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.

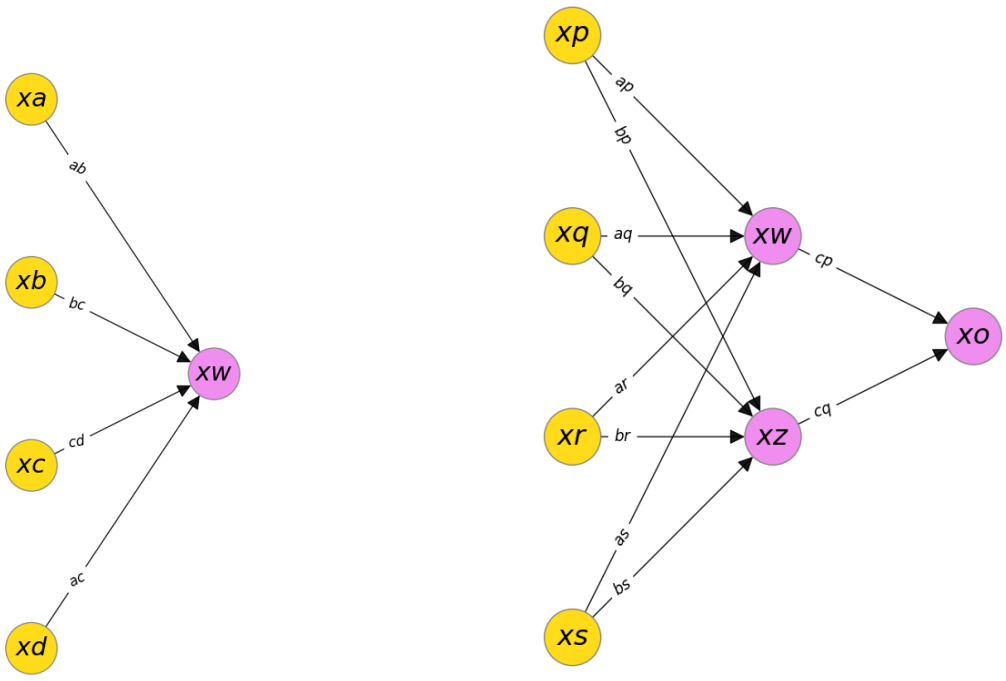

Figure 1: Network Structure of One-NN and Multi-NN

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.

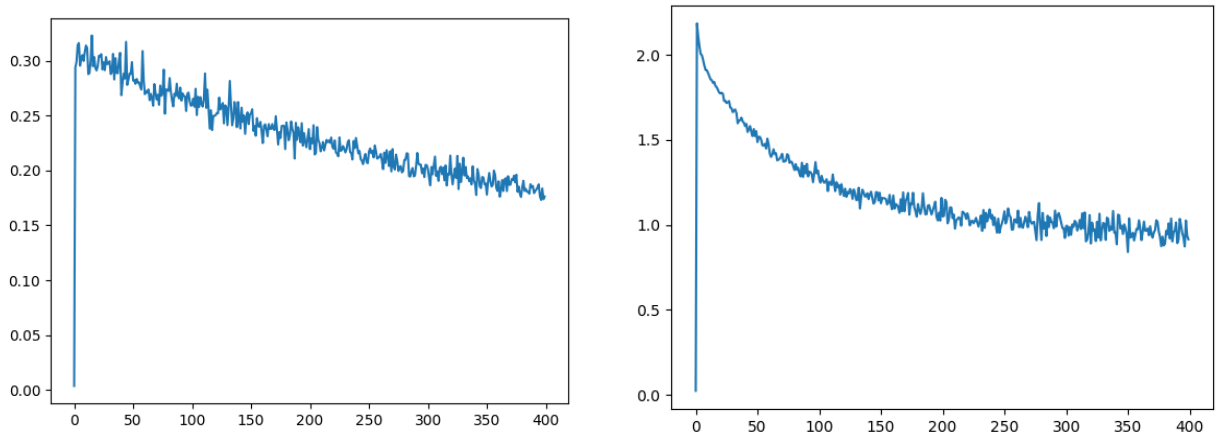

Figure 2: Handcrafted vs Torch Implementation of One Neuron

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.

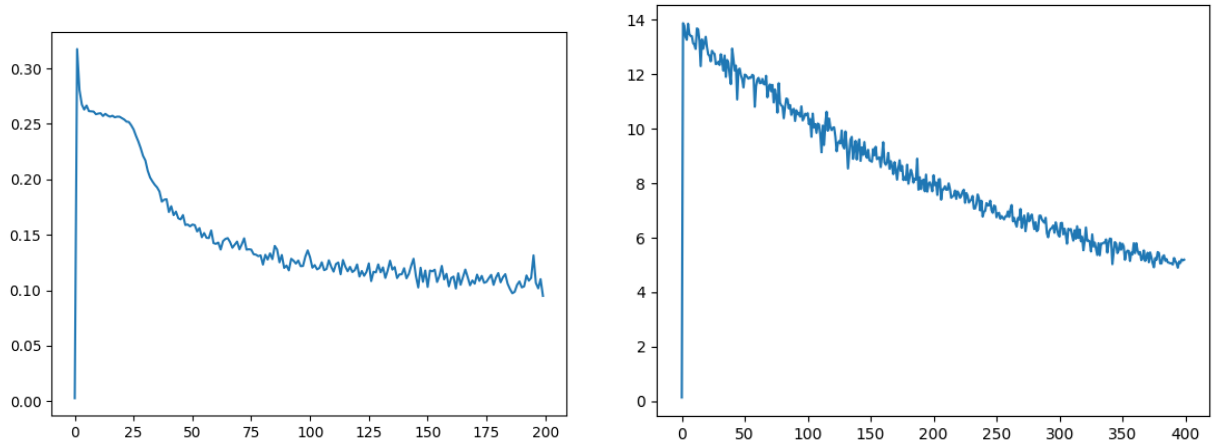

Figure 3: Handcrafted vs Torch Implementation of Multi-Neuron

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,
- η is the learning rate,
- β is the momentum factor ($0 \leq \beta \leq 1$).

A higher β 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 β_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 β_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:

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

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.

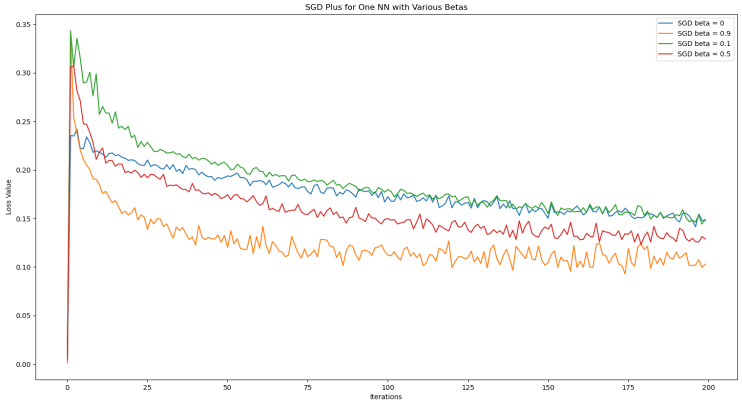

Figure 4: SGD+ - Loss vs Beta values for One-NN

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 fasted, while the higher beta value such as 0.9 converges more smoothly.

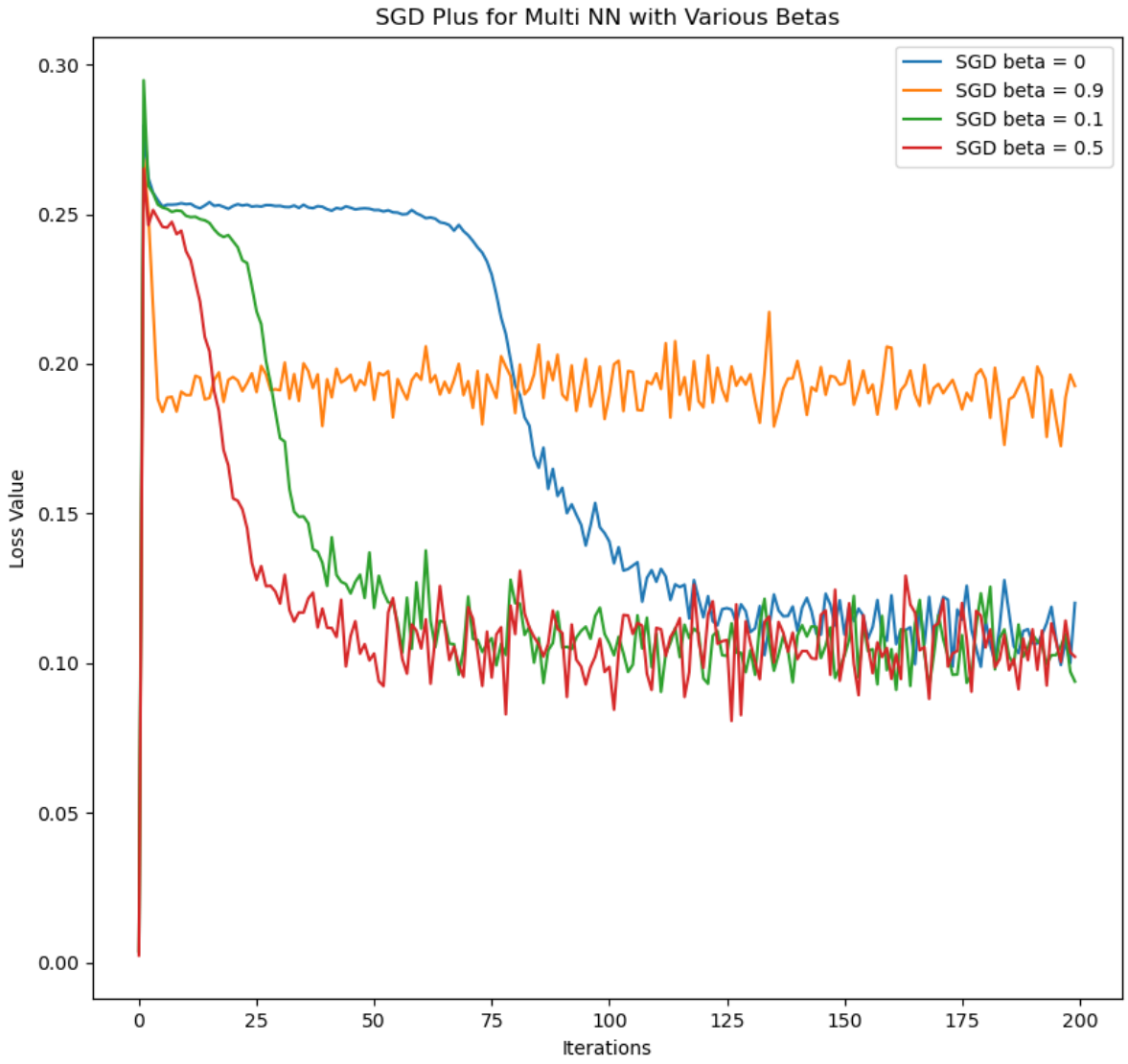

Figure 5: SGD+ - Loss vs Beta values for Multi-NN

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:

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

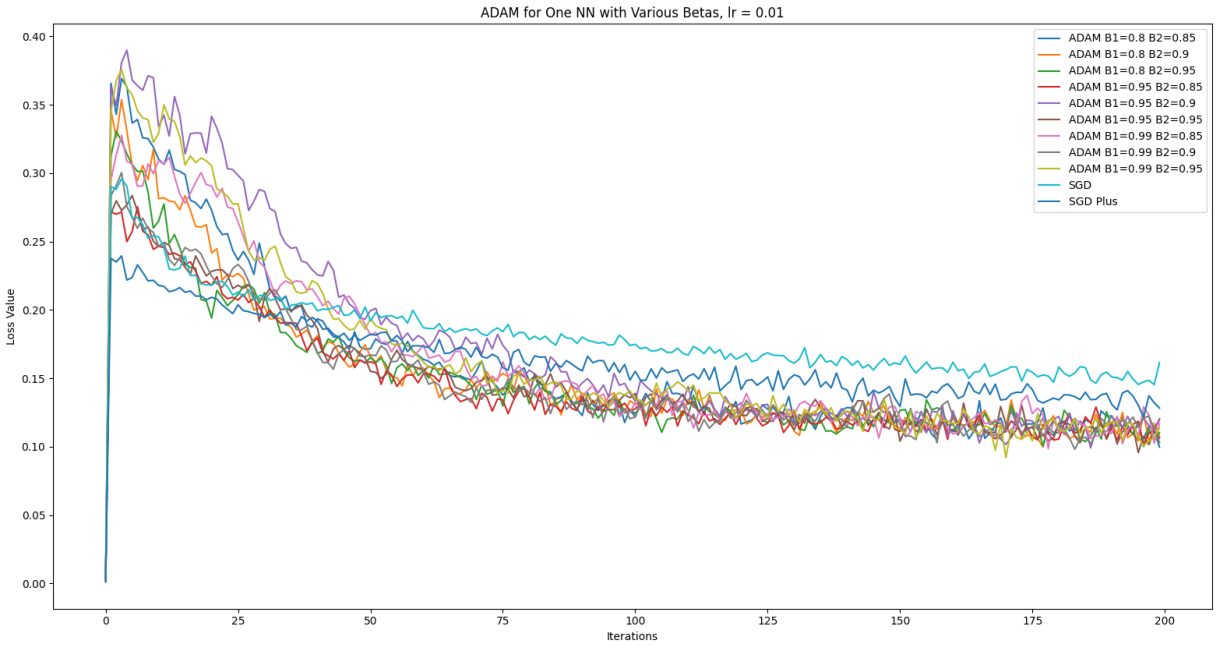

Figure 6: ADAM - Loss vs Beta values for One-NN, lr = 1e-2

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 1-e2 shows the best convergence to the lowest learning rate.

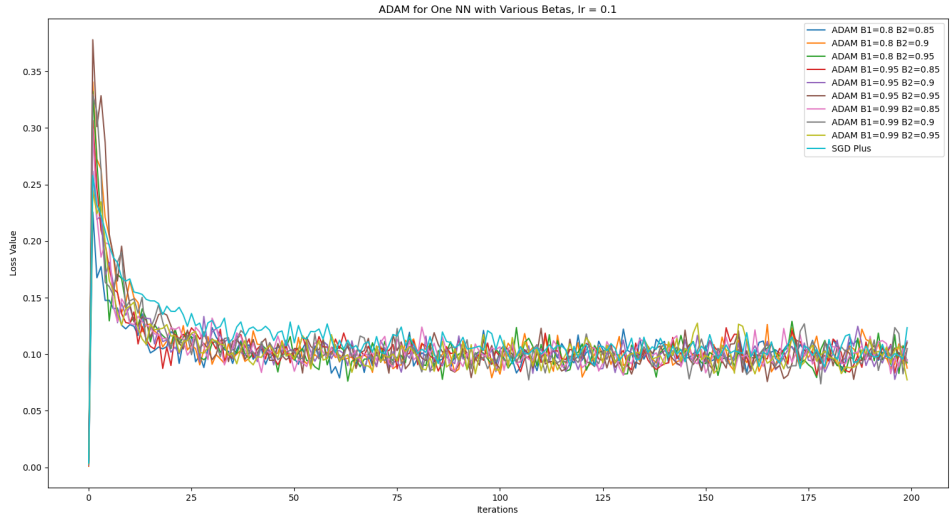

Figure 7: ADAM - Loss vs Beta values for One-NN, lr = 1e-1

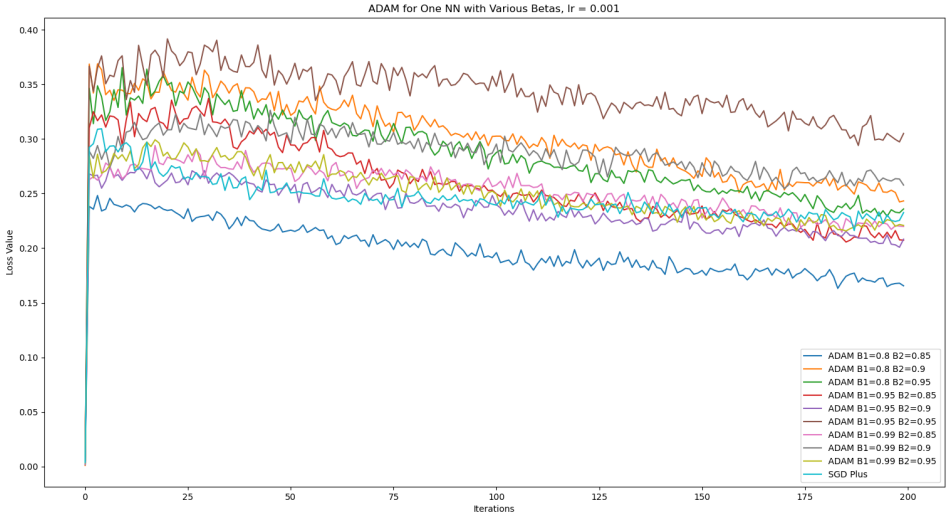

Figure 8: ADAM - Loss vs Beta values for One-NN, lr = 1e-3

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.

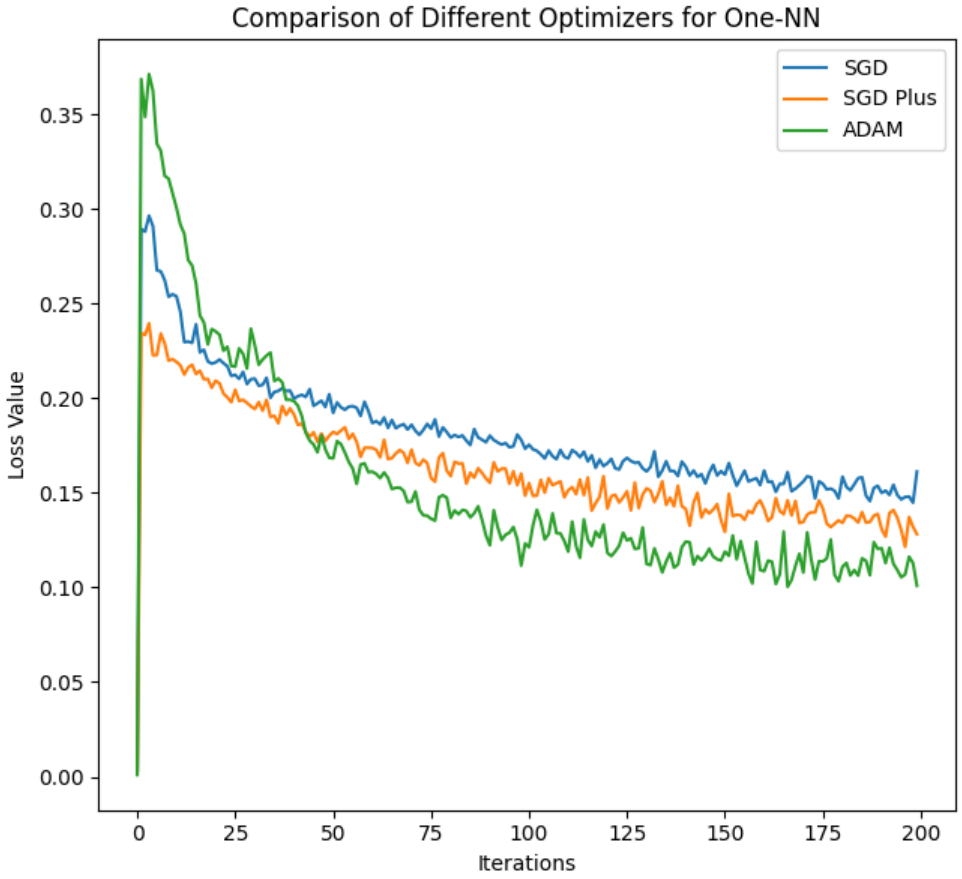

Figure 9: Final One Neuron Comparison

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.

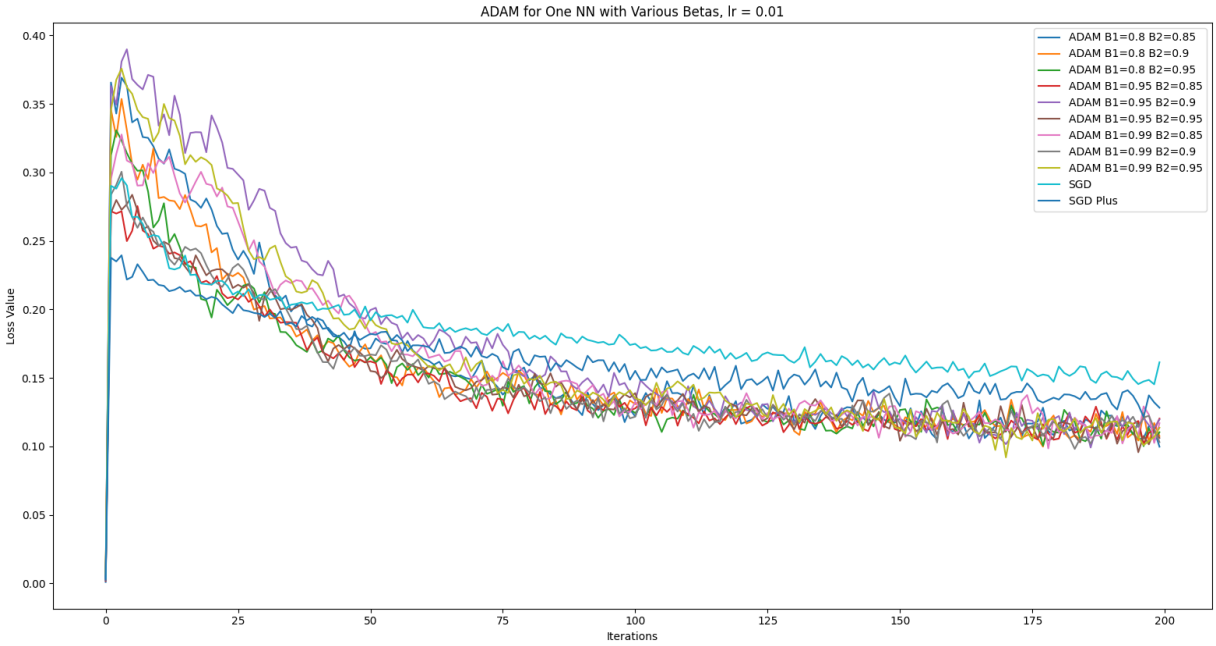

Figure 10: ADAM - Loss vs Beta values for Multi-NN, lr = 1e-2

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.

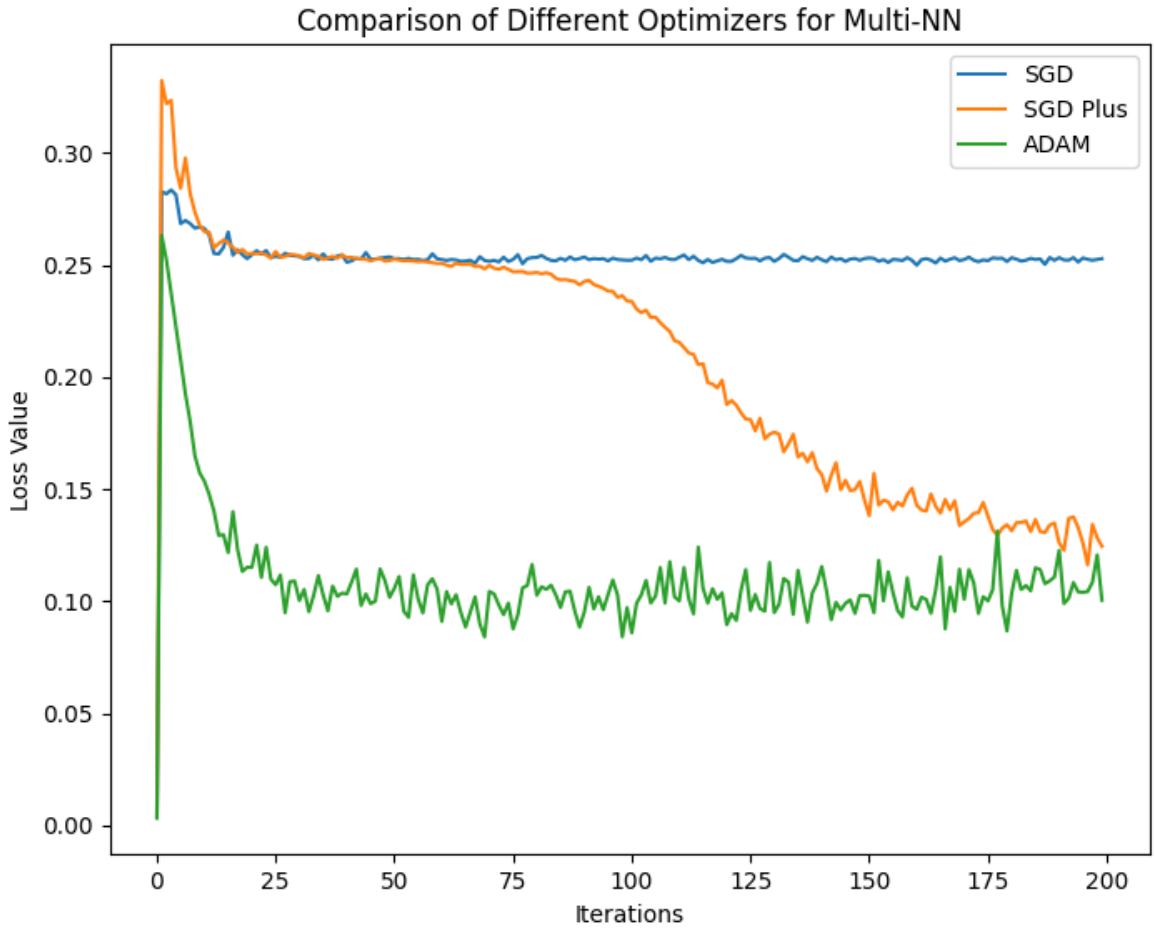

Figure 11: Final Multi Neuron Comparison

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.

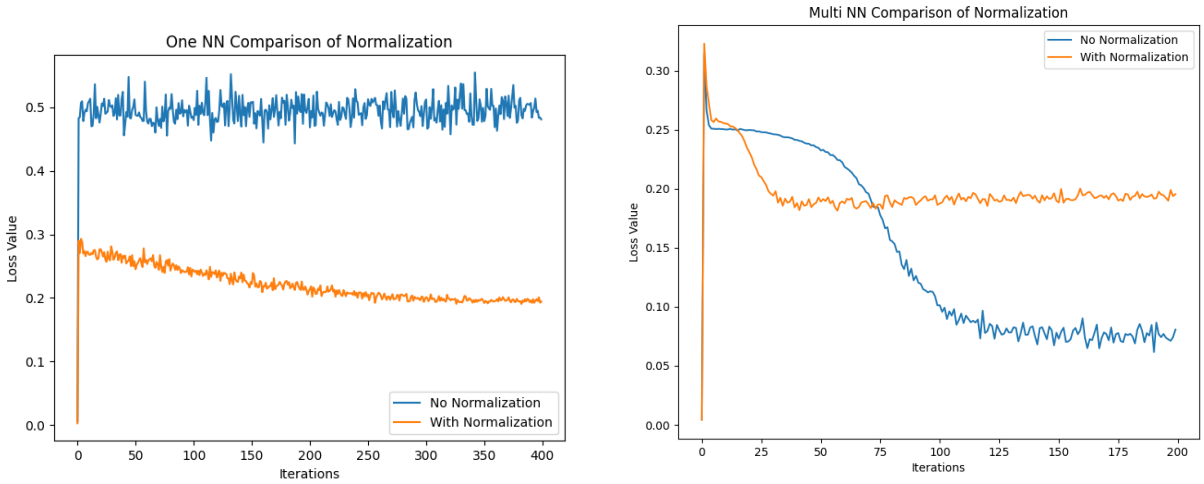

Figure 12: Comparison of Effects of Data Normalization

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.

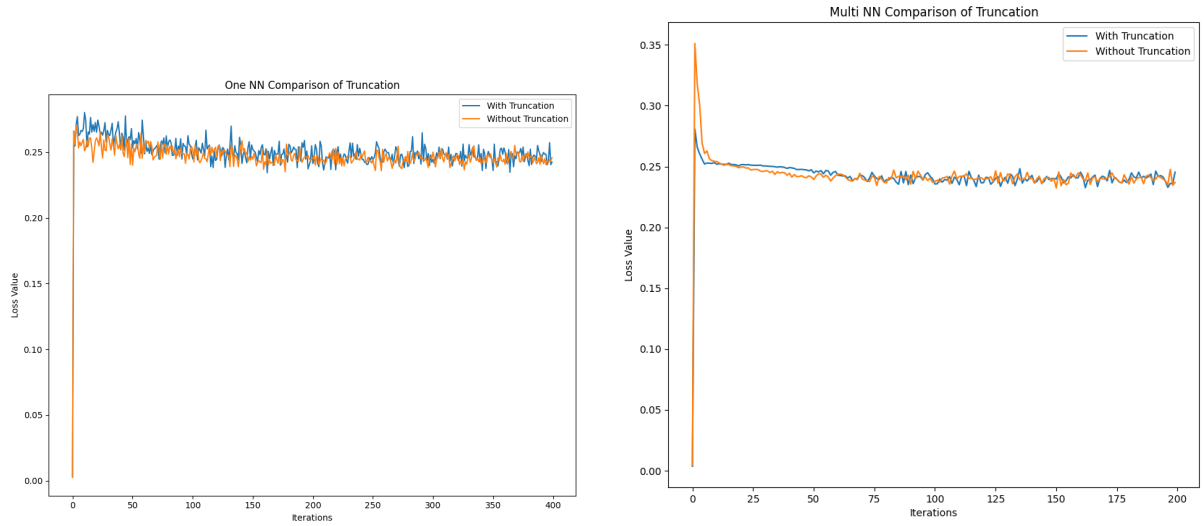

Figure 13: Comparison of Effects of Truncation

6 Source Code

hw3.py

```
from ComputationalGraphPrimer import ComputationalGraphPrimer
import numpy as np
import random, operator, math, time
import matplotlib.pyplot as plt
from tqdm import tqdm

random.seed(0)
np.random.seed(0)


class SGDplus(ComputationalGraphPrimer):
    def __init__(self,  *args, **kwargs): # Now includes beta factor for momentum, and run plus option so we can get back vanilla sgd
        super().__init__(*args, **kwargs)

    def run_training_loop_one_neuron_model(self, training_data, beta):
        # Initialize learnable params for network -----------------
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
        self.bias = random.uniform(0,1)

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

        # Dataloader business -----------------
        class DataLoader:
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]   ## Associate label 1 with each sample

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

            def _getitem(self):
                cointoss = random.choice([0,1])
    ## When a batch is created by getbatch(), we want the

    ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
```

```
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

        def getbatch(self):
            batch_data,batch_labels = [],[]
 ## First list for samples, the second for labels
            maxval = 0.0
 ## For approximate batch data normalization
            for _ in range(self.batch_size):
                item = self._getitem()
                if np.max(item[0]) > maxval:
                    maxval = np.max(item[0])
                batch_data.append(item[0])
                batch_labels.append(item[1])
            batch_data = [item/maxval for item in batch_data]
 ## Normalize batch data
            batch = [batch_data, batch_labels]
            return batch

    data_loader = DataLoader(training_data, batch_size=self.batch_size
)
    loss_running_record = []
    i = 0
    avg_loss_over_iterations = 0.0
##  Average the loss over iterations for printing out

    # Actually running training -----------------
    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples_in_batch = data[0]
        class_labels_in_batch = data[1]

        y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(
data_tuples_in_batch)     ##  FORWARD PROP of data
        loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
for i in range(len(class_labels_in_batch))])  ##  Find loss
        avg_loss_over_iterations += loss / float(len(
class_labels_in_batch))

        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_iterations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_iterations)
           # print("[iter=%d]  loss = %.4f" %  (i+1,
avg_loss_over_iterations))                 ## Display average loss
            avg_loss_over_iterations = 0.0
                        ## Re-initialize avg loss

        y_errors_in_batch = list(map(operator.sub,
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
    [1])
116
117             def _getitem(self):
118                 cointoss = random.choice([0,1])
     ## When a batch is created by getbatch(), we want the
119
     ##   samples to be chosen randomly from the two lists
120                 if cointoss == 0:
121                     return random.choice(self.class_0_samples)
122                 else:
123                     return random.choice(self.class_1_samples)
124
125             def getbatch(self):
126                 batch_data,batch_labels = [],[]
     ## First list for samples, the second for labels
127                 maxval = 0.0
     ## For approximate batch data normalization
128                 for _ in range(self.batch_size):
129                     item = self._getitem()
130                     if np.max(item[0]) > maxval:
131                         maxval = np.max(item[0])
132                     batch_data.append(item[0])
133                     batch_labels.append(item[1])
134                 batch_data = [item/maxval for item in batch_data]
     ## Normalize batch data
135                 batch = [batch_data, batch_labels]
136                 return batch
137
138         # Initialize Learnable params
139         self.vals_for_learnable_params = {param: random.uniform(0,1) for
    param in self.learnable_params}
140         self.bias =   {i : [random.uniform(0,1) for j in range( self.
    layers_config[i] ) ]  for i in range(1, self.num_layers)}
141         data_loader = DataLoader(training_data, batch_size=self.batch_size
    )
142         loss_running_record = []
143         i = 0
144         avg_loss_over_iterations = 0.0
145
146         # Create new variables to keep track of velocities:
    -----------------
147         self.weight_velocities = {param: 0 for param in self.
    learnable_params}
148         self.bias_velocity = []
149         for layer_size in self.layers_config:
150             self.bias_velocity.append(np.zeros(layer_size)) # individual
    bias for each neuron in each layer
151
152         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)
                                ## FORW PROP works by side-effect
157             predicted_labels_for_batch = self.forw_prop_vals_at_layers[
    self.num_layers-1]           ## Predictions from FORW PROP
158             y_preds =  [item for sublist in  predicted_labels_for_batch
    for item in sublist]       ## Get numeric vals for predictions
159             loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
    range(len(class_labels))])  ## Calculate loss for batch
160             loss_avg = loss / float(len(class_labels))
                                ## Average the loss over batch
161             avg_loss_over_iterations += loss_avg
                                ## Add to Average loss over iterations
162             if i%(self.display_loss_how_often) == 0:
163                 avg_loss_over_iterations /= self.display_loss_how_often
164                 loss_running_record.append(avg_loss_over_iterations)
165                # print("[iter=%d]  loss = %.4f" %  (i+1,
    avg_loss_over_iterations))
166                 avg_loss_over_iterations = 0.0
167             y_errors_in_batch = list(map(operator.sub, class_labels,
    y_preds))
168             self.backprop_and_update_params_multi_neuron_model(y_preds,
    y_errors_in_batch, beta)
169         return loss_running_record
170
171     def backprop_and_update_params_multi_neuron_model(self, predictions,
    y_errors, beta):
172         ## Eq. (24) on Slide 73 of my Week 3 lecture says we need to store
     backproped errors in each layer leading up to the last:
173         pred_err_backproped_at_layers =   [ {i : [None for j in range(
    self.layers_config[i] ) ]
174                                                                 for i in
     range(self.num_layers)} for _ in range(self.batch_size) ]
175         ## This will store "\delta L / \delta w" you see at the LHS of the
     equations on Slide 73:
176         partial_of_loss_wrt_params = {param : 0.0 for param in self.
    all_params}
177         ## For estimating the changes to the bias to be made on the basis
    of the derivatives of the Sigmoids:
178         bias_changes =   {i : [0.0 for j in range( self.layers_config[i] )
     ]  for i in range(1, self.num_layers)}
179
180         for b in range(self.batch_size):
181             pred_err_backproped_at_layers[b][self.num_layers - 1] = [
    y_errors[b] ]
182             for back_layer_index in reversed(range(1,self.num_layers)):
                 ## For the 3-layer network, the first val for
    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
                    * pred_err_backproped_at_layers[b][back_layer_index][i]
197
                                                for i in range(len(
    vars_in_layer))])
198                 for j,var in enumerate(vars_in_layer):
199                     layer_params = self.layer_params[back_layer_index][j]
              ##  ['cp', 'cq']   for the end layer
200                     input_vars_to_param_map = self.var_to_var_param[var]
              ## These two statements align the    {'xw': 'cp', 'xz': 'cq'}
201                     param_to_vars_map = {param : var for var, param in
    input_vars_to_param_map.items()}   ##   and the input vars   {'cp': 'xw
    ', 'cq': 'xz'}
202
203                     ##  Update the partials of Loss wrt to the learnable
    parameters between the current layer
204                     ##  and the previous layer. You are accumulating these
     partials over the different training
205                     ##  data samples in the batch being processed.  For
    each training data sample, the formula
206                     ##  being used is shown in Eq. (29) on Slide 77 of my
    Week 3 slides:
207                     for i,param in enumerate(layer_params):
```

```
                        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]
                ##  We will now estimate the change in the bias that needs
     to be made at each node in the previous layer
                ##  from the derivatives the sigmoid at the nodes in the
    current layer and the prediction error as
                ##  backproped to the previous layer nodes:
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        if back_layer_index-1 > 0:
                            bias_changes[back_layer_index-1][k] +=
    pred_err_backproped_at_layers[b][back_layer_index - 1][k] *
    deriv_sigmoids[b][j]

        ## 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]


class ADAMOptimizer(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.epsilon = 1e-8 # for division by zero

    def run_training_loop_one_neuron_model(self, training_data, beta1,
    beta2):
        self.vals_for_learnable_params = {param: random.uniform(0,1) for
    param in self.learnable_params}
        self.bias = random.uniform(0,1)

```

```
        class DataLoader:
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.
    training_data[0]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.
    training_data[1]]   ## Associate label 1 with each sample

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

            def _getitem(self):
                cointoss = random.choice([0,1])
     ## When a batch is created by getbatch(), we want the

     ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]
     ## First list for samples, the second for labels
                maxval = 0.0
     ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]
     ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

        data_loader = DataLoader(training_data, batch_size=self.batch_size
    )
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0
    ##  Average the loss over iterations for printing out

        # 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

```

```
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 
```

```
            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

    def run_training_loop_multi_neuron_model(self, training_data, beta1,
    beta2):

        class DataLoader:

            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.
    training_data[0]]    ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.
    training_data[1]]    ## Associate label 1 with each sample

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

            def _getitem(self):
                cointoss = random.choice([0,1])
    ## When a batch is created by getbatch(), we want the

    ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]
    ## First list for samples, the second for labels
                maxval = 0.0
    ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
```

```
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]
 ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

    self.vals_for_learnable_params = {param: random.uniform(0,1) for
param in self.learnable_params}
    self.bias =   {i : [random.uniform(0,1) for j in range( self.
layers_config[i] ) ]  for i in range(1, self.num_layers)}
    data_loader = DataLoader(training_data, batch_size=self.batch_size
)
    loss_running_record = []
    i = 0
    avg_loss_over_iterations = 0.0

    # 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))

    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples = data[0]
        class_labels = data[1]
        self.forward_prop_multi_neuron_model(data_tuples)
        predicted_labels_for_batch = self.forw_prop_vals_at_layers[
self.num_layers-1]
        y_preds =  [item for sublist in  predicted_labels_for_batch
for item in sublist]
        loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
range(len(class_labels))])
        loss_avg = loss / float(len(class_labels))
        avg_loss_over_iterations += loss_avg
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_iterations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_iterations)
            #print("[iter=%d]  loss = %.4f" %  (i+1,
avg_loss_over_iterations))
            avg_loss_over_iterations = 0.0
        y_errors_in_batch = list(map(operator.sub, class_labels,
y_preds))
        self.backprop_and_update_params_multi_neuron_model(y_preds,
y_errors_in_batch, beta1, beta2)
    return loss_running_record
```

```
396
397     def backprop_and_update_params_multi_neuron_model(self, predictions,
    y_errors, beta1, beta2):
398         pred_err_backproped_at_layers =   [ {i : [None for j in range(
    self.layers_config[i] ) ]
399                                                                 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)):
                 ## For the 3-layer network, the first val for
    back_layer_index is 2 for the 3rd layer
407                 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
408                 deriv_sigmoids =  self.gradient_vals_for_layers[
    back_layer_index]   ## This is a list eight one-element lists, one for
    each batch element
409                 vars_in_layer  =  self.layer_vars[back_layer_index]
                  ## A list like ['xo']
410                 vars_in_next_layer_back  =  self.layer_vars[
    back_layer_index - 1]   ## A list like ['xw', 'xz']
411                 vals_for_input_vars_dict =  dict(zip(
    vars_in_next_layer_back, self.forw_prop_vals_at_layers[back_layer_index
     - 1][b]))
412                 ## For the next statement, note that layer_params are
    stored in a dict like
413                 ##       {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br',
     'bs']], 2: [['cp', 'cq']]}
414                 ## "layer_params[idx]" is a list of lists for the link
    weights in layer whose output nodes are in layer "idx"
415                 layer_params = self.layer_params[back_layer_index]
416                 transposed_layer_params = list(zip(*layer_params))
                  ## Creating a transpose of the link matrix, See Eq. 30 on
    Slide 77
417                 for k,var1 in enumerate(vars_in_next_layer_back):
418                     for j,var2 in enumerate(vars_in_layer):
419                         pred_err_backproped_at_layers[b][back_layer_index
    - 1][k] = sum([self.vals_for_learnable_params[transposed_layer_params[k
    ][i]]
420
421                 * pred_err_backproped_at_layers[b][back_layer_index][i]
                                                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]
                ##  ['cp', 'cq']   for the end layer
424                     input_vars_to_param_map = self.var_to_var_param[var]
                ## These two statements align the    {'xw': 'cp', 'xz': 'cq'}
425                     param_to_vars_map = {param : var for var, param in
    input_vars_to_param_map.items()}   ##   and the input vars   {'cp': 'xw
    ', 'cq': 'xz'}
426
427                     for i,param in enumerate(layer_params):
428                         partial_of_loss_wrt_params[param]   +=
    pred_err_backproped_at_layers[b][back_layer_index][j] * \
429
    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
435         ## Now update the learnable parameters.  The loop shown below
    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
439             # Update params using ADAM Formulas ======================
440
441             self.m_moments[param] = (self.m_moments[param] * beta1) + ((1
    - beta1) * partial_of_loss_wrt_param) # m(t+1) Update formula
442             self.v_moments[param] = (self.v_moments[param] * beta2) + ((1
    - beta2) * (partial_of_loss_wrt_param ** 2)) # v(t+1) Update formula
443
444             mhat = self.m_moments[param] / (1 - beta1 ** self.k) # Unbias
    the moments from 0
445             vhat = self.v_moments[param] / (1 - beta2 ** self.k)
446
447             step = self.learning_rate * (mhat / math.sqrt(vhat + self.
    epsilon))
448             self.vals_for_learnable_params[param] += step
449
450         ##  Finally we update the biases at all the nodes that aggregate
    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)
```

```
                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

if __name__ == '__main__':
    # ================= Config Parameters===========
    lr = 1e-2

    # ================= Initialization of One NN with SGD plus optimization: ==================
    #Run with only SGD then SGD plus
    # cgp =  SGDplus(
    #            one_neuron_model = True,
    #            expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
    #            output_vars = ['xw'],
    #            dataset_size = 5000,
    #            learning_rate = lr,
    #            training_iterations = 20000,
    #            batch_size = 8,
    #            display_loss_how_often = 100,
    #            debug = True,
    #   )
    # cgp.parse_expressions()
    # training_data = cgp.gen_training_data()
    # sgd_losses = cgp.run_training_loop_one_neuron_model(training_data, beta = 0)
    # sgdplus_losses = cgp.run_training_loop_one_neuron_model(training_data, beta = 0.9)

    # betas = [0, 0.9, 0.1, 0.5]

    # for beta in betas:
    #     sgd_losses = cgp.run_training_loop_one_neuron_model(training_data, beta = beta)
    #     plt.plot(sgd_losses, label= "SGD beta = "+str(beta))

    # plt.legend()
    # plt.xlabel('Iterations')
    # plt.ylabel('Loss Value')
    # plt.title('SGD Plus for One NN with Various Betas')
    # plt.show()
```

```
    # ================= Initialization of Multi NN with SGD plus optimization: ==================
    cgp = SGDplus(
                num_layers = 3,
                layers_config = [4,2,1],                          # num of nodes in each layer
                expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                               'xz=bp*xp+bq*xq+br*xr+bs*xs',
                               'xo=cp*xw+cq*xz'],
                output_vars = ['xo'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 20000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
       )

    cgp.parse_multi_layer_expressions()
    training_data = cgp.gen_training_data()
    # betas = [0, 0.9, 0.1, 0.5]

    # for beta in betas:
    #     sgd_losses = cgp.run_training_loop_multi_neuron_model(training_data, beta = beta)
    #     plt.plot(sgd_losses, label= "SGD beta = "+str(beta))
    sgd_losses = cgp.run_training_loop_multi_neuron_model(training_data, beta = 0)
    sgdplus_losses = cgp.run_training_loop_multi_neuron_model(training_data, beta = 0.5)

    # plt.legend()
    # plt.xlabel('Iterations')
    # plt.ylabel('Loss Value')
    # plt.title('SGD Plus for Multi NN with Various Betas')
    # plt.show()

    # ================= Initialization of One NN with ADAM optimization: ==================
    # cgp =  ADAMOptimizer(
    #            one_neuron_model = True,
    #            expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
    #            output_vars = ['xw'],
    #            dataset_size = 5000,
    #            learning_rate = lr,
    #            training_iterations = 20000,
    #            batch_size = 8,
    #            display_loss_how_often = 100,
    #            debug = True,
    #    )
    # cgp.parse_expressions()
```

```
    # training_data = cgp.gen_training_data()

    # beta1s, beta2s = [0.8, 0.95, 0.99], [0.85, 0.9, 0.95]
    # for beta1 in beta1s:
    #     for beta2 in beta2s:
    #         start_time = time.time()
    #         adamlosses = cgp.run_training_loop_one_neuron_model(training_data, beta1 = beta1, beta2= beta2)
    #         plt.plot(adamlosses, label= "ADAM B1="+str(beta1)+" B2="+str(beta2))
    #         time_taken = time.time() - start_time

    #         final_loss = adamlosses[-1]
    #         minloss = np.min(adamlosses[10:])
    #         print(f'B1: {beta1}, B2: {beta2}[Min loss: {minloss}, final loss: {final_loss} Time taken: {time_taken}s')

    # plt.plot(sgd_losses, label = "SGD")

    # plt.plot(sgdplus_losses, label = "SGD Plus")
    # plt.legend()
    # plt.xlabel('Iterations')
    # plt.ylabel('Loss Value')
    # plt.title('ADAM for One NN with Various Betas, lr = '+ str(lr))
    # plt.show()

    # beta1, beta2 = 0.95, 0.85
    # adamlosses = cgp.run_training_loop_one_neuron_model(training_data, beta1 = beta1, beta2= beta2)

    # ================ ADAM Optimizer for Multi NN ====================

    cgp = ADAMOptimizer(
                num_layers = 3,
                layers_config = [4,2,1],                        # num of nodes in each layer
                expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                               'xz=bp*xp+bq*xq+br*xr+bs*xs',
                               'xo=cp*xw+cq*xz'],
                output_vars = ['xo'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 20000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
       )

    cgp.parse_multi_layer_expressions()
    training_data = cgp.gen_training_data()
```

```
    # beta1s, beta2s = [0.8, 0.95, 0.99], [0.85, 0.9, 0.95]

    # for beta1 in beta1s:
    #     for beta2 in beta2s:
    #         start_time = time.time()
    #         adamlosses = cgp.run_training_loop_multi_neuron_model(
training_data, beta1 = beta1, beta2= beta2)
    #         plt.plot(adamlosses, label= "ADAM B1="+str(beta1)+" B2="+str
(beta2))
    #         time_taken = time.time() - start_time

    #         final_loss = adamlosses[-1]
    #         minloss = np.min(adamlosses[10:])

    #         print(f'B1: {beta1}, B2: {beta2}[Min loss: {minloss}, final
loss: {final_loss} Time taken: {time_taken}s')

    # plt.plot(sgd_losses, label = "SGD Plus")
    # plt.legend()
    # plt.xlabel('Iterations')
    # plt.ylabel('Loss Value')
    # plt.title('ADAM for Multi NN with Various Betas, lr = '+ str(lr))
    # plt.show()

    beta1, beta2 = 0.99, 0.9
    adamlosses = cgp.run_training_loop_multi_neuron_model(training_data,
beta1 = beta1, beta2= beta2)

    plt.plot(sgd_losses, label = "SGD")
    plt.plot(sgdplus_losses, label = 'SGD Plus')
    plt.plot(adamlosses, label = 'ADAM')
    plt.xlabel('Iterations')
    plt.ylabel('Loss Value')
    plt.title('Comparison of Different Optimizers for Multi-NN')
    plt.legend()
    plt.show()
```

Code Listing 1: hw3.py

extra$_c$*red.py*

```
from ComputationalGraphPrimer import ComputationalGraphPrimer
import numpy as np
import random, operator, math, time
import matplotlib.pyplot as plt
from tqdm import tqdm
from collections import defaultdict

random.seed(0)
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)
        - 1
22          return scaled_data
23
24      def run_training_loop_one_neuron_model(self, training_data, normalize
       = True, truncate = False):
25          # Initialize learnable params for network -----------------
26          self.vals_for_learnable_params = {param: random.uniform(0,1) for
       param in self.learnable_params}
27          self.bias = random.uniform(0,1)
28
29          # Truncate data values
30
31          if truncate:
32              class0, class1 = self.truncate(training_data[0]), self.
       truncate(training_data[1])
33              training_data[0], training_data[1] = class0, class1
34          # Dataloader business -----------------
35          class DataLoader:
36              def __init__(self, training_data, batch_size):
37                  self.training_data = training_data
38                  self.batch_size = batch_size
39                  self.class_0_samples = [(item, 0) for item in self.
       training_data[0]]   ## Associate label 0 with each sample
40                  self.class_1_samples = [(item, 1) for item in self.
       training_data[1]]   ## Associate label 1 with each sample
41
42              def __len__(self):
43                  return len(self.training_data[0]) + len(self.training_data
       [1])
44
45              def _getitem(self):
46                  cointoss = random.choice([0,1])
        ## When a batch is created by getbatch(), we want the
47
        ##   samples to be chosen randomly from the two lists
48                  if cointoss == 0:
49                      return random.choice(self.class_0_samples)
50                  else:
51                      return random.choice(self.class_1_samples)
```

```
52
53              def getbatch(self):
54                  batch_data,batch_labels = [],[]
     ## First list for samples, the second for labels
55                  maxval = 0.0
     ## For approximate batch data normalization
56                  for _ in range(self.batch_size):
57                      item = self._getitem()
58                      if np.max(item[0]) > maxval:
59                          maxval = np.max(item[0])
60                      batch_data.append(item[0])
61                      batch_labels.append(item[1])
62                  if normalize: batch_data = [item/maxval for item in
    batch_data]           ## Normalize batch data
63                  batch = [batch_data, batch_labels]
64                  return batch
65
66          data_loader = DataLoader(training_data, batch_size=self.batch_size
    )
67          loss_running_record = []
68          i = 0
69          avg_loss_over_iterations = 0.0
    ##  Average the loss over iterations for printing out
70
71          # Actually running training -----------------
72          for i in range(self.training_iterations):
73              data = data_loader.getbatch()
74              data_tuples_in_batch = data[0]
75              class_labels_in_batch = data[1]
76
77              y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(
    data_tuples_in_batch)      ##  FORWARD PROP of data
78              loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
    for i in range(len(class_labels_in_batch))])  ##  Find loss
79              avg_loss_over_iterations += loss / float(len(
    class_labels_in_batch))
80
81              if i%(self.display_loss_how_often) == 0:
82                  avg_loss_over_iterations /= self.display_loss_how_often
83                  loss_running_record.append(avg_loss_over_iterations)
84                 # print("[iter=%d]  loss = %.4f" %  (i+1,
    avg_loss_over_iterations))                 ## Display average loss
85                  avg_loss_over_iterations = 0.0
                              ## Re-initialize avg loss
86
87              y_errors_in_batch = list(map(operator.sub,
    class_labels_in_batch, y_preds))
88              self.backprop_and_update_params_one_neuron_model(
    data_tuples_in_batch, y_preds, y_errors_in_batch, deriv_sigmoids)  ##
    BACKPROP loss
89          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])
        ## When a batch is created by getbatch(), we want the
105
        ##   samples to be chosen randomly from the two lists
106                 if cointoss == 0:
107                     return random.choice(self.class_0_samples)
108                 else:
109                     return random.choice(self.class_1_samples)
110
111             def getbatch(self):
112                 batch_data,batch_labels = [],[]
        ## First list for samples, the second for labels
113                 maxval = 0.0
        ## For approximate batch data normalization
114                 for _ in range(self.batch_size):
115                     item = self._getitem()
116                     if np.max(item[0]) > maxval:
117                         maxval = np.max(item[0])
118                     batch_data.append(item[0])
119                     batch_labels.append(item[1])
120                 if normalize: batch_data = [item/maxval for item in
        batch_data]             ## Normalize batch data
121                 batch = [batch_data, batch_labels]
122                 return batch
123
124         # Truncate data values
125         if truncate:
126             class0, class1 = self.truncate(training_data[0]), self.
        truncate(training_data[1])
127             training_data[0], training_data[1] = class0, class1
128         # Initialize Learnable params
129         self.vals_for_learnable_params = {param: random.uniform(0,1) for
        param in self.learnable_params}
```

```
        self.bias =   {i : [random.uniform(0,1) for j in range( self.layers_config[i] ) ]  for i in range(1, self.num_layers)}
        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0

        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            self.forward_prop_multi_neuron_model(data_tuples)
                                ## FORW PROP works by side-effect
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_layers-1]           ## Predictions from FORW PROP
            y_preds =  [item for sublist in  predicted_labels_for_batch for item in sublist]       ## Get numeric vals for predictions
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss / float(len(class_labels))
                                ## Average the loss over batch
            avg_loss_over_iterations += loss_avg
                                ## Add to Average loss over iterations
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
               # print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))
                avg_loss_over_iterations = 0.0
            y_errors_in_batch = list(map(operator.sub, class_labels, y_preds))
            self.backprop_and_update_params_multi_neuron_model(y_preds, y_errors_in_batch)
        return loss_running_record

if __name__ == '__main__':
    # ================= Compare One NN with and without normalizaiton: ==================
    #Run with only SGD then SGD plus
    # cgp =  CustomDataloader(
    #             one_neuron_model = True,
    #             expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
    #             output_vars = ['xw'],
    #             dataset_size = 5000,
    #             learning_rate = 1e-3,
    #             training_iterations = 40000,
    #             batch_size = 8,
    #             display_loss_how_often = 100,
    #             debug = True,
    #    )
```

```
    # cgp.parse_expressions()
    # training_data = cgp.gen_training_data()
    # custom_losses_onenn = cgp.run_training_loop_one_neuron_model(training_data, normalize= False)
    # regular_losses_onenn = cgp.run_training_loop_one_neuron_model(training_data, normalize= True )

    # plt.plot(custom_losses_onenn, label = "No Normalization")
    # plt.plot(regular_losses_onenn, label = 'With Normalization')
    # plt.legend()
    # plt.xlabel('Iterations')
    # plt.ylabel('Loss Value')
    # plt.title('One NN Comparison of Normalization')
    # plt.show()

    # ================= Compare One NN with and without normalizaiton: ==================
    # #Run with only SGD then SGD plus
    # cgp = CustomDataloader(
    #            num_layers = 3,
    #            layers_config = [4,2,1],                          # num of nodes in each layer
    #            expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
    #                           'xz=bp*xp+bq*xq+br*xr+bs*xs',
    #                           'xo=cp*xw+cq*xz'],
    #            output_vars = ['xo'],
    #            dataset_size = 5000,
    #            learning_rate = 9e-2,
    #            training_iterations = 20000,
    #            batch_size = 8,
    #            display_loss_how_often = 100,
    #            debug = True,
    #   )

    # cgp.parse_multi_layer_expressions()
    # training_data = cgp.gen_training_data()
    # custom_losses_onenn = cgp.run_training_loop_multi_neuron_model(training_data, normalize= False)
    # regular_losses_onenn = cgp.run_training_loop_multi_neuron_model(training_data, normalize= True )

    # plt.plot(custom_losses_onenn, label = "No Normalization")
    # plt.plot(regular_losses_onenn, label = 'With Normalization')
    # plt.legend()
    # plt.xlabel('Iterations')
    # plt.ylabel('Loss Value')
    # plt.title('Multi NN Comparison of Normalization')
    # plt.show()

    # # ================= Compare One NN with and without truncation: ==================
```

```
    # #Run with only SGD then SGD plus
    # cgp =  CustomDataloader(
    #            one_neuron_model = True,
    #            expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
    #            output_vars = ['xw'],
    #            dataset_size = 5000,
    #            learning_rate = 1e-3,
    #            training_iterations = 40000,
    #            batch_size = 8,
    #            display_loss_how_often = 100,
    #            debug = True,
    #   )
    # cgp.parse_expressions()
    # training_data = cgp.gen_training_data()
    # print(training_data)
    # custom_losses_onenn = cgp.run_training_loop_one_neuron_model(training_data, truncate= True)
    # regular_losses_onenn = cgp.run_training_loop_one_neuron_model(training_data, truncate= False )

    # plt.plot(custom_losses_onenn, label = "With Truncation")
    # plt.plot(regular_losses_onenn, label = 'Without Truncation')
    # plt.legend()
    # plt.xlabel('Iterations')
    # plt.ylabel('Loss Value')
    # plt.title('One NN Comparison of Truncation')
    # plt.show()


    # ================= Compare One NN with and without truncation: ==================
    #Run with only SGD then SGD plus
    cgp = CustomDataloader(
               num_layers = 3,
               layers_config = [4,2,1],                          # num of nodes in each layer
               expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                              'xz=bp*xp+bq*xq+br*xr+bs*xs',
                              'xo=cp*xw+cq*xz'],
               output_vars = ['xo'],
               dataset_size = 5000,
               learning_rate = 9e-2,
               training_iterations = 20000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

    cgp.parse_multi_layer_expressions()
    training_data = cgp.gen_training_data()

```

```
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$_c$*red.py*