

1 Introduction

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

2 Theoretical Background

2.1 Gradient Descent

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

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

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

2.2 Stochastic Gradient Descent

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

2.3 Stochastic Gradient Descent Plus

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

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

$$p_{t+1} = p_t - \alpha \times v_{t+1} \quad (3)$$

where v_t is the step size at parameter t .

2.4 Adam Optimizer

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

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

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

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

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

$$\hat{m}_k = \frac{m_k}{\sqrt{1 - \beta_1^k}} \quad (7)$$

and \hat{v} is

$$\hat{v}_k = \frac{v_k}{\sqrt{1 - \beta_2^k}} \quad (8)$$

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

3 Methodology

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

1. We create two new sub classes that inherit from the **ComputationalGraphPrimer** class for the SGD+ and Adam approach
2. The **run_training_loop_one_neuron_model** and **run_training_loop_multi_neuron_model** methods are created again in their respective subclasses so that the method from the parent class is overridden. These methods are modified to initialize new variables for updating the learnable parameters
 - (a) For SGD+, we introduce a list storing all the step sizes, momentum, and a new bias factor that updates the bias based on the momentum. The momentum is set to 0.85
 - (b) For Adam, we introduce β_1 , β_2 , ϵ , a list of weights for moment m and moment v , and their respective biases. β_1 is set to 0.9, β_2 is set to 0.999, and ϵ is set to $1e-6$
3. The **backprop_and_update_params_one_neuron_model** and **backprop_and_update_params_multi_neuron_model** are created again in their respective subclasses so that the method from the parent class is overridden. These methods are modified to include Equations (2) and (3) for SGD+ and Equations (4)-(8) for Adam to update the step size, learnable parameters, and bias.
4. The losses for SGD, SGD+, and Adam are displayed in a graph for two different learning rates as shown in the results section

3.1 One Neuron Classifier

```
1 # Import Libraries
2 import random
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import operator
6 from ComputationalGraphPrimer import *
7
8 # Constants
9 SEED = 512
10 random.seed(SEED)
11 np.random.seed(SEED)
12
13
14 class ComputationalGraphPrimerSGDPlus(ComputationalGraphPrimer):
15     def __init__(self, *args, **kwargs):
16         super().__init__(*args, **kwargs) # Inheriting from the parent
17         class
18
19     # Modifying and Overriding the run_training_loop_one_neuron_model to
20     implement SGD+
```

```

19     # mu is between [0,1]
20     def run_training_loop_one_neuron_model(self, training_data, mu=0.5):
21         #####Copied from the original
function#####
22         self.vals_for_learnable_params = {param: random.uniform(0,1) for
param in self.learnable_params} # initializing learnable parameters
with random numbers from a uniform distribution over the interval (0,1)
23
24         self.bias = random.uniform(0,1)                ## Adding the
bias improves class discrimination.
25
26                                     ## We
initialize it to a random number.
27
28         class DataLoader:
29             """
30             To understand the logic of the dataloader, it would help if
you first understand how
31             the training dataset is created. Search for the following
function in this file:
32
33                                     gen_training_data(self)
34
35             As you will see in the implementation code for this method,
the training dataset
36             consists of a Python dict with two keys, 0 and 1, the former
points to a list of
37             all Class 0 samples and the latter to a list of all Class 1
samples. In each list,
38             the data samples are drawn from a multi-dimensional Gaussian
distribution. The two
39             classes have different means and variances. The
dimensionality of each data sample
40             is set by the number of nodes in the input layer of the neural
network.
41
42             The data loader's job is to construct a batch of samples drawn
randomly from the two
43             lists mentioned above. And it must also associate the class
label with each sample
44             separately.
45             """
46             def __init__(self, training_data, batch_size):
47                 self.training_data = training_data
48                 self.batch_size = batch_size
49                 self.class_0_samples = [(item, 0) for item in self.
training_data[0]]    ## Associate label 0 with each sample
50                 self.class_1_samples = [(item, 1) for item in self.
training_data[1]]    ## Associate label 1 with each sample
51
52             def __len__(self):
53                 return len(self.training_data[0]) + len(self.training_data
[1])
54
55             def __getitem__(self):

```

```

55         cointoss = random.choice([0,1])
56         ## When a batch is created by getbatch(), we want the
57         ## samples to be chosen randomly from the two lists
58         if cointoss == 0:
59             return random.choice(self.class_0_samples)
60         else:
61             return random.choice(self.class_1_samples)
62
63     def getbatch(self):
64         batch_data, batch_labels = [], []
65         ## First list for samples, the second for labels
66         maxval = 0.0
67         ## For approximate batch data normalization
68         for _ in range(self.batch_size):
69             item = self._getitem()
70             if np.max(item[0]) > maxval:
71                 maxval = np.max(item[0])
72             batch_data.append(item[0])
73             batch_labels.append(item[1])
74         batch_data = [item/maxval for item in batch_data]
75         ## Normalize batch data
76         batch = [batch_data, batch_labels]
77         return batch
78
79     #
80     #####
81
82     # Modified part of the function
83     self.mu = mu
84     self.bias_factor = 0 # Update the bias, the factor depends on the
85     current mu
86     self.step_sizes = [0 for i in range(len(self.learnable_params) +
87     1)]
88
89     ##### Copied from the original
90     function#####
91     data_loader = DataLoader(training_data, batch_size=self.batch_size
92     )
93     loss_running_record = []
94     i = 0
95     avg_loss_over_iterations = 0.0
96     ## Average the loss over iterations for printing out
97
98     ## every N iterations during the training loop.
99     for i in range(self.training_iterations):
100         data = data_loader.getbatch()
101         data_tuples = data[0]
102         class_labels = data[1]
103         y_preds, deriv_sigmoids = self.forward_prop_one_neuron_model(
104         data_tuples)
105         ## FORWARD PROP of data
106         loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
107         range(len(class_labels))]) ## Find loss
108         loss_avg = loss / float(len(class_labels))

```

```

94         ## Average the loss over batch
95         avg_loss_over_iterations += loss_avg
96         if i%(self.display_loss_how_often) == 0:
97             avg_loss_over_iterations /= self.display_loss_how_often
98             loss_running_record.append(avg_loss_over_iterations)
99             print("[iter=%d]  loss = %.4f" % (i+1,
100             avg_loss_over_iterations))          ## Display average loss
101             avg_loss_over_iterations = 0.0
102             ## Re-initialize avg loss
103             y_errors = list(map(operator.sub, class_labels, y_preds))
104             y_error_avg = sum(y_errors) / float(len(class_labels))
105             deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(
106             class_labels))
107             data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
108             data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
109             [float(len(class_labels))] * len(
110             class_labels) ))
111             self.backprop_and_update_params_one_neuron_model(y_error_avg,
112             data_tuple_avg, deriv_sigmoid_avg)      ## BACKPROP loss
113             # plt.figure()
114             # plt.plot(loss_running_record)
115             # plt.show()
116             return loss_running_record
117
118         #
119         #####
120
121     # Modify backpropagation function for one_neuron_model -
122     backpropagating the loss and updating the values of the learnable
123     parameters.
124     def backprop_and_update_params_one_neuron_model(self, y_error,
125     vals_for_input_vars, deriv_sigmoid):
126         """
127         As should be evident from the syntax used in the following call to
128         backprop function,
129
130         self.backprop_and_update_params_one_neuron_model( y_error_avg,
131         data_tuple_avg, deriv_sigmoid_avg)
132
133         ~~~
134
135         the values fed to the backprop function for its three arguments
136         are averaged over the training
137         samples in the batch. This in keeping with the spirit of SGD that
138         calls for averaging the
139         information retained in the forward propagation over the samples
140         in a batch.
141
142         See Slide 59 of my Week 3 slides for the math of back propagation
143         for the One-Neuron network.
144         """
145         input_vars = self.independent_vars
146         vals_for_input_vars_dict = dict(zip(input_vars, list(
147         vals_for_input_vars)))

```

```

129         vals_for_learnable_params = self.vals_for_learnable_params
130         for i,param in enumerate(self.vals_for_learnable_params):
131             ## Calculate the next step in the parameter hyperplane
132             #####Modified
133             #####
134             self.step_sizes[i + 1] = (self.mu * self.step_sizes[i]) + (
135                 self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]]
136                 * deriv_sigmoid)
137             self.step_sizes[i] = self.step_sizes[i + 1] # Save the new
138             current step size in the previous iteration position
139
140             ## Update the learnable parameters
141             # self.vals_for_learnable_params[param] += -self.learning_rate
142             * self.step_sizes[i + 1]
143             self.vals_for_learnable_params[param] += self.step_sizes[i +
144 1]
145
146         self.bias_factor = (self.mu * self.bias_factor) + (self.
147         learning_rate * y_error * deriv_sigmoid) ## Update the bias
148         self.bias += self.bias_factor
149         #
150         #####
151
152     class ComputationalGraphPrimerAdam(ComputationalGraphPrimer):
153         def __init__(self, *args, **kwargs):
154             super().__init__(*args, **kwargs)
155
156         # Modifying and Overriding the run_training_loop_one_neuron_model to
157         # implement SGD+
158         # Beta1 and Beta2 are close to 1
159         def run_training_loop_one_neuron_model(self, training_data, beta1=0.9,
160             beta2=0.999, e=1e-6):
161             #####Copied from the original
162             function#####
163             self.vals_for_learnable_params = {param: random.uniform(0,1) for
164             param in self.learnable_params} # initializing learnable parameters
165             with random numbers from a uniform distribution over the interval (0,1)
166
167             self.bias = random.uniform(0,1) ## Adding the
168             bias improves class discrimination.
169
170             ## We
171             initialize it to a random number.
172
173             class DataLoader:
174                 """
175                 To understand the logic of the dataloader, it would help if
176                 you first understand how
177                 the training dataset is created. Search for the following
178                 function in this file:
179
180                     gen_training_data(self)
181
182                 As you will see in the implementation code for this method,

```

```

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

170
171     The data loader's job is to construct a batch of samples drawn
randomly from the two
172     lists mentioned above. And it must also associate the class
label with each sample
173     separately.
174     """
175     def __init__(self, training_data, batch_size):
176         self.training_data = training_data
177         self.batch_size = batch_size
178         self.class_0_samples = [(item, 0) for item in self.
training_data[0]]    ## Associate label 0 with each sample
179         self.class_1_samples = [(item, 1) for item in self.
training_data[1]]    ## Associate label 1 with each sample

180
181     def __len__(self):
182         return len(self.training_data[0]) + len(self.training_data
[1])

183
184     def _getitem(self):
185         cointoss = random.choice([0,1])
## When a batch is created by getbatch(), we want the
186
##     samples to be chosen randomly from the two lists
187         if cointoss == 0:
188             return random.choice(self.class_0_samples)
189         else:
190             return random.choice(self.class_1_samples)

191
192     def getbatch(self):
193         batch_data, batch_labels = [], []
## First list for samples, the second for labels
194         maxval = 0.0
## For approximate batch data normalization
195         for _ in range(self.batch_size):
196             item = self._getitem()
197             if np.max(item[0]) > maxval:
198                 maxval = np.max(item[0])
199                 batch_data.append(item[0])
200                 batch_labels.append(item[1])
201         batch_data = [item/maxval for item in batch_data]
## Normalize batch data
202         batch = [batch_data, batch_labels]

```



```

203         return batch
204
205     #
206     #####
207     # Modified part of the function
208     self.beta1, self.beta2 = beta1, beta2
209     self.e = e
210     self.m_db, self.v_db = 0, 0
211     self.m_dw = [0 for i in range(len(self.learnable_params) + 1)] # m
212     self.v_dw = [0 for i in range(len(self.learnable_params) + 1)] # v
213
214     #####Copied from the original
215     function#####
216     data_loader = DataLoader(training_data, batch_size=self.batch_size
217     )
218     loss_running_record = []
219     i = 0
220     avg_loss_over_iterations = 0.0
221     ## Average the loss over iterations for printing out
222
223     ## every N iterations during the training loop.
224     for i in range(self.training_iterations):
225         data = data_loader.getbatch()
226         data_tuples = data[0]
227         class_labels = data[1]
228         y_preds, deriv_sigmoids = self.forward_prop_one_neuron_model(
229         data_tuples)
230         ## FORWARD PROP of data
231         loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
232         range(len(class_labels))]) ## Find loss
233         loss_avg = loss / float(len(class_labels))
234         ## Average the loss over batch
235         avg_loss_over_iterations += loss_avg
236         if i%(self.display_loss_how_often) == 0:
237             avg_loss_over_iterations /= self.display_loss_how_often
238             loss_running_record.append(avg_loss_over_iterations)
239             print("[iter=%d] loss = %.4f" % (i+1,
240             avg_loss_over_iterations)) ## Display average loss
241             avg_loss_over_iterations = 0.0
242             ## Re-initialize avg loss
243             y_errors = list(map(operator.sub, class_labels, y_preds))
244             y_error_avg = sum(y_errors) / float(len(class_labels))
245             deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(
246             class_labels))
247             data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
248             data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
249             [float(len(class_labels))] * len(
250             class_labels) ))
251             self.backprop_and_update_params_one_neuron_model(y_error_avg,
252             data_tuple_avg, deriv_sigmoid_avg, i + 1) ## BACKPROP loss
253             # plt.figure()
254             # plt.plot(loss_running_record)
255             # plt.show()

```

243
244
245
246

247
248

249
250
251

252
253
254

255
256
257

258
259
260
261
262
263
264
265
266

267
268
269
270
271

272
273
274
275
276

```
    return loss_running_record

#
#####

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

        self.backprop_and_update_params_one_neuron_model( y_error_avg,
data_tuple_avg, deriv_sigmoid_avg)

        the values fed to the backprop function for its three arguments
are averaged over the training
        samples in the batch. This in keeping with the spirit of SGD that
calls for averaging the
        information retained in the forward propagation over the samples
in a batch.

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

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

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

```

277         self.vals_for_learnable_params[param] += mk_hat / np.sqrt(
vk_hat + self.e)
278
279         # Inspired by: https://towardsdatascience.com/how-to-implement-an-
adam-optimizer-from-scratch-76e7b217f1cc
280         # Inspired by: https://www.youtube.com/watch?v=JXQT\_vxqwIs&
ab\_channel=DeepLearningAI
281         self.m_db = (self.beta1 * self.m_db) + (1 - self.beta1) * (self.
learning_rate * y_error * deriv_sigmoid)
282         self.v_db = (self.beta2 * self.v_db) + (1 - self.beta2) * (self.
learning_rate * y_error * deriv_sigmoid) ** 2
283
284         m_db_hat = self.m_db / (1 - self.beta1 ** k)
285         v_db_hat = self.v_db / (1 - self.beta1 ** k)
286
287         self.bias += m_db_hat / np.sqrt(v_db_hat + self.e) ## Update the
bias
288         #
#####
289
290 def sgd_plus(lr=1e-3, mu=0.9):
291     cgp = ComputationalGraphPrimerSGDPlus(
292         one_neuron_model = True,
293         expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
294         output_vars = ['xw'],
295         dataset_size = 5000,
296         learning_rate = lr,
297         training_iterations = 40000,
298         batch_size = 8,
299         display_loss_how_often = 100,
300         debug = True,
301     )
302
303     cgp.parse_expressions()
304     # cgp.display_one_neuron_network()
305
306     training_data = cgp.gen_training_data()
307     loss_per_iteration = cgp.run_training_loop_one_neuron_model(
training_data, mu=mu)
308
309     return loss_per_iteration
310
311 def adam(lr=1e-3):
312     cgp = ComputationalGraphPrimerAdam(
313         one_neuron_model = True,
314         expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
315         output_vars = ['xw'],
316         dataset_size = 5000,
317         learning_rate = lr,
318         training_iterations = 40000,
319         batch_size = 8,
320         display_loss_how_often = 100,
321         debug = True,

```

```

322     )
323
324     cgp.parse_expressions()
325     # cgp.display_one_neuron_network()
326
327     training_data = cgp.gen_training_data()
328     loss_per_iteration = cgp.run_training_loop_one_neuron_model(
training_data )
329
330     return loss_per_iteration
331
332 def sgd(lr=1e-3):
333     cgp = ComputationalGraphPrimer(
334         one_neuron_model = True,
335         expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
336         output_vars = ['xw'],
337         dataset_size = 5000,
338         learning_rate = lr,
339         training_iterations = 40000,
340         batch_size = 8,
341         display_loss_how_often = 100,
342         debug = True,
343     )
344
345     cgp.parse_expressions()
346     # cgp.display_one_neuron_network()
347
348     training_data = cgp.gen_training_data()
349     loss_per_iteration = cgp.run_training_loop_one_neuron_model(
training_data )
350
351     return loss_per_iteration
352
353 def plot_losses(sgd, sgd_plus, adam, lr):
354     number_of_iterations = len(adam)
355     plt.plot(range(number_of_iterations), sgd, label="SGD Loss")
356     plt.plot(range(number_of_iterations), sgd_plus, label="SGD+ Loss")
357     plt.plot(range(number_of_iterations), adam, label="Adam Loss")
358
359     plt.title(f"Loss per Iteration for Different Optimizers for One Neuron
Model for Learning Rate: {lr}")
360     plt.xlabel("Iteration Number")
361     plt.ylabel("Loss")
362     plt.legend(loc="upper left")
363
364     plt.show(); quit()
365     plt.savefig(r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW3/
one_neuron_" + str(lr) + "_learning_rate.png", dpi=200)
366
367 if __name__ == "__main__":
368     lr = 1e-3
369     sgd_loss = sgd(lr)
370     sgd_plus_loss = sgd_plus(lr)
371     adam_loss = adam(lr)

```

```

372
373 plot_losses(sgd_loss, sgd_plus_loss, adam_loss, lr)

```

Listing 1: SGD+ and Adam for One Neuron Classifier

3.2 Multi-Neuron Classifier

```

1 # Import Libraries
2 import random
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import operator
6 from ComputationalGraphPrimer import *
7
8 # Constants
9 SEED = 1234
10 random.seed(SEED)
11 np.random.seed(SEED)
12
13 class ComputationalGraphPrimerSGDPlus(ComputationalGraphPrimer):
14     def __init__(self, *args, **kwargs):
15         super().__init__(*args, **kwargs) # Inheriting from the parent
16         class
17
18         # Modifying and Overriding the run_training_loop_one_neuron_model to
19         # implement SGD+
20         # mu is between [0,1]
21
22         def run_training_loop_multi_neuron_model(self, training_data, mu=0.5):
23             ##### Copied from the original
24             function#####
25
26             class DataLoader:
27                 """
28                 To understand the logic of the dataloader, it would help if
29                 you first understand how
30                 the training dataset is created. Search for the following
31                 function in this file:
32
33                 gen_training_data(self)
34
35                 As you will see in the implementation code for this method,
36                 the training dataset
37                 consists of a Python dict with two keys, 0 and 1, the former
38                 points to a list of
39                 all Class 0 samples and the latter to a list of all Class 1
40                 samples. In each list,
41                 the data samples are drawn from a multi-dimensional Gaussian
42                 distribution. The two
43                 classes have different means and variances. The
44                 dimensionality of each data sample
45                 is set by the number of nodes in the input layer of the neural
46                 network.

```

```

36         The data loader's job is to construct a batch of samples drawn
37         randomly from the two
38         lists mentioned above. And it must also associate the class
39         label with each sample
40         separately.
41         """
42         def __init__(self, training_data, batch_size):
43             self.training_data = training_data
44             self.batch_size = batch_size
45             self.class_0_samples = [(item, 0) for item in self.
training_data[0]]    ## Associate label 0 with each sample
46             self.class_1_samples = [(item, 1) for item in self.
training_data[1]]    ## Associate label 1 with each sample
47
48         def __len__(self):
49             return len(self.training_data[0]) + len(self.training_data
[1])
50
51         def __getitem__(self):
52             cointoss = random.choice([0,1])
53             ## When a batch is created by getbatch(), we want the
54             ## samples to be chosen randomly from the two lists
55             if cointoss == 0:
56                 return random.choice(self.class_0_samples)
57             else:
58                 return random.choice(self.class_1_samples)
59
60         def getbatch(self):
61             batch_data, batch_labels = [], []
62             ## First list for samples, the second for labels
63             maxval = 0.0
64             ## For approximate batch data normalization
65             for _ in range(self.batch_size):
66                 item = self.__getitem__()
67                 if np.max(item[0]) > maxval:
68                     maxval = np.max(item[0])
69                 batch_data.append(item[0])
70                 batch_labels.append(item[1])
71             batch_data = [item/maxval for item in batch_data]
72             ## Normalize batch data
73             batch = [batch_data, batch_labels]
74             return batch
75
76         #
77         #####
78
79         # Modified part of the function
80         self.mu = mu
81         self.step_sizes = [0 for i in range(len(self.learnable_params) +
1)]
82         self.bias_factor = 0
83
84         ##### Copied from the original
85         function#####

```

```

76     """
77     The training loop must first initialize the learnable parameters.
78     Remember, these are the
79     symbolic names in your input expressions for the neural layer that
80     do not begin with the
81     letter 'x'. In this case, we are initializing with random numbers
82     from a uniform distribution
83     over the interval (0,1).
84     """
85     self.vals_for_learnable_params = {param: random.uniform(0,1) for
86 param in self.learnable_params}
87
88     self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)
89 ]
90     ## Adding the bias to each layer improves
91
92     ## class discrimination. We initialize it
93
94     ## to a random number.
95
96     data_loader = DataLoader(training_data, batch_size=self.batch_size
97 )
98
99     loss_running_record = []
100     i = 0
101     avg_loss_over_iterations = 0.0
102     ## Average the loss over iterations for printing out
103
104     ## every N iterations during the training loop.
105     for i in range(self.training_iterations):
106         data = data_loader.getbatch()
107         data_tuples = data[0]
108         class_labels = data[1]
109         self.forward_prop_multi_neuron_model(data_tuples)
110         ## FORW PROP works by side-effect
111         predicted_labels_for_batch = self.forw_prop_vals_at_layers[
112 self.num_layers-1]
113         ## Predictions from FORW PROP
114         y_preds = [item for sublist in predicted_labels_for_batch
115 for item in sublist]
116         ## Get numeric vals for predictions
117         loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
118 range(len(class_labels))])
119         ## Calculate loss for batch
120         loss_avg = loss / float(len(class_labels))
121         ## Average the loss over batch
122         avg_loss_over_iterations += loss_avg
123         ## Add to Average loss over iterations
124         if i%(self.display_loss_how_often) == 0:
125             avg_loss_over_iterations /= self.display_loss_how_often
126             loss_running_record.append(avg_loss_over_iterations)
127             print("[iter=%d] loss = %.4f" % (i+1,
128 avg_loss_over_iterations))
129             ## Display avg loss
130             avg_loss_over_iterations = 0.0
131             ## Re-initialize avg-over-iterations loss
132             y_errors = list(map(operator.sub, class_labels, y_preds))
133             y_error_avg = sum(y_errors) / float(len(class_labels))
134             self.backprop_and_update_params_multi_neuron_model(y_error_avg
135 , class_labels)
136             ## BACKPROP loss

```

```

111     # plt.figure()
112     # plt.plot(loss_running_record)
113     # plt.show()
114
115     return loss_running_record
116
117     #
118     #####
119
120     # Modify backpropagation function for one_neuron_model -
121     # backpropagating the loss and updating the values of the learnable
122     # parameters.
123     def backprop_and_update_params_multi_neuron_model(self, y_error,
124     class_labels):
125         """
126         First note that loop index variable 'back_layer_index' starts with
127         the index of
128         the last layer. For the 3-layer example shown for 'forward',
129         back_layer_index
130         starts with a value of 2, its next value is 1, and that's it.
131
132         Stochastic Gradient Gradient calls for the backpropagated loss to
133         be averaged over
134         the samples in a batch. To explain how this averaging is carried
135         out by the
136         backprop function, consider the last node on the example shown in
137         the forward()
138         function above. Standing at the node, we look at the 'input'
139         values stored in the
140         variable "input_vals". Assuming a batch size of 8, this will be
141         list of
142         lists. Each of the inner lists will have two values for the two
143         nodes in the
144         hidden layer. And there will be 8 of these for the 8 elements of
145         the batch. We average
146         these values 'input_vals' and store those in the variable "
147         input_vals_avg". Next we
148         must carry out the same batch-based averaging for the partial
149         derivatives stored in the
150         variable "deriv_sigmoid".
151
152         Pay attention to the variable 'vars_in_layer'. These store the
153         node variables in
154         the current layer during backpropagation. Since back_layer_index
155         starts with a
156         value of 2, the variable 'vars_in_layer' will have just the single
157         node for the
158         example shown for forward(). With respect to what is stored in
159         vars_in_layer', the
160         variables stored in 'input_vars_to_layer' correspond to the input
161         layer with
162         respect to the current layer.
163         """

```



```

144     # backproped prediction error:
145     pred_err_backproped_at_layers = {i : [] for i in range(1,self.
num_layers-1)}
146     pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
147     for back_layer_index in reversed(range(1,self.num_layers)):
148         input_vals = self.forw_prop_vals_at_layers[back_layer_index
-1]
149         input_vals_avg = [sum(x) for x in zip(*input_vals)]
150         input_vals_avg = list(map(operator.truediv, input_vals_avg, [
float(len(class_labels))] * len(class_labels)))
151         deriv_sigmoid = self.gradient_vals_for_layers[
back_layer_index]
152         deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
153         deriv_sigmoid_avg = list(map(operator.truediv,
deriv_sigmoid_avg,
154                                     [float(len(
class_labels))] * len(class_labels)))
155         vars_in_layer = self.layer_vars[back_layer_index]
156         ## a list like ['xo']
157         vars_in_next_layer_back = self.layer_vars[back_layer_index -
1]
158         ## a list like ['xw', 'xz']
159         layer_params = self.layer_params[back_layer_index]
160         ## note that layer_params are stored in a dict like
161         ## {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', '
bs']], 2: [['cp', 'cq']]}
162         ## "layer_params[idx]" is a list of lists for the link weights
in layer whose output nodes are in layer "idx"
163         transposed_layer_params = list(zip(*layer_params))
164         ## creating a transpose of the link matrix
165         backproped_error = [None] * len(vars_in_next_layer_back)
166         for k, varr in enumerate(vars_in_next_layer_back):
167             for j, var2 in enumerate(vars_in_layer):
168                 backproped_error[k] = sum([self.
vals_for_learnable_params[transposed_layer_params[k][i]] *
169                 pred_err_backproped_at_layers[back_layer_index][i]
170                 for i in range(len(
vars_in_layer))])
171                 ## deriv_sigmoid_avg[i] for i
in range(len(vars_in_layer))]
172         pred_err_backproped_at_layers[back_layer_index - 1] =
backproped_error
173         input_vars_to_layer = self.layer_vars[back_layer_index-1]
174         for j, var in enumerate(vars_in_layer):
175             layer_params = self.layer_params[back_layer_index][j]
176             ## Regarding the parameter update loop that follows, see
the Slides 74 through 77 of my Week 3
177             ## lecture slides for how the parameters are updated
using the partial derivatives stored away
178             ## during forward propagation of data. The theory
underlying these calculations is presented
179             ## in Slides 68 through 71.

```

```

179         for i,param in enumerate(layer_params):
180             gradient_of_loss_for_param = input_vals_avg[i] *
pred_err_backproped_at_layers[back_layer_index][j]
181             #####Modified
#####
182             self.step_sizes[i + 1] = (self.mu * self.step_sizes[i
]) + (self.learning_rate * gradient_of_loss_for_param *
deriv_sigmoid_avg[j])
183             self.step_sizes[i] = self.step_sizes[i + 1]
184             self.vals_for_learnable_params[param] += self.
step_sizes[i + 1]
185
186             self.bias_factor = (self.mu * self.bias_factor) + (self.
learning_rate * sum(pred_err_backproped_at_layers[back_layer_index]) \
187
* sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))
188             self.bias[back_layer_index-1] += self.bias_factor
189             #
#####

190
191 class ComputationalGraphPrimerAdam(ComputationalGraphPrimer):
192     def __init__(self, *args, **kwargs):
193         super().__init__(*args, **kwargs) # Inheriting from the parent
class
194
195     # Modifying and Overriding the run_training_loop_one_neuron_model to
implement SGD+
196     # mu is between [0,1]
197
198     def run_training_loop_multi_neuron_model(self, training_data, beta1
=0.9, beta2=0.999, e=1e-6):
199         #####Copied from the original
function#####
200         class DataLoader:
201             """
202             To understand the logic of the dataloader, it would help if
you first understand how
203             the training dataset is created. Search for the following
function in this file:
204
205             gen_training_data(self)
206
207             As you will see in the implementation code for this method,
the training dataset
208             consists of a Python dict with two keys, 0 and 1, the former
points to a list of
209             all Class 0 samples and the latter to a list of all Class 1
samples. In each list,
210             the data samples are drawn from a multi-dimensional Gaussian
distribution. The two
211             classes have different means and variances. The
dimensionality of each data sample
212             is set by the number of nodes in the input layer of the neural

```

```

network.
213
214     The data loader's job is to construct a batch of samples drawn
randomly from the two
215     lists mentioned above. And it must also associate the class
label with each sample
216     separately.
217     """
218     def __init__(self, training_data, batch_size):
219         self.training_data = training_data
220         self.batch_size = batch_size
221         self.class_0_samples = [(item, 0) for item in self.
training_data[0]]    ## Associate label 0 with each sample
222         self.class_1_samples = [(item, 1) for item in self.
training_data[1]]    ## Associate label 1 with each sample
223
224     def __len__(self):
225         return len(self.training_data[0]) + len(self.training_data
[1])
226
227     def _getitem(self):
228         cointoss = random.choice([0,1])
229         ## When a batch is created by getbatch(), we want the
230
231         ## samples to be chosen randomly from the two lists
232         if cointoss == 0:
233             return random.choice(self.class_0_samples)
234         else:
235             return random.choice(self.class_1_samples)
236
237     def getbatch(self):
238         batch_data, batch_labels = [], []
239         ## First list for samples, the second for labels
240         maxval = 0.0
241         ## For approximate batch data normalization
242         for _ in range(self.batch_size):
243             item = self._getitem()
244             if np.max(item[0]) > maxval:
245                 maxval = np.max(item[0])
246             batch_data.append(item[0])
247             batch_labels.append(item[1])
248         batch_data = [item/maxval for item in batch_data]
249         ## Normalize batch data
250         batch = [batch_data, batch_labels]
251         return batch
252
253     #
254     #####
255
256     # Modified part of the function
257     self.beta1, self.beta2 = beta1, beta2
258     self.e = e
259     self.m_db, self.v_db = 0.0, 0.0
260     self.m_dw = [0.0 for i in range(len(self.learnable_params) + 1)] #
m

```

```

253     self.v_dw = [0.0 for i in range(len(self.learnable_params) + 1)] #
254     v
255
256     #####Copied from the original
function#####
257     """
258     The training loop must first initialize the learnable parameters.
259     Remember, these are the
260     symbolic names in your input expressions for the neural layer that
261     do not begin with the
262     letter 'x'. In this case, we are initializing with random numbers
263     from a uniform distribution
264     over the interval (0,1).
265     """
266     self.vals_for_learnable_params = {param: random.uniform(0,1) for
param in self.learnable_params}
267
268     self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)
]
269     ## Adding the bias to each layer improves
270     ## class discrimination. We initialize it
271     ## to a random number.
272
273     data_loader = DataLoader(training_data, batch_size=self.batch_size
)
274
275     loss_running_record = []
276     i = 0
277     avg_loss_over_iterations = 0.0
278     ## Average the loss over iterations for printing out
279
280     ## every N iterations during the training loop.
281     for i in range(self.training_iterations):
282         data = data_loader.getbatch()
283         data_tuples = data[0]
284         class_labels = data[1]
285         self.forward_prop_multi_neuron_model(data_tuples)
286         ## FORW PROP works by side-effect
287         predicted_labels_for_batch = self.forw_prop_vals_at_layers[
self.num_layers-1]
288         ## Predictions from FORW PROP
289         y_preds = [item for sublist in predicted_labels_for_batch
for item in sublist]
290         ## Get numeric vals for predictions
291         loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
range(len(class_labels))])
292         ## Calculate loss for batch
293         loss_avg = loss / float(len(class_labels))
294         ## Average the loss over batch
295         avg_loss_over_iterations += loss_avg
296         ## Add to Average loss over iterations
297         if i%(self.display_loss_how_often) == 0:
298             avg_loss_over_iterations /= self.display_loss_how_often
299             loss_running_record.append(avg_loss_over_iterations)
300             print("[iter=%d] loss = %.4f" % (i+1,
avg_loss_over_iterations))
301             ## Display avg loss

```

```

288         avg_loss_over_iterations = 0.0
289         ## Re-initialize avg-over-iterations loss
290         y_errors = list(map(operator.sub, class_labels, y_preds))
291         y_error_avg = sum(y_errors) / float(len(class_labels))
292         self.backprop_and_update_params_multi_neuron_model(y_error_avg
, class_labels, i+1)      ## BACKPROP loss
293         # plt.figure()
294         # plt.plot(loss_running_record)
295         # plt.show()
296
297         return loss_running_record
298
299     #
300     #####
301
302     # Modify backpropagation function for one_neuron_model -
303     backpropagating the loss and updating the values of the learnable
304     parameters.
305     def backprop_and_update_params_multi_neuron_model(self, y_error,
306     class_labels, iteration):
307         """
308         First note that loop index variable 'back_layer_index' starts with
309         the index of
310         the last layer. For the 3-layer example shown for 'forward',
311         back_layer_index
312         starts with a value of 2, its next value is 1, and that's it.
313
314         Stochastic Gradient Gradient calls for the backpropagated loss to
315         be averaged over
316         the samples in a batch. To explain how this averaging is carried
317         out by the
318         backprop function, consider the last node on the example shown in
319         the forward()
320         function above. Standing at the node, we look at the 'input'
321         values stored in the
322         variable "input_vals". Assuming a batch size of 8, this will be
323         list of
324         lists. Each of the inner lists will have two values for the two
325         nodes in the
326         hidden layer. And there will be 8 of these for the 8 elements of
327         the batch. We average
328         these values 'input_vals' and store those in the variable "
329         input_vals_avg". Next we
330         must carry out the same batch-based averaging for the partial
331         derivatives stored in the
332         variable "deriv_sigmoid".
333
334         Pay attention to the variable 'vars_in_layer'. These store the
335         node variables in
336         the current layer during backpropagation. Since back_layer_index
337         starts with a
338         value of 2, the variable 'vars_in_layer' will have just the single
339         node for the

```

```

321     example shown for forward(). With respect to what is stored in
vars_in_layer', the
322     variables stored in 'input_vars_to_layer' correspond to the input
layer with
323     respect to the current layer.
324     """
325     # backproped prediction error:
326     pred_err_backproped_at_layers = {i : [] for i in range(1,self.
num_layers-1)}
327     pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
328     for back_layer_index in reversed(range(1,self.num_layers)):
329         input_vals = self.forw_prop_vals_at_layers[back_layer_index
-1]
330         input_vals_avg = [sum(x) for x in zip(*input_vals)]
331         input_vals_avg = list(map(operator.truediv, input_vals_avg, [
float(len(class_labels))] * len(class_labels)))
332         deriv_sigmoid = self.gradient_vals_for_layers[
back_layer_index]
333         deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
334         deriv_sigmoid_avg = list(map(operator.truediv,
deriv_sigmoid_avg,
335                                     [float(len(
class_labels))] * len(class_labels)))
336         vars_in_layer = self.layer_vars[back_layer_index]
337         ## a list like ['xo']
338         vars_in_next_layer_back = self.layer_vars[back_layer_index -
1]
339         ## a list like ['xw', 'xz']
340
341         layer_params = self.layer_params[back_layer_index]
342         ## note that layer_params are stored in a dict like
343         ## {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', '
bs']], 2: [['cp', 'cq']]}
344         ## "layer_params[idx]" is a list of lists for the link weights
in layer whose output nodes are in layer "idx"
345         transposed_layer_params = list(zip(*layer_params)) ##
creating a transpose of the link matrix
346
347         backproped_error = [None] * len(vars_in_next_layer_back)
348         for k,varr in enumerate(vars_in_next_layer_back):
349             for j,var2 in enumerate(vars_in_layer):
350                 backproped_error[k] = sum([self.
vals_for_learnable_params[transposed_layer_params[k][i]] *
351                 pred_err_backproped_at_layers[back_layer_index][i]
for i in range(len(
vars_in_layer))])
352                 deriv_sigmoid_avg[i] for i
in range(len(vars_in_layer))])
353         pred_err_backproped_at_layers[back_layer_index - 1] =
backproped_error
354         input_vars_to_layer = self.layer_vars[back_layer_index-1]
355         for j,var in enumerate(vars_in_layer):
356             layer_params = self.layer_params[back_layer_index][j]
357             ## Regarding the parameter update loop that follows, see

```

```

the Slides 74 through 77 of my Week 3
357         ## lecture slides for how the parameters are updated
using the partial derivatives stored away
358         ## during forward propagation of data. The theory
underlying these calculations is presented
359         ## in Slides 68 through 71.
360         for i,param in enumerate(layer_params):
361             gradient_of_loss_for_param = input_vals_avg[i] *
pred_err_backproped_at_layers[back_layer_index][j]
362             #####Modified
#####
363             self.m_dw[i + 1] = (self.beta1 * self.m_dw[i]) + (1 -
self.beta1) * (gradient_of_loss_for_param * deriv_sigmoid_avg[j])
364             self.m_dw[i] = self.m_dw[i + 1] # Save the new current
moment in the previous iteration position
365
366             self.v_dw[i + 1] = (self.beta2 * self.v_dw[i]) + (1 -
self.beta2) * (gradient_of_loss_for_param * deriv_sigmoid_avg[j]) ** 2
367             self.v_dw[i] = self.v_dw[i + 1] # Save the new current
moment in the previous iteration position
368
369             ## Update the learnable parameters
370             mk_hat = self.m_dw[i + 1] / (1 - self.beta1 **
iteration)
371             vk_hat = self.v_dw[i + 1] / (1 - self.beta2 **
iteration)
372
373             self.vals_for_learnable_params[param] += self.
learning_rate * mk_hat / np.sqrt(vk_hat + self.e)
374
375             # Inspired by: https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b217f1cc
376             # Inspired by: https://www.youtube.com/watch?v=JXQT\_vxqwIs&ab\_channel=DeepLearningAI
377             self.m_db = (self.beta1 * self.m_db) + (1 - self.beta1) * (sum
(pred_err_backproped_at_layers[back_layer_index]) \
378             * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))
379             self.v_db = (self.beta2 * self.v_db) + (1 - self.beta2) * (sum
(pred_err_backproped_at_layers[back_layer_index]) \
380             * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)) ** 2
381
382             m_db_hat = self.m_db / (1 - self.beta1 ** iteration)
383             v_db_hat = self.v_db / (1 - self.beta2 ** iteration)
384
385             self.bias[back_layer_index-1] += self.learning_rate * m_db_hat
/ np.sqrt(v_db_hat + self.e) ## Update the bias
386
387             #
#####
388
389 def sgd_plus(lr=1e-3, mu=0.9):

```

```

390     cgp = ComputationalGraphPrimerSGDPlus(
391         num_layers = 3,
392         layers_config = [4,2,1], # num of
nodes in each layer
393         expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
394                       'xz=bp*xp+bq*xq+br*xr+bs*xs',
395                       'xo=cp*xw+cq*xz'],
396         output_vars = ['xo'],
397         dataset_size = 5000,
398         learning_rate = lr,
399         training_iterations = 40000,
400         batch_size = 8,
401         display_loss_how_often = 100,
402         debug = True,
403     )
404
405     cgp.parse_multi_layer_expressions()
406     # cgp.display_multi_neuron_network()
407
408     training_data = cgp.gen_training_data()
409     loss_per_iteration = cgp.run_training_loop_multi_neuron_model(
training_data, mu)
410
411     return loss_per_iteration
412
413 def adam(lr=1e-3):
414     cgp = ComputationalGraphPrimerAdam(
415         num_layers = 3,
416         layers_config = [4,2,1], # num of
nodes in each layer
417         expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
418                       'xz=bp*xp+bq*xq+br*xr+bs*xs',
419                       'xo=cp*xw+cq*xz'],
420         output_vars = ['xo'],
421         dataset_size = 5000,
422         learning_rate = lr,
423         training_iterations = 40000,
424         batch_size = 8,
425         display_loss_how_often = 100,
426         debug = True,
427     )
428
429     cgp.parse_multi_layer_expressions()
430     # cgp.display_multi_neuron_network()
431
432     training_data = cgp.gen_training_data()
433     loss_per_iteration = cgp.run_training_loop_multi_neuron_model(
training_data )
434
435     return loss_per_iteration
436
437 def sgd(lr=1e-3):
438     cgp = ComputationalGraphPrimer(
439         num_layers = 3,

```



```

440         layers_config = [4,2,1],                                # num of
nodes in each layer
441         expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
442                        'xz=bp*xp+bq*xq+br*xr+bs*xs',
443                        'xo=cp*xw+cq*xz'],
444         output_vars = ['xo'],
445         dataset_size = 5000,
446         learning_rate = lr,
447         training_iterations = 40000,
448         batch_size = 8,
449         display_loss_how_often = 100,
450         debug = True,
451     )
452
453     cgp.parse_multi_layer_expressions()
454     training_data = cgp.gen_training_data()
455
456     loss_per_iteration = cgp.run_training_loop_multi_neuron_model(
training_data )
457     return loss_per_iteration
458
459 def plot_losses(sgd, sgd_plus, adam, lr):
460     number_of_iterations = len(adam)
461     plt.plot(range(number_of_iterations), sgd, label="SGD Loss")
462     plt.plot(range(number_of_iterations), sgd_plus, label="SGD+ Loss")
463     plt.plot(range(number_of_iterations), adam, label="Adam Loss")
464
465     plt.title(f"Loss per Iteration for Different Optimizers for Multi-
Neuron Model for Learning Rate: {lr}")
466     plt.xlabel("Iteration Number")
467     plt.ylabel("Loss")
468     plt.legend(loc="lower left")
469
470     plt.show(); quit()
471     plt.savefig(r"/Users/nikitaravi/Documents/Academics/ECE 60146/HW3/
multi_neuron_" + str(lr) + "_learning_rate.png", dpi=200)
472
473 if __name__ == "__main__":
474     lr = 1e-3
475     sgd_loss = sgd(lr)
476     sgd_plus_loss = sgd_plus(lr)
477     adam_loss = adam(lr)
478
479     plot_losses(sgd_loss, sgd_plus_loss, adam_loss, lr)

```

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

4 Results

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

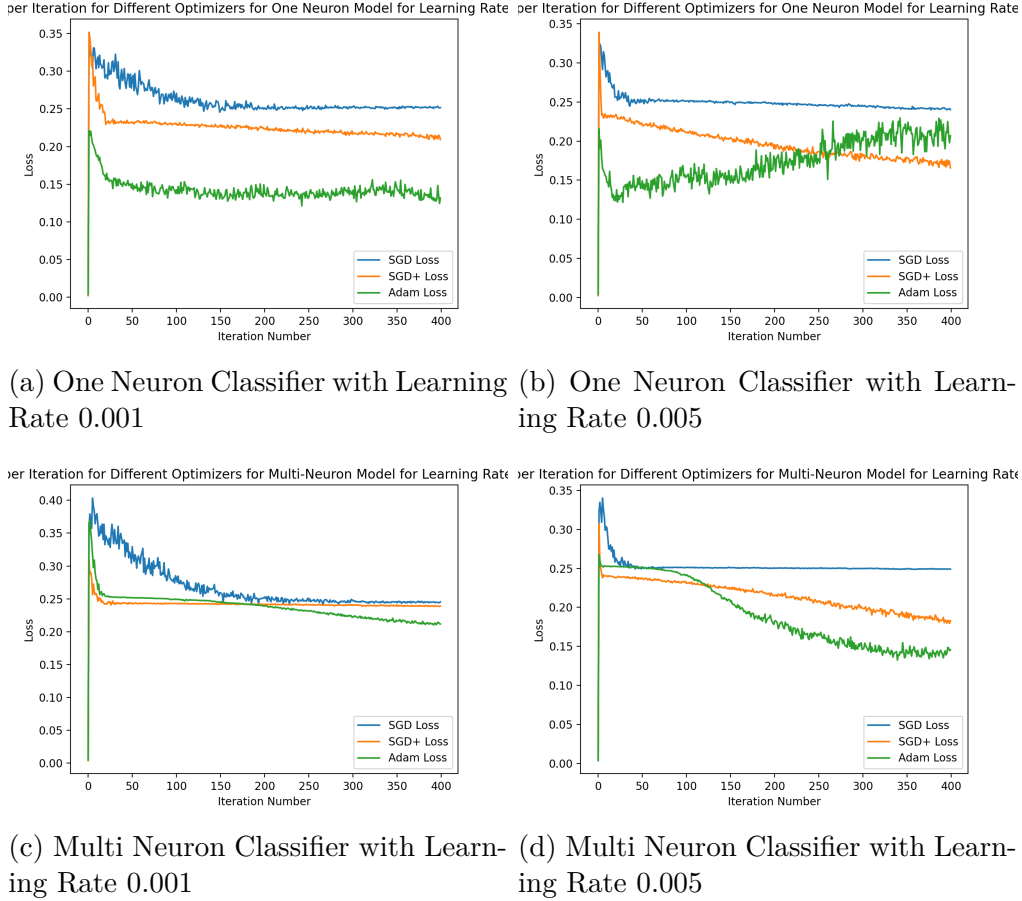


Figure 1: Losses per Iteration

5 Evaluation

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

In the multi-neuron classifier, the Adam optimizer significantly outperforms the SGD+ and SGD for both learning rates as it gets lower loss at each iteration. For the lower learning rate, both SGD+ and Adam are close to converging to a minimum whereas with the learning

rate of 0.005, both SGD+ and Adam keep oscillating and decreasing at each iteration, which suggests its taking longer to reach the minimum of the cost function.