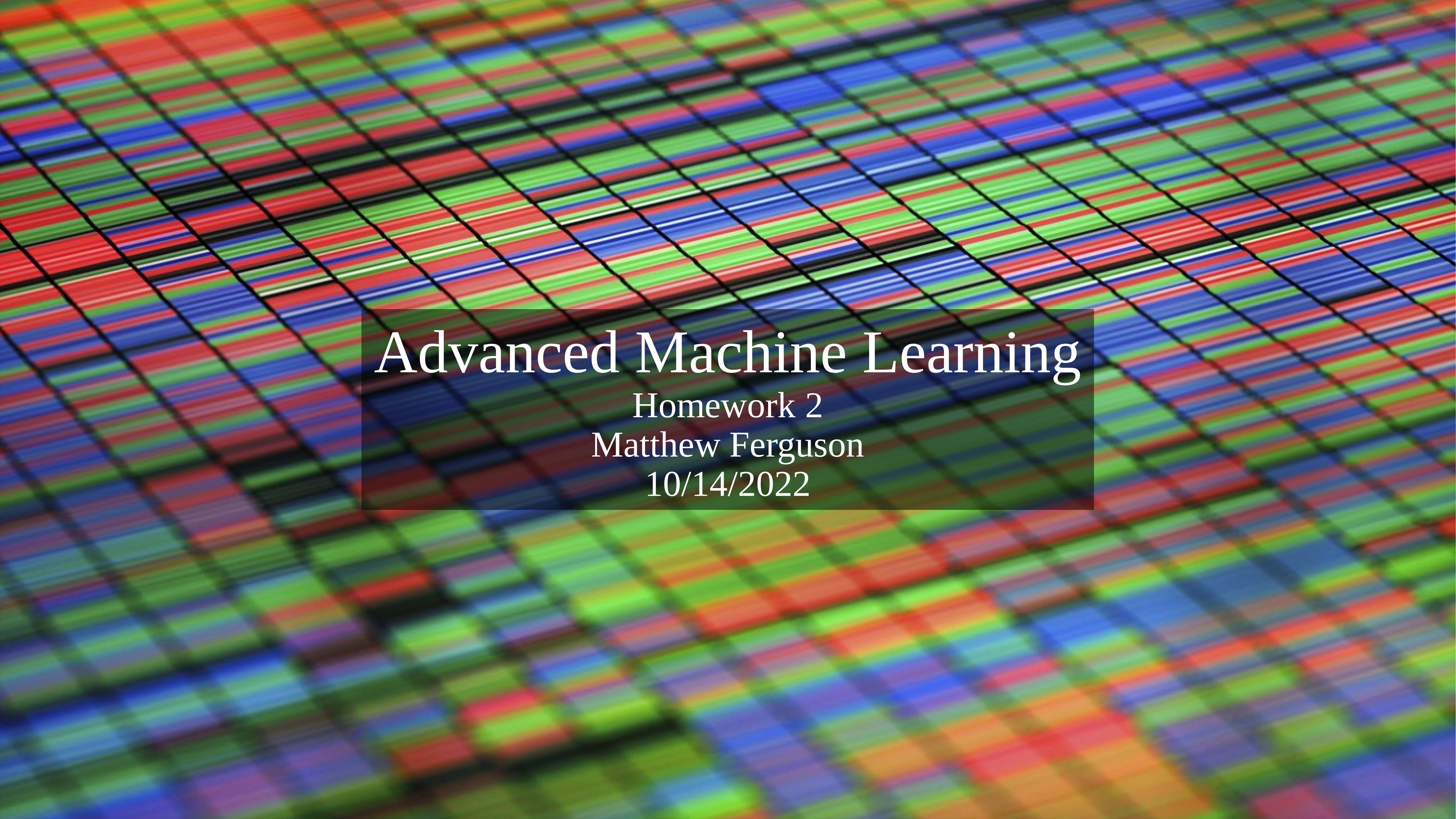


Advanced Machine Learning

Homework 2

Matthew Ferguson

10/14/2022

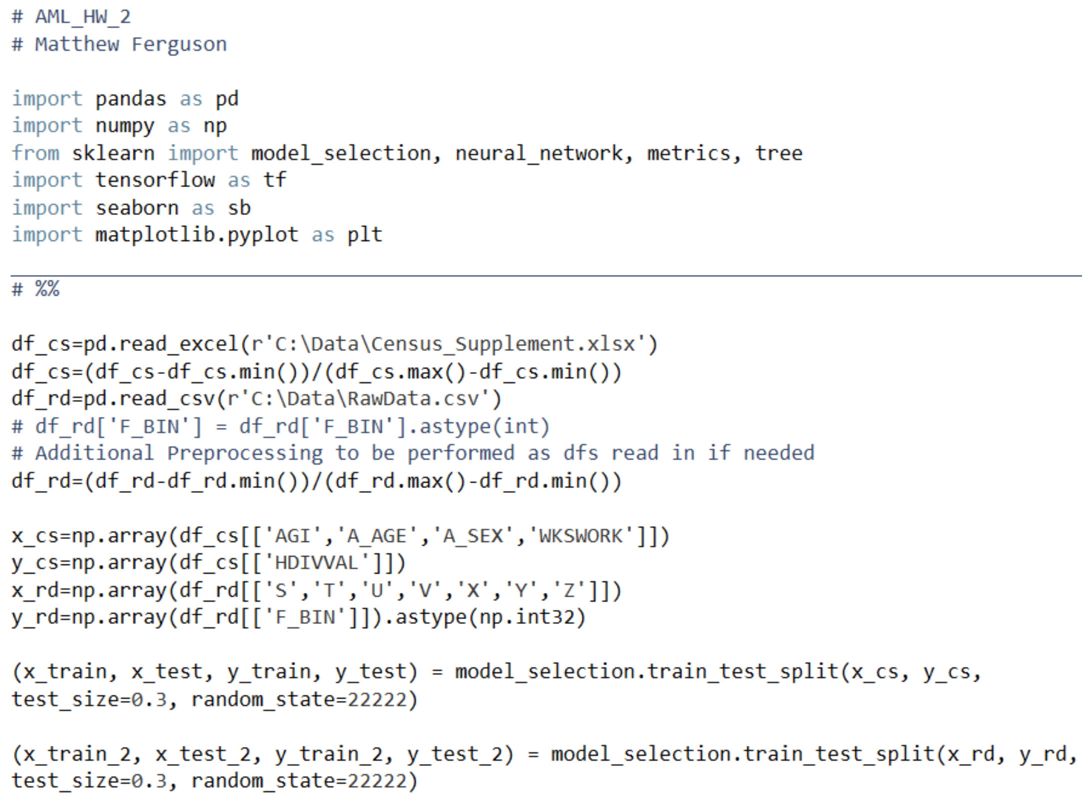
Requirements Digest

1. Create a MLP regression model using sklearn on the census dataset.
   - Records node weights after each epoch
   - Create two plots. Loss on training and test set versus epoch and weights versus epoch.
   - Generate plots for several ANNs: hidden nodes 3 to 6, and L2 regularization coefficients of 0,0.0001, 0.001, and 0.01. Display and analyze.
2. Create a MLP classifier on the raw dataset.
   - Experiment with network sizes and other parameters to find the maximum accuracy.
   - Examine the dataset and determine the maximum accuracy.
3. Use TensorFlow to develop a regression MLP on the census dataset.
   - Use any desired network size, find the best model and justify.
   - Report the final model performance using performance metrics and plot a learning history.

Preprocessing Code

We see the necessary library imports for the assignment are made in cell 0. Pandas, and NumPy are used for data manipulation. Sklearn and TensorFlow are used for training machine learning models. Seaborn and matplotlib plot our resulting performance metrics.

We proceed to cell 1 and load two data frames, one for the census data and one for the raw data. We perform some preprocessing by changing the target labels in the raw data from ‘FALSE’ and ‘TRUE’ to 0 and 1 so that is easily interpreted as binary by sklearn/python. We inspected the dataset and did not find any missing or erroneous values, therefore additional preprocessing was not necessary, but we would have accomplished it here if needed. We normalize our datasets and then we extract the desired features and targets of these datasets. We split them into test and training sets for future model training.



```
# AML_HW_2
# Matthew Ferguson

import pandas as pd
import numpy as np
from sklearn import model_selection, neural_network, metrics, tree
import tensorflow as tf
import seaborn as sb
import matplotlib.pyplot as plt
```

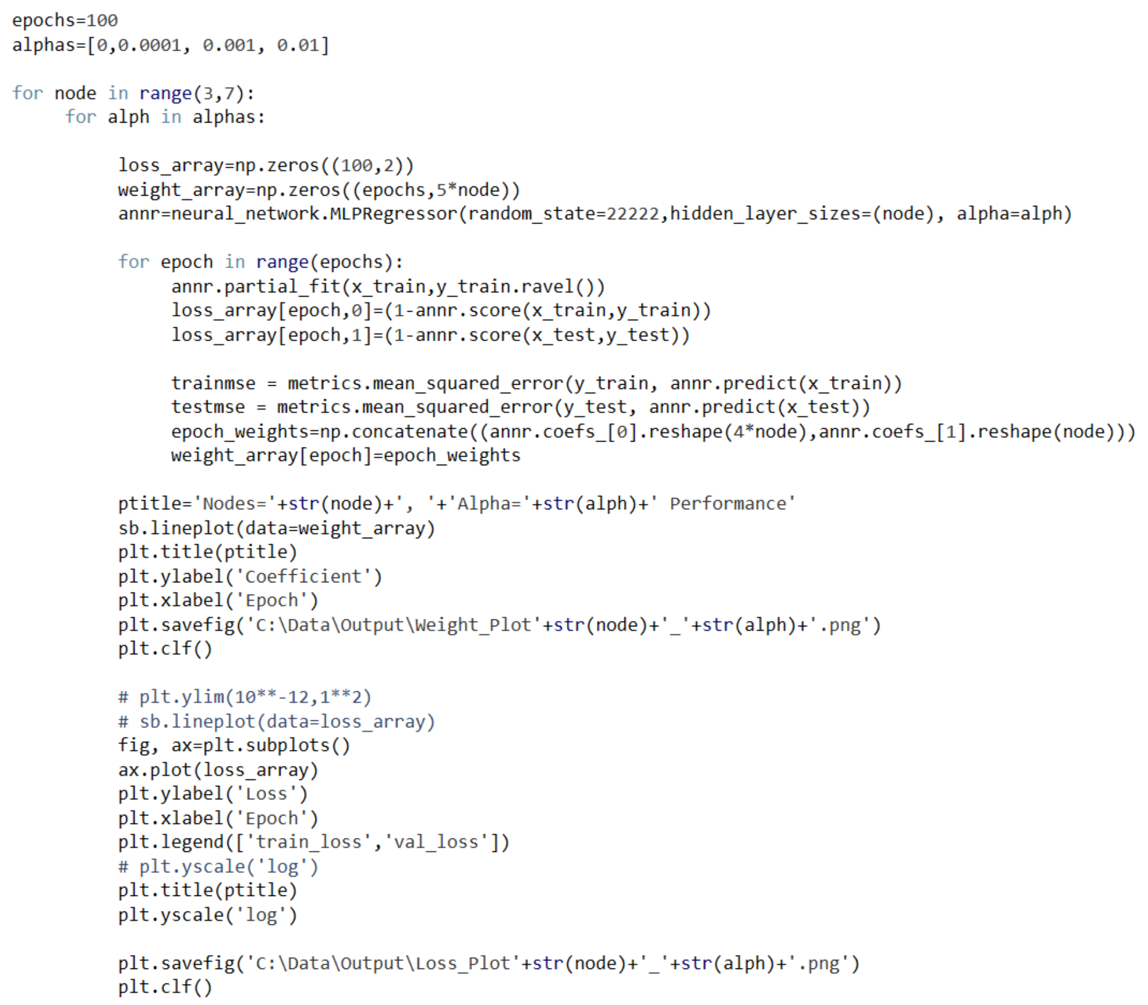
```
# %%

df_cs=pd.read_excel(r'C:\Data\Census_Supplement.xlsx')
df_cs=(df_cs-df_cs.min())/(df_cs.max()-df_cs.min())
df_rd=pd.read_csv(r'C:\Data\RawData.csv')
# df_rd['F_BIN'] = df_rd['F_BIN'].astype(int)
# Additional Preprocessing to be performed as dfs read in if needed
df_rd=(df_rd-df_rd.min())/(df_rd.max()-df_rd.min())

x_cs=np.array(df_cs[['AGI','A_AGE','A_SEX','WKSWORK']])
y_cs=np.array(df_cs[['HDIVVAL']])
x_rd=np.array(df_rd[['S','T','U','V','X','Y','Z']])
y_rd=np.array(df_rd[['F_BIN']]).astype(np.int32)

(x_train, x_test, y_train, y_test) = model_selection.train_test_split(x_cs, y_cs,
test_size=0.3, random_state=22222)

(x_train_2, x_test_2, y_train_2, y_test_2) = model_selection.train_test_split(x_rd, y_rd,
test_size=0.3, random_state=22222)
```

Part 1 Code

For part 1 we developed a set of regression neural networks. We iterate across node sizes of 3 to 6 for a single hidden layer and we iterate over L2 regularization values of 0, 0.001, 0.001, and 0.01.

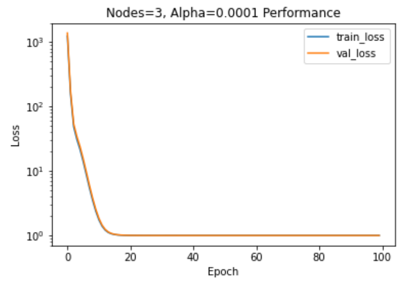
We create arrays to hold our loss values and neural net weights. We create a neural network regressor object and then begin partial fitting. We fit over 100 epochs and calculate the relevant performance metrics. These metrics are stored in arrays that are updated during each epoch.

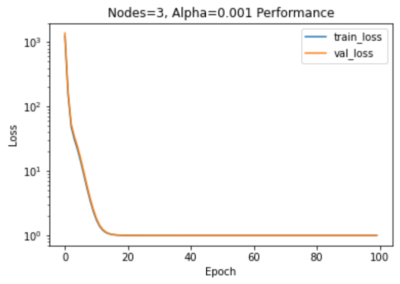
After fitting is complete for a neural network architecture, we output our weight and loss plots, which are shown next.

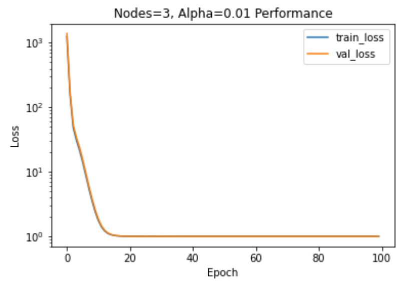


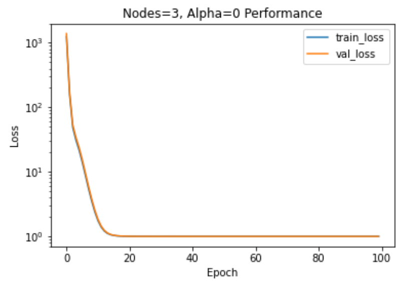
```
epochs=100
alphas=[0,0.0001, 0.001, 0.01]

for node in range(3,7):
    for alph in alphas:

        loss_array=np.zeros((100,2))
        weight_array=np.zeros((epochs,5*node))
        annr=neural_network.MLPRegressor(random_state=22222,hidden_layer_sizes=(node), alpha=alph)

        for epoch in range(epochs):
            annr.partial_fit(x_train,y_train.ravel())
            loss_array[epoch,0]=(1-annr.score(x_train,y_train))
            loss_array[epoch,1]=(1-annr.score(x_test,y_test))

            trainmse = metrics.mean_squared_error(y_train, annr.predict(x_train))
            testmse = metrics.mean_squared_error(y_test, annr.predict(x_test))
            epoch_weights=np.concatenate((annr.coefs_[0].reshape(4*node),annr.coefs_[1].reshape(node)))
            weight_array[epoch]=epoch_weights

        ptitle='Nodes='+str(node)+', '+'Alpha='+str(alph)+' Performance'
        sb.lineplot(data=weight_array)
        plt.title(ptitle)
        plt.ylabel('Coefficient')
        plt.xlabel('Epoch')
        plt.savefig('C:\Data\Output\Weight_Plot'+str(node)+'_'+str(alph)+'.png')
        plt.clf()

        # plt.ylim(10**-12,1**2)
        # sb.lineplot(data=loss_array)
        fig, ax=plt.subplots()
        ax.plot(loss_array)
        plt.ylabel('Loss')
        plt.xlabel('Epoch')
        plt.legend(['train_loss','val_loss'])
        # plt.yscale('log')
        plt.title(ptitle)
        plt.yscale('log')

        plt.savefig('C:\Data\Output\Loss_Plot'+str(node)+'_'+str(alph)+'.png')
        plt.clf()
```

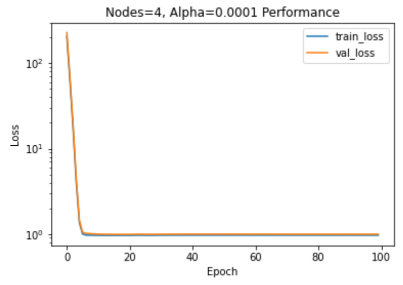
Part 1 Loss Plots

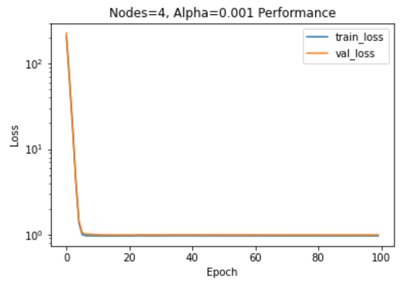


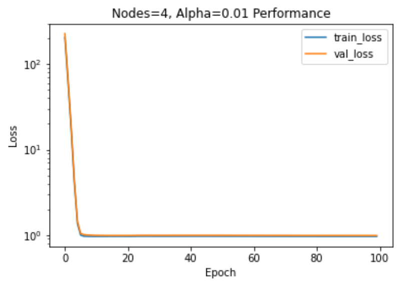


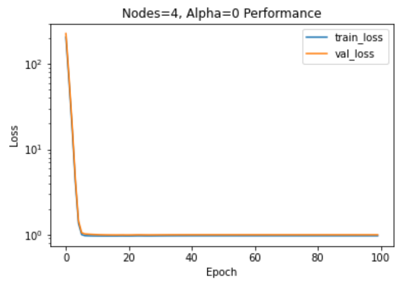


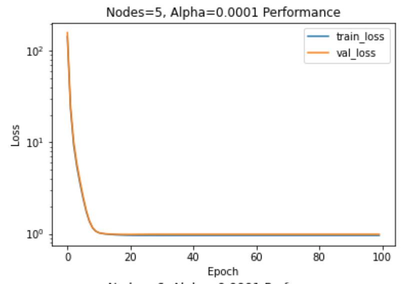


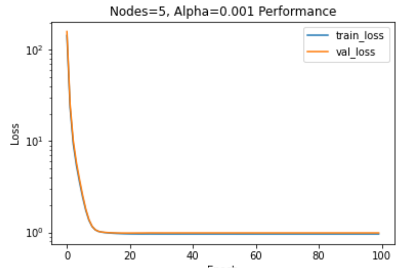


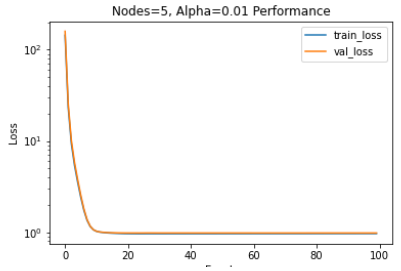


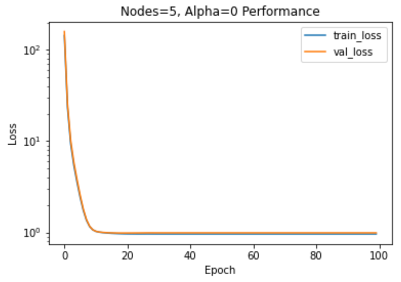


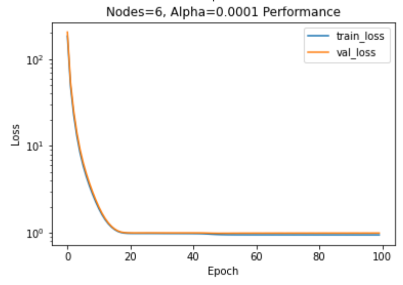


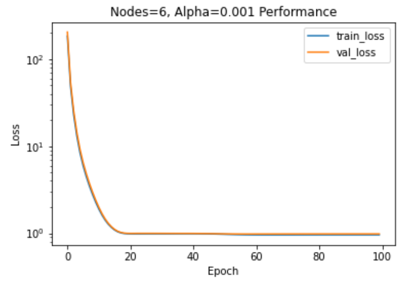


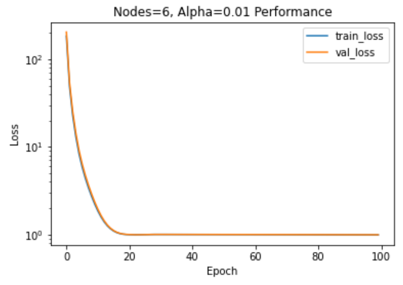


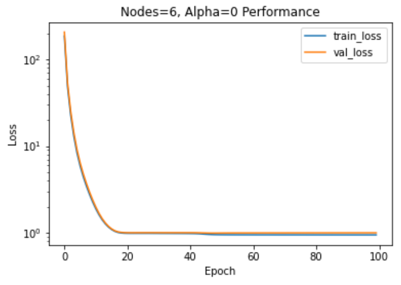






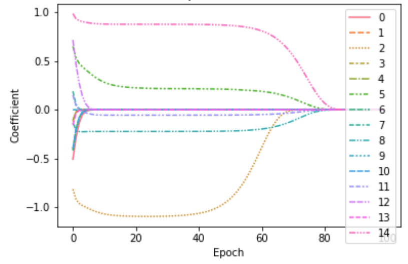




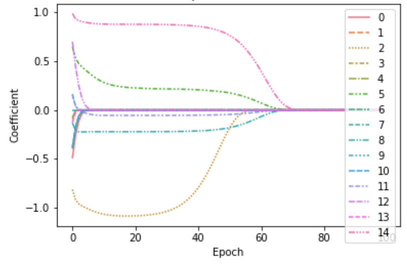


Part 1 Weight Plots

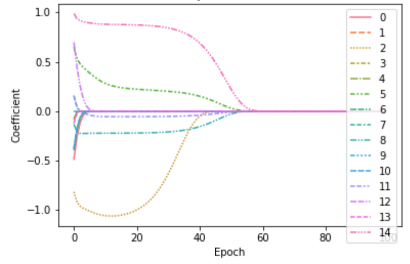
Nodes=3, Alpha=0.0001 Performance



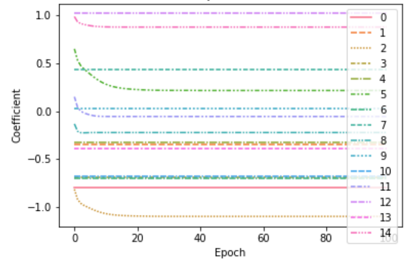
Nodes=3, Alpha=0.001 Performance



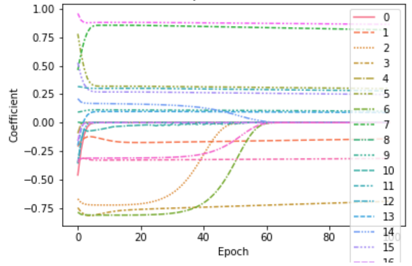
Nodes=3, Alpha=0.01 Performance



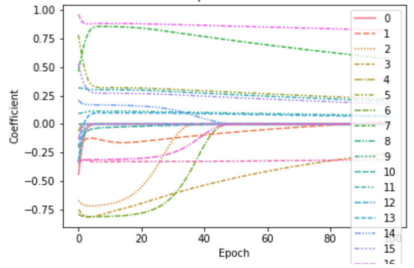
Nodes=3, Alpha=0 Performance



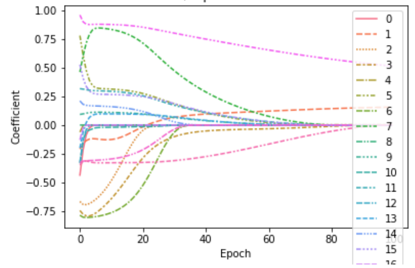
Nodes=4, Alpha=0.0001 Performance



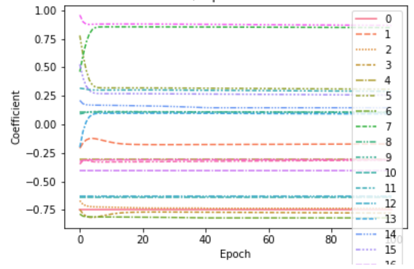
Nodes=4, Alpha=0.001 Performance



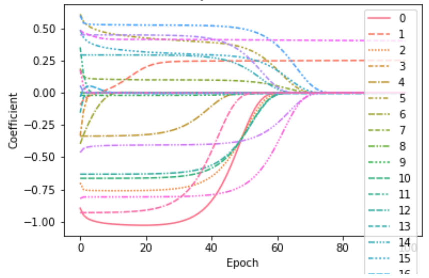
Nodes=4, Alpha=0.01 Performance



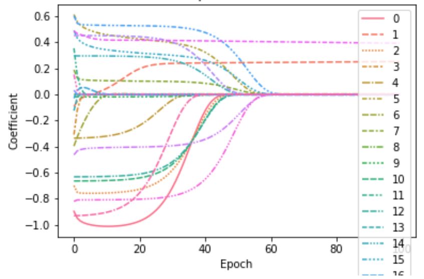
Nodes=4, Alpha=0 Performance



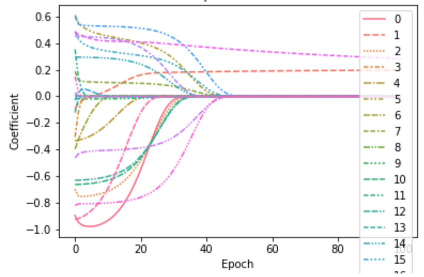
Nodes=5, Alpha=0.0001 Performance



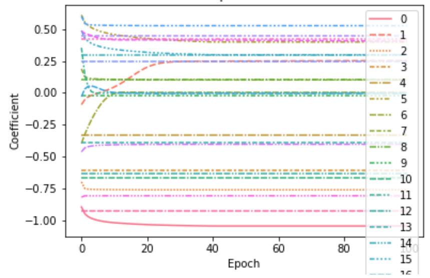
Nodes=5, Alpha=0.001 Performance



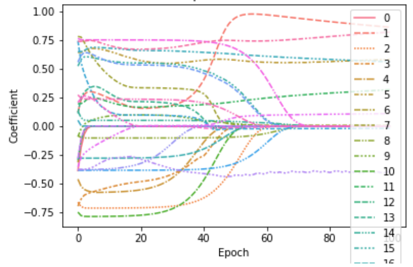
Nodes=5, Alpha=0.01 Performance



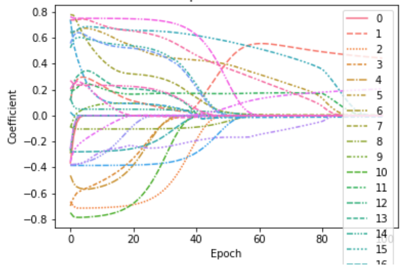
Nodes=5, Alpha=0 Performance



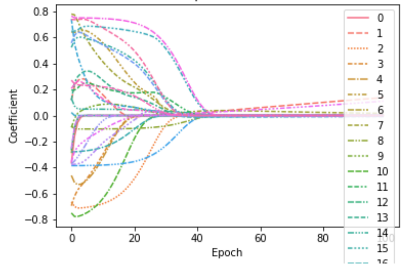
Nodes=6, Alpha=0.0001 Performance



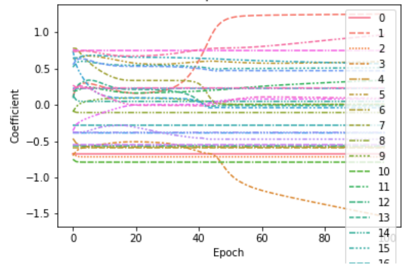
Nodes=6, Alpha=0.001 Performance



Nodes=6, Alpha=0.01 Performance



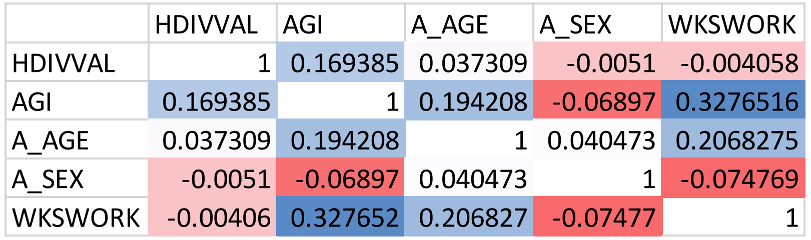
Nodes=6, Alpha=0 Performance



Part 1 Discussion

In the weight plots we see the weights converge quite rapidly for all L2 regularization sizes save alpha=0. The weights are failing to regularize (reach lower values) because we intentionally set the L2 regularization to zero. Our weights are not being regularized and intuitively should remain large in the case where L2 regularization is 0. This is an expected result. Our highest L2 regularization of 0.01 appears to have the most ‘gravity’ and pulls weight values down over epochs towards values near zero.

Regarding our learning rate plots the generalization gap appears small. The network architecture that reaches the loss on order of 10 to 0 fastest was the architecture with layer size of 4. We can inspect the correlation matrix on the right and note that our target HDIVVAL has insignificant to zero correlation with the features save for AGI (0.17). AGI only has a small correlation. We experimented with ylim to limit the range of values shown to greater highlight the generalization gap however the default library scaling was used in the end due to curve clipping.

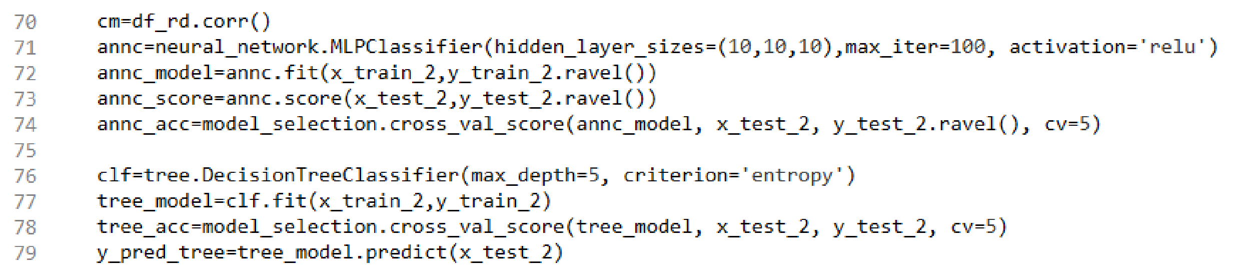


	HDIVVAL	AGI	A_AGE	A_SEX	WKSWORK
HDIVVAL	1	0.169385	0.037309	-0.0051	-0.004058
AGI	0.169385	1	0.194208	-0.06897	0.3276516
A_AGE	0.037309	0.194208	1	0.040473	0.2068275
A_SEX	-0.0051	-0.06897	0.040473	1	-0.074769
WKSWORK	-0.00406	0.327652	0.206827	-0.07477	1

Part 2 Code

We create a correlation matrix in order to gain a deeper understanding of the relationship between part 2 features and the part 2 target. This correlation matrix is displayed later. We create a neural network classifier object with 3 hidden layers. We tested various layer sizes and layers. We tested various layer sizes and activation functions, and we note they did not affect accuracy. We fit our neural network model and obtain performance metrics.

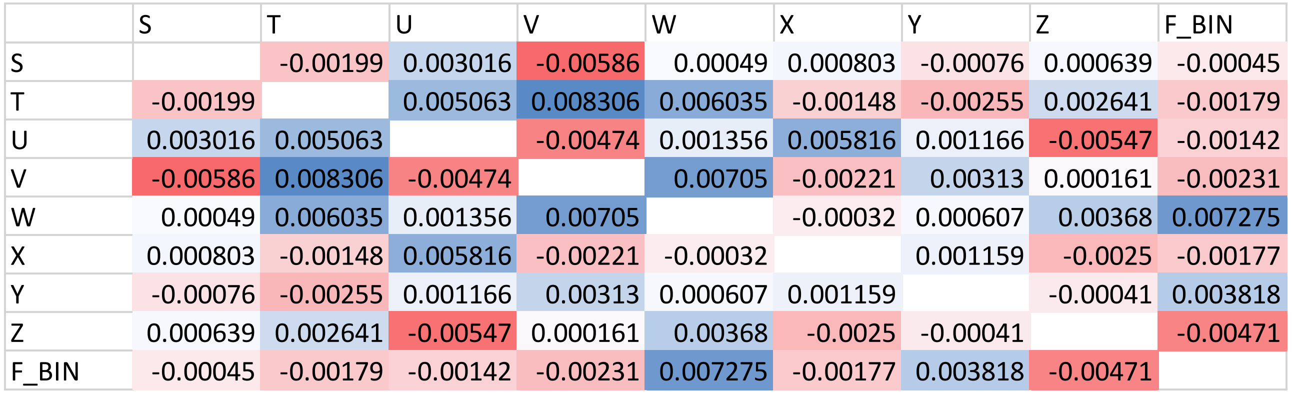
Next as a sanity check we created a decision tree model in order to gauge the performance of our neural network. We fit this model and obtain performance metrics for comparison purposes.



```
cm=df_rd.corr()
annc=neural_network.MLPClassifier(hidden_layer_sizes=(10,10,10),max_iter=100, activation='relu')
annc_model=annc.fit(x_train_2,y_train_2.ravel())
annc_score=annc.score(x_test_2,y_test_2.ravel())
annc_acc=model_selection.cross_val_score(annc_model, x_test_2, y_test_2.ravel(), cv=5)

clf=tree.DecisionTreeClassifier(max_depth=5, criterion='entropy')
tree_model=clf.fit(x_train_2,y_train_2)
tree_acc=model_selection.cross_val_score(tree_model, x_test_2, y_test_2, cv=5)
y_pred_tree=tree_model.predict(x_test_2)
```

Part 2 Discussion

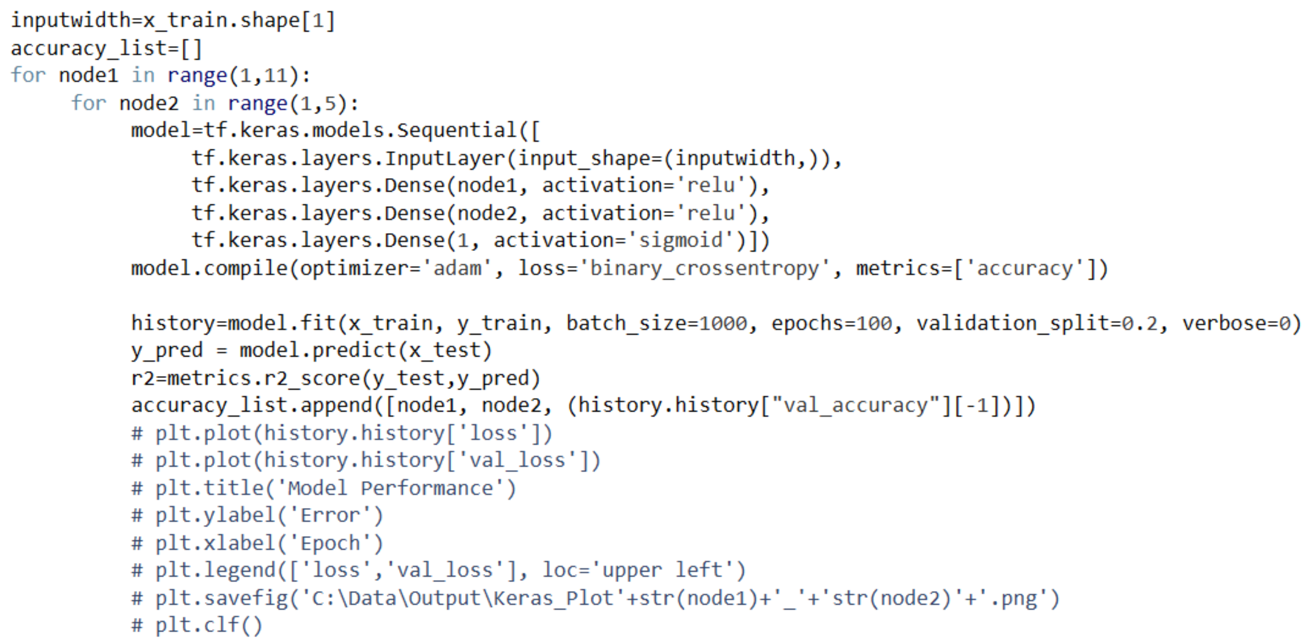
We created a set of neural network classifiers and measured their performance on the raw dataset. We experimented with three layers and node sizes from 3 to 10 as well as activation functions of rectifier and sigmoid. We achieved the same classification accuracy of 59.1% across neural network architectures. This maximum accuracy figure was reconfirmed with a decision tree model. This is due to the lack of correlation between the features and our target. We are aided in understanding our dataset with the below correlation matrix. The maximum absolute correlation with our target is 0.007 which is statistically insignificant. The reason the maximum accuracy is 59.1% is because 59.1% of targets in our dataset are false. A model achieves the maximum possible accuracy by always guessing false and does so because in this case it cannot find a statistically significant relationship between the features and target that improves on this kind of blind guessing. Inspection of the predictions made by our model on test data confirm it predicts only false values!



	S	T	U	V	W	X	Y	Z	F_BIN
S		-0.00199	0.003016	-0.00586	0.00049	0.000803	-0.00076	0.000639	-0.00045
T	-0.00199		0.005063	0.008306	0.006035	-0.00148	-0.00255	0.002641	-0.00179
U	0.003016	0.005063		-0.00474	0.001356	0.005816	0.001166	-0.00547	-0.00142
V	-0.00586	0.008306	-0.00474		0.00705	-0.00221	0.00313	0.000161	-0.00231
W	0.00049	0.006035	0.001356	0.00705		-0.00032	0.000607	0.00368	0.007275
X	0.000803	-0.00148	0.005816	-0.00221	-0.00032		0.001159	-0.0025	-0.00177
Y	-0.00076	-0.00255	0.001166	0.00313	0.000607	0.001159		-0.00041	0.003818
Z	0.000639	0.002641	-0.00547	0.000161	0.00368	-0.0025	-0.00041		-0.00471
F_BIN	-0.00045	-0.00179	-0.00142	-0.00231	0.007275	-0.00177	0.003818	-0.00471	

Part 3 Code

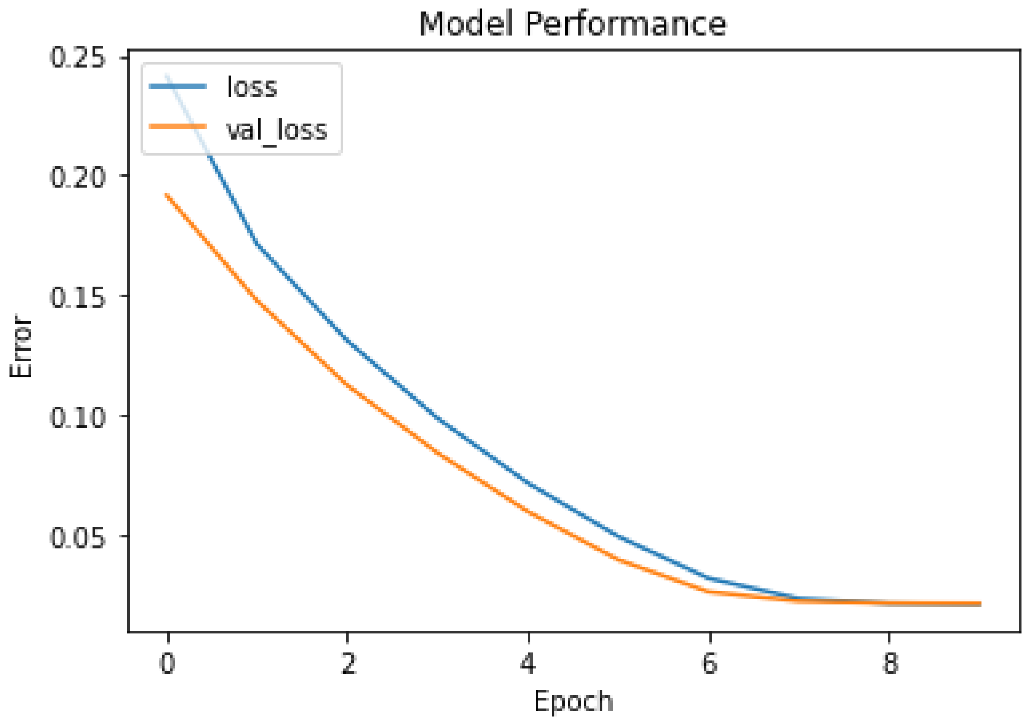
For part 3 we developed a set of tensor flow models. We iterate over two hidden layers with layer sizes of 1 to 4 and 1 to 10. We use the rectifier activation function on our hidden layers and sigmoid on our output layer. We compiled our model using the adam optimizer and binary cross entropy loss. We fit our model on the test and training sets of our census data and obtain performance metrics. We then proceed to plot these performance metrics.



```python
inputwidth=x_train.shape[1]
accuracy_list=[]
for node1 in range(1,11):
    for node2 in range(1,5):
        model=tf.keras.models.Sequential([
            tf.keras.layers.InputLayer(input_shape=(inputwidth,)),
            tf.keras.layers.Dense(node1, activation='relu'),
            tf.keras.layers.Dense(node2, activation='relu'),
            tf.keras.layers.Dense(1, activation='sigmoid')])
        model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

        history=model.fit(x_train, y_train, batch_size=1000, epochs=100, validation_split=0.2, verbose=0)
        y_pred = model.predict(x_test)
        r2=metrics.r2_score(y_test,y_pred)
        accuracy_list.append([node1, node2, (history.history["val_accuracy"][-1])])
        # plt.plot(history.history['loss'])
        # plt.plot(history.history['val_loss'])
        # plt.title('Model Performance')
        # plt.ylabel('Error')
        # plt.xlabel('Epoch')
        # plt.legend(['loss','val_loss'], loc='upper left')
        # plt.savefig('C:\Data\Output\Keras_Plot'+str(node1)+'_'+'str(node2)'+'.png')
        # plt.clf()
```

Part 3 Discussion

Using TensorFlow we developed a regression MLP to predict HDDIVVAL from the same features used in part 1. We iterate through two hidden layers. Layer 1 iterated from node sizes of 1 to 10. Layer 2 iterated from node sizes of 1 to 4. We see zero change in final accuracies regardless of network architecture tested. Final validation accuracy obtained was 67.71%. Though inspection of the history dictionary revealed that accuracy would rapidly converge on this final value in early epochs. TensorFlow, based on this limited experiment, was an efficient library at finding the maximum accuracy on this dataset. This dataset is relatively low information gain per feature based on the correlation matrix presented in part 1 discussion. AGI had the best correlation at 0.17 with the target giving our model only limited predictive ability. See our table of validation accuracy on the left, every model had a final validation accuracy of 67.71%. Inspection of our loss versus epoch plot shows that our model's generalization gap is low and disappears entirely by epoch 10. We trained on 100 epochs and zoomed in here to show the generalization gap.



Validation Accuracy	Layer 1 Nodes	Layer 2 Nodes
0.6771	1	1
0.6771	2	1
0.6771	3	1
0.6771	4	1
0.6771	5	1
0.6771	6	1
0.6771	7	1
0.6771	8	1
0.6771	9	1
0.6771	10	1
0.6771	1	2
0.6771	2	2
0.6771	3	2
0.6771	4	2
0.6771	5	2
0.6771	6	2
0.6771	7	2
0.6771	8	2
0.6771	9	2
0.6771	10	2
0.6771	1	3
0.6771	2	3
0.6771	3	3
0.6771	4	3
0.6771	5	3
0.6771	6	3
0.6771	7	3
0.6771	8	3
0.6771	9	3
0.6771	10	3
0.6771	1	4
0.6771	2	4
0.6771	3	4
0.6771	4	4
0.6771	5	4
0.6771	6	4
0.6771	7	4
0.6771	8	4
0.6771	9	4
0.6771	10	4