BME 646/ ECE695DL: Homework 3

Shaswata Roy

7 February 2022

1 Introduction

This assignment helps understand the use of various optimizers while performing stochastic gradient descent while updating the parameters.

2 Methodology

Computational graph primer was installed and after glancing through the code changes were made to optimize the stochastic gradient descent.

3 Implementation and Results

one_neuron_classifier_sgd_plus.py

```
import random
import numpy as np
import matplotlib.pyplot as plt
import operator
seed = 0
random.seed(seed)
np.random.seed(seed)
from ComputationalGraphPrimer import *


class ComputationalGraphPrimerSGDPlus(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        if 'mu' in kwargs:
            self.mu = kwargs.pop('mu')
        else:
            self.mu = 0
        super().__init__(*args, **kwargs)
        self.prev_step=[]
```

```
def backprop_and_update_params_one_neuron_model(self, y_error,
vals_for_input_vars, deriv_sigmoid):
    input_vars = self.independent_vars
    vals_for_input_vars_dict = dict(zip(input_vars,list(vals_for_input_vars)))
    for i, param in enumerate(self.vals_for_learnable_params):
        step = self.mu*self.prev_step[i] + self.learning_rate * y_error
        *vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
        self.vals_for_learnable_params[param] += step
        self.prev_step[i] = step
    self.step_size = self.mu*self.step_size
    +self.learning_rate*y_error*deriv_sigmoid
    self.bias += self.step_size

def reset_step(self):
    for i in range(len(self.learnable_params)):
        if len(self.prev_step) < len(self.vals_for_learnable_params):
            self.prev_step.append(0)
        else:
            self.prev_step[i] = 0

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)
    self.step_size = 0

    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

    data_loader = DataLoader(training_data, batch_size=self.batch_size)
    loss_running_record = []
    i = 0
    avg_loss_over_literations = 0.0
    self.reset_step()
    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)

    return loss_running_record

if __name__ == '__main__':
    cgp = ComputationalGraphPrimerSGDPlus(
                   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()
    training_data = cgp.gen_training_data()

    loss_sgd = cgp.run_training_loop_one_neuron_model( training_data )

    cgp_sgd_plus = ComputationalGraphPrimerSGDPlus(
                  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,
                  mu = 0.99
         )

    cgp_sgd_plus.parse_expressions()
    training_data = cgp_sgd_plus.gen_training_data()

    loss_sgd_plus = cgp_sgd_plus.run_training_loop_one_neuron_model( training_data )

    plt.figure()
    plt.plot(loss_sgd, label = "SGD Training Loss")
    plt.plot(loss_sgd_plus, label = "SGD+ Training Loss")
    plt.xlabel('iterations')
    plt.ylabel('loss')
    plt.title('SGD+ vs SGD Loss One Neuron')
    plt.legend()
    plt.show()
```

multi_neuron_classifier_sgd_plus.py

```
import random
import numpy as np
import matplotlib.pyplot as plt
import operator
seed = 0
```

```
random.seed(seed)
np.random.seed(seed)
from ComputationalGraphPrimer import *


class ComputationalGraphPrimerSGDPlus(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        if 'mu' in kwargs:
            self.mu = kwargs.pop('mu')
        super().__init__(*args, **kwargs)
        self.prev_step=[]

    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 l
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## a list l

            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',
            ## "layer_params[idx]" is a list of lists for the link weights in layer whose o
            transposed_layer_params = list(zip(*layer_params))         ## creating a transp

            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
            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.mu*self.prev_step[i]+self.learning_rate *
                    gradient_of_loss_for_param * deriv_sigmoid_avg[j]
                    self.vals_for_learnable_params[param] += step
                    self.prev_step[i] = step
            self.step_size = self.mu*self.step_size+self.learning_rate * sum(pred_err_backpr
            sum(deriv_sigmoid_avg) / len(deriv_sigmoid_avg)
            self.bias[back_layer_index - 1] += self.step_size

    def reset_step(self):
        for i in range(len(self.learnable_params)):
            if len(self.prev_step) < len(self.vals_for_learnable_params):
                self.prev_step.append(0)
            else:
                self.prev_step[i] = 0

    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) for _ in range(self.num_layers-1)]
        self.step_size = 0

        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

        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_literations = 0.0
        self.reset_step()

        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)

        return loss_running_record

cgp = ComputationalGraphPrimerSGDPlus(
               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,
                  mu = 0
         )

if __name__ =='__main__':
    cgp.parse_multi_layer_expressions()

    training_data = cgp.gen_training_data()

    loss_sgd = cgp.run_training_loop_one_neuron_model( training_data )

    cgp_sgd_plus = ComputationalGraphPrimerSGDPlus(
                   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,
                   mu = 0.99
         )

    cgp_sgd_plus.parse_multi_layer_expressions()

    training_data = cgp_sgd_plus.gen_training_data()

    loss_sgd_plus = cgp_sgd_plus.run_training_loop_one_neuron_model( training_data )

    plt.figure()
    plt.plot(loss_sgd, label = "SGD Training Loss")
    plt.plot(loss_sgd_plus, label = "SGD+ Training Loss")
    plt.xlabel('iterations')
    plt.ylabel('loss')
    plt.title('SGD+ vs SGD Loss Multi Neuron')
    plt.legend()
```

```python
plt.show()
```

3.1 Outputs

Figure 1: One Neuron SGD without Optimization with learning rate:10^{-3},batch size:8.

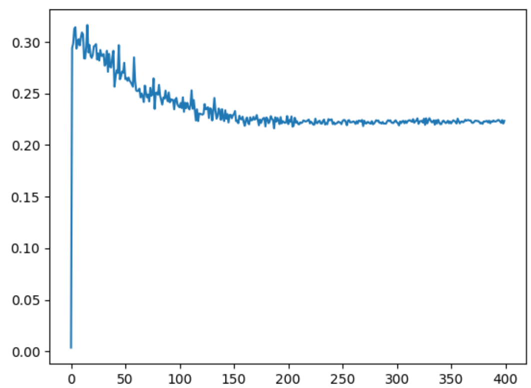


Figure 2: Multi Neuron SGD verified with Torch and learning rate:10^{-6},batch size:8

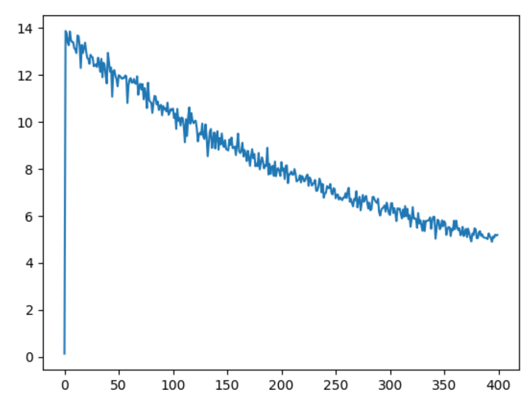


Figure 3: One Neuron SGD verified with Torch and learning rate:10^{-3},batch size:8

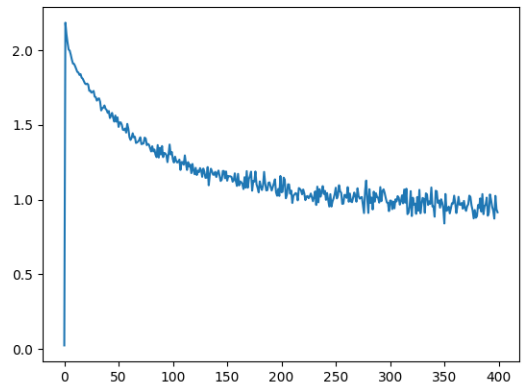


Figure 4: One Neuron SGD comparison with and without optimization and learning rate:10^{-3},batch size:8,$\mu = 0.99$

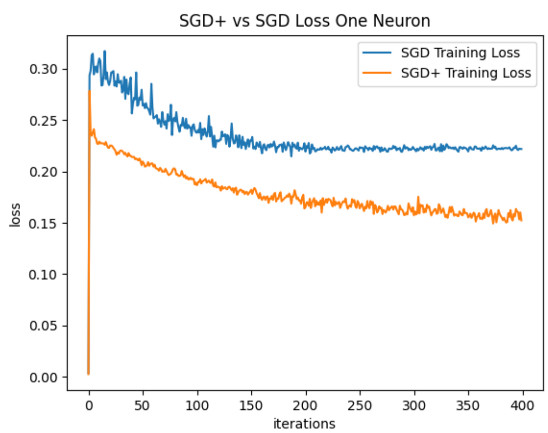
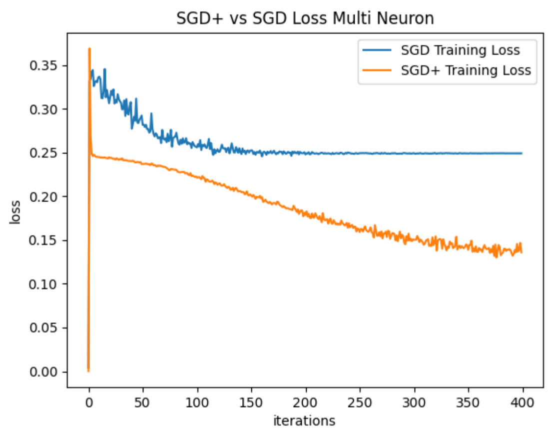


Figure 5: Multi Neuron SGD comparison with and without optimization and learning rate:10^{-3},batch size:8,$\mu = 0.99$



Hence optimization using momentum has clearly lead to lower loss and better accuracy.

4 Lessons Learned

- Optimizing SGD using momentum
- Most of the times momentum will work well when slightly above 0.9
- The code to model neural networks from scratch

5 Suggested Enhancements

A lot of the code is still a black box.