Lesson 3: Introduction to Neural Networks

Notes by: jvinniec

This lesson teaches the concepts behind how neural networks operate and how they are trained using data.

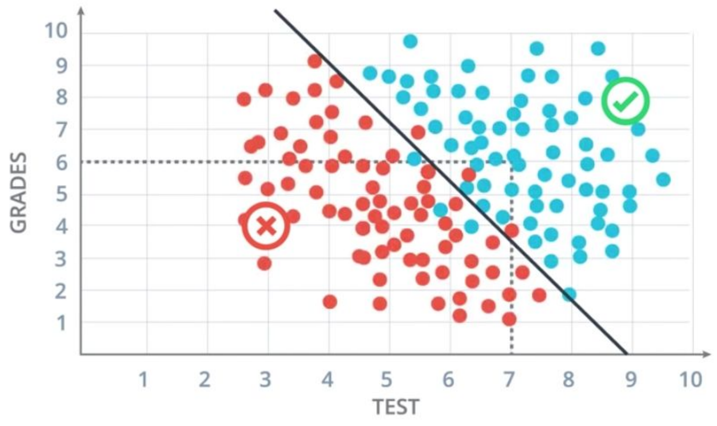
Classification Problems

In a classification problem we are typically presented with some sample of data that we want to classify as belonging to a specific group.

Example: we have a list of students' test scores and grades and we want to use that information to determine which of those students will be admitted to a university.

In order to make a prediction we would start with a sample of student's grades and test scores for whom we know whether or not they have been admitted. Using this information we could then predict the rest of the students admittance status.

The following image demonstrates this by showing the grades (y-axis), test scores (x-axis), and admittance status (color) of a sample of students. A clear separation between the accepted and rejected students is visible (solid black line) which we can use to predict whether a student is accepted or rejected.



Open questions from the above example:

- How do we determine where to put the line?
- Is a line the best way to separate the two groups? Can we determine a more complex separation between the groups?
- How to ensure that our separation actually generalizes to the rest of the students well?

Linear Boundaries

The equation for a line can be written as

$$x_2 = m\,x_1 + b$$

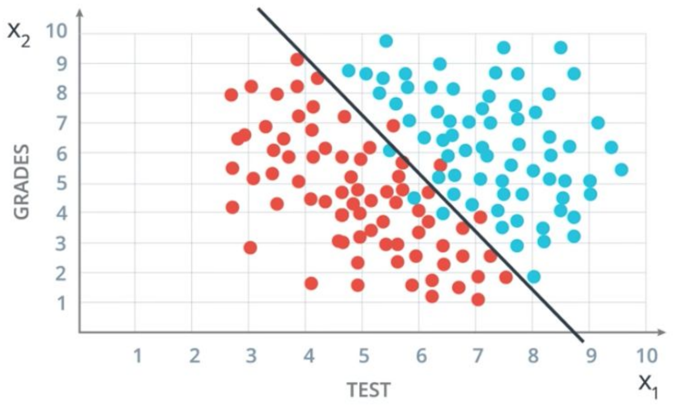
where *m* is the slope of the line and *b* is the point where the line crosses the y-axis (also known as the intercept). The line in the previous example is written as:

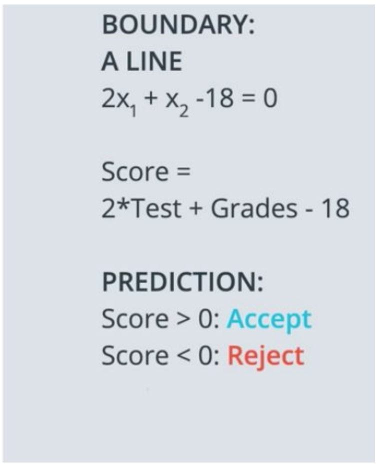
$$x_2 = \ -2\,x_1 + 18$$

or:

$$x_2 + 2\,x_1 - 18 = 0$$

where x_1 is the 'test score' and x_2 is their 'grades'.





BOUNDARY:
A LINE
$2x_1 + x_2 - 18 = 0$

Score =
2*Test + Grades - 18

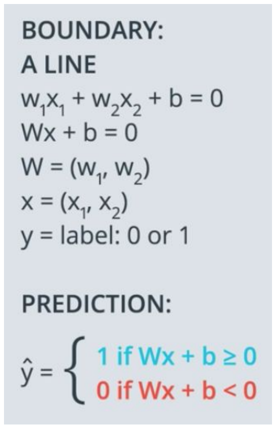
PREDICTION:
Score > 0: Accept
Score < 0: Reject

Now we know that if a student is above this line we predict that they are accepted and if they fall below the line they are rejected:

$$x_2 + 2\,x_1 - 18 > 0 \ \rightarrow accepted$$
$$x_2 + 2\,x_1 - 18 < 0 \rightarrow rejected$$

Conventions in this course:

- x = (x_1, x_2, …) will represent the values for the identifying data (in the above example this is 'test scores' and 'grades')
- y = 0 or 1 will represent the labels on our data, or the category to which a given data point belongs (this is the acceptance status in the above example)
- we add a **'hat'** to y to indicate it is our *predicted* label)
- W = (w1, w2, …) will represent the weights that we apply to our data
- b will represent biases that we apply to improve the prediction



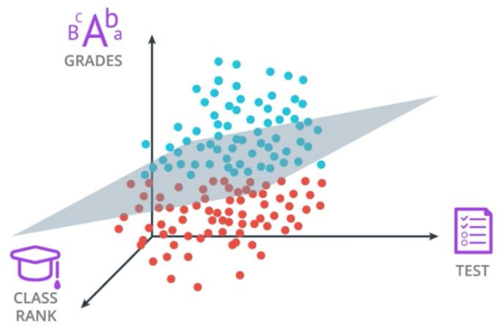
BOUNDARY:
A LINE
$w_1x_1 + w_2x_2 + b = 0$
$Wx + b = 0$
$W = (w_1, w_2)$
$x = (x_1, x_2)$
y = label: 0 or 1

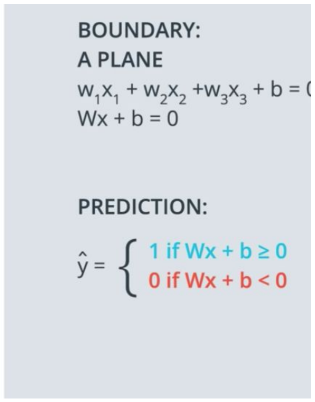
PREDICTION:

$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

Higher Dimensions

We can always add more dimensions to the above example (like adding class rank, or additional test scores). In the case of 3 variables, our boundary *line* becomes a boundary *plane*:




BOUNDARY:
A PLANE
$w_1x_1 + w_2x_2 + w_3x_3 + b = ($
$Wx + b = 0$

PREDICTION:

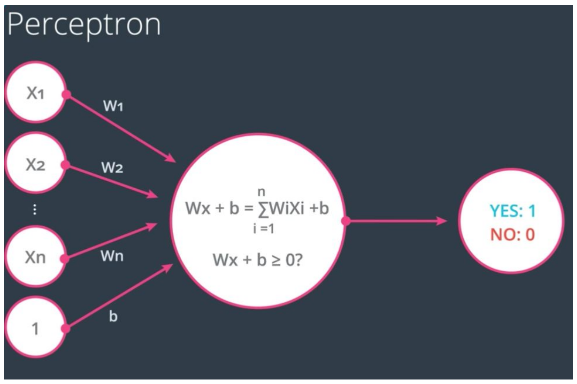
$$\hat{y} = \begin{cases} 1 \text{ if } Wx + b \geq 0 \\ 0 \text{ if } Wx + b < 0 \end{cases}$$

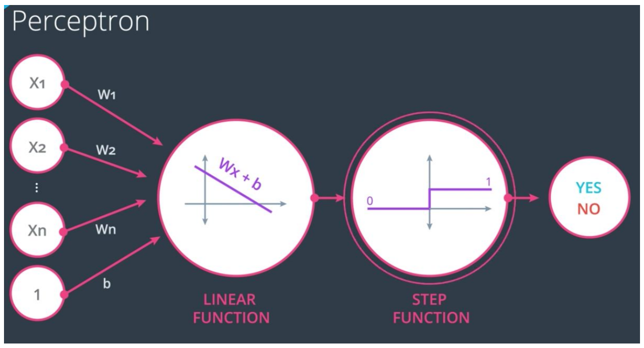
From here it's easy to see how the formula for the boundary grows with additional variables used in our classification problem:

$$W_1x_1 + W_2x_2 + ... + W_nx_n + b = 0$$

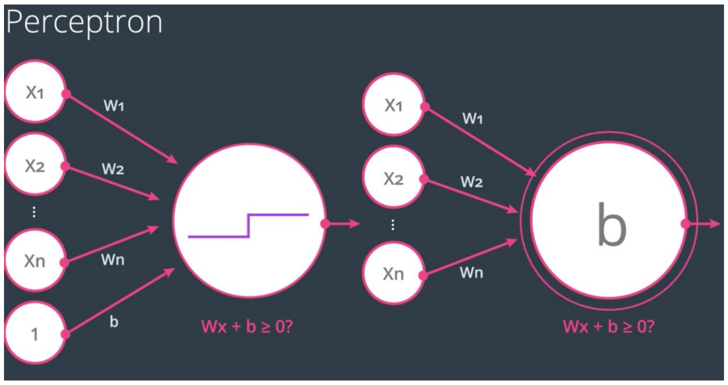
Perceptrons

Perceptrons are algorithms used in supervised learning in **binary classifiers** (classifiers with two possible outputs), such as the classification problem we've been using above.

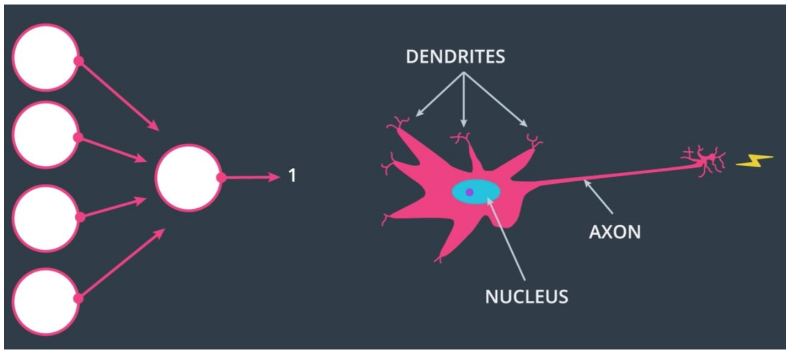




There are two ways to visually represent perceptrons:



Perceptrons are typically described as ‘neural networks’ because they look similar to how neurons in the brain look and work:



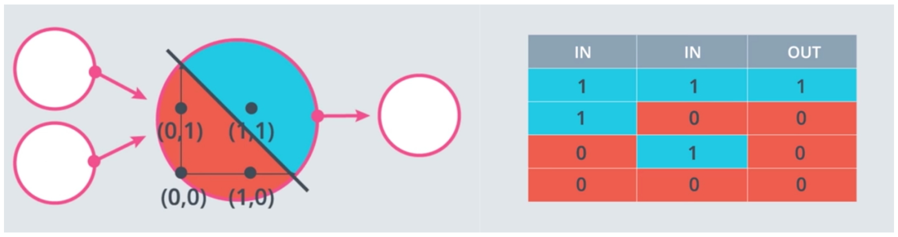
In the future we will use the *outputs* from one perceptron as *inputs* to another one.

Perceptrons as Logical Operators

Perceptrons can also serve as logical operators (think ‘AND’, ‘OR’, and ‘NOT’ operators in a programming expression).

AND: ‘true’ if both inputs are 1, else ‘false’

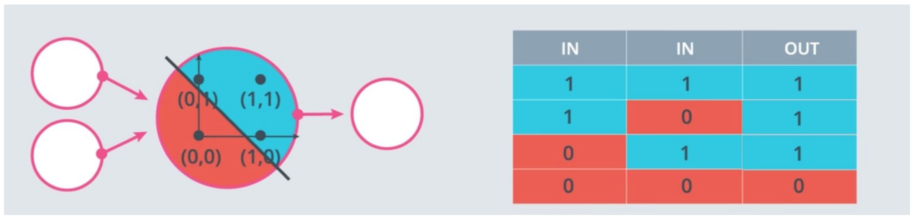
The boundary: $1x_1 + 1x_2 - 1.5 \ = 0$



IN	IN	OUT
1	1	1
1	0	0
0	1	0
0	0	0

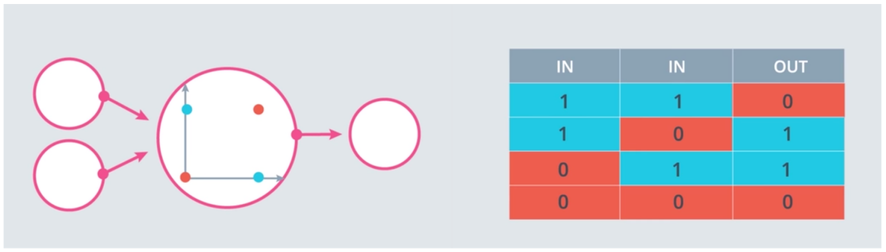
OR: ‘true’ if at least one of the inputs is 1, else ‘false’

The boundary: $1x_1 + 1x_2 - 0.5 \ = 0$



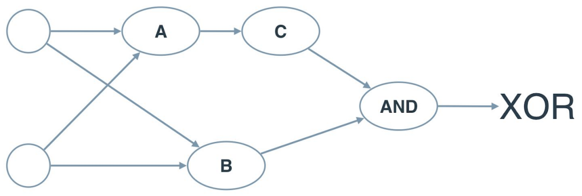
IN	IN	OUT
1	1	1
1	0	1
0	1	1
0	0	0

NOT: ‘true’ if input is 0, else ‘false’ (reverses the input)
XOR: ‘true’ if only one of the inputs is 1, else ‘false’



IN	IN	OUT
1	1	0
1	0	1
0	1	1
0	0	0

The **XOR** operator cannot be constructed using a single-layer perceptron like the AND, OR, and NOT operators. Instead, we need to use a multi-layer perceptron!



Question: Given that each of A,B and C are one the operators we’ve learned above (AND, OR, NOT) which of these are they?

First, we can assume that ‘C’ is the NOT operator, since it only takes a single input. So, given this, let’s build a few tables:

If A=OR, B=AND

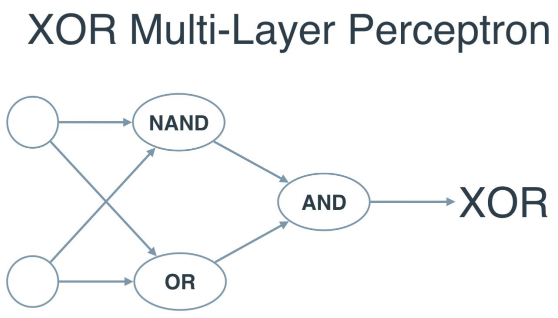
X1	X2	A (OR) out	B (AND) out	C (NOT) out	OUT
1	1	1	1	0	0
1	0	1	0	0	0
0	1	1	0	0	0
0	0	0	0	1	0

If A=AND, B=OR

X1	X2	A (AND) out	B (OR) out	C (NOT) out	OUT
1	1	1	1	0	0
1	0	0	1	1	1

0	1	0	1	1	1
0	0	0	0	1	0

So, we conclude that ‘A=AND’ and ‘B=OR’.

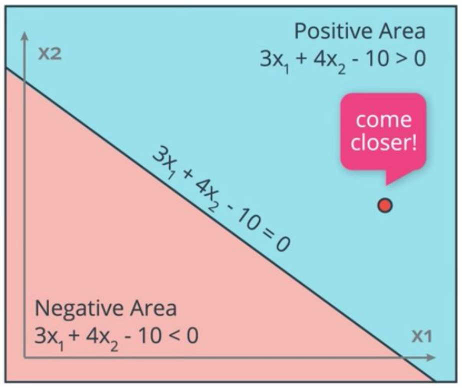


Perceptron Trick

Above, we were coding the perceptrons by hand, but how do we make the computer do it for us? The trick is that the algorithm uses the following procedure:

1. Start with random values for the weights.
2. Evaluate all the points to determine where we have incorrectly classified points.
3. Use misclassified points to adjust the weights & biases:
    a. If prediction = 0
        i. Adjust the weight in the following manner:
        for i = 1...n, $w_i=w_i+a*x_i$ where a = learning rate
        ii. Adjust the bias to b+a
    b. If prediction = 1
        i. Adjust the weight in the following manner:
        for i = 1...n, $w_i=w_i - a*x_i$ where a = learning rate
        ii. Adjust the bias to b - a
4. Take the updated line and repeat from step 2
5. Iterate on steps 2-4 until we are satisfied with the results.

Negative area point misclassified:

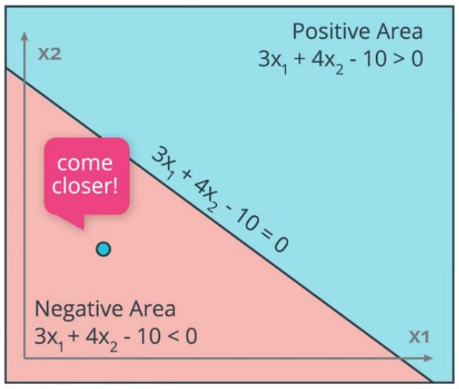


LINE: $3x_1 + 4x_2 - 10 = 0$
POINT: (4,5)
LEARNING RATE: 0.1

$$\begin{array}{cccc} & 3 & 4 & -10 \\ - & 0.4 & 0.5 & 0.1 \\ \hline & 2.6 & 3.5 & -10.1 \end{array}$$

NEW LINE
$2.6x_1 + 3.5x_2 - 10.1 = 0$

Positive area point misclassified:



LINE: $3x_1 + 4x_2 - 10 = 0$
POINT: (1,1)
LEARNING RATE: 0.1

$$\begin{array}{cccc} & 3 & 4 & -10 \\ + & 0.1 & 0.1 & 0.1 \\ \hline & 3.1 & 4.1 & -9.9 \end{array}$$

NEW LINE
$3.1x_1 + 4.1x_2 - 9.9 = 0$

Question: For the second example, where the line is described by $3x_1 + 4x_2 - 10b = 0$ and we use a learning rate of 0.1, how many times would you have to apply the perceptron trick to move the line to a position where the blue point (1,1) is correctly classified?

Solution:
Since the blue point is at (x_1=1, x_2=1), then each time we adjust the line we increase it by

$$(learning\ rate)\ * (x_1^2 + x_2^2 + 1)\ =\ 0.1 \times (1 + 1 + 1)\ =\ 0.3$$

So, if the line formula initially evaluates to:

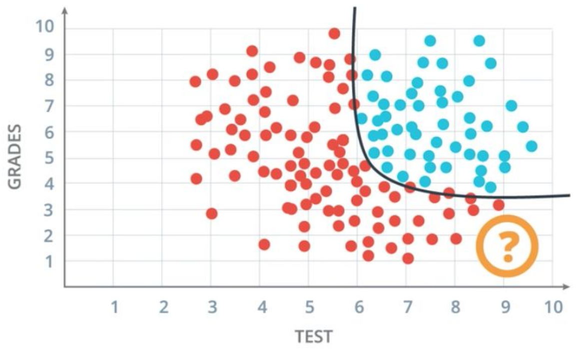
$3x_1 + 4x_2 - 10b$ = 3+4-10 = -3

we need to adjust it enough times to increase that result to 0:

$$-3 + n * 0.3 = 0 \quad \longrightarrow \quad n = 10$$

So we need to take **10 steps** to get the final answer.

What do we do when the data we want to classify has a non-linear boundary? For example, the below example is much better classified with a curved line than with a straight line.



In this case, we'll need to modify our perceptron algorithm.

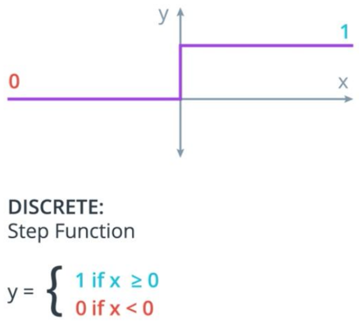
Error Function

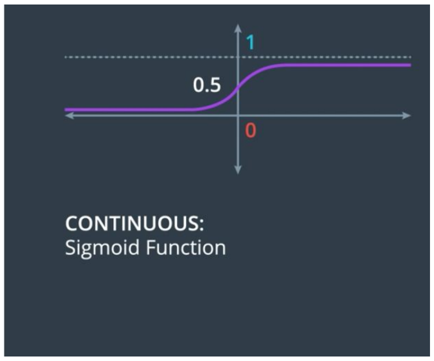
An error function is simply a measure of how far away we are from our goal. It is important for our error function to be a **continuous**, **differentiable** function.

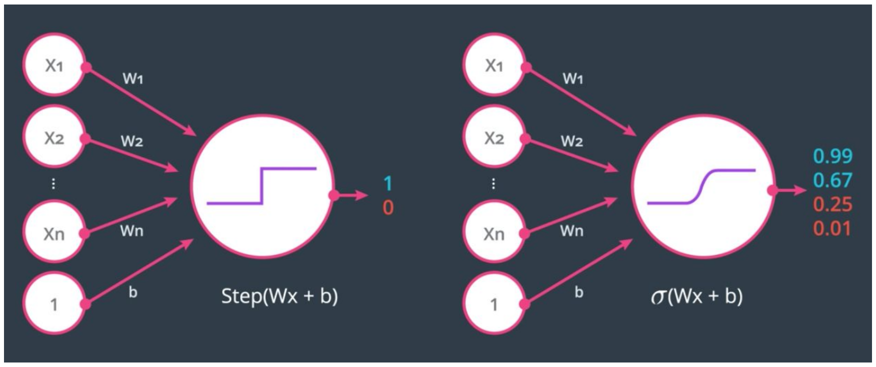
Discrete vs. Continuous

Until now we've been dealing with **discrete** predictions (i.e. 0 or 1). On the other hand, a **continuous** prediction can be thought of as a likelihood (e.g. 68.7% likely to belong to a certain group). For discrete predictions we used the step-function, but for continuous predictions we instead use the sigmoid function:

- Step function => discrete prediction
- Sigmoid function => continuous prediction







We see in the above, the sigmoid function translates the score into a probability for a given category. It is given by the formula:

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

Multiclass Classification & Softmax

So far we have only considered that our problem has 2 outputs. But what do we do when we have multiple possible classes (e.g. classifying an image of an animal as a duck, beaver, or walrus)?

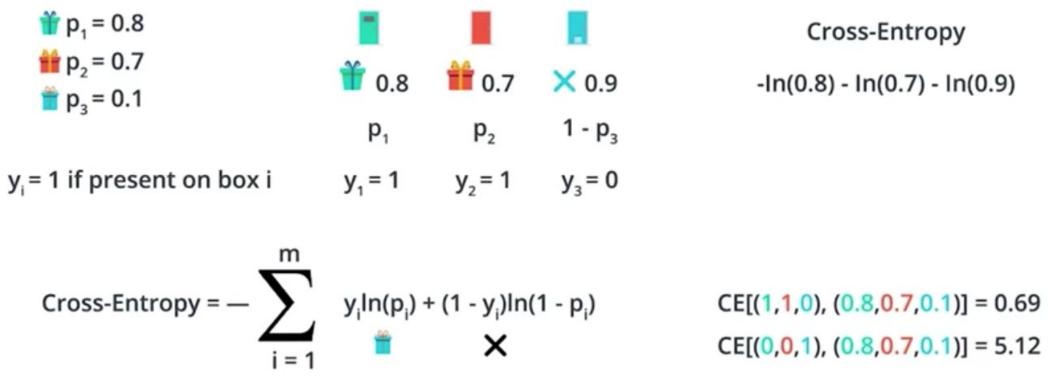
Softmax function: Consider we have a linear function returning the scores for all categories in our classifier. These scores are represented by Z_1, Z_2, …, Z_n. Then the probability of a given class is given by:

$$P(class\ i) = e^{i} / (e^{Z1} + ... + e^{Zn})$$

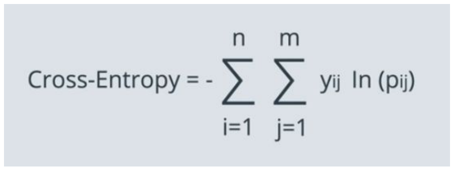
One-Hot Encoding: When we have multiple classes, we give the classification as a vector of 0's (for the incorrect class) and a 1 (for the correct class). So in our above example:

- Duck = [1,0,0]
- Beaver = [0,1,0]
- Walrus = [0,0,1]

Cross-Entropy: Sum of negative-logarithms of the probabilities of each point being classified correctly. Better predictions will result in a lower value for the cross-entropy, thus we try to *minimize* cross-entropy. An example of how to calculate cross-entropy is given below:


$p_1 = 0.8$
$p_2 = 0.7$
$p_3 = 0.1$
$y_i = 1$ if present on box i
0.8
0.7
0.9
p_1
p_2
$1 - p_3$
$y_1 = 1$
$y_2 = 1$
$y_3 = 0$
Cross-Entropy
-ln(0.8) - ln(0.7) - ln(0.9)
$\text{Cross-Entropy} = -\sum_{i=1}^{m} y_i \ln(p_i) + (1 - y_i)\ln(1 - p_i)$
CE[(1,1,0), (0.8,0.7,0.1)] = 0.69
CE[(0,0,1), (0.8,0.7,0.1)] = 5.12

And for multiple classes we have the formula for cross-entropy as (n=number of features, m=number of data points):


$\text{Cross-Entropy} = -\sum_{i=1}^{n} \sum_{j=1}^{m} y_{ij} \ln(p_{ij})$

Question: Are these two formula the same for n=2 (i.e. for only 2 features)?

Solution: Multi-class cross-entropy is given by:

$$CE =- \sum_{i=1}^{n} \sum_{j=1}^{m} y_{ij} ln\left(p_{ij}\right)$$

When n=2 we get:

$$CE =- \sum_{i=1}^{2} \sum_{j=1}^{m} y_{ij} ln\left(p_{ij}\right) =- \sum_{j=1}^{m} y_{j1} ln\left(p_{j1}\right) + y_{j2} ln\left(p_{j2}\right)$$

Note that when we have only two classifications *y_2*=(1-*y_1*) and *p_2*=(1-*p_1*) giving us

$$CE =- \sum_{j=1}^{m} y_{j1} ln\left(p_{j1}\right) + \left(1 - y_{j1}\right) ln\left(1 - p_{j1}\right)$$

Or more simply:

$$CE =- \sum_{j=1}^{m} y_{j} ln\left(p_{j}\right) + \left(1 - y_{j}\right) ln\left(1 - p_{j}\right)$$

Which is exactly our formula for a 2 class cross-entropy

Logistic Regression

The **logistic regression** algorithm functions like this:

- Take your data
- Pick a random model
- Calculate the error
- Minimize the error and obtain a better model

Gradient Descent

We can picture our error function as a multi-dimensional plane. As we adjust the weights (*W_i*) and bias (*b*) we notice that adjusting them in a particular direction will reduce the error function. The magnitude and direction of this change is known as the **gradient**. Since we move in the direction in which the gradient is decreasing, we call this **gradient descent**.

For a mathematical derivation of the gradient descent formula, see Lesson 3.23. The short story is we get a formula for the gradient of:

$$\nabla E =- (y - \widehat{y})\,(x_1, ..., x_n, 1)$$

We can take from this formula 2 things:

- The magnitude of the gradient gets smaller the closer we get to the true value
- The magnitude of the gradient gets larger farther from the true value

The gradient descent algorithm results in the following change to the weights and bias:

$$w_i' = w_i - \alpha[-(y-\hat{y})\,x_1] \;=\; w_i + \alpha(y-\hat{y})\,x_1$$

$$b' \;=\; b + \alpha\,(y-\hat{y})$$

Note that we are taking the average, so by convention:

$$\alpha = (1/m) * (learning\ rate)$$

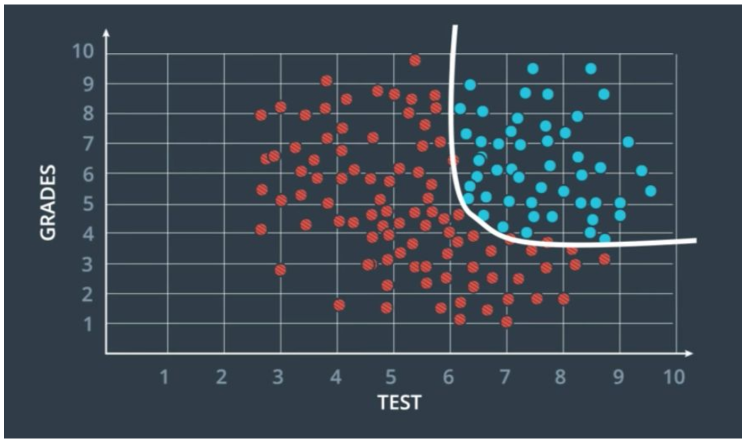
Logistic Regression Algorithm

Now we have all the parts we need to run the logistic regression algorithm:

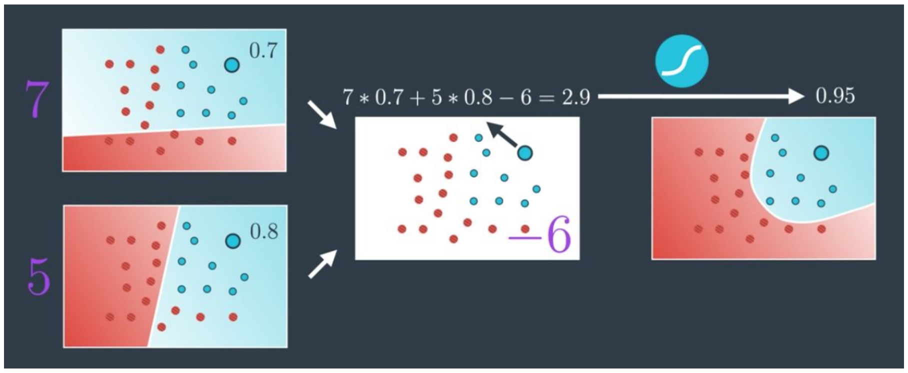
- Take your data
- Pick a random model
    - i.e. initialize your weights and bias to random values. It could be that we chose the initial values close to where we think the best fit is, which could reduce the number of epochs we need to train.
- Calculate the error
    - This is done using the formula we have above:
      $E = \; -y\,log(\hat{y}) \;-\; (1-y)\,log(1-\hat{y})$
- Minimize the error and obtain a better model
    - This is done by adjusting the weights of the model (as described above) through the gradient descent method.

Non-linear Data

This is where we discuss how to generalize the algorithms we've discussed so far so tha they apply to classification boundaries of any shape and size. Recall the problem we discussed above in which we have data that might be best described by a non-linear line:

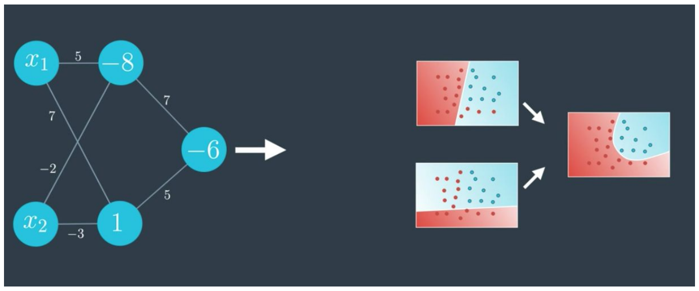


Up to now we've been dealing with single perceptron (or neuron) neural networks. By combining multiple perceptrons we can obtain the behavior that we want!



The image above demonstrates how this works. It demonstrates how we can take two perceptrons that represent models with linear boundaries and treat their outputs as inputs to a third perceptron. We can then apply the sigmoid function to the third perceptron and get our final prediction, **which now has a non-linear boundary!**

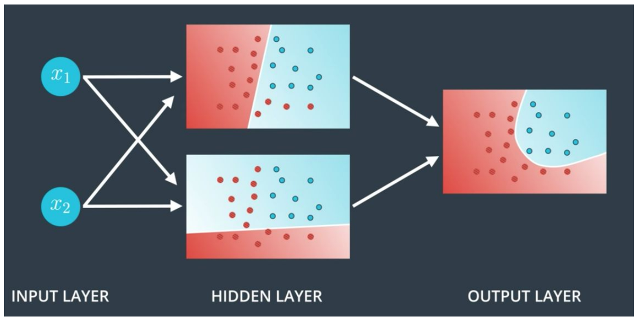
Here's another example demonstrating how the input data is connected to two neurons which in tern are connected to a single neuron that outputs our prediction.



Multiple Layers

Some notation before we continue:

- Input layer: The very first layer that contains the data we want to classify. There is one node for each data point that we want to classify (e.g. for each pixel in an image) plus one extra node for the bias.
- Output layer: The layer that tells us the probability of a single set of inputs (e.g. a single image) belonging to a given classification. There is one node in the output layer for each possible classification.
- Hidden layer(s): These are every layer between the input layer and the output layer. In theory there is no limit to the number of hidden layers or nodes per hidden layer that can be used in a neural network. Each hidden layer also has one node for the bias.



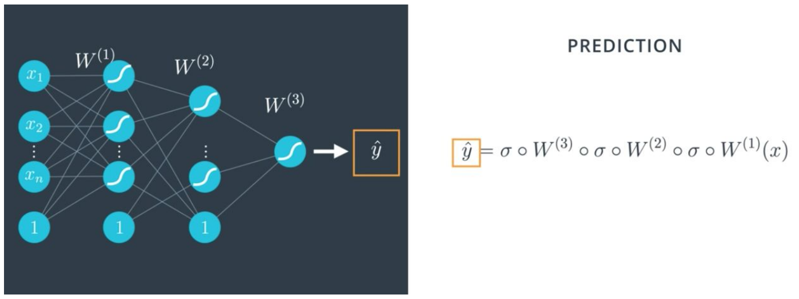
Neural networks can be (and typically are) more complicated than the example given above. For example we can:

- Add more nodes to the input, hidden, and output layers
- Add more hidden layers

Feedforward

Feedforward is the process neural networks use to turn the input into an output. It's the process of taking every input (all the x's) and passing their values through the entire network to obtain the outputs.

In the case of multiple layers, our prediction value now becomes a bit more complicated. For example, consider a neural network with 3 hidden layers and an output layer:


$W^{(1)}$
$W^{(2)}$
$W^{(3)}$
x_1
x_2
x_n
1
1
1
$\hat{y}$
PREDICTION
$\boxed{\hat{y}} = \sigma \circ W^{(3)} \circ \sigma \circ W^{(2)} \circ \sigma \circ W^{(1)}(x)$

The formula for the prediction for a 3 hidden layer network is now:

$$\hat{y} = \sigma\left(W^{(3)} \circ \sigma\left(W^{(2)} \circ \sigma\left(W^{(1)}(x)\right)\right)\right)$$

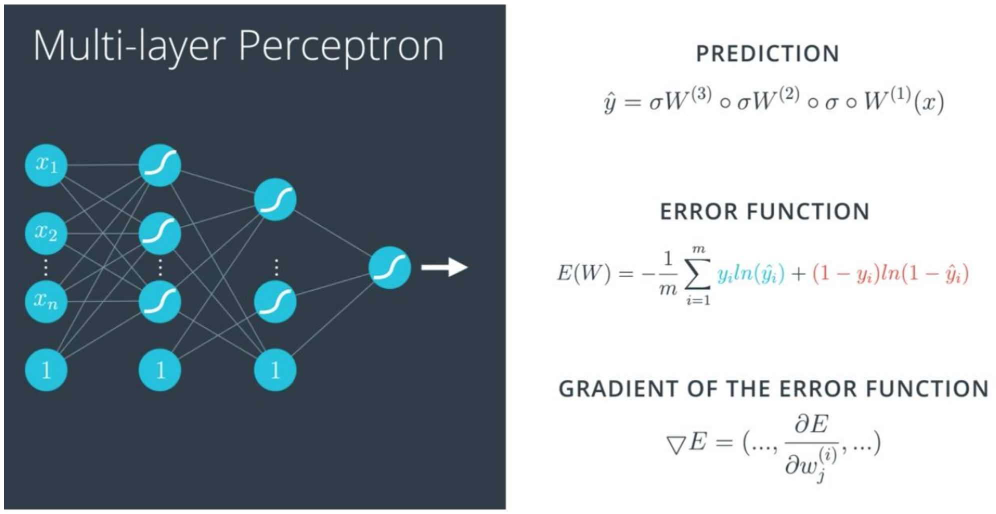
Our prediction function may be more complicated, but our error formula remains unchanged from before:

$$E = -\frac{1}{m}\sum_{i=1}^{m} y_i \, log(\hat{y}_i) + (1 - y_i)\, log(1 - \hat{y}_i)$$

Backpropagation

For training the neural networks, we use **backpropagation**. This process consists of:

- Doing a feedforward operation (compute the probabilities)
- Comparing the output of the model with the desired output (see next step)
- Calculating the error
- Running the feedforward operation backwards (hence the term backpropagation) to spread the error to each of the weights
- Use this to update the weights, and get a better model
- Continue this until we have a model that is good.

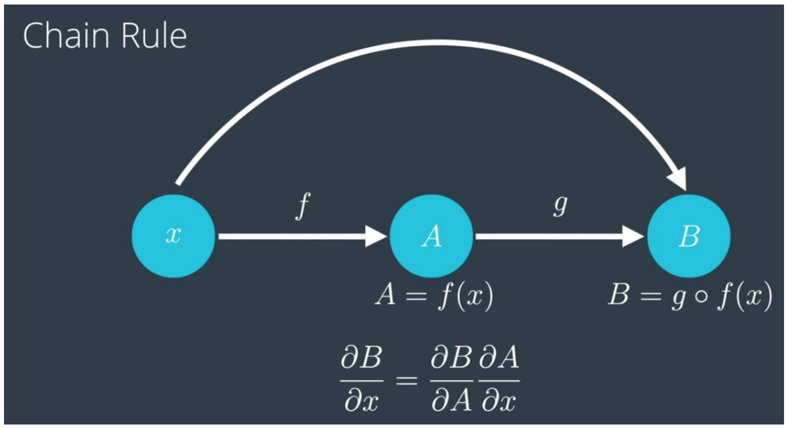

Multi-layer Perceptron
x_1
x_2
x_n
1
PREDICTION
$\hat{y} = \sigma W^{(3)} \circ \sigma W^{(2)} \circ \sigma \circ W^{(1)}(x)$
ERROR FUNCTION
$E(W) = -\frac{1}{m}\sum_{i=1}^{m} y_i ln(\hat{y_i}) + (1-y_i)ln(1-\hat{y}_i)$
GRADIENT OF THE ERROR FUNCTION
$\bigtriangledown E = (..., \frac{\partial E}{\partial w_j^{(i)}}, ...)$

(note there is an error in the above image, the second hidden layer should have inputs going to the sigmoid function, not the bias)

Chain rule

The chain rule is a useful formula for calculating partial derivatives of functions of functions of variables.

Consider the image below:


Chain Rule
x
f
A
g
B
$A = f(x)$
$B = g \circ f(x)$
$\frac{\partial B}{\partial x} = \frac{\partial B}{\partial A}\frac{\partial A}{\partial x}$

Here we have:

$$B = g(f(x))$$

Using the chain rule we can calculate the partial derivative of B with respect to x as:

$$\frac{\delta B}{\delta x} = \frac{\delta g}{\delta f}\frac{\delta f}{\delta x}$$

So we see that the partial derivative of a function of functions can be written as the product of partial derivatives down the chain of those functions.

As an example, we can compute the derivative of the sigmoid function:

$$\sigma(x) = (1 + e^{-x})^{-1} = g(f(h(x)))$$

$$\sigma'(x) \;=\; \frac{\delta g}{\delta f}\frac{\delta f}{\delta h}\frac{\delta h}{\delta x}$$

Where

$$h(x) \;= e^{-x}\,,\;\; \frac{\delta h}{\delta x} \;=\; -\,e^{-x}$$

$$f(h) \;=\; 1 + h\,,\;\; \frac{\delta f}{\delta h} = 1$$

$$g(f) \;= f^{-1}\,,\;\; \frac{\delta g}{\delta f} \;=\; -f^{-2}$$

Which substituting into the above now gives:

$$\sigma'(x) = \left(-f^{-2}\right)\cdot(1)\cdot(-\,e^{-x})$$

Plugging in for *f(h(x))*:

$$\sigma'(x) = \left(-\,(1+e^{-x})^{-2}\right)\cdot(1)\cdot(-\,e^{-x})$$

And simplifying a little:

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

Noting that the first term is just σ(x) and the last two terms are equivalent to 1-σ(x):

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

Training Optimization

This section covers ways in which we can optimize our model and assess its performance.

Testing

Testing is an important part of building our models. It involves taking a separate sample of data (not used in the training) to measure the accuracy of our trained model. This provides insight into how well our model will ultimately generalize to data that it hasn't seen yet.

Types of Errors

Underfitting

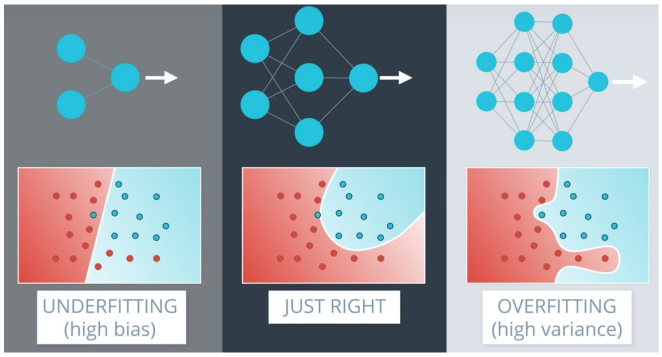
Using a model that is too simplistic for the problem. Also known as 'error due to bias'. Some causes of underfitting:

- Model is too simplistic to accurately model the data
- Model needs additional training

Overfitting

Using a model that is too complex for the problem meaning it doesn't generalize to new data. Also known as 'error due to variance'. Some causes of overfitting:

- Model is overly complex
- Model has trained to long and has memorized the data

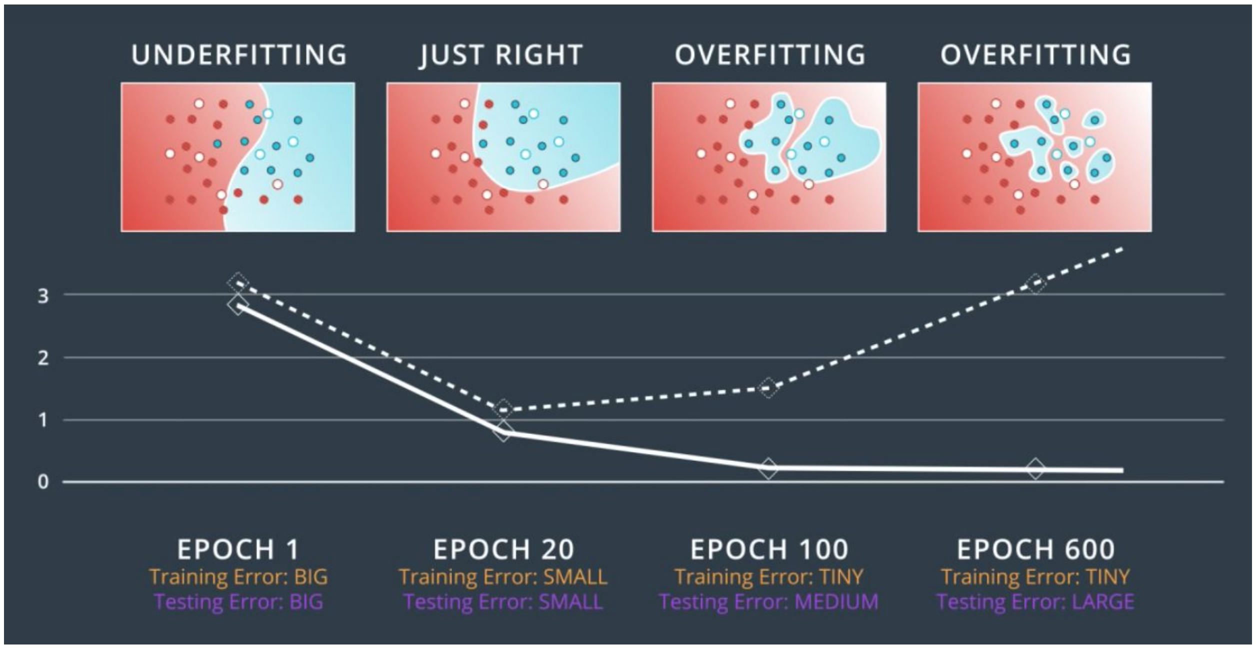


Typically, we error on the side of an overly complex model, while also applying several techniques to reduce the impact of overfitting.

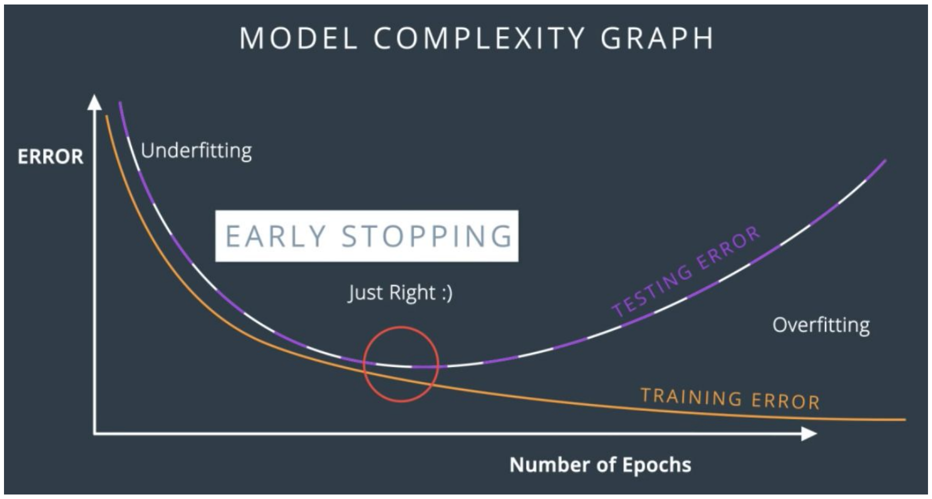
Techniques to prevent overfitting

Early Stopping

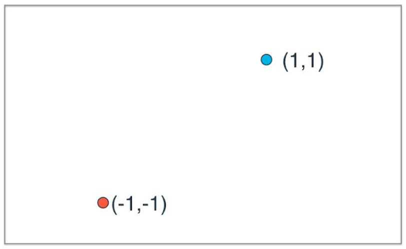
One of the issues (as mentioned above) with fitting complex models is training them for too long. This results in the model learning the training data itself, rather than general trends in the data. If we compare the error for a sample of training data and a separate sample of testing data, we see that over time the training error always goes down. However, the testing error increases when we begin to overfit.



We can visualize this by looking at the following model complexity graph. The point where the testing error reaches a minimum is the point at which we stop training to prevent overfitting. This is known as '**Early Stopping**'.



Question: For the following two points, which solution (i.e. model) gives the smaller error?



Quiz: Which gives a smaller error?

Prediction: $\hat{y} = \sigma(w_1x_1 + w_2x_2 + b)$

○ **Solution 1:** $x_1 + x_2$

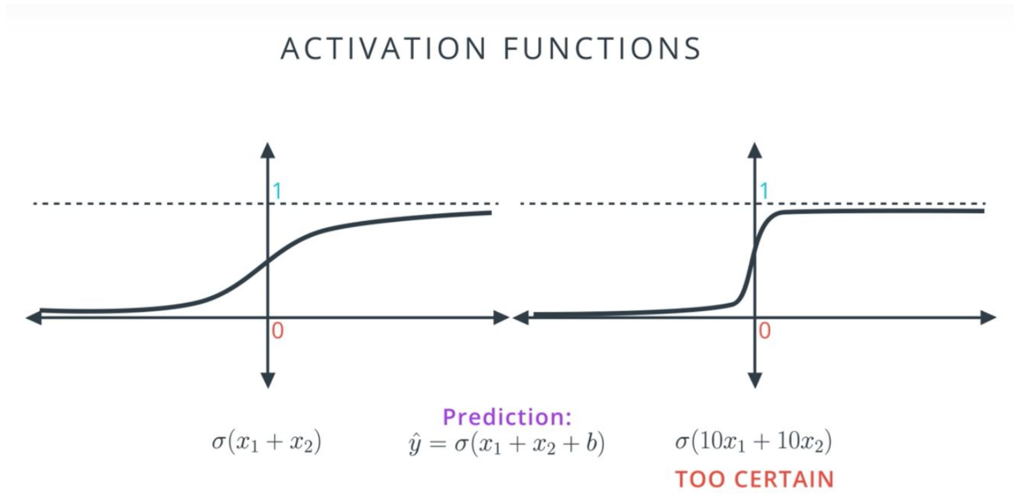
○ **Solution 2:** $10x_1 + 10x_2$

Solution:
Remember that to compute the error we first compute the prediction value:

$$\widehat{y}_1(1, 1) = \sigma(2) = 0.88,\ \widehat{y}_1(-1, -1) = \sigma(-2) = 0.12$$
$$\widehat{y}_2(1, 1) = \sigma(20) = 1.0,\ \widehat{y}_2(-1, -1) = \sigma(-20) = 0$$

We don't really need to evaluate the error function. We can see from these values that the predictions from 'Solution 2' are much closer to the true values than 'Solution 1'. So the answer is **Solution 2**.

What gives in the above question? Both of the two equations represent the same boundary (i.e. they are scalar multiples of each other). The answer is in the sigmoid function. Ideally, we would have a sigmoid function that is smooth (like in solution 1).



The second solution is too certain of itself and gives us little room for adjustment.

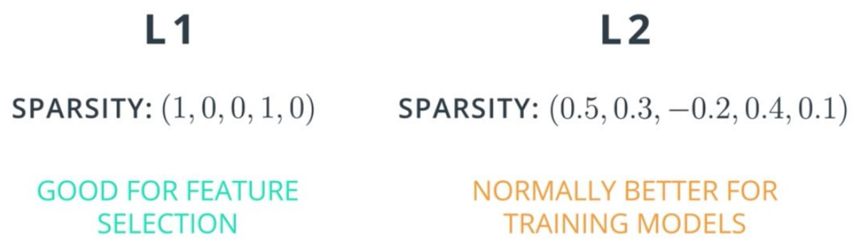
Regularization

So, under the assumption that the large weights imply overfitting, we need to add a term that penalizes larger weights (we call this regularization)

$$\text{L1 ERROR FUNCTION} = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \boxed{\lambda(|w_1| + ... + |w_n|)}$$

$$\text{L2 ERROR FUNCTION} = -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \boxed{\lambda({w_1}^2 + ... + {w_n}^2)}$$

How do we choose the type of regularization to use?

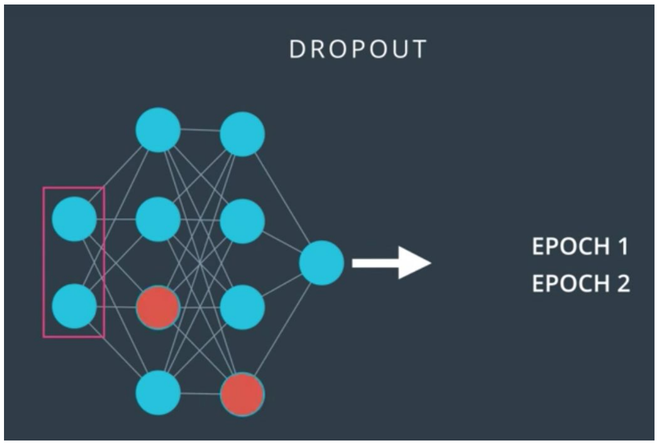


L1	L2
SPARSITY: $(1, 0, 0, 1, 0)$	SPARSITY: $(0.5, 0.3, -0.2, 0.4, 0.1)$
GOOD FOR FEATURE SELECTION	NORMALLY BETTER FOR TRAINING MODELS

We will typically use L2 normalization in this course.

Dropout

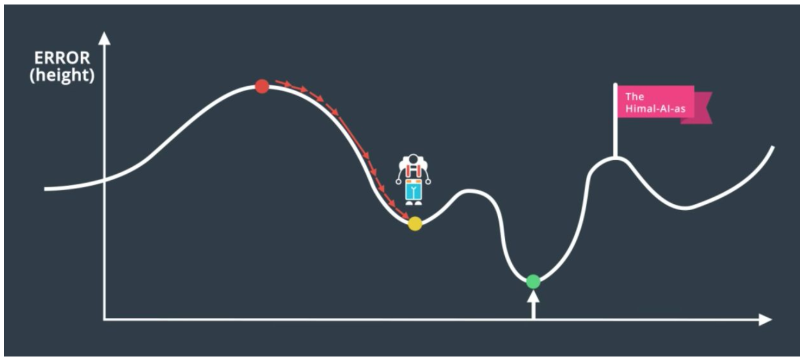
To prevent certain nodes from dominating the training, we can randomly turn off nodes in each epoch. This forces the nodes that were not participating as strongly to contribute more.



Typically we will specify the probability that nodes will drop out.

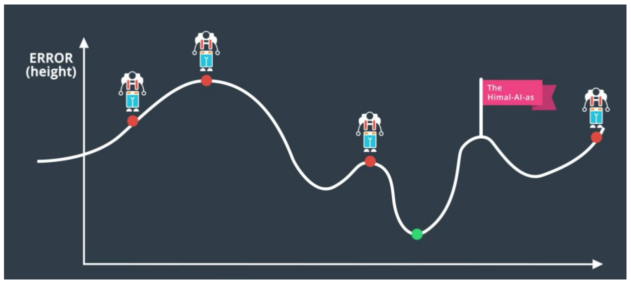
Problem: Local Minima

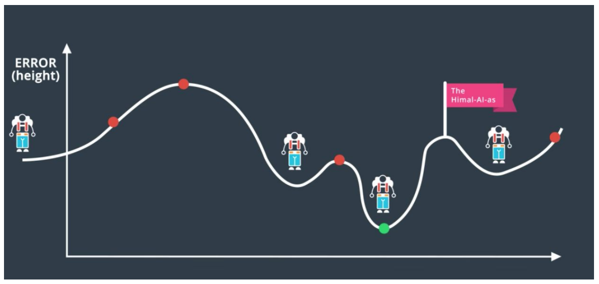
A local minimum happens when we reach a point where the error is increasing in all directions, however a better minimum (and values for our model parameters) exists.



Solution: Random Restart

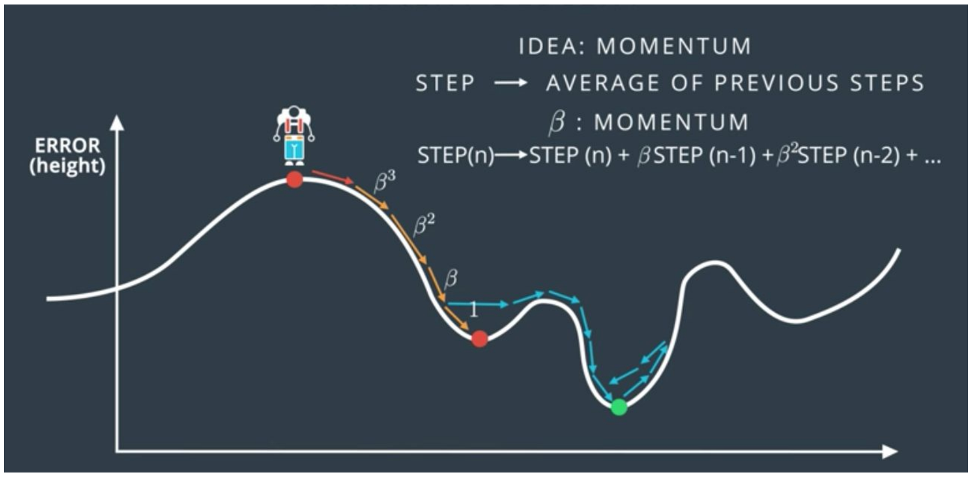
Here we start our model training from a few random locations and apply gradient descent to all of them. The idea is that some of them should find the global minimum (or at least a better local minimum).





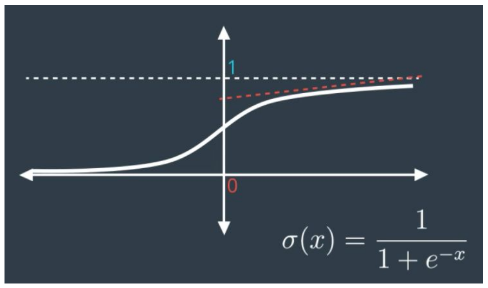
Solution: Momentum

We can use the size of previous steps to add 'momentum' to each step, effectively allowing us to overcome a local minimum, pushing us (hopefully) into the region of global minimum.


IDEA: MOMENTUM
STEP → AVERAGE OF PREVIOUS STEPS
β : MOMENTUM
STEP(n) → STEP (n) + βSTEP (n-1) + β^2STEP (n-2) + ...
ERROR (height)
β^3
β^2
β
1

Problem: Vanishing Gradient

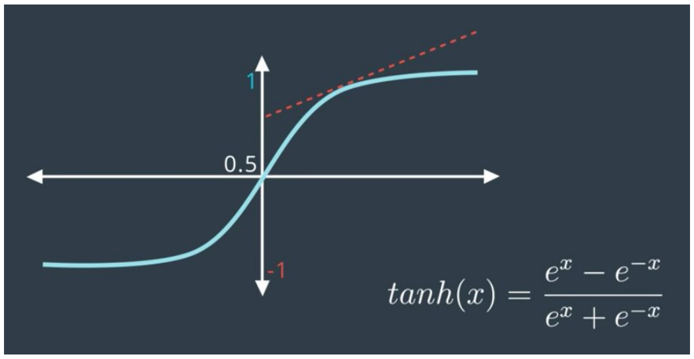
One problem with the sigmoid function is that as we move farther left or right, the derivative of the sigmoid function becomes closer and closer to zero. This means that our gradients are small, which yield an even smaller adjustment to our weights. Small adjustments could cause a very slow training of our model.



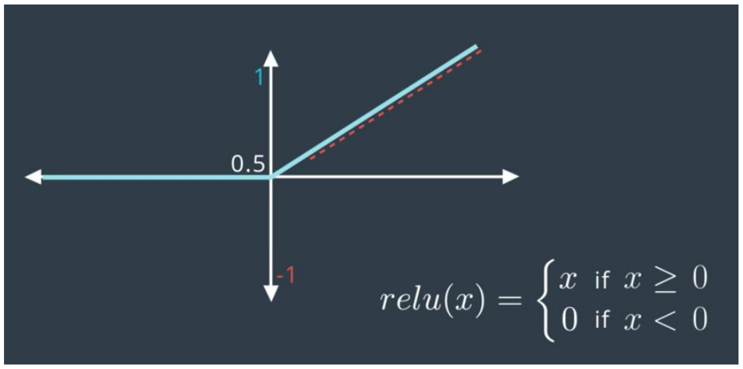
Solution: Other activation functions

There are other activation functions that we can use.

- **Hyperbolic Tangent** (tanh): Because the function spans the range [-1,1] (instead of [0,1] like the sigmoid function) it's derivatives are actually larger.



- **Rectified Linear Unit** (ReLU): This function has a constant derivative of 1 above x=0, which makes it useful in regression problems. Note that the last output layer will still use a sigmoid function when we're doing classification problems.



Batch vs. Stochastic Gradient descent

One of the largest issues with training time is how long it takes to forward-feed all of the data through the network and compute the gradients for every data point. This is called 'batch gradient descent'.

Batch Gradient Descent

Batch gradient descent takes all of our data in every epoch, computes the contribution to the gradient for every data point, and uses this to adjust the weights.

Stochastic Gradient Descent

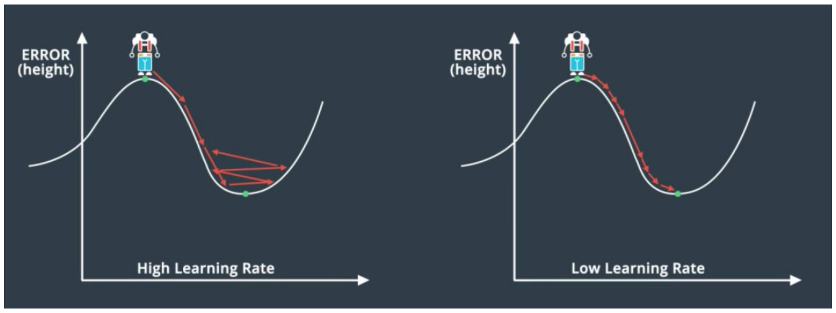
Stochastic gradient descent splits the data into several batches. Each epoch then uses one of these batches to adjust the network. This means that for the same number of computations we can move the weights much closer to the global minimum.

Problem: What Learning Rate To Use?

One of the big questions when building our models is what value to use for the learning rate?

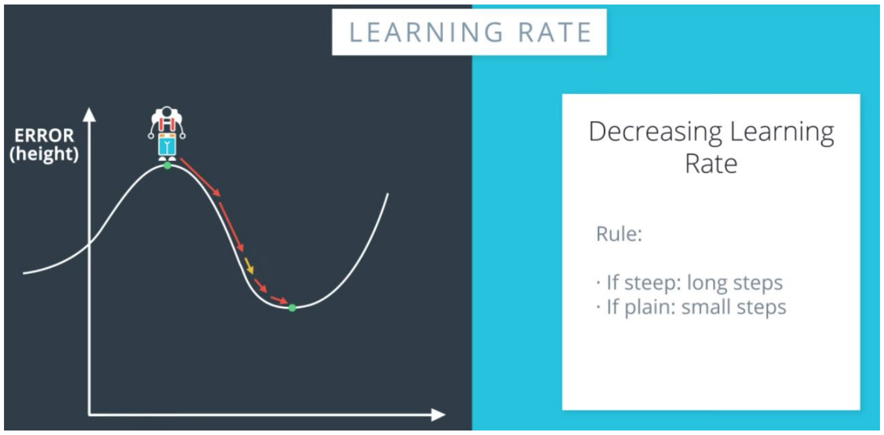
- High learning rate: The model will train more quickly in the beginning, but might miss the minimum and not settle on the best values. Models might be more error prone.

- Low learning rate: The model will train very slowly.



Solution: Self adjusting learning rate

The best learning rate is one that is large when the error is changing dramatically (i.e. far from the minimum), and small when the error is changing slowly (i.e. near the minimum).



Error Functions Around the World (Here be Dad jokes)

Here are the error functions that we met in this lesson:

- Mount Errorest
- Mount Kilimanjerror

Here's some we didn't meet:

- Mount Reinerror
- Mount Ves-oops-vius
- Eyjafvillajökull