5 Perceptrons - the first generation neural networks

Contents

5 Perceptrons - the first generation neural networks **1**
5.1 The standard paradigm for statistical pattern recognition . . . 1
5.2 Perceptron learning . . . 2
5.2.1 Using the same rule for learning the weights as that for the biases . . . 2
5.2.2 The perceptron convergence procedure: Training binary output neurons as classifiers . . . 2
5.3 A geometrical view of perceptrons learning . . . 2
5.3.1 Weight space . . . 2
5.4 Why the learning procedure works . . . 4
5.5 What perceptron can't do . . . 4

Abstract

Perceptrons were popularised by Frank Rosenblatt in the early 1960's, who appeared to have a "very powerful learning algorithm" (see section 5.2.2) and grand claims were made for what they could learn to do. Unfortunately, his data was often bias. For example, he claimed that this algorithm could distinguish between tanks and trucks after processing many learning examples. What was not realised was that all the pictures of the trucks were taking on a cloudy day, whilst the pictures of tanks were taken on a clear sunny day, and all the perceptron was doing was measuring the total intensity of all the pixels. Likewise, Minsky and Papert in 1968 published a book called "Perceptrons" that analysed that they could do and show their limitations. Many people thought these limitations applied to all neural networks (even though Minsky and Papert knew the book did not answer this).

In this chapter we describe the Perceptron, how learning proceeds, and the limitations of using perceptrons. (Useful resource `http://neuralnetworksanddeeplearning.com/chap1.html#perceptrons`.)

5.1 The standard paradigm for statistical pattern recognition

Definition 5.1. A **perceptron** is an algorithm for learning a binary classifier.

Remark. A useful way you can think about the perceptron is that it is a device that makes decisions by weighing up evidence - the more weight a particular input has indicates that that input is more important (w.r.t. the other inputs).

Note. There are actually many different kinds of perceptrons; the standard kind, typically called an **alpha perceptron** (see figure 5.2), consists of some inputs which are then converted into feature activities.

The standard paradigm for statistical pattern recognition involves:

1. Convert the raw input vector into a vector of feature activations.
   (a) "We use handwritten programs based on commonsense to define the features (so this part of the system does not learn)."
2. *Learn* how to weight each of the feature activations to get a single scalar quantity.
3. If this quantity is above some threshold, decide that the input vector is a positive example of the target class.

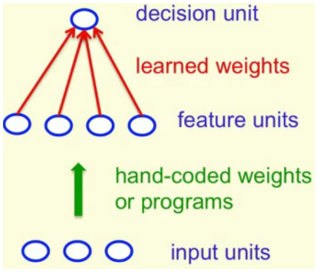

Figure 5.1: The standard perceptron architecture; a perceptron is a particular example of a statistical pattern recognition system.

5.2 Perceptron learning

5.2.1 Using the same rule for learning the weights as that for the biases

The perceptron uses binary threshold neurons (recall chapter 1) as its decision units. A threshold is equivalent to having a negative bias; that is, $y = \mathbb{I}_{[\sum_i x_i w_i \geq \theta]} \iff y = \mathbb{I}_{[z \geq 0]}$ where $b = -\theta (= w_0)$.

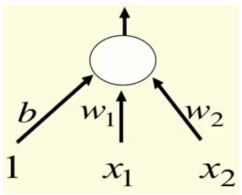

Figure 5.2: **Biases as weights**: We can avoid having to figure out a separate learning rule for the bias by using this trick - a bias is *exactly equivalent* to a weight on an extra input line that always has an activity of 1. We can now learn a bias as if it were a weight.

Remark. Biologically, the bias is a measure of how easy it is to get the perceptron to *fire*.

5.2.2 The perceptron convergence procedure: Training binary output neurons as classifiers

To train binary output neurons as classifiers, we can use the following steps:

1. **Deal with threshold:** Add an extra component with value 1 to each input vector (i.e. $\mathbf{x} = (1, \mathbf{x})$). The bias weight on this component is minus the threshold (so we can "forget" the threshold, i.e. $\mathbf{w} = (b, \mathbf{w})$).

2. Pick training cases using any policy that ensures that every training case will keep getting picked. Update the weights as follows:

   (a) If the output unit is correct, leave its weight alone.

   (b) If the output unit is incorrectly outputs a 0, add the input vector to the weight vector.

   (c) If the output unit is incorrectly outputs a 1, subtract the input vector from the weight vector.

Remark. The set of candidate hypotheses considered in perceptron learning is the set of all possible real-valued weight vectors $H = \{\mathbf{w} : \mathbf{w} \in \mathbb{R}^n\}$.

Note. This is ***guaranteed*** to find a set of weights that gets the right answer for every training case...IF ANY SUCH SET EXISTS! Unfortunately for many interesting problems, *no such set of weights exist*; this depends very much on what features you use. So for many problems, the difficult bit is deciding what features to use.

5.3 A geometrical view of perceptrons learning

We are going to start thinking about hyperplanes in high-demensional spaces; this is difficult. To deal with say, hyperplanes in a 14-dimensional space, visualise a 3-D space and say "14" to yourself very loudly (everyone does it).

Note. Going from 13-D to 14-D creates as much extra complexity as going from 2-D to 3-D.

5.3.1 Weight space

- The weight-space has 1-D per weight; a point in the space represents a particular setting of all the weights.

- Assuming that we have eliminated the threshold, each *training case* can be represented as a *hyperplane* through the origin.

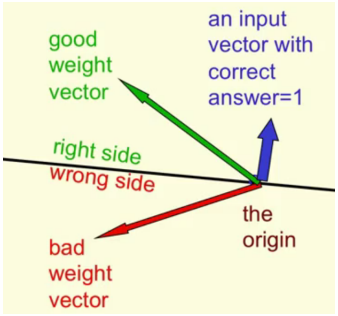

Figure 5.3: **An input vector with correct answer=1:** Each training case defines a plane (black line) - the plane goes through the origin and is perpendicular to the input vector (blue arrow). For each training case, the weights must lie on *one* side of this hyperplane to get the *answer correct*; on one side of the plane the output is wrong because the scalar product of the weight vector with the input vector, $z = \sum_i x_i w_i = \|\mathbf{x}\| \, \|\mathbf{w}\| \cos\theta$, has the wrong sign (it should be positive so that $z \geq 0 \Rightarrow y = \mathbb{I}_{[z \geq 0]} = 1$, so that the angle should be less than 90°).

Remark. We can think of the inputs as partitioning the space into two halves - weights lying in one half will get the answer correct while on the other half they will give the incorrect answer (which half is determined by the output class). That is, the inputs will *constrain* the set of weights that give the correct classification results.

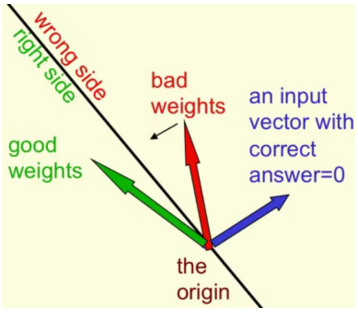

Figure 5.4: **An input vector with correct answer=0:** This time for each training case, the weights must lie on *"other"* side of the hyperplane to get the *answer correct*, because we want the *scalar product* of the weight vector with the input vector, $z = \sum_i x_i w_i = \|\mathbf{x}\| \, \|\mathbf{w}\| \cos\theta$ to be less than 0 (so $y = \mathbb{I}_{[z \geq 0]} = 0$); that is the angle should be greater than 90°, so the sign is negative.

If there are any weight vectors that get the right answer for all cases, they lie in a hyper-cone with its apex at the origin.

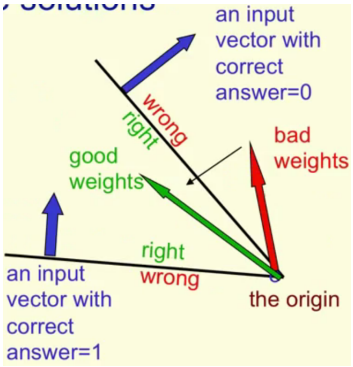

Figure 5.5: **The Cone of Feasible Solutions:** To get all training cases right, we need to find a point on the right side of all the planes - *inside the cone of feasible solution.* Of course there *may not exist* such a cone - it may be that there are no weight vectors that get the right answer for all the training cases (but if there does exist such, it'll be inside the cone).

Remark. **The problem is convex:** If we find 2 good weight vectors (that lie in the cone for all training cases), the average of them is also a good weight vector. In general machine learning problems, if you can find a *convex* problem it makes life a lot easier.

5.4 Why the learning procedure works

Definition 5.2. Generously feasible weight vectors are weight vectors that lie within the feasible region by a margin at least as great as the length of the input vector that defines each constraint plane.

Definition 5.3. The **squared distance** of a weight vector is the squared sum of all orthogonal vectors.

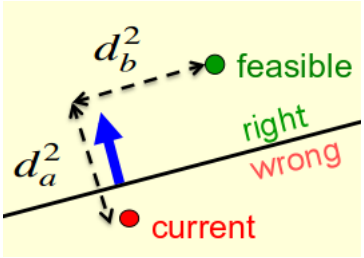

Figure 5.6: The **squared distance** of a (feasible) weight vector.

Theorem 5.1. *Every time the perceptron makes a mistake, the squared distance to all of generously feasible weight vectors is always decreased by at least the squared length of the input (or update) vector.*

Proof. (Sketch)

Each time the perceptron makes a mistake, the current weight vector moves to decrease its squared distance from every generously feasible weight vector by at least the squared length of the current input vector.

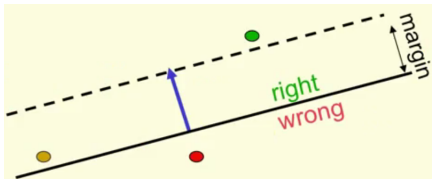

Figure 5.7: Note in this case, the gold point is not generously feasible (as it is not within the feasible region by a margin at least as great as the length of the input vector), so we omit it from the squared distance caluculation.

Assuming none of the input vectors are infinitesimally small, after a finite number of mistakes the weight vector must lie in the feasible region (IF THIS REGION EXISTS), which would stop it making mistakes. □

5.5 What perceptron can't do

- If you are allowed to choose the features by hand and if you use enough features, you can make a perceptron do almost almost anything.
- But once the hand-coded features have been determined, there are very strong limitations on what a perceptron can learn.

Example 5.1. *XOR*

Consider the task of establishing if two single bit features are the same:

Positive Cases (same)	Negative Cases (different)
$(1,1) \Rightarrow 1$	$(0,1) \Rightarrow 0$
$(0,0) \Rightarrow 1$	$(1,0) \Rightarrow 0$

Table 5.1: The 4 input-output pairs give 4 inequalities that are impossible to satisfy:

$$\left.\begin{array}{l} 1 \cdot w_1 + 1 \cdot w_2 \geq \theta \\ 0 \cdot w_1 + 0 \cdot w_2 \geq \theta \\ 0 \cdot w_1 + 1 \cdot w_2 < \theta \\ 1 \cdot w_1 + 0 \cdot w_2 < \theta \end{array}\right\} \Rightarrow \begin{array}{c} w_1 + w_2 \geq \theta \\ 0 \geq \theta \\ w_2 < \theta \\ w_1 < \theta \end{array}$$

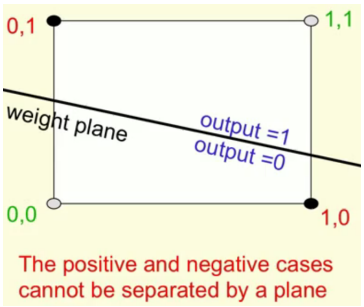

Figure 5.8: This set of training examples is not **linearly separable** (the green points cannot be separated from the red points by a straight line) - thus the Boolean function XOR cannot be represented by perceptrons.

Remark. A perceptron can implement a NAND gate (see the useful reference in the abstract), and the NAND gate is universal for computation (that is, we can build any computation up out of NAND gates). Since NAND gates are universal for computation, it follows that perceptrons are also universal for computation.

- Minsky and Papert proved that the above learning procedure converges within a finite number of iterations, *provided the training examples are linearly separable.* If the data are not linearly separable, convergence is not assured.
  - Minsky and Papert's "Group Invariance Theorem" says that the part of a perceptron that learns cannot learn to recognise patterns that have undergone transformations that form a group (such has translations with **wrap-around**).
- To deal with such transformations, a perceptron needs to use multiple feature units (*so the tricky part of pattern recognition must be solved by the hand-coded feature detectors, not the learning procedure*).

Remark. This illustrates that for neural networks to be powerful, we need them to be able to learn the feature detectors (it's not enough to just learn the weights of feature detectors) - hence second generation neural networks.