Linear classification

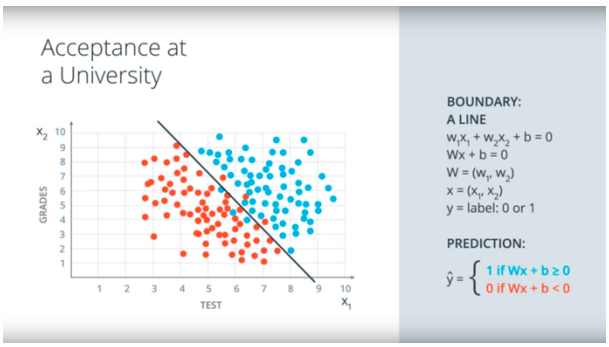
***Objective**. Find the best way to linearly split the two classes.*

***Solution**. A linear equation that fits the data:*

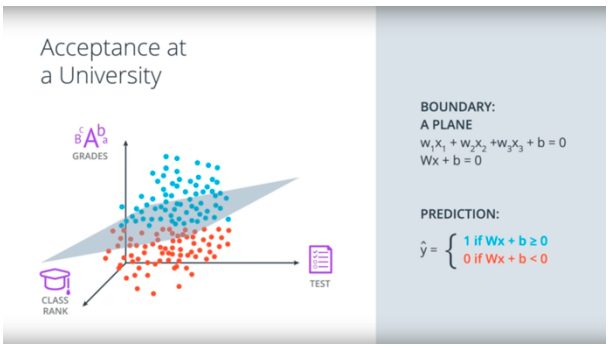
$$w_1x_1 + w_2x_2 + \cdots + w_nx_n + b = 0$$

***Clarifications**. w_i and b are the coefficients of the linear model. w_i represent the weights and b represent the bias unit. The objective is to find the best coefficients.*

2-d case (2 features):



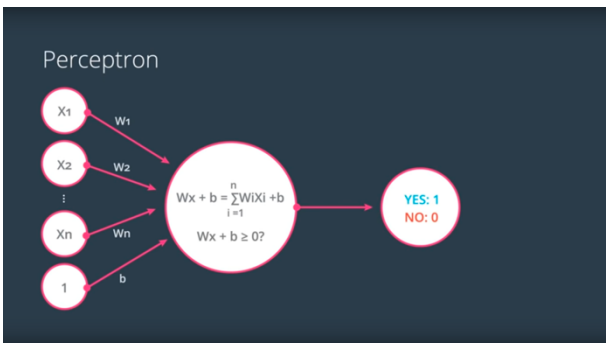
3-d case (3 features):



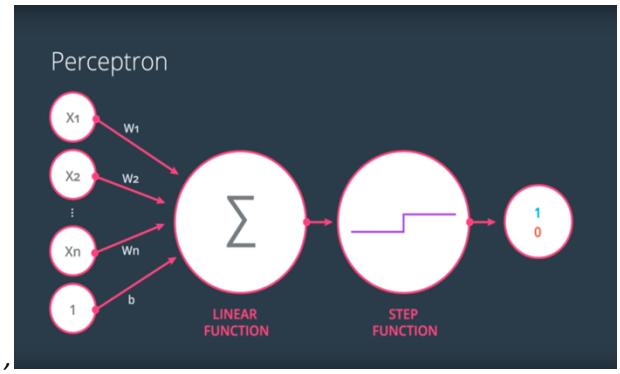
This can be further generalized to n-d hyperplane to accommodate n features.

Perceptron

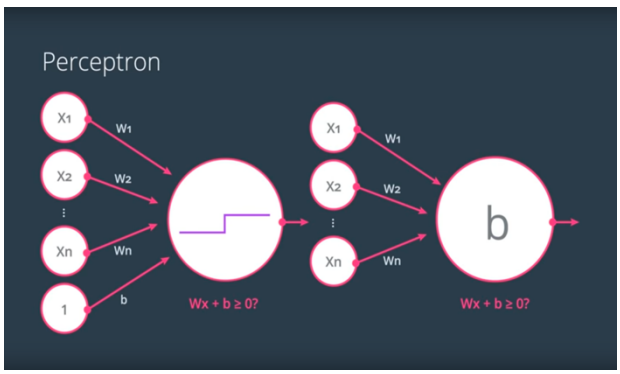
***Definition**. Perceptron is a building block of NN. It helps to find coefficients of a linear equation and define linear decision boundary.*



***Step function.** Perceptron uses the step function to determine whether a point belongs to one class or another. If output of the linear function is more than the threshold (more than zero in our case), perceptron says that the point belongs to class 1, otherwise, it will say that the point belongs to class 0.*

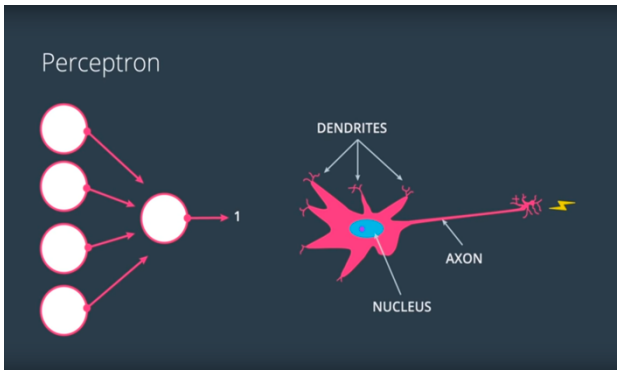


***Bias unit.** There are two ways to represent the bias unit. One way is to add one more feature to the feature set that will be always equal to 1, since the bias unit does not interact with the feature values directly. The second way is to incorporate the bias unit into the neuron that does the computation.*



Brain

Some similarities between artificial neurons and brain neurons:



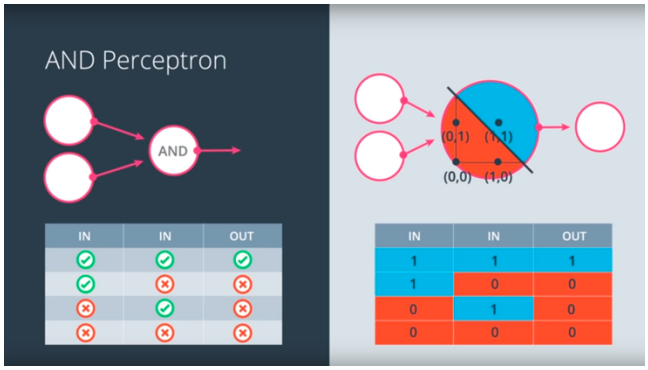
Perceptron and logical operators

Perceptron can function as common logical operators.

For example, ***AND*** *can be represented by the following equation:*

$$1 * x_1 + 1 * x_2 - 1.5 = 0$$

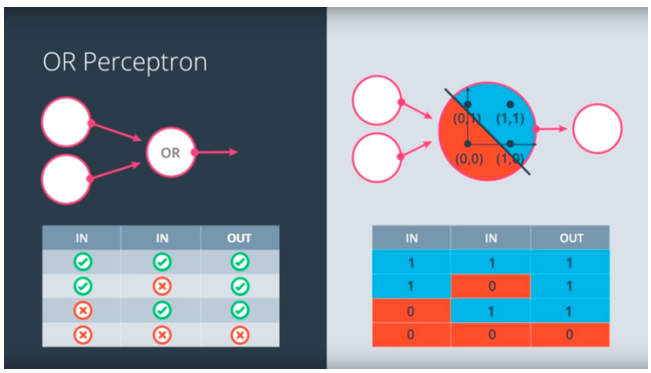
where $w_1 = 1$, $w_2 = 1$ *and* $b = -1.5$



While, ***OR*** *can be represented by the following equation:*

$$1 * x_1 + 1 * x_2 - 1 = 0$$

where $w_1 = w_2 = 1$, *and* $b = -1$

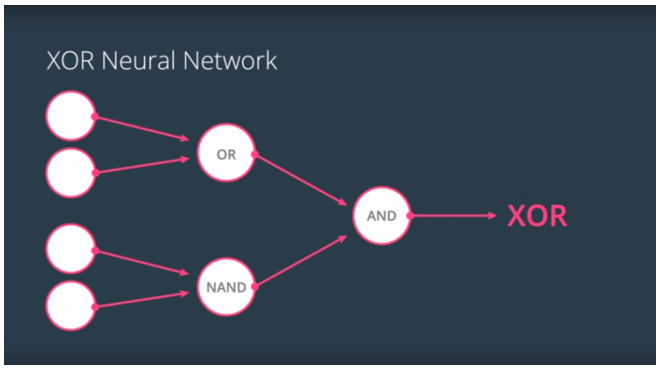


For ***NOT*** *we need only one input, therefore we will set the second weight to zero:*

$$-2 * x_1 + 0 * x_2 + 1 = 0$$

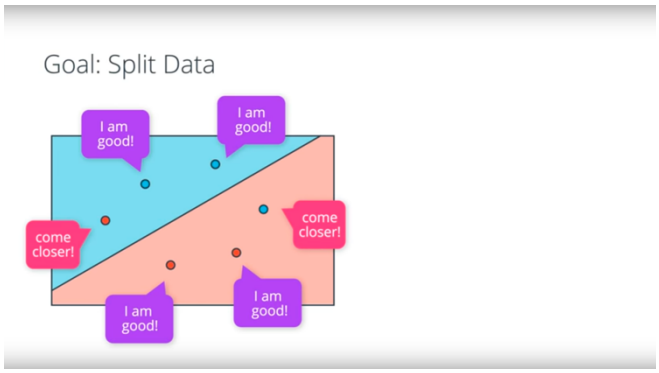
Therefore, if $x_1 = 0$, *then the activation output will be* 1, *if* $x_1 = 1$, *the activation output will be* 0.

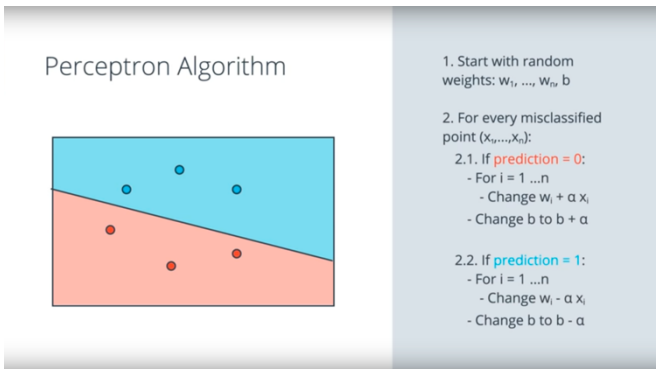
However, XOR operator cannot be represented by just one perceptron unit. We have to combine different perceptrons to assemble it:



Perceptron algorithm

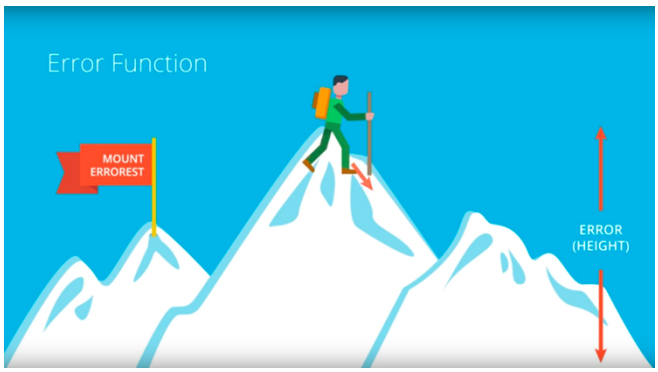
Algorithm*: Iteratively move the line in the direction of misclassified points so eventually it will correctly classify them (if the data is linearly separable).*



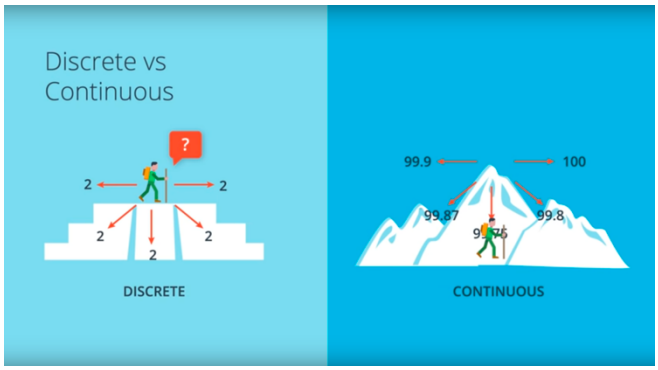


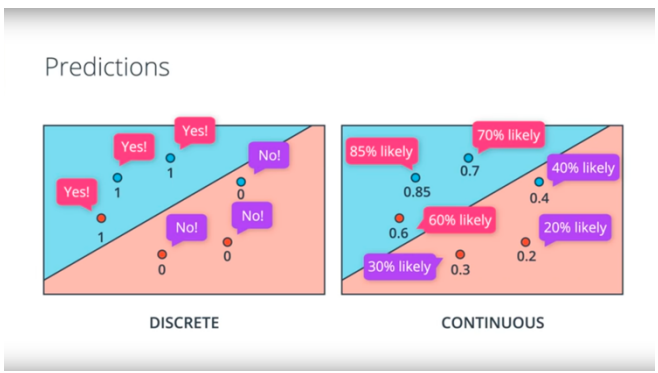
Error functions

***Definition**. Error function if a way to measure how far we are from the optimal solution.*



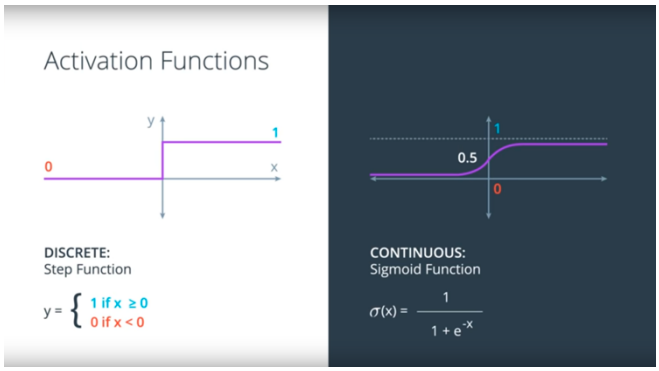
Error function should be able to see arbitrary small variations to show which way to move, that is why it should be continuous. If it is discreet, then there is no way to tell where to go, because every direction may look the same.

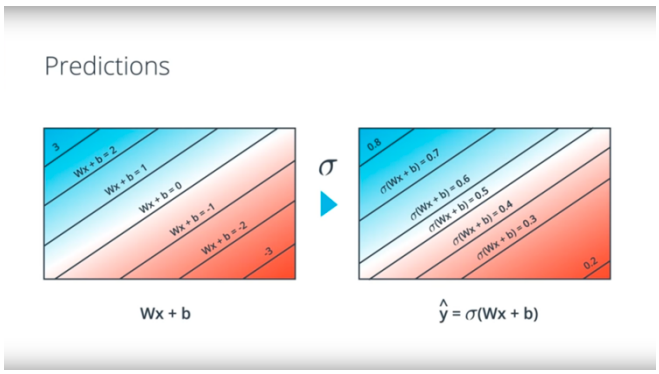




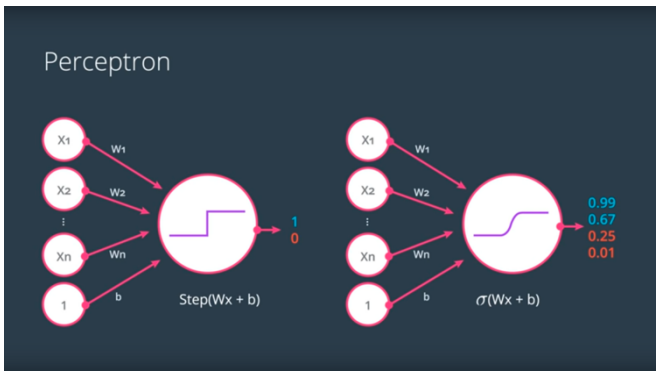
Sigmoid function

***Definition**. Sigmoid function is a function that applies sigmoid nonlinearity to the linear combination and outputs a value between* 0 *and* 1.



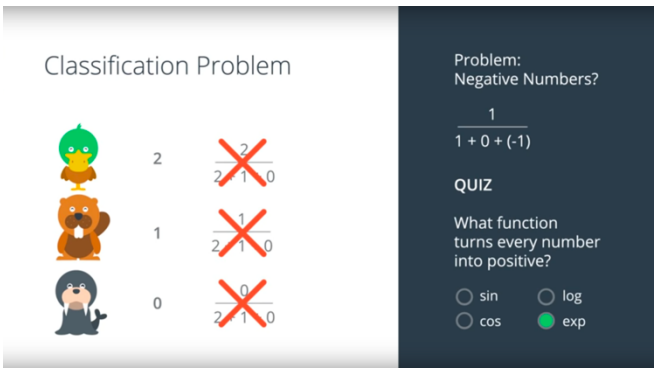


Sigmoid function allows to have fine-grained predictions that can be interpreted as probabilities:

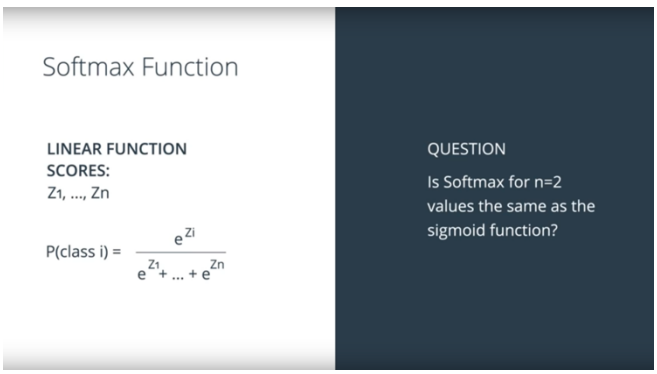


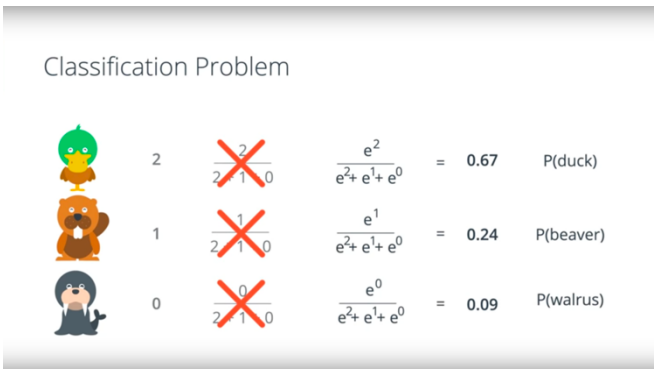
Softmax function

Definition. *Softmax function is a generalization of sigmoid function that is used in the multi-class classification problems (while sigmoid is inherently binary). Softmax function normalizes the scores so that the final outputs for all classes sums up to 1 while individual outputs represent the probability of a point to belong to some class.*



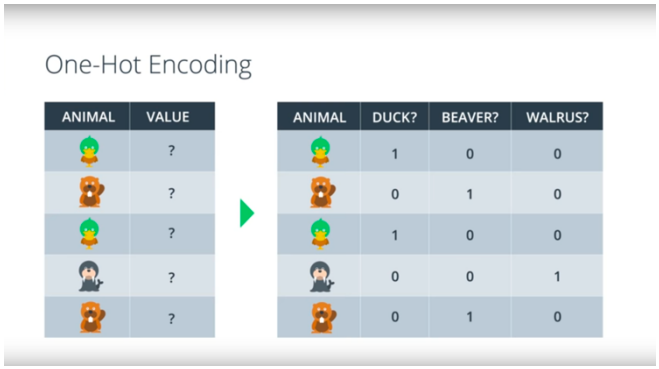
If $n = 2$, softmax function is essentially the same as sigmoid function:





One-hot encoding

Definition. *One-hot encoding is a way to convert categorical variables to binary vectors:*

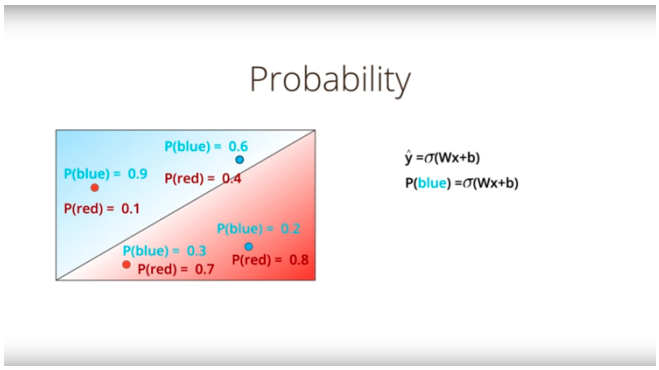


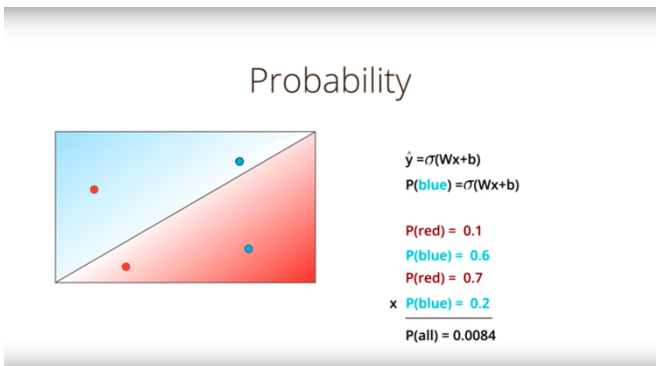
Maximum likelihood

How to understand whether a model is good or bad? Compare it to the reality! For example, suppose you build a model to predict weather, if the model says that there is 90% chance of rain and 10% chance of sun, but in reality, you see absolutely cloudless sky, then the model is bad.

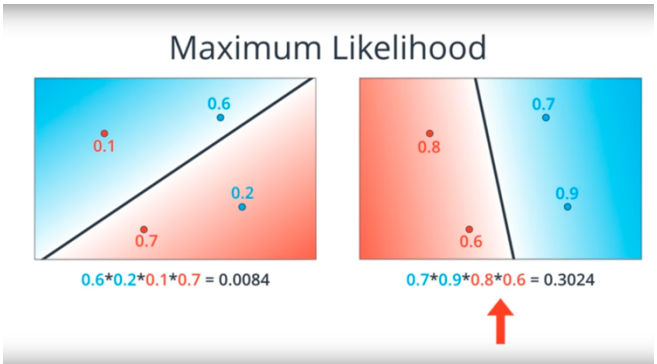
Definition. *Likelihood is a function that describes the plausibility of a model parameter value, given specific observed data.*

Definition. *Maximum Likelihood is a state when the model has parameters which maximize the likelihood function.*



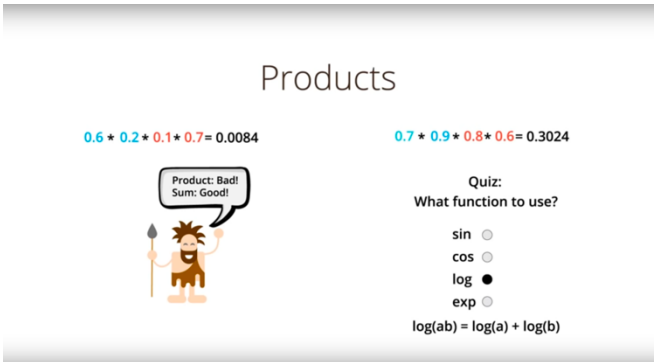


In order to compute the likelihood, we need to take the product of probabilities of every single data point:

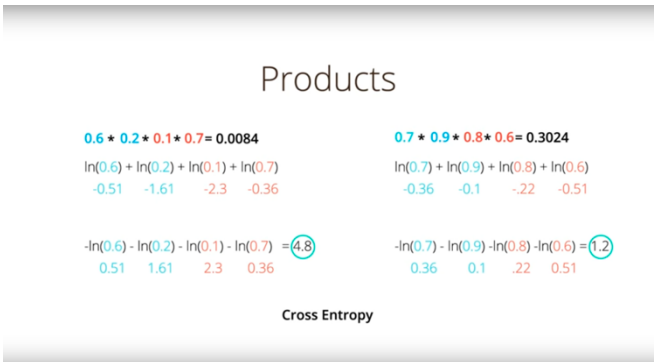


Cross-entropy

Definition. *Cross-entropy is an elaboration of maximum likelihood principle. The main motivation is to turn products into sums with the* $\log$ *function. Deep Learning usually involves a lot of manipulations with the numbers in* $[0,1]$ *interval, but when we multiply a lot of small numbers together the resulting product can be vanishingly small so it is better to work with sums instead of products.*

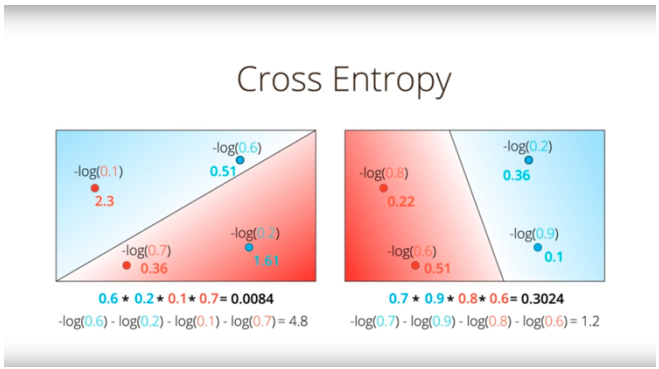


Log function allows to turn products into sums. However, $\log$ *of a number smaller than* 1 *is negative, so we should negate the whole expression to get some meaningful positive estimation of how well the model is doing:*

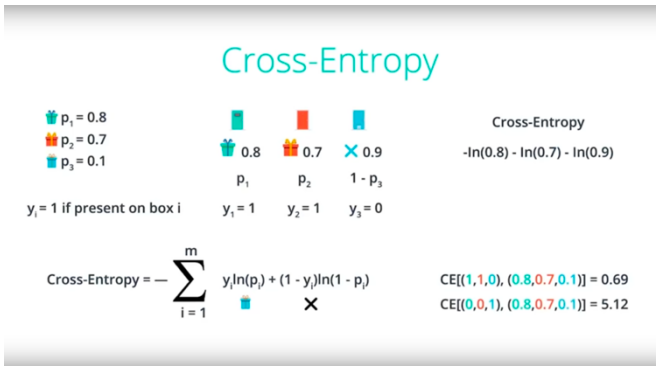


Objective. *Minimize the cross-entropy and maximize the likelihood*

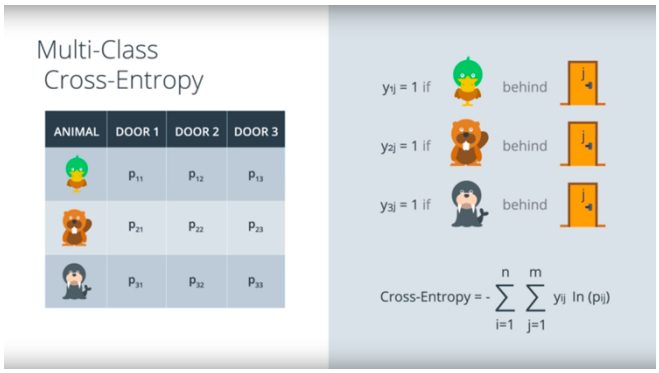
Cross-entropy favors the same (as maximum likelihood estimation) model:



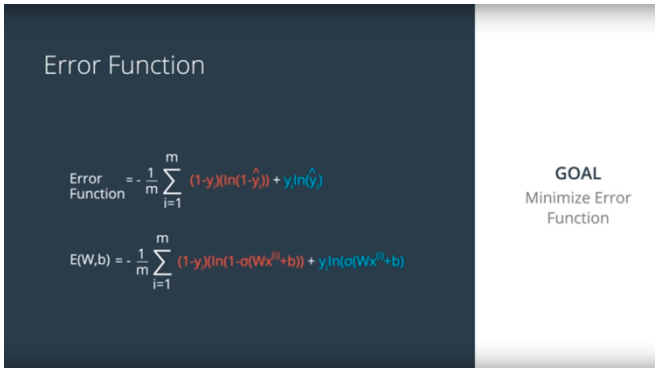
Full formula for cross-entropy (m *– number of examples):*



Generalized formula for cross-entropy (m *– number of classes):*

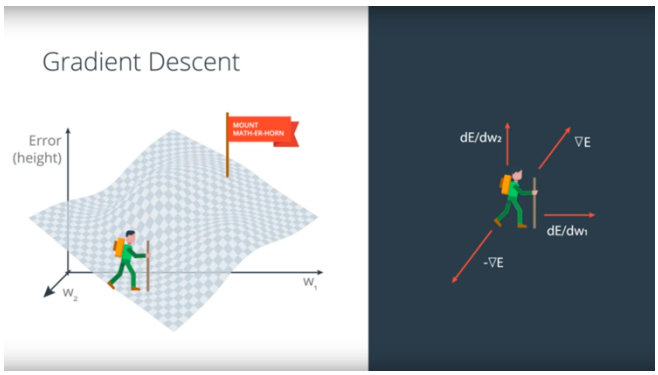


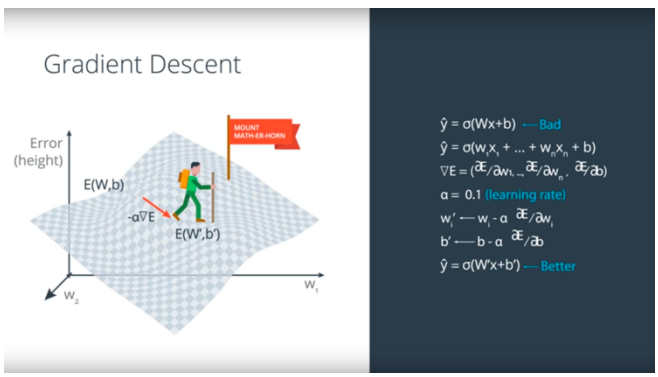
Cross-entropy will serve as the error function in logistic regression:



Gradient descent

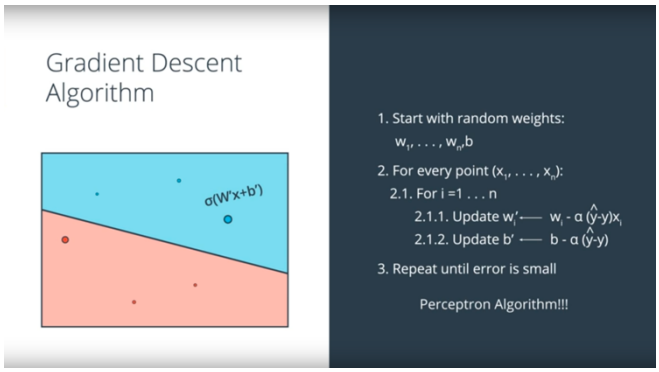
***Definition**. Gradient descend is an optimization algorithm for finding the minimum (possibly local) of a function. It is an iterative process that involves takings steps in the direction opposite of the gradient:*



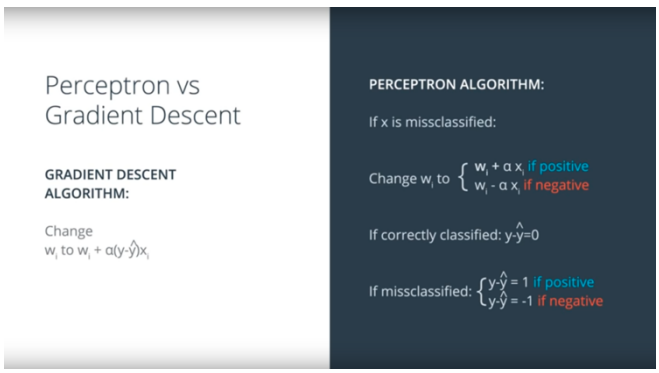


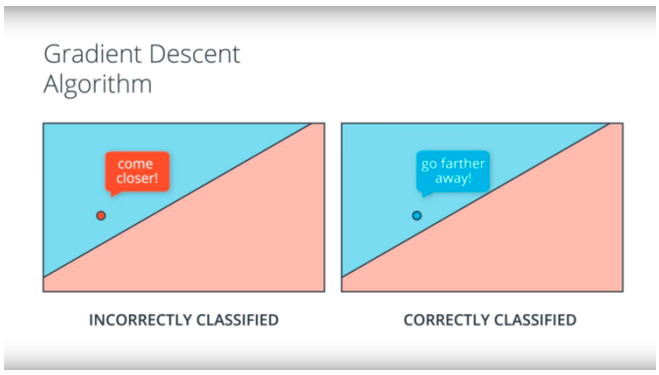
Logistic regression

***Definition**. Logistic regression is basically a linear regression combined with logistic (sigmoid) function. Gradient descent is used to optimize the parameters.*



*The learning process is similar to perceptron trick. However, gradient descent updates the weights **not only** when a point is misclassified, but also when the prediction for that point is correct, making that prediction more certain:*

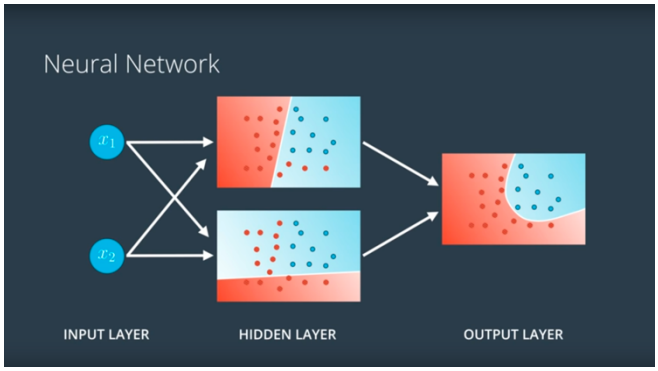


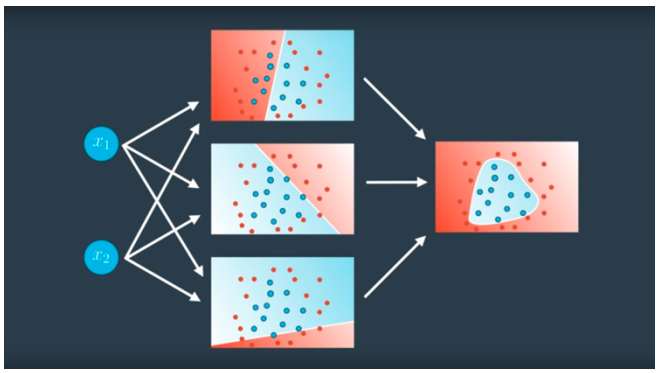


Neural networks

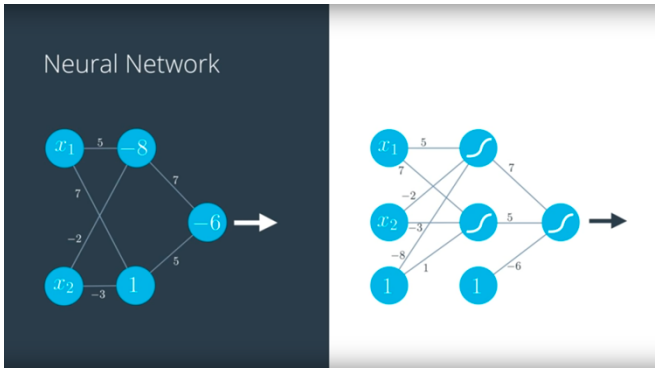
A lot of problems require nonlinear decision boundary:

For nonlinearity we can merge two or more neurons together:

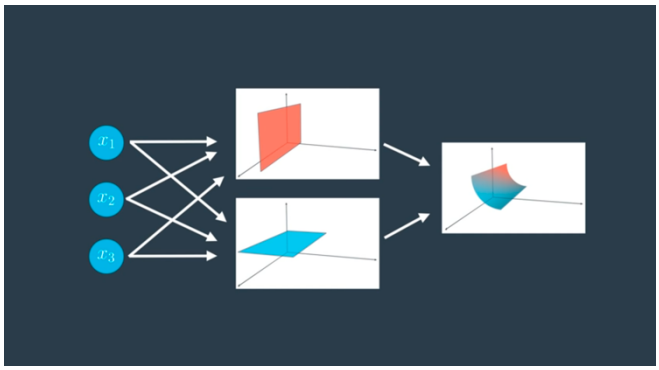




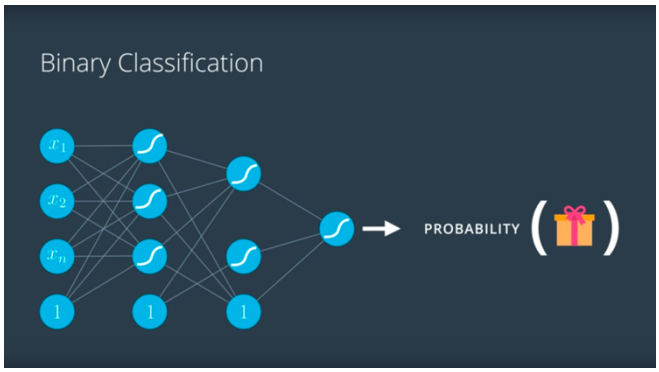
***Reminder**: there are two ways to represent the bias unit. In the case of neural network, each layer will have a bias unit.*



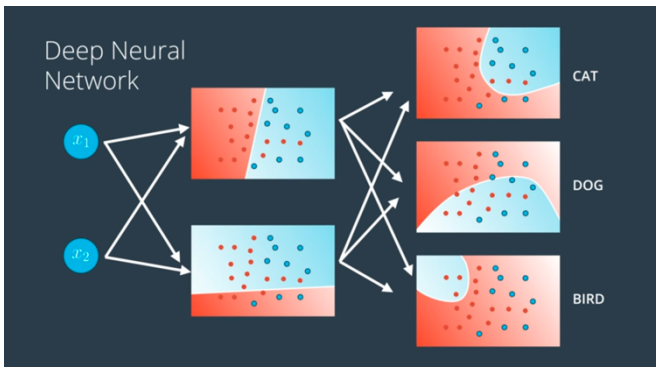
3-d example of nonlinear decision boundary:

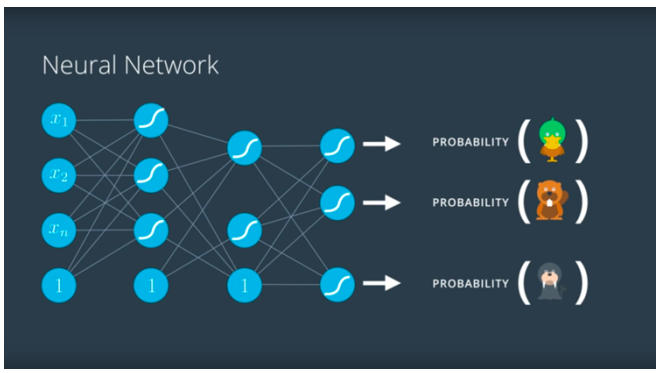


Binary classification can be performed with just one output neuron:



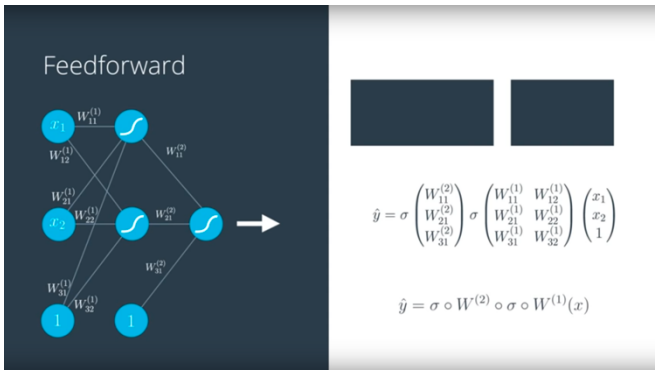
However, for multiclass classification we will need n output neurons, where n is the number of classes:

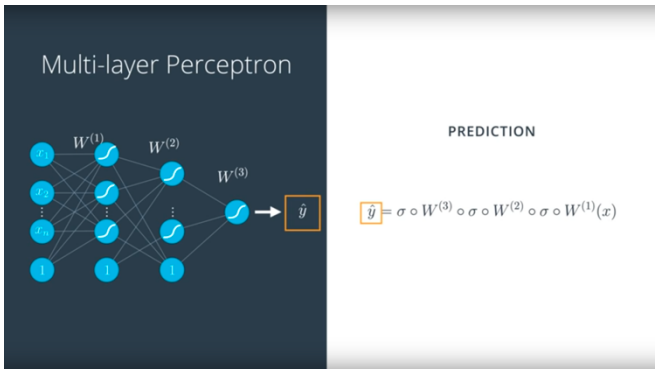


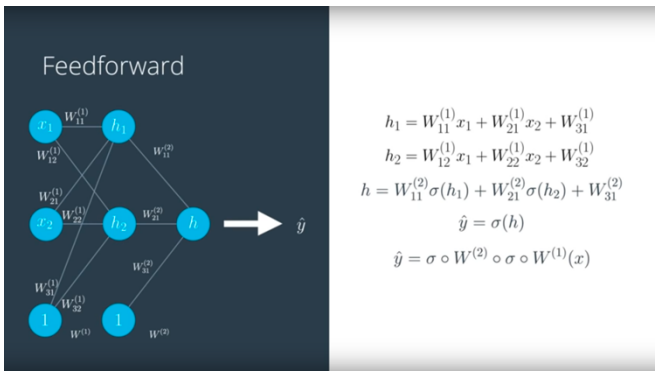


Feedforward

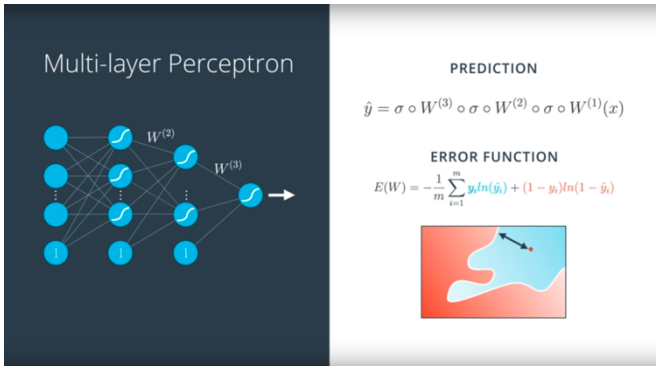
*During the **feedforwarding** step, we apply a sequence of linear models and sigmoid functions to the input and propagate it through all neurons to get the final predictions. The process is similar to the process employed in the simple (one-neuron) perceptron, but instead of just one transformation, the original input undergoes a series of transformations through numerous neurons. Output of one neuron becomes input to another neuron.*





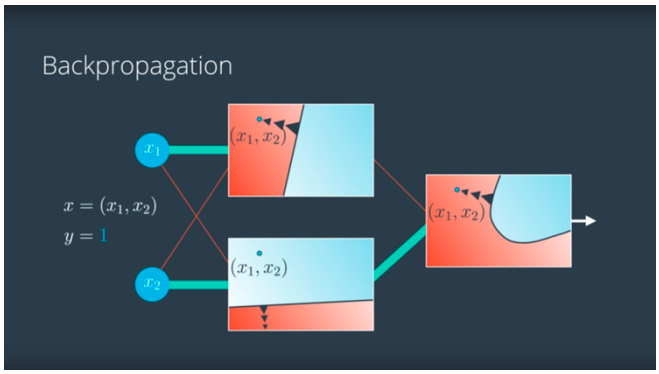


The error function remains the same in the essence, but the expression for the prediction $\hat{y}$ is more complicated:

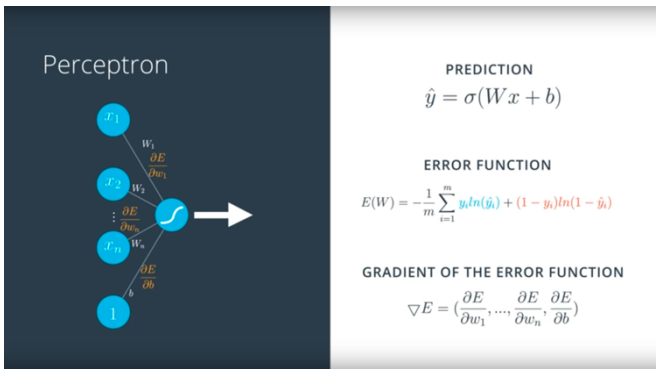


Backpropagation

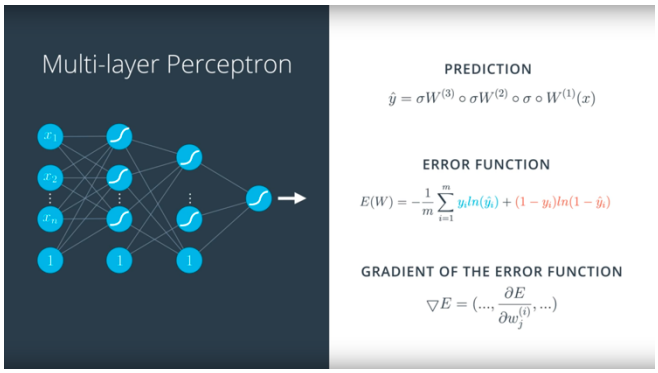
*Feedforwarding is basically a composition of a bunch of functions. **Backpropagation** involves taking derivatives of each component of that composition in order to know how each parameter influence the error function. This allows us to adjust each weight accordingly.*



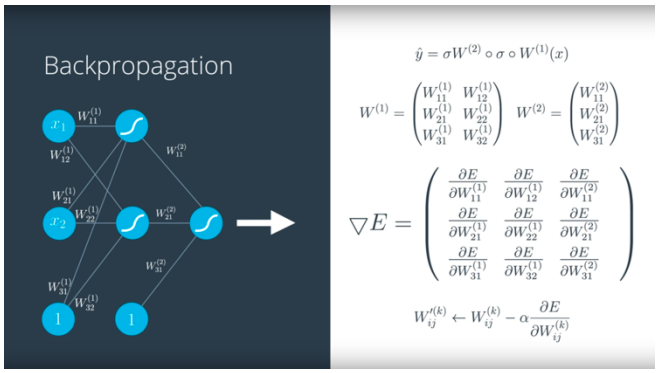
In the case of one-neuron perceptron the process is straightforward:

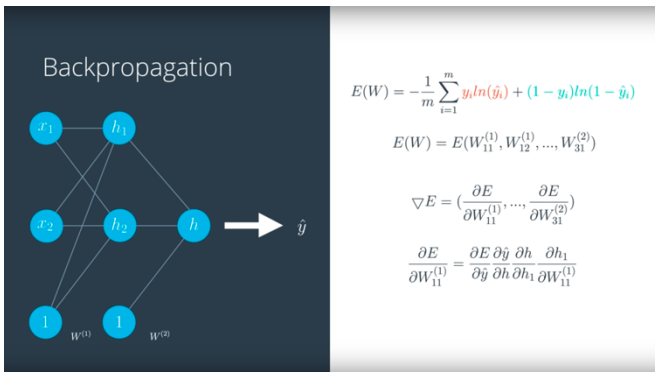


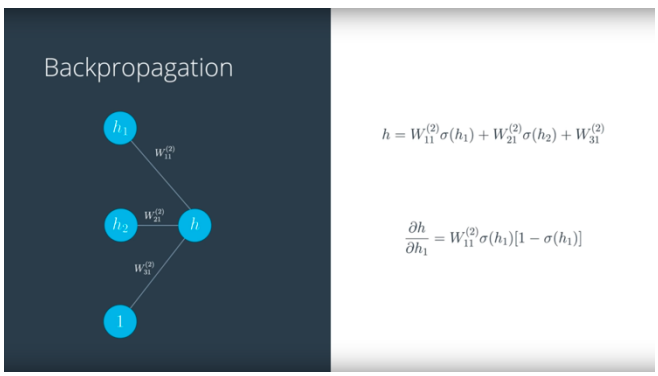
Multilayer networks require more effort:



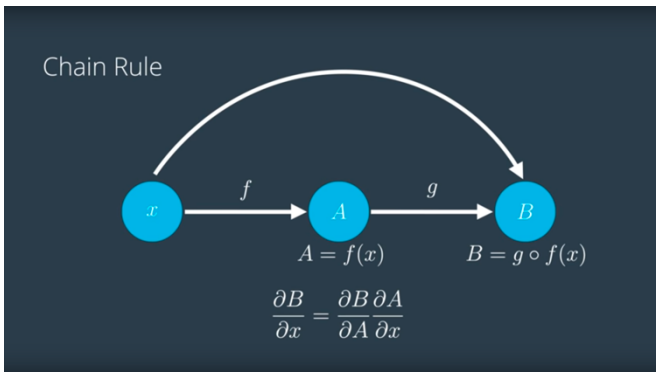
∇E – is a matrix that contain all derivatives (one derivative per every single weight). Each weight is updated by taking the opposite direction of its derivative.







***Reminder**. Chain rules allows to take derivatives of complex functions:*



***Reminder**. The derivative of the sigmoid function:*

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Testing

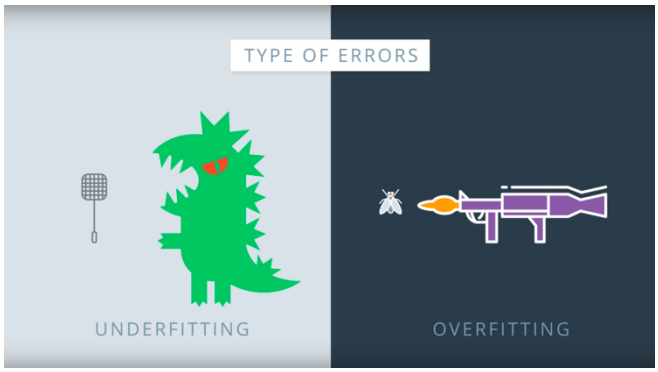
It is important to be able to test different models to have independent evaluation of their performance:

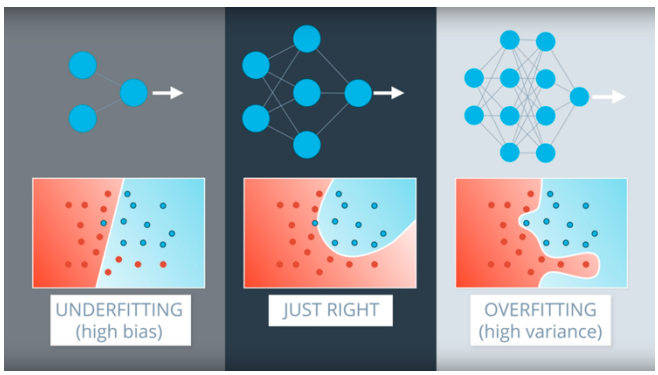
In order to do the testing, we should split the data into training and testing sets. The model should not see the testing set during the training:

Underfitting and overfitting

***Definition**. Underfitting is trying to solve a problem with oversimplified solution.*

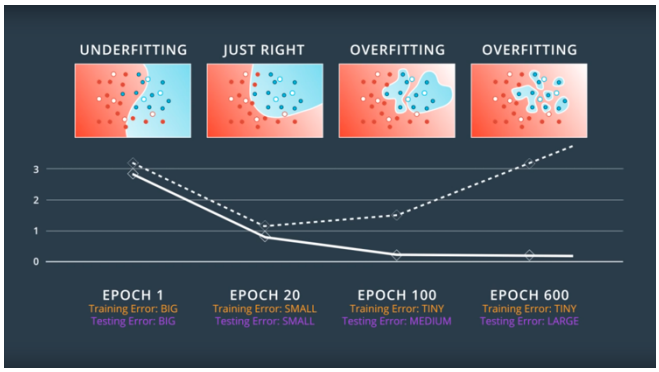
***Definition**. Overfitting is trying to solve a problem with overcomplicated solution.*



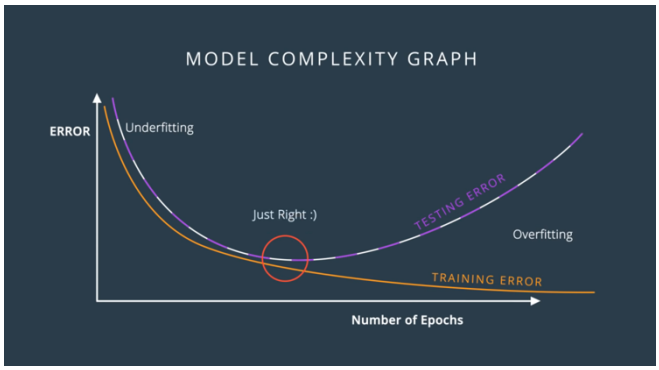


Early stopping

How to understand when the model is just right (it neither underfits nor overfits)? You can plot the train error versus test error, the model is good when both of them are small:

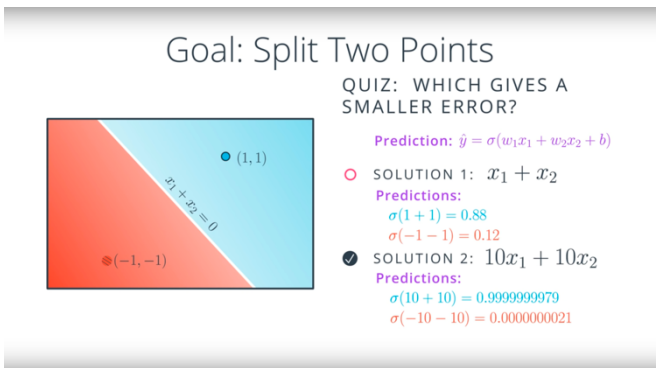


***Definition**. Early stopping is stopping the training process when test error starts to go up.*

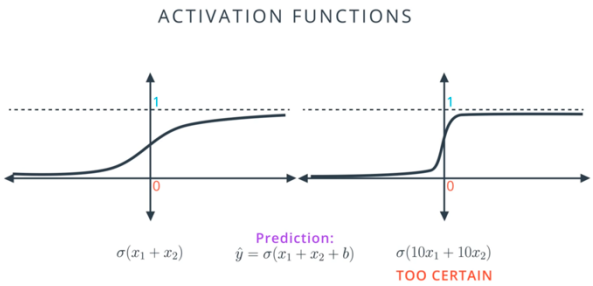


Regularization

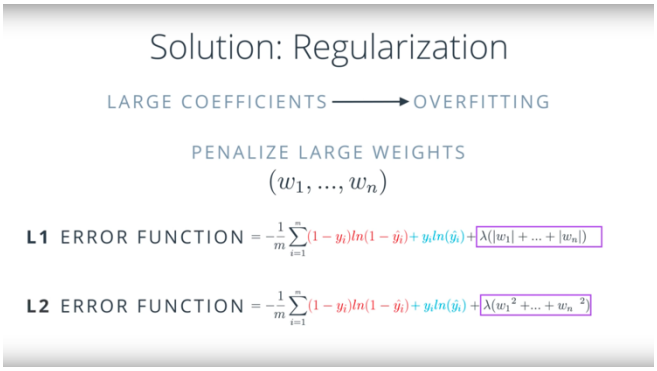
One way to cause overfitting is by allowing a model to have arbitrary large weights:



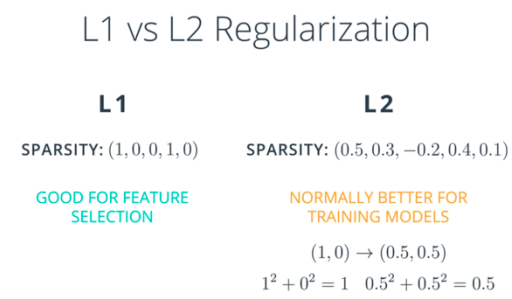
The model becomes too certain as the result of this:



Definition. *Regularization means punishing large weights by adding special regularization term to the error function. This regularization term consists of magnitude of weights multiplied by hyperparameter λ. λ – is a number in the* $[0,1]$ *interval and it tells how much we should punish large weights.*



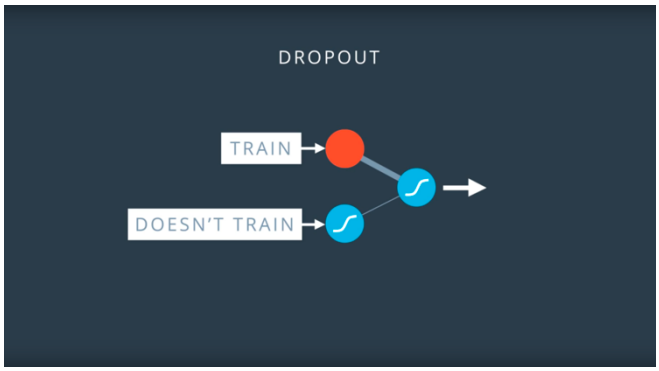
There are two ways to perform regularization: L1 and L2.



Dropout

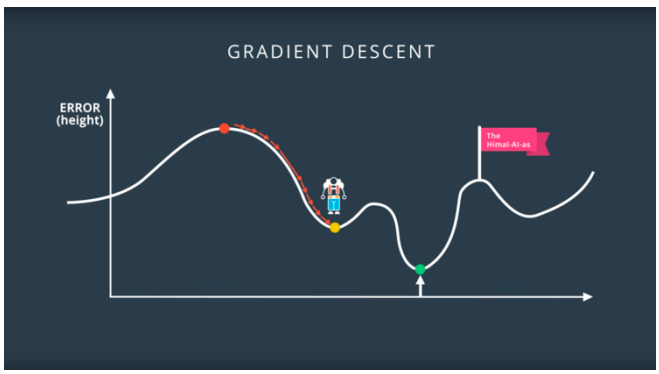
Another way to prevent overfitting is to use dropout.

Definition. *Dropout is a technique that involves freezing a neuron temporarily so it will not participate in the training in order for other neurons to have more 'responsibility'.*



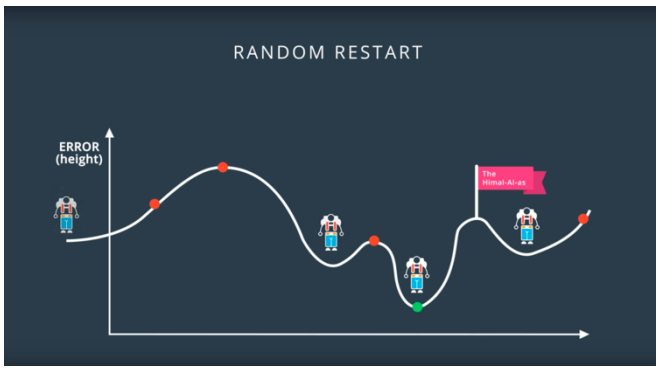
Local minimum

Sometimes the model can be stuck in the suboptimal solution. There are a couple of ways to overcome this problem: random restart and momentum are one of them.



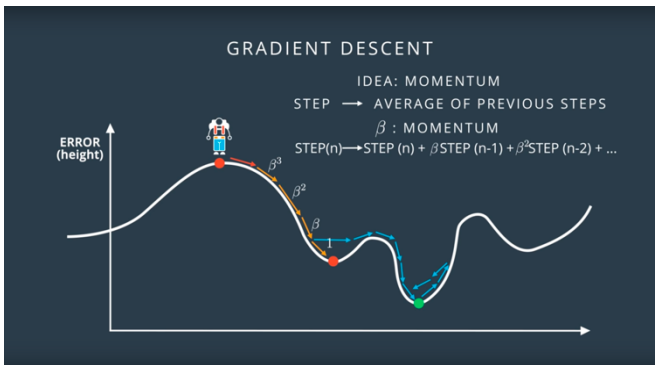
Random restart

Random restart is a restart of the training process with differently initialized weights.



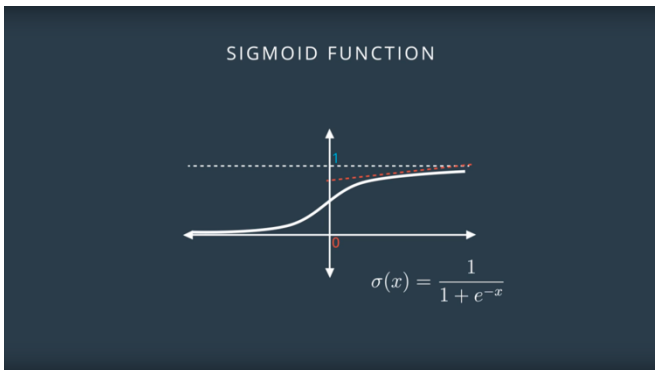
Momentum

Momentum technique allows to jump through the local minimum:

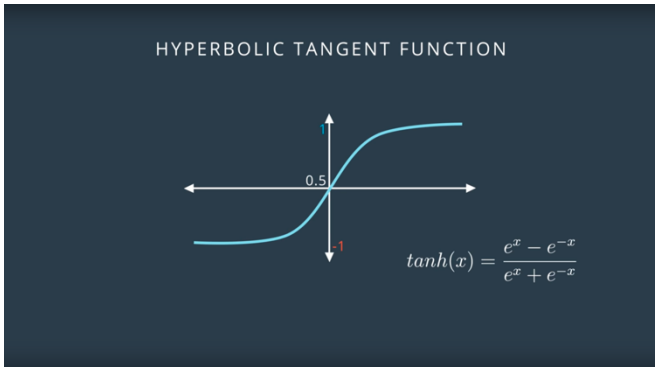


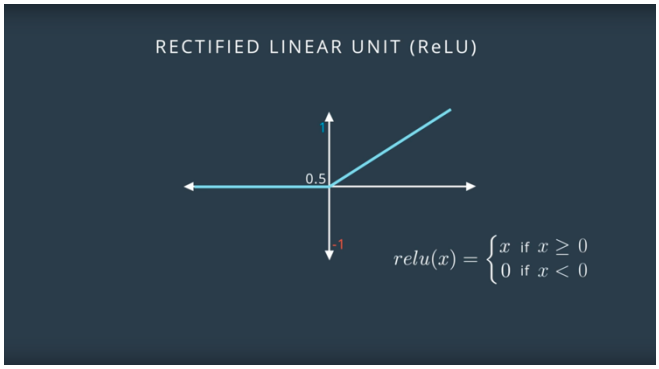
Vanishing gradient and activation functions

The backpropagation step involves taking a lot of products of the numbers below zero, so sometimes gradients become vanishingly small as the slope of sigmoid function if quite small when the function approaches 1 *or* 0*. So, there are some other activation functions that can help to diminish the problem.*



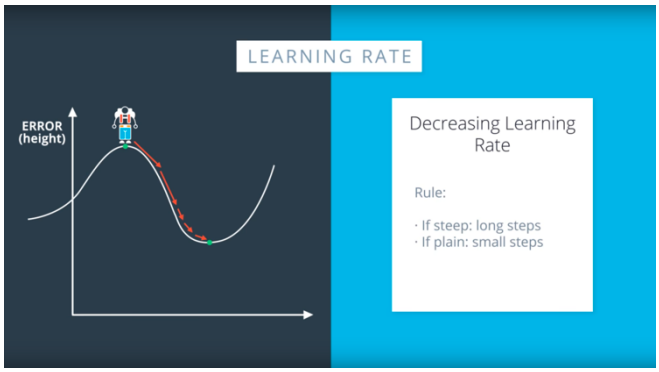
Correction (from the Udacity)*. For the plots of* $tahn(x)$ *and* $relu(x)$ *the origin should be labeled with a value* $y = 0$*, not* 0.5*.*





Learning rate decay

Learning rate decay is a process of decreasing the learning rate over the course of training. This sometimes helps to prevent jumping over the minimum.



The rule of thumb: when your network does not train well, try to decrease the learning rate.