BME 646/ ECE695DL: Homework 3

Varun Aggarwal

February 8, 2022

1 Introduction

This project was aimed at analyzing the *ComputationalGraphPrimer* 1.0.8 library and modifying it to incorporate Stochastic Gradient Descent with momentum.

2 Methodology

The seven tasks in this assignment were tackled as follows:

Task 1

Version 1.0.8 of *ComputationalGraphPrimer* was downloaded from this link. Additionally, I went through the documentation of library to understand its structure.

Task 2

The following files were executed using python. Additionally, the output graphs were saved to disk using *matplotlib*.

- one_neuron_classifier.py (Figure 1 Left)
- multi_neuron_classifier.py (Figure 1 Right)

Task 3

The output of each of the python files Task 2 was a graph of Loss vs Iteration. These graphs are shown in Figure 2. The takeaway from the graphs was that the network was able to train.

Task 4, 5 & 6

In these three task, *torch_nn* implementation was used for training both the one neuron and multi neuron model. The code for *verify_with_torchnn.py* is given in section 3. The output for these tasks indicate that implementation for training one and multi neuron model is correct in *ComputationalGraphPrimer*. Additionally, it was noted that *torch_nn* was much faster although the loss was higher compared to *ComputationalGraphPrimer* implementation. Hyper-parameter tuning might lead to lower loss values with *torch_nn* although it was not required to be done in either of the tasks hence was skipped. Another observation was that *torch_nn* implementation plateaued much faster (Figure 3.

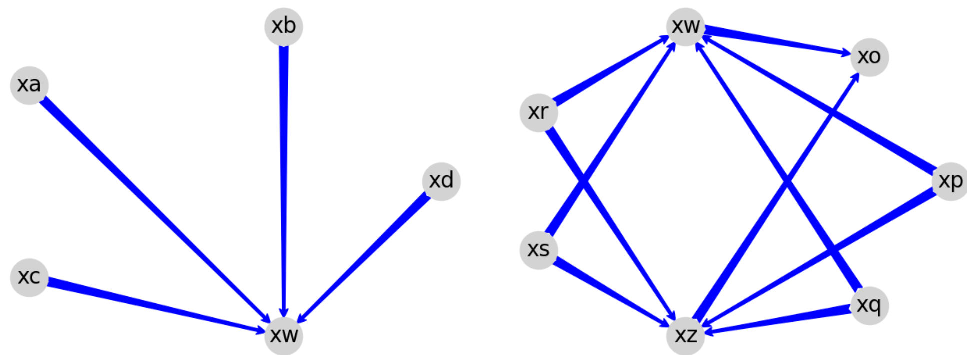

Figure 1: Left: Network graph for single neuron model, Right: Network graph for multi neuron model

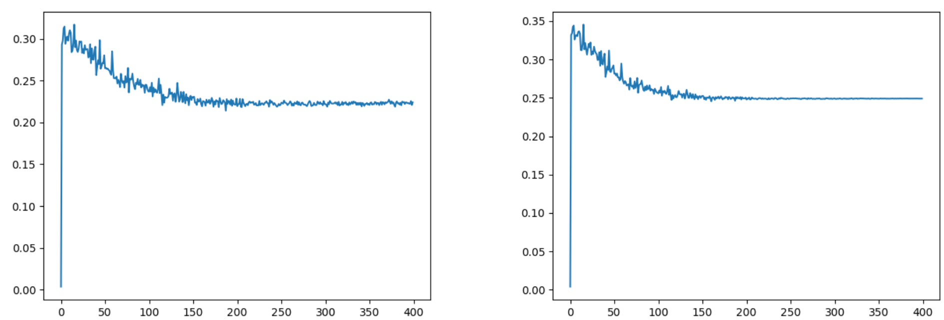

Figure 2: Left: Loss vs Iteration for single neuron model, Right: Loss vs Iteration for multi neuron model

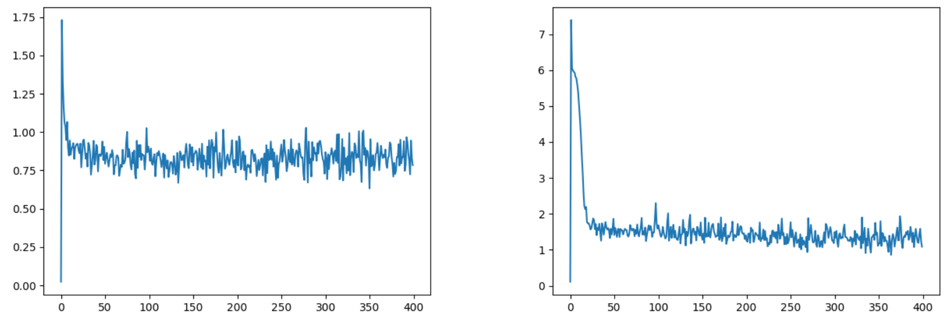

Figure 3: Left: Loss vs Iteration for single neuron model, Right: Loss vs Iteration for multi neuron model

Task 7

This was a central task of the homework that required me to implement a SGD optimizer with momentum. The implementation was done using python. Complete code is given in section 3. Overall, the steps were as follows:

Step1: create a new class *cgpSuperCharged*

Step2: Inherit *ComputationalGraphPrimer* class

Step3: declare instance variable *mu* in *cgpSuperCharged*'s. *mu* is used to define momentum rate.

Step4: modify back propagation code from *ComputationalGraphPrimer* library to replace vanilla SGD optimizer with momentum SGD

Step5: momentum required keeping track of parameter history, hence two variables, namely, *step_hist* and *bias_history*, were incorporated in the code.

Step6: Overall, compared to Vanilla SGD, SGD with momentum lead to faster convergence and lower loss (Figure 4)

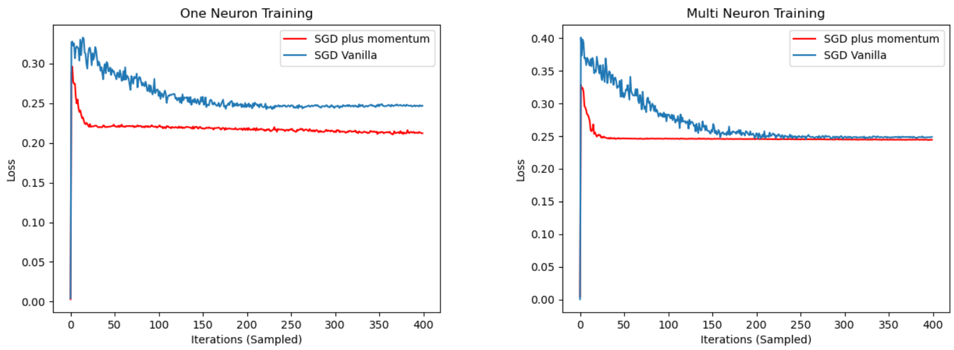

Figure 4: Left: Comparison of Vanilla SGD and SGD with momentum for one neuron model, Right: Comparison of Vanilla SGD and SGD with momentum for multi neuron model

3 Implementation

CODE-one_neuron_classifier_sgd_plus

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

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

# inherited class
class cgpSuperCharged(ComputationalGraphPrimer):
    def __init__(self, mu=0.0, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.mu = mu

    def backprop_and_update_params_one_neuron_model(
        self, y_error_avg, data_tuple_avg, deriv_sigmoid_avg
    ):
        """
        This function is copied over from
        ComputationalGraphPrimer.py Version 1.0.8
        Modified by: Varun Aggarwal

        Modifications:
```

```
    Added SGDplusMomentum
    """
    input_vars = self.independent_vars
    vals_for_input_vars_dict = dict(zip(input_vars, list(data_tuple_avg)
        ↪ ))
    vals_for_learnable_params = self.vals_for_learnable_params

    # preparing varibles
    step_hist = list(np.zeros(len(self.vals_for_learnable_params)))
    bias_hist = 0

    for i, param in enumerate(self.vals_for_learnable_params):
        ## calculate the next step in the parameter hyperplane

        # representing in same notation as the HW text
        g_tp1 = (
            y_error_avg
            * vals_for_input_vars_dict[input_vars[i]]
            * deriv_sigmoid_avg
        )

        step = self.mu * self.step_hist[i] + self.learning_rate * g_tp1
        self.vals_for_learnable_params[param] += step

        # update step_hist
        self.step_hist[i] = step

    ## Bias momentum step
    self.bias_hist = (
        self.mu * self.bias_hist
        + self.learning_rate * y_error_avg * deriv_sigmoid_avg
    )
    self.bias += self.bias_hist

def run_training_loop_one_neuron_model(self, training_data):
    """
    This function is copied over from
    ComputationalGraphPrimer.py Version 1.0.8
    Modified by: Varun Aggarwal

    Modifications:
    initializing step_hist and bias_hist
```

```
"""
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

# preparing varibles
self.bias_hist = 0
self.step_hist = list(np.zeros(len(self.learnable_params)))

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


# SGD with momentum
cgp = cgpSuperCharged(
    one_neuron_model=True,
    expressions=["xw=ab*xa+bc*xb+cd*xc+ac*xd"],
    output_vars=["xw"],
    dataset_size=5000,
    learning_rate=1e-3,
    training_iterations=40000,
    batch_size=16,
    display_loss_how_often=100,
    debug=True,
    mu=0.9,
)

# Vanilla SGD
cgp_original = 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,
```

```
    training_iterations=40000,
    batch_size=16,
    display_loss_how_often=100,
    debug=True,
)

plt.show(block=True)
# Loss with SGDmomentum
cgp.parse_expressions()
training_data = cgp.gen_training_data()
loss_running_record_mu = cgp.run_training_loop_one_neuron_model(
    ↪ training_data)

# Loss with VanillaSGD
cgp_original.parse_expressions()
training_data = cgp_original.gen_training_data()
loss_running_record = cgp_original.run_training_loop_one_neuron_model(
    ↪ training_data)

# Plotting Loss
plt.figure()
plt.plot(loss_running_record_mu, color="red")
plt.plot(loss_running_record)
plt.legend(["SGD␣plus␣momentum", "SGD␣Vanilla"])
plt.title("One␣Neuron␣Training")
plt.xlabel("Iterations␣(Sampled)")
plt.ylabel("Loss")
plt.savefig("../output/one_with_momentum.png")
```

CODE-multi_neuron_classifier_sgd_plus

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

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

# inherited class
class cgpSuperCharged(ComputationalGraphPrimer):
    def __init__(self, mu=0.0, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.mu = mu

    def backprop_and_update_params_multi_neuron_model(self, y_error,
        ↪ class_labels):
        """
        This function is copied over from
        ComputationalGraphPrimer.py Version 1.0.8
        Modified by: Varun Aggarwal

        Modifications:
        Added SGDplusMomentum
        """
        # 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']]}
## "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))
            ]
        )
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):

                # representing in same notation as the HW text
                g_tp1 = (
                    input_vals_avg[i]
                    * pred_err_backproped_at_layers[back_layer_index][j]
                ) * deriv_sigmoid_avg[j]

                step = self.mu * self.step_hist[i] + self.learning_rate *
                    ↪ g_tp1
                self.vals_for_learnable_params[param] += step

                # update step_hist
                self.step_hist[i] = step

        ## Bias momentum step
        self.bias_hist = self.mu * self.bias_hist + self.learning_rate *
            ↪ sum(
            pred_err_backproped_at_layers[back_layer_index]
        ) * sum(deriv_sigmoid_avg) / len(deriv_sigmoid_avg)
        self.bias[back_layer_index - 1] += self.bias_hist

def run_training_loop_multi_neuron_model(self, training_data):
    """
    This function is copied over from
    ComputationalGraphPrimer.py Version 1.0.8
    Modified by: Varun Aggarwal

    Modifications:
    initializing step_hist and bias_hist
    """

    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

# preparing varibles
self.bias_hist = 0
self.step_hist = list(np.zeros(len(self.learnable_params)))
```

```
        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


# SGD with momentum
cgp = cgpSuperCharged(
    num_layers=3,
    layers_config=[4, 2, 1],
    expressions=[
        "xw=ap*xp+aq*xq+ar*xr+as*xs",
        "xz=bp*xp+bq*xq+br*xr+bs*xs",
```

```
        "xo=cp*xw+cq*xz",
    ],
    output_vars=["xo"],
    dataset_size=5000,
    learning_rate=1e-3,
    training_iterations=40000,
    batch_size=8,
    display_loss_how_often=100,
    debug=True,
    mu=0.9,
)

# Vanilla SGD
cgp_original = ComputationalGraphPrimer(
    num_layers=3,
    layers_config=[4, 2, 1],
    expressions=[
        "xw=ap*xp+aq*xq+ar*xr+as*xs",
        "xz=bp*xp+bq*xq+br*xr+bs*xs",
        "xo=cp*xw+cq*xz",
    ],
    output_vars=["xo"],
    dataset_size=5000,
    learning_rate=1e-3,
    training_iterations=40000,
    batch_size=8,
    display_loss_how_often=100,
    debug=True,
)

# Loss with SGDmomentum
cgp.parse_multi_layer_expressions()
training_data = cgp.gen_training_data()
loss_running_record_mu = cgp.run_training_loop_multi_neuron_model(
    ↪ training_data)

# Loss with VanillaSGD
cgp_original.parse_multi_layer_expressions()
training_data = cgp_original.gen_training_data()
loss_running_record = cgp_original.run_training_loop_multi_neuron_model(
    ↪ training_data)
```

```
# Plotting Loss
plt.figure()
plt.plot(loss_running_record_mu, color="red")
plt.plot(loss_running_record)
plt.legend(["SGD plus momentum", "SGD Vanilla"])
plt.title("Multi Neuron Training")
plt.xlabel("Iterations (Sampled)")
plt.ylabel("Loss")
plt.savefig("../output/multi_with_momentum.png")
```

CODE-verify_with_torchnn.py

```
import random
import numpy
import torch
import os

import sys
sys.path.append("..")

seed = 0
random.seed(seed)
numpy.random.seed(seed)
torch.manual_seed(seed)
os.environ['PYTHONHASHSEED'] = str(seed)

from ComputationalGraphPrimer import *

cgp = ComputationalGraphPrimer(
             expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'], # Only used to
                ↪ determine the data dimensionality
             dataset_size = 5000,
             # learning_rate = 1e-6, # For the multi-neuron option below
            # learning_rate = 1e-3, # For the one-neuron option below
            learning_rate = 5 * 1e-2, # Also for the one-neuron option
               ↪ below
             training_iterations = 40000,
             batch_size = 8,
             display_loss_how_often = 100,
     )


# 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,
```

```
# training_iterations = 40000,
# batch_size = 8,
# display_loss_how_often = 100,
# )

## This call is needed for generating the training data:

# multilayer
# cgp.parse_multi_layer_expressions()
# training_data = cgp.gen_training_data()
# cgp.run_training_with_torchnn('multi_neuron', training_data) ## (B)

# one neuran
cgp.parse_expressions()
training_data = cgp.gen_training_data()
cgp.run_training_with_torchnn('one_neuron', training_data) ## (A)
```

4 Lessons Learned

Once again, the programming homework was straightforward and covered the basics of optimizer implementation. It was cumbersome to go through code of *ComputationalGraphPrimer* which was understandable as generally going through code written by someone else is difficult to comprehend. I did not find any major issues with implementation of SGD with momentum. I was primarily confused about vector dimensions although, using debugger in PyCharm helped.

5 Suggested Enhancements

The assignment could be given in form of a single python file with missing code block for SGD with momentum. By doing so, the need for extensively understanding the code for *ComputationalGraphPrimer* could have been avoided. Although on the hindsight, I am glad to have gone through the code.