Michael Ferko

Dr. Jacek Zurada

ECE 614 Deep Learning

24 February 2020

Lab 2: Classification

Objective:

Build a multilayer perceptron-based classifier model for the Fashion-MNIST Image Dataset. Utilize/start from the LAB 2 coding and modify it according to the tasks below. The data source is https://github.com/zalandoresearch/fashion-mnist. To become more familiar with the data, accessing and checking this paper will be helpful: https://arxiv.org/pdf/1708.07747.pdf.

Tasks

1. Try several architectures, use RELUs and linear layer(s)

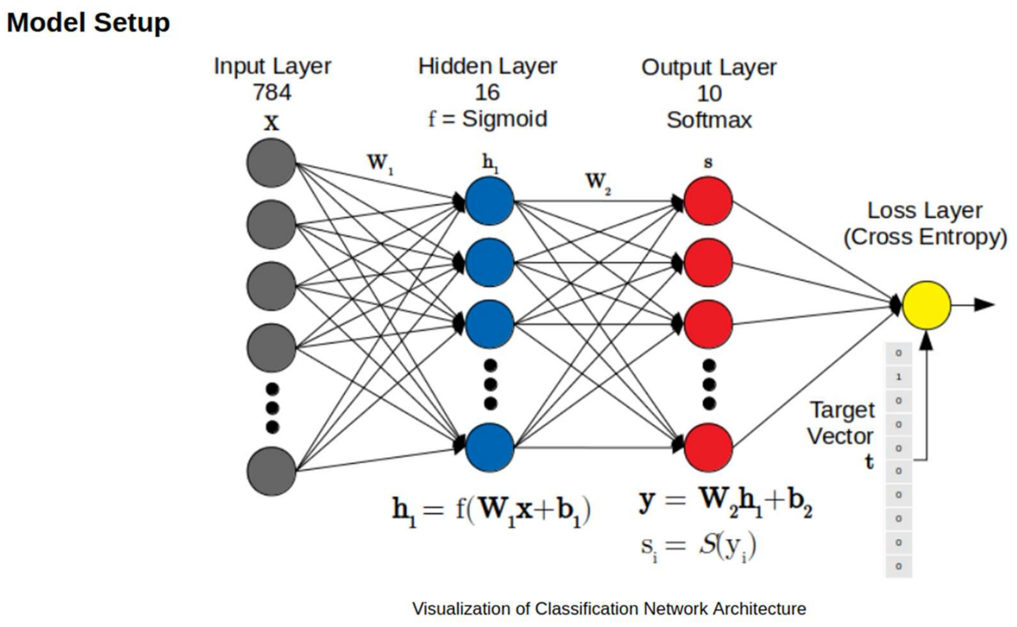
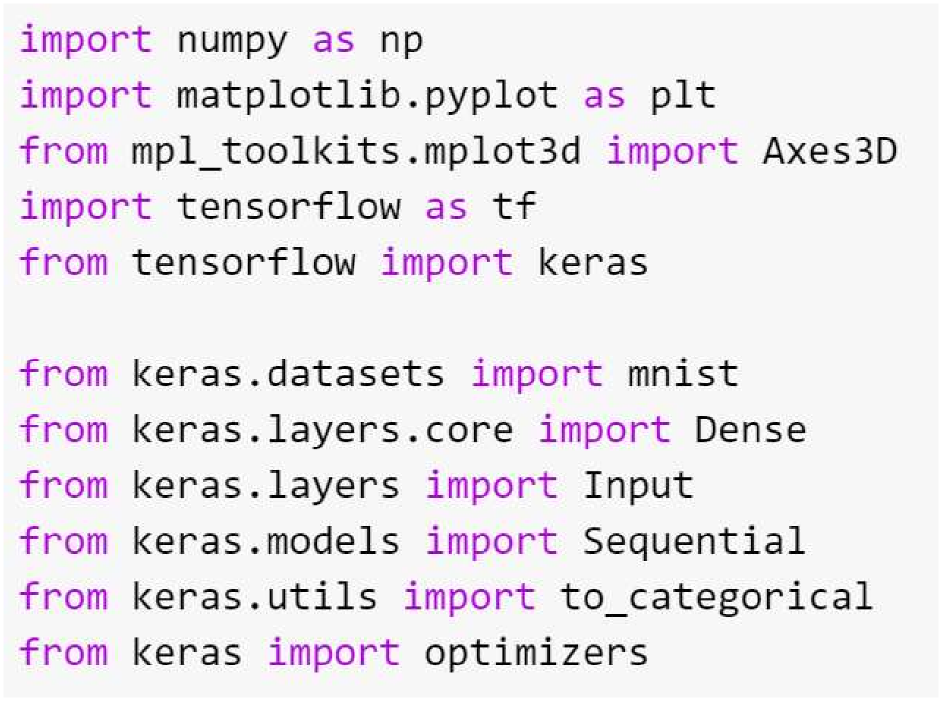


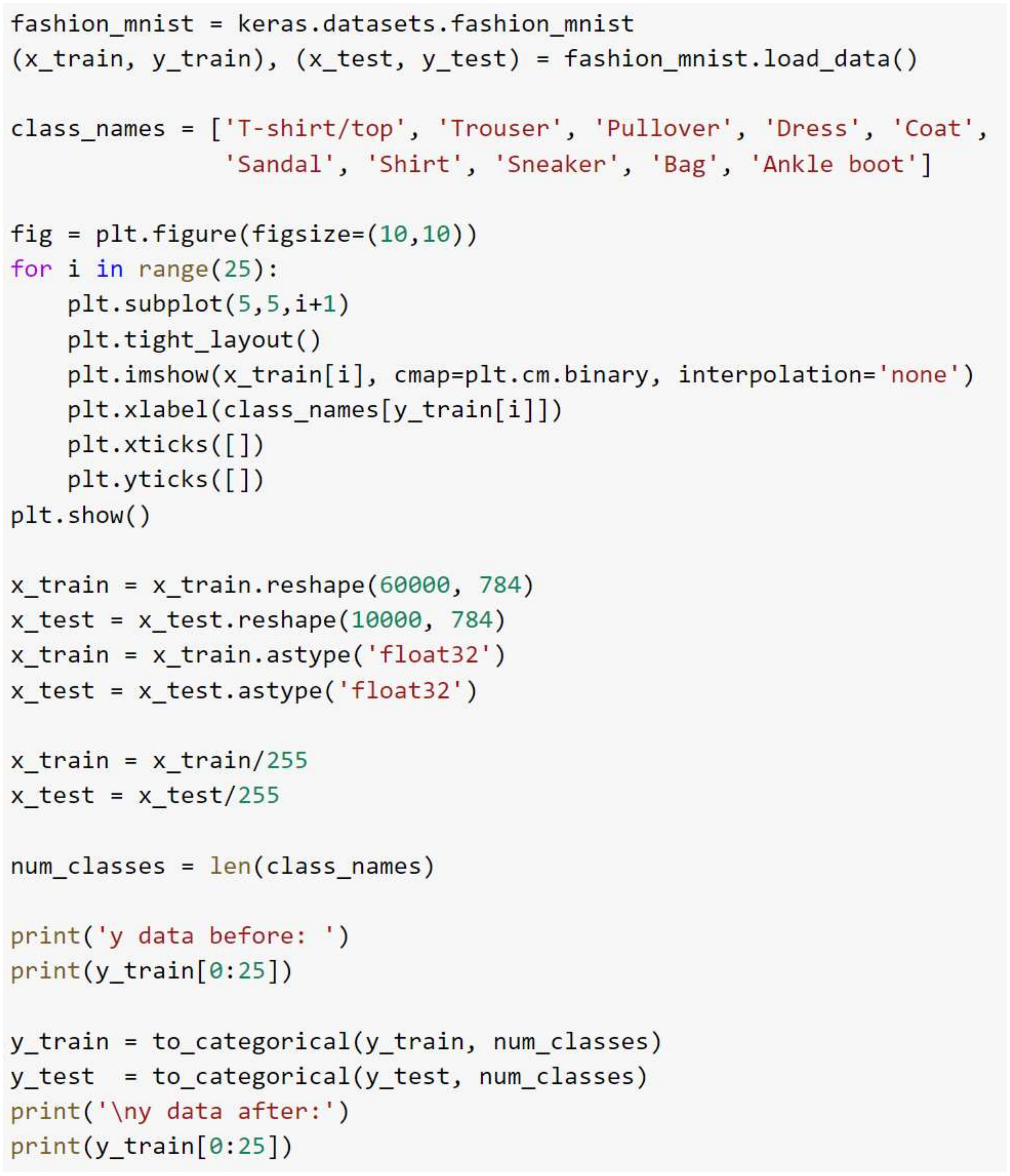
Figure 1: Visualization of Classification Network Architecture



```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import tensorflow as tf
from tensorflow import keras

from keras.datasets import mnist
from keras.layers.core import Dense
from keras.layers import Input
from keras.models import Sequential
from keras.utils import to_categorical
from keras import optimizers
```

Figure 2: Import Keras from Tensorflow



```
fashion_mnist = keras.datasets.fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
               'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']

fig = plt.figure(figsize=(10,10))
for i in range(25):
    plt.subplot(5,5,i+1)
    plt.tight_layout()
    plt.imshow(x_train[i], cmap=plt.cm.binary, interpolation='none')
    plt.xlabel(class_names[y_train[i]])
    plt.xticks([])
    plt.yticks([])
plt.show()

x_train = x_train.reshape(60000, 784)
x_test = x_test.reshape(10000, 784)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

x_train = x_train/255
x_test = x_test/255

num_classes = len(class_names)

print('y data before: ')
print(y_train[0:25])

y_train = to_categorical(y_train, num_classes)
y_test  = to_categorical(y_test, num_classes)
print('\ny data after:')
print(y_train[0:25])
```

Figure 3: Changes Made to Data Loading and Reshaping of Fashion _MNIST

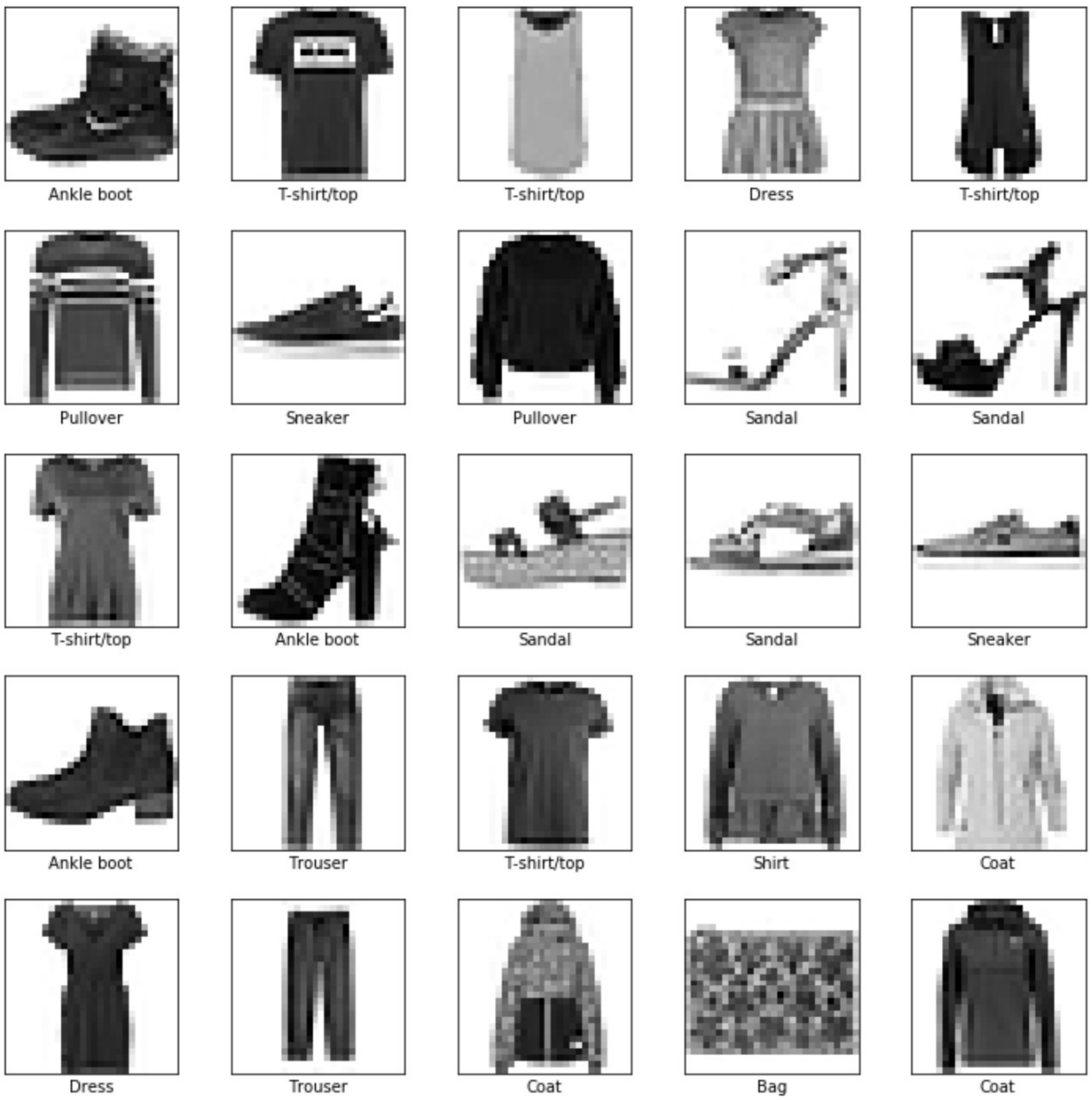
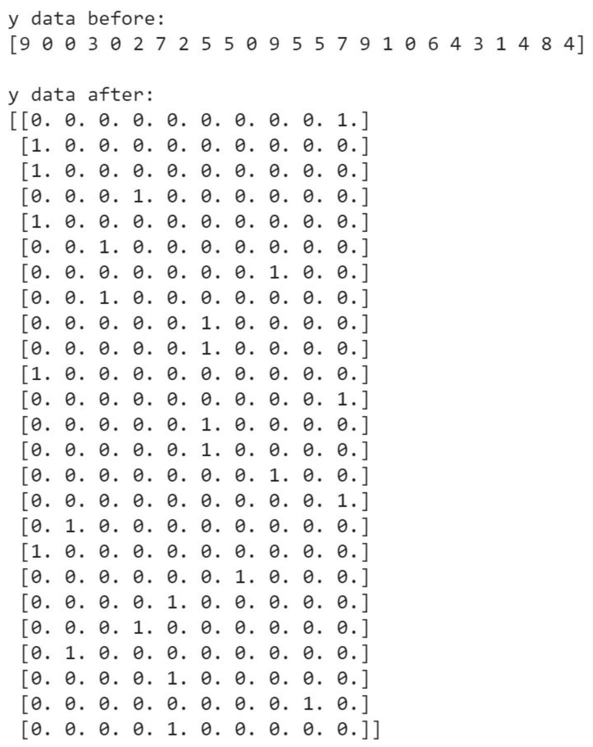


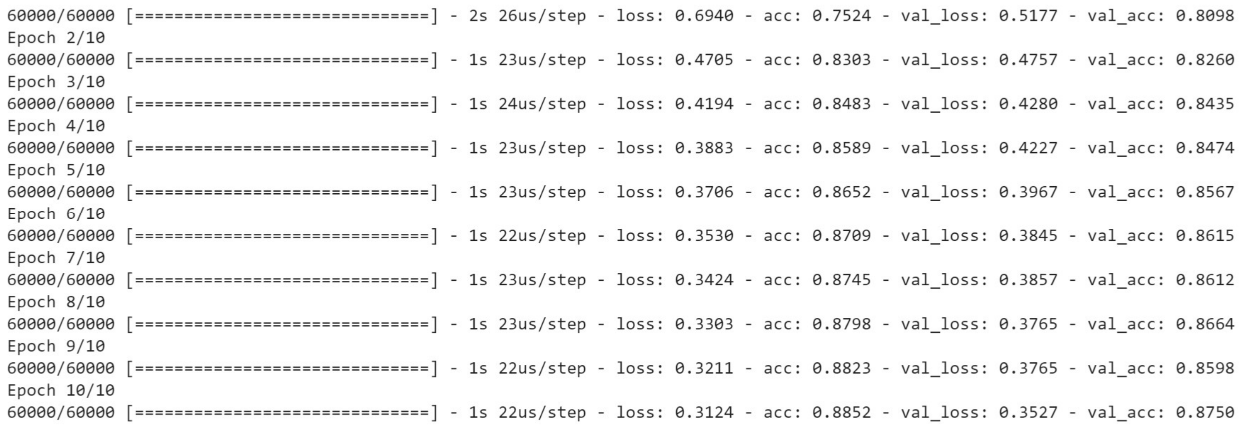
Figure 4: First 25 Training Images



```
y data before:
[9 0 0 3 0 2 7 2 5 5 0 9 5 5 7 9 1 0 6 4 3 1 4 8 4]

y data after:
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]]
```

Figure 5: Comparison of First 25 Image Training Data before and after classification



```
60000/60000 [==============================] - 2s 26us/step - loss: 0.6940 - acc: 0.7524 - val_loss: 0.5177 - val_acc: 0.8098
Epoch 2/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.4705 - acc: 0.8303 - val_loss: 0.4757 - val_acc: 0.8260
Epoch 3/10
60000/60000 [==============================] - 1s 24us/step - loss: 0.4194 - acc: 0.8483 - val_loss: 0.4280 - val_acc: 0.8435
Epoch 4/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.3883 - acc: 0.8589 - val_loss: 0.4227 - val_acc: 0.8474
Epoch 5/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.3706 - acc: 0.8652 - val_loss: 0.3967 - val_acc: 0.8567
Epoch 6/10
60000/60000 [==============================] - 1s 22us/step - loss: 0.3530 - acc: 0.8709 - val_loss: 0.3845 - val_acc: 0.8615
Epoch 7/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.3424 - acc: 0.8745 - val_loss: 0.3857 - val_acc: 0.8612
Epoch 8/10
60000/60000 [==============================] - 1s 23us/step - loss: 0.3303 - acc: 0.8798 - val_loss: 0.3765 - val_acc: 0.8664
Epoch 9/10
60000/60000 [==============================] - 1s 22us/step - loss: 0.3211 - acc: 0.8823 - val_loss: 0.3765 - val_acc: 0.8598
Epoch 10/10
60000/60000 [==============================] - 1s 22us/step - loss: 0.3124 - acc: 0.8852 - val_loss: 0.3527 - val_acc: 0.8750
```

Figure 6: Initial Model Training

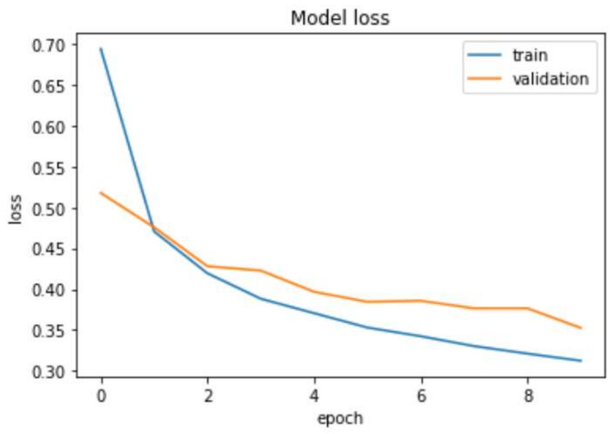


Figure 7(a): Initial Model Loss

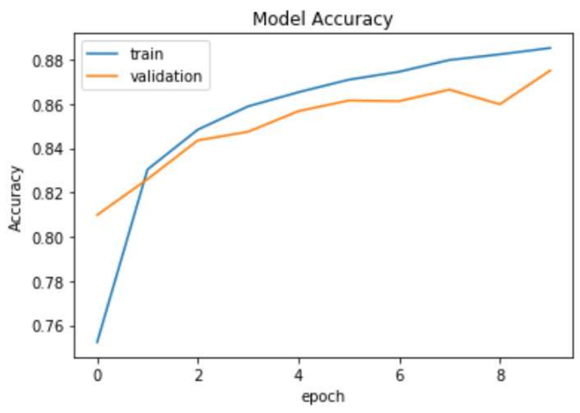


Figure 7(b): Initial Model Accuracy

This Initial Model is the 784x64x64x10 architecture with 1 784 neuron input layer 2 64 neuron relu hidden layers and a 10 neuron softmax output layer. For simplification I started with a learning rate of lr = 0.1 and a momentum of zero. Next I will make a table just as in Lab 2 to observe the validation accuracy of several different architectures to fulfill Task 1. First I compared the number of layers to see which had higher validation accuracies for the number of neurons in each layer.

Activation Function	784x16x10 Architecture	784x32x10 Architecture	784x64x10 Architecture
relu	0.8482	0.8579	0.8624
X	784x16x16x10 Architecture	784x32x32x10 Architecture	784x64x64x10 Architecture
relu	0.8566	0.8568	0.875

Table 1: Comparison of Several Architectures

I noticed that the 128 neuron layers seemed to fail and give bad validation accuracies of 0.1 so I eliminated using 128 neurons. The 2 hidden layers of 64 neurons seems to be the best architecture just as it was in Lab 2. This validation accuracy is very close to the highest validaion accuracies found in the "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms" Article referenced in the Lab 2 assignment instructions. The article compares various classifiers for a few parameters and the MNIST data set used in Lab 2 vs the Fashion-MNIST data set we use in the Lab 2 assignment. The highest accuracy for the Fashion-MNIST listed is for the Linear SVC (Support Vector Classifier) at an accuracy of 0.897 for parameters c = 10 and kernel = rbf (radial basis function).

2. Next, I will explore various learning and momentum constants. After simulating a learning rate of lr = 0.2, I found only 3 architectures worked for a changing learning rate, so I only observed those architectures then after.

Learning Rate (lr)	784x16x10 Architecture	784x32x10 Architecture	784x64x10 Architecture
0.2	0.8554	0.1	0.8645
X	784x16x16x10 Architecture	784x32x32x10 Architecture	784x64x64x10 Architecture
0.2	0.1	0.1	0.8647
X	784x16x10 Architecture	784x64x10 Architecture	784x64x64x10 Architecture
0.3	0.832	0.1	0.1

Table 2: Validation Accuracy for a change in Learning Rate

Clearly, the models fail to produce accurate validation accuracies for a change in learning rate. Now, I will change the learning rate back to lr=0.1 and observe the changes for momentum. After trying a momentum of 0.3 and 0.4, I observed that the 2 layers of 32 neurons had a higher validation accuracy than the single layer of 32 neurons, so I will continue observing the 2 layers of 32 instead of the single layer of 32.

Momentum	784x16x10 Architecture	784x32x10 Architecture	784x64x10 Architecture
0.1	0.8526	0.8622	0.8654
X	784x16x16x10 Architecture	784x32x32x10 Architecture	784x64x64x10 Architecture
0.1	0.8463	0.8559	0.8691
X	784x16x10 Architecture	784x32x10 Architecture	784x64x10 Architecture
0.2	0.8478	0.8518	0.8643

X	784x16x16x10 Architecture	784x32x32x10 Architecture	784x64x64x10 Architecture
0.2	0.8406	0.8611	0.1
X	784x16x10 Architecture	784x32x10 Architecture	784x64x10 Architecture
0.3	0.8463	0.8503	08692
X	784x16x16x10 Architecture	784x32x32x10 Architecture	784x64x64x10 Architecture
0.3	0.1	0.8572	X
X	784x16x10 Architecture	784x32x10 Architecture	784x64x10 Architecture
0.4	0.8479	0.8593	0.8704
X	784x16x16x10 Architecture	784x32x32x10 Architecture	784x64x64x10 Architecture
0.4	X	0.8635	X
X	784x16x10 Architecture	784x32x32x10 Architecture	784x64x10 Architecture
0.5	0.8526	0.8702	0.8738
X	784x16x10 Architecture	784x32x32x10 Architecture	784x64x10 Architecture
0.6	0.8542	0.8671	0.8735
X	784x16x10 Architecture	784x32x32x10 Architecture	784x64x10 Architecture
0.7	0.8512	0.1	0.8678
X	784x16x10 Architecture	784x32x32x10 Architecture	784x64x10 Architecture
0.8	0.8531	X	0.8672
X	784x16x10 Architecture	784x32x32x10 Architecture	784x64x10 Architecture
0.9	0.8389	X	0.8530

X	784x16x10 Architecture	784x32x32x10 Architecture	784x64x10 Architecture
1.0	0.1702	X	0.2390

Table 3: Validation Accuracies for a changing Momentum

The peak of validation accuracy for the Fashion-MNIST data seems to come from the 784x64x10 architecture. Even changing the learning rate and momentum did not change that outcome. As far as the peak of learning rate and momentum, I would say the learning rate didn't seem to make a big difference. However, the momentum seemed to get a validation accuracy close to the highest 0.87-ish with the trade-off of reaching the result faster. From Table 3, I would say that the momentum peaked at 0.5 and steadily declined for the 3 architectures of a single 16 neuron layer, 2 32 neuron layers and the single 64 neuron layer architecture. Based on these results, I will continue to only observe the three architectures that gave the best validation accuracy results.

3. Explore L2 and L1 regularizations

Trying to simplify the model by adjusting for overfitting. Checking to see which weight adjustment will regularize the loss function best.

Regularization

λ = regularization strength (hyperparameter)

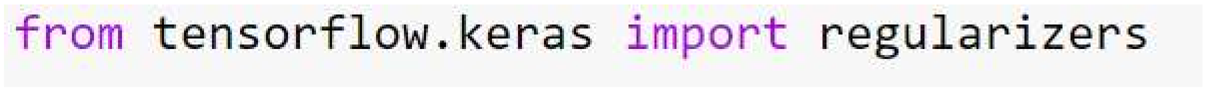
$$L(W) = \frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W)$$

Figure 8: Equation for Loss Data Function With Adjustment for Weight Regularization

L1 (Laplace) Regularization: A type of regularization that penalizes weights in proportion to the sum of the absolute values of the weights. In models relying on sparse features, L1 regularization helps drive the weights of irrelevant or barely relevant features to exactly 0, which removes those features from the model. Contrast with L2 regularization.

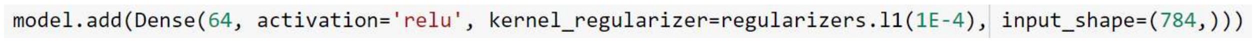
$$R(W) = \sum_k \sum_l |W_{k,l}|$$

Figure 9: L1 Weight Regularization Equation



```
from tensorflow.keras import regularizers
```

Figure 10: Import Used for Regulizers



```
model.add(Dense(64, activation='relu', kernel_regularizer=regularizers.l1(1E-4), input_shape=(784,)))
```

Figure 11: Code Changes made for L1 Regularization in Loss Function

L1 Regularization Value	784x16x10 Architecture	784x32x32x10 Architecture	784x64x10 Architecture
0.001	0.8214	0.7956	0.7961
1E-4	0.8504	0.8533	0.8596
1E-5	0.8485	0.8438	0.8576

Table 4: Testing L1 Regularization Values

Clearly, a L1 regularization value of 1E-4 is the better fit for all architectures. This doesn't seem to be making the validation accuracy any better. Now I will check L2 Regularization where I just change L1 to L2 in the code.

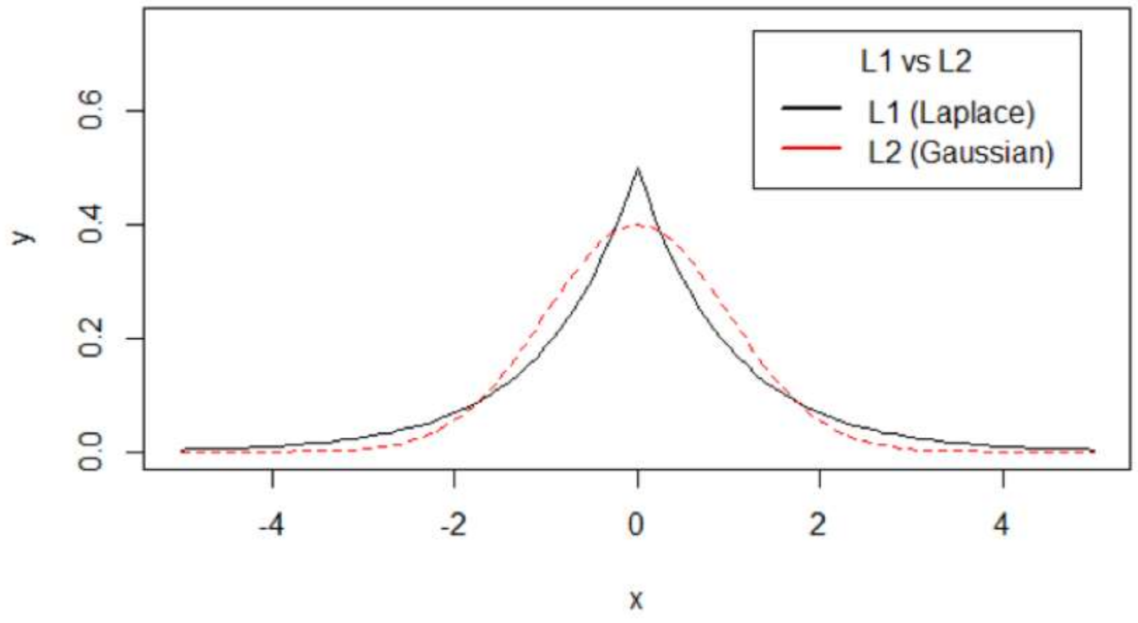


Figure 12: Graphical Comparison of L1 and L2 Regularizations

L2 (Gaussian) Regularization: A type of regularization that penalizes weights in proportion to the sum of the squares of the weights. L2 regularization helps drive outlier weights (those with high positive or low negative values) closer to 0 but not quite to 0. (Contrast with L1 regularization.) L2 regularization always improves generalization in linear models.

L2 Regularization Value	784x16x10 Architecture	784x32x32x10 Architecture	784x64x10 Architecture
0.001	0.8373	0.8309	0.8486
1E-4	0.8363	0.8596	0.8641
1E-5	0.8501	0.8441	0.8445

Table 5: Testing L2 Regularization Values

As we can see from Table 5, the best Regularization values in general are for 1E-4. So now we can create a complete Table for L1 and L2 Regularization for all architectures for a value of 1E-4.

X	784x16x10 Architecture	784x32x10 Architecture	784x64x10 Architecture
L1 @ 1E-4	0.8504	0.8505	0.8596
L2 @ 1E-4	0.8363	0.8635	0.8641
X	784x16x16x10 Architecture	784x32x32x10 Architecture	784x64x64x10 Architecture
L1 @ 1E-4	0.8401	0.8533	0.8686
L2 @ 1E-4	0.8514	0.8596	0.8648

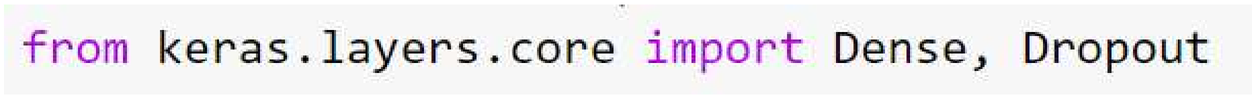
Table 6: Validation Accuracies for L1/L2 Regularization of 6 Different Architectures

I also checked the multi-hidden layers to see if regularization of just the first layer or just the second layer would have much of an effect, but I didn't notice too much of an increase or decrease to conclude much of a difference.

By comparing the results from Table 6 to the results of Table 1 we see an increase of validation accuracy based on L2 Regularization of the following architectures: 784x32x32x10, 784x64x10 and 784x32x10. We also see an increase in validation accuracy for L1 Regularization for the 784x16x10 architecture, but I am skeptical to see a conclusive increase. In fact I would say that in general L2 regularization seems to increase validation accuracy for half the architectures, making it the prime regularization method over L1.

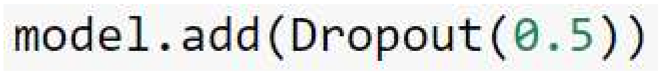
4. Check if other Keras options (like Dropout or Optimizers) would help for your best models:

Dropout: A form of regularization useful in training neural networks. Dropout regularization works by removing a random selection of a fixed number of the units in a network layer for a single gradient step. The more units dropped out, the stronger the regularization. This is analogous to training the network to emulate an exponentially large ensemble of smaller networks.



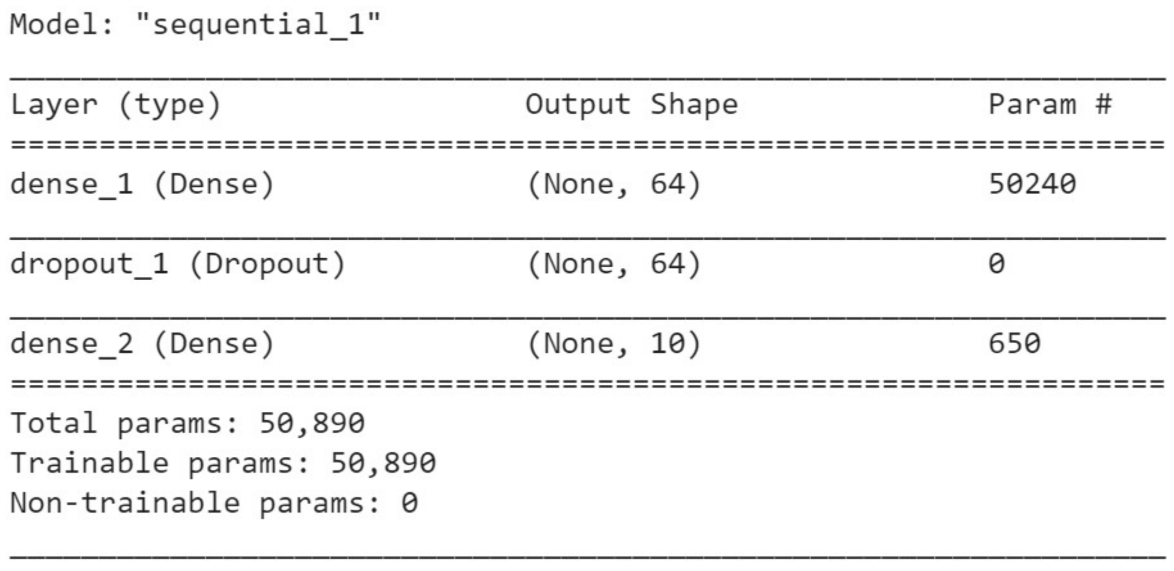
```
from keras.layers.core import Dense, Dropout
```

Figure 13: Import Adjustment for Dropout



```
model.add(Dropout(0.5))
```

Figure 14: Code Used to Create Dropout Layers



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_1 (Dense)              (None, 64)                50240     
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0         
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650       
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

Figure 15: Model Setup from Code Adjustments

Next, I need to see how adjusting the Dropout rate behaves:

Dropout Rate	784x16x10 Architecture	784x32x32x10 Architecture	784x64x10 Architecture
0.2	0.84	0.8509	0.8541
0.3	0.8301	0.8365	0.8571
0.4	0.8225	0.8355	0.8560
0.5	0.8093	0.8115	0.8543

Table 7: Behavior Validation Accuracy as a Result of increasing Dropout Rate from 0.2 to 0.5

I then tried adjusting the number of dropout layers by using 1 before or after the 2nd hidden layer and adjusting the dropout rate for the first and 2nd dropout layer but found no difference for these variations of the multilayer architectures. However, after 0.851reviewing the dropout coding I realized that a dropout needs to occur after each hidden layer anyways.

Next I tried a few values between 0.2 and 0.3. I won't add another table, but based on the same procedure as in Table 7, I narrowed the optimal Dropout rate to 0.26.

Now I will make a complete list of the effects a dropout rate of 0.26 has on the 6 architectures.

Dropout Rate	784x16x10 Architecture	784x32x10 Architecture	784x64x10 Architecture
0.26	0.8346	0.8519	0.8654
X	784x16x16x10 Architecture	784x32x32x10 Architecture	784x64x64x10 Architecture
0.26	0.8123	0.8371	0.8551

Table 8: Validation Accuracies for 6 Architectures Based on a Dropout Rate of 0.26

By comparing the results of Table 8 to the results of Table 1 we see that Dropout regularization only helped the 784x64x10 architecture. So, the current best model could be the 784x64x10 model with a learning rate of 0.5, L2 regularization of 1E-4 and a Dropout rate of 0.26. However, after running this model, I was getting a bad validation accuracy continuously folding as 0.1 for each epoch. So I adjusted the learning rate back down to 0.1. I was going to check how other optimizers would effect my model, but I haven't seen much of an increase to help the model after trying SGD w/ gradient clipping, adam, RMSProp and SMORMS3.

In conclusion I was able to achieve the best increase of validation accuracy for the L2 regularization combined with the Dropout regularization of 0.26. I consistently was getting a validation accuracy over 0.86. Unfortunately, the best model is still the 784x64x64x10 original model where I obtained 0.875. After running this same model again 5 times, I believe this final validation to be consistently the highest. However I believe the combined L2 regularization with the dropout regularization (drop out rate of 0.26) of the model 784x64x10 is the best model improvement that I had seen.