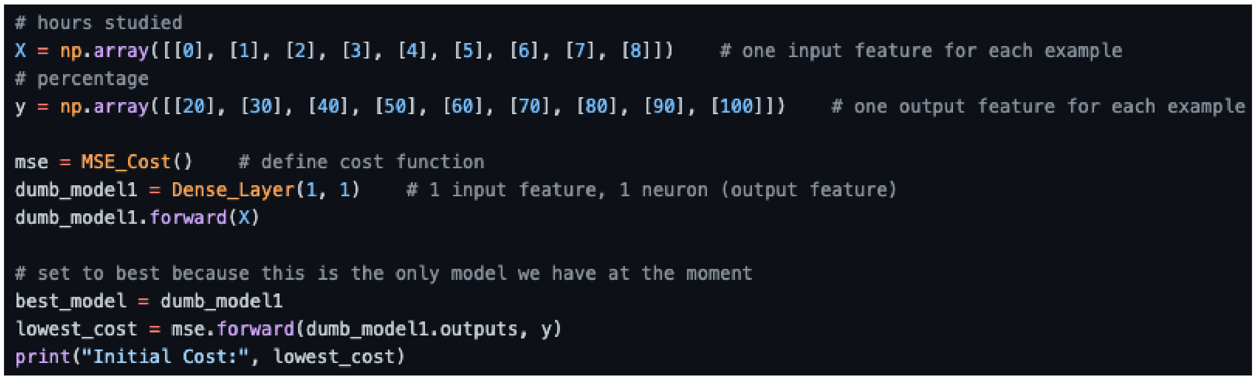
Chapter 03| Possible Error Reduction Methods

~~~~~~

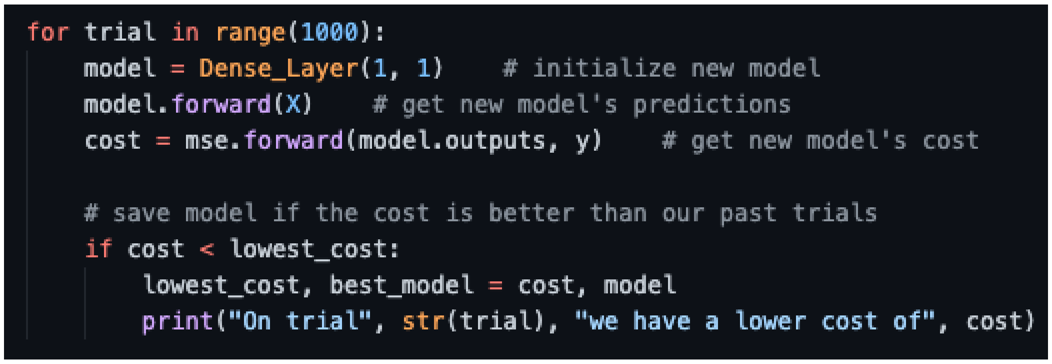
Random, Random, Random!

The most simplest method is to continuously initialize new models with different parameters (weights & biases) with a loop and keep the model that is the “smartest” which will be the model that has the lowest cost. Here is the code:

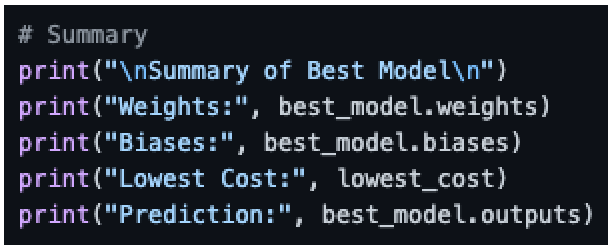


```
# hours studied
X = np.array([[0], [1], [2], [3], [4], [5], [6], [7], [8]])    # one input feature for each example
# percentage
y = np.array([[20], [30], [40], [50], [60], [70], [80], [90], [100]])    # one output feature for each example

mse = MSE_Cost()    # define cost function
dumb_model1 = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
dumb_model1.forward(X)

# set to best because this is the only model we have at the moment
best_model = dumb_model1
lowest_cost = mse.forward(dumb_model1.outputs, y)
print("Initial Cost:", lowest_cost)
```

This part is to set things up for the loop and then we update best_model and the lowest_cost when the new model we initialize has a cost lower than lowest_cost. Since our dataset does not consist of outliers, we will use the MSE cost function.



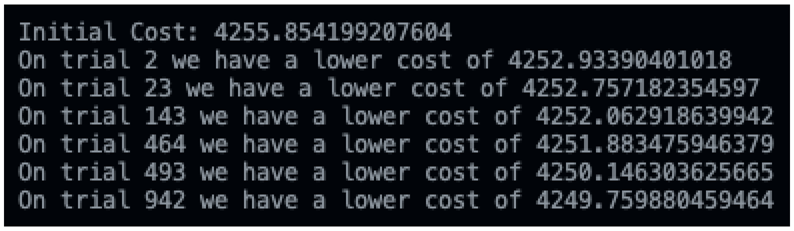
```
for trial in range(1000):
    model = Dense_Layer(1, 1)    # initialize new model
    model.forward(X)    # get new model's predictions
    cost = mse.forward(model.outputs, y)    # get new model's cost

    # save model if the cost is better than our past trials
    if cost < lowest_cost:
        lowest_cost, best_model = cost, model
        print("On trial", str(trial), "we have a lower cost of", cost)
```



```
# Summary
print("\nSummary of Best Model\n")
print("Weights:", best_model.weights)
print("Biases:", best_model.biases)
print("Lowest Cost:", lowest_cost)
print("Prediction:", best_model.outputs)
```

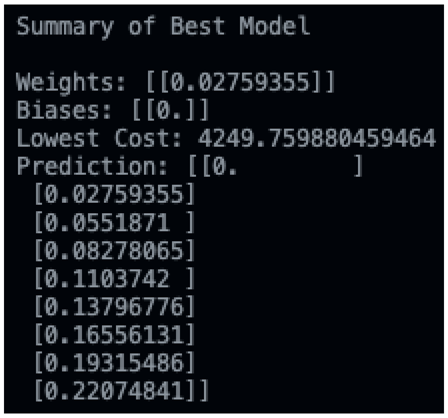
We will initialize 1000 new models to find better models. Also we print out the times we do get a smarter model to update us. Lastly, we summarize the characteristics of the best model we get out of the 1000 we tried.

Here are the updates from our 1000 initializations.
~~~~~~



```
Initial Cost: 4255.854199207604
On trial 2 we have a lower cost of 4252.93390401018
On trial 23 we have a lower cost of 4252.757182354597
On trial 143 we have a lower cost of 4252.062918639942
On trial 464 we have a lower cost of 4251.883475946379
On trial 493 we have a lower cost of 4250.146303625665
On trial 942 we have a lower cost of 4249.759880459464
```

As you can see, to continue getting more and more smarter models, the number of models we need to initialize increases exponentially so this method takes way too long. We begin by only waiting 2 trials, then 21 trials, 120 trials, 321 trials, 29 trials, and then 449 trials!

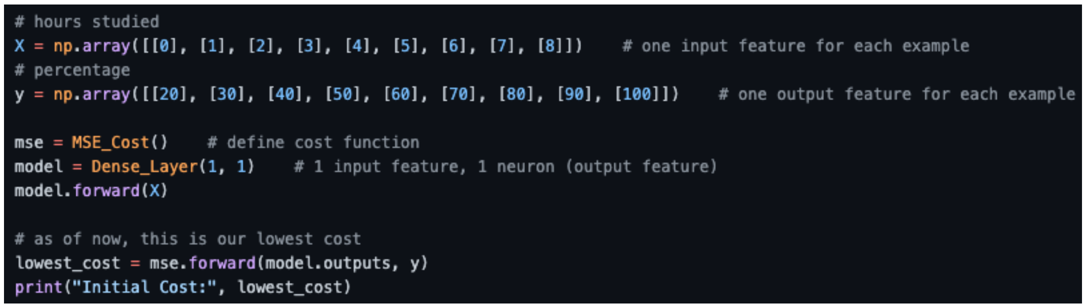


```
Summary of Best Model

Weights: [[0.02759355]]
Biases: [[0.]]
Lowest Cost: 4249.759880459464
Prediction: [[0.        ]
 [0.02759355]
 [0.0551871 ]
 [0.08278065]
 [0.1103742 ]
 [0.13796776]
 [0.16556131]
 [0.19315486]
 [0.22074841]]
```

To evaluate the performance of the best model based on the summary, we can take a look at the weights and biases. Remember $\hat{y} = wX + b$, where in math we have $y = mx + b$ and w is technically the slope with b the y-intercept. If we take a look at the slope of our dataset, for each extra hour of studying, we get an increase of 10% on the test. Therefore, the slope/weight should be 10 but for our best model we have ~0.0276, very far off! Also for the y-intercept of our dataset, no studying results in a 20% on the test. Therefore, the y-intercept/bias should be 20 but we have 0 because in our initialization method of the model, all biases are 0. As a result, constantly creating models does not help, we need something less random and more structured. One way is to **modify/update** the parameters instead of creating new ones.

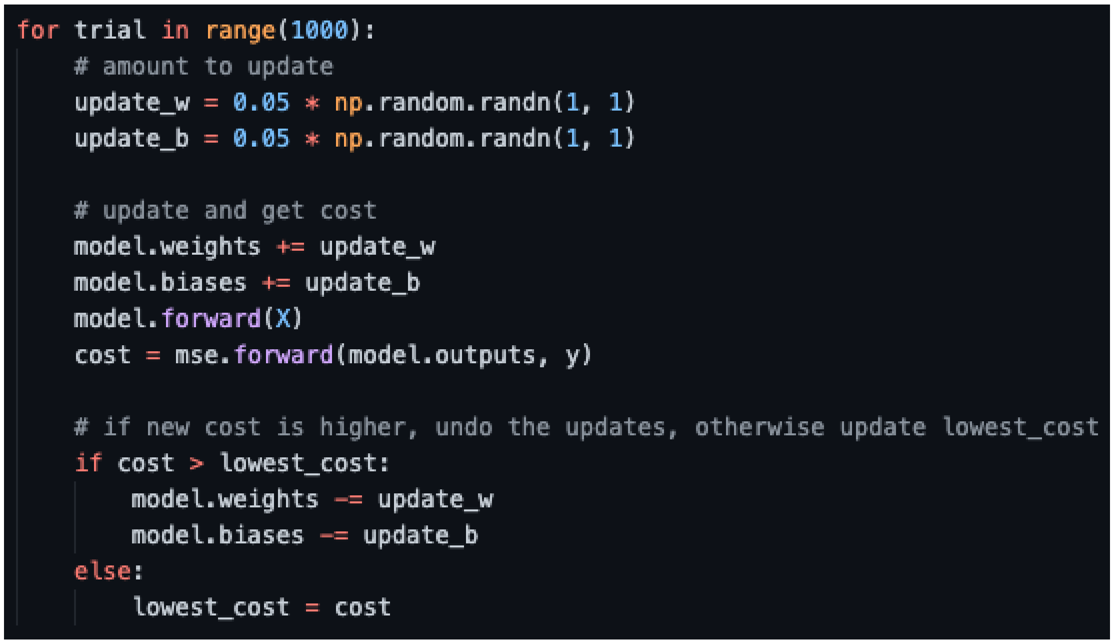
Random Updates!

Here is the code: The setup is almost the same but without the best_model variable.



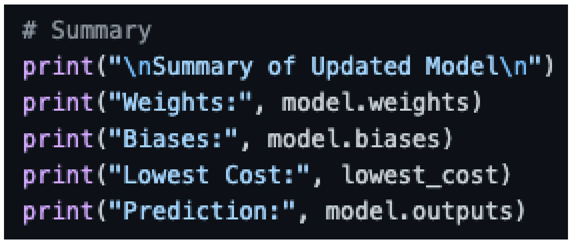
```python
# hours studied
X = np.array([[0], [1], [2], [3], [4], [5], [6], [7], [8]])    # one input feature for each example
# percentage
y = np.array([[20], [30], [40], [50], [60], [70], [80], [90], [100]])    # one output feature for each example

mse = MSE_Cost()    # define cost function
model = Dense_Layer(1, 1)    # 1 input feature, 1 neuron (output feature)
model.forward(X)

# as of now, this is our lowest cost
lowest_cost = mse.forward(model.outputs, y)
print("Initial Cost:", lowest_cost)
```

This is due to the fact that we are **modifying/updating** the parameters rather than creating new parameters so we stick with the same model. Now the loop for this method is more complex



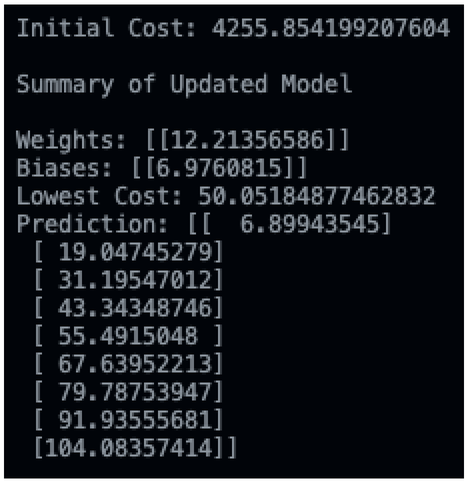
```python
for trial in range(1000):
    # amount to update
    update_w = 0.05 * np.random.randn(1, 1)
    update_b = 0.05 * np.random.randn(1, 1)

    # update and get cost
    model.weights += update_w
    model.biases += update_b
    model.forward(X)
    cost = mse.forward(model.outputs, y)

    # if new cost is higher, undo the updates, otherwise update lowest_cost
    if cost > lowest_cost:
        model.weights -= update_w
        model.biases -= update_b
    else:
        lowest_cost = cost
```

We ask numpy to generate some update values for us and use them to get a new prediction from the model and a new cost. If the cost is higher than our lowest_cost we undo the update as it is harmful, otherwise, we update lowest_cost to our new cost the lowest.



```python
# Summary
print("\nSummary of Updated Model\n")
print("Weights:", model.weights)
print("Biases:", model.biases)
print("Lowest Cost:", lowest_cost)
print("Prediction:", model.outputs)
```

Lastly, the summary section is mostly the same.



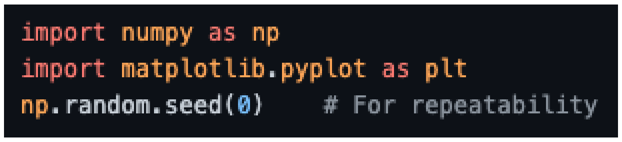
```
Initial Cost: 4255.854199207604

Summary of Updated Model

Weights: [[12.21356586]]
Biases: [[6.9760815]]
Lowest Cost: 50.05184877462832
Prediction: [[  6.89943545]
 [ 19.04745279]
 [ 31.19547012]
 [ 43.34348746]
 [ 55.4915048 ]
 [ 67.63952213]
 [ 79.78753947]
 [ 91.93555681]
 [104.08357414]]
```

The result is amazing! The cost jumped all the way down from ~4K (ha, caught in 4K) to ~50! Huge improvement. As a result, updating parameters is more **effective** than continuously getting new ones.

"You get what you get (parameters), and you don't throw a fit (find new models)".

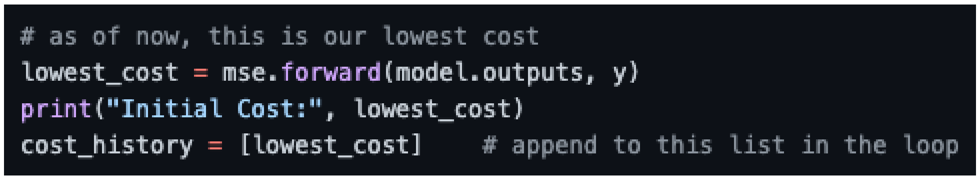
This method is more **effective** in a sense that it can make the model smarter but is it **efficient** in a sense that it can make the model smarter fast?

Inevitability, this depends on the amount of times our random updates harmed the model where the lowest_cost value stayed the same. To check the progress of our lowest_cost we can create a graph of the lowest_cost vs. the number of times we randomly updated the model. To create graphs in Python we can use the 3rd party library, Matplotlib.
Here is the pip install link for Matplotlib: https://pypi.org/project/matplotlib/
As for reference if major changes in Matplotlib occur in later versions, this book uses **Matplotlib 3.4.0.** Here is the new code:



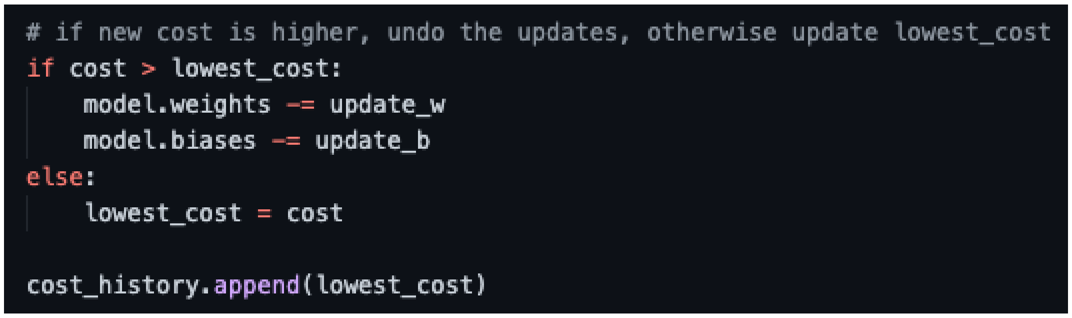
```python
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(0)     # For repeatability
```

We only import the pyplot (python plot) module of the package. Now for any plotting to occur, we need a set of values. This set will be the values of the lowest_cost.

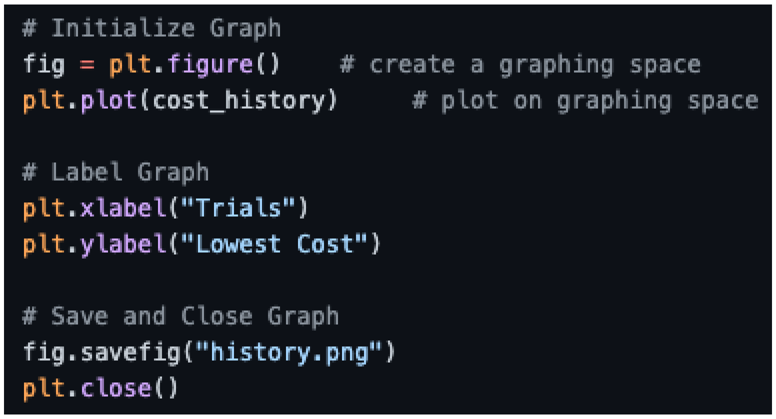


```python
# as of now, this is our lowest cost
lowest_cost = mse.forward(model.outputs, y)
print("Initial Cost:", lowest_cost)
cost_history = [lowest_cost]     # append to this list in the loop
```

As a result, we start off with a list that contains only 1 element.



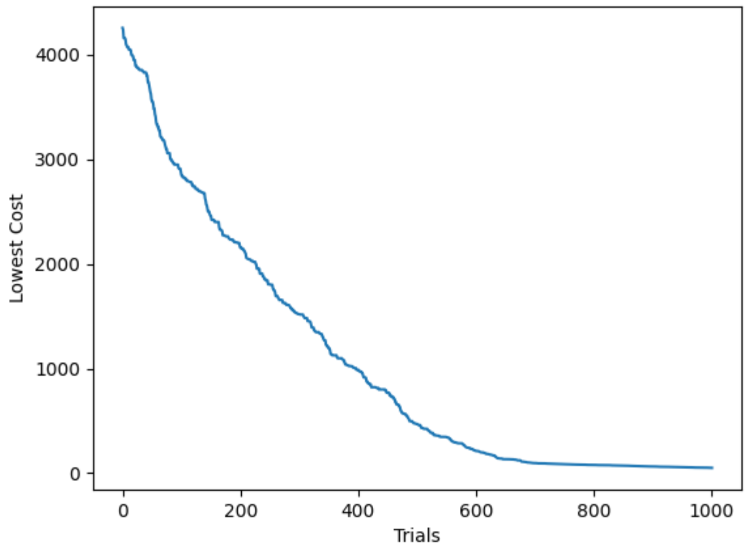
```python
# if new cost is higher, undo the updates, otherwise update lowest_cost
if cost > lowest_cost:
    model.weights -= update_w
    model.biases -= update_b
else:
    lowest_cost = cost

cost_history.append(lowest_cost)
```



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

# Label Graph
plt.xlabel("Trials")
plt.ylabel("Lowest Cost")

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

At the end of each trial we append our lowest cost to the list, thus creating a history of the variable, lowest_cost.

To begin, we need to create a graphing space which is a **figure** that can be created by *plt.figure()*. Next we plot our data on this figure using *plt.plot()*. Note that we only passed one argument (the y-axis data), because matplotlib will automatically create the x-axis data by starting from 0 to the number of y-axis values if only one argument is passed. Next, we label the x-axis and y-axis using *plt.xlabel()*, *plt.ylabel()* respectively. Lastly, we save the graph to a .png file and then close the figure.



As we can see this method overall will help the model learn but sometimes the updates slow down the model from learning. At the same time, it's nice how random updates can help a model learn, however our example is too oversimplified, one input feature, one neuron means 2 parameters (1 weight, 1 bias). When dealing with real world problems, thousands, millions, and now billions of parameters are used in a neural network. As a result, blindly updating these parameters will be very inefficient. To improve this method of updating parameters, we need to not blindly update them but update them on a rule, one of which is **gradient descent**. The next unit will have gradient descent as the main focus and will be relatively long as this concept transformed the field of AI. It may sure be complex at first but in reality it builds on the concept of **slope** from math.