```python
%matplotlib inline

import tensorflow as tf

from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

import matplotlib.pyplot as plt

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import optimizers
from tensorflow.keras.layers import Dropout
from tensorflow.keras import regularizers

import numpy as np
from numpy.random import seed, randint
```

Sentiment Analysis

In this exercise we use the IMDb-dataset, which we will use to perform a sentiment analysis. The code below assumes that the data is placed in the same folder as this notebook. We see that the reviews are loaded as a pandas dataframe, and print the beginning of the first few reviews.

```python
import numpy as np
import pandas as pd

reviews = pd.read_csv('reviews.txt', header=None)
labels = pd.read_csv('labels.txt', header=None)
Y = (labels=='positive').astype(np.int_)


print(type(reviews))
print(reviews[0])
print(Y.head())
reviews.columns
```

```
<class 'pandas.core.frame.DataFrame'>
0        bromwell high is a cartoon comedy . it ran at ...
1        story of a man who has unnatural feelings for ...
2        homelessness  or houselessness as george carli...
3        airport    starts as a brand new luxury    pla...
4        brilliant over  acting by lesley ann warren . ...
                                ...
24995    i saw  descent  last night at the stockholm fi...
24996    a christmas together actually came before my t...
24997    some films that you pick up for a pound turn o...
24998    working  class romantic drama from director ma...
24999    this is one of the dumbest films  i  ve ever s...
Name: 0, Length: 25000, dtype: object
   0
0  1
1  0
2  1
3  0
4  1
```

```
Index([0], dtype='int64')
```

We need to encode the labels as it is necessary for the Neural network to work.

```python
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder(sparse=False)
Y_one_hot = encoder.fit_transform(Y.to_numpy().reshape(-1, 1))
Y_one_hot
```

```
c:\ProgramData\anaconda3\Lib\site-packages\sklearn\preprocessing\_encoders.py:868: FutureWarning: `sparse` was renamed to `sparse_output` in version 1.2 and will be removed in 1.4. `sparse_output` is ignored unless you leave `sparse` to its default value.
  warnings.warn(
```

```
array([[0., 1.],
       [1., 0.],
       [0., 1.],
       ...,
       [1., 0.],
       [0., 1.],
       [1., 0.]])
```

(a) Split the reviews and labels in test, train and validation sets. The train and validation sets will be used to train your model and tune hyperparameters, the test set will be saved for testing. Use the `CountVectorizer` from `sklearn.feature_extraction.text` to create a Bag-of-Words representation of the reviews. Only use the 10,000 most frequent words (use the `max_features` -parameter of `CountVectorizer`).

```python
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split

vectorizer = CountVectorizer(max_features=10000)
X = vectorizer.fit_transform(reviews[0])

X_train, X_test, Y_train, Y_test = train_test_split(X, Y_one_hot, test_size=0.2, random_state=42)

vectorizer.get_feature_names_out()
```

```
array(['aaron', 'abandon', 'abandoned', ..., 'zoom', 'zorro', 'zu'],
      dtype=object)
```

(b) Explore the representation of the reviews. How is a single word represented? How about a whole review?

```python
X.toarray()
```

```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]], dtype=int64)
```

A single word is represented as one number. The whole review is represented as 10 000 features corresponding to the most frequent words in all reviews. When the word is occurring in the one specific review its count is equal to the number of times it occurs in that review. For instance, the feature names which are ['dog', 'cat', 'dolphin'] in a review: 'I love a cat, but dolphin is my favourite. I could marry a dolphin' will result in a matrix: [0, 1, 2].

(c) Train a neural network with a single hidden layer on the dataset, tuning the relevant hyperparameters to optimize accuracy.

```python
seed(0)
tf.random.set_seed(0)

num_classes = len(np.unique(Y_train))

model = Sequential()
model.add(Dense(units = 32, activation='tanh', input_shape=(X_train.shape[1],))) # add a hidden layer
model.add(Dense(units=num_classes, activation='softmax'))  # Output layer

sgd = optimizers.SGD(learning_rate = 0.01)
model.compile(loss = 'categorical_crossentropy', optimizer = sgd, metrics = ['accuracy'])

history = model.fit(X_train, Y_train, epochs = 40, verbose = 1, validation_split = 0.2)
```

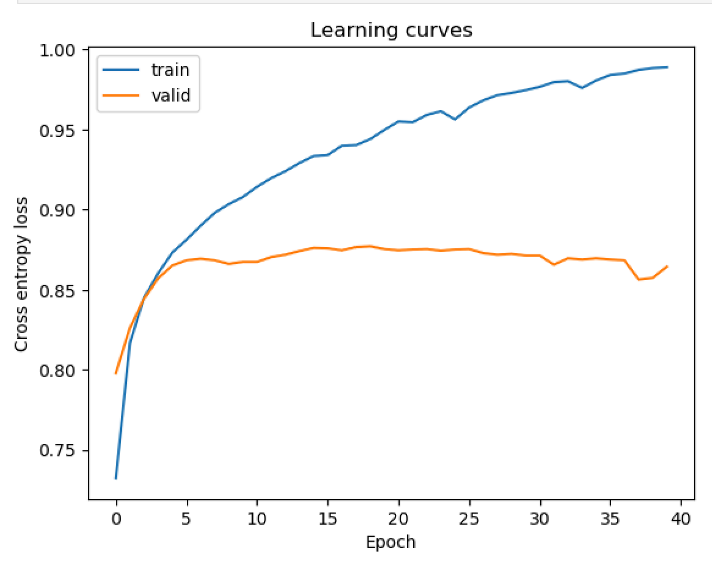
```
Epoch 1/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.6812 - loss: 0.5998 - val_accuracy: 0.7980 - val_loss: 0.4446
Epoch 2/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.8065 - loss: 0.4382 - val_accuracy: 0.8263 - val_loss: 0.3877
Epoch 3/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 8s 15ms/step - accuracy: 0.8404 - loss: 0.3811 - val_accuracy: 0.8445 - val_loss: 0.3556
Epoch 4/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 7s 13ms/step - accuracy: 0.8564 - loss: 0.3437 - val_accuracy: 0.8570 - val_loss: 0.3332
Epoch 5/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 8s 16ms/step - accuracy: 0.8705 - loss: 0.3158 - val_accuracy: 0.8650 - val_loss: 0.3190
Epoch 6/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 8s 11ms/step - accuracy: 0.8790 - loss: 0.2931 - val_accuracy: 0.8683 - val_loss: 0.3111
Epoch 7/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 10s 9ms/step - accuracy: 0.8886 - loss: 0.2738 - val_accuracy: 0.8692 - val_loss: 0.3079
Epoch 8/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - accuracy: 0.8980 - loss: 0.2565 - val_accuracy: 0.8683 - val_loss: 0.3066
Epoch 9/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 8s 16ms/step - accuracy: 0.9046 - loss: 0.2423 - val_accuracy: 0.8660 - val_loss: 0.3072
Epoch 10/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 8s 16ms/step - accuracy: 0.9082 - loss: 0.2313 - val_accuracy: 0.8673 - val_loss: 0.3075
Epoch 11/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - accuracy: 0.9135 - loss: 0.2210 - val_accuracy: 0.8673 - val_loss: 0.3075
Epoch 12/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 8s 15ms/step - accuracy: 0.9187 - loss: 0.2093 - val_accuracy: 0.8702 - val_loss: 0.3071
Epoch 13/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 6s 13ms/step - accuracy: 0.9228 - loss: 0.1987 - val_accuracy: 0.8717 - val_loss: 0.3068
Epoch 14/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 7s 15ms/step - accuracy: 0.9274 - loss: 0.1895 - val_accuracy: 0.8740 - val_loss: 0.3056
Epoch 15/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 9s 17ms/step - accuracy: 0.9316 - loss: 0.1809 - val_accuracy: 0.8760 - val_loss: 0.3048
Epoch 16/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.9316 - loss: 0.1785 - val_accuracy: 0.8758 - val_loss: 0.3046
Epoch 17/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9382 - loss: 0.1668 - val_accuracy: 0.8745 - val_loss: 0.3058
Epoch 18/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9375 - loss: 0.1670 - val_accuracy: 0.8765 - val_loss: 0.3058
Epoch 19/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9412 - loss: 0.1601 - val_accuracy: 0.8770 - val_loss: 0.3081
Epoch 20/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9474 - loss: 0.1506 - val_accuracy: 0.8752 - val_loss: 0.3117
Epoch 21/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9545 - loss: 0.1345 - val_accuracy: 0.8745 - val_loss: 0.3151
Epoch 22/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9498 - loss: 0.1459 - val_accuracy: 0.8750 - val_loss: 0.3190
Epoch 23/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9566 - loss: 0.1323 - val_accuracy: 0.8752 - val_loss: 0.3223
Epoch 24/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9588 - loss: 0.1282 - val_accuracy: 0.8742 - val_loss: 0.3270
Epoch 25/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9503 - loss: 0.1455 - val_accuracy: 0.8750 - val_loss: 0.3276
Epoch 26/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9636 - loss: 0.1197 - val_accuracy: 0.8752 - val_loss: 0.3367
Epoch 27/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 6s 11ms/step - accuracy: 0.9668 - loss: 0.1095 - val_accuracy: 0.8727 - val_loss: 0.3392
Epoch 28/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 6s 12ms/step - accuracy: 0.9688 - loss: 0.1079 - val_accuracy: 0.8717 - val_loss: 0.3482
Epoch 29/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 9s 9ms/step - accuracy: 0.9706 - loss: 0.1017 - val_accuracy: 0.8723 - val_loss: 0.3441
Epoch 30/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 8s 14ms/step - accuracy: 0.9725 - loss: 0.0984 - val_accuracy: 0.8712 - val_loss: 0.3466
Epoch 31/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 9s 11ms/step - accuracy: 0.9748 - loss: 0.0942 - val_accuracy: 0.8712 - val_loss: 0.3506
Epoch 32/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 10s 10ms/step - accuracy: 0.9786 - loss: 0.0831 - val_accuracy: 0.8655 - val_loss: 0.3774
Epoch 33/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 9s 18ms/step - accuracy: 0.9786 - loss: 0.0838 - val_accuracy: 0.8695 - val_loss: 0.3613
Epoch 34/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.9714 - loss: 0.1176 - val_accuracy: 0.8687 - val_loss: 0.3813
Epoch 35/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9776 - loss: 0.0888 - val_accuracy: 0.8695 - val_loss: 0.3792
Epoch 36/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9831 - loss: 0.0722 - val_accuracy: 0.8687 - val_loss: 0.3787
Epoch 37/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9835 - loss: 0.0708 - val_accuracy: 0.8683 - val_loss: 0.3883
Epoch 38/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9873 - loss: 0.0604 - val_accuracy: 0.8562 - val_loss: 0.4301
Epoch 39/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9887 - loss: 0.0567 - val_accuracy: 0.8572 - val_loss: 0.4319
Epoch 40/40
500/500 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.9883 - loss: 0.0550 - val_accuracy: 0.8643 - val_loss: 0.4075
```

40 epochs seems like a lot as it took over 3 minutes to run but with a learning rate of 0.01 it slowly reached 95% accuracy on the train data.

```python
print("Loss + accuracy on train data: {}".format(model.evaluate(X_train, Y_train)))
```

```
625/625 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9749 - loss: 0.0763
Loss + accuracy on train data: [0.1383526772260666, 0.953249990940094]
```

```python
plt.figure()
plt.title("Learning curves")
plt.xlabel("Epoch")
plt.ylabel("Cross entropy loss")
plt.plot(history.history['accuracy'], label = 'train')
plt.plot(history.history['val_accuracy'], label = 'valid')
plt.legend()
plt.show()
```




```python
seed(0)
tf.random.set_seed(0)

num_classes = len(np.unique(Y_train))

model = Sequential() # initialize a neural network
model.add(Dense(units = 50, activation='sigmoid', input_shape=(X_train.shape[1],))) # add a hidden layer
model.add(Dense(units=num_classes, activation='silu'))  # Output layer
# The Swish (or Silu) activation function is a smooth, non-monotonic function that is unbounded above and bounded below.

adam = optimizers.Adam(learning_rate = 0.01)
model.compile(loss = 'binary_crossentropy', optimizer = adam, metrics = ['accuracy'])
#Try different loss functions and different optimizers

history = model.fit(X_train, Y_train, epochs = 15, verbose = 1, validation_split = 0.2)
```

```
Epoch 1/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.4974 - loss: 4.3899 - val_accuracy: 0.5017 - val_loss: 4.3044
Epoch 2/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.4991 - loss: 4.1630 - val_accuracy: 0.5035 - val_loss: 4.2642
Epoch 3/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.5016 - loss: 4.1436 - val_accuracy: 0.5265 - val_loss: 4.3364
Epoch 4/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 5ms/step - accuracy: 0.5157 - loss: 4.1082 - val_accuracy: 0.5170 - val_loss: 4.4202
Epoch 5/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.5189 - loss: 4.1083 - val_accuracy: 0.5217 - val_loss: 4.4747
Epoch 6/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.5332 - loss: 4.1174 - val_accuracy: 0.5213 - val_loss: 4.4857
Epoch 7/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.5429 - loss: 4.1865 - val_accuracy: 0.8625 - val_loss: 6.8746
Epoch 8/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.6975 - loss: 4.4579 - val_accuracy: 0.6208 - val_loss: 4.4582
Epoch 9/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.6038 - loss: 6.0122 - val_accuracy: 0.6398 - val_loss: 4.6645
Epoch 10/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.7029 - loss: 4.1930 - val_accuracy: 0.6768 - val_loss: 4.5153
Epoch 11/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.6986 - loss: 4.1378 - val_accuracy: 0.7128 - val_loss: 4.6033
Epoch 12/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.7573 - loss: 3.9660 - val_accuracy: 0.8608 - val_loss: 1.6045
Epoch 13/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 7ms/step - accuracy: 0.9354 - loss: 0.5987 - val_accuracy: 0.8758 - val_loss: 1.5021
Epoch 14/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 4s 7ms/step - accuracy: 0.9608 - loss: 0.4466 - val_accuracy: 0.8730 - val_loss: 1.4477
Epoch 15/15
500/500 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.9701 - loss: 0.3030 - val_accuracy: 0.8648 - val_loss: 1.5573
```

Okay interesting, this one did almost the same job as the previous one but in less epochs, so I will go with this one.

```python
print("Loss + accuracy on train data: {}".format(model.evaluate(X_train, Y_train)))
```

```
625/625 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.9733 - loss: 0.2724
Loss + accuracy on train data: [0.5141831636428833, 0.9526000022888184]
```

(d) Test your sentiment-classifier on the test set.

```python
print("Loss + accuracy on TEST data: {}".format(model.evaluate(X_test, Y_test)))
```

```
157/157 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8643 - loss: 1.5914
Loss + accuracy on TEST data: [1.574142575263977, 0.8622000217437744]
```

That's not bad, 86 percent is a decent result, but it could be a sign that the model is overfitting slightly.

(e) Use the classifier to classify a few sentences you write yourselves.

```python
comments = ["Honestly, I haven't seen a movie that left me so unsatisfied in a long time.", # 0 - negative
            'Oh my good, this movie was crazy.',  # 1 - positive yet ambigous
            'Spending three hours to watch that?', # 0 - negative
            'Three hours well spent!',# 1 - positive but also ambiguous
            'I really liked this movie, I would highly recommend it to anyone who is into comedy!', # 1 - definitely positive
            'The plot is unconventional and unexpected. It is a refreshing act in comparison to what is served to the masses these days.' # 1 - positive but bitter
            ]

pd_comments =  pd.DataFrame(comments)
trans_comments=vectorizer.transform(pd_comments[0])


probabilities = np.array(model.predict(trans_comments))
predictions = np.argmax(probabilities, axis = 1) #what does the model predict

print("Predictions = {}".format(predictions[0:30]))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 87ms/step
Predictions = [0 0 0 0 1 1]
```

It is doing pretty well. Maybe the problem was that I came with the mindset of trying to trick the model by making something sound bad yet making it a positive comment. On the ones that are straightforward it does a good job. Let's try getting few reviews from the internet about Barbie and Oppenheimer.

```python
oppenheimer_barbie = ["You'll have to have your wits about you and your brain fully switched on watching Oppenheimer as it could easily get away from a nonattentive viewer. This is intelligent filmmaking which shows it's audience great respect. It
                      "I really wanted to like this movie but I struggled to stay awake through it. For me it had none of the nuanced beauty of The Imitation Game, but rather a lot of political waffle, which left me sadly caring more about when it
                      "As much as it pains me to give a movie called 'Barbie' a 10 out of 10, I have to do so. It is so brilliantly handled and finely crafted, I have to give the filmakers credit. Yes, I am somewhat conservative person and former
                      "Margot Robbie and Ryan Gosling are really great in their roles of Barbie and Kent. Gosling is specially hilarious. I expected a funny, cool, deep and entertaining movie, but I was highly disappointed. The movie is so terribl
#So I added two reviews for each movie, [1, 0, 1, 0]


pd_comments =  pd.DataFrame(oppenheimer_barbie)
trans_comments=vectorizer.transform(pd_comments[0])


probabilities = np.array(model.predict(trans_comments))
predictions = np.argmax(probabilities, axis = 1) #what does the model predict

print("Predictions = {}".format(predictions[0:30]))
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 30ms/step
Predictions = [1 0 1 0]
```

On the real data the model doesn't miss! That is so interesting to see, but also something to be aware of as we can predict the people's emotions with a machine learning model and this gives a lot of ground for a malicous usage of such great tools.