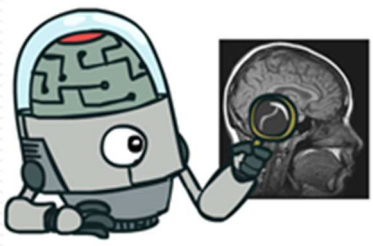
MACHINE LEARNING FOR DATA MINING
LECTURE 3-1: NEURAL NETWORKS

REPRESENTATION, MODELS AND APPLICATIONS

Siamak Sarmady (Urmia University of Technology)
Geoffrey Hinton (Toronto University)
Andrew Ng (Stanford University)

Machine Learning and Neural Nets



What is Machine Learning?

Difficult tasks:

- **Object Recognition:** It is very hard to write programs that solve problems like recognizing a three-dimensional object from a novel viewpoint in new lighting conditions in a cluttered scene.
  - We don't know what program to write because we don't know how (the algorithm) its done in our brain.
  - Even if we had a good idea about how to do it, the program might be horrendously complicated.

What is Machine Learning?

Uncertain and changing conditions:

- **Fraud Detection:** It is hard to write a program to compute the probability that a credit card transaction is fraudulent.
  - There may not be any rules that are both simple and reliable. We need to combine a very large number of weak rules.
  - Fraud is a moving target. The program needs to keep changing.

The Machine Learning Approach

- Instead of writing a program by hand for each specific task, we collect lots of examples that specify the correct output for a given input.
- A machine learning algorithm then takes these examples and produces a program that does the job.
  - The program produced by the learning algorithm may look very different from a typical hand-written program. It may contain millions of numbers (or just a few e.g. linear regression).
  - If we do it right, the program works for new cases as well as the ones we trained it on.
  - If the data changes the program can change too by retraining on the new data.
- Massive amounts of computation are now cheaper than paying someone to write a task-specific program.

Some examples of tasks best solved by learning (specially Neural Nets)

Example tasks suitable for Neural Networks:

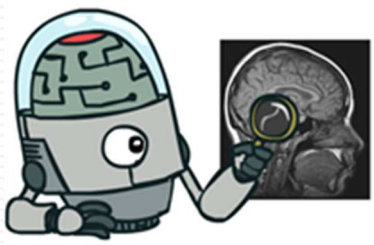
- **Recognizing patterns:**
  - Objects in real scenes
  - Facial identities or facial expressions
  - Spoken words
- **Recognizing anomalies:**
  - Unusual sequences of credit card transactions
  - Unusual patterns of sensor readings in a nuclear power plant
  - Unusual pattern of activities/packets on a network
- **Prediction:**
  - Which movies or products will a person like?
  - Future stock prices or currency exchange rates

Reasons to study and use neural networks

Why we study Neural Networks:

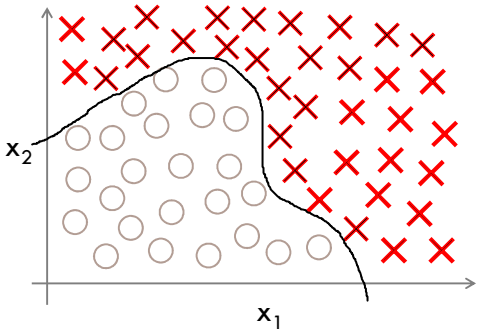
1. To understand how the brain works
   - It is big and complicated and dies when you poke it around.
   - So we need to use computer simulations instead
2. To understand the style of parallel computation of neurons
   - Very different from sequential computation
     - should be good for things that brains are good at (e.g. vision)
     - Should be bad for things that brains are bad at (e.g. 23 x 71)
3. To solve practical problems using learning algorithms similar to the brain's (this course)
   - Learning algorithms can be useful even if they do not exactly work like the brain

Non-linear Hypotheses



Complex Non-linear problems

- Sometimes it is not possible to separate classes with linear boundries. In these cases we need classifiers that are able to find non-linear boundries between classes. Finding the coefficients ($\theta_{0...}$ θ_n) of those polynomial terms is difficult but possible when we have only two features (i.e. x_1 and x_2).



$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1 x_2 + \theta_4 x_1^2 x_2 + \theta_5 x_1^3 x_2 + \theta_6 x_1 x_2^2 + \ldots$$

We have terms for every rank of x_1 and x_2 , we need to find their coefficients

For 3rd degree we have 4x4 terms (powers of x1*powers of x2)

Complex Non-linear problems

- For just two features, it is possible to find a class boundry (e.g. by curve fitting).
- But when the number of features increase (e.g. To 100) then it is almost impossible to use curve fitting method to find the boundries.
- If we want to use previous method, we will have huge number of polynomial terms and finding the coefficients of those terms needs huge amount of computation (almost impossible).

Complex Non-linear problems

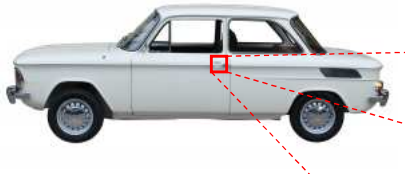
- Assume we change the House price prediction problem in a way that it becomes a binary classification problem. For example, having 100 parameters about every house we want to predict whether it will be sold in the next 6 months.
- Even if we just want to consider short quadratic (2nd order of $x_i.x_j$ form only) terms, we will have large number of terms (almost 5000, i.e. We need to find $\theta_1 ... \theta_{5000}$).
- That would convert the problem into solving a $O(n^2)$ problem.
- Now notice that quadratic terms are not enough for classification of most non-linear problems.

x_1 = size
x_2 = #bedrooms
x_3 = #floors
x_4 = age
...
x_{100} =

What is this?

- Assume we want to supply images to a machine learnin algorithm and see whether it is a car or not. We see an image of a car, but what a computer sees, is a series of numbers.
- So the computer should look at this matrix of pixel densities and guess whether it is a car or not (or possibly tell us it is a door handle of a car).

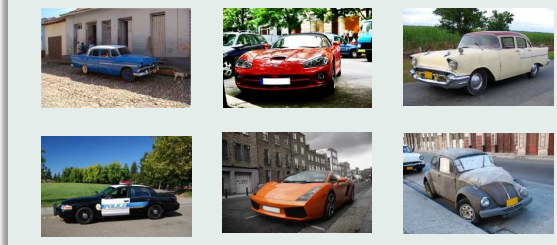
We see this:



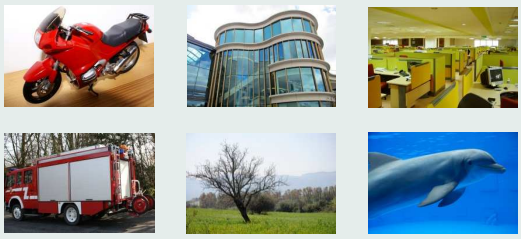
But the camera sees this:

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	75	65
20	43	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50

Computer Vision: Car detection



Cars



Not a car

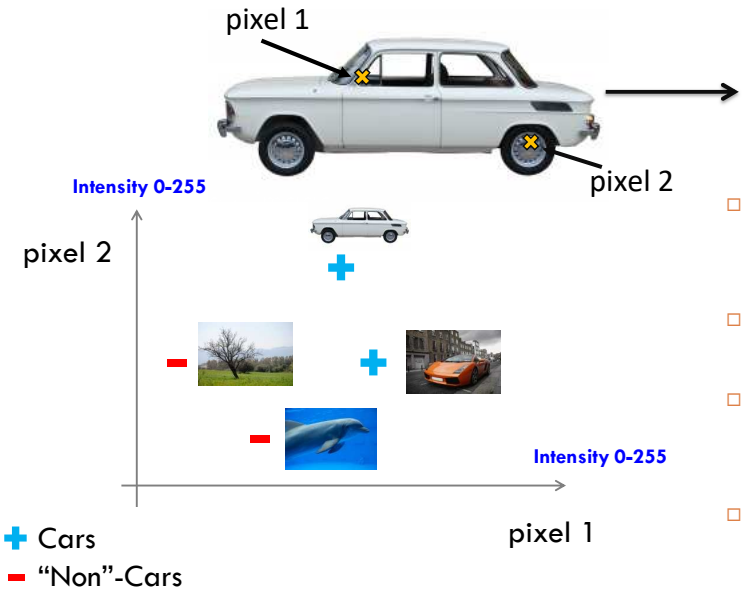
When we want to train a car detection classifier:

- We prepare a training set of both cars and none-cars. The training set includes correct labels (car/ not a car).
- We then test it with a new not-seen data.

Testing:

What is this? Car/Not a car

Computer Vision: Car detection

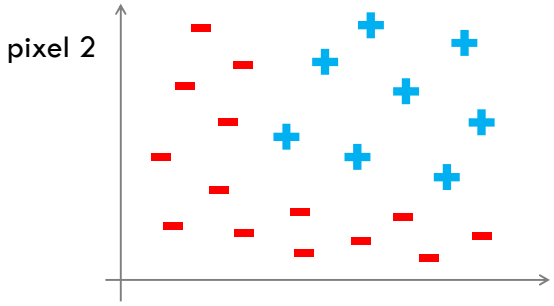


Learning Algorithm

- In order to see why we need a non-linear hypothesis (classifier function), let's see an example.
- Let's say we want to classify cars based on just 2 pixels (features).
- Assume the size of the pictures are same and we pick the same 2 pixels from images.
- We place the images on the graph, based on the intensity of the 2 pixels in each image.

Computer Vision: Car detection

Learning Algorithm

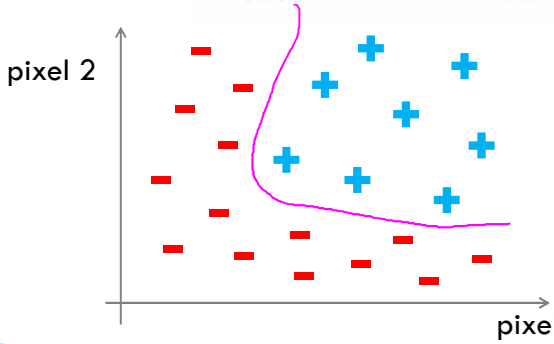


- Cars
- "Non"-Cars

- If we do this for more and more images, perhaps we find out the cars and non-cars fall in different regions of the space.
- So we need a non-linear hypothesis (classifier function) to separate these two classes.

Computer Vision: Car detection

Learning Algorithm



- Cars
- "Non"-Cars

50 x 50 pixel images→ 2500 pixels

$n = 2500$ (7500 if RGB)

$$x = \begin{bmatrix} \text{pixel 1 intensity} \\ \text{pixel 2 intensity} \\ \vdots \\ \text{pixel 2500 intensity} \end{bmatrix}$$

Each pixel represented by a number between 0-255.

If we wanted to find a hypothesis using all quadratic features of the form ($x_i * x_j$), we would need almost "3 million" parameters (θ_0, ...) to find and store (just for gray scale).

Regression method not practical

- In the presented example, we need to find almost 3 million values, so that we can draw the boundry between classes $O(n^2/2)$ or $O(n^2)$.
- So finding a boundry using curve fitting methods won't work (computationally) for solving complex problems with large number of features.
- In this lecture we are going to learn about Neural Networks, which are a much better way to find non-linear hypothesis even when we have large number of features.

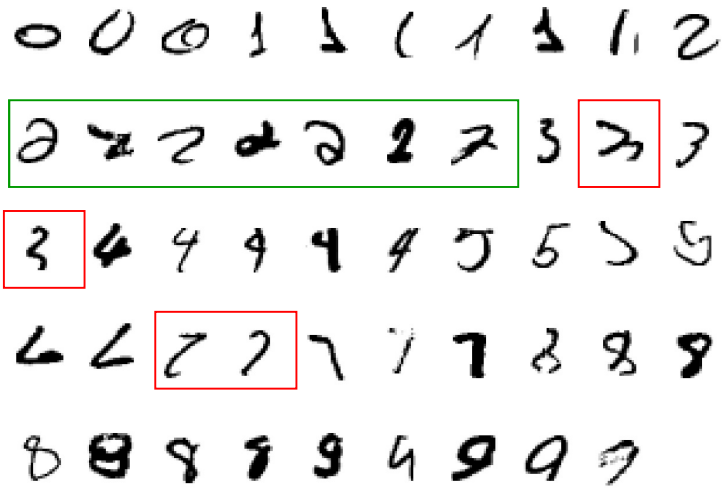
A standard example of machine learning – Data Sets

- In order to determine the performance of different algorithms, we use specific datasets.
- If researchers use the same data set, we can compare the results…

MNIST Dataset:

- The MNIST database of hand-written digits is the machine learning equivalent of fruit flies. So hand-writing recognition is a standarad example of machine learning.
  - They are publicly available and we can learn them quite fast in a moderate-sized neural net.
  - We know a huge amount about how well various machine learning methods do on MNIST.

It is very hard to say what makes a 2

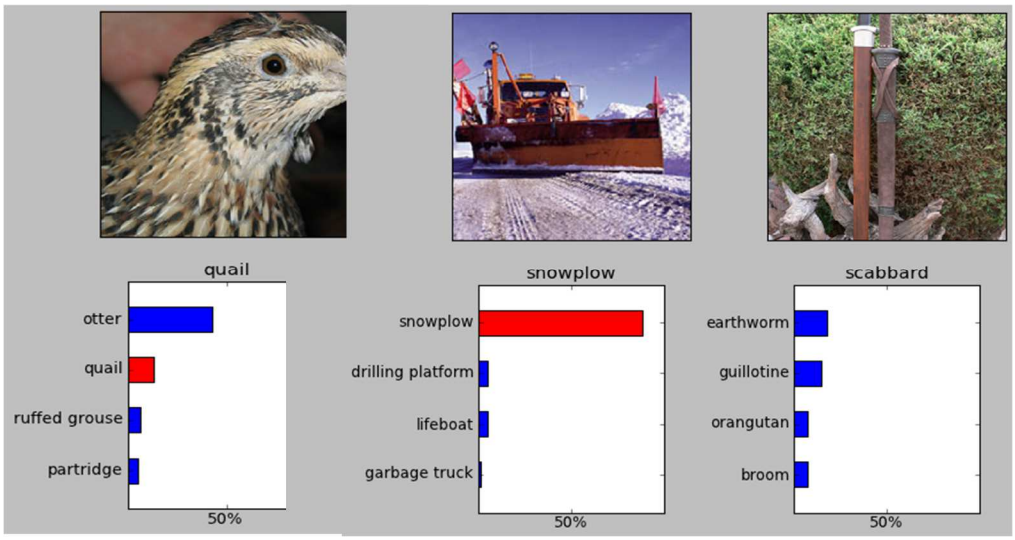


Beyond MNIST: The ImageNet task

ImageNet Dataset:

- 1000 different object classes
- 1.3 million high-resolution training images
- From the web
  - Best system in 2010 competition got 47% error for its first choice and 25% error for its top 5 choices.
- Jitendra Malik (an eminent neural net sceptic) said that this competition is a good test of whether deep neural networks work well for object recognition.
  - A very deep neural net (Krizhevsky et. al. 2012) gets less than 40% error for its first choice and less than 20% for its top 5 choices.

Some examples from an earlier version of the net



It can deal with a wide range of objects

The ImageNet, Deep Neural Networks: Better than Humans

Microsoft Software Recognizes Images Better Than Humans Do

AND IT'S A FOUNDATIONAL STEP IN THE NEXT WAVE OF INTERACTION.

2 NOTES 7 PIN 76 PLUS 236 SHARE 452 TWEET 920 LIKE

It's not the most gorgeous photography you've ever seen. ImageNet features 1.2 million pictures of mundane items—a photocopier shoved in the corner of an office, a bowl of oatmeal on a table, a pile of logs, a giant sign shaped like an ear of corn, an elbow. But ImageNet is important: It's the central collection of images scientists around the world use to teach their software image recognition, and then test it, too.

Every year, algorithms get better at identifying what's in these images. But Microsoft Research has just announced a major milestone: Its software was able to identify the contents of 100,000 test images in ImageNet with a 4.94% error rate, while humans have scored a 5.1% error rate in the same test in the past. In other words, Microsoft hasn't just beaten every competitor in the industry; they've beaten humans at their own game.

http://www.fastcodesign.com/3042239/microsoft-software-recognizes-images-better-than-humans-do

February 11, 2015

The Speech Recognition Task

- A speech recognition system has several stages:
  - **Pre-processing:** Convert the sound wave into a vector of acoustic coefficients. Extract a new vector about every 10 milliseconds.
  - **The acoustic model:** Use a few adjacent vectors of acoustic coefficients to place bets on which part of which phoneme is being spoken (classification to phonemes).
  - **Decoding:** Find the sequence of bets that does the best job of fitting the acoustic data and also fitting a model of the kinds of things people say (classification to words and statements).
- Deep neural networks pioneered by George Dahl and Abdel-rahman Mohamed are now replacing the previous machine learning method for the acoustic model. The initial work could reach only 20% speaker independent error.

Word error rates from MSR, IBM, & Google

- (Hinton et. al. IEEE Signal Processing Magazine, Nov 2012)

The task	Hours of training data	Deep neural network	Gaussian Mixture Model	GMM with more data
Switchboard (Microsoft Research)	309	18.5%	27.4%	18.6% (2000 hrs.)
English broadcast news (IBM)	50	17.5%	18.8%	
Google voice search (android 4.1)	5,870	12.3% (and falling)		16.0% (>>5,870 hrs.)

Microsoft's speech recognition is now just as accurate as humans

According to a paper published yesterday, a team of Microsoft engineers in the Artificial Intelligence and Research division reported their system reached a word error rate (WER) of 5.9 percent, a figure that is roughly equal to that of human abilities. "We've reached human parity," said Xuedong Huang, the company's chief speech scientist. "This is a historic achievement."

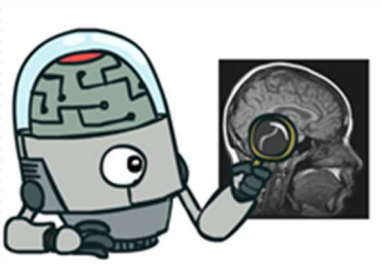
After decades of testing, the milestone comes on the heels of last month's 'close but no cigar' score of 6.3 WER and figures to have wide-reaching implications as the battle for digital assistant supremacy heats up. Cortana , Xbox , and Windows could see the biggest initial impact. To achieve these levels of accuracy, researchers employed deep neural networks to store significant amounts of data — called training sets — that helped systems recognize patterns from human input. Sounds and images were both used to train the network to utilize its stored data more efficiently. Researchers want to be clear that parity is far from perfection. In this case, it just means it's as good as humans, and we're far from flawless.

Moving forward, the team hopes to achieve even higher levels of accuracy as well as ensure that speech recognition works better in real-world situations, such as noisy restaurants, crowded streets, and in powerful winds. In the future, the team dreams of a system that will not just recognize speech, but truly understand it.

We're still a ways off, but the future consists of a world where we no longer have to understand computers, they have to understand us.

http://thenextweb.com/microsoft/2016/10/18/microsofts-speech-recognition-is-now-just-as-accurate-as-humans/
October 25, 2016

Neurons and the Brain

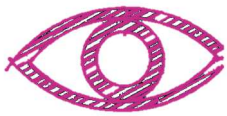
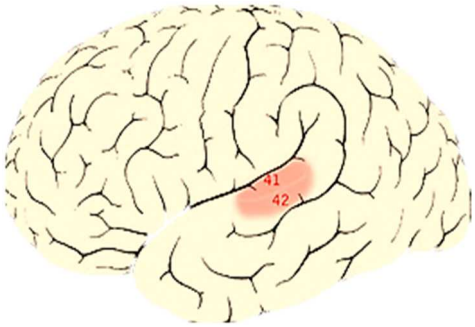


Neural Networks

- **Origins:** scientists tried to create algorithms that mimic the brain.
- **First period:** widely used in "80s-early 90s". Popularity diminished in late 90s because
  - Calculating the network parameters (i.e. weights) needed too much computation.
  - Simple networks (i.e. few layers) would not provide good accuracy.
- **Recent resurgence:** new techniques allowed building larger networks for complicated applications
  - New training techniques were developed
  - Larger memories became available
  - Parallel programming and graphic cards (GPGPUs) were used to speed up the training process
  - Specialized hardware were developed for neural networks
  - Deep neural networks were invented and used, which show huge performance in very difficult tasks

The "one learning algorithm" hypothesis

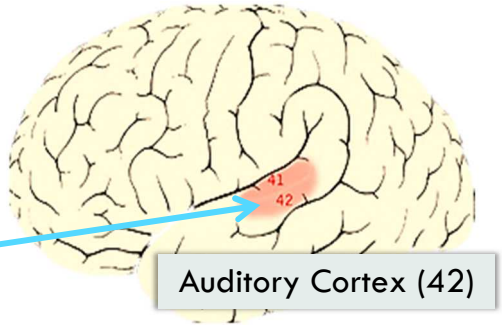
- **Hypothesis:** the brain does not use thousands of different algorithms for each of the things we do with our brains (e.g. solve maths, process images, understand voice)
  - Rather it uses a "single learning algorithm


[Roe et al., 1992]

The "one learning algorithm" hypothesis

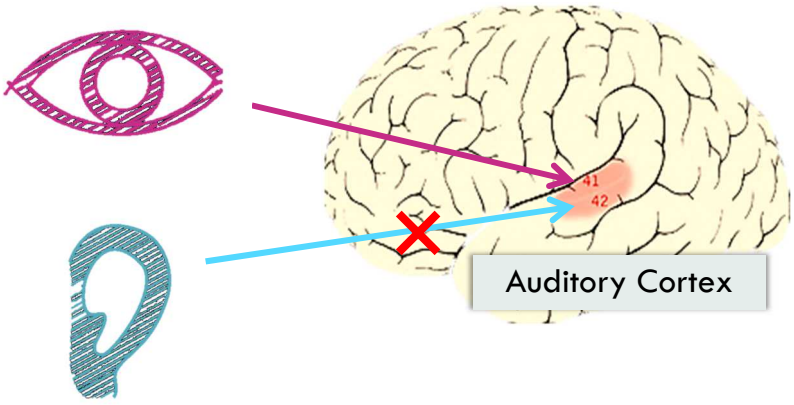
- Different parts of the brain have been identified to do specific tasks.
- **Auditory cortex:** is the part of brain which receives the auditory signals (sensory information) and allows us to understand voices

[Roe et al., 1992]

The "one learning algorithm" hypothesis

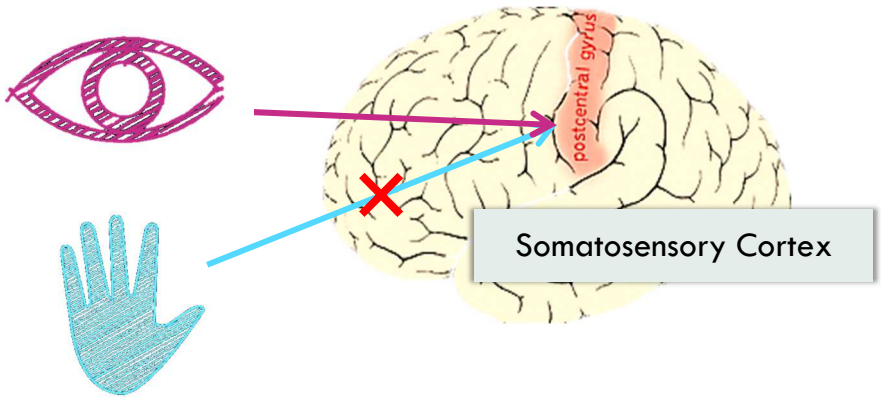
- **An experiment by Neuro-scientists (on animals):**
  - The signals coming from ears to auditory cortex were disconnected. The signals coming from eyes were rewired to that cortex
  - After a while the cortex learns to see (exactly like the animal could see before)



[Roe et al., 1992]

The "one learning algorithm" hypothesis

- The neuro-rewiring experiment was repeated with Somatosensory (touch) section of the mind and the signals from eyes were connected to that part of the brain and it could learn to see again.



[Metin & Frost, 1989]

Modularity and the brain

- **Brain structure:** different bits (places) of the brain cortex do different things, but the brain cells are the same all over the brain.
- **Neurons:** general purpose computing cells of brain (similar to FPGAs).
  - They can perform tasks in parallel manner.
- **The number:** Most neurons (out of 100 billion) are in brain.
- **Neurons die:** most of the neurons are not replaced when they die. We continue to learn and work despite the continuous loss of neurons.
- **Damage tolerance:** similar to the brain ANNs are insensitive to small damages

The "one learning algorithm" hypothesis

- So it seems we can plug in any type of sensory data to brain, and it can learn to deal with that...

A few other examples (next page):

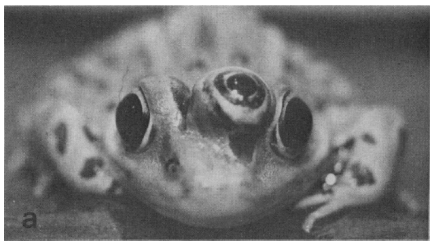
- **Upper left:** Brain port, gets a picture using a camera and converts the low pixel gray image to an array of electrical signals (with different intensities) that are transfered to the surface of your tounge. If you close your mind, after a while you can see with the instrument.
- **Top-right:** Estimating the distance and finding obstacles by clapping (or making noises from mouth), blinds have used this method
- **Bottom-left:** A direction sensor producing different buzzing sounds can give a human sense of direction
- **Bottom-right:** If you plug a third eye into a frog, it can learn to see with it.

Sensor representations in the brain

Seeing with your tongue

Human echolocation (sonar)

Haptic belt: Direction sense



Implanting a 3rd eye

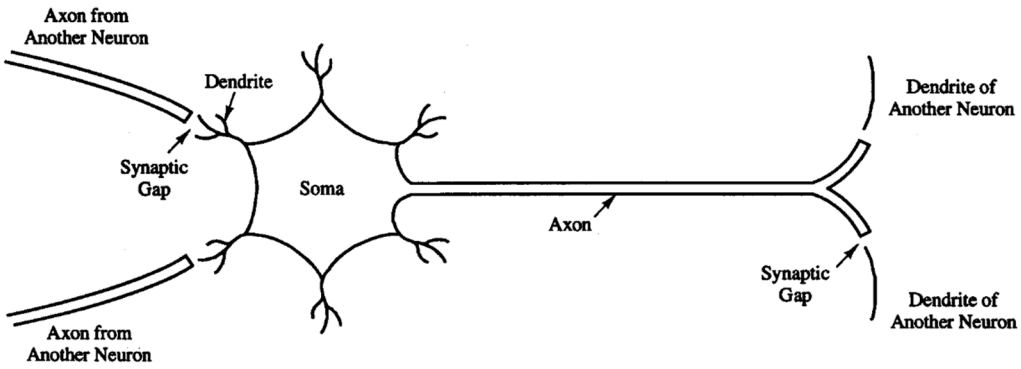
[BrainPort; Welsh & Blasch, 1997; Nagel et al., 2005; Constantine-Paton & Law, 2009]

Artificial Neural Networks

- Instead of writing thousands of different programs, we might be able to figure out how the brain does all these wonderful things and build an approximation of it to solve our problems.
- What we build should automatically learn from data.
- Perhaps this idea finally brings us to the dream of building AI.
- In this section we study how the brain works...

Neuron in the brain

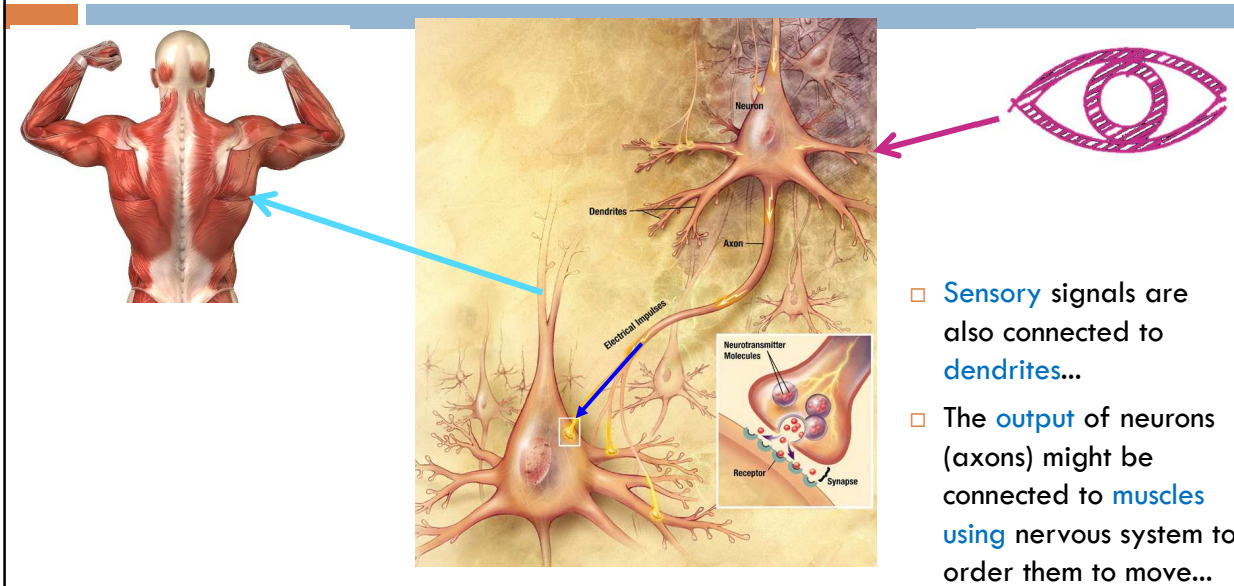
- Brain is packed of cells called nurons.
- A single nuron in the brain comprises of a Nucleus (or Soma)
- Input wires called dendrites, and outputs called axons.
- Axons are used to send signals to other neurons.



Neuron

- So a simplistic view of a neuron is a computing unit, which receives some inputs (in the form of electrical signals), and does some calculations on the input data.
- It then sends the output to other nodes (other neurons) through its axons (again in the form of electrical signals).
- This is basically the process with which the human thoughts happen
  - Neurons processing inputs and sending messages to other neurons.
- Neurons in the brain have a response time of around 10^{-3} seconds in comparison to a 10^{-9} seconds response time in semiconductors… but brain is a massively parallel structure that uses almost 10 billions of nerons to process information.

Neurons in the brain

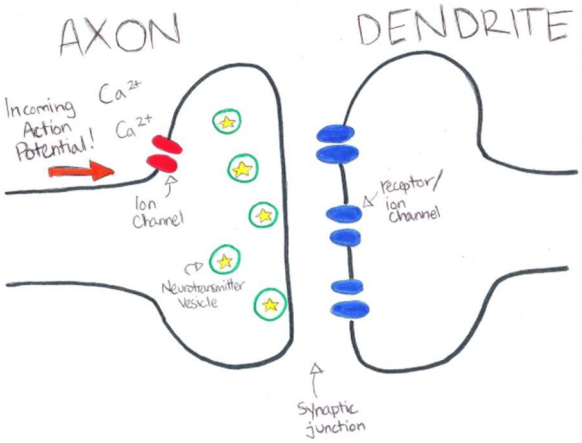


- Sensory signals are also connected to dendrites...
- The output of neurons (axons) might be connected to muscles using nervous system to order them to move...

[Credit: US National Institutes of Health, National Institute on Aging]

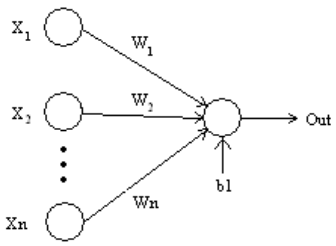
Synapses

- Axons typically contact dendritic trees at synapses
- A spike of activity in the axon, causes charge to be transmitted across synaptic gap (by means of chemical process and ions) into the dendrites of the next level neurons.



How the brain works on one slide!

- The effect of each input line on the neuron is controlled by a synaptic weight. The weights can be positive or negative.
- Synaptic weights are modified by experience (learning) so that the whole network can perform useful computations:
  - Recognizing objects, understanding language, making plans, controlling the body.
- You have about 10^{11} neurons each with about 10^4 weights.
  - A huge number of weights (10^{15}) can affect the computation in a very short time. Much better bandwidth than a workstation.

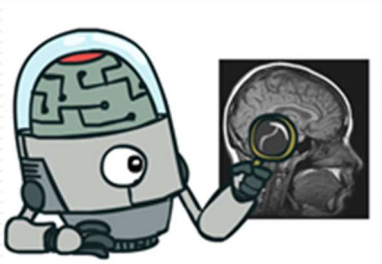


IBM's Brain-Inspired Computer Chip

The IBM neuro-synaptic computer chip consists of 1 million programmable neurons and 256 million programmable synapses conveying signals between the digital neurons. Each of chip's 4,096 neuro-synaptic cores includes the entire computing package—memory, computation, and communication. They all operate in parallel based on "event-driven" computing, similar to the signal spikes and cascades of activity when human brain cells work in concert. Such architecture helps to bypass the bottleneck in traditional computing where program instructions and operation data cannot pass through the same route simultaneously.

http://spectrum.ieee.org/tech-talk/computing/hardware/ibms-braininspired-computer-chip-comes-from-the-future
Spectrum August 2014

Models of Single Neurons



Abstract neurons

- To model things, we abstract and simplify them
  - Abstraction removes complicated details that are not essential for understanding the main principles.
  - After understanding the basics, its easy to add details to make the model more faithful
- Building models that are not accurate is still useful (we must remember that they are wrong!)
  - For example, neurons that communicate real values rather than discrete spikes of activity.

Linear neurons

- These are simple but computationally limited
  - If we can make them learn we may get insight into more complicated neurons.

$$y = b + \sum_i x_i w_i$$

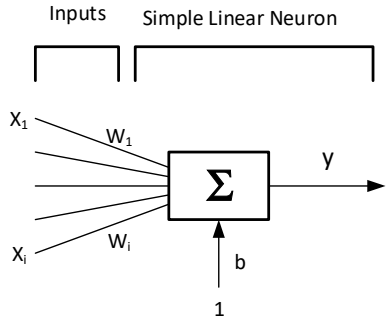
bias

i_{th} input

output

index over input connections

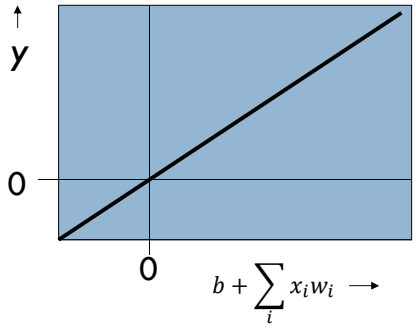
weight of the i_{th} input



Linear neurons

- If we plot that, we will have the below graph.
- The output value continiously increases with the input

$$y = b + \sum_i x_i w_i$$

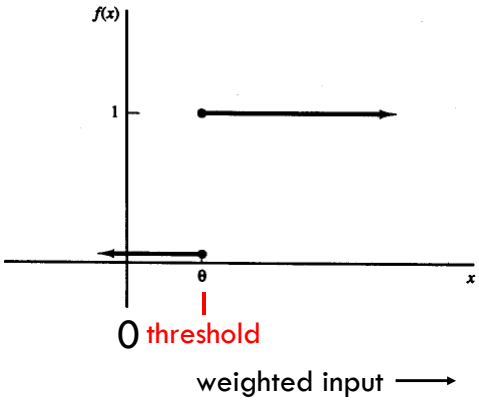


- Linear neurons are suitable for Linear regression (fitting a straight line to some data) among other usages …

Binary threshold neurons (McCulloch-Pitts Neuron)

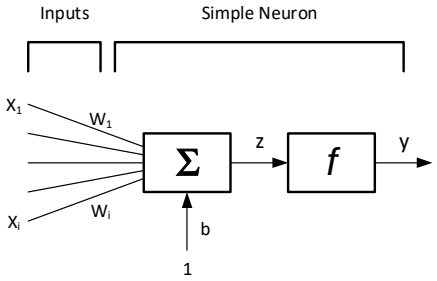
- **McCulloch-Pitts Neuron (1943):**
  1. First compute a weighted sum of the inputs.
  2. Then send out a fixed size spike of activity if the weighted sum exceeds a threshold.

  - McCulloch and Pitts (influenced Von Neumann) thought that each spike is like the truth value of a logical proposition, and each neuron combines truth values to compute the truth value of another proposition!



Binary threshold neurons

- There are two equivalent ways to write the equations for a binary threshold neuron:
- First formulation is just like the second (if we assume θ=-b)
  - θ and b determine the threshold



$$z = \sum_i x_i w_i \qquad y = \begin{cases} 1, & if\ z \geq \theta \\ 0, & otherwise \end{cases}$$

θ=-b

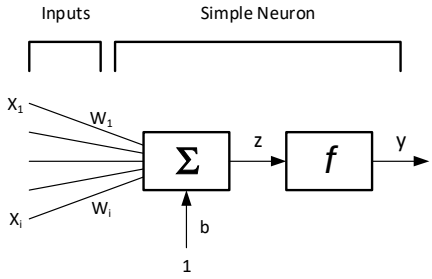
$$z = b + \sum_i x_i w_i \qquad y = \begin{cases} 1, & if\ z \geq 0 \\ 0, & otherwise \end{cases}$$

Rectified Linear Neurons – ReLU (linear threshold neurons)

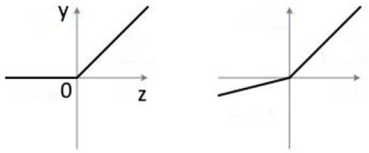
- They compute a linear weighted sum of their inputs.
- The output is a non-linear function of the total input.

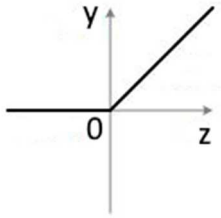
$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} z, & if\ z > 0 \\ 0, & otherwise \end{cases}$$



- **ReLU vs. PReLU:**
  for PReLU, the coefficient of the negative part is not constant and is adaptively learned.



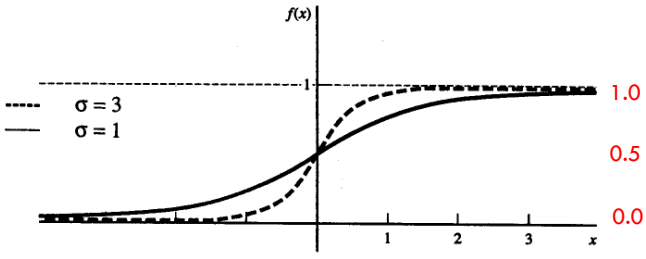


Sigmoid (Logistic) neurons

- This neuron gives a real-valued, smooth and bounded output.
  - Logistic sigmoid (or binary sigmoid) is a function that produces a value of (0,1)
  - The function has nice derivatives (which can easily be calculated from the function itself). This makes learning process easy.

$$f'(x) = \sigma f(x)[1 - f(x)]$$

  - Steepness depends on σ. Larger σ produces more steep function.



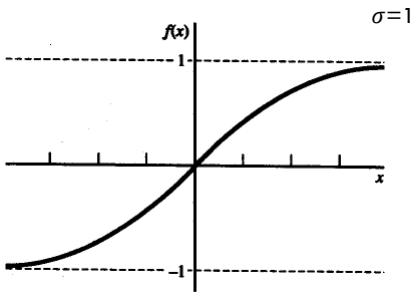
$$x = b + \sum_i x_i w_i \quad y = \frac{1}{1 + e^{-\sigma x}}$$

Sigmoid (Bipolar and Hyperbolic Tangent) neurons

- **Bipolar sigmoid:** Logistic sigmoid can be scaled to have any range, such as -1 and 1

$$g(x) = 2f(x) - 1 = \frac{2}{1 + e^{-\sigma x}} = \frac{1 - e^{-\sigma x}}{1 + e^{-\sigma x}}$$

$$g'(x) = \frac{\sigma}{2}[1 + g(x)][1 - g(x)]$$



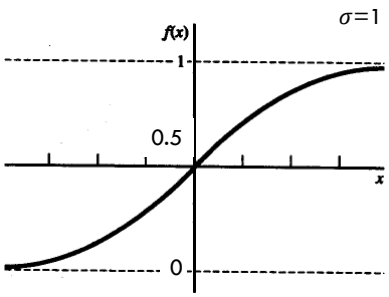
- **Hyperbolic tangent (Tanh):** is bipolar sigmoid function with σ=2

$$h(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

$$x = b + \sum_i x_i w_i \; h(x) = \frac{1 - e^{-\sigma x}}{1 + e^{-\sigma x}}$$

Stochastic binary neurons

- Same equations as logistic units
  - But they treat the output of the logistic as the probability of producing a spike in a short time window. So the output is a 0 or 1 (depending on the probability and a random decision)



- Can do a similar trick with rectified linear neuron:
  - The output is treated as the Poisson rate for spikes. The rate of producing spike is calculated.

$$x = b + \sum_i x_i w_i \quad y = \frac{1}{1 + e^{-\sigma x}}$$

Neuron Model

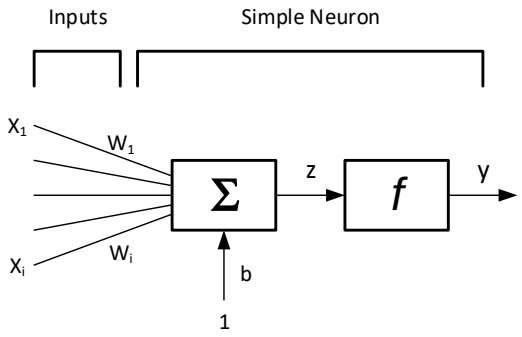
$x_0 = 1, w_0 = b$ bias

$X = \begin{bmatrix} x_1 \\ \dots \\ x_n \end{bmatrix}$ inputs

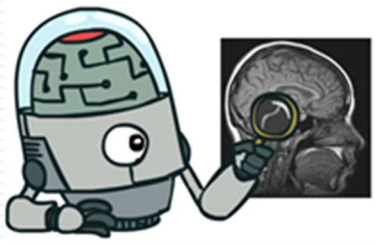
$W = \begin{bmatrix} w_1 \\ \dots \\ w_n \end{bmatrix}$ weights

$y = f(z) = f(WX + x_0.w_0)$ outputs

f threshold function

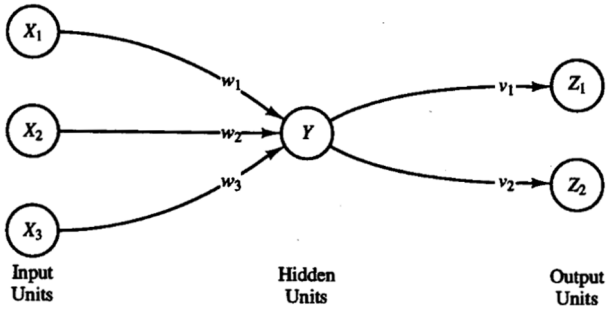


Networks



Feed-forward neural networks (MLP: Multilayer Perceptron)

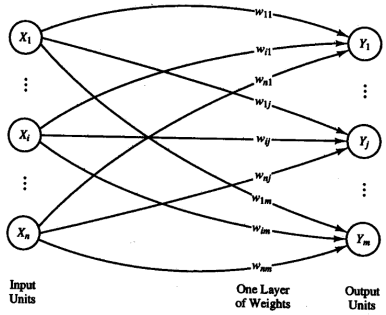
- The image below shows a very simple neural network with just one hidden layer.
  - Presence of a hidden unit (with a non-linear activation function), gives it the ability to solve many more problems (non-linear) that can be solved in comparison to a network with only input and output units.
  - But the training of the network (finding optimal weights) is now more difficult.



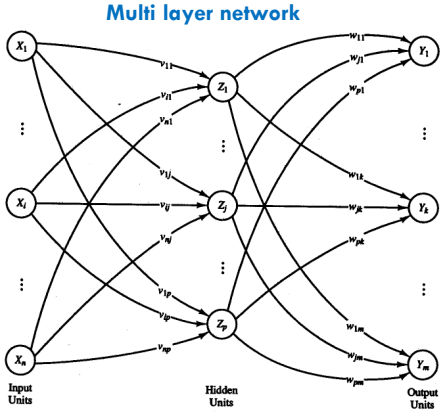
Feed-forward neural networks (MLP: Multilayer Perceptron)

- Within each layer, neurons usually have the same activation function and the same pattern of connection to other neurons.
- The input layers are not counted as a layer because they don't perform computation.

Single layer (since input is not counted)

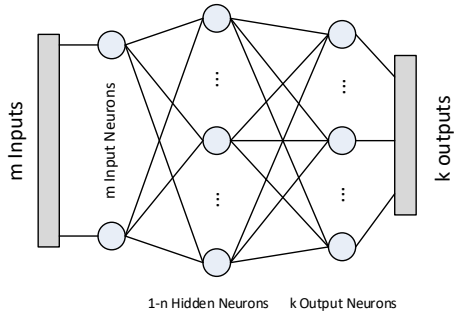


Multi layer network



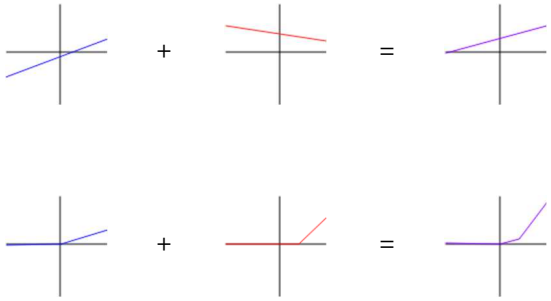
Feed-forward neural networks (MLP: Multilayer Perceptron)

- These are the most common type of neural network in practical applications.
  - The first layer is the input and the last layer is the output.
  - The data flows forward from inputs to hidden layer neurons and finally to outputs.
  - If there are too many hidden layers, we call them "deep" neural networks (they are a bit different in structure though).
- They compute a series of transformations that change the similarities between cases.
  - The activities of the neurons in each layer are a non-linear function of the activities in the layer behind.



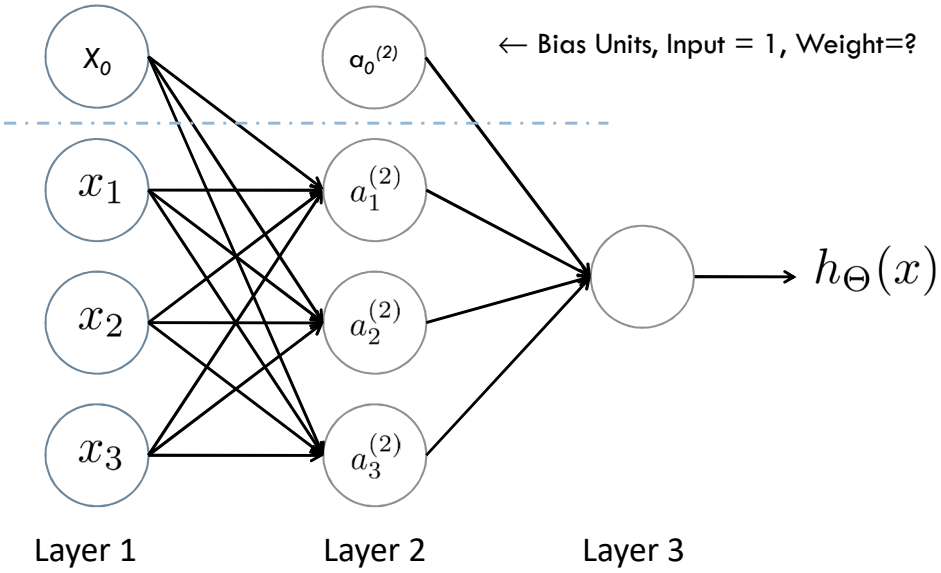
Feed-forward neural networks - Why non-linear layers

- Activation function for input units is the **identity function**.
- In order to achieve advantages of multi-layer (compared with single-layer) nets, non-linear activation functions are used.
  - Using two or more layers of linear processing is no different from using a single layer of linear function.

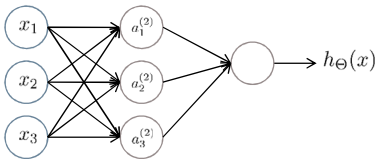


Multi-layer Neural Network - Weights and Output

- Inputs flow forward and simple calculations (multiplication and summation) are performed.
- The difficult step is the calculation of weights
- Bias units are independent inputs with a value of 1 (the weight connecting them to each neuron differ)



Multi-layer Neural Network - Calculation of Output



$a_i^{(j)}$ = "activation" of unit i in layer j

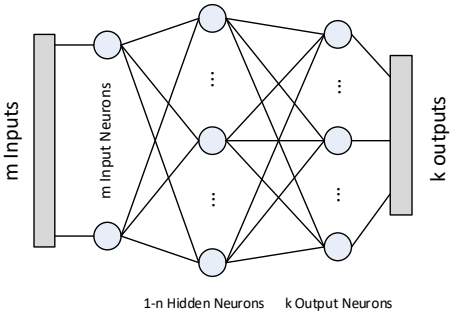
$\Theta^{(j)}$ = matrix of weights controlling function mapping from layer j+1 to layer j

$$a_1^{(2)} = g(\Theta_{10}^{(1)}x_0 + \Theta_{11}^{(1)}x_1 + \Theta_{12}^{(1)}x_2 + \Theta_{13}^{(1)}x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)}x_0 + \Theta_{21}^{(1)}x_1 + \Theta_{22}^{(1)}x_2 + \Theta_{23}^{(1)}x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)}x_0 + \Theta_{31}^{(1)}x_1 + \Theta_{32}^{(1)}x_2 + \Theta_{33}^{(1)}x_3)$$
$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)}a_0^{(2)} + \Theta_{11}^{(2)}a_1^{(2)} + \Theta_{12}^{(2)}a_2^{(2)} + \Theta_{13}^{(2)}a_3^{(2)})$$

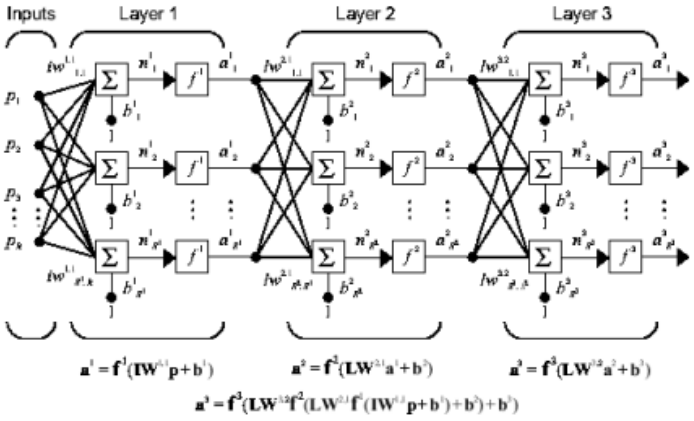
If network has s_j units in layer j, S_{j+1} units in layer j+1, then $\theta^{(j)}$ will be of dimension $s_{j+1} \times (s_j + 1)$

Multi-layer Feedforward Network

2 (or 3) layers: 1 input, 1 hidden, 1 output



n layers: 1 input, n-2 hidden, 1 output



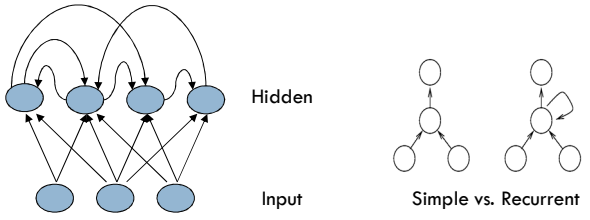
$$\mathbf{a}^1 = \mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p}+\mathbf{b}^1) \qquad \mathbf{a}^2 = \mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{a}^1+\mathbf{b}^2) \qquad \mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{a}^2+\mathbf{b}^3)$$

$$\mathbf{a}^3 = \mathbf{f}^3(\mathbf{LW}^{3,2}\mathbf{f}^2(\mathbf{LW}^{2,1}\mathbf{f}^1(\mathbf{IW}^{1,1}\mathbf{p}+\mathbf{b}^1)+\mathbf{b}^2)+\mathbf{b}^3)$$

Recurrent networks

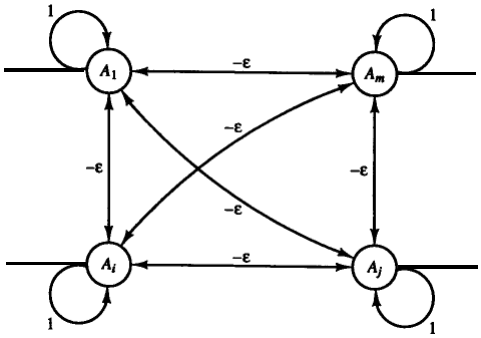
- These have directed cycles in their connection graph (feedbacks).
  - That means you can sometimes get back to where you started by following the arrows. So information can run in cycles.
  - As a result they can show dynamic temporal behaviors.
  - The output of each neuron is determined for succeeding time steps (time slices)
- They can have complicated dynamics and this can make them very difficult to train.
  - There is a lot of interest at present in finding efficient ways of training recurrent nets.
- They are more biologically realistic.

In recurrent nets with multiple hidden layers some of the hidden→hidden connections might be missing.



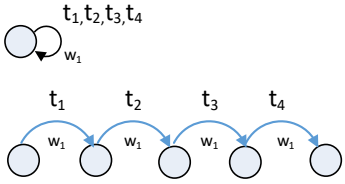
Recurrent networks

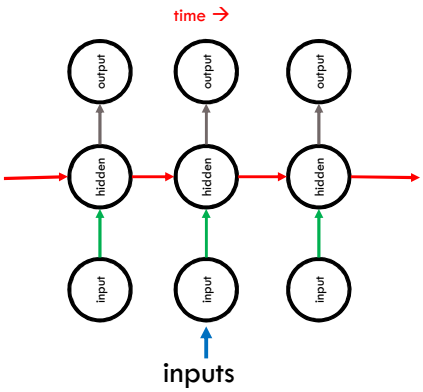
- The picture shows another recurrent network. In this network output from every neuron connects to all others (even itself).
- The competitive interconnections have weights of –e
- The connections compete to determine the result (output) of each neuron
- This network is a symmetrically connected recurrent network…



Recurrent neural networks for modeling sequences

- Recurrent neural networks are suitable way to model sequential data:
  - They are equivalent to very deep nets with one hidden layer per time slice.
  - Except that they use the same weights at every time slice and they get input at every time slice
- They have the ability to remember information in their hidden state for a long time (i.e. learning from historical data).
  - But its very hard to train them to use this potential.





An example of what recurrent neural nets can now do

- Ilya Sutskever (2011) trained a special type of recurrent neural net to predict the next character in a sequence.
- After training for a long time on a string of half a billion characters from English Wikipedia, he got it to generate new text.
  - It generates by predicting the probability distribution for the next character and then sampling a character from this distribution.
  - It actually uses 86 characters to show alphabets, punctuations etc.
  - The next slide shows an example of the kind of text it generates. Notice how much it knows!

Sutskever, Ilya, James Martens, and Geoffrey E. Hinton. "Generating text with recurrent neural networks." In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1017-1024. 2011.

Some text generated one character at a time

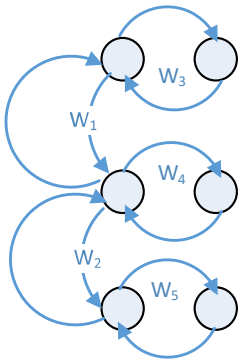
- In 1974 Northern Denver had been overshadowed by CNL, and several Irish intelligence agencies in the Mediterranean region. However, on the Victoria, Kings Hebrew stated that Charles decided to escape during an alliance. The mansion house was completed in 1882, the second in its bridge are omitted, while closing is the proton reticulum composed below it aims, such that it is the blurring of appearing on any well-paid type of box printer.

- Produced on character at a time
- Some parts actually good English
- Some parts make sense
- Syntax is very good
- There are some meaning and context issues…
- Again remember it has been produced one character at a time!

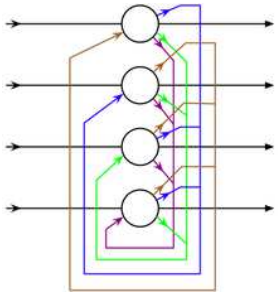
* by Ilya Sutskever's recurrent neural network

Symmetrically connected networks

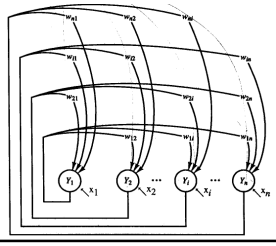
- These are like recurrent networks, but the connections between units are symmetrical (they have the same weight in both directions).
  - John Hopfield (and others) realized that symmetric networks are much easier to analyze than recurrent networks.
  - They are also more restricted in what they can do. because they obey an energy function.
    - For example, they cannot model cycles.
- Symmetrically connected nets without hidden units are called "Hopfield nets".



A single-layer net in which all units function as both input and output units.

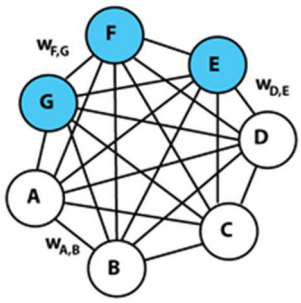


Hopfield Net (wiki)

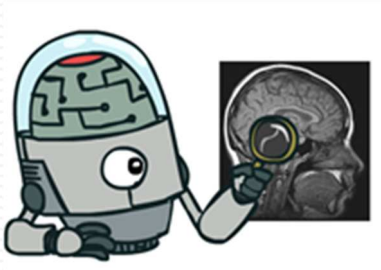


Symmetrically connected networks with hidden units

- These are called "Boltzmann machines".
  - They are much more powerful models than Hopfield nets.
  - They are less powerful than recurrent neural networks.
  - They have a beautifully simple learning algorithm.
  - White units are visible, blue ones are hidden.
- In "restricted Boltzman machines", there are only links between hidden and visible units (not between hidden or visible anymore)

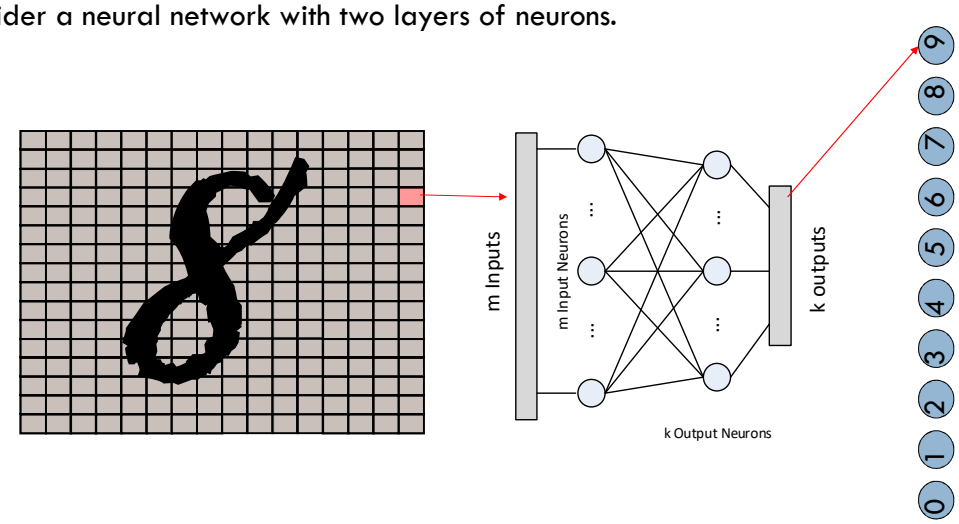


Simple Example of Learning



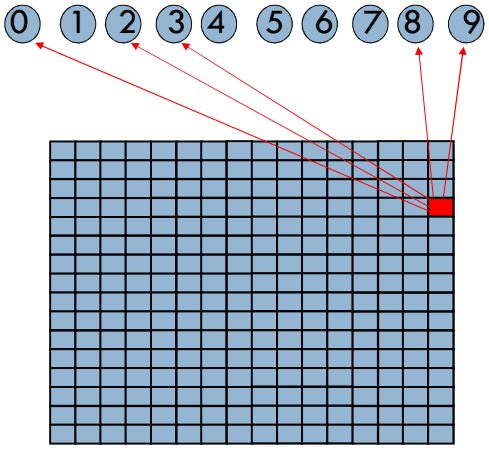
Handwriting Recognition using Neural Network

- Consider a neural network with two layers of neurons.



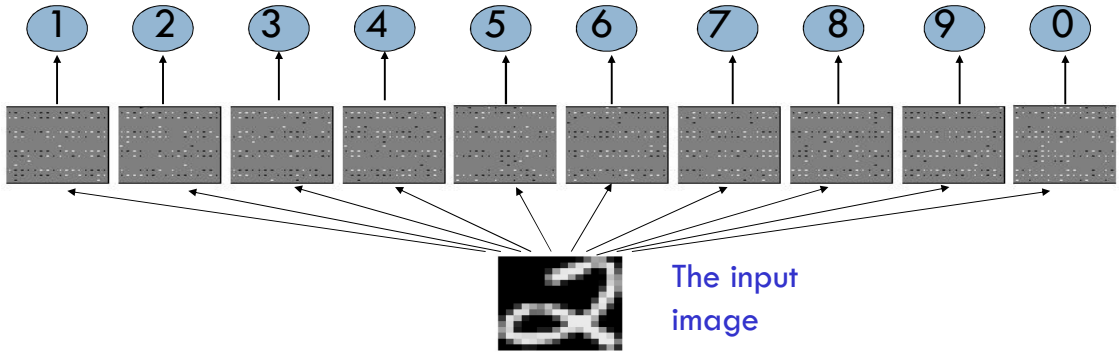
A very simple way to recognize handwritten shapes

- In the network:
  - neurons in the input layer represent pixel intensities.
  - neurons in the output layer represent known shapes.
- A pixel gets to vote for each shape if it has ink on it.
  - Each inked pixel can vote for several different shapes.
- The shape that gets the most votes wins.



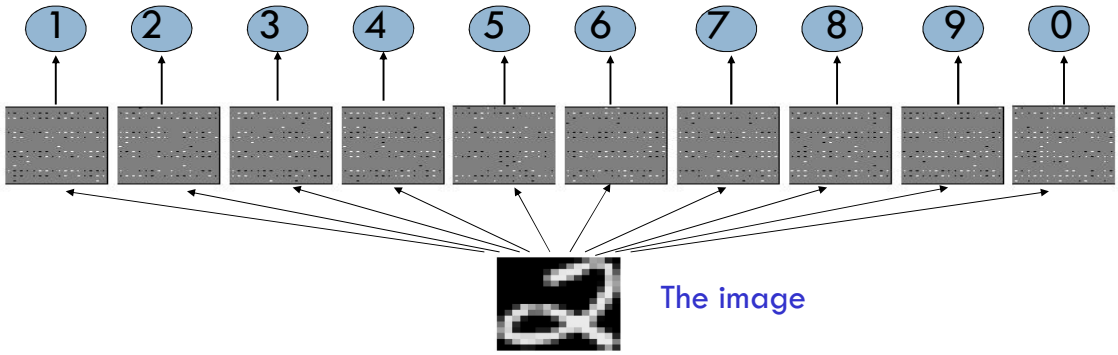
How to display the weights

- Give each output unit its own "map" of the input image and display the weight coming from each pixel in the location of that pixel in the map.
- Use a black or white blob with the area representing the magnitude of the weight and the color representing the sign.

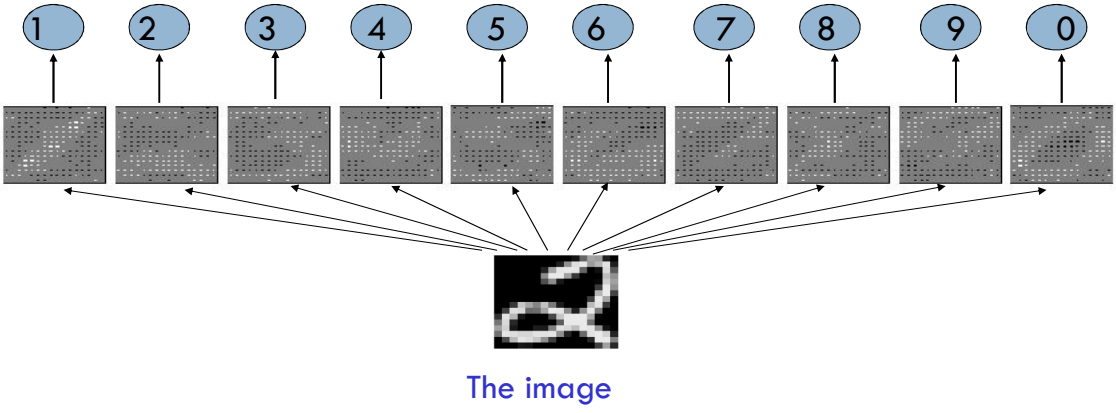


How to learn the weights

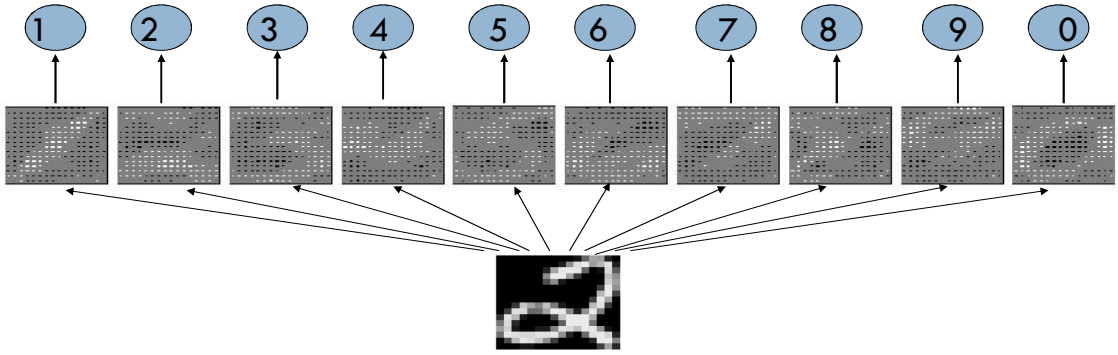
- Show the network an image and increment the weights from active pixels to the correct class.
- Then decrement the weights from active pixels to whatever class the network guesses.



Visualizing Weights

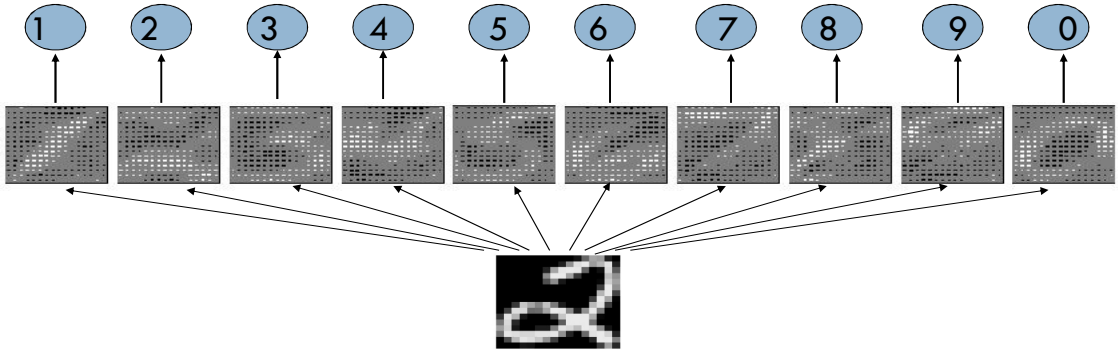


Visualizing Weights



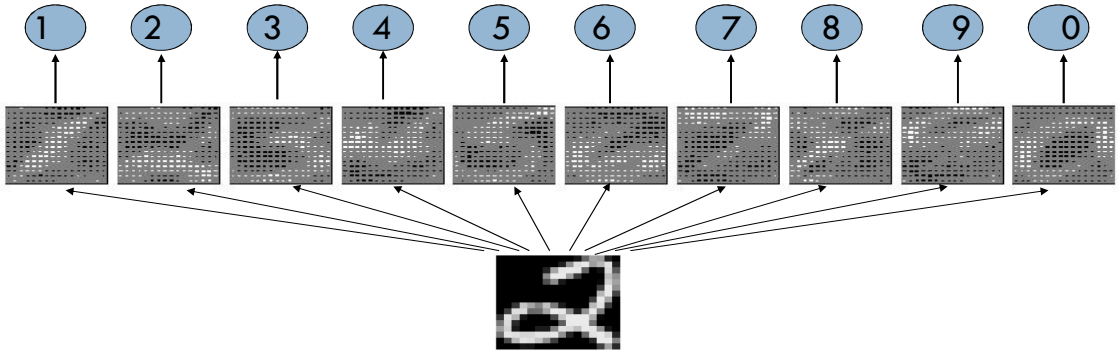
The image

Visualizing Weights



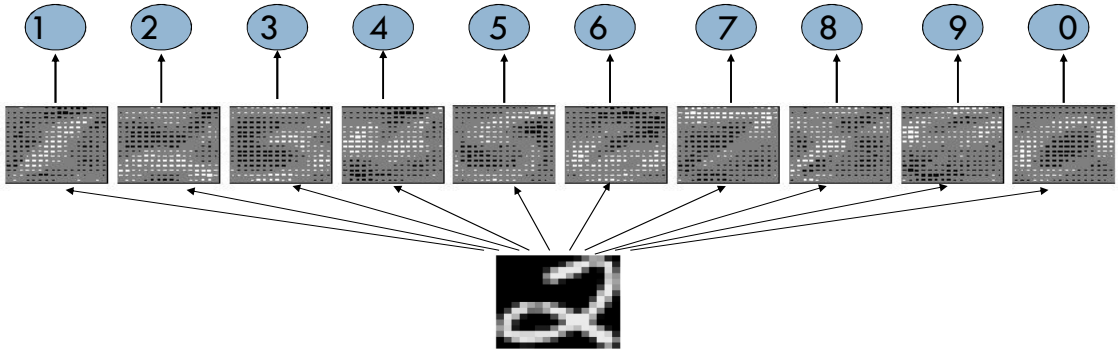
The image

Visualizing Weights



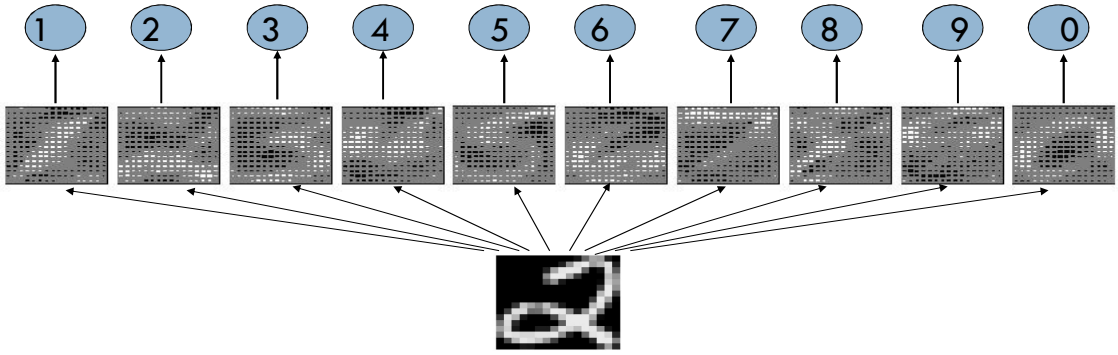
The image

Visualizing Weights



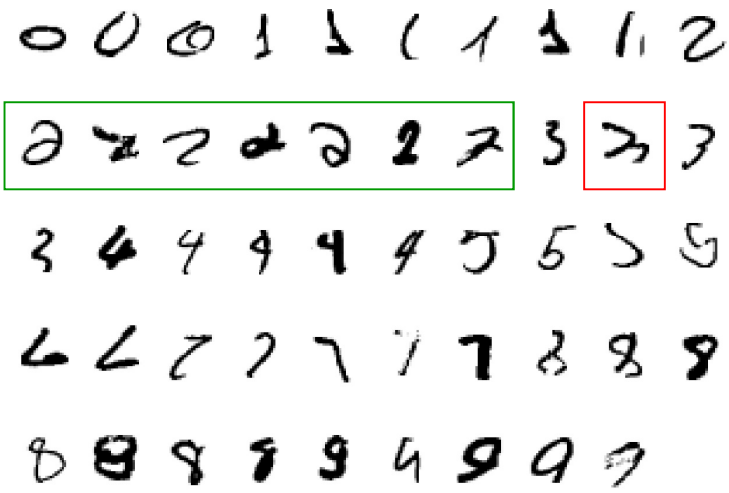
The image

The learned weights

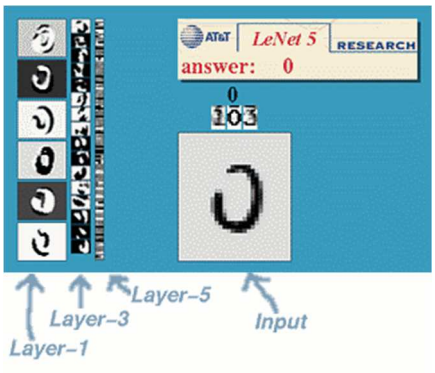


The image

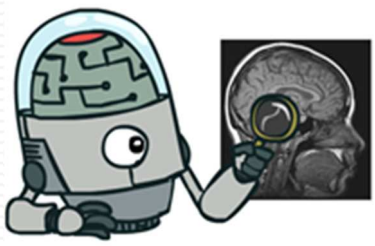
Digits recognized correctly the first time they are seen



Lenet (Yann LeCun)



First Gen. Networks: Perceptron



The history of perceptron

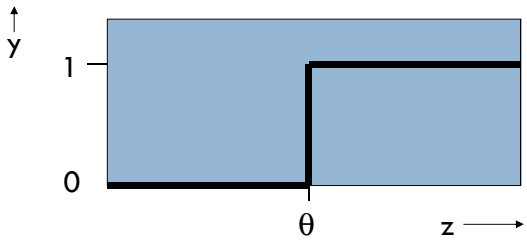
- They were popularized by Frank Rosenblatt in the early 1960's.
  - Can perform binary classification
  - They appeared to have a very powerful learning algorithm.
  - Lots of big claims were made for what they could learn to do.
- In 1969, Minsky and Papert published a book called "Perceptrons" that analyzed what they could do and showed their limitations.
  - Many people thought these limitations applied to all neural network models.
- The perceptron learning procedure is still widely used today for tasks with enormous feature vectors that contain many millions of features (e.g. google uses them).

Binary threshold neurons (decision units)

- McCulloch-Pitts (1943)
  - First compute a weighted sum of the inputs from other neurons (plus a bias).
  - Then output a 1 if the weighted sum exceeds zero (or a specific θ).
  - Notice that inputs are linearly combined, then used to make a binary classification…

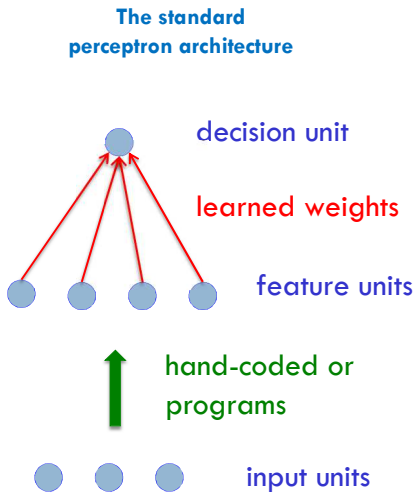
$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1, & if\ z \geq \theta \\ 0, & otherwise \end{cases}$$



The standard paradigm for statistical pattern recognition

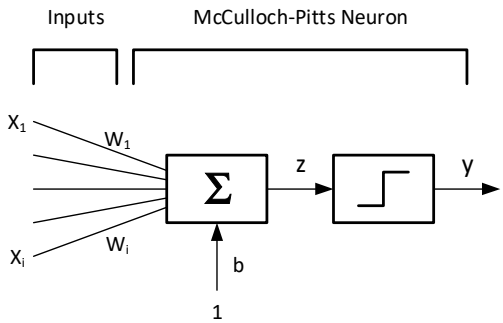
1. **Convert the raw input vector into a vector of feature activations:**
   Use hand-written programs based on common-sense to define the features (e.g. extract some features from objects in an image).
2. **Learn how to weight each of the feature activations to get a single scalar quantity:**
   how much importance each feature gets in determining whether the output is of the class you are trying to determine (in favor: +, or against: -).
3. **Binary Classification:** If this quantity is above some threshold, decide that the input vector is a positive example of the target class (i.e. it is from a specific class).



Perceptron using Neurons

- After the linear networks, the *perceptron* is the simplest type of neural network and it is typically used for classification. In the one-output case, it consists of a neuron with a step function.

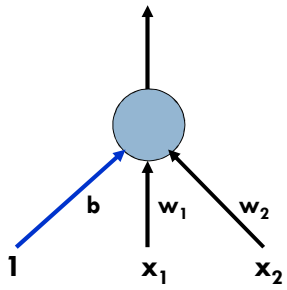
- Binary classifier
- Linear only



How to learn biases

- **Bias as a weight:** We can avoid having to figure out a separate learning rule for the bias by using a trick:
  - A bias is exactly equivalent to a weight on an extra input line that always has an activity of 1.
  - We can now learn a bias as if it were a weight.
- **Threshold:** threshold is equivalent to having a negative bias (-b).
- **Bias weight:** we add an extra component with value 1 to each input vector. The "bias" weight on this component is minus the threshold (-θ). Now we can forget the threshold.

Example: If input vector is [0.9 1.6 0.5 1.2] then convert it into [1 0.9 1.6 0.5 1.2], the first 1 is supplied to the Bias.



Perceptron convergence procedure - Training binary output neurons as classifiers

- **Initial weights:** for the initial value of weights (W), we can assign zero to all weights or just use random numbers. The weights should eventually converge to correct values.
- **Setting the weights:**
  - pick training cases using any policy that ensures that every training case will keep getting picked.
  - Now adjust the weights:
    - If the output unit is correct, leave its weights alone.
    - If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
    - If the output unit incorrectly outputs a one, subtract the input vector from the weight vector.
- **Stop the training:** this is guaranteed to find a set of weights that gets the right answer for all the training cases if any such set exists. We continue the process until we see no further improvement in the performance…

Perceptron convergence procedure - Training binary output neurons as classifiers

- **Example:** so assuming that X is an input vector, y is the output and W are the weights:
  - If y=0 or y=1 and class is correct, don't touch weights
  - If y=0 and correct class is 1: W = W + X
  - If y=1 and correct class is 0: W = W – X
- **Learning rate:** sometimes we use a η parameter to adjust the learning rate:
  - If y=0 and correct class is 1: W = W + η X
  - If y=1 and correct class is 0: W = W – η X
- In previous case η = 1 but we can adjust it to smaller value to reduce learning rate.

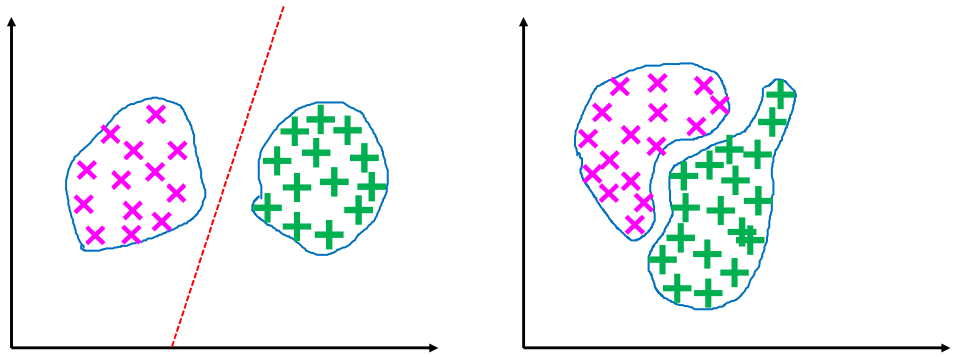
Limitations of Perceptron

The limitations of Perceptron

- In the late 1980s neural network research came to a halt:
  - It was shown that the Perceptron has limitations.
  - However most researchers did not know and understand what are those limitations.
  - They assumed Neural networks in general have serious limitations.
  - In fact those limitations only applied to Perceptron.
  - Another reason was the large amount of calculations needed to set the weights on big networks. This was later solved by the appearance of powerful CPUs and GPUs and bigger RAM memories.

The limitations of Perceptron

- The main limit is that, a perceptron can only be used "linearly separable" classes of data.



What binary threshold neurons cannot do

- A binary threshold output unit cannot even tell if two single bit features are the same!

  Positive cases (same): (1,1) → 1; (0,0) → 1

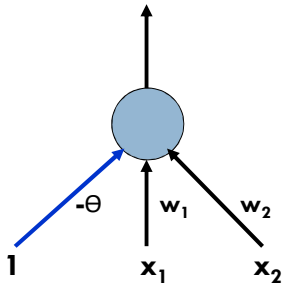
  Negative cases (different): (1,0) → 0; (0,1) → 0

- The four input-output pairs give four inequalities that are impossible to satisfy:

$w_1 * 1 + w_2 * 1 \geq \Theta$ (1), $w_1 * 0 + w_2 * 0 \geq \Theta$ (2)
$w_1 * 1 + w_2 * 0 < \Theta$ (3), $w_1 * 0 + w_2 * 1 < \Theta$ (4)

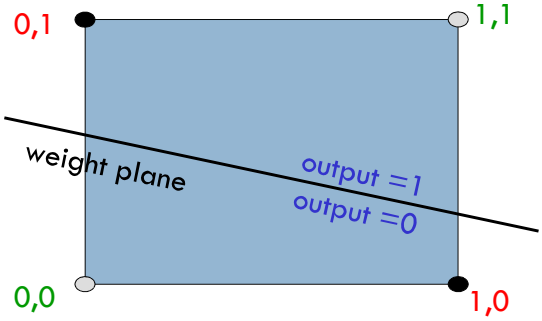
That is:
$\Theta \leq 0$ (2)
$w_1 < \Theta$ (3)
$w_2 < \Theta$ (4)
$w_1 + w_2 \geq \Theta$ (1) → $2\Theta - \Delta \geq \Theta$ (Θ neg. and Δ pos.)

Impossible to have all above inequalities altogether



A geometric view of what binary threshold neurons cannot do

- Imagine "data-space" in which the axes correspond to components of an input vector.
  - Each input vector is a point in this space.
  - A weight vector defines a plane in data-space.
  - The weight plane is perpendicular to the weight vector and misses the origin by a distance equal to the threshold.



The positive and negative cases cannot be separated by a plane

Multiclass Classification

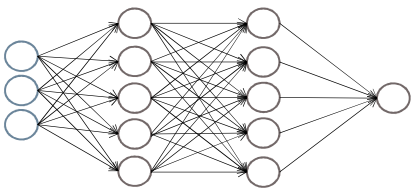
Binary and Multi-class classification

Single output unit: Binary Classification

- The output unit can be used to classify objects into two classes.
  - Car vs. Truck
  - Car vs. Not Car

Car

Truck



Multiple output units: One-vs-all.

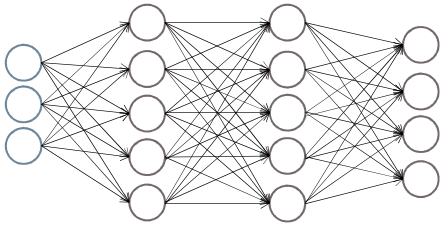

Pedestrian


Car


Motorcycle


Truck

X



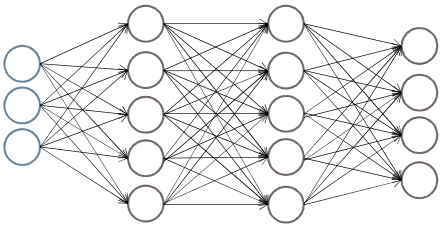
Y

Output with higher value normally represents the target class

$$Pedestrian = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} \quad Car = \begin{bmatrix}0\\1\\0\\0\end{bmatrix} \quad Motorcycle = \begin{bmatrix}0\\0\\1\\0\end{bmatrix} \quad Truck = \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$$

Multi-class classifier

X



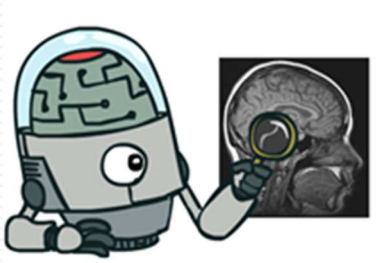
Y

$$Y \in \left\{\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}\right\}$$

Training set: (x_1,y_1), (x_2,y_2), (x_3,y_3) ….

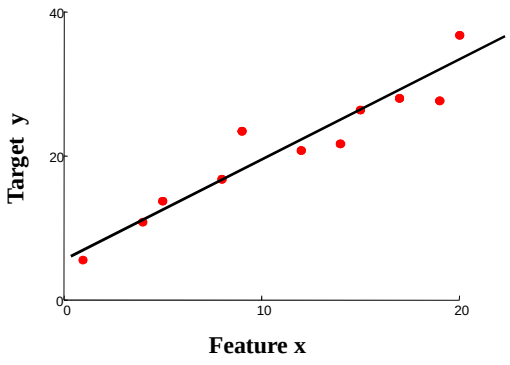
- Each x_i is a vector, that has as many components as the input neurons
- y_i is one of: $\begin{bmatrix}1\\0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\0\\1\end{bmatrix}$

Linear Neuron Learning and Linear Regression



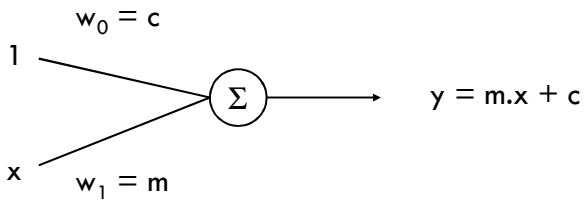
Linear regression

- Let's assume function f(x) is a good fit for the above data
- The equation of the line is y = m.x + c

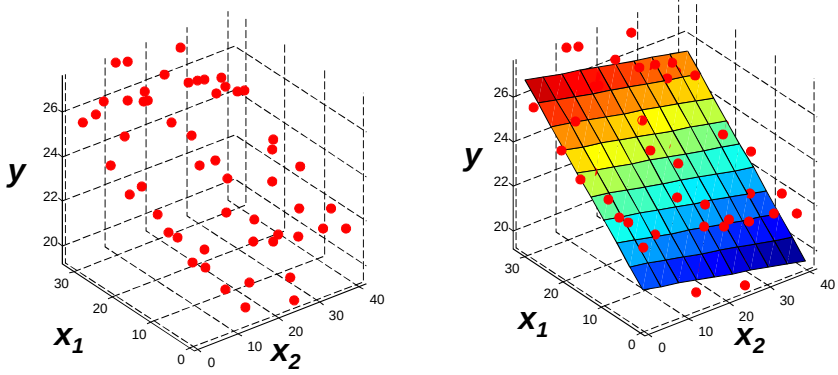


Linear regression

- We can build the line model using a single linear neuron.
- Assuming that the learning process finds proper values for w_0 and w_1 we will have a linear fit. Therefore we can use the linear neuron as a function estimator (regression function).
- We can use more than one input if we have more than one feature (and still a linear function could be fit to the data): $y = w_0 + w_1x_1 + w_2x_2 + \ldots$



More dimensions



$$y = w_0 + w_1x_1 + w_2x_2 + \ldots$$

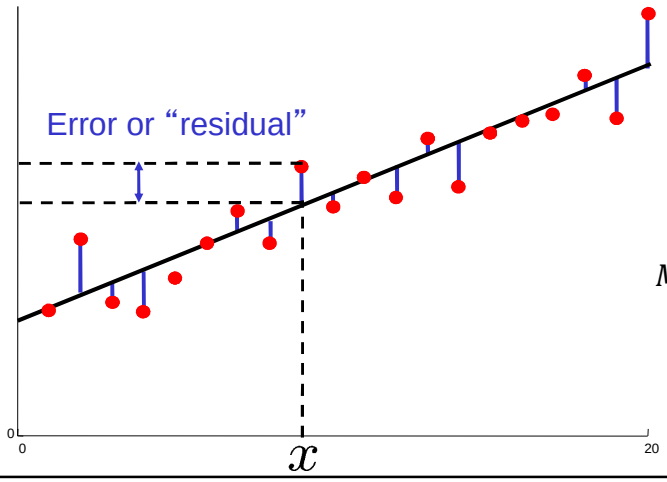
(c) Alexander Ihler

Mean Squared Error

- The fitted line predicts y values for every x, but those predicted values have errors. Since errors are sometimes positive and sometimes negative we need to either use square or abstract value to add their size only. We then calculate the average over data points.

Actual Value: y

Prediction: y'



$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y'_i - y_i)^2$$

Mean squared error

- How can we quantify the error?
  - Using MSE is computationally convenient (more later). We call it MSE cost function:

$$MSE = \frac{1}{m}\sum_{i=1}^{n}(y'_i - y_i)^2$$

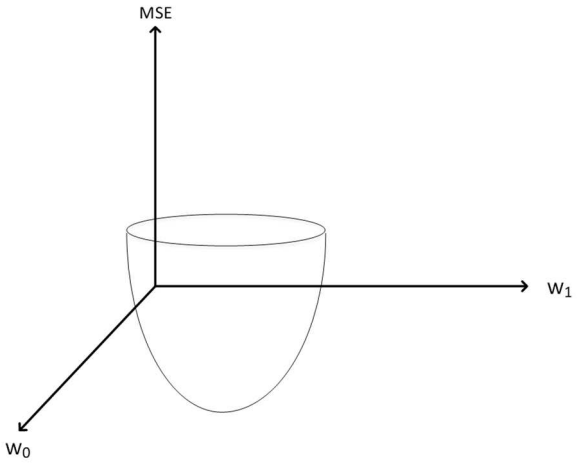
- We could choose something else, of course… the purpose is to minimize the deviation of the actual data from the line we find…

Learning and Mean Squared Error

- During the learning process, we try to find values for w_0 and w_1 (i.e. m and c) that produce less error.
- It means, in order to evaluate how good the weights are, every time we consider new values for weights, we check the error by comparing the prediction the line produces with the actual value the training data provides.
- We try to minimize the error, by optimizing weight values, i.e. finding weights that represent the data better.

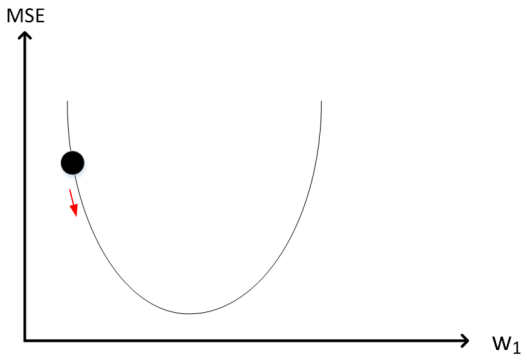
Visualizing Cost Function

- For different values of the two weights (which span the w_0w_1 plane) we have different error values (MSE). In a specific point the error will hopefully be minimum and that's the point we want to find (specific w_0 and w_1 values).



Steepest Descent

- If we ignore the second weight and just consider the relation between MSE and w_1 we will have a 2D graph like below. The error will be minimum at an extreme and at values higher and lower than that the error will be higher.
- If it wasn't going to converge then it wouldn't possibly be useful.



We should move in a direction that decreases the slope (derivative) and riches where derivative is 0.

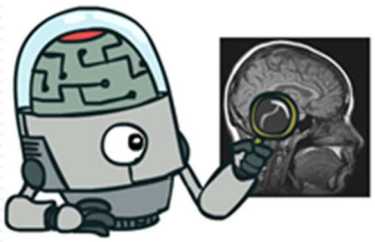
Since we have two variables (w_0, w_1) affecting E, we should use partial derivative.

$$\frac{\partial E}{\partial w_1}$$

Finding Weights and Usage

- There are different methods to find the weights.
  - Gradually descent (with an adjustable rate) toward the deepest point of error as described in previous page.
  - We can use optimization methods (like hill climbing etc.) or gradient decent
- After finding the weights we can use the trained neuron to predict values…

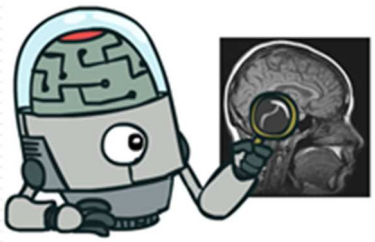
Setting the weights (Training the Network)



Setting the weights (Training the Network)

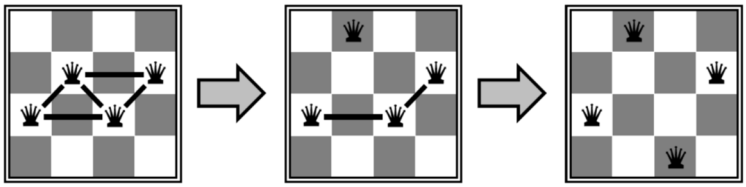
- The method of setting weights is an important characteristic of different nets.
  - **Supervised:** A sequence of training vectors (or patterns) and an associated target output vector (pattern or class) is presented to the network. The weights are adjusted using a learning algorithm.
    - **Pattern Association:** You put a pattern on input, you get a pattern at the output (sometimes called an associative memory)
    - **Pattern classification:** You put a pattern on input, you get a class number at the output
  - **Unsupervised:** the net modifies the weights so that the most similar input vectors are assigned to the same output (or cluster unit). These are called self organizing maps (such as Kohonen maps)

Training the Network: Genetic Algorithm



Iterative Improvement Algorithms (for solving CSPs)

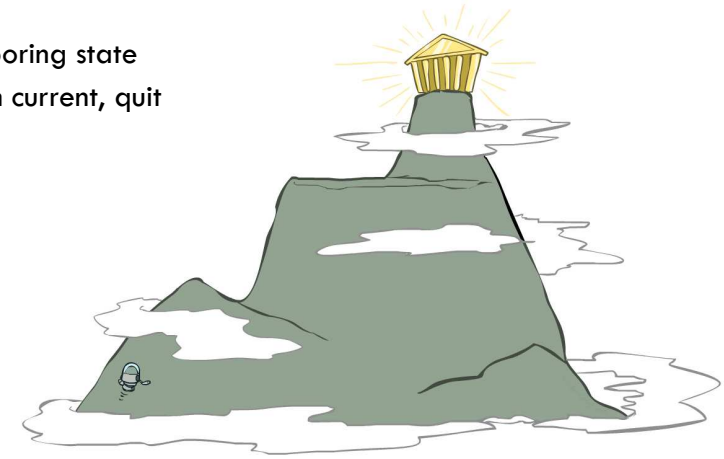
- **Local search methods:** typically work with "complete" states, i.e., all variables assigned
  - **Initial:** Start with fully assigned values to all variables
  - **Selection:** Take an assignment with unsatisfied constraints
  - **Improve:** *reassign* variable values



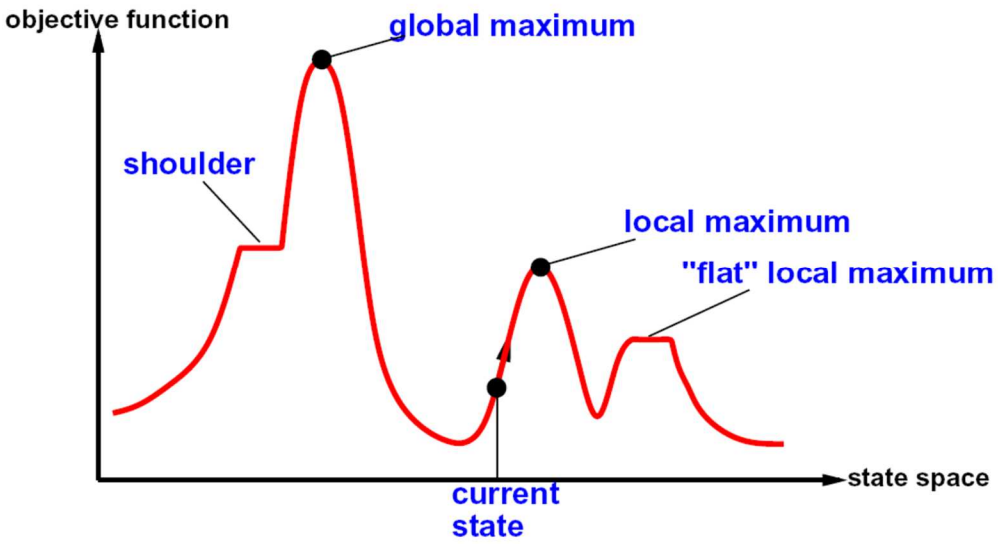
- **States:** 4 queens in 4 columns ($4^4 = 256$ states)
- **Operators:** move queen in column
- **Goal test:** no attacks
- **Evaluation:** c(n) = number of attacks

Hill Climbing

- Simple, general idea:
  - **Initial:** Start wherever
  - **Repeat:** move to the best neighboring state
  - **Stop:** If no neighbors better than current, quit

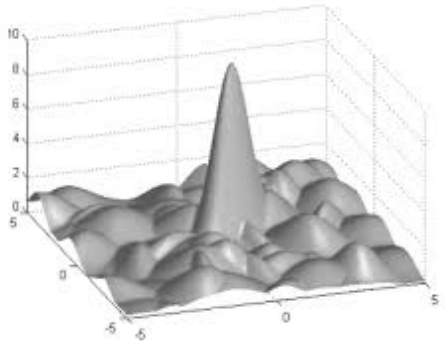


Hill Climbing Diagram



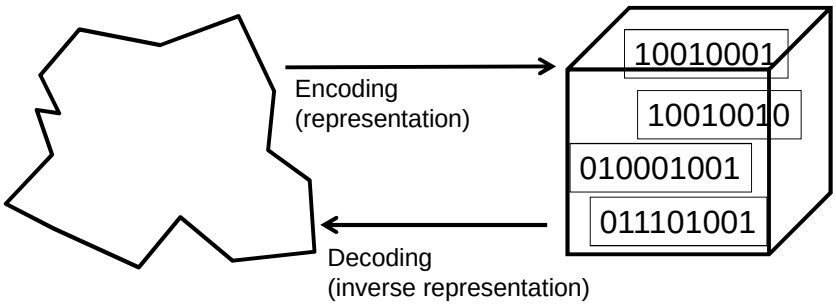
A typical genetic algorithm implementation

- Genetic Algorithm is an optimization and search algorithm.
- It uses a crossover and fittest selection method for local search (i.e. finding local optimal)
- It uses a mutation function for global search (i.e. finding global optimal)



Representation

Phenotype (problem) space Genotype (Genetic) space



Initial Random Population

- **Population**
  - A pool of population (chromosomes) is randomly created
  - For example, the size of population is considered 100 (dividable by 4)
- **Fitness Function**
  - A fitness function is defined which determines how fit each of the chromosomes are.

010111011	F=1.5
011101001	F=2.3
…	
111001101	F=1.4
001001001	F=3.3
110101001	F=2.2
011000001	F=0.9
001100001	F=1.8
111101000	F=2.15
011100001	F=3.1
000101001	F=0.5

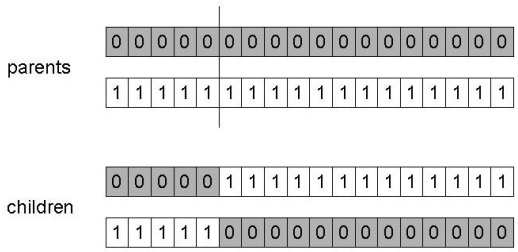
Sorting and Parent Selection

- **Sorting**
  - We sort the population by their fitness
- **Parent Selection**
  - We keep the top 50 individuals (more fit)
  - We throw away 50 worse half
  - Randomly select pairs

010111011	F=3.5
011101001	F=3.4
111001101	F=3.4
001001001	F=3.35
110101001	F=3.3
011000001	F=3.1
001100001	F=2.9
111101000	F=2.85
011100001	F=2.8
000101001	F=2.75
…	

Production, Step 1

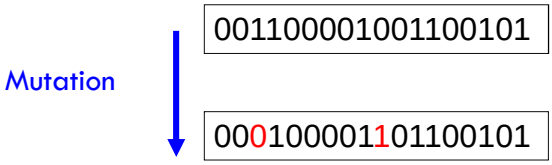
- **Crossover (recombination)**
  - We have 50 parent chromosomes, each pair will produce 2 Childs.
  - Choose a random point on the two parents
  - Split parents at this crossover point
  - Create children by exchanging tails
  - P_c typically in range (0.6, 0.9)



010111011
011101001
111001101
001001001
110101001
011000001
001100001
111101000
...

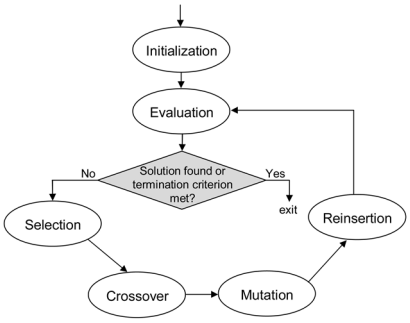
Production, Step 2

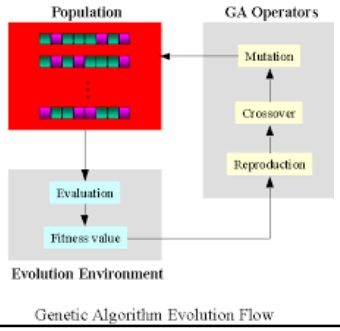
- **Mutation**
  - Alter each gene independently with a probability p_m
  - p_m is called the mutation rate, typically between 1/pop_size and 1/chromosome_length
  - The mutation rate should normally be very small (or it will ruin good results)
  - We may consider another rate which randomly determines which/how many of the children are mutated.



Loop and Conclusion

- **Loop and Stop Condition**
  - Now we sort the population again (which is now again 100)
  - We calculate average fitness.
  - We repeat reproduction.
  - We stop whenever the improvement in average fitness is less than a specific percentage
- **Conclusion**
  - Now we sort again and select the best individual (the most fit) and use it as the answer





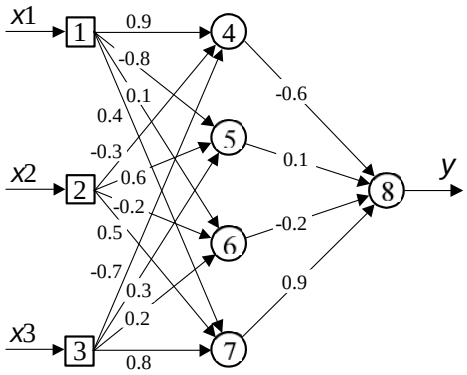
Genetic Algorithm Evolution Flow

Evolutionary neural networks

- One of the most common difficulties of using neural networks is their training process.
- The back-propagation learning algorithm (will be discussed later) cannot guarantee an optimal solution.
- In real-world applications, the back-propagation algorithm might converge to a set of sub-optimal weights from which it cannot escape. As a result, the neural network is often unable to find a desirable solution to a problem at hand.

Encoding a set of weights in a chromosome

From neuron	12		34		5678			
To neuron								
1	00		00		0000			
2	00		00		0000			
3	00		00		0000			
4	0.9	-0.3	-0.7	0	0000			
5	-0.8	0.6	0.3	0	0000			
6	0.1	-0.2	0.2	0	0000			
7	0.4	0.5	0.8	0	0000			
8	00		0	-0.6	0.1	-0.2	0.9	0



Chromosome	0.9	-0.3	-0.7	-0.8	0.6	0.3	0.1	-0.2	0.2	0.4	0.5	0.8	-0.6	0.1	-0.2	0.9

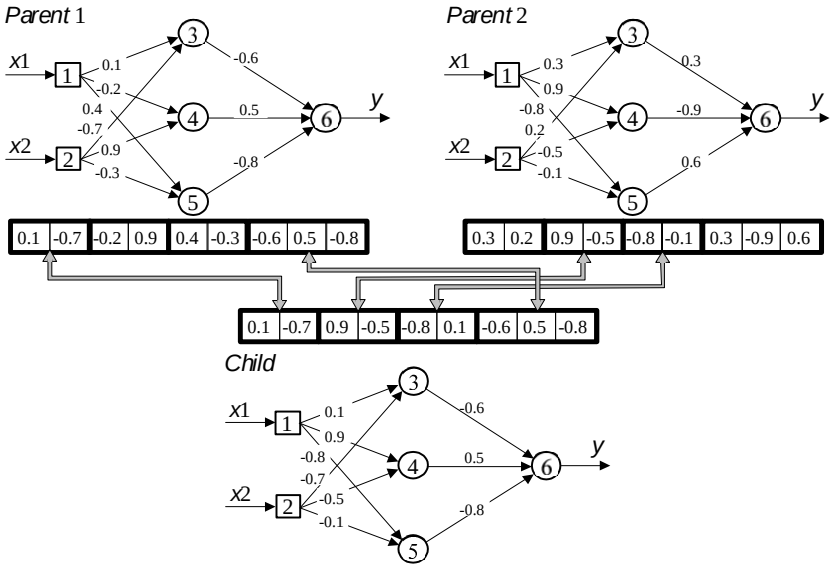
Fitness Function

- The second step is to define a fitness function for evaluating the chromosome's performance. This function must estimate the performance of a given neural network. We can apply here a simple function defined by the sum of squared errors.
- The training set of examples is presented to the network, and the sum of squared errors is calculated. The smaller the sum, the fitter the chromosome. **The genetic algorithm attempts to find a set of weights that minimizes the sum of squared errors.**

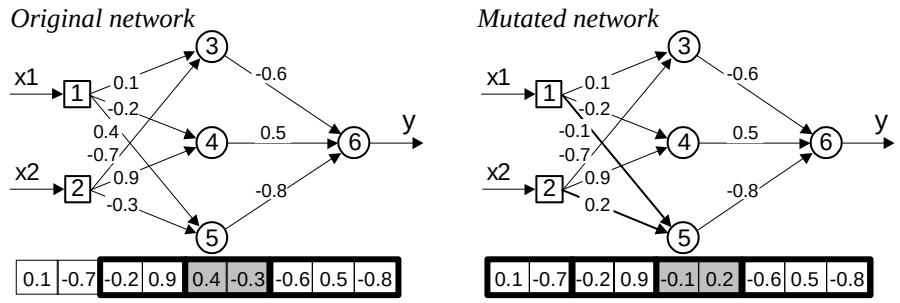
Genetic Operators

- The third step is to choose the genetic operators – crossover and mutation. A crossover operator takes two parent chromosomes and creates a single or two children with genetic material from both parents. Each gene in the child's chromosome is represented by the corresponding gene of the randomly selected parent.
- A mutation operator selects a gene in a chromosome and adds a small random value between –1 and 1 to each weight in this gene.

Crossover in weight optimization



Mutation in weight optimization



Optimum Weights

- As described earlier, optimum weights for the network should produce the least error (MSE for regression, Percentage for the classification) on the training data set.
- We might divide the Data into three parts for this method:
  - Training Data Set (e.g. 70%): Used with Genetic Algorithm for training
  - Validation Data Set (e.g. 15%): Used to see whether genetic algorithm optimization loop should stop or not. If the performance for the training set increases but the performance for the validation set decreases, it means we have started to over fit the model to the training set.
  - Test Data Set (e.g. 15%): Use to determine the performance on a separate unseen data set
- The use of validation data set is optional. The stop condition could be derived from the training data set itself (if the average error does not improve, then stop). But using validation data set is recommended.

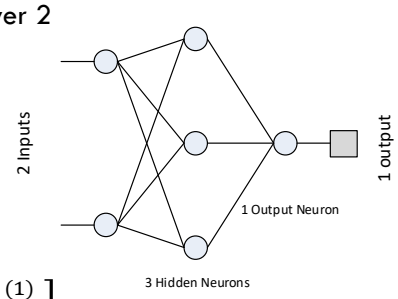
Optimum Weights

- After an optimum chromosome (which contains weights for all inputs in the network) is found, we use those weights on our network.
- We provide the test performance (percentage of the correct classifications on test data set) as the specification of the trained network.

Back Propagation

Output Calculation using Matrix Operations

- Network output can be calculated by means of matrix calculations.
- Assume X contains our example inputs and $W^{(1)}$ contains the first layer of weights (each row represents the input weights of each neuron in layer 2 or hidden layer).



$$X = \begin{bmatrix} 3 & 5 \\ 5 & 1 \\ 10 & 2 \end{bmatrix} \qquad W^{(1)} = \begin{bmatrix} w_{11}{}^{(1)} & w_{12}{}^{(1)} & w_{13}{}^{(1)} \\ w_{21}{}^{(1)} & w_{22}{}^{(1)} & w_{23}{}^{(1)} \end{bmatrix}$$

$$z^{(2)} = \begin{bmatrix} 3w_{11}{}^{(1)} + 5w_{21}{}^{(1)} & 3w_{12}{}^{(1)} + 5w_{22}{}^{(1)} & 3w_{13}{}^{(1)} + 5w_{23}{}^{(1)} \\ 5w_{11}{}^{(1)} + 1w_{21}{}^{(1)} & 5w_{12}{}^{(1)} + 1w_{22}{}^{(1)} & 5w_{13}{}^{(1)} + 1w_{23}{}^{(1)} \\ 10w_{11}{}^{(1)} + 2w_{21}{}^{(1)} & 10w_{12}{}^{(1)} + 2w_{22}{}^{(1)} & 10w_{13}{}^{(1)} + 2w_{23}{}^{(1)} \end{bmatrix}$$

$z^{(2)} = XW^{(1)}$ (1) activations of hidden layer

Output Calculation using Matrix Operations

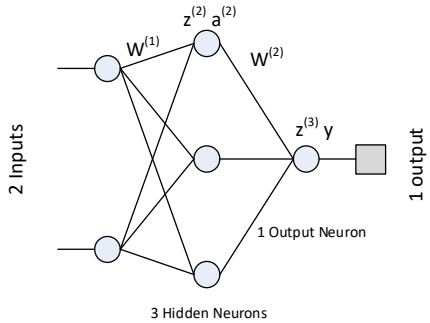
- Output is now calculated as:

$z^{(2)} = XW^{(1)}$ (1) is 3x3 (each line for 1 example)

$a^{(2)} = f(z^{(2)})$ (2) is 3x3 (each line for one example)

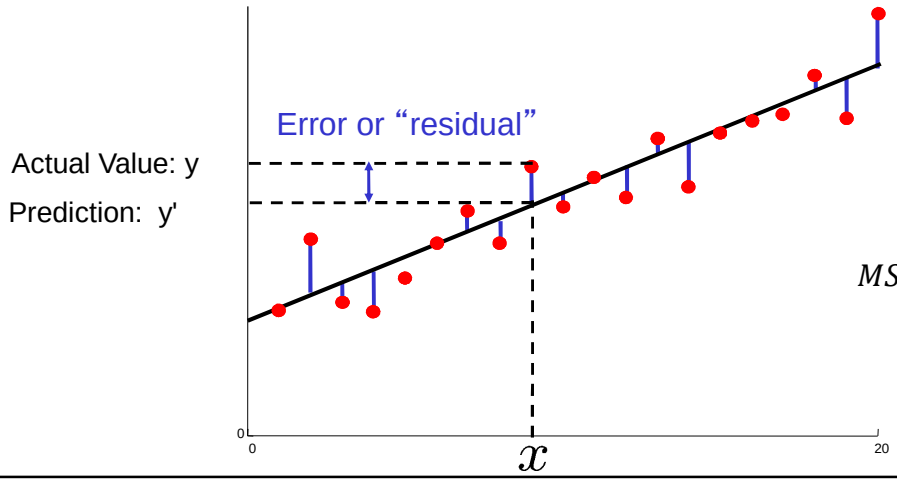
$z^{(3)} = a^{(2)}W^{(2)}$ (3) is "3x3 multiplied by 3x1"

$\hat{y} = f(z^{(3)})$ (4) is a 3x1, one output for each input example



Cost function - Mean Squared Error

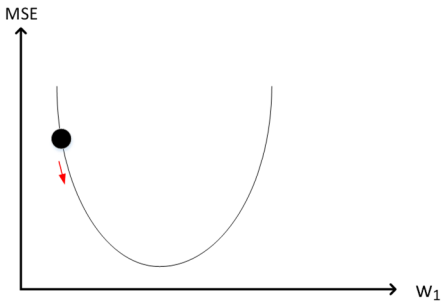
- Training the network for a regression problem, means modifying weights so that the cost function is minimized.



$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

Weight optimization

- If we had just one weight to adjust, We would change w_1 so that the cost function (MSE) becomes minimum.
- If we want to check 1000 values for w1 and see which one produces smaller MSE, it would make sense for one weight. But if we have say 10 weights to optimize, we will need 1000^5 combinations of weights to check.
- It is better to check the derivative to know, how and how much we should change the weight



We should move in a direction that decreases the slope (derivative) and riches where derivative is 0.

If we have more than one weight (w_0, w_1, ...) affecting MSE, we should use partial derivative.

$$\frac{\partial E}{\partial w_1}$$

Weight optimization

- We have these equations

$$z^{(2)} = XW^{(1)} \quad (1)$$

$$a^{(2)} = f(z^{(2)}) \quad (2)$$

$$z^{(3)} = a^{(2)}W^{(2)} \quad (3)$$

$$\hat{y} = f(z^{(3)}) \quad (4)$$

$$J = \sum_{i=1}^{n} \frac{1}{2}(y_i - \hat{y}_i)^2 \quad (5)$$

- Now we want to minimize cost function or J. We can combine above equations and have the following. Note that all examples are used at the same time to calculate cost function.

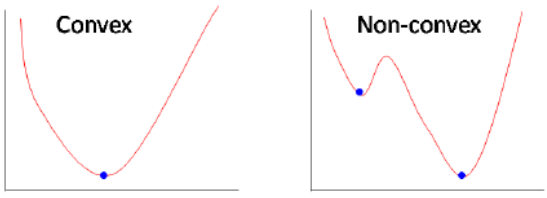
$$J = \sum \frac{1}{2}(y - f(f(XW^{(1)})W^{(2)}))^2$$

Gradient Descent

- Now we use the derivatives of the cost function to know whether we should increase or decrease each of the weights. We initially start by big jumps, if we pass the extreme point or reach it, then we rectify our jumps.

$$\frac{\partial J}{\partial w_i}$$

- How if the function J we want to optimize is not convex? The cost function we chose is convex! So no worries… also even if the function was non-convex, if we used our examples one by one (with a stochastic method like genetic algorithm), it would not matter whether the cost function is convex. We could still find a good optimal point.



Gradient Descent

- In order to perform gradient decent, we need to do modify to series of weights:

$$W^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix}$$

$$W^{(2)} = \begin{bmatrix} w_{11}^{(2)} \\ w_{12}^{(2)} \\ w_{13}^{(2)} \end{bmatrix}$$

- So we need to calculate the cost function changes in relation to changes of each of the two series.

$$\frac{\partial J}{\partial W^{(2)}} \quad \frac{\partial J}{\partial W^{(1)}}$$

Gradient Descent

$$\frac{\partial J}{\partial W^{(1)}} = \begin{bmatrix} \frac{\partial J}{\partial w_{11}^{(1)}} & \frac{\partial J}{\partial w_{12}^{(1)}} & \frac{\partial J}{\partial w_{13}^{(1)}} \\ \frac{\partial J}{\partial w_{21}^{(1)}} & \frac{\partial J}{\partial w_{22}^{(1)}} & \frac{\partial J}{\partial w_{23}^{(1)}} \end{bmatrix}$$

$$\frac{\partial J}{\partial W^{(2)}} = \begin{bmatrix} \frac{\partial J}{\partial w_{11}^{(2)}} \\ \frac{\partial J}{\partial w_{12}^{(2)}} \\ \frac{\partial J}{\partial w_{13}^{(2)}} \end{bmatrix}$$

Gradient Descent

- In this equation summation calculates the error over all examples. According "sum of derivatives" rule, we can bring out the summation and for the time being care only about a single example.

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial \sum_{i=1}^{n} \frac{1}{2}(y_i - \hat{y}_i)^2}{\partial W^{(2)}} = \sum \frac{\partial \frac{1}{2}(y_i - \hat{y}_i)^2}{\partial W^{(2)}}$$

- To keep things simple, we ignore about the summation for now (and consider a single example) and later consider it.

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial \frac{1}{2}(y - \hat{y})^2}{\partial W^{(2)}}$$

Gradient Descent

- Chain rule:

$$(f \circ g)' = (f' \circ g) \cdot g'$$

e.g.: $\frac{d}{dx}(3x + 2x^2)^2 = 2(3x + 2x^2)(3 + 6x)$

- Other form (in fact is the same as previous one):

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \qquad :: \qquad \frac{d\text{fog}}{dx} = \frac{d\text{fog}}{dg} \cdot \frac{dg}{dx}$$

- y is constant (won't change) in relation to W but $\hat{y}$ does:

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial \frac{1}{2}(y - \hat{y})^2}{\partial W^{(2)}} = -(y - \hat{y}) \frac{\partial \hat{y}}{\partial W^{(2)}}$$

Gradient Descent

- So:

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial \frac{1}{2}(y-\hat{y})^2}{\partial W^{(2)}} = -(y-\hat{y})\frac{\partial \hat{y}}{\partial W^{(2)}}$$

- In order to calculate , from (4) we have $\hat{y} = \mathrm{f}(z^{(3)})$ so

$$\frac{\partial J}{\partial W^{(2)}} = -(y-\hat{y})\frac{\partial \hat{y}}{\partial z^{(3)}}.\frac{\partial z^{(3)}}{\partial W^{(2)}}$$

- We know:

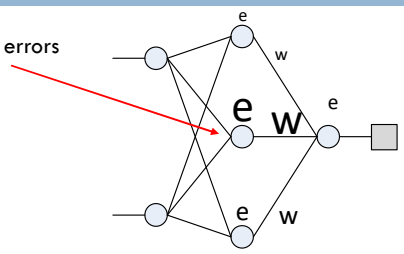
$$f(z) = \frac{1}{1+e^{-z}} \quad \text{and} \quad f'(z) = \frac{e^{-z}}{(1+e^{-z})^z}$$

Gradient Descent

- So we replace $\frac{\partial \hat{y}}{\partial z^{(3)}}$ with $f'(z^{(3)})$:

$$\frac{\partial J}{\partial W^{(2)}} = -(y-\hat{y})\frac{\partial \hat{y}}{\partial z^{(3)}}.\frac{\partial z^{(3)}}{\partial W^{(2)}} = -(y-\hat{y})f'(z^{(3)})\frac{\partial z^{(3)}}{\partial W^{(2)}}$$

More responsible for errors



- We now need to do something about $\frac{\partial z^{(3)}}{\partial W^{(2)}}$ considering that $z^{(3)} = a^{(2)}W^{(2)}$ (3)
- The relation between z(3) and W(2) is linear (e.q. 3) and the derivative $\frac{\partial z^{(3)}}{\partial W^{(2)}} = a^{(2)}$
- The neuron at previous layer which has higher weight is more responsible for the error at the output…

Gradient Descent

- We saw:

$$\frac{\partial J}{\partial W^{(2)}} = -(y-\hat{y}) f'\left(z^{(3)}\right) \frac{\partial z^{(3)}}{\partial W^{(2)}}$$

- First term is a matrix of 3x1, second term is 3x1. These two are multiplied as "Scalar Multiplication"

$$\begin{bmatrix} -y_1 - \widehat{y_1} \\ -y_2 - \widehat{y_2} \\ -y_3 - \widehat{y_3} \end{bmatrix} \begin{bmatrix} f'(z_1^{(3)}) \\ f'(z_2^{(3)}) \\ f'(z_3^{(3)}) \end{bmatrix} = \begin{bmatrix} \delta_1^{(3)} \\ \delta_2^{(3)} \\ \delta_3^{(3)} \end{bmatrix}$$

$\delta^{(3)}$ is called Back propagating error

- $\frac{\partial z^{(3)}}{\partial W^{(2)}}$ is the activity of 2nd layer neurons (arriving at 3rd layer):

$$\frac{\partial z^{(3)}}{\partial W^{(2)}} = \begin{bmatrix} a_{11}{}^{(2)} & a_{12}{}^{(2)} & a_{13}{}^{(2)} \\ a_{21}{}^{(2)} & a_{22}{}^{(2)} & a_{23}{}^{(2)} \\ a_{31}{}^{(2)} & a_{32}{}^{(2)} & a_{33}{}^{(2)} \end{bmatrix}$$

Gradient Descent

- We saw:

$$\delta^{(3)} = -(y-\hat{y}) f'\left(z^{(3)}\right) \qquad (6)$$

$$\frac{\partial J}{\partial W^{(2)}} = (a^{(2)})^T \delta^{(3)}$$

- Now we have everything to optimize weights of $W^{(2)}$ in a way that J is minimized.

- In next step we should use chain rule to calculate $\frac{\partial J}{\partial W^{(1)}}$ so that we can optimize the values of first weights layer

Gradient Descent

- To find optimal weights for $W^{(1)}$, we act almost the same:

$$\frac{\partial J}{\partial W^{(1)}} = \frac{\partial \frac{1}{2}(y-\hat{y})^2}{\partial W^{(1)}} = -(y-\hat{y})\frac{\partial \hat{y}}{\partial W^{(1)}} = -(y-\hat{y})\frac{\partial \hat{y}}{\partial z^{(3)}}.\frac{\partial z^{(3)}}{\partial W^{(1)}} = -(y-\hat{y})f'(z^{(3)})\frac{\partial z^{(3)}}{\partial W^{(1)}}$$

$$= \delta^{(3)}\frac{\partial z^{(3)}}{\partial W^{(1)}} = \delta^{(3)}\frac{\partial z^{(3)}}{\partial a^{(2)}}\frac{\partial z^{(2)}}{\partial W^{(1)}}$$

- Still there is a linear relation between first layer weights $W^{(2)}$, and activations of first layer $a^{(2)}$

$$= \delta^{(3)}\frac{\partial z^{(3)}}{\partial a^{(2)}}\frac{\partial z^{(2)}}{\partial W^{(1)}} = \delta^{(3)}\,(W^{(2)})^T\frac{\partial a^{(2)}}{\partial z^{(2)}}\frac{\partial z^{(2)}}{\partial W^{(1)}} = \delta^{(3)}\,(W^{(2)})^T f'\left(z^{(2)}\right)\frac{\partial z^{(2)}}{\partial W^{(1)}}$$

- And there is a linear relation between $z^{(2)}$ and inputs, i.e. $z^{(2)} = W^{(1)}.X$ so $\frac{\partial z^{(2)}}{\partial W^{(1)}} = X$

$$= X^T\delta^{(3)}\,(W^{(2)})^T f'\left(z^{(2)}\right)$$

Gradient Descent

- So:

$$\frac{\partial J}{\partial W^{(1)}} = X^T\delta^{(3)}\,(W^{(2)})^T f'\left(z^{(2)}\right)$$

Where

$$\delta^{(3)} = -(y-\hat{y})f'\left(z^{(3)}\right)$$

And

$$f'(z) = \frac{e^{-z}}{(1+e^{-z})^z}$$

- We may also consider $\delta^{(2)} = \delta^{(3)}\,(W^{(2)})^T f'\left(z^{(2)}\right)$ which is called error back propagated to layer 1
- Now we can optimize the $W^{(1)}$ too…

Gradient Descent

- If we have more layers we basically use the chain rule like we did for 2 layers to calculate the $\frac{\partial J}{\partial W^{(?)}}$ of that layer in order to optimize its weights.

- Like we mentioned earlier, in order to find suitable values for weights we start from a random weight and reduce or increase it in relatively large steps (depending on the derivative value).