Chapter 10| Multi-Class Classification

~~~~~~

Structure of Datasets

Unlike binary classification where the model sorts examples into **2 classes**, multi-class classification is where the model sorts examples into **multiple different classes**. Both of these problems are part of the classification task in AI but their only difference is with the number of classes the model has to sort examples into. Binary classification deals with **2** (by its name, **binary**) and multi-class classification deals with **more than 2**.

To start, let's review the process for creating any 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.

We are not going to collect data in this book 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)? Just like how in binary classification, the classes are **represented** by **positive integers**. Consider inputting an image into a model and predicting whether the image is a picture of a computer monitor, computer mouse, or
~~~~~~

computer keyboard. We can set the class of a computer monitor to class 0, class of a computer mouse to class 1, and class of a computer keyboard to class 2. Notice that we did not have to stick with only 0 and 1. **For more classes/categories**, we would just use the **next positive integer**.

Now how do we get our model (neural network) to output (predict) **only positive integers**? Well the answer is that we don't. Just like binary classification, all that is needed is the model to **predict** a **probability/confidence** that the image is a particular class/category. So after we input the image to the model, we would like to know the model's confidence that it is a picture of a computer monitor, the model's confidence that it is a picture of a computer mouse, and the model's confidence that it is a picture of a computer keyboard.

That is 3 different predictions we would like from the model (**one probability for each class**). As a result, for this problem we would like 3 output features. In general, **for each CLASS in multi-class classification, we would like another output feature**. For example, if there were 3 classes, 3 output features are needed, 5 classes means that 5 output features are needed, and 7 classes result in 7 output features.

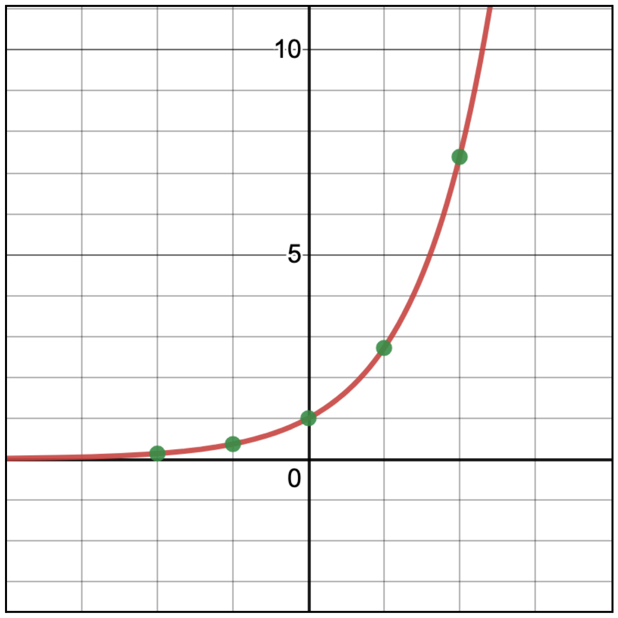
Now the question is what activation function in the output layer can result in **probabilities** (restricted to the 0-1 range). We **could** use the sigmoid activation function for each output feature but this does not guarantee that adding up the probabilities for each class results in **100%.** Since there are only the classes of a computer monitor, mouse, and keyboard all the probabilities for each class **should** add up to 100%, there are no other classes. As a result, we need a new activation function for the output layer. This activation function is called the **Softmax activation** which will be explained in the next section.

Softmax Activation Function

The softmax activation function is based on pulling out names from a hat. Suppose the task of predicting whether an image contains a computer monitor, mouse or keyboard. If the model predicts a 2 for the monitor, 3 for the mouse, and 5 for the keyboard, this would be the equivalent of 2 entries of the word "monitor", 3 entries of the word "mouse" and 5 entries of the word "keyboard" inside of the hat. The probability of any word being pulled out of the hat would be the **number of entries of that word (i)** divided by the **total number of entries in the hat (n)**: $\frac{i}{n}$. As a result, the model predicts that there is a $\frac{2}{2+3+5} = \frac{2}{10} = 20\%$ chance that the image contains a monitor, a $\frac{3}{2+3+5} = \frac{3}{10} = 30\%$ chance that the image contains a mouse, and a $\frac{5}{2+3+5} = \frac{5}{10} = 50\%$ chance that the image contains a keyboard. Despite the satisfying fact that the probabilities add up to 100%, this algorithm breaks when the model outputs a negative value for one of its output features.

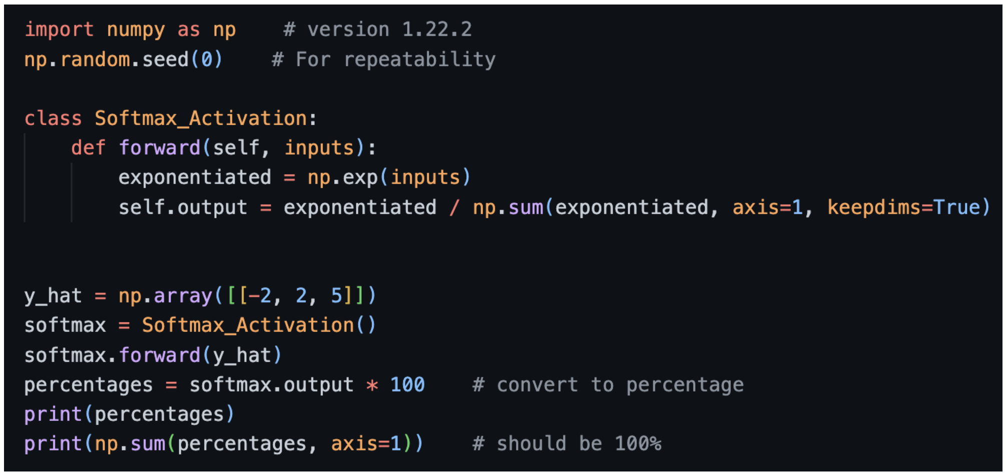
For example, if the model predicts a -2 for the monitor, 2 for the mouse, and 5 for the keyboard, there is a $\frac{-2}{-2+2+5} = \frac{-2}{5} =- 40\%?$ chance that the image contains a monitor, a $\frac{2}{-2+2+5} = \frac{2}{5} = 40\%$ chance that the image contains a mouse, and a $\frac{5}{-2+2+5} = \frac{5}{5} = 100\%$ chance that the image contains a keyboard. The probabilities no longer add up to 100% and there is a negative probability: -40%. Clearly, we need a better algorithm. This is where softmax comes in.

Instead of **directly** using the **raw** outputs from the model, softmax takes the **raw** outputs and **exponentiates** them before adding entries into the hat. By using softmax, the model's predictions are now: a $\frac{e^{-2}}{e^{-2}+e^{2}+e^{5}} \approx 0.087\%$ chance for a monitor, a $\frac{e^{2}}{e^{-2}+e^{2}+e^{5}} \approx 4.74\%$ chance for a mouse, and a $\frac{e^{5}}{e^{-2}+e^{2}+e^{5}} \approx 95.2\%$ chance for a keyboard. The e is Euler's Number: ~2.71828182846. Now the model is very confident that the image contains a computer keyboard due to the nature of exponentiation:



Here is a graph of e^x. Softmax interprets negative values as the model's way of saying that the example is **clearly** not that class, thus after exponentiation, the value becomes less than 1. This explains the less than 0.1% chance for a monitor. As the model's outputs become more and more positive, softmax interprets it as the model's way of saying that the example is **very likely** to be that particular class. This explains the 95% chance for a keyboard. Most importantly, even after exponentiating the model's output, all the probabilities summed up equal 100% because all exponentiated values are positive values.

Now that we know how the softmax activation function works, here is the code that implements the forward method (the backward method of the softmax activation function alone is very complex, so we won't explain the backward method):



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

class Softmax_Activation:
    def forward(self, inputs):
        exponentiated = np.exp(inputs)
        self.output = exponentiated / np.sum(exponentiated, axis=1, keepdims=True)


y_hat = np.array([[-2, 2, 5]])
softmax = Softmax_Activation()
softmax.forward(y_hat)
percentages = softmax.output * 100    # convert to percentage
print(percentages)
print(np.sum(percentages, axis=1))    # should be 100%
```

In the forward method, we use *np.exp* to exponentiate the values and save them to *exponentiated*. The output of softmax would then be *exponentiated* divided by **each**

example's sum of exponentiated values which is done by passing the keyword argument *axis=1* (column) to signify that for **each example** we would like to add up its exponentiated columns. Lastly, the keyword argument *keepdims* is set to True to prevent the result from becoming a row vector (only 1 row) since the sum is for each example (multiple rows).

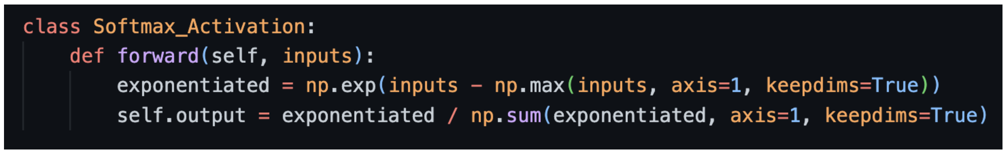
To test our softmax activation function, we are going to input [-2, 2, 5]. When we define the NumPy array for this, we use double brackets because the first bracket indicates the rows (1 example) and the second bracket indicates the columns (3 output features). Next, we instantiate a softmax activation, call the forward method, and convert its output to a percentage by multiplying it by 100 and storing the result into *percentages*. Lastly, we print the percentages and for each example, we add its percentages up to check that the result is 100%. Here is the output from the terminal:

```
[[8.67881295e-02 4.73847131e+00 9.51747406e+01]]
[100.]
```

Remember that the e-02 means the number in front of it multiplied by 0.01 (10 to the power of -2), e+00 means to multiply the number in front of it by 1 (10 to the power of 0), and e+01 means to multiply the number in front of it by 10 (10 to the power of 1). For further reassurance, the last line of output was 100% so we did this correctly. There is still one more thing to fix with our softmax activation function.

If the model outputs a huge number like 800, then we will hit an error in which e^{800} is too big of a number to work with in python. Doing this operation would create an **overflow error** (number is too large). To combat this overflow error, we can subtract each output by the maximum output of the example. Doing this would bring all the outputs to be less than or equal to 0 (the largest number subtracted by itself is 0 so any other number subtracted by the maximum is negative).

Raising e (Euler's number) to 0 is a 1 and raising e to any negative numbers results in a value less than 1. Now the exponentiated values can never create an overflow error. At the same time, the probabilities that the model predicts do not change if we perform the subtraction before exponentiation. Here is the new code for softmax:



```python
class Softmax_Activation:
    def forward(self, inputs):
        exponentiated = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))
        self.output = exponentiated / np.sum(exponentiated, axis=1, keepdims=True)
```

Now before we exponentiate the values, we subtract them by each example's maximum value by using *np.max()* with *axis=1* and *keepdims* set to True for the same reason for setting *keepdims* to True when we call *np.sum()*. Since the result of this softmax activation function is the same as the previous, there is no need to show the terminal output.

In the next section, we are going to use a new cost function for multi-class classification since the cost function for binary classification only deals with 2 classes (if y=0 and if y=1).

Categorical Cross Entropy Cost

How do we calculate the cost for multiple classes? Well we actually do this almost the same way we calculate the cost for 2 classes.

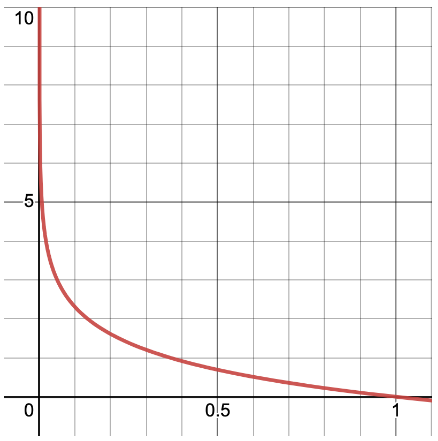
$$\boxed{BCE = \underbrace{[y * \; -ln(\hat{y})]}_{\text{Part 1}} + \underbrace{[(1-y) \; * \; -ln(1-\hat{y})]}_{\text{Part 2}}}$$

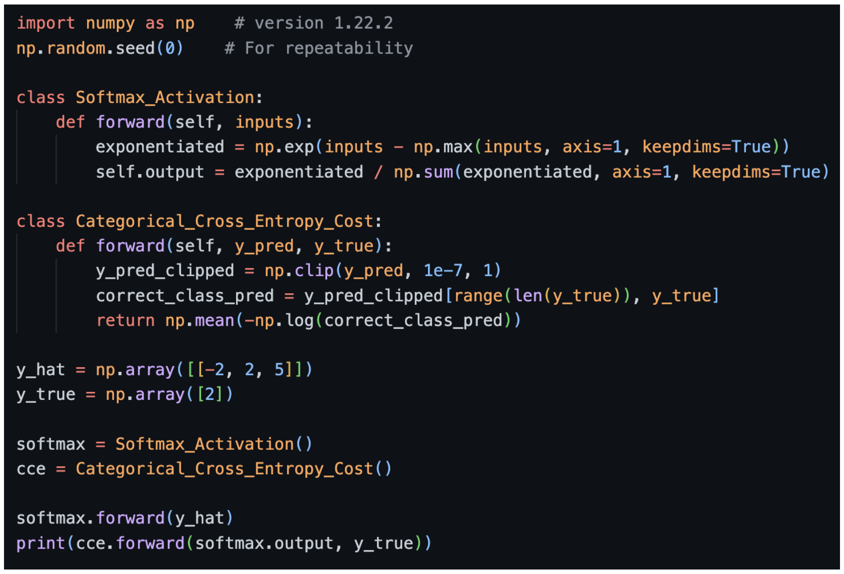
We know that ŷ is the probability that the example is class 1, but how do we calculate the probability that the example is class 0? Since we only have 2 classes all we have to do is subtract ŷ from 1. If we modify the equation where p_1 is the probability that the example is class 1 and p_0 is the probability that the example is class 0, we get:

$$\boxed{BCE = \underbrace{[y * \; -ln(p_1)]}_{\text{Part 1}} + \underbrace{[(1-y) * \; -ln(p_0)]}_{\text{Part 2}}}$$

As we can see since y can only be 0 or 1, the cost would either be part 1 or part 2. In each part, **the cost is the negative natural logarithm of the model's predicted probability for the correct class**. If the correct class is class 1, the cost is part 1 or the negative natural logarithm of p_1 (model's predicted probability for class 1). If the correct class is class 0, the cost is part 2 or the negative natural log of p_0 (model's predicted probability for class 0).

Hence, for multi-class classification, all we have to do is compute the negative natural logarithm of the model's predicted probability for the correct class. Here is a graph of the cost function for any correct class:
Since probabilities are between 0 and 1 all the inputs to the negative natural logarithm are between 0 and 1. As the model's prediction gets closer to 1 (higher predicted probability of the correct class) the cost drops to 0. As the model's prediction gets closer to 0 (low predicted probability of the correct class) the cost explodes to infinity. Now here is the code for the new cost function:





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

class Softmax_Activation:
    def forward(self, inputs):
        exponentiated = np.exp(inputs - np.max(inputs, axis=1, keepdims=True))
        self.output = exponentiated / np.sum(exponentiated, axis=1, keepdims=True)

class Categorical_Cross_Entropy_Cost:
    def forward(self, y_pred, y_true):
        y_pred_clipped = np.clip(y_pred, 1e-7, 1)
        correct_class_pred = y_pred_clipped[range(len(y_true)), y_true]
        return np.mean(-np.log(correct_class_pred))

y_hat = np.array([[-2, 2, 5]])
y_true = np.array([2])

softmax = Softmax_Activation()
cce = Categorical_Cross_Entropy_Cost()

softmax.forward(y_hat)
print(cce.forward(softmax.output, y_true))
```

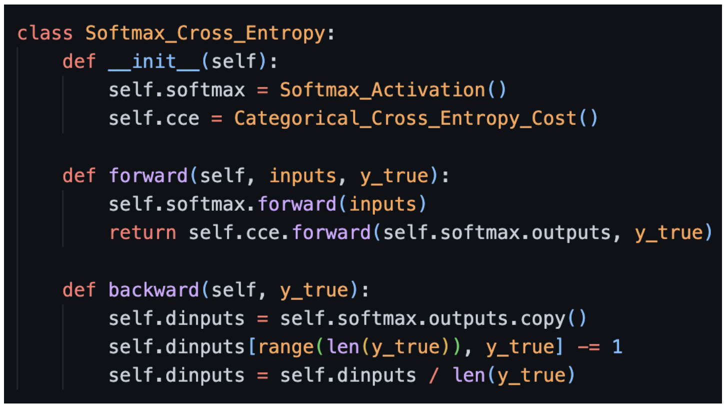
This cost function for multi-class classification is called **Categorical Cross Entropy**. First we clip the predictions where the lower bound is 1e-7 or 1 * 10^{-7} so the cost does not explode to infinity if we predict very close to 0%. The upper bound stays at 1 because the cost will not explode if we predict very close to 100%.

Next, to get the model's predicted probability for the correct class, we can index by using NumPy. The row index will iterate through from 0 and stop at the *len(y_true)* and for each example, the column index will be that row's correct class. For example, we will index column 0 of that current example's prediction if *y_true* for that current row is 0 (class 0). This is the beauty of NumPy where we did not have to use a for-loop to iterate through the array to index the probabilities for the correct class.

Continuing with our old [-2, 2, 5] example we set *y_true* to be 2 which means that class 2 is the correct class. After that, we instantiate a softmax activation and cost function, call the forward methods of both and print the return value from the cost function. This is the terminal output: 0.04945560969564589

We have a very low cost because the model's probability prediction on class 2 (the correct class) is very high, ~95%. This means that the model was very close to the correct class so the error/cost, the model receives for this example is very low. We still have error/cost even though the model's probability prediction is very close to 100% because ideally, we would like the model to have even more confidence that the predicted class is the correct class.

We covered the forward method of both the new activation function for the output layer and the new cost function for multi-class classification, but why did we not cover the backward methods? In reality, the backward methods of softmax and categorical cross entropy done separately is very complicated. However, when we combine softmax and categorical cross entropy together, the backward method of the combination is very simple. Here is the code to show you what I mean:



```
class Softmax_Cross_Entropy:
    def __init__(self):
        self.softmax = Softmax_Activation()
        self.cce = Categorical_Cross_Entropy_Cost()

    def forward(self, inputs, y_true):
        self.softmax.forward(inputs)
        return self.cce.forward(self.softmax.outputs, y_true)

    def backward(self, y_true):
        self.dinputs = self.softmax.outputs.copy()
        self.dinputs[range(len(y_true)), y_true] -= 1
        self.dinputs = self.dinputs / len(y_true)
```

To combine softmax and categorical cross entropy together, we define a new class which contains a `__init__` method to instantiate a softmax activation and cost function.
The `forward` method would then just be calling the forward methods of the softmax activation and returning the output of the cost function.

The `backward` method is a different story however. We first make a copy of the outputs from the softmax activation which are
the model's predicted probabilities. Next, we index the model's predicted probabilities for the correct class (same technique of indexing in the `forward` method of the cost function) and subtracted 1 (100%) from those probabilities. Lastly we divide every model prediction by the total number of examples, `len(y_true)`.

This is so that we stay mathematically consistent with the forward method of the cost function where we took the **average/mean** of each example's cost. Taking the **average/mean** is the equivalent of using a fraction, 1/**n** (where **n** is the number of examples) of each example's cost and adding them up to get the overall cost.

Before putting the calculation in the backward method into perspective on what it means, let's continue with our old example. Now after we define our demonstrator data, we instantiate the combination, print out the cost after calling the forward method, call the backward method and print out the `dinputs` attribute.
Here is the output from the terminal:

```
0.04945560969564589
[[ 0.00086788  0.04738471 -0.04825259]
```

Since class 0 and class 1 are the incorrect classes for this example, its derivatives/slopes are **positive**, which means that if the model increases its output for class 0 or class 1 (model's probability prediction on the correct class will decrease), the cost will **increase**. The derivatives for the incorrect classes will always stay positive since in the backward method we do not change them from the output of the softmax activation which is always positive (probabilities are positive values). We only divide the derivatives for the incorrect classes by the number of examples which is a positive divided by a positive, keeping the derivatives/slopes a **positive** value.

On the other hand, since class 2 is the correct class for this example, its derivative/slope is **negative**, which means that if the model increases its output for class 2 (model's probability prediction on the correct class will increase), the cost will **decrease**. The derivative for the correct class will always be negative because in the backward method we subtract the output of the softmax activation (probabilities are always less than 1 or 100%) by 1 to get a negative result. Dividing the negative derivatives for the correct classes by the number of examples (a positive value), keeps the derivative/slope a **negative** value.

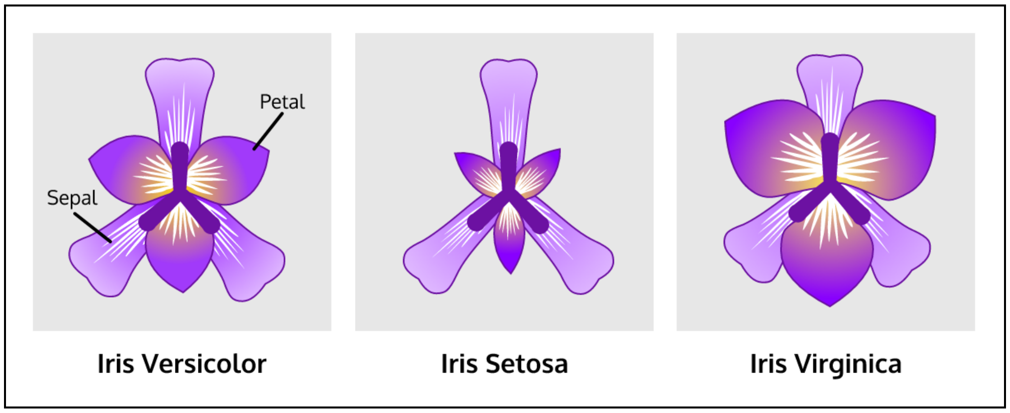
By having positive derivative values for the incorrect class and negative derivatives values for the correct class, helps the model figure out which outputs to not increase (decrease) and which outputs to increase to decrease the overall cost.

Now that we have the forward and backward methods, a combined activation function for the output layer and a cost function to use in multi-class classification, we can start creating a multi-class classifier. In the next section, we are going to be downloading, reading and setting up a dataset for the multi-class classification problem.

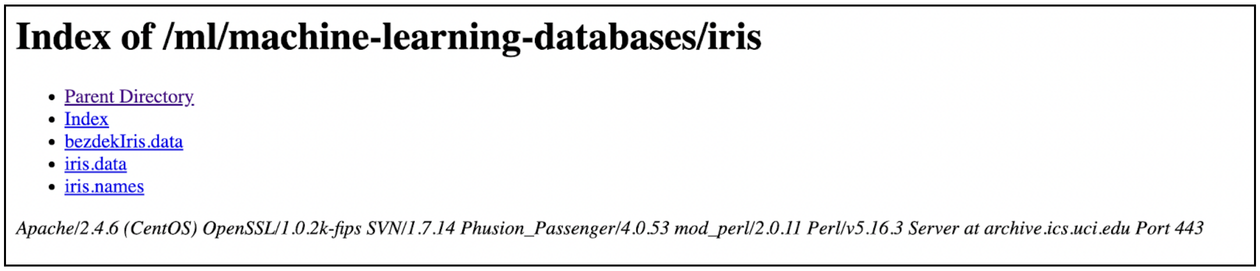
Dataset & Setup

To give an example of a multi-class classifier being trained, we are going to use the Iris Dataset. The Iris Dataset is a very popular and very beginner-friendly dataset since there are only 150 examples and 4 input features. These input features are characteristics of a certain flower from the Iris flower family. The task is that given those 4 input features (sepal length, sepal width, petal length, petal width), we need to classify whether the flower is an Iris Setosa, Iris Versicolour, or an Iris Virginica flower.

Here is a image from Codecademy that shows the 3 different flowers and the difference between a **sepal** and a **petal**:



Now let's download this dataset.
Go to https://archive.ics.uci.edu/ml/machine-learning-databases/iris/:



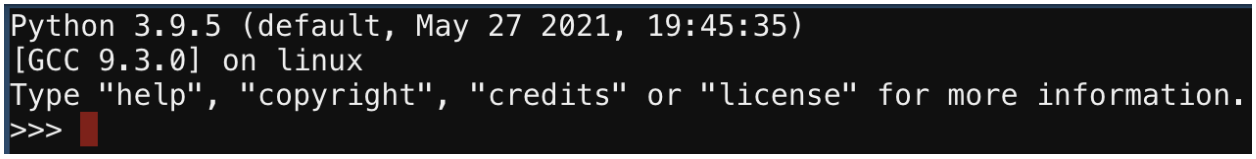
Index of /ml/machine-learning-databases/iris

- Parent Directory
- Index
- bezdekIris.data
- iris.data
- iris.names

Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips SVN/1.7.14 Phusion_Passenger/4.0.53 mod_perl/2.0.11 Perl/v5.16.3 Server at archive.ics.uci.edu Port 443

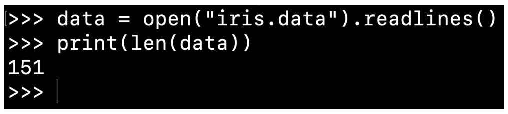
Click on *iris.data* which will download a *.data* file onto your computer. Move *iris.data* into the directory you are working at. Even though this dataset is in a *.data* file we can still use Python to open this file, read it and convert it into a *.csv* file so that we can easily use Pandas to get the dataset.

Like always, we start by launching the command prompt/line or terminal on your computer, navigate to the directory the ***iris.data*** file is located in and then use the 'python' command to launch the shell. The shell should look something like this:



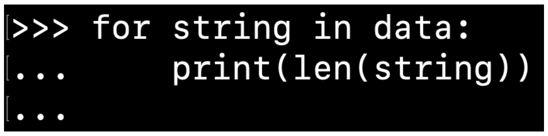
```
Python 3.9.5 (default, May 27 2021, 19:45:35)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

First we need to understand the way the data is formatted. To do this we start by using the built-in python function, open() and input the name of the *.data* file. Next we use the .readlines() method to obtain a list of strings (contents of each line of the file). After that, we print the length of the data list to make sure we have 150 examples.



```
>>> data = open("iris.data").readlines()
>>> print(len(data))
151
>>>
```

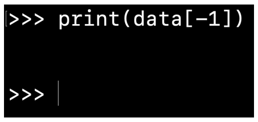
However, instead we see that there are 151 lines in the file. To find which example is faulty, we can print the length of each string in the list:



```
>>> for string in data:
...     print(len(string))
...
```

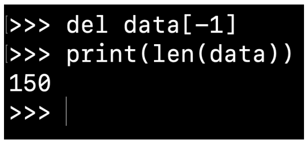
To do this, we use a for-loop to iterate through each string in the data list. The output should be a long line of 28s then 32s, then 31s, and lastly a 1.

Now we can assume that the last string in the list is not an example that we can use for training but assumptions can be faulty themselves so let's check that the last element in the list is faulty:



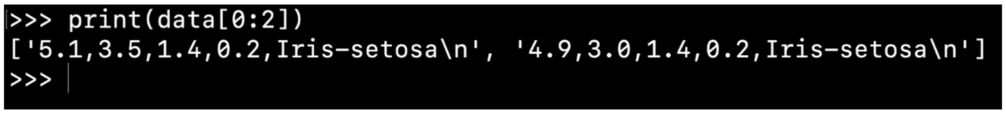
```
>>> print(data[-1])


>>>
```

We have evidence that the last element in the list is faulty so let's get rid of it by using the del statement:



```
>>> del data[-1]
>>> print(len(data))
150
>>>
```

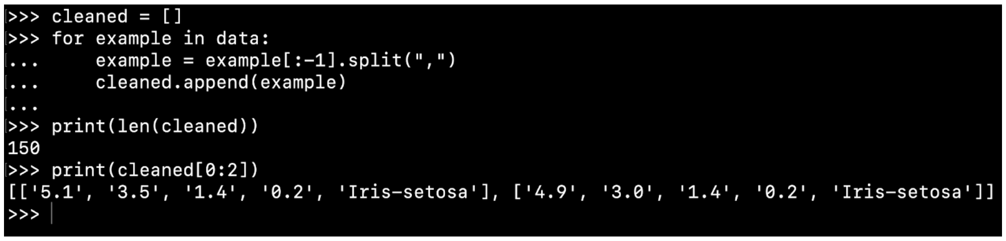
By printing the new length of the list, we see that we have the correct 150 examples. Now we can actually understand the way the data is formatted:



```
>>> print(data[0:2])
['5.1,3.5,1.4,0.2,Iris-setosa\n', '4.9,3.0,1.4,0.2,Iris-setosa\n']
>>>
```

We print the first two elements of the list and see that the input features are separated by commas (,) and the correct class (flower type) is separated from the last input feature by another comma. Lastly, to signal the end of the example there is a newline character (\n).

Since all the examples are in a string, we need to convert each example into a list with 5 elements (4 input features and the last element containing the correct class). By doing so, we have a list of lists which can be converted to Pandas easily:



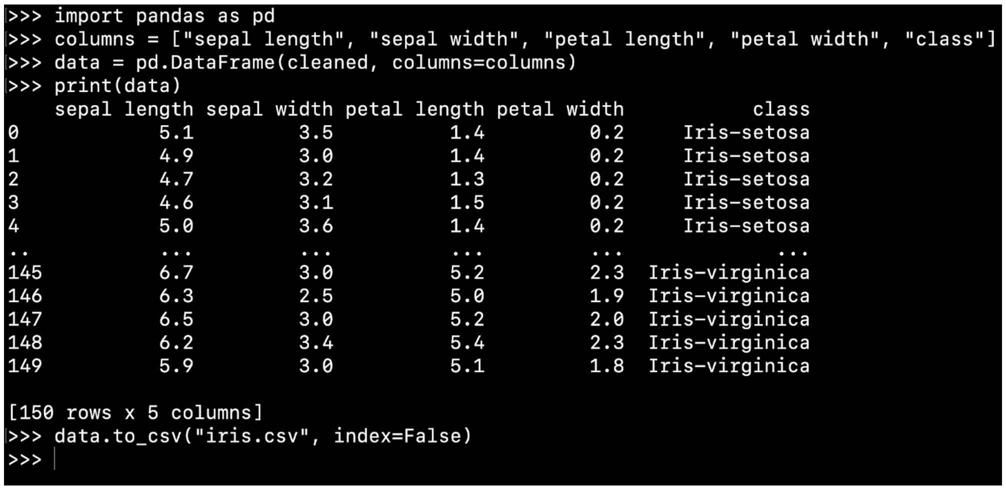
```
>>> cleaned = []
>>> for example in data:
...     example = example[:-1].split(",")
...     cleaned.append(example)
...
>>> print(len(cleaned))
150
>>> print(cleaned[0:2])
[['5.1', '3.5', '1.4', '0.2', 'Iris-setosa'], ['4.9', '3.0', '1.4', '0.2', 'Iris-setosa']]
>>>
```

First we start with an empty list cleaned which will be a list of lists to contain all the examples that are converted into a list. Next, we iterate through each example with a for-loop. For each example, we first get rid of the newline character (\n) by indexing everything except the last character and then we use the split method of a string. The split method splits the string each time there is a comma (,) which is the argument to the method and each substring from the split string is put into a list. The resulting list after the whole string is split is returned so now the list is stored in example.

After splitting, the list contains the 5 elements we mentioned earlier. After we have successfully converted each example into a list, we append that list to cleaned to store all the examples that have been cleaned (converted into a list).

Lastly, we print the length of cleaned and see that we still have the 150 examples and check the first two examples. Sure enough, the first two examples have successfully been converted from a string of elements separated by commas to a list of 5 elements.

Now we can save cleaned into a **csv** file using pandas:

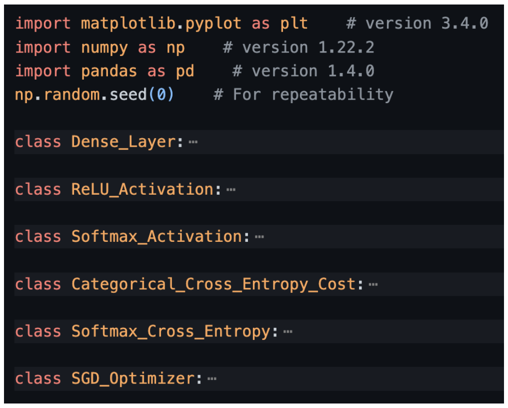


```
>>> import pandas as pd
>>> columns = ["sepal length", "sepal width", "petal length", "petal width", "class"]
>>> data = pd.DataFrame(cleaned, columns=columns)
>>> print(data)
    sepal length sepal width petal length petal width           class
0            5.1         3.5          1.4         0.2     Iris-setosa
1            4.9         3.0          1.4         0.2     Iris-setosa
2            4.7         3.2          1.3         0.2     Iris-setosa
3            4.6         3.1          1.5         0.2     Iris-setosa
4            5.0         3.6          1.4         0.2     Iris-setosa
..           ...         ...          ...         ...             ...
145          6.7         3.0          5.2         2.3  Iris-virginica
146          6.3         2.5          5.0         1.9  Iris-virginica
147          6.5         3.0          5.2         2.0  Iris-virginica
148          6.2         3.4          5.4         2.3  Iris-virginica
149          5.9         3.0          5.1         1.8  Iris-virginica

[150 rows x 5 columns]
>>> data.to_csv("iris.csv", index=False)
>>>
```

First we have to import pandas. Next we need to create column names so that we do not forget what each column represents. Then, we create a pandas object by pd.DataFrame which we input the list of lists, cleaned and set the keyword argument columns to columns. The print statement shows that we have successfully converted the list of lists into a pandas object. Lastly, we save the pandas object to a csv file by using the to_csv method which takes in the file location we would like to save it to. The keyword argument index is set to False because we do not need another column that gives the example number of each example since we would not need to figure out which example is which.

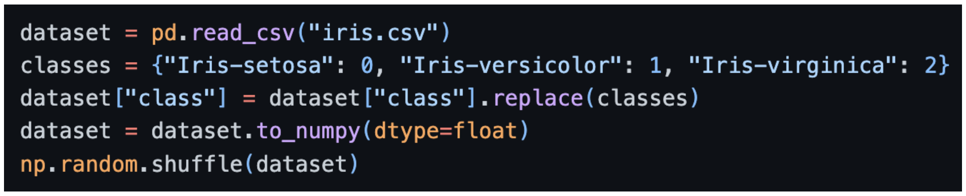
Now there should be a csv file in the directory your python shell is working in similar to the csv located here.

Now that we have a dataset in the format that we can work with easily, let's create the python script that will start training a multi-class classifier (here is the full code):



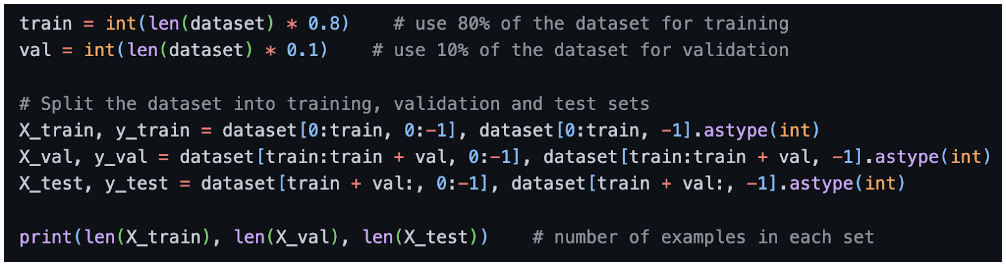
```
import matplotlib.pyplot as plt     # version 3.4.0
import numpy as np     # version 1.22.2
import pandas as pd     # version 1.4.0
np.random.seed(0)     # For repeatability

class Dense_Layer:⋯

class ReLU_Activation:⋯

class Softmax_Activation:⋯

class Categorical_Cross_Entropy_Cost:⋯

class Softmax_Cross_Entropy:⋯

class SGD_Optimizer:⋯
```

First we import matplotlib to plot the history of the performance of the model during training. We also need to import numpy to store the data loaded by pandas. Copy and paste the dense layer class, relu activation class for the hidden layers, softmax activation class for the output layer, the cost function, the combination of the softmax activation and cost function, and the SGD optimizer. The three dots after each class definition hides the code to save space.



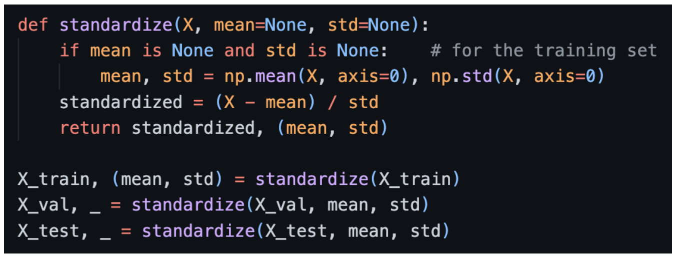
```
dataset = pd.read_csv("iris.csv")
classes = {"Iris-setosa": 0, "Iris-versicolor": 1, "Iris-virginica": 2}
dataset["class"] = dataset["class"].replace(classes)
dataset = dataset.to_numpy(dtype=float)
np.random.shuffle(dataset)
```

We can load the data by using `pd.read_csv` and input the path to the **csv** file we created. To convert each class into numbers we start by creating a dictionary that maps "Iris-setosa" to class 0, "Iris versicolor" to class 1, and "Iris virginica" to class 2. Now we can use the `replace` method (use the dictionary as the argument) of the class column of the dataset and set that to be as a replacement for the class column of the dataset. Next, we convert the pandas object to a NumPy object using the `to_numpy` method of the pandas object. Lastly, we use `np.random.shuffle` so that we can split the dataset into training, validation, and testing sets in case the examples of the dataset are arranged in a way that adds bias into the model.



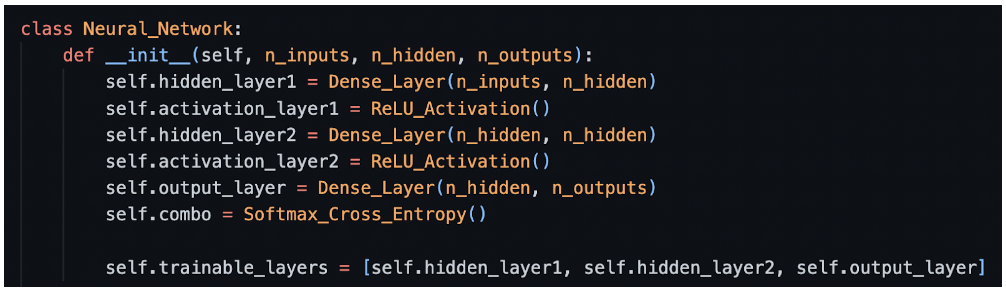
```
train = int(len(dataset) * 0.8)     # use 80% of the dataset for training
val = int(len(dataset) * 0.1)     # use 10% of the dataset for validation

# 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)

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

We still use 80% of the dataset for training, 10% for validation and 10% for the test set. Next we split the dataset using the same method as we did for the binary classification dataset. Lastly, we print the number of examples in each set of examples.



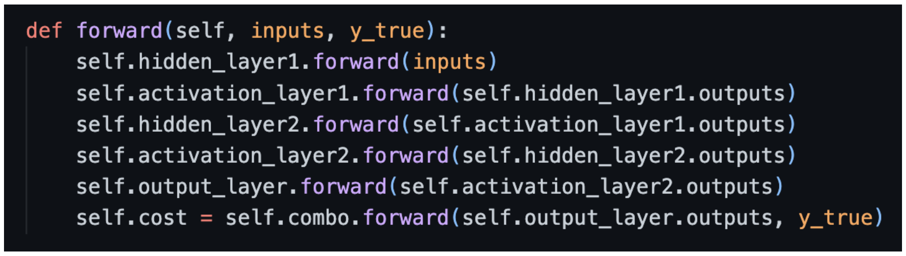
```
def standardize(X, mean=None, std=None):
    if mean is None and std is None:    # for the training set
        mean, std = np.mean(X, axis=0), np.std(X, axis=0)
    standardized = (X - mean) / std
    return standardized, (mean, std)

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

Now we can standardize the dataset to get all the data ready for training. The reason why we chose standardization over normalization is because we are dealing with Iris flowers, where the physical characteristics of each flower should be pretty similar to each other in a range of values. Thus, the relationships would closely match the result of standardization.



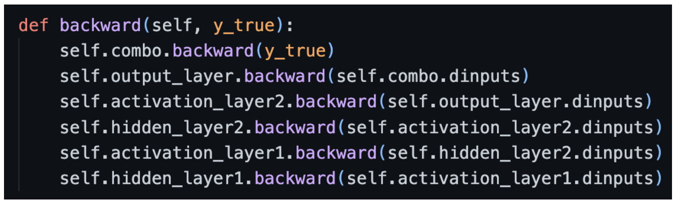
```
class Neural_Network:
    def __init__(self, n_inputs, n_hidden, n_outputs):
        self.hidden_layer1 = Dense_Layer(n_inputs, n_hidden)
        self.activation_layer1 = ReLU_Activation()
        self.hidden_layer2 = Dense_Layer(n_hidden, n_hidden)
        self.activation_layer2 = ReLU_Activation()
        self.output_layer = Dense_Layer(n_hidden, n_outputs)
        self.combo = Softmax_Cross_Entropy()

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

For our model, we will only have 2 hidden layers and 1 output layer because the Iris dataset does not require a large, complex model. Right after the output layer we add in the combination of the softmax activation and cost function. Now `self.trainable_layers` consists of only the 2 hidden layers and the output layer.



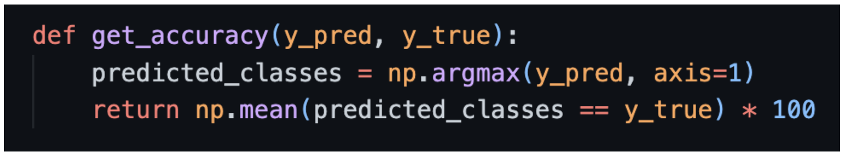
```
def forward(self, inputs, y_true):
    self.hidden_layer1.forward(inputs)
    self.activation_layer1.forward(self.hidden_layer1.outputs)
    self.hidden_layer2.forward(self.activation_layer1.outputs)
    self.activation_layer2.forward(self.hidden_layer2.outputs)
    self.output_layer.forward(self.activation_layer2.outputs)
    self.cost = self.combo.forward(self.output_layer.outputs, y_true)
```

The `forward` method of the model calls the `forward` method of each item part of the model in the same order that the `__init__` method defined them.



```python
def backward(self, y_true):
    self.combo.backward(y_true)
    self.output_layer.backward(self.combo.dinputs)
    self.activation_layer2.backward(self.output_layer.dinputs)
    self.hidden_layer2.backward(self.activation_layer2.dinputs)
    self.activation_layer1.backward(self.hidden_layer2.dinputs)
    self.hidden_layer1.backward(self.activation_layer1.dinputs)
```

The `backward` method of the model calls the `backward` method of each item part of the model but in the **reverse** order that each `forward` method of each item part of the model was called in the `forward` method of the model.



```python
def get_accuracy(y_pred, y_true):
    predicted_classes = np.argmax(y_pred, axis=1)
    return np.mean(predicted_classes == y_true) * 100
```

Lastly, we add in a function to get the accuracy of the model on a set of data. To get the class that the model predicts the example is, we use `np.argmax` which will give us the index of the maximum value in the inputted data (y_pred). Essentially for each example, we are trying to find which index/class/column (axis=1) has the largest probability from the prediction of the model (y_pred).

After that, we create a boolean array by comparing whether each example's predicted class matches the correct class. Each False value in the boolean array is equivalent to a 0 and each True value in the boolean array is equivalent to a 1 so by using `np.mean` we are adding up all the values inside the boolean array and dividing the sum by the total number of values in the boolean array. The result after multiplying by 100 is the percentage of examples that the model has predicted correctly on, or otherwise known as the accuracy of the model on that particular set of data.

Before we continue in initializing the model, setting up the training loop, and plotting the history of the performance of the model, we are going to use a different training method that is very commonly used these days. In the next section, we will explain the new training method that is useful in reducing training time.

Mini-Batch Gradient Descent

Now in AI, more and more harder problems require more and more complex models. However as the model complexity increases, running a full forward and backward pass through the same sized dataset takes more time. The reason for this is because with more complex models, the number of layers and number of neurons increases so more calculations are needed to get the outputs of each new layer of neurons.

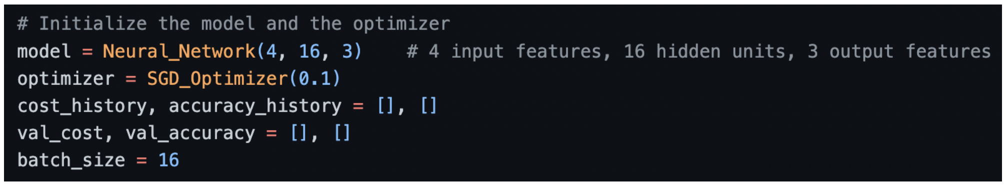
As a result, we need to train the model with more efficient methods. One strategy is through **Mini-Batch Gradient Descent**. Mini-Batch GD suggests that instead of taking a full forward pass and backward pass through the model using the **whole** dataset just to update the parameters **once**, we should split the training set into **mini-batches** of data.

Each time we **go through** (forward pass, backward pass, and update parameters) **one batch of data**, we call that a **step**. After one step, we move on to the next mini-batch of data. After going through every **batch of data**, we call that a completed **epoch** and move on to the next epoch where we start again at the first batch of data.

Now after only going through **a portion** of the dataset, we can update the parameters which means the model can **make progress more earlier and often** since each dataset would contain multiple batches of data.

Mini-Batch GD is also used on large datasets or a combination of a large dataset and complex model for the same advantages: **efficiency and more updates in a shorter amount of time**. Now although for the Iris dataset, the dataset is not large and our model isn't complex, we are using it on this task to **easily demonstrate** its advantages and disadvantage (we will see its disadvantage very clearly after seeing the results).

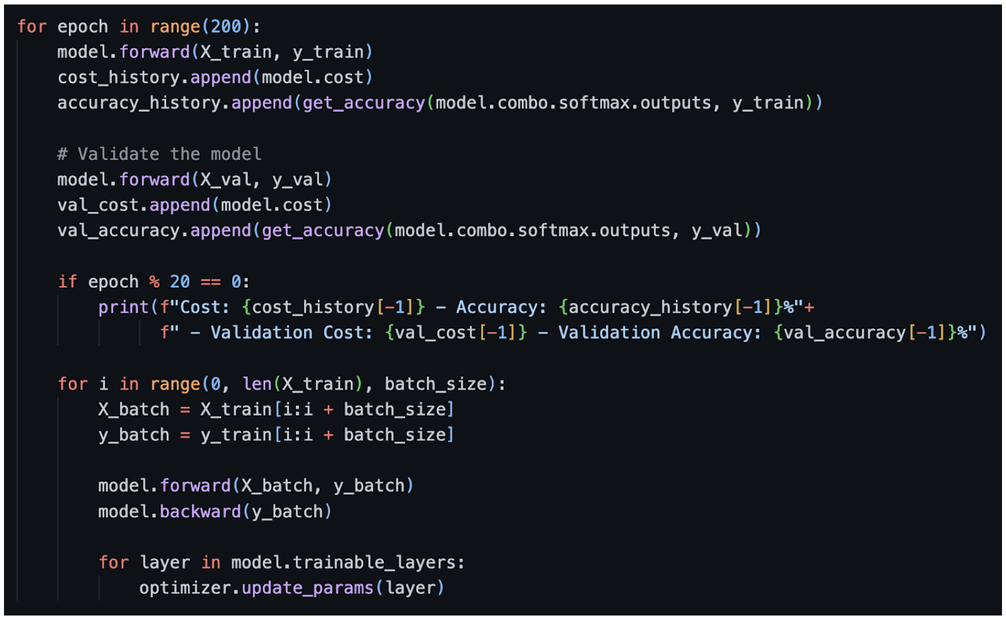
Now let's continue with this following code:



```python
# Initialize the model and the optimizer
model = Neural_Network(4, 16, 3)    # 4 input features, 16 hidden units, 3 output features
optimizer = SGD_Optimizer(0.1)
cost_history, accuracy_history = [], []
val_cost, val_accuracy = [], []
batch_size = 16
```

Our model accepts 4 input features, has 16 hidden units for each of its 2 hidden layers and 3 output features for its output layer. The optimizer is a simple SGD optimizer (no need for stronger optimizers due to the simplicity of the Iris dataset problem) with a learning rate set at 0.1. Like before, we always record the model's cost and accuracy on the training & validation set after every epoch.

Lastly, we have a `batch_size` hyperparameter which specifies the number of examples we would like to have in each batch during each **step**. Common settings of this parameter are 16, 32, 64, 128, and 256. It is possible that by setting the batch sizes to powers of 2, speeds up the training process due to the fact that computers use binary to store and manage data. Here is the new training loop:



```python
for epoch in range(200):
    model.forward(X_train, y_train)
    cost_history.append(model.cost)
    accuracy_history.append(get_accuracy(model.combo.softmax.outputs, y_train))

    # Validate the model
    model.forward(X_val, y_val)
    val_cost.append(model.cost)
    val_accuracy.append(get_accuracy(model.combo.softmax.outputs, y_val))

    if epoch % 20 == 0:
        print(f"Cost: {cost_history[-1]} - Accuracy: {accuracy_history[-1]}%"+
              f" - Validation Cost: {val_cost[-1]} - Validation Accuracy: {val_accuracy[-1]}%")

    for i in range(0, len(X_train), batch_size):
        X_batch = X_train[i:i + batch_size]
        y_batch = y_train[i:i + batch_size]

        model.forward(X_batch, y_batch)
        model.backward(y_batch)

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

We have 200 epochs and at the start of every epoch, we store the cost and accuracy of the model on the **full training set** & **full validation set** to our empty lists of history. Every 20 epochs, we print a summary of the model's cost and accuracy on the training and validation set to give us a progress report.

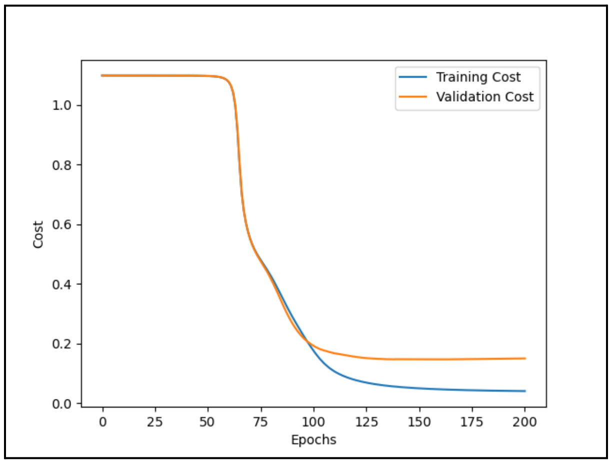
Now we create another for-loop which starts at 0 and stops at the number of examples in the training set. The last argument to the **range()** function is the **step value**. On every iteration, the variable, *i*, will increase by the step value until it hits the **stop value**. During every step, we index for a batch of X and a batch of y, do a forward pass on the batch of data we just indexed, do a backward pass on the same batch of data, and then update the parameters of the model.

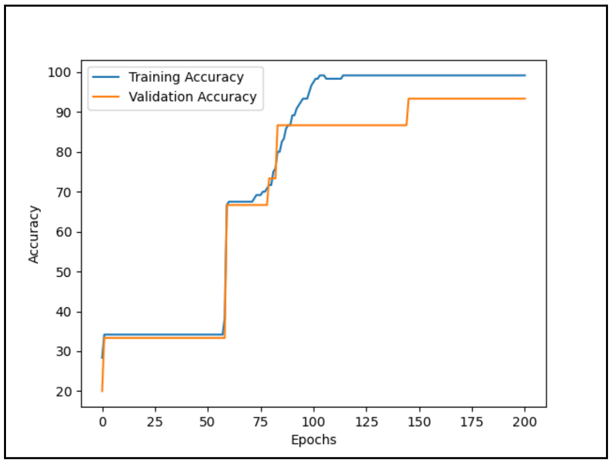
Almost everything is the same. The forward, backward, and update structure of gradient descent stays the exact same. The only difference is the data that we decide to use for the forward and backward passes through the data.

Since the code for checking the ending cost and accuracy on all 3 sets of data (training, validation, and testing) and the plotting for the graphs of the history of the costs and accuracy is the same, we are not going to show the code here to save space. Here is the first and last 3 lines of the terminal and the graphs we have created:

```
120 15 15
Final Cost: 0.039878516784072136 - Final Accuracy: 99.16666666666667%
Final Val Cost: 0.1494242749609158 - Final Val Accuracy: 93.33333333333333%
Test Set Results ~ Cost: 0.15657102882902457 - Accuracy: 93.33333333333333%
```

120 examples in training set, 15 examples validation set, 15 examples in test set





We see that now with only 200 epochs we can fully train a model with Mini-Batch Gradient Descent and that only took less than 5 seconds (less than 10 seconds for slower computers). In reality 200 epochs is a lot for larger models and bigger datasets. If the Iris dataset had more examples where each epoch could contain more steps, we could possibly fully train this model in less than 100 epochs.

It may seem that the model is slightly overfitting and that is true mainly because our training set only contains 120 examples with 4 input features each so a model with 2 hidden layers of 16 hidden units each could easily start to overfit. The best way to fix this overfitting is to get more data for a larger dataset or to lower the number of hidden units in each hidden layer. However, that is for you to try to gain even more experience training models alone.

Here is the disadvantage of using mini-batch gradient descent. We see that the training set has 120 examples and the model's accuracy on the training set has been 99.16666666666667% starting at around 120 epochs. This means that the model has been **incorrectly predicting** the class for **1 example** for **more than 80 epochs**. The Iris dataset is a very easy dataset so why is this the case? Well this is because of the behavior of how we update parameters in mini-batch gradient descent.

Since in every step, we use a **different** batch of data to update the parameters of the model with, the model will never be able to **converge** in the cost function for all the examples in the training set. The model will only be able to **wander around closely** to the **converging** location of the cost function because the **different** batches of data result in **different** updates to the parameters of the model.

These updates aim to lower the **loss** (cost/error/penalty of the model on a mini-batch/subset of the training set) of the current batch itself rather than the **cost** (error/penalty of the model on the whole training set) of the model. However since each mini-batch of data represents a portion of the training set, decreases in the loss can result in decreases in the cost.

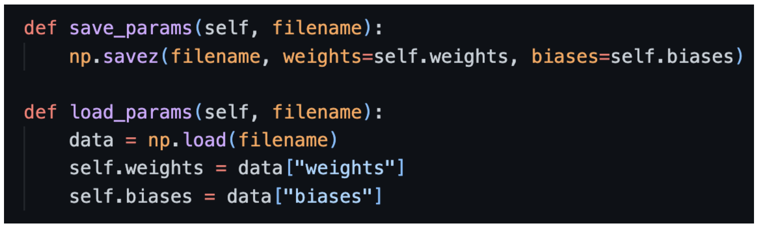
The problem arises towards the end of learning where each mini-batch of data lowers their losses but that may result in an increase in the loss of other batches of data (keeping the model from decreasing the overall cost faster). The best way to combat this disadvantage is to use a learning rate decay which forces the model to slow its updates. Eventually the learning rate decays too close to 0 to actually make any meaningful nudges to the parameters of the model. At this state, you may **assume** that the model has **converged**.

We are now almost done with this multi-class classification problem. Let's say that we will deploy this model into the real world to predict whether an Iris flower is a Setosa, Versicolour, or a Virginica. How do we save the model so we can use it whenever we need to (it would be a waste of time to retrain models)? The solution will be discussed in the next section.

Saving/Loading Models

When we save a model, we are **not only** saving the **model's parameters** (weights & biases) but also we save the **normalization/standardization** function information. For normalization, we would need to save the minimum & maximum value of each input feature in the training set. For standardization, we would need to save the mean & standard deviation value of each input feature in the training set. This way, when we use the model on a new set of data, we can normalize/standardize the data without loading and re-calculating these values.

First let's start with saving & loading the parameters of the model. To do this, we are going to modify the `Dense_Layer` class to include a `save_params` method and a `load_params` method. Here is the code:



```
def save_params(self, filename):
    np.savez(filename, weights=self.weights, biases=self.biases)

def load_params(self, filename):
    data = np.load(filename)
    self.weights = data["weights"]
    self.biases = data["biases"]
```

In the `save_params` method, we require a `filename` parameter which will be the file location where we save the parameters. In the `load_params` method, the `filename` parameter is used to specify the file location and the parameters of that dense layer.

To save both the weights and biases into one file by use `np.savez` which saves them into a **.npz** file. The first argument is the `filename` parameter and then we can pass any number of keyword arguments. The keyword name (`weights`, `biases`) are used to differentiate the 2 different NumPy arrays when we load the file.

To load the file, we use `np.load` which takes in `filename` as its argument. Next, we can get the saved weights array and biases array by simply indexing for the keyword names we used to save the arrays with.

To simulate the process of saving the model's parameters after training, we are going to train the model again and then add a few lines of code after we graph the cost and accuracy graphs to save the parameters and standardization function information.

Thus, we are going to copy and paste the code in the mini-batch gradient descent script except for the `Dense_Layer` class since we need to use the modified `Dense_Layer` class. We are also going to save all the files into a "saved/" subdirectory. To create subdirectories in the current directory our script is in, we can use the `os` module:

```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
import os    # Python Version 3.9.7
np.random.seed(0)    # For repeatability
```

The next piece of code goes after the plotting of the cost and accuracy graphs:

```python
# Save the model
print("\nTraining Complete. Saving Model...")

os.makedirs("saved", exist_ok=True)
layers = ["hidden1", "hidden2", "output"]
for layer, name in zip(model.trainable_layers, layers):
    layer.save_params("saved/" + name + ".npz")

np.savez("saved/standardization.npz", mean=mean, std=std)
print("Model Saved.")
```

First we inform ourselves that we are going to save the model. Next we use `os.makedirs` and pass the argument as the new subdirectory name and the keyword argument `exist_ok` is set to True so that no error comes up if your directory already contains that subdirectory.

Next there is a list of layer names which will serve as the file name we are going to save the parameters to. Hidden layer 1's parameters are saved to "saved/hidden1.npz", hidden layer 2's parameters are saved to "saved/hidden2.npz", and the output layer's parameters are saved to "saved/output.npz". Next, for each layer that we iterate through the zipped actual **layer object** and **layer name,** we call the `save_params` method of the actual layer object.

Lastly, we use `np.savez` again to save the mean and standard deviation of the training set to "saved/standardization.npz" and then print that the model has been saved.

Before moving on to loading the model, we need to make sure that the model we load from the parameters matches the actual model after training. One way we can check is to load the model and get its accuracy on the full dataset. The model's accuracy on the full dataset can be found by looking at the first and last lines of the terminal output:

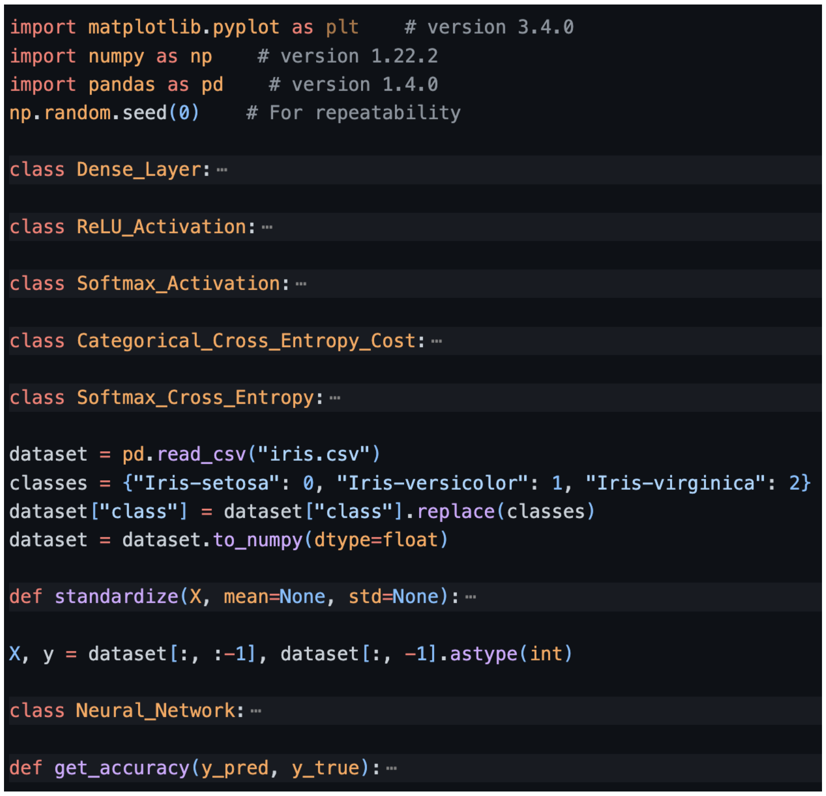
```
120 15 15
Final Cost: 0.039878516784072136 - Final Accuracy: 99.16666666666667%
Final Val Cost: 0.1494242749609158 - Final Val Accuracy: 93.33333333333333%
Test Set Results ~ Cost: 0.15657102882902457 - Accuracy: 93.33333333333333%
```

Training Complete. Saving Model...
Model Saved.

We see that the training set has 120 examples and the model has a $99.1\overline{6}\%$ accuracy on the training set. This means that the model has predicted 119 examples correctly (multiply number of examples by accuracy) from the training set. The validation set has 15 examples and the model has a $93.\overline{3}\%$ accuracy so the model has predicted 14 examples correctly from the validation set. Lastly, the test set has 15 examples and the model has a $93.\overline{3}\%$ accuracy so the model has predicted 14 examples correctly from the test set.

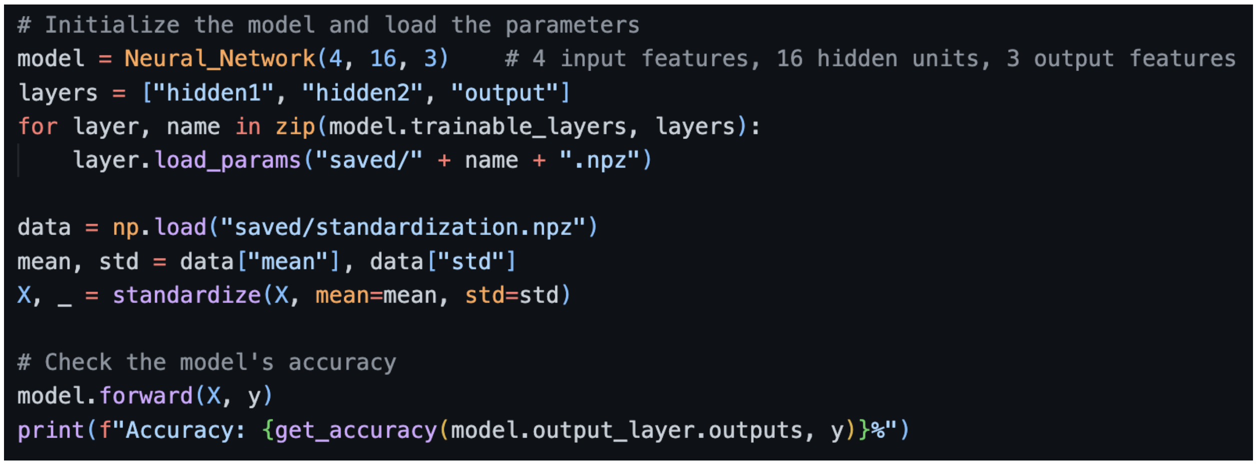
In total, the model has predicted 147 examples correctly out of the 150 examples, from the whole dataset (includes training, validation, and test set) so we should expect a 98% accuracy ($\frac{147}{150} * 100$) from the model on the full dataset.

Here is the code to load the model:



```
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 Softmax_Activation:⋯

class Categorical_Cross_Entropy_Cost:⋯

class Softmax_Cross_Entropy:⋯

dataset = pd.read_csv("iris.csv")
classes = {"Iris-setosa": 0, "Iris-versicolor": 1, "Iris-virginica": 2}
dataset["class"] = dataset["class"].replace(classes)
dataset = dataset.to_numpy(dtype=float)

def standardize(X, mean=None, std=None):⋯

X, y = dataset[:, :-1], dataset[:, -1].astype(int)

class Neural_Network:⋯

def get_accuracy(y_pred, y_true):⋯
```

First we need to import all the required libraries. Next, we copy-and-paste the classes that are parts of the neural network. Then, we load the dataset using pandas, replace the “class” column and convert the pandas object to a NumPy array. Instead of splitting the dataset to training, validation, and test sets, we use the full dataset. Lastly, copy-and-paste the neural network class and function to get the accuracy.



```python
# Initialize the model and load the parameters
model = Neural_Network(4, 16, 3)    # 4 input features, 16 hidden units, 3 output features
layers = ["hidden1", "hidden2", "output"]
for layer, name in zip(model.trainable_layers, layers):
    layer.load_params("saved/" + name + ".npz")

data = np.load("saved/standardization.npz")
mean, std = data["mean"], data["std"]
X, _ = standardize(X, mean=mean, std=std)

# Check the model's accuracy
model.forward(X, y)
print(f"Accuracy: {get_accuracy(model.output_layer.outputs, y)}%")
```

First we have to instantiate a model and for each layer that we iterate through the zipped actual **layer object** and **layer name, we** call the `load_params` method of the actual layer object. Next, we load the standardization function information using `np.load` and index for the keyword arguments, “mean” and “std” that we saved the arrays with. Then, we standardize the whole dataset using the loaded information.

Since the data is now standardized and the model’s saved parameters are loaded, we can call the `forward` method of the model to get its predictions. Lastly, we print the output of our `get_accuracy` function to make sure that the model loaded matches the actual model after training. Here is the output: Accuracy: 98.0%.

That's our model, 98% accuracy on the full dataset!

And that marks the **end** of an **introduction** to AI. See? Calculus concepts in AI can be dropped down to algebra concepts, which isn’t very hard to learn. Now that you have gotten an introduction to AI, you can go create your own AI models.

They might not be very robust since these are basic neural networks but at least you can create your own AI models now. If you would like to continue your journey into AI, you may start AI projects to get more experience in this field or if you think you can handle it, try learning about **deep neural networks**. Best of luck and thank you for reading my book!