Deep Learning - Assignment 2

MLP in Keras

Introduction

This project will predict a class (0, 1) using multi-layer perceptron's built in python using Keras. A multilayer perceptron (MLP) is a type of feedforward neural network that consist of at least three layers of nodes and uses supervised learning techniques for training. Eleven MLP networks were built using the fit class data set (fit.csv) and tested using a test data set (test.csv).

Part 1

Model specifications

One MLP was built with the following specifications:

1. network specifications:
   - Input dimension of 8
   - One dense hidden layer with 3 nodes, sigmoid activation function
   - One dense output layer with 2 nodes, softmax activation function

2. compile specifications:
   - Use the SGD optimizer: learning rate of 0.3, momentum of 0.2, Set decay to 0.0, nesterov to False
   - Use binary cross entropy for your loss function
   - Use accuracy as a performance metric

3. Training specifications:
   - Use 10% of the data for validation
   - Train for at least 500 epochs
   - Use a batch size of 100

Results

Part 1	Loss	Accuracy	Val Loss	Val Accuracy	Test Accuracy
MLP	0.3306	0.8343	0.1651	0.9474	86.17%

Table 1: Part 1 MLP Results

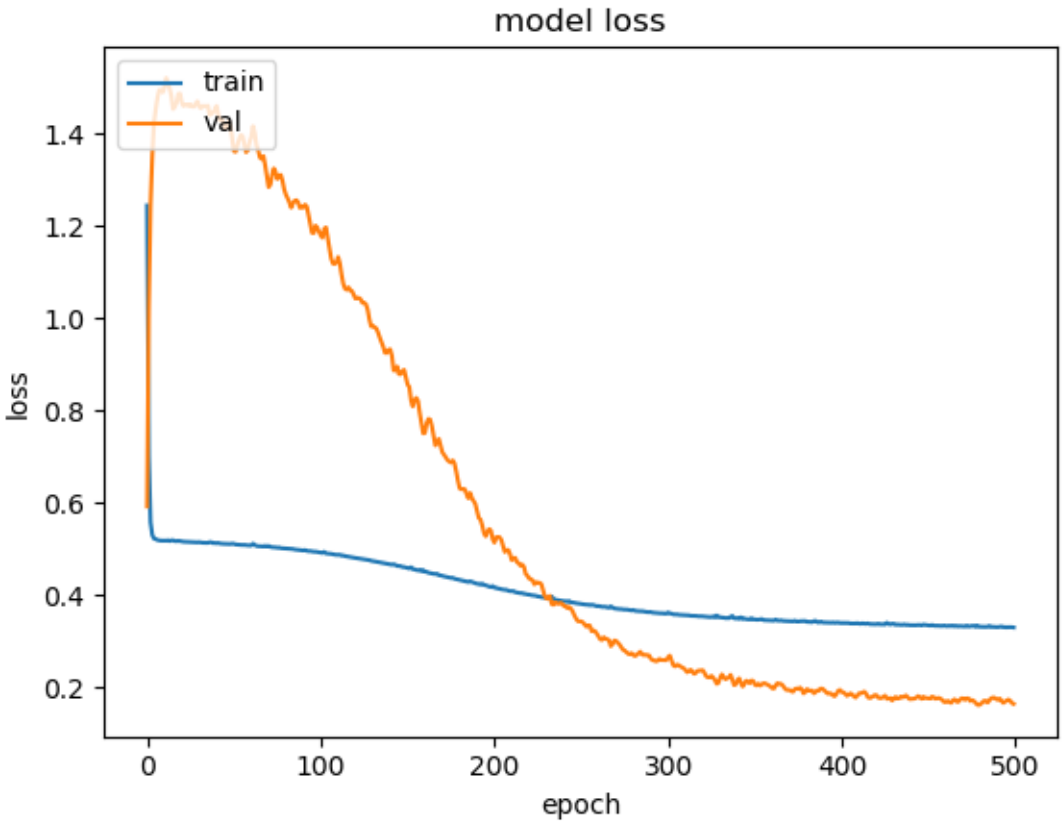

Table 2: Part 1 Model Loss

Analysis

The MLP made in part 1 has a high accuracy of 86.17%. We can determine this is a well-trained model based on the model loss plot. If the training loss is much lower than validation loss, then this means the network might be overfitting. If the training loss is about equal to the validation loss, then the model is underfitting.

Part 2

Model specifications

Six separate MLPs were built with the following changes to the MLP from Part 1:

1. hidden layer activation of Rectified Linear Unit ("ReLu").
2. hidden layer with 1, 2, 4 and 5 nodes.
3. two hidden layers, each with 3 nodes (sigmoid activation was used for the new hidden layer)

Results

Part 2	Loss	Accuracy	Val Loss	Val Accuracy	Test Accuracy
ReLu	0.3028	0.8521	0.0652	1.0000	84.04%
1 hidden node	0.3384	0.8343	0.2356	0.8947	88.30%
2 hidden nodes	0.3222	0.8284	0.1514	0.9474	86.17%
4 hidden nodes	0.3214	0.8343	0.1557	0.9474	87.23%
5 hidden nodes	0.3267	0.8343	0.1538	0.9474	87.23%
2 hidden layers	0.4925	0.7870	1.2389	0.0000	70.21%

Table 3: Part 2 MLP results

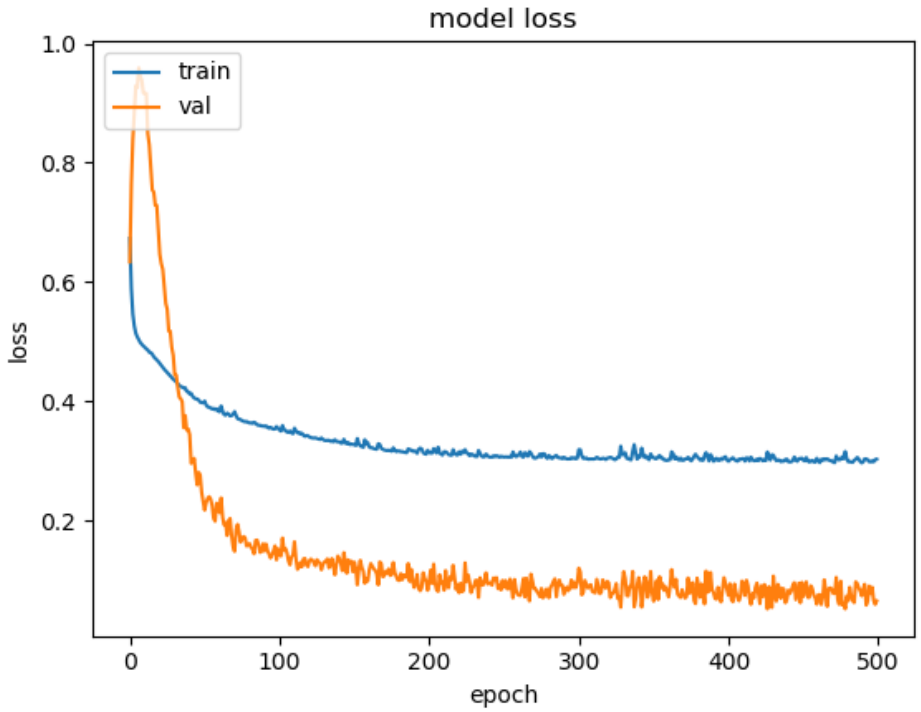

Table 4: Rectified Linear Unit Loss function Model Loss

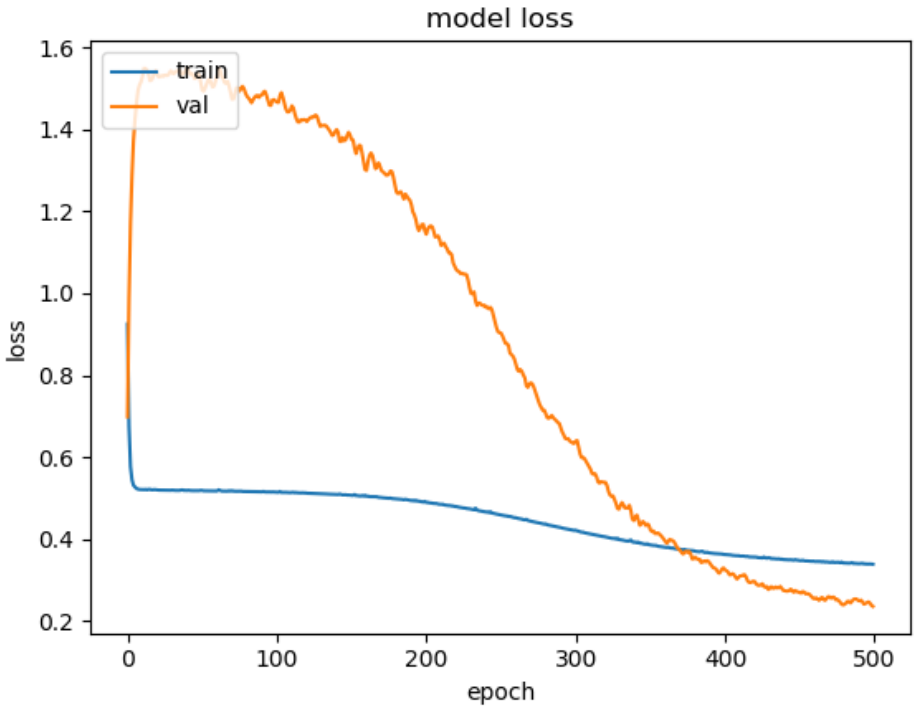

Table 5: 1 Hidden Layer Node Model Loss

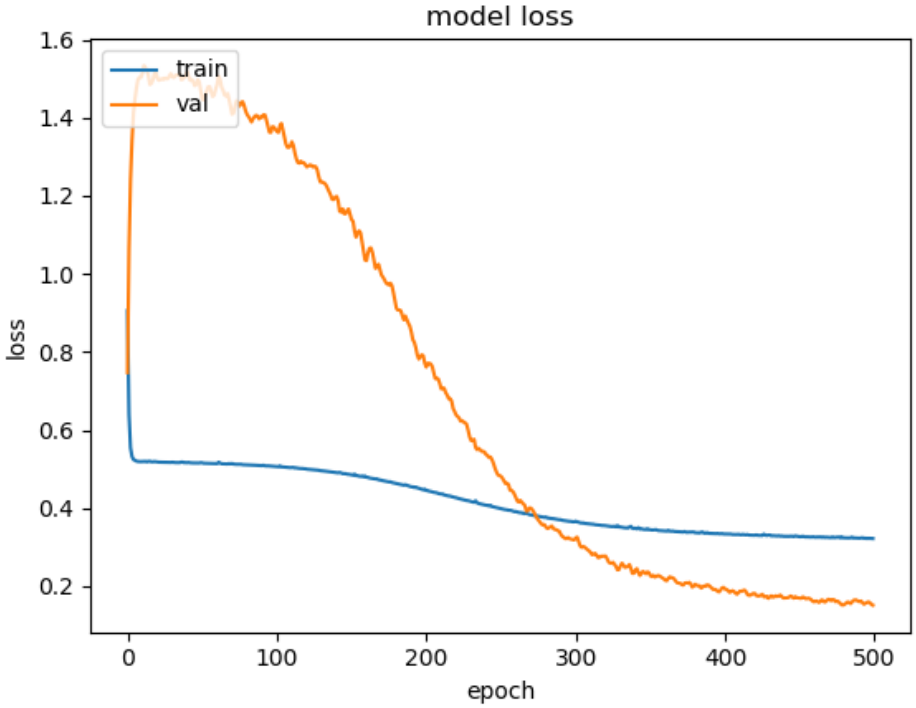

Table 6: 2 Hidden Layer Nodes Model Loss

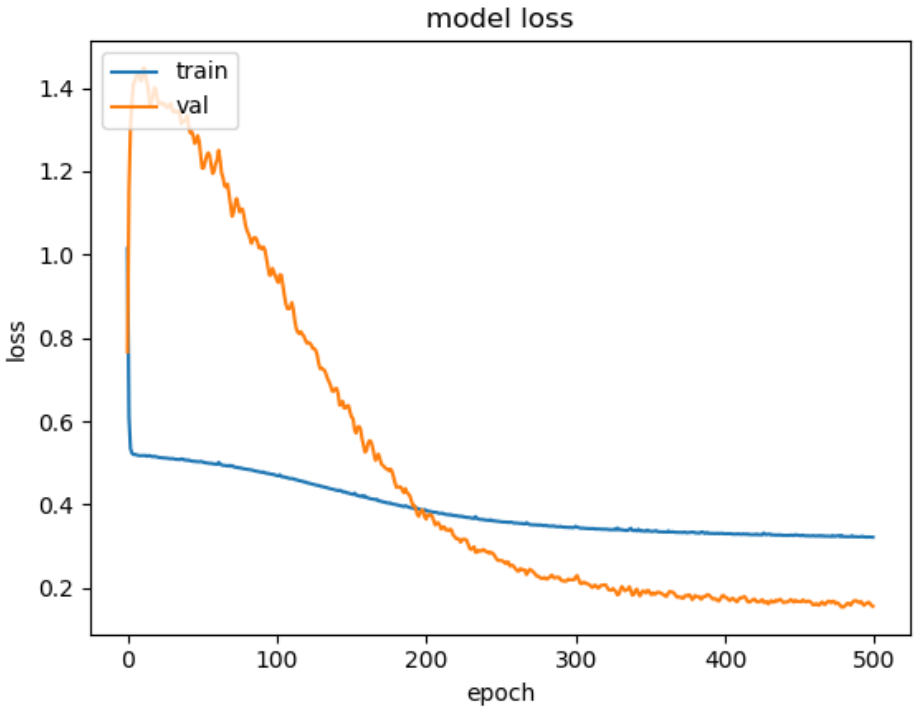

Table 7: 4 Hidden Layer Nodes Model Loss

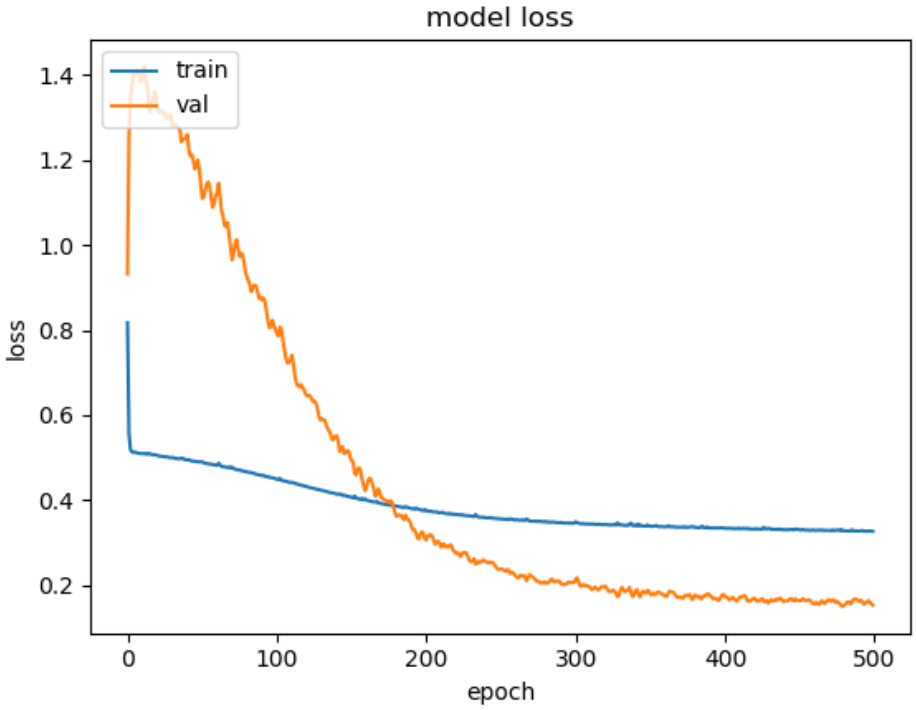

Table 8: 5 Hidden Layer Nodes Model Loss

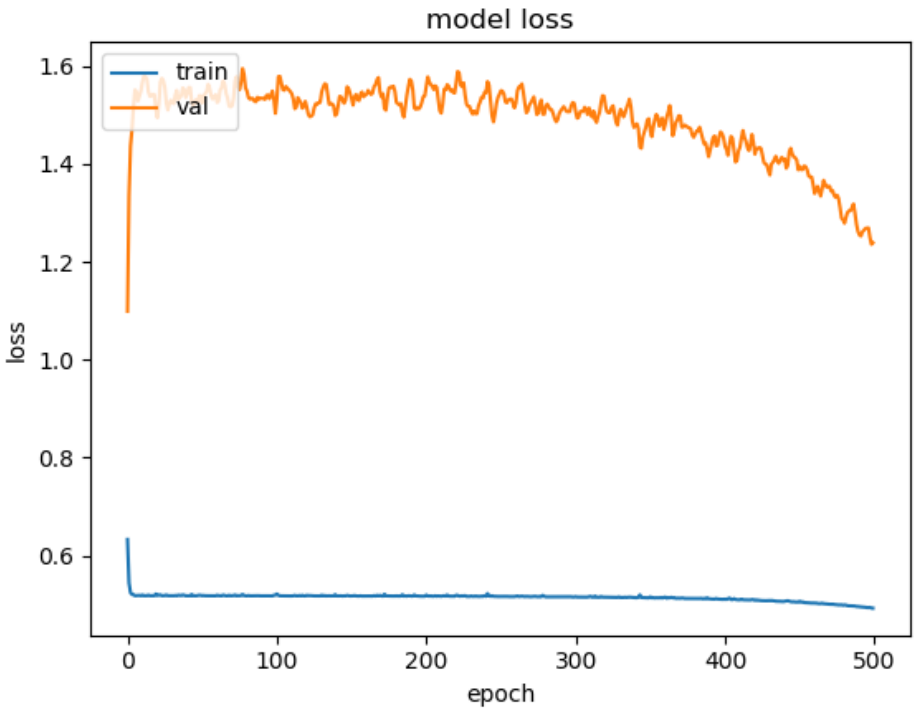

Table 9: 2 Hidden Layers Model Loss

Comparison

Rectified Linear Unit activation trains the model a lot quicker than sigmoid activation. With an accuracy of 84.04%, ReLu is slightly less accurate then the model from part 1. Because ReLu trains quicker then other activation function, it is better used on larger and complex datasets.

The model that had the most accuracy had one node in the hidden layer. With a difference of 2.13%, this is not a big difference. when there were four and five nodes in the hidden layer they performed less then the model with one node and more than the models with two and three nodes. The model from part 1 with three nodes was the less accurate out of these five models. It is safe to say that even though one hidden node seems simple it may be the best model for a dataset.

The model with two hidden layers, with accuracy of 70.21%, was the less accurate compare to the other models thus far. This model has a very high validation compared to the training loss showing that the model is overfitting. The validation loss continued to decrease after each epoch and it may end up being a well-trained model but not for much more epochs.

Part 3

Model specifications

Four separate MLPs were built with the following changes to the MLP from Part 1:

1. Learning rate of 0.2, 0.1 and 0.01.
2. 'adam' optimizer

Results

Part 3	Loss	Accuracy	Val Loss	Val Accuracy	Test Accuracy
LR = 0.2	0.3510	0.8166	0.2147	0.8947	88.30%
LR = 0.1	0.4451	0.7929	0.7341	0.5263	80.85%
LR = 0.01	0.5170	0.7870	1.4963	0.0000	70.21%
Adam optimizer	0.2553	0.8876	0.3273	0.9474	81.91%

Table 10: Part 3 MLP results

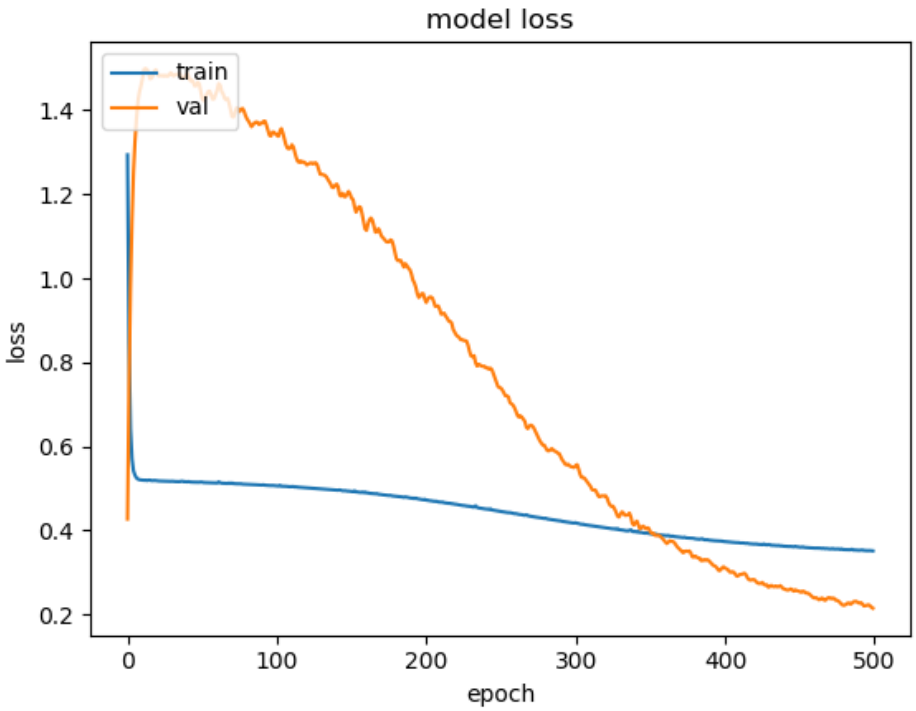

Table 11: Learning Rate of 0.2 Model Loss

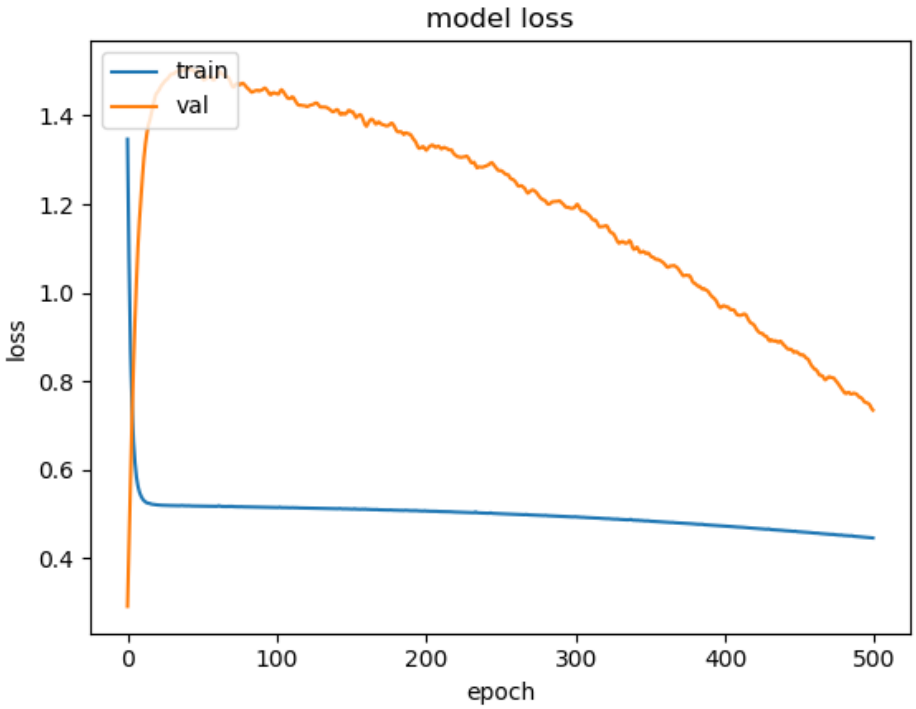

Table 12: Learning Rate of 0.1 Model Loss

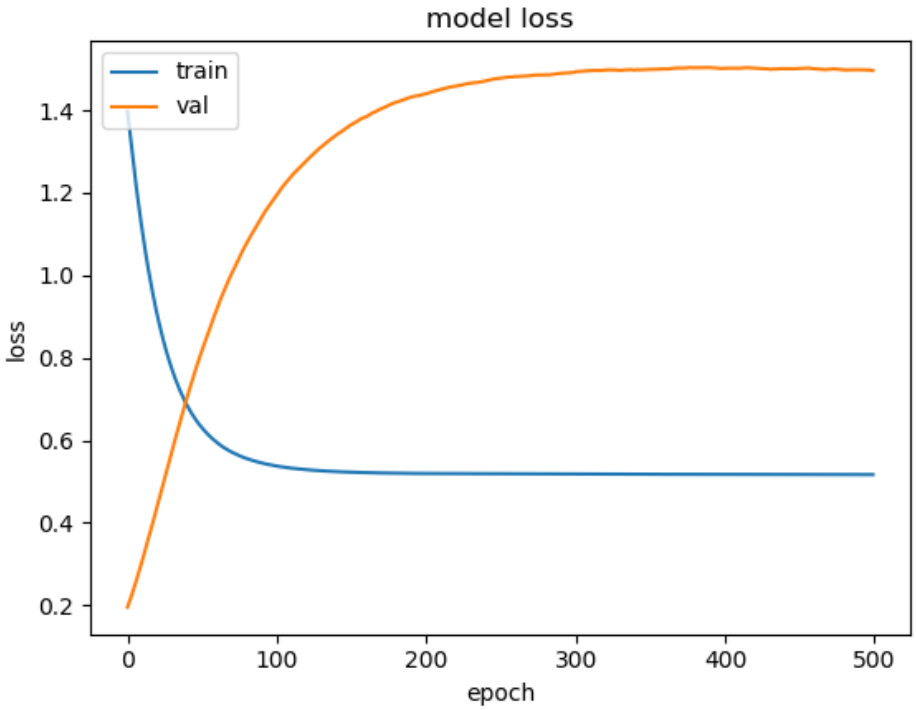

Table 13: Learning Rate of 0.01 Model Loss

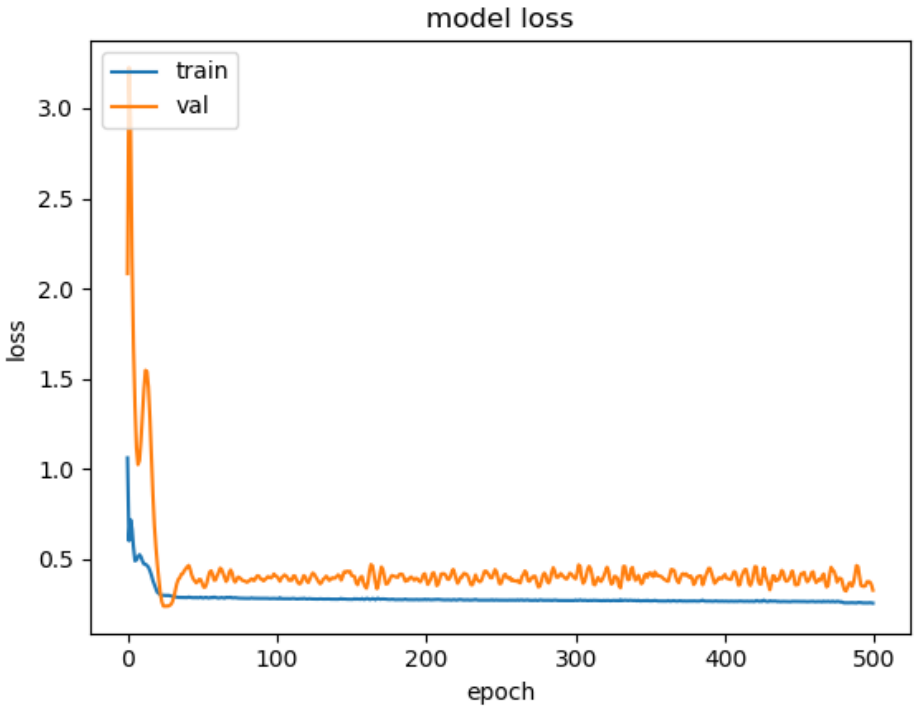

Table 14: 'adam' Optimizer Model Loss

Part 3	Loss	Accuracy	Val Loss	Val Accuracy	Test Accuracy
LR = 0.2	0.3510	0.8166	0.2147	0.8947	88.30%
LR = 0.1	0.4451	0.7929	0.7341	0.5263	80.85%
LR = 0.01	0.5170	0.7870	1.4963	0.0000	70.21%
Adam optimizer	0.2553	0.8876	0.3273	0.9474	81.91%

Comparison

Model with learning rate of 0.2 had the same accuracy as the model with one hidden node in part 2. These models were the top performing in the project.

The model with learning rate of 0.01 had the same accuracy as the model with two hidden layers. These models were the least performing models in the project. Along with the model with learning rate of 0.1, these models have high validation compared to the training loss showing that the model is overfitting. The validation loss continued to decrease after each epoch and it may end up being a well-trained model with more epochs.

The model with Adam optimizer, having an accuracy of 81.91%, was also a low performing model compared to others in this project and that in part 1. Adam is a good optimizer algorithm as it has little memory requirements, large data, and noisy data. Like RuLu, this optimizer trains the model quicker than the other optomizers.

Conclusion

Part 2 shows that the more node in a hidden layer does not necessarily produce a better model. The model with one node out performed all other models in the project and performed equally as model with learning rate 0.2.

Rectified Linear Unit activation function and Adam optimizer algorithm performance was average but the models built quicker than any others. These models are better for larger more noisy data sets.

Models with smaller learning rates and more hidden layers did not perform well. The loss trend of all these models suggest that if trained with more epochs then these models could possible performed as good as the rest. However, because this requires more time to train it may be unnecessary and would be better to use a simpler model.

Code

```
#imports
import keras
from keras.models import Sequential
from keras.layers import Dense, Activation
from keras import optimizers
from keras import backend as K
from keras.utils import np_utils
import matplotlib.pyplot as plt
import numpy as np

# Global Parameters
# random seed
np.random.seed(1234)
# mlp model
model = 'mlp'
# batch size and number of training epochs
batch_size = 100
nb_epoch = 500

 # mlp model
def mlp_model():
 model = Sequential()
 model.add(Dense(3, input_dim=8))
 model.add(Activation('sigmoid'))
 model.add(Dense(2))
 model.add(Activation('softmax'))
 return model

 # load mnist data, format for mlp model
def load_data():
 data_dir = "C:/Users/camer/Desktop/Deep Learning/Assignment2/"
 train = np.genfromtxt(data_dir+"fit.csv", delimiter=',')
 test = np.genfromtxt(data_dir+"test.csv", delimiter=',')
 X_train, y_train = train[:,:8], train[:,8:]
```

```
  X_test, y_test = test[:,:8], test[:,8:]
  X_train = X_train.astype('float32')
  X_test = X_test.astype('float32')
  Y_train = np_utils.to_categorical(y_train, 2)
  Y_test = np_utils.to_categorical(y_test, 2)
  return X_train, Y_train, X_test, Y_test

  # plots training and validation loss
def plot_losses(history):
  plt.plot(history.history['loss'])
  plt.plot(history.history['val_loss'])
  plt.title('model loss')
  plt.ylabel('loss')
  plt.xlabel('epoch')
  plt.legend(['train', 'val'], loc='upper left')
  plt.show()

# make mlp model
x_train, y_train, x_test, y_test = load_data()
model = mlp_model()

# define optimizer
sgd = optimizers.SGD(lr=0.3, momentum=0.2, decay=0.0, nesterov=False)

# compile model and specifications
model.compile(loss='binary_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

# train model on training data
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=nb_epoch,
          verbose=1, validation_split=0.1)

# scores model on test data for chosen metric (accuracy)
score = model.evaluate(x_test, y_test, verbose=0)

# print accuracy
print(score[1])

# plots loss for training and validation data
plot_losses(history)

# ends session, avoids potential error on program exit
K.clear_session()
```