6 Linear neurons - a second generation neural network

Contents

Abstract

This chapter introduces the learning algorithm for a linear neuron. This is quite like the learning algorithm for a perceptron, but it achieves something different;

- The perception convergence procedure works by ensuring that when we change the weights, we get closer to a good set of weights.
  - This type of guarantee cannot be extended to more complex networks; *so "multi-layer" neural networks do not use perceptron learning.*
- In a linear neuron, the outputs are always getting closer to the target outputs.

(C.f this to chapter 3 of my linear regression notes from the Machine Learning course.)

6.1 The need for a new type of neuron[1]

- For more complicated networks, we want to make a small change in some weight (or bias) to create only a small corresponding change in the output from the network.
- We could use this fact to modify the weights and biases to get our network to behave more in the manner we want.

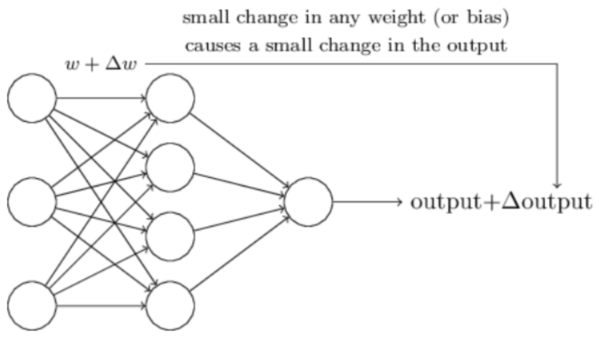
small change in any weight (or bias)
causes a small change in the output
$w + \Delta w$
output+Δoutput

Figure 6.1: We want small changes in weights to produce small changes in the output of a neural network. We can then change the weights and biases over and over to produce better and better output; the network would be learning.

Example 6.1. ***Handwriting recognition***
Suppose the network was mistakenly classifying an image as an "8" when it should be a "9". We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a "9".

The problem is that this is **not** what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way. So while your "9" might now be classified correctly, the behaviour of the

[1] `http://neuralnetworksanddeeplearning.com/chap1.html#sigmoid_neurons`

network on all the other images is likely to have completely changed in some hard-to-control way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour.

6.2 Learning the weights of a linear neuron (aka. linear filters)

The aim of learning with linear neurons is to minimise the error summed over all training cases. The error is the squared difference between the desired output and the actual output.

Notation 1. The **linear neuron** has a real-valued output which is a weighted sum of its inputs (including the bias as a weight with input value 1):

$$y = \mathbf{w}^T \mathbf{x}$$

Note. We could find $\mathbf{w}$ analytically (i.e. $A\mathbf{x} = \mathbf{b}$), however we want to develop an iterative method that can be generalised to multi-layer, non-linear neural networks.

Definition 6.1. The **residual error** is $t - y$ (i.e. the difference between the target output and the actual output).

Definition 6.2. The **delta-rule** (aka. **LMS**, **Least-Mean-Square**) for learning the weights of linear neurons is

$$\Delta w_i = \varepsilon x_i (t - y)$$

where ε :=learning rate, some positive constant to moderate the degree to which weights are changed at each step (can be chosen to make the calculations easier). It is usually set to some small value, e.g. 0.1, and sometimes made to decay as the number of weight-tuning iterations increases.

Note. By making the learning rate small enough we can get as close as we desire to the best answer; but making it too small results in slow convergence.

Remark. ***Comparison of the learning rules of percerptons and linear neurons***

- The delta-rule converges only asymptotically, but converges regardless of whether the training data are linearly separable.
- The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, *provided* the training examples are linearly separable.

6.3 Deriving the delta rule

We can define a measure for training error (which is especially convenient when using the chain rule) as

$$E(\mathbf{w}) \quad = \quad \frac{1}{2} \sum_{d \in D} (t_d - y_d)^2$$

(we characterise E as a function of $\mathbf{w}$ because the linear unit output y depends on this weight vector.). Such an function is called a **cost** (or **loss** or **objective**) **function**, and this particular function is known as the **quadratic cost function** (aka. **mean squared error** or **MSE**).

Now the batch delta rule changes the weights in proportion to the (negative, as we are decreasing to a minimum) error derivatives summed over all training cases ($d \in D$); so

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\left[\frac{1}{2}\sum_{d \in D}(t_d - y_d)^2\right] \\
&= \frac{1}{2} 2 \sum_{d \in D} (t_d - y_d) \frac{\partial}{\partial w_i}\left[(t_d - y_d)\right] \\
&= \sum_{d \in D} (t_d - y_d) \frac{\partial}{\partial w_i}\left[\left(t_d - \mathbf{w}^T \mathbf{x}_d\right)\right] \\
&= \sum_{d \in D} (t_d - y_d) \left(-(x_i)_d\right) \\
\therefore \Delta w_i &\propto -\frac{\partial E}{\partial w_i} \\
\Rightarrow \Delta w_i &= -\varepsilon \frac{\partial E}{\partial w_i} = \sum_{d \in D} \varepsilon (x_i)_d (t_d - y_d)
\end{aligned}$$

6.4 The error surface for the weights of a linear neuron

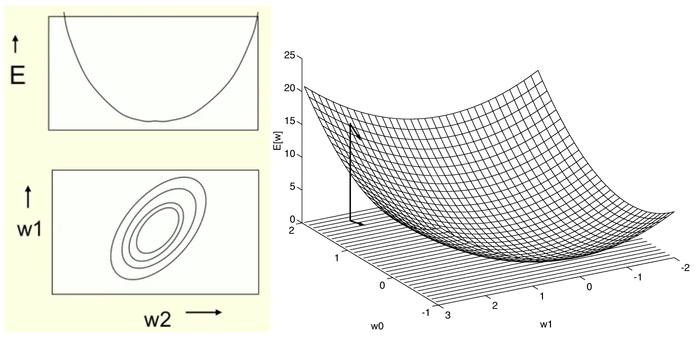

Figure 6.2: The error surface lies in a space with a horizontal axis for each weight, and one vertical axis for the error. For a linear neuron with a squared error, it is a quadratic bowl (i.e. vertical cross-sections are parabolas, whilst horizontal cross-sections are ellipses).

Remark. WARNING: For multi-layer, non-linear nets the error surface is much more complicated.

6.5 Online learning vs. batch learning

These two algorithms visit different sets of points during adaptation, however they still both converge to the same minimum.

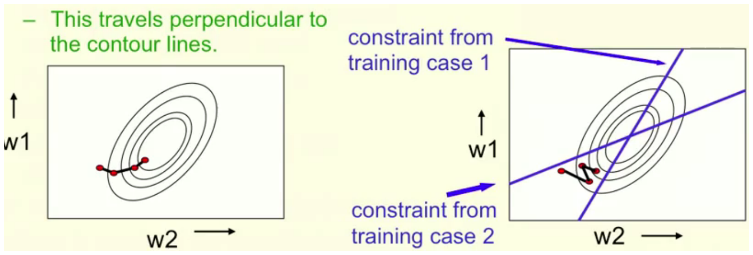
– This travels perpendicular to the contour lines.
w1
w2
constraint from training case 1
w1
constraint from training case 2
w2

Figure 6.3: **Batch (aka. offline) learning (left):** Yields a more stable descent (namely steepest descent) to a local minimum, since each update is performed based on all training cases; $\mathbf{w} \to \mathbf{w} + \Delta\mathbf{w}$, where $\Delta\mathbf{w} = -\varepsilon\nabla E(\mathbf{w})$ for the step size ε.
Online learning (right): We change the weights according to each specific training case - from the initial red dot, we move in a direction perpendicular to the constraint from training case 1, then we move in a direction perpendicular to the constraint from training case 2.

6.6 Why learning can be slow

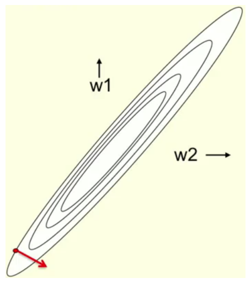

Figure 6.4: If the ellipse is very elongated, the direction of steepest descent is almost perpendicular to the direction towards the minimum; the red gradient vector has a large component along the short axis of the ellipse, and a small component along the long axis of the ellipse (just the opposite to what we want).