ECE 60146 HW3 Report

Zhengxin Jiang
(jiang839@purdue.edu)

1 Description of SGDplus and Adam

1.1 SGDplus

The optimizer of SGDplus takes the idea of momentum that the optimizer will remember the step size used in the last iteration and decide the step size for current iteration based on the last step size. The formula can be written as follows:

$$v_{t+1} = \mu * v_t + g_{t+1}$$
$$p_{t+1} = p_t - lr * v_{t+1}$$

where μ is the parameter for the momentum scalar with a chosen value.

1.2 Adam

The key idea of Adam is to do joint estimate on a running-average basis the first moment and second moment. In this way both moments are used to determine the step size. The formulas of Adam are shown below:

$$m_{t+1} = \beta_1 * m_t + (1 - \beta_1) * g_{t+1}$$
$$v_{t+1} = \beta_2 * v_t + (1 - \beta_2) * g_{t+1}^2$$
$$\hat{m_{t+1}} = \frac{m_{t+1}}{1 - \beta_1^{t+1}}$$
$$\hat{v_{t+1}} = \frac{v_{t+1}}{1 - \beta_2^{t+1}}$$
$$p_{t+1} = p_t - lr * \frac{\hat{m_{t+1}}}{\sqrt{\hat{v_{t+1}} + \epsilon}}$$

where β_1 and β_2 are two scalars with typical values of $\beta_1 = 0.9$ and $\beta_2 = 0.99$.

2 Plots of The One-neuron Classifier

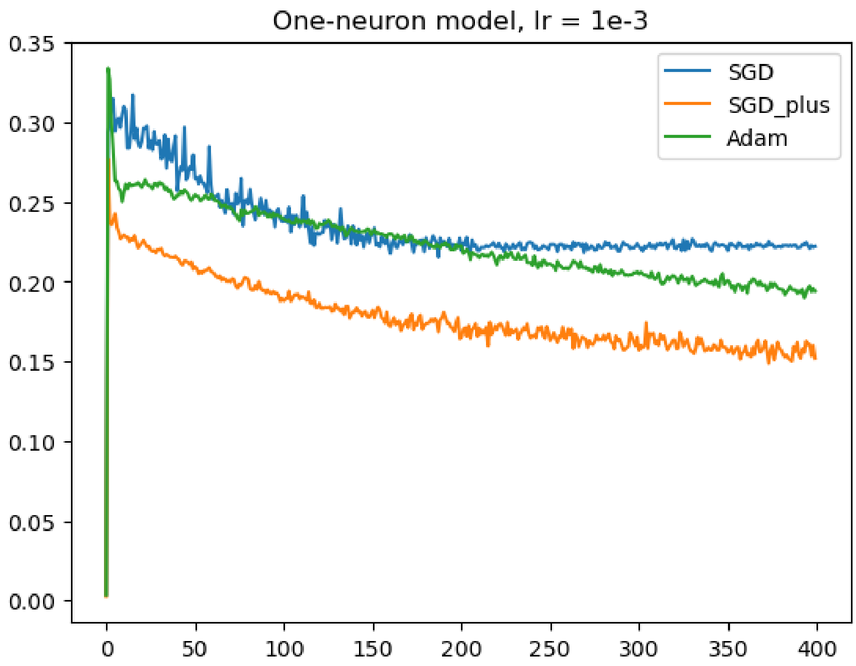

Figure 1: Plot of all optimizers, one-neuron classifier, lr = 1e-3

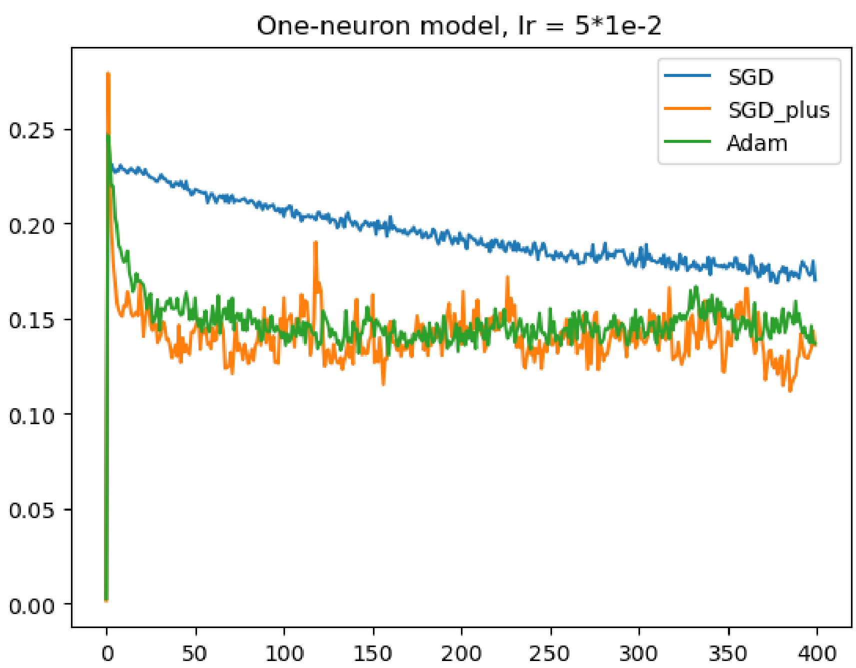

Figure 2: Plot of all optimizers, one-neuron classifier, lr = 5*1e-2

3 Plots of The Multi-neuron Classifier

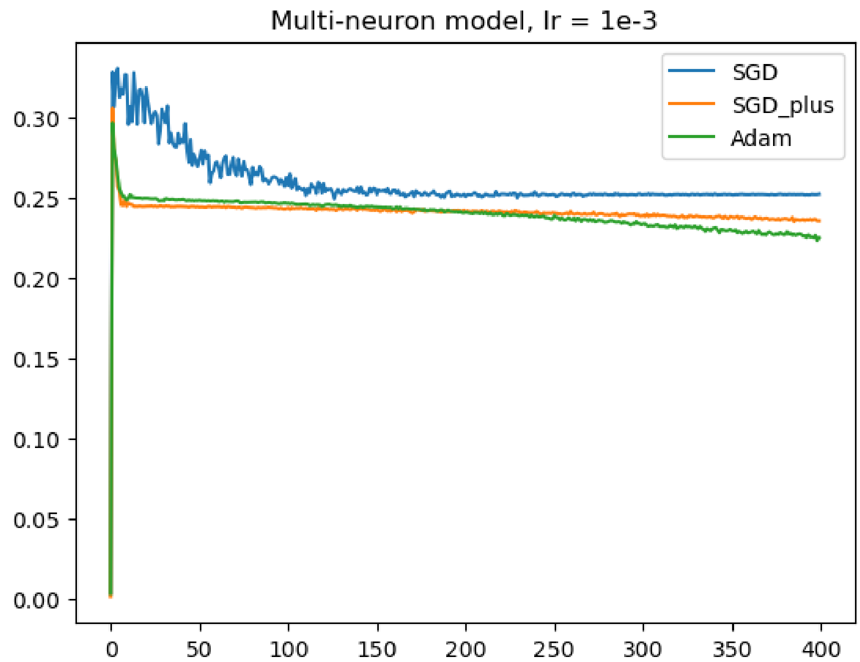

Figure 3: Plot of all optimizers, multi-neuron classifier, lr = 1e-3

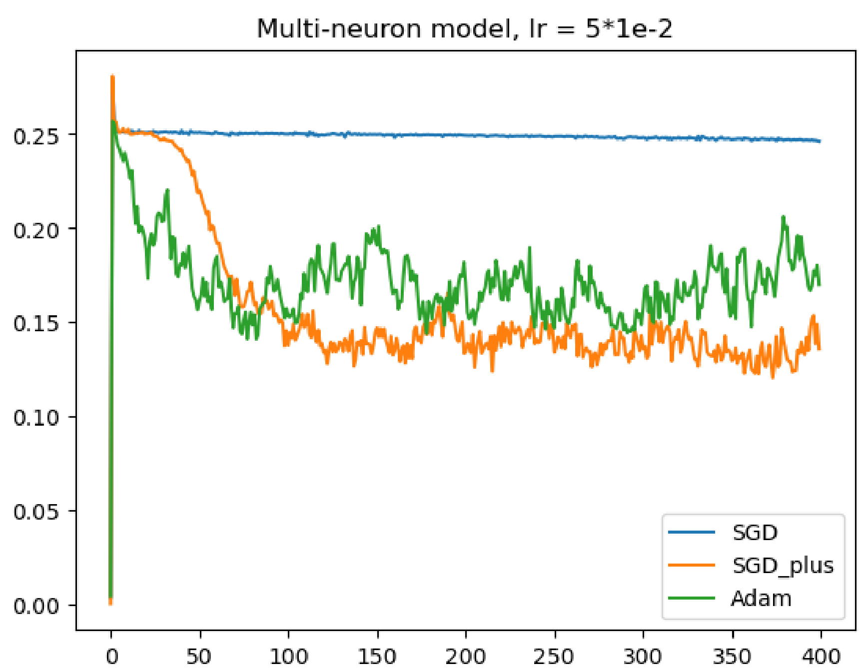

Figure 4: Plot of all optimizers, multi-neuron classifier, lr = 5*1e-2

4 Discussion

For the optimizers of SGDplus and Adam, both of them have better performance on training loss than SGD on both one and multi neuron models. The loss of SGDplus and Adam also goes down much quicker than SGD on the start of training.

5 Source code

```
# ECE60146 HW3
# Zhengxin Jiang
# jiang839

import random
import numpy
import operator
import matplotlib.pyplot as plt
import math

seed = 0
random.seed(seed)
numpy.random.seed(seed)

from ComputationalGraphPrimer import *



# Class of SGD (for returning loss record)
class SGD(ComputationalGraphPrimer):
    #
        ####################################################################################################

    ######################################## one neuron model
        ###########################################
    def run_training_loop_one_neuron_model(self, training_data):
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.
            learnable_params}

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

        class DataLoader:
            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


    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

def run_training_loop_multi_neuron_model(self, training_data):

    class DataLoader:
        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

    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


# Class of SGDplus
class SGD_plus(ComputationalGraphPrimer):

    #
        ##########################################################################################
    ########################################## one neuron model
        ############################################
    def run_training_loop_one_neuron_model(self, training_data, mu=0.99):
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.
            learnable_params}
```

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

######## Added parameters for SGDplus #########
self.mu = mu
self.last_step = np.zeros(len(self.vals_for_learnable_params))
self.last_step_bias = 0
###############################################

class DataLoader:
    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


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


    def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars,
        deriv_sigmoid):

        input_vars = self.independent_vars
        input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]
        param_to_vars_map = {param : var for var, param in input_vars_to_param_map.items()
            }
        vals_for_input_vars_dict = dict(zip(input_vars, list(vals_for_input_vars)))
        vals_for_learnable_params = self.vals_for_learnable_params

        ############## Back propagation of SGDplus #####################
        for i,param in enumerate(self.vals_for_learnable_params):
            ## Calculate the next step in the parameter hyperplane
            step = self.mu * self.last_step[i] + y_error * vals_for_input_vars_dict[
                param_to_vars_map[param]] * deriv_sigmoid
            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += self.learning_rate * step
            self.last_step[i] = step
```

```
    step_bias = self.mu * self.last_step_bias + y_error * deriv_sigmoid
    self.bias += self.learning_rate * step_bias ## Update the bias
    self.last_step_bias = step_bias
    #################################################################

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

####################################### multi neuron model
    ##########################################
def run_training_loop_multi_neuron_model(self, training_data, mu=0.95):

    class DataLoader:
        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

    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.
######## Added parameters for SGDplus #########
self.mu = mu
self.last_step = {}
for back_layer_index in reversed(range(1,self.num_layers)):
    vars_in_layer = self.layer_vars[back_layer_index]
    for j,var in enumerate(vars_in_layer):
        layer_params = self.layer_params[back_layer_index][j]
        for i,param in enumerate(layer_params):
            self.last_step[param] = 0

self.last_step_bias = np.zeros(self.num_layers-1)
################################################

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

    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels):
        # 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]

            ############## Back propagation of SGDplus #####################
            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]

            step = self.mu * self.last_step[param] + gradient_of_loss_for_param *
                deriv_sigmoid_avg[j]
            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += self.learning_rate * step
            self.last_step[param] = step

    step_bias = self.mu * self.last_step_bias[back_layer_index-1] + sum(
        pred_err_backproped_at_layers[back_layer_index]) * sum(deriv_sigmoid_avg)/
        len(deriv_sigmoid_avg)
    self.bias[back_layer_index-1] += self.learning_rate * step_bias ## Update the
        bias
    self.last_step_bias[back_layer_index-1] = step_bias
    #################################################################


# Class of Adam
class Adam(ComputationalGraphPrimer):

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

    ######################################### one neuron model
        ###########################################
    def run_training_loop_one_neuron_model(self, training_data, beta1=0.9, beta2=0.99):
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.
            learnable_params}

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

        ######## Added parameters for Adam #########
        self.beta1 = beta1
        self.beta2 = beta2
        self.last_step_m = np.zeros(len(self.vals_for_learnable_params))
        self.last_step_v = np.zeros(len(self.vals_for_learnable_params))
        self.last_step_bias_m = 0
        self.last_step_bias_v = 0
        ##############################################


        class DataLoader:
            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


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()

        return loss_running_record

    def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars,
        deriv_sigmoid, iter_num):

        input_vars = self.independent_vars
        input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]
        param_to_vars_map = {param : var for var, param in input_vars_to_param_map.items()
            }
        vals_for_input_vars_dict = dict(zip(input_vars, list(vals_for_input_vars)))
        vals_for_learnable_params = self.vals_for_learnable_params

        ############## Back propagation of Adam #####################
        for i,param in enumerate(self.vals_for_learnable_params):
            ## Calculate the next step in the parameter hyperplane
            step_m = self.beta1 * self.last_step_m[i] + (1-self.beta1) * y_error *
                vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoid
            step_v = self.beta2 * self.last_step_v[i] + (1-self.beta2) * (y_error *
                vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoid)**2

            step_m_corrected = step_m / (1-self.beta1**iter_num)
            step_v_corrected = step_v / (1-self.beta2**iter_num)

            step = step_m_corrected / math.sqrt(step_v_corrected + 1e-5)

            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += self.learning_rate * step
            self.last_step_m[i] = step_m
            self.last_step_v[i] = step_v


        step_bias_m = self.beta1 * self.last_step_bias_m + (1-self.beta1) * y_error *
            deriv_sigmoid
        step_bias_v = self.beta2 * self.last_step_bias_v + (1-self.beta2) * (y_error *
            deriv_sigmoid)**2

        step_bias_m_corrected = step_bias_m / (1-self.beta1**iter_num)
        step_bias_v_corrected = step_bias_v / (1-self.beta2**iter_num)

        step_bias = step_bias_m_corrected / math.sqrt(step_bias_v_corrected + 1e-5)

        self.bias += self.learning_rate * step_bias ## Update the bias
```

```
        self.last_step_bias_m = step_bias_m
        self.last_step_bias_v = step_bias_v
        ##################################################################

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

    ######################################## multi neuron model
        ##########################################
    def run_training_loop_multi_neuron_model(self, training_data, beta1=0.9, beta2=0.99):

        class DataLoader:
            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

        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.
######## Added parameters for Adam #########
self.beta1 = beta1
self.beta2 = beta2
self.last_step_m = {}
self.last_step_v = {}
for back_layer_index in reversed(range(1,self.num_layers)):
    vars_in_layer = self.layer_vars[back_layer_index]
    for j,var in enumerate(vars_in_layer):
        layer_params = self.layer_params[back_layer_index][j]
        for i,param in enumerate(layer_params):
            self.last_step_m[param] = 0
            self.last_step_v[param] = 0

self.last_step_bias_m = np.zeros(self.num_layers-1)
self.last_step_bias_v = np.zeros(self.num_layers-1)
##############################################

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

    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels,
        iter_num):
        # 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]

            ############## Back propagation of Adam #####################
            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]

                ## Calculate the next step in the parameter hyperplane
                step_m = self.beta1 * self.last_step_m[param] + (1-self.beta1) *
                    gradient_of_loss_for_param * deriv_sigmoid_avg[j]
                step_v = self.beta2 * self.last_step_v[param] + (1-self.beta2) * (
                    gradient_of_loss_for_param * deriv_sigmoid_avg[j])**2

                step_m_corrected = step_m / (1-self.beta1**iter_num)
                step_v_corrected = step_v / (1-self.beta2**iter_num)

                step = step_m_corrected / math.sqrt(step_v_corrected + 1e-5)

                ## Update the learnable parameters
                self.vals_for_learnable_params[param] += self.learning_rate * step
                self.last_step_m[param] = step_m
                self.last_step_v[param] = step_v


        step_bias_m = self.beta1 * self.last_step_bias_m[back_layer_index-1] + (1-self
            .beta1) * sum(pred_err_backproped_at_layers[back_layer_index]) * sum(
            deriv_sigmoid_avg)/len(deriv_sigmoid_avg)
        step_bias_v = self.beta2 * self.last_step_bias_v[back_layer_index-1] + (1-self
            .beta2) * (sum(pred_err_backproped_at_layers[back_layer_index]) * sum(
            deriv_sigmoid_avg)/len(deriv_sigmoid_avg))**2

        step_bias_m_corrected = step_bias_m / (1-self.beta1**iter_num)
        step_bias_v_corrected = step_bias_v / (1-self.beta2**iter_num)

        step_bias = step_bias_m_corrected / math.sqrt(step_bias_v_corrected + 1e-5)

        self.bias[back_layer_index-1] += self.learning_rate * step_bias ## Update the
            bias
        self.last_step_bias_m[back_layer_index-1] = step_bias_m
        self.last_step_bias_v[back_layer_index-1] = step_bias_v
        ######################################################################


##### Main #####

# One-neuron model

# lr = 1e-3
lr = 5 * 1e-2
```

```
sgd = SGD(
              one_neuron_model = True,
              expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
              output_vars = ['xw'],
              dataset_size = 5000,
              learning_rate = lr,
#             learning_rate = 5 * 1e-2,
              training_iterations = 40000,
              batch_size = 8,
              display_loss_how_often = 100,
              debug = True,
      )

sgd.parse_expressions()
training_data = sgd.gen_training_data()
sgd_loss = sgd.run_training_loop_one_neuron_model( training_data )

sp = SGD_plus(
              one_neuron_model = True,
              expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
              output_vars = ['xw'],
              dataset_size = 5000,
              learning_rate = lr,
#             learning_rate = 5 * 1e-2,
              training_iterations = 40000,
              batch_size = 8,
              display_loss_how_often = 100,
              debug = True,
      )


sp.parse_expressions()
training_data = sp.gen_training_data()
sp_loss = sp.run_training_loop_one_neuron_model( training_data )

adam = Adam(
              one_neuron_model = True,
              expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
              output_vars = ['xw'],
              dataset_size = 5000,
              learning_rate = lr,
#             learning_rate = 5 * 1e-2,
              training_iterations = 40000,
              batch_size = 8,
              display_loss_how_often = 100,
              debug = True,
      )


adam.parse_expressions()
training_data = adam.gen_training_data()
adam_loss = adam.run_training_loop_one_neuron_model( training_data )
```

```
# Multi-neuron model

lr = 1e-3
# lr = 5 * 1e-2

sgd = SGD(
             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,
# learning_rate = 5 * 1e-2,
             training_iterations = 40000,
             batch_size = 8,
             display_loss_how_often = 100,
             debug = True,
     )

sgd.parse_multi_layer_expressions()
training_data = sgd.gen_training_data()
sgd_loss = sgd.run_training_loop_multi_neuron_model( training_data )

sp = SGD_plus(
             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,
# learning_rate = 5 * 1e-2,
             training_iterations = 40000,
             batch_size = 8,
             display_loss_how_often = 100,
             debug = True,
     )


sp.parse_multi_layer_expressions()
training_data = sp.gen_training_data()
sp_loss = sp.run_training_loop_multi_neuron_model( training_data )

adam = Adam(
             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,
# learning_rate = 5 * 1e-2,
              training_iterations = 40000,
              batch_size = 8,
              display_loss_how_often = 100,
              debug = True,
      )


adam.parse_multi_layer_expressions()
training_data = adam.gen_training_data()
adam_loss = adam.run_training_loop_multi_neuron_model( training_data )


# Plot
plt.figure()
plt.plot(sgd_loss, label = 'SGD')
plt.plot(sp_loss, label = 'SGD_plus')
plt.plot(adam_loss, label = 'Adam')
plt.legend()
plt.title('Multi-neuron␣model,␣lr␣=␣1e-3')
plt.show()
```