Zad.1 Kod znajduje się na udostępnionym githubie. Aby nie powielać informacji w sprawozdaniu pozwoliłam sobie ominąć opis tego zadania. Dodatkowo w konwersajcji podczas pierwszych zajęć wyczytałam, że Pan pokazał i zrobił wraz z grupą wszystkie pierwsze 4 zadania i sprawozdanie należało oddać od 5 zadania w górę. Nie wiem na czym ostatecznie stanęło. Liczę, że w poniższym sprawozdaniu widać ogrom włożonego czasu i chęci, aby zrealizować ze zrozumieniem wszystkie zadania.

Zad. 2 Neuronowa sieć gęsta dla problemu klasyfikacji recenzji filmowych z bazy IMDB.

Mamy tutaj do czynienia z analizą sentymentalną:

Jest to proces obliczeniowego identyfikowania i kategoryzowania opinii z kawałków tekstu i określania czy nastawienie opiniodawcy wobec konkretnego zagadnienia produktu jest pozytywne, negatywne, czy neutralne.

```
[18] from tensorflow.keras.models import Sequential
     from tensorflow.keras.layers import Embedding, Dense, LSTM, add

     model = Sequential()
     model.add(Embedding(max_features, 128)) # Osadzmy 88587 elementów w 128 przestrzeniach wektorowych
     model.add(LSTM(64, dropout=0.2, recurrent_dropout=0.2)) #LSTM zostają mu przekazane wektory z wcześniejszej wartwy
     # algorytm LSTM koduje je i zwraca binarną label = etykietkę, stąd 1 neuron w Dense warstwie
     model.add(Dense(1, activation='sigmoid')) #y_train są to wartości binarne, dlatego najlepiej przyjąć dla bin wartości funkcjęw kształcie "S"

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

```
model.fit(x_train_pad, y_train,
          batch_size=32,
          epochs=2,
          validation_split=0.3)
```

```
Epoch 1/2
547/547 [==============================] - 150s 269ms/step - loss: 0.5118 - accuracy: 0.7343 - val_loss: 0.3573 - val_accuracy: 0.8484
Epoch 2/2
547/547 [==============================] - 146s 268ms/step - loss: 0.1959 - accuracy: 0.9284 - val_loss: 0.3627 - val_accuracy: 0.8447
<tensorflow.python.keras.callbacks.History at 0x7fe42f9a95f8>
```

Pytania i eksperymenty:

Czy sprawność sieci zmieni się jeżeli:

- będzie tylko jedna warstwa gęsta?

Zostawiłam warstwę Dense i nic zaskakującego się nie pojawiło. Wyniki są beznadziejne jak można było przewidzieć.

```
Epoch 1/2
547/547 [==============================] - 1s 2ms/step - loss: 2845.4465 - accuracy: 0.5052 - val_loss: 269.0479 - val_accuracy: 0.4969
Epoch 2/2
547/547 [==============================] - 1s 2ms/step - loss: 86.4147 - accuracy: 0.4963 - val_loss: 40.0186 - val_accuracy: 0.4932
<tensorflow.python.keras.callbacks.History at 0x7fe426d42898>
```

- będą trzy warstwy gęste?

Jest to wyjściowa forma.

- liczba jednostek w warstwie/warstwach zmniejszy się?

W warstwie LSTM liczbę jednostek zmniejszyłam do 32 i oto wyniki:

```
Epoch 1/2
547/547 [==============================] - 131s 234ms/step - loss: 0.5245 - accuracy: 0.7173 - val_loss: 0.3549 - val_accuracy: 0.8512
Epoch 2/2
547/547 [==============================] - 128s 234ms/step - loss: 0.1829 - accuracy: 0.9332 - val_loss: 0.4625 - val_accuracy: 0.8247
<tensorflow.python.keras.callbacks.History at 0x7fe42f5b74e0>
```

Najwyraźniej nie ma to znaczącego wpływu, ponieważ wyniki są bardzo zbliżone do tych uzyskanych z liczbą jednostek równą 64, a nawet precyzja dla zbioru uczącego się tutaj jest większa. Za to istotnie przyspieszyło proces z 270ms/step na 234ms/step.

- liczba jednostek w warstwie/warstwach zwiększy się?

```
[33] model.fit(x_train_pad, y_train,
               batch_size=32,
               epochs=2,
               validation_split=0.3)
```

```
Epoch 1/2
547/547 [==============================] - 229s 413ms/step - loss: 0.5574 - accuracy: 0.6932 - val_loss: 0.3425 - val_accuracy: 0.8513
Epoch 2/2
547/547 [==============================] - 225s 412ms/step - loss: 0.2158 - accuracy: 0.9176 - val_loss: 0.4058 - val_accuracy: 0.8421
<tensorflow.python.keras.callbacks.History at 0x7fe430800908>
```

Wyniki są bardzo podobne do tych z mniejszą liczbą jednostek, lecz lekko gorsze. Za to warto zwrócić uwagę na to że czas uczenia jest krótszy.

Dodam od siebie, ze jestem świadoma, że gdybym dodała znacznie więcej neuronów to mogłoby dojść do overfittingu - przetrenowania modelu i wyniki były by bardzo niezadowalające.

- zamiast entropii krzyżowej wykorzystamy inną funkcję straty, np. błąd średniokwadratowy?

Wykorzystałam proponowany ‘mse’

```
Epoch 1/2
547/547 [==============================] - 155s 280ms/step - loss: 0.2029 - accuracy: 0.6998 - val_loss: 0.1117 - val_accuracy: 0.8471
Epoch 2/2
547/547 [==============================] - 153s 280ms/step - loss: 0.0558 - accuracy: 0.9299 - val_loss: 0.1237 - val_accuracy: 0.8320
<tensorflow.python.keras.callbacks.History at 0x7fe4274726d8>
```

Wyniki są prawie takie jak oryginał. Finalnie zbiór testowy wypada lepiej pod względem trafności (accuacy) z funkcją straty binary_crossentropy, natomiast dla zestawu uczącego się wynik poprawił się nieznacznie.

- w ostatniej warstwie funkcją aktywacji będzie tangens hiperboliczny?

```
Epoch 1/2
547/547 [==============================] - 150s 270ms/step - loss: 0.7436 - accuracy: 0.6746 - val_loss: 0.3955 - val_accuracy: 0.8381
Epoch 2/2
547/547 [==============================] - 148s 270ms/step - loss: 0.2646 - accuracy: 0.9074 - val_loss: 0.4999 - val_accuracy: 0.8324
<tensorflow.python.keras.callbacks.History at 0x7fe4243ff748>
```

Czas trwania epoki jest praktycznie taki sam. Trafność (accuracy) modelu jest gorsza w porównaniu z funkcją sigmoid, a strata (loss) znacznie wyższa. Dla danego zbioru i problemu lepsza okazała się funkcja sigmoid.

Na podstawie powyższych eksperymentów wybierz najlepsze wartości hiperparametrów sieci (liczba warstw, liczba jednostek w warstwie, funkcje aktywacji, rozmiar wsadu, liczba epok, itd.) Dla najlepszych wartości hiperparametrów wytrenuj sieć na całym zbiorze treningowym (25000 próbek). Sprawdź trafność klasyfikacji (accuracy) tak wytrenowanej sieci na zbiorze testowym.

```
#Przetestowanie modelu z najlepszymi parametrami

score, acc = model.evaluate(x_test_pad, y_test)
print('Wynik testu: ', score)
print('Trafność testu: ', acc)
```

```
782/782 [==============================] - 15s 19ms/step - loss: 0.3901 - accuracy: 0.8383
Wynik testu:  0.3900516927242279
Trafność testu:  0.8383200168609619
```

Zad.3

Czy dla sieci neuronowej i zbioru „reuters” (https://keras.io/api/datasets/reuters/) sprawność zmieni się w stosunku do Rozdziału 3.6 [1] lub [2] jeżeli:

- będzie tylko jedna warstwa gęsta?
- będą trzy warstwy gęste?
- liczba jednostek w warstwie/warstwach zmniejszy się?
- liczba jednostek w warstwie/warstwach zwiększy się?
- rozmiar wsadu zmniejszy się?
- rozmiar wsadu zwiększy się?

Dla najlepszych wartości hiperparametrów znalezionych w trakcie powyższych eksperymentów, wytrenuj sieć na całym zbiorze treningowym. Sprawdź trafność klasyfikacji (accuracy) tak wytrenowanej sieci na zbiorze testowym.

Jest to prawie takie samo zadanie jak zad.1. dlatego pozwolę sobie je pominąć. Pewnie Pan na zajęciach omawiał poszczególne wariacje i było to bardzo kształcące. Ja sama przeklikując żadnych nowych wniosków nie wyciągnę.

Zad 4

1) Eksperymentalnie dobierz najlepsze wartości hiperparametrów sieci neuronowej dla zadania regresji dla danych ze zbioru „Boston Housing” (https://keras.io/api/datasets/boston_housing/).

Zadanie polega na przewidzeniu ceny domów. W zbiorze mamy 13 feutures, czyli cech z których oszacowywana będzie cena dla konkretnego domu, a wierszy jest 506.

Przeprowadziłam normalizację funkcji/cech. Normalizacja cech polega na odjęciu średniej cech od każdej cechy i podzieleniu każdego wyniku przez odchylenie standardowe.

Oczywiście taka normalizacja jest konieczna, ponieważ chcemy, aby dane były jakoś jednostkowe, a nie rozproszone.

```
import numpy as np

train_mean = np.mean(x_train, axis=0)
train_std = np.std(x_train, axis=0)
x_train = (x_train - train_mean) / train_std
print(f'Zbiór uczący: {x_train.shape}, zbiór walidacyjny: {x_test.shape}')
x_train[1]
```

```
Zbiór uczący: (404, 13), zbiór walidacyjny: (102, 13)
array([-0.40342651,  2.99178419, -1.33391162, -0.25683275, -1.21518188,
        1.89434613, -1.91036058,  1.24758524, -0.85646254, -0.34843254,
       -1.71818909,  0.43190599, -1.32920239])
```

W modelu zastosowałam rmsprop jako funkcję optymalizującą proces uczenia się. Wynika to z podstawowej wiedzy statystycznej, czym jest rmse:

Podstawowy błąd średniokwadratowy (RMSE) to odchylenie standardowe reszt (błędów przewidywania). Reszty są miarą odległości od punktów danych linii regresji; RMSE jest miarą rozłożenia tych reszt. Innymi słowy, informuje, jak skoncentrowane są dane wokół linii najlepszego dopasowania. Podstawowy błąd średniokwadratowy jest powszechnie stosowany w klimatologii, prognozowaniu i analizie regresji w celu weryfikacji wyników eksperymentalnych.

Matryka ‘mae’ ukazuje nam mean absolute error, czyli średni błąd bezwzględny, a ‘mse’ to to co opisałam powyżej - jest to różnica między estymatorem, a wartością estymowaną.

Model stopuje się po 50 epokach w których nie doszło do poprawy wyniku straty zbioru walidacyjnego ‘val_loss’

```
from tensorflow.keras.models import Sequential
from keras import layers
from tensorflow import keras

def build_model():
  model = keras.Sequential()
  model.add(layers.Dense(20, activation='relu', input_shape=(x_train.shape[1],)))
 # model.add(layers.Dense(64, activation='relu'))
  model.add(layers.Dense(1))

  model.compile(optimizer='rmsprop',
                loss='mse',
                metrics=['mae', 'mse'])
  return model
```

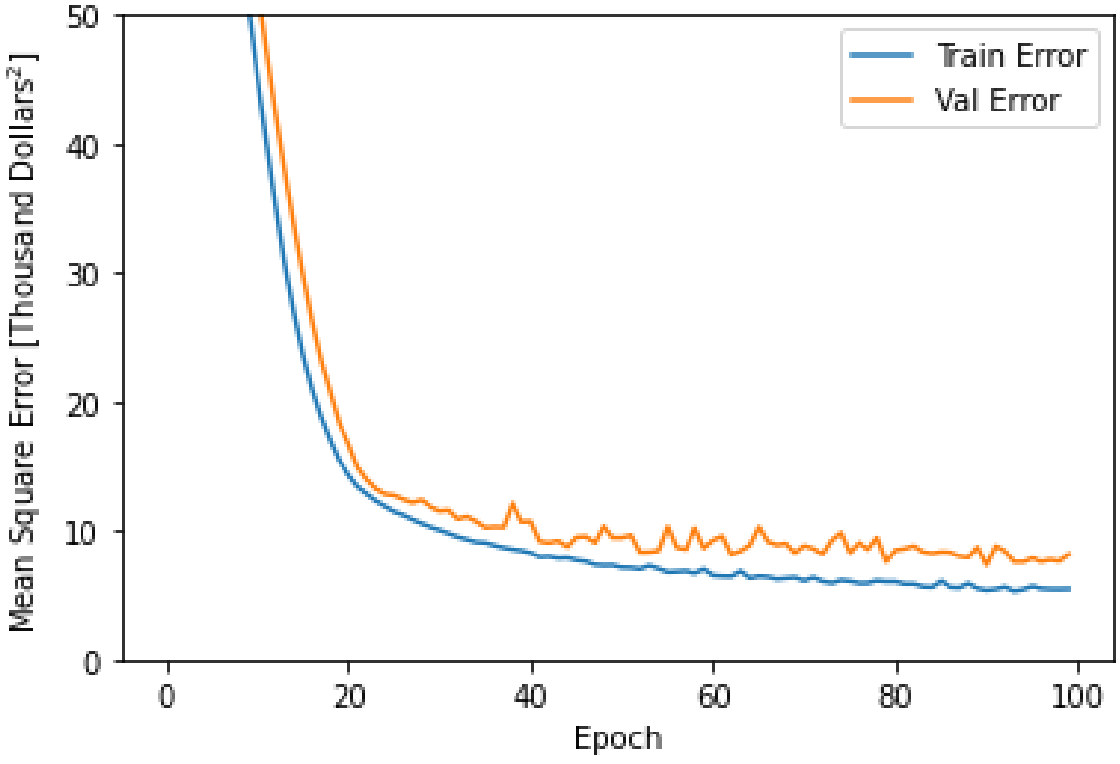
```
import pandas as pd #for visualization

class PrintDot(keras.callbacks.Callback):
    def on_epoch_end(self, epoch, logs):
        if epoch % 10 == 0: print('')
        print('.', end='')

model = build_model()

early_stop = keras.callbacks.EarlyStopping(monitor='val_loss', patience=50)
history = model.fit(x_test, y_test,
                    epochs=100,
                    verbose=0,
                    validation_split = 0.1,
                    callbacks=[early_stop, PrintDot()])
```

Model i wyniki:

```
hist = pd.DataFrame(history.history)
hist['epoch'] = history.epoch

# sprawdzam RMSE miara, aby porównać się z najlepszymi osiągami: https://www.kaggle.com/c
rmse_final = np.sqrt(float(hist['val_mae'].tail(1)))
print()
print('Final Root Mean Square Error on validation set: {}'.format(round(rmse_final, 3)))
```

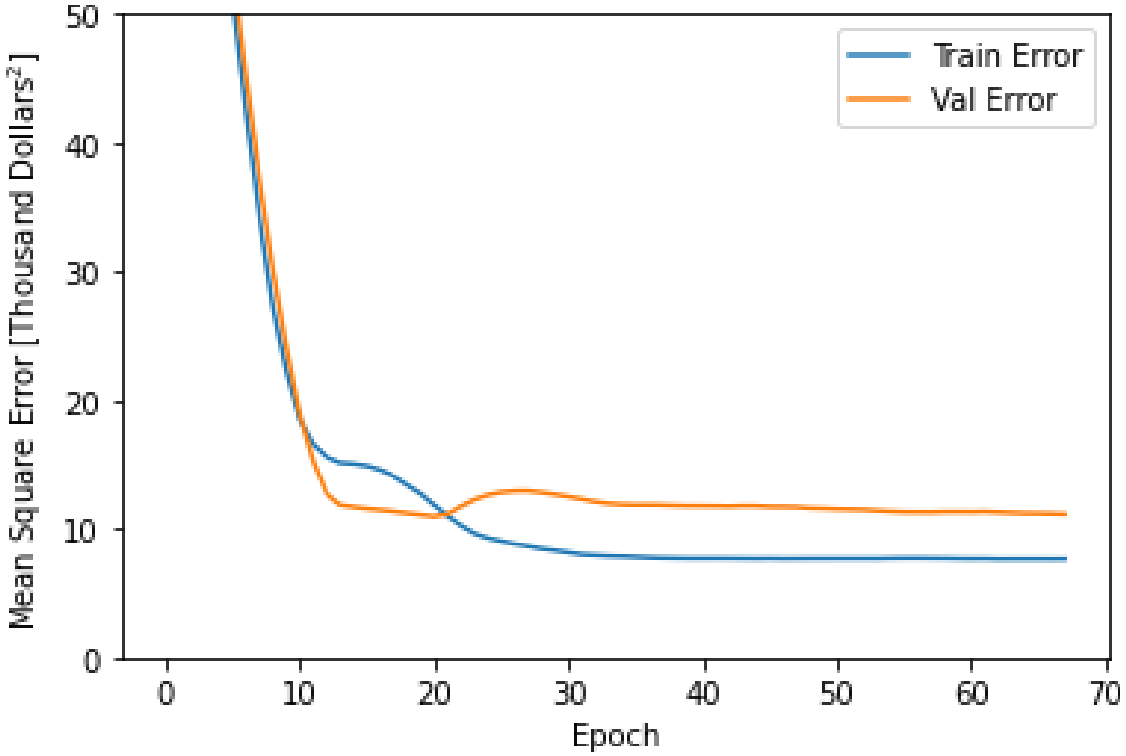
```
.....
.....
.....
.....
.....
.....
.....
.....
.....
.....
Final Root Mean Square Error on validation set: 2.86
```

Wizualizacja loss:



Doszło do overfittingu, pomimo, że zmniejszyłam ilość warstw i neuronów.

Zmienię funkcję optymalizacji na adam:



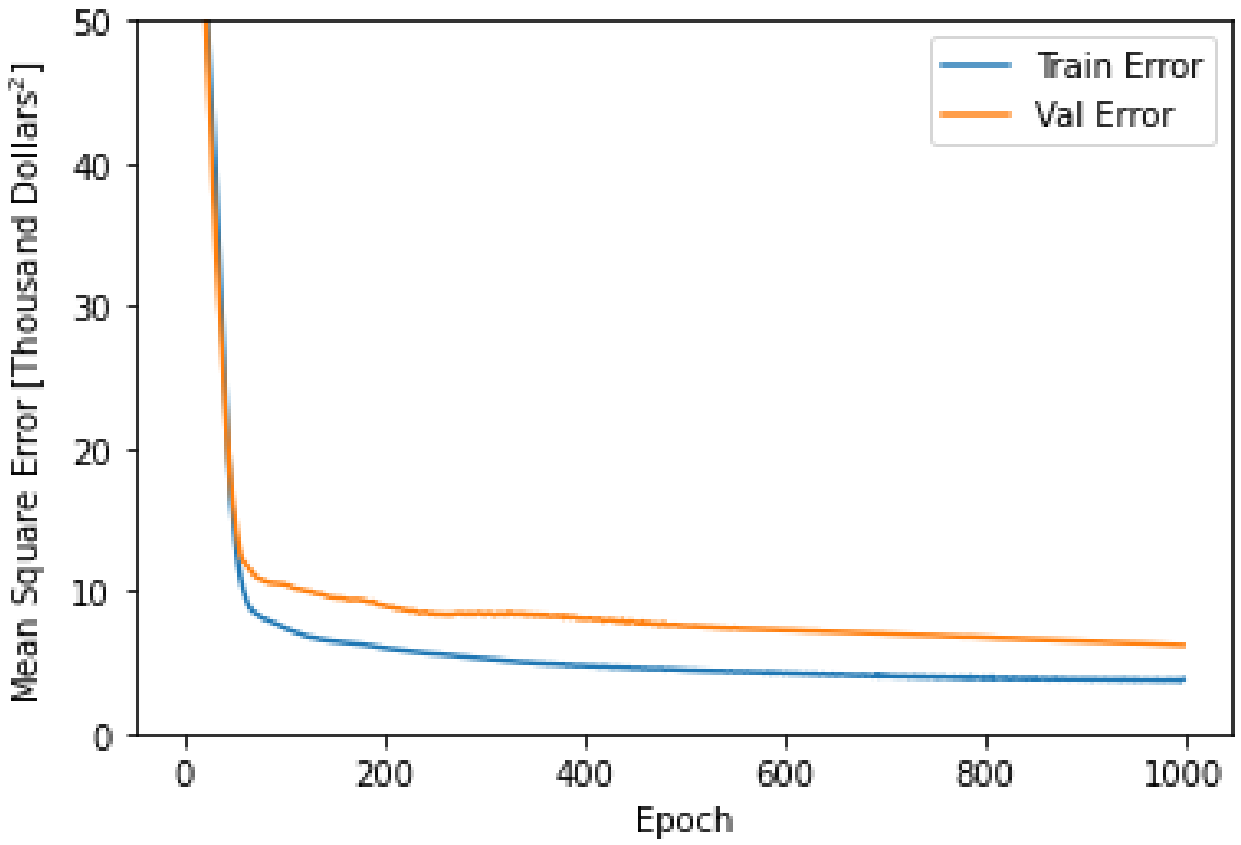
Final Root Mean Square Error on validation set: 3.337

Zdecydowana poprawa, teraz dodam ilość epok i odkomentuję warstw.

No niestety znowu doszło do overfittingu, lecz przy zachowanej ilości warstw i neuronów, a epochs=1000 i optimizer=’adam’ uzyskałam świetne rezultaty. Bardzo niski błąd:

Final Root Mean Square Error on validation set: 2.488

I brak overfittingu:



2) Sprawdzenie modelu na zbiorze testowym już nie pała takim optymizmem:

```
x_test_norm = (x_test - train_mean) / train_std
mse, score, _ = model.evaluate(x_test_norm, y_test)
rmse = np.sqrt(mse)
print('Root Mean Square Error on test set: {}'.format(round(rmse, 3)))
print(score)
```

```
4/4 [==============================] - 0s 2ms/step - loss: 31.8926 - mae: 4.3649 - mse: 31.8926
Root Mean Square Error on test set: 5.647
4.364914894104004
```

3) Czy zastosowanie K-składowej walidacji krzyżowej poprawia proces doboru wartości hiperparametrów?

Nie ma poprawy:

```
k = 4
num_val_samples = len(x_train) // k
num_epochs = 100
all_scores = []

for i in range(k):
    print(f'Processing fold # {i}')
    val_data = x_train[i * num_val_samples: (i+1) * num_val_samples]
    val_targets = y_train[i * num_val_samples: (i+1) * num_val_samples]

    partial_train_data = np.concatenate(
                            [x_train[:i * num_val_samples],
                            x_train[(i+1) * num_val_samples:]],
                            axis=0)
    partial_train_targets = np.concatenate(
                            [y_train[:i * num_val_samples],
                            y_train[(i+1)*num_val_samples:]],
                            axis=0)
    model = build_model()
    model.fit(partial_train_data,
              partial_train_targets,
              epochs=num_epochs,
              verbose=0)
    val_mse, _, _ = model.evaluate(val_data, val_targets)
    all_scores.append(hist['val_mae'])
```

```
Processing fold # 0
4/4 [==============================] - 0s 3ms/step - loss: 26.9341 - mae: 3.5551 - mse: 26.9341
Processing fold # 1
4/4 [==============================] - 0s 3ms/step - loss: 24.4590 - mae: 3.7403 - mse: 24.4590
Processing fold # 2
4/4 [==============================] - 0s 3ms/step - loss: 30.3019 - mae: 3.9647 - mse: 30.3019
Processing fold # 3
4/4 [==============================] - 0s 3ms/step - loss: 39.1068 - mae: 4.4166 - mse: 39.1068
```

Dodatkowe wnioski: Pomimo, że posłużyłam się RMSE, aby ukazać błędy predykcji i jest to dość miarodajne tutaj to jednak warto pamiętać, że dla zbioru z odstającymi danymi wynik może być wyolbrzymiony. RMSE jest na to wrażliwe, bo wpływ każdego błędu na RMSE jest wprost proporcjonalne do kwadratu błędu.

Podobno w rzeczywistości i tak używa się kilku metryk na oszacowanie regresji, więc nie ma co się nadto przejmować powyższą uwagą.

Zad.5

Pobierz zbiór Iris (https://www.tensorflow.org/datasets/catalog/iris?hl=en) z modułu TensorFlow Datasets. Dobierz najlepsze wartości hiperparametrów sieci na podstawie:

[na ocenę 3.0]: K-składowej walidacji krzyżowej.

K-składowa walidacja krzyżowa. Najpierw ze zbioru danych wydzielany jest zbiór testowy. Następnie pozostałe dane są dzielone na K składowych. Każda z tych składowych jest następnie zbiorem walidacyjnym, a pozostałe K-1 składowych składa się na zbiór

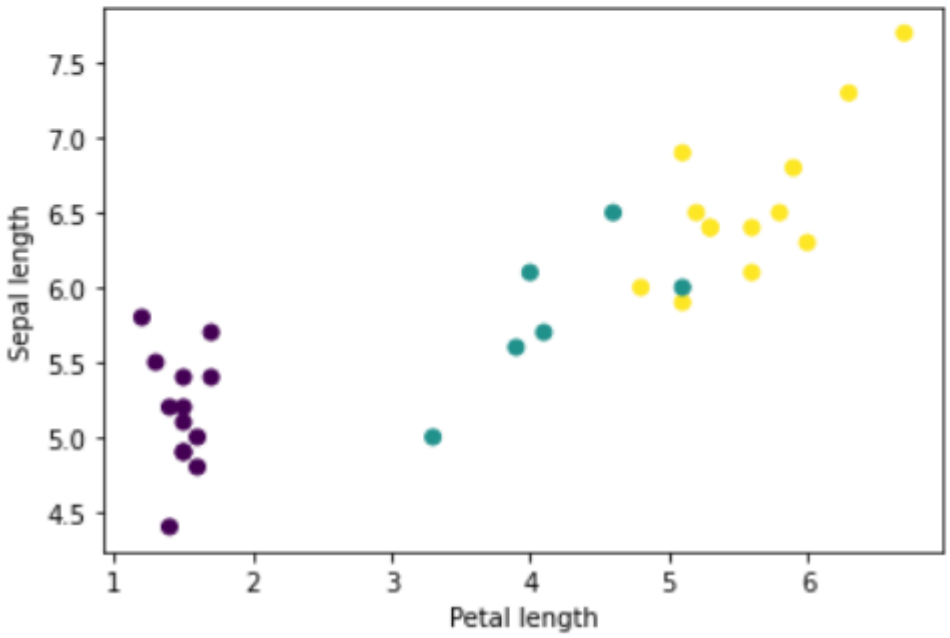
treningowy. Walidacja następuje na podstawie średnich danych walidacyjnych z K treningów. Na podstawie walidacji wybieramy najlepsze wartości hiperparametrów, a następnie uczymy sieć na całym zbiorze danych oprócz części testowej. Ocena wyuczonej sieci następuje na zbiorze testowym.

W sprawozdaniu zamieść i opisz:

- wykorzystany kod,
- sposób doboru hiperparametrów,
- wykresy funkcji straty i trafności dla walidacji,
- wyniki uzyskane dla najlepszej kombinacji parametrów oraz rysunek finalnej sieci wykonany przy wykorzystaniu keras.utils.plot_model,
- Wnioski.

Do wizualizacji tych danych warto użyć scatter plot. Funkcja ta ukaże rozkład danych i ich relacje:

```
#Wizualizacja danych, klastry
plt.scatter(features['petal_length'],
            features['sepal_length'],
            c=labels,
            cmap='viridis')

plt.xlabel("Petal length")
plt.ylabel("Sepal length")
plt.show()
```



Teraz dane wystarczy podzielić na wejścia wyjścia:

```
#podzial na wej i wyj
X=df.iloc[:,0:4].values
Y=df.iloc[:,4:5].values
```

Dane ze zbioru iris z tensorflow są już znormalizowane. Należy jednak pamiętać o kodowaniu labels, po pierwsze ze słownych danych do danych matematycznych – tutaj już tak jest- a następnie najlepiej do one hot encode.

One hot różni się tym od innego kodowania, że długość tego wektora równa jest ilości klas do określenia, bo w miejscu danej kategorii mamy 1, a w całej reszcie zera. Dzięki takiemu kodowaniu danych nasz algorytm będzie lepiej działać w kontekście przewidywania.

```
from keras.utils import np_utils
from sklearn.preprocessing import LabelEncoder
import numpy as np

y = np.ravel(Y)
# Encode classes as integers
encoder = LabelEncoder()
encoder.fit(y)
encoded_Y = encoder.transform(y)

# One hot encode integer labels
one_hot_Y = np_utils.to_categorical(encoded_Y)
```

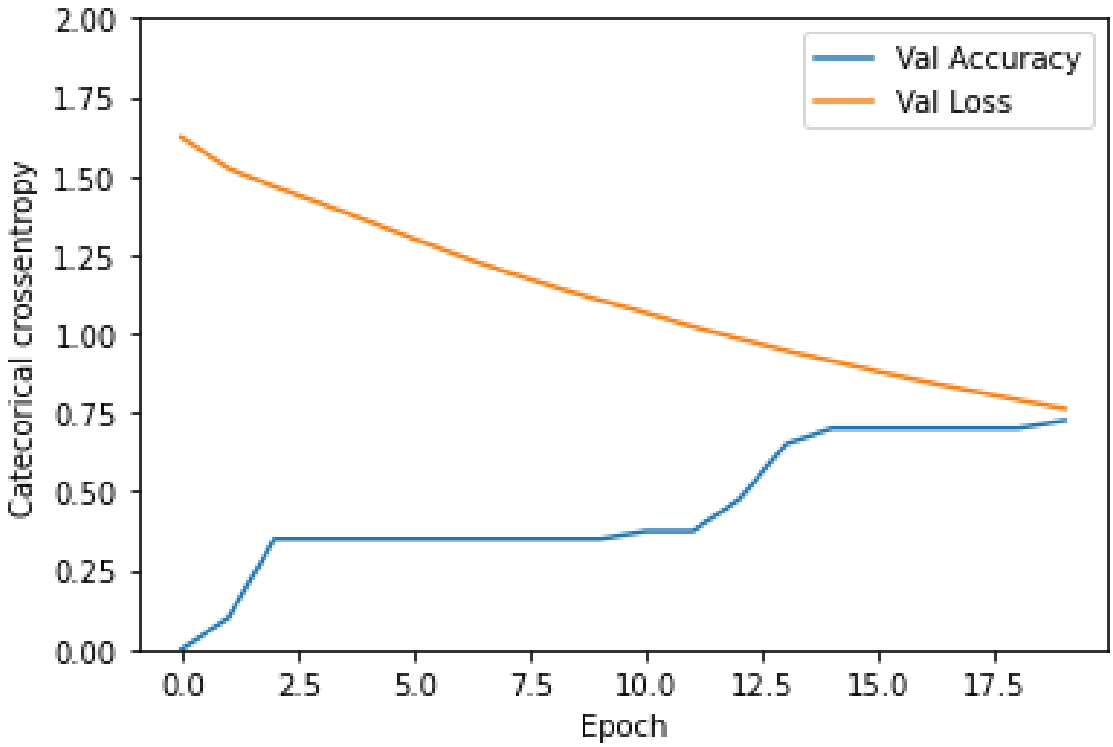
Model będziemy wywoływać w pętli, dlatego zaprogramowałam go zdefiniowałam oddzielnie:

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential

def create_model():
  model = Sequential()
  model.add(Dense(32, input_shape=(4,)))
  model.add(Dense(32, activation = 'relu'))
  model.add(Dense(3, activation = 'softmax'))

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

  return model
```

Hiperparametry dobierałam w oparciu o wiedzę teoretyczną, jak i manualnie opierając się o uzyskiwane wyniki z K-fold cross validation.

- Funkcję aktywacyjną wybrałam relu, aby pozbyć się wartości ujemnych, a następnie softmax - znormalizowana funkcja wykładnicza, najczęściej stosowana do klasyfikacji wielowymiarowej.
- Liczba neuronów w ostatniej warstwie oczywiście wynosi 3, tyle ile przewidujemy klas do wyszczególnienia.
- W powyższych warstwach ilość neuronów wynosi tyle ile zasugerowałam się z wyników k-fold cross validation. Tzn. Nie za dużo, aby nie doprowadzić do overfittingu, ani nie za mało, aby model był efektywny i najlepiej jego accuracy było około 0.9.

Wykresy funkcji straty i trafności dla walidacji:



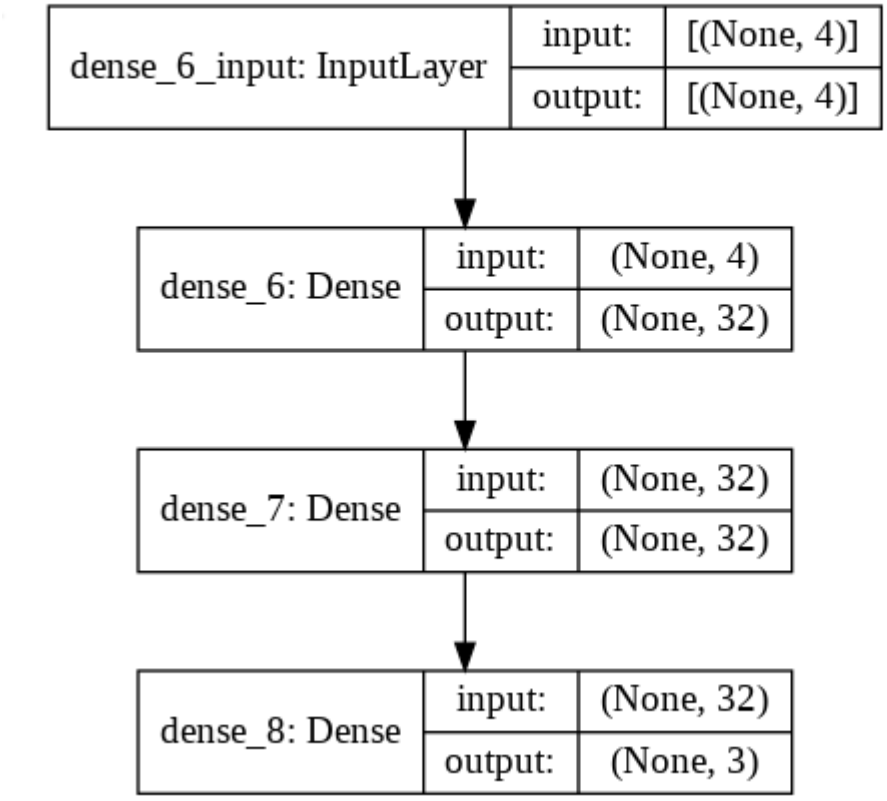
Wyniki uzyskane dla finalnej sieci:

```
hist = pd.DataFrame(history.history)
print(hist)
hist['epoch'] = history.epoch
```

```
        loss  accuracy  val_loss  val_accuracy
0   1.823387    0.0000  1.622671         0.000
1   1.639541    0.0000  1.524101         0.100
2   1.551514    0.2375  1.465351         0.350
3   1.493246    0.3500  1.413116         0.350
4   1.438758    0.3500  1.357222         0.350
5   1.374286    0.3500  1.301039         0.350
6   1.317081    0.3500  1.245505         0.350
7   1.255898    0.3500  1.194836         0.350
8   1.201226    0.3500  1.149601         0.350
9   1.155920    0.3500  1.106575         0.350
10  1.106637    0.3500  1.065807         0.375
11  1.064507    0.3500  1.023254         0.375
12  1.015945    0.4000  0.983756         0.475
13  0.973994    0.4875  0.946947         0.650
14  0.937038    0.6250  0.913540         0.700
15  0.899694    0.7000  0.879633         0.700
16  0.865403    0.7000  0.848422         0.700
17  0.829917    0.7000  0.819319         0.700
18  0.804831    0.7000  0.790182         0.700
19  0.765236    0.7000  0.762716         0.725
```

Rysunek finalnej sieci:

```
] dot_img_file = '/tmp/model_1.png'
  tf.keras.utils.plot_model(model, to_file=dot_img_file, show_shapes=True)
```



Finalna wartość trafności (accuracy):

```
# Build the oos prediction list and calculate the error.
oos_y = np.concatenate(oos_y)
oos_pred = np.concatenate(oos_pred)
#oos_y_compare = np.argmax(oos_y,axis=1) # For accuracy calculation

score = metrics.accuracy_score(oos_y, oos_pred)
print(f"Final score (accuracy): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat( [df, oos_y, oos_pred],axis=1 )
```

```
Final score (accuracy): 0.8166666666666667
```

Wnioski:

Dzięki K-fold cross validation jestem w stanie uzyskać predictions na podstawie całego zbioru danych.

Analizując cały kod i jego wyniki można zaobserwować, że udało mi się osiągnąć całkiem pozytywne wyniki trafności. Wnioskować z tego można, że dla takiego zbioru jak iris wystarczy 20 epok i podział na 3 zestawy.

Im więcej zestawów tym dłuższy czas obliczania, stąd wartość 3 jest bardzo optymalna.

Zadanie 5. b)

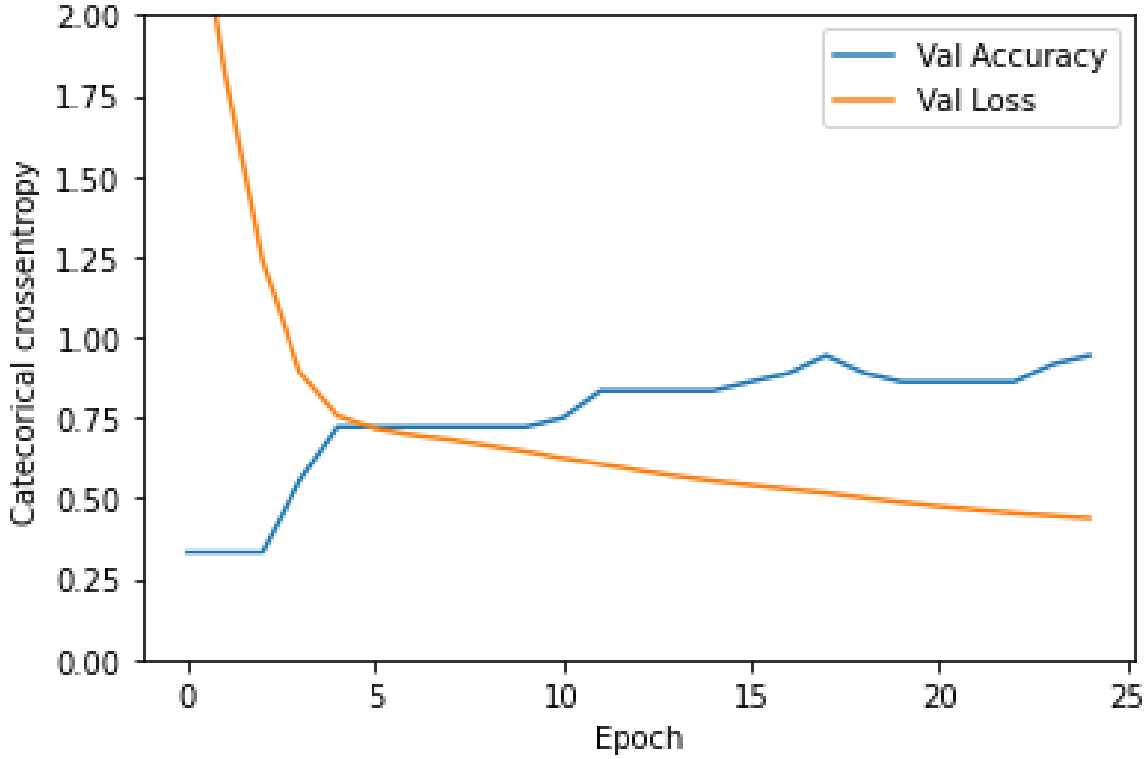
K-składowa walidacja krzyżowa ze zbiorem walidacyjnym i testowym. Dane są dzielone na K składowych. Każda z tych składowych jest następnie zbiorem testowym, a z pozostałych K-1 składowych każda staje się zbiorem walidacyjnym, a pozostałe K-2 składowych składa się na zbiór treningowy. Walidacja następuje na podstawie średnich danych walidacyjnych z K*(K-1) treningów. Na podstawie walidacji wybieramy najlepsze wartości hiperparametrów, a następnie uczymy sieć K-krotnie – za każdym razem na wszystkich K-1 składowych, które nie są w danym przebiegu zbiorem testowym. Ocena wyuczonej sieci jest średnim wynikiem dla K zbiorów testowych.

Kod w zasadzie jest bardzo zbliżony do tego z podpunktu a). Głónie różni się on częścią odpowiedzialną za podział zbioru iris:

```
from sklearn.model_selection import train_test_split
# Keep a 10% holdout
x_main, x_holdout, y_main, y_holdout = train_test_split(
    X, Y, test_size=0.10)
```

Początkowo wartość accuracy była w okolicach 0.7. Chciałam polepszyć wynik zwiększając liczbę epok do 30. Zdecydowanie o za dużo, ponieważ to spowodowało overfitting po 25 epoce. Ustawiłam 25 epok i oto rezultaty:

```
Fold score (accuracy): 0.9444444444444444
2/2 [==============================] - 0s 3ms/step - loss: 0.4394 - accuracy: 0.9444
Model evaluation  [0.43944770097732544, 0.9444444179534912]
```



Wyniki dla zbioru testowego:

```
score = metrics.accuracy_score(y_holdout, holdout_pred)
print(f"Final score (accuracy): {score}")

# Write the cross-validated prediction
oos_y = pd.DataFrame(oos_y)
oos_pred = pd.DataFrame(oos_pred)
oosDF = pd.concat( [df, oos_y, oos_pred],axis=1 )
```

```
Final score (accuracy): 0.8333333333333334
```

Wnioski:

Podpunkt b był dla mnie zdecydowanie łatwiejszym zadaniem, gdy już nauczyłam się implementacji k-fold cross-validation dla problemu klasyfikacji i zrozumiałam jakie dają one możliwości.

Wnioski niczym się nie różnią, poza tym, że udało mi się poprawić rezultaty i zarazem uniknąć overfittingu.

Zadanie. 7

Problem: klasyfikacja jednoetykietowa (dyskretna) dla zbioru pochodzącego z Kaggle lub UCI Machine Learning Repository.

Trochę teorii. Czym jest regresja liniowa?

Regresja liniowa dopasowuje model liniowy ze współczynnikami w = (w1,…, wp) w celu zminimalizowania resztkowej sumy kwadratów między obserwowanymi celami w zbiorze danych a celami przewidywanymi przez przybliżenie liniowe.

Ja wybrałam zbiór: Graduate Admission 2

https://www.kaggle.com/mohansacharya/graduate-admissions

Zaczęłam od wizualizacji i analizy danych. Do tego użyłam funkcji: info, describe i corr oraz biblioteki seaborn, która świetnie sprawdza się przy tworzeniu przejrzystych grafów:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 500 entries, 0 to 499
Data columns (total 9 columns):
 #   Column             Non-Null Count  Dtype  
---  ------             --------------  -----  
 0   Serial No.         500 non-null    int64  
 1   GRE Score          500 non-null    int64  
 2   TOEFL Score        500 non-null    int64  
 3   University Rating  500 non-null    int64  
 4   SOP                500 non-null    float64
 5   LOR                500 non-null    float64
 6   CGPA               500 non-null    float64
 7   Research           500 non-null    int64  
 8   Chance of Admit    500 non-null    float64
dtypes: float64(4), int64(5)
memory usage: 35.3 KB
```

Rys.1 Wynik dla data.info()

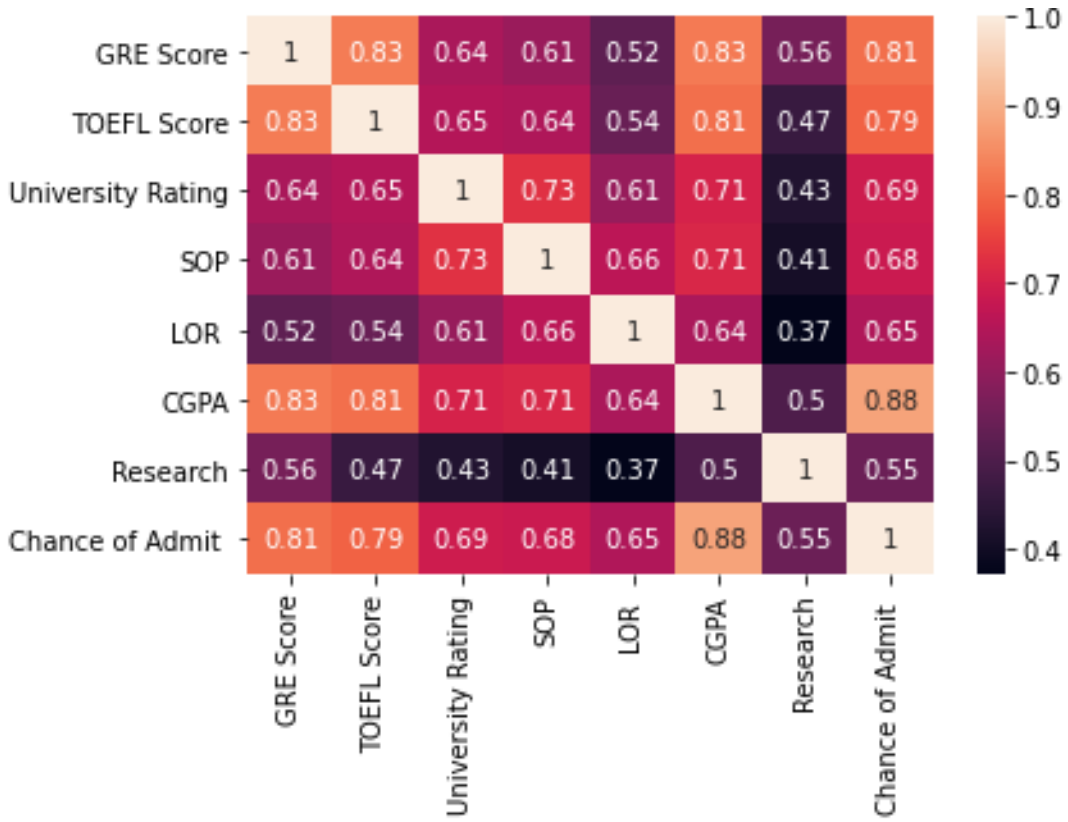
Usunęłam nic nie wnoszącą do modelu kolumnę “Serial No.”.

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
count	500.000000	500.000000	500.000000	500.000000	500.00000	500.000000	500.000000	500.00000
mean	316.472000	107.192000	3.114000	3.374000	3.48400	8.576440	0.560000	0.72174
std	11.295148	6.081868	1.143512	0.991004	0.92545	0.604813	0.496884	0.14114
min	290.000000	92.000000	1.000000	1.000000	1.00000	6.800000	0.000000	0.34000
25%	308.000000	103.000000	2.000000	2.500000	3.00000	8.127500	0.000000	0.63000
50%	317.000000	107.000000	3.000000	3.500000	3.50000	8.560000	1.000000	0.72000
75%	325.000000	112.000000	4.000000	4.000000	4.00000	9.040000	1.000000	0.82000
max	340.000000	120.000000	5.000000	5.000000	5.00000	9.920000	1.000000	0.97000

Rys.2 Wynik dla **data.describe()**

	GRE Score	TOEFL Score	University Rating	SOP	LOR	CGPA	Research	Chance of Admit
GRE Score	1.000000	0.827200	0.635376	0.613498	0.524679	0.825878	0.563398	0.810351
TOEFL Score	0.827200	1.000000	0.649799	0.644410	0.541563	0.810574	0.467012	0.792228
University Rating	0.635376	0.649799	1.000000	0.728024	0.608651	0.705254	0.427047	0.690132
SOP	0.613498	0.644410	0.728024	1.000000	0.663707	0.712154	0.408116	0.684137
LOR	0.524679	0.541563	0.608651	0.663707	1.000000	0.637469	0.372526	0.645365
CGPA	0.825878	0.810574	0.705254	0.712154	0.637469	1.000000	0.501311	0.882413
Research	0.563398	0.467012	0.427047	0.408116	0.372526	0.501311	1.000000	0.545871
Chance of Admit	0.810351	0.792228	0.690132	0.684137	0.645365	0.882413	0.545871	1.000000

Rys.3 Wynik dla **data.corr()**



Rys.4 Wynik dla **sns.heatmap(data.corr(), annot=True)**

Porównując powyższy graf, a wyniki dla data.corr można zdecydowanie przyznać, że przejrzystość pozwala na szybsze i trafniejsze wnioski. Wnioski które ja wyciągnęłam są takie, że parametry 'CGPA','GRE Score','TOEFL Score' są mocno powiązane z zależną zmienną ‘Chance of Admit’.

Mogłabym użyć wszystkich dostępnych danych do modelu regresji, czyli tu do przewidzenia szans na dostanie się do uczelni, ale bardziej efektywne jest to co zrobiłam poniżej. Najpierw analizuję sama dane i oceniam co ma znaczący wpływ na wartość będącą naszym szukanym y. Oczywiście mogłabym szukać czegoś innego. Wszystko jest kwestią zadanego pytania. Następnie dobieram tylko te najistotniejsze, najprzydatniejsze dane. Zmniejsza to zapotrzebowanie na zasoby, które musiałyby analizować cały zestaw danych.

Dodatkowo jeszcze zwizualizowałam sobie zależność każdej z tych kolumn osobmno z ‘Chance of Admin’. Do ich obejrzenia zapraszam do kodu. Nie chcę niepotrzebnie wydłużać sprawozdania.

Poniższy kod przedstawia wyłuskanie potrzebnych danych, podzielenie ich na odpowiednie zestawy dzięki funkcji **train_test_split().** Następnie wyknetnanie modelu dokonującego regresji liniowej, uczenie go i wreszcie przetestowanie za pomocą predykcji na zbiorze testowym.

```
x = data[['CGPA','GRE Score','TOEFL Score']]
```

```
y = data[['Chance of Admit ']]
```

```
# Split data for test and train the model.
x_train, x_test, y_train, y_test = train_test_split(x,y,random_state=0,test_size=.20)
```

```
# x shape
print(x.shape)
print(x_train.shape)
```

```
(500, 3)
(400, 3)
```

```
#I could do this like 4th task (Boston Housing) but I try new way
#object
linreg = LinearRegression()
# fiting our data for training
linreg.fit(x_train,y_train)
```

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

```
# our model is ready to predict y.
y_predict = linreg.predict(x_test)
# our model prediction
y_predict[:10]
```

```
array([[0.62802152],
       [0.83209435],
       [0.79119818],
       [0.85046086],
       [0.60222891],
       [0.66888863],
       [0.55297329],
       [0.68293116],
       [0.55296602],
       [0.7583334 ]])
```

```
#Compare y_test & y_predict to find model acc
# y test
y_test[:10]
```

	Chance of Admit
90	0.64
254	0.85
283	0.80
445	0.91
461	0.68
15	0.54
316	0.54
489	0.65
159	0.52
153	0.79

```
metrics.mean_absolute_error(y_test,y_predict)
#Świetnie! My model is in 96% correct
```

```
0.04688621359385711
```

Rys.5 i 6 Przedstawienie wyniku średniego błędu bezwzględnego dla stworzonego modelu.

MAE - średni błąd bezwzględny - to średnia z sumy różnic pomiędzy wartością rzeczywistą, a bezwzgędną (dodatnich) wartością przewidywaną. Minusem tej metryki w analizowaniu jakości modelu jest fakt, że nie uwzględnia ona wartości ujemnych i nie jest miarodajna do bardzo dużych błędów jak MSE. Za to RMSE już tak i logarytmiczny jej odpowiednik.

Omówienie gotowej funkcji **LinearRegresion**():

Zgodnie z dokumentacją biblioteki **sklearn.linear_model** funkcja ta rozwiązuje problem regresji zgodnie z definicją, którą przytoczyłam na samym początku opisu tego zadania. Warto zaznaczyć, że możliwe jest wykorzystanie następujących metod:

fit(X, y[, sample_weight])	Fit linear model.
get_params([deep])	Get parameters for this estimator.
predict(X)	Predict using the linear model.
score(X, y[, sample_weight])	Return the coefficient of determination R^2 of the prediction.
set_params(**params)	Set the parameters of this estimator.

Wnioski:

Uzyskany wynik świadczy o 96% skuteczności w przewidywaniu trafnych wyników dla nowych, nieznanych mu zbiorów danych. Jest to naprawdę świetny rezultat przy niskim nakładzie pracy. Warto było wykorzystać gotową funkcję jako zamiast samemu tworzyć głęboką sieć neuronową, skoro od razu otrzymaliśmy tak pozytywny wynik. W przeciwnym razie rozwiązałabym powyższy problem bardziej manualnie albo jeszcze jest możliwość przekazanie estymatora z konkretnymi parametrami, które chcemy ustawić dla naszego modelu.

Zadanie 6. Znajdź zbiór danych odpowiedni do zadania klasyfikacji jednoetykietowej (dyskretnej). Zbiór może pochodzić z Kaggle (https://www.kaggle.com/datasets) lub UCI Machine Learning Repository (https://archive.ics.uci.edu/ml/datasets.php) lub z innego źródła (po uzgodnieniu z prowadzącym). Zastanów się jak wykorzystać sieć neuronową do rozwiązania tego zadania klasyfikacji. Zaproponuj strukturę sieci (rozważ także wykorzystanie regularyzacji i porzucania) a następnie eksperymentalnie dobierz wartości hiperparametrów sieci maksymalizujące efektywność sieci na zbiorze testowym.

Subiektywnie znalazłam bardzo ciekawy zbiór https://www.kaggle.com/iarunava/cell-images-for-detecting-malaria dotyczy on komórek zarodźca malarycznego w komórkach człowieka. Jest on dla mnie ciekawy z kilku powodów, po pierwsze pokazuje, jak programiści mogą wesprzeć najistotniejszą gałąź gospodarki czy społeczeństw, czyli medycynę odpowiedzialną za ludzkie zdrowie. Po drugie, dane te składają się z 2 zbiorów zdjęć, łącznie o wielkości 675 MB i 27,558 tysięcy zdjęć i idealnie pasują do zadania klasyfikacji jednoetykietowej. Klasyfikacja ta opiera się na detekcji zdjęć, odpowiadając na pytanie, czy komórki na nich zawarte zawierają zarodziec Malarii, czy nie. Po trzecie, które już jest w 100% subiektywne, łączy się z moim wyborem profilu biologiczno-chemicznego w LO.

Wykonałam to zadanie jako ostatnie, dlatego można zobaczyć tutaj zastosowanie sieci konwolucyjnej, czyli głęboka sieć poprzedzona zabiegami filtrowania i poolingu. Mam nadzieję, że będzie to odczytane to im plus.

Przeciwności jakie napotkałam to pobranie poszczególnych zdjęć z folderów umieszczonych na gdrive. W zadaniu 7 poradziłam sobie świetnie z połączeniem gdrive z colab, tutaj niestety po 3h walk odpóściłam. Rozwiązaniem tej przeszkody okazał się dostępny notatnik w kaggle.com. Ich sesje są ograniczone czasowo, lecz praca przebiega równie gładko.

Dane są już zawarte w folderze input, który jest automatycznie dołączony do mojego notatnika, gdy otwieram go z konkretnej bazy danych. Oto implementacja jej załadowania i przygotowania do przydatności w sieci neuronowej, m.in. normalizacja:

```
data=[]
labels=[]
Parasitized=os.listdir("../input/cell-images-for-detecting-malaria/cell_images/Parasitized/")
for a in Parasitized:
    try:
        image=cv2.imread("../input/cell-images-for-detecting-malaria/cell_images/Parasitized/"+a)
        image_from_array = Image.fromarray(image, 'RGB')
        size_image = image_from_array.resize((50, 50))
        data.append(np.array(size_image))
        labels.append(0)
    except AttributeError:
        print("")
```

```
Uninfected=os.listdir("../input/cell-images-for-detecting-malaria/cell_images/Uninfected/")
for b in Uninfected:
    try:
        image=cv2.imread("../input/cell-images-for-detecting-malaria/cell_images/Uninfected/"+b)
        image_from_array = Image.fromarray(image, 'RGB')
        size_image = image_from_array.resize((50, 50))
        data.append(np.array(size_image))
        labels.append(1)
    except AttributeError:
        print("")
```

Rys.1 Zaciągnięcie danych i ich wstępne przygotowanie.

```
Cells=np.array(data)
labels=np.array(labels)
```

```
np.save("Cells",Cells)
np.save("labels",labels)
```

+ Code + Markdown

```
s=np.arange(Cells.shape[0])
np.random.shuffle(s)
Cells=Cells[s]
labels=labels[s]
```

```
num_classes=len(np.unique(labels))
len_data=len(Cells)
```

```
(x_train,x_test)=Cells[(int)(0.1*len_data):],Cells[:(int)(0.1*len_data)]
x_train = x_train.astype('float32')/255 # As I'm working on image data we are normalizing data by divinding 255.
x_test = x_test.astype('float32')/255
train_len=len(x_train)
test_len=len(x_test)
```

```
(y_train,y_test)=labels[(int)(0.1*len_data):],labels[:(int)(0.1*len_data)]
```

```
#Doing One hot encoding as classifier has multiple classes
y_train=keras.utils.to_categorical(y_train,num_classes)
y_test=keras.utils.to_categorical(y_test,num_classes)
```

Rys. 2 i 3 Normalizacja i one-hot-encoding

Na powyższych rysunkach można również zauważyć, że zapisałam przygotowane odpowiednio zdęcia w zmiennych Cells i labels, aby o wiele szybciej je załadowywać w kolejnych krokach implementacji sieci klasyfikacji.

Dane zostały połączone z dwóch zbiorów, które były równo oddzielone na dwa foldery, aby teraz wprowadzić randomowość tego jakie dane będą w zbiorze testowym, a jakie w trenującym, wskazane jest poddanie ich przemieszaniu. Odpowiedzialna jest za to funkcja shuffle() z biblioteki numpy i klasy random.

Model składa się z wielu warstw. Na początek poddaje zdjęcia 3 filtrom, które dzięki poolingowi są uwypuklane (najprościej tłumacząc). Następnie warstwa Flatten() powoduje, że ze zdjęć będących macierzą 50x50x3 powstaje 1x2304, czyli wektor. Oczywiście przemnażając 50x50x3 nie uzyskamy 2304, lecz poniższe podsumowanie wyjaśnia skąd bierze się ta wartość.

```python
#creating sequential model
model=Sequential()
model.add(Conv2D(filters=16,kernel_size=2,padding="same",activation="relu",input_shape=(50,50,3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32,kernel_size=2,padding="same",activation="relu"))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64,kernel_size=2,padding="same",activation="relu"))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(500,activation="relu"))
model.add(Dropout(0.2))
model.add(Dense(2,activation="softmax"))#2 represent output layer neurons
model.summary()
```

Rys. 4 Implementacja finalnego modelu, z najlepszymi hiperparametrami.

Pozwoliłam sobie na tę postać wizualizacji sieci neuronowej, gdyż wydaje mi się ona rozbudowana, przejrzysta i pełna wartościowych informacji. Wiemy co się dzieje na każdym etapie trenowania.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 50, 50, 16)        208
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 25, 25, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 25, 25, 32)        2080
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 12, 12, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 12, 64)        8256
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0
_________________________________________________________________
dropout (Dropout)            (None, 6, 6, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 2304)              0
_________________________________________________________________
dense (Dense)                (None, 500)               1152500
_________________________________________________________________
dropout_1 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 1002
=================================================================
Total params: 1,164,046
Trainable params: 1,164,046
Non-trainable params: 0
_________________________________________________________________
```

Rys. 5 Podsumowanie wyżej zaimplementowanego modelu.

```
# compile the model with loss as categorical_crossentropy and using adam optimizer you can test result by trying RMSProp
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Rys. 6 Kod kompilacyny wyżej stworzonego modelu.

```
history = model.fit(x_train,y_train,batch_size=50,epochs=20,verbose=1,validation_split = 0.1)
```

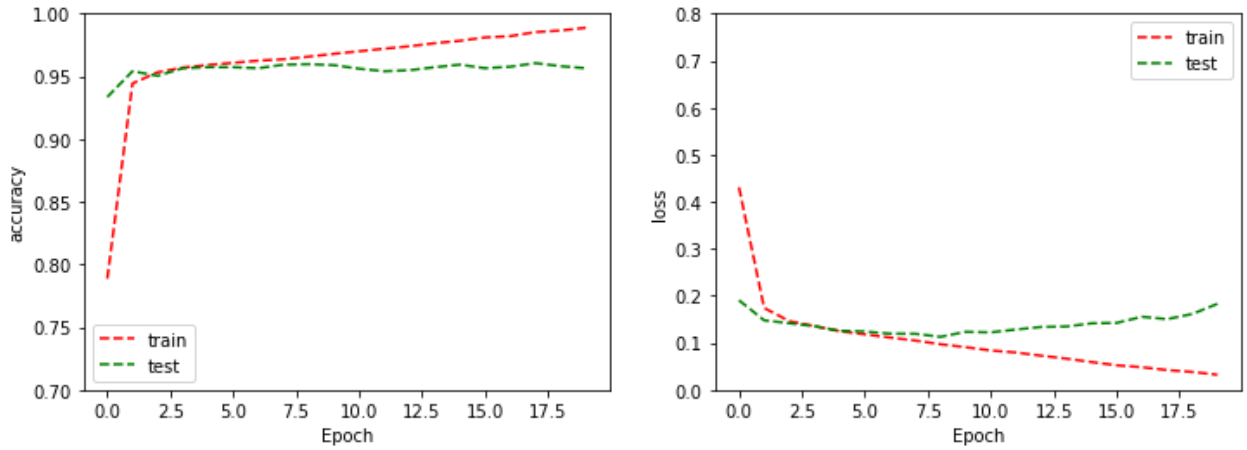
```
Epoch 1/20
447/447 [==============================] - 24s 53ms/step - loss: 0.4307 - accuracy: 0.7887 - val_loss: 0.1905 - val_accuracy: 0.9335
Epoch 2/20
447/447 [==============================] - 24s 53ms/step - loss: 0.1741 - accuracy: 0.9443 - val_loss: 0.1478 - val_accuracy: 0.9541
Epoch 3/20
447/447 [==============================] - 23s 52ms/step - loss: 0.1461 - accuracy: 0.9532 - val_loss: 0.1416 - val_accuracy: 0.9504
Epoch 4/20
447/447 [==============================] - 24s 54ms/step - loss: 0.1356 - accuracy: 0.9570 - val_loss: 0.1353 - val_accuracy: 0.9565
Epoch 5/20
447/447 [==============================] - 23s 52ms/step - loss: 0.1247 - accuracy: 0.9591 - val_loss: 0.1255 - val_accuracy: 0.9573
Epoch 6/20
447/447 [==============================] - 23s 52ms/step - loss: 0.1180 - accuracy: 0.9608 - val_loss: 0.1240 - val_accuracy: 0.9573
Epoch 7/20
447/447 [==============================] - 24s 53ms/step - loss: 0.1112 - accuracy: 0.9625 - val_loss: 0.1195 - val_accuracy: 0.9565
Epoch 8/20
447/447 [==============================] - 23s 52ms/step - loss: 0.1048 - accuracy: 0.9636 - val_loss: 0.1192 - val_accuracy: 0.9593
Epoch 9/20
447/447 [==============================] - 24s 53ms/step - loss: 0.0970 - accuracy: 0.9657 - val_loss: 0.1128 - val_accuracy: 0.9597
Epoch 10/20
447/447 [==============================] - 24s 55ms/step - loss: 0.0907 - accuracy: 0.9679 - val_loss: 0.1234 - val_accuracy: 0.9589
Epoch 11/20
447/447 [==============================] - 24s 54ms/step - loss: 0.0837 - accuracy: 0.9700 - val_loss: 0.1223 - val_accuracy: 0.9561
Epoch 12/20
447/447 [==============================] - 24s 53ms/step - loss: 0.0794 - accuracy: 0.9720 - val_loss: 0.1283 - val_accuracy: 0.9541
Epoch 13/20
447/447 [==============================] - 23s 52ms/step - loss: 0.0727 - accuracy: 0.9740 - val_loss: 0.1338 - val_accuracy: 0.9549
Epoch 14/20
447/447 [==============================] - 24s 54ms/step - loss: 0.0664 - accuracy: 0.9764 - val_loss: 0.1347 - val_accuracy: 0.9573
Epoch 15/20
447/447 [==============================] - 23s 52ms/step - loss: 0.0591 - accuracy: 0.9783 - val_loss: 0.1415 - val_accuracy: 0.9593
Epoch 16/20
447/447 [==============================] - 23s 52ms/step - loss: 0.0522 - accuracy: 0.9810 - val_loss: 0.1424 - val_accuracy: 0.9565
Epoch 17/20
447/447 [==============================] - 24s 54ms/step - loss: 0.0481 - accuracy: 0.9820 - val_loss: 0.1554 - val_accuracy: 0.9577
Epoch 18/20
447/447 [==============================] - 23s 52ms/step - loss: 0.0421 - accuracy: 0.9853 - val_loss: 0.1505 - val_accuracy: 0.9605
Epoch 19/20
447/447 [==============================] - 24s 53ms/step - loss: 0.0381 - accuracy: 0.9865 - val_loss: 0.1613 - val_accuracy: 0.9581
Epoch 20/20
447/447 [==============================] - 23s 52ms/step - loss: 0.0321 - accuracy: 0.9887 - val_loss: 0.1830 - val_accuracy: 0.9565
```

Rys.7 i 8 Kod i wyniki dopasowywania, czyli trenowanie modelu.

```
accuracy = model.evaluate(x_test, y_test, verbose=1)
print('\n', 'Test_Accuracy:-', accuracy[1])
```

```
87/87 [==============================] - 1s 13ms/step - loss: 0.1749 - accuracy: 0.9568

 Test_Accuracy:- 0.9568058252334595
```

Rys. 9 Wynik trafności modelu na zestawie testowym.



Rys.10 Wizualizacja funkcji straty i trafności dla zestawu walidacyjnego.

Powyższa wizualizacja ukazuje nam overfitting dopiero po wartości 5.0 i to tylko dla zestawu uczącego się.

Jako ostatnie zadanie, które wykonałam, sądzę, że widać progres osiągnięty od zadania pierwszego. Trafność na poziomie 96% w zaokrągleniu i użycie dropout, aby uniknąć przedwczesnego overfittingu mówią same za siebie.