Introduction

The Iris Flower Classification is a dataset consisting of different combinations of lengths and widths of a flower's sepal and petal, representing one of three possible flower classifications: Iris-setosa, Iris-versicolor, and Iris-virginica. The objective is to predict the class of a flower given its sepal and petal dimensions. To approximate the function described above, a simple neural network can be used.

What is a Neural Network?

A neural network is composed of multiple layers of neurons. None of the layers have to contain the same amount of neurons as another. There are three types of layers: the input layer, the hidden layers, and the output layer. The input layer, as its name suggests, propagates the input data. In this dataset, the inputs consist of the petal length, petal width, sepal length, and sepal width. The hidden layers integrate the information of the layer before them and propagate their output to the next layer in the series. The output layer undergoes the same procedure as the hidden layers, except it does not propagate forward (since there are no layers after it). Each output is compared to a target value, or the value the output is supposed to be, which is present in the training data. Squaring the difference between the output and the target returns a cost. The most prominent mechanism of the neural network is to minimize this cost (explained in "Backpropagation"). By minimizing the squared difference between the output and the target, the network can more accurately represent any function, thereby making it a "universal function approximator".

Forward Propagation

Before we get started, it is important to note that ***each layer can be thought of as a vector, and each neuron can be thought of as an index in the vector.*** As explained in "What is a Neural Network?", the input layer is composed of the values that are fed into the network. From here on, the procedure is the same. Per layer, each neuron has a unique weight vector, which is multiplied by the input vector calculated in the previous layer. This value is then added to a bias vector, which is also unique to each neuron. The resulting sum is fed through an activation function, which normalizes data to prevent exploding gradients, which are gradients that approach infinity. The output of the activation function is the activation value of the neuron and becomes an input for the next layer (unless it is the output layer).

Cost Function

Once the network propagates the input data forward, it gives an output vector. The target and output vectors are fed through a cost function, which returns a cost vector. As explained in "What is a Neural Network?", each value in the target vector is subtracted from the corresponding value of the output vector. The resulting value is squared and becomes an input in the cost vector.

Backpropagation

Backpropagation is the heart of the neural network. To minimize the cost function, its gradient must be found with respect to every weight and bias in the network. The gradient is subtracted from each of the cost function's components (e.g. gradient with respect to a weight is subtracted from the weight). As the name suggests, backpropagation starts in the output layer and propagates backward towards the input layer. The cost gradient is derived through the use of the chain rule.

Consider the output layer. Let:

i = input vector calculated in the previous layer during forward propagation

Note: i = the previous layer's a

w = weight matrix

b = bias vector

z = neuron's value *before* activation

a = neuron's value *after* activation

c = cost

t = targets

$$\vec{z} = \overrightarrow{(wi)} + \vec{b}$$
$$\vec{a} = \sigma(\vec{z})$$
$$\vec{s} = softmax(\vec{a})$$
$$\vec{c} = (\vec{s} - \vec{t})^2$$

$$(\vec{i},\ w,\ \vec{b}) \rightarrow \vec{z} \rightarrow \vec{a} \rightarrow \vec{s} \rightarrow \vec{c}$$

To get the gradient of the cost function, we must calculate the derivative of the cost function with respect to the weight vector. We implement the chain rule as follows:

$$\frac{\delta \vec{c}}{\delta w} = \frac{\delta \vec{z}}{\delta w}\frac{\delta \vec{a}}{\delta \vec{z}}\frac{\delta \vec{s}}{\delta \vec{a}}\frac{\delta \vec{c}}{\delta \vec{s}} = \vec{i} * \sigma'(\vec{z}) * softmax'(\vec{a}) * 2(\vec{s} - \vec{t})$$

Let the current layer (output layer) have k neurons, and let the previous layer have j neurons. As a result, the weight matrix for the current layer has k rows and j columns. To calculate each index of the weight matrix:

$$\textit{Iterate through all } k, \textit{ and for each iteration of } k, \textit{ iterate through all } j:$$
$$\frac{\delta \vec{c}}{\delta w_{kj}} = i_j * \sigma'(z_k) * softmax'(\vec{a})_k * 2(s_k - t_k)$$

softmax'(a) * 2(s-t) becomes the activation gradient (dc/da) of this layer

The gradient with respect to the bias is the same except the gradient of the output before activation is equal to 1

$$\frac{\delta \vec{c}}{\delta \vec{b}} = \frac{\delta \vec{z}}{\delta \vec{b}} \frac{\delta \vec{a}}{\delta \vec{z}} \frac{\delta \vec{s}}{\delta \vec{a}} \frac{\delta \vec{c}}{\delta \vec{s}} = 1 * \sigma'(\vec{z}) * softmax'(\vec{a}) * 2(\vec{s} - \vec{t})$$

To calculate each index of the bias vector:

$$\textit{Iterate through all } k:$$
$$\frac{\delta \vec{c}}{\delta b_k} = 1 * \sigma'(z_k) * softmax'(\vec{a})_k * 2(s_k - t_k)$$

<u>Consider the hidden layers. Let:</u>

O - k = the layer index, where k is the number of layers before the output layer

i = input vector calculated in the previous layer during forward propagation

Note: *i* = the previous layer's a

w = weight matrix

b = bias vector

z = neuron's value *before* activation

a = neuron's value *after* activation

c = cost

t = targets

For layer O - 1

Note that dc/da for this layer is color-coded in red

$$(\vec{i},\ w,\ \vec{b}) \rightarrow \vec{z} \rightarrow \vec{a}$$
$$\left(\vec{a},\ w^{O},\ \vec{b}^{O}\right) \rightarrow \vec{z}^{O} \rightarrow \vec{s} \rightarrow \vec{c}$$

$$\frac{\delta \vec{c}}{\delta w} = \frac{\delta \vec{z}}{\delta w}\frac{\delta \vec{a}}{\delta \vec{z}}\boxed{\frac{\delta \vec{z}^{O}}{\delta \vec{a}}\frac{\delta \vec{s}}{\delta \vec{z}^{O}}\frac{\delta \vec{c}}{\delta \vec{s}}}$$

$$\frac{\delta \vec{c}}{\delta \vec{a}} = w^{O} * s'\left(\vec{z}^{O}\right) * 2(\vec{s} - \vec{t})$$

For layer O - 2

$$(\vec{i},\ w,\ \vec{b}) \rightarrow \vec{z} \rightarrow \vec{a}$$
$$\left(\vec{a},\ w^{next},\ \vec{b}^{next}\right) \rightarrow \vec{z}^{next} \rightarrow \vec{a}^{next}$$
$$\left(\vec{a}^{next},\ w^{O},\ \vec{b}^{O}\right) \rightarrow \vec{z}^{O} \rightarrow \vec{s} \rightarrow \vec{c}$$

$$\frac{\delta \vec{c}}{\delta w} = \frac{\delta \vec{z}}{\delta w}\frac{\delta \vec{a}}{\delta \vec{z}}\boxed{\frac{\delta \vec{z}^{next}}{\delta \vec{a}}\frac{\delta \vec{a}^{next}}{\delta \vec{z}^{next}}\frac{\delta \vec{z}^{O}}{\delta \vec{a}^{next}}\frac{\delta \vec{s}}{\delta \vec{z}^{O}}\frac{\delta \vec{c}}{\delta \vec{s}}}$$

Note how each dc/dw always starts with (dz/dw) * (da/dz), and how the boxed expressions are always dc/da. A conclusion can be drawn:

$$\frac{\delta \vec{c}}{\delta w} = \frac{\delta \vec{z}}{\delta w}\frac{\delta \vec{a}}{\delta \vec{z}}\frac{\delta \vec{c}}{\delta \vec{a}}$$

The dc/da of this layer contains the dc/da of the next layer. This allows us to form a general statement true for all of the activation gradients in the current layer.

$$\frac{\delta \vec{c}}{\delta \vec{a}} = w^{next} * \sigma'\left(\vec{z}^{next}\right) * w^{O} * s'\left(\vec{z}^{O}\right) * 2(\vec{s} - \vec{t})$$
$$= w^{next} * \sigma'\left(\vec{z}^{next}\right) * \frac{\delta \vec{c}}{\delta \vec{a}^{next}}$$

Since the activation value for this neuron affects each weight in the next layer, along with the activation values and activation gradient of the next layer, we have to sum up all of the activation gradients in the next layer with respect to the neuron in this layer.

Make sure to record the activation gradient for each neuron in the current layer (remember that the current layer has k neurons)

$$\frac{\delta \vec{c}}{\delta a_k} = \sum_{i=0}^{\#neurons\ in\ next\ layer} \left(w_{ik}^{next} * \sigma'\left(z_i^{next}\right) * \frac{\delta \vec{c}}{\delta a_i^{next}} \right)$$

Remember that the current layer has k neurons, and the previous one has j neurons. The weight matrix has k rows and j columns. The general statement for all of the weight values would be:

$$\textit{Iterate through all } k, \textit{ and for each iteration of } k, \textit{ iterate through all } j: \quad \frac{\delta \vec{c}}{\delta w_{kj}} = i_j * \sigma'(z_k) * \frac{\delta \vec{c}}{\delta a_k}$$

Otherwise known as:

$$\textit{Iterate through all } k, \textit{ and for each iteration of } k, \textit{ iterate through all } j: \quad \frac{\delta \vec{c}}{\delta w_{kj}} = i_j * \sigma'(z_k) * \sum_{i=0}^{\#neurons\ in\ next\ layer} \left(w_{ik}^{next} * \sigma'\left(z_i^{next}\right) * \frac{\delta \vec{c}}{\delta a_i^{next}} \right)$$

The gradient with respect to the bias is the same except the gradient of the output before activation is equal to 1

$$\textit{Iterate through all } k: \quad \frac{\delta \vec{c}}{\delta b_k} = 1 * \sigma'(z_k) * \frac{\delta \vec{c}}{\delta a_k}$$

Otherwise known as:

$$\textit{Iterate through all } k: \quad \frac{\delta \vec{c}}{\delta b_k} = 1 * \sigma'(z_k) * \sum_{i=0}^{\#\textit{neurons in next layer}} \left(w_{ik}^{next} * \sigma'\left(z_i^{next}\right) * \frac{\delta \vec{c}}{\delta a_i^{next}} \right)$$

Methods

Regarding the actual method of approach, a matrix class was created. The class contains the following methods:

- Matrix multiplication
- Dot product
- Randomize
- Matrix addition
- Matrix subtraction
- Multiplication/Division by a scalar
- Square values in a matrix
- Apply activation function
- Apply the derivative of the activation function
- Apply softmax
- Apply derivative of softmax
- Get magnitude

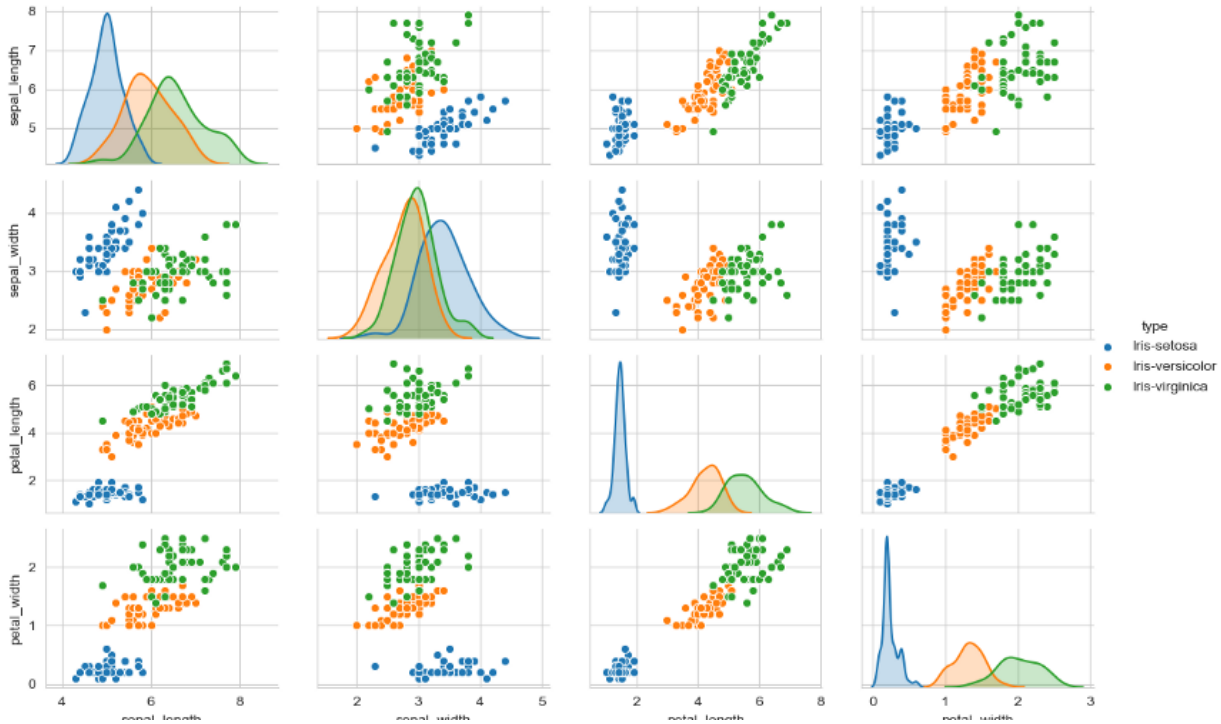
The weight matrices were 2-dimensional, while the layers and biases were 1-dimensional (a matrix with any n amount of rows and 1 column).

With the core logic out of the way, it is time to consider the parameters and functions used for constructing the network. The final model consists of a 3 layer feed-forward architecture with the following layer topology: 4 inputs, 8 hidden neurons, and 3 outputs. The hidden layer uses the Tanh activation function, and the output layer uses Softmax. The training was performed over 5000 epochs (the number of times the neural network backpropagates and updates the weights and biases) with a learning rate of 0.22 and momentum of 0.12; with MSE as the cost function.

Results

The neural network consistently predicts the Iris flower dataset with an average accuracy rate of 99%. Occasionally it will give a lower accuracy rate due to the MSE getting stuck in a local minimum. The neural network matches the setosa species 100% of the time. The error occurs when the network attempts to predict species versicolor or species virginica. The reason for the inaccuracy could be linked to the similarities between the dimensions of both species, thereby producing mixed results.

Here is a visualization of the data:



(Source: https://medium.com/@avulurivenkatasaireddy/exploratory-data-analysis-of-iris-data-set-using-python-823e54110d2d)

In each dot plot, the blue dots (representing Iris-setosa) are separated from the cluster of green and orange dots. This observation most-likely explains the accuracy of the Iris-setosa predictions. The cluster between Iris-versicolor and Iris-virginica would severely hinder the approximated function's ability to differentiate between the two species.

A potential solution to the conundrum is implementing a normalization of the inputs. Having skewed inputs would be a possible cause of gradients that tend toward an input of greater value. Another idea would be to increase the momentum scalar. Doing so would allow the MSE to avoid getting stuck in a local minimum with more ease. An implementation of mini-batch gradient descent (instead of the traditional batch gradient descent used to train this dataset) would also help the MSE in this manner.