Deep learning

Feed-forward and convolutional neural networks for image recognition

Erik-Jan van Kesteren

Supervised machine learning

1. Regression (predicting continuous outcomes)
2. Model evaluation
3. Classification (predicting discrete outcomes)
4. **Deep learning**

Today

- Introduction to neural networks
- Feed-forward & deep neural networks
- Estimation / optimization
- Convolutional neural networks
- Battling the curse of dimensionality

Introduction

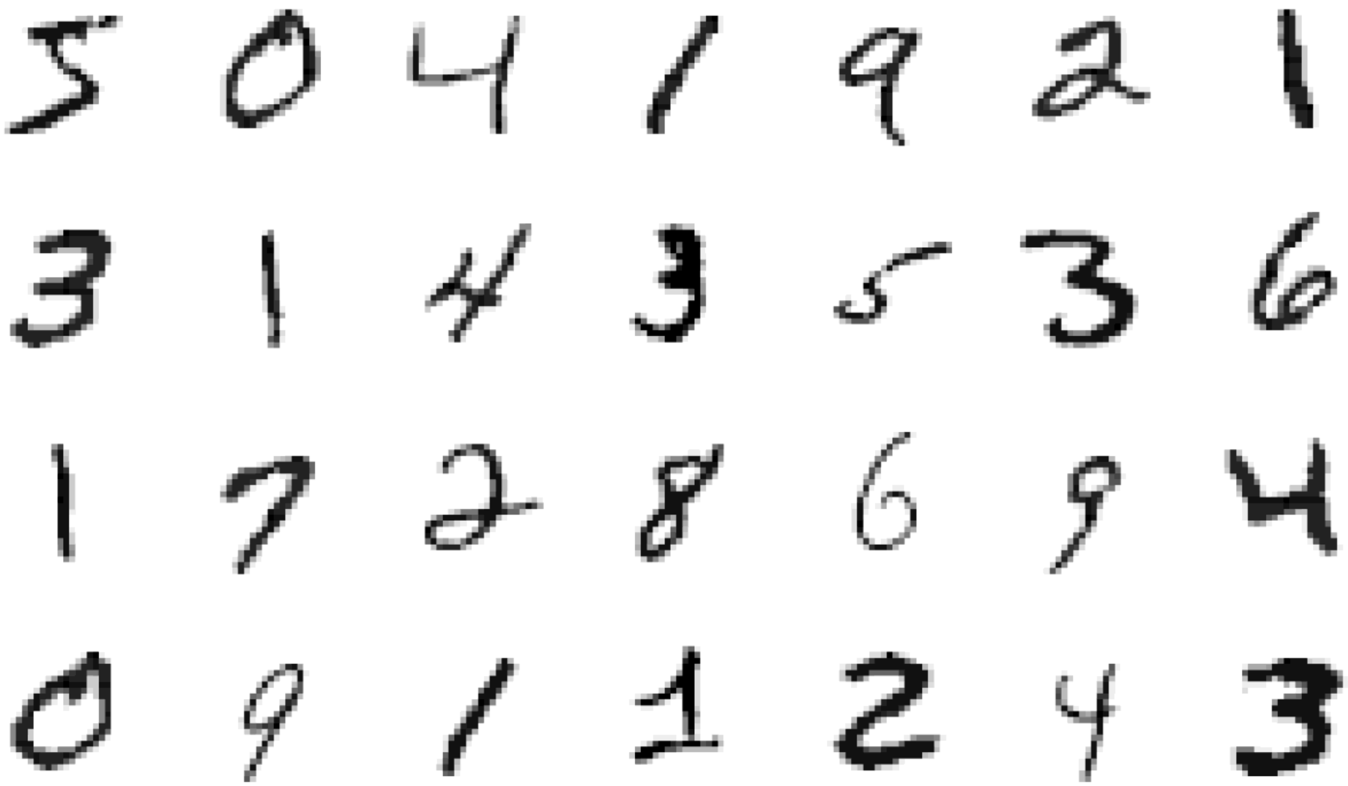
Why should we learn this?

State-of-the-art performance on various tasks

- Text prediction (your phone's keyboard)
- Text mining (at the end of this course!)
- Forecasting
- Object recognition
- Sound recognition
- Spam filtering
- Image generation
- Style transfer
- Image denoising
- Compression (dimension reduction)
- ...

“Hello world” of neural networks

- MNIST (Modified National Institute of Standards and Technology)
- Handwritten digits
- 28 * 28 pixels
- 60 000 training images and 10 000 testing images



So what is a neural network?

Neural networks

$$y = f(\boldsymbol{x}) + \epsilon$$

- Neural networks are a way to specify $f(\boldsymbol{x})$
- You can display $f(\boldsymbol{x})$ graphically

- Let's graphically represent linear regression!

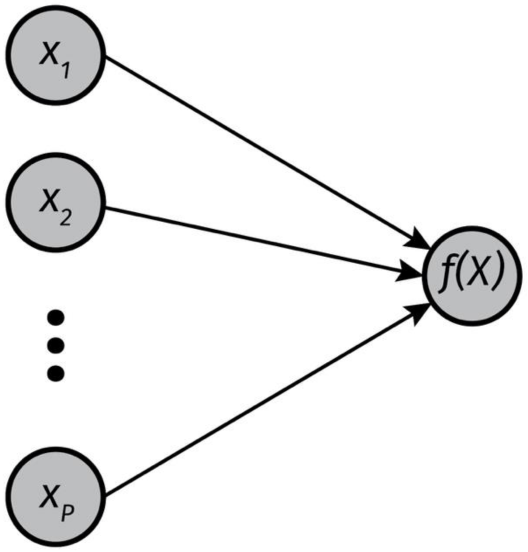
$$f(\boldsymbol{x}_i) = \sum_{p=1}^{P} \beta_p x_{pi}$$

Linear regression as neural net

Graphical representation

- Parameters are arrows
- Arrows ending in a node are summed together
- Intercept is not drawn

$$f(\boldsymbol{x}_i) = \alpha + \sum_{p=1}^{P} \beta_p x_{pi}$$

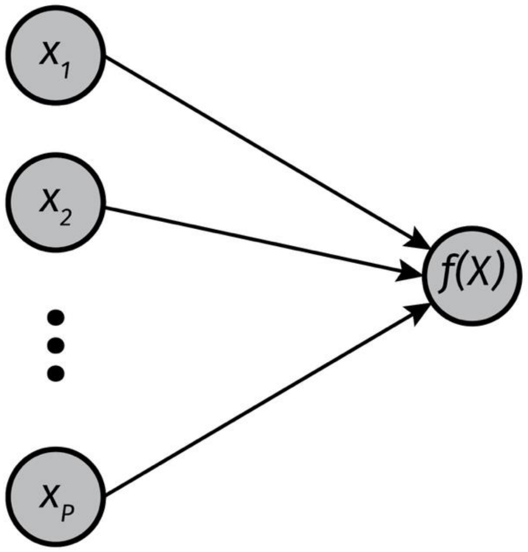


Linear regression as neural net

Neural network jargon

- Parameter = **weight**
- Intercept = **bias**

$$f(\boldsymbol{x}_i) = \boldsymbol{\beta} + \sum_{p=1}^{P} \boldsymbol{w}_p x_{pi}$$



Single layer neural networks

$$y = f(\boldsymbol{x}) + \epsilon$$

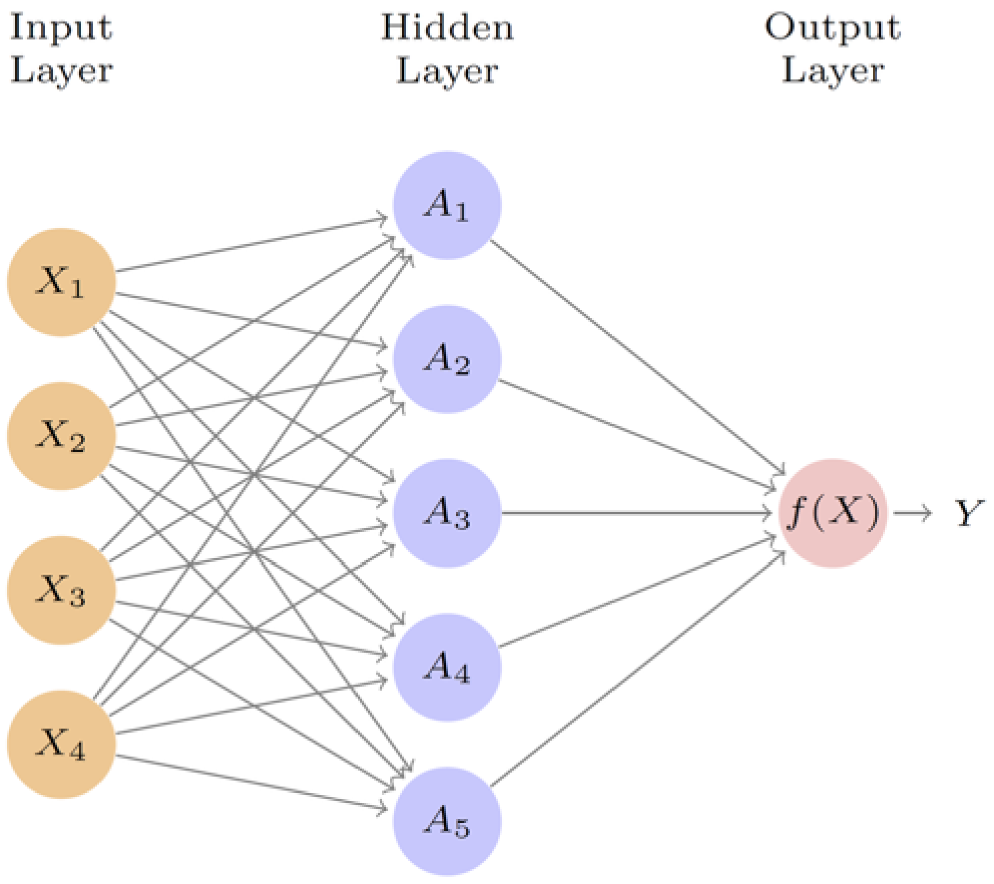
Specify a layer with K *hidden units* called A

$$f(\boldsymbol{x}) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$$

Where

$$A_k = h_k(\boldsymbol{x}) = g\left(w_{0k} + \sum_{p=1}^{P} w_{pk} x_p\right)$$

Single layer neural networks

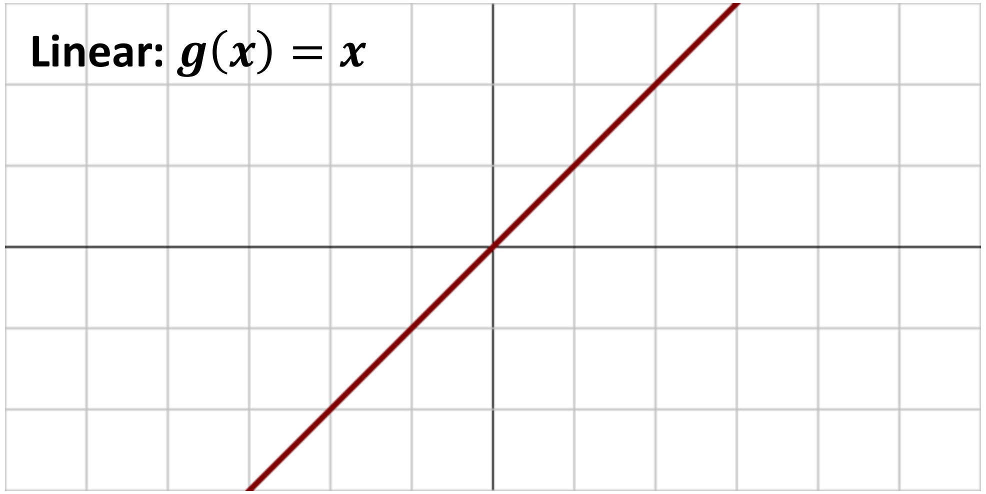


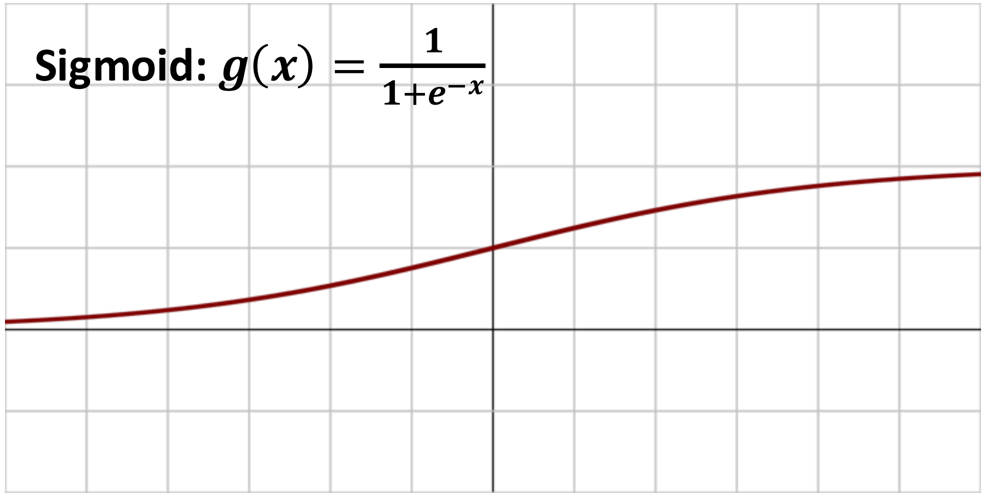
Single layer neural networks

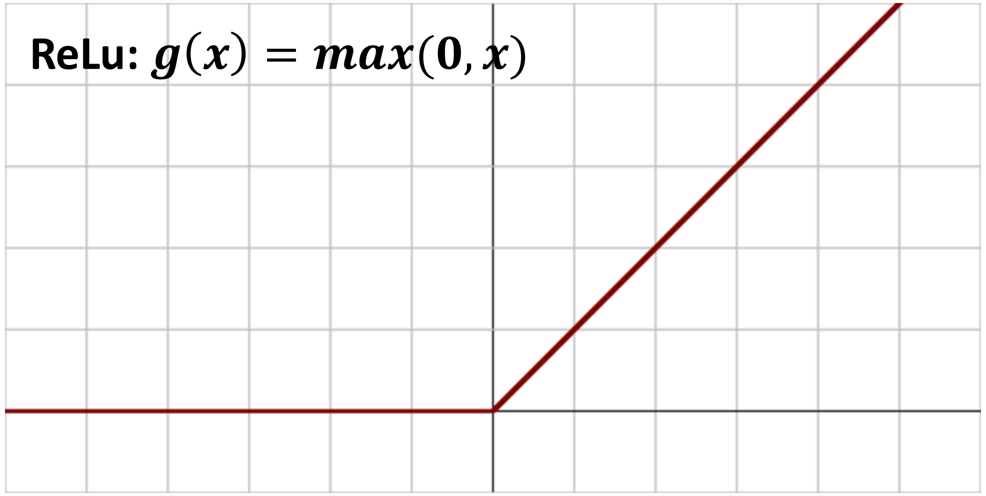
- What about the function $g(\cdot)$?
- This is called the **activation function**
- A transformation of the linear combination of predictors

$$h_k(\boldsymbol{x}) = g\left(w_{0k} + \sum_{p=1}^{P} w_{pk} x_p\right)$$

Activation functions







- Rectified linear (ReLu) is most popular nowadays
- Nonlinearity necessary! Otherwise: collapse to linear regression

Activation functions

We can go wider

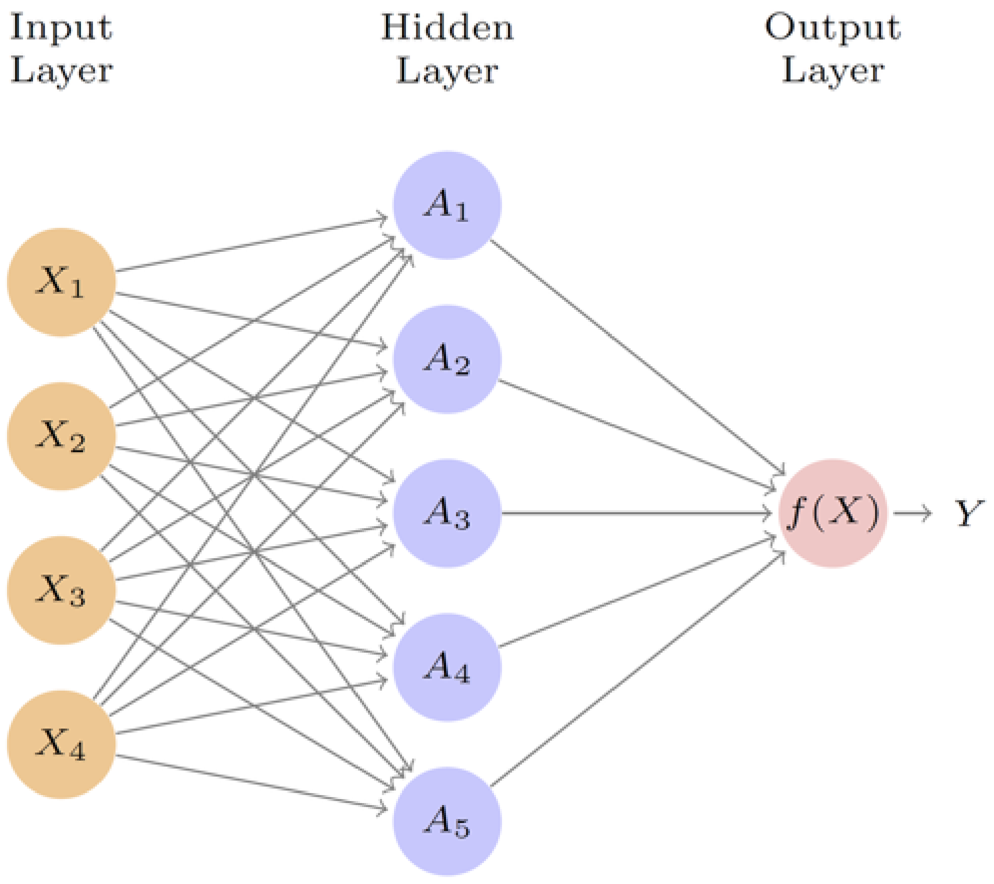
- More hidden units -> more transformations of input
- Similar to basis functions, feature engineering

Universal function approximation theorem

Any “well-behaved” function can be represented by neural net of sufficient width with nonlinear activation function

(you may need an inconvenient number of hidden units!)

Single layer neural networks



Let’s take it further

Feed-forward neural networks

We can go deeper

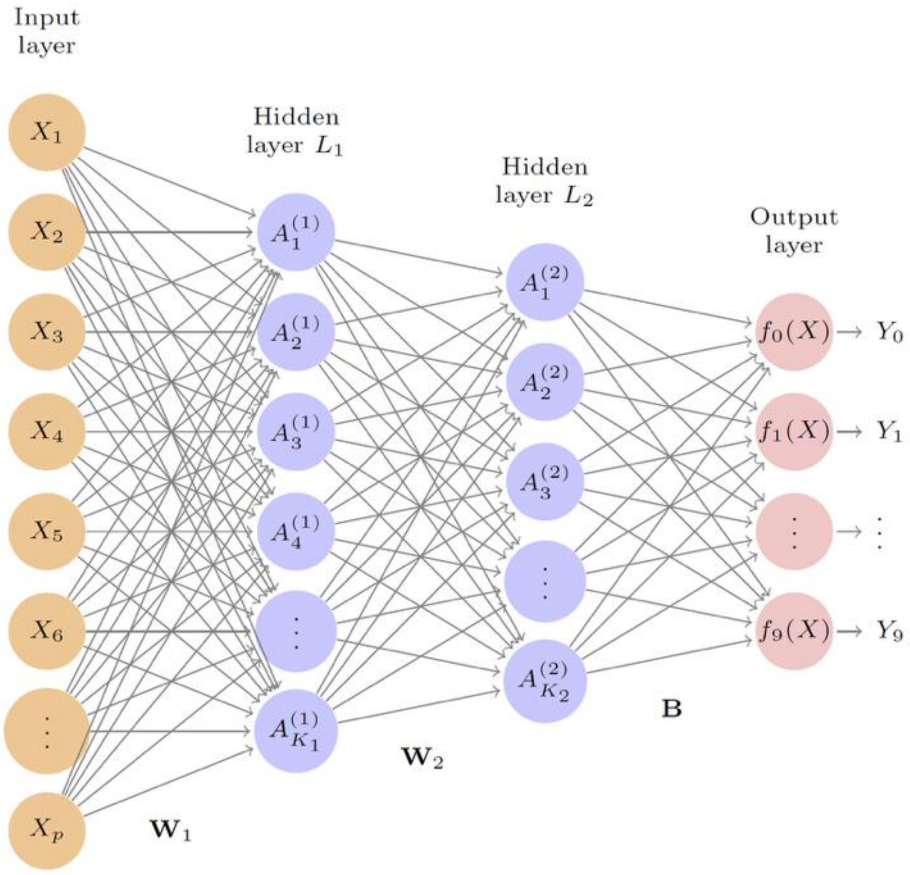
- More hidden layers after one another
- Higher-order features composed of lower-order features

Universal function approximation theorem, version 2

Any “well-behaved” function can be represented by neural net of sufficient *depth* with nonlinear activation function

(deep neural nets may be more tractable than wide)

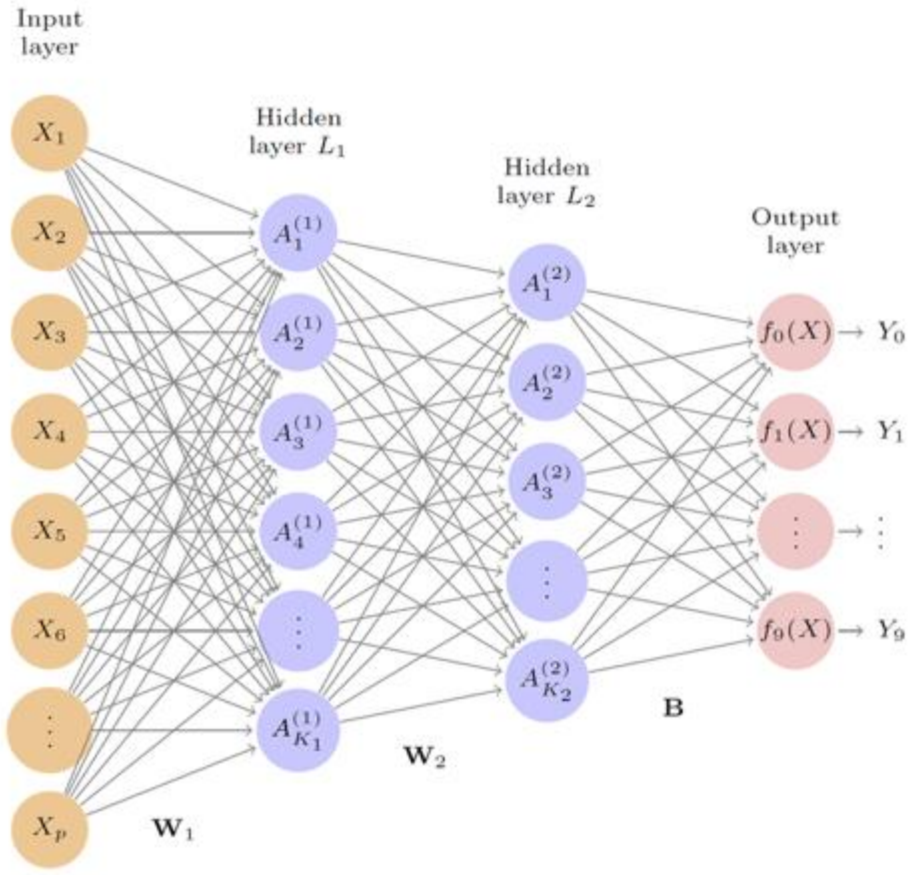
Feed-forward neural networks



Feed-forward neural networks

Feed-forward network architecture defined by:

- Number of layers
- Number of hidden units in each layer
- Activation function for each layer
- Activation function for output layer



Prediction for MNIST

Each example has:

- 28*28 = 784 input features
  - Values between 0-255 (8 bit)
  - Usually normalized to be 0-1
  - 1 = black, 0 = white, 0.5 = grey
- 10 outcome categories (0-9)
  - One-hot encoding for outcome
  - (cool way to say dummy coding)
  - 1 = 0 1 0 0 0 0 0 0 0 0
  - 5 = 0 0 0 0 0 1 0 0 0 0

Keras!

```r
library(keras)

model_dff <-
  keras_model_sequential() %>%
  layer_flatten(input_shape = c(28, 28)) %>%
  layer_dense(units = 256, activation = "relu") %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dense(10, activation = "softmax")
```

Keras!

```
summary(model_dff)

Layer (type)                        Output Shape                       Param #
==========================================================================================
flatten (Flatten)                   (None, 784)                        0

__________________________________________________________________________________________
dense_1 (Dense)                     (None, 256)                        200960

__________________________________________________________________________________________
dense_2 (Dense)                     (None, 128)                        32896

__________________________________________________________________________________________
dense_3 (Dense)                     (None, 10)                         1290
==========================================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0

__________________________________________________________________________________________
```

How to estimate parameters?

Estimating parameters

- We need some way to measure how well the network does
- Parameters that make the network perform well are good!

- Remember ML estimation: finding $\hat{\theta}$ maximizing $p(y|\hat{\theta})$
- Remember OLS estimation: finding $\hat{\beta}$ minimizing $\sum(y - X\hat{\beta})^2$
- Same for neural nets: we minimize some loss function $L(\theta)$

Loss function

- For continuous outcomes you can use squared error (same as linear regression!)

$$L(\theta) = (f(X_i; \theta) - y_i)^2$$

- For binary outcomes you can use binary cross-entropy (same as logistic regression!)

$$L(\theta) = -\left(y_i \log\left(f(X_i; \theta)\right) + (1 - y_i) \log\left(f(X_i; \theta)\right)\right)$$

Loss function

- What do the parameters need to be in order to minimize loss?
- We don’t know this!
- But we might know the *direction* in which we need to move to increase the loss
- This direction is called the **gradient** (of loss w.r.t parameters)

$$g(\theta) = \nabla_\theta = \frac{\partial}{\partial\theta} L(\theta)$$

- (Looks scary, but it’s just a number for each parameter)

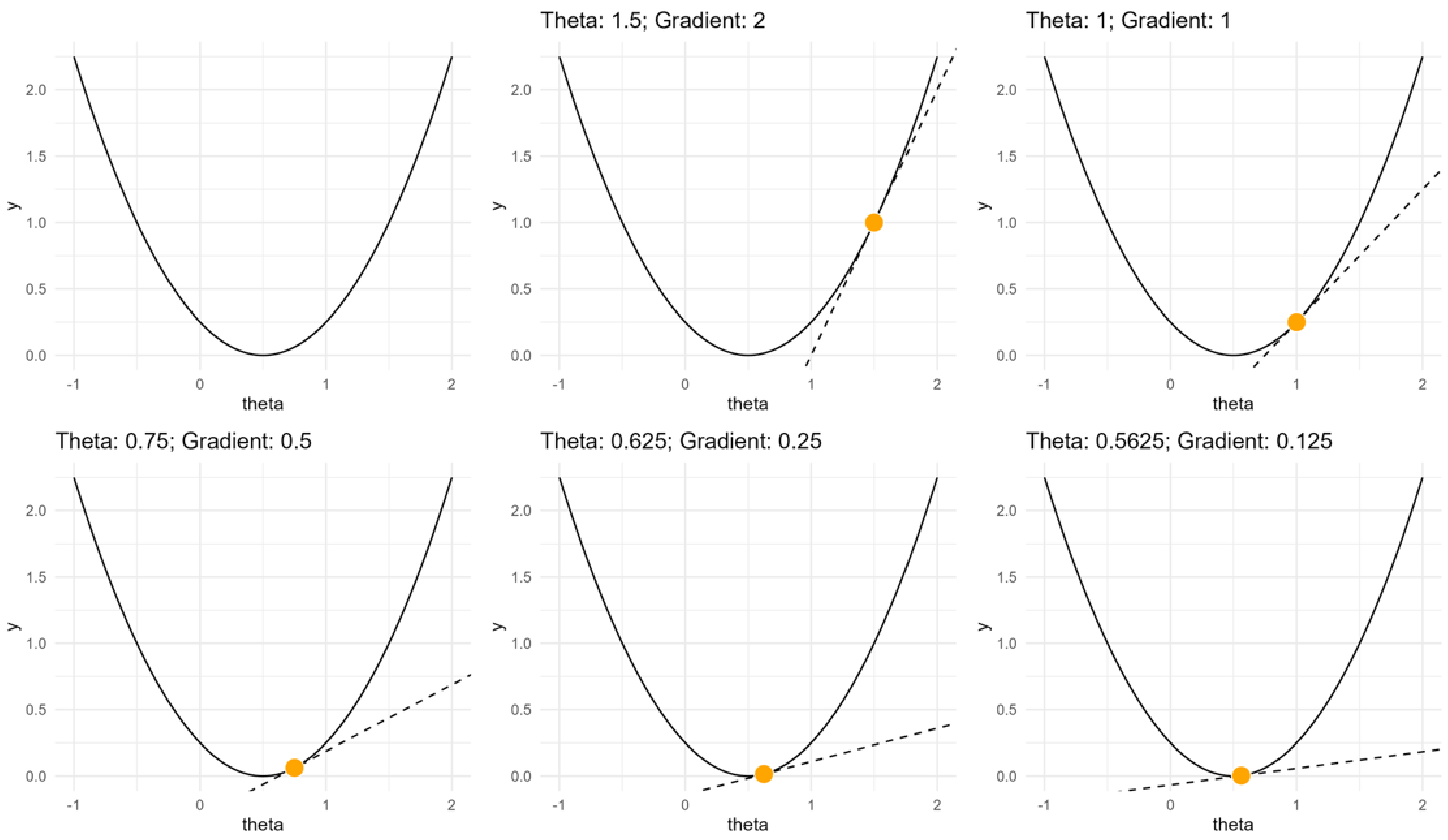
Gradient descent

Iteration: step of size λ in the direction of the negative gradient

$$\theta^{(j+1)} = \theta^{(j)} - \lambda \cdot g\left(\theta^{(j)}\right)$$

Let's try it out with a simple example!

- $L(\theta) = \theta^2 - \theta + 0.25$
- $g(\theta) = 2\theta - 1$
- $\lambda = 0.25$



Theta: 1.5; Gradient: 2
Theta: 1; Gradient: 1
Theta: 0.75; Gradient: 0.5
Theta: 0.625; Gradient: 0.25
Theta: 0.5625; Gradient: 0.125
y
theta
0.0
0.5
1.0
1.5
2.0
-1
0
1
2

Stochastic gradient descent

- Instead of computing the gradients w.r.t. the entire loss function, only use a random **batch** of data
- Take a step after each batch (e.g., 32 rows)
- If batch size = 1, take a step after each example
- Common batch sizes: 32, 64, 128, 256, 512
- One look at the full data = 1 **epoch**

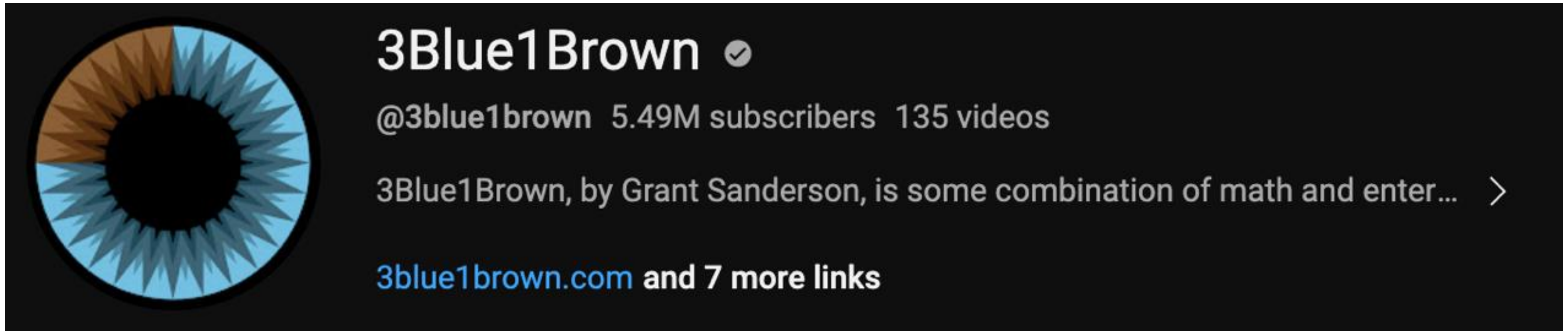
Gradient computation

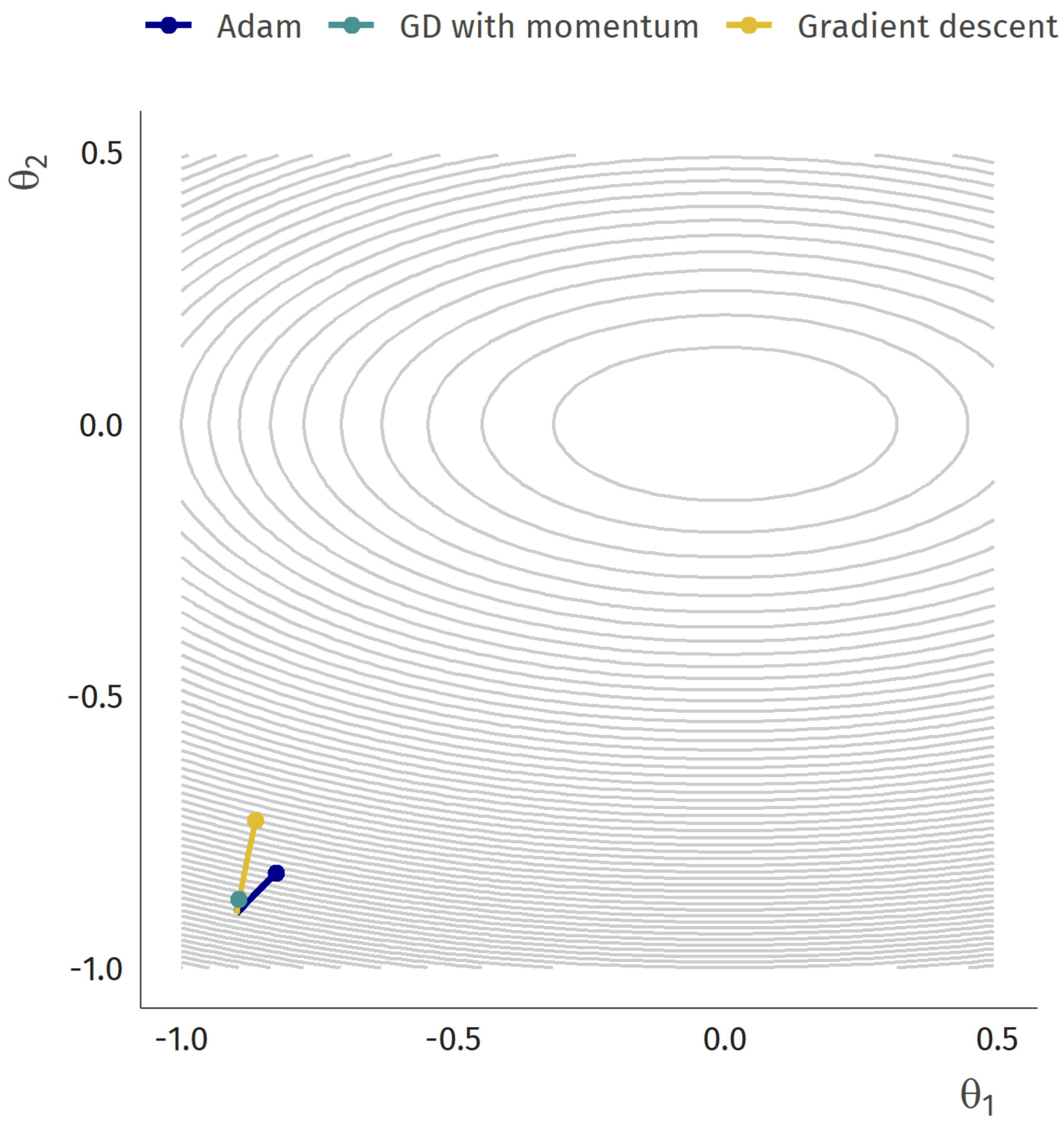
- But in neural networks, how do we compute gradients?
- We have functions of functions!
- Software like tensorflow / Keras / torch does this for you!
- **Backpropagation**: smart repeated use of the *chain rule* to compute derivatives

- Software also implements gradient descent (and friends)

Nice visual explanations

https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi





Adam
GD with momentum
Gradient descent
θ_2
0.5
0.0
-0.5
-1.0
-1.0
-0.5
0.0
0.5
θ_1

Programming pattern: training

```
model_dff %>%
  compile(
    loss = "sparse_categorical_crossentropy",
    optimizer = "adam"
  )

model_dff %>%
  fit(
    x = X,
    y = y,
    batch_size = 32,
    epochs = 10
  )
```

Conclusion: training

- We need a loss function (e.g., squared error)
- We need gradients (how to change θ to reduce $L(\theta)$)
- Gradient descent: take steps in direction of -gradient
- Stochastic GD: do this with data batches
- Software handles all of this (**black box!!**)

- Advantage: we can focus on the **architecture**

Different architectures

- By adjusting the arrows, layers, and activation functions, you can create models that are tailored to specific data, e.g.
- Convolutional (CNN): images, text, sound
- Recurrent (RNN): time series, text
- Graph (GNN): networks
- …

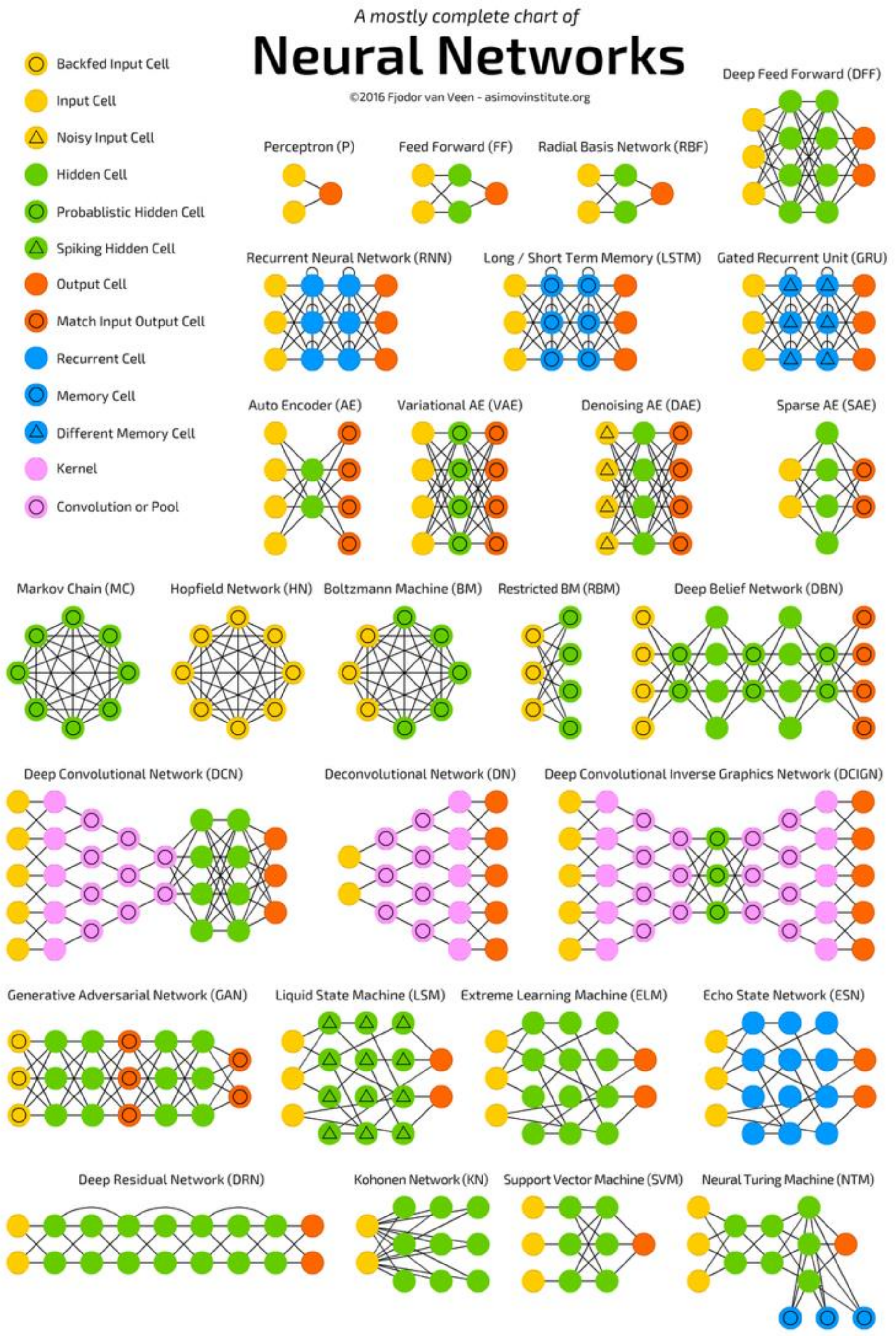
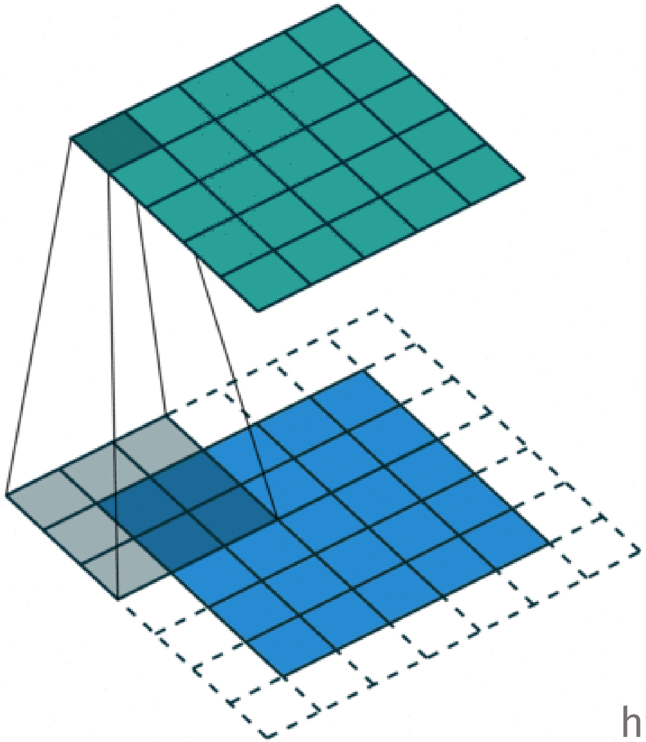


Image processing with convolutional neural networks

What is a convolution

- Convolution is applying a **kernel** (filter) over an image
- Filter defines which **feature** is important in the image



https://github.com/vdumoulin/conv_arithmetic

What is a convolution

$$\text{Original Image} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \\ j & k & l \end{bmatrix}.$$

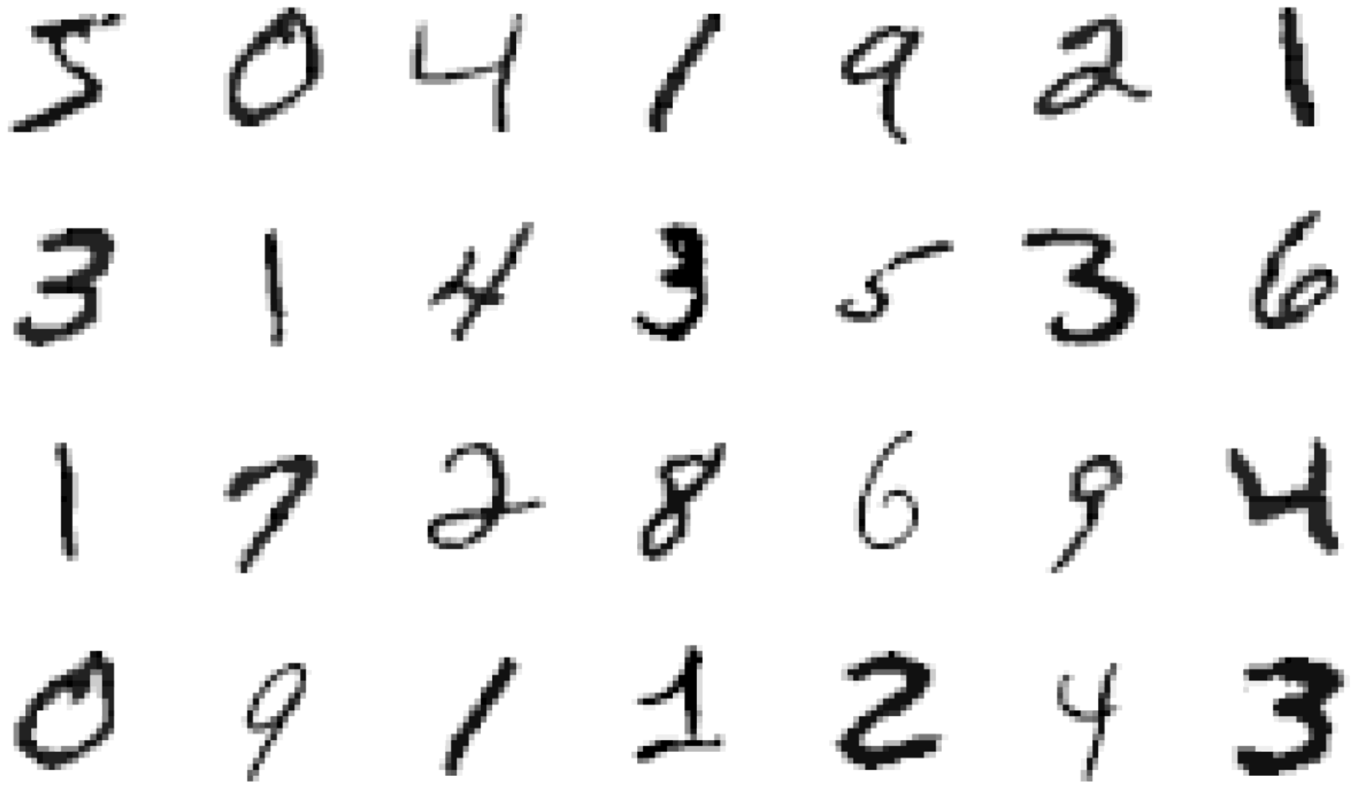
Now consider a 2×2 filter of the form

$$\text{Convolution Filter} = \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix}.$$

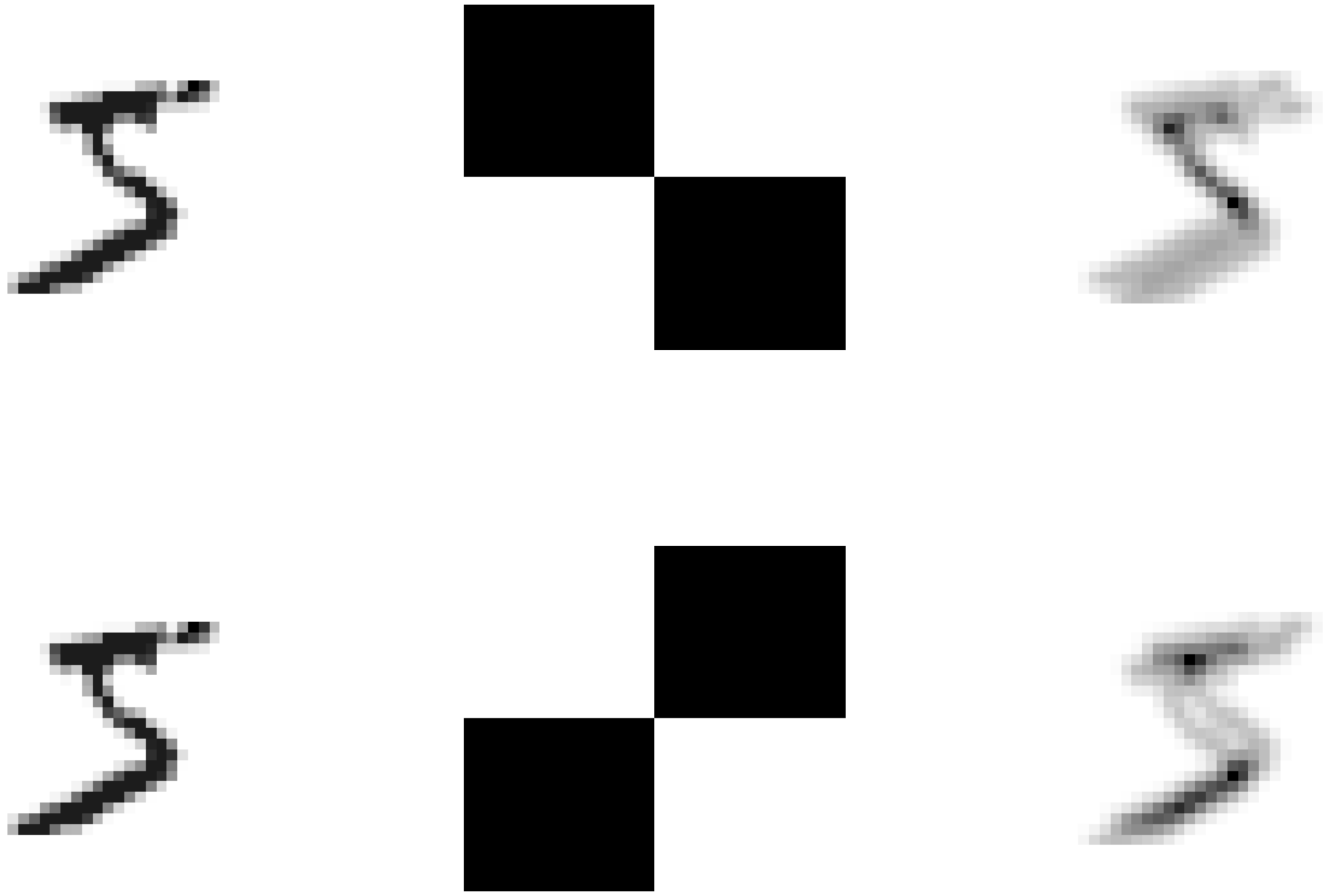
When we *convolve* the image with the filter, we get the result[8]

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

Back to MNIST



Detecting diagonal lines with convolution



Convolution layers

- A convolutional neural network is a NN with one or more **convolution layers**
- The parameters / weights in a convolution layer are the elements of the filter
- The filter is **learnt** by the network!

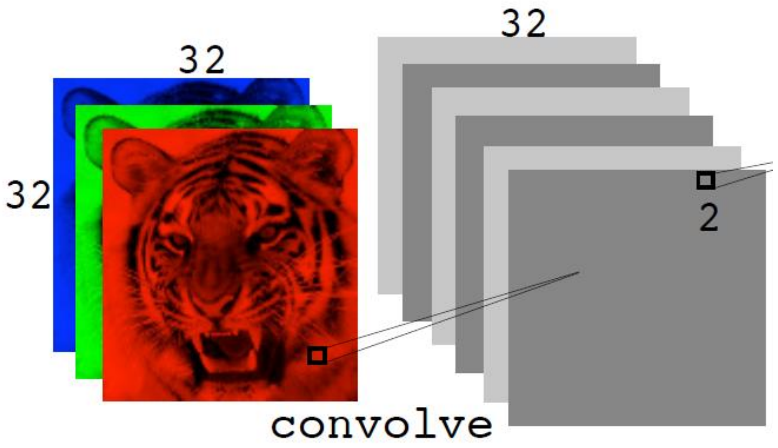


FIGURE 10.8. *Architectu*
Convolution layers are inte
size by a factor of 2 in both

Convolution layers

- In each convolution layer, you can create multiple filters
- Number of parameters is function of:
  - Number of filters (e.g. 6)
  - Size of each filter (e.g. 2x2)
  - NOT the input dimension!
- **Parameter sharing**

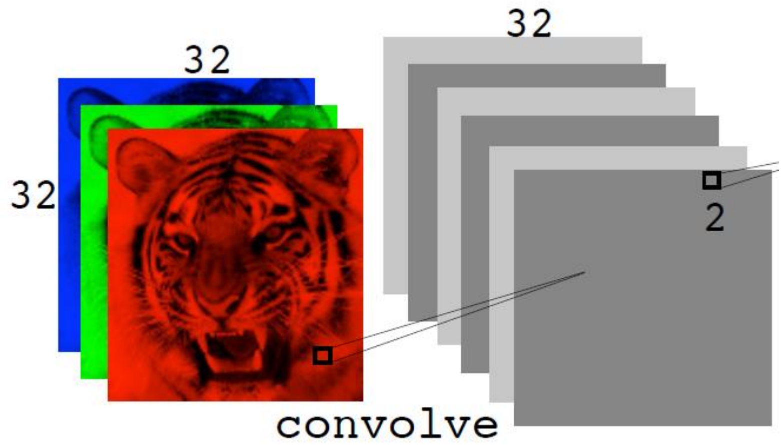


FIGURE 10.8. *Architectu*
Convolution layers are inte
size by a factor of 2 in both

Pooling layer

- Convolution layers are usually followed by a **pooling layer**
- Reduces dimensionality
- **Location invariance**: Robustness against pixel shift / small rotations
- **Max pool** most common

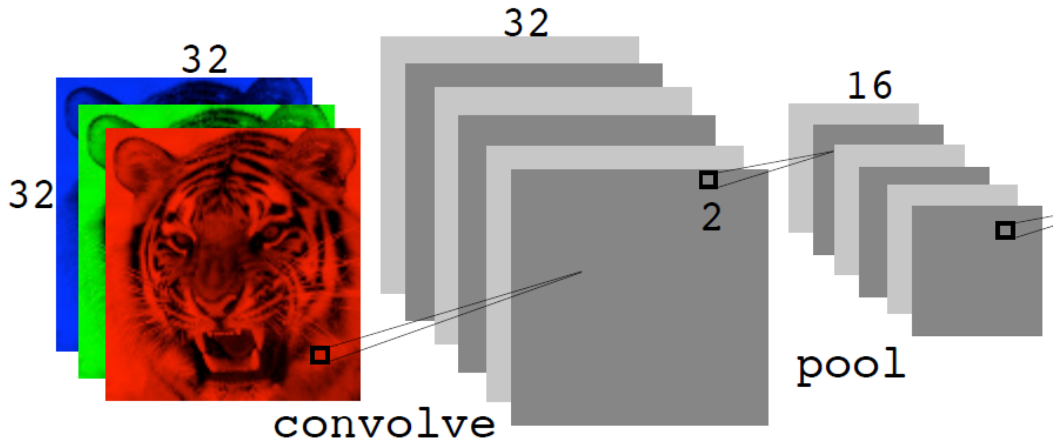


FIGURE 10.8. *Architecture of a*
Convolution layers are interspersed
size by a factor of 2 in both

Pooling layer

$$\text{Max pool} \begin{bmatrix} 1 & 2 & 5 & 3 \\ 3 & 0 & 1 & 2 \\ 2 & 1 & 3 & 4 \\ 1 & 1 & 2 & 0 \end{bmatrix} \to \begin{bmatrix} 3 & 5 \\ 2 & 4 \end{bmatrix}.$$

Architecture of a CNN

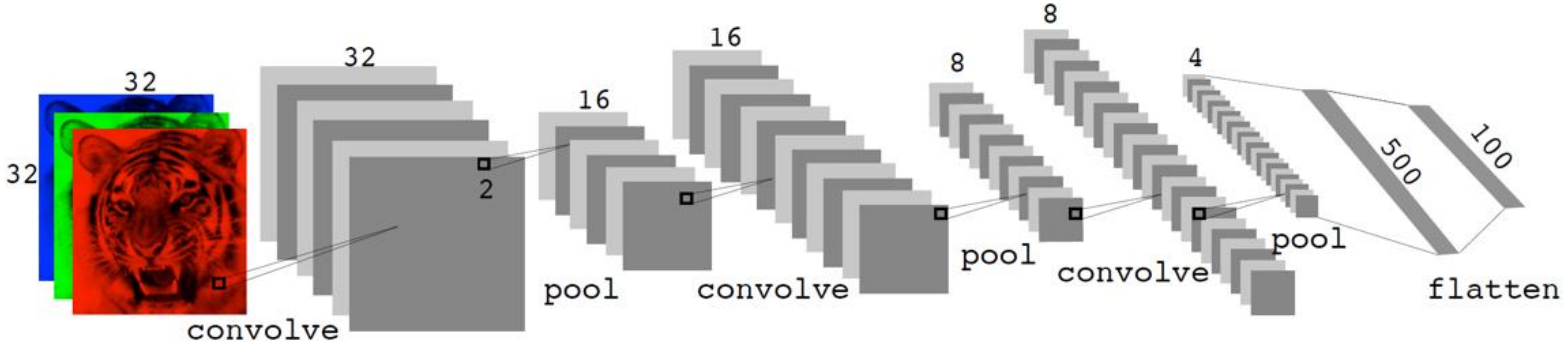


FIGURE 10.8. *Architecture of a deep CNN for the* CIFAR100 *classification task. Convolution layers are interspersed with* 2×2 *max-pool layers, which reduce the size by a factor of 2 in both dimensions.*

Applying CNN to MNIST

```
model_cnn <-
  keras_model_sequential(input_shape = c(28, 28, 1)) %>%
  layer_conv_2d(6, c(5, 5)) %>%
  layer_max_pooling_2d(pool_size = c(4, 4)) %>%
  layer_flatten() %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dense(10, activation = "softmax")
```

Model summary

```
summary(model_cnn)
Layer (type)                        Output Shape                  Param #
==============================================================================
conv2d_2 (Conv2D)                   (None, 24, 24, 6)             156
______________________________________________________________________________
max_pooling2d_2 (MaxPooling2D)      (None, 6, 6, 6)               0
______________________________________________________________________________
flatten_3 (Flatten)                 (None, 216)                   0
______________________________________________________________________________
dense_8 (Dense)                     (None, 32)                    6944
______________________________________________________________________________
dense_7 (Dense)                     (None, 10)                    330
==============================================================================
Total params: 7,430
Trainable params: 7,430
Non-trainable params: 0
```

Compare to feed-forward model

```
summary(model_dff)

Layer (type)                        Output Shape                      Param #
==========================================================================
flatten (Flatten)                   (None, 784)                       0
__________________________________________________________________________
dense_1 (Dense)                     (None, 256)                       200960
__________________________________________________________________________
dense_2 (Dense)                     (None, 128)                       32896
__________________________________________________________________________
dense_3 (Dense)                     (None, 10)                        1290
==========================================================================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
__________________________________________________________________________
```

Applying CNN to MNIST

```
model_cnn %>%
  compile(
    loss = "sparse_categorical_crossentropy",
    optimizer = "adam"
  )

model_cnn %>%
  fit(
    x = mnist$train$x,
    y = mnist$train$y,
    epochs = 10,
    validation_split = 0.2,
    verbose = 2
  )
```

Performance comparison: DFF

```
    pred
obs     0    1    2    3    4    5    6    7    8    9
  0   975    0    1    0    0    1    0    0    2    1
  1     0 1129    1    1    0    1    1    2    0    0
  2     4    0 1015    2    0    0    3    2    6    0
  3     0    0    6  991    0    4    0    4    1    4
  4     3    2    2    0  952    0    5    2    0   16
  5     3    0    0   10    0  869    4    0    3    3
  6     6    2    0    1    2    4  942    0    1    0
  7     2    5    6    2    0    0    0 1004    3    6
  8     6    0    2    3    2    4    1    4  946    6
  9     4    3    0    2    3    1    1    3    1  991
```

Performance comparison: CNN

```
   pred
obs    0    1    2    3    4    5    6    7    8    9
  0  971    0    1    0    1    1    2    1    2    1
  1    0 1126    2    1    0    0    2    0    4    0
  2    1    1 1020    1    1    0    0    1    6    1
  3    0    0    2  997    0    5    0    1    2    3
  4    0    0    1    0  970    0    0    0    1   10
  5    2    0    0    3    0  881    3    0    2    1
  6    5    2    0    0    5    2  941    0    3    0
  7    1    3   15    3    0    1    0  994    3    8
  8    5    0    3    2    0    0    2    2  956    4
  9    1    1    0    1    4    5    0    2    6  989
```

Performance comparison: CNN

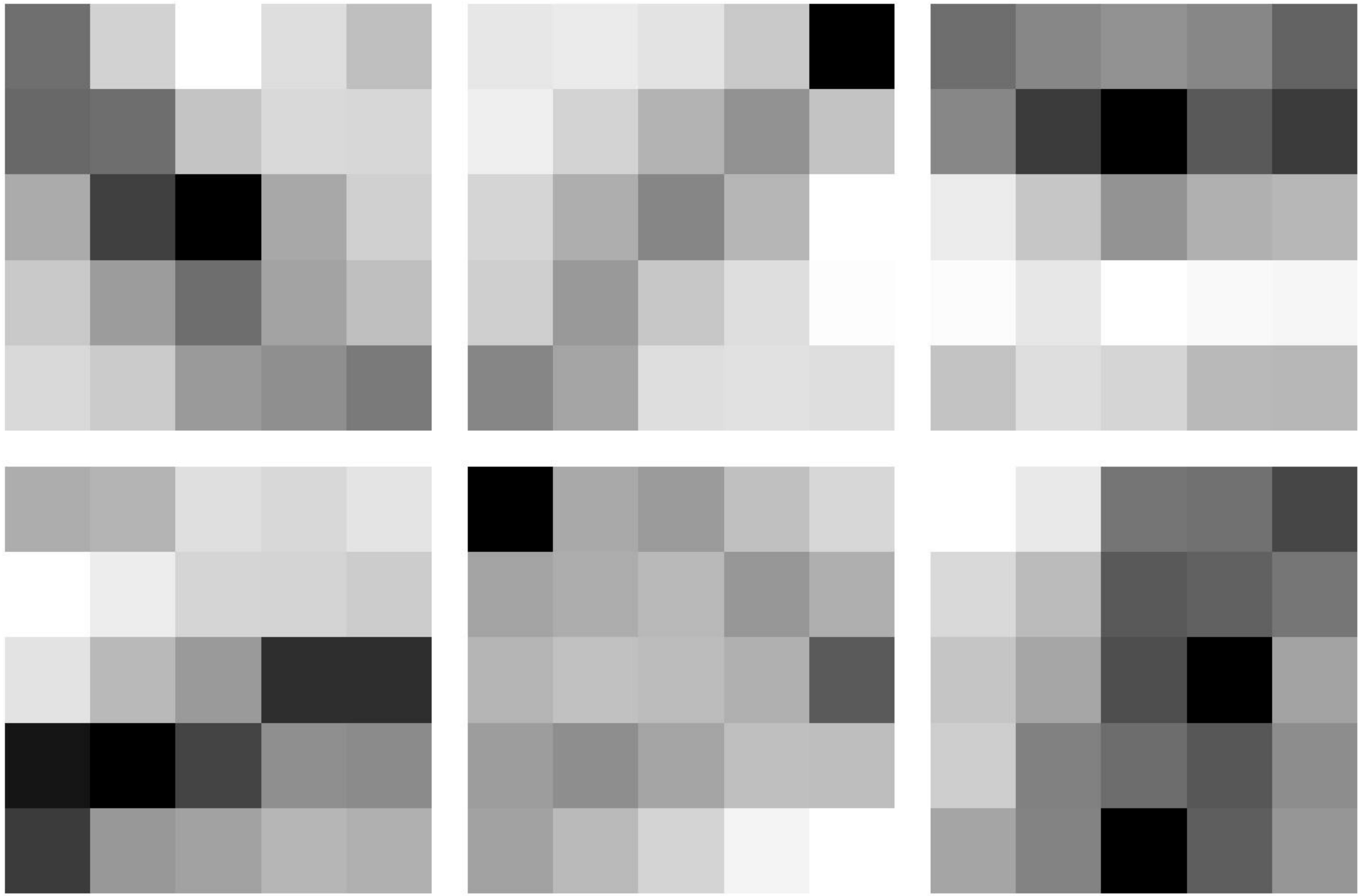
```
# accuracy
sum(diag(cmat_dff)) / sum(cmat_dff)

#> [1] 0.9814

sum(diag(cmat_cnn)) / sum(cmat_cnn)

#> [1] 0.9845
```

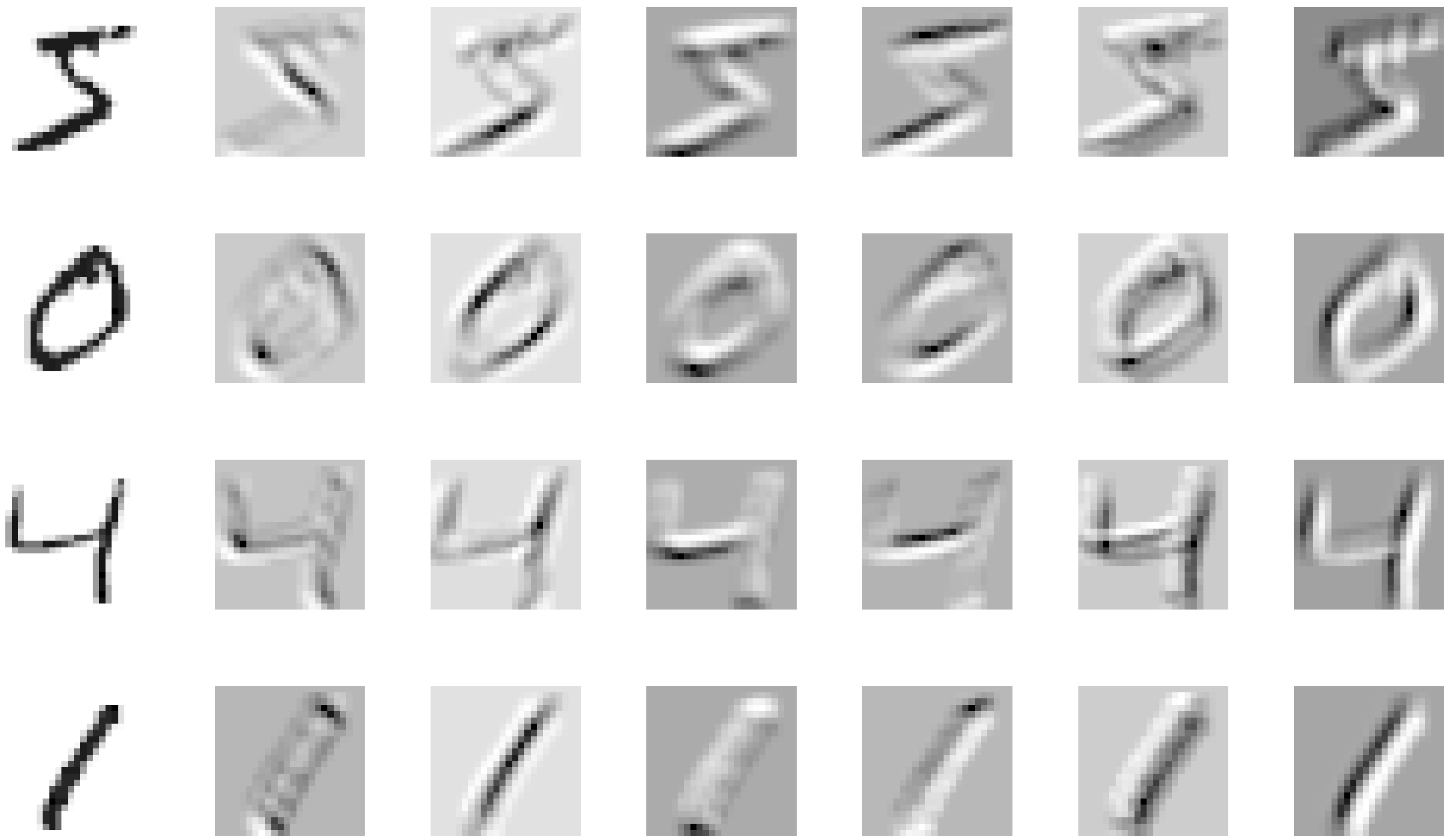
What about the features?

Unboxing the black box

- Extract the weights of the convolution layer to find the features (filters) that were learnt
- Apply the filters to some example images to get an idea of which features are discriminative for the different numbers

(There are other advanced methods, like layerwise relevance propagation, shapley values, …)

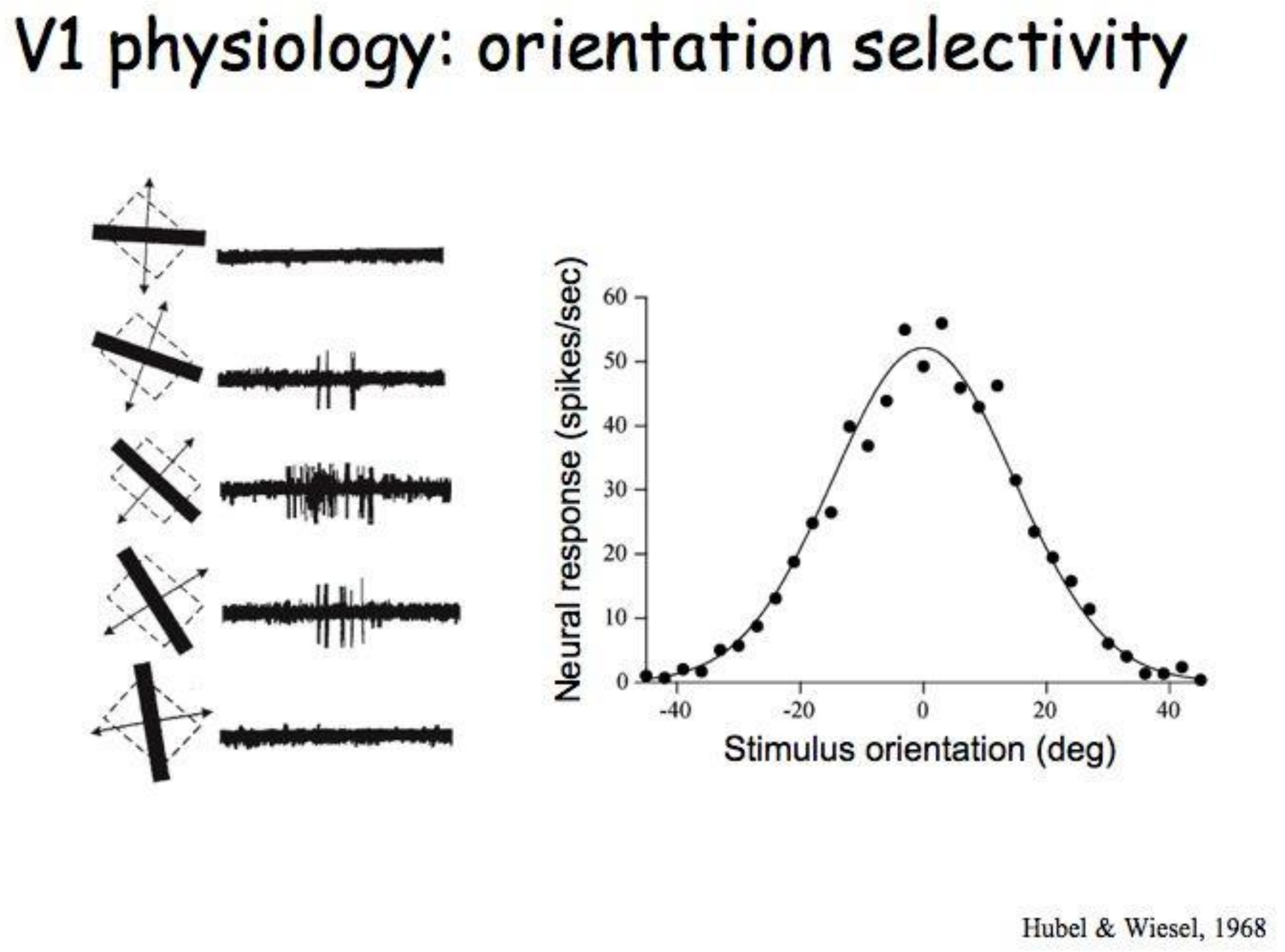




Similarity to visual brain area

These learnt filters are similar to monkey visual area 1 (V1) neuron sensitivities

http://www.cns.nyu.edu/~david/courses/perception/lecturenotes/V1/lgn-V1.html



Cool hack: pretrained CNNs

- Download the convolutional layer weights from existing neural network trained on many images
- Apply them to your own images
- Result: a feature vector per image
- Use these feature vectors as input dataset for:
  - Deep feedforward neural network
  - Logistic regression
  - Support vector machine
  - …
- This can work really well!!

Conclusion: CNN

- Convolution = applying kernel (filter) over an image
- CNNs employ convolution layers
  - Parameter sharing
  - Feature detection
- Followed by pooling layers
  - location invariance
- State-of-the art in image recognition
- Use pretrained networks as a quick proxy

Battling the curse of dimensionality

Regularization in NNs

- We may have thousands or even millions of parameters
- How can we avoid overfitting?
- How can we fight the curse of dimensionality?
- NNs are not magic: we need **regularization.**
  - Regularization is anything which introduces bias in the parameters to improve generalization (Goodfellow et al., 2016)

Regularization in NNs

- **Convolution**: parameters are set to be equal to one another in different areas of image (parameter sharing)
- **L1 or L2 penalty** applied to weights is common in neural networks (keras can do it!)
- **Dropout regularization**: In each iteration, only update a subset of the parameters
- **Early stopping**: Do not train for many epochs, but only until validation set loss does not improve
- **Data augmentation**: Add shifted / rotated versions of images to input (upside-down tiger is still a tiger!)

Conclusion

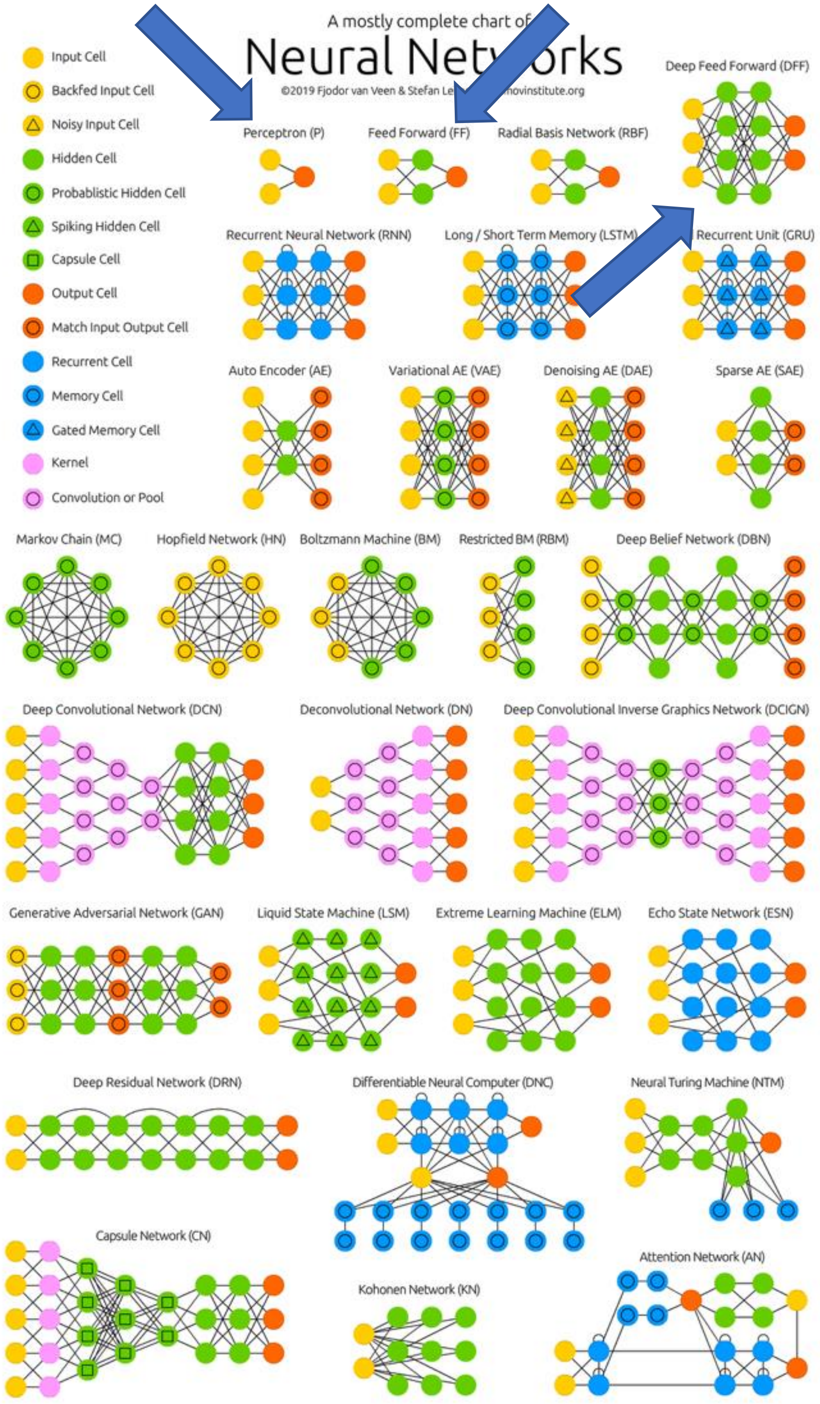
- Introduction to neural networks
- Feed-forward & deep neural networks
- Estimation / optimization
- Convolutional neural networks
- Battling the curse of dimensionality

Epilogue: neural network zoo

Neural network zoo

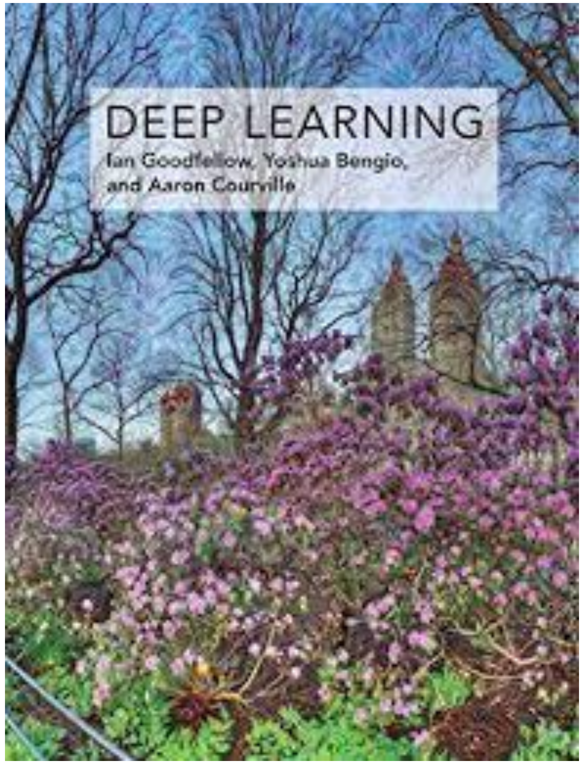
- You can see how far we got:
  - Perceptron (nonlinear regression)
  - Feed forward
  - Deep feed forward
  - Deep convolutional network

- There is much more ☺

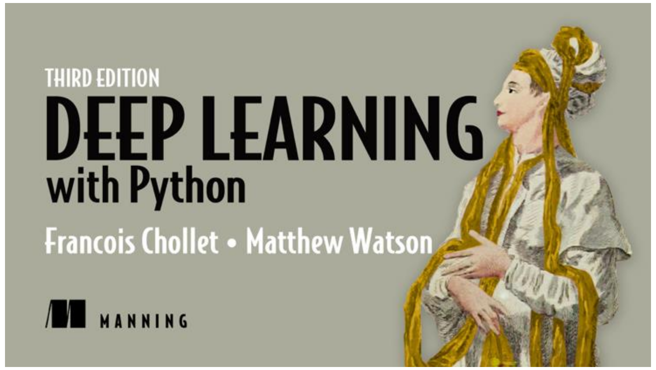


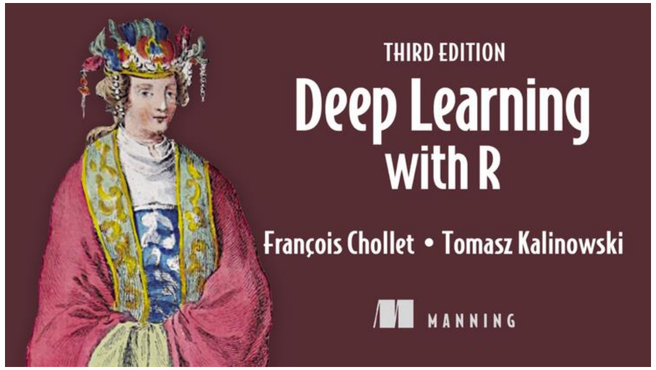
Deep learning in practice

- Good places to start:
  - https://keras.rstudio.com/
- ISLR Chapter 10



Goodfellow et al.





Chollet (R/Python version)

This was just the start

- Recurrent neural networks: for sequences
- Transformers (basis of all current LLMs)
- Autoencoders (nonlinear dimension reduction)
- Graph neural networks (GNNs; for “graph data”)
- Diffusion models
- …