ECE60146 - HW3 Report Alim Karimi

February 6, 2023

Part 1: Description of SGD+ and Adam.

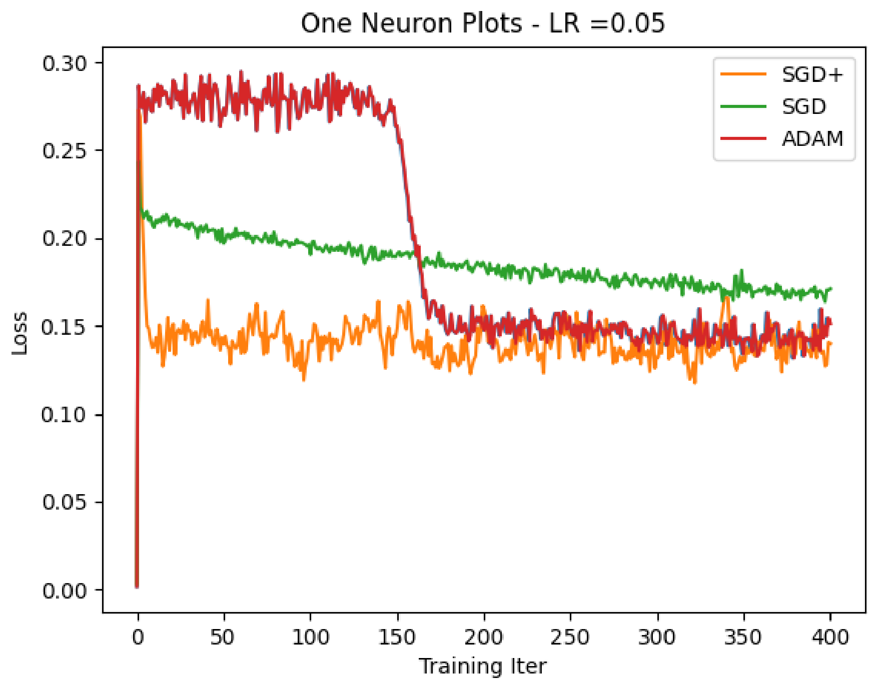
SGD+ helps solve the problem of getting oscillatory behavior near the true minimum by introducing a momentum coefficient. This momentum coefficient, μ, multiplies with the previous step size (v_t), and adds that resulting product to the current gradient (g_{t+1}). The result of that addition is the new step size (i.e, $v_{t+1} = \mu * v_t + g_{t+1}$). This step size is then multiplied by the constant learning rate, so, the set of parameter values is actually dependent on the previous step size! Mathematically, this is represented as $p_{k+1} = p_k - \alpha * v_{t+1}$, where α is a fixed learning rate. Overall, this makes the step sizes larger if the previous step size and current gradient are in the same direction (because instead of multiplying learning rate by just the current gradient, we are adding a fraction of the previous gradient as well). On the other hand, if the previous step size and current gradient point in different directions (i.e, one is positive and the other is negative), then, the next set of parameters values is incremented more slowly than without SGD+.

SGD+ assumes that we should set a fixed learning rate for each parameter. However, it does not make sense to have a fixed learning rate for each parameter when the resulting gradient is sparse. In order to adaptively set a learning rate, AdaGrad aims to divide the sum of all the parameters in that dimension by the learning rate and then multiply that quotient by the parameter (i.e, the next step is $(\alpha / \sum(g_{k,i})^2)* g_{k,i}$. So, if the current gradient is zero, we actually do not update that parameter. However, as the learning process goes on, the sum of that certain parameter increases monotonically and therefore the impact of the adaptive learning rate actually becomes smaller, until we increase the step size by infinitesimally small steps. This becomes counterproductive to learning when there are actually discriminatory features in the data. Therefore, a separate algorithm named RMSprop actually keeps a running average of the gradient, which doesn't increase monotonically. This effectively multiplies the k-th iteration of i-th component of the gradient $g_{k\,i}$ by the learning rate α , as shown here: $[\, \alpha / sqrt(\sum ((g(k,\ i))2\ +\ \varepsilon)\,]\ *\ g(k,\ i)$, where ε is a very small value to prevent division by zero. The Adam optimizer put these two ideas (adaptive momentum due to better convergence to a minimum and adaptive gradient to deal with sparse gradients) into one algorithm. Adam keeps a running average of both of these factors in two "moments". The first moment, $m_{t+1} = \beta_1 * m_t + (1 - \beta_1) * g_{t+1}$, where $\beta_1 = 0.9$ and stays constant. The second moment, $v_{t+1} = \beta_2 * v_t + (1 - \beta_2) * (g_{t+1})^2$. The moments are initialized as $m_0 = 0$ and $v_0 = 0$, which makes resulting next values of the moments quite small at the beginning. Therefore, we adjust the moments to $m\hat{}_k = m_k / (1 - \beta_1^k)$ and $v\hat{}_k = v_k / (1 - \beta_2^k)$, where k is the training iteration of the parameter in question. $v\hat{}_k$ and $m\hat{}_k$ are then both contributing to the next step size by the next parameters, $p_{t+1} = p_t - \alpha * (m\hat{} / sqrt(v\hat{} + \varepsilon))$.

Part 2: Comparative Plots for SGD, SGD+, and Adam

Part 2a: One Neuron Classifier Plots (Figure 1 and 2)

Figure 1 below



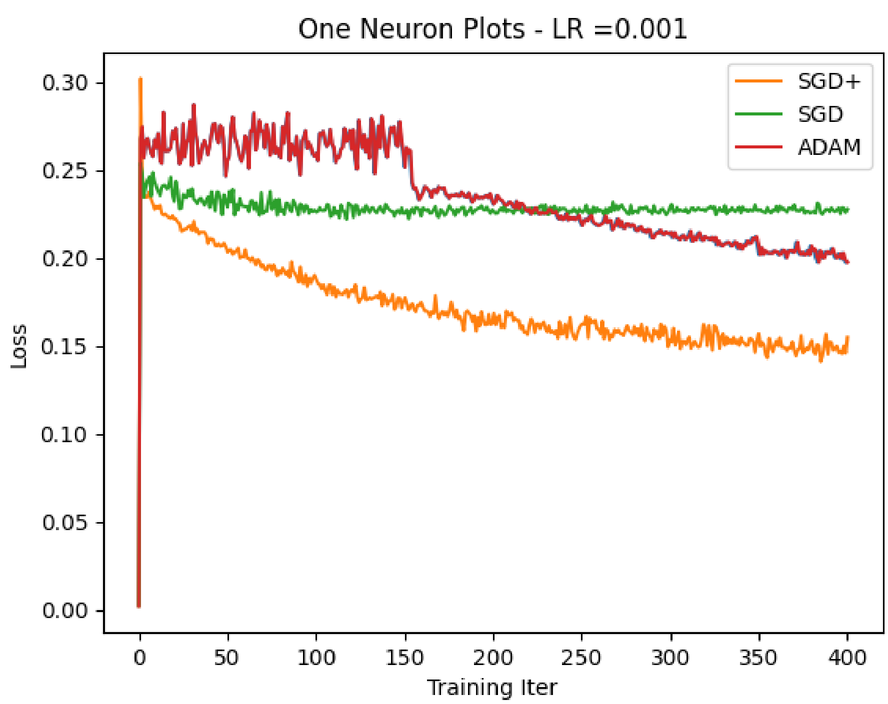
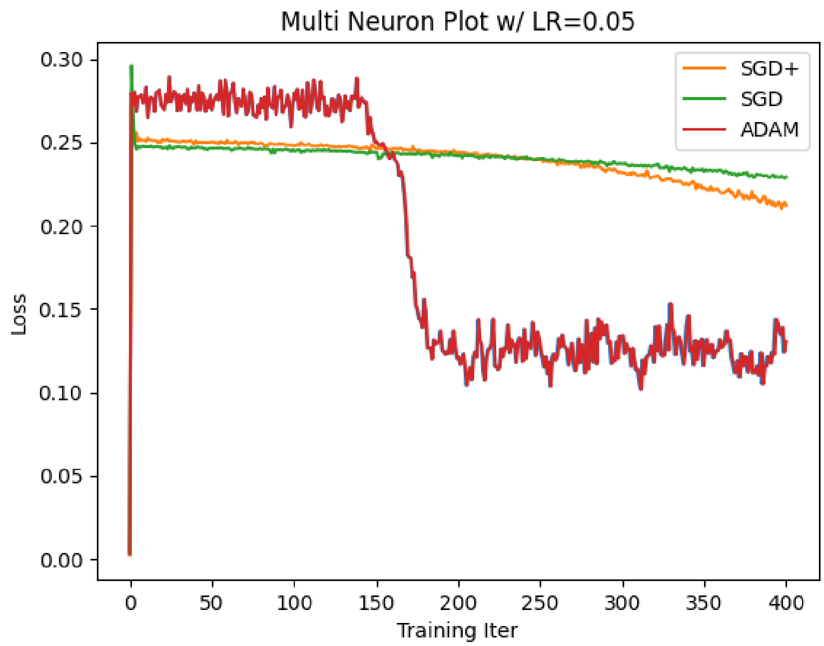


Figure 2 Above

Part 2b: Multi Neuron Classifier Plots
Figure 3 Below



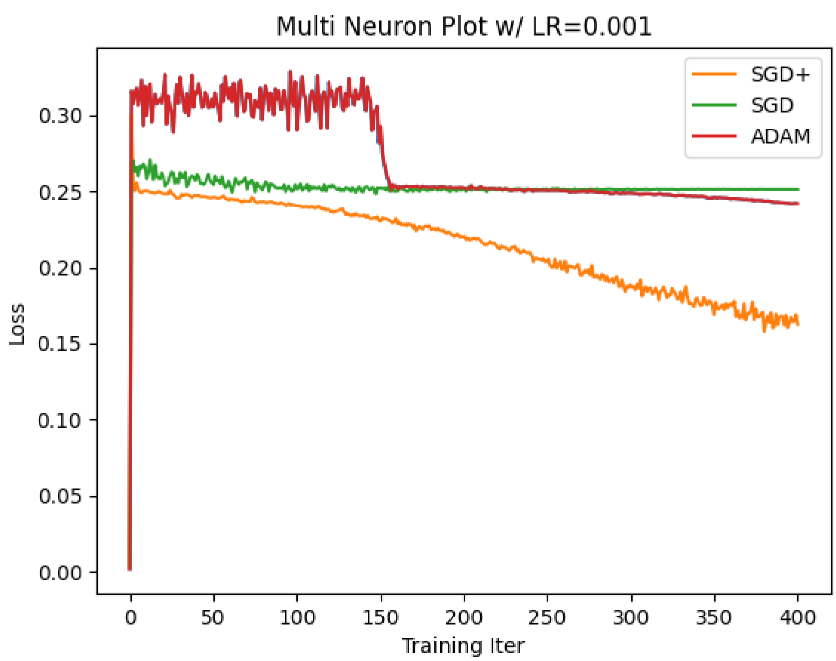


Figure 4 Above

Part 2c: Discussion of findings

Overall, SGD+ and Adam both perform better than SGD in both cases (with learning rate set to 0.001 or 0.05). It was interesting to note that Adam had a significant jump about halfway through the training process. It isn't super clear why Adam doesn't seem to learn anything and then all of a sudden takes large leaps. Adam also performed better compared to SGD+ with a higher learning rate (learning rate set to 0.05 lead to Adam having a loss between 0.10 and 0.15) in both the multi neuron case, but Adam and SGD+ had the same loss after 40,000 iterations in the single neuron case (both losses were around 0.14). SGD+ on the other hand, performed better than Adam when the learning rate was set to 0.001 (had a loss around 0.16 for both the single and multi neuron cases). Lastly, Adam seems to oscillate much more than either SGD or SGD+. SGD and SGD+ seem to decrease monotonically while Adam can have some small oscillations when it is already at a smaller loss.

Part 3: Code

Summary: I created 2 classes (one for Adam, one for SGD+) that inherit ComputationalGraphPrimer and overrode the multi neuron and one neuron backpropagation functions in each of those classes. I also created code that runs each scenario (one neuron for SGD, SGD+, and Adam and multi neuron for SGD, SGD+, Adam) and plots the results. Furthermore, I added empty lists in the training loop functions which were passed to the overridden functions. However, since those are so small, I didn't explicitly add them here. Additionally, one can see the overriding function definitions contain more input parameters than in the original CGP code.

Here is the class definition for the SGD+ class:

```
from ComputationalGraphPrimer import *
from CGP_subclasses_ADAM import *

loss_in_run = []
loss_sgd_plus = []
loss_adam = []
loss_sgd_plus_MN = []
class Subclasses_SGD_plus(ComputationalGraphPrimer):
   def __init__(self):
       super(Subclasses_SGD_plus, self).__init__(  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 = 1e-6,
```

```
            learning_rate = 1e-3,
            #learning_rate = 5 * 1e-2,
            training_iterations = 40000,
            batch_size = 8,
            display_loss_how_often = 100,
            debug = True,
    ) #initialize base class ComputationalGraphPrimer
```

This is the overriding function of the backpropagation for SGD+ in the one neuron case. Note that the parts that were modified are denoted (about halfway through the function).

```
   def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars,
deriv_sigmoid, vt1_list, bias_vt1_list, k):

       momentum = 0.99

       input_vars = self.independent_vars
       input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]
       param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()}
       vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
       vals_for_learnable_params = self.vals_for_learnable_params
       for i,param in enumerate(self.vals_for_learnable_params):

           ## Calculate the next step in the parameter hyperplane
           #            step = self.learning_rate * y_error *
vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid

           ###MODIFCATION TO CODE FOR SGD+ STARTS BELOW
           gradient =   y_error * vals_for_input_vars_dict[param_to_vars_map[param]] *
deriv_sigmoid  #orig step computation.
            ####THIS CODE BELOW INITIALIZES Vt TO 0 IF THERE IS NO PREV ITERATION.
            ### IF THERE IS NO PREV ITER: the step size is just equal to the gradient
Gt+1, which was computed above as step
            ### IF THERE IS A PREV ITERATION. Vt+1 = MOMENTUM * Vt + GRADIENT
(computed as the variable "step" in code already)
           if (len(vt1_list) <4 ):
               mu_vt = 0 #momentum * vt = momentum * 0 = 0
               vt1_list.append(mu_vt + gradient) #append Vt+1 to step list. We will
need to access this as Vt for future computations.
           else:
               vt1 = (momentum * vt1_list[-4]) + gradient
```

```
                vt1_list.append(vt1)
            #step = (momentum * prev_step) + step #here, step is Vt+1, momentum is mu,
step_list[i-1] is Vt, and step is Gt+1
            ## Update the learnable parameters
            self.vals_for_learnable_params[param] =
self.vals_for_learnable_params[param] + (self.learning_rate * vt1_list[-1]) #implement
-momentum * dpk-1
        bias_gradient = y_error * deriv_sigmoid #COMPUTE GRADIENT OF BIAS TERM.
        if (len(bias_vt1_list) == 0): #SAME LOGIC AS VT1 LIST, BUT KEEPING TRACK OF
BIAS GRADIENTS
            mu_vt = 0 #INITIALIZE MU TO 0
            bias_vt1_list.append(mu_vt + bias_gradient)
        else:
            bias_vt1 = (momentum * bias_vt1_list[-1]) + bias_gradient
            bias_vt1_list.append(bias_vt1)
        self.bias = self.bias + self.learning_rate * bias_vt1_list[-1] #UPDATE
PARAMETER FOR BIAS
        ###END MODIFICATION FOR SGD+
```

This is the overriding function of the backpropagation for SGD+ in the multi neuron case. Note that the parts that were modified are denoted.

```
def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels,
Vt1_list, bias_vt1_list):

        momentum = 0.99
        pred_err_backproped_at_layers = {i : [] for i in range(1,self.num_layers-1)}
        pred_err_backproped_at_layers[self.num_layers-1] = [y_error]

        for back_layer_index in reversed(range(1,self.num_layers)):
            input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
            input_vals_avg = [sum(x) for x in zip(*input_vals)]
            input_vals_avg = list(map(operator.truediv, input_vals_avg,
[float(len(class_labels))] * len(class_labels)))
            deriv_sigmoid =  self.gradient_vals_for_layers[back_layer_index]
            deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
            deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
                                                             [float(len(class_labels))]
* len(class_labels)))
            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']
```

```
            layer_params = self.layer_params[back_layer_index]
            ## 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"
            transposed_layer_params = list(zip(*layer_params))          ## creating a
transpose of the link matrix

            backproped_error = [None] * len(vars_in_next_layer_back)
            for k,varr in enumerate(vars_in_next_layer_back):
                for j,var2 in enumerate(vars_in_layer):
                    backproped_error[k] =
sum([self.vals_for_learnable_params[transposed_layer_params[k][i]] *

pred_err_backproped_at_layers[back_layer_index][i]
                                                  for i in range(len(vars_in_layer))])
#                                                   deriv_sigmoid_avg[i] for i in
range(len(vars_in_layer))])
            pred_err_backproped_at_layers[back_layer_index - 1]  =  backproped_error
            input_vars_to_layer = self.layer_vars[back_layer_index-1]




            for j,var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]

                ##  Regarding the parameter update loop that follows, see the Slides 74
through 77 of my Week 3
                ##  lecture slides for how the parameters are updated using the partial
derivatives stored away
                ##  during forward propagation of data. The theory underlying these
calculations is presented
                ##  in Slides 68 through 71.
                for i,param in enumerate(layer_params):
                    #### START CODE FOR MULTI NEURON SGD+
                    #Below, compute the gradient for linear operations chained with the
activiation function
                    gradient_of_loss_for_param = input_vals_avg[i] *
pred_err_backproped_at_layers[back_layer_index][j] * deriv_sigmoid_avg[j]
```

```
                    if len(Vt1_list) < 10: #for the first 10 parameters, Vt = 0, so,
Vt+1 = momentum * Vt + Gt+1 will be just be Gt+1
                        mu_vt = 0 #set mu to 0 for initial values
                        Vt1_list.append(mu_vt + gradient_of_loss_for_param)
                    else:
                        Vt1 = momentum * Vt1_list[-10] + gradient_of_loss_for_param #if
the length of the list is greater than 10
                        #then we are on the second iteration of that parameter and can
look 10 indices back on the list to get
                        #the previous step size
                        Vt1_list.append(Vt1)
                    self.vals_for_learnable_params[param] += self.learning_rate *
Vt1_list[-1]  #update current parameters based on
                    #current gradient and previous step size
                    #just the gradient.
            bias_gradient = sum(pred_err_backproped_at_layers[back_layer_index]) \
                                                                                *
sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)
            if (len(bias_vt1_list) <2 ): #same logic to update bias gradient, except,
there are only 2 bias parameters
                #in the network. Make mu 0 for the first iteration (i.e, the 2
parameters for the first backprop)
                mu_vt = 0
                bias_vt1_list.append(mu_vt + bias_gradient)
            else: #use the last corresponding bias parameter to compute the next step
                bias_vt1_list.append(momentum * bias_vt1_list[-2] + bias_gradient)
            #print("len(bias_vt1_list):", len(bias_vt1_list))
            self.bias[back_layer_index-1] += self.learning_rate * bias_vt1_list[-1]
            #END CODE FOR MULTI NEURON SGD+
```

This is the new class for Adam which inherits from CGP:

```
from ComputationalGraphPrimer import *

class Subclasses_ADAM(ComputationalGraphPrimer):
   def __init__(self):
       super(Subclasses_ADAM, self).__init__(   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 = 1e-6,
            #learning_rate = 1e-3,
            learning_rate = 5 * 1e-2,
            #learning_rate = 1e-2,
            training_iterations = 40000,
            batch_size = 8,
            display_loss_how_often = 100,
            debug = True,
    ) #initialize base class ComputationalGraphPrimer
       self.loss_oneN
       self.loss_multiN
```

Below is the overriding class for the backpropagation to do the one neuron Adam optimization.

```
def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars,
deriv_sigmoid, mt_list, vt_list, mt_bias_list, vt_bias_list, iteration):

       input_vars = self.independent_vars
       input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]
       param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()}
       vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
       vals_for_learnable_params = self.vals_for_learnable_params
       for i,param in enumerate(self.vals_for_learnable_params):
           ## Calculate the next step in the parameter hyperplane
           ### START CODE FOR ONE NEURON ADAM IMPLEMENTATION
           B1 = 0.9
           B2 = 0.99
           grad = y_error * vals_for_input_vars_dict[param_to_vars_map[param]] *
deriv_sigmoid
           if ((len(mt_list) <4 ) and (len(vt_list) < 4)): #FIRST 4 PARAMETERS IN THE
FIRST ITERATION ARE HAVE vt and mt = 0
               mt_list.append((1 - B1) * grad)
               vt_list.append((1 - B2) * grad**2)
               mt_list[i] = mt_list[i] / (1 - B1) #technically, B1 is raised to the
power of the iteration, but in this case iter = 1
               vt_list[i] = vt_list[i] / (1 - B2) #technically, B2 is raised to the
power of the iteration, but in this case iter = 1
           else:
               power = iteration + 1
```

```
                #COMPUTE THE MOMENTS USING THE PREVIOUS MOMENTS
                mt_list.append(B1 * mt_list[-4] + ((1 - B1) * grad))
                vt_list.append(B2 * vt_list[-4] + ((1 - B2) * grad**2))
                mt_list[-1] = mt_list[-1] / (1 - B1**power) #ADJUST MT FOR THE
INITIALIZATION OF MT=0
                vt_list[-1] = vt_list[-1] / (1 - B2**power) #ADJUST VT FOR THE
INITIALIZATION OF VT=0
            ## Update the learnable parameters
            epsilon = 1e-08
            self.vals_for_learnable_params[param] =
self.vals_for_learnable_params[param] + ((self.learning_rate * (mt_list[-1]) /
(np.sqrt(vt_list[-1] + epsilon))))
        bias_gradient = y_error * deriv_sigmoid
        #SAME LOGIC AS ABOVE, BUT FOR BIAS GRADIENTS
        if ((len(mt_bias_list) == 0) and (len(vt_bias_list) == 0)):
            mt_bias_list.append((1 - B1) * bias_gradient)
            mt_bias_list[0] = mt_bias_list[0] / (1 - B1) #technically, B1 is raised to
power of iteration, but, here, B1^1, so it is not explicitly written
            vt_bias_list.append((1 - B2) * bias_gradient**2)
            vt_bias_list[0] = vt_bias_list[0] / (1 - B2)
        else:
            power = iteration + 1
            mt_bias_list.append(B1 * mt_bias_list[-1] + (1 - B1) * bias_gradient)
            vt_bias_list.append(B2 * vt_bias_list[-1] + (1 - B2) * bias_gradient**2)
            mt_bias_list[-1] = mt_bias_list[-1] / (1 - B1**power)
            vt_bias_list[-1] = vt_bias_list[-1] / (1 - B2**power)
        #UPDATE THE BIAS PARAMETER
        self.bias = self.bias + (self.learning_rate * (mt_bias_list[-1] /
np.sqrt(vt_bias_list[-1] + epsilon)))
```

Below is the overriding class for the backpropagation function to do the multi neuron Adam optimization.

```
def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels,
mt_list, vt_list, mt_bias_list, vt_bias_list, iteration):



        pred_err_backproped_at_layers = {i : [] for i in range(1,self.num_layers-1)}
        pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
        for back_layer_index in reversed(range(1,self.num_layers)):
            input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
            input_vals_avg = [sum(x) for x in zip(*input_vals)]
```

```
            input_vals_avg = list(map(operator.truediv, input_vals_avg,
[float(len(class_labels))] * len(class_labels)))
            deriv_sigmoid =  self.gradient_vals_for_layers[back_layer_index]
            deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
            deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
                                                                 [float(len(class_labels))]
* len(class_labels)))
            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']

            layer_params = self.layer_params[back_layer_index]
            ## 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"
            transposed_layer_params = list(zip(*layer_params))         ## creating a
transpose of the link matrix

            backproped_error = [None] * len(vars_in_next_layer_back)
            for k,varr in enumerate(vars_in_next_layer_back):
                for j,var2 in enumerate(vars_in_layer):
                    backproped_error[k] =
sum([self.vals_for_learnable_params[transposed_layer_params[k][i]] *

pred_err_backproped_at_layers[back_layer_index][i]
                                              for i in range(len(vars_in_layer))])
#                                               deriv_sigmoid_avg[i] for i in
range(len(vars_in_layer))])
            pred_err_backproped_at_layers[back_layer_index - 1]  =  backproped_error
            input_vars_to_layer = self.layer_vars[back_layer_index-1]

            for j,var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]
                ##  Regarding the parameter update loop that follows, see the Slides 74
through 77 of my Week 3
                ##  lecture slides for how the parameters are updated using the partial
derivatives stored away
                ##  during forward propagation of data. The theory underlying these
calculations is presented
```

```
                ##  in Slides 68 through 71.
                for i,param in enumerate(layer_params):
                    ###START CODE FOR MULTI NEURON ADAM
                    B1 = 0.9
                    B2 = 0.99
                    epsilon = 1e-08
                    gradient_of_loss_for_param = input_vals_avg[i] *
pred_err_backproped_at_layers[back_layer_index][j] * deriv_sigmoid_avg[j]
                    power = iteration + 1
                    if (len(mt_list) < 10 and len(vt_list) < 10):
                        #initialize vt and mt to 0.
                        mt_list.append((1 - B1) * gradient_of_loss_for_param)
                        vt_list.append((1 - B2) * gradient_of_loss_for_param**2)
                        mt_list[-1] = mt_list[-1] / (1 - B1) #ADJUST MT FOR
INITIALIZING MT0 TO 0
                        vt_list[-1] = vt_list[-1] / (1 - B2) #ADJUST VT FOR
INITIALIZING VT0 TO 0
                    else:
                        #IF WE HAVE A PREVIOUS VALUE OF MT AND VT, USE THE EQUATION
BELOW TO INCORPORATE
                        #PREVIOUS MT AND VT. SINCE WE ARE STORING 10 ALL LINEAR
PARAMETERS NETWORKS IN THIS LIST
                        #WE NEED TO LOOK BACK BY 10 TO GET THE PREVIOUS MT OR VT
                        mt_list.append(B1 * mt_list[-10] + (1 - B1) *
gradient_of_loss_for_param)
                        vt_list.append(B2 * vt_list[-10] + (1 - B2) *
gradient_of_loss_for_param**2)
                        mt_list[-1] = mt_list[-1] / (1 - B1**power)
                        vt_list[-1] = vt_list[-1] / (1 - B2**power)
                    self.vals_for_learnable_params[param] += self.learning_rate *
mt_list[-1] / np.sqrt(vt_list[-1] + epsilon)
            power = iteration + 1
            bias_gradient = sum(pred_err_backproped_at_layers[back_layer_index]) \
                                                                            *
sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)
            #SAME LOGIC AS COMPUTING THE UPDATES TO THE LINEAR WEIGHTS BUT FOR THE BIAS
WEIGHTS
            if ((len(mt_bias_list) < 2) and (len(vt_bias_list) < 2)):
                mt_bias_list.append((1 - B1) * bias_gradient)
                vt_bias_list.append((1 - B2) * bias_gradient**2)
                mt_bias_list[-1] = mt_bias_list[-1] / (1-B1)
                vt_bias_list[-1] = vt_bias_list[-1] / (1 - B2)
```

```
            else:
                mt_bias_list.append((B1 * mt_bias_list[-2]) + (1 - B1) * bias_gradient)
                vt_bias_list.append((B2 * vt_bias_list[-2]) + (1 - B2) *
bias_gradient**2)
                mt_bias_list[-1] = mt_bias_list[-1] / (1 - B1**power)
                vt_bias_list[-1] = vt_bias_list[-1] / (1 - B2**power)
            self.bias[back_layer_index-1] = self.bias[back_layer_index-1] +
self.learning_rate * (mt_bias_list[-1] / np.sqrt(vt_bias_list[-1] + epsilon))
```

Plotting code:

```
# #CODE FOR SGD+ ONE NEURON NTWK
sub = Subclasses_SGD_plus()
sub.parse_expressions()
#sub.display_one_neuron_network()
training_data = sub.gen_training_data()
sub.run_training_loop_one_neuron_model( training_data )
loss_sgd_plus = loss_in_run #copy current loss into another obj so that we can store
losses for normal SGD run in loss
# ###END CODE FOR SGD+ ONE NEURON NTWK

# ###CODE FOR SGD ONE NEURON NETWORK #####
cgp = ComputationalGraphPrimer(
              one_neuron_model = True,
              expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
              output_vars = ['xw'],
              dataset_size = 5000,
              learning_rate = 1e-3,
              #learning_rate = 5 * 1e-2,
              training_iterations = 40000, #originally 40000
              batch_size = 8,
              display_loss_how_often = 100, #orig 100
              debug = True,
     )
cgp.parse_expressions()
#cgp.display_network1()
#cgp.display_network2()
#cgp.display_one_neuron_network()
training_data = cgp.gen_training_data()
cgp.run_training_loop_one_neuron_model( training_data )
```

```
loss_sgd_orig = cgp.loss_oneN
print(cgp.loss_oneN)
# # ### END CODE FOR SGD ONE NEURON NETWORK #####

###CODE FOR ADAM ONE NEURON ####
adam = Subclasses_ADAM()
adam.somefunc(5)
adam.parse_expressions()
#adam.display_one_neuron_network()
training_data = adam.gen_training_data()
adam.run_training_loop_one_neuron_model( training_data )
loss_adam = adam.loss_oneN
### END CODE FOR ADAM ONE NEURON ###


##PLOTTING CODE FOR ONE NEURON ###
x_axis_len = len(loss_sgd_plus)
# x_axis_len = len(loss_adam)
#print(loss_sgd_plus)
#print('sgd_orig:', loss_sgd_orig)
plt.plot(np.linspace(0, x_axis_len, x_axis_len), loss_sgd_plus, label = "SGD+")
plt.plot(np.linspace(0, x_axis_len, x_axis_len), loss_sgd_orig, label = "SGD")
plt.plot(np.linspace(0, x_axis_len, x_axis_len), loss_adam, label = "ADAM")
plt.xlabel("Training Iter")
plt.ylabel("Loss")
plt.legend()
plt.title("One Neuron Plots - LR =" + str(adam.learning_rate))
plt.show()

#CODE FOR SGD+ MULTI NEURON NETWORK
sub = Subclasses_SGD_plus()
sub.parse_multi_layer_expressions()
#sub.display_multi_neuron_network()
training_data = sub.gen_training_data()
sub.run_training_loop_multi_neuron_model( training_data )
loss_sgd_plus_MN = loss_in_run
##END CODE FOR SGD+ MULTI NEURON NETWORK


##CODE FOR SGD MULTI NEURON NETWORK #####
cgp = ComputationalGraphPrimer(
              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 = 1e-6,
              #learning_rate = 1e-3,
              learning_rate = 5 * 1e-2,
              #learning_rate = 1e-2,
              training_iterations = 40000,
              batch_size = 8,
              display_loss_how_often = 100,
              debug = True,
     )
cgp.parse_multi_layer_expressions()
#cgp.display_multi_neuron_network()
training_data = cgp.gen_training_data()
cgp.run_training_loop_multi_neuron_model( training_data )
loss_sgd_orig = cgp.loss_multiN
## END CODE FOR SGD MULTI NEURON NETWORK #####



##CODE FOR ADAM MULTI NEURON ####
adam = Subclasses_ADAM()
adam.parse_multi_layer_expressions()
#adam.display_multi_neuron_network
training_data = adam.gen_training_data()
adam.run_training_loop_multi_neuron_model(training_data)
loss_adam_multiN = adam.loss_multiN
##END CODE FOR ADAM MULTI NEURON


###PLOTTING CODE FOR MULTI NEURON ####
x_axis_len = len(loss_adam_multiN)
plt.plot(np.linspace(0, x_axis_len, x_axis_len), loss_sgd_plus_MN, label = "SGD+")
plt.plot(np.linspace(0, x_axis_len, x_axis_len), loss_sgd_orig, label = "SGD")
plt.plot(np.linspace(0, x_axis_len, x_axis_len), loss_adam_multiN, label = "ADAM")
plt.xlabel("Training Iter")
plt.ylabel("Loss")
```

```
plt.legend()
plt.title("Multi Neuron Plot w/ LR=" + str(adam.learning_rate))
plt.show()
```