Module 11: Neural Networks

TMA4268 Statistical Learning V2020

Stefanie Muff, Department of Mathematical Sciences, NTNU

April xx, 2020

Introduction

- Neural networks (NN) were first introduced in the 1990's.
- Shift from statistics to computer science and machine learning, as they are highly parameterized
- Statisticians were skeptical: "It's just a nonlinear model".
- After the first hype, NNs were pushed aside by boosting and support vector machines.
- Revival since 2010: The emergence of *Deep learning* as a consequence of improved computer resources, some innovations, and applications to image and video classification, and speech and text processing

Learning material for this module

(on the reading list)

- Classnotes 25.03.2019
- Classnotes 28.03.2019

See also *References and further reading* (last slide), for further reading material.

What will you learn?

- Why a module on neural networks?
- Translating from statistical to neural networks language
  - linear regression
  - logistic regression
  - multiclass (multinomial) regression
- Feedforward networks
  - one hidden layer
  - universal approximation theorem

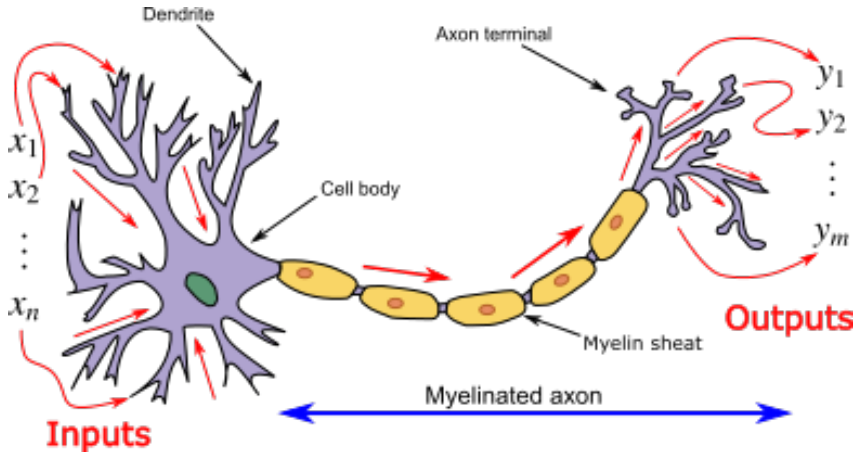
- Neural network parts
  - model: architecture, activation functions
  - method: loss fuction
  - algorithm: how to estimate parameters, gradient descent and back-propagation
  - recent developents
- Deep learning
  - the timeline
  - Keras
- References

Why a module on neural networks?

- Every day you read about the success of AI, machine learning — and in particular deep learning.
- In the last five years the field of deep learning has gone from low level performance to excellent performance — particularly in image recognition and speech transcription.

- Deep learning is based on a layered artificial neural network structure.

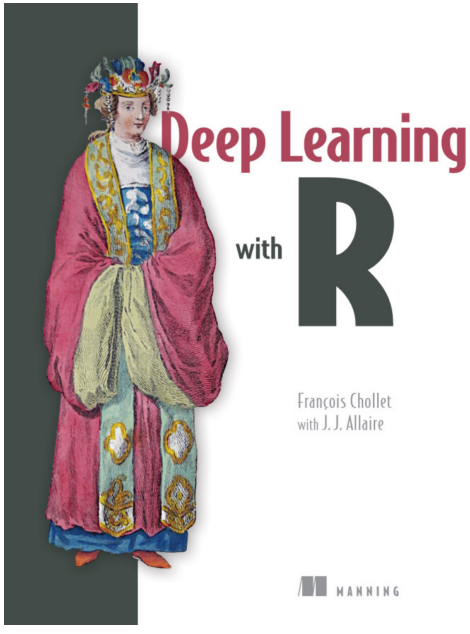
But, what is a *neural network*?

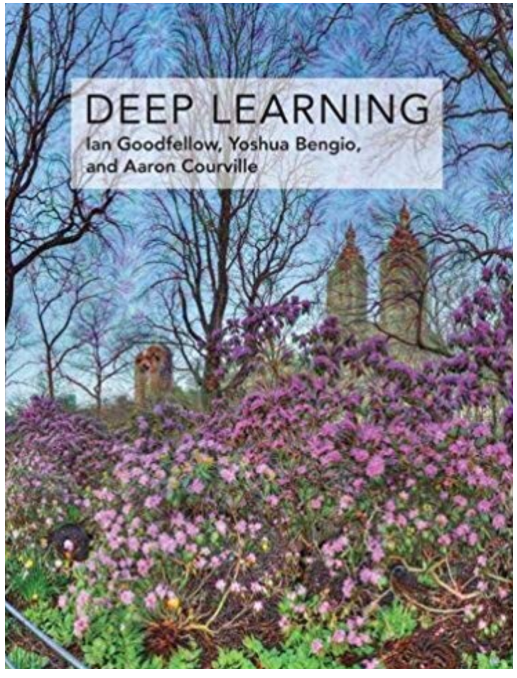


Neuron and myelinated axon, with signal flow from inputs at dendrites to outputs at axon terminals. Image credits: By Egm4313.s12 (Prof. Loc Vu-Quoc) https://commons.wikimedia.org/w/index.php?curid=72816083

- Understanding the basic building blocks of artificial neural network, and how they are connected to concepts in mathematical statistics might be useful!
- The focus here will mainly be to relate what we have learned from regression and classification in this course to the basic framework of artificial neural network,
- and we will touch upon the new and hot stuff.

- There are several (self-study) learning resources (some listed under 'further references') that the student my turn to for further knowledge into deep learning, but this presentation is heavily based on Chollet and Allaire (2018), with added formulas and theory.
- There is a new IT3030 deep learning course at NTNU.





AI, machine learning and statistics

- Artificial intelligence dates back to the 1950s, and can be seen as *the effort to automate intellectual tasks normally performed by humans* (page 4, Chollet and Allaire (2018)).
- AI was first based on hardcoded rules, but turned out to be intractable for solving more complex, fuzzy problems.
- With the field of *machine learning* the shift is that a system is *trained* rather than explicitly programmed.

Machine learning

- Machine learning is related to mathematical statistics, but differ in many ways (as we will soon see).
- In short, machine learning attachs much larger, and more complex data sets than what is usually done in statistics.
- The focus in machine learning is not on mathematical theory, but is more oriented towards *engineering*, and ideas are proven *empirically* rather than theoretically (which is the case in mathematical statistics).
- According to Chollet and Allaire (2018) (page 19):

  Machine learning isn't mathematics or physics, where major advancements can be done with a pen and a piece of paper. It's an engineering science.

From statistics to artificial neural networks

Recapitulate from Module 3 with the bodyfat dataset that contained the following variables.

- `bodyfat`: % of body fat.
- `age`: age of the person.
- `weight`: body weighth.
- `height`: body height.
- `neck`: neck thickness.
- `bmi`: bmi.
- `abdomen`: circumference of abdomen.
- `hip`: circumference of hip.

We will now look at modelling the `bodyfat` as response and using all other variables as covariates - this will give us

- one numerical output (response), and
- seven covariates
- one intercept

Let n be the number of observations in the training set, here $n = 243$.

Multiple linear regression model

(from Module 3)

We assume

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i = \boldsymbol{x}_i^T \boldsymbol{\beta} + \varepsilon_i \ , \qquad (1)$$

for $i = 1, \ldots, n$, where x_{ij} is the value jth predictor for the ith datapoint, and $\boldsymbol{\beta}^\top = (\beta_0, \beta_1, \ldots, \beta_p)$ the regression coeffficients.

We used the compact matrix notation for all observations $i = 1, \ldots, n$ together:

$$\boldsymbol{Y} = \boldsymbol{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon} \ .$$

Assumptions:

1. $\mathrm{E}(\boldsymbol{\varepsilon}) = \mathbf{0}$.
2. $\mathrm{Cov}(\boldsymbol{\varepsilon}) = \mathrm{E}(\boldsymbol{\varepsilon}\boldsymbol{\varepsilon}^T) = \sigma^2\mathbf{I}$.
3. The design matrix has full rank, $\mathrm{rank}(\boldsymbol{X}) = p + 1$. (We assume $n >> (p + 1)$.)

The classical *normal* linear regression model is obtained if additionally

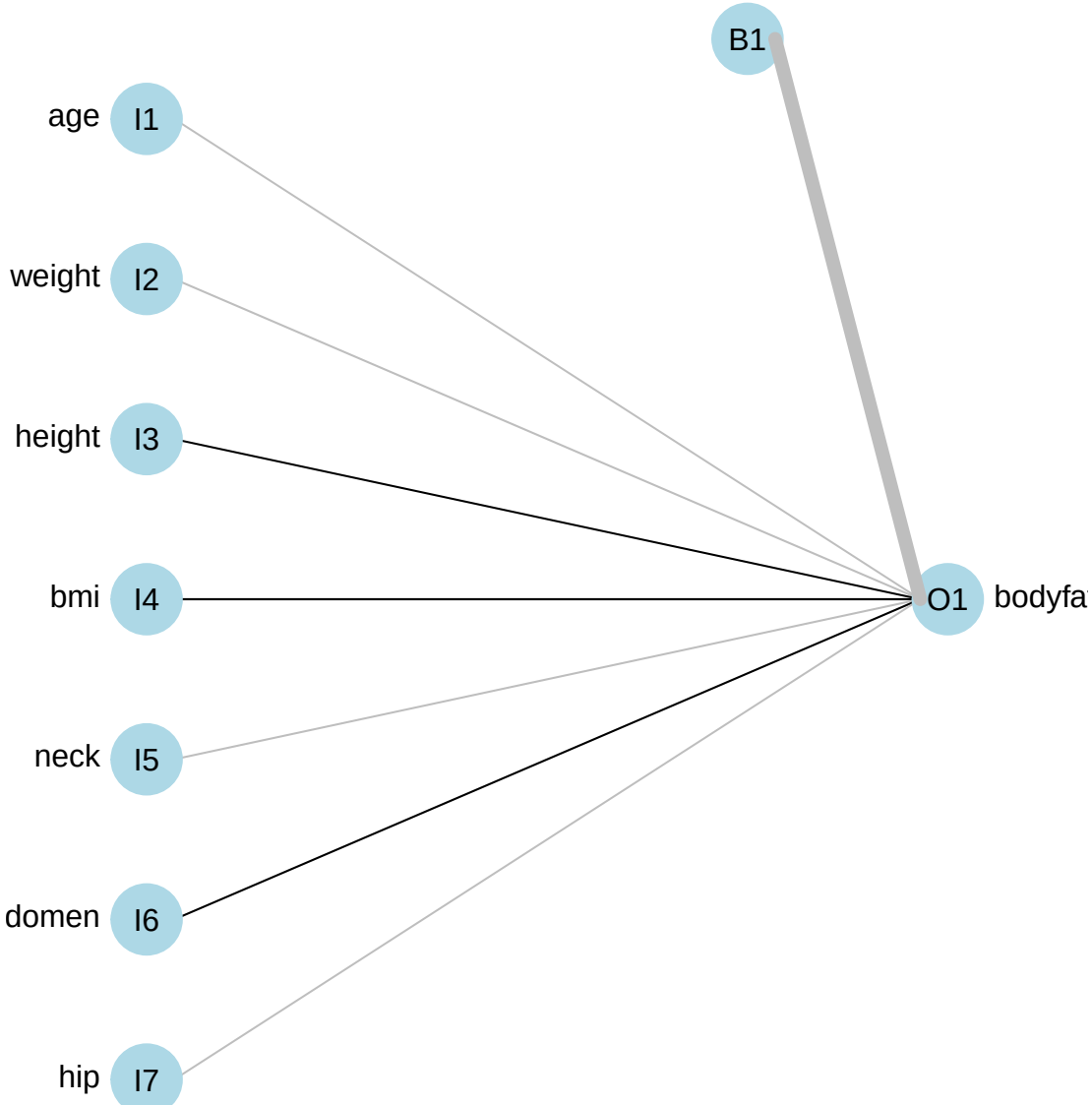
4. $\boldsymbol{\varepsilon} \sim N_n(\mathbf{0}, \sigma^2\boldsymbol{I})$ holds. Here N_n denotes the n-dimensional multivarate normal distribution.

From statistical model to network architecture

We need *new concepts*:

- Our covariates are now presented as *input nodes* in an *input layer*.
- The intercept is presented as a node, and is called a *bias* node. (New meaning to us!)
- The response is presented as one *output node* in an *output layer*.
- The regression coefficients are called *weights* and are often written on the lines (arrows) from the inputs to the output node.
- All lines going into the output node signifies that we multiply the covariate values in the input nodes with the weights (regression coefficients), and then sum. This sum can be sent through a socalled *activition function*. The activation function for linear regression is just the identity function.

```
## # weights:  8
## initial  value 3867668.136237
## iter  10 value 4470.341961
## final  value 4415.453729
## converged
```



The regression function is then rewritten from the linear regression case

$$Y_i = \beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + \cdots + \beta_p x_{ip} + \varepsilon_i \ ,$$

to the neural network case

$$y_1(\boldsymbol{x}_i) = \phi_o(w_0 + w_1 x_{i1} + \cdots + w_p x_{ip}) \ ,$$

where $\phi_o(x) = x$.

- We do not say anything of what is random and fixed, and do not make any assumption distribution of a random variable.
- In the statistics world we would have written $\hat{y}_1(\boldsymbol{x}_i)$ to specify that we are estimating a predicted value of the response for the given covariate value. To be able to distinguish this predicted response from the observed response we use the notation:

$$\hat{y}_1(\boldsymbol{x}_i) = \phi_o(w_0 + w_1 x_{i1} + \cdots + w_p x_{ip})$$

The only difference to our MLR model is then that we would have called the ws $\hat{\beta}$s instead.

Statistical parameter estimation

In multiple linear regression, the parameters $\boldsymbol{\beta}$ are estimated with maximum likelihood and least squares. These two methods give the same estimator when we assume the normal linear regression model. The estimator $\hat{\boldsymbol{\beta}}$ is found by minimizing the RSS for a multiple linear regression model:

$$\begin{aligned}\text{RSS} &= \sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \sum_{i=1}^{n}(y_i - \hat{\beta}_0 - \hat{\beta}_1 x_{i1} - \hat{\beta}_2 x_{i2} - ... - \hat{\beta}_p x_{ip})^2 \\ &= \sum_{i=1}^{n}(y_i - \boldsymbol{x}_i^T\boldsymbol{\beta})^2 = (\boldsymbol{Y} - \boldsymbol{X}\hat{\boldsymbol{\beta}})^T(\boldsymbol{Y} - \boldsymbol{X}\hat{\boldsymbol{\beta}}) \ .\end{aligned}$$

This problem has a solution given on closed form:

$$\hat{\boldsymbol{\beta}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{Y} \ .$$

Neural networks: loss function and gradient descent

We now translate what we did for the regression setup into the neural networks world:

1. Replace the parameters $\boldsymbol{\beta}$ with *network weights* $\boldsymbol{w}$.
2. Replace the RSS in our *training data set* with the following *loss function* (*mean squared error*)

$$J(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_1(\boldsymbol{x}_i))^2 \ ,$$

where $J(\boldsymbol{w})$ indicates that the unknown parameters are the weights $\boldsymbol{w}$.

3. Replace *minimizing* the loss function (RSS) via
   - calculate the derivative of the loss function with respect to each of our parameters
   - solve the $(p+1)$ linear equations.

   $\rightarrow$ *more general minization procedures* that work also when the loss function does not have a closed form.

Finding optimal weights (schematic description)

1. Initialize weights by random numerical values.
2. Calculate the predicted values $\hat{y}_1(\boldsymbol{x}_i)$ for each observation.
3. Calculate the loss function $J(\boldsymbol{w})$, which we want to minimize.
4. Calculate the *gradient of a function* (it gives the *direction of the steepest ascent*) in the $(p+1)$- dimensional space of real numbers (for $p+1$ weights).
5. Update weights in our $p+1$-dimensional in the direction of the *negative of the gradient* we *decreases the loss function* most quickly. The *step length* ("*learning rate*") determines how far we go in an iteration.
6. Iterate steps 2–6 until convergence.
7. Return final weights.

Figures that give good illustration of the optimization problem.

Chollet and Allaire (2018):

- 2.11: SGD down a 1D loess curve
- 2.12: Gradient descent down a 2D loss surface
- 2.13: local and global minimum

(see board)

Finding optimal weights: Algorithm

1. Let $t = 0$ and denote the given initial values for the weights $\boldsymbol{w}^{(t)}$,
2. *propagate* the observations through the network and calculate the predictions $\hat{y}_1(\boldsymbol{x}_i)$
3. calculate the loss function $J(\boldsymbol{w}^{(0)})$,
4. find the gradient (direction) in the $(p+1)$-dimensional space of the weights, and evaluate this at the current weight values $\nabla J(\boldsymbol{w}^{(0)}) = \frac{\partial J}{\partial \boldsymbol{w}}(\boldsymbol{w}^{(0)})$
5. go with a given step length (learning rate) λ in the direction of the negative of the gradient of the loss function to get updated values for the weights

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \lambda \nabla J(\boldsymbol{w}^{(t)})$$

6. Set $t = t + 1$, go to 2. and continue to 6. several times until you arrive at a (local) optimum
7. The final values of the weights in that $(p+1)$ dimensional space are our parameter estimates and your network is *trained* and can be used for prediction on a test set.

Backpropagation

- In neural networks the gradient part of the gradient descent algorithm is implemented efficiently in an algorithm called *backpropagation* (see later).

Here we compare

- the MLR solution with `lm`
- the neural network solution with `nnet` [1]

[1] which in fact improves upon the gradient descent with Hessian information and the BFGS-algorithmBFGS is a quasi-Newton method (also known as a variable metric algorithm), specifically that published simultaneously in 1970 by Broyden, Fletcher, Goldfarb and Shanno. This uses function values and gradients to build up a picture of the surface to be optimized.

Parameter estimation vs. network weights

```
fit = lm(bodyfat ~ age + weight + height + bmi + neck + abdomen +
    hip, data = d.bodyfat)
fitnnet = nnet(bodyfat ~ age + weight + height + bmi + neck + abdomen +
    hip, data = d.bodyfat, linout = TRUE, size = 0, skip = TRUE, maxit = 1000,
    entropy = FALSE)
```

```
## # weights:  8
## initial  value 749757.662821
## iter  10 value 4471.140893
## final  value 4415.453729
## converged
```

```
cbind(fitnnet$wts, fit$coefficients)
```

```
##                       [,1]          [,2]
## (Intercept) -9.748914e+01 -9.748903e+01
## age         -9.607573e-04 -9.607669e-04
## weight      -6.292827e-01 -6.292820e-01
## height       3.974891e-01  3.974884e-01
## bmi          1.785333e+00  1.785330e+00
## neck        -4.945730e-01 -4.945725e-01
## abdomen      8.945188e-01  8.945189e-01
## hip         -1.255551e-01 -1.255549e-01
```

Logistic regression: Diabetes

Aim is to predict if a person has diabetes[2]. The data is from a population of women of Pima Indian heritage in the US, available in the R `MASS` package. The following information is available for each woman:

- `diabetes`: `0`= not present, `1`= present
- `npreg`: number of pregnancies
- `glu`: plasma glucose concentration in an oral glucose tolerance test
- `bp`: diastolic blood pressure (mmHg)
- `skin`: triceps skin fold thickness (mm)
- `bmi`: body mass index (weight in kg/(height in m)^2)
- `ped`: diabetes pedigree function.
- `age`: age in years

[2]Logistic regression is the "hello world" of machine learning.

The statistical model

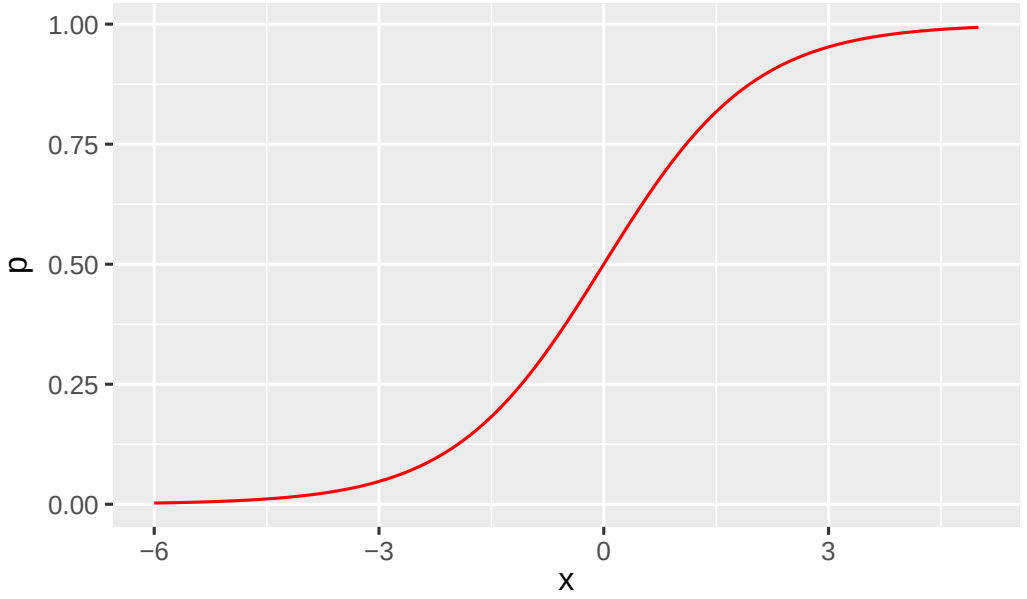
- $i = 1, \ldots, n$ observations in the training set. We will use r (instead of p) to be the number of covariates, to avoid confusion with the probability p.
- The reponse Y_i is coded ($\mathcal{C} = \{1, 0\}$ or {success, failure}) with

$$Y_i = \begin{cases} 1 \text{ with probability } p_i, \\ 0 \text{ with probability } 1 - p_i. \end{cases}$$

In logistic regression we *link* together our covariates $\boldsymbol{x}_i$ with this probability p_i using a *logistic function*.

$$p_i = \frac{\exp(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_r x_{ir})}{1 + \exp(\beta_0 + \beta_1 x_{i1} + \cdots + \beta_r x_{ir})}$$
$$= \frac{1}{1 + \exp(-\beta_0 - \beta_1 x_{i1} - \cdots - \beta_r x_{ir})}$$

This function is S-shaped, and ranges between 0 and 1 (so the p_i is between 0 and 1).



Parameter estimation in the statistical model
(Maximum likelihood)

We assume that pairs of covariates and responses $\{x_i, y_i\}$ are measured independently of each other. Given n such observation pairs, the likelihood function of a logistic regression model can be written as:

$$L(\boldsymbol{\beta}) = \prod_{i=1}^{n} L_i(\boldsymbol{\beta}) = \prod_{i=1}^{n} f(y_i; \boldsymbol{\beta}) = \prod_{i=1}^{n} (p_i)^{y_i} (1 - p_i)^{1-y_i},$$

where $\boldsymbol{\beta} = (\beta_0, \beta_1, \beta_2, \ldots, \beta_r)^T$ enters into p_i.
It is easier (and equivalent) to maximize the log-likelihood:

$$\ln(L(\boldsymbol{\beta})) = l(\boldsymbol{\beta}) = \sum_{i=1}^{n} \Big(y_i \ln p_i + (1 - y_i) \ln(1 - p_i) \Big) \ .$$

- To maximize the log-likelihood function we find the $r+1$ partial derivatives (to form the gradient), and set equal til 0.
- This gives us a set of $r+1$ non-linear equations in the β_js.
- This set of equations does not have a closed form solution.
- These equations are therefore solved numerically, using the *Newton-Raphson algorithm* (or Fisher Scoring):

$$\beta^{(t+1)} = \beta^{(t)} + \boldsymbol{F}(\boldsymbol{\beta}(t))^{-1} s(\beta^{(t)}) \ ,$$

where the gradient of the log-likelihood $\boldsymbol{s}(\boldsymbol{\beta}) = \frac{\partial l}{\partial \boldsymbol{\beta}}$ is called the score vector, and here the new quantity $\boldsymbol{F}(\boldsymbol{\beta}^{(t)})^{-1}$ is called the inverse *expected Fisher information matrix* and is the expected value of the negative of the gradient of the score vector (the negative of the Hessian matrix of the loglikelihood), and also the covariance matrix of the score vector (graphical description on the board).[3]

[3] Observe that we here are maximizing so we are going in the direction of the gradient, not the negative of the direction which is needed when we minimize.

The neural network model: architecture and activation function

- Remember: in the neural network (NN) version of the MLR, we had:

$$y_1(\boldsymbol{x}_i) = \phi_o(w_0 + w_1 x_{i1} + \cdots + w_r x_{ir}) \ ,$$

  where $\phi_o(x) = x$.
- In the NN version of logistic regression we instead have the logistic function as the function ϕ_o, that is

$$\phi_o(x) = \frac{1}{1 + \exp(-x)} \ .$$

This is now referred to as the *sigmoid* activation function and is often denoted $\sigma(x)$. Again, we prefer to use $\hat{y}_1(\boldsymbol{x}_i)$ and get:

$$\hat{y}_1(\boldsymbol{x}_i) = \frac{1}{1 + \exp(-(w_0 + w_1 x_{i1} + \cdots + w_r x_{ir}))} \in (0, 1) \ .$$

Neural networks: loss function and gradient descent

- For parameter estimation we looked at maximizing the log-likelihood of the statistical model.
- For neural networks the negative of the binomial loglikelihood is a scaled version of the *binomial cross-entropy loss*

$$J(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n}(y_i \ln(\hat{y}_1(\boldsymbol{x}_i)) + (1-y_i)\ln(1-\hat{y}_1(\boldsymbol{x}_i)) \ .$$

- Optimization is done also with gradient descent, but we need the chain rule (due to the activation function) to get the partial derivatives for the gradient direction.
- Same algorithm as for the linear case is also applicable now, with the modification to the activation and loss function given here.

Parameter estimation vs. network weights

```r
fitlogist = glm(diabetes ~ npreg + glu + bp + skin + bmi + ped + age,
    data = train, family = binomial(link = "logit"))
summary(fitlogist)
```

```
##
## Call:
## glm(formula = diabetes ~ npreg + glu + bp + skin + bmi + ped +
##     age, family = binomial(link = "logit"), data = train)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.9830  -0.6773  -0.3681   0.6439   2.3154
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -9.773062   1.770386  -5.520 3.38e-08 ***
## npreg        0.103183   0.064694   1.595  0.11073
## glu          0.032117   0.006787   4.732 2.22e-06 ***
## bp          -0.004768   0.018541  -0.257  0.79707
## skin        -0.001917   0.022500  -0.085  0.93211
## bmi          0.083624   0.042827   1.953  0.05087 .
## ped          1.820410   0.665514   2.735  0.00623 **
## age          0.041184   0.022091   1.864  0.06228 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 256.41  on 199  degrees of freedom
## Residual deviance: 178.39  on 192  degrees of freedom
## AIC: 194.39
##
## Number of Fisher Scoring iterations: 5
```

```r
set.seed(787879)
library(nnet)
fitnnet = nnet(diabetes ~ npreg + glu + bp + skin + bmi + ped + age,
    data = train, linout = FALSE, size = 0, skip = TRUE, maxit = 1000,
    entropy = TRUE, Wts = fitlogist$coefficients + rnorm(8, 0, 0.1))
```

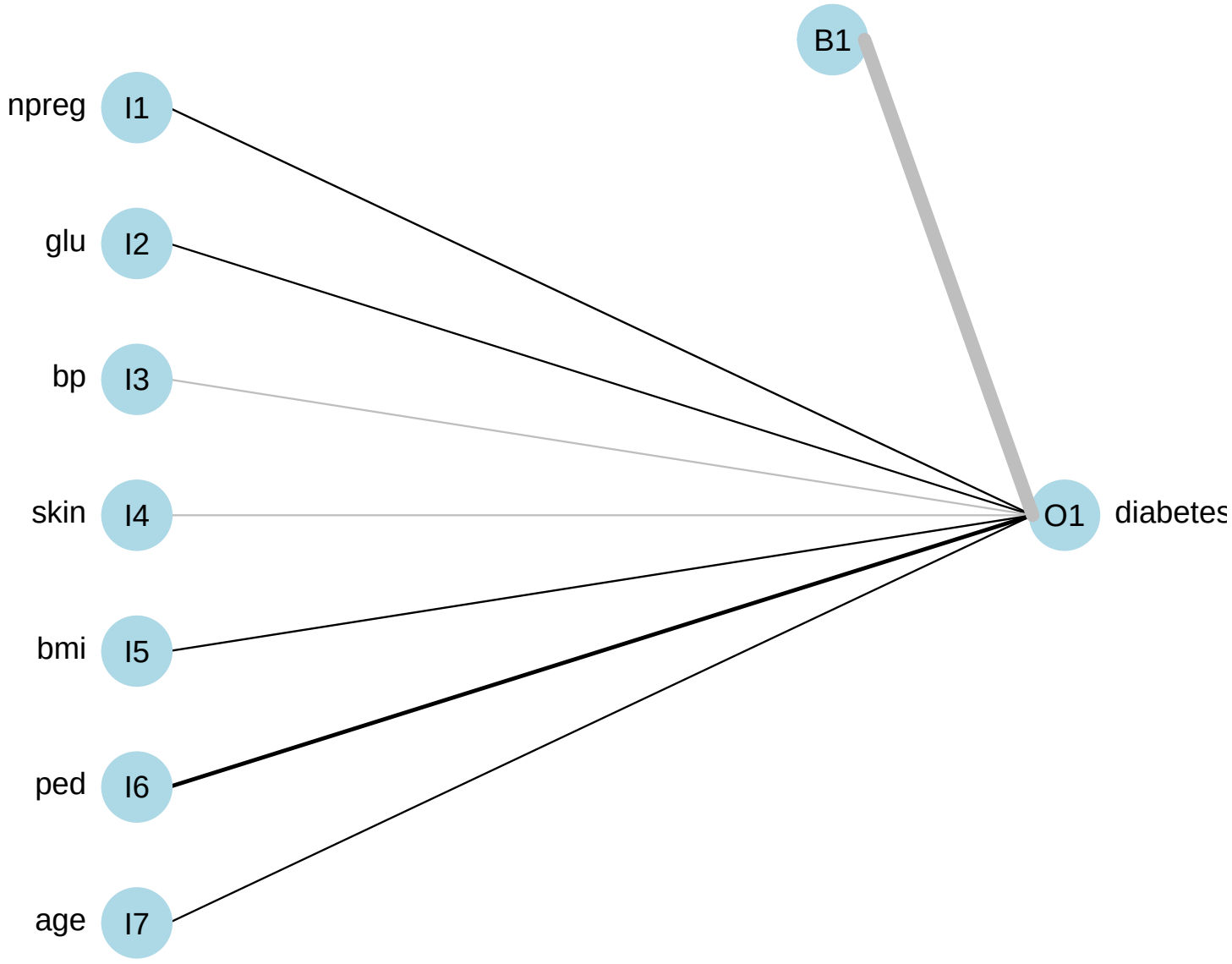
```
## # weights:  8
## initial  value 213.575955
## iter  10 value 89.511044
## final  value 89.195333
## converged
```

```r
# entropy=TRUE because default is least squares
cbind(fitnnet$wts, fitlogist$coefficients)
```

```
##                      [,1]         [,2]
## (Intercept) -9.773046277 -9.773061533
## npreg        0.103183171  0.103183427
## glu          0.032116832  0.032116823
## bp          -0.004767678 -0.004767542
## skin        -0.001917105 -0.001916632
## bmi          0.083624151  0.083623912
## ped          1.820397792  1.820410367
## age          0.041183744  0.041183529
```

In the `nnet` R package a slightly different version is used, `entropy=maximum conditional likelihood` which is half the deviance for the logistic regression model.

```
plotnet(fitnnet)
```



But, there may also exist local minima.

```
set.seed(123)
fitnnet = nnet(diabetes ~ npreg + glu + bp + skin + bmi + ped + age,
    data = train, linout = FALSE, size = 0, skip = TRUE, maxit = 10000,
    entropy = TRUE, Wts = fitlogist$coefficients + rnorm(8, 0, 1))
```

```
## # weights:  8
## initial  value 24315.298582
## final  value 12526.062906
## converged
```

```
cbind(fitnnet$wts, fitlogist$coefficients)
```

```
##                     [,1]         [,2]
## (Intercept)   -36.733537 -9.773061533
## npreg         -77.126994  0.103183427
## glu         -2984.409175  0.032116823
## bp          -1835.934259 -0.004767542
## skin         -718.072629 -0.001916632
## bmi          -818.561311  0.083623912
## ped            -8.687473  1.820410367
## age          -773.023878  0.041183529
```

Multiclass regression

Which type of iris species?

The `iris` flower data set was introduced by the British statistician and biologist Ronald Fisher in 1936, and we studied that in module 4 on classification.

- Three plant species {setosa, virginica, versicolor} (50 observation of each), and
- Four features: `Sepal.Length`, `Sepal.Width`, `Petal.Length` and `Petal.Width`.

The aim is to predict the species of an iris plant.

Statistical model

& We only briefly mentioned this method in module 4 (there our focus was on LDA and QDA when we had more than two classes)[4].

Assumptions:

- We have independent observation pairs $(Y_i, \boldsymbol{x}_i)$, where the covariate vector $\boldsymbol{x}_i$ consists of the same measurements for each response category (that is, not different covariate types that are measured for each response category.
- Each observation can only belong to one response class, $Y_i = c$ for $c = 1, \ldots, C$.

Dummy variable coding of the response in a C-dimensional vector: $\boldsymbol{y}_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$ with a value of 1 in the c^{th} element of $\boldsymbol{y}_i$ if the class is c.

[4]The full theoretical treatment is given in the statistics course TMA4315 Generalized linear models, Module 6 on categorical regression (nominal response)

- Probabilities that the response is category c for subject i

$$p_{ic} = P(Y_i = c) \ ,$$

  where $\sum_{c=1}^{C} p_{ic} = 1$. Note that in statistics we don't include p_{i1} in the modelling because $p_{i1} = 1 - \sum_{c=2}^{C} p_{ic}$, and $\beta_1 = 0$.
- A generalization of the logistic regression model (for two classes) can be used. The model we fit is:

$$p_{ic} = P(Y_i = c) = \frac{\exp(\boldsymbol{x}_i^T \boldsymbol{\beta}_c)}{1 + \sum_{s=2}^{C} \exp(\boldsymbol{x}_i^T \boldsymbol{\beta}_s)}$$

An observation is classified to the class with the highest probability, $\underset{c}{\text{argmax}}(p_{ic})$.

Parameter estimation in the statistical model

- The likelihood of the multinomial regression model can be written as

$$\ln(L(\boldsymbol{\beta}) \propto \sum_{i=1}^{n} \sum_{c=1}^{C} y_{ic} \ln(p_{ic}) \ ,$$

  where $p_{iC} = 1 - p_{i1} - p_{i2} - \cdots - p_{i,C-1}$, and the regression parameters enter via the p_{ic}s.
- Parameter estimation is done in the same way as for the logistic regression, with the Fisher scoring algorithm (with score vector and Fisher information matrix).
- However (and this might be confusing), an efficient function in R also relies on neural networks for optimization (see below).

Neural network architecture and activation function

- The neural network uses the dummy variable coding of the responses, called *one-hot coding*.
- Builds an output layer with C nodes and corresponding 0/1 targets (responses).

Softmax

- The activation function for the ouput layer is called *softmax*. For each class $c = 1, \ldots, C$ it is given as

$$\hat{y}_c(\boldsymbol{x}_i) = \frac{\exp(\boldsymbol{x}_i^T \boldsymbol{w}_c)}{\sum_{s=1}^{C} \exp(\boldsymbol{x}_i^T \boldsymbol{w}_s)} \ ,$$

  where each $\boldsymbol{w}_s$ is a $r+1$ dimensional vector of weights.
- Note: there is some redundancy here, since $\sum_{c=1}^{C} \hat{y}_c(\boldsymbol{x}_i) = 1$, so we could have had $C-1$ output nodes, but this is not done.
- The focus of neural networks is not to interpret the weights, and there is no need to assume full rank of a matrix with output nodes.

Q: How many parameters are we estimating?

Neural networks: loss function and gradient descent

- For parameter estimation we looked at maximizing the log-likelihood of the statistical model. For neural networks the negative of the multinomial loglikelihood is a scaled version of the *categorical cross-entropy loss*

$$J(\boldsymbol{w}) = -\frac{1}{n}\sum_{i=1}^{n}\frac{1}{C}\sum_{c=1}^{C}(y_{ic}\ln(\hat{y}_c(\boldsymbol{x}_i)) \ .$$

- The optimization is done using gradient descent, with minor changes from what was done for the logistic regression due to the added sum and the small change in the activation function.

Fitting multinomial regression vs a neural network

First select a training sample

```
library(nnet)
set.seed(123)
train = sample(1:150, 50)
iris_train = ird[train, ]
iris_test = ird[-train, ]
```

Then fit the `nnet()` (by default using the softmax activation function)

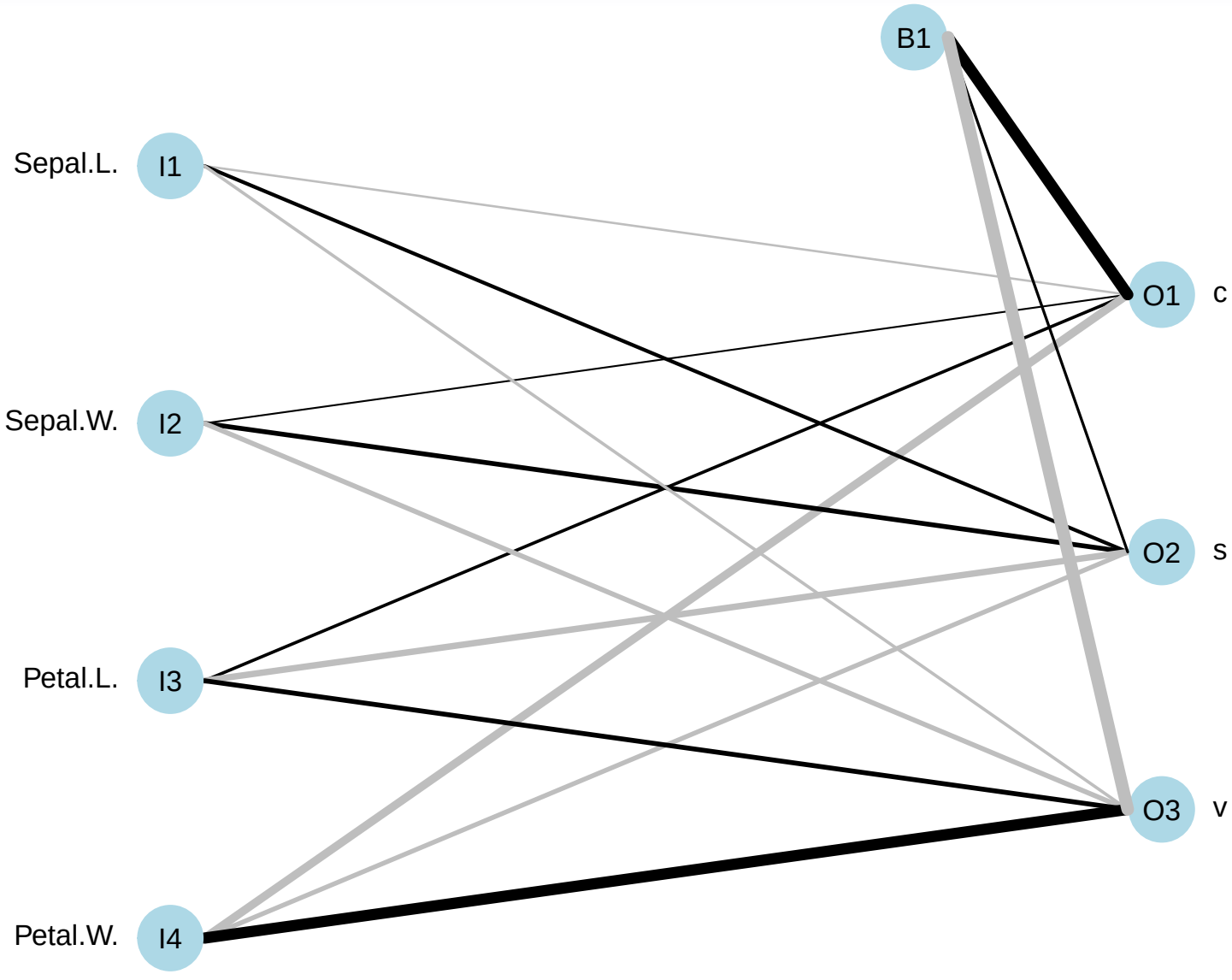
```
iris.nnet <- nnet(species ~ ., data = ird, subset = train, size = 0,
    skip = TRUE, rang = 0.1, decay = 5e-04, maxit = 200)
```

```
## # weights:  15
## initial  value 56.193086
## iter  10 value 4.700163
## iter  20 value 1.483907
## iter  30 value 1.369780
## iter  40 value 1.316379
## iter  50 value 1.300036
## iter  60 value 1.296889
## iter  70 value 1.296349
## iter  80 value 1.296154
## iter  90 value 1.295839
## iter 100 value 1.295822
## iter 110 value 1.295808
## iter 120 value 1.295798
## final  value 1.295798
```

How many weights have been estimated? What does the graph look like?

```
summary(iris.nnet)
```

```
## a 4-0-3 network with 15 weights
## options were - skip-layer connections  softmax modelling  decay=5e-04
##  b->o1 i1->o1 i2->o1 i3->o1 i4->o1
##  17.10  -1.04   0.25   2.68 -11.43
##  b->o2 i1->o2 i2->o2 i3->o2 i4->o2
##   2.03   3.64   5.52  -8.30  -5.39
##  b->o3 i1->o3 i2->o3 i3->o3 i4->o3
## -19.13  -2.60  -5.78   5.63  16.82
```


B1
Sepal.L.
I1
Sepal.W.
I2
Petal.L.
I3
Petal.W.
I4
O1
c
O2
s
O3
v

Fitting the multinomial regression. This is also done with nnet, but using a wrapper `multinom`, which uses the `entropy` activation function.

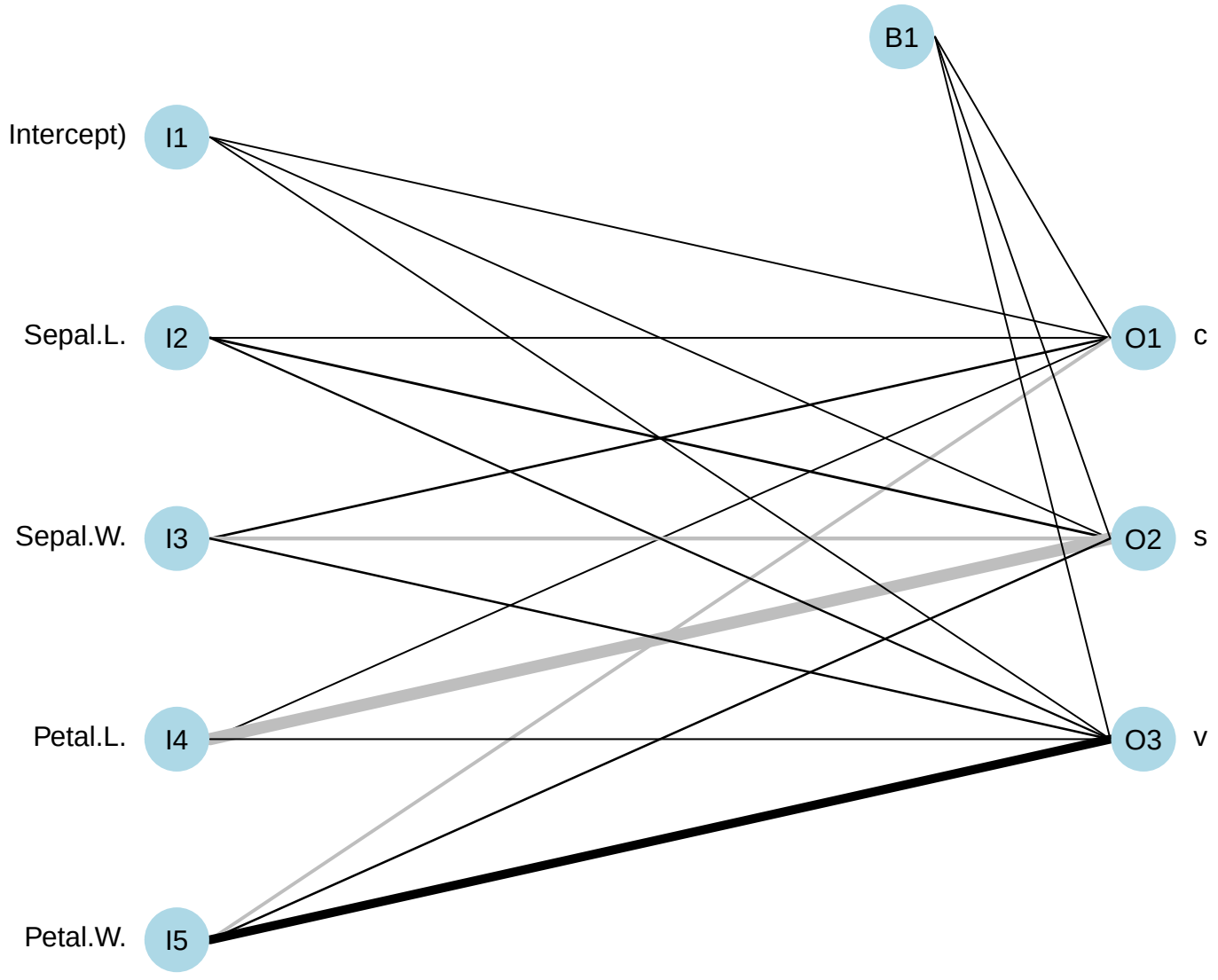
```
library(caret)
fit=multinom(species~ -1 + .,family=multinomial, data=iris_train)
```

```
## # weights:  15 (8 variable)
## initial  value 54.930614
## iter  10 value 4.353139
## iter  20 value 0.139411
## iter  30 value 0.065218
## iter  40 value 0.056419
## iter  50 value 0.045548
## iter  60 value 0.020867
## iter  70 value 0.016116
## iter  80 value 0.012952
## iter  90 value 0.012787
## iter 100 value 0.009090
## final  value 0.009090
## stopped after 100 iterations
```

```
summary(fit)
```

```
## Call:
## multinom(formula = species ~ -1 + ., data = iris_train, family = multinomial)
##
## Coefficients:
##    Sepal.L.  Sepal.W.  Petal.L.  Petal.W.
## s  -5.91976  21.30247 -12.52073 -2.774511
```

Problem: `multinom()` seems to fit an intercept *plus* an offset note (B):



The performance of multinomial regression vs nnet

```
testclass = predict(fit, new = iris_test)
confusionMatrix(data = testclass, reference = iris_test$species)$table
```

```
##           Reference
## Prediction  c  s  v
##          c 38  0  4
##          s  0 28  0
##          v  0  0 30
```

```
table(predict(iris.nnet, iris_test, type = "class"), iris_test$species)
```

```
##
##      c  s  v
##   c 37  0  4
##   s  0 28  0
##   v  1  0 30
```

For more on multinomial regression with R, check here.

Summing up

- Fitting a *multiple linear regression*
  - can be achived by building a neural network with one input layer and one node in the output layer,
  - linear activation function,
  - and mean squared loss,
- *Classification with two classes* can be performed using logistic regression, and this
  - by building a neural network with one input layer and one node in the output layer,
  - sigmoid activation function,
  - and binary cross-entropy loss.
  - Remember: this will only give linear boundaries between the classes (in the output space).

- Classification with C classes, $C > 2$, can be performed using multinomial regression,
  - by building a neural network with one input layer and C nodes in the output layer,
  - softmax activation function,
  - and categorical cross-entropy loss.
  - Also here: this will only give linear boundaries between the classes (in the output space).

- We have seen that parameters (weights) can be found using gradient descent algorithms:
  - activation function "must" be differentiable
  - step length (learning rate) must be set
- But, we have now only looked at linear models — which give linar boundaries in the classification case — now we need to move on to allow for non-linearities!

Feedforward networks

In feedforward networks we have only connections (weights) forward in the network, and no feedback connctions that sends the output of the model back into the network. The word *multi-layer perceptron* (MLP) and *sequentially layered* networks are also used

The examples of MLR, logistic regression and multiclass regression are examples of feedfeedward networks without any socalled *hidden layers* (between the input and output layers).

We may have no hidden layer, one (to be studied next), or many.

The number of hidden layers is called the *depth* of the network, and the number of nodes in a layer is called the *width* of the layer.

The idea of using many layers of many nodes is inspired from neuroscience, but today we don't have the goal to model the brain — but instead to approximate function to perform statistical generalizations and maybe also insight into the problem at hand.

Now we will see how adding *hidden layers* with *non-linear activation functions* between the input and output layer will make nonlinear statistical models.

The single hidden layer feedforward network

The word neuron and node can be used interchangeably. We stick with greek letters α and β for parameters, but call them weights. We use the following notation:

1. Inputs (input layer nodes), $j = 1, \ldots p$: $x_1, x_2, \ldots, x_p$, or as a vector $\boldsymbol{x}$.
2. The nodes in the hidden layer, $m = 1, \ldots, M$: $z_1, z_2, \ldots, z_m$, or as vector $\boldsymbol{z}$, and the hidden layer activation function ϕ_h.

$$z_m(\boldsymbol{x}) = \phi_h(\alpha_{0m} + \sum_{j=1}^{p} \alpha_{jm} x_j)$$

where α_{jm} is the weight from input j to hidden node m, and α_{0m} is the bias term for the mth hidden node. The hidden nodes can be thought of as *latent variables*.

3. The node(s) in the output layer, $c = 1, \ldots C$: $y_1, y_2, \ldots, y_C$, or as vector $\boldsymbol{y}$, and output layer activation function ϕ_o.

$$\hat{y}_c(\boldsymbol{x}) = \phi_o(\beta_{0c} + \sum_{m=1}^{M} \beta_{mc} z_m(\boldsymbol{x}))$$

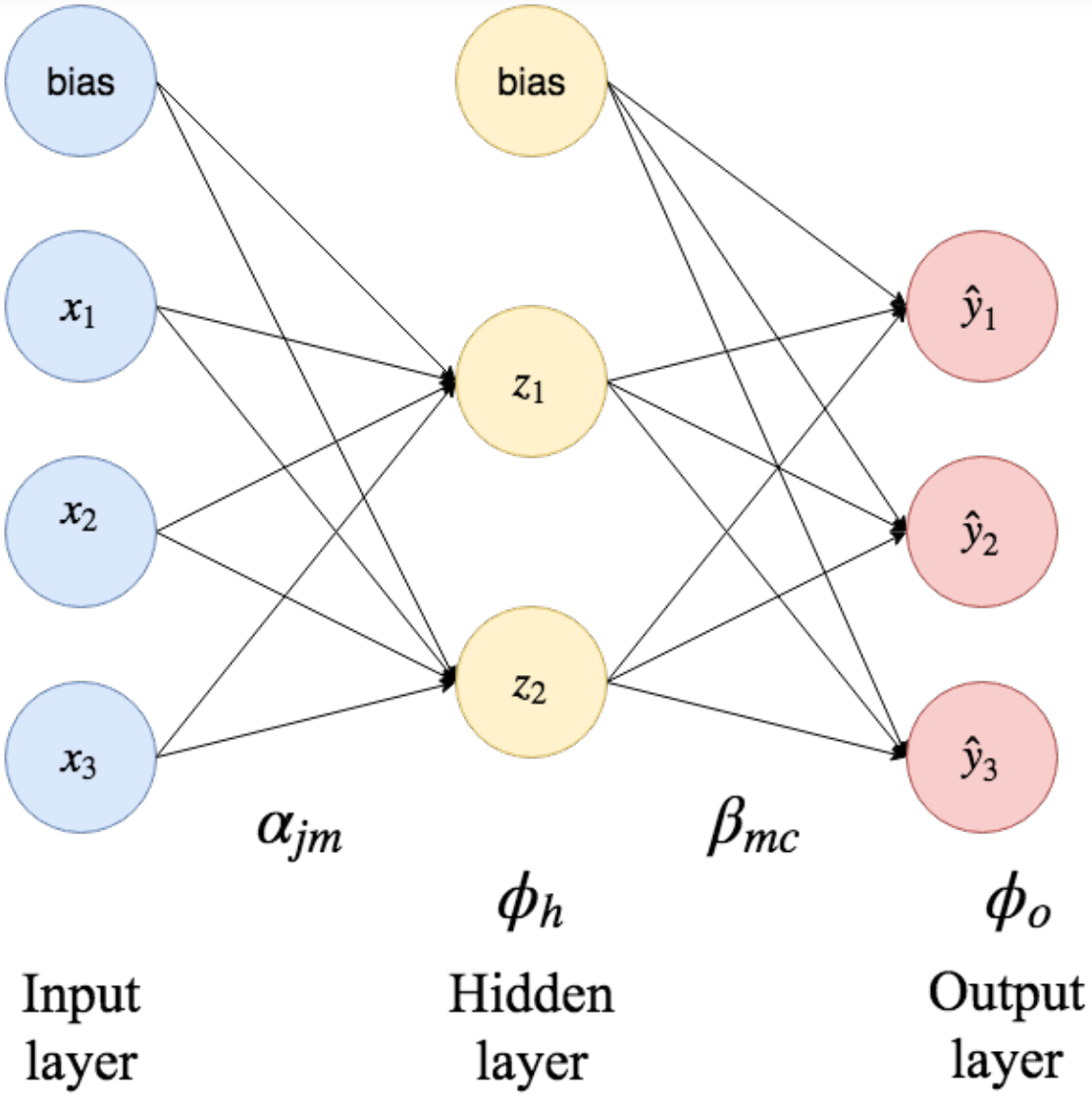
where β_{mc} is from hidden neuron m to ouput node c, and β_{0c} is the bias term for the cth output node.

To sum up: $c = 1, \ldots C$: $y_1, y_2, \ldots, y_C$, where y_c is given as

$$\hat{y}_c(\boldsymbol{x}) = \phi_o(\beta_{0c} + \sum_{m=1}^{M} \beta_{mc} z_m) = \phi_o(\beta_{0c} + \sum_{m=1}^{M} \beta_{mc} \phi_h(\alpha_{0m} + \sum_{j=1}^{p} \alpha_{jm} x_j))$$

Hands on:

- Identify p, M, C in the network figure below, and relate that to the $y_c(\boldsymbol{x})$ equation.
- How many parameters need to be estimated for this network?
- What decides the value of p and C?
- What is the connection between problem, ϕ_o and C?


bias
bias
x_1
x_2
x_3
z_1
z_2
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
α_{jm}
β_{mc}
ϕ_h
ϕ_o
Input layer
Hidden layer
Output layer

Special case with linear activation function for the hidden layer

$$\hat{y}_c(\boldsymbol{x}) = \phi_o(\beta_{0c} + \sum_{m=1}^{M} \beta_{mc} z_m) = \phi_o(\beta_{0c} + \sum_{m=1}^{M} \beta_{mc} \phi_h(\alpha_{0m} + \sum_{j=1}^{p} \alpha_{jm} x_j))$$

Here we assume that $\phi_h(z) = z$, called linear or identity activiation:

$$\hat{y}_c(\boldsymbol{x}) = \phi_o(\beta_{0c} + \sum_{m=1}^{M} \beta_{mc}(\alpha_{0m} + \sum_{j=1}^{p} \alpha_{jm} x_j))$$

Q: Does this look like something you have seen before?

For regression (linear output activation also) we have looked at a similar model in Module 6: principal component regression. Then the weights α_{jm} were not estimated, but were defined to be the eigenvectors corresponding to the of the mth largest eigenvalue of the estimated covariance matrix of the covariates $\boldsymbol{x}_i$. Therefore we may think of the hidden nodes as *latent variables*. In PCR only the βs were estimated using the responses (and the latent variables).

In Module 6 we also touched upon Partial least squares, where the latent variables were found with the help of the response y. Here both the αs and βs were estimated.

In Module 4 we only considered "ordinary logistic regression", but we could have used the principal components in place of the original covariates — which then could have been depictured as a feedforward network with one hidden layer, but where only the βs were estimated.

Universal approximation property

We may think of the goal of a feedforward network to approximate some function f, mapping our input vector $\boldsymbol{x}$ to an output value $\boldsymbol{y}$ (as we saw with regression and classification).
What type of mathematical function can a feedforward neural network with one hidden layer and linear output activation represent?

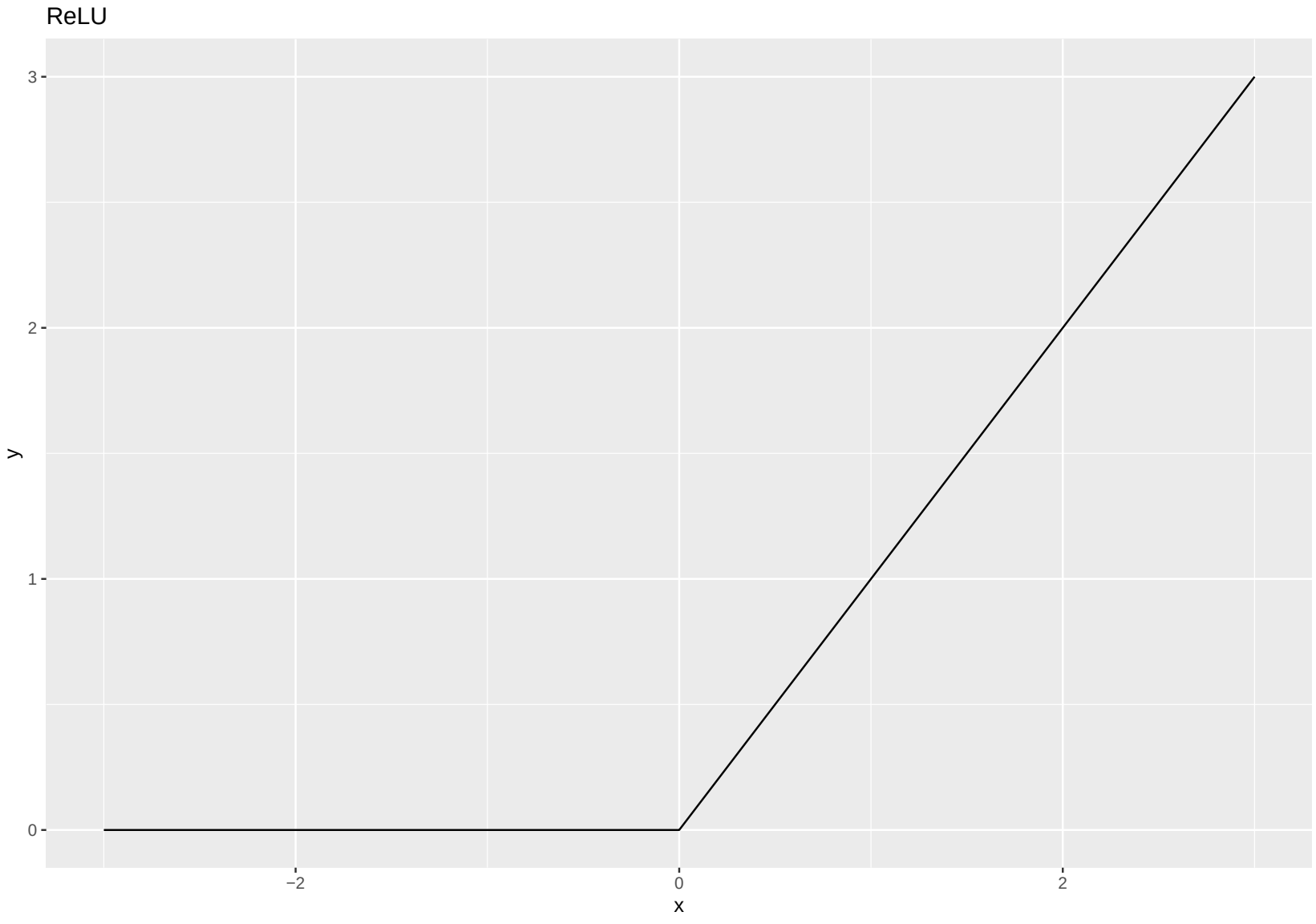
According to Goodfellow, Bengio, and Courville (2016), page 198: The *universal approximation theorem* says that a feedforward network with

- a *linear output layer*
- at least one hidden layer with a "squashing" activation function and "enough" hidden units

can approximate any (Borel measurable) function from one finite-dimensional space (our input layer) to another (our output layer) with any desired non-zero amount of error.

The theorem has been shown both for the sigmoid (logistic) and ReLU activation functions in the hidden layer.

The rectified linear unit ReLU $\phi_h(a) = \max(0, a)$



Earlier, the default activation function for the hiddan layer was the sigmoid, but now the ReLU is default activiation function to be used in the hidden layer(s) of a feedforward network — in particular when more than one hidden layer is used. The ReLU function is piecewise linear, but in total non-linear.

Even though a large feedforward network with one hidden layer may be able to represent a desired function, we may not be able to estimate the parameters of the function,

- we may choose a too many or too few nodes in the hidden layer
- our optimization routine may fail
- we may overfit/underfit the training data

Therefore, sometimes networks with more than one hidden layer is used — with fewer total number of nodes but more layers. A network with many hidden layers is called a *deep network*.

In module 7 we looked at additive models of "complex" functions (splines) of one covariate each.

Now we look at many rather simple non-linear function of linear combinations of covariates, and non-linear functions of non-linear functions of linear combinations of covariates.

Q: Is one better than the other when it comes to interpretation, and to prediction?

The `nnet` R package

In order to not make this too complex (since we only have one week to work with this module), we will use the `nnet` R package by Brian Ripley (instead of the currently very popular `keras` package for deep learning — however, we will also present the `keras` package for completeness).

`nnet` fits one hidden layer with sigmoid activiation function. The implementation is not gradient descent, but instead BFGS using `optim`.

Description: Fit single-hidden-layer neural network, possibly with skip-layer connections.

Usage: (for formula class): `nnet(formula, data, weights, ..., subset, na.action, contrasts = NULL)`

`nnet(x, y, ...)`

Some arguments to `nnet()`

- formula, or x and y
- x= input variables, matrix or data frame
- y= response (target) values, if factor=classification
- size: number of units in the hidden layer. Can be zero if there are skip-layer units.
- data: the dataset to be used
- subset: index of entries in data that is the training sample

If the response in formula is a factor, an appropriate classification network is constructed; this has one output and entropy fit if the number of levels is two, and a number of outputs equal to the number of classes and a softmax output stage for more levels.

- linout: switch for linear output units. Default logistic output units.
- entropy: switch for entropy (= maximum conditional likelihood) fitting. Default by least-squares.
- softmax switch for softmax (log-linear model) and maximum conditional likelihood fitting. linout, entropy, softmax and censored are mutually exclusive.
- censored: A variant on softmax, in which non-zero targets mean possible classes. Thus for softmax a row of (0, 1, 1) means one example each of classes 2 and 3, but for censored it means one example whose class is only known to be 2 or 3.
- skip: switch to add skip-layer connections from input to output.
- decay: parameter for weight decay. Default 0.
- maxit: maximum number of iterations. Default 100.
- MaxNWts: The maximum allowable number of weights. There is no intrinsic limit in the code, but increasing MaxNWts will probably allow fits that are very slow and time-consuming.

Examples

Boston house prices

The objective here is to predict the median price of homes in a given Boston suburb in the mid-1970s, and 10 input variables are given. This data set is both available in the `MASS` and `keras` R package.

Preparing the data

- Few data points: only 506, split between 404 training samples and 102 test samples (this split already done in the `keras` library)
- Each feature in the input data (for example, the crime rate) has a different scale, some values are proportions, which take values between 0 and 1; others take values between 1 and 12, others between 0 and 100, and so on.
- The targets are the median values of owner-occupied homes, in thousands of dollars.

```
library(keras)
dataset <- dataset_boston_housing()
c(c(train_data, train_targets), c(test_data, test_targets)) %<-%
str(train_targets)
```

```
##  num [1:404(1d)] 15.2 42.3 50 21.1 17.7 18.5 11.3 15.6 15.6 1
```

```
head(train_data)
```

```
##          [,1] [,2]  [,3] [,4]  [,5]  [,6]  [,7]    [,8] [,9] [,
## [1,] 1.23247  0.0  8.14    0 0.538 6.142  91.7 3.9769    4
## [2,] 0.02177 82.5  2.03    0 0.415 7.610  15.7 6.2700    2
## [3,] 4.89822  0.0 18.10    0 0.631 4.970 100.0 1.3325   24
## [4,] 0.03961  0.0  5.19    0 0.515 6.037  34.5 5.9853    5
## [5,] 3.69311  0.0 18.10    0 0.713 6.376  88.4 2.5671   24
## [6,] 0.28392  0.0  7.38    0 0.493 5.708  74.3 4.7211    5
##       [,12] [,13]
## [1,] 396.90 18.72
## [2,] 395.38  3.11
## [3,] 375.52  3.26
## [4,] 396.90  8.01
```

A widespread best practice to deal with such data is to do feature-wise normalization. This is to make the optimization easier with gradient based methods.

```
org_train = train_data
mean <- apply(train_data, 2, mean)
std <- apply(train_data, 2, sd)
train_data <- scale(train_data, center = mean, scale = std)
test_data <- scale(test_data, center = mean, scale = std)
```

Note that the quantities used for normalizing the test data are computed using the training data. You should never use in your workflow any quantity computed on the test data, even for something as simple as data normalization. This also means that we need to do this standardization again (from scratch) if we need to do cross-validation.

Just checking out one hidden layer with 5 units to get going.

```
library(nnet)
fit5 <- nnet(train_targets ~ ., data = train_data, size = 5, lin
    maxit = 1000)
```

```
## # weights:  76
## initial  value 245152.741465
## iter  10 value 12420.525228
## iter  20 value 6902.467151
## iter  30 value 5723.819800
## iter  40 value 5177.095194
## iter  50 value 4689.188029
## iter  60 value 4451.205619
## iter  70 value 4357.164555
## iter  80 value 4254.337102
## iter  90 value 4118.662278
## iter 100 value 4026.432205
## iter 110 value 3918.047699
## iter 120 value 3738.715855
## iter 130 value 3406.126780
## iter 140 value 3168.917490
```

Now, use cross-validation to find the best number of hidden nodes — but that took some time, so only results shown below. Remember the scaling need to be done within the loop.

```
grid = c(5, 10, 15, 20, 25, 30, 50)

k <- 4
set.seed(123)
indices <- sample(1:nrow(train_data))
folds <- cut(indices, breaks = k, labels = FALSE)
# have now assigned the traning data to 4 diffeent folds all of
# the same size (101)

resmat = matrix(NA, ncol = k, nrow = length(grid))
for (j in 1:k) {
    thistrain = (1:dim(train_data)[1])[folds != j]
    thisvalid = (1:dim(train_data)[1])[folds == j]
    mean <- apply(org_train[thistrain, ], 2, mean)
    std <- apply(org_train[thistrain, ], 2, sd)
    new <- scale(org_train, center = mean, scale = std)
    for (i in 1:length(grid)) {
        thissize = grid[i]
```

The best model here was the model with 50 nodes, the largest model we tried. Fitting that model on the full training set and testing on the test set

```
library(nnet)
fit50 <- nnet(train_targets ~ ., data = train_data, size = 50, l
    maxit = 5000)
```

```
## # weights:  751
## initial  value 270057.408487
## iter  10 value 7120.629568
## iter  20 value 1912.248122
## iter  30 value 1045.291511
## iter  40 value 654.075165
## iter  50 value 483.020417
## iter  60 value 381.456615
## iter  70 value 303.378804
## iter  80 value 206.528830
## iter  90 value 117.599100
## iter 100 value 75.531273
## iter 110 value 48.326269
## iter 120 value 31.066476
```

Using `nnet`

- only one hidden layer available
- only sigmoid hidden layer activation function
- optimization done with an optimization method where the learning rate is automatically calculated
- weight decay (L_2 regularization as for the ridge regression) is available (see below)
- optimization is supposed to be done until convergence
- the possible choice of hyper parameter is the number of notes in the hidden layer

We suggest you use `nnet` in Compulsory exercise 2, Problem 3.

Neural network parts

We now focus on the different elements of neural networks.

Illustration drawn in class, motivated from Figure 1.9/3.1 from Chollet and Allaire (2018).

Model

Output layer activation

These choices have been guided by solutions in statistics (multiple linear regression, logistic regression, multiclass regression)

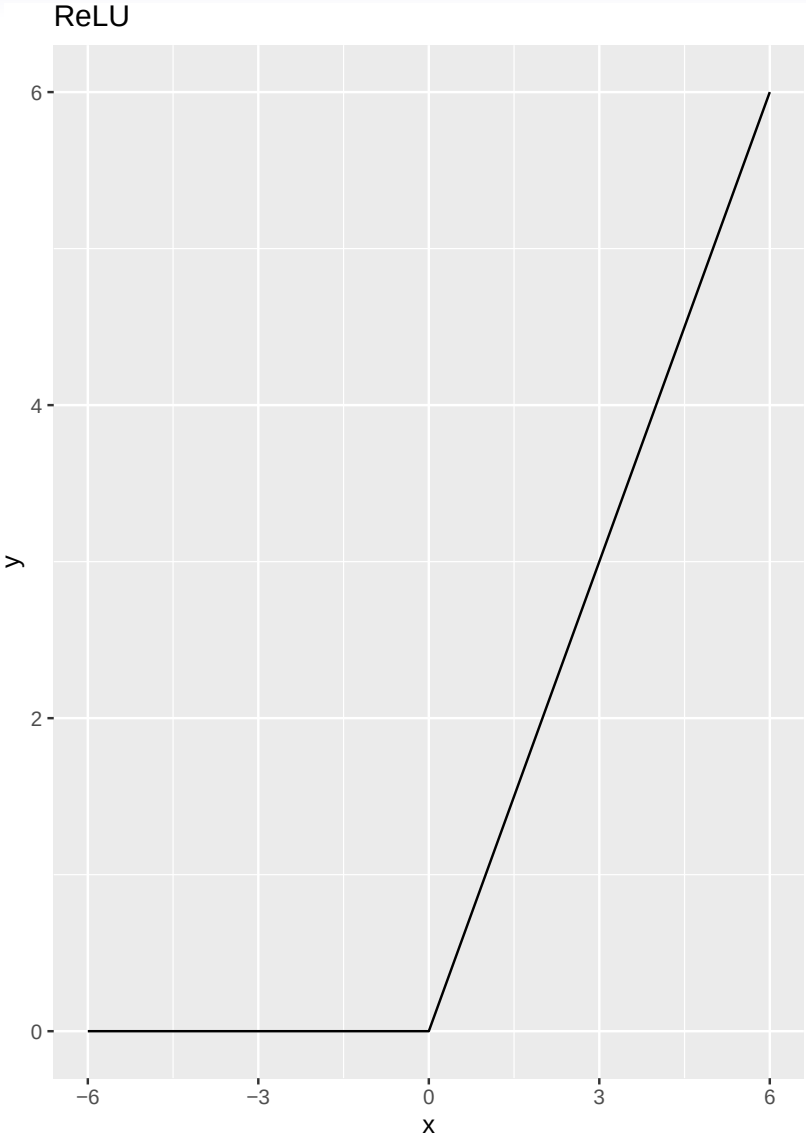
- Linear: for regression problems
- Sigmoid: for two-class classification problems
- Softmax: for more than two classes classification problems

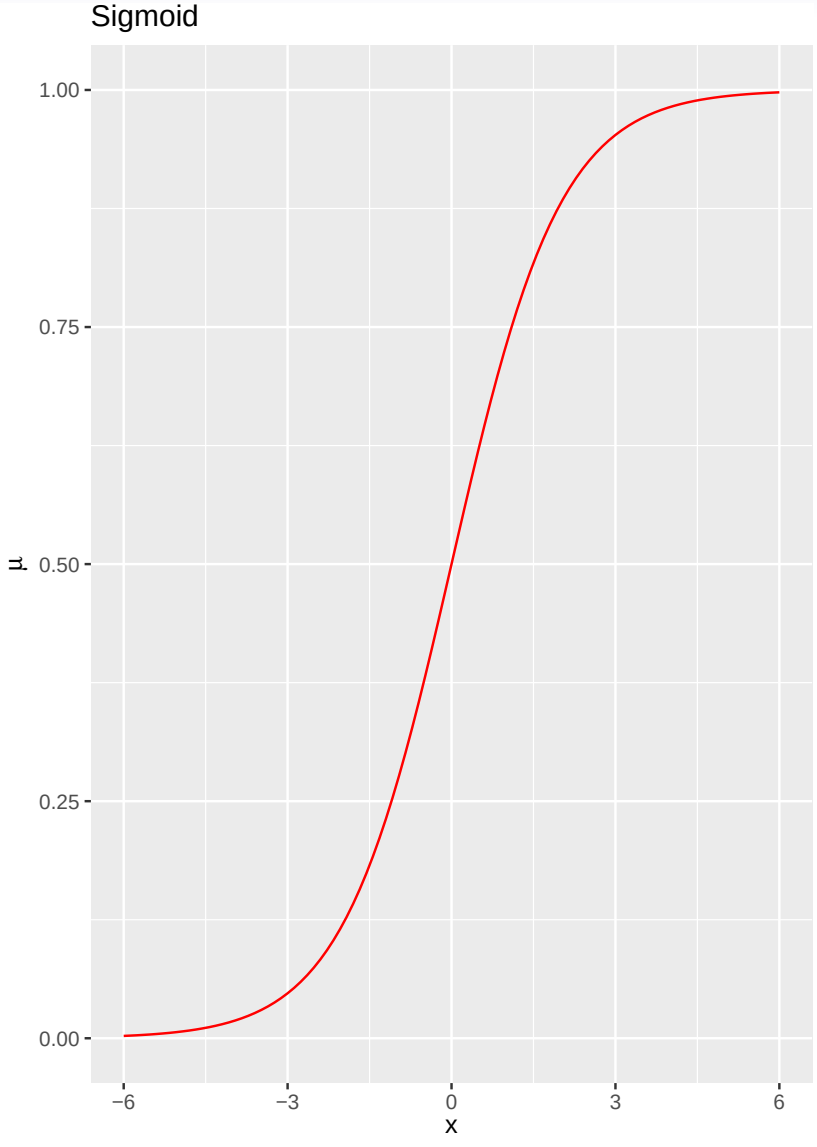
Remark: it is important that the output activation is matched with an appropriate loss function.

Hidden layer activation

Q: Why can we not just use linar activation function in all hidden layers?

A: Then each layer would only be able to do linear transformations of the input data and a deep stack of linear layers would still implement a linear operation. The activation functions sigmoid and relu add non-linearity to the model. And, the universial approximation property is dependent on a squashing type activation function.


ReLU
6
4
2
0
y
−6
−3
0
3
6
x


Sigmoid
1.00
0.75
0.50
0.25
0.00
μ
−6
−3
0
3
6
x

- Sigmoid: $g(a) = 1/(1 + \exp(-a))$. The previous standard choice. These units were found to saturate at a high value when the input was very positive and at a low value when the input was very negative, and that they were only strongly sensitive when the input was close to 0. This saturation lead to problems in the gradient descent routines.

However, for our Compulsory exercise 2, Problem 3, there should be quite ok to use sigmoid hidden layer activation in the `nnet` R package (default hidden layer activation is sigmoid, and that can not be changed).

- ReLU: $\phi_h(a) = \max(0, a)$. The standard choice for deep networks (many hidden layers and many nodes) today.

The function is piecewise linear, but in total non-linear. It has shown to be easy to use with gradient descent — even though the function is not differentiable at 0. As we will touch upon later, we don't expect to train a network until the gradient is 0. The derivative from the left at 0 is 0, and the derivative from the right is 1. See Goodfellow, Bengio, and Courville (2016) page 192 for a discussion on this topic.

When initializing the parameter going into a ReLU node, the advice is to set weight of the bias node to be small and positive, e.g. 0.1, which makes it likely that node will be active (larger than 0) for most inputs in the training set.

Goodfellow, Bengio, and Courville (2016), page 226, reports this (replacing sigmoid with ReLU) to be one of the major changes that have improved the performance of the feedforward networks.

ReLU can also be motivated from biology.

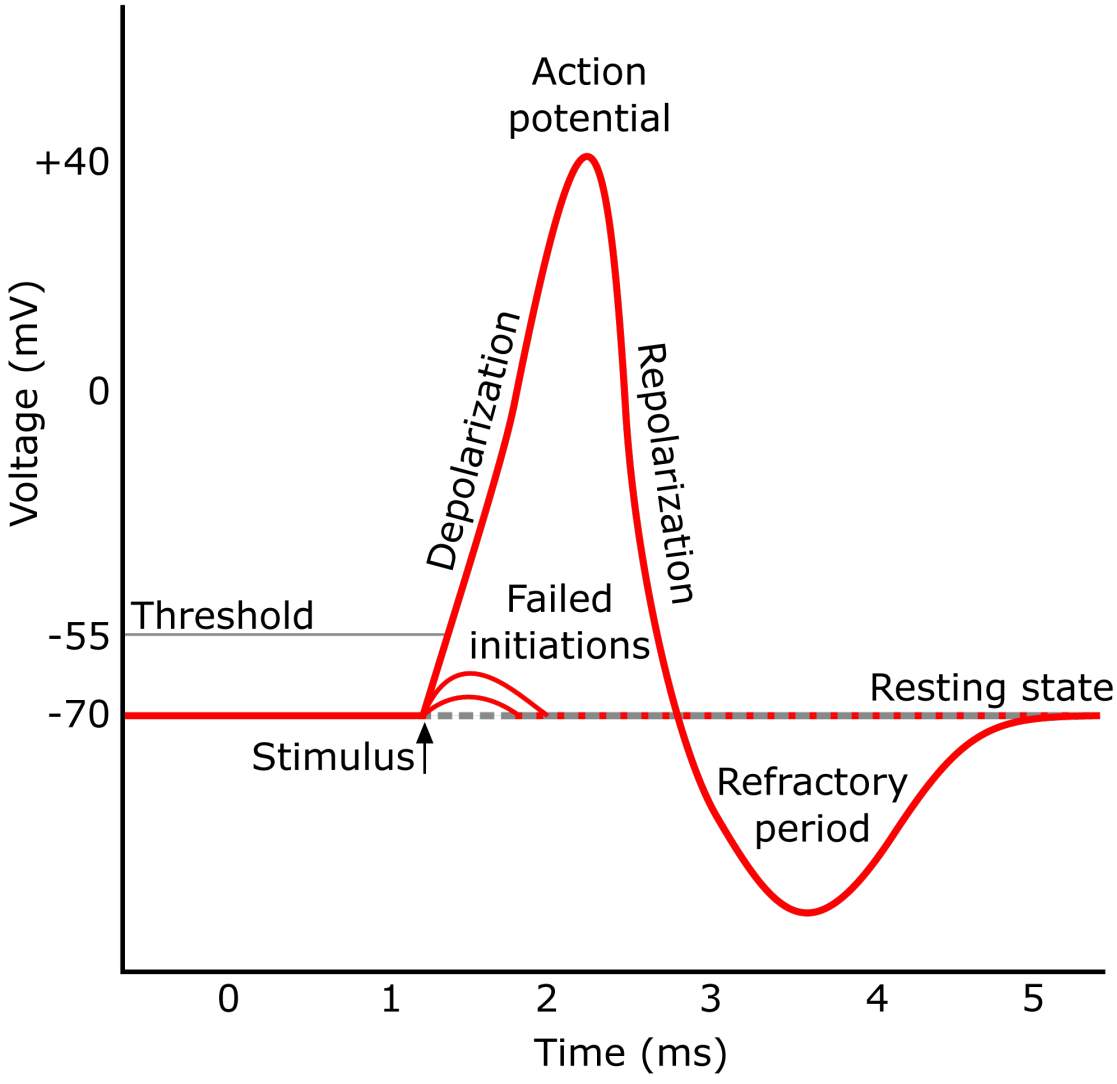
- For some inputs a biological neuron can be completely inactive
- For some inputs a biological neuron output can be proportional to the input
- But, most of the time a biological neuron is inactive.

According to Goodfellow, Bengio, and Courville (2016), page 197, hidden unit design is an *active area of research.*

Other possible choices for hidden layer activation functions are:

- Radial basis functions: as we looked at in Module 9.
- Softplus: $\phi_h(a) = \ln(1 + \exp(a))$
- Hard tanh: $\phi_h(a) = \max(-1, \min(1, a))$

Neural motivation for squashing functions



https://commons.wikimedia.org/wiki/File:Action_potential.svg

Network architecture

How many nodes in the network and how are the nodes connected to eachother? This depends on the problem, and here experience is important.

We will only consider feedforward networks, where all nodes in one layer is connected to all the nodes in the next layer. The layers are then *fully connected* and *dense*.

Choosing architecture to us is to chose the *depth of the network (how many hidden layers)* and the *width of each layer*.

However, the recent practice, see e.g. Chollet and Allaire (2018), Section 4.5.6 and Goodfellow, Bengio, and Courville (2016), page 229, is to

- choose a too large network (too many nodes and/or too many layers) so that if trained until convergence (optimum) then the this would result in overfitting, and
- then use other means to avoid this (various variants of regularization and hyperparameter optimization).

This makes the choice of network architecture to be to *choose a large enough network*. See also Hyperparameter optimization below.

Method

The method part is to choose the loss function for the output layer. The choice of loss function is closely related to the output layer activation function. To sum up, the popular problem types, output activation and loss functions are:

Problem	Output activation	Loss function
Regression	`linear`	`mse`
Classification (C=2)	`sigmoid`	`binary_crossentropy`
Classification (C>2)	`softmax`	`categorical_crossentropy`

where the mathematical formulas are given for n training samples.

Let all network parameters (weights) be denoted by $\boldsymbol{\theta}$.

Mean squared error

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_1(\boldsymbol{x}_i))^2$$

$\hat{y}_1(\boldsymbol{x}_i)$ is the output from the linear output node, and y_i is the response.

Binary cross-entropy

$$J(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}(y_i \ln(\hat{y}_1(\boldsymbol{x}_i)) + (1-y_i)\ln(1-\hat{y}_1(\boldsymbol{x}_i))$$

where $\hat{y}_1(\boldsymbol{x}_i)$ is the output from the sigmoid output node, and y_i is the 0/1 observed class.

Categorical cross-entropy

$$J(\boldsymbol{\theta}) = -\frac{1}{n}\sum_{i=1}^{n}\frac{1}{C}\sum_{c=1}^{C}(y_{ic}\ln(\hat{y}_c(\boldsymbol{x}_i))$$

where $\hat{y}_c(\boldsymbol{x}_i)$ is the output from the softmax output node c and y_{ic} is the observed one-hot coding (0/1) for class c.

Overall

Due to how estimation is done (see below), the loss functions chosen "need" to be:

- differentiable
- possible to compute for each single training data point (or a mini-batch — to be explained soon)

In the 1980-1990s the mean squared error was the prominent loss function also for classification problems, but this has subsequently changed — motivated by the spread of maximum likelihood from statistics to machine learning.
Observe that we have only given the formula for the loss function, and not explicitly assumed anything about any probability distribution of the responses (not even assumed that the responses are random variables). However, we know which statistical model assumptions would give the loss functions as related to the negative of the loglikelihood.

Optimizors

Let the unknown parameters be denoted $\boldsymbol{\theta}$ (what we have previously denotes as αs and βs), and the loss function to be minimized $J(\boldsymbol{\theta})$.
Illustration: Figure 1.9 from Chollet and Allaire (2018) (again).

Gradient descent

Let $\nabla J(\boldsymbol{\theta}^{(t)})$ be the gradient of the loss function evaluated at the current estimate $\boldsymbol{\theta}^{(t)}$, then a step $t+1$ of the algorithm the new values (estimates) of of the parameters are given as:

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \lambda \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})$$

The *learning rate* λ is often set to some small value. In `keras` the default learning rate is 0.01.

Remember that the gradient is the vector of partical derivative of the loss function with respect to each of the parameter in the network.

Q: Why are we moving in the direction of the negative of the gradient? Why not the positive?

Mini-batch stochastic gradient descent (SGD)

The loss function is computed as a mean over all training examples.

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} J(\boldsymbol{x}_i, y_i)$$

This means that the gradient will also be a mean over the gradient contribution from each training example.

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{x}_i, y_i)$$

To build a network that generalizes well, it is important to have many training examples, but that would make us spend a lot of time and computer resources at calculating each gradient descent step.

We observe that we may see the gradient as an average over many individual gradients, and think of this as an estimator for an expectation. This expectation can we (also) approximate by the average gradient over just a *mini-batch* (random sample) of the observations.

The idea here is that the optimizer will converge much faster if they can rapidly compute approximate estimates of the gradient, instead of slowly computing the exact gradient (using all training data).

In addition with multicore systems, mini-batches may be processed in parallell and the batch size is often a power of 2 (32 or 256).

It also turns out that small batches also serves as a regularization effect maybe due to the variability they bring to the optimization process.

In the 4th video (on backpropagation) from 3Blue1Brown there is nice example of one trajectory from gradient decent and one from SGD (13:50 minutes into the video):
https://www.youtube.com/watch?v=tIeHLnjs5U8

This means that for (mini-batch) stochastic gradient descent we do as follows:

1. Divide all the training samples randomly into mini-batches.
2. For each mini-batch: Make predictions of the reponses in the mini-batch in a *forward pass*.
3. Compute the loss for the training data in this batch.
4. Compute the gradient of the loss with regard to the model's parameters (*backward pass*) based on the training data in the batch. $\nabla^*_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})$
5. Update all weighs, but just using the average gradient from the mini-batch $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \lambda \nabla^*_{\boldsymbol{\theta}} J(\boldsymbol{\theta}^{(t)})$
6. Repeat 2-5 until convergence. (Could have gone back to 1, but often not done.)

- The algorithm defined above is called mini-batch SGD. The Stochastic part comes from the fact that we are randomly sampling batches from the training data.
- Stochastic gradient descent (SGD) for size equals to 1.
- Mini-batch SGD is a compromise between SGD (one sample per iteration) and full gradient descent (full dataset per iteration)

Backpropagation algorithm

Computing the analytical expression for the gradient ∇J is not difficult, but the numerical evaluation may be expensive. The Backpropagation algorithm is an simple and inexpensive way to calculate the gradient.

The chain rue is used to compute derivatives of functions of other functions where the derivatives are known, this is done efficiently with backpropagation.

Backpropagation starts with the value of the loss function and works backward from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter have in the loss value.

More information:

- Mathematical details in Goodfellow, Bengio, and Courville (2016) Section 6.5 (pages 204-238),
- 3Blue1Brown: video overview: https://www.youtube.com/watch?v=Ilg3gGewQ5U and chain rule maths https://www.youtube.com/watch?v=tIeHLnjs5U8

Variations of SDG — with adaptive learning rates

General concept:

- momentum term: previous gradients are allowed to contribute.

Named variants: In `keras` the "default optimizer" is RMSprop.

- AdaGrad: individually adapt the learning rates of all model parameters by scaling them inversely proportional to the square root of the sum of all their historical squared values. (Nice properties for convex problems, but with non-linear hidden activation function we do not have a convex problem.)
- RMSprop: modification to AdaGrad in non-convex setting. Scales with exponentially weighted moving average instead of all historical squared gradient values.

Further reading on optimizers

- Keras documentation for Optimizers
- An overview of gradient descent optimization algorithms
- Overview of mini-batch gradient descent
- Andrew Ng explains about RMSprop in Coursera course: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

Regularization

Goodfellow, Bengio, and Courville (2016), Chapter 7, define regularization as *any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.*

We looked at regularization in Module 6, where our aim was to trade *increased bias* for *reduced variance*. Another way of looking at this is we need not focus entirely on finding the model of "correct size", but instead find a large model that has been regularized properly.

In Module 6 we looked in particular at adding a penalty to the loss function. The penalties we looked at were of type absolute value of parameter (L_1, lasso, where we looked at this as model selection) and square value of parameter (L_2, ridge regression). This can also done for neural networks.

In neural networks, *weight decay* is the expression for adding a L_2-penalty to the loss function, and is available in the `nnet` R package.

Other versions of regularization are *dataset augmentation* and *label smoothing*:

- Dataset augmentation means adding fake data to the dataset, in order that the trained model will generalize better. For some learning task it is straightforward to create the new fake data — for image data this can be done by rotating and scaling the images.
- Label smoothing is motivated by the fact that the training data may contain errors in the reponses recorded, and replaced the one-hot coding for C classes with $\epsilon/(C-1)$ and $1-\epsilon$ for some small ϵ.

Early stopping

(Based on Goodfellow, Bengio, and Courville (2016), Section 7.8)
The most commonly used for of regularization is *early stopping*.
If we have chosen a sufficiently large model with the capacity to overfit the training data, we would observe that the training error decreases steadily during training, but the error on the validation set at some point begins to increase.
If we stop the learning early and return the parameters giving the test performance on the validation set, this model would hopefully be a better model than if we trained the model until convergence — and this model will then also give a better test set error.

It is possible to think of the number of *training steps* as a hyperparameter. This hyperparameter can easily be set, and the cost is just that we need to monitor the performance on the validation set during training. Alternatively, cross-validation can be used.

One strategy is to first find the optimal stopping time for the training based on the valiation set (or with cross-validation with small data sets), and then retrain the full training set (including the validation part) and stop at the selected stopping time.

Why is early stopping a regularization technique? By early stopping the optimization procedure has only seen a relatively small part of the parameter space in the neighbourhood of the intitial parameter value. See Goodfellow, Bengio, and Courville (2016), page 250 for a relationship with L_2 regularization.

Hyperparameter optimization

How to avoid overfitting:

- reduce network size,
- collect more observations,
- regularization (including early stopping and drop-out)

The network architacture, the number of batches to run before terminating the optimzation, the drop-out rate, are all examples of hyperparameters that need to be chosen in a sensible way before fitting the final model.
It is important that the hyperparameters are chosen on a validation set or by cross-validation.
However, a "popular" term is *validation-set overfitting* and refers to using the validation set to decide many hyperparameters, so many that you may effectively overfit the validation set.

In statistics we use design of experiments to explore these hyperparameters, and just using marginal grids (one hyperparameter at a time) is common in machine learning.

Example on DOE:hyperparameter optimization with boosting (which of cause also can be used for neural networks). Article: Design of experiments and response surface methodology to tune machine learning hyperparameters, with a random forest case-study (2018), Gustavo A. Lujan-Moreno, Phillip R. Howard, Omar G. Rojas, Douglas Montgomery, Expert Systems with Applications, Volume 109, https://doi.org/10.1016/j.eswa.2018.05.024

Dropout

(Based on Goodfellow, Bengio, and Courville (2016), Section 7.12, and Chollet and Allaire (2018) 4.4.3)
Dropout was developed by Geoff Hinton and his students.

- During training: randomly *dropout* (set to zero) outputs in a layer. Drop-out rates may be chosen between 0.2 and 0.5.
- During test: not dropout, but scale done the layer output values by a factor equal to the drop-out rate (since now more units are active than we had during training)

Alternatively, the drop-out and scaling (now upscaling) can be done during training.

One way to look at dropout is on the lines of what we did in Module 8 when we used bootstrapping to produced many data sets and then fitted a model to each of them and then took the average (bagging). But randomly dropping out outputs in a layer, this can be looked as mimicking bagging — in an efficient way.

See Goodfellow, Bengio, and Courville (2016), Section 7.12 for more insight into the mathematics behind drop-out.

https://www.reddit.com/r/MachineLearning/comments/4w6tsv/ama_we_are_the_google_brain_team_wed_love_to/

The following is a direct quotation.

figplucker: **How was 'Dropout' conceived? Was there an 'aha' moment?**

geoffhinton (2 years ago)

There were actually three aha moments. One was in about 2004 when Radford Neal suggested to me that the brain might be big because it was learning a large ensemble of models. I thought this would be a very inefficient use of hardware since the same features would need to be invented separately by different models. Then I realized that the "models" could just be the subset of active neurons. This would allow combinatorially many models and might explain why randomness in spiking was helpful.

Soon after that I went to my bank. The tellers kept changing and I asked one of them why. He said he didn't know but they got moved around a lot. I figured it must be because it would require cooperation between employees to successfully defraud the bank. **This made me realize that randomly removing a different**

Metrics

Provides different forms to measure how well the predictions are compared with the true values.

- `accuracy`: Percentage of correct classifications
- `mae`: Mean absolute error.

These metrics can be monitored during training (on validation set) or in the end on the test set, and can be the basis for the choice when to top training.

Deep learning

Timeline

(based on Chollet and Allaire (2018))

Neural networks were investigated in "toy form" in the 1950s. The first big step was taken in the 1980s when the backpropagation algorithm were developed (rediscovered) to perform gradient descent in an efficient way.

In 1989 (Bell Labs, Yann LeCun) used convolutional neural networks to classifying handwritten digits, and *LeNet* was used in the US Postal Service for reading ZIP codes in the 1990s.

Not so much seen (?) activity in the neural network field in the 2000s.

In 2011 neural networks with many layers (and trained with GPUs) were performing well on image classification tasks. The *ImageNet* classification challenge (classify high resolution colour images into 1k different categories after training on 1.4M images) was won by solutions with deep convolutional neural networks (convnets). In 2011 the accuracy was 74.3%, in 2012 83.6% and in 2015 96.4%.

From 2012, convnets is the general solution for computer vision tasks. Other application areas are natural language processing.

Deep?

Deep learning does not mean a deeper understanding, but refers to sucessive layers of representations - where the number of layers gives the *depth* of the model. Often tens to hundreds of layers are used. Deep neural networks are not seen as models of the brain, and are not related to neurobiology.

A deep network can be seen as many stages of *information-destillation* (Chollet and Allaire (2018), page 9), where each stage performes a simple data transformation. These transformations are not curated by the data analyst, but is estimated in the network.

In statistics we first select a set of inputs, then look at how these inputs should be transformed (projections in simple form or high-dimensional and nonlinear forms), before we apply some statistical methods. This transformation step can be called *feature engineering* and has been automated in deep learning.

Deep Learning is an algorithm which has no theoretical limitations of what it can learn; the more data you give and the more computational time you provide, the better it is.

Geoffrey Hinton (Google)

In addition, this built-in feature engineering of the deep network is not performed in a greedy fashion, but *jointly* with estimating/learning the full model.

The success of deep learning is dependent upon the breakthroughts in hardware development, expecially with faster CPUs and massively parallell graphical processing units (GPUs). Tensor processing units (TPUs) is the next step.

Achievements of deep learning includes high quality (near-human to super human) image classification, speech recognition, handwriting transcription, machine translation, digital assistants, autonomous driving, advertise targeting, web searches, playing Go and chess.

The R keras package

Earlier good programming skills in C++ was essential to work in deep learning. In addition also skills on programming for GPUs were needed (e.g NVIDIA CUDA programming interface). With the launch of the Keras library now users may only need basic skills in Python or R.

Keras can be seen as a way to use LEGO bricks in deep learning. To quote the web-page:

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.

Tensorflow is a *symbolic-tensor manipulation* framework that also can perform autodifferentiation.

A tensor is defined by its number of axis (below), shape (dimension) and data type (double, integer, character)

- 0D tensor is a scalar
- 1D tensor is a vector
- 2D tensor is a matrix - and the two *axis* of the tensor is the rows and columns
- 3D tensor generalization of a matrix, and may be used for time series data
- 4D tensors may be used for images (samples, height, width, channels), and
- 5D tensors may be used for video (as 4D, plus frames).

Tensor operations (reshaping, dot product) are performed in TensorFlow.

The use of Tensorflow in R Keras is referred to as "using Tensorflow as a backend engine". Tensorflow is the default backend.

More information on the R solution: https://keras.rstudio.com/

Cheat-sheet for R Keras:
https://github.com/rstudio/cheatsheets/raw/master/keras.pdf

Simple data analysis workflow

1. Select training and test data
2. Define the model(s): layers, nodes in layers, activation function
3. Configure the learning process by choosing a loss function, an optimizer, and some metrics to monitor. If needed decide on an evaluation protocol (validation set, cross-validation, iterated cross-validation)
4. Perform any needed preprocessing of data to fit the choices in 2 and 3
5. Fit the model to the data
6. Make preditions for test data or use the model

Chollet and Allaire (2018) Section 4.5 has the following recommentations for step 3+5:

- Develop a first model that performs better than a basic baseline (maybe the basic baseline could be standard statistics solutions)
- Develop a model that overfits the data
- Regularize and tune hyperparameters

Here is a tutorial:
https://keras.rstudio.com/articles/tutorial_overfit_underfit.html

MNIST dataset

This is a larger image version of the handwritten digits data set (a different version, ZIP-codes is found under Recommended exercises). This data analysis is based on https://www.math.ntnu.no/emner/TMA4268/2018v/11NN/8-neural_networks_mnist.html and the `R` `keras` cheat sheet. An advanced version using convolutional neural nets is found here: https://www.math.ntnu.no/emner/TMA4268/2018v/11NN/12-neural_networks_convolution_mnist.html

Objective: classify the digit contained in an image (128×128 greyscale). Problem type: Multiclass classification based on image data.

Labels for the training data:

```
## train_labels
##    0    1    2    3    4    5    6    7    8    9
## 5923 6742 5958 6131 5842 5421 5918 6265 5851 5949
```

1. Training and test data

60 000 images for training and 10 000 images for testing.

```
# Training data
train_images <- mnist$train$x
train_labels <- mnist$train$y

# Test data
test_images <- mnist$test$x
test_labels <- mnist$test$y
org_testlabels <- test_labels
```

The `train_images` is a tensor (generalization of a matrix) with 3 axis, `(samples, height, width)`.

2. Defining the model

In this case we are using a `layer_dense` (fully connected) which expects an input tensor of rank equal to two `(sample, features)` where each `sample` should contain `28*28=784` pixels. Adding a bias term (intercept) is default for `layer_dense`.

```
network <- keras_model_sequential() %>% layer_dense(units = 512,
    input_shape = c(28 * 28)) %>% layer_dense(units = 10, activa
summary(network)
```

```
## Model: "sequential"
## _________________________________________________________________
## Layer (type)                 Output Shape
## =================================================================
## dense (Dense)                (None, 512)
## _________________________________________________________________
## dense_1 (Dense)              (None, 10)
## =================================================================
## Total params: 407,050
## Trainable params: 407,050
## Non-trainable params: 0
```

3. Configure the learning process

We next need to choose the loss function we will use, which is cross-entropy, and then the version of the optimizer. Here htis is RMSprop. Finally, which measure - metrics - do we want to monitor in our training phase? Here we choose accuracy (=percentage correctly classified).

```
network %>% compile(optimizer = "rmsprop", loss = "categorical_c
    metrics = c("accuracy"))
```

4. Preprocessing to match with model inputs and outputs

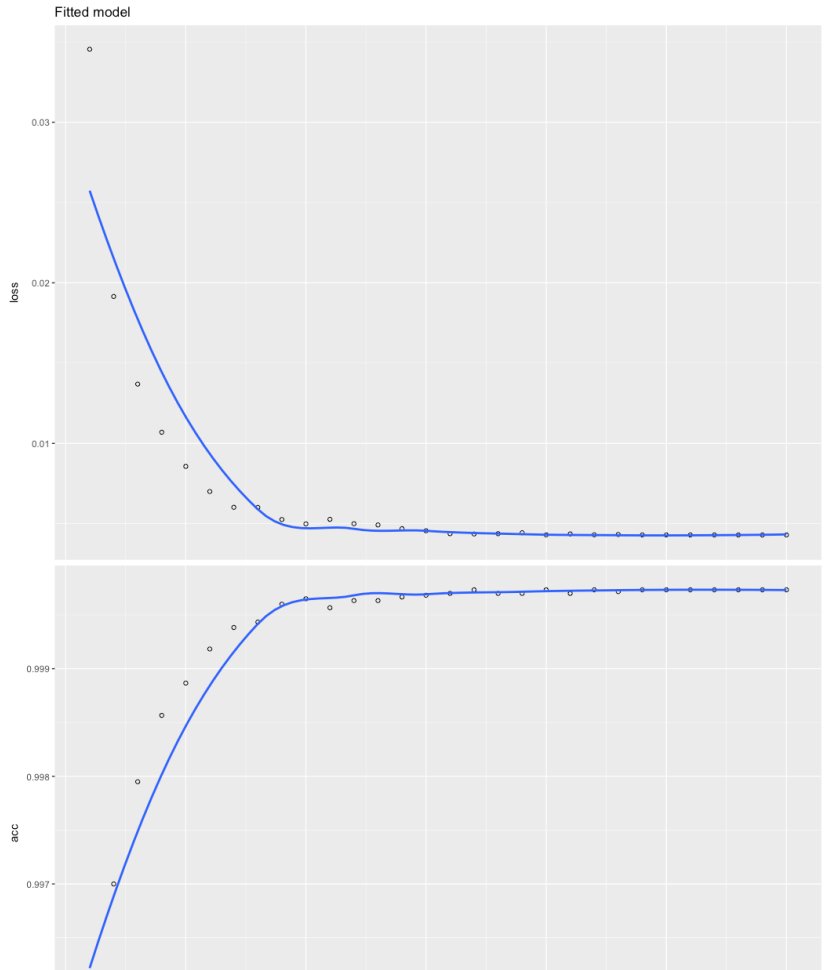
The training data is scored in an array of dimension (60000, 28, 28) of type integer with values in the [0, 255] interval. The data must be reshaped into the shape the network expects (28*28). In addition the grey scale values are scales to be in the [0, 1] interval.
Also, the response must be transformed from 0-10 to a vector of 0s and 1s (dummy variable coding) aka one-hot-coding.

```
train_images <- array_reshape(train_images, c(60000, 28 * 28))
train_images <- train_images/255
train_labels <- to_categorical(train_labels)

test_images <- array_reshape(test_images, c(10000, 28 * 28))
test_images <- test_images/255
test_labels <- to_categorical(test_labels)
```

5. Fit the model

```
fitted<-network %>% fit(train_images, train_labels,
 epochs = 30, batch_size = 128)
library(ggplot2)
plot(fitted)+ggtitle("Fitted model")
```



6. Evaluation and prediction

```
network %>% evaluate(test_images, test_labels)
testres = network %>% predict_classes(test_images)
# $loss [1] 0.1194063 $acc [1] 0.9827
confusionMatrix(factor(testres), factor(org_testlabels))
```

```
Confusion Matrix and Statistics

          Reference
Prediction    0    1    2    3    4    5    6    7    8    9
         0  971    0    3    0    1    2    5    1    1    1
         1    1 1128    2    0    0    0    2    2    2    3
         2    1    1 1009    3    3    0    2    7    2    0
         3    0    1    2  997    0   11    1    3    4    3
         4    1    0    2    0  969    1    4    2    3    7
         5    0    1    0    1    0  871    3    0    1    4
         6    3    2    2    0    3    4  940    0    1    0
         7    1    1    4    2    1    0    0 1002    2    3
         8    2    1    7    0    0    2    1    4  953    1
         9    0    0    1    7    5    1    0    7    5  987
```

Kaggle

Kaggle has hosted machine-learning competitions since 2010, and by looking at solutions to competitions it is possible to get an overview of what works. In 2016-2017 gradient boosting methods won the competitions with structured data ("shallow" learning problems), while deeplearning won perceptual problems (as image classification), Chollet and Allaire (2018) (page 18). Kaggle has helped (and is helping) the rise in deep learning.

Other analyses

Boston housing price

taken from Chollet and Allaire (2018):
https://www.math.ntnu.no/emner/TMA4268/2018v/11NN/11-neural_networks_boston_housing.html

Movie data base

https://www.math.ntnu.no/emner/TMA4268/2018v/11NN/9-neural_networks_imdb.html

Reuters data

https://www.math.ntnu.no/emner/TMA4268/2018v/11NN/10-neural_networks_reuters.html

Summing up

- Feedforward network architecture: mathematical formula - layers of multivariate transformed (`relu`, `linear`, `sigmoid`) inner products - sequentially connected.
- What is the number of parameters that need to be estimated? Intercept term (for each layer) is possible and is referred to as "bias term".
- Loss function to minimize (on output layer): regression (mean squared), classification binary (binary crossentropy), classification multiple classes (categorical crossentropy) — and remember to connect to the correct choice of output activiation function: mean squared loss goes with linear activiation, binary crossentropy with sigmoid, categorical crossentropy with softmax.
- How to minimize the loss function: gradient based (chain rule) back-propagation - many variants.
- Technicalities: `nnet` in R
- Optional: `keras` in R. Use of tensors. Piping sequential layers, piping to estimation and then to evaluation (metrics).

Not covered

(The first two topics are covered in Chollet and Allaire (2018))
Recurrent networks: extending the feedforward network to also have feedback connections. This is a popular type of network to analyse time series data and natural language applications.
Convolutional networks: some layers in a sequential network contain operations specially suitable for grid-like topology (images). Convolution is used in place of general layer (where we do matrix multiplication) in at least one layer. A popular operation is *pooling*.
Explanable AI (XAI): how to use methods on the network or the predictions of the network to figure out the underlying reasoning of the network. Popular methods are called LIME (local linear regression), Shaply (concept from game theory). The `DALEX` R package contains different socalled *explainers*
https://arxiv.org/abs/1806.08915.

References and further reading

- https://youtu.be/aircAruvnKk from 3BLUE1BROWN - 4 videos - using the MNIST-data set as the running example
- Look at how the hidden layer behave: https://playground.tensorflow.org
- Friedman, Hastie, and Tibshirani (2001),Chapter 11: Neural Networks
- Efron and Hastie (2016), Chapter 18: Neural Networks and Deep Learning
- Chollet and Allaire (2018)
- Goodfellow, Bengio, and Courville (2016) (to be used in IT3030)
- Explaining backpropagation http://neuralnetworksanddeeplearning.com/chap2.html
- Slides from MA8701 (Thiago Martins) https://www.math.ntnu.no/emner/MA8701/2019v/DeepLearning/

Acknowledgements

Chollet, François, and J. J. Allaire. 2018. *Deep Learning with R.* Manning Press.
https://www.manning.com/books/deep-learning-with-r.

Efron, Bradley, and Trevor Hastie. 2016. *Computer Age Statistical Inference - Algorithms, Evidence, and Data Science.* Cambridge University Press.

Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2001. *The Elements of Statistical Learning.* Vol. 1. Springer series in statistics New York.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press.