BME 646/ ECE695DL: Homework 3

Bianjiang Yang

6 Feb 2022

1 Introduction

The main goal of this homework3 is to develop a greater appreciation for the optimizations used for estimating the step size in the hyperplane spanned all the learnable parameters in an SGD based approach to updating the values of the parameters. SGD with momentum is method which helps accelerate gradients vectors in the right directions, thus leading to faster converging. It is one of the most popular optimization algorithms and many state-of-the-art models are trained using it. To be specific, in this work, we need to add momentum to SGD to make an SGD+.

2 Methodology

Packages: re, matplotlib.pyplot, operator, numpy, random, ComputationalGraphPrimer
Language: Python3
System: Ubuntu 20.04.2
Learning principle:

$$p_{t+1} = p_t - lr * g_{t+1} \tag{1}$$

Momentum: Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillations. It does this by adding a fraction v of the update vector of the past time step to the current update vector:

$$v_{t+1} = \mu * v_t - lr * g_{t+1} \tag{2}$$

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

Explanation for the code: The code I submitted has been modified to be runable without importing ComputationalGraphPrimer, so you can simply run it by excuting the script 'python3 one_neuron_classifier_sgd_plus.py' and 'python3 multi_neuron_classifier_sgd_plus.py'. Furthermore, for simplicity, *I deleted unused functions and classes* in ComputationalGraphPrimer. You should modify the savefig directory or just use the plt.show() function to show the loss plots.

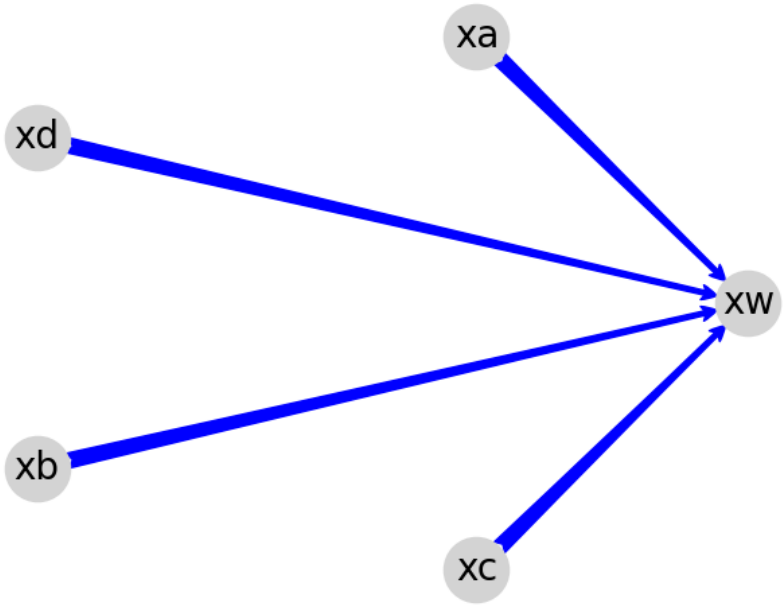

Figure 1: one neuron classifier computational graph

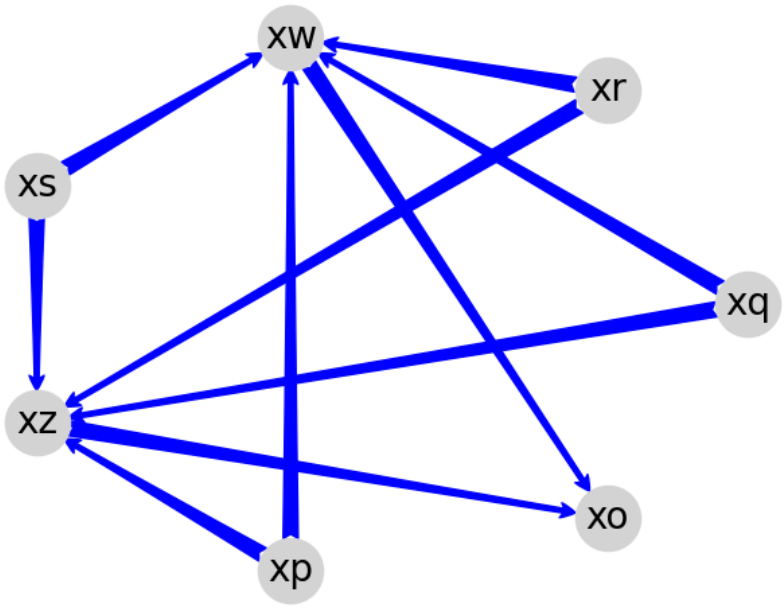

Figure 2: multi neuron classifier computational graph

3 Implementation and Results

One neuron classifier sgd+

```
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
import time

class Exp:
    def __init__(self, exp, body, dependent_var, right_vars, right_params):
        self.exp = exp
        self.body = body
        self.dependent_var = dependent_var
        self.right_vars = right_vars
        self.right_params = right_params

#_______________________________  ComputationalGraphPrimer Class Definition  ________________________________

class ComputationalGraphPrimer(object):

    def __init__(self, *args, **kwargs ):
        if args:
            raise ValueError(
                   '''ComputationalGraphPrimer constructor can only be called with keyword arguments for
                      the following keywords: expressions, output_vars, dataset_size, grad_delta,
                      learning_rate, display_loss_how_often, one_neuron_model, training_iterations,
                      batch_size, num_layers, layers_config, epochs, and debug''')
        expressions = output_vars = dataset_size = grad_delta = display_loss_how_often = learning_rate = one_neu
        if 'one_neuron_model' in kwargs              :   one_neuron_model = kwargs.pop('one_neuron_model')
        if 'batch_size' in kwargs                    :   batch_size = kwargs.pop('batch_size')
        if 'num_layers' in kwargs                    :   num_layers = kwargs.pop('num_layers')
        if 'layers_config' in kwargs                 :   layers_config = kwargs.pop('layers_config')
        if 'expressions' in kwargs                   :   expressions = kwargs.pop('expressions')
        if 'output_vars' in kwargs                   :   output_vars = kwargs.pop('output_vars')
        if 'dataset_size' in kwargs                  :   dataset_size = kwargs.pop('dataset_size')
```

```
if 'learning_rate' in kwargs                 :   learning_rate = kwargs.pop('learning_rate')
if 'training_iterations' in kwargs           :   training_iterations = \
                                                          kwargs.pop('training_iterations')
if 'grad_delta' in kwargs                    :   grad_delta = kwargs.pop('grad_delta')
if 'display_loss_how_often' in kwargs        :   display_loss_how_often = kwargs.pop('display_loss_how_o
if 'epochs' in kwargs                        :   epochs = kwargs.pop('epochs')
if 'debug' in kwargs                         :   debug = kwargs.pop('debug')
if 'mu' in kwargs                            :   self.mu = kwargs.pop('mu')
if len(kwargs) != 0: raise ValueError('''You have provided unrecognizable keyword args''')
self.one_neuron_model =  True if one_neuron_model is not None else False
if training_iterations:
    self.training_iterations = training_iterations
self.batch_size  =  batch_size if batch_size else 4
self.num_layers = num_layers
if layers_config:
    self.layers_config = layers_config
if expressions:
    self.expressions = expressions
if output_vars:
    self.output_vars = output_vars
if dataset_size:
    self.dataset_size = dataset_size
if learning_rate:
    self.learning_rate = learning_rate
else:
    self.learning_rate = 1e-6
if grad_delta:
    self.grad_delta = grad_delta
else:
    self.grad_delta = 1e-4
if display_loss_how_often:
    self.display_loss_how_often = display_loss_how_often
if dataset_size:
    self.dataset_input_samples  = {i : None for i in range(dataset_size)}
    self.true_output_vals       = {i : None for i in range(dataset_size)}
self.vals_for_learnable_params = None
self.epochs = epochs
if debug:
    self.debug = debug
else:
    self.debug = 0
self.independent_vars = None
self.gradient_of_loss = None
self.gradients_for_learnable_params = None
self.expressions_dict = {}
self.LOSS = []                               ##  loss values for all iterations of training
```

```
    self.all_vars = set()
    if (one_neuron_model is True) or (num_layers is not None):
        self.independent_vars = []
        self.learnable_params = []
    else:
        self.independent_vars = set()
        self.learnable_params = set()
    self.dependent_vars = {}
    self.depends_on = {}                        ##  See Introduction for the meaning of this
    self.leads_to = {}                          ##  See Introduction for the meaning of this
    self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

def parse_expressions(self):
    '''
    This method creates a DAG from a set of expressions that involve variables and learnable
    parameters. The expressions are based on the assumption that a symbolic name that starts
    with the letter 'x' is a variable, with all other symbolic names being learnable parameters.
    The computational graph is represented by two dictionaries, 'depends_on' and 'leads_to'.
    To illustrate the meaning of the dictionaries, something like "depends_on['xz']" would be
    set to a list of all other variables whose outgoing arcs end in the node 'xz'.  So
    something like "depends_on['xz']" is best read as "node 'xz' depends on ...." where the
    dots stand for the array of nodes that is the value of "depends_on['xz']".  On the other
    hand, the 'leads_to' dictionary has the opposite meaning.  That is, something like
    "leads_to['xz']" is set to the array of nodes at the ends of all the arcs that emanate
    from 'xz'.
    '''
    self.exp_objects = []
    for exp in self.expressions:
        left,right = exp.split('=')
        self.all_vars.add(left)
        self.expressions_dict[left] = right
        self.depends_on[left] = []
        parts = re.findall('([a-zA-Z]+)', right)
        right_vars = []
        right_params = []
        for part in parts:
            if part.startswith('x'):
                self.all_vars.add(part)
                self.depends_on[left].append(part)
                right_vars.append(part)
            else:
                if self.one_neuron_model is True:
                    self.learnable_params.append(part)
                else:
                    self.learnable_params.add(part)
                right_params.append(part)
```

```
        exp_obj = Exp(exp, right, left, right_vars, right_params)
        self.exp_objects.append(exp_obj)
    if self.debug:
        print("\n\nall variables: %s" % str(self.all_vars))
        print("\n\nlearnable params: %s" % str(self.learnable_params))
        print("\n\ndependencies: %s" % str(self.depends_on))
        print("\n\nexpressions dict: %s" % str(self.expressions_dict))
    for var in self.all_vars:
        if var not in self.depends_on:              # that is, var is not a key in the depends_on dict
            if self.one_neuron_model is True:
                self.independent_vars.append(var)
            else:
                self.independent_vars.add(var)
    self.input_size = len(self.independent_vars)
    if self.debug:
        print("\n\nindependent vars: %s" % str(self.independent_vars))
    self.dependent_vars = [var for var in self.all_vars if var not in self.independent_vars]
    self.output_size = len(self.dependent_vars)
    self.leads_to = {var : set() for var in self.all_vars}
    for k,v in self.depends_on.items():
        for var in v:
            self.leads_to[var].add(k)
    if self.debug:
        print("\n\nleads_to dictionary: %s" % str(self.leads_to))


### Introduced in 1.0.5
##########################################################################################################
########################################## one neuron model ##############################################
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).
    """
    import copy
    self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}
    self.bias = random.uniform(0,1)
    self.momentum = copy.deepcopy(self.vals_for_learnable_params)
    for i in self.momentum.keys():
        self.momentum[i] = 0
    self.m_bias = 0
    # self.mu = 0.99

    class DataLoader:
```

```
    """
    The data loader's job is to construct a batch of randomly chosen samples from the
    training data.  But, obviously, it must first associate the class labels 0 and 1 with
    the training data supplied to the constructor of the DataLoader.   NOTE:  The training
    data is generated in the Examples script by calling 'cgp.gen_training_data()' in the
    ****Utility Functions*** section of this file.  That function returns two normally
    distributed set of number with different means and variances.  One is for key value '0'
    and the other for the key value '1'.  The constructor of the DataLoader associated a'
    class label with each sample separately.
    """
    def __init__(self, training_data, batch_size):
        self.training_data = training_data
        self.batch_size = batch_size
        self.class_0_samples = [(item, 0) for item in self.training_data[0]]
        self.class_1_samples = [(item, 1) for item in self.training_data[1]]
    def __len__(self):
        return len(self.training_data[0]) + len(self.training_data[1])
    def _getitem(self):
        cointoss = random.choice([0,1])
        if cointoss == 0:
            return random.choice(self.class_0_samples)
        else:
            return random.choice(self.class_1_samples)
    def getbatch(self):
        batch_data,batch_labels = [],[]
        maxval = 0.0
        for _ in range(self.batch_size):
            item = self._getitem()
            if np.max(item[0]) > maxval:
                maxval = np.max(item[0])
            batch_data.append(item[0])
            batch_labels.append(item[1])
        batch_data = [item/maxval for item in batch_data]
        batch = [batch_data, batch_labels]
        return batch

data_loader = DataLoader(training_data, batch_size=self.batch_size)
loss_running_record = []
i = 0
avg_loss_over_literations = 0.0
for i in range(self.training_iterations):
    data = data_loader.getbatch()
    data_tuples = data[0]
    class_labels = data[1]
    y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tuples)
    loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])
```

```
            loss_avg = loss / float(len(class_labels))
            avg_loss_over_literations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_literations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_literations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_literations))
                avg_loss_over_literations = 0.0
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
            data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
            data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                      [float(len(class_labels))] * len(class_labels) ))
            self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg, deriv_sigmoid_avg)

        return loss_running_record

    def forward_prop_one_neuron_model(self, data_tuples_in_batch):
        """
        As the one-neuron model is characterized by a single expression, the main job of this function is
        to evaluate that expression for each data tuple in the incoming batch.  The resulting output is
        fed into the sigmoid activation function and the partial derivative of the sigmoid with respect
        to its input calculated.

        See Slides 103 through 108 of Week 3 slides for the logic implemented  here.
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        """
        output_vals = []
        deriv_sigmoids = []
        for vals_for_input_vars in data_tuples_in_batch:
            input_vars = self.independent_vars                    ## this is just a list of vars for input nodes
            vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
            exp_obj = self.exp_objects[0]
            output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict, self.vals_for_learnable_p
            output_val = output_val + self.bias
            ## apply sigmoid activation (output confined to [0.0,1.0] interval)
            output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))
            ## calculate partial of the activation function as a function of its input
            deriv_sigmoid = output_val * (1.0 - output_val)
            output_vals.append(output_val)
            deriv_sigmoids.append(deriv_sigmoid)
        return output_vals, deriv_sigmoids


    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 Slides 103 through 108 of Week 3 slides for the logic implemented  here.
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
    """
    input_vars = self.independent_vars
    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  \
               + self.mu * self.momentum[param]
        self.vals_for_learnable_params[param] += step
        self.momentum[param] = step
    step = self.mu * self.m_bias + self.learning_rate * y_error * deriv_sigmoid
    self.bias += step    ## the step to take for the bias
    self.m_bias = step
######################################################################################################


def eval_expression(self, exp, vals_for_vars, vals_for_learnable_params, ind_vars=None):
    self.debug1 = False
    if self.debug1:
        print("\n\nSTEP1: [original expression] exp: %s" % exp)
    if ind_vars is not None:
        for var in ind_vars:
            exp = exp.replace(var, str(vals_for_vars[var]))
    else:
        for var in self.independent_vars:
            exp = exp.replace(var, str(vals_for_vars[var]))
    if self.debug1:
        print("\n\nSTEP2: [replaced ars by their vals] exp: %s" % exp)
    for ele in self.learnable_params:
        exp = exp.replace(ele, str(vals_for_learnable_params[ele]))
    if self.debug1:
        print("\n\nSTEP4: [replaced learnable params by their vals] exp: %s" % exp)
    return eval( exp.replace('^', '**') )


def gen_training_data(self):
```

```
        """
        This 2-class dataset is used for the demos in the following Examples directory scripts:

                    one_neuron_classifier.py
                    multi_neuron_classifier.py
                    multi_neuron_classifier.py

        The classes are labeled 0 and 1.  All of the data for class 0 is simply a list of
        numbers associated with the key 0.  Similarly all the data for class 1 is another list of
        numbers associated with the key 1.

        For each class, the dataset starts out as being standard normal (zero mean and unit variance)
        to which we add a mean value of 2.0 for class 0 and we add mean value of 4 to the square of
        the original numbers for class 1.
        """
        num_input_vars = len(self.independent_vars)
        training_data_class_0 = []
        training_data_class_1 = []
        for i in range(self.dataset_size //2):
            # Standard normal means N(0,1), meaning zero mean and unit variance
            # Such values are significant in the interval [-3.0,+3.0]
            for_class_0 = np.random.standard_normal( num_input_vars )
            for_class_1 = np.random.standard_normal( num_input_vars )
            # translate class_1 data so that the mean is shifted to +4.0 and also
            # change the variance:
            for_class_0 = for_class_0 + 2.0
            for_class_1 = for_class_1 * 2 + 4.0
            training_data_class_0.append( for_class_0 )
            training_data_class_1.append( for_class_1 )
        training_data = {0 : training_data_class_0, 1 : training_data_class_1}
        return training_data

#_________________________  End of ComputationalGraphPrimer Class Definition _________________________


#_____________________________    Test code follows    _________________________________

if __name__ == '__main__':
    import random
    import numpy

    seed = 0
    random.seed(seed)
    numpy.random.seed(seed)
    # import sys
    # sys.path.append("/home/yangbj/695/hw3/ComputationalGraphPrimer-1.0.8/")
```

```
# from ComputationalGraphPrimer import *

cgp1 = ComputationalGraphPrimer(
    one_neuron_model=True,
    expressions=['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
    output_vars=['xw'],
    dataset_size=5000,
    learning_rate=1e-3,
    #               learning_rate = 5 * 1e-2,
    training_iterations=40000,
    batch_size=8,
    display_loss_how_often=100,
    debug=True,
    mu = 0.99
)

cgp1.parse_expressions()

training_data = cgp1.gen_training_data()
loss_running_record1 = cgp1.run_training_loop_one_neuron_model(training_data)


cgp2 = ComputationalGraphPrimer(
    one_neuron_model=True,
    expressions=['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
    output_vars=['xw'],
    dataset_size=5000,
    learning_rate=1e-3,
    #               learning_rate = 5 * 1e-2,
    training_iterations=40000,
    batch_size=8,
    display_loss_how_often=100,
    debug=True,
    mu = 0
)

cgp2.parse_expressions()
# cgp2.display_network2()
loss_running_record2 = cgp2.run_training_loop_one_neuron_model(training_data)

plt.figure()
plt.plot(loss_running_record1)
plt.plot(loss_running_record2)
plt.show()
plt.savefig("/home/yangbj/695/hw3/imgs/" + "one_neuron_loss" + ".jpg")
```

Multi neuron classifier sgd+

```
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
import time

class Exp:
    def __init__(self, exp, body, dependent_var, right_vars, right_params):
        self.exp = exp
        self.body = body
        self.dependent_var = dependent_var
        self.right_vars = right_vars
        self.right_params = right_params

#_______________________________  ComputationalGraphPrimer Class Definition  ________________________________

class ComputationalGraphPrimer(object):

    def __init__(self, *args, **kwargs ):
        if args:
            raise ValueError(
                   '''ComputationalGraphPrimer constructor can only be called with keyword arguments for
                      the following keywords: expressions, output_vars, dataset_size, grad_delta,
                      learning_rate, display_loss_how_often, one_neuron_model, training_iterations,
                      batch_size, num_layers, layers_config, epochs, and debug''')
        expressions = output_vars = dataset_size = grad_delta = display_loss_how_often = learning_rate = one_neu
        if 'one_neuron_model' in kwargs              :   one_neuron_model = kwargs.pop('one_neuron_model')
        if 'batch_size' in kwargs                    :   batch_size = kwargs.pop('batch_size')
        if 'num_layers' in kwargs                    :   num_layers = kwargs.pop('num_layers')
        if 'layers_config' in kwargs                 :   layers_config = kwargs.pop('layers_config')
        if 'expressions' in kwargs                   :   expressions = kwargs.pop('expressions')
        if 'output_vars' in kwargs                   :   output_vars = kwargs.pop('output_vars')
        if 'dataset_size' in kwargs                  :   dataset_size = kwargs.pop('dataset_size')
        if 'learning_rate' in kwargs                 :   learning_rate = kwargs.pop('learning_rate')
        if 'training_iterations' in kwargs           :   training_iterations = \
```

```
                                                        kwargs.pop('training_iterations')
if 'grad_delta' in kwargs                    :   grad_delta = kwargs.pop('grad_delta')
if 'display_loss_how_often' in kwargs        :   display_loss_how_often = kwargs.pop('display_loss_how_o
if 'epochs' in kwargs                        :   epochs = kwargs.pop('epochs')
if 'debug' in kwargs                         :   debug = kwargs.pop('debug')
if 'mu' in kwargs                            :   self.mu_multi = kwargs.pop('mu')
if len(kwargs) != 0: raise ValueError('''You have provided unrecognizable keyword args''')
self.one_neuron_model =  True if one_neuron_model is not None else False
if training_iterations:
    self.training_iterations = training_iterations
self.batch_size  =  batch_size if batch_size else 4
self.num_layers = num_layers
if layers_config:
    self.layers_config = layers_config
if expressions:
    self.expressions = expressions
if output_vars:
    self.output_vars = output_vars
if dataset_size:
    self.dataset_size = dataset_size
if learning_rate:
    self.learning_rate = learning_rate
else:
    self.learning_rate = 1e-6
if grad_delta:
    self.grad_delta = grad_delta
else:
    self.grad_delta = 1e-4
if display_loss_how_often:
    self.display_loss_how_often = display_loss_how_often
if dataset_size:
    self.dataset_input_samples  = {i : None for i in range(dataset_size)}
    self.true_output_vals       = {i : None for i in range(dataset_size)}
self.vals_for_learnable_params = None
self.epochs = epochs
if debug:
    self.debug = debug
else:
    self.debug = 0
self.independent_vars = None
self.gradient_of_loss = None
self.gradients_for_learnable_params = None
self.expressions_dict = {}
self.LOSS = []                               ##  loss values for all iterations of training
self.all_vars = set()
if (one_neuron_model is True) or (num_layers is not None):
```

```
        self.independent_vars = []
        self.learnable_params = []
    else:
        self.independent_vars = set()
        self.learnable_params = set()
    self.dependent_vars = {}
    self.depends_on = {}                        ##  See Introduction for the meaning of this
    self.leads_to = {}                          ##  See Introduction for the meaning of this
    self.device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

def parse_multi_layer_expressions(self):
    '''
    This method is a modification of the previous expression parser and meant specifically
    for the case when a given set of expressions are supposed to define a multi-layer neural
    network.  The naming conventions for the variables, which designate  the nodes in the layers
    of the network, and the learnable parameters remain the same as in the previous function.
    '''
    self.exp_objects = []
    self.layer_expressions = {i: [] for i in range(1, self.num_layers)}
    self.layer_exp_objects = {i: [] for i in range(1, self.num_layers)}
    ## A deque is a double-ended queue in which elements can inserted and deleted at both ends.
    all_expressions = deque(self.expressions)
    for layer_index in range(self.num_layers - 1):
        for node_index in range(self.layers_config[layer_index + 1]):
            self.layer_expressions[layer_index + 1].append(all_expressions.popleft())
    print("\n\nself.layer_expressions: ", self.layer_expressions)
    self.layer_vars = {i: [] for i in range(self.num_layers)}  # layer indexing starts at 0
    self.layer_params = {i: [] for i in range(1, self.num_layers)}  # layer indexing starts at 1
    for layer_index in range(1, self.num_layers):
        for exp in self.layer_expressions[layer_index]:
            left, right = exp.split('=')
            self.all_vars.add(left)
            self.expressions_dict[left] = right
            self.depends_on[left] = []
            parts = re.findall('([a-zA-Z]+)', right)
            right_vars = []
            right_params = []
            for part in parts:
                if part.startswith('x'):
                    self.all_vars.add(part)
                    self.depends_on[left].append(part)
                    right_vars.append(part)
                else:
                    if (self.one_neuron_model is True) or (self.num_layers is not None):
                        self.learnable_params.append(part)
                    else:
```

```
                    self.learnable_params.add(part)
                right_params.append(part)
            self.layer_vars[layer_index - 1] = right_vars
            self.layer_vars[layer_index].append(left)
            self.layer_params[layer_index].append(right_params)
            exp_obj = Exp(exp, right, left, right_vars, right_params)
            ##  when num_layers is defined and >0, the sequence of expression in
            ##  self.exp_objects would correspond to layers
            self.layer_exp_objects[layer_index].append(exp_obj)
        if self.debug:
            print("\n\nall variables: %s" % str(self.all_vars))
            print("\n\nlearnable params: %s" % str(self.learnable_params))
            print("\n\ndependencies: %s" % str(self.depends_on))
            print("\n\nexpressions dict: %s" % str(self.expressions_dict))

        for var in self.all_vars:
            if var not in self.depends_on:  # that is, var is not a key in the depends_on dict
                if (self.one_neuron_model is True) or (self.num_layers is not None):
                    self.independent_vars.append(var)
                else:
                    self.independent_vars.add(var)
        self.input_size = len(self.independent_vars)
        if self.debug:
            print("\n\nindependent vars: %s" % str(self.independent_vars))
        self.dependent_vars = [var for var in self.all_vars if var not in self.independent_vars]
        self.output_size = len(self.dependent_vars)
        self.leads_to = {var: set() for var in self.all_vars}
        for k, v in self.depends_on.items():
            for var in v:
                self.leads_to[var].add(k)
        if self.debug:
            print("\n\nleads_to dictionary: %s" % str(self.leads_to))
    print("\n\nself.layer_vars: ", self.layer_vars)
    print("\n\nself.layer_params: ", self.layer_params)
    print("\n\nself.layer_exp_objects: ", self.layer_exp_objects)

    ### Introduced in 1.0.5
    ##########################################################################################################
    ######################################### multi neuron model #############################################
def run_training_loop_multi_neuron_model(self, training_data):

    class DataLoader:
        def __init__(self, training_data, batch_size):
            self.training_data = training_data
            self.batch_size = batch_size
            self.class_0_samples = [(item, 0) for item in self.training_data[0]]
```

```
        self.class_1_samples = [(item, 1) for item in self.training_data[1]]

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

    def _getitem(self):
        cointoss = random.choice([0, 1])
        if cointoss == 0:
            return random.choice(self.class_0_samples)
        else:
            return random.choice(self.class_1_samples)

    def getbatch(self):
        batch_data, batch_labels = [], []
        maxval = 0.0
        for _ in range(self.batch_size):
            item = self._getitem()
            if np.max(item[0]) > maxval:
                maxval = np.max(item[0])
            batch_data.append(item[0])
            batch_labels.append(item[1])
        batch_data = [item / maxval for item in batch_data]
        batch = [batch_data, batch_labels]
        return batch

        ## We must initialize the learnable parameters

self.vals_for_learnable_params = {param: random.uniform(0, 1) for param in self.learnable_params}
self.bias = [random.uniform(0, 1) for _ in range(self.num_layers - 1)]
import copy
self.momentum_multi = copy.deepcopy(self.vals_for_learnable_params)
for i in self.momentum_multi.keys():
    self.momentum_multi[i] = 0
self.mu_bias_multi = [0 for _ in range(self.num_layers - 1)]
# self.mu_multi = 0.99

data_loader = DataLoader(training_data, batch_size=self.batch_size)
loss_running_record = []
i = 0
avg_loss_over_literations = 0.0
for i in range(self.training_iterations):
    data = data_loader.getbatch()
    data_tuples = data[0]
    class_labels = data[1]
    self.forward_prop_multi_neuron_model(data_tuples)
    predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_layers - 1]
```

```
            y_preds = [item for sublist in predicted_labels_for_batch for item in sublist]
            loss = sum([(abs(class_labels[i] - y_preds[i])) ** 2 for i in range(len(class_labels))])
            loss_avg = loss / float(len(class_labels))
            avg_loss_over_literations += loss_avg
            if i % (self.display_loss_how_often) == 0:
                avg_loss_over_literations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_literations)
                print("[iter=%d]  loss = %.4f" % (i + 1, avg_loss_over_literations))
                avg_loss_over_literations = 0.0
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)
        return loss_running_record

    def forward_prop_multi_neuron_model(self, data_tuples_in_batch):
        """

        See Slides 103 through 108 of Week 3 slides for the logic implemented  here.
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

        During forward propagation, we push each batch of the input data through the
        network.  In order to explain the logic of forward, consider the following network
        layout in 4 nodes in the input layer, 2 nodes in the hidden layer, and 1 node in
        the output layer.

                                 input

                                   x                                                x = node

                                   x         x|                                     | = sigmoid activation
                                                      x|
                                   x         x|

                                   x

                                layer_0    layer_1    layer_2


        In the code shown below, the expressions to evaluate for computing the
        pre-activation values at a node are stored at the layer in which the nodes reside.
        That is, the dictionary look-up "self.layer_exp_objects[layer_index]" returns the
        Expression objects for which the left-side dependent variable is in the layer
        pointed to layer_index.  So the example shown above, "self.layer_exp_objects[1]"
        will return two Expression objects, one for each of the two nodes in the second
        layer of the network (that is, layer indexed 1).
```

```
    The pre-activation values obtained by evaluating the expressions at each node are
    then subject to Sigmoid activation, followed by the calculation of the partial
    derivative of the output of the Sigmoid function with respect to its input.

    In the forward, the values calculated for the nodes in each layer are stored in
    the dictionary

                    self.forw_prop_vals_at_layers[ layer_index ]

    and the gradients values calculated at the same nodes in the dictionary:

                    self.gradient_vals_for_layers[ layer_index ]

    """
    self.forw_prop_vals_at_layers = {i: [] for i in range(self.num_layers)}
    self.gradient_vals_for_layers = {i: [] for i in range(1, self.num_layers)}
    for vals_for_input_vars in data_tuples_in_batch:
        self.forw_prop_vals_at_layers[0].append(vals_for_input_vars)
        for layer_index in range(1, self.num_layers):
            input_vars = self.layer_vars[layer_index - 1]
            if layer_index == 1:
                vals_for_input_vars_dict = dict(zip(input_vars, list(vals_for_input_vars)))
            output_vals_arr = []
            gradients_val_arr = []
            for exp_obj in self.layer_exp_objects[layer_index]:
                output_val = self.eval_expression(exp_obj.body, vals_for_input_vars_dict,
                                                  self.vals_for_learnable_params, input_vars)
                output_val = output_val + self.bias[layer_index - 1]
                ## apply sigmoid activation:
                output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))
                output_vals_arr.append(output_val)
                ## calculate partial of the activation function as a function of its input
                deriv_sigmoid = output_val * (1.0 - output_val)
                gradients_val_arr.append(deriv_sigmoid)
                vals_for_input_vars_dict[exp_obj.dependent_var] = output_val
            self.forw_prop_vals_at_layers[layer_index].append(output_vals_arr)
            self.gradient_vals_for_layers[layer_index].append(gradients_val_arr)

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

    See Slides 103 through 108 of Week 3 slides for the logic implemented  here.
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

    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 stores 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 la
    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_laye
            pred_err_backproped_at_layers[back_layer_index - 1] = backproped_error
            input_vars_to_layer = self.layer_vars[back_layer_index - 1]
            for j, var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]
                for i, param in enumerate(layer_params):
                    gradient_of_loss_for_param = input_vals_avg[i] * \
                                                 pred_err_backproped_at_layers[back_layer_index][j]
                    step = self.learning_rate * gradient_of_loss_for_param * deriv_sigmoid_avg[j] \
                           + self.mu_multi * self.momentum_multi[param]
                    self.vals_for_learnable_params[param] += step
                    self.momentum_multi[param] = step
            step = self.learning_rate * sum(pred_err_backproped_at_layers[back_layer_index]) \
                   * sum(deriv_sigmoid_avg) / len(deriv_sigmoid_avg) + self.mu_multi * self.mu_bias_multi[
                       back_layer_index - 1]
            self.bias[back_layer_index - 1] += step
            self.mu_bias_multi[back_layer_index - 1] = step


    def eval_expression(self, exp, vals_for_vars, vals_for_learnable_params, ind_vars=None):
        self.debug1 = False
        if self.debug1:
            print("\n\nSTEP1: [original expression] exp: %s" % exp)
        if ind_vars is not None:
            for var in ind_vars:
                exp = exp.replace(var, str(vals_for_vars[var]))
        else:
            for var in self.independent_vars:
                exp = exp.replace(var, str(vals_for_vars[var]))
        if self.debug1:
            print("\n\nSTEP2: [replaced ars by their vals] exp: %s" % exp)
        for ele in self.learnable_params:
            exp = exp.replace(ele, str(vals_for_learnable_params[ele]))
        if self.debug1:
            print("\n\nSTEP4: [replaced learnable params by their vals] exp: %s" % exp)
        return eval( exp.replace('^', '**') )


    def gen_training_data(self):
        """
        This 2-class dataset is used for the demos in the following Examples directory scripts:

                    one_neuron_classifier.py
                    multi_neuron_classifier.py
                    multi_neuron_classifier.py
```

```
        The classes are labeled 0 and 1.  All of the data for class 0 is simply a list of
        numbers associated with the key 0.  Similarly all the data for class 1 is another list of
        numbers associated with the key 1.

        For each class, the dataset starts out as being standard normal (zero mean and unit variance)
        to which we add a mean value of 2.0 for class 0 and we add mean value of 4 to the square of
        the original numbers for class 1.
        """
        num_input_vars = len(self.independent_vars)
        training_data_class_0 = []
        training_data_class_1 = []
        for i in range(self.dataset_size //2):
            # Standard normal means N(0,1), meaning zero mean and unit variance
            # Such values are significant in the interval [-3.0,+3.0]
            for_class_0 = np.random.standard_normal( num_input_vars )
            for_class_1 = np.random.standard_normal( num_input_vars )
            # translate class_1 data so that the mean is shifted to +4.0 and also
            # change the variance:
            for_class_0 = for_class_0 + 2.0
            for_class_1 = for_class_1 * 2 + 4.0
            training_data_class_0.append( for_class_0 )
            training_data_class_1.append( for_class_1 )
        training_data = {0 : training_data_class_0, 1 : training_data_class_1}
        return training_data

#_________________________  End of ComputationalGraphPrimer Class Definition ___________________________


#_______________________________    Test code follows    _________________________________

if __name__ == '__main__':
    import random
    import numpy

    seed = 0
    random.seed(seed)
    numpy.random.seed(seed)
    # import sys
    # sys.path.append("/home/yangbj/695/hw3/ComputationalGraphPrimer-1.0.8/")
    # from ComputationalGraphPrimer import *

    cgp1 = 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,
    #               learning_rate = 5 * 1e-2,
    training_iterations=40000,
    batch_size=8,
    display_loss_how_often=100,
    debug=True,
    mu = 0.99
)

cgp1.parse_multi_layer_expressions()

training_data = cgp1.gen_training_data()
loss_running_record1 = cgp1.run_training_loop_multi_neuron_model(training_data)


cgp2 = 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,
    #               learning_rate = 5 * 1e-2,
    training_iterations=40000,
    batch_size=8,
    display_loss_how_often=100,
    debug=True,
    mu = 0
)

cgp2.parse_multi_layer_expressions()
# cgp2.display_network2()
loss_running_record2 = cgp2.run_training_loop_multi_neuron_model(training_data)

plt.figure()
plt.plot(loss_running_record1)
plt.plot(loss_running_record2)
plt.show()
plt.savefig("/home/yangbj/695/hw3/imgs/" + "multi_neuron_loss" + ".jpg")
```

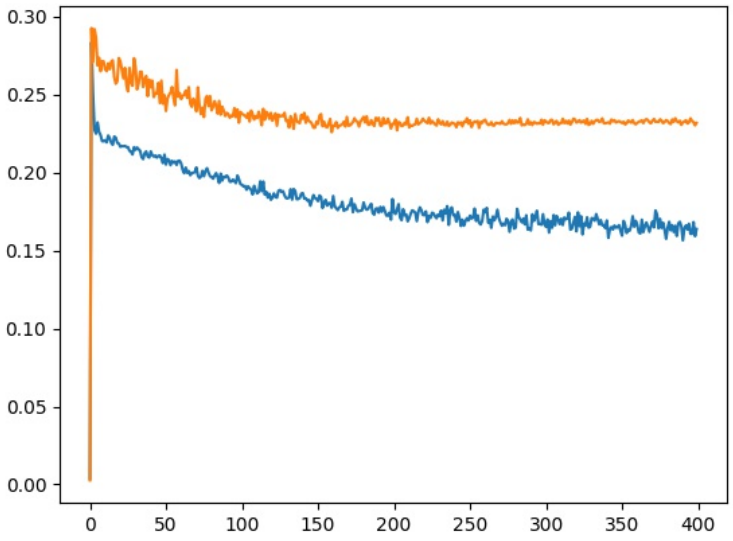

Figure 3: one neuron loss

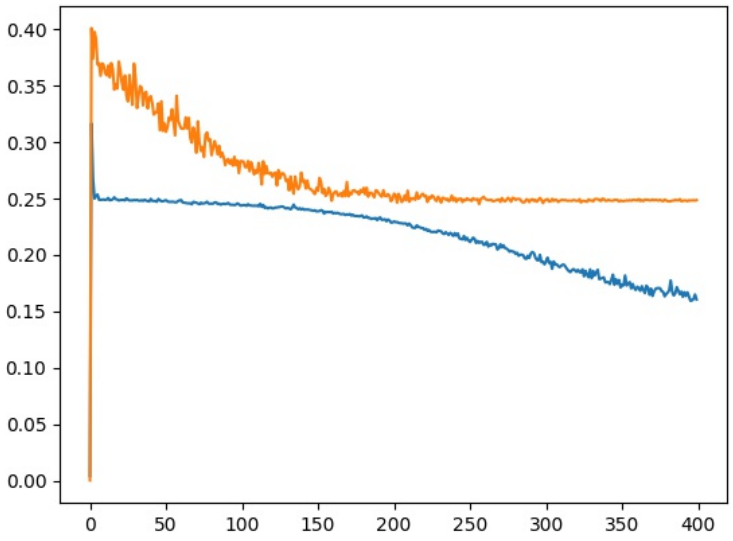

Figure 4: multi neuron loss

The results in Fig. 3 and Fig. 4 above shows that:
(1)After adding momentum to sgd, the loss typically will drop more, which will result in better performance for the classifier.
(2)After adding momentum to sgd, the optimizer sgd+ tends to be more effective even after 300 iterations of training, giving more potential to optimize the model. However, without momentum, the model quickly converged with a higher loss and poorer performance.

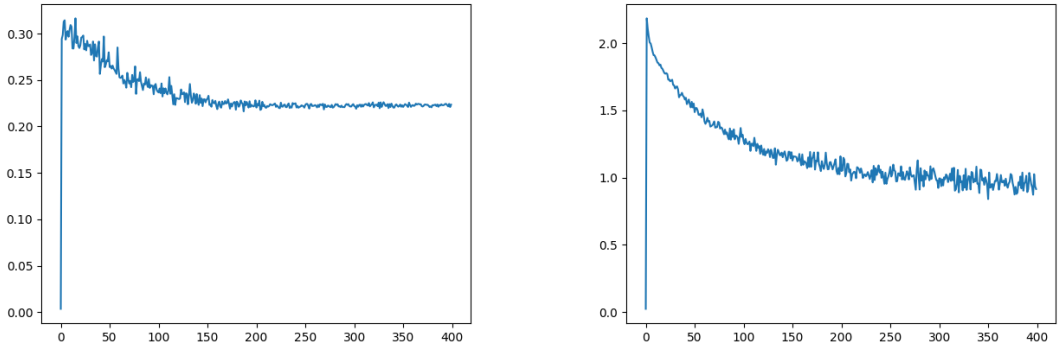

Figure 5: one neuron classifier and verified with torchnn

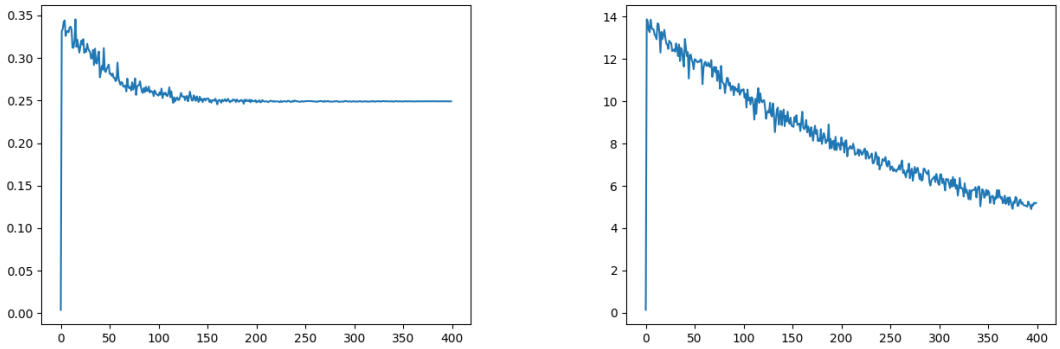

Figure 6: multi neuron classifier and verified with torchnn

Compare with torch.nn The results in Fig. 5 and Fig. 6 above shows that:
torch.nn is more efficient in optimizing the parameters because even after 400 iterations of training, the loss is still decreasing. However, the optimization of the handcrafted network is slow and the network nearly converged and stuck after 200 iterations.

4 Lessons Learned

The hurdles faced and the techniques employed to overcome them:

(1) When initializing an dict class by copying another dict class, we need to use deep copy or the original dict will be changed later; For example,

```
import copy
self.momentum_multi = copy.deepcopy(self.vals_for_learnable_params)
for i in self.momentum_multi.keys():
    self.momentum_multi[i] = 0
```

In this code, if I just simply use "$self.momentum\_multi = self.vals\_for\_learnable\_params$", that would not work. Anyway, using a similar intialization method as the $self.vals\_for\_learnable\_params$ is also acceptable, like "$self.vals\_for\_learnable\_params = \{param : random.uniform(0, 1) for param in self.learnable\_params\}$"

(2) The bias should also be updated using momentum;

(3) For multi neuron classifier, the momentum for the bias should be a num of layers long list or array. However, for one neuron classifier, it can only be a scalar.

5 Suggested Enhancements

To the best of my understanding, we only did a few modifications to the optimizer, which is, adding a momentum term. It could be a good exercise for better understanding of the momentum and regular learning procedure, However, I think it can be improved to incorporate more tasks such as third order momentum or something like that.