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.

$$\sigma (WX) = 1 / (1 + e^{-WX})$$

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

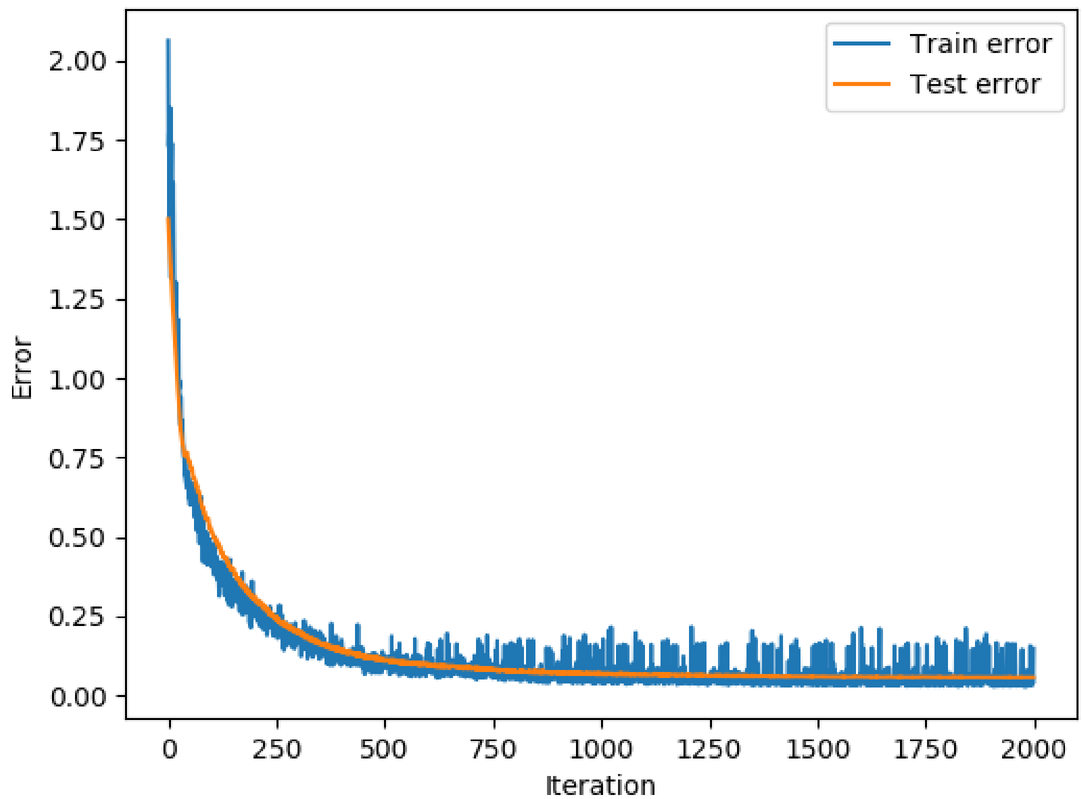
Training and Test error changing rate along iterations:

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.

<u>Histogram on activation values:</u>

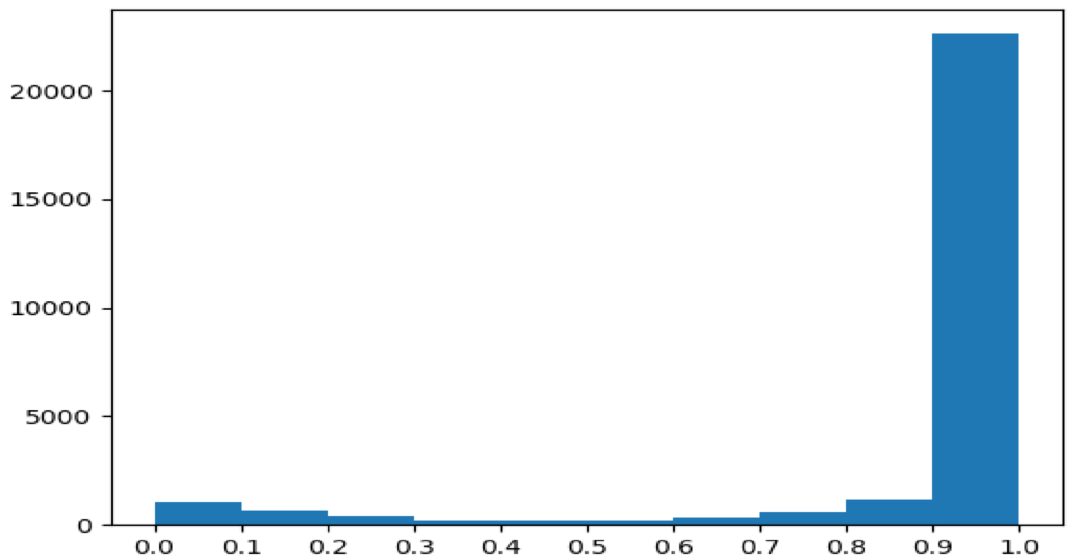
We had another type of plots to understand the distribution of ***all activation values*** along the iterations. We used Sigmoid as our activation unit.

Configuration:

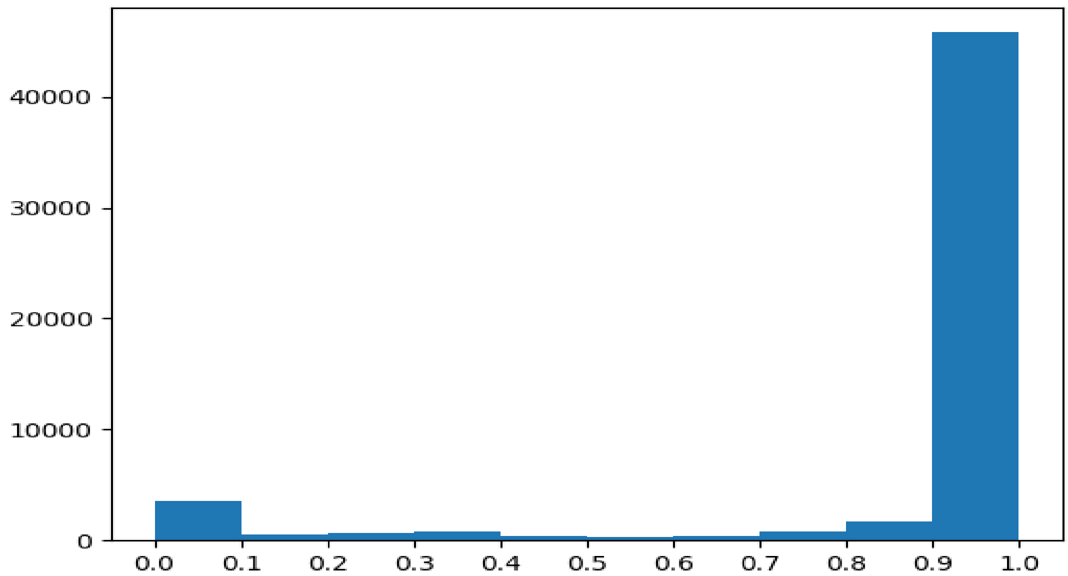
- Learning rate = 0.01
- Number of epochs = 2000
- Activation unit = Sigmoid
- Number of units in hidden layer = 10
- Chart display frequency = 25

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

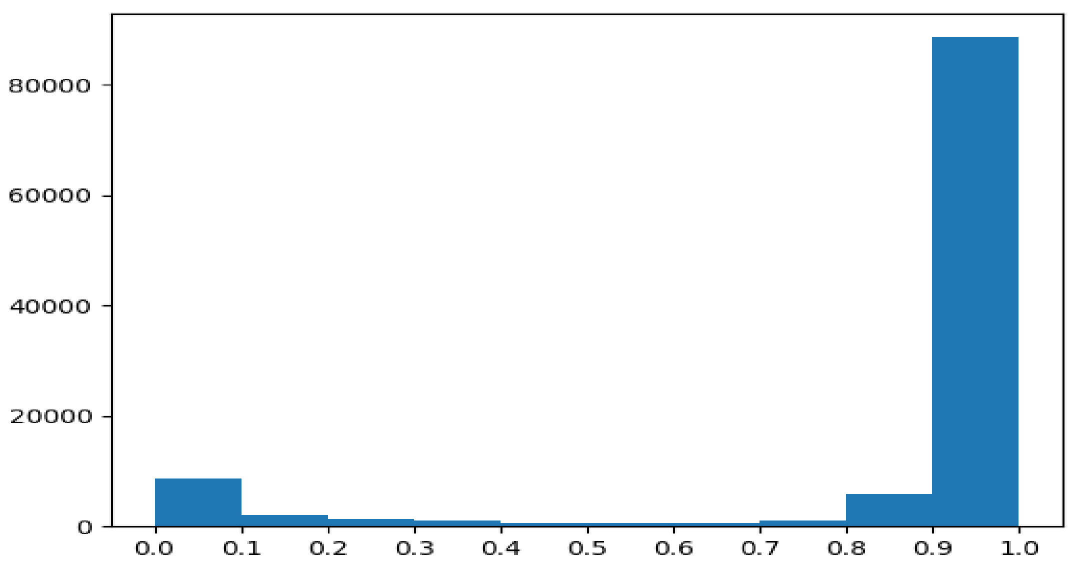
After 25 iterations:



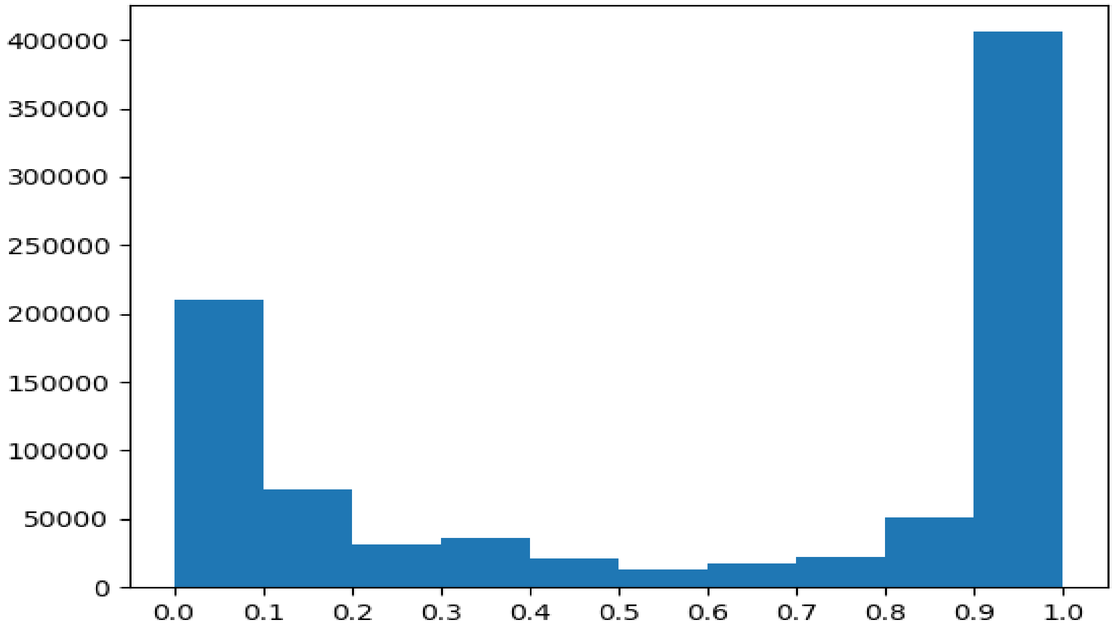
After 50 iterations:



After 100 iterations:



Finally after 500 iterations:



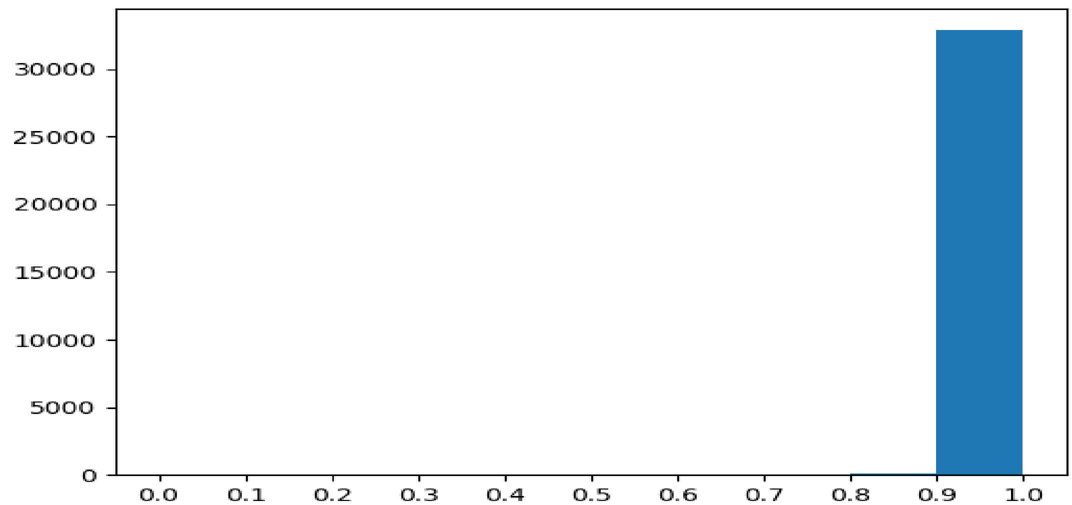
From the distribution of activation values along number of iterations, we can see that the number of lower activation values are increasing with the increase of number of iterations.

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

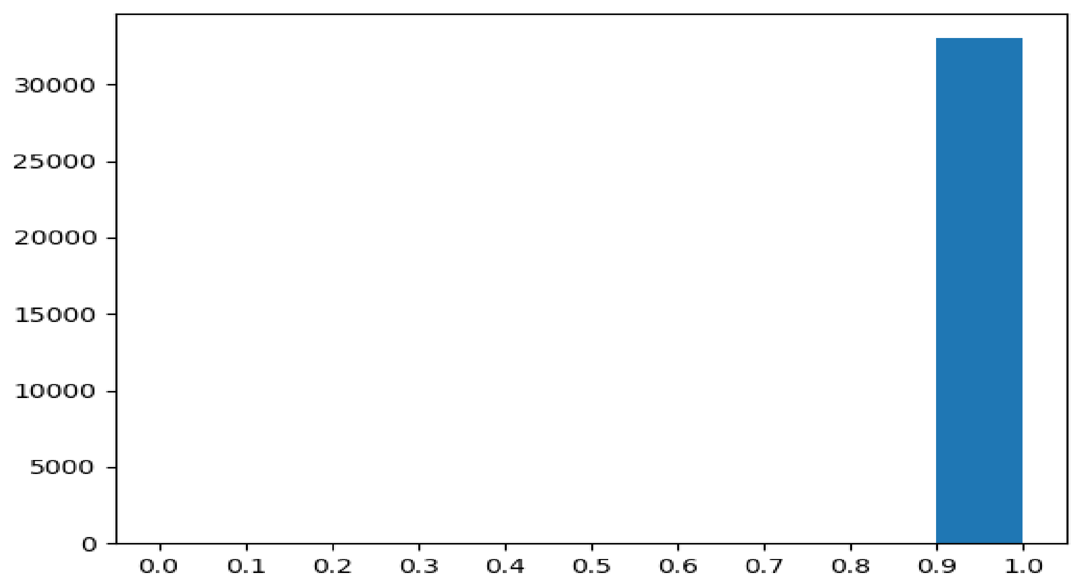
Configuration:

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

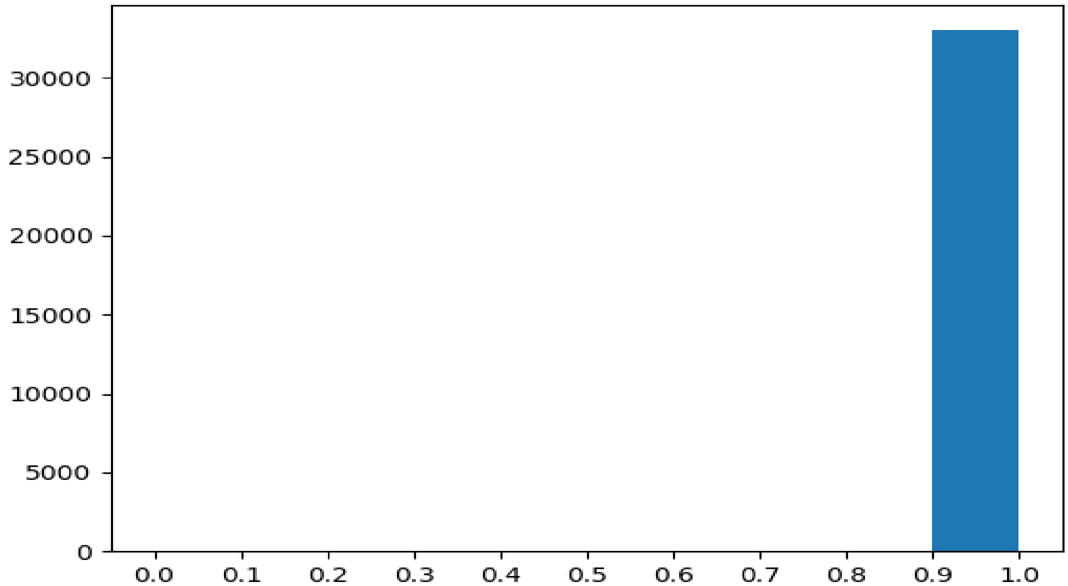
Hidden layer neuron 01:



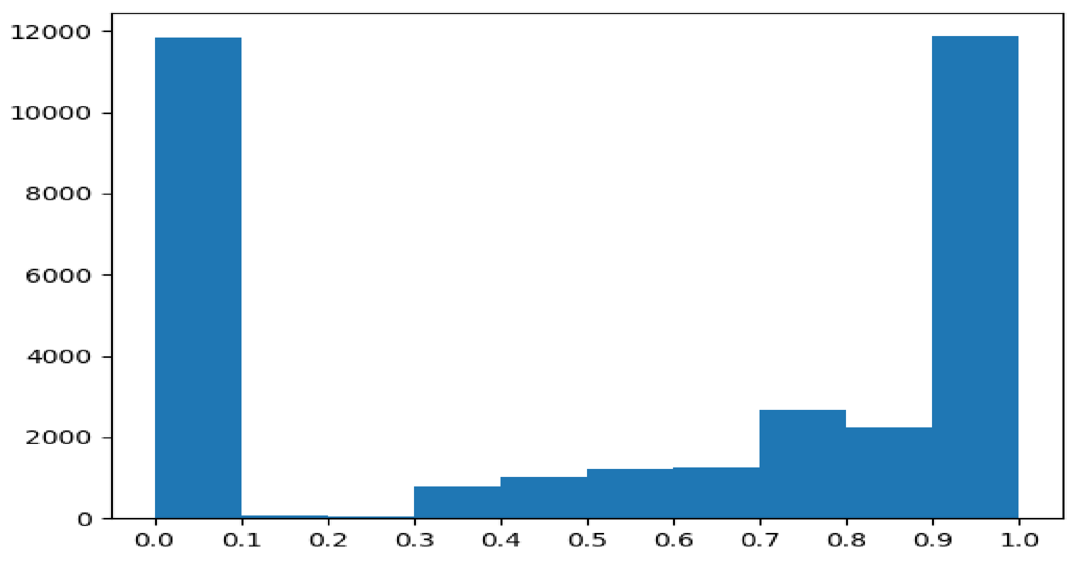
Hidden layer neuron 02:



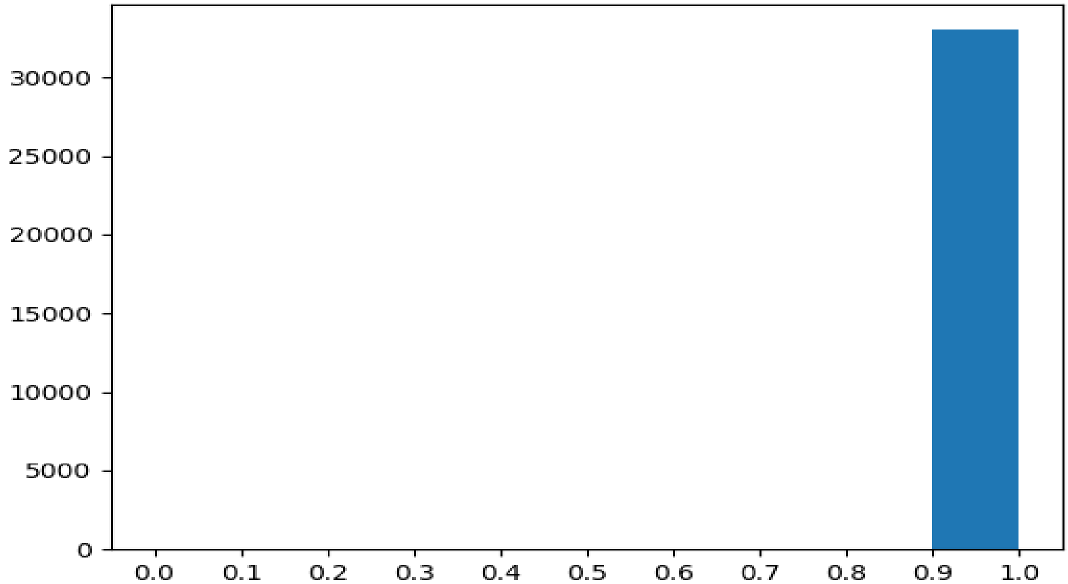
Hidden layer neuron 03:



Hidden layer neuron 04:



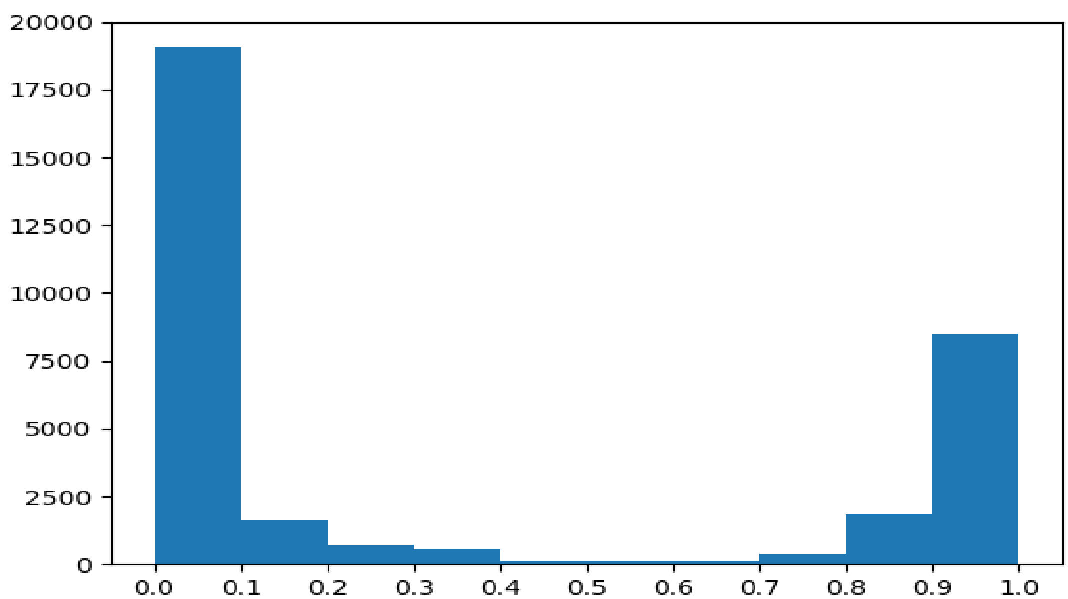
Hidden layer neuron 05:



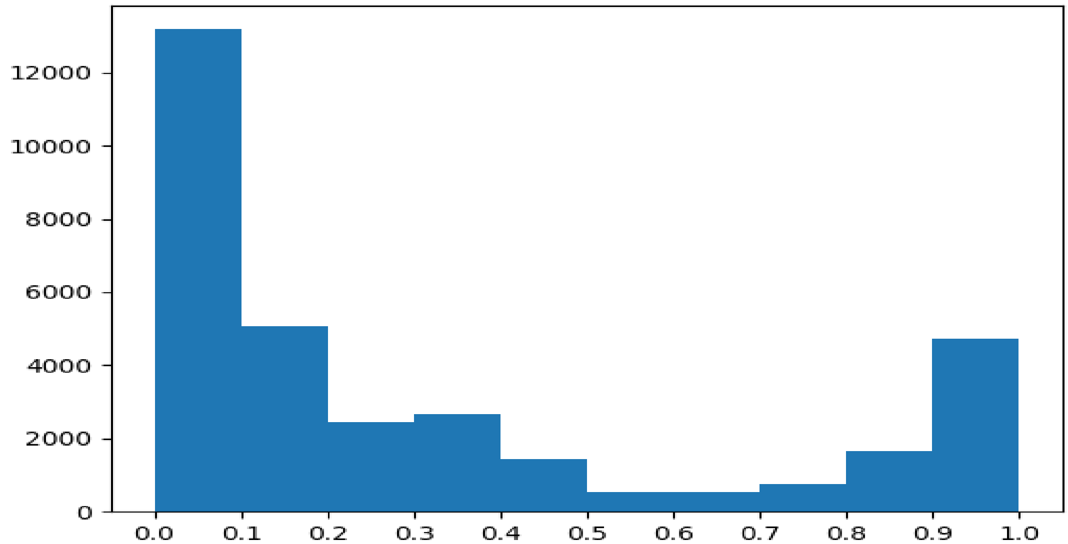
From the plots, we can see that Unit no 01, 02, 03, and 05 have almost similar distribution, but ***unit 04 has a different distribution***. So different neurons can have either similar, or different distributions.

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

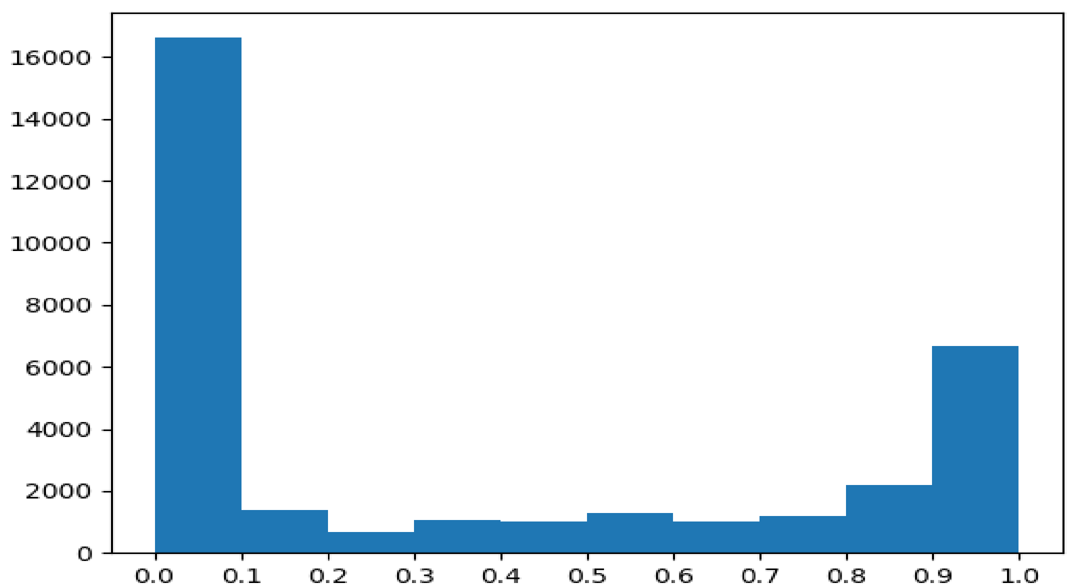
Output layer neuron 01:



Output layer neuron 02:



Output layer neuron 03:



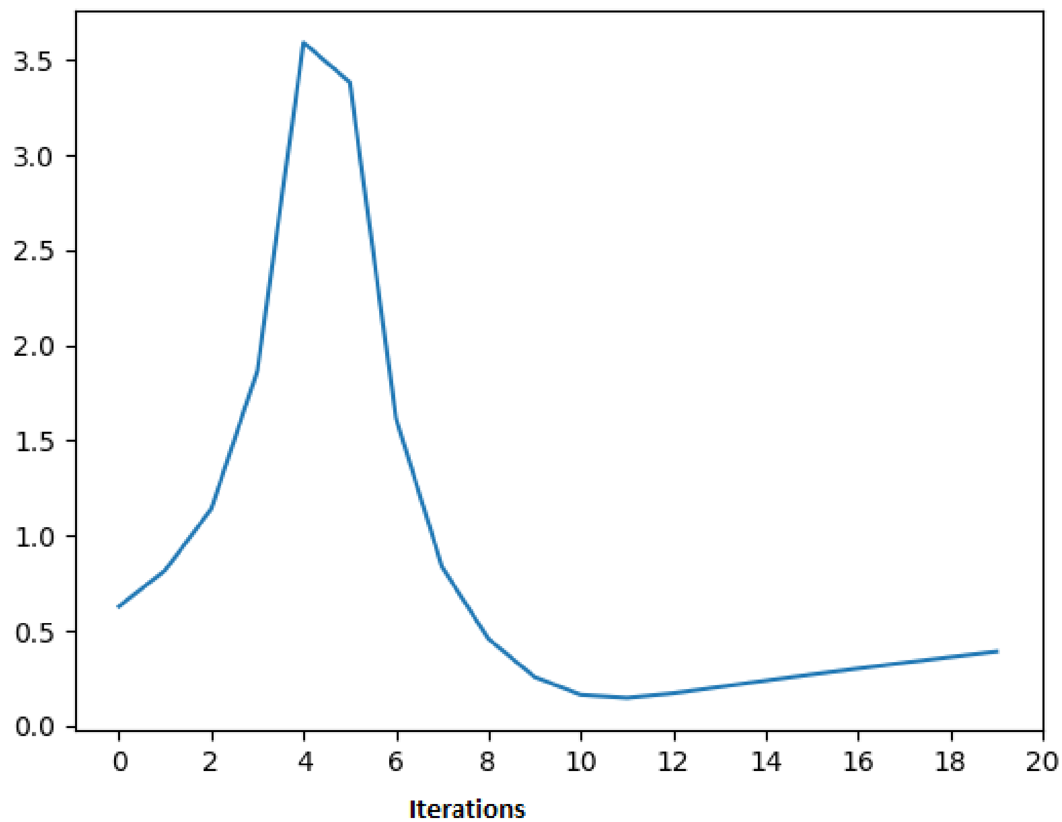
So, from all the distribution of activations, we can say that different neurons can have different distributions, although there is a high chance of having similar distribution among same layer neurons.

<u>Weight changes:</u>

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

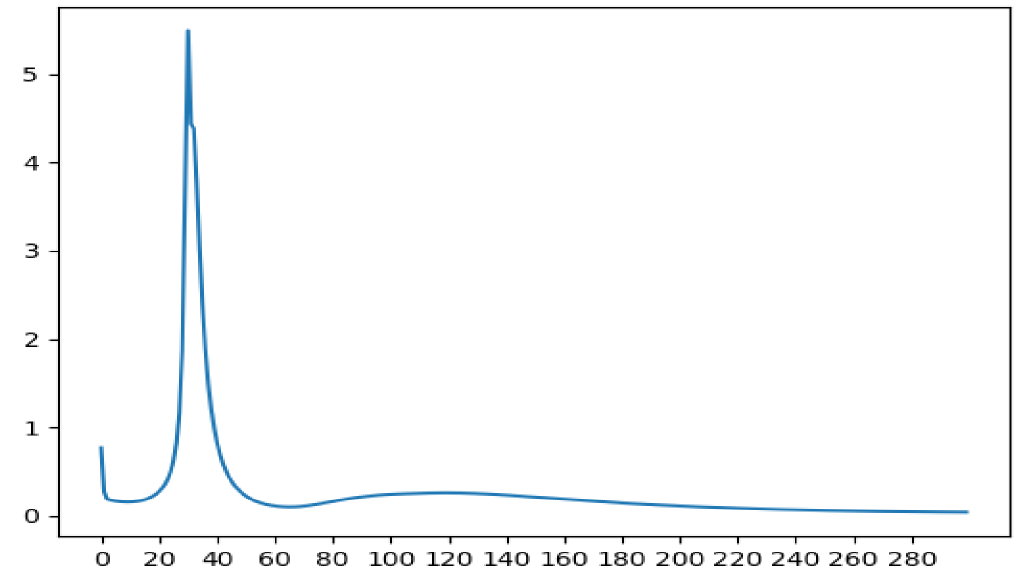


Now, we want to check all the weights from ***different layers become stable around the same time***.

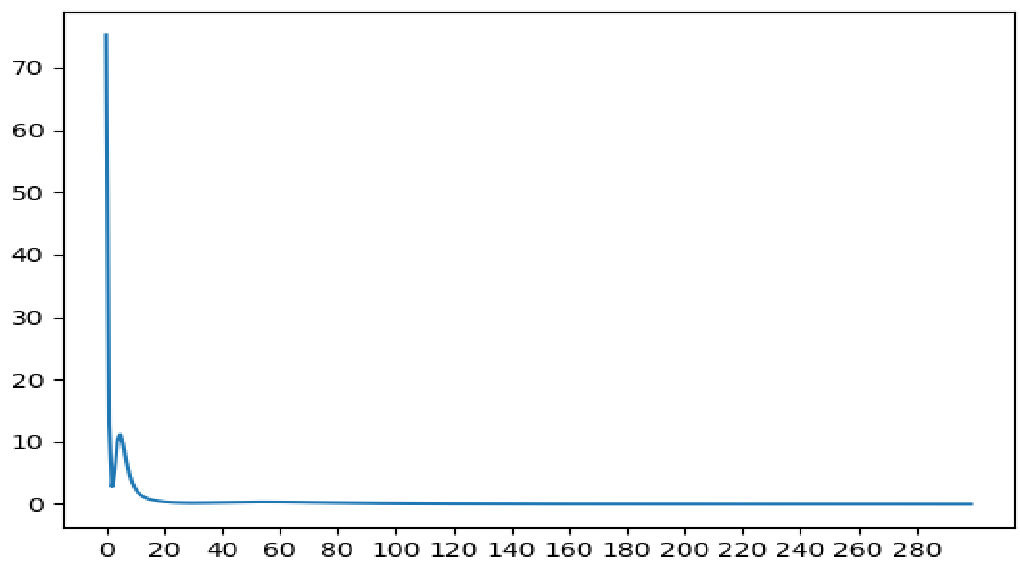
Configurations:

- 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.

<u>Experiments with different parameters:</u>

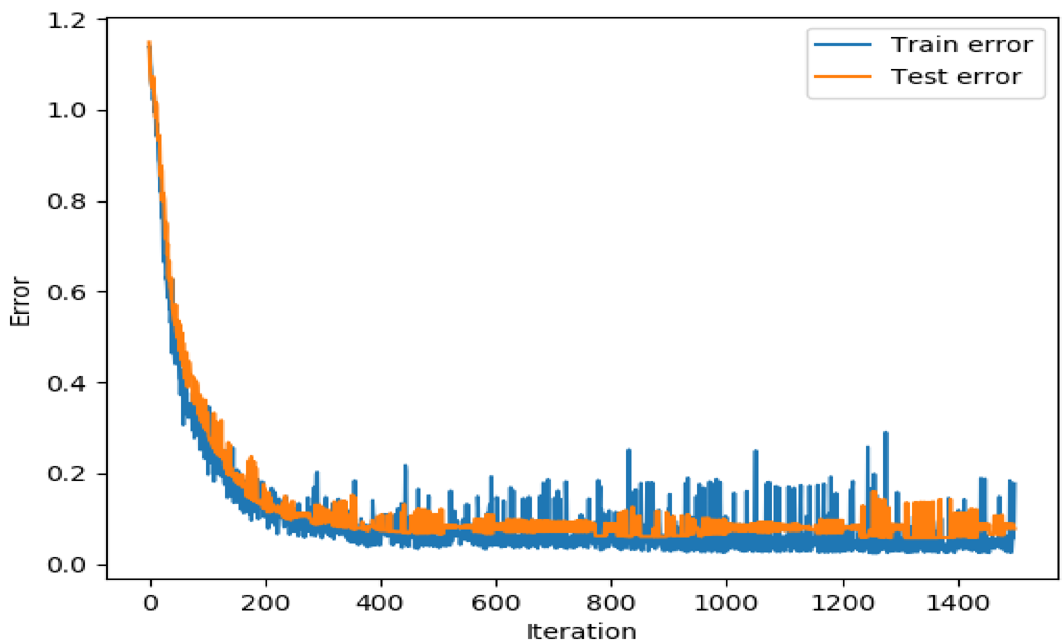
Nonlinearity:

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

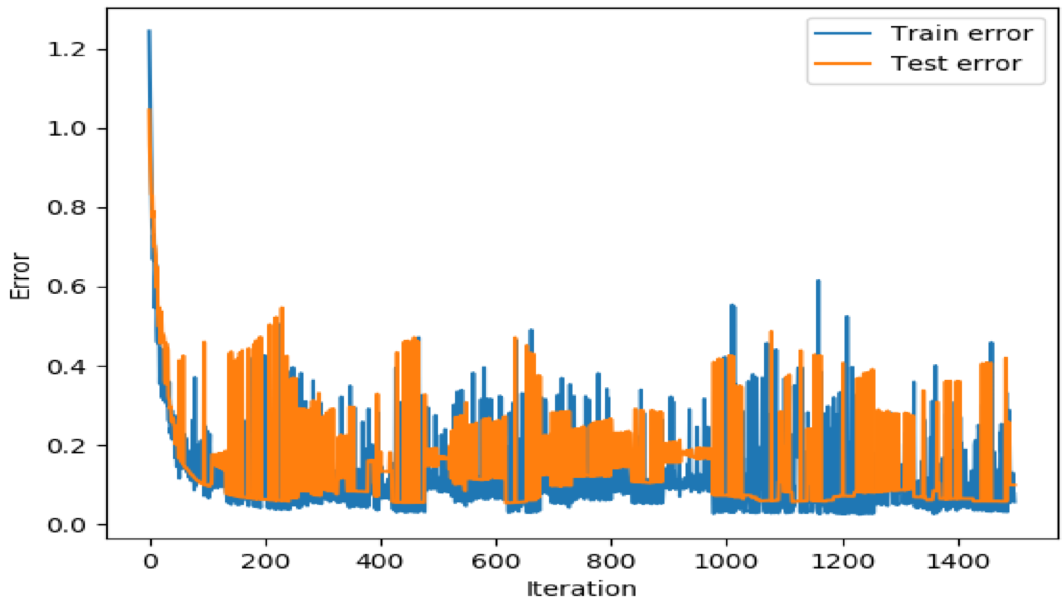
Configuration:

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

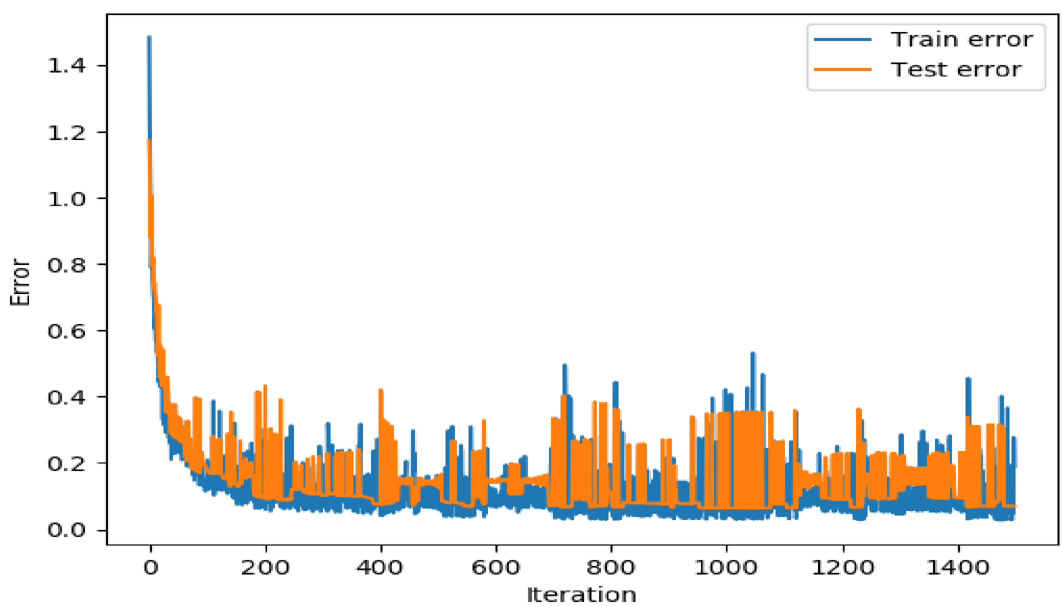
Sigmoid:



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.

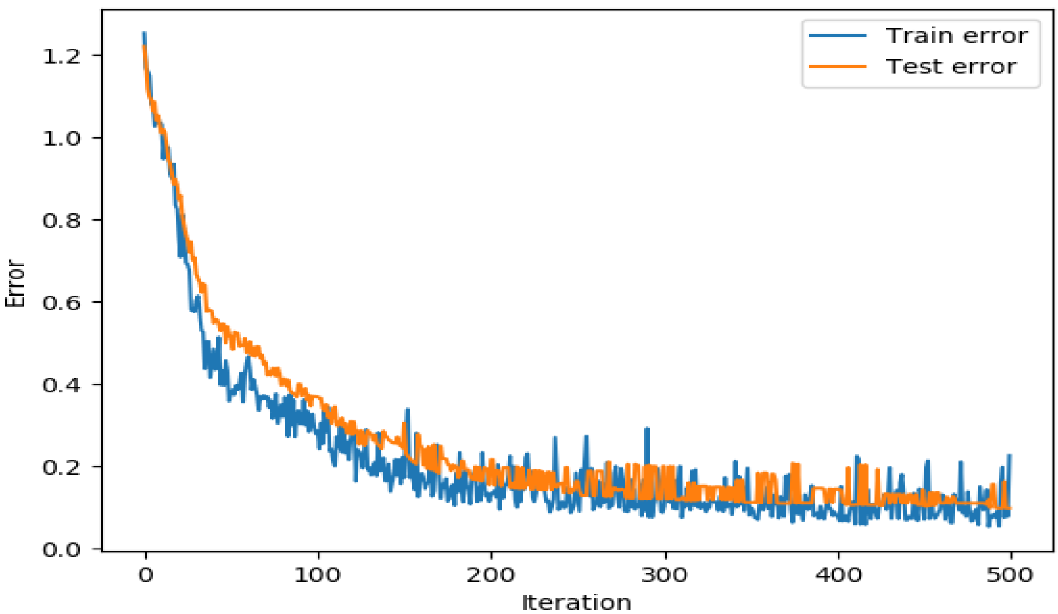
Number of Neurons in Hidden Layers:

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

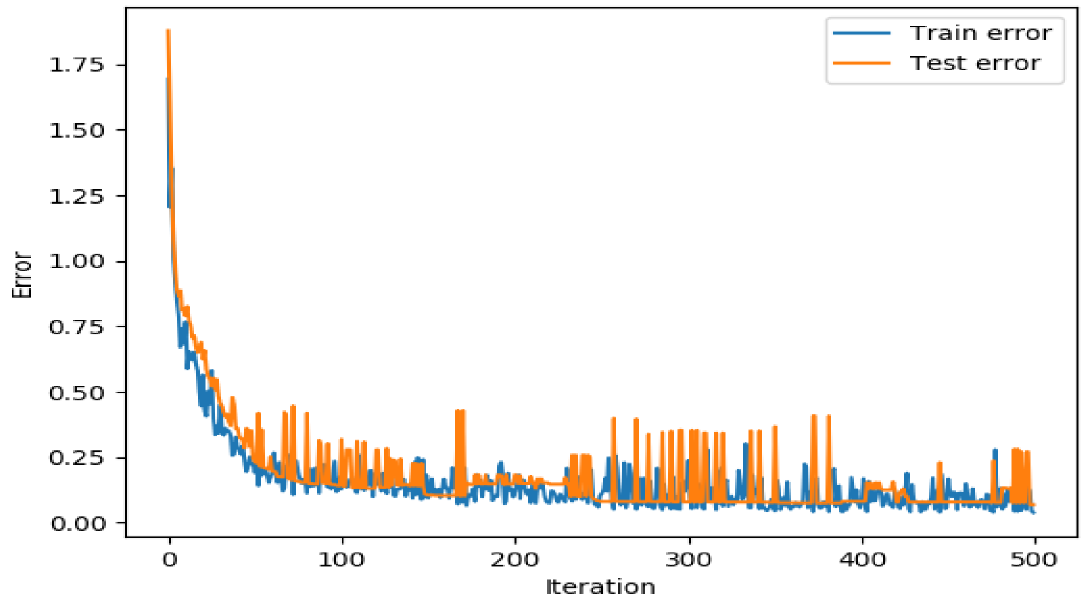
Configuration:

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

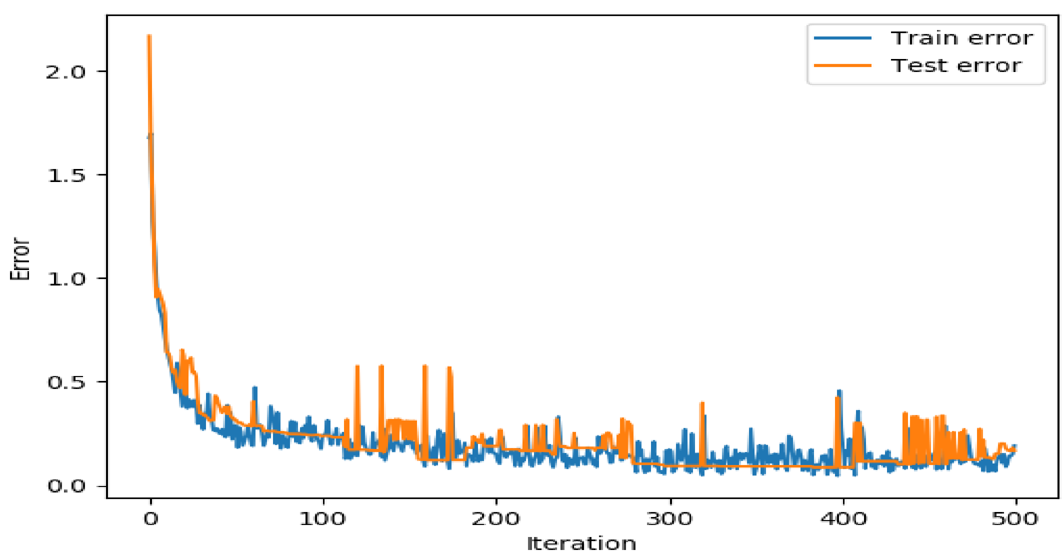
Number of neurons: 05



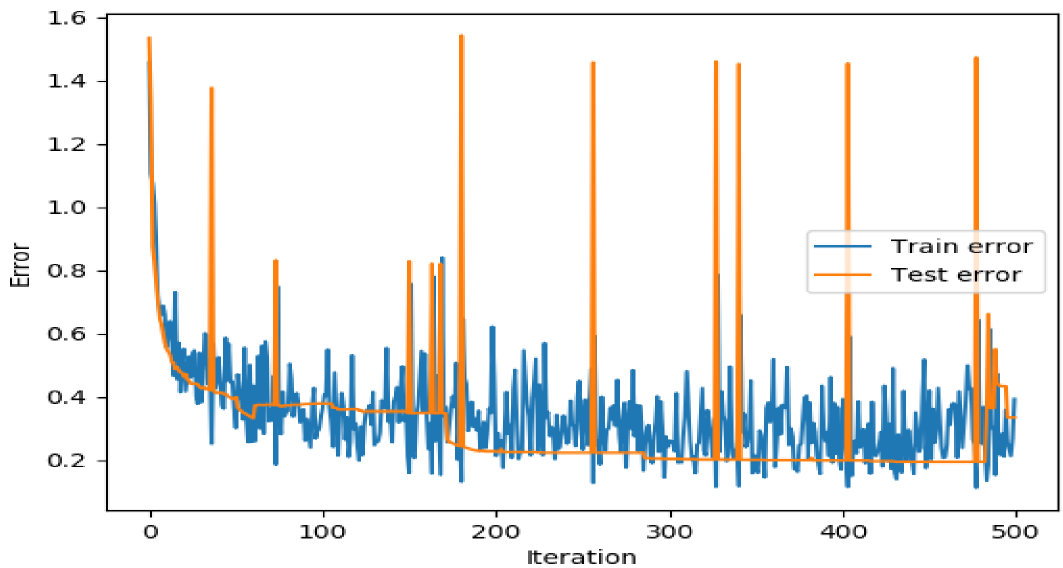
Number of neurons: 10



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.

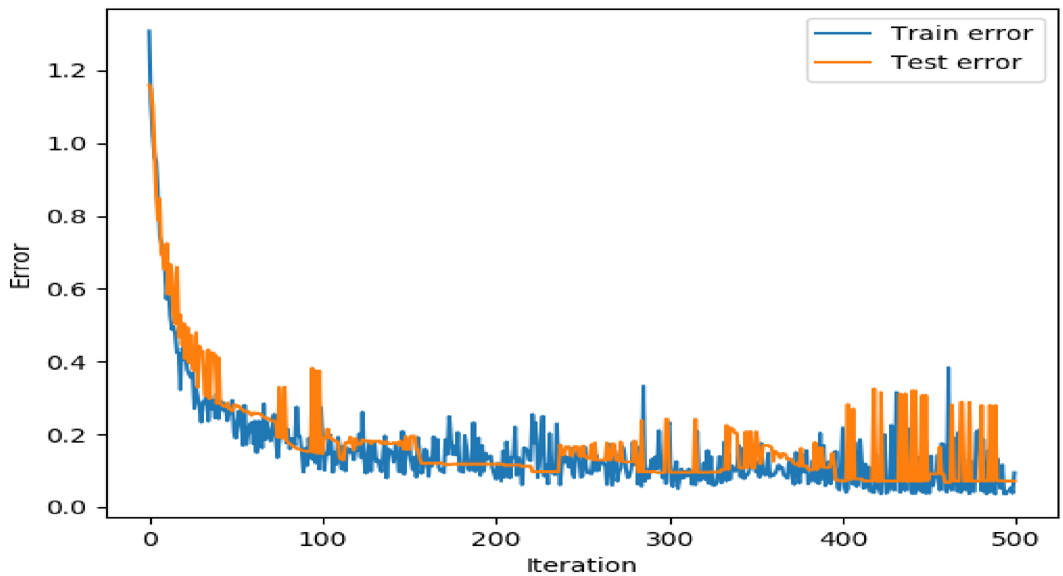
Number of hidden layer:

We had an experiment by adding more hidden layers.

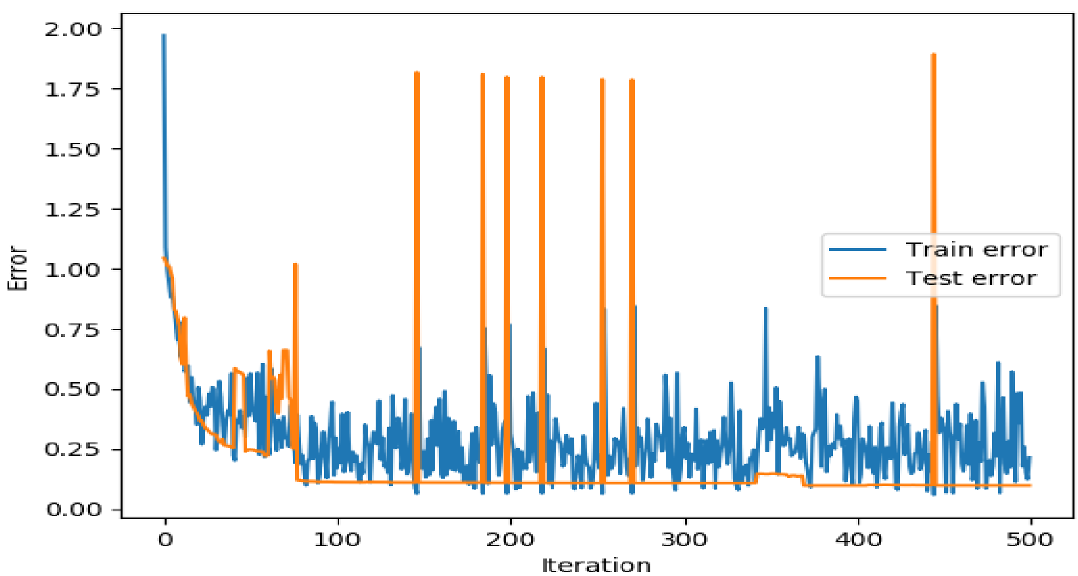
Configuration:

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

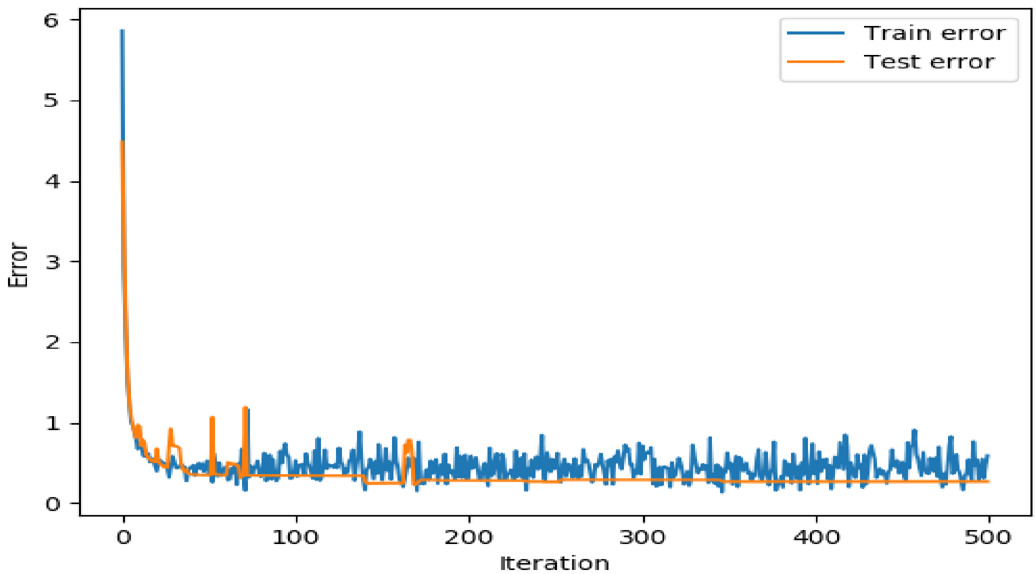
Number of hidden layer:1



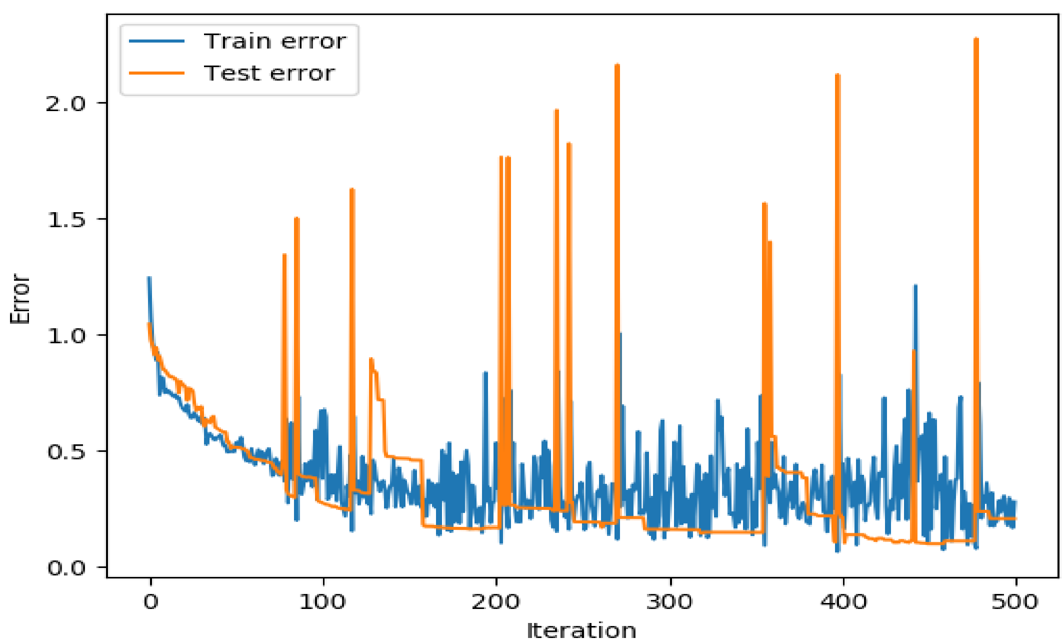
Number of hidden layer:2



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)
```

```python
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()
```