Contents

ECE60146 DL HW3

Akshita Kamsali, akamsali@purdue.edu

February 6, 2023

Note: bold lower case letters indicate vectors and bold upper case letters indicate matrices

1 Overview

Develop a deeper understanding of popular step optimizations used in practice namely SGD+ and AdaM.

1.1 SGD+

SGD+ algorithm is a slight modification on SGD algorithm, where we remember previous step size and use a fraction(μ) to calculate the current step size along with the current gradient. This modifies our update equation for every parameter

from:

$$p_{t+1} = p_t - \eta g_{t+1}$$

to:

$$v_{t+1} = \mu v_t + g_{t+1}$$

$$p_{t+1} = p_t - \eta v_{t+1}$$

where $\mu \in [0, 1]$ is the momentum parameter, η = learning rate, g_{t+1} = gradient of the loss function with respect to the learnable parameters

p_t = learnable parameters

v_0 = all 0's

This computation is done for as many iterations (epochs) as the user decides, t is the iteration number.

1.2 AdaM

AdaM is also an extension to vanilla SGD, where step update is modified

from:

$$p_{t+1} = p_t - \eta g_{t+1}$$

to:

$$m_{t+1} = \beta_1 * m_t + (1 - \beta_1) * g_{t+1}$$

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

$$\hat{m}_{t+1} = \frac{m_t}{(1 - \beta_1^t)}$$

$$\hat{v}_{t+1} = v_t/(1 - \beta_2^t)$$

$$p_{t+1} = p_t - \eta(\hat{m}_{t+1}/\sqrt{\hat{v}_{t+1}})$$

where m and v are the first and second momentum parameters
η = learning rate
g_{t+1} = gradient of the loss function with respect to the learnable parameters
p_t = learnable parameters
m_0 = all 0's
v_0 = all 0's

In AdaM, we take into both first and second moments, and control decay rate for $k^t h$ iteration as an inverse exponential relation with the betas. This also significantly affects the step size. Another key point is, while implementing, we add a small value ϵ, usually order 1e-7 to denominator of final step size to avoid `NaN` in square root calculation.

2 Results

Figures 1 and 2 show the model architecture used for training. Figures 3 to 5 show training loss for one neuron model for SGD, SGD+(two moment parameters, $\mu = 0.5, 0.9$) and AdaM ($\beta_1 = 0.9, \beta_2 = 0.99$) for different learning rates (lr = 0.001, 0.003, 0.005)

Figures 6 to 8 show training loss for multi neuron model for SGD, SGD+(two moment parameters, $\mu = 0.5, 0.9$) and AdaM ($\beta_1 = 0.9, \beta_2 = 0.99$) for different learning rates (lr = 0.001, 0.003, 0.005)

Number of iterations are 40K, with avergaing performed at every 100 steps as done in Avi's default implementation.

3 Discussion

SGD+ decreases the training loss in most cases for both one and multi-neuron models. However, it seems to be highly sensitive to the parameter μ. Also, another observation is increasing μ need not decrease the loss/improve the performance as we see figure 3.

We observe a significant increase in performance, i.e., a decrease in training loss with AdaM optimizer in both one and multi neuron case. We kept the AdaM parameters $\beta_1 = 0.9$ and $\beta_2 = 0.99$ fixed through various learning rates iterations for both the models as it is a standard practice. Given, rest of the parameters are constant, the learning rate seems to have a significant influence on AdaM's performance.

Increasing learning rate from 1e-3 to 5e-3 showed very linear dependence for SGD and SGD+. However, AdaM seems to fluctuate/oscillate upon increasing LR as seen in figures 5, 7 and 8.

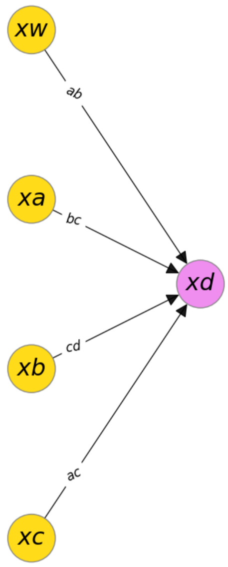

Figure 1: One Neuron Model

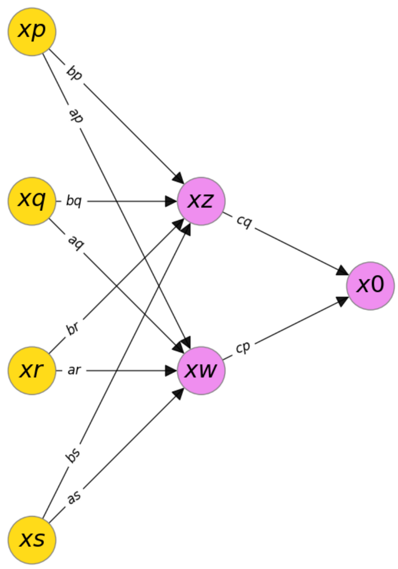

Figure 2: MultiNeuron Model

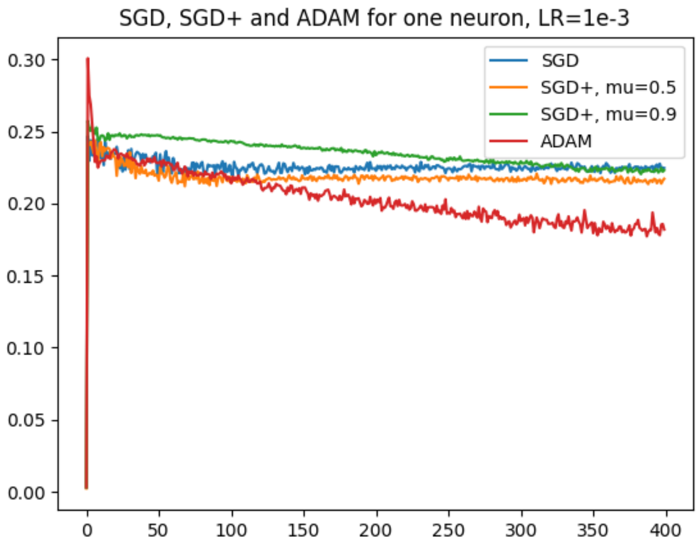

Figure 3: One Neuron, LR=1e-3

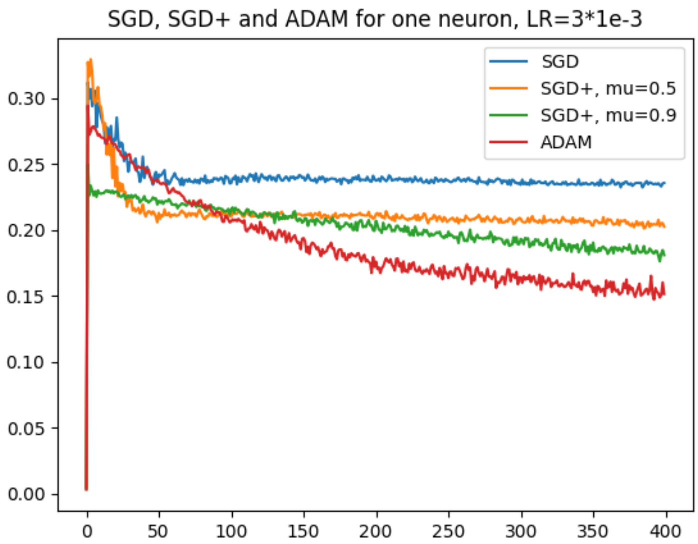

Figure 4: One Neuron, LR=3*1e-3

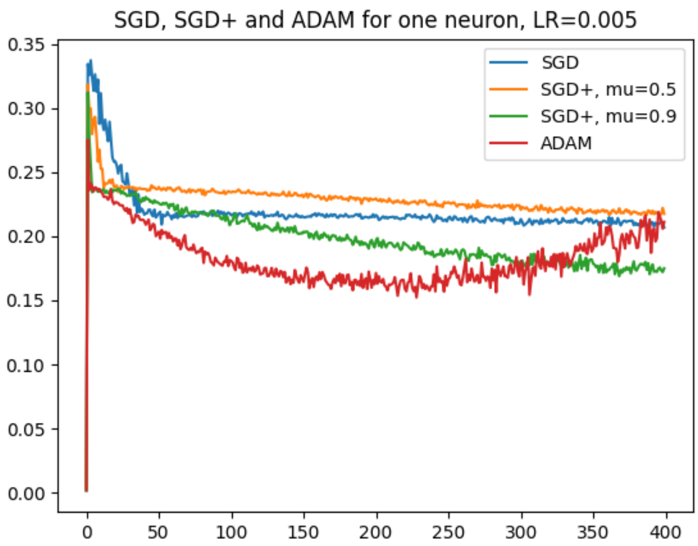

Figure 5: One Neuron, LR=5*1e-3

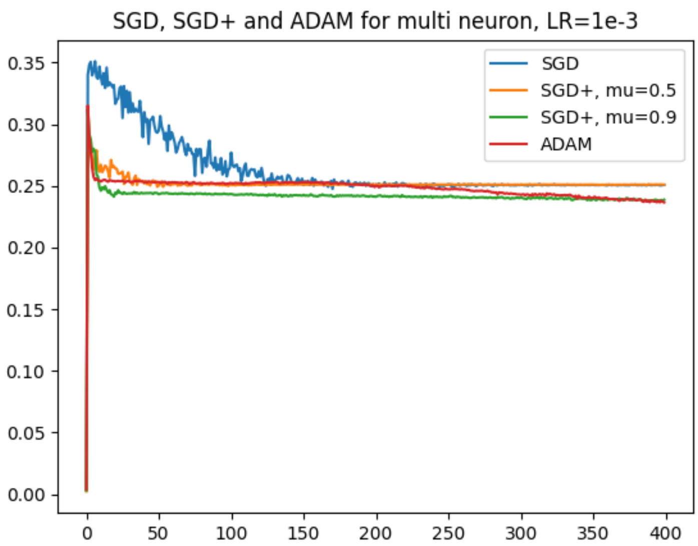

Figure 6: Multi Neuron, LR=1e-3

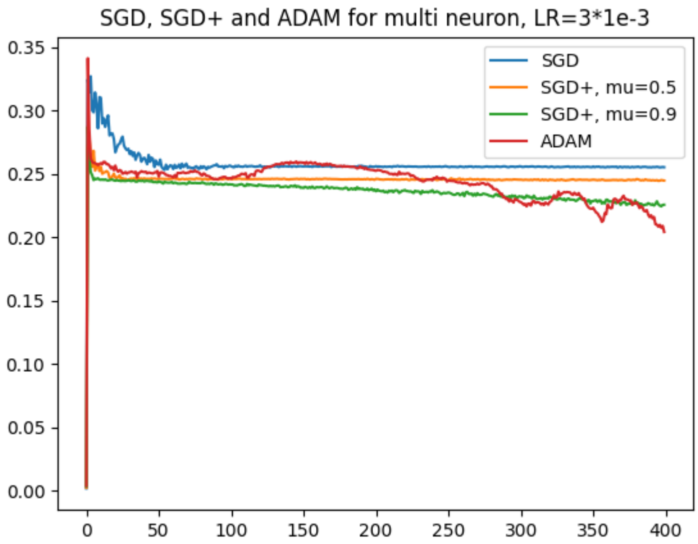

Figure 7: Multi Neuron, LR=3*1e-3

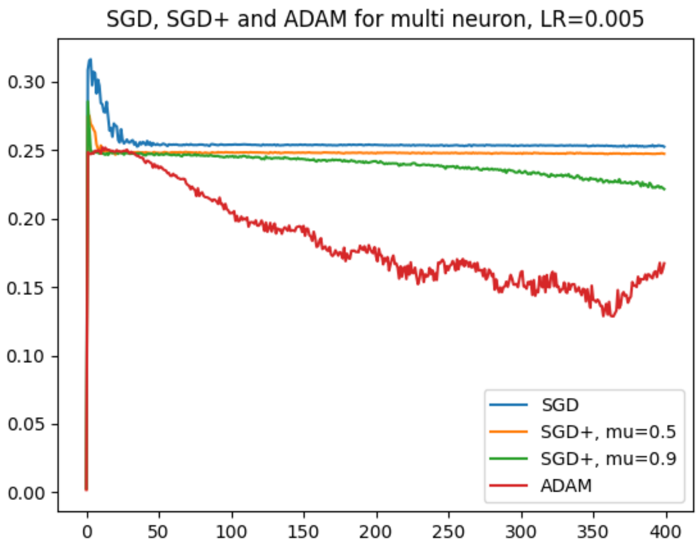

Figure 8: Multi Neuron, LR=5*1e-3

In all the runs, SGD adn SGD+ pretty much flattens out half way through, i.e., 20K iterations, however, AdaM shows no such pattern.

At the end of 40K iterations, SGD and SGD+ for both single and mutineuron shows slight increase in performance for all learning rates. AdaM's performance increases and then decreased with increased lr in single neuron model. With Multi neuron, we have increased performance but also has more oscillations as pointed out earlier.

4 Code

4.1 SGD and SGD+ for one neuron

```
from ComputationalGraphPrimer import ComputationalGraphPrimer
import random
import numpy as np
import operator


class SGDplus(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # self.momentum = mu

    def backprop_and_update_params_one_neuron(self, y_error, vals_for_input_vars, deriv_sigmoid):
        """
        @akamsali:
        modified from the backprop_and_update_params_one_neuron_model method in the ComputationalGraphPrimer class
        in the ComputationalGraphPrimer.py file.

        The modification is to use the SGD+ algorithm to update the step size

        from:
        p_{t+1} = p_t - \eta g_{t+1}

        to:
        v_{t+1} = \mu v_t + g_{t+1}
        p_{t+1} = p_t - \eta v_{t+1}

        where \mu is the momentum parameter
        \eta = learning rate
        g_{t+1} = gradient of the loss function with respect to the learnable parameters (deriv_sigmoid)
        p_t = learnable parameters
        v_0 = all 0's
        """

        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):
            ## @akamsali: Calculate the next step in the parameter hyperplane

            self.v[i] = self.momentum * self.v[i] + (
                y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
            )
            step = self.learning_rate * self.v[i]
```

```python
            ##  @akamsali: Update the learnable parameters
            self.vals_for_learnable_params[param] += step

        ##  @akamsali: Update the bias
        self.v_bias = self.learning_rate * y_error * deriv_sigmoid + (
            self.momentum * self.v_bias
        )
        self.bias += self.v_bias

    def train_one_neuron(self, training_data, mu=None):
        """
        @akamsali: Taking Avi's code as is for training a one neuron model.  The only modification
        is to return the loss running record so that we can plot it later.
        """

        """
        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).
        """
        # @akamsali: initialise moments
        # zero implies SGD, non-zero is SGD+
        if mu is None:
            self.momentum = 0
        else:
            self.momentum = mu
        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:
            """
            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

    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
    )
    self.v = [0] * (
        len(self.vals_for_learnable_params)
    )  ## Initialize the velocity vector to all 0's
    self.v_bias = 0
    ##    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(
                y_error_avg, data_tuple_avg, deriv_sigmoid_avg
            ) ## BACKPROP loss
        # plt.figure()
        return loss_running_record
        # plt.show()
```

code/sgdp.py

4.2 SGD and SGD+ for multi neuron

```
from ComputationalGraphPrimer import ComputationalGraphPrimer
import operator
import random
import numpy as np
from tqdm import tqdm

class SGDPlusMultiNeuron(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)

    def backprop_and_update_params_multineuron(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]
            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]
                    # @akamsali: update the velocity parameter and use
                    self.v[param] = gradient_of_loss_for_param * deriv_sigmoid_avg[j] \
                                        + (self.momentum * self.v[param])

                    step = self.learning_rate * self.v[param]

                    self.vals_for_learnable_params[param] += step
            #  @akamsali:  update the bias parameters
```

```
                self.v_bias[back_layer_index -1] = (self.learning_rate * sum(pred_err_backproped_at_layers[back_layer_index]) \
                                                                  * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)) \
                                                                  + (self.momentum * self.v_bias[back_layer_index -1])
                self.bias[back_layer_index-1] += self.v_bias[back_layer_index -1]
    #######################################################################################################
    # @akamsali: modified func call name and take in momentum value \mu
    def train_multineuron(self, training_data, mu=None):


        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


        """
        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}
        # @akamsali: initialise v as 0 for learnable parameters
        self.v = {param: 0 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.
        # @akamsali: initialise bias velocity to zero
        self.v_bias = [0] * (self.num_layers-1)
        # @akamsali: initialise moments
        # zero implies SGD, non-zero is SGD+
        if mu is None:
            self.momentum = 0
        else:
            self.momentum = mu

        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 tqdm(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))
            # @akamsali: change to modified backprop
            self.backprop_and_update_params_multineuron(y_error_avg, class_labels)
     ## BACKPROP loss

        return loss_running_record
```

code/sgd_mn.py

4.3 AdaM for one neuron

```python
from ComputationalGraphPrimer import ComputationalGraphPrimer
import random
import numpy as np
import operator

class myADAM(ComputationalGraphPrimer):
    '''
    @akamsali:
    modified from the backprop_and_update_params_one_neuron_model method in the
    ComputationalGraphPrimer class
    in the ComputationalGraphPrimer.py file.

    The modification is to use the SGD+ algorithm to update the step size

    from:

    $p_{t+1} = p_t - \eta g_{t+1}$

    to:

    $m_{t+1} = \beta_1 * m_t + (1-\beta_1) * g_{t+1}$

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

    $p_{t+1} = p_t - \eta (\hat{m}_{t+1} / \sqrt{\hat{v}_{t+1}})$

    $\hat{m}_{t+1} = m_{t}/(1 - \beta_1^t)$

    $\hat{v}_{t+1} = v_{t}/(1 - \beta_2^t)$


    where m and v are the first and second momentum parameters

    \eta = learning rate

    $g_{t+1}$ = gradient of the loss function with respect to the learnable
    parameters (input val * deriv_sigmoid)

    $p_t$ = learnable parameters

    $m_0$ = all 0's

    $v_0$ = all 0's
    '''
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def backprop_and_update_parama_bias(self, y_error, vals_for_input_vars,
    deriv_sigmoid):
        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):
            ## @akamsali: Calculate the next step in the parameter hyperplane
            g_t = y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
            m_val = self.beta_1 * self.m[param] + (1-self.beta_1) * g_t
            v_val = self.beta_2 * self.v[param] + (1-self.beta_2) * (g_t**2)
            m_hat = m_val / (1 - self.beta_1 ** self.time[param] )
```

```
            v_hat = v_val / (1 - self.beta_2 ** self.time[param] )

            step = self.learning_rate * m_hat / np.sqrt(v_hat + self.epsilon)
            ##  @akamsali: Update the learnable parameters
            self.vals_for_learnable_params[param] += step
            self.m[param] = m_val
            self.v[param] = v_val
            self.time[param] += 1

        ## @akamsali: Update the bias
        m_bias_val = self.beta_1 * self.m_bias + (1 - self.beta_1) * (y_error * deriv_sigmoid)
        v_bias_val = self.beta_2 * self.v_bias + (1 - self.beta_2) * ((y_error * deriv_sigmoid)**2)

        m_bias_hat = m_bias_val / (1 - (self.beta_1 ** self.time_bias))
        v_bias_hat = v_bias_val / (1 - (self.beta_2 ** self.time_bias))
        bias_step = self.learning_rate * (m_bias_hat / np.sqrt(v_bias_hat + 1e-7))
        self.time_bias += 1
        self.bias += bias_step
        self.m_bias = m_bias_val
        self.v_bias = v_bias_val


    def train(self, training_data, beta_1=0.9, beta_2=0.99, epsilon=1e-7):
        """
        @akamsali: Taking Avi's code as is for training a one neuron model.  The only modification
        is to return the loss running record so that we can plot it later.
        """

        """
        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
        ) ## Adding the bias improves class discrimination.
        ##   We initialize it to a random number.
        # @akamsali: initialise parameters and bias parameter moments
        self.beta_1 = beta_1
        self.beta_2 = beta_2
        self.epsilon = epsilon
        self.m = {param: 0 for param in self.learnable_params}
        self.v = {param: 0 for param in self.learnable_params}
        self.time = {param: 1 for param in self.learnable_params}
        self.time_bias = 1
        self.m_bias = 0
        self.v_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
```

```python
            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_parama_bias(
                y_error_avg, data_tuple_avg, deriv_sigmoid_avg
            )  ## BACKPROP loss
        # plt.figure()
        return loss_running_record
        # plt.show()
```

code/ADAM_on.py

4.4 AdaM for multi neuron

```python
from ComputationalGraphPrimer import ComputationalGraphPrimer
import operator
import random
import numpy as np
from tqdm import tqdm

class myADAMMultiNeuron(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs) -> None:
        super().__init__(*args, **kwargs)

    def backprop_and_update_params_multineuron(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]
            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):

                    # @akamsali: update the velocity parameter and use
                    g_t = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j] * deriv_sigmoid_avg[j]

                    m_val = self.beta_1 * self.m[param] + (1-self.beta_1) * g_t
                    v_val = self.beta_2 * self.v[param] + (1-self.beta_2) * (g_t**2)
                    m_hat = m_val / (1 - self.beta_1 ** self.time[param] )
                    v_hat = v_val / (1 - self.beta_2 ** self.time[param] )

                    # @akamsali: updatethe learnable parameters
                    step = self.learning_rate * m_hat / np.sqrt(v_hat + self.epsilon)
                    self.vals_for_learnable_params[param] += step
                    # s@akamsali: tore the current values of first and second moment parameters
                    # for next iteration of training
                    self.m[param] = m_val
```

```
                        self.v[param] = v_val
                        self.time[param] += 1 # update time step

                ## @akamsali: Update the bias
                m_bias_val = self.beta_1 * self.m_bias[back_layer_index -1] + \
                            (1 - self.beta_1) * (np.sum(pred_err_backproped_at_layers[back_layer_index]) * np.mean(deriv_sigmoid_avg))
                v_bias_val = self.beta_2 * self.v_bias[back_layer_index -1] + \
                            (1 - self.beta_2) * (np.sum(pred_err_backproped_at_layers[back_layer_index]) * np.mean(deriv_sigmoid_avg)**2)

                m_bias_hat = m_bias_val / (1 - (self.beta_1 ** self.time_bias[back_layer_index -1]))
                v_bias_hat = v_bias_val / (1 - (self.beta_2 ** self.time_bias[back_layer_index -1]))

                ##  @akamsali:  Update the bias parameters
                bias_step = self.learning_rate * (m_bias_hat / np.sqrt(np.abs(v_bias_hat) + self.epsilon))
                # print(f"v_bias_hat: {v_bias_hat}")
                # np.sqrt(v_bias_hat + 1e-7)
                self.bias += bias_step

                #  @akamsali: store the current values of first and second moment parameters
                # for next iteration of training

                self.m_bias[back_layer_index -1] = m_bias_val
                self.v_bias[back_layer_index -1] = v_bias_val
                self.time_bias[back_layer_index -1] += 1 # update time step


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

    # @akamsali: modified func call name and take in momentum value \mu
    def train_multineuron(self, training_data, beta_1=0.9, beta_2=0.99, epsilon=1e-7):


        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


    """
    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.
    # @akamsali: set hyperparameters
    self.beta_1 = beta_1
    self.beta_2 = beta_2
    self.epsilon = epsilon
    # @akamsali: initialise learnable parameter moments
    self.m = {param: 0 for param in self.learnable_params}
    self.v = {param: 0 for param in self.learnable_params}
    self.time = {param: 1 for param in self.learnable_params}

    # @akamsali: initialise bias parameter moments
    self.time_bias = [1]*(self.num_layers-1)
    self.m_bias = [0]*(self.num_layers-1)
    self.v_bias = [0]*(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.
```

```python
        for i in tqdm(range(self.training_iterations)):
            data = data_loader.getbatch()
            data_tuples = data[0]
            # print(data_tuples)
            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))
            # @akamsali: change to modified backprop
            self.backprop_and_update_params_multineuron(y_error_avg, class_labels)      ## BACKPROP loss

        return loss_running_record
```

code/ADAM_multi.py

4.5 Training code

```python
from sgdp import SGDplus
from sgd_mn import SGDPlusMultiNeuron
from ADAM_on import myADAM
from ADAM_multi import myADAMMultiNeuron

import matplotlib.pyplot as plt

lr = 5*1e-3
# initialise SGD and SGD+ for one neuron
sgd_on = SGDplus(
                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,
        )

```

```python

sgd_on.parse_expressions()
training_data = sgd_on.gen_training_data()
# SGD
sgd_on_loss_0 = sgd_on.train_one_neuron(training_data)
# SGD+ with mu = 0.5, 0.9
sgd_on_loss_5 = sgd_on.train_one_neuron(training_data, mu=0.5)
sgd_on_loss_9 = sgd_on.train_one_neuron(training_data, mu=0.9)

# initialise SGD and SGD+ for multi neuron
sgd_mn = SGDPlusMultiNeuron(
                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_mn.parse_multi_layer_expressions()

# SGD
sgd_mn_loss_0 = sgd_mn.train_multineuron(training_data)
# SGD+ with mu = 0.5, 0.9
sgd_mn_loss_5 = sgd_mn.train_multineuron(training_data, mu=0.5)
sgd_mn_loss_9 = sgd_mn.train_multineuron(training_data, mu=0.9)


# initialise ADAM for one neuron
adam_on = myADAM(
                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_on.parse_expressions()
# training_data = cgp.gen_training_data()

# loss_0 = cgp.train(training_data)
loss = adam_on.train(training_data)
# plt.plot(loss)

# loss_9 = cgp.train(training_data, mu=0.9)
# plt.plot(loss_0)
```

```python
# plt.plot(loss_9)

# initialise ADAM for multi neuron
adam_mn = myADAMMultiNeuron(
                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_mn.parse_multi_layer_expressions()

training_data = adam_mn.gen_training_data()
adam_mn_loss = adam_mn.train_multineuron(training_data)


# PLOT one neuron model losses
plt.plot(sgd_on_loss_0, label='SGD')
plt.plot(sgd_on_loss_5, label='SGD+, mu=0.5')
plt.plot(sgd_on_loss_9, label='SGD+, mu=0.9')
plt.plot(loss, label='ADAM')
plt.legend()
plt.title(f'SGD, SGD+ and ADAM for one neuron, LR={lr}')
plt.savefig(f"one_neuron_5")

# PLOT multi neuron model losses

plt.plot(sgd_mn_loss_0, label='SGD')
plt.plot(sgd_mn_loss_5, label='SGD+, mu=0.5')
plt.plot(sgd_mn_loss_9, label='SGD+, mu=0.9')
plt.plot(adam_mn_loss, label='ADAM')
plt.legend()
plt.title(f'SGD, SGD+ and ADAM for multi neuron, LR={lr}')
plt.savefig(f"multi_neuron_5")
```

code/main.py