebno 10 neurona da se enkodira u šta je mreža klasificirala ulazni podatak. Pri tome, za ovaj slučaj vrijednost *i-tog* neurona predstavlja vjerovatnoću koju je mreža dodijelila da ulazni podatak pripada *i-toj* klasi i očigledno zbir svih vrijednosti na neuronima na posljednjem sloju tada mora biti jednak 1.

Između ova dva sloja se nalazi skup skrivenih slojeva. Pri odabiru broja skrivenih slojeva potrebno je postići balans - ni previše ali ni premalo slojeva, u zavisnosti od problema. Pored odabira broja skrivenih slojeva, potrebno je

odabrati i broj neurona u njima. U većini jednostavnijih problema je sasvim dovoljno da svi skriveni slojevi imaju isti broj neurona.

Osim odabira o broju i veličini slojeva, potrebno je i odabrati koju vrstu slojeva koristiti. Postoji mnogo slojeva, a najjednostavniji je `Dense` sloj, sa kojim će se i raditi na ovoj vježbi. Ovaj sloj se još naziva i **potpuno povezani sloj** (eng. *fully connected layer*), jer su kod njega svi ulazi spojeni sa svim izlazima prethodnog sloja. Koristi se kada povezanost može postojati između bilo koje dva atributa.

Posljednja stvar koju treba odabrati prilikom dizajna arhitekture mreže su aktivacijske funkcije za neurone. Kao generalno pravilo za skrivene slojeve najčešće se koristi *relu (Rectified Linear Units)* aktivacijska funkcija. Sofisticiraniji radovi nekada koriste nadogradnje na ovu funkciju poput *leakyReLU*, *ELU*, *GELU*, itd. Za aktivacijsku funkciju izlaznog sloja, koristi se *softmax* kod višeklasne klasifikacije, a *sigmoid* kod binarne klasifikacije.

Koristeći Keras se iznad objašnjeno vrlo jednostavno primjenjuje. Primjer za model koji bi mogao poslužiti za binarnu klasifikaciju je dat sljedećim kodom:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(8, activation='relu', input_shape=(4000,)))
model.add(layers.Dense(8, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

Prvo se definiše sekvencijalni model u kojeg se redom dodaju željeni slojevi. Ulazni sloj nije potrebno eksplicitno navoditi već se on određuje ulaznom dimenzijom prvog skrivenog sloja. Tako se, u primjeru iznad, dodaje `Dense` sloj sa 8 neurona, ReLU aktivacijskom funkcijom, a parametrom `input_shape` se definiše da ulazni sloj ima 4000 elemenata (ovo mora odgovarati broju atributa skupa podataka). Drugi skriveni sloj se dodaje na isti način, osim što nije potrebno specificirati ulaznu dimenziju jer se ona automatski određuje na osnovu prethodnog sloja. Posljednji sloj je izlazni sloj, a pošto je ovo model za binarnu klasifikaciju, on će imati jedan neuron i aktivacijsku funkciju `sigmoid`.

2.2.3 Konfigurisanje optimizatora i hiperparametara

Nakon što je definisana arhitektura mreže, potrebno je definisati:

- Mjeru performanse mreže prilikom treniranja koja se nastoji optimizirati – funkcija gubitka;
- Optimizator - odnosno algoritam koji će tokom procesa treniranja nastojati unaprijediti mjeru performanse mreže;
- Metrike koje će se pratiti na modelu.

Funkcija gubitka za regresiju je srednja kvadratna greška, za binarnu klasifikaciju binarna unakrsna entropija (eng. *binary cross entropy*), a za višeklasnu klasifikaciju kategorička unakrsna entropija (eng. *categorical cross entropy*).

Za optimizator je nešto širi izbor, no često se koriste RMSprop i Adam. Oba ova algoritma su zasnovana na gradijentu. U okviru Kerasa podrazumijevana početna vrijednost stope učenja za oba ova algoritma je 0.001.

Metrika je primarno definisana domenom problema i o tome šta je bitno za neki konkretan problem. Međutim, metrika koja se koristi u gotovo svim primjenama je tačnost i ona mjeri broj ispravno klasificiranih instanci podataka u odnosu na ukupan broj podataka koji su podvrgnuti klasifikaciji.

U Kerasu se sve pomenuto može postaviti jednom linijom koda:

```
model.compile(optimizer='adam',loss='binary_crossentropy',metrics=['accuracy'])
```

Ovaj način će definisati Adam optimizator sa podrazumijevanim postavkama, no ukoliko je potrebno eksplicitno podesiti parametre optimizatora to se može učinti sljedećim kodom:

```
opt = keras.optimizers.Adam(learning_rate=0.01)
model.compile(loss='categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Na samom kraju definisanja modela, potrebno je konfigurisati i **hiperparametre**. To su parametri koji su eksterni u odnosu na model i ne mogu se estimirati na osnovu podataka. U hiperparametre spadaju:

- Stopa učenja - kontroliše koliko brzo model uči (može biti konstanta ili adaptivna);
- Veličina *batch*-a - kontroliše koliko instanci se obradi prije nego što se ažuriraju težine modela;
- Broj epoha - predstavlja broj prolaza kroz sve podatke trening skupa u procesu treniranja.

2.2.4 Proces treniranja i tumačenja rezultata

Proces treniranja neuronskih mreža je računski zahtjevan i ovisno od modela, skupa podataka i hiperparametara može trajati i veći broj dana. Kako bi se ubrzao proces treniranja, poželjno je koristiti GPU ukoliko je kompatibilna sa TensorFlow. U Kerasu se treniranje izvršava poprilično jednostavno pozivom metode fit nad definisanim modelom. Primjer poziva je dat ispod.

```
history = model.fit(x_train, y_train, epochs=20, batch_size=64, validation_split
    =0.2)
```

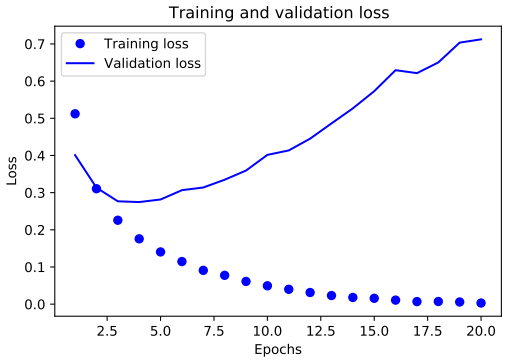
Prvi parametar metode `fit` su podaci za treniranje, drugi parametar je niz labela koje odgovaraju skupu iz prvog parametra. Zatim se specificira broj epoha, veličina *batch*-a te udio podataka koji će se uzeti iz trening skupa i koristiti se za validaciju. Nakon pozivanja ove metode dobija se prikaz o progresu treniranja. Primjer takvog prikaza se može vidjeti na slici 3.

```
Epoch 1/20
30/30 [==============================] - 3s 75ms/step - loss: 0.5874 - accuracy: 0.7139 - val_loss: 0.3841 - val_accuracy: 0.87
35
Epoch 2/20
30/30 [==============================] - 1s 36ms/step - loss: 0.3270 - accuracy: 0.9047 - val_loss: 0.3102 - val_accuracy: 0.88
63
Epoch 3/20
30/30 [==============================] - 1s 36ms/step - loss: 0.2326 - accuracy: 0.9246 - val_loss: 0.2898 - val_accuracy: 0.88
51
Epoch 4/20
30/30 [==============================] - 1s 28ms/step - loss: 0.1840 - accuracy: 0.9404 - val_loss: 0.2735 - val_accuracy: 0.89
41
Epoch 5/20
30/30 [==============================] - 1s 29ms/step - loss: 0.1446 - accuracy: 0.9569 - val_loss: 0.3141 - val_accuracy: 0.87
61
```

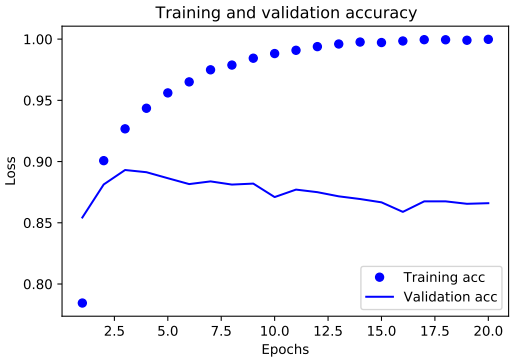
Slika 3: Prikaz procesa treniranja koristeći Keras. U realnom vremenu se prikazuju vrijednosti funkcije gubitka nad trening i validacijskim skupovima, kao i vrijednosti tačnosti. Prikazuje se koja se trenutno epoha trenira kao i koliko vremena traje treniranje po epohi.

Kao rezultat metoda `fit` vraća `History` objekat koji sadrži dosta korisnih informacija o procesu treniranja, među kojima je i dictionary pod nazivom `history` u kome su vrijednosti funkcije gubitka i metrika koje su definisane za model pri kompajliranju za svaku epohu nad skupom za treniranje i za validaciju (ukoliko je validacija vršena). Na osnovu tih podataka se vrši analiza i donosi odluka o tome da li je model dovoljno dobar i spreman da se koristi nad pravim, odnosno testnim podacima. Sljedećim kodom se može nacrtati vrijednosti funkcije gubitka i tačnosti po epohama:

```
acc = history.history['accuracy']
loss_values = history.history['loss']
val_loss_values = history.history['val_loss']
epochs = range(1, len(acc) + 1)
plt.plot(epochs, loss_values, 'bo', label='Training loss')
plt.plot(epochs, val_loss_values, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```



(a) Prikaz vrijednosti funkcije gubitka.



(b) Prikaz tačnosti modela.

Slika 4: Prikaz rezultata nakon treniranja. Ovo je klasičan primjer *overfitting*-a jer se vrijednosti metrika i funkcije gubitka značajno počinju razlikovati nad validacijskom i trening skupu.

```
plt.show()

plt.clf()
acc_values = history.history['accuracy']
val_acc_values = history.history['val_accuracy']
plt.plot(epochs, acc_values, 'bo', label='Training acc')
plt.plot(epochs, val_acc_values, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
```

Kao rezultat ovog koda dobija se prikaz kao na slici 4.
Ukoliko se odluči da model treba promijeniti te izvršiti ponovno treniranje, potrebno je resetovati ono što je model do sada naučio. Nije dovoljno samo rekompajlirati model, već je potrebno model opet u potpunosti definisati ili alternativno pohraniti težine modela prije početka treniranja te ih kasnije ponovo učitati. Sljedeći kod ilustruje spašavanje i učitavanje težina u model:

```
# ... Definisanje modela ...

model.save_weights('model.h5') # Spasavanje modela

# ... Ucitavanje koeficijenata spasenog modela ...

model.load_weights('model.h5') # Ponovno ucitavanje modela
```

2.2.5 Korištenje treniranog modela

Kada su postignuti zadovoljavajući rezultate pri treniranju modela, on se može koristiti nad testnim skupom. Ovo se radi pomoću metode `evaluate`. Primjer njenog poziva je dat kao:

```
results = model.evaluate(x_test, y_test)
```

Osnovni parametri koje funkcija `evaluate` prima su skup atributa podataka za testiranje i niz odgovarajućih labela. Kao rezultat se dobija vrijednost funkcije gubitka i vrijednosti metrika nad testnim skupom podataka.

U stvarnoj primjeni koristi se metoda `predict` koja daje rezultat klasifikacije modela na nekom podatku ili skupu podataka koji se želi klasificirati i za koji nije poznata tačna klasa. Za ovu svrhu se i pravi i trenira model kako bi on dao što bolju predikciju tačne klase koja nije poznata. Primjer poziva ove funkcije je dat sa kodom:

```
prediction = model.predict(real_data)
```

3 Zadaci za rad u laboratoriji

Predviđeno je da se svi zadaci u nastavku rade u sklopu Jupyter Notebook okruženja. Svaki podzadatak treba biti zasebna Jupyter ćelija.

Zadatak 1 - Binarna klasifikacija - Klasifikacija vina

U ovom zadatku se radi najjednostavniji oblik klasifikacije, a to je binarna klasifikacija na osnovu numeričkih podataka u vidu tekstualne datoteke. Cilj ovog zadatka je da se napravi i istrenira neuronska mreža koja će klasificirati vino na osnovu raznih karakteristika kao crno ili bijelo (slika 5). Skup podataka se sastoji iz dvije datoteke, jedne za crna vina i druge za bijela.

Slika 5: Bijelo i crno vino - tipičan primjer binarne klasifikacije.

a) Učitati podatke za crno i bijelo vino koji se nalaze u CSV datotekama *winequality-red.csv* i *winequality-white.csv* u varijable `red` i `white` respektivno. Pri ovome postaviti parametar `separator` da bude znak tačka-zarez (*;*). U odgovarajućim `DataFrame` objektima dodati novu kolonu `label` koja će imati vrijednost 0 za podatke bijelog vina, a vrijednost 1 za podatke crnog vina. Nakon ovoga spojiti ih u jedan `DataFrame` objekat `wines`.

b) Izvršiti prikaz osnovnih podataka spojenog skupa podataka korištenjem `describe` metode te nacrtati histograme korištenjem *hist* metode. Šta možete zaključiti o podacima?

c) Iz `wines` izdvojiti `X` - karakteristike i `y` - labele. Zatim korištenjem funkcije `train_test_split` podijeliti podatke na one za treniranje i one za testiranje pri čemu 20% podataka trebaju biti podaci za testiranje;

d) Formirati sekvencijalni Keras model koji će imati 2 `Dense sloja` sa po 8 neurona i `relu` aktivacijskom funkcijom. Prvom sloju kao `input_shape` parametar proslijediti vrijednost `(12,)` s obzirom na to da skup podataka ima 12 značajki. Treći sloj, koji je u ovom slučaju izlazni sloj, postaviti da također bude `Dense`, ali sa samo jednim neuronom;

e) Kompajlirati model tako da koristi `adam` optimizator, za funkciju gubitka koristiti `binary_crossentropy`, a kao metriku odabrati `accuracy`;

f) Model istrenirati na 20 epoha, sa veličinom *batch-a* 16. Kolika je postignuta tačnost, a kolika vrijednost funkcije gubitka na kraju treniranja?

g) Ponoviti postupak[1], no ovaj put prije treniranja izvršiti standardizaciju, odnosno skaliranje atributa korištenjem `StandardScaler` objekat iz `sklearn.preprocessing` modula. Kolika je sada tačnost nakon treniranja? Uporediti rezultate sa onim bez skaliranja;

h) Model evaluirati nad testnim skupom podataka. Kolika je postignuta tačnost modela?

[1] Obavezno ponovo kreirati model ispočetka kako bi se istrenirani parametri ponovo reinicijalizirali