**Taking a Step Back**

**The Perceptron**

* Simplest type neuron
* Specifically, supervised algorithm for binary classification
  + Means it’s used to determine whether a set of inputs belong to a certain class or not
* Goal: To learn a threshold
* Mathematically, expressed as follows

A close up of a clock

Description automatically generated

* If the sum of the weights + bias (= -threshold) is greater than 0, the neuron “fires” and it outputs 1
  + I.e. a positive instance
* Else, neuron assigns it a value of 0
* Notice that each feature of the input vector gets multiplied by a corresponding weight
* This makes sense because we want to weight certain elements of an input vector more than others.
  + For example, if I’m building a model that determines whether I will be sick or not that is dependent on who else is sick, th4e weather, and what I’ve been eating
  + I will want to weight if other people are sick more than the other two
* **Linear separability**
  + A property of two sets of points
  + Two sets are defined as linearly separable if there is a line such that one set of points lies completely on one side and the other set of points lie on the other side A picture containing object, clock

    Description automatically generated
* Perceptron algorithm will still work for linearly inseparable vectors
  + But it will never reach a point where **every single vector is classified correctly**
* The weights in a perceptron are updated 
* Where dj is the label and yi(t) is the prediction generated by the model
* The limitation of the perceptron is that **its output is completely binary**
* But what if we output for small changes in the input to affect the output?
  + What if a small change in an input feature will increase the likelihood of an output being a positive instance without guaranteeing that it becomes 1?
  + Meaning what if an input feature increases the output from 0 to 0.5 – this can’t be done with a simple perceptron
* Solution? The sigmoid neuron (or another activation function)

**Sigmoid Neuron**

* Takes the output of the neuron z (sum of the input features \* weight + bias) and applies the following formula:
* This allows the output of the neuron to be a probability distribution between 0 and 1
* As z becomes very large, the output will be 1 and as z becomes very small, the output will be 0 which makes sense

**Backpropagation**

**Clearest explanation in series:** <https://www.youtube.com/watch?v=Zr5viAZGndE&feature=youtu.be>

* For large artificial neural networks, backpropagation is used to update the weights
* It’s a recursive application of the chain rule, which allows us to compute the derivative of the loss with respect to each weight
* To do this:
  + Represent your network as a computation graph
  + At each node – where some arithmetic is applied – assign a variable like q or z etc. and compute the derivative of these intermediate variables with respect to the input
  + Start at the end and work your way backwards

A picture containing clock, meter

Description automatically generated

* Idea is that we have some number that represents the gradients with respect to the direct output
* And at each node, we can backtrack using the chain rule by multiplying this value we found from previous nodes (dL/dz) with the derivative of the direct output z with respect to each input
* And then we send this value back to the immediate surrounding
* Reason this is so much simpler:
  + We’re only ever working with local gradients and using the chain rule to multiply it with incoming gradients
  + Did not have to derive a complicated expression for the gradient from the final output layer through intricate expressions
* **IMPORTANT NODE** 
  + When building these computation graphs, can choose how complicated/simple the graph can be
  + Can choose to write the graph at the level of the simplest possible computations e.g. addition, multiplication, exponentiation etc.
* Patterns in backward flow:
  + **Add gate:** gradient distributor
  + **Max gate:** gradient router (i.e. sends all of the gradient to one of the variables and 0 to the other variable)
  + **Multiplication gate:** gradient switcher (take the upstream gradient and scale it by the input of the other branch

A picture containing game

Description automatically generated

* What happens when our inputs are now vectors?
  + The local gradients (dz/dx) shown in the images above now become the Jacobian matrix
    - **Jacobian matrix –** matrix representing the first partial derivatives with respect to each element A screenshot of a cell phone

      Description automatically generated
    - Can see from the example above that the number of columns become the number of variables that each function (in each row) has
  + \*\* NOTE the gradient of a vector is always going to be the same size as the vector
  + Each element in the gradient vector shows how much this individual element affects our final output
  + In the implementation of a graph structure, we build individual nodes that implement the forward() / backward () API
    - Forward computes the operation and saves any necessary intermediates needed for backprop
    - Backward applies chain rule to compute the gradient of the loss with respect to each of the inputs

**Neural Networks**

* Type of function where we stack simpler functions in order to create a more complex function representation
* An n-layer neural net does not mean the network has n hidden layers – means model has n layers of linear computations
  + Thus a 2 layer neural net has 1 hidden layer
  + A screenshot of a cell phone

    Description automatically generated3 layer neural net has 2 hidden layers etc.
* **DISCLAIM:** All of this assumes input data has features arranged by rows – for a more concrete and clear explanation scroll below
* **Key point:** Imagine for one instance what the dimension of the input layer would be
* In the picture above, each input vector has 3 elements thus it would be a 3 x 1 matrix
* **For m samples of input data, the dimension of the input vector becomes 3 x m**
  + We accelerate computations by combining multiple computations with matrices
  + We could represent weights as individual numbers and loop through the number of samples but this is **more computationally expensive**
* Thus the dimension of the input data: **number of features x size/number of input data** 
  + E.g. if you had 3 features, weight, height, and IQ for 100 students
  + Dimension of the input would be 3 x 100
* For one piece of input data (with however many features), the dimension of the (output of the) hidden layer is number of nodes by 1
  + For m pieces of data, the dimension of the (output of the) hidden layer
* At any hidden layer, the dimension is = **no. of nodes x no. of input data**
* Recall h1 = activation function((W.x) + b))
* **IMPORTANT CLARIFICATION POINT:**
  + The above is the DIMENSION OF THE WEIGHTS of the first layer
  + It’s NOT the dimension of the output of the first hidden layer
  + Output of the hidden layer can be calculated as follows:
  + Remember the calculation applies is h1 = W . x
* The dimension of the weights of the first hidden layer becomes m x n as such:
  + m: **number of nodes**
  + n: **number of features**
* Dimension of the bias is the same as the dimension of the output hidden layer (since bias is added to W.x)
* This can be done for all of the layers and eventually the output layer
* Just as a disclaimer, you should never have to include the sample size in any of the dimension of the weights, hidden layers, output etc.
* Using the below equations – designed so that you never have to include sample size

**Matrix Multiplication Explained Finally**

<https://medium.com/from-the-scratch/deep-learning-deep-guide-for-all-your-matrix-dimensions-and-calculations-415012de1568>

* There are two ways of arranging input data:
  + Arranging the features as rows and columns as samples

A screenshot of a cell phone

Description automatically generated

* + This is the equivalent of:

|  |  |  |
| --- | --- | --- |
| **Height** | 168 | 180 |
| **IQ** | 120 | 115 |

* + If the data is arranged like this – use Y = W.X + B
  + **Important Point:** When used in this format, the dimension of the hidden layer is (hidden nodes by 1)
* The second way to arrange is by arranging the features as the columns

A screenshot of a cell phone

Description automatically generated

|  |  |
| --- | --- |
| **Height** | **IQ** |
| 168 | 120 |
| 180 | 115 |

* If the data is arranged like this, do Y = X . W + b
* **Important Point:** When used in this format, the dimension of the hidden layer is (1 by hidden nodes)