ISTANBUL TECHNICAL UNIVERSITY

ELEKTRICAL AND ELECTRONICS FACULTY

ARTIFICIAL NEURAL NETWORKS AND SYSTEM IDENTIFICATION FOR CONTROL

Homework - I

Prof. Serhat Şeker

Eren Çakmak - 504181134

5 March 2019

Introduction

Aim of this homework is to train an artificial neural network as it is illustrated in Figure 1. As it can be seen in the figure, one only one perceptron is trained. The input X and the desired output T data is given below.

$$X = \begin{pmatrix} 0.5 & 0.8 & 0.9 \\ 0.3 & 0.7 & 0.8 \\ 0.1 & 0.4 & 0.6 \end{pmatrix}, T = \begin{pmatrix} 0.8 \\ 0.7 \\ 0.4 \end{pmatrix}$$

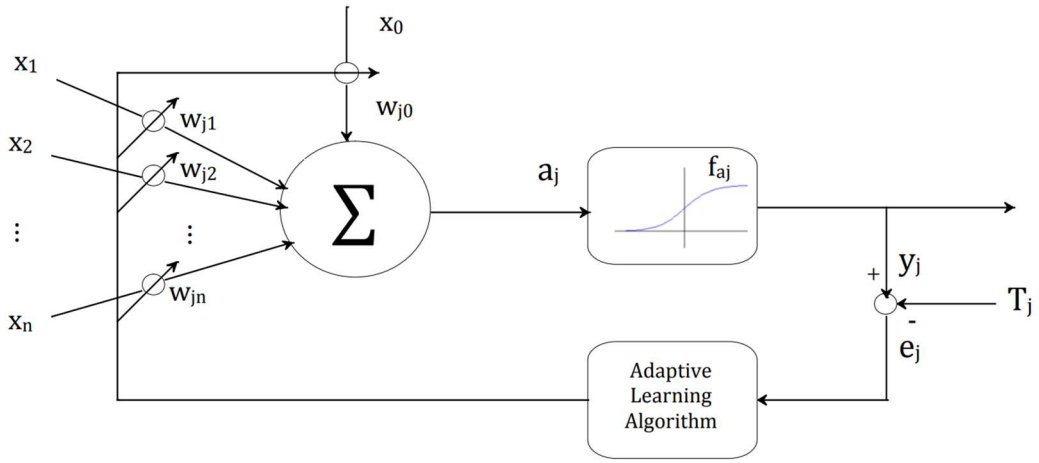

Figure 1: One node artificial neuron.

What is wanted from the perceptron is, learning that when stimulated by a 3x1 vector output will be the second element of the vector. Thus, the desired values for each vector equals to middle element of the input.

The adaptive learning algorithm is chosen as the Backpropagation. The algorithm is trained with a number of learning rates. Additionally, a validation data is derived to observe the output. The validation data is given below.

$$X\_valid = \begin{pmatrix} 0.2 & 0.8 & 0.8 \\ 0.9 & 0.9 & 0.1 \\ 0.7 & 0.1 & 0.7 \\ 0.9 & 0.9 & 0.9 \end{pmatrix}, T\_valid = \begin{pmatrix} 0.8 \\ 0.9 \\ 0.1 \\ 0.9 \end{pmatrix}$$

The training algorithm is written in python 2.7. The code is attached to Appendix A, and also can be accessed from my github repository: https://github.com/erenleicter/ANN-Sys-Iden-Cont.

The Backpropagation Algorithm

As it is given in the Figure 1, the adaptive learning algorithm updates the weights according to the error. Thus at first, the relationship between error and the weights must be analyzed. As all optimization algorithms, the backpropagation optimizes the cost function. Assume that the

cost function is E for our example. Then, the change in E depending on weights $\frac{\partial E}{\partial w_i}$, can be expressed as follows by using the chain rule of derivative for our network:

$$\frac{\partial E_p}{\partial w_i} = \frac{\partial E_p}{\partial e_p}\frac{\partial e_p}{\partial y_p}\frac{\partial y_p}{\partial V_p}\frac{\partial V_p}{\partial w_i}$$

where,

E_p: E for p^{th} vector in a batch.

e_p: error for p^{th} vector in a batch.

y_p: output for p^{th} vector in a batch.

V_p: net input of nonlinear (activation) function for p^{th} vector in a batch.

E_p: E for p^{th} vector in a batch.

w_i: i^{th} weight.

x_i: i^{th} value of p^{th} vector in a batch.

From the basic definitions of artificial neural networks, these terms can be achieved as follows,

$$\frac{\partial V_p}{\partial w_i} = \frac{\partial(\sum x_{pi} w_i)}{\partial w_i} = x_{pi}$$

$$\frac{\partial y_p}{\partial V_p} = \frac{\partial f(V_p)_p}{\partial V_p}$$

$$\frac{\partial e_p}{\partial y_p} = \frac{\partial(t_p - y_p)}{\partial y_p} = -1$$

$$\frac{\partial E_p}{\partial w_i} = -\frac{\partial E_p}{\partial e_p}\frac{\partial f(V_p)_p}{\partial V_p} x_{pi}$$

Assume that E is the mean square error (total error) and nonlinear function is sigmoid function,

$$\frac{\partial f(V_p)_p}{\partial V_p} = \frac{\partial \frac{1}{1+\exp(Vp)}}{\partial V_p} = \frac{1+\exp(Vp)}{(1+\exp(Vp))^2} - \frac{1}{(1+\exp(Vp))^2} = f(V_p)(1 - f(V_p))$$

$$\frac{\partial E_p}{\partial e_p} = \frac{\partial(\frac{e_p^2}{2})}{\partial e_p} = e_p$$

,henceforth, the $\frac{\partial E}{\partial w_i}$ becomes,

$$\frac{\partial E_p}{\partial w_i} = -e_p f(V_p)(1 - f(V_p)) x_{pi}$$

It can be understood that if x_{pi} and error are both positive, related weights and total error are inversely proportional. Hence, to minimize the total error related weights must be increased; which is also can be intuitively understood that, if y is smaller than t, related weights must be increased. But the equation () also shows that weights and total error are proportional. For that reason, increasing in weight to minimize total error can be shown with "η" proportional gain which is also called "learning rate" as follows:

$$\Delta_p w_i = \eta e_p f(V_p)(1 - f(V_p))x_{pi}$$

If the differential is taken as subtraction of different state of weights, then the update function is found as follows:

$$w_i(n+1) = w_i(n) + \eta e_p f(V_p)(1 - f(V_p))x_{pi}$$

This equation is the backpropagation update function for our example.

Results

Results are given figure 2 and figure 3. The figure 2 shows the total training error, validation error, and the output for the test data versus iterations for different learning rates. The figure 3 shows how weights change in each iteration for different learning rates. There are 4 weights because the first weight belongs to the bias (-1).

As it can be seen from figure 2, for 1.0 learning rate at 10^{th} iteration both validation and total error converges to 0 but the output for test data is the worst of the results. However, for 1.0 learning rate at 1000^{th} iteration, even though total error and the output for test data seems reasonable, validation data is at its worst. Considering the 0.06 total error means approximately ±0.346, the system is not ready at that state. For 1.0 learning rate, after 10000^{th} iterations, both validation and training errors converge to 0 and output of the system is around 0.54 and has the error of 0,0008 which seem successful.

In different learning rates, the algorithm follows the same regime. Because the system output is always in linear region of sigmoid function, change in learning rate does not affect anything but the settling time.

As it can be intuitively seen the solution is approximately [0, 0, 1, 0], the results show that neural networks also found reasonable solution.

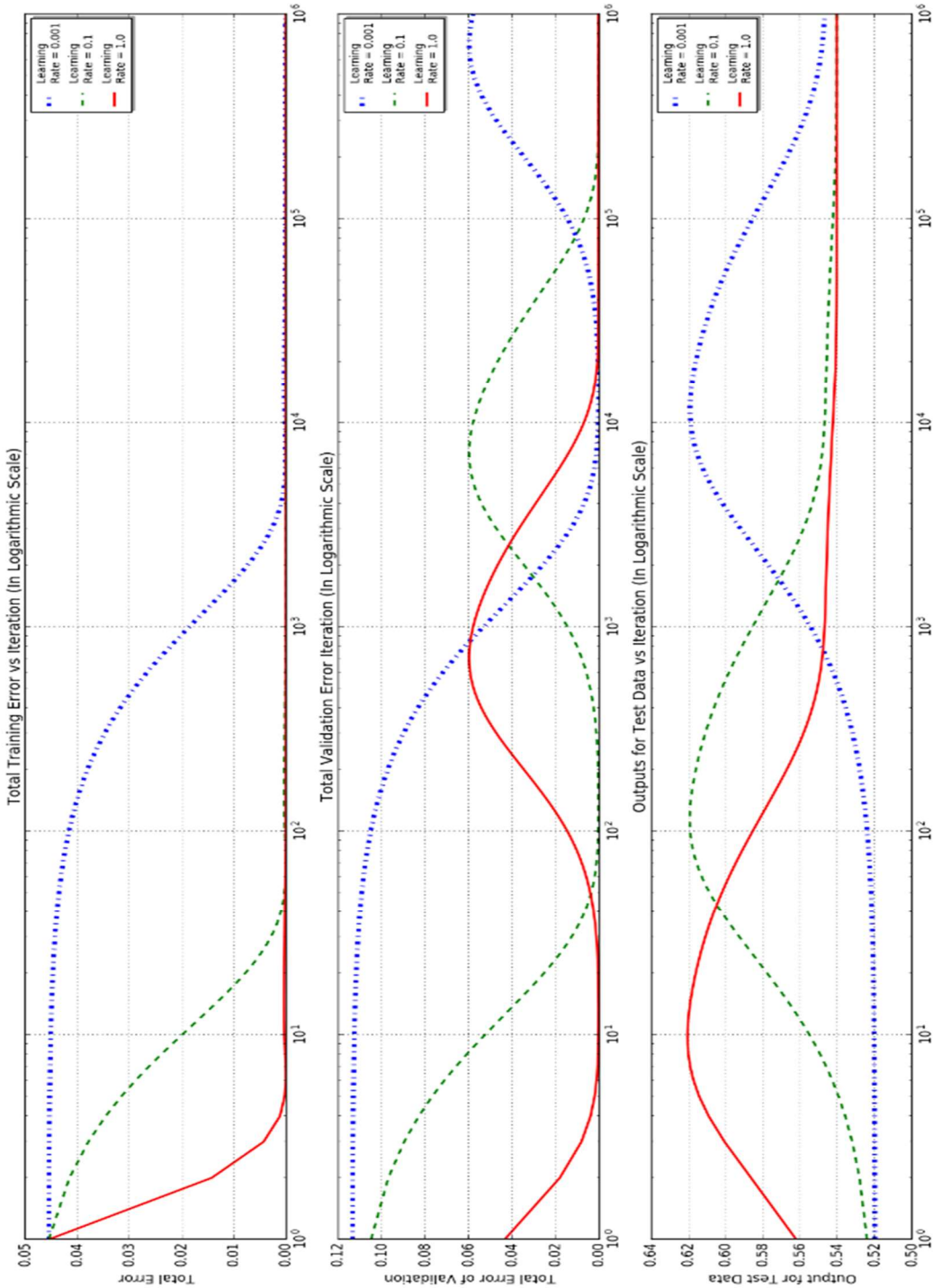

Figure 2: The total training error, validation error, and the output for the test data versus iterations for different learning rates.

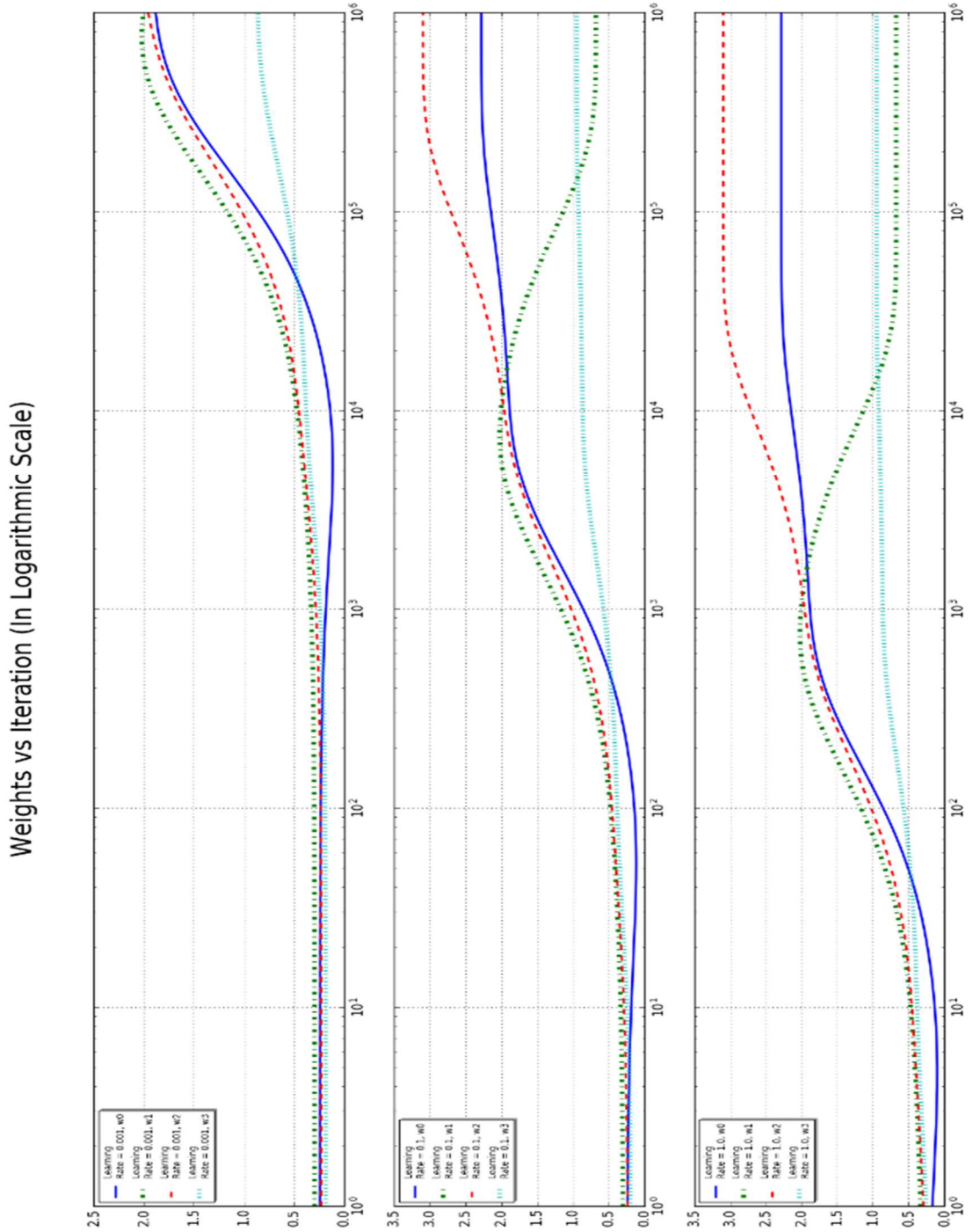

Figure 3: How weights change in each iteration for different learning rates

Appendix A

Code

```
"""
MIT License

Copyright (c) 2019 erenleicter

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
"""
import numpy as np
from matplotlib import pyplot as plt

# X: input
X = np.array(([-1, .5, .8, .9],
              [-1, .3, .7, .8],
              [-1, .1, .4, .6]), dtype=np.float64)

# T: desired input
T = np.array(([.8],
              [.7],
              [.4]), dtype=np.float64)

# X: validation input
X_validation = np.array(([-1, .2, .8, .8],
                         [-1, .9, .9, .1],
                         [-1, .7, .1, .7],
                         [-1, .9, .9, .9]), dtype=np.float64)

# T: validation desired input
T_validation = np.array(([.8],
                         [.9],
                         [.1],
                         [.9],), dtype=np.float64)
# X: test input
X_test = np.array(([-1, .2, .5, .8]), dtype=np.float64)


class OneNodeNeuron(object):
    """
    This class initializes n input m output one node neuron. Hence no hidden layers included.
    """
    def __init__(self, input_size, output_size, learning_rate):
        """
        For n x m node:

        :param input_size: n (int).
        :param output_size: m (int).
        :param learning_rate: Learning rate
        """
        self.inputSize = input_size
        self.outputSize = output_size
        self.lr = learning_rate
        self.W1 = np.random.rand(self.inputSize, self.outputSize)

    @staticmethod
    def activation_function(v):
        """
```

```
    This staticmethod only contains sigmoid function as the activation function.

    :param v: V net input.
    :return: Output value of the node.
    """
    # Sigmoid function.
    return 1 / (1 + np.exp(-v))

def feed_forward(self, x):
    """
    This method returns the output for this one node network.

    :param x: Input data, numpy.array(n, x).
    :return: The output.
    """
    self.V = np.dot(x, self.W1)
    y = self.activation_function(self.V)
    self.y = y
    return y

def activation_function_derivative(self, v):
    """
    This method only contains derivative of sigmoid function as the activation function.

    :param v: V net input.
    :return: Derivative of F(V).
    """
    # Derivative of sigmoid.
    return self.activation_function(v) * (1 - self.activation_function(v))

def back_propagation(self, x, t, y):
    """
    Back propagation algorithm for one node.

    :param x: Input data, numpy.array(n, x).
    :param t: Desired output data, numpy.array(n).
    :param y: Output value of the node.
    :return: None.
    """

    self.error = t - y
    self.update = self.error * (self.activation_function_derivative(self.V))

    self.W1 += self.lr * x.T.dot(self.update)

def train(self, x, t, iterate, outputs_lr, total_error_lr, valid_error_lr, weights_lr):
    """
    This method does feed forward and back propagation in each iteration.

    :param x: Input data, numpy.array(n, x).
    :param t: Desired output data, numpy.array(n).
    :param iterate: Number of iteration.
    :param outputs_lr: Outputs for test data in each iteration.
    :param total_error_lr: Total errors in each iteration.
    :param valid_error_lr: Validation errors in each iteration.
    :param weights_lr: Weights data in each iteration.
    :return: None.
    """

    for ith_element in xrange(iterate):
        y = self.feed_forward(x)
        self.back_propagation(x, t, y)

        outputs_lr[ith_element] = self.feed_forward(X_test)
        total_error_lr[ith_element] = self.total_error()
        valid_error_lr[ith_element] = self.validation_total_error()
        for kth_element in range(len(x[0])):
            weights_lr[kth_element, ith_element] = self.W1[kth_element].copy()

@staticmethod
def save_weights(name, w):
    """
    This static method saves the weights as .txt file.

    :param name: Name of the file, string.
    :param w: Weight vector, np.array().
```

```
        :return: None.
        """
        np.savetxt("weights.txt", w, fmt="%s")

    def use_same_weights(self, weights):
        """
        This method allows the using the saved weights.

        :param weights: np.rand(n, m)
        :return:
        """
        self.W1 = weights.copy()

    def test(self, x):
        """
        This method just test the current weighted network

        :param x: Input data, numpy.array(n, x).
        :return:
        """
        self.feed_forward(x)

    def validation_total_error(self):
        """

        :return:
        """
        error_validation = T_validation - self.feed_forward(X_validation)
        total_error = error_validation.sum()
        return total_error**2 / 2

    def total_error(self):
        e = self.error.sum()
        return e**2 / 2


# Iteration number
iterations = 1000000

# Learning rates
lr1 = 0.001
lr2 = 0.1
lr3 = 1.

# Instances
neuron1 = OneNodeNeuron(len(X[0]), 1, lr1)
neuron2 = OneNodeNeuron(len(X[0]), 1, lr2)
neuron3 = OneNodeNeuron(len(X[0]), 1, lr3)

# Initializes
total_error_lr1 = np.zeros(iterations)
total_error_lr2 = np.zeros(iterations)
total_error_lr3 = np.zeros(iterations)

valid_error_lr1 = np.zeros(iterations)
valid_error_lr2 = np.zeros(iterations)
valid_error_lr3 = np.zeros(iterations)

outputs_lr1 = np.zeros(iterations)
outputs_lr2 = np.zeros(iterations)
outputs_lr3 = np.zeros(iterations)

weights_lr1 = np.zeros([len(X[0]), iterations])
weights_lr2 = np.zeros([len(X[0]), iterations])
weights_lr3 = np.zeros([len(X[0]), iterations])

# This is used to start all network with the same weights
W1 = neuron1.W1.copy()

# Trainings
neuron1.train(X, T, iterations, outputs_lr1, total_error_lr1, valid_error_lr1, weights_lr1)

neuron2.use_same_weights(W1)
neuron2.train(X, T, iterations, outputs_lr2, total_error_lr2, valid_error_lr2, weights_lr2)

neuron3.use_same_weights(W1)
```

```
neuron3.train(X, T, iterations, outputs_lr3, total_error_lr3, valid_error_lr3, weights_lr3)

# Saving last weights as txt file
neuron1.save_weights("first_neuron", neuron1.W1)
neuron2.save_weights("second_neuron", neuron2.W1)
neuron3.save_weights("third_neuron", neuron2.W1)

# Testing
neuron1.test(X_test)
neuron2.test(X_test)
neuron3.test(X_test)

final_results = [neuron1.test(X_test), neuron2.test(X_test), neuron3.test(X_test)]
print("Final results: \n{}".format(final_results))

# Plots

iterations_fig = np.arange(1, iterations+1)

# First figure
fig, axs = plt.subplots(3, 1)

label_names = ['Learning \nRate = ' + str(lr1),
               'Learning \nRate = ' + str(lr2),
               'Learning \nRate = ' + str(lr3)]

axs[0].set_title('Total Training Error vs Iteration (In Logarithmic Scale)')
axs[0].set_ylabel('Total Error')
axs[0].semilogx(iterations_fig, total_error_lr1, label=label_names[0], lw=4, ls='-.',)# c='black')
axs[0].semilogx(iterations_fig, total_error_lr2, label=label_names[1], lw=2, ls='--',)# c='black')
axs[0].semilogx(iterations_fig, total_error_lr3, label=label_names[2], lw=2,)# c='black')

axs[1].set_title('Total Validation Error Iteration (In Logarithmic Scale)')
axs[1].set_ylabel('Total Error of Validation')
axs[1].semilogx(iterations_fig, valid_error_lr1, label=label_names[0], lw=4, ls='-.',)# c='black')
axs[1].semilogx(iterations_fig, valid_error_lr2, label=label_names[1], lw=2, ls='--',)# c='black')
axs[1].semilogx(iterations_fig, valid_error_lr3, label=label_names[2], lw=2,)# c='black')

axs[2].set_title('Outputs for Test Data vs Iteration (In Logarithmic Scale)')
axs[2].set_ylabel('Output for Test Data')
axs[2].semilogx(iterations_fig, outputs_lr1, label=label_names[0], lw=4, ls='-.',)# c='black')
axs[2].semilogx(iterations_fig, outputs_lr2, label=label_names[1], lw=2, ls='--',)# c='black')
axs[2].semilogx(iterations_fig, outputs_lr3, label=label_names[2], lw=2,)# c='black')

axs[0].legend(loc='upper right', shadow=True, fontsize='small')
axs[1].legend(loc='upper right', shadow=True, fontsize='small')
axs[2].legend(loc='upper right', shadow=True, fontsize='small')

axs[0].grid(True)
axs[1].grid(True)
axs[2].grid(True)

# Second figure
fig1, ax = plt.subplots(3, 1)
fig1.suptitle('Weights vs Iteration (In Logarithmic Scale)', fontsize=24)

line_style = '-'
line_width = 2

for i in range(len(X[0])):
    ax_label0 = label_names[0] + ', w' + str(i)
    ax_label1 = label_names[1] + ', w' + str(i)
    ax_label2 = label_names[2] + ', w' + str(i)

    if i == 1:
        line_style = '-.'
        line_width = 4
    elif i == 2:
        line_style = '--'
        line_width = 2
    elif i == 3:
        line_style = ':'
        line_width = 4

    ax[0].semilogx(iterations_fig, weights_lr1[i, :], label=ax_label0, lw=line_width, ls=line_style)
    ax[1].semilogx(iterations_fig, weights_lr2[i, :], label=ax_label1, lw=line_width, ls=line_style)
```

```
    ax[2].semilogx(iterations_fig, weights_lr3[i, :], label=ax_label2, lw=line_width, ls=line_style)

ax[0].legend(loc='upper left', shadow=True, fontsize='x-small')
ax[1].legend(loc='upper left', shadow=True, fontsize='x-small')
ax[2].legend(loc='upper left', shadow=True, fontsize='x-small')

ax[0].grid(True)
ax[1].grid(True)
ax[2].grid(True)

plt.show()
```