Module 11: Deep Learning and Neural Networks

TMA4268 Statistical Learning V2024

Stefanie Muff, Department of Mathematical Sciences, NTNU

April 18 and 19, 2024

Acknowledgements

- Some of this material was (in a modified version) created by Mette Langaas who has put a lot of effort in creating this module in its original version. Thanks to Mette for the permission to use the material!
- Some of the figures and slides in this presentation are taken (or are inspired) from James et al. (2021).

Learning material for this module

- James et al (2021): An Introduction to Statistical Learning. Chapter 10.
- All the material presented on these module slides and in class.
- Videos on neural networsk and back propagation
  - Video 1
  - Video 2
  - Video 3
  - Video 4

Secondary material (not compulsory):

- Background material: Chapters 6-8 Goodfellow, Bengio, and Courville (2016) https://www.deeplearningbook.org

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

What will you learn?

- Deep learning: The timeline
- Single and multilayer feed-forward networks
- Convolutional neural networks (CNNs)
- Recurrent neural networks (RNNs)
- When to use deep learning?

Introduction: Time line

- 1950's: First neural networks (NN) in "toy form".
- 1980s: the backpropagation algorithm was rediscovered.
- 1989: (Bell Labs, Yann LeCun) used convolutional neural networks to classifying handwritten digits.
- 2000s: After the first hype, NNs were pushed aside by boosting and support vector machines in the 2000s.
- Since 2010: Revival! 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.

- Brain neurons – inspiration for neural networks.

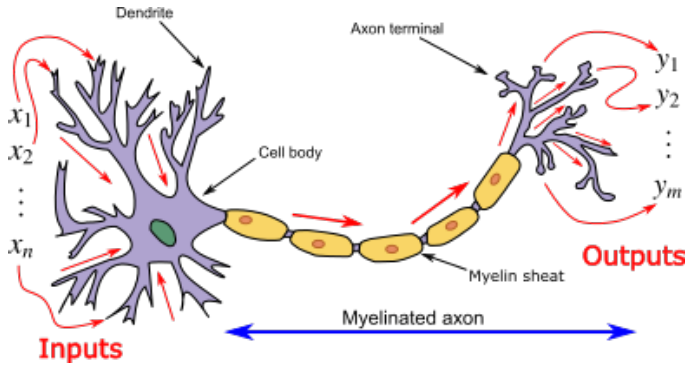
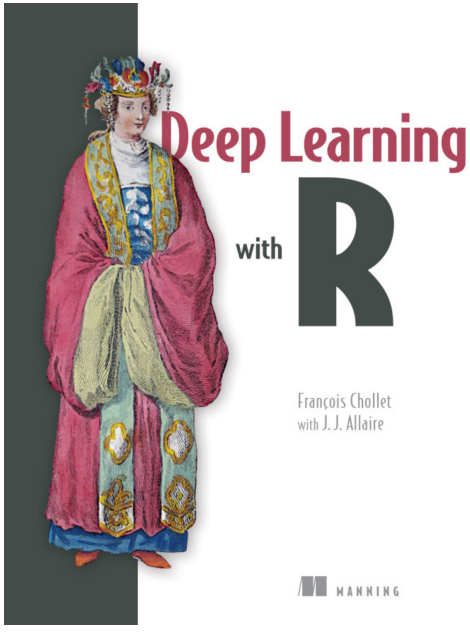


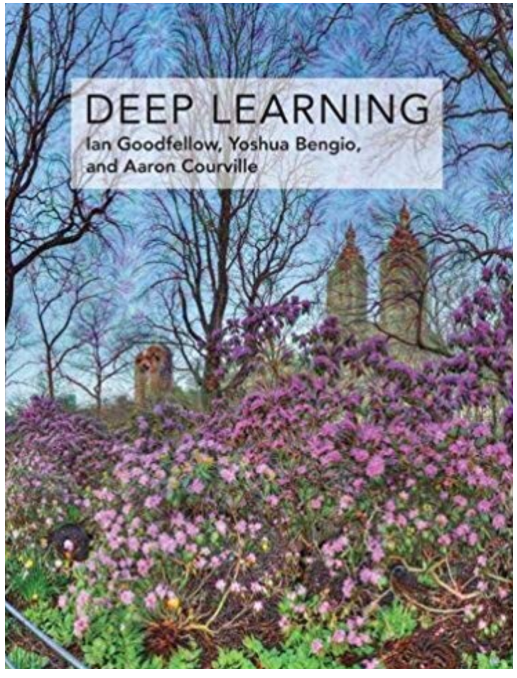
Image credits: By Egm4313.s12 (Prof. Loc Vu-Quoc)
https://commons.wikimedia.org/w/index.php?curid=72816083

- Shift from statistics to computer science and machine learning, as they are highly parameterized.

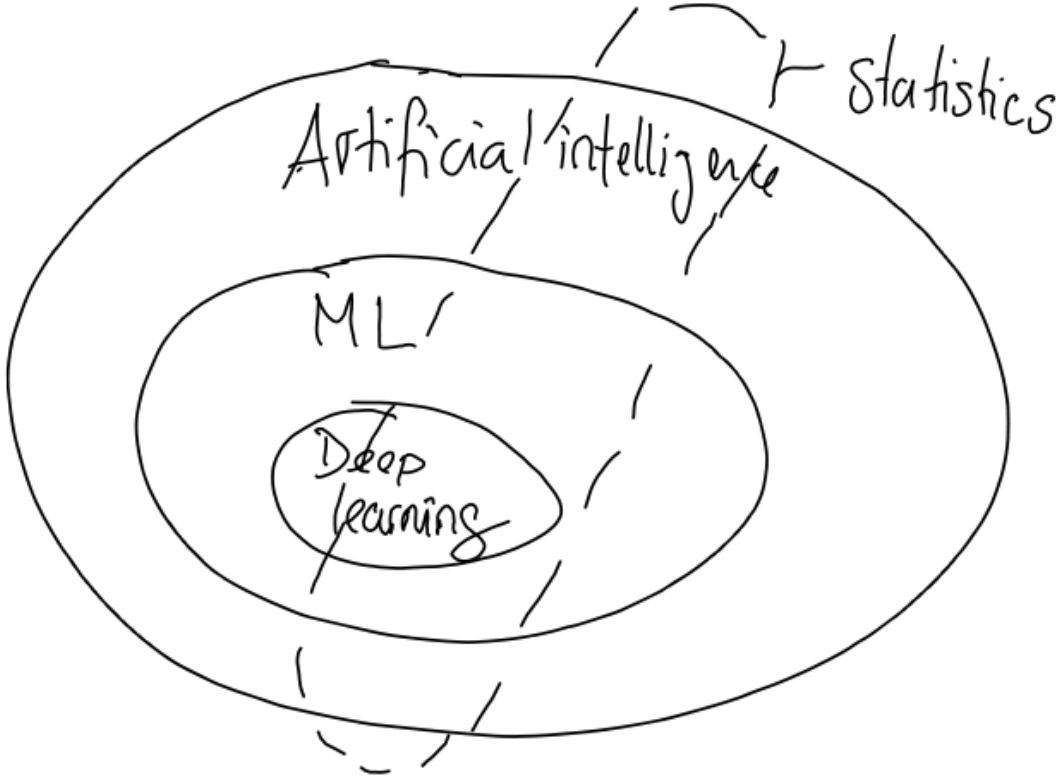
- Statisticians were skeptical: "It's just a nonlinear model".

- There are several learning resources (some listed under ‘further references’) that you my turn to for further knowledge into deep learning.
- There is a new IT3030 deep learning course at NTNU.





AI, machine learning and statistics



AI

- Artificial intelligence (AI) 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 (like in chess programs), but turned out to be intractable for solving more complex, fuzzy problems.

Machine learning

- With the field of *machine learning* the shift is that a system is *trained* rather than explicitly programmed.
- ML deals with much larger and more complex data sets than what is usually done in statistics.
- The focus in ML is 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, [...] it's an engineering science.

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)

- *Deep* does not refer to a deeper understanding.
- Rather, deep referes to the *layers of representation*, for example in a neural network.

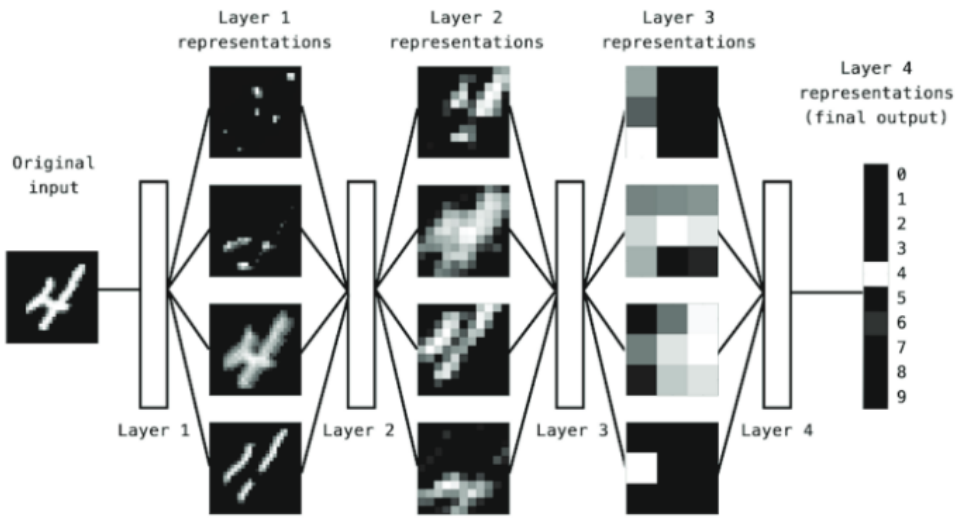


Figure 1.6 Deep representations learned by a digit classification model

- In 2011 neural networks with many layers were performing well on image classification tasks.

- The *ImageNet* classification challenge (classify high resolution color images into 1k different categories after training on 1.4M images) was won by solutions with deep convolutional neural networks (CNNs). In 2011 the accuracy was 74.3%, in 2012 83.6% and in 2015 96.4%.

- Since 2012, CNNs are the general solution for computer vision tasks. Other application area: natural language processing.

Task: Check out this website that gives overview over winning solutions:

https://mlcontests.com/state-of-competitive-machine-learning-2022/#winning-solutions

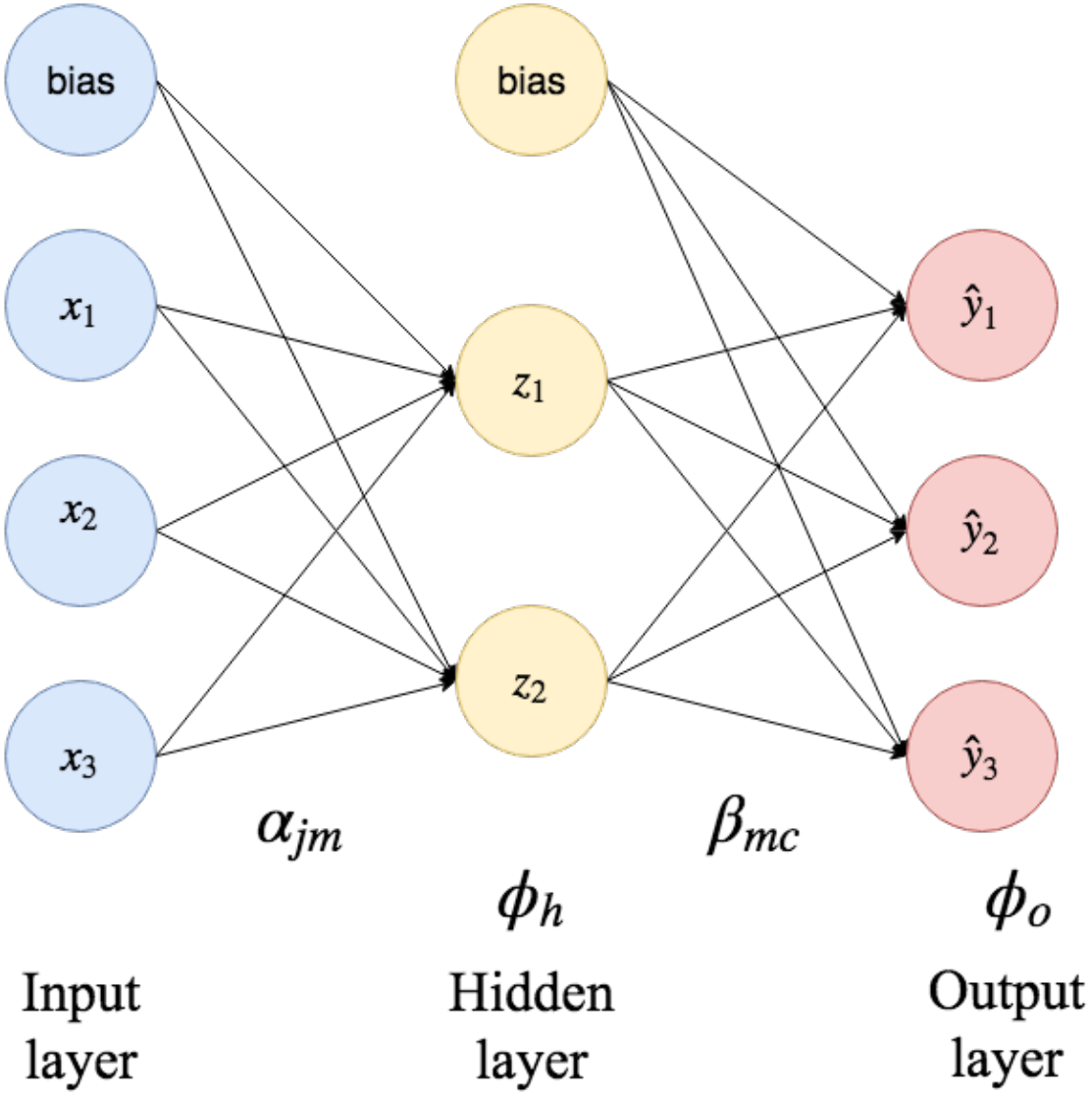
The success of deep learning is dependent upon the breakthroughts in

- *hardware* development, expecially with faster CPUs and massively parallell graphical processing units (GPUs).
- *datasets* and benchmarks (internet/tech data).
- improved *algorithms*.

Achievements of deep learning include

- high quality (near-human to super human) image classification,
- speech recognition,
- handwriting transcription,
- autonomous driving, and more!

Feedforward networks



The single hidden layer feedforward network

The nodes are also called *neurons*.

Notation

1. Inputs: p input layer nodes $\boldsymbol{x}^\top = (x_1, x_2, \ldots, x_p)$.
2. The nodes z_m in the hidden layer, $m = 1, \ldots, M$; as vector $\boldsymbol{z}^\top = (z_1, \ldots, z_M)$, and the hidden layer activation function $g()$.

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

where α_{jm} is the weight[1] from input j to hidden node m, and α_{0m} is the bias term for the mth hidden node.

[1] We stick with greek letters α and β for parameters, but call them weights.

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 $f()$.

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

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

4. Taken together

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

Hands on:

- Identify p, M, C in the network figure above, and relate that to the $y_c(\boldsymbol{x})$ equation.
- How many parameters (the α and βs) need to be estimated for this network?
- What determines the values of p and C?
- How is M determined?

Special case: linear activation function for the hidden layer

If we assume that $g(z) = z$ (linear or identity activiation):

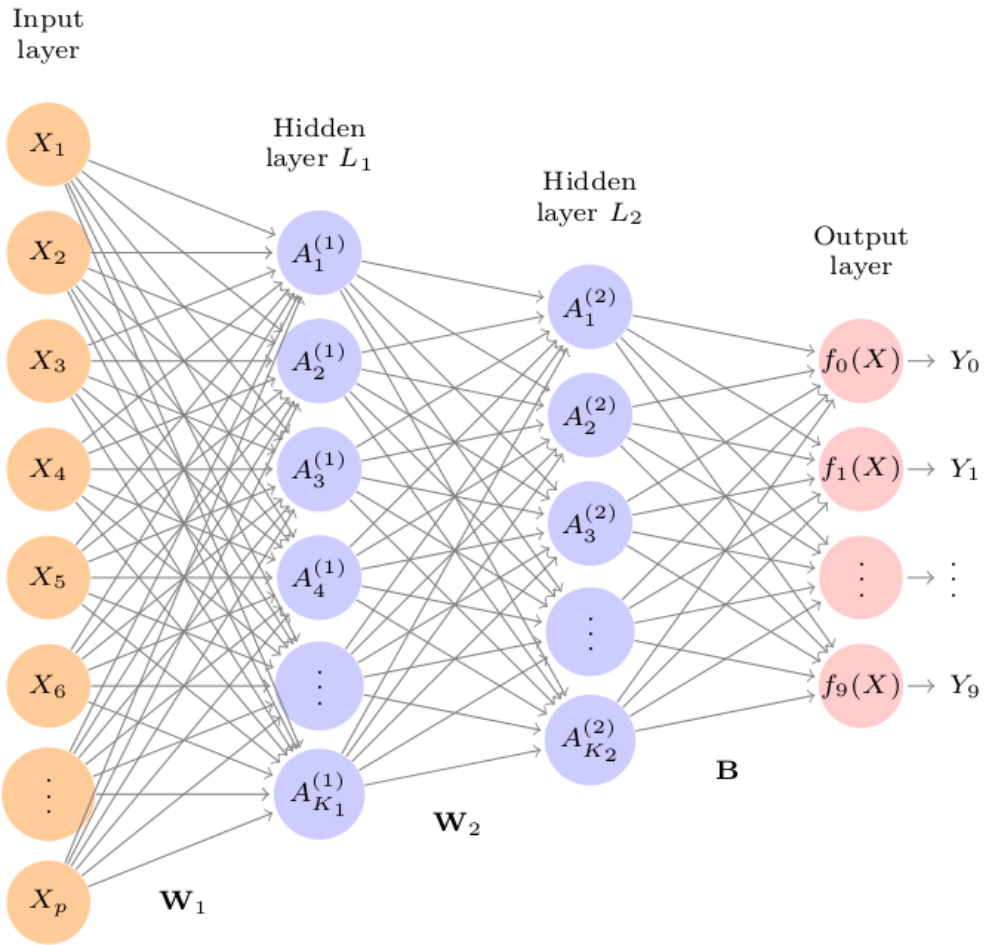
$$\hat{y}_c(\boldsymbol{x}) = f(\beta_{0c} + \sum_{m=1}^{M} \beta_{mc}(\alpha_{0m} + \sum_{j=1}^{p} \alpha_{jm} x_j))$$

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

A:

Multilayer neural networks

Alternative: networks with more than one hidden layer. A network with *many hidden layers* is called a *deep network*.



(Fig 10.4 James et al. (2021))

- The idea of the multilayer NN is exactly the same as for the single-layer version.

- $f_m(X)$ is a (transformation of) the linear combination of the last layer.

Outcome encoding

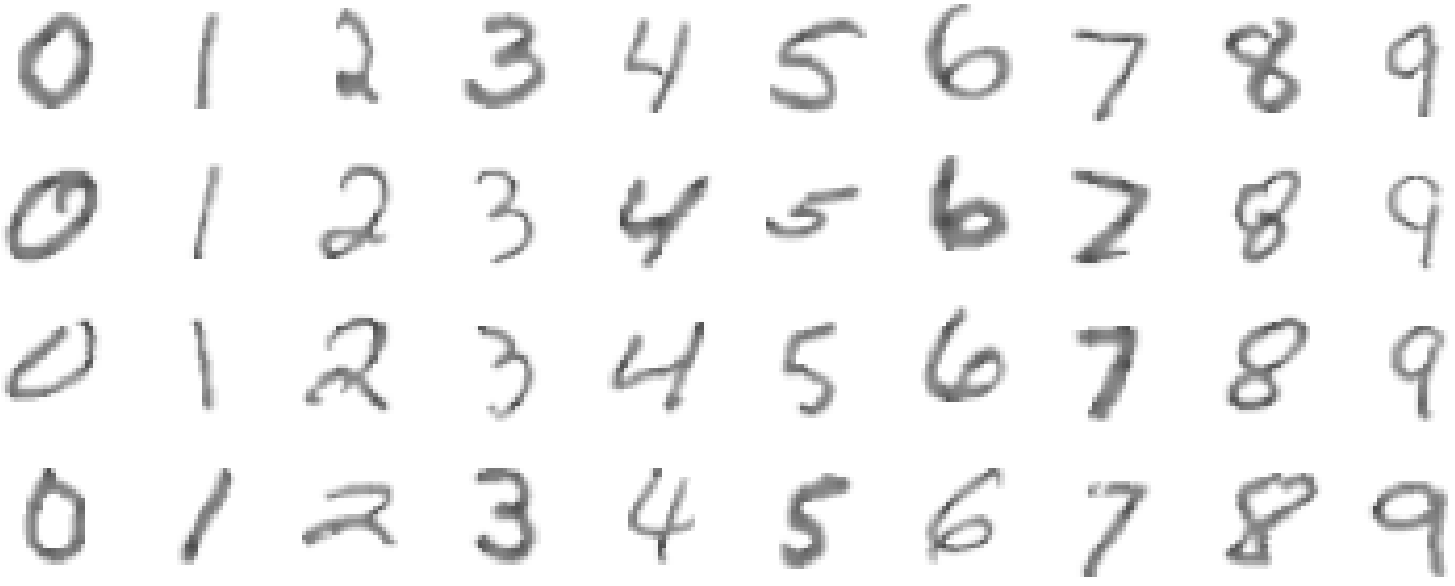
- *Continuous* and *binary* may only have one output node ($y_i =$ observed value).

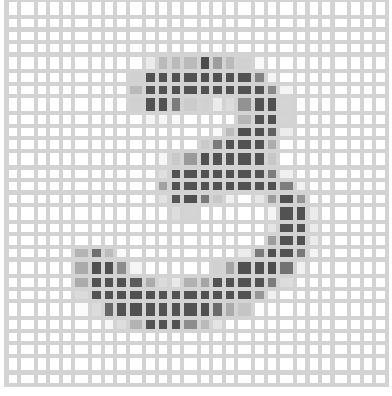
- For C *categories*, we have C output nodes, where we encode the output as $Y = (Y_1, Y_2, \ldots, Y_C)$, where $\boldsymbol{y}_i = (0, 0, \ldots, 0, 1, 0, \ldots, 0)$ with a value of 1 in the c^{th} element of $\boldsymbol{y}_i$ if the class is c. This is called *one-hot encoding* or *dummy encoding*.

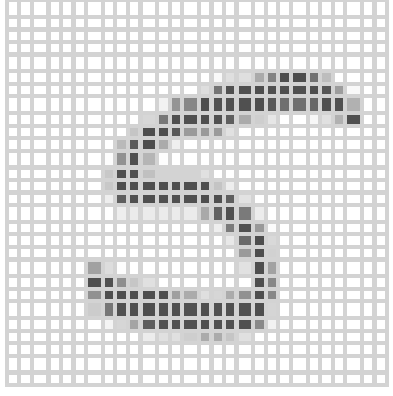
Example: MNIST dataset

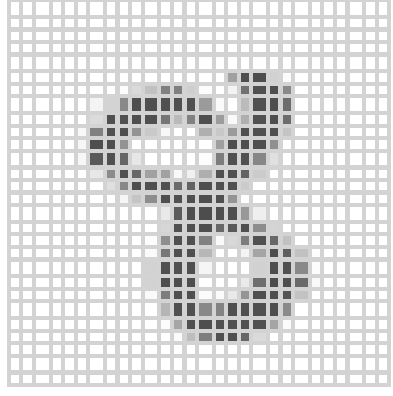
- Aim: Classification of handwritten digits.
- Categorical outcome $C = 0, 1, \ldots, 9$.
- This data has been analysed with a single-layer feed-forward network in a previous version of the course: https://www.math.ntnu.no/emner/TMA4268/2018v/11NN/8-neural_networks_mnist.html.

Objective: classify the digit contained in an image (28×28 greyscale).









Neural network parts

We now focus on the different elements of neural networks.

1) Output layer activation
2) Hidden layer activation
3) Network architecture
4) Loss function
5) Optimizers

1) Output layer activation

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

- *Linear activation*: for *continuous outcome* (regression problems)

$$f(X) = X \ .$$

- *Sigmoid activation*: for *binary outcome* (two-class classification problems)

$$f(X) = \Pr(Y = 1|X) = \frac{1}{1 + \exp(-X)} = \frac{\exp(X)}{1 + \exp(X)} \ .$$

- *Softmax*: for *multinomial/categorical outcome* (multi-class classification problems)

$$f_m(X) = \Pr(Y = m|X) = \frac{\exp(Z_m)}{\sum_{s=1}^{C} \exp(Z_s)} \ .$$

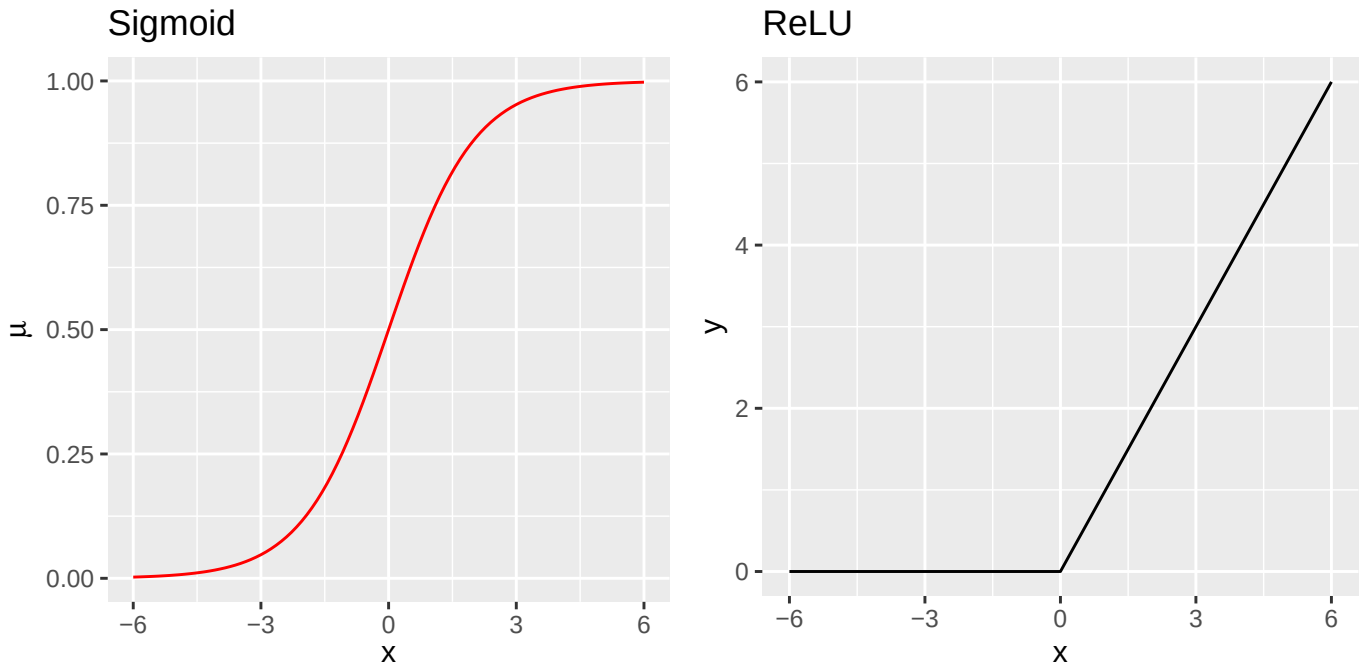
Note that we denote by Z_m the value in the output node m *before* the output layer activation.

2) Hidden layer activation

(See chapter 6.3 in Goodfellow, Bengio, and Courville (2016))

Very common:

- The **sigmoid** $g = \sigma(x) = 1/(1 + \exp(-x))$ (logistic) activation functions.
- The **rectified linear unit (ReLU)** $g(x) = \max(0, x)$ activation functions.



Among all the possibilities, ReLU is nowadays the most popular one. Why?

- The function is piecewise linear, but *in total non-linear*.
- Replacing sigmoid with ReLU is reported to be one of the major changes that have improved the performance of the feedforward networks[2].

[2] Goodfellow et al, Section 6.6

Universal approximation property

- Think of the goal of a feedforward network to approximate some function f, mapping our input vector $\boldsymbol{x}$ to an output value $\boldsymbol{y}$.

- What type of mathematical function can a feedforward neural network with one hidden layer and linear output activation represent?

[3] Goodfellow et al 2016, Section 6.4.1, https://www.deeplearningbook.org

Universal approximation property

- Think of the goal of a feedforward network to approximate some function f, mapping our input vector $\boldsymbol{x}$ to an output value $\boldsymbol{y}$.

- What type of mathematical function can a feedforward neural network with one hidden layer and linear output activation represent?

The *universal approximation theorem*[3] says that a feedforward network with

- a *linear output layer*
- at least one hidden layer with a "squashing" activation function (e.g., ReLU or sigmoid) 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.

[3] Goodfellow et al 2016, Section 6.4.1, https://www.deeplearningbook.org

3) Network architecture

Network architecture contains three components:

- *Width*: How many nodes are in each layer of the network?
- *Depth*: How deep is the network (how many hidden layers)?
- *Connectivity*: How are the nodes connected to each other?

Especially the connectivity depends on the problem, and here experience is important.

- We will consider *feedforward networks*, *convolutional neural networks (CNNs)* and *recursive neural networks (RNNs)*.

4) Loss function ("Method")

- The choice of the loss function is closely related to the output layer activation function.
- Most popular problem types, output activation and loss functions:

Problem	Output nodes	Output activation	Loss function
Regression	1	`linear`	`mse`
Classification (C=2)	1	`sigmoid`	`binary_crossentropy`
Classification (C>2)	C	`softmax`	`categorical_crossentropy`

- Regression: Loss function *MSE* for a given set of parameters $\boldsymbol{\theta}$:

$$R(\boldsymbol{\theta}) = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

- Classification: *Cross-entropy*

$$R(\boldsymbol{\theta}) = -\sum_{i=1}^{n} \sum_{m=1}^{C} y_i \log f_m(x_i) \ ,$$

- with special case for $C = 2$ (*binary cross-entropy loss*):

$$R(\boldsymbol{\theta}) = -\sum_{i=1}^{n} y_i \log f_m(x_i) + (1 - y_i) \log(1 - f_m(x_i)) \ .$$

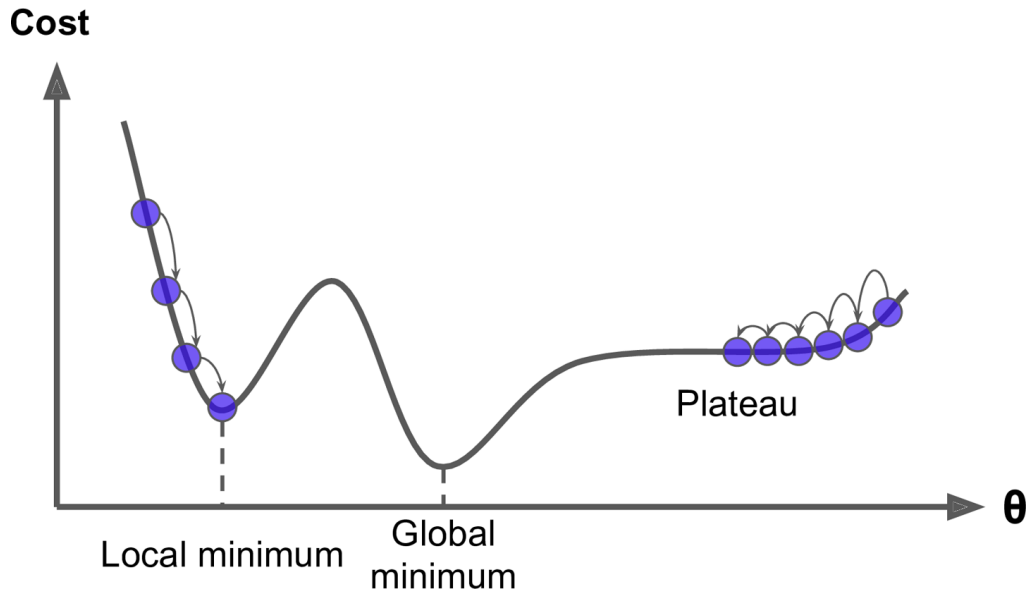
5) Optimizors

Let the unknown parameters be denoted $\boldsymbol{\theta}$ (what we have previously denoted as αs and βs), and the loss function to be minimized $R(\boldsymbol{\theta})$.

- Gradient descent
- Mini-batch stochastic gradient descent (SGD) and true SGD
- Backpropagation

Gradient descent

- We minimize a *cost function* by iteratively tweaking the parameters along the negative gradient.



(https://github.com/SoojungHong/MachineLearning/wiki/Gradient-Descent)

- In practice this happens in a *high-dimensional* parameters space, along the partial derivatives for each parameter.

Finding optimal weights: Gradient descent algorithm

1. Let $t = 0$ and denote the given initial values for the parameters $\boldsymbol{\theta}^{(t)}$.

2. Until finding a (local) optimum, repeat a) to e)
   a) Calculate the predictions $\hat{y}_1(\boldsymbol{x}_i)$.
   b) Calculate the loss function $R(\boldsymbol{\theta}^{(t)})$.
   c) Find the gradient (direction) in the $(p+1)$-dimensional space of the weights, and evaluate this at the current weight values $\nabla R(\boldsymbol{\theta}^{(t)}) = \frac{\partial R}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}^{(t)})$.
   d) Go with a given step length (*learning rate*) λ in the direction of the negative of the gradient of the loss function to get

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

   e) Set $t = t + 1$.

3. The final values of the weights in that $(p+1)$ dimensional space are our parameter estimates and your network is *trained*.

Full vs. stochastic gradient descent (SGD)

- Note that in *full gradient descent*, the loss function is computed as a mean over all training samples:

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

- The gradient is *an average over many individual gradients* from the training sample. You can think of this as an estimator for an expectation

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

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

Mini-batch stochastic gradient descent (SGD)

Crucial idea:
The expectation can be approximated by the average gradient over just a *mini-batch* (random sample) of the observations.

Advantages:

- The optimizer will converge much faster if it can rapidly compute approximate estimates of the gradient.
- Mini-batches may be processed *in parallel*, and the batch size is often a power of 2 (32 or 256).
- Small batches also bring in a *regularization effect*, due to the variability they bring to the optimization process.

Mini-batch stochastic gradient descent

1. Divide all the training samples randomly into *mini-batches*.

2. Until convergence, repeat a) to d)
   a) For each mini-batch: Make predictions of the reponses in the mini-batch in a *forward pass*.
   b) Compute the loss for the training data in this batch.
   c) Compute the gradient $\nabla^*_{\boldsymbol{\theta}} R(\boldsymbol{\theta}^{(t)})$ of the loss with regard to the model's parameters (*backward pass*) based on the training data in the batch.
   d) Update all weighs, but just using the *average gradient* from the mini-batch $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \lambda \nabla^*_{\boldsymbol{\theta}} R(\boldsymbol{\theta}^{(t)})$

3. Network is *trained*; return parameter estimates.

Special case: *True SGD* involves only *one sample* (mini-batch size 1).
$\rightarrow$ Mini-batch SGD is a compromise between SGD (one sample per iteration) and full gradient descent (full dataset per iteration)

In the 3rd video (on backpropagation) from 3Blue1Brown there is nice example of one trajectory from gradient decent and one from SGD (10:10 minutes into the video):
https://www.youtube.com/watch?v=Ilg3gGewQ5U&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=3

Backpropagation algorithm

- *Backpropagation* is a simple and inexpensive way *to calculate the gradient.*
- Computing the analytical expression for the gradient ∇J is not difficult, but *the numerical evaluation may be expensive.*
- The *chain rule* is used to compute derivatives of functions of other functions where the derivatives are known. This is efficiently done with backpropagation.

More background:

- James et al. (2021) Chapter 10.7 (this is part of the compulsory course material - study yourself).

- Mathematical details: Goodfellow, Bengio, and Courville (2016) Section 6.5.

- 3Blue1Brown videos:
  https://www.youtube.com/watch?v=Ilg3gGewQ5U and
  https://www.youtube.com/watch?v=tIeHLnjs5U8

Regularization

- Often: more weights than data samples → danger for over-fitting.
- Regularization: *any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error* [4].
- Remember:
  - **Module 6**: The aim of regularization was to trade *increased bias* for *reduced variance*. The ideas was to add a penalty to the loss function.
  - **Module 9**: Tree pruning and L1 and L2 regularization in XGBoost.

[4] Goodfellow et al, Chapter 7

Regularization in neural networks

- *Ridge/Lasso penalization*: In neural networks this means adding an L_2 or L_1-penalty to the loss function to *penalize large weights* [5]:

$$\tilde{J}(\boldsymbol{w}) = R(\boldsymbol{w}) + \lambda \boldsymbol{w}^\top \boldsymbol{w}$$
$$\tilde{J}(\boldsymbol{w}) = R(\boldsymbol{w}) + \alpha ||\boldsymbol{w}||$$

- *Data augmentation*: Adding "fake data" to the dataset, in order that the trained model will generalize better. For example: rotating and scaling images.
- *Label smoothing*: 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*
- *Dropout*
- SGD is also a form of regularization (prevents over-fitting).

[5] see chapter 10.7.2 in the course book

Early stopping

Based on Goodfellow, Bengio, and Courville (2016), Section 7.8

- The most commonly used for of regularization.
- For a sufficiently large model with the capacity to over-fit the training data, we observe that the training error decreases steadily during training, but the error on the validation set at some point begins to increase.
- The idea: Return the parameters that (earlier) gave the best performance on the validation set, before convergence on the training data.

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) some outputs in a given layer at each iteration. Drop-out rates may be chosen between 0.2 and 0.5.
- During test: no dropout, but scale down 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.

Dropout

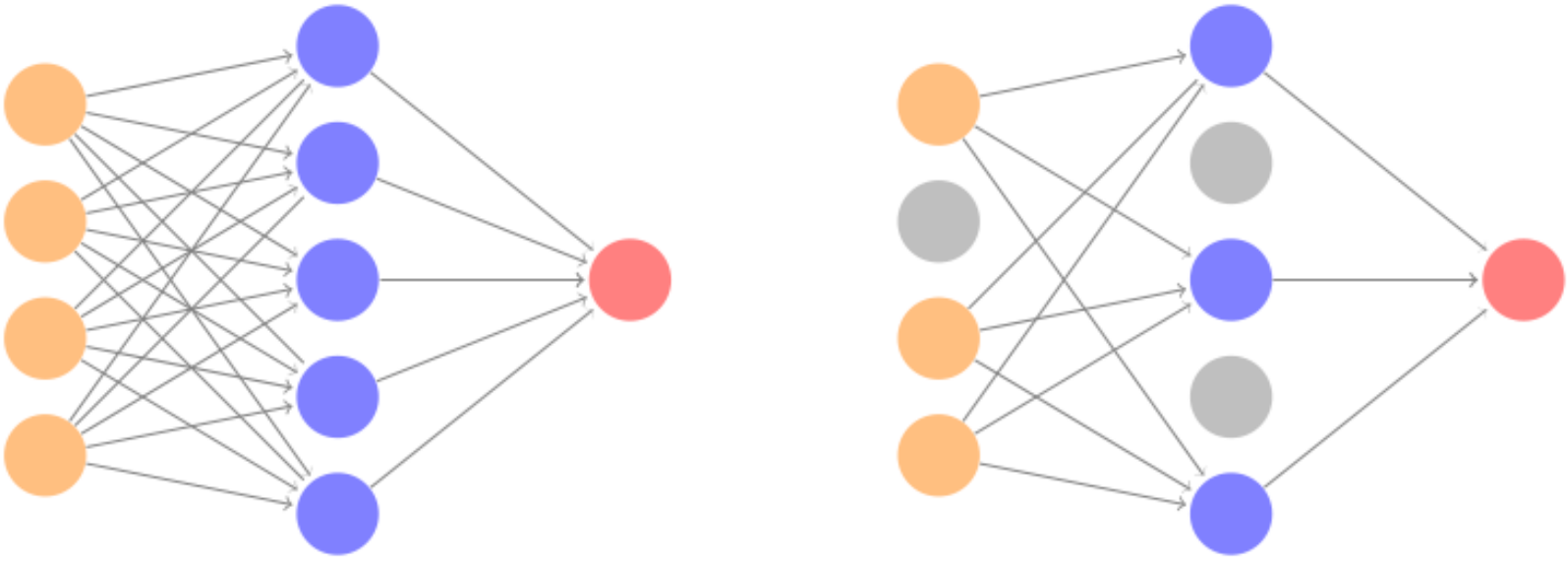


Fig 10.19, James et al. (2021)

Ways to avoid overfitting

There are **hyperparameters** when building and fitting a neural network, like the network architecture, the number of batches to run before terminating the optimization, the drop-out rate, etc.

To avoid overfit, we have some strategies:

- Reduce network size.
- Collect more observations.
- Regularization.

It is important that the hyperparameters are chosen on a validation set or by cross-validation.

However, we may run into *validation-set overfitting*: when using the validation set to decide many hyperparameters, so many that you may effectively overfit the validation set.

How to fit those models?

- We will use both the rather simple `nnet` R package by Brian Ripley and the currently very popular `keras` package for deep learning (the `keras` package will be presented later).
- `nnet` fits *one hidden layer* with *sigmoid activiation function*. The implementation is not gradient descent, but instead BFGS using `optim`.
- Type `?nnet()` into your R-console to see the arguments of `nnet()`.

An example

Boston house prices

Objective: To predict the median price of owner-occupied homes in a given Boston suburb in the mid-1970s using 10 input variables.
This data set is both available in the `MASS` and `keras` R package.

Read and check the data file:

```
library(MASS)
data(Boston)
dataset <- Boston
head(dataset)
```

```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio  black lstat
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3 396.90  4.98
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8 396.90  9.14
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8 392.83  4.03
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7 394.63  2.94
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7 396.90  5.33
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7 394.12  5.21
##   medv
## 1 24.0
## 2 21.6
## 3 34.7
## 4 33.4
## 5 36.2
## 6 28.7
```

Preparation: Split into training and test data:

```
set.seed(123)
tt.train <- sort(sample(1:506,404,replace=FALSE))
train_data <- dataset[tt.train,1:13]
train_targets <- dataset[tt.train,14]

test_data <- dataset[-tt.train,1:13]
test_targets <- dataset[-tt.train,14]
```

- To make the optimization easier with gradient based methods do *feature-wise normalization*.

```
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**: 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.

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

```
library(nnet)
fit5<- nnet(train_targets~., data=train_data,size=5,linout=TRUE,maxit=1000,trace=F)
```

Calculate the MSE and the mean absolute error:

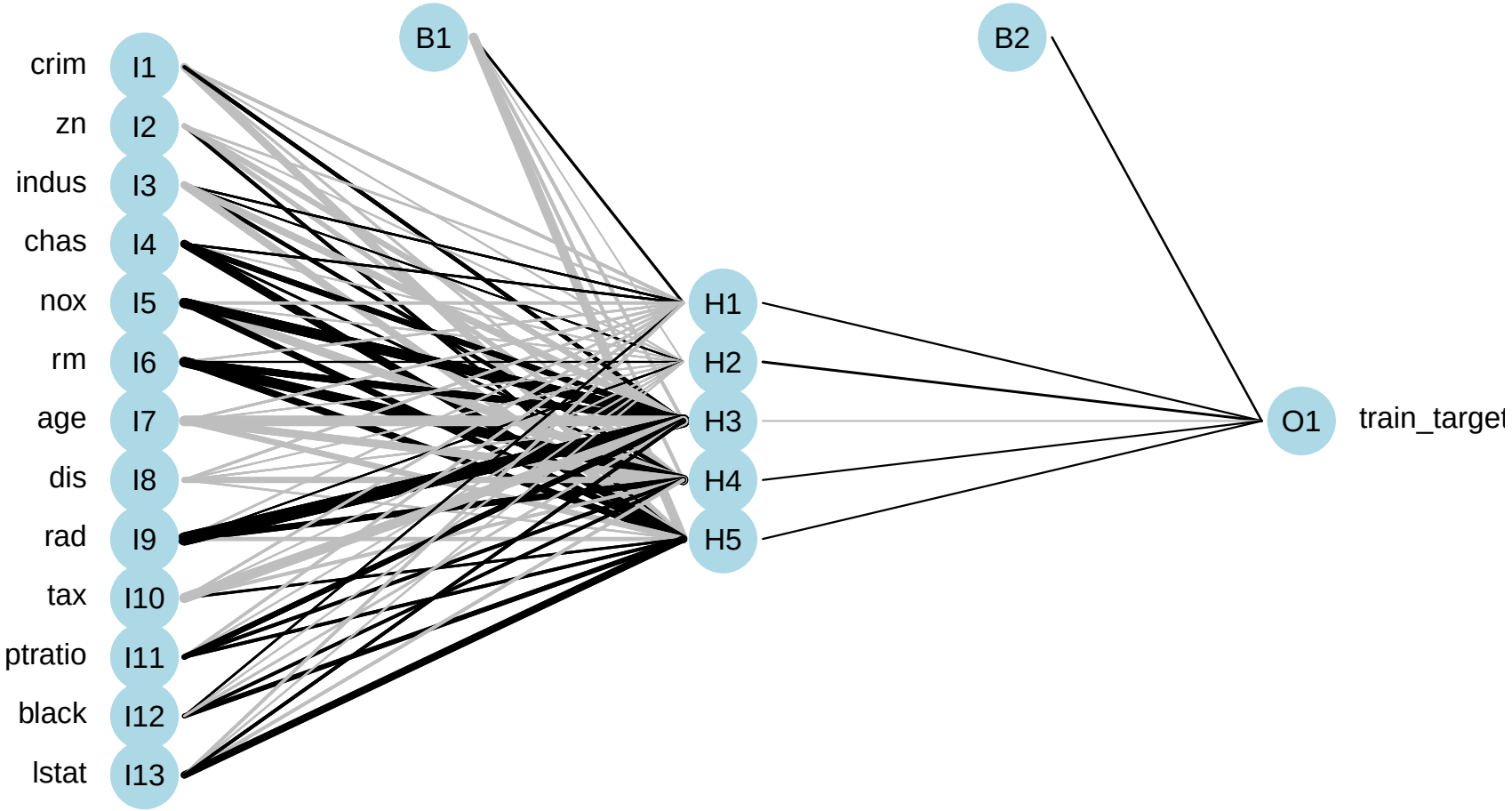
```
pred=predict(fit5,newdata=test_data,type="raw")
mean((pred[,1]-test_targets)^2)
```

```
## [1] 31.10861
```

```
mean(abs(pred[,1]-test_targets))
```

```
## [1] 2.91189
```

```
library(NeuralNetTools)
plotnet(fit5)
```



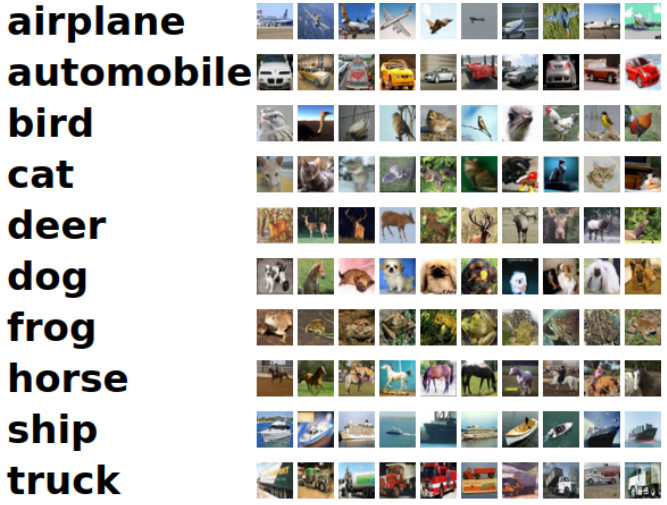
Boston example using `keras`

See recommended exercise.

Convolutional neural networks (CNNs)

- Motivated by image classification.

- Example: the CIFAR-100 dataset (https://www.cs.toronto.edu/~kriz/cifar.html): Images of 100 categories with 600 images each.



(Example from the CIFAR-10 data set with only 10 classes).

- Idea of CNNs: recognize features and patterns.
- The network identifies *low-level features* (edges, color patched etc).
- These low-level features are then combined into *higher-level features*.
- Two types of layers: *Convolution layers* and *pooling layers*.

Convolution layers

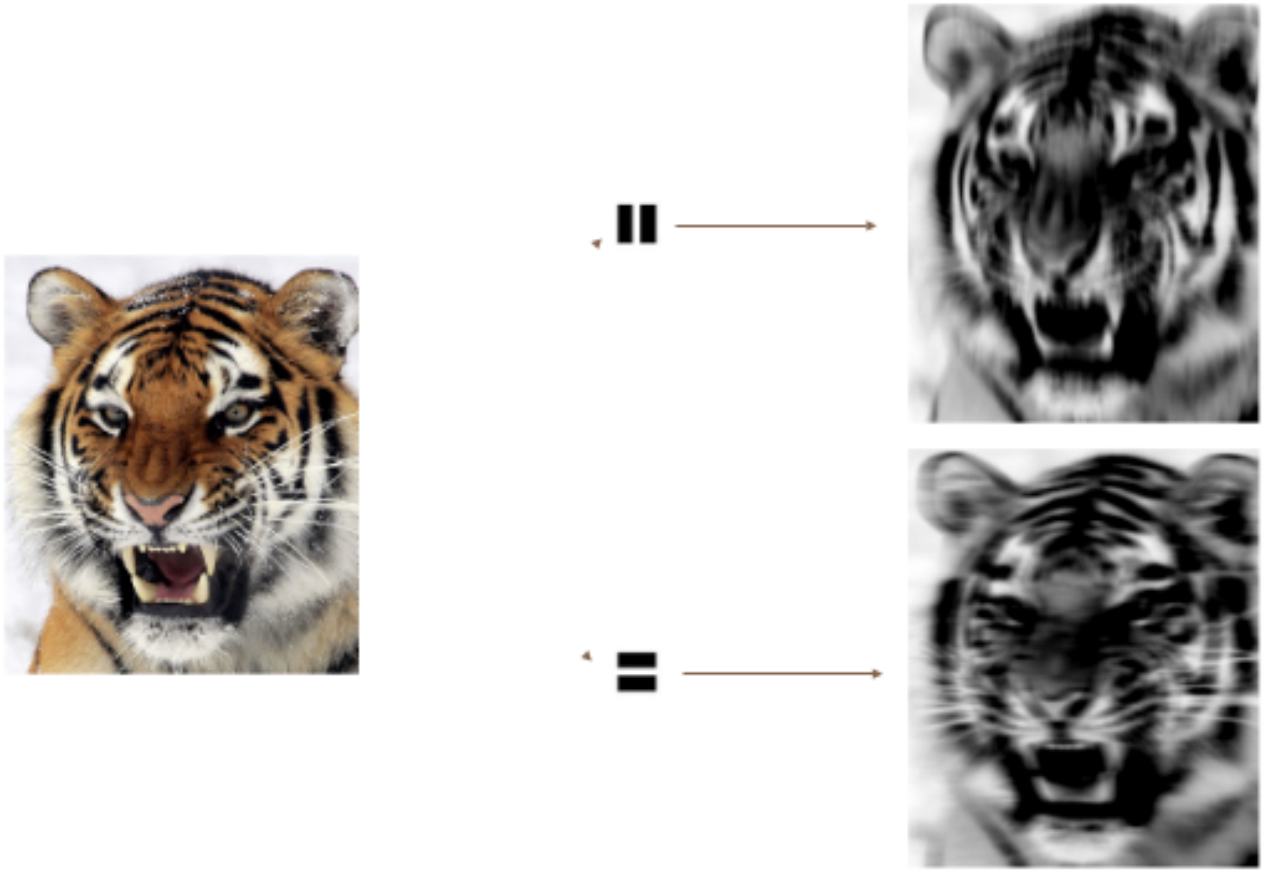
- Composed of *filters*.
- Example:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix} \quad \text{Convolved with} \quad \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}$$

$\rightarrow$ Convolved image:

The filter highlights regions in the image that are similar to the filter itself.

Filtering for vertical or horizontal stripes:



(Figure 10.7)

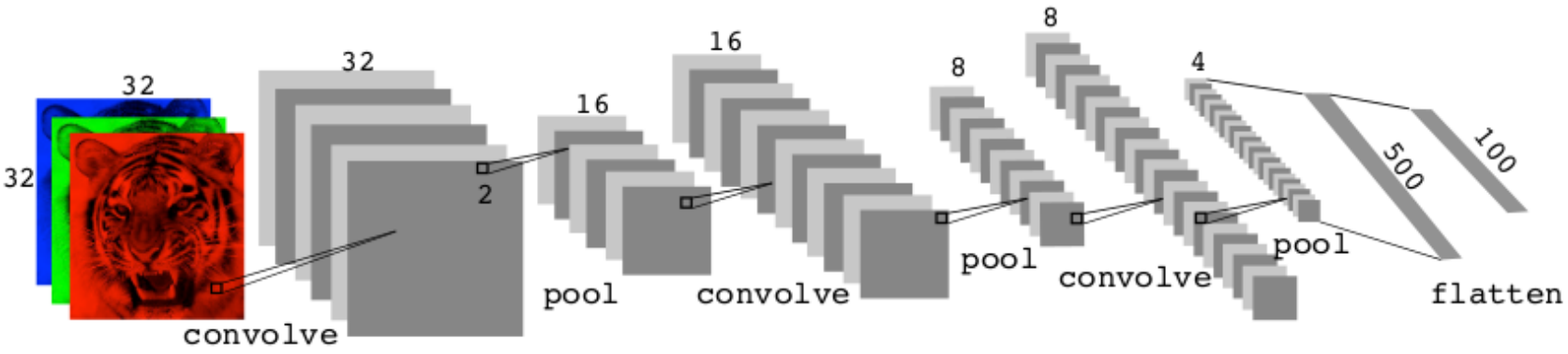
- In *image processing* we would use predefined (fixed) filters.
- In CNNs, the idea is that the filters are *learned*.
- One filter is applied to each color (red, green, blue), so three convolutions are happening in parallel and then immediately summed up.
- In addition, we can use K different filters in a convolution step. This produces 3D feature maps (of depth K).
- The convolved image is then also processed with the ReLU activation function.

Pooling layers

- Idea: consense/summarize information about the image.
- *Max pool*: Use the maximum value in each 2×2 block.

$$\begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}$$

In a CNN, we now combine convolution and pooling steps interatively:



- The number of channels after a convolution step is the number of filters (K) that is used in this iteration.
- The dimension of the 2D images after a pooling step is reduced, depending on the dimension of the filter (e.g., 2×2 reduces each dimension by a factor of 2).
- In the end, all the dimensions are *flattened* (pixels become ordered in 2D).
- The output layer has a *softmax* activation function since the aim is classification.

Data augmentation

- Very simple idea: Make the analysis more robust by including replicated, but slightly modified pictures of the original data.

- Example:

Figure 10.9 of James et al. (2021)

Examples

See

- Section 10.3.5 in the book.
- Examples in the recommended exercise 11.

Recurrent neural networks (RNNs)

- Suitable for data with sequential character.
- Examples: Text documents, time series (temperature, stock prices, music, speech,...)
- The input object X is a sequence.
- In the most simple case, the output Y is a single value (continuous, binary or a category).
- More advanced RNNs are able to map sequences to sequences (*Seq2Seq*)[6] in language modeling, and much more!

[6] Google Translate uses this technique, for example

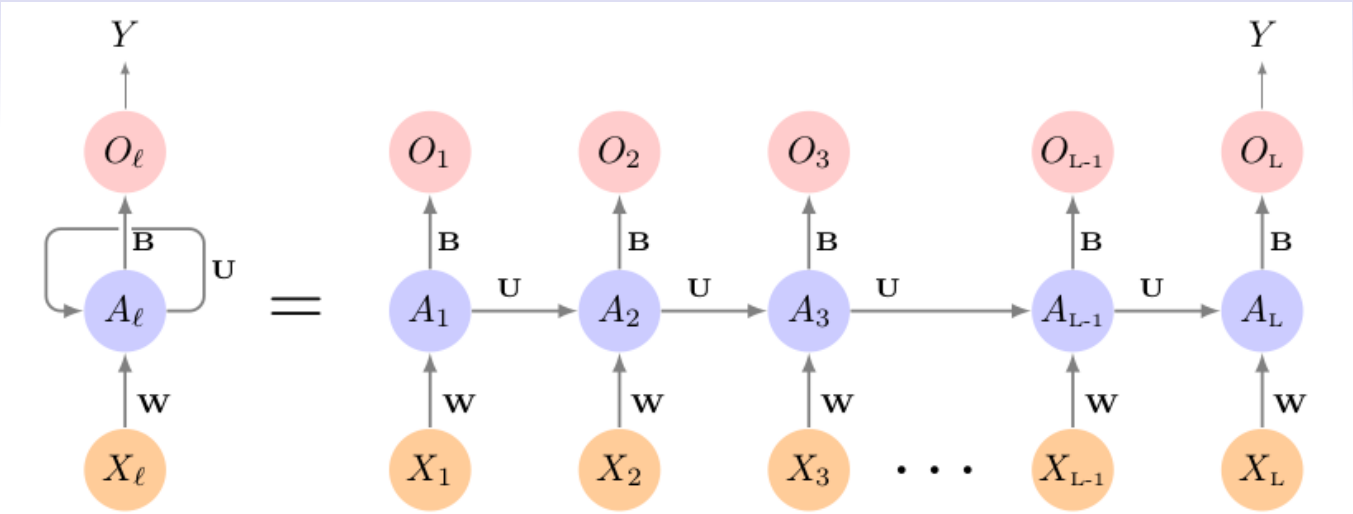


Figure 10.12 of James et al. (2021)

- Observed sequence $X = \{X_1, \ldots, X_L\}$, where each $X_l^\top = (X_{l1}, \ldots, X_{lp})$ is an input vector at point l in the sequence.
- Sequence of hidden layers $\{A_1, \ldots, A_L\}$, where each A_l is a layer of K units $A_l^\top = (A_{l1}, \ldots, A_{lK})$.
- A_{lk} is determined as

$$A_{lk} = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_{lj} + \sum_{s=1}^{K} u_{ks} A_{l-1,s}) \ , \qquad (1)$$

with hidden layer activation function $g()$ (e.g., ReLU).

- The output is determined as

$$O_l = \beta_0 + \sum_{k=1}^{K} \beta_k A_{lk} \ ,$$

  potentially with a sigmoid or softmax output activation for binary or categorical outcome.

- Note: The weights $\boldsymbol{W}$, $\boldsymbol{U}$ and $\boldsymbol{B}$ are the *same* at each point in the sequence. This is called *weight sharing*.

Fitting the weights in an RNN

- Minimize a *loss function*. In regression problems:

$$\text{Loss} = (Y - O_L)^2 \ .$$

- Only the *last observation* is relevant. How can this be meaningful?
- Reason: each element X_l contributes to O_L via equation (1).
- For input sequences (x_i, y_i), we minimize $\sum_{i=1}^{n}(y_i - o_{iL})^2$.
- Note: $x_i = \{x_{i1}, \ldots, x_{iL}\}$ is a sequence of *vectors*.

Why are the outputs $O_1, \ldots, O_{L-1}$ there at all?

Why are the outputs $O_1, \ldots, O_{L-1}$ there at all?

A:

- They come for free (same weights $\boldsymbol{B}$).
- Sometimes, the output is a whole sequence.

Example of an RNN: Time series forecasting

Trading statistics from New York Stock exchange:

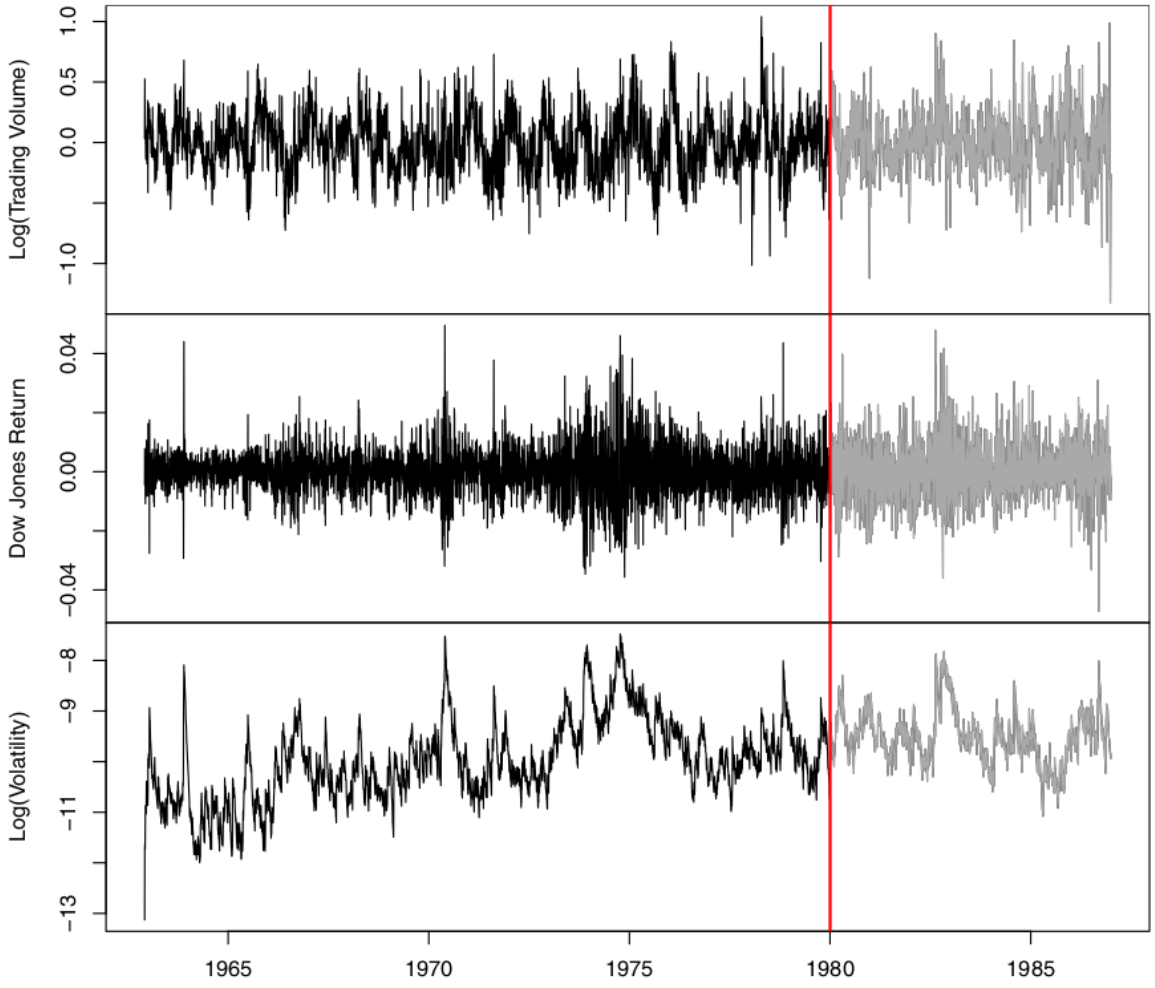


Figure 10.14 of James et al. (2021)

Observations:

- Every day ($t = 1, \ldots, 6051$) we measure three things, denoted as (v_t, r_t, z_t).
- All three series have high *auto-correlation*.

Aim:

- Predict the trading volume v_t on day t from
  - v_{t-1}, v_{t-2}, $\ldots$,
  - r_{t-1}, r_{t-2}, $\ldots$, and
  - z_{t-1}, z_{t-2}, $\ldots$

But, how do we represent this problem in terms of Figure 10.12?

The idea is to extract shorter series up to a *lag* of length L:

$$X_1 = \begin{pmatrix} v_{t-L} \\ r_{t-L} \\ z_{t-L} \end{pmatrix}, \quad X_2 = \begin{pmatrix} v_{t-L+1} \\ r_{t-L+1} \\ z_{t-L+1} \end{pmatrix}, ..., \quad X_L = \begin{pmatrix} v_{t-1} \\ r_{t-1} \\ z_{t-1} \end{pmatrix}, \quad Y = v_t$$

- And then continue to formulate the model as indicated in Figure 10.12.

When to use deep learning?

- We have learned about many new "fancy" and trendy methods. But is it always worth using the most advanced ones?
- Important: Try the simple methods as well. Sometimes they perform quite well.
- Advantage of, for example, simple linear regression?

Example: The Hitters data set

- Remember from Chapter 6: Prediction of `Salary` for 263 baseball players.

We compare

- Linear model with 20 parameters.
- Lasso with CV, where 12 variables remain in the model
- A NN with one hidden layer and 64 units. The model has 1049 parameters.

Model	# Parameters	Mean Abs. Error	Test Set R^2
Linear Regression	20	254.7	0.56
Lasso	12	252.3	0.51
Neural Network	1409	257.4	0.54

TABLE 10.2. *Prediction results on the* `Hitters` *test data for linear models fit by ordinary least squares and lasso, compared to a neural network fit by stochastic gradient descent with dropout regularization.*

Conclusions?

- ...
- ...
- ...

DL in Medicine

Lots of papers like those:

Review article

Comparison of machine learning and logistic regression models in predicting acute kidney injury: A systematic review and meta-analysis

Xuan Song [a], Xinyan Liu [a], Fei Liu [b], Chunting Wang [c]

Comparison of logistic regression and machine learning methods for predicting postoperative delirium in elderly patients: A retrospective study

Yu-xiang Song, [1,2] Xiao-dong Yang, [3] Yun-gen Luo, [1,2] Chun-lei Ouyang, [1] Yao Yu, [1] Yu-long Ma, [1] Hao Li, [1] Jing-sheng Lou, [1] Yan-hong Liu, [1] Yi-qiang Chen,[✉3] Jiang-bei Cao,[✉1] and Wei-dong Mi[✉1]

▸ Author information ▸ Article notes ▸ Copyright and License information Disclaimer

Conclusions

The optimal application of the logistic regression model could provide quick and convenient POD risk identification to help improve the perioperative management of surgical patients because of its better sensitivity, fewer variables, and easier interpretability than the machine learning model.

DL/ML in Ecology

Very recent paper:

Methods in Ecology and Evolution | BRITISH ECOLOGICAL SOCIETY

REVIEW | Open Access

Machine learning and deep learning—A review for ecologists

Maximilian Pichler, Florian Hartig

First published: 13 February 2023 | https://doi.org/10.1111/2041-210X.14061 | Citations: 1

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) (used in IT3030) https://www.deeplearningbook.org/
- 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.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning.* 2nd ed. Vol. 112. Springer.