ECE 695DL - Deep Learning - HW3 Report

Brad Fitzgerald

February 8, 2022

1 Introduction

Successful training of a neural network involves minimization of a loss function in order to find parameters values that result in better model performance. One basic method for minimizing a loss function is using the gradient descent algorithm, however in its simplest form this method suffers from the potential of getting "trapped" in local minimums of the loss function. Stochastic gradient descent (SGD), in which model parameters are updated based on multiple training samples (i.e. a batch) at a time, can help overcome this issue. Yet, without some form of step size optimization, plain SGD can suffer from slow convergence. In this assignment, we add momentum to a "from-scratch" implementation of SGD in order to improve performance and convergence speed of network training on a single-neuron and on a multi-neuron network.

2 Methodology

The central theory for completing this assignment - the theory of how to implement momentum to improve learning in SGD - was described in the assignment prompt in Section 1.1.7. Specifically, momentum was added to the existing SGD learning framework by implementing Equation 2 from the assignment instructions. The momentum parameter μ was set to be 0.99. This code was developed using Google Colab.

This implementation was completed by first installing the Computational Graph Primer software version 1.0.8 (provided by Prof. Kak at `https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.0.8.html`). The existing `ComputationalGraphPrimer` class was imported and used to create two child classes called `CGP` and `CGP_SGDplus`. The first class, `CGP`, served as a nearly exact copy of the imported class with the simple modification that the class method `run_training_loop_multi_neuron_model` was edited to return the `loss_running_record` variable (so that the performance of the initial training method could be compared with the performance of the updated method). This change was implemented for both of the new created classes.

The second class, `CGP_SGDplus`, was used to add the momentum method to the SGD parameter updates. To do this, new class variables called `prev_step` and `prev_bias_step` were created to keep track of the previous steps taken for each learnable parameter and bias values, respectively. The class methods in which parameter updates occur were adjusted to function as described by the momentum equations (Equation 2) described by the assignment instructions.

3 Implementation and Results

3.1 Single Neuron Model

The code used to adjust the training of the one-neuron-model is shown below (also see script named `one_neuron_classifier_sgd_plus.py`). The `run_training_loop_one_neuron_model` method was overwritten in the child class `CGP_SGDplus` to include the definition of new class variables `prev_step` and `prev_bias_step` to keep track of previous parameter steps (code lines 123-124). The `backprop_and_update_params_one_neuron_model` method was overwritten to implement learning steps which incorporated momentum (code lines 212-217). A comparison of the training loss for the one-neuron model produced by the original SGD implementation with the SGD+ (with momentum) implementation is shown in Figure 1.

```
#!/usr/bin/env python
##  one_neuron_classifier.py

"""
A one-neuron model is characterized by a single expression that you see in the value
supplied for the constructor parameter "expressions".  In the expression supplied, the
names that being with 'x' are the input variables and the names that begin with the
other letters of the alphabet are the learnable parameters.
"""

import random
import numpy
import sys
sys.path.append("/work/BF/Classes/695/HW3/ComputationalGraphPrimer-1.0.8/ComputationalGraphPrimer/")


seed = 0
random.seed(seed)
numpy.random.seed(seed)

###############################################################################

# I note here that the original Computational Graph Primer code used in this
# script was designed by Professor Avi Kak, Purdue University, shared via
# his website at
# https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.0.8.html.

# The methods re-defined below are copies of Prof. Kak's code with minor
# changes made to add momentum for the Deep Learning 2022 HW3 assignment.

###############################################################################

from ComputationalGraphPrimer import *

class CGP(ComputationalGraphPrimer):
  def run_training_loop_one_neuron_model(self, training_data):
        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
        self.bias = random.uniform(0,1)

        class DataLoader:
            """
            The data loader's job is to construct a batch of randomly chosen samples from the
            training data.  But, obviously, it must first associate the class labels 0 and 1 with
            the training data supplied to the constructor of the DataLoader.   NOTE:  The training
            data is generated in the Examples script by calling 'cgp.gen_training_data()' in the
            ****Utility Functions*** section of this file.  That function returns two normally
            distributed set of number with different means and variances.  One is for key value '0'
            and the other for the key value '1'.  The constructor of the DataLoader associated a'
            class label with each sample separately.
            """
```

```
            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]]
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                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_literations = 0.0
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tuples)
            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_literations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_literations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_literations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_literations))
                avg_loss_over_literations = 0.0
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
            data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
            data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                  [float(len(class_labels))] * len(class_labels) ))
            self.backprop_and_update_params_one_neuron_model(y_error_avg, \
                                  data_tuple_avg, deriv_sigmoid_avg)
        plt.figure()
        plt.plot(loss_running_record)
        #plt.show()
        return loss_running_record

class CGP_SGDplus(ComputationalGraphPrimer):
    def run_training_loop_one_neuron_model(self, training_data, mu):
        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
        self.bias = random.uniform(0,1)
        self.mu = mu
        self.prev_step = np.zeros(len(cgp.vals_for_learnable_params))
        self.prev_bias_step = 0

        class DataLoader:
            """
            The data loader's job is to construct a batch of randomly chosen samples from the
            training data.  But, obviously, it must first associate the class labels 0 and 1 with
            the training data supplied to the constructor of the DataLoader.   NOTE:  The training
            data is generated in the Examples script by calling 'cgp.gen_training_data()' in the
            ****Utility Functions*** section of this file.  That function returns two normally
            distributed set of number with different means and variances.  One is for key value '0'
            and the other for the key value '1'.  The constructor of the DataLoader associated a'
            class label with each sample separately.
            """
            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]]
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
            maxval = 0.0
            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]
            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_literations = 0.0
    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples = data[0]
        class_labels = data[1]
        y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tuples)
        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_literations += loss_avg
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_literations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_literations)
            print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_literations))
            #print("Im actually doing it")
            avg_loss_over_literations = 0.0
        y_errors = list(map(operator.sub, class_labels, y_preds))
        y_error_avg = sum(y_errors) / float(len(class_labels))
        deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
        data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
        data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                  [float(len(class_labels))] * len(class_labels) ))
        self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg, deriv_sigmoid_avg)

    plt.figure()
    plt.plot(loss_running_record)
    #plt.show()
    return loss_running_record

  def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid):
    """
    As should be evident from the syntax used in the following call to backprop function,

        self.backprop_and_update_params_one_neuron_model( y_error_avg, data_tuple_avg, deriv_sigmoid_avg)
                                                                    ^^^            ^^^               ^^^
    the values fed to the backprop function for its three arguments are averaged over the training
    samples in the batch.  This in keeping with the spirit of SGD that calls for averaging the
    information retained in the forward propagation over the samples in a batch.

    See Slides 103 through 108 of Week 3 slides for the logic implemented  here.
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    """
    input_vars = self.independent_vars
    vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
    vals_for_learnable_params = self.vals_for_learnable_params
    for i,param in enumerate(self.vals_for_learnable_params):
        ## calculate the next step in the parameter hyperplane
        step = self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid + \
        self.mu*self.prev_step[i]
        self.vals_for_learnable_params[param] += step
        self.prev_step[i] = step
    bias_step = self.learning_rate * y_error * deriv_sigmoid + self.mu*self.prev_bias_step
    self.bias += bias_step    ## the step to take for the bias
    self.prev_bias_step = bias_step


cgp_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 = 1e-3,
#                learning_rate = 5 * 1e-2,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
      )

cgp = CGP(
                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,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )


cgp.parse_expressions()
cgp_plus.parse_expressions()

#cgp.display_network1()
#cgp.display_network2()

training_data = cgp.gen_training_data()

loss_sgd = cgp.run_training_loop_one_neuron_model( training_data )
loss_sgdplus = cgp_plus.run_training_loop_one_neuron_model( training_data , 0.99 )

import matplotlib.pyplot as plt
plt.figure()
plt.plot(loss_sgd , label = 'SGD Loss')
plt.plot(loss_sgdplus , label = 'SGD+ Loss')
plt.legend()
plt.xlabel('Iterations / 100')
plt.ylabel('Training Loss')
#plt.show()
plt.savefig('one_neuron_loss.jpg')
```

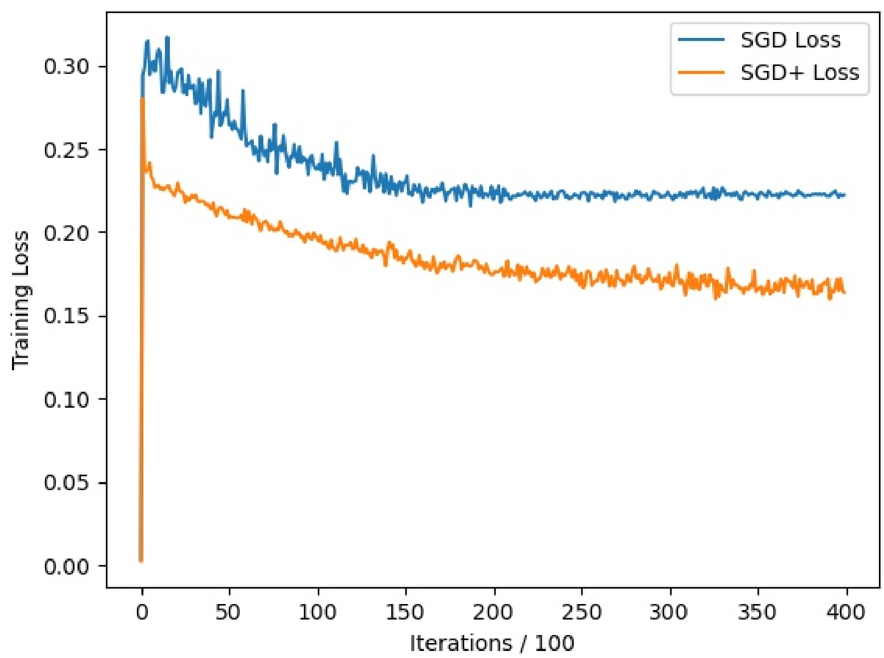

Figure 1: Original SGD training loss (blue) vs. updated SGD+ (with momentum) training loss (orange) for the one-neuron model.

3.2 Multi-Neuron Model

The code used to adjust the training of the multi-neuron-model is shown below (also see script named `multi_neuron_classifier_sgd_plus.py`). The `run_training_loop_multi_neuron_model` method was overwritten in the child class `CGP_SGDplus` to include the definition of new class variables `prev_step` and `prev_bias_step` to keep track of previous parameter steps (code lines 124-131). The `backprop_and_update_params_multi_neuron_model` method was overwritten to implement learning steps which incorporated momentum (code lines 198-204). A comparison of the training loss for the multi-neuron model produced by the original SGD implementation with the SGD+ (with momentum) implementation is shown in Figure 2.

```
#!/usr/bin/env python

##  multi_neuron_classifier.py

import random
import numpy

import sys
sys.path.append("/work/BF/Classes/695/HW3/ComputationalGraphPrimer-1.0.8/ComputationalGraphPrimer/")

seed = 0
random.seed(seed)
numpy.random.seed(seed)

############################################################################

# I note here that the original Computational Graph Primer code used in this
# script was designed by Professor Avi Kak, Purdue University, shared via his
# website at
# https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.0.8.html.

# The methods re-defined below are copies of Prof. Kak's code with minor
# changes made to add momentum for the Deep Learning 2022 HW3 assignment.

############################################################################

from ComputationalGraphPrimer import *

class CGP(ComputationalGraphPrimer):
  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]]
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                batch = [batch_data, batch_labels]
                return batch

        ## We must initialize the learnable parameters
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
        self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]

        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_literations = 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)
            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_literations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_literations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_literations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_literations))
                avg_loss_over_literations = 0.0
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)
        plt.figure()
        plt.plot(loss_running_record)
        #plt.show()
        return loss_running_record

class CGP_SGDplus(ComputationalGraphPrimer):
    def run_training_loop_multi_neuron_model(self, training_data, mu):
        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]]
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                batch = [batch_data, batch_labels]
                return batch

        ## We must initialize the learnable parameters
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
        self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]
        self.mu = mu
        ################################################################################################
        self.prev_bias_step = np.zeros(self.num_layers)

        self.prev_step = {}
        ind,val = enumerate(cgp.layer_params)
        for a in ind:
          for b in range(len(cgp.layer_params[val[a]])):
            for c in range(len(cgp.layer_params[val[a]][b])):
              self.prev_step[cgp.layer_params[val[a]][b][c]] = 0
        ################################################################################################

        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_literations = 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)
            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_literations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_literations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_literations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_literations))
                avg_loss_over_literations = 0.0
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)
        plt.figure()
        plt.plot(loss_running_record)
        #plt.show()
        return loss_running_record
```

```
    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels):
        """

        """
        # backproped prediction error:
        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']]}


            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]
                for i,param in enumerate(layer_params):
                    gradient_of_loss_for_param = input_vals_avg[i] * \
                            pred_err_backproped_at_layers[back_layer_index][j]
                    step = self.learning_rate * gradient_of_loss_for_param * deriv_sigmoid_avg[j] + \
                            self.mu*self.prev_step[param]
                    self.vals_for_learnable_params[param] += step
                    self.prev_step[param] = step

            bias_step = self.learning_rate * sum(pred_err_backproped_at_layers[back_layer_index]) * \
                    sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg) + \
                    self.mu*self.prev_bias_step[back_layer_index-1]
            self.bias[back_layer_index-1] += bias_step
            self.prev_bias_step[back_layer_index-1] = bias_step


cgp = CGP(
               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-3,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

cgp_plus = 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 = 1e-3,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

cgp.parse_multi_layer_expressions()
cgp_plus.parse_multi_layer_expressions()


#cgp.display_network1()
#cgp.display_network2()

training_data = cgp.gen_training_data()

loss_sgd = cgp.run_training_loop_multi_neuron_model( training_data )
```

```
loss_sgd_plus = cgp_plus.run_training_loop_multi_neuron_model( training_data , 0.99 )


plt.figure()
plt.plot(loss_sgd , label = 'SGD Loss')
plt.plot(loss_sgd_plus , label = 'SGD+ Loss')
plt.legend()
plt.xlabel('Iterations / 100')
plt.ylabel('Training Loss')
#plt.show()
plt.savefig('multi_neuron_loss.jpg')
```

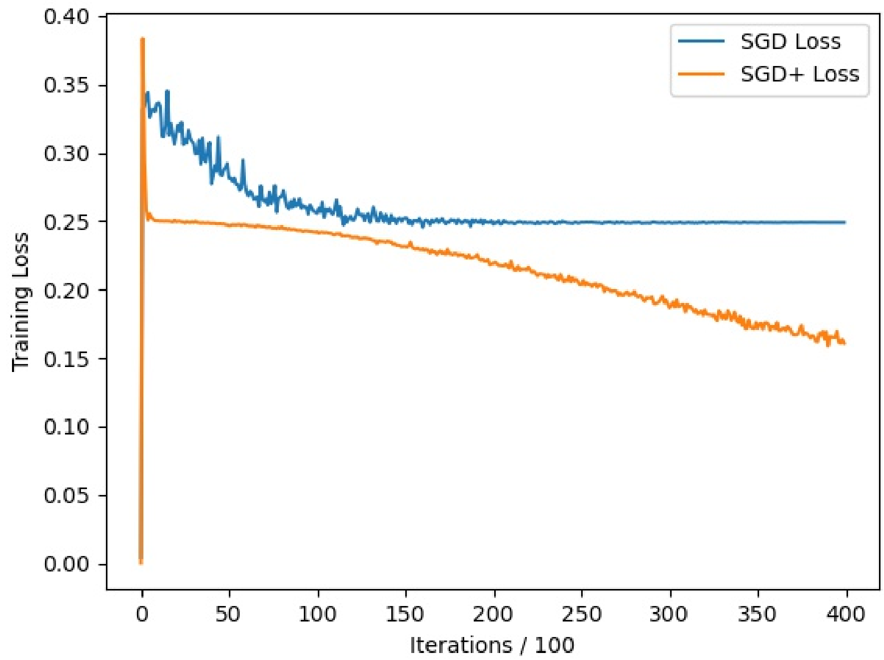

Figure 2: Original SGD training loss (blue) vs. updated SGD+ (with momentum) training loss (orange) for multi-neuron model.

4 Lessons Learned

One of the major lessons I learned in this assignment was the usefulness of the class inheritance structure in Python. As a novice in Python I was tempted to copy and paste the entire Computational Graph Primer class definition in my new script, but later realized that I could adjust the code in a much more efficient way by creating a child class and only overwriting the necessary methods. In addition, I learned a bit about momentum from getting to experiment with different values for the momentum parameter μ. I tested valued ranging from 0 to 1 and found that a value very close to, but still less than, 1 provided the optimal performance, while setting the value to exactly 1 caused erratic behavior in the training loss. Finally, I struggled a bit with understanding how to create a good variable to store the previous parameter update steps in the multi-neuron model, due to the fact that the parameters and parameter values were not stored in "arrays" like I would be previously used to. This provided a good opportunity to better understand ways of structuring lists and dictionaries in Python.

5 Suggested Enhancements

My only minor suggestion is that newer versions of the "ComputationalGraphPrimer" code might benefit from already including a line of code at the end of the "run_training_loop..." class methods which returns the "loss_running_record" variable, so that we don't have to modify the original code just to be able to store the training loss for the default implementation. Otherwise, I thought this assignment was very educational and appreciated the opportunity to learn about the basic workings of neural network training.