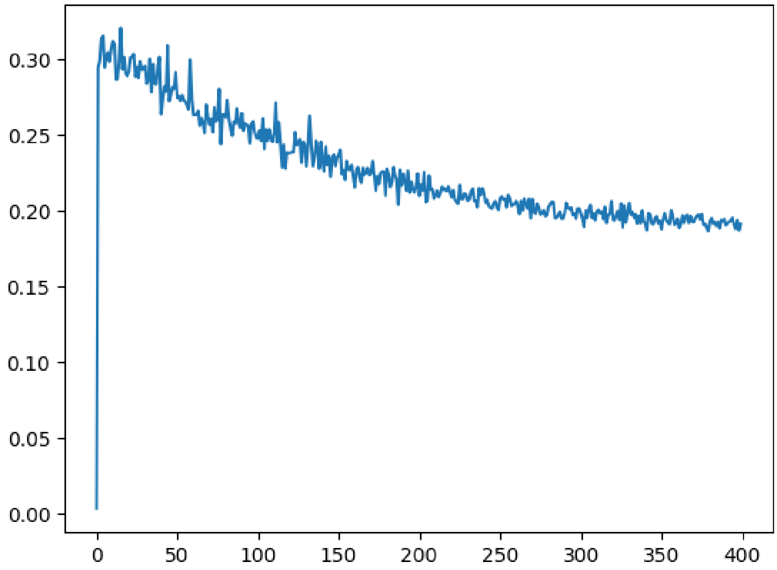
HOMEWORK 3 ECE60146

ALEXANDRE OLIVÉ PELLICER

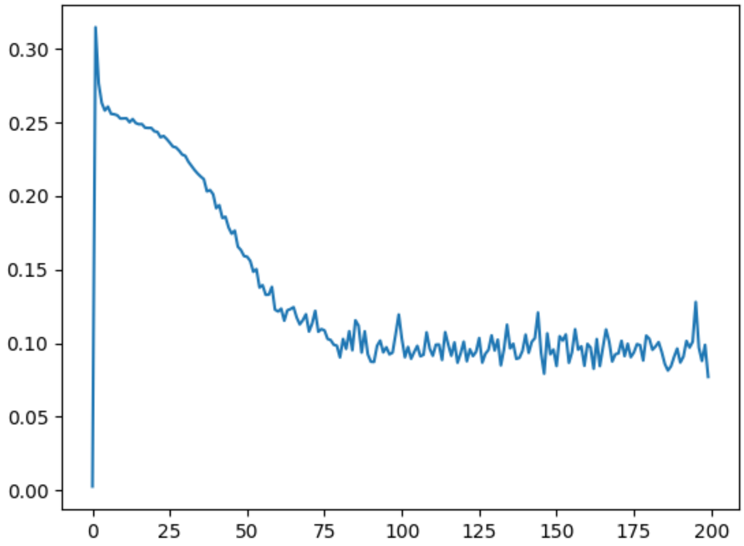
2. Becoming Familiar with the Premier

2.2

Output one_neuron_classifier.py:

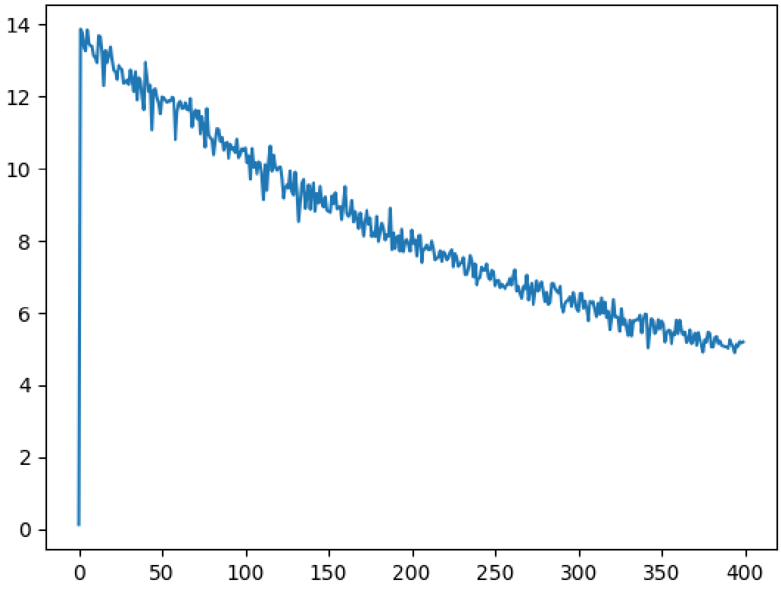


Output multi_neuron_classifier.py:



2.3

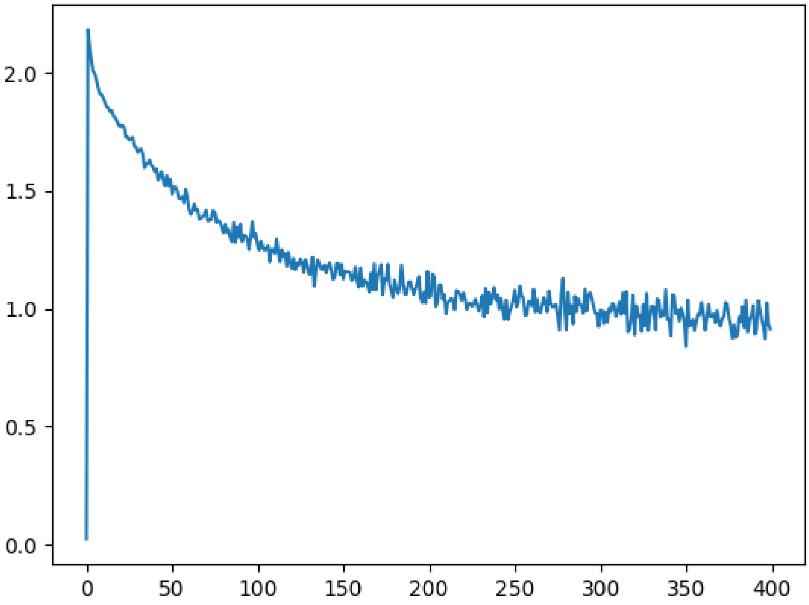
Output verify_with_torchnn.py comparing to multi_neuron:



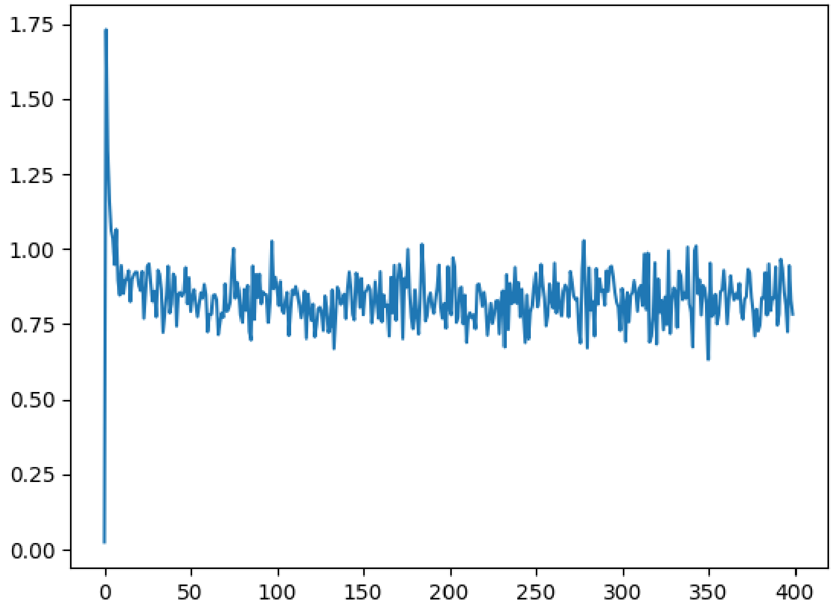
Comparing the obtained graph with the second graph obtained in 2.2 we see that the performance we are obtaining with torch.nn is worse than the performance obtained with the handcrafted network. It is important to mention that the random initialization of the weights affects the performance of the training.

2.4

Output verify_with_torchnn.py comparing to one_neuron learning_rate = 1e-3:



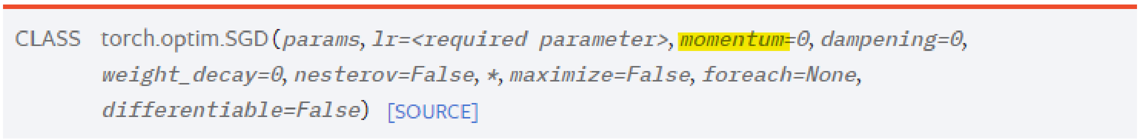
Output verify_with_torchnn.py comparing to one_neuron learning_rate = 5 * 1e-2:



Comparing both obtained graph with the first graph obtained in 2.2 we see that the performance we are obtaining with torch.nn is again worse than the performance obtained with the handcrafted network. It is important to mention that the random initialization of the weights affects the performance of the training.

2.5

The variable I think that corresponds to μ in the torch implementation of SGD is the variable called momentum:



```
CLASS  torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0,
       weight_decay=0, nesterov=False, *, maximize=False, foreach=None,
       differentiable=False)  [SOURCE]
```

3. Programming Tasks

Description of SGD+

SGD+, also referred to as SGD with momentum, is an extension of the basic SGD method with the addition of momentum. Momentum is introduced to enhance the optimization process by adding a fraction of the previous update to the current update. This helps to smooth out fluctuations in the optimization process and improves convergence.

The update formula is expressed as follows:

$$v_{t+1} = \mu * v_t + g_{t+1}$$

$$p_{t+1} = p_t - lr * v_{t+1}$$

Where g_{t+1} represents the gradient of the Loss function with respect to the parameters and μ and lr represent the momentum scalar and learning rate respectively.

Description of Adam

Adam is an optimization algorithm that combines ideas from both momentum-based methods and RMSprop. It is known for its efficiency and effectiveness in training deep neural networks.

In Adam, the learning rates are adapted for each parameter individually, based on the first-order moments and second-order moments of the gradients. This adaptability allows Adam to handle sparse gradients.

The update formula is expressed as follows:

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

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

$$p_{t+1} = p_t - lr * \frac{\widehat{m}_{t+1}}{\sqrt{\hat{v}_{t+1} + \epsilon}}$$

$$\widehat{m}_{t+1} = \frac{m_{t+1}}{1 - {\beta_1}^{t+1}}$$

$$\hat{v}_{t+1} = \frac{v_{t+1}}{1 - {\beta_2}^{t+1}}$$

One-neuron classifier plots

Figure 1: One Neuron learning rate = 1e-1

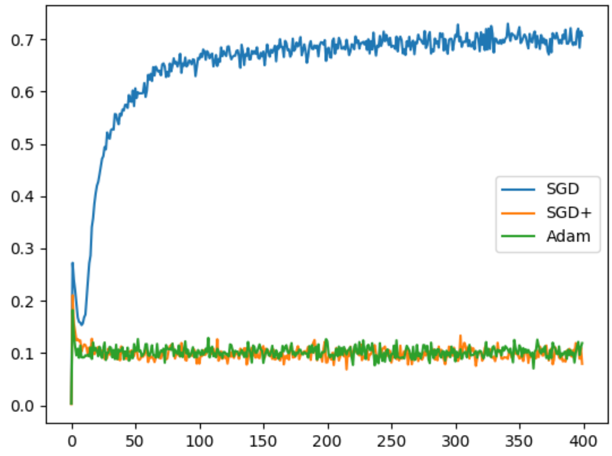


Figure 1: One Neuron learning rate = 1e-2

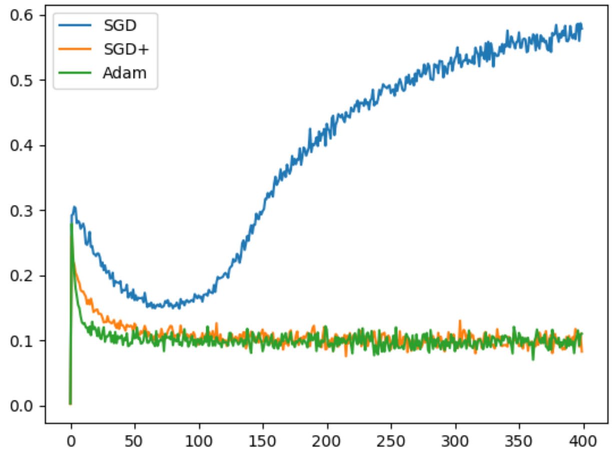


Figure 1: One Neuron learning rate = 1e-3

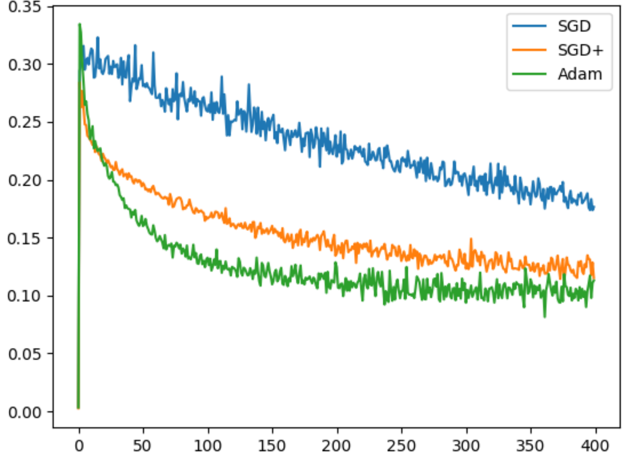
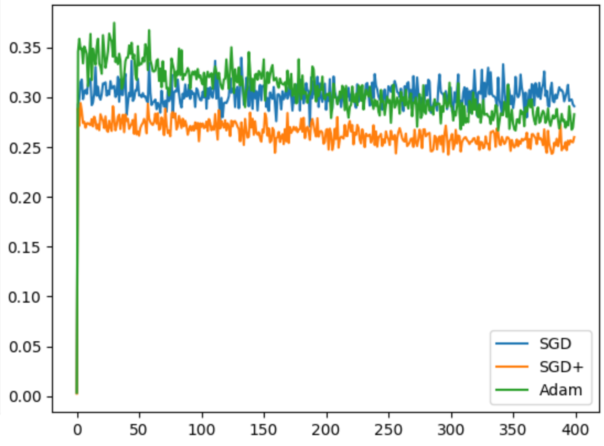


Figure 1: One Neuron learning rate = 1e-5



Observations:

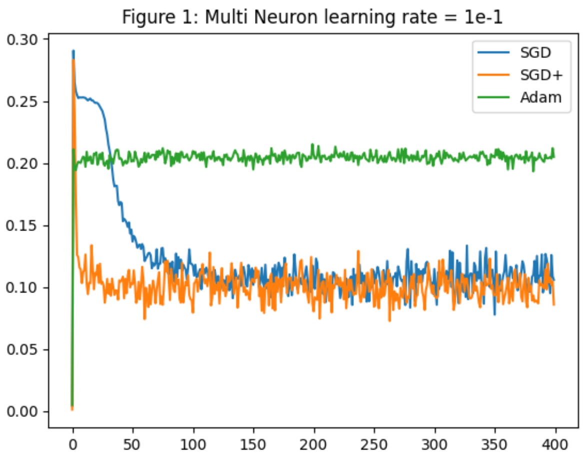
With these 3 different plots we can see the influence and the consequences of choosing a specific learning rate for training a model.

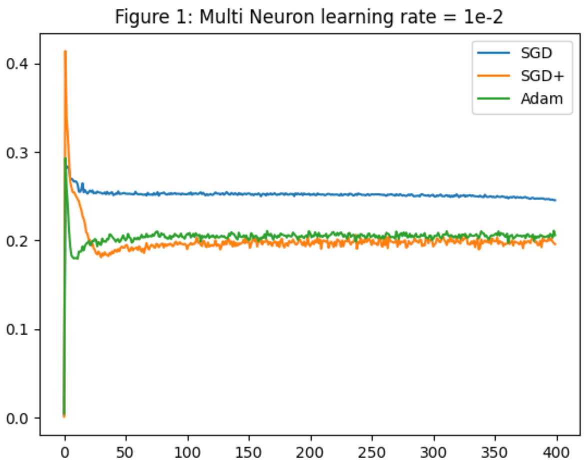
We can see that when we use a small learning rate (lr = 1e-5) the steps towards the minimum of the Loss Function are very small. Thus, in the three optimizers we can see that the loss value doesn't change a lot throughout the different iterations.

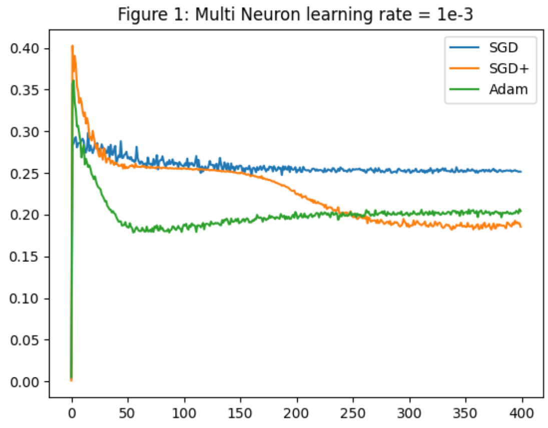
When using a learning rate of 1e-3 we can say that we achieve the desired behavior. We can see that SGD+ and Adam optimizers converge to a loss value close to 0.12. The Vanilla SGD also progresses towards this value. We can clearly see that by using SGD+ and Adam the Lost Function converges faster.

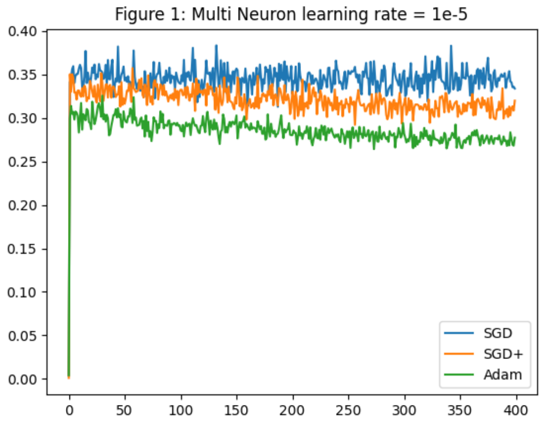
Finally, we can also see the consequences of using a high value for the learning rate (lr = 1e-1 and lr = 1e-2). We can see that the Vanillas SGD suffers an exploding loss. This is that SGD overshoots the minimum and starts oscillating or diverging.

Multi-neuron classifier plots









Observations:

From the previous representations we can obtain the following conclusions:

We see that the behavior when using the lr=1e-5 is the same as the one obtained when working with one single neuron.

When using lr=1e-1 we can see that SGD and SGD+ converge towards a value around 0.10. Nevertheless, the Adam optimizer diverges.

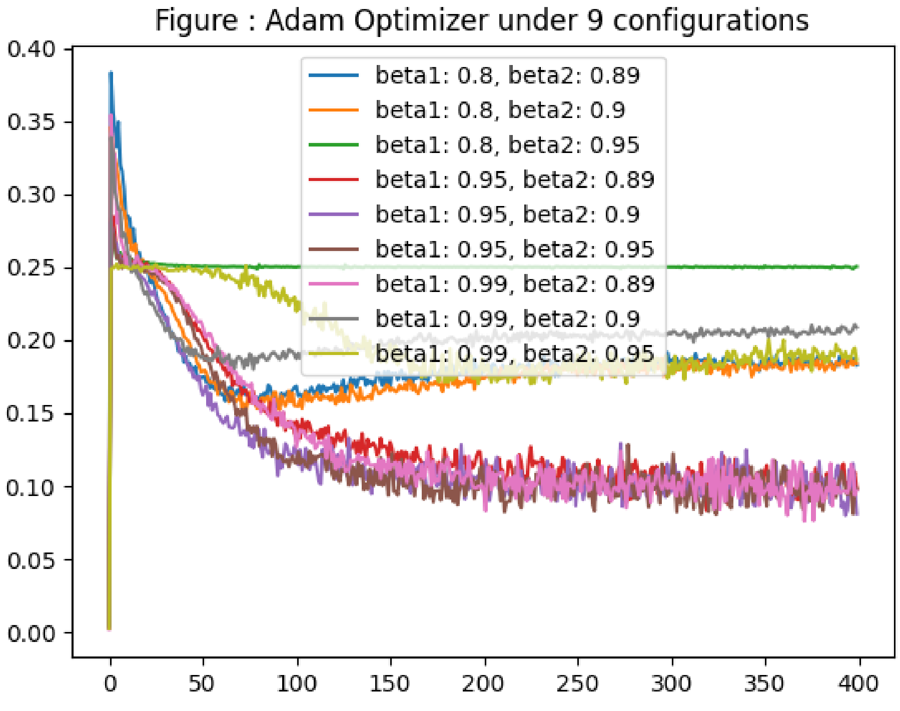
When using the lr=1e-2 and lr=1e-3 we see that the 3 optimizers converge to a Loss which is around 0.2. In both cases, we can see that both SGD+ and Adam outperform the Vanilla SGD being SGD+ the one that offers a better performance.

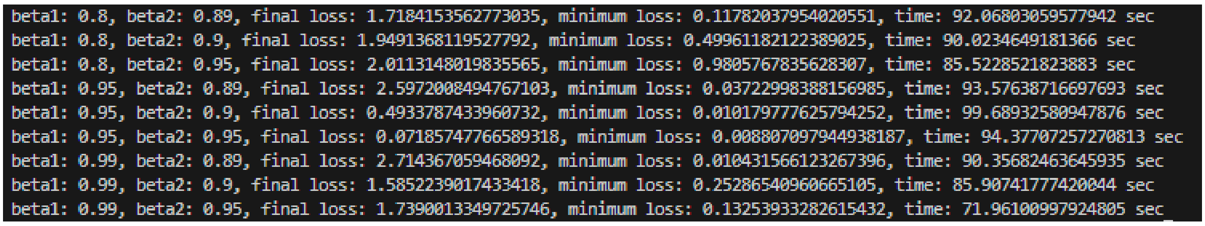
Discussion comparing the performance of the 3 optimizers

From the graphs obtained before we can obtain the following conclusions:

- The 3 optimizers are very sensitive to the value of their hyperparameters. By changing the values of the hyperparameters, the performance can vary a lot.
- When using common values for the learning rates (lr = 0.01 or lr = 0.001) the optimizers SGD+ and Adam always outperform the Vanilla SGD.
- Using learning rates too big or too smalls led to non-desired performances:
    - When the learning rate is very small (1e-5), in the 3 cases the method doesn't converge to a minimum.
    - When the learning rate is very big (1e-1) we have seen that for the one neuron case the Vanilla SGD diverges and for the multi neuron case, the Adam diverges. Nevertheless, the other optimizers converge.

Discussion comparing the performance of the Adam optimizer under the 9 configurations





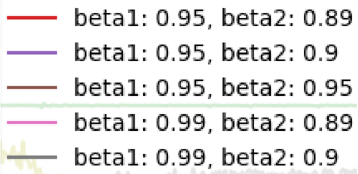
```
beta1: 0.8, beta2: 0.89, final loss: 1.7184153562773035, minimum loss: 0.11782037954020551, time: 92.06803059577942 sec
beta1: 0.8, beta2: 0.9, final loss: 1.9491368119527792, minimum loss: 0.49961182122389025, time: 90.0234649181366 sec
beta1: 0.8, beta2: 0.95, final loss: 2.0113148019835565, minimum loss: 0.9805767835628307, time: 85.5228521823883 sec
beta1: 0.95, beta2: 0.89, final loss: 2.5972008494767103, minimum loss: 0.03722998388156985, time: 93.57638716697693 sec
beta1: 0.95, beta2: 0.9, final loss: 0.4933787433960732, minimum loss: 0.010179777625794252, time: 99.68932580947876 sec
beta1: 0.95, beta2: 0.95, final loss: 0.07185747766589318, minimum loss: 0.008807097944938187, time: 94.37707257270813 sec
beta1: 0.99, beta2: 0.89, final loss: 2.714367059468092, minimum loss: 0.010431566123267396, time: 90.35682463645935 sec
beta1: 0.99, beta2: 0.9, final loss: 1.5852239017433418, minimum loss: 0.25286540960665105, time: 85.90741777420044 sec
beta1: 0.99, beta2: 0.95, final loss: 1.7390013349725746, minimum loss: 0.13253933282615432, time: 71.96100997924805 sec_
```

The experiments have been run using:

- Learning rate = 1e-3
- Training iterations = 40000
- Batch size = 8

From the obtained results we can conclude that:

- Again, we can see that the performance of the Adam optimizer is very sensitive to the value of its hyperparameters. We see that changing the value of the betas from 0.8 to 0.99 can lead to a big difference in the performance. Nevertheless, it looks like there are some combinations that offer a similar performance.
- From the obtained results, we see that the best performance is obtained when the values of beta1 and beta2 is the following:



- We can see that there are only 20 seconds of difference between the combination that took more time to train than the one that took less time.

Source code

3 OPTIMIZERS FOR THE ONE NEURON CLASSIFIER

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

# Code used to evaluate the 3 optimizers for one neuron

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 *

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

        self.bias = random.uniform(0,1)                   ## Adding the
bias improves class discrimination.
                                                          ##   We
initialize it to a random number.

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if
you first understand how
            the training dataset is created.  Search for the following
function in this file:

                             gen_training_data(self)
            As you will see in the implementation code for this method,
the training dataset
            consists of a Python dict with two keys, 0 and 1, the former
points to a list of
            all Class 0 samples and the latter to a list of all Class 1
samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian
distribution.  The two
            classes have different means and variances.  The
dimensionality of each data sample
            is set by the number of nodes in the input layer of the
neural network.

            The data loader's job is to construct a batch of samples
drawn randomly from the two
            lists mentioned above.  And it mush also associate the 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]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in
self.training_data[1]]   ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) +
len(self.training_data[1])

            def _getitem(self):
                cointoss =
random.choice([0,1])                            ## When a batch is
created by getbatch(), we want the

 ##   samples to be chosen randomly from the two lists
                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 =
[],[]                            ## First list for samples, the second
for labels
```

```
                maxval =
0.0                                                ## For approximate
batch data normalization
                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]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch


        data_loader = DataLoader(training_data,
batch_size=self.batch_size)
        loss_running_record_1 = []
        i = 0
        avg_loss_over_iterations =
0.0                                    ##  Average the loss over
iterations for printing out

 ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples_in_batch = data[0]
            class_labels_in_batch = data[1]
            y_preds, deriv_sigmoids
=  self.forward_prop_one_neuron_model(data_tuples_in_batch)     ##  FORWA
RD PROP of data
            loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
for i in range(len(class_labels_in_batch))])  ##  Find loss
            avg_loss_over_iterations += loss /
float(len(class_labels_in_batch))
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record_1.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1,
avg_loss_over_iterations))                 ## Display average loss
                avg_loss_over_iterations =
0.0                                                      ## Re-initialize
avg loss
            y_errors_in_batch = list(map(operator.sub,
class_labels_in_batch, y_preds))
            self.backprop_and_update_params_one_neuron_model(data_tuples_
in_batch, y_preds, y_errors_in_batch, deriv_sigmoids)  ## BACKPROP loss
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()
        return loss_running_record_1

class SGDPlusComputationalGraphPrimer (ComputationalGraphPrimer):
    ##MODIFIED CODE---------------------------------------------
    #Initialize new atributes
    def __init__(self, u=0, *args, **kwargs):
        self.u = u
        super().__init__(*args, **kwargs)
    ##END MODIFIED CODE---------------------------------------------

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

        self.bias = random.uniform(0,1)                   ## Adding the
bias improves class discrimination.
                                                          ##   We
initialize it to a random number.
        ##MODIFIED CODE---------------------------------------------
        #We initialize auxiliar variables used to update
parameters
        self.v_t = {param: 0 for param in self.learnable_params}
        self.bias_v_t = 0
        ##END MODIFIED CODE-----------------------------------------

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if
you first understand how
            the training dataset is created.  Search for the following
function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method,
the training dataset
            consists of a Python dict with two keys, 0 and 1, the former
points to a list of
```

```
            all Class 0 samples and the latter to a list of all Class 1
samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian
distribution.  The two
            classes have different means and variances.  The
dimensionality of each data sample
            is set by the number of nodes in the input layer of the
neural network.

            The data loader's job is to construct a batch of samples
drawn randomly from the two
            lists mentioned above.  And it mush also associate the 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]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in
self.training_data[1]]   ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) +
len(self.training_data[1])

            def _getitem(self):
                cointoss =
random.choice([0,1])                            ## When a batch is
created by getbatch(), we want the

 ##   samples to be chosen randomly from the two lists
                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 =
[],[]                            ## First list for samples, the second
for labels
                maxval =
0.0                                               ## For approximate
batch data normalization
                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]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch


        data_loader = DataLoader(training_data,
batch_size=self.batch_size)
        loss_running_record_2 = []
        i = 0
        avg_loss_over_iterations =
0.0                                    ##  Average the loss over
iterations for printing out

 ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples_in_batch = data[0]
            class_labels_in_batch = data[1]
            y_preds, deriv_sigmoids
=  self.forward_prop_one_neuron_model(data_tuples_in_batch)     ##  FORWA
RD PROP of data
            loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
for i in range(len(class_labels_in_batch))])  ##  Find loss
            avg_loss_over_iterations += loss /
float(len(class_labels_in_batch))
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record_2.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1,
avg_loss_over_iterations))                 ## Display average loss
                avg_loss_over_iterations =
0.0                                                     ## Re-initialize
avg loss
            y_errors_in_batch = list(map(operator.sub,
class_labels_in_batch, y_preds))
            self.backprop_and_update_params_one_neuron_model(data_tuples_
in_batch, y_preds, y_errors_in_batch, deriv_sigmoids)  ## BACKPROP loss
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()
        return loss_running_record_2

    def backprop_and_update_params_one_neuron_model(self,
data_tuples_in_batch, predictions, y_errors_in_batch, deriv_sigmoids):
        """
        This function implements the equations shown on Slide 61 of my
Week 3 presentation in our DL
        class at Purdue.  All four parameters defined above are lists of
what was either supplied to the
        forward prop function or calculated by it for each training data
sample in a batch.
        """
        input_vars = self.independent_vars
        input_vars_to_param_map =
self.var_to_var_param[self.output_vars[0]]                  ## These two
statements align the
        param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()}   ##   the input vars
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## For each param, sum the partials from every training data
sample in batch
            partial_of_loss_wrt_param = 0.0
            for j in range(self.batch_size):
                vals_for_input_vars_dict =  dict(zip(input_vars,
list(data_tuples_in_batch[j])))
                partial_of_loss_wrt_param   +=   -  y_errors_in_batch[j]
* vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[j]
            partial_of_loss_wrt_param /=  float(self.batch_size)

            ##MODIFIED CODE-------------------------------------------------
            #Apply SGD+ formulas
            #NESTEROV
            v_t_plus_1 = self.u * self.v_t[param] +
partial_of_loss_wrt_param
            step = self.learning_rate * v_t_plus_1
            self.vals_for_learnable_params[param] -= step
            self.v_t[param] = v_t_plus_1

        partial_of_loss_wrt_param=0.0
        for j in range(self.batch_size):
            #We keep the consistency of using a negative value for the
partial_of_loss_wrt_param as given in the initial code for the weights
            partial_of_loss_wrt_param   +=   -  y_errors_in_batch[j] *
deriv_sigmoids[j]
        partial_of_loss_wrt_param /=  float(self.batch_size)

        #Apply SGD+ formulas
        #NESTEROV
        bias_v_t_plus_1 = self.u * self.bias_v_t +
partial_of_loss_wrt_param
        step_bias = self.learning_rate * bias_v_t_plus_1
        self.bias -= step_bias
        self.bias_v_t = bias_v_t_plus_1
        ##END MODIFIED CODE---------------------------------------------

class AdamComputationalGraphPrimer (ComputationalGraphPrimer):
    ##MODIFIED CODE---------------------------------------------
    #Initialize new atributes
    def __init__(self, epsilon, beta1=0, beta2=0, *args, **kwargs):
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = epsilon
        super().__init__(*args, **kwargs)
    ##END MODIFIED CODE---------------------------------------------


    def backprop_and_update_params_one_neuron_model(self, data_tuples_in_batch, predictions, y_errors_in_batch, deriv_sigmoids, t):
        """
        This function implements the equations shown on Slide 61 of my Week 3 presentation in our DL
        class at Purdue.  All four parameters defined above are lists of what was either supplied to the
        forward prop function or calculated by it for each training data sample in a batch.
        """
        input_vars = self.independent_vars
        input_vars_to_param_map = self.var_to_var_param[self.output_vars[0]]                  ## These two statements align the
        param_to_vars_map = {param : var for var, param in input_vars_to_param_map.items()}   ##   the input vars
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## For each param, sum the partials from every training data sample in batch
            partial_of_loss_wrt_param = 0.0
            for j in range(self.batch_size):
                vals_for_input_vars_dict =  dict(zip(input_vars, list(data_tuples_in_batch[j])))
                partial_of_loss_wrt_param   +=   -  y_errors_in_batch[j] * vals_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[j]
            partial_of_loss_wrt_param /=  float(self.batch_size)

            ##MODIFIED CODE---------------------------------------------
            #Apply Adam formulas
            m_t_plus_1 = self.beta1 * self.m_t[param] + (1 - self.beta1) * partial_of_loss_wrt_param
            v_t_plus_1 = self.beta2 * self.v_t[param] + (1 - self.beta2) * (partial_of_loss_wrt_param**2)

            m_t_plus_1_hat = m_t_plus_1 / (1 - self.beta1**t)
            v_t_plus_1_hat = v_t_plus_1 / (1 - self.beta2**t)

            step = self.learning_rate * (m_t_plus_1_hat /
np.sqrt(v_t_plus_1_hat + self.epsilon))
            self.vals_for_learnable_params[param] -= step

            self.v_t[param] = v_t_plus_1
            self.m_t[param] = m_t_plus_1

        partial_of_loss_wrt_param=0.0
        for j in range(self.batch_size):
            #We keep the consistency of using a negative value for the
partial_of_loss_wrt_param as given in the initial code for the weights
            partial_of_loss_wrt_param   +=   -  y_errors_in_batch[j] *
deriv_sigmoids[j]
        partial_of_loss_wrt_param /=  float(self.batch_size)

        #Apply Adam formulas
        bias_m_t_plus_1 = self.beta1 * self.bias_m_t + (1 - self.beta1) *
(partial_of_loss_wrt_param)
        bias_v_t_plus_1 = self.beta2 * self.bias_v_t + (1 - self.beta2) *
(partial_of_loss_wrt_param**2)

        bias_m_t_plus_1_hat = bias_m_t_plus_1 / (1 - self.beta1**t)
        bias_v_t_plus_1_hat = bias_v_t_plus_1 / (1 - self.beta2**t)

        bias_step = self.learning_rate * (bias_m_t_plus_1_hat /
np.sqrt(bias_v_t_plus_1_hat + self.epsilon))

        self.bias -= bias_step

        self.bias_v_t = bias_v_t_plus_1
        self.bias_m_t = bias_m_t_plus_1
        ##END MODIFIED CODE---------------------------------------------

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

```
        self.bias = random.uniform(0,1)                   ## Adding the
bias improves class discrimination.
                                                          ##    We
initialize it to a random number.
        ##MODIFIED CODE---------------------------------------------
        #We initialize auxiliar variables used to update parameters
        self.m_t = {param: 0 for param in self.learnable_params}
        self.v_t = {param: 0 for param in self.learnable_params}
        self.bias_m_t = 0
        self.bias_v_t = 0
        ##END MODIFIED CODE-----------------------------------------

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if
you first understand how
            the training dataset is created.  Search for the following
function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method,
the training dataset
            consists of a Python dict with two keys, 0 and 1, the former
points to a list of
            all Class 0 samples and the latter to a list of all Class 1
samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian
distribution.  The two
            classes have different means and variances.  The
dimensionality of each data sample
            is set by the number of nodes in the input layer of the
neural network.

            The data loader's job is to construct a batch of samples
drawn randomly from the two
            lists mentioned above.  And it mush also associate the 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]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in
self.training_data[1]]   ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) +
len(self.training_data[1])

            def _getitem(self):
                cointoss =
random.choice([0,1])                            ## When a batch is
created by getbatch(), we want the

 ##   samples to be chosen randomly from the two lists
                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 =
[],[]                            ## First list for samples, the second
for labels
                maxval =
0.0                                               ## For approximate
batch data normalization
                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]          ## Normalize 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                                    ##  Average the loss over
iterations for printing out

 ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples_in_batch = data[0]
            class_labels_in_batch = data[1]
            y_preds, deriv_sigmoids
=  self.forward_prop_one_neuron_model(data_tuples_in_batch)     ##  FORWA
RD PROP of data
            loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
for i in range(len(class_labels_in_batch))])  ##  Find loss
            avg_loss_over_iterations += loss /
float(len(class_labels_in_batch))
            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))                 ## Display average loss
                avg_loss_over_iterations =
0.0                                                     ## Re-initialize
avg loss
            y_errors_in_batch = list(map(operator.sub,
class_labels_in_batch, y_preds))

            ### CODE MODIFIED--------------------------------------------
--------
            #We pass the iteration useful to update the parameters with
Adam
            self.backprop_and_update_params_one_neuron_model(data_tuples_
in_batch, y_preds, y_errors_in_batch, deriv_sigmoids, t = i+1)  ##
BACKPROP loss
            ### END CODE MODIFIED----------------------------------------
------------

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

#Run the training using the 3 optimizers and plotting the results
cgp = NewComputationalGraphPrimer(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

cgp.parse_expressions()
training_data_cgp = cgp.gen_training_data()
SGD_loss = cgp.run_training_loop_one_neuron_model( training_data_cgp )
```

```
cgp_SGDPlus = SGDPlusComputationalGraphPrimer(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
               u = 0.9
      )

cgp_SGDPlus.parse_expressions()
training_data_cgp_SGDPlus = cgp_SGDPlus.gen_training_data()
SGD_plus_loss = cgp_SGDPlus.run_training_loop_one_neuron_model(
training_data_cgp_SGDPlus )

cgp_Adam = AdamComputationalGraphPrimer(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
               beta1 = 0.9,
               beta2 = 0.99,
               epsilon = 1e-8
      )

cgp_Adam.parse_expressions()
training_data_cgp_Adam = cgp_Adam.gen_training_data()
SGD_adam_loss = cgp_Adam.run_training_loop_one_neuron_model(
training_data_cgp_Adam )


plt.figure()
plt.plot(SGD_loss)
plt.plot(SGD_plus_loss)
plt.plot(SGD_adam_loss)
plt.legend(["SGD","SGD+","Adam"])
plt.title("Figure 1: One Neuron learning rate = 1e-2")
plt.show()
```

3 OPTIMIZERS FOR THE MULTI NEURON CLASSIFIER

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

# Code used to evaluate the 3 optimizers for multi neuron
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 *

class New_Multi_ComputationalGraphPrimer (ComputationalGraphPrimer):
    #We override this method for the plot
    def run_training_loop_multi_neuron_model(self, training_data):

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if
you first understand how
            the training dataset is created.  Search for the following
function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method,
the training dataset
            consists of a Python dict with two keys, 0 and 1, the former
points to a list of
            all Class 0 samples and the latter to a list of all Class 1
samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian
distribution.  The two
            classes have different means and variances.  The
dimensionality of each data sample
            is set by the number of nodes in the input layer of the
neural network.

            The data loader's job is to construct a batch of samples
drawn randomly from the two
            lists mentioned above.  And it mush also associate the 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]]    ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in
self.training_data[1]]    ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) +
len(self.training_data[1])

            def _getitem(self):
                cointoss =
random.choice([0,1])                            ## When a batch is
created by getbatch(), we want the

 ##   samples to be chosen randomly from the two lists
                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 =
[],[]                            ## First list for samples, the second
for labels
                maxval =
0.0                                               ## For approximate
batch data normalization
                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]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

        ##  The training loop must first initialize the learnable
parameters.  Remember, these are the
        ##  symbolic names in your input expressions for the neural layer
that do not begin with the
        ##  letter 'x'.  In this case, we are initializing with random
numbers from a uniform distribution
        ##  over the interval (0,1):
```

```
        self.vals_for_learnable_params = {param: random.uniform(0,1) for
param in self.learnable_params}
        ##  In the same  manner, we must also initialize the biases at
each node that aggregates forward
        ##  propagating data:
        self.bias =   {i : [random.uniform(0,1) for j in range(
self.layers_config[i] ) ]  for i in range(1, self.num_layers)}
        data_loader = DataLoader(training_data,
batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations =
0.0                                          ##  Average the loss over
iterations for printing out

      ##    every N iterations during the training loop.
        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)
                          ## FORW PROP works by side-effect
            predicted_labels_for_batch =
self.forw_prop_vals_at_layers[self.num_layers-1]           ## Predictions
from FORW PROP
            y_preds =  [item for sublist
in  predicted_labels_for_batch  for item in sublist]       ## Get numeric
vals for predictions
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss /
float(len(class_labels))                                              ##
Average the loss over batch
            avg_loss_over_iterations +=
loss_avg                                                    ## Add to
Average loss over iterations
            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))                  ## Display avg loss
                avg_loss_over_iterations =
0.0                                                      ## Re-initialize
avg-over-iterations loss
            y_errors_in_batch = list(map(operator.sub, class_labels,
y_preds))
            self.backprop_and_update_params_multi_neuron_model(y_preds,
y_errors_in_batch)
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()
        return loss_running_record

class SGDPlus_Multi_ComputationalGraphPrimer(ComputationalGraphPrimer):
    ##MODIFIED CODE---------------------------------------------
    #Initialize new atributes
      def __init__(self, u=0, *args, **kwargs):
            self.u = u
            super().__init__(*args, **kwargs)
    ##END MODIFIED CODE---------------------------------------------

      def backprop_and_update_params_multi_neuron_model(self,
predictions, y_errors):
        """
        First note that loop index variable 'back_layer_index' starts
with the index of
        the last layer.  For the 3-layer example shown for 'forward',
back_layer_index
        starts with a value of 2, its next value is 1, and that's it.

        In the code below, the outermost loop is over the data samples in
a batch. As shown
        on Slide 73 of my Week 3 lecture, in order to calculate the
partials of Loss with
        respect to the learnable params, we need to backprop the
prediction errors and
        the gradients of the Sigmoid.  For the purpose of satisfying the
requirements of
        SGD, the backprop of the prediction errors and the gradients
needs to be carried
        out separately for each training data sample in a batch.  That's
what the outer
        loop is for.

        After we exit the outermost loop, we average over the results
obtained from each
        training data sample in a batch.

        Pay attention to the variable 'vars_in_layer'.  These store the
node variables in
        the current layer during backpropagation.
        """
        ## Eq. (24) on Slide 73 of my Week 3 lecture says we need to
store backproped errors in each layer leading up to the last:
        pred_err_backproped_at_layers =   [ {i : [None for j in range(
self.layers_config[i] ) ]
                                                                  for i
in range(self.num_layers)} for _ in range(self.batch_size) ]
```

```
        ## This will store "\delta L / \delta w" you see at the LHS of
the equations on Slide 73:
        partial_of_loss_wrt_params = {param : 0.0 for param in
self.all_params}
        ## For estimating the changes to the bias to be made on the basis
of the derivatives of the Sigmoids:
        bias_changes =   {i : [0.0 for j in range( self.layers_config[i]
) ]  for i in range(1, self.num_layers)}
        for b in range(self.batch_size):
            pred_err_backproped_at_layers[b][self.num_layers - 1] = [
y_errors[b] ]
            for back_layer_index in
reversed(range(1,self.num_layers)):             ## For the 3-layer
network, the first val for back_layer_index is 2 for the 3rd layer
                input_vals =
self.forw_prop_vals_at_layers[back_layer_index -1]     ## This is a list
of 8 two-element lists  --- since we have two nodes in the 2nd layer
                deriv_sigmoids
=  self.gradient_vals_for_layers[back_layer_index]   ## This is a list
eight one-element lists, one for each batch element
                vars_in_layer  =  self.layer_vars[back_layer_index]
          ## A list like ['xo']
                vars_in_next_layer_back  =  self.layer_vars[back_layer_in
dex - 1]   ## A list like ['xw', 'xz']

                vals_for_input_vars_dict
=  dict(zip(vars_in_next_layer_back,
self.forw_prop_vals_at_layers[back_layer_index - 1][b]))
                ## For the next statement, 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"

                layer_params =
self.layer_params[back_layer_index]
                transposed_layer_params =
list(zip(*layer_params))                  ## Creating a transpose of the
link matrix, See Eq. 30 on Slide 77

                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        pred_err_backproped_at_layers[b][back_layer_index
- 1][k] =
sum([self.vals_for_learnable_params[transposed_layer_params[k][i]]

             * pred_err_backproped_at_layers[b][back_layer_index][i]
```

```
                                        for i in
range(len(vars_in_layer))])
                for j,var in enumerate(vars_in_layer):
                    layer_params =
self.layer_params[back_layer_index][j]           ##  ['cp', 'cq']   for
the end layer
                    input_vars_to_param_map =
self.var_to_var_param[var]            ## These two statements align
the    {'xw': 'cp', 'xz': 'cq'}
                    param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()}   ##   and the input vars   {'cp': 'xw',
'cq': 'xz'}

                    ##  Update the partials of Loss wrt to the learnable
parameters between the current layer
                    ##  and the previous layer. You are accumulating
these partials over the different training
                    ##  data samples in the batch being processed.  For
each training data sample, the formula
                    ##  being used is shown in Eq. (29) on Slide 77 of my
Week 3 slides:
                    for i,param in enumerate(layer_params):
                        partial_of_loss_wrt_params[param]   +=   pred_err
_backproped_at_layers[b][back_layer_index][j] * \
                                                                        v
als_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[b][j]
                ##  We will now estimate the change in the bias that
needs to be made at each node in the previous layer
                ##  from the derivatives the sigmoid at the nodes in the
current layer and the prediction error as
                ##  backproped to the previous layer nodes:
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        if back_layer_index-1 > 0:
                            bias_changes[back_layer_index-1][k] +=
pred_err_backproped_at_layers[b][back_layer_index - 1][k] *
deriv_sigmoids[b][j]

        ##MODIFIED CODE------------------------------------------------
        ## Now update the learnable parameters.  The loop shown below
carries out SGD mandated averaging
        for param in partial_of_loss_wrt_params:
            #We keep the consistency of using a negative value for the
partial_of_loss_wrt_param as given in the initial code
            #Apply SGD+ formulas
            partial_of_loss_wrt_param = -
partial_of_loss_wrt_params[param] /  float(self.batch_size)
```

```
            v_t_plus_1 = self.u * self.v_t[param] +
partial_of_loss_wrt_param
            step = self.learning_rate * v_t_plus_1
            self.vals_for_learnable_params[param] -= step
            self.v_t[param] = v_t_plus_1

        ##  Finally we update the biases at all the nodes that aggregate
data:
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                #We keep the consistency of using a negative value for
the partial_of_loss_wrt_param as given in the initial code
                #Apply SGD+ formulas
                bias_v_t_plus_1 = self.u * self.bias_v_t[layer_index][k]
- ( bias_changes[layer_index][k] / float(self.batch_size) )
                step_bias = self.learning_rate * bias_v_t_plus_1
                self.bias[layer_index][k] -= step_bias
                self.bias_v_t[layer_index][k] = bias_v_t_plus_1
        ##END MODIFIED CODE---------------------------------------------

      def run_training_loop_multi_neuron_model(self, training_data):

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if
you first understand how
            the training dataset is created.  Search for the following
function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method,
the training dataset
            consists of a Python dict with two keys, 0 and 1, the former
points to a list of
            all Class 0 samples and the latter to a list of all Class 1
samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian
distribution.  The two
            classes have different means and variances.  The
dimensionality of each data sample
            is set by the number of nodes in the input layer of the
neural network.

            The data loader's job is to construct a batch of samples
drawn randomly from the two
            lists mentioned above.  And it mush also associate the 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]]    ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in
self.training_data[1]]    ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) +
len(self.training_data[1])

            def _getitem(self):
                cointoss =
random.choice([0,1])                            ## When a batch is
created by getbatch(), we want the

 ##   samples to be chosen randomly from the two lists
                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 =
[],[]                            ## First list for samples, the second
for labels
                maxval =
0.0                                               ## For approximate
batch data normalization
                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]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

        ##  The training loop must first initialize the learnable
parameters.  Remember, these are the
        ##  symbolic names in your input expressions for the neural layer
that do not begin with the
        ##  letter 'x'.  In this case, we are initializing with random
numbers from a uniform distribution
```

```
        ##  over the interval (0,1):
        self.vals_for_learnable_params = {param: random.uniform(0,1) for
param in self.learnable_params}
        ##  In the same  manner, we must also initialize the biases at
each node that aggregates forward
        ##  propagating data:
        self.bias =   {i : [random.uniform(0,1) for j in range(
self.layers_config[i] ) ]  for i in range(1, self.num_layers)}
        data_loader = DataLoader(training_data,
batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations =
0.0                                          ##  Average the loss over
iterations for printing out

      ##    every N iterations during the training loop.
        ##MODIFIED CODE---------------------------------------------
        #We initialize auxiliar variables used to update
parameters

        self.v_t = {param: 0 for param in self.learnable_params}
        self.bias_v_t =  {i : [0 for j in range( self.layers_config[i] )
]  for i in range(1, self.num_layers)}
        ##END MODIFIED CODE-----------------------------------------

        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)
                          ## FORW PROP works by side-effect
            predicted_labels_for_batch =
self.forw_prop_vals_at_layers[self.num_layers-1]          ## Predictions
from FORW PROP
            y_preds =  [item for sublist
in  predicted_labels_for_batch  for item in sublist]       ## Get numeric
vals for predictions
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss /
float(len(class_labels))                                            ##
Average the loss over batch
            avg_loss_over_iterations +=
loss_avg                                                   ## Add to
Average loss over iterations
            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))                  ## Display avg loss
                avg_loss_over_iterations =
0.0                                                     ## Re-initialize
avg-over-iterations loss
            y_errors_in_batch = list(map(operator.sub, class_labels,
y_preds))
            self.backprop_and_update_params_multi_neuron_model(y_preds,
y_errors_in_batch)
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()
        return loss_running_record

class Adam_Multi_ComputationalGraphPrimer(ComputationalGraphPrimer):
    ##MODIFIED CODE---------------------------------------------
    #Initialize new atributes
      def __init__(self, beta1=0, beta2=0, *args, **kwargs):
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = 1e-8
        super().__init__(*args, **kwargs)
    ##END MODIFIED CODE---------------------------------------------

      def backprop_and_update_params_multi_neuron_model(self,
predictions, y_errors, t):
        """
        First note that loop index variable 'back_layer_index' starts
with the index of
        the last layer.  For the 3-layer example shown for 'forward',
back_layer_index
        starts with a value of 2, its next value is 1, and that's it.

        In the code below, the outermost loop is over the data samples in
a batch. As shown
        on Slide 73 of my Week 3 lecture, in order to calculate the
partials of Loss with
        respect to the learnable params, we need to backprop the
prediction errors and
        the gradients of the Sigmoid.  For the purpose of satisfying the
requirements of
        SGD, the backprop of the prediction errors and the gradients
needs to be carried
        out separately for each training data sample in a batch.  That's
what the outer
        loop is for.

        After we exit the outermost loop, we average over the results
obtained from each
```

```
        training data sample in a batch.

        Pay attention to the variable 'vars_in_layer'.  These store the
node variables in
        the current layer during backpropagation.
        """
        ## Eq. (24) on Slide 73 of my Week 3 lecture says we need to
store backproped errors in each layer leading up to the last:
        pred_err_backproped_at_layers =   [ {i : [None for j in range(
self.layers_config[i] ) ]
                                                                  for i
in range(self.num_layers)} for _ in range(self.batch_size) ]
        ## This will store "\delta L / \delta w" you see at the LHS of
the equations on Slide 73:
        partial_of_loss_wrt_params = {param : 0.0 for param in
self.all_params}
        ## For estimating the changes to the bias to be made on the basis
of the derivatives of the Sigmoids:
        bias_changes =   {i : [0.0 for j in range( self.layers_config[i]
) ]  for i in range(1, self.num_layers)}
        for b in range(self.batch_size):
            pred_err_backproped_at_layers[b][self.num_layers - 1] = [
y_errors[b] ]
            for back_layer_index in
reversed(range(1,self.num_layers)):             ## For the 3-layer
network, the first val for back_layer_index is 2 for the 3rd layer
                input_vals =
self.forw_prop_vals_at_layers[back_layer_index -1]     ## This is a list
of 8 two-element lists  --- since we have two nodes in the 2nd layer
                deriv_sigmoids
=  self.gradient_vals_for_layers[back_layer_index]   ## This is a list
eight one-element lists, one for each batch element
                vars_in_layer  =  self.layer_vars[back_layer_index]
          ## A list like ['xo']
                vars_in_next_layer_back  =  self.layer_vars[back_layer_in
dex - 1]   ## A list like ['xw', 'xz']
                vals_for_input_vars_dict
=  dict(zip(vars_in_next_layer_back,
self.forw_prop_vals_at_layers[back_layer_index - 1][b]))
                ## For the next statement, 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"
                layer_params =
self.layer_params[back_layer_index]
```

```
        transposed_layer_params =
list(zip(*layer_params))                  ## Creating a transpose of the
link matrix, See Eq. 30 on Slide 77
        for k,var1 in enumerate(vars_in_next_layer_back):
            for j,var2 in enumerate(vars_in_layer):
                pred_err_backproped_at_layers[b][back_layer_index
- 1][k] =
sum([self.vals_for_learnable_params[transposed_layer_params[k][i]]

      * pred_err_backproped_at_layers[b][back_layer_index][i]

                    for i in
range(len(vars_in_layer))])
        for j,var in enumerate(vars_in_layer):
            layer_params =
self.layer_params[back_layer_index][j]          ##  ['cp', 'cq']   for
the end layer
            input_vars_to_param_map =
self.var_to_var_param[var]            ## These two statements align
the    {'xw': 'cp', 'xz': 'cq'}
            param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()}   ##   and the input vars   {'cp': 'xw',
'cq': 'xz'}

            ##  Update the partials of Loss wrt to the learnable
parameters between the current layer
            ##  and the previous layer. You are accumulating
these partials over the different training
            ##  data samples in the batch being processed.  For
each training data sample, the formula
            ##  being used is shown in Eq. (29) on Slide 77 of my
Week 3 slides:
            for i,param in enumerate(layer_params):
                partial_of_loss_wrt_params[param]   +=   pred_err
_backproped_at_layers[b][back_layer_index][j] * \
                                                                    v
als_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[b][j]
        ##  We will now estimate the change in the bias that
needs to be made at each node in the previous layer
        ##  from the derivatives the sigmoid at the nodes in the
current layer and the prediction error as
        ##  backproped to the previous layer nodes:
        for k,var1 in enumerate(vars_in_next_layer_back):
            for j,var2 in enumerate(vars_in_layer):
                if back_layer_index-1 > 0:
                    bias_changes[back_layer_index-1][k] +=
pred_err_backproped_at_layers[b][back_layer_index - 1][k] *
deriv_sigmoids[b][j]
```

```
        ##MODIFIED CODE------------------------------------------------------------
        ## Now update the learnable parameters.  The loop shown below
carries out SGD mandated averaging
        for param in partial_of_loss_wrt_params:
            #We keep the consistency of using a negative value for the
partial_of_loss_wrt_param as given in the initial code
            partial_of_loss_wrt_param = -
partial_of_loss_wrt_params[param] /  float(self.batch_size)
            #Apply Adam formulas
            m_t_plus_1 = self.beta1 * self.m_t[param] + (1 - self.beta1)
* partial_of_loss_wrt_param
            v_t_plus_1 = self.beta2 * self.v_t[param] + (1 - self.beta2)
* (partial_of_loss_wrt_param**2)

            m_t_plus_1_hat = m_t_plus_1 / (1 - self.beta1**t)
            v_t_plus_1_hat = v_t_plus_1 / (1 - self.beta2**t)

            step = self.learning_rate * (m_t_plus_1_hat /
np.sqrt(v_t_plus_1_hat + self.epsilon))
            self.vals_for_learnable_params[param] -= step

            self.v_t[param] = v_t_plus_1
            self.m_t[param] = m_t_plus_1


        ##  Finally we update the biases at all the nodes that aggregate
data:
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                #We keep the consistency of using a negative value for
the partial_of_loss_wrt_param as given in the initial code
                partial_of_loss_wrt_param = - (
bias_changes[layer_index][k] / float(self.batch_size) )
                #Apply Adam formulas
                bias_m_t_plus_1 = self.beta1 *
self.bias_m_t[layer_index][k] + (1 - self.beta1) *
(partial_of_loss_wrt_param)
                bias_v_t_plus_1 = self.beta2 *
self.bias_v_t[layer_index][k] + (1 - self.beta2) *
(partial_of_loss_wrt_param**2)

                bias_m_t_plus_1_hat = bias_m_t_plus_1 / (1 -
self.beta1**t)
                bias_v_t_plus_1_hat = bias_v_t_plus_1 / (1 -
self.beta2**t)

                bias_step = self.learning_rate * (bias_m_t_plus_1_hat /
np.sqrt(bias_v_t_plus_1_hat + self.epsilon))
```

```
                self.bias[layer_index][k] -= bias_step

                self.bias_v_t[layer_index][k] = bias_v_t_plus_1
                self.bias_m_t[layer_index][k] = bias_m_t_plus_1
        ##END MODIFIED CODE---------------------------------------------



      def run_training_loop_multi_neuron_model(self, training_data):

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if
you first understand how
            the training dataset is created.  Search for the following
function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method,
the training dataset
            consists of a Python dict with two keys, 0 and 1, the former
points to a list of
            all Class 0 samples and the latter to a list of all Class 1
samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian
distribution.  The two
            classes have different means and variances.  The
dimensionality of each data sample
            is set by the number of nodes in the input layer of the
neural network.

            The data loader's job is to construct a batch of samples
drawn randomly from the two
            lists mentioned above.  And it mush also associate the 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]]    ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in
self.training_data[1]]    ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) +
len(self.training_data[1])
```

```
      def _getitem(self):
        cointoss =
random.choice([0,1])                        ## When a batch is
created by getbatch(), we want the

 ##   samples to be chosen randomly from the two lists
        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 =
[],[]                ## First list for samples, the second
for labels
        maxval =
0.0                                   ## For approximate
batch data normalization
        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]      ## Normalize batch data
        batch = [batch_data, batch_labels]
        return batch

    ##  The training loop must first initialize the learnable
parameters.  Remember, these are the
    ##  symbolic names in your input expressions for the neural layer
that do not begin with the
    ##  letter 'x'.  In this case, we are initializing with random
numbers from a uniform distribution
    ##  over the interval (0,1):
    self.vals_for_learnable_params = {param: random.uniform(0,1) for
param in self.learnable_params}
    ##  In the same  manner, we must also initialize the biases at
each node that aggregates forward
    ##  propagating data:
    self.bias =   {i : [random.uniform(0,1) for j in range(
self.layers_config[i] ) ]  for i in range(1, self.num_layers)}
    data_loader = DataLoader(training_data,
batch_size=self.batch_size)
    loss_running_record = []
    i = 0
```

```
        avg_loss_over_iterations = 
0.0                                          ##  Average the loss over 
iterations for printing out

      ##    every N iterations during the training loop.
        ##MODIFIED CODE------------------------------------------
        #We initialize auxiliar variables used to update 
parameters

        self.m_t = {param: 0 for param in self.learnable_params}
        self.v_t = {param: 0 for param in self.learnable_params}
        self.bias_m_t = {i : [0 for j in range( self.layers_config[i] ) 
]  for i in range(1, self.num_layers)}
        self.bias_v_t = {i : [0 for j in range( self.layers_config[i] ) 
]  for i in range(1, self.num_layers)}
        ##END MODIFIED CODE--------------------------------------

        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)
                              ## FORW PROP works by side-effect
            predicted_labels_for_batch = 
self.forw_prop_vals_at_layers[self.num_layers-1]           ## Predictions 
from FORW PROP
            y_preds =  [item for sublist 
in  predicted_labels_for_batch  for item in sublist]       ## Get numeric 
vals for predictions
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in 
range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss / 
float(len(class_labels))                                              ## 
Average the loss over batch
            avg_loss_over_iterations += 
loss_avg                                                    ## Add to 
Average loss over iterations
            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))                  ## Display avg loss
                avg_loss_over_iterations = 
0.0                                                      ## Re-initialize 
avg-over-iterations loss
            y_errors_in_batch = list(map(operator.sub, class_labels, 
y_preds))
            ##MODIFIED CODE--------------------------------------------
            #We pass the iteration useful to update the parameters with
Adam
            self.backprop_and_update_params_multi_neuron_model(y_preds,
y_errors_in_batch, t=i+1)
            ##END MODIFIED CODE----------------------------------------
-
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()
        return loss_running_record

#Run the training using the 3 optimizers and plotting the results
cgp = New_Multi_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 = 5e-3,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

cgp.parse_multi_layer_expressions()
training_data_cgp = cgp.gen_training_data()
SGD_loss = cgp.run_training_loop_multi_neuron_model( training_data_cgp )


cgp_SGDPlus = SGDPlus_Multi_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 = 5e-3,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
               u=0.9
      )
```

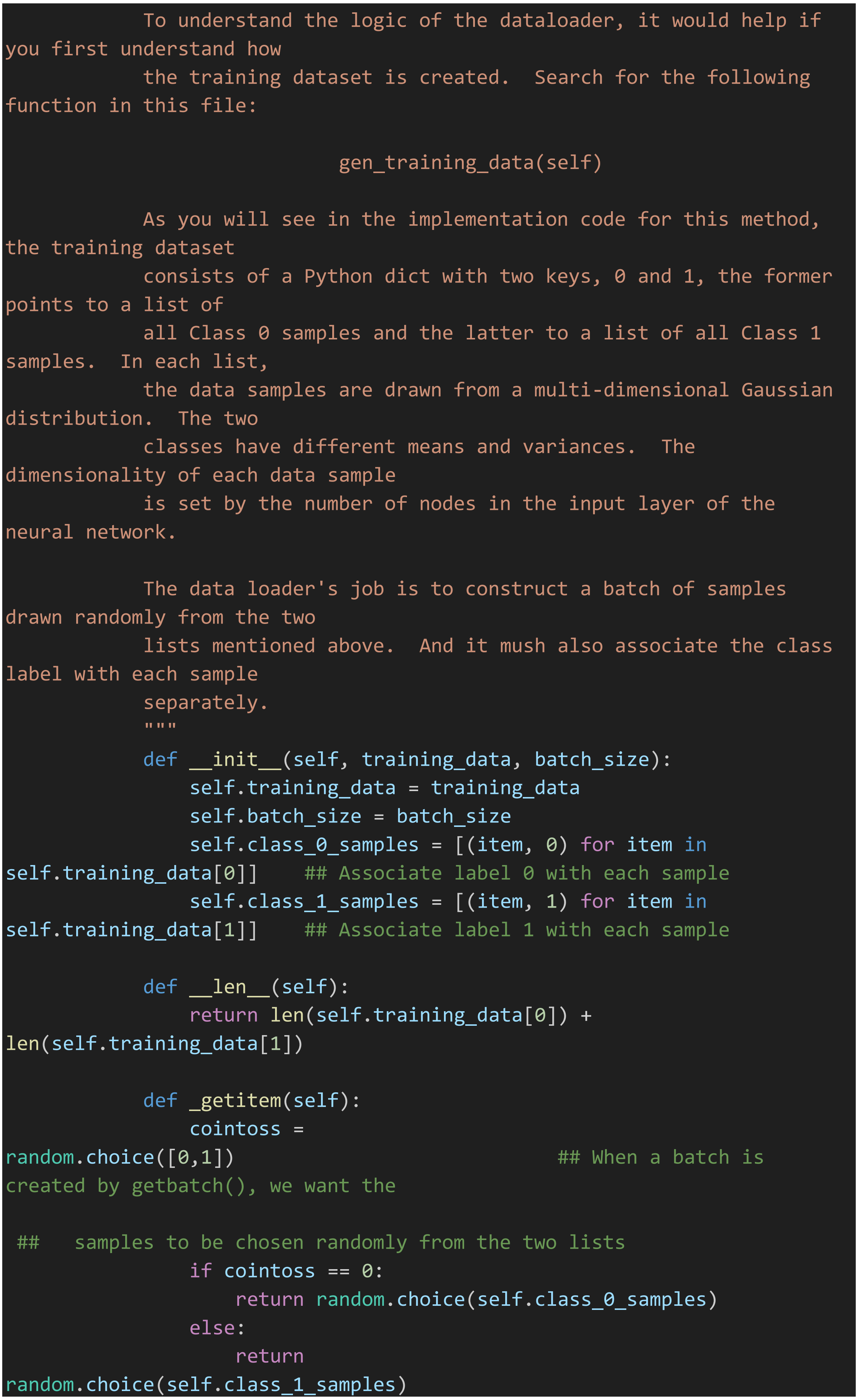
```
cgp_SGDPlus.parse_multi_layer_expressions()
training_data_cgp_SGDPlus = cgp_SGDPlus.gen_training_data()
SGD_plus_loss = cgp_SGDPlus.run_training_loop_multi_neuron_model(
training_data_cgp_SGDPlus )

cgp_Adam = Adam_Multi_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 = 5e-3,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
               beta1 = 0.9,
               beta2 = 0.99

      )

cgp_Adam.parse_multi_layer_expressions()
training_data_cgp_Adam = cgp_Adam.gen_training_data()
SGD_adam_loss = cgp_Adam.run_training_loop_multi_neuron_model(
training_data_cgp_Adam )


plt.figure()
plt.plot(SGD_loss)
plt.plot(SGD_plus_loss)
plt.plot(SGD_adam_loss)
plt.legend(["SGD","SGD+","Adam"])
plt.title("Figure 1: Multi Neuron learning rate = 5e-3")
plt.show()
```

EVALUATION OF THE 9 COMBINATIONS OF BETAS IN ADAM:

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

# Code used to evaluate the 9 configurations of the parameteters beta1
and beta2 of the Adam optimizer
import random
import numpy as np
import operator
import matplotlib.pyplot as plt
import time

seed = 100
random.seed(seed)
np.random.seed(seed)

from ComputationalGraphPrimer import *

class Adam_Multi_ComputationalGraphPrimer(ComputationalGraphPrimer):
      def __init__(self, beta1=0, beta2=0, *args, **kwargs):
        self.beta1 = beta1
        self.beta2 = beta2
        self.epsilon = 1e-8
        super().__init__(*args, **kwargs)

      def backprop_and_update_params_multi_neuron_model(self,
predictions, y_errors, t):
        """
        First note that loop index variable 'back_layer_index' starts
with the index of
        the last layer.  For the 3-layer example shown for 'forward',
back_layer_index
        starts with a value of 2, its next value is 1, and that's it.

        In the code below, the outermost loop is over the data samples in
a batch. As shown
        on Slide 73 of my Week 3 lecture, in order to calculate the
partials of Loss with
        respect to the learnable params, we need to backprop the
prediction errors and
        the gradients of the Sigmoid.  For the purpose of satisfying the
requirements of
        SGD, the backprop of the prediction errors and the gradients
needs to be carried
        out separately for each training data sample in a batch.  That's
what the outer
        loop is for.

        After we exit the outermost loop, we average over the results
obtained from each
        training data sample in a batch.

        Pay attention to the variable 'vars_in_layer'.  These store the
node variables in
        the current layer during backpropagation.
        """
        ## Eq. (24) on Slide 73 of my Week 3 lecture says we need to
store backproped errors in each layer leading up to the last:
        pred_err_backproped_at_layers =   [ {i : [None for j in range(
self.layers_config[i] ) ]
                                                                  for i
in range(self.num_layers)} for _ in range(self.batch_size) ]
        ## This will store "\delta L / \delta w" you see at the LHS of
the equations on Slide 73:
        partial_of_loss_wrt_params = {param : 0.0 for param in
self.all_params}
        ## For estimating the changes to the bias to be made on the basis
of the derivatives of the Sigmoids:
        bias_changes =   {i : [0.0 for j in range( self.layers_config[i]
) ]  for i in range(1, self.num_layers)}
        for b in range(self.batch_size):
            pred_err_backproped_at_layers[b][self.num_layers - 1] = [
y_errors[b] ]
            for back_layer_index in
reversed(range(1,self.num_layers)):             ## For the 3-layer
network, the first val for back_layer_index is 2 for the 3rd layer
                input_vals =
self.forw_prop_vals_at_layers[back_layer_index -1]     ## This is a list
of 8 two-element lists  --- since we have two nodes in the 2nd layer
                deriv_sigmoids
=  self.gradient_vals_for_layers[back_layer_index]   ## This is a list
eight one-element lists, one for each batch element
                vars_in_layer  =  self.layer_vars[back_layer_index]
          ## A list like ['xo']
                vars_in_next_layer_back  =  self.layer_vars[back_layer_in
dex - 1]   ## A list like ['xw', 'xz']
                vals_for_input_vars_dict
=  dict(zip(vars_in_next_layer_back,
self.forw_prop_vals_at_layers[back_layer_index - 1][b]))
                ## For the next statement, 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"
                layer_params =
self.layer_params[back_layer_index]
                transposed_layer_params =
list(zip(*layer_params))                ## Creating a transpose of the
link matrix, See Eq. 30 on Slide 77
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        pred_err_backproped_at_layers[b][back_layer_index
- 1][k] =
sum([self.vals_for_learnable_params[transposed_layer_params[k][i]]

             * pred_err_backproped_at_layers[b][back_layer_index][i]
```

```
                                        for i in
range(len(vars_in_layer))])
                for j,var in enumerate(vars_in_layer):
                    layer_params =
self.layer_params[back_layer_index][j]           ##  ['cp', 'cq']   for
the end layer
                    input_vars_to_param_map =
self.var_to_var_param[var]           ## These two statements align
the    {'xw': 'cp', 'xz': 'cq'}
                    param_to_vars_map = {param : var for var, param in
input_vars_to_param_map.items()}   ##   and the input vars   {'cp': 'xw',
'cq': 'xz'}

                    ##  Update the partials of Loss wrt to the learnable
parameters between the current layer
                    ##  and the previous layer. You are accumulating
these partials over the different training
                    ##  data samples in the batch being processed.  For
each training data sample, the formula
                    ##  being used is shown in Eq. (29) on Slide 77 of my
Week 3 slides:
                    for i,param in enumerate(layer_params):
                        partial_of_loss_wrt_params[param]   +=   pred_err
_backproped_at_layers[b][back_layer_index][j] * \
                                                                        v
als_for_input_vars_dict[param_to_vars_map[param]] * deriv_sigmoids[b][j]
                ##  We will now estimate the change in the bias that
needs to be made at each node in the previous layer
                ##  from the derivatives the sigmoid at the nodes in the
current layer and the prediction error as
                ##  backproped to the previous layer nodes:
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        if back_layer_index-1 > 0:
                            bias_changes[back_layer_index-1][k] +=
pred_err_backproped_at_layers[b][back_layer_index - 1][k] *
deriv_sigmoids[b][j]

        ## Now update the learnable parameters.  The loop shown below
carries out SGD mandated averaging
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = -
partial_of_loss_wrt_params[param] /  float(self.batch_size)
            # step = self.learning_rate * partial_of_loss_wrt_param
            # self.vals_for_learnable_params[param] += step

            m_t_plus_1 = self.beta1 * self.m_t[param] + (1 - self.beta1)
* partial_of_loss_wrt_param
```

```
            v_t_plus_1 = self.beta2 * self.v_t[param] + (1 - self.beta2) 
* (partial_of_loss_wrt_param**2)

            m_t_plus_1_hat = m_t_plus_1 / (1 - self.beta1**t)
            v_t_plus_1_hat = v_t_plus_1 / (1 - self.beta2**t)

            step = self.learning_rate * (m_t_plus_1_hat / 
np.sqrt(v_t_plus_1_hat + self.epsilon))
            self.vals_for_learnable_params[param] -= step

            self.v_t[param] = v_t_plus_1
            self.m_t[param] = m_t_plus_1


        ##  Finally we update the biases at all the nodes that aggregate 
data:
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                #self.bias[layer_index][k]  +=  self.learning_rate * ( 
bias_changes[layer_index][k] / float(self.batch_size) )
                partial_of_loss_wrt_param = - ( 
bias_changes[layer_index][k] / float(self.batch_size) )
                bias_m_t_plus_1 = self.beta1 * 
self.bias_m_t[layer_index][k] + (1 - self.beta1) * 
(partial_of_loss_wrt_param)
                bias_v_t_plus_1 = self.beta2 * 
self.bias_v_t[layer_index][k] + (1 - self.beta2) * 
(partial_of_loss_wrt_param**2)

                bias_m_t_plus_1_hat = bias_m_t_plus_1 / (1 - 
self.beta1**t)
                bias_v_t_plus_1_hat = bias_v_t_plus_1 / (1 - 
self.beta2**t)

                bias_step = self.learning_rate * (bias_m_t_plus_1_hat / 
np.sqrt(bias_v_t_plus_1_hat + self.epsilon))

                self.bias[layer_index][k] -= bias_step

                self.bias_v_t[layer_index][k] = bias_v_t_plus_1
                self.bias_m_t[layer_index][k] = bias_m_t_plus_1



      def run_training_loop_multi_neuron_model(self, training_data):

        class DataLoader:
            """
```



```
            To understand the logic of the dataloader, it would help if 
you first understand how
            the training dataset is created.  Search for the following 
function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, 
the training dataset
            consists of a Python dict with two keys, 0 and 1, the former 
points to a list of
            all Class 0 samples and the latter to a list of all Class 1 
samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian 
distribution.  The two
            classes have different means and variances.  The 
dimensionality of each data sample
            is set by the number of nodes in the input layer of the 
neural network.

            The data loader's job is to construct a batch of samples 
drawn randomly from the two
            lists mentioned above.  And it mush also associate the 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]]    ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in 
self.training_data[1]]    ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + 
len(self.training_data[1])

            def _getitem(self):
                cointoss = 
random.choice([0,1])                            ## When a batch is 
created by getbatch(), we want the

 ##   samples to be chosen randomly from the two lists
                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 =
[],[]                            ## First list for samples, the second
for labels
                maxval =
0.0                                               ## For approximate
batch data normalization
                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]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

        ##  The training loop must first initialize the learnable
parameters.  Remember, these are the
        ##  symbolic names in your input expressions for the neural layer
that do not begin with the
        ##  letter 'x'.  In this case, we are initializing with random
numbers from a uniform distribution
        ##  over the interval (0,1):
        self.vals_for_learnable_params = {param: random.uniform(0,1) for
param in self.learnable_params}
        ##  In the same  manner, we must also initialize the biases at
each node that aggregates forward
        ##  propagating data:
        self.bias =   {i : [random.uniform(0,1) for j in range(
self.layers_config[i] ) ]  for i in range(1, self.num_layers)}
        data_loader = DataLoader(training_data,
batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations =
0.0                                          ##  Average the loss over
iterations for printing out

      ##    every N iterations during the training loop.


        self.m_t = {param: 0 for param in self.learnable_params}
        self.v_t = {param: 0 for param in self.learnable_params}
        self.bias_m_t = {i : [0 for j in range( self.layers_config[i] )
]  for i in range(1, self.num_layers)}
```

```
        self.bias_v_t = {i : [0 for j in range( self.layers_config[i] )
]  for i in range(1, self.num_layers)}

        min_loss = 1000

        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)
                          ## FORW PROP works by side-effect
            predicted_labels_for_batch =
self.forw_prop_vals_at_layers[self.num_layers-1]           ## Predictions
from FORW PROP
            y_preds =  [item for sublist
in  predicted_labels_for_batch  for item in sublist]       ## Get numeric
vals for predictions
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
range(len(class_labels))])  ## Calculate loss for batch
            #We save the value of the minimum loss
            if loss<min_loss:
                  min_loss = loss
            loss_avg = loss /
float(len(class_labels))                                              ##
Average the loss over batch
            avg_loss_over_iterations +=
loss_avg                                                    ## Add to
Average loss over iterations
            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))                  ## Display avg loss
                avg_loss_over_iterations =
0.0                                                      ## Re-initialize
avg-over-iterations loss
            y_errors_in_batch = list(map(operator.sub, class_labels,
y_preds))
            ##MODIFIED CODE---------------------------------------------
            self.backprop_and_update_params_multi_neuron_model(y_preds,
y_errors_in_batch, t=i+1)
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()
        return loss_running_record, loss, min_loss
            ##END MODIFIED CODE----------------------------------------
-
```

```
beta1 = [0.8, 0.95, 0.99]
beta2 = [0.89, 0.9, 0.95]

loss = []
final_loss = []
minimum_loss = []
time_ = []
beta1_ = []
beta2_ = []


for b1 in beta1:
      for b2 in beta2:
        start_time = time.time()
        cgp = Adam_Multi_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,
                      debug = True,
                      beta1 = b1,
                      beta2 = b2,
              )

        cgp.parse_multi_layer_expressions()

        training_data = cgp.gen_training_data()

        loss_running_record, last_loss, min_loss =
cgp.run_training_loop_multi_neuron_model( training_data )
        end_time = time.time()
        elapsed_time = end_time - start_time

        loss.append(loss_running_record)
        final_loss.append(last_loss)
        minimum_loss.append(min_loss)
        time_.append(elapsed_time)
        beta1_.append(b1)
        beta2_.append(b2)

legend = []
```

```
plt.figure()
for i, _ in enumerate(beta1_):
  plt.plot(loss[i])
  leg= f"beta1: {beta1_[i]}, beta2: {beta2_[i]}"
  legend.append(leg)
plt.legend(legend)
plt.title("Figure : Adam Optimizer under 9 configurations")
plt.show()

for i, _ in enumerate(beta1_):
  print(f"beta1: {beta1_[i]}, beta2: {beta2_[i]}, final loss: {final_loss[i]}, minimum loss: {minimum_loss[i]}, time: {time_[i]} sec")
```