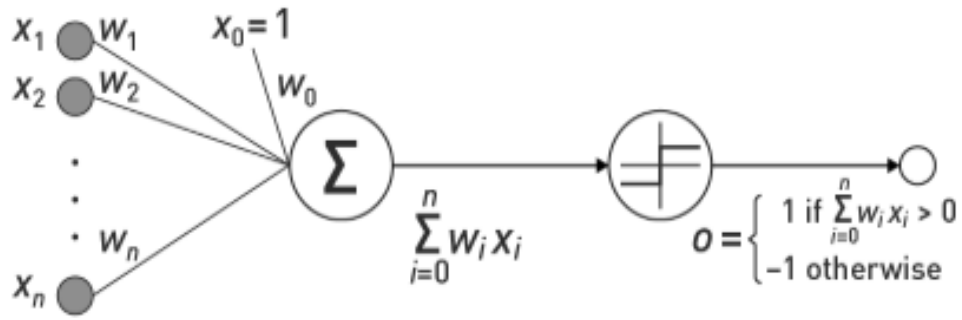
Neural Nets

Nishanth Nair

March 20,2020

Overview



$$o(x_i, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + \ldots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

Figure 1: Basic Perceptron model

- In a simple perceptron,
  - Inputs **X** have a weight **W** and a bias term W_0
  - The perceptron moddel is : $\sum W.X + W_0$
  - **W** implies how much weight to give a particular input.
  - **W_0**, the bias,can be considered the threshold the other inputs need to reach before having any effect.
  - For example, if $W_0 = 10$, the effects of $\sum W.X$ wont start, unless its value surparsses 10.
  - Once the value is surparssed, the effect is solely based on the value of **W**.
  - Thus the term bias.
  - W_0 allows for non-zero threshold
- **Discriminative classifier**: Learns decision boundary
- Perceptron fires if predicted value is positive
- Difference between logistic regression and neuron:
  - Logistic regression uses a linear activation like sigmoid

  - Neuron uses a non-linear activation like a step function. This allows to model non linear functions.
- Single perceptron does not work for data that is not linearly seperable. The XOR problem.
  - Solution: To use more than 1 perceptron

Learning Weights

- Given that we have input and output (i.e training data), how do we learn the appropriate weights?
- There are several ways:
  - Rosenblatts algorithm (aka perceptron training rule)
  - Winnow: binary attributes, multiplicative weights, adjustable threshold
  - Gradient descent

Rossenblatts Algorithm or Training Rule

```
initialize weights randomly;
while <termination condition> not met:
    initialize Δw_j = 0
    for each training example:
        compute ŷ_i
        for each weight, w_j:
            Δw_j = Δw_j + η.(y_i − ŷ_i).x_i
    for each weight, w_j:
        w_j = w_j + Δw_j
```

- Error driven algorithm: Only updates on failure
- Online algorithm: only considers one instance at a time
- Guaranteed to converge if solution exists, i.e
  - Training data is linearly separable (i.e., no noise).
    - Otherwise it fails. Major limitation.
  - Learning rate is sufficiently small.
    - In the proof, it has to be infinitesimally small.

Gradient Descent

- If you ignore output step function of perceptron, the output can be written as: $\hat{Y}_i = \sum_{i=0}^{n} w_i.x_i$
- Error is a quadratic function: $J(\theta) = \frac{1}{2n}\sum_n (y_i - \hat{y}_i)^2$
- Since we have a convex function, we can use gradient descent. Thus, update rule is:

$$\theta_j = \theta_j - \alpha.(y_i - \hat{y}_i).x_i$$

Training rule vs Gradient Descent

- Perceptron:
  - $\hat{y}_i$ is a step function:either 0 or 1.
  - Guaranteed to work if data are linearly separable
  - Requires sufficiently small learning rate.
- Gradient descent:
  - $\hat{y}_i$ is a smooth function:continuous.
  - Guaranteed to converge to minimum error
  - Requires sufficiently small learning rate.
  - Works when data contain noise
  - Works when data are not linearly separable

Distributed Perceptron

The algorithms listed above work on single machines. How do we distribute this workload for large scale data?

Four possible options:

- Serial (all data):
  - The classifier returned if trained serially on all the available data (e.g., on a single computer); mistake driven
- Serial (subsampling):
  - Where we shard the data, select one shard randomly, and train serially
- Parallel (parameter mix):
  - Learned locally on each shard; parallel strategy with uniform mixing
- Parallel (iterative parameter mix):
  - Parallel strategy with averaged weight redistributed on iteration

The approach **Iterative parameter mixing** works best in a distributed environment. The approach is as follows:

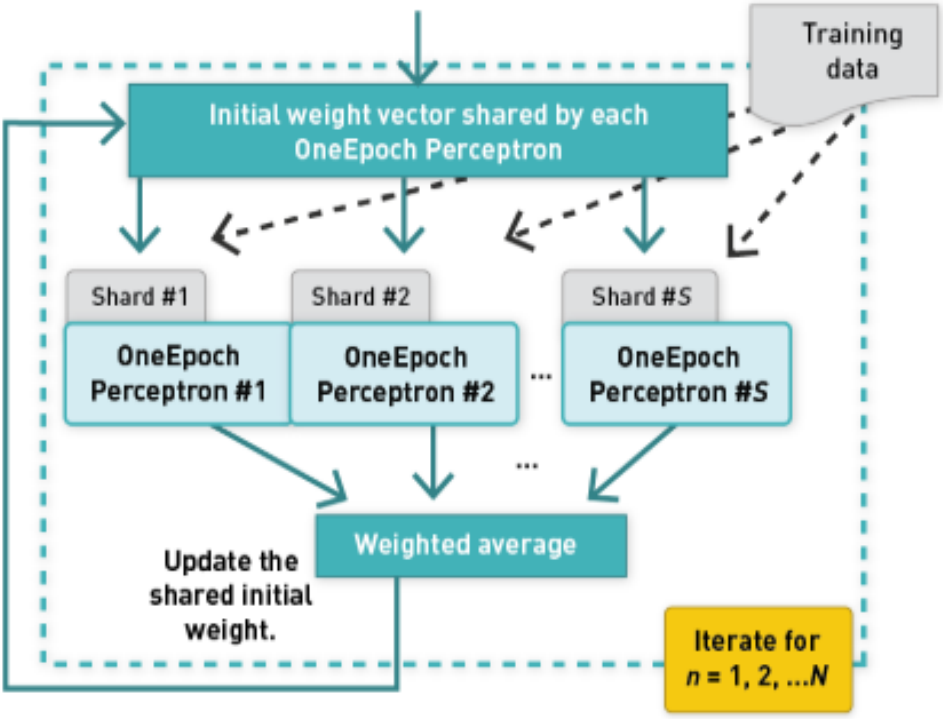


Figure 2: Iterative parameter mixing

- Divide the data into N shards
- Run one epoch locally in each shard to get N perceptrons
- Compute the weighted average of the weights.
- Update initial weight vector and run the next epoch and so on until convergence.

Some additional points to note:

- Guaranteed to converge and separate the data (if separable)
- Results in fast and accurate classifiers
- Trade-off between high parallelism and slow convergence.
  - The more shards you have,the slower the convergence.
  - Example depicted below:

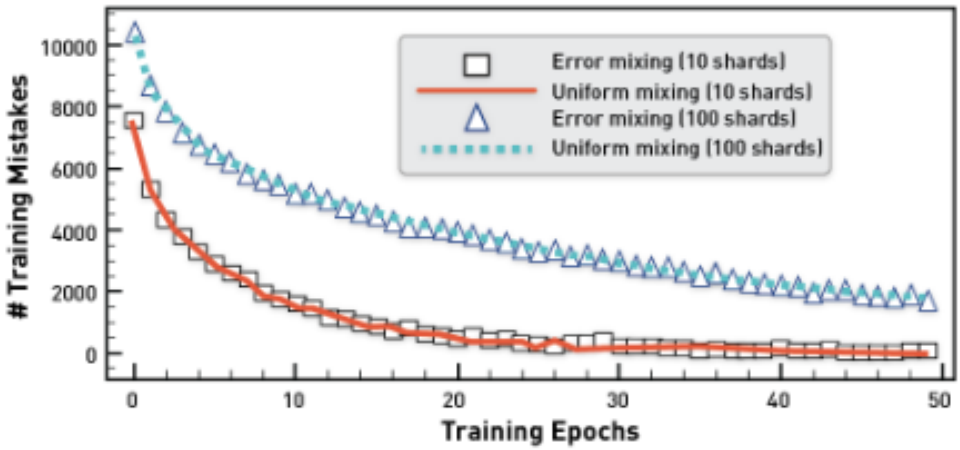


Figure 3: Iterative parameter mixing - Different shard sizes

Multilayer Networks

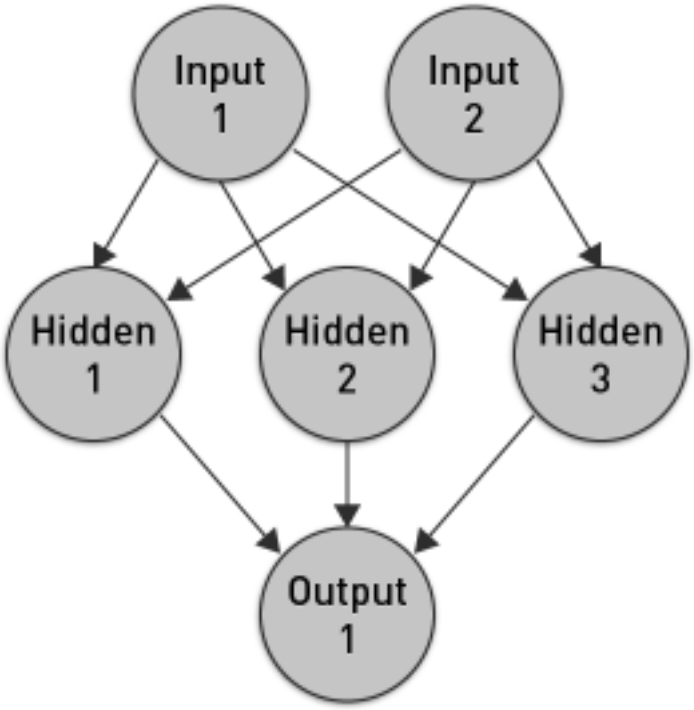


Figure 4: Basic 2-Layer model

- Two layer network: Beacuse it has 2 layers of weights
- Two-layer networks are universal function approximators: can approximate any function (universal approxiation theorem)

- Can solve XOR problem
- A neural network is considered a deep neural network if it contains 2 or more hidden layers.

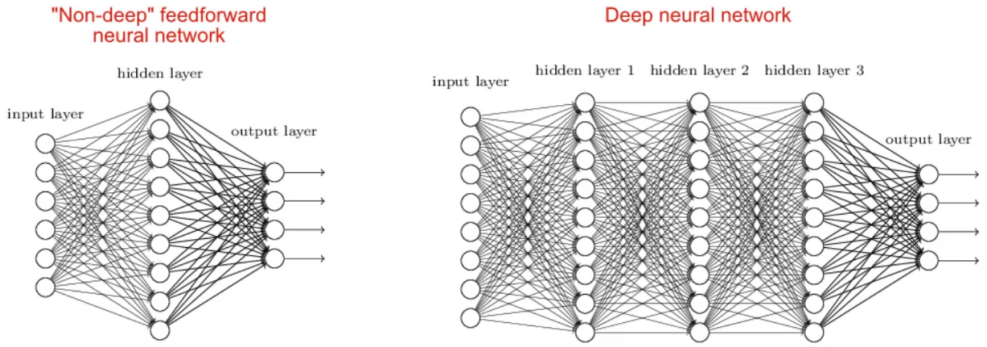


Figure 5: multilayer-networks

Universal Approximation Theorem

Let F be a continuous function on a bounded subset of D-dimensional space. Then there exists a two-layer neural network F' with a finite number of hidden units that approximate F arbitrarily well. i.e.

$$\forall x \in \text{domain of } F, |F(x) - F'(x)| < \epsilon$$

Activation Functions

- Assume, activation function is f(Z), where, $Z = \sum w_i.x_i + W_0$
- In classification tasks, its useful to have all outputs fall between 0 and 1.
- These are called activation functions. These can then be used for probability assignments for each class.

Sigmoid or Logistic function

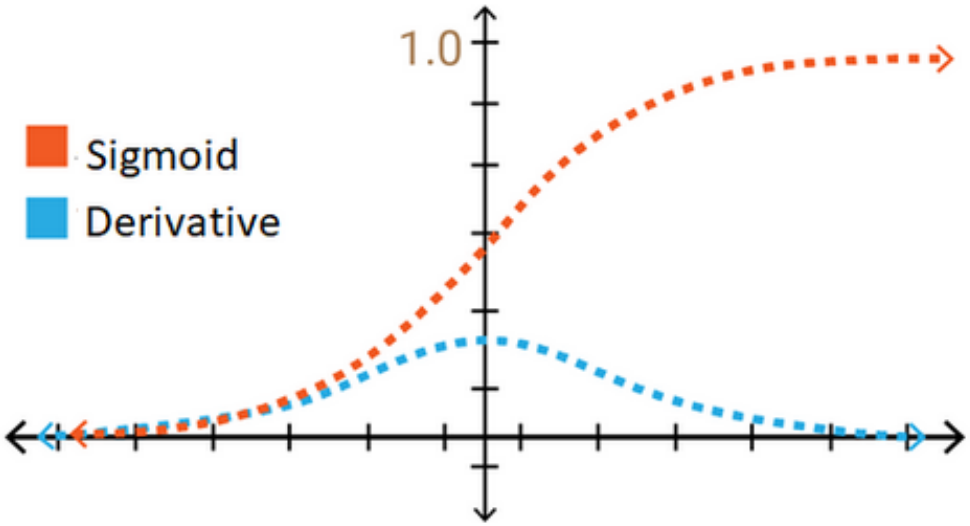


Figure 6: Sigmoid Function

- **Function**: $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Range**: (0,1)
- **Advantages**
  - Smooth Gradient, preventing jumps in output values
  - Output values bound between 0 and 1, normalizing the output of each neuron.
  - Clear predictions
    - For X above 2 or below -2, tends to bring the $\hat{Y}$ value (the prediction) to the edge of the curve, very close to 1 or 0.
    - This enables clear predictions.
- **Disadvantages**
  - Vanishing gradient:
    - For very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem.
    - This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.
  - Outputs not zero centered
  - Computationally expensive

Hyperbolic Tangent function

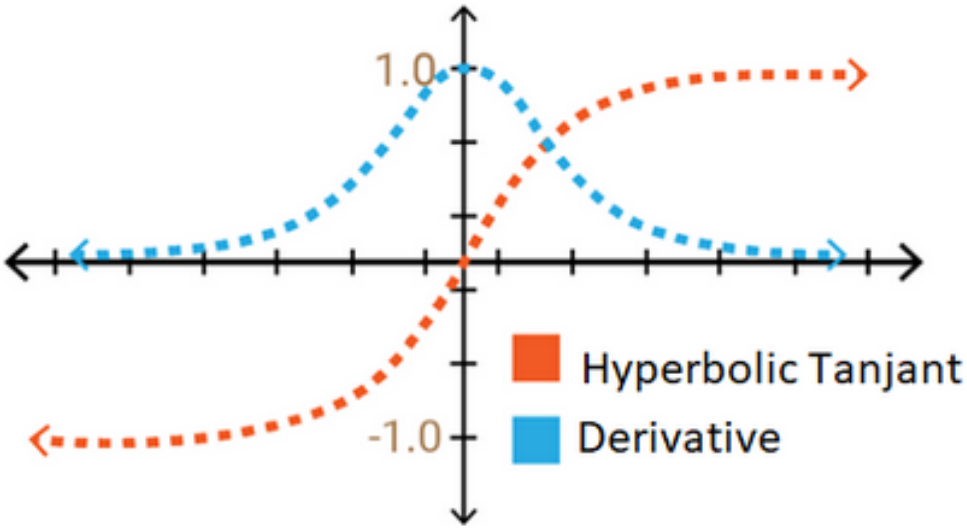


Figure 7: Hyperbolic Tangent Function

- **Function**: $tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- **Range**: (-1,1)
- **Advantages**
  - Zero centered—making it easier to model inputs that have strongly negative, neutral, and strongly positive values.
  - Similar to sigmoid.
- **Disadvantages**
  - Same as sigmoid.

Rectified Linear Unit (ReLU)

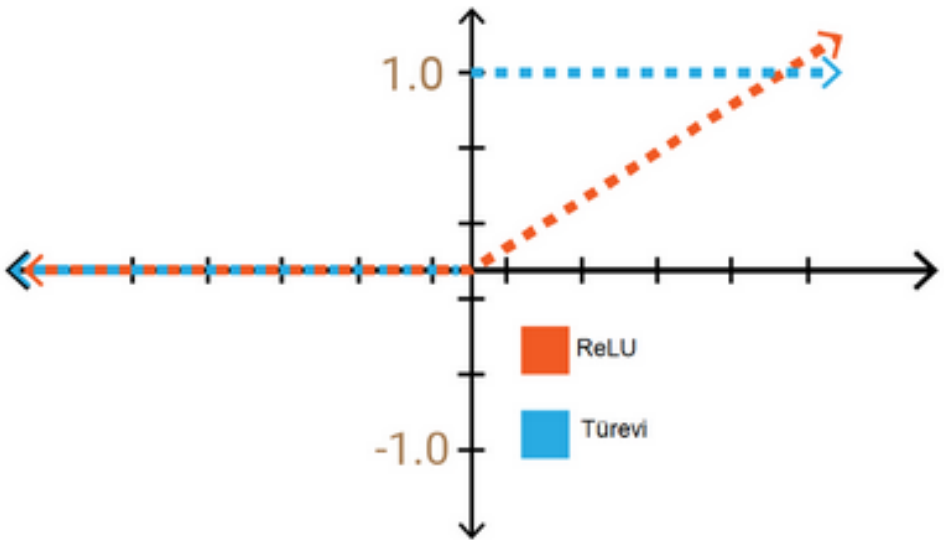


Figure 8: ReLU Function and derivative

- **Function**:

$$f(x) = \begin{cases} 0 & \forall x < 0 \\ x & \forall x \geq 0 \end{cases}$$

- **Range**: $[0,\infty]$
- **Advantages**
  - Computationally efficient—allows the network to converge very quickly
  - Non-linear—although it looks like a linear function, ReLU has a derivative function and allows for backpropagation
- **Disadvantages**
  - The Dying ReLU problem : when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.

Leaky ReLU

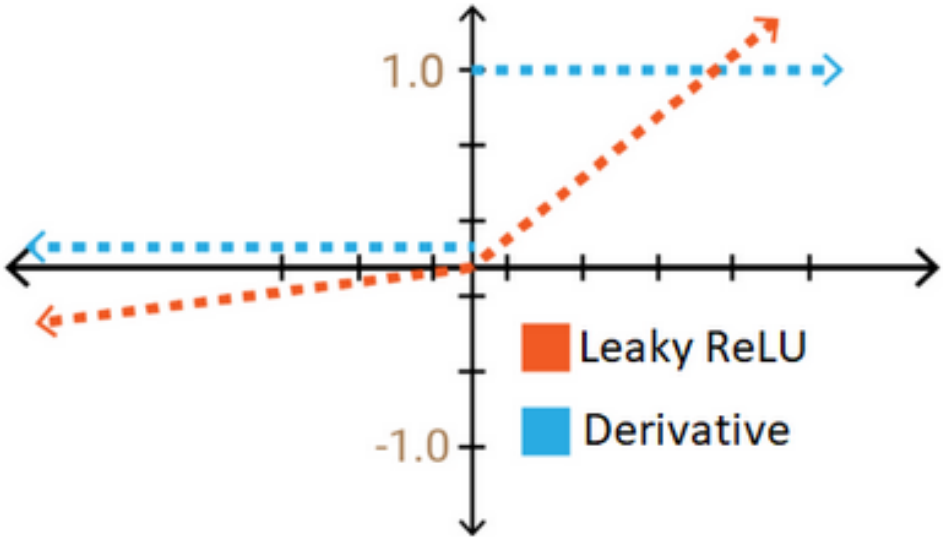


Figure 9: Leaky ReLU Function and derivative

- **Function**:

$$f(x) = \begin{cases} 0.01 \times x & \forall x < 0 \\ x & \forall x \geq 0 \end{cases}$$

- **Range**: $(-\infty, \infty)$
- **Advantages**
  - Prevents dying ReLU problem—this variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values
  - Similar to ReLU
- **Disadvantages**
  - Results not consistent—leaky ReLU does not provide consistent predictions for negative input values.

Softmax Function

- **Function**: $\sigma(y)_i = \frac{e^{y_i}}{\sum_k e^{y_k}}$, where, i = 1...k and y = $(y_1, y_2..y_k)$
- **Range**: (0,1)
- **Advantages**

- Able to handle multiple classes(vs Only one class in other activation functions.
- Normalizes the outputs for each class between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific class.
- Useful for output neurons: typically Softmax is used only for the output layer, for neural networks that need to classify inputs into multiple categories.

Swish Function

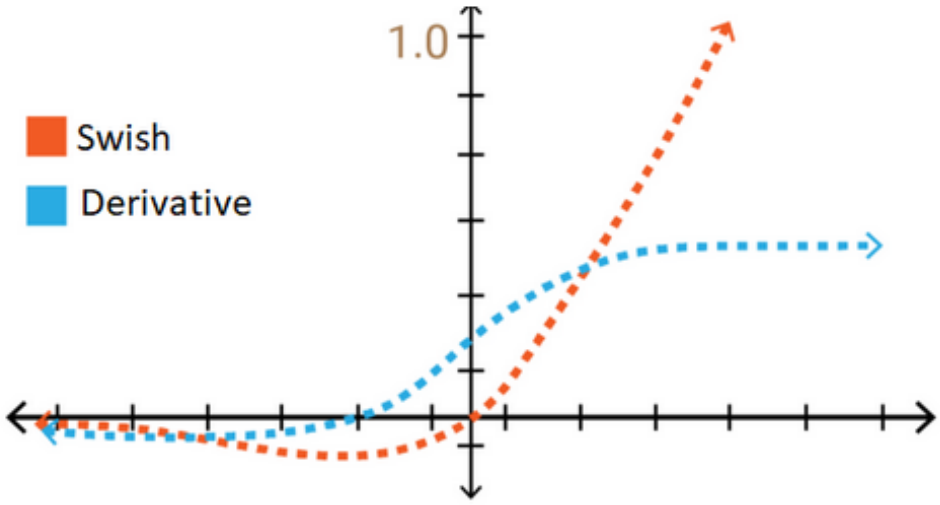


Figure 10: Swish Function and derivative

- **Function**: $f(x) = x.sigmoid(x) = \frac{x}{1+e^{-x}}$
- **Range**: $(-\infty, \infty)$
- **overview**
  - A new, self-gated activation function discovered by researchers at Google. (self-gated means that the gate is actually the sigmoid of activation itself.)
  - Tends to work better than ReLU on deeper models across a number of challenging data sets.
  - Similarity to ReLU make it easy for practitioners to replace ReLUs with Swish units in any neural network.

Multi-class networks

Two types of multi-class networks:

- Non-Exclusive classes
  - A single datapoint can have multiple classes aasigned
  - Eg: photos can have mmultiple tags
- Mutually Exclusive classes
  - Only one class per data point
- Easiest way to handle multi-class is to have one output node per class:

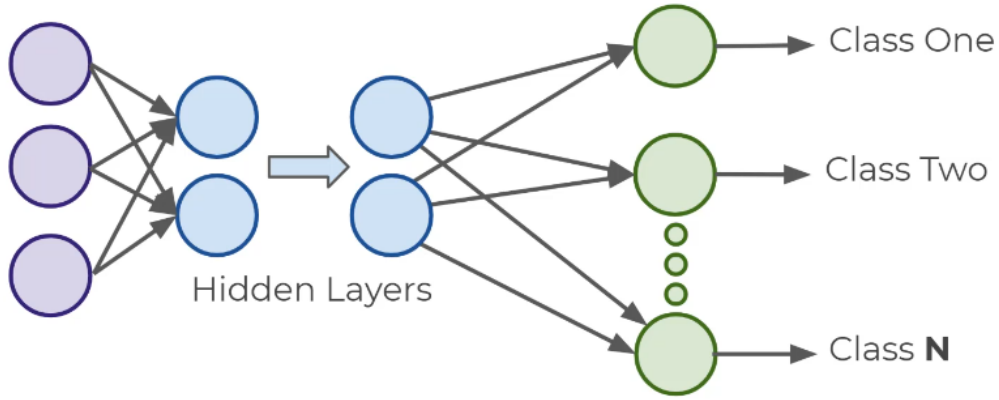


Figure 11: Multi-class network

- **One hot encoding**:
  - Data needs to be prepped appropriately to handle multi-class outputs.
  - We cannot use strings like "red","blue" etc.
  - Instead, they will need to be **one-hot encoded**.
  - That is, each value should be transformed to its own column having either a 1 or zero as value.
  - Below are 2 examples:

Data Point 1	RED
Data Point 2	GREEN
Data Point 3	BLUE
...	...
Data Point N	RED

	RED	GREEN	BLUE
Data Point 1	1	0	0
Data Point 2	0	1	0
Data Point 3	0	0	1
...	...	...	...
Data Point N	1	0	0

Figure 12: One-hot encoding for mutually exclusive classes

Data Point 1	A,B
Data Point 2	A
Data Point 3	C,B
...	...
Data Point N	B

	A	B	C
Data Point 1	1	1	0
Data Point 2	1	0	0
Data Point 3	0	1	1
...	...	...	...
Data Point N	0	1	0

Figure 13: One-hot encoding for Non-exclusive classes

- Activation Function for multi-class:
  - Non-Exclusive
    - Use sigmoid.
    - Each neuron outputs a value between 0 and 1 indicating probability of that class.
    - Using a threshold, say 0.5, datapoint is assigned all classes above that threshold.

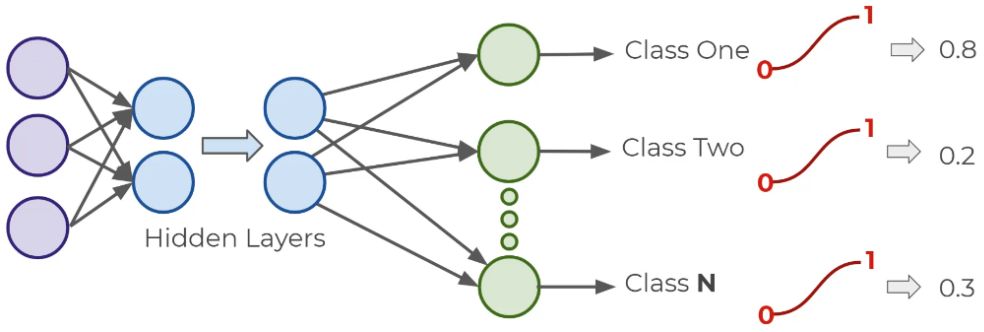


Figure 14: Activation for non exclusive classes

- Mutually exclusive:

$$\text{Use softmax, } \sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}, \text{where, } i = 1...k$$

  - k = Numbler of classes
  - Softmax computes probability distribution of a class over k different classes.
  - Sum of all probabilities will be equal to 1. Range is 0,1
  - Example, [red,blue,green] = [0.1,0.6,0.3].
  - Choose class with highest probability.

Learning the weights: Backpropogation

- For perceptron, we ignored step function to use gradient descent.
- In a multi-layer network, if we ignore the activation, we end up with a linear function.
- Non linear step function is not differentiable thus gradient descent fails.
- Nonlinear, differentiable function required for backpropagation.
  - Sigmoid Function
    - $\sigma(x) = \frac{1}{1+e^{-x}}$
    - Based on biology of neurons
    - Nice property of derivative: $\frac{d\sigma(x)}{dx} = \sigma(x).(1 - \sigma(x))$

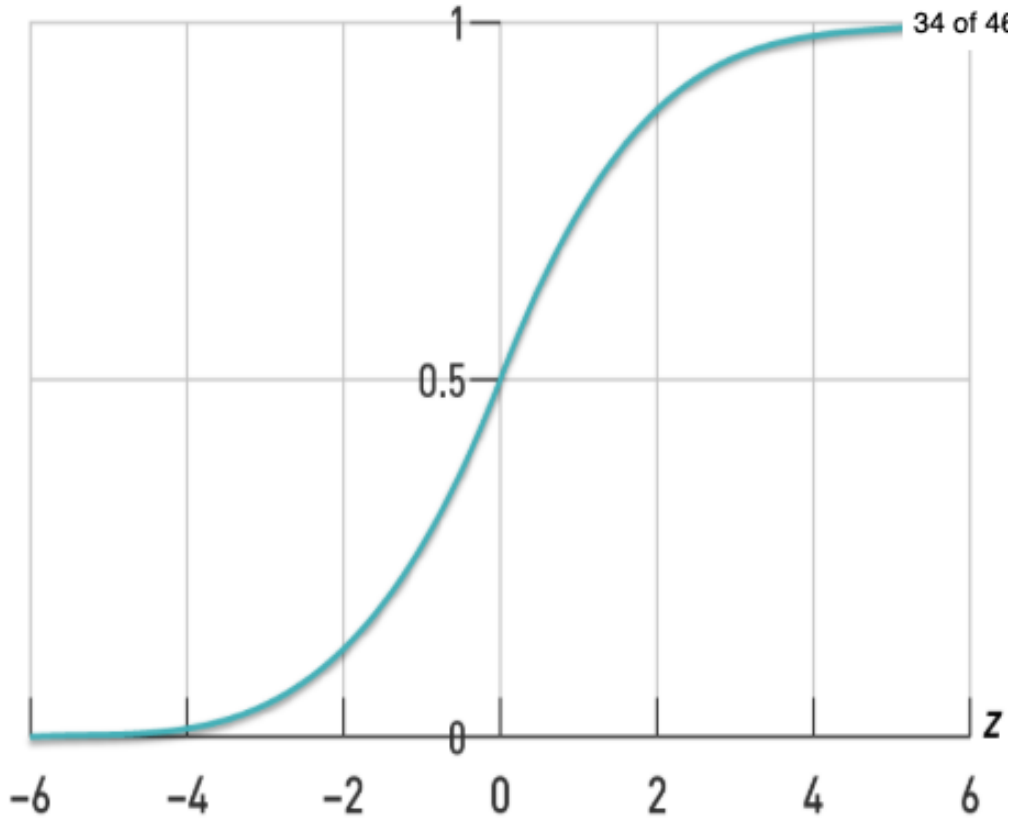


Figure 15: Sigmoid Function

  - Hyperbolic Tangent Function

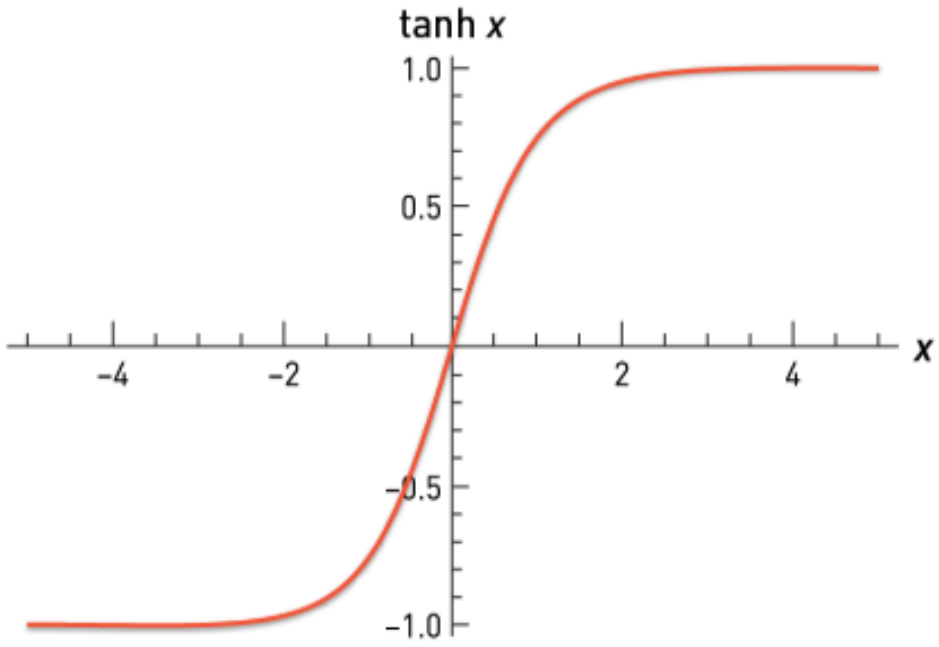


Figure 16: Tanh Function

– Anyother function having a similar shape.

Why Backpropogation?

- In a perceptron, We know what the correct value y_i is thus what the predicted value $\hat{y}$ should be.
- In a multilayer network,
- We know what the correct value y_i is thus what the predicted value $\hat{y}$ should be only for the output layer.
- For the hidden layers, we don't know what that output of each layer should be, and thus the regular approach of finding the weights doesn't work.
- This is what backprop is designed to solve.

Backpropogation Intiution

Computational Graphs

Computational graphs are a nice way to think about mathematical expressions.

For example, consider the expression $e = (a+b) \times (b+1)$.This can be broken down as follows:

$$c = a + b$$
$$d = b + 1$$
$$e = c \times d$$

The computational Graph would be as represented:

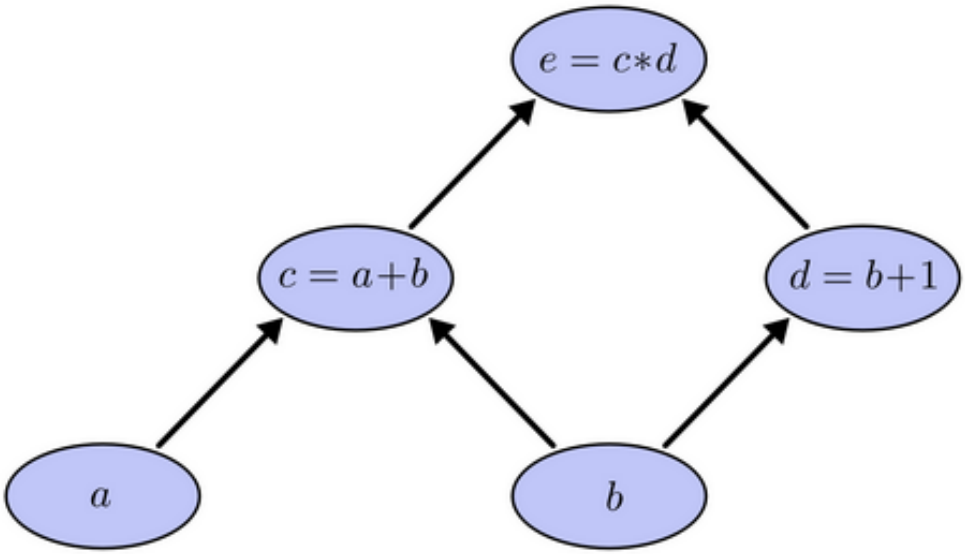


Figure 17: Computational graph for $e = (a + b) \times (b + 1)$

These sorts of graphs come up all the time in computer science, especially in talking about functional programs. They are very closely related to the notions of dependency graphs and call graphs. They're also the core abstraction behind the popular deep learning framework Theano.

Derivatives on Computational Graphs

If one wants to understand derivatives in a computational graph, the key is to understand derivatives on the edges.

If a directly affects c, then we want to know how it affects c.

If a changes a little bit, how does c change?

We call this the partial derivative of c with respect to a.

To evaluate the partial derivatives in this graph, we need the **sum rule** and the **product rule**:

$$\frac{\partial(a+b)}{\partial a} = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1 \text{ , (Sum Rule)}$$

$$\frac{\partial(uv)}{\partial u} = u.\frac{\partial v}{\partial u} + v.\frac{\partial u}{\partial u} = v \text{ , (Product Rule)}$$

Now, In our example, $e = (a+b) \times (b+1)$, if we compute derivatives on the computational graph, we get:

$$\frac{\partial c}{\partial a} = 1$$

$$\frac{\partial c}{\partial b} = 1$$

$$\frac{\partial d}{\partial b} = 1$$

$$\frac{\partial e}{\partial c} = d$$

$$\frac{\partial e}{\partial d} = c$$

Lets assume, a = 2 and b = 1, The computation graph can now be represented as:

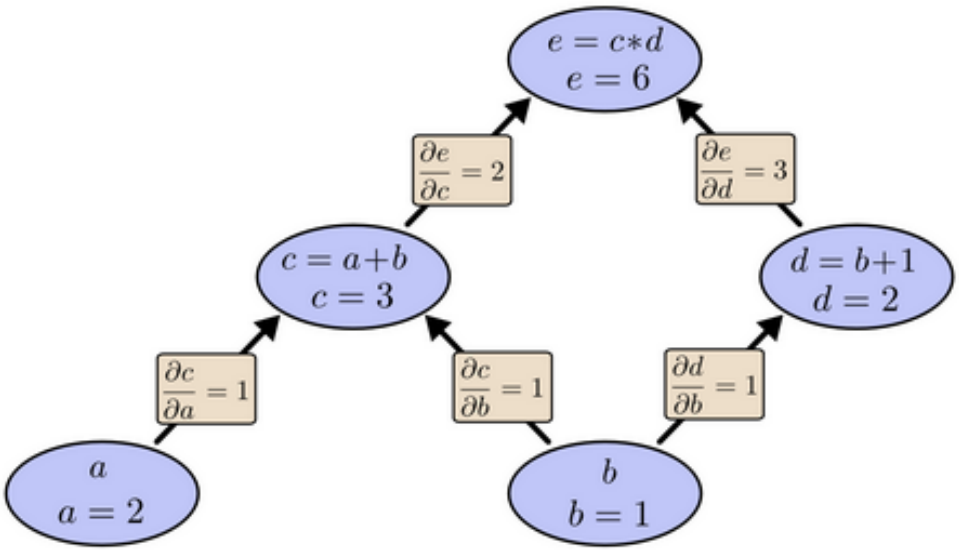


Figure 18: Computational graph for $e = (a + b) \times (b + 1)$

What if we want to understand how nodes that aren't directly connected affect each other?

For example, how does a affect e?

- If we change 'a' at a speed of 1, 'c' also changes at a speed of 1.
- In turn, c changing at a speed of 1 causes e to change at a speed of 2.
- So e changes at a rate of 1 x 2 with respect to a.

The general rule is: - Sum over all possible paths from one node to the other, multiplying the derivatives on each edge of the path together. - For example:

$$\frac{\partial e}{\partial b} = (1 \times 2) + (1 \times 3)$$

The problem with just **summing over the paths** is that it's very easy to get a combinatorial explosion in the number of possible paths.

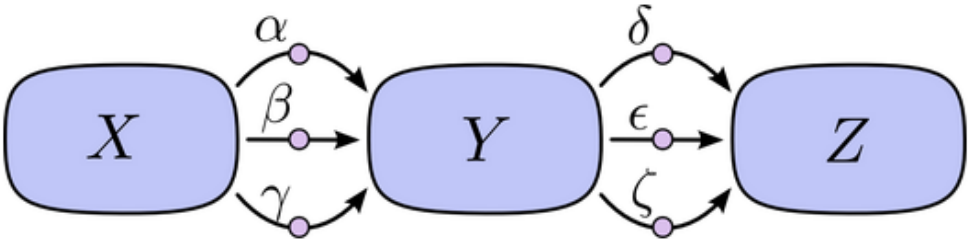


Figure 19: Multiple paths

In the above diagram, there are three paths from X to Y, and a further three paths from Y to Z. The above only has nine paths, but it would be easy to have the number of paths to grow exponentially as the graph becomes more complicated.

A dynamic Programming Approach:

Instead of just naively summing over the paths, it would be much better to factor them.This is where **forward-mode differentiation** and **reverse-mode differentiation** come in. They're algorithms for efficiently computing the sum by factoring the paths.

Instead of summing over all of the paths explicitly, they compute the same sum more efficiently by merging paths back together at every node. In fact, both algorithms touch each edge exactly once.

Forward-mode differentiation

- Starts at an input to the graph and moves towards the end.
- At every node, it sums all the paths feeding in.
- Each of those paths represents one way in which the input affects that node.
- By adding them up, we get the total way in which the node is affected by the input, it's derivative.
- Forward-mode differentiation tracks how one input affects every node.
- Applies the operator $\frac{\partial}{\partial X}$ to every node.

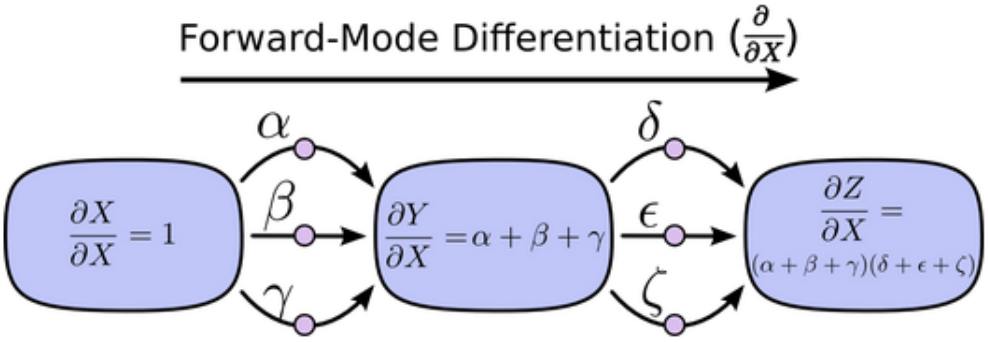


Figure 20: Forward-mode differentiation

Reverse-mode differentiation,

- Starts at an output of the graph and moves towards the beginning.
- At each node, it merges all paths which originated at that node.
- Reverse-mode differentiation tracks how every node affects one output.
- Applies the operator $\frac{\partial Z}{\partial}$ to every node.

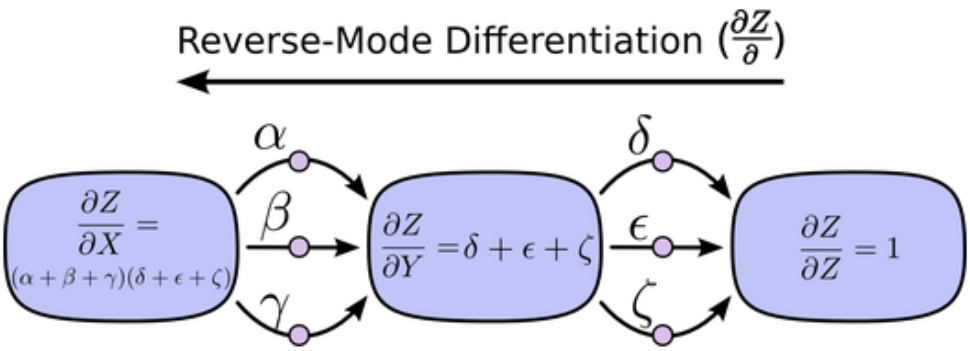


Figure 21: Reverse-mode differentiation

Considering the example we have, Is there an advantage to reverse mode differentiation?

With **forward-mode differentiation**, consider the node 'b'. We can get the derivative of every node with respect to 'b'. So we can compute how b affects every node in the graph. See below:

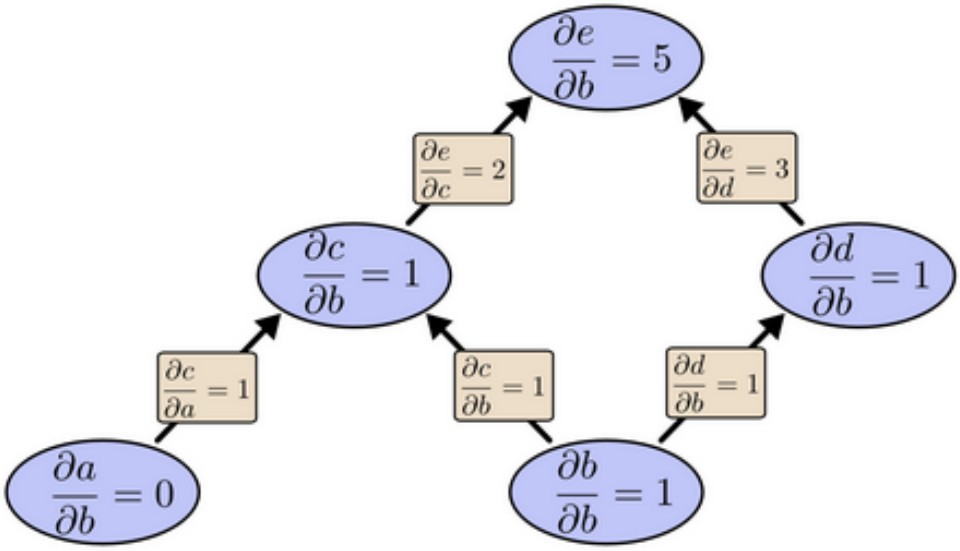


Figure 22: Forward-mode example, wrt b

With **reverse-node differentiation**, we get the derivative of e with respect to every other node in the graph

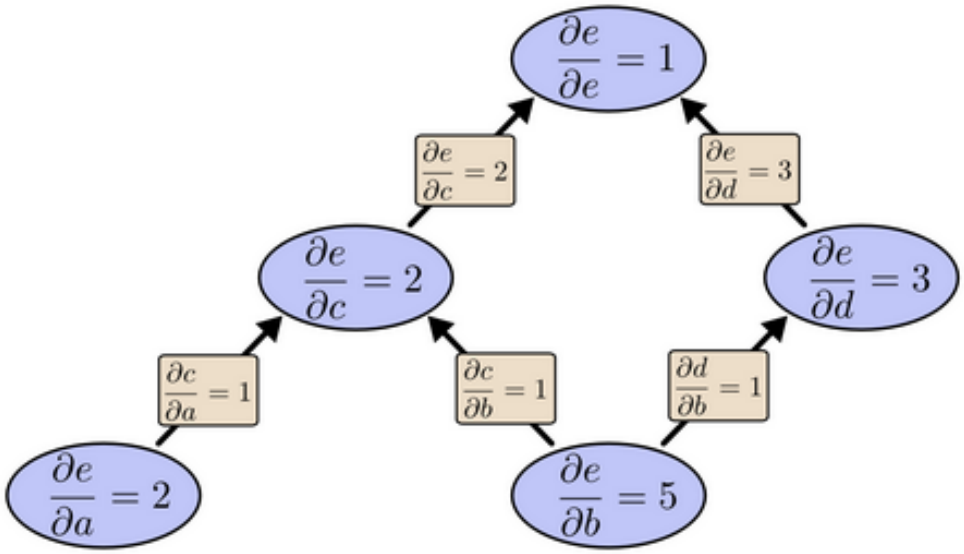


Figure 23: Reverse-mode example, wrt e

Advantage of reverse mode differentiation:

- Imagine a function with a million inputs and one output.
  - Forward-mode differentiation would require us to go through the graph a million times to get the derivatives.
  - Reverse-mode differentiation can get them all in one fell swoop! A speed up of a factor of a million is pretty nice!
- Where the reverse-mode gives the derivatives of one output with respect to all inputs, the forward-mode gives us the derivatives of all outputs with respect to one input.
  - If one has a function with lots of outputs, forward-mode differentiation can be much, much, much faster.

Backpropogation: Notation and Equations

Notation

- Weights: w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer.
- Bias: b^l_j is the bias of the j^{th} neuron in the l^{th} layer.
- Input: z^l_j is the input to the j^{th} neuron in the l^{th} layer.
- Activation: a^l_j is the activation of the j^{th} neuron in the l^{th} layer.
  - Activation a^l_j is related to the activations in the (l-1)th layer by the following:

$$z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_k$$

$$a^l_j = \sigma(z^l_j)$$

- In vector form, this gives us:

$$z^l = w^l a^{l-1} + b^l$$

$$a^l = \sigma(z^l)$$

- Output: a^L is the activation of the last layer
- Hadamard product, $s \odot t$ is element-wise multiplication. For example:

$$\begin{bmatrix}1\\2\end{bmatrix} \odot \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}1\times 3\\2\times 4\end{bmatrix} = \begin{bmatrix}3\\8\end{bmatrix}$$

- Cost finction, $C = C(a^L)$
- Error, δ^l_j is the error at neuron j in layer l:

$$\delta^l_j = \frac{\partial C}{\partial z^l_j}$$

$$\delta^l = \frac{\partial C}{\partial z^l} \text{ (In vector form)}$$

Fundamental equations of Back-propogation

- **Equation 1**: Error in the output layer

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma^{'}(z_j^L)$$

$$\delta^L = \nabla_{a^L} C \odot \sigma^{'}(z^L) \text{ (In vector form)}$$

where, ∇_a is the Jacobian matrix of the gradient of the cost cost over activation at layer L

$\sigma^{'}(z^L)$ measures how fast the activation function changes at z^L

and, $\sigma^{'}(z^L) = \frac{\partial a^L}{\partial z^L} = \frac{\partial \sigma(z^L)}{\partial z^L}$

- **Equation 2**: Error in layer "l", interms of error at layer "l+1"

$$\delta^l = [(w^{l+1})^T \delta^{l+1}] \odot [\sigma^{'}(z^l)]$$

where,

$(w^{l+1})^T$ is the transpose of the weight matrix w^{l+1} at the $(l+1)^{th}$ layer

- **Equation 3**: Rate of change of cost with respect to any bias in the network

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l$$

That is, rate of change of cost w.r.t bias for a specific neuron is exactly equal to the error at that neuron

- **Equation 4**: Rate of change of cost with respect to any weight in the network

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$$

$$\frac{\partial C}{\partial w} = a_{in}\delta_{out} \text{ (in simplified terms)}$$

Back Propogation: Algorithm

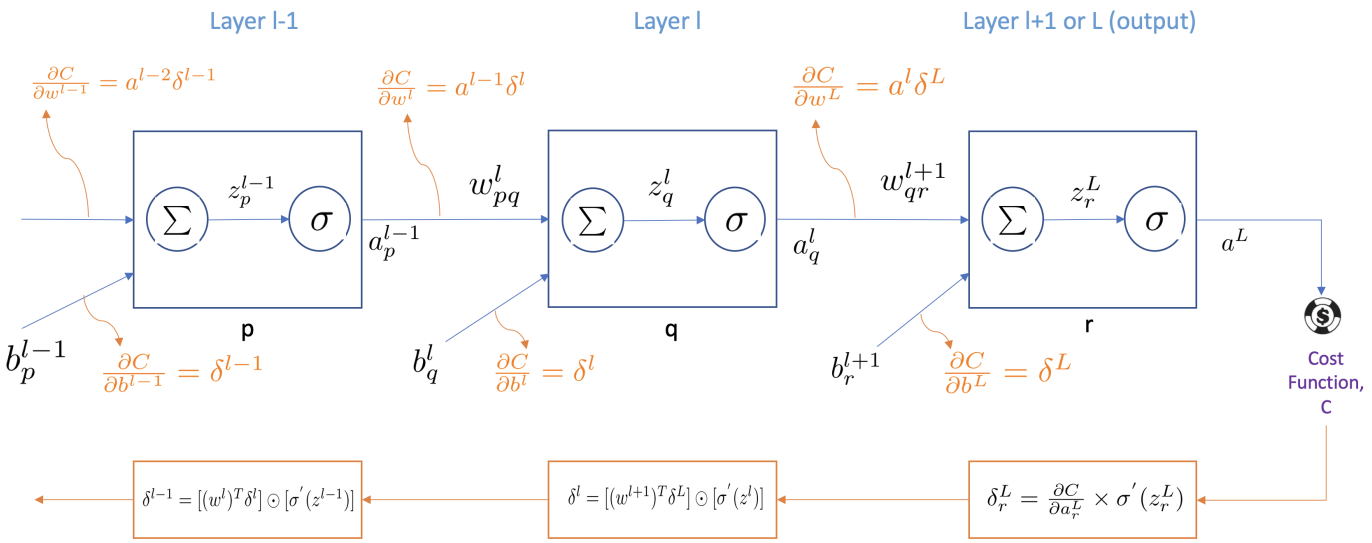


Figure 24: Notation and Equations

The algorithm is as follows:

1. **Input**: Set the input X for the input layer.
2. For each Training example, do the following:
    1. **Forward Propogation**: For each layer, l = 2,3..L, compute:

$$z^{l} = w^{l}.a^{l-1} + b^{l}$$

$$a^{l} = \sigma(z^{l})$$

    2. **Output Error**: Compute error for output layer

$$\delta^{L} = \nabla_{a^{L}} C \odot \sigma'(z^{L})$$

    3. **Backpropogate error**: For each layer, l = L-1,L-2...2, compute

$$\delta^l = [(w^{l+1})^T \delta^{l+1}] \odot [\sigma^{'}(z^l)]$$

$$\frac{\partial C}{\partial b^l} = \delta^l$$

$$\frac{\partial C}{\partial w^l} = a^{l-1}.\delta^l$$

3. **Gradient Descent**: For each layer l = 2,3... L, update the weights as follows:

$$w^l \rightarrow w^l - \alpha.\frac{\partial C}{\partial w^l}$$

$$b^l \rightarrow b^l - \alpha.\frac{\partial C}{\partial b^l}$$

Refer: http://neuralnetworksanddeeplearning.com/chap2.html