Neural Networks

Joaquin Vanschoren, Mykola Pechenizkiy, Anne Driemel

March 30, 2016

Course overview: http://openml.github.io/course

```
options(warn=-1)
library(mlr) # ML library
library(OpenML) # Import and share data, flows, results
library(ggplot2) # Plotting
library(cowplot) # Plot styling
library(rattle) # Plotting trees
library(grid) # For linegraphs
library(reshape2) # Reformatting data
```

1.1 Neural Networks

Reverse-engineering the brain

- Very efficient for vision, manipulation, speech understanding, language
- Not so well suited for high-dimensional data, large complex network data

1.1.1 Overview of Algorithms

- Perceptrons
  - Linear classifier based on neuron
- Gradient Descent
  - Essential part of many kinds of learning algorithms
- Feedforward (multilayer) networks
  - Simplest deep learning algorithm
- Backpropagation
  - Most widely used algorithm for training neural nets

1.1.2 Brain vs CPU

- Lots of research into mapping and studying the brain
- $\sim$ 10^10 neurons, switching in $\sim$ .001s (transistor $\sim$ GHz)
- $\sim$ 10^5 connections per neuron (transistor $\sim$ 10)
- Consumes $\sim$ 40W
- Recognizes scene in $\sim$ .1s (massive parallellism!)
- Visual cortex (V1) $\sim$ 140 million neurons
  - Next cortices (V2-V5) do increasingly complex processing

1.1.3 Example: Character recognition

```
mnist_data = getOMLDataSet(did = 554)$data # MNIST dataset from OpenML
par( mfrow = c(10,10), mai = c(0,0,0,0)) # 10x10 grid
for(i in 1:100){ # Convert first 100 rows to 28x28 matrix and plot
    y = matrix(as.matrix(mnist_data[i, 1:784]), nrow=28)
    image( y[,nrow(y):1], axes = FALSE, col = gray(255:0 / 255))}
```

1.1.4 How neurons work

- Structure: dendrites, cell body, axon, synapses

- Neuron triggers, sends action potential (voltage) through axon
- Transmission across synapse is chemical
  - Action potential causes release of neurotransmitters
  - Diffuse over to other side, create signal in next dendrite
  - Synapse connection can be weak or strong
- **Hebbian learning**: neurons that fire together, wire together
  - Synapses grow stronger through experience
  - Long-term potentiation: more dendritic receptors

1.1.5 Perceptron: Representation

```
perceptron = function() {
grid.lines(c(0.6, 0.9), rep(0.5, 2), arrow = arrow(type = "closed", length = unit(0.1, "inches")))
for (j in 1:4) {
  i <- 0.5 + c(0, 2:4)[j]
  l <- sqrt(0.5^2 + (0.5 - i/5)^2) / 0.1
  grid.lines(c(0.1, 0.6 - 0.5/l), c(i/5, 0.5 - (0.5 - i/5)/l), arrow = arrow(type = "closed",
      length = unit(0.1, "inches")))
  grid.circle(0.1, i/5, r = 0.05, gp = gpar(fill = "white"))
  n <- c("p", 3:1)[j]
  grid.text(substitute(x[n], list(n = n)), 0.1, i/5)
  grid.text(substitute(w[n], list(n = n)), 0.3, unit(0.5 - (0.5 - i/5) / 0.5 * 0.3, "npc") + unit
      (1, "lines"))
}
grid.circle(0.6, 0.5, r = 0.1, gp = gpar(fill = "white"))
```

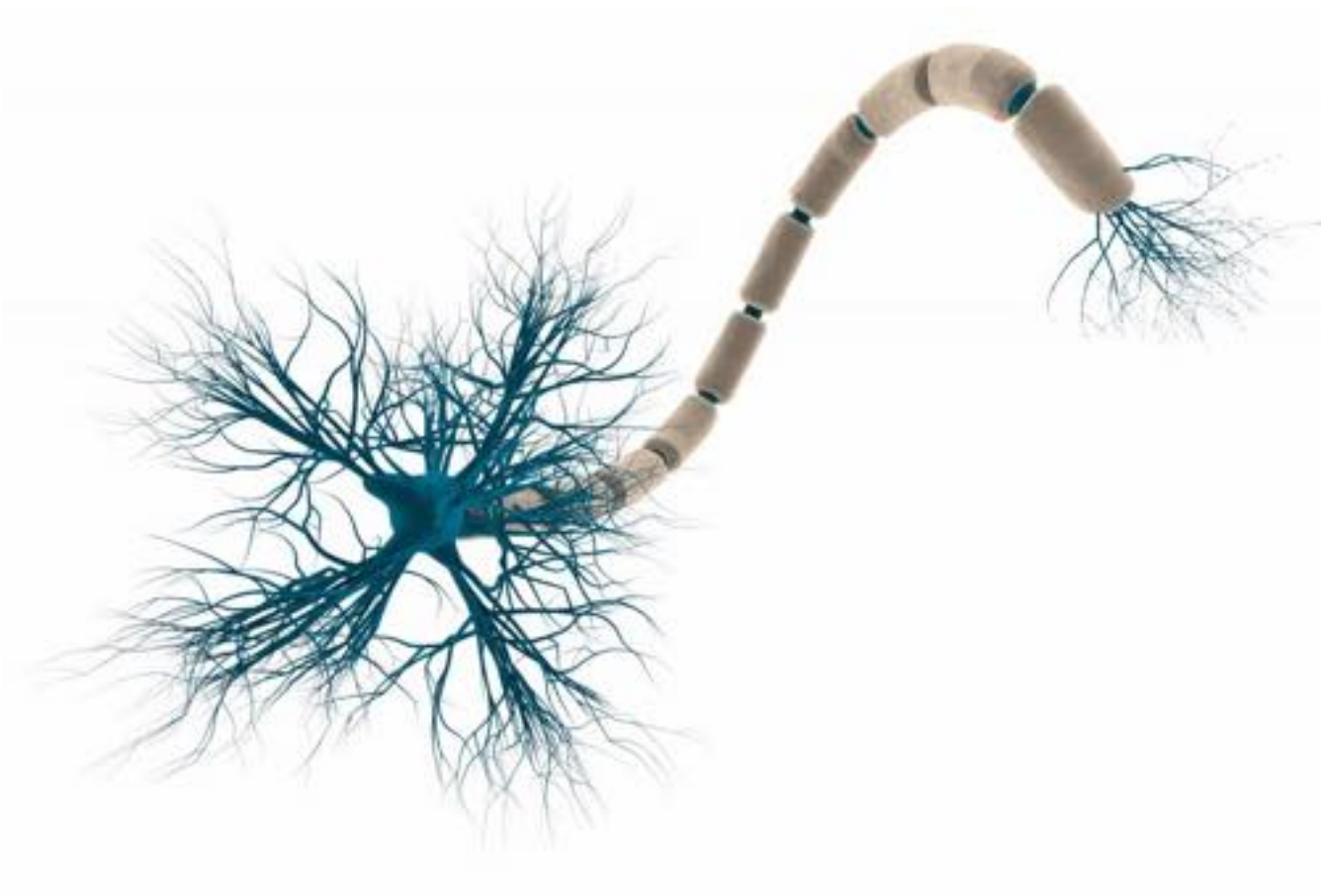
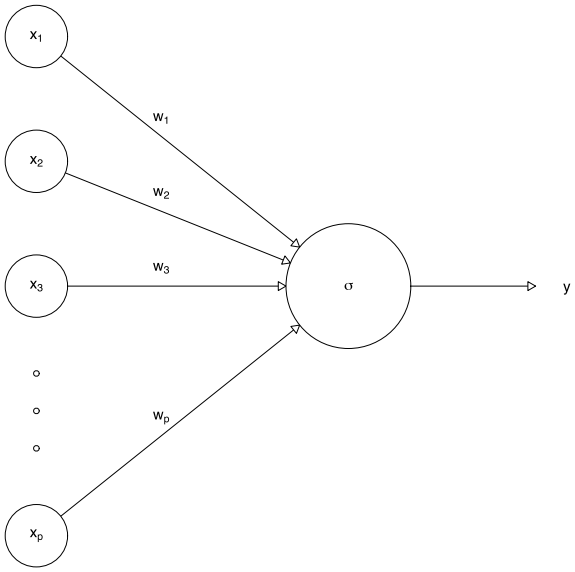


Figure 1.1: A Neuron

```
grid.text(expression(sigma), 0.6, 0.5)
grid.points(unit(rep(0.1, 3), "npc"),
           unit(seq(1.2, 1.8, length.out = 3)/5, "npc"), size = unit(0.5, "char"),
           pch = 1, gp = gpar(fill = 1))
grid.text("y", 0.95, 0.5)
}
```

```
perceptron()
```



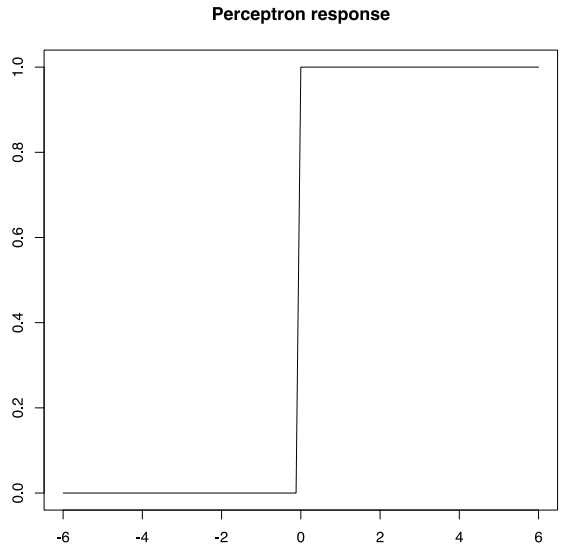
- p binary inputs $x_1..x_p$
- One binary output $y = f(x)$
- Single neuron, every connection has weight $w_1..w_p \in IR$
- Neuron receives weighted sum of inputs
- Threshold function σ defines when neuron fires

$$f(x) = \sigma(\sum_{i=1}^{p} w_i x_i) = \sigma(w \cdot x) = \begin{cases} 1, & w \cdot x \geq \theta \\ 0, & \text{otherwise} \end{cases}$$

- Mathematical simplification: add bias $w_0 = -\theta$, $x_0 = 1$

$$f(x) = \sigma(\sum_{i=0}^{p} w_i x_i) = \sigma(w \cdot x) = \begin{cases} 1, & w \cdot x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

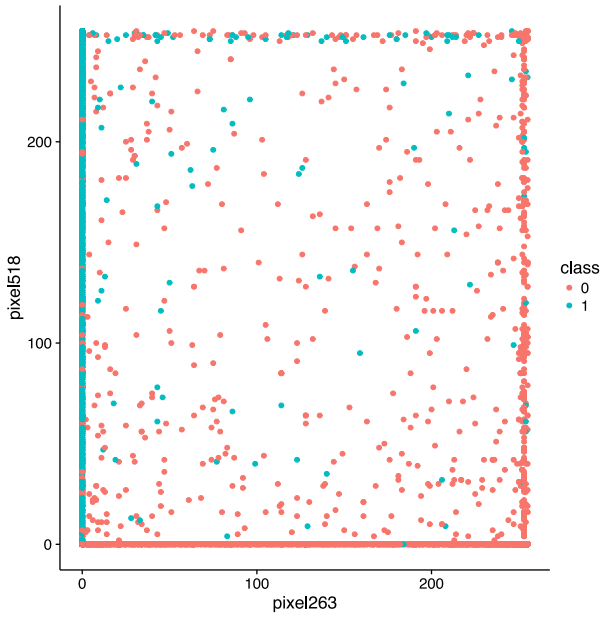
```
px <- seq(-6, 6, length.out = 101)
plot(px, px>=0, type = "l", xlab = "", ylab = "", main = "Perceptron response")
```



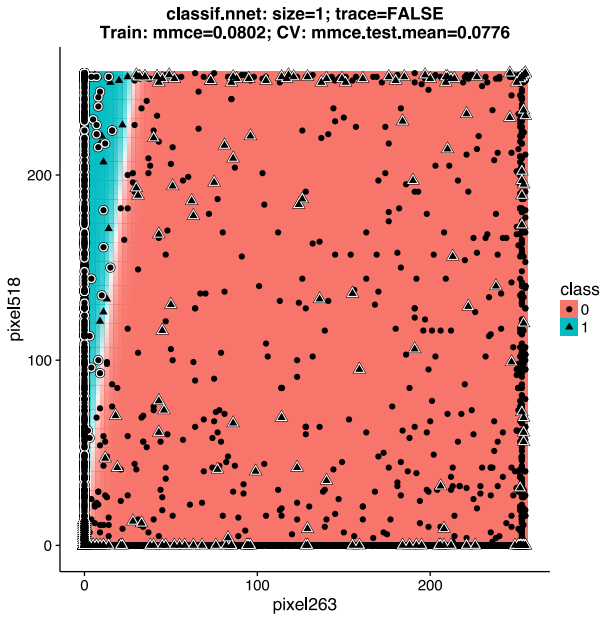
1.1.6 What can perceptrons learn?

- Can it learn logical gates (AND, OR, XOR)?
- Shall we try MNIST?
  - 784 inputs, lots of weights to learn.

```
mnist_bin = droplevels(subset(mnist_data, class %in% c(0,1))) #MNIST with only 0 and 1 (binary)
# Now, plot output for 2 random pixels
ggplot(data=mnist_bin, aes(x=pixel263, y=pixel518, color=class)) + geom_point()
```



```
mnist_task = makeClassifTask(data = mnist_bin, target = "class")
# We're mimicking a perceptron by constructing a neural net with 1 hidden node
# Actually this is a sigmoid perceptron, but still quite close
perceptron = makeLearner("classif.nnet", par.vals = list(size = 1, trace = FALSE))
plotLearnerPrediction(perceptron, mnist_task, features = c("pixel263", "pixel518")) + theme_
    cowplot()
```



Perceptrons can only learn linear functions

- Indeed:

$$f(x) = \sigma(\sum_{i=0}^{p} w_i x_i) = w_0 x_0 + w_1 x_1 + .. + w_p x_p$$

- In n dimensions, perceptron can only learn hyperplanes
- Simple concepts like XOR cannot be learned
  - Requires a 2-layer perceptron
- Hence, we need more than 1 neuron, we want networks of these

1.1.7 Perceptron: Optimization (training)

- Perceptron $\sim$ device that makes decisions by weighing up evidence
- You have to learn how much each input affects the desired outcome
- Like a social network: should I go see movie X?
  - Each friend possibly influences your decision
  - You learn whose recommendations to trust more than others
- How to learn those weights?

Perceptron training rule

- Initialize weights randomly

- With every example i, update weights:

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta(t - o)x_i$$

with:
- t = c(x): target value
- o: perceptron output
- η: small constant (e.g. 0.1) called learning rate

Why does this work?

- If outputting correct value: do nothing
- If outputting incorrect value:
  - Predict 0 when target is 1: $t - o = 1$
    - Increase weight of inputs that are hot
  - Predict 1 when target is 0: $t - o = -1$
    - Decrease weight of inputs that are hot

Convergence

- Any guarantee that it will converge?
- Proof that it will converge if:
  - η is sufficiently small
  - Training data is linearly separable
    - with noise, no convergence
    - Δw_i will move hyperplane towards noise, misclassify other points
    - 'solution': stop after t iterations

1.1.8 Gradient descent

- Still learn hyperplane, but differently
  - Most robust, and generalizes to complex networks
- First, consider linear unit:

$$o = w_0 + w_1 x_1 + .. + w_p x_p$$

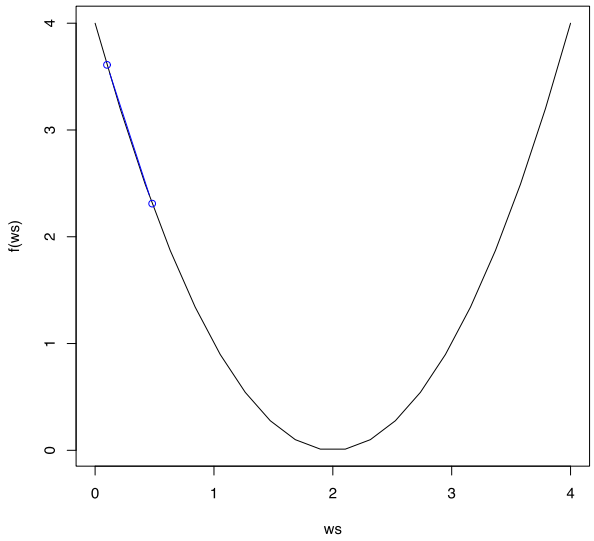
- Let's learn weights that minimize the squared error:

$$E[w] = \frac{1}{2} \sum_{d=1}^{n} (t_d - o_d)^2$$

with n training examples
- Quadratic function of linear function of weight = quadratic functions of weights
  - Much easier to learn (no local mimima)
  - True for perceptron (not general network)

How do you find optimum?

```
ws = seq(0,4,len=20) # for plotting
f = function(w) { (w-2)^2 } # function to optimize
plot(ws , f(ws), type="l") # Plot target
w = c(0.1,0.1-0.1*2*(0.1-2)) # We'll see later how to compute this
lines (w, f(w), type="b",col="blue")
```



- Gradient:

$$\nabla E[w] = \left[\frac{\delta E}{\delta w_0}, \frac{\delta E}{\delta w_1}, \ldots, \frac{\delta E}{\delta w_p}\right]$$

- Vector where each component is partial derivative of error with respect to that weight
  - Large value: move in that direction a lot
- Training rule:

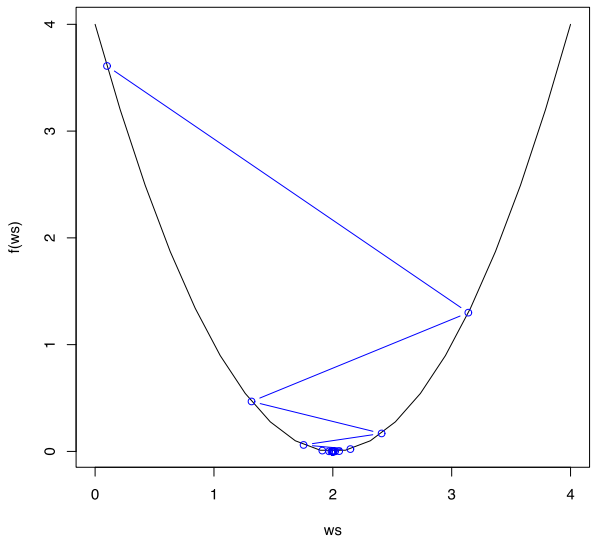
$$\Delta w = -\eta E[w]$$

  i.e.

$$\Delta w_i = -\eta \frac{\delta E}{\delta w_i}$$

```
grad = function(w){2*(w-2)} # gradient df/dw

w = 0.1 # initialize (first guess)
learningRate = 0.8 # try smaller and larger values
wtrace = w # initialize (first guess)
ftrace = f(w) # store y-values
for (step in 1:100) {
w = w - learningRate*grad(w) # gradient descent update
wtrace = c(wtrace,w) # store next x-value
ftrace = c(ftrace,f(w)) # store next y-value
}
```

```
plot(ws , f(ws), type="l") # Plot target
lines( wtrace , ftrace , type="b",col="blue") # Plot steps
```



Computing the gradient

$$\frac{\delta E}{\delta w_i} = \frac{\delta}{\delta w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \tag{1.1}$$

$$= \frac{1}{2}\sum_d \frac{\delta}{\delta w_i}(t_d - o_d)^2 \tag{1.2}$$

$$= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\delta}{\delta w_i}(t_d - o_d) \tag{1.3}$$

$$= \sum_d (t_d - o_d)\frac{\delta}{\delta w_i}(t_d - w\cdot x_d) \tag{1.4}$$

$$= \sum_d (t_d - o_d)(-x_{i,d}) \tag{1.5}$$

Gradient descent

- Initialize each w_i to some small random value
- Until termination condition is met:
  - Initialize each Δw_i to 0
  - For each (x,t) in training examples, do:

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

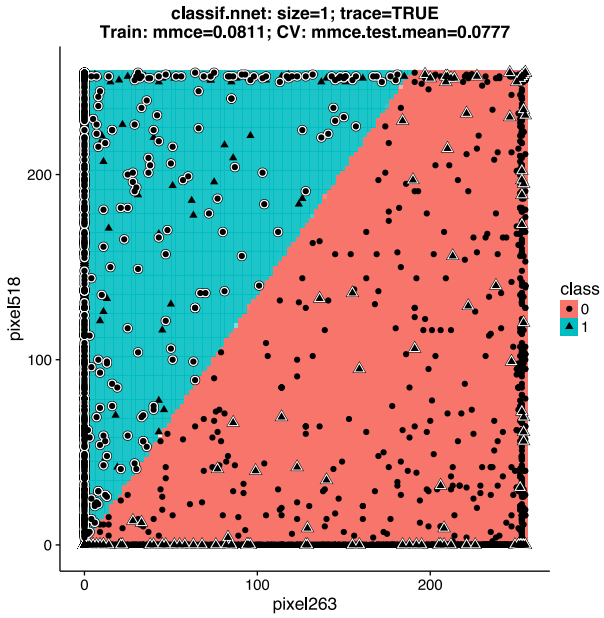
  - For each linear unit weight w_i, do:

$$w_i \leftarrow w_i + \Delta w_i$$

```r
# Print the gradient descent trace. Note that the implementation does multiple restarts.
perceptron = makeLearner("classif.nnet", par.vals = list(size = 1, trace = TRUE))
plotLearnerPrediction(perceptron, mnist_task, features = c("pixel263", "pixel518")) + theme_
    cowplot()
```

```
# weights:  5
initial  value 9530.621811
iter  10 value 4169.605599
iter  20 value 3724.220723
final  value 3631.881549
converged
# weights:  5
initial  value 9940.421613
iter  10 value 3198.200825
iter  20 value 3107.115886
iter  30 value 3101.952981
iter  40 value 3100.440099
iter  50 value 3100.342121
final  value 3100.340094
converged
# weights:  5
initial  value 9070.492085
iter  10 value 3108.088777
iter  20 value 3098.514438
iter  30 value 3098.487609
iter  40 value 3098.385022
final  value 3098.380162
converged
# weights:  5
initial  value 10763.113457
iter  10 value 3221.535928
iter  20 value 3176.593675
iter  30 value 3056.479477
iter  40 value 2988.343142
iter  50 value 2988.140473
final  value 2988.140068
converged
# weights:  5
initial  value 9557.067527
iter  10 value 3133.056692
iter  20 value 2992.199415
iter  30 value 2964.606684
final  value 2964.438636
converged
# weights:  5
initial  value 9998.141410
iter  10 value 3131.036746
iter  20 value 3092.113207
iter  30 value 3091.195607
iter  30 value 3091.195607
iter  30 value 3091.195607
final  value 3091.195607
converged
# weights:  5
initial  value 11961.605832
iter  10 value 3335.052515
iter  20 value 3095.471140
iter  30 value 3093.161882
iter  40 value 3091.584110
iter  40 value 3091.584100
final  value 3091.583987
converged
# weights:  5
initial  value 9425.305410
iter  10 value 3835.032518
iter  20 value 3050.753854
iter  30 value 3026.429341
```

```
final  value 3026.429013
converged
# weights:  5
initial  value 10708.858053
iter  10 value 3262.151504
iter  20 value 3057.142801
iter  30 value 3044.504783
final  value 3043.655040
converged
# weights:  5
initial  value 9551.760135
iter  10 value 3376.106008
iter  20 value 3052.876431
iter  30 value 3024.650467
final  value 3024.642549
converged
# weights:  5
initial  value 9741.689711
iter  10 value 3744.021070
iter  20 value 3302.774107
iter  30 value 3192.853632
iter  40 value 3108.244907
iter  50 value 3100.216152
iter  60 value 3096.131583
iter  70 value 3096.107401
iter  80 value 3096.071203
final  value 3096.062682
converged
```



Gradient descent: conclusions

- Guaranteed to converge to hypothesis with minumum squared error
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data is not separable by H
- In general, there is no closed form solution for $\frac{\delta E}{\delta w_i} = 0$, so we need gradient descent

Stochastic gradient descent

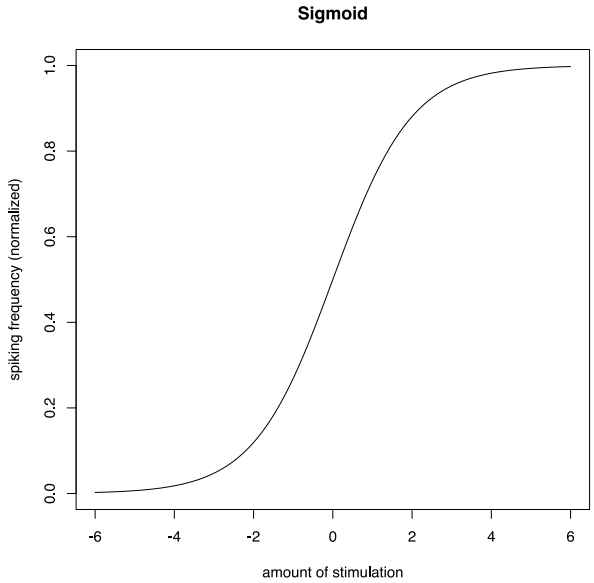
- Batch mode: look at all examples, compute gradient, update by that gradient
  - Ensures guarantees we just discussed
- Stochastic gradient descent (incremental mode):
  - Update weights after each training example
  - Doesn't go straight to goal, but 'zigzags' towards it
- Works well in practice
  - Faster: you do more updates (even if they zigzag)
  - Noise may be helpful: avoids small local optima (bouncing ball)

1.1.9 Refining the model of the neuron

- Action potentials are not big jolts, but short-lived spikes
- Neuron spikes more frequently the more it is stimulated
  - spike frequency ν
  - saturates at max. frequency
- The amount of stimulation is called membrane potential u
- Spike frequency is non-linear function of membrane potential
  - Sigmoid function, or S-curve
  - Most important curve in the world: phase transitions

$$\nu = \sigma(u) = \frac{1}{1 + e^{-u}}$$

```
library(pracma)
plot(px, sigmoid(px, a = 1), type = "l",
       ylab = "spiking frequency (normalized)", xlab = "amount of stimulation", main = "Sigmoid")
```



1.1.10 Neural net

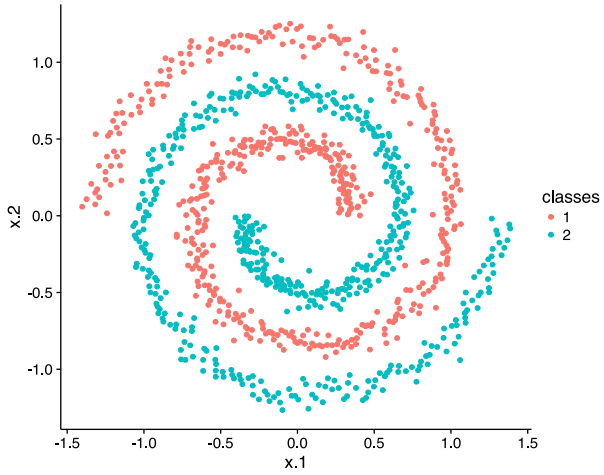
- Representation
  - Many neuron-like units
  - Many weighted interconnections
  - Highly parallel processes
- Optimization
  - Learning: tune weights automatically

```
library(devtools)
source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466
    c1474d0a505ff044412703516c34f1a4684a5/nnet_plot_update.r')

library(clusterGeneration)

seed.val<-2
set.seed(seed.val)

num.vars<-8
num.obs<-1000

#input variables
cov.mat<-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars<-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)

#output variables
parms<-runif(num.vars,-10,10)
y1<-rand.vars %*% matrix(parms) + rnorm(num.obs,sd=20)
parms2<-runif(num.vars,-10,10)
y2<-rand.vars %*% matrix(parms2) + rnorm(num.obs,sd=20)

#final datasets
rand.vars<-data.frame(spirals)
resp<-data.frame(y1,y2)
names(resp)<-c('Y1','Y2')
dat.in<-data.frame(resp,rand.vars)
```

```
SHA-1 hash of file is 74c80bd5ddbc17ab3ae5ece9c0ed9beb612e87ef
```

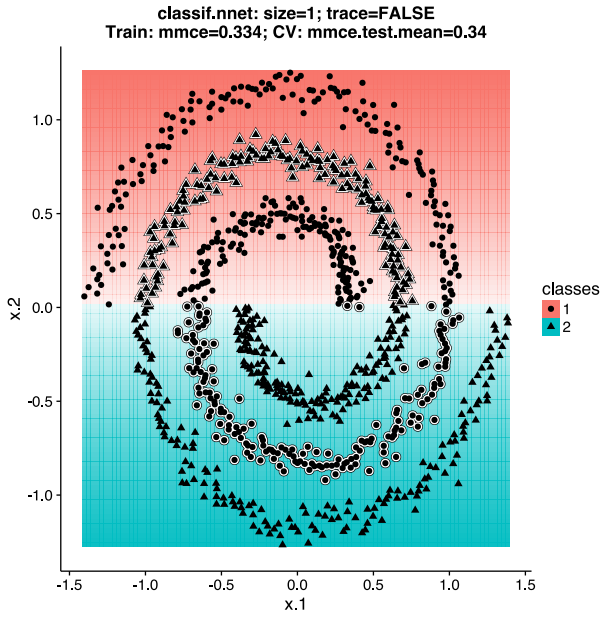
Let's start with a simpler dataset

```
library(mlbench)
spirals = as.data.frame(mlbench.spirals(n=1000, cycles=1.5, sd=0.05))
spiral_task = makeClassifTask(data = spirals, target = "classes")
ggplot(data=spirals, aes(x=x.1, y=x.2, color=classes)) + geom_point() + coord_fixed(ratio=1)
```



The perceptron can't learn it

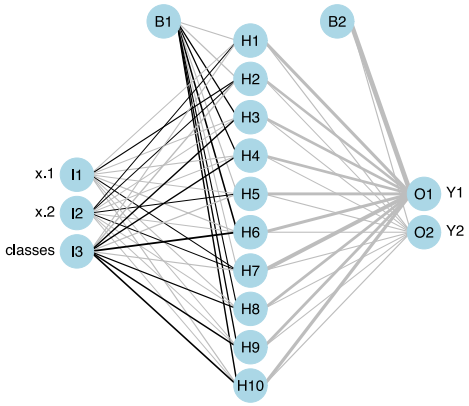
```
# The perceptron again
mlp = makeLearner("classif.nnet", par.vals = list(size = 1, trace = FALSE))
plotLearnerPrediction(mlp, spiral_task, features = c("x.1", "x.2")) + theme_cowplot()
```



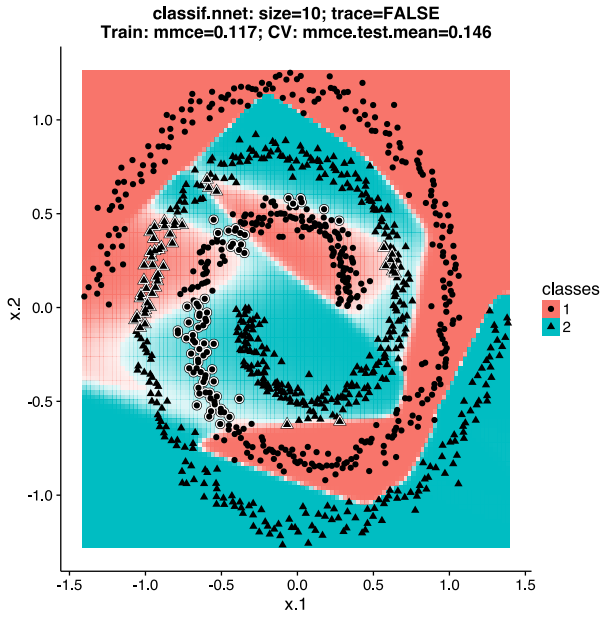
We can build a neural net, with 1 layer of hidden nodes - Black = positive weights, gray = negative, thickness = magnitude - B1, B2,... are the bias layers

```
library(nnet)
```

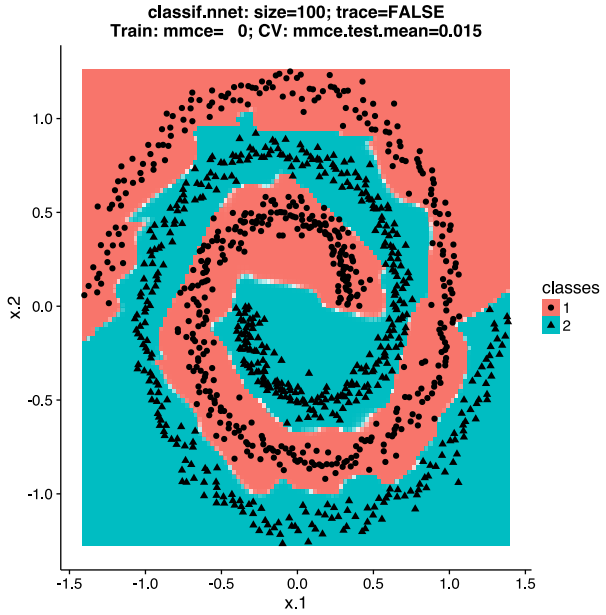
```
mod1<-nnet(rand.vars,resp,data=spirals,size=10,trace=F)
plot.nnet(mod1)
```




```
# 1 hidden layer, 10 nodes
mlp = makeLearner("classif.nnet", par.vals = list(size = 10, trace = FALSE))
plotLearnerPrediction(mlp, spiral_task, features = c("x.1", "x.2")) + theme_cowplot()
```



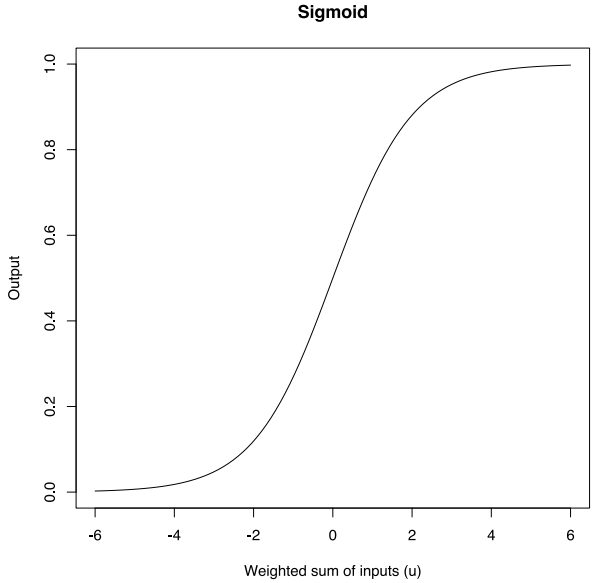
```r
# 1 hidden layer, 100 nodes
mlp = makeLearner("classif.nnet", par.vals = list(size = 100, trace = FALSE))
plotLearnerPrediction(mlp, spiral_task, features = c("x.1", "x.2")) + theme_cowplot()
```



1.1.11 Learning the network weights

- With a networks of neurons, you could learn much more complex functions
- But how to learn them?
  - Perceptron rule never generalized to work on networks
- Gradient descent!
  - But, network of linear functions is still a linear function
  - We need a non-linear function in the nodes
  - Preferably one that is easily differentiable
  - And inspired by the brain

```r
plot(px, sigmoid(px, a = 1), type = "l",
       ylab = "Output", xlab = "Weighted sum of inputs (u)", main = "Sigmoid")
```



Nice property:

$$\frac{\delta\sigma(u)}{\delta u} = \sigma(u)(1-\sigma(u))$$

- When we do gradient descent, we can replace differentials easily

Computing the gradient for sigmoid units

$$\frac{\delta E}{\delta w_i} = \frac{\delta}{\delta w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \tag{1.6}$$

$$= \frac{1}{2}\sum_d \frac{\delta}{\delta w_i}(t_d - o_d)^2 \tag{1.7}$$

$$= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\delta}{\delta w_i}(t_d - o_d) \tag{1.8}$$

$$= \sum_d (t_d - o_d)\frac{\delta}{\delta w_i}(-\sigma(u_d)) \tag{1.9}$$

$$= -\sum_d (t_d - o_d)\frac{\delta\sigma(u_d)}{\delta u_d}\frac{\delta u_d}{\delta w_i} \tag{1.10}$$

$$= -\sum_d (t_d - o_d)\sigma(u_d)(1-\sigma(u_d))x_{i,d} \tag{1.11}$$

$$\tag{1.12}$$

$$u_d = w \cdot x_d$$

1.1.12 Backpropagation

- Propagate example through the network
- Back-propagate the error signal from output back through the layers

- How does changing the j weights of neuron k change the final error? $\frac{\delta E}{\delta w_{j,k}}$

- To compute those, we introduce an intermediate quantity: the 'error' in the signal of neuron k:

$$\delta_k = -\frac{\delta E}{\delta u_k}$$

- The errors δ_k can be computed based on the errors in the next layer
- Order of computation is back to front, ultimately relating δ_k to $\frac{\delta E}{\delta w_{j,k}}$

Understanding δ_k

- Imagine a demon that adds a little change Δu_k to the input of neuron k
  - The neuron now outputs $\sigma(u_k + \Delta u_k)$ instead of $\sigma(u_k)$
  - Propagates through network, ultimately causing an error $\frac{\delta E}{\delta u_k}\Delta u_k$
- A good demon helps you improve the error by trying to find a Δu_k that reduces the error
  - If $\frac{\delta E}{\delta u_k}$ is large, choose Δu_k to reduce it

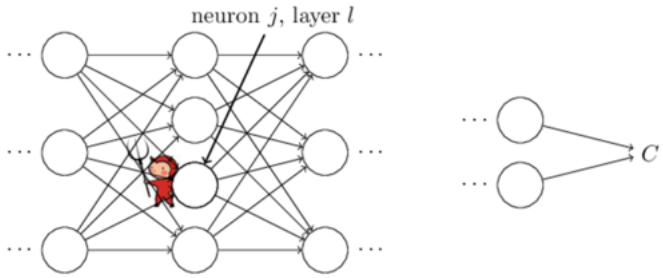


Figure 1.2: Tampering with the weights creates an error downstream.

For hidden node h, one layer before nodes k:

$$\frac{\delta E}{\delta u_h} = \sum_{k \in outs(h)} \frac{\delta E}{\delta u_k}\frac{\delta u_k}{\delta u_h} \quad \text{(sum rule + chain rule)} \tag{1.13}$$

$$= \sum_{k \in outs(h)} -\delta_k \frac{\delta u_k}{\delta u_h} \tag{1.14}$$

$$= \sum_{k \in outs(h)} -\delta_k \frac{\delta u_k}{\delta o_h}\frac{\delta o_h}{\delta u_h} \tag{1.15}$$

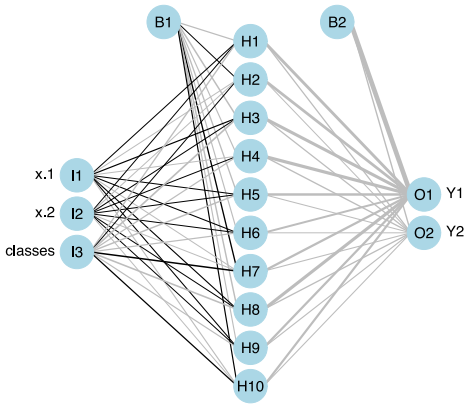
$$= \sum_{k \in outs(h)} -\delta_k w_{k,h}\frac{\delta o_h}{\delta u_h} \tag{1.16}$$

$$= \sum_{k \in outs(h)} -\delta_k w_{k,h} o_h(1 - o_h) \tag{1.17}$$

$$\tag{1.18}$$

$$\delta_h = o_h(1 - o_h) \sum_{k \in outs(h)} \delta_k w_{k,h}$$

```
mod1<-nnet(rand.vars,resp,data=spirals,size=10,trace=F)
plot.nnet(mod1)
```



Backpropagation algorithm

- Initialize all weights to small random numbers
- Until convergence, do:
  - For each training example, do:
    - Forward pass: propagate example, compute outputs
    - For each output unit k:

$$\delta_k = o_k(1 - o_k)(t_k - o_k)$$

    - For each hidden unit h:

$$\delta_h = o_h(1 - o_h) \sum_{k \in outs(h)} \delta_k w_{k,h}$$

    - Update each network weight:

$$w_{i,j} = w_{i,j} + \Delta w_{i,j} \quad \text{where} \quad \Delta w_{i,j} = \eta \delta_j x_{i,j}$$

- You can use batch or stochastic gradient descent
- There exist efficient techniques using matrix products

Properties of backprop

- Does gradient descent over entire network weight vector
- Works on any directed graph (doesn't require layers)
- Will find a local (not global) optimum
  - Works well in practice, with restarts (epochs)
  - Learning rate will never be optimal everywhere (slow)
- Speedup by including weight momentum α

  - Like a ball accellerating down the hill
  - Also helps avoid local minima

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Minimizes error over training example (can still overfit)
- Training can take thousands of iterations (slow)
- Using networks after training is very fast

Properties of backprop

- Convergence: avoid local minima
  - Add momentum
  - Stochastic gradient descent
  - Restart with different initial weights
  - Initialize weights near zero: start with near-linear networks
- Expressiveness
  - Can approximate any bounded continuous function, with arbitrarily small error, by 1-layer network
    - Linear combination of sigmoid functions
  - Any function can be approximated, with arbitrarily small error, by 2-layer network.

1.1.13 Overfitting

- If you keep adding hidden units, you can just memorize the input
- Penalize large weights by complexity penalty γ (weight decay):

$$E(w) = \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{k,d} - o_{k,d})^2 + \gamma \sum_{i,j} w_{j,i}^2$$

- Train on target slopes as well as values:
  - Penalty on difference between target slope and actual slope:

$$E(w) = \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} \left[(t_{k,d} - o_{k,d})^2 + \mu \sum_{j \in inputs} \left(\frac{\delta t_{k,d}}{\delta x_d^j} - \frac{\delta o_{k,d}}{\delta x_d^j} \right)^2 \right]$$

- Weight sharing: translate the same weight over layer
  - in vision problems: nearby weights should have similar values
  - convolutional neural networks
- Early stopping
  - Use a validation set and detect when test error goes up
  - Equivalent to weight decay if you start with small weights

Understanding the weights: Autoencoders

- Output layer must exactly return the input
- Forced to code the information in fewer bits
- Sparse autoencoders: use many more nodes in hidden layer, but switch most of them off at any given time

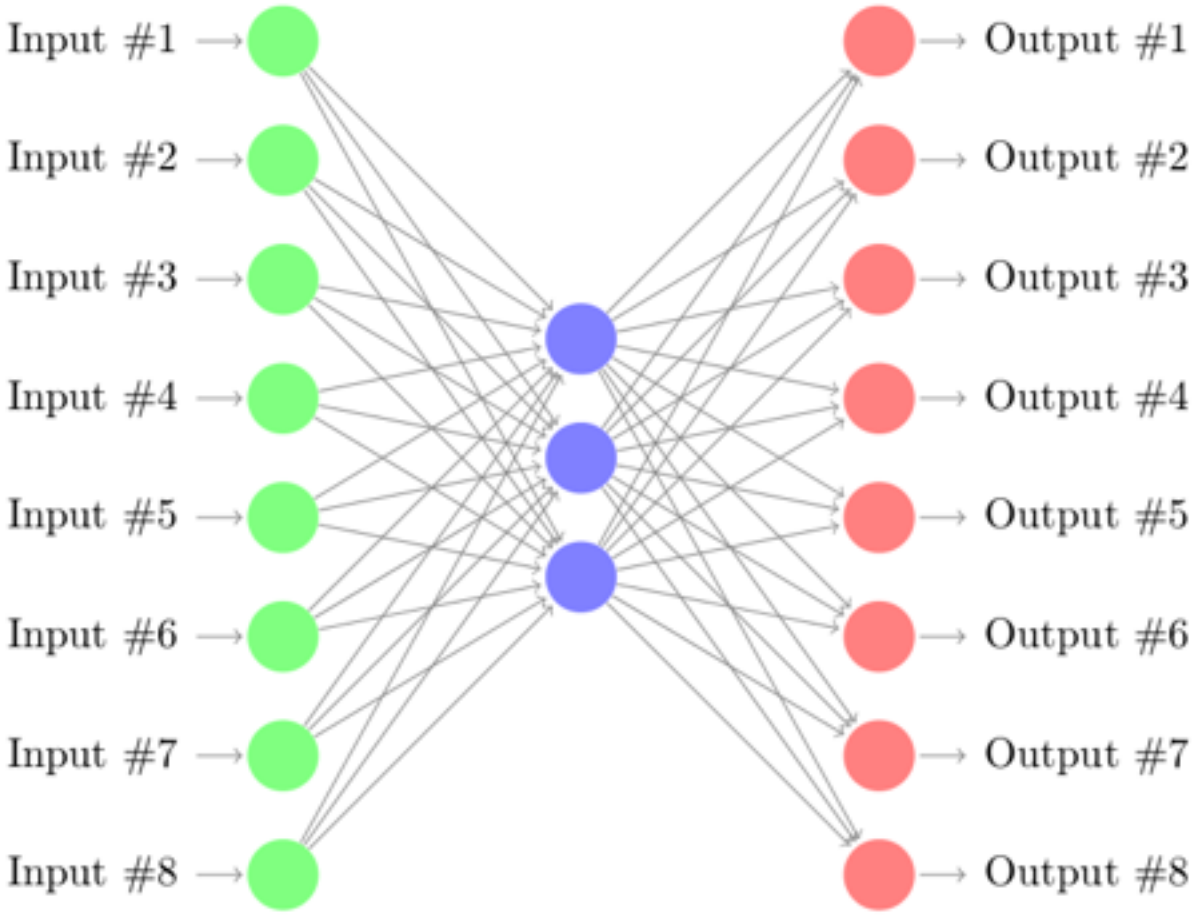


Figure 1.3: An autoencoder.

Stacked autoencoders

- Using backprop for deep networks is extremely hard
  - Error signal diffuses until nothing is left
- Sparse autoencoders can be pre-trained by greedy layer-wise training
  - Freeze weights of the other layers, train one layer at a time
  - Use raw output of previous layers to train subsequent layers
- A softmax function can be used in the last layer to squash the activation to a probability value (as in logistic regression)

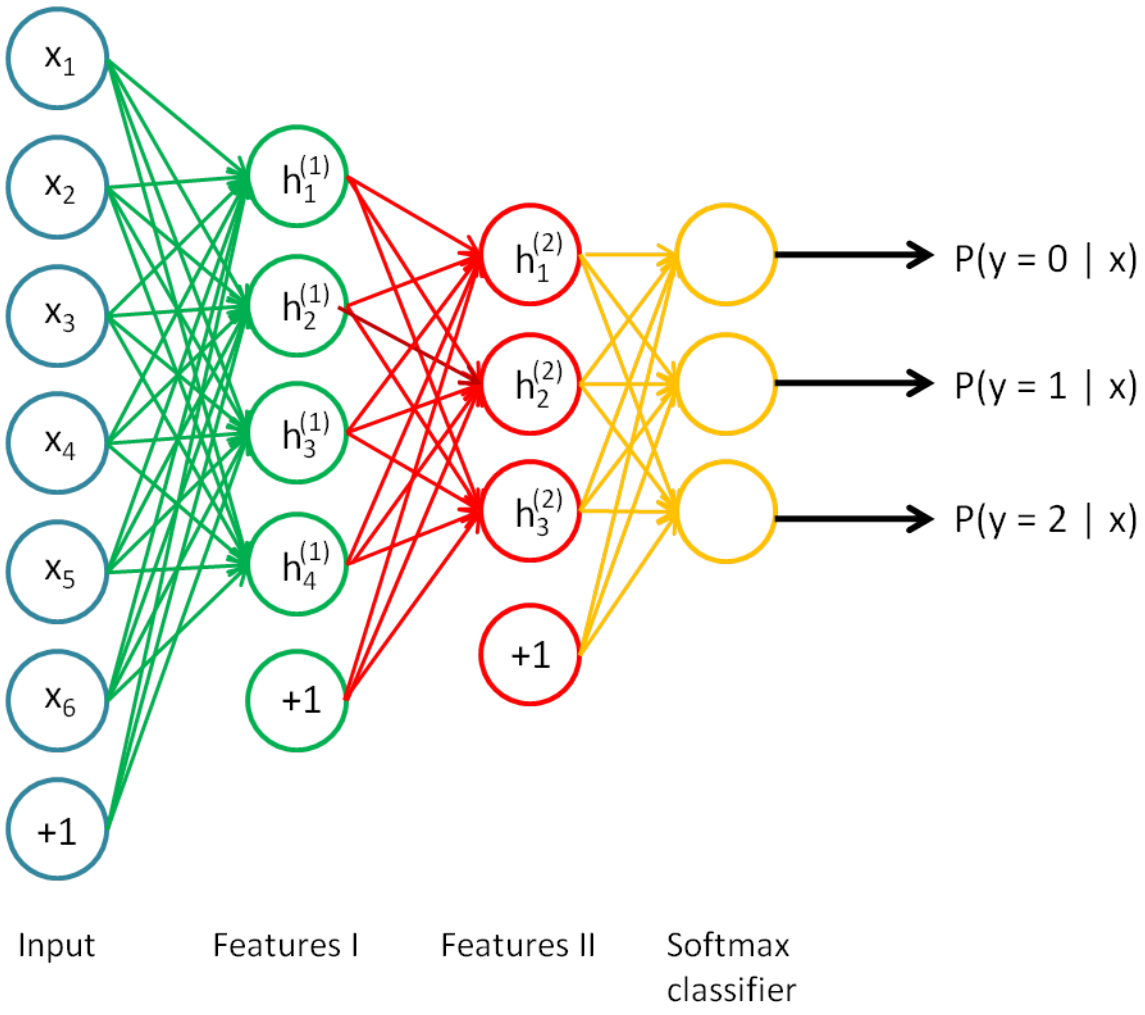


Figure 1.4: Stacked autoencoders

Stacked sparse autoencoders encode increasingly complex features - First layer tends to learn first-order features of the raw input (e.g. edges of image) - Second layer learns second-order features (e.g. contours, corners),...

1.1.14 Convolutional neural network

- For audio, video, images (smooth data)

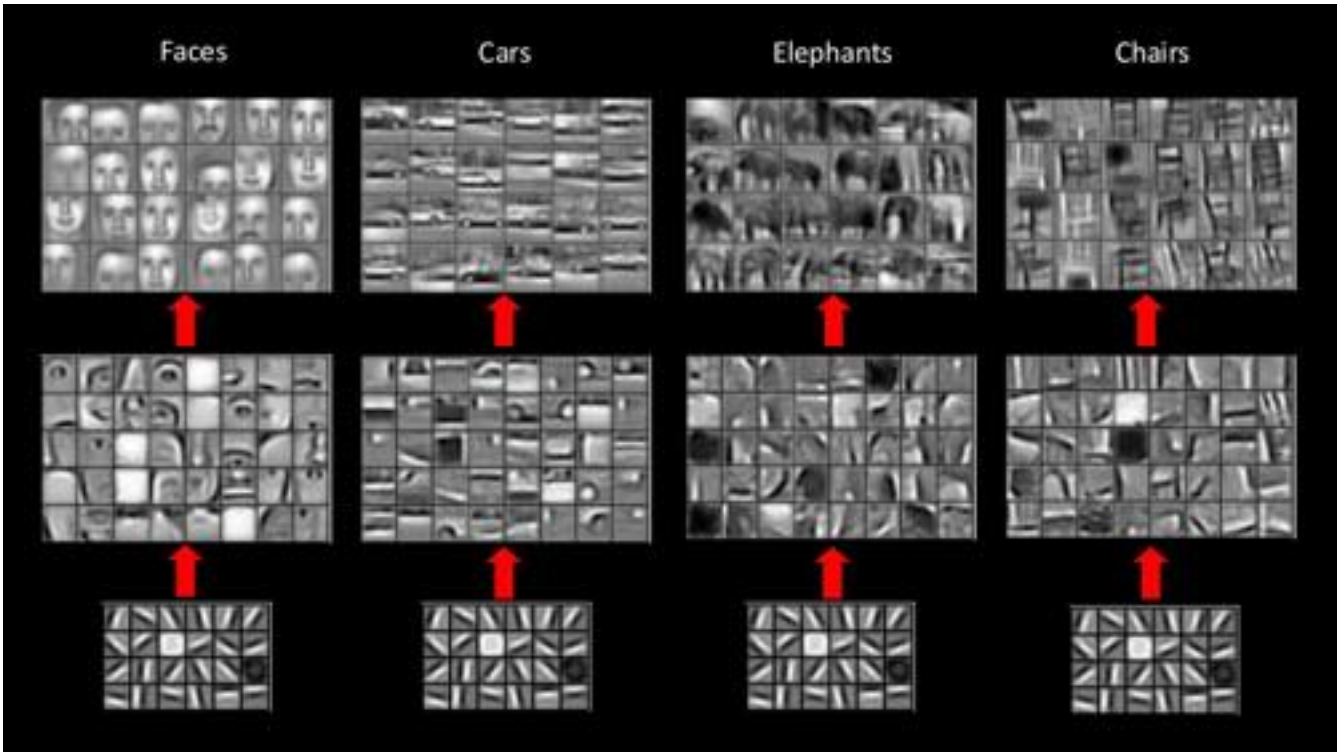
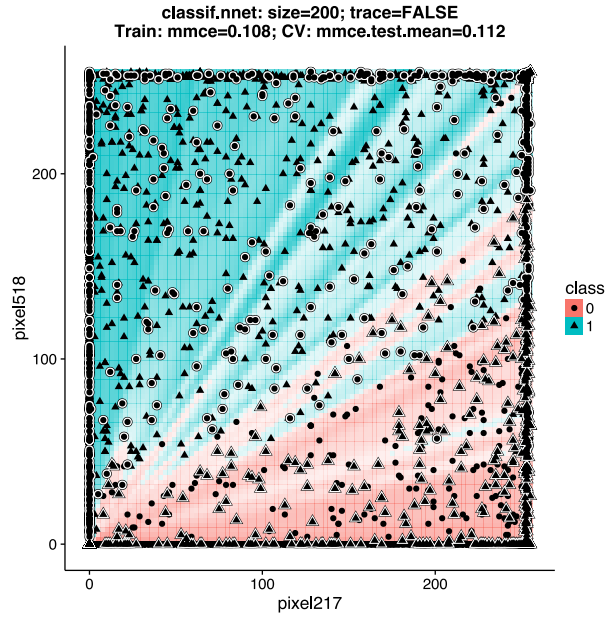


Figure 1.5: Feature learning with stacked autoencoders

- Uses multiple copies of the same neuron
- Every input neuron only wired together with nearby nodes

```
mlp = makeLearner("classif.nnet", par.vals = list(size = 200, trace = FALSE))
plotLearnerPrediction(mlp, mnist_task, features = c("pixel217", "pixel518")) + theme_cowplot()
```



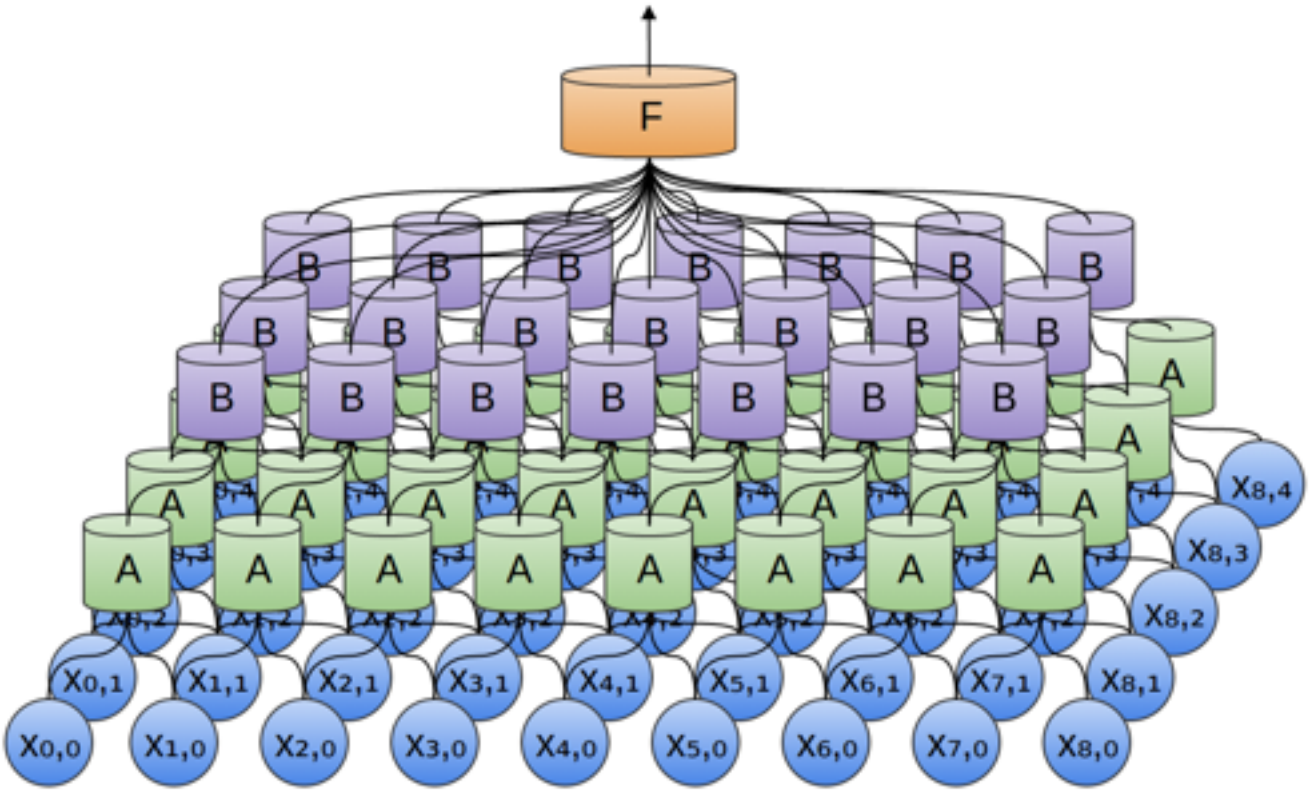


Figure 1.6: Convolutional neural network

1.1.15 Questions?

1.1.16 Acknowledgements

Thanks to Bernd Bischl and Tobias Glasmachers for useful input.