Chapter 09| Binary Classification

~~~~~~

Introduction to Classification

Unlike regression, where we predict values for a given context based on the input information, in classification, we **sort** the input information into different categories.

In the classification problem, there are two types, **binary classification** and **multi-class classification**. Binary classification deals with the model sorting incoming data into 2 classes (categories). Examples of binary classification include predicting whether a tumor is cancerous or not, rainy or sunny weather, dog or cat image, etc. Multi-class classification is as obvious as its name suggests, where the model sorts incoming data into multiple (more than 2) classes (categories). Examples include, predicting the digit written from handwriting, predicting the language a video is in, etc.

Even though, in essence, these two types of classification are doing the same thing, sorting input data into categories, we deal with these problems differently. In this unit however, we are going to focus on binary classification. To start, let's review the process for creating a regression model:

1. Training Dataset – Create a dataset that has input data (X) which has a variety of examples. If needed, normalize/standardize the input data. From the input features of each example, create the corresponding "**correct answers**" (y) for the model to compare its predictions to and improve.

2. Define Neural Network/Model – Make sure the number of input features matches the number of input features of the training dataset. Make sure the number of output features matches the number of output features of the training dataset.

3. Training – Define Cost Function, Optimizer, and Learning Rate Decay if needed. For a number of epochs (full pass through the dataset), get the model's predictions (forward pass), get the cost, calculate the derivative cost function w.r.t the parameters (backward pass) (slope of cost-versus-parameter graph), update parameters with the optimizer, update the learning rate, if needed.
~~~~~~

Let's start with step 1, but thinking from a classification perspective. For the purpose of this book, we are not going to collect data but rather download data from online to save time since this is just an introduction. As a result, the input data (X) for classification would have different input features to be normalized/standardized.

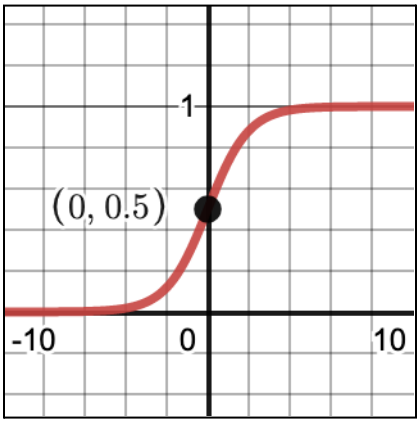
How about the "correct answers" (y)? In regression these were just all real numbers (any value) but in binary classification, how do we represent class/category A and class/category B? Well just like how we abstracted the classes/categories into A and B we would abstract the classes into numbers, 0 and 1. Any data that fits into class/category A could be represented by a 0, any data that fits into class/category B could be represented by a 1. To make this concept more tangible, suppose we are predicting whether the next day would have sunny or rainy weather, we could represent sunny weather as 0 and rainy weather as 1.

Now how do we get our model (neural network) to output (predict) a 0 or a 1? The most intuitive way is to produce confidence/probability levels from the outputs of the output layer. We would map the outputs of the output layer to a range of 0% to 100%. If the result is 100%, then the model is 100% confident or predicts a 100% probability that the next day will have rainy weather. If the result is 50%, then the model is 50% confident or predicts a 50% probability that the next day will have rainy weather. A result of 0% means that the model is 0% confident or predicts a 0% probability that the next day will have rainy weather. To get the confidence/probability of the prediction of the model for sunny weather the next day, simply subtract the rainy day probability/confidence from **100%.**

As a result, if the result is 100%, the model is 0% (**100%** - 100%) confident or predicts a 0% probability that the next day will have sunny weather. A result of 50% means that the model is 50% (**100%** - 50%) confident or predicts a 50% probability that the next day will have sunny weather. A result of 0% means that the model is 100% (**100%** - 0%) confident or predicts a 100% probability that the next day will have sunny weather. Although we only discussed the [0%, 50%, 100%] results, the interpretation of any result between [0%, 100%] is still the same.
Since we need to map the outputs of a layer of neurons, specifically the output layer to different outputs, we need an activation function. This activation function may be used only for the output layer or used for all layers (hidden and output layers), this depends on the human creating the model.

The activation function, sigmoid, was introduced before, when we first introduced the concept of activation functions.

Sigmoid: $a(z) = \frac{1}{1+e^{-z}}$



The *e* is not another variable, instead the *e* is an irrational number, more specifically, Euler's number ≈ 2.71828183.

As you can see, the range (possible output values) of sigmoid is [0, 1], perfect for producing the confidence/probability level.

Negative outputs from the output layer are negative inputs to the sigmoid function so this results in sigmoid outputting probabilities or a confidence less than 0.5 (50%). Positive outputs from the output layer are positive inputs to the sigmoid function so this results in sigmoid outputting probabilities or a confidence greater than 0.5 (50%). If the output layer outputs a 0, the sigmoid activation function returns a probability/confidence of 50%.

Note that for binary classification, the output layer only has one output feature for the sigmoid activation function. The only output feature is the probability/confidence of the model for sorting a certain example into one class or the other class.

The equation above represents the calculation for the forward pass of the sigmoid activation function, how about the backward pass? Remember that the backward pass is all about calculating the analytical derivative of the function (slope at a point). Using algebra to calculate the analytical derivative of sigmoid takes many steps and simply using calculus would be better. Despite the difficulty of finding the analytical derivative, the result is very easy to calculate: a(z) * [1 - a(z)], is the output of the sigmoid function multiplied by the quantity 1 minus the output of the sigmoid function. Here is the code:

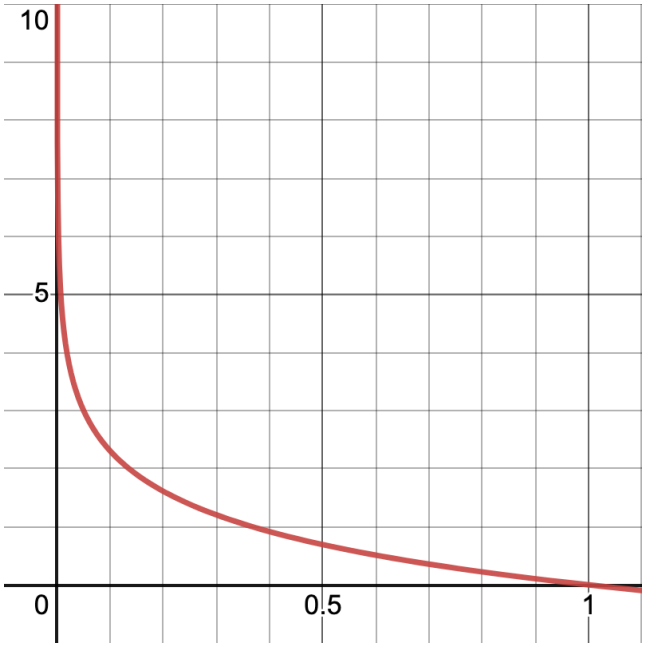
```python
class Sigmoid_Activation:
    def forward(self, inputs):    # inputs are outputs from dense layer
        self.outputs = 1 / (1 + np.exp(-inputs))

    def backward(self, dvalues):    # dvalues has same shape as inputs
        self.dinputs = dvalues * (self.outputs * (1 - self.outputs))
```

In the forward method we store the outputs as *self.outputs*. To raise e to the power of the negative inputs we use *np.exp* and the argument to it is the power we want to raise e to. In the backward method we calculate the analytical derivative of the sigmoid function and then use chain rule by multiplying *dvalues* which should be *dinputs* from the cost function.

Now we have completed steps 1 and 2 for creating a model. We have defined how we want the training data structured and added an activation function. Now that we have got to the backward method of the sigmoid activation function, that requires *dinputs* from the cost function, we need to figure out what our cost function will be. We could use MSE or MAE but there is a problem.

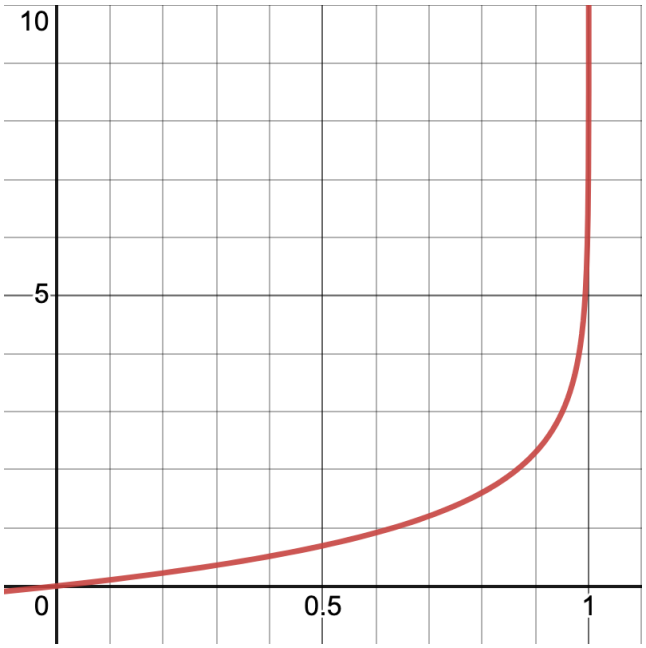
All the correct answers would either be 0 or 1 and the predictions would always be in the range of [0, 1]. As a result, the difference between the predictions and correct answers would always be less than 1. The model would not get penalized very much for predictions far off anymore. To fix this problem, we have to use a different cost function that is effective for binary classification.

Let's start with the case where the correct answer is 1 (class 1). If the prediction is 1 we would like the cost to be 0 and any predictions further and further should have exponentially increasing cost to penalize the model. A good equation to use would be: **-ln(ŷ)**. A graph of the equation is one the right. The y-axis represents the cost and the x-axis represents the model's prediction, ŷ.



To think about this intuitively, we use the probability/confidence of the model that the example is class 1 and the greater ŷ is, the smarter the model is since the correct class is class 1 and thus the smaller the cost becomes.

How about the case where the correct answer is 0 (class 0)? We would like the cost to be 0 if the model's prediction is 0 and an exponentially increasing cost as the model's prediction is further and further away from the correct answer. The equation we can use would be: **-ln(1- ŷ)**. A graph of the equation is on the right. The y-axis represents the cost and the x-axis represents the model's prediction, ŷ.



Thinking about this intuitively, we use the probability/confidence of the model that the example is class 0 which is found by subtracting ŷ from 1. The smaller ŷ is, the larger the probability/confidence for class 0, the smarter the model is since the correct class is class 0 and thus the smaller the cost becomes.

Now how do we combine these two equations to form a cost function?
Luckily there is a mathematical trick to combine these two equations into one. This type of cost function for binary classification is called **Binary Cross-Entropy** (BCE):

$$BCE = \underbrace{[y * \ -ln(\hat{y})]}_{\text{Part 1}} + \underbrace{[(1-y) * \ -ln(1-\hat{y})]}_{\text{Part 2}}$$

To better understand how this equation works, we split this equation into two parts. In part 1, **if the correct answer (y) is 1**, we have 1 * -ln(ŷ) = **-ln(ŷ)**. In part 2, we would have (1 - 1) * -ln(1 - ŷ) = 0 * -ln(1 - ŷ) = **0**. As a result, adding these two parts together:
BCE = -ln(ŷ) + 0 = **-ln(ŷ)**, exactly what we defined previously for **when y = 1**.
TLDR: Part 1 of this cost function is for **when y = 1**.

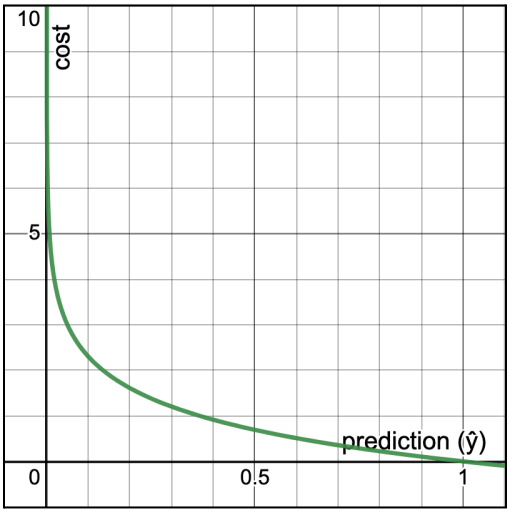
What **if the correct answer (y) is 0**? In part 1, we have 0 * -ln(ŷ) = **0**. In part 2, we would have (1 - 0) * -ln(1 -ŷ) = 1 * -ln(1 - ŷ) = **-ln(1 - ŷ)**. Adding these two parts together: BCE = 0 + (-ln(1 - ŷ)) = **-ln(1 - ŷ)** which is exactly what we defined previously **when y = 0**. TLDR: Part 2 of this cost function is for **when y = 0**.

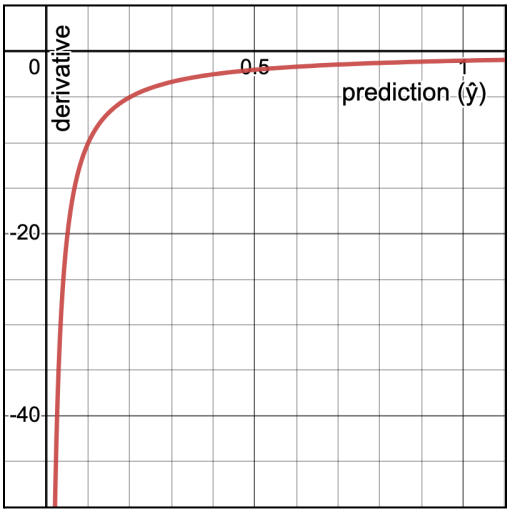
In essence, **when y = 1**, part 2 equals 0, leaving **-ln(ŷ)** which is part 1 and **when y = 0**, part 1 equals 0, leaving **-ln(1 - ŷ)** which is part 2.

This is a very neat mathematical trick! Since the outputs from the sigmoid activation function have one input feature, we only have to average the individual costs of each example for this cost function. Moving on to the backward pass.

Just like the forward pass, for the backward pass we would have to split the function into 2 parts, when y = 1 and when y = 0. The derivative (slope at any point) on the BCE cost function when y = 1, is $-\frac{1}{\hat{y}}$ and when y = 0, is $\frac{1}{1-\hat{y}}$. Here is a PDF if you would like to see the derivation process for getting the analytical derivative of BCE. Before combining these 2 parts, let's understand what the 2 equations actually mean.

When y = 1, the cost is **-ln(ŷ)** and the derivative (slope at any point) on BCE is $-\frac{1}{\hat{y}}$.

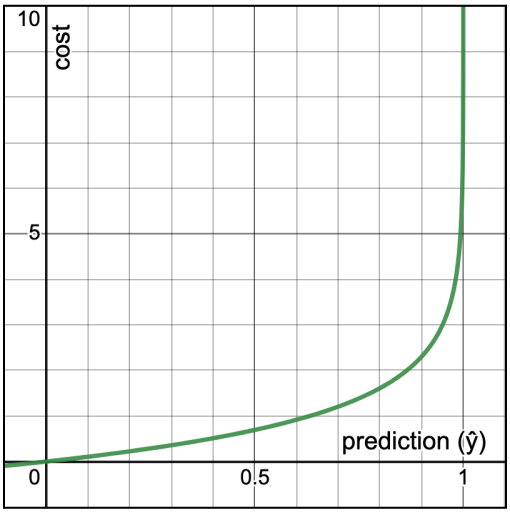


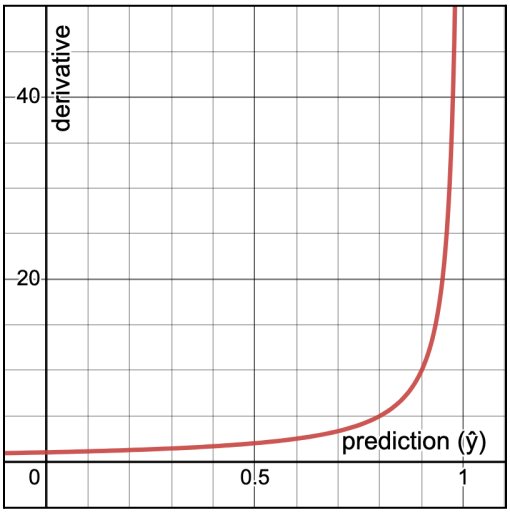


As the prediction gets closer to the correct answer (1), the amount of decrease in the cost becomes smaller.

All increases in ŷ mean that the prediction is closer to the correct answer (1), resulting in a **decrease** in the cost. This explains why all the derivatives are **negative**.

When y = 0, the cost is -ln(1 - ŷ) and the derivative (slope at any point) on BCE is $\frac{1}{1-\hat{y}}$.





As the prediction gets further away from the correct answer (0), the amount of increase in the cost becomes larger.

All increases in ŷ mean that the prediction is farther to the correct answer (0), resulting in an **increase** in the cost. This explains why all the derivatives are **positive**.

Notice in the graphs when y = 0, as ŷ approaches the wrong answer (0), the cost & derivative values **explode** and when y = 1, as ŷ approaches the wrong answer (1), the cost & derivative values **explode** as well. By **explode**, we mean that the value is **unbounded** which is -∞ or +∞. Since computers can't work with infinite numbers, we are going to need to handle the extreme values when coding the BCE class.

Now we can combine these 2 parts through the same process in the forward method. In the forward method we multiplied y to part 1 and multiplied (1 - y) to part 2 and then added part 1 and part 2 together. As a result this is BCE's backward method:

$$\underbrace{-\frac{y}{\hat{y}}}_{\text{Part 1}} + \underbrace{\frac{1-y}{1-\hat{y}}}_{\text{Part 2}}$$

Same mathematical trick: **When y = 1**, part 2 equals 0, leaving part 1: $-\frac{1}{\hat{y}}$ which is $-\frac{y}{\hat{y}}$ after substituting the y in the numerator as 1. **When y = 0**, part 1 equals 0, leaving part 2: $\frac{1}{1-\hat{y}}$ which is $\frac{1-y}{1-\hat{y}}$ after substituting the y in the numerator as 0.

Now that we have gone over the forward & backward method of BCE, here is the code for a **partial** implementation of BCE (we still have to handle for the extreme values):

```
class BCE_Cost:
    def forward(self, y_pred, y_true):
        return np.mean((y_true * -np.log(y_pred)) + ((1 - y_true) * -np.log(1 - y_pred)))

    def backward(self, y_pred, y_true):
        self.dinputs = -(y_true / y_pred) + ((1 - y_true) / (1 - y_pred))
        self.dinputs /= len(y_pred)
```

Remember that *y_pred* and *y_true* both have shapes **[# of examples, 1]**, since the only output feature is the **probability/confidence** level from the model. To get the natural logarithm in the forward method, we use ***np.log()*** which is the equivalent as **ln()** in math. Lastly we **average** the cost (error) of each example by using ***np.mean()***. In the backward method, each example is divided by the **# of examples, *len(y_pred)***, after calculating their derivatives.

The reason for this division is ultimately because in the forward method we are taking the **average/mean** of each example's cost. This is the equivalent of taking a **fraction** of each example's cost and then adding them up. To be mathematically consistent, we have to use a **fraction** of each example's derivative in the backward method. The **fraction** used in the forward method comes from the mathematical definition of an **average/mean** which is (1/**n**) where **n** is the # of examples. To use this **fraction** in the backward method we just have to divide by **n** for each example.

Now how do we handle the extreme prediction values? When y = 1, the extreme prediction value that makes the cost and derivative explode is when the prediction is 0. When y = 0, the extreme prediction value that makes the cost and derivative explode is when the prediction is 1. The only way to not hit any errors is to **artificially modify** the model's prediction before calculating the cost or derivative.

More specifically, set bounds on the prediction of the model. For any prediction value less than 1e-7 ($1 * 10^{-7}$) which is 0.0000001, we **artificially modify** them to be equal to exactly 1e-7. This is beneficial for when y = 1 and the prediction is 0. For any prediction value greater than 1 - 1e-7 ($1 - (1 * 10^{-7})$) which is 0.9999999, we **artificially modify** them to be equal to exactly 1 - 1e-7. This is beneficial for when y = 0 and the prediction is 1.

This process of **artificially modifying** the model's prediction with bounds is called **clipping** and can be used in both the forward and backward pass of BCE. So now we can fully implement BCE. Here is the code:

```
class BCE_Cost:
    def forward(self, y_pred, y_true):
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
        return np.mean((y_true * -np.log(y_pred_clipped)) + \
            ((1 - y_true) * -np.log(1 - y_pred_clipped)))

    def backward(self, y_pred, y_true):
        y_pred_clipped = np.clip(y_pred, 1e-7, 1 - 1e-7)
        self.dinputs = -(y_true / y_pred_clipped) + \
            ((1 - y_true) / (1 - y_pred_clipped))

        self.dinputs /= len(y_pred_clipped)
```

To **clip** the model's predictions we can use *np.clip()* where the 1st argument requires the NumPy array that needs to be **clipped**, the 2nd argument sets the lower bound (smallest value accepted), and the 3rd argument sets the upper bound (largest value accepted).

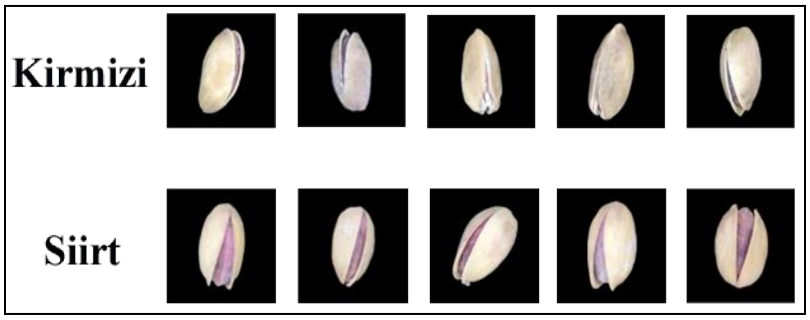
After clipping, we would expect that there is no value in *y_pred_clipped* less than 0.0000001 and no value greater than 0.9999999. Lastly, remember in the forward and backward passes, replace *y_pred* with *y_pred_clipped*. Of course, if you like seeing RED ERROR MESSAGES, you don't have to. Just kidding 😂, no one likes seeing RED ERROR MESSAGES.

Now from the cost function we need to update the parameters using an optimizer and if needed an optimizer paired with a learning rate decayer. For binary classification we needed a "special" way to store the correct answers (y), a "special" activation function and a "special" cost function but there is no "special" optimizer or learning rate decayer. As a result, now we have almost all the tools needed to begin training a binary classification model. Training requires a dataset for a model to predict on. So in the next section we will be downloading a dataset to demonstrate a binary classification model.

Dataset Download

For an example of a binary classification model, we are going to be using a fun dataset. More specifically, this dataset is going to be about....Pistachios! Obviously 🙄 from the image on the right.

The dataset we are using is the **Pistachio Dataset** where the input to the model are the physical characteristics of a pistachio and the model has to predict whether that pistachio is a **Kirmizi Pistachio** or a **Siirt Pistachio**. Below are examples of them for comparison (image from this paper):

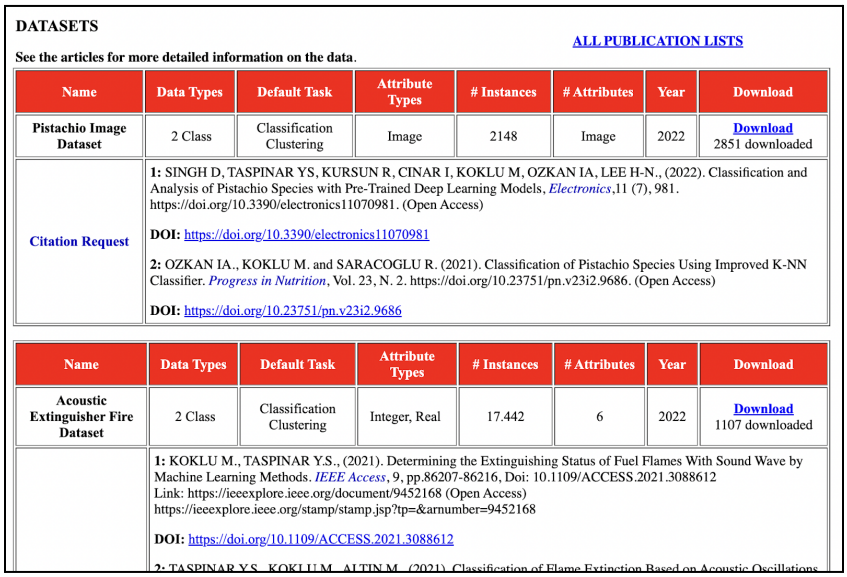


Rather than working directly with image data where the dataset is full of images, we are going to use a dataset that gives 16 different characteristics of each pistachio.

The main reason for doing this is because working with images usually requires a different type of neural network, a CNN (**c**onvolutional **n**eural **n**etwork). This book is only an introduction to AI so we are only going to cover regression, and the 2 types of classification problems that neural networks can handle.

First let's download the dataset itself so that you can better understand what AI task we are dealing with.

1. Go to https://www.muratkoklu.com/datasets/. The site should look something like this:



DATASETS

ALL PUBLICATION LISTS

See the articles for more detailed information on the data.

Name	Data Types	Default Task	Attribute Types	# Instances	# Attributes	Year	Download
Pistachio Image Dataset	2 Class	Classification Clustering	Image	2148	Image	2022	**Download** 2851 downloaded
Citation Request	**1:** SINGH D, TASPINAR YS, KURSUN R, CINAR I, KOKLU M, OZKAN IA, LEE H-N., (2022). Classification and Analysis of Pistachio Species with Pre-Trained Deep Learning Models, *Electronics*,11 (7), 981. https://doi.org/10.3390/electronics11070981. (Open Access) **DOI:** https://doi.org/10.3390/electronics11070981 **2:** OZKAN IA., KOKLU M. and SARACOGLU R. (2021). Classification of Pistachio Species Using Improved K-NN Classifier. *Progress in Nutrition*, Vol. 23, N. 2. https://doi.org/10.23751/pn.v23i2.9686. (Open Access) **DOI:** https://doi.org/10.23751/pn.v23i2.9686						

Name	Data Types	Default Task	Attribute Types	# Instances	# Attributes	Year	Download
Acoustic Extinguisher Fire Dataset	2 Class	Classification Clustering	Integer, Real	17.442	6	2022	**Download** 1107 downloaded
	1: KOKLU M., TASPINAR Y.S., (2021). Determining the Extinguishing Status of Fuel Flames With Sound Wave by Machine Learning Methods. *IEEE Access*, 9, pp.86207-86216, Doi: 10.1109/ACCESS.2021.3088612 Link: https://ieeexplore.ieee.org/document/9452168 (Open Access) https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9452168 **DOI:** https://doi.org/10.1109/ACCESS.2021.3088612 **2:** TASPINAR Y.S., KOKLU M., ALTIN M., (2021). Classification of Flame Extinction Based on Acoustic Oscillations						

2. Scroll down until you find the **Pistachio Dataset** (should be somewhere near the middle of the webpage). The table with a description of the dataset should look something like the image below.

Name	Data Types	Default Task	Attribute Types	# Instances	# Attributes	Year	Download
Pistachio Dataset	2 Class	Classification Clustering	Integer, Real	2148	16 28	2021	**Download** 512 downloaded
Citation Request	OZKAN IA., KOKLU M. and SARACOGLU R. (2021). Classification of Pistachio Species Using Improved K-NN Classifier. *Progress in Nutrition*, Vol. 23, N. 2. https://doi.org/10.23751/pn.v23i2.9686. **DOI:**https://doi.org/10.23751/pn.v23i2.9686						

3. Click on the blue Download button located on the right side of the table. You should find a ZIP file, named *Pistachio_Dataset.zip* on your computer now.

4. Open up the ZIP file, extract the contents and you will find a folder in the same directory named *Pistachio_Dataset*, which should contain 2 other folders: *Pistachio_16_Features_Dataset* & *Pistachio_28_Features_Dataset*. In the 16 feature folder, copy/duplicate the **XLSX** file to the directory you are working in (where your Python scripts for learning AI is in).

Now you have successfully downloaded your second dataset! Even though this dataset is not in a **csv** file, **Pandas** is a very powerful python library, allowing us to also read **xlsx** files. In the next section we will read the dataset in Python, prepare the data to be usable in training, initialize the model, cost function, optimizer and begin the training process for a binary classifier!

Training Setup & Initial Training

To read a **xlsx** file, we can use ***pandas.read_excel()***. However this function requires the optional dependency, openpyxl, "a python library to read/write Excel 2010 xlsx/xlsm files". Here is the pip install link for openpyxl: https://pypi.org/project/openpyxl/3.0.10/. As for reference if major changes in openpyxl occur in later versions, this book uses **openpyxl 3.0.10**.

When working with a new dataset, it is best to see one operation at a time to familiarize yourself with the characteristics of the dataset itself. Thus, launch the command prompt/line or terminal on your computer, navigate to the directory the **csv** is located in and then use the 'python' command to launch the shell. The shell should look something like this:

```
Python 3.9.5 (default, May 27 2021, 19:45:35)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Reading the **xlsx** file through pandas allows us to convert the data into a NumPy array as input to the model for training. As a result, let's import both NumPy and Pandas.

```
>>> import numpy as np    # version 1.22.2
>>> import pandas as pd    # version 1.4.0
>>>
```

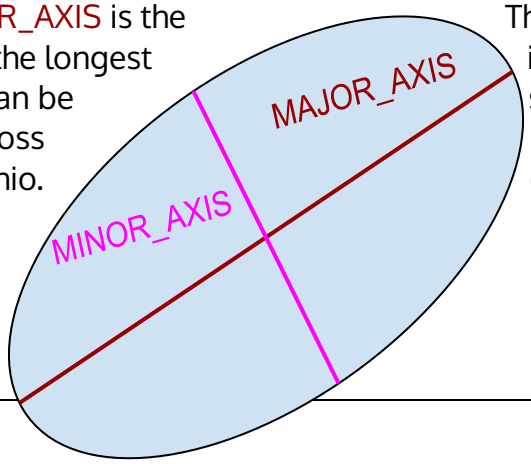
Now to read the dataset we use ***pandas.read_excel()*** and input the file name of the **xlsx** file. We save the data to ***dataset*** and then we print it to see the contents.

```
>>> dataset = pd.read_excel("Pistachio_16_Features_Dataset.xlsx")
>>> print(dataset)
        AREA  PERIMETER  MAJOR_AXIS  ...  SHAPEFACTOR_3  SHAPEFACTOR_4             Class
0      63391  1568.4050    390.3396  ...         0.5297         0.8734  Kirmizi_Pistachio
1      68358  1942.1870    410.8594  ...         0.5156         0.9024  Kirmizi_Pistachio
2      73589  1246.5380    452.3630  ...         0.4579         0.9391  Kirmizi_Pistachio
3      71106  1445.2610    429.5291  ...         0.4907         0.9755  Kirmizi_Pistachio
4      80087  1251.5240    469.3783  ...         0.4628         0.9833  Kirmizi_Pistachio
...      ...        ...         ...  ...            ...            ...                ...
2143   85983  1157.1160    444.3447  ...         0.5545         0.9900     Siit_Pistachio
2144   85691  2327.3459    439.8794  ...         0.5639         0.8892     Siit_Pistachio
2145  101136  1255.6190    475.2161  ...         0.5702         0.9987     Siit_Pistachio
2146   97409  1195.2150    452.1823  ...         0.6066         0.9989     Siit_Pistachio
2147   78466  2356.9080    445.9131  ...         0.5024         0.8667     Siit_Pistachio

[2148 rows x 17 columns]
>>>
```

We see that there are 2148 rows & 17 columns. That means there are 2148 examples for the model to train on and each example has 16 input features, 1 output feature. The 1 output feature would be the last column, Class, since we are going to predict whether it is a Kirmizi Pistachio or a Siit Pistachio. Now let's figure out what types of input features we have. We can do this by printing the *.columns* attribute of *dataset*.

```
>>> print(dataset.columns)
Index(['AREA', 'PERIMETER', 'MAJOR_AXIS', 'MINOR_AXIS', 'ECCENTRICITY',
       'EQDIASQ', 'SOLIDITY', 'CONVEX_AREA', 'EXTENT', 'ASPECT_RATIO',
       'ROUNDNESS', 'COMPACTNESS', 'SHAPEFACTOR_1', 'SHAPEFACTOR_2',
       'SHAPEFACTOR_3', 'SHAPEFACTOR_4', 'Class'],
      dtype='object')
>>>
```

For simplicity we are only going to use 8 of the 16 input features, AREA, PERIMETER, MAJOR_AXIS, MINOR_AXIS, ASPECT_RATIO, ROUNDNESS, SHAPEFACTOR_1, SHAPEFACTOR_2. Below is a table that gives a description for each input feature.

Input Feature	Description
AREA	the number of pixels the pistachio took up in the image
PERIMETER	circumference (length of the pistachio border in the image)
MAJOR_AXIS & MINOR_AXIS	The MAJOR_AXIS is the length of the longest line that can be drawn across the pistachio.  The MINOR_AXIS is the length of shortest line that can be drawn across the pistachio.

ASPECT_RATIO	ratio between MAJOR_AXIS and MINOR_AXIS (calculated by MAJOR_AXIS ÷ MINOR_AXIS
ROUNDNESS	measure to find the close the pistachio resembles a perfect circle
SHAPEFACTOR_1 & SHAPEFACTOR_2	SHAPEFACTOR_1 is the ratio between MAJOR_AXIS and AREA (calculated by MAJOR_AXIS ÷ AREA). SHAPEFACTOR_2 is the ratio between MINOR_AXIS and AREA (calculated by MINOR_AXIS ÷ AREA)

To use only those 8 input features, we have to drop the un-needed input features, ECCENTRICITY, EQDIASQ, SOLIDITY, CONVEX_AREA, EXTENT, COMPACTNESS, SHAPEFACTOR_3, SHAPEFACTOR_4. To do this we need to first create a list of the names of these columns we want to drop.

```
>>> to_drop = ['ECCENTRICITY', 'EQDIASQ', 'SOLIDITY', 'CONVEX_AREA', 'EXTENT', 'COMPACTNESS', 'SHAPEFACTOR_3', 'SHAPEFACTOR_4']
>>>
```

Now we can call the .drop() method by passing the column labels, *to_drop*, and since these labels are found on the column axis, we have to pass in *axis=1*.

```
>>> dataset = dataset.drop(labels=to_drop, axis=1)
>>> print(dataset)
       AREA  PERIMETER  MAJOR_AXIS  MINOR_AXIS  ...  ROUNDNESS  SHAPEFACTOR_1  SHAPEFACTOR_2            Class
0     63391  1568.4050    390.3396    236.7461  ...     0.3238         0.0062         0.0037  Kirmizi_Pistachio
1     68358  1942.1870    410.8594    234.7525  ...     0.2277         0.0060         0.0034  Kirmizi_Pistachio
2     73589  1246.5380    452.3630    220.5547  ...     0.5951         0.0061         0.0030  Kirmizi_Pistachio
3     71106  1445.2610    429.5291    216.0765  ...     0.4278         0.0060         0.0030  Kirmizi_Pistachio
4     80087  1251.5240    469.3783    220.9344  ...     0.6425         0.0059         0.0028  Kirmizi_Pistachio
...     ...        ...         ...         ...  ...        ...            ...            ...                ...
2143  85983  1157.1160    444.3447    248.8627  ...     0.8070         0.0052         0.0029     Siit_Pistachio
2144  85691  2327.3459    439.8794    278.9297  ...     0.1988         0.0051         0.0033     Siit_Pistachio
2145 101136  1255.6190    475.2161    271.3299  ...     0.8061         0.0047         0.0027     Siit_Pistachio
2146  97409  1195.2150    452.1823    274.5764  ...     0.8569         0.0046         0.0028     Siit_Pistachio
2147  78466  2356.9080    445.9131    258.5125  ...     0.1775         0.0057         0.0033     Siit_Pistachio

[2148 rows x 9 columns]
>>>
```

The number of rows stayed the same which is expected and now we have 9 columns, 8 input features and 1 output feature. Now before we convert this to a NumPy array we have to convert the Class values into 0s and 1s. We will convert Kirmizi Pistachio to a 0 and a Siit Pistachio to a 1.

```
>>> classes = {"Kirmizi_Pistachio": 0, "Siit_Pistachio": 1}
>>> dataset['Class'] = dataset['Class'].replace(classes)
>>> print(dataset)
       AREA  PERIMETER  MAJOR_AXIS  MINOR_AXIS  ...  ROUNDNESS  SHAPEFACTOR_1  SHAPEFACTOR_2  Class
0     63391  1568.4050    390.3396    236.7461  ...     0.3238         0.0062         0.0037      0
1     68358  1942.1870    410.8594    234.7525  ...     0.2277         0.0060         0.0034      0
2     73589  1246.5380    452.3630    220.5547  ...     0.5951         0.0061         0.0030      0
3     71106  1445.2610    429.5291    216.0765  ...     0.4278         0.0060         0.0030      0
4     80087  1251.5240    469.3783    220.9344  ...     0.6425         0.0059         0.0028      0
...     ...        ...         ...         ...  ...        ...            ...            ...    ...
2143  85983  1157.1160    444.3447    248.8627  ...     0.8070         0.0052         0.0029      1
2144  85691  2327.3459    439.8794    278.9297  ...     0.1988         0.0051         0.0033      1
2145 101136  1255.6190    475.2161    271.3299  ...     0.8061         0.0047         0.0027      1
2146  97409  1195.2150    452.1823    274.5764  ...     0.8569         0.0046         0.0028      1
2147  78466  2356.9080    445.9131    258.5125  ...     0.1775         0.0057         0.0033      1

[2148 rows x 9 columns]
>>>
```

First we need to create a dictionary where the "Kirmizi_Pistachio" key maps to the value of 0 and the "Siit_pistachio" key maps to the value of 1. Next we index for the "Class" column of the dataset called the ***.replace()*** method. This will replace all "Kirmizi_Pistachio" values found in the "Class" column of the dataset to a 0 and all "Siit_pistachio" values found in the "Class" column of the dataset to a 1 because it uses the values found in the "Class" column of the dataset as keys to the inputted dictionary and sets that value corresponding value of the key from the inputted dictionary.

Now that all the data in the dataset are numerical values, we can convert the data to a NumPy array.

```
>>> dataset = dataset.to_numpy(dtype=float)
>>> print(dataset.shape)
(2148, 9)
>>>
```

To convert we use the **.to_numpy()** method and pass in the keyword argument dtype=float since now all the data values are floats.

We print out the shape of the dataset to check that we still have the same number of rows and columns. The last thing to do now is to split the data into X and y:

```
>>> X, y = dataset[:, 0:-1], dataset[:, -1].astype(int)
>>> print(X.shape, y.shape)
(2148, 8) (2148,)
>>>
```

To get X we index all the examples of the dataset and all the columns of the dataset except for the last column which contains the correct answer. To get y we index all the examples of the dataset and the last column of the dataset. At the same time, we convert y into integers by using the method, **.astype()** since there are no floats in y When we print the shapes of X and y, the number of examples/rows, 2148 is correct and the number of input features/columns for X, 8 is correct. However, there are a number of output features/columns for y. This is because the data only has one column, resulting in a column vector instead of a matrix. To make y a matrix we need to add a dimension:

```
>>> y = np.expand_dims(y, axis=-1)
>>> print(y.shape)
(2148, 1)
>>>
```

We can do this by using **np.expand_dims()** where the first argument is the NumPy array to add a dimension to and the keyword argument, axis is -1, which means we want a new dimension to the end of the array. The shape, the number of columns is now what we expected, 1 output feature.

Now that we have went through loading the data step-by-step, we can now convert this into a script, here is the code:

Now that the process of reading the the data file is explained, the print statements are removed except for checking the shapes of X and y to make sure there are no bugs.
The output from the terminal should be: (2148, 8) (2148, 1)
Now we have all the tools we need to start training a binary classifier.

```python
import numpy as np    # version 1.22.2
import pandas as pd    # version 1.4.0
np.random.seed(0)    # For repeatability

dataset = pd.read_excel("Pistachio_16_Features_Dataset.xlsx")
to_drop = ['ECCENTRICITY', 'EQDIASQ', 'SOLIDITY', 'CONVEX_AREA',
           'EXTENT', 'COMPACTNESS', 'SHAPEFACTOR_3', 'SHAPEFACTOR_4']

dataset = dataset.drop(to_drop, axis=1)

# Convert the Classes to 0s and 1s
classes = {"Kirmizi_Pistachio": 0, "Siit_Pistachio": 1}
dataset['Class'] = dataset['Class'].replace(classes)

# Split the dataset into X and y
dataset = dataset.to_numpy(dtype=float)
X, y = dataset[:, 0:-1], dataset[:, -1].astype(int)
y = np.expand_dims(y, axis=-1)
print(X.shape, y.shape)
```

This section is going to focus on the code to start training, here is the code:

```python
import matplotlib.pyplot as plt    # version 3.4.0
import numpy as np    # version 1.22.2
import pandas as pd    # version 1.4.0
np.random.seed(0)    # For repeatability

class Dense_Layer:…

class ReLU_Activation:…

class Sigmoid_Activation:…

class BCE_Cost:…

class SGD_Optimizer:…
```

First let's import matplotlib to plot the history of the cost after training and all the tools related to building the model before we read the data. Copy and paste the dense layer class, relu activation class for the hidden layers, sigmoid activation class for the output layer, the cost function, and the SGD optimizer. The code in the classes are hidden to save space.

Now let's standardize X, input to the model, before we define the model:

```python
def standardize(X):
    mean_of_zero = X - np.mean(X, axis=0)    # bringing data to have mean of 0
    X = mean_of_zero / np.std(X, axis=0)    # brinding data to have standard deviation of 1
    return X

X = standardize(X)    # standardize the data
```

The reason why we chose standardization over normalization is because we are dealing with pistachios, the physical characteristics of them should be pretty similar to each other in a range of values so their relationships would closely match the result of standardization. This piece of code comes after we split the X and y from the dataset.

Now let's define the model, we learned from the previous unit that a neural network with multiple hidden layers and hidden units is beneficial so let's start with 3 hidden layers, each with 64 hidden units:

```
class Neural_Network:
    def __init__(self, n_inputs, n_hidden, n_outputs):    # n_hidden is the number of hidden neurons
        self.hidden_layer1 = Dense_Layer(n_inputs, n_hidden)
        self.activation1 = ReLU_Activation()
        self.hidden_layer2 = Dense_Layer(n_hidden, n_hidden)
        self.activation2 = ReLU_Activation()
        self.hidden_layer3 = Dense_Layer(n_hidden, n_hidden)
        self.activation3 = ReLU_Activation()
        self.output_layer = Dense_Layer(n_hidden, n_outputs)
        self.output_activation = Sigmoid_Activation()
        self.cost_function = BCE_Cost()

        self.trainable_layers = [self.hidden_layer1, self.hidden_layer2,
                                 self.hidden_layer3, self.output_layer]
```

The main difference is that in between the initialization of the output layer and initialization of the cost function, we have to squish in the initialization of the output activation function. Here is the forward method:

```
def forward(self, inputs, y_true):
    self.hidden_layer1.forward(inputs)
    self.activation1.forward(self.hidden_layer1.outputs)
    self.hidden_layer2.forward(self.activation1.outputs)
    self.activation2.forward(self.hidden_layer2.outputs)
    self.hidden_layer3.forward(self.activation2.outputs)
    self.activation3.forward(self.hidden_layer3.outputs)
    self.output_layer.forward(self.activation3.outputs)
    self.output_activation.forward(self.output_layer.outputs)
    self.cost = self.cost_function.forward(self.output_activation.outputs, y_true)
```

The main difference is that in between the forward method of the output layer and the forward method of the cost function, we have to squish in the forward method of the output activation that takes in the output of the output layer as input and modify the forward method of the cost function to take in the output of the output activation instead of the output of the output layer itself. Here is the backward method:

```
def backward(self, y_true):
    self.cost_function.backward(self.output_activation.outputs, y_true)
    self.output_activation.backward(self.cost_function.dinputs)
    self.output_layer.backward(self.output_activation.dinputs)
    self.activation3.backward(self.output_layer.dinputs)
    self.hidden_layer3.backward(self.activation3.dinputs)
    self.activation2.backward(self.hidden_layer3.dinputs)
    self.hidden_layer2.backward(self.activation2.dinputs)
    self.activation1.backward(self.hidden_layer2.dinputs)
    self.hidden_layer1.backward(self.activation1.dinputs)
```

The main difference is that in between the backward method of the cost function and the backward method of the cost function, we have to squish in the backward method of the output activation that takes in the ***dinputs*** of the cost function as input and modify the backward method of the output layer to take in the ***dinputs*** of the output

activation instead of the ***dinputs*** of the cost function itself. Before we continue, let's define a function to get the accuracy (percentage of examples the model predicts correctly) of the model:

```
def get_accuracy(y_pred, y_true):
    predicted_classes = y_pred.copy()
    predicted_classes[y_pred < 0.5] = 0
    predicted_classes[y_pred >= 0.5] = 1
    return np.mean(predicted_classes == y_true) * 100
```

To start, we make a copy of the model's predicted probabilities so that when we are converting the predicted probabilities into a discrete class, Kirmizi (0) or Siit (1), we do not accidentally modify the original model's predicted probabilities.

Remember that a Kirmizi pistachio was converted to a value of 0 while a Siit pistachio was converted to a value of 1 and that the model predicts the probability that the pistachio is a Siit. As a result we can index all the examples that the model predicted with a probability less than 50% that the pistachio was a Siit, then we convert those predictions to a Kirmizi pistachio. Next we can index all the examples that the model predicted with a probability greater than or equal to 50% that the pistachio was a Siit, then we convert those predictions to a Siit pistachio.

After that, by *y_pred == y_true*, NumPy generates a boolean array where a value of 1 means that the model's prediction was equal to the correct answer so the model is correct while a value of 0 means that the model's prediction was not equal to the correct answer so the model is incorrect. As a result, when we take the average of the array by ***np.mean()***, all the values of the array are added together to get the total number of examples the model predicted correctly and then divided by the total number of examples the model predicted on. To convert the result into a percentage to represent the accuracy of the model, we multiply by 100. Now here is the initializations:

```
# Initialize the model and the optimizer
model = Neural_Network(8, 16, 1)     # 8 input features, 16 hidden units, 1 output feature
optimizer = SGD_Optimizer(0.01)    # learning rate is 0.01
cost_history, accuracy_history = [], []
```

The model has 8 input features, 16 hidden units for each of its 3 hidden layers, and 1 output feature. The optimizer we will use is SGD with a learning rate of 0.01. At the same time during the training loop we are going to keep track of the cost and accuracy to make sure the model is making progress. Here is the training loop:

```
for i in range(100):
    model.forward(X, y)
    cost_history.append(model.cost)
    accuracy_history.append(get_accuracy(model.output_activation.outputs, y))

    if i % 10 == 0:
        print(f"Cost: {cost_history[-1]} - Accuracy: {accuracy_history[-1]}%")

    model.backward(y)
    for layer in model.trainable_layers:
        optimizer.update_params(layer)
```

We are going to train for 100 epochs and every epoch we do a forward pass, update the history of the cost and accuracy.

Every 10 epochs we will print an update to the cost and accuracy of the model. Lastly, we do a backward pass so that the optimizer can update the parameters of each layer with parameters.

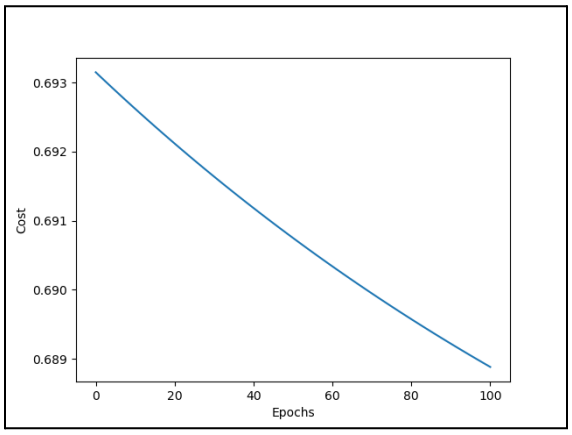
```
# Check Ending Cost and Accuracy
model.forward(X, y)
cost_history.append(model.cost)
accuracy_history.append(get_accuracy(model.output_activation.outputs, y))
print(f"Final Cost: {cost_history[-1]} - Final Accuracy: {accuracy_history[-1]}%")
```

At the end we check the ending cost and accuracy of the model and then we can create a graph of the cost history and accuracy history:

```
# Cost History Graph
fig = plt.figure()
plt.plot(cost_history)
plt.xlabel("Epochs")
plt.ylabel("Cost")
fig.savefig("cost.png")
plt.close()
```

```
# Accuracy History Graph
fig = plt.figure()
plt.plot(accuracy_history)
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
fig.savefig("accuracy.png")
plt.close()
```

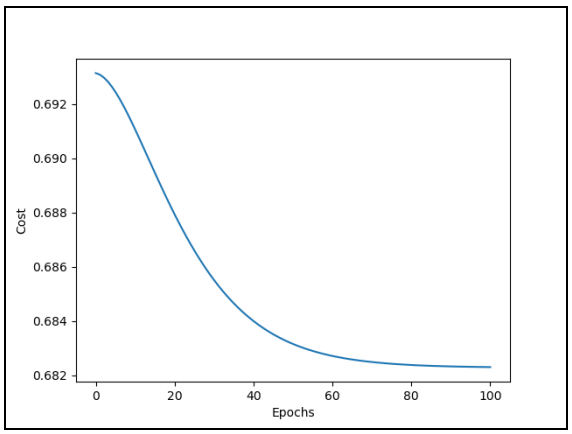
The graphs of the cost history and accuracy history are saved to *cost.png* and *accuracy.png*, respectively. Time to run the script! Hopefully we get some good initial results.



The cost seems to be going down steadily but very slow in the span of 100 epochs. To speed up the model's progress, allowing the cost to decrease more, let's try adding in momentum to the optimizer.

We are going to use a mu of 0.9 to add momentum to the optimizer. Here is the new code:

```
# Initialize the model and the optimizer
model = Neural_Network(8, 16, 1)     # 8 input features, 16 hidden units, 1 output feature
optimizer = SGD_Optimizer(0.01, mu=0.9)    # learning rate is 0.01, mu is 0.9
cost_history, accuracy_history = [], []
```

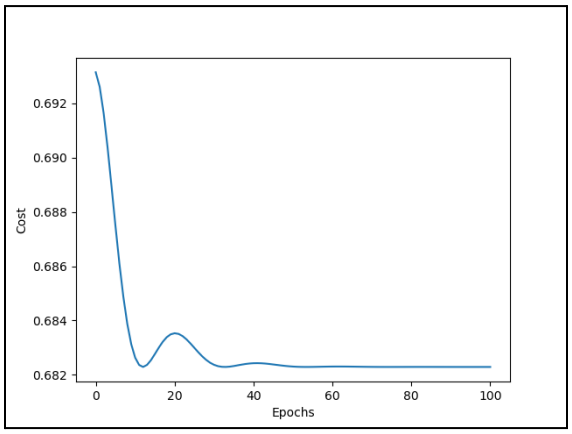


Remember to also save the cost and accuracy graphs to different filenames, *cost2.png* and *accuracy2.png*, respectively.

Now the cost has dropped by 0.01 but we see that progress is slowing down. Is the model smart now? Has it converged? Unfortunately this is not the case. We can see this since the accuracy of the model after training is only ~57%. Very low accuracy since this translates to a F grade.

The reason might be that the learning rate is too low, decreasing the updates parameters, and thus the cost decreases very slowly. Let's increase the learning rate from 0.01 to 0.1. Here is the code (save the cost and accuracy graphs to *cost3.png* and *accuracy3.png*, respectively):

```
# Initialize the model and the optimizer
model = Neural_Network(8, 16, 1)     # 8 input features, 16 hidden units, 1 output feature
optimizer = SGD_Optimizer(0.1, mu=0.9)    # learning rate is 0.1, mu is 0.9
cost_history, accuracy_history = [], []
```



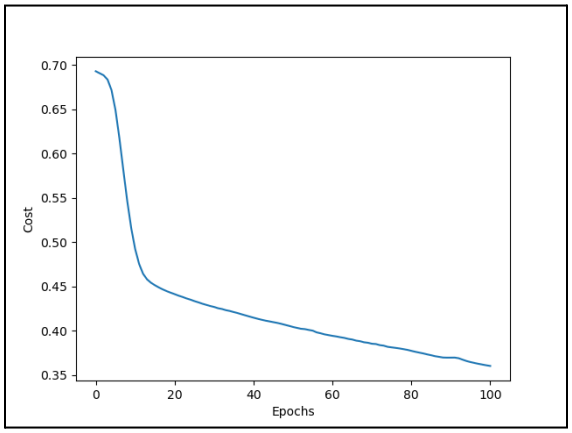
What is this? Why is the cost jumping up and down? This means that the learning rate is too high for some parameters, so those parameters may be jumping up and down, affecting the other parameters that result in this behavior of the cost. Now we should switch the optimizer to RMSProp to eliminate this issue.

To do that, replace the *SGD_Optimizer* class with the *RMSProp_Optimizer* class. Here is the code (*cost4.png* and *accuracy4.png* are the new file locations):

```python
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",
        # so 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
```

```python
# Initialize the model and the optimizer
model = Neural_Network(8, 16, 1)     # 8 input features, 16 hidden units, 1 output feature
optimizer = RMSProp_Optimizer(0.01)    # learning rate of 0.01
cost_history, accuracy_history = [], []
```

From the cost graph above we can easily see that the learning rate of 0.1 is too high so we decide to choose a learning rate of 0.01 instead.



We see that the cost is now steadily decreasing when we use the RMSProp optimizer over the SGD optimizer. However it is worth noting that the cost has jumped up a little around 50 epochs and 80 epochs. This signifies that we now need to add a learning rate decayer before training for more epochs. Copy and paste the *Learning_Rate_Decayer* class right after the definition of the *RMSProp_Class* and right before reading the data file.

Here is the code (*cost5.png* and *accuracy5.png* are the new file locations):

```python
# Initialize the model and the optimizer
model = Neural_Network(8, 16, 1)     # 8 input features, 16 hidden units, 1 output feature
optimizer = RMSProp_Optimizer(0.01)    # learning rate of 0.01
decayer = Learning_Rate_Decayer(optimizer, 0.01)
cost_history, accuracy_history = [], []
```

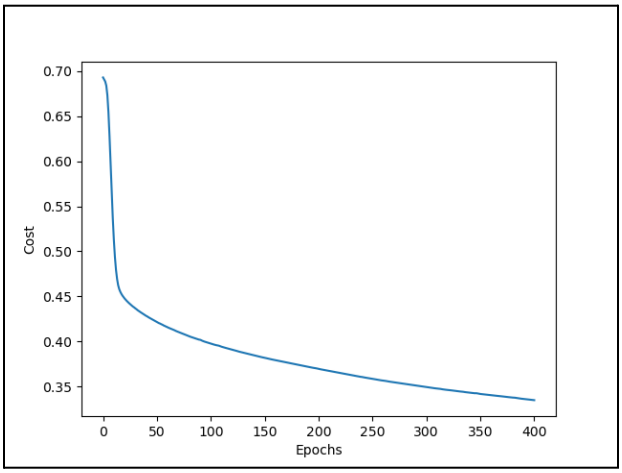
Recall that when we use a learning rate decay, the learning rate as a function of the number of epoch is defined as $\alpha(t) = \frac{a_0}{1+kt}$ where α is the new learning rate, α_0 is the initial learning rate (0.01), t is the number of epochs, and k is the decay rate (0.02). As a result, the denominator increases by 1 every time the product of k and t increases

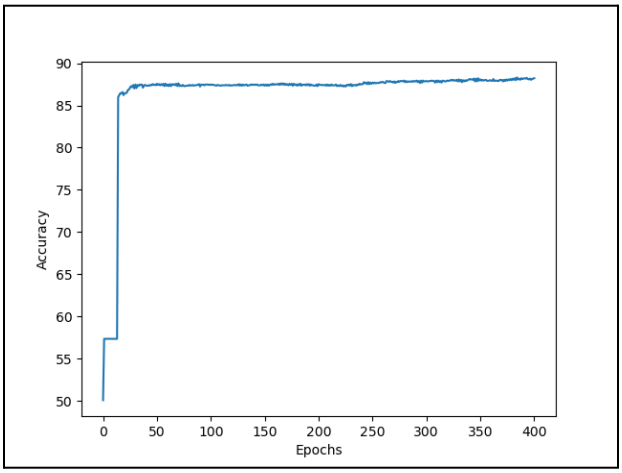
by 1, and by setting k equal to 0.02, we have the denominator to increase by 1 every 50 epochs. To update the learning rate every epoch, we call the *.update_learning_rate()* method of the decayer:

```
for i in range(400):
    model.forward(X, y)
    cost_history.append(model.cost)
    accuracy_history.append(get_accuracy(model.output_activation.outputs, y))

    if i % 40 == 0:
        print(f"Cost: {cost_history[-1]} - Accuracy: {accuracy_history[-1]}%")

    model.backward(y)
    for layer in model.trainable_layers:
        optimizer.update_params(layer)

    decayer.update_learning_rate()
```

Now we train the model for 400 epochs, and give a progress check every 40 epochs. Only after we update the parameters, we update the learning rate.

The cost of the model is still continuously dropping while the accuracy of the model is jumping up and down, trying very hard to pass the 90% accuracy mark:





The reason why the accuracy is jumping up and down but the cost is continuously decreasing is because when we update the parameters, we calculate their derivatives/slopes relative to the cost function rather than the accuracy function. As a result, the updates to the parameters directly decrease the cost but does not directly increase the accuracy. The accuracy is only influenced on whether the model predicted above a 50% probability or less.

Before we keep training the model, in the next section we are going to check the **reliability** of the model.

Checking the Model's Reliability

Okay so what does it mean for us to check the model's reliability? This means we test the model on examples it has not **"seen"** before (examples the model did not use while training). If the model predicts with pretty good accuracy on those examples, the model is **reliable** since when we deploy the model in the real world, almost all if not all the examples the model predicts on are examples the model has not seen before.

How do we check the model's reliability? Well first we have to split the whole dataset into a **Training Set** and a **Test Set**. The training set will be used by the model during **training** and after training, to **test** the model's ability on examples it has not seen before, we are going to use the test set. Here is the code:

```
# Convert the Classes to 0s and 1s
classes = {"Kirmizi_Pistachio": 0, "Siit_Pistachio": 1}
dataset['Class'] = dataset['Class'].replace(classes)
dataset = dataset.to_numpy(dtype=float)
np.random.shuffle(dataset)

# Split the dataset into training and testing sets
n = int(len(dataset) * 0.8)    # use 80% of the dataset for training and 20% for testing
X_train, y_train = dataset[0:n, 0:-1], dataset[0:n, -1].astype(int)
X_test, y_test = dataset[n:, 0:-1], dataset[n:, -1].astype(int)
y_train, y_test = np.expand_dims(y_train, axis=-1), np.expand_dims(y_test, axis=-1)
print(X_train.shape, y_train.shape, X_test.shape, y_test.shape)
```

After we convert the correct answers to 0s and 1s and into a NumPy array, we use *np.random.shuffle()* to randomly shuffle the order of the examples in the dataset in case the examples of the dataset are arranged in a way that adds bias into the model.

Next we have to decide how many examples we want in the training set and the test set. Usually the training set is made up of the majority of examples while the test set is made up of only a minority of examples. In this case, the variable *n* is the number of examples we are going to use in the training set, which is 80% of the total number of examples in the dataset. Index for the first and stop at the n^{th} example for the training set and index for the n^{th} example to the last example for the test set.

Lastly, remember to add in the second dimension for the y in both the training and test set to bring them into a matrix format. The print statements are to inform us the number of examples in the training set and test set, rather than a ratio (80%). Now that there is a training and testing set. How do we standardize the data?

```
def standardize(X, mean=None, std=None):
    if mean is None and std is None:    # for the training set
        mean, std = np.mean(X, axis=0), np.std(X, axis=0)
    standardized = (X - mean) / std
    return standardized, (mean, std)

X_train, (mean, std) = standardize(X_train)
X_test, _ = standardize(X_test, mean, std)
```

Remember that we want to test the model on data it has not seen before.

As a result, we cannot use the mean or standard deviation of the test set to standardize the data, otherwise we would leak some information about the test set to the model.

We can only use the mean and standard deviation of the training set to standardize both the training and test set. We still have to standardize the test set because if we did not, the input data would not be in the same ranges the model is used to during training.

So to standardize the data, we have to modify the *standardize()* function. The first argument is the input data and then there are two keyword arguments that are for standardizing the test set using the statistics of the training set. As a result for the training set we only pass in the input data, store the mean and standard deviation to *mean* and *std*, respectively.

For both the training and test set we standardize them by the *mean* and *std* (calculated for the training set, from the two keyword arguments for the test set). Lastly, we return the standardized data along with a tuple containing *mean* and *std*. We only need that tuple when we standardize the training set so we can use those as input to the function when we standardize the test set. Since we do not need the mean and standard deviation again when we standardize the test set, we set the tuple to an underscore.

Now since we changed the variable name containing the training data from *X* and *y* to *X_train* and *y_train*, respectively, in the training loop we have to replace all references to *X* and *y* to *X_train* and *y_train*, respectively. Out of the training loop, we need to test the model on the testing set:

```
# Check Ending Cost and Accuracy
model.forward(X_train, y_train)
cost_history.append(model.cost)
accuracy_history.append(get_accuracy(model.output_activation.outputs, y_train))
print(f"Final Cost: {cost_history[-1]} - Final Accuracy: {accuracy_history[-1]}%")

# Test the model on the test set
model.forward(X_test, y_test)
accuracy = get_accuracy(model.output_activation.outputs, y_test)
print(f"Test Set Results ~ Cost: {model.cost} - Accuracy: {accuracy}%")
```

Any original references to *X* and *y* are replaced by *X_train* and *y_train*, respectively. Next we test the model on the test set by using the *.forward()* method with the input data as *X_test*

and the correct answers as *y_test* and then get the accuracy of the model by using the outputs of the output activation function and correct answers. Lastly, we print the cost and accuracy of the model on the test set. Here are the first and last two lines of the terminal output:
(1718, 8) (1718, 1) (430, 8) (430, 1)
Final Cost: 0.2858925488065632 - Final Accuracy: 88.64959254947613%
Test Set Results ~ Cost: 0.36376594858298006 - Accuracy: 83.72093023255815%

We see that we train the model with 1718 examples and test the model with 430 examples by the 1st line. By comparing the test set results with the training set results we see that they are pretty similar but the results of the test set is lagging behind. The behavior of test set results, where the model has not seen some examples before is normal but how do we close this gap to ensure better results when we deploy the model into the real world?

To close this gap we use **regularization** and **dropout** but more on that later. First, we need to understand what happens to the model's performance during training. At the beginning, the model should be equally good/bad in both the training and test set since the model is initialized with random parameters. Later however, sometime during training the model, its performance on the training set starts to deviate (get better) than its performance on the test set.

The question now is how do we know **when to stop training** so that the model's performance on "unseen" examples is not too horrible compared to the training examples? One solution is to calculate the model's performance on the test set during training but this ultimately results in adding bias since the model would be **indirectly trained** by examples in the test set. This is due to the fact that we are using the test set to find the optimal time to stop training so that the model's performance is maximized on the test set.

As a result, we have to create a **validation set** in addition to the training and testing sets. This **validation set** will ultimately answer the question: **when to stop training** without exposing any information about the test set to the model until after training.

Here is the code (80% of the data is used for training the model, 10% of the data is used for the validation set, and the last 10% is used for the test set):

```
train = int(len(dataset) * 0.8)    # use 80% of the dataset for training
val = int(len(dataset) * 0.1)    # use 10% of the dataset for validation

# Split the dataset into training, validation and test sets
X_train, y_train = dataset[0:train, 0:-1], dataset[0:train, -1].astype(int)
X_val, y_val = dataset[train:train + val, 0:-1], dataset[train:train + val, -1].astype(int)
X_test, y_test = dataset[train + val:, 0:-1], dataset[train + val:, -1].astype(int)

y_train = np.expand_dims(y_train, axis=-1)
y_val = np.expand_dims(y_val, axis=-1)
y_test = np.expand_dims(y_test, axis=-1)

print(len(X_train), len(X_val), len(X_test))    # number of examples in each set
```

The variables *train* and *val* contain the number of examples we use for the training set and validation set, respectively. Now the training set is indexed by starting with the first example and stopping at *train*. The validation set is indexed by starting at *train* and stopping at (*train + val*). The test set is indexed by starting at (*train + val*) to the end of the dataset. Lastly, we print the number of examples in each set of examples.

```
X_train, (mean, std) = standardize(X_train)
X_val, _ = standardize(X_val, mean, std)
X_test, _ = standardize(X_test, mean, std)
```

Now we normalize the validation set the same way we normalize the test set so that we do not leak potential information about the validation set.

```
# Initialize the model and the optimizer
model = Neural_Network(8, 16, 1)     # 8 input features, 16 hidden units, 1 output feature
optimizer = RMSProp_Optimizer(0.01)    # learning rate of 0.01
decayer = Learning_Rate_Decayer(optimizer, 0.02)
cost_history, accuracy_history = [], []
val_cost, val_accuracy = [], []

for i in range(400):
    # Get Validation Cost and Accuracy
    model.forward(X_val, y_val)
    val_cost.append(model.cost)
    val_accuracy.append(get_accuracy(model.output_activation.outputs, y_val))
```

To calculate the model's performance on the validation set during training, we initialize two more empty lists, *val_cost* and *val_accuracy* to store the history of the cost of the model on the validation set and the history of the accuracy of the model on validation set, respectively. Next, in the very beginning of each epoch we first store the model's performance on the validation set before continuing on with the regular process in the training loop. To display the model's performance on the validation set during training:

```
if i % 40 == 0:
    print(f"Cost: {cost_history[-1]} - Accuracy: {accuracy_history[-1]}%"+
          f" - Validation Cost: {val_cost[-1]} - Validation Accuracy: {val_accuracy[-1]}%")
```

After training, we can check the model's performance on three different sets of examples now:

```
# Check Ending Cost and Accuracy
model.forward(X_train, y_train)
cost_history.append(model.cost)
accuracy_history.append(get_accuracy(model.output_activation.outputs, y_train))
print(f"Final Cost: {cost_history[-1]} - Final Accuracy: {accuracy_history[-1]}%")

model.forward(X_val, y_val)
val_cost.append(model.cost)
val_accuracy.append(get_accuracy(model.output_activation.outputs, y_val))
print(f"Final Val Cost: {val_cost[-1]} - Final Val Accuracy: {val_accuracy[-1]}%")

# Test the model on the test set
model.forward(X_test, y_test)
accuracy = get_accuracy(model.output_activation.outputs, y_test)
print(f"Test Set Results ~ Cost: {model.cost} - Accuracy: {accuracy}%")
```

Since we have been storing the history of the training and validation sets, we should continue to add to the history of these sets of examples. Lastly, we can graph the history of the model's cost on the training and validation sets together. The history of the model's accuracy on the training and validation sets can also be graphed together:

```
# Cost History Graph
fig = plt.figure()
plt.plot(cost_history, label="Training Cost")
plt.plot(val_cost, label="Validation Cost")
plt.xlabel("Epochs")
plt.ylabel("Cost")
plt.legend()
fig.savefig("cost2.png")
plt.close()

# Accuracy History Graph
fig = plt.figure()
plt.plot(accuracy_history, label="Training Accuracy")
plt.plot(val_accuracy, label="Validation Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.legend()
fig.savefig("accuracy2.png")
plt.close()
```

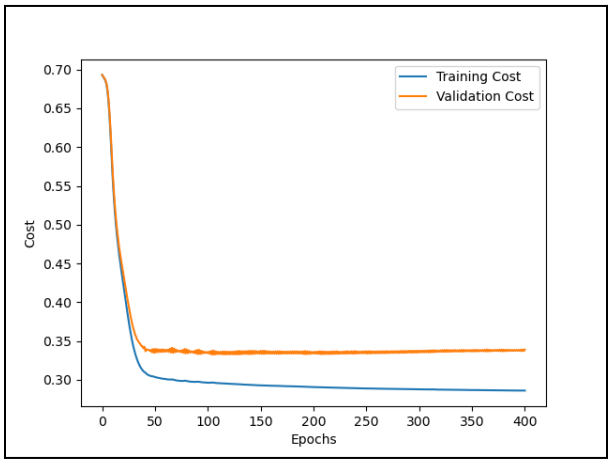
All we have to do is to use *plt.plot()* and pass in the data we would like to plot but we also have to pass in the keyword argument label which gives a name to each set of data we plot. To differentiate the different colors of data we add in *plt.legend()*, which adds a legend to the figure that corresponds the different colors to the name of the data we plotted. Everything else stays the same.

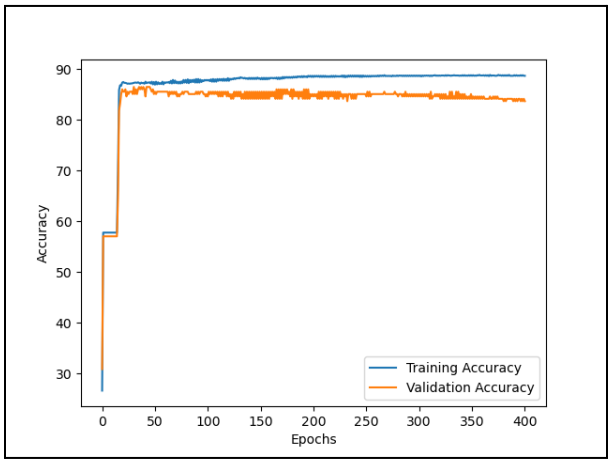
Now here is the output of the 1st and last three lines from the terminal:

1718 214 216
Final Cost: 0.2858925488065632 - Final Accuracy: 88.64959254947613%
Final Val Cost: 0.33870627643723356 - Final Val Accuracy: 83.64485981308411%
Test Set Results ~ Cost: 0.38859358672737715 - Accuracy: 83.79629629629629%

We still train with 1718 examples but now the test set is made up of 216 examples while the validation set is made up of 214 examples.

Here are the graphs of the history of costs and accuracy that we have created:

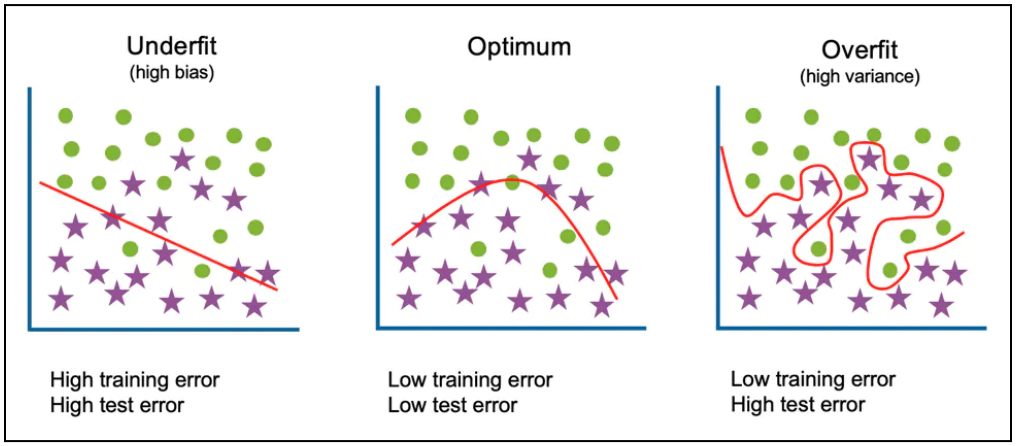




As the model's **cost** slowly **decreases** on the **training set**, the model's **cost** is slowly **increasing** on the **validation set**. As the model's **accuracy** slowly **increases** on the **training set**, the model's **accuracy** is slowly **decreasing** on the **validation set**.

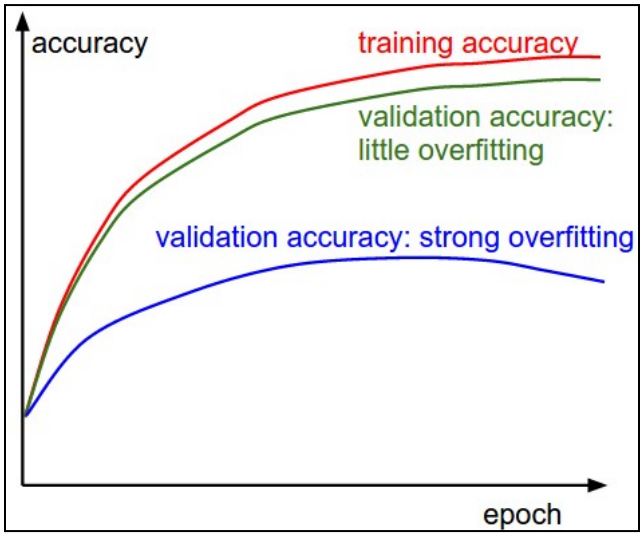
The obvious takeaway is that as that model performs better on the training set, the model starts to perform worse on the validation set. This means that the model is **overfitting**. **Overfitting** is when the model begins to use its **parameters** to **memorize** the characteristics of the training set so that it begins to **trade off generalizations** (these are beneficial to the model on examples it has not seen) for **better training costs and accuracies or perfection** (through the use of parameters for **memorization**).

To better understand how the model overfits by using its parameters to memorize characteristics of the training set, here is a diagram from IBM:



When we overfit, the model starts to aim for **perfection** rather than finding the general trend (**generalizations**). The optimum has a good balance of classifying good vs. wrong where the model is not predicting too many examples wrong since it has made **generalizations** about the data. Underfitting is somewhat like having a really small model. For example, one hidden layer which would lead to the model having low performance on any set of examples. Our current situation is the one on the right; the model is overfitting the data. As a result, we need to move the model one to the left.

Here is a plot from [Stanford University's CS231n Course Notes]:



Comparing our accuracy graph to this, we can see that the good news is that the model is not **strongly overfitting** since its validation accuracy difference from the training accuracy is not as large. However, the model is not **overfitting a little** since currently our validation accuracy is beginning to trend **slightly** downwards. While training a model, it is best to aim for the model to only **overfit a little** or less for a higher performing model on unseen data.

Despite the model becoming a bad AI, we can prevent the model from overfitting through the use of **regularization** and/or **dropout**. This way we can reach the optimal model condition. We will start with **regularization** in the next section. See ya there!!

Regularization

So to prevent the model from **overfitting** or just memorization, we have to understand how the model uses its parameters to memorize. Memorization begins when the **magnitudes** of the weights and biases become very large. This means that the absolute value of the weights and biases are very high (very positive/negative values).

One way to prevent this is to **penalize** the model for having large magnitudes of parameters. What does it mean to **penalize**? Well if the model predicts a probability very far from the correct answer, we **penalize** it through the cost function. As a result, we can add the **penalization** to the cost function, and thus increase the cost of the model. The result would be: Cost = Original Cost + Parameter Penalizations.

Now when we train the model, we want the overall cost to decrease. This can only be done by decreasing the Original Cost while decreasing the Parameter Penalizations. The only way to decrease the parameter penalizations is to **lower** the **magnitude** of the parameters. Hence, we force the model to have better predictions while having low magnitude parameters. Inevitably, the model will be unable to **memorize** since this would increase the cost, but the model can now **generalize** more.

So how exactly can penalize the parameters to add Parameter Penalizations to the cost function? Well, there are 2 different ways, L_1 Regularization and L_2 Regularization. Currently, L_1 Regularization would be used in combination with L_2 Regularization mostly because the penalties by L_1 Regularization are not strong enough so it is barely used alone. Here are the calculations done by L_1 Regularization & L_2 Regularization:

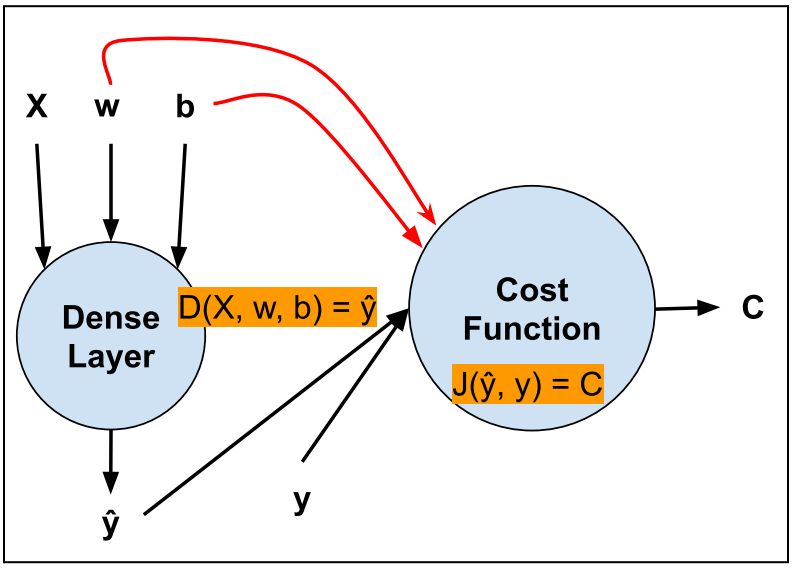
L_1 Regularization	L_2 Regularization
$\lambda \times \lvert p \rvert$	$\lambda \times p^2$

The λ is the Greek lowercase letter lambda which represents the **regularization strength**. The p is the parameter value. For each parameter, based on the type of regularization method, we calculate the penalty and add it to the cost.

L_1 Regularization only uses the absolute value of the parameter. L_2 Regularization has a very strong regularization strength since the parameter value is squared, resulting in a huge penalty for larger and larger parameter magnitudes.

The λ is another **hyper-parameter**, where its value is decided by the human. Usually λ is a value between 0 and 1. A lambda of 0 where λ = 0 means that there is no regularization used since multiplying anything parameter value by 0 is 0, adding nothing to the overall cost.

Since L_1 Regularization is not very strong, we are going to skip the backward method for L_1 Regularization and move on to the backward method for L_2 Regularization. The difference with regularization is that we do not need to use the chain rule:



With regularization the parameters are directly inputted into the cost function so the regularization derivative does not require chain rule. The derivative of the parameters would be the **sum** of the original derivative (calculated by the chain rule) and the regularization derivative. Now let's calculate the regularization derivative:

$$\frac{dL_2(p)}{dp} = \frac{L_2(p+\epsilon) - L_2(p)}{(p+\epsilon) - p} = \frac{[\lambda * (p+\epsilon)^2] - (\lambda * p^2)}{(p-p) + \epsilon}$$

L_2 is the function that takes in the parameter, p. To calculate the derivative we use the $\frac{f(x+\epsilon) - f(x)}{(x+\epsilon) - x}$ format. We substitute $[\lambda * (p + \varepsilon)^2]$ in place of $L_2(p + \varepsilon)$ and $(\lambda * p^2)$ in place of $L_2(p)$. In the denominator we group the like terms.

$$= \frac{[\lambda * (p^2 + 2p\epsilon + \epsilon^2)] - \lambda p^2}{\epsilon} = \frac{\lambda * [p^2 + 2p\epsilon + \epsilon^2 - p^2]}{\epsilon}$$

Knowing that $(a + b)^2$ is equal to $a^2 + 2ab + c^2$, we expand the numerator's first term (y_2): $[\lambda * (p + \varepsilon)^2]$ by having a = p, b = ε, and we distribute the negative sign in front of the

numerator's second term (y_1): ($\lambda * p^2$). In the denominator, the like terms are combined. Lastly, from both terms in the numerator we factor out the lambda, λ.

$$= \frac{\lambda * [(p^2 - p^2) + (2p\epsilon + \epsilon^2)]}{\epsilon} = \frac{\lambda\epsilon * (2p + \epsilon)}{\epsilon} = \lambda * (2p + \epsilon) = \lambda * 2p$$

Next we group the like terms in the numerator and combine them. (p^2 - p^2) cancel out so we are left with λ * ($2p\varepsilon + \varepsilon^2$). We can factor out the ε from both terms to get $\lambda\varepsilon$ * ($2p + \varepsilon$). The ε in the numerator and denominator cancel out leaving λ * ($2p + \varepsilon$). Lastly, we can substitute 0 in place of ε, since ε is a tiny positive value close to 0, which makes the 2nd term: ε, negligible. The result is λ * 2p or 2λp. Here is a PDF of steps that got us the analytical derivative.

Now that we have covered both the forward method & backward method of L_2 Regularization, let's implement this in code:

```
class Dense_Layer:
    def __init__(self, n_inputs, n_neurons, regularization_lambda=0):
        self.weights = 0.01 * np.random.randn(n_inputs, n_neurons)
        self.biases = np.zeros([1, n_neurons])
        self.regularization_lambda = regularization_lambda
```

Since the regularization derivative is directly added to the original derivative, we can modify the *Dense_Layer* to include the regularization derivative in the backward method. We modified the *__init__* method to include the keyword argument *regularization_lambda* which is 0 by default (no regularization) and store it into *self.regularization_lambda*.
Next, only if *self.regularization_lambda* is greater than 0, which means we regularize, we add the regularization derivative to *self.dweights* & *self.dbiases*. Nothing changes for *self.dinputs* since we are **regulating** the parameters, not the inputs.

```
def backward(self, dvalues):
    self.dweights = np.dot(self.inputs.T, dvalues)
    self.dbiases = np.sum(dvalues, axis=0, keepdims=True)
    self.dinputs = np.dot(dvalues, self.weights.T)

    if self.regularization_lambda > 0:
        self.dweights += 2 * self.regularization_lambda * self.weights
        self.dbiases += 2 * self.regularization_lambda * self.biases
```

Note: We use *regularization_lambda* because *lambda* is an expression in Python.
Now let's redefine our dense layers in the model to add regularization:

```
class Neural_Network:
    def __init__(self, n_inputs, n_hidden, n_outputs):    # n_hidden is the number of hidden neurons
        self.hidden_layer1 = Dense_Layer(n_inputs, n_hidden, regularization_lambda=0.01)
        self.activation1 = ReLU_Activation()
        self.hidden_layer2 = Dense_Layer(n_hidden, n_hidden, regularization_lambda=0.01)
        self.activation2 = ReLU_Activation()
        self.hidden_layer3 = Dense_Layer(n_hidden, n_hidden,  regularization_lambda=0.01)
        self.activation3 = ReLU_Activation()
        self.output_layer = Dense_Layer(n_hidden, n_outputs, regularization_lambda=0.01)
        self.output_activation = Sigmoid_Activation()
        self.cost_function = BCE_Cost()

        self.trainable_layers = [self.hidden_layer1, self.hidden_layer2,
                                 self.hidden_layer3, self.output_layer]
```

For each dense layer we added the keyword argument, using a regularization strength (lambda, λ) of 0.01. Now let's actually add the regularization penalty to the cost:

```
def get_regularization_penalty(layers, regularization_lambda):
    penalty = 0
    for layer in layers:
        penalty += np.sum(np.square(layer.weights))
        penalty += np.sum(np.square(layer.biases))
    return regularization_lambda * penalty
```

We create a function that takes in the layers that the model has parameters in and the regularization strength. We initialize the *penalty* to be 0. Next for each layer, we add the sum of the square of the weights/biases to the *penalty*. Lastly, we return the product of the regularization strength by the *penalty*.

Consider parameters *a*, *b*, and *c*, the regularization penalty would be (λ * a^2) + (λ * b^2) + (λ * c^2). When we factor the λ out, we get λ * (a^2 + b^2 + c^2). That explains why we can multiply the regularization strength after we sum up all the squares of the parameters.

In the training loop, where we run the model through the training set, we set *regularization_penalty* to the output of the function we just defined. *model.trainable_layers* gets us the layers that the model has parameters in and the regularization strength is 0.01. When we add the cost to the history list, we add the sum of the *model.cost* and *regularization_penalty*. We do not add the regularization penalty to the model's cost on the validation/test set because **regularization** only applies to **training** data as we want to **regularize** the model during **training**.

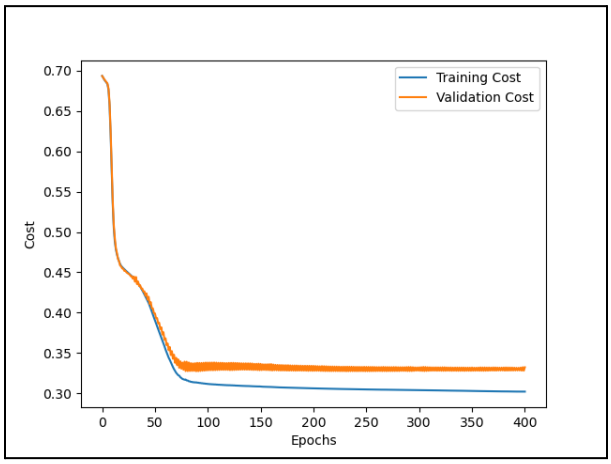
```
# Check Ending Cost and Accuracy
model.forward(X_train, y_train)
regularization_penalty = get_regularization_penalty(model.trainable_layers, 0.01)
cost_history.append(model.cost + regularization_penalty)
accuracy_history.append(get_accuracy(model.output_activation.outputs, y_train))
print(f"Final Cost: {cost_history[-1]} - Final Accuracy: {accuracy_history[-1]}%")
```

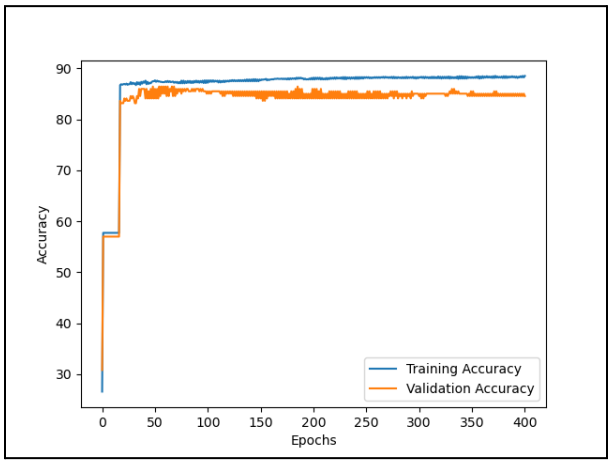
After the training loop, when we check the ending cost of the model, we also add the regularization penalty to the training set only. There is no change needed when we create the graphs so here is the last three lines of output from the terminal:
Final Cost: 0.6820153535842979 - Final Accuracy: 57.741559953434226%
Final Val Cost: 0.6832926560455054 - Final Val Accuracy: 57.009345794392516%
Test Set Results ~ Cost: 0.6901466805311924 - Accuracy: 54.629629629629626%

What happened? Why is the accuracy so low for all sets of data examples? The reason is because the regularization strength is too high. Recall that now the cost is now: Original Cost + Parameter Penalizations. When the regularization strength is too high, the model decides to prioritize lowering the cost from Parameter Penalizations over lowering the cost from the original cost function itself.

The penalty on the model for a parameter is $\lambda * p^2$ so to lower the cost from Parameter Penalizations, the model decides to break all the parameter values very close to 0. When this happens, the model is unable to actually learn anything as the parameter values hold no importance but to lower the cost from Parameter Penalizations.

As a result of the regularization strength being too high, we should lower the regularization strength to 0.001. So for every 0.01 value that represents the regularization strength, replace them with a 0.001. The code script itself that reflects those changes is available here. The last three lines of output from the terminal & the graphs of the cost history:
Final Cost: 0.3023853685859088 - Final Accuracy: 88.53317811408614%
Final Val Cost: 0.33251591823582904 - Final Val Accuracy: 84.57943925233646%
Test Set Results ~ Cost: 0.37077449431040477 - Accuracy: 85.18518518518519%





Now we see that as the model's cost slowly decreases on the training set, the model's cost is also slightly decreasing on the validation set. As the model's accuracy slowly increases on the training set, the model's accuracy is slightly increasing on the validation set. How about the final cost and accuracy of the model? Let's compare final results with regularization to without regularization:

Set of Examples	Cost		Accuracy	
	Without Reg.	With Reg.	Without Reg.	With Reg.
Training Set	~0.286	~0.302	~88.65%	~88.53%
Val. Set	~0.3387	~0.3325	~83.645%	~84.579%
Test Set	~0.3886	~0.3708	~83.796%	~85.185%

We see that although there is a slight increase in the model's cost on the training set and a slight decrease in the model's accuracy on the training set when we have regulation, there are decreases in the model's cost on the validation and test set and there are increases on the model's accuracy on the validation and test set.

This indicates that with regularization, we are **regulating** the model's parameters to aim for **generalization**, resulting in gains in the model's metrics on examples it has not seen (validation & test set), instead of **memorization**, resulting in negative gains in the model's metrics on the training set. Even though the model's metrics on the training set decreased, we would rather aim for **generalization** rather than the model performing extremely well on the training set.

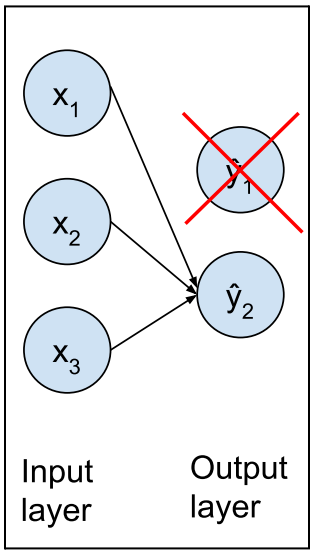
Regularization is not the only way to prevent the model from overfitting, we can also use another technique called **dropout** which we will cover in the next section. Instead of training this binary classifier all the way through until it is smart together, I will leave that task for you. This will allow you to experiment with the regularization strength hyperparameter while maybe adding more hidden units to gain more experience in training your own models. Also the gains in the validation and test set are pretty small so try increasing the regularization strength and see if that would help the model on those sets of examples.

Dropout

Dropout is actually a very simple concept in AI as it actually does what its name suggests **"drop out of ____"** maybe fill that blank with high school. Anyways, think of the neurons in the model as high school students and each layer in the model as a high school that the neurons attend. Each high school (layer) has its own **dropout rate**, just like in real life, and some neurons decide to **drop out**.

So essentially, this means that the neurons (students) leave the layer (high school). In real life, these students might not go back to high school. However in the context of AI, after some neurons (students) leave the layer (high school), we actually **force** the **dropped out** neurons to come back to high school (be part of the layer again).

The main point that should be taken away from the analogy is that in **dropout**, each layer will have a certain percentage of the neurons **"dropping out"** and that later we will need those **"dropped out"** neurons back. Dropout is only used in hidden layers and this visual shows why:

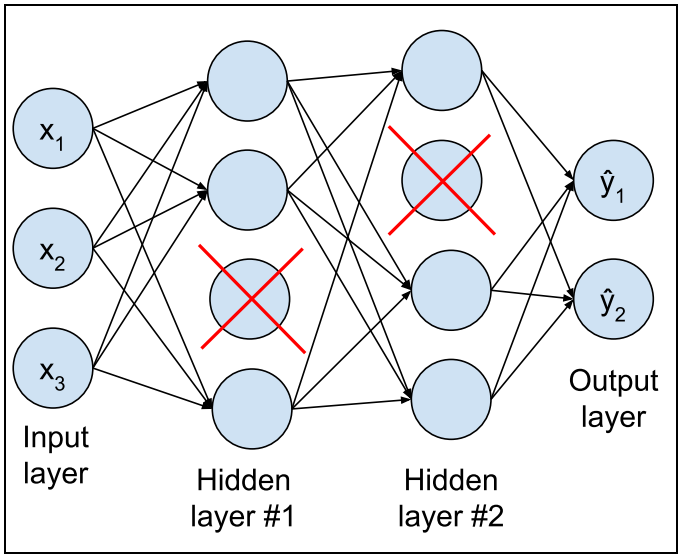


In the input layer, there are no neurons, only input features and thus we cannot **"drop out"** any neurons. In the output layer, we have neurons but we see that if we **"drop out"** any neurons, that neuron does not receive any input from the previous layers and thus cannot output anything. This is a problem since the output layer is where the model makes its predictions so **"dropping out"** any neurons in the output layer would result in no prediction from that particular output feature, creating problems.

At the moment, you probably have a really vague understanding of what it actually means to **"drop out"** neurons in the context of AI. Let's make this idea more concrete with the following diagram:

Here is a model with 3 input features, 4 hidden units for each of the 2 hidden layers, and 2 output features. Continuing with the analogy, let's say that Hidden layer #1 is High School A with a **dropout rate** of **25%** and Hidden layer #2 is High School B also with a **dropout rate** of **25%**. This means that for every 4 neurons, 1 neuron will drop out. The **dropped out** neurons are shown with a red X on top of them.

As you can see, the **dropped out** neurons receive no inputs and therefore do not output anything. The rest of the neurons in the model do not input anything into the **dropped out** neurons and do not receive any output from the **dropped out** neurons.



As a result, the dropped out neurons have no connection to the model or model's output anymore. So now that we understand what it means to **drop out** a neuron, how do we use dropout during training?

So on every epoch, we randomly select some neurons in the hidden layers to **drop out**. Now with the new model **missing** some neurons we do a forward pass to get the model's predictions. Next, we do a backward pass using the model that is **missing** some neurons. After that we **only update the neurons that have NOT been dropped out** which means that the **gradients** of the **dropped out neurons** are 0. Lastly, the dropped out neurons (high schoolers) are **forced** back into their layers (high school) and we move on to the next epoch.

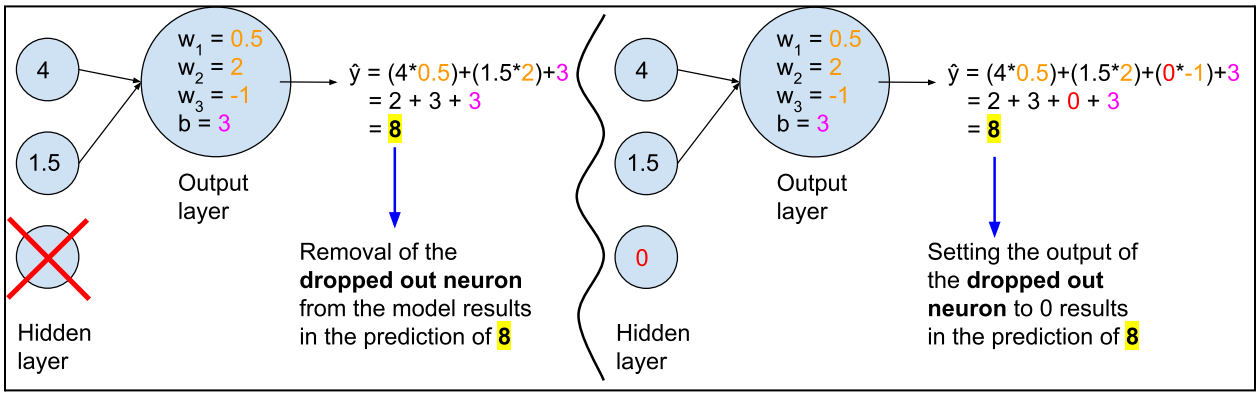
A note on the process in which we randomly select neurons to **dropout**. If a layer has a **50% dropout rate** that means that **each neuron in that layer** has a **50% chance** of being **dropped out**. To demonstrate, suppose we have 4 neurons (a, b, c, and d). For neuron a, we flip a **fair coin** (50% chance landing on heads, 50% chance landing on tails). If the coin lands on tails, neuron a is dropped out and we move on to the next neurons, flipping coins for each of them.

As a result, **it is possible** that no neurons are dropped out or all neurons are dropped out since each coin toss is **independent** of the previous coin toss. Instead of actually picking 50% of the neurons in the hidden layer to drop out, we **give each neuron the same chance** of dropping out. In the end, the **percentage** of neurons dropped out from the hidden layer would be **around 50%**.
So how does **dropout** help solve the problem of **overfitting**? Well sometimes during overfitting, only a few number of **neurons** actually affect the model's predictions.

Through the random selection of neurons to drop out makes the model learn to **not rely on just a single or a few neurons** to predict because **ANY** neuron could be **dropped out**. This creates a **regularization effect** because we are **regulating** the **model's use** of its neurons. As the effect of each neuron on the model's predictions increases, the model may be able to be more complicated, increasing its chances to **generalize** rather than **memorize**.

Lastly, dropout is **only used on the training set** because its purpose is for the model not to overfit **during training**. Consequently, the **dropped out neurons** also come back to high school (their layers) when the model predicts on the validation/test set or any other data the model has not seen before.

Now that we know how dropout works, how do we implement this algorithm?
The best way is to **NOT modify** the dense layer, but to **add** a **dropout layer** after the activation function but before the next dense layer. In the dropout layer, whichever neurons we randomly selected, we **set** the **output** of that activated neuron to 0. To understand how this works, consider a hidden layer of 3 hidden units and an output layer of 2 output features:



Since setting the output of a particular neuron to 0 (easy to implement) yields the same result as modifying the behavior of the dense layer (harder to implement), we should **add** a **dropout layer** which sets the selected neurons to be dropped out to 0. Here is the code for the new dropout layer:

```python
class Layer_Dropout:
    def __init__(self, probability):
        # probability of keeping a neuron
        self.probability = probability

    def forward(self, inputs, training=True):
        if training:     # if the input data is part of the training set, use dropout
            self.mask = np.random.binomial(1, self.probability, size=inputs.shape)
            self.outputs = (inputs * self.mask) / self.probability
        else:
            self.outputs = inputs.copy()

    def backward(self, dvalues):
        self.dinputs = dvalues * self.mask
```

In the *__init__* method, we ask for the probability of keeping a neuron and save it to *self.probability*. We do this instead of asking for the dropout rate because this makes the code in the forward method easier. Since we only do dropout on the training set, we include a parameter *training* which would be True if *inputs* is part of the training set, False otherwise. If *inputs* is not part of the training set, then we automatically set *self.outputs* to a copy of *inputs*.

To randomly select the neurons to drop out we use *np.random.binomial()*. The first argument is always 1 since we are only going to need 1 set of random neurons to drop out for each time the forward method is called. The output is saved to *self.mask* which is a NumPy array of *size* (*inputs.shape*) with 0s and 1s (probability of 1 is defined by the 2nd argument: *self.probability*). In the next line when we multiply *inputs* by *self.mask*, the dropped out neurons will have an output of 0 (any 0s from *self.mask* multiplied by anything is 0). The neurons that are not dropped out will be multiplied by 1 (not changing the original neuron value).

Lastly, since in the next layer, there are fewer input features then normal, the prediction of the neuron: $\hat{y} = (w_1 * X_1) + (w_2 * X_2) + ... + b$, would be lower than normal. As a result, we need to scale the outputs of the current layer up so that the prediction of the neuron in the next layer is closer to normal. To scale the outputs of the current layer up, we divide *self.outputs* by *self.probability*. For example if **half** of the neurons were dropped out, then we would assume that the prediction of the neuron in the next layer would also be **halved**. By dividing the result by 0.5 (probability of keeping a neuron), we are essentially multiplying the outputs of the current layer by 2, **doubling** the outputs of the current layer which scales the next layer's prediction back to normal.

Remember that in dropout the dropped out neurons are no longer connected to the model and thus the model's output/prediction. As a result during backpropagation, the gradients of those dropped out neurons would be 0. We set them to 0 by multiplying *dvalues* by *self.mask* in the backward method. The 0s in *self.mask* represent the dropped out neurons so multiplying anything by that 0 results in a 0. When 1s in *self.mask* are multiplied by *dvalues*, there is no change to *dvalues*.

Now we can add in this new dropout layer into our model:

```
class Neural_Network:
    def __init__(self, n_inputs, n_hidden, n_outputs):    # n_hidden is the number of hidden neurons
        self.hidden_layer1 = Dense_Layer(n_inputs, n_hidden)
        self.activation1 = ReLU_Activation()
        self.drop1 = Layer_Dropout(0.8)
        self.hidden_layer2 = Dense_Layer(n_hidden, n_hidden)
        self.activation2 = ReLU_Activation()
        self.drop2 = Layer_Dropout(0.8)
        self.hidden_layer3 = Dense_Layer(n_hidden, n_hidden)
        self.activation3 = ReLU_Activation()
        self.drop3 = Layer_Dropout(0.8)
        self.output_layer = Dense_Layer(n_hidden, n_outputs)
        self.output_activation = Sigmoid_Activation()
        self.cost_function = BCE_Cost()

        self.trainable_layers = [self.hidden_layer1, self.hidden_layer2,
                                 self.hidden_layer3, self.output_layer]
```

In between each activation layer and the next dense layer, we add a dropout layer with a 80% probability to keep a neuron (20% dropout rate).

```
def forward(self, inputs, y_true, training=True):
    self.hidden_layer1.forward(inputs)
    self.activation1.forward(self.hidden_layer1.outputs)
    self.drop1.forward(self.activation1.outputs, training=training)
    self.hidden_layer2.forward(self.drop1.outputs)
    self.activation2.forward(self.hidden_layer2.outputs)
    self.drop2.forward(self.activation2.outputs, training=training)
    self.hidden_layer3.forward(self.drop2.outputs)
    self.activation3.forward(self.hidden_layer3.outputs)
    self.drop3.forward(self.activation3.outputs, training=training)
    self.output_layer.forward(self.drop3.outputs)
    self.output_activation.forward(self.output_layer.outputs)
    self.cost = self.cost_function.forward(self.output_activation.outputs, y_true)
```

Now in the forward method of the model, we include the *training* parameter which is set to True on default. This parameter is used as an argument when we call the forward method of the dropout layer. In between each call to the forward method of the activation layer and to the forward method of the next dense layer, we call the forward method of the dropout layer. The backward method of the model calls the backward methods in the reverse order of calling the forward methods so we won't show the backward method of the model this time.

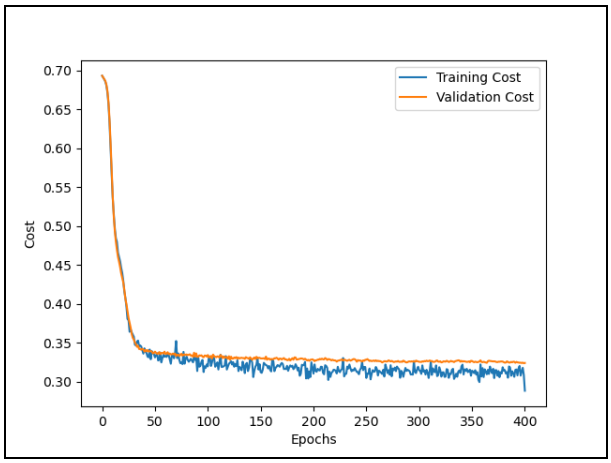
```python
for i in range(400):
    # Get Validation Cost and Accuracy
    model.forward(X_val, y_val, training=False)
    val_cost.append(model.cost)
    val_accuracy.append(get_accuracy(model.output_activation.outputs, y_val))

    model.forward(X_train, y_train)
    cost_history.append(model.cost)
    accuracy_history.append(get_accuracy(model.output_activation.outputs, y_train))
```

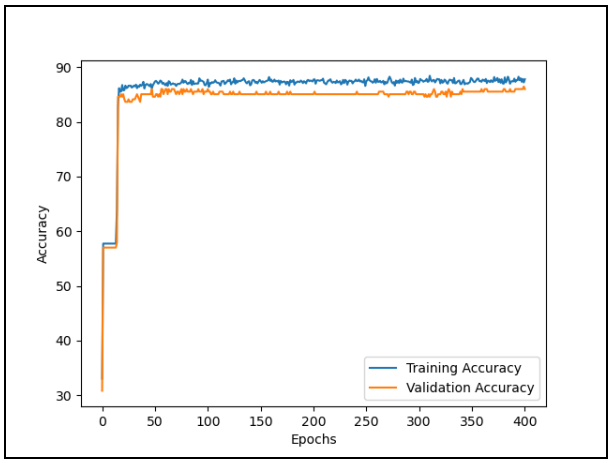
In the training loop, when we get the validation metrics by calling the forward method of the model, we set the keyword argument, *training* to False. However since by default *training* is set to True we can leave the forward method for the training set as it was.

```python
# Check Ending Cost and Accuracy
model.forward(X_train, y_train, training=False)
cost_history.append(model.cost)
accuracy_history.append(get_accuracy(model.output_activation.outputs, y_train))
print(f"Final Cost: {cost_history[-1]} - Final Accuracy: {accuracy_history[-1]}%")

model.forward(X_val, y_val, training=False)
val_cost.append(model.cost)
val_accuracy.append(get_accuracy(model.output_activation.outputs, y_val))
print(f"Final Val Cost: {val_cost[-1]} - Final Val Accuracy: {val_accuracy[-1]}%")

# Test the model on the test set
model.forward(X_test, y_test, training=False)
accuracy = get_accuracy(model.output_activation.outputs, y_test)
print(f"Test Set Results ~ Cost: {model.cost} - Accuracy: {accuracy}%")
```

Lastly, after the training loop when we get the ending metrics by calling the forward method of the model, we set *training* to False for all 3 sets of data. This includes the training set because at the end of training, we do not need to prevent overfitting anymore and also without dropout we can see the actual model's performance (no dropped out neurons) on the training set.

There is no change needed when we create the graphs so here is the last three lines of output from the terminal and the graphs we have created:
Final Cost: 0.2886016687437633 - Final Accuracy: 87.77648428405122%
Final Val Cost: 0.3242930836066085 - Final Val Accuracy: 85.98130841121495%
Test Set Results ~ Cost: 0.37157598603623554 - Accuracy: 84.72222222222221%





We cannot do anything about the constant jumping up and down of the cost in the training set because this was created by the random neurons constantly being dropped out and put back into the model. The large drop in the training cost at the last epoch is outside the training loop where we set *training* to False, stopping dropout. Thus, when there is no dropout, we can clearly see that the model does a little better on the training set. The validation cost & accuracy now follows the training cost & accuracy more closely meaning that gains in the training set are due to more **generalization**.

Lastly, let's compare final results with dropout to without dropout:

Set of Examples	Cost		Accuracy	
	Without Drop.	With Drop.	Without Drop.	With Drop.
Training Set	~0.286	~0.2886	~88.65%	~87.78%
Val. Set	~0.3387	~0.3243	~83.645%	~85.981%
Test Set	~0.3886	~0.3716	~83.796%	~84.722%

Same with regularization, the training set metrics have decreased a bit but results in gains in the metrics for the validation and test set. As a result, L_2 Regularization and dropout were successful techniques in preventing the model from overfitting.

This concludes our binary classification journey together, to gain more experience in training models, go ahead and finish training this binary classifier until it becomes **smart** (not only on the training set though). In the next unit we will continue on the classification problem but through a multi-class classification problem.