CS396: Selected CS2
(Deep Learning for visual recognition)

Spring 2022

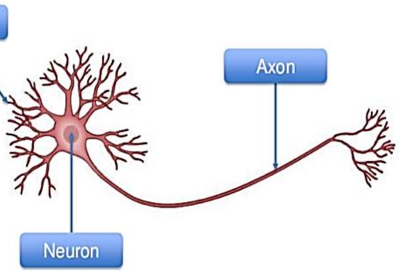
Dr. Wessam EL-Behaidy
Associate Professor, Computer Science Department,
Faculty of Computers and Artificial Intelligence,
Helwan University.

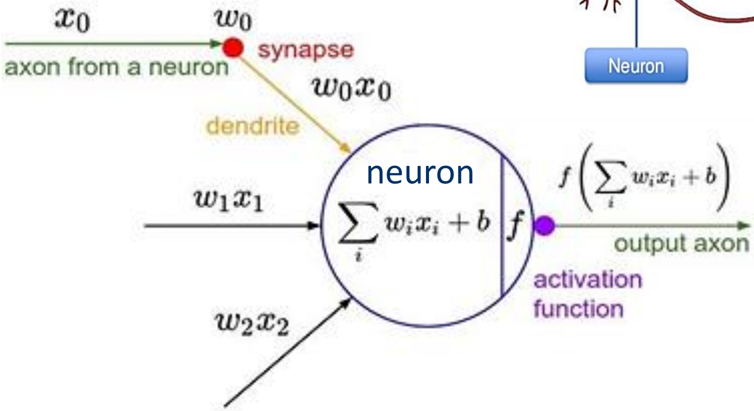
Lecture 2: Review

Neural Networks: Neurons and the Brain

- The origins: Algorithms that try to mimic the brain.
- Was very widely used in the 80s and early 90s; popularity diminished in the late 90s.
- Recent resurgence: State-of-the-art technique for many applications

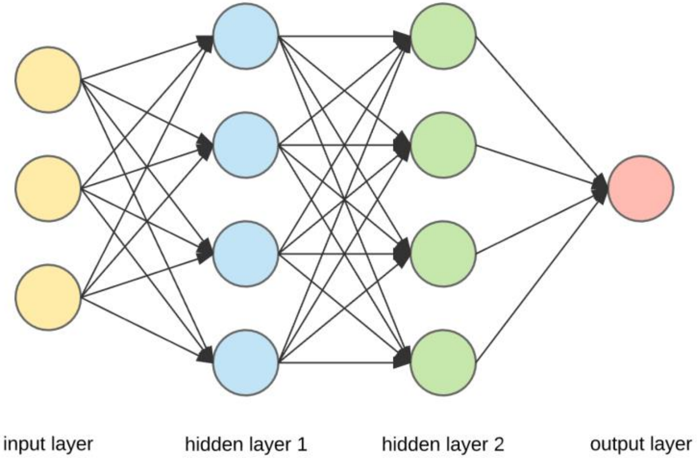
Biological Neuron Vs. Artificial Neuron





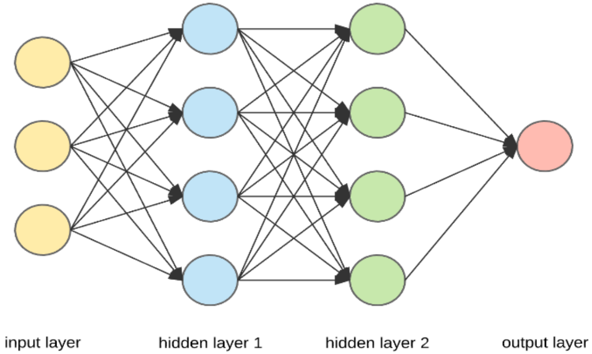
Artificial Neural networks (ANN)

Artificial Neural Networks (ANN) are multi-layer **fully-connected** neural nets. Every node in one layer is connected to every other node in the next layer.



Artificial Neural networks (ANN) (Cont.)

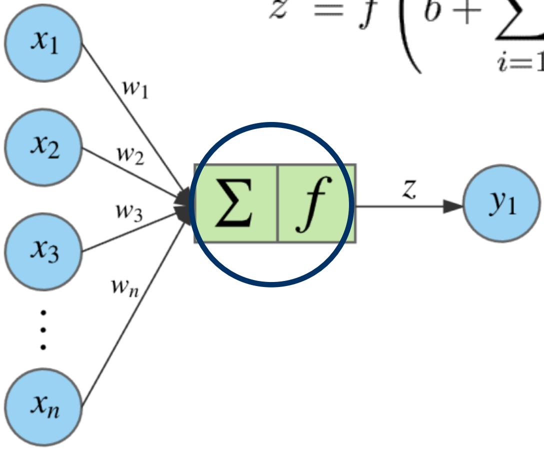
They consist of an **input layer**, one or multiple **hidden layers**, and an **output layer**. To make the network deeper, the number of hidden layers is increased



Artificial Neuron

A given node takes the weighted sum of its inputs plus bias, and passes it through a non-linear **activation function**.

$$z = f\left(b + \sum_{i=1}^{n} x_i w_i\right)$$



n is the number of inputs for the node (cell).

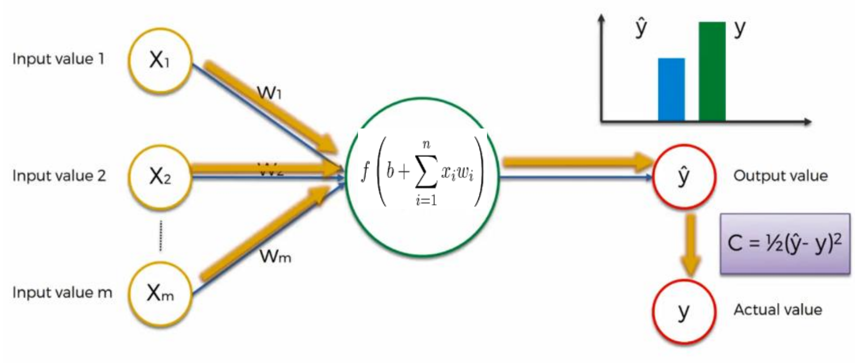
ω_i is the weight of the input sample

x_i is the input sample (feature)

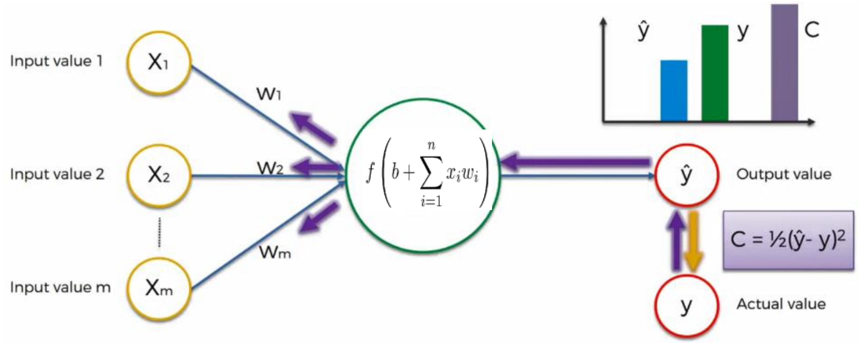
f is the activation fn.

z is the hypothesis fn.

Forward-Propagation



Backward-Propagation



Why activation function is used?

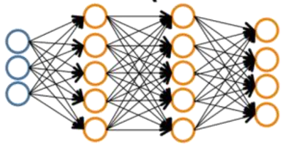
$$z = f\left(b + \sum_{i=1}^{n} x_i w_i\right)$$

- If we do not apply a Activation function then the output signal would simply be a simple ***linear function****.*
- Without activation function our Neural network would not be able to learn and model other complicated kinds of data such as images, videos, audio, speech etc.
- *Hence using a non linear Activation we are able to generate non-linear mappings from inputs to outputs.*

Important feature of a Activation function is that it should be **differentiable**

Multi-Class NN

Neural Network (Classification)



$\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})\}$

$L =$ total no. of layers in network

$s_l =$ no. of units (not counting bias unit) in layer l

Binary classification

$y = 0$ or 1

1 output unit

$\mathrm{K} = 2$

$s_L = 1$ (*output layer*)

Multi-class classification (K classes)

$y \in \mathbb{R}^K$ E.g. $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$, $\begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

pedestrian car motorcycle truck

K output units

$s_1 = 3, s_2 = 5, s_3 = 5, s_4 = 4$

$s_L = K$

$K \geq 3$

Andrew Ng

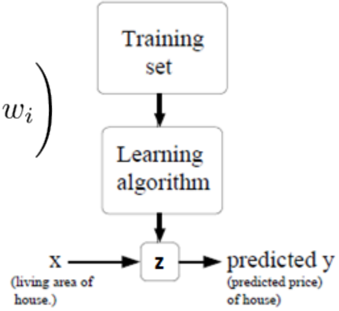
To Learn the Neuron

To learn a model of NN, it needs:

1- Hypothesis: $z = f\left(b + \sum_{i=1}^{n} x_i w_i\right)$

2-Cost Function: $J(\omega_0, \omega_1)$

3-Goal: minimize $J(\omega_0, \omega_1)$



Andrew Ng

Optimization

Optimization is the process of finding the set of parameters/weights that minimize the **cost function**.

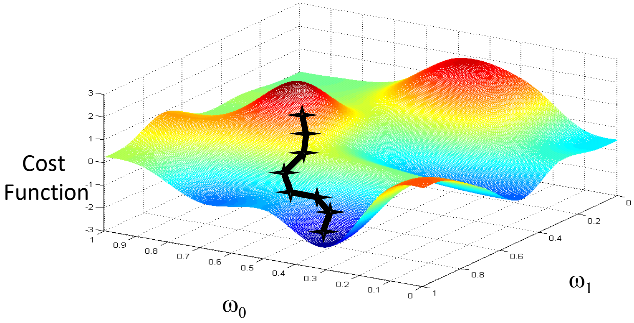
Strategy 1: A first very bad idea solution: RANDOM SEARCH

- is to simply try out many different random parameters and keep track of what works best. (iterative refinement.)
- start with random weights and iteratively refine them over time to get lower cost

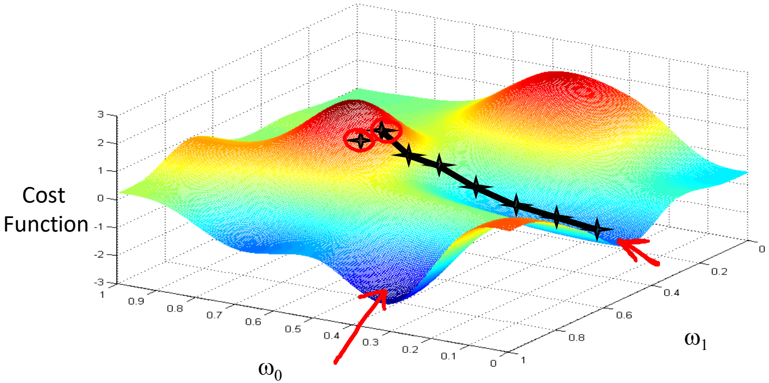
Optimization

Strategy 2: Following the Gradient

- Compute the best direction along which we should change our parameter (weight) vector that is mathematically guaranteed to be the direction of the **steepest descend.**
- This direction will be related to the **gradient** of the cost function.



Gradient Descent



Andrew Ng

Two ways to compute the gradient

There are two ways to compute the gradient:

1) **Numerical gradient:** A slow, approximate but easy way to implement. Approximate (since we have to pick a small value of *h*, while the true gradient is defined as the limit as *h* goes to zero), and that it is very computationally expensive to compute

2) **Analytic gradient:** A fast, exact but more error-prone way that requires **calculus**. It allows us to derive a direct formula for the gradient (no approximations) that is also very fast to compute.

Always use analytic gradient, but check implementation with numerical gradient. This is called **a gradient check.**

Gradient Descent

- We make steps down the cost function in the direction with the steepest descent, and the size of each step is determined by the parameter **α**, which is called the **learning rate.**
- Type equation here.
- The gradient descent algorithm is:

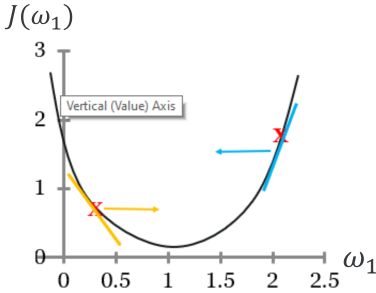
repeat until **convergence** {

$$\omega_j := \omega_j - \alpha \frac{\partial}{\partial \omega_j} J(\omega_0, \omega_1)$$

(for $j = 0$ and $j = 1$)

}

Learning rate (step size)



Positive slope (positive number)→ ω_1 will decrease
Negative slope (negative number)→ ω_1 will increase

Andrew Ng

Gradient Descent variants

There are three variants of gradient descent based on the amount of data used to calculate the gradient:

1. Batch gradient descent
2. Stochastic gradient descent
3. Mini-batch gradient descent

Batch Gradient Descent

Batch Gradient Descent, aka **Vanilla gradient descent**, calculates the error for each observation in the dataset but performs an update only after all observations have been evaluated.

One cycle through the entire training dataset is called a training epoch. Therefore, it is often said that batch gradient descent performs model updates at the end of each training epoch.

Batch gradient descent is not often used, because it represents a huge consumption of computational resources, as the entire dataset needs to remain in memory.

Stochastic Gradient Descent (SGD)

Stochastic gradient descent, often abbreviated **SGD**, is a variation of the gradient descent algorithm that calculates the error and updates the model for each example in the training dataset.

SGD is usually faster than batch gradient descent, but its frequent updates cause a higher variance in the error rate, that can sometimes jump around instead of decreasing.

The noisy update process can allow the model to **avoid local minima** (e.g. premature convergence).

Mini-Batch Gradient Descent

Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.

It is the *most common implementation* of gradient descent used in the field of deep learning.

It splits the training dataset into **small batches** that are used to calculate model error and update model coefficients.

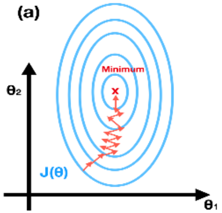
"Batch size" commonly used as power of 2: 32, 64, 128, 256, and so on.

Gradient Descent in Practice I - Feature Scaling

- We can speed up gradient descent by having each of our input values in roughly the **same range**.

Rule a thumb regarding acceptable ranges

- -3 to +3 is generally fine - any bigger bad
- -1/3 to +1/3 is ok - any smaller bad



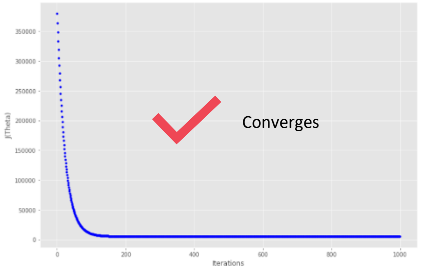
Un-scaled features

Feature Scaling	Mean Normalization
$x_i = \dfrac{x_i}{max_{x_i}}$ $max_{x_i} = largest\ value\ of\ x$	$x_i = \dfrac{x_i - \mu_i}{s_i}$ $\mu_i = \ mean\ value\ of\ x$ $s_i =$ range of values (max – min) or the standard deviation.

Gradient Descent in Practice II - Learning Rate

- **Debugging gradient descent.**
- Make a plot with *number of iterations* of gradient descent on the $x-axis$ and the cost function $J(\omega)$ on the $y-axis$.
- $J(\omega)$ should decrease after each iteration else decrease α

Automatic convergence test. Declare convergence if $J(\omega)$ decreases by less than ε in one iteration, where ε is some small value such as 10^{-3}.
However in practice it's difficult to choose this threshold value.

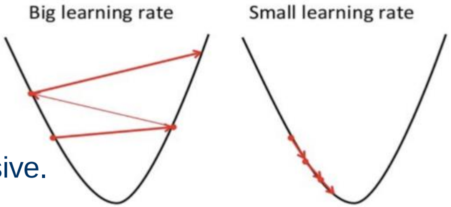


Learning Rate

The gradient tells us the direction, but it does not tell us how far along this direction we should step.

The **learning rate (step size)** determines how big the step would be on each iteration. It determines how fast or slow we will move towards the optimal weights.

If the **learning rate is large**,
it may fail to converge and
overshoot the minimum.
If the **learning rate is very small**,
it would take long time to converge
And become computationally expensive.



The most commonly used rates are :

0.001, 0.003, 0.01 (default), 0.03, 0.1, 0.3

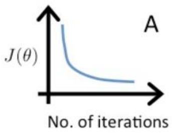
Gradient Descent in Practice II - Learning Rate

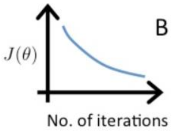
It has been proven that if learning rate α is sufficiently small, then $J(\omega)$ will decrease on every iteration. When Gradient Descent can't decrease the cost-function anymore and remains more or less on the same level, we say it has **converged**.

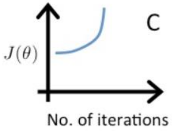
Note:
If you see in the plot that your learning curve is just going up and down, without really reaching a lower point, you also should try to decrease the learning rate.

If α is too small:
slow convergence.

If α is too high:
the cost function is increasing



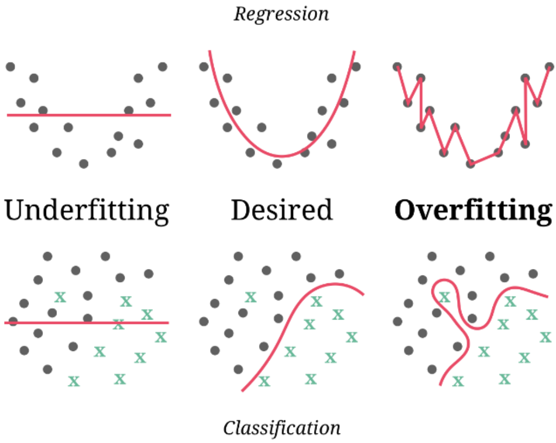




The Problem of Overfitting

- **Underfitting**, or **high bias**, is when the form of our hypothesis function h maps poorly to the trend of the data. It is usually caused by a function that is too simple or uses too few features.

- At the other extreme, **overfitting**, or **high variance**, is caused by a hypothesis function that fits the available data but does not generalize well to predict new data. It is usually caused by a complicated function that creates a lot of unnecessary curves and angles unrelated to the data.



Model Diagnosis

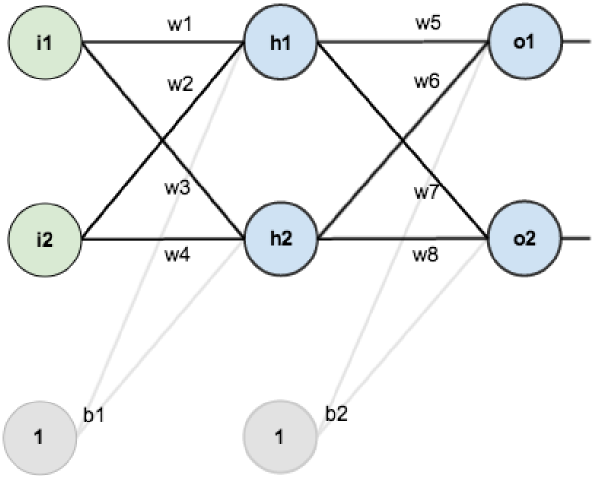
Debugging a learning algorithm:

Suppose you have implemented regularized linear regression to predict housing prices. However, when you test your hypothesis in a new set of houses, you find that it makes unacceptably large errors in its prediction. What should you try next?

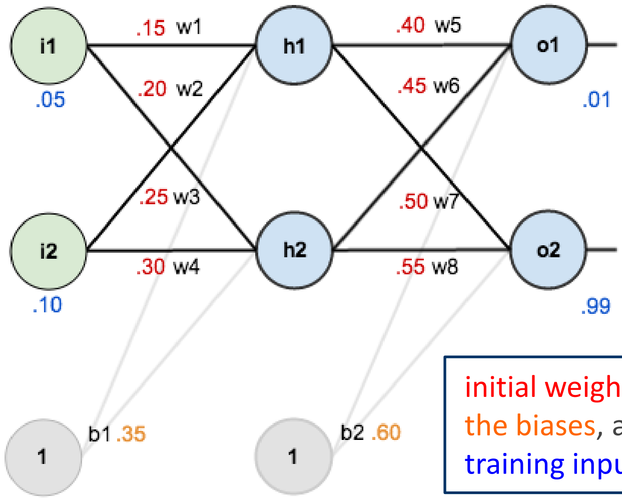
- Get more training examples *fix high variance*
- Try smaller sets of features *fix high variance*
- Try getting additional features *fix high bias*
- Try adding polynomial features $(x_1^2, x_2^2, x_1x_2, \text{etc})$ *fix high bias*
- Try decreasing λ *fix high bias*
- Try increasing λ *fix high variance*

Andrew Ng

Numerical Example [Matt Mazur]

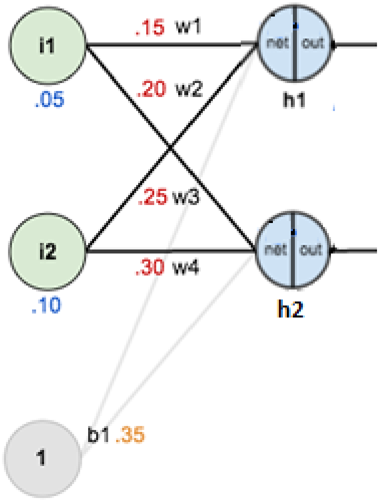


Numerical Example [Matt Mazur]



initial weights
the biases, and
training inputs/outputs

Numerical Example: Forward Pass-hidden layer



Calculate total net input for h_1 (net_{h1})

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

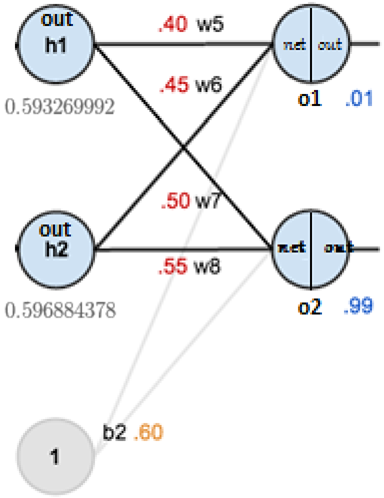
Apply the sigmoid function to get the output of h_1 (out_{h1})

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

Carrying out the same process for h_2, we get:

$$out_{h2} = 0.596884378$$

Numerical Example: Forward Pass-output layer



Repeat same process for output layer: Calculate total net input for o_1 (net_{o1})

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1$$

$$= 1.105905967$$

Apply the sigmoid function to get the output of o_1 (out_{o1})

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

Carrying out the same process for o_2, we get:

$$out_{o2} = 0.772928465$$

Numerical Example: Calculate Total Error

Calculate the error for each output neuron using the squared error function and sum them to get the total error:

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

For example, the target output for o1 is 0.01 but the neural network output 0.75136507, therefore its error is:

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

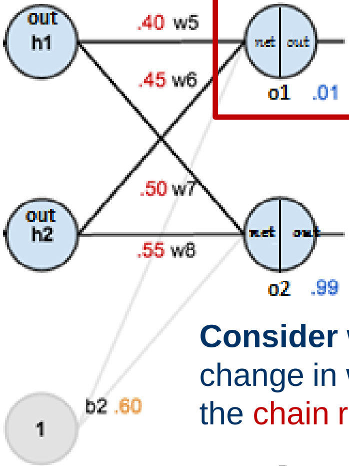
Repeating this process for o2 (remembering that the target is 0.99) we get:

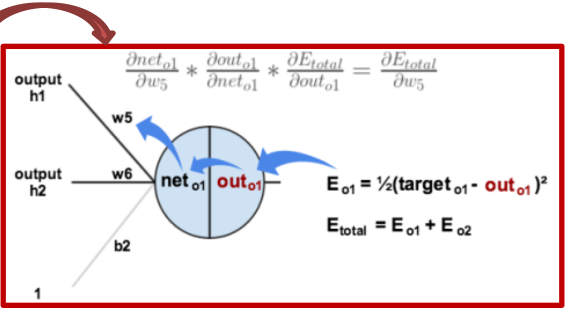
$$E_{o2} = 0.023560026$$

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

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

Numerical Example: Backward Pass-output layer



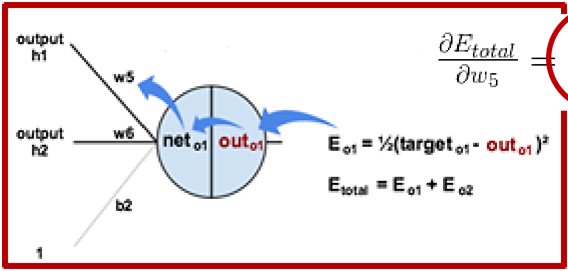


$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$
output h1
w5
output h2
w6
net $_{o1}$
out $_{o1}$
$E_{o1} = ½(target_{o1} - out_{o1})^2$
$E_{total} = E_{o1} + E_{o2}$
b2
1

Consider w5, we want to know how much a change in w5 affects the total error. By applying the chain rule we know that:

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

Numerical Example: Backward Pass-output layer



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

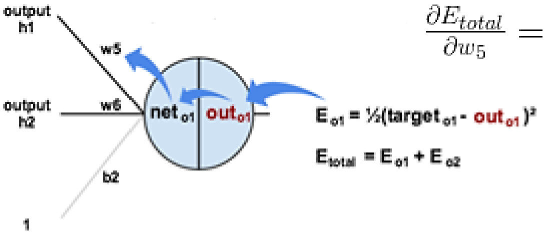
1- The total error change with respect to the output o1

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

Numerical Example: Backward Pass-output layer


output h1
output h2
w5
w6
b2
1
net_{o1}
out_{o1}
E_{o1} = ½(target_{o1} - out_{o1})²
E_{total} = E_{o1} + E_{o2}

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

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

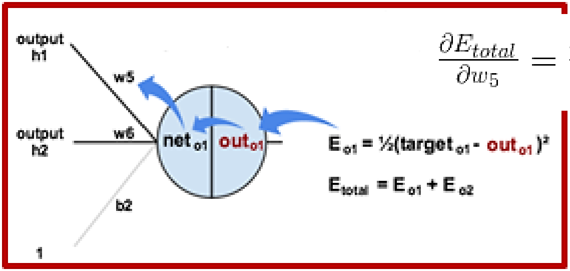
$$\frac{d\sigma(x)}{dx} = \sigma(x)\,(1-\sigma(x))$$

2- The output o1 change with respect to its total net input

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

Numerical Example: Backward Pass-output layer



$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2$

$E_{total} = E_{o1} + E_{o2}$

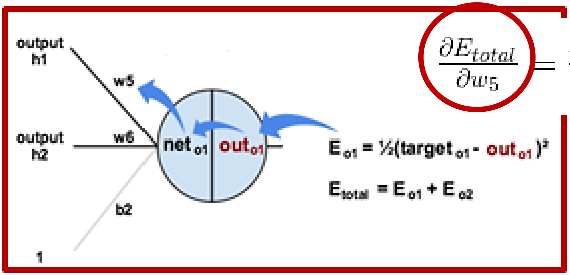
$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \boxed{\frac{\partial net_{o1}}{\partial w_5}}$$

3- The total net input of o1 change with respect to w5

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

Numerical Example: Backward Pass-output layer


output h1
w5
output h2
w6
b2
1
net o1
out o1
$E_{o1} = ½(\text{target}_{o1} - \text{out}_{o1})^2$
$E_{total} = E_{o1} + E_{o2}$

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

4- Putting all together

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

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

Numerical Example: Backward Pass-output layer

5- 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):

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

- We can repeat this process to all weights of output layer to get the new weights w6, w7 and w8:

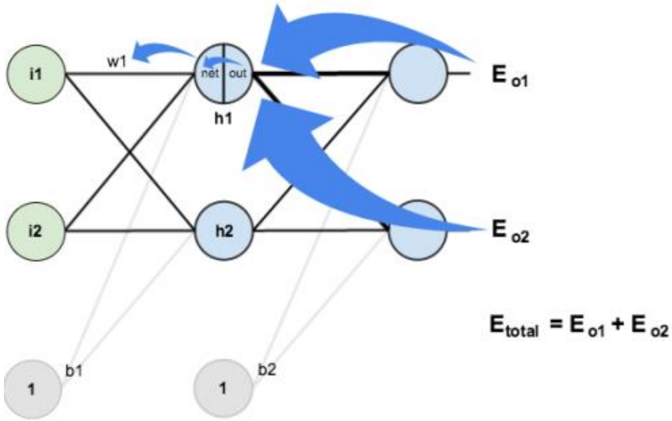
$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

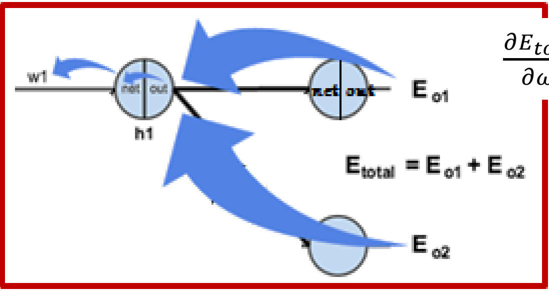
Numerical Example: Backward Pass-hidden layer

we'll continue the backwards pass by calculating new values for w1, w2,w3 and w4



$$\frac{\partial E_{total}}{\partial \omega_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial \omega_1}$$ where, $$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

Numerical Example: Backward Pass-hidden layer



$$\frac{\partial E_{total}}{\partial \omega_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial \omega_1}$$

Where,

$$\frac{\partial E_{total}}{\partial out_{h1}} = \boxed{\frac{\partial E_{o1}}{\partial out_{h1}}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

1- Starting with: $\frac{\partial E_{o1}}{\partial out_{h1}} = \underbrace{\frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}}} * \frac{\partial net_{o1}}{\partial out_{h1}}$

$= 0.74136507 * 0.186815602$

$= 0.138498562$

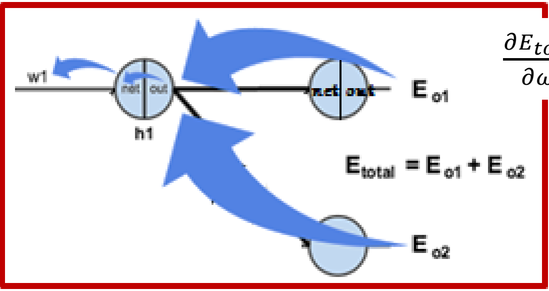
using values calculated earlier

$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$

$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40 \rightarrow$ Not updated weight

$$\frac{\partial E_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

Numerical Example: Backward Pass-hidden layer



$$\frac{\partial E_{total}}{\partial \omega_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial \omega_1}$$

Where,

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

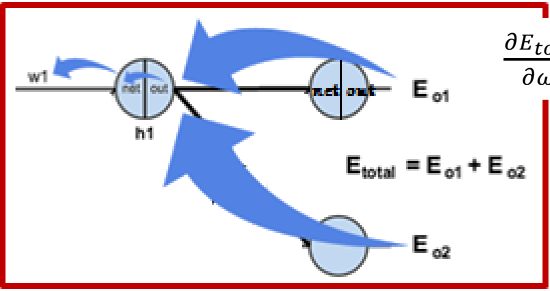
Following the same process for $\frac{\partial E_{o2}}{\partial out_{h1}}$, we get:

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

Therefore:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

Numerical Example: Backward Pass-hidden layer



$$\frac{\partial E_{total}}{\partial \omega_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \overset{2}{\frac{\partial out_{h1}}{\partial net_{h1}}} * \overset{3}{\frac{\partial net_{h1}}{\partial \omega_1}}$$

Where,

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

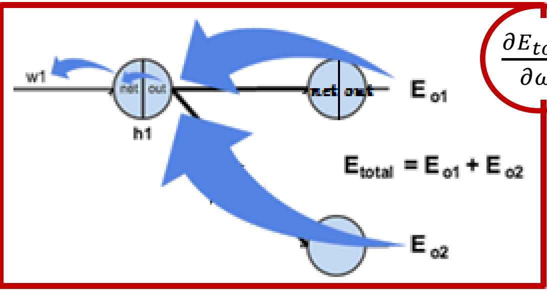
2- $out_{h1} = \frac{1}{1+e^{-net_{h1}}}$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

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

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

Numerical Example: Backward Pass-hidden layer



$$\boxed{\frac{\partial E_{total}}{\partial \omega_1}} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial \omega_1}$$

Where,

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

4- Put all together

$$\frac{\partial E_{total}}{\partial \omega_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial \omega_1}$$

$$\frac{\partial E_{total}}{\partial \omega_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

Numerical Example: Backward Pass- hidden layer

5- We can update w1:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

- We can repeat this process to all weights of hidden layer to get the new weights w2, w3 and w4:

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

Finally,

- When we fed forward the 0.05 and 0.1 inputs originally,
    - Total error was 0.298371109.
- After this first round of backpropagation,
    - Total error is now 0.291027924.
- But after repeating this process 10,000 times, for example,
    - Total error drops 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).

This lecture references

- https://www.superdatascience.com/blogs/the-ultimate-guide-to-artificial-neural-networks-ann
- https://www.youtube.com/watch?v=GvQwE2OhL8I

- **Matt Amzur:** https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
- https://towardsdatascience.com/applied-deep-learning-part-1-artificial-neural-networks-d7834f67a4f6