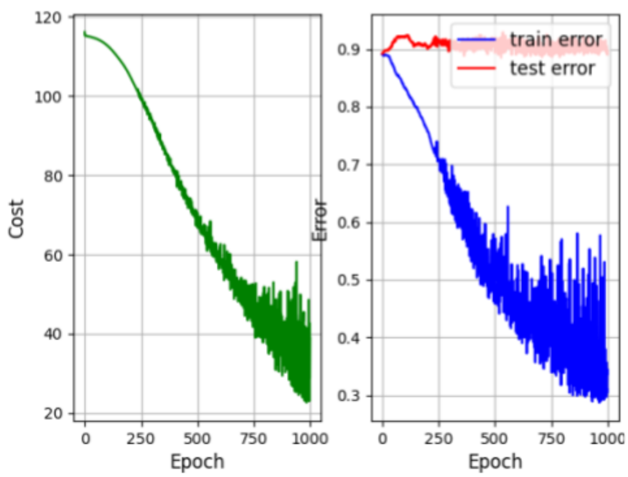
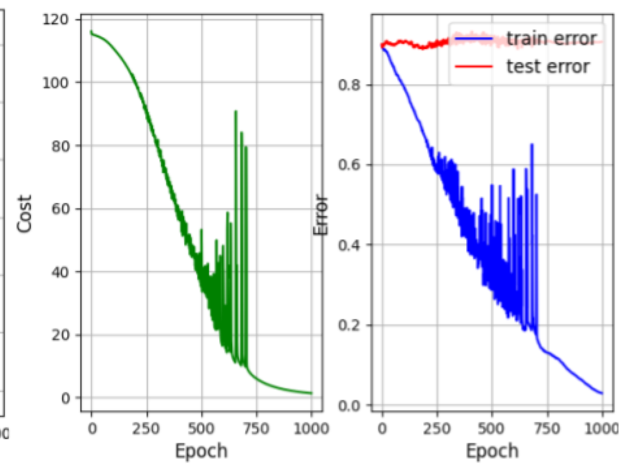
Assignment requirement:

A. **Increase the number of neurons in the two hidden layers from (32, 16) to (64, 32), compare the training/testing error in the two cases, and explain it.**
    a. *For 10% of the total image data set and hidden layers 32, 16 we get pretty terrible results with both the training data set and testing data set.*
    b. *For 10% of the total image data set and hidden layers 64, 32 we get a much better accuracy for the training data set but our testing error is still garbage*
    c. *The testing error/accuracy is terrible and makes sense considering weve decreased our sample size which hurts the generalization of our model*



10% image data set (5000 images) with (**32, 16**) hidden layers (1000 epochs)

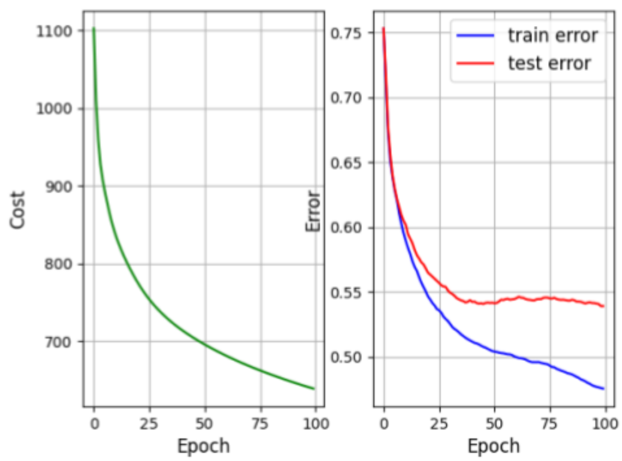
Epoch: 1000 | AvgCost: 0.778 Train/Test ACC: 0.663/0.103



10% image data set (5000 images) with (**64, 32**) hidden layers (1000 epochs)

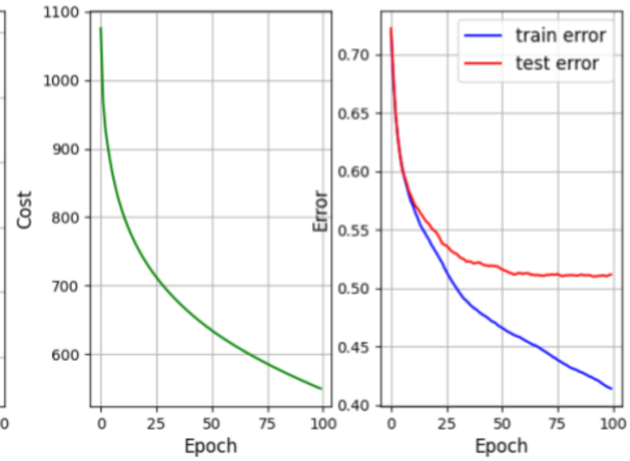
Epoch: 1000 | AvgCost: 0.027 Train/Test ACC: 0.971/0.094

    d. *For the full image data set and hidden layers 32, 16 we get much smoother accuracy curves over time. However, because it was the full data set I only did 100 epochs. It was observed for 1000 epochs that the cost and error also fluctuated quite a bit around the 500-700 epoch.*
    e. *For the full image data set and hidden layers 64, 32 we decrease our accuracy/ increase our error and decrease our cost by about 2 units.*
    f. *However, compared to the 10% data set cases, our testing error and accuracy (aka our generalization, is much better. This makes sense considering we increased our sample size which can benefit the generalization of our model*



Full 50000 image data set with (**32, 16**)
hidden layers (100 epochs)

Epoch: 100 | AvgCost: 1.278 Train/Test ACC:
0.525/0.461



Full 50000 image data set with (**64, 32**)
hidden layers (100 epochs)

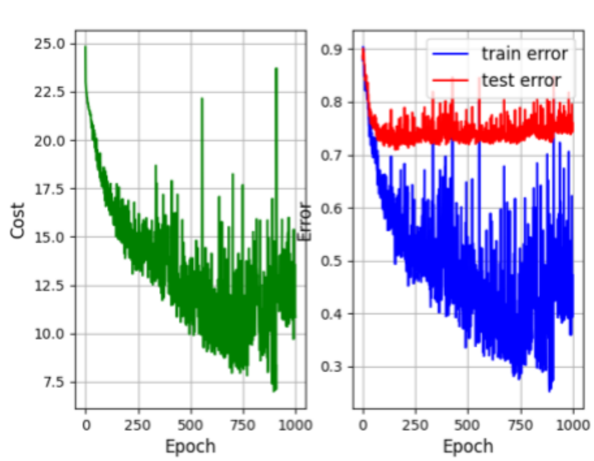
Epoch: 100 | AvgCost: 1.099 Train/Test ACC:
0.586/0.488

In general, increasing the neurons in each hidden layer has a positive effect on the training accuracy of our neural network but not necessary the testing accuracy.

##
##
##

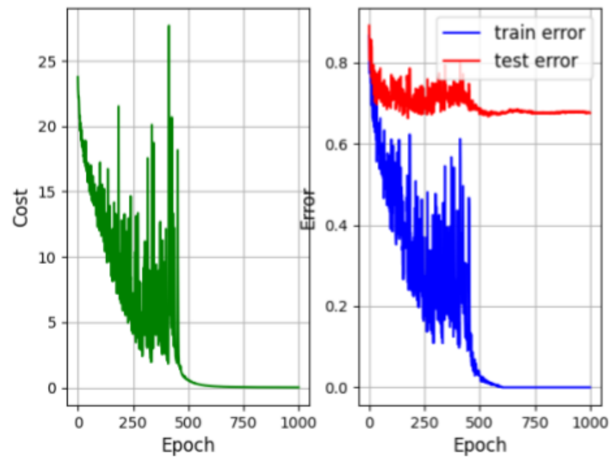
B. Replace the sigmoid function with the ReLU function, compare the training/testing error in the two cases, and explain it.

a. *Below are the results for 16 and 32 hidden layers(left) and 64 32 hidden layers(right). Compared to the sigmoid function (for the 64/32 case), with the all other variables held the same, it seems that the using the Relu activation function converges faster than the sigmoid function. That is, it brings the cost quicker to zero. It also seems that it has has generalized better than the sigmoid function by observing the increase in testing accuracy.*



10% image data set (5000 images) with **(32, 16)** hidden layers (1000 epochs) Relu Func

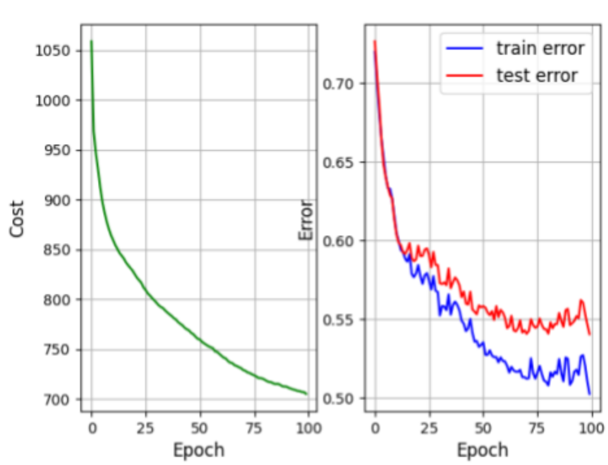
Epoch: 1000 | AvgCost: 1.084 Train/Test ACC: 0.528/0.240



10% image data set (5000 images) with **(64, 32)** hidden layers (1000 epochs) Relu Func

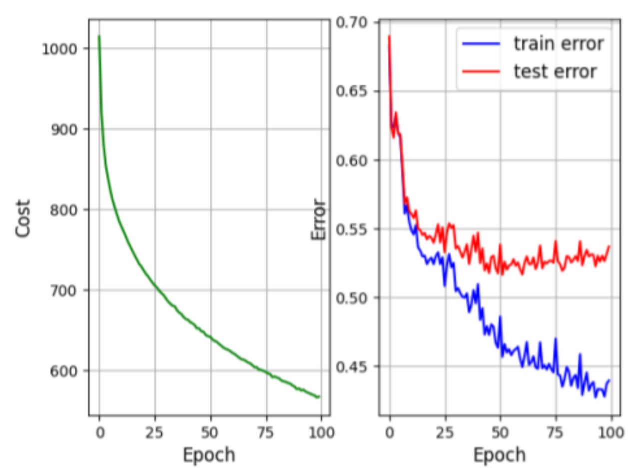
Epoch: 1000 | AvgCost: 0.001 Train/Test ACC: 1.000/0.324

b. *Below, we can see that results after using the full data set. There seems to be very little difference between the above graphs which use the Relu function and the sigmoid graphs in part A. The only noticeable difference is the increase in chatter in the accuracy for both the trained and testing results.*



Full image data set (50000 images) with **(32, 16)** hidden layers (100 epochs) Relu Func

Epoch: 100 | AvgCost: 1.411 Train/Test ACC: 0.497/0.459



Full image data set (50000 images) with **(64, 32)** hidden layers (100 epochs) Relu Func

Epoch: 100 | AvgCost: 1.134 Train/Test ACC: 0.560/0.463

Below are screenshots of my code. Notable changes occur on lines. For example on line 73 it is required to change the size to (32, 32, 3). As a result it is necessary to add line

79: flattened = tf.layers.flatten(tf_x, name='flatten'). Also it is necessary to change how we increment batch size on lines 125 and 126.

```python
import tensorflow.compat.v1 as tf

import numpy as np
import numpy.matlib
import numpy as np
import matplotlib.pyplot as plt
import random
from tensorflow.keras import datasets
from tensorflow.keras.utils import to_categorical

#########################
### Loading CIFAR-10 Dataset
#########################

(train_images, train_labels), (test_images, test_labels) = datasets.cifar10.load_data()

# Normalize pixel values to be between 0 and 1, reshape
train_images, test_images = [train_images / 255.0, test_images / 255.0]

train_images = np.array(random.sample(list(train_images),5000))
train_labels = np.array(random.sample(list(train_labels),5000))

test_images = np.array(random.sample(list(test_images),1000))
test_labels = np.array(random.sample(list(test_labels),1000))

#test_images = train_images[:10]
#test_labels = train_labels[:10]

#train_images = train_images[:100]
#train_labels = train_labels[:100]



class_names = ['airplane', 'automobile', 'bird', 'cat',
                'deer','dog', 'frog', 'horse',
                'ship', 'truck']


train_labels = to_categorical(train_labels, 10)
test_labels = to_categorical(test_labels, 10)
print(train_labels[0])

print(train_images.shape)
print(train_labels.shape)

print(test_images.shape)
print(test_labels.shape)

#########################
### Hpyer parameters
#########################

# Hyperparameters
training_epochs = 500
learning_rate = 0.1
batch_size = 100

img_h = img_w = 32              # CIFAR-10 images are 32x32
n_input = 32, 32, 3                   # 1024 x 1


n_hidden_1 = 64
n_hidden_2 = 32
n_classes = 10

#########################
### Neural Network DEFINITION
#########################

g = tf.Graph()
with g.as_default():
    # Input data
    tf_x = tf.placeholder(tf.float32, [None,32,32,3], name='features')
    # Input labels
    tf_y = tf.placeholder(tf.float32, [None,n_classes], name='targets')


    #figurd out need to flatten this for some reason other wise the logits/out_layer will be 1024 x 10 instead of batc
    flattened = tf.layers.flatten(tf_x, name='flatten')
    # The first hidden layer
    fc1 = tf.layers.dense(inputs=flattened, units=n_hidden_1, activation=tf.nn.sigmoid, name='fc1')
    # The second hidden layer
    fc2 = tf.layers.dense(inputs=fc1, units=n_hidden_2, activation=tf.nn.sigmoid, name='fc2')
# The third hidden layerqqqq
    out_layer = tf.layers.dense(inputs=fc2, units=n_classes, activation=None, name='out_layer')




    # Cost and optimizer
    loss = tf.nn.softmax_cross_entropy_with_logits(logits=out_layer, labels=tf_y)
    cost = tf.reduce_mean(loss, name='cost')
    # Training method is gradient descent
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
    train = optimizer.minimize(cost, name='train')

    # Prediction
    correct_prediction = tf.equal(tf.argmax(tf_y, 1), tf.argmax(out_layer, 1))
    # Prediction accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')
    # create saver object
    saver = tf.train.Saver()


#########################
### TRAINING & EVALUATION
#########################
cost_vector = np.zeros(training_epochs)
train_acc_vector = np.zeros(training_epochs)
test_acc_vector = np.zeros(training_epochs)
```

```python
    # Cost and optimizer
    loss = tf.nn.softmax_cross_entropy_with_logits(logits=out_layer, labels=tf_y)
    cost = tf.reduce_mean(loss, name='cost')
    # Training method is gradient descent
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
    train = optimizer.minimize(cost, name='train')

    # Prediction
    correct_prediction = tf.equal(tf.argmax(tf_y, 1), tf.argmax(out_layer, 1))
    # Prediction accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32), name='accuracy')
    # create saver object
    saver = tf.train.Saver()


#########################
### TRAINING & EVALUATION
#########################
cost_vector = np.zeros(training_epochs)
train_acc_vector = np.zeros(training_epochs)
test_acc_vector = np.zeros(training_epochs)




with tf.Session(graph=g) as sess:

    sess.run(tf.global_variables_initializer())

    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = train_images.shape[0] // batch_size
        print(total_batch)
        for i in range(total_batch):

            bx = train_images[i*batch_size:(i+1)*batch_size]
            by = train_labels[i*batch_size:(i+1)*batch_size]

            _, c = sess.run(['train', 'cost:0'], feed_dict={'features:0': bx, 'targets:0': by})
            avg_cost += c

        train_acc_vector[epoch] = sess.run('accuracy:0', feed_dict={'features:0': train_images,
                                                    'targets:0': train_labels})

        test_acc_vector[epoch] = sess.run('accuracy:0', feed_dict={'features:0': test_images,
                                                    'targets:0': test_labels})
        cost_vector[epoch] = avg_cost

        print("Epoch: %03d | AvgCost: %.3f Train/Test ACC: %.3f/%.3f" % (epoch + 1, avg_cost / (i + 1),train_acc_vec


#########################
### Visualizing The Result
#########################

Fontsize = 12
fig, _axs = plt.subplots(nrows=1, ncols=2)
axs = _axs.flatten()

l01, = axs[0].plot(range(training_epochs), cost_vector,'g')
axs[0].set_xlabel('Epoch',fontsize=Fontsize)
axs[0].set_ylabel('Cost',fontsize=Fontsize)
axs[0].grid(True)

l11, = axs[1].plot(range(training_epochs), 1-train_acc_vector,'b')
l12, = axs[1].plot(range(training_epochs), 1-test_acc_vector,'r')
axs[1].set_xlabel('Epoch',fontsize=Fontsize)
axs[1].set_ylabel('Error',fontsize=Fontsize)
axs[1].grid(True)
axs[1].legend(handles = [l11, l12], labels = ['train error', 'test error'],loc = 'upper right', fontsize=Fontsize)

plt.show()
```

I would also like to note that implementing the neural network using keras sequential model, was 10 times easier to implement and used less code than the code you supplied and resulted in even better training performance. I.e. model = models.Sequential(). I achieved upwards of 70% accuracy for both training and testing.