**Homework 2**

**COSC 6342: Machine Learning**

**Submitted by**

**S M Salah Uddin Kadir (1800503)**

**Rubayat Jinnah (1891217)**

The dataset contains the attribute values of Iris flowers. We will classify Iris flowers based on the length and width measurements of their sepals and petals. So, there are four features,

* Sepal length
* Sepal width
* Petal length
* Petal width

The categorized output is the corresponding species. There are three types of possible species:

* Iris setosa
* Iris virginica
* Iris versicolor

In our dataset, there are 110 examples that we have used to train our model, and 30 examples for the validation of the model. So,

* Train batch size = 110
* Test batch size = 30

We build our model using one hidden layer with 10 hidden units. We are categorizing dataset into 3 classes based on 4 attributes. So,

* Input layer size = 4
* Output layer size = 3
* Hidden layer size = 10

We can set different parameters for learning rate, number of iterations, and hidden layer size. We set different values to find the best learning rate for this model.

* Learning rate = 0.01
* Number of epochs = 500

We used Sigmoid as our activation unit which is a nonlinear activation function.

σ (WX) = 1 / (1 + e­-WX)

We draw different types of plot to understand the changes of the model over time. We used a parameter to set the frequency of drawing those plots.

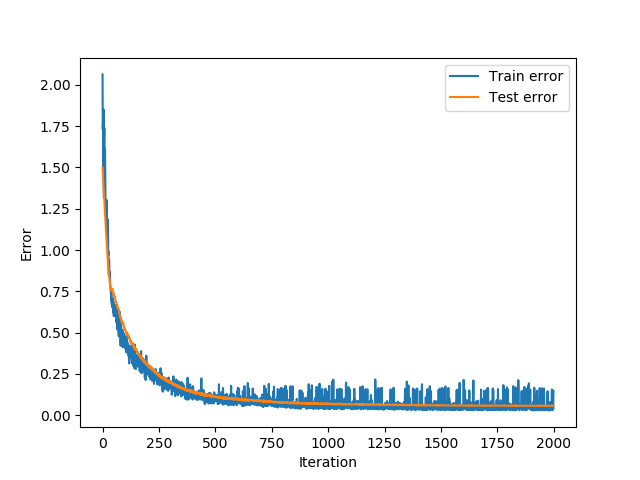
* Chart display frequency = 10

**Training and Test error changing rate along iterations:**

We had a plot to visualize the rate of changing errors along each iteration for the training and the test dataset.

***Configuration:***

* Learning rate = 0.001
* Number of epochs = 2000
* Activation unit = Relu
* Number of units in hidden layer = 10



From the figure, we can see that the training and testing errors have decreased over the number of iterations. From the graph we can conclude that the model did not overfit over the training dataset. We know there are two reason of overfitting,

* Random errors or noise
* Coincidental patterns

So, we can also conclude that the examples are also evenly distributed over the training and testing dataset, and there is no random noise or coincidental patterns.

**Histogram on activation values:**

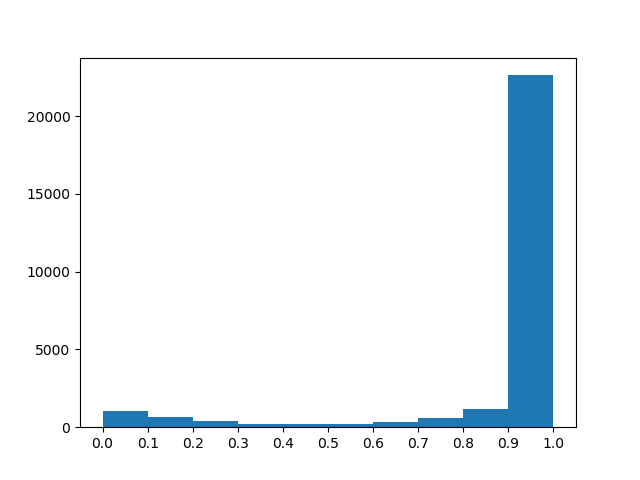
We had another type of plots to understand the distribution of ***all activation values*** along the iterations. We used Sigmoid as our activation unit.

***Configuration:***

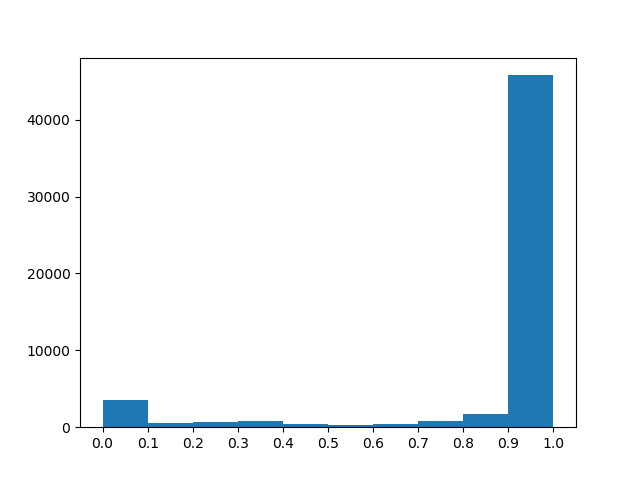
* Learning rate = 0.01
* Number of epochs = 2000
* Activation unit = Sigmoid
* Number of units in hidden layer = 10
* Chart display frequency = 25

We observed the distribution for the all activation values along iterations.

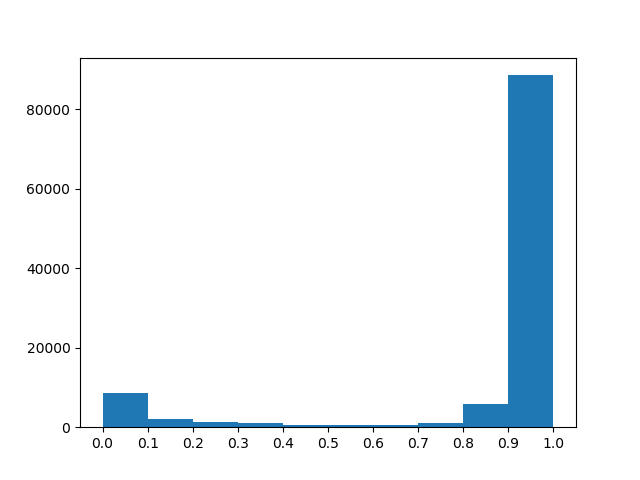
***After 25 iterations:***



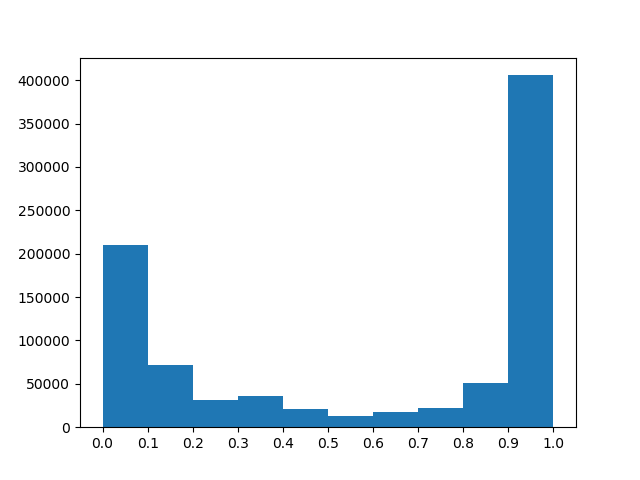
***After 50 iterations:***



***After 100 iterations:***



***Finally after 500 iterations:***



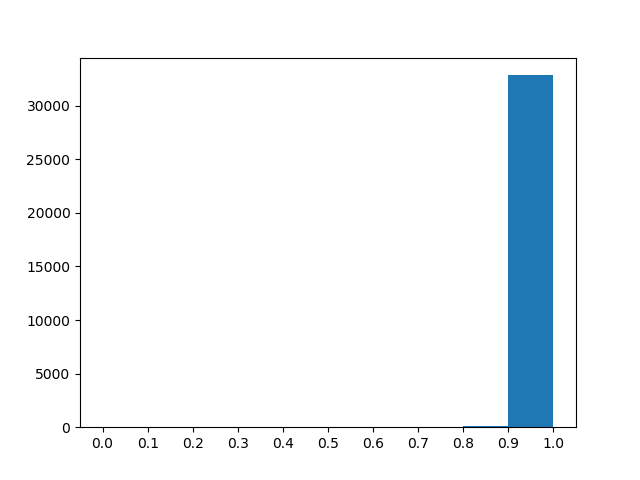
From the distribution of activation values along number of iterations, we can see that the number of lower activation values are increasing with the increase of number of iterations.

Now, Lets check the distribution of activation for ***different neurons in the same layer***.

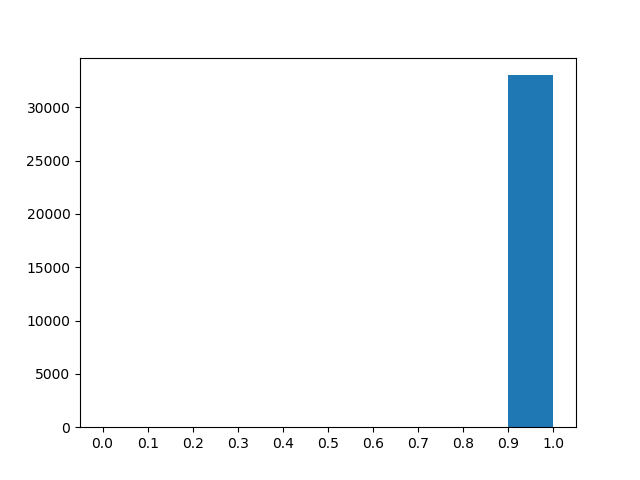
***Configuration:***

* Learning rate = 0.1
* Number of epochs = 300
* Activation unit = Sigmoid
* Number of units in hidden layer = 5

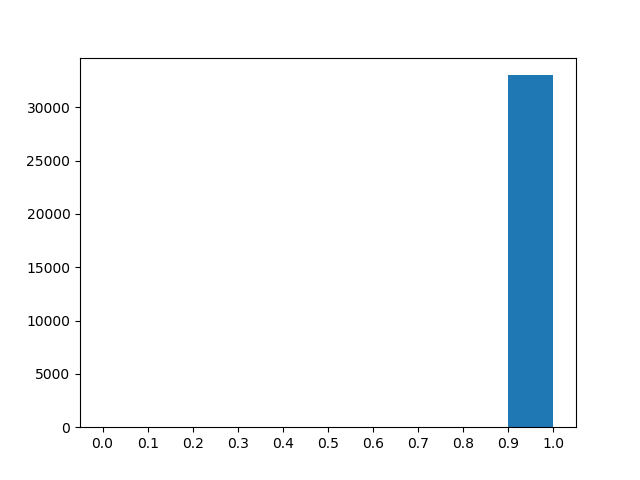
***Hidden layer neuron 01:***



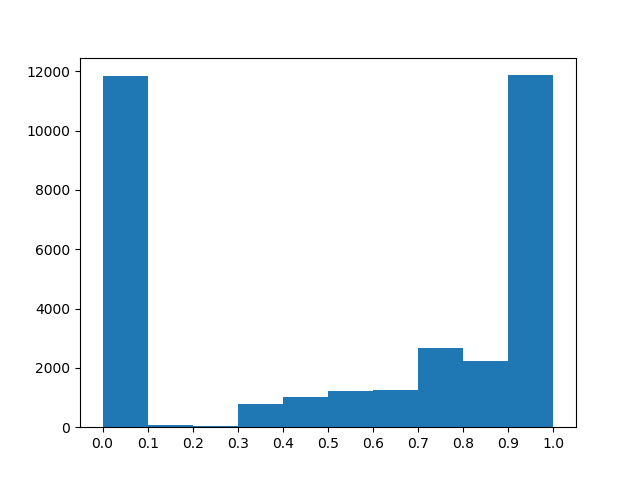
***Hidden layer neuron 02:***



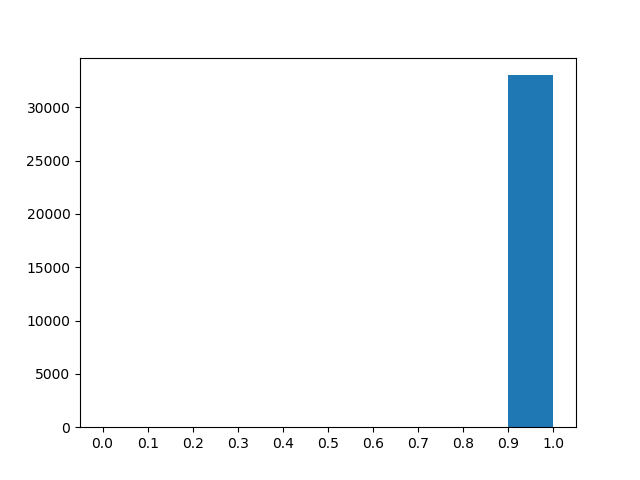
***Hidden layer neuron 03:***



***Hidden layer neuron 04:***



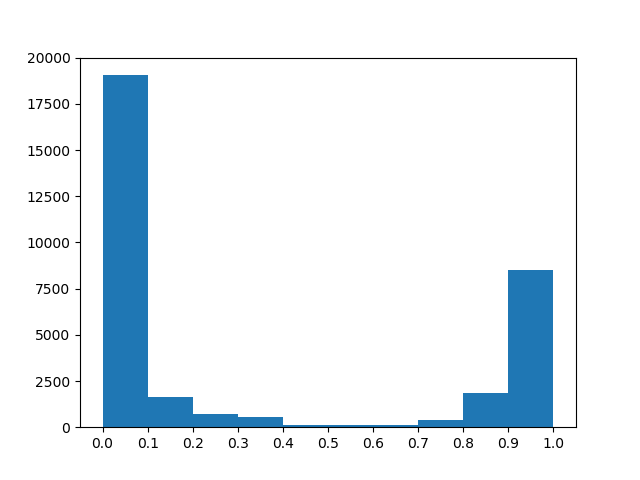
***Hidden layer neuron 05:***



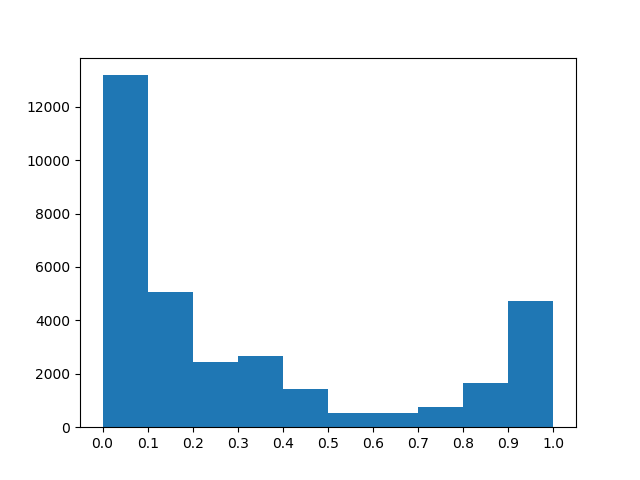
From the plots, we can see that Unit no 01, 02, 03, and 05 have almost similar distribution, but ***unit 04 has a different distribution***. So different neurons can have either similar, or different distributions.

Now, lets check the distribution of activation from ***different layer neurons***,

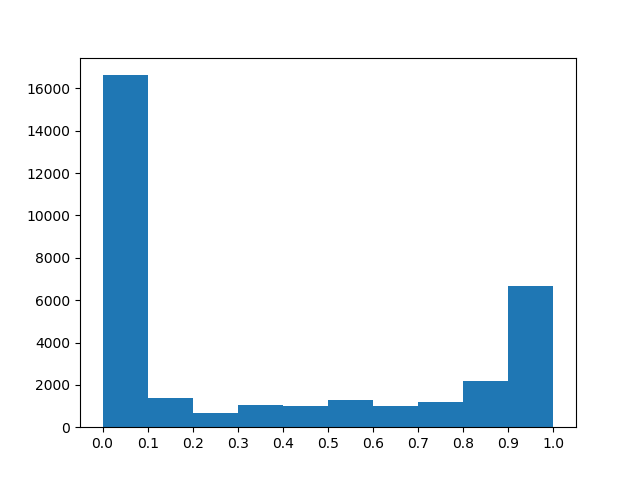
***Output layer neuron 01***:



***Output layer neuron 02***:



***Output layer neuron 03:***



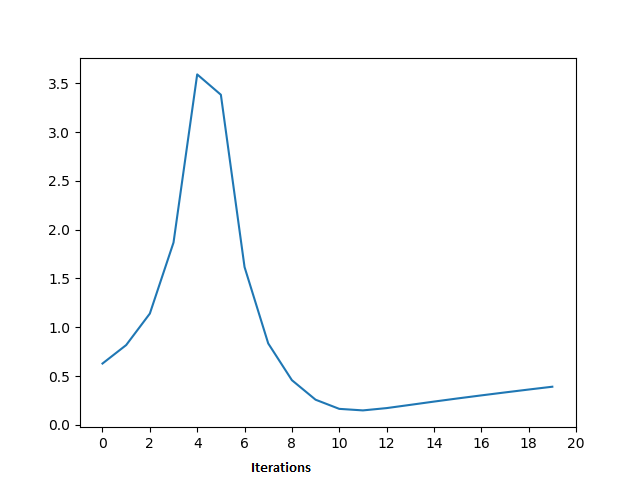
So, from all the distribution of activations, we can say that different neurons can have different distributions, although there is a high chance of having similar distribution among same layer neurons.

**Weight changes:**

We drew a plot to visualize the weight changes for hidden units along iterations.

**Configurations:**

* Learning rate = 0.1
* Number of epochs = 20
* Activation unit = Sigmoid
* Number of units in hidden layer = 5

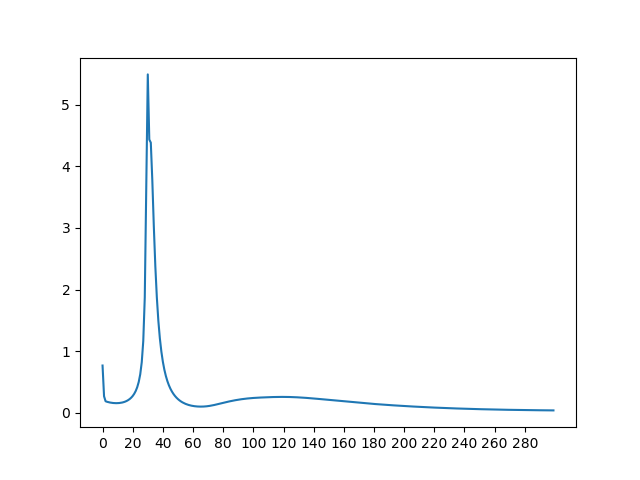


Now, we want to check all the weights from ***different layers become stable around the same time***.

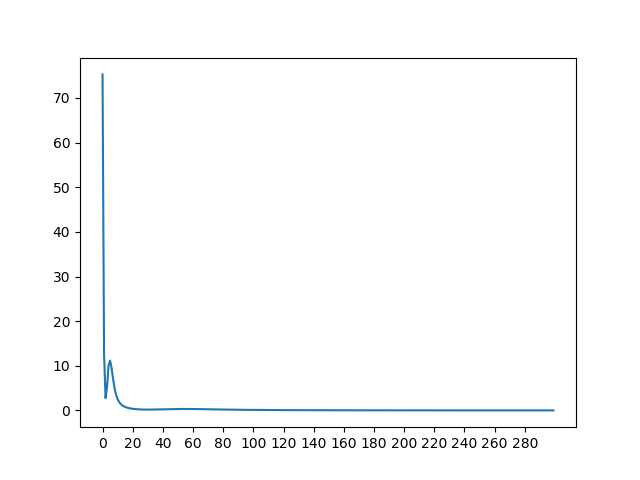
***Configurations:***

* Number of epochs = 300

***Weight changes for hidden layer:***



***Weight changes for output layer:***



From the two plots, we can see that output layer weights have been stable very fast than the hidden layer weights. So, we can conclude that different layers weights become stable in different time, and the same layer weights become stable around the same time.

It seems that the weight stability happens in reverse order that from output layer to input layer.

**Experiments with different parameters:**

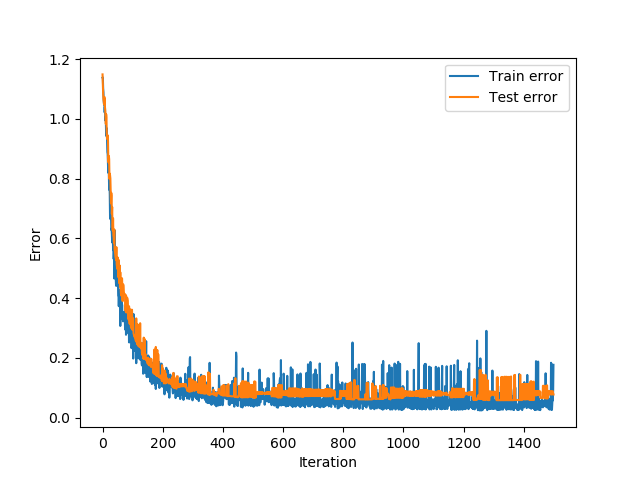
**Nonlinearity:**

We had an experiment on ***nonlinearity*** applying different parameters.

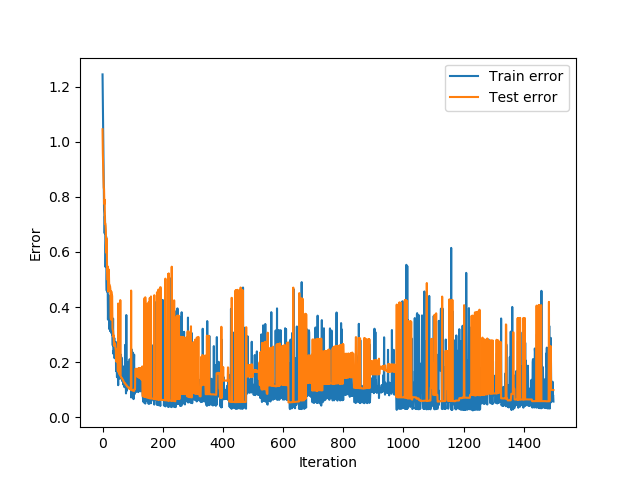
***Configuration:***

* Learning rate = 0.01
* Number of epochs = 1500
* Number of hidden layer = 1
* Number of units in hidden layer = 10

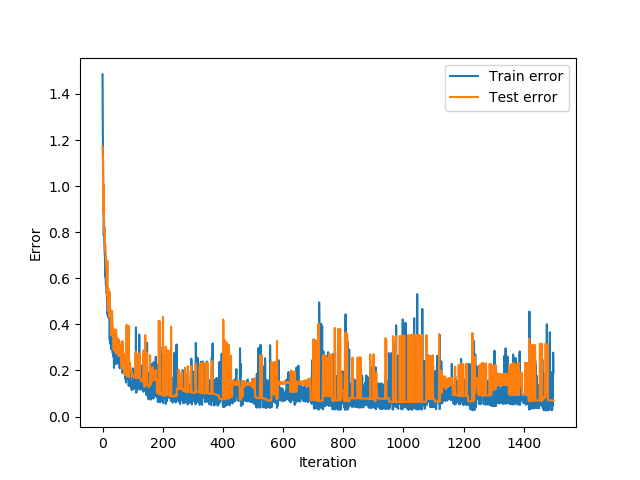
***Sigmoid:***



***Tanh:***



***Relu:***



From all the plots, we can see that Sigmoid function is working better for this configuration and the dataset. It seems that the convergence is oscillating for Relu and Tanh activation function when the learning rate 0.01.

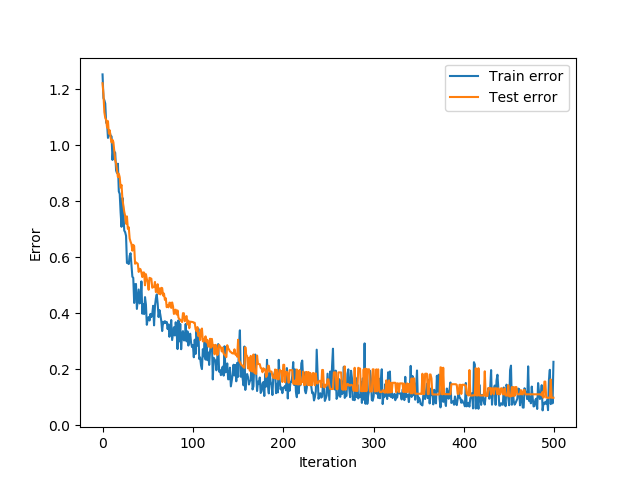
**Number of Neurons in Hidden Layers:**

We had an experiment by changing the number of neurons of hidden layers.

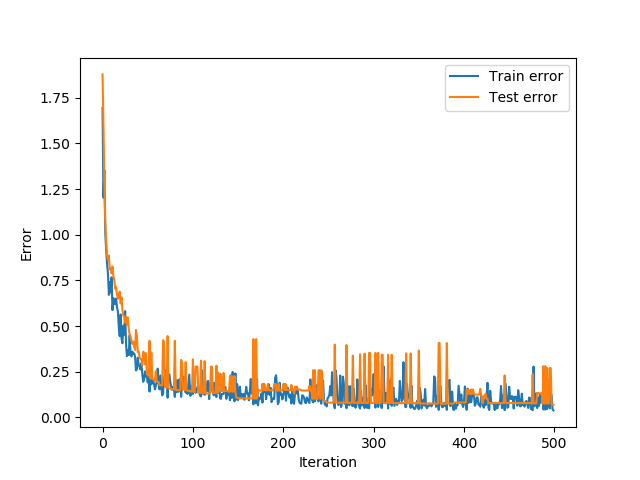
***Configuration:***

* Activation unit = relu
* Learning rate = 0.01
* Number of epochs = 500
* Number of hidden layer = 1

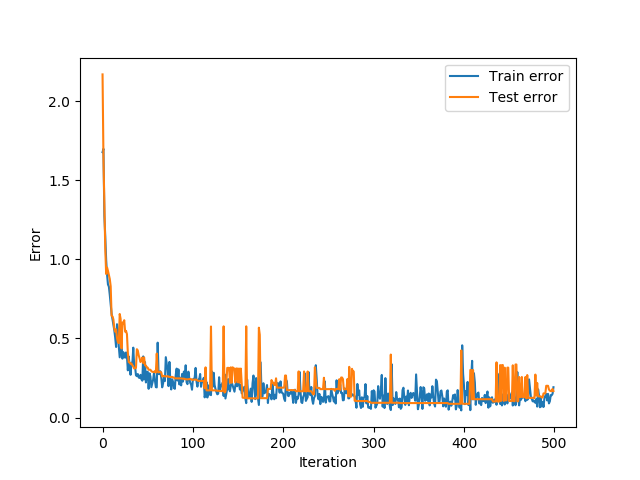
**Number of neurons: 05**



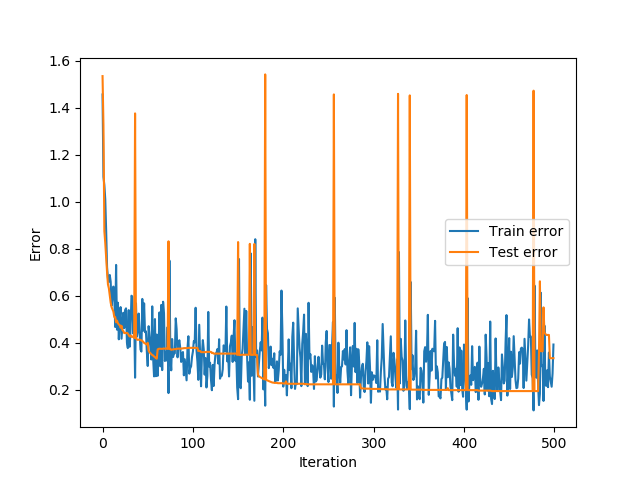
**Number of neurons: 10**



**Number of neurons: 20**



**Number of neurons: 40**



From the plots, we can see that the loss rate is lower for this dataset and the configuration when the number of neurons of hidden layer is 20.

We can also notice that the loss rate is also lower when the number of neurons for hidden layer is 5, but the convergence is slower.

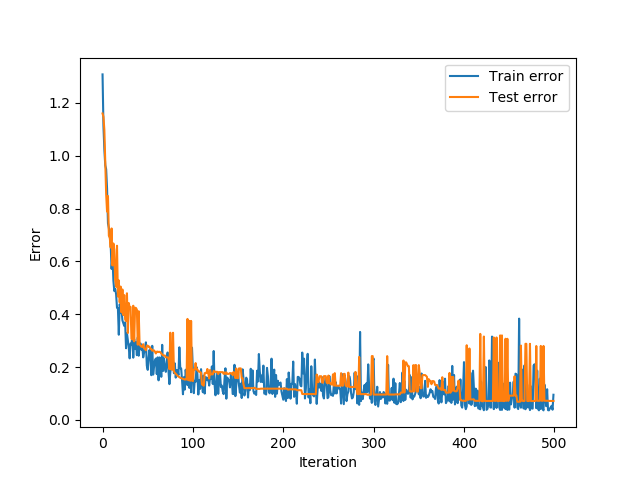
**Number of hidden layer:**

We had an experiment by adding more hidden layers.

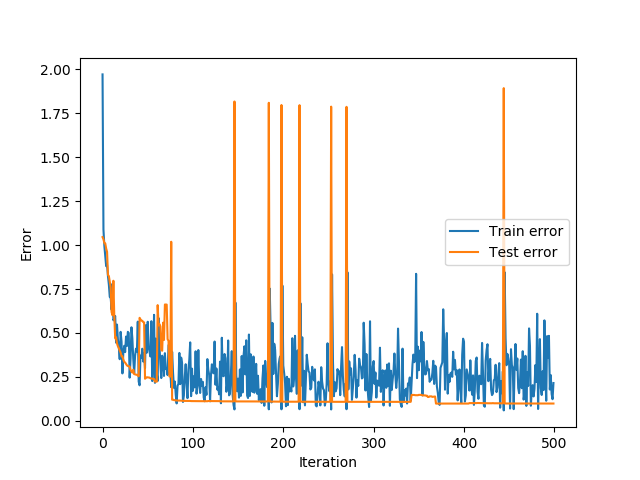
***Configuration:***

* Activation unit = relu
* Learning rate = 0.01
* Number of epochs = 500
* Number of neurons of each hidden layer = 10

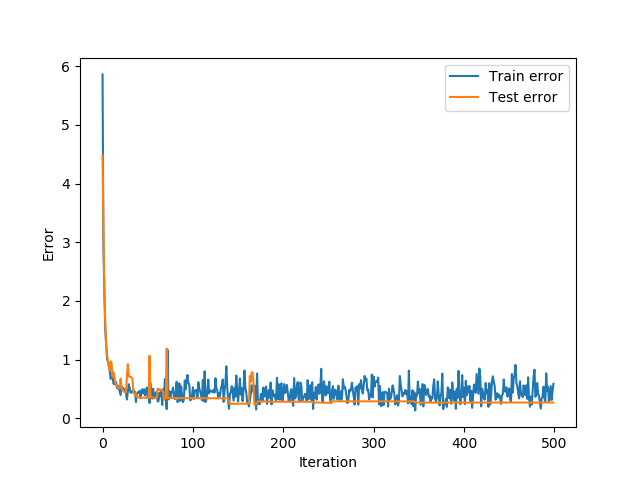
***Number of hidden layer:1***



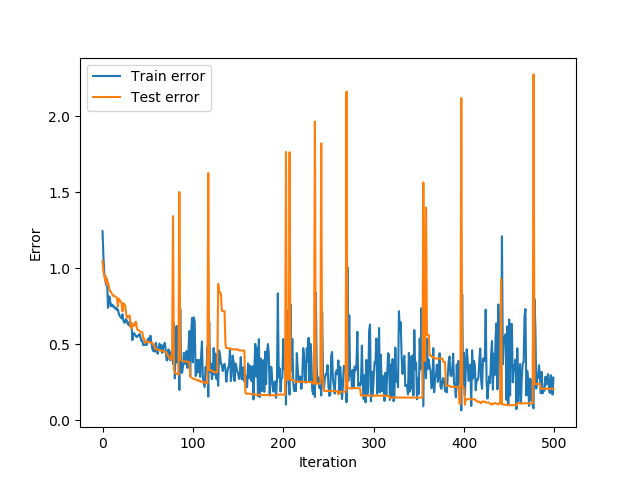
**Number of hidden layer:2**



**Number of hidden layer:3**



**Number of hidden layer: 04**



From the plots, we can see that our loss rate has been minimized for this dataset when there are three hidden layers. By increasing or decreasing the number of hidden layers from 3 for this model configuration the loss rate is higher.

***Program code:***

We had two implementations for this homework using the same dataset.

* Implementation using Tensorflow keras
* Implementation using only python applying formula

I modified the program files for different plots, and I removed that modification after generating the plot. If I would keep all the modification then the code would be very complex. So, I am adding here only the original base program.

***Implementation using Tensorflow keras***

**from** \_\_future\_\_ **import** absolute\_import, division, print\_function, unicode\_literals  
  
**import** os  
**import** matplotlib.pyplot **as** plt  
**import** tensorflow **as** tf  
**from** tensorflow.keras.models **import** load\_model  
  
  
**def** pack\_features\_vector(features, labels):  
 features = tf.stack(list(features.values()), axis=1)  
 **return** features, labels  
  
  
**def** get\_dataset(url, batch\_size, column\_names, label\_names):  
 train\_dataset\_fp = tf.keras.utils.get\_file(fname=os.path.basename(url), origin=url)  
 **print**(**"Local copy of the dataset file: {}"**.format(train\_dataset\_fp))  
  
 train\_dataset = tf.data.experimental.make\_csv\_dataset(  
 train\_dataset\_fp,  
 batch\_size,  
 column\_names=column\_names,  
 label\_name=label\_names,  
 num\_epochs=1)  
  
 **return** train\_dataset

**def** create\_the\_model():  
 model = tf.keras.Sequential([  
 tf.keras.layers.Dense(first\_hidden\_layer\_size, activation=activation\_unit, input\_shape=(input\_layer\_size,)),  
 tf.keras.layers.Dense(second\_hidden\_layer\_size, activation=activation\_unit),  
 tf.keras.layers.Dense(output\_layer\_size)  
 ])  
  
 **return** model  
  
  
**def** gradient\_function(model, input\_features, true\_output):  
 **with** tf.GradientTape() **as** tape:  
 predicted\_output = model(input\_features)  
 loss\_object = tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True)  
 loss\_value = loss\_object(y\_true=true\_output, y\_pred=predicted\_output)  
  
 gradients = tape.gradient(loss\_value, model.trainable\_variables)  
 **return** loss\_value, gradients  
  
  
**def** train\_the\_model(model, train\_dataset):  
 train\_loss\_results = []  
 train\_accuracy\_results = []  
  
 **for** epoch **in** range(num\_epochs):  
 epoch\_loss\_avg = tf.keras.metrics.Mean()  
 epoch\_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()  
  
 **for** features, label **in** train\_dataset:  
 **print**(features)  
 loss\_value, gradients = gradient\_function(model, features, label) optimizer = tf.keras.optimizers.Adam(learning\_rate=learn\_rate)  
 optimizer.apply\_gradients(zip(gradients, model.trainable\_variables))  
  
epoch\_loss\_avg(loss\_value) *# Add current batch loss* epoch\_accuracy(label, model(features)) *# Compare predicted label to actual label* train\_loss\_results.append(epoch\_loss\_avg.result())  
 train\_accuracy\_results.append(epoch\_accuracy.result())  
  
 **if** epoch % 25 == 0:  
 **print**(**"Epoch {:03d}: Loss: {:.3f}, Accuracy: {:.3%}"**.format(epoch, epoch\_loss\_avg.result(), epoch\_accuracy.result()))  
  
 **return** model, train\_loss\_results, train\_accuracy\_results  
  
  
**def** plot\_dataset(dataset):  
 features, labels = next(iter(dataset))  
 **print**(features)  
  
 plt.scatter(features[**'petal\_length'**],  
 features[**'sepal\_length'**],  
 c=labels,  
 cmap=**'viridis'**)  
  
 plt.xlabel(**"Petal length"**)  
 plt.ylabel(**"Sepal length"**)  
 plt.show()  
  
  
**def** visualize\_loss\_function\_over\_time(train\_loss\_results, train\_accuracy\_results):  
 fig, axes = plt.subplots(2, sharex=True, figsize=(12, 8))  
 fig.suptitle(**'Training Metrics'**)  
  
 axes[0].set\_ylabel(**"Loss"**, fontsize=14)  
 axes[0].plot(train\_loss\_results)  
  
 axes[1].set\_ylabel(**"Accuracy"**, fontsize=14)  
 axes[1].set\_xlabel(**"Epoch"**, fontsize=14)  
 axes[1].plot(train\_accuracy\_results)  
 plt.show()  
  
**def** evaluate\_test\_data(model, test\_dataset):  
 test\_accuracy = tf.keras.metrics.Accuracy()  
  
 **for** (features, label) **in** test\_dataset:  
 logits = model(features)  
 prediction = tf.argmax(logits, axis=1, output\_type=tf.int32)  
 test\_accuracy(prediction, label)  
  
 **print**(**"Test set accuracy: {:.3%}"**.format(test\_accuracy.result()))  
  
  
**def** build\_model():  
 train\_dataset = get\_dataset(train\_dataset\_url, train\_batch\_size, column\_names, label\_name)  
*# plot\_dataset(train\_dataset)* train\_dataset = train\_dataset.map(pack\_features\_vector)  
 model = create\_the\_model()  
 model, train\_loss\_results, train\_accuracy\_results = train\_the\_model(model, train\_dataset)  
 visualize\_loss\_function\_over\_time(train\_loss\_results, train\_accuracy\_results)  
  
 model.save(model\_name)  
 **return** model  
  
  
**def** test\_the\_model(model\_name):  
 model = load\_model(model\_name, compile=False)  
  
 test\_dataset = get\_dataset(test\_dataset\_url, test\_batch\_size, column\_names, label\_name)  
*# plot\_dataset(test\_dataset)* test\_dataset = test\_dataset.map(pack\_features\_vector)  
 evaluate\_test\_data(model, test\_dataset)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 train\_dataset\_url = **"https://storage.googleapis.com/download.tensorflow.org/data/iris\_training.csv"** test\_dataset\_url = **"https://storage.googleapis.com/download.tensorflow.org/data/iris\_test.csv"** activation\_unit = tf.nn.relu  
 learn\_rate = 0.01  
 num\_epochs = 200  
 train\_batch\_size = 110  
 test\_batch\_size = 30  
 model\_name = **'hw2\_trained\_model.h5'** class\_names = [**'Iris setosa'**, **'Iris versicolor'**, **'Iris virginica'**]  
 column\_names = [**'sepal\_length'**, **'sepal\_width'**, **'petal\_length'**, **'petal\_width'**, **'species'**]  
 feature\_names = column\_names[:-1]  
 label\_name = column\_names[-1]  
  
 input\_layer\_size = len(feature\_names)  
 first\_hidden\_layer\_size = 10  
 second\_hidden\_layer\_size = 10  
 output\_layer\_size = len(class\_names)  
  
 model = build\_model()

*test\_the\_model(model\_name)*

**Implementation using only python applying formula  
  
from** \_\_future\_\_ **import** absolute\_import, division, print\_function, unicode\_literals  
  
**import** os  
**import** matplotlib.pyplot **as** plt  
**import** tensorflow **as** tf  
**import** numpy **as** np  
**import** math  
  
  
**def** get\_dataset(url, batch\_size, column\_names, label\_names):  
 train\_dataset\_fp = tf.keras.utils.get\_file(fname=os.path.basename(url), origin=url)  
 **print**(**"Local copy of the dataset file: {}"**.format(train\_dataset\_fp))  
  
 train\_dataset = tf.data.experimental.make\_csv\_dataset(  
 train\_dataset\_fp,  
 batch\_size,  
 column\_names=column\_names,  
 label\_name=label\_names,  
 num\_epochs=1)  
  
 **return** train\_dataset *#returns (features, label) pairs; where features is a dictionary: {'feature\_name': value}***def** pack\_features\_vector(features, labels):  
 features = tf.stack(list(features.values()), axis=1)  
 **return** features, labels  
  
  
**def** sigmoid(wx):  
 **return** (1/(1+ math.exp(wx \* -1)))  
  
  
*# Propagating Forward: Hidden Layer. σ ( WX )***def** hidden\_layer\_node\_update(feature, first\_hidden\_layer, all\_activation\_value):  
 **for** i **in** range(hidden\_layer\_size):  
 wx = hidden\_layer\_weights[i] \* feature  
 sigma\_wx = sigmoid(sum(wx))  
 first\_hidden\_layer.append(sigma\_wx)  
 all\_activation\_value.append(sigma\_wx)  
  
 **return** first\_hidden\_layer  
  
  
*# Propagating Forward: Output Layer σ ( WX )***def** output\_layer\_node\_update(first\_hidden\_layer, output\_layer):  
 **for** i **in** range(output\_layer\_size):  
 wx = output\_layer\_weights[i] \* first\_hidden\_layer  
 sigma\_wx = sigmoid(sum(wx))  
 output\_layer.append(sigma\_wx)  
  
  
*# Backpropagation: Output layer delta δk = Ok (1-Ok)(tk – Ok)***def** compute\_output\_layer\_delta(output\_layer, true\_label, output\_delta):  
 output\_bool = [0] \* output\_layer\_size  
 output\_bool[true\_label.numpy()] = 1  
  
 **for** i **in** range(output\_layer\_size):  
 delta = output\_layer[i] \* (1 - output\_layer[i]) \* (output\_bool[i] - output\_layer[i])  
 output\_delta.append(delta)  
  
  
*# Backpropagation: Weight of hidden layer to output layer; Wji = Wji + ΔWji  
# ΔWji = η δj Xji***def** update\_output\_layer\_weight(first\_hidden\_layer, output\_delta):  
 **for** i **in** range(hidden\_layer\_size):  
 **for** j **in** range(output\_layer\_size):  
 delta\_w = learn\_rate \* first\_hidden\_layer[i] \* output\_delta[j]  
 output\_layer\_weights[j][i] = output\_layer\_weights[j][i] + delta\_w  
  
  
*# Backpropagation: Hidden layer delta, δh = Oh(1-Oh) Σk Wkh δk***def** compute\_hidden\_layer\_delta(first\_hidden\_layer, output\_delta, hidden\_delta):  
 **for** i **in** range(hidden\_layer\_size):  
 sum\_delta = 0  
 **for** j **in** range(output\_layer\_size):  
 sum\_delta = sum\_delta + output\_layer\_weights[j][i] \* output\_delta[j]  
 delta = first\_hidden\_layer[i] \* (1 - first\_hidden\_layer[i]) \* sum\_delta  
 hidden\_delta.append(delta)  
  
  
*# Backpropagation: Weight of input layer to hidden layer; Wji = Wji + ΔWji  
# ΔWji = η δj Xji***def** update\_hidden\_layer\_weight(input\_layer, hidden\_delta):  
 **for** i **in** range(input\_layer\_size):  
 **for** j **in** range(hidden\_layer\_size):  
 delta\_w = learn\_rate \* input\_layer[i] \* hidden\_delta[j]  
 hidden\_layer\_weights[j][i] = hidden\_layer\_weights[j][i] + delta\_w  
  
  
**def** error\_chart(train\_lost\_result, test\_loss\_result):  
 plt.plot(train\_lost\_result, label=**"Train error"**)  
 plt.plot(test\_loss\_result, label=**"Test error"**)  
  
 plt.xlabel(**"Iteration"**)  
 plt.ylabel(**"Error"**)  
 plt.legend()  
 plt.show()  
  
  
**def** histogram\_with\_activation\_value(all\_activation\_value):  
 **print** (len(all\_activation\_value))  
 bins = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]  
 plt.hist(all\_activation\_value, bins, histtype=**'bar'**, linewidth=0)  
 plt.xticks(np.arange(0, 1.01, step=0.1))  
 plt.ylabel(**"Number of activation values"**)  
 plt.show()  
  
  
**def** draw\_weight\_changes\_plot(angle\_changes):  
 plt.plot(angle\_changes)  
 plt.xticks(np.arange(0, 21, step=2))  
 plt.show()  
  
  
*# Using least square method***def** calculate\_error(predicted\_value, true\_value):  
 **return** ((predicted\_value[true\_value.numpy()] - 1) \* (predicted\_value[true\_value.numpy()] - 1))  
  
  
**def** get\_test\_dataset\_error(test\_dataset):  
 **for** features, labels **in** test\_dataset:  
 error = 0  
 all\_test\_activation\_value = []  
 **for** i **in** range(test\_batch\_size):  
 input\_layer = features[i]  
 label = labels[i]  
  
 hidden\_layer = []  
 output\_layer = []  
 hidden\_layer\_node\_update(input\_layer, hidden\_layer, all\_test\_activation\_value)  
 output\_layer\_node\_update(hidden\_layer, output\_layer)  
  
 error = error + calculate\_error(output\_layer, label)

**return** error/test\_batch\_size  
  
  
*# radian = cos-1((a1b1 + a2b2)/||a|| \* ||b||)***def** weight\_change\_angle(previous\_weights, updated\_weights):  
 changed\_weights = previous\_weights \* updated\_weights  
 a = 0  
 b = 0  
 total\_changed\_weights = 0  
 **for** i **in** range(hidden\_layer\_size):  
 **for** j **in** range(input\_layer\_size):  
 total\_changed\_weights = total\_changed\_weights + changed\_weights[i][j]  
 a = a + (previous\_weights[i][j] \* previous\_weights[i][j])  
 b = b + (updated\_weights[i][j] \* updated\_weights[i][j])  
 a = math.sqrt(a)  
 b = math.sqrt(b)  
  
 radian = math.acos(total\_changed\_weights/(a\*b))  
 degree = radian \* (180/math.pi)  
 **return** degree  
  
  
**def** train\_the\_model(train\_dataset, test\_dataset):  
 **for** features, labels **in** train\_dataset:  
 train\_errors = []  
 test\_errors = []  
 all\_activation\_value = []  
 angle\_changes = []  
  
 **for** epoch **in** range(num\_epochs):  
 error = 0  
 previous\_weights = hidden\_layer\_weights.copy()  
  
 **for** i **in** range(train\_batch\_size):  
 input\_layer = features[i]  
 label = labels[i]  
  
 hidden\_layer = []  
 output\_layer = []  
 hidden\_layer\_node\_update(input\_layer, hidden\_layer, all\_activation\_value)  
 output\_layer\_node\_update(hidden\_layer, output\_layer)  
  
 error = error + calculate\_error(output\_layer, label) output\_delta = []  
 compute\_output\_layer\_delta(output\_layer, label, output\_delta)  
 update\_output\_layer\_weight(hidden\_layer, output\_delta)  
  
 hidden\_delta = []  
 compute\_hidden\_layer\_delta(hidden\_layer, output\_delta, hidden\_delta)  
 update\_hidden\_layer\_weight(features[i], hidden\_delta)  
  
 train\_error = error/train\_batch\_size  
 train\_errors.append(train\_error)  
 test\_error = get\_test\_dataset\_error(test\_dataset)  
 test\_errors.append(test\_error)  
 **print** (**"epoch: "**, epoch, **"; train error: "**, train\_error, **"; test error: "**, test\_error)  
  
 updated\_weights = hidden\_layer\_weights.copy()  
 angle = weight\_change\_angle(previous\_weights, updated\_weights)  
 angle\_changes.append(angle)  
  
 **if** ((epoch+1) % chart\_display\_frequency) == 0:  
 error\_chart(train\_errors, test\_errors)  
 histogram\_with\_activation\_value(all\_activation\_value)  
 draw\_weight\_changes\_plot(angle\_changes)  
  
 **break  
  
  
def** build\_model():  
 train\_dataset = get\_dataset(train\_dataset\_url, train\_batch\_size, column\_names, label\_name)  
 train\_dataset = train\_dataset.map(pack\_features\_vector)  
  
 test\_dataset = get\_dataset(test\_dataset\_url, test\_batch\_size, column\_names, label\_name)  
 test\_dataset = test\_dataset.map(pack\_features\_vector)  
  
 train\_the\_model(train\_dataset, test\_dataset)  
  
  
**def** predict():  
 test\_dataset = get\_dataset(test\_dataset\_url, test\_batch\_size, column\_names, label\_name)  
 test\_dataset = test\_dataset.map(pack\_features\_vector)  
  
 j=0  
 **for** (features, label) **in** test\_dataset:  
 **for** i **in** range(test\_batch\_size):  
 first\_hidden\_layer = []  
 output\_layer = []  
 hidden\_layer\_node\_update(features[i], first\_hidden\_layer)  
 output\_layer\_node\_update(first\_hidden\_layer, output\_layer)  
  
 max\_value = max(output\_layer)  
 max\_index = output\_layer.index(max\_value)  
  
 **print** (label[i])  
 **print**(output\_layer)  
  
 **if**(label[i].numpy() == max\_index):  
 j = j+ 1  
  
 **break  
 print** (**"Prediction accuracy: "**, j)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 train\_dataset\_url = **"https://storage.googleapis.com/download.tensorflow.org/data/iris\_training.csv"** test\_dataset\_url = **"https://storage.googleapis.com/download.tensorflow.org/data/iris\_test.csv"**learn\_rate = 0.1  
 num\_epochs = 500  
 train\_batch\_size = 110  
 test\_batch\_size = 30  
 chart\_display\_frequency = 10  
  
 class\_names = [**'Iris setosa'**, **'Iris versicolor'**, **'Iris virginica'**]  
 column\_names = [**'sepal\_length'**, **'sepal\_width'**, **'petal\_length'**, **'petal\_width'**, **'species'**]  
 feature\_names = column\_names[:-1]  
 label\_name = column\_names[-1]  
  
 input\_layer\_size = len(feature\_names)  
 hidden\_layer\_size = 10  
 output\_layer\_size = len(class\_names)  
  
 hidden\_layer\_weights = np.random.random((hidden\_layer\_size, input\_layer\_size))  
 output\_layer\_weights = np.random.random((output\_layer\_size, hidden\_layer\_size))  
  
 build\_model()  
  
*# predict()*