hw3_ChengjunGuo

Chengjun Guo

February 2023

1 SGD+

Equation:

$$v_{t+1} = \mu * v_t + g_{t+1}$$
$$p_{t+1} = p_t - \mathrm{lr} * v_{t+1}$$

In the equations, p means the parameter and t means the previous iteration. μ is the momentum coefficient that inherit the step of previous step. g is the gradient of loss. Compared to SGD, this method is basically inheriting the speed in previous step. With this algorithm, the learning rate would be more adaptive than original SGD algorithm and it will converge faster.

2 Adam

Equations:

$$m_{t+1} = \beta_1 * m_t + (1 - \beta_1) * g_{t+1}$$
$$v_{t+1} = \beta_2 * v_t + (1 - \beta_2) * (g_{t+1})^2$$
$$p_{t+1} = p_t - \mathrm{lr} * \frac{\hat{m}}{\sqrt{\hat{v}_{t+1} + \epsilon}}$$

where

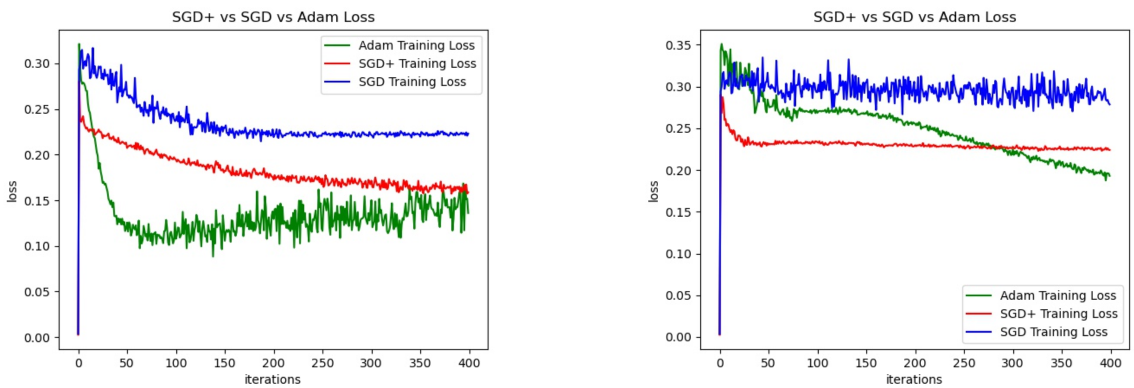
$$\hat{m}_k = \frac{m_k}{1 - \beta_1^k}$$
$$\hat{v}_k = \frac{v_k}{1 - {\beta_2}^k}$$

β_1 and β_2 are the user defined variables. Adam is combining the momentum based logic and the sparse gradient into a single algorithm. Here's the motion of development from adagrad to rmsprop. The adagrad is developed based on the intuition that whenever a partial derivative becomes non-zero, the rareness of such occurrences could mean that those dimensions carry high class discriminatory information and it should take larger steps. Then adagrad runs into problem that the monotonically increasing value for the denominator could case the learning rate for a parameter to become vanishing small. RMSprop replace the summation in denominator with its average over training iterations to fix it.

Then adam comes to use the momentum and be adaptive to different component of gradient.

3 Plots

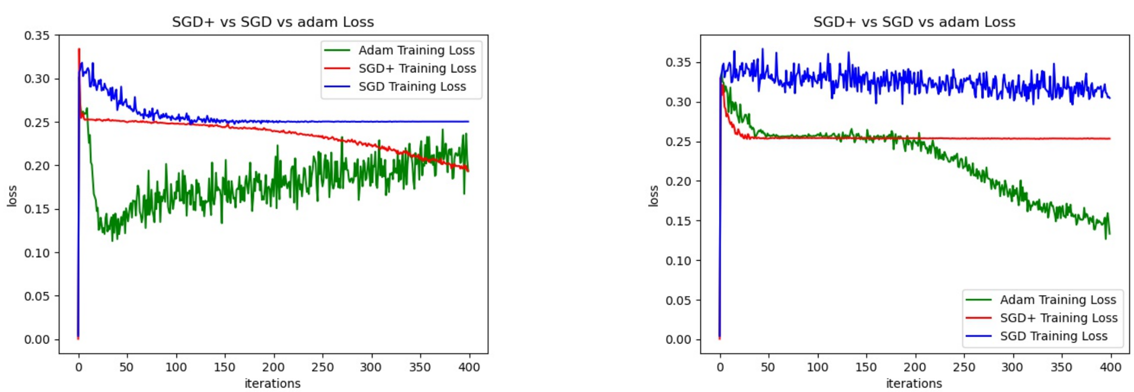
3.1 one neuron



(a) 1e-3 (b) 5e-5

Figure 1: one neuron

3.2 multi neuron



(a) 1e-3 (b) 5e-5

Figure 2: multi neuron

4 Discussion

Based on the training loss plots, we can find that one neuron is more stabilize than multi neuron. When the learning rate is 1e-3, it can be seen that Adam converge faster than sgd+ and sgd+ converge faster than sgd. Adam is increasing after it achieves a far lower loss than the sgd+. When the learning rate is 5e-5, sgd is not converging and sgd+ converge faster at faster and adam converge faster after that.Adam and sgd+ is better at handling low learning rate with the momentum.

5 Code

One neuron:

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

## one_neuron_classifier.py

"""
A one-neuron model is characterized by a single
    expression that you see in the value
supplied for the constructor parameter "expressions". In
     the expression supplied, the
names that being with 'x' are the input variables and the
     names that begin with the
other letters of the alphabet are the learnable
    parameters.
"""
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
import sys
sys.path.append( "E:\ECE60146DL\hw3_new\
    ComputationalGraphPrimer-1.1.2\
    ComputationalGraphPrimer")

import sys,os,os.path
import numpy as np
import re
import operator
import math
import random
import torch
from collections import deque
```

```
import copy
import matplotlib.pyplot as plt
import networkx as nx


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

from ComputationalGraphPrimer import *


class ComputationalGraphPrimerPlus(
    ComputationalGraphPrimer):
     def run_training_loop_one_neuron_model(self,
         training_data, momentum_coe):
          """
          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.gamma = momentum_coe
          self.bias = random.uniform(0, 1)  ## Adding the
              bias improves class discrimination.
          self.prev_grad = {param: 0 for param in self.
              learnable_params}
          self.prev_bias = 0
         ##  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 = []
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]
    y_preds, deriv_sigmoids = self.
        forward_prop_one_neuron_model(data_tuples)
            ##  FORWARD PROP of data
```

```
        loss = sum([(abs(class_labels[i] - y_preds[i
           ])) ** 2 for i in range(len(class_labels))
           ]) ## Find loss
        loss_avg = loss / float(len(class_labels))
           ## Average the loss over batch
        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)) ## Display
                average loss
            avg_loss_over_iterations = 0.0 ## Re-
               initialize avg loss
        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) ## BACKPROP loss
    return loss_running_record
    # plt.figure()
    # plt.plot(loss_running_record)
    # plt.show()

def backprop_and_update_params_one_neuron_model(self,
     y_error, vals_for_input_vars, deriv_sigmoid):
    """
    As should be evident from the syntax used in the
       following call to backprop function,
```

```
    self.
        backprop_and_update_params_one_neuron_model
        ( y_error_avg , data_tuple_avg ,
        deriv_sigmoid_avg)
                                                            ^^^

                                                              ^^^

                                                              ^^^

the values fed to the backprop function for its
    three arguments are averaged over the training
samples in the batch.  This in keeping with the
    spirit of SGD that calls for averaging the
information retained in the forward propagation
    over the samples in a batch.

See Slide 59 of my Week 3 slides for the math of
    back propagation for the One-Neuron network.
"""
input_vars = self.independent_vars
input_vars_to_param_map = self.var_to_var_param[
    self.output_vars[0]]
param_to_vars_map = {param: var for var, param in
     input_vars_to_param_map.items()}
vals_for_input_vars_dict = dict(zip(input_vars,
    list(vals_for_input_vars)))
vals_for_learnable_params = self.
    vals_for_learnable_params
for i, param in enumerate(self.
    vals_for_learnable_params):
    ## Calculate the next step in the parameter
        hyperplane
    #             step = self.learning_rate *
        y_error * vals_for_input_vars_dict[
        input_vars[i]] * deriv_sigmoid
    grad = y_error * vals_for_input_vars_dict[
        param_to_vars_map[param]] * deriv_sigmoid
    step = self.learning_rate * grad + self.gamma
         * self.prev_grad[param]
    ## Update the learnable parameters
    self.prev_grad[param] = step
    self.vals_for_learnable_params[param] += step
grad = y_error * deriv_sigmoid
```

```
        self.prev_bias = self.gamma * self.prev_bias +
            self.learning_rate * grad
        self.bias += self.prev_bias



class ComputationalGraphPrimerAdam(
    ComputationalGraphPrimer):
    def run_training_loop_one_neuron_model(self,
        training_data, beta1,beta2):
        """
        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.epsilon = 1e-8
        self.beta1 = beta1
        self.beta2 = beta2
        self.bias = random.uniform(0, 1)  ## Adding the
            bias improves class discrimination.
        self.prev_m = {param: 0 for param in self.
            learnable_params}
        self.prev_v = {param: 0 for param in self.
            learnable_params}
        self.prev_biasm = 0
        self.prev_biasv = 0
        ##  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 = []
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]
    y_preds, deriv_sigmoids = self.
        forward_prop_one_neuron_model(data_tuples)
          ##  FORWARD PROP of data
```

```
        loss = sum([(abs(class_labels[i] - y_preds[i
            ])) ** 2 for i in range(len(class_labels))
            ]) ## Find loss
        loss_avg = loss / float(len(class_labels))
            ## Average the loss over batch
        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)) ## Display
                 average loss
            avg_loss_over_iterations = 0.0 ## Re-
                initialize avg loss
        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) ## BACKPROP loss
    return loss_running_record
    # plt.figure()
    # plt.plot(loss_running_record)
    # plt.show()

def backprop_and_update_params_one_neuron_model(self,
     y_error, vals_for_input_vars, deriv_sigmoid):
    """
    As should be evident from the syntax used in the
        following call to backprop function,
```

```
    self.
        backprop_and_update_params_one_neuron_model
        ( y_error_avg , data_tuple_avg ,
        deriv_sigmoid_avg)
                                                            ^^^

                                                              ^^^

                                                              ^^^

the values fed to the backprop function for its
    three arguments are averaged over the training
samples in the batch. This in keeping with the
    spirit of SGD that calls for averaging the
information retained in the forward propagation
    over the samples in a batch.

See Slide 59 of my Week 3 slides for the math of
    back propagation for the One-Neuron network.
"""
input_vars = self.independent_vars
input_vars_to_param_map = self.var_to_var_param[
    self.output_vars[0]]
param_to_vars_map = {param: var for var, param in
     input_vars_to_param_map.items()}
vals_for_input_vars_dict = dict(zip(input_vars,
    list(vals_for_input_vars)))
vals_for_learnable_params = self.
    vals_for_learnable_params
for i, param in enumerate(self.
    vals_for_learnable_params):
    ## Calculate the next step in the parameter
        hyperplane
    #             step = self.learning_rate *
        y_error * vals_for_input_vars_dict[
        input_vars[i]] * deriv_sigmoid
    grad = y_error * vals_for_input_vars_dict[
        param_to_vars_map[param]] * deriv_sigmoid
    m = self.beta1 * self.prev_m[param] + (1-self
        .beta1) * grad
    v = self.beta2 * self.prev_v[param] + (1-self
        .beta2) * (grad ** 2)
    step = self.learning_rate * ((self.prev_m[
        param]/(1-self.beta1 ** (i+1)))/np.sqrt((
```

```
                self.prev_v[param]/(1-self.beta1 ** (i+1))
                )+self.epsilon))
            ## Update the learnable parameters
            self.prev_m[param] = m
            self.prev_v[param] = v
            self.vals_for_learnable_params[param] += step
        grad = y_error * deriv_sigmoid
        self.prev_biasv = self.beta1 * self.prev_biasm +
            (1-self.beta1) * grad
        self.prev_biasm = self.beta2 * self.prev_biasv +
            (1-self.beta2) * (grad ** 2)
        self.bias -= self.learning_rate * (self.
            prev_biasm/(self.prev_biasv+self.epsilon))


cgpp = ComputationalGraphPrimerPlus(
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd
                    '],
                output_vars = ['xw'],
                dataset_size = 5000,
                # learning_rate = 1e-3,
               learning_rate = 5 * 1e-5,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
      )
cgpa = 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,
               learning_rate = 5 * 1e-5,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
      )

cgpp.parse_expressions()
cgpa.parse_expressions()
#cgp.display_network1()
```

```
# cgpp.display_network2()
training_data = cgpp.gen_training_data()
loss = cgpp.run_training_loop_one_neuron_model(
    training_data ,0.0)
loss_plus = cgpp.run_training_loop_one_neuron_model(
    training_data , 0.99)
loss_adam = cgpa.run_training_loop_one_neuron_model(
    training_data , 0.9, 0.99)

plt.figure()
plt.ylabel('loss')
plt.xlabel('iterations')
plt.title('SGD+ vs SGD vs Adam Loss')
plt.plot(loss_adam, label = 'Adam Training Loss', color='
    g')
plt.plot(loss_plus, label = 'SGD+ Training Loss', color='
    r')
plt.plot(loss, label = 'SGD Training Loss', color='b')
plt.legend()
#plt.show()
plt.savefig("one_neuron_loss_alt.jpg")
```

multi neuron:

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

## multi_neuron_classifier.py

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


                        input
```

```
                                                    x

                                                       x = node

                                                    x         x|

                                                       | = sigmoid
                                                       activation
                                                                        x|
                                                    x         x|

                                                    x

                                                 layer_0    layer_1
                                                    layer_2


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

        forward_prop_multi_neuron_model()
and
        backprop_and_update_params_multi_neuron_model()

Both of these functions are called by the training
    function:

        run_training_loop_multi_neuron_model()

"""
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
import sys
sys.path.append( "E:\ECE60146DL\hw3_new\
    ComputationalGraphPrimer-1.1.2\
    ComputationalGraphPrimer")

import sys,os,os.path
import numpy as np
import re
import operator
import math
```

```
import random
import torch
from collections import deque
import copy
import matplotlib.pyplot as plt
import networkx as nx

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

from ComputationalGraphPrimer import *

class ComputationalGraphPrimerPlus(
    ComputationalGraphPrimer):
     def run_training_loop_multi_neuron_model(self,
         training_data, momentum):
          self.gamma = momentum
          self.prev_grad = {param: 0 for param in self.
              learnable_params}
          self.prev_bias = [0 for _ in range(self.
              num_layers-1)]
          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}

self.bias = [random.uniform(0, 1) for _ in
             range(self.num_layers - 1)]  ##
                 Adding the bias to each layer
                 improves
##   class discrimination. We initialize it
##   to a random number.

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 = 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)  ## BACKPROP
                loss
        return loss_running_record



def backprop_and_update_params_multi_neuron_model(
    self, y_error, class_labels):
    """
```

```
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.

Stochastic Gradient Gradient calls for the
    backpropagated loss to be averaged over
the samples in a batch.  To explain how this
    averaging is carried out by the
backprop function, consider the last node on the
    example shown in the forward()
function above.  Standing at the node, we look at
     the 'input' values stored in the
variable "input_vals".  Assuming a batch size of
    8, this will be list of
lists. Each of the inner lists will have two
    values for the two nodes in the
hidden layer. And there will be 8 of these for
    the 8 elements of the batch.  We average
these values 'input vals' and store those in the
    variable "input_vals_avg".  Next we
must carry out the same batch-based averaging for
     the partial derivatives stored in the
variable "deriv_sigmoid".

Pay attention to the variable 'vars_in_layer'.
    These store the node variables in
the current layer during backpropagation.  Since
    back_layer_index starts with a
value of 2, the variable 'vars_in_layer' will
    have just the single node for the
example shown for forward(). With respect to what
     is stored in vars_in_layer', the
variables stored in 'input_vars_to_layer'
    correspond to the input layer with
respect to the current layer.
"""
# backproped prediction error:
pred_err_backproped_at_layers = {i: [] for i in
    range(1, self.num_layers - 1)}
pred_err_backproped_at_layers[self.num_layers -
    1] = [y_error]
for back_layer_index in reversed(range(1, self.
    num_layers)):
```

```
input_vals = self.forw_prop_vals_at_layers[
    back_layer_index - 1]
input_vals_avg = [sum(x) for x in zip(*
    input_vals)]
input_vals_avg = list(map(operator.truediv,
    input_vals_avg, [float(len(class_labels))]
     * len(class_labels)))
deriv_sigmoid = self.gradient_vals_for_layers
    [back_layer_index]
deriv_sigmoid_avg = [sum(x) for x in zip(*
    deriv_sigmoid)]
deriv_sigmoid_avg = list(map(operator.truediv
    , deriv_sigmoid_avg,
                              [float(len(
                                 class_labels)
                                 )] * len(
                                 class_labels)
                                 ))
vars_in_layer = self.layer_vars[
    back_layer_index]  ## a list like ['xo']
vars_in_next_layer_back = self.layer_vars[
    back_layer_index - 1]  ## a list like ['xw
    ', 'xz']

layer_params = self.layer_params[
    back_layer_index]
## note that layer_params are stored in a
    dict like
##     {1: [['ap', 'aq', 'ar', 'as'], ['bp',
    'bq', 'br', 'bs']], 2: [['cp', 'cq']]}
## "layer_params[idx]" is a list of lists for
     the link weights in layer whose output
    nodes are in layer "idx"
transposed_layer_params = list(zip(*
    layer_params))  ## creating a transpose of
     the link matrix

backproped_error = [None] * len(
    vars_in_next_layer_back)
for k, varr in enumerate(
    vars_in_next_layer_back):
     for j, var2 in enumerate(vars_in_layer):
         backproped_error[k] = sum([self.
             vals_for_learnable_params[
             transposed_layer_params[k][i]] *
                                        pred_err_backproped_at_layers
```

```
                                          [
                                          back_layer_index
                                          ][i]
                                        for i in
                                          range(
                                          len(
                                          vars_in_layer
                                          ))])
#

    deriv_sigmoid_avg[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]
    ## Regarding the parameter update loop
        that follows, see the Slides 74
        through 77 of my Week 3
    ## lecture slides for how the parameters
         are updated using the partial
        derivatives stored away
    ## during forward propagation of data.
        The theory underlying these
        calculations is presented
    ## in Slides 68 through 71.
    for i, param in enumerate(layer_params):
        gradient_of_loss_for_param =
            input_vals_avg[i] *
            pred_err_backproped_at_layers[
            back_layer_index][j]
        grad = gradient_of_loss_for_param *
            deriv_sigmoid_avg[j]
        self.prev_grad[param] = self.
            learning_rate * grad + self.gamma
            * self.prev_grad[param]
        self.vals_for_learnable_params[param]
             += self.prev_grad[param]
self.prev_bias[back_layer_index - 1] = self.
    learning_rate * sum(
    pred_err_backproped_at_layers[
    back_layer_index]) * sum(deriv_sigmoid_avg
    )/len(deriv_sigmoid_avg) + self.gamma *
```

```
            self.prev_bias[back_layer_index -1]
        self.bias[back_layer_index - 1] += self.
            prev_bias[back_layer_index - 1]



class ComputationalGraphPrimerAdam(
    ComputationalGraphPrimer):
     def run_training_loop_multi_neuron_model(self,
         training_data, beta1, beta2):
          self.beta1 = beta1
          self.beta2 = beta2
          self.epsilon = 1e-8
          self.prev_m = {param: 0 for param in self.
              learnable_params}
          self.prev_v = {param: 0 for param in self.
              learnable_params}
          self.prev_biasv = [0 for _ in range(self.
              num_layers-1)]
          self.prev_biasm = [0 for _ in range(self.
              num_layers - 1)]
          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}

self.bias = [random.uniform(0, 1) for _ in
             range(self.num_layers - 1)]  ##
                 Adding the bias to each layer
                 improves
##   class discrimination. We initialize it
##   to a random number.

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 = 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)  ## BACKPROP
            loss
    return loss_running_record



def backprop_and_update_params_multi_neuron_model(
    self, y_error, class_labels):
    """
    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.

Stochastic Gradient Gradient calls for the
    backpropagated loss to be averaged over
the samples in a batch.  To explain how this
    averaging is carried out by the
backprop function, consider the last node on the
    example shown in the forward()
function above.  Standing at the node, we look at
     the 'input' values stored in the
variable "input_vals".  Assuming a batch size of
    8, this will be list of
lists. Each of the inner lists will have two
    values for the two nodes in the
hidden layer. And there will be 8 of these for
    the 8 elements of the batch.  We average
these values 'input vals' and store those in the
    variable "input_vals_avg".  Next we
must carry out the same batch-based averaging for
     the partial derivatives stored in the
variable "deriv_sigmoid".

Pay attention to the variable 'vars_in_layer'.
    These store the node variables in
the current layer during backpropagation.  Since
    back_layer_index starts with a
value of 2, the variable 'vars_in_layer' will
    have just the single node for the
example shown for forward(). With respect to what
     is stored in vars_in_layer', the
variables stored in 'input_vars_to_layer'
    correspond to the input layer with
respect to the current layer.
"""
# backproped prediction error:
pred_err_backproped_at_layers = {i: [] for i in
    range(1, self.num_layers - 1)}
pred_err_backproped_at_layers[self.num_layers -
    1] = [y_error]
for back_layer_index in reversed(range(1, self.
    num_layers)):
     input_vals = self.forw_prop_vals_at_layers[
```

```
    back_layer_index - 1]
input_vals_avg = [sum(x) for x in zip(*
    input_vals)]
input_vals_avg = list(map(operator.truediv,
    input_vals_avg, [float(len(class_labels))]
     * len(class_labels)))
deriv_sigmoid = self.gradient_vals_for_layers
    [back_layer_index]
deriv_sigmoid_avg = [sum(x) for x in zip(*
    deriv_sigmoid)]
deriv_sigmoid_avg = list(map(operator.truediv
    , deriv_sigmoid_avg,
                              [float(len(
                                  class_labels)
                                  )] * len(
                                  class_labels)
                                  ))
vars_in_layer = self.layer_vars[
    back_layer_index]  ## a list like ['xo']
vars_in_next_layer_back = self.layer_vars[
    back_layer_index - 1]  ## a list like ['xw
    ', 'xz']

layer_params = self.layer_params[
    back_layer_index]
## note that layer_params are stored in a
    dict like
##     {1: [['ap', 'aq', 'ar', 'as'], ['bp',
    'bq', 'br', 'bs']], 2: [['cp', 'cq']]}
## "layer_params[idx]" is a list of lists for
     the link weights in layer whose output
    nodes are in layer "idx"
transposed_layer_params = list(zip(*
    layer_params))  ## creating a transpose of
     the link matrix

backproped_error = [None] * len(
    vars_in_next_layer_back)
for k, varr in enumerate(
    vars_in_next_layer_back):
     for j, var2 in enumerate(vars_in_layer):
         backproped_error[k] = sum([self.
             vals_for_learnable_params[
             transposed_layer_params[k][i]] *
                                  pred_err_backproped_at_layers
                                      [
```

```
                                                back_layer_index
                                                ][i]
                                              for i in
                                                range(
                                                len(
                                                vars_in_layer
                                                ))])
#

    deriv_sigmoid_avg[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]
    ## Regarding the parameter update loop
        that follows, see the Slides 74
        through 77 of my Week 3
    ## lecture slides for how the parameters
         are updated using the partial
        derivatives stored away
    ## during forward propagation of data.
        The theory underlying these
        calculations is presented
    ## in Slides 68 through 71.
    for i, param in enumerate(layer_params):
        gradient_of_loss_for_param =
            input_vals_avg[i] *
            pred_err_backproped_at_layers[
            back_layer_index][j]
        grad = gradient_of_loss_for_param *
            deriv_sigmoid_avg[j]
        m = self.beta1 * self.prev_m[param] +
             (1 - self.beta1) * grad
        v = self.beta2 * self.prev_v[param] +
             (1 - self.beta2) * (grad ** 2)
        step = self.learning_rate * ((self.
            prev_m[param] / (1 - self.beta1 **
             (i + 1))) / np.sqrt((self.prev_v[
            param] / (1 - self.beta1 ** (i +
            1))) + self.epsilon))
        self.prev_m[param] = m
        self.prev_v[param] = v
```

```
                self.vals_for_learnable_params[param]
                    += step
            grad = sum(pred_err_backproped_at_layers[
                back_layer_index]) * sum(deriv_sigmoid_avg
                )/len(deriv_sigmoid_avg)
            self.prev_biasv[back_layer_index - 1] = self.
                beta1 * self.prev_biasm[back_layer_index -
                 1] + (1 - self.beta1) * grad
            self.prev_biasm[back_layer_index - 1] = self.
                beta2 * self.prev_biasv[back_layer_index -
                 1] + (1 - self.beta2) * (grad ** 2)
            self.bias -= self.learning_rate * (self.
                prev_biasm[back_layer_index - 1]/(self.
                prev_biasv[back_layer_index - 1]+self.
                epsilon))



cgp = ComputationalGraphPrimerPlus(
               num_layers = 3,
               layers_config = [4,2,1],
                                           # num of nodes
                   in each layer
               expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs
                   ',
                               'xz=bp*xp+bq*xq+br*xr+bs*xs
                                   ',
                               'xo=cp*xw+cq*xz'],
               output_vars = ['xo'],
               dataset_size = 5000,
               # learning_rate = 1e-3,
              learning_rate = 5 * 1e-5,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
       )
cgpa = 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,
                 learning_rate = 5 * 1e-5,
                  training_iterations = 40000,
                  batch_size = 8,
                  display_loss_how_often = 100,
                  debug = True,
      )

cgp.parse_multi_layer_expressions()
cgpa.parse_multi_layer_expressions()

#cgp.display_network1()
# cgp.display_network2()

training_data = cgp.gen_training_data()

loss = cgp.run_training_loop_multi_neuron_model(
    training_data, 0.0 )
loss_plus = cgp.run_training_loop_multi_neuron_model(
    training_data, 0.99)
loss_adam = cgpa.run_training_loop_multi_neuron_model(
    training_data,0.9,0.99)
plt.figure()
plt.ylabel('loss')
plt.xlabel('iterations')
plt.title('SGD+ vs SGD vs adam Loss')
plt.plot(loss_adam, label = 'Adam Training Loss', color='
    g')
plt.plot(loss_plus, label = 'SGD+ Training Loss', color='
    r')
plt.plot(loss, label = 'SGD Training Loss', color='b')
plt.legend()
#plt.show()
plt.savefig("multi_neuron_loss_alt2.jpg")
```