HW 3: Gradient Optimization Methods

David E. Farache, Email ID: dfarache@purdue.edu

February 7, 2023

1 Introduction

The power of deep learning neural networks comes from its method to calculate and minimize its loss. Loss is typically calculated using a cost function including mean absolute error (MAE), mean standard error (MSE), and root means standard error (RMSE). It typically finds the minimum by a slope value that would lead to a lower value in a hyperplane. Gradient descent (GD), finds the minimum to enter into the lowest point of the hyper-lane that can not be envisioned as there is no ability to comprehend multi-dimensional landscape or it being to expensive to explore the entire area.

This assignment focuses on adding different optimization methods for an existing single-neuron and multi-neuron models. The code already had a method known as stochastic gradient descent (SGD) and we further add better supposedly better performing optimization methods of stochastic gradient descent plus moment (SGD+) and Adam.[2]

2 Background

As previously stated the code already had SGD within the existing code that uses the standard gradient descent method but adds a stochastic jump. This is to combat the over-shooting nature of standard GD. The over-shooting occurs due to the GD relying on the gradient which becomes larger as one nears the minimum, causing an over-estimation and arriving at the other end of the drop instead of the bottom. SGD can be explained mathematically by the equation below:

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

Where p_t is the learn-able parameter at index t, lr is the learning rate which can also by represented α, and g_t, the gradient at index t. Although SGD resolves the issue of gradient values adjustment being too large, the stochastic jumps can lead position away from the minimum. The random jumps are performed per piece of data, meaning it could move in the opposing direction if the dice roll is not in one's favor. This is adjusted by a method called SDG+ which introduces 'momentum' to reduce the effect of incorrect stochastic jumping. In this case the momentum takes into account the proper direction given by the favorable direction and reduces the effect of in corrected exploration. Momentum (v_t) can be described by equation 2, where μ is the momentum scalar:

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

The SDG+ equation is the same format as SGD but swapping the gradient term for the momentum:

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

The final optimization method was Adam which combines the advantages of Adaptive Gradients, and Root Mean Square Propagation optimizers. Adam differs from SGD and SGD+, in that it has an evolving learning rate based on the first moment (m_t) and the second moment (v_t) which comes from teh Adaptive Gradient optimizer.[1] It does so by calculating an exponential moving average of the gradient and using parameters β_1 and β_2 to control the decay rate, inspired by the Root Mean Square Propagation optimizer, as can be seen in the equations below:[3]

$$m_{t+1} = \beta_1 * m_t + (1 - \beta_1) * g_{t+1} \tag{4}$$

$$v_{t+1} = \beta_2 * v_t + (1 - \beta_2) * (g_{t+1})^2 \tag{5}$$

Given this, Adam does initialize its first and second momentum to zero which biases the value toward zero iteration. To resolve this, was add bias-corrected values for momentum using the equations below:[3]

$$\hat{m} = \frac{m_t}{1 - \beta_1} \tag{6}$$

$$\hat{v} = \frac{v_t}{1 - \beta_2} \tag{7}$$

The following position is then calculated using the equation below with an added parameter of ϵ to assure that fraction can not go to infinity as the initial momentum term is zero.

$$p_{t+1} = p_t - lr * \frac{\hat{m}}{\sqrt{\hat{v}_{t+1} + \epsilon}} \tag{8}$$

3 Methodology

For this assignment the ComputationalGraphPrimer-1.1.2 library was utilized, with the addition of an input called optimizer which enables the user to select the optimizer. The default value is to 'SGD' but swapping entails feeding in 'Adam' or 'SGD+'.

```
if 'optimizer' in kwargs                    :   optimizer = kwargs.pop('
    optimizer') #Added
if optimizer:
            self.optimizer = optimizer
        else:
            self.optimizer = 'SGD'
```

Listing 1: *Added Section to Library*

A new class was created ComputationGraphPrimerUpdate() that inherits the properties of the ComputationalGraphPrimer() class. The run_training_loop_one_neuron_model, forward_prop_one_neuro and backprop_and_update_params_one_neuron_model were copied from the library with an SGD_plus and Adam method added. These methods were copedi from the backprop_and_update_params_on method and changed to function as the optimizer they were named after. The following code was added to run_training_loop_one_neuron_model for selection of the optimizer.

```
 if self.optimizer == 'SGD':
self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg,
    deriv_sigmoid_avg)     ## BACKPROP loss
if self.optimizer == 'SGD+':
self.SGD_plus(y_error_avg, data_tuple_avg, deriv_sigmoid_avg)     ## BACKPROP
    loss
if self.optimizer == 'Adam':
```

```
self.Adam(y_error_avg, data_tuple_avg, deriv_sigmoid_avg, i + 1)     ## BACKPROP
    loss
```

Listing 2: *Selection Code for Optimization Function One Neuron*

Again for the multi-neuron file the ComputationGraphPrimerUpdate() inherits the properties of the ComputationalGraphPrimer() class. The run_training_loop_multi_neuron_model, forward_prop_multi_neuron_model, and backprop_and_update_params_multi_neuron_model were copied from the library with an SGD_plus and Adam method added. These methods were copedi from the backprop_and_update_params_multi_neuron_model method and changed to function as the optimizer they were named after. The following code was added to run_training_loop_multi_neuro
for selection of the optimizer.

```
if self.optimizer == 'SGD':
    self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels
    )     ## BACKPROP loss
if self.optimizer == 'SGD+':
    self.SGD_plus(y_error_avg, class_labels)     ## BACKPROP loss
if self.optimizer == 'Adam':
    self.Adam(y_error_avg, class_labels, i+1)     ## BACKPROP loss
```

Listing 3: *Selection Code for Optimization Function Multi Neuron*

It should be noted that for all optimization methods introduced, the bias is update separately from position but within the same epoch.

4 Task 1: One Neuron

4.1 SGD+

This part is the global variables added to the run_training_loop_one_neuron_model function where mu = 0.7.

```
# Add for SG Optimization
self.mu = mu
self.step = 0
self.bias_change = 0
```

Listing 4: *Initial variables for SGD+*

The SGD+ code follows the logic seen in equation 2 and 3 where step is equal v_t and learn-able parameter is p_t. Bias is the updated by a variable called bias change.

```
def SGD_plus(self, y_error, vals_for_input_vars, deriv_sigmoid):
    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
        self.step = (self.mu * self.step) + (self.learning_rate * y_error *
   vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid)
            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += self.step

    self.bias_change = (self.mu * self.bias_change) + (self.learning_rate *
   y_error * deriv_sigmoid)   ## Update the bias
```

```
    self.bias += self.bias_change
```

Listing 5: *SGD+ code for one neuron*

4.2 Adam

This part is the global variables added to the run_training_loop_one_neuron_model function where beta1 = 0.9, beta2 = 0.99, and epsilon = 1e-6.

```python
# Added for Atom Optimization
self.beta1 = beta1
self.beta2 = beta2
self.epsilon = epsilon
self.m_t = 0
self.v_t = 0
self.m_td =  0
self.v_td = 0
```

Listing 6: *Initial variables for Adam*

For Adam, we follow the same logic as equations 4-8. With m_td and m_t being the fist momentum term and v_td and v_t being the second momentum term. In this case g_t or the gradient is equivalent to self (learning_rate * y_error vals_for_input_vars_dict[input_vars [i]] * deriv_sigmoid). The calculation for the following point is done within the for-loop shown while the update for bias is done outside.

```python
def Adam(self, y_error, vals_for_input_vars, deriv_sigmoid, iter):
    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):
        # Code refrence from: https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b217f1cc

            ## Calculate momentum gradient
            self.m_td =  (self.beta1 * self.m_td) + (1 - self.beta1) * (self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid)
            self.v_td =  (self.beta2 * self.v_td) + (1 - self.beta2) * (self.learning_rate * y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid)**2

            mk_hat = self.m_td / (1 - self.beta1 ** iter)
            vk_hat = self.v_td / (1 - self.beta2 ** iter)

            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += mk_hat / np.sqrt(vk_hat + self.epsilon)

    ## Calculate momentum 1 and 2
    self.m_t = (self.beta1 * self.m_t) + (1 - self.beta1) * (self.learning_rate * y_error * deriv_sigmoid)
    self.v_t = (self.beta2 * self.v_t) + (1 - self.beta2) * (self.learning_rate * y_error * deriv_sigmoid) ** 2

    ## Get values to update bias
    m_hat = self.m_t / (1 - self.beta1 ** iter)
    v_hat = self.v_t / (1 - self.beta2 ** iter)

    self.bias += m_hat / np.sqrt(v_hat + self.epsilon) ## Update the bias
```

Listing 7: *Adam optimization calcuation for one neuron*

5 Task 2: Multi-Neuron

5.1 SGD+

This part is the global variables added to the run_training_loop_one_neuron_model function where mu = 0.6.

```
# Add for SG Optimization
self.mu = mu
self.step = 0
self.bias_change = 0
```

Listing 8: *Initial variables for SGD+*

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

self.bias_change = (self.mu * self.bias_change) + (self.learning_rate * sum(pred_err_backproped_at_layers[back_layer_index]) \ * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))
self.bias[back_layer_index-1] += self.bias_change
```

Listing 9: *Code for SGD+*

5.2 Adam

This part is the global variables added to the run_training_loop_one_neuron_model function where beta1 = 0.9, beta2 = 0.99, and epsilon = 1e-6.

```
# Added for Atom Optimization
 self.beta1 = beta1
self.beta2 = beta2
self.epsilon = epsilon
self.m_t = 0
self.v_t = 0
self.m_td =  0
self.v_td = 0
```

Listing 10: *Initial variables for Adam*

For Adam, we follow the same logic as equations 4-8. With m_td adn m_t being the fist momentum term and v_td and v_t being the second momentum term for finding the following point. In this case g_t is equal to input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j] * deriv_sigmoid_avg[j]. The calculation for the following point is done within the for-loop shown while the update for bias is done outside.

```
for j,var in enumerate(vars_in_layer):
    layer_params = self.layer_params[back_layer_index][j]
    ##  Regarding the parameter update loop that follows, see the Slides 74 through 77 of my Week 3
    ##  lecture slides for how the parameters are updated using the partial derivatives stored away
    ##  during forward propagation of data. The theory underlying these calculations is presented
    ##  in Slides 68 through 71.
    for i,param in enumerate(layer_params):
        gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]
    # Code refrence from: https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b217f1cc

        ## Calculate momentum gradient
        self.m_td = (self.beta1 * self.m_td) + (1 - self.beta1) * (gradient_of_loss_for_param * deriv_sigmoid_avg[j])
        self.v_td = (self.beta2 * self.v_td) + (1 - self.beta2) * (gradient_of_loss_for_param * deriv_sigmoid_avg[j]) ** 2

        mk_hat = self.m_td / (1 - self.beta1 ** iter)
        vk_hat = self.v_td / (1 - self.beta2 ** iter)

        ## Update the learnable parameters
        self.vals_for_learnable_params[param] += self.learning_rate * mk_hat / np.sqrt(vk_hat + self.epsilon)

    self.m_t = (self.beta1 * self.m_t) + (1 - self.beta1) * (sum(pred_err_backproped_at_layers[back_layer_index]) \
    * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))
    self.v_t = (self.beta2 * self.v_t) + (1 - self.beta2) * (sum(pred_err_backproped_at_layers[back_layer_index]) \
    * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)) ** 2

    m_hat = self.m_t / (1 - self.beta1 ** iter)
    v_hat = self.v_t / (1 - self.beta2 ** iter)

    self.bias[back_layer_index-1] += self.learning_rate * m_hat / np.sqrt(v_hat + self.epsilon) ## Update the bias
```

Listing 11: *Code for Adam*

6 Task 3: Final Results

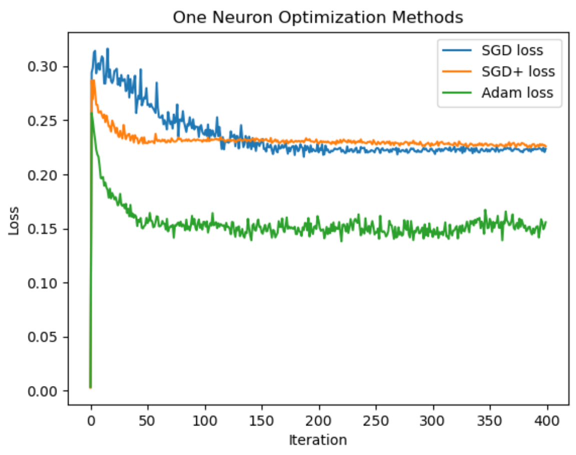
The results for all optimization models are displayed below along with the code utilized to develop the plots. There two different codes for one and multi neuron but the changes just calling the function for set model.

```python
def runningOpt(optName):
    loss_iteration_list = []
    for opt in optName:
        cgp = ComputationGraphPrimerUpdate(
                    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,
                    optimizer = opt
        )

        cgp.parse_expressions()
        # cgp.display_one_neuron_network()

        training_data = cgp.gen_training_data()
        loss_per_iteration = cgp.run_training_loop_one_neuron_model(
    training_data )
        #print(loss_per_iteration)
        loss_iteration_list.append(loss_per_iteration)
     return loss_iteration_list

optLoss = runningOpt(['SGD', 'SGD+', 'Adam'])

plt.plot(range(len(optLoss[0])), optLoss[0], label="SGD loss")
plt.plot(range(len(optLoss[1])), optLoss[1], label="SGD+ loss")
plt.plot(range(len(optLoss[2])), optLoss[2], label="Adam loss")

plt.title("One Neuron Optimization Methods")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend(loc="upper right")

plt.show()
```

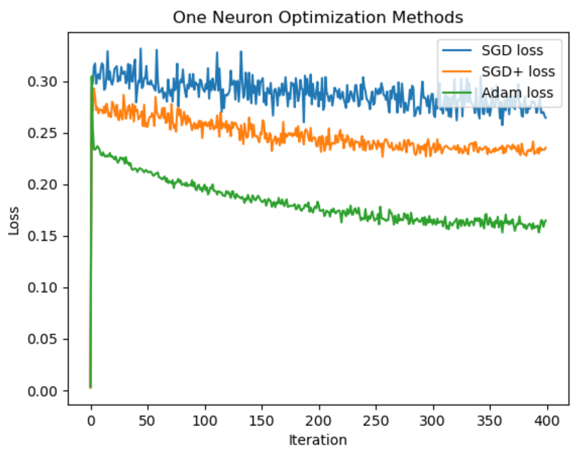
Listing 12: *Plotting code for one neuron*

As can be seen in Figure 1 a and b Adam optimizer outperforms both the SGD and the SGD+ optimizer which is expected. What is interesting is that SGD+ out performs SGD expected for later on for the learning rate of 1e-3. It does appear that a lower learning rate leads to the same results but in a delayed time line as Figure 1 b seems to be their earlier section of Figure 1 a.

```python
def runningOpt(optName):
    loss_iteration_list = []
    for opt in optName:
        cgp = ComputationGraphPrimerUpdate(
```



(a) *Learning Rate: 1e-3*

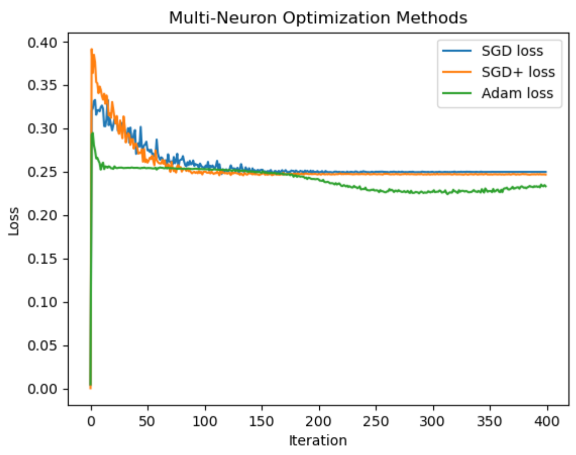


(b) *Learning Rate: 1e-4*

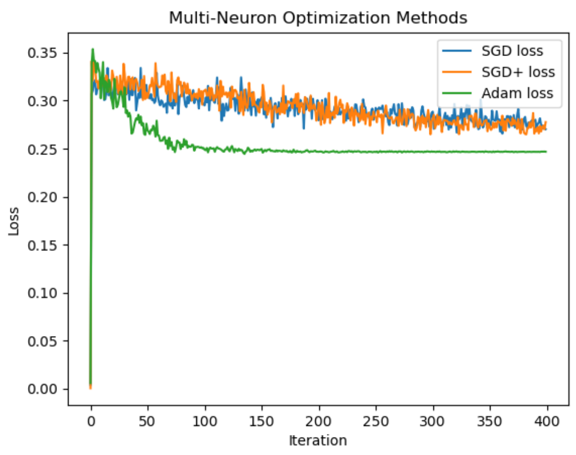
Figure 1: *One neuron loss over iteration comparison between SGD, SGD+, and Adam*

```python
                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,
                optimizer = opt,
        )

        cgp.parse_multi_layer_expressions()
        # cgp.display_one_neuron_network()

        training_data = cgp.gen_training_data()
        loss_per_iteration = cgp.run_training_loop_multi_neuron_model( training_data )
        #print(loss_per_iteration)
        loss_iteration_list.append(loss_per_iteration)
    return loss_iteration_list

optLoss = runningOpt(['SGD', 'SGD+', 'Adam'])

plt.plot(range(len(optLoss[0])), optLoss[0], label="SGD loss")
plt.plot(range(len(optLoss[1])), optLoss[1], label="SGD+ loss")
plt.plot(range(len(optLoss[2])), optLoss[2], label="Adam loss")

plt.title("Multi-Neuron Optimization Methods")
plt.xlabel("Iteration")
plt.ylabel("Loss")
plt.legend(loc="upper right")

plt.show()
```

Listing 13: *Plotting code for multi-neuron*



(a) *Learning Rate: 1e-3*



(b) *Learning Rate: 1e-4*

Figure 2: *Multi neuron loss over iteration comparison between SGD, SGD+, and Adam*

As can be seen in Figure 2, Adam optimizer outperforms both the SGD and the SGD+ optimizer except for the range between 100 and 200 iterations for learning rate 1e-3. For learning rate 1e-3, again SGD+ performs better than SGD early on but as iterations increase the trend is the same, while for learnign rate 1e-4, the results are near indentical which is suprising. The expected result was for SGD+ to provide better results than SGD. The smaller learning rate does improve the initial loss for SGD+ which could be do to more time given to learn.

Between Figures 1 and 2, it can be seen that the loss is lower for the one-neuron case than the multi-neuron which was the opposite to the expected result. This could be due to the one neuron case overfitting which is more prone to as it is a single neuron, meaning that although the loss is lower, the model is worse. For both cases, it does appear that a smaller learning rate leads to the same results as a larger on at its initial section, this makes sense but is probably only true for this set of data and results.

7 Lessons learned

This assignment was an introduction to gradient descent and optimization methods for neural networks. I have used neural networks with the understanding of general theory and knowing SGD but I have now been able to go deeper with SGD+ and understand what Adam actually does.

References

[1] Jason Brownlee. Code adam optimization algorithm from scratch. `https://machinelearningmastery.com/adam-optimization-from-scratch/`, 2021.

[2] Vitaly Bushaev. Stochastic gradient descent with momentum. https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d, 2017.

[3] Enoch Kan. How to implement an adam optimizer from scratch. https://towardsdatascience.com/how-to-implement-an-adam-optimizer-from-scratch-76e7b217f1cc, 2020.