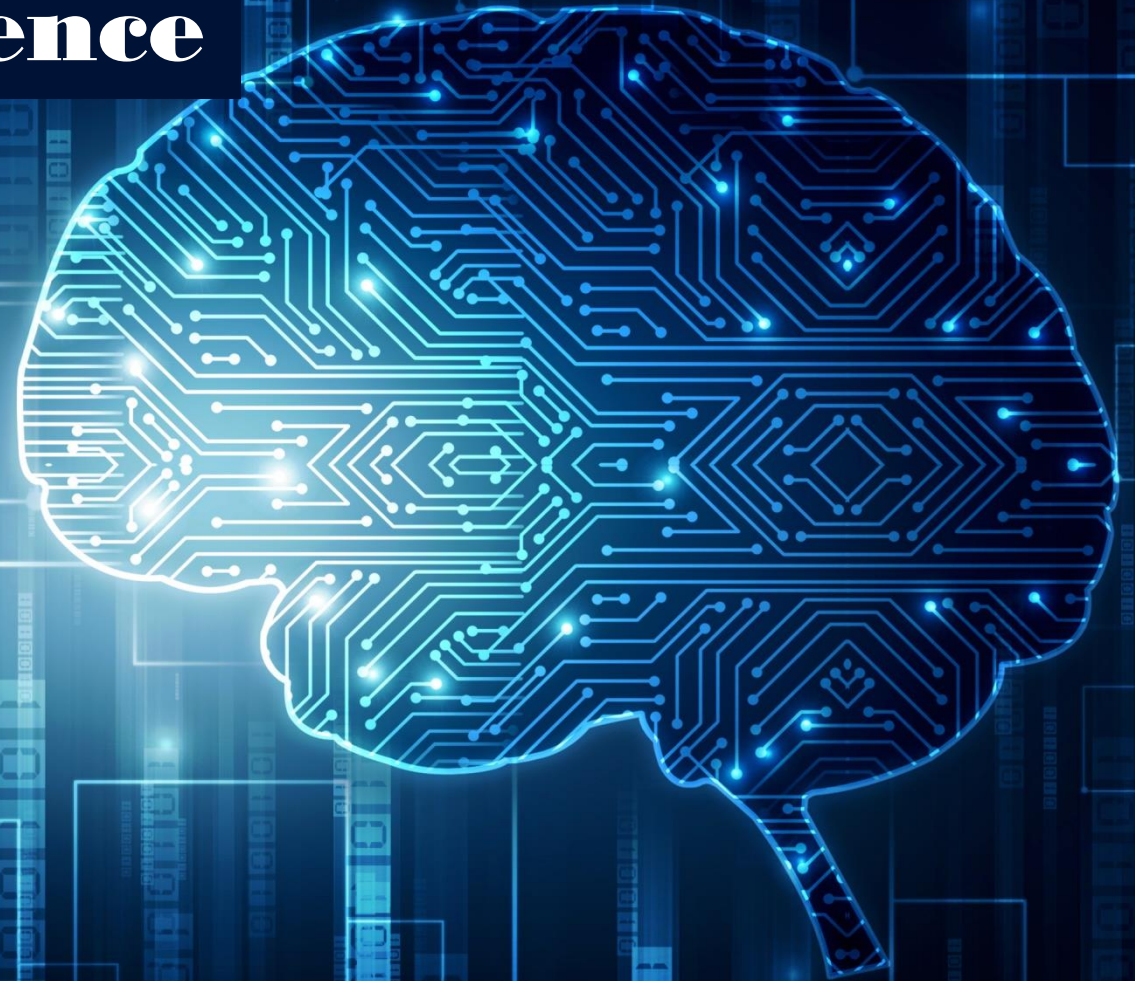


Computational Neuroscience

Chapter (1) – Part (2)

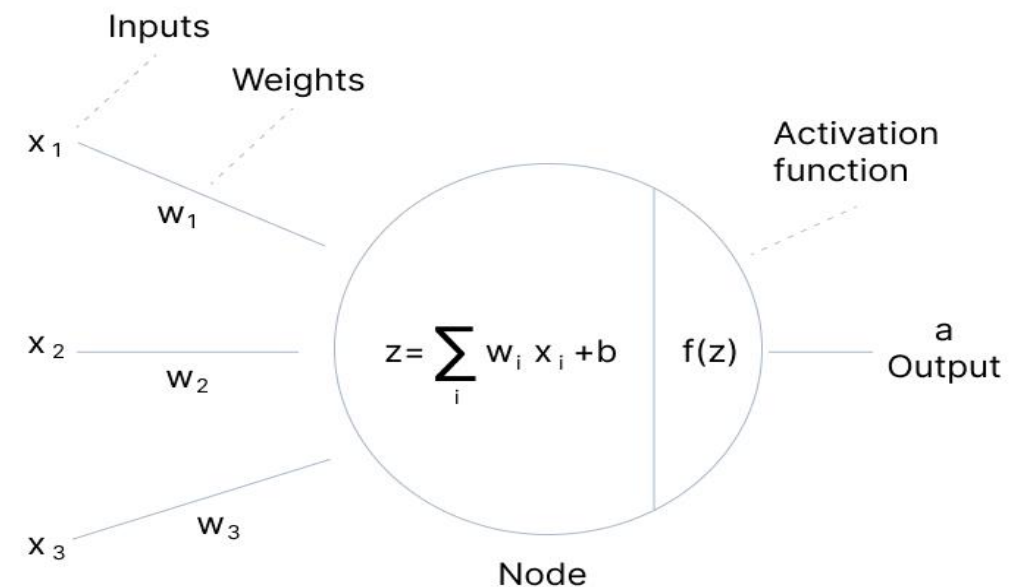
Edited by

A. Prof. Noha El-Attar



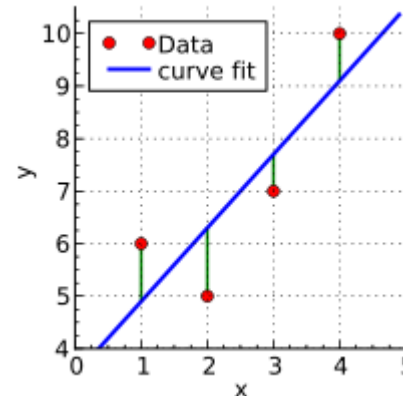
What is a Neural Network Activation Function?

- **An Activation Function** decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations.
- The role of the Activation Function is to derive output from a set of input values fed to a node (or a layer).

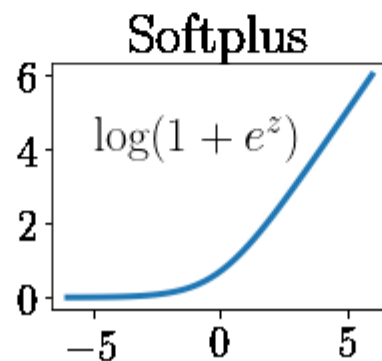
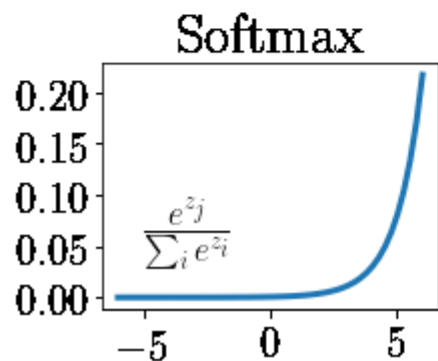
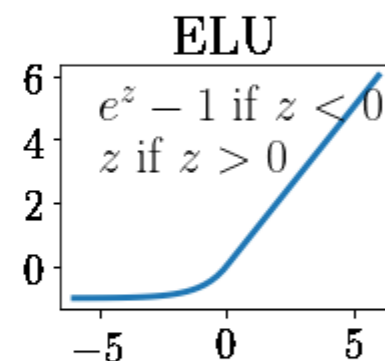
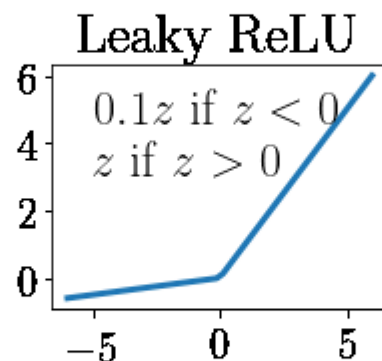
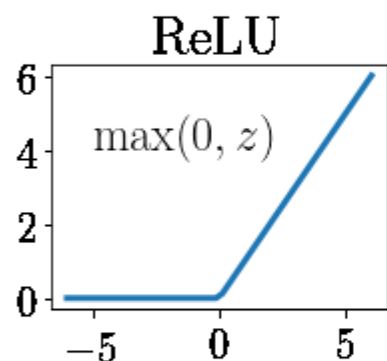
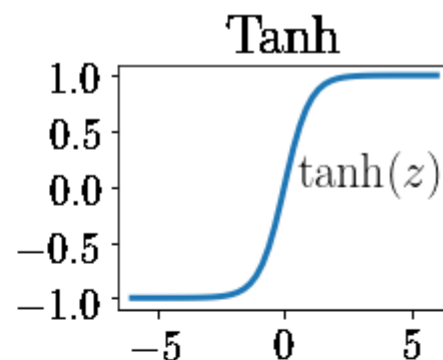
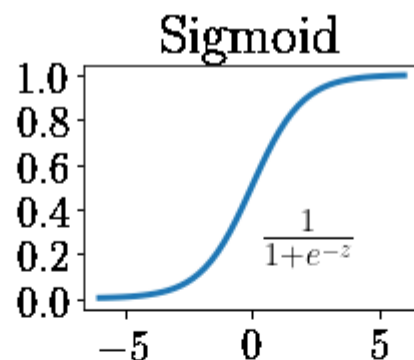
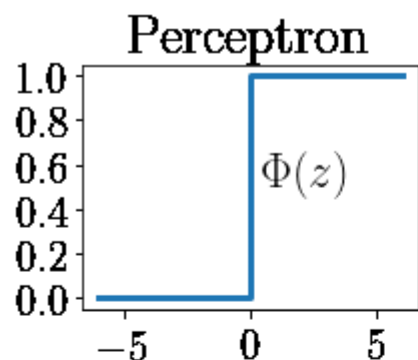


What happened if we don't apply ACTIVATION FUNCTION to our Network?

- Imagine a neural network without the activation functions. In that case, **every neuron will only be performing a linear transformation on the inputs using the weights and biases**. Although linear transformations make the neural network simpler, but this network would be less powerful and will not be able to learn the complex patterns from the data.
- A neural network without an activation function is essentially just a **linear regression model**.



Types of Activation Functions



1. Binary Step Function

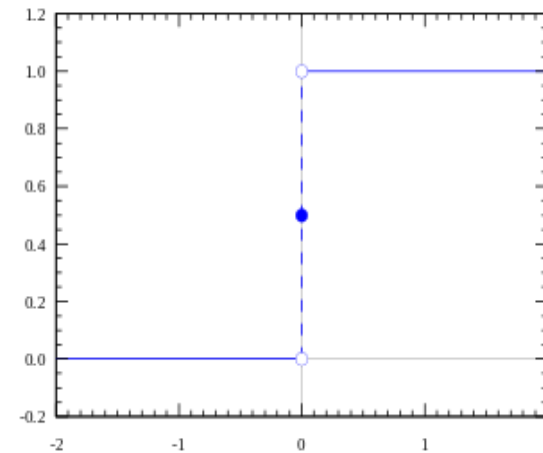
- If the input to the activation function is greater than a threshold, then the neuron is activated, else it is deactivated

- **Example:**

```
def binary_step(x):  
    if x<0:  
        return 0  
    else:  
        return 1  
binary_step(5), binary_step(-1)
```

Output:

(1,0)



2. Sigmoid function

- It is one of the most widely used non-linear activation function. Sigmoid transforms the values between the range 0 and 1.

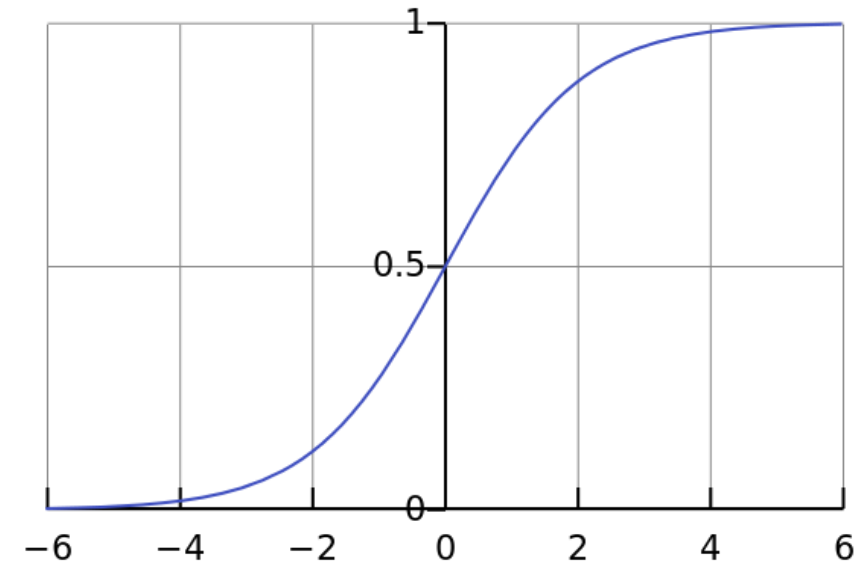
$$A = \frac{1}{1+e^{-x}}$$

- Example:

```
import numpy as np
def sigmoid_function(x):
    z = (1/(1 + np.exp(-x)))
    return z
sigmoid_function(7),sigmoid_function(-22)
```

Output:

(0.9990889488055994, 2.7894680920908113e-10)



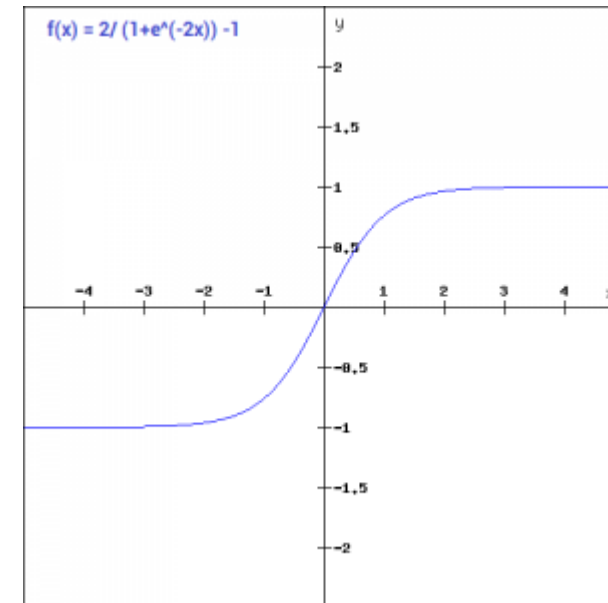
3. Tanh

- This looks very similar to sigmoid. In fact, it is a scaled sigmoid function!
- The range of values in this case is from -1 to 1.

$$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$$

$$\tanh(x) = 2 \operatorname{sigmoid}(2x) - 1$$

The sigmoid function has a slower rate of change than the tanh function



4. ReLU

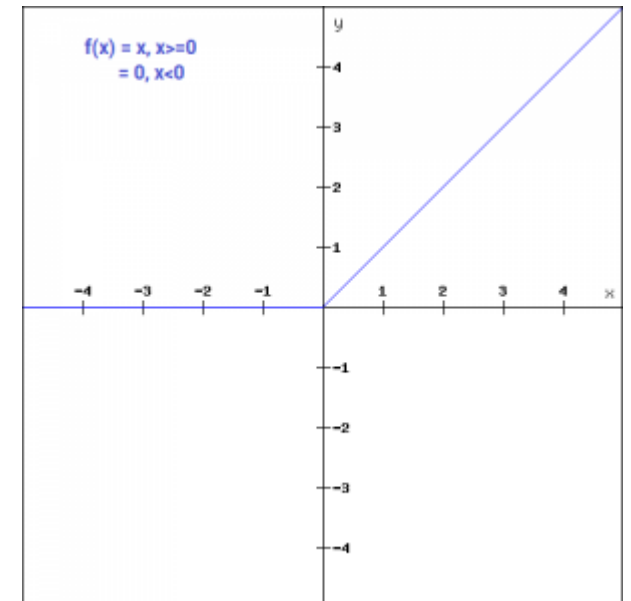
- The ReLU function is another non-linear activation function that has gained popularity in the deep learning domain. ReLU stands for Rectified Linear Unit. **The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.**
- This means that the neurons will only be deactivated if the output of the linear transformation is less than 0.

$$f(x) = \max(0, x)$$

- **Example:**

```
def relu_function(x):  
    if x < 0:  
        return 0  
    else:  
        return x  
relu_function(7), relu_function(-7)
```

Output:
(7, 0)



5. Leaky ReLu

- Leaky ReLU function is an improved version of the ReLU function. As we saw that for the ReLU function, the gradient is 0 for $x < 0$, which would deactivate the neurons in that region.
- Leaky ReLU is defined to address this problem. Instead of defining the Relu function as 0 for negative values of x , we define it as an extremely small linear component of x .

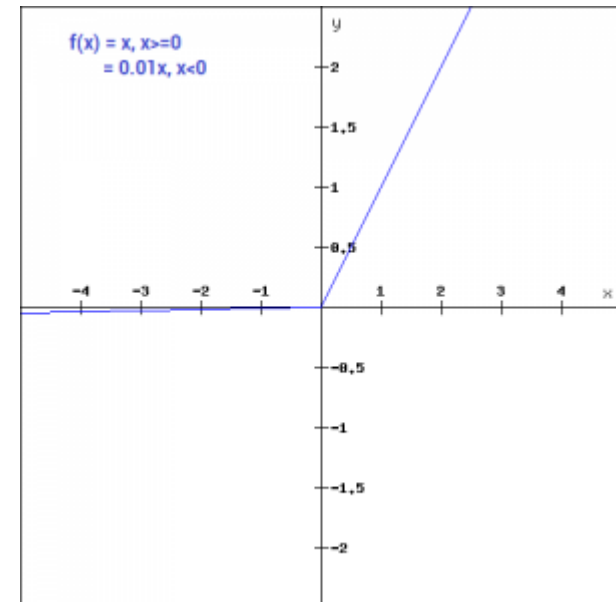
$$\begin{aligned} f(x) &= 0.01x, x < 0 \\ &= x, x \geq 0 \end{aligned}$$

- Example:**

```
def relu_function(x):  
    if x < 0:  
        return 0.01 * x  
    else:  
        return x  
relu_function(7), relu_function(-7)
```

Output:

(7, -0.07)



6. Softmax

- **Softmax function** is often described as a combination of multiple sigmoids. We know that sigmoid returns values between 0 and 1, which can be treated as probabilities of a data point belonging to a particular class. Thus sigmoid is widely used for binary classification problems.
- The softmax function can be used for **multiclass classification problems**. This function returns the probability for a data point belonging to each individual class.

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- **Example:** if you have 3 classes, there would be three neurons in the output layer. Suppose you got the output from the neurons as [1.2 , 0.9 , 0.75].
- Applying the softmax function over these values, you will get the following result [0.42 , 0.31, 0.27].
- These represent the probability for the data point belonging to each class. Note that the sum of all the values is 1.

Choosing the right Activation Function

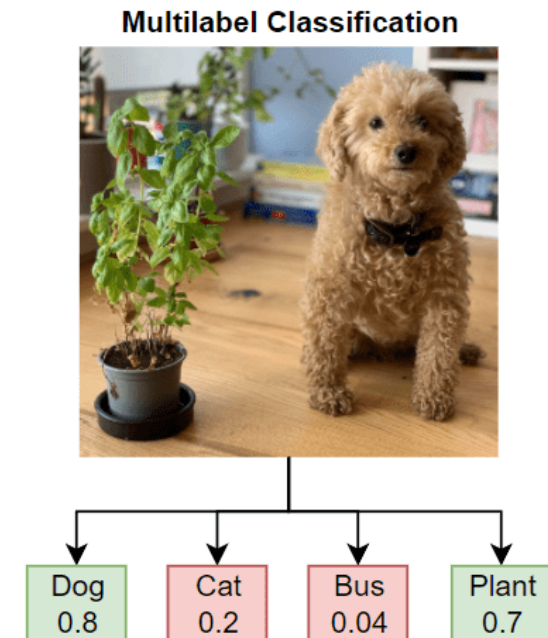
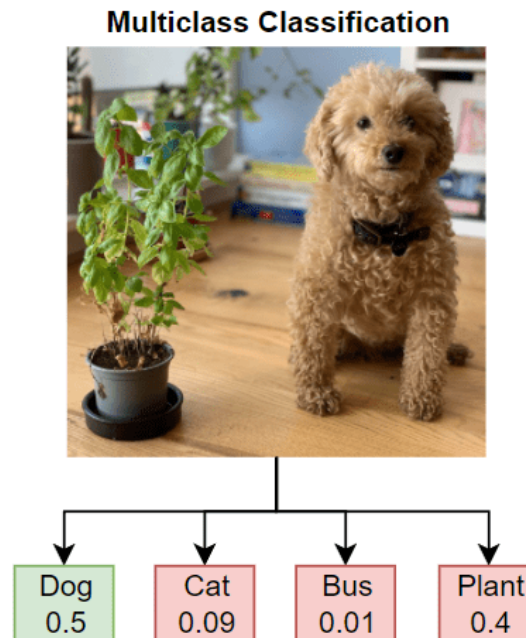
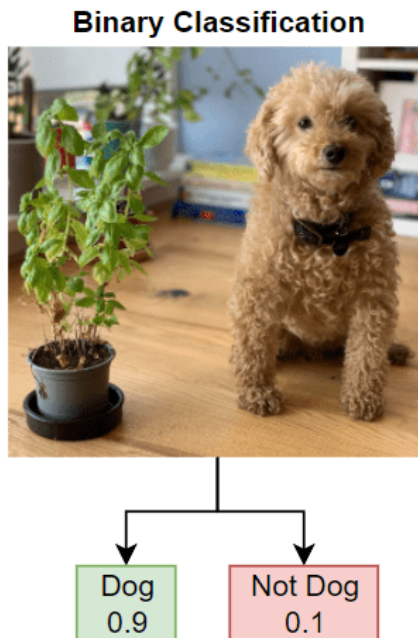


Self study

- When you know the function you are trying to approximate, you can choose an activation function which will approximate the function faster leading to faster training process.
- However depending upon the properties of the problem we might be able to make a better choice for easy and quicker convergence of the network.
 1. Sigmoid functions and their combinations generally work better in the case of classifiers
 2. Sigmoids and tanh functions are sometimes avoided due to the vanishing gradient problem
 3. ReLU function is a general activation function and is used in most cases these days
 4. If we encounter a case of dead neurons in our networks the leaky ReLU function is the best choice
 5. Always keep in mind that ReLU function should only be used in the hidden layers
 6. As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case ReLU doesn't provide with optimum results

Finally, a few rules for choosing the activation function for your output layer based on the type of prediction problem that you are solving:

- **Regression** - Linear Activation Function
- **Binary Classification**—Sigmoid/Logistic Activation Function
- **Multiclass Classification**—Softmax
- **Multilabel Classification**—Sigmoid



- The activation function used in **hidden layers** is typically chosen based on the type of neural network architecture.
 - Convolutional Neural Network (CNN): ReLU activation function.
 - Recurrent Neural Network: Tanh and/or Sigmoid activation function.

Network Design Issues

- Initial weights (small random values $\in [-1,1]$)
- Activation function
- Error estimation
- Weights adjusting
- Number of neurons
- Data representation
- Size of training set

Error Estimation

- There are several types of measures that use to find the differences between the predicted value by a model and the actual value, such as:
- **Root mean square error (RMSE):** a frequently-used measure. $RMSE = \sqrt{\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2}$
- **Mean squared error** $MSE = \frac{1}{n} \sum (y - \hat{y})^2$
- **Mean absolute error:** $MAE = \frac{1}{n} \sum_{j=1}^n |y_j - \hat{y}_j|$
- **Relative Squared Error (RSE)**
- **Relative Absolute Error (RAE)**

Adjusting Weights

The process of learning in an ANN involves adjusting the weights to reduce the error between the predicted output and the actual output.

This is typically done using a learning algorithm like [backpropagation](#) combined with an optimization technique such as [gradient descent](#).

During [backpropagation](#), the error is calculated at the output and propagated back through the network to update the weights in a way that minimally **reduces the overall error**.

Learning Rate [0,1]: [hyperparameter](#) that determines the size of the steps taken during the weight update process. ***The higher the value of α or η , the faster the network learns. However, a higher learning rate also results in lower generalization capability.*** The learning rate, therefore, plays a critical role in determining how significantly the weights are adjusted during each iteration of the training process.

What is a hyperparameter?

A hyperparameter is a parameter that is set before the learning process begins. These parameters are tunable and can directly affect how well a model trains.

Adjusting Weights - Cont..

Regularization of Weights

To prevent **overfitting**, where the ANN performs well on the training data but poorly on unseen data, regularization techniques can be applied to the weights.

Regularization methods like L1 and L2 add a penalty term to the loss function based on the magnitude of the weights.

This encourages the network to maintain smaller weights, leading to simpler models that generalize better to new data.

What is Overfitting?

When a model performs very well for training [data](#) but has poor performance with test data (new data), it is known as overfitting. In this case, the machine learning model learns the details and noise in the training data such that it negatively affects the performance of the model on test data. Overfitting can happen due to low bias and high variance.

Reasons for Overfitting

- Data used for training is not cleaned and contains noise (garbage values) in it
- The model has a high variance
- The size of the training dataset used is not enough
- The model is too complex

Ways to Tackle Overfitting

- Using K-fold cross-validation
- Using Regularization techniques such as Lasso and Ridge
- Training model with sufficient data

What is Underfitting?

When a model has not learned the patterns in the training data well and is unable to generalize well on the new data, it is known as underfitting. An underfit model has poor performance on the training data and will result in unreliable predictions. Underfitting occurs due to high bias and low variance.

Reasons for Underfitting

- Data used for training is not cleaned and contains noise (garbage values) in it
- The model has a high bias
- The size of the training dataset used is not enough
- The model is too simple

Ways to Tackle Underfitting

- Increase the number of features in the dataset
- Increase model complexity
- Reduce noise in the data
- Increase the duration of training the data

Number of Neurons and Hidden Layers

How can you determine number of Hidden Layers?

| Number of Hidden Layers | Result |
|-------------------------|---|
| none | Only capable of representing linear separable functions or decisions. |
| 1 | Can approximate any function that contains a continuous mapping from one finite space to another. |
| 2 | Can represent an arbitrary decision boundary to arbitrary accuracy with rational activation functions and can approximate any smooth mapping to any accuracy. |

Number of Neurons and Hidden Layers

How can you determine number of neurons in Hidden Layers?

There are many rule-of-thumb methods for determining the correct number of neurons to use in the hidden layers, such as:

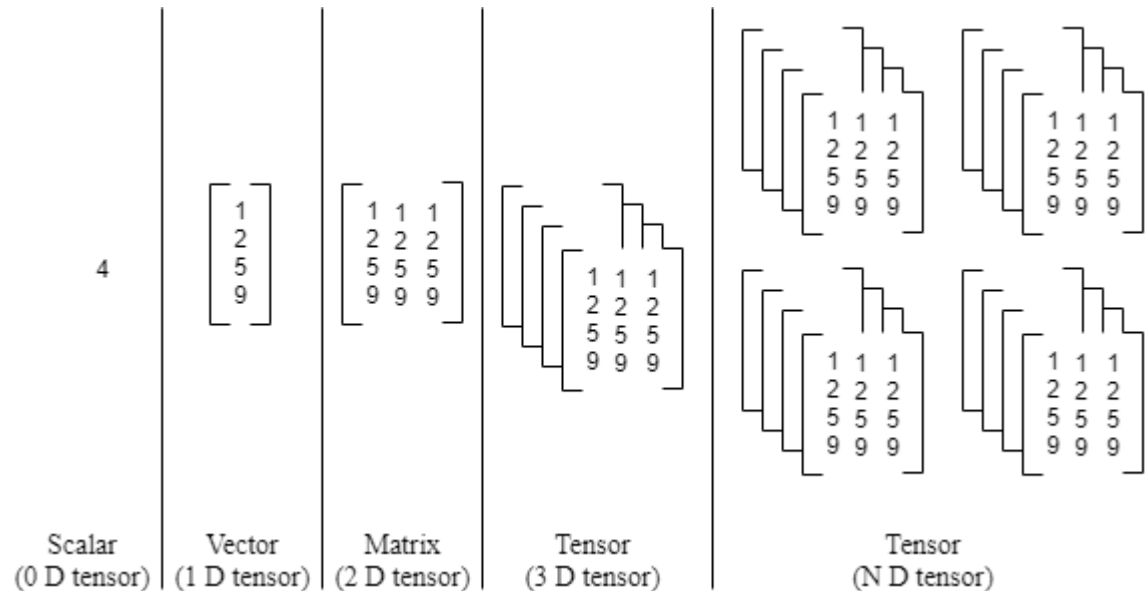
1. The number of hidden neurons should be **between the size of the input layer and the size of the output layer**.
2. The number of hidden neurons should be **$2/3$ the size of the input layer, plus the size of the output layer**.
3. The number of hidden neurons should be **less than twice the size of the input layer**.

Moreover, the number of neurons and number layers required for the hidden layer also depends upon training cases, amount of outliers, the complexity of, data that is to be learned, and the type of activation functions used.

Data Representation

All machine learning systems use Tensors (multidimensional arrays) as their basic data structure.

1. ***Scalars (0D tensors)***
2. ***Vectors (1D tensors)***
3. ***Matrices (2D tensors)***
4. ***3D tensor and higher-dimensional tensor***



Data Representation

A real-world example of a data tensor:

1. **Vector data:** It is a 2D tensor of shape(samples, features)
2. **Time series data or sequence data:** It is a 3D tensor of shape(samples, timesteps, features)
3. **Images:** It is a 4D tensor of shape(samples, height, width, channel).
4. **Video:** It is a 5D tensor of shape(samples, frames, height, width, channel).

BACK PROPAGATION

(BACKWARD PROPAGATION OF ERRORS)

- Backpropagation is a common method for training a neural network.
- The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.
- To tune the weights between the hidden layer and the input layer, we need to know the error at the hidden layer, but we know the error only at the output layer (*We know the correct output from the training sample and we also know the output predicted by the network.*).

BACK PROPAGATION

- For a particular neuron in output layer

$$\text{for all } j \{ \mathbf{W}_{j,i} = \mathbf{W}_{j,i} + a * g'(\text{sum of all inputs}) * (T-A) * P(j) \}$$

- This equation tunes the weights between the output layer and the hidden layer.
- For a particular neuron j in hidden layer, we propagate the error backwards from the output layer, thus

$$\text{Error} = \mathbf{W}_{j,1} * E_1 + \mathbf{W}_{j,2} * E_2 + \dots \text{ for all the neurons in output layer.}$$

- Thus, for a particular neuron in hidden layer

$$\text{for all } k \{ \mathbf{W}_{k,j} = \mathbf{W}_{k,j} + a * g'(\text{sum of all inputs}) * (T-A) * P(k) \}$$

- This equation tunes the weights between the hidden layer and the input layer.

Learning by Gradient Descent Error Minimization

- The Perceptron learning rule is an algorithm that adjusts the network weights w_{ij} to minimize the difference between the actual outputs out_j and the target outputs $target_j^p$. We can quantify this difference by defining the **Sum Squared Error** function, summed over all output units j and all training patterns p :

$$E(w_{mn}) = \frac{1}{2} \sum_p \sum_j \left(target_j^p - out_j(in_i^p) \right)^2$$

Learning by Gradient Descent Error Minimization

- It is the general aim of network **learning** to minimize this error by adjusting the weights w_{mn} . Typically we make a series of small adjustments to the weights $w_{mn} \rightarrow w_{mn} + \Delta w_{mn}$ until the error $E(w_{mn})$ is 'small enough'. We can determine which direction to change the weights in by looking at the gradients (i.e. partial derivatives) of E with respect to each weight w_{mn} . Then the **gradient descent update equation** (with positive learning rate η) is

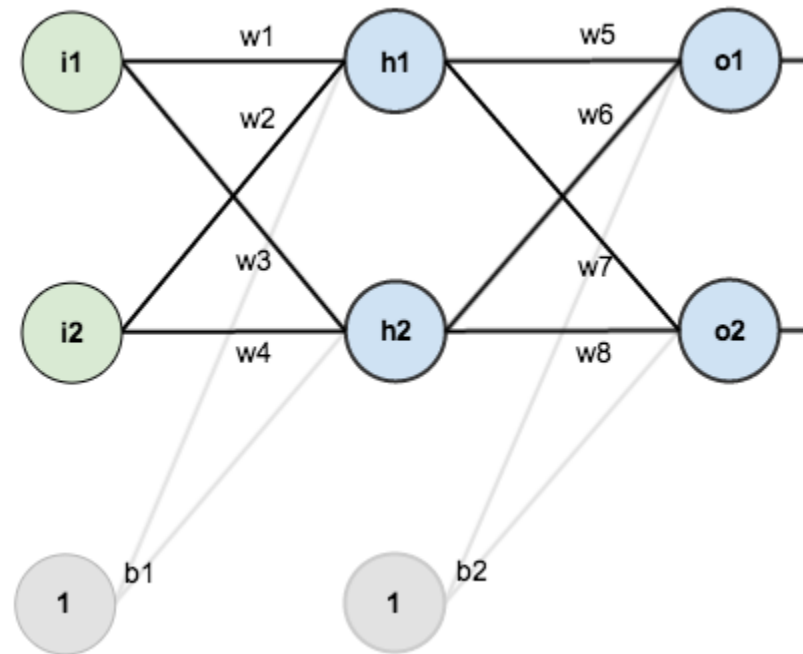
$$\Delta w_{kl} = -\eta \frac{\partial E(w_{mn})}{\partial w_{kl}}$$

which can be applied iteratively to minimize the error.

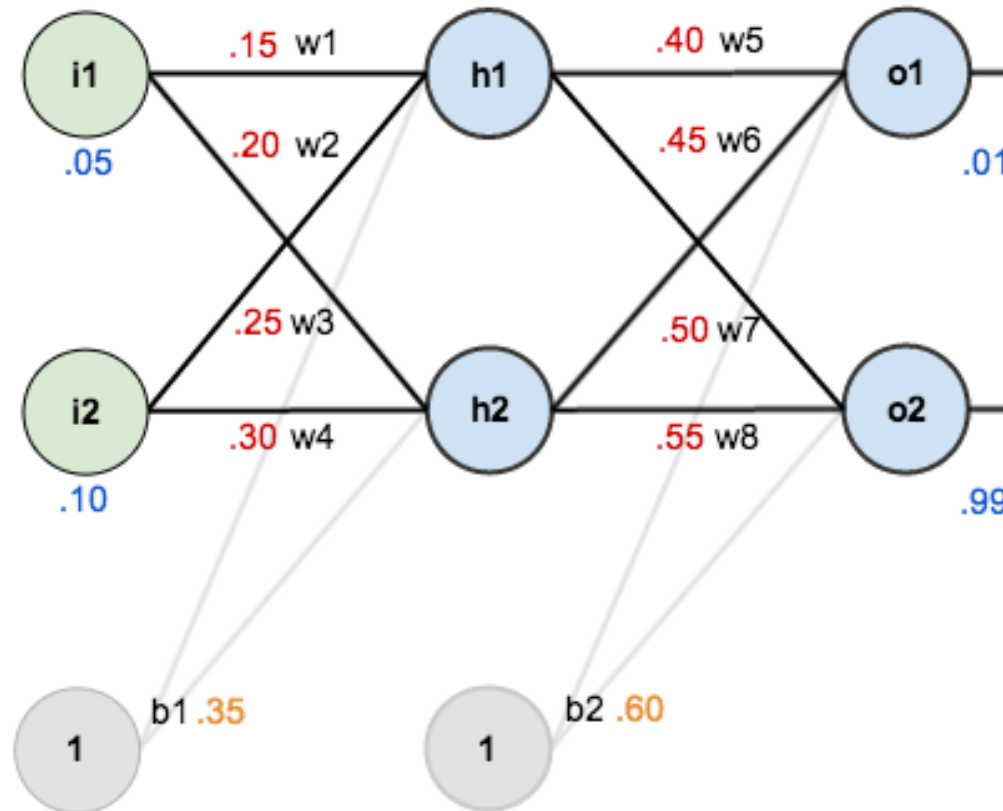
BACK PROPOGATION BY EXAMPLE

<https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/>

- Consider a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.



In order to have some numbers to work with, here are the initial weights, the biases, and training inputs/outputs: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.



The Forward Pass

- To begin, let's see what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10. To do this we'll feed those inputs forward through the network.
- We figure out the *total net input* to each hidden layer neuron, *squash* the total net input using an *activation function* (here we use the *sigmoid function*), then repeat the process with the output layer neurons.
- Here's how we calculate the total net input for h_1 :

$$\begin{aligned}net_{h_1} &= w_1 * i_1 + w_2 * i_2 + b_1 * 1 \\net_{h_1} &= 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775\end{aligned}$$

We then squash it using the logistic function to get the output of h_1

$$out_{h_1} = \frac{1}{1 + e^{-net_{h_1}}} = \frac{1}{1 + e^{-0.3775}} = 0.5933$$

- Carrying out the same process for h_2 we get $out_{h_2} = 0.5969$.
- We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.
- Here's the output for o_1 :

$$\begin{aligned}
 net_{o_1} &= w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1 \\
 net_{o_1} &= 0.4 * 0.5933 + 0.45 * 0.5969 + 0.6 * 1 = 1.1059 \\
 out_{o_1} &= \frac{1}{1 + e^{-net_{o_1}}} = \frac{1}{1 + e^{1.1059}} = 0.7514.
 \end{aligned}$$

and carrying out the same process for o_2 we get $out_{o_2} = 0.7729$.

Calculating the Total Error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

$$E_{total} = \frac{1}{2} \sum_{i=1}^n (target - output)^2$$

For example, the target output for o_1 is 0.01 but the neural network output 0.7514, therefore its error is:

$$E_{total_{o_1}} = \frac{1}{2} \sum_{i=1}^n (target_{o_1} - output_{o_1})^2 = \frac{1}{2} (0.01 - 0.7514)^2 = 0.2748$$

Repeating this process for o_2 (remembering that the target is 0.99) we get: $E_{total_{o_2}} = 0.0236$.

The total error for the neural network is the sum of these errors:

$$E_{total} = E_{total_{o_1}} + E_{total_{o_2}} = 0.2748 + 0.0236 = 0.2984.$$

The Backwards Pass

- Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer to the target output, thereby minimizing the error for each output neuron and the network as a whole.

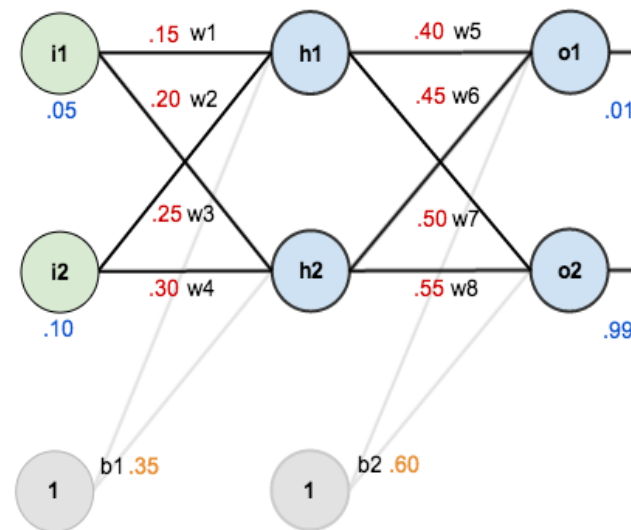
Output Layer

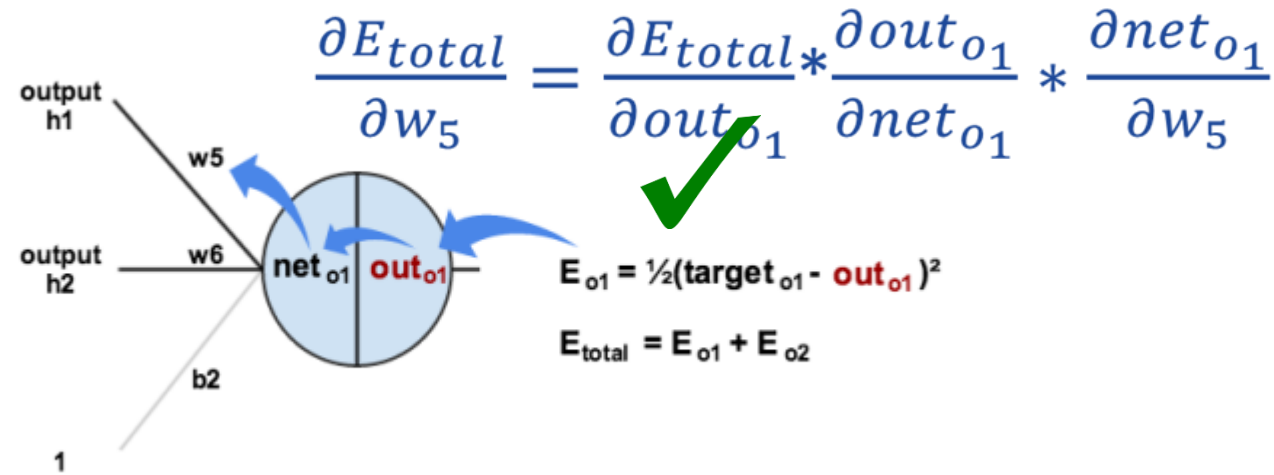
Consider w_5 . We want to know how much a change in w_5 affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$ (the partial derivative of E_{total} with respect to w_5).

You can also say “the gradient with respect to w_5 ”.)

- By applying the chain rule we know that:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$






- We need to figure out each piece in this equation.
- First, how much does the total error change with respect to the output?

$$E_{total} = \frac{1}{2}(\text{target}_{o1} - \text{output}_{o1})^2 + \frac{1}{2}(\text{target}_{o2} - \text{output}_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(\text{target}_{o1} - \text{output}_{o1}) = -(0.01 - 0.7514) = 0.7414$$

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial w_5}$$


- Next, how much does the output of o_1 change with respect to its total net input?
- The partial derivative of the logistic function is the output multiplied by 1 minus the output:

$$\frac{\partial out_{o_1}}{\partial net_{o_1}} = out_{o_1}(1 - out_{o_1}) = 0.7514(1 - 0.7514) = 0.1868$$

Finally, how much does the total net input of o_1 change with respect to w_5 ?

$$\frac{\partial net_{o_1}}{\partial w_5} = out_{h_1} = 0.5933$$

- Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.7414 * 0.1868 * 0.5933 = 0.0822$$

- To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5):

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.0822 = 0.3589$$

- We can repeat this process to get the new weights w_6 , w_7 , and w_8

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

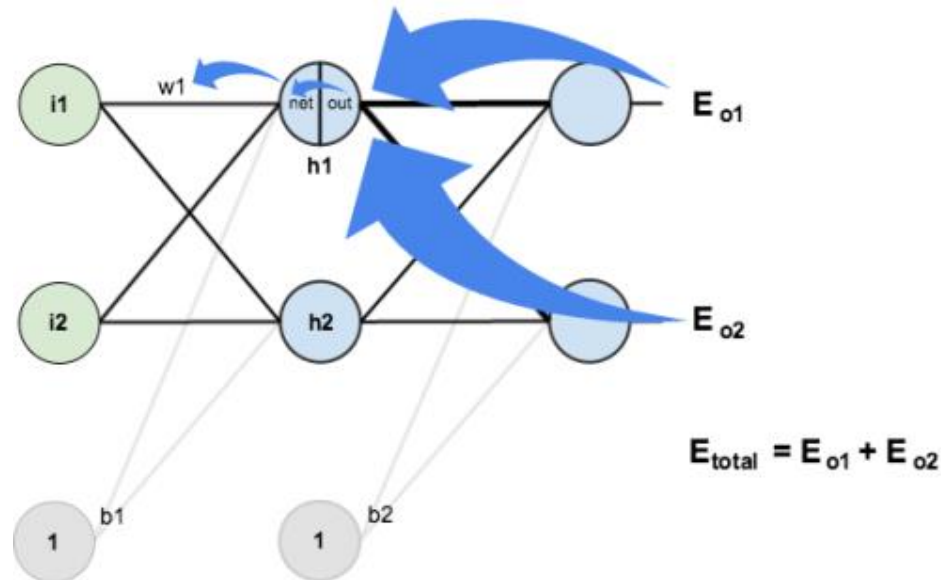
- We perform the actual updates in the neural network *after* we have the new weights leading into the hidden layer neurons.

Hidden Layer

- Next, we'll continue the backwards pass by calculating new values for w_1 , w_2 , w_3 , and w_4 .
- Big picture, here's what we need to figure out:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$
$$\downarrow$$
$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$



- We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out_{h_1} affects both out_{o_1} and out_{o_2} therefore the $\frac{\partial E_{total}}{\partial out_{h_1}}$ needs to take into consideration its effect on the both output neurons:

$$\frac{\partial E_{total}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial out_{h_1}} + \frac{\partial E_{o_2}}{\partial out_{h_1}}$$

$$\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial out_{h_1}}$$

$$\frac{\partial E_{o_1}}{\partial net_{o_1}} = \frac{\partial E_{o_1}}{\partial out_{o_1}} * \frac{\partial out_{o_1}}{\partial net_{h_1}} = 0.7414 * 0.1868 = 0.1385$$

$$net_{o_1} = w_5 * out_{h_1} + w_6 * out_{h_2} + b_2 * 1$$

$$\frac{\partial net_{o_1}}{\partial out_{h_1}} = w_5 = 0.40$$

Plugging them in: $\frac{\partial E_{o_1}}{\partial out_{h_1}} = \frac{\partial E_{o_1}}{\partial net_{o_1}} * \frac{\partial net_{o_1}}{\partial out_{h_1}} = 0.1385 * 0.40 = 0.0554$

- Following the same process for $\frac{\partial E_{o2}}{\partial out_{h1}}$: $\frac{\partial E_{o2}}{\partial out_{h1}} = -0.0190$.
- Hence, $\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.0554 - 0.0190 = 0.0364$.
- Now that we have $\frac{\partial E_{total}}{\partial out_{h1}}$, we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and then $\frac{\partial net_{h1}}{\partial w_1}$.

$$out_{h1} = \frac{1}{1 + e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.5933(1 - 0.5933) = 0.2413$$

- We calculate the partial derivative of the total net input to h1 with respect to w_1 the same as we did for the output neuron:

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1$$

- Putting it all together:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h_1}} * \frac{\partial out_{h_1}}{\partial net_{h_1}} * \frac{\partial net_{h_1}}{\partial w_1} = 0.0364 * 0.2413 * 0.05 = 0.00044$$

We can now update w_1 :

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.00044 = 0.1498$$

$$w_2^+ = 0.1996$$

$$w_3^+ = 0.2498$$

$$w_4^+ = 0.2995$$

Finally, we've updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.2984. After this first round of backpropagation, the total error is now down to 0.2910. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).

Don't forget to solve the rest of the example