Too simple example: perceptron

Let's take a quick detour into something that looks like a neural network, is very simple, but exhibits a different sort of complexity than our simple neuron. (This is mostly of historical interest, to understand the evolution of the field.)

Perceptron is a supervised algorithm. Given inputs $D = \{(x_j, d_j)\}$, want classifier f.

We'll call $x_{j,i} = x_i^{(j)}$ the value of feature i in training datum j. We set $x_{j,0} \equiv 1$. This way we don't have to think about bias.

Perceptron algorithm:

$$\text{At each output, } f(x; w, b) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

- Initialise weights randomly.
- $\forall x_j \in$ inputs, compute output: $y_j(t) = f(w(t) \cdot x_j)$
- Update weights

A bit more detail on weight update:

v1

$$w(t+1) = \begin{cases} w(t) - r(t) & \text{if false positive} \\ w(t) & \text{if no error} \\ w(t) + r(t) & \text{if fail to recognise} \end{cases}$$

v2 ($d =$ desired, $y =$ observed) :

$$d(t) - y(t) = \begin{cases} -1 & \text{if false positive} \\ 0 & \text{if no error} \\ 1 & \text{if fail to recognise} \end{cases}$$

v3 $w_i(t+1) = w_i(t) + r(t) \cdot (d_j - y_j(t))x_{j,i}$ for all features i

The weight are updated at the last step after each training sample.

Converges if separable.
Note we don't need a learning rate.

A perceptron with two inputs looks like this:

$$w_1 x_1 + w_2 x_2 = T$$

(where T is a threshold). So this is a line.

Note:

It turns out that the perceptron algorithm is unstable and prone to many problems on deep or non-linear networks.

Note:

We also call the separating plane (hyperplane) a decision boundary.
The function we also call a discriminant or decision function.
At the boundary, the decision function is zero.

Recall geometry: decision boundary is perpendicular to weight vector.

Proof. Consider x_1 and x_2 on the decision boundary. Then $f(x_1; w, b) = 0$ and $f(x_2; w, b) = 0$.
So $w \cdot (x_1 - x_2) = 0$. □

Next Steps

Idea: maybe add some layers in the middle.

What would we put there?

Maybe choose not to care, call them "hidden layers".

Neuron activity at each layer must be a non-linear function of previous layer

If more than two hidden layers, then we call it deep

How do we train such a thing? We'll see that in more detail next week.

Activation function: How the neuron decides when to fire.

Things we do in the middle.

- sigmoid neurons
- ReLU neurons
- ReLU + softmax neurons
- Pooling, e.g. max (a form of down sampling)
- Dropout

Perceptron and MNIST

MNIST is the "hello world" of ML.

Talk about OvO, OvA. Introduce voting.

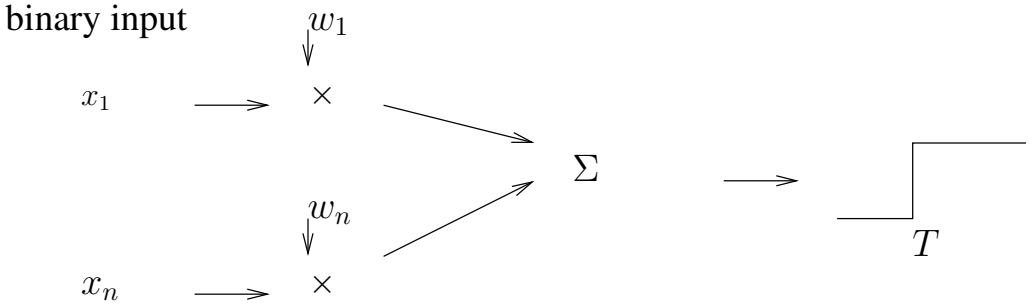
Note that many problems admit multiple solutions. The perceptron will pick one, but not necessarily the best. The "perceptron of optimum stability" (SVM) was designed to solve this problem.

ANNs don't necessarily find the best solution.

Multilayer Perceptron (MLP)

- Not perceptron
- Multiple layers
- First layer is linear
- Later layers use non-linear activation functions (typically sigmoid)
- Feedforward
- Can find non-linear separators

Training a single neuron



Let's start with a performance function $P = \|d - z\|$.
Why don't we like that?

So instead use $P = \|d - z\|^2 = (d - z)^2$.

(Draw w_1-w_2 plot with isoclines. Show four candidate steps. Mention 60 million parameters in Hinton's model. Talk about exponential explosion with number of weights.)

So we can follow the gradient:

$$\frac{\partial P}{w_1} \quad , \quad \frac{\partial P}{w_2}$$

$$\Delta w = r\left(\frac{\partial P}{w_1}\mathbf{i} + \frac{\partial P}{w_2}\mathbf{j}\right)$$

What goes wrong?

Computer scientists were stuck on this for a quarter century. Then Paul Werbos showed in his 1974 PhD dissertation how to train neural networks using back-propagation of errors.

Two problems:

1. Thresholds are annoying. We don't want $z = f(x, w, T)$ but $z = f(x, w)$.

So let's add an extra weight w_0 and input $x_0 = -1$. Then set $w_0 \equiv T$.

(Show that this moves the threshold step to 0. So we don't have to pay attention to it anymore.)

2. We're using a step function, which isn't continuous.

Let's smooth it out. Use sigmoid function.

$$\beta = \frac{1}{1 + e^{-\alpha}}$$

(Walk through how to graph this.)

Summary: What have we done ?

- Built a neuron
- Described how to do gradient descent
- Modified our activation function to be compatible with gradient descent