

Contents

1	Overview	2
1.1	SGD+	2
1.2	AdaM	2
2	Results	3
3	Discussion	3
4	Code	8
4.1	SGD and SGD+ for one neuron	8
4.2	SGD and SGD+ for multi neuron	11
4.3	AdaM for one neuron	15
4.4	AdaM for multi neuron	18
4.5	Training code	22

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^th 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.

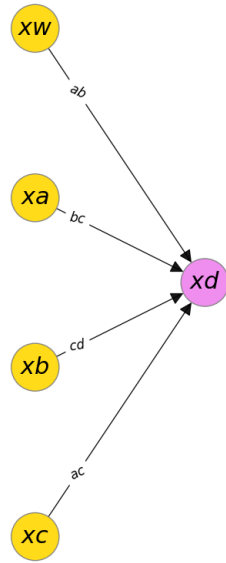


Figure 1: One Neuron Model

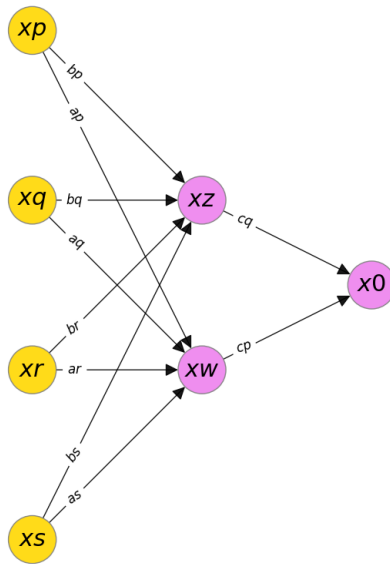


Figure 2: MultiNeuron Model

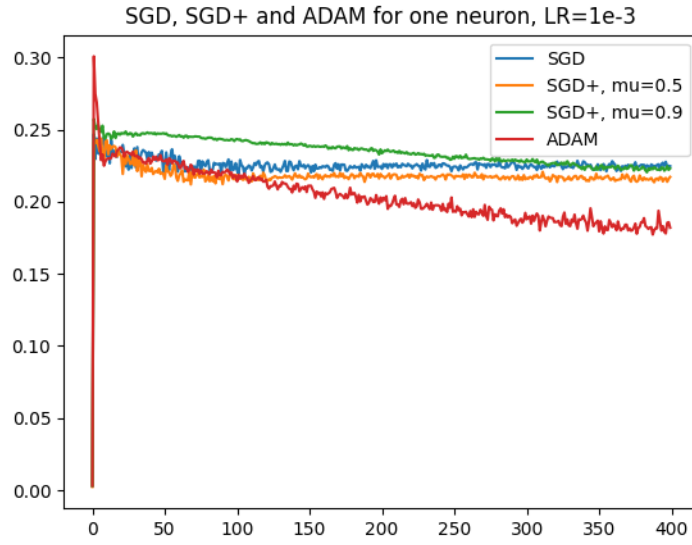


Figure 3: One Neuron, LR=1e-3

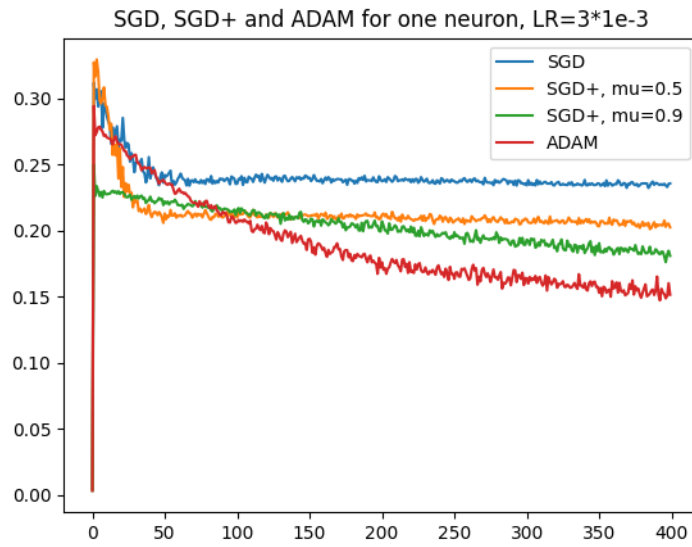


Figure 4: One Neuron, LR=3*1e-3

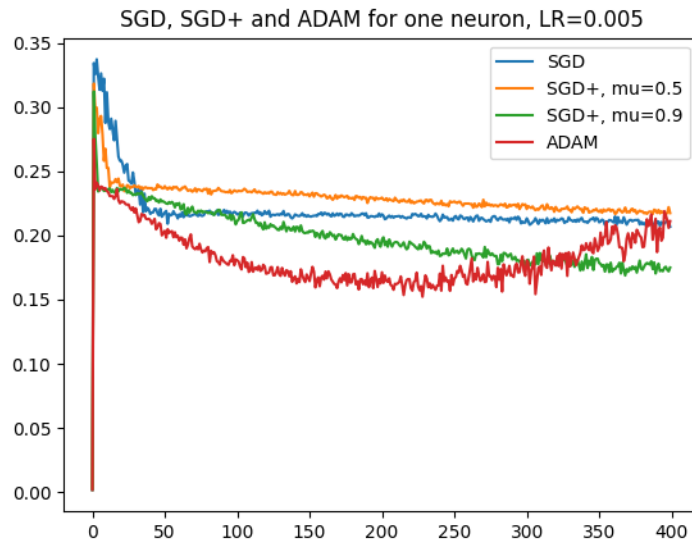


Figure 5: One Neuron, LR=5*1e-3

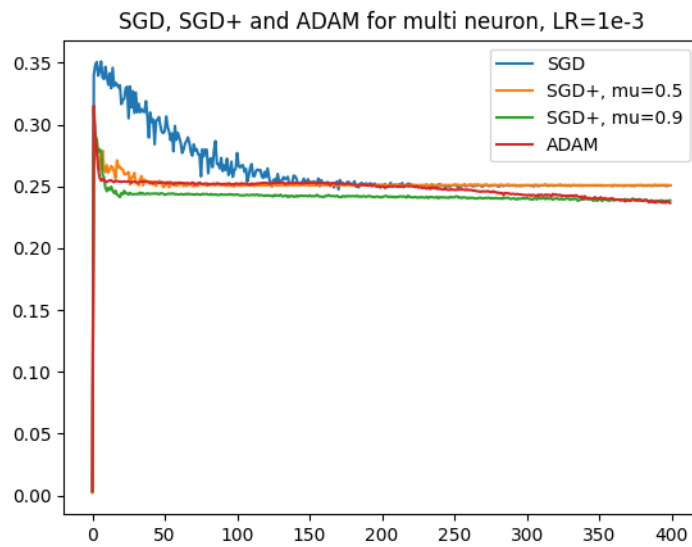


Figure 6: Multi Neuron, LR=1e-3

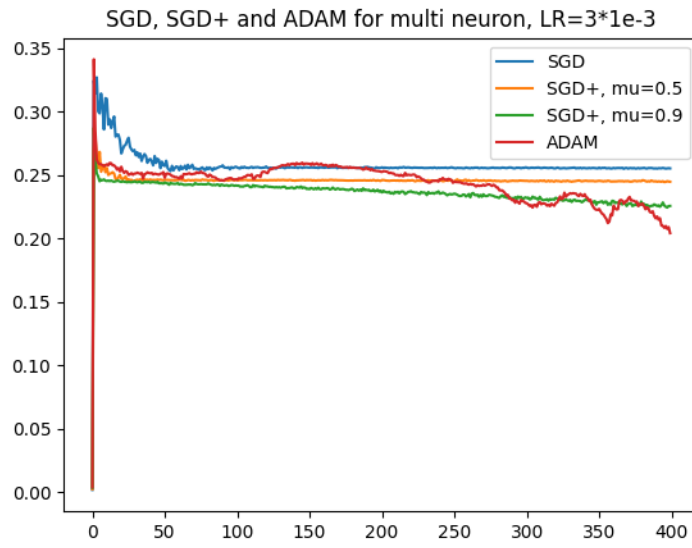


Figure 7: Multi Neuron, LR=3*1e-3

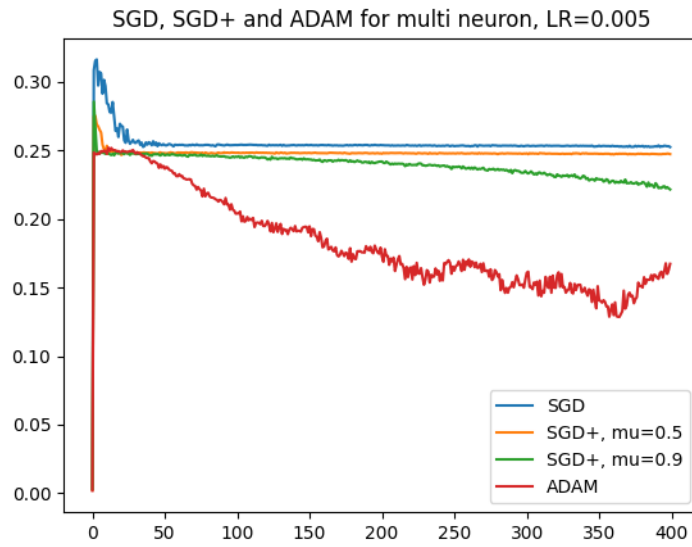


Figure 8: Multi Neuron, LR=5*1e-3

In all the runs, SGD and 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

```

1 from ComputationalGraphPrimer import ComputationalGraphPrimer
2 import random
3 import numpy as np
4 import operator
5
6
7 class SGDplus(ComputationalGraphPrimer):
8     def __init__(self, *args, **kwargs):
9         super().__init__(*args, **kwargs)
10        # self.momentum = mu
11
12    def backprop_and_update_params_one_neuron(self, y_error, vals_for_input_vars,
13        deriv_sigmoid):
14        """
15        @akamsali:
16        modified from the backprop_and_update_params_one_neuron_model method in the
17        ComputationalGraphPrimer class
18        in the ComputationalGraphPrimer.py file.
19
20        The modification is to use the SGD+ algorithm to update the step size
21
22        from:
23         $p_{t+1} = p_t - \eta g_{t+1}$ 
24
25        to:
26         $v_{t+1} = \mu v_t + g_{t+1}$ 
27         $p_{t+1} = p_t - \eta v_{t+1}$ 
28
29        where  $\mu$  is the momentum parameter
30         $\eta$  = learning rate
31         $g_{t+1}$  = gradient of the loss function with respect to the learnable
32        parameters (deriv_sigmoid)
33         $p_t$  = learnable parameters
34         $v_0$  = all 0's
35        """
36
37    input_vars = self.independent_vars
38    vals_for_input_vars_dict = dict(zip(input_vars, list(vals_for_input_vars)))
39    vals_for_learnable_params = self.vals_for_learnable_params
40    for i, param in enumerate(self.vals_for_learnable_params):
41        ## @akamsali: Calculate the next step in the parameter hyperplane
42
43        self.v[i] = self.momentum * self.v[i] + (
44            y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
45        )
46        step = self.learning_rate * self.v[i]

```



```

44         ## @akamsali: Update the learnable parameters
45         self.vals_for_learnable_params[param] += step
46
47     ## @akamsali: Update the bias
48     self.v_bias = self.learning_rate * y_error * deriv_sigmoid + (
49         self.momentum * self.v_bias
50     )
51     self.bias += self.v_bias
52
53 def train_one_neuron(self, training_data, mu=None):
54     """
55     @akamsali: Taking Avi's code as is for training a one neuron model. The only
56     modification
57     is to return the loss running record so that we can plot it later.
58     """
59
60     """
61     The training loop must first initialize the learnable parameters. Remember,
62     these are the
63     symbolic names in your input expressions for the neural layer that do not
64     begin with the
65     letter 'x'. In this case, we are initializing with random numbers from a
66     uniform distribution
67     over the interval (0,1).
68     """
69     # @akamsali: initialise moments
70     # zero implies SGD, non-zero is SGD+
71     if mu is None:
72         self.momentum = 0
73     else:
74         self.momentum = mu
75     self.vals_for_learnable_params = {
76         param: random.uniform(0, 1) for param in self.learnable_params
77     }
78
79     self.bias = random.uniform(
80         0, 1
81     ) ## Adding the bias improves class discrimination.
82     ## We initialize it to a random number.
83
84     class DataLoader:
85         """
86         To understand the logic of the dataloader, it would help if you first
87         understand how
88         the training dataset is created. Search for the following function in
89         this file:
90
91             gen_training_data(self)
92
93         As you will see in the implementation code for this method, the training
94         dataset
95         consists of a Python dict with two keys, 0 and 1, the former points to a
96         list of
97         all Class 0 samples and the latter to a list of all Class 1 samples. In
98         each list,
99         the data samples are drawn from a multi-dimensional Gaussian distribution
100         . The two
101         classes have different means and variances. The dimensionality of each
102         data sample
103         is set by the number of nodes in the input layer of the neural network.

```

```

93
94     The data loader's job is to construct a batch of samples drawn randomly
95     from the two
96     lists mentioned above. And it must also associate the class label with
97     each sample
98     separately.
99     """
100
101     def __init__(self, training_data, batch_size):
102         self.training_data = training_data
103         self.batch_size = batch_size
104         self.class_0_samples = [
105             (item, 0) for item in self.training_data[0]
106         ] ## Associate label 0 with each sample
107         self.class_1_samples = [
108             (item, 1) for item in self.training_data[1]
109         ] ## Associate label 1 with each sample
110
111     def __len__(self):
112         return len(self.training_data[0]) + len(self.training_data[1])
113
114     def _getitem(self):
115         cointoss = random.choice(
116             [0, 1]
117         ) ## When a batch is created by getbatch(), we want the
118         ## samples to be chosen randomly from the two lists
119         if cointoss == 0:
120             return random.choice(self.class_0_samples)
121         else:
122             return random.choice(self.class_1_samples)
123
124     def getbatch(self):
125         batch_data, batch_labels = (
126             [],
127             [],
128         ) ## First list for samples, the second for labels
129         maxval = 0.0 ## For approximate batch data normalization
130         for _ in range(self.batch_size):
131             item = self._getitem()
132             if np.max(item[0]) > maxval:
133                 maxval = np.max(item[0])
134             batch_data.append(item[0])
135             batch_labels.append(item[1])
136         batch_data = [
137             item / maxval for item in batch_data
138         ] ## Normalize batch data
139         batch = [batch_data, batch_labels]
140         return batch
141
142     data_loader = DataLoader(training_data, batch_size=self.batch_size)
143     loss_running_record = []
144     i = 0
145     avg_loss_over_iterations = (
146         0.0 ## Average the loss over iterations for printing out
147     )
148     self.v = [0] * (
149         len(self.vals_for_learnable_params)
150     ) ## Initialize the velocity vector to all 0's
151     self.v.bias = 0
152     ## every N iterations during the training loop.

```

```

151     for i in range(self.training_iterations):
152         data = data_loader.getbatch()
153         data_tuples = data[0]
154         class_labels = data[1]
155         y_preds, deriv_sigmoids = self.forward_prop_one_neuron_model(
156             data_tuples
157         ) ## FORWARD PROP of data
158         loss = sum(
159             [
160                 (abs(class_labels[i] - y_preds[i])) ** 2
161                 for i in range(len(class_labels))
162             ]
163         ) ## Find loss
164         loss_avg = loss / float(len(class_labels)) ## Average the loss over
batch
165         avg_loss_over_iterations += loss_avg
166         if i % (self.display_loss_how_often) == 0:
167             avg_loss_over_iterations /= self.display_loss_how_often
168             loss_running_record.append(avg_loss_over_iterations)
169             # print("[iter=%d] loss = %.4f" % (i+1, avg_loss_over_iterations))
170             ## Display average loss
171             avg_loss_over_iterations = 0.0 ## Re-initialize avg loss
172             y_errors = list(map(operator.sub, class_labels, y_preds))
173             y_error_avg = sum(y_errors) / float(len(class_labels))
174             deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
175             data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
176             data_tuple_avg = list(
177                 map(
178                     operator.truediv,
179                     data_tuple_avg,
180                     [float(len(class_labels))] * len(class_labels),
181                 )
182             )
183             self.backprop_and_update_params_one_neuron(
184                 y_error_avg, data_tuple_avg, deriv_sigmoid_avg
185             ) ## BACKPROP loss
186         # plt.figure()
187         return loss_running_record
188         # plt.show()

```

code/sgdp.py

4.2 SGD and SGD+ for multi neuron

```

1 from ComputationalGraphPrimer import ComputationalGraphPrimer
2 import operator
3 import random
4 import numpy as np
5 from tqdm import tqdm
6
7 class SGDPlusMultiNeuron(ComputationalGraphPrimer):
8     def __init__(self, *args, **kwargs) -> None:
9         super().__init__(*args, **kwargs)
10
11     def backprop_and_update_params_multineuron(self, y_error, class_labels):
12         # backproped prediction error:
13         pred_err.backproped_at_layers = {i : [] for i in range(1, self.num_layers
-1)}

```

```

14         pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
15     for back_layer_index in reversed(range(1, self.num_layers)):
16         input_vals = self.forw_prop_vals_at_layers[back_layer_index - 1]
17         input_vals_avg = [sum(x) for x in zip(*input_vals)]
18         input_vals_avg = list(map(operator.truediv, input_vals_avg, [float(
19             len(class_labels)) * len(class_labels))])
20         deriv_sigmoid = self.gradient_vals_for_layers[back_layer_index]
21         deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
22         deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
23                                     [float(len(
24             class_labels)) * len(class_labels))])
25         vars_in_layer = self.layer_vars[back_layer_index] #
26         # a list like ['xo']
27         vars_in_next_layer_back = self.layer_vars[back_layer_index - 1] #
28         # a list like ['xw', 'xz']
29
30         layer_params = self.layer_params[back_layer_index]
31         ## note that layer_params are stored in a dict like
32         ## {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']],
33         2: [['cp', 'cq']]}
34         ## "layer_params[idx]" is a list of lists for the link weights in
35         layer whose output nodes are in layer "idx"
36         transposed_layer_params = list(zip(*layer_params)) ##
37         creating a transpose of the link matrix
38
39         backproped_error = [None] * len(vars_in_next_layer_back)
40         for k, varr in enumerate(vars_in_next_layer_back):
41             for j, var2 in enumerate(vars_in_layer):
42                 backproped_error[k] = sum([self.vals_for_learnable_params[
43                     transposed_layer_params[k][i]] *
44                     pred_err_backproped_at_layers[
45                     back_layer_index][i]
46                     for i in range(len(vars_in_layer))])
47             #
48             range(len(vars_in_layer))]
49         pred_err_backproped_at_layers[back_layer_index - 1] =
50         backproped_error
51         input_vars_to_layer = self.layer_vars[back_layer_index-1]
52         for j, var in enumerate(vars_in_layer):
53             layer_params = self.layer_params[back_layer_index][j]
54             ## Regarding the parameter update loop that follows, see the
55             Slides 74 through 77 of my Week 3
56             ## lecture slides for how the parameters are updated using the
57             partial derivatives stored away
58             ## during forward propagation of data. The theory underlying
59             these calculations is presented
60             ## in Slides 68 through 71.
61             for i, param in enumerate(layer_params):
62                 gradient_of_loss_for_param = input_vals_avg[i] *
63                 pred_err_backproped_at_layers[back_layer_index][j]
64                 # @akamsali: update the velocity parameter and use
65                 self.v[param] = gradient_of_loss_for_param *
66                 deriv_sigmoid_avg[j] \
67                     + (self.momentum * self.v[param])
68
69                 step = self.learning_rate * self.v[param]
70
71                 self.vals_for_learnable_params[param] += step
72             # @akamsali: update the bias parameters

```

```

57         self.v_bias[back_layer_index - 1] = (self.learning_rate * sum(
pred_err_backproped_at_layers[back_layer_index]) \
58                                                     * sum(
deriv_sigmoid_avg)/len(deriv_sigmoid_avg)) \
59                                                     + (self.momentum *
self.v_bias[back_layer_index - 1])
60         self.bias[back_layer_index - 1] += self.v_bias[back_layer_index - 1]
61     #
#####
62 # @akamsali: modified func call name and take in momentum value \mu
63 def train_multineuron(self, training_data, mu=None):
64
65     class DataLoader:
66     def __init__(self, training_data, batch_size):
67         self.training_data = training_data
68         self.batch_size = batch_size
69         self.class_0_samples = [(item, 0) for item in self.training_data[0]]
70     ## Associate label 0 with each sample
71         self.class_1_samples = [(item, 1) for item in self.training_data[1]]
72     ## Associate label 1 with each sample
73
74     def __len__(self):
75         return len(self.training_data[0]) + len(self.training_data[1])
76
77     def _getitem(self):
78         cointoss = random.choice([0,1])
79     ## When a
80     batch is created by getbatch(), we want the
81     ##
82     samples to be chosen randomly from the two lists
83     if cointoss == 0:
84         return random.choice(self.class_0_samples)
85     else:
86         return random.choice(self.class_1_samples)
87
88     def getbatch(self):
89         batch_data, batch_labels = [], []
90     ## First
91     list for samples, the second for labels
92     maxval = 0.0
93     ## For
94     approximate batch data normalization
95     for _ in range(self.batch_size):
96         item = self._getitem()
97         if np.max(item[0]) > maxval:
98             maxval = np.max(item[0])
99         batch_data.append(item[0])
100         batch_labels.append(item[1])
101     batch_data = [item/maxval for item in batch_data]
102     ##
103     Normalize batch data
104     batch = [batch_data, batch_labels]
105     return batch
106
107     """
108     The training loop must first initialize the learnable parameters. Remember,
109     these are the
110     symbolic names in your input expressions for the neural layer that do not
111     begin with the
112     letter 'x'. In this case, we are initializing with random numbers from a
113     uniform distribution

```

```

102         over the interval (0,1).
103         """
104         self.vals_for_learnable_params = {param: random.uniform(0,1) for param in
self.learnable_params}
105         # @akamsali: initialise v as 0 for learnable parameters
106         self.v = {param: 0 for param in self.learnable_params}
107         self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]    ##
Adding the bias to each layer improves
108
109         class discrimination. We initialize it                                ##
110
111         to a random number.
112         # @akamsali: initialise bias velocity to zero
113         self.v.bias = [0] * (self.num_layers-1)
114         # @akamsali: initialise moments
115         # zero implies SGD, non-zero is SGD+
116         if mu is None:
117             self.momentum = 0
118         else:
119             self.momentum = mu
120
121         data_loader = DataLoader(training_data, batch_size=self.batch_size)
122         loss_running_record = []
123         i = 0
124         avg_loss_over_iterations = 0.0    ##
Average the loss over iterations for printing out
125
126         every N iterations during the training loop.
127         for i in tqdm(range(self.training_iterations)):
128             data = data_loader.getbatch()
129             data_tuples = data[0]
130             class_labels = data[1]
131             self.forward_prop_multi_neuron_model(data_tuples)
132             ## FORW PROP works by side-effect
133             predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.
num_layers-1]    ## Predictions from FORW PROP
134             y_preds = [item for sublist in predicted_labels_for_batch for item in
sublist]    ## Get numeric vals for predictions
135             loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(
class_labels))])    ## Calculate loss for batch
136             loss_avg = loss / float(len(class_labels))
137             ## Average the loss over batch
138             avg_loss_over_iterations += loss_avg
139             ## Add to Average loss over iterations
140             if i%(self.display_loss_how_often) == 0:
141                 avg_loss_over_iterations /= self.display_loss_how_often
142                 loss_running_record.append(avg_loss_over_iterations)
143                 # print("[iter=%d] loss = %.4f" % (i+1, avg_loss_over_iterations))
144                 ## Display avg loss
145                 avg_loss_over_iterations = 0.0
146                 ## Re-initialize avg-over-iterations loss
147             y_errors = list(map(operator.sub, class_labels, y_preds))
148             y_error_avg = sum(y_errors) / float(len(class_labels))
149             # @akamsali: change to modified backprop
150             self.backprop_and_update_params_multineuron(y_error_avg, class_labels)
151             ## BACKPROP loss
152
153         return loss_running_record

```

code/sgd.mn.py

4.3 AdaM for one neuron

```

1 from ComputationalGraphPrimer import ComputationalGraphPrimer
2 import random
3 import numpy as np
4 import operator
5
6 class myADAM(ComputationalGraphPrimer):
7     '''
8     @akamsali:
9     modified from the backprop_and_update_params_one_neuron_model method in the
10     ComputationalGraphPrimer class
11     in the ComputationalGraphPrimer.py file.
12
13     The modification is to use the SGD+ algorithm to update the step size
14     from:
15
16      $p_{t+1} = p_t - \eta g_{t+1}$ 
17
18     to:
19
20      $m_{t+1} = \beta_1 * m_t + (1-\beta_1) * g_{t+1}$ 
21
22      $v_{t+1} = \beta_2 * v_t + (1-\beta_2) * g_{t+1}^2$ 
23
24      $p_{t+1} = p_t - \eta (\hat{m}_{t+1} / \sqrt{\hat{v}_{t+1}})$ 
25
26      $\hat{m}_{t+1} = m_t / (1 - \beta_1^t)$ 
27
28      $\hat{v}_{t+1} = v_t / (1 - \beta_2^t)$ 
29
30     where m and v are the first and second momentum parameters
31
32      $\eta$  = learning rate
33
34      $g_{t+1}$  = gradient of the loss function with respect to the learnable
35     parameters (input_val * deriv_sigmoid)
36
37      $p_t$  = learnable parameters
38
39      $m_0$  = all 0's
40
41      $v_0$  = all 0's
42     '''
43     def __init__(self, *args, **kwargs):
44         super().__init__(*args, **kwargs)
45
46     def backprop_and_update_parama-bias(self, y_error, vals_for_input_vars,
47     deriv_sigmoid):
48         input_vars = self.independent_vars
49         vals_for_input_vars_dict = dict(zip(input_vars, list(vals_for_input_vars)))
50         vals_for_learnable_params = self.vals_for_learnable_params
51         for i, param in enumerate(self.vals_for_learnable_params):
52             ## @akamsali: Calculate the next step in the parameter hyperplane
53             g_t = y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
54             m_val = self.beta_1 * self.m[param] + (1-self.beta_1) * g_t
55             v_val = self.beta_2 * self.v[param] + (1-self.beta_2) * (g_t**2)
56             m_hat = m_val / (1 - self.beta_1 ** self.time[param] )

```

```

56         v_hat = v_val / (1 - self.beta_2 ** self.time[param] )
57
58         step = self.learning_rate * m_hat / np.sqrt(v_hat + self.epsilon)
59         ## @akamsali: Update the learnable parameters
60         self.vals_for_learnable_params[param] += step
61         self.m[param] = m_val
62         self.v[param] = v_val
63         self.time[param] += 1
64
65         ## @akamsali: Update the bias
66         m_bias_val = self.beta_1 * self.m_bias + (1 - self.beta_1) * (y_error *
67         deriv_sigmoid)
68         v_bias_val = self.beta_2 * self.v_bias + (1 - self.beta_2) * ((y_error *
69         deriv_sigmoid)**2)
70
71         m_bias_hat = m_bias_val / (1 - (self.beta_1 ** self.time_bias))
72         v_bias_hat = v_bias_val / (1 - (self.beta_2 ** self.time_bias))
73         bias_step = self.learning_rate * (m_bias_hat / np.sqrt(v_bias_hat + 1e-7))
74         self.time_bias += 1
75         self.bias += bias_step
76         self.m_bias = m_bias_val
77         self.v_bias = v_bias_val
78
79 def train(self, training_data, beta_1=0.9, beta_2=0.99, epsilon=1e-7):
80     """
81     @akamsali: Taking Avi's code as is for training a one neuron model. The only
82     modification
83     is to return the loss running record so that we can plot it later.
84     """
85     """
86     The training loop must first initialize the learnable parameters. Remember,
87     these are the
88     symbolic names in your input expressions for the neural layer that do not
89     begin with the
90     letter 'x'. In this case, we are initializing with random numbers from a
91     uniform distribution
92     over the interval (0,1).
93     """
94
95     self.vals_for_learnable_params = {
96         param: random.uniform(0, 1) for param in self.learnable_params
97     }
98
99     self.bias = random.uniform(
100         0, 1
101     ) ## Adding the bias improves class discrimination.
102     ## We initialize it to a random number.
103     # @akamsali: initialise parameters and bias parameter moments
104     self.beta_1 = beta_1
105     self.beta_2 = beta_2
106     self.epsilon = epsilon
107     self.m = {param: 0 for param in self.learnable_params}
108     self.v = {param: 0 for param in self.learnable_params}
109     self.time = {param: 1 for param in self.learnable_params}
110     self.time_bias = 1
111     self.m_bias = 0
112     self.v_bias = 0

```



```

110 class DataLoader:
111
112     def __init__(self, training_data, batch_size):
113         self.training_data = training_data
114         self.batch_size = batch_size
115         self.class_0_samples = [
116             (item, 0) for item in self.training_data[0]
117         ] ## Associate label 0 with each sample
118         self.class_1_samples = [
119             (item, 1) for item in self.training_data[1]
120         ] ## Associate label 1 with each sample
121
122     def __len__(self):
123         return len(self.training_data[0]) + len(self.training_data[1])
124
125     def _getitem(self):
126         cointoss = random.choice(
127             [0, 1]
128         ) ## When a batch is created by getbatch(), we want the
129         ## samples to be chosen randomly from the two lists
130         if cointoss == 0:
131             return random.choice(self.class_0_samples)
132         else:
133             return random.choice(self.class_1_samples)
134
135     def getbatch(self):
136         batch_data, batch_labels = (
137             [],
138             [],
139         ) ## First list for samples, the second for labels
140         maxval = 0.0 ## For approximate batch data normalization
141         for _ in range(self.batch_size):
142             item = self._getitem()
143             if np.max(item[0]) > maxval:
144                 maxval = np.max(item[0])
145             batch_data.append(item[0])
146             batch_labels.append(item[1])
147         batch_data = [
148             item / maxval for item in batch_data
149         ] ## Normalize batch data
150         batch = [batch_data, batch_labels]
151         return batch
152
153     data_loader = DataLoader(training_data, batch_size=self.batch_size)
154     loss_running_record = []
155     i = 0
156     avg_loss_over_iterations = (
157         0.0 ## Average the loss over iterations for printing out
158     )
159
160
161
162     ## every N iterations during the training loop.
163     for i in range(self.training_iterations):
164         data = data_loader.getbatch()
165         data_tuples = data[0]
166         class_labels = data[1]
167         y_preds, deriv_sigmoids = self.forward_prop_one_neuron_model(
168             data_tuples
169         ) ## FORWARD PROP of data

```

```

170         loss = sum(
171             [
172                 (abs(class_labels[i] - y_preds[i])) ** 2
173                 for i in range(len(class_labels))
174             ]
175         ) ## Find loss
176         loss_avg = loss / float(len(class_labels)) ## Average the loss over
batch
177         avg_loss_over_iterations += loss_avg
178         if i % (self.display_loss_how_often) == 0:
179             avg_loss_over_iterations /= self.display_loss_how_often
180             loss_running_record.append(avg_loss_over_iterations)
181             # print("[iter=%d] loss = %.4f" % (i+1, avg_loss_over_iterations))
182             ## Display average loss
183             avg_loss_over_iterations = 0.0 ## Re-initialize avg loss
184             y_errors = list(map(operator.sub, class_labels, y_preds))
185             y_error_avg = sum(y_errors) / float(len(class_labels))
186             deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
187             data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
188             data_tuple_avg = list(
189                 map(
190                     operator.truediv,
191                     data_tuple_avg,
192                     [float(len(class_labels))] * len(class_labels),
193                 )
194             )
195             self.backprop_and_update_params_bias(
196                 y_error_avg, data_tuple_avg, deriv_sigmoid_avg
197             ) ## BACKPROP loss
198             # plt.figure()
199             return loss_running_record
200             # plt.show()

```

code/ADAM_on.py

4.4 AdaM for multi neuron

```

1 from ComputationalGraphPrimer import ComputationalGraphPrimer
2 import operator
3 import random
4 import numpy as np
5 from tqdm import tqdm
6
7 class myADAMMultiNeuron(ComputationalGraphPrimer):
8     def __init__(self, *args, **kwargs) -> None:
9         super().__init__(*args, **kwargs)
10
11     def backprop_and_update_params_multineuron(self, y_error, class_labels):
12         # backproped prediction error:
13         pred_err_backproped_at_layers = {i : [] for i in range(1, self.num_layers
-1)}
14         pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
15         for back_layer_index in reversed(range(1, self.num_layers)):
16             input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
17             input_vals_avg = [sum(x) for x in zip(*input_vals)]
18             input_vals_avg = list(map(operator.truediv, input_vals_avg, [float(
len(class_labels))] * len(class_labels))))
19             deriv_sigmoid = self.gradient_vals_for_layers[back_layer_index]

```

```

20         deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
21         deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
22                                     [float(len(
class_labels))] * len(class_labels)))
23         vars_in_layer = self.layer_vars[back_layer_index] #
# a list like ['xo']
24         vars_in_next_layer_back = self.layer_vars[back_layer_index - 1] #
# a list like ['xw', 'xz']
25
26         layer_params = self.layer_params[back_layer_index]
27         ## note that layer_params are stored in a dict like
28         ## {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']],
29         2: [['cp', 'cq']]}
30         ## "layer_params[idx]" is a list of lists for the link weights in
31         layer whose output nodes are in layer "idx"
32         transposed_layer_params = list(zip(*layer_params)) ##
33         creating a transpose of the link matrix
34
35         backproped_error = [None] * len(vars_in_next_layer_back)
36         for k, varr in enumerate(vars_in_next_layer_back):
37             for j, var2 in enumerate(vars_in_layer):
38                 backproped_error[k] = sum([self.vals_for_learnable_params[
39 transposed_layer_params[k][i]] *
40 back_layer_index][i]
41                                     for i in range(len(vars_in_layer))])
42         #
43         range(len(vars_in_layer))]
44         pred_err_backproped_at_layers[back_layer_index - 1] =
45         backproped_error
46         input_vars_to_layer = self.layer_vars[back_layer_index - 1]
47         for j, var in enumerate(vars_in_layer):
48             layer_params = self.layer_params[back_layer_index][j]
49             ## Regarding the parameter update loop that follows, see the
50             Slides 74 through 77 of my Week 3
51             ## lecture slides for how the parameters are updated using the
52             partial derivatives stored away
53             ## during forward propagation of data. The theory underlying
54             these calculations is presented
55             ## in Slides 68 through 71.
56             for i, param in enumerate(layer_params):
57
58                 # @akamsali: update the velocity parameter and use
59                 g_t = input_vals_avg[i] * pred_err_backproped_at_layers[
60 back_layer_index][j] * deriv_sigmoid_avg[j]
61
62                 m_val = self.beta_1 * self.m[param] + (1 - self.beta_1) * g_t
63                 v_val = self.beta_2 * self.v[param] + (1 - self.beta_2) * (g_t
64 **2)
65
66                 m_hat = m_val / (1 - self.beta_1 ** self.time[param])
67                 v_hat = v_val / (1 - self.beta_2 ** self.time[param])
68
69                 # @akamsali: update the learnable parameters
70                 step = self.learning_rate * m_hat / np.sqrt(v_hat + self.
71 epsilon)
72
73                 self.vals_for_learnable_params[param] += step
74                 # s@akamsali: store the current values of first and second
75                 moment parameters
76
77                 # for next iteration of training
78                 self.m[param] = m_val

```

```

63         self.v[param] = v_val
64         self.time[param] += 1 # update time step
65
66         ## @akamsali: Update the bias
67         m_bias_val = self.beta_1 * self.m_bias[back_layer_index -1] + \
68             (1 - self.beta_1) * (np.sum(pred_err_backproped_at_layers
69 [back_layer_index]) * np.mean(deriv_sigmoid_avg))
70         v_bias_val = self.beta_2 * self.v_bias[back_layer_index -1] + \
71             (1 - self.beta_2) * (np.sum(pred_err_backproped_at_layers
72 [back_layer_index]) * np.mean(deriv_sigmoid_avg)**2)
73
74         m_bias_hat = m_bias_val / (1 - (self.beta_1 ** self.time_bias[
75 back_layer_index -1]))
76         v_bias_hat = v_bias_val / (1 - (self.beta_2 ** self.time_bias[
77 back_layer_index -1]))
78
79         ## @akamsali: Update the bias parameters
80         bias_step = self.learning_rate * (m_bias_hat / np.sqrt(np.abs(
81 v_bias_hat) + self.epsilon))
82         # print(f"v_bias_hat: {v_bias_hat}")
83         # np.sqrt(v_bias_hat + 1e-7)
84         self.bias += bias_step
85
86         # @akamsali: store the current values of first and second moment
87         parameters
88         # for next iteration of training
89
90         self.m_bias[back_layer_index -1] = m_bias_val
91         self.v_bias[back_layer_index -1] = v_bias_val
92         self.time_bias[back_layer_index -1] += 1 # update time step
93
94 #
95 #####
96
97 # @akamsali: modified func call name and take in momentum value \mu
98 def train_multineuron(self, training_data, beta_1=0.9, beta_2=0.99, epsilon=1e-7)
99 :
100
101     class DataLoader:
102         def __init__(self, training_data, batch_size):
103             self.training_data = training_data
104             self.batch_size = batch_size
105             self.class_0_samples = [(item, 0) for item in self.training_data[0]]
106             ## Associate label 0 with each sample
107             self.class_1_samples = [(item, 1) for item in self.training_data[1]]
108             ## Associate label 1 with each sample
109
110         def __len__(self):
111             return len(self.training_data[0]) + len(self.training_data[1])
112
113         def __getitem__(self):
114             cointoss = random.choice([0,1])
115             batch is created by getbatch(), we want the
116
117             samples to be chosen randomly from the two lists
118             if cointoss == 0:
119                 return random.choice(self.class_0_samples)
120             else:

```

```

110         return random.choice(self.class_1_samples)
111
112     def getbatch(self):
113         batch_data, batch_labels = [], []
114         maxval = 0.0
115         for _ in range(self.batch_size):
116             item = self._getitem()
117             if np.max(item[0]) > maxval:
118                 maxval = np.max(item[0])
119             batch_data.append(item[0])
120             batch_labels.append(item[1])
121         batch_data = [item/maxval for item in batch_data]
122         batch = [batch_data, batch_labels]
123         return batch
124
125     """
126     The training loop must first initialize the learnable parameters. Remember,
127     these are the
128     symbolic names in your input expressions for the neural layer that do not
129     begin with the
130     letter 'x'. In this case, we are initializing with random numbers from a
131     uniform distribution
132     over the interval (0,1).
133     """
134     self.vals_for_learnable_params = {param: random.uniform(0,1) for param in
135     self.learnable_params}
136
137     self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]
138
139     # @akamsali: set hyperparameters
140     self.beta_1 = beta_1
141     self.beta_2 = beta_2
142     self.epsilon = epsilon
143     # @akamsali: initialise learnable parameter moments
144     self.m = {param: 0 for param in self.learnable_params}
145     self.v = {param: 0 for param in self.learnable_params}
146     self.time = {param: 1 for param in self.learnable_params}
147
148     # @akamsali: initialise bias parameter moments
149     self.time_bias = [1]*(self.num_layers-1)
150     self.m_bias = [0]*(self.num_layers-1)
151     self.v_bias = [0]*(self.num_layers-1)
152
153     data_loader = DataLoader(training_data, batch_size=self.batch_size)
154     loss_running_record = []
155     i = 0
156     avg_loss_over_iterations = 0.0
157
158     every N iterations during the training loop.

```

```

158         for i in tqdm(range(self.training_iterations)):
159             data = data_loader.getbatch()
160             data_tuples = data[0]
161             # print(data_tuples)
162             class_labels = data[1]
163             self.forward_prop_multi_neuron_model(data_tuples)
164             ## FORW PROP works by side-effect
165             predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.
num_layers-1] ## Predictions from FORW PROP
166             y_preds = [item for sublist in predicted_labels_for_batch for item in
sublist] ## Get numeric vals for predictions
167             loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(
class_labels))]) ## Calculate loss for batch
168             loss_avg = loss / float(len(class_labels))
169             ## Average the loss over batch
170             avg_loss_over_iterations += loss_avg
171             ## Add to Average loss over iterations
172             if i%(self.display_loss_how_often) == 0:
173                 avg_loss_over_iterations /= self.display_loss_how_often
174                 loss_running_record.append(avg_loss_over_iterations)
175                 # print("[iter=%d] loss = %.4f" % (i+1, avg_loss_over_iterations))
176                 ## Display avg loss
177                 avg_loss_over_iterations = 0.0
178                 ## Re-initialize avg-over-iterations loss
179                 y_errors = list(map(operator.sub, class_labels, y_preds))
180                 y_error_avg = sum(y_errors) / float(len(class_labels))
181                 # @akamsali: change to modified backprop
182                 self.backprop_and_update_params_multineuron(y_error_avg, class_labels)
183             ## BACKPROP loss
184
185         return loss_running_record

```

code/ADAM_multi.py

4.5 Training code

```

1 from sgdp import SGDplus
2 from sgd_mn import SGDPlusMultiNeuron
3 from ADAM_on import myADAM
4 from ADAM_multi import myADAMMultiNeuron
5
6 import matplotlib.pyplot as plt
7
8 lr = 5*1e-3
9 # initialise SGD and SGD+ for one neuron
10 sgd_on = SGDplus(
11     one_neuron_model = True,
12     expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
13     output_vars = ['xw'],
14     dataset_size = 5000,
15     learning_rate = lr,
16     # learning_rate = 5 * 1e-2,
17     training_iterations = 40000,
18     batch_size = 8,
19     display_loss_how_often = 100,
20     debug = True,
21 )
22

```

```

23
24 sgd_on.parse-expressions()
25 training_data = sgd_on.gen-training-data()
26 # SGD
27 sgd_on_loss_0 = sgd_on.train-one-neuron(training_data)
28 # SGD+ with mu = 0.5, 0.9
29 sgd_on_loss_5 = sgd_on.train-one-neuron(training_data, mu=0.5)
30 sgd_on_loss_9 = sgd_on.train-one-neuron(training_data, mu=0.9)
31
32 # initialise SGD and SGD+ for multi neuron
33 sgd_mn = SGDPlusMultiNeuron(
34     num_layers = 3,
35     layers_config = [4,2,1], # num of nodes in
        each layer
36     expressions = [ 'xw=ap*xp+aq*xq+ar*xr+as*xs',
37                     'xz=bp*xp+bq*xq+br*xr+bs*xs',
38                     'xo=cp*xw+cq*xz' ],
39     output_vars = [ 'xo' ],
40     dataset_size = 5000,
41     learning_rate = lr,
42     # learning_rate = 5 * 1e-2,
43     training_iterations = 40000,
44     batch_size = 8,
45     display_loss_how_often = 100,
46     debug = True,
47 )
48
49 sgd_mn.parse-multi-layer-expressions()
50
51 # SGD
52 sgd_mn_loss_0 = sgd_mn.train-multineuron(training_data)
53 # SGD+ with mu = 0.5, 0.9
54 sgd_mn_loss_5 = sgd_mn.train-multineuron(training_data, mu=0.5)
55 sgd_mn_loss_9 = sgd_mn.train-multineuron(training_data, mu=0.9)
56
57
58 # initialise ADAM for one neuron
59 adam_on = myADAM(
60     one_neuron_model = True,
61     expressions = [ 'xw=ab*xa+bc*xb+cd*xc+ac*xd' ],
62     output_vars = [ 'xw' ],
63     dataset_size = 5000,
64     learning_rate = lr,
65     # learning_rate = 5 * 1e-2,
66     training_iterations = 40000,
67     batch_size = 8,
68     display_loss_how_often = 100,
69     debug = True,
70 )
71
72
73 adam_on.parse-expressions()
74 # training_data = cgp.gen-training-data()
75
76 # loss_0 = cgp.train(training_data)
77 loss = adam_on.train(training_data)
78 # plt.plot(loss)
79
80 # loss_9 = cgp.train(training_data, mu=0.9)
81 # plt.plot(loss_0)

```

```

82 # plt.plot(loss_9)
83
84 # initialise ADAM for multi neuron
85 adam_mn = myADAMMultiNeuron(
86     num_layers = 3,
87     layers_config = [4,2,1], # num of nodes in
88     each_layer
89     expressions = [ 'xw=ap*xp+aq*xq+ar*xr+as*xs',
90                     'xz=bp*xp+bq*xq+br*xr+bs*xs',
91                     'xo=cp*xw+cq*xz' ],
92     output_vars = [ 'xo' ],
93     dataset_size = 5000,
94     learning_rate = lr,
95     # learning_rate = 5 * 1e-2,
96     training_iterations = 40000,
97     batch_size = 8,
98     display_loss_how_often = 100,
99     debug = True,
100 )
101 adam_mn.parse_multi_layer_expressions()
102
103 training_data = adam_mn.gen_training_data()
104 adam_mn_loss = adam_mn.train_multineuron(training_data)
105
106
107 # PLOT one neuron model losses
108 plt.plot(sgd_on_loss_0, label='SGD')
109 plt.plot(sgd_on_loss_5, label='SGD+, mu=0.5')
110 plt.plot(sgd_on_loss_9, label='SGD+, mu=0.9')
111 plt.plot(loss, label='ADAM')
112 plt.legend()
113 plt.title(f'SGD, SGD+ and ADAM for one neuron, LR={lr}')
114 plt.savefig(f"one-neuron-5")
115
116 # PLOT multi neuron model losses
117
118 plt.plot(sgd_mn_loss_0, label='SGD')
119 plt.plot(sgd_mn_loss_5, label='SGD+, mu=0.5')
120 plt.plot(sgd_mn_loss_9, label='SGD+, mu=0.9')
121 plt.plot(adam_mn_loss, label='ADAM')
122 plt.legend()
123 plt.title(f'SGD, SGD+ and ADAM for multi neuron, LR={lr}')
124 plt.savefig(f"multi-neuron-5")

```

code/main.py