ECE695DL: Homework 3

Jorge Loria

1 March 2021

one_neuron_classifier_sgd_plus.py

```
#!/usr/bin/env python
import sys
import os

## Based on the One neuron classifier found in the examples folder of
## ComputationalGraphPrimer 1.0.5

dir_comprimer = "D:/Documents/Clases_2021-Spring/hw3/
ComputationalGraphPrimer-1.0.5"
if os.path.isdir(dir_comprimer):
    sys.path.append(dir_comprimer)
else:
    print('cool, hope it exists')


"""
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 numpy as np
import operator
import random
```

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

from ComputationalGraphPrimer import *

class ComputationalPrimerModifidiedJorgeOneNeuron(ComputationalGraphPrimer):

    def __init__(self, *args, **kwargs):
        self.loss_running_record = []
        super(ComputationalPrimerModifidiedJorgeOneNeuron, self)
        .__init__(*args, **kwargs)

    ## Based on the function from the ComputationalGraphPrimer 1.0.5
    ## I tried using the inheritance ideas, but loss running loss record
    ## is not defined in a nice way to be able to plot other stuff
    at the same time...
    def run_training_loop_one_neuron_model(self,mu=0.5):
        self.mu = mu
        # self.nu_t_params = 0 ## fix dims
        # self.nu_t_bias = 0 ## fix dims
        self.nu_t_params = {param: 0 for param in self.learnable_params}
        ## based on the original code
        self.nu_t_bias = 0 ## fix dims
        self.epoch = 0
        # super(ComputationalPrimerModifidiedJorgeOneNeuron,
        self).run_training_loop_one_neuron_model()


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

        class DataLoader:
           def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in
                self.training_data[0]]
                self.class_1_samples = [(item, 1) for item in
                self.training_data[1]]
           def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])
           def _getitem(self):
                cointoss = random.choice([0, 1])
                if cointoss == 0:
```

```
                return random.choice(self.class_0_samples)
            else:
                return random.choice(self.class_1_samples)

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

```
        # plt.figure()
        # plt.plot(self.loss_running_record)
        # plt.show()

            ## Based on the function from the ComputationalGraphPrimer 1.0.5
    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.
        """
        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):
            # self.nu_t_params =
            self.nu_t_params[param] = self.nu_t_params[param] * self.mu +
            self.learning_rate * y_error *
            vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid

            self.vals_for_learnable_params[param] += self.nu_t_params[param]
        self.nu_t_bias = self.nu_t_bias * self.mu + self.learning_rate *
        y_error * deriv_sigmoid
        self.bias += self.nu_t_bias
        self.epoch += 1

lr = 1e-3

cgp0 = ComputationalPrimerModifidiedJorgeOneNeuron(
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = lr,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 10,
               display_loss_how_often = 100,
               debug = True,
      )


cgp0.parse_expressions()

#cgp.display_network1()
# cgp0.display_network2()

cgp0.gen_training_data()

cgp1 = ComputationalPrimerModifidiedJorgeOneNeuron(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = lr,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 10,
               display_loss_how_often = 100,
               debug = True,
      )


cgp0.parse_expressions()

# cgp.display_network1()
# cgp0.display_network2()
## set the seed first here,
## this way we can compare the actual results
seed = 0
random.seed(seed)
np.random.seed(seed)

cgp0.gen_training_data()

import matplotlib.pyplot as plt
cgp0.run_training_loop_one_neuron_model(mu=0.99)
l1 = cgp0.loss_running_record
```

```
cgp1.parse_expressions()

# cgp1.display_network1()
# cgp1.display_network2()


## set the seed second here,
## this way we can compare the actual results!!
## note that when we use momentum equal to zero
## is equal as doing the usual sgd, without the plus

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

cgp1.gen_training_data()
cgp1.run_training_loop_one_neuron_model(mu=0)

l2 = cgp1.loss_running_record

plt.plot(l1, label='Loss of SGD+')
plt.plot(l2, label='Loss of SGD not plus')
plt.title('Comparison of SGD + and SGD\nfor one neuron')
plt.xlabel('Iterations/100')
plt.ylabel('Loss :(')


plt.legend()


plt.savefig('one_neuron_loss.jpg')
```

multi_neuron_classifier_sgd_plus.py

```
#!/usr/bin/env python
import sys
import os
```

```
## Based on the One neuron classifier found in the examples folder of
## ComputationalGraphPrimer 1.0.5

dir_comprimer = "D:/Documents/Clases_2021-Spring/hw3/
ComputationalGraphPrimer-1.0.5"
if os.path.isdir(dir_comprimer):
    sys.path.append(dir_comprimer)
else:
    print('cool, hope it exists')


"""
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 numpy as np
import operator
import random


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

from ComputationalGraphPrimer import *

class ComputationalPrimerModifidiedJorgeMuchNeurons(ComputationalGraphPrimer):

    def __init__(self, *args, **kwargs):
        super(ComputationalPrimerModifidiedJorgeMuchNeurons, self).
        __init__(*args, **kwargs)

    def run_training_loop_multi_neuron_model(self, mu=0.5):
        self.mu = mu
        self.nu_t_params = {param: 0 for param in self.learnable_params}
        ## based on the original code
        self.nu_t_bias = [0 for _ in range(self.num_layers-1)]
        self.epoch = 0
        self.loss_running_record = []
```

```
training_data = self.training_data
self.vals_for_learnable_params = {param: random.uniform(0, 1)
for param in self.learnable_params}
self.bias = [random.uniform(0, 1) for _ in range(self.num_layers - 1)]
class DataLoader:
    def __init__(self, training_data, batch_size):
        self.training_data = training_data
        self.batch_size = batch_size
        self.class_0_samples = [(item, 0) for item in self.training_data[0]]
        self.class_1_samples = [(item, 1) for item in self.training_data[1]]
    def __len__(self):
        return len(self.training_data[0]) + len(self.training_data[1])
    def _getitem(self):
        cointoss = random.choice([0, 1])
        if cointoss == 0:
             return random.choice(self.class_0_samples)
        else:
            return random.choice(self.class_1_samples)

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

data_loader = DataLoader(self.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)
        self.loss_running_record = 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 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 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]
                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.nu_t_params[param] = self.nu_t_params[param]*self.mu +
                    step
                    self.vals_for_learnable_params[param] +=
                    self.nu_t_params[param]#/(1+self.epoch)
            bias_step = self.learning_rate *
            sum(pred_err_backproped_at_layers[back_layer_index]) *
            sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)
            self.nu_t_bias[back_layer_index-1] =
            self.nu_t_bias[back_layer_index-1]*self.mu + bias_step
            self.bias[back_layer_index-1]
            +=self.nu_t_bias[back_layer_index-1]#/(1+self.epoch)
        self.epoch += 1
    ########################################################################


lr = 1e-3
cgpMulti = ComputationalPrimerModifidiedJorgeMuchNeurons(
               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 = lr,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 10,
               display_loss_how_often = 100,
               debug = True,
      )
```

```
cgpMulti.parse_multi_layer_expressions()

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

cgpMulti.gen_training_data()

cgpMulti.run_training_loop_multi_neuron_model(mu=0.99)

l1 = cgpMulti.loss_running_record

####


cgpMulti0 = ComputationalPrimerModifidiedJorgeMuchNeurons(
               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 = lr,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 10,
               display_loss_how_often = 100,
               debug = True,
      )

cgpMulti0.parse_multi_layer_expressions()

# cgp.display_network1()
# cgpMulti0.display_network2()

## as in the one neuron loss, we set the same seed so the comparison is
## fair, and we use the same data to compare :D
seed = 0
random.seed(seed)
np.random.seed(seed)

cgpMulti0.gen_training_data()

cgpMulti0.run_training_loop_multi_neuron_model(mu=0)
```

```
l2 = cgpMulti0.loss_running_record

import matplotlib.pyplot as plt


plt.figure()
plt.plot(l1, label='Loss of SGD+')
plt.plot(l2, label='Loss of SGD not plus')
plt.title('Comparison of SGD + and SGD\nfor several neurons')
plt.xlabel('Iterations/100')
plt.ylabel('Loss :(')


plt.legend()


plt.savefig('multi_neuron_loss.jpg')
```

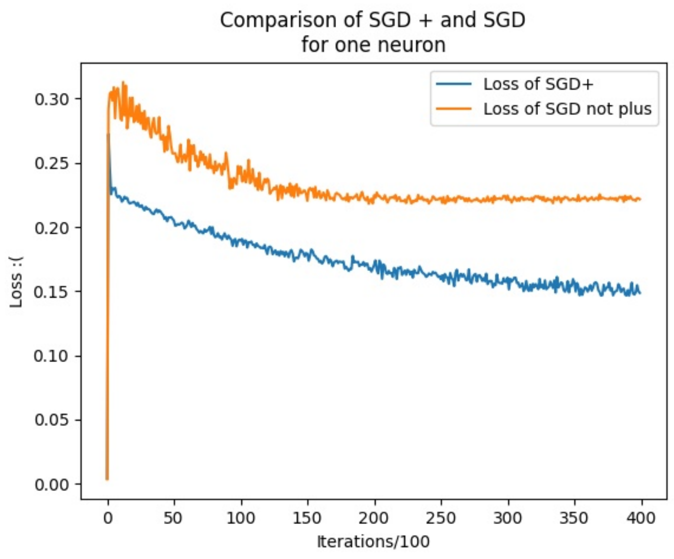

Figure 1: one_neuron_loss.jpg

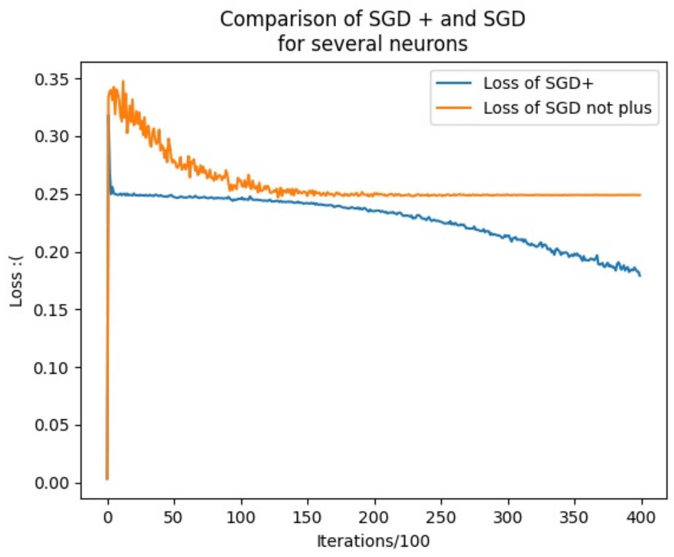

Figure 2: multi_neuron_loss.jpg