ECE 60146 Deep Learning **Name:** *Nikita Ravi (ravi30)*
Homework 3
02/06/2023

1 Introduction

The purpose of this homework assignment is to understand step size optimization in deep learning. The assignment dives into and compares the efficiency of the Stochastic Gradient Descent (SGD), SGD+, and the Adam optimizers.

2 Theoretical Background

2.1 Gradient Descent

Gradient descent is an iterative approach where the objective is to take a step towards the descent (negative gradient of the hyperplane) and converge at the global minimum point of the loss function and identify the parameters that give the minimum loss. The equation for gradient descent is as follows:

$$p_{t+1} = p_t - \alpha \times g_{t+1} \tag{1}$$

Where p_t is the learnable parameter at parameter t, α is the learning rate, and g_t is the gradient of the cost function at parameter t

2.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) uses the same principle, eq. (1), as the normal gradient descent approach. The only thing it differs in is that unlike gradient descent, SGD doesn't use all the training data at once at each iteration. SGD updates its learnable parameters by using only small batches of randomly drawn training samples. This is because if the current solution point in the parameter hyperplane is at a local minimum in the cost-function surface corresponding to the current batch, it would be highly unlikely that the same point would be a local minimum for the cost-function surfaces corresponding to the future randomly drawn batches of samples.

2.3 Stochastic Gradient Descent Plus

Stochastic Gradient Descent Plus (SGD+) is an extension of SGD where we now introduce a factor called momentum, μ. The purpose of momentum is to dampen the oscillation around the minimum by retaining a fraction of the previous gradient. The equation for SGD+ is therefore

$$v_{t+1} = (\mu \times v_t) + g_{t+1} \tag{2}$$

$$p_{t+1} = p_t - \alpha \times v_{t+1} \tag{3}$$

where v_t is the step size at parameter t.

2.4 Adam Optimizer

The Adam optimizer keeps a running average of both the first and second moment of gradients, and takes both these moments into consideration for calculating the step size, thus adapting the learning rate and converging at the minimum quicker. The equations for Adam is as shown below

$$m_{t+1} = \beta_1 \times m_t + (1 - \beta_1) \times g_{t+1} \tag{4}$$

$$v_{t+1} = \beta_2 \times v_t + (1 - \beta_2) \times g_{t+1}^2 \tag{5}$$

$$p_{t+1} = p_t - \alpha \times \frac{\hat{m}}{\sqrt{\hat{v} + \epsilon}} \tag{6}$$

Where m_t and v_t are moments at parameter t and $\hat{m}$ is

$$\hat{m_k} = \frac{m_k}{\sqrt{1 - \beta_1^k}} \tag{7}$$

and $\hat{v}$ is

$$\hat{v_k} = \frac{v_k}{\sqrt{1 - \beta_2^k}} \tag{8}$$

where k is the iteration k and v_k and m_k is the moment at parameter t at iteration k.

3 Methodology

Modifications are made to Professor Kak's professor to implement SGD+ and the Adam optimizers.

1. We create two new sub classes that inherit from the **ComputationalGraphPrimer** class for the SGD+ and Adam approach

2. The **run_training_loop_one_neuron_model** and **run_training_loop_multi_neuron_model** methods are created again in their respective subclasses so that the method from the parent class is overridden. These methods are modified to initialize new variables for updating the learnable parameters

   (a) For SGD+, we introduce a list storing all the step sizes, momentum, and a new bias factor that updates the bias based on the momentum. The momentum is set to 0.85

   (b) For Adam, we introduce β_1, β_2, ϵ, a list of weights for moment m and moment v, and their respective biases. β_1 is set to 0.9, β_2 is set to 0.999, and ϵ is set to 1e$-$6

3. The **backprop_and_update_params_one_neuron_model** and **backprop_and_update_params_multi_neuron_model** are created again in their respective subclasses so that the method from the parent class is overridden. These methods are modified to include Equations (2) and (3) for SGD+ and Equations (4)-(8) for Adam to update the step size, learnable parameters, and bias.

4. The losses for SGD, SGD+, and Adam are displayed in a graph for two different learning rates as shown in the results section

3.1 One Neuron Classifier

```
# Import Libraries
import random
import numpy as np
import matplotlib.pyplot as plt
import operator
from ComputationalGraphPrimer import *

# Constants
SEED = 512
random.seed(SEED)
np.random.seed(SEED)


class ComputationalGraphPrimerSGDPlus(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs) # Inheriting from the parent class

    # Modifying and Overriding the run_training_loop_one_neuron_model to implement SGD+
```

```
    # mu is between [0,1]
    def run_training_loop_one_neuron_model(self, training_data, mu=0.5):
        #########################################Copied from the original function###################################
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params} # initializing learnable parameters with random numbers from a uniform distribution over the interval (0,1)

        self.bias = random.uniform(0,1)                   ## Adding the bias improves class discrimination.
                                                          ##   We initialize it to a random number.

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly from the two
            lists mentioned above.  And it mush also associate the class label with each sample
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]   ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
```

```
                cointoss = random.choice([0,1])
     ## When a batch is created by getbatch(), we want the

     ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]
     ## First list for samples, the second for labels
                maxval = 0.0
     ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]
     ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

        #
    ###################################################################################

        # Modified part of the function
        self.mu = mu
        self.bias_factor = 0 # Update the bias, the factor depends on the
    current mu
        self.step_sizes = [0 for i in range(len(self.learnable_params) +
    1)]

        #########################################Copied from the original
    function###################################
        data_loader = DataLoader(training_data, batch_size=self.batch_size
    )
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0
    ##  Average the loss over iterations for printing out

     ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(
    data_tuples)               ##  FORWARD PROP of data
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
    range(len(class_labels))])  ##  Find loss
            loss_avg = loss / float(len(class_labels))
```

```
                              ##  Average the loss over batch
            avg_loss_over_iterations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1,
avg_loss_over_iterations))                 ## Display average loss
                avg_loss_over_iterations = 0.0
                             ## Re-initialize avg loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(
class_labels))
            data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
            data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                 [float(len(class_labels))] * len(
class_labels) ))
            self.backprop_and_update_params_one_neuron_model(y_error_avg,
data_tuple_avg, deriv_sigmoid_avg)     ## BACKPROP loss
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()
        return loss_running_record

        #
##################################################################################

 # Modify backpropagation function for one_neuron_model -
backpropagating the loss and updating the values of the learnable
parameters.
 def backprop_and_update_params_one_neuron_model(self, y_error,
vals_for_input_vars, deriv_sigmoid):
        """
        As should be evident from the syntax used in the following call to
 backprop function,

           self.backprop_and_update_params_one_neuron_model( y_error_avg,
data_tuple_avg, deriv_sigmoid_avg)
                                                                   ^^^
           ^^^                 ^^^
        the values fed to the backprop function for its three arguments
are averaged over the training
        samples in the batch.  This in keeping with the spirit of SGD that
 calls for averaging the
        information retained in the forward propagation over the samples
in a batch.

        See Slide 59 of my Week 3 slides for the math of back propagation
for the One-Neuron network.
        """
        input_vars = self.independent_vars
        vals_for_input_vars_dict = dict(zip(input_vars, list(
vals_for_input_vars)))
```

```python
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## Calculate the next step in the parameter hyperplane
            ##################################################Modified ##########################################
            self.step_sizes[i + 1] = (self.mu * self.step_sizes[i]) + (self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid)
            self.step_sizes[i] = self.step_sizes[i + 1] # Save the new current step size in the previous iteration position

            ## Update the learnable parameters
            # self.vals_for_learnable_params[param] += -self.learning_rate * self.step_sizes[i + 1]
            self.vals_for_learnable_params[param] += self.step_sizes[i + 1]

        self.bias_factor = (self.mu * self.bias_factor) + (self.learning_rate * y_error * deriv_sigmoid) ## Update the bias
        self.bias += self.bias_factor
    # ###################################################################################

class ComputationalGraphPrimerAdam(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    # Modifying and Overriding the run_training_loop_one_neuron_model to implement SGD+
    # Beta1 and Beta2 are close to 1
    def run_training_loop_one_neuron_model(self, training_data, beta1=0.9, beta2=0.999, e=1e-6):
        #########################################Copied from the original function###################################
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params} # initializing learnable parameters with random numbers from a uniform distribution over the interval (0,1)

        self.bias = random.uniform(0,1)                   ## Adding the bias improves class discrimination.
                                                          ##   We initialize it to a random number.

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method,
```

```
    the training dataset
    consists of a Python dict with two keys, 0 and 1, the former points to a list of
    all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
    the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
    classes have different means and variances.  The dimensionality of each data sample
    is set by the number of nodes in the input layer of the neural network.

    The data loader's job is to construct a batch of samples drawn randomly from the two
    lists mentioned above.  And it mush also associate the class label with each sample
    separately.
    """
    def __init__(self, training_data, batch_size):
        self.training_data = training_data
        self.batch_size = batch_size
        self.class_0_samples = [(item, 0) for item in self.training_data[0]]   ## Associate label 0 with each sample
        self.class_1_samples = [(item, 1) for item in self.training_data[1]]   ## Associate label 1 with each sample

    def __len__(self):
        return len(self.training_data[0]) + len(self.training_data[1])

    def _getitem(self):
        cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                   ##   samples to be chosen randomly from the two lists
        if cointoss == 0:
            return random.choice(self.class_0_samples)
        else:
            return random.choice(self.class_1_samples)

    def getbatch(self):
        batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
        maxval = 0.0                                               ## For approximate batch data normalization
        for _ in range(self.batch_size):
            item = self._getitem()
            if np.max(item[0]) > maxval:
                maxval = np.max(item[0])
            batch_data.append(item[0])
            batch_labels.append(item[1])
        batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
        batch = [batch_data, batch_labels]
```

```
                return batch

        #
    ########################################################################################

        # Modified part of the function
        self.beta1, self.beta2 = beta1, beta2
        self.e = e
        self.m_db, self.v_db = 0, 0
        self.m_dw = [0 for i in range(len(self.learnable_params) + 1)] # m
        self.v_dw = [0 for i in range(len(self.learnable_params) + 1)] # v


        #########################################Copied from the original
    function####################################
        data_loader = DataLoader(training_data, batch_size=self.batch_size
    )
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0
    ##  Average the loss over iterations for printing out

     ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(
    data_tuples)                ##  FORWARD PROP of data
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
    range(len(class_labels))])  ##  Find loss
            loss_avg = loss / float(len(class_labels))
                                ##  Average the loss over batch
            avg_loss_over_iterations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1,
    avg_loss_over_iterations))                  ## Display average loss
                avg_loss_over_iterations = 0.0
                               ## Re-initialize avg loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(
    class_labels))
            data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
            data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                [float(len(class_labels))] * len(
    class_labels) ))
            self.backprop_and_update_params_one_neuron_model(y_error_avg,
    data_tuple_avg, deriv_sigmoid_avg, i + 1)     ## BACKPROP loss
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()
```

```python

        return loss_running_record

        # #################################################################################

    # Modify backpropagation function for one_neuron_model - backpropagating the loss and updating the values of the learnable parameters.
    def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid, k):
        """
        As should be evident from the syntax used in the following call to backprop function,

           self.backprop_and_update_params_one_neuron_model( y_error_avg, data_tuple_avg, deriv_sigmoid_avg)
                                                                     ^^^                ^^^                   ^^^
        the values fed to the backprop function for its three arguments are averaged over the training
        samples in the batch.  This in keeping with the spirit of SGD that calls for averaging the
        information retained in the forward propagation over the samples in a batch.

        See Slide 59 of my Week 3 slides for the math of back propagation for the One-Neuron network.
        """
        input_vars = self.independent_vars
        vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## Calculate the next step in the parameter hyperplane
            ################################Modified#################################
            self.m_dw[i + 1] = (self.beta1 * self.m_dw[i]) + ((1 - self.beta1) * (self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid))
            self.m_dw[i] = self.m_dw[i + 1] # Save the new current moment in the previous iteration position

            self.v_dw[i + 1] = (self.beta2 * self.v_dw[i]) + ((1 - self.beta2) * (self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid)**2)
            self.v_dw[i] = self.v_dw[i + 1] # Save the new current moment in the previous iteration position

            ## Update the learnable parameters
            mk_hat = self.m_dw[i + 1] / (1 - self.beta1 ** k)
            vk_hat = self.v_dw[i + 1] / (1 - self.beta2 ** k)

```

```
            self.vals_for_learnable_params[param] += mk_hat / np.sqrt(vk_hat + self.e)

        # Inspired by: https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b217f1cc
        # Inspired by: https://www.youtube.com/watch?v=JXQT_vxqwIs&ab_channel=DeepLearningAI
        self.m_db = (self.beta1 * self.m_db) + (1 - self.beta1) * (self.learning_rate * y_error * deriv_sigmoid)
        self.v_db = (self.beta2 * self.v_db) + (1 - self.beta2) * (self.learning_rate * y_error * deriv_sigmoid) ** 2

        m_db_hat = self.m_db / (1 - self.beta1 ** k)
        v_db_hat = self.v_db / (1 - self.beta1 ** k)

        self.bias += m_db_hat / np.sqrt(v_db_hat + self.e) ## Update the bias
    # ##################################################################################

def sgd_plus(lr=1e-3, mu=0.9):
    cgp = ComputationalGraphPrimerSGDPlus(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = lr,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

    cgp.parse_expressions()
    # cgp.display_one_neuron_network()

    training_data = cgp.gen_training_data()
    loss_per_iteration = cgp.run_training_loop_one_neuron_model(training_data, mu=mu)

    return loss_per_iteration

def adam(lr=1e-3):
    cgp = ComputationalGraphPrimerAdam(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = lr,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
```

```
    )

    cgp.parse_expressions()
    # cgp.display_one_neuron_network()

    training_data = cgp.gen_training_data()
    loss_per_iteration = cgp.run_training_loop_one_neuron_model(
   training_data )

    return loss_per_iteration

def sgd(lr=1e-3):
    cgp = ComputationalGraphPrimer(
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )

    cgp.parse_expressions()
    # cgp.display_one_neuron_network()

    training_data = cgp.gen_training_data()
    loss_per_iteration = cgp.run_training_loop_one_neuron_model(
   training_data )

    return loss_per_iteration

def plot_losses(sgd, sgd_plus, adam, lr):
    number_of_iterations = len(adam)
    plt.plot(range(number_of_iterations), sgd, label="SGD Loss")
    plt.plot(range(number_of_iterations), sgd_plus, label="SGD+ Loss")
    plt.plot(range(number_of_iterations), adam, label="Adam Loss")

    plt.title(f"Loss per Iteration for Different Optimizers for One Neuron
    Model for Learning Rate: {lr}")
    plt.xlabel("Iteration Number")
    plt.ylabel("Loss")
    plt.legend(loc="upper left")

    plt.show(); quit()
    plt.savefig(r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW3/
   one_neuron_" + str(lr) + "_learning_rate.png", dpi=200)

if __name__ == "__main__":
    lr = 1e-3
    sgd_loss = sgd(lr)
    sgd_plus_loss = sgd_plus(lr)
    adam_loss = adam(lr)
```

```python

    plot_losses(sgd_loss, sgd_plus_loss, adam_loss, lr)
```

Listing 1: SGD+ and Adam for One Neuron Classifier

3.2 Multi-Neuron Classifier

```python
# Import Libraries
import random
import numpy as np
import matplotlib.pyplot as plt
import operator
from ComputationalGraphPrimer import *

# Constants
SEED = 1234
random.seed(SEED)
np.random.seed(SEED)

class ComputationalGraphPrimerSGDPlus(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs) # Inheriting from the parent class

    # Modifying and Overriding the run_training_loop_one_neuron_model to implement SGD+
    # mu is between [0,1]

    def run_training_loop_multi_neuron_model(self, training_data, mu=0.5):
        ########################################Copied from the original function####################################
        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

```

```
        The data loader's job is to construct a batch of samples drawn randomly from the two
        lists mentioned above.  And it mush also associate the class label with each sample
        separately.
        """
        def __init__(self, training_data, batch_size):
            self.training_data = training_data
            self.batch_size = batch_size
            self.class_0_samples = [(item, 0) for item in self.training_data[0]]    ## Associate label 0 with each sample
            self.class_1_samples = [(item, 1) for item in self.training_data[1]]    ## Associate label 1 with each sample

        def __len__(self):
            return len(self.training_data[0]) + len(self.training_data[1])

        def _getitem(self):
            cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                       ##   samples to be chosen randomly from the two lists
            if cointoss == 0:
                return random.choice(self.class_0_samples)
            else:
                return random.choice(self.class_1_samples)

        def getbatch(self):
            batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
            maxval = 0.0                                               ## For approximate batch data normalization
            for _ in range(self.batch_size):
                item = self._getitem()
                if np.max(item[0]) > maxval:
                    maxval = np.max(item[0])
                batch_data.append(item[0])
                batch_labels.append(item[1])
            batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
            batch = [batch_data, batch_labels]
            return batch
    # ####################################################################################

    # Modified part of the function
    self.mu = mu
    self.step_sizes = [0 for i in range(len(self.learnable_params) + 1)]
    self.bias_factor = 0

    ########################################Copied from the original function####################################
```

```
        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]      ## Adding the bias to each layer improves
                                                                                  ##   class discrimination. We initialize it
                                                                                  ##   to a random number.

        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                                          ##  Average the loss over iterations for printing out
                                                                                 ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            self.forward_prop_multi_neuron_model(data_tuples)                                ## FORW PROP works by side-effect
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_layers-1]      ## Predictions from FORW PROP
            y_preds =  [item for sublist in  predicted_labels_for_batch  for item in sublist]  ## Get numeric vals for predictions
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss / float(len(class_labels))                                         ## Average the loss over batch
            avg_loss_over_iterations += loss_avg                                              ## Add to Average loss over iterations
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))            ## Display avg loss
                avg_loss_over_iterations = 0.0                                                ## Re-initialize avg-over-iterations loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)      ## BACKPROP loss
```

```
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()

        return loss_running_record

        # ###################################################################################

    # Modify backpropagation function for one_neuron_model - backpropagating the loss and updating the values of the learnable parameters.
    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels):
        """
        First note that loop index variable 'back_layer_index' starts with the index of
        the last layer.  For the 3-layer example shown for 'forward', back_layer_index
        starts with a value of 2, its next value is 1, and that's it.

        Stochastic Gradient Gradient calls for the backpropagated loss to be averaged over
        the samples in a batch.  To explain how this averaging is carried out by the
        backprop function, consider the last node on the example shown in the forward()
        function above.  Standing at the node, we look at the 'input' values stored in the
        variable "input_vals".  Assuming a batch size of 8, this will be list of
        lists. Each of the inner lists will have two values for the two nodes in the
        hidden layer. And there will be 8 of these for the 8 elements of the batch.  We average
        these values 'input vals' and store those in the variable "input_vals_avg".  Next we
        must carry out the same batch-based averaging for the partial derivatives stored in the
        variable "deriv_sigmoid".

        Pay attention to the variable 'vars_in_layer'.  These store the node variables in
        the current layer during backpropagation.  Since back_layer_index starts with a
        value of 2, the variable 'vars_in_layer' will have just the single node for the
        example shown for forward(). With respect to what is stored in vars_in_layer', the
        variables stored in 'input_vars_to_layer' correspond to the input layer with
        respect to the current layer.
        """
```

```
        # backproped prediction error:
        pred_err_backproped_at_layers = {i : [] for i in range(1,self.num_layers-1)}
        pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
        for back_layer_index in reversed(range(1,self.num_layers)):
            input_vals = self.forw_prop_vals_at_layers[back_layer_index-1]
            input_vals_avg = [sum(x) for x in zip(*input_vals)]
            input_vals_avg = list(map(operator.truediv, input_vals_avg, [float(len(class_labels))] * len(class_labels)))
            deriv_sigmoid =  self.gradient_vals_for_layers[back_layer_index]
            deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
            deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
                                                                              [float(len(class_labels))] * len(class_labels)))
            vars_in_layer  =  self.layer_vars[back_layer_index]                 ## a list like ['xo']
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## a list like ['xw', 'xz']

            layer_params = self.layer_params[back_layer_index]
            ## note that layer_params are stored in a dict like
                ##     {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']], 2: [['cp', 'cq']]}
            ## "layer_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"
            transposed_layer_params = list(zip(*layer_params))         ## creating a transpose of the link matrix

            backproped_error = [None] * len(vars_in_next_layer_back)
            for k,varr in enumerate(vars_in_next_layer_back):
                for j,var2 in enumerate(vars_in_layer):
                    backproped_error[k] = sum([self.vals_for_learnable_params[transposed_layer_params[k][i]] *
                                               pred_err_backproped_at_layers[back_layer_index][i]
                                               for i in range(len(vars_in_layer))])
#                                               deriv_sigmoid_avg[i] for i in range(len(vars_in_layer))])
            pred_err_backproped_at_layers[back_layer_index - 1]  =  backproped_error
            input_vars_to_layer = self.layer_vars[back_layer_index-1]
            for j,var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]
                ##  Regarding the parameter update loop that follows, see the Slides 74 through 77 of my Week 3
                ##  lecture slides for how the parameters are updated using the partial derivatives stored away
                ##  during forward propagation of data. The theory underlying these calculations is presented
                ##  in Slides 68 through 71.
```

```python
                for i,param in enumerate(layer_params):
                    gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]
                    #############################################Modified##################################
                    self.step_sizes[i + 1] = (self.mu * self.step_sizes[i]) + (self.learning_rate * gradient_of_loss_for_param * deriv_sigmoid_avg[j])
                    self.step_sizes[i] = self.step_sizes[i + 1]
                    self.vals_for_learnable_params[param] += self.step_sizes[i + 1]

            self.bias_factor = (self.mu * self.bias_factor) + (self.learning_rate * sum(pred_err_backproped_at_layers[back_layer_index]) \
 * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))
            self.bias[back_layer_index-1] += self.bias_factor
 # ###################################################################################

class ComputationalGraphPrimerAdam(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs) # Inheriting from the parent class

    # Modifying and Overriding the run_training_loop_one_neuron_model to implement SGD+
    # mu is between [0,1]

    def run_training_loop_multi_neuron_model(self, training_data, beta1=0.9, beta2=0.999, e=1e-6):
        #########################################Copied from the original function###################################
        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural
```

```
network.

        The data loader's job is to construct a batch of samples drawn randomly from the two
        lists mentioned above.  And it mush also associate the class label with each sample
        separately.
        """
        def __init__(self, training_data, batch_size):
            self.training_data = training_data
            self.batch_size = batch_size
            self.class_0_samples = [(item, 0) for item in self.training_data[0]]    ## Associate label 0 with each sample
            self.class_1_samples = [(item, 1) for item in self.training_data[1]]    ## Associate label 1 with each sample

        def __len__(self):
            return len(self.training_data[0]) + len(self.training_data[1])

        def _getitem(self):
            cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                       ##   samples to be chosen randomly from the two lists
            if cointoss == 0:
                return random.choice(self.class_0_samples)
            else:
                return random.choice(self.class_1_samples)

        def getbatch(self):
            batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
            maxval = 0.0                                               ## For approximate batch data normalization
            for _ in range(self.batch_size):
                item = self._getitem()
                if np.max(item[0]) > maxval:
                    maxval = np.max(item[0])
                batch_data.append(item[0])
                batch_labels.append(item[1])
            batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
            batch = [batch_data, batch_labels]
            return batch
    # ####################################################################################

    # Modified part of the function
    self.beta1, self.beta2 = beta1, beta2
    self.e = e
    self.m_db, self.v_db = 0.0, 0.0
    self.m_dw = [0.0 for i in range(len(self.learnable_params) + 1)] # m
```

```
        self.v_dw = [0.0 for i in range(len(self.learnable_params) + 1)] # v


        #########################################Copied from the original function###################################
        """
        The training loop must first initialize the learnable parameters. Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]      ## Adding the bias to each layer improves
                                                                                ##   class discrimination. We initialize it
                                                                                ##   to a random number.

        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                                         ##  Average the loss over iterations for printing out
                                                                                ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            self.forward_prop_multi_neuron_model(data_tuples)                   ## FORW PROP works by side-effect
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_layers-1]      ## Predictions from FORW PROP
            y_preds =  [item for sublist in  predicted_labels_for_batch for item in sublist]  ## Get numeric vals for predictions
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss / float(len(class_labels))                          ## Average the loss over batch
            avg_loss_over_iterations += loss_avg                                ## Add to Average loss over iterations
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))            ## Display avg loss
```

```
                avg_loss_over_iterations = 0.0                 ## Re-initialize avg-over-iterations loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels, i+1)      ## BACKPROP loss
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()

        return loss_running_record

        # ####################################################################################

    # Modify backpropagation function for one_neuron_model - backpropagating the loss and updating the values of the learnable parameters.
    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels, iteration):
        """
        First note that loop index variable 'back_layer_index' starts with the index of
        the last layer.  For the 3-layer example shown for 'forward', back_layer_index
        starts with a value of 2, its next value is 1, and that's it.

        Stochastic Gradient Gradient calls for the backpropagated loss to be averaged over
        the samples in a batch.  To explain how this averaging is carried out by the
        backprop function, consider the last node on the example shown in the forward()
        function above.  Standing at the node, we look at the 'input' values stored in the
        variable "input_vals".  Assuming a batch size of 8, this will be list of
        lists. Each of the inner lists will have two values for the two nodes in the
        hidden layer. And there will be 8 of these for the 8 elements of the batch.  We average
        these values 'input vals' and store those in the variable "input_vals_avg".  Next we
        must carry out the same batch-based averaging for the partial derivatives stored in the
        variable "deriv_sigmoid".

        Pay attention to the variable 'vars_in_layer'.  These store the node variables in
        the current layer during backpropagation.  Since back_layer_index starts with a
        value of 2, the variable 'vars_in_layer' will have just the single node for the
```

```
        example shown for forward(). With respect to what is stored in vars_in_layer’, the
        variables stored in ’input_vars_to_layer’ correspond to the input layer with
        respect to the current layer.
        """
        # backproped prediction error:
        pred_err_backproped_at_layers = {i : [] for i in range(1,self.num_layers-1)}
        pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
        for back_layer_index in reversed(range(1,self.num_layers)):
            input_vals = self.forw_prop_vals_at_layers[back_layer_index-1]
            input_vals_avg = [sum(x) for x in zip(*input_vals)]
            input_vals_avg = list(map(operator.truediv, input_vals_avg, [float(len(class_labels))] * len(class_labels)))
            deriv_sigmoid =  self.gradient_vals_for_layers[back_layer_index]
            deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
            deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
                                                                        [float(len(class_labels))] * len(class_labels)))
            vars_in_layer  =  self.layer_vars[back_layer_index]      ## a list like [’xo’]
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## a list like [’xw’, ’xz’]

            layer_params = self.layer_params[back_layer_index]
            ## note that layer_params are stored in a dict like
                ##     {1: [[’ap’, ’aq’, ’ar’, ’as’], [’bp’, ’bq’, ’br’, ’bs’]], 2: [[’cp’, ’cq’]]}
            ## "layer_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"
            transposed_layer_params = list(zip(*layer_params))         ## creating a transpose of the link matrix

            backproped_error = [None] * len(vars_in_next_layer_back)
            for k,varr in enumerate(vars_in_next_layer_back):
                for j,var2 in enumerate(vars_in_layer):
                    backproped_error[k] = sum([self.vals_for_learnable_params[transposed_layer_params[k][i]] *
                                               pred_err_backproped_at_layers[back_layer_index][i]
                                               for i in range(len(vars_in_layer))])
#                                               deriv_sigmoid_avg[i] for i in range(len(vars_in_layer))])
            pred_err_backproped_at_layers[back_layer_index - 1]  =  backproped_error
            input_vars_to_layer = self.layer_vars[back_layer_index-1]
            for j,var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]
                ##  Regarding the parameter update loop that follows, see
```

```
the Slides 74 through 77 of my Week 3
                ##  lecture slides for how the parameters are updated using the partial derivatives stored away
                ##  during forward propagation of data. The theory underlying these calculations is presented
                ##  in Slides 68 through 71.
                for i,param in enumerate(layer_params):
                    gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]
                    #############################################Modified###################################
                    self.m_dw[i + 1] = (self.beta1 * self.m_dw[i]) + (1 - self.beta1) * (gradient_of_loss_for_param * deriv_sigmoid_avg[j])
                    self.m_dw[i] = self.m_dw[i + 1] # Save the new current moment in the previous iteration position

                    self.v_dw[i + 1] = (self.beta2 * self.v_dw[i]) + (1 - self.beta2) * (gradient_of_loss_for_param * deriv_sigmoid_avg[j]) ** 2
                    self.v_dw[i] = self.v_dw[i + 1] # Save the new current moment in the previous iteration position

                    ## Update the learnable parameters
                    mk_hat = self.m_dw[i + 1] / (1 - self.beta1 ** iteration)
                    vk_hat = self.v_dw[i + 1] / (1 - self.beta2 ** iteration)

                    self.vals_for_learnable_params[param] += self.learning_rate * mk_hat / np.sqrt(vk_hat + self.e)

            # Inspired by: https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b217f1cc
            # Inspired by: https://www.youtube.com/watch?v=JXQT_vxqwIs&ab_channel=DeepLearningAI
            self.m_db = (self.beta1 * self.m_db) + (1 - self.beta1) * (sum(pred_err_backproped_at_layers[back_layer_index]) \
                                                                                                 * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))
            self.v_db = (self.beta2 * self.v_db) + (1 - self.beta2) * (sum(pred_err_backproped_at_layers[back_layer_index]) \
                                                                                                 * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)) ** 2

            m_db_hat = self.m_db / (1 - self.beta1 ** iteration)
            v_db_hat = self.v_db / (1 - self.beta2 ** iteration)

            self.bias[back_layer_index-1] += self.learning_rate * m_db_hat / np.sqrt(v_db_hat + self.e) ## Update the bias

    # ##########################################################################################

def sgd_plus(lr=1e-3, mu=0.9):
```

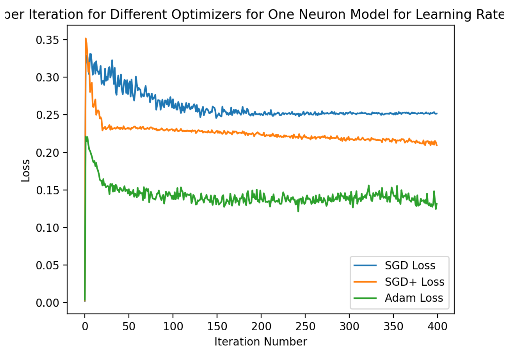
```
    cgp = ComputationalGraphPrimerSGDPlus(
                num_layers = 3,
                layers_config = [4,2,1],                          # num of nodes in each layer
                expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                               'xz=bp*xp+bq*xq+br*xr+bs*xs',
                               'xo=cp*xw+cq*xz'],
                output_vars = ['xo'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
      )

    cgp.parse_multi_layer_expressions()
    # cgp.display_multi_neuron_network()

    training_data = cgp.gen_training_data()
    loss_per_iteration = cgp.run_training_loop_multi_neuron_model(training_data, mu)

    return loss_per_iteration

def adam(lr=1e-3):
    cgp = ComputationalGraphPrimerAdam(
                num_layers = 3,
                layers_config = [4,2,1],                          # num of nodes in each layer
                expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                               'xz=bp*xp+bq*xq+br*xr+bs*xs',
                               'xo=cp*xw+cq*xz'],
                output_vars = ['xo'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
      )

    cgp.parse_multi_layer_expressions()
    # cgp.display_multi_neuron_network()

    training_data = cgp.gen_training_data()
    loss_per_iteration = cgp.run_training_loop_multi_neuron_model(training_data )

    return loss_per_iteration

def sgd(lr=1e-3):
    cgp = ComputationalGraphPrimer(
                num_layers = 3,
```

```
                layers_config = [4,2,1],                              # num of nodes in each layer
                expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                               'xz=bp*xp+bq*xq+br*xr+bs*xs',
                               'xo=cp*xw+cq*xz'],
                output_vars = ['xo'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
      )

    cgp.parse_multi_layer_expressions()
    training_data = cgp.gen_training_data()

    loss_per_iteration = cgp.run_training_loop_multi_neuron_model( training_data )
    return loss_per_iteration

def plot_losses(sgd, sgd_plus, adam, lr):
    number_of_iterations = len(adam)
    plt.plot(range(number_of_iterations), sgd, label="SGD Loss")
    plt.plot(range(number_of_iterations), sgd_plus, label="SGD+ Loss")
    plt.plot(range(number_of_iterations), adam, label="Adam Loss")

    plt.title(f"Loss per Iteration for Different Optimizers for Multi-Neuron Model for Learning Rate: {lr}")
    plt.xlabel("Iteration Number")
    plt.ylabel("Loss")
    plt.legend(loc="lower left")

    plt.show(); quit()
    plt.savefig(r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW3/multi_neuron_" + str(lr) + "_learning_rate.png", dpi=200)

if __name__ == "__main__":
    lr = 1e-3
    sgd_loss = sgd(lr)
    sgd_plus_loss = sgd_plus(lr)
    adam_loss = adam(lr)

    plot_losses(sgd_loss, sgd_plus_loss, adam_loss, lr)
```

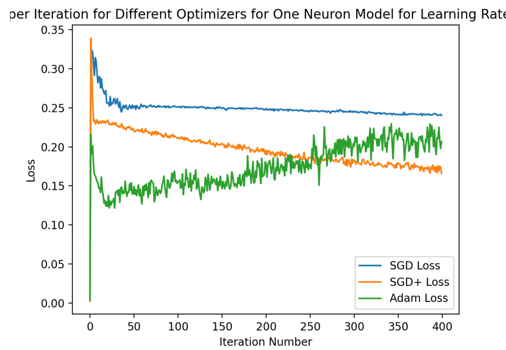
Listing 2: SGD+ and Adam for Multi-Neuron Classifier

4 Results

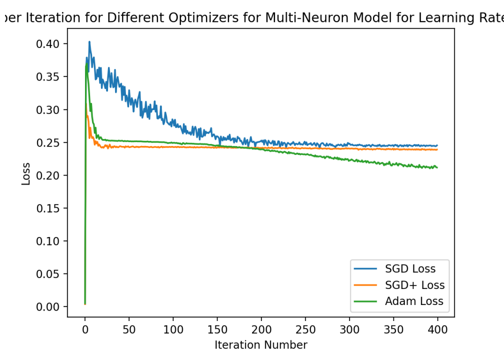
The following figure illustrates the loss from SGD, SGD+, and Adam optimizers at each iteration for a learning rate of 0.001 and 0.005 when using either one-neuron classifier or multi-neuron classifier.



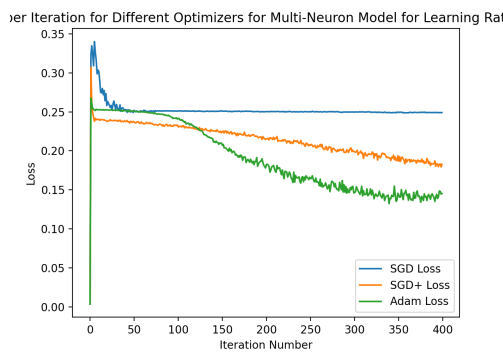
(a) One Neuron Classifier with Learning Rate 0.001



(b) One Neuron Classifier with Learning Rate 0.005



(c) Multi Neuron Classifier with Learning Rate 0.001



(d) Multi Neuron Classifier with Learning Rate 0.005

Figure 1: Losses per Iteration

5 Evaluation

In the one neuron classifier, the Adam optimizer outperformed SGD+ and SGD for a learning rate of 0.001 because it achieved less loss than the other two. However, the Adam optimizer oscillates a lot more as it attempts to converge at the global minimum unlike SGD+ which seems to have converged a lot quicker than SGD. On the other hand, with a learning rate of 0.005, SGD and SGD+ have similar performances but the Adam optimizer get significantly worse at each iteration.

In the multi-neuron classifier, the Adam optimizer significantly outperforms the SGD+ and SGD for both learning rates as it gets lower loss at each iteration. For the lower learning rate, both SGD+ and Adam are close to converging to a minimum whereas with the learning rate of 0.005, both SGD+ and Adam keep oscillating and decreasing at each iteration, which suggests its taking longer to reach the minimum of the cost function.