

# ECE 60146 Homework 3

Arpita Rattan – rattan@purdue.edu

February 2nd 2025

## 1 Comparison of Hand-crafted vs Torch Neural Network

To compare the implementations of a hand-crafted neural network, vs one with a similar architecture implemented using the pytorch library we analyze the loss produced by both types of networks. Below the architecture for each network type can be seen.

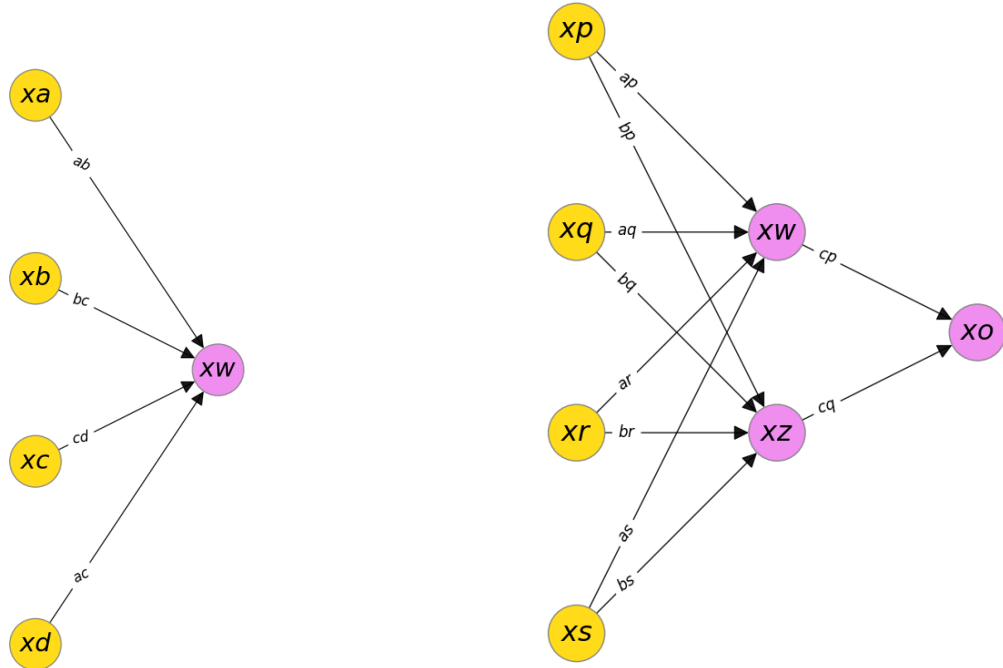


Figure 1: Network Structure of One-NN and Multi-NN

For the one neuron network, it can be seen that the hand crafted network ends with a better final loss value of about 0.2 compared to the torch implementation value of 1. This however does not mean it is a better implementation, as the torch implementation converges faster as seen by the curve in the decline of the loss, as compared to the almost linear decrease of the handcrafted loss.

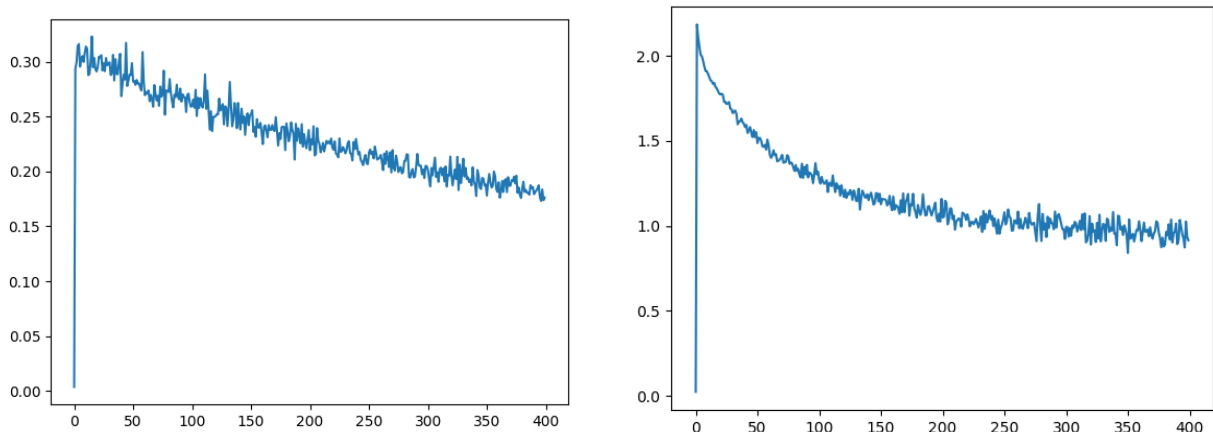


Figure 2: Handcrafted vs Torch Implementation of One Neuron

For the multi-neuron network, the handcrafted version displays an interesting loss function that starts to decrease, plateaus then exponentially decreases again. This is very unstable behavior that is not seen in the torch implementation, which displays a more linear decrease that has not plateau'd yet, meaning that further learning can take place. While the handcrafted version has a better rate of convergence, the torch method has a more stable approach as it reaches minimum loss.

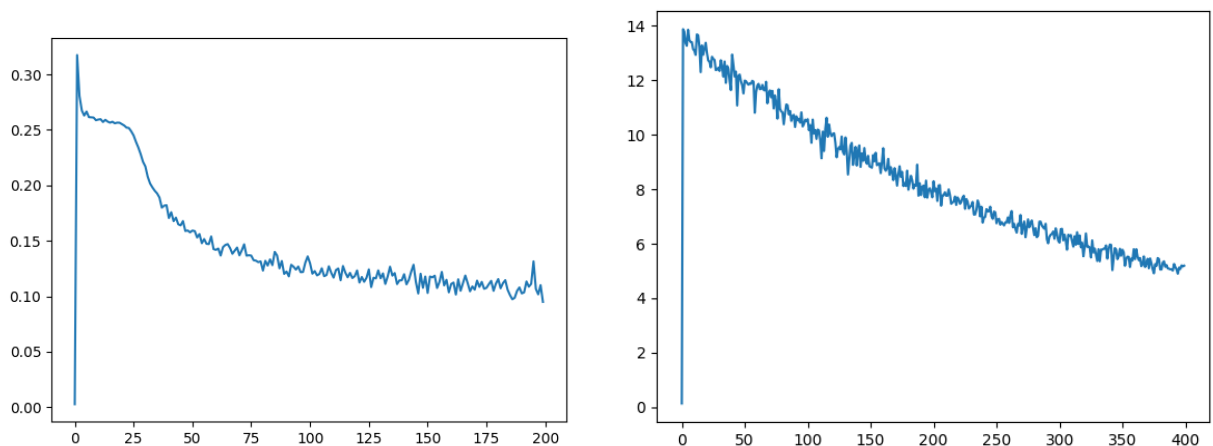


Figure 3: Handcrafted vs Torch Implementation of Multi-Neuron

## 2 Optimization Theory

### 2.1 SGD+ Optimizer

Gradient Descent suffers from oscillations when encountering valleys in the loss landscape, and can often get lost in local minima. To counteract this, SGD with Momentum incorporates a momentum term accelerates updates along consistent gradient directions while dampening oscillations in changing directions, creating a smoother convergence.

The update equations are:

$$v_{t+1} = \beta v_t + g_t \tag{1}$$

$$w_{t+1} = w_t - \eta \cdot v_{t+1} \tag{2}$$

where:

- $v_t$  is the velocity at step  $t$ ,
- $g_t$  is the gradient at  $t$ ,
- $w_t$  represents model weights,
- $\eta$  is the learning rate,
- $\beta$  is the momentum factor ( $0 \leq \beta \leq 1$ ).

A higher  $\beta$  gives more weight to past updates, leading to smoother and often faster convergence as it takes into account the optimization past.

### 2.2 ADAM Optimizer

While SGD+ accounts for first-order momentum, Adam extends this by incorporating second-order momentum as well. The first moment estimate (moving average of the gradient) is computed as:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_t \tag{3}$$

where  $\beta_1$  controls the decay rate of past gradients.

The second moment estimate (moving average of squared gradients) is:

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_t^2 \tag{4}$$

where  $\beta_2$  controls the decay of past squared gradients.

Since both  $m_t$  and  $v_t$  are biased towards zero in early training steps, we correct its bias using:

$$\hat{m}_{t+1} = \frac{m_{t+1}}{1 - \beta_1^t} \tag{5}$$

$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - \beta_2^t} \tag{6}$$

Finally, the model weights are updated using:

$$p_{t+1} = p_t - \frac{\eta \cdot m_{t+1}^{\hat{}}}{\sqrt{v_{t+1}^{\hat{}} + \epsilon}} \quad (7)$$

By considering both the mean and variance of past gradients, Adam adapts learning rates per parameter, leading to faster and more stable convergence compared to SGD+.

### 3 SGD+ Implementation

Implementing SGD+ into the existing computational graph primer class involved editing the 'Run training loop' and 'Backprop and update parameters' functions for both one single neuron and the multi-layer neural net. The following subsections highlight the individual implementations and results for each different architecture.

#### 3.1 One-NN Architecture

To alter the One-NN architecture, I updated the backprop function of SGD to include the momentum calculations as shown below:

```
...
self.weight_velocities[param] = beta * self.weight_velocities[param] +
    partial_of_loss_wrt_param
self.vals_for_learnable_params[param] += self.weight_velocities[param] *
    self.learning_rate

y_error_avg = sum(y_errors_in_batch) / float(self.batch_size)
deriv_sigmoid_avg = sum(deriv_sigmoids) / float(self.batch_size)

self.bias_velocity = (self.bias_velocity * beta) + (y_error_avg *
    deriv_sigmoid_avg)    ## Update the
                           bias including momentum now
self.bias += self.bias_velocity * self.learning_rate
```

Where the weight velocities and bias velocity are initialized in the training loop as:

```
# Create new variables to keep track of velocities: -----
self.weight_velocities = {param: 0 for param in self.learnable_params}
self.bias_velocity = 0
```

The results of this code for various Beta values can be seen in the plot below. The best performing Beta was 0.9 as seen by the fastest convergence to the lowest loss.

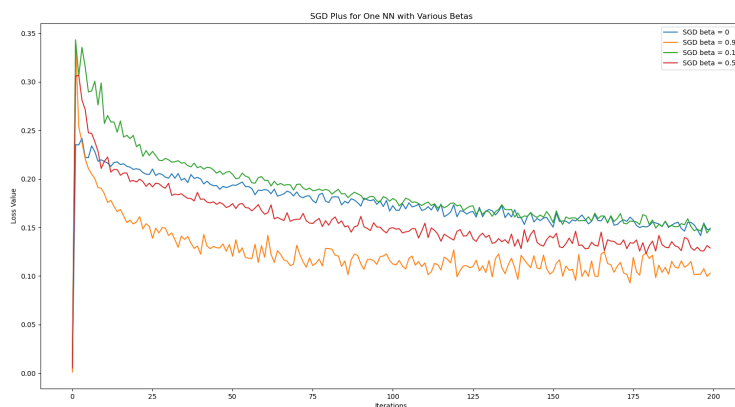


Figure 4: SGD+ - Loss vs Beta values for One-NN

## 3.2 Multi-NN Architecture

To alter the Multi-NN architecture, I first initialized variables to hold my velocities for each neuron, with the bias being applied to each individual neuron as defined by the layer config in the object initialization.

```
...
# Create new variables to keep track of velocities: -----
self.weight_velocities = {param: 0 for param in self.learnable_params}
self.bias_velocity = []
for layer_size in self.layers_config:
    self.bias_velocity.append(np.zeros(layer_size)) # individual bias for
                                                    each neuron in each layer
```

I further altered the code to include the weight and bias updates for each layer as required. The code for this can be seen below:

```
## Now update the learnable parameters. The loop shown below carries out
                                SGD mandated averaging
for param in partial_of_loss_wrt_params:
    partial_of_loss_wrt_param = partial_of_loss_wrt_params[param] / float
                                (self.batch_size)

# Update parameters using SGD Plus
self.weight_velocities[param] = beta * self.weight_velocities[param] +
                                partial_of_loss_wrt_param
self.vals_for_learnable_params[param] += self.weight_velocities[param]
                                * self.learning_rate

## Finally we update the biases at all the nodes that aggregate data:
for layer_index in range(1, self.num_layers):
    for k in range(self.layers_config[layer_index]):
        self.bias_velocity[layer_index][k] = (self.bias_velocity[
                                layer_index][k] * beta) + (
                                bias_changes[layer_index][k]
                                / float(self.batch_size))
                                ## Update the bias including
                                momentum now
        self.bias[layer_index][k] += self.learning_rate * self.
                                bias_velocity[layer_index][k]
```

The results for SGD plus as applied to the multi-neuron network can be seen below, with a beta value of 0.5 performing the best and converging the fastest, while the higher beta value such as 0.9 converges more smoothly.

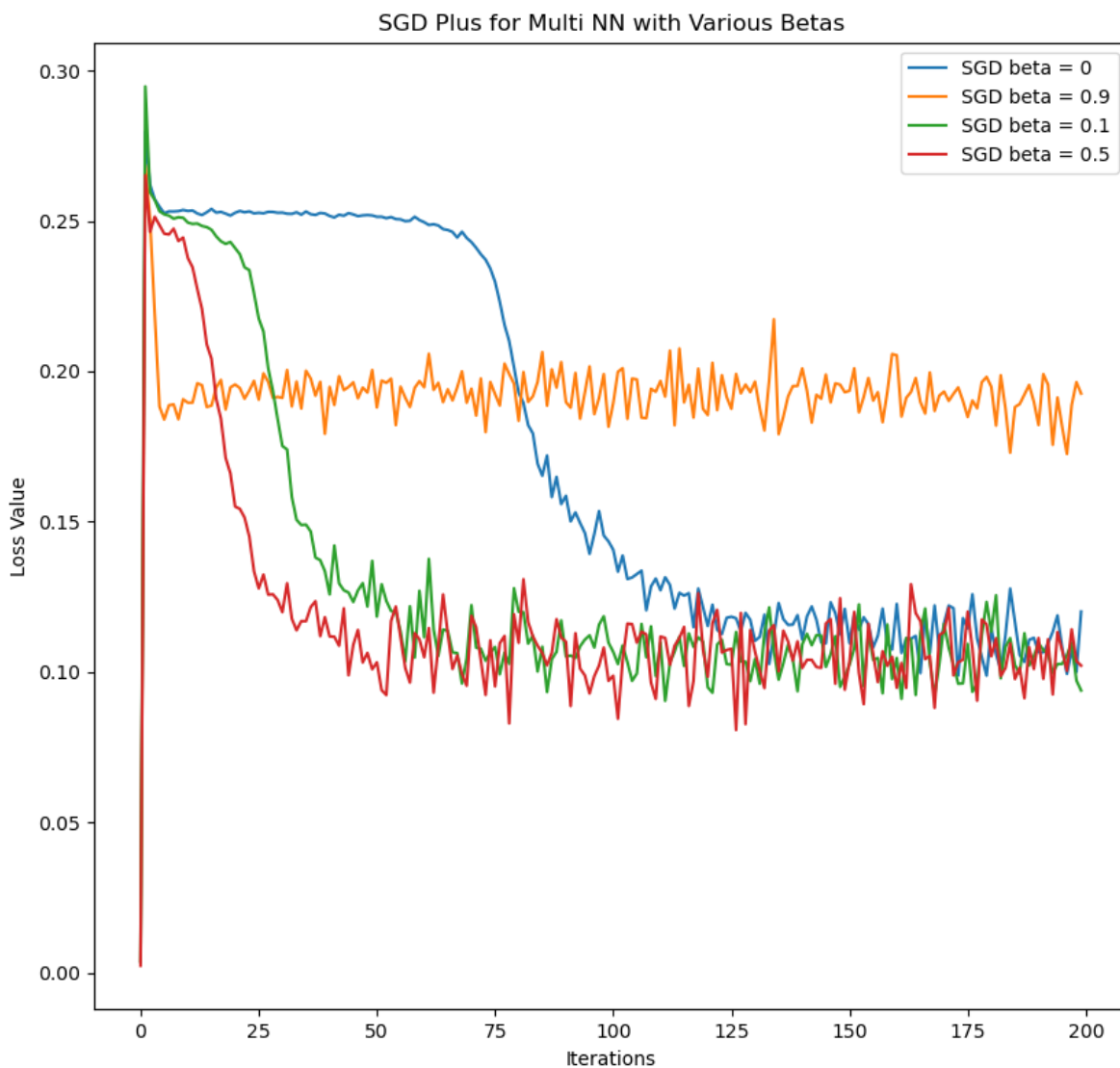


Figure 5: SGD+ - Loss vs Beta values for Multi-NN

## 4 ADAM Implementation

Similar to the implementation of SGD+, I altered the existing computational graph primer class by editing the 'Run training loop' and 'Backprop and update parameters' functions for both one single neuron and the multi-layer neural net to now include the appropriate formulae for ADAM optimizer with two moments. The following subsections highlight the individual implementations and results for each different architecture.

### 4.1 One-NN Architecture

To update the training and backprop loops to include ADAM, I first initialize the momentums as follows:

```
...
# To keep track of both types of moments:
self.m_moments, self.v_moments = {param: 0 for param in self.
                                   learnable_params}, {param: 0 for
                                                         param in self.learnable_params}
self.k = 0 # Timestep initialization
```

And then further updated the weights and biases using the ADAM formulae as follows:

```
...
self.k += 1 # Update for each timestep
...
# Update learnable params using both moments
self.m_moments[param] = (self.m_moments[param] * beta1) + (1 - beta1)
                        * partial_of_loss_wrt_param # m(t
                        +1) Update formula
self.v_moments[param] = (self.v_moments[param] * beta2) + ((1 - beta2)
                        * (partial_of_loss_wrt_param **
                        2)) # v(t+1) Update formula

mhat = self.m_moments[param] / (1 - beta1 ** self.k) # Unbias the
                                                moments from 0
vhat = self.v_moments[param] / (1 - beta2 ** self.k)

step = self.learning_rate * (mhat / math.sqrt(vhat + self.epsilon))
self.vals_for_learnable_params[param] -= step

y_error_avg = sum(y_errors_in_batch) / float(self.batch_size)
deriv_sigmoid_avg = sum(deriv_sigmoids) / float(self.batch_size)
self.bias += step + self.learning_rate * y_error_avg * deriv_sigmoid_avg
## Update the bias
```

I ran various experiments with many different values for Beta, the following graph shows 9 different combinations of the following beta values:

$$\beta_1 = [0.8, 0.95, 0.99] \beta_2 = [0.85, 0.9, 0.95]$$

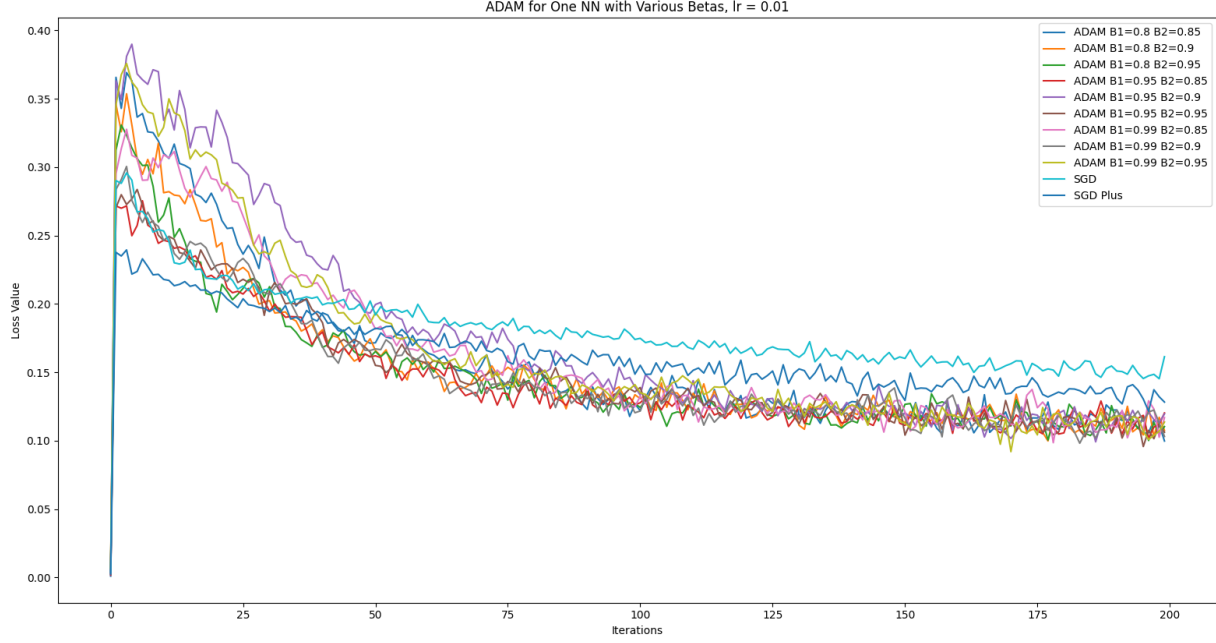


Figure 6: ADAM - Loss vs Beta values for One-NN,  $lr = 1e-2$

The table shows the minimum loss, final loss, and time taken to run each training for each Beta combination. The combination  $\beta_1 = 0.95$ ,  $\beta_2 = 0.85$  can be seen in the graph to give the best optimization, as it converges faster than SGDplus to a lower loss.

| B1   | B2   | Min Loss | Final Loss | Time Taken (s) |
|------|------|----------|------------|----------------|
| 0.8  | 0.85 | 0.1006   | 0.1290     | 3.7616         |
| 0.8  | 0.9  | 0.0972   | 0.1042     | 3.3658         |
| 0.8  | 0.95 | 0.1076   | 0.1119     | 3.8178         |
| 0.95 | 0.85 | 0.1032   | 0.1124     | 4.3197         |
| 0.95 | 0.9  | 0.1036   | 0.1112     | 3.7547         |
| 0.95 | 0.95 | 0.1065   | 0.1088     | 3.5935         |
| 0.99 | 0.85 | 0.1071   | 0.1244     | 3.7211         |
| 0.99 | 0.9  | 0.1043   | 0.1215     | 3.6014         |
| 0.99 | 0.95 | 0.1040   | 0.1219     | 3.5244         |

Table 1: Loss and time taken for different values of B1 and B2

I also experimented with various learning rates as can be seen below, but our initial lr of  $1e-2$  shows the best convergence to the lowest learning rate.

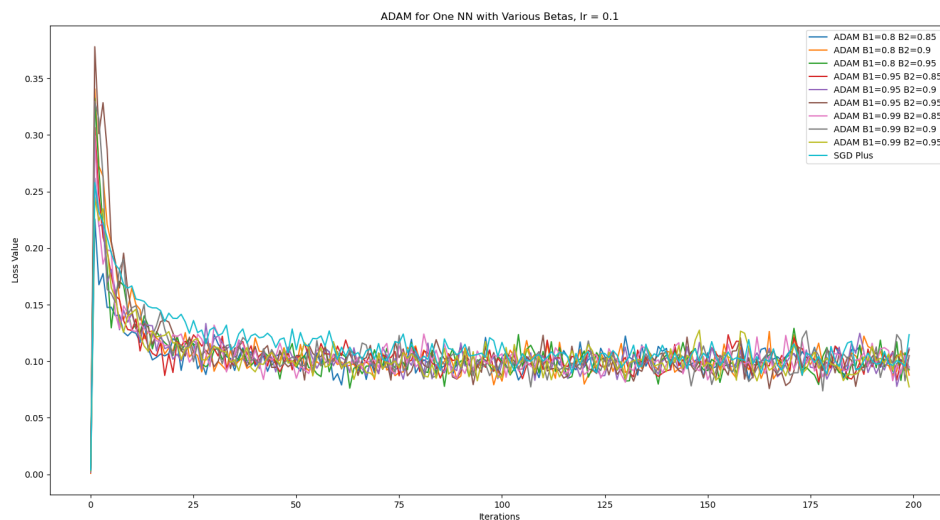


Figure 7: ADAM - Loss vs Beta values for One-NN, lr =  $1e-1$

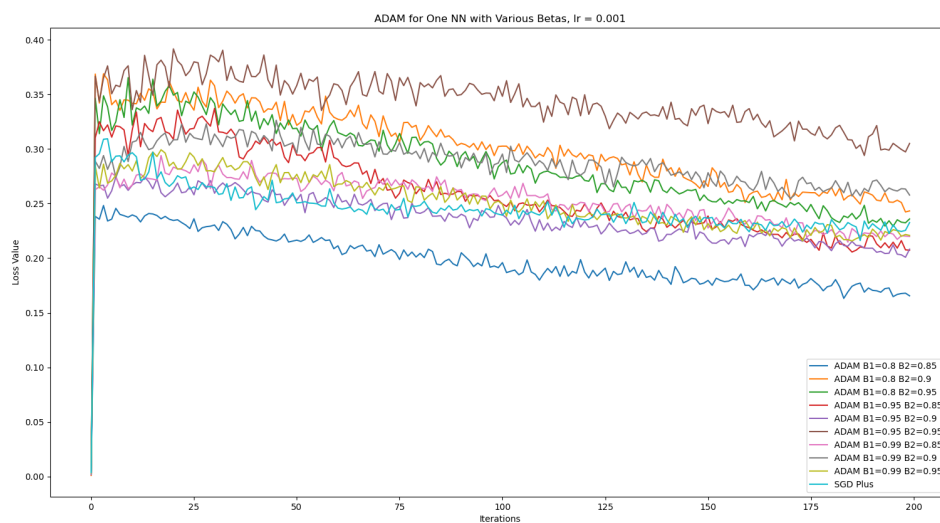


Figure 8: ADAM - Loss vs Beta values for One-NN, lr =  $1e-3$

The final configurations for the One neuron architecture were conducted with an SGD+ beta value of 0.9, and ADAM values of 0.95 and 0.85. The below graph shows the loss comparison. As expected, vanilla SGD performs the worst, with SGD having a smoother convergence and then ADAM having a mixture of better smoothness as well as a faster convergence.



Figure 9: Final One Neuron Comparison

## 4.2 Multi-NN Architecture

To update the code for multi neuron, I updated the training loop and the backpropagation functions to include the following:

```
# To keep track of both types of moments:
self.m_moments, self.v_moments = {param: 0 for param in self.
                                   learnable_params}, {param: 0 for
                                   param in self.learnable_params}

self.k = 0 # Timestep initialization
self.m_bias, self.v_bias = [], []

for layer_size in self.layers_config:
    self.m_bias.append(np.zeros(layer_size)) # individual bias for each
                                              neuron in each layer
    self.v_bias.append(np.zeros(layer_size))
```

The code above initializes layer wise moments for each neuron. The code below actually performs the gradient update:

```
## Now update the learnable parameters. The loop shown below carries out
                                   SGD mandated averaging
for param in partial_of_loss_wrt_params:
    partial_of_loss_wrt_param = partial_of_loss_wrt_params[param] / float
                               (self.batch_size)

    # Update params using ADAM Formulas =====
    self.m_moments[param] = (self.m_moments[param] * beta1) + ((1 - beta1)
                                                                * partial_of_loss_wrt_param) # m
                                                                (t+1) Update formula
    self.v_moments[param] = (self.v_moments[param] * beta2) + ((1 - beta2)
                                                                * (partial_of_loss_wrt_param **
                                                                2)) # v(t+1) Update formula

    mhat = self.m_moments[param] / (1 - beta1 ** self.k) # Unbias the
                                                         moments from 0
    vhat = self.v_moments[param] / (1 - beta2 ** self.k)

    step = self.learning_rate * (mhat / math.sqrt(vhat + self.epsilon))
    self.vals_for_learnable_params[param] += step

## Finally we update the biases at all the nodes that aggregate data:
for layer_index in range(1, self.num_layers):
    for k in range(self.layers_config[layer_index]):
        gradt = bias_changes[layer_index][k] # / float(self.batch_size)
        self.m_bias[layer_index][k] = beta1 * self.m_bias[layer_index][k]
                                         + (1-beta1) * gradt
        self.v_bias[layer_index][k] = beta2 * self.v_bias[layer_index][k]
                                         + (1-beta2) * (gradt ** 2)

        mhat = self.m_bias[layer_index][k] / (1 - beta1 ** self.k) #
                                                Unbias the moments from 0
        vhat = self.v_bias[layer_index][k] / (1 - beta2 ** self.k)
```

```

step = self.learning_rate * (mhat / math.sqrt(vhat + self.epsilon)
                             )
self.bias[layer_index][k] -= self.learning_rate * step

```

The graph below shows my comparison of the various combinations of betas as previously seen in the One neuron case. According to this graph, we select the combination of 0.99 and 0.9 as our best ADAM optimized model.

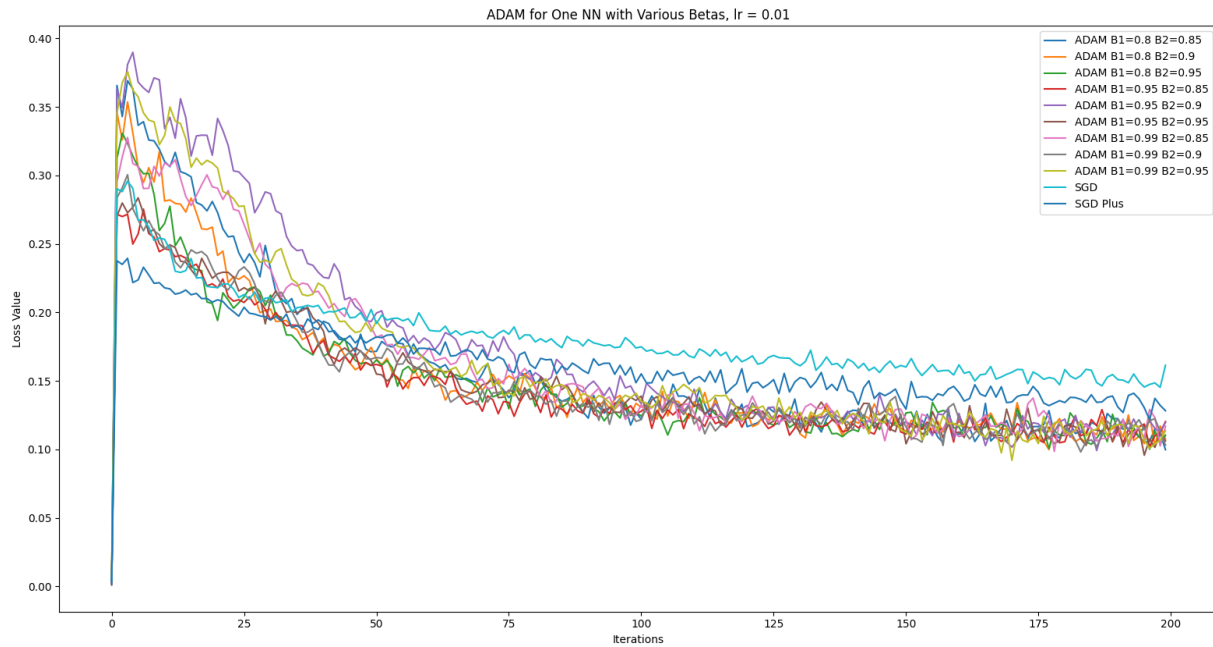


Figure 10: ADAM - Loss vs Beta values for Multi-NN, lr = 1e-2

The following table shows the various data gathered from the above experiment. One thing of note when comparing this to the One neuron case is that I ran this on a different machine, hence attributing to the longer run time.

| B1   | B2   | Min Loss | Final Loss | Time Taken (s) |
|------|------|----------|------------|----------------|
| 0.8  | 0.85 | 0.0814   | 0.1066     | 22.5600        |
| 0.8  | 0.9  | 0.1152   | 0.1205     | 22.3222        |
| 0.8  | 0.95 | 0.1110   | 0.1193     | 22.0714        |
| 0.95 | 0.85 | 0.1522   | 0.1673     | 22.2502        |
| 0.95 | 0.9  | 0.1208   | 0.1373     | 22.8845        |
| 0.95 | 0.95 | 0.1228   | 0.1330     | 22.1774        |
| 0.99 | 0.85 | 0.1111   | 0.1256     | 22.3001        |
| 0.99 | 0.9  | 0.0818   | 0.1049     | 22.1742        |
| 0.99 | 0.95 | 0.0795   | 0.0878     | 21.8879        |

Table 2: Loss and time taken for different values of B1 and B2

Finally, the configuration to compare the optimizers involved using a beta value of 0.5 for SGD+ and values 0.9, 0.99 for ADAM. The graph clearly shows how ADAM is superior in how it converges quicker and to a lower loss, and is followed by SGD+, which while it converges to a similar final loss, takes about 90 more iterations to do so.

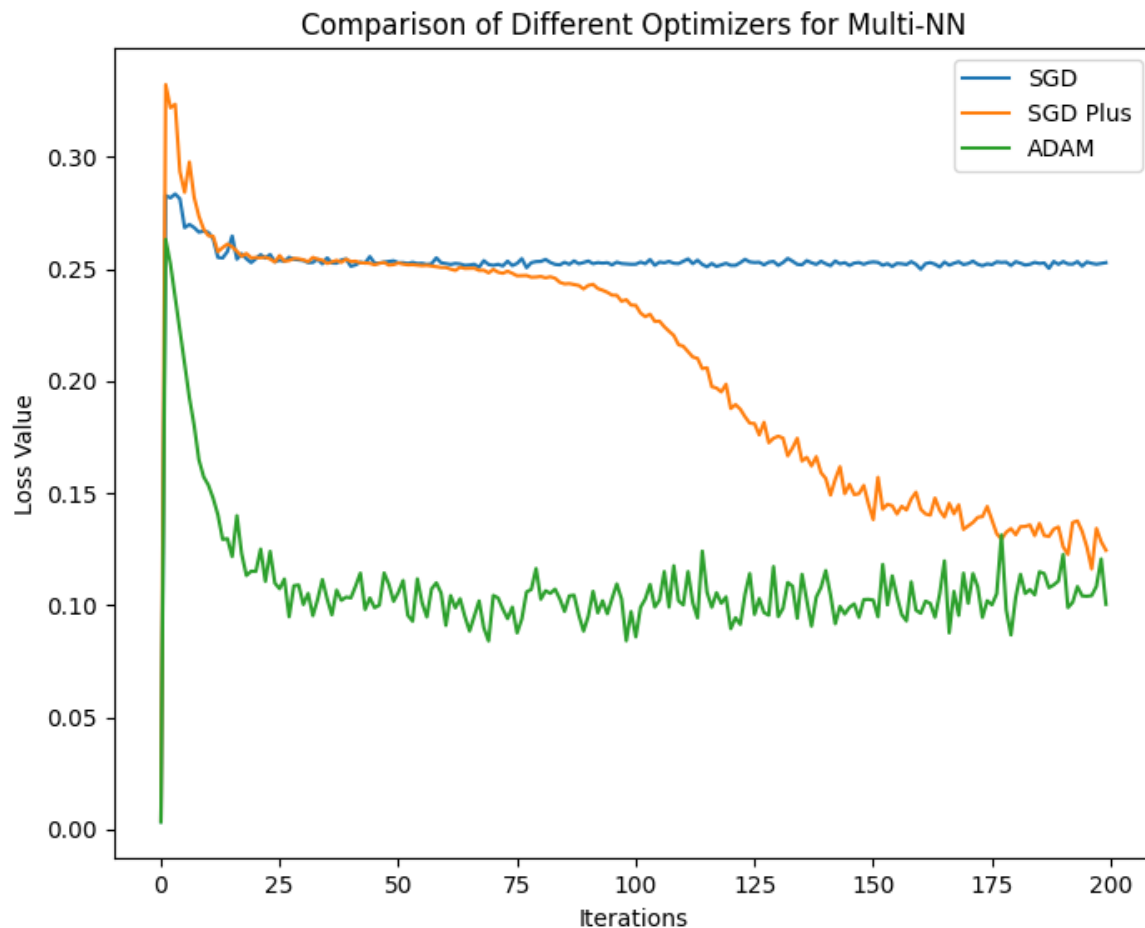


Figure 11: Final Multi Neuron Comparison

## 5 Extra Credit

### 5.1 Comparison of Normalization Effect

For the first part of the extra credit, we compare the effect of data normalization of the batches on the loss values. I implemented a normalization flag that triggers the appropriate line of code, and below can be seen the results for both one neuron and multi neuron networks. As seen clearly, the loss is much lower for the network operating on normalized data instead of it not being normalized, as well as having it converge faster.

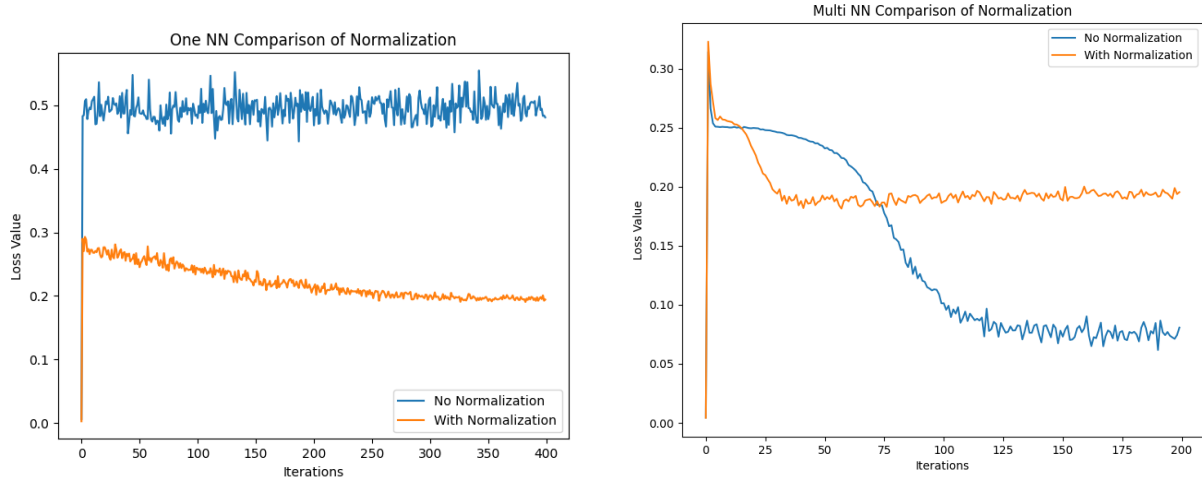


Figure 12: Comparison of Effects of Data Normalization

## 5.2 Manual Truncation

For manual truncation and remapping to  $[-1, 1]$  I first found out the mean and standard deviation of the Gaussian for each class, then performed the truncation. I then applied the remapping using min max normalization. The results of this can be seen below, the graphs show that truncation does not have an effect on the data when normalization is already taking place.

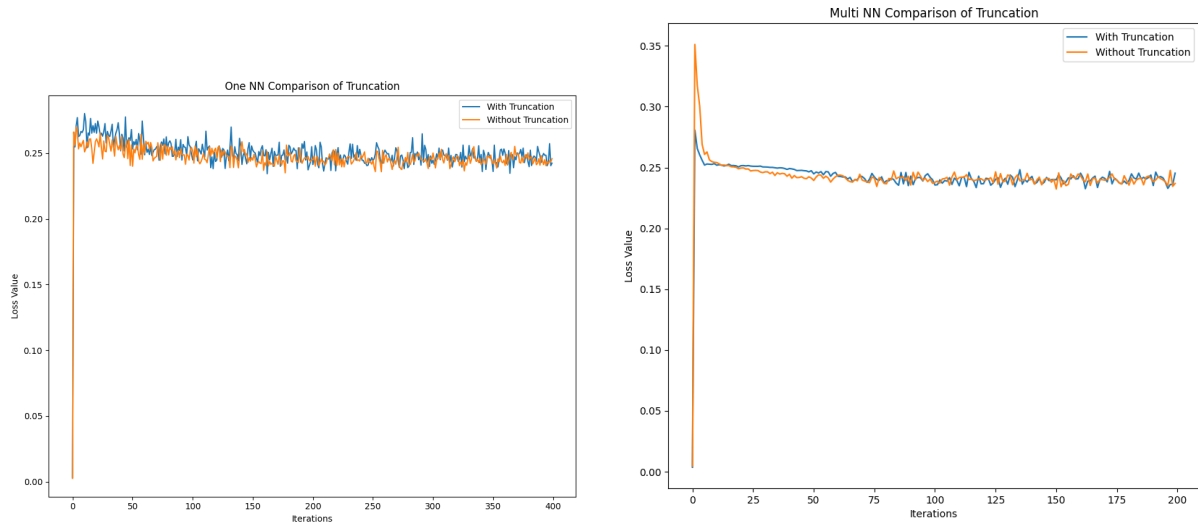


Figure 13: Comparison of Effects of Truncation

## 6 Source Code

### hw3.py

```
1 from ComputationalGraphPrimer import ComputationalGraphPrimer
2 import numpy as np
3 import random, operator, math, time
4 import matplotlib.pyplot as plt
5 from tqdm import tqdm
6
7 random.seed(0)
8 np.random.seed(0)
9
10
11 class SGDplus(ComputationalGraphPrimer):
12     def __init__(self, *args, **kwargs): # Now includes beta factor for
13         momentum, and run plus option so we can get back vanilla sgd
14         super().__init__(*args, **kwargs)
15
16     def run_training_loop_one_neuron_model(self, training_data, beta):
17         # Initialize learnable params for network -----
18         self.vals_for_learnable_params = {param: random.uniform(0,1) for
19 param in self.learnable_params}
20         self.bias = random.uniform(0,1)
21
22         # Create new variables to keep track of velocities:
23         -----
24         self.weight_velocities = {param: 0 for param in self.
25 learnable_params}
26         self.bias_velocity = 0
27
28         # Dataloader business -----
29         class DataLoader:
30             def __init__(self, training_data, batch_size):
31                 self.training_data = training_data
32                 self.batch_size = batch_size
33                 self.class_0_samples = [(item, 0) for item in self.
34 training_data[0]] ## Associate label 0 with each sample
35                 self.class_1_samples = [(item, 1) for item in self.
36 training_data[1]] ## Associate label 1 with each sample
37
38             def __len__(self):
39                 return len(self.training_data[0]) + len(self.training_data
40 [1])
41
42             def __getitem__(self):
43                 cointoss = random.choice([0,1])
44                 ## When a batch is created by getbatch(), we want the
45                 ## samples to be chosen randomly from the two lists
46                 if cointoss == 0:
```

```

39         return random.choice(self.class_0_samples)
40     else:
41         return random.choice(self.class_1_samples)
42
43     def getbatch(self):
44         batch_data, batch_labels = [], []
45         ## First list for samples, the second for labels
46         maxval = 0.0
47         ## For approximate batch data normalization
48         for _ in range(self.batch_size):
49             item = self._getitem()
50             if np.max(item[0]) > maxval:
51                 maxval = np.max(item[0])
52             batch_data.append(item[0])
53             batch_labels.append(item[1])
54         batch_data = [item/maxval for item in batch_data]
55         ## Normalize batch data
56         batch = [batch_data, batch_labels]
57         return batch
58
59     data_loader = DataLoader(training_data, batch_size=self.batch_size
60 )
61
62     loss_running_record = []
63     i = 0
64     avg_loss_over_iterations = 0.0
65     ## Average the loss over iterations for printing out
66
67     # Actually running training -----
68     for i in range(self.training_iterations):
69         data = data_loader.getbatch()
70         data_tuples_in_batch = data[0]
71         class_labels_in_batch = data[1]
72
73         y_preds, deriv_sigmoids = self.forward_prop_one_neuron_model(
74 data_tuples_in_batch)    ## FORWARD PROP of data
75         loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
76 for i in range(len(class_labels_in_batch))])    ## Find loss
77         avg_loss_over_iterations += loss / float(len(
78 class_labels_in_batch))
79
80         if i%(self.display_loss_how_often) == 0:
81             avg_loss_over_iterations /= self.display_loss_how_often
82             loss_running_record.append(avg_loss_over_iterations)
83             # print("[iter=%d]  loss = %.4f" % (i+1,
84 avg_loss_over_iterations))    ## Display average loss
85             avg_loss_over_iterations = 0.0
86             ## Re-initialize avg loss
87
88         y_errors_in_batch = list(map(operator.sub,
89 class_labels_in_batch, y_preds))

```

```

78         self.backprop_and_update_params_one_neuron_model(
data_tuples_in_batch, y_preds, y_errors_in_batch, deriv_sigmoids, beta)
    ## BACKPROP loss
79     return loss_running_record # Return loss for own plotting function
80
81     def backprop_and_update_params_one_neuron_model(self,
data_tuples_in_batch, predictions, y_errors_in_batch, deriv_sigmoids,
beta):
82
83         input_vars = self.independent_vars
84         input_vars_to_param_map = self.var_to_var_param[self.output_vars
[0]]
    ## These two statements align the input vars
85         param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()}
86
87         for i,param in enumerate(self.vals_for_learnable_params):
88             ## For each param, sum the partials from every training data
sample in batch
89             partial_of_loss_wrt_param = 0.0
90             for j in range(self.batch_size):
91                 vals_for_input_vars_dict = dict(zip(input_vars, list(
data_tuples_in_batch[j])))
92                 partial_of_loss_wrt_param += y_errors_in_batch[j] *
vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[j]
93                 partial_of_loss_wrt_param /= float(self.batch_size)
94
95             ##====Update Parameters using SGD Plus=====
96             self.weight_velocities[param] = beta * self.weight_velocities[
param] + partial_of_loss_wrt_param
97             self.vals_for_learnable_params[param] += self.
weight_velocities[param] * self.learning_rate
98
99             y_error_avg = sum(y_errors_in_batch) / float(self.batch_size)
100             deriv_sigmoid_avg = sum(deriv_sigmoids) / float(self.batch_size)
101
102             #self.bias_velocity = (self.bias_velocity * beta) + (y_error_avg *
deriv_sigmoid_avg)    ## Update the bias including momentum now
103             self.bias += (self.bias_velocity * beta) + (self.learning_rate *
y_error_avg * deriv_sigmoid_avg)
104
105     def run_training_loop_multi_neuron_model(self, training_data, beta):
106
107         class DataLoader:
108             def __init__(self, training_data, batch_size):
109                 self.training_data = training_data
110                 self.batch_size = batch_size
111                 self.class_0_samples = [(item, 0) for item in self.
training_data[0]]
112                 self.class_1_samples = [(item, 1) for item in self.
training_data[1]]
113

```

```

114         def __len__(self):
115             return len(self.training_data[0]) + len(self.training_data
116 [1])
117
118         def _getitem(self):
119             cointoss = random.choice([0,1])
120             ## When a batch is created by getbatch(), we want the
121             ## samples to be chosen randomly from the two lists
122             if cointoss == 0:
123                 return random.choice(self.class_0_samples)
124             else:
125                 return random.choice(self.class_1_samples)
126
127         def getbatch(self):
128             batch_data, batch_labels = [], []
129             ## First list for samples, the second for labels
130             maxval = 0.0
131             ## For approximate batch data normalization
132             for _ in range(self.batch_size):
133                 item = self._getitem()
134                 if np.max(item[0]) > maxval:
135                     maxval = np.max(item[0])
136                 batch_data.append(item[0])
137                 batch_labels.append(item[1])
138             batch_data = [item/maxval for item in batch_data]
139             ## Normalize batch data
140             batch = [batch_data, batch_labels]
141             return batch
142
143         # Initialize Learnable params
144         self.vals_for_learnable_params = {param: random.uniform(0,1) for
145 param in self.learnable_params}
146         self.bias = {i : [random.uniform(0,1) for j in range( self.
147 layers_config[i] ) ] for i in range(1, self.num_layers)}
148         data_loader = DataLoader(training_data, batch_size=self.batch_size
149 )
150         loss_running_record = []
151         i = 0
152         avg_loss_over_iterations = 0.0
153
154         # Create new variables to keep track of velocities:
155         -----
156         self.weight_velocities = {param: 0 for param in self.
157 learnable_params}
158         self.bias_velocity = []
159         for layer_size in self.layers_config:
160             self.bias_velocity.append(np.zeros(layer_size)) # individual
161 bias for each neuron in each layer
162
163         for i in range(self.training_iterations):

```

```

153         data = data_loader.getbatch()
154         data_tuples = data[0]
155         class_labels = data[1]
156         self.forward_prop_multi_neuron_model(data_tuples)
157             ## FORW PROP works by side-effect
158         predicted_labels_for_batch = self.forw_prop_vals_at_layers[
159 self.num_layers-1]             ## Predictions from FORW PROP
160         y_preds = [item for sublist in predicted_labels_for_batch
161 for item in sublist]             ## Get numeric vals for predictions
162         loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
163 range(len(class_labels))])             ## Calculate loss for batch
164         loss_avg = loss / float(len(class_labels))
165             ## Average the loss over batch
166         avg_loss_over_iterations += loss_avg
167             ## Add to Average loss over iterations
168         if i%(self.display_loss_how_often) == 0:
169             avg_loss_over_iterations /= self.display_loss_how_often
170             loss_running_record.append(avg_loss_over_iterations)
171             # print("[iter=%d] loss = %.4f" % (i+1,
172 avg_loss_over_iterations))
173             avg_loss_over_iterations = 0.0
174         y_errors_in_batch = list(map(operator.sub, class_labels,
175 y_preds))
176         self.backprop_and_update_params_multi_neuron_model(y_preds,
177 y_errors_in_batch, beta)
178         return loss_running_record
179
180     def backprop_and_update_params_multi_neuron_model(self, predictions,
181 y_errors, beta):
182         ## Eq. (24) on Slide 73 of my Week 3 lecture says we need to store
183         ## backproped errors in each layer leading up to the last:
184         pred_err_backproped_at_layers = [ {i : [None for j in range(
185 self.layers_config[i] ) ]
186                                     for i in
187 range(self.num_layers)} for _ in range(self.batch_size) ]
188         ## This will store "\delta L / \delta w" you see at the LHS of the
189         ## equations on Slide 73:
190         partial_of_loss_wrt_params = {param : 0.0 for param in self.
191 all_params}
192         ## For estimating the changes to the bias to be made on the basis
193         ## of the derivatives of the Sigmoids:
194         bias_changes = {i : [0.0 for j in range( self.layers_config[i] )
195 ] for i in range(1, self.num_layers)}
196
197         for b in range(self.batch_size):
198             pred_err_backproped_at_layers[b][self.num_layers - 1] = [
199 y_errors[b] ]
200             for back_layer_index in reversed(range(1,self.num_layers)):
201                 ## For the 3-layer network, the first val for
202                 ## back_layer_index is 2 for the 3rd layer

```

```

183         input_vals = self.forw_prop_vals_at_layers[
back_layer_index - 1]      ## This is a list of 8 two-element lists ---
since we have two nodes in the 2nd layer
184         deriv_sigmoids = self.gradient_vals_for_layers[
back_layer_index]      ## This is a list eight one-element lists, one for
each batch element
185         vars_in_layer = self.layer_vars[back_layer_index]
## A list like ['xo']
186         vars_in_next_layer_back = self.layer_vars[
back_layer_index - 1]      ## A list like ['xw', 'xz']
187         vals_for_input_vars_dict = dict(zip(
vars_in_next_layer_back, self.forw_prop_vals_at_layers[back_layer_index
- 1][b]))
188         ## For the next statement, note that layer_params are
stored in a dict like
189         ##      {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br',
'bs']], 2: [['cp', 'cq']]}
190         ## "layer_params[idx]" is a list of lists for the link
weights in layer whose output nodes are in layer "idx"
191         layer_params = self.layer_params[back_layer_index]
192         transposed_layer_params = list(zip(*layer_params))
## Creating a transpose of the link matrix, See Eq. 30 on
Slide 77
193         for k, var1 in enumerate(vars_in_next_layer_back):
194             for j, var2 in enumerate(vars_in_layer):
195                 pred_err_backproped_at_layers[b][back_layer_index
- 1][k] = sum([self.vals_for_learnable_params[transposed_layer_params[k
][i]]
196
197                 * pred_err_backproped_at_layers[b][back_layer_index][i]
198
199                 for i in range(len(
vars_in_layer))])
200         for j, var in enumerate(vars_in_layer):
201             layer_params = self.layer_params[back_layer_index][j]
202             ## ['cp', 'cq'] for the end layer
203             input_vars_to_param_map = self.var_to_var_param[var]
204             ## These two statements align the      {'xw': 'cp', 'xz': 'cq'}
205             param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()}      ## and the input vars      {'cp': 'xw
', 'cq': 'xz'}
206
207             ## Update the partials of Loss wrt to the learnable
parameters between the current layer
208             ## and the previous layer. You are accumulating these
partials over the different training
209             ## data samples in the batch being processed. For
each training data sample, the formula
210             ## being used is shown in Eq. (29) on Slide 77 of my
Week 3 slides:
211             for i, param in enumerate(layer_params):

```

```

208         partial_of_loss_wrt_params[param] +=
pred_err_backproped_at_layers[b][back_layer_index][j] * \
209
vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[b][
j]
210         ## We will now estimate the change in the bias that needs
to be made at each node in the previous layer
211         ## from the derivatives the sigmoid at the nodes in the
current layer and the prediction error as
212         ## backproped to the previous layer nodes:
213         for k,var1 in enumerate(vars_in_next_layer_back):
214             for j,var2 in enumerate(vars_in_layer):
215                 if back_layer_index-1 > 0:
216                     bias_changes[back_layer_index-1][k] +=
pred_err_backproped_at_layers[b][back_layer_index - 1][k] *
deriv_sigmoids[b][j]
217
218         ## Now update the learnable parameters. The loop shown below
carries out SGD mandated averaging
219         for param in partial_of_loss_wrt_params:
220             partial_of_loss_wrt_param = partial_of_loss_wrt_params[param]
/ float(self.batch_size)
221
222         # Update parameters using SGD Plus
223         self.weight_velocities[param] = beta * self.weight_velocities[
param] + partial_of_loss_wrt_param
224         self.vals_for_learnable_params[param] += self.
weight_velocities[param] * self.learning_rate
225
226         ## Finally we update the biases at all the nodes that aggregate
data:
227         for layer_index in range(1,self.num_layers):
228             for k in range(self.layers_config[layer_index]):
229                 self.bias_velocity[layer_index][k] = (self.bias_velocity[
layer_index][k] * beta) + (bias_changes[layer_index][k] / float(self.
batch_size)) ## Update the bias including momentum now
230                 self.bias[layer_index][k] += self.learning_rate * self.
bias_velocity[layer_index][k]
231
232
233 class ADAMOptimizer(ComputationalGraphPrimer):
234     def __init__(self, *args, **kwargs):
235         super().__init__(*args, **kwargs)
236         self.epsilon = 1e-8 # for division by zero
237
238     def run_training_loop_one_neuron_model(self, training_data, beta1,
beta2):
239         self.vals_for_learnable_params = {param: random.uniform(0,1) for
param in self.learnable_params}
240         self.bias = random.uniform(0,1)
241

```

```

242     class DataLoader:
243         def __init__(self, training_data, batch_size):
244             self.training_data = training_data
245             self.batch_size = batch_size
246             self.class_0_samples = [(item, 0) for item in self.
training_data[0]]    ## Associate label 0 with each sample
247             self.class_1_samples = [(item, 1) for item in self.
training_data[1]]    ## Associate label 1 with each sample
248
249         def __len__(self):
250             return len(self.training_data[0]) + len(self.training_data
[1])
251
252         def _getitem(self):
253             cointoss = random.choice([0,1])
254             ## When a batch is created by getbatch(), we want the
255             ## samples to be chosen randomly from the two lists
256             if cointoss == 0:
257                 return random.choice(self.class_0_samples)
258             else:
259                 return random.choice(self.class_1_samples)
260
261         def getbatch(self):
262             batch_data, batch_labels = [], []
263             ## First list for samples, the second for labels
264             maxval = 0.0
265             ## For approximate batch data normalization
266             for _ in range(self.batch_size):
267                 item = self._getitem()
268                 if np.max(item[0]) > maxval:
269                     maxval = np.max(item[0])
270                 batch_data.append(item[0])
271                 batch_labels.append(item[1])
272             batch_data = [item/maxval for item in batch_data]
273             ## Normalize batch data
274             batch = [batch_data, batch_labels]
275             return batch
276
277     data_loader = DataLoader(training_data, batch_size=self.batch_size
)
278
279     loss_running_record = []
280     i = 0
281     avg_loss_over_iterations = 0.0
282     ## Average the loss over iterations for printing out
283
284     # To keep track of both types of moments:
285     self.m_moments, self.v_moments = {param: 0 for param in self.
learnable_params}, {param: 0 for param in self.learnable_params}
286     self.k = 0 # Timestep initialization

```

```

282         for i in range(self.training_iterations):
283             data = data_loader.getbatch()
284             data_tuples_in_batch = data[0]
285             class_labels_in_batch = data[1]
286             y_preds, deriv_sigmoids = self.forward_prop_one_neuron_model(
data_tuples_in_batch)    ## FORWARD PROP of data
287             loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
for i in range(len(class_labels_in_batch))])    ## Find loss
288             avg_loss_over_iterations += loss / float(len(
class_labels_in_batch))
289             if i%(self.display_loss_how_often) == 0:
290                 avg_loss_over_iterations /= self.display_loss_how_often
291                 loss_running_record.append(avg_loss_over_iterations)
292                 #print("[iter=%d]  loss = %.4f" % (i+1,
avg_loss_over_iterations))    ## Display average loss
293                 avg_loss_over_iterations = 0.0
294                 y_errors_in_batch = list(map(operator.sub,
class_labels_in_batch, y_preds))
295                 self.backprop_and_update_params_one_neuron_model(
data_tuples_in_batch, y_preds, y_errors_in_batch, deriv_sigmoids, beta1
, beta2)  # Backprop now includes betas
296
297         return loss_running_record
298
299     def backprop_and_update_params_one_neuron_model(self,
data_tuples_in_batch, predictions, y_errors_in_batch, deriv_sigmoids,
beta1, beta2):
300         input_vars = self.independent_vars
301         input_vars_to_param_map = self.var_to_var_param[self.output_vars
[0]]    ## These two statements align the
302         param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()}    ## the input vars
303         self.k += 1 # Update for each timestep
304
305         for i,param in enumerate(self.vals_for_learnable_params):
306             ## For each param, sum the partials from every training data
sample in batch
307             partial_of_loss_wrt_param = 0.0
308             for j in range(self.batch_size):
309                 vals_for_input_vars_dict = dict(zip(input_vars, list(
data_tuples_in_batch[j])))
310                 partial_of_loss_wrt_param += - y_errors_in_batch[j] *
vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[j]
311                 partial_of_loss_wrt_param /= float(self.batch_size)
312
313             # Update learnable params using both moments
314             self.m_moments[param] = (self.m_moments[param] * beta1) + (1 -
beta1) * partial_of_loss_wrt_param # m(t+1) Update formula
315             self.v_moments[param] = (self.v_moments[param] * beta2) + ((1
- beta2) * (partial_of_loss_wrt_param ** 2)) # v(t+1) Update formula
316

```

```

317         mhat = self.m_moments[param] / (1 - beta1 ** self.k) # Unbias
the moments from 0
318         vhat = self.v_moments[param] / (1 - beta2 ** self.k)
319
320         step = self.learning_rate * (mhat / math.sqrt(vhat + self.
epsilon))
321         self.vals_for_learnable_params[param] -= step
322
323         y_error_avg = sum(y_errors_in_batch) / float(self.batch_size)
324         deriv_sigmoid_avg = sum(deriv_sigmoids) / float(self.batch_size)
325
326         self.bias += step + self.learning_rate * y_error_avg *
deriv_sigmoid_avg    ## Update the bias
327
328     def run_training_loop_multi_neuron_model(self, training_data, beta1,
beta2):
329
330         class DataLoader:
331
332             def __init__(self, training_data, batch_size):
333                 self.training_data = training_data
334                 self.batch_size = batch_size
335                 self.class_0_samples = [(item, 0) for item in self.
training_data[0]]    ## Associate label 0 with each sample
336                 self.class_1_samples = [(item, 1) for item in self.
training_data[1]]    ## Associate label 1 with each sample
337
338             def __len__(self):
339                 return len(self.training_data[0]) + len(self.training_data
[1])
340
341             def _getitem(self):
342                 cointoss = random.choice([0,1])
343                 ## When a batch is created by getbatch(), we want the
344                 ## samples to be chosen randomly from the two lists
345                 if cointoss == 0:
346                     return random.choice(self.class_0_samples)
347                 else:
348                     return random.choice(self.class_1_samples)
349
350             def getbatch(self):
351                 batch_data, batch_labels = [], []
352                 ## First list for samples, the second for labels
353                 maxval = 0.0
354                 ## For approximate batch data normalization
355                 for _ in range(self.batch_size):
356                     item = self._getitem()
357                     if np.max(item[0]) > maxval:
358                         maxval = np.max(item[0])
359                     batch_data.append(item[0])

```

```

357         batch_labels.append(item[1])
358         batch_data = [item/maxval for item in batch_data]
359     ## Normalize batch data
360     batch = [batch_data, batch_labels]
361     return batch
362
363     self.vals_for_learnable_params = {param: random.uniform(0,1) for
364 param in self.learnable_params}
365     self.bias = {i : [random.uniform(0,1) for j in range( self.
366 layers_config[i] ) ] for i in range(1, self.num_layers)}
367     data_loader = DataLoader(training_data, batch_size=self.batch_size
368 )
369     loss_running_record = []
370     i = 0
371     avg_loss_over_iterations = 0.0
372
373     # To keep track of both types of moments:
374     self.m_moments, self.v_moments = {param: 0 for param in self.
375 learnable_params}, {param: 0 for param in self.learnable_params}
376     self.k = 0 # Timestep initialization
377     self.m_bias, self.v_bias = [], []
378
379     for layer_size in self.layers_config:
380         self.m_bias.append(np.zeros(layer_size)) # individual bias for
381 each neuron in each layer
382         self.v_bias.append(np.zeros(layer_size))
383
384     for i in range(self.training_iterations):
385         data = data_loader.getbatch()
386         data_tuples = data[0]
387         class_labels = data[1]
388         self.forward_prop_multi_neuron_model(data_tuples)
389         predicted_labels_for_batch = self.forw_prop_vals_at_layers[
390 self.num_layers-1]
391         y_preds = [item for sublist in predicted_labels_for_batch
392 for item in sublist]
393         loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
394 range(len(class_labels))])
395         loss_avg = loss / float(len(class_labels))
396         avg_loss_over_iterations += loss_avg
397         if i%(self.display_loss_how_often) == 0:
398             avg_loss_over_iterations /= self.display_loss_how_often
399             loss_running_record.append(avg_loss_over_iterations)
400             #print("[iter=%d] loss = %.4f" % (i+1,
401 avg_loss_over_iterations))
402             avg_loss_over_iterations = 0.0
403             y_errors_in_batch = list(map(operator.sub, class_labels,
404 y_preds))
405             self.backprop_and_update_params_multi_neuron_model(y_preds,
406 y_errors_in_batch, beta1, beta2)
407         return loss_running_record

```

```

396
397     def backprop_and_update_params_multi_neuron_model(self, predictions,
398     y_errors, beta1, beta2):
399         pred_err_backproped_at_layers = [ {i : [None for j in range(
self.layers_config[i] ) ]
for i in
range(self.num_layers)} for _ in range(self.batch_size) ]
400         self.k += 1 # Update time step
401
402         partial_of_loss_wrt_params = {param : 0.0 for param in self.
all_params}
403         bias_changes = {i : [0.0 for j in range( self.layers_config[i] )
] for i in range(1, self.num_layers)}
404         for b in range(self.batch_size):
405             pred_err_backproped_at_layers[b][self.num_layers - 1] = [
y_errors[b] ]
406             for back_layer_index in reversed(range(1,self.num_layers)):
407                 ## For the 3-layer network, the first val for
back_layer_index is 2 for the 3rd layer
408                 input_vals = self.forw_prop_vals_at_layers[
back_layer_index - 1] ## This is a list of 8 two-element lists ---
since we have two nodes in the 2nd layer
409                 deriv_sigmoids = self.gradient_vals_for_layers[
back_layer_index] ## This is a list eight one-element lists, one for
each batch element
410                 vars_in_layer = self.layer_vars[back_layer_index]
## A list like ['xo']
411                 vars_in_next_layer_back = self.layer_vars[
back_layer_index - 1] ## A list like ['xw', 'xz']
412                 vals_for_input_vars_dict = dict(zip(
vars_in_next_layer_back, self.forw_prop_vals_at_layers[back_layer_index
- 1][b]))
413                 ## For the next statement, note that layer_params are
stored in a dict like
414                 ## {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br',
'bs']], 2: [['cp', 'cq']]}
415                 ## "layer_params[idx]" is a list of lists for the link
weights in layer whose output nodes are in layer "idx"
416                 layer_params = self.layer_params[back_layer_index]
transposed_layer_params = list(zip(*layer_params))
417                 ## Creating a transpose of the link matrix, See Eq. 30 on
Slide 77
418                 for k,var1 in enumerate(vars_in_next_layer_back):
419                     for j,var2 in enumerate(vars_in_layer):
420                         pred_err_backproped_at_layers[b][back_layer_index
- 1][k] = sum([self.vals_for_learnable_params[transposed_layer_params[k
][i]]
421
422                         * pred_err_backproped_at_layers[b][back_layer_index][i]
423
424                         for i in range(len(

```

```

vars_in_layer)))]
422         for j,var in enumerate(vars_in_layer):
423             layer_params = self.layer_params[back_layer_index][j]
424             ## ['cp', 'cq'] for the end layer
425             input_vars_to_param_map = self.var_to_var_param[var]
426             ## These two statements align the {'xw': 'cp', 'xz': 'cq'}
427             param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()} ## and the input vars {'cp': 'xw
', 'cq': 'xz'}

428         for i,param in enumerate(layer_params):
429             partial_of_loss_wrt_params[param] +=
pred_err_backproped_at_layers[b][back_layer_index][j] * \
vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[b][
j]
430         for k,var1 in enumerate(vars_in_next_layer_back):
431             for j,var2 in enumerate(vars_in_layer):
432                 if back_layer_index-1 > 0:
433                     bias_changes[back_layer_index-1][k] +=
pred_err_backproped_at_layers[b][back_layer_index - 1][k] *
deriv_sigmoids[b][j]

434         ## Now update the learnable parameters. The loop shown below
435         carries out SGD mandated averaging
436         for param in partial_of_loss_wrt_params:
437             partial_of_loss_wrt_param = partial_of_loss_wrt_params[param]
/ float(self.batch_size)

438         # Update params using ADAM Formulas =====
439
440         self.m_moments[param] = (self.m_moments[param] * beta1) + ((1
- beta1) * partial_of_loss_wrt_param) # m(t+1) Update formula
441         self.v_moments[param] = (self.v_moments[param] * beta2) + ((1
- beta2) * (partial_of_loss_wrt_param ** 2)) # v(t+1) Update formula
442
443         mhat = self.m_moments[param] / (1 - beta1 ** self.k) # Unbias
the moments from 0
444         vhat = self.v_moments[param] / (1 - beta2 ** self.k)
445
446         step = self.learning_rate * (mhat / math.sqrt(vhat + self.
epsilon))
447         self.vals_for_learnable_params[param] += step
448
449         ## Finally we update the biases at all the nodes that aggregate
450         data:
451         for layer_index in range(1,self.num_layers):
452             for k in range(self.layers_config[layer_index]):
453                 gradt = bias_changes[layer_index][k]# / float(self.
batch_size)

```

```

454         self.m_bias[layer_index][k] = beta1 * self.m_bias[
layer_index][k] + (1-beta1) * gradt
455         self.v_bias[layer_index][k] = beta2 * self.v_bias[
layer_index][k] + (1-beta2) * (gradt ** 2)
456
457         mhat = self.m_bias[layer_index][k] / (1 - beta1 ** self.k)
# Unbias the moments from 0
458         vhat = self.v_bias[layer_index][k] / (1 - beta2 ** self.k)
459
460         step = self.learning_rate * (mhat / math.sqrt(vhat + self.
epsilon))
461         self.bias[layer_index][k] -= self.learning_rate * step
462
463 if __name__ == '__main__':
464     # ===== Config Parameters =====
465     lr = 1e-2
466
467     # ===== Initialization of One NN with SGD plus
optimization: =====
468     #Run with only SGD then SGD plus
469     # cgp = SGDplus(
470     #         one_neuron_model = True,
471     #         expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
472     #         output_vars = ['xw'],
473     #         dataset_size = 5000,
474     #         learning_rate = lr,
475     #         training_iterations = 20000,
476     #         batch_size = 8,
477     #         display_loss_how_often = 100,
478     #         debug = True,
479     #     )
480     # cgp.parse_expressions()
481     # training_data = cgp.gen_training_data()
482     # sgd_losses = cgp.run_training_loop_one_neuron_model(training_data,
beta = 0)
483     # sgdplus_losses = cgp.run_training_loop_one_neuron_model(
training_data, beta = 0.9)
484
485     # betas = [0, 0.9, 0.1, 0.5]
486
487     # for beta in betas:
488     #     sgd_losses = cgp.run_training_loop_one_neuron_model(
training_data, beta = beta)
489     #     plt.plot(sgd_losses, label= "SGD beta = "+str(beta))
490
491     # plt.legend()
492     # plt.xlabel('Iterations')
493     # plt.ylabel('Loss Value')
494     # plt.title('SGD Plus for One NN with Various Betas')
495     # plt.show()

```

```

496 # ===== Initialization of Multi NN with SGD plus
optimization: =====
497 cgp = SGDplus(
498     num_layers = 3,
499     layers_config = [4,2,1], # num of
nodes in each layer
500     expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
501                   'xz=bp*xp+bq*xq+br*xr+bs*xs',
502                   'xo=cp*xw+cq*xz'],
503     output_vars = ['xo'],
504     dataset_size = 5000,
505     learning_rate = lr,
506     training_iterations = 20000,
507     batch_size = 8,
508     display_loss_how_often = 100,
509     debug = True,
510 )
511
512 cgp.parse_multi_layer_expressions()
513 training_data = cgp.gen_training_data()
514 # betas = [0, 0.9, 0.1, 0.5]
515
516 # for beta in betas:
517 #     sgd_losses = cgp.run_training_loop_multi_neuron_model(
training_data, beta = beta)
518 #     plt.plot(sgd_losses, label= "SGD beta = "+str(beta))
519 sgd_losses = cgp.run_training_loop_multi_neuron_model(training_data,
beta = 0)
520 sgdplus_losses = cgp.run_training_loop_multi_neuron_model(
training_data, beta = 0.5)
521
522 # plt.legend()
523 # plt.xlabel('Iterations')
524 # plt.ylabel('Loss Value')
525 # plt.title('SGD Plus for Multi NN with Various Betas')
526 # plt.show()
527
528 # ===== Initialization of One NN with ADAM optimization:
=====
529 # cgp = ADAMOptimizer(
530 #     one_neuron_model = True,
531 #     expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
532 #     output_vars = ['xw'],
533 #     dataset_size = 5000,
534 #     learning_rate = lr,
535 #     training_iterations = 20000,
536 #     batch_size = 8,
537 #     display_loss_how_often = 100,
538 #     debug = True,
539 # )
540 # cgp.parse_expressions()

```

```

541     # training_data = cgp.gen_training_data()
542
543     # betas1s, beta2s = [0.8, 0.95, 0.99], [0.85, 0.9, 0.95]
544     # for beta1 in betas1s:
545     #     for beta2 in beta2s:
546     #         start_time = time.time()
547     #         adamlosses = cgp.run_training_loop_one_neuron_model(
training_data, beta1 = beta1, beta2= beta2)
548     #         plt.plot(adamlosses, label= "ADAM B1="+str(beta1)+" B2="+str
(beta2))
549     #         time_taken = time.time() - start_time
550
551     #         final_loss = adamlosses[-1]
552     #         minloss = np.min(adamlosses[10:])
553     #         print(f'B1: {beta1}, B2: {beta2}[Min loss: {minloss}, final
loss: {final_loss} Time taken: {time_taken}s')
554
555     # plt.plot(sgd_losses, label = "SGD")
556
557     # plt.plot(sgdplus_losses, label = "SGD Plus")
558     # plt.legend()
559     # plt.xlabel('Iterations')
560     # plt.ylabel('Loss Value')
561     # plt.title('ADAM for One NN with Various Betas, lr = '+ str(lr))
562     # plt.show()
563
564     # beta1, beta2 = 0.95, 0.85
565     # adamlosses = cgp.run_training_loop_one_neuron_model(training_data,
beta1 = beta1, beta2= beta2)
566
567     # ===== ADAM Optimizer for Multi NN =====
568
569     cgp = ADAMOptimizer(
570         num_layers = 3,
571         layers_config = [4,2,1], # num of
nodes in each layer
572         expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
573                       'xz=bp*xp+bq*xq+br*xr+bs*xs',
574                       'xo=cp*xw+cq*xz'],
575         output_vars = ['xo'],
576         dataset_size = 5000,
577         learning_rate = lr,
578         training_iterations = 20000,
579         batch_size = 8,
580         display_loss_how_often = 100,
581         debug = True,
582     )
583
584     cgp.parse_multi_layer_expressions()
585     training_data = cgp.gen_training_data()
586

```

```

587     # betas1s, beta2s = [0.8, 0.95, 0.99], [0.85, 0.9, 0.95]
588
589     # for beta1 in betas1s:
590     #     for beta2 in beta2s:
591     #         start_time = time.time()
592     #         adamlosses = cgp.run_training_loop_multi_neuron_model(
training_data, beta1 = beta1, beta2= beta2)
593     #         plt.plot(adamlosses, label= "ADAM B1="+str(beta1)+" B2="+str
(beta2))
594     #         time_taken = time.time() - start_time
595
596     #         final_loss = adamlosses[-1]
597     #         minloss = np.min(adamlosses[10:])
598
599     #         print(f'B1: {beta1}, B2: {beta2}[Min loss: {minloss}, final
loss: {final_loss} Time taken: {time_taken}s')
600
601     # plt.plot(sgd_losses, label = "SGD Plus")
602     # plt.legend()
603     # plt.xlabel('Iterations')
604     # plt.ylabel('Loss Value')
605     # plt.title('ADAM for Multi NN with Various Betas, lr = '+ str(lr))
606     # plt.show()
607
608     beta1, beta2 = 0.99, 0.9
609     adamlosses = cgp.run_training_loop_multi_neuron_model(training_data,
beta1 = beta1, beta2= beta2)
610
611     plt.plot(sgd_losses, label = "SGD")
612     plt.plot(sgdplus_losses, label = 'SGD Plus')
613     plt.plot(adamlosses, label = 'ADAM')
614     plt.xlabel('Iterations')
615     plt.ylabel('Loss Value')
616     plt.title('Comparison of Different Optimizers for Multi-NN')
617     plt.legend()
618     plt.show()

```

Code Listing 1: hw3.py

### *extra<sub>c</sub>red.py*

```

1 from ComputationalGraphPrimer import ComputationalGraphPrimer
2 import numpy as np
3 import random, operator, math, time
4 import matplotlib.pyplot as plt
5 from tqdm import tqdm
6 from collections import defaultdict
7
8 random.seed(0)
9 np.random.seed(0)

```

```

10
11 class CustomDataloader(ComputationalGraphPrimer):
12     def __init__(self, *args, **kwargs):
13         super().__init__(*args, **kwargs)
14
15     def truncate(self, data):
16         mu, std = np.mean(data), np.std(data)
17         lower_bound, upper_bound = mu - 5 * std, mu + 5 * std
18
19         truncated_data = np.clip(data, lower_bound, upper_bound)
20         min_val, max_val = truncated_data.min(), truncated_data.max()
21         scaled_data = 2 * (truncated_data - min_val) / (max_val - min_val)
22         - 1
23         return scaled_data
24
25     def run_training_loop_one_neuron_model(self, training_data, normalize
26     = True, truncate = False):
27         # Initialize learnable params for network -----
28         self.vals_for_learnable_params = {param: random.uniform(0,1) for
29         param in self.learnable_params}
30         self.bias = random.uniform(0,1)
31
32         # Truncate data values
33
34         if truncate:
35             class0, class1 = self.truncate(training_data[0]), self.
36             truncate(training_data[1])
37             training_data[0], training_data[1] = class0, class1
38         # Dataloader business -----
39         class DataLoader:
40             def __init__(self, training_data, batch_size):
41                 self.training_data = training_data
42                 self.batch_size = batch_size
43                 self.class_0_samples = [(item, 0) for item in self.
44                 training_data[0]] ## Associate label 0 with each sample
45                 self.class_1_samples = [(item, 1) for item in self.
46                 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
50                 [1])
51
52             def _getitem(self):
53                 cointoss = random.choice([0,1])
54                 ## When a batch is created by getbatch(), we want the
55                 ## samples to be chosen randomly from the two lists
56                 if cointoss == 0:
57                     return random.choice(self.class_0_samples)
58                 else:
59                     return random.choice(self.class_1_samples)

```

```

52         def getbatch(self):
53             batch_data, batch_labels = [], []
54             ## First list for samples, the second for labels
55             maxval = 0.0
56             ## For approximate batch data normalization
57             for _ in range(self.batch_size):
58                 item = self._getitem()
59                 if np.max(item[0]) > maxval:
60                     maxval = np.max(item[0])
61                 batch_data.append(item[0])
62                 batch_labels.append(item[1])
63             if normalize: batch_data = [item/maxval for item in
batch_data]
64             ## Normalize batch data
65             batch = [batch_data, batch_labels]
66             return batch
67
68     data_loader = DataLoader(training_data, batch_size=self.batch_size
69 )
70
71     loss_running_record = []
72     i = 0
73     avg_loss_over_iterations = 0.0
74     ## Average the loss over iterations for printing out
75
76     # Actually running training -----
77     for i in range(self.training_iterations):
78         data = data_loader.getbatch()
79         data_tuples_in_batch = data[0]
80         class_labels_in_batch = data[1]
81
82         y_preds, deriv_sigmoids = self.forward_prop_one_neuron_model(
data_tuples_in_batch) ## FORWARD PROP of data
83         loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
for i in range(len(class_labels_in_batch))]) ## Find loss
84         avg_loss_over_iterations += loss / float(len(
class_labels_in_batch))
85
86         if i%(self.display_loss_how_often) == 0:
87             avg_loss_over_iterations /= self.display_loss_how_often
88             loss_running_record.append(avg_loss_over_iterations)
89             # print("[iter=%d] loss = %.4f" % (i+1,
avg_loss_over_iterations)) ## Display average loss
90             avg_loss_over_iterations = 0.0
91             ## Re-initialize avg loss
92
93         y_errors_in_batch = list(map(operator.sub,
class_labels_in_batch, y_preds))
94         self.backprop_and_update_params_one_neuron_model(
data_tuples_in_batch, y_preds, y_errors_in_batch, deriv_sigmoids) ##
BACKPROP loss
95     return loss_running_record # Return loss for own plotting function

```

```

90
91     def run_training_loop_multi_neuron_model(self, training_data,
normalize = True, truncate = False):
92
93         class DataLoader:
94             def __init__(self, training_data, batch_size):
95                 self.training_data = training_data
96                 self.batch_size = batch_size
97                 self.class_0_samples = [(item, 0) for item in self.
training_data[0]]
98                 self.class_1_samples = [(item, 1) for item in self.
training_data[1]]
99
100             def __len__(self):
101                 return len(self.training_data[0]) + len(self.training_data
[1])
102
103             def _getitem(self):
104                 cointoss = random.choice([0,1])
105                 ## When a batch is created by getbatch(), we want the
106                 ## samples to be chosen randomly from the two lists
107                 if cointoss == 0:
108                     return random.choice(self.class_0_samples)
109                 else:
110                     return random.choice(self.class_1_samples)
111
112             def getbatch(self):
113                 batch_data, batch_labels = [], []
114                 ## First list for samples, the second for labels
115                 maxval = 0.0
116                 ## For approximate batch data normalization
117                 for _ in range(self.batch_size):
118                     item = self._getitem()
119                     if np.max(item[0]) > maxval:
120                         maxval = np.max(item[0])
121                     batch_data.append(item[0])
122                     batch_labels.append(item[1])
123                 if normalize: batch_data = [item/maxval for item in
batch_data]
124                 ## Normalize batch data
125                 batch = [batch_data, batch_labels]
126                 return batch
127
128             # Truncate data values
129             if truncate:
130                 class0, class1 = self.truncate(training_data[0]), self.
truncate(training_data[1])
131                 training_data[0], training_data[1] = class0, class1
132             # Initialize Learnable params
133             self.vals_for_learnable_params = {param: random.uniform(0,1) for
param in self.learnable_params}

```

```

130         self.bias = {i : [random.uniform(0,1) for j in range( self.
layers_config[i] ) ] for i in range(1, self.num_layers)}
131         data_loader = DataLoader(training_data, batch_size=self.batch_size
)
132         loss_running_record = []
133         i = 0
134         avg_loss_over_iterations = 0.0
135
136         for i in range(self.training_iterations):
137             data = data_loader.getbatch()
138             data_tuples = data[0]
139             class_labels = data[1]
140             self.forward_prop_multi_neuron_model(data_tuples)
141                 ## FORW PROP works by side-effect
142             predicted_labels_for_batch = self.forw_prop_vals_at_layers[
self.num_layers-1]
143                 ## Predictions from FORW PROP
144             y_preds = [item for sublist in predicted_labels_for_batch
for item in sublist]
145                 ## Get numeric vals for predictions
146             loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
range(len(class_labels))])
147                 ## Calculate loss for batch
148             loss_avg = loss / float(len(class_labels))
149                 ## Average the loss over batch
150             avg_loss_over_iterations += loss_avg
151                 ## Add to Average loss over iterations
152             if i%(self.display_loss_how_often) == 0:
153                 avg_loss_over_iterations /= self.display_loss_how_often
154                 loss_running_record.append(avg_loss_over_iterations)
155                 # print("[iter=%d] loss = %.4f" % (i+1,
avg_loss_over_iterations))
156                 avg_loss_over_iterations = 0.0
157             y_errors_in_batch = list(map(operator.sub, class_labels,
y_preds))
158             self.backprop_and_update_params_multi_neuron_model(y_preds,
y_errors_in_batch)
159             return loss_running_record
160
161 if __name__ == '__main__':
162     # ===== Compare One NN with and without normalizaiton:
=====
163     #Run with only SGD then SGD plus
164     # cgp = CustomDataloader(
165     #     one_neuron_model = True,
166     #     expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
167     #     output_vars = ['xw'],
168     #     dataset_size = 5000,
169     #     learning_rate = 1e-3,
170     #     training_iterations = 40000,
171     #     batch_size = 8,
172     #     display_loss_how_often = 100,
173     #     debug = True,
174     # )

```

```

169 # cgp.parse_expressions()
170 # training_data = cgp.gen_training_data()
171 # custom_losses_onenn = cgp.run_training_loop_one_neuron_model(
training_data, normalize= False)
172 # regular_losses_onenn = cgp.run_training_loop_one_neuron_model(
training_data, normalize= True )
173
174 # plt.plot(custom_losses_onenn, label = "No Normalization")
175 # plt.plot(regular_losses_onenn, label = 'With Normalization')
176 # plt.legend()
177 # plt.xlabel('Iterations')
178 # plt.ylabel('Loss Value')
179 # plt.title('One NN Comparison of Normalization')
180 # plt.show()
181
182 # ===== Compare One NN with and without normalizaiton:
=====
183 # #Run with only SGD then SGD plus
184 # cgp = CustomDataloader(
185 #     num_layers = 3,
186 #     layers_config = [4,2,1], # num of
nodes in each layer
187 #     expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
188 #                   'xz=bp*xp+bq*xq+br*xr+bs*xs',
189 #                   'xo=cp*xw+cq*xz'],
190 #     output_vars = ['xo'],
191 #     dataset_size = 5000,
192 #     learning_rate = 9e-2,
193 #     training_iterations = 20000,
194 #     batch_size = 8,
195 #     display_loss_how_often = 100,
196 #     debug = True,
197 # )
198
199 # cgp.parse_multi_layer_expressions()
200 # training_data = cgp.gen_training_data()
201 # custom_losses_onenn = cgp.run_training_loop_multi_neuron_model(
training_data, normalize= False)
202 # regular_losses_onenn = cgp.run_training_loop_multi_neuron_model(
training_data, normalize= True )
203
204 # plt.plot(custom_losses_onenn, label = "No Normalization")
205 # plt.plot(regular_losses_onenn, label = 'With Normalization')
206 # plt.legend()
207 # plt.xlabel('Iterations')
208 # plt.ylabel('Loss Value')
209 # plt.title('Multi NN Comparison of Normalization')
210 # plt.show()
211
212 # # ===== Compare One NN with and without truncation:
=====

```

```

213 # #Run with only SGD then SGD plus
214 # cgp = CustomDataloader(
215 #     one_neuron_model = True,
216 #     expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
217 #     output_vars = ['xw'],
218 #     dataset_size = 5000,
219 #     learning_rate = 1e-3,
220 #     training_iterations = 40000,
221 #     batch_size = 8,
222 #     display_loss_how_often = 100,
223 #     debug = True,
224 # )
225 # cgp.parse_expressions()
226 # training_data = cgp.gen_training_data()
227 # print(training_data)
228 # custom_losses_onenn = cgp.run_training_loop_one_neuron_model(
training_data, truncate= True)
229 # regular_losses_onenn = cgp.run_training_loop_one_neuron_model(
training_data, truncate= False )
230
231 # plt.plot(custom_losses_onenn, label = "With Truncation")
232 # plt.plot(regular_losses_onenn, label = 'Without Truncation')
233 # plt.legend()
234 # plt.xlabel('Iterations')
235 # plt.ylabel('Loss Value')
236 # plt.title('One NN Comparison of Truncation')
237 # plt.show()
238
239
240 # ===== Compare One NN with and without truncation:
=====
241 #Run with only SGD then SGD plus
242 cgp = CustomDataloader(
243     num_layers = 3,
244     layers_config = [4,2,1], # num of
nodes in each layer
245     expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
246                   'xz=bp*xp+bq*xq+br*xr+bs*xs',
247                   'xo=cp*xw+cq*xz'],
248     output_vars = ['xo'],
249     dataset_size = 5000,
250     learning_rate = 9e-2,
251     training_iterations = 20000,
252     batch_size = 8,
253     display_loss_how_often = 100,
254     debug = True,
255 )
256
257 cgp.parse_multi_layer_expressions()
258 training_data = cgp.gen_training_data()
259

```

```

260     custom_losses_onenn = cgp.run_training_loop_multi_neuron_model(
training_data, truncate= True)
261     regular_losses_onenn = cgp.run_training_loop_multi_neuron_model(
training_data, truncate= False )
262
263     plt.plot(custom_losses_onenn, label = "With Truncation")
264     plt.plot(regular_losses_onenn, label = 'Without Truncation')
265     plt.legend()
266     plt.xlabel('Iterations')
267     plt.ylabel('Loss Value')
268     plt.title('Multi NN Comparison of Truncation')
269     plt.show()

```

Code Listing 2: *extra<sub>c</sub>red.py*