Imperial College London

Machine Learning
Back Propagation

Murray Shanahan

Overview

- The basics
- The maths
- Finishing touches
- Taking things further

Motivation

- Machine learning achieved some notable success in the 2000s using *deep neural networks* (deep learning)
- These are *feedforward* networks of artificial neurons (or *perceptrons*) with many hierarchical layers
- The early layers learn low-level features (eg: edges), and the later layers learn high-level features (eg: cats)
- The foundation for most of these techniques is a learning rule called *back propagation*
- These notes draw on a tutorial by Michael Nielsen:

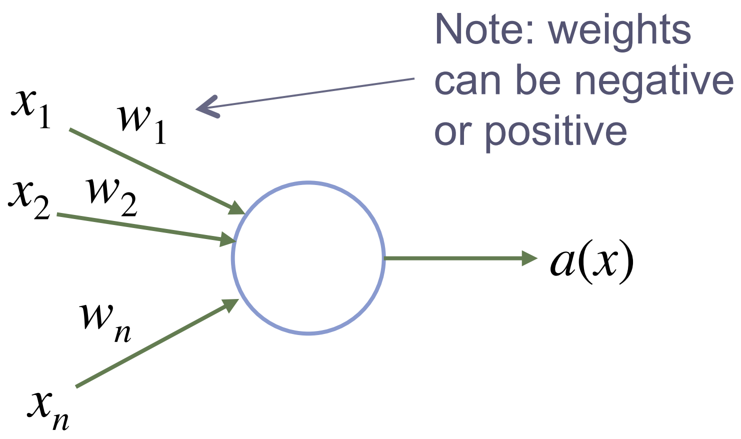
  http://neuralnetworksanddeeplearning.com/index.html

The basics

Artificial Neurons

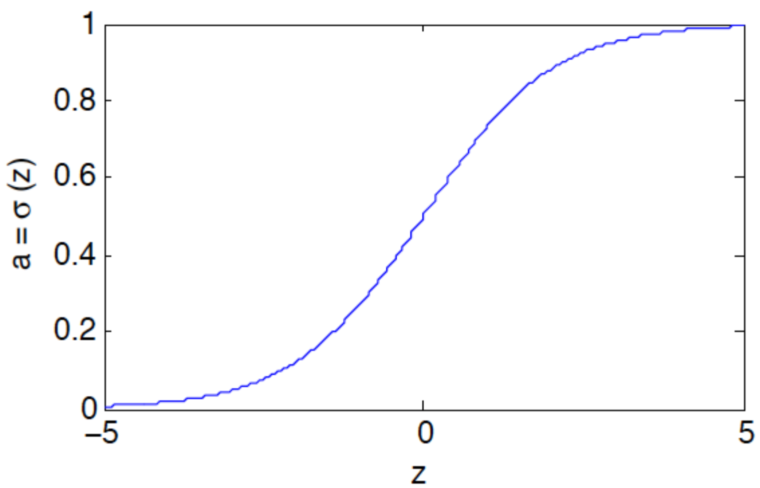
- Artificial neurons compute some function a (called the *activation function*) of the sum z of their weighted inputs

$$z(x) = \sum_{i=1}^{n} w_i x_i$$

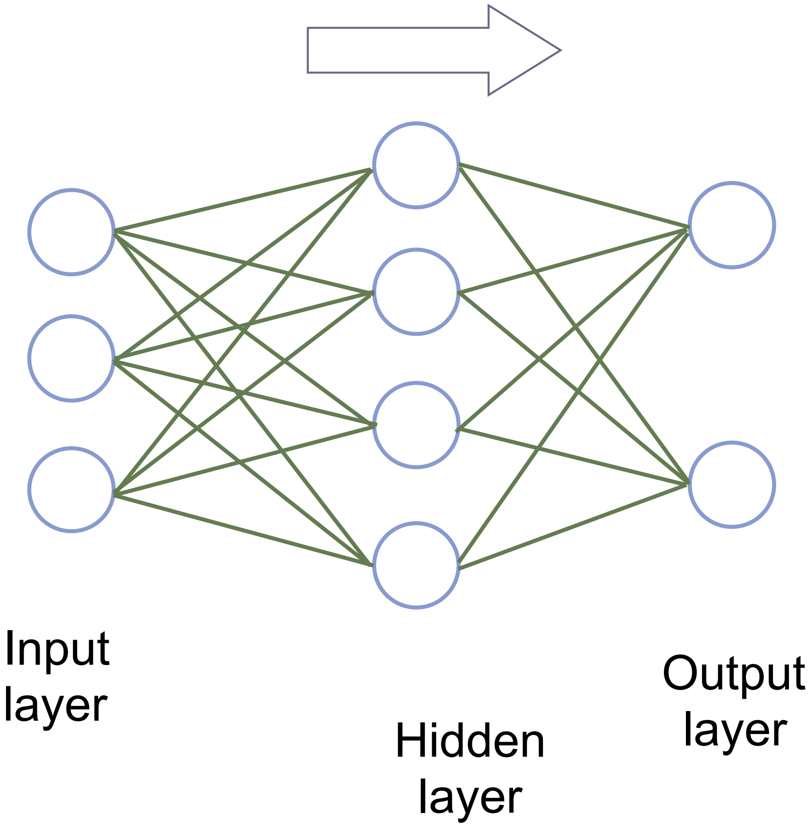


- We'll use the sigmoid function

$$a(x) = \sigma(z(x)) = \frac{1}{1 + e^{-z(x)}}$$



Feedforward Networks



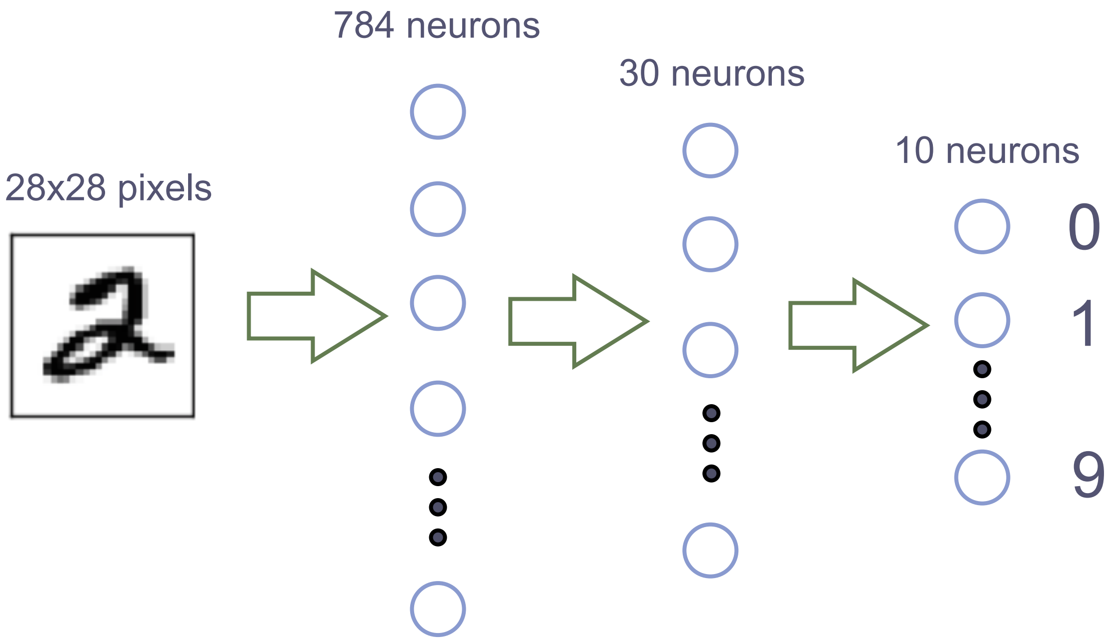
- Back propagation works with *feedforward* networks
- A feedforward network comprises an input layer, an output layer, and a number of hidden layers
- All connections go from the i^{th} layer to the i+1th
- The output of the network is obtained by computing the outputs of each layer in turn, starting from the input layer

Training a Network

- The function a network computes is determined by all its weights
- Suppose we want a network that computes some function, but we don't know what weights to use
- The back propagation rule allows us to train the network using lots of example input-output pairs, adjusting the weights to gradually improve its performance
- Learning to recognise handwritten digits is a standard benchmark

A fragment of the MNIST dataset
of handwritten digits
(http://neuralnetworksanddeeplearning.com)

Recognising Digits



A network for handwritten digit recognition

- For this task, the input is a 28x28 array of pixels. So the input layer comprises 784 neurons
- The output layer comprises 10 neurons, one for each possible digit
- Let's use a single hidden layer of 30 neurons

The Cost Function

- We need a measure of how well a given network performs
- Suppose we want the network to compute a vector $y(x)$
- Let a_i^l denote the output of the i^{th} neuron in the l^{th} layer
- Then the *cost* C for a set of examples of x is

$$C = \frac{1}{2n}\sum_x \frac{1}{m}\sum_i \left(y_i(x) - a_i^L\right)^2$$

where n is the number of examples and m is the number of neurons in the output layer L

The Basic Idea

- Suppose we initialise the network with random weights from a normal distribution with mean 0
- The challenge is to read in a set of training examples and adjust the weights up or down to reduce the cost function averaged over that training set
- When presented with an unseen set of test examples, the network should then be better than it was before the weights were adjusted
- The network will get better and better if we do this lots of times

Cost Gradients

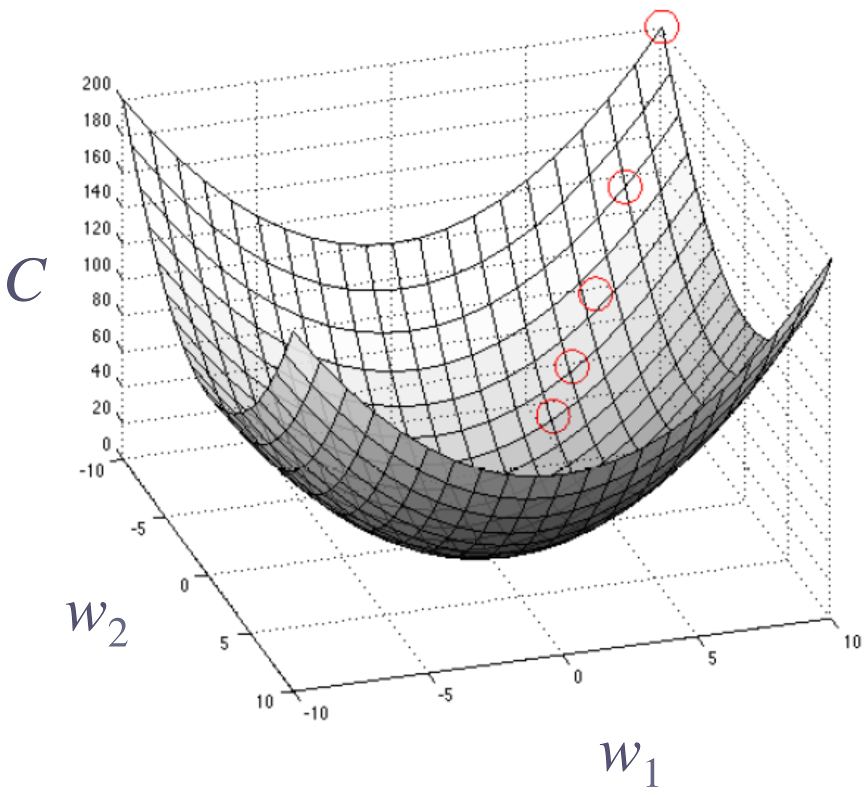
- But how do we adjust the weights?
- For each training example, we need to know the gradient of the cost with respect to each weight. In other words we need an expression for

$$\Delta_{jkl} = \frac{\partial C}{\partial w_{jk}^{l}}$$

where w_{jk}^{l} is the weight of the connection from neuron k in layer l-1 to neuron j in layer l

- Then we can nudge w_{jk}^{l} in the direction of $-\Delta_{jkl}$

Gradient Descent



- More precisely, each weight is repeatedly updated as follows

$$w^{l}_{jk} \rightarrow w^{l}_{jk} - \eta \Delta_{jkl}$$

where η is the *learning rate*

- The resulting algorithm performs *gradient descent* on the weights in order to minimise the cost function
- η has to be just right to ensure that the algorithm converges quickly without over-fitting

The maths

Output Layer Cost Gradients 1

- For the output layer L, we know
  - How cost varies with the output of neuron j
  - How the output of neuron j varies with the weighted sum of its inputs, and
  - How the weighted sum of its inputs varies with the weight of the connection from neuron k in layer L-1

$$\frac{\partial C}{\partial a_j^L} = \frac{1}{n}\sum_x \left(a_j^L - y_j(x)\right)$$

$$\frac{\partial a_j^L}{\partial z_j^L} = \sigma'(z_j^L)$$

$$\frac{\partial z_j^L}{\partial w_{jk}^L} = a_k^{L-1}$$

Output Layer Cost Gradients 2

- So we can apply the chain rule to get the expression we need for neurons in the output layer

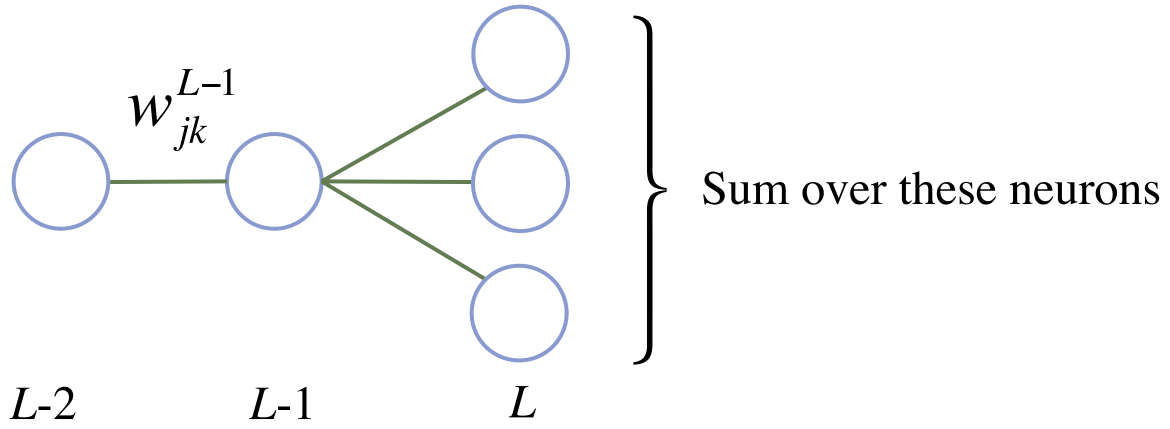
$$\frac{\partial C}{\partial w_{jk}^{L}} = \frac{\partial z_{j}^{L}}{\partial w_{jk}^{L}} \cdot \frac{\partial a_{j}^{L}}{\partial z_{j}^{L}} \cdot \frac{\partial C}{\partial a_{j}^{L}}$$

$$\frac{\partial C}{\partial w_{jk}^{L}} = a_{k}^{L-1} \sigma'(z_{j}^{L}) \frac{1}{n} \sum_{x} \left(a_{j}^{L} - y_{j}(x) \right)$$

Hidden Layer Cost Gradients 1

- But what about neurons further back, in the hidden layers?
- Let's consider layer L-1. Using the chain rule, we have

$$\frac{\partial C}{\partial w_{jk}^{L-1}} = \frac{\partial z_j^{L-1}}{\partial w_{jk}^{L-1}} \cdot \frac{\partial a_j^{L-1}}{\partial z_j^{L-1}} \cdot \sum_i \frac{\partial z_i^L}{\partial a_j^{L-1}} \cdot \frac{\partial a_i^L}{\partial z_i^L} \cdot \frac{\partial C}{\partial a_i^L}$$



Hidden Layer Cost Gradients 2

- Similarly, for hidden layer L-2 (if there is one) we get

$$\frac{\partial C}{\partial w_{jk}^{L-2}} = \frac{\partial z_j^{L-2}}{\partial w_{jk}^{L-2}} \cdot \frac{\partial a_j^{L-2}}{\partial z_{ij}^{L-2}} \cdot \sum_{i1} \left(\frac{\partial z_{i1}^{L-1}}{\partial a_j^{L-2}} \cdot \frac{\partial a_{i1}^{L-1}}{\partial z_{i1}^{L-1}} \cdot \sum_{i2} \left(\frac{\partial z_{i2}^{L}}{\partial a_{i1}^{L-1}} \cdot \frac{\partial a_{i2}^{L}}{\partial z_{i2}^{L}} \cdot \frac{\partial C}{\partial a_{i2}^{L}} \right) \right)$$

- Differentiating, we get

$$\frac{\partial C}{\partial w_{jk}^{L-2}} = a_k^{L-3} \sigma'(z_j^{L-2}) \sum_{i1} \left(w_{i1j}^{L-1} \sigma'(z_{i1}^{L-1}) \sum_{i2} \left(w_{i2i1}^{L} \sigma'(z_{i2}^{L}) \frac{\partial C}{\partial a_{i2}^{L}} \right) \right)$$

Hidden Layer Cost Gradients 3

- Rearranging, we get

$$\frac{\partial C}{\partial w_{jk}^{L-2}} = a_k^{L-3} \sum_{i1} \left(w_{i1j}^{L-1} \sum_{i2} \left(w_{i2i1}^{L} \frac{\partial C}{\partial a_{i2}^{L}} \sigma'(z_{i2}^{L}) \right) \sigma'(z_{i1}^{L-1}) \right) \sigma'(z_j^{L-2})$$

- Now we can see a way of simplifying the expression

$$\delta_j^{L-2} \qquad \delta_{i1}^{L-1} \qquad \delta_{i2}^{L}$$

$$\frac{\partial C}{\partial w_{jk}^{L-2}} = a_k^{L-3} \boxed{\sum_{i1} \left(w_{i1j}^{L-1} \boxed{\sum_{i2} \left(w_{i2i1}^{L} \boxed{\frac{\partial C}{\partial a_{i2}^{L}} \sigma'(z_{i2}^{L})} \right) \sigma'(z_{i1}^{L-1})} \right) \sigma'(z_j^{L-2})}$$

The Main Equations

- Generalising gives us the three main equations of back propagation in terms of the error δ at each neuron

$$\frac{\partial C}{\partial w_{jk}^{l}} = a_k^{l-1}\delta_j^l \qquad \text{Eq1}$$

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \qquad \text{Eq2}$$

$$\delta_j^l = \sum_i \left(w_{ij}^{l+1}\delta_i^{l+1}\right)\sigma'(z_j^l) \qquad \text{Eq3}$$

- Note the form of Eq3 entails that δ is computed from the output layer back. Hence the term back propoagation

The finishing touches

More Derivatives

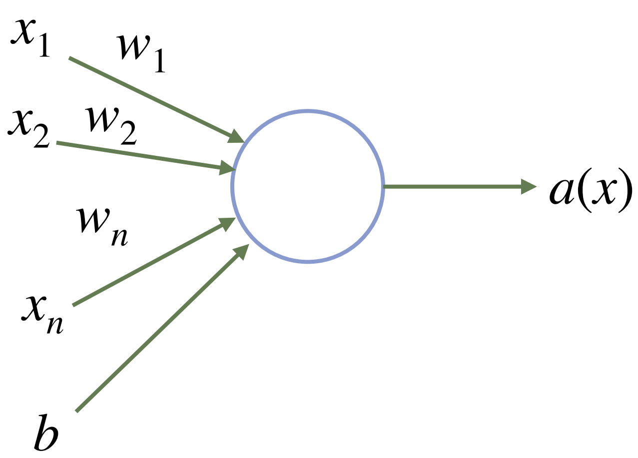
- Just a few more details are needed to give the complete algorithm
- Eq2 is neutral about the cost function, so we need to plug in the derivative of the one we're using
- And we need to know the derivative of the sigmoid function

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \qquad \text{Eq2}$$

$$\frac{\partial C}{\partial a_i^L} = \frac{1}{n} \sum_x \left(a_i^L - y_i(x) \right)$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Adding Biases 1



- For back propagation to work effectively, every neuron is given an extra input, its bias
- This is simply added to the weighted sum of inputs z

$$z(x) = b + \sum_{i=1}^{n} w_i x_i$$

- Biases are randomly initialised and adjusted during learning, just like weights

Adding Biases 2

- Now we need an expression for the gradient of cost with respect to bias
- The derivation is analogous to that for weights, but a bit simpler. The result is the 4th and final basic equation of back propagation

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \qquad \text{Eq4}$$

- Whenever we update weights, we also update biases

$$b_j^l \to b_j^l - \eta \frac{\partial C}{\partial b_j^l}$$

Stochastic Gradient Descent

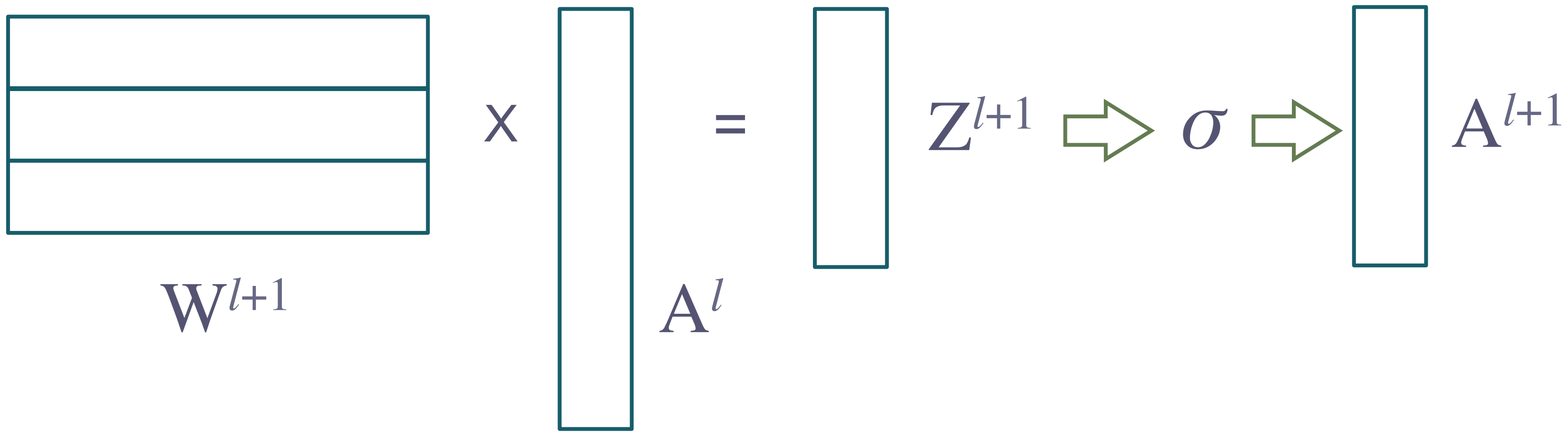
- Ideally, to compute the necessary cost gradients, we would average over the whole training set
- But it's faster to use *stochastic* gradient descent
- The idea is to repeatedly pick a small sample of the training set (called a *mini-batch*), to estimate the gradients based on that sample, and then update the weights and biases accordingly
- More precisely, the training set is randomly shuffled, then chopped up into mini-batches, which are processed one at a time in this way
- And this is done several times

Putting it all Together

```
repeat
   shuffle training set
   partition training set into mini-batches
   for each mini-batch mb
      initialise all estimates of dC/dw and dC/db
      for each training example in mb
         for i = 2 to L
            compute z for each neuron in layer i
            compute a for each neuron in layer i
         for i = L to 2 step -1
            compute δ for each neuron in layer i
         for i = 2 to L
            update dC/dw for each weight w using δ
            update dC/db for each bias b using δ
      for i = 2 to L
         update weights using dC/dw
         update biases using dC/db
```

Acceleration

- The key to getting algorithms like this to be effective is to exploit the available computing power
- A central idea is vectorisation — turning inefficient for-loops into efficient matrix operations
- A good example is calculating the outputs of neurons



The Results

- Each pass through the entire dataset is known as an *epoch*
- Using the back propagation algorithm on the MNIST handwritten digit dataset, it's possible to obtain a success rate better than 95% using
  - a three-layer network with 30 hidden nodes
  - a mini-batch size of 10
  - a learning rate of η = 3.0
  - 50,000 training examples
  - in just three epochs
- On a contemporary (2015) laptop this takes a few minutes

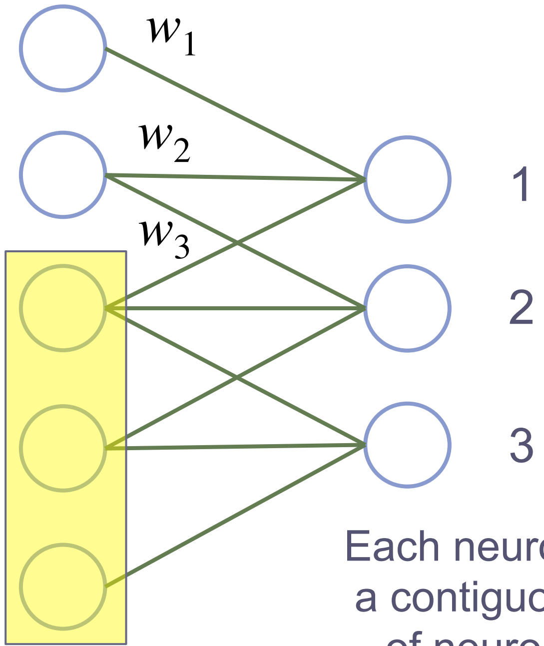
Taking things further

Variations on the Theme

- There are many variations on the basic theme of the algorithm described
- The activation function can be varied (eg: tanh)
- The cost function can be varied (eg: cross-entropy)
- The neuron can be made stochastic (eg: Boltzmann machines)
- The architecture of the network can be varied
  - Convolutional networks
  - Autoencoders
  - Stacked hidden layers

Convolutional Neural Networks

Weights w_1 to w_3 are shared with neurons 2 and 3

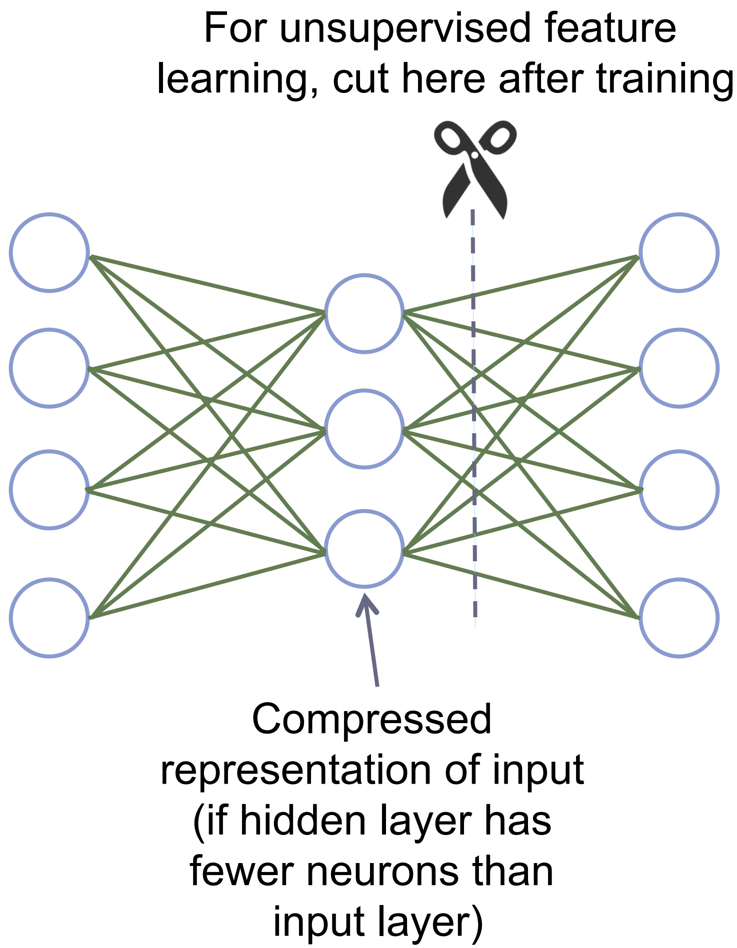


Each neuron looks at a contiguous subset of neurons in the previous layer

- *Convolutional neural networks* (CNNs) are useful for spatially-structured data (eg: vision)
- A CNN has two main features
  - Instead of all-to-all connectivity, neurons in hidden layers are connected to *spatially contiguous subsets* of neurons in previous layers
  - Weights are *shared* among neurons rather than being unique to each connection

Autoencoders

- In an *autoencoder*, the output layer has the same number of neurons as the input layer
- The cost function compares the input layer to the output layer, and tries to minimise the difference
- Training the network turns the hidden layer(s) into a compressed representation of the input
- This can be used for image reconstruction or denoising
- And for unsupervised feature learning



Further Study

- The excellent tutorial from Michael Nielsen that I based some of the notes on

  http://neuralnetworksanddeeplearning.com/index.html

- A very good and thorough series of tutorials from the developers of Theano, a Python deep learning library

  http://deeplearning.net/tutorial/contents.html