Chapter 02| Step #1 of Training: Defining Error

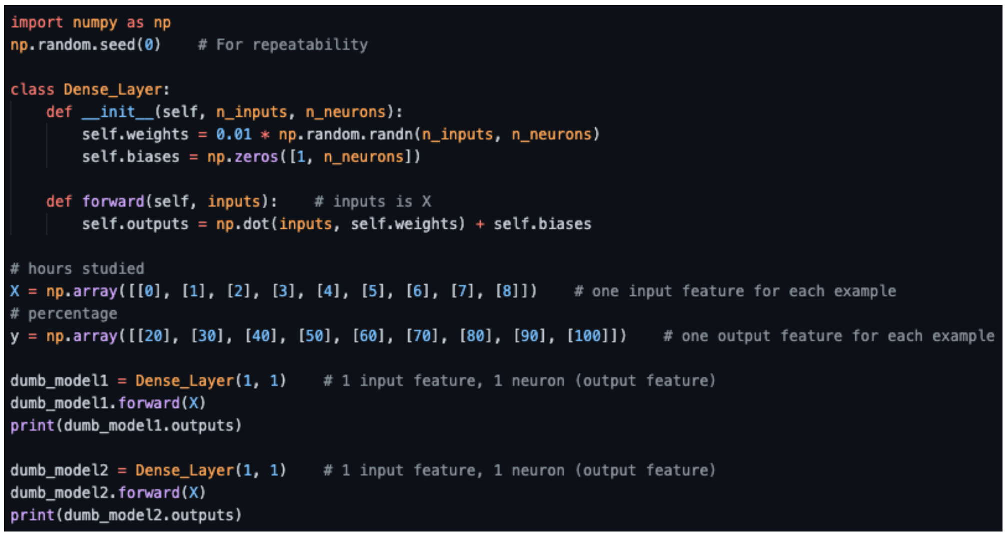
~~~~~~

Why do we need to Define a Metric for Error?

To understand this, we need to simplify our model to only have one neuron and receive only one input feature. We want the model to predict your test score based on how long you spent studying. Therefore, the input feature is the hours spent studying. Let's say the test was multiple choice (5 choices) so if you didn't study you would get 20% (random guessing), each extra hour you study, your score goes up by 10%. So studying for 1 hour gets you 30%, 2 hours is 40%, and so on. To get a 100% on your test, you need to study for 8 hours. As a result, here is your dataset:

Hours Studied	0	1	2	3	4	5	6	7	8
Percentage	20	30	40	50	60	70	80	90	100

Let's convert this into some code:



```
import numpy as np
np.random.seed(0)    # For repeatability

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.outputs = np.dot(inputs, self.weights) + self.biases

# 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

dumb_model1 = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
dumb_model1.forward(X)
print(dumb_model1.outputs)

dumb_model2 = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
dumb_model2.forward(X)
print(dumb_model2.outputs)
```

Here is dumb model 1's prediction:

```
[[0.        ]
 [0.01764052]
 [0.03528105]
 [0.05292157]
 [0.07056209]
 [0.08820262]
 [0.10584314]
 [0.12348366]
 [0.14112419]]
```

Here is dmb model 2's prediction:

```
[[0.        ]
 [0.00400157]
 [0.00800314]
```

We can easily see that dumb model 1 is actually smarter than dumb model 2 by a little bit because of the random initialization of the parameters of the layer.
~~~~~~

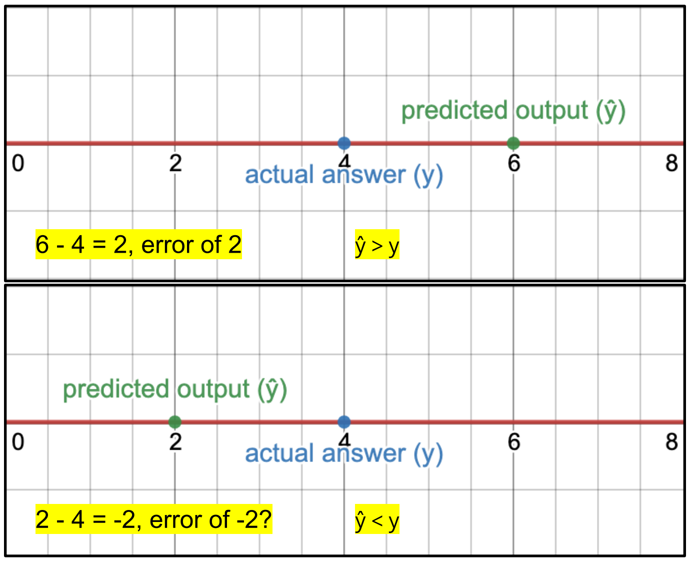
```
[0.01200472]
[0.01600629]
[0.02000786]
[0.02400943]
[0.028011  ]
[0.03201258]].
```

Now the question is, how to we **"tell"** dumb model 1 that it is smarter than dumb model 2 so that dumb model 1 **"knows"** that is a better model than dumb model 2.

The answer is to define a metric to calculate **how far away each example the model is to the correct output (y)**. This metric can be thought of as the **error** of the model, more specifically the **cost** of the model. Also there are many ways we can calculate cost and each type has its own pros and limitations. In the next section we are going to take a look at the two most common **cost functions** (function that calculates cost of a model) in **regression** (predicting values).

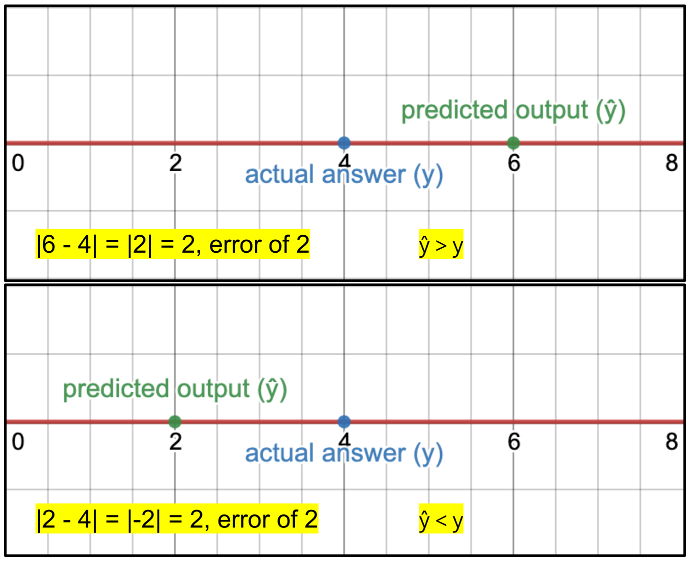
Metrics for Error (Cost Functions)

There are two common cost functions used in regression, **M**ean **A**bsolute **E**rror (MAE) and **M**ean **S**quared **E**rror (MSE).



Let's start with MAE and break down the acronym. First let's talk about **absolute error**, one way to measure error is to simply subtract the predicted output and actual answer ($\hat{y} - y$). The problem with this is that if the predicted output is greater than the actual answer ($\hat{y} > y$) the error is positive which makes sense. However if the output is less than the actual answer ($\hat{y} < y$) the error is negative which doesn't make sense. Negative error?

In the previous figure, both predicted outputs are 2 units away from the actual answer but subtracting the predicted output and actual answer give different outputs. One of the (positive) error values makes sense while the other (negative) error values does not.



To fix this problem we add an absolute value function to our calculation.

As a result, we have: |ŷ - y|. The figure on the left shows that it works!

What we have just defined is the absolute error of MAE: |ŷ - y|. However works for only one example with one output feature. To calculate the cost of one example, with **n** output features, we take the average of each output features cost:

$$\frac{|\hat{y}_1 - y_1| + |\hat{y}_2 - y_2| + \,..+ |\hat{y}_n - y_n|}{n} = \frac{1}{n}\sum_{i=0}^{n} |\hat{y}_i - y_i|$$

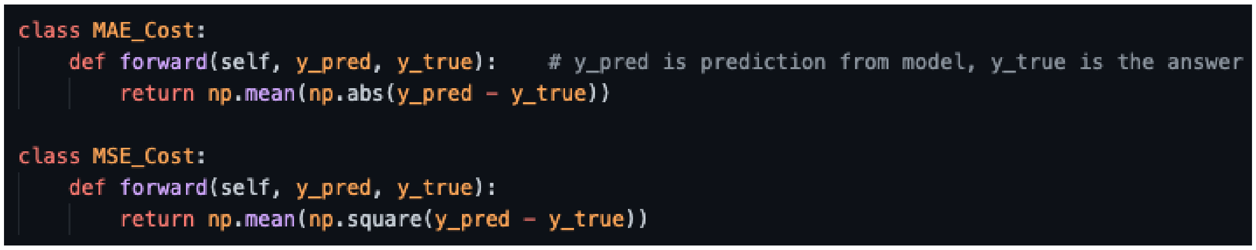
The Σ symbol is the uppercase Greek letter sigma. In math we call it a summation because it will add (sum) all the terms by plugging the bottom integer (i) inside the function, increase the variable i by 1, plug in new i into the function to add to the first result, and repeat until i is equal to n, we add the result of plugging in the new i to our sum of values one last time. After adding all the resulting values together we multiply that by the reciprocal of n (1/n) to get the cost of that example. Since in statistics, the **mean** of a set of values is the **average** (sum all the values and divide by the number of values) that is where the mean in MAE comes from.

For multiple examples with multiple output features we take the average/mean of each example's average/mean cost of output features. If **n** is the number of examples, **j** is the number of output features:

$$\frac{[(\frac{1}{j}\sum_{i=0}^{j} |\hat{y}_{1i} - y_{1i}|) + (\frac{1}{j}\sum_{i=0}^{j} |\hat{y}_{2i} - y_{2i}|) + \,..+ (\frac{1}{j}\sum_{i=0}^{j} |\hat{y}_{ni} - y_{ni}|)]}{n}$$

Basically we are taking the average value of this matrix: |ŷ - y| since ŷ and y are matrices with the same exact shape. As a result: **MAE = mean(|ŷ - y|)**.

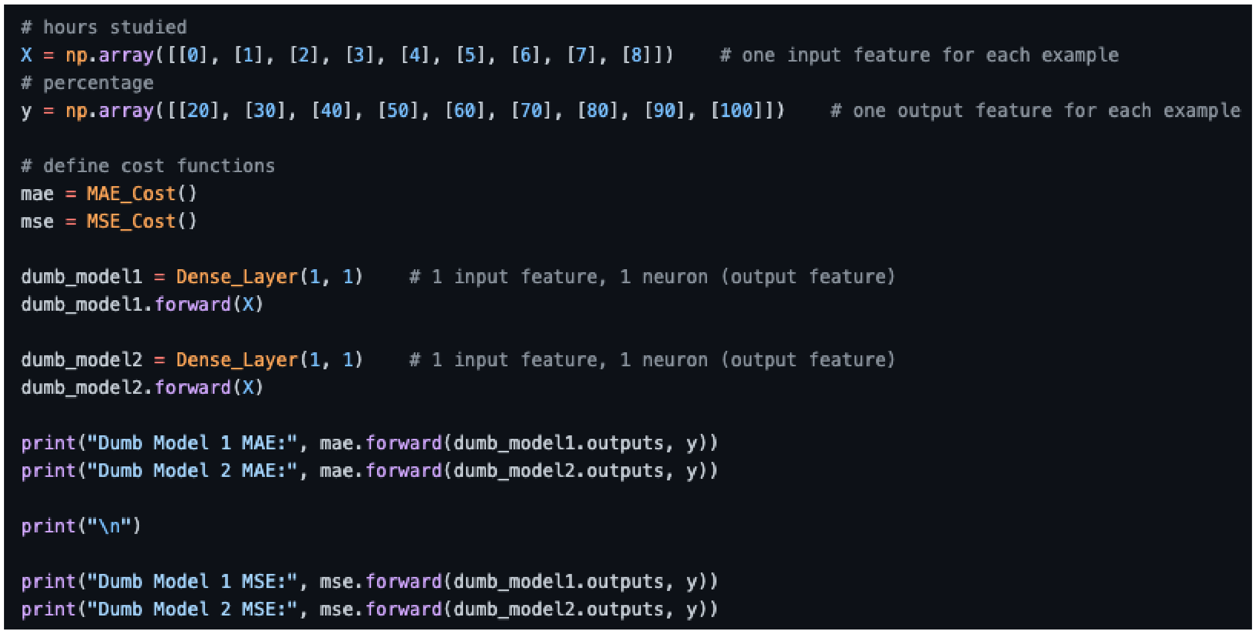
Now the only difference between MAE and MSE is that MSE uses the squared function not the absolute function to keep error positive. **MSE = mean($(\hat{y} - y)^2$)**. This change results in different characteristics that we will discuss later. For now, here is the code for both loss functions:



```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))

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

The cost functions do not contain anything so we can skip the __init__ method. *np.mean()* takes the average/mean of the inputted numpy array. *np.abs()* creates abs in the inputted numpy array, just kidding! *np.abs()* takes the absolute value of the inputted numpy array while *np.square()* squares the inputted numpy array.



```python
# 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

# define cost functions
mae = MAE_Cost()
mse = MSE_Cost()

dumb_model1 = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
dumb_model1.forward(X)

dumb_model2 = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
dumb_model2.forward(X)

print("Dumb Model 1 MAE:", mae.forward(dumb_model1.outputs, y))
print("Dumb Model 2 MAE:", mae.forward(dumb_model2.outputs, y))

print("\n")

print("Dumb Model 1 MSE:", mse.forward(dumb_model1.outputs, y))
print("Dumb Model 2 MSE:", mse.forward(dumb_model2.outputs, y))
```

Output

```
Dumb Model 1 MAE: 59.9294379061613
Dumb Model 2 MAE: 59.98399371166532

Dumb Model 1 MSE: 4255.854199207604
Dumb Model 2 MSE: 4264.212732073809
```

As we can see that since dumb model 1 is smarter its cost (error) is lower than dumb model 2 in both MAE and MSE. However, for both models MSE is higher than MAE. The reason is because the predictions of dumb model 1 and dumb model 2 are so far away from the actual answer so that squaring the differences results in huge values.

With this characteristic, MSE tends to work better than MAE on datasets without outliers since the squaring of the differences results in models trying to learn faster in the early stages so that its cost is not hugely inflated.

On the other hand, MAE tends to work better than MSE on datasets with outliers since an outlier on MSE can hugely increase the overall cost of the model due to the square of the MSE. Therefore during training MSE gives a greater influence on the model to get the outlier correct than MAE (due to the linear nature of the absolute value function) even though that example is an outlier, making MAE more robust on datasets with some outliers.

Now that we can tell the model how well it's doing based on an error metric, how do we **make** the model **use** this **information** to get smarter? We will partially answer that question in the next unit.