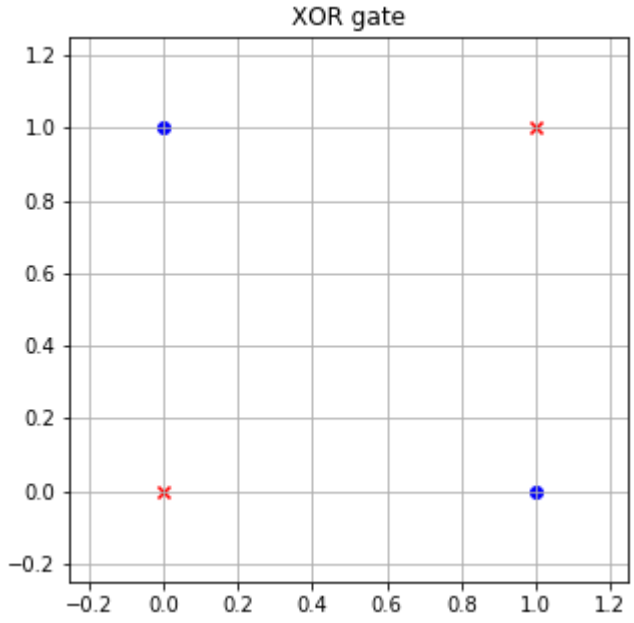
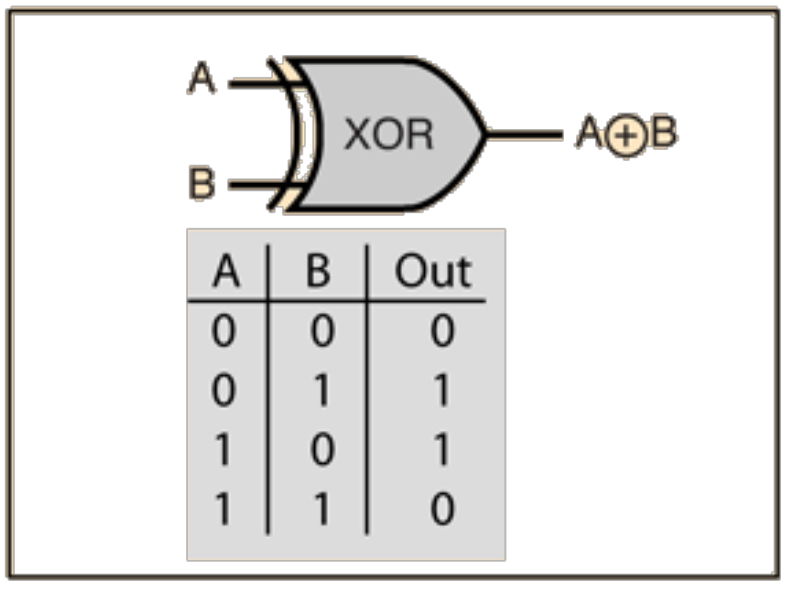
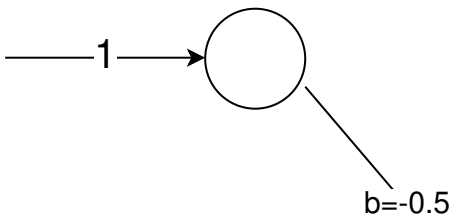
Multi-Layer Perceptron (MLP) for XOR gate

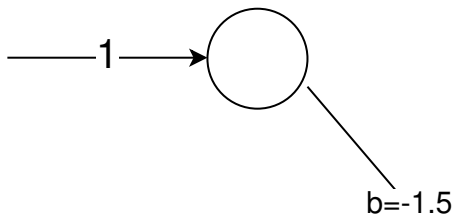
The perceptron that we used for the OR and AND gates can only give us a **linear boundary**. It is evident that it is simply not possible for a linear decision boundary to correctly classify all the points in XOR gate.



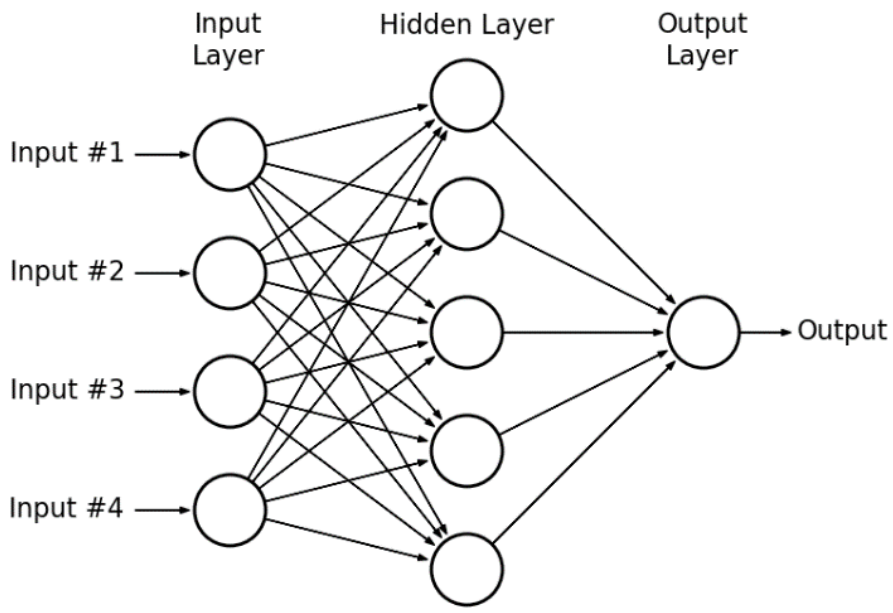


It is pretty common for real-life classification problems to have non-linear decision boundaries. That is why we add hidden layers to the neural networks. The layers other than the input and output layers in the network are called hidden layers. Deep Learning typically means that the network is pretty deep (many hidden layers!).





Formulating Multi-Layer Perceptron (MLP) (or Fully Connected Network)

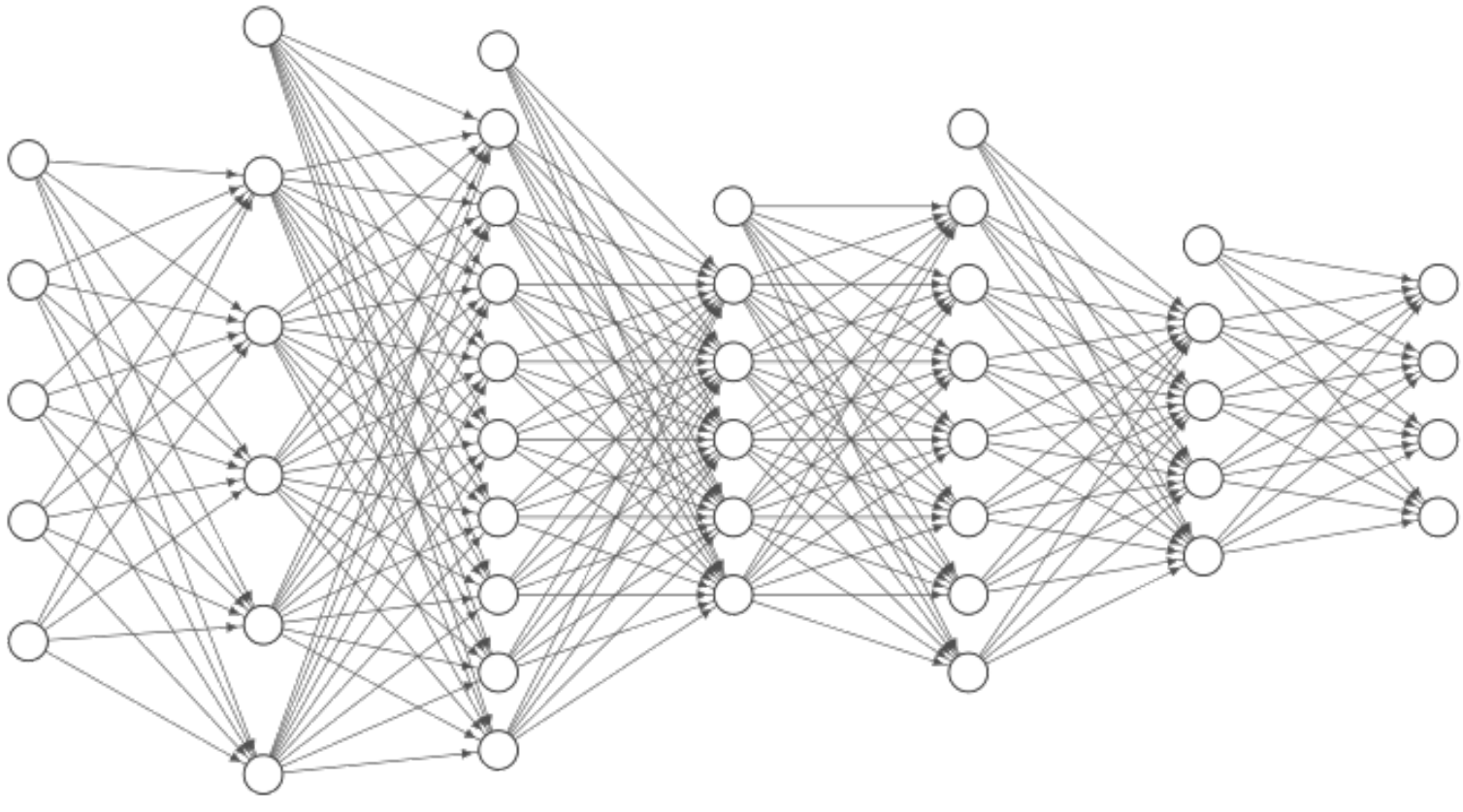


Notations:

- Each unit represented as a circle is called a node.
- The first layer of the network is called the input layer.
  - It takes input values for the examples.
  - The number of nodes in the input layer is equal to the number of independent variables in the dataset.
- The last layer is called the output layer.
  - It makes the prediction.
  - For a binary classification, there is a single node in the output layer.
  - For multiclass (more than two) problem, the number of output nodes equals the number of classes.
- All the intermediate layers are called hidden layers and there can be several of them.
- For the vanilla network, every node from one layer is connected to every node in the next layer and the connections are given by weights.
- All layers except the output layer have a bias node, though they are often not shown in the neural network diagrams.
  - The bias always gets the input of 1
  - The bias has no incoming connection from the nodes in the previous layers, unlike other nodes in the hidden layers.
  - The bias is connected to each node in the next layer, just like other nodes.
- The output/activation of each layer is calculated in two steps: weighted sum of the incoming input followed by the activation function (for eg. the unit step function).

$$z_k = a_{k-1} * W_k + b_k$$
$$a_k = g_k(z_k)$$

- The output/activation of each hidden layer becomes the input of the next layer.
- All the nodes in a layer share the same activation function but the activation functions can differ from layer to layer.
- The activation functions as well as addition of hidden layers contributes to the non-linearity in the model.
- The weights and bias of a neural network are learned using the training examples.



Probability for classification

In the examples so far, the network will output either a 0 or 1 for negative or positive class respecively.

We would usually want our classifier to give us the probabilities corresponding to each class, instead of the class label. Can you guess why?

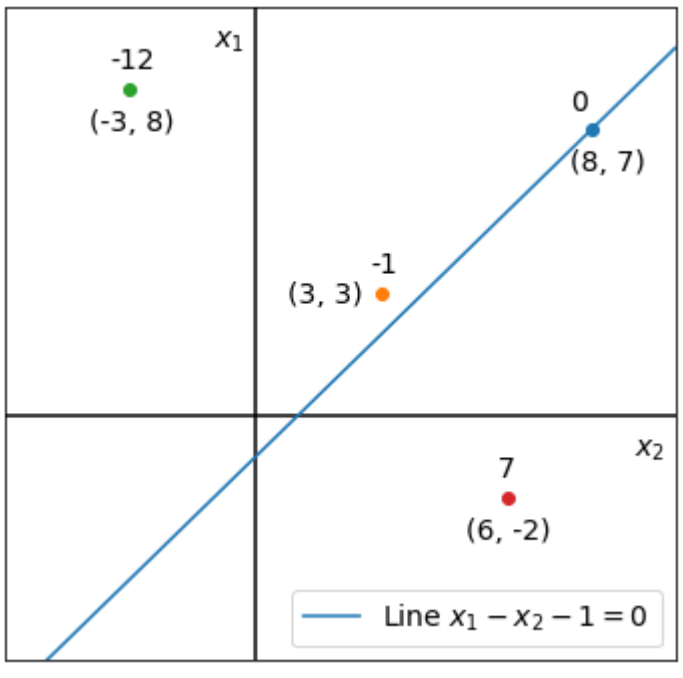
A: The examples farther from the decision boundary should be classified with more certainity (or higher probability) than those closer to the decision boundary.

Our network currently outputs either 1 or 0 to denote the positive or negative class respectively by making use of unit step function. We want to model our network to **output p, the probability that an observation belongs to the positive class**. From this, we can easily compute $1 - p$, the probability that an observation belongs to the negative class.

What can we use instead of unit step function in the output layer to give us a probability in proportion to the distance of a point from the decision boundary?

Let us first start with a math question: How do we mathematically quantify how close a point is to the decision boundary?

A: The expression $x_1 - x_2 - 1$ have values higher in magnitude for points away from the line and lower values for points closer to the line.



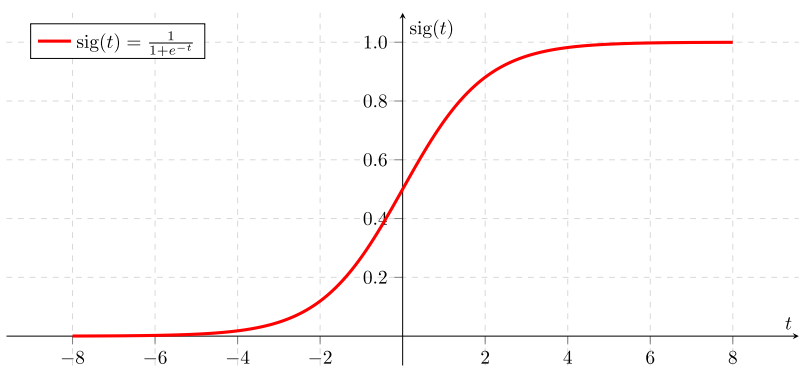
The magintiude of the expression $w_1 * x_1 + w_2 * x_2 + \cdots + w_n * x_n + b$ is higher for points away from the decision boundary and its sign tells us which region (positive or negative) that point fall into.

If we combine the above expression with a suitable function, it can give us the probability for the positive class.

Sigmoid function as activation function

The **sigmoid function** is defined as follows:

$$g(t) = \frac{1}{1 + e^{-t}}$$



The S-shaped curve is called sigmoid because of its shape and it is widely used in population growth models and hence, the name logistic (https://en.wikipedia.org/wiki/Logistic_function).

Observations:

- The output of the sigmoid lies between 0 and 1, which is the same as the range of probability.
- It approximates to 1 for large positive values, whereas it converges to 0 for large negative values.
- In view of the above equation for logistic regression and the properties of sigmoid logistic function, the points farther away from the line will be classified with a high probability to one class or the other, whereas the probability will be closer to 0 for points close to the line.

$$p = sig(w_1 * x_1 + w_2 * x_2 + \cdots + w_n * x_n + b)$$

In general, we set the threshold for probability to be 0.5. This means:

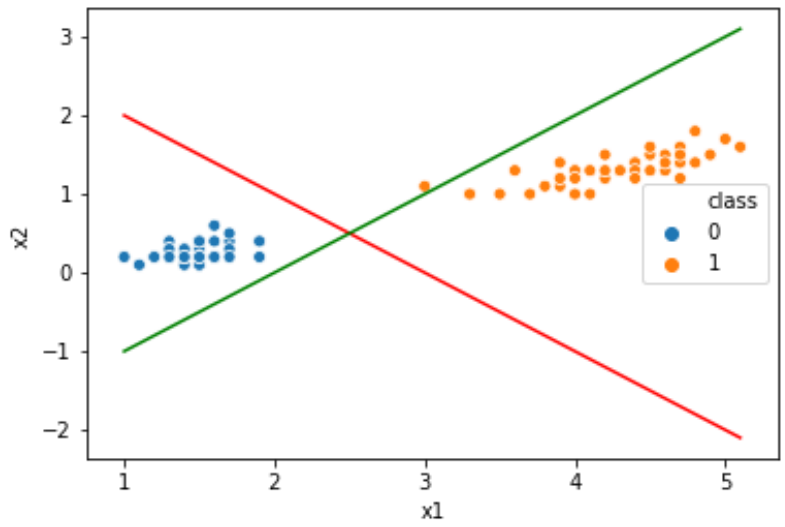
- Whenever $w_1 * x_1 + w_2 * x_2 + \cdots + w_n * x_n + b$ is positive, it is classified to the positive class and the probability will be higher if the magnitude is large.
- Whenever $w_1 * x_1 + \cdots + w_n * x_n + b < 0$ is negative, it is classified to the negative class and the probability will be higher if the magnitude is large.
- The points for which the value for $w_1 * x_1 + \cdots + w_n * x_n + b$ is not large in magnitude have probabilities that are closer to 0.5. Such points needs to be classified with extra care, as we will see later on in the topic of evaluation metrics.

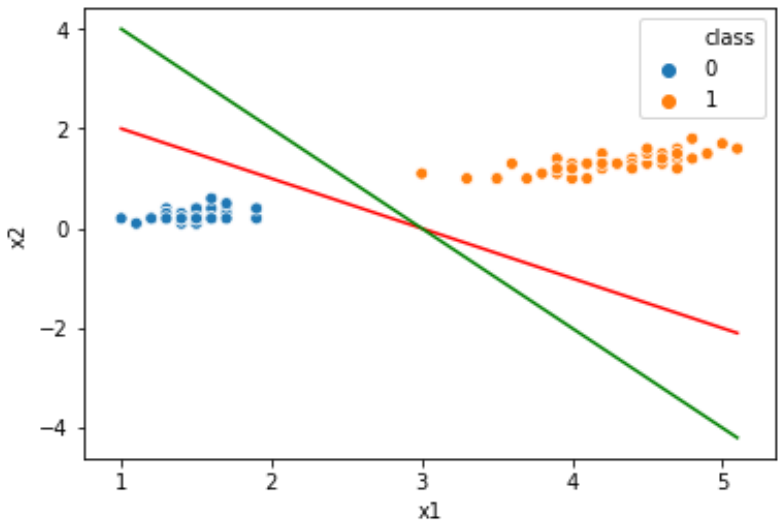
Formulating Loss or Cost of Classification

To learn a good decision boundary, we need to make use of our training data for **finding the optimal values for the weights and biases**. We have used trial-and-error in the examples so far which clearly would not scale. Before we learn the algorithms we can use to train the network (training means learning weights and biases), we should be able to mathematically measure the "goodness" of a decision boundary.

For the following two examples,

- Which of the below two lines - red or green is a better decision boundary?
- How do we decide that? What metrices should we use to guide that decision?
- Should we only consider what percentage of points are correctly classified? If not, what else?





For calculating the loss or cost of classification, we should aim for:

- classifying the points correctly
- maximizing the distance of correctly classified points from the decision boundary

Our network outputs the probability which carries information about the distance of points from the decision boundary. We use the log loss function, also known as cross-entropy function, defined below to quantify the loss or cost function.

Note: *There is an exponential in the definition of sigmoid function $g(t) = \frac{1}{1+e^{-t}}$, so we will be taking the logarithm (natural or another base would not make a difference) of the output of our network. Without going into too many details, the insight for using logarithm can also be gained by the derivation of probabilities using the Maximum Likelihood Principle for a simple neural network without hidden layers. See here (http://rasbt.github.io/mlxtend/user_guide/classifier/LogisticRegression/) and here (https://houxianxu.github.io/2015/04/23/logistic-softmax-regression/). We will skip it because it is not relevant to the topics in this course.*

Recall that p is the probability that the data point belongs to the positive class with label 1.
For points with label $y = 1$, the cost is

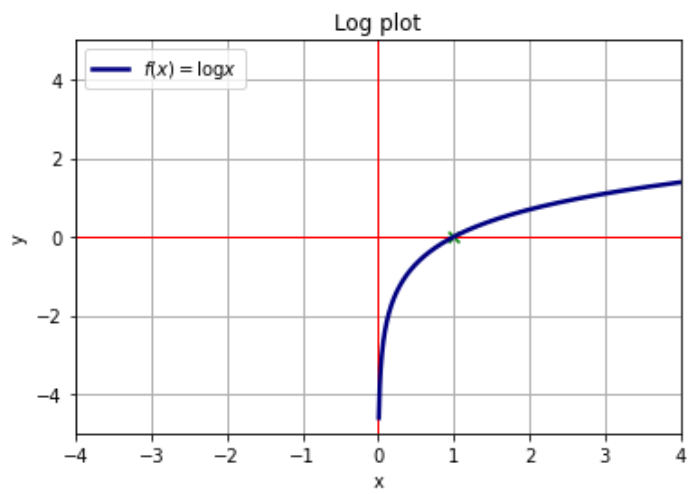
$$c(y, p) = -\log(p) \qquad \text{if} \quad y = 1$$

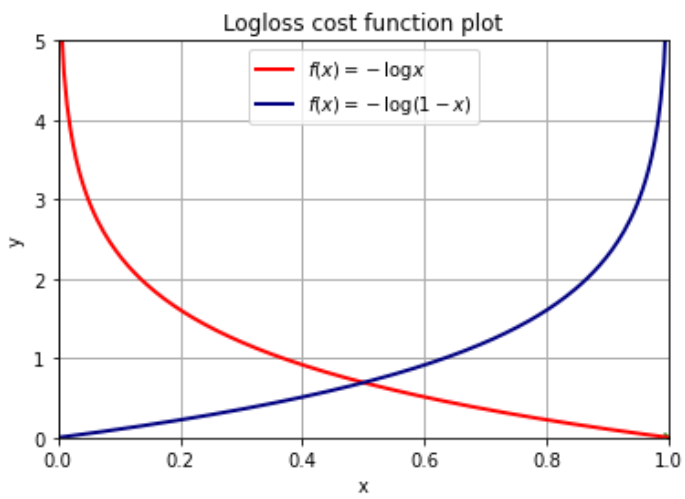
whereas for points with label $y = 0$, the cost is

$$c(y, p) = -\log(1 - p) \quad \text{if} \quad y = 0$$

Recall that:

$$\lim_{p \to 1} \log(p) = 0 \quad \text{and} \quad \lim_{p \to 0} \log(p) = -\infty$$





Observations:
For data point with the true class $y = 1$

- As the predicted probability $p \to 1$, the cost $c \to 0$.
- As the predicted probability $p \to 0$, the cost $c \to \infty$.

For data point with the true class $y = 0$

- As the predicted probability $p \to 0$, the cost $c \to 0$.
- As the predicted probability $p \to 1$, the cost $c \to \infty$.

The cost (also known as loss function) function takes the average over the costs for all points. The costs for the two classes $y = 0$ and $y = 1$ can be summed up in the following formula.

$$J = \frac{1}{N}\sum_{i=1}^{N} c(y, p) = -\frac{1}{N}\sum_{i=1}^{N} \left(y \log(a_2) + (1 - y) \log(1 - a_2)\right)$$

where $p = Prob(y = 1)$.

The cost function $J(p)$ is effectively a function of the weights and the biases b, that is $J(w, b)$ where w and b is the placeholder for all weights and biases in the network. Can you see how?

Note that x and y are given to us and hence, are constants when it comes to training whereas the weights and the biases are the variables for which we will find the optimal values.

The cost function J is something we want to minimize. How do we find the minima for a multivariate function J? Suppose J is a function of a single variable w, how would you solve it?

Gradients

Suppose you are standing at the top of a hill and want to descend to the plain. If you do not have any specific destination in mind, but want to take the least number of steps to reach the plain, what would be your **strategy for each step on your way**?

Have you noticed the paths followed by the creeks along the mountains?

Ans: You pick the direction of the steepest descent at each step.

Let us formulate this optimization strategy in mathematical terms.

Q: Given a curve represented by a function $J(w)$, how do you get the slope of the curve at each point?

Q: Given a curve represented by a multi-variable function $J(w_1, w_2, \ldots, w_n, b)$, how do you get the slope of the curve at each point?

What are gradients?

Gradients can be thought of as an extension of derivatives. For a multivariable function f, the gradient (https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/partial-derivative-and-gradient-articles/a/the-gradient) of f is the vector of partial derivatives.

$$\nabla f(x_1, x_2, \ldots, x_n) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

Geometrically, the gradient points in the direction of the steepest slope.

Questions to ponder over:

- If we move in the direction of steepest descent, are we always guaranteed to reach the minimum point?
- If the answer is no to the above question, are there any cases for the function J for which

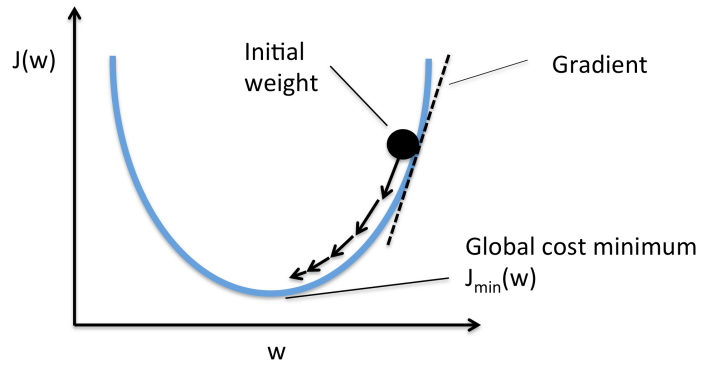
Gradient Descent Algorithm

Gradient Descent algorithm is used to **iteratively update the weights using the training examples so as to minimize the cost function** J.

The weights are updated in the direction of the steepest descent of the cost function J in each iteration.

$$w := w - \alpha \nabla J$$

where ∇J is the gradient of the cost function J and α is the learning rate that determines the size of steps that we take descending on the path of gradient.



Minimizing the cost function using gradient descent

In a nutshell, the learning process can be summarized as iteratively updating the weights using the training data to keep on ***minimizing the cost function***. The gist of the learning process for the neural network is the same, though the formulation of the cost function and the equation for calculating y for a given x will vary a lot depending on the architecture of the neural network.

Let us formalize the neural networks introduced so far. There are two parts to the learning process:

1. Forward propagation: used to predict the output by propagating the input in the forward direction
2. Backward propagation: used to compute the weight updates for each layer by propagating the cost/error in the backward direction.

Break for a pop quiz!

- Please sign into your Google Classroom.
- You may also want to grab pen and paper if it helps you to think clearly.
- Please do not use google for answering the questions on the quiz.

Announcements:

- The first assignment will be released today after class and it is due before next class on Feb 1st at 4:15 pm. Please submit the assignment using the option to "Turn in" in the Classroom.
- You should figure out the backprop exercise in the lecture notes first before coding the backprop in the assignment.
- If you want to discuss the problems with your classmates or ask me questions, please come to my office hours on Zoom using the same link on Monday and Thursday from 12:30-1:30 pm.
- There will be 8 pop quizzes in total. Each quiz is worth 1.5 points, so 12 points in total. You only

need 9 points to score 100% for the quiz component of the total grade but if you score extra points, they will be adjusted towards In-class tests and participation grade if needed.

Forward propagation

The process of propagating the output of each layer in the forward direction to consequently get the final output is called forward propagation.

The output of each hidden layer becomes the input of the next layer. The output is also called the activation for a layer.

The output/activation for each layer is computed in two steps:

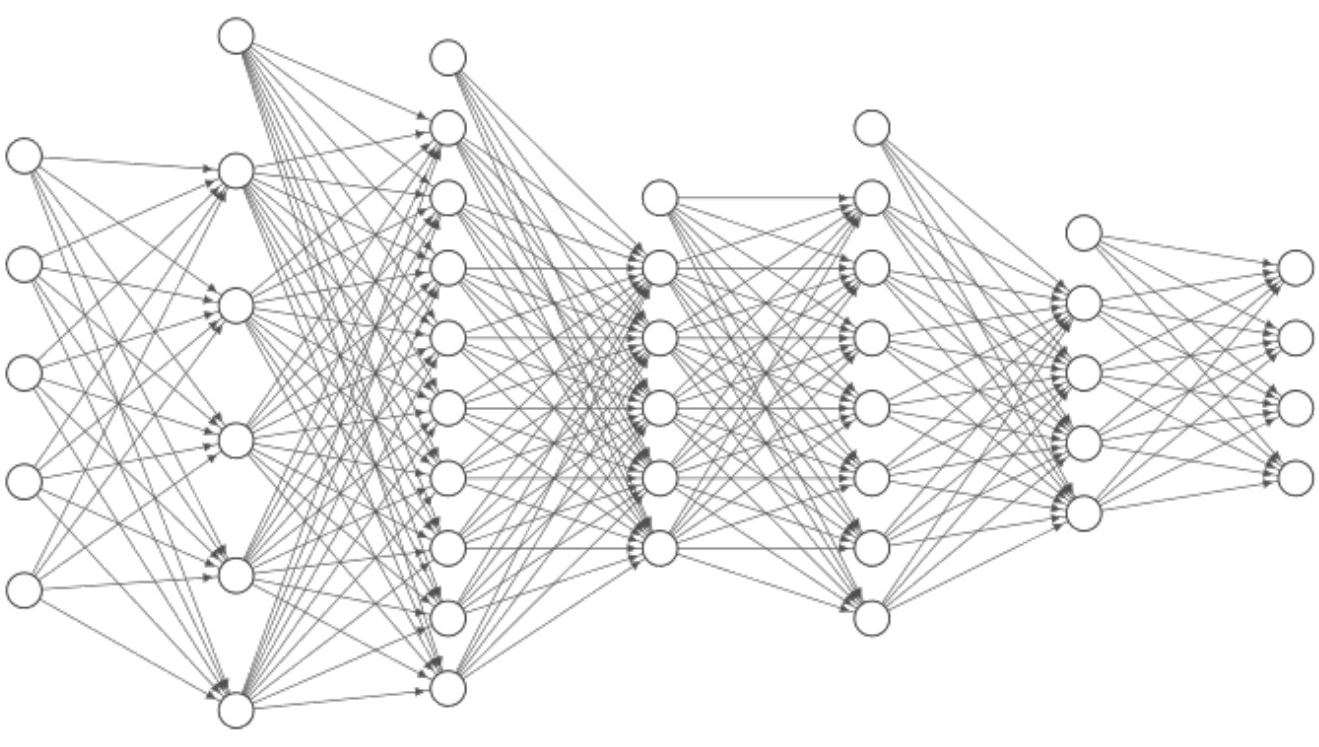
- The weighted sum of the inputs, say z_i
- The activation function is applied to the above sum z_i to produce the activation a_i

Equations:

$$
\begin{aligned}
z^{(1)} &= x\, W^{(1)} + b^{(1)} \\
a^{(1)} &= g_1(z^{(1)}) \\
z^{(2)} &= a^{(1)} W^{(2)} + b^{(2)} \\
a^{(2)} &= g_2(z^{(2)}) \\
&\vdots \\
z^{(n)} &= a^{(n-1)} W^{(n)} + b^{(n)} \\
a^{(n)} &= g_3(z^{(n)})
\end{aligned}
$$

and so on till the final output $y_{pred} = a^{(n)}$.

Convention:
$z^{(i)}$: weighted averages of the output from the $(i-1)^{th}$ layer
$a^{(i)}$: activation/output of the i^{th} layer
g_i: activation layer of the i^{th} layer
$W^{(i)}$: Weight matrices connecting two layers
$b^{(i)}$: Bias vector for the i-th layer



Backward propagation

The process of propagating the cost in the backward direction to compute the gradients for each layer so as to update the weights and bias is called backward propagation.

Equations:

$$W^{(n)} := W^{(n)} - \frac{1}{m}\alpha\frac{\partial J}{\partial W^{(n)}}$$

$$b^{(n)} := b^{(n)} - \frac{1}{m}\alpha\frac{\partial J}{\partial b^{(n)}}$$

$$\vdots$$

$$\vdots$$

$$W^{(1)} := W^{(1)} - \frac{1}{m}\alpha\frac{\partial J}{\partial W^{(1)}}$$

$$b^{(1)} := b^{(1)} - \frac{1}{m}\alpha\frac{\partial J}{\partial b^{(1)}}$$

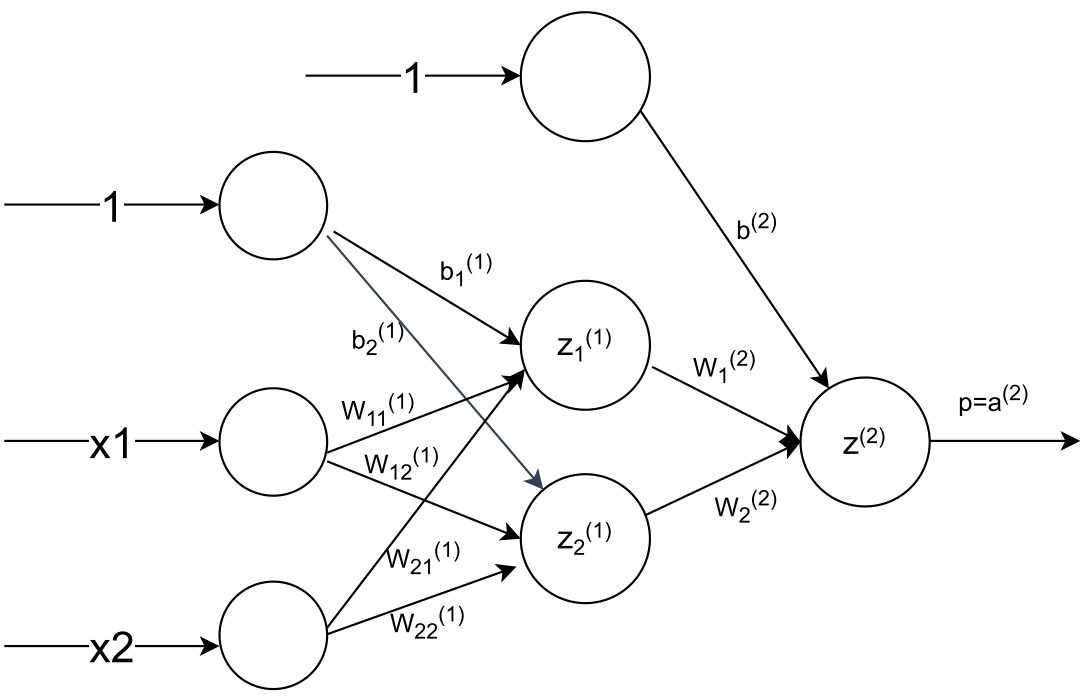
Here, α is the learning rate that is multiplied to the gradients to tune the size of each weight/bias update. m is the number of training examples.

The gradients are computed using the chain rule for derivatives, as illustrated below.

One pass of each forward and backward propagation is called an iteration. When all the training

Derivation of Backpropagation equations

We will derive the equation for the neural network with a single hidden layer shown below:



Let us first write down the equations for **forward propagation** that we will use to derivae the gradients for backpropagation:

$$(x_1, x_2) \rightarrow \left(z_1^{(1)}, z_2^{(1)}\right) \rightarrow \left(a_1^{(1)}, a_2^{(1)}\right) \rightarrow z^{(2)} \rightarrow p = a^{(2)}$$

$$z_1^{(1)} = w_{11}^{(1)}x_1 + w_{21}^{(1)}x_2 + b_1^{(1)} \qquad a_1^{(1)} = g(z_1^{(1)})$$

$$z_2^{(1)} = w_{12}^{(1)}x_1 + w_{22}^{(1)}x_2 + b_2^{(1)} \qquad a_2^{(1)} = g(z_2^{(1)})$$

$$z^{(2)} = w_1^{(2)}a_1^{(1)} + w_2^{(2)}a_2^{(1)} + b^{(2)}$$

We know the derivative for the sigmoid activation function

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

and for the logloss cost function:

$$J = -\left(y \log(a_2) + (1-y) \log(1-a_2)\right)$$
$$\frac{dJ}{} = -\left(\frac{y}{} - \frac{1-y}{}\right) = \frac{a_2 - y}{}$$

We start calculating the gradients from the last node and propagate backwards using the chain rule for partial derivatives:

$$\frac{\partial J}{\partial a^{(2)}} = \frac{a^{(2)} - y}{a^{(2)}\left(1 - a^{(2)}\right)}$$
$$\frac{\partial J}{\partial z^{(2)}} = \frac{\partial J}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} = \frac{a^{(2)} - y}{a^{(2)}\left(1 - a^{(2)}\right)} a^{(2)}\left(1 - a^{(2)}\right) = a^{(2)} - y$$
$$\frac{\partial J}{\partial w_1^{(2)}} = \frac{\partial J}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w_1^{(2)}} = \left(a^{(2)} - y\right)\ a_1^{(1)}$$
$$\frac{\partial J}{\partial w_2^{(2)}} = \frac{\partial J}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w_2^{(2)}} = \left(a^{(2)} - y\right)\ a_2^{(1)}$$
$$\frac{\partial J}{\partial b^{(2)}} = \frac{\partial J}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial b^{(2)}} = \left(a^{(2)} - y\right)$$
$$\frac{\partial J}{\partial a_1^{(1)}} = \frac{\partial J}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_1^{(1)}} = \left(a^{(2)} - y\right)\ w_1^{(2)}$$
$$\frac{\partial J}{\partial a_2^{(1)}} = \frac{\partial J}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a_2^{(1)}} = \left(a^{(2)} - y\right)\ w_2^{(2)}$$
$$\frac{\partial J}{\partial z_1^{(1)}} = \frac{\partial J}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial z_1^{(1)}} = \left(a^{(2)} - y\right)\ w_1^{(2)}\ a_1^{(1)}\left(1 - a_1^{(1)}\right)$$
$$\frac{\partial J}{\partial z_2^{(1)}} = \frac{\partial J}{\partial a_2^{(1)}} \frac{\partial a_2^{(1)}}{\partial z_2^{(1)}} = \left(a^{(2)} - y\right)\ w_2^{(2)}\ a_2^{(1)}\left(1 - a_2^{(1)}\right)$$
$$\frac{\partial J}{\partial w_{11}^{(1)}} = \frac{\partial J}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{11}^{(1)}} = \left(a^{(2)} - y\right)\ w_1^{(2)}\ a_1^{(1)}\left(1 - a_1^{(1)}\right)\ x_1$$
$$\frac{\partial J}{\partial w_{21}^{(1)}} = \frac{\partial J}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial w_{21}^{(1)}} = \left(a^{(2)} - y\right)\ w_1^{(2)}\ a_1^{(1)}\left(1 - a_1^{(1)}\right)\ x_2$$
$$\frac{\partial J}{\partial w_{12}^{(1)}} = \frac{\partial J}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_{12}^{(1)}} = \left(a^{(2)} - y\right)\ w_2^{(2)}\ a_2^{(1)}\left(1 - a_2^{(1)}\right)\ x_1$$
$$\frac{\partial J}{\partial w_{22}^{(1)}} = \frac{\partial J}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial w_{22}^{(1)}} = \left(a^{(2)} - y\right)\ w_2^{(2)}\ a_2^{(1)}\left(1 - a_2^{(1)}\right)\ x_2$$
$$\frac{\partial J}{\partial b_1^{(1)}} = \frac{\partial J}{\partial z_1^{(1)}} \frac{\partial z_1^{(1)}}{\partial b_1^{(1)}} = \left(a^{(2)} - y\right)\ w_1^{(2)}\ a_1^{(1)}\left(1 - a_1^{(1)}\right)$$
$$\frac{\partial J}{\partial b_2^{(1)}} = \frac{\partial J}{\partial z_2^{(1)}} \frac{\partial z_2^{(1)}}{\partial b_2^{(1)}} = \left(a^{(2)} - y\right)\ w_2^{(2)}\ a_2^{(1)}\left(1 - a_2^{(1)}\right)$$

Vectorizing the Forward and Backward Propagation equations

Forward propagation:

$$(x_1, x_2) \rightarrow \left(z_1^{(1)}, z_2^{(1)}\right) \rightarrow \left(a_1^{(1)}, a_2^{(1)}\right) \rightarrow z^{(2)} \rightarrow p = a^{(2)}$$

with the equations:

$$z_1^{(1)} = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + b_1^{(1)} \qquad a_1^{(1)} = g(z_1^{(1)}) \qquad z^{(2)} = w_1^{(2)} a_1^{(1)} + w_2^{(2)} a_2^{(1)} + b^{(2)}$$
$$z_2^{(1)} = w_{12}^{(1)} x_1 + w_{22}^{(1)} x_2 + b_2^{(1)} \qquad a_2^{(1)} = g(z_2^{(1)})$$

Converting the set of equations into matrix operations:

$$\left(z_1^{(1)}, z_2^{(1)}\right) = (x_1, x_2) \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{pmatrix} + \left(b_1^{(1)}, b_2^{(1)}\right) \qquad \text{and} \qquad \left(a_1^{(1)}, a_2^{(1)}\right) = \left(g(z_1^{(1)}), g(z_2^{(1}\right.$$

In the vectorized form, the equations are:

$$z^{(1)} = x\, W^{(1)} + b^{(1)}, \qquad a^{(1)} = g(z^{(1)}), \qquad z^{(2)} = a^{(1)} W^{(2)} + b^{(2)}, \qquad a^{(2)} = g(z^{(2)})$$

where

$$z^{(1)} = \left(z_1^{(1)}, z_2^{(1)}\right), \qquad x = (x_1, x_2), \qquad W^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{pmatrix}, \qquad b^{(1)} = \left(b_1^{(1)}, b_2^{(1)}\right),$$

In the vectorized form, the forward propagation is given by:

$$x \longrightarrow z^{(1)} = x\, W^{(1)} + b^{(1)} \longrightarrow a^{(1)} = g(z^{(1)}) \longrightarrow z^{(2)} = a^{(1)} W^{(2)} + b^{(2)} \longrightarrow a^{(2)} = g(z^{(2)}) -$$

Can you write the vectorized equations for the backpropagation based on the partial derivatives calculated above? (This exercise will be crucial for implementing the back propagation code in the first assignment.)

$$\begin{matrix} dW^{(1)} = ? \\ db^{(1)} = ? \end{matrix} \longleftarrow dz^{(1)} = ? \longleftarrow \begin{matrix} dW^{(2)} = ? \\ da^{(1)} = ? \\ db^{(2)} = ? \end{matrix} \longleftarrow dz^{(2)} = ? \longleftarrow$$

where

$$dW^{(1)} = \begin{pmatrix} \frac{\partial J}{\partial w_{11}^{(1)}} & \frac{\partial J}{\partial w_{12}^{(1)}} \\ \frac{\partial J}{\partial w_{21}^{(1)}} & \frac{\partial J}{\partial w_{22}^{(1)}} \end{pmatrix} \qquad db^{(1)} = \begin{pmatrix} \frac{\partial J}{\partial b_1^{(1)}}, & \frac{\partial J}{\partial b_2^{(1)}} \end{pmatrix} \qquad dz^{(1)} = \begin{pmatrix} \frac{\partial J}{\partial z_1^{(1)}}, & \frac{\partial J}{\partial z_2^{(1)}} \end{pmatrix}$$

$$dW^{(2)} = \begin{pmatrix} \frac{\partial J}{\partial w_1^{(2)}} \\ \frac{\partial J}{\partial w_2^{(2)}} \end{pmatrix} \qquad db^{(2)} = \frac{\partial J}{\partial b^{(2)}} \qquad dz^{(2)} = \frac{\partial J}{\partial z^{(2)}}$$

Note: It is important to get the order of matrices and matrix operations right. You should check that your solution works even if you change the number of nodes in the input and hidden layer nodes.

Tip: You can express some of these gradients in terms of other gradients and use that to simplify the equations. For example, can you write $db^{(2)}$ in terms of $dz^{(2)}$? This simplification will also be helpful while implementing the code.

Overfitting and Underfitting to the curve

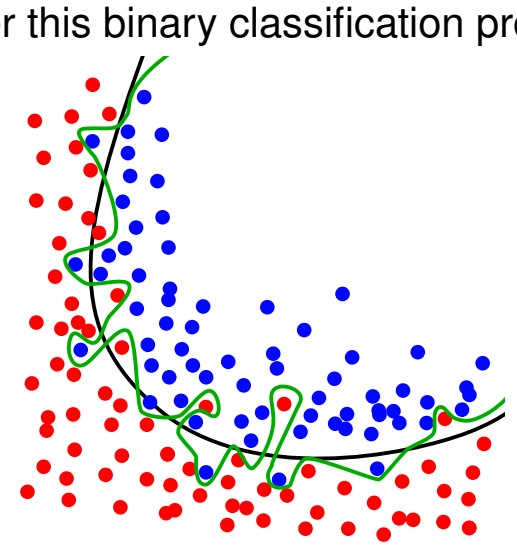
Three classifiers A, B and C are trained on a given labeled dataset. The accuracy of the trained classifiers in predicting the labels correctly on the same dataset is as follows.

Models	Accuracy
Model A	90%
Model B	80%
Model C	70%

Clearly, model A is better at predicting labels for the training data than model B and C. Do you think model A will do a better job in predicting labels for yet unseen data as well?

When should we stop the iterative learning process? Until the cost function has reached its minimum value?

To answer the question, let us consider this binary classification problem with two variables (features).



- Which of the two decision boundaries (black or green) will have a lower value for the cost function?
- Which decision boundary would you prefer for classifying the unseen examples?

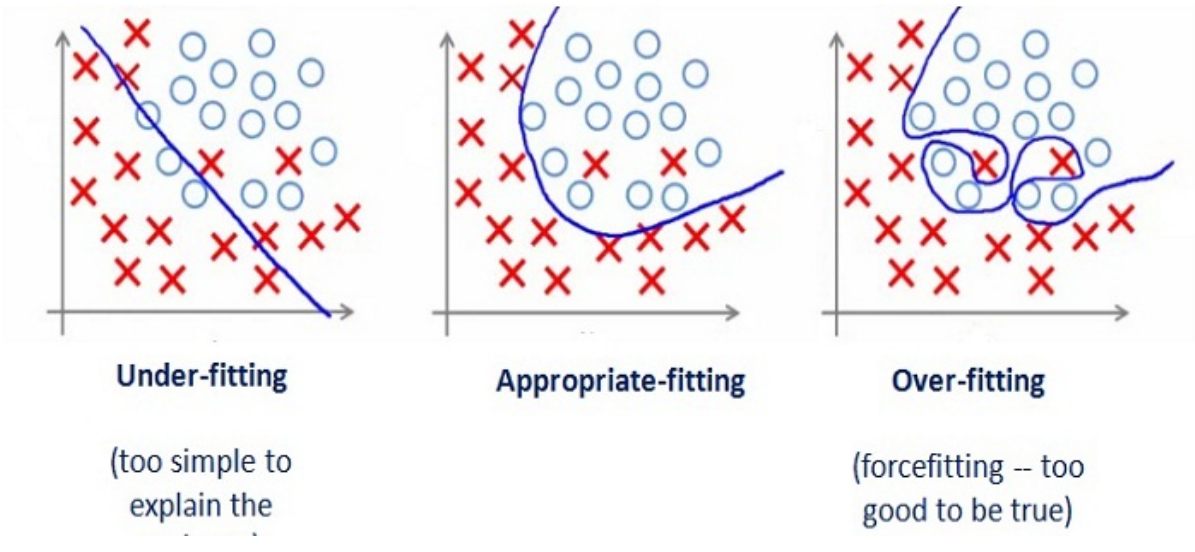
Since the cost function is calculated solely based on the training dataset, minimizing it too much might mean that the network does not generalize well to unseen examples. This is called overfitting.

Over-fitting and under-fitting to the training set

The models can over-train on a dataset, that is they learn the dataset so well that they do not generalize well to the examples outside of that dataset.

If we try to fit too complex of a curve as the decision boundary separating the classes and we don't have enough training examples to estimate the parameters for the curve, then we suffer from over-fitting.

On the other hand, if we try separating the classes with an over-simplified curve as the decision boundary and we have enough training examples to estimate a curve that would be a better fit, then we suffer from under-fitting.



variance)

How do we know whether our model is overfitting or underfitting to the training set?

Answer: At the beginning, we save some examples as the validation set and use it to test the performance of the model.

Models	Accuracy on the training set	Accuracy on the validation set
Model A	90%	70%
Model B	80%	75%
Model C	70%	65%

- With this additional information, can you guess which model will likely perform better for the unseen data?
- Which of these three models would you suspect for overfitting to the training data?
- Which of these three models would you suspect for underfitting to the training data?

Key take-aways so far:

- Always save some examples from the datasets for testing model performance.
- Pay attention to the model performance on the validation set rather than solely on the training set.
- Watch out for both under-fitting and over-fitting.