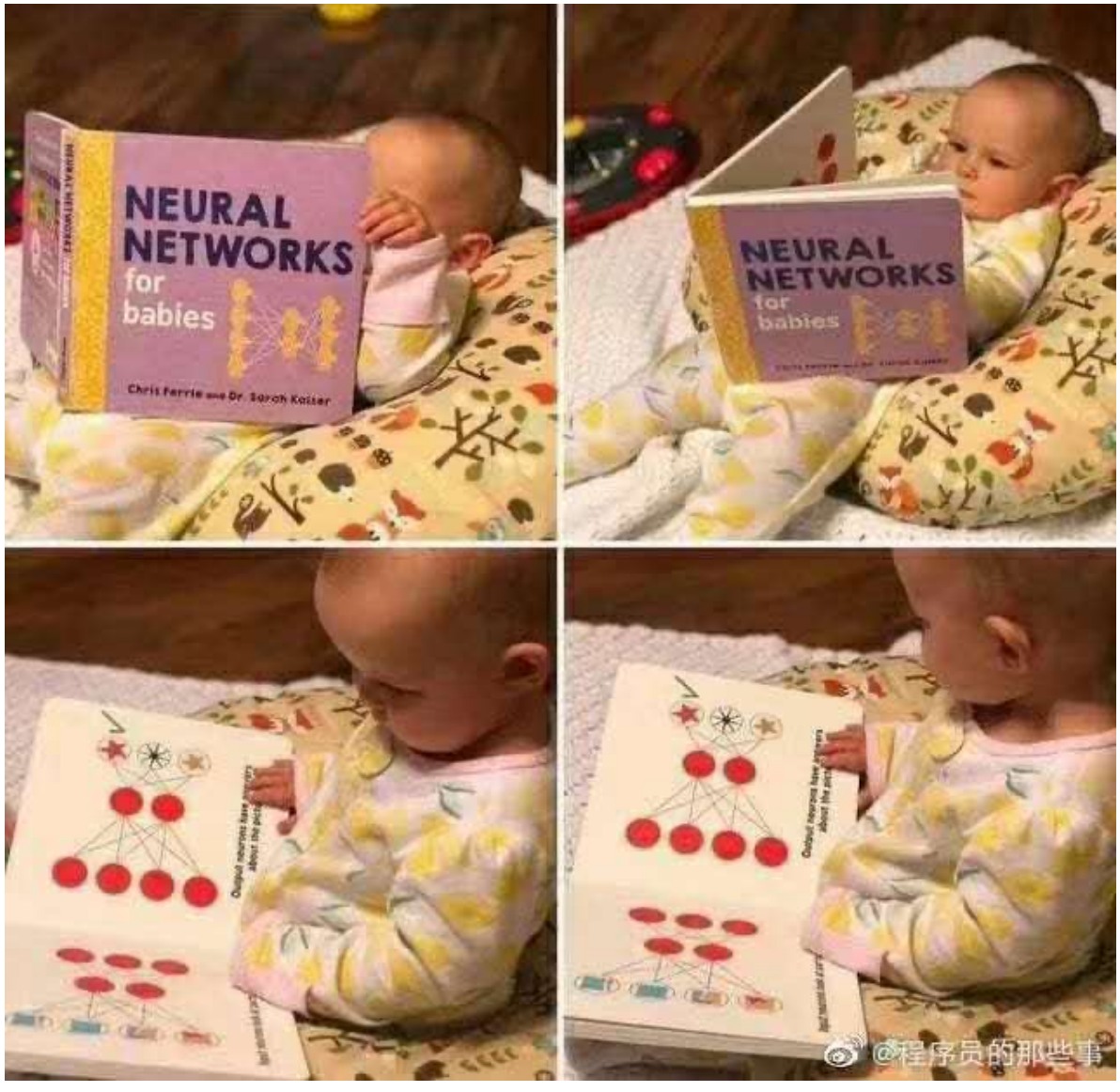
# STA6933: Advanced Topics in Statistical Learning

## Chapter 7: Neural Networks

Min Wang

Department of Management Science and Statistics
The University of Texas at San Antonio

min.wang3@utsa.edu


NEURAL NETWORKS
for babies
Chris Ferrie and Dr. Sarah Kaiser
NEURAL NETWORKS
for babies
@程序员的那些事

# Neural networks

- Lots of buzz words, but what do they mean?
- Definitions:
  - AI (Artificial Intelligence): human-like machines or programs
  - ML (Machine Learning): Algorithms that learn from data
  - DL (Deep Learning): A type of ML algorithm, using neural networks (typically with many layers)
- Some potential answers
  - A universal function approximator
  - A feature extractor
  - A model generalizer

# Types of AI apps explained

## Machine learning

AI that uses current and historical data, along with algorithms, to make software more accurate at predicting outcomes without being programmed to do so.

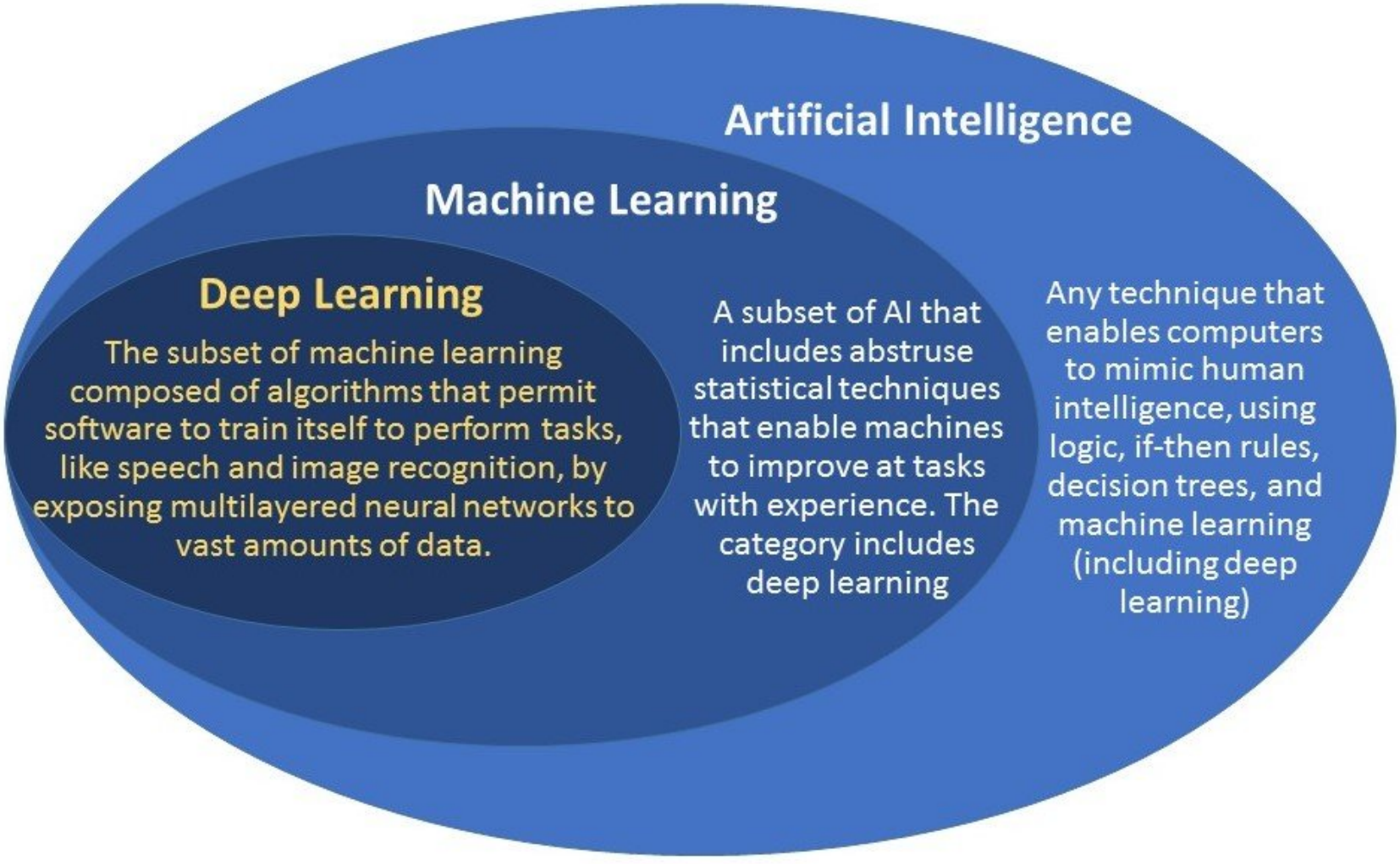
## Deep learning

A type of machine learning that imitates the way humans gain knowledge, making the process of collecting, analyzing and interpreting large amounts of data faster and easier.

## Neural network

A system of hardware and software designed to recognize patterns and operate similar to neurons in the human brain.

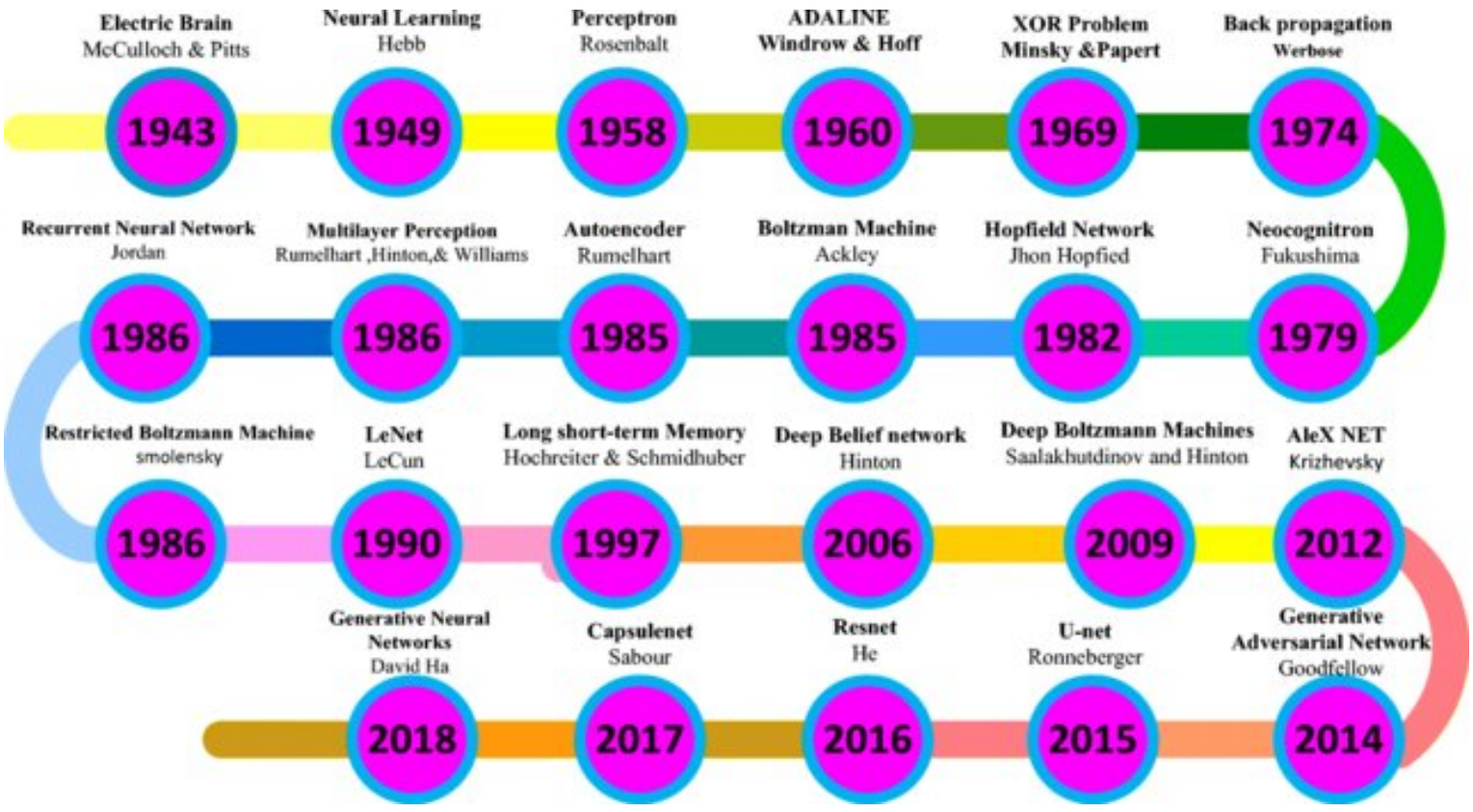
ICONS: IRENE/ADOBE STOCK, ELENABSL/ADOBE STOCK


Artificial Intelligence
Any technique that enables computers to mimic human intelligence, using logic, if-then rules, decision trees, and machine learning (including deep learning)
Machine Learning
A subset of AI that includes abstruse statistical techniques that enable machines to improve at tasks with experience. The category includes deep learning
Deep Learning
The subset of machine learning composed of algorithms that permit software to train itself to perform tasks, like speech and image recognition, by exposing multilayered neural networks to vast amounts of data.

# Neural networks

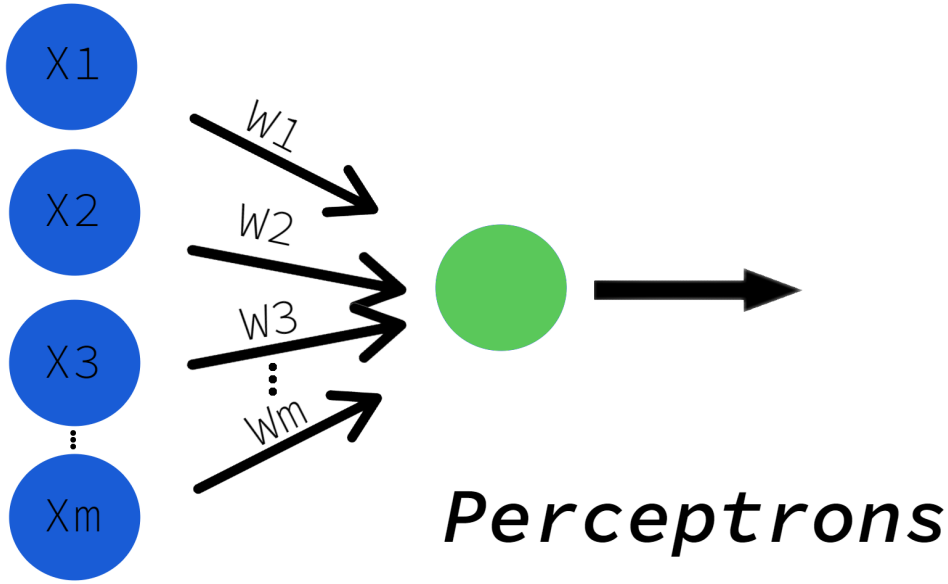
- Consider humans:
  - Neuron switching time $\sim 0.001$ second
  - Number of neurons $\sim 10^{10}$
  - Connections per neuron $\sim 10^{4-5}$
  - Scene recognition time $\sim 0.1$ second
  - 100 inference steps doesn't seem like enough
  - Massively parallel computation
- Neural networks properties:
  - Many neuron-like threshold switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed process
  - Emphasis on tuning weights automatically

# Brief history of neural networks

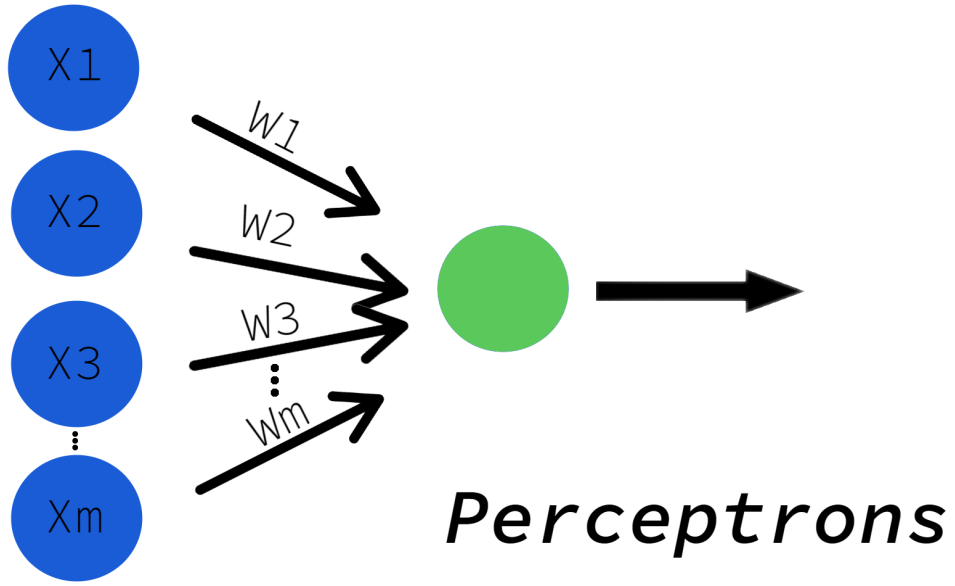


# Perceptrons

- One type of artificial neuron is called a perceptron
- Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts
- A perceptron takes several binary inputs, $x_1, x_2, \cdots$, and produces a single binary output



# Perceptrons




- Here $m$ inputs are given (could be more or less)
- Rosenblatt proposed a simple rule to compute the output, with weights, $w_1, w_2, \cdots w_m$, which are real numbers expressing the importance of the respective inputs to the output
- The neuron's output, 0 or 1, is determined by whether the weighted $\sum_{j=1}^{m} w_j x_j$ is less than or greater than some threshold value

# A simple decision

- Just like the weights, the threshold is a real number which is a parameter of the neuron
- In an algebraic terms

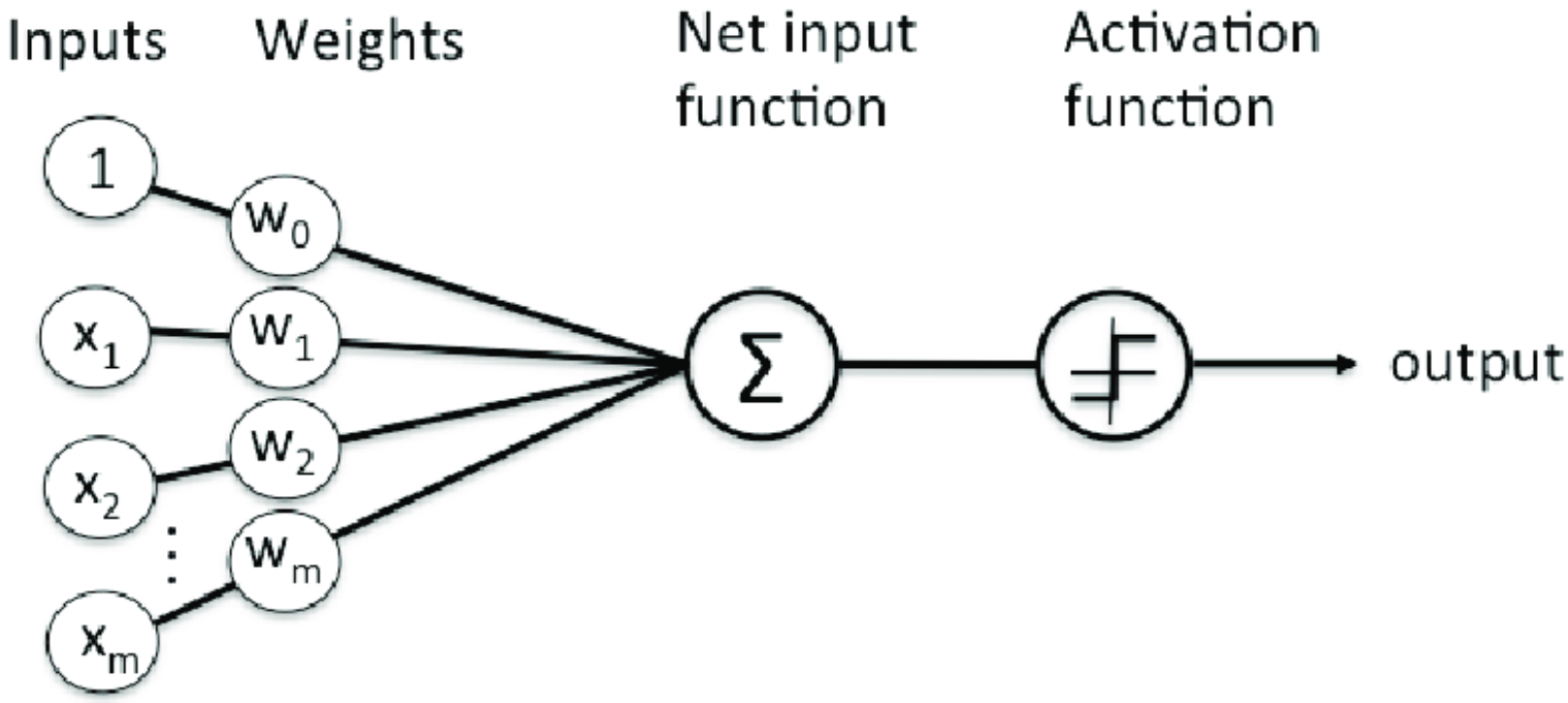
$$\text{output} = \begin{cases} 0, & \text{If } \sum_j w_j x_j \leq \text{ threshold}, \\ 1, & \text{If } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

- When we use inner-product notation and move the threshold to the left side, we have

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

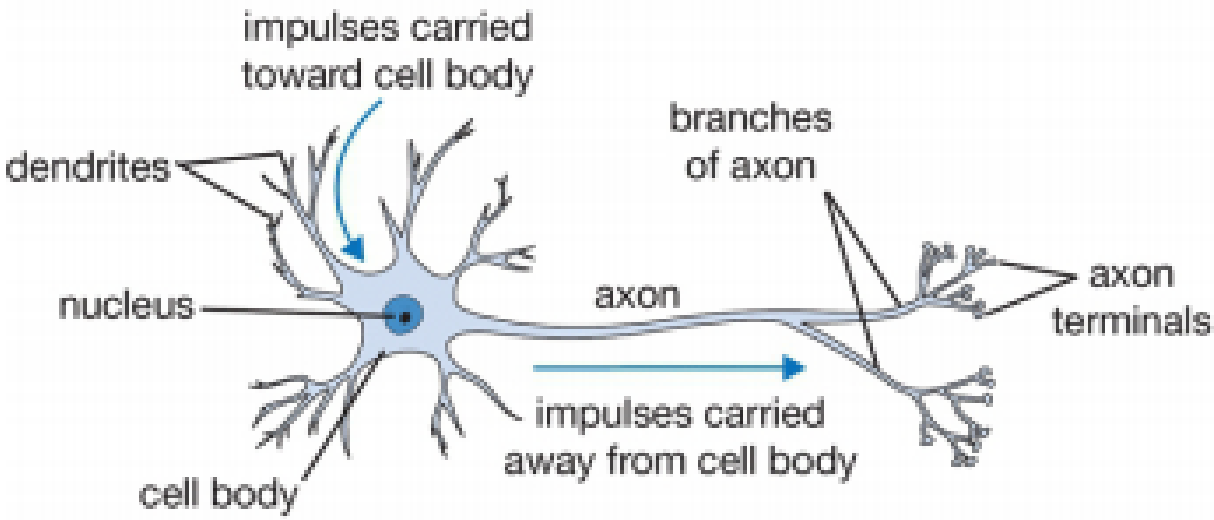
# Organization of a perceptron

Frank Rosenblatt's Perceptron (1958)

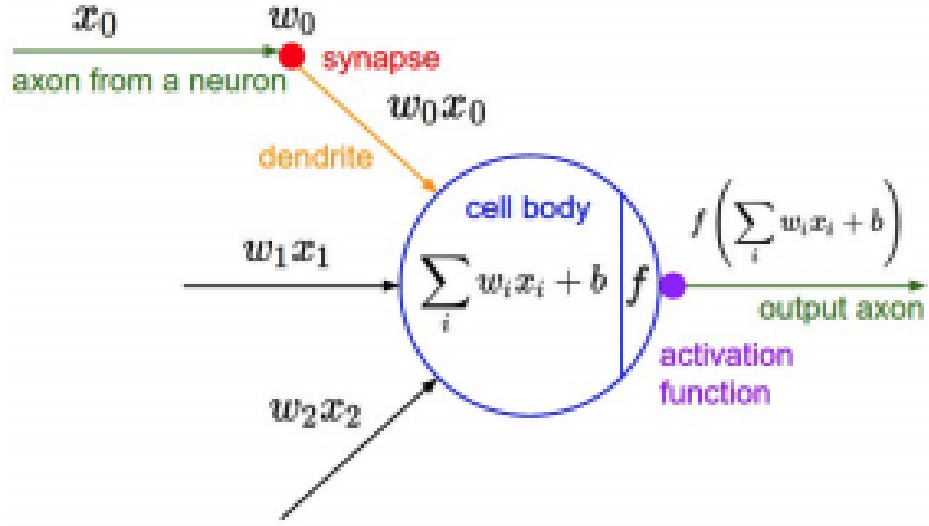


# Neuron and basic computational unit of the brain

- Many machine learning methods inspired by biology, e.g., the (human) brain



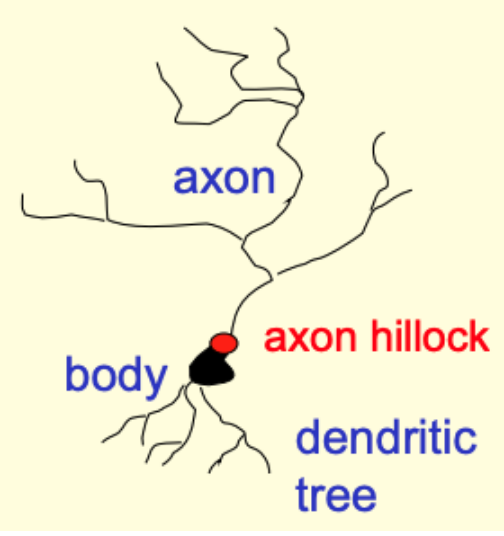

- A mathematical model of the neuron in a neural network



# Neural networks

- To understand how the brain actually works
  - It is very big and very complicated and made of stuff that dies when you poke it around
  - We need to use computer simulations
- To understand a style of parallel computation inspired by neurons and their adaptive connections
  - Very different style from sequential computation
- To solve practical problems by using novel learning algorithms inspired by the brain
  - Learning algorithms can be very useful even if they are not how the brain actually works

# Neural networks



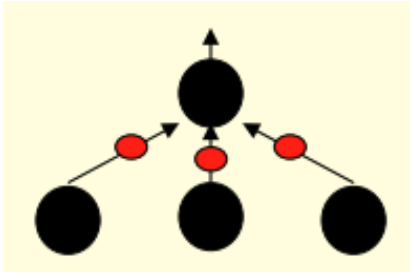

- Gross physical structure:
  - There is one axon that branches
  - There is a dendritic tree that collects input from other neurons
- Axons typically contact dendritic trees at synapses
  - A spike of activity in the axon causes charge to be injected into the post-synaptic neuron
- Spike generation:
  - There is an axon hillock that generates outgoing spikes whenever enough charge has flowed in at synapses to depolarize the cell membrane

# Synapses and how synapses adapt

- When a spike of activity travels along an axon and arrives at a synapse it causes vesicles of transmitter chemical to be released
  - There are several kinds of transmitter
- The transmitter molecules diffuse across the synaptic cleft and bind to receptor molecules in the membrane of the post-synaptic neuron thus changing their shape
  - This opens up holes that allow specific ions in or out
- The effectiveness of the synapse can be changed:
  - vary the number of vesicles of transmitter
  - vary the number of receptor molecules
- Synapses are slow, but they have advantages over a random access memory (RAM)
  - They are very small and very low-power
  - They adapt using locally available signals
  - But what rules do they use to decide how to change?

# How the brain works

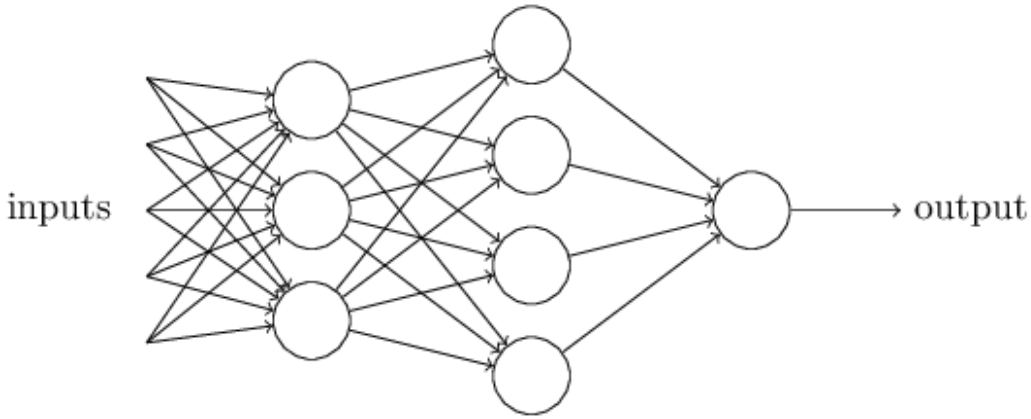
- Each neuron receives inputs from other neurons
  - A few neurons also connect to receptors
  - Cortical neurons use spikes to communicate



- The effect of each input line on the neuron is controlled by a synaptic weight, which can be positive or negative
- The synaptic weights adapt so that the whole network learns to perform useful computations for 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 can affect the computation in a very short time

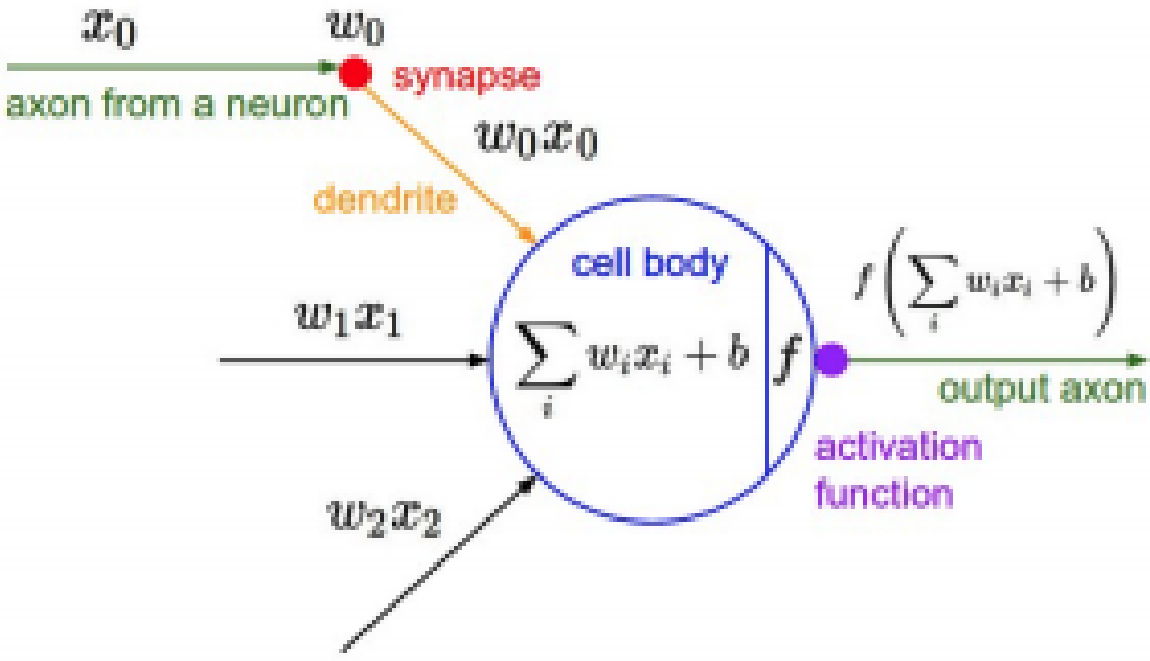
# A network of perceptrons

- Obviously, the perceptron isn't a complete model of human decision-making!
- One perception illustrates how a perceptron can weigh up different kinds of evidence in order to make decisions
- A complex network of perceptrons are the ones that make decisions




- To the example we discussed, the first layer of perceptrons makes three simple decisions, by weighing the input evidence
- The second layer of perceptrons, if it exists, makes decisions by weighing up the results from the first layer of decision-making

# Back to the math model of a neuron

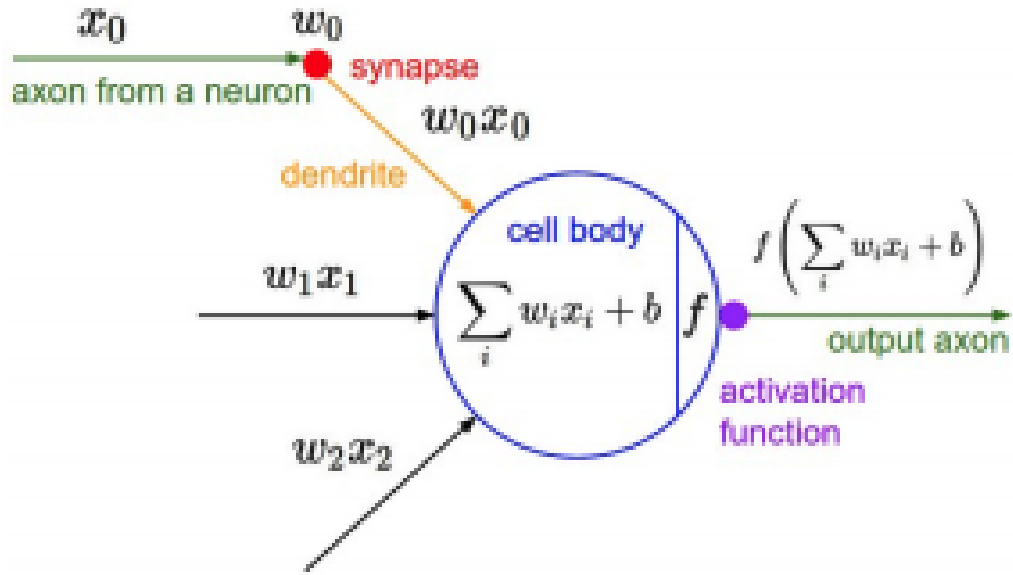



- The activation function in the diagram is called the non-linear function applied to the weighted input sum to produce the output of the artificial neuron
  - In the case of Rosenblatt's Perceptron, the function just a thresholding operation
- An important shortcoming of a perceptron is that a small change in the input values can cause a large change the output because each node (or neuron) only has two possible states: 0 or 1

# Sigmoid neuron

- For one of such activation function, we could simply have the neuron emit the value:

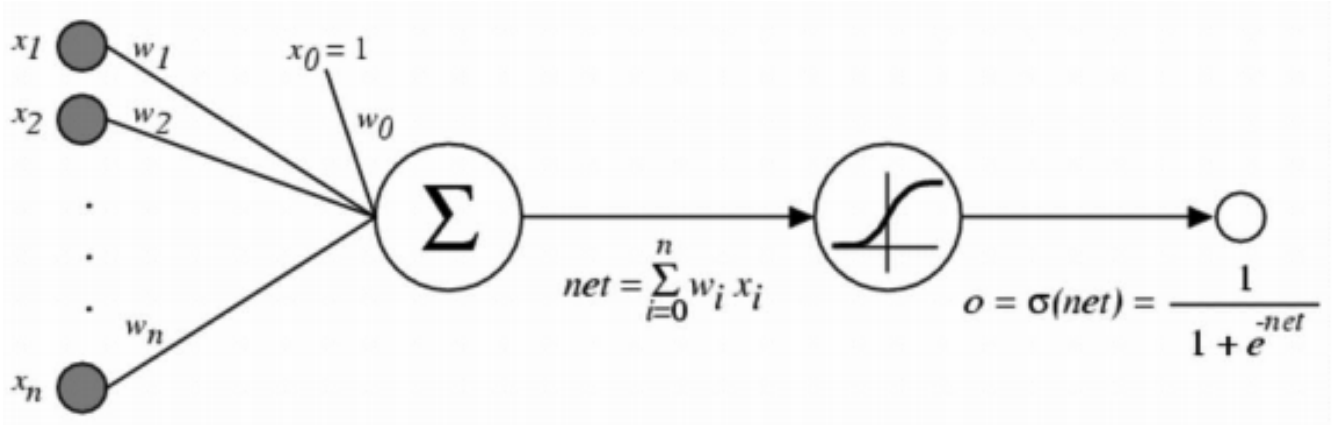
$$\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) = \frac{1}{1 + e^{-(\boldsymbol{w} \cdot \boldsymbol{x} + b)}}$$




- For a particularly positive or negative value of $\boldsymbol{w} \cdot \boldsymbol{x} + b$, the result will be nearly the same as with the perceptron (i.e., near 0 or 1)
- For values close to the boundary of the separating hyperplane, values near 0.5 will be emitted

# Sigmoid neuron (continued)

- With the Sigmoid neuron, we could have




- This activation function perfectly mimics logistic regression, and in fact uses the logit function to do so
- In the neural network literature, the logit function is called the sigmoid function, thus leading to the name sigmoid neuron for a neuron that uses it's logic
- Notice that the previous restriction to binary inputs was not at all needed, and can be easily replaces with continuous input without an changes needed to the formulas

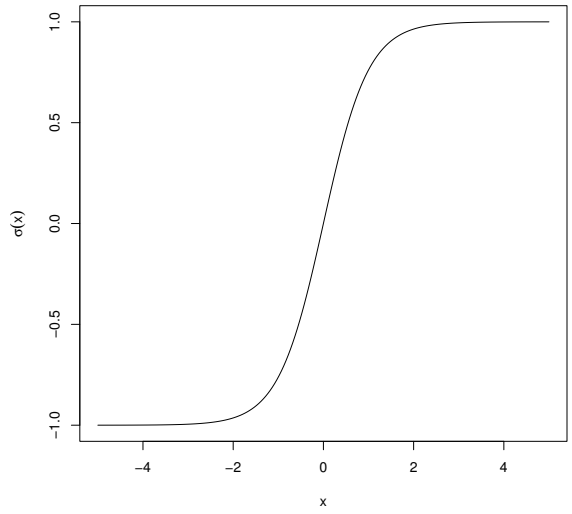
# Some common activation functions

- Using a logistic, or sigmoid, activation function has some benefits in being able to easily take derivatives and the interpret them using logistic regression
  - $\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) = \frac{1}{1+e^{-(\boldsymbol{w} \cdot \boldsymbol{x} + b)}}$
- Other choices have certain benefits that have recently grown in popularity. Some of these include:
  - hyperbolic tan: $\tanh(z) = 2\sigma(2z) - 1$
  - rectified linear unit: $ReLU(z) = \max\{0, z\}$
  - leaky rectified linear unit: $\max\{0.1 * z, z\}$
  - maxout: $\sigma(\boldsymbol{x}) = \max\{\boldsymbol{w}_1 \cdot \boldsymbol{x} + b_1, \boldsymbol{w}_2 \cdot \boldsymbol{x} + b_2, \ldots, \boldsymbol{w}_k \cdot \boldsymbol{x} + b_k\}$

# Sigmoid activation function

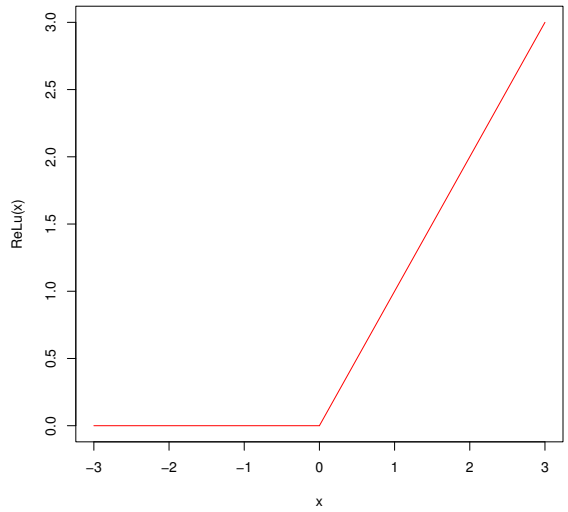
- $\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) = \frac{1}{1+e^{-(\boldsymbol{w} \cdot \boldsymbol{x} + b)}}$
- Pros:
  - Smooth gradient, preventing jumps in output values
  - Output values bound between 0 and 1, normalizing the output of each neuron
  - Clear predictions — For $X$ above 2 or below $-2$, tends to bring the $Y$ value (the prediction) to the edge of the curve, very close to 1 or 0. This enables clear predictions
- Cons:
  - Vanishing gradient — for very high or very low values of $X$, there is almost no change to the prediction, causing a vanishing gradient problem
  - This can result in the network refusing to learn further, or being too slow to reach an accurate prediction
  - Outputs not zero centered
  - Computationally expensive

# TanH / Hyperbolic Tangent activation function



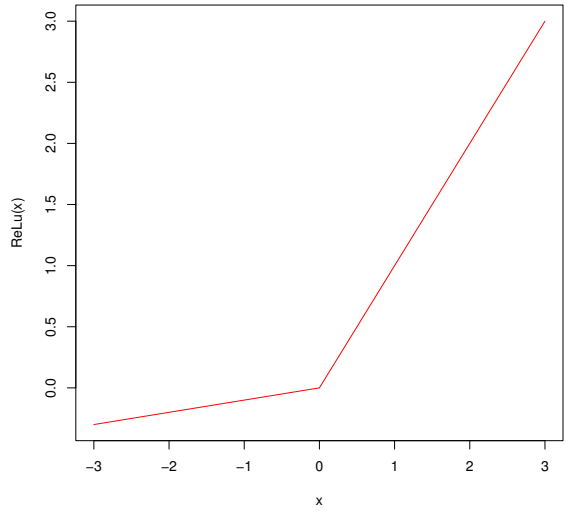

- $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- Pros:
  - Zero centered — making it easier to model inputs that have strongly negative, neutral, and strongly positive values
  - Otherwise like the Sigmoid function
- Cons:
  - Like the Sigmoid function

# ReLU (Rectified Linear Unit) activation function




- $ReLU(z) = \max\{0, z\}$
- Pros:
  - Computationally efficient — allows the network to converge very quickly
  - Non-linear — although it looks like a linear function, ReLU has a derivative function and allows for backpropagation
- Cons:
  - The Dying ReLU problem — when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn

# Leaky ReLU activation function




- Pros:
  - Prevents dying ReLU problem — this variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values
  - Otherwise like ReLU
- Cons:
  - Results not consistent— leaky ReLU does not provide consistent predictions for negative input values

# Gradient descent: an analogy

A person is stuck in the mountains and is trying to get down. There is heavy fog such that visibility is extremely low

- The person can look at the steepness of the hill at their current position, then proceeding in the direction with the steepest descent (i.e., downhill)
- The measure of the steepness is differentiation (the slope of the error surface can be calculated by taking the derivative of the squared error function at that point)
- Using this method, the person would eventually find a way down the mountain or possibly get stuck in some hole (i.e., local minimum or saddle point), like a mountain lake

# Gradient descent: an analogy

A person is stuck in the mountains and is trying to get down. There is heavy fog such that visibility is extremely low

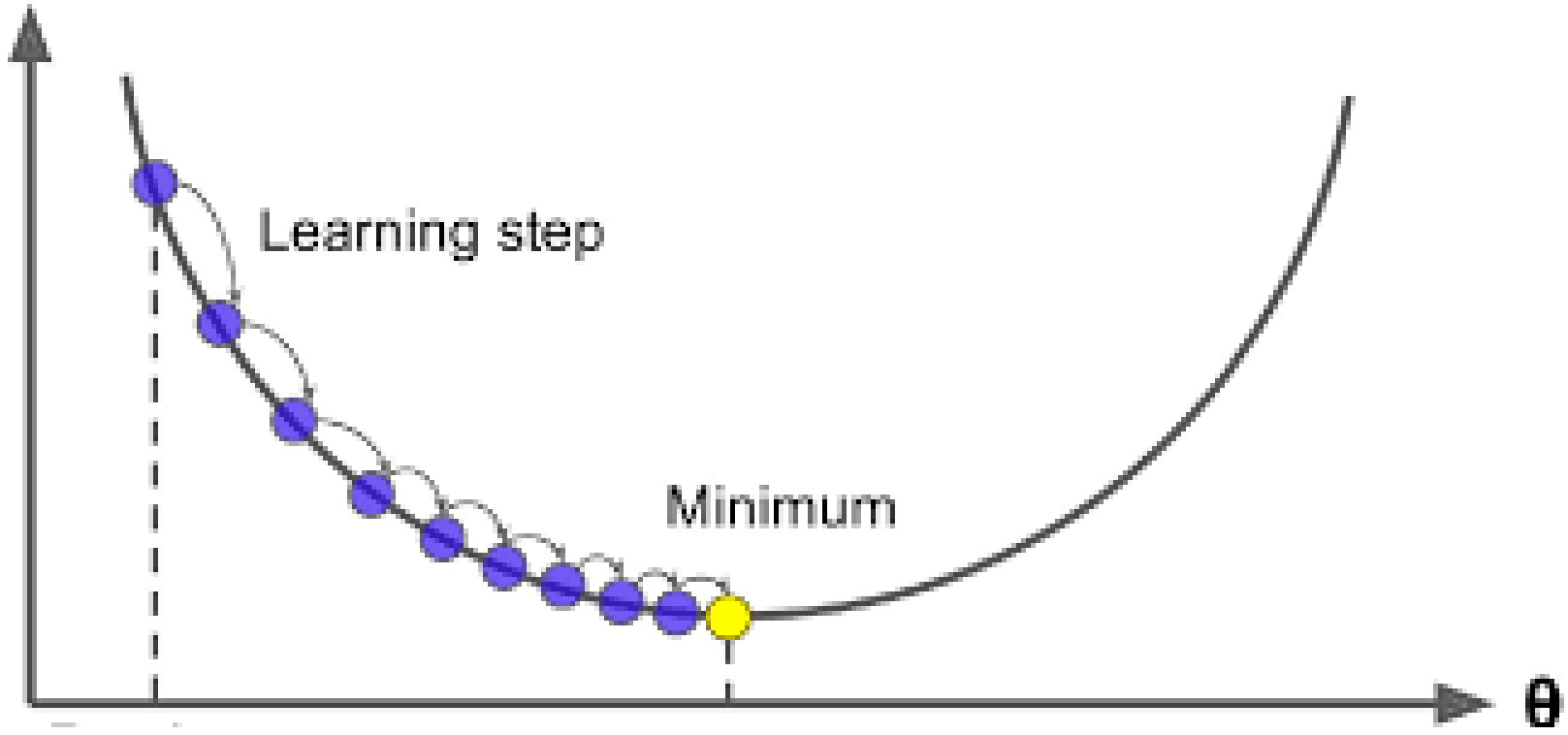
- If the steepness of the hill is not immediately obvious with simple observation, but rather it requires a sophisticated instrument to measure, which the person happens to have at the moment
- It takes quite some time to measure the steepness of the hill with the instrument
- The person should minimize the use of the instrument due to time constraint, say, before sunset
- The difficulty then is choosing the frequency at which they should measure the steepness of the hill so not to go off track (this could be determined by step size)

# Gradient descent: concept

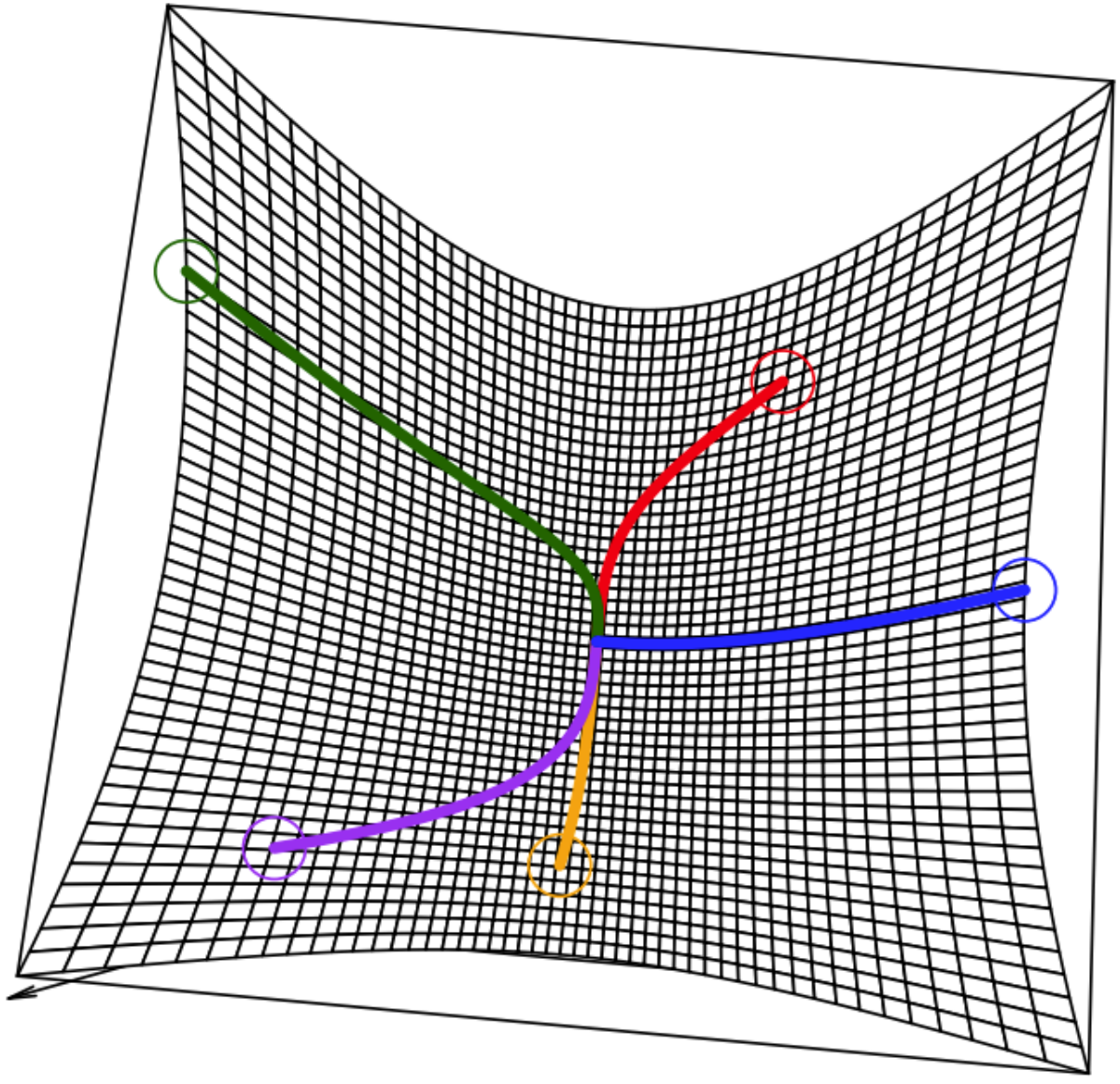
Suppose we have a function $f : \mathcal{R}^n \to \mathcal{R}$, which is convex and differentiable. We want to solve $\min_{\boldsymbol{x} \in \mathcal{R}^n} f(\boldsymbol{x})$

- Consider a strategy as follows:
  - Choose an initial value $\boldsymbol{x}^{(0)} \in R^n$
  - For each $k$, repeatedly update $\boldsymbol{x}^{(k)} = \boldsymbol{x}^{k-1} - \gamma_k \nabla f(\boldsymbol{x}^{(k-1)})$, for $k = 1, 2, \ldots$
  - Stop at some point

# Illustration of gradient descent



# Illustration of gradient descent

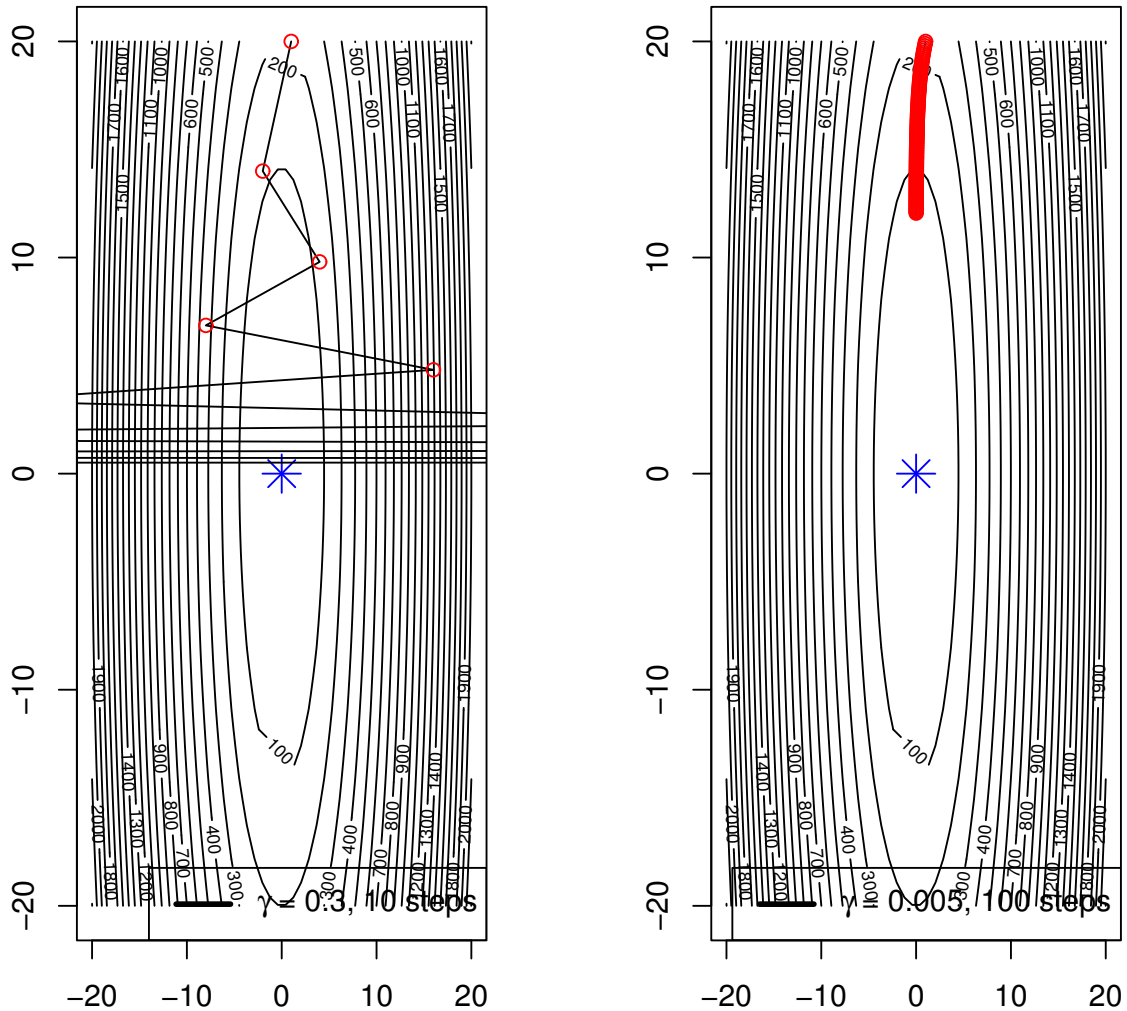


# Gradient descent: some math

- Suppose we want to minimize the function $f(\boldsymbol{x})$
- For any given point $\boldsymbol{x} \in R^n$ with $f(\boldsymbol{x})$, suppose we want to find a point $\boldsymbol{x}_1 \in R^n$ such that $f(\boldsymbol{x}_1) < f(\boldsymbol{x})$
- Fact: when $\|\boldsymbol{x}_1 - \boldsymbol{x}\|$ is small, we know $f(\boldsymbol{x}_1) \approx f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{x}_1 - \boldsymbol{x})$
- We want to choose $\boldsymbol{x}_1$ such that $\boldsymbol{\epsilon} = \boldsymbol{x}_1 - \boldsymbol{x}$ will give the greatest decrease of $f$
- Fact: for a given vector $\boldsymbol{x}$, the inner product of $\boldsymbol{x}^t\boldsymbol{y}$ is minimized when $\boldsymbol{y}$ moves towards the opposite direction of $\boldsymbol{x}$, $\boldsymbol{x}^t\boldsymbol{y} =< \boldsymbol{x}, \boldsymbol{y} >= \|\boldsymbol{x}\|\|\boldsymbol{y}\| \cos(\theta)$, which is minimized when $\cos(\theta) = -180$

# How to choose steps

- Fixed step size: simply take $\gamma_k = \gamma$, for $k = 1, 2, \ldots$
- It can diverge if $\gamma$ is too large; can be slow if $\gamma$ is too small



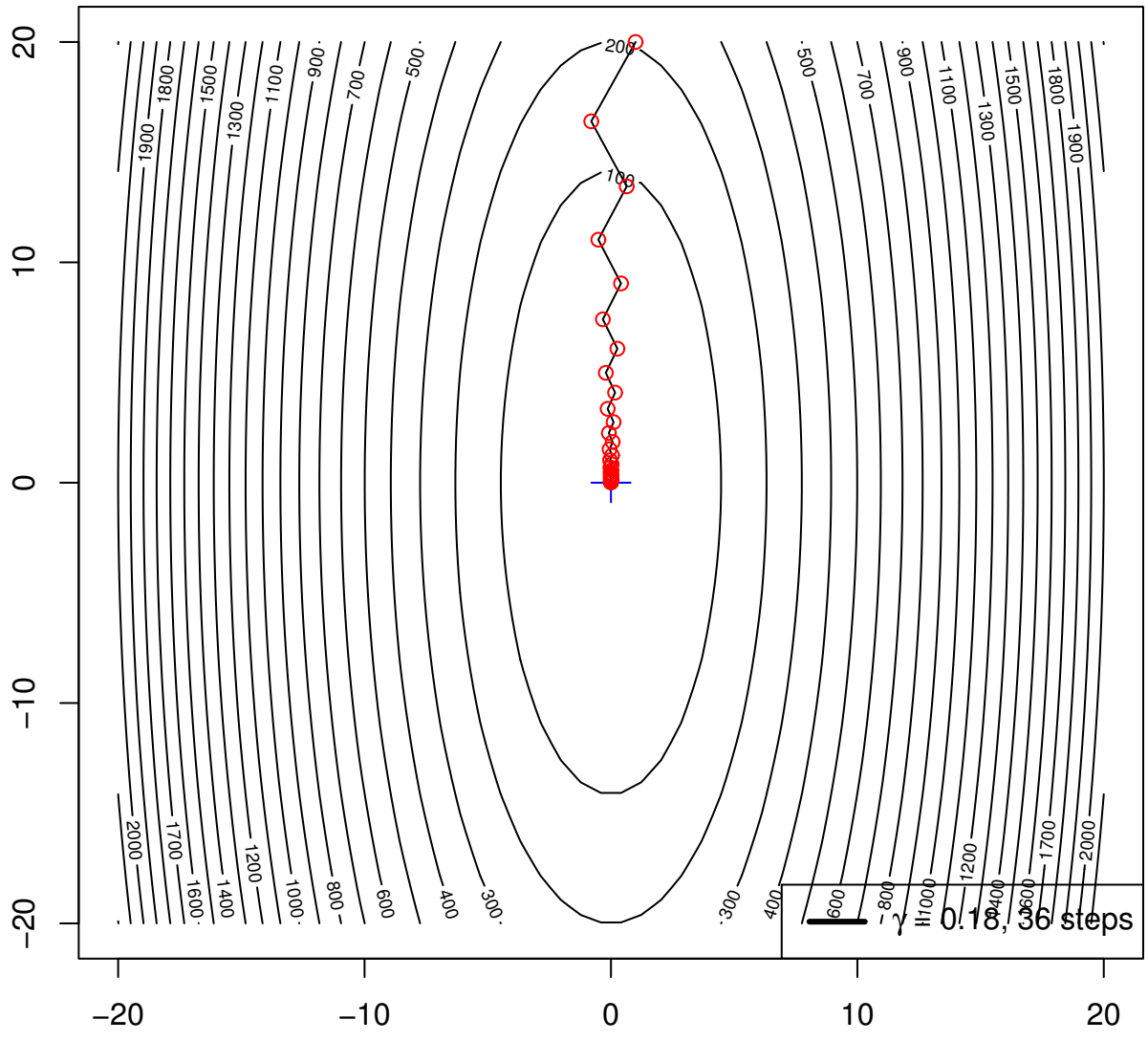
# R codes for the previous plots

```
#How to choose steps
f = function(x){(10*x[1]^2 + x[2]^2)/2}
df = function(x){c(10*x[1],x[2])}
lx = ly = -20;ux = uy = 20
g = 0.3

x = seq(lx,ux,length = 50)
y = seq(ly,uy,length = 50)
zz = outer(x,y, FUN = function(x,y) (10*x^2 + y^2)/2)

z = c(1,20)
z1 = z - g*df(z)
r = cbind(rbind(z,z1),c(f(z),f(z1)))
for(i in 1:10){
  a = z1
  z1 = z1 - g*df(z1)
  z = a
  r = rbind(r,c(z1,f(z1)))
}

par(mfrow = c(1,2))
contour(x,y,zz, nlevels = 30, xlim = c(lx,ux),ylim = c(ly,uy))
points(0,0,pch =8, col = "blue", cex = 2)
lines(r[,1],r[,2])
points(r[,1],r[,2],col = "red")
legend("bottomright", border = "green", lwd = 3,
       legend = expression(paste(gamma, " = 0.3, 10 steps")))
```

# R codes for the previous plots

```
g = 0.005

x = seq(lx,ux,length = 50)
y = seq(ly,uy,length = 50)
zz = outer(x,y, FUN = function(x,y) (10*x^2 + y^2)/2)

z = c(1,20)
z1 = z - g*df(z)
r = cbind(rbind(z,z1),c(f(z),f(z1)))
for(i in 1:100){
  a = z1
  z1 = z1 - g*df(z1)
  z = a
  r = rbind(r,c(z1,f(z1)))
}
contour(x,y,zz, nlevels = 30, xlim = c(lx,ux),ylim = c(ly,uy))
points(0,0,pch = 8, col = "blue", cex = 2)
lines(r[,1],r[,2])
points(r[,1],r[,2],col = "red")
legend("bottomright", border = "green", lwd = 3,
      legend = expression(paste(gamma,  " = 0.005, 100 steps")))
```

# When the step size is right



# R codes for the previous plot

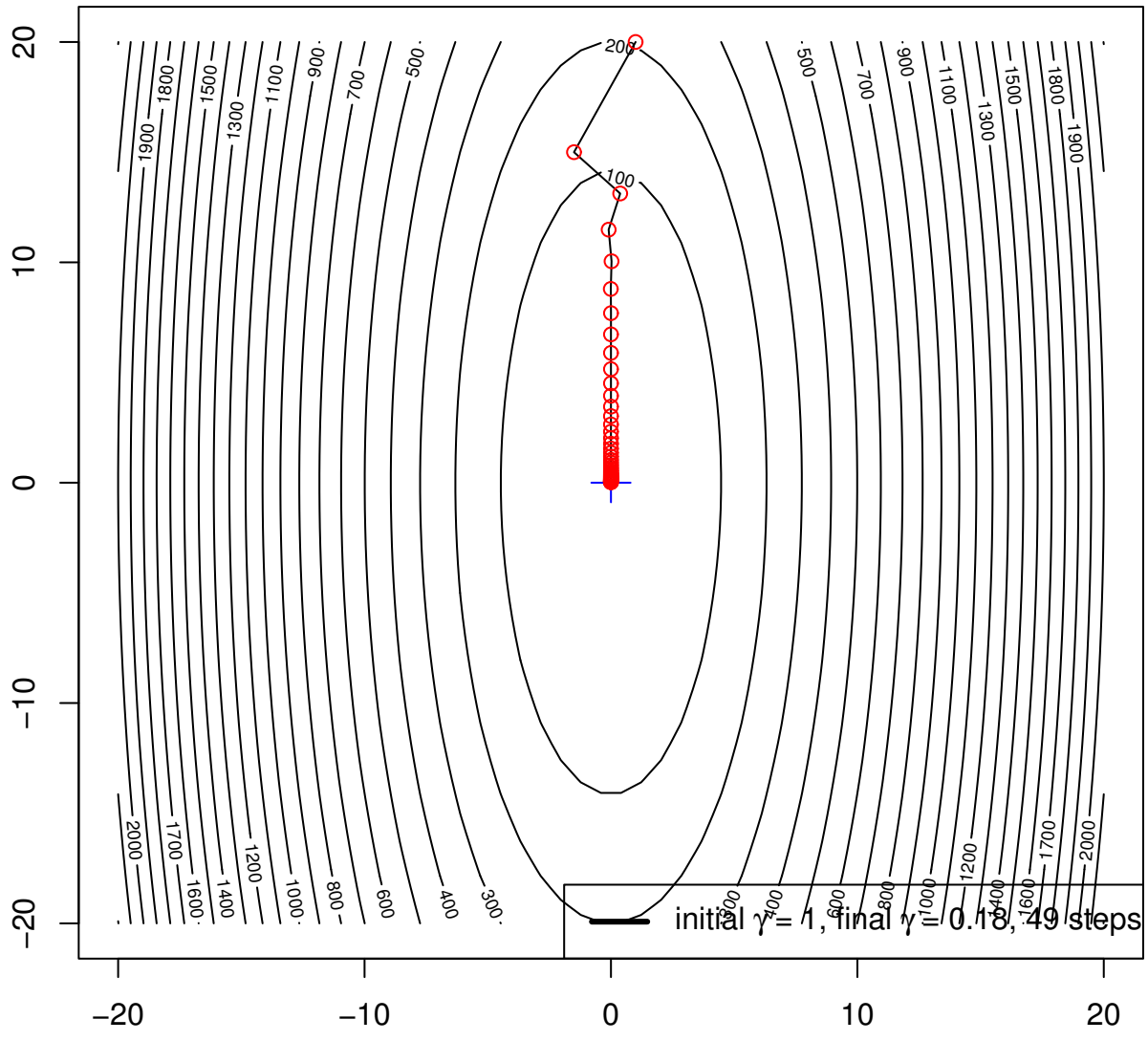
```
## When the Step Size Is Right
g = 0.18
x = seq(lx,ux,length = 50)
y = seq(ly,uy,length = 50)
zz = outer(x,y, FUN = function(x,y) (10*x^2 + y^2)/2)

z = c(1,20)
z1 = z - g*df(z)
r = cbind(rbind(z,z1),c(f(z),f(z1)))
while(abs(f(z1)-f(z))>0.0001){
  a = z1
  z1 = z1 - g*df(z1)
  z = a
  r = rbind(r,c(z1,f(z1)))
}
contour(x,y,zz, nlevels = 30, xlim = c(lx,ux),ylim = c(ly,uy))
points(0,0,pch =3, col = "blue", cex = 2)
lines(r[,1],r[,2])
points(r[,1],r[,2],col = "red")
legend("bottomright", border = "green", lwd = 3,
       legend = expression(paste(gamma, " = 0.18, 36 steps")))
```

# Backtracking line search

Choose the step size adaptively

- Fix two parameters $0 < \beta < 1$, and $0 < \alpha \leq \frac{1}{2}$
- At each iteration, start with $\gamma$
  - if $f(\boldsymbol{x} - \gamma \nabla f(\boldsymbol{x})) > f(\boldsymbol{x}) - \alpha\gamma \|\nabla f(\boldsymbol{x})\|_2^2$, shrink $\gamma = \beta\gamma$
  - else perform $\boldsymbol{x}^{new} = \boldsymbol{x} - \gamma \nabla f(\boldsymbol{x})$

# Example revisited ($\alpha = \beta = 0.5$)



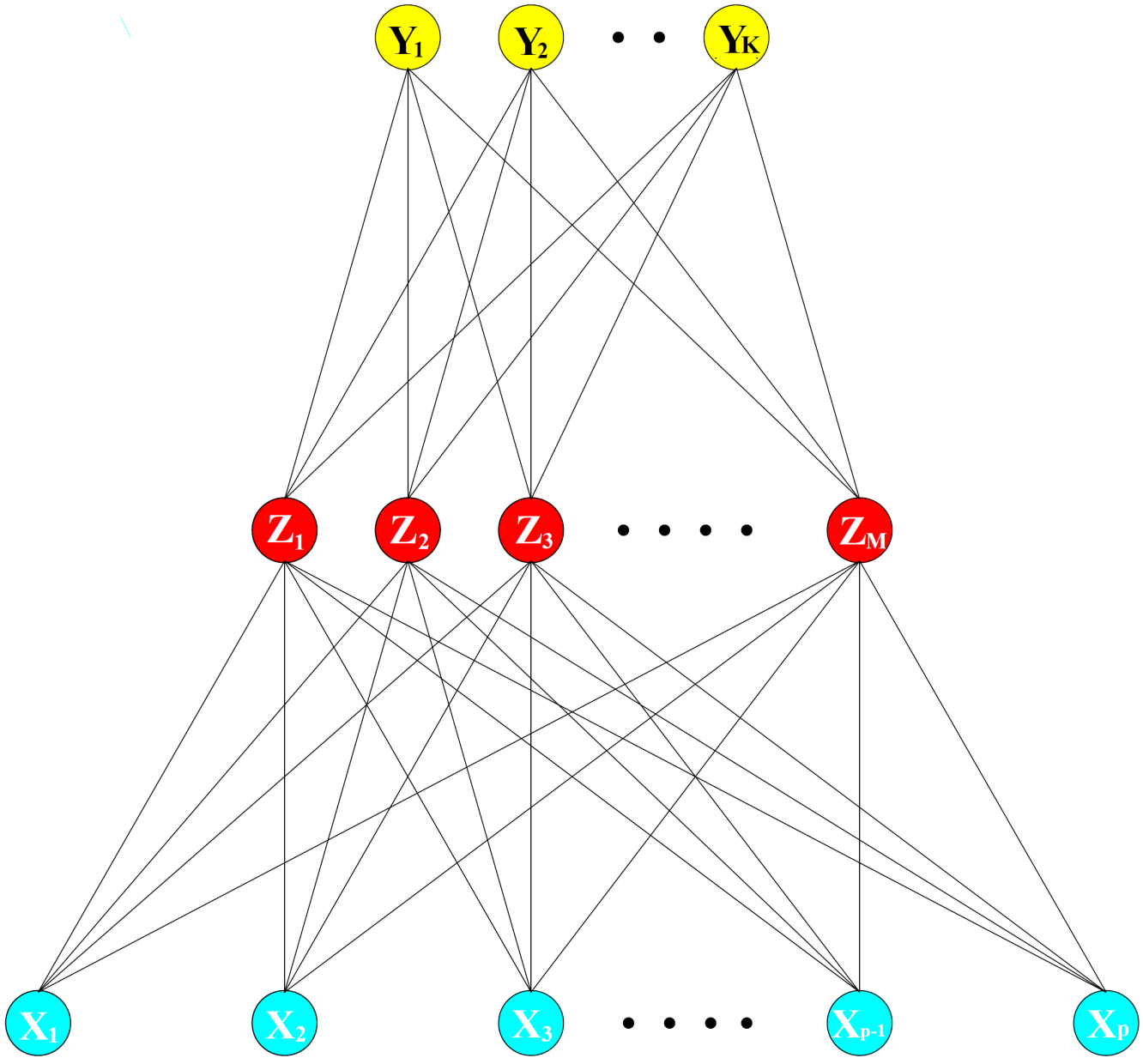
# R codes for the previous plot

```
## Example Revisited
library(geometry)
alpha = beta = 0.5; g = 1
z = c(1,20)
c = 0
while(f(z - g*df(z)) > f(z) - alpha * g * dot(df(z),df(z))){
  g = beta*g
  c = c+1
}
z1 = z - g*df(z)
r = cbind(rbind(z,z1),c(f(z),f(z1)))
d = 1
while(abs(f(z1)-f(z))>0.0001){
  a = z1
  while(f(a - g*df(a)) > f(a) - alpha * g * dot(df(a),df(a))){
    g = beta*g
    c = c+1
  }
  z1 = z1 - g*df(z1)
  z = a
  r = rbind(r,c(z1,f(z1)))
  d = d+1
}
contour(x,y,zz, nlevels = 30, xlim = c(lx,ux),ylim = c(ly,uy))
points(0,0,pch =3, col = "blue", cex = 2)
lines(r[,1],r[,2])
points(r[,1],r[,2],col = "red")
legend("bottomright", border = "green", lwd = 3,
       legend = expression(paste("initial ",gamma, " = 1", ", final ", gamma, " = 0.18, 49 steps")))
```

# Neural networks

- The term *neural network* has evolved to encompass a large class of models and learning methods.
- We focus the most widely used "vanilla" neural net, sometimes called the single hidden layer back-propagation network, or single layer perceptron.
- The central idea is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features.
- Neural network can be viewed as a multi-stage regression or classification model, typically represented by a *network diagram*.

# Schematic of a Single Hidden Layer Network



# Neural Networks

- For $K$-class classification, there are $K$ units at the top, with the $k$th unit modeling the probability of class $k$. There are $K$ target measurements $Y_k$, $k = 1, \ldots, K$, each being coded as a 0/1 variable for the $k$th class.
- Derived features $Z_m$ are created from linear combinations of the inputs, and then the target $Y_k$ is modeled as a function of linear combinations of the $Z_m$

$$
\begin{aligned}
Z_m &= \sigma(\alpha_{0m} + \boldsymbol{\alpha}_m^T \boldsymbol{X}), m = 1, \ldots, M \\
T_k &= \beta_{0k} + \boldsymbol{\beta}_k^T \boldsymbol{Z}, k = 1, \ldots, K \\
f_k(\boldsymbol{X}) &= g_k(\boldsymbol{T}), k = 1, \ldots, K
\end{aligned}
$$

where $\boldsymbol{Z} = (Z_1, \ldots, Z_M)$ and $\boldsymbol{T} = (T_1, \ldots, T_K)$

# Sigmoid activation function and softmax

- The activation function is $\sigma(v)$ is usually chosen to be the sigmoid
  - Sometimes Gaussian radial basis functions are used for the $\sigma(v)$, producing what is known as a radial basis function network
- Often, the neural network diagram drawn as in last page has additional bias unit feeding into every unit in the hidden and output layers
  - Such bias unit captures the intercepts $\alpha_{0m}$ and $\beta_{0k}$ in the previous model
- The output function $g_k(\boldsymbol{T})$ allows a final transformation of the vector of outputs $\boldsymbol{T}$
  - For regression typically the identity function $g_k(\boldsymbol{T}) = T_k$ is chosen
  - In $K$-class classification people prefer the *softmax* function

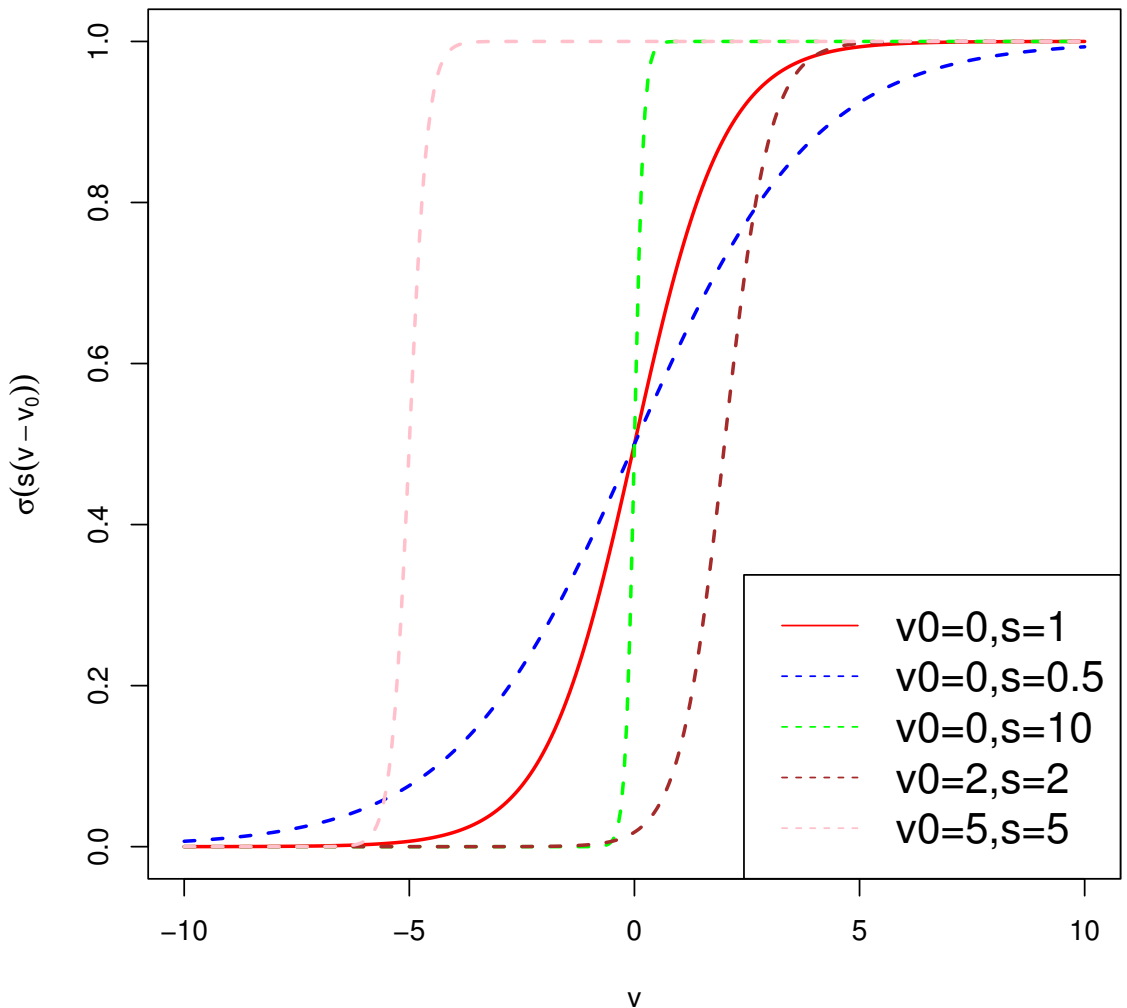
$$g_k(\boldsymbol{T}) = \frac{e^{T_k}}{\sum_{\ell=1}^{K} e^{T_\ell}}, \text{ for } k = 1, \ldots, K$$

# Hidden layers and activation functions $\sigma$

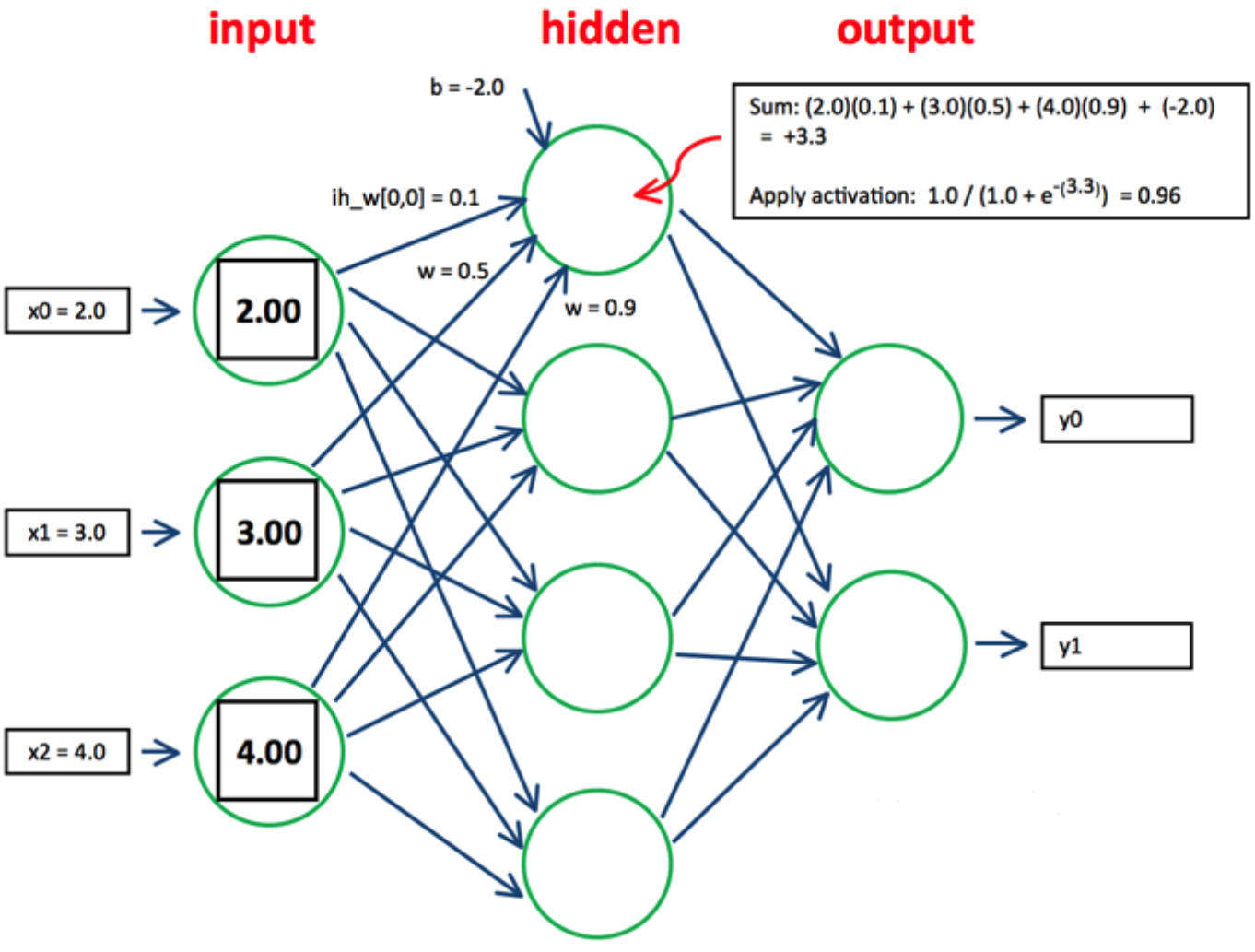
- The derived features $Z_m$, are often called hidden units because the values $Z_m$ are not directly observed
  - Another name in Statistics about these features are called latent variables
- In general there can be more than one hidden layer
  - Usually a network with more than two layers (including outputs) is a deep learning network
- On the other hand, We can think of the $Z_m$ as a basis expansion of the original inputs $\boldsymbol{X}$
  - The neural network is then a standard linear model, or linear multinomial model, etc.
- If the activation function $\sigma$ is the density function, then the entire model collapses to a linear model in the inputs
  - A neural network can be thought of as a nonlinear generalization of the linear model, both for regression and classification
  - By introducing the nonlinear transformation $\sigma$, it greatly enlarges the class of linear models

# A better understanding of the sigmoid function

- $$\sigma(s(v - v_0)) = \frac{1}{1 + e^{-s(v-v_0)}}$$



# An illustration



# An example: infert data

- Infertility after spontaneous and induced abortion
  - This is a matched case-control study
- Use four input variables, *age*, *parity*, *induced*, and *spontaneous* to predict the binary output variable: case.

# An example: infert data

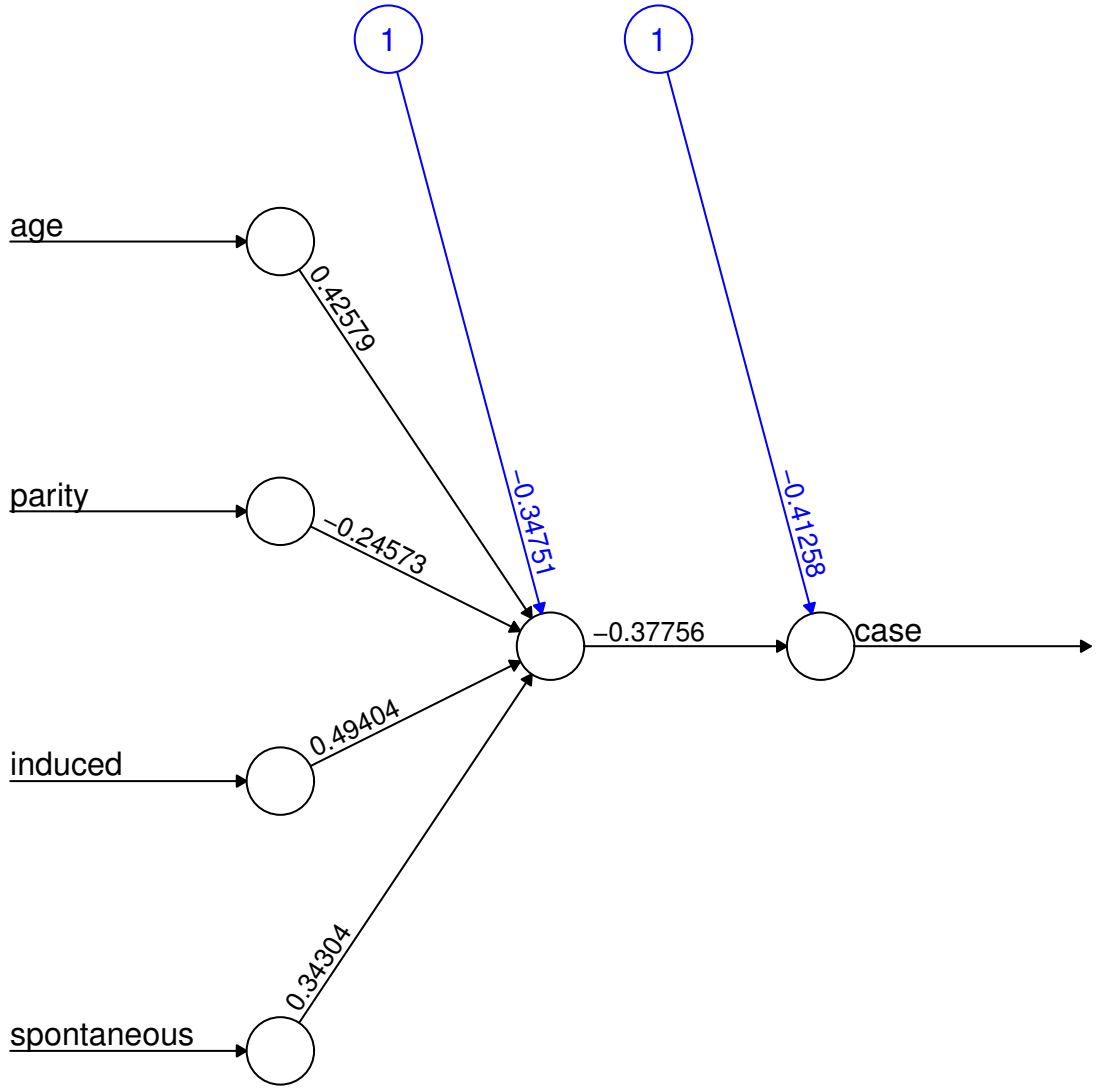
```
> ## An example: infert data
> df = infert[,c("age","parity","induced","spontaneous","case")]
> summary(df)
      age            parity         induced        spontaneous
 Min.   :21.00   Min.   :1.000   Min.   :0.0000   Min.   :0.0000
 1st Qu.:28.00   1st Qu.:1.000   1st Qu.:0.0000   1st Qu.:0.0000
 Median :31.00   Median :2.000   Median :0.0000   Median :0.0000
 Mean   :31.50   Mean   :2.093   Mean   :0.5726   Mean   :0.5766
 3rd Qu.:35.25   3rd Qu.:3.000   3rd Qu.:1.0000   3rd Qu.:1.0000
 Max.   :44.00   Max.   :6.000   Max.   :2.0000   Max.   :2.0000
      case
 Min.   :0.0000
 1st Qu.:0.0000
 Median :0.0000
 Mean   :0.3347
 3rd Qu.:1.0000
 Max.   :1.0000
> |
```

# Case versus others

```
> library(neuralnet)
> set.seed(1)
> tr.ind = sample(seq_len(nrow(df)),floor(0.7*nrow(df)))
> df.tr = df[tr.ind,]
> df.te = df[-tr.ind,]
> inf.nn = neuralnet(case ~ ., data = df.tr, hidden = 1, err.fct = "ce", linear.output = F, rep = 3)
> # linear.output should be FALSE for categorical outputs
> # Default act.fct is "sigmoid" (logistic)
> # err.fct is either "sse" or "ce" (cross-entropy)
> # hidden is the number of nodes in the layer
> #
> # Default algorithm is "Resilient Backpropagation" (rprop+) with weight backtracking
>
> inf.nn$weights[[1]][[1]]
           [,1]
[1,] -0.3475133
[2,]  0.4257883
[3,] -0.2457305
[4,]  0.4940425
[5,]  0.3430388
> inf.nn$weights[[1]][[2]]
           [,1]
[1,] -0.4125794
[2,] -0.3775565
>
```

# Neural network plot (rep = 1)

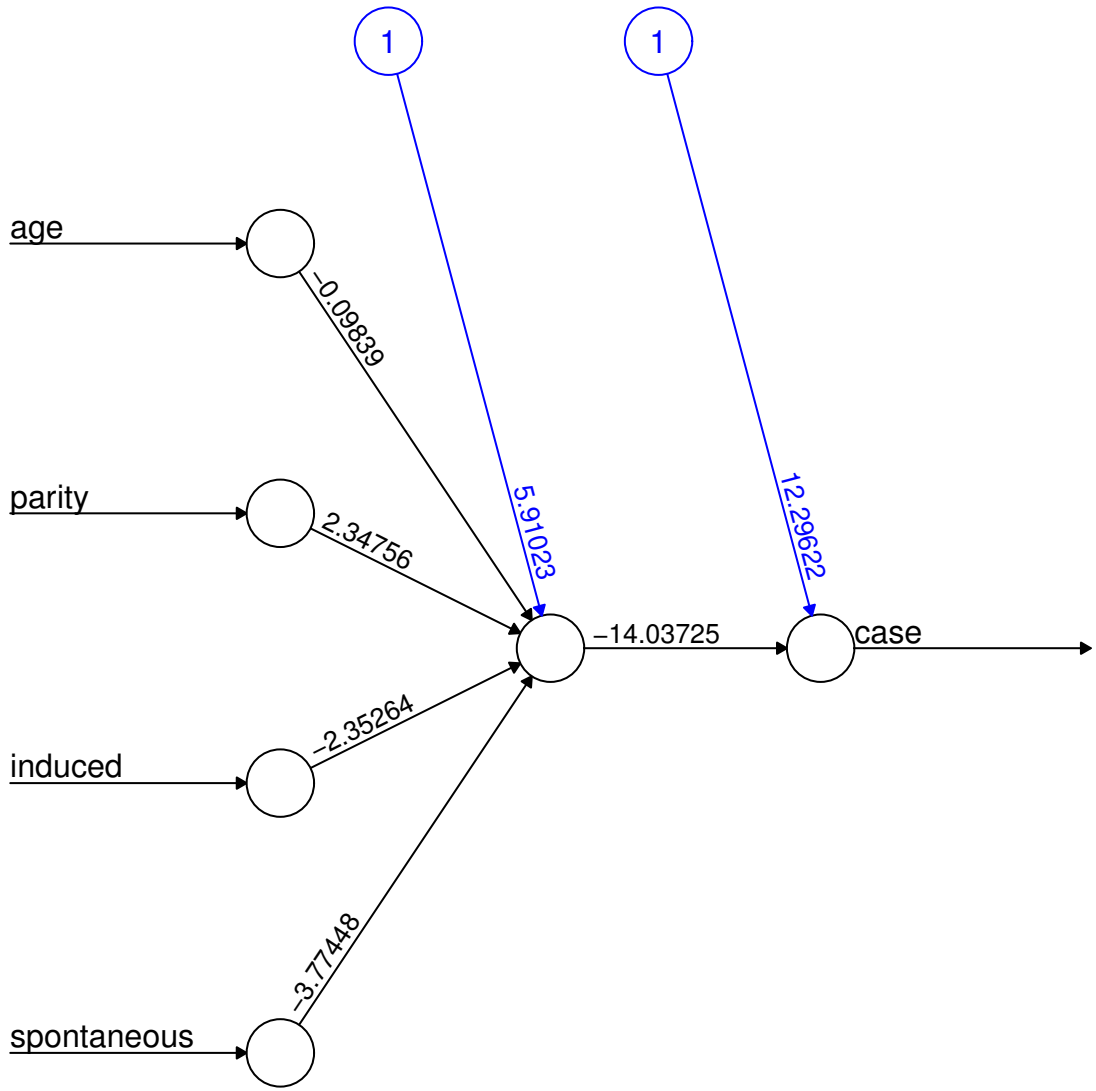
- plot(inf.nn, rep = 1)




Error: 107.398571  Steps: 11

# Neural network plot (rep = 2)

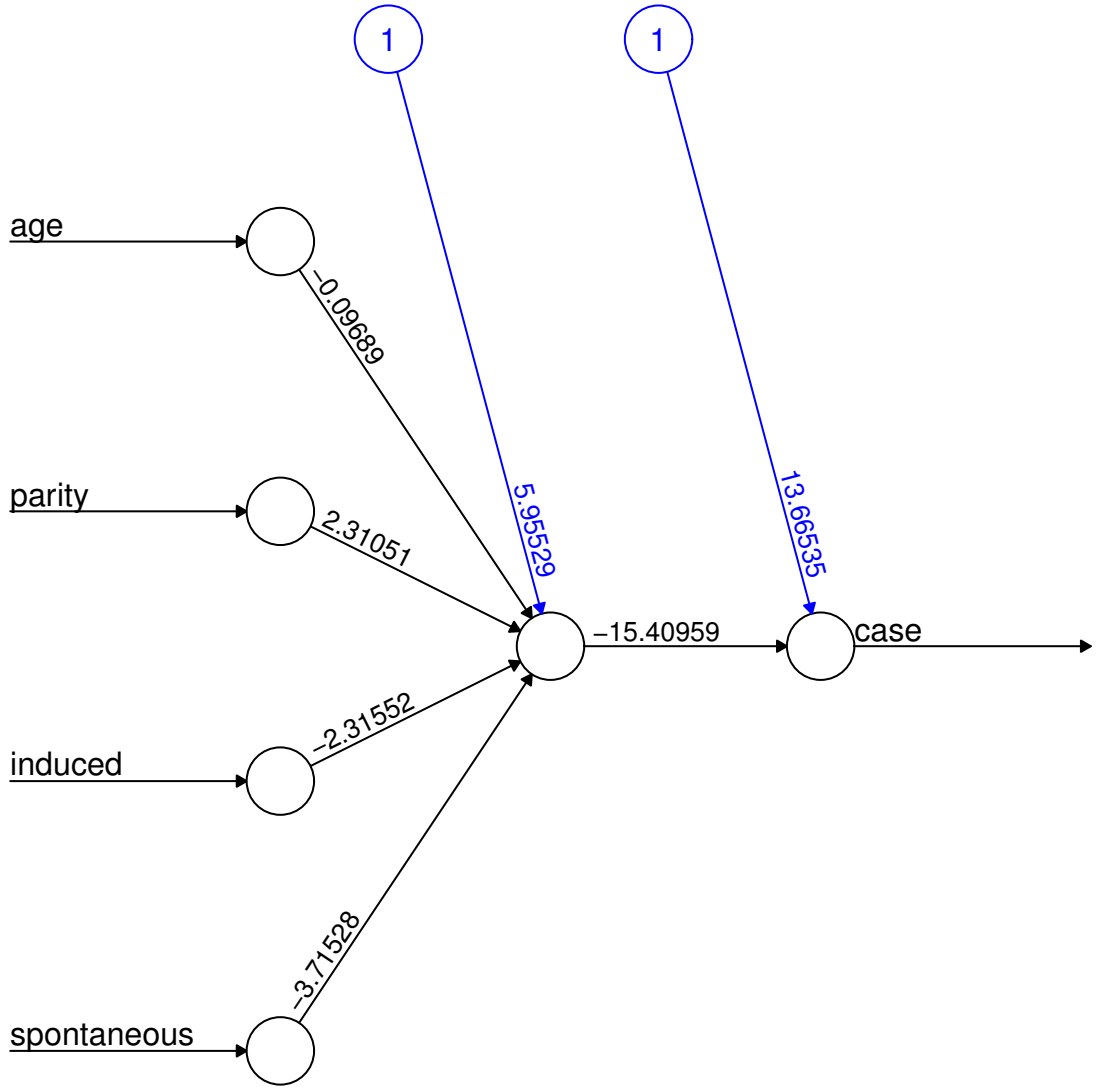
- plot(inf.nn, rep = 2)




Error: 80.787765  Steps: 8445

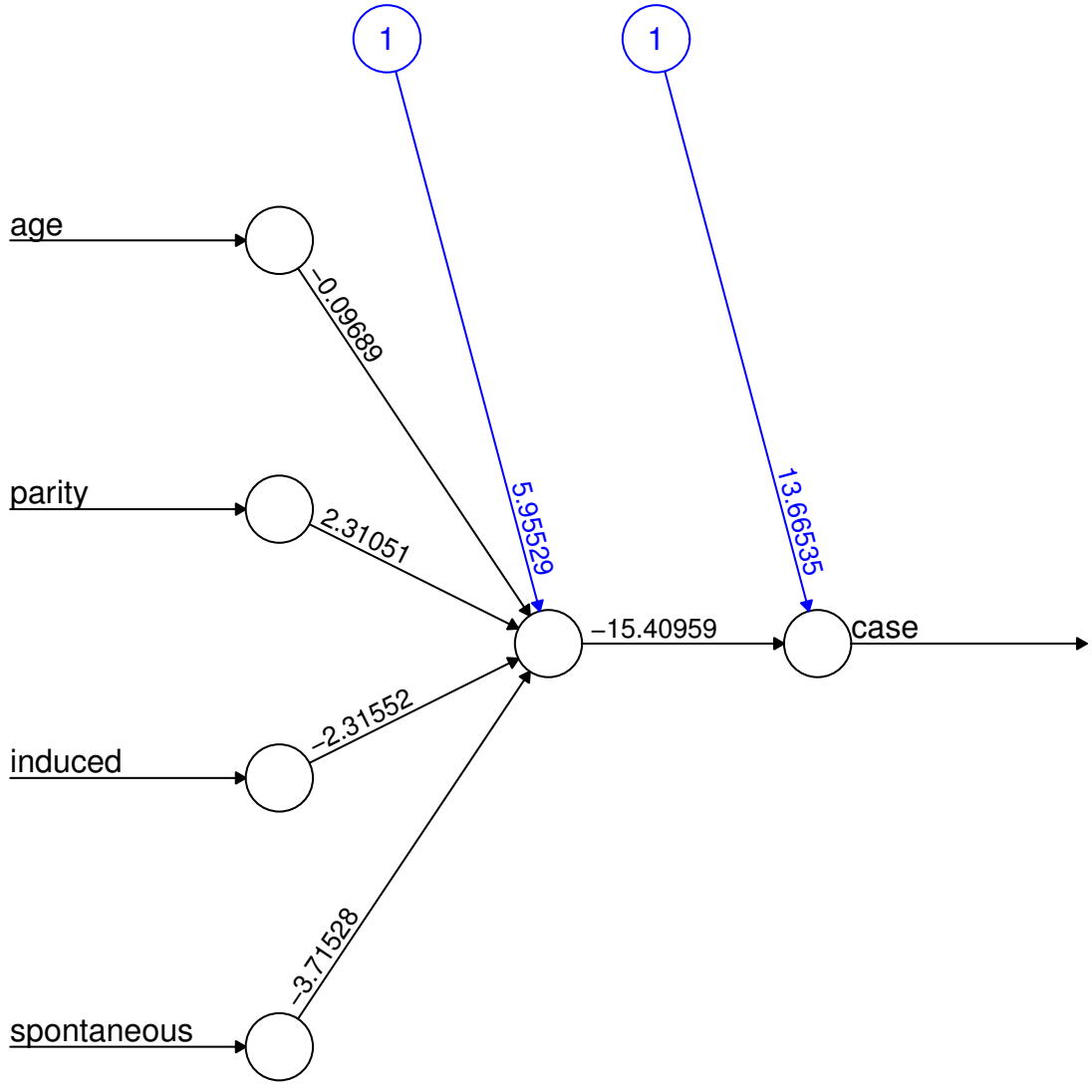
# Neural network plot (rep = 3)

- plot(inf.nn, rep = 3)




Error: 80.765512  Steps: 31915

# Neural network plot (lowest error)




Error: 80.765512  Steps: 31915

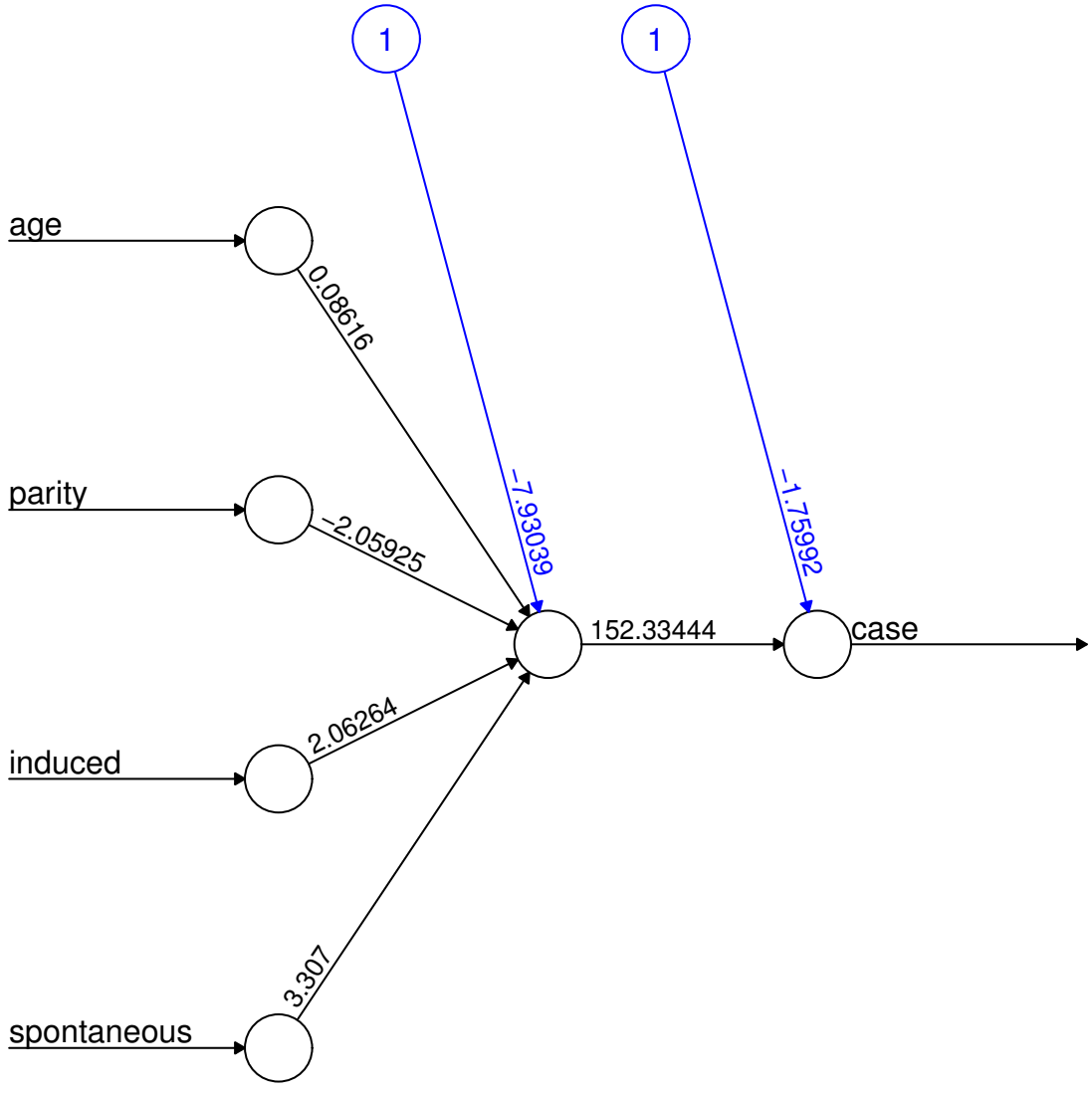
# R codes for the previous plots

```
> ## Neural network plot (rep = 1)
> plot(inf.nn, rep = 1)
> ## Neural network plot (rep = 2)
> plot(inf.nn, rep = 2)
>
> ## Neural network plot (rep = 3)
> plot(inf.nn, rep = 3)
> ## Neural network plot (lowest error)
> plot(inf.nn, rep = "best"); rep.best = which.min(inf.nn$result.matrix[1,])
> rep.best = which.min(inf.nn$result.matrix[1,])
> rep.best
[1] 3
> ## Neural network plot (lowest error)
> plot(inf.nn, rep = "best");
> rep.best = which.min(inf.nn$result.matrix[1,])
> rep.best
[1] 3
```

# Neural network plot (lowest error)

```
> ## NN weights details
>
> inf.pred = predict(inf.nn, df.te[1,],all.units = T, rep = rep.best)
> # 1. Use 1st line in the test set; 2. all.units = T shows details
> inf.pred
[[1]]
  age parity induced spontaneous
3  39      6       2           0

[[2]]
       [,1]
3 0.9999889

[[3]]
       [,1]
3 0.1487966

> n1 = inf.nn$weights[[rep.best]][[1]][1] + as.matrix(df.te[1,1:4]) %*%
+   inf.nn$weights[[rep.best]][[1]][2:5]
> nn1 = 1/(1+exp(-n1))
> n2 = inf.nn$weights[[rep.best]][[2]][1] + nn1 *
+   inf.nn$weights[[rep.best]][[2]][2]
> nn2 = 1/(1+exp(-n2))
> print(c(nn1,nn2))
[1] 0.9999889 0.1487966
> df.te[1,]
  age parity induced spontaneous case
3  39      6       2
```

# One more try

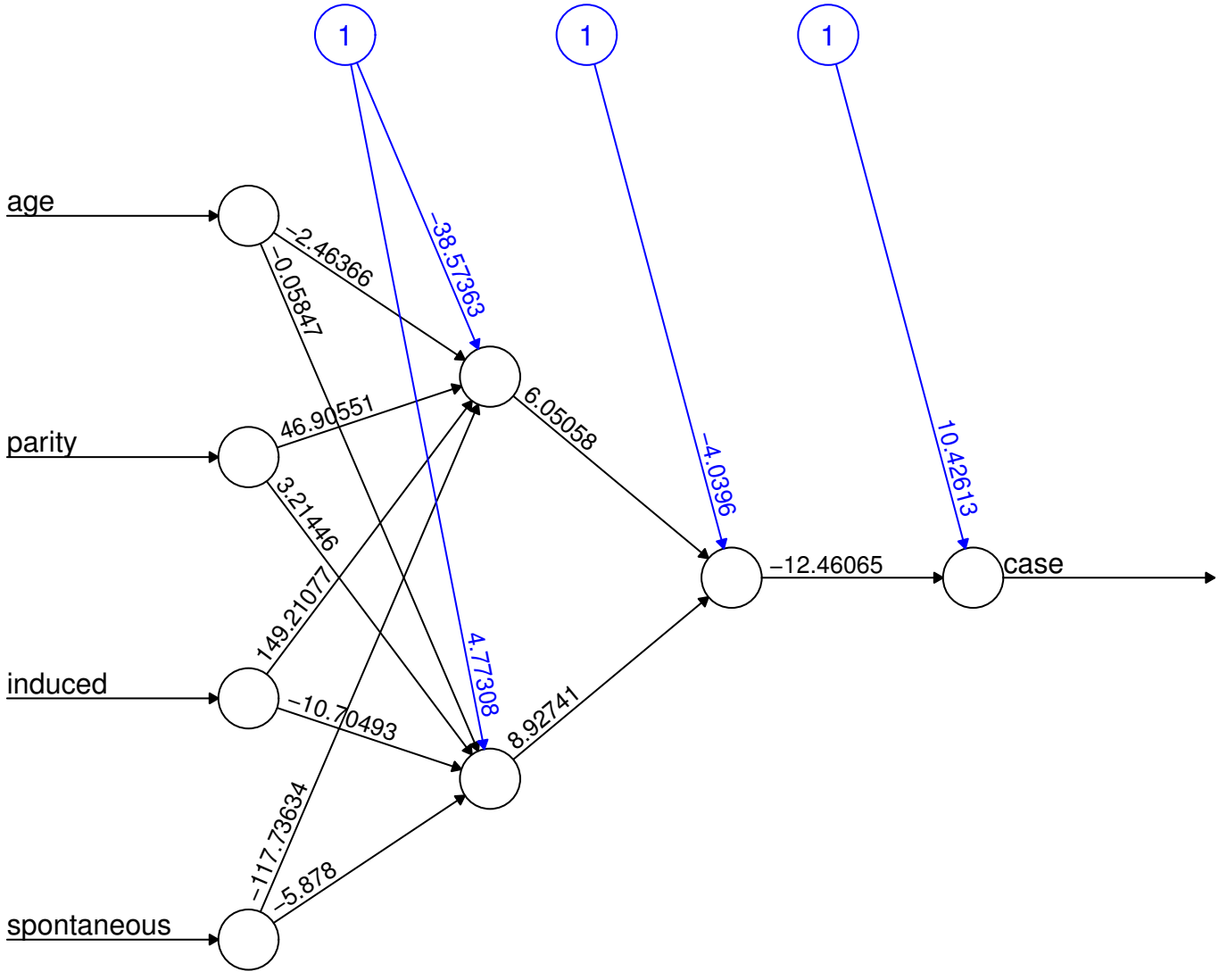



Error: 80.617812  Steps: 8040

# R codes for the previous plot

```
> ## One more try
> set.seed(12345)
> inf.nn1 = neuralnet(case ~ ., data = df.tr, hidden = 1, err.fct = "ce", linear.output = F)
> inf.pred = predict(inf.nn1, df.te[1,])
> plot(inf.nn1, rep = "best")
> inf.pred
      [,1]
3 0.1468525
>
```

# More nodes and layers in NN




Error: 77.554818 Steps: 16897

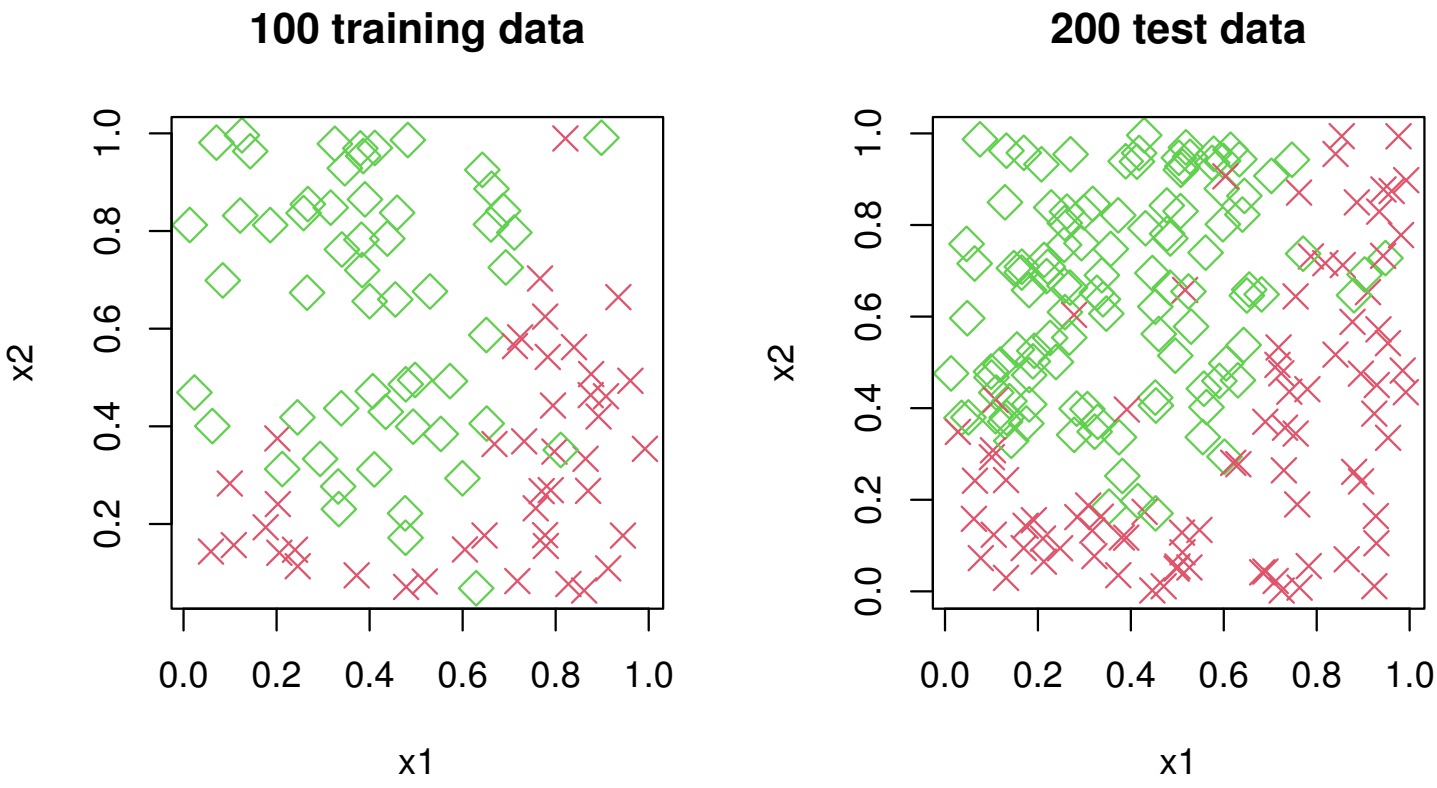
# R codes for the previous plot

```
> ## More nodes and layers in NN
> set.seed(123)
> inf2.1.nn = neuralnet(case ~ ., data = df.tr, hidden = c(2,1), err.fct = "ce",
+   linear.output = F,stepmax = 1000000)
> plot(inf2.1.nn, rep = "best")
> predict(inf2.1.nn,df.te[1,])
      [,1]
3 0.116498
```

# Test errors

```
> ## Test Errors
> df.te[1,]
  age parity induced spontaneous case
3  39      6       2           0    1
> df.pred = predict(inf.nn, newdata = df.te, rep = rep.best)
> df1.pred = predict(inf.nn1, newdata = df.te)
> df2.1.pred = predict(inf2.1.nn, newdata = df.te)
> mean(round(df.pred,0) != df.te$case)
[1] 0.2666667
> mean(round(df1.pred,0) != df.te$case)
[1] 0.2666667
> mean(round(df2.1.pred,0) != df.te$case)
[1] 0.2933333
>
>
```

# A simulated data

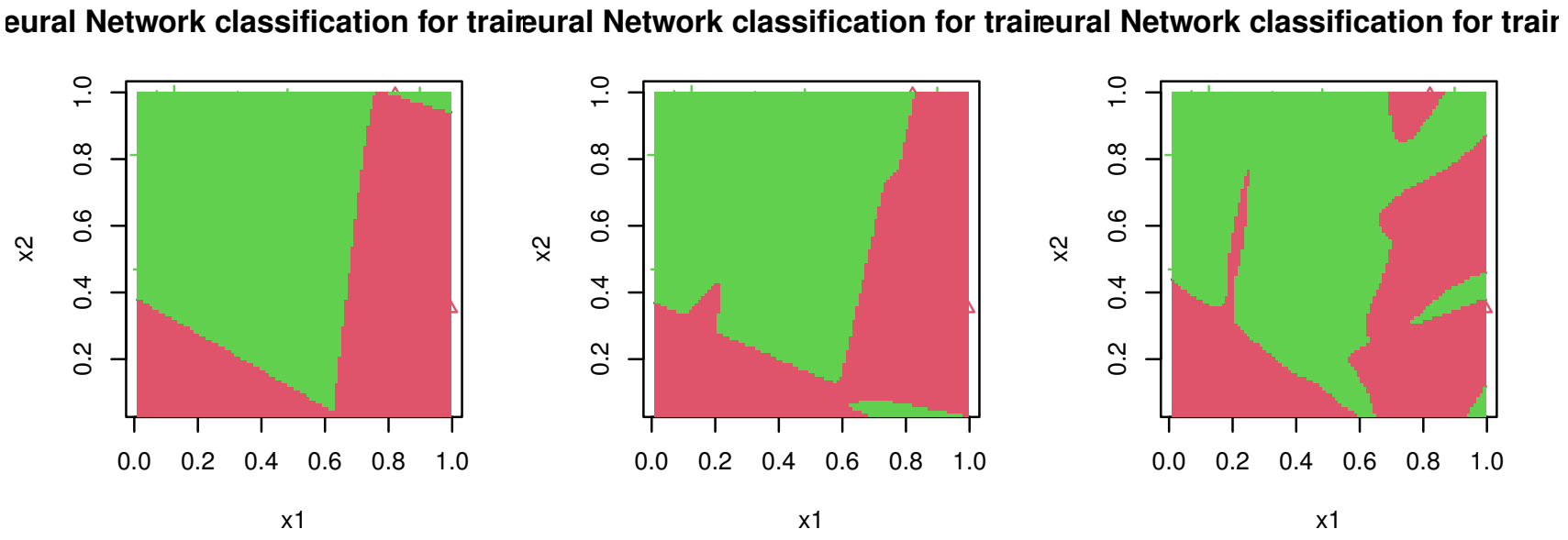


# R codes for the previous plot

```
## A simulated data
y.fun = function(x,alpha){
  (alpha*((2*x)^2*exp(-1+2*x +0.3*(2*x)^2)-2*x)+(1-alpha)*(0.5-(2*x)^2/2-0.8*(2*x)^3)+0.5)/2
}
m = 300
alpha = 0.52
beta = 0.05
set.seed(1)
x = runif(2*m)
z = rbinom(m,1,beta)
df = data.frame(x1=x[1:m],x2=x[(m+1):(2*m)], z= z)
df$y = ifelse(df$x2 > y.fun(df$x1,alpha),1,0)
df$y = as.factor(ifelse(df$z==1,1-df$y,df$y))
df$z = NULL

tr.ind = 1:100
df.tr = df[tr.ind,]
df.te = df[-tr.ind,]
par(mfrow = c(1,2))
par(pty = "s")
plot(df.tr$x1,df.tr$x2, xlab = "x1",ylab = "x2", col = as.numeric(df.tr$y)+1,
  pch = as.numeric(df.tr$y)+3, main = "100 training data",cex = 1.5)
par(pty = "s")
plot(df.te$x1,df.te$x2, xlab = "x1",ylab = "x2", main = "200 test data",col = as.numeric(df.te$y)+1,
  pch = as.numeric(df.te$y)+3,cex = 1.5)
```

# Representational power

- Neural network with at least one hidden layer is a universal approximator (can represent any function)

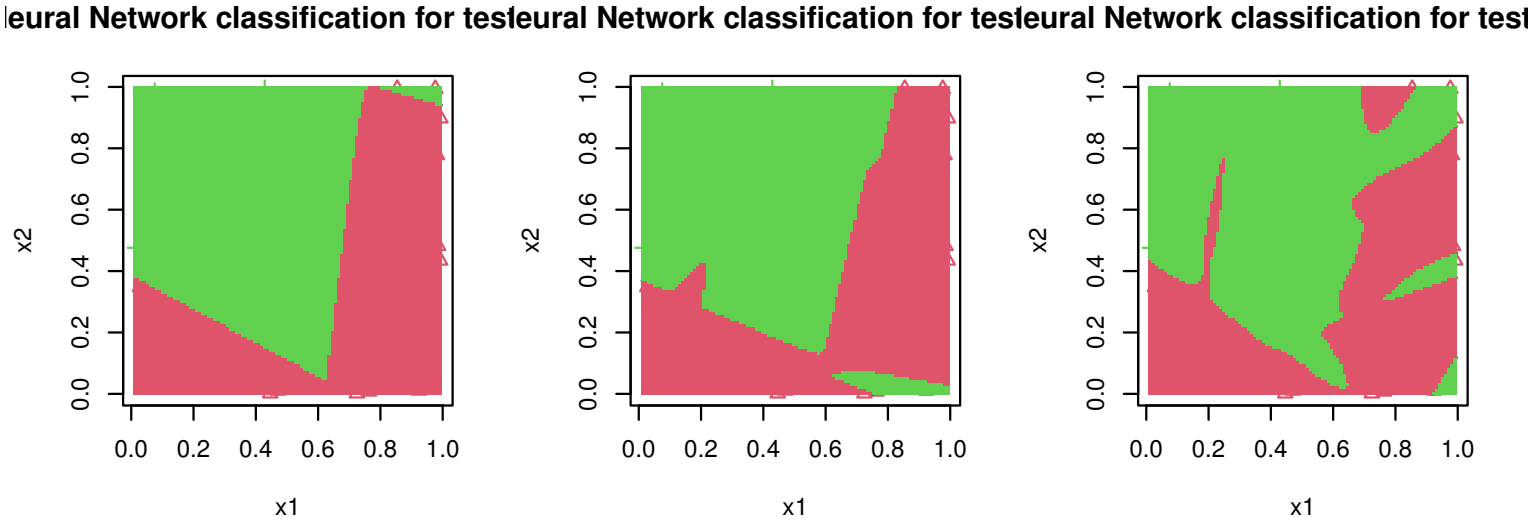


# R codes for the previous plot

```
> ## Representational Power
> par(mfrow = c(1,3))
> par(pty = "s")
> set.seed(1)
> sim.nn3 = neuralnet(y ~ ., df.tr, hidden = 3, linear.output = F, err.fct = "ce", stepmax = 1000000)
> class.plot(sim.nn3,df, train.index = tr.ind, method = "nn")
> set.seed(5)
> sim.nn6 = neuralnet(y ~ ., df.tr, hidden = 6, linear.output = F, err.fct = "ce", stepmax = 1000000)
> class.plot(sim.nn6,df, train.index = tr.ind, method = "nn")
> set.seed(12)
> sim.nn12 = neuralnet(y ~ ., df.tr, hidden = 12, linear.output = F, err.fct = "ce", stepmax = 1000000)
> class.plot(sim.nn12,df, train.index = tr.ind, method = "nn")
>
> print(rbind(c("train err for 3 nodes","train err for 6 nodes","train err for 12 nodes"),
+   c(mean(apply(predict(sim.nn3,df.tr),1,which.max)-1 != df.tr$y),
+     mean(apply(predict(sim.nn6,df.tr),1,which.max)-1 != df.tr$y),
+     mean(apply(predict(sim.nn12,df.tr),1,which.max)-1 != df.tr$y))))
     [,1]                  [,2]                  [,3]
[1,] "train err for 3 nodes" "train err for 6 nodes" "train err for 12 nodes"
[2,] "0.03"                  "0.03"                  "0"
>
```

# Test data

- The capacity of the network increases with more hidden units and more hidden layers
- Why go deeper? [Read Do Deep Nets Really Need to be Deep?, Jimmy Ba and Rich Caruana]



# R codes for the previous plot

```
## Test data
print(rbind(c("test err for 3 nodes","test err for 6 nodes","test err for 12 nodes"),
  c(mean(apply(predict(sim.nn3,df.te),1,which.max)-1 != df.te$y),
    mean(apply(predict(sim.nn6,df.te),1,which.max)-1 != df.te$y),
    mean(apply(predict(sim.nn12,df.te),1,which.max)-1 != df.te$y))))

par(mfrow = c(1,3))
par(pty = "s")
class.plot(sim.nn3,df, train.index = tr.ind, method = "nn",train = F)
class.plot(sim.nn6,df, train.index = tr.ind, method = "nn",train = F)
class.plot(sim.nn12,df, train.index = tr.ind, method = "nn",train = F)
```

# Fitting neural networks

- For all the weights, we denoted them by $\boldsymbol{\theta}$, which consists of

$$\{\alpha_{0m}, \alpha_m; m = 1, \ldots, M\} M(p+1) \text{ weights},$$
$$\{\beta_{0k}, \beta_k; k = 1, 2, \ldots, K\} K(M+1) \text{ weights}$$

- For regression, we use sum-of-squared errors as our measure of fit (error function)

$$R(\boldsymbol{\theta}) = \sum_{k=1}^{K} \sum_{i=1}^{n} (y_{ik} - f_k(\boldsymbol{x}_i))^2$$

- For classification we use either squared error or deviance

$$R(\boldsymbol{\theta}) = -\sum_{k=1}^{K} \sum_{i=1} y_{ik} \log f_k(\boldsymbol{x}_i)$$

with a corresponding classifier as $G(\boldsymbol{x}) = \arg\max_k f_k(\boldsymbol{x})$

- With the softmax activation function and the cross-entropy error function, the neural network model is exactly a linear logistic regression model in the hidden units, and all the parameters are estimated by maximum likelihood

# Fitting neural networks (continued)

- Typically we do not want the global minimizer of $R(\theta)$
  - This is likely to overfit
- Instead some regularization is needed
  - achieved directly through a penalty term
  - or indirectly by early stopping
- The generic approach to minimizing $R(\theta)$ is by gradient descent
  - It is called **backpropagation**
  - This can be computed by a forward and backward sweep over the network, keeping track only of quantities local to each unit

# Backpropagation

- Consider in the regression case,

$$R(\boldsymbol{\theta}) = \sum_{k=1}^{K} \sum_{i=1}^{n} (y_{ik} - f_k(\boldsymbol{x}_i))^2 = \sum_{i=1}^{n} R_i$$

with derivatives

$$\frac{\partial R_i}{\partial \beta_{km}} = -2(y_{ik} - f_k(\boldsymbol{x}_i)) g_k'(\boldsymbol{\beta}_k^T \boldsymbol{z}_i) z_{mi} = \delta_{ki} z_{mi},$$

$$\frac{\partial R_i}{\partial \alpha_{ml}} = -\sum_{k=1}^{K} 2(y_{ik} - f_k(\boldsymbol{x}_i)) g_k'(\boldsymbol{\beta}_k^T \boldsymbol{z}_i) \beta_{km} \sigma'(\boldsymbol{\alpha}_m^T \boldsymbol{x}_i) x_{il} = s_{mi} x_{il}$$

- A gradient descent update at the (r+1)st iteration is

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma_r \sum_{i=1}^{n} \frac{\partial R_i}{\partial \beta_{km}^{(r)}}, \quad \alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \gamma_r \sum_{i=1}^{n} \frac{\partial R_i}{\partial \alpha_{ml}^{(r)}} \quad (1)$$

where $\gamma_r$ is the learning rate

# Backpropagation equations

- The quantities $\delta_{ki}$ and $s_{mi}$ are "errors" from the current model at the output and hidden layer units, respectively
  - These errors satisfy the backpropagation equations

$$s_{mi} = \sigma'(\boldsymbol{\alpha}_m^T \boldsymbol{x}_i) \sum_{k=1}^{K} \beta_{km} \delta_{ki}$$

- Using the backpropagation equations, the gradient descent updates can be implemented with a two-pass algorithm
  - In the forward pass, the current weights are fixed and the predicted values $\hat{f}_k(\boldsymbol{x}_i)$ can be computed
  - In the backward pass, the errors $\delta_{ki}$ are computed, and then backpropagated via the backpropagation equations to give the errors $s_{mi}$

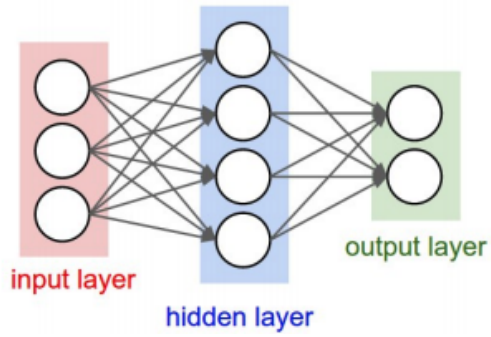
# Backpropagation, delta rule, and batch learning

- This two-pass procedure is what is known as backpropagation
  - It has also been called the delta rule (Widrow and Hoff, 1960)
- The advantages of backpropagation are its simple, local nature
  - In the back propagation algorithm, each hidden unit passes and receives information only to and from units that share a connection
  - It can be implemented efficiently on a parallel architecture computer
- The computational components for cross-entropy have the same form
- The updates in (1) are a kind of batch learning, with the parameter updates being a sum over all of the training cases
  - Learning can also be carried out online processing each observation one at a time, updating the gradient after each training case, and cycling through the training cases many times

# The learning rate and disadvantage of backpropagation

- The learning rate $\gamma_r$ for batch learning is usually taken to be a constant, and can also be optimized by a line search that minimizes the error function at each update
- With online learning $\gamma_r$ should decrease to zero as the iteration $r \to \infty$
  - This is a form of *stochastic approximation* (Robbins and Munro, 1951)
  - Results in this field ensure convergence if $\gamma_r \to 0, \sum_r \gamma_r = \infty$, and $\sum_r \gamma_r^2 < \infty$ (e.g., $\gamma_r = r^{-1}$)
- Backpropagation can be very slow, and for that reason is usually not the method of choice
- Better approaches to fitting include conjugate gradients and variable metric methods
  - These avoid explicit computation of the second derivative matrix while still providing faster convergence

# Forward pass: what does the network compute?



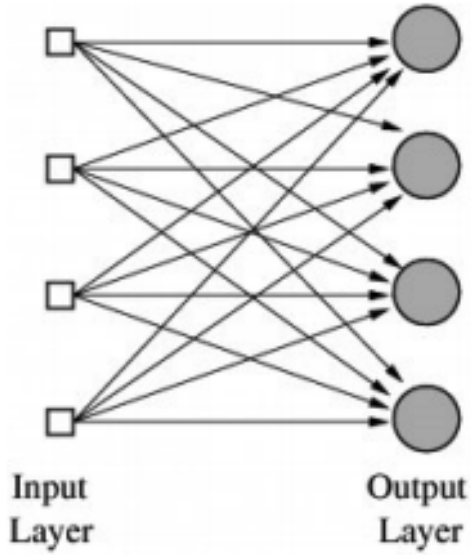

- Output of the network can be written as:

$$Z_m(\boldsymbol{x}) = \sigma(\alpha_{0m} + \boldsymbol{\alpha}^T \boldsymbol{x})$$
$$f_k(\boldsymbol{x}) = g_k(\beta_{0k} + \boldsymbol{\beta}_k^T \boldsymbol{Z})$$

- $m$ indexing hidden units and $k$ indexing the output units
- Activation functions $\sigma$, $g$: sigmoid/logistic, tanh, or rectified linear (ReLU)

# Special case

- What is a single layer (no hidden nodes) network with a sigmoid activation function?




- Network

$$\sigma_k(\boldsymbol{x}) = \frac{1}{1 + e^{-z_k}}$$

$$z_k = w_{0k} + \boldsymbol{w}_k^T \boldsymbol{x}$$

- Logistic regression!

# Training neural networks

- All we need are to find weights $\boldsymbol{\theta}$:

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{n} \text{loss}(f(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

  - Define a loss function, e.g.:
  - Squared-error loss: $\frac{1}{2}\sum_k (f_k(\boldsymbol{x}_i) - y_{ik})^2$
  - Cross-entropy loss: $-\sum_k y_{ik} \log(f_k(\boldsymbol{x}_i)$

- Gradient descent:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \lambda \frac{\partial R}{\partial \boldsymbol{\theta}^{(t)}}$$

  where $\lambda$ is the learning rate and $R$ is error/loss

# Useful derivatives

| Function name | Function form | derivative |
|---|---|---|
| Sigmoid | $\sigma(z) = \frac{1}{1+e^{-z}}$ | $\sigma(z)(1-\sigma(z))$ |
| Tanh | $\tanh(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$ | $\frac{1}{\cosh^2(z)}$[1] |
| ReLU | $\mathrm{ReLU}(z) = \max\{0, z\}$ | $\begin{cases} 1, & z > 0 \\ 0, & z \leq 0 \end{cases}$ |

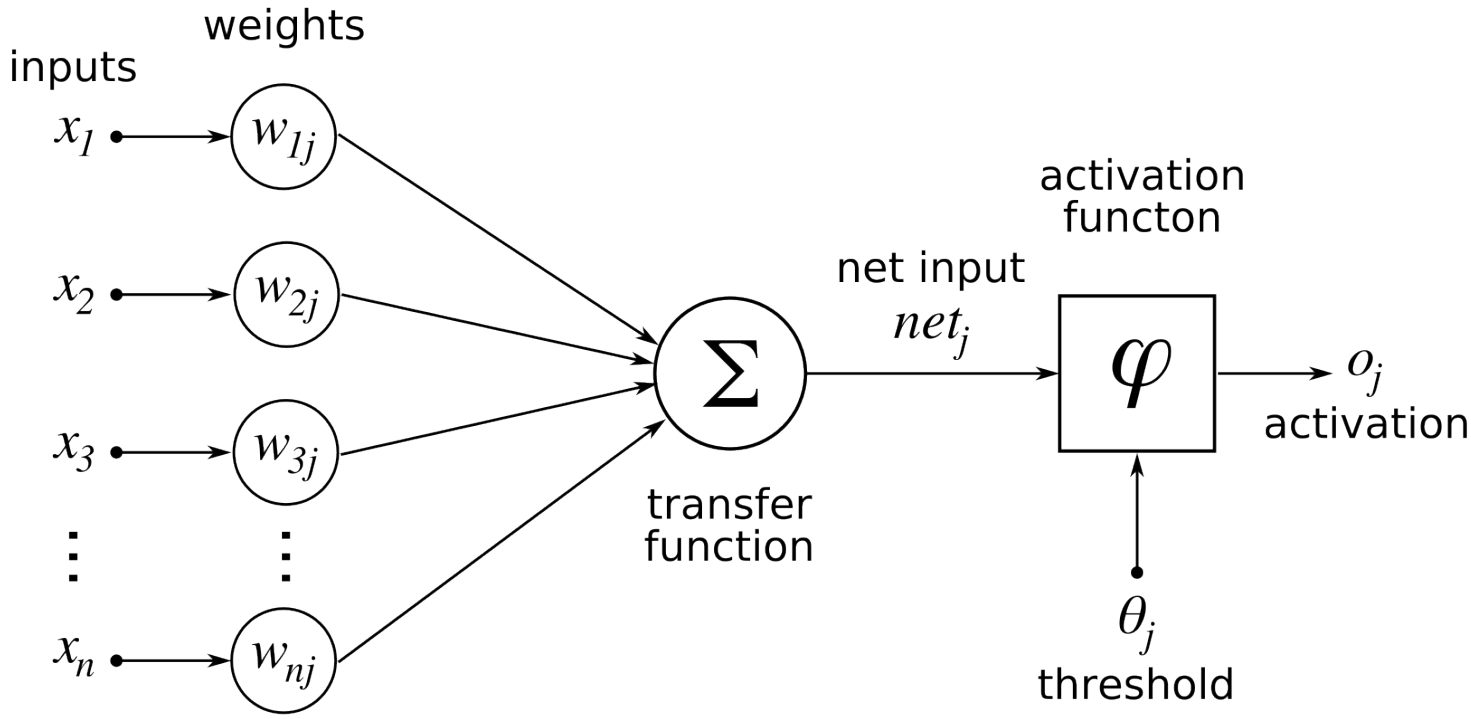
[1] $\cosh(z) = \frac{1}{2}(e^{z} + e^{-z})$

# Training neural networks: backpropagation

- **Backpropagation**: an efficient method for computing gradients needed to perform gradient-based optimization of the weights in a multi-layer network
- Training neural nets: Loop until convergence
  For each example $i$
  1. Given input $\boldsymbol{x}_i$, propagate activity forward ($\boldsymbol{x}_i \rightarrow \boldsymbol{Z}_i \rightarrow \boldsymbol{f}_i$) (forward pass)
  2. Propagate gradients backward (backward pass)
  3. Update each weight (via gradient descent)
- Given any error function $R$, activation functions $\sigma()$ and $g()$, just need to derive gradients

# Backpropagation illustration with sigmoid activation function

- Here is a diagram

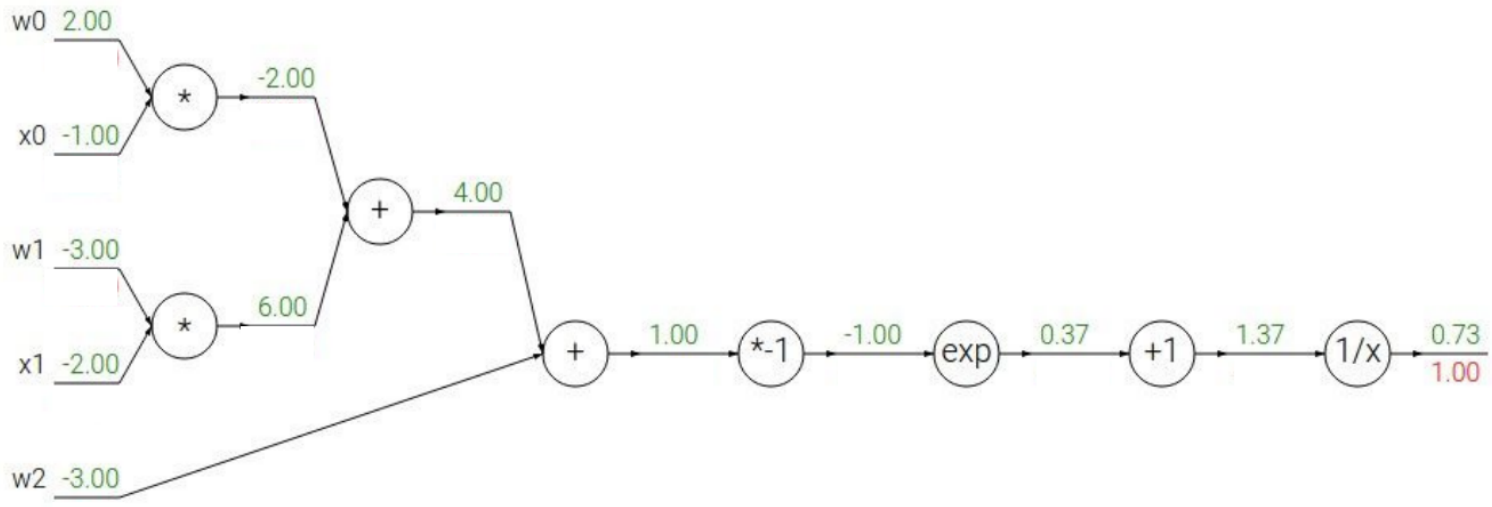


# The structure with squared-error loss

- For input $\boldsymbol{x}$, we find weights $(w_0, \boldsymbol{w})$ such that (forward)
  - $z = \boldsymbol{w}^t \boldsymbol{x} + w_0$
  - $y = \sigma(z) = \frac{1}{1+e^{-z}}$
  - $\mathcal{L} = \frac{1}{2}(y-t)^2 = \frac{1}{2}(\sigma(\boldsymbol{w}^t \boldsymbol{x} + w_0) - t)^2$
- Gradient calculations
  - $\frac{\partial \mathcal{L}}{\partial w_i} = \frac{\partial \mathcal{L}}{\partial \sigma}\frac{\partial \sigma}{\partial w_i} = (\sigma(\boldsymbol{w}^t \boldsymbol{x} + w_0) - t)\sigma'(\boldsymbol{w}^t \boldsymbol{x} + w_0)x_i$
  - $\frac{\partial \mathcal{L}}{\partial w_0} = \frac{\partial \mathcal{L}}{\partial \sigma}\frac{\partial \sigma}{\partial w_0} = (\sigma(\boldsymbol{w}^t \boldsymbol{x} + w_0) - t)\sigma'(\boldsymbol{w}^t \boldsymbol{x} + w_0)$
  - $\nabla \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$
  - We actually can separate the derivatives as follows: $\mathcal{L}'(y) = (y-t)$, $y'(\sigma) = 1$, $\sigma'(z) = \sigma(z)(1-\sigma(z))$, $\frac{\partial z}{\partial w_i} = x_i$, and $\frac{\partial z}{\partial w_0} = 1$

# Backpropagation illustration: a simple example

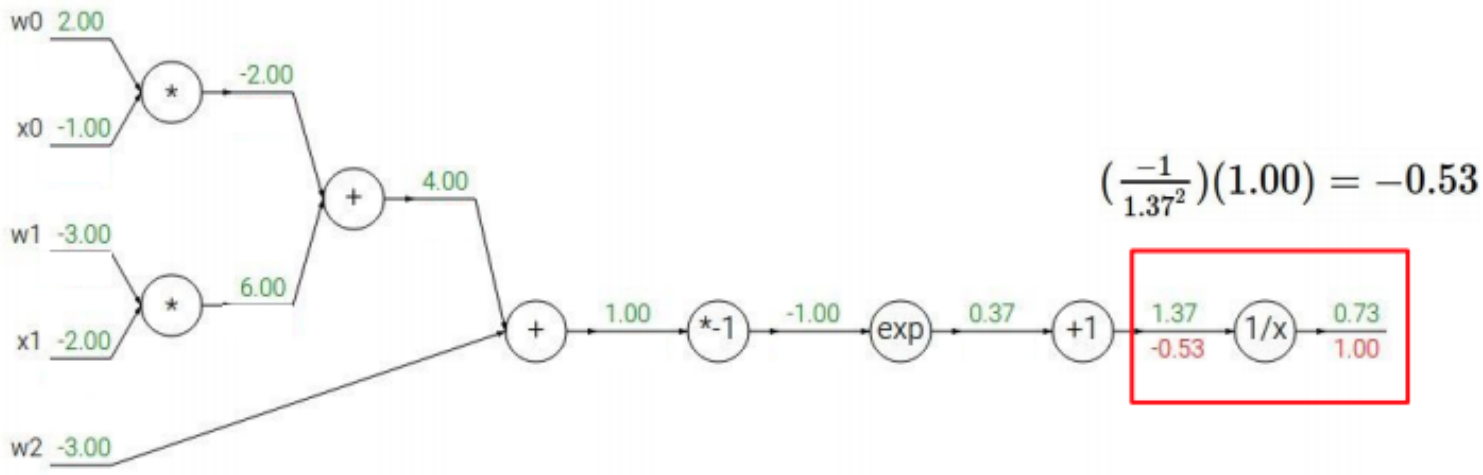
- We separate the form $f(\boldsymbol{w}, \boldsymbol{x}) = \frac{1}{1+e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$




- The green values are the inputs, weights, and forward computed numbers
- The red values would be derivatives
  - For the first one, we have $\frac{df}{df} = 1$

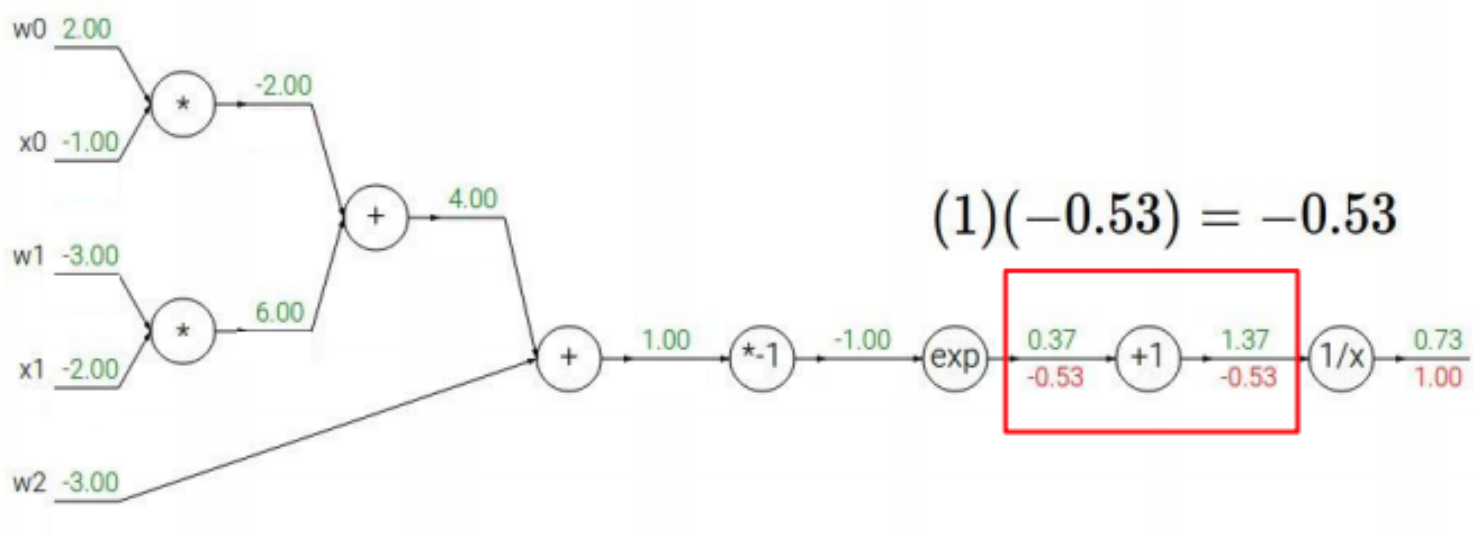
# Example (continued)

- Now going backwards:
  - $f(x) = 1/x$ and $f'(x) = -1/x^2$



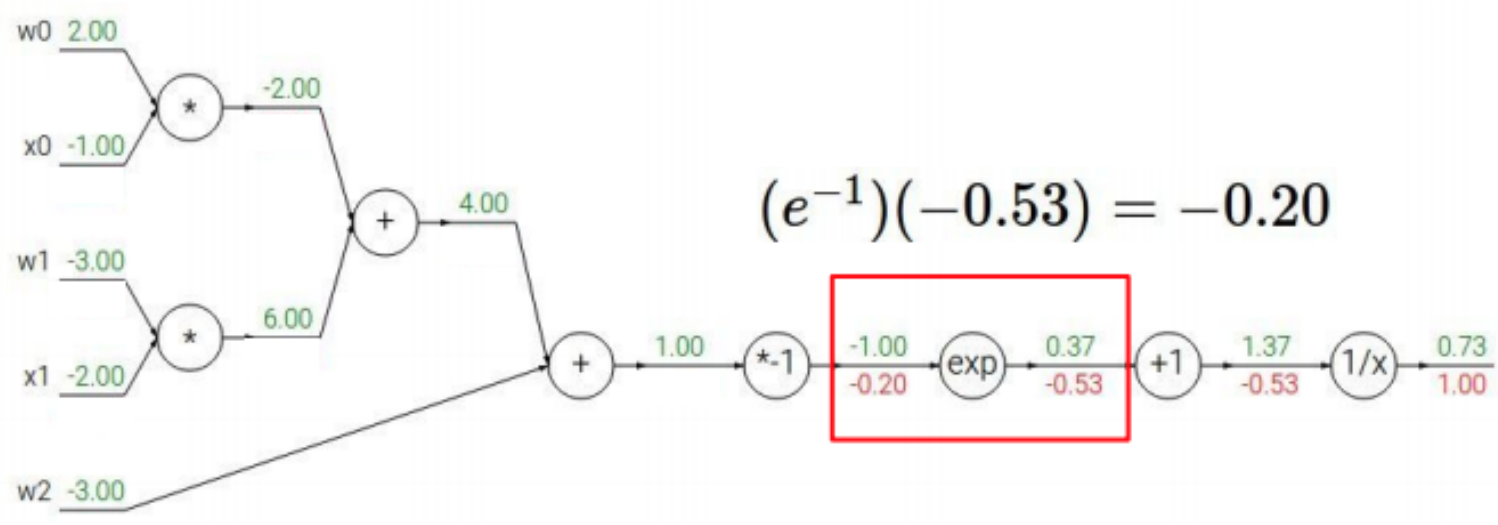
# Example (continued)

- Next one
  - $f(x) = x$ and $f'(x) = 1$



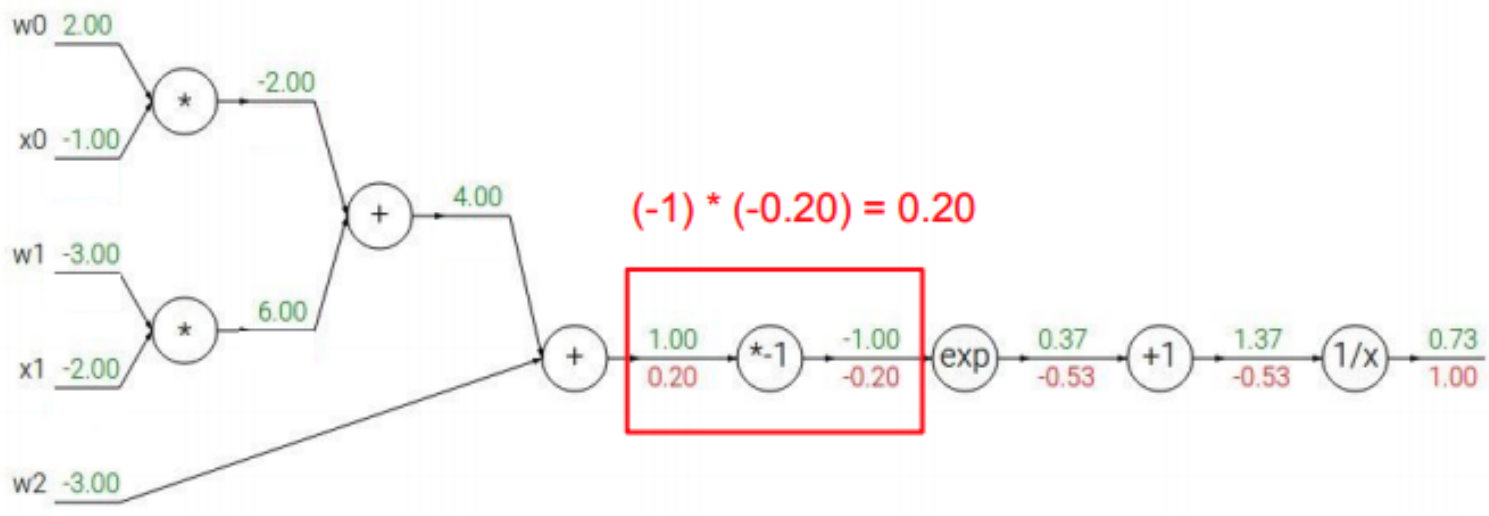
# Example (continued)

- Next one
  - $f(x) = e^x$ and $f'(x) = e^x$



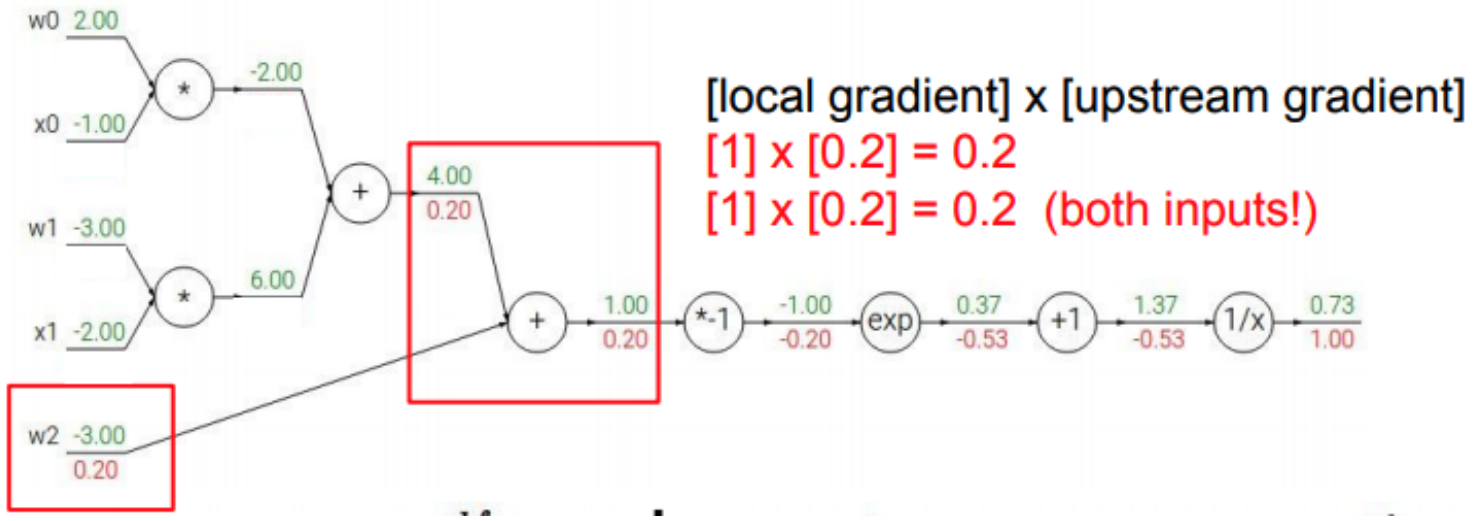
# Example (continued)

- Next one
  - $f(x) = ax$ and $f'(x) = a$



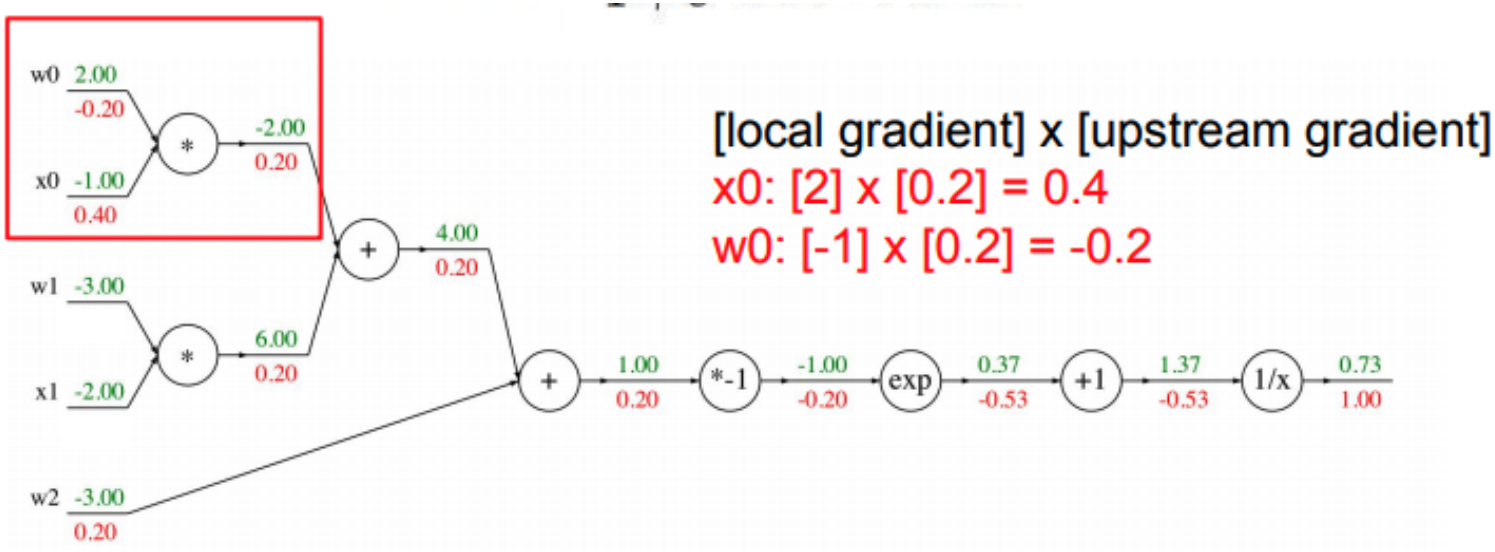
# Example (continued)

- Next one has two nodes, both with simple derivatives
  - $f(x) = ax$ and $f'(x) = a$



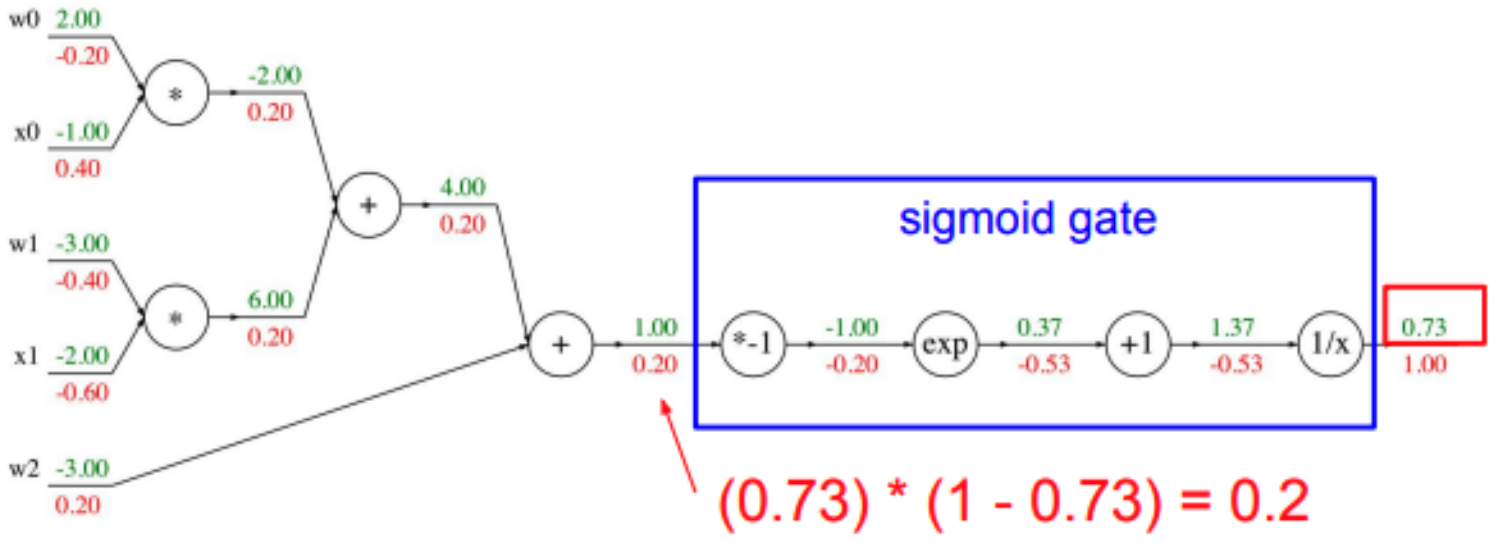
# Example (continued)

- Another one



# Example (continued)

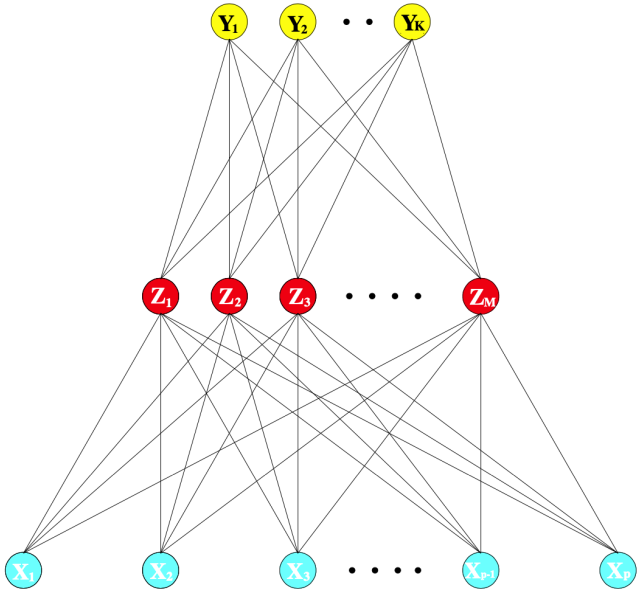
- Now take a look at the sigmoid function as a whole, called sigmoid gate
  - $\sigma(z) = \frac{1}{1+e^{-z}}$ which has $\sigma'(z) = \sigma(z)(1 - \sigma(z))$



# Key idea behind backpropagation

- We do not have targets for a hidden unit, but we can compute how fast the error changes as we change its activity
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined
  - We can compute error derivatives for all the hidden units efficiently
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit
- This is just the chain rule!

# Computing gradients: single layer network




$$Z_m = \sigma(\alpha_{0m} + \boldsymbol{\alpha}_m^T \boldsymbol{X}), m = 1, \ldots, M$$
$$T_k = \beta_{0k} + \boldsymbol{\beta}_k^T \boldsymbol{Z}, k = 1, \ldots, K$$
$$f_k(\boldsymbol{X}) = g_k(\boldsymbol{T}), k = 1, \ldots, K$$

- Error gradients for single layer network:

$$\frac{\partial R}{\partial \alpha_{im}} = \frac{\partial R}{\partial g_k} \frac{\partial g_k}{\partial z_k} \frac{\partial z_k}{\partial \alpha_{ik}}$$

- Error gradient is computable for any continuous activation function $\sigma()$, and any continuous error function

# Some issues in training neural networks

Starting values:

- Note that if the weights are near zero, then the operative part of the sigmoid is roughly linear
  - Hence the neural network collapses into an approximately linear model
- Often starting values for weights are chosen to be random values near zero
  - The model usually starts out nearly linear, and becomes nonlinear as the weights increase
- However, use of exact zero weights leads to zero derivatives and perfect symmetry, and the algorithm never moves
- On the other hand, starting with large weights often leads to poor solutions

# Some issues in training neural networks (continued)

Overfitting

- The training data contains information about the regularities in the mapping from input to output. But it also contains *noise*
  - The target values may be unreliable
  - There is sampling error: There will be accidental regularities just because of the particular training cases that were chosen
- When we fit the model, it cannot tell which regularities are real and which are caused by sampling error
  - So it fits both kinds of regularity
  - If the model is very flexible it can model the sampling error really well. This could be a disaster

# Preventing overfitting

- Use a model that has the right capacity:
  - enough to model the true regularities
  - not enough to also model the spurious regularities (assuming they are weaker)
- Standard ways to limit the capacity of a neural net:
  - Limit the number of hidden units
  - Limit the norm of the weights
  - Stop the learning before it has time to overfit

# Limiting the size of the weights

- *Weight-decay* involves adding an extra term to the cost function that penalizes the squared weights
  $R(\boldsymbol{\theta})^* = R(\boldsymbol{\theta}) + \frac{\lambda}{2} J(\boldsymbol{\theta}) = R(\boldsymbol{\theta}) + \frac{\lambda}{2} \sum_j \theta_j^2$
- Keeps weights small unless they have big error derivatives
  $\frac{\partial R^*}{\partial \theta_j} = \frac{\partial R}{\partial \theta_j} + \lambda \theta_j$
  - When $\frac{\partial R^*}{\partial \theta_j} = 0$ (extremum), $\theta_j = -\frac{1}{\lambda} \frac{\partial R}{\partial \theta_j}$
- The Effect of Weight-decay
  - It prevents the network from using weights that it does not need
  - This can often improve generalization a lot
  - It helps to stop it from fitting the sampling error
  - It makes a smoother model in which the output changes more slowly as the input changes

# Scaling of the Inputs

- The scaling of the inputs determines the effective scaling of the weights in the bottom layer
  - It can have a large effect on the quality of the final solution
- At the outset it is best to standardize all inputs to have mean zero and standard deviation one
  - This ensures all inputs are treated equally in the regularization process, and allows one to choose a meaningful range for the random starting weights
  - With standardized inputs, it is typical to take random uniform weights over the range $[-0.7, +0.7]$

# Number of hidden units and layers

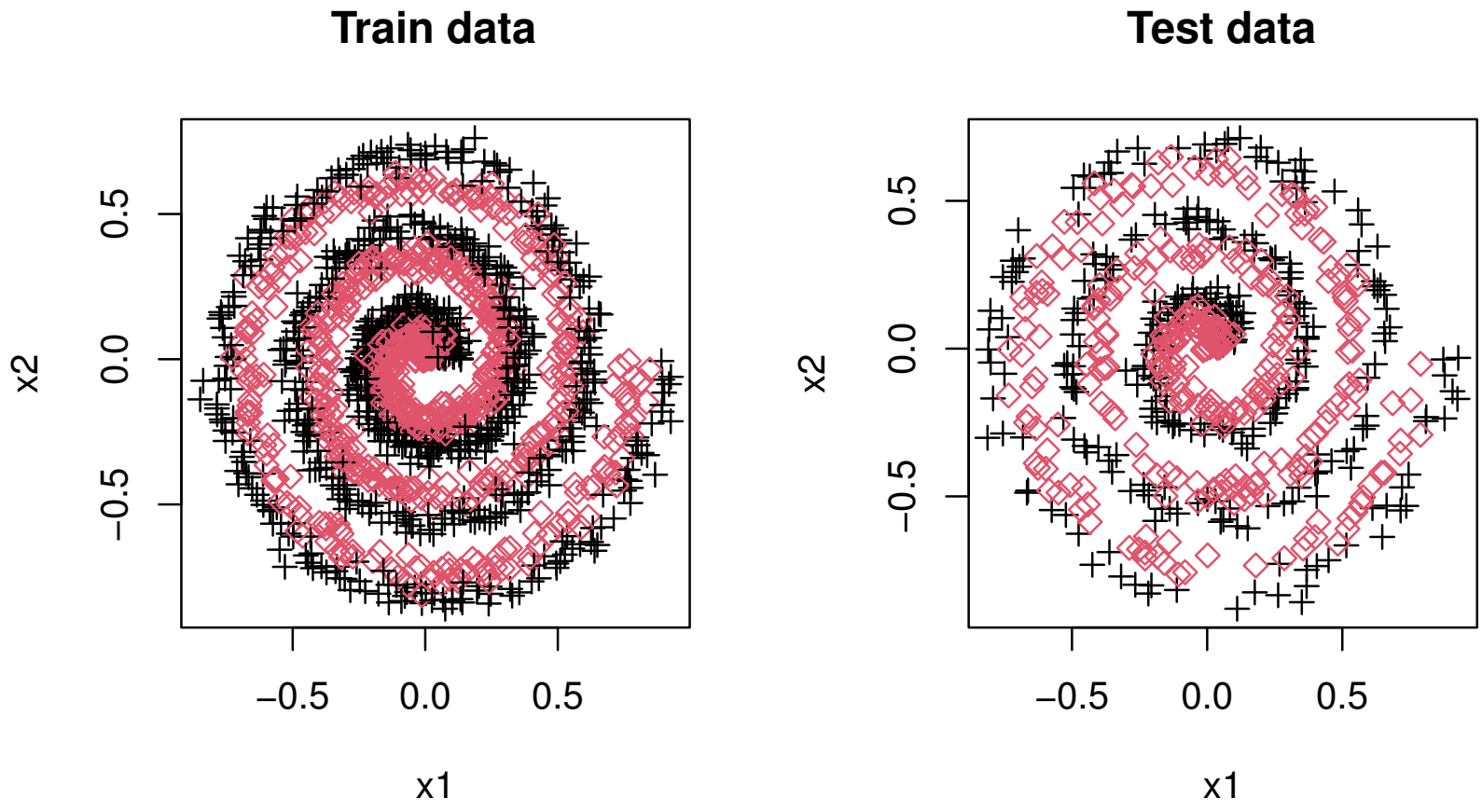
- In general, it is better to have too many hidden units than too few
  - With too few hidden units, the model might not have enough flexibility to capture the nonlinearities in the data
  - With too many hidden units, the extra weights can be shrunk toward zero if appropriate regularization is used
- Typically the number of hidden units is somewhere in the range of 5 to 100, with the number increasing with the number of inputs and number of training cases
  - It is most common to put down a reasonably large number of units and train them with regularization
  - It seems unnecessary to use cross-validation to estimate the optimal number of units if cross-validation is used to estimate the regularization parameter
  - Use of multiple hidden layers allows construction of hierarchical features at different levels of resolution

# Multiple minima

- The error function $R(\theta)$ is nonconvex, possessing many local minima
  - The final solution obtained is quite dependent on the choice of starting weights
  - One must at least try a number of random starting configurations, and choose the solution giving lowest (penalized) error
- A possible better approach is to use the average predictions over the collection of networks as the final prediction
  - This is preferable to averaging the weights, since the nonlinearity of the model implies that this averaged solution could be quite poor
- Another approach is via bagging, which averages the predictions of networks training from randomly perturbed versions of the training data

# Another simulated data

- 2000 observation, with 70% in training data



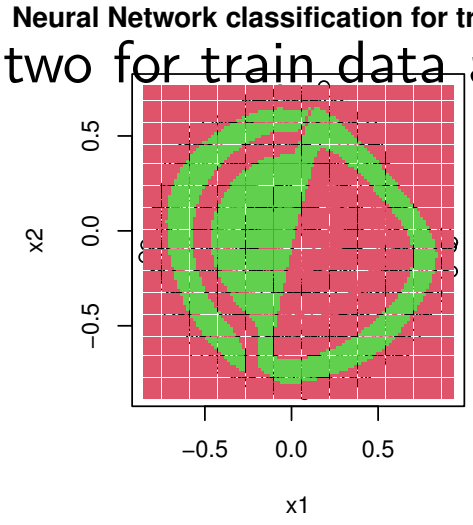
# R code for the previous plot

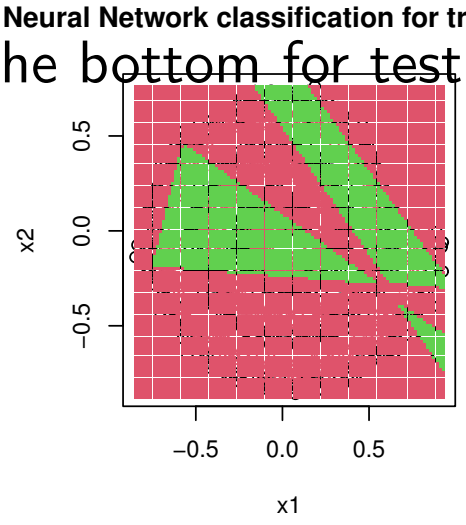
- Data is available on the blackboard

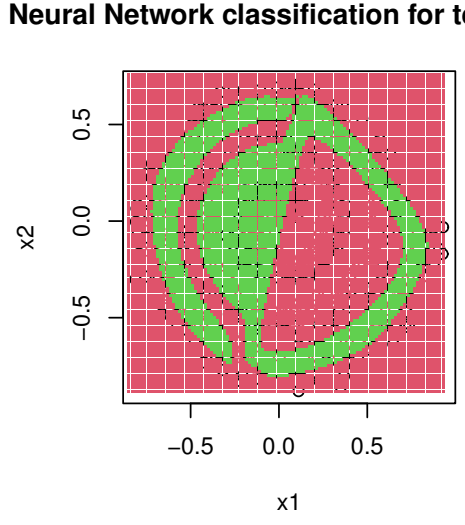
```
## Another simulated data
sim.nn.list = readRDS("C://Users/likeb/Desktop/StatisticalLearning/ClassSlides/Data/simulation.neuralnet.rds")
par(mfrow = c(1,2))
df.tr = sim.nn.list[["df.tr"]]
df.te = sim.nn.list[["df.te"]]
df = sim.nn.list[["df"]]
tr.ind = sim.nn.list[["tr.ind"]]
sim.nn1 = sim.nn.list[["sim.nn1"]]
sim.nn2 = sim.nn.list[["sim.nn2"]]
sim.nn3 = sim.nn.list[["sim.nn3"]]
plot(df.tr$x1,df.tr$x2, col = df.tr$y+1,pch = 2*df.tr$y+3,xlab = "x1",ylab = "x2",main = "Train data")
plot(df.te$x1,df.te$x2, col = df.te$y+1,pch = 2*df.te$y+3,xlab = "x1",ylab = "x2",main = "Test data")
```

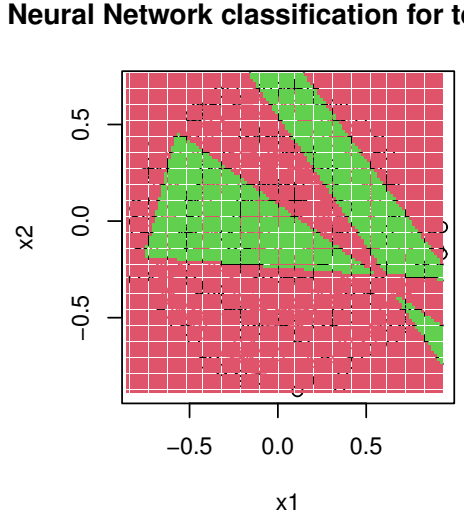
# Two neural network model results

- The top two for train data and the bottom for test data



# R code for the previous plot

- Data is available on the blackboard

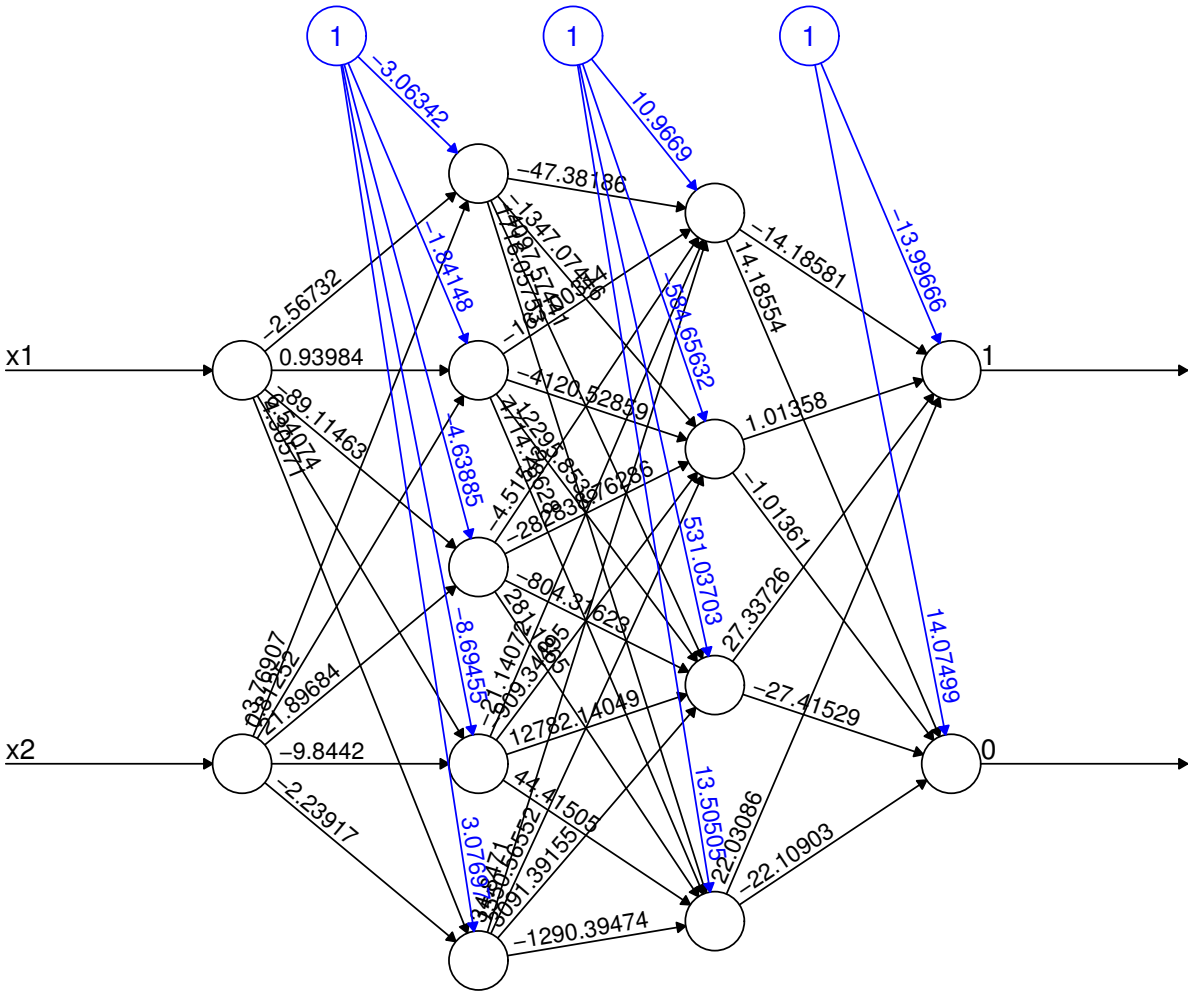
```
## Two neural network model results
par(mfrow = c(2,2))
class.plot(sim.nn1,df,train.index = tr.ind,method = "nn")
class.plot(sim.nn2,df,train.index = tr.ind,method = "nn")
class.plot(sim.nn1,df,train.index = tr.ind,method = "nn",train = F)
class.plot(sim.nn2,df,train.index = tr.ind,method = "nn",train = F)
```

# The models and errors

```
> ## The models and Errors
>
> sim.nn1$call
neuralnet(formula = as.factor(y) ~ ., data = df.tr, hidden = c(5,
    4), threshold = 0.1, stepmax = 1e+07, err.fct = "ce",
    linear.output = F)
> sim.nn2$call
neuralnet(formula = as.factor(y) ~ ., data = df.tr, hidden = 5,
    threshold = 0.05, stepmax = 1e+06, err.fct = "ce",
    linear.output = F)
> tr1.pred = apply(predict(sim.nn1,df.tr),1,which.max)-1
> tr2.pred = apply(predict(sim.nn2,df.tr),1,which.max)-1
> te1.pred = apply(predict(sim.nn1,df.te),1,which.max)-1
> te2.pred = apply(predict(sim.nn2,df.te),1,which.max)-1
> print(rbind(c("train err for nn1","test err for nn1"),c(round(mean(tr1.pred != df.tr$y),2),round(mean(te1.pred != df.te$y),2))))
     [,1]              [,2]
[1,] "train err for nn1" "test err for nn1"
[2,] "0.28"            "0.32"
> print(rbind(c("train err for nn2","test err for nn2"),c(round(mean(tr2.pred != df.tr$y),2),round(mean(te2.pred != df.te$y),2))))
     [,1]              [,2]
[1,] "train err for nn2" "test err for nn2"
[2,] "0.39"            "0.4"
> print(rbind(c("time spent to calculate nn1 =",sim.nn.list[["time.nn1"]]),
+   c("time spent to calculate nn2 =",sim.nn.list[["time.nn2"]])))
     [,1]                          [,2]
[1,] "time spent to calculate nn1 =" "49.12783m"
[2,] "time spent to calculate nn2 =" "7.87s"
>
```

# Plot for network 1

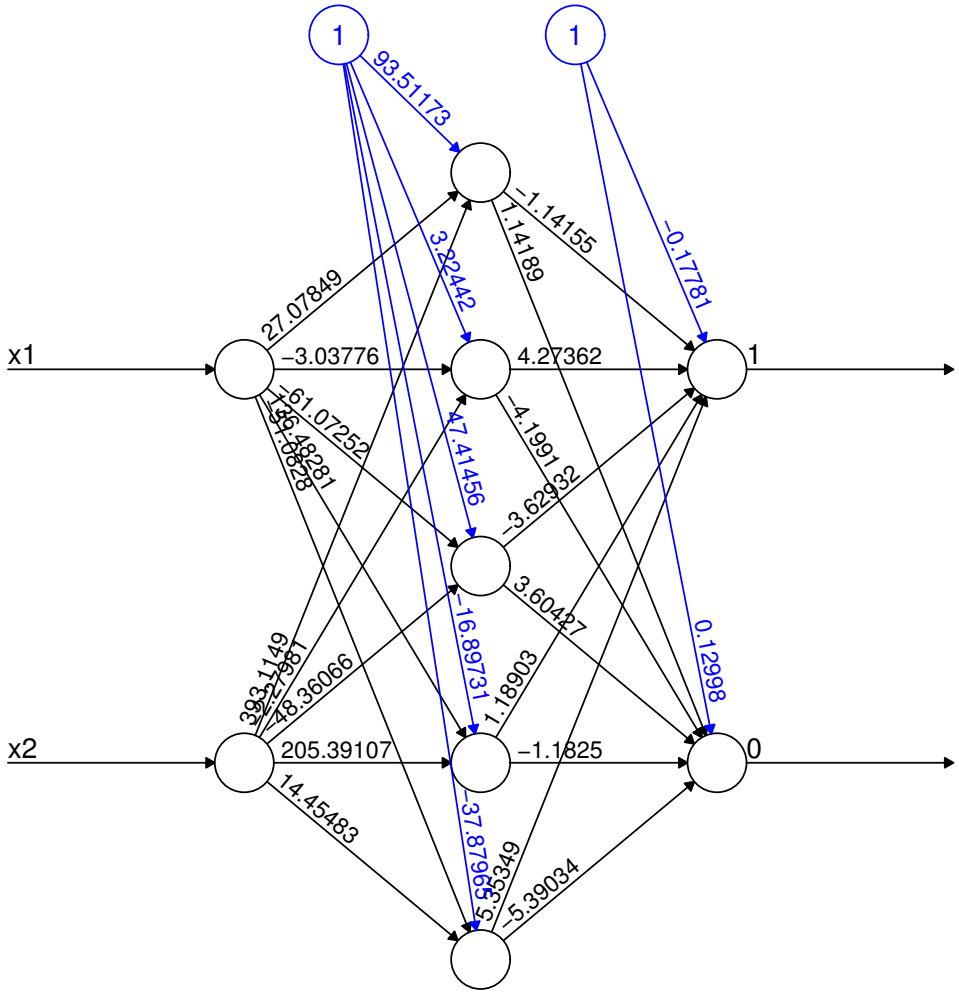
- plot(sim.nn1,rep = "best")




Error: 1372.775734 Steps: 2829022

# Plot for network 2

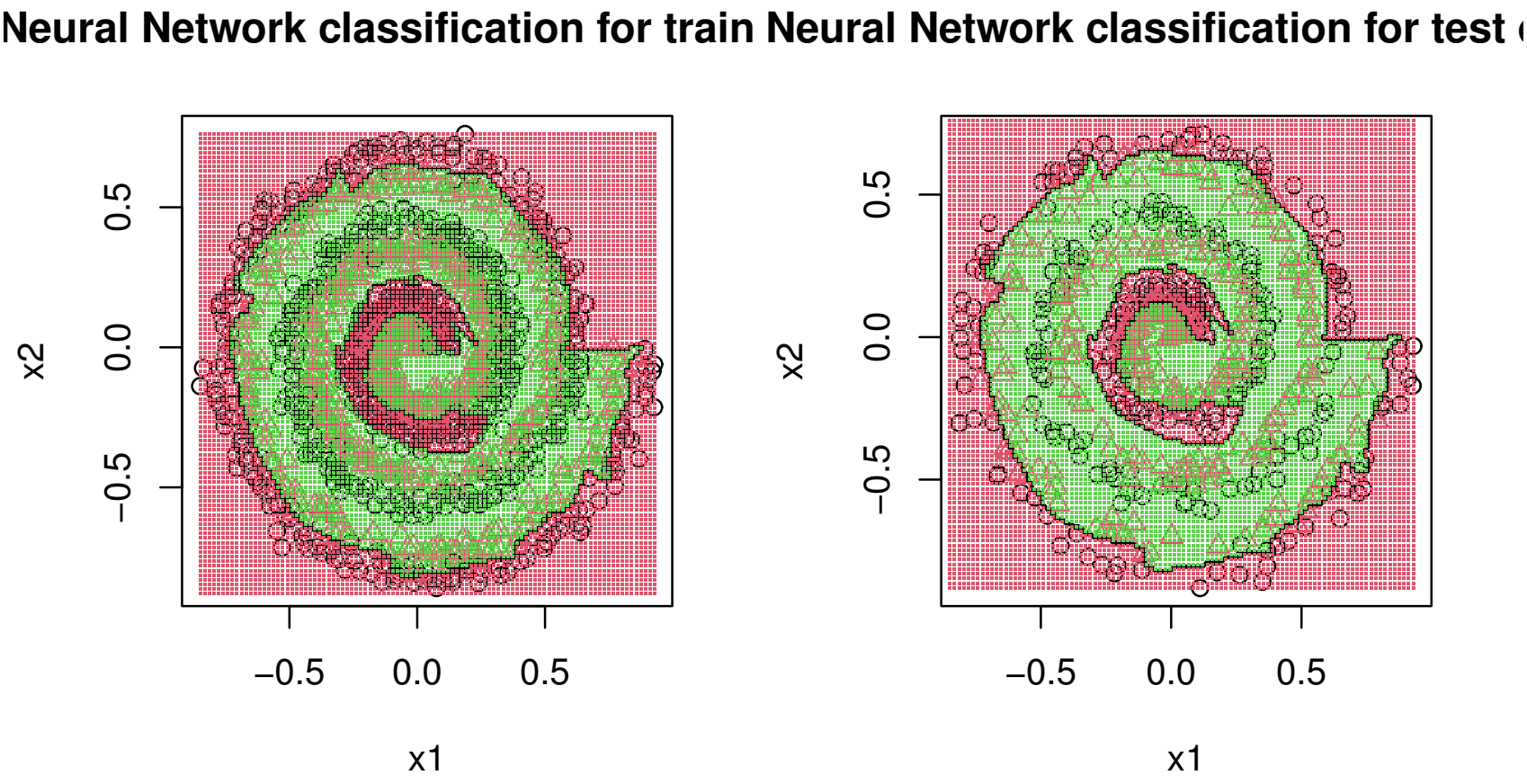
- plot(sim.nn2,rep = "best")




Error: 1799.857776  Steps: 11031

# A deeper learning model

- We consider a deeper learning model

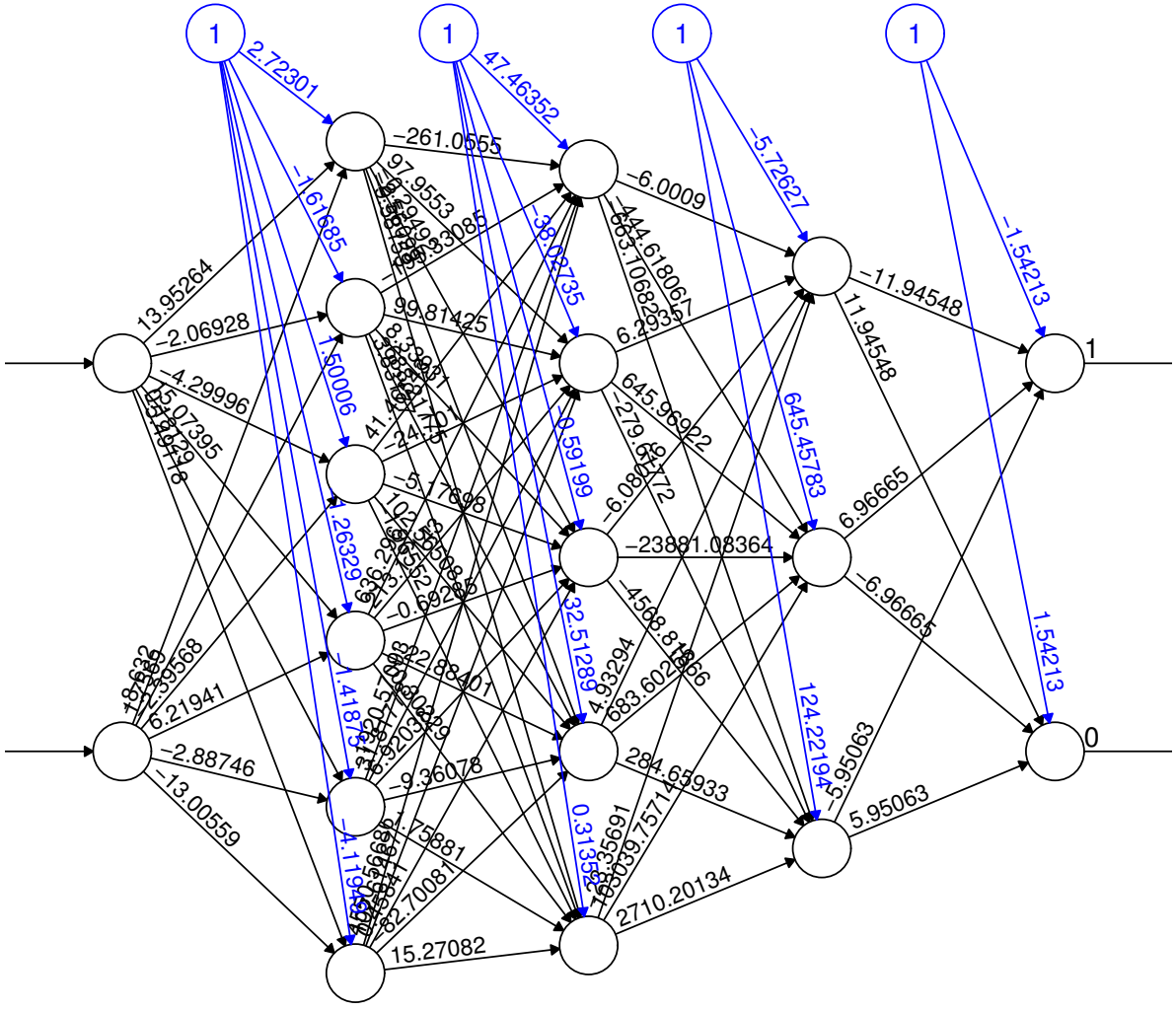


# R code for the previous plot

```
> ## A deeper learning model
> par(mfrow = c(1,2))
> class.plot(sim.nn3,df,train.index = tr.ind,method = "nn")
> class.plot(sim.nn3,df,train.index = tr.ind,method = "nn",train = F)
> sim.nn3$call
neuralnet(formula = as.factor(y) ~ ., data = df.tr, hidden = c(6,
    5, 3), threshold = 0.2, stepmax = 1e+07, err.fct = "ce",
    linear.output = F)
> tr3.pred = apply(predict(sim.nn3,df.tr),1,which.max)-1
> te3.pred = apply(predict(sim.nn3,df.te),1,which.max)-1
> print(rbind(c("train err for nn3","test err for nn3"),c(round(mean(tr3.pred != df.tr$y),2),round(mean(te3.pred != df.te$y),2))))
     [,1]              [,2]
[1,] "train err for nn3" "test err for nn3"
[2,] "0.23"              "0.27"
> print(c("time spent to calculate nn3 =",sim.nn.list[["time.nn3"]]))
[1] "time spent to calculate nn3 =" "1.384262h"
>
```

# Plot for network 3

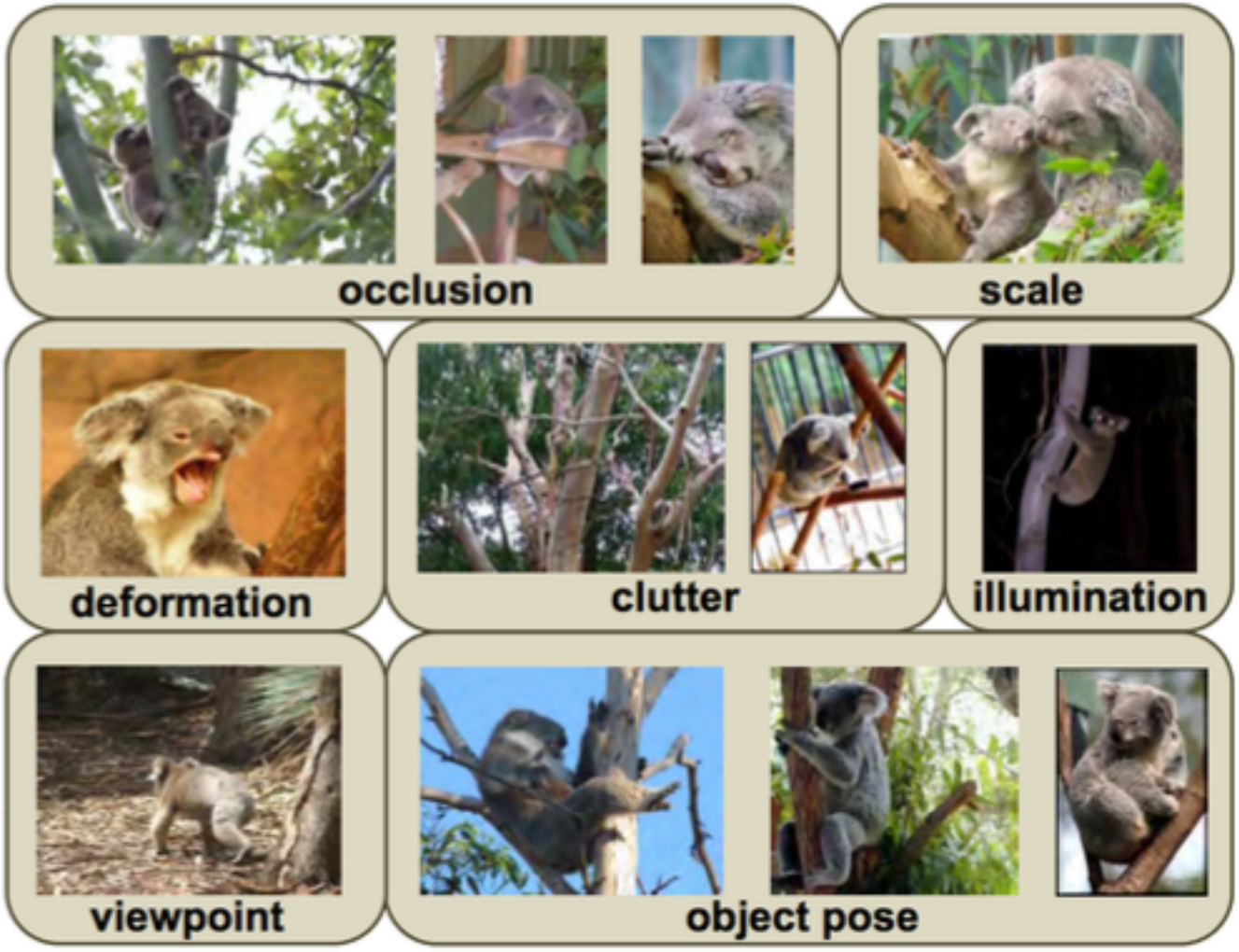
- plot(sim.nn3,rep = "best")




Error: 1167.391165 Steps: 3329984

# Deep learning for object recognition

- Problems in object recognition: difficult scene conditions



# Problems in object recognition: huge within-class variations

- Recognition is mainly about modeling variation






# Problems in object recognition

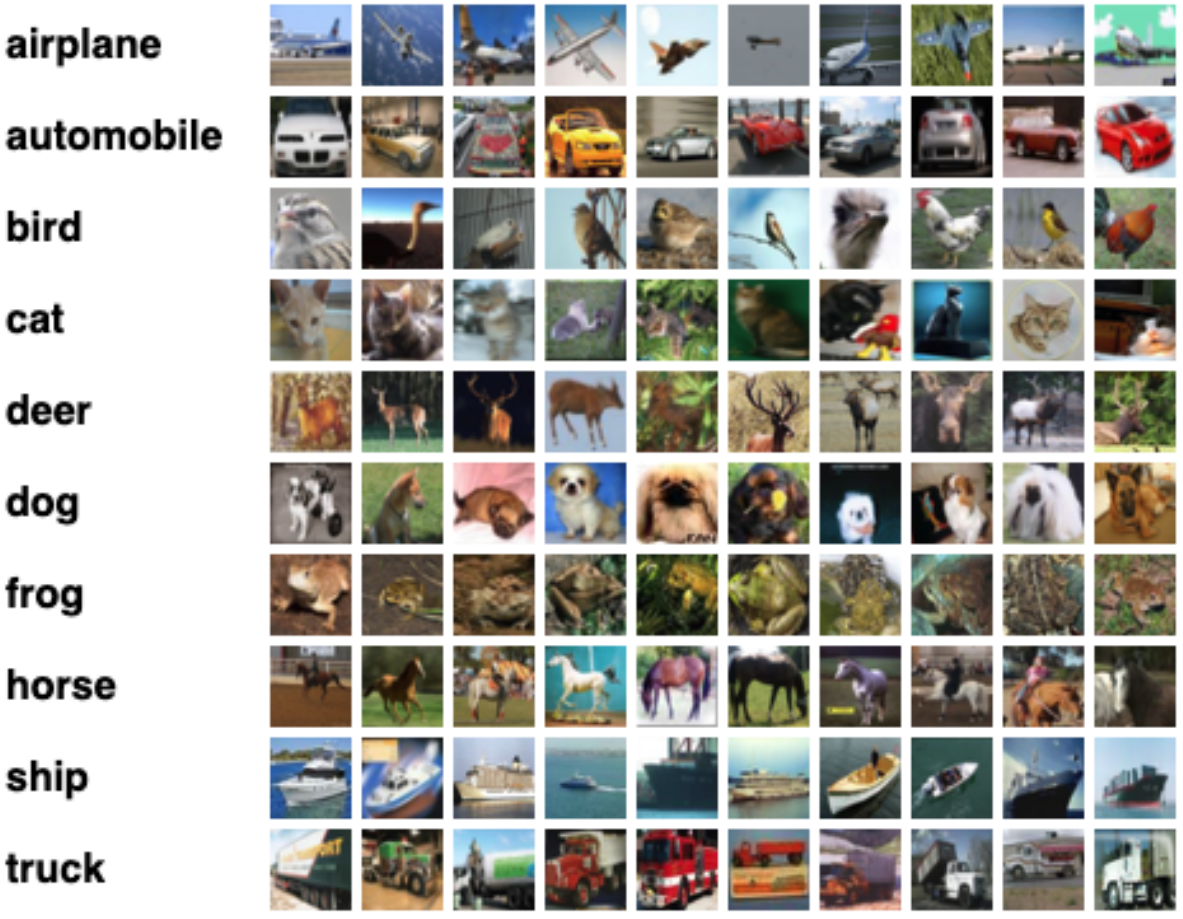
- Tones of classes

# Neural nets for object recognition

- People are very good at recognizing shapes
  - Intrinsically difficult, computers are bad at it
- Why is it difficult?
  - Segmentation: Real scenes are cluttered
  - Invariance: We are very good at ignoring all sorts of variations that do not affect shape
  - Deformations: Natural shape classes allow variations (faces, letters, chairs)
  - A huge amount of computation is required

# The CIFAR-10 dataset

- CIFAR (Canadian Institute For Advanced Research)
  - 10: ten classes
  - Images are only of size 32x32x3 (32 wide, 32 high, 3 color channels)



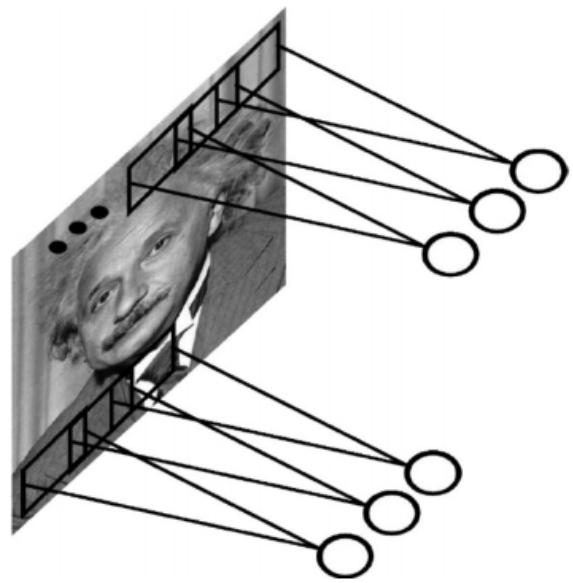
# How to deal with large input spaces

- How can we apply neural nets to images?
  - Images can have millions of pixels, i.e., $\boldsymbol{x}$ is very high dimensional
  - How many parameters do I have?
    - In CIFAR-10, there would be $32 \times 32 \times 3 = 3072$ weights with 1 node
    - But for more high-resolution images?
    - $200 \times 200 \times 3 = 120,000$; can quickly increase computing time
  - Prohibitive to have fully-connected layers
- What can we do?
  - Use a locally connected layer

# Basic idea

- Idea: statistics are similar at different locations (Lecun 1998)
- Connect each hidden unit to a small input patch and share the weight across space
- This is called a convolution layer

Example:
$200 \times 200$ image
100 filters
Filter size $10 \times 10$
10K parameters



# Architecture of a CNN

- A convolutional neural network consists of an input and an output layer, as well as multiple hidden layers
  - The hidden layers consist of a series of convolutional layers, pooling layers, fully connected layers and normalization layers
  - The layers are called hidden because their inputs and outputs are masked by the activation function and final convolution
- The neural network called convolution is by convention
  - Mathematically, it is technically a sliding dot product or cross-correlation

# Convolution

- If $X$ and $Y$ are independent with densities $f$ and $g$, respectively, the density function of $X + Y$ is

$$f_{X+Y}(t) = \begin{cases} \int_{-\infty}^{\infty} f_X(x) g_Y(t-x) dx, & \text{continuous} \\ \sum_{t=-\infty}^{\infty} f_X(x) g_Y(t-x), & \text{discrete} \end{cases}$$

- In mathematics, a convolution of two discrete functions is

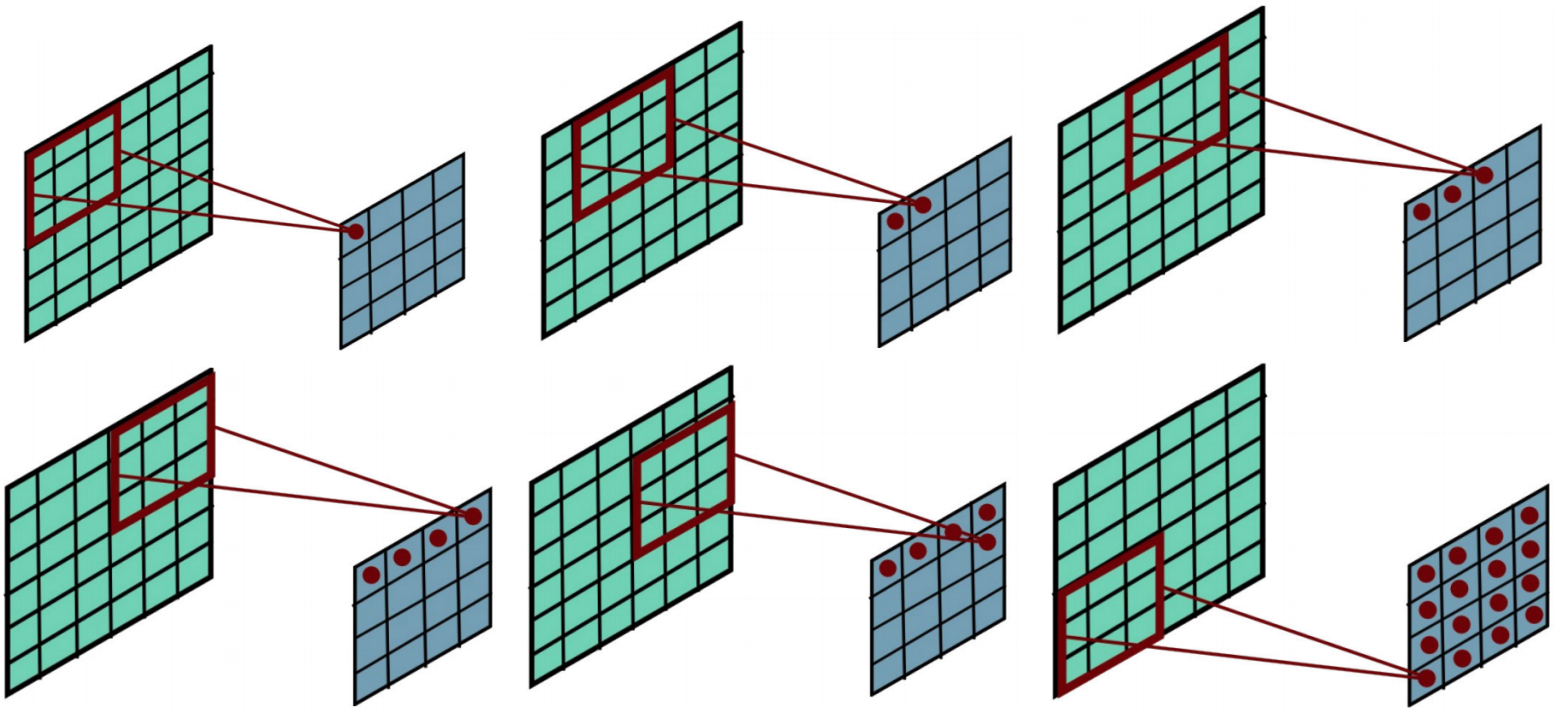
$$(f * g)(t) = \sum_{m=-\infty}^{\infty} f(m) g(t-m)$$

- Convolutional neural networks usually perform 2D convolution on images:

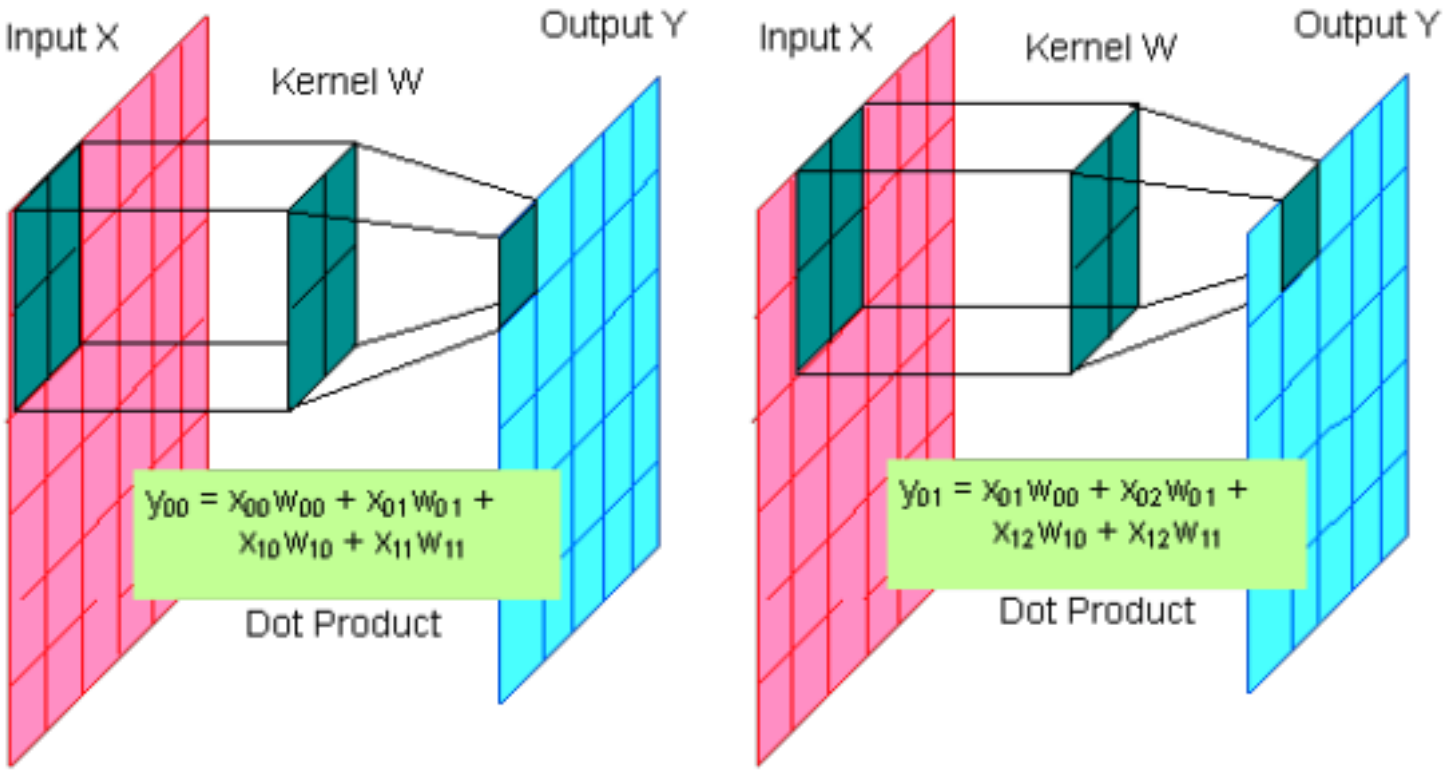
$$(f * g)(x, y) = \sum_{m=-M}^{M} \sum_{n=-N}^{N} f(x-n, y-m) g(n, m)$$

where $g$ is a kernel function

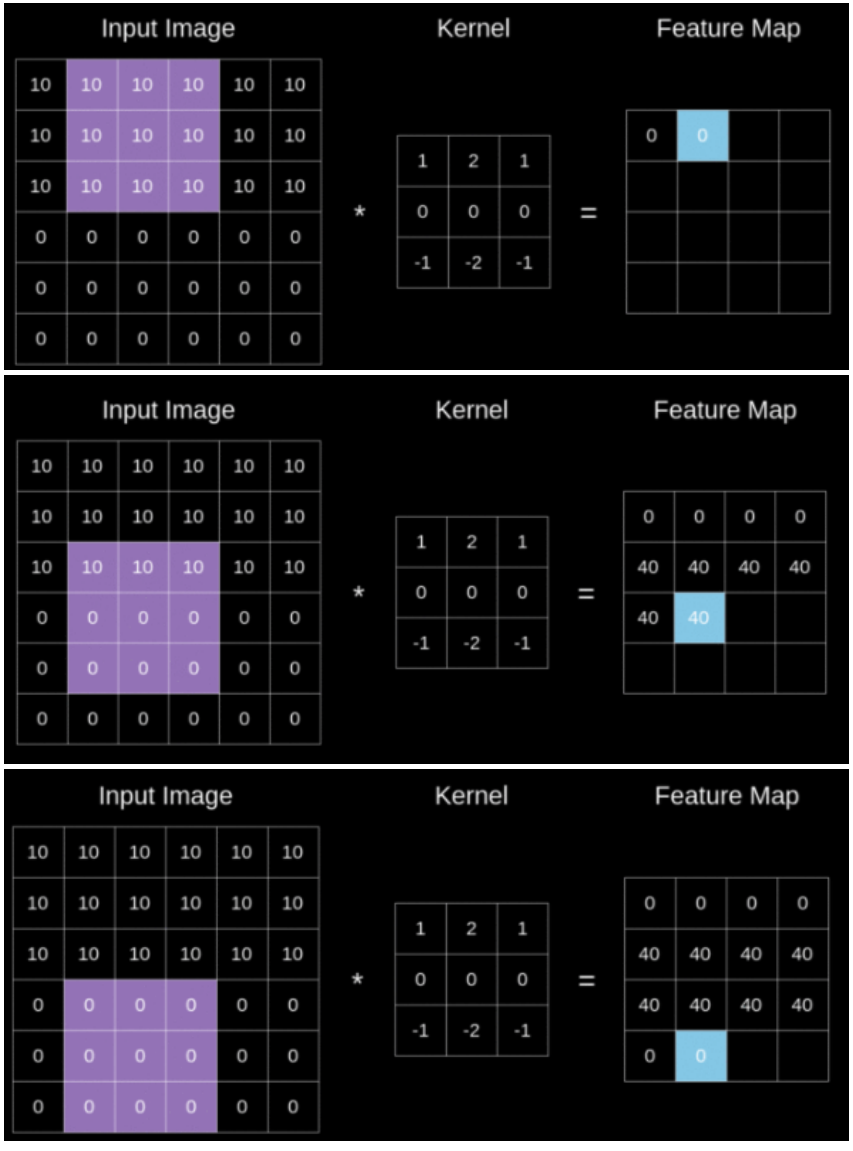
# CNN convolutional layer



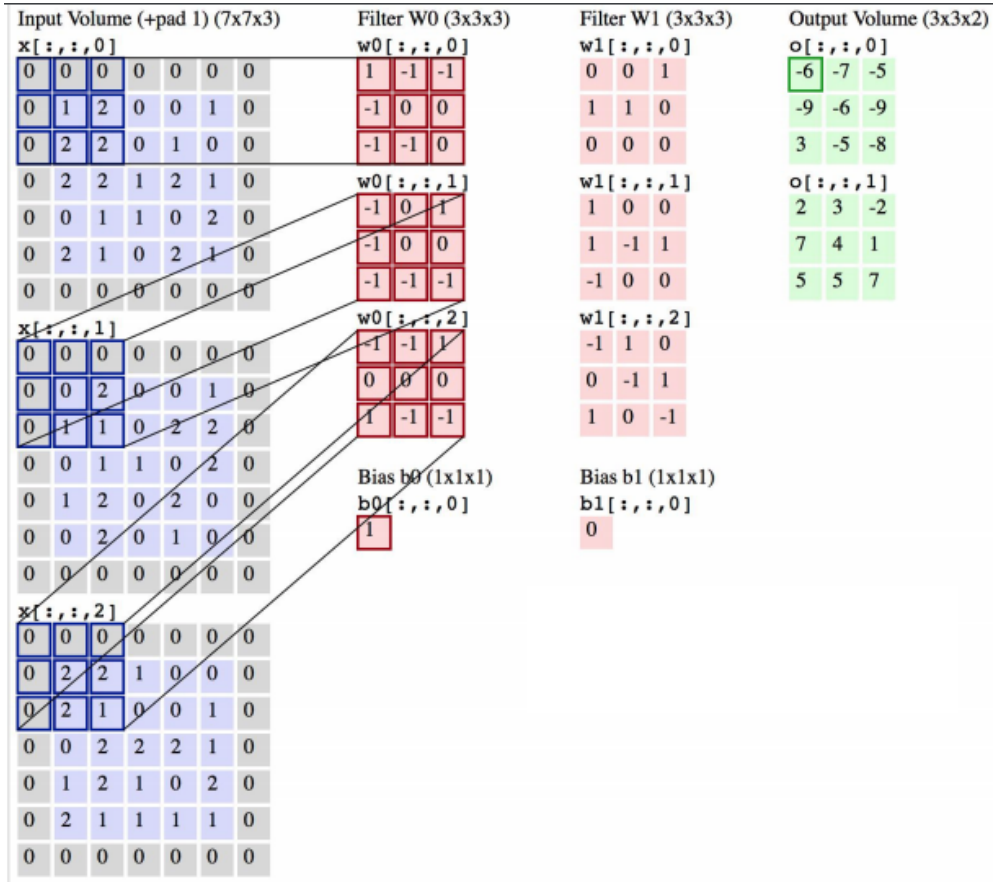
# CNN convolutional layer with kernel



# CNN convolutional layer with kernel (continued)

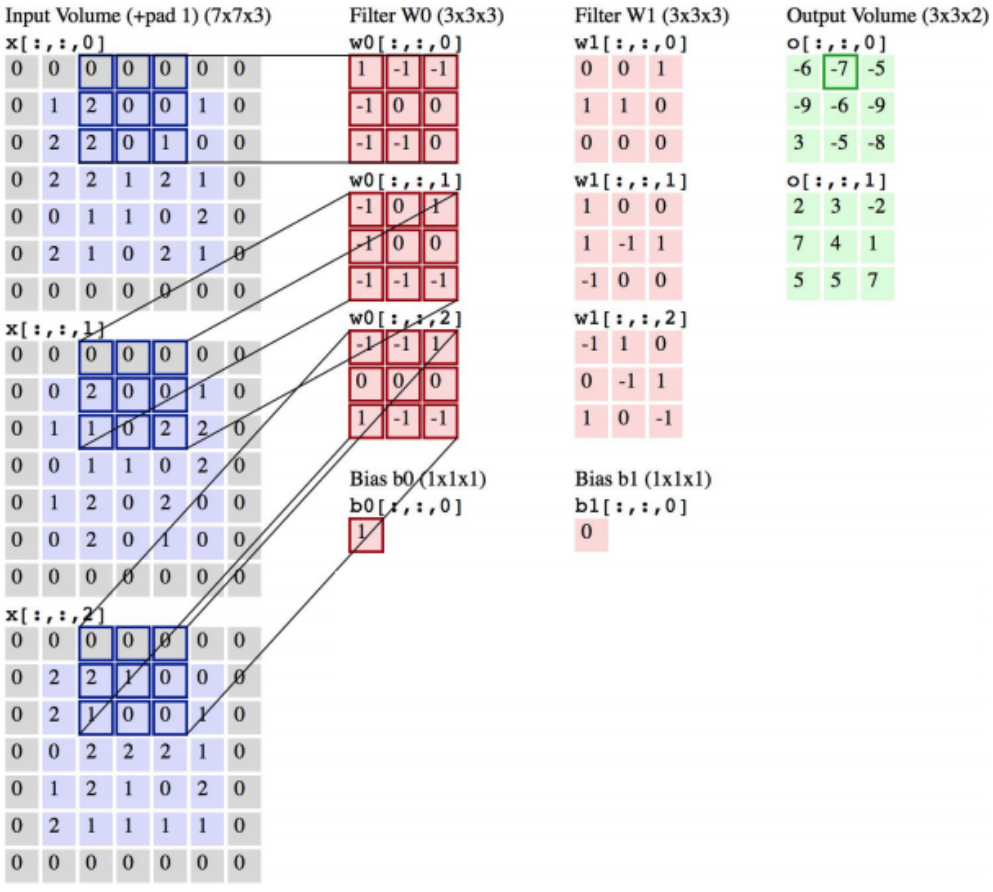


# Convolutional layer structure



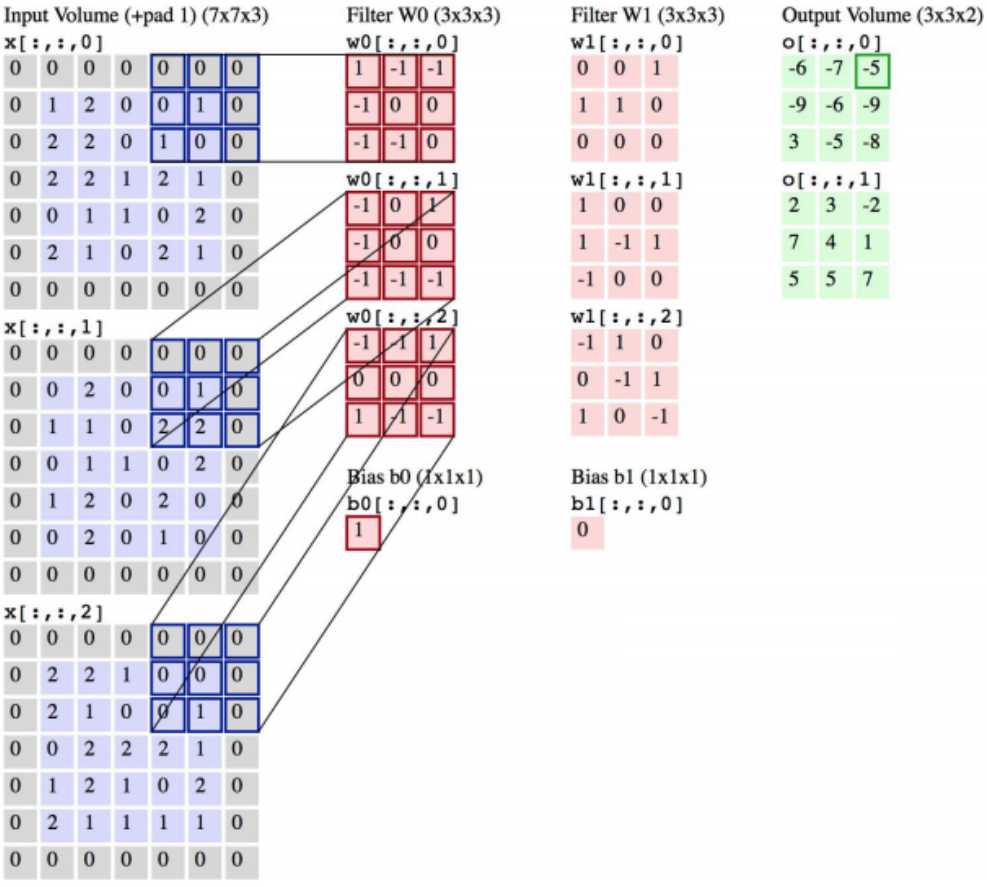

- The process is a 2D convolution on the inputs
- The "dot products" between weights and inputs are "integrated" across "channels"
- Filter weights are shared across receptive fields. The filter has same number of layers as input volume channels, and output volume has same "depth" as the number of filters

# Convolutional layer structure (continued)




- The process is a 2D convolution on the inputs
- The "dot products" between weights and inputs are "integrated" across "channels"
- Filter weights are shared across receptive fields. The filter has same number of layers as input volume channels, and output volume has same "depth" as the number of filters

# Convolutional layer structure

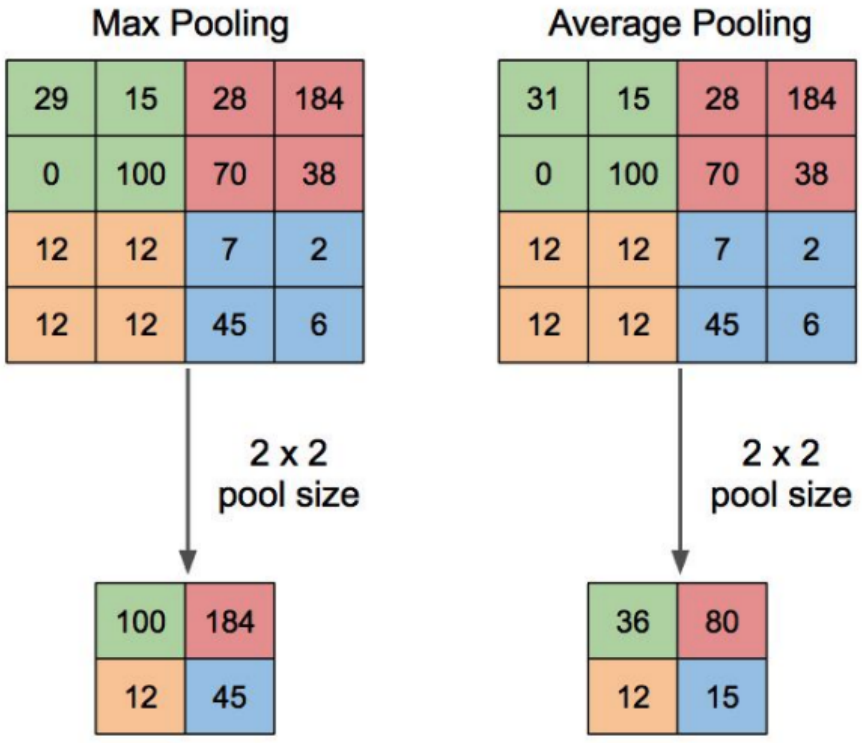



- The process is a 2D convolution on the inputs
- The "dot products" between weights and inputs are "integrated" across "channels"
- Filter weights are shared across receptive fields. The filter has same number of layers as input volume channels, and output volume has same "depth" as the number of filters

# Convolution layer calculations

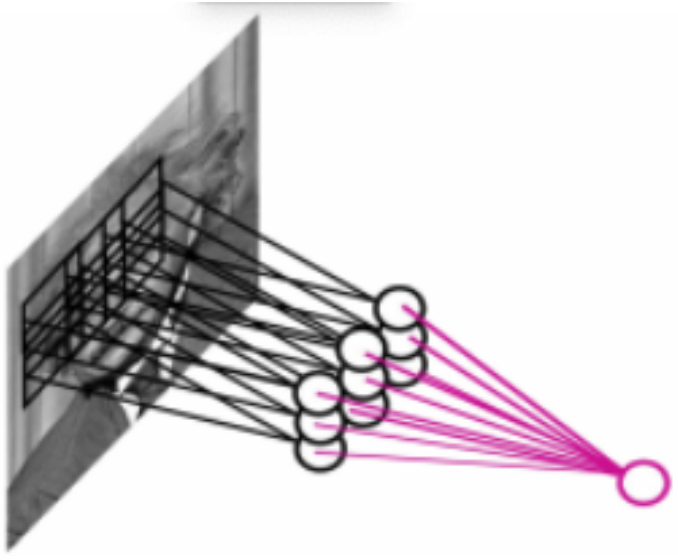
- Input volume of size $W_1 \times H_1 \times D_1$ $(5 \times 5 \times 3)$
- Requires four hyperparameters:
  - Number of filters $K$ $(3)$
  - Spatial extent $F$ $(3)$
  - The Stride $S$ $(1)$
  - The amount of zero padding $P$ $(1)$
- Produces a volume of size $W_2 \times H_2 \times D_2$ output
  - $W_2 = \frac{W_1 - F + 2P}{S} + 1$ $(\frac{5-3+(2)(1)}{2} + 1 = 3)$
  - $H_2 = \frac{H_1 - F + 2P}{S} + 1$ $(\frac{5-3+(2)(1)}{2} + 1 = 3)$
  - $D_2 = K$ $(3)$

# Pooling layer




- Convolutional layers provide activation maps
- Pooling layer applies non-linear downsampling on activation maps
- Pooling is aggressive (discard info); the trend is to use smaller filter size and abandon pooling

# Pooling



- By pooling filter responses at different locations we gain robustness to the exact spatial location of features
- Choose a window size and then return an operator of the input window
- Often use max pooling or average
- Why?
  - non-linearity
  - robustness to position shifts
  - reduce dimensionality

# Pooling layer calculations

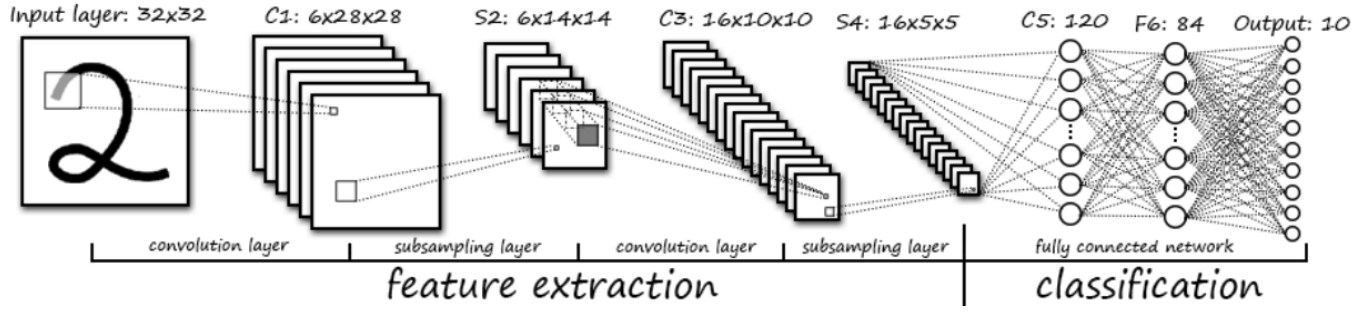
- Input of size $W_1 \times H_1 \times D_1$
- Requires two hyperparameters
  - Spatial extent $F$
  - The stride $S$
- Produces a volume of size $W_2 \times H_2 \times D_2$ output
  - $W_2 = \frac{W_1 - F}{S} + 1$
  - $H_2 = \frac{H_1 - F}{S} + 1$
  - $D_2 = D_1$
- Introduces zero parameters since it computes a fixed function of the input
- Note that it is not common to use zero-padding for pooling layer

# Fully connected layer

- Just as regular neural network
- Can view as the final learning phase, which maps extracted visual features to desired outputs
- Usually adaptive to classification/encoding tasks
- Common output is a vector, which is then passed through softmax to represent confidence of classification
- The outputs can also be used as “bottleneck”

# LeNeT-5: An earlier CNN

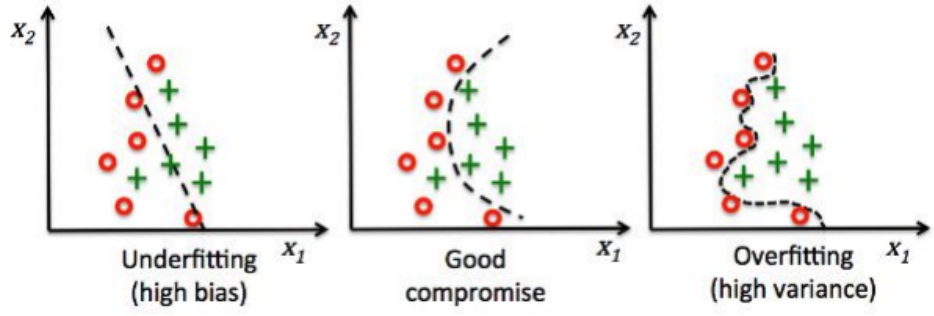
- In 1989, Yann LeCun et al. first applied the backpropagation algorithm to practical applications, and believed that the ability to learn network generalization could be greatly enhanced by providing constraints from the task's domain
- He combined a convolutional neural network trained by backpropagation algorithms to read handwritten numbers and successfully applied it in identifying handwritten zip code numbers provided by the US Postal Service
- This was the prototype of what later came to be called LeNet



# Loss Layer and regularization

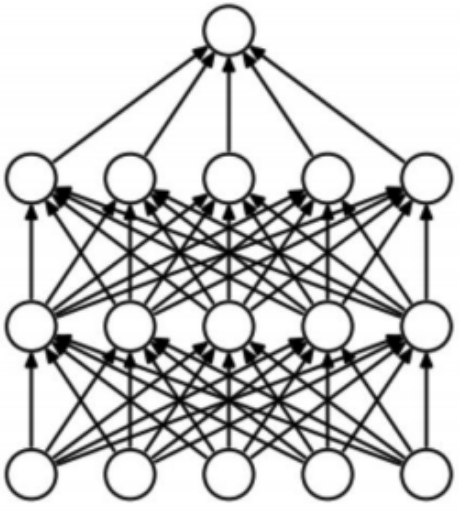
- Loss layer
  - $L_1$, $L_2$ loss
  - Cross-Entropy loss (works well for classification, e.g., image classification)
  - Hinge Loss
  - Huber Loss, more resilient to outliers with smooth gradient
  - Regularization: To prevent overfitting with huge amount of training data

- L1 / L2
- Dropout
- Batch norm
- Gradient clipping
- Max norm constraint

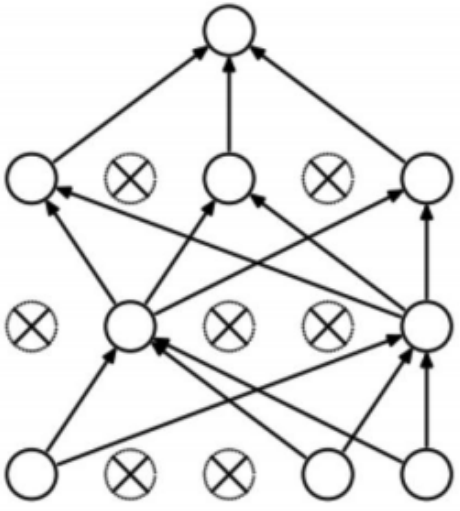


# Dropout

- During training, randomly ignore activations by probability $p$
- During testing, use all activations but scale them by $p$
- Effectively prevent overfitting by reducing correlation between neurons



(a) Standard Neural Net



(b) After applying dropout.

# Batch normalization

- Makes networks robust to bad initialization of weights
- Usually inserted right before activation layers
- Reduce covariance shift by normalizing and scaling inputs
- The scale and shift parameters are trainable to avoid losing stability of the network

**Input:** Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: $\gamma, \beta$
**Output:** $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

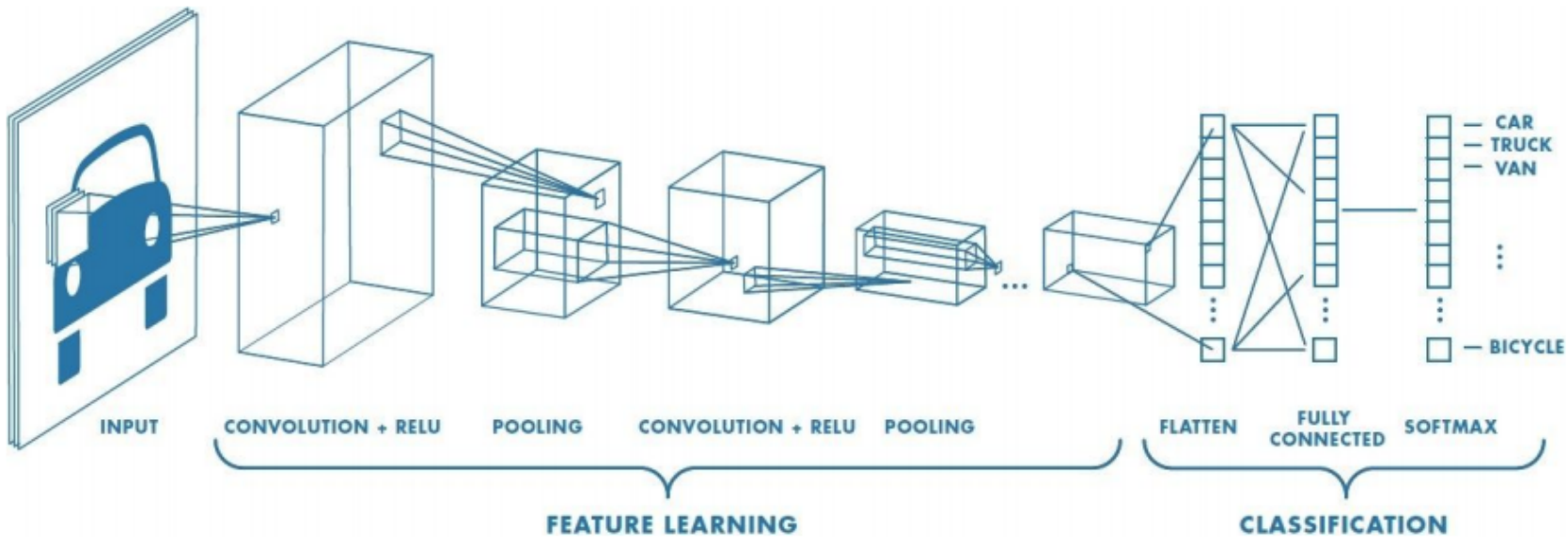
$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

**Algorithm 1:** Batch Normalizing Transform, applied to activation $x$ over a mini-batch.

# An illustration of CNN architecture

- In R, you may use package keras() to run CNN



# R Resources

- Fitting a neural network in R; neuralnet package. Link
- R-Session 11 - Statistical Learning - Neural Networks. Link

# References


- *The Elements of Statistical Learning, 2nd Edition,* by Hastie, T., Tibshirani, R. and Friedman, J. (2009). `http://statweb.stanford.edu/~tibs/ElemStatLearn/`
- Prof. Xiaogang Su's Data Mining and Statistical Learning I. `https://sites.google.com/site/xgsu00/`.
- Prof. Yaser Abu-Mostafa's Lecture. ▸ Link