ECE 60146 Homework 3
Nikkhil Vijaya Sankar – nvijayas@purdue.edu

1- Exploring the Primer

Figure 1 shows the loss-vs-iterations plot produced by multi_neuron_classifer.py and Figure 2 shows the loss-vs-iterations plot produced by verify_with_torchnn.py in the multi neuron configuration. From inspecting these plots, we can see that both models are learning since the loss is continually decreasing. However, when we take a closer look at the values of the loss. The torch.nn model performs worse in as its final loss value hovered around 6, while the manually created multi neuron model's loss hovered around 0.10. We also notice that in Figure 1 the loss didn't improve after about 60 iterations suggesting that learning is complete. However, in Figure 2 we can see that the loss is continually improving suggesting that more iterations of learning can take place. We can also notice that torch.nn is more smooth in its loss than the manual method.

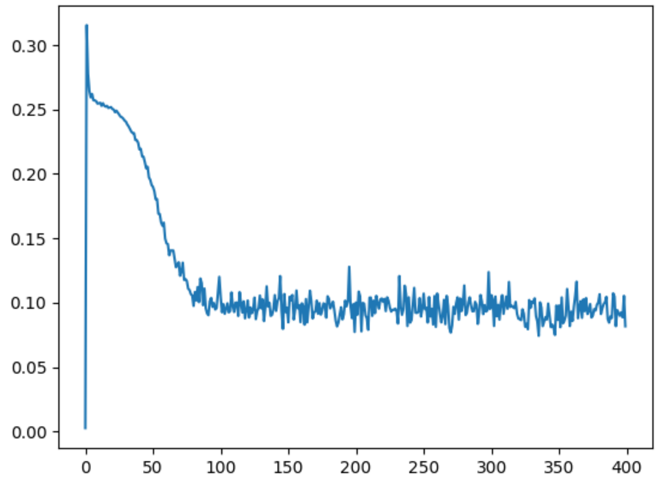


Figure 1: loss-vs-iteration plot of multi_neuron_classifer.py

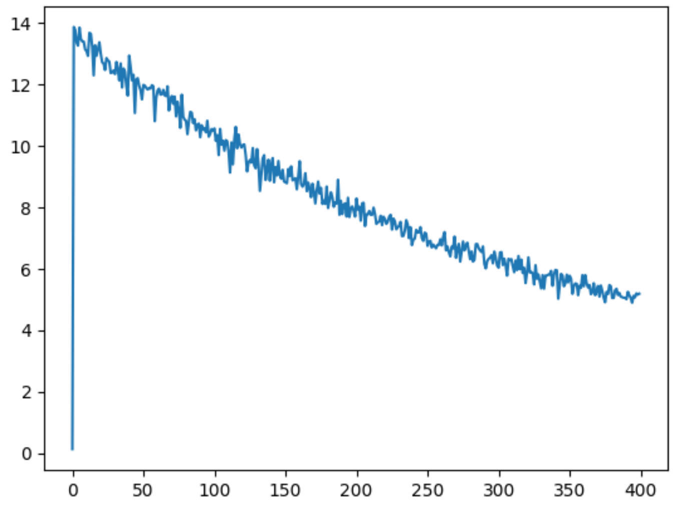


Figure 2: loss-vs-iteration plot of verify_with_torchnn.py (multi-neuron config)

Figure 3 and Figure 4 show the same plots as Figure 1 and Figure 2 but for the one-neuron scenario. A similarity in Figures 3 and 4 is that both have decreasing loss suggesting that both models are indeed learning. However, it's the rate at which they are learning is the difference. For instance, Figure 3's loss is more linear whereas Figure 4's loss is more exponential decay. However, Figure 4's model performs much worse as its loss hovers around 1 at the end of training, but the loss for Figure 3's model hovers around 0.20 at the end of training. Nevertheless, using torch.nn is still better to use as it includes optimizations to better update the parameters.

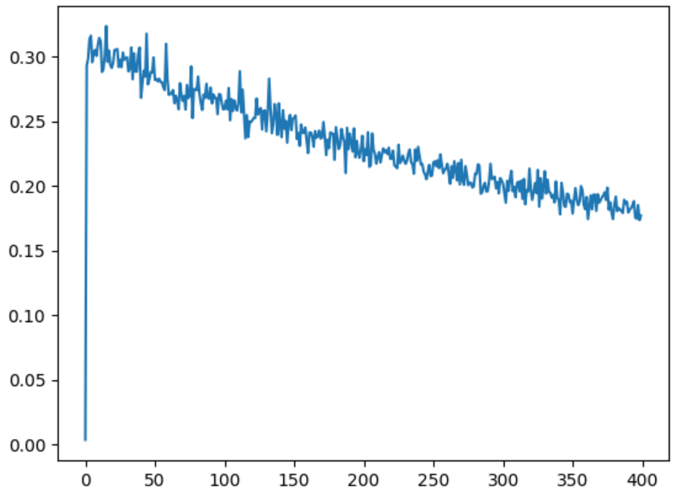


Figure 3: loss-vs-iteration plot of one_neuron_classifer.py

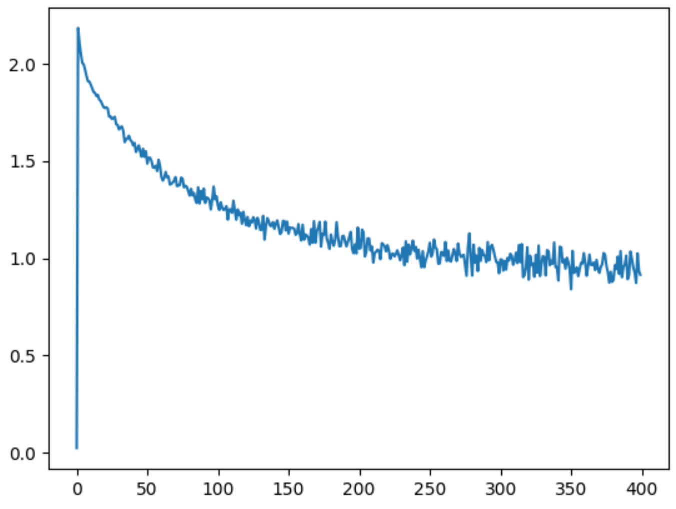


Figure 4: loss-vs-iteration plot of verify_with_torchnn.py (one-neuron config)

2- Description of SGD+ and Adam

a. SGD+

We all know that Stochastic Gradient Decent (SGD) is a popular method to optimize a loss function in any machine learning model. SGD+ is an extension of SGD that uses the concept of momentum to speed up convergence. It does so by increasing the values of the gradients of the parameters. In SGD's journey of finding the minimum of the loss's hyperplane, if we realize that we have been correctly descending to the bottom of the valley, then we can make a greater jump to get to the bottom faster. This faster jump or the idea of taking a larger leap is called momentum.

The typical update rule for SGD is as follows, $\mathrm{p_{t+1} = p_t - lr \cdot g_{t+1}}$. Here p_{t+1} is the update to be made, the current update p_t, the learning rate lr, and the next gradient g_{t+1}.

To convert the mentioned equation to SGD+, we need to consider the previous update (the step that was taken last iteration). We still want to make sure we consider the current update as well, since if the current gradient is the opposite direction of the previous update, then we know we went passed the minimum and we need to self-correct. Therefore, we take a fraction of the previous update. The fractional amount of the previous update to consider is determined by the variable called momentum. The concepts can be mathematically represented by the following equations.

$$v_{t+1} = \mathrm{\mu} \cdot v_t + g_{t+1}$$
$$p_{t+1} = p_t - lr \cdot v_{t+1}$$

In these equations, the momentum is μ, the previous update is v_t, and v_{t+1} can be seen as a temporary variable.

When comparing the equations of SGD and SGD+, we can see that there is an additional step before we modify the update in SGD+. That additional step is essentially, $\mu \cdot v_t$. The value of μ can range from 0 to 1. A value of 1 means we consider the entire amount of the previous update. Alternatively, a value of 0 means we don't consider the previous update at all. Notice that setting $\mu = 0$ simplifies the equations to be the same as SGD.

b. ADAM

Adaptive Moment Estimation, or ADAM for short, is an improved optimizer and is known to work very well. Adam combines the concepts of momentum and adaptability. We already know that momentum helps reach convergence faster. It does so by checking the previous update and if it points at the same direction as the current update, then it will take a larger jump. The concept of adaptability addresses the issue of sparse gradients. It is often the case that gradients are zeros, meaning there is not much to learn at a particular iteration in training. However, in the rare instance that there is a non-zero gradient, then we know this particular gradient contains vital information, and as a result we should give it more attention.

The following equations show mathematically what is mentioned in the previous paragraph.

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

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

$$p_{t+1} = p_t - lr \cdot \frac{\widehat{m_{t+1}}}{\sqrt{\widehat{v_{t+1}} + \epsilon}}$$

The first equation incorporates momentum by taking a running weighted average of the gradients. As we can see, the weight of the previous gradients is determined by β_1. Setting $\beta_1 = 0$ results in not using momentum at all. The second equation incorporates adaptability by taking a running weighted average of the gradient's variance. Like the case with β_1, setting $\beta_2 = 0$ will mean not considering the previous variances. The third equation is our parameter update equation. The difference here from the original SGD update equation is that the gradient term is replaced with $\frac{\widehat{m_{t+1}}}{\sqrt{\widehat{v_{t+1}} + \epsilon}}$. The numerator of this fraction is essentially the gradient with momentum. The denominator of the fraction handles the inverse relationship with variance. The logic behind this is that when high variance is observed Adam needs to slow down its learning. Additionally, when low variance is observed Adam needs to speed up learning because the gradients don't contain new knowledge to gain. Lastly, the ϵ is added to avoid a divide by zero error: typically, $\epsilon = 1e - 8$.

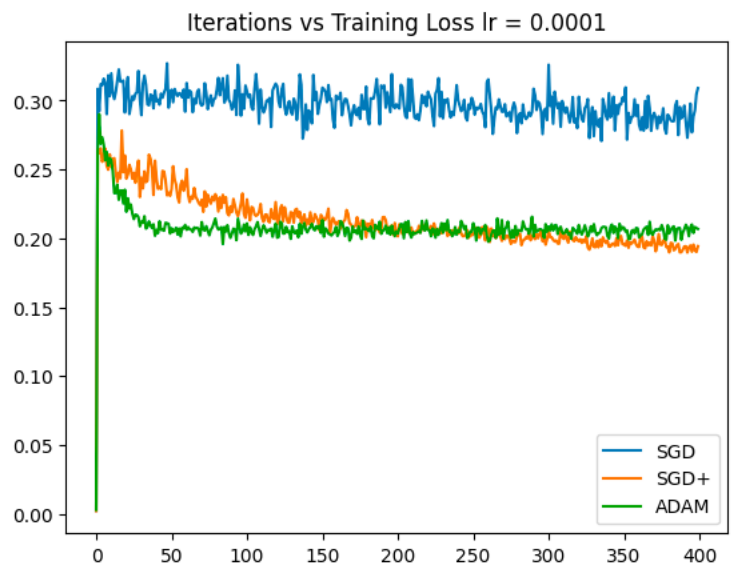
While this is the essence of Adam, an issue occurs in practice that requires us to make one small change. In the initial iterations of training, $m_t = v_t = 0$. These sway the averages of the mean and variance to 0 in the initial iterations. As a result, the following equations are used as corrected values for m_t and v_t.

$$m_{\text{corrected}} = \frac{m}{1-\beta_1^k} \qquad v_{\text{corrected}} = \frac{v}{1-\beta_2^k}$$

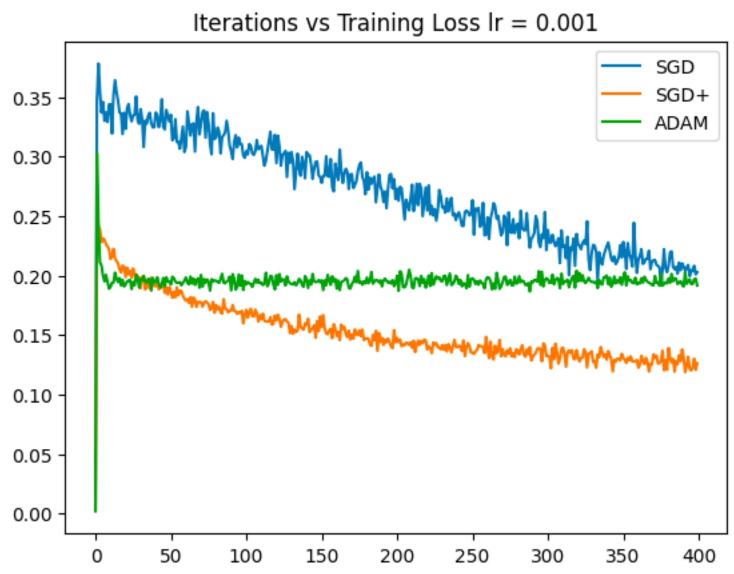
Here k indicates what iteration the training loop is in. The logic here is the following. In the beginning stages of training (small k), the denominator will be a small value resulting in large adjustments to m_t and v_t. While we approach the later stages of training (large k), the denominator will approach 1 and not correct our m_t and v_t. This is perfect because it's in the beginning where we need correction to avoid the 0 bias caused by setting $m_t = v_t = 0$.

3- Implementation

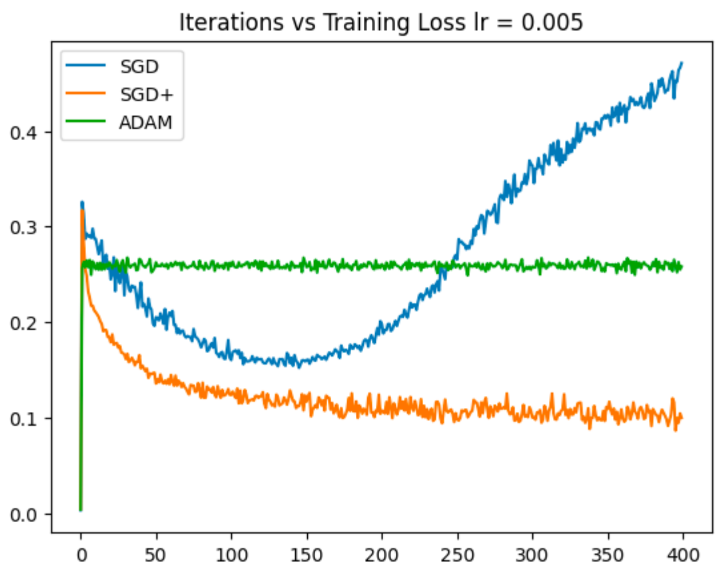
a. One Neuron



Figures 5: One Neuron Loss Plot (lr = 0.0001)



Figures 6: One Neuron Loss Plot (lr = 0.001)

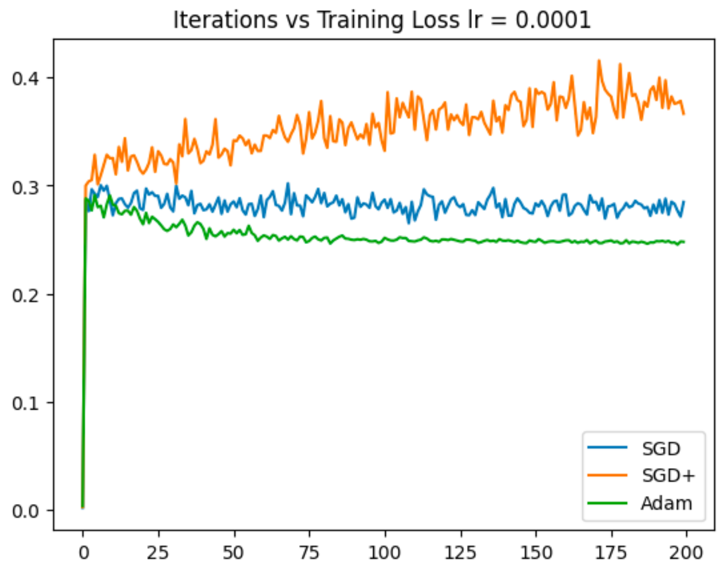


Figures 7: One Neuron Loss Plot (lr = 0.005)

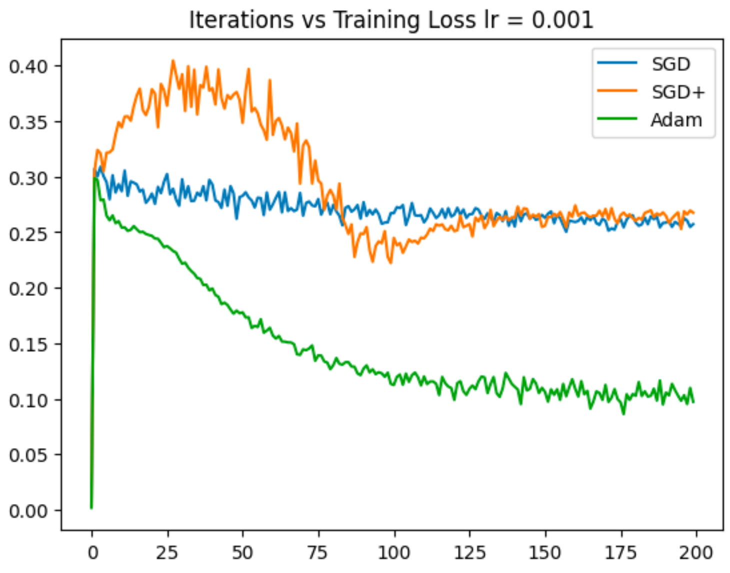
Figure 5 – 7 show the iterations vs loss plot of the one neuron model at different learning rates with 3 different optimizers: SGD, SGD+, and Adam. In regard to smoothness, it is very evident that Adam has the smoothest loss plots. We can see this since the green line in all these plots are very thin meaning there is less variation. SGD+ is also considerably smooth but not as smooth as the Adam. However, the least smooth is SGD. Figure 5 shows that when the learning rate is low, the SGD loss is very volatile and not smooth at all. Regarding convergence, we can

see that with a small learning rate (0.0001) all the optimizers take longer to learn. While Adam stabilizes at a loss slightly above 0.20 very quickly, SGD+ can continually decrease its loss ever so slowly: shown in Figure 5. With a medium learning rate (0.001), SGD performs much better than how it did with a slower learning rate (Figure 6). Adam can stabilize its loss much quicker but doesn't decrease the loss at all (Figure 6). However, SGD+ outperforms both optimizers by getting to a loss below 0.15 (Figure 6). Lastly, with a high learning rate (0.005) we in fact see divergence in SGD. This is likely due to SGD overshooting the minimum and never being able to return back to the bottom of the valley (Figure 7). This is where we see the beauty of momentum as SGD+ is able to remember the past gradients and continue its path to convergence (Figure 7). Adam again stabilizes very quickly but is not able to improve its loss (Figure 7).

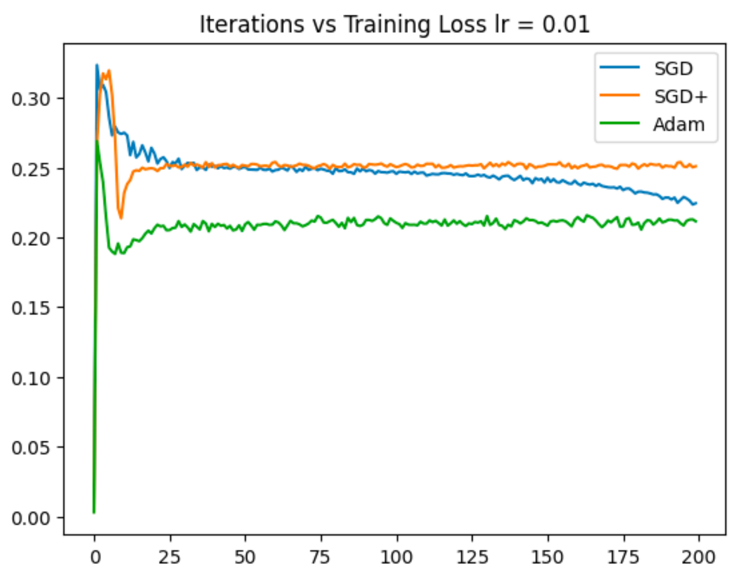
b. Multi Neuron



Figures 8: Multi Neuron Loss Plot (lr = 0.0001)



Figures 9: Multi Neuron Loss Plot (lr = 0.01)



Figures 10: Multi Neuron Loss Plot (lr = 0.01)

Figure 8 – 10 show the iterations vs loss plot of the multi neuron model at different learning rates with 3 different optimizers: SGD, SGD+, and Adam. In regard to smoothness, a high learning rate yielded the smoothest loss for all 3 optimizers (Figure 10). In general, Adam had smooth loss curves except in the case of Figure 9 where Adam had larger variances towards the end of training. SGD has overall the least smoothness. SGD did however smooth out at the end of training in Figure 9. Regarding convergence, different learning rates didn't change if a model converged or not, except for the case in SGD+ with a small learning rate (Figure 8). This observation is peculiar and could be a result of bad initialization weights. It is clear that a learning rate of 0.001 with Adam yielded the lowest loss (Figure 9). Additionally, SGD and SGD+ performed the best with a higher learning rate of 0.01 (Figure 10). In fact, SGD's loss starts to decrease toward the end of training suggesting that the loss can be further improved with more iterations (Figure 10).

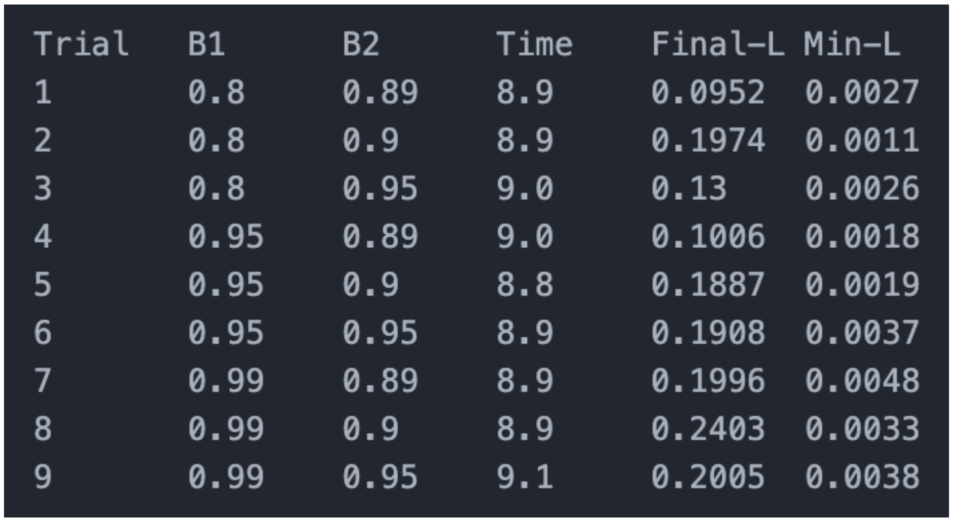
c. Findings

A lot can be learned from the conducted experiments. The overall take from this is that Adam tends to produce smoother loss curves which can be attributed to the consideration of the second moment. Additionally, Adam may not converge to the lowest loss, but it will stabilize the fastest. This is likely due to the combination of momentum and adaptability. If we were to rank the optimizer, SGD+ will be the runner up as it too can converge quickly but not as fast as Adam. This again is due to the consideration of momentum. The additional benefit of momentum is that SGD+ is able to avoid divergence where ordinary SGD cannot. As seen in the case of Figure 7, SGD overshot its estimation but since SGD+ is able to remember its previous gradient it avoided that issue. This brings SGD to last place where it experiences high variance and generally takes longer to converge.

It is important to mention that the different learning rates play a curial role in the performance of these optimizers. Given enough effort, one can easily find the right learning rate that will sway one optimizer to perform better than the other. This only shows that hyperparameter tuning is critical when training deep learning models.

d. Adam Optimizer Performance

Figure 11 shows the results of testing Adam optimizer with various beta 1 and beta 2 values. From the results the duration of training with different beta values plays no significant role. All the models finished training in about 8.8 to 9.1 seconds. The worst performing models occurred in trials 8 and 9. This was when the beta 1 value was 0.99 and beta 2 values were 0.9 and 0.95. The best performing model was trial 1 where beta 1 = 0.8 and beta 2 = 0.89. The second best model was trial 4 with beta 1 = 0.95 and beta 2 = 0.89. It's important to make clear that I determine "best model" by looking at the lowest final loss. Based on these trends, it seems that Adam performs better when both beta values are a little less than the normal 0.9 and 0.99 values that are traditionally used. This means Adam performs better when given slightly more importance to the current first and second moments. That is essentially the effect that the beta values play in Adam. The lower the beta values, the more weightage is given to the current moments when averaging. Inversely, the higher the beta values, the less weightage is given to the current moments when averaging.



```
Trial   B1      B2      Time    Final-L Min-L
1       0.8     0.89    8.9     0.0952  0.0027
2       0.8     0.9     8.9     0.1974  0.0011
3       0.8     0.95    9.0     0.13    0.0026
4       0.95    0.89    9.0     0.1006  0.0018
5       0.95    0.9     8.8     0.1887  0.0019
6       0.95    0.95    8.9     0.1908  0.0037
7       0.99    0.89    8.9     0.1996  0.0048
8       0.99    0.9     8.9     0.2403  0.0033
9       0.99    0.95    9.1     0.2005  0.0038
```

Figures 11: Table of Tested B1 and B2 Values for Adam

4- Source Code

```python
# %%
import sys
sys.path.append("/Users/nikkhil/Documents/ECE 60146/hw3/ComputationalGraphPrimer-1.1.4/ComputationalGraphPrimer")

import random
import numpy as np
import matplotlib.pyplot as plt
import operator
from ComputationalGraphPrimer import *
import time

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

# %% [markdown]
# Create Subclass that returns the loss

# %%
# NOTE: This subclass is created so the loss is returned so the values can be used for plotting
class ComputationalGraphPrimer_ReturnLoss(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super(ComputationalGraphPrimer_ReturnLoss, self).__init__(*args, **kwargs)

    # NOTE: Code is taken from https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.1.4_CodeOnly.html
    #       The only modification made is the loss_running_record list is returned and the generated plots are commented out
    def run_training_loop_one_neuron_model(self, training_data):
        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
                                                                           ## every N iterations during the training loop.
        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))
            self.backprop_and_update_params_one_neuron_model(data_tuples_in_batch, y_preds, y_errors_in_batch, deriv_sigmoids)  ## BACKPROP loss
        #plt.figure()
        #plt.plot(loss_running_record)
        #plt.show()
        return loss_running_record

    # NOTE: Code is taken from https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.1.4_CodeOnly.html
    #       The only modification made is the loss_running_record list is returned and the generated plots are commented out
    def run_training_loop_multi_neuron_model(self, training_data):

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

# %% [markdown]
# Create SGD+ (one & multi neuron)

# %%
```

```python
# NOTE: Create a subclass of ComputationalGraphPrimer - SGD+
#       In this class the backprop functions are modified to use SGD+
class ComputationalGraphPrimer_ReturnLoss_SGD_Plus(ComputationalGraphPrimer_ReturnLoss):
    def __init__(self, *args, **kwargs):
        super(ComputationalGraphPrimer_ReturnLoss_SGD_Plus, self).__init__(*args, **kwargs)

    # NOTE: Function created to set up all variables needed for carrying out momentum
    def initializeMomentumStuff_OneNeuron(self, momentum):
        self.mu = momentum
        # Previous updates for weights - set everything to 0 at first
        self.param_updates = {key: 0.0 for key in self.learnable_params}
        # Previous update for bias - set to 0 at first
        self.bias_update = 0.0

    # NOTE: Function created to set up all variables needed for carrying out momentum
    def initializeMomentumStuff_MultiNeuron(self, momentum):
        self.muMulti = momentum
        # Previous updates for weights - set everything to 0 at first
        self.param_updates = {key: 0.0 for key in self.learnable_params}
        # Previous update for bias - set to 0 at first
        self.bias_updates = {i: [0.0 for _ in range(self.layers_config[i])] for i in range(1, self.num_layers)}

    # NOTE: Code is taken from https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.1.4_CodeOnly.html
    #       Most of this function is the same as Prof Avinash Kak's.
    #       Modifications made are indicated by comments.
    def backprop_and_update_params_one_neuron_model(self, data_tuples_in_batch, predictions, y_errors_in_batch, deriv_sigmoids):
        input_vars = self.independent_vars
        input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]  ## These two statements align the
        param_to_vars_map = {param : var for var, param in input_vars_to_param_map.items()}   ##   the input vars
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## For each param, sum the partials from every training data sample in batch
            partial_of_loss_wrt_param = 0.0
            for j in range(self.batch_size):
                vals_for_input_vars_dict =  dict(zip(input_vars, list(data_tuples_in_batch[j])))
                partial_of_loss_wrt_param   +=   -  y_errors_in_batch[j] * vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[j]
            partial_of_loss_wrt_param /=  float(self.batch_size)
            # Modification ---- Start
            # Here I am using the past parameter update and momentum to slighly modify the gradient
            # This is then used to update self.vals_for_learnable_params[param]
            # Equation set 2 from the hw description file is being implemented here
            self.param_updates[param] = self.mu * self.param_updates[param] - self.learning_rate * partial_of_loss_wrt_param
            self.vals_for_learnable_params[param] += self.param_updates[param]
            # Modification ---- End

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

        # Modification ---- Start
        # Like the weights the bias is also being updated in the same manner
        # Again equation set 2 from the hw description file is being implemented here
        self.bias_update = self.mu * self.bias_update + self.learning_rate * y_error_avg * deriv_sigmoid_avg
        self.bias += self.bias_update
        # Modification ---- End

    # NOTE: Code is taken from https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.1.4_CodeOnly.html
    #       Most of this function is the same as Prof Avinash Kak's.
    #       Modifications made are indicated by comments.
    def backprop_and_update_params_multi_neuron_model(self, predictions, y_errors):
        ## 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:
        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) ]
        ## This will store "\delta L / \delta w" you see at the LHS of the equations on Slide 73:
        partial_of_loss_wrt_params = {param : 0.0 for param in self.all_params}
        ## For estimating the changes to the bias to be made on the basis of the derivatives of the Sigmoids:
        bias_changes =   {i : [0.0 for j in range( self.layers_config[i] ) ] for i in range(1, self.num_layers)}
        for b in range(self.batch_size):
                pred_err_backproped_at_layers[b][self.num_layers - 1] = [ y_errors[b] ]
                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
                    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
                    deriv_sigmoids =  self.gradient_vals_for_layers[back_layer_index]   ## This is a list eight one-element lists, one for each batch element
                    vars_in_layer  =  self.layer_vars[back_layer_index]                 ## A list like ['xo']
                    vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## A list like ['xw', 'xz']
                    vals_for_input_vars_dict =  dict(zip(vars_in_next_layer_back, self.forw_prop_vals_at_layers[back_layer_index - 1][b]))
                    ## For the next statement, note that layer_params are stored in a dict like
                    ##       {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']], 2: [['cp', 'cq']]}
                    ## "layer_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"
                    layer_params = self.layer_params[back_layer_index]
                    transposed_layer_params = list(zip(*layer_params))         ## Creating a transpose of the link matrix, See Eq. 30 on Slide 77
                    for k,var1 in enumerate(vars_in_next_layer_back):
                        for j,var2 in enumerate(vars_in_layer):
                            pred_err_backproped_at_layers[b][back_layer_index - 1][k] = sum([self.vals_for_learnable_params[transposed_layer_params[k][i]]
                                                                                  * pred_err_backproped_at_layers[b][back_layer_index][i]
                                                                                  for i in range(len(vars_in_layer))])
                    for j,var in enumerate(vars_in_layer):
                        layer_params = self.layer_params[back_layer_index][j]         ##  ['cp', 'cq']   for the end layer
                        input_vars_to_param_map = self.var_to_var_param[var]          ## These two statements align the    {'xw': 'cp', 'xz': 'cq'}
                        param_to_vars_map = {param : var for var, param in input_vars_to_param_map.items()}   ##   and the input vars   {'cp': 'xw', 'cq': 'xz'}

                        ##  Update the partials of Loss wrt to the learnable parameters between the current layer
                        ##  and the previous layer. You are accumulating these partials over the different training
                    ##  data samples in the batch being processed.  For each training data sample, the formula
                    ##  being used is shown in Eq. (29) on Slide 77 of my Week 3 slides:
                    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)
            # Modification ---- Start
            # Here I am using the past parameter update and momentum to slighly modify the gradient
            # This is then used to update self.vals_for_learnable_params[param]
            # Equation set 2 from the hw description file is being implemented here
            self.param_updates[param] = self.muMulti * self.param_updates[param] - self.learning_rate * partial_of_loss_wrt_param
            self.vals_for_learnable_params[param] += self.param_updates[param]
            # Modification ---- End

        ##  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]):
                # Modification ---- Start
                # Like the weights the bias is also being updated in the same manner
                # Again equation set 2 from the hw description file is being implemented here
                # A temporary variable 'temp_bias_change' is created to avoid having 1 long line of code
                temp_bias_change = bias_changes[layer_index][k] / float(self.batch_size)
```

```python
                self.bias_updates[layer_index][k] = self.muMulti * self.bias_updates[layer_index][k] + self.learning_rate * temp_bias_change
                self.bias[layer_index][k]  += self.bias_updates[layer_index][k]
                # Modification ---- End

# %% [markdown]
# Create ADAM (one & multi neuron)

# %%
# NOTE: Create a subclass of ComputationalGraphPrimer - ADAM
class ComputationalGraphPrimer_ReturnLoss_ADAM(ComputationalGraphPrimer_ReturnLoss):
    def __init__(self, *args, **kwargs):
        super(ComputationalGraphPrimer_ReturnLoss_ADAM, self).__init__(*args, **kwargs)

    # NOTE: Function created to set up all variables needed for carrying out adam
    # Epsilon is hard coded to be 1e-8 as it is the standard mentioned in PyTorch documentation
    def initializeADAMStuff_OneNeuron(self, beta1, beta2, epsilon=1e-8):
        self.b1 = beta1
        self.b2 = beta2
        self.ep = epsilon
        self.timestep = 0
        self.m = {param: 0 for param in self.learnable_params}
        self.v = {param: 0 for param in self.learnable_params}

    # NOTE: Function created to set up all variables needed for carrying out adam
    # Epsilon is hard coded to be 1e-8 as it is the standard mentioned in PyTorch documentation
    def initializeADAMStuff_MultiNeuron(self, beta1, beta2, epsilon=1e-8):
        self.b1 = beta1
        self.b2 = beta2
        self.ep = epsilon
        self.timestep = 0
        # For weights
        self.m = {param: 0 for param in self.learnable_params}
        self.v = {param: 0 for param in self.learnable_params}
        # For bias
        self.m_bias = {layer: [0 for _ in range(self.layers_config[layer])] for layer in range(1, self.num_layers)}
        self.v_bias = {layer: [0 for _ in range(self.layers_config[layer])] for layer in range(1, self.num_layers)}

    # NOTE: Code is taken from https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.1.4_CodeOnly.html
    #       Most of this function is the same as Prof Avinash Kak's.
    #       Modifications made are indicated by comments.
    def backprop_and_update_params_one_neuron_model(self, data_tuples_in_batch, predictions, y_errors_in_batch, deriv_sigmoids):

        # Modification ---- Start
        self.timestep += 1
        # Modification ---- End

        input_vars = self.independent_vars
        input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]    ## These two statements align the
        param_to_vars_map = {param : var for var, param in input_vars_to_param_map.items()}   ##   the input vars
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## For each param, sum the partials from every training data sample in batch
            partial_of_loss_wrt_param = 0.0
            for j in range(self.batch_size):
                vals_for_input_vars_dict =  dict(zip(input_vars, list(data_tuples_in_batch[j])))
                partial_of_loss_wrt_param   +=   -  y_errors_in_batch[j] * vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[j]
            partial_of_loss_wrt_param /=  float(self.batch_size)

            # Modification ---- Start
            # Here I am using Equation set 3 from the hw description file
            # The m_corrected and v_corrected are from Week 3's presentaion slide 118
            # These values are then used to update self.vals_for_learnable_params[param]
            # The bias is updated in teh same manner
            self.m[param] = self.b1 * self.m[param] + (1-self.b1) * partial_of_loss_wrt_param
            self.v[param] = self.b2 * self.v[param] + (1-self.b2) * (partial_of_loss_wrt_param ** 2)

            m_corr = self.m[param] / (1 - (self.b1**self.timestep))
            v_corr = self.v[param] / (1 - (self.b2**self.timestep))

            self.vals_for_learnable_params[param] -= self.learning_rate * m_corr / ((v_corr + self.ep) ** 0.5)
            self.bias -= self.learning_rate * m_corr / ((v_corr + self.ep) ** 0.5)
                # Modification ---- End

            # NOTE: Code is taken from https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.1.4_CodeOnly.html
            #       Most of this function is the same as Prof Avinash Kak's.
            #       Modifications made are indicated by comments.
            def backprop_and_update_params_multi_neuron_model(self, predictions, y_errors):

            # Modification ---- Start
            self.timestep += 1
            # Modification ---- End

            ## 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:
            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) ]
            ## This will store "\delta L / \delta w" you see at the LHS of the equations on Slide 73:
            partial_of_loss_wrt_params = {param : 0.0 for param in self.all_params}
            ## For estimating the changes to the bias to be made on the basis of the derivatives of the Sigmoids:
            bias_changes =   {i : [0.0 for j in range( self.layers_config[i] ) ] for i in range(1, self.num_layers)}
            for b in range(self.batch_size):
            pred_err_backproped_at_layers[b][self.num_layers - 1] = [ y_errors[b] ]
            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
            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
            deriv_sigmoids =  self.gradient_vals_for_layers[back_layer_index]   ## This is a list eight one-element lists, one for each batch element
            vars_in_layer  =  self.layer_vars[back_layer_index]                 ## A list like ['xo']
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## A list like ['xw', 'xz']
            vals_for_input_vars_dict =  dict(zip(vars_in_next_layer_back, self.forw_prop_vals_at_layers[back_layer_index - 1][b]))
            ## For the next statement, note that layer_params are stored in a dict like
```

```
                    ##       {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']], 2: [['cp', 'cq']]}
                    ## "layer_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"
                    layer_params = self.layer_params[back_layer_index]
                    transposed_layer_params = list(zip(*layer_params))         ## Creating a transpose of the link matrix, See Eq. 30 on Slide 77
                    for k,var1 in enumerate(vars_in_next_layer_back):
                        for j,var2 in enumerate(vars_in_layer):
                            pred_err_backproped_at_layers[b][back_layer_index - 1][k] = sum([self.vals_for_learnable_params[transposed_layer_params[k][i]]
                                                                                   * pred_err_backproped_at_layers[b][back_layer_index][i]
                                                                                   for i in range(len(vars_in_layer))])
                    for j,var in enumerate(vars_in_layer):
                        layer_params = self.layer_params[back_layer_index][j]         ##  ['cp', 'cq']   for the end layer
                        input_vars_to_param_map = self.var_to_var_param[var]          ## These two statements align the    {'xw': 'cp', 'xz': 'cq'}
                        param_to_vars_map = {param : var for var, param in input_vars_to_param_map.items()}   ##   and the input vars   {'cp': 'xw', 'cq': 'xz'}

                        ##  Update the partials of Loss wrt to the learnable parameters between the current layer
                        ##  and the previous layer. You are accumulating these partials over the different training
                        ##  data samples in the batch being processed.  For each training data sample, the formula
                        ##  being used is shown in Eq. (29) on Slide 77 of my Week 3 slides:
                        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]

```

```python
        ## Now update the learnable parameters.  The loop shown below carries out Adam
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = partial_of_loss_wrt_params[param] / float(self.batch_size)
            # Modification ---- Start
            # Here I am using Equation set 3 from the hw description file
            # The m_corrected and v_corrected are from Week 3's presentaion slide 118
            # These values are then used to update self.vals_for_learnable_params[param]
            self.m[param] = self.b1 * self.m[param] + (1-self.b1) * partial_of_loss_wrt_param
            self.v[param] = self.b2 * self.v[param] + (1-self.b2) * (partial_of_loss_wrt_param ** 2)

            m_corr = self.m[param] / (1 - (self.b1 ** self.timestep))
            v_corr = self.v[param] / (1 - (self.b2 ** self.timestep))

            self.vals_for_learnable_params[param] += self.learning_rate * m_corr / ((v_corr + self.ep) ** 0.5)
            # Modification ---- End

        ##  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]):
                # Modification ---- Start
                # The bias is updated in the same manner as the weights were updated
                # Again temp variable temp_bias_change for easy code readability
                temp_bias_change = (bias_changes[layer_index][k] / float(self.batch_size))

                self.m_bias[layer_index][k] = self.b1 * self.m_bias[layer_index][k] + (1 - self.b1) * temp_bias_change
                self.v_bias[layer_index][k] = self.b2 * self.v_bias[layer_index][k] + (1 - self.b2) * (temp_bias_change ** 2)

                m_bias_corr = self.m_bias[layer_index][k] / (1 - (self.b1 ** self.timestep))
                v_bias_corr = self.v_bias[layer_index][k] / (1 - (self.b2 ** self.timestep))

                self.bias[layer_index][k] += self.learning_rate * m_bias_corr / ((v_bias_corr + self.ep) ** 0.5)
                # Modification ---- End

```

```python

# %% [markdown]
# Test One Neuron (SGD, SGD+, ADAM)

# %%
lr = 5e-3

# %%
# Train One Neuron - SGD
oneN_sgd = ComputationalGraphPrimer_ReturnLoss(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 = 40000, batch_size = 8, display_loss_how_often = 100, debug = False,)

oneN_sgd.parse_expressions()
training_data = oneN_sgd.gen_training_data()
loss_oneN_sgd = oneN_sgd.run_training_loop_one_neuron_model(training_data)

# %%
# Train One Neuron - SGD+
oneN_sgdP = ComputationalGraphPrimer_ReturnLoss_SGD_Plus(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 = 40000, batch_size = 8, display_loss_how_often = 100,debug = False,)

oneN_sgdP.parse_expressions()
training_data = oneN_sgdP.gen_training_data()
oneN_sgdP.initializeMomentumStuff_OneNeuron(momentum=0.9)
loss_oneN_sgdP = oneN_sgdP.run_training_loop_one_neuron_model(training_data)

# %%
# Train One Neuron - ADAM
oneN_adam = ComputationalGraphPrimer_ReturnLoss_ADAM(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 = 40000, batch_size = 8, display_loss_how_often = 100, debug = False,)

oneN_adam.parse_expressions()
training_data = oneN_adam.gen_training_data()
oneN_adam.initializeADAMStuff_OneNeuron(beta1=0.9, beta2=0.99)
loss_oneN_adam = oneN_adam.run_training_loop_one_neuron_model(training_data)

```

```python
# %%
# Pot all the losses
plt.figure()
plt.plot(loss_oneN_sgd, label='SGD')
plt.plot(loss_oneN_sgdP, label='SGD+')
plt.plot(loss_oneN_adam, label='ADAM')
plt.legend()
plt.title(f"Iterations vs Training Loss lr = {lr}")
plt.show()

# %% [markdown]
# Test Multi Neuron (SGD, SGD+, ADAM)

# %%
#lr = 9e-2
lr = 1e-2
#lr = 5e-2
#lr = .01

# %%
multiN_sgd = ComputationalGraphPrimer_ReturnLoss(num_layers = 3, layers_config = [4,2,1], 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 = False,)

multiN_sgd.parse_multi_layer_expressions()
training_data = multiN_sgd.gen_training_data()
loss_multiN_sgd = multiN_sgd.run_training_loop_multi_neuron_model(training_data)

# %%
multiN_sgdP = ComputationalGraphPrimer_ReturnLoss_SGD_Plus(num_layers = 3,layers_config = [4,2,1], 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 = False,)

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

```
multiN_sgdP.initializeMomentumStuff_MultiNeuron(momentum=0.9)
loss_multi_SGD = multiN_sgdP.run_training_loop_multi_neuron_model(training_data)

# %%
multiN_adam = ComputationalGraphPrimer_ReturnLoss_ADAM(num_layers = 3,layers_config = [4,2,1], 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 = False,)

multiN_adam.parse_multi_layer_expressions()
training_data = multiN_adam.gen_training_data()
multiN_adam.initializeADAMStuff_MultiNeuron(beta1=0.9, beta2=0.99)
loss_multi_Adam = multiN_adam.run_training_loop_multi_neuron_model(training_data)

# %%
# Pot all the losses
plt.figure()
plt.plot(loss_multiN_sgd, label='SGD')
plt.plot(loss_multi_SGD, label='SGD+')
plt.plot(loss_multi_Adam, label='Adam')
plt.title(f"Iterations vs Training Loss lr = {lr}")
plt.legend()
plt.show()

# %% [markdown]
# ADAM - Hyperparamter Tuning - B1 and B2

# %%
# NOTE: The following code loops through all beta1 and beta2 values.
#       The code then creates a new cgp and trains with the beta values
#       The results are then saved to the results list

B1 = [0.8 , 0.95, 0.99]
B2 = [0.89, 0.9 , 0.95]

i = 1
results = []
for b1 in B1:
    for b2 in B2:

testAdam = ComputationalGraphPrimer_ReturnLoss_ADAM(num_layers = 3,layers_config = [4,2,1], 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 = False,)
testAdam.parse_multi_layer_expressions()
training_data = testAdam.gen_training_data()
testAdam.initializeADAMStuff_MultiNeuron(beta1=b1, beta2=b2)

# Start time measurement
start_time = time.time()

loss_testAdam = testAdam.run_training_loop_multi_neuron_model(training_data)

# End time measurement
end_time = time.time()
duration = end_time - start_time

# Save metrics results list
finalLoss = loss_testAdam[-1]
minLoss = min(loss_testAdam)

result = f"{i}\t{b1}\t{b2}\t{round(duration, 1)}\t{round(finalLoss, 4)}\t{round(minLoss, 4)}"
results.append(result)
i+=1

# %%
# Results are printed
print(f"Trial\tB1\tB2\tTime\tFinal-L\tMin-L")
for r in results:
print(r)

```