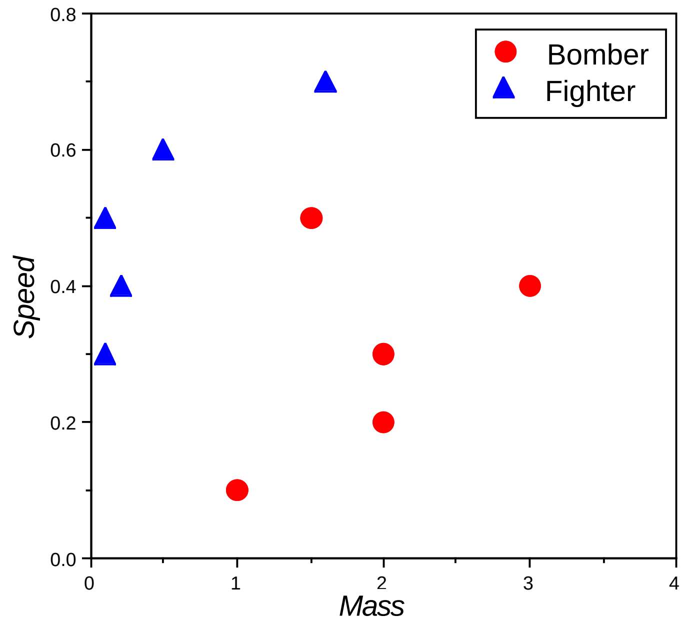
Learning and Generalization in Single Layer Perceptrons

Introduction to Neural Networks : Lecture 4

What Can Perceptrons Do?

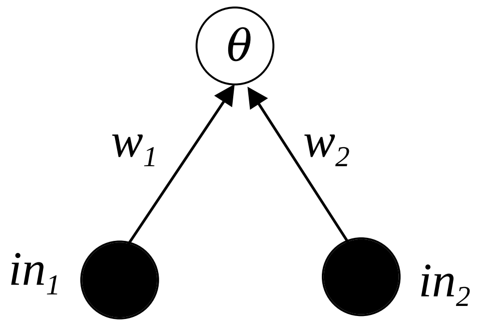
How do we know if a simple Perceptron is powerful enough to solve a given problem? If it can’t even do XOR, why should we expect it to be able to deal with our aeroplane classification example, or real world tasks that will be even more complex than that?



In this lecture, we shall look at the limitations of Perceptrons, and how we can find their connection weights without having to compute and solve large numbers of inequalities.

Decision Boundaries in Two Dimensions

For simple logic gate problems, it is easy to visualise what the neural network is doing. It is forming ***decision boundaries*** between classes. Remember, the network output is:



$$out = \mathrm{sgn}(w_1 in_1 + w_2 in_2 - \theta)$$

The decision boundary (between $out = 0$ and $out = 1$) is at

$$w_1 in_1 + w_2 in_2 - \theta = 0$$

i.e. along the straight line:

$$in_2 = \left(\frac{-w_1}{w_2}\right) in_1 + \left(\frac{\theta}{w_2}\right)$$

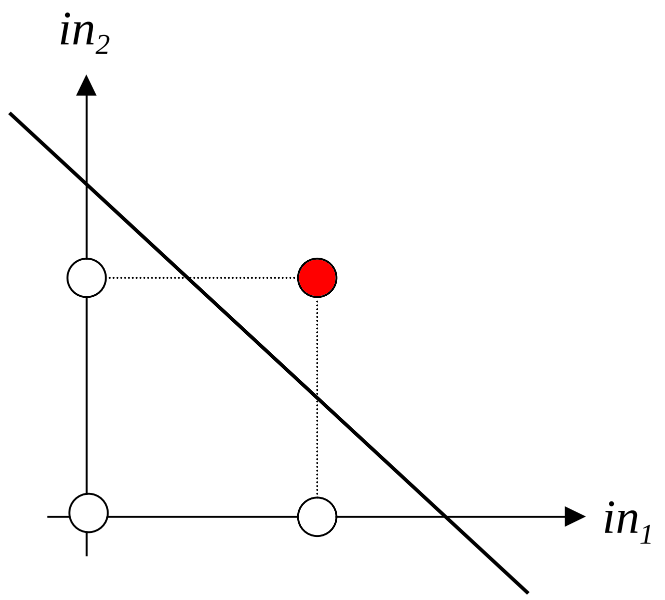
So, in two dimensions the decision boundaries are always straight lines.

Decision Boundaries for AND and OR

We can easily plot the decision boundaries we found by inspection last lecture:

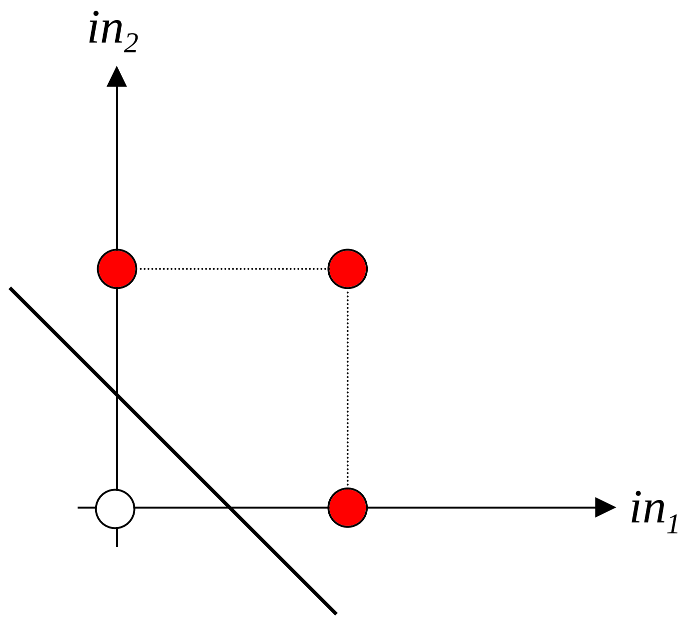
AND

$w_1 = 1,\ \ w_2 = 1,\ \ \theta = 1.5$



OR

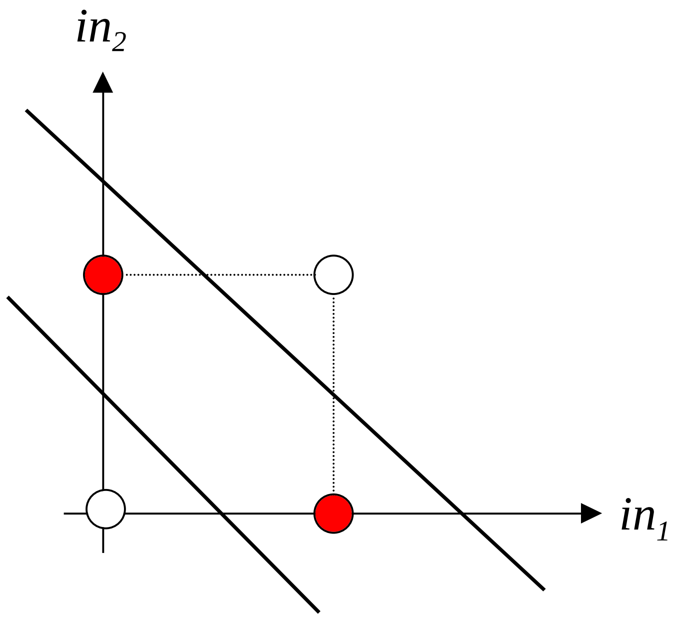
$w_1 = 1,\ \ w_2 = 1,\ \ \theta = 0.5$



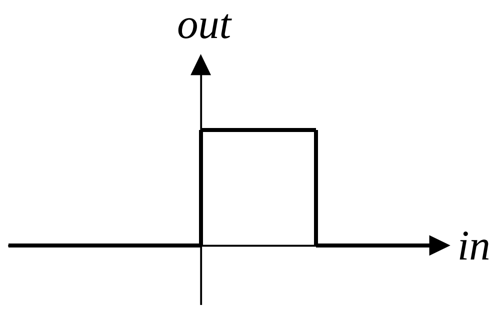
The extent to which we can change the weights and thresholds without changing the output decisions is now clear.

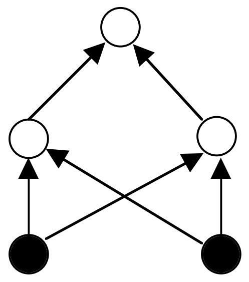
Decision Boundaries for XOR

The difficulty in dealing with XOR is beginning to look obvious. We need two straight lines to separate the different outputs/decisions:



$\Rightarrow$ *e.g.*





There are two obvious remedies: either change the transfer function so that it has more than one decision boundary, or use a more complex network that is able to generate more complex decision boundaries.

Decision Hyperplanes and Linear Separability

If we have two inputs, then the weights define a decision boundary that is a one dimensional straight line in the two dimensional ***input space*** of possible input values.

If we have *n* inputs, the weights define a decision boundary that is an *n-1* dimensional ***hyperplane*** in the *n* dimensional input space:

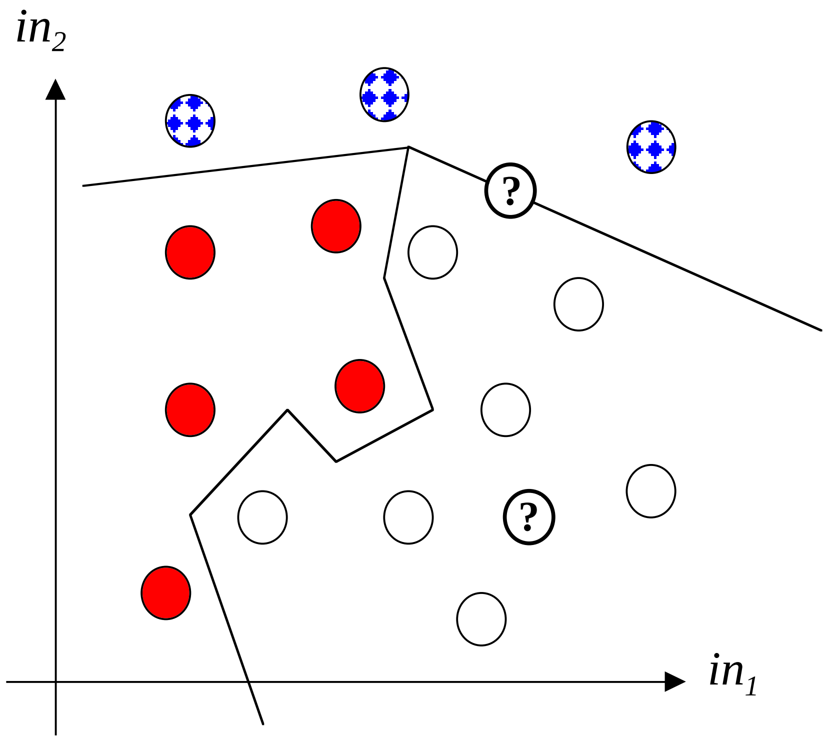
$$w_1 in_1 + w_2 in_2 + \ldots + w_n in_n - \theta = 0$$

This hyperplane is clearly still linear (i.e. straight/flat) and can still only divide the space into two regions. We still need more complex transfer functions, or more complex networks, to deal with XOR type problems.

Problems with input patterns which can be classified using a single hyperplane are said to be ***linearly separable***. Problems (such as XOR) which cannot be classified in this way are said to be ***non-linearly separable***.

General Decision Boundaries

Generally, we will want to deal with input patterns that are not binary, and expect our neural networks to form complex decision boundaries, e.g.



We may also wish to classify inputs into many classes (such as the three shown here).

Learning and Generalization

A network will also produce outputs for input patterns that it was not originally set up to classify (shown with question marks), though those classifications may be incorrect.

There are two important aspects of the network's operation to consider:

Learning The network must learn decision surfaces from a set of ***training patterns*** so that these training patterns are classified correctly.

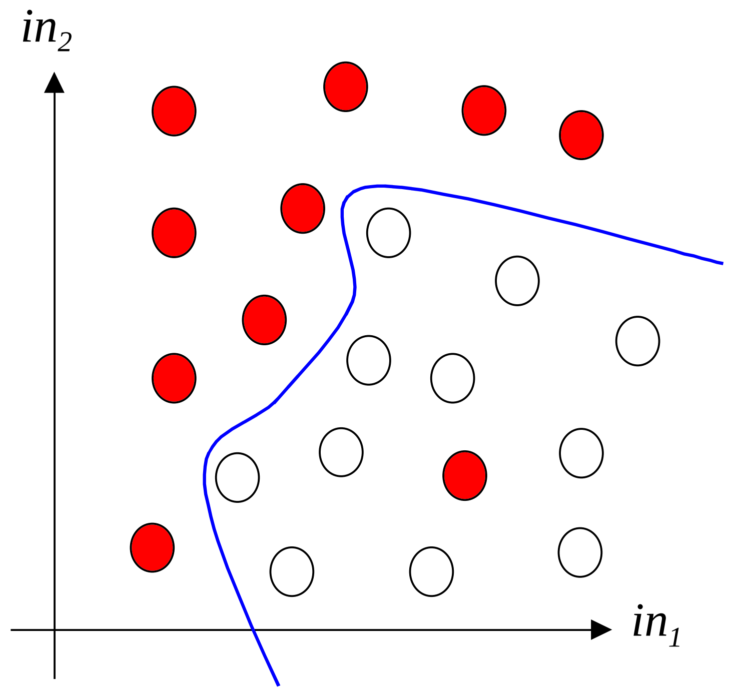
Generalization After training, the network must also be able to generalize, i.e. correctly classify ***test patterns*** it has never seen before.

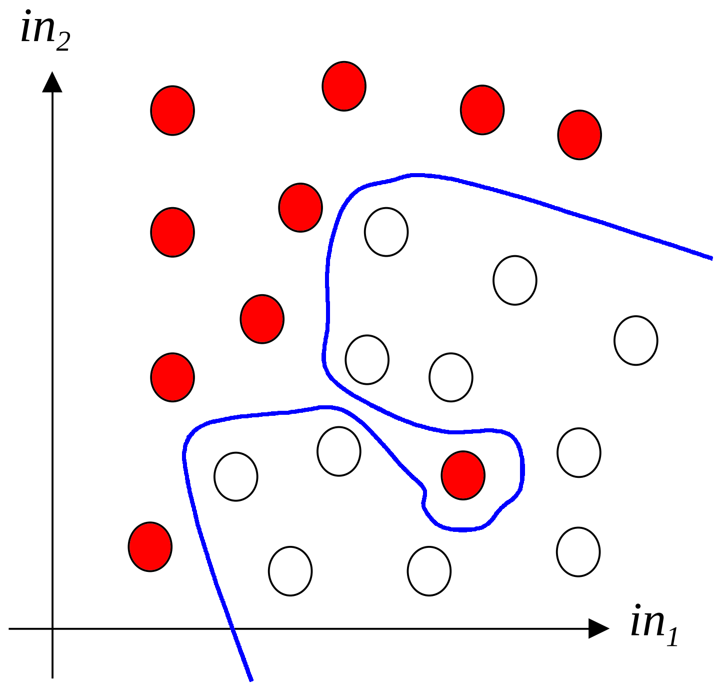
Usually we want our neural networks to learn well, and also to generalize well.

Sometimes, the training data may contain errors (e.g. noise in the experimental determination of the input values, or incorrect classifications). In this case, learning the training data perfectly may make the generalization worse. There is an important ***trade-off*** between learning and generalization that arises quite generally.

Generalization in Classification

Suppose the task of our network is to learn a classification decision boundary:

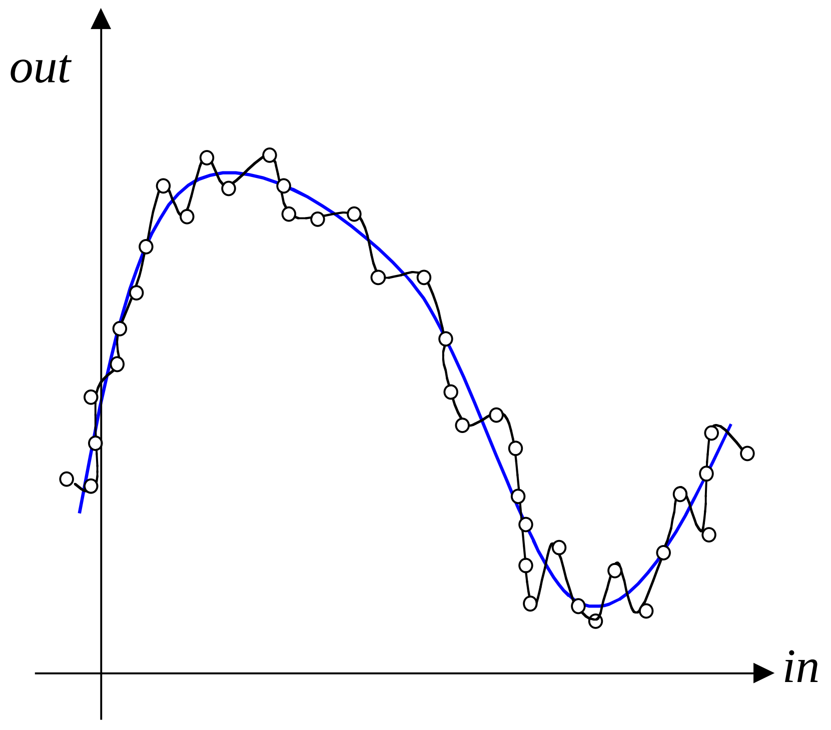




Our aim is for the network to generalize to classify new inputs appropriately. If we know that the training data contains noise, we don't necessarily want the training data to be classified totally accurately, as that is likely to reduce the generalization ability.

Generalization in Function Approximation

Suppose we wish to recover a function for which we only have noisy data samples:



We can expect the neural network output to give a better representation of the underlying function if its output curve does not pass through all the data points. Again, allowing a larger error on the training data is likely to lead to better generalization.

Training a Neural Network

Whether our neural network is a simple Perceptron, or a much more complicated multi-layer network with special activation functions, we need to develop a systematic procedure for determining appropriate connection weights.

The general procedure is to have the network ***learn*** the appropriate weights from a representative set of training data.

In all but the simplest cases, however, direct computation of the weights is intractable.

Instead, we usually start off with ***random initial weights*** and adjust them in small steps until the required outputs are produced.

We shall now look at a brute force derivation of such an ***iterative learning algorithm*** for simple Perceptrons. Then, in later lectures, we shall see how more powerful and general techniques can easily lead to learning algorithms which will work for neural networks of any specification we could possibly dream up.

Perceptron Learning

For simple Perceptrons performing classification, we have seen that the decision boundaries are hyperplanes, and we can think of ***learning*** as the process of shifting around the hyperplanes until each training pattern is classified correctly.

Somehow, we need to formalise that process of "shifting around" into a systematic algorithm that can easily be implemented on a computer.

The "shifting around" can conveniently be split up into a number of small steps.

If the network weights at time t are $w_{ij}(t)$, then the shifting process corresponds to moving them by an amount $\Delta w_{ij}(t)$ so that at time $t+1$ we have weights

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

It is convenient to treat the thresholds as weights, as discussed previously, so we don't need separate equations for them.

Formulating the Weight Changes

Suppose the target output of unit j is $targ_j$ and the actual output is $out_j = \text{sgn}(\sum in_i w_{ij})$, where in_i are the activations of the previous layer of neurons (e.g. the network inputs). Then we can just go through all the possibilities to work out an appropriate set of small weight changes, and put them into a common form:

If $out_j = targ_j$ do nothing — Note $targ_j - out_j = 0$

so $w_{ij} \rightarrow w_{ij}$

If $out_j = 1$ and $targ_j = 0$ — Note $targ_j - out_j = -1$

then $\sum in_i w_{ij}$ is too large

first when $in_i = 1$ decrease w_{ij}

so $w_{ij} \rightarrow w_{ij} - \eta = w_{ij} - \eta\, in_i$

and when $in_i = 0$ w_{ij} doesn't matter

so $w_{ij} \rightarrow w_{ij} - 0 = w_{ij} - \eta\, in_i$

so $w_{ij} \rightarrow w_{ij} - \eta\, in_i$

If $out_j = 0$ and $targ_j = 1$ Note $targ_j - out_j = 1$

then $\sum in_i w_{ij}$ is too small

first when $in_i = 1$ increase w_{ij}

so $w_{ij} \rightarrow w_{ij} + \eta = w_{ij} + \eta\, in_i$

and when $in_i = 0$ w_{ij} doesn't matter

so $w_{ij} \rightarrow w_{ij} - 0 = w_{ij} + \eta\, in_i$

so $w_{ij} \rightarrow w_{ij} + \eta\, in_i$

It has become clear that each case can be written in the form:

$$w_{ij} \rightarrow w_{ij} + \eta\, (targ_j - out_j)\, in_i$$

$$\Delta w_{ij} = \eta\, (targ_j - out_j)\, in_i$$

This weight update equation is called the ***Perceptron Learning Rule***. The positive parameter η is called the ***learning rate*** or ***step size*** – it determines how smoothly we shift the decision boundaries.

Convergence of Perceptron Learning

The weight changes Δw_{ij} need to be applied repeatedly – for each weight w_{ij} in the network, and for each training pattern in the training set. One pass through all the weights for the whole training set is called one ***epoch*** of training.

Eventually, usually after many epochs, when all the network outputs match the targets for all the training patterns, all the Δw_{ij} will be zero and the process of training will cease. We then say that the training process has ***converged*** to a solution.

It can be shown that if there does exist a possible set of weights for a Perceptron which solves the given problem correctly, then the Perceptron Learning Rule will find them in a finite number of iterations

Moreover, it can be shown that if a problem is linearly separable, then the Perceptron Learning Rule will find a set of weights in a finite number of iterations that solves the problem correctly.

Overview and Reading

1. Neural network classifiers learn decision boundaries from training data.
2. Simple Perceptrons can only cope with linearly separable problems.
3. Trained networks are expected to generalize, i.e. deal appropriately with input data they were not trained on.
4. One can train networks by iteratively updating their weights.
5. The Perceptron Learning Rule will find weights for linearly separable problems in a finite number of iterations.

Reading

1. Gurney: Sections 3.3, 3.4, 3.5, 4.1, 4.2, 4.3, 4.4.
2. Beale & Jackson: Sections 3.3, 3.4, 3.5, 3.6, 3.7
3. Callan: Sections 1.2, 1.3, 2.2, 2.3