Chapter 05| Stochastic Gradient Descent & Learning Rate

~~~~~~

Gradient Descent **Without** Learning Rate

Using the partial derivatives, we can determine which direction to update. A **negative (-)** partial derivative of the cost function w.r.t. some variable **x** means that we need to **increase (+) x**. A **positive (+)** partial derivative of the cost function w.r.t. some variable **x** means that we need to **decrease (-) x**. As a result, the **update direction** will be the **negative** of the partial derivative.

Now the question is how much do we increase/decrease **x**? Well if the **update direction** is determined by only **part** of the partial derivative (the sign, positive/negative), then maybe the **update amount** is determined by the **other part** of the partial derivative (the magnitude/value). Following this train of thought, let's create a function that updates the parameters of the model based on the partial derivatives of the cost function w.r.t. the parameters. Here is the code:

```
def update_params(layer):    # dense layer
    layer.weights += -layer.dweights
    layer.biases += -layer.dbiases
```

The argument *layer* requires a dense layer object. To check if the cost goes down:

```
# hours studied
X = np.array([[0], [1], [2], [3], [4], [5], [6], [7], [8]])    # one input feature for each example
# percentage
y = np.array([[20], [30], [40], [50], [60], [70], [80], [90], [100]])    # one output feature for each example

mse = MSE_Cost()    # define cost function
model = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
model.forward(X)
print(model.weights, model.biases)
print("Original Cost:", mse.forward(model.outputs, y))

mse.backward(model.outputs, y)
model.backward(mse.dinputs)

# Check New Cost
update_params(model)
model.forward(X)
print(model.weights, model.biases)
print("New Cost:", mse.forward(model.outputs, y))
```

[[0.01764052]] [[0.]]
Original Cost: 4255.854199207604
[[612.55127013]] [[119.85887581]]
~~~~~~

New Cost: 8720874.619018847

In unit 4, we established that the optimum weight value is 10 and the optimum bias value is 20. This means our rule for **update direction** is correct where we increase the weight and bias, but the **update amount** is wrong. Increasing the weight value from ~0.02 all the way to ~610 is **too much increase** and increasing the bias value from 0 all the way to ~120 is **too much increase**. We will fix this issue in the next section.

Gradient Descent **With** Learning Rate

Instead of using the **whole partial derivative** as the **update amount** we must use a **fraction of the partial derivative** to determine the **update amount**. The **fraction of the partial derivative** is called a **gradient** which we use to **descend** the cost function.

$$G_w = \alpha *- \frac{\partial C}{\partial w},\ G_b = \alpha *- \frac{\partial C}{\partial b}$$

To get the gradient, we multiply the negative of the partial derivative by a scalar (greek letter alpha). Notice that the value of alpha is universal (the same) for the gradient of the weights and the gradient of the biases. This value of alpha is called the **learning rate** and it is up to us to determine the best value of alpha to train with. Knowing that, it may seem trivial to determine the **best value of alpha** since don't we want a smart model fast? Meaning, don't we want the model to **learn fast**? So just set the **learning rate** to something like 999,999,999 and we are done...

In reality, if you analyzed the gradient equations, you may notice that in the last section, our alpha (learning rate) was equal to 1 since multiplying the partial derivative by 1 results in the gradient equal to the exact same value as the partial derivative. As we saw, the result of the learning rate **only** equal 1 is the gradient being too **high**, since we increased the weight and bias by **too much**. Thus, we need to use a learning rate **less than** 1. Here is the code:

```
class SGD_Optimizer:
    def __init__(self, learning_rate):
        self.lr = learning_rate

    def update_params(self, layer):    # dense layer
        layer.weights += self.lr * -layer.dweights
        layer.biases += self.lr * -layer.dbiases
```

An **optimizer** is one that uses a **fraction** of the **partial derivative** to determine the **update amount** (gradient). The optimizer that we have defined here is **S**tochastic **G**radient **D**escent. Historically, SGD means that we **only** use

the partial derivative of the cost function on **one example** (out of all, in the dataset) to calculate the gradient that allows us to descend the cost function. However, now, terms have merged and the standard optimizer **for gradient descent** is called SGD regardless of the number of examples we use. When we initialize the optimizer, we need the learning rate which we abbreviate with lr. It should be easy to figure out what changed in the update_params method based on the gradient equations.

```
# hours studied
X = np.array([[0], [1], [2], [3], [4], [5], [6], [7], [8]])    # one input feature for each example
# percentage
y = np.array([[20], [30], [40], [50], [60], [70], [80], [90], [100]])    # one output feature for each example

mse = MSE_Cost()    # define cost function
model = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
optimizer = SGD_Optimizer(0.1)    # learning rate of 0.1

model.forward(X)
print(model.weights, model.biases)
print("Original Cost:", mse.forward(model.outputs, y))

mse.backward(model.outputs, y)
model.backward(mse.dinputs)

# Check New Cost
optimizer.update_params(model)
model.forward(X)
print(model.weights, model.biases)
print("New Cost:", mse.forward(model.outputs, y))
```

Here is the rest of the code, the first block is the same. In the second block, we define the optimizer with a learning rate of 0.1. The 3rd and 4th block of code is the same. In the last block, we use our optimizer's update_params **method** instead of the update_params function since we replaced the function update_params with a class.

```
[[0.01764052]] [[0.]]
Original Cost: 4255.854199207604
[[61.27100348]] [[11.98588758]]
New Cost: 56361.31807188052
```

As you can see, using a learning rate did help in making sure we do not increase the weights and biases **too much**. Despite the improvement the learning rate is still too high.

Typical values of learning rates are **less than** 0.1 (we used 0.1 and 1 to show the effects of a high learning rate). Here is the code with a learning rate of 0.01:

```
mse = MSE_Cost()    # define cost function
model = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
optimizer = SGD_Optimizer(0.01)    # learning rate of 0.01
```

Now when we define the optimizer, the learning rate is 0.01.

```
[[0.01764052]] [[0.]]
Original Cost: 4255.854199207604
[[6.14297682]] [[1.19858876]]
New Cost: 1270.8364603819844
```

The output clearly shows that this learning rate is small enough so that we do not increase the weight and bias **too much**. Now to continue making the model smarter, we need to **continuously** update the parameters by **continuously** calculating the **partial derivatives** of cost function w.r.t. the weight and bias because with each update the **partial derivatives** of cost function w.r.t. the weight and bias changes as the model ends up at different locations of the cost function. Here is the code:

```python
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(0)    # For repeatability
```

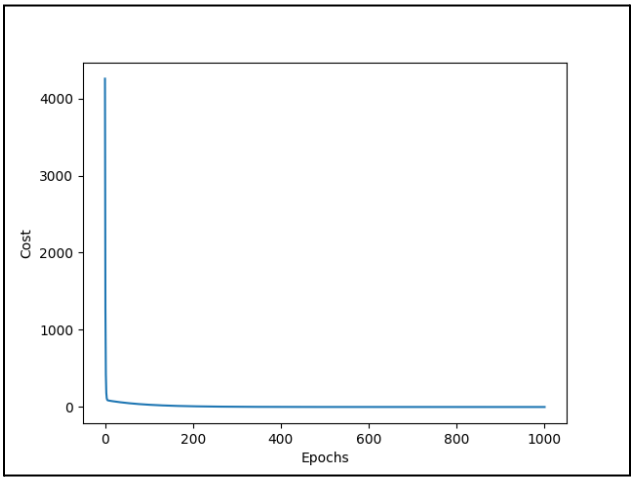
In the imports we need to also import matplotlib to graph the cost history.

```python
for epochs in range(1000):
    # forward pass
    model.forward(X)
    cost_history.append(mse.forward(model.outputs, y))

    # backward pass
    mse.backward(model.outputs, y)
    model.backward(mse.dinputs)

    optimizer.update_params(model)    # update

# Check New Cost
model.forward(X)
print(model.weights, model.biases)
cost_history.append(mse.forward(model.outputs, y))
print("New Cost:", cost_history[-1])

# Initialize Graph
fig = plt.figure()    # create a graphing space
plt.plot(cost_history)     # plot on graphing space

# Label Graph
plt.xlabel("Epochs")
plt.ylabel("Cost")

# Save and Close Graph
fig.savefig("history.png")
plt.close()
```

We create an empty list *cost_history* as a placeholder for a history of cost values to graph while we are training. Each time we do a forward pass and backward pass to update parameters based on **all the examples of the dataset** during the training loop, it is called an **epoch**. When we get the cost from each forward pass we append that to *cost_history*. At the end of training, we get the final cost value of the model. To graph, we create a graphing space (figure to graph on) with *plt.figure()*. Plot *cost_history* with *plt.plot()*, remember when no x-axis is given, matplotlib automatically fills in with 0 to the length of the inputted y-axis values.

Label the x-axis and y-axis with *plt.xlabel* (label the string "Epochs") and *plt.ylabel* (label the string "Cost") respectively, and then finish off by saving the graph to *history.png* using *fig.savefig()* and then close the graph (stop editing) with *plt.close()*.

[[0.01764052]] [[0.]]
Original Cost: 4255.854199207604
[[10.01035889]] [[19.94203801]]
New Cost: 0.0009885003413084448



Not much to see from the graph except that the cost went down **a lot**. The model is finally smart! The new cost is so low ~0.001 while the weight is ~10 with the bias being ~20, what we established as ideal values in unit 4. Congratulations on training (making your) first model smart and able to predict your test score based on how long you studied!

Okay, this example must have been too easy to go through and the model is pretty useless since there are many other factors that determine your test score and usually the data is not perfectly **linear**. However, we used a very simple example so that the process of learning gradient descent would not be too hard. Also, to keep things simple, we are still using this example in the next unit.

As you can see, going through the trouble of getting the analytical derivatives pay off because we do not waste time by updating parameters **randomly**. If you go back to unit 4, updating parameters randomly only got our cost down to ~50 when trying 1,000 times, however with SGD 1,000 epochs gets our cost all the way down to ~0.001!

Limitations of SGD

Given the fact that we had to update the model's parameters **1,000 times**, signifies that SGD does indeed make the model smart but takes a **long time** to do so. As a result, let's find why SGD is **so slow** at training models. To do this we cannot analyze the result, the cost graph we must go deeper by a level. We need to analyze **what created** the result (cost lower)? The answer is the parameter updates. Hence, we should create a graph of the parameter value during training as well. Here is the code:

```
model.forward(X)
print(model.weights, model.biases)
cost_history = []    # append to this list in the loop
weight_history, bias_history = [], []    # append in loop
print("Original Cost:", mse.forward(model.outputs, y))

for epochs in range(1000):
    # forward pass
    model.forward(X)
    weight_history.append(float(model.weights))
    bias_history.append(float(model.biases))
    cost_history.append(mse.forward(model.outputs, y))

    # backward pass
    mse.backward(model.outputs, y)
    model.backward(mse.dinputs)

    optimizer.update_params(model)    # update

# Check New Cost
model.forward(X)
weight_history.append(float(model.weights))
bias_history.append(float(model.biases))
print(model.weights, model.biases)
cost_history.append(mse.forward(model.outputs, y))
print("New Cost:", cost_history[-1])
```

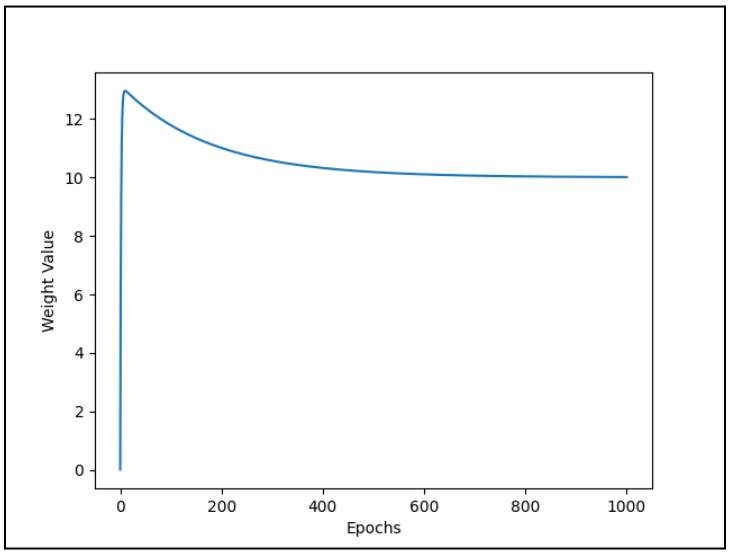
In addition to the empty *cost_history* list when we are getting the initial cost of the model, we have an empty *weight_history* and *bias_history* list. In the training loop, right after we get the model's predictions, we append the weight and bias of the model.

Since there is only one input feature and one neuron (output feature), *model.weights* is a [1, 1] matrix which can be converted to a float before appending to the weight history.

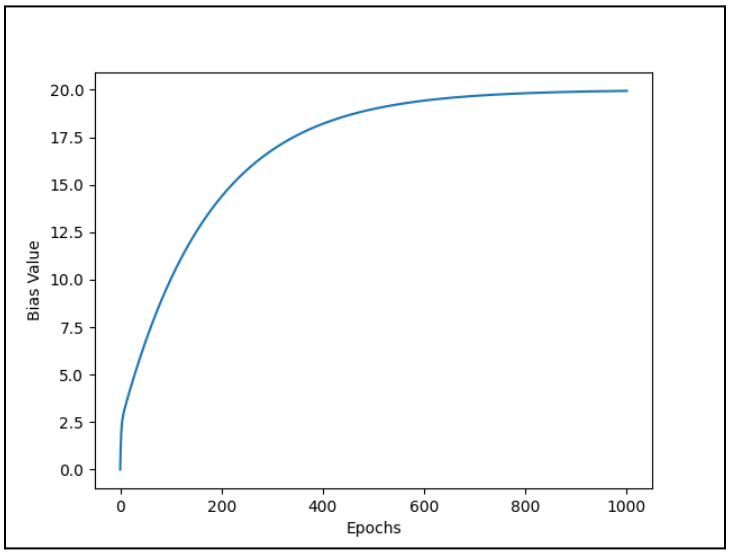
For the bias, since there is only one neuron (output feature) so *model.biases* is a [1, 1] matrix which can also be converted to a float before appending to the bias history. After both appends we can append the cost to the cost history. Lastly, after we get the new model's prediction (after training), we append the final weight and bias.

```python
def graph(data, xlabel, ylabel, fname):
    # Initialize Graph
    fig = plt.figure()    # create a graphing space
    plt.plot(data)     # plot on graphing space

    # Label Graph
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)

    # Save and Close Graph
    fig.savefig(fname)
    plt.close()

graph(weight_history, "Epochs", "Weight Value", "weight.png")
graph(bias_history, "Epochs", "Bias Value", "bias.png")
graph(cost_history, "Epochs", "Cost", "cost.png")
```

To save space, we create a graphing function. The *data* argument is the y-axis values (matplotlib fills the x-axis with integers from 0 to the number of elements in *data*), *xlabel* and *ylabel* should be strings that label the graph while *fname* is the filename to save the graph to.



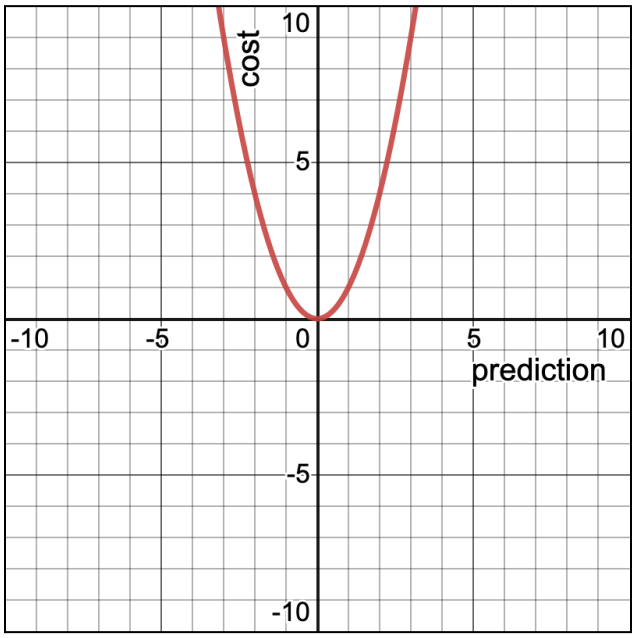
As you can see, for the weight our learning rate is **too high**, we increased the weight **too much**. Also, when we are decreasing the weight value, the learning rate is **too low**, we decreased the weight **too little**. This is mainly due to the fact that the partial derivative of the cost function w.r.t. the weight decreases: 2X * (ŷ - y), as the better weight value make ŷ closer and closer to y.



For the bias, our learning rate is **too low all the time**, we increased the bias **too little**. As a result, one **global learning rate** can be **too high** for one parameter but **too low** for another parameter. To fix this, the **AdaGrad** and **RMSProp** optimizers were created.

At the same time, to help when the learning is **too low**, where updates are small, we add **momentum** to SGD along with **learning rate decay**. Since sometimes both a **global learning rate** is a problem where the learning rate is **too low**, the **Adam** optimizer combines **RMSProp** and **momentum** together.

We will talk about the bolded and highlighted words in the next unit. For now let's better understand why the partial derivative of the cost function w.r.t. a parameter decreases as the better parameter value make ŷ closer and closer to y, by analyzing a graph of the MSE cost function.



Let's say this is the cost function for one example where the correct answer (y) is 0 (cost is 0, the model has zero error/cost when the prediction is the exact same as the correct answer). As you can see as the prediction (ŷ), gets closer and closer to 0 (the correct answer), the slope from **very steep**, drops to **barely steep**. This characteristic of MSE, is the reason why training slows down (updates to the parameters) become smaller and smaller.

Now you may think about MAE, however the dynamic nonlinearity of MSE makes it superior than MAE for 2 reasons:

1. During training: Parameters are updated as fast as possible in the beginning since the further the model's predictions to the actual prediction, the cost grows exponentially.
2. Even though parameter updates become smaller and smaller, fixing this issue with optimizers is easy.

Since MAE's graph looks like a V, where half is a linear slope and the other half another linear slope. Due to the linear slope, the partial derivative of MAE w.r.t. the parameters

are always the same unless another parameter is increased or decreased too much. As a result, under MAE, parameters are more likely to be increased or decreased too much unless the learning rate is decayed. We would like to save time by not increasing or decreasing parameters too much, that way we do not need to take more epochs going in the opposite direction to revert the mistake. Also, MAE does not have a speed boost in the early stages of training that quickly allow the model to arrive at good parameters. This is due to MSE's cost exponentially increasing as the prediction gets farther and farther away from the correct answer. MAE is not a heavy penalizer.

Now that we understand where SGD goes wrong, let's go fix those problems in the next unit.