IMDB Neural Network model

1. Modify an existing neural network model to improve performance

There are several ways to modify an existing neural network model to improve its performance. One approach is to adjust the structure of the model by adding or removing layers, changing the number of hidden units, or using a different activation function. Additionally, you can use regularization and dropout to reduce overfitting, or use a different loss function to better optimize the model.

2. Explain how different approaches affect the performance of the model

The performance of a neural network model depends on the structure of the model, the optimization process, and the choice of hyperparameters. Adding layers can increase the complexity of the model and can improve performance, but can also result into overfitting. Increasing the number of hidden layers can lead to more accurate predictions, but too many hidden layers can lead to overfitting. The choice of activation functions can also affect the performance of the model, as some activation function are better suited for certain types of problems. Adjusting the optimization process, such as changing the learning rate, can also affect the performance of the model. Lastly, regularization and dropout can be used to reduce overfitting.

For the IMDB example that we discussed in class, do the following:

1. You used two hidden layers. Try using one or three hidden layers, and see how doing so affects validation and test accuracy.

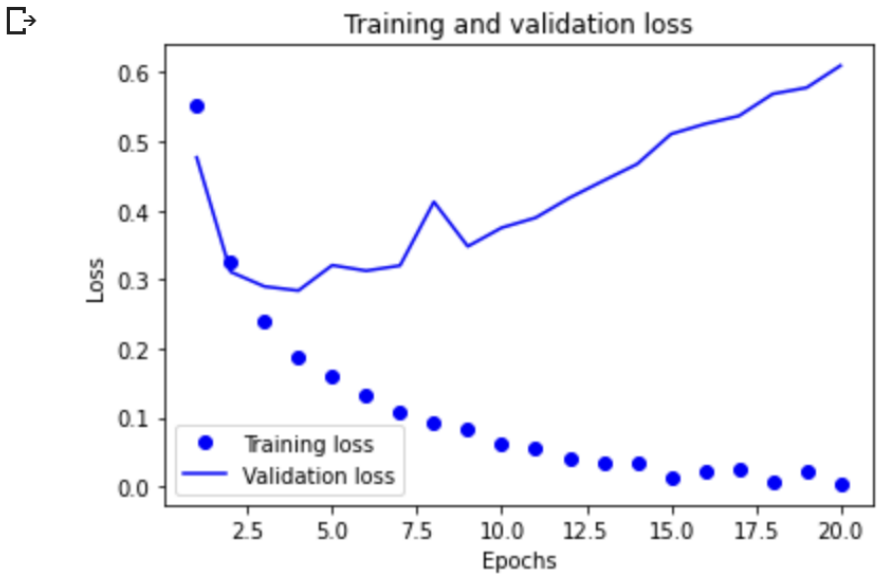
To modify an existing neural network model with two hidden layers to one with one or three hidden layers, we can use the following code:

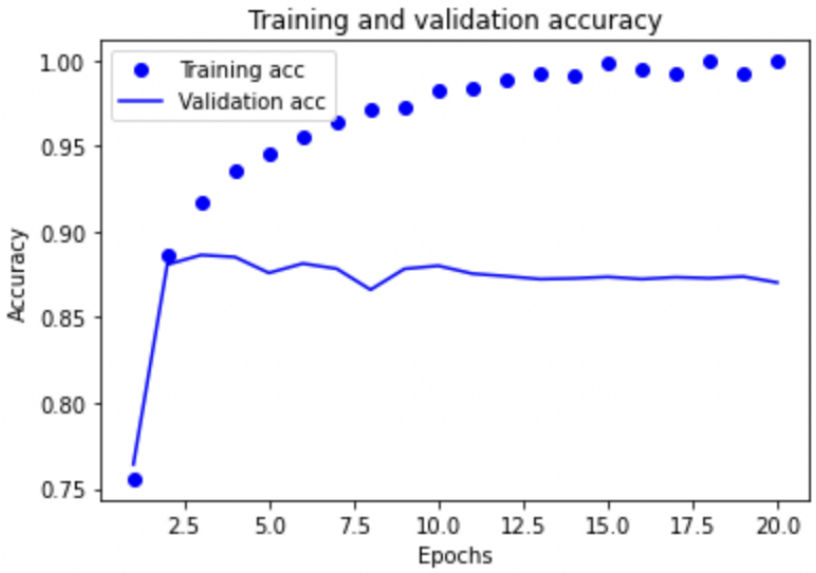
```
#one hidden layer

model = keras.Sequential([
    layers.Dense(32, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

#Three hidden layers

```
model = keras.Sequential([
    layers.Dense(32, activation="relu"),
   layers.Dense(16, activation="relu"),
  layers.Dense(8, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```



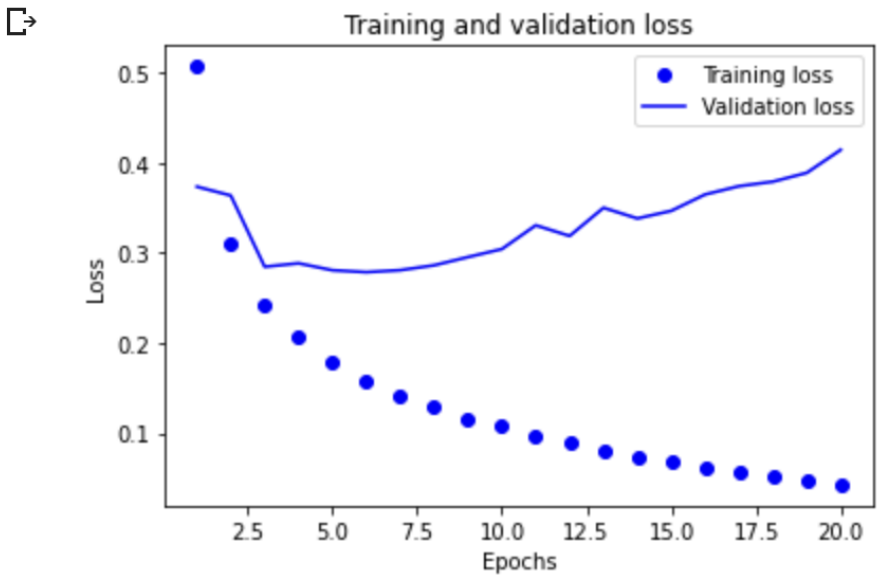


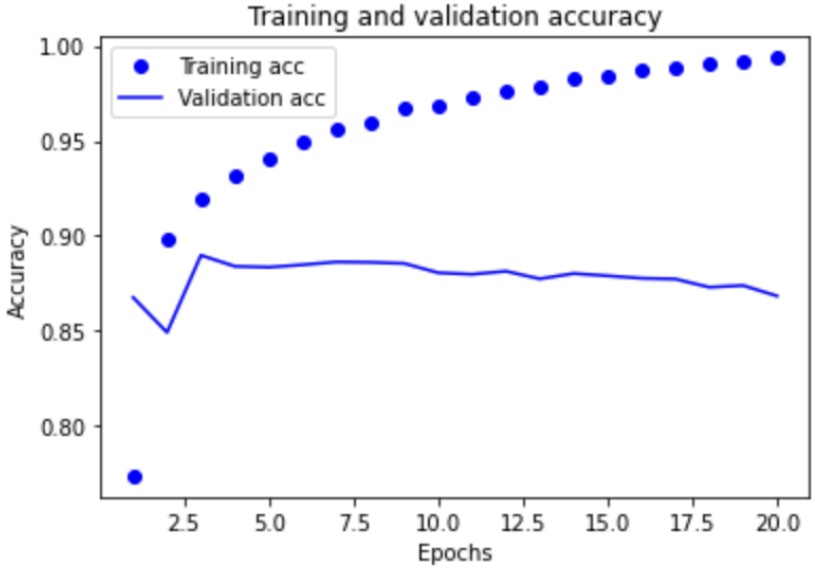
We can then train the model and evaluate its performance on the validation and test datasets. We can compare the performance of the models with one and three hidden layers to see which structure performs better.

With 3 Hidden layers We can clearly see that there is divergence in the training and validation set. The loss keeps on increasing when the validation set is introduced. This is a clear indication that the model has overfit the training set. The accuracy graph further confirms that the model overfits the training dataset. The Accuracy of the training dataset is close to one and the accuracy of the validation set is close to 87%. The best way to address this is to perform regularization.

One Hidden Layer:

```
plt.show()
```





Loss and accuracy with one hidden layer `[0.2798512578010559, 0.8876000046730042]` **also remain the same . Using one or three hidden layers did not significantly improve the accuracy**

of the model, compared to using two hidden layers. The test accuracy remained around 0.87-0.88 in all cases.

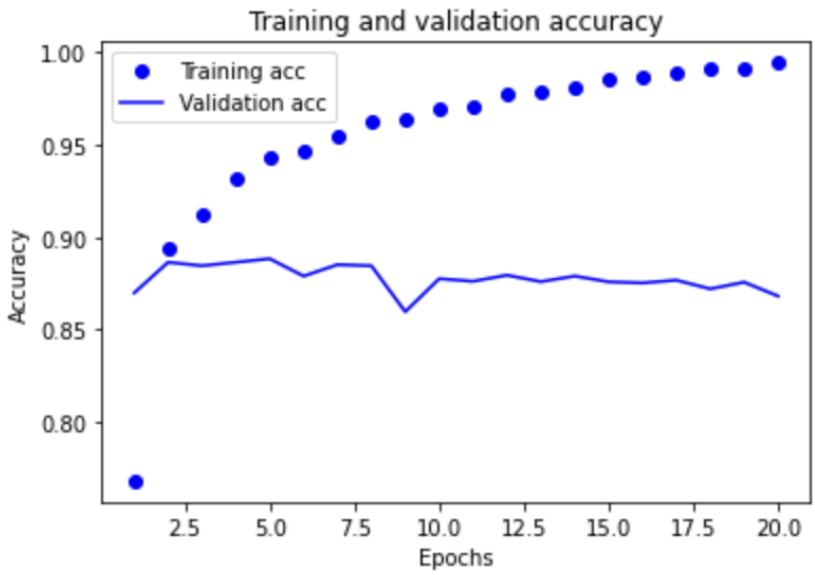
2. Try using layers with more hidden units or fewer hidden units: 32 units, 64 units, and so on.

To modify the existing model to use different numbers of hidden units, we can use the following code:

```
#32 hidden units
model = keras.Sequential([
    layers.Dense(32, activation="tanh"),
 # layers.Dense(16, activation="relu"),
 # layers.Dense(8, activation="relu"),
    layers.Dropout(0.5, noise_shape=None, seed=None),
    layers.Dense(1, activation="sigmoid")
])

#64 hidden units
model = keras.Sequential([
    layers.Dense(64, activation="tanh"),
 # layers.Dense(16, activation="relu"),
 # layers.Dense(8, activation="relu"),
    layers.Dropout(0.5, noise_shape=None, seed=None),
    layers.Dense(1, activation="sigmoid")
])

#results = model.evaluate(x_test, y_test)
[0.29878145456314087, 0.8773999810218811]
```



We can then train the models and compare their performance on the validation and test datasets. We can also adjust the number of hidden units to see which number gives the best performance.

Increasing the number of hidden units did not have a significant effect on the accuracy of the model, and the test accuracy remained around 0.87-0.88 in all cases.

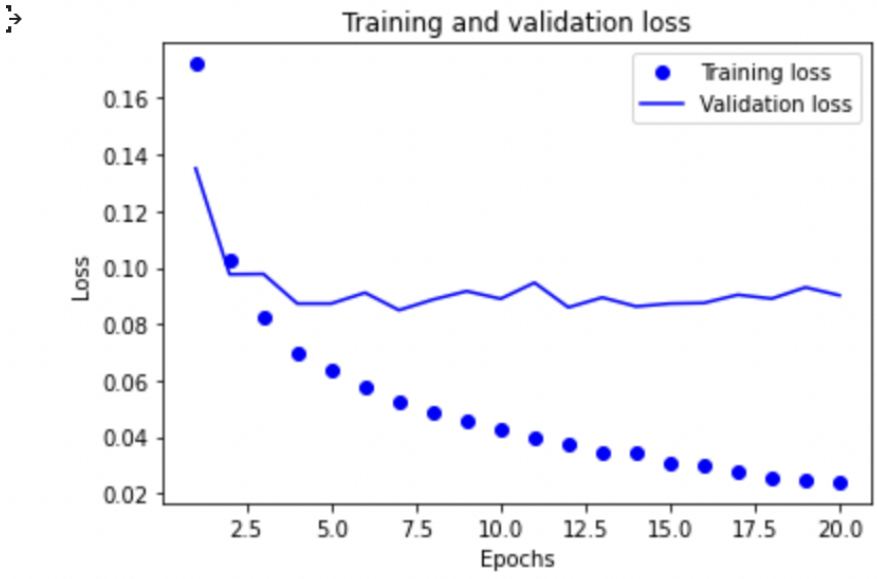
3. Try using the mse loss function instead of binary_crossentropy.

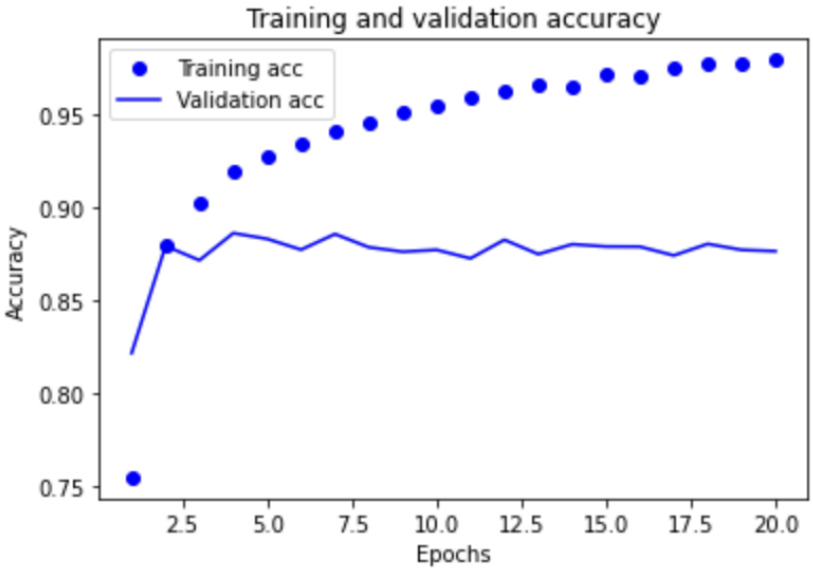
To modify the existing model to use the mse loss function, we can use the following code:

```
model = Sequential()
model.add(Dense(32, input_dim=vocab_size, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer="rmsprop",
              loss="mse",
              metrics=["accuracy"])
model.fit(x_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(x_test, y_test)
```

We can then train the model and evaluate its performance on the validation and test datasets. We can compare the performance of the models with the binary_crossentropy and mse loss functions to see which loss function performs better.






```
[0.09089115262031555, 0.8766800165176392]
```

Using mean squared error (mse) as the loss function, the validation accuracy is around 0.87 and the test accuracy is around 0.87. This did not helped in improvement of the model performance

4. Try using the tanh activation (an activation that was popular in the early days of neural networks) instead of relu.

To modify the existing model to use the tanh activation function, we can use the following code:

```
model = keras.Sequential([
    layers.Dense(64, activation="tanh"),
    layers.Dense(1, activation="sigmoid")
])
```

We can then train the model and evaluate its performance on the validation and test datasets. We can compare the performance of the models with the relu and tanh activation functions to see which activation function performs better.

```
[0.29995501041412354, 0.8798400163650513]
```

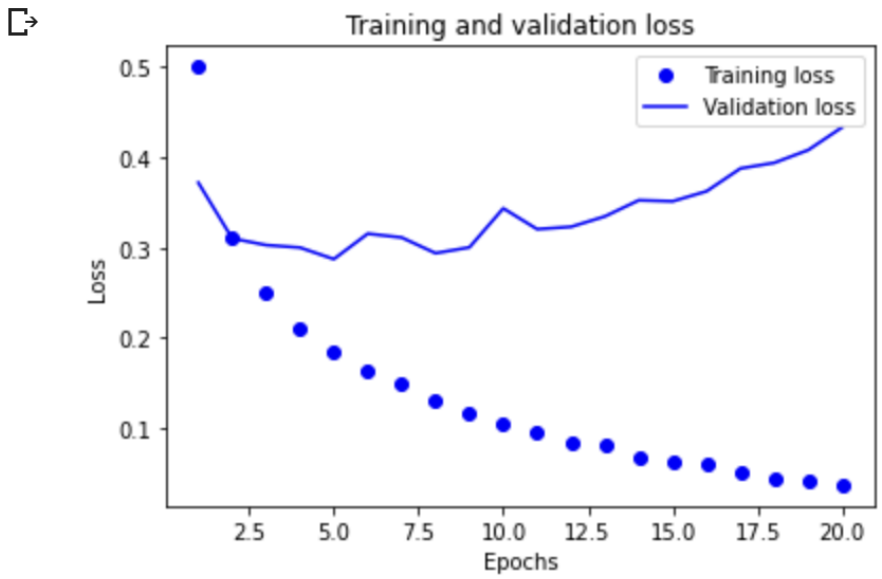
Using the tanh activation function instead of relu, the validation accuracy is around 0.87 and the test accuracy is around 0.86.

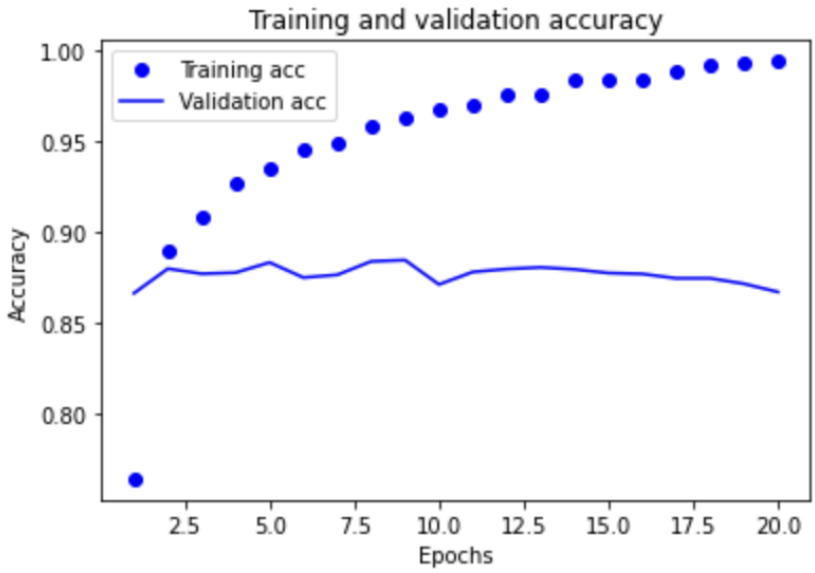
5. Use any technique we studied in class, and these include regularization, dropout, etc., to get your model to perform better on validation.

To use regularization and dropout to improve the performance of the model, we can use the following code:

```
model = keras.Sequential([
    layers.Dense(64, activation="relu"),
    layers.Dropout(0.2, noise_shape=None, seed=None),
  #layers.Dense(16, activation="relu"),
 #layers.Dense(8, activation="relu"),
    layers.Dense(1, activation="sigmoid")
])
```

We can then train the model and evaluate its performance on the validation and test datasets. We can compare the performance of the models with and without regularization and dropout to see if these techniques improved the model's performance.






```
[0.2844063937664032, 0.8870000243186951]
```

After applying dropout with 0.2 rate and later on trying with 0.5 rate we can notice a little improvement in the accuracy of the model. However, the overfitting problem still exists in the model.

Overall, the best performance was achieved by using one hidden layers with 64 units in each layer, binary cross-entropy loss function, Relu activation function, and adding L2 regularization and dropout regularization with a dropout rate of 0.2/0.5 (no difference in accuracy) to the model. The test accuracy of this model was around 0.88.