SPRING 2023 ECE 60146 – Homework 3

Sahithi Kodali – 34789866
kodali1@purdue.edu

SGD+ Optimization:

SGD+ optimization works by incorporating momentum in the Stochastic Gradient Descent (SGD) algorithm. When we use momentum, it means that the optimization works by taking a step in the current iteration based on the current gradient value and the remembered step size of the previous iteration. The step sizes are calculated separately for each learnable parameters and this makes the optimization much smoother to incorporate previous iteration values. The equations used in SGD+ optimization are as below.

$$s_{t+1} = \mu * s_t + g_{t+1}$$

$$p_{t+1} = p_t - \alpha * s_{t+1}$$

Here, the first equation shows the increase of the step size (s) using the previous step size and the current gradient value (g) and (μ) is the scalar momentum initialized in the range [0,1]. The second equation shows the parameter (p) update using the learning rate (α) and step size updated and the (t) indicates the iteration.

Adam Optimization:

Adam optimization can be said as an upgrade to the SGD+ algorithm that incorporates first and second moments of gradients to update the parameters. It tracked the changing averages of the first and second moments of gradients using the decay rates β_1 and β_2. To overcome the zero-initialization cost during the moments initialization, these values are bias-corrected before updating the learnable parameters. The equations used in Adam optimization are as below.

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

$$v_{t+1} = \beta_1 * v_t + (1- \beta_2) * (g_{t+1})^2$$

$$m_{t+1}` = m_{t+1} / (1- \beta_1^t)$$

$$v_{t+1}` = v_{t+1} / (1- \beta_2^t)$$

$$pt_{+1} = p_t - \alpha * m_{t+1}`/\sqrt{v_{t+1}` + \in}$$

Here, the first moment i.e., mean is (m) and the second moment i.e., variance is (v) and (t) indicates iteration. The decay rates β_1 and β_2 are initialized to 0.9 and 0.99 respectively. The parameters (p) are updated using the learning rate (α) and the corrected values (m` and v`), along with an epsilon (∈) which is very close to zero is added in the denominator for numerical stability.

Results:

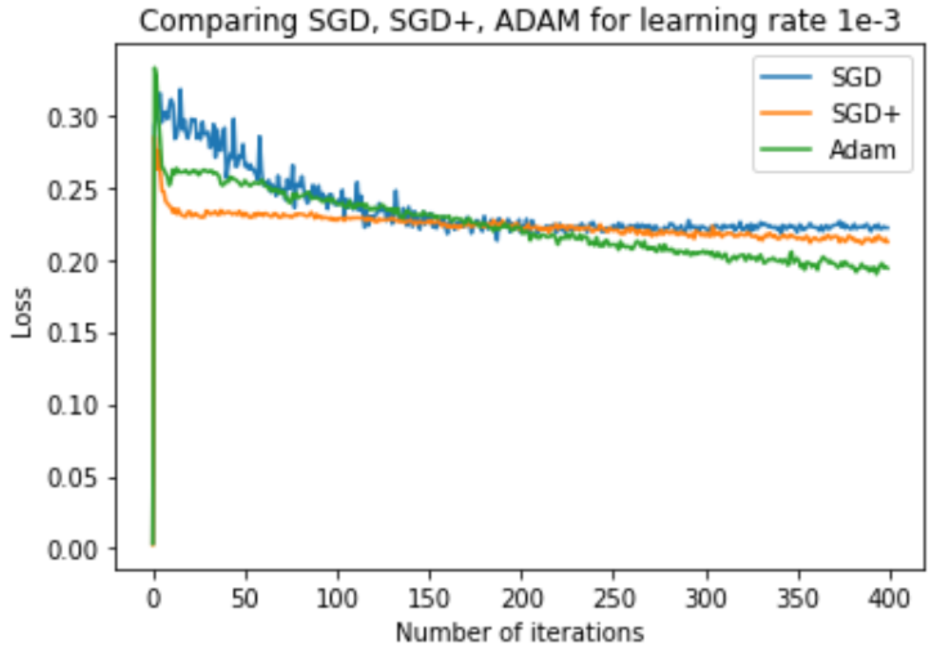

Figure 1: Plot comparing training loss vs iteration for SGD, SGD+, Adam optimizers with learning rate 1e-3 for One_neuron_classifier

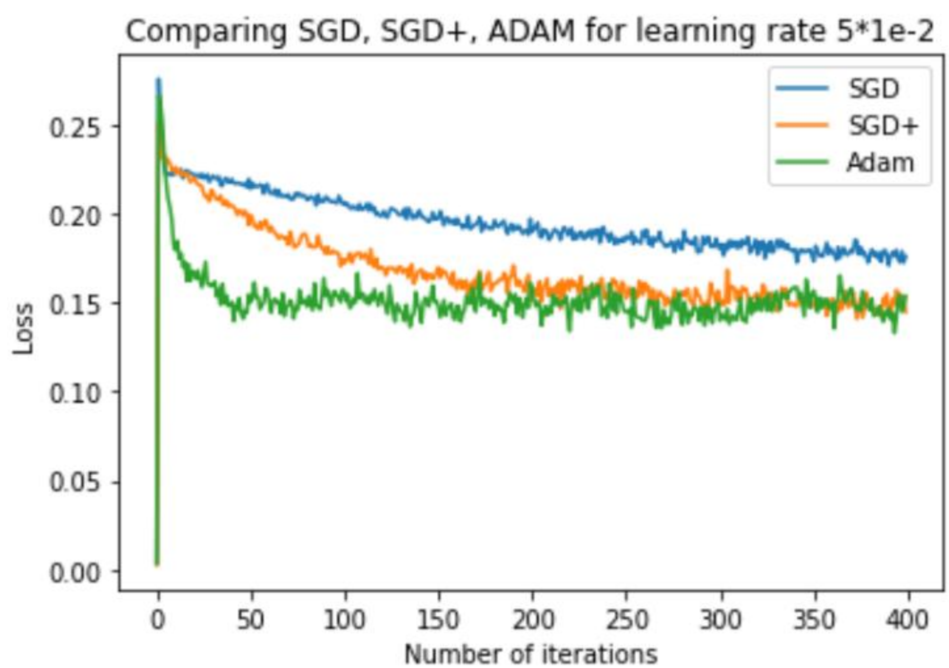

Figure 2: Plot comparing training loss vs iteration for SGD, SGD+, Adam optimizers with learning rate 5*1e-2 for One_neuron_classifier

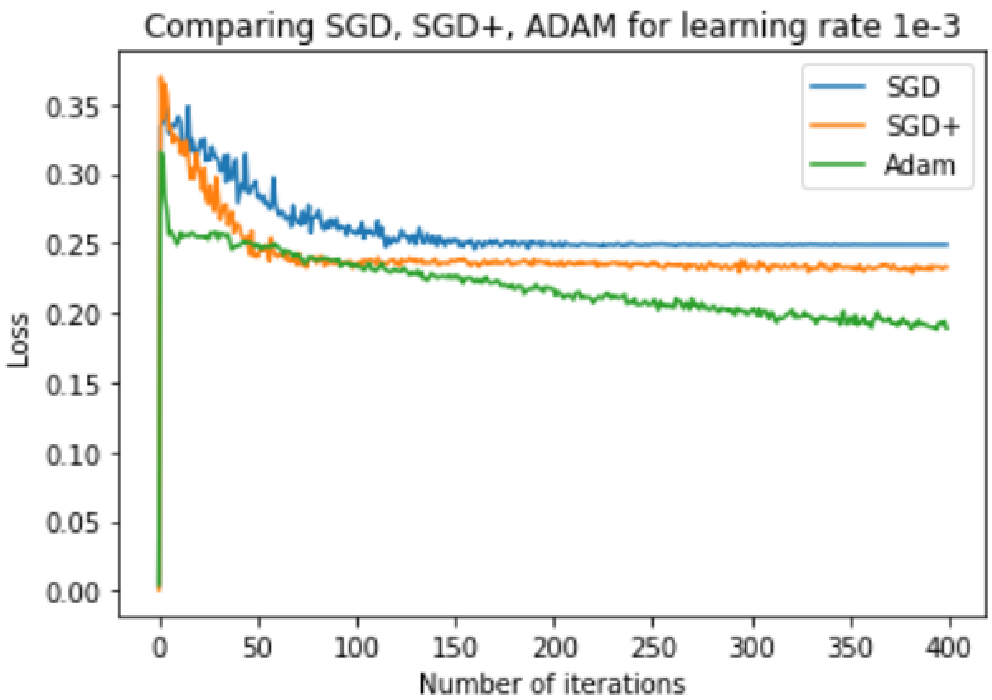

Figure 3: Plot comparing training loss vs iteration for SGD, SGD+, Adam optimizers with learning rate 1e-3 for Multi_neuron_classifier

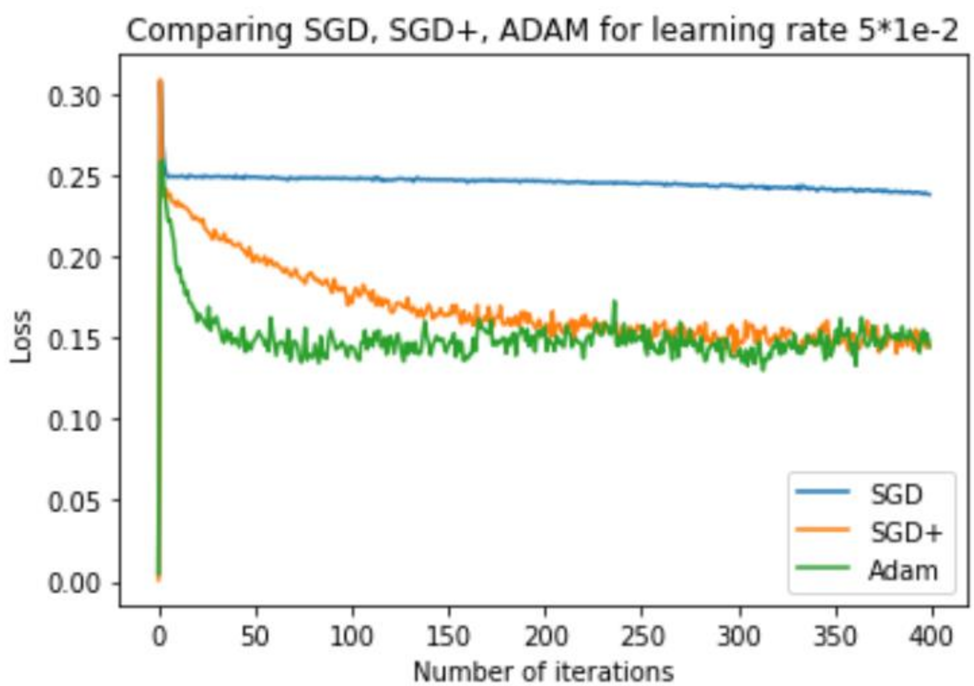

Figure 4: Plot comparing training loss vs iteration for SGD, SGD+, Adam optimizers with learning rate 5*1e-2 for Multi_neuron_classifier

Observations:

The Figures 1 and 2 showcase the results for one-neuron classifier with learning rate 1e-3 (0.001) and 5*1e-2 (0.05) respectively. We can observe in figure-1 with smaller learning rate that the algorithm took longer to reach the minimal loss for all the optimizers by observing the curve that is decreasing with the iterations. However, in the figure-2 with higher learning rate the minimum loss is attained quickly but we can observe the curve to be very sensitive to changes. Adding to that the minimal loss of about ~ 0.15 is also obtained by using larger learning rate and the figure 1 with smaller learning rate stopped at about ~ 0.20 showing the effect of learning rate on the loss. In spite of good learning the choice of learning rate needs to be decided based on the learning and model requirements. Finally, we can say that the Adam optimizer performed better among the three optimizers, with SGD+ taking the next best place. This can be expected considering the algorithm behind Adam optimizer as explained above in the report.

The Figures 3 and 4 showcase the results for multi-neuron classifier with learning rate 1e-3 (0.001) and 5*1e-2 (0.05) respectively. We can observe in figure-1 with smaller learning rate that the algorithm took longer to reach the minimal loss for all the optimizers by observing the curve that is decreasing with the iterations. However, in the figure-2 with higher learning rate the minimum loss is attained quickly but we can observe the curve to be very sensitive to changes. Adding to that the minimal loss of about ~ 0.14 is also obtained by using larger learning rate and the figure 1 with smaller learning rate stopped at about ~ 0.18 showing the effect of learning rate on the loss. We can observe the above explained behavior to be similar to the one neuron model, however the multi-neuron model has slightly higher hand in reducing the loss. Finally, we can say that the Adam optimizer performed better among the three optimizers, with SGD+ taking the next best place which is again similar to the behavior in single neuron classifier.

The above experiments are performed by setting momentum = 0.9 and epsilon = 1e-3. Finally, we can conclude that Adam optimizer performed very well in both models compared to other optimizers, with multi-neuron model performing slightly better than the one-neuron model using all the optimizers. The learning rates also showed their behavior as expected in the plots.

Source Code:

The lines related to plotting the loss vs iterations are commented in ComputationalGraphPrimer.py file and the loss is returned to plot all the graphs together. Below are the codes for One-neuron and Multi-neuron classifier along with the subclasses of SGD+ and Adam inherited from SGD.

```python
##Subclass for SGD+ optimization
class SGDPlus_CGP(ComputationalGraphPrimer):
    def __init__(self, momentum = 0.5, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.momentum = momentum
        self.step_size = {}
```

```
    def backprop_and_update_params_one_neuron_model(self, y_error, 
vals_for_input_vars, deriv_sigmoid):
        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 using momentum
            if param not in self.step_size:
                self.step_size[param] = 0
            mu_vt = self.momentum * self.step_size[param]
            step = mu_vt + (self.learning_rate * y_error * 
vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoid)
            self.step_size[param] = step

            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += step

        ## Update the bias
        if "bias" not in self.step_size:
            self.step_size["bias"] = 0
        mu_vt_bias = self.momentum * self.step_size["bias"]
        step_bias = mu_vt_bias + self.learning_rate * y_error * deriv_sigmoid
        self.step_size["bias"] = step_bias
        self.bias += step_bias

#Subclass for Adam optimization
class Adam_CGP(ComputationalGraphPrimer):
    def __init__(self, beta1 = 0.9, beta2 = 0.99, epsilon = 1e-8,  *args, 
**kwargs):
        super().__init__(*args, **kwargs)
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        self.k = 0
        self.m = {}
        self.v = {}
```

```
    def backprop_and_update_params_one_neuron_model(self, y_error,
vals_for_input_vars, deriv_sigmoid):
        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

        #increase for each iteration
        self.k += 1
        for i,param in enumerate(self.vals_for_learnable_params):

            ## Calculate the next step in the parameter hyperplane using momentum
            if param not in self.m:
                self.m[param] = 0
            if param not in self.v:
                self.v[param] = 0
            grad = y_error * vals_for_input_vars_dict[param_to_vars_map[param]] *
deriv_sigmoid

            #update first and second moments
            self.m[param] = self.beta1 * self.m[param] + (1 - self.beta1) * grad
            self.v[param] = self.beta2 * self.v[param] + (1 - self.beta2) *
grad**2

            #get corrected moments
            m_corr = self.m[param] / (1 - self.beta1**self.k)
            v_corr = self.v[param] / (1 - self.beta2**self.k)

            ## Update the learnable parameters
            step = self.learning_rate * m_corr / (numpy.sqrt(v_corr) +
self.epsilon)
            self.vals_for_learnable_params[param] += step

        #update bias
        if "bias" not in self.m:
            self.m["bias"] = 0
        if "bias" not in self.v:
            self.v["bias"] = 0
        grad_bias = y_error * deriv_sigmoid

        self.m["bias"] = self.beta1 * self.m["bias"] + (1 - self.beta1) *
grad_bias
```

```
        self.v["bias"] = self.beta2 * self.v["bias"] + (1 - self.beta2) *
grad_bias**2

        m_corr = self.m["bias"] / (1 - self.beta1**self.k)
        v_corr = self.v["bias"] / (1 - self.beta2**self.k)

        step_bias = self.learning_rate * m_corr / (numpy.sqrt(v_corr) +
self.epsilon)
        self.bias += step_bias


#!/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 matplotlib.pyplot as plt

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

from ComputationalGraphPrimer import *

###SGD
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,
               batch_size = 8,
```

```
                display_loss_how_often = 100,
                debug = True,
      )

cgp.parse_expressions()
cgp.display_one_neuron_network()
training_data = cgp.gen_training_data()
cgp_loss = cgp.run_training_loop_one_neuron_model( training_data )

###SGD+
cgpP = SGDPlus_CGP(
                momentum = 0.7,
                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,
      )

cgpP.parse_expressions()
cgpP.display_one_neuron_network()
training_data = cgpP.gen_training_data()
cgpP_loss = cgpP.run_training_loop_one_neuron_model( training_data )

###Adam
cgpA = Adam_CGP(
                beta1 = 0.9,
                beta2 = 0.99,
                epsilon = 1e-7,
                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,
      )
```

```
cgpA.parse_expressions()
cgpA.display_one_neuron_network()
training_data = cgpA.gen_training_data()
cgpA_loss = cgpA.run_training_loop_one_neuron_model( training_data )

###plot all graphs

plt.figure()
plt.plot(cgp_loss, label="SGD")
plt.plot(cgpP_loss, label="SGD+")
plt.plot(cgpA_loss, label="Adam")
plt.legend()
plt.xlabel('Number of iterations')
plt.ylabel('Loss')
plt.title("Comparing SGD, SGD+, ADAM for learning rate 5*1e-2")
plt.show()


#!/usr/bin/env python

##  multi_neuron_classifier.py

"""
The main point of this script is to demonstrate saving the partial derivatives
during the
forward propagation of data through a neural network and using that information
for
backpropagating the loss and for updating the values for the learnable
parameters.  The
script uses the following 4-2-1 network layout, with 4 nodes in the input layer,
2 in
the hidden layer and 1 in the output layer as shown below:


                               input

                                 x                                            x
= node

                                 x         x|                                 |
= sigmoid activation
                                                     x|
                                 x         x|
```

```
                                 x

                             layer_0     layer_1     layer_2


To explain what information is stored during the forward pass and how that
information is used during the backprop step, see the comment blocks associated
with
the functions

         forward_prop_multi_neuron_model()
and
         backprop_and_update_params_multi_neuron_model()

Both of these functions are called by the training function:

         run_training_loop_multi_neuron_model()

"""

import random
import numpy
import matplotlib.pyplot as plt

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

from ComputationalGraphPrimer import *

###SGD
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-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.display_multi_neuron_network()
training_data = cgp.gen_training_data()
cgp_loss = cgp.run_training_loop_multi_neuron_model( training_data )

###SGD+
cgpP = SGDPlus_CGP(
                momentum = 0.7,
                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,
       )

cgpP.parse_expressions()
cgpP.display_one_neuron_network()
training_data = cgpP.gen_training_data()
cgpP_loss = cgpP.run_training_loop_one_neuron_model( training_data )

###Adam
cgpA = Adam_CGP(
                beta1 = 0.9,
                beta2 = 0.99,
                epsilon = 1e-7,
                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,
       )
```

```
cgpA.parse_expressions()
cgpA.display_one_neuron_network()
training_data = cgpA.gen_training_data()
cgpA_loss = cgpA.run_training_loop_one_neuron_model( training_data )

###plot all graphs

plt.figure()
plt.plot(cgp_loss, label="SGD")
plt.plot(cgpP_loss, label="SGD+")
plt.plot(cgpA_loss, label="Adam")
plt.legend()
plt.xlabel('Number of iterations')
plt.ylabel('Loss')
plt.title("Comparing SGD, SGD+, ADAM for learning rate 5*1e-2")
plt.show()
```

****