# Neural Networks

*High level notes*



Credit
http://neuralnetworksanddeeplearning.com/chap1.html

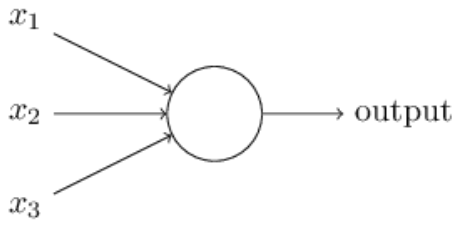
# Artificial Neurons

## Perceptron

Perceptrons were developed in the 1950s and 1960s... Today, it's more common to use other models of artificial neurons...

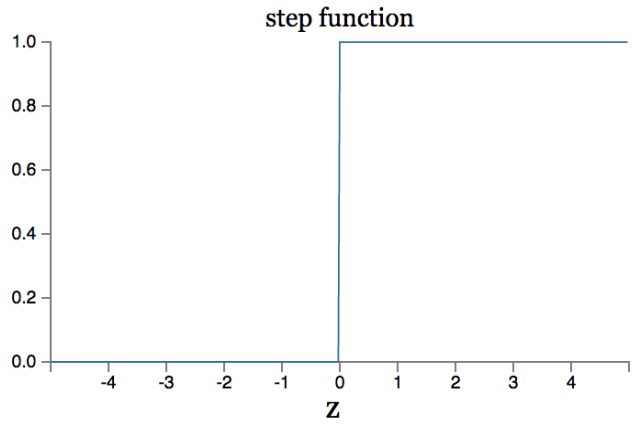
### Characteristics

- Many binary input values
- Single binary output value
- Weighted inputs
- Inputs are summed
- Has a threshold
- Activates if sum > threshold

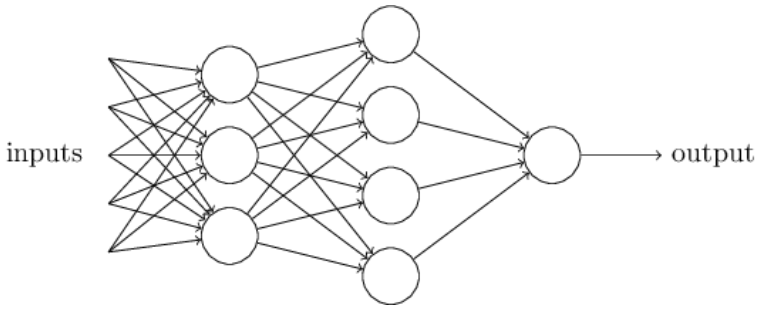



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

### Activation Function



### Network




### Layer

Depicted as a vertical column of perceptrons (neurons).

### Single output

Perceptrons have a single output value. The arrows indicate the output from a perceptron is used as input to several other perceptrons (easier than one line that splits).

### Bias

*Edit: no, this is wrong.*

Simply negative threshold. Allows a simplified activation function. An inequality against 0. Fire if sum is > 0, otherwise don't fire.

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Given a threshold -3 as an example, w * x > -3 is equivalent to w * x + 3 > 0. Another way to look at this. Instead of totalling all inputs and testing if they exceed a threshold, we'll prime the neuron with a negative threshold and only fire if the sum of all inputs exceeds 0.

### NAND Gate

Imagine a perceptron with a threshold of -3. It therefore has a bias of 3 (-threshold). It will fire if the sum of all weights * inputs > -3. It will also fire if the sum of all weights * inputs + 3 <= 0.

Imagine it has two inputs with weights of -2.

| input | w*x | + b | = sum | output |
|---|---|---|---|---|
| 00 | -2*0 + -2*0 | + 3 | = 3 | 1 |
| 10 | -2*1 + -2*0 | + 3 | = 1 | 1 |
| 01 | -2*0 + -2*1 | + 3 | = 1 | 1 |
| 11 | -2*1 + -2*1 | + 3 | = -1 | 0 |

Now, we can see the output is 1 unless both inputs are not 1.

### Multiple Outputs, Same Destination

A perceptron output can connect multiple times to another's input. Really, this is the same as a single connection with twice the weight. So, it doesn't matter much.

### Trainability

In order for a network to learn, it needs to be able to make small changes in the final output toward the right answer. We'd make a small change to the weights and biases of the neurons in order to get a small change in output. However, with binary inputs and outputs, a small change in weights and biases in a perceptron network can radically change the final output. We can cause a flip from 0 to 1 easily, setting off a chain reaction of radical change. This makes perceptrons not well suited for controlled learning. We need something better. Like...

## Sigmoid Neurons

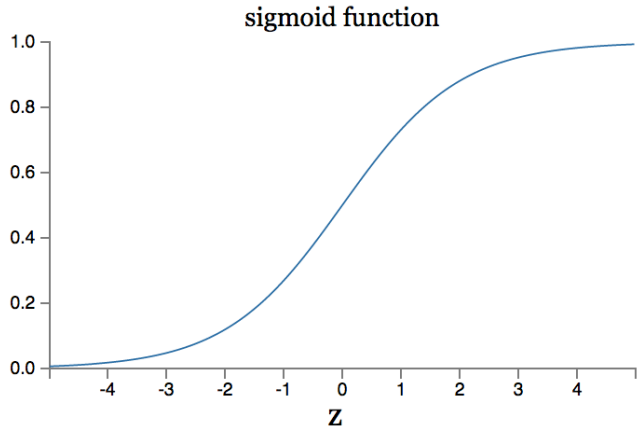
*a.k.a. Logistic Neurons*

Largely the same as perceptrons, except the activation function is smoothed by a logistic function; a sigmoid function.

### Characteristics

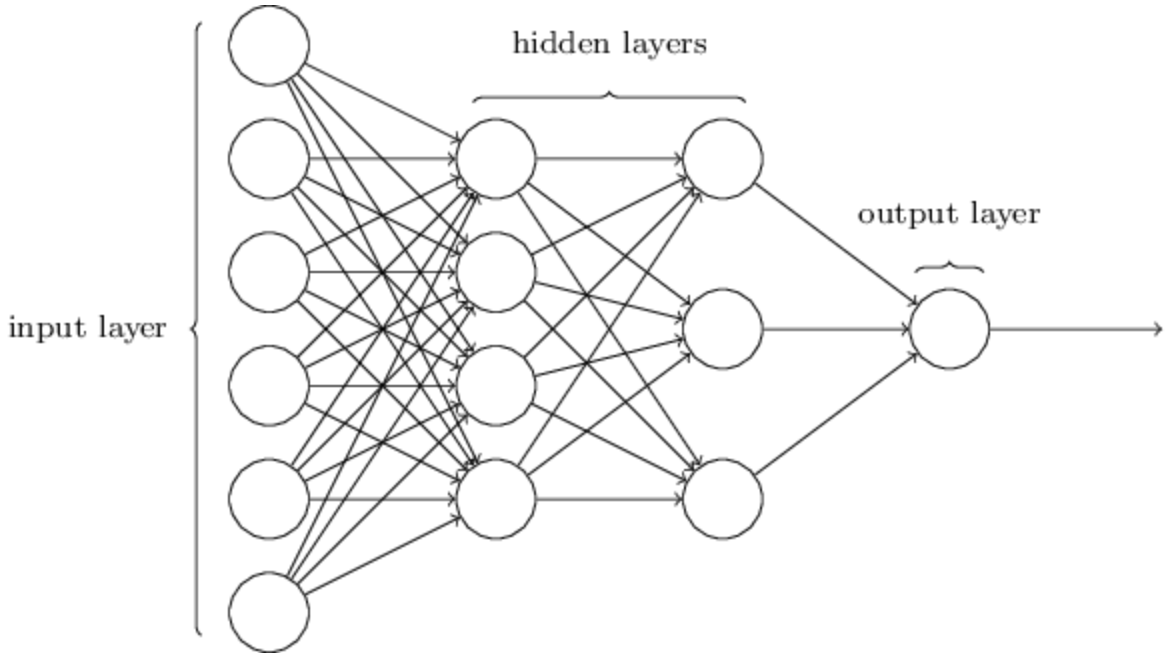
- Many float input values (0-1)
- Single output value (0-1) - sigmoid, not linear
- Weighted inputs
- Inputs are summed
- Has a threshold
- Activates if sum > threshold

### Activation Function




The smoothness of the activation function (sigmoid) means that small changes Δwj in the weights and Δb in the bias will produce a small change Δoutput in the output from the neuron.

# Neural Network Architecture




The above is a 4 layer network with 2 hidden layers. "Hidden" sounds mysterious, it only means "not an input or an output". Sometimes confusingly called *multilayer perceptrons* or *MLPs*, despite being made up of sigmoid neurons, not perceptrons.

## Input/Output Layer Design

The design of the input and output layers in a network is often straightforward. To determine whether a handwritten image depicts a "9" or not, encode the pixel intensities (scaled 0-1) into the input neurons. A 64x64 image would have 4,096 input neurons. The output layer would contain just a single neuron where output < 0.5 = “not a 9" and vice versa.

*Fun fact: the human eye has 120-130 million rods and cones each feeding a synapse.*

## Hidden Layer Design

It's not possible to sum up the design process for the hidden layers with simple rules of thumb. Neural networks researchers developed many design heuristics for hidden layers. Example, they can help determine how to tradeoff the number of hidden layers against the time required to train the network.

## Types of Networks

### Feedforward

Output from one layer is used as input to the next layer, no loops.

### Recurrent networks

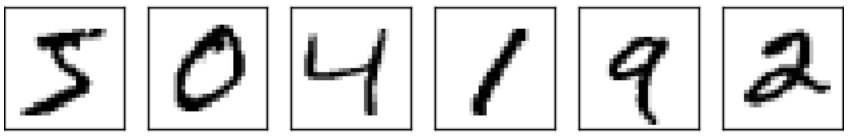
Neural networks in which feedback loops are possible.

Recurrent neural nets have been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are (at least to date) less powerful.  Closer in spirit to how our brains work.

# Handwritten Digit Classifier

### Segmentation problem

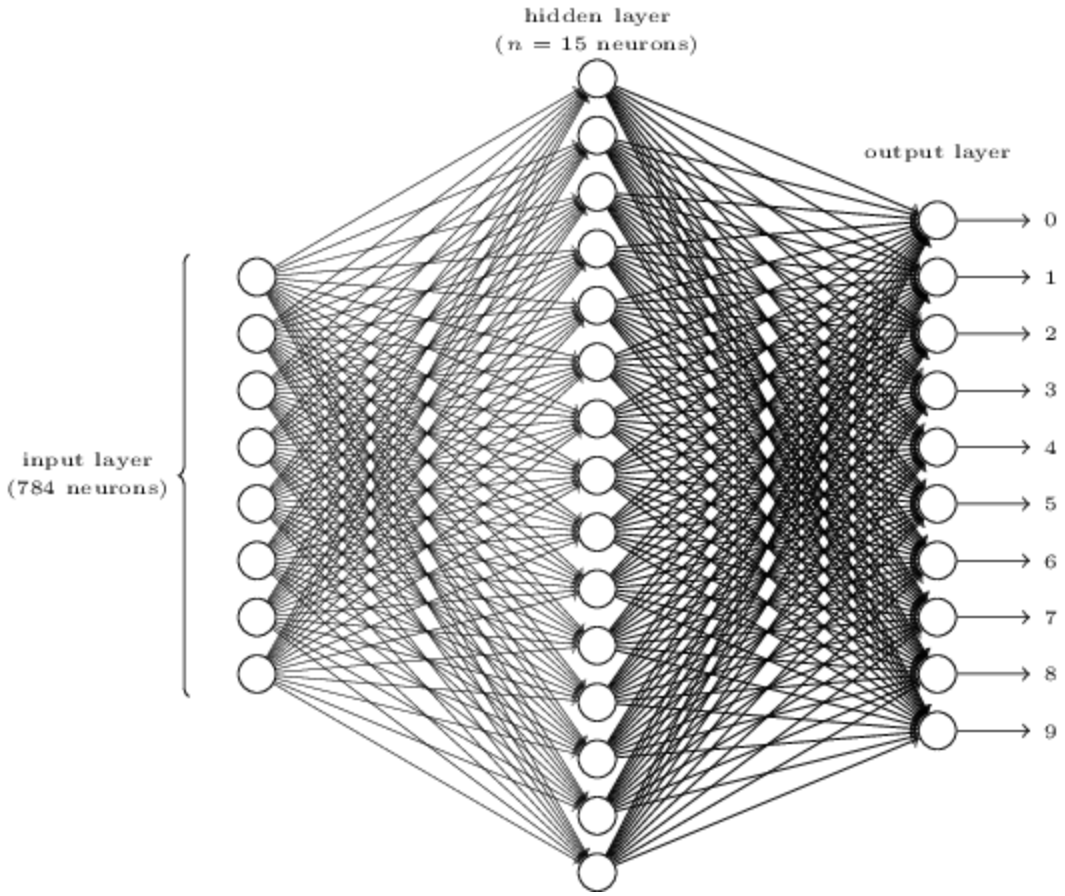
Break up the image into digits.



Segmentation is not so difficult to solve, once you have a good way of classifying individual digits. One approach is to trial many different ways of segmenting the image, using the individual digit classifier to score each trial segmentation.

### Network Design

We will use a three-layer neural network:




Input layer contains 784=28×28 neurons receiving a pixel intensity float value of 0 (black) to 1 (white). Single hidden layer with 15 neurons. Output layer has 10 neurons, one for each digit. Can be done with 4 output neurons, treating them as binary (2^4=16 outcomes). Turns out, 10 output neurons learns to recognize the digits better.

# Learning With Gradient Descent

### MNIST data set

70,000 handwritten digits (see images above) from 250 people and their classifications. 60k for training, 10k for testing. 28x28 grayscale pixels.

Regard each training input x as a 28×28=784-dimensional vector.
Denote corresponding desired output by y=y(x), where y is a 10-dimensional vector.

If an image, x, depicts a 6, then y(x)=(0,0,0,0,0,0,1,0,0,0)T. “T” indicating transpose row vector to column vector.

### Cost Function

*a.k.a loss or objective function*

An algorithm letting us find weights and biases so that the output from the network approximates y(x) for all training inputs x. To quantify how well we're achieving this goal we define a cost function.

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

We'll call *C* the *quadratic* cost function; it's also sometimes known as the *mean squared error* or just *MSE*. We want to find a set of weights and biases which make the cost, C(w,b), as small as possible. We'll do that using an algorithm known as gradient descent.

We want the cost of the weights and biases, *C(w,b)*, to be smaller when the outputs, *y(x),* are closer to the correct output, *a*, and larger when the outputs are further from the correct output. This allows us to quantify progress (or not) after making small changes to the weights and biases.

### Gradient Descent

...you lost me at derivatives. Hope I don't need to know this part, cause I'm skipping it.

In general, it's a method for solving minimization problems. This entire process has lots of variables like MNIST, network architecture, weights, biases, etc. It can be minimized. Realizing this method, we can also minimize the cost function (problem with lots of variables) in the same way.

...lots of math

## Implementing Our Network

..starts with a `git clone`, I can do this :) Python and Numpy here. Wonder if I can re-implement this in JS? Notes to follow...