NUS National University of Singapore

School of Computing

Decision Making Technologies for Business

2018/2019 Semester I

Associate Professor HUANG, Ke-Wei

Today’s Class

Part I: Overview of Neural Network (NN), which is also called Artificial Neural Network (ANN)

- How to decide activation functions?
- How many layers?
- How many nodes?

Part II: Brief Review of Gradient Descent Method

Part III: Understand Back-Propagation & Learning Rate

- How to choose learning rate?

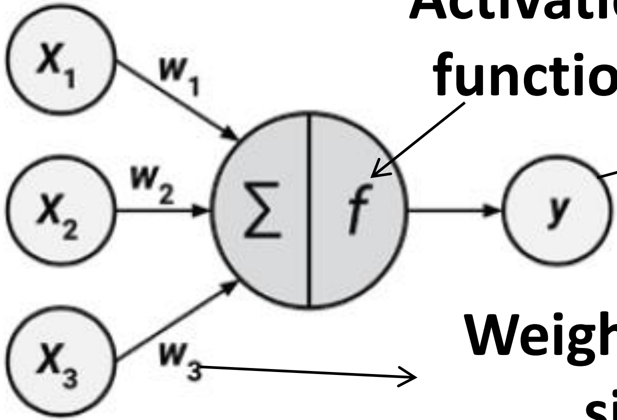
Neural Network: Overview

- Started by psychologists and neurobiologists to develop and test computational analogues of neurons.
- Most references will claim that “brain uses a network of connected neurons to learn”, this motivates the design of artificial neural network for learning.
- The idea of this method is old but it gains popularity not until 1980s and became a hype in academics in 1990s.
- The reason is that one weakness of the original NN method is “very slow”; without fast enough computer, this method is meaningless.
- Algorithms with wide range of successful applications: NN => SVM => Deep Learning NN

Each Neuron in NN

Input vector x: this can be original features or the output from other neurons.

Activation function



Final Output or Output as an input to other neurons

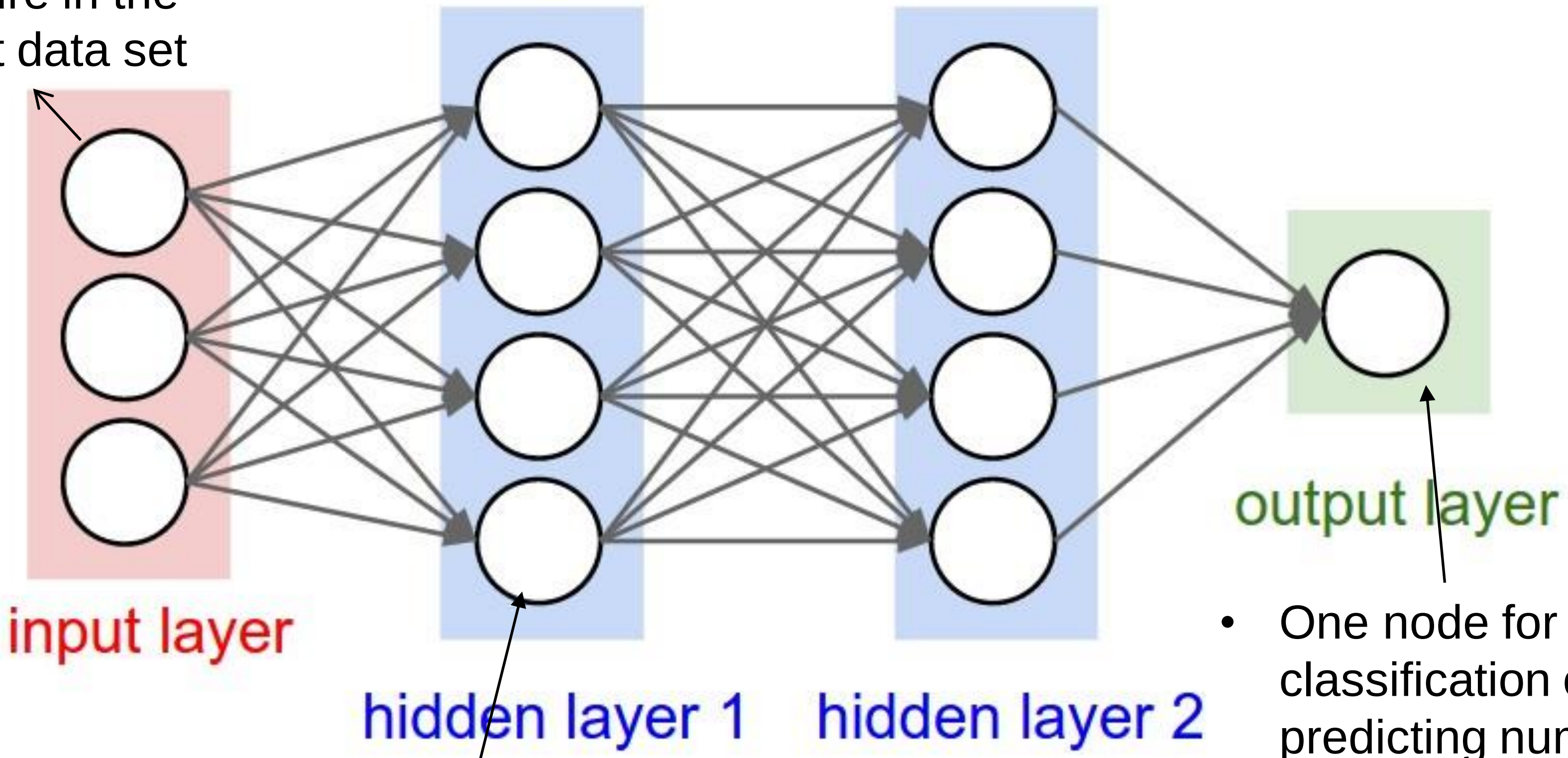
Weight vector w. This is similar to beta coefficients in regression.

$$y(x) = f\left(\sum_{i=1}^{n} w_i x_i\right)$$

Function f() is specified by the users as an input parameter. The user can try various different specification of f(). More on this soon.

A General Multi-Layer Neural Network

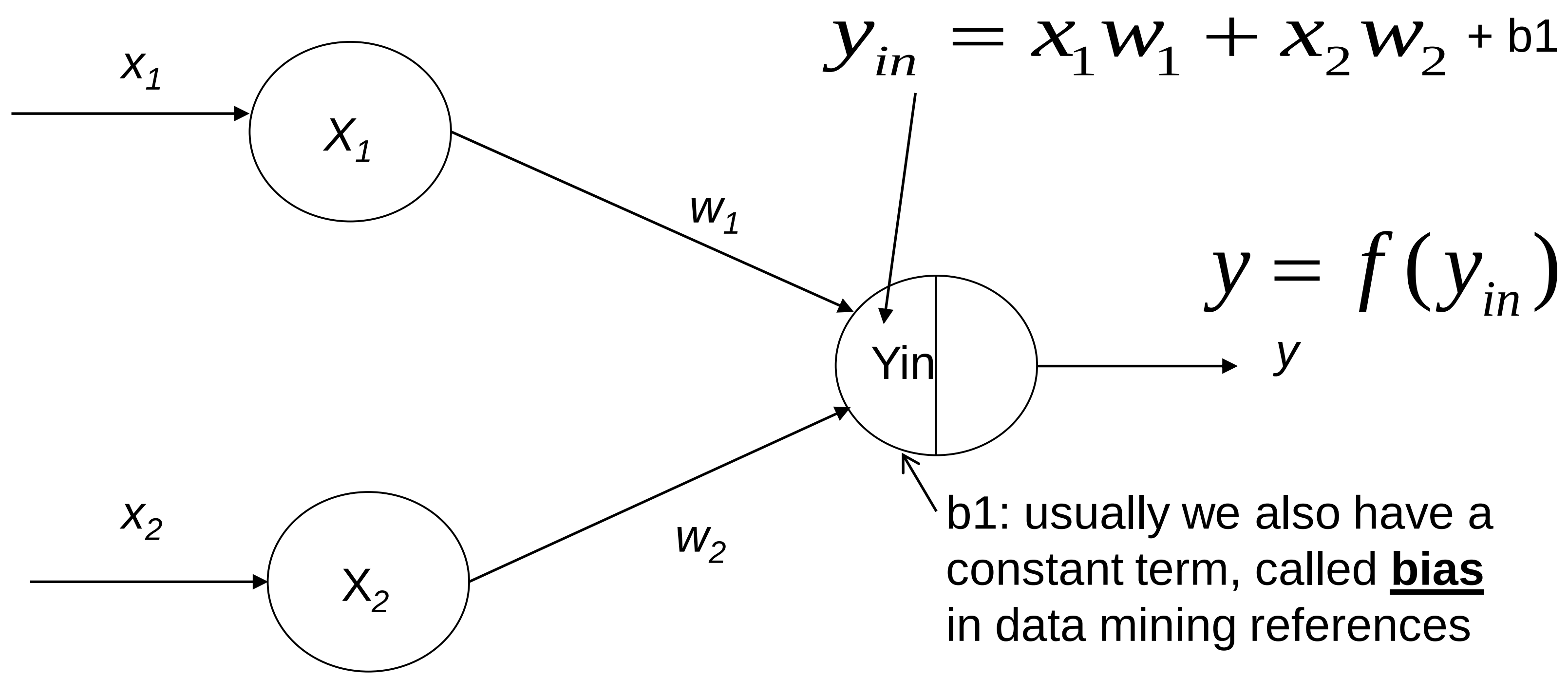
Each one is a feature in the input data set



- One node for binary classification or predicting numbers.
- K output nodes for predicting K class problems with each output is prob(type-i).

Each Circle in the hidden layer and output layer is a Neuron on the previous slide

The Simplest Neural Network



x_1
X_1
w_1
x_2
X_2
w_2
Yin
y
$y_{in} = x_1 w_1 + x_2 w_2$ + b1
$y = f(y_{in})$
b1: usually we also have a constant term, called **bias** in data mining references

- In this simplest example, we have only two input attributes, no hidden layer, one output node Y. When f() is linear, this is equivalent to linear regression. ⬄ The objective function of NN is to find weights to minimize the sum of squared error, which is exactly the same as the objective function in linear regression.
- Similarly, logistic regression is a special case of NN.

Components of Multi-Layer Neural Network

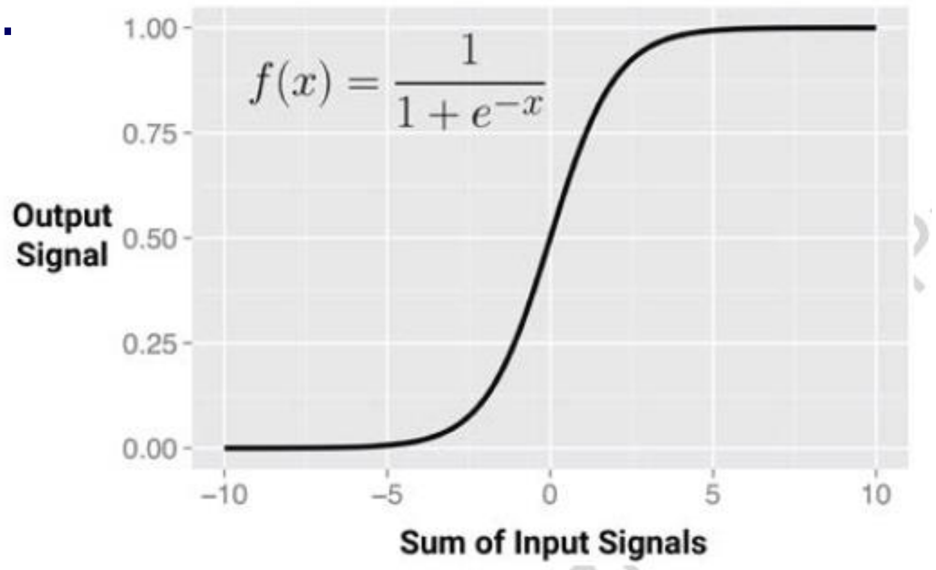
- Inputs from you = Model specification:

1. The activation function used at all nodes
    1. typical package only allows one function for all hidden nodes but conceptually different activation functions can be used at different nodes
    2. Several packages allow you to have different function at the output layer. (One function for hidden nodes and the other for output layer)
2. The number of hidden layers
3. The number of nodes at each layer
4. Back-propagation methods and Learning Rate

- Given 1-4 and the data at the input (features) and output layer (label) in your training set, the NN algorithm will adjust the weights of all neurons to minimize the sum of squared errors at the output layer, similar to OLS regression.

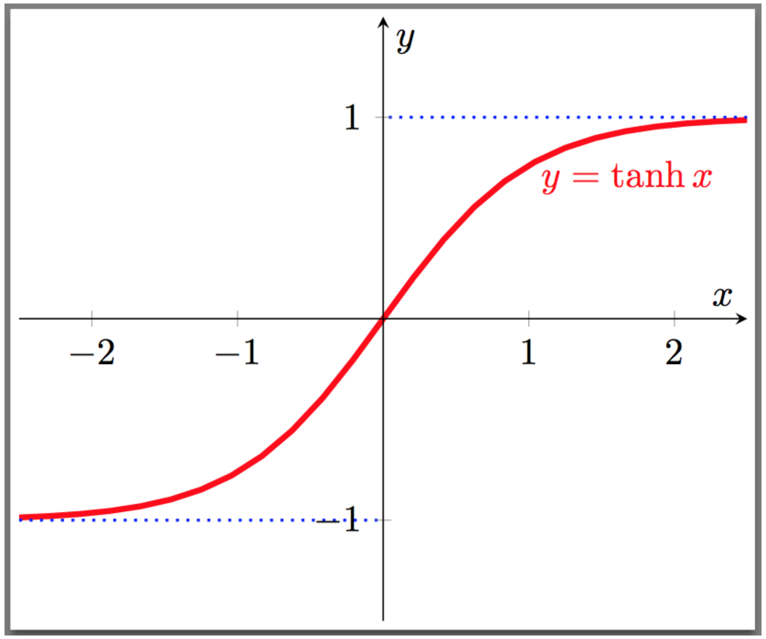
Input 1: Activation Functions

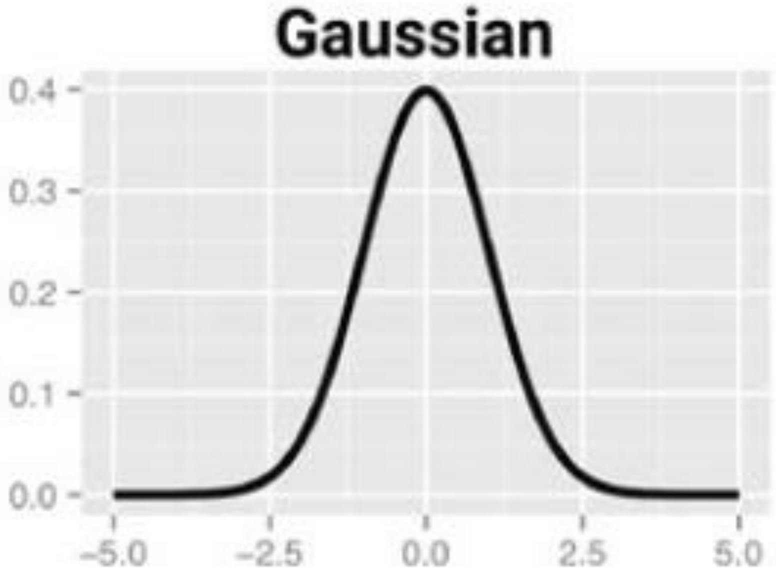
- There are many possible activation functions tried in the literature.
- You should try all to check which works the best in 10-fold cross-validation.
- Linear function is a theoretical benchmark and it won't work well in practice (not flexible enough).
- One old and commonly used alternative is the **sigmoid activation function** (the *logistic* sigmoid) shown in the following figure.



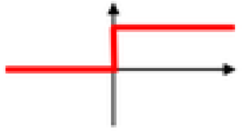
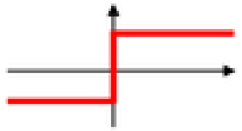
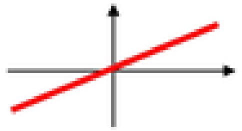
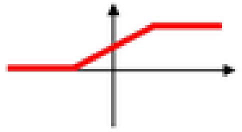
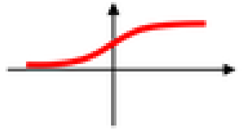
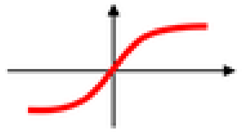
Activation Functions

- Hyperbolic Tangent Function and Gaussian Radial Basis Functions (Gaussian density function) are also very popular choices.





Common Activation Functions

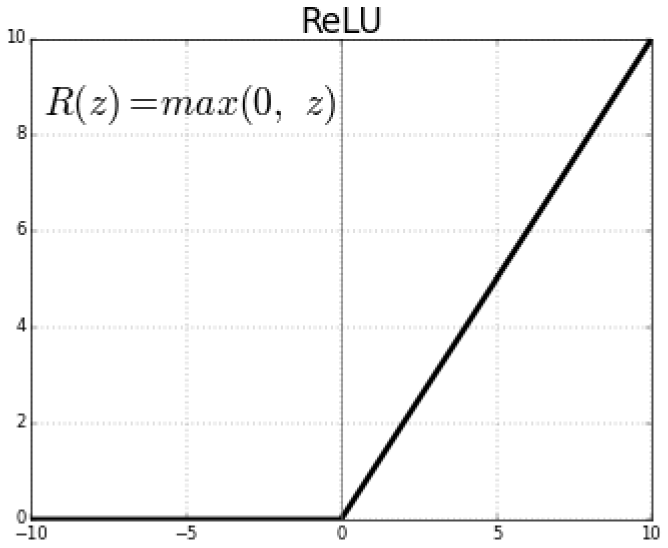
Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \dfrac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$	Multi-layer NN	

ReLU Activation Function

- This is gaining popular in some applications.
- ReLu means Rectifier Linear Unit

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

- The benefit is its slope is either 0 or 1, which improves the speed of neural network training (explained soon).



Squashing Property

- It's important to recognize that for many activation functions, the range of input values that can affect the output is relatively narrow.
- For example, in the case of sigmoid, the output signal is always nearly 0 (or 1) for an input signal below *-5 (*or above *+5)*, respectively.
- Because this essentially squeezes the input values into a smaller range of outputs, activation functions like the sigmoid are sometimes called **squashing functions**.
- **You should better standardize all of your input variables.**
- Outliers in features are less harmful in NN.

Input 2: Number of Layers

- The answer could be surprising to you. For the traditional, standard NN => one hidden layer (but with many nodes) is good enough!
- This theorem is called "**Universal approximation theorem**".
- One hidden layer with sufficient neurons (under very mild assumptions on the activation function in the footnote below) can be used to approximate any continuous function to an arbitrary precision over a finite interval.
- At the same time, multiple layers may learn a complicated pattern faster with fewer number of nodes.

Input 3: Number of Nodes

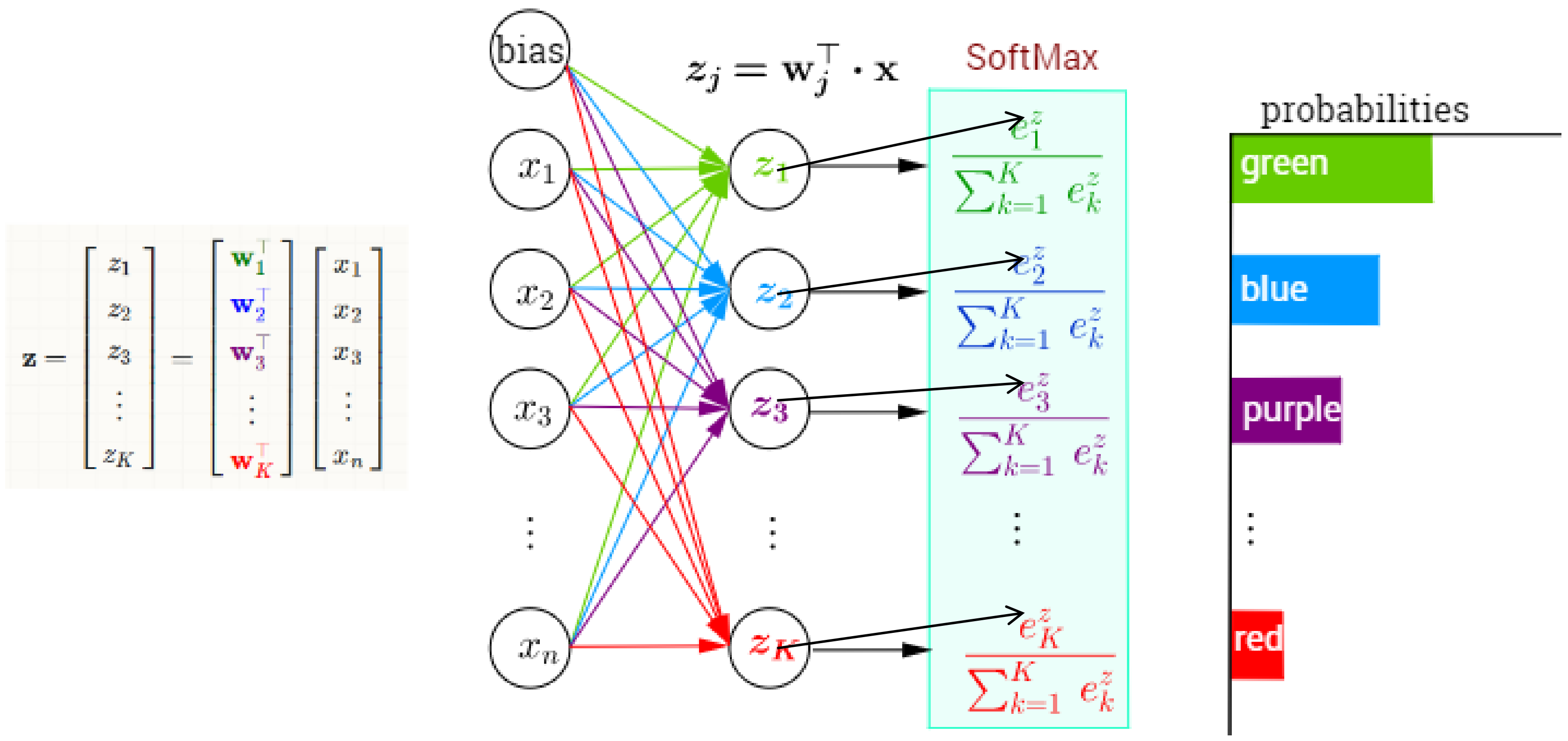
- The input level is predetermined and is the number of features in the input data.
- The output layer is predetermined by the number of numeric outcomes to be modelled or the number of class levels in the outcome.
  - When predicting K-classes, we will use the logistic function (also called soft-max function) to model the predicted probability of each class.
- However, the number of hidden nodes is left to the user to decide during the parameter tuning.
- Unfortunately, there is no reliable rule to determine the number of neurons in the hidden layer.

Number of Nodes of Hidden Layer

- More nodes allows the NN to capture more complicated function between X and Y.
  - More closely mirrors the patterns of the training data.
  - But this runs a risk of overfitting; it may generalize poorly to future data for prediction.
  - Whether it is good to add more nodes depends on your unobservable true model/pattern is a complicated pattern or not.
- The best practice is to use the fewest nodes that result in adequate performance in a validation dataset that is not used in training NN.
- See Neural Network Playground Demo http://playground.tensorflow.org
- Also bear in mind that in general, more nodes will take more time to train NN.

Soft-Max Function for the Output Layer

Multi-Class Classification with NN and SoftMax Function



$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_K \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^\top \\ \mathbf{w}_2^\top \\ \mathbf{w}_3^\top \\ \vdots \\ \mathbf{w}_K^\top \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$
bias
x_1
x_2
x_3
x_n
$z_j = \mathbf{w}_j^\top \cdot \mathbf{x}$
z_1
z_2
z_3
z_K
SoftMax
$\frac{e_1^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_2^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_3^z}{\sum_{k=1}^K e_k^z}$
$\frac{e_K^z}{\sum_{k=1}^K e_k^z}$
probabilities
green
blue
purple
red

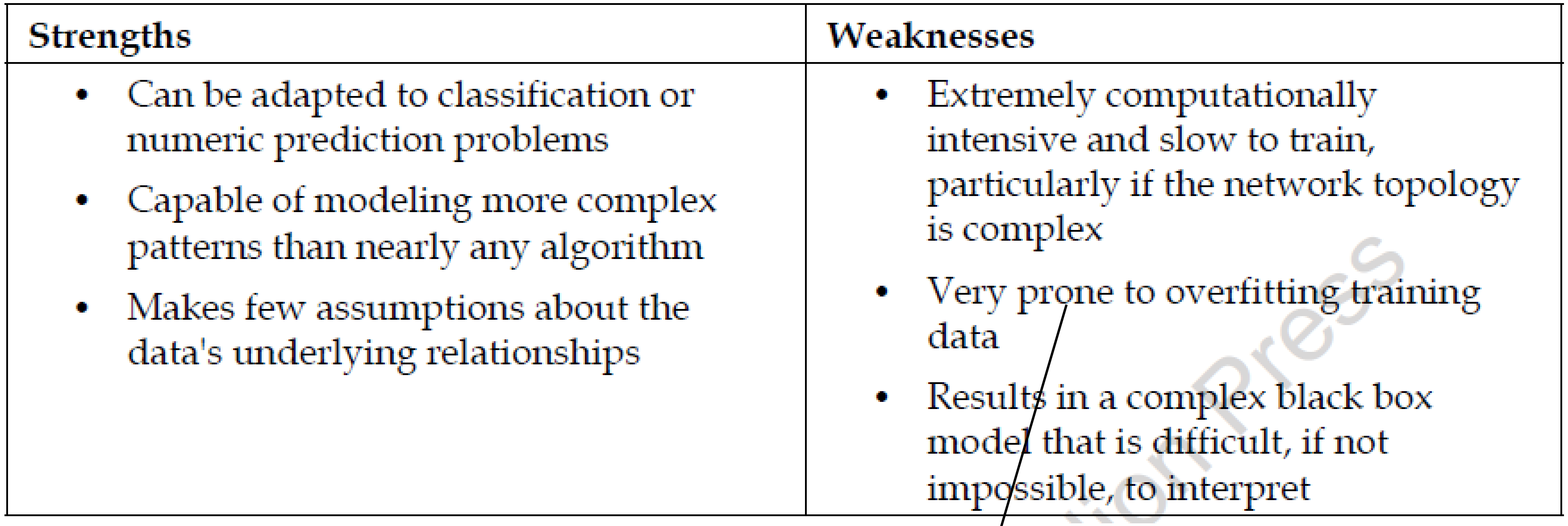
Lessons Learned from Playground

- By 2 original features, linear activation function, one node, one layer, NN can learn only one out of 4 cases. It does not help much even if you add more nodes and/or more layers.
- **<u>Power of features engineering</u>**! With the right features, you can learn difficult cases with one node, one layer, linear activation function
  - X12 means X1^2, X22 means X2^2. Just these two can help you learn circle.
  - X1*X2 can help you learn the exclusive OR case, which is similar to XOR function, an important function (gate) in computer engineering.
- By original features, with 1 or 2 nodes, 1 layer, you cannot learn circle by each of 4 activation functions.
- With 3 nodes, except linear function, all 3 can learn circle but Sigmoid is relatively slow, ReLU is fast but "ugly".

Lessons Learned from Playground

- For learning XOR, we need roughly 4 nodes and except linear function, NN will work to some extent. ReLu works quite well. The other two can provide approximations.
- How to learn the most challenging Spiral case? Without features engineering, it seems difficult to me even with 2 layers and 8 nodes each. With features engineering it is still challenging but possible (with tanh, 1 layer, 5 nodes, NN can learn something similar to a spiral shape).
- The benefits of 2 layers is not obvious (only speed?).
- Tanh function is similar to sigmoid function but indeed performs better in these 4 types of problems.

Strengths and Weakness of NN



Strengths	Weaknesses
• Can be adapted to classification or numeric prediction problems • Capable of modeling more complex patterns than nearly any algorithm • Makes few assumptions about the data's underlying relationships	• Extremely computationally intensive and slow to train, particularly if the network topology is complex • Very prone to overfitting training data • Results in a complex black box model that is difficult, if not impossible, to interpret

Users can specified quite a number of parameters.

Deep Learning Neural Network?

- Deep Neural Network (DNN) is a family of algorithms that extend the basic neural network.
- The most significant feature of DNN is: A network with many layers.
- There are 3 major types of DNN

1. Convolutional Neural Networks (CNN) for image and audio processing.
2. Recurrent Neural Networks and its extensions (e.g., LSTM). (t-1) output used as input to t ⇔ useful for Natural Language Processing or for time series prediction.
3. Auto-encoder network or embedding methods for finding latent factors behind high dimensional input features.
4. Others?

Why Many Layers Work?

- Some of you may wonder why DNN has many layers while we claimed one layer is enough for NN?
- RNN-LSTM is essentially still basic NN but with more complicated time series dependencies.
- CNN and Autoencoder have many layers because goal is to extract useful information from the high-dimensional input features (such as pixels in image and words space in NLP), then we do dimension reduction. In other words, those layers are designed to do “features engineering”, “data augmentation”, and/or dimension reduction.

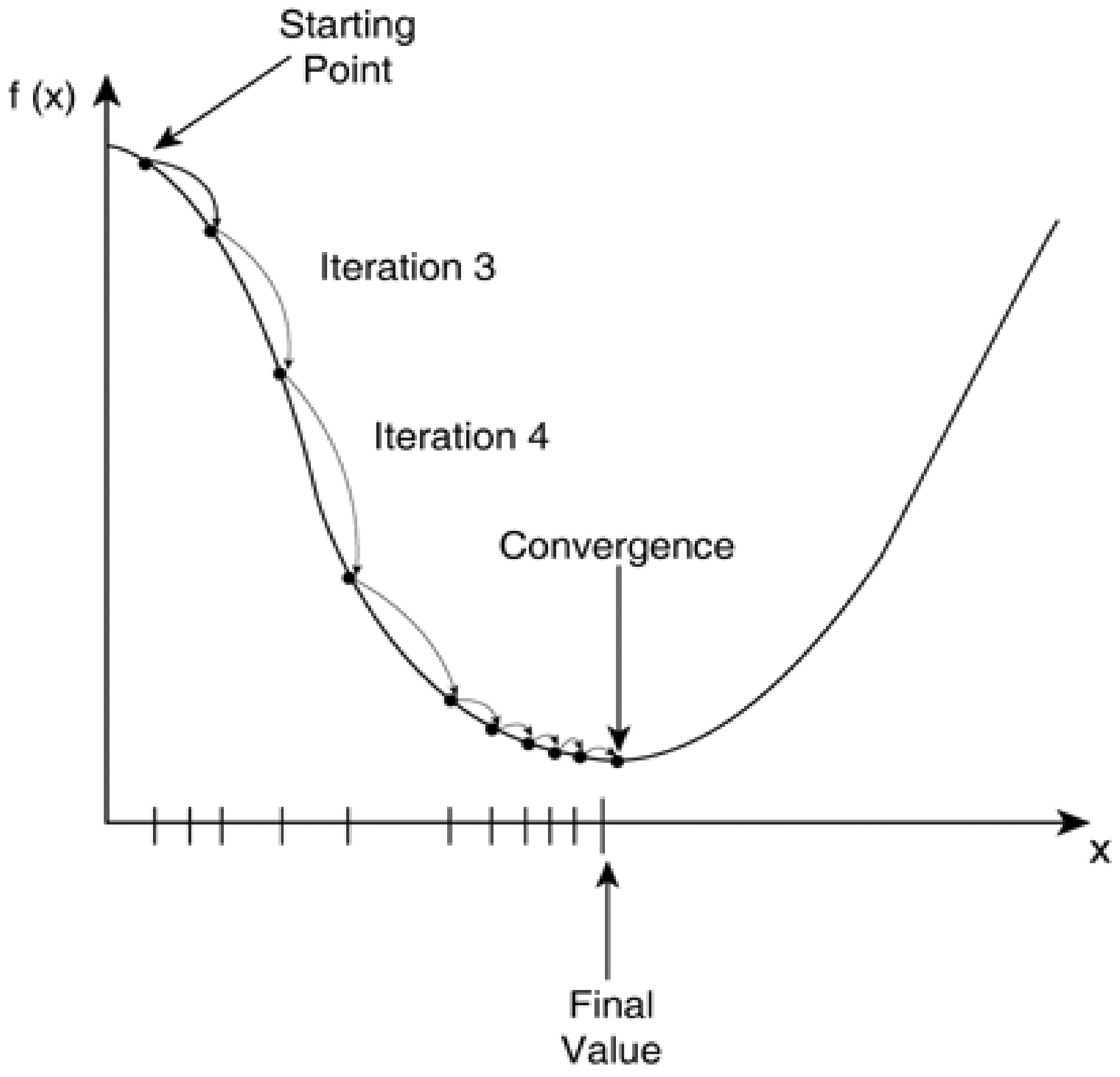
Part II: Review of Numerical Optimization

1. To explain how to train a neural network, we need to explain the gradient descent method first.

2. The goal of this kind of method is to find the global maximum (or equivalently minimum) of any function ⇔ max f(X) or min f(X) by changing the value of X.

3. In one dimensional space, it is easy to visualize this method.

Part II: Review of Numerical Optimization



Gradient Descent Algorithm

There is a learning rate (step), α, as an input parameter.

1. Randomly select the initial value of x0.

2. If you know the value of df(x)/dx, then set
   $x_{(i+1)} = x_{(i)} - \alpha * df(x = x_{(i)})/dx$

3. If $f(x_{(i+1)}) - f(x_{(i)}) < -e$ for minimization, Repeat step 2.

- If you do not know the derivatives of f(x), then we can numerically estimate it by f(x + a small number), but this will be a slow method.
- This algorithm can be easily generalized to high dimensional maximization of $f(x_1, x_2, \ldots, x_n)$ by x.

Remarks

1. There are quite a number of advanced numerical optimization algorithms. For example, Newton method, Conjugate Gradient Method, BFGS, …etc. Those are learning subjects of phd students. You don’t need to know the technical details.

2. The modern neural network training and deep learning neural network training are typically based on variations of an advanced method called **Stochastic gradient descent** (some explanations in the footnote), the property of which is complicated and is beyond the scope of this module.

Part III: How to Train NN?

1. Initialize weights to small random numbers.
2. Propagate the inputs forward.
3. Calculate the error at the output layer.
4. **The key and classical idea is "Back-propagate"** the error for updating weights and biases at each node. The formula for updating weights depends on the activation function and is similar to gradient descent optimization "locally at that node".
5. Terminating condition (when error improvement is very small, etc.), similar to numerical optimization algorithms.

Back-Propagation

- The weights are modified to **minimize the mean squared error** (or other metrics) between the network's prediction and the actual target value.
- Modifications are made in the "**backwards**" direction: from the output layer, through each hidden layer down to the first hidden layer, hence "**backpropagation**".
- "**backpropagation**" is employed because we may have many weights and it becomes slow or not feasible to find the optimal weights all at once.
- Conceptually, "**backpropagation**" is similar to we apply steepest descent method locally from the backward nodes to the earlier nodes so that we can numerically find the weights that **minimize the mean squared error**.

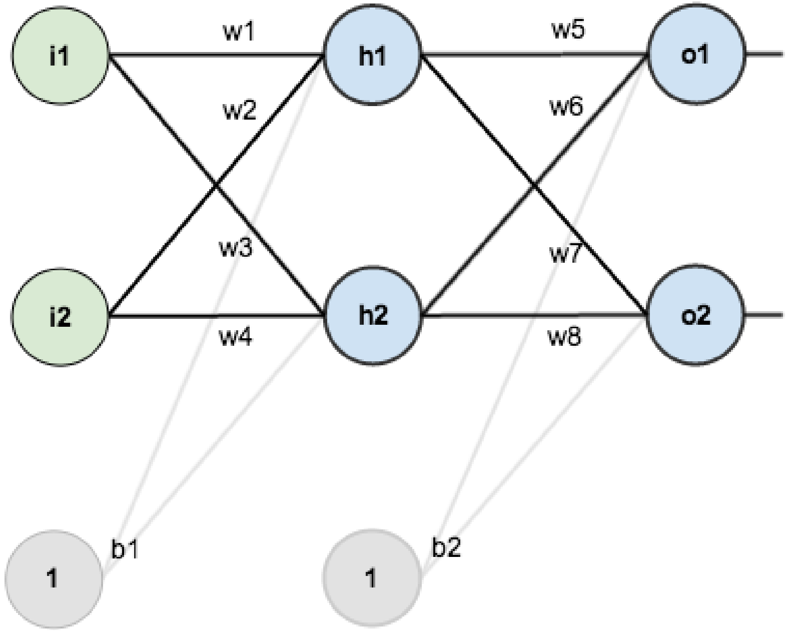
Parameters Tuning Related to Backpropogation

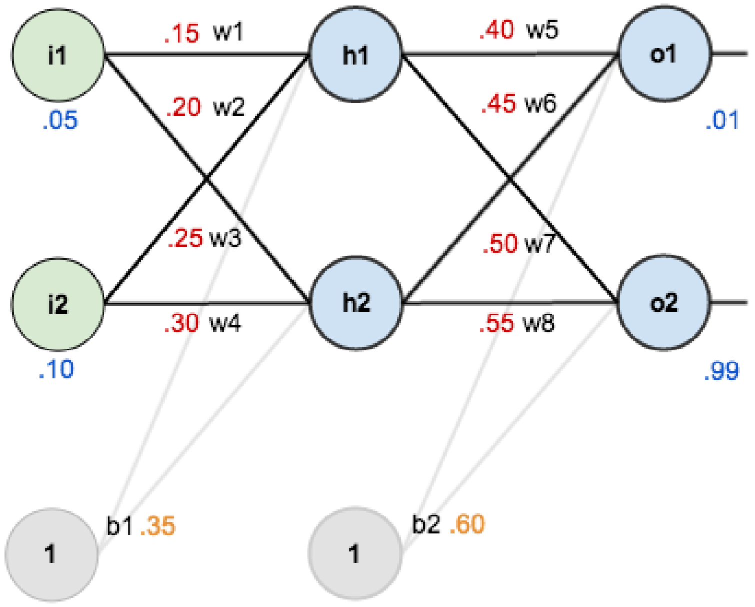
There are several implications

1. Results from NN is a bit random. You re-run and you will get different results (from the same optimal solution).
2. Results from NN may be very different (you reach different local optimal solution)
    1. Better to try several times with different initial values.
3. You can decide terminal conditions such as how many iterations, and threshold for increments are small enough. But if you set it to stop earlier (smaller iterations, larger threshold of improvement), you bear higher risk that results are not really optimal.

Example

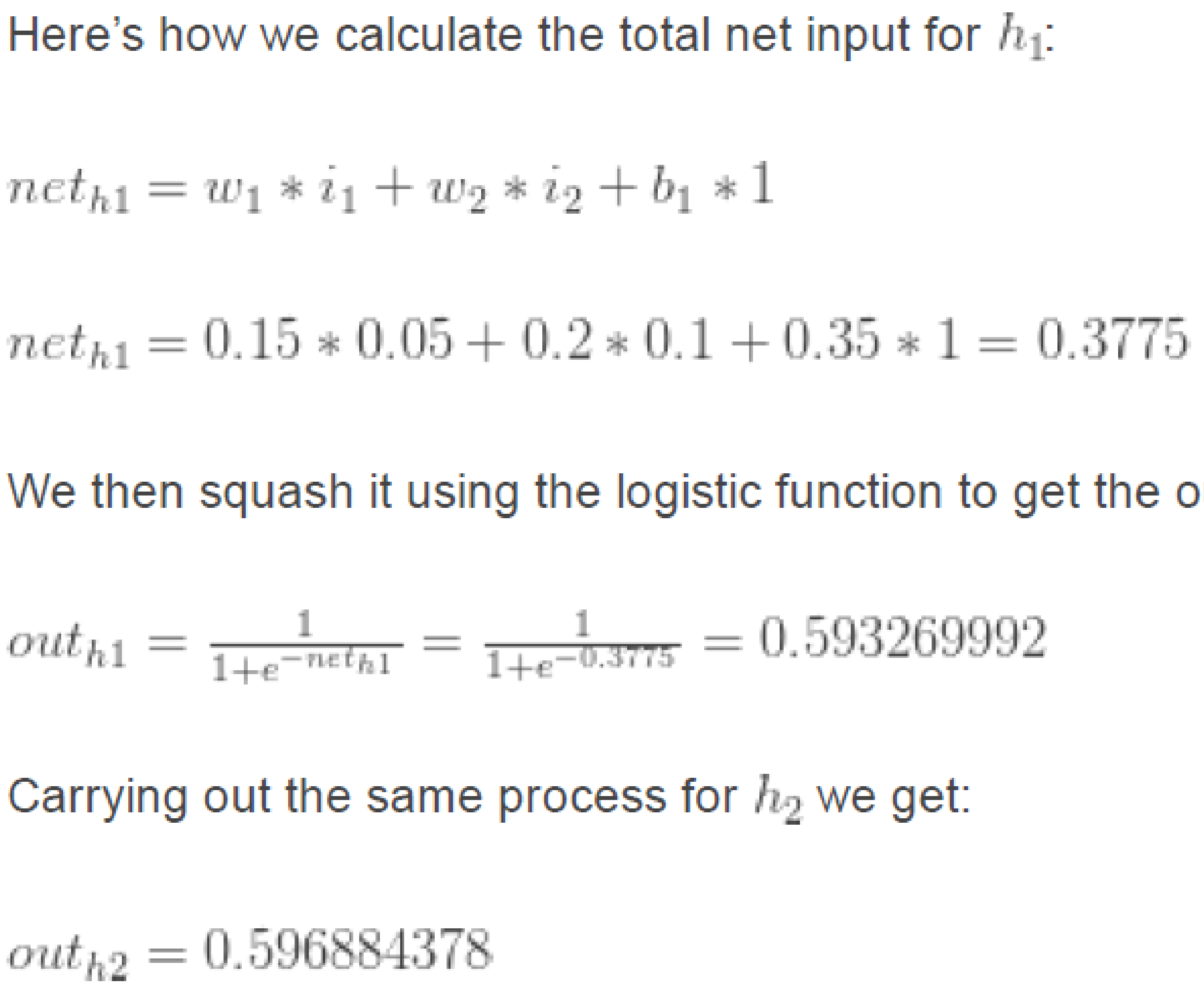
- Reference: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
- 2 inputs, 1 hidden layer with 2 nodes, 2 output nodes. 8 weights and 2 bias (b1, b2) to decide to minimize sum of squared errors.





Example: Feed Forward Step

Given the input data i1, i2 and all initial weights, we feedforward to make a prediction.



Here's how we calculate the total net input for h_1:

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

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

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

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

Carrying out the same process for h_2 we get:

$$out_{h2} = 0.596884378$$

Example: Forward Pass

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:

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

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

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

And carrying out the same process for o_2 we get:

$$out_{o2} = 0.772928465$$

Forward-Pass => Cal. Error

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

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

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

Repeating this process for o_2 (remembering that the target is 0.99) we get:

$$E_{o2} = 0.023560026$$

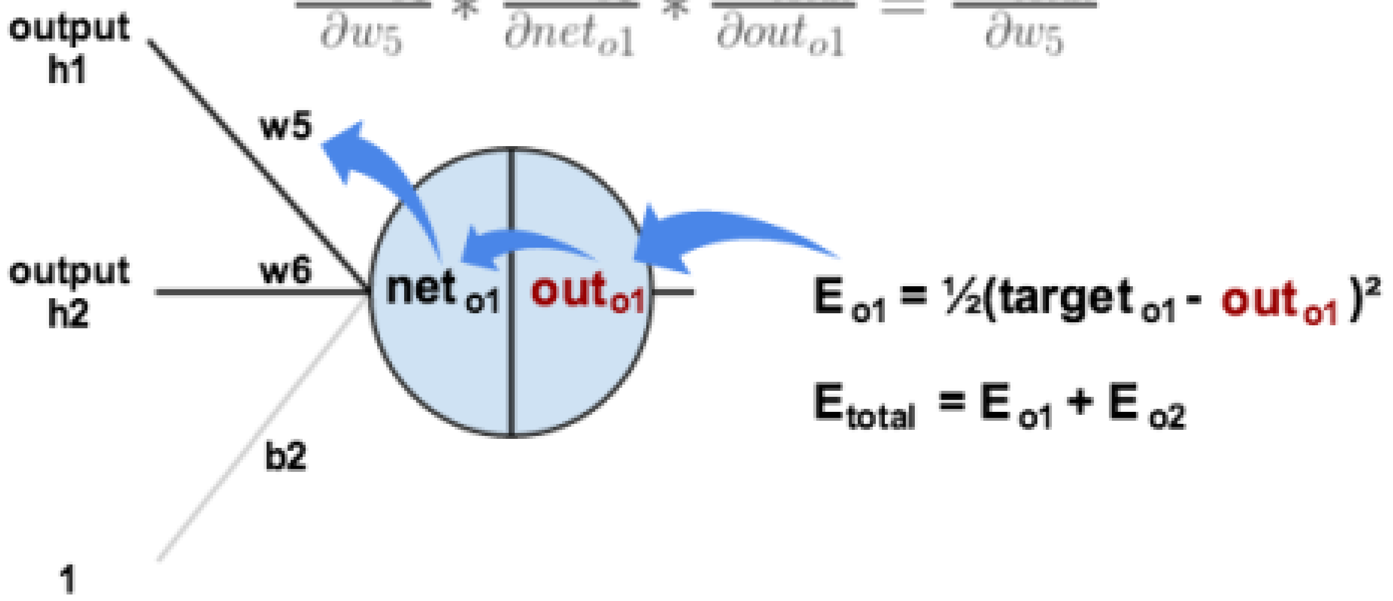
The total error for the neural network is the sum of these errors:

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

The Back-Propagation

- The idea is that if we adjust all 8 weights all together, it may be too complicated when the number of weights increases. The idea is to adjust the weights in each layer and each node only => another example of “Greedy” algorithm used in CS.
- Starting from the output node, we try to calculate if we change the weight by 1 unit, how much is the effect on the error rate => we try to find the gradient given by

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



The Back-Propagation

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

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

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

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

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

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

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

The Back-Propagation

Putting it all together:

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

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

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):

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

The **learning rate** is an important parameter neural network. It is 0.5 in this example.

The Back-Propagation

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

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

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:

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

It is even more complicated for algebra details but is conceptually similar. You can stop here.

How to choose the Learning Rate?

- The learning objectives of these complicated are two-fold: (1) understand back-propagation and (2) understand learning rate.
- But how to choose learning rate in practice?
- Large learning rate: faster to find a solution but you may overshoot and never find the best pattern that fits your training data.
- For example, by playground, predict a circle. activation function is tanh, 1 layer, 3 nodes, learning rate=0.1, you can get a triangle fast.
  - If you choose the smallest learning rate, you need to wait very long.
  - If you set learning rate at 1, you will see what “overshoot” means.

How to choose the Learning Rate?

- Conceptually, you should choose the largest learning rate that can still identify the underlying pattern.
  - Starting with a sane value such as 0.01 and then doing cross-validation to find an optimal value. Typical values range over a few orders of magnitude from 0.0001 up to 1.
  - If your NN is too slow, you can try increasing the learning rate.

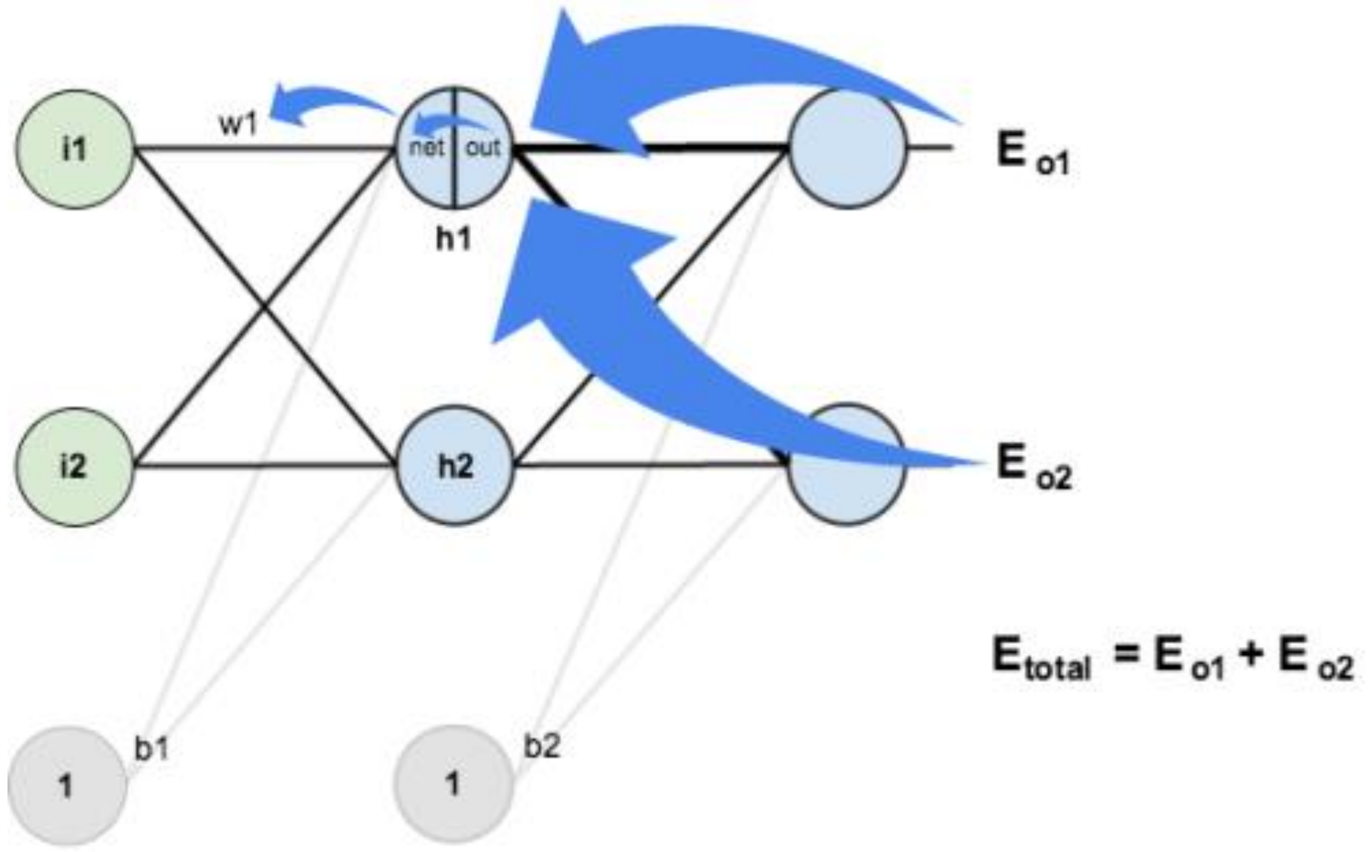
- There are some evolving research that looks at how to automate the selection of learning more optimally and adaptively.
  - Zeiler, Matthew D. "ADADELTA: an adaptive learning rate method." *arXiv preprint arXiv:1212.5701* (2012).
  - Cyclical Learning Rates for Training Neural Networks published in IEEE Winter Conference on Applications of Computer Vision (WACV), 2017.

Appendix: Back-Propagation

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

$$\downarrow$$

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



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

The Complicated Back-Propagation

Starting with $\frac{\partial E_{o1}}{\partial out_{h1}}$:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

We can calculate $\frac{\partial E_{o1}}{\partial net_{o1}}$ using values we calculated earlier:

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

And $\frac{\partial net_{o1}}{\partial out_{h1}}$ is equal to w_5:

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

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

Plugging them in:

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

The Complicated Back-Propagation

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

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

Therefore:

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

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 $\frac{\partial net_{h1}}{\partial w}$ for each weight:

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

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

The Complicated Back-Propagation

We calculate the partial derivative of the total net input to h_1 with respect to w_1 the same as we did for the output neuron:

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

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

Putting it all together:

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

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

We can now update w_1:

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