**Primer on Artificial Neural Networks**

If the reader is familiar with Artificial Neural Networks, this section can be skipped.

Artificial Neural Networks (ANN) take their name and their inspiration from the human brain, wherein a single neuron is connected to, and therefore, receives and transmits, information to and from other neurons. While modern ANNs have evolved beyond this initial understanding, simpler ANNs, such the Multi-Layer Perceptron used in this paper, are closer in spirit to the original formulation.

# Perceptron and Simple Artificial Neural Networks

In an ANN, we begin with a single neuron, the *perceptron*, which acts, in its simplest form, as linear regression. They are commonly shown in the following pictorial:

A picture containing text, line, past

Description automatically generated

In this diagram, our perceptron recieves from the *Input Layer* , multiplies by to get , then adds a final term to get . Now, if the final step, is the identity function, then we have a simple linear equation and our goal is to solve for , the *weights*, and , the *bias*, to get an output that we would like. This is linear regression. However, the power of perceptrons is that we can vary the piece, the *activation function*, for various needs. A short table is provided with some common activations functions and why they might be used: [1]

|  |  |  |
| --- | --- | --- |
|  |  | This function has a range of , and so is used for probabilities. |
|  |  | This function has a range of , and so is used for probabilities, as well. |
|  |  | Acts as linear regression, but with only positive values. |
|  |  | Linear Regression |

These activation functions allow an ANN to capture non-linearities in the data.

Often, multiple perceptrons (which we will refer to as *neurons* from now on) are used, in which case, we will have a layer of neurons all receiving the input, applying their weights, biases, and activation functions to it, then feeding their results into a final *Output Layer*. Depending on the type of problem, the output layer will have one or more neurons and the activation function will be tailored to the task, either *classification*, wherein we wish to determine which of several *classes* the input belongs to, or *regression*, wherein we wish to provide a continuous value to the input.

From here, we can begin adding more *Hidden Layers* with more neurons, each with their own biases, weights, and activation functions, to capture more difficult and non-linear relationships between the inputs and the targets, thus moving from a simple ANN to a multi-layered *Deep Neural Network* (DNN).

**Deep Neural Networks**

# Part 1: Structure Overview [2]

Formally, we can symbolize the structure of an 𝐿-layered DNN as follows:

Our data has the form 𝑋 = (𝑛𝑥, 𝑚), where 𝑛𝑥 is the number of features of the instances and 𝑚 is the number of training examples, and 𝑌 = (1, 𝑚), a vector of the *actual* values or classifications associated with each in .Given that we have the actual values of 𝑌, this is a *Supervised Learning* problem, wherein we hope to train our DNN from known values to be able to generalize onto new data.

To keep track of our layer and data instance, we use  to denote the 𝑙𝑡ℎ layer and  for the 𝑖𝑡h data instance. Each layer has 𝑛[𝑙] neurons. We will use vectorized notation to talk about all the weights, biases, and operations taking place in each layer.

Define 𝐴[𝑙] as the output of layer 𝑙 over all instances 𝑖. This means that 𝐴[0] = 𝑋. Let 𝑊[𝑙] be the matrix of all the weights of the 𝑙𝑡ℎ layer and 𝑏[𝑙] the matrix of all the biases of the 𝑙𝑡ℎ layer. 𝑊[𝑙] is an (𝑛[𝑙], 𝑛[𝑙−1]) matrix (note that the number of columns of this matrix is the number of neurons from the previous layer, as the information layer 𝑙 will receive is the outputs of each of the 𝑙 − 1𝑡ℎ layer’s neurons), and 𝑏[𝑙] is an (𝑛[𝑙], 1) matrix.

Then, our DNN looks like:

Here,   are the *predictions* our DNN has given us for each of our instances.

In more succinct form, we define the output of layer 𝑙 as: [3]

and our DNN can be written as:

where the  is *composition* operation. (*Machine Learning in Finance*, page 113).

# Part 2: Feedforward Step

We call this process of feeding into the *Feedforward Step* and results in our first attempt at predictions,  .

From this, we can identify the trainable *parameters*, which are: [2]

Our goal is to update to provide increasingly accurate predictions.

For this, we need the *Cost Function*: [2]

Here, is the *Loss Function* and depends on the application, i.e., how we determine the performance of our model. Essentially, we are averaging over how close our models’ predictions, , are to the known answers, . Now, the better we are, the smaller this value will be. Our goal is thus to update our parameters to *minimize* .

To do this, we will run through the process of *Back-propagation* using *Gradient Descent.*

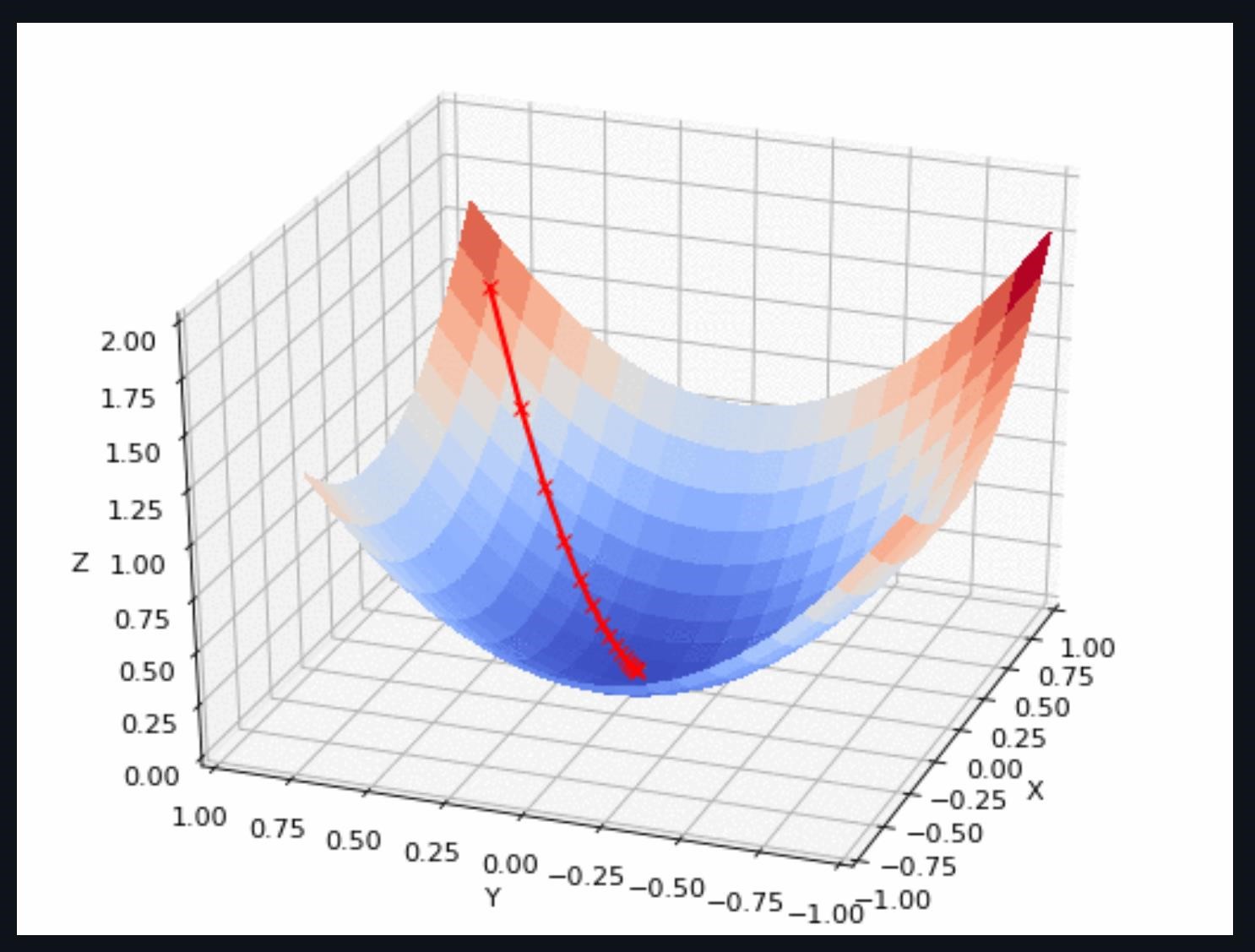
# Part 3: Back-propagation and Gradient Descent

We begin with a feedforward step, passing our 𝑋 into our DNN for the 𝑘𝑡ℎ time, giving us  , our output from pass 𝑘. We then wish to update our weights and biases for all layers based on this information.

We determine the iterative change in the weights and biases with the following equation: [2, 4]

In the above equations, is the set of weights associated with the neuron of the layer during the feedforward iteration, and is the bias of the same neuron in the same iteration. We add a term, , which is the *Batch Size*, or the number of instances to compute before updating the parameters.If 𝑠𝑘 = 1, we have simple *Stochastic Gradient Descent*. If it is more than 1, we have *Batch Gradient Descent*. To avoid overshooting the minimum, we include a decay term, , which will typically decrease as the number of iterations increases.

In words, for each training instance, we perform a forward pass to get , measure the average error, then determine the contribution to the error made by each neuron (so, the weights and biases of that neuron) by finding the gradient of the loss function with respect to each set of weights and biases. From here, we reduce the weights and biases of that neuron by our algorithm above. Thus, we wander around the function until we find a local minimum, which, from above, will mean that our   is as accurate as can be.

Visually, we have the following, where the contour function is our 

In this image, we have the loss function, and we are attempting to find the bottom, the deep-blue region. Our steps are the red x marks, and we see that we choose to step *down* (hence the negative in our update equations) in the direction of steepest slope. Also, we notice that the steps become smaller, a result of our 𝑡𝑘, to prevent us from accidentally beginning to climb up the other side.