Catherine Zhu
ELEC 576 - Assignment 2

1. Visualizing a CNN with CIFAR10

a) CIFAR10 Dataset:

To import the data, I modified the function main as the following according to the trainCifarStarterCode.py. I used 800 images in each class of the training set and 100 images in each class of the testing sets. The batchsize was set to 50. The nchannels was set to 1and image size was wet to 28, since each image is grayscale and of size 28 X 28.

```
def main():
    ntrain = 800  # per class
    ntest = 100  # per class
    nclass = 10  # number of classes
    imsize = 28
    nchannels = 1
    batchsize = 50

    Train = np.zeros((ntrain * nclass, imsize, imsize, nchannels))
    Test = np.zeros((ntest * nclass, imsize, imsize, nchannels))
    LTrain = np.zeros((ntrain * nclass, nclass))
    LTest = np.zeros((ntest * nclass, nclass))

    itrain = -1
    itest = -1
    for iclass in range(0, nclass):
        for isample in range(0, ntrain):
            path = 'CIFAR10/Train/%d/Image%05d.png' % (iclass, isample)
            im = imageio.imread(path);  # 28 by 28
            im = im.astype(float) / 255
            itrain += 1
            Train[itrain, :, :, 0] = im
            LTrain[itrain, iclass] = 1  # 1-hot lable
        for isample in range(0, ntest):
            path = 'CIFAR10/Test/%d/Image%05d.png' % (iclass, isample)
            im = imageio.imread(path);  # 28 by 28
            im = im.astype(float) / 255
            itest += 1
            Test[itest, :, :, 0] = im
            LTest[itest, iclass] = 1  # 1-hot lable

    sess = tf.InteractiveSession()
```

To visualize the dataset's size, the following code were run:

```
import glob
import os
import re
import numpy as np
from skimage import io
from keras.utils.np_utils import to_categorical
from sklearn.preprocessing import OneHotEncoder

def read_cifar10():
    cifar_labels = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',
'ship', 'truck']
    file_spec = '*.png'
    train_path = os.path.join('CIFAR10/Train/*/', file_spec)
    test_path = os.path.join('CIFAR10/Test/*/', file_spec)

    train_collection= io.imread_collection(train_path)
    test_collection = io.imread_collection(test_path)
    # Normalization of pixel values (to [0-1] range)
    train_images = np.stack(train_collection).astype(float) / 255
    test_images = np.stack(test_collection).astype(float) / 255

    #Use sklearn package for one-hot-encoder
    label_encoder = OneHotEncoder()
    train_label =[]
    test_label =[]
    for file in train_collection.files:
```

```
            train_label.append(int(re.split("/",file)[2]))
        train_set=pd.DataFrame(train_label)
        train_labels=label_encoder.fit_transform(train_set[[0]]).toarray()
        #label_encoder.fit
        for file in test_collection.files:
            test_label.append(int(re.split("/",file)[2]))
        test_set=pd.DataFrame(test_label)
        test_labels=label_encoder.fit_transform(test_set[[0]]).toarray()
        return train_images, train_labels, test_images, test_labels

    X_train, y_train, X_test, y_test = read_cifar10()
    X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

The size of the training dataset, training label, testing dataset, and testing labels were: ((10000, 28, 28), (10000, 10), (1000, 28, 28), (1000, 10))

b) Train LeNet5 on CIFAR10:

Code: https://colab.research.google.com/drive/1HGXxCdpQDidrM-zc0WMA76-HI0cg5hN8?authuser=1#scrollTo=ezrU_Qq3WPVE

Each layer of the network was shown in the following:

```
# tf variable for the data, remember shape is [None, width, height, numberOfChannels]
tf_data = tf.placeholder(tf.float32, [None, imsize, imsize, nchannels])
# tf variable for labels
tf_labels = tf.placeholder(tf.float32, [None, nclass])

# --------------------------------------------------
# model
# create your model
```

1. Convolutional layer with kernel 5 x 5 and 32 filter maps followed by ReLU
2. Max Pooling layer subsampling by 2

```
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
h_conv1 = tf.nn.relu(conv2d(tf_data, W_conv1) + b_conv1)  #
h_pool1 = max_pool_2x2(h_conv1)
```

3. Convolutional layer with kernel 5 x 5 and 64 filter maps followed by ReLU
4. Max Pooling layer subsampling by 2

```
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)  #
h_pool2 = max_pool_2x2(h_conv2)
```

5. Fully Connected layer that has input 7*7*64 and output 1024

```
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 7 * 7 * 64])
h_fc1 = tf.nn.tanh(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)  #
# dropout
keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

6. Fully Connected layer that has input 1024 and output 10 (for the classes)

```
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
# --------------------------------------------------
```

```
    # loss
    # set up the loss, optimization, evaluation, and accuracy

    7. Softmax layer (Softmax Regression + Softmax Nonlinearity)

    cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=tf_labels,
    logits=y_conv))
    correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(tf_labels, 1))

    # --------------------------------------------------
    # optimization
    optimizer = tf.train.AdamOptimizer(1e-3).minimize(cross_entropy)
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')

    sess.run(tf.initialize_all_variables())

    # setup as [batchsize, width, height, numberOfChannels] and use np.zeros()
    batch_xs = np.zeros((batchsize, imsize, imsize, nchannels))
    # setup as [batchsize, the how many classes]
    batch_ys = np.zeros((batchsize, nclass))
    step_size = 3000

    train_accuracy_list = []
    train_loss_list = []
    step_list = []
    test_accuracy_list = []
    test_loss_list = []
    for i in range(step_size):  # try a small iteration size once it works then continue
        perm = np.arange(ntrain * nclass)
        np.random.shuffle(perm)
        for j in range(batchsize):
            batch_xs[j, :, :, :] = Train[perm[j], :, :, :]
            batch_ys[j, :] = LTrain[perm[j], :]
        train_accuracy = accuracy.eval(feed_dict={
            tf_data: batch_xs, tf_labels: batch_ys, keep_prob: 0.5})
        train_accuracy_list.append(train_accuracy)
        train_loss = cross_entropy.eval(feed_dict={
            tf_data: batch_xs, tf_labels: batch_ys, keep_prob: 0.5})
        train_loss_list.append(train_loss)
        step_list.append(i)

        test_accuracy = accuracy.eval(feed_dict={tf_data: Test, tf_labels: LTest, keep_prob: 1.0})
        test_accuracy_list.append(test_accuracy)
        test_loss = cross_entropy.eval(feed_dict={tf_data: Test, tf_labels: LTest, keep_prob: 1.0})
        test_loss_list.append(test_loss)

        if i % 1000 == 0 or i == step_size:
            # calculate train accuracy and print it
            print("step %d, training accuracy %g" % (i, train_accuracy))
            print("step %d, training loss %g" % (i, train_loss))

        optimizer.run(
            feed_dict={tf_data: batch_xs, tf_labels: batch_ys, keep_prob: 0.5})  # dropout only during
    training
    print("test accuracy %g" % accuracy.eval(feed_dict={tf_data: Test, tf_labels: LTest, keep_prob: 1.0}))
```

To plot the train/test accuracy and train/test loss. I have added the following code:

```
plt.figure(figsize=(12, 9))
plt.subplot(1, 2, 1)
plt.plot(np.array(step_list), train_accuracy_list, label='Train')
plt.plot(np.array(step_list), test_accuracy_list, label='Test')
plt.title('Accuracy')
plt.legend()
plt.xlabel('Steps')
plt.ylabel('Accuracy')
plt.ylim(0, 1)

plt.subplot(1, 2, 2)
plt.plot(np.array(step_list), train_loss_list, label='Train')
plt.plot(np.array(step_list), test_loss_list, label='Test')
plt.title('Cross Entropy')
```

Catherine Zhu
ELEC 576 - Assignment 2

```
        plt.legend()
        plt.xlabel('Steps')
        plt.ylabel('Loss')
```

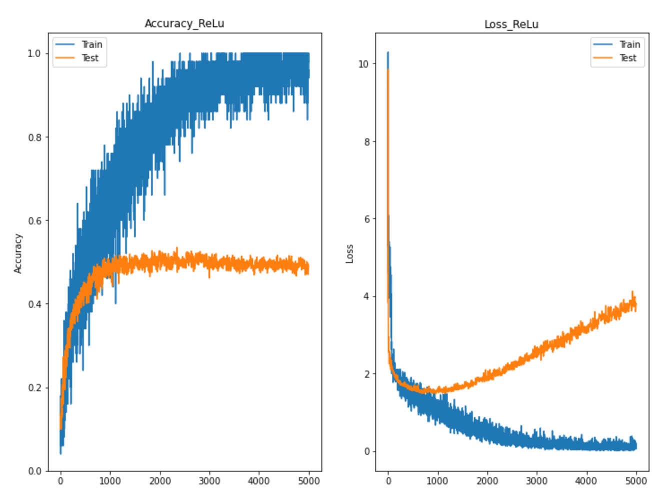
The following hyperparameters were tested in our analyses:

1. ReLu activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 +5000 iterations.
2. Tanh activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 +5000 iterations.
3. Tanh activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 10000 + 5000 iterations.
4. ReLu activation function + AdamOptimizer + learning rate 1e-4 + training sample size: 8000+5000 iterations.
5. Tanh activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 +8000 iterations.
6. ReLu activation function + AdamOptimizer + learning rate 1e-5 + training sample size: 8000 + 5000 iterations.
7. Tanh activation function + GradientDescentOptimizer + learning rate 1e-3 + training sample size: 8000 + 5000 iterations
8. sigmoid activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 + 5000 iterations.
9. Tanh activation function + Xavier initializer (`initial =tf.compat.v1.keras.initializers.glorot_normal();W = initial (shape)`) + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 + 5000 iterations.

Results:

1. ReLu activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 +5000 iterations.

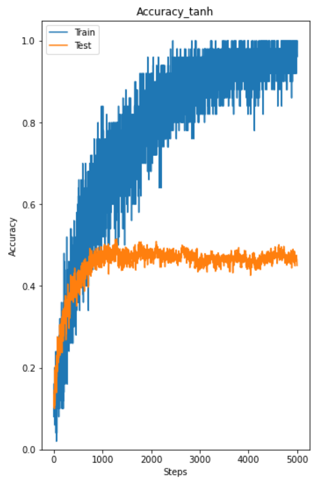
```
step 0, training accuracy 0.06
step 0, training loss 5.29658
step 1000, training accuracy 0.68
step 1000, training loss 1.07742
step 2000, training accuracy 0.82
step 2000, training loss 0.435025
step 3000, training accuracy 0.96
step 3000, training loss 0.122239
step 4000, training accuracy 0.98
step 4000, training loss 0.0586619
test accuracy 0.49
```

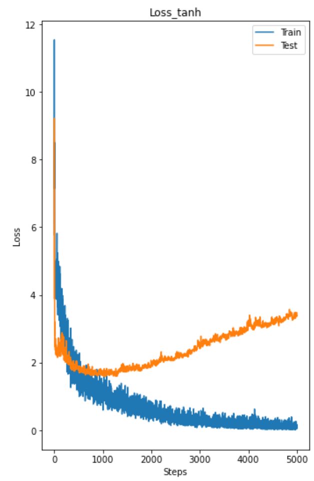


Catherine Zhu
ELEC 576 - Assignment 2

2. Tanh activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 +5000 iterations.

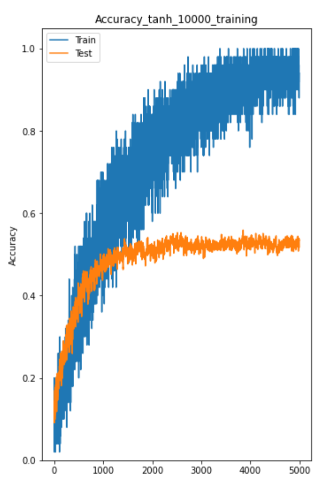
```
step 0, training accuracy 0.16
step 0, training loss 5.78253
step 1000, training accuracy 0.7
step 1000, training loss 0.885759
step 2000, training accuracy 0.76
step 2000, training loss 0.498296
step 3000, training accuracy 0.96
step 3000, training loss 0.269278
step 4000, training accuracy 0.96
step 4000, training loss 0.0822848
test accuracy 0.447
```

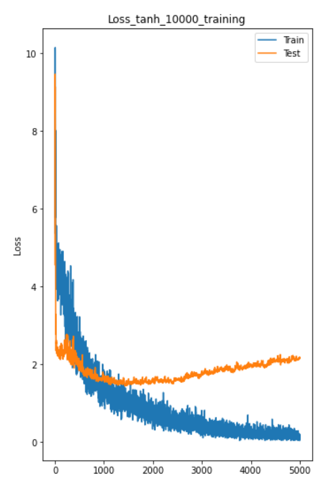




3. Tanh activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 10000 + 5000 iterations.

```
step 0, training accuracy 0.12
step 0, training loss 5.37518
step 1000, training accuracy 0.56
step 1000, training loss 1.09585
step 2000, training accuracy 0.58
step 2000, training loss 0.909169
step 3000, training accuracy 0.92
step 3000, training loss 0.192808
step 4000, training accuracy 0.96
step 4000, training loss 0.193473
test accuracy 0.519
```

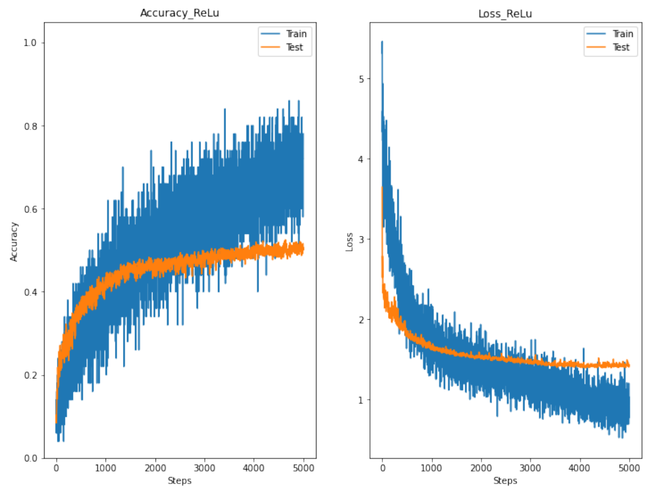




4. ReLu activation function + AdamOptimizer + learning rate 1e-4 + training sample size: 8000+5000 iterations.

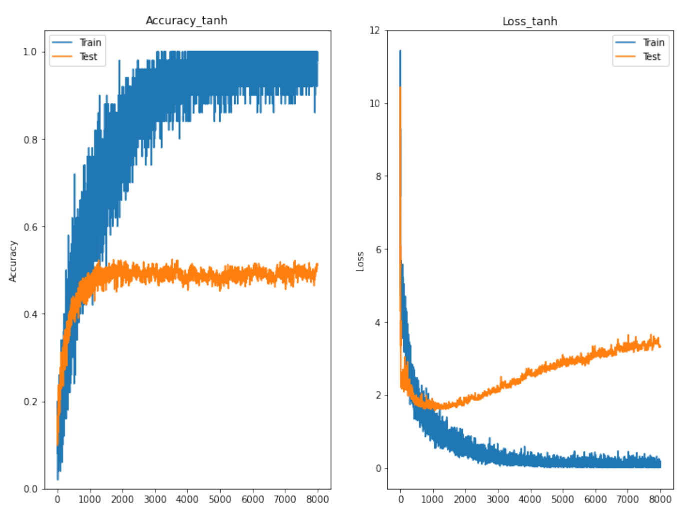
```
step 0, training accuracy 0.08
step 0, training loss 5.31366
```

```
step 1000, training accuracy 0.38
step 1000, training loss 2.16572
step 2000, training accuracy 0.48
step 2000, training loss 1.33932
step 3000, training accuracy 0.64
step 3000, training loss 0.951593
step 4000, training accuracy 0.64
step 4000, training loss 1.13764
test accuracy 0.505
```



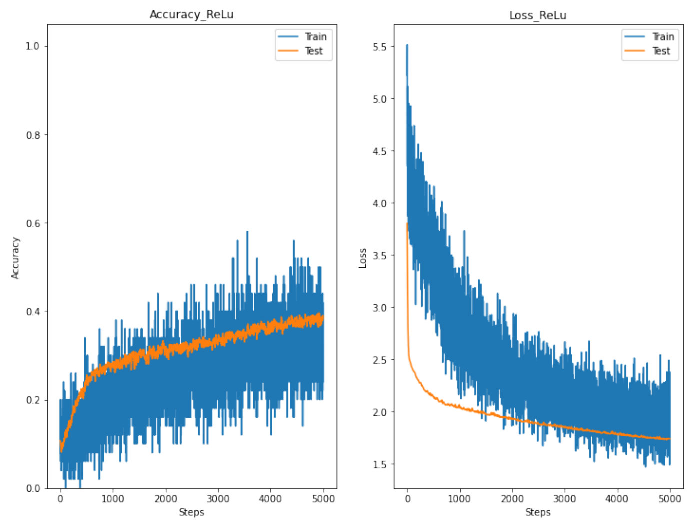
5. Tanh activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 +8000 iterations.

```
step 0, training accuracy 0.1
step 0, training loss 5.86789
step 1000, training accuracy 0.72
step 1000, training loss 0.913591
step 2000, training accuracy 0.74
step 2000, training loss 0.566394
step 3000, training accuracy 0.9
step 3000, training loss 0.201429
step 4000, training accuracy 0.9
step 4000, training loss 0.25615
step 5000, training accuracy 0.96
step 5000, training loss 0.199501
step 6000, training accuracy 0.94
step 6000, training loss 0.168871
step 7000, training accuracy 0.96
step 7000, training loss 0.105653
test accuracy 0.51
```



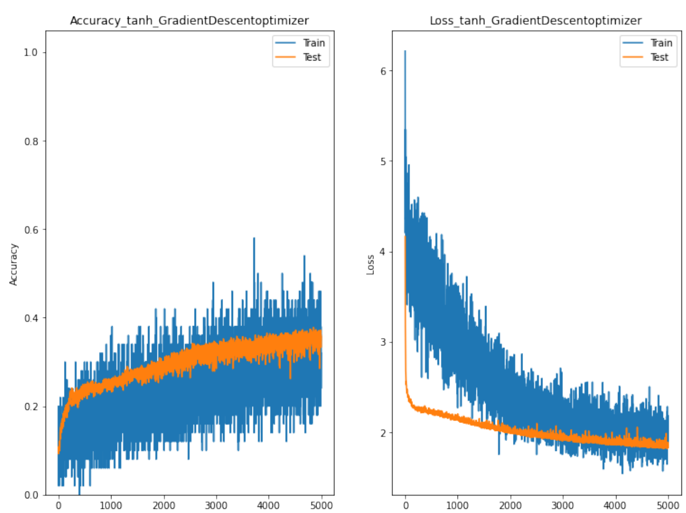
6. ReLu activation function + AdamOptimizer + learning rate 1e-5 + training sample size: 8000 + 5000 iterations.

```
step 0, training accuracy 0.08
step 0, training loss 5.21778
step 1000, training accuracy 0.16
step 1000, training loss 2.91727
step 2000, training accuracy 0.28
step 2000, training loss 2.27441
step 3000, training accuracy 0.38
step 3000, training loss 1.93839
step 4000, training accuracy 0.32
step 4000, training loss 2.25674
test accuracy 0.393
```



7. Tanh activation function + GradientDescentOptimizer + learning rate 1e-3 + training sample size: 8000 + 5000 iterations

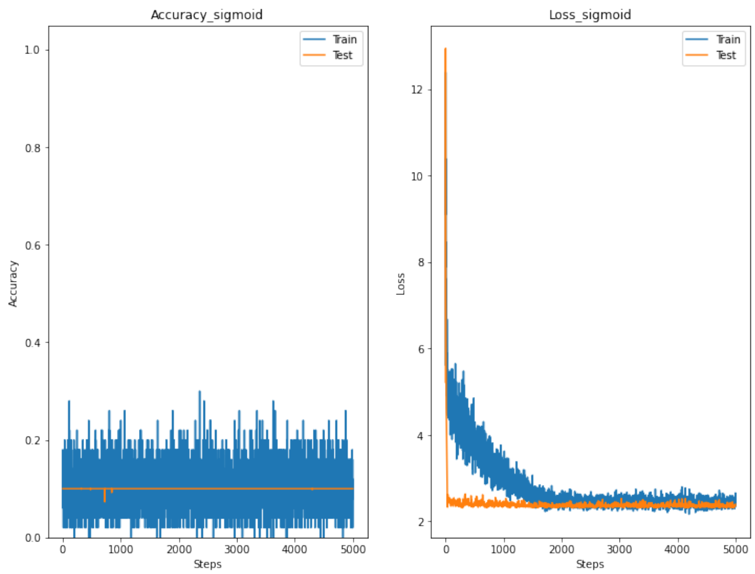
```
step 0, training accuracy 0.12
step 0, training loss 6.21313
step 1000, training accuracy 0.24
step 1000, training loss 3.01556
step 2000, training accuracy 0.26
step 2000, training loss 2.16182
step 3000, training accuracy 0.26
step 3000, training loss 2.38066
step 4000, training accuracy 0.22
step 4000, training loss 1.86151
test accuracy 0.348
```



8. sigmoid activation function + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 + 5000 iterations.

```
step 0, training accuracy 0.08
step 0, training loss 5.62024
step 1000, training accuracy 0.24
```

```
step 1000, training loss 3.01812
step 2000, training accuracy 0.12
step 2000, training loss 2.46636
step 3000, training accuracy 0.04
step 3000, training loss 2.3243
step 4000, training accuracy 0.06
step 4000, training loss 2.55688
test accuracy 0.1
```



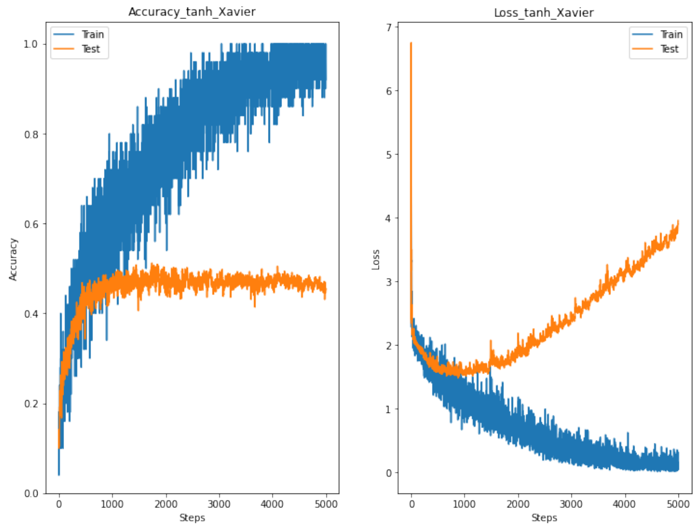
9. Tanh activation function + Xavier initializer + AdamOptimizer + learning rate 1e-3 + training sample size: 8000 + 5000 iterations

The weight initializer was modified as following. Here we use the Xavier initializer:

```
def weight_variable(shape):
        initial = tf.compat.v1.keras.initializers.glorot_normal()
        W = initial (shape)
        return tf.Variable(W)
```

```
step 0, training accuracy 0.18
step 0, training loss 2.28702
step 1000, training accuracy 0.64
step 1000, training loss 1.0656
step 2000, training accuracy 0.72
step 2000, training loss 0.648412
step 3000, training accuracy 0.9
step 3000, training loss 0.249307
step 4000, training accuracy 0.92
step 4000, training loss 0.290149
test accuracy 0.452
```

Catherine Zhu
ELEC 576 - Assignment 2



From the above analyses, we could see that ReLu performed better than Tanh activation, which performed better than sigmoid activation. As the number of iterations increased, the test accuracy also increased. As the number of the training samples increased, the accuracy increased. In addition, Adam Optimizer worked better than GradientDescent Optimizer did. With the addition of Xavier initializer, the model performed slightly better than using truncated normal with standard deviation 0.1. The learning rate 1e-4 seemed to work the best compared to 1e-3 and 1e-5, using tanh activation functions, 8000 training samples, and 5000 iterations.

c) Visualize the Trained Network: Visualize the first convolutional layer's weights.

The mean, standard deviation, and the variance of the two convolutional layers with ReLu activations were calculated using the following code:

```
# calculate statistics of the activations
h1 = h_conv1.eval(feed_dict = {tf_data: Test, tf_labels: LTest, keep_prob: 1.0})
h2 = h_conv2.eval(feed_dict = {tf_data: Test, tf_labels: LTest, keep_prob: 1.0})
mean_h1 = np.mean(np.array(h1))
std_h1 = np.std(np.array(h1))
var_h1 = np.var(np.array(h1))
mean_h2 = np.mean(np.array(h2))
std_h2 = np.std(np.array(h2))
var_h2 = np.var(np.array(h2))
print("ReLu activation 1: mean %g, standard deviation %g, variance %g" % (mean_h1, std_h1,
var_h1))
print("ReLu activation 2: mean %g, standard deviation %g, variance %g" % (mean_h2, std_h2,
var_h2))
```

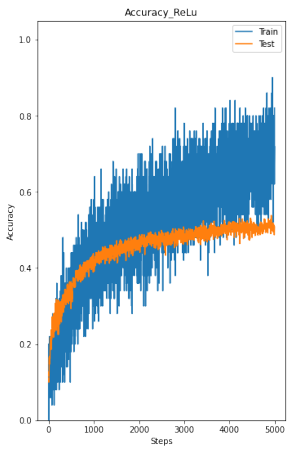
To show the first convolutional layer's weights. The following codes were added.

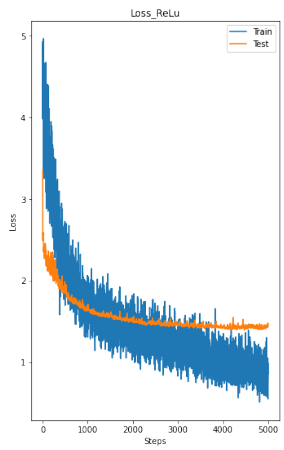
```
n_filters, ix = 32, 1
for i in range(n_filters):
f = first_conv_weight[:, :, :, i]
# specify subplot and turn off axis
ax = plt.subplot(4, 8, ix)
# plot filter channel in grayscale
plt.imshow(f[:, :, 0], cmap='gray')
ix += 1
plt.axis('off')
# show the figure
plt.show()
```

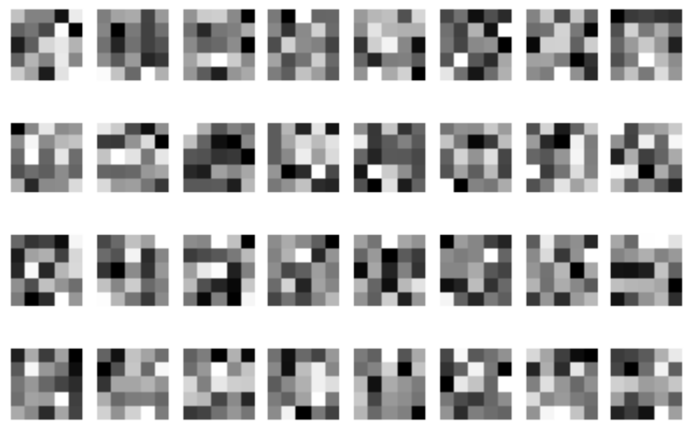
The results were shown in the following:

```
step 0, training accuracy 0.2
step 0, training loss 4.87657
step 1000, training accuracy 0.3
step 1000, training loss 1.89349
step 2000, training accuracy 0.38
```

```
step 2000, training loss 1.71414
step 3000, training accuracy 0.56
step 3000, training loss 1.4345
step 4000, training accuracy 0.78
step 4000, training loss 0.924481

test accuracy 0.508

ReLu activation 1: mean 0.0741311, standard deviation 0.0843887, variance 0.00712145
ReLu activation 2: mean 0.0240031, standard deviation 0.0536668, variance 0.00288012
```







2 . Visualizing and Understanding Convolutional Networks

<u>Paper summary</u>

This paper introduced a multi-layered Deconvolutional Network (deconvnet) to visualize and understand large convolutional network models. The network model consisted of eight convnet layers. The first five layers had a deconvnet attached via a switch, following by the two conventional fully-connected networks before the last class softmax layer. Each of the five convnet layer had set of learned filters that passes though relu activation functions then max pooled and normalized across each feature map. In the deconvnet, the feature maps that were constructed by the convnet were set as input while all other activations in the layer were muted. The deconvnet reversed the actions in the convnet by first unpooled the maxima in each pooling layer, then used a relu non-lineality to reconstruct the signals from each layer. Each deconvnet allows us to view a given convnet activation, showing distinct patterns that activate the feature map from the training samples. From the paper, the first layer seemed to pick up the essence of images, such as color blocks and region/location of the images. The second layer seemed to be able to identify the edges and the shape of the objects. As the network got deeper, the deeper layers could learn and capture more complex patterns and eventually could reconstruct whole new images that resemble the real objects. This method makes the deep of the network a vital point, as mentioned in the paper.

3 Build and Train an RNN on MNIST

a) Setup an RNN:

The setup was shown as following:
To load the RNN cell, certain modifications were needed:

```
import numpy as np
import tensorflow as tf
from tensorflow.python.ops import rnn, rnn_cell
from tensorflow.keras import optimizers
import matplotlib.pyplot as plt
%tensorflow_version 1.15
```

Catherine Zhu
ELEC 576 - Assignment 2

```
if (tf.__version__.split('.')[0] == '2'):
    import tensorflow.compat.v1 as tf
    tf.disable_v2_behavior()
```

The parameters were:

```
learningRate = 1e-3
trainingIters = 60000
batchSize = 100
displayStep = 10

nInput = 28  # we want the input to take the 28 pixels
nSteps = 28  # every 28
nHidden = 128  # number of neurons for the RNN
nClasses = 10  # this is MNIST so you know
```

The model was set up as following:

```
def RNN(x, weights, biases):
    x = tf.transpose(x, [1, 0, 2])
    x = tf.reshape(x, [-1, nInput])
    x = tf.split(x, nSteps, 0)  # configuring so you can get it as needed for the 28 pixels
    lstmCell = rnn_cell.BasicRNNCell(num_units=nHidden)
    outputs, states = rnn.static_rnn(lstmCell, x,
                                     dtype=tf.float32)
    return tf.matmul(outputs[-1], weights['out']) + biases['out']
```

Optimization, cost, evaluation, and accuracy were set up as following:

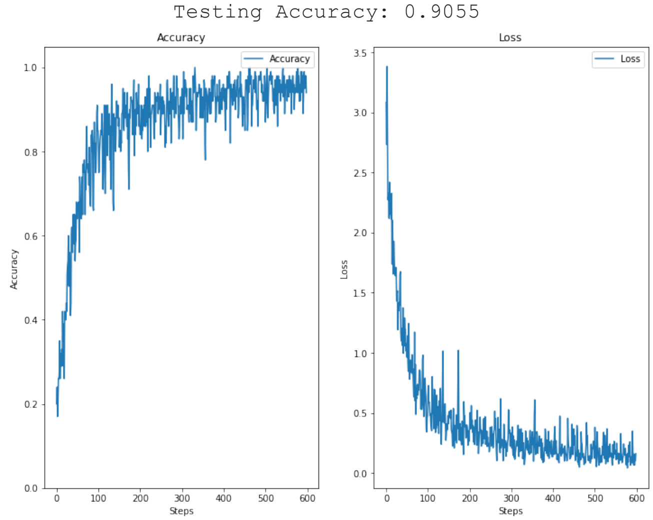
```
#optimization
#create the cost, optimization, evaluation, and accuracy
#for the cost softmax_cross_entropy_with_logits seems really good

cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=pred, labels=y))
optimizer = tf.train.RMSPropOptimizer(learning_rate=learningRate).minimize(cost)
correctPred = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correctPred, tf.float32))
```

The following parameters were tuned:

- Learning rate: 1e-3, 1e-4
- Batchsize: 50, 100
- Iteration: 60,000, 100,000. 150,000

Learning rate: 1e-3, iteration 60,000, batchsize: 100:

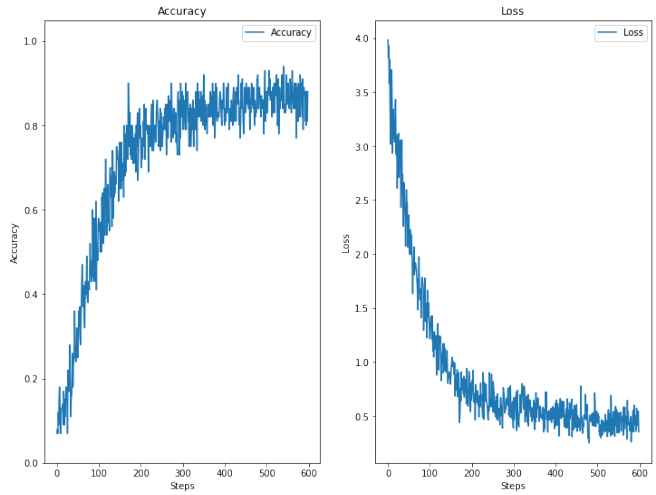
```
Testing Accuracy: 0.9055
```



Learning rate: 1e-4, iteration 60,000, batchsize: 100:

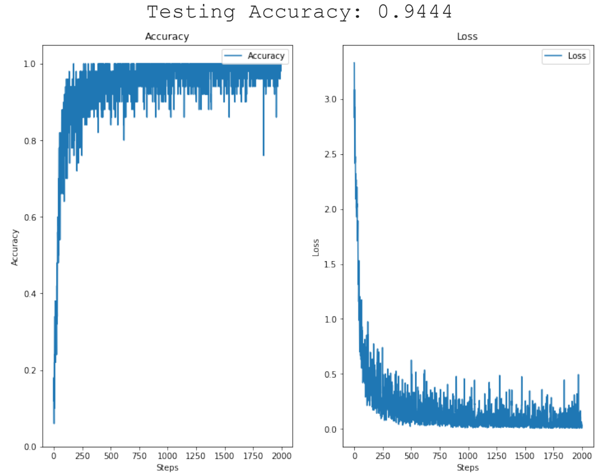
```
Testing Accuracy: 0.8637
```

Catherine Zhu
ELEC 576 - Assignment 2



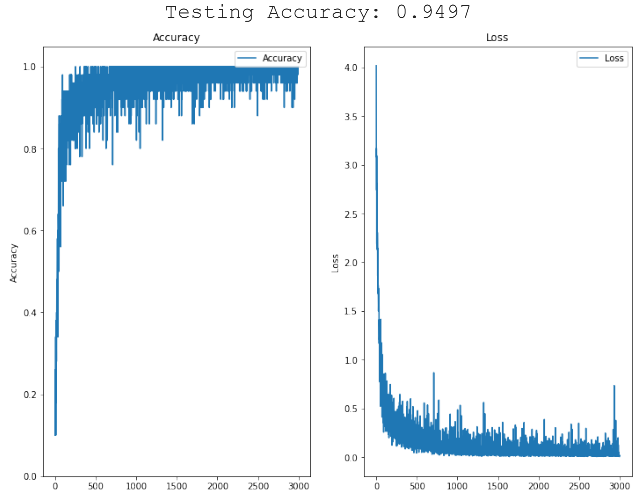
Learning rate: 1e-4, iteration 100,000, batchsize: 50:

```
Testing Accuracy: 0.9444
```



Learning rate: 1e-3, iteration 150,000, batchsize: 50:

```
Iter 149000, Minibatch Loss= 0.011603, Training Accuracy= 1.00000
Iter 149500, Minibatch Loss= 0.016334, Training Accuracy= 1.00000
Optimization finished
Testing Accuracy: 0.9497
```



The model with the 1e-3 learning rate, 150,000 iterations, and 50 batch size seemed to give the highest testing accuracy 0.95 for basic rnn.

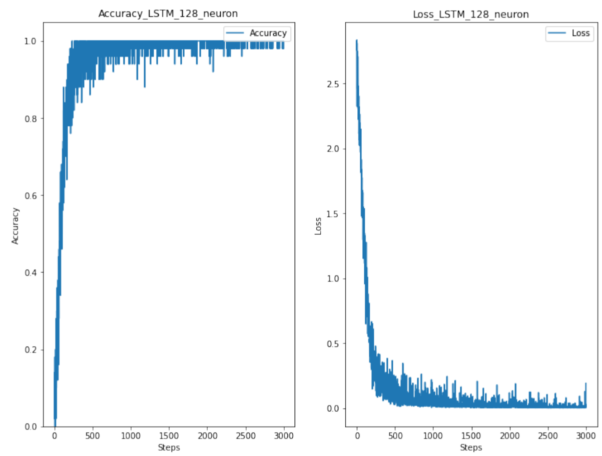
Catherine Zhu
ELEC 576 - Assignment 2

b) How about using an LSTM or GRU: change line 35 in the starter code (see below) to use LSTM and GRU instead of RNN.

LSTM:

```
lstmCell = rnn_cell.BasicLSTMCell(nHidden, forget_bias=1.0)
learningRate = 1e-3
trainingIters = 150000
batchSize = 50
displayStep = 10

nInput = 28  # we want the input to take the 28 pixels
nSteps = 28  # every 28
nHidden = 128  # number of neurons for the RNN
nClasses = 10  # this is MNIT so you know
```

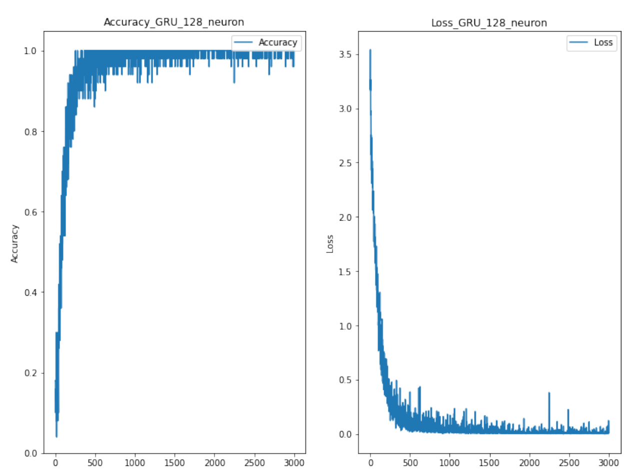
```
Testing Accuracy: 0.9801
Testing loss: 0.061787646
```



GRU result:

```
lstmCell = rnn_cell.GRUCell(nHidden)
learningRate = 1e-3
trainingIters = 150000
batchSize = 50
displayStep = 10
nInput = 28  # we want the input to take the 28 pixels
nSteps = 28  # every 28
nHidden = 128  # number of neurons for the RNN
nClasses = 10  # this is MNIT so you know
```

```
Testing Accuracy: 0.9756
Testing loss: 0.07563189
```

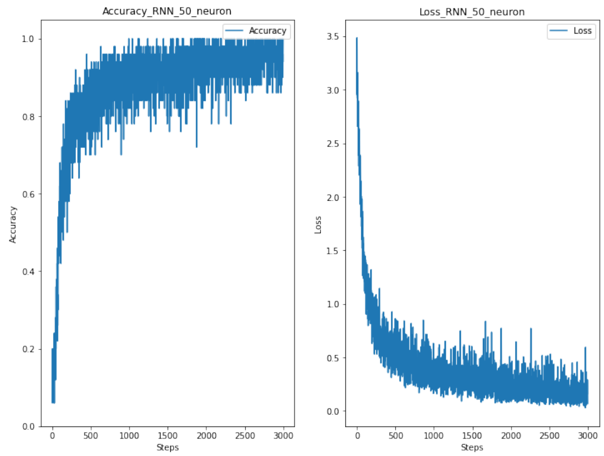


Catherine Zhu
ELEC 576 - Assignment 2

GRU seemed to perform better than LSTM, and basic RNN under the condition: with the 1e-3 learning rate, 150,000 iterations, and 50 batch size. We then changed the number of hidden units to 50 and 300 and the results are shown in the following:

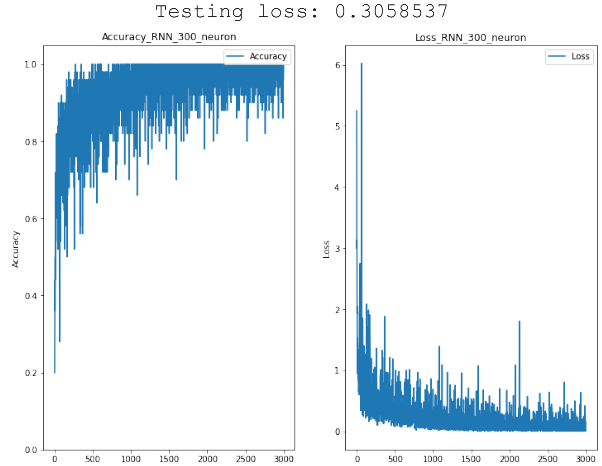
RNN_50_neurons:

```
Testing Accuracy: 0.9232
Testing loss: 0.25384653
```



RNN_300_neurons:

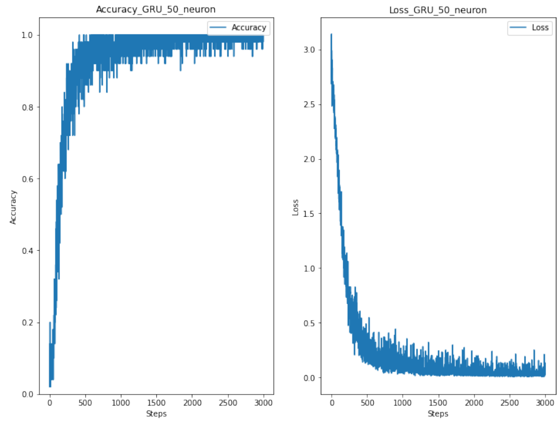
```
Testing Accuracy: 0.9207
Testing loss: 0.3058537
```



GRU_50_neurons:

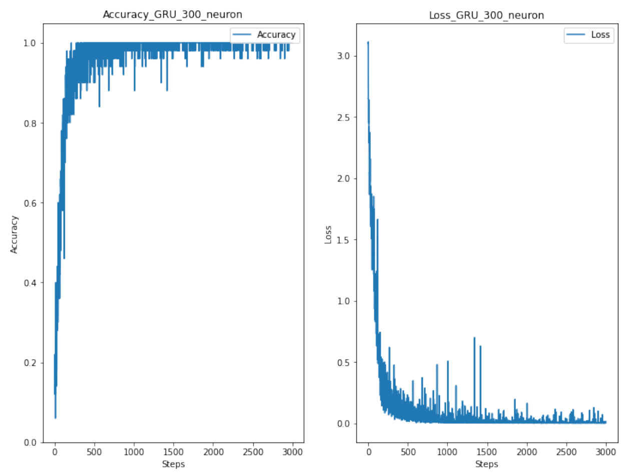
```
Testing Accuracy: 0.9728
Testing loss: 0.08430433
```

Catherine Zhu
ELEC 576 - Assignment 2



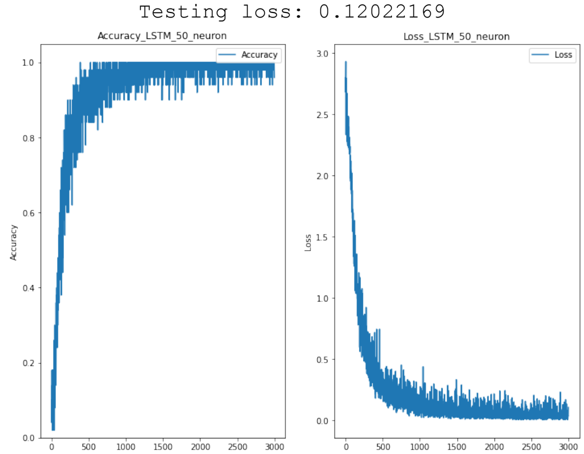
GRU_300_neurons:

```
Testing Accuracy: 0.9823
Testing loss: 0.058735345
```



LSTM_50_neurons:

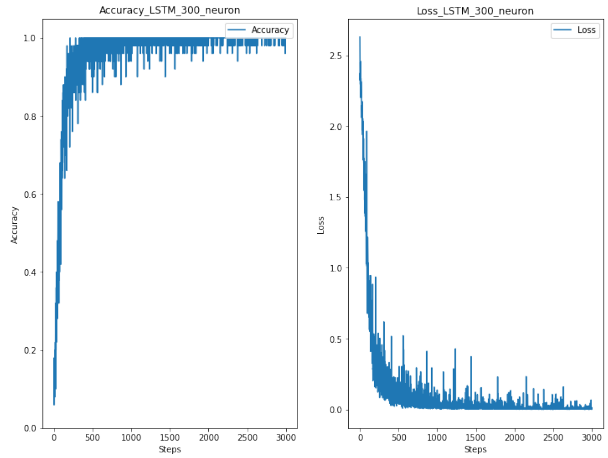
```
Testing Accuracy: 0.9639
Testing loss: 0.12022169
```



LSTM_300_neurons:

```
Testing Accuracy: 0.9821
```

Testing loss: 0.05691859



As the size of the hidden units increased from 50, 128, to 300, the testing accuracy increased slightly. However, as the number of hidden units increased, the runtime increased. Using the same parameters, the GRU model performed slightly better than the LSTM model, which performed better than the basic RNN model.

c) Compare against the CNN: Compare with training using convnet in assignment 1 and describe any similarities or differences.

CNN from assignment 1 gave test accuracy of 0.9861 which is the highest among basic_RNN, GRU, and LSTM, which were around 0.983. Both the CNN and RNN are similar that they both use neural network to learn and as the number of iterations increases, the neural network can adjust based on loss functions and finally recognize patterns and features of the data. Both RNN and CNN could suffer from the vanishing and exploding gradient problem and both types of network share the parameters across different time point to decrease the computational cost. CNN contains convolutional layers which consists of filters to transform and process the data before feeding to the next layer. CNN performs better with visual data or data that has spatial information and is considered more powerful compared to RNN. CNNs use connectivity pattern between the neurons with fixed size inputs and generates fixed size outputs. Unlike the CNN, RNN can learn from the past information in a sequence. RNN uses time-series information. The network reuses activation functions from past datapoints in the sequence as well as the new input at specific time point. RNN includes less feature compatibility but can handle arbitrary input/output lengths.

Reference:
https://www.telusinternational.com/articles/difference-between-cnn-and-rnn
https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/
https://www.tutorialspoint.com/tensorflow/tensorflow_cnn_and_rnn_difference.htm