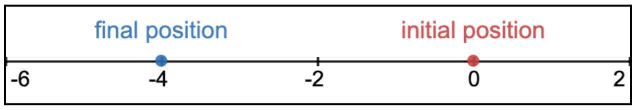
Chapter 06| Other Types of Optimizers

~~~~~~

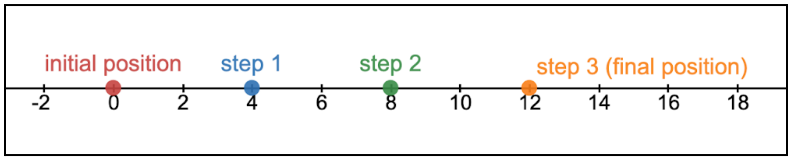
Momentum

This optimizer is heavily influenced by the physics properties of a particle rolling down a hill (cost function). As the particle rolls, its speed increases in the beginning due to an acceleration created by forces exerted on the particle. However, its speed will decrease as it reaches the bottom of the hill mainly due to friction exerting most of the **net force** (sum of the forces) on the particle. Before getting into the details of how momentum works, to better understand this optimizer, let's teach some basic physics.

The **velocity** of a particle is **different** from the **speed** of a particle. If you think about a **speed**ometer of a car, it gives you the **speed** of the car but it does not tell you the **direction** the car is moving, as a result, **speed** is always a positive value. If the **velocity** of a particle is **negative**, it is assumed that the particle is moving to the **left**, and if the **velocity** of a particle is **positive**, it is assumed that the particle is moving to the **right**. Picture a number line, and let's say you start at position 0 and take a step to the left:



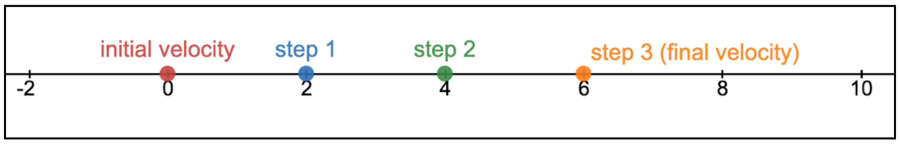
In this case, your **speed** is 4 units per step while your **velocity** is -4 units per step, where the negative indicates the **direction** (left). If you take 3 steps to the right, when starting from position 0:



The **speed** is still 4 units per step but your **velocity** is now 4 units per step, where the direction is to the right.

Knowing the difference between **speed** and **velocity** is essential because based on the number line: **velocity** is the rate at which the position changes. Speed cannot be the rate at which the position changes because it is **always positive** where you are **always moving to the right**. As a result, we can mathematically define the position **x** of a particle starting at position 0, given **n** the number of steps as: $x(n) = v*n$ where **v** is the **velocity** of the particle. If the **velocity** is 4, each step taken **increases** the position by 4. In almost all everyday life, the **velocity** of a particle does not stay the same, sometimes the velocity increases and sometimes the velocity decreases. The rate at which the
~~~~~~

velocity changes is the **acceleration**. On a velocity number line, if you start at rest (velocity equal to 0), but your acceleration is 2 the whole time:



This should be pretty simple to understand. Note that since you cannot raise your foot and take a **fraction** of a **step**, to end up in a new position, we will change the **velocity** using the **acceleration**, and then using the new velocity, we will change the **position**. A To mathematically define the velocity **v**, given **n**, the number of steps: v(n) = a*n, where **a** is the acceleration of the particle. Note that x(n) works under the condition that the velocity of the particle is always constant (the velocity is the same) and v(n) works under the condition that the acceleration of the particle is always constant.

As a result, it seems as if to update the position of the particle, we first update the velocity using v(n) and then update the position using x(n).

The problem is, a ball rolling down a hill does not have a constant acceleration, the velocity increases (positive acceleration) in the beginning when the ball is released and then decreases (negative acceleration) as the ball reaches the bottom of the hill, so at the same time, the ball does not have a constant velocity so x(n) and v(n) are both invalid. Here is how we fix the problem. Since, if you take a step, your position will increase by your velocity, but if you take a step, your velocity will increase by your acceleration. As a result, to take a step, use your **current** acceleration and add it to your **current** velocity, then use your **current** velocity to update your position. In code, we would have something like:

```
v += a
x += v
```

Now to make things more realistic with friction, we use the variable μ (greek lowercase letter mu) which will be a value between 0 and 1 to represent the **coefficient of friction**.

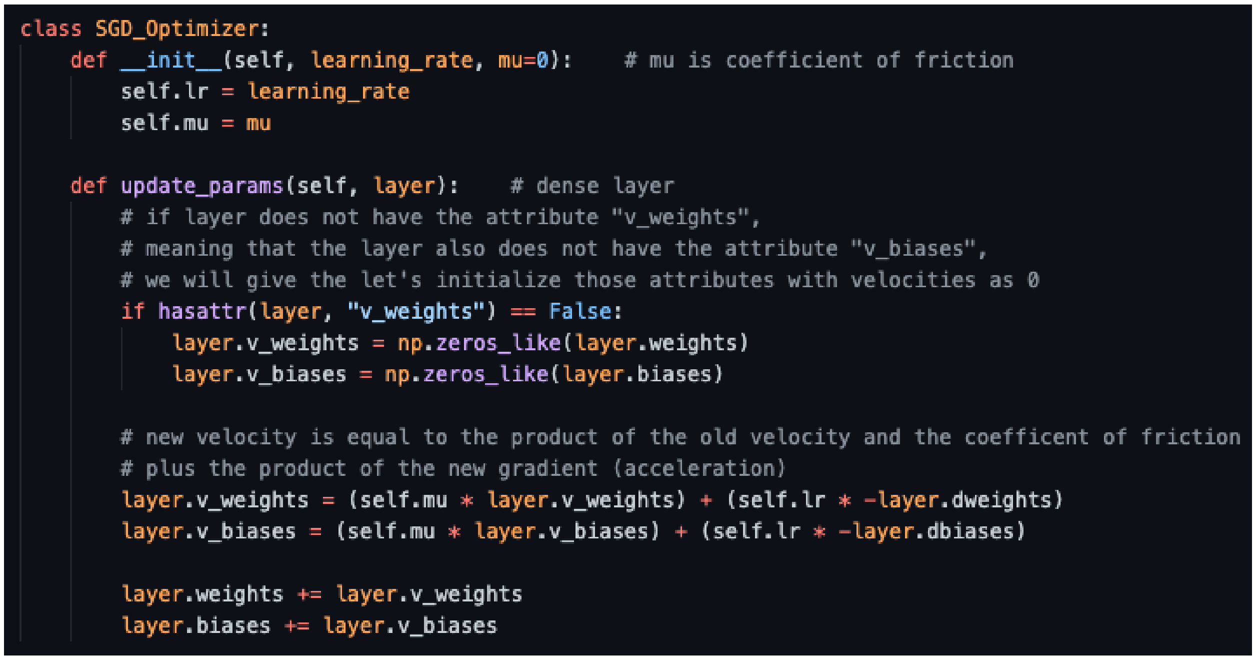
```
v = (μ * v) + a
x += v
```

The **coefficient of friction** will determine how much of the **current velocity** to keep. For example, if μ=0.5 and the current velocity is 4, we keep 50% of the current velocity to get 2 then add that to whatever the acceleration is to get our new velocity which we use to update the position. To finish off, we need to determine what our acceleration will be. Remember that acceleration of a particle is determined by the **net force** (sum of the forces) exerted on a particle. The forces exerted on a particle depends on the location of the particle. What else depends on the location of a particle? The **gradient** (learning rate * negative partial derivative of cost function w.r.t. a parameter). Knowing this, we can say that the acceleration is equal to the **gradient**:

$v = (\mu * v) + (\alpha * -\partial C)$

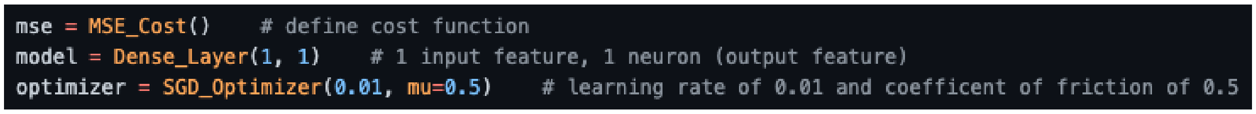
$x \mathrel{+}= v$

Now let's see how we actually implement this in code:



```
class SGD_Optimizer:
    def __init__(self, learning_rate, mu=0):    # mu is coefficient of friction
        self.lr = learning_rate
        self.mu = mu

    def update_params(self, layer):    # dense layer
        # if layer does not have the attribute "v_weights",
        # meaning that the layer also does not have the attribute "v_biases",
        # we will give the let's initialize those attributes with velocities as 0
        if hasattr(layer, "v_weights") == False:
            layer.v_weights = np.zeros_like(layer.weights)
            layer.v_biases = np.zeros_like(layer.biases)

        # new velocity is equal to the product of the old velocity and the coefficent of friction
        # plus the product of the new gradient (acceleration)
        layer.v_weights = (self.mu * layer.v_weights) + (self.lr * -layer.dweights)
        layer.v_biases = (self.mu * layer.v_biases) + (self.lr * -layer.dbiases)

        layer.weights += layer.v_weights
        layer.biases += layer.v_biases
```

In the initialization method, we add the keyword argument of mu where by default, we do not have momentum, since 0 multiplied by the current velocity is 0. In the update_params method, we begin by first checking that we have a velocity for each parameter. *np.zeros_like()*, takes in a numpy array and returns a numpy array with the same shape and type of the inputted array but all elements are equal to 0. This means, we are releasing a ball from rest (velocity of 0). Next, we update the velocities and then use the updated velocities to update the parameters. The only other change:



```
mse = MSE_Cost()    # define cost function
model = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
optimizer = SGD_Optimizer(0.01, mu=0.5)    # learning rate of 0.01 and coefficent of friction of 0.5
```

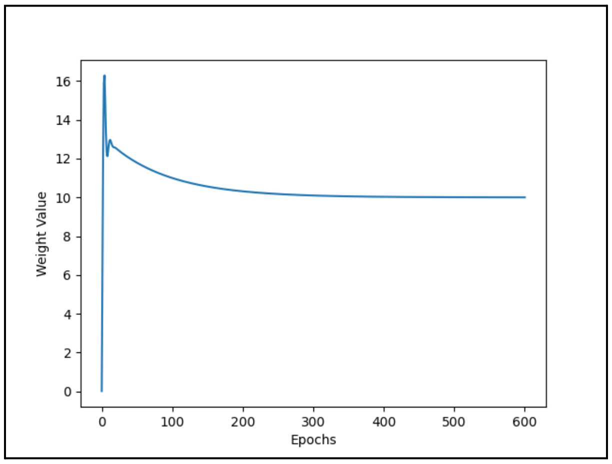
is the initialization of the optimizer where we use momentum by adding a coefficient of friction of 0.5 and change the number of epochs to 600. It is a misnomer to call mu the momentum of the particle when it is there to help slow the ball by applying friction.

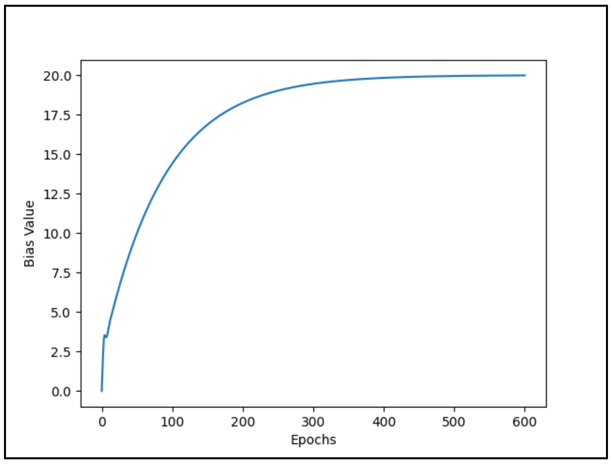
```
[[0.01764052]] [[0.]]
Original Cost: 4255.854199207604
[[10.00301801]] [[19.98311309]]
```

The new cost value: 8.390551839550478e-05 may seem weird with the e-05

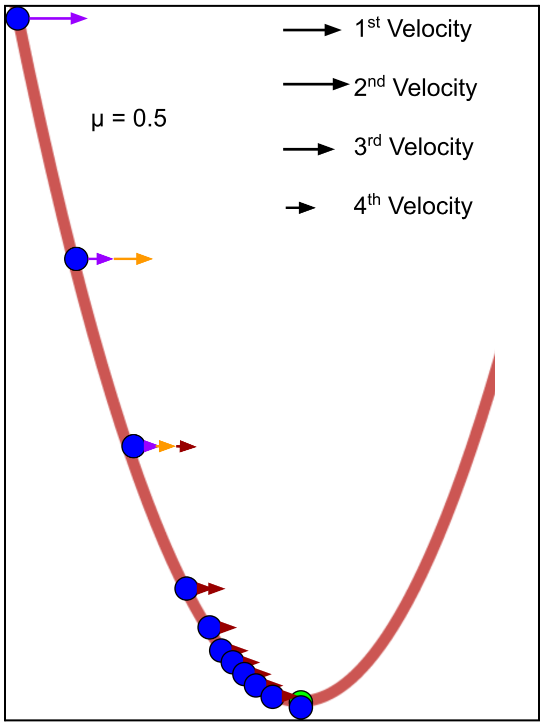
New Cost: 8.390551839550478e-05

at the end of the number, but in reality this is technically scientific notation where the new cost value is 8.390551839550478 * 10^{-5}. The e signifies multiplied by 10 to the power of and the remaining characters signifies the power to raise 10 to. As a result, our cost is actually ~0.000084. Compared to the cost we got in the last unit, ~0.0009, we did better in 400 less epochs!





As you can see, we still have the problem of the learning rate being too high for the weight due to increasing weight too much but the learning rate too low for the bias due to increasing the bias too slowly. However, overall, momentum did increase the bias faster and after increasing the weight too much, momentum did decrease the weight faster. To understand why momentum makes learning faster check out the graphic on the right.



The gradient is represented by the arrow furthest from the dot (model's location), and the arrows in between the dot and gradient are the velocity of the model after friction has been applied. The coefficient of friction (μ) is 0.5. As you can see, the model's first step only consists of the gradient. However, the model's second step consists of half the first step's gradient **plus** the gradient at the 2nd dot. The result is a velocity a little greater than the velocity of the 1st dot. This means that the acceleration is **positive** at the 2nd dot, where the ball is **speeding up** as it descends the hill.

The model's third step consists of half of half of the 1st step's gradient (a quarter) of the first step's gradient **plus** half of the gradient at the 2nd dot **plus** the gradient at the 3rd dot. The result is a velocity a little less than the velocity of the 3rd dot. This means that the acceleration is **negative** at the 3rd dot, where the ball is **slowing down** due to friction as the ball reaches the bottom of the hill. The model's fourth step consists of one-eighth of the 1st step's gradient **plus** one-quarter of the 2nd step's gradient **plus** one-half of the 3rd step's gradient **plus** the gradient at the 4th dot. The first two fractions of the gradient are so tiny, it's not visible on the graphic and the visible gradients are the tiniest. This result is a tiny velocity for the fourth step and all the other steps that come after.

Without momentum (the extra velocities from previous gradients), the model would need to take much more steps to reach the lowest cost. As a result, with the momentum optimizer, if the gradients are consistently in the same direction, the velocity will increase in the beginning similar to what happens when a ball begins to roll down a hill. As a result, we need to fix the problem of increasing the weight too much (relates to a ball rolling down a hill with too much velocity that it begins to roll a little up a hill next to the bottom of the hill it just fell from), that way, the model learns even faster as the bias velocity can increase more in the beginning so that we do not need extra epochs for the bias to increase more towards the end. We can do this in the next section, by using different learning rates for each parameter.

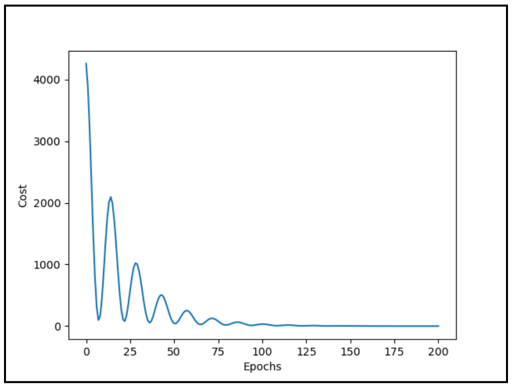
Root **M**ean **S**quare **P**ropagation (RMSProp)

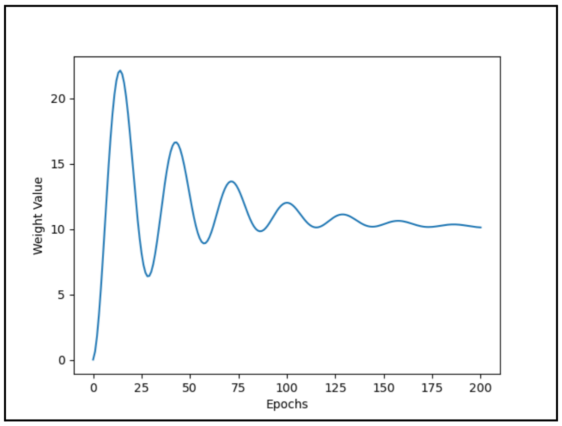
The name of this optimizer, RMSProp, mainly describes the calculations to get different learning rates for each parameter. This optimizer still includes the global learning rate alpha, α, but this is divided by a parameter-specific value to get the different learning rates for each parameter. The parameter-specific value is mainly determined by the amount each parameter is updated.

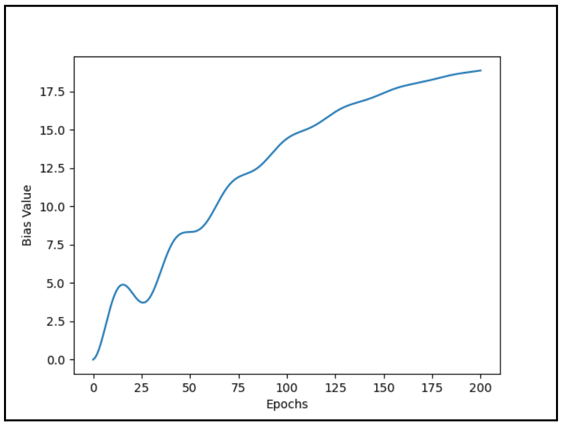
To understand the calculations used, let's first exaggerate our situation where we can decrease the coefficient of friction by **increasing** mu, meaning that friction would not slow us down as much since we keep **more** of the current velocity. Here is the code:

```
mse = MSE_Cost()     # define cost function
model = Dense_Layer(1, 1)     # 1 input feature, 1 neuron (output feature)
optimizer = SGD_Optimizer(0.001, mu=0.95)     # learning rate of 0.001 and coefficent of friction of 0.95
```

Also since we have a bigger velocity, our updates will be bigger so we will not need too many epochs so change the number of epochs to 200. Here the ending cost value and ending weight value is not our concern. Check out the history of the cost, weights, and biases.



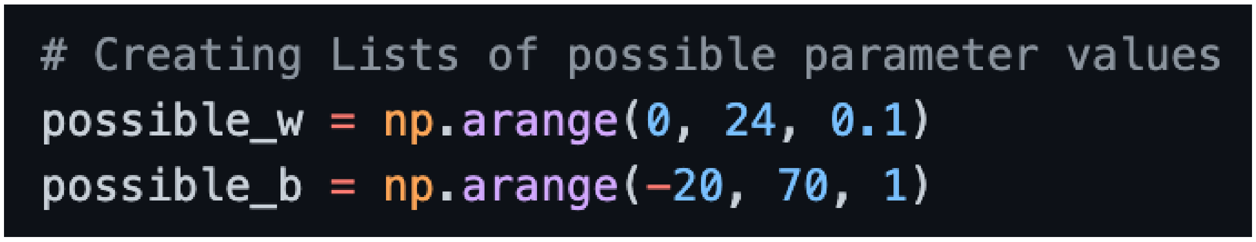




Everything is wiggling! The cost is wiggling because the weight value is jumping around, when it increases too much, it decreases too much, then increases too much, and so on. The causes of the derivatives of the cost w.r.t. the bias values to be inconsistent, slowing down the model's ability to reach the optimal value for the bias.

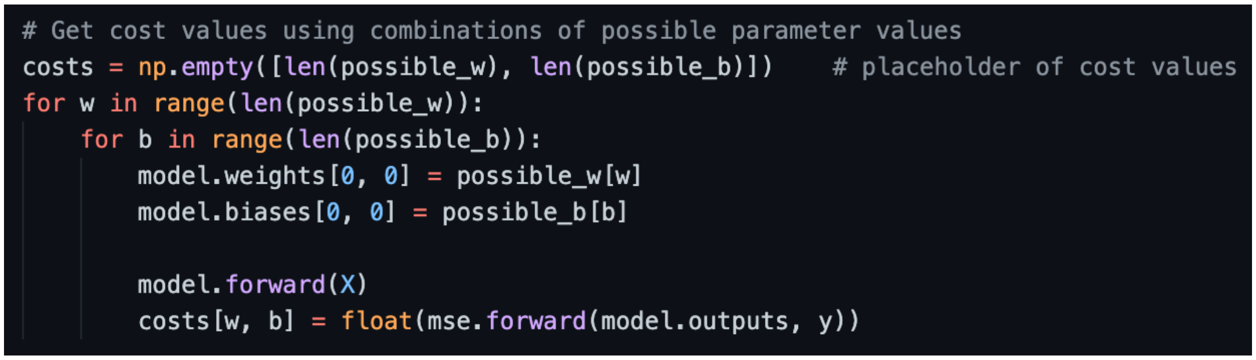
We can visualize the model jumping around using a **contour plot**. Contour plots can represent 3-dimensional surfaces by using a 2-dimensional surface. The plot is almost the same as a scatter plot where each point is defined by coordinates, (x, y). However, instead of seeing only x as the independent variable, y is also the independent variable. In this case, our independent variables are our parameters, say possible bias values on the x-axis and possible weight values on the y-axis. **The result of changing our independent variables (parameters of the model), are different predictions that produce different cost values**. As a result our dependent variable would be the cost. We represent **changes in cost values** by using the graphing space to create lines. Crossing each line by changing values for parameters, means different coordinates on the graph, means that the **cost value has changed**. Each line is differentiated with different colors, similar colored lines have similar cost values.

Here is the code to create a contour plot (we delete all the code right after we print the new cost and replace the plotting of the cost, weights and bias with the contour plot):



```python
# Creating Lists of possible parameter values
possible_w = np.arange(0, 24, 0.1)
possible_b = np.arange(-20, 70, 1)
```

The function *np.arange()* is almost equivalent to python's built-in *range()* function except that *np.arange()* can handle steps (last argument) that are less than 1. As a result for the weights, we have a list with values starting at 0, then increasing in increments of 0.1 until the next value is greater than or equal to 24 in which we **exclude/do not include** 24. For the biases, we have a list with values starting at -20, then increasing in increments of 1 until the next value is greater than or equal to 70 in which we **exclude/do not include** 70.



```python
# Get cost values using combinations of possible parameter values
costs = np.empty([len(possible_w), len(possible_b)])    # placeholder of cost values
for w in range(len(possible_w)):
    for b in range(len(possible_b)):
        model.weights[0, 0] = possible_w[w]
        model.biases[0, 0] = possible_b[b]

        model.forward(X)
        costs[w, b] = float(mse.forward(model.outputs, y))
```

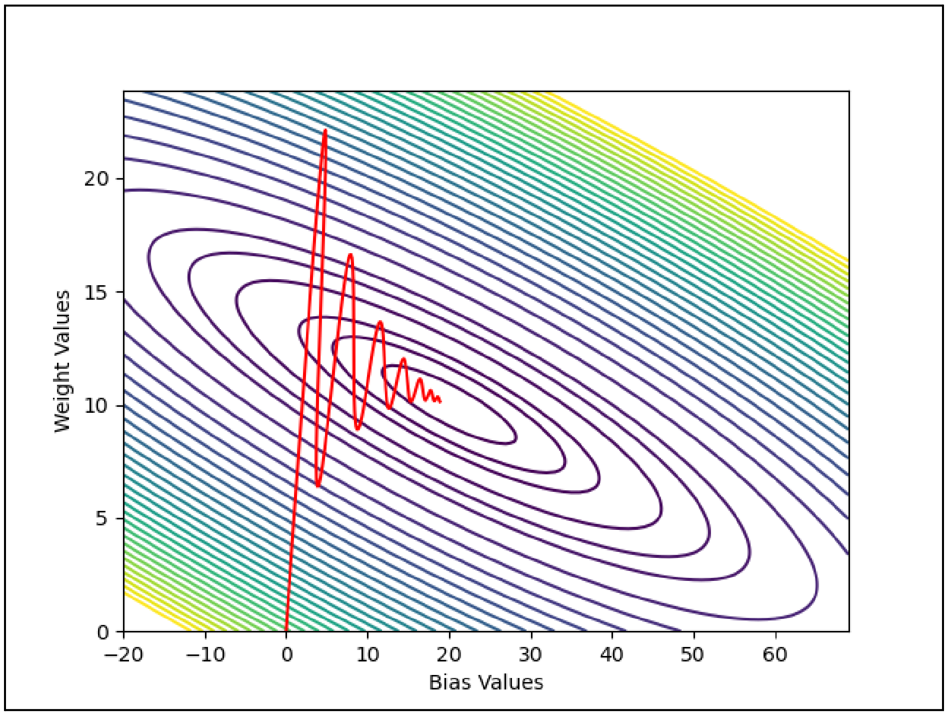
Now we create a placeholder of cost values using *np.empty()*, returning an empty array with the number of rows equal to the number of possible weight values we chose and then number of columns equal to the number of possible bias values we chose. For each possible weight value we have and each possible bias value we have, we can create combinations of possible parameter values (coordinates) on a graph. Using these parameter values we change the model's parameters to agree and then get a prediction. Using the new prediction, we fill in the corresponding cost element with its corresponding cost from the cost function that is interpreted as a float.

```python
# Setup for Plotting
fig = plt.figure()
levels = list(np.arange(20, 100, 40))
levels.extend(list(np.arange(100, 400, 100)))
levels.extend(list(np.arange(400, 6000, 200)))
```

Levels is a list of cost values we should draw a line for.

```python
# Plotting
plt.contour(possible_b, possible_w, costs, levels=levels)
plt.plot(bias_history, weight_history, color='red')
plt.xlabel("Bias Values")
plt.ylabel("Weight Values")
plt.savefig("contour.png")
plt.close()
```

Since rows are vertically stacked and columns are horizontally stacked, the rows (weight values) are plotted on the y-axis (2nd argument) and the columns (bias values) are plotted on the x-axis (1st argument). Here is the contour plot (contour.png):



As you can see, the high cost values are the yellowish lines and the smaller cost values are the purple lines. The middle where each line (ellipse) revolves around, is the lowest cost value where the weight is 10 and the bias is 20. The red line is the path of the model where the weight is increasing too much and decreasing too much while the bias is barely making progress. As a result, the updates of the weight are huge in comparison to the updates of the bias. The goal of RMSProp is to lower the updates in the weights and increase the updates in the bias. This is due to the fact that the updates in the weights are **high** and the updates in the bias are **low**. As a result, RMSProp will lower the update amount if the past few updates are high and will increase the update amount if the past few updates are low.

For RMSProp to know if the past few updates are low or high, we use parameter-specific values of *cache*. After every epoch, we change *cache* by using a **weighted average** (keyword "average" creates the "mean" in RMSProp). The equation for cache is as follows:

cache_w = (decay_rate * cache_w) + [(1 - decay_rate) * ∂C_w^2]
cache_b = (decay_rate * cache_b) + [(1 - decay_rate) * ∂C_b^2]

The *decay_rate* is usually 0.9, or it could be 0.99, or 0.999. For example, if the *decay_rate* was 0.9, the cache for a weight value would be 90% of the past cache the weight value plus 10% of the weight derivative **squared** (keyword "squared" creates the "square" in RMSProp). To provide intuition for why the *decay_rate* variable is called the "decay rate" with the decay_rate equal to 0.9:

When	Cache	Interpretation
Before Updates	0	No Cache
After 1st epoch	$(0.9 * 0) + (0.1 * \partial C_1^2) = 0.1\partial C_1^2$	**10%** of 1st derivative squared
After 2nd epoch	$(0.9 * 0.1\partial C_1^2) + (0.1 * \partial C_2^2) = 0.09\partial C_1^2 + 0.1\partial C_2^2$	**9%** of 1st derivative squared plus **10%** of 2nd derivative squared
After 3rd epoch	$[0.9 * (0.09\partial C_1^2 + 0.1\partial C_2^2) + (0.1 * \partial C_3^2) = 0.081\partial C_1^2 + 0.09\partial C_2^2 + 0.1\partial C_3^2$	**8.1%** of 1st derivative squared plus **9%** of 2nd derivative squared plus **10%** of 3rd derivative squared

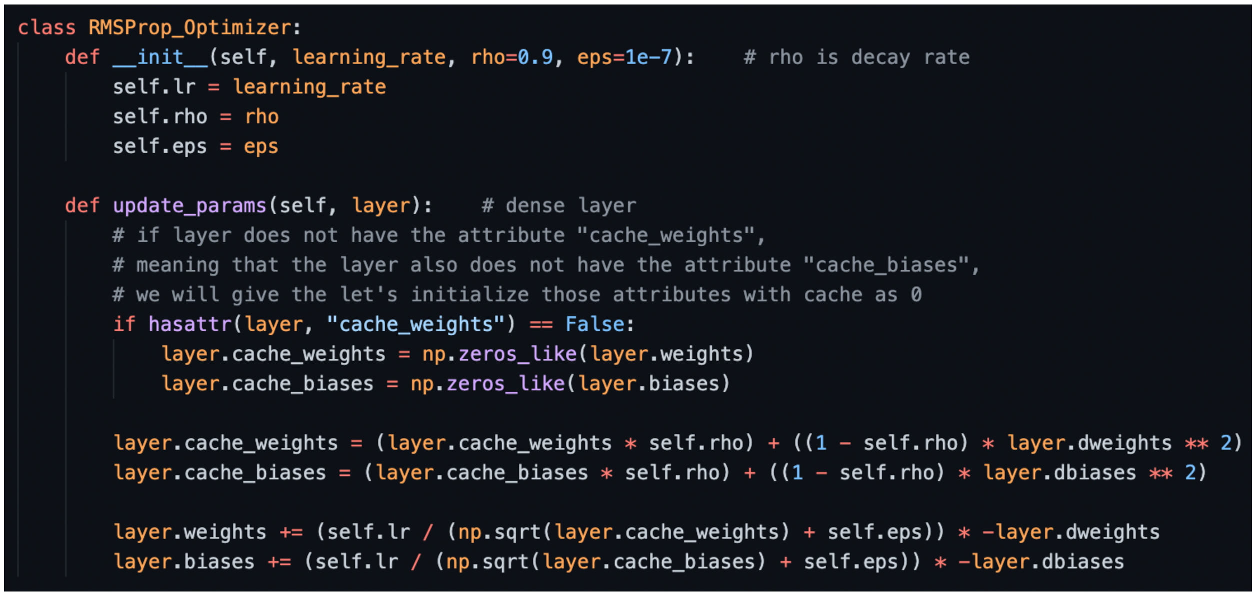
*"derivative" should be "partial derivative" but since this table uses one variable, using "derivative" is safe

As you can see we slowly **decay** the squared gradients using the **decay rate**. Now that we have the **cache**, how is the cache used in RMSProp? In RMSProp gradients are calculated via:

$$G_w = \frac{\alpha}{\sqrt{cache_w} + \epsilon} * -\frac{\partial C}{\partial w},\ G_b = \frac{\alpha}{\sqrt{cache_b} + \epsilon} * -\frac{\partial C}{\partial b}$$

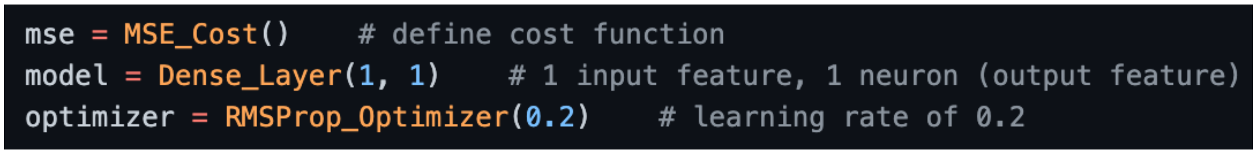
The new learning rate would be the global learning rate alpha, α divided by the quantity of the square root of the corresponding cache value plus epsilon. Note that the purpose of this epsilon is to prevent dividing by zero when the cache is 0, so epsilon is a **tiny** value close to 0 like 1e-7.

Due to **higher derivatives**, which means **higher updates** in the weight that grow the cache, the denominator of the new learning rate increases and as a result, will have **smaller updates**. Due to **smaller derivatives**, which means **smaller updates** in the bias, the cache will barely grow **relative** to the cache of the weight, the new learning rate **relative** to the weight will be higher and as a result, **seem** to give **bigger updates**. In essence, RMSProp helps by increasing the updates in parameters that have had smaller updates relative to the parameters that have had larger updates which will get smaller updates. Now here is the code for this new optimizer:



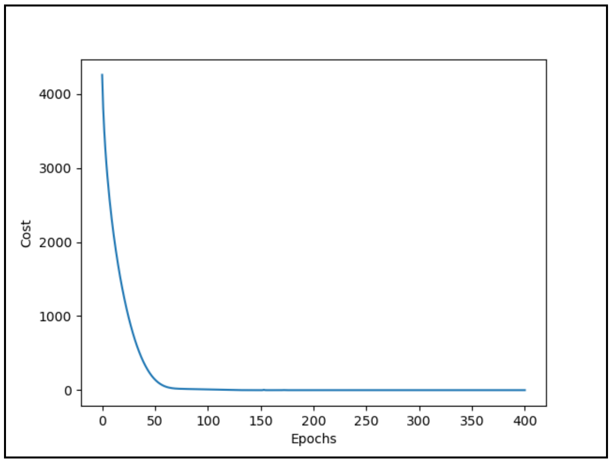
```
class RMSProp_Optimizer:
    def __init__(self, learning_rate, rho=0.9, eps=1e-7):    # rho is decay rate
        self.lr = learning_rate
        self.rho = rho
        self.eps = eps

    def update_params(self, layer):    # dense layer
        # if layer does not have the attribute "cache_weights",
        # meaning that the layer also does not have the attribute "cache_biases",
        # we will give the let's initialize those attributes with cache as 0
        if hasattr(layer, "cache_weights") == False:
            layer.cache_weights = np.zeros_like(layer.weights)
            layer.cache_biases = np.zeros_like(layer.biases)

        layer.cache_weights = (layer.cache_weights * self.rho) + ((1 - self.rho) * layer.dweights ** 2)
        layer.cache_biases = (layer.cache_biases * self.rho) + ((1 - self.rho) * layer.dbiases ** 2)

        layer.weights += (self.lr / (np.sqrt(layer.cache_weights) + self.eps)) * -layer.dweights
        layer.biases += (self.lr / (np.sqrt(layer.cache_biases) + self.eps)) * -layer.dbiases
```

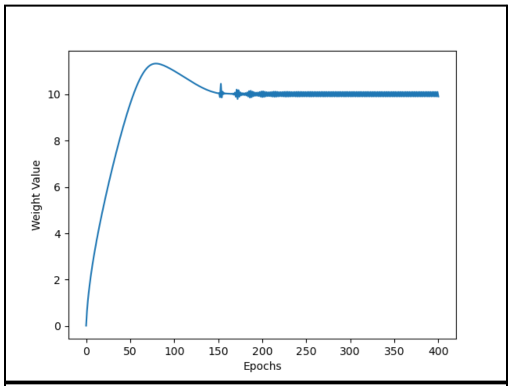
In the __init__ method we initialize the learning rate, rho, and epsilon, *rho* is the decay rate and *eps* is epsilon. Almost the same as momentum, we need a matrix for values for each parameter but instead of initializing velocities at 0, we initialize the cache for each parameter at 0. Next, we update the cache and then update the parameters using the modified learning rates.

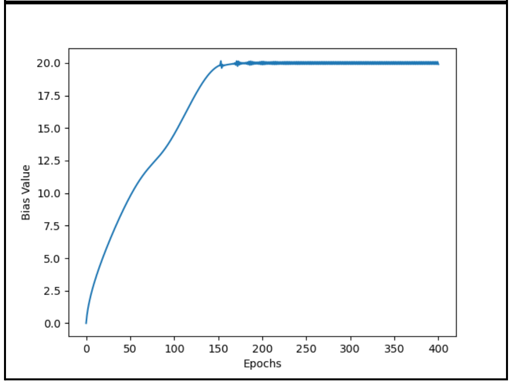


```
mse = MSE_Cost()     # define cost function
model = Dense_Layer(1, 1)     # 1 input feature, 1 neuron (output feature)
optimizer = RMSProp_Optimizer(0.2)     # learning rate of 0.2
```

We initialize RMSProp with a learning rate of 0.2 and change the number of epochs to 400. Now let's compare the history of the cost, weights and biases to momentum.

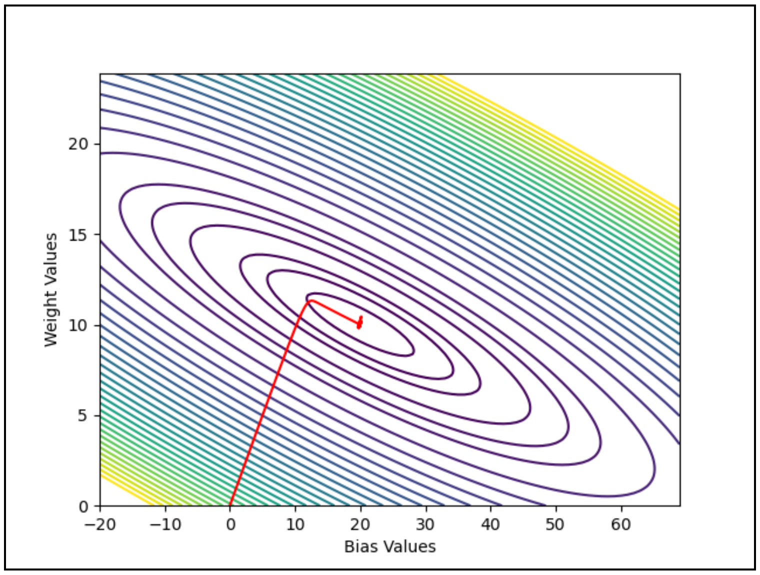






As you can see, the weight has **dampened** updates and the bias has updates that are more **consistent** without becoming smaller very early. The only problem is that the learning rate towards the end of training is too high. That is what is causing the darker lines on the graph. The darker lines mean that the weight and bias are jumping around their optimal value but they are also close to their optimal value. Looking at the final weight and value, we have ~9.9 and ~19.9 respectively. As a result, the model is unable to "**settle down**" by having parameters closer to the optimal value.

This is the motivation for learning rate decay, lowering the learning rate over time to help the model "**settle down**" into a lower cost. To better understand the meaning of "**settle down**", let's see a contour plot to see the model's path during the end of training. Here is the code (the format is the same, except the *SGD_Optimizer* class is replaced with the *RMSProp_Optimizer* class, the optimizer is initialized with a learning rate of 0.2 and trains for 400 epochs).



It seems like there is a red dot in the middle of the contour plot but actually that is the model moving around the optimal weight and bias value, trying to get to an even lower cost. The jumping around means that the model's learning rate is too high as it constantly increases/decreases both the weight and bias too much.

Before going into the next section, there is another optimizer that is similar to RMSProp, which is AdaGrad, **Ada**ptive **Gradient**. The only difference in AdaGrad is the calculation of the cache, the cache is increased by the squared gradient. Instead of taking a **weighted average** of squared gradients, AdaGrad keeps the old squared gradients in the cache forever. *In my opinion*, RMSProp is **more adaptive** than AdaGrad because the **more recent** squared gradients are given more **weight** in the **weighted average**. However, do not take my opinion seriously, as that is only based on the math behind how AdaGrad and RMSProp calculate cache for each parameter.

Now in the next section, let's help this model **"settle down"** for a lower cost.

Learning Rate Decay

Learning Rate **Decay** is all about **decreasing** the learning rate during training to help models **"settle down"** in the cost function where the model can **less aggressively** update its parameters as the model gets close to an optimal location.

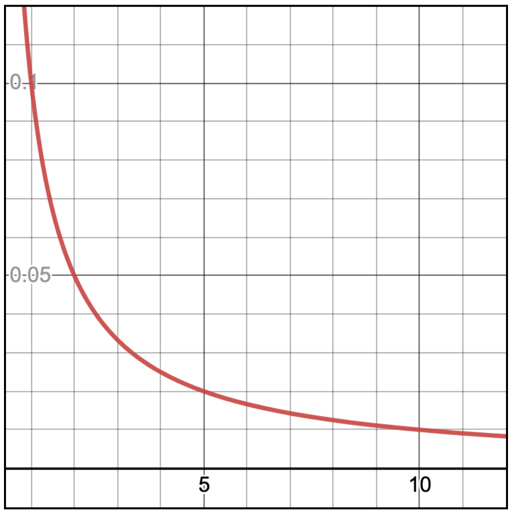
I will only present **one possible learning rate decay method**. There are way too many possible learning rate decay methods out there, each have pros and cons so further study is recommended. However, the purpose of all the learning rate decay methods are the same, to decrease the learning rate during training. Each method decays the learning rate by different **amounts** each time during training, and the **frequency** the learning rate is decayed can vary. Okay, enough speculation, let's get into it.

So the learning rate, α for a given epoch, *t*, and the initial learning rate α_0 is as follows:

$$\alpha(t) = \frac{a_0}{1+kt}$$

So *k* is a constant that we have to define at the beginning of training (just like how we have to define α_0 at the beginning of training). If we are on epoch 0, which is technically when we have not started training, the denominator will equal 1 so the learning rate will be α_0 divided by 1 which is just α_0, the initial learning rate.

Basically we are dividing the learning rate by a value greater than or equal to 1. The value of the denominator increases linearly as it is a linear function, y = mx + b. Think of x as the epoch number, *t*, the slope, m, as *k* the constant, and the y-intercept, b as 1.



As a result, the denominator increases by *k* (a constant amount) every time an epoch passes (t increases by 1). However, this does not mean that the learning rate decreases by the same amount after every epoch. If we plot the learning rate on the y-axis and the value of the denominator in the x-axis, and the initial learning rate is 0.1, we get the graph on the right. The y-value is the initial learning rate (0.1) when the x-value is 1 because the x-axis represents the **denominator**. For those close to the end of high-school math, the graph on the right resembles a rational function (1/x).

Our learning rate decay method quickly decreases the learning rate in the first few epochs. We do this because if you look at the cost history from RMSProp, the cost is relatively low after 50 epochs and then the cost begins to slowly decrease. When the cost begins to slowly decrease however, we need to decay the learning rate slower because we do not want learning to **stop** but learning so that the model can "**settle down**" by reaching lower parts of the cost function.

Now that we slightly understand the learning rate decay method more by analyzing each part of the equation, let's implement this bad boy! Here is the code:

```python
class Learning_Rate_Decayer:
    def __init__(self, optimizer, decay_factor):
        self.epochs = 0
        self.optimizer = optimizer
        self.initial_lr = optimizer.lr
        self.decay_factor = decay_factor

    def update_learning_rate(self):
        self.epochs += 1
        self.optimizer.lr = self.initial_lr / (1 + (self.decay_factor * self.epochs))
```

In the __init__ method we first start with the number of epochs equal 0 since when we initialize the decayer, training should have not started yet. The first argument takes in the optimizer so we can update the learning rate of the optimizer in the *update_learning_rate* method and get α_0 (the initial learning rate) from the optimizer's current learning rate. The last argument is the *decay_factor*. Usually after we call the optimizer's *update_params* method, one epoch passes and we call the decayer's *update_learning_rate* method. Thus, in the *update_learning_rate* method we first increase the number of epochs by 1 and then calculate the new learning rate.

```python
mse = MSE_Cost()    # define cost function
model = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
optimizer = RMSProp_Optimizer(0.2)    # learning rate of 0.2
decayer = Learning_Rate_Decayer(optimizer, 0.01)
```

Right after we initialize the optimizer, we initialize the decayer with a *decay_rate* of 0.01. To better interpret the *decay_rate*, this means that the denominator increases by 1 every 100 epochs (1 / 0.01 = 100). Recall that the denominator is $1 + kt$, if t was 100, with k = 0.01, the product of k and t is 1, which increases the whole denominator by 1. Now so far we have been tracking the model's cost, weight and bias, let's also track the optimizer's learning rate.

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

In addition, we add an empty list as a placeholder for *lr_history*. To append *lr_history* and change the learning rate of the optimizer, we need to make some changes to the training loop.

```python
for epochs in range(400):
    # 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))
    lr_history.append(optimizer.lr)

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

    # update parameters & learning rate
    optimizer.update_params(model)
    decayer.update_learning_rate()
```

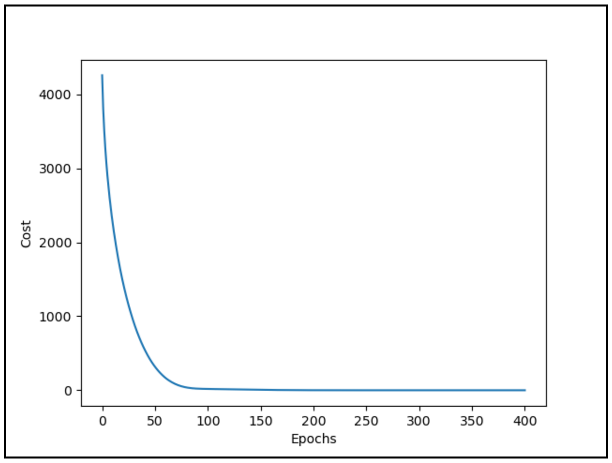
We append the optimizer's learning rate to *lr_history* right after we append the model's cost to *cost_history*. Also, right after we update the model's parameters, we update the optimizer's learning rate. Let's also add the learning rate at the end of the training loop (last line of code below).

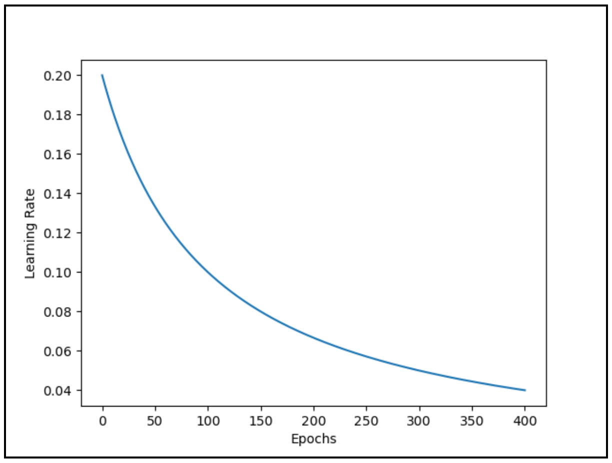
```python
# Check New Cost
model.forward(X)
weight_history.append(float(model.weights))
bias_history.append(float(model.biases))
print(model.weights, model.biases)
cost_history.append(mse.forward(model.outputs, y))
print("New Cost:", cost_history[-1])
lr_history.append(optimizer.lr)
```

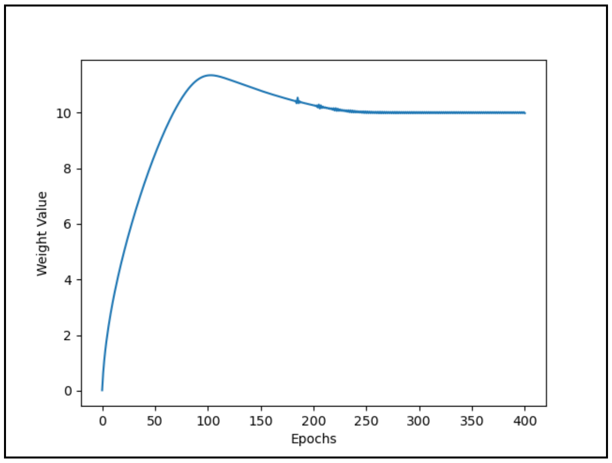
Oh, don't forget about graphing!

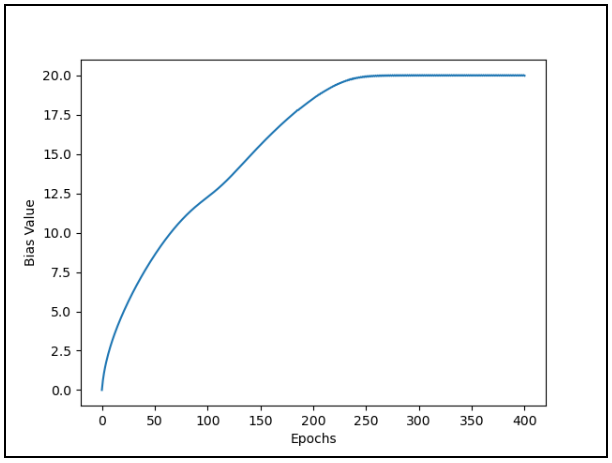
```python
graph(weight_history, "Epochs", "Weight Value", "weight.png")
graph(bias_history, "Epochs", "Bias Value", "bias.png")
graph(cost_history, "Epochs", "Cost", "cost.png")
graph(lr_history, "Epochs", "Learning Rate", "lr.png")
```

Good thing, we have the *graph* function where we have the y-axis representing the learning rate. Save the graph to *lr.png*.



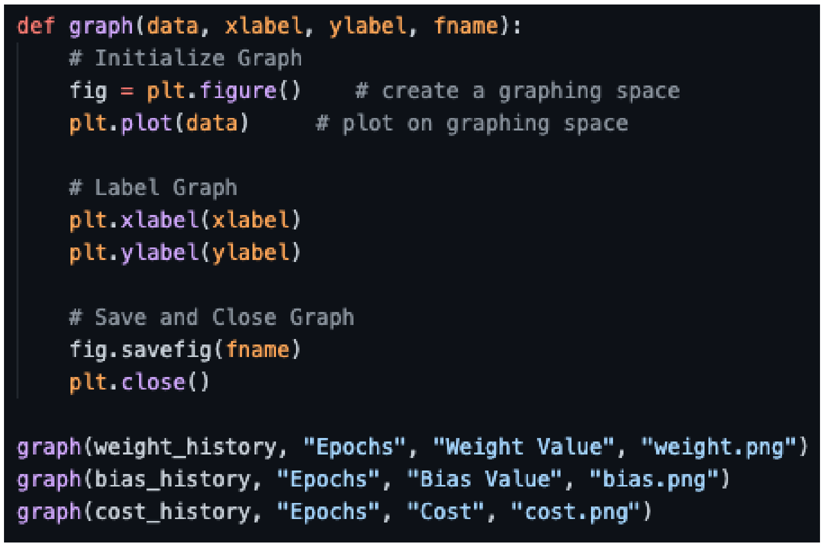




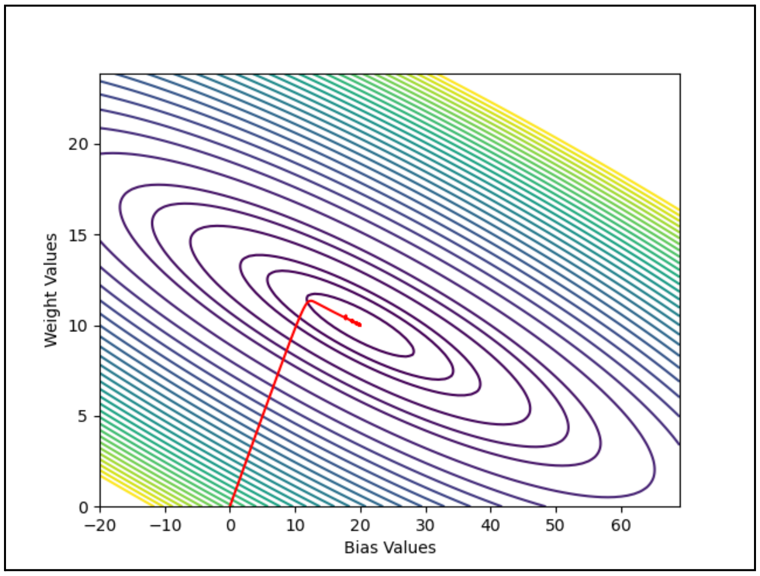


This is much less chaotic because the flat line towards the end of training is no longer as dark! Looking at the ending weight and bias, we get ~9.98 and ~19.98 respectively. We did indeed **"settle down"** the model as it is able to be much closer to the optimal values. With learning rate decay, for both parameters, they are about 0.02 from the optimal values while without learning rate decay, for both parameters, they are about 0.1 from the optimal values. We can further see the **"settle down"** with another contour plot.

Here is the code (copy and paste the learning rate decay class, initialize the learning rate decayer after initializing the optimizer, add a learning rate history placeholder, append during the training loop, and append after the training loop, the rest of the code stays the same).



```
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")
```



Instead of the main wiggle being vertical (wiggling up and down), the main wiggle is horizontal (wiggling left to right). Based on the cost function’s contour plot, the main wiggle being horizontal is much calmer. This is because there are more **contour lines** when moving up and down, as a result, changing the weight value will change the cost much more than changing the bias value.

Recall that each contour line means the cost changes, as a result, wiggling vertically (jumping around the optimal weight value), will change the cost significantly relative to wiggling horizontally (jumping around the optimal bias value).

Now the first 200 of the 400 epochs get the parameters close to the optimal value, the rest of epochs get the parameters **closer** to the optimal value. Reaching an optimal value **very close** to the optimal value means that the model has **converged** (it is **smart**, there is no need for more training). As you can see, getting the model **close** to the optimal value (first 200 epochs) is easier than getting the model **closer** to the optimal value (last 200 epochs). As a result, let’s use less epochs to get the model **close** to the optimal value. To do this, without increasing the weight or bias too much, we need to combine **momentum** and **RMSProp** together. The combination results in a new optimizer, Adam which will be discussed in the next section.

Adaptive **M**omentum (Adam)

It may seem easy to combine **momentum** and **RMSProp** but recall the velocity calculation for momentum and cache calculation for RMSProp:
$v = (\mu * v) + (\alpha * -\partial C)$ cache = (decay_rate * cache) + [(1 - decay_rate) * ∂C^2]
We use the learning rate to figure out the velocity of the parameter but the learning rate of a parameter depends on the RMSProp calculation for the cache of a parameter. However, both the calculations for a new velocity or cache, the first term uses a variable we set (for velocity, μ, and for cache, *decay_rate*). In the second term, for the velocity we use the partial derivative of the cost function w.r.t. the parameter but for the cache, we use the squared partial derivative of the cost function w.r.t. the parameter.

All of this so far is fine, except when velocity uses α (the learning rate) in the second term since the learning rate of a parameter depends on the cache of a parameter. To fix this, we need to get rid of α in the second time for velocity. Instead we can "steal" from RMSProp. What we will do is replace α with $(1 - \mu)$, so that the velocity calculation is now a **weighted average** like RMSProp. What we get is:

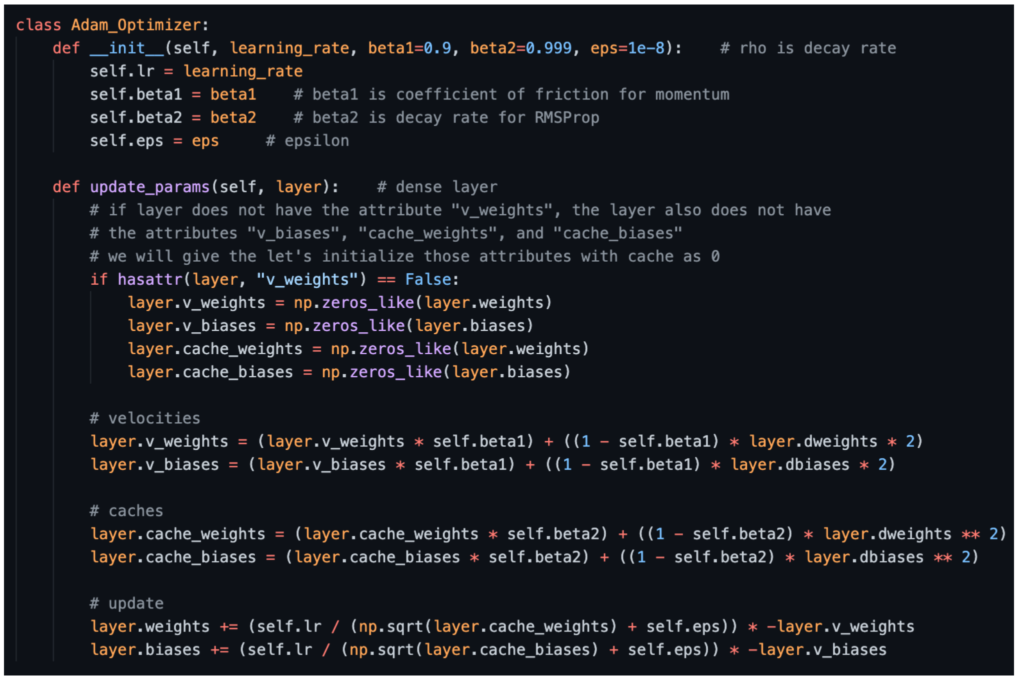
$v = (\mu * v) + [(1 - \mu) * \partial C)]$

Instead of using the negative derivative, we use the positive derivative because you can think of v as the velocity but like a new derivative to use to calculate the gradient. So now our parameter update will be:

$$G = \frac{\alpha}{\sqrt{cache} + \epsilon} * - v$$

Treating v as a new derivative, we negate the derivative in the gradient calculation.

To clean things up, here is a **draft** of the code for Adam:



```python
class Adam_Optimizer:
    def __init__(self, learning_rate, beta1=0.9, beta2=0.999, eps=1e-8):    # rho is decay rate
        self.lr = learning_rate
        self.beta1 = beta1    # beta1 is coefficient of friction for momentum
        self.beta2 = beta2    # beta2 is decay rate for RMSProp
        self.eps = eps     # epsilon

    def update_params(self, layer):    # dense layer
        # if layer does not have the attribute "v_weights", the layer also does not have
        # the attributes "v_biases", "cache_weights", and "cache_biases"
        # we will give the let's initialize those attributes with cache as 0
        if hasattr(layer, "v_weights") == False:
            layer.v_weights = np.zeros_like(layer.weights)
            layer.v_biases = np.zeros_like(layer.biases)
            layer.cache_weights = np.zeros_like(layer.weights)
            layer.cache_biases = np.zeros_like(layer.biases)

        # velocities
        layer.v_weights = (layer.v_weights * self.beta1) + ((1 - self.beta1) * layer.dweights * 2)
        layer.v_biases = (layer.v_biases * self.beta1) + ((1 - self.beta1) * layer.dbiases * 2)

        # caches
        layer.cache_weights = (layer.cache_weights * self.beta2) + ((1 - self.beta2) * layer.dweights ** 2)
        layer.cache_biases = (layer.cache_biases * self.beta2) + ((1 - self.beta2) * layer.dbiases ** 2)

        # update
        layer.weights += (self.lr / (np.sqrt(layer.cache_weights) + self.eps)) * -layer.v_weights
        layer.biases += (self.lr / (np.sqrt(layer.cache_biases) + self.eps)) * -layer.v_biases
```

In place of *mu* is *beta1* and in place of *decay_rate* is *beta2*. Default values for *beta1* (coefficient of friction for momentum)is 0.9, for *beta2* (decay rate) is 0.999, and for epsilon is 1e-8. Now this is a **draft!!!!** We still need to implement **bias-correction**. Hold up, the **bias** in this case **does not** refer to the model's bias parameter. Recall, how decay rate in RMSProp works (beta2 or decay rate is equal to 0.999):

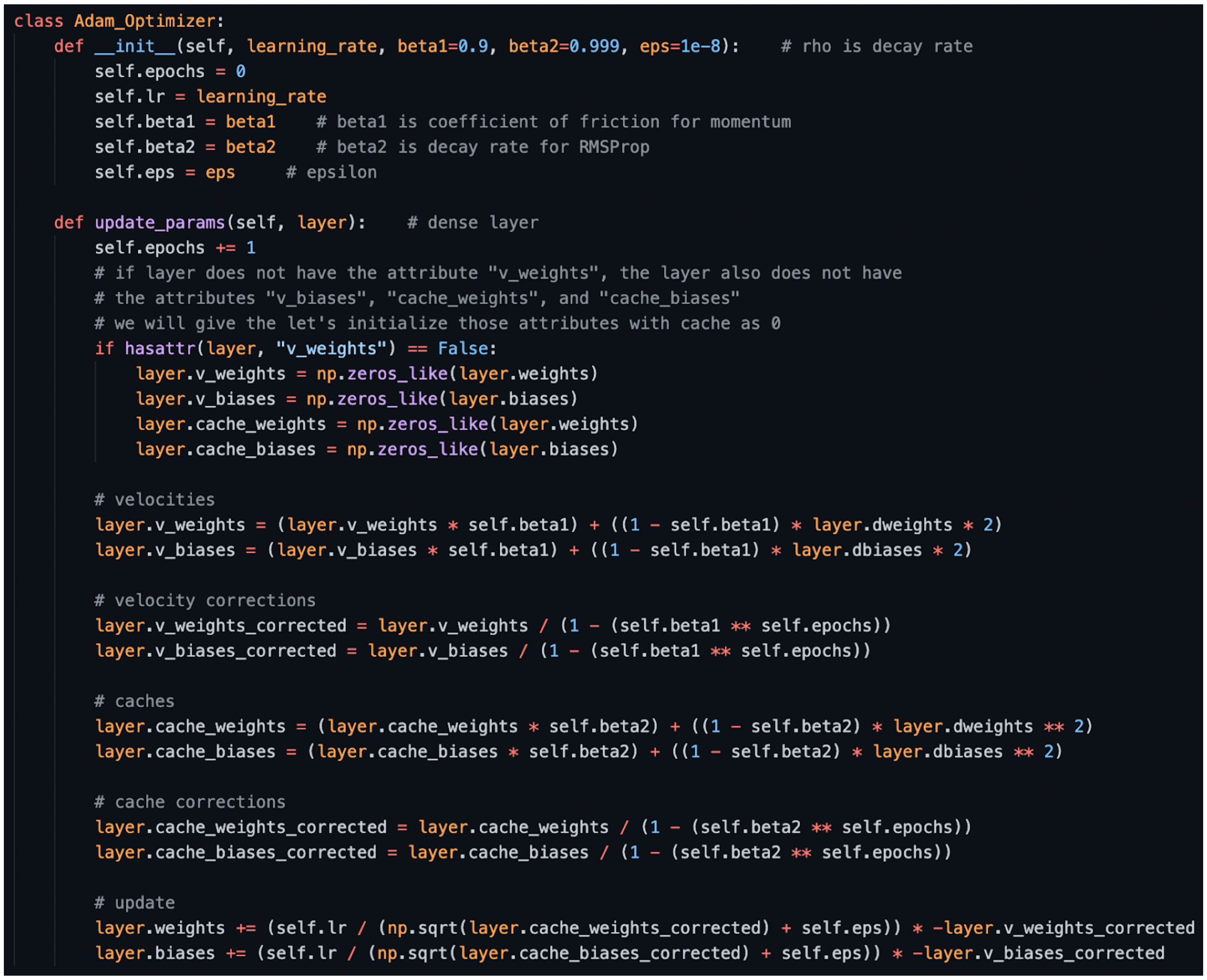
When	Cache	Interpretation
Before Updates	0	No Cache
After 1st epoch	$(0.999 * 0) + (0.001 * G_1^2)$ $= 0.001G_1^2$	**0.1%** of 1st derivative squared
After 2nd epoch	$(0.999 * 0.001G_1^2) + (0.001 * G_2^2)$ $= 0.000999G_1^2 + 0.001G_2^2$	**0.0999%** of 1st derivative squared plus **0.1%** of 2nd derivative squared

The problem here is that there is **"bias"** (the type always in humans). The cache in the very beginning will be **too small** since we use 0.1% of the 1st squared derivative after the 1st epoch. In reality, we are doing a weighted average without any other quantities as a result, we should be using 100% of the 1st squared derivative after the 1st epoch. This goes the same for the weighted average for the velocities, we need to **correct this "bias"** that the cache or velocity is low at the beginning of training. Here is the **bias correction calculation**:

$$v_c = \frac{v}{1-b_1^{\ t}} \qquad cache_c = \frac{cache}{1-b_2^{\ t}}$$

The velocity corrected is the velocity divided by the difference of 1 minus and beta1 to the power of t, current epoch number. The cache corrected is the cache divided by the difference of 1 and beta2 to the power of t. If we are on epoch 1 (we are currently in process of doing the first update), beta1 and beta2 to the first power would be beta1 and beta2 respectively so if beta1 = 0.9 and beta2 = 0.999. Now the denominator is 0.1 and 0.001 respectively. As a result, we are **undoing** the multiplication (weighted average) we did to get the velocity or cache. As t grows, taking beta1 and beta2 to the power of t, bases less than one, beta1 and beta2 to the power of gets closer and closer to 0. Thus, the denominator becomes closer to 0 so the **bias correction** slowly disappears as there is no more need for it in later epochs.

Here is the **full** code for Adam:



```python
class Adam_Optimizer:
    def __init__(self, learning_rate, beta1=0.9, beta2=0.999, eps=1e-8):    # rho is decay rate
        self.epochs = 0
        self.lr = learning_rate
        self.beta1 = beta1    # beta1 is coefficient of friction for momentum
        self.beta2 = beta2    # beta2 is decay rate for RMSProp
        self.eps = eps     # epsilon

    def update_params(self, layer):    # dense layer
        self.epochs += 1
        # if layer does not have the attribute "v_weights", the layer also does not have
        # the attributes "v_biases", "cache_weights", and "cache_biases"
        # we will give the let's initialize those attributes with cache as 0
        if hasattr(layer, "v_weights") == False:
            layer.v_weights = np.zeros_like(layer.weights)
            layer.v_biases = np.zeros_like(layer.biases)
            layer.cache_weights = np.zeros_like(layer.weights)
            layer.cache_biases = np.zeros_like(layer.biases)

        # velocities
        layer.v_weights = (layer.v_weights * self.beta1) + ((1 - self.beta1) * layer.dweights * 2)
        layer.v_biases = (layer.v_biases * self.beta1) + ((1 - self.beta1) * layer.dbiases * 2)

        # velocity corrections
        layer.v_weights_corrected = layer.v_weights / (1 - (self.beta1 ** self.epochs))
        layer.v_biases_corrected = layer.v_biases / (1 - (self.beta1 ** self.epochs))

        # caches
        layer.cache_weights = (layer.cache_weights * self.beta2) + ((1 - self.beta2) * layer.dweights ** 2)
        layer.cache_biases = (layer.cache_biases * self.beta2) + ((1 - self.beta2) * layer.dbiases ** 2)

        # cache corrections
        layer.cache_weights_corrected = layer.cache_weights / (1 - (self.beta2 ** self.epochs))
        layer.cache_biases_corrected = layer.cache_biases / (1 - (self.beta2 ** self.epochs))

        # update
        layer.weights += (self.lr / (np.sqrt(layer.cache_weights_corrected) + self.eps)) * -layer.v_weights_corrected
        layer.biases += (self.lr / (np.sqrt(layer.cache_biases_corrected) + self.eps)) * -layer.v_biases_corrected
```

Since we need the epoch number in the bias correction, we initialize self.epochs in the __init__ method as 0. In the beginning of update_params we increase the epoch count so that self.epochs represents the current epoch we are on. We correct the velocities of both the weights and biases by dividing by the difference between 1 and beta1 raised to the current epoch number. We correct the cache of both the weights and biases by dividing the difference between 1 and beta2 raised to the current epoch number. Note that when we update the weights and biases be used the corrected cache and velocities and that we do not perform corrections on the original variable. We create a new variable to perform corrections.

Let's start with a learning rate of 0.1:

```
[[0.01764052]] [[0.]]
Original Cost: 4255.854199207604
[[10.59794441]] [[16.79866031]]
New Cost: 3.038974168134756
```

As you can see the weight is close to its optimal value and the bias still needs to make some progress.

Hyperparameter Tuning

Up until now, the variables we set:

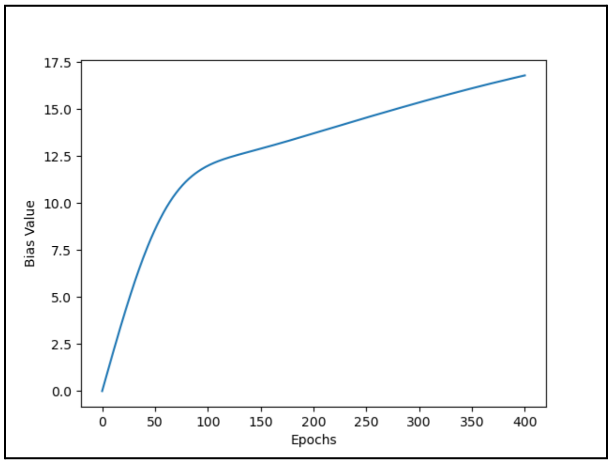
Optimizer/Extension	Symbol	Meaning	Common Value(s)
SGD	α	Learning Rate	less than 0.1 (0.01, 0.001)
Momentum	μ	Coefficient of Friction	0.5, 0.9, 0.95, 0.99
RMSProp	*NA*	Decay Rate	0.9, 0.99, 0.999
RMSProp	ε	Epsilon	1e-04 ~ 1e-08
Learning Rate Decay	k	Decay Factor	*depends*

Since all of these optimizers/extensions have the learning rate as a parameter, learning rate was only introduced in SGD to prevent repetition. There is not a symbol for the decay_rate of RMSProp and the learning rate decay factor depends on the training progress and is therefore different for each model. In Adam, the parameters, *beta1, beta2,* and *eps*, usually are not changed and remain 0.9, 0.999, and 1e-08, respectively. These parameters are not parameters that the model can update, as a result, we call them **hyperparameters** to differentiate.

Hyperparameters, unlike parameters (weights & biases) that the model updates, are what the human/creator of the model must choose to train the model. You will spend most of your time collecting data to train the model and training the model through the selection of hyperparameters, little time is needed to define the model. As a result, it is best to gain hyperparameter tuning experience early as there really is not a clearly defined method to effectively choose hyperparameters for training yet due to how recent the field of AI is.

Either you, yourself conduct trial-and-error or write another program that conducts trial-and-error by randomly generating hyperparameter values and training models on these and use the hyperparameter combinations that produce the lowest cost. Mainly this section, you will find hyperparameters yourself as the theories/methods proposed for randomly generating hyperparameters are too complex and out of scope for only an introduction to AI. We do go in-depth, but are unable to dive too deep.
First, examine/analyze the cost, weight and bias graphs created from the output of the last section. What graph should we focus on? Pause and figure this out yourself.

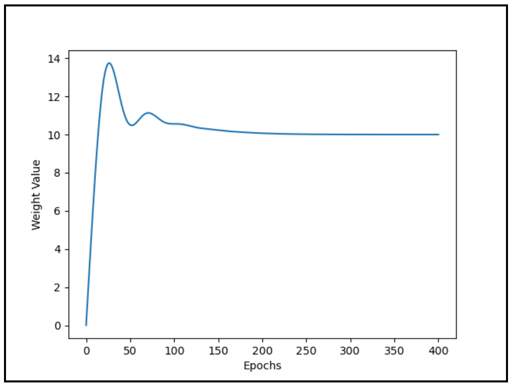
~~~~~~



At this point, looking at the plot of the bias, as we are training from the last section,

The bias increased by 10 in the first 100 epochs and then the learning slowed down a lot to the point that in the span of 300 epochs, we could not even increase by another 10. As a result, we should increase the learning rate so the bias can learn faster. The question is how much? Keeping the number of epochs at 400, see if you can get a cost lower than ~0.000084 (8.4e-05, the result from the momentum section).
~~~~~~

Pushing up to 0.4, what graph should we focus on? We see the weight increasing after it increased too much even though the weight was already above 10.



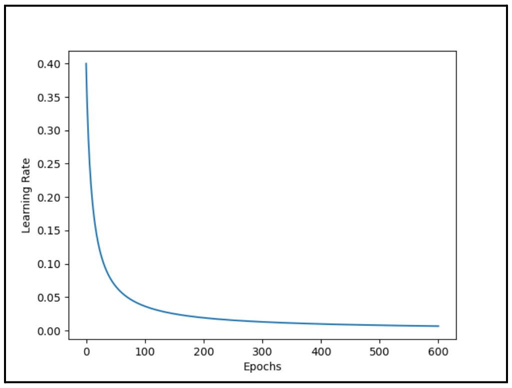
The weight should continue decreasing to 10 instead of jumping back up. Since the model is updating parameters wrong in the middle of training, the learning rate needs to be reduced during training to prevent this behavior. Let’s add in learning rate decay and since the average learning rate during training is smaller, updates are smaller, let's give the model with 600 epochs.

Here is the code for adding in learning rate decay with a decay factor of 0.1.

```
[[0.01764052]] [[0.]]
Original Cost: 4255.854199207604
[[11.01739362]] [[14.5434682]]
New Cost: 8.824249133125665
```

Here the weight is a little too high but the bias still needs to make progress. Even though we allowed the model to update 200 more times (200 more epochs), we still have work to do.

DO NOT INCREASE THE NUMBER OF EPOCHS



Doing so will waste your time due to the learning rate decayer decaying the learning rate to almost 0. As a result, either the decay factor of 0.1 is too high or the initial learning rate of 0.4 is too low. You figure that out and tune only the learning rate and decay factor hyperparameters. Make sure the weight does not increase even though it is already above 10. LEAVE THE NUMBER OF EPOCHS AT 600. Please continue reading after you have tried at least 20 different combinations of learning rate and decay factors and I will present my solution.

~~~~~~

Here is the code for my result. Making sure that the weight does not increase even though it's already above 10, I receive the output:

```
[[0.01764052]] [[0.]]
Original Cost: 4255.854199207604
[[10.02210644]] [[19.88169107]]
New Cost: 0.004150968736347434
```

It is possible, you have received a lower cost than me in 600 epochs without the the weight increasing when it’s already above 10. If so, Great Job! If you failed to do better than me, A + for effort!

The purpose of this section was to gain experience of tuning hyperparameters and to better understand the challenges that come with training a model. Also this is a good example that using Adam, a more complex optimizer that combines momentum and RMSProp, does not always guarantee a lower cost. The cost of 0.004 in 600 epochs did not beat the momentum optimizer cost but our results are comparable to RMSProp but with 200 more epochs. In the next unit, we will use what we have learned to create and train a model on a real-life dataset.
~~~~~~