ECE 60146: Deep Learning Homework 3

Mehmet Berk Sahin, sahinm@purdue.edu, 34740048

January 30, 2023

1 Introduction

The purpose of this homework is to develop a greater understanding of the step size optimization logic that can be applied to the training stage of deep neural networks. In Section 2, I showed the results of the programs after running them and I made a few comments on them to show you I completed the instructions. In Section 3, I explained my class definition (source code) and discussed what modifications I did upon `ComputationlGraphPrimer` class to implement `SGD+` and `Adam` optimization algorithms. You can see the necessary figures for the homework in that section. While doing this homework, I utilized Prof. Kak's slides on Autograd.

2 Becoming Familiar with the Primer

In this section, I mentioned the steps I followed for Section 2 of the homework. And I put the update rules for `SGD+` and `Adam` for future reference in Section 3. First, I downloaded `tar.gz` and install `ComputationalGraphPrimer` version 1.0.9 from this link, which is shared by TA. I did the setup as explained in Prof. Kak's documentation page for the Primer. Then, under the Examples directory, I executed the following scripts:

```
python3 one_neuron_classifier.py
```

```
python3 multi_neuron_classifier.py
```

The final output of these scripts is plots of the training loss history. They can be seen below:

(a) Training of one neuron classifier

(b) Training of multi neuron classifier

Figure 1: Results of handcrafted classifiers

I executed the following script, which is based on `PyTorch`, to compare the performance of Prof. Kak's handcrafted package with `torch.nn` package:

```
python3 verify_with_torchnn.py
```

The result I got is given below:

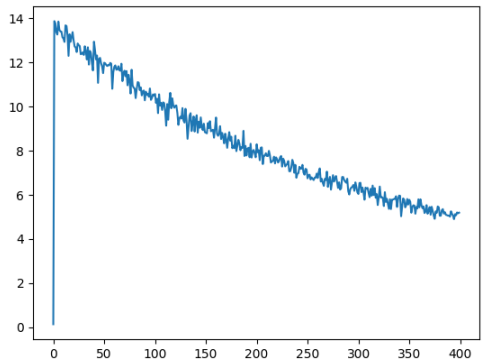


Figure 2: Training of multi neuron classifier (the same one in Figure 1b) with `torch.nn`

Comparing Figure 1b with Figure 2, one can say that `torch.nn` is more efficient because, after 400 iterations, it still continues to decrease its loss whereas handcrafted multi-neuron classifier already converges at that point. Another observation is that the scale of the loss functions is not the same. After these, I also compared the result of `torch.nn` for a single neuron model with Figure 1a. To do so, I uncommented the lines for single neuron case in the source code and commented the lines for multi neuron case and run the following script:

```
python3 verify_with_torchnn.py
```

The one neuron classifier training plot with `torch.nn` package is given below:

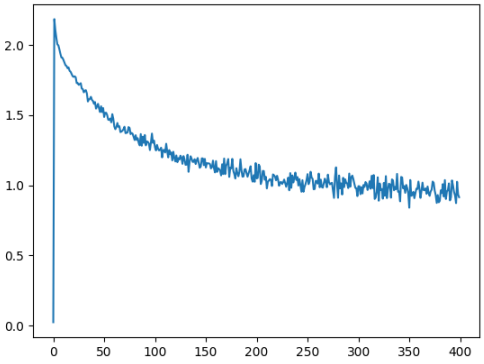


Figure 3: Training of one neuron classifier (the same one in Figure 1a) with `torch.nn`

It also seems better than the handcrafted package because it keeps learning over 400 iterations whereas the handcrafted single-neuron classifier stopped learning after roughly 220 iterations. And the slope at the initial points are higher in Figure 3 than in 1a. Also, scaling is different here

too. To sum up, there are significant improvements in the optimization of the loss functions with the codes implemented in `torch.nn`. This achievement can be attributed to the step optimization methods like `SGD` with momentum and `Adam`, which we will discuss next.

One observation that I did in the code in Version 1.0.9 of the Primer is it does NOT use any step optimization in SGD updates. It only performs Vanilla `SGD`, which does the following:

$$p^i_{t+1} = p^i_t - \alpha * g^i_{t+1} \tag{1}$$

where p^i_t denotes the i^{th} learnable parameter, $i \in \{1, 2, ..., N\}$ and N is the number of learnable parameters, α is the learning rate (adjusts the size of the update), and g^i_{t+1} is the gradient of the loss with respect to i^{th} learnable parameter. t subscripts denote the values of these parameters at time t. In this homework, our goal is to implement step-size optimization to training with the one-neuron and multi-neuron networks in `ComputationalGraphPrimer` package. We are asked to implement two step-optimizers: `SGD+` (SGD with momentum) and `Adam` (Adaptive Moment Estimate).

SGD with Momentum (SGD+)

The basic idea of SGD with momentum is by utilizing previous iterates, we are aiming for faster convergence to global minima or local minima which is close to global minima. To perform `SGD+`, following recursive formula is implemented:

$$\begin{aligned} v^i_{t+1} &= \mu * v^i_t + g^i_{t+1} \\ p^i_{t+1} &= p^i_t - \alpha * v^i_{t+1} \end{aligned} \tag{2}$$

Parameter names are the same as in the `SGD` except now instead of sole gradient, we have v^i_t, which is the weighted summation of the current gradient of the i^{th} learnable parameter with its previous time step updates. This is useful when the loss function is not well-behaved and this may cause lot of oscillations in the SGD steps. Adding momentum term to the SGD help optimizer accelerate gradient vectors in the right directions thus, result in faster convergence than SGD.

Adaptive Moment Estimation (Adam)

Nowadays, a lot of Deep Learning practitioners use `Adam` algorithm as a step-optimizer. The key idea behind Adam is it takes the best parts of `AdaGrad` and `RMSprop` and combines them under a single, more efficient algorithm. The update rule is given below:

$$\begin{aligned} m^i_{t+1} &= \beta_1 * m^i_t + (1 - \beta_1) * g^i_{t+1} \\ v^i_{t+1} &= \beta_2 * v^i_t + (1 - \beta_2) * (g^i_{t+1})^2 \\ p^i_{t+1} &= p^i_t - \alpha * \frac{\hat{m}^i_{t+1}}{\sqrt{\hat{v}^i_{t+1} + \epsilon}} \end{aligned} \tag{3}$$

The parameters used previously have the same meanings. β_1 and β_2 controls the decay rates for the moments and they are generally set to 0.9 and 0.99, respectively. And there are two variables

having hats. They are defined as

$$\hat{m}_t^i = \frac{m_t^i}{1-\beta_1^t}$$
$$\hat{v}_t^i = \frac{v_t^i}{1-\beta_2^t} \tag{4}$$

Adam is plug-and-play algorithm. That is, it does not require too much hyperparameter optimization. It adapts itself to the problem and it is recommended to be used when the gradients are sparse and/or noisy. Adam algorithm basically adapts itself to the gradient descent after every step so that it is controlled and remain unbiased in finding the minima. This feature makes it overcome bad local minimas. Although Adam is efficient algorithm, it biases the values of the two moments toward zero that is why updates in (4) are done. I studied these step-optimizers from Prof. Kak's Week 3 Slides on Autograd.

3 Programming Task

This part is allocated to the programming task in the homework, which is two-fold: implementing SGD+ and Adam based on the basic (vanilla) SGD in `one_neuron_classifier.py` and `multi_neuron_classifier.py`. I studied the logic of `one_neuron_classifier.py` and `multi_neuron_classifier.` explained in Prof. Kak's slides. Then, instead of overwriting the main module file `ComputationalGraphPrimer.py`, I created a subclass, which is called `MyComputationalGraphPrimer`, and overwrote the relevant functions. I will first show the general structure of my source code and then I will get into the details of the functions I defined. Lastly, I discuss the results and compare the performance of different step-optimizers. The general structure of my source code can be seen below:

```
from ComputationalGraphPrimer import *
import random
import numpy as np
import operator
import matplotlib.pyplot as plt


class MyComputationalGraphPrimer(ComputationalGraphPrimer):

    def __init__(self, optimizer='sgd', momentum=None, beta1=None, beta2=None, *args, **kwargs):...

    # training loop for one neuron classifier
    def run_training_loop_one_neuron_model(self, training_data):...

    # backpropagation and update of one neuron classifier
    def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid, iter):...

    # training loop for multi neuron classifier
    def run_training_loop_multi_neuron_model(self, training_data):...

    # backpropagation and update of multi neuron classifier
    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels, iter):...

# compares SGD, SGD+ and Adam with one neuron classifier
def one_neuron_experiment():...

# compares SGD, SGD+ and Adam with multi neuron classifier
def multi_neuron_experiment():...

# main code
if __name__ == "__main__":...
```

Figure 4: General structure of my source code

I defined `MyComputationalGraphPrimer` class as a subclass of `ComputationalGraphPrimer`. I overwrite 5 functions to implement step-optimizers. They can be seen in Figure 4. Instead of defining new functions, it is a better idea to overwrite the existing functions than modify the

original module as long as their intended behaviors are not changed. Because modifying the original module may give unexpected results, which can be very difficult to debug sometimes. Other than 5 instance methods that I overwrote, I defined two functions outside the class definition. They perform the training of one-neuron and multi-neuron classifiers and plot the results. Lastly, under the if statement, I run the main code, which I will show at the end. Let's start with the initialization function.

3.1 `MyComputationalGraphPrimer.__init__()`

Henceforth, whenever I say superclass it refers to `ComputationlGraphPrimer` class and subclass refers to `MyComputationlGraphPrimer` class, which is defined by me in Figure 4. Since we are asked to modify the superclass to implement step-optimizers such as SGD+ and Adam, we should have additional parameters like β_1, β_2, etc. Thus, I redefined the initialization class for my subclass definition, which can be seen on Figure 5.

```
# I implemented this function (MBS)
def __init__(self, optimizer='sgd', momentum=None, beta1=None, beta2=None, *args, **kwargs):
    super(MyComputationalGraphPrimer, self).__init__(*args, **kwargs)
    # set the parameters that are entered
    if optimizer:
        self.optimizer = optimizer
    if momentum:
        self.momentum = momentum
    if beta1:
        self.beta1 = beta1
    if beta2:
        self.beta2 = beta2
    # set necessary parameters depending on the optimizer
    if self.optimizer == "sgd+":
        # Keeps the updates for momentum SGD
        self.step_sizes = None
        self.bias_m = None
    elif self.optimizer == "adam":
        # bias corrected moments
        self.moment1 = None # corresponds to m
        self.moment2 = None # corresponds to v
        self.bias_m1 = None
        self.bias_m2 = None
```

Figure 5: Initialization function for `MyComputationalGraphPrimer`

In the first line under the function, I called the initialization method of the super class to inherit all the attributes and perform the actions done in that function. In addition to them, I also defined new parameters specialized to the optimizer, which can be 'sgd', 'sgd+' or 'adam'. If the user does not enter any optimizer, by default model will be run with Vanilla SGD. Note that when I called the superclass' init function, I also entered the `*args` and `**kwargs`, which makes my function to utilize the same parameters that are also used by the superclass. Other than these explanations, code is self-explanatory at this section.

3.2 `MyComputationalGraphPrimer.run_training_loop_one_neuron_model()`

This instance method is defined to modify the training loop of one neuron model for step-optimizers. I do not put the entire code for this function because I copy and paste the original functon from

the superclass and did some modifications. In the code, I pointed out these modifications via comments. So, I will only show the modifications that I made and for the rest, you can check the source code. I added the following code snippet to the beginning of the function:

```
######################################################################
# initialize the previous moments or step sizes (MBS)
if self.optimizer == "sgd+":
    self.step_sizes = {param_: 0 for param in self.learnable_params}
    self.bias_m = 0
elif self.optimizer == "adam":
    self.moment1 = {param_: 0 for param in self.learnable_params}
    self.moment2 = {param: 0 for param in self.learnable_params}
    self.bias_m1 = self.bias_m2 = 0
######################################################################
```

Figure 6: First modification to `run_training_loop_one_neuron_model`

This code basically generates dictionary whose keys are the name of the learnable parameters and whose values are the value of moment in the previous iteration. For example, in SGD+ update rule (2), v_t^i values are kept in `self.step_sizes` for each learnable parameter. This is also true for the biases and moments in Adam optimizer. Second modification I did is given below:

```
    ######################################################################
    # runs the backpropagation algorithm (I added iteration number as a parameter, MBS)
    self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg,
                                                     deriv_sigmoid_avg, iter=i+1)
    ######################################################################

######################################################################
# I commented these plots to display them later (MBS)
#plt.figure()
#plt.plot(loss_running_record)
#plt.show()
return loss_running_record # returns the loss history (MBS)
######################################################################
```

Figure 7: Second modification to `run_training_loop_one_neuron_model`

As you noticed, there is an indentation difference because the first line is in the training loop. It is called lastly to perform a backpropagation algorithm for one neuron model. To calculate the definition in (4), I needed to keep track of the iteration count so I entered that as a parameter to backpropagation function. Moreover, I wanted to display the plots in one figure instead of displaying the seperately so I commented the plot functions and returned the training loss history for later use.

3.3 `MyComputationalGraphPrimer.backprop_and_update_params_one_neuron_model()`

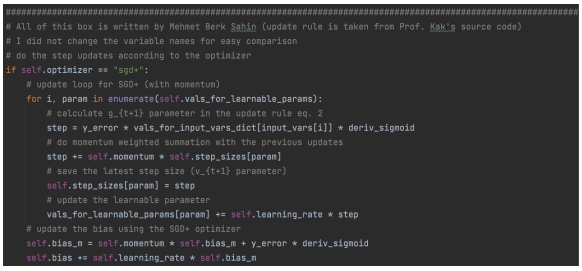
This instance method has the implementation of backpropagation algorithm for one neuron case and it performs updates for learnable parameters. Although I did not change the backpropagation algorithm, I added new step update features for different optimizers. Since the picture is too large to put here at once, I decided to explain the code step-by-step.

```
# backpropagation and update of one neuron classifier (I added iter as a parameter, MBS)
def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid, iter):
    input_vars = self.independent_vars
    vals_for_input_vars_dict = dict(zip(input_vars, vals_for_input_vars))
    vals_for_learnable_params = self.vals_for_learnable_params
```

Figure 8: `backprop_and_update_params_one_neuron_model` part 1

This part is identically the same as the superclass' function. It basically creates a dictionary whose keys are learnable parameter names and whose values are the values of these learnable parameters.

Then, implemented backpropagation under 3 if-else statements, which are specific to different optimizers.



```
##########################################################################################################
# All of this box is written by Mehmet Berk Sahin (update rule is taken from Prof. Kak's source code)
# I did not change the variable names for easy comparison
# do the step updates according to the optimizer
if self.optimizer == "sgd+":
    # update loop for SGD+ (with momentum)
    for i, param in enumerate(self.vals_for_learnable_params):
        # calculate g_{t+1} parameter in the update rule eq. 2
        step = y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
        # do momentum weighted summation with the previous updates
        step += self.momentum * self.step_sizes[param]
        # save the latest step size (v_{t+1} parameter)
        self.step_sizes[param] = step
        # update the learnable parameter
        vals_for_learnable_params[param] += self.learning_rate * step
    # update the bias using the SGD+ optimizer
    self.bias_m = self.momentum * self.bias_m + y_error * deriv_sigmoid
    self.bias += self.learning_rate * self.bias_m
```

Figure 9: First modification to `backprop_and_update_params_one_neuron_model` (part 2)

In Figure 9, you can see the first if statement and if you notice the comments, this is the first modification I did on `backprop_and_update_params_one_neuron_model`. It checks whether the optimizer is SGD+ (SGD with momentum). If so, it performs (2) recursive formula. Here, note that `self.step_sizes[param]` is equivalent to v_t^i, `self.momentum` is equivalent to μ and `step` corresponds to g_{t+1}^i in (2). The same procedure is implemented for bias terms as well outside the for loop. You can also check the comments.

```
elif self.optimizer == "adam":
    # update loop for Adam
    for i, param in enumerate(self.vals_for_learnable_params):
        # calculate g_{t+1} in the update rule eq. 3
        step = y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
        # do moment updates m_{t+1} and v_{t+1}, respectively
        self.moment1[param] = self.beta1 * self.moment1[param] + (1 - self.beta1) * step
        self.moment2[param] = self.beta2 * self.moment2[param] + (1 - self.beta2) * step ** 2
        # do bias correction for moments eq. 38 in Prof. Kak's Autograd slide
        m_unit = self.moment1[param] / (1 - self.beta1 ** iter)
        v_unit = self.moment2[param] / (1 - self.beta2 ** iter)
        # update the learnable parameters with bias-corrected moments
        self.vals_for_learnable_params[param] += self.learning_rate * m_unit / ((v_unit + 1e-8) ** 0.5)
    # calculate g_{t+1} in the update rule eq. 3 for bias term
    step = y_error * deriv_sigmoid
    # moment updates and bias corrections for the bias term
    self.bias_m1 = self.beta1 * self.bias_m1 + (1 - self.beta1) * step
    self.bias_m2 = self.beta2 * self.bias_m2 + (1 - self.beta2) * step ** 2
    m_unit = self.bias_m1 / (1 - self.beta1 ** iter)
    v_unit = self.bias_m2 / (1 - self.beta2 ** iter)
    # update the bias learnable parameter
    self.bias += self.learning_rate * m_unit / ((v_unit + 1e-8) ** 0.5)
```

Figure 10: Second modification to `backprop_and_update_params_one_neuron_model` (part 3)

If the optimizer is not chosen as 'sgd+', instead it is picked as 'adam', then the program gets into the if statement implemented in Figure 10. In that code snippet, `self.moment1[param]`, `self.moment2[param]`, `m_unit`, `v_unit`, `self.beta1`, `self.beta2` and `step` correspond to m_{t+1}^i, v_{t+1}^i, $\hat{m}_t^i$, $\hat{v}_t^i$, β_1, β_2, and g_{t+1}^i, respectively. Exactly the same updates as in (3) and (4) are followed for learnable parameters including the bias term in the implementation. You can check the comments.

```
else:
    # Do normal SGD (This part is same as super class' SGD implementation)
    for i, param in enumerate(self.vals_for_learnable_params):
        step = self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
        self.vals_for_learnable_params[param] += step
    self.bias += self.learning_rate * y_error * deriv_sigmoid
##########################################################################################################
```

Figure 11: `backprop_and_update_params_one_neuron_model` part 4 (no modification)

If the optimizer is neither SGD+ nor Adam, it is accepted as Vanilla SGD by default and classic

SGD update in (1) is performed. This part of the code was not modified. At this point modifications for the one-neuron classifier are completed. Let's discuss the modifications for the multi-neuron classifier.

3.4 `MyComputationalGraphPrimer.run_training_loop_multi_neuron_model`

Modifications for multi-neuron classifier are similar to the one-neuron case. They are just the generalizations of the single neuron case. Again, instead of putting the all code snippet I will go step-by-step and show you the modified sections of the code. For the full version, you can check the source code.

At the beginning of the function, I initialized the memory for moments of the learnable parameters as follows:

```
##########################################################################
# Initializes the necessary parameters for each learnable parameter
if self.optimizer == "sgd+":
    self.step_sizes = {param: 0 for param in self.learnable_params}
    self.bias_m = [0 for _ in range(self.num_layers - 1)]
elif self.optimizer == "adam":
    self.moment1 = {param: 0 for param in self.learnable_params}
    self.moment2 = {param: 0 for param in self.learnable_params}
    self.bias_m1 = [0 for _ in range(self.num_layers - 1)]
    self.bias_m2 = [0 for _ in range(self.num_layers - 1)]
##########################################################################
```

Figure 12: First modification to `run_training_loop_multi_neuron_model`

I kept all the moments for learnable parameters in a dictionary whose keys and values are learnable parameter names and their moment values, respectively. Different than the one-neuron case, since there are multi-layers, I kept a list of moments for biases for each layer because there is a single bias per layer.

```
    ##########################################################################################################
    # runs backpropagation algorithm for multi neuron case (I added iter parameter, MBS)
    self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels, iter=i+1)
    ##########################################################################################################
##########################################################################
# I commented these to diplay them later (MBS)
#plt.figure()
#plt.plot(loss_running_record)
#plt.show()
return loss_running_record # return the loss (MBS)
##########################################################################
```

Figure 13: Second modification to `run_training_loop_multi_neuron_model`

The second modification I did is I added the `iter` parameter to the backpropagation function to calculate $\hat{m}_t^i$ and $\hat{v}_t^i$ properly. Lastly, I commented on the plotting functions and return the loss history to display the loss history of the optimizers together in the main code.

3.5 `MyComputationalGraphPrimer.backprop_and_update_params_multi_neuron_model`

In this instance method, I modified the update rule according to the entered optimizer method which can be SGD+ or Adam for the multi-neuron case. Here, I only showed the modifications and I copied the rest from the original implementation in the superclass. You can check the full version from my source code.

The for loop in Figure 14 iterates through the learnable parameters in the current layer which is another for loop but is not visible in the code snippet. The naming of the parameters is the same as in the single-neuron case and the update rules are the same. In fact, the implementation is almost

```
##########################################################################################################
    # I modified this section of the code (MBS)
    for i, param in enumerate(layer_params):
        # calculate g_{t+1} in the update rule eq. 2
        step = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j] * deriv_sigmoid_avg[j]
        if self.optimizer == "sgd+":
            # calculate the momentum weighted step size
            self.step_sizes[param] = self.momentum * self.step_sizes[param] + step
            # save the last step size, v_{t+1}
            step = self.step_sizes[param]
        elif self.optimizer == "adam":
            # calculate and save moments m_{t+1} and v_{t+1}, respectively (eq. 3 in hw)
            self.moment1[param] = self.beta1 * self.moment1[param] + (1 - self.beta1) * step
            self.moment2[param] = self.beta2 * self.moment2[param] + (1 - self.beta2) * step ** 2
            # do bias corrections
            m_unit = self.moment1[param] / (1 - self.beta1 ** iter)
            v_unit = self.moment2[param] / (1 - self.beta2 ** iter)
            # calculate the bias-corrected step size
            step = m_unit / ((v_unit + 1e-8) ** 0.5)
        # update the learnable parameter according to the given optimizer
        self.vals_for_learnable_params[param] += self.learning_rate * step
```

Figure 14: First modification to `MyComputationalGraphPrimer.backprop_and_update_params_multi_neuron_model`

the same as the one-neuron case. For the biases, the following code snippet was implemented in Figure 15.

```
# calculate g_{t+1} in update rules given in hw3
step = sum(pred_err_backproped_at_layers[back_layer_index]) * sum(deriv_sigmoid_avg) / len(deriv_sigmoid_avg)
# choose the update rule according to the optimizer name
if self.optimizer == "sgd+":
    # calculate the new step size (v_{t+1}) by momentum weighted summation
    prev_bias = self.bias_m[back_layer_index - 1]  # v_t
    self.bias_m[back_layer_index - 1] = self.momentum * prev_bias + step
    step = self.bias_m[back_layer_index - 1]  # v_{t+1}
elif self.optimizer == "adam":
    # calculate and save moments m_{t+1} and v_{t+1} for biases, respectively (eq. 3 in hw)
    self.bias_m1[back_layer_index - 1] = self.beta1 * self.bias_m1[back_layer_index - 1] + (1 - self.beta1) * step
    self.bias_m2[back_layer_index - 1] = self.beta2 * self.bias_m2[back_layer_index - 1] + (1 - self.beta2) * step ** 2
    # do bias corrections
    m_unit = self.bias_m1[back_layer_index - 1] / (1 - self.beta1 ** iter)
    v_unit = self.bias_m2[back_layer_index - 1] / (1 - self.beta2 ** iter)
    # calculate the bias corrected step size
    step = m_unit / ((v_unit + 1e-8) ** 0.5)
# do update according to the given optimizer and resulted step size
self.bias[back_layer_index - 1] += self.learning_rate * step
##########################################################################################################
```

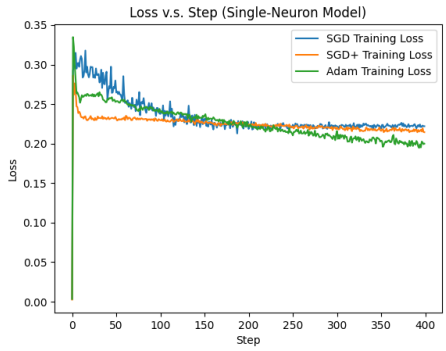
Figure 15: Second modification to `MyComputationalGraphPrimer.backprop_and_update_params_multi_neuron_model`

This follows exactly the same update rules as in the single neuron case, which are (1) (2) and (3). Parameter namings are the same so I do not explain them again but different from previous implementations, `self.bias_m`, `self.bias_m1`, and `self.bias_m2` are in the form of a list. They keep the moments for the biases at each layer. Lastly, if the optimizer is neither SGD+ nor Adam, it is accepted as SGD by default and classic vanilla SGD is performed.

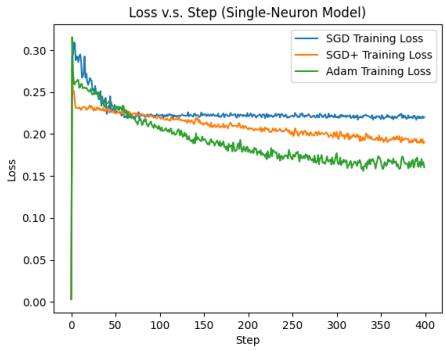
3.6 Results

I run 2 experiments with different learning rates for one-neuron and multi-neuron classifiers. For single-neuron case, expressions, output variables, dataset size, number of iterations, and batch size are chosen as $xw = ab * xa + bc * xb + cd * xc + ac * xd$, xw, 5000, 40000, and 8, respectively. For SGD+, the momentum term is set to 0.9 and for Adam, β_1 and β_2 are set to 0.9 and 0.99, respectively. In the two experiments of the single-neuron case, the value of the learning rate was changed to observe its effect on learning. In the first experiment, it is set to 10^{-3} and in the second, it is set to 3×10^{-3}.

Results in Figure 16 demonstrate the convergence of the different step-optimizers. Owing to its momentum mechanism, SGD+ is the first to decay but it stucks at the local minimum whereas



(a) Training loss of one-neuron classifiers with learning rate = 10^{-3}



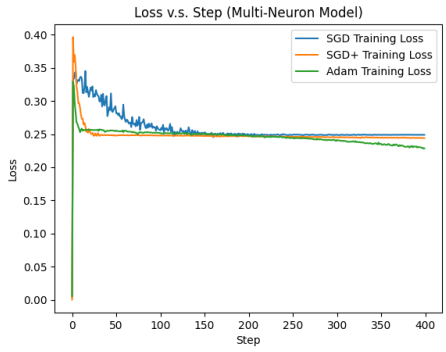
(b) Training loss of one-neuron classifiers with learning rate = 3×10^{-3}

Figure 16: Results of one-neuron classifiers for different learning rates

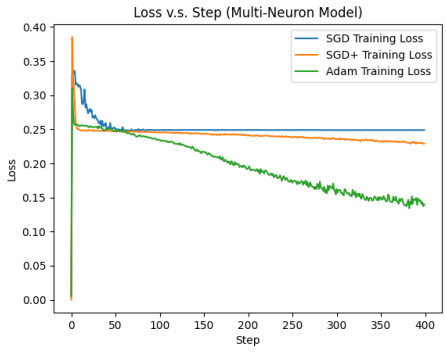
Adam also decays fast but thanks to its two-moment terms, it decays more slowly (controlled) than SGD+ and this prevents sticking at the local minimum. And since SGD is doing basic stochastic updates without paying attention its previous iterates, it stuck at the worst local minima and obtained the highest loss, that is, the worst performance. For the multi-neuron classifiers, exact same parameters in the single-neuron case are used again with the following exceptions. The expression or architecture of the neural network is

$$\begin{aligned} xw &= ap * xp + aq * xq + ar * xr + as * xs \\ xz &= bp * xp + bq * xq + br * xr + bs * xs \\ xo &= cp * xw + cq * xz \end{aligned} \tag{5}$$

where xo is the output variable. Thus, the number of layers is 3.



(a) Training loss of multi-neuron classifiers with learning rate = 10^{-3}



(b) Training loss of multi-neuron classifiers with learning rate = 3×10^{-3}

Figure 17: Results of multi-neuron classifiers for different learning rates

As you can see in the above results, the order of the losses for different optimizers did not change as in the one-neuron model, however, this does not mean that it will be always the case. Two experiments were done with different learning rate parameters: 10^{-3} and 3×10^{-3}. After increasing the learning rate, the performance gap (loss difference) between the models increased. Especially, Adam optimizer performed significantly better than SGD+ and SGD. Furthermore, in Figure 17a, it

seems like all the step-optimizers are stuck at similar local minima with small differences. However, after increasing the learning rate Adam seems to be freed from that local minima and achieves much better loss value, which may increase the performance of the model tremendously. Due to their momentum mechanisms, Adam and SGD+ performed faster decay than SGD. Lastly, using the same learning rate, 3×10^{-3}, the multi-neuron classifier performed better than the single-neuron classifier with the Adam optimizer. I think adding more layers to the network increases the generalization and learning capability of the network and therefore, decreases the training loss and hopefully will decrease the test loss as well. I believe it does not make sense to discuss the experiment functions that I wrote and showed in Figure 4 because it is very similar to the example cases in the Example directory of `ComputationalGraphPrimer` and in the main code I only called those two functions twice with some print statements. You can check my source code for more detail.

4 Lessons Learned

In this homework, I develop a greater appreciation for the step size optimization logic that is widely used in deep neural network training. I learned to implement SGD with momentum and Adam optimizers for single-neuron and multi-neuron cases. Also, I learned how to modify a superclass by inheriting it and creating another subclass, which performs the intended action. Moreover, I observed how the performance of the neural networks changed with respect to the batch size, learning rate and optimizer. I learned how to implement the entire training, which includes training loop, forward propagation, and backpropagation with updates, of single-neuron and multi-neuron architectures.

```
from ComputationalGraphPrimer import *
import random
import numpy as np
import operator
import matplotlib.pyplot as plt


class MyComputationalGraphPrimer(ComputationalGraphPrimer):

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

    # I implemented this function (MBS)
    def __init__(self, optimizer='sgd', momentum=None, beta1=None, beta2=None, *args, **kwargs):
        super(MyComputationalGraphPrimer, self).__init__(*args, **kwargs)
        # set the parameters that are entered
        if optimizer:
            self.optimizer = optimizer
        if momentum:
            self.momentum = momentum
        if beta1:
            self.beta1 = beta1
        if beta2:
            self.beta2 = beta2
        # set necessary parameters depending on the optimizer
        if self.optimizer == "sgd+":
            # Keeps the updates for momentum SGD
            self.step_sizes = None
            self.bias_m = None
        elif self.optimizer == "adam":
            # bias corrected moments
            self.moment1 = None # corresponds to m
            self.moment2 = None # corresponds to v
            self.bias_m1 = None
            self.bias_m2 = None


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


    # training loop for one neuron classifier
    def run_training_loop_one_neuron_model(self, training_data):
        """
        DISCLAIMER: I copied this function from Prof. Kak's ComputationalGraphPrimer source code.
        I marked the adjustments I made by adding MBS (Mehmet Berk Sahin) at the end of the comments.
        And I pointed the modifications out by showing them in a box with '#' symbol.

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

        self.bias = random.uniform(0, 1)   ## Adding the bias improves class discrimination.

        ###################################################################
        # initialize the previous moments or step sizes (MBS)
        if self.optimizer == "sgd+":
            self.step_sizes = {param : 0 for param in self.learnable_params}
            self.bias_m = 0
        elif self.optimizer == "adam":
            self.moment1 = {param : 0 for param in self.learnable_params}
            self.moment2 = {param: 0 for param in self.learnable_params}
            self.bias_m1 = self.bias_m2 = 0
        ###################################################################

        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 ecah 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_literations = 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_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))  ## Display average loss
            avg_loss_over_literations = 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)))
        ##################################################################
        # runs the backpropagation algorithm (I added iteration number as a parameter, MBS)
        self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg,
                                                         deriv_sigmoid_avg, iter=i+1)
        ##################################################################

    ##################################################################
    # I commented these plots to display them later (MBS)
    #plt.figure()
    #plt.plot(loss_running_record)
    #plt.show()
    return loss_running_record # returns the loss history (MBS)
    ##################################################################

# backpropagation and update of one neuron classifier (I added iter as a parameter, MBS)
def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid, iter):
    input_vars = self.independent_vars
    vals_for_input_vars_dict = dict(zip(input_vars, vals_for_input_vars))
    vals_for_learnable_params = self.vals_for_learnable_params
```

```
##########################################################################################################
        # All of this box is written by Mehmet Berk Sahin (update rule is taken from Prof. Kak's source code)
        # I did not change the variable names for easy comparison
        # do the step updates according to the optimizer
        if self.optimizer == "sgd+":
            # update loop for SGD+ (with momentum)
            for i, param in enumerate(self.vals_for_learnable_params):
                # calculate g_{t+1} parameter in the update rule eq. 2
                step = y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
                # do momentum weighted summation with the previous updates
                step += self.momentum * self.step_sizes[param]
                # save the latest step size (v_{t+1} parameter)
                self.step_sizes[param] = step
                # update the learnable parameter
                vals_for_learnable_params[param] += self.learning_rate * step
            # update the bias using the SGD+ optimizer
            self.bias_m = self.momentum * self.bias_m + y_error * deriv_sigmoid
            self.bias += self.learning_rate * self.bias_m
        elif self.optimizer == "adam":
            # update loop for Adam
            for i, param in enumerate(self.vals_for_learnable_params):
                # calculate g_{t+1} in the update rule eq. 3
                step = y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
                # do moment updates m_{t+1} and v_{t+1}, respectively
                self.moment1[param] = self.beta1 * self.moment1[param] + (1 - self.beta1) * step
                self.moment2[param] = self.beta2 * self.moment2[param] + (1 - self.beta2) * step ** 2
                # do bias correction for moments eq. 38 in Prof. Kak's Autograd slide
                m_unit = self.moment1[param] / (1 - self.beta1 ** iter)
                v_unit = self.moment2[param] / (1 - self.beta2 ** iter)
                # update the learnable parameters with bias-corrected moments
                self.vals_for_learnable_params[param] += self.learning_rate * m_unit / ((v_unit + 1e-8) ** 0.5)
            # calculate g_{t+1} in the update rule eq. 3 for bias term
            step = y_error * deriv_sigmoid
            # moment updates and bias corrections for the bias term
            self.bias_m1 = self.beta1 * self.bias_m1 + (1 - self.beta1) * step
            self.bias_m2 = self.beta2 * self.bias_m2 + (1 - self.beta2) * step ** 2
            m_unit = self.bias_m1 / (1 - self.beta1 ** iter)
            v_unit = self.bias_m2 / (1 - self.beta2 ** iter)
            # update the bias learnable parameter
            self.bias += self.learning_rate * m_unit / ((v_unit + 1e-8) ** 0.5)
        else:
            # Do normal SGD (This part is same as super class' SGD implementation)
            for i, param in enumerate(self.vals_for_learnable_params):
                step = self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid
                self.vals_for_learnable_params[param] += step
            self.bias += self.learning_rate * y_error * deriv_sigmoid

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

    # training loop for multi neuron classifier
    def run_training_loop_multi_neuron_model(self, training_data):
        """
        DISCLAIMER: I copied this function from Prof. Kak's ComputationalGraphPrimer source code.
        I marked the adjustments I made by adding MBS (Mehmet Berk Sahin) at the end of the comments.
        And I pointed the modifications out by showing them in a box with '#' symbol.
        """

        ##########################################################################
        # Initializes the necessary parameters for each learnable parameter (MBS)
        if self.optimizer == "sgd+":
            self.step_sizes = {param: 0 for param in self.learnable_params}
            self.bias_m = [0 for _ in range(self.num_layers - 1)]
        elif self.optimizer == "adam":
            self.moment1 = {param: 0 for param in self.learnable_params}
            self.moment2 = {param: 0 for param in self.learnable_params}
            self.bias_m1 = [0 for _ in range(self.num_layers - 1)]
            self.bias_m2 = [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 ecah sample
            self.class_1_samples = [(item, 1) for item in
                                    self.training_data[1]]  ## Associate label 1 with ecah 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_literations = 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_literations += loss_avg  ## Add to Average loss over iterations
        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))  ## Display avg loss
            avg_loss_over_literations = 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))

##########################################################################################################
        # runs backpropagation algorithm for multi neuron case (I added iter parameter, MBS)
        self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels, iter=i+1)
```

```
##########################################################################################################
        ##########################################################################
        # I commented these to diplay them later (MBS)
        #plt.figure()
        #plt.plot(loss_running_record)
        #plt.show()
        return loss_running_record # return the loss (MBS)
        ##########################################################################

    # backpropagation and update of multi neuron classifier
    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels, iter):
        """
        DISCLAIMER: I copied this function from Prof. Kak's ComputationalGraphPrimer source code.
        I marked the adjustments I made by adding MBS (Mehmet Berk Sahin) at the end of the comments.
        And I pointed the modifications out by showing them in a box with '#' symbol.

        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 patial derivatives stored away
                ##  during forward propagation of data. The theory underlying these calculations is presented
                ##  in Slides 68 through 71.

##########################################################################################################
                # I modified this section of the code (MBS)
                for i, param in enumerate(layer_params):
```

```
                    # calculate g_{t+1} in the update rule eq. 2
                    step = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j] *
deriv_sigmoid_avg[j]
                    if self.optimizer == "sgd+":
                        # calculate the momentum weighted step size
                        self.step_sizes[param] = self.momentum * self.step_sizes[param] + step
                        # save the last step size, v_{t+1}
                        step = self.step_sizes[param]
                    elif self.optimizer == "adam":
                        # calculate and save moments m_{t+1} and v_{t+1}, respectively (eq. 3 in hw)
                        self.moment1[param] = self.beta1 * self.moment1[param] + (1 - self.beta1) * step
                        self.moment2[param] = self.beta2 * self.moment2[param] + (1 - self.beta2) * step ** 2
                        # do bias corrections
                        m_unit = self.moment1[param] / (1 - self.beta1 ** iter)
                        v_unit = self.moment2[param] / (1 - self.beta2 ** iter)
                        # calculate the bias-corrected step size
                        step = m_unit / ((v_unit + 1e-8) ** 0.5)
                    # update the learnable parameter according to the given optimizer
                    self.vals_for_learnable_params[param] += self.learning_rate * step
            # calculate g_{t+1} in update rules given in hw3
            step = sum(pred_err_backproped_at_layers[back_layer_index]) * sum(deriv_sigmoid_avg) /
len(deriv_sigmoid_avg)
            # choose the update rule according to the optimizer name
            if self.optimizer == "sgd+":
                # calculate the new step size (v_{t+1}) by momentum weighted summation
                prev_bias = self.bias_m[back_layer_index - 1] # v_t
                self.bias_m[back_layer_index - 1] = self.momentum * prev_bias + step
                step = self.bias_m[back_layer_index - 1] # v_{t+1}
            elif self.optimizer == "adam":
                # calculate and save moments m_{t+1} and v_{t+1} for biases, respectively (eq. 3 in hw)
                self.bias_m1[back_layer_index - 1] = self.beta1 * self.bias_m1[back_layer_index - 1] + (1 -
self.beta1) * step
                self.bias_m2[back_layer_index - 1] = self.beta2 * self.bias_m2[back_layer_index - 1] + (1 -
self.beta2) * step ** 2
                # do bias corrections
                m_unit = self.bias_m1[back_layer_index - 1] / (1 - self.beta1 ** iter)
                v_unit = self.bias_m2[back_layer_index - 1] / (1 - self.beta2 ** iter)
                # calculate the bias corrected step size
                step = m_unit / ((v_unit + 1e-8) ** 0.5)
            # do update according to the given optimizer and resulted step size
            self.bias[back_layer_index - 1] += self.learning_rate * step

##########################################################################################################
# compares SGD, SGD+ and Adam with one neuron classifier
def one_neuron_experiment(learning_rate=5e-3):
    # for reproducibility
    seed = 0
    random.seed(seed)
    np.random.seed(seed)
    # SGD
    sgd_model = MyComputationalGraphPrimer(
        one_neuron_model=True,
        expressions=['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
        output_vars=['xw'],
        dataset_size=5000,
        learning_rate=learning_rate,
        #               learning_rate = 5 * 1e-2,
        training_iterations=40000,
        batch_size=8,
        display_loss_how_often=100,
        debug=True,
        optimizer="sgd"
    )
    # SGD+
    sgdp_model = MyComputationalGraphPrimer(
        one_neuron_model=True,
        expressions=['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
        output_vars=['xw'],
        dataset_size=5000,
        learning_rate=learning_rate,
        #               learning_rate = 5 * 1e-2,
        training_iterations=40000,
        batch_size=8,
        display_loss_how_often=100,
        debug=True,
        optimizer="sgd+",
        momentum=0.9
    )
    # ADAM
    adam_model = MyComputationalGraphPrimer(
        one_neuron_model=True,
```

```
        expressions=['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
        output_vars=['xw'],
        dataset_size=5000,
        learning_rate=learning_rate,
        #               learning_rate = 5 * 1e-2,
        training_iterations=40000,
        batch_size=8,
        display_loss_how_often=100,
        debug=True,
        optimizer="adam",
        beta1=0.9,
        beta2=0.99
    )

    # set the parameters
    sgd_model.parse_expressions()
    sgdp_model.parse_expressions()
    adam_model.parse_expressions()
    # generate a shared training data
    training_data = sgd_model.gen_training_data()
    # start training of each model
    loss_sgd_cgp = sgd_model.run_training_loop_one_neuron_model(training_data)
    print("Training for SGD with single neuron is completed.")
    loss_sgdp_cgp = sgdp_model.run_training_loop_one_neuron_model(training_data)
    print("Training for SGD+ with single neuron is completed.")
    loss_adam_cgp = adam_model.run_training_loop_one_neuron_model(training_data)
    print("Training for ADAM with single neuron is completed.")
    # plot the training histories
    plt.figure()
    plt.plot(loss_sgd_cgp)
    plt.plot(loss_sgdp_cgp)
    plt.plot(loss_adam_cgp)
    plt.legend(["SGD Training Loss", "SGD+ Training Loss", "Adam Training Loss"])
    plt.title("Loss v.s. Step (Single-Neuron Model)")
    plt.xlabel("Step")
    plt.ylabel("Loss")
    plt.show()

# compares SGD, SGD+ and Adam with multi neuron classifier
def multi_neuron_experiment(learning_rate=5e-3):
    # for reproducibility
    seed = 0
    random.seed(seed)
    np.random.seed(seed)

    # SGD for multi-neuron classifier
    sgd_model = MyComputationalGraphPrimer(
                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 = learning_rate,
#                learning_rate = 5 * 1e-2,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    # SGD+ for multi-neuron classifier
    sgdp_model = MyComputationalGraphPrimer(
                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 = learning_rate,
#                learning_rate = 5 * 1e-2,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
                optimizer='sgd+',
                momentum=0.9
    )
    # ADAM for multi-neuron classifier
    adam_model = MyComputationalGraphPrimer(
```

```
                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 = learning_rate,
#                learning_rate = 5 * 1e-2,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
                optimizer='adam',
                beta1=0.9,
                beta2=0.99
    )

    # setup
    sgd_model.parse_multi_layer_expressions()
    sgdp_model.parse_multi_layer_expressions()
    adam_model.parse_multi_layer_expressions()
    # generate data
    training_data = sgd_model.gen_training_data()
    # start learning
    loss_sgd_cgp = sgd_model.run_training_loop_multi_neuron_model(training_data)
    print("Training for SGD with multi neuron is completed.")
    loss_sgdp_cgp = sgdp_model.run_training_loop_multi_neuron_model(training_data)
    print("Training for SGD+ with multi neuron is completed.")
    loss_adam_cgp = adam_model.run_training_loop_multi_neuron_model(training_data)
    print("Training for ADAM with multi neuron is completed.")
    # plot the training histories for multi neuron classifier
    plt.figure()
    plt.plot(loss_sgd_cgp)
    plt.plot(loss_sgdp_cgp)
    plt.plot(loss_adam_cgp)
    plt.legend(["SGD Training Loss", "SGD+ Training Loss", "Adam Training Loss"])
    plt.title("Loss v.s. Step (Multi-Neuron Model)")
    plt.xlabel("Step")
    plt.ylabel("Loss")
    plt.show()

# main code
if __name__ == "__main__":
    # for reproducibility
    seed = 0
    random.seed(seed)
    np.random.seed(seed)
    # experiments for single-neuron case
    print("#"*30)
    print(f"The experiment for single-neuron classifier with learning rate 1e-3 is being started...")
    one_neuron_experiment(learning_rate=1e-3)
    print("The experiment for single-neuron class with learning rate 1e-3 was ended.")
    print(f"The experiment for single-neuron classifier with learning rate 3e-3 is being started...")
    one_neuron_experiment(learning_rate=3e-3)
    print("The experiment for single-neuron class with learning rate 3e-3 was ended.")
    # experiments for multi-neuron case
    print("#" * 30)
    print("The experiment for multi-neuron classifier with learning rate 1e-3 is being started...")
    multi_neuron_experiment(learning_rate=1e-3)
    print("The experiment for multi-neuron class with learning rate 1e-3 was ended.")
    print("The experiment for multi-neuron classifier with learning rate 3e-3 is being started...")
    multi_neuron_experiment(learning_rate=3e-3)
    print("The experiment for multi-neuron class with learning rate 3e-3 was ended.")

    print("Homework 3 code come to an end. ")
```