Homework 3 - Deep Learning (ECE 60146)

Souradip Pal
pal43@purdue.edu
PUID 0034772329

February 6, 2023

1 Introduction

In this homework, the programming tasks give an overview of how to use *ComputationalGraphPrimer* python module to create and understand computational graphs from scratch which is one of the core tools for training neural networks. It also gives a holistic picture of the Stochastic Gradient Descent algorithm which is used in the optimization of the loss functions and a few of its more popular and effective variants like the Stochastic Gradient Descent with Momentum and Adaptive Moment Estimation. In this assignment, these algorithms were implemented and were used to train single-layer and multi-layer classifier networks created using *ComputationalGraphPrimer* package, and their performances were compared based on different hyper-parameters. Some ideas related to the logic of the optimization algorithms were taken from the lecture slides presented by Prof. Kak and from the previous year's solutions for completing this assignment.

2 Methodology

The vanilla **Stochastic Gradient Descent(SGD)** algorithm updates the parameter based on a batch of training samples instead of the entire set of samples from the dataset. This helps in overcoming the issue of getting stuck in a local minimum while optimizing a loss function. However, vanilla SGD has the issue of slow convergence and oscillations when close to the optimum if the step size is fixed. Thus the variants of SGD introduce some form of step size optimization which help in faster and better convergence of the algorithms. Two popular variants of the SGD algorithms have been implemented from scratch in this assignment and used for training a single-layer and a multi-layer neural network. The two algorithms are described in the following sections.

2.1 Stochastic Gradient Descent with Momentum (SGD+)

In vanilla SGD, the parameters are updated using the gradient of the loss with respect to the corresponding parameter as follows:

$$p_{t+1} = p_t - \alpha \nabla_p L(p_t) \tag{1}$$

Here, p_t is the value of the parameter at iteration t, L is the loss function and α is the learning rate. In SGD+, the concept of momentum was introduced for step size optimization in which the value of the step in the previous iteration is used along with the current value of the gradient for updating the value of the parameters in each of the current iterations. The following equations show the way in

which the learnable parameters are updated using momentum.

$$v_{t+1} = \mu v_t - \alpha \nabla_p L(p_t) \tag{2}$$

$$p_{t+1} = p_t + v_{t+1} \tag{3}$$

Here, v_t stores the information of the step size used in the previous iteration, and $\mu \in [0, 1]$, known as the momentum coefficient, decides the weight given to the previous gradient step. Initially, v_0 is kept as 0. Thus SGD+ maintains a running decaying average mean of the past gradient steps and uses its value to stay on course towards the optimum. If the latest gradient step is similar to its previous gradient step, it would effectively double the step size whereas if the current gradient step is not in accordance with the previous gradient step due to possible reversal in direction then it would effectively reduce the step size drastically.

2.2 Adaptive Moment Estimation (Adam)

This approach extends the SGD+ algorithm and addresses the problem of Sparse Gradients by using a combination of the momentum term and the adaptive gradient term. In gradient adaptation, different learning rates are used for different parameters so that the overall step is not influenced by the large number of parameters having only zero gradients. This algorithm keeps track of the running averages of both the first and second-order moments of the gradients which are then used in calculating the step size. The following equations show the way in which the learnable parameters are updated using Adam.

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) g_{t+1} \tag{4}$$

$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) g_{t+1}^2 \tag{5}$$

$$\hat{m_{t+1}} = \frac{m_{t+1}}{1 - \beta_1^{(t+1)}} \tag{6}$$

$$\hat{v_{t+1}} = \frac{v_{t+1}}{1 - \beta_2^{(t+1)}} \tag{7}$$

$$p_{t+1} = p_t - \alpha \frac{\hat{m_{t+1}}}{\sqrt{\hat{v_{t+1}} + \epsilon}} \tag{8}$$

Here, $g_{t+1} = \nabla_p L(p_t)$, m_t and v_t maintain the first and second order moments respectively which are initialized to 0 at the start of the training. $\hat{m}_t$ and $\hat{v}_t$ are values of the moments after bias correction needed for zero initialization, β_1 and β_2 controls the decay rates for the moments and α is the learning rate. The constant ϵ is set to a small value to avoid division by 0 in the denominator.

3 Implementation and Results

3.1 Task 3: Programming Tasks

3.1.1 One Neuron Classifier

- **SGD**: The python script present in the *one_neuron_classifier.py* was executed from the *Examples* directory in the *ComputationalGraphPrimer* module to train a single-layer neural network using SGD. The method *run_training_loop_one_neuron_model* is called for running the training iterations which in turn calls the method *backprop_and_update_params_one_neuron_model* for updating the learnable parameters in each iteration. The training process was run for two different learning rates(α) **0.001** and **0.005**. Fig 1 shows the plot of the training loss values at every 100 iterations for the two different learning rates.

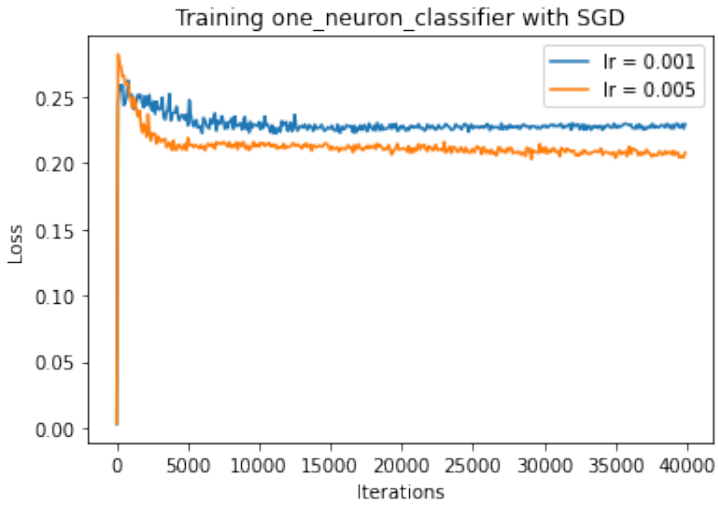


Figure 1: SGD training loss for one neuron classifier with learning rates 0.001 and 0.005

- **SGD+** : In this task, a new subclass *ComputationalGraphPrimerSGDPlus* of the base *ComputationalGraphPrimer* class was created and the corresponding *run_training_loop_one_neuron_model* and *backprop_and_update_params_one_neuron_model* methods were overridden in accordance with the logic of the SGD+ algorithm. In this case, the momentum coefficient (μ) was taken as **0.9** for this experiment since the typical value for μ is close to 1.0. Moreover, v_t is initialized to 0.0 for each parameter including the bias. Keeping other hyper-parameters except for the learning rate as default, this new module was used for training a single-layer neural network and its running losses were captured for every 100 iterations and displayed in a plot. The training process was run for two different learning rates(α) **0.001** and **0.005**. Fig 2 shows the plot of the training loss values at every 100 iterations for the two different learning rates.

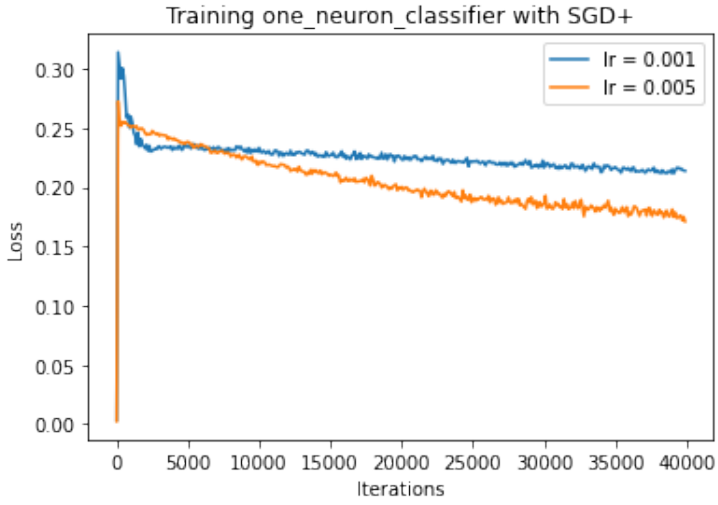


Figure 2: SGD+ (with momentum) training loss for one neuron classifier with learning rates 0.001 and 0.005

- **Adam**: Another new subclass *ComputationalGraphPrimerAdam* was also created and its gradient update methods were modified in accordance with the Adam algorithm. For the Adam optimization algorithm, the hyper-parameters were set as follows: $\beta_1 = 0.9$, $\beta_2 = 0.99$, $\epsilon = 10^{-8}$ since the typical values for β_1, β_2 are close to 1.0. Moreover, m_t, v_t are initialized to 0.0 for each parameter including the bias. This new module was used for training a single-layer neural network and its running losses were captured for every 100 iterations and displayed in a plot. The training process was run for two different learning rates(α) **0.001** and **0.005** as done with SGD+. Fig 3 shows the plot of the training loss values at every 100 iterations for the two different learning rates.

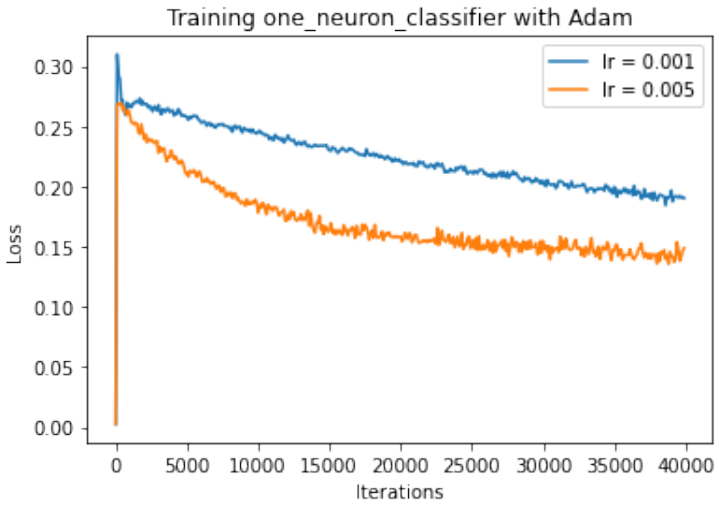
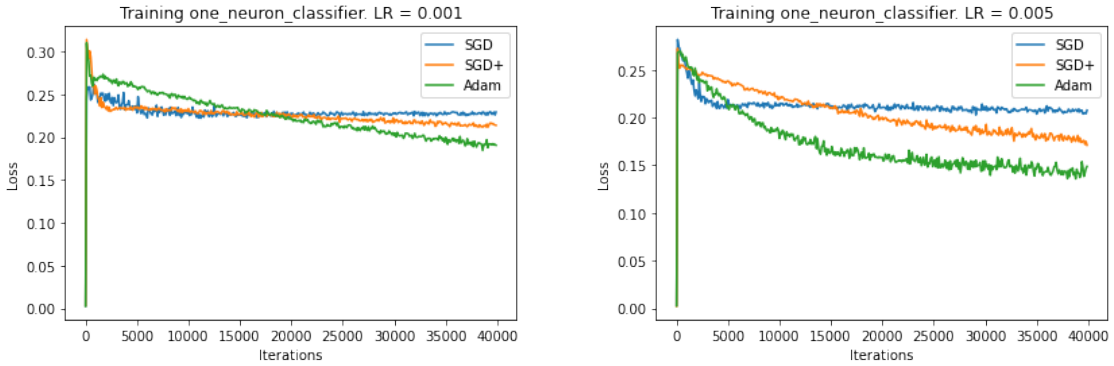


Figure 3: Adam training loss for one neuron classifier

- **Results & Comparison**: Fig 4 shows the performance comparison between the different optimization techniques used for training the one-neuron classifier model. It is evident from the plot that Adam is successful in finding a much better minimum value of the loss function than SGD and SGD+. It can be seen that SGD with momentum does perform better than the vanilla SGD. It is also noticed that with a different learning rate of 0.005, the training loss decreases in a faster and much more gradual fashion. However, with a larger learning rate, the loss function is prone to oscillation near the optimum in spite of the convergence being much faster. Thus learning rate also plays an important role in gradient descent algorithms.

3.1.2 Multi Neuron Classifier

- **SGD**: The python script present in the *multi_neuron_classifier.py* was executed from the *Examples* directory in the *ComputationalGraphPrimer* module to train a multi-layer neural network using SGD. The method *run_training_loop_multi_neuron_model* is called for running the training iterations which in turn calls the method *backprop_and_update_params_multi_neuron_model* for updating the learnable parameters in each iteration and for each layer. The training process was run for two different learning rates(α) **0.001** and **0.005**. Fig 5 shows the plot of the training loss values at every 100 iterations for the two different learning rates.
- **SGD+** : In this task, the gradient updates in *backprop_and_update_params_multi_neuron_model* method of the newly created subclass *ComputationalGraphPrimerSGDPlus* were overridden in accordance with the equations of SGD+ algorithm. The variable initialization was performed in



(a) SGD, SGD+, Adam training losses for one neuron classifier with learning rate 0.001

(b) SGD, SGD+, Adam training losses for one neuron classifier with learning rate 0.005

Figure 4: Comparison of SGD, SGD+, Adam training losses for one neuron classifier

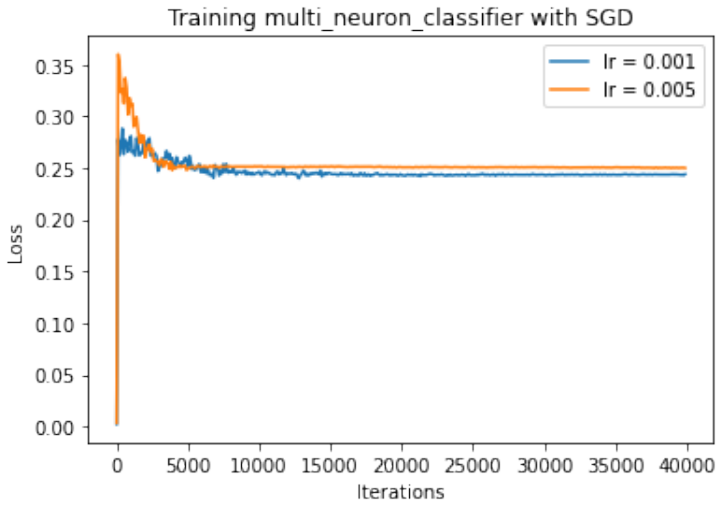


Figure 5: SGD training loss for a multi-neuron classifier with learning rates 0.001 and 0.005

the *run_training_loop_multi_neuron_model* method setting v_t for each parameter to 0.0 in each of the layers. The momentum coefficient (μ) was taken as **0.9** for this experiment. Keeping other hyper-parameters except for the learning rate as default, this new module was used for training a multi-layer neural network and its running losses were captured for every 100 iterations and displayed in a plot. The training process was run for two different learning rates(α) **0.001** and **0.005**. Fig 6 shows the plot of the training loss values at every 100 iterations for the two different learning rates

- **Adam**: The gradient updates in *backprop_and_update_params_multi_neuron_model* method of the newly created subclass *ComputationalGraphPrimerAdam* were modified in accordance with the Adam algorithm. The variables were initialized in *run_training_loop_multi_neuron_model* method. For the Adam optimization algorithm, the hyper-parameters were set as follows: $\beta_1 = 0.9$, $\beta_2 = 0.99$, $\epsilon = 10^{-8}$. This overridden method was used for training a multi-layer neural network and its running losses were captured for every 100 iterations and displayed in a plot. The training

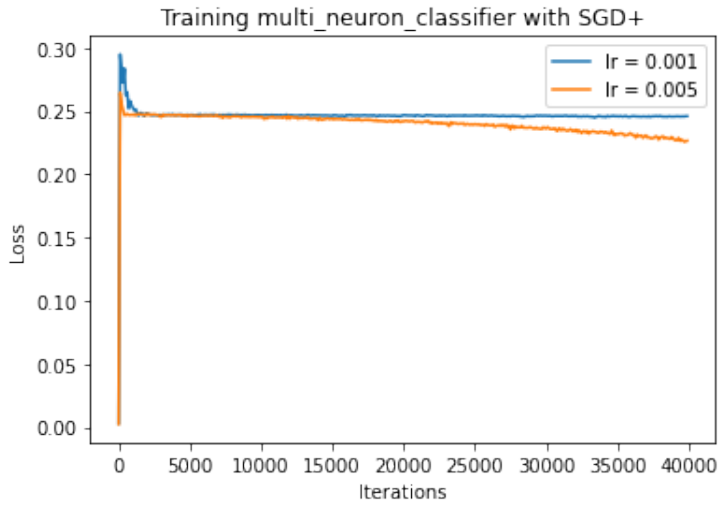


Figure 6: SGD+ (with momentum) training loss for multi-neuron classifier with learning rates 0.001 and 0.005

process was run for two different learning rates(α) **0.001** and **0.005**. Fig 7 shows the plot of the training loss values at every 100 iterations for the two different learning rates.

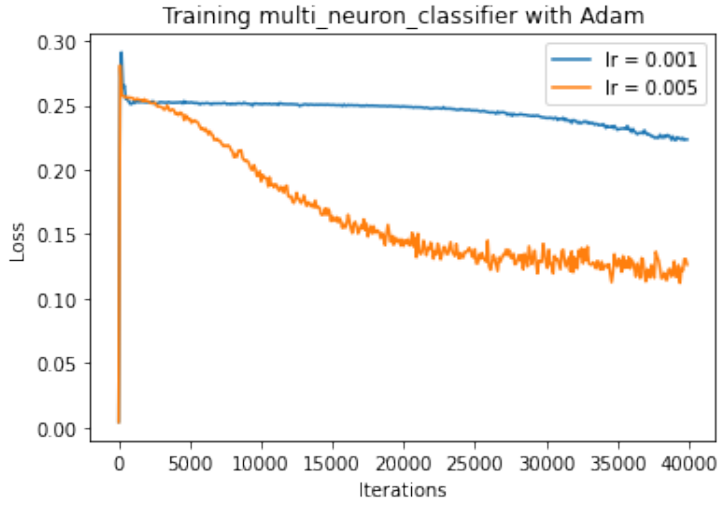
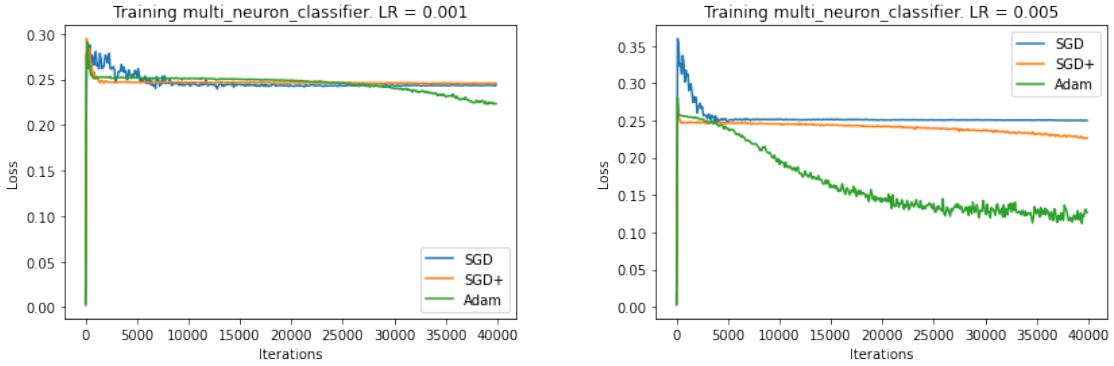


Figure 7: Adam training loss for a multi-neuron classifier with learning rates 0.001 and 0.005

- **Results & Comparison**: Fig 8 shows the performance comparison between the different optimization techniques used for training the multi-neuron classifier model. It is evident from the plot that Adam clearly outperforms SGD and SGD+ in finding a much better minimum value of the loss function. This is because, in the multi-neuron case with the increase in the number of learnable parameters, the problem of Sparse Gradients is much more prevalent which is addressed by Adam via gradients adaptation. It can be seen that SGD with momentum does perform better than the vanilla SGD. It can also be seen that with a different learning rate of 0.005, the training loss decreases in a faster and much more gradual fashion similar to the

one-neuron classifier case. With a larger learning rate, the loss function is prone to oscillation near the optimum as well. Thus choosing the appropriate learning rate also plays an important role in the case of multi-layered networks.



(a) SGD, SGD+, Adam training losses for multi neuron classifier with learning rate 0.001

(b) SGD, SGD+, Adam training losses for multi neuron classifier with learning rate 0.005

Figure 8: Comparison of SGD, SGD+, Adam training losses for multi neuron classifier

4 Conclusion

In conclusion, Stochastic Gradient Descent is a powerful technique for optimizing loss functions while training single and multi-layered neural networks. Although the vanilla SGD suffers from convergence issues and does not perform well in some cases, those are overcome by better variants of SGD like SGD+ and Adam. The hyper-parameters also play an important role in the convergence of the training loss. For example, Fig 9 shows how the training loss is sensitive towards the momentum coefficient while training the one-neuron model using SGD+. If the momentum coefficient is closer to 1.0, the performance is better. Hence, choosing good hyper-parameters is essential in making the training much more efficient and stable.

5 Source Code

```
# -*- coding: utf-8 -*-
"""hw3_SouradipPal.ipynb

Automatically generated by Colaboratory.

Original file is located at
    https://colab.research.google.com/drive/1RxGexYVoxE5hwchlgXHGQAmR0w_EQaUu
"""

from google.colab import drive
drive.mount('/content/drive')

# Commented out IPython magic to ensure Python compatibility.
# %cd /content/drive/MyDrive/Purdue/ECE60146/HW3

!wget -O ComputationalGraphPrimer-1.1.2.tar.gz \
    https://engineering.purdue.edu/kak/distCGP/ComputationalGraphPrimer-1.1.2.tar.gz?download

!tar -xvf  ComputationalGraphPrimer-1.2.0.tar.gz

# Commented out IPython magic to ensure Python compatibility.
```

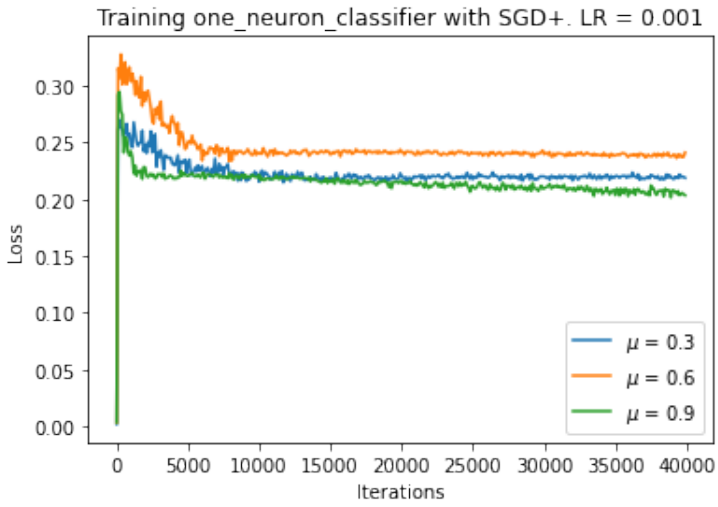


Figure 9: SGD+ training loss for one-neuron classifier with momentum(μ) 0.3, 0.6, and 0.9

```
# %cd ComputationalGraphPrimer-1.1.2

!python setup.py install

# %load_ext autoreload
# %autoreload 2

import random
import numpy as np
import operator
import matplotlib.pyplot as plt

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

from ComputationalGraphPrimer import *

# Running an example computational graph
cgp = ComputationalGraphPrimer(
                        expressions = ['xx=xa^2',
                                       'xy=ab*xx+ac*xa',
                                       'xz=bc*xx+xy',
                                       'xw=cd*xx+xz^3'],
                        output_vars = ['xw'],
                        dataset_size = 10000,
                        learning_rate = 1e-6,
                        grad_delta    = 1e-4,
                        display_loss_how_often = 1000,
                )

cgp.parse_expressions()
cgp.display_network2()
cgp.gen_gt_dataset(vals_for_learnable_params = {'ab':1.0, 'bc':2.0, 'cd':3.0, 'ac':4.0})
cgp.train_on_all_data()
cgp.plot_loss()

# Custom ComputationalGraphPrimer class to return running losses
class CustomComputationalGraphPrimer(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super(CustomComputationalGraphPrimer, self).__init__(*args, **kwargs)

    ##########################################################################################################
    ########################################## one neuron model ##############################################
    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)

    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_iterations = 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_iterations += loss_avg
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_iterations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_iterations)
            print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))
            avg_loss_over_iterations = 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 running losses for plotting and comparison
    return  loss_running_record


##########################################################################################################
######################################## multi neuron model ##############################################
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

        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_iterations = 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_iterations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))
                avg_loss_over_iterations = 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 running losses for plotting and comparison
        return loss_running_record

# Helper Method for plots
def plot_loss(loss, model, optimizer, lr):
    plt.figure()
    plt.plot(np.arange(len(loss))*100, loss)
    plt.title(f'Training {model} using {optimizer} (LR = {lr})')
    plt.xlabel('Iterations')
    plt.ylabel('Loss')
    plt.show()

# Training one_neuron_classifier with SGD. Learning rate = 0.001
cgp = CustomComputationalGraphPrimer(
                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 = 8,
                display_loss_how_often = 100,
                debug = True,
      )


cgp.parse_expressions()
cgp.display_one_neuron_network()
training_data = cgp.gen_training_data()
```

```
one_neuron_classifier_running_loss = cgp.run_training_loop_one_neuron_model(training_data)

# Plotting One Neuron Classifier Loss
plot_loss(one_neuron_classifier_running_loss, 'one_neuron_classifier', 'SGD', 0.001)

# Training one_neuron_classifier with SGD. Learning rate = 0.005
cgp = CustomComputationalGraphPrimer(
              one_neuron_model = True,
              expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
              output_vars = ['xw'],
              dataset_size = 5000,
              learning_rate = 5 * 1e-3,
              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()
one_neuron_classifier_running_loss_2 = cgp.run_training_loop_one_neuron_model(training_data)

# Plotting One Neuron Classifier Loss
plot_loss(one_neuron_classifier_running_loss_2, 'one_neuron_classifier', 'SGD', 0.005)

# Training multi_neuron_classifier with SGD. Learning rate = 0.001.
cgp_multi = CustomComputationalGraphPrimer(
              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,
      )

cgp_multi.parse_multi_layer_expressions()
cgp_multi.display_multi_neuron_network()
training_data = cgp_multi.gen_training_data()
multi_neuron_classifier_running_loss = cgp_multi.run_training_loop_multi_neuron_model(training_data)

# Plotting Multi Neuron Classifier Loss
plot_loss(multi_neuron_classifier_running_loss, 'multi_neuron_classifier', 'SGD', 0.001)

# Training multi_neuron_classifier with SGD. Learning rate = 0.005
cgp_multi = CustomComputationalGraphPrimer(
              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 = 5 * 1e-3,
              training_iterations = 40000,
              batch_size = 8,
              display_loss_how_often = 100,
              debug = True,
      )

cgp_multi.parse_multi_layer_expressions()
cgp_multi.display_multi_neuron_network()
training_data = cgp_multi.gen_training_data()
multi_neuron_classifier_running_loss_2 = cgp_multi.run_training_loop_multi_neuron_model(training_data)

# Plotting Multi Neuron Classifier Loss
plot_loss(multi_neuron_classifier_running_loss_2, 'multi_neuron_classifier', 'SGD', 0.005)

# Subclass of ComputationalGraphPrimer having optimizer as SGD with Momentum
```

```
class ComputationalGraphPrimerSGDPlus(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super(ComputationalGraphPrimerSGDPlus, self).__init__(*args, **kwargs)

    ##########################################################################################################
    ######################################## one neuron model ################################################

    def run_training_loop_one_neuron_model(self, training_data, mu):
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
        self.bias = random.uniform(0,1)

        ################################### Source Code Modification Start ###################################
        ## Initialize v_t and mu for SGD with momentum for single-layer model
        self.mu = mu
        self.v_weights = {param: 0.0 for param in self.learnable_params}
        self.v_bias = 0.0
        ################################### Source Code Modification End #####################################

        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_iterations = 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_iterations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))
                avg_loss_over_iterations = 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 running losses for plotting and comparison
        return loss_running_record
```

```
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):

    ################################## Source Code Modification Start ####################################
        ## Calculate gradient update and v_(t+1)
        g =  y_error * vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoid
        self.v_weights[param] = self.mu * self.v_weights[param] + self.learning_rate * g

        ## Calculate the next step in the parameter hyperplane
        step = self.v_weights[param]

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

    ## Calculate v_(t+1) for bias
    self.v_bias = self.mu * self.v_bias + self.learning_rate * y_error * deriv_sigmoid

    ## Update the bias
    self.bias += self.v_bias
    ################################## Source Code Modification End ####################################


#############################################################################################################
######################################## multi neuron model #################################################

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

    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)]

    ################################## Source Code Modification Start ####################################
    ## Initialize v_t and mu for SGD with momentum for multi-layer model
    self.mu = mu
    self.v_weights = {param: 0.0 for param in self.learnable_params}
    self.v_bias = [0.0 for _ in range(self.num_layers-1)]

    ################################## Source Code Modification End ####################################
```

```
    data_loader = DataLoader(training_data, batch_size=self.batch_size)
    loss_running_record = []
    i = 0
    avg_loss_over_iterations = 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_iterations += loss_avg
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_iterations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_iterations)
            print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))
            avg_loss_over_iterations = 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 running losses for plotting and comparison
    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]
        vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]

        layer_params = self.layer_params[back_layer_index]
        transposed_layer_params = list(zip(*layer_params))

        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):
        ################################## Source Code Modification Start ###################################

                ## Calculate gradient update and v_(t+1)
                gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]
                g = gradient_of_loss_for_param * deriv_sigmoid_avg[j]
                self.v_weights[param] = self.mu * self.v_weights[param] + self.learning_rate * g

                ## Calculate the next step in the parameter hyperplane
                step = self.v_weights[param]

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

        ## Calculate gradient update and v_(t+1) for bias
        g = sum(pred_err_backproped_at_layers[back_layer_index]) * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)
        self.v_bias[back_layer_index-1] = self.mu * self.v_bias[back_layer_index-1] + self.learning_rate * g

        ## Update the bias
        self.bias[back_layer_index-1] += self.v_bias[back_layer_index-1]
```

```
            ################################## Source Code Modification End ###################################

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

# Training one_neuron_classifier with ComputationalGraphPrimer
# using optimizer as SGD with Momentum. Learning Rate = 0.001, mu = 0.9
cgp_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,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True
      )


cgp_plus.parse_expressions()
cgp_plus.display_one_neuron_network()
training_data = cgp_plus.gen_training_data()
one_neuron_classifier_sgdp_running_loss = cgp_plus.run_training_loop_one_neuron_model(training_data, 0.9)

# Plotting One Neuron Classifier Loss
plot_loss(one_neuron_classifier_running_loss, 'one_neuron_classifier', 'SGD+', 0.001)

# Training one_neuron_classifier with ComputationalGraphPrimer
# using optimizer as SGD with Momentum. Learning Rate = 0.005, mu = 0.9
cgp_plus = ComputationalGraphPrimerSGDPlus(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 5 * 1e-3,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True
      )


cgp_plus.parse_expressions()
cgp_plus.display_one_neuron_network()
training_data = cgp_plus.gen_training_data()
one_neuron_classifier_sgdp_running_loss_2 = cgp_plus.run_training_loop_one_neuron_model(training_data, 0.9)

# Plotting One Neuron Classifier Loss
plot_loss(one_neuron_classifier_running_loss_2, 'one_neuron_classifier', 'SGD+', 0.005)

# Training multi_neuron_classifier with ComputationalGraphPrimer
# using optimizer as SGD with Momentum. Learning Rate = 0.001, mu = 0.9
cgp_plus_multi = 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,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

cgp_plus_multi.parse_multi_layer_expressions()
cgp_plus_multi.display_multi_neuron_network()
training_data = cgp_plus_multi.gen_training_data()
multi_neuron_classifier_sgdp_running_loss = cgp_plus_multi.run_training_loop_multi_neuron_model(training_data, 0.9)

# Plotting Multi Neuron Classifier Loss
plot_loss(multi_neuron_classifier_sgdp_running_loss, 'multi_neuron_classifier', 'SGD+', 0.001)

# Training multi_neuron_classifier with ComputationalGraphPrimer
```

```
# using optimizer as SGD with Momentum. Learning Rate = 0.005, mu = 0.9
cgp_plus_multi = 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 = 5 * 1e-3,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
     )

cgp_plus_multi.parse_multi_layer_expressions()
cgp_plus_multi.display_multi_neuron_network()
training_data = cgp_plus_multi.gen_training_data()
multi_neuron_classifier_sgdp_running_loss_2 = cgp_plus_multi.run_training_loop_multi_neuron_model(training_data, 0.9)

# Plotting Multi Neuron Classifier Loss
plot_loss(multi_neuron_classifier_sgdp_running_loss_2, 'multi_neuron_classifier', 'SGD+', 0.005)

# Subclass of ComputationalGraphPrimer having optimizer as Adam
class ComputationalGraphPrimerAdam(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super(ComputationalGraphPrimerAdam, self).__init__(*args, **kwargs)

    ##########################################################################################################
    ######################################### one neuron model ###############################################
    def run_training_loop_one_neuron_model(self, training_data, beta_1, beta_2, eps):
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = random.uniform(0,1)

        ################################### Source Code Modification Start ###################################
        ## Initialize m_t, v_t, beta_1, beta_2 and epsilon for Adam for single-layer model
        self.beta_1 = beta_1
        self.beta_2 = beta_2
        self.eps = eps

        self.v_weights = {param: 0.0 for param in self.learnable_params}
        self.v_bias = 0.0
        self.m_weights = {param: 0.0 for param in self.learnable_params}
        self.m_bias = 0.0

        ################################### Source Code Modification End #####################################

        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_iterations = 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_iterations += loss_avg
        if i%(self.display_loss_how_often) == 0:
            avg_loss_over_iterations /= self.display_loss_how_often
            loss_running_record.append(avg_loss_over_iterations)
            print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))
            avg_loss_over_iterations = 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, i+1)

    ## return running losses for plotting and comparison
    return loss_running_record

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

################################### Source Code Modification Start ###################################
        ## Calculate gradient update, m_(t+1) and v_(t+1)
        g = y_error * vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoid
        self.m_weights[param] = self.beta_1 * self.m_weights[param] + (1 - self.beta_1) * g
        self.v_weights[param] = self.beta_2 * self.v_weights[param] + (1 - self.beta_2) * g**2

        ## Apply bias correction
        m_corrected = self.m_weights[param] / (1 - self.beta_1**iter)
        v_corrected = self.v_weights[param] / (1 - self.beta_2**iter)

        ## Calculate the next step in the parameter hyperplane
        step = self.learning_rate* (m_corrected / (np.sqrt(v_corrected) + self.eps))

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

    ## Calculate gradient update, m_(t+1) and v_(t+1) for bias
    self.m_bias = self.beta_1 * self.m_bias + (1 - self.beta_1) * (y_error * deriv_sigmoid)
    self.v_bias = self.beta_2 * self.v_bias + (1 - self.beta_2) * (y_error * deriv_sigmoid)**2

    ## Apply bias correction
    m_corrected = self.m_bias / (1 - self.beta_1**iter)
    v_corrected = self.v_bias / (1 - self.beta_2**iter)

    ## Update the bias
    self.bias += self.learning_rate * (m_corrected/np.sqrt(v_corrected + self.eps))

################################### Source Code Modification End #####################################

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

#######################################################################################################
######################################### multi neuron model ##########################################
def run_training_loop_multi_neuron_model(self, training_data, beta_1, beta_2, eps):
```

```
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


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)]

################################### Source Code Modification Start ###################################
## Initialize m_t, v_t, beta_1, beta_2 and epsilon for Adam for multi-layer model
self.beta_1 = beta_1
self.beta_2 = beta_2
self.eps = eps

self.v_weights = {param: 0.0 for param in self.learnable_params}
self.v_bias = [0.0 for _ in range(self.num_layers-1)]
self.m_weights = {param: 0.0 for param in self.learnable_params}
self.m_bias = [0.0 for _ in range(self.num_layers-1)]

################################### Source Code Modification End #####################################

data_loader = DataLoader(training_data, batch_size=self.batch_size)
loss_running_record = []
i = 0
avg_loss_over_iterations = 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_iterations += loss_avg
    if i%(self.display_loss_how_often) == 0:
        avg_loss_over_iterations /= self.display_loss_how_often
        loss_running_record.append(avg_loss_over_iterations)
        print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))
        avg_loss_over_iterations = 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, i+1)

## return running losses for plotting and comparison
return loss_running_record
```

```
    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels, iter):
        # 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]
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]

            layer_params = self.layer_params[back_layer_index]
            transposed_layer_params = list(zip(*layer_params))

            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):

    ################################### Source Code Modification Start ###################################

                    ## Calculate gradient update, m_(t+1) and v_(t+1)
                    gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]
                    g = gradient_of_loss_for_param * deriv_sigmoid_avg[j]
                    self.m_weights[param] = self.beta_1 * self.m_weights[param] + (1 - self.beta_1) * g
                    self.v_weights[param] = self.beta_2 * self.v_weights[param] + (1 - self.beta_2) * g**2

                    ## Apply bias correction
                    m_corrected = self.m_weights[param] / (1 - self.beta_1**iter)
                    v_corrected = self.v_weights[param] / (1 - self.beta_2**iter)

                    ## Calculate the next step in the parameter hyperplane
                    step = self.learning_rate* (m_corrected / np.sqrt(v_corrected + self.eps))

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

            ## Calculate gradient update, m_(t+1) and v_(t+1) for bias
            g = sum(pred_err_backproped_at_layers[back_layer_index]) * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)
            self.m_bias[back_layer_index-1] = self.beta_1 * self.m_bias[back_layer_index-1] + (1 - self.beta_1) * g
            self.v_bias[back_layer_index-1] = self.beta_2 * self.v_bias[back_layer_index-1] + (1 - self.beta_2) * g**2

            ## Apply bias correction
            m_corrected = self.m_bias[back_layer_index-1] / (1 - self.beta_1**iter)
            v_corrected = self.v_bias[back_layer_index-1] / (1 - self.beta_2**iter)

            ## Update the bias
            self.bias[back_layer_index-1] += self.learning_rate * (m_corrected/np.sqrt(v_corrected + self.eps))

    ################################### Source Code Modification End #####################################

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

# Training one_neuron_classifier with ComputationalGraphPrimer using optimizer
# as Adam. Learning Rate = 0.001, beta_1 = 0.9, beta_2 = 0.99, eps = 1e-8
cgp_adam = ComputationalGraphPrimerAdam(
               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 = 8,
               display_loss_how_often = 100,
               debug = True
```

```
      )


cgp_adam.parse_expressions()
cgp_adam.display_one_neuron_network()
training_data = cgp_adam.gen_training_data()
one_neuron_classifier_adam_running_loss = cgp_adam.run_training_loop_one_neuron_model(training_data, 0.9, 0.99, 1e-8)

# Plotting One Neuron Classifier Loss
plot_loss(one_neuron_classifier_adam_running_loss, 'one_neuron_classifier', 'Adam', 0.001)

# Training one_neuron_classifier with ComputationalGraphPrimer using optimizer
# as Adam. Learning Rate = 0.005, beta_1 = 0.9, beta_2 = 0.99, eps = 1e-8
cgp_adam = ComputationalGraphPrimerAdam(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 5 * 1e-3,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True
      )


cgp_adam.parse_expressions()
cgp_adam.display_one_neuron_network()
training_data = cgp_adam.gen_training_data()
one_neuron_classifier_adam_running_loss_2 = cgp_adam.run_training_loop_one_neuron_model(training_data, 0.9, 0.99, 1e-8)

# Plotting One Neuron Classifier Loss
plot_loss(one_neuron_classifier_adam_running_loss_2, 'one_neuron_classifier', 'Adam', 0.005)

# Training multi_neuron_classifier with ComputationalGraphPrimer using optimizer
# as Adam. Learning Rate = 0.001, beta_1 = 0.9, beta_2 = 0.99, eps = 1e-8
cgp_adam_multi = ComputationalGraphPrimerAdam(
               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,
               debug = True,
      )

cgp_adam_multi.parse_multi_layer_expressions()
cgp_adam_multi.display_multi_neuron_network()
training_data = cgp_adam_multi.gen_training_data()
multi_neuron_classifier_adam_running_loss = cgp_adam_multi.run_training_loop_multi_neuron_model(training_data, 0.9, 0.99, 1e-8)

# Plotting Multi Neuron Classifier Loss
plot_loss(multi_neuron_classifier_adam_running_loss, 'multi_neuron_classifier', 'Adam', 0.001)

# Training multi_neuron_classifier with ComputationalGraphPrimer using optimizer
# as Adam. Learning Rate = 0.005, beta_1 = 0.9, beta_2 = 0.99, eps = 1e-8
cgp_adam_multi = ComputationalGraphPrimerAdam(
               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 = 5 * 1e-3,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )
```

```
cgp_adam_multi.parse_multi_layer_expressions()
cgp_adam_multi.display_multi_neuron_network()
training_data = cgp_adam_multi.gen_training_data()
multi_neuron_classifier_adam_running_loss_2 = cgp_adam_multi.run_training_loop_multi_neuron_model(training_data, 0.9, 0.99, 1e-8)

# Plotting Multi Neuron Classifier Loss
plot_loss(multi_neuron_classifier_adam_running_loss_2, 'multi_neuron_classifier', 'Adam', 0.005)

# Plotting One Neuron Classifier Losses. LR = 0.001.
plt.figure()
x = np.arange(len(one_neuron_classifier_running_loss))*100
plt.plot(x, one_neuron_classifier_running_loss, label='SGD')
plt.plot(x, one_neuron_classifier_sgdp_running_loss, label='SGD+')
plt.plot(x, one_neuron_classifier_adam_running_loss, label='Adam')
plt.title('Training one_neuron_classifier. LR = 0.001')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plotting One Neuron Classifier Losses. LR = 0.005
plt.figure()
x = np.arange(len(one_neuron_classifier_running_loss))*100
plt.plot(x, one_neuron_classifier_running_loss_2, label='SGD')
plt.plot(x, one_neuron_classifier_sgdp_running_loss_2, label='SGD+')
plt.plot(x, one_neuron_classifier_adam_running_loss_2, label='Adam')
plt.title('Training one_neuron_classifier. LR = 0.005')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plotting Multi Neuron Classifier Losses. LR = 0.001
plt.figure()
x = np.arange(len(multi_neuron_classifier_running_loss))*100
plt.plot(x, multi_neuron_classifier_running_loss, label='SGD')
plt.plot(x, multi_neuron_classifier_sgdp_running_loss, label='SGD+')
plt.plot(x, multi_neuron_classifier_adam_running_loss, label='Adam')
plt.title('Training multi_neuron_classifier. LR = 0.001')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plotting Multi Neuron Classifier Losses. LR = 0.005.
plt.figure()
x = np.arange(len(multi_neuron_classifier_running_loss))*100
plt.plot(x, multi_neuron_classifier_running_loss_2, label='SGD')
plt.plot(x, multi_neuron_classifier_sgdp_running_loss_2, label='SGD+')
plt.plot(x, multi_neuron_classifier_adam_running_loss_2, label='Adam')
plt.title('Training multi_neuron_classifier. LR = 0.005')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Training one_neuron_classifier with ComputationalGraphPrimer
# using optimizer as SGD with Momentum. Learning Rate = 0.001
# mu = [0.3, 0.6, 0.9]
cgp_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,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True
     )


cgp_plus.parse_expressions()
cgp_plus.display_one_neuron_network()
training_data = cgp_plus.gen_training_data()
```

```
mu_values = [0.3, 0.6, 0.9]

plt.figure()
for mu in mu_values:
    loss = cgp_plus.run_training_loop_one_neuron_model(training_data, mu)
    plt.plot(np.arange(len(loss))*100, loss, label=f'$\mu$ = {mu}')

plt.title('Training one_neuron_classifier with SGD+. LR = 0.001')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Training multi_neuron_classifier with ComputationalGraphPrimer
# using optimizer as SGD with Momentum. Learning Rate = 0.001
# mu = [0.3, 0.6, 0.9]
cgp_plus_multi = 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,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
     )

cgp_plus_multi.parse_multi_layer_expressions()
cgp_plus_multi.display_multi_neuron_network()
training_data = cgp_plus_multi.gen_training_data()

plt.figure()
for mu in mu_values:
    loss = cgp_plus_multi.run_training_loop_multi_neuron_model(training_data, mu)
    plt.plot(np.arange(len(loss))*100, loss, label=f'$\mu$ = {mu}')

plt.title('Training multi_neuron_classifier with SGD+. LR = 0.001')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plotting One Neuron Classifier Losses with SGD
plt.figure()
x = np.arange(len(one_neuron_classifier_running_loss))*100
plt.plot(x, one_neuron_classifier_running_loss, label='lr = 0.001')
plt.plot(x, one_neuron_classifier_running_loss_2, label='lr = 0.005')
plt.title('Training one_neuron_classifier with SGD')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plotting One Neuron Classifier Losses with SGD+
plt.figure()
x = np.arange(len(one_neuron_classifier_sgdp_running_loss))*100
plt.plot(x, one_neuron_classifier_sgdp_running_loss, label='lr = 0.001')
plt.plot(x, one_neuron_classifier_sgdp_running_loss_2, label='lr = 0.005')
plt.title('Training one_neuron_classifier with SGD+')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plotting One Neuron Classifier Losses with Adam
plt.figure()
x = np.arange(len(one_neuron_classifier_adam_running_loss))*100
plt.plot(x, one_neuron_classifier_adam_running_loss, label='lr = 0.001')
plt.plot(x, one_neuron_classifier_adam_running_loss_2, label='lr = 0.005')
plt.title('Training one_neuron_classifier with Adam')
plt.xlabel('Iterations')
plt.ylabel('Loss')
```

```
plt.legend()
plt.show()

# Plotting Multi Neuron Classifier Losses with SGD
plt.figure()
x = np.arange(len(multi_neuron_classifier_running_loss))*100
plt.plot(x, multi_neuron_classifier_running_loss, label='lr = 0.001')
plt.plot(x, multi_neuron_classifier_running_loss_2, label='lr = 0.005')
plt.title('Training multi_neuron_classifier with SGD')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plotting Multi Neuron Classifier Losses with SGD+
plt.figure()
x = np.arange(len(multi_neuron_classifier_sgdp_running_loss))*100
plt.plot(x, multi_neuron_classifier_sgdp_running_loss, label='lr = 0.001')
plt.plot(x, multi_neuron_classifier_sgdp_running_loss_2, label='lr = 0.005')
plt.title('Training multi_neuron_classifier with SGD+')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()

# Plotting Multi Neuron Classifier Losses with Adam
plt.figure()
x = np.arange(len(multi_neuron_classifier_adam_running_loss))*100
plt.plot(x, multi_neuron_classifier_adam_running_loss, label='lr = 0.001')
plt.plot(x, multi_neuron_classifier_adam_running_loss_2, label='lr = 0.005')
plt.title('Training multi_neuron_classifier with Adam')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.legend()
plt.show()
```