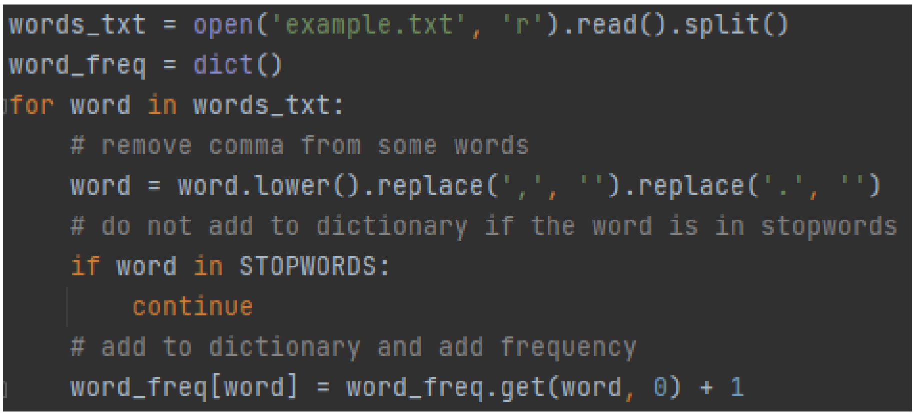
AI & Automation Exam 2

05/06/2022

Mads Hegewald (EAAMHEG)

Exercise 1

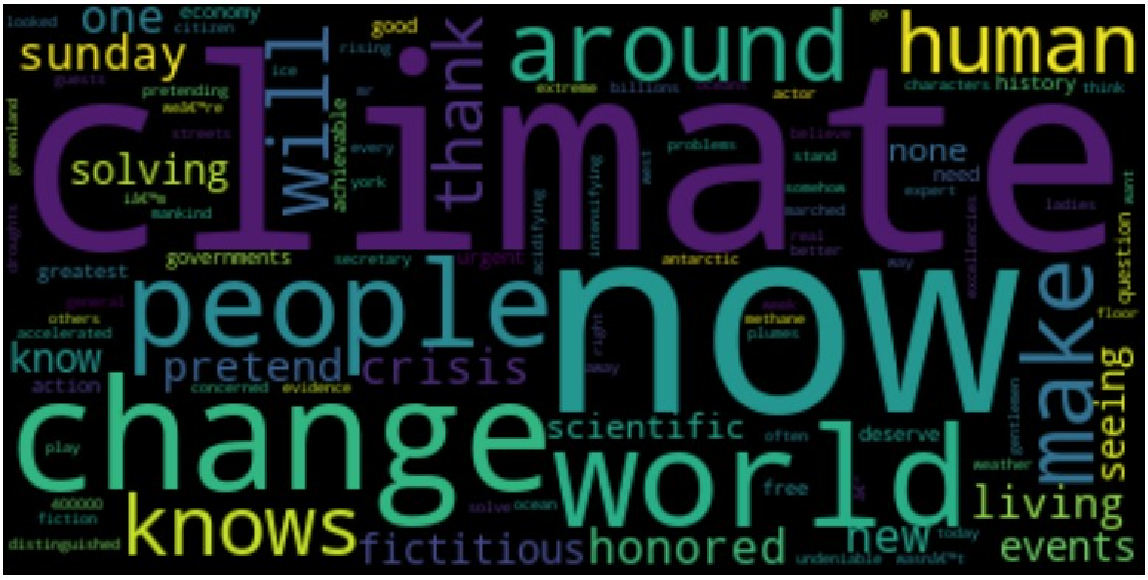
The first example of wordcloud is where I have used a file called “example.txt” in the directory “exercise1”, which is the transcript of the speech by Leonardo DiCaprio at the 2014 UN Climate Summit. First I read the txt file where I used .split() to get each individual word, followed by removing the comma “,” or dot “.” at the end of some of the words. After I added the words to a dictionary to keep track of the frequency of the words, but I did not want to add words such as “the”, “as”, “for” etc. So, I filtered these words out by using checking if the word was in the STOPWORDS from wordcloud. This is all done using the following code:



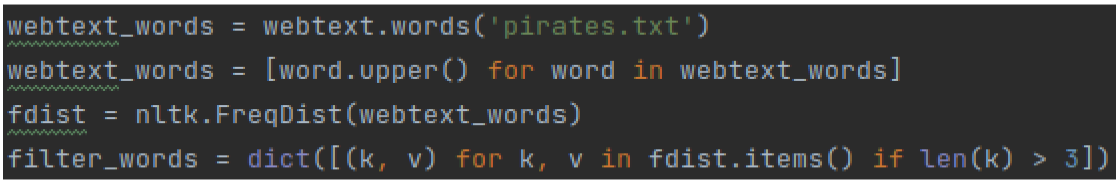
```
words_txt = open('example.txt', 'r').read().split()
word_freq = dict()
for word in words_txt:
    # remove comma from some words
    word = word.lower().replace(',', '').replace('.', '')
    # do not add to dictionary if the word is in stopwords
    if word in STOPWORDS:
        continue
    # add to dictionary and add frequency
    word_freq[word] = word_freq.get(word, 0) + 1
```

Running the code shows the following wordcloud. One thing I noticed was how I wanted it to say “climate change” in one word but after some experimenting, I could not find a solid solution to that problem and stuck with splitting the words when a space between them occurred.

Wordcloud of Leonardo DiCaprio speech read from text file



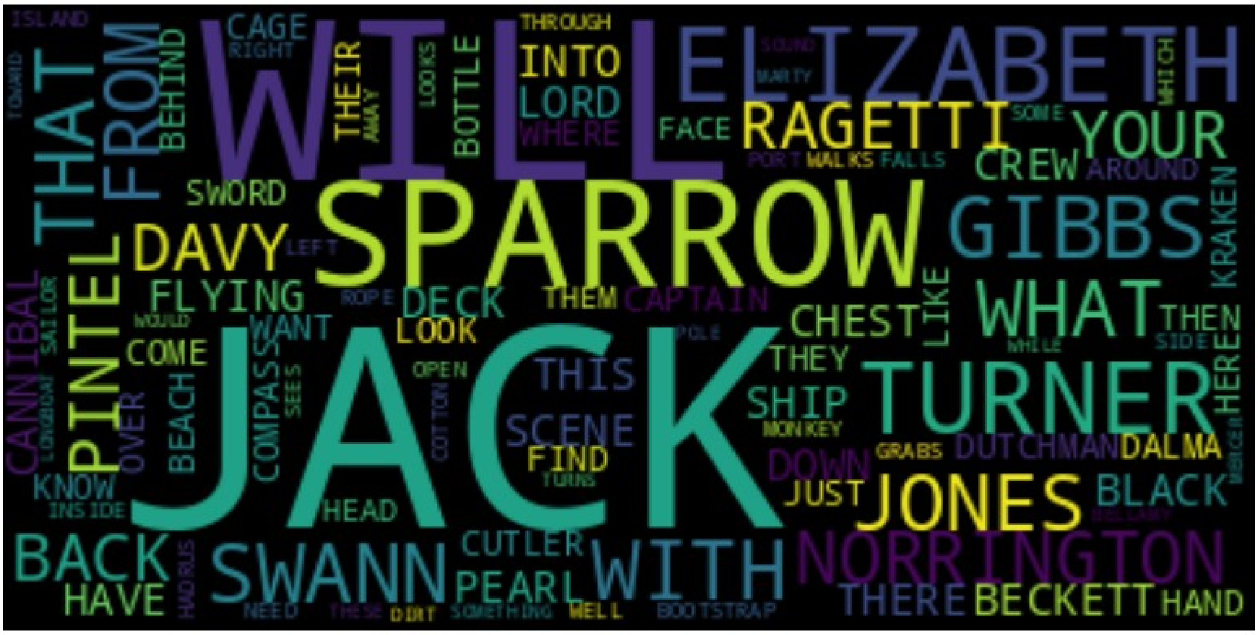
For my second example I tried using a webtext from nltk.corpus called “pirate.txt” located at the same place as the example with “firefox.txt”. It seems like the text file contains a script of some sorts of the Pirates of the Caribbean movie. First we load in the words from webtext and use the nltk.FreqDist to create a dictionary with the frequencies of each word, this still contains all the stopwords mentioned previously. To sort these out I used a line from the example we had during one of the lessons, where we basically sort out all the words that are smaller than 3. This is done in the following code:



```
webtext_words = webtext.words('pirates.txt')
webtext_words = [word.upper() for word in webtext_words]
fdist = nltk.FreqDist(webtext_words)
filter_words = dict([(k, v) for k, v in fdist.items() if len(k) > 3])
```

After this we can simply just create a wordcloud object, where we generate the words from frequencies in the list “filter_words” and we get the following wordcloud.

Wordcloud of pirates.txt from webtext

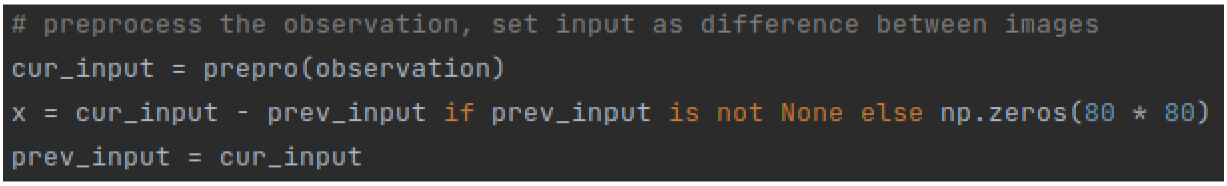


Exercise 2

a)

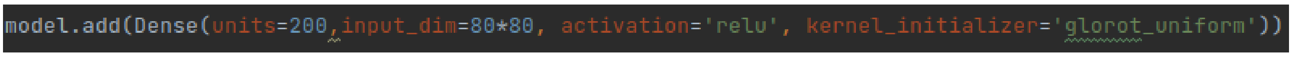
For this exercise I chose the “pong_train.py” program. The design of the neural net is just one hidden layer with 200 neurons, so pretty simple.

The input for the neural net are the images of the pong table. Although before it is fed to the neural network there is some preprocessing happening to the image. It is converted from 210x160x3 to 80x80 or basically a 6400x1 matrix. There is also other preprocessing done to the image such as downsampling the image and removing backgroudn and colour. It is not enough with only using 1 image since the neural net also needs to know the direction of the pong to determine whether the paddle should move up or down. So we are actually feeding the neural net the difference between the two frames. The input for the neural network is obtained through these lines (input is x):



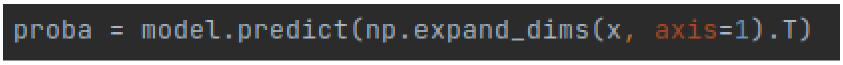
```
# preprocess the observation, set input as difference between images
cur_input = prepro(observation)
x = cur_input - prev_input if prev_input is not None else np.zeros(80 * 80)
prev_input = cur_input
```

The input layer can also be seen in the network at these lines of code:



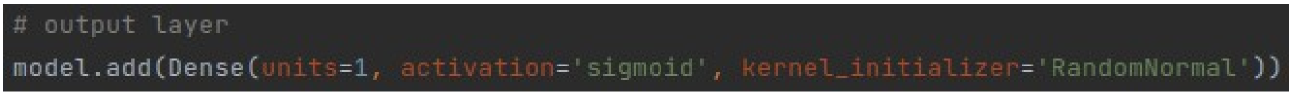
```
model.add(Dense(units=200,input_dim=80*80, activation='relu', kernel_initializer='glorot_uniform'))
```

The output of the neural net is achieved through the following line:



```
proba = model.predict(np.expand_dims(x, axis=1).T)
```

Here the ”proba” is a probability calculated/predicted by the network, which we then use to decide whether we want to take an UP or DOWN action. We can see the output layer in the neural net here:

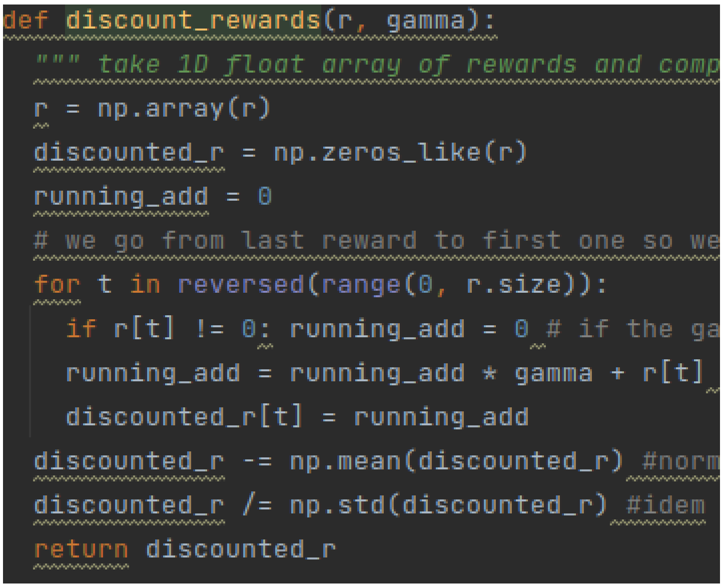


```
# output layer
model.add(Dense(units=1, activation='sigmoid', kernel_initializer='RandomNormal'))
```

b)

They way the neural net is trained is at the end of each episode we fit the model with the data gathered in x_train and y_train as well the discounted reward.

Below is the method for the rewards given to train the model:



```
def discount_rewards(r, gamma):
    """ take 1D float array of rewards and comp
    r = np.array(r)
    discounted_r = np.zeros_like(r)
    running_add = 0
    # we go from last reward to first one so we
    for t in reversed(range(0, r.size)):
        if r[t] != 0: running_add = 0 # if the ga
        running_add = running_add * gamma + r[t]
        discounted_r[t] = running_add
    discounted_r -= np.mean(discounted_r) #norm
    discounted_r /= np.std(discounted_r) #idem
    return discounted_r
```

The sample_weights in the fit method updates the weights of the model during training. The weights are updated with the reward gained from the discounted_reward method. Basically what this does is transform the list of rewards so that it encourages any action that can lead to a positive reward. So any action that might lead to a positive reward tunes the weights of the model to keep predicting this winning action. The same can be said for actions that yield a negative reward, where it tunes the weights not to take these negative actions.

c)

Experience replay is about storing the agent’s experiences at each time-step. This means we save the data that is discovered (state, action, reward, next state) and put it into a mini-batch, which the agent can then randomly draw from. The use of previous experience helps improve the policy and actions taken will be closer the the optimal ones. Experience replay is helpful when we need to get the most out of the available data, since the agent can learn with it multiple times.

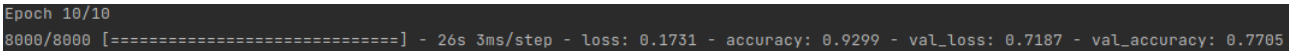
DQN target network is a copy of state-action value function (or Q-function) that is not changed for a fixed amount of time steps so we get stable learning for this period. If we do not have a target network, the Q-learning targets would keep changing making it more unstable (since the updated values fluctuate). Basically we use the target network to take the next state and predict the best action (Q-value) out of all the actions that can be taken from that state.

These techniques help with catastrophic forgetting, which is a term used when the network is trained on a specific task and the weights are adjusted to it. We then introduce a new task and the weights are adjusted for the new task, meaning that it forgets the knowledge of the previously learned task.

Exercise 3

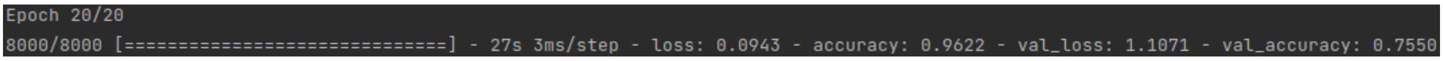
a)

The initial accuracy of the model when run is around 9.3 at epoch 10:



```
Epoch 10/10
8000/8000 [==============================] - 26s 3ms/step - loss: 0.1731 - accuracy: 0.9299 - val_loss: 0.7187 - val_accuracy: 0.7705
```

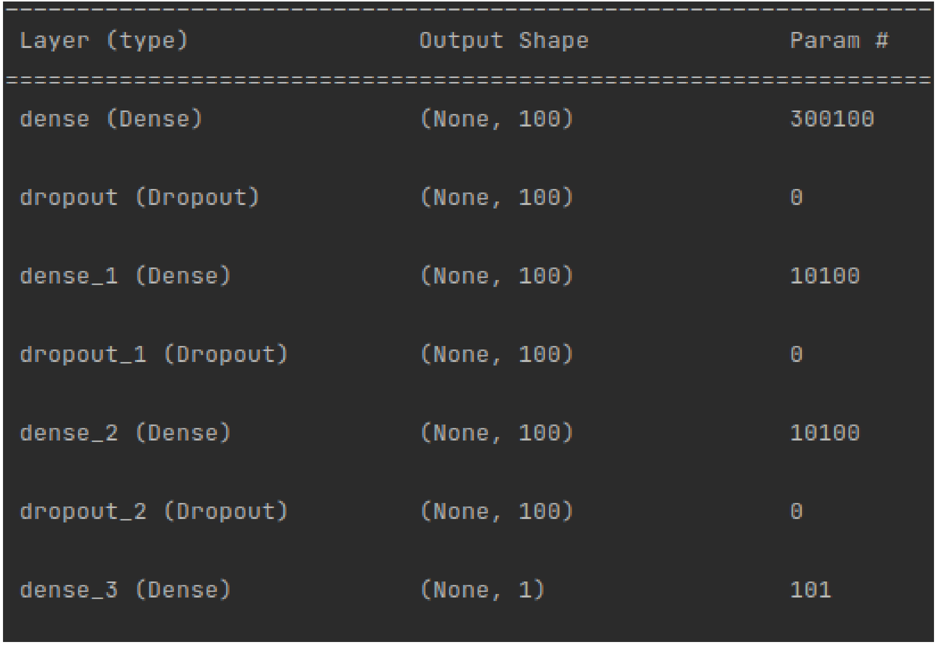
For my first experiment I tried increasing the epochs to make it train for twice the amount (increased from 10 to 20), where I was expecting an increase in accuracy:



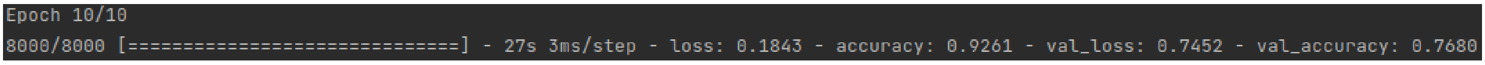
```
Epoch 20/20
8000/8000 [==============================] - 27s 3ms/step - loss: 0.0943 - accuracy: 0.9622 - val_loss: 1.1071 - val_accuracy: 0.7550
```

A slight increase of about 0.03, which is a nice increase but not a lot considering it trained for twice as long. Even though we gained more accuracy on the training set, the accuracy for the validation set declined a little (0.015).

In my next experiment I tried adding another dense layer with 100 neurons followed by a dropout layer. So the model looks like this:



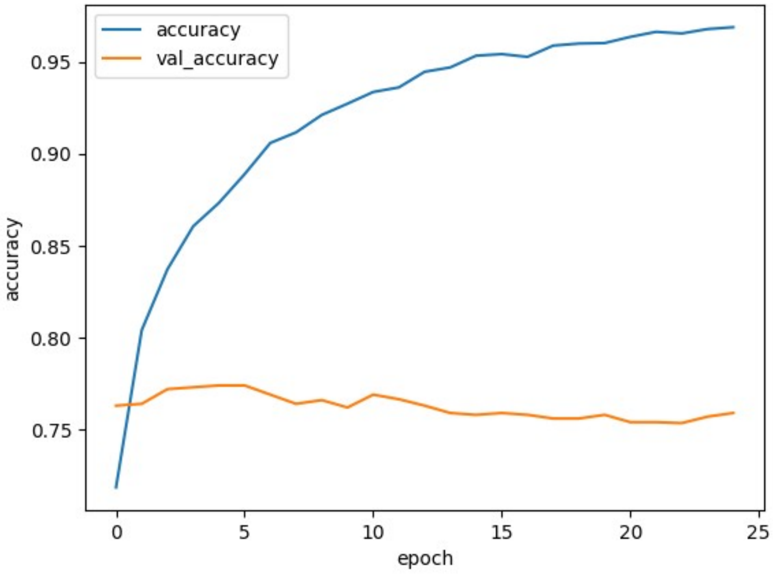
```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 100)               300100

 dropout (Dropout)           (None, 100)               0

 dense_1 (Dense)             (None, 100)               10100

 dropout_1 (Dropout)         (None, 100)               0

 dense_2 (Dense)             (None, 100)               10100

 dropout_2 (Dropout)         (None, 100)               0

 dense_3 (Dense)             (None, 1)                 101
```

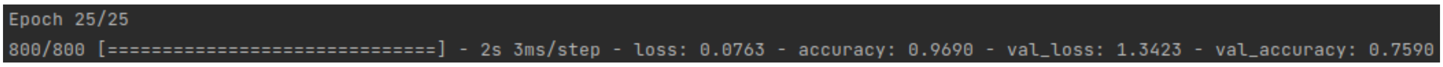


```
Epoch 10/10
8000/8000 [==============================] - 27s 3ms/step - loss: 0.1843 - accuracy: 0.9261 - val_loss: 0.7452 - val_accuracy: 0.7680
```

This did not increase the accuracy and actually worsened it, so just increasing layers this way did not produce a positive result. Except for a slight increase in validation accuracy compared to the previous experiment.

For some of my final experiments I tried changing the overall design of the neural net to make it less complicated. I would want to try and remove a layer as well as some of the neurons on the layer, the single dropout layer will stay with the same ratio (0.5). But I will also let it run for a few more epochs (25) to see if there will be any difference.





```
Epoch 25/25
800/800 [==============================] - 2s 3ms/step - loss: 0.0763 - accuracy: 0.9690 - val_loss: 1.3423 - val_accuracy: 0.7590
```

No increase in validation accuracy but a slight increase in accuracy on the training data.

b)

Most of the improvements (outside of the neural net) can be done in the prepossessing step, so right before we feed the data to the neural net. We can try and clean the data from unnecessary fields/labels or add more data that will help the neural net decide the sentiment. There are probably a lot of unnecessary stuff on tweets that we clean.

Some of the prepossessing could be stemming. Stemming is about shortening a word to its stem/root format. An example for this could be "writing" where the stem for the word is "write". This process helps reduce the complexity meanwhile still retaining the meaning carried by the words.

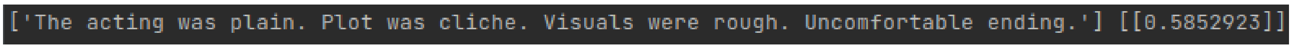
The last suggestion is lemmatization, which is about normalizing words. Here we get a root word instead of a root stem, the output of stemming. After lemmatization we will get a word that has the same meaning as the original word. An example for this could be the words "is", "was", "were" are converted back to the root word "be".

c)

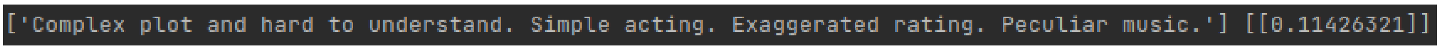
If the given score is closer to 0 it means a negative review and the closer to 1 the more positive the review, where 0.5 would be neutral.

In my own examples I wanted to try some sentences which might seem natural or are

harder to define as positive or negative. Although the words to describe the plot, music, acting etc. might seem negative they can don't have to be in most circumstances (in my opinion). I did this in the first 2 examples, where I got the following scores:
I thought the score for the first experiment would be more neutral (closer to 0.5), but the neural net predicted that it was a more negative review.



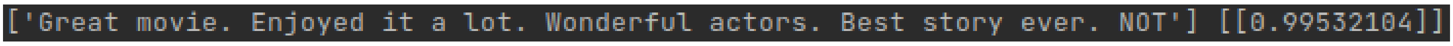
```
['The acting was plain. Plot was cliche. Visuals were rough. Uncomfortable ending.'] [[0.5852923]]
```



```
['Complex plot and hard to understand. Simple acting. Exaggerated rating. Peculiar music.'] [[0.11426321]]
```

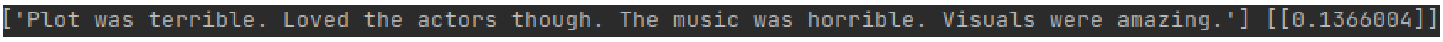
The second experiment result was closer to what I was expecting from the experiment since I was going for a neutral score.

Next I wanted to try the example we have been talking about during the lessons a lot of the time. Here I wrote some really positive sentences to start with and ended the review with a "NOT" indicating it was not a positive review.



```
['Great movie. Enjoyed it a lot. Wonderful actors. Best story ever. NOT'] [[0.99532104]]
```

This result was exactly as I was expecting as well as what we have commented on during class, the neural net did not understand it and deemed it as an overly positive review.
Lastly I wanted to try with mixed sentences, both negative and positive sentences in the review. Result is as follows:



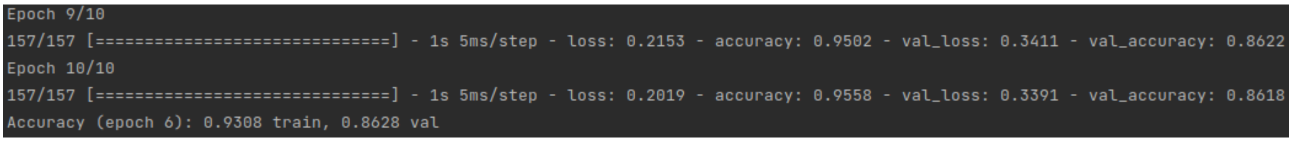
```
['Plot was terrible. Loved the actors though. The music was horrible. Visuals were amazing.'] [[0.1366004]]
```

Apparently the negative sentences/words outweighed the positive ones by a lot, causing the neural net to deem the review as negative.

Exercise 4

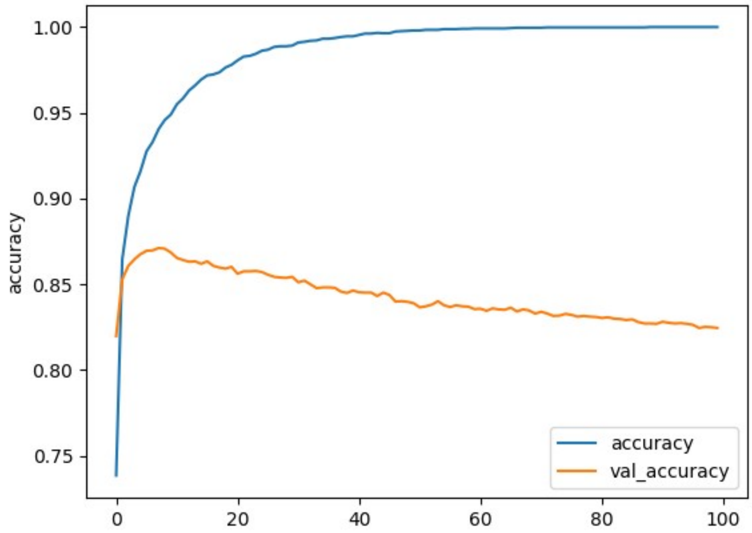
a)

The initial run received following accuracy:



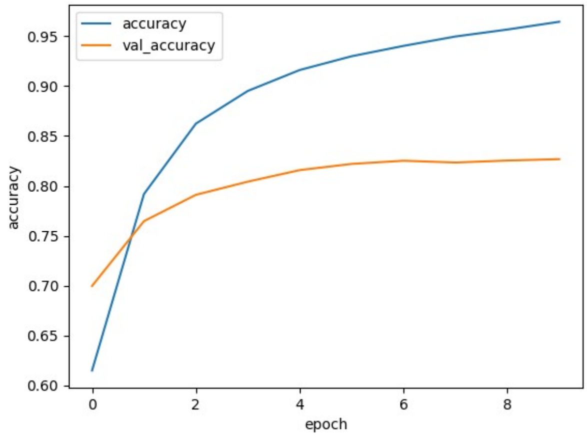
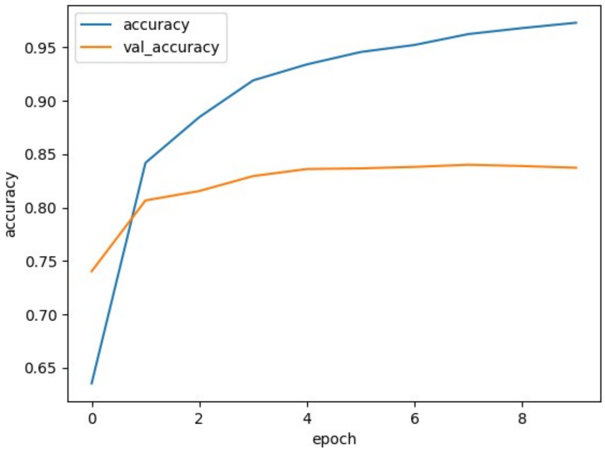
```
Epoch 9/10
157/157 [==============================] - 1s 5ms/step - loss: 0.2153 - accuracy: 0.9502 - val_loss: 0.3411 - val_accuracy: 0.8622
Epoch 10/10
157/157 [==============================] - 1s 5ms/step - loss: 0.2019 - accuracy: 0.9558 - val_loss: 0.3391 - val_accuracy: 0.8618
Accuracy (epoch 6): 0.9308 train, 0.8628 val
```

Running the simple model for 100 epochs yields:

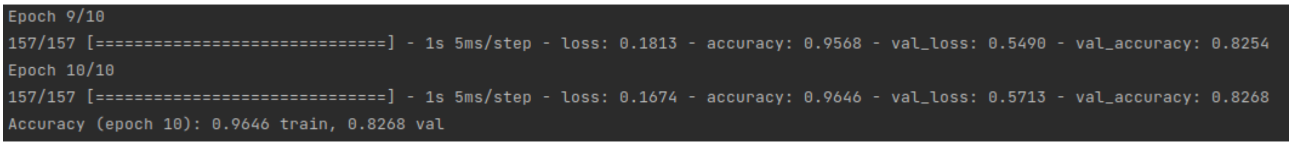


We can see that it quickly reaches an accuracy greater than 0.95 in 20 epochs, where it slowly increases after. The increase in accuracy gets smaller and the loss becomes smaller as well the closer it gets to an accuracy of 1, but once it reaches around 90 epochs it reaches 1 in accuracy. Interestingly the validation accuracy keeps decreasing, meaning it does worse when presented to new test samples (overfitted).

For the other small adjustments I made I tried changing the activation function from sigmoid to one experiment with tanh and one with relu (still keeping it at 10 epochs).

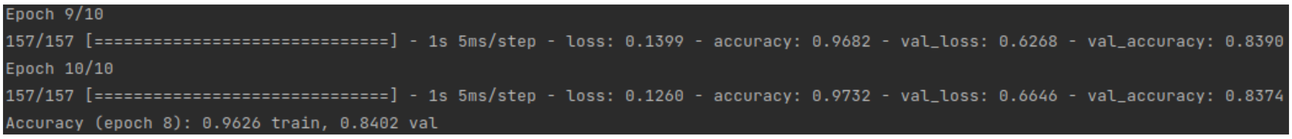
Tanh activation	Relu activation
	

Tanh



```
Epoch 9/10
157/157 [==============================] - 1s 5ms/step - loss: 0.1813 - accuracy: 0.9568 - val_loss: 0.5490 - val_accuracy: 0.8254
Epoch 10/10
157/157 [==============================] - 1s 5ms/step - loss: 0.1674 - accuracy: 0.9646 - val_loss: 0.5713 - val_accuracy: 0.8268
Accuracy (epoch 10): 0.9646 train, 0.8268 val
```

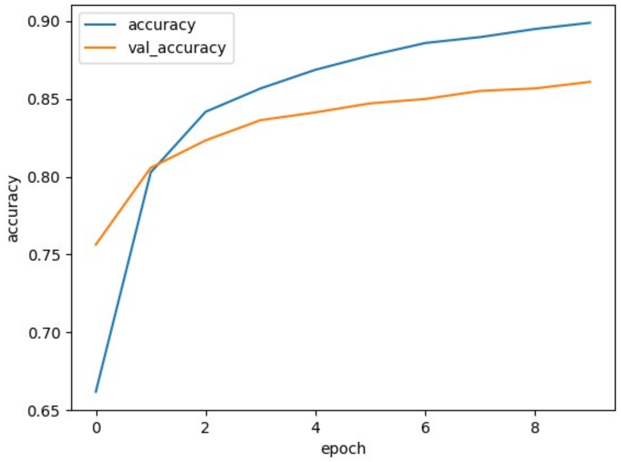
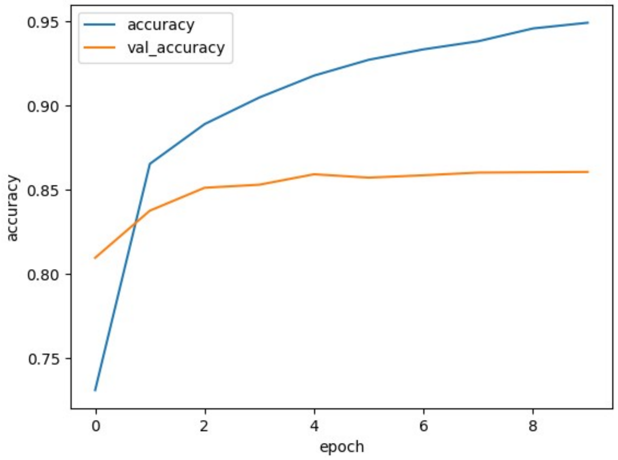
Relu



```
Epoch 9/10
157/157 [==============================] - 1s 5ms/step - loss: 0.1399 - accuracy: 0.9682 - val_loss: 0.6268 - val_accuracy: 0.8390
Epoch 10/10
157/157 [==============================] - 1s 5ms/step - loss: 0.1260 - accuracy: 0.9732 - val_loss: 0.6646 - val_accuracy: 0.8374
Accuracy (epoch 8): 0.9626 train, 0.8402 val
```

Both experiments ends up doing better than the initial one, getting a higher accuracy at epoch 10, but the accuracy on the validation set did not increase compared to the initial run.

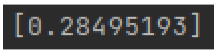
For other minor changes, I tried changing the optimizer to see if this would yield more accuracy. The plots are as follows:

Adamax	Nadam
	

The change in optimizer did not give better overall stats for the network, though the Nadam optimizer was very close to have the same accuracy as the initial run.

b)

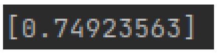
The first review I gave the model was a negative review on the movie "Morbius" where it only had a rating of 1/10 stars. The model defined the review as a more negative, which is correct in this case. I would have predicted the score to in the range of 0 to 0.2 since the rating was bad.



```
[0.28495193]
```

Maybe the reason why it didn't quite achieve the score was because the reviewer didn't have quite a detailed explanation to why they thought it was this bad, or maybe the reasons weren't described well enough.

The second review I gave the model was a positive review of the movie “Mission Impossible Fallout” with a rating of 9/10. Here I expected a result close to 1 since the review was overly positive, but it only scored it at around 0.75.



```
[0.74923563]
```

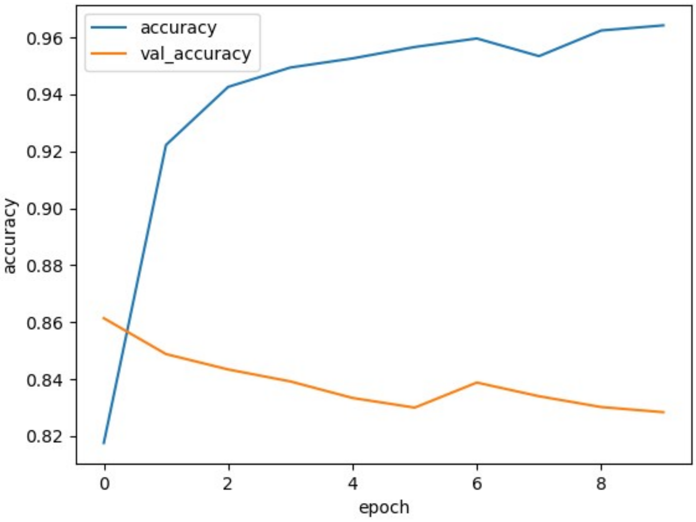
In this review it was talking about how this movie is the new standard for non-superhero movies and so on, making it one of the best action movies according to this person’s opinion. The score was probably lower due to the review not being very long and also the reviewer comparing it to other well rated actions movies (James Bond- and the Bourne movies). The network does not have any knowledge of these movies so the information is deemed useless.

c)

From the graph we can see that the model does extremely well on the training data, but when it comes to the validation set it does not perform very well. It even decreases meaning that the model is overfitting the data. This can be seen by looking at the “accuracy” increasing by a lot while “val_accuracy” is decreasing from the very start.

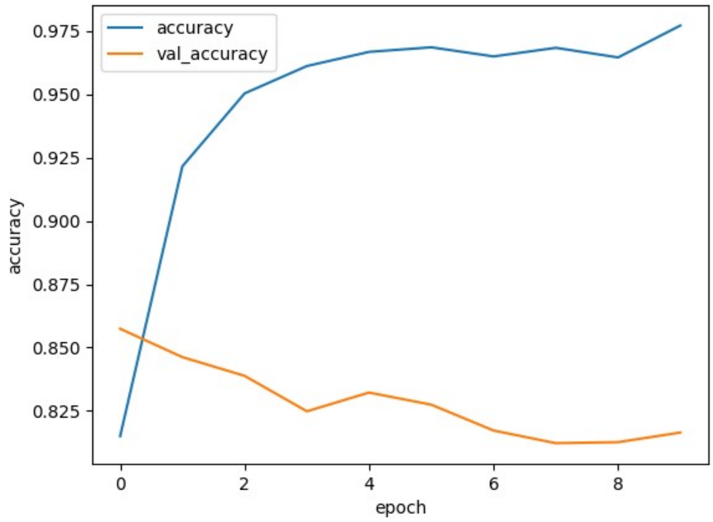
d)

If we can see that the model perform really well on the training data, but when it comes to actual samples it perform not so well, it means that the model is overfitting. This can be seen if the “accuracy” is really good but the “val_accuracy” is not good. There are a few ways to handle overfitting, one is changing the complexity of the neural net. If the neural net is too complex it could lead to overfitting. Another way for dealing with overfitting is regularization. In this case we could regularize with either a dropout layer, batch normalization or L2 weight regularization.

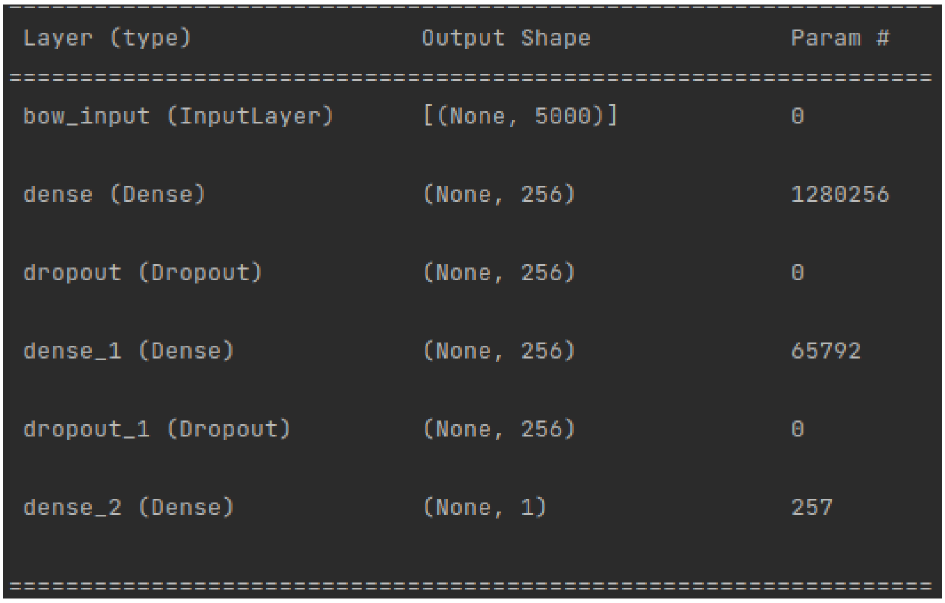


The plot above shows us that adding dropout and and regularization didn’t really improve the overall network, since the training accuracy does not reach 1 and that validation accuracy may start at a slightly higher value (0.01 higher), but the overall result is still worse.

In another attempt to improve network I decreased the neurons (N) in each layer to 64 instead of 128 and decreased the dropout ratio from 0.5 to 0.2. The accuracy may have increased, but the validation accuracy took did not seem to improve.

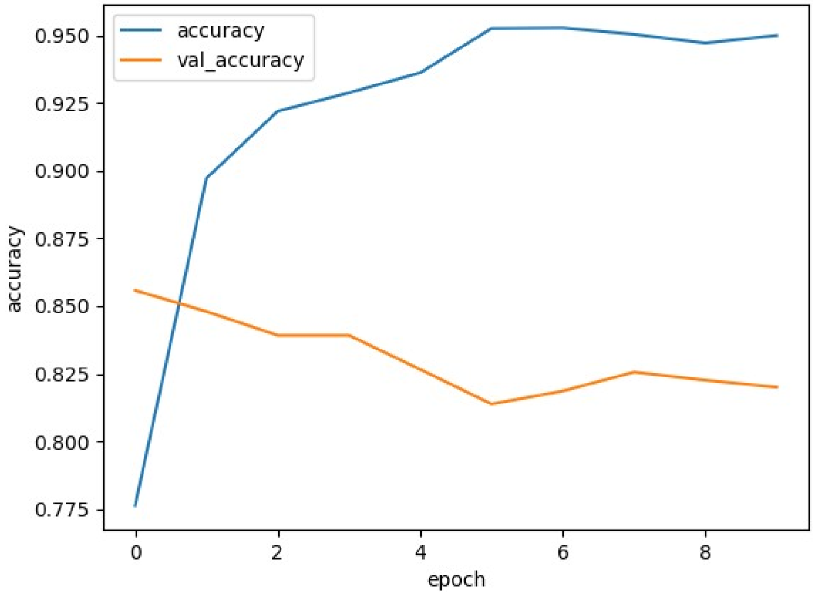


For my last experiment I wanted to see what would happen if we might make the network a little more complex. Here was the network I ended up with:



```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 bow_input (InputLayer)      [(None, 5000)]            0

 dense (Dense)               (None, 256)               1280256

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 256)               65792

 dropout_1 (Dropout)         (None, 256)               0

 dense_2 (Dense)             (None, 1)                 257


=================================================================
```

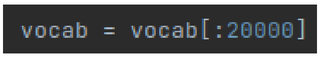
Not very complex but quite a few neurons and layers compared to before (double the amount).



In conclusion the network should not be too complex otherwise it will have trouble not overfitting the data. The optimal approach to this is probably having a more simple network where we still get a stable accuracy as well as validation accuracy.

e)

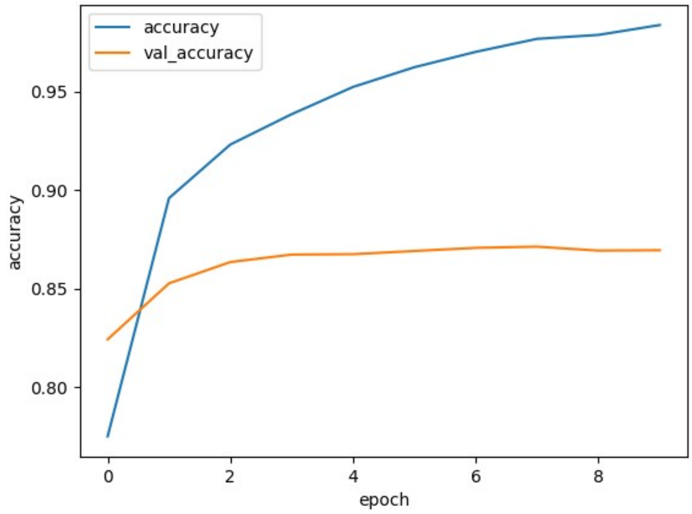
Changing the amount of words is done through the following line:

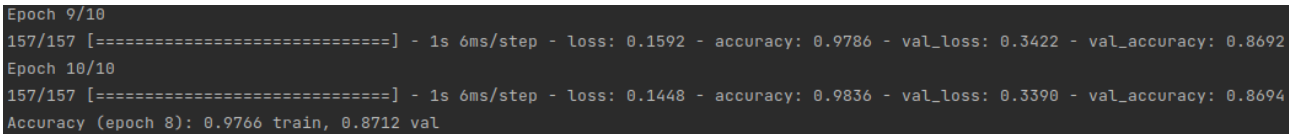


```
vocab = vocab[:20000]
```

For this exercise it would not allow me to train with all 90000 (cause of system memory issues), so I shortened it to 20000 which was allowed.

The experiment went as follows (simple model):





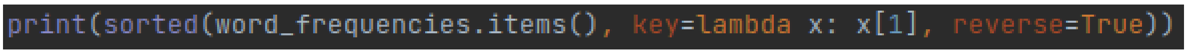
```
Epoch 9/10
157/157 [==============================] - 1s 6ms/step - loss: 0.1592 - accuracy: 0.9786 - val_loss: 0.3422 - val_accuracy: 0.8692
Epoch 10/10
157/157 [==============================] - 1s 6ms/step - loss: 0.1448 - accuracy: 0.9836 - val_loss: 0.3390 - val_accuracy: 0.8694
Accuracy (epoch 8): 0.9766 train, 0.8712 val
```

The accuracy increase was at 0.03 as well as a small increase in the validation accuracy to almost 0.87. So increasing the amount of data is useful for improving the model, but that is only if your computer can actually handle all the extra data as well as the extra time it takes to train on the increased data.

Exercise 5

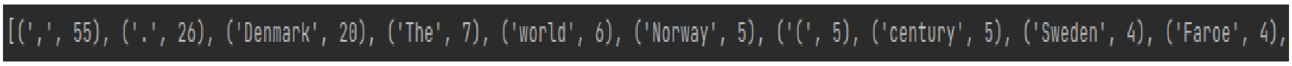
a)

To make it easier for myself too see which the most frequently used words were, I sorted the list of word_frequencies by the frequency value.



```
print(sorted(word_frequencies.items(), key=lambda x: x[1], reverse=True))
```

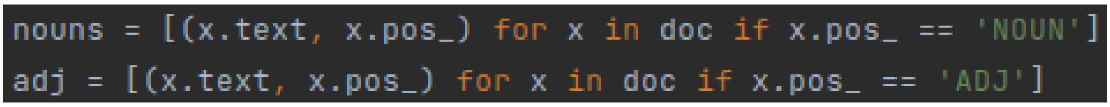
We have "Denmark" which is used 20 times, next is "The" with 7 usages. The words "Norway" and "century" both occur 5 times, and then we have a lot of words sharing a frequency of 4.



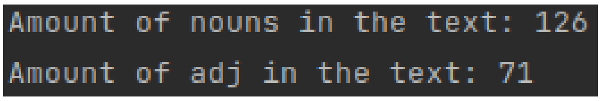
```
[(',', 55), ('.', 26), ('Denmark', 20), ('The', 7), ('world', 6), ('Norway', 5), ('(', 5), ('century', 5), ('Sweden', 4), ('Faroe', 4),
```

b)

To find out the number of nouns and verbs in the text, I looped through the words and checked the .pos_ field on every word. This will return what type the word is, where "NOUN" for noun and "ADJ" for adjectives. After this it's simply just adding them to their own list. We can then see all the nouns and adjectives as well as getting the total amount of each, which is 126 nouns and 71 adjectives. The following code represents what is described:



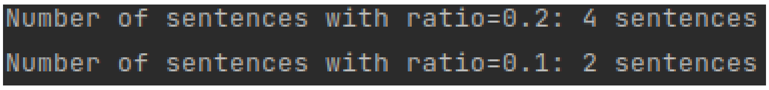
```
nouns = [(x.text, x.pos_) for x in doc if x.pos_ == 'NOUN']
adj = [(x.text, x.pos_) for x in doc if x.pos_ == 'ADJ']
```



```
Amount of nouns in the text: 126
Amount of adj in the text: 71
```

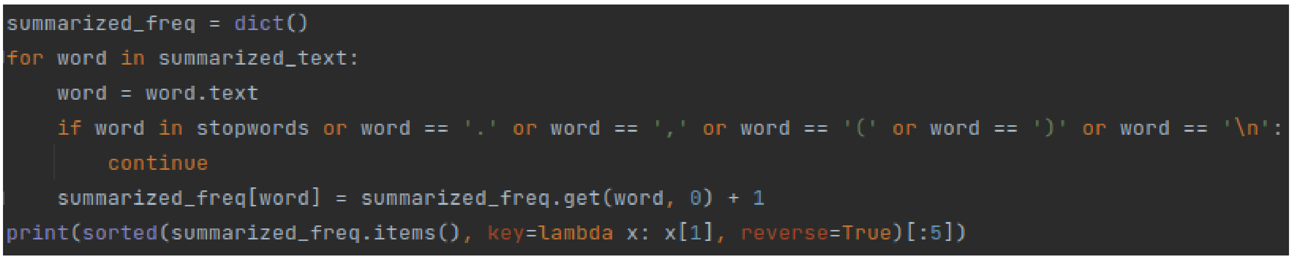
c)

The summarize function summarizes the input text based on a given ratio which in this case is 0.2. This ratio determines the number of sentences of the original text to be chosen for the summary. In the original text we have 26 sentences and with a ratio of 0.2 we end up with 4 sentences. With a ratio of 0.1 we only get 2 sentences.
The way it chooses the sentences is based on a variation of the TextRank algorithm.

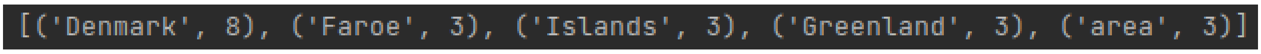


```
Number of sentences with ratio=0.2: 4 sentences
Number of sentences with ratio=0.1: 2 sentences
```

To get the 5 most frequent word I created a dictionary and added the words to it if they were not in the stopwords or were a symbol such as a parenthesis.
Then I sorted the dictionary based on the value (frequency) and printed the top 5.
The 5 most frequent words used with a ratio of 0.1 is as seen below:



```
summarized_freq = dict()
for word in summarized_text:
    word = word.text
    if word in stopwords or word == '.' or word == ',' or word == '(' or word == ')' or word == '\n':
        continue
    summarized_freq[word] = summarized_freq.get(word, 0) + 1
print(sorted(summarized_freq.items(), key=lambda x: x[1], reverse=True)[:5])
```



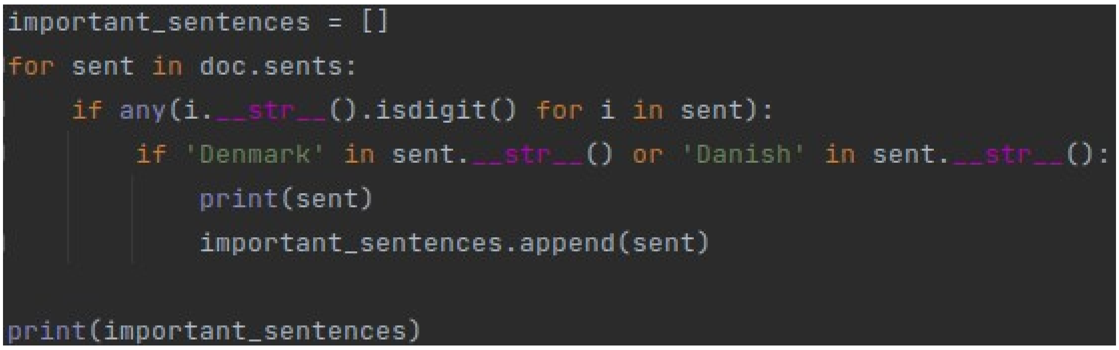
```
[('Denmark', 8), ('Faroe', 3), ('Islands', 3), ('Greenland', 3), ('area', 3)]
```

d)

The numbers in the text include primarily dates and some land area measured in km2. To find out more information about the numbers we can look at the entity labels with nlp. I found out that there are types “DATE”, “QUANTITY”, “MONEY”, and “CARDINAL”.

To find the information that best describes Denmark we could look at the numbers and then the words in that sentence. If the sentence contains the word Denmark or DK we can say that it is more likely to better describe it. The flaw with this is that if it doesn't contain Denmark or DK it would be deemed as not important. The other flaw is that it will take every sentence with numbers and the word ”Denmark” or ”Danish” in it, even if they have no correlation at all.

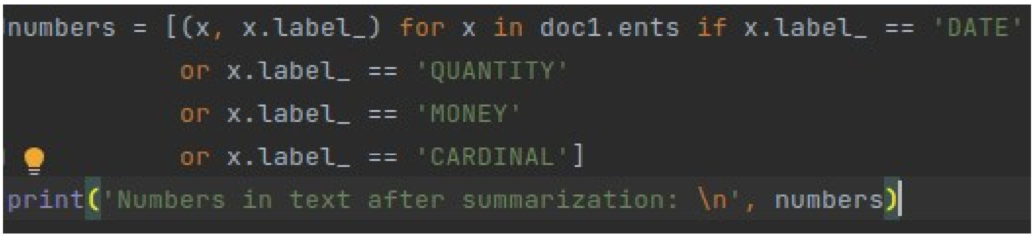
This idea has been coded like this:



```
important_sentences = []
for sent in doc.sents:
    if any(i.__str__().isdigit() for i in sent):
        if 'Denmark' in sent.__str__() or 'Danish' in sent.__str__():
            print(sent)
            important_sentences.append(sent)

print(important_sentences)
```

After this we could look at the words close to the numbers, which are the most likely words that describe what the number is about (did not code this part).

Another way to find the important numbers for Denmark is by summarizing the text, in this experiment i did it with a ratio of 0.2, we look at the entities at the summarized text and find the numbers there. With a few lines of code we can see all the numbers along with their label in list.



```
numbers = [(x, x.label_) for x in doc1.ents if x.label_ == 'DATE'
           or x.label_ == 'QUANTITY'
           or x.label_ == 'MONEY'
           or x.label_ == 'CARDINAL']
print('Numbers in text after summarization: \n', numbers)
```