Deep learning 1

Feed-forward neural networks

Daniel Oberski, Robert Bagheri

Today

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

Introduction

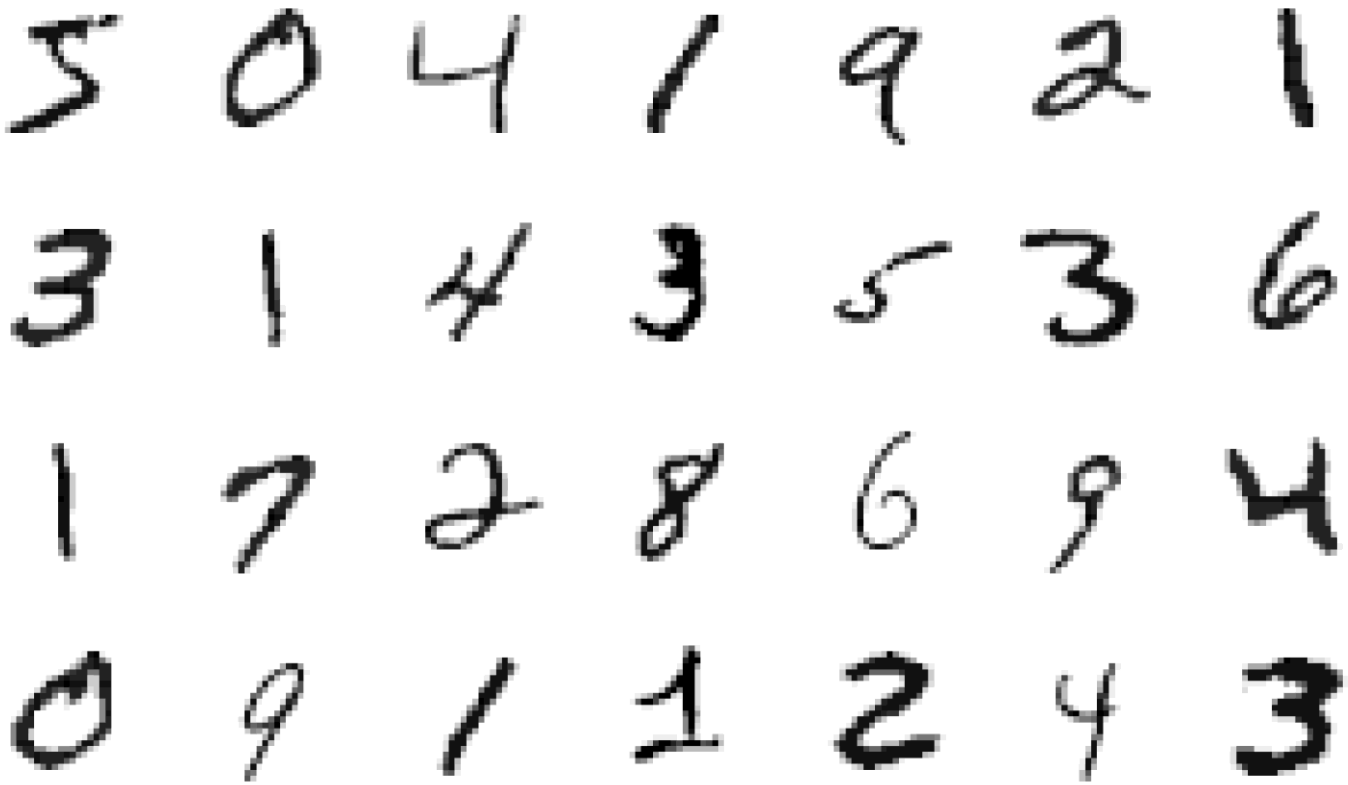
Why should we learn this?

State-of-the-art performance on various tasks

- Language models (obviously!)
- 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!

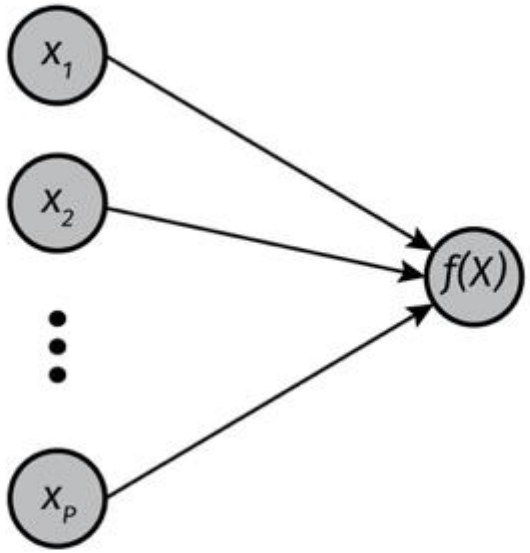
$$f(\boldsymbol{x}_i) = \sum_{p=1}^{P} \beta_p x_{pi}$$

Linear regression as neural net

Graphical representation

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

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

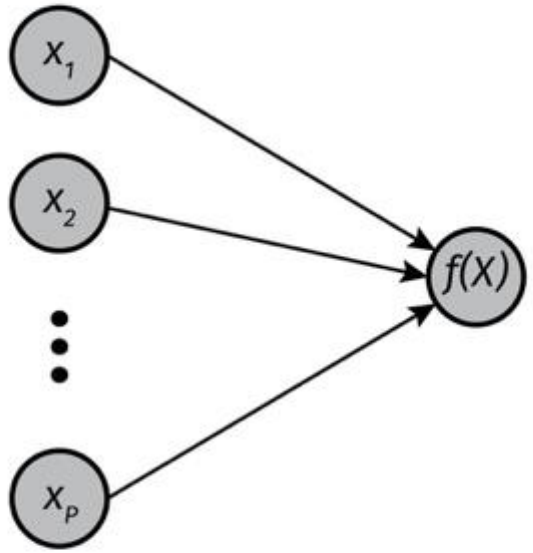


Linear regression as neural net

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

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

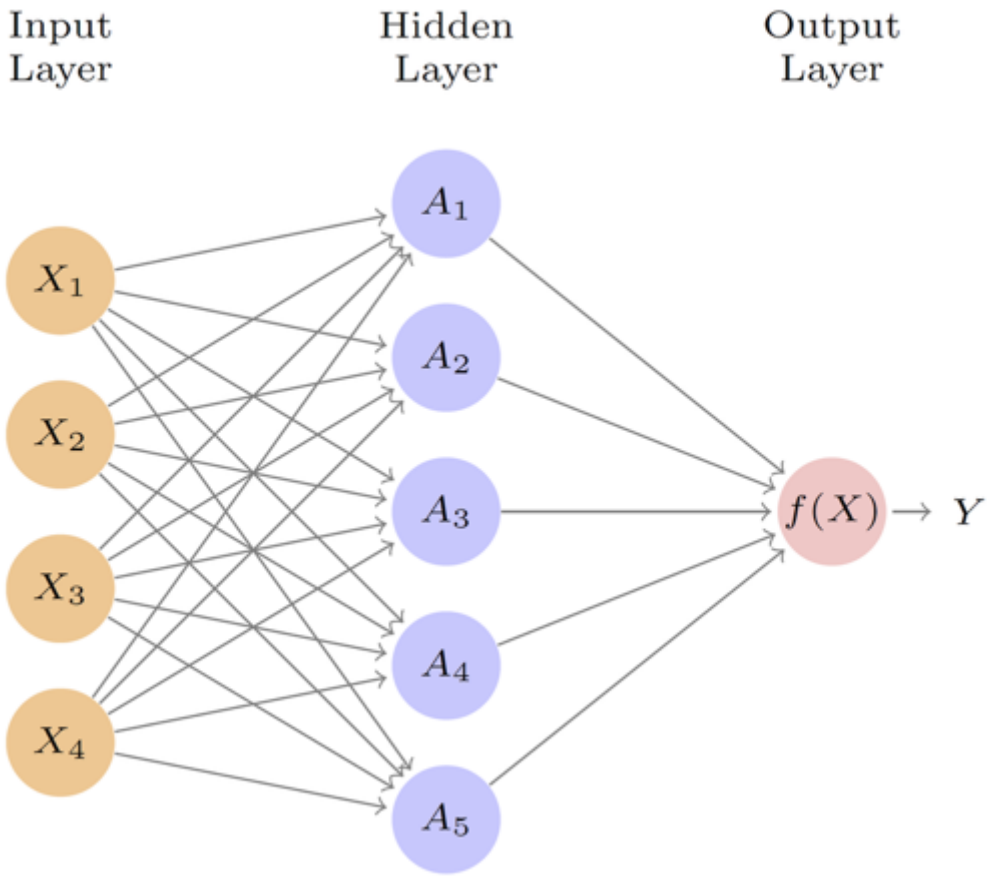
Specify a layer with K *hidden units* called A

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

Where

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

Single layer neural networks

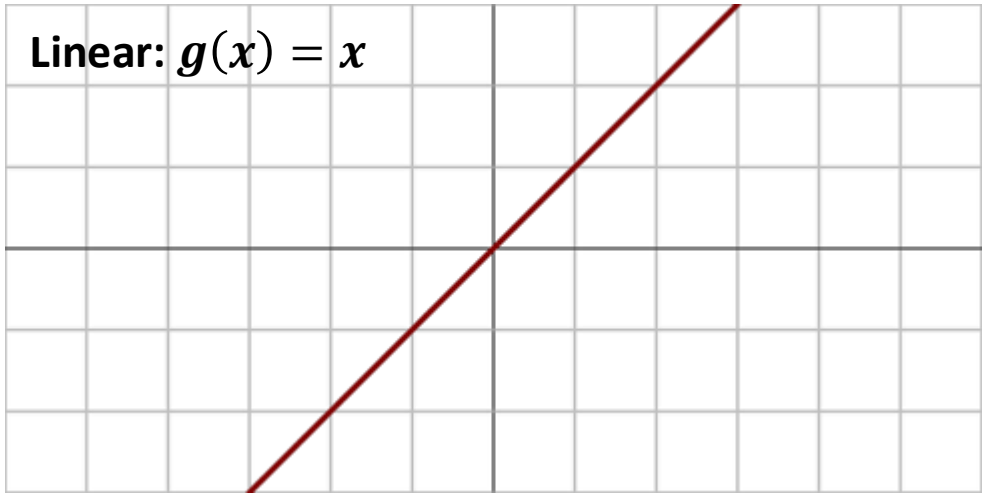


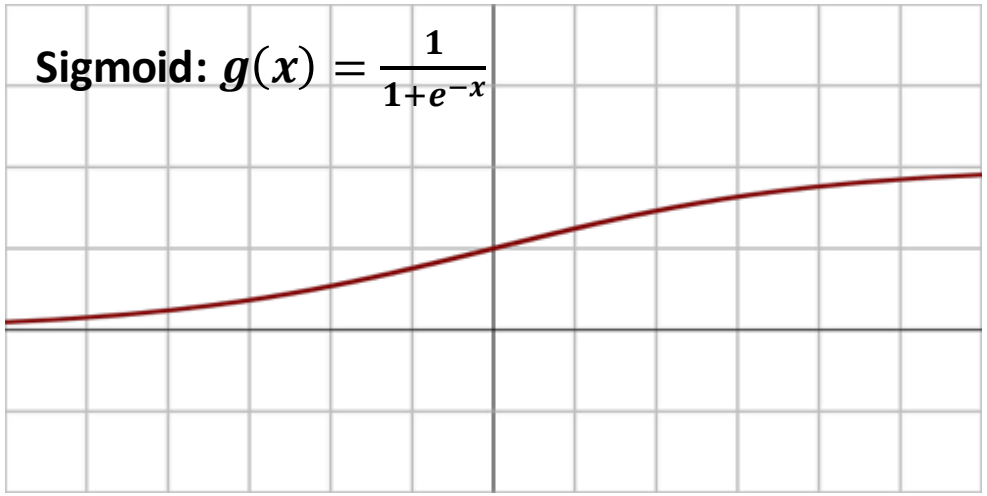
Single layer neural networks

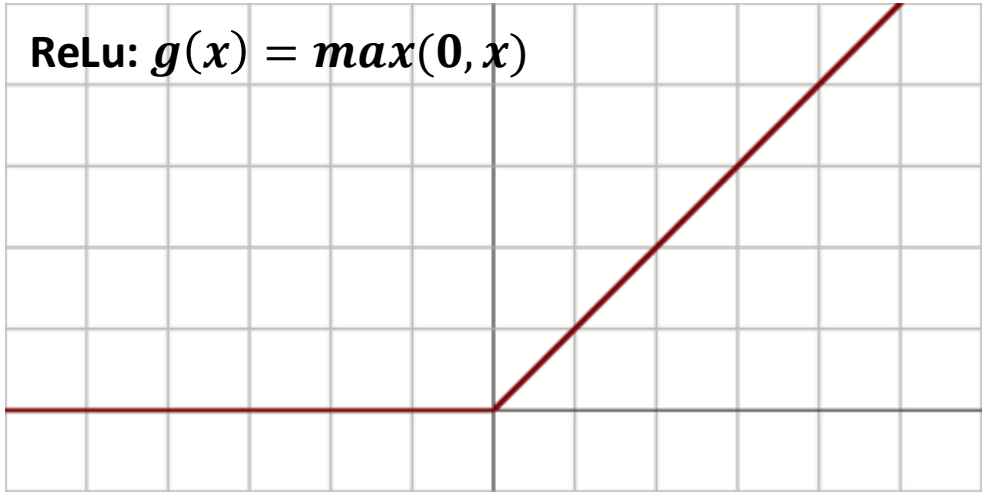
- What about the function $g(\cdot)$?
- This is called the **activation function**
- A transformation of the linear combination of predictors

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

Activation functions







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

Activation functions

We can go wider

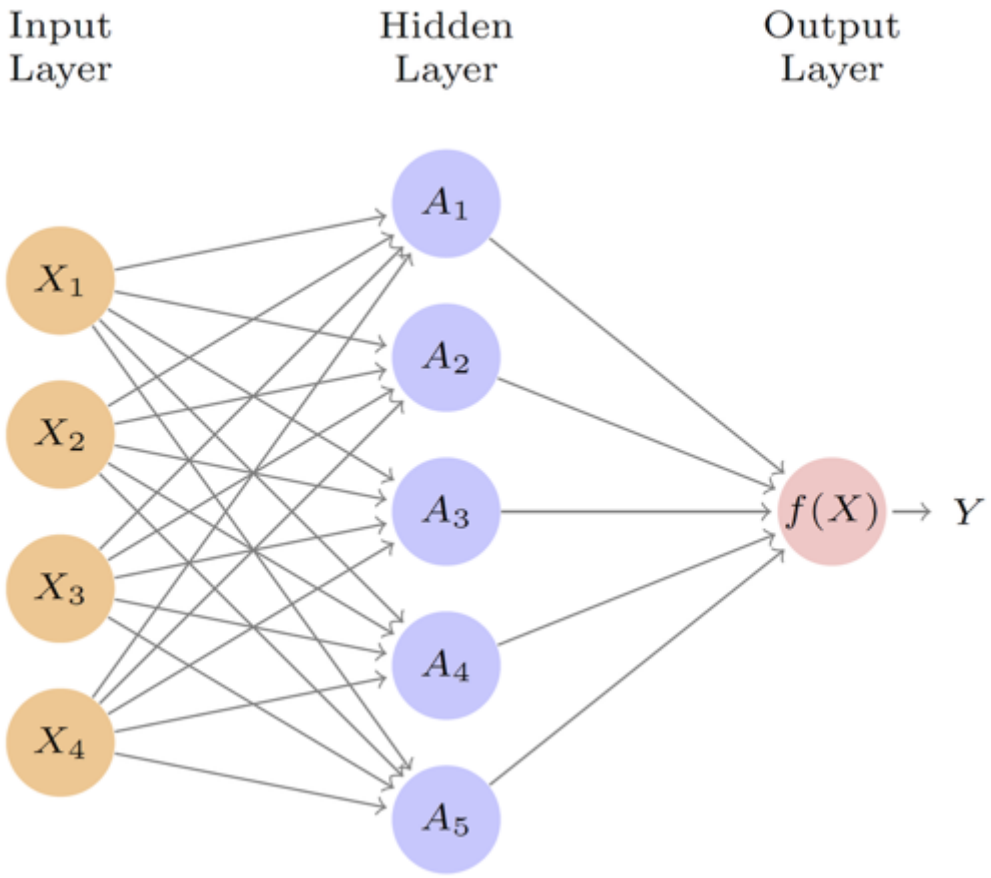
- More hidden units -> more transformations of input
- Similar to basis functions, feature engineering

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

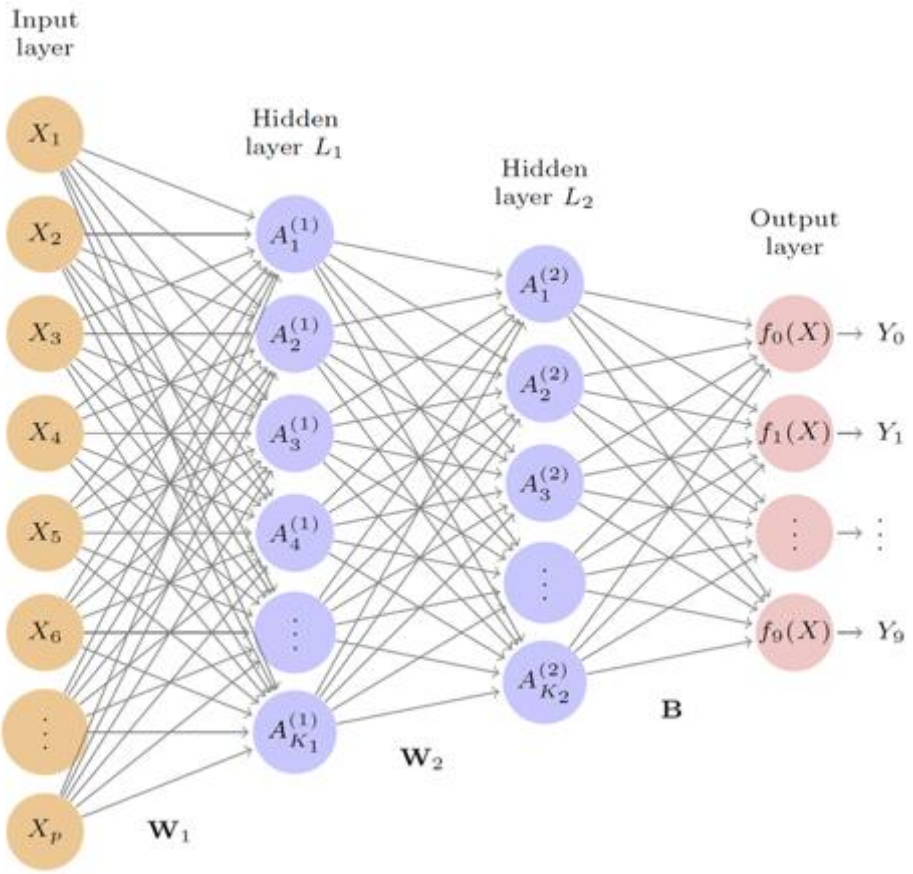
- More hidden layers after one another
- Higher-order features composed of lower-order features

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)

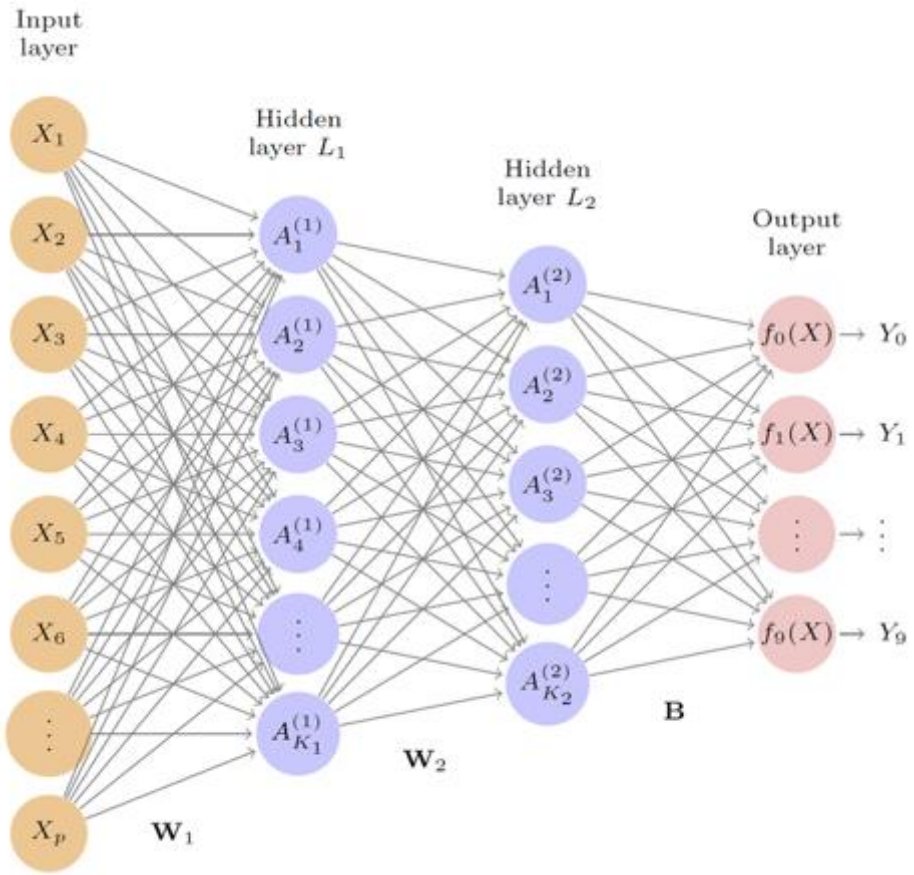
Feed-forward neural networks



Feed-forward neural networks

Feed-forward network architecture defined by:

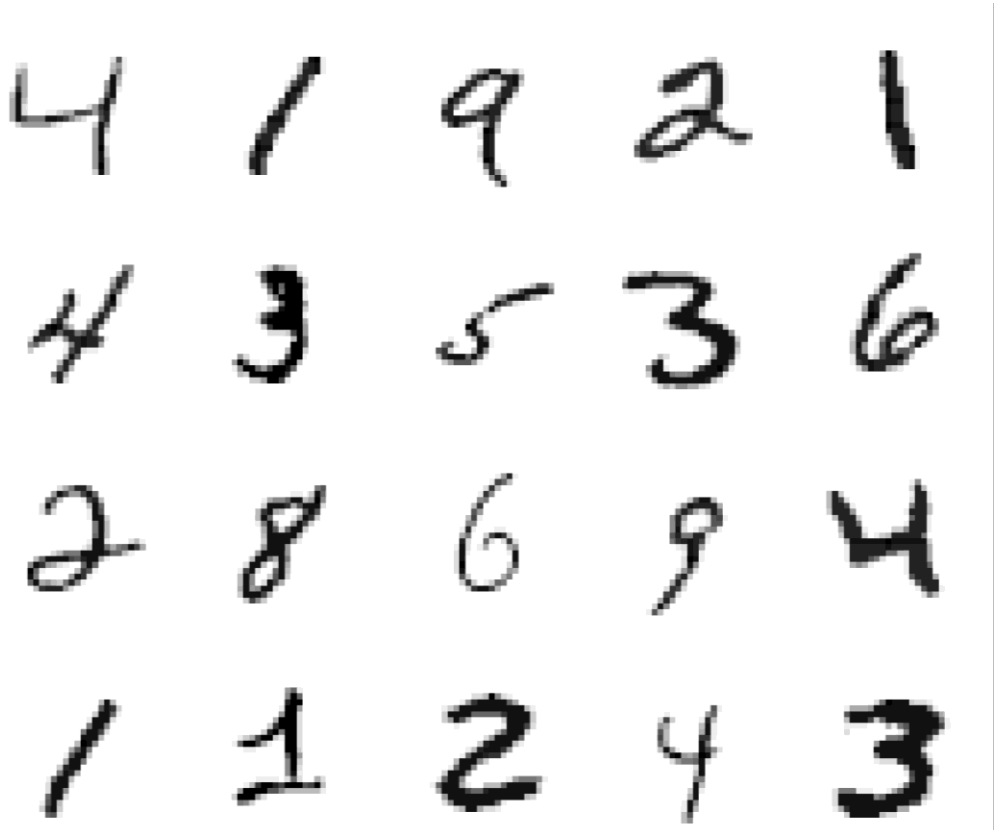
- Number of layers
- Number of hidden units in each layer
- Activation function for each layer
- Activation function for output layer



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!

```
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 train the model?
(estimate the values of 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!)

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

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)

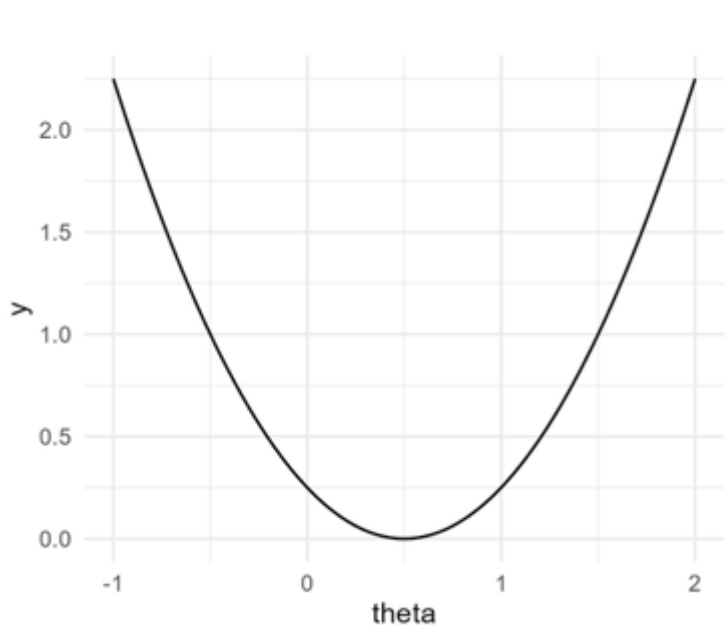
Gradient descent

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

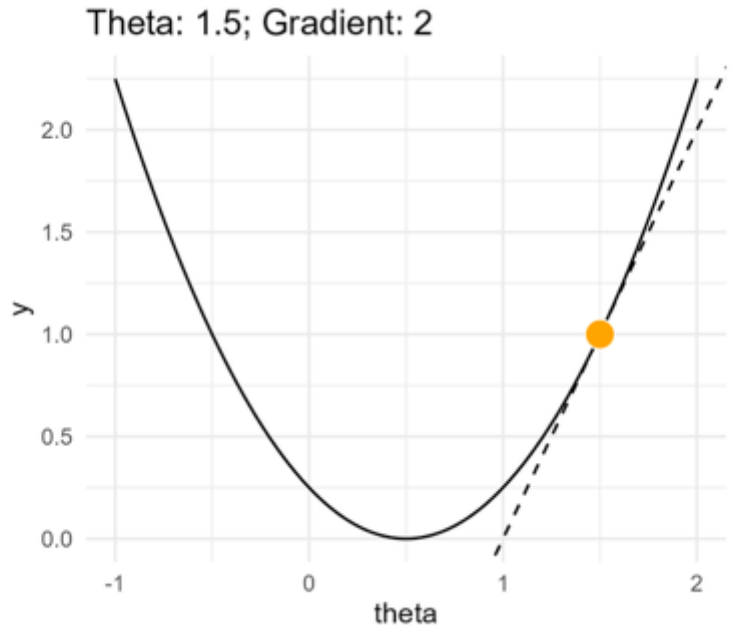
$$\theta^{(j+1)} = \theta^{(j)} - \lambda \cdot g\left(\theta^{(j)}\right)$$

Let's try it out with a simple example!

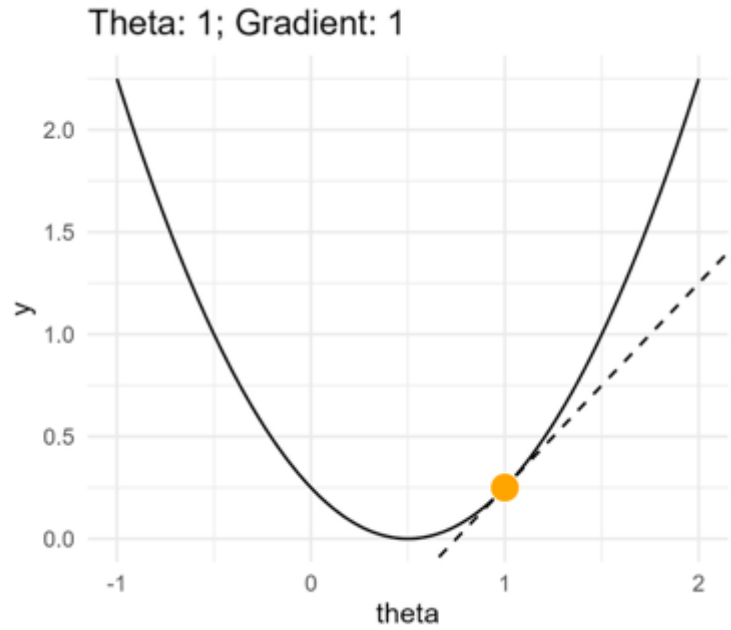
- $L(\theta) = \theta^2 - \theta + 0.25$
- $g(\theta) = 2\theta - 1$
- $\lambda = 0.25$



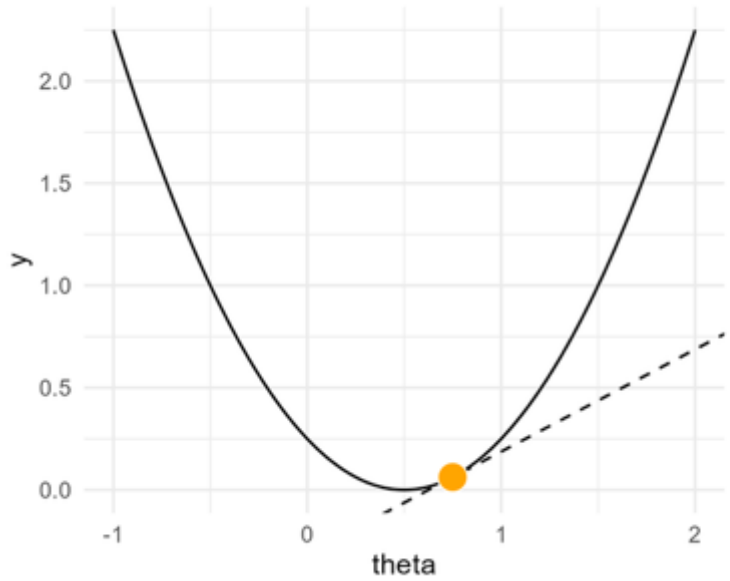
Theta: 1.5; Gradient: 2



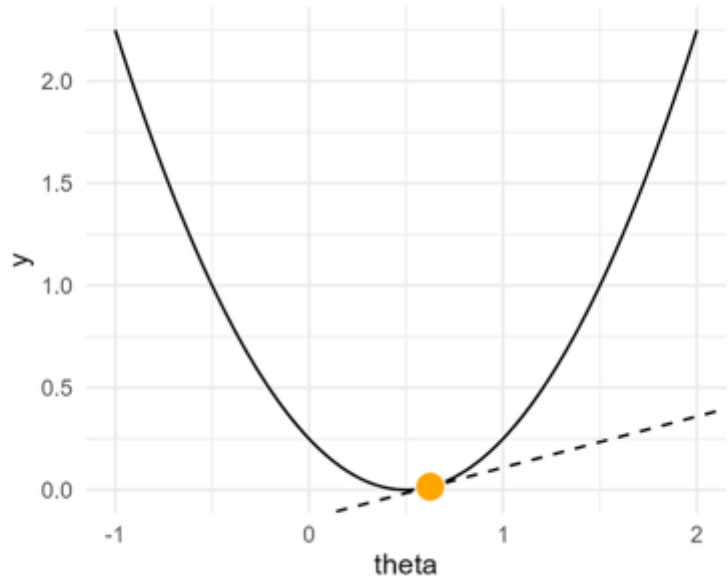
Theta: 1; Gradient: 1



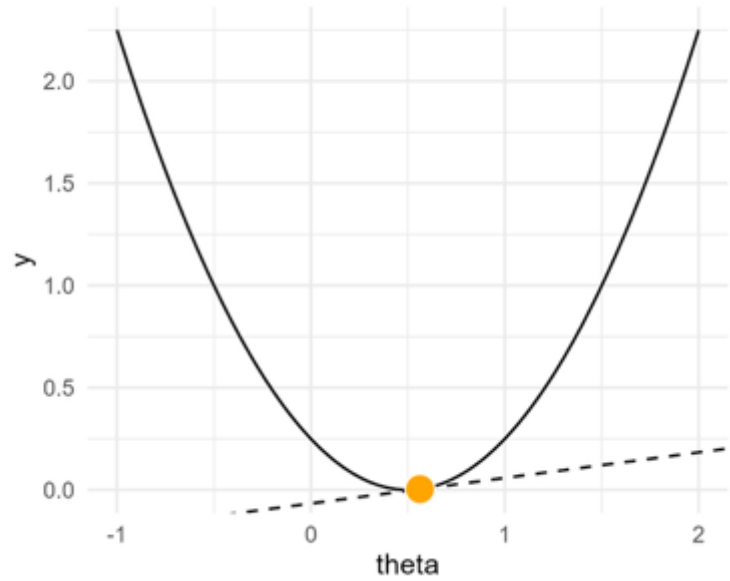
Theta: 0.75; Gradient: 0.5



Theta: 0.625; Gradient: 0.25



Theta: 0.5625; Gradient: 0.125



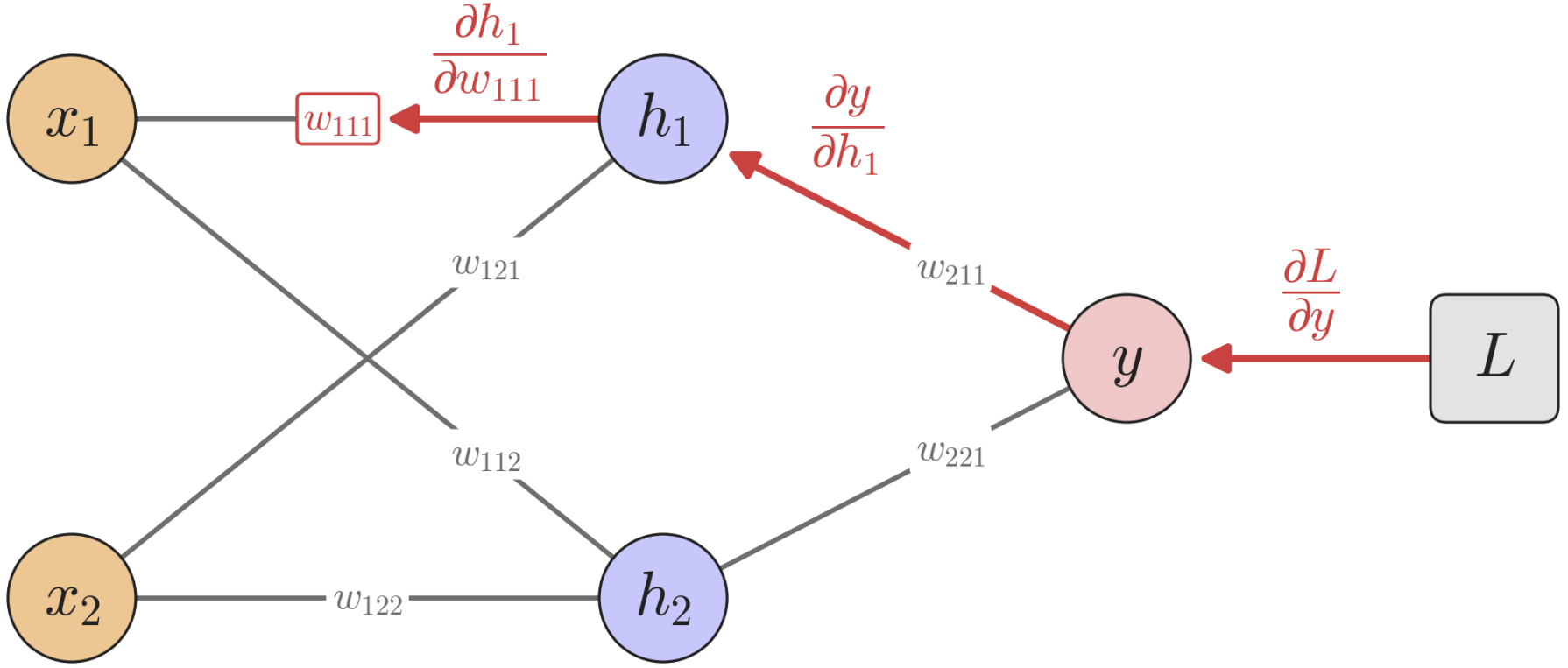
Stochastic gradient descent

- 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 (composite functions)
- **Backpropagation**: smart repeated use of the *chain rule* to compute derivatives

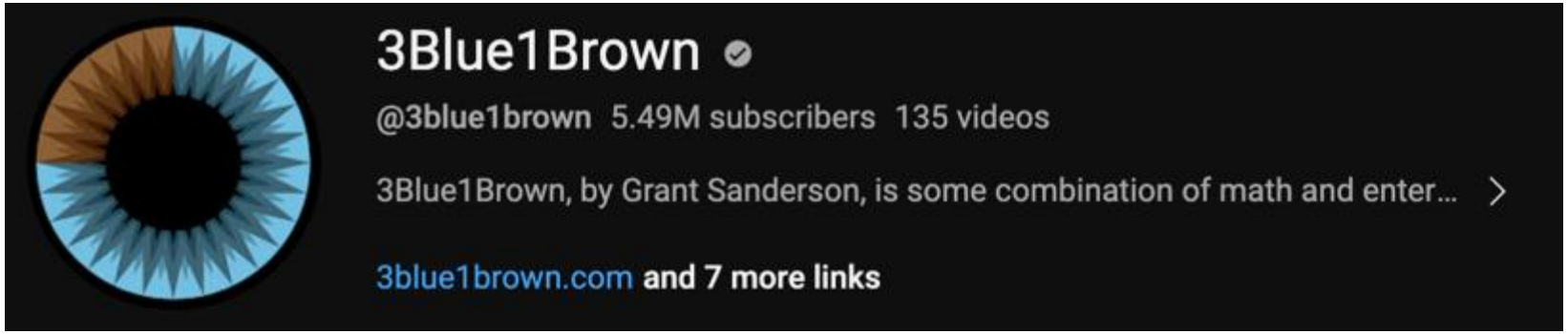
Chain rule used for backpropagation

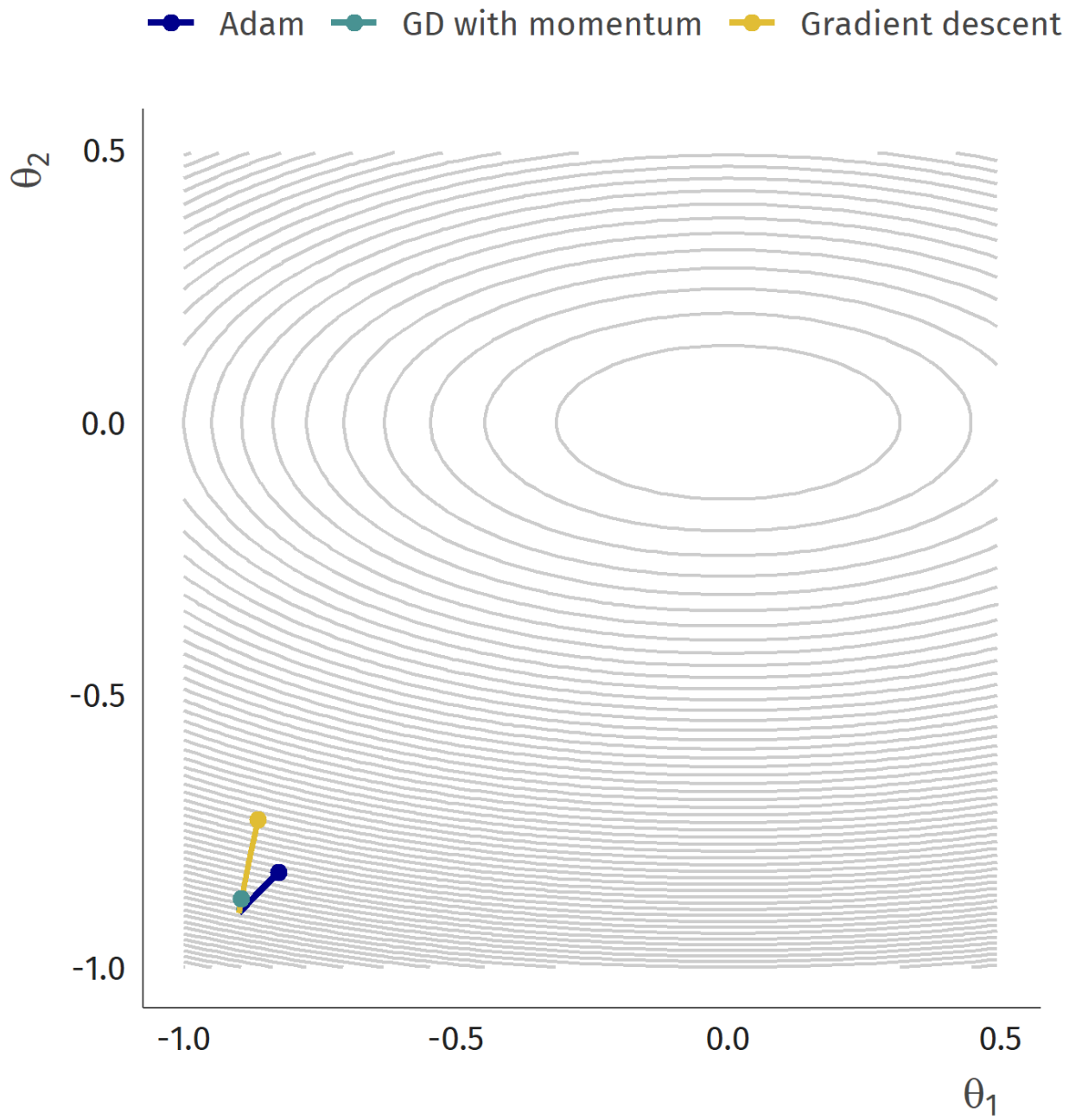


$$\frac{\partial L}{\partial w_{111}} = \frac{\partial L}{\partial y} \cdot \frac{\partial y}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_{111}}$$

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
  )
```

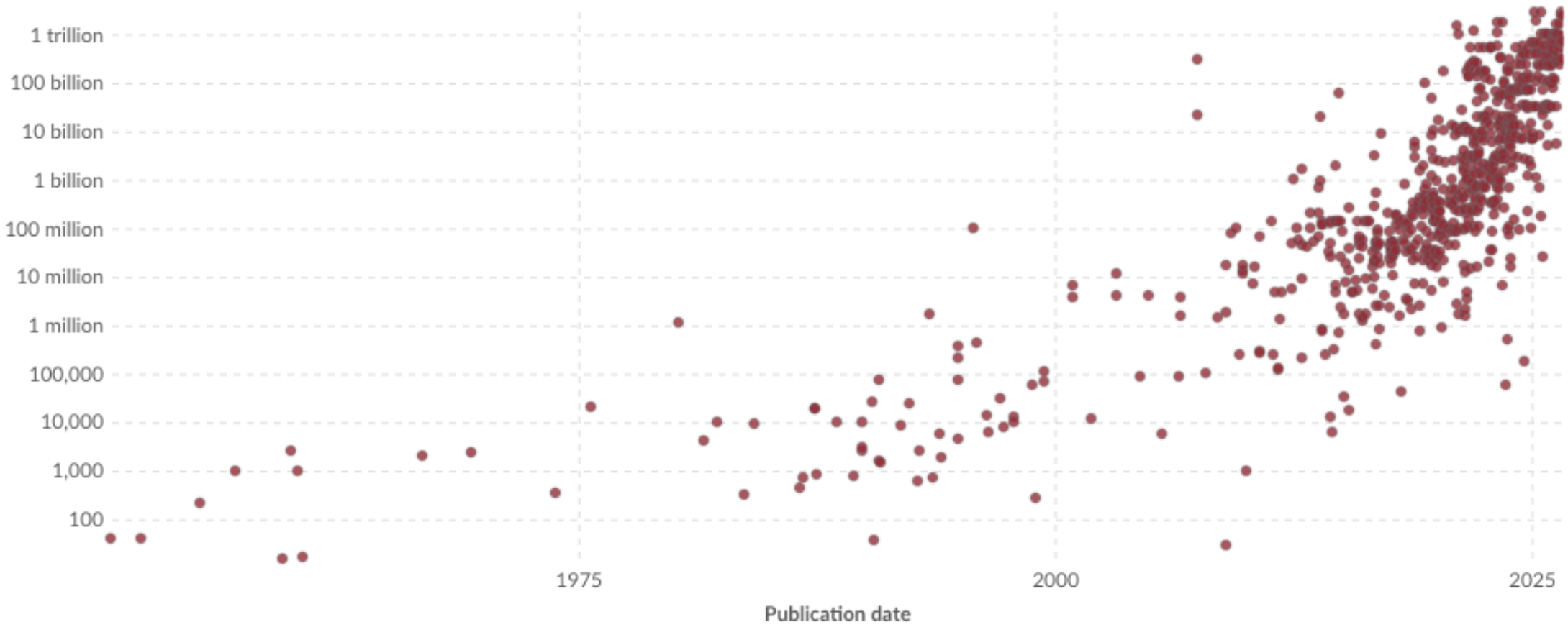
Parameters in notable artificial intelligence systems

Our World in Data

Parameters are variables in an AI system whose values are adjusted during training to establish how input data gets transformed into the desired output; for example, the connection weights in an artificial neural network.

Table | Scatter

Settings



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?
- We need **regularization.**
- Regularization is anything which introduces bias in the parameters to improve generalization (Goodfellow et al., 2016)

Regularization in NNs

- **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!)

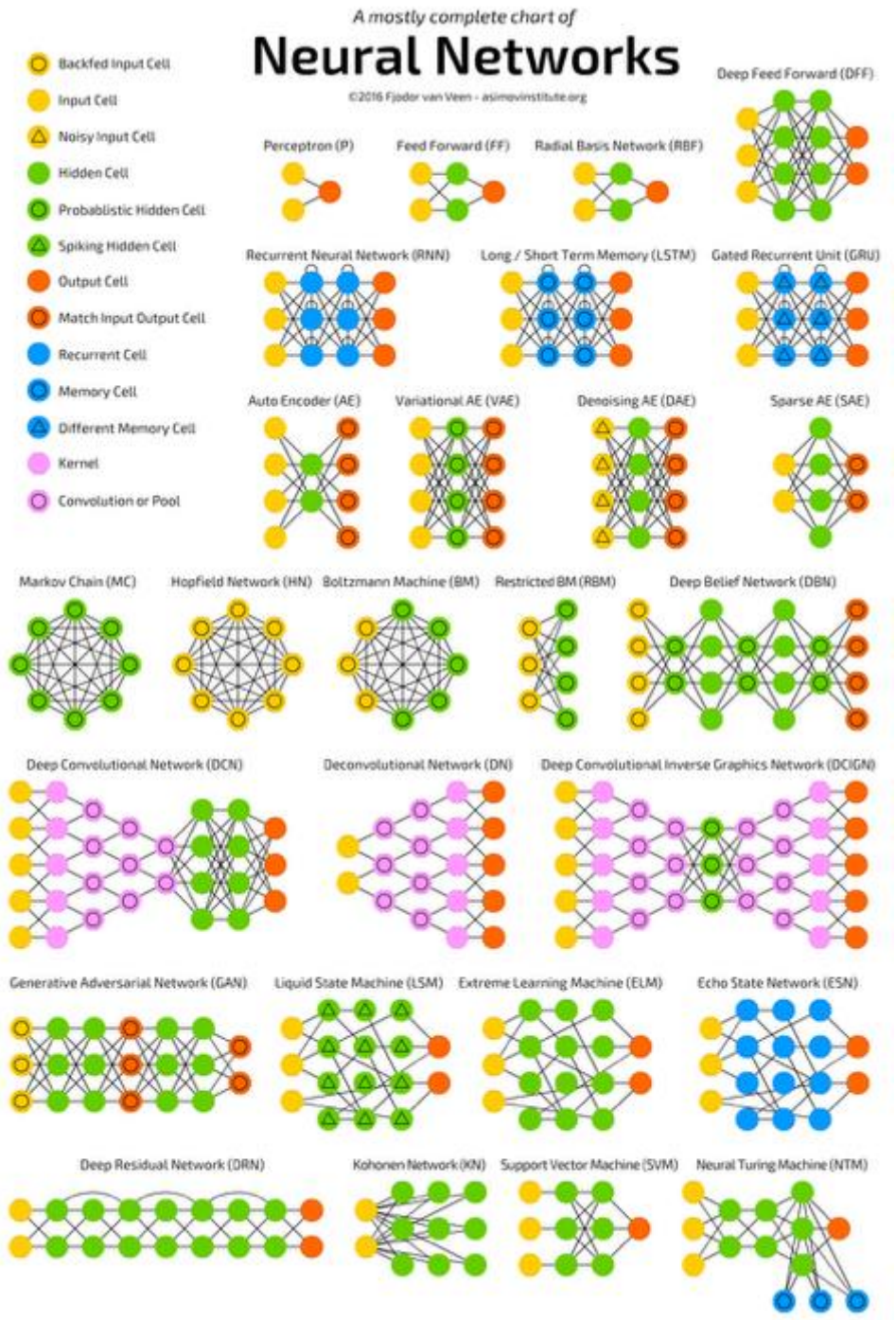
Summary of training

- We need a loss function (e.g., squared error)
- We need gradients (how to change θ to reduce $L(\theta)$)
- We need regularization
- 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
- …

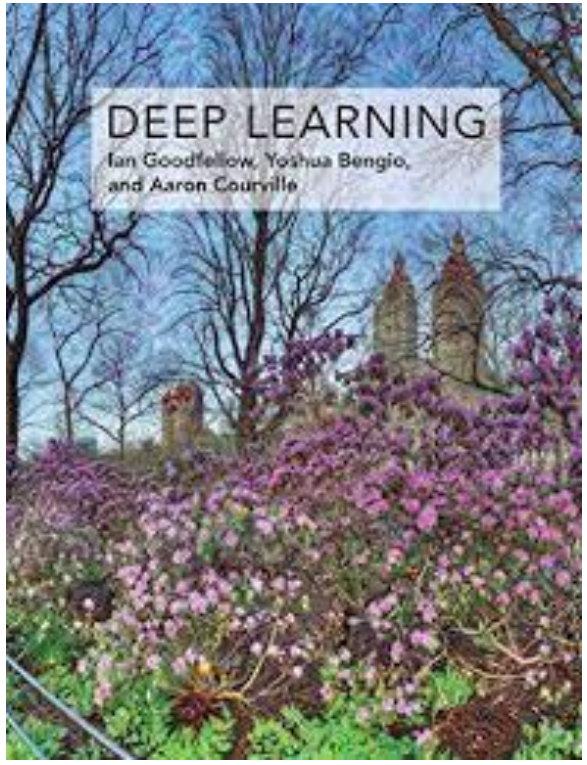


This was just the start

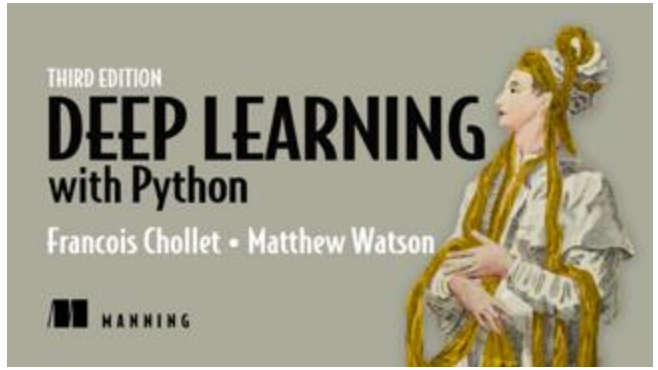
- Convolutional neural networks for images
- Recurrent neural networks: for sequences
- Transformers (basis of all current LLMs)
- Autoencoders (nonlinear dimension reduction)
- Graph neural networks (GNNs; for “graph data”)
- Diffusion models
- …

Further reading

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



Goodfellow et al.





Chollet (R/Python version)

Conclusion

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