CS5010

Lecture 6, Neural Networks, Chapter 20 section 5

November 29, 2021

Contents

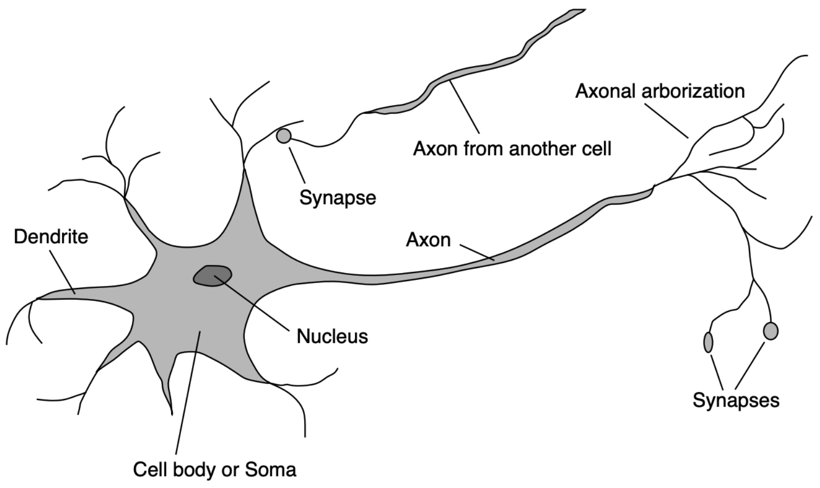
1 Outline

- Brains
- Neural Networks
- Perceptrons
- Multilayer Perceptrons
- Applications of neural networks

2 Brains

Neural networks are **very loosely** inspired by the structure of the brain.

$10^1$1 neurons of > 20 types, $10^1$4 synapses, 1ms - 10ms cycle time. Signals are noisy "spike trains" of electrical potential.



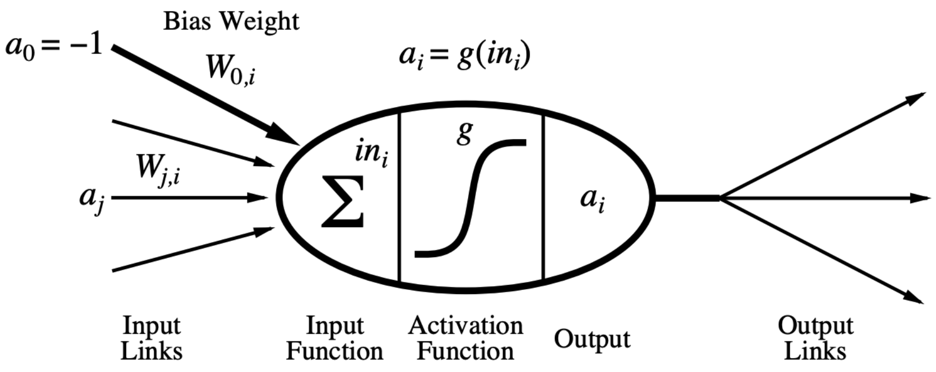
Dendrites correspond to the input weights of an artificial neuron and the one axon is the output in this comparison. A unit combines many inputs into one output in this case.

3 Neural Networks

3.1 McCulloch-Pitts "unit"

There are many different artificial neurons. This is one of the first ones "introduced". Output is a "squashed" linear function of the inputs:

$$a_i \leftarrow g(in_i) = g(\sum_j W_{j,i} a_j)$$



i = 1:N is the number of units in the net

j = 1:M is the number of input sources

- Greater weight is applied to the inputs which are more important.
- Inputs are weighted and combined via the input function. Typically using matrix multiplication (NNs all about linear algebra) and then this combination is passed to some activation function g.

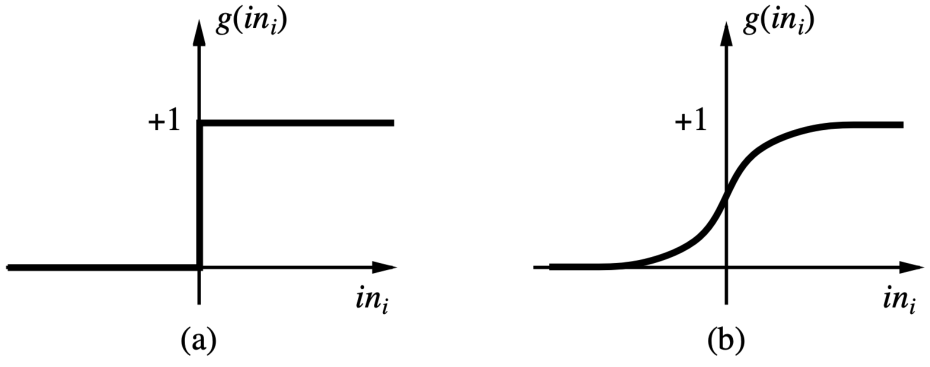
A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do.

3.2 Activation functions

In the previous diagram, g is our activation function and this can take different forms.

(a) is a step function or threshold function. $g(in_i)$ is zero for all $in_i < 0$ and 1 otherwise.

(b) is a sigmoid function $1/(1 + e^{-x})$. Hyperbolic function tanh is also often used.

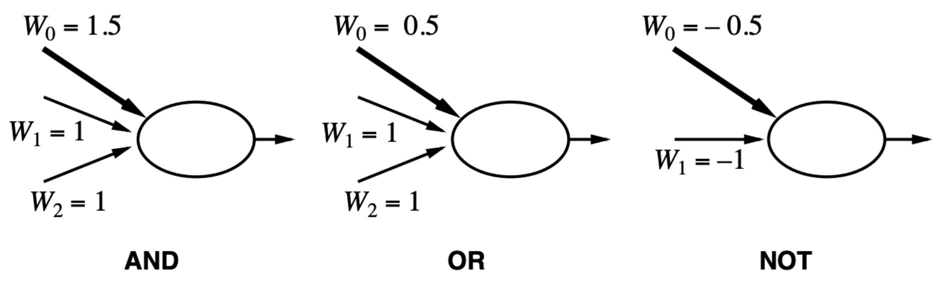


Changing the bias weight $W_{0,i}$ moves the threshold location.

Recall that $y = mx + c$ as a simple regression model, where m is the slope or rate of change of the prediction model and c moves the threshold location - it is a bias term.

- Sigmoid function maps all inputs into the range (0,1) and is comparable to a smooth step function.
- In modern NNs, there are a vast number of activation functions that can be used.
- In the step function, the output is either on or off. In the sigmoid it is neither fully on or off. In both cases the input weights are adjusted to affect the output value of the function.

3.3 Implementing logical functions



McCulloch and Pitts: every Boolean function can be implemented

Both arguments and results are from a two-valued set, normally denoted $\{0, 1\}$ or $\{T, F\}$

OA: correct information but not massively relevant and requires a tremendous amount of background knowledge.

3.4 Network structures

There are many types of network and neuron. OA advises that the key concept here is that the networks are constructed of multiple neurons. This combination of simple neurons connected in a clever way allows us to perform many different complicated tasks.

Feed-forward networks:

- single-layer perceptrons
- multi-layer perceptrons

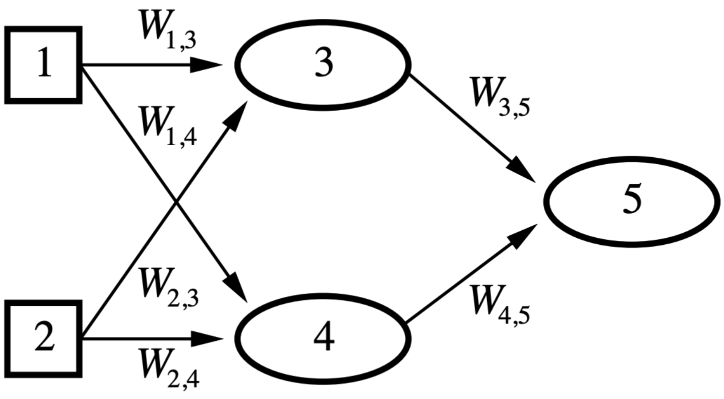
Feed forward networks implement functions and have no internal state. Feed forward networks are arranged into layer of neurons. There is information flow from one layer to the next. First layer is your input, then to second layer and so on, the last layer is the output layer. Information does not move back to previous layers, it only goes from one layer to the next.

Recurrent networks (networks that have loops, certain neurons have outputs which feed back into being inputs for the same neuron):

- Hopfield networks have symmetric weights ($W_{i,j} = W_{i,j}$), $g(x) = sign(x)$, $a_i = \pm 1$; holographic associative memory
- Boltzmann machines use stochastic activation functions, $\approx$ MCMC in Bayes nets
- recurrent neural nets have directed cycles with delays $\implies$ have internal state (like flip-flops), can oscillate etc.

Due to their looping nature, recurrent networks are difficult to analyse mathematically.

3.5 Feed-forward example



Feed-forward network = a parameterized family of non-linear functions. Using the above diagram (where a_n corresponds to the output at node n) and working backwards from the output node, we find:

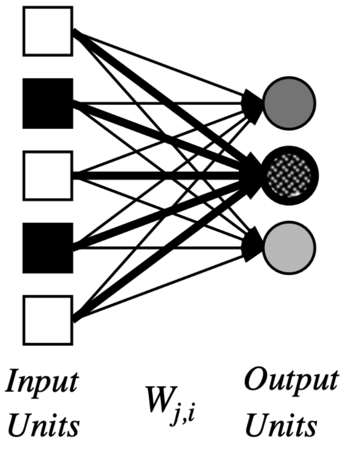
$$\begin{aligned} a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\ &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \end{aligned}$$

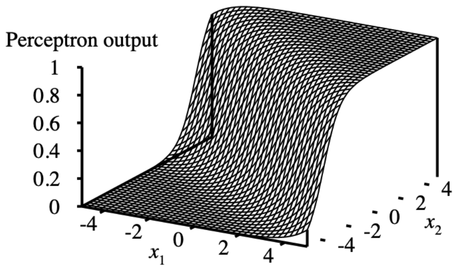
Constrained by the number of neurons and layers but by adjusting weights we can change the function: do learning this way!

As before, g is the activation function. This example has no bias term

The bias term is a weight that can be added at each layer in addition to the weighted sum of the inputs. It serves the same function as the intercept added in a linear equation.

4 Single-layer perceptrons





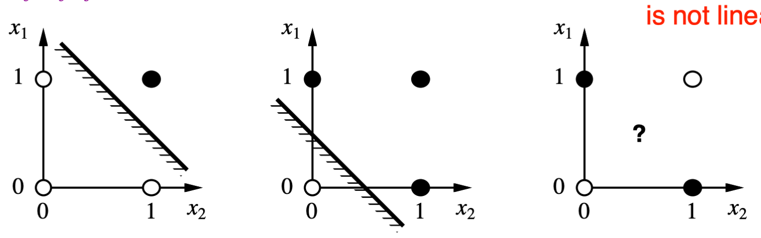
Output units all operate separately - no shared weights Adjusting weights moves the location, orientation, and steepness of cliff. Put simply, adjusting the weights is adjusting their importance.

4.1 Expressiveness of perceptrons

Consider a perceptron with g = step function (Rosenblatt, 1957, 1960). Can represent AND, OR, NOT, majority, etc., but not XOR. Represents a linear separator in input space:

$\Sigma_j W_j x_j > 0$ or $\mathbf{W} \cdot \mathbf{x} > 0$

We say that the XOR space is not linearly separable



(a) x_1 and x_2 (b) x_1 or x_2 (c) x_1 xor x_2

This was originally a problem when neural networks were first discovered/used. We can't represent XOR with only one layer with a step function. This is a constraint of the step function and it's one of the reasons why in more recent years that different activation functions are used. With a smooth (differentiable) function we can make small adjustments.

Minsky & Papert (1969) pricked the neural network balloon.

Most Boolean functions are not linearly separable, so these networks have very limited applicability

4.2 Perceptron learning

Learn by adjusting weights to reduce error on training set.

The squared error for an example with input x and true output y is

$$E = \frac{1}{2}Err^2 \equiv \frac{1}{2}(y - h_{\mathbf{W}}(\mathbf{x}))^2$$

h is a function that combines weights and activation to produce outputs from inputs x

We square the error so that the error function is a quadratic curve for which the minimum will be where the derivative is zero. Also by squaring, when we differentiate we get a linear function which we can then solve

If we consider a Taylor or Maclaurin series we know that every function in a certain locality can be represented by a linear term (straight line approximation). This is a quadratic approximation to some complex function.

We want to minimise the error with respect to the weights (we can change the weights, the other aspects such as the input and correct answer are fixed).

Perform optimisation search by gradient descent. We differentiate the error with respect to one of the weights (using the chain rule):

$$\frac{\partial E}{\partial W_j} = Err \times \frac{\partial Err}{\partial W_j} = Err \times \frac{\partial}{\partial W_j}(y - g(\sum_{j=0}^{n} W_j x_j))$$

$$= -Err \times g^{'}(in) \times x_j$$

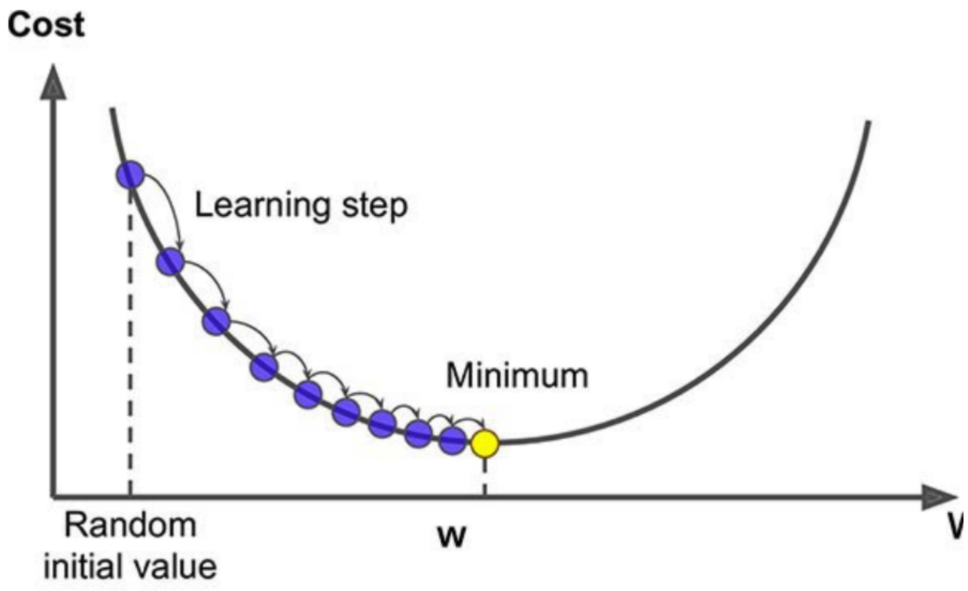
Simple weight update rule:

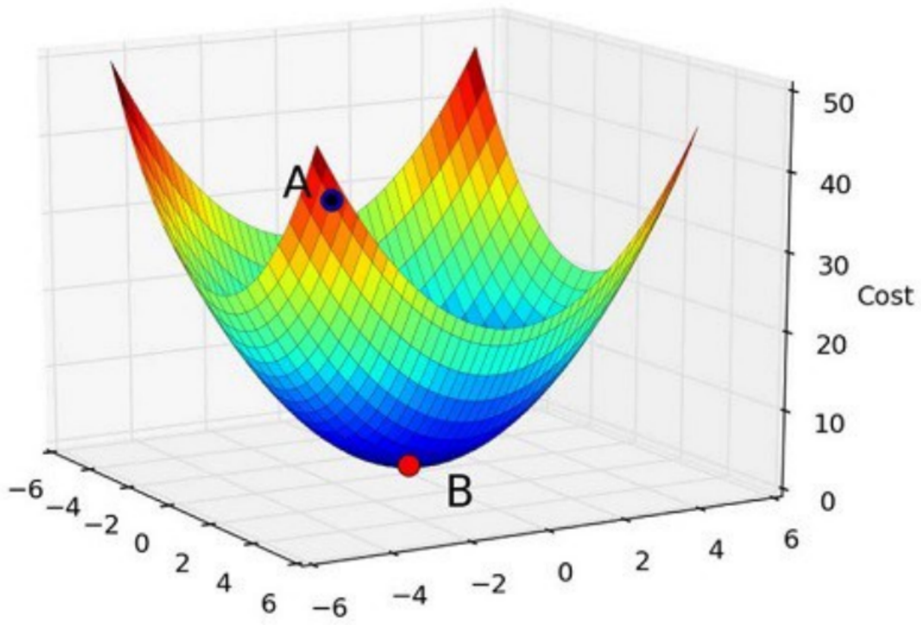
$$W_j \leftarrow W_j + \alpha \times Err \times g^{'}(in) \times x_j$$

where α is the learning rate and dictates how big our steps are for each iteration of gradient descent. Oggie makes the analogy of getting in the shower and turning the tap to find the right temperature, the learning rate here would be how much you rotate the tap each time.

E.g., +ve error $\Longrightarrow$ increase network output $\Longrightarrow$ increase weights on +ve inputs, decrease weights on -ve inputs

4.3 Gradient descent

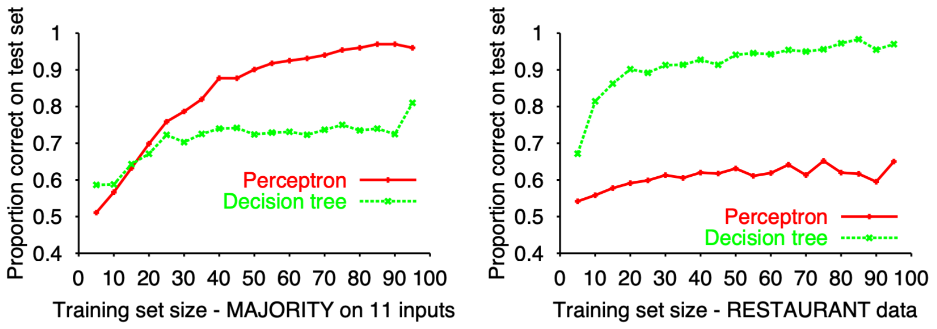




Shows the steps and goal of gradient descent in 2 and 3 dimensions respectively.

4.4 Perceptron learning continued

Perceptron learning rule converge to a consisten function for any linearly seperable data set.

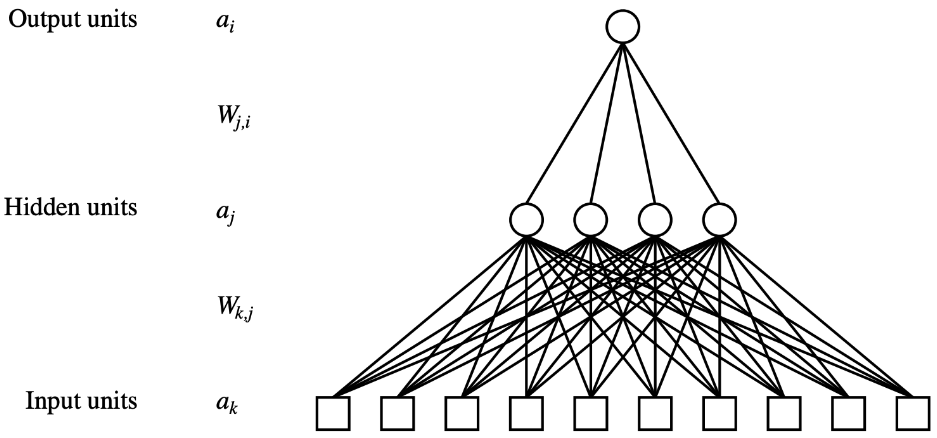


Perceptron learns majority function easily, Decision tree learning (DTL) is hopeless. DTL learns restaurant function easily, perceptron cannot represent it.

There is no such thing as a superior learner, different algorithms perform better given the circumstances.

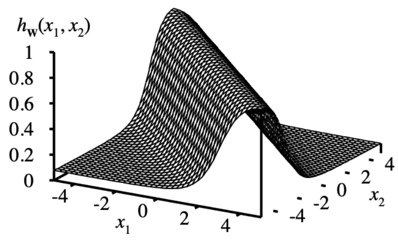
5 Multilayer perceptrons

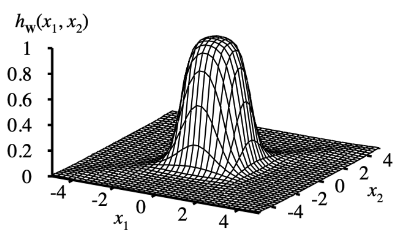
Layers are usually fully connected; numbers of hidden units typically chosen by hand.



5.1 Expressiveness of MLPs

All continuous functions w/ 2 layers, all functions w/ 3 layers. Our constraint here is the number of the layers.





Combine two opposite-facing threshold functions to make a ridge.

Combine two perpendicular ridges to make a bump.

Add bumps of various sizes and locations to fit any surface.

Proof requires exponentially many hidden units (cf DTL proof)

5.2 Back-propagation learning

This is extending the learning process and is the means by which we adjust the weights in a perceptron. We propagate the error between our output and the correct answer in our training data back through the network and use this to adjust the weights at each stage. All being well, this minimises the error in subsequent feed-forward iterations.

Output layer: same as for single-layer perceptron,

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

where $\Delta_i = Err_i \times g^{'}(in_i)$

Hidden layer: back-propagate the error from the output layer (proportionally across the neurons in the previous layer):

$$\Delta_j = g^{'}(in_j) \sum_i W_{j,i} \Delta_i$$

Update rule for weights in hidden layer:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

(Most neuroscientists deny that back-propagation occurs in the brain)

5.3 Back-propagation derivation

Repeated application of differentiation and chain rule.

The squared error on a single example is defined as

$$E = \frac{1}{2}\sum_i (y_i - a_i)^2$$

Where the sum is over the nodes in the output layer.

$$\frac{\partial E}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial a_i}{\partial W_{j,i}} = -(y_i - a_i)\frac{\partial g(in_i)}{\partial W_{j,i}}$$

$$= -(y_i - a_i)g^{'}(in_i)\frac{\partial in_i}{\partial W_{j,i}} = -(y_i - a_i)g^{'}(in_i)\frac{\partial}{\partial W_{j,i}}(\sum_j W_{j,i}a_j)$$

$$= -(y_i - a_i)g^{'}(in_i)a_j = -a_j\Delta_i$$

The analogy to this is we're standing on the side of a foggy mountain and need to try and guess where the peak is. We can determine the slope in our immediate vicinity and guess that the peak is likely to be up the hill along the steepest slope. This is effectively what back propagation is doing.

Same idea: quadratic error surface and find weights where error is (close to) zero

The hidden layers make this difficult: our input data doesn't tell us what the values at internal nodes should be, only the values at the output layer

The below is taking the error from the final layer and distributing it back across all of the previous layers and nodes proportional to their weights.

$$\frac{\partial E}{\partial W_{k,j}} = -\sum_i (y_i - a_i)\frac{\partial g(in_i)}{\partial W_{k,j}}$$

$$= -\sum_i (y_i - a_i)g^{'}(in_i)\frac{\partial in_i}{\partial W_{k,j}} = -\sum_i \Delta_i \frac{\partial}{\partial W_{k,j}}(\sum_j W_{j,i}a_j)$$

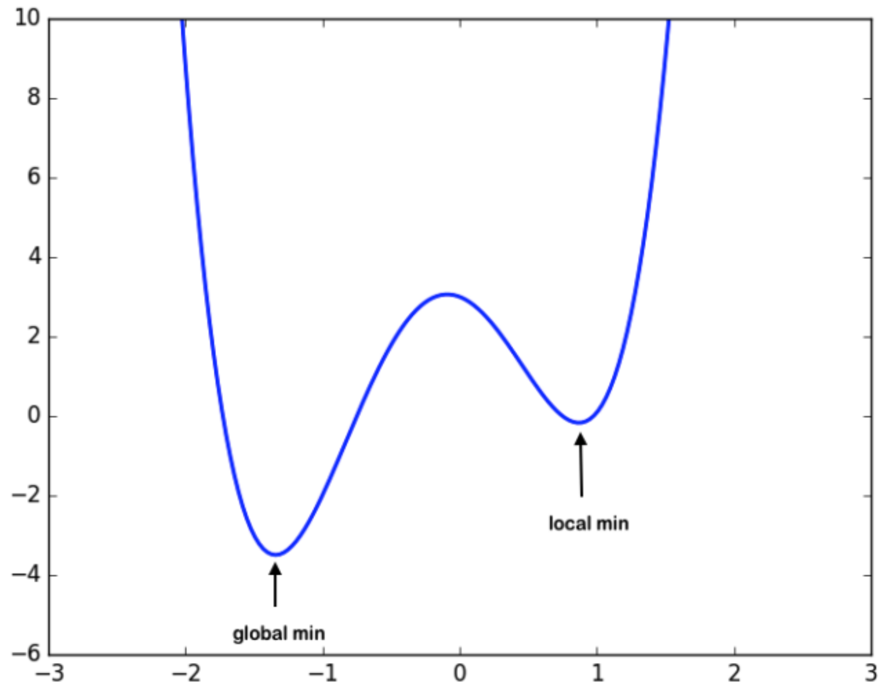
$$= -\sum_i \Delta_i W_{j,i} \frac{\partial a_j}{\partial W_{k,j}} = -\sum_i \Delta_i W_{j,i} \frac{\partial g(in_j)}{\partial W_{k,j}}$$

$$= -\sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial in_j}{\partial W_{k,j}}$$

$$= -\sum_i \Delta_i W_{j,i} g'(in_j) \frac{\partial}{\partial W_{k,j}} (\sum_k W_{k,j} a_k)$$

$$= -\sum_i \Delta_i W_{j,i} g'(in_j) a_k = -a_k \Delta_j$$

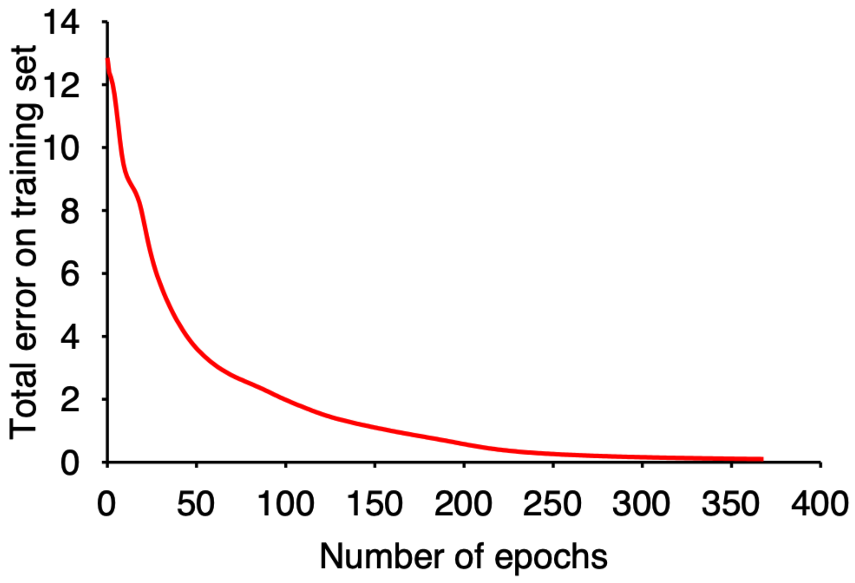
In English, hidden node j is responsible for some fraction of the error at each output node it is connected to. So we divide total error by the strengths of connections and propagate our search for zero derivatives back through the layers



The above diagram demonstrates the risk of gradient descent. We aren't guaranteed to reach a global minimum and could get trapped in a local minimum. When we have lots of parameters, we end up with lots of local minima.

5.4 Back-propagation learning - continued

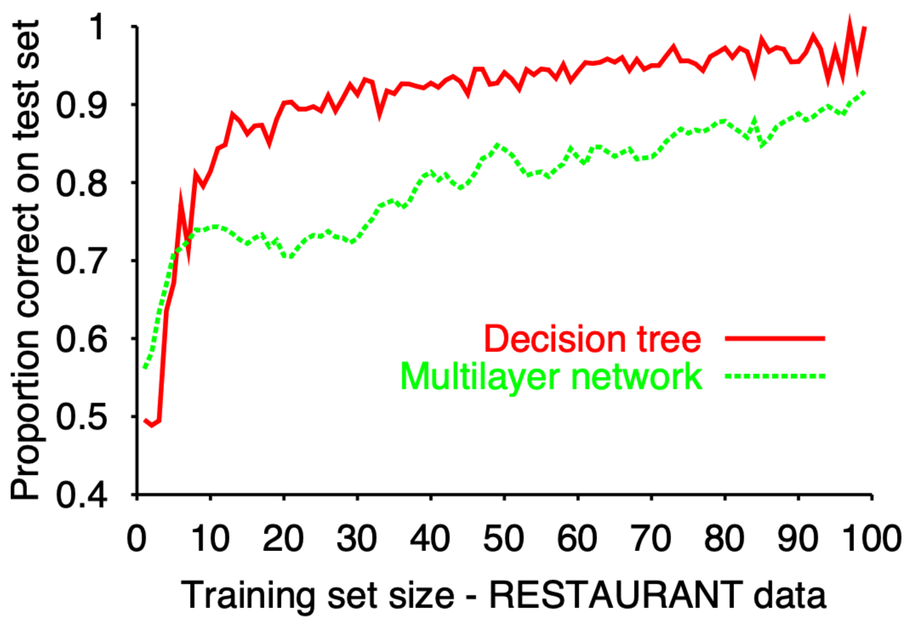
At each epoch (iteration of forward and back propagation), sum gradient updates for all examples and apply Training curve for 100 restaurant examples: finds exact fit



Typical problems are slow convergence and local minima. If your error isn't decreasing as epochs grow then there is an issue somewhere.

The error surface is quadratic near a minimum, but there can be many minima

Learning curve for MLP with 4 hidden units:

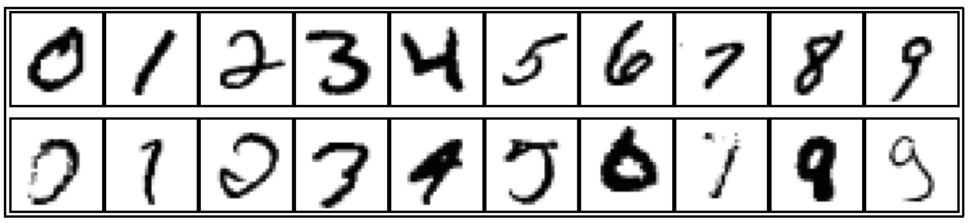


MLPs are quite good for complex pattern recognition tasks, but resulting hypotheses cannot be understood and interpreted easily. This is a big focus of current research so that we can better interpret NNs.

White box AI and black box AI: I can explain DT predictions to a 10 year old child - I typically can't understand the MLP predictions myself

6 Handwritten digit recognition

OA: "this section is useless"



3-nearest-neighbour = 2.4% error 400-300-10 unit MLP = 1.6% error LeNet: 768-192-30-10 unit MLP = 0.9% error

Current best (kernel machines, vision algorithms) ≈ 0.6% error

Current best changes all the time. The MLP, SVM, RF, etc. communities are in competition

7 Summary

- Most brains have lots of neurons; each neuron ≈ linear-threshold unit (?)
- Perceptrons (one-layer networks) insufficiently expressive
- Multi-layer networks are sufficiently expressive; can be trained by gradient descent, i.e., error back-propagation
- Many applications: speech, driving, handwriting, fraud detection, etc.
- Engineering, cognitive modelling, and neural system modelling subfields have largely diverged