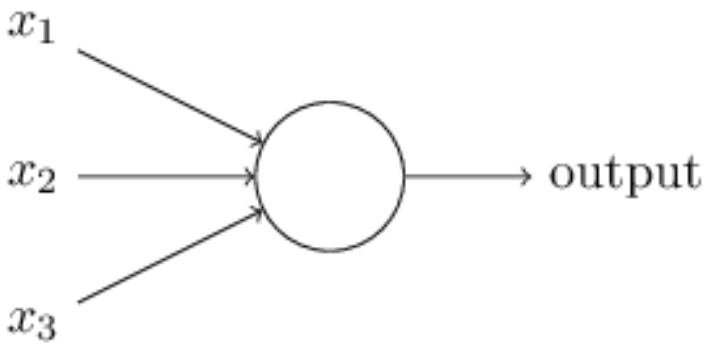
Neuralnetworksanddeeplearning.com

Concepts:

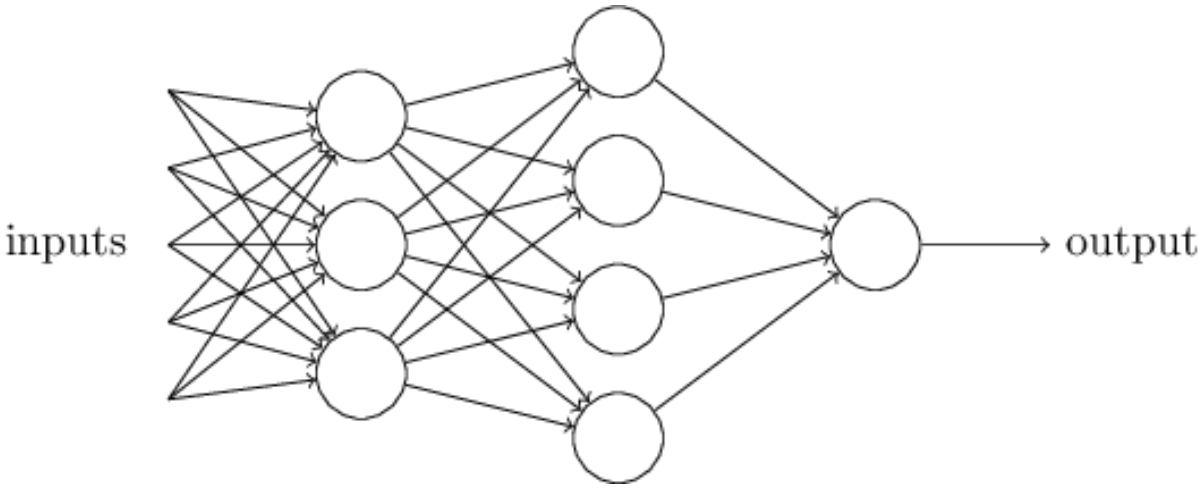
PERCEPTRON:



Takes multiple inputs and produces one output, based one whether the scalar multiple of each input and their respective weight is greater than or less than some numeric threshold value that

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

Most neural networks have more than one abstraction layer that evaluates the inputs by multiplying to their respective weights. This enables sophisticated decision making by making decisions based on the results of the first layer of perceptrons:



Here each perceptron has only one output (even though it looks like they have multiple). Each output is routed to more than one perceptron in the second layer because it is routed as input to multiple perceptrons in the second layer.

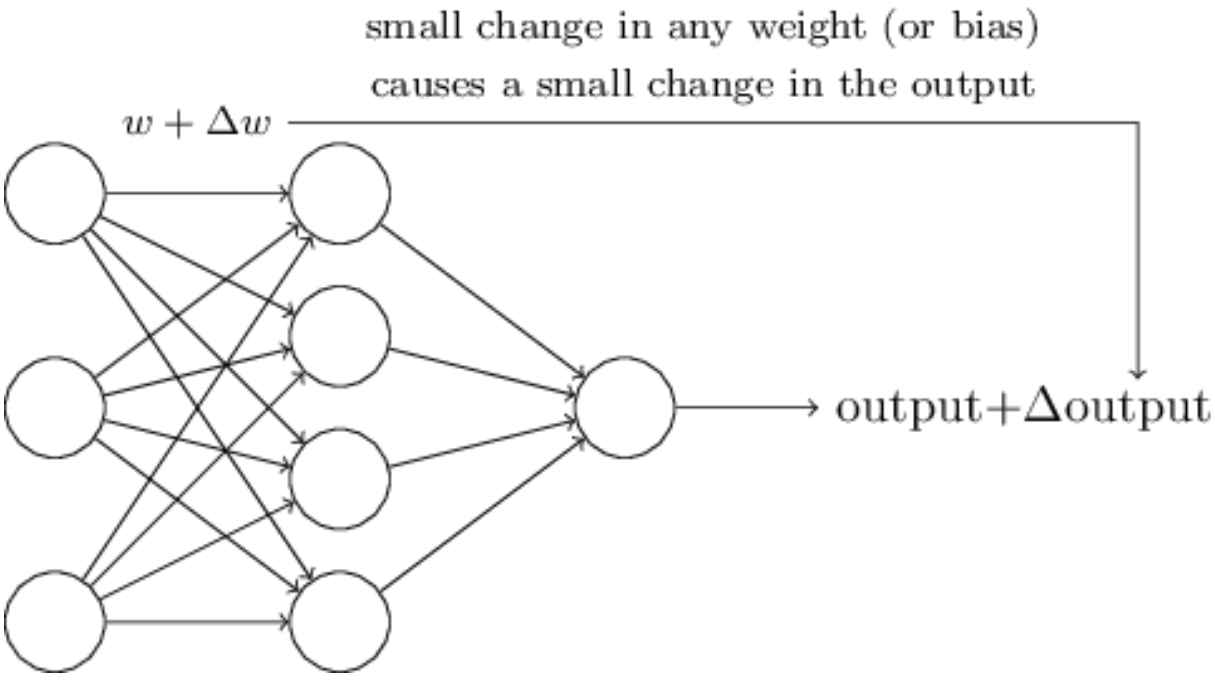
weight w
input x
threshold b which becomes the ‘bias’
w*x > b
w*x - b > 0

x(-2) + x(-2) + 3 > 0

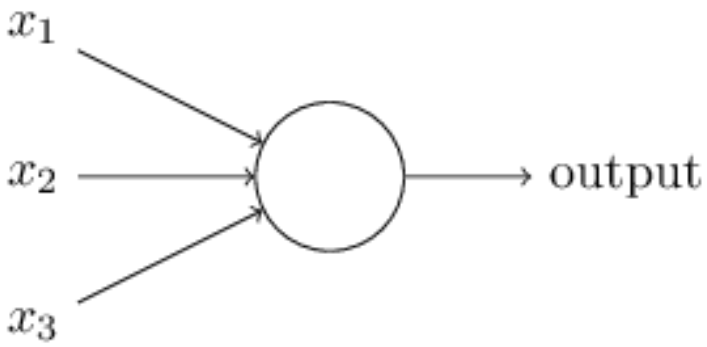
Perceptrons are similar to NAND gates in the sense that they can simulate the same thing and thus are fundamental to computing, but with one huge huge upside: they can learn. A.k.a. we can devise learning algorithms that automatically tune the weights and cases of a network of artificial neurons. Thus they can solve seemingly unconventional problems automatically.

SIGMOID NEUTRONS:

Learning happens when a small change to a weight causes only a small change in the output. In this way, the network can process relativity.



Actually, with perceptrons, when the weights change to accommodate one input and its desired output, they can have changed drastically for all other inputs. We don’t want this. We want gradual learning. So we use sigmoid neurons:



Looks similar to a perceptron, except the math underlying it is far more adept for learning. It has float inputs (decimals) rather than simply integers, and it has an output that is a float as well (decimal) that is not 0 or 1, but rather σ(w*x +b), where σ is called the sigmoid function, (more generally, the sigmoid function is what we use in our model as the 'activation function')

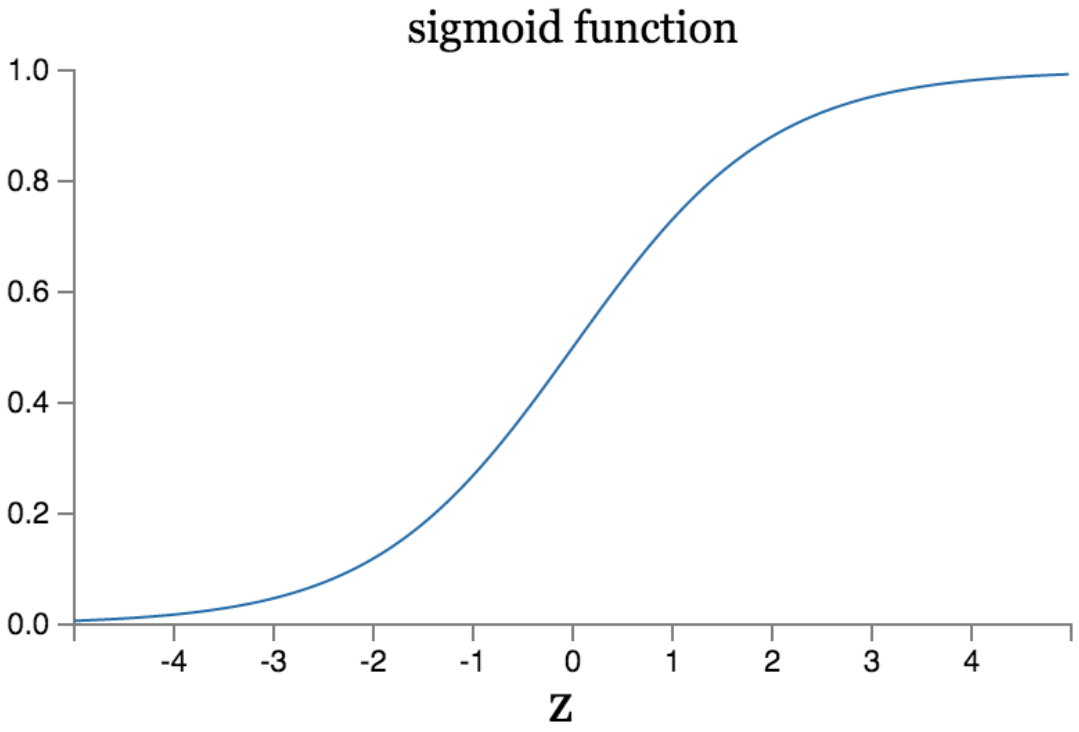
$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

nd is defined by σ(z) = 1/(1+e^(-z))

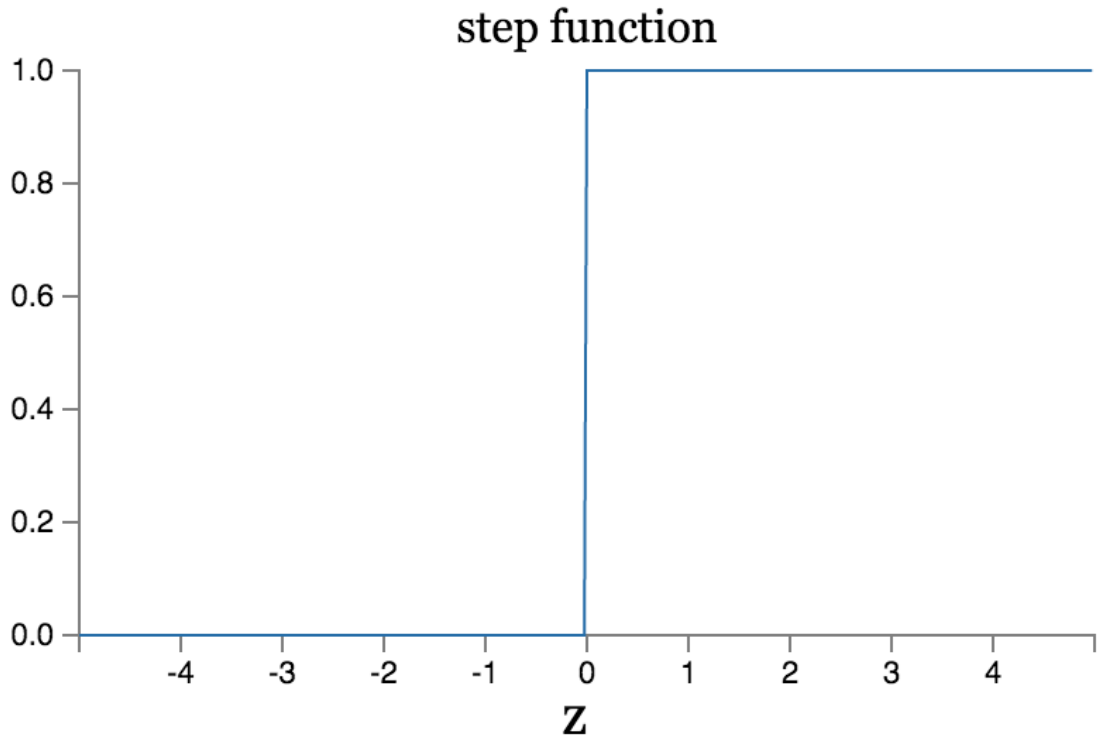
$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

This is called either the sigmoid function or the logistic function.
For very large values of z, e^(-z) approaches 0, so sigma(z) approaches 0. This models the perceptron behavior.
For very small values of z, e^(-z) approaches infinity, so sigma(z) approaches 1. This also models the perceptron.
It is only for intermediate values between large and small that the sigmoid function

differs from the perceptron.



This is in fact just a smoothed out version of the perceptron (which in graph form is a step function (either the neuron fires, or it does not).



The smooth curve allows us to see small changes in weight result in small changes in output, and thus the neural network can learn.

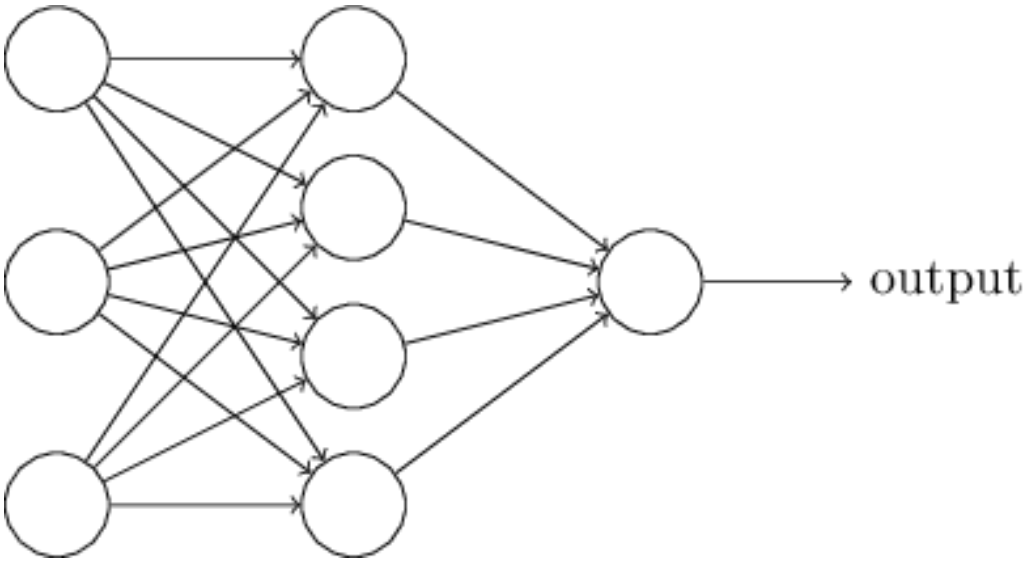
The changes in output can simply be calculated as a linear combination of changes in the weights and bias. (The linearity of this function is what allows us to make a small change in weight and get a small change in output)

$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b,$$

So what about when we actually want the neural network to output say, either 9 not a nine? We can actually just use a convention on a sigmoid function rather than using a perceptron. This convention says that if the output is above a 0.5, we will output 9, and if it is below 0.5, we will output not a 9.

Multiplying all weights and biases by some c>0 does not change the output of the sigmoid function because the bias and the weights balance each other out.

The architecture of neural networks:



Design for input and output layers of the neural network is often simple:
Example, to recognize a 9 digit in a 64x64 pixel grayscale image:

input: 64x64 = 4096 input neurons, each a pixel, with intensities scaled appropriately from 0 to 1.

output: one single neuron with output values over 0.5 indicating this is a9, and output values lower than 0.5 indication this is not a 9.

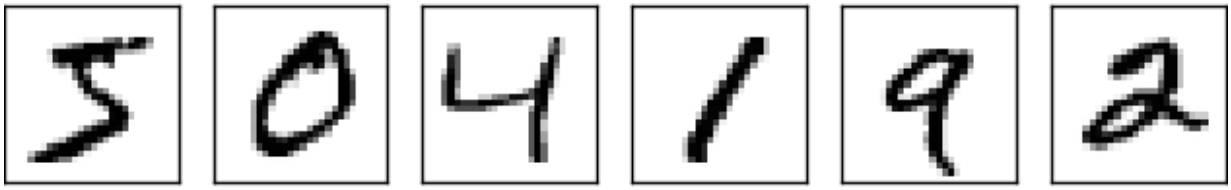
hidden layers: this is the heuristic and art that goes into designing an efficient and effective neural network.

The design in which output from past hidden layers is fed to the next hidden layer is called *feedforward* design.

There exist also *recurrent neural networks*, where neurons don't just fire once for no time duration, but fire for a limited duration. Here, they can pass info back to past layers, because one neuron's output only affects its input at some later time (once it's done firing), not at the same time it's producing that output.
Recurrent neural networks however, require more intricate learning algorithms and thus have ben less useful, but they far more accurately model human cognition.

A simple network to classify handwritten digits:

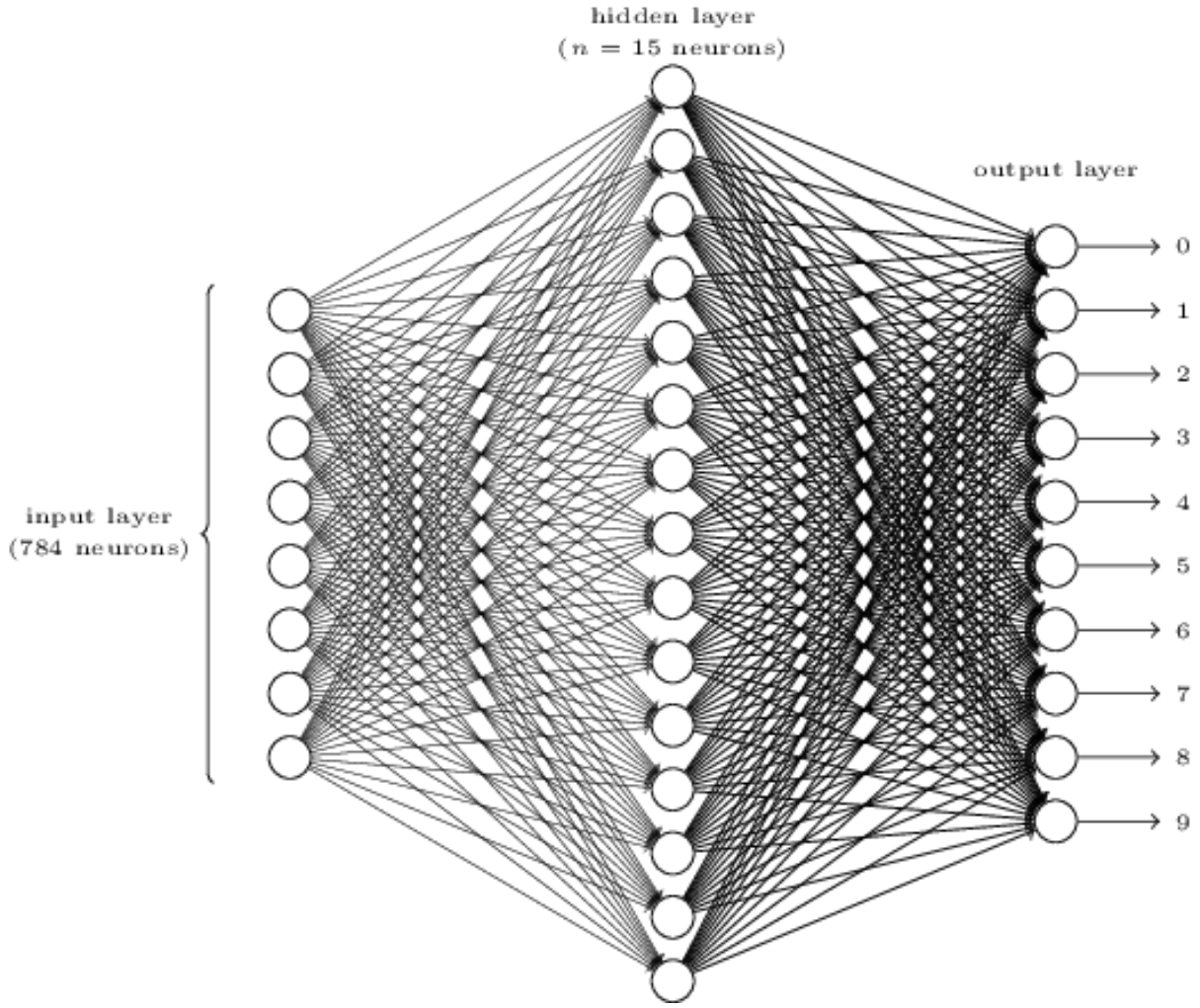
from this input above, we need to do two things: break it up into pieces, and then classify each digit as the number it represents.



One way to break it into digits is to try a couple random places, and then run the digit recognizer. Based on the confidence with which the digit recognizer gives its output, we will know if it's having a lot of trouble, and we'll know we probably didn't divide up the digits correctly, so we send a signal to redivide them in a slightly different way, until we have found the best place at which the classification network is confident in its results.

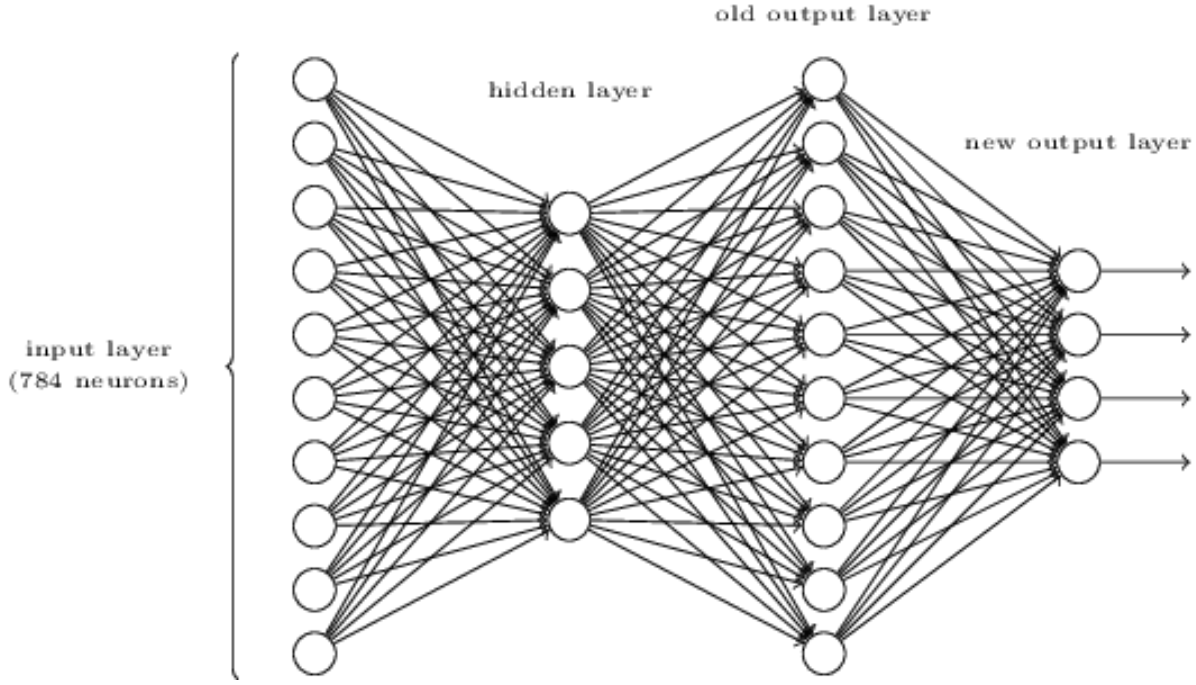
So above should be recognized as 5.

For recognizing digits, it's best to use a three-layer neural network:



We are using for training, images that are 28x28 pixels = 784 pixels, so we will have 784 input neurons. The hidden layer neurons are just 15, but again, this is where the thinking comes in, so here is where we can experiment with the number of neurons. The output layer contains 9 neurons, one for each digit that we could recognize. Their outputs are floats 0.0 through 1.0, representing the degree of confidence with which the network believes the init data has matched the abstraction present in the hidden layer. The neuron with the highest degree of certainty from the output is the one whose number we will return as the guess for what the digit is.

Why do we use 10 output neurons to represent the numbers, rather than four outputs, and represent the output in binary? Turns out the 10 is more accurate. But how could we have known?
You just have to test diff algorithms.
The one with only four outputs turns out having trouble recognizing which part of the image, we look to recognize by overlaying with an image of the correct digit and weighing more heavily those pixels which overlap, is more significant towards determining the identity of a digit as a whole, so the 10 digit approach works best. But again, this is just a heuristic.



Exercise: convert the output of the neural network from digit to binary:
Who knows….

LEARNING WITH GRADIENT DESCENT:

Goal in training a neural network:

-minimize weights and biases which minimize the quadratic cost function C(w,b)

We can use the MNIST standard library to train our network. They give us training data and test data.
First we train the network with the training data:

We must make some estimation of how far off our current output is when using the current weights and bias. We also want to be able to calculate the error even if it is between two values. So if i recognize two images out of ten, then change my weights, and still only recognize two images out of ten, i can tell whether i got closer or farther from recognizing three out of ten. Squaring the error accounts for this (somehow.. ?).

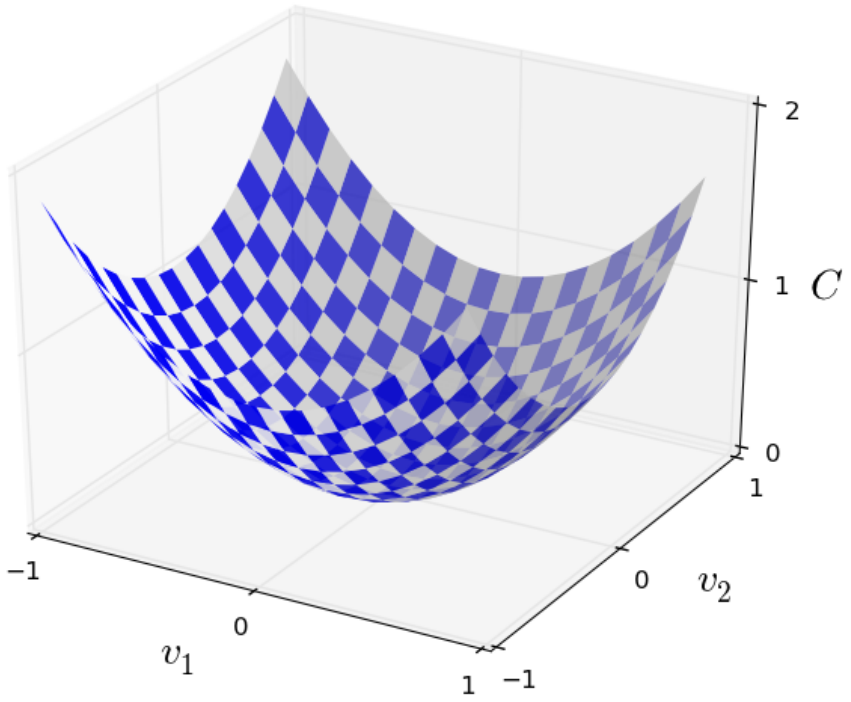
We use the “cost function” (also called ‘loss,’ or ‘objective’ function):

The cost/error using a set of biases and weights is equal to the sum of the positive difference between expected and actual output squared for each training input, all over 2 times the number of training inputs. By using the sum, we attempt to find the weight and biases with the overall least error, considering all the training data.

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

This is called the quadratic cost curve, and to minimize the error in our training data, we abstract our classification accuracy (how many outputs actually match what we want) to a quadratic cost curve, and then minimize that instead. Now, how we can minimize:
To understand the methodology, it's best to generalize and abstract away all details of neural networks, and just focus on finding the global minimum of a function with multiple inputs.
So we can look at some function C(v1, v2). If it were a function of just a couple variables, we may be able to use calculus to take derivatives and then find the extremes therein, and find the global minima with that data. But with many many (perhaps millions) inputs, we need a better technique:

Gradient Descent:



The cost square equation gives us the difference between the expected and the actual

output. This is like an elevation on a hill. If the error is smaller (what we want is 0), we move towards the lower part of the hill. Goal: reach the absolute minimum.

What we do it find the change in error between the point we're currently at (a set of weights and biases), and point near where we currently are (a similar, but slightly different set of weights and biases).

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2.$$

This is calculated by taking partial derivatives of the squared cost function in respect to each variable we change (e.g. v1 and v2) times their respective changes in each variable. This effectively gives us the relation between a change in v and the change in C at that particular point (the slope of the gradient).
We want a negative change in C (indicative of movement down the gradient towards the global minimum).
We can also represent this change in

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T.$$

C as the gradient C (the slopes/partials of each var respect to C) times the changes in each var:

Gradient above (del or nabla. *not delta*)

$$\Delta C \approx \nabla C \cdot \Delta v.$$

Change in C above (gradient times change in v).

We want to move down the curve. So how can we choose some v such that we always move down?

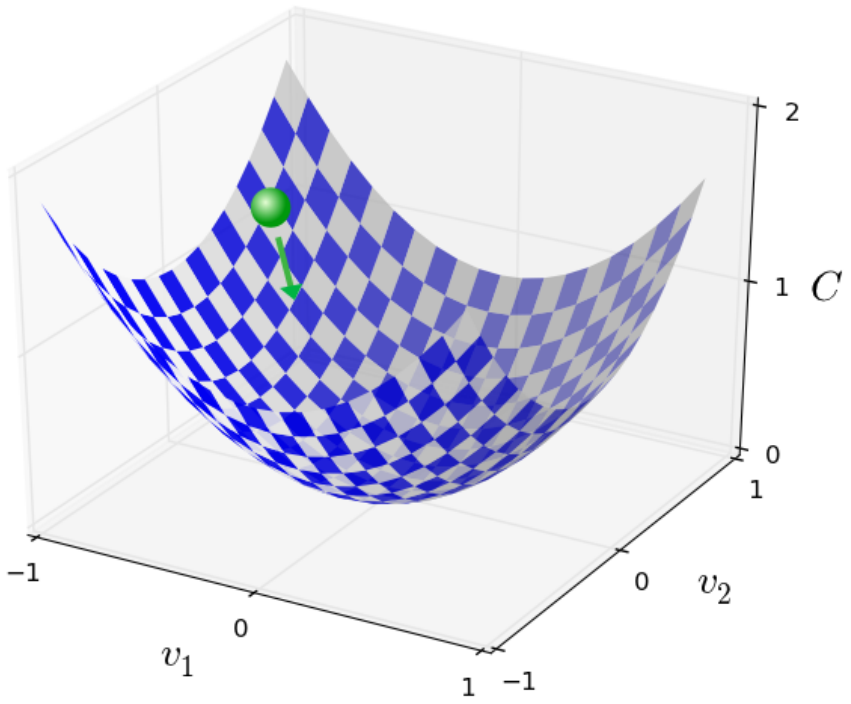
$$\Delta v = -\eta \nabla C,$$

if we change each variable by nabla Cost (gradient of the hill) times the negative of the learning rate (determines the step) (negative so that we move down always) we will always move down because we get delta C as:
deltaC = nablaC * nablaC * -learningRate
deltaC = Math.pow (nablaC, 2) * (-learningRate)
therefore deltaC is always negative.

This is actually proven to be the learning rate that always takes you on the steepest descent path from the point you currently stand on.

$$v \rightarrow v' = v - \eta \nabla C.$$

Update rule: above we move each input by this much



The ball moves down the hill (a.k.a. error gets smaller!!!)

Pay attention to learningRate so that it's not too big (we could overstep the minimum, but also not too small (algorithm could take a long time to run).

This approach is easily generalizable to many variables rather than just one. The update rule stays the same:

v—>v' = v-n(nablaC)
(where n is learningRate and nablaC is the gradient of C)

applying gradient descent to neural networks:

in the equation we are minimizing, we replace the inputs v1, k1, v2, k2… with the weights and biases w1, b1, w2, b2…

thus our position is now w1, b1. and the gradient vector nabla C is not in terms of dC/dv1 and dC/dv2, but rather dC/db1 and dC/dw1.
update rule now takes w to w' and b to b', but those now equal w1-n(dC/dw1) and b1-n(dC/db1).

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Now we have the full set of tools for gradient descent with neural networks.
However, it is slow.
Because the cost function is a summation function, and as such has to iterate over every single input. This takes a lot of time. So we use:
stochastic gradient descent:
like polling in the election, stochastic gradient descent takes a sample and generalizes to get a fast estimate. This is in effect all we need with learning (to move towards zero error).

We randomly choose a *"mini-batch*" of inputs to calculate the cost function from. With this, we calculate the cost function and perform the update rule, then take another sample and keep going until we have iterated through the entire set of inputs. This constitutes one "*epoch".*

The cost function from a mini-batch closely mimics the real cost function(It's pretty close).

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

Thus:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j},$$

Then, performing the update rule with the mini-batch:

$$w_k \to w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \to b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

.

WE NOW HAVE ALL THE COMPONENTS TO WRITE A NEURAL NET THAT RECOGNIZES DIGITS:

From the inputs *w* and *b,*
-choose a mini batch.
-for that mini batch:
Cost function over the mini batch:
C(w,b) = (1/2n)SUMMATION over the number of elements in the mini batch Math.pow(Math.abs(y(x)-a),2)
**note y(x) is the actual output and a is the output we get from our neural network*
Nabla (for each element of min batch???):
Nabla/gradient of C = <partial C/partial b1 , partial C/partial w1>TRANSPOSE
*nablaC for mini batch: (1/m) * Summation of (nabla C for a single input) for all elements in the mini batch*

Delta C: (calculated once the min batch is done running, so just once for the mini batch)

*deltaC = nablaC * delta b*

Update function (once for the min batch):

*w—>w' = w - (learningRate/m)*Summation over all elem in the mini batch (partial C / partial Ws)*

*b—>b' = b -b (learningRate/m) * Summation over all elem in the mini batch (partial C / partial Bs)*

-choose another random mini batch and run it again.

-once the epoch is done, run another epoch yo!

So,

functions that run once per set of inputs in the min batch then average over the mini batch:

1. cost function (error) components
2. nabla for each component set of the mini batch
3. —
4. —

functions that run once per mini batch are

1. calculation of cost itself
2. nabla gradient of C overall
3. delta C
4. runs once for each weight: update function