Michael Ferko

Dr. Jacek Zurada

ECE 614 Deep Learning

24 February 2020

Lab 2: Classification

Classification of MNIST 70,000 Handwritten Digits 0-9 Image Data Set

Objective:

To understand, implement and optimize a classifier of MNIST (planar 2D patterns) with multilayer perceptrons using SoftMax Neurons.

Tasks

1. Run the code as given for the initial model 784x16x10 (see Fig.1). Do this cell by cell by clicking the Run button. In the visualization code segment, monitor the plot of the validation loss and of the validation accuracy for classification. Note the final value of validation accuracy in the last row.

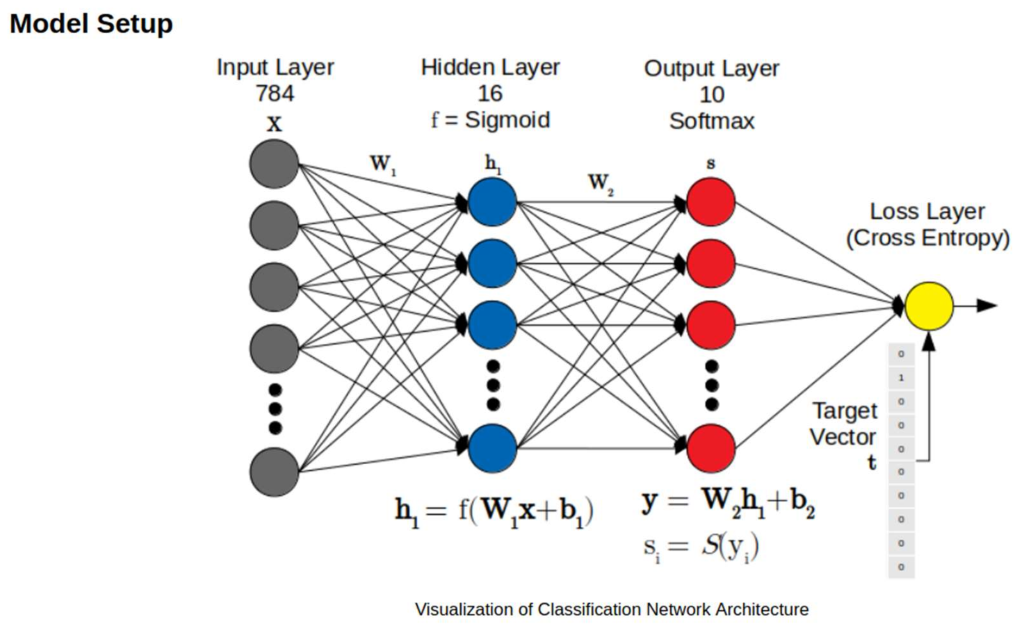
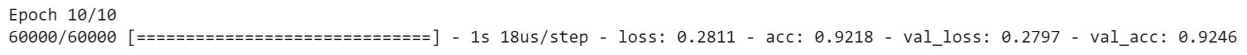

Model Setup
Input Layer 784 x
Hidden Layer 16 f = Sigmoid
Output Layer 10 Softmax
W1
h1
W2
s
Loss Layer (Cross Entropy)
Target Vector t
0 1 0 0 0 0 0 0 0 0
$\mathbf{h}_1 = f(\mathbf{W}_1\mathbf{x}+\mathbf{b}_1)$
$\mathbf{y} = \mathbf{W}_2\mathbf{h}_1+\mathbf{b}_2$
$s_i = S(y_i)$
Visualization of Classification Network Architecture

Figure 1: Visualization of Classification Network Architecture



```
Epoch 10/10
60000/60000 [==============================] - 1s 18us/step - loss: 0.2811 - acc: 0.9218 - val_loss: 0.2797 - val_acc: 0.9246
```

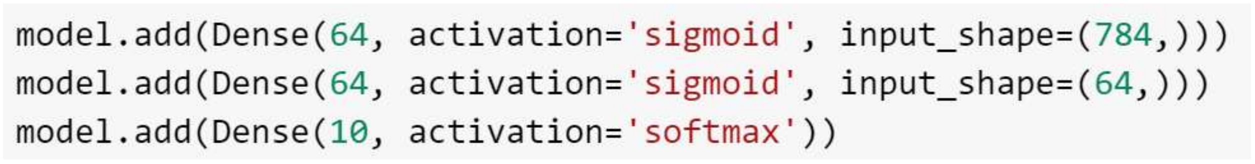
Figure 2: Noted value of Validation accuracy in last row val_acc = 0.9246

2. Now insert final value of validation accuracy as the first row in the report table that has 3 rows (one for each activation f-n) and 3 columns (one for each architecture) and the heading row/column. The table should list the best validation accuracy of classification achieved with a tested model. NOTE: "validation" terms refer actually to the "testing" because of how the python compiles the code.

Activation Function	784x16x10 Architecture	784x64x64x10 Architecture	784x64x10 Architecture
sigmoid	0.9246	0.9278	0.9335
relu	0.9455	0.9788	0.9654
tanh	0.9408	0.9696	0.9599

Table 1: Final Values of Validation Accuracy for 3 Activation Functions and 3 Architectures

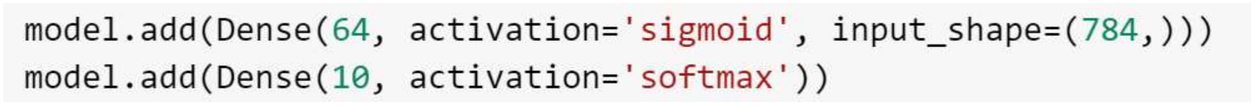
3. Modify the initial architecture as in the original code by changing the sizes and number of the hidden layers 784xH1x(H2-if-needed)x10 (keep the output layer always intact with 10 softmax neurons). As an example, try two hidden layers with 64 neurons per layer: 784x64x64x10. Modify the model again, by picking another architecture (smaller?/larger?). In sum, you need to try 3 architectures.



```
model.add(Dense(64, activation='sigmoid', input_shape=(784,)))
model.add(Dense(64, activation='sigmoid', input_shape=(64,)))
model.add(Dense(10, activation='softmax'))
```

Figure 3: Changes made for two 64 Neurons hidden layers Architecture

After trying several different architctures on sigmoidal activation function and comparing the validation accuracy I found that 2 layers was giving a lower validation accuracy. I tried 32, 64 and 128 to see varying powers of 2 neurons in the hidden neural layers and found 64 neurons in one hidden layer gave the highest validation accuracy.



```
model.add(Dense(64, activation='sigmoid', input_shape=(784,)))
model.add(Dense(10, activation='softmax'))
```

Figure 4: Changes made for one 64 Neurons hidden layers Architecture

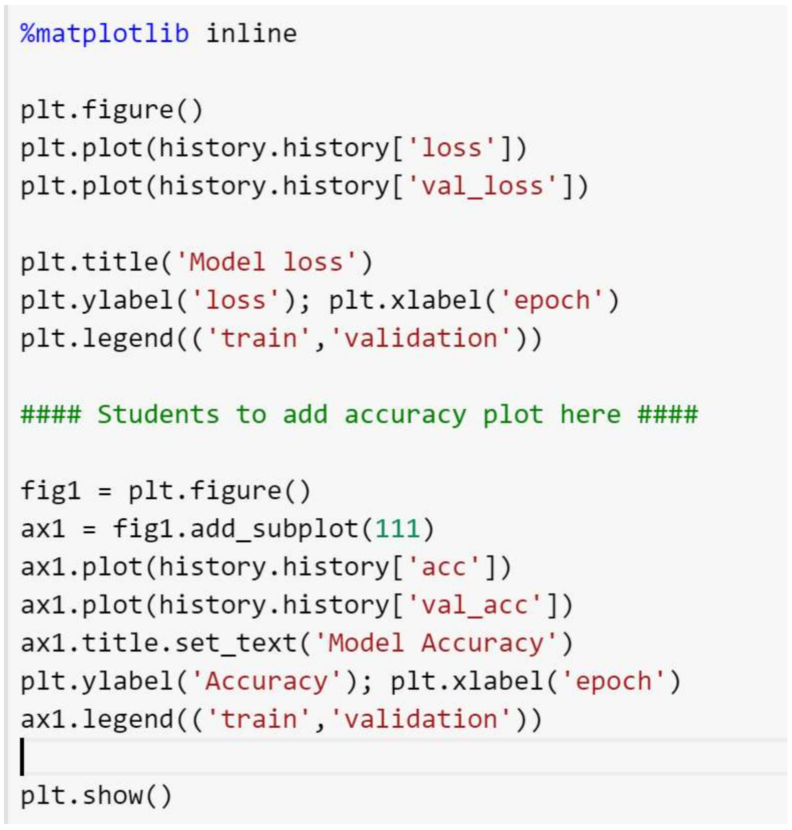
For each selected architecture, you'll need to adjust the weight visualization part to produce correct figure structures and their sizes. For each of the three architectures try three arbitrary selected activation functions, giving the total of 9 cases (or 9 final networks).

4. Insert the final validation accuracies for each network into the report table. Select the best two of 9 cases tested, note the models' details and highlight their best final validation accuracy of classification. Note that Task 7 will require comparison of these two most accurate models of Task 2 with different networks specified in Task 7, so preferably keep the two top models saved.

Top 2 final validation Accuracies are highlighted in Table 1.

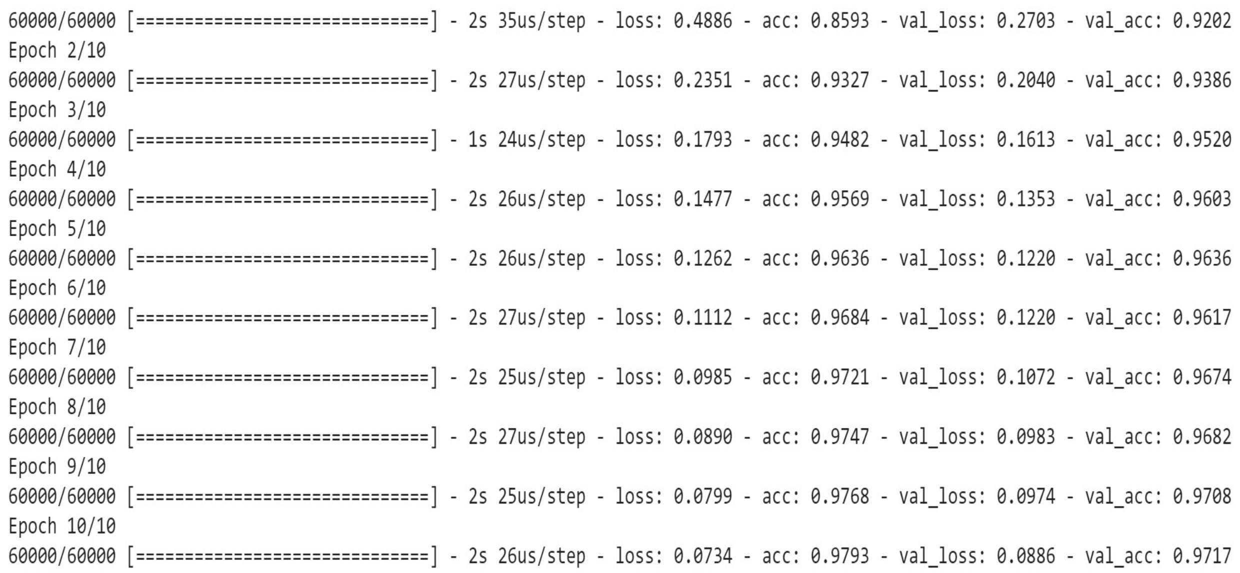
5. For the very top model provide the full documentation of complete run, including the model loss for training curves and weight visualizations in each layer.

Top Model from Table 1 is the relu activation function with the 784x64x64x10 architecture. The code changes made for showing model accuracy for this architecture are shown in Figure 5:



```
%matplotlib inline

plt.figure()
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])

plt.title('Model loss')
plt.ylabel('loss'); plt.xlabel('epoch')
plt.legend(('train','validation'))

#### Students to add accuracy plot here ####

fig1 = plt.figure()
ax1 = fig1.add_subplot(111)
ax1.plot(history.history['acc'])
ax1.plot(history.history['val_acc'])
ax1.title.set_text('Model Accuracy')
plt.ylabel('Accuracy'); plt.xlabel('epoch')
ax1.legend(('train','validation'))

plt.show()
```

Figure 5: Coding Changes for Model Accuracy



```
60000/60000 [==============================] - 2s 35us/step - loss: 0.4886 - acc: 0.8593 - val_loss: 0.2703 - val_acc: 0.9202
Epoch 2/10
60000/60000 [==============================] - 2s 27us/step - loss: 0.2351 - acc: 0.9327 - val_loss: 0.2040 - val_acc: 0.9386
Epoch 3/10
60000/60000 [==============================] - 1s 24us/step - loss: 0.1793 - acc: 0.9482 - val_loss: 0.1613 - val_acc: 0.9520
Epoch 4/10
60000/60000 [==============================] - 2s 26us/step - loss: 0.1477 - acc: 0.9569 - val_loss: 0.1353 - val_acc: 0.9603
Epoch 5/10
60000/60000 [==============================] - 2s 26us/step - loss: 0.1262 - acc: 0.9636 - val_loss: 0.1220 - val_acc: 0.9636
Epoch 6/10
60000/60000 [==============================] - 2s 27us/step - loss: 0.1112 - acc: 0.9684 - val_loss: 0.1220 - val_acc: 0.9617
Epoch 7/10
60000/60000 [==============================] - 2s 25us/step - loss: 0.0985 - acc: 0.9721 - val_loss: 0.1072 - val_acc: 0.9674
Epoch 8/10
60000/60000 [==============================] - 2s 27us/step - loss: 0.0890 - acc: 0.9747 - val_loss: 0.0983 - val_acc: 0.9682
Epoch 9/10
60000/60000 [==============================] - 2s 25us/step - loss: 0.0799 - acc: 0.9768 - val_loss: 0.0974 - val_acc: 0.9708
Epoch 10/10
60000/60000 [==============================] - 2s 26us/step - loss: 0.0734 - acc: 0.9793 - val_loss: 0.0886 - val_acc: 0.9717
```

Figure 6: Model training History

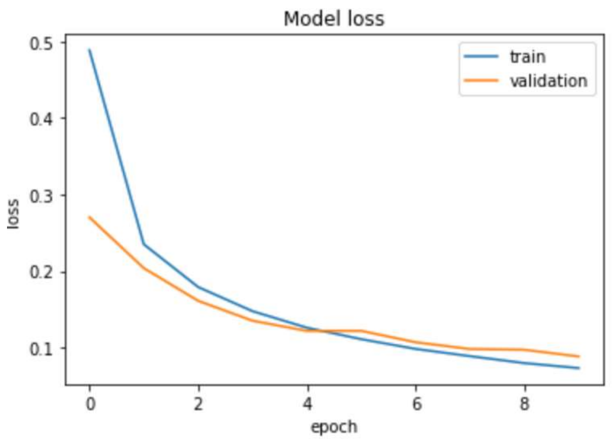


Figure 7(a): Model Loss Training Curve

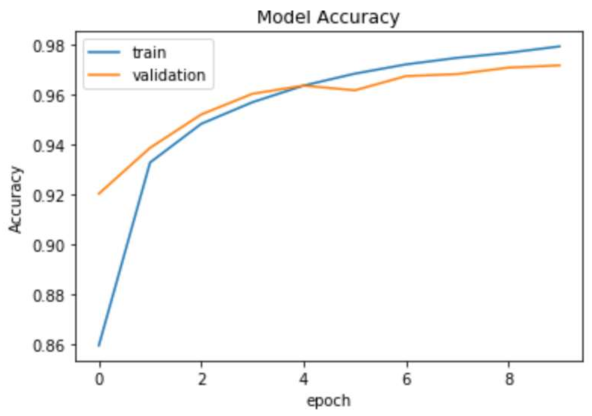
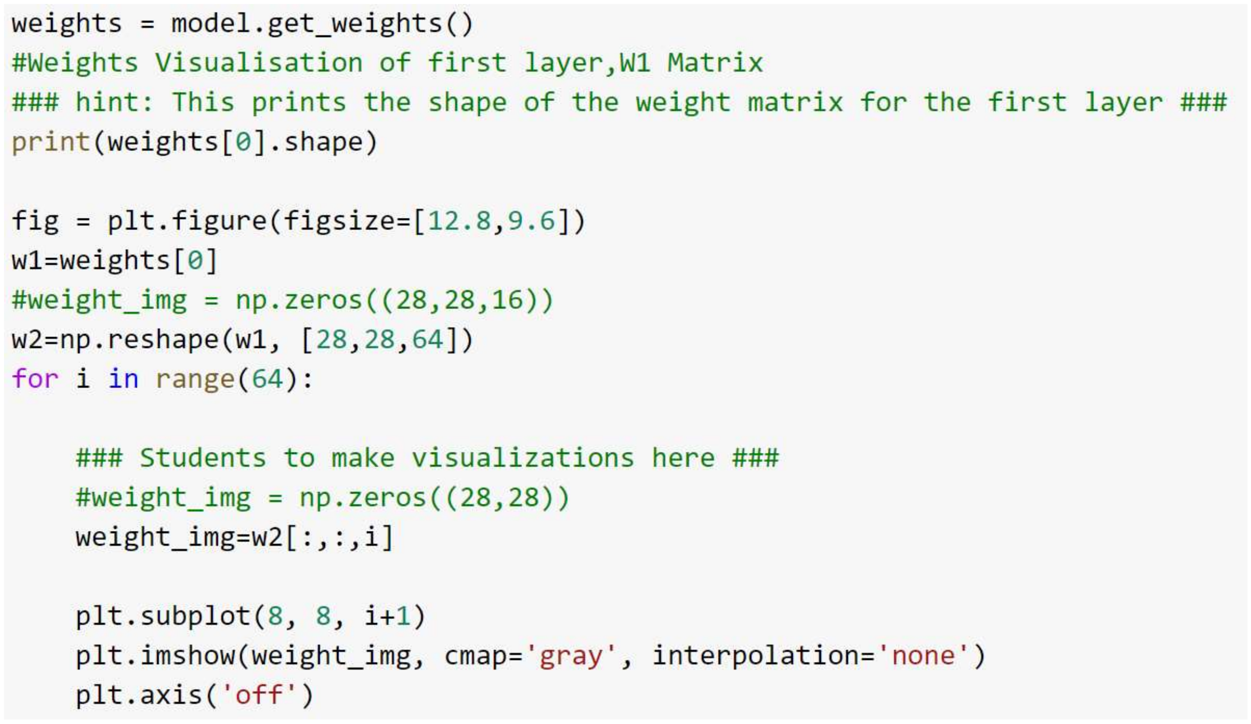
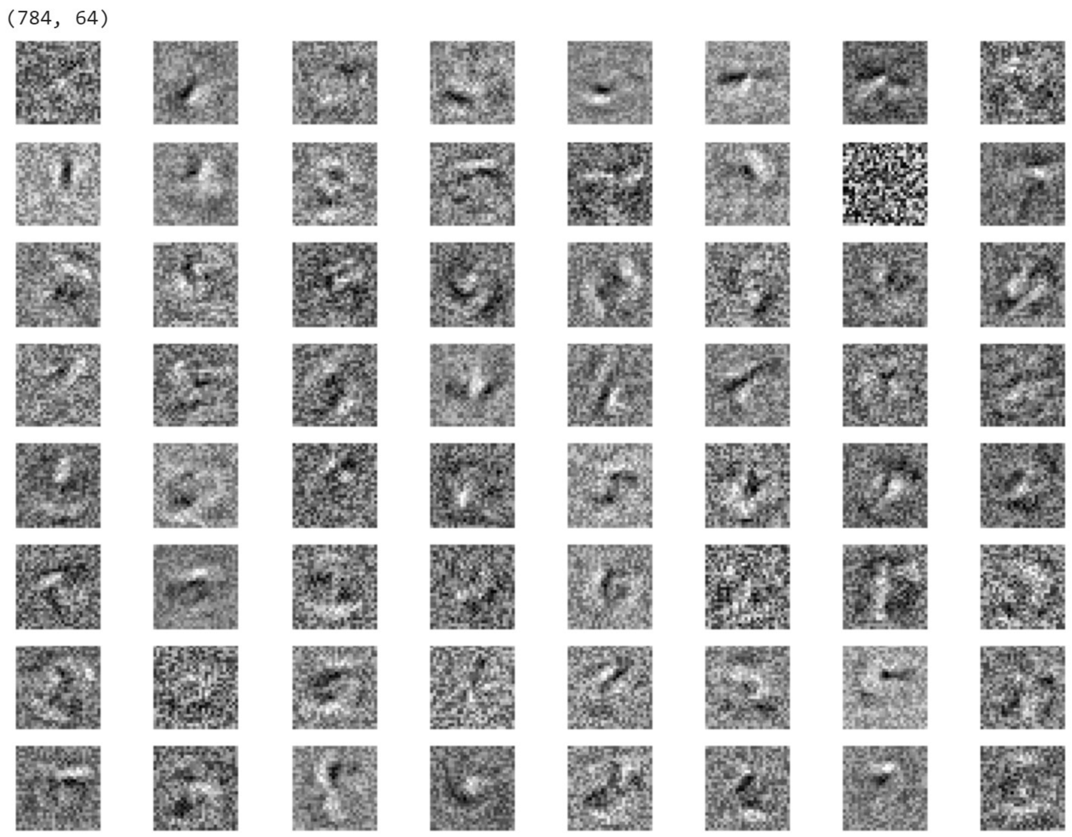


Figure 7(b): Model Accuracy



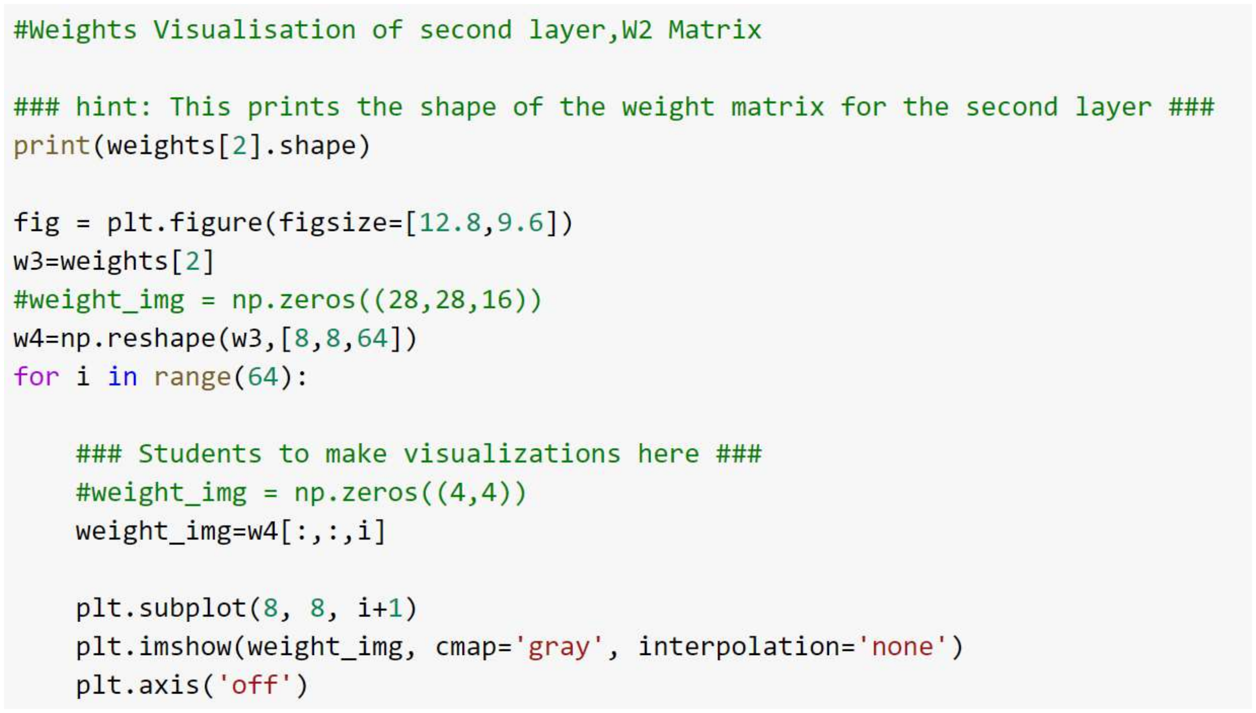
```
weights = model.get_weights()
#Weights Visualisation of first layer,W1 Matrix
### hint: This prints the shape of the weight matrix for the first layer ###
print(weights[0].shape)

fig = plt.figure(figsize=[12.8,9.6])
w1=weights[0]
#weight_img = np.zeros((28,28,16))
w2=np.reshape(w1, [28,28,64])
for i in range(64):

    ### Students to make visualizations here ###
    #weight_img = np.zeros((28,28))
    weight_img=w2[:,:,i]

    plt.subplot(8, 8, i+1)
    plt.imshow(weight_img, cmap='gray', interpolation='none')
    plt.axis('off')
```

Figure 8: Code changes made for Visualization of First Layer Weights



```
(784, 64)
```

Figure 9: Visualization of First Layer Weights



```
#Weights Visualisation of second layer,W2 Matrix

### hint: This prints the shape of the weight matrix for the second layer ###
print(weights[2].shape)

fig = plt.figure(figsize=[12.8,9.6])
w3=weights[2]
#weight_img = np.zeros((28,28,16))
w4=np.reshape(w3,[8,8,64])
for i in range(64):

    ### Students to make visualizations here ###
    #weight_img = np.zeros((4,4))
    weight_img=w4[:,:,i]

    plt.subplot(8, 8, i+1)
    plt.imshow(weight_img, cmap='gray', interpolation='none')
    plt.axis('off')
```

Figure 10: Code changes made for Visualization of Second Layer Weights

```
(64, 64)
```

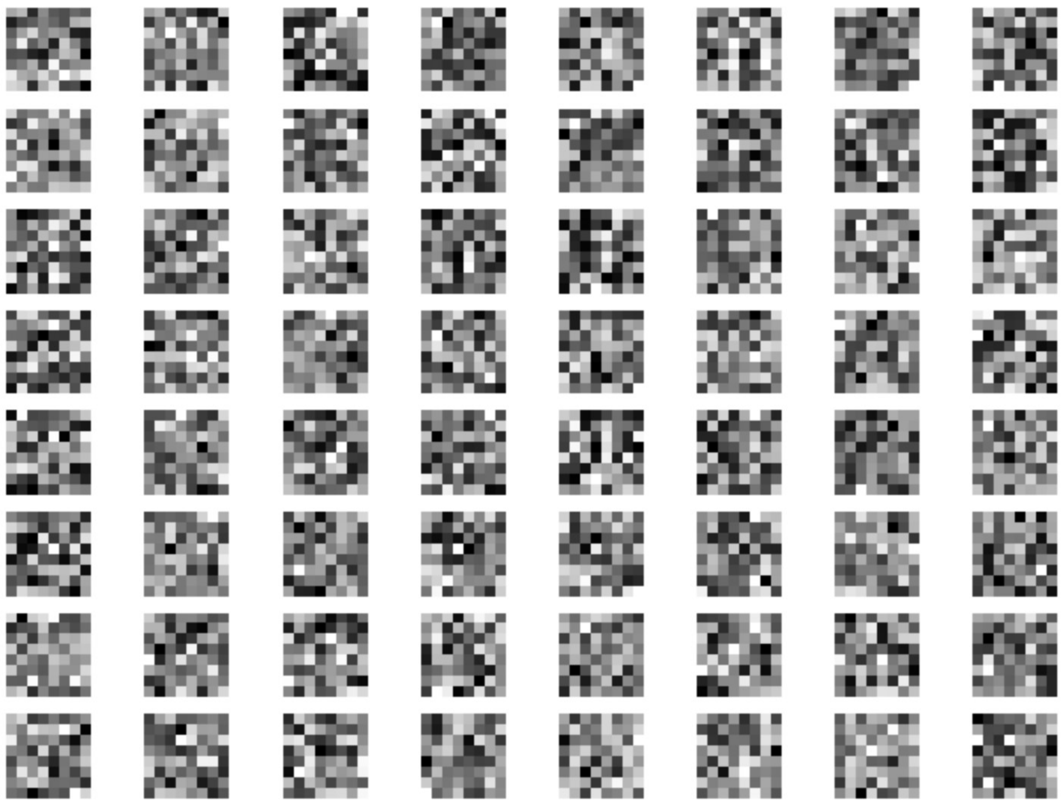
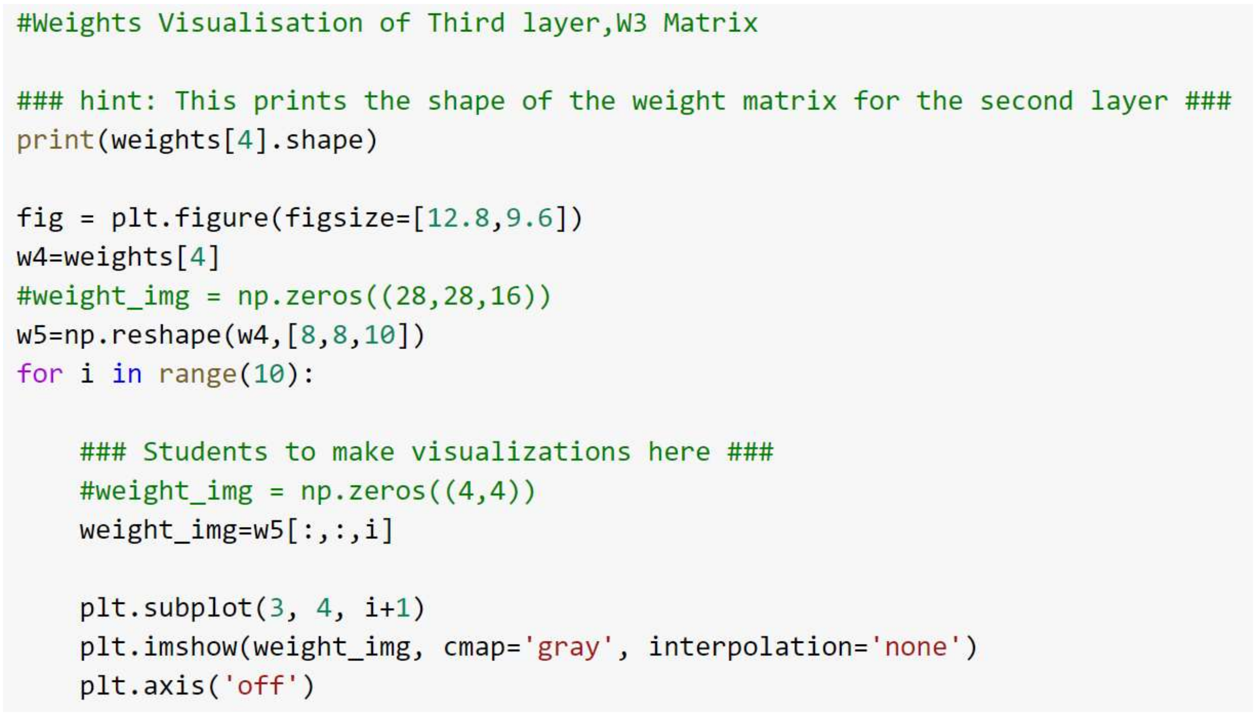


Figure 11: Visualization of Second Layer Weights



```
#Weights Visualisation of Third layer,W3 Matrix

### hint: This prints the shape of the weight matrix for the second layer ###
print(weights[4].shape)

fig = plt.figure(figsize=[12.8,9.6])
w4=weights[4]
#weight_img = np.zeros((28,28,16))
w5=np.reshape(w4,[8,8,10])
for i in range(10):

    ### Students to make visualizations here ###
    #weight_img = np.zeros((4,4))
    weight_img=w5[:,:,i]

    plt.subplot(3, 4, i+1)
    plt.imshow(weight_img, cmap='gray', interpolation='none')
    plt.axis('off')
```

Figure 12: Code changes made for Visualization of Third Layer Weights

```
(64, 10)
```

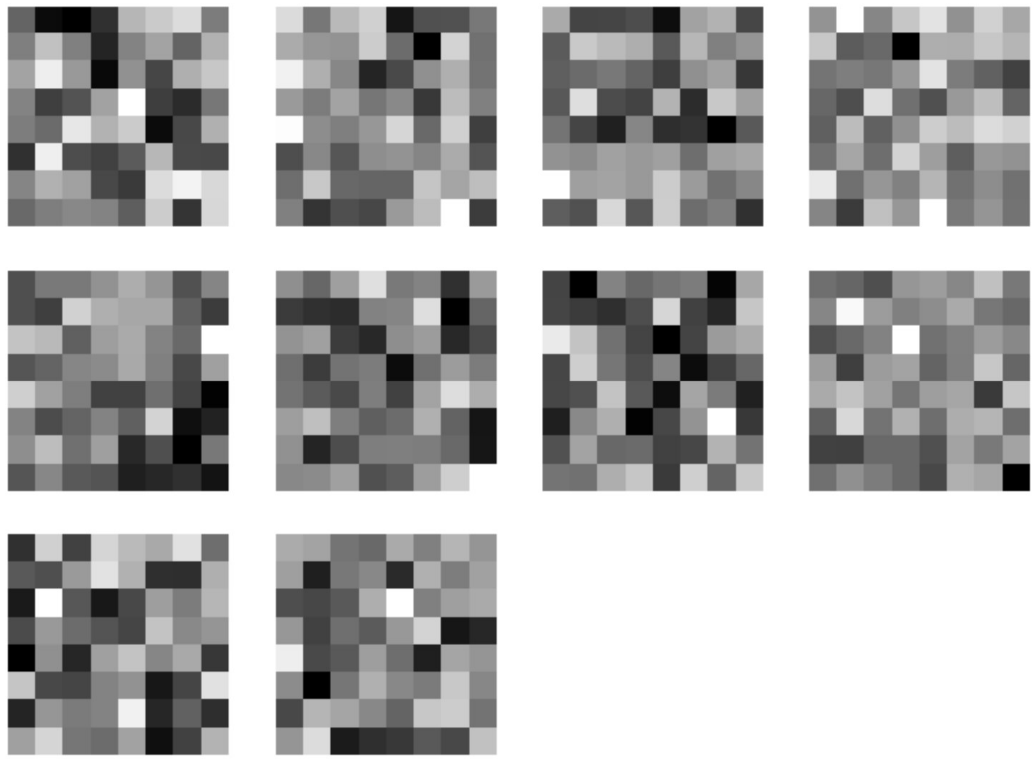


Figure 13: Visualization of Second Layer Weights

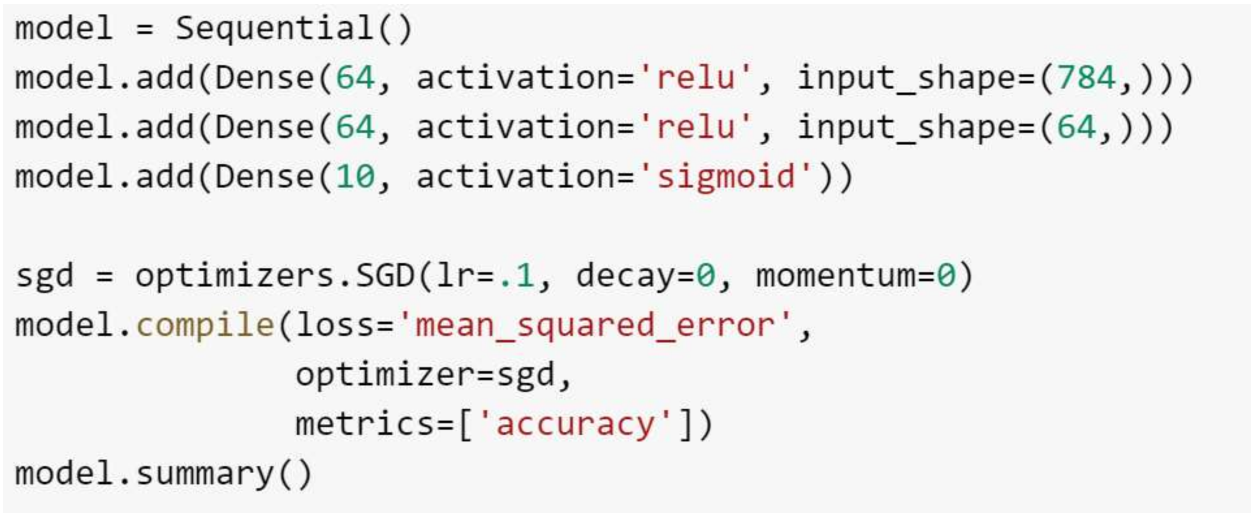
6. By inspecting the report table, comment on any interesting results in the validation loss and accuracy vs. model architectures (layer number and sizes, activation functions, etc).

As I stated before. I fiddled with many different type of architectures for the sigmoidal activation function, but now that I have proceeded to this step in Lab 2 I have noticed a better way to optimize the architecture. Since the activation functions have the best model accuracy in the order relu > tanh > sigmoid I should have tried fiddling with the relu activation function. I didn't notice any difference with using the relu activation function in one layer and the sigmoid in another layer, but now that I have there was a definite increase in model accuracy for sigmoid layer with a relu layer, but still lower than the two relu layers.

After fiddling with the two relu layer concept I found that lowering the layer size to 32 decreased the accuracy, but also less of a decrease in accuracy when one layer was 64. I then observed a decrease in accuracy slightly lower than the 32 when increasing the layer size to 128, but a decrease in accuracy when one layer was 128 and the other was 64 neurons. I then tried the same procedure for 3 layers and only decreased model accuracy no matter what.

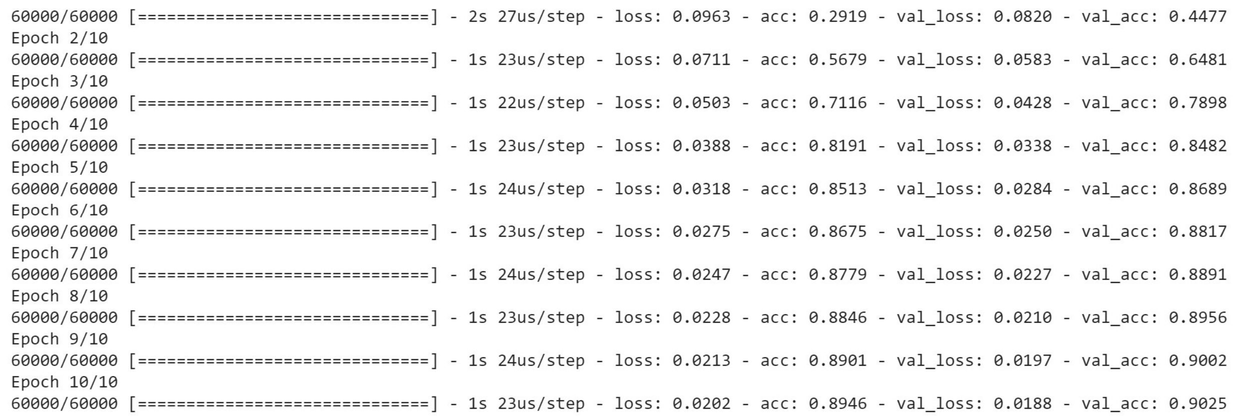
It may be concluded that from all of these procedures for finding an optimal layer size, activation function and number of layers. The best performance observed has been by the 784x64x64x10 2 relu hidden layers. Also, I noticed that the model loss and model accuracy validations and train lines would intersect at incremented epochs inversely proportional to the changes observed through the previously explained procedure. i.e. the point of intersection of the two lines would be at a later epoch when final model validation accuracy would decrease, and the intersection of the lines would be at a sooner epoch when the model accuracy would increase.

7. Change the classifier's loss function from categorical cross entropy CE (see Fig.2) to Mean Squared Error (MSE or mse). This will likely require change from softmax output neurons back to the classic tanh or sigmoidal type neurons. Compare your results with cross entropy validation accuracy for the two best models selected out of 9 from Task 2. Choose the better performing models for any future tasks.



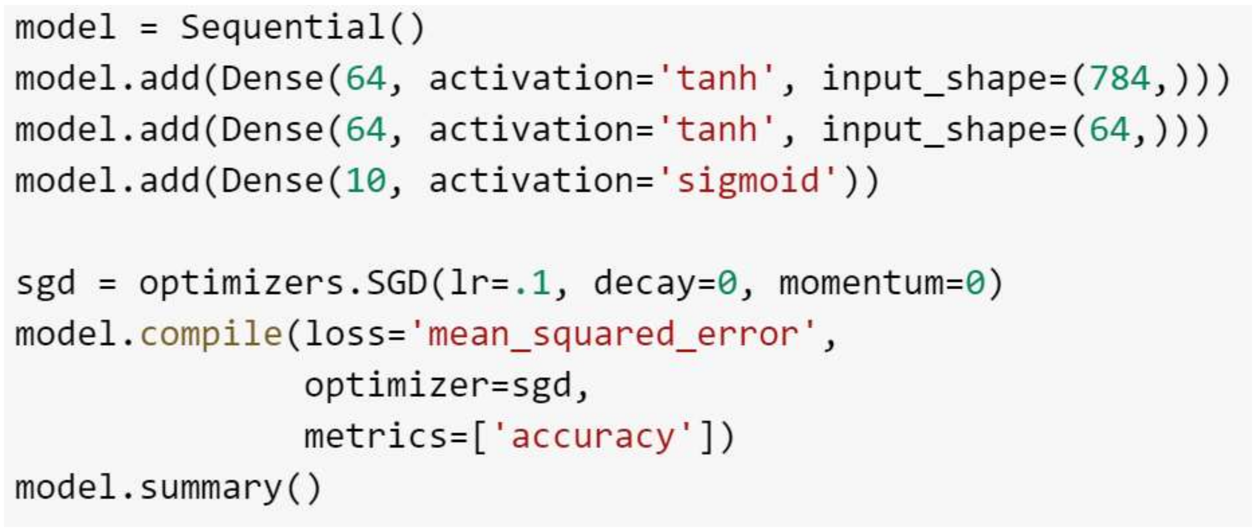
```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(784,)))
model.add(Dense(64, activation='relu', input_shape=(64,)))
model.add(Dense(10, activation='sigmoid'))

sgd = optimizers.SGD(lr=.1, decay=0, momentum=0)
model.compile(loss='mean_squared_error',
              optimizer=sgd,
              metrics=['accuracy'])
model.summary()
```

Figure 14: Code Changes Made Best Architecture from Table 1



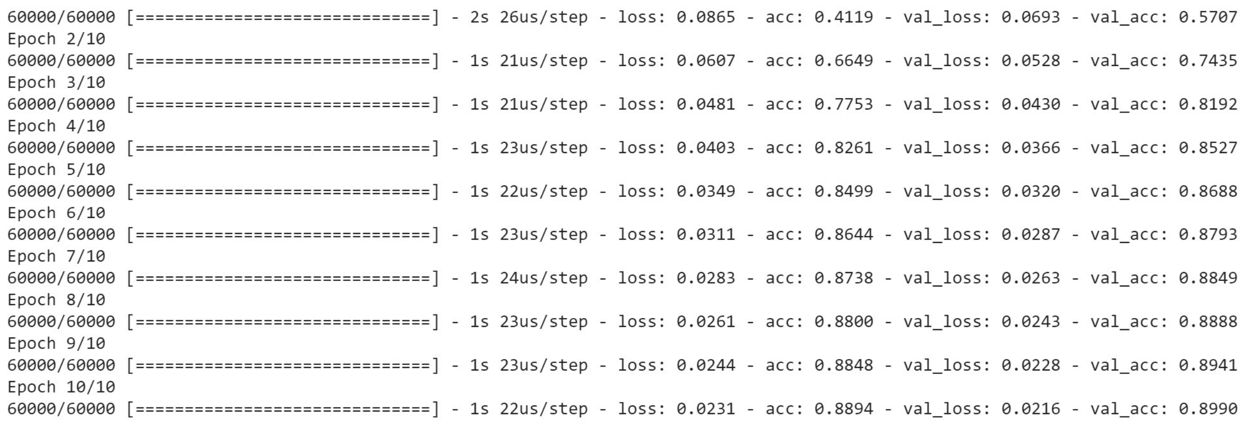
```
60000/60000 [==============================] - 2s 27us/step - loss: 0.0963 - acc: 0.2919 - val_loss: 0.0820 - val_acc: 0.4477
Epoch 2/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.0711 - acc: 0.5679 - val_loss: 0.0583 - val_acc: 0.6481
Epoch 3/10
60000/60000 [==============================] - 1s 22us/step - loss: 0.0503 - acc: 0.7116 - val_loss: 0.0428 - val_acc: 0.7898
Epoch 4/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.0388 - acc: 0.8191 - val_loss: 0.0338 - val_acc: 0.8482
Epoch 5/10
60000/60000 [==============================] - 1s 24us/step - loss: 0.0318 - acc: 0.8513 - val_loss: 0.0284 - val_acc: 0.8689
Epoch 6/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.0275 - acc: 0.8675 - val_loss: 0.0250 - val_acc: 0.8817
Epoch 7/10
60000/60000 [==============================] - 1s 24us/step - loss: 0.0247 - acc: 0.8779 - val_loss: 0.0227 - val_acc: 0.8891
Epoch 8/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.0228 - acc: 0.8846 - val_loss: 0.0210 - val_acc: 0.8956
Epoch 9/10
60000/60000 [==============================] - 1s 24us/step - loss: 0.0213 - acc: 0.8901 - val_loss: 0.0197 - val_acc: 0.9002
Epoch 10/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.0202 - acc: 0.8946 - val_loss: 0.0188 - val_acc: 0.9025
```

Figure 15: Model History Showing Final Validation Accuracy of 0.9025



```
model = Sequential()
model.add(Dense(64, activation='tanh', input_shape=(784,)))
model.add(Dense(64, activation='tanh', input_shape=(64,)))
model.add(Dense(10, activation='sigmoid'))

sgd = optimizers.SGD(lr=.1, decay=0, momentum=0)
model.compile(loss='mean_squared_error',
              optimizer=sgd,
              metrics=['accuracy'])
model.summary()
```

Figure 16: Code Changes Made 2nd Best Architecture from Table 1



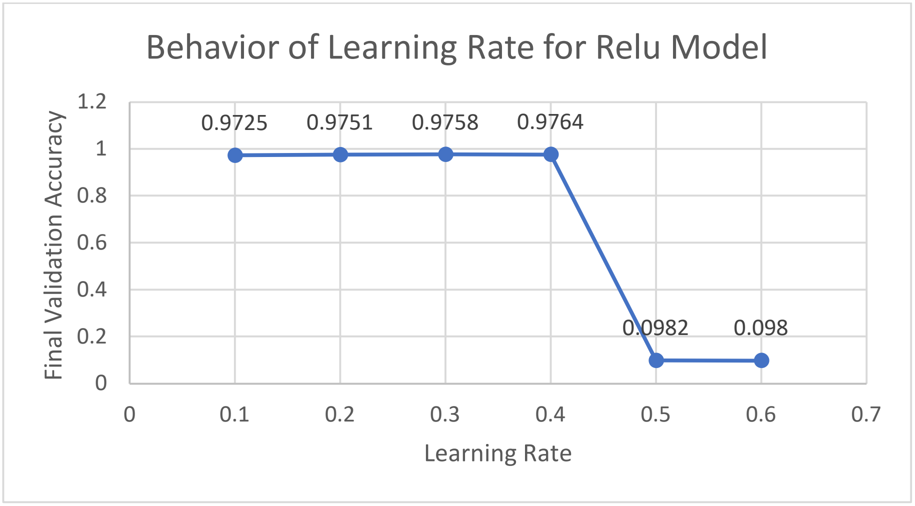
```
60000/60000 [==============================] - 2s 26us/step - loss: 0.0865 - acc: 0.4119 - val_loss: 0.0693 - val_acc: 0.5707
Epoch 2/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.0607 - acc: 0.6649 - val_loss: 0.0528 - val_acc: 0.7435
Epoch 3/10
60000/60000 [==============================] - 1s 21us/step - loss: 0.0481 - acc: 0.7753 - val_loss: 0.0430 - val_acc: 0.8192
Epoch 4/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.0403 - acc: 0.8261 - val_loss: 0.0366 - val_acc: 0.8527
Epoch 5/10
60000/60000 [==============================] - 1s 22us/step - loss: 0.0349 - acc: 0.8499 - val_loss: 0.0320 - val_acc: 0.8688
Epoch 6/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.0311 - acc: 0.8644 - val_loss: 0.0287 - val_acc: 0.8793
Epoch 7/10
60000/60000 [==============================] - 1s 24us/step - loss: 0.0283 - acc: 0.8738 - val_loss: 0.0263 - val_acc: 0.8849
Epoch 8/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.0261 - acc: 0.8800 - val_loss: 0.0243 - val_acc: 0.8888
Epoch 9/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.0244 - acc: 0.8848 - val_loss: 0.0228 - val_acc: 0.8941
Epoch 10/10
60000/60000 [==============================] - 1s 22us/step - loss: 0.0231 - acc: 0.8894 - val_loss: 0.0216 - val_acc: 0.8990
```

Figure 17: Model History Showing Final Validation Accuracy of 0.8990

Cross entropy validation accuracy is much higher than the results for mean squared error as shown from Figure 15 and 17. Therefore, I will continue to use the 784x64x64x10 architecture with the 2 relu layers or 2 tanh layers as the better performing models from Table 1.

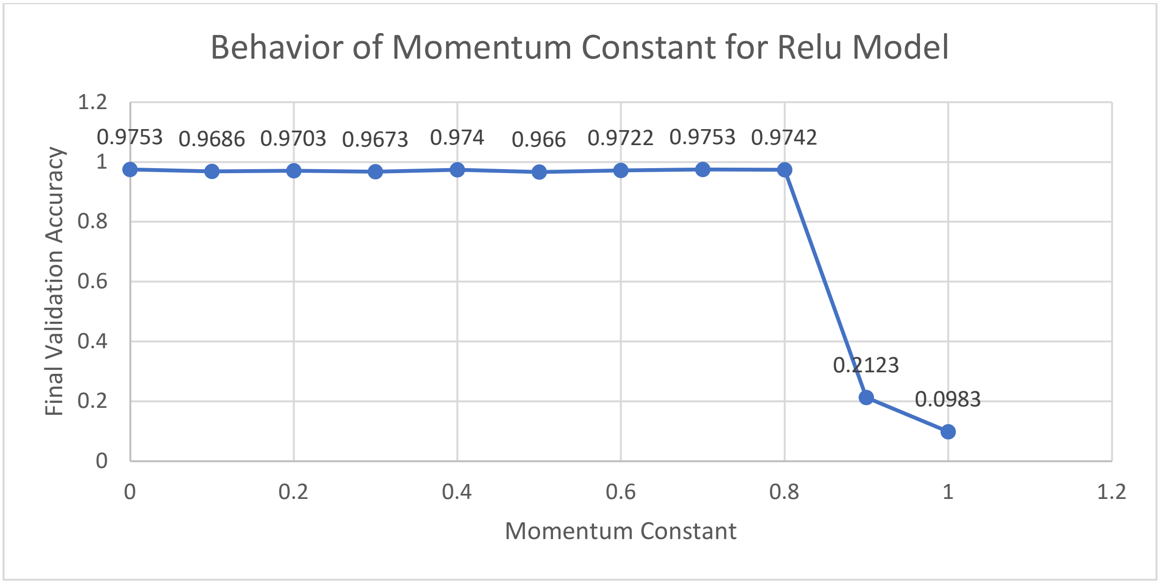
8. Try to get experiment outside the box for the best accuracy by tuning your best model through adapting the architecture, choice of activation f-n, the lr and momentum constant. Stay within resasonable reason (say 30 minutes of time) as the number of combinations grows exponentially. Answer by listing the optimum hyperparameters and accuracies.

First, I will try adjusting the learning rate up, and monitor its behavior related to the model accuracy.



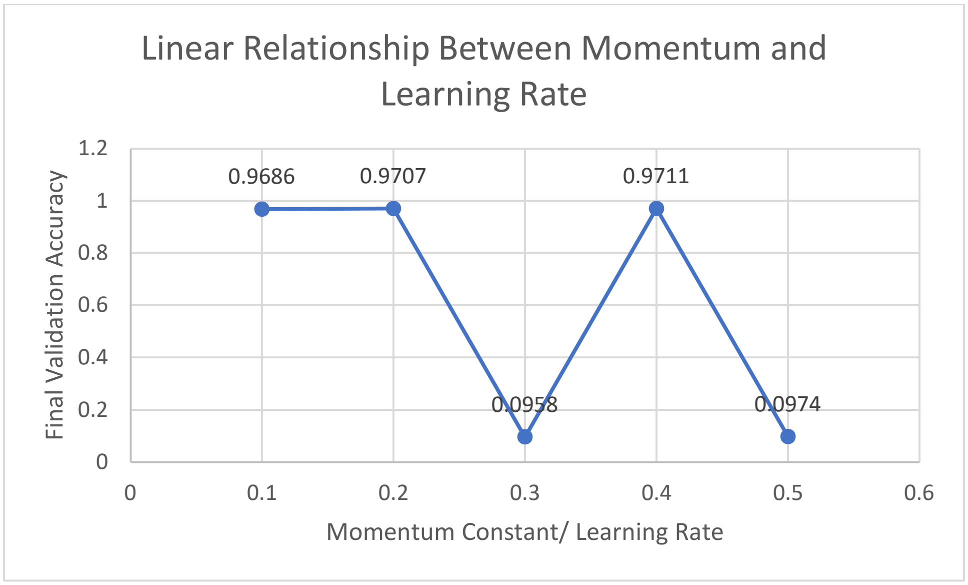
Graph 1: Behavior of Learning Rate for Relu Model

Now, I will change the learning rate back to lr = 0.1 and monitor the changes in the momentum constant.



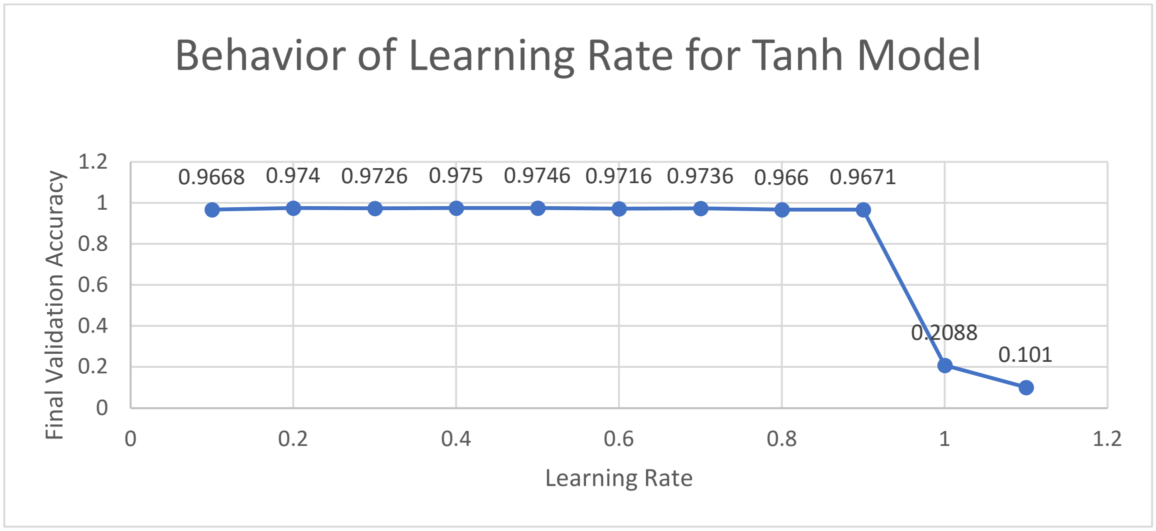
Graph 2: Behavior of Momentum Constant for Relu Model

Now that I know the individual behavior, I will see how linearly increasing the momentum and learning rate will effect the final validation accuracy.

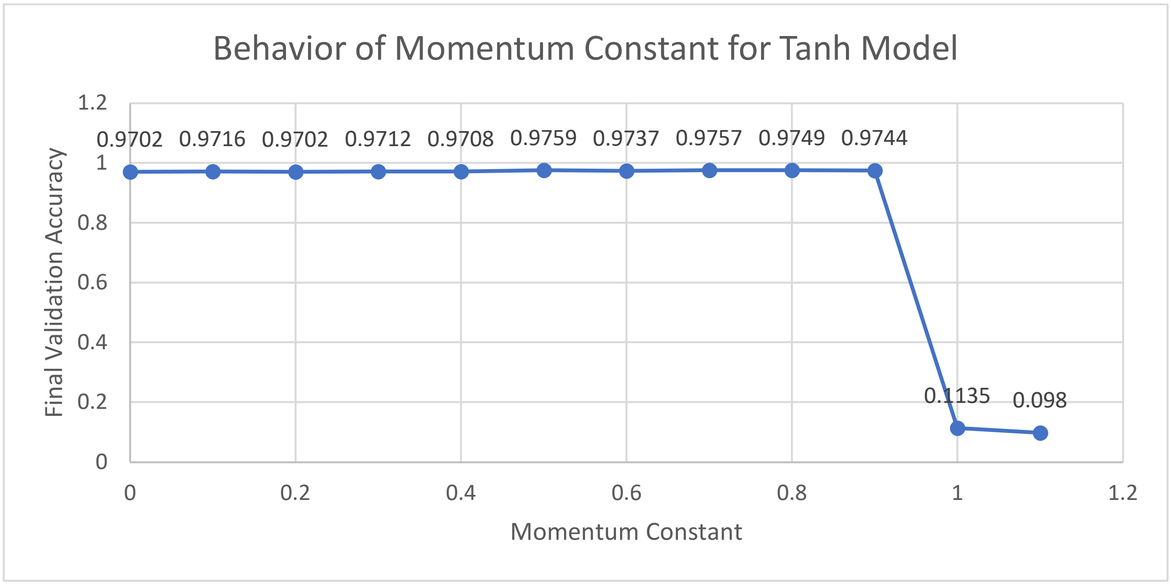


Graph 3: Linear Relationship Between Momentum and Learning Rate

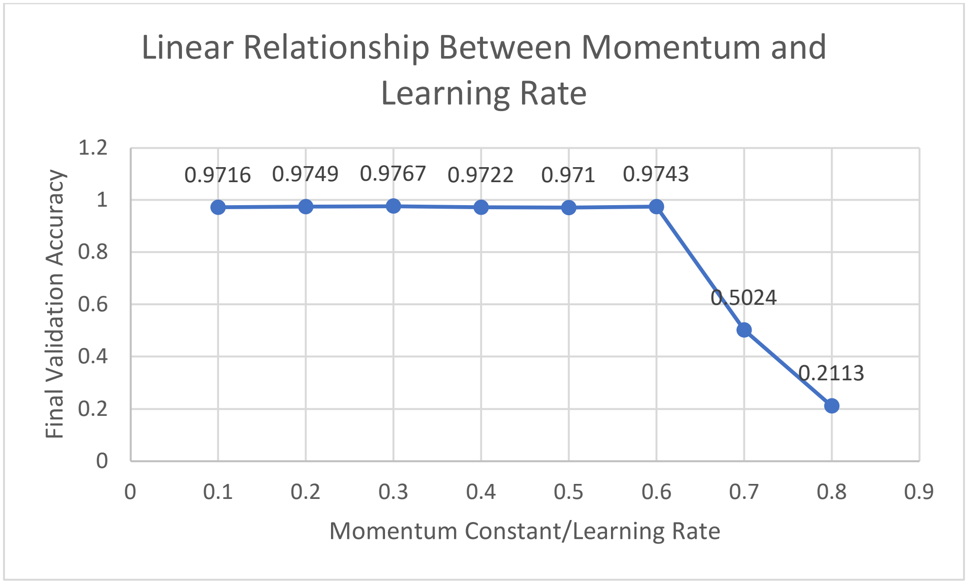
From all this data, we can see that the greatest value of final validation accuracy is observed in graph 1 where the learning rate is lr = 0.4 and the momentum is 0. The best final validation accuracy for momentum comes at 0.4, but a lr = 0.4 along with a momentum of 0.4 did not increase the best final validation accuracy. Now I will proceed to see the behavior in the same 3 ways for the 2nd best architecture.



Graph 4: Behavior of Learning Rate for Tanh Model



Graph 5: Behavior of Momentum Constant for Tanh Model



Graph 6: Linear Relationship Between Momentum and Learning Rate

The best final validation accuracy comes from a momentum of 0.5 learning rate of 0.1 in graph 5, and a learning rate of 0.4 and momentum of 0. Now the linear behavior didn't show much, but offsetting and checking the final validation accuracy for learning rate of 0.4 and momentum of 0.5 the final validation accuracy is 0.9745. So unfortunately increasing both doesn't give a great effect. So, the best value selectable for the tanh architecture is a momentum of 0.5 and learning rate of 0.1. But this accuracy is still not higher than the relu architecture. Therefore, the best architecture observable comes from the 784x64x64x10 architecture with 2 relu hidden layers and a SoftMax output layer and a learning rate of lr = 0.4, momentum of 0 which yields a final validation accuracy of val_acc = 0.9764.