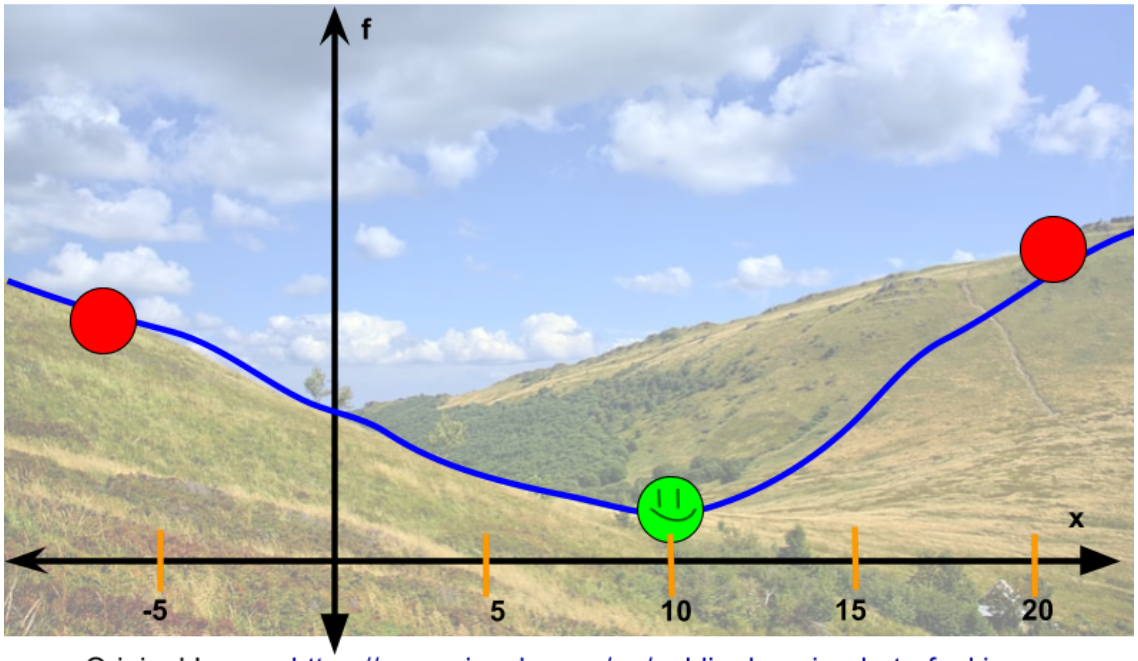
Chapter 04| Gradient Descent: The Perfect Algorithm To Use In Training

~~~~~~

Connection Of Hill To Cost Function & Using Slope at a Point (derivative) to determine Update Direction

To understand **gradient descent** you must first understand the name of this method. Let us say that you plot a graph of the function f(x). The output of f(x) is the cost of the model so f(x) is the cost function. Also the input x, is one of the weight values of the weight matrix. The graph could look something like a hill:



Original Image: https://www.piqsels.com/en/public-domain-photo-fpgkj

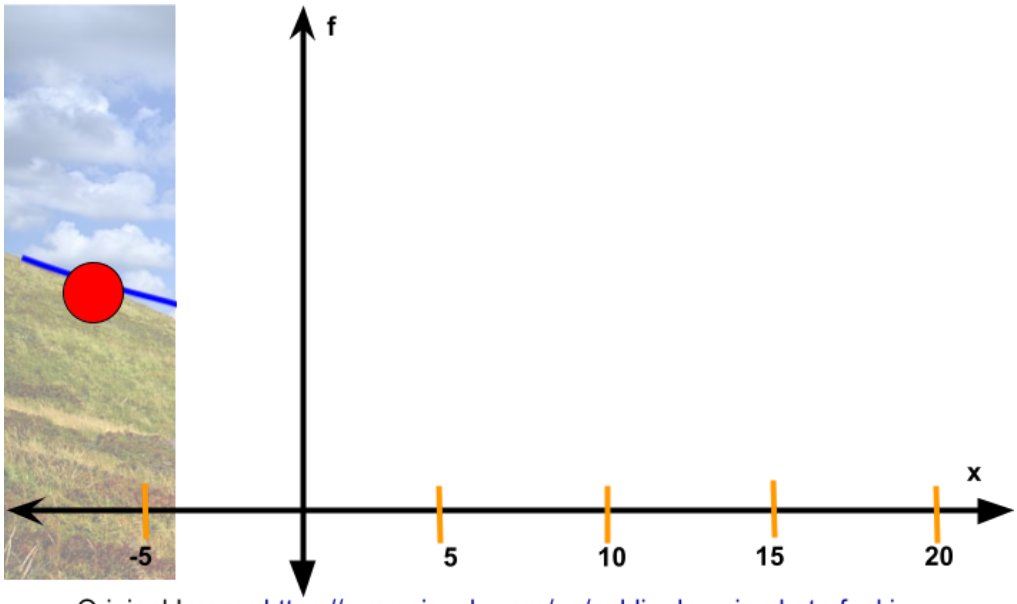
As you can see the weight value of 10 results in the lowest cost for the model. Since when the model's weight value reaches 10, the model is smart (determined from cost) and we are happy, there is a green happy-face at the point on the cost function where the weight value is 10. Note that even slightly increasing/decreasing this weight value will increase the cost.
~~~~~~

Now usually when we initialize a model, we may end up starting from the red-dot on the left or from the red-dot on the right. Mainly this is because it is really hard during initialization to randomly initialize at the perfect location (seen from the 1000 model initializations in the last unit). As a result, we can only make the model smarter by moving the red-dot through updating the weight values. Thus, our goal is to **descend** this hill from our starting position.

Using a graph to **descend** is similar to navigating to a location using a map since you can "see" all of your surroundings. You know where you are on the map based on the objects near you (on the graph, based on the current cost), and you know where you want to go (on the graph, based on the lowest possible cost). However in reality creating these graphs to figure out the best weight value for **one parameter** does not make sense. Mainly due to the fact that we need a graph of the cost function in relation to **each** parameter. When solving real-world problems, thousands of parameters are not enough! Currently (as the book is being written) the range is between millions to billions! The work of creating that many graphs is just making your computer a slave instead of making your computer smart.

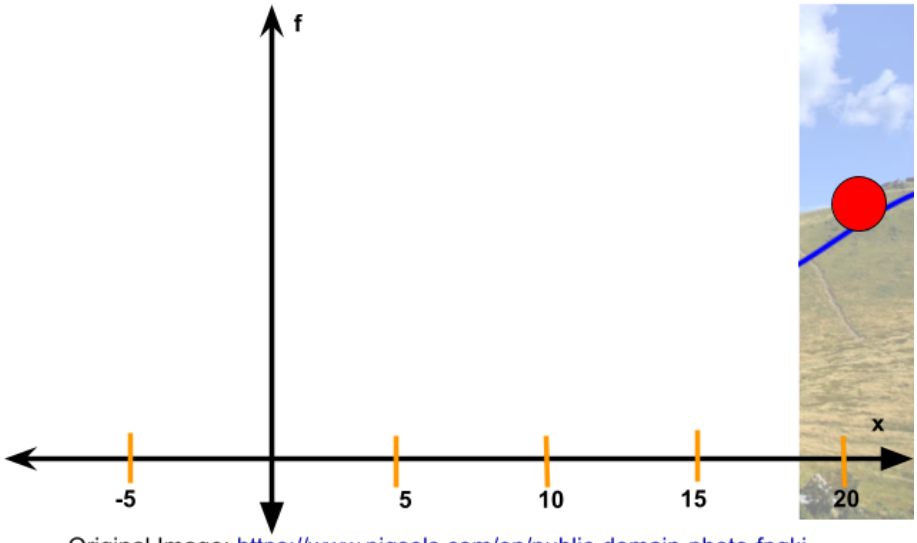
At the same time, thinking about multiple 3-dimensional graphs (1 dimension for each weight value, so the max is 2 weight values, with the last dimension being the cost) may help but creates the need for combinations of weight values (cost graph in relation to weight #1 and weight #2, cost graph in relation to weight #1 and weight #3, etc., or cost graph in relation to weight #2 and weight #3, cost graph in relation to weight #2 and weight #4, etc.) To fully solve the problem of the combinations is to have a graph that is much more than 3-dimensions (crazy!). One dimension for each parameter plus one more for the cost. The only limitation is the 3-dimension world we perceive to live in so graphs with more dimensions than the world we perceive to live wouldn't really make sense to have.

So if we cannot use a graph to **descend**, how do we descend? The answer is through **gradients**. **Gradient descent is all about descending the cost function using gradients.** Wait, what is a **gradient**? A **gradient** is a fraction of the **slope at a point**. In high school math, slope only exists in linear functions (lines) however looking at the graph of our cost function we can't calculate the slope because it is not linear. To understand how to calculate the **slope at a point** in a **nonlinear** graph, we need to imagine that the graph (hill-world) is covered in **thick** fog.



Original Image: https://www.piqsels.com/en/public-domain-photo-fpgkj

So thick, you cannot see much in front and behind yourself. Now your graph seems pretty useless as you have no idea where your destination is (lowest-cost). Your graph only shows the cost values of the nearest weight values. How do you descend? In reality, this is what your model sees, only a portion of the cost (hill) world. To reinforce this idea, here is the fog-covered graph from the other possible initialization spot:



Original Image: https://www.piqsels.com/en/public-domain-photo-fpgkj

With **slope in linear** graphs in mind, and the new graphs under the fogged world, take this chance to think about how we can **descend** from a point. The next page will hopefully validate and expand on your hypothesis.

Please note that this unit will focus on determining the slope at a point on the graph to help determine whether to increase/decrease the weight value (move left or right on the graph). The next unit will focus on determining **how much** to increase/decrease by using a fraction of the slope we determined in this unit.

As you can see from the new (partial-complete) graphs, the graph almost looks straight (linear). In essence, we zoomed into a part of the graph when we added the fog. In fact, zooming into any graph by a huge amount will make the graph look linear. At the same time, the intervals each tick-mark will be separated by will become smaller and smaller, approaching zero (0). Since only a **tiny** region of the graph is linear, we can only calculate the slope/average rate of change over a **tiny** interval. The **tinier** the interval gets, the more accurate our **slope at that point** is.

To make things easier, case #1 will be the model's initialization at the red-dot on the left while case #2 will be the model's initialization at the red-dot on the right.

In case #1 we can clearly see that the slope is **negative** because a slight **increase** in the weight value will **decrease** the cost and a slight **decrease** in the weight value will **increase** the cost. As a result we should **increase** the weight value when the slope at the point the model is on the cost function is **negative**.

On the other hand, in case #2 we can clearly see that the slope is **positive** because a slight **increase** in the weight value will **increase** the cost and a slight **decrease** in the weight value will **decrease** the cost. As a result we should **decrease** the weight value when the slope at the point the model is on the cost function is **positive**.

This means that when we add a fraction of the slope to update a parameter, we actually add a fraction of the **negative** of the slope. In the next section, we will begin to calculate these slopes.

Numerical Derivative

To calculate the **slope at a point**, we need a **tiny** interval. For example, we have the cost function, J(x) and our model is currently at (x, y) → (2, J(2)). To create a **tiny** interval we need a **tiny** value which we can call **epsilon** (abbreviation – "**eps**", symbol – **ε**, greek letter lowercase epsilon). For this example we will have **ε = 0.0001**. As you can see **ε** has to be a **tiny** value close to zero. Now our **tiny** interval will be [2, 2+ε] → [2, 2.0001]. As a result, the **slope at the point (x, J(x)) =** $\frac{J(x+\varepsilon)-J(x)}{(x+\varepsilon)-x} \rightarrow \frac{J(2.0001)-J(2)}{2.0001-2}$.

This equation should be familiar to the basic slope calculation from high school math: $\frac{y_2-y_1}{x_2-x_1}$ or $\frac{f(x_2)-f(x_1)}{x_2-x_1}$ where $x_2 = x + \varepsilon$ and $x_1 = x$ while y_2 or $f(x_2) = J(x_2)$ or $J(x + \varepsilon)$ and y_1 or $f(x_1) = J(x_1)$ or $J(x)$.

Now here is the code to calculate the cost and to make sure our update rule from the previous section works:

```
# 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)
og_cost = mse.forward(model.outputs, y)    # original cost
EPS = 1e-4    # epsilon, (1 * (10 ** -4))

print("Weights:", model.weights)
print("Biases:", model.biases)
print("Cost:", og_cost)
print("EPSILON:", EPS)
```

The only thing new should be the value set to epsilon (1e-4). 1**e**-4 is a shorthand method of saying 1 * **10** $^{-4}$. This is also equivalent to the 1 being placed in the 4th decimal place (0.0001). Next we calculate the slope:

```
# (x2, y2)
model.weights += EPS    # x2
x2 = float(model.weights)    # since there is only 1 weight value
model.forward(X)
cost2 = mse.forward(model.outputs, y)    # y2
model.weights -= EPS    # reset to original value

# (x1, y1)
model.weights -= EPS    # x1
x1 = float(model.weights)    # since there is only 1 weight value
model.forward(X)
cost1 = mse.forward(model.outputs, y)    # y2
model.weights += EPS    # reset to original value

slope = (cost2 - cost1) / (x2 - x1)
print(slope)
```

Remember that the x in J(x) is a weight value and since we only have 1 neuron and 1 input feature there is only 1 weight value in the weight matrix. We can add epsilon to only one variable/element at a time. Next we store that new value as x_2 and get a new prediction from the model. After that we input our new prediction to the cost function which will be y_2. Now we have the coordinates (x_2, y_2) and we need to reset the weight of the model to the original value. To get the coordinates (x_1, y_1), we do the same thing except we subtract epsilon and when resetting the weight to the original value, we add epsilon. Lastly, we calculate the slope using the slope equation and print it out.

To check that our rule from the last section works:

```
# increase/decrease
if slope < 0:
    print("slope is negative, we need to increase the weight to decrease the cost")
    model.weights += EPS
elif slope > 0:
    print("slope is positive, we need to decrease the weight to decrease the cost")
    model.weights -= EPS
else:
    print("oh no! :(")

# check if cost decreased
model.forward(X)
new_cost = mse.forward(model.outputs, y)
print(new_cost)
if new_cost < og_cost:
    print("Change:", new_cost-og_cost)
else:
    print("oh no! :(")
```

If the slope is negative (less than zero) we increase the weight by ε (epsilon) and if the slope is positive (greater than zero) we decrease the weight by ε (epsilon). To check that the cost decreases by our rule, we get a new prediction from the model, get the new cost from the new prediction and print it. Also if the cost did decrease we print out the change. Here is the output:

```
Weights: [[0.01764052]]
Biases: [[0.]]
Cost: 4255.854199207604
EPSILON: 0.0001
-612.5336296054267
slope is negative, we need to increase the weight to decrease the cost
4255.792946071311
Change: -0.06125313629308948
```

As you can see, the slope is about **-610**. Also, increasing the weight by epsilon, the cost did indeed decrease and by ~0.06. If you truly understand the concept of slope, increasing the weight by 1 will decrease the cost by **about 610** because slope tells how much the output changes when increasing the input by 1. Note that our cost function is still **nonlinear** but when we zoomed in by a **huge** amount so that the interval was **tiny**, we **perceive** that part of the cost function as **linear**, as a result, the cost will decrease by **about 610** when increasing the weight by 1. To validate this statement, change the amount we increase the weight in the first if-statement to 1 and check the new cost and the amount the cost changed.

Now we only calculated the slope at a point for one variable, the weight, how about the bias? Similar to the weight, on the hill graph of the cost function, the y-axis is the cost while the x-axis is now bias values instead of weight values. We zoom into the point the model is at on the cost function, calculate the slope at a point using a **tiny interval**, but this time we slightly increase/decrease the bias instead, by the same ε (epsilon) value. After slightly increasing/decreasing the bias, we get a new prediction from the model that is used to get the new cost. Lastly, we calculate the slope at the model's location by the same equation but with x being the bias instead of the x being the weight. Here is the code (the first two code blocks shouldn't be foriegn):

```
# 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)
og_cost = mse.forward(model.outputs, y)    # original cost
EPS = 1e-4    # epsilon, (1 * (10 ** -4))

print("Weights:", model.weights)
print("Biases:", model.biases)
print("Cost:", og_cost)
print("EPSILON:", EPS)
```

```
# (x2, y2) -- w
model.weights += EPS    # x2
x2 = float(model.weights)    # since there is only 1 weight value
model.forward(X)
cost2 = mse.forward(model.outputs, y)    # y2
model.weights -= EPS    # reset to original value

# (x1, y1) -- w
model.weights -= EPS    # x1
x1 = float(model.weights)    # since there is only 1 weight value
model.forward(X)
cost1 = mse.forward(model.outputs, y)    # y2
model.weights += EPS    # reset to original value

slope_w = (cost2 - cost1) / (x2 - x1)
print("Slope of w:", slope_w)

# increase/decrease -- w
if slope_w < 0:
    print("slope of w is negative, we need to increase the weight to decrease the cost")
    model.weights += EPS
elif slope_w > 0:
    print("slope of w is positive, we need to decrease the weight to decrease the cost")
    model.weights -= EPS
else:
    print("oh no! :(")
```

The next code block should be the same to the code block above but involving the bias instead of the weight.

```python
# (x2, y2) -- b
model.biases += EPS    # x2
x2 = float(model.biases)    # since there is only 1 bias value
model.forward(X)
cost2 = mse.forward(model.outputs, y)    # y2
model.biases -= EPS    # reset to original value

# (x1, y1) -- b
model.biases -= EPS    # x1
x1 = float(model.biases)    # since there is only 1 bias value
model.forward(X)
cost1 = mse.forward(model.outputs, y)    # y2
model.biases += EPS    # reset to original value

slope_b = (cost2 - cost1) / (x2 - x1)
print("Slope of b:", slope_b)

# increase/decrease -- b
if slope_b < 0:
    print("slope of b is negative, we need to increase the bias to decrease the cost")
    model.biases += EPS
elif slope_b > 0:
    print("slope of b is positive, we need to decrease the bias to decrease the cost")
    model.biases -= EPS
else:
    print("oh no! :(")
```

Lastly to check if our rule from the last section works (output right below):

```python
# check if cost decreased
model.forward(X)
new_cost = mse.forward(model.outputs, y)
print(new_cost)
if new_cost < og_cost:
    print("Change:", new_cost-og_cost)
else:
    print("oh no! :(")
```

```
Weights: [[0.01764052]]
Biases: [[0.]]
Cost: 4255.854199207604
EPSILON: 0.0001
Slope of w: -612.5336296054267
slope of w is negative, we need to increase the weight to decrease the cost
Slope of b: -119.85807581368135
slope of b is negative, we need to increase the bias to decrease the cost
4255.78096027373
Change: -0.07323893387456337
```

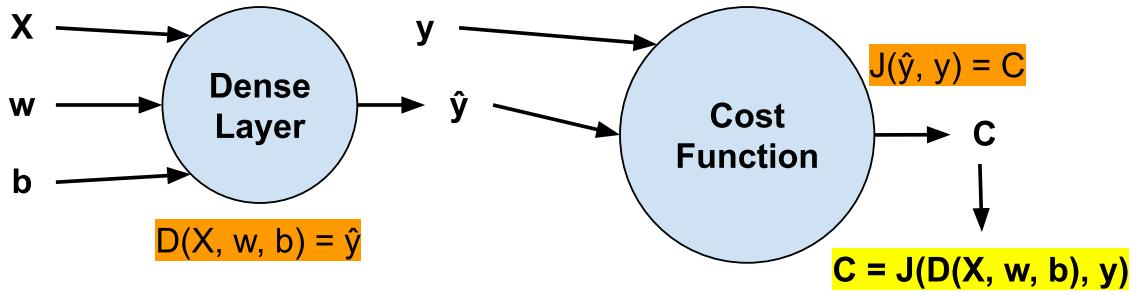
Our rule makes sense in relation to our data. From the last unit we established that the weight (slope of the model) should be 10 and the bias (y-intercept of the model) should be 20. As seen from printed out weight and bias from the output of the script, we do need to increase both the weight and bias. Increasing the weight and the bias by ε (epsilon) decreased the cost by ~0.07, better than ~0.06!

Lastly, if you truly understand the concept of slope, increasing the weight and bias by 1 will decrease the cost by **about** (**610+120=730**). To validate this statement, change the amount we increase the weight in the first if-statement to 1 and check the new cost and the amount the cost changed.

What we have just did – calculating the slope at a point while leaving all the variables constant except one (when calculating slope of w, no change in b, when calculating slope of b, no change in w) – is a **derivative** (in calculus), more specifically given that only one variable allowed to change is a **partial derivative**. Also since we calculated the **partial derivative** by the basic slope formula, we got the **numerical derivative**. If we have more weight values (by having more neurons and input features) and more bias values (by having more neurons), calculating the **partial derivatives numerically** is a lot of work. Luckily, we can get **partial derivatives analytically** which involves using properties of calculus. We will explore the **chain rule of differentiation** (calculus terminology) to find the **analytical derivative** in the next section.

Calculus Chain Rule for High Schoolers

To begin, let's check out a diagram of our current process to determine the cost of our model. For each function, we will define it mathematically:



The output of the dense layer/function, D(X, w, b) gives the output ŷ. The extra variables inside the function mean that the function takes in multiple variables to get the output. For the cost function, J(ŷ, y) gives the output C (for cost). This should be pretty straightforward. However, knowing that we get a composite function (a function that gets its input from outputs of other functions), where the cost function uses the output of D(X, w, b) to get its input of ŷ.

Now if we want the **slope of the weight on the cost function at a point**, imagine a graph that has the x-axis as the weight values and the y-axis as the cost values. If we want the **slope of the bias on the cost function at a point**, imagine a graph that has the x-axis as the bias values and the y-axis as the cost values. According to the chain rule of differentiation from calculus (∂ symbol, means partial derivative):

$\frac{\partial J(\hat{y}, y)}{\partial w} = \frac{\partial J(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w}$ and $\frac{\partial J(\hat{y}, y)}{\partial b} = \frac{\partial J(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial b}$

🤯🤯🤯🤯, this must be what is happening in your head right now. Don't worry, let's break calculus down into basic high-school math terminology.

Let's start with finding the slope of the weight on the cost function **at a point**, $\frac{\partial J(\hat{y}, y)}{\partial w}$ is called the **partial derivative** of the cost function w.r.t. (with respect to) the weight function. This is equal to the slope of the cost function **at the point** where **ŷ** is on the cost function multiplied by the slope of ŷ **at the point** where the **weight** is on ŷ. (graph with x-axis being the weight values and y-axis being the ŷ values). In the equation, $\frac{\partial J(\hat{y}, y)}{\partial w} = \frac{\partial J(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w}$, the **partial derivative** of the cost function w.r.t. the

weight is equal to the **partial derivative** of the cost function w.r.t. ŷ multiplied by the **partial derivative** of ŷ w.r.t. the weight.

On the other hand, the slope of the bias on the cost function **at a point**, $\frac{\partial J(\hat{y}, y)}{\partial b}$ is called the **partial derivative** of the cost function w.r.t. (with respect to) the bias. This is equal to the slope of the cost function **at the point** where ŷ is on the cost function multiplied by the slope of ŷ **at the point** where the bias is on ŷ (graph with x-axis being the bias values and y-axis being the ŷ values). In the equation, $\frac{\partial J(\hat{y}, y)}{\partial b} = \frac{\partial J(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial b}$, the **partial derivative** of the cost function w.r.t. the bias is equal to the **partial derivative** of the cost function w.r.t. ŷ multiplied by the **partial derivative** of ŷ w.r.t. the bias.

The reason why we add in ŷ, but not any other variable is because to get the cost, we need ŷ and y but y does not depend on w for whatever value it stores (shown in the diagram). Continuing, to get ŷ, we need X, w, and b, as a result, ŷ is an intermediate variable between w and the cost. The same reasoning is true for the bias equation however, instead of y not depending on w for its value, we do not use y because y does not depend on b for its value.
It is best to resort back to basic understanding of slope instead of using code to validate chain rule since code hides the mathematical operation being performed.

Slope can sometimes be referred to on a graph as "rise over run", where it is calculated by the **change in** y divided by the **change in** x. As a result, the equation for slope is either $\frac{\Delta y}{\Delta x}$ or $\frac{y_2 - y_1}{x_2 - x_1}$ where Δ is the greek letter, capital **delta**, meaning "**change in**".
When calculating the **slope at a point**, the **change in** the variable on the x-axis is **tiny** and to be more specific is **infinitesimal/infinitely small**, while keeping all the other variables except the variable on the y-axis constant, we get a **partial derivative**.
Knowing all this, let's translate this $\frac{\partial J(\hat{y}, y)}{\partial w}$, into an equation for slope. If C is the output of J(ŷ, y) then we get: $\frac{\Delta C}{\Delta w}$ or $\frac{C_2 - C_1}{w_2 - w_1}$. Basic elementary multiplication tells us that multiplying anything by 1 should not change its value. To be more specific, let us

multiply $\frac{\Delta C}{\Delta w}$ by $\frac{\Delta \hat{y}}{\Delta \hat{y}}$ which is the same as $\frac{\Delta C}{\Delta \hat{y}} * \frac{\Delta \hat{y}}{\Delta w}$ or $\frac{C_2 - C_1}{\hat{y}_2 - \hat{y}_1} * \frac{\hat{y}_2 - \hat{y}_1}{w_2 - w_1}$. That is the translated version of $\frac{\partial J(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w}$! At the same time we did not change the value of $\frac{\Delta C}{\Delta w}$ when multiplying by $\frac{\Delta \hat{y}}{\Delta \hat{y}}$. As a result, $\frac{\partial J(\hat{y}, y)}{\partial w} = \frac{\partial J(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w}$ is a true statement. Using the same logic and translations you can prove to yourself that $\frac{\partial J(\hat{y}, y)}{\partial b} = \frac{\partial J(\hat{y}, y)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial b}$ is also a true statement.

Now that we know how chain rule works, why do we need to know this? When we find the analytical derivative of the cost function w.r.t. ŷ and the analytical derivative of ŷ w.r.t. w and b in the last two sections of this unit, the simplification and purpose of this chain rule becomes crystal clear.

Cost Function Backward Method (Analytical Derivative)

Now the first step to getting the analytical derivative of the cost function w.r.t. the weight ($\frac{\partial J(\hat{y}, y)}{\partial w}$) or the analytical derivative of the cost function w.r.t. the cost bias ($\frac{\partial J(\hat{y}, y)}{\partial w}$) is to get the analytical derivative of cost function w.r.t. $\hat{y}$ ($\frac{\partial J(\hat{y}, y)}{\partial \hat{y}}$).
To do this, we are going to use the structure of the equation formulated in the numerical derivative section (**slope at the point (x, J(x)) =** $\frac{J(x+\varepsilon)-J(x-\varepsilon)}{(x+\varepsilon)-(x-\varepsilon)}$) and simplify **everything**! Let's start with MSE's analytical derivative:

$$\frac{J(\hat{y}+\epsilon, y)-J(\hat{y}-\epsilon, y)}{(\hat{y}+\epsilon)-(\hat{y}-\epsilon)} = \frac{[(\hat{y}+\epsilon)-y]^2-[(\hat{y}-\epsilon)-y]^2}{(\hat{y}+\epsilon)+(-\hat{y}+\epsilon)}$$

Substitute $[(\hat{y} + \varepsilon) - y]^2$ in place of $J(\hat{y} + \varepsilon, y)$.
Substitute $[(\hat{y} - \varepsilon) - y]^2$ in place of $J(\hat{y} - \varepsilon, y)$.
Distribute the negative in front of the denominator's second term (x_1): $\hat{y} + \varepsilon$.

~~~~~~

$$= \frac{[(\hat{y}+\epsilon)^2 - 2y(\hat{y}+\epsilon) + y^2] - [(\hat{y}-\epsilon)^2 - 2y(\hat{y}-\epsilon) + y^2]}{(\hat{y}-\hat{y}) + (\epsilon+\epsilon)}$$

Knowing that $(a + b)^2$ is equal to $a^2 + 2ab + c^2$, we expand the numerator's first term (y_2): $[(\hat{y} + \varepsilon) - y]^2$ by having $a = (\hat{y} + \varepsilon)$, $b = -y$, and expand the numerator's second term (y_1): $[(\hat{y} - \varepsilon) - y]^2$) by having $a = (\hat{y} - \varepsilon)$, $b = -y$. In the denominator we group like terms.

~~~~~~

$$= \frac{[\hat{y}^2 + 2\hat{y}\epsilon + \epsilon^2 - 2y(\hat{y}+\epsilon) + y^2] - [\hat{y}^2 - 2\hat{y}\epsilon + \epsilon^2 - 2y(\hat{y}-\epsilon) + y^2]}{2\epsilon}$$

Expand $(\hat{y} + \varepsilon)$ by the same rule $(a + b)^2 = a^2 + 2ab + c^2$ where $a = \hat{y}$ and $b = \varepsilon$.
Expand $(\hat{y} - \varepsilon)$ by the same rule $(a + b)^2 = a^2 + 2ab + c^2$ where $a = \hat{y}$ and $b = -\varepsilon$.
Lastly we simplify the denominator by combining the grouped like terms.

$$= \frac{[\hat{y}^2 + 2\hat{y}\epsilon + \epsilon^2 - 2y(\hat{y}+\epsilon) + y^2] + [-\hat{y}^2 + 2\hat{y}\epsilon - \epsilon^2 + 2y(\hat{y}-\epsilon) - y^2]}{2\epsilon}$$

Distribute the negative in front of the numerator's second term (y_1):
$[\hat{y}^2 - 2\hat{y}\varepsilon + \varepsilon^2 - 2y(\hat{y} - \varepsilon) + y^2]$.

~~~~~~

$$= \frac{(\hat{y}^2 - \hat{y}^2) + [2\hat{y}\epsilon + 2\hat{y}\epsilon] + [\epsilon^2 - \epsilon^2] + [-2y(\hat{y} + \epsilon) + 2y(\hat{y} + \epsilon)] + [y^2 - y^2]}{2\epsilon}$$

In the numerator, we group like terms together. The only terms in the numerator that aren't "like terms" are the terms that are not colored since they need to be simplified.

~~~~~~

$$= \frac{4\hat{y}\epsilon + [-2y(\hat{y} + \epsilon) + 2y(\hat{y} - \epsilon)]}{2\epsilon} = \frac{4\hat{y}\epsilon + (-2y\hat{y} - 2y\epsilon) + (2y\hat{y} - 2y\epsilon)}{2\epsilon}$$

Simplify the numerator by combining the grouped like terms. Next, we simplify the first term inside the brackets by distributing/multiplying -2y to ŷ and ε. We simplify the second term inside the brackets by distributing/multiplying 2y to ŷ and -ε.

~~~~~~

$$= \frac{4\hat{y}\epsilon + (-2y\hat{y} + 2y\hat{y}) + (-2y\epsilon - 2y\epsilon)}{2\epsilon} = \frac{4\hat{y}\epsilon - 4y\epsilon}{2\epsilon} = \frac{4(\hat{y}\epsilon - y\epsilon)}{2\epsilon}$$

After that, we group the like terms which will be simplified by being combined.

Also, factor out the common factor (4) from both terms in the numerator.

~~~~~~

$$= \frac{2(\hat{y}\epsilon - y\epsilon)}{\epsilon} = \frac{2\epsilon(\hat{y} - y)}{\epsilon} = 2(\hat{y} - y)$$

Divide 4 (from numerator) by 2 (from denominator). Factor out the common ε from both terms in the numerator. Divide ε (from numerator) by ε (from denominator) to finally, **finally**, get the analytical derivative of MSE: 2(ŷ - y). Here is a PDF of steps that got us the analytical derivative.

Note that 2(ŷ - y) is the analytical derivative for one example's one output feature. For multiple examples with multiple output features we remember what we did to MSE for multiple examples with multiple output features. We took the **average/mean** of each example's **average/mean** cost of the output features to get a scalar from a matrix with shape [n, f] (**n** is the number of examples, **f** is the number of output features). Taking the average/mean consisted of taking all the elements of the matrix and adding/summing them then dividing by the total number of elements (n*f).

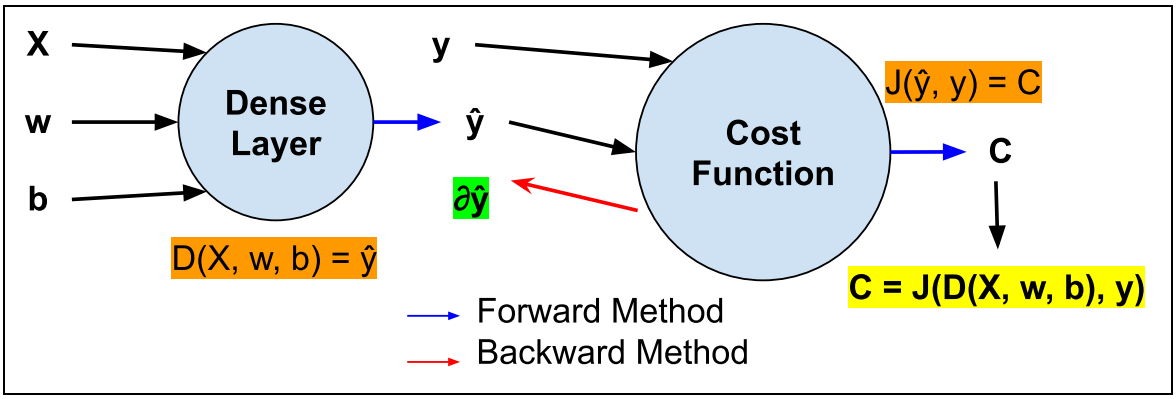
If you think about having a graph of the cost function for each example's output features values on the x-axis and all with the cost values on the y-axis, the result of the analytical derivative should be a matrix of shape [n, f]. This is because each example's output features will have a different partial derivative (slope at a point) in respect to the cost function. Therefore we cannot add/sum all the values of the resulting matrix. At the same time, it is important to realize that partial derivatives of any function, f(x), w.r.t. a variable x will have the same shape of x.

What we can do is divide each element in the matrix by the total number of elements (n*f). A good explanation for doing this may be by looking at the cost function for only one example:

$\frac{|\hat{y}_1 - y_1| + |\hat{y}_2 - y_2| + .. + |\hat{y}_j - y_j|}{j}$, where j is the number of output features

$= \frac{1}{j}|\hat{y}_1 - y_1| + \frac{1}{j}|\hat{y}_2 - y_2| + . . + \frac{1}{j}|\hat{y}_j - y_j|$ In reality, what we are doing in the one example cost function, is taking a **fraction** of the error/cost from each output feature. Even though in the actual cost function we are taking the **average/mean** of the cost from each example's **average/mean** of the output features, we are just multiplying (1/n) to each of the terms. As a result, since the cost function is calculated by a **fraction** of each element's cost, it only makes sense to also use a **fraction** of each partial derivative/slope. Thus, the analytical derivative of MSE with **n** examples and **f** output features: $\frac{2}{(n*f)} * (\hat{y} - y)$.

Getting the **cost** in the cost function **from** $\hat{y}$, is called the forward method of the cost function since the arrows point forward **to** the cost function.
Getting the **partial derivative** of the cost function **w.r.t.** $\hat{y}$, is called the backward method of the cost function since the arrows point backward **from** the cost function.



Now let's implement the backward method of the cost function. Here is the code:

```
class MSE_Cost:
    def forward(self, y_pred, y_true):
        return np.mean(np.square(y_pred - y_true))

    def backward(self, y_pred, y_true):
        self.dinputs = (2 / y_pred.size) * (y_pred - y_true)
```

The only new part is the .size attribute of y_pred. The .size attribute of a numpy array returns the number of elements in the matrix. So if the matrix has shape [n, f] the .size attribute will return the product of n and f. Also, self.dinputs is the partial ***d*erivative** of the cost w.r.t. to the inputs (either ŷ or y, in this case ŷ because y is an independent variable in relation to the model).

~~~~~~

Now what about the partial derivative of MAE w.r.t. ŷ? The best way to keep things at a high-schooler level is to define MAE as a piecewise function because absolute values can be tricky.

First, the only reason we need an absolute value function in MAE is because when ŷ < y, the error (ŷ - y) would be negative and that does not make sense. As a result, we need to turn the result of (ŷ - y) when ŷ < y, positive. Turning a negative value positive is the same as multiplying the negative value by -1: -2*-1=2. As a result this can be our piecewise function of MAE:
~~~~~~

$$MAE = \begin{cases} -(\hat{y} - y), & \text{if } \hat{y} < y \\ \hat{y} - y, & \text{if } \hat{y} > y \\ 0, & \text{if } \hat{y} = y \end{cases}$$

Note that these are only for one example with one output feature. If ŷ is **exactly** equal to y that means that the model predicted that example's output feature 100% correctly and therefore should not have any error.

Now let's get the analytical derivative of MAE, piece-by-piece. Starting with **ŷ < y**:

$$\frac{J(\hat{y}+\epsilon, y) - J(\hat{y}-\epsilon, y)}{(\hat{y}+\epsilon) - (\hat{y}-\epsilon)} = \frac{-[(\hat{y}+\epsilon) - y] - [-[(\hat{y}-\epsilon) - y]]}{(\hat{y}+\epsilon) + (-\hat{y}+\epsilon)}$$

Substitute -[(ŷ + ε) - y] in place of J(ŷ + ε, y). Substitute -[(ŷ - ε) - y] in place of J(ŷ - ε, y). Distribute the negative in front of the denominator's second term (x_1): ŷ + ε.

~~~~~~

$$= \frac{[-(\hat{y}+\epsilon) + y] + [(\hat{y}-\epsilon) - y]}{(\hat{y}-\hat{y}) + (\epsilon+\epsilon)} = \frac{(-\hat{y}-\epsilon+y) + (\hat{y}-\epsilon-y)}{2\epsilon}$$

Distribute the negative for the numerator's first term. Double negative for the numerator's second term creates a positive. In the denominator we group like terms. Next, we distribute the negative in front of (ŷ + ε) in the numerator's first term and simplify the denominator by combining the grouped like terms.

~~~~~~

$$= \frac{(-\hat{y}+\hat{y}) + (-\epsilon-\epsilon) + (y-y)}{2\epsilon} = \frac{-2\epsilon}{2\epsilon} = -1$$

After that, we group the like terms which will be simplified by being combined.

Lastly, divide -2ε (from numerator) by 2ε (from denominator) to get -1.
Now when **ŷ > y**:

$$\frac{J(\hat{y}+\epsilon, y) - J(\hat{y}-\epsilon, y)}{(\hat{y}+\epsilon) - (\hat{y}-\epsilon)} = \frac{[(\hat{y}+\epsilon) - y] - [(\hat{y}-\epsilon) - y]}{(\hat{y}+\epsilon) + (-\hat{y}+\epsilon)}$$

Substitute [(ŷ + ε) - y] in place of J(ŷ+ε,y). Substitute [(ŷ - ε) - y] in place of J(ŷ-ε,y).

Next, distribute the negative in front of the denominator's second term (x_1): ŷ + ε.

$$= \frac{[(\hat{y} + \epsilon) - y] + [-(\hat{y} - \epsilon) + y]}{(\hat{y} - \hat{y}) + (\epsilon + \epsilon)} = \frac{(\hat{y} + \epsilon - y) + (-\hat{y} + \epsilon + y)}{2\epsilon}$$

Distribute the negative in front of the numerator's second term (y_1): [(ŷ - ε) - y]. In the denominator we group like terms. Next, we distribute the negative in front of (ŷ - ε) in the numerator's second term and simplify the denominator by combining the grouped like terms.

~~~~~~

$$= \frac{(\hat{y} - \hat{y}) + (\epsilon + \epsilon) + (-y + y)}{2\epsilon} = \frac{2\epsilon}{2\epsilon} = 1$$

After that, we group the like terms which will be simplified by being combined.

Lastly, divide 2ε (from numerator) by 2ε (from denominator) to get 1.

~~~~~~

When **ŷ = y**:

$$\frac{J(\hat{y} + \epsilon, y) - J(\hat{y} - \epsilon, y)}{(\hat{y} + \epsilon) - (\hat{y} - \epsilon)} = \frac{0 - 0}{(\hat{y} + \epsilon) + (-\hat{y} + \epsilon)}$$

Substitute 0 in place of both J(ŷ + ε, y) and J(ŷ - ε, y). Distribute the negative in front of the denominator's second term (x_1): ŷ + ε.

~~~~~~

$$= \frac{0}{(\hat{y} - \hat{y}) + (\epsilon + \epsilon)} = \frac{0}{2\epsilon} = \frac{0}{2(0)} = \frac{0}{0} = \text{undefined!}$$

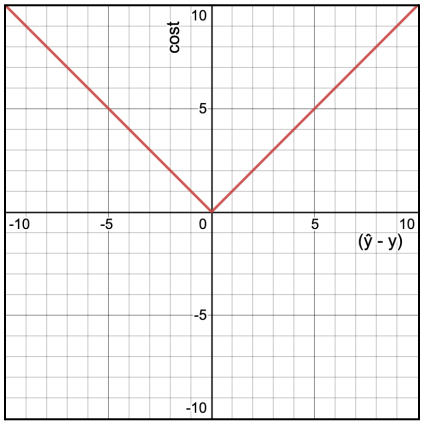
After that, we group the like terms which will be simplified by being combined. Now to continue, we can only substitute ε = 0 since epsilon has to be an infinitesimal/infinitely small value close to zero. Simplify the denominator and we see that **in calculus,** the analytical derivative of MAE **does not exist** (DNE) when ŷ=y since it is undefined. However, **in the context of AI**, the analytical derivative of MAE **does exist** when ŷ=y. We can break the rules of calculus when we can/need to!
~~~~~~

To figure out what the analytical derivative of MAE should be when ŷ=y, we need to know what it means to the cost when ŷ=y, and recall our update rule. When ŷ=y, the cost is zero because the predicted value is exactly the same as the correct value, as a result we should not penalize the model when the model has no error. Since we are getting the partial derivative of MAE w.r.t. ŷ, we are getting the **slope** of the cost function at the point where ŷ is on the cost function. Meaning that we can **increase/decrease** ŷ to lower the cost by using the **slope**. A **positive** slope indicates that **decreasing** ŷ will lower the cost and a **negative** slope indicates that **increasing** ŷ will lower the cost.

When ŷ=y on the cost function, the cost is 0 and to lower the cost would mean to get a negative cost/error which does not make sense. As a result, the slope **cannot be positive or negative**! The only real number that is neither positive nor negative is **zero**. We can summarize our findings for MAE as follows:

$$\frac{\partial MAE}{\partial \hat{y}} = \begin{cases} -1, & \text{if } \hat{y} < y \\ 1, & \text{if } \hat{y} > y \\ 0, & \text{if } \hat{y} = y \end{cases}$$

Here is a PDF of steps that got us the analytical derivative. Before we define the partial derivative of MAE w.r.t. ŷ that has multiple examples and output features, we are going to find out why when ŷ < y and when ŷ > y, the analytical derivative is so simple. To start off, here is a graph of the absolute value function:



As you can see the graph of the cost function when ŷ < y is on the left of the y-axis and the graph of the cost function when ŷ > y is on the right of the y-axis. By only focusing on the graph when ŷ < y, the graph is linear with a slope of -1! By only focusing on the graph when ŷ > y, the graph is linear with a slope of 1! As a result, the absolute value function is just a piecewise function composed of two linear functions!

Now by the same logic from the partial derivative of MSE w.r.t. ŷ with multiple examples and output features, we divide all the elements of the partial derivative of MAE w.r.t. ŷ with only one example and output feature by the product of the **n** (number of examples) and **f** (output features). Here is the code:

```python
class MAE_Cost:
    def forward(self, y_pred, y_true):    # y_pred is prediction from model, y_true is the answer
        return np.mean(np.abs(y_pred - y_true))

    def backward(self, y_pred, y_true):
        self.dinputs = np.sign(y_pred - y_true) / y_pred.size
```

The only new function is *np.sign()* which takes in a numpy array and outputs a -1 for negative elements, 0 for elements that equal 0, and 1 for positive elements.

Nice job on finishing the analytical derivative of MSE and MAE. We calculated those analytical derivatives from scratch, without much calculus but with a lot of algebra! To complete the chain rule, we create a backward method for the dense layer in the next section. Trust me, the algebra to get the analytical derivative of the dense layer (ŷ) w.r.t. the weight and the bias is much easier than the cost functions.

Dense Layer Backward Method (Analytical Derivative)

Let's start with the analytical derivative of the dense layer (ŷ) w.r.t. the weight:

$$\frac{\partial D(X,w,b)}{\partial w} = \frac{D(X,w+\epsilon,b) - D(X,w-\epsilon,b)}{(w+\epsilon)-(w-\epsilon)} = \frac{[X(w+\epsilon)+b] - [X(w-\epsilon)+b]}{(w+\epsilon)+(-w+\epsilon)}$$

Substitute [X(w + ε) + b] in place of D(X, w + ε, b). Substitute [X(w) + b] in place of D(X, w, b). Distribute the negative in front of the denominator's second term (x_1).

~~~~~~

$$= \frac{[X(w+\epsilon)+b] + [-X(w-\epsilon)-b]}{(w-w)+(\epsilon+\epsilon)} = \frac{[Xw+X\epsilon+b] + [-Xw+X\epsilon-b]}{2\epsilon}$$

After distributing the negative in front of the numerator's second term (y_1): [X(w)+b] and grouping the like terms in the denominator, distribute X to (w + ε) for the numerator's first term (y_2) and distribute -X to w for the numerator's first term (y_1). Next we simplify the denominator by combining the grouped like terms.

$$= \frac{(Xw-Xw)+(X\epsilon+X\epsilon)+(b-b)}{2\epsilon} = \frac{2X\epsilon}{2\epsilon} = \frac{X\epsilon}{\epsilon} = X$$

Now we can group the like terms in the numerator which can be simplified by being combined. To continue simplifying, we divide 2 (from numerator) by 2 (from denominator) and divide ε (from numerator) by ε (from denominator) to get **X**.

~~~~~~

Now the analytical derivative of the dense layer (ŷ) w.r.t. the bias:

$$\frac{\partial D(X,w,b)}{\partial b} = \frac{D(X,w,b+\epsilon) - D(X,w,b-\epsilon)}{(b+\epsilon)-(b-\epsilon)} = \frac{[Xw+(b+\epsilon)] - [Xw+(b-\epsilon)]}{(b+\epsilon)+(-b+\epsilon)}$$

Substitute [Xw + (b + ε)] in place of D(X, w, b + ε). Substitute [Xw + (b - ε)] in place of D(X, w, b - ε). Distribute the negative in front of the denominator's second term (x_1).

~~~~~~

$$= \frac{[Xw+(b+\epsilon)] + [-Xw-(b-\epsilon)]}{(b-b)+(\epsilon+\epsilon)} = \frac{[Xw+b+\epsilon] + [-Xw-b+\epsilon]}{2\epsilon}$$
~~~~~~

After distributing the negative in front of the numerator's second term (y_1): [Xw+(b - ε)] and grouping the like terms in the denominator, we distribute the negative in front of (b-ε) from the numerator's second term (y_1) and simplify the denominator by combining the grouped like terms.

~~~~~~

$$= \frac{(Xw - Xw) + (b - b) + (\epsilon + \epsilon)}{2\epsilon} = \frac{2\epsilon}{2\epsilon} = \frac{\epsilon}{\epsilon} = 1$$

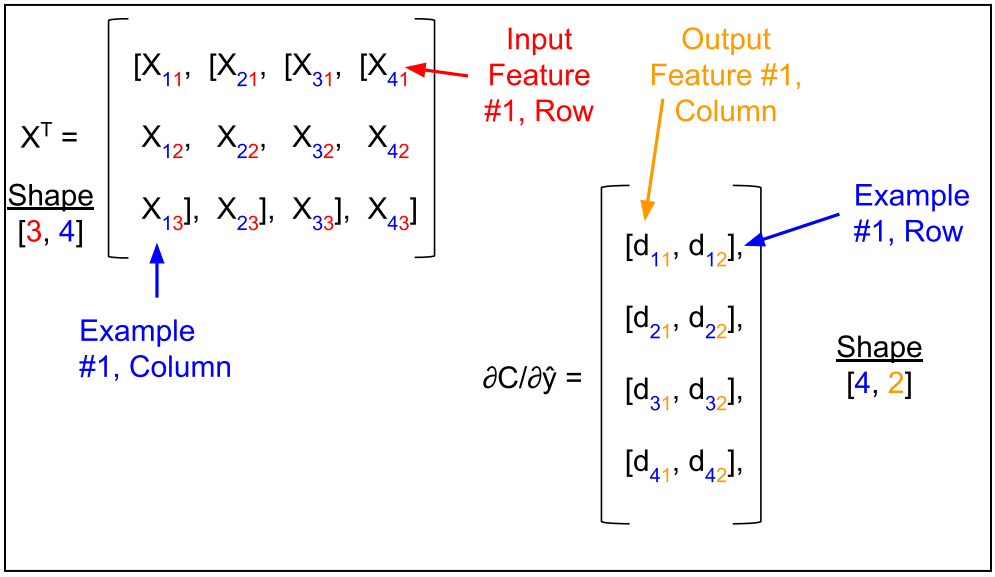
Now we can group the like terms in the numerator which can be simplified by being combined. To continue simplifying, we divide 2 (from numerator) by 2 (from denominator) and divide ε (from numerator) by ε (from denominator) to get **1**.

Now let's understand why the analytical derivatives are so simple. The dense layer itself is a linear equation (y = mx + b), ŷ = Xw + b. The inputs (X) are scaled up/down by the weight (w) and then added to the bias (b). Now remember in the y = mx + b equation, m is the slope, **when** X is plotted on the x-axis and y is plotted on the y-axis. To **view X as the slope (m)** we need to view w as the input (X) **when** w is plotted on the x-axis and ŷ is plotted on the y-axis. How about the bias? Since the equation to get ŷ connects the bias through the **addition** operator, increasing the bias by a number **n**, means that the output increases by the same amount, **n**. Since the **change in ŷ** (Δŷ) is the same as the **change in the bias** (Δb), the slope is always a constant, 1. Understand that this result is mainly due to the nature of the **addition** operator.

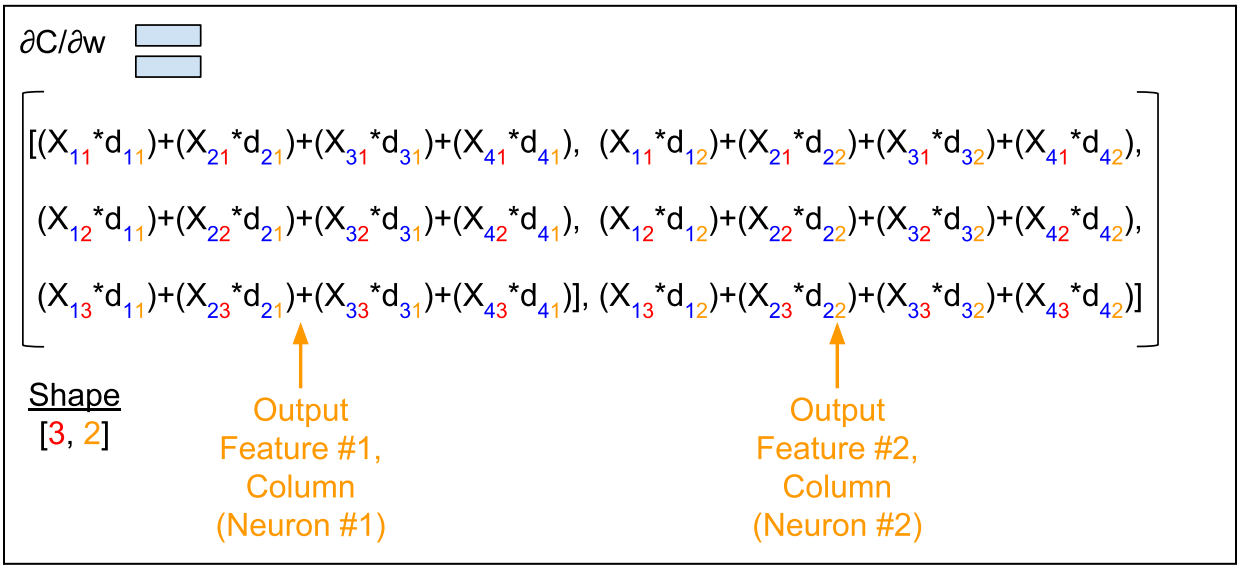
Simple? If everything was simple, anyone could create an AI, the hard part begins now. How do we get the correct shapes? Here is what I mean:
X – [**n**, **if**], w – [**if**, **of**], b – [1, **of**], ∂C/∂ŷ – [**n**, **of**], where ***n*** is the number of examples, ***if*** is the number of input features, ***of*** is the number of output features. How do we multiply ∂C/∂ŷ by X so that the resulting shape (∂C/∂w) has the same shape as w and how do we convert ∂C/∂ŷ so that the resulting shape (∂C/∂b) has the same shape as b?

Since ∂C/∂ŷ and X are both matrices with number of rows the same, we could perform a dot product. Remember that the result of a dot product with the first matrix having shape [a, b] and the second matrix having shape [b, c] will be a matrix with shape [a, c]. As a result our first matrix will have to have one of its dimensions equal to *if*. Clearly, that will be X, however we also need the number of rows equal to *if*, as a result, we need to transpose X to get the shape [if, n]. Thus, ∂C/∂w is equal to the dot product of
~~~~~~

X^T [if, n] with $\partial C/\partial \hat{y}$ [n, of]. Before moving on to get the bias, let's understand what we are actually doing in the dot product.



This is what we have if we have a model that takes in 3 input features and 2 output features. Also, let's say we have 4 examples. Each element in each matrix is indexed by **subscripts** where the first number indicates the row and the second number indicates the column. Note that in X^T our indices are based on the format of the **original matrix** (X). To make notation easier, in the $\partial C/\partial \hat{y}$ matrix we use indices of a matrix "d", so assume that the $\partial C/\partial \hat{y}$ matrix is the **same exact** as the "d" matrix.



Now there is a lot to unpack but understand that each neuron's $\partial C/\partial w$ which is a column vector has 3 elements (one for each **input feature**). Let's first understand the building block, for example: $(X_{11}*d_{11})$, we multiply the first example's 1st input feature

by the derivative of the cost function w.r.t. the first example's 1st output feature to get the derivative of the cost function on the first example w.r.t. the 1st neuron's 1st weight. As a result, the derivative of the cost function w.r.t. the 1st neuron's 1st weight, $(\partial C/\partial w)_{11}$, is the **sum** of all the derivatives of the cost function on each **example** w.r.t. the 1st neuron's 1st weight. The same rule applies to other weight values.

To understand why we add, we need to understand that the derivative of the cost function w.r.t. some variable **x** is the same as the **effect on the cost** when **slightly increasing x** (by definition of the analytical derivative). When we **slightly increase** the 1st neuron's 1st weight, the output of **each example's** 1st output feature will change, and as a result, will **effect the cost**. As seen from the numerical derivative section, when two variables change (say *f* and *g*), the **effect on the cost** is about the sum of the **effect on the cost** when slightly increasing *f* (leave other independent variables constant, ∂C/∂f) with the **effect on the cost** when slightly increasing *g* (leave other independent variables constant, ∂C/∂g). This is my interpretation of the sum, so keep your eyes open if you can find a better interpretation that is more intuitive.

This should be enough to implement the backward method to get ∂C/∂w but how about ∂C/∂b? Here is a recap: b – [1, **of**], ∂C/∂ŷ – [**n**, **of**], where ***n*** is the number of examples, ***of*** is the number of output features. Also, ∂ŷ/∂b = 1. Obviously, we need to make the examples dimension of ∂C/∂ŷ disappear. Here is how we can do that:

$$\partial\hat{y}/\partial b = \left[\left[(d_{11}+d_{21}+d_{31}+d_{41}), (d_{12}+d_{22}+d_{32}+d_{32}) \right] \right]$$

Shape
[1, 2]

Output Feature #1, Column (Neuron #1)

Output Feature #2, Column (Neuron #2)

The derivative of the cost function w.r.t. the 1st neuron's bias, $(\partial C/\partial b)_1$, is the **sum** of all the derivatives of the cost function on each **example** w.r.t. the 2st neuron's bias. Similarity, the derivative of the cost function w.r.t. the 2nd neuron's bias, $(\partial C/\partial b)_2$, is the **sum** of all the derivatives of the cost function on each **example** w.r.t. the 2nd neuron's bias.

The reason why we add is similar to the logic when adding to find ∂C/∂w. When **slightly increasing** the 1st neuron's bias, the output of **each example's** 1st output

feature will change, and as a result, will **effect the cost**. When multiple variables (**each example's** 1st output feature) change, the **effect on the cost** is about the sum of each **variable's effect on the cost**. Now that we understand how to get ∂C/∂w and ∂C/∂b, here is the code:

```
class Dense_Layer:
    def __init__(self, n_inputs, n_neurons):
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros([1, n_neurons])

    def forward(self, inputs):    # inputs is X
        self.inputs = inputs
        self.outputs = np.dot(inputs, self.weights) + self.biases

    def backward(self, dvalues):    # dvalues is .dinputs from cost function
        self.dweights = np.dot(self.inputs.T, dvalues)
        self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
```

We updated the forward method to save the inputs we receive so that we do not need to pass the inputs again in the backward method. The argument *dvalues* is the partial ***d*erivative** of the cost function w.r.t. to ŷ. Just to make things simpler we just call them *dvalues*. Now when we get *self.dbiases*, we use the *np.sum* function which takes in a numpy array and sums the values along **axis**. Axis 0 is the row axis/dimension of the array/matrix, and axis 1 is the column axis/dimension of the array/matrix. Since the first integer in python is 0 and when we index the format is (row, column), row is first (axis 0) then column (axis 1).

When we sum along the **row** axis, it means that the **row** dimension will disappear. Remember that b – [1, **of**], ∂C/∂ŷ – [**n**, **of**], where ***n*** is the number of examples, ***of*** is the number of output features, we **do need** the **row** dimension to disappear. Each column's value will be the sum of all the elements of that **column** (elements of a **column** go by **rows**). There is a catch to numpy's "disappear" meaning, the result with **keepdims**=False is *self.dbiases* becoming a vector from a matrix and we do not want that since *self.biases* is a matrix. As a result, we need to **keep** the **row** dimension but we set the shape of the row dimension to 1.

Now we are complete! Here is the code to validate chain rule:

```
# 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)
cost = mse.forward(model.outputs, y)

mse.backward(model.outputs, y)
model.backward(mse.dinputs)
print("Slope of w:", model.dweights)
print("Slope of b:", model.dbiases)
```

Slope of w: [[-612.5336296]]
Slope of b: [[-119.85887581]]

These are the exact same values from the numerical derivative! The reason why we are doing this? The numerical derivative uses a lot of computing power and the code to keep all independent variables related to the model while slightly increasing/decreasing one to get the partial derivative is messy and a lot of work. How about the reason for chain rule? Why not just one backward method to get the analytical derivative? If there was no chain rule:

$$\frac{\partial C}{\partial w} = \frac{J(D(X, w+\varepsilon, b), y) - J(D(X, w-\varepsilon, b), y)}{(w+\varepsilon)-(w-\varepsilon)}, \quad \frac{\partial C}{\partial b} = \frac{J(D(X, w, b+\varepsilon), y) - J(D(X, w, b-\varepsilon), y)}{(b+\varepsilon)-(b-\varepsilon)}$$

Imagine substituting the function D in place of ŷ to make the algebra even messier! Thanks to the chain rule, we can break things down into one step at a time.