Supervised learning, perceptrons and the cerebellum

Dr Amelia Burroughs

Learning Objectives

- Define supervised learning and how it differs from unsupervised learning

- Understand the aim of using neural networks for pattern classification

- Describe how a set of examples of stimuli and correct responses can be used to train an artificial neural network to respond correctly
  - via changes in synaptic weights

- Describe how the delta learning rule is used to perform pattern recognition in a perceptron

- Discuss the limitation of the perceptron

- Discuss the use of multi-layered networks to overcome the limitations of the perceptron, and possible applications of such networks.

Supervised learning

Supervised learning is where the agent receives information about how well it performed

- inputs are associated with a correct output and the network receives some degree of supervision to reach this output.

This is different to unsupervised learning (eg. Hopfield networks and models of CA3 In the hippocampus)

Supervised learning

- the network receives varying degrees of supervision:
- fully supervised (each example includes correct output)
- occasional reward or punishment (reinforcement learning)
- evolution / genetic algorithms

The **perceptron** is a machine that performs supervised learning.

Rosenblatt’s Perceptron

The perceptron is a machine that does supervised learning:

- it makes guesses, it is then told whether or not its guess is correct, and then makes another guess.

They were first discovered in 1957 by Frank Rosenblatt

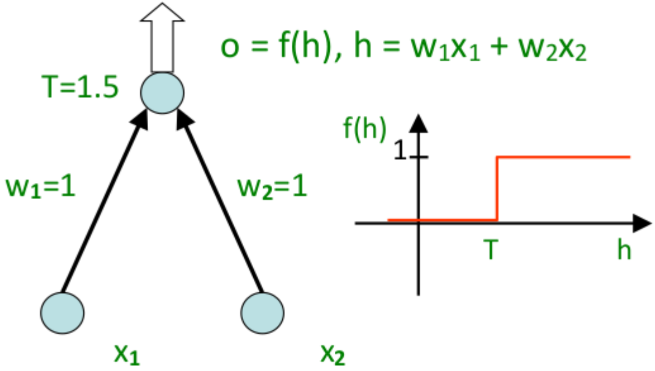
- it was claimed that they would solve problems from object recognition to consciousness
- they can be considered as the forbearer of deep learning, but the original perceptron proved quite limited in artificial intelligence.

It does, however, appear to describe some neuronal processes, if we ignore the implementation details.

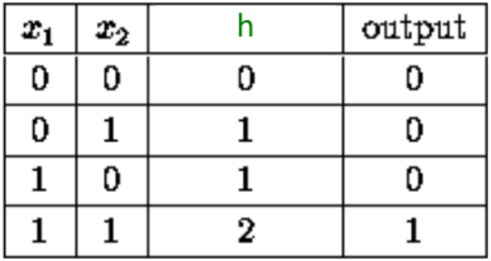
The Perceptron

A perceptron is made of two layers of neurons: an input layer and an output layer of McCulloch-Pitts neurons.

For simplicity let’s assume the output layer has a single neuron:



Consider the network with weights w_1 and $w_2 = 1$, threshold T=1.5 and a threshold logic transfer function.

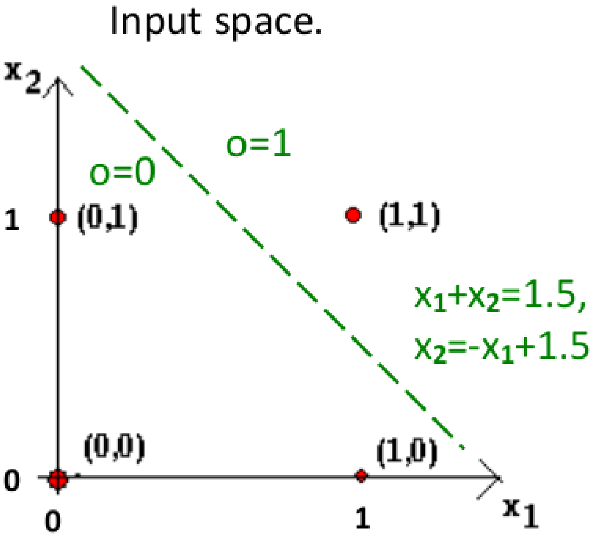


x_1	x_2	h	output
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

It performs the logical ‘AND’ function

Linear classifier

This enables the perceptron to perform a linear separation of categories:



The line separating o=1 from o=0 is where the net input equals the threshold (T=1.5 In the previous example).

Changing the weights changes the line.

Therefore, learning to classify two groups of input patterns means finding weights that separates the two groups!

Perceptron as a linear classifier

Thus, the perceptron works only if there is a hyperplane dividing the data into two

- with one class of data on one side and one on the other.

If the data is linearly separable the perceptron is guaranteed to converge to a solution which manages this separation. But, there will not be a unique hyperplane separating the two classes:

The perceptron won't, typically, find what you might regard as the 'best' hyperplane (where by best you might mean the line that is, in some sense, as far from the individual data points as possible; finding that best hyperplane is the idea behind the support vector machine).

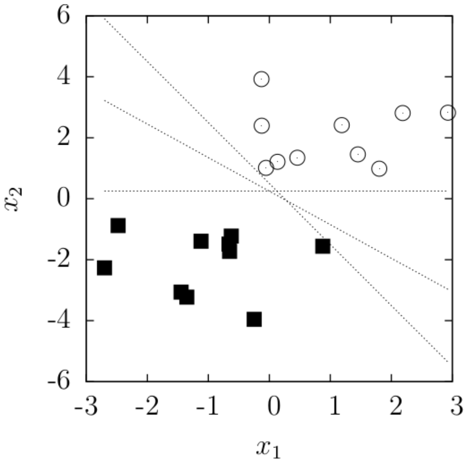


Figure 2: All of these lines separate the points into two classes

Perceptron learning rule

Supervised Learning rule: **the 'delta rule'**

In supervised learning, each input is associated with a correct output.

For a given input, if the output from the perceptron is y, and the actual value of the output should be d, the error in the output is d − y.

The error is extended to (d_i^n – y_i^n) for i neurons and n patterns.

The term (d_i^n – y_i^n) is also known as the delta δ_i^n , which is the error made by the output i for input pattern n.

The perceptron learning rule is to change the w_j weight by an amount proportional to the error and how much x_j was 'to blame' for the error

The perceptron learning rule can be motivated by thinking about an error minimisation – we will come back to this.

Perceptron learning rule

Supervised Learning rule: **the ‘delta rule’**

1) Given a training set of inputs x^n (x_1^n, x_2^n, x_3^n) with target output values d^n (d_1^n, d_2^n, d_3^n), where n=1,2,3…

2) Present the input pattern n, and find the corresponding output values y^n (y_1^n, y_2^n, y_3^n)

3) Change the connection weights according to the delta rule:

$$\delta w_i = \eta(d_i^n - y_i^n)x_i \qquad \text{where } \eta\text{=learning rate and } w_i^{new} \rightarrow w_i^{old} + \delta w_i$$

In this way, if x_j was positive and y was too big, this would make w_j smaller so in future y would be smaller for the same input.

4) Present the next input pattern n ⟶ n+1

Rosenblatt’s perceptron 1962

1) Present a set of input patterns X^n

- some of these patterns are to be detected with target $d^n = 1$
- others are ‘foils’ to be ignored with target $d^n=0$

2) The ‘delta’ rule is applied to each pattern:

$$\delta w_i = -\eta(d_i^n - y_i^n)x_i \qquad \text{and } w_i^{new} \rightarrow w_i^{old} + \delta w_i$$

3) After many presentations of the entire training set of patterns (in some random order) the perceptron can find the best linear discriminator of target patterns from foil patterns

Notice that w_i only changes if $d^n \neq y^n$ and $x_j^n \neq 0$.

If $d^n > y^n$ then w_j increases; if $d^n < y^n$ then w_j decreases, ie. the delta rule changes the weights so as to **reduce the error**

Threshold values

The problem: The delta rule only learns weights

- how do we find the value for the threshold, T ?

The solution: You can do the same thing you do to the weights for the threshold values.

This is a more modern approach to the perceptron rule and this is why these days smooth (or mostly-smooth) activation functions are used in artificial neurons.

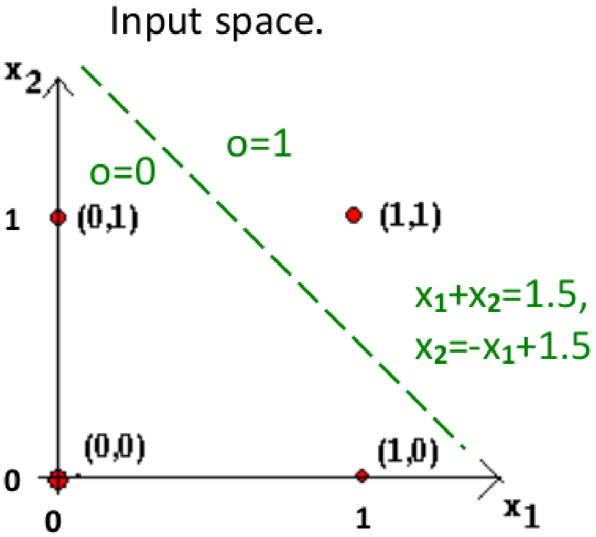
Just use T=0 and add another input x_0 = -1

- then the weight w_0 can signal the threshold value: the condition for the output to be active: $w_1x_1 + w_2x_2 + \dots > T$ is the same as:

$w_0x_0 + w_1x_1 + w_2x_2 + \dots > 0$ if $x_0 = -1$ and $w_0 = T$

$\delta\theta = \eta(d - y)$ and $\theta \rightarrow \theta + \delta\theta$

Threshold values



The line separating o=1 from o=0 is where the net input equals the threshold (T=1.5 In the previous example).

Changing the weights changes the line.

The delta rule finds the connection weight w_0 (which is the same as finding T).

Note about outputs: In reality perceptrons can have many output units (forming a single-layer neural network) – each output is trained using the delta rule independently of the others.

Limitations

The basic limitation of the perceptron is that it has only one input layer and one output layer

- this means it can only solve linearly separable problems

Hidden processing units (extra layers) can offer more processing power

These days artificial neural networks have more than one layer

- this complicates the idea of adjusting the w_i in a way that is weighted by x_i (which adjusts weights according to how much it is 'to blame' for the error)

Back propagation resolves this

- the error essentially propagates backwards from layer to layer

Another way to think of it is as doing gradient descent on all the weights.

At the moment the back propagation algorithm is not considered very biological, though it is very possible that when we understand why deep learning networks are so effective it will be clear that the important aspects of the algorithm can be recognized in neuronal dynamics.

Multi-layer perceptrons

The problem: How do you train the connection weights in a multi-layer perceptron if the ‘internal representation’ is not known

- i.e. d_i^n is not known for the ‘hidden layer’ and x_j^n is not known for the output?

The solution: If the neurons use a smooth (continous) transfer function then changing a connection weight from any (active) neuron anywhere in the network will change the output firing rate slightly.

- if the output y_i^n is now closer to the target value d_i^n then the change was good…

… But! The effect of changing a connection weight depends on the values of the errors ($d_i^n – y_i^n$), connection weights w_{ij} and activations further up the network rather than each layer.

Error back propagation

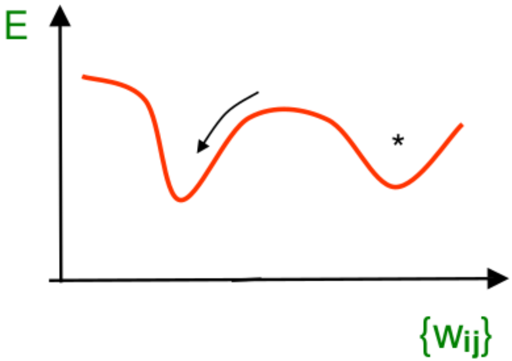
Error back-propagation is also known as the generalised delta rule.

It works like this:

1) Present input pattern n and work out all other activations (a ‘forward’ pass)
2) Work out the errors made by the outputs; $\delta_i^n = (d_i^n – y_i^n)$
3) Work out the contributions of units in layers lower down to output errors (a ‘backward’ pass). This depends on connections w ij to output, giving a ‘delta’ for that layer: $\delta_j^{n2} = \sum_i w_{ij} \delta_i^n$.
4) Similarly, for the next layer: $\delta_k^{n3} = \sum_j w_{jk} \delta_j^{n2}$.
5) Change connection weights in each layer using the delta rule.
6) Repeat for next pattern.
7) Repeat for whole training set many times.

This rule is equivalent to working out the total squared error on the whole set of training patterns: $E = \sum_n \sum_i (d_i^n – y_i^n)^2$ and changing each weight to reduce this.

Problems with error back-propagation



1) Local minima in error: you can’t be sure it will find the best set of weights, maybe just finds those that can’t be improved by any small change

2) Learning and generalisation depends on the choice of architecture

3) Not biologically plausible since the connection weights are changed using non-local information

However, with enough neurons and enough training data (and recent increases in processing power), these networks outperform other methods in some tasks.

How may the brain work as a perceptron?

Sensorimotor control model in the cerebellum

The problem: to generate a motor signal to achieve a desired state from given current state (e.g. play a stroke in tennis).

The solution:

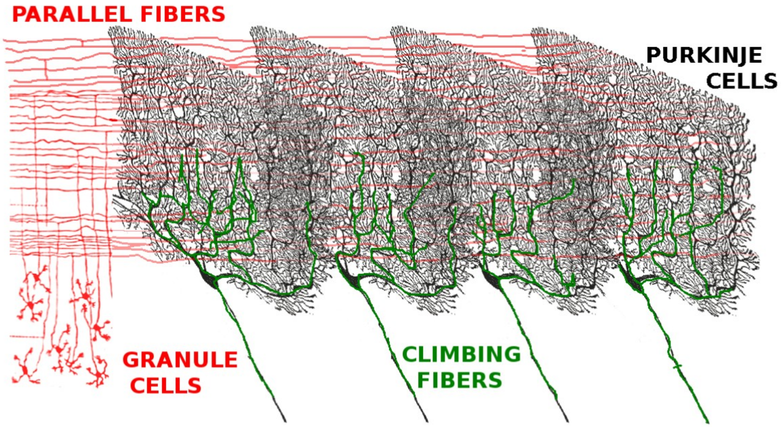
1. Train the upper network to predict the (sensory) effect of an action (motor) signal (i.e. a forward model).

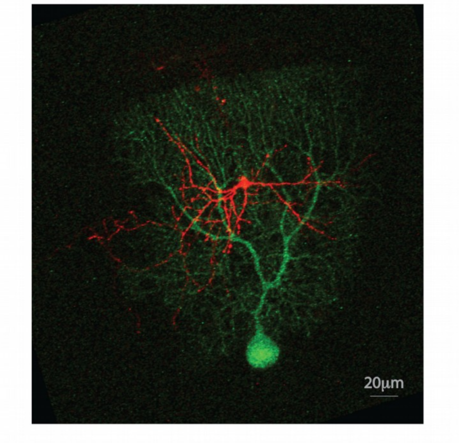
2. Train the lower network to produce the correct action (motor) signal to achieve a desired sensory effect (i.e. an inverse model). Fix the connection weights in the upper network to use it as feedback controller. Use back-propagation of the error (difference between desired and observed effect) to change weights in the lower network.

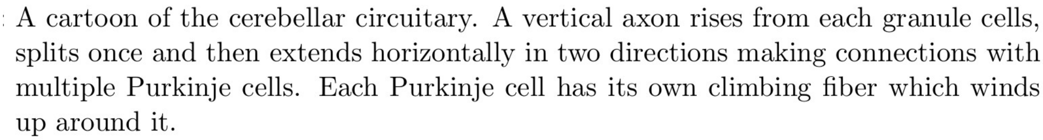
The cerebellum

The circuitry within the cerebellum can be thought of as a perceptron and can generate desired movements via the back-propagation of errors.

The anatomy of the cerebellum

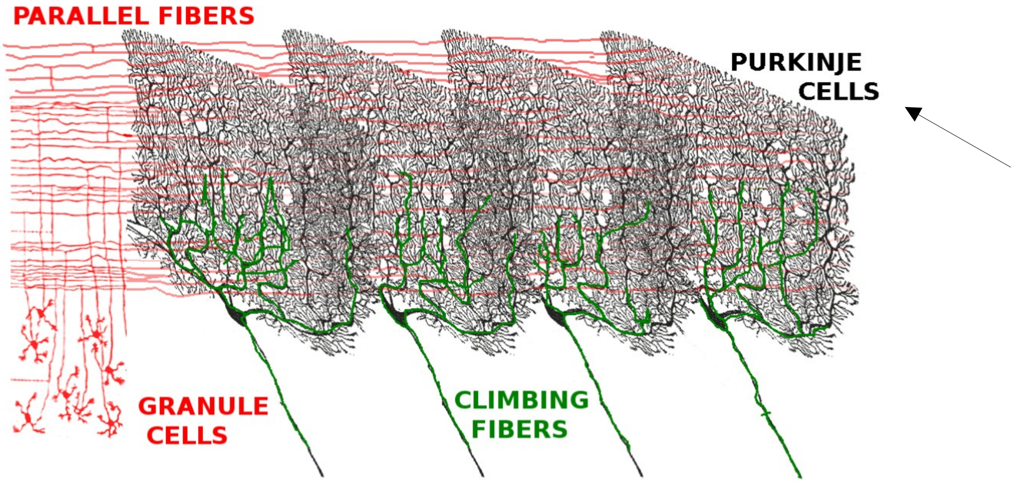





A cartoon of the cerebellar circuitary. A vertical axon rises from each granule cells, splits once and then extends horizontally in two directions making connections with multiple Purkinje cells. Each Purkinje cell has its own climbing fiber which winds up around it.

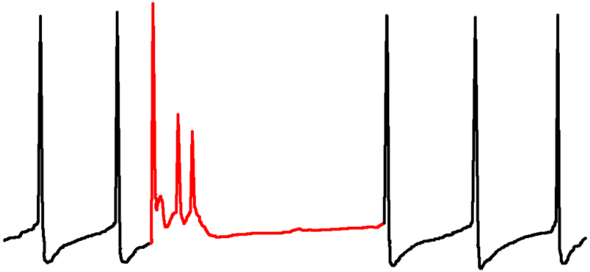
Purkinje cells

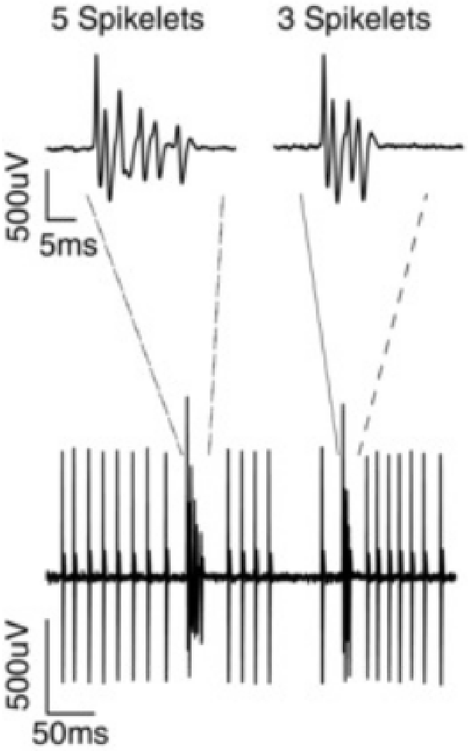
Purkinje cells receive two excitatory inputs, **weak inputs from parallel fibres**, axons that run perpendicular to the planes of the Purkinje cell dendritic arbors, and a **strong input from a climbing fibre**, a single axon which winds around the Purkinje cell and makes multiple contacts with it.



Purkinje cells are inhibitory

Purkinje cell spiking





A Purkinje cell complex spike is shown in red. Simple spikes (normal action potentials) are shown in black.

A purkinje cell complex spike has a long duration and consists of low amplitude secondary spikes called spikelets.

They are also followed by a long refractory period (~50ms).

The cerebellum as a perceptron

Most ideas about cerebellar function believe it is implicated in the **calculation of motor signals**, and/or for **predicting** the sensory or proprioceptive consequences of motor actions and **correcting errors** in these signals by improving motor behaviour.

The Marr-Albus Model

1) Connections from the parallel fibres to Purkinje cells may act like a perceptron (they form the input and output layers)

2) Climbing fibres carry an error signal that computes how far away a movement was from the desired movement (d-y)

The cerebellum as a perceptron

If y is the output from the Purkinje cell, then, taking into account the fact that **Purkinje cells are inhibitory**:

$$y = -\sum w_i x_i$$

where the **x_is are the activities in the parallel fibers** and w_i is the strength of the synapse from the ith parallel fiber to the Purkinje cell.

According to the perceptron learning rule there is a desired output, d, and the synapses are adjusted according to:

$$\Delta w_i = -\eta(d - y)x_i$$

where η is a small learning rate.

The idea in the Marr-Albus model is that the **climbing fiber carries the error** signal d − y.

The cerebellum as a perceptron

Let’s have a simple example,

say w = (1, 1, 1, 1) initially, and the input x = (1, 0, 1, 0) is supposed to have the output d = −1, then:

$$y = -\mathbf{w} \cdot \mathbf{x} = -2$$

so $d - y = 1$ and $\Delta w = -\eta(1, 0, 1, 0)$ and if $\eta = 0.25$

(remember that: $w_i^{new} \rightarrow w_i^{old} + \delta w_i$)

after learning we would have w = (0.75, 1, 0.75, 1), so:

$$y = -\mathbf{w} \cdot \mathbf{x} = -1.5$$

and the error has fallen to $d - y = 0.5$.

The cerebellum as a perceptron

Conversely, imagine w = (1, 1, 1, 1) but x = (1, 1, 0, 0) is intended to represent d = −3, here:

$$y = -\mathbf{w} \cdot \mathbf{x} = -2$$

so d − y = −1 and Δw = −η(−1, −1, 0, 0) and after learning with η = 0.25 we would have w = (1.25, 1.25, 1, 1).

We therefore need both positive and negative errors with positive errors associated with synaptic depression and negative errors with synaptic potentiation

In experiments, climbing fibre activity greater than or less than the average corresponds to decreases and increases in synapse strengths.

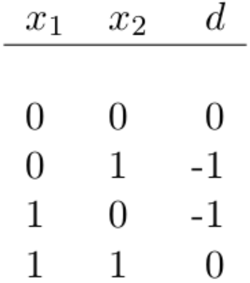
More complex spikes = less simple spikes

But this is too simple!

Synapses form the parallel fibers can only be positive, not negative.

We need to account for this by including inhibitory cells in the network.

Furthermore, a linear model like that being used here can’t learn complicated patterns like the XOR pattern:



x_1	x_2	d
0	0	0
0	1	-1
1	0	-1
1	1	0

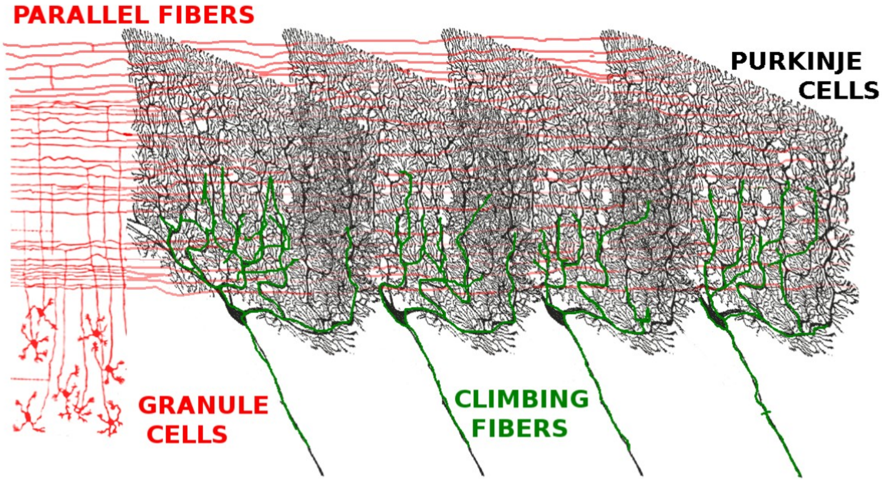
Learning a pattern like this requires a network with more than one layer.

In the cerebellum this is provided by the granule cell layer.

The rest of the cerebellar circuitry

Each granule cell receives input from between one and seven mossy fibers.

There are 3 × 10^{11} granule cells, roughly 100 to 150 times the number of mossy fibers.



Finally there are large inhibitory cells called Golgi cells in the network, these have long time constants, which provides delays

- this is very useful for motor control where different muscles move at different times, or in cases where different motor consequence unfold at different times during a movement.

Conclusions

Perceptrons are an example of a network that performs supervised learning

- they perform very well in linear separation

Perceptrons learn using a delta, or error back propagation, rule:

$$\Delta w_i = \eta(d_i^n - y_i^n)x_i$$

The cerebellum can be seen to work as a perceptron

- parallel fibres provide the input
- Purkinje cells provide the inhibitory output
- these are connected with a dynamic synapse with weight, w
- climbing fibers provide an error signal

But we need more layers to perform more difficult motor sequences