Applied Deep Learning

Dr. Philippe Blaettchen
Bayes Business School (formerly Cass)

www.bayes.city.ac.uk

Learning objectives of today

Goals: Understand the difficulties in using neural networks in practice, and how we can implement the more advanced concepts that overcome these difficulties

- Bias and variance, as well as regularization tools
- The problem of vanishing and exploding gradients, as well as slow learning
- Hyperparameter tuning

How will we do this?

- We briefly go through the individual issues that may arise
- We then see how each of the solutions presented can be implemented in TensorFlow
- The notebook can serve as a lookup for typical operations you might want to use when training a neural network

Softmax activation

Multi-class classification

- We again make a linear transformation with a matrix of weights: $\boldsymbol{z}^{[L](i)} = \boldsymbol{a}^{[L-1](i)}\boldsymbol{W}^{[L]} + \boldsymbol{b}^{[L]}$

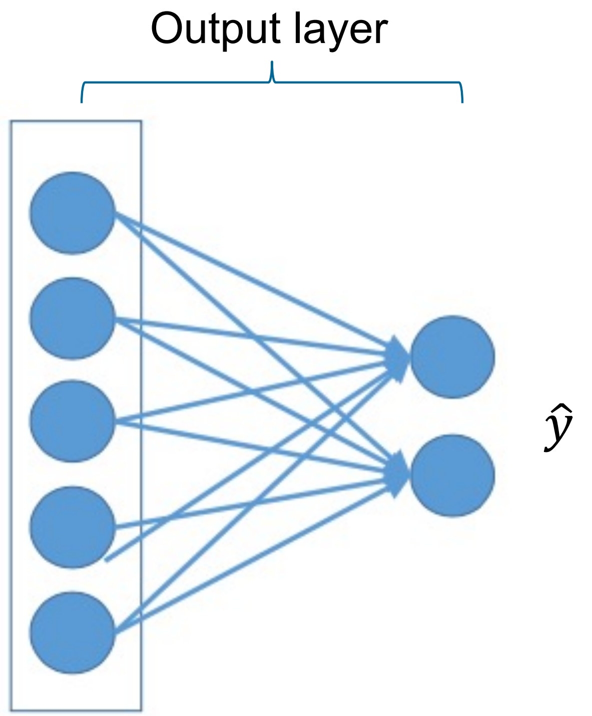
- Note that $\boldsymbol{z}^{[L](i)} = \begin{pmatrix} z_1^{[L](i)} & z_2^{[L](i)} & \cdots & z_K^{[L](i)} \end{pmatrix}$

- We then use the softmax function on each of the outputs:

$$\hat{y}_k^{(i)} = f\left(\boldsymbol{z}^{[L](i)}\right) = \frac{e^{-z_k^{[L](i)}}}{\sum_{k=1}^{K} e^{-z_k^{[L](i)}}}$$

- This implies that $\hat{y}_k^{(i)} \in (0,1)$ and $\sum_{k=1}^{K} \hat{y}_k^{(i)} = 1$

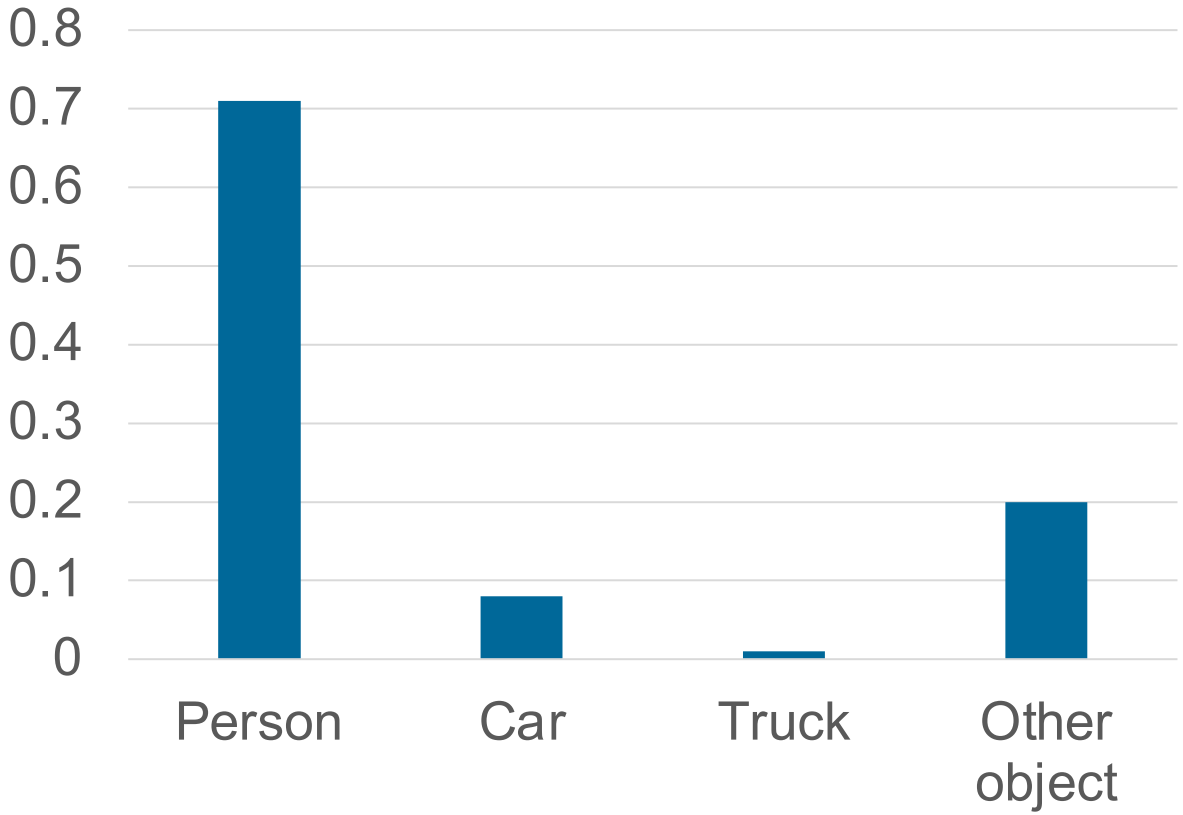
- Hence, we can interpret $\hat{y}_k^{(i)}$ as the probability that $y^{(i)} = k$ ("belongs to class k")



Source: Liang

Softmax output

- E.g., when performing object recognition, we might represent our prediction $\hat{\boldsymbol{y}}^{(i)}$ as

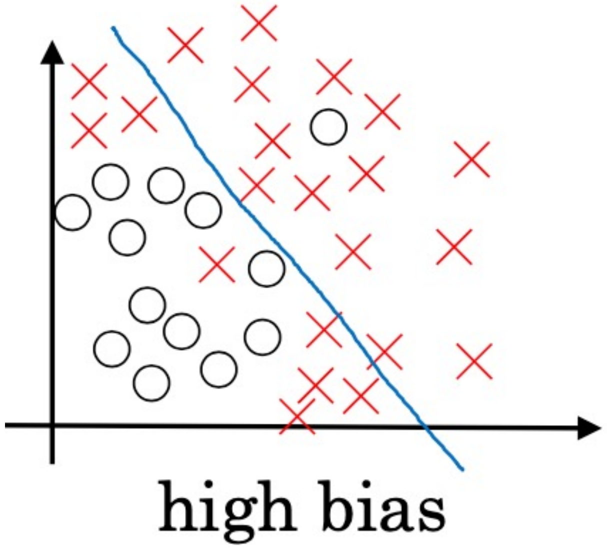


Let’s take a look in Python

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Bias-variance trade-off

Bias and variance



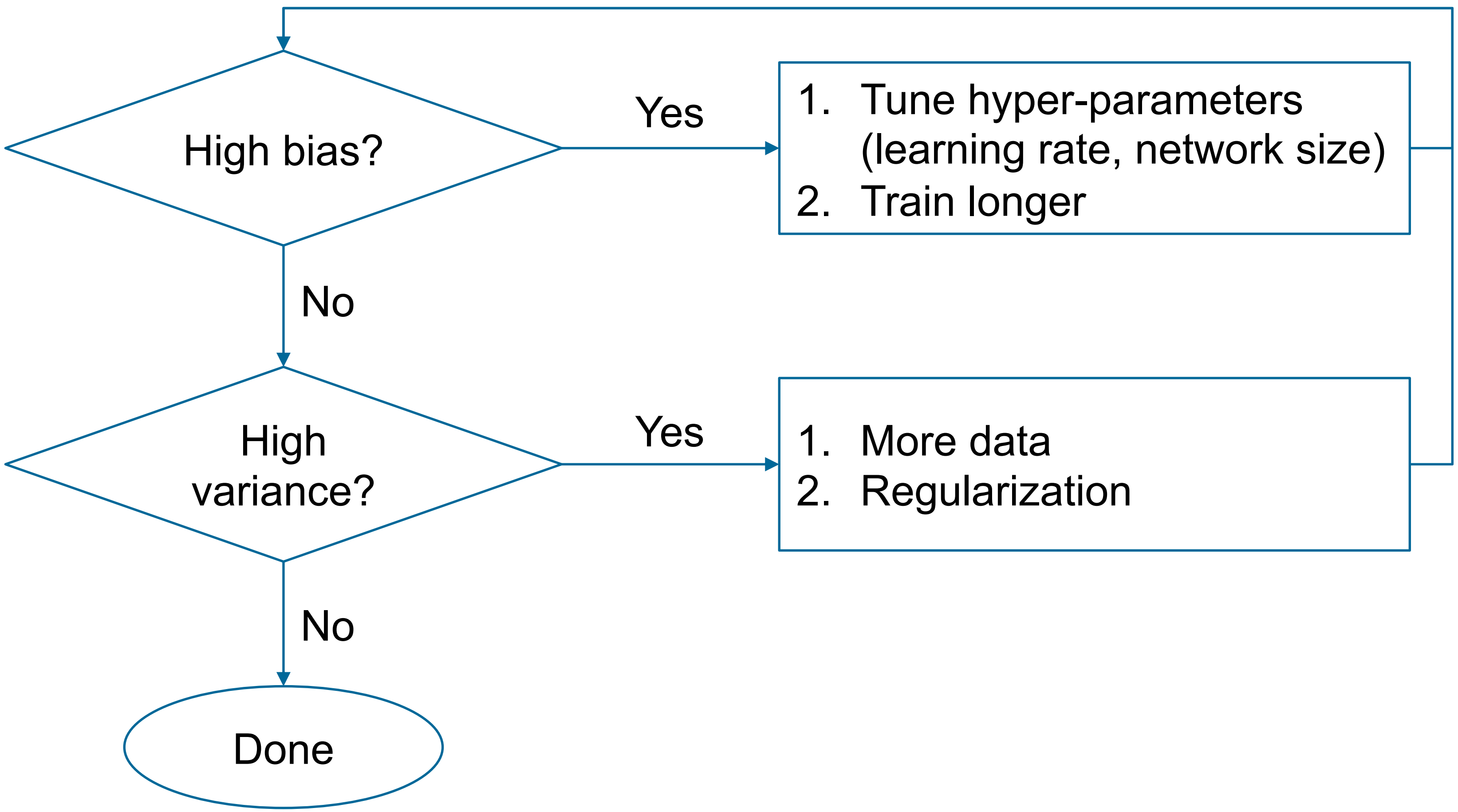
Source: DeeplearningAI

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Recognizing bias and variance

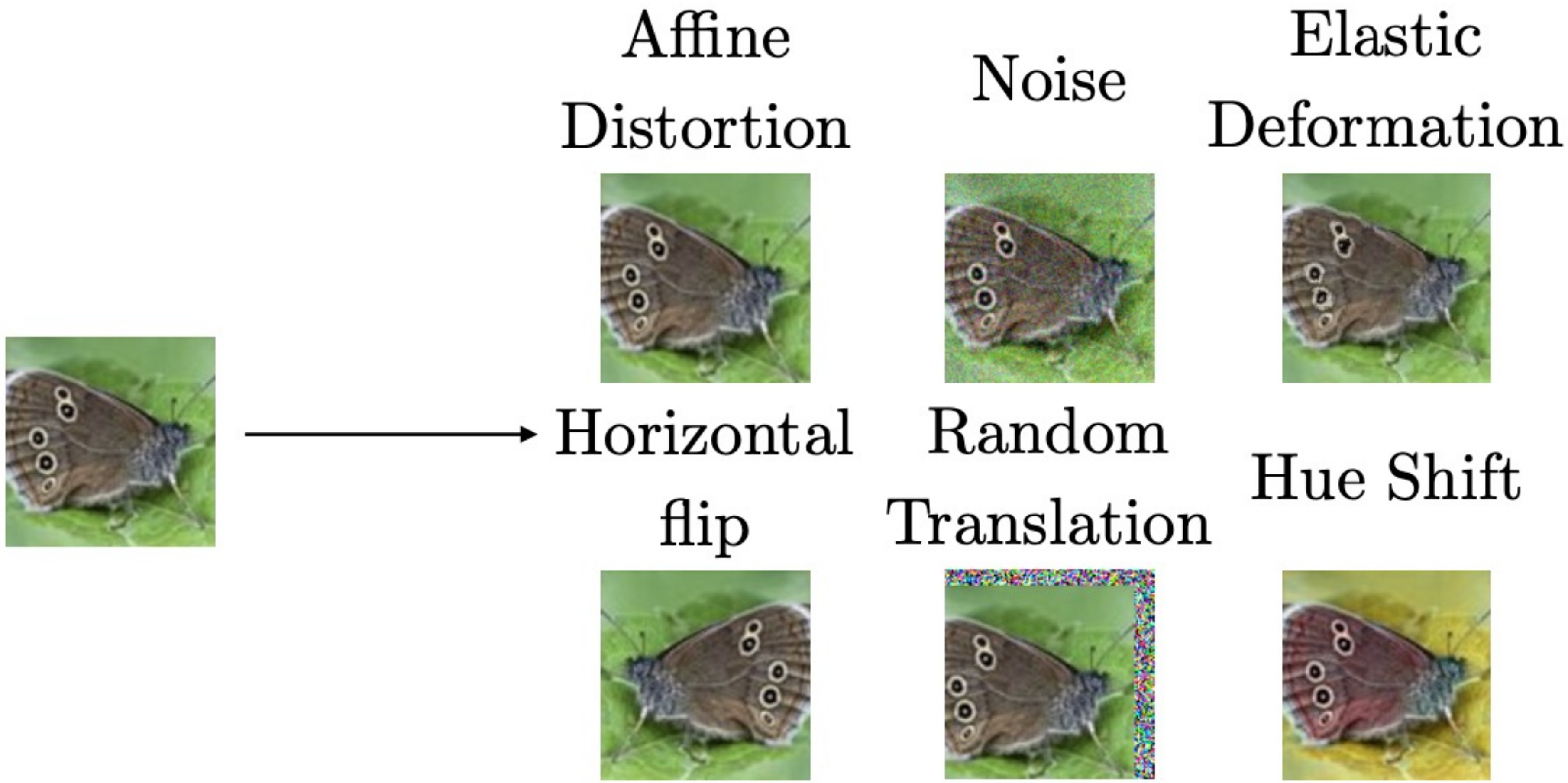
"Bayes error"		1%	1%
	(Avoidable) bias		
Error on training set		2%	10%
	Variance		
Error on validation set		10%	12%
		High variance	High bias

How to deal with bias and variance



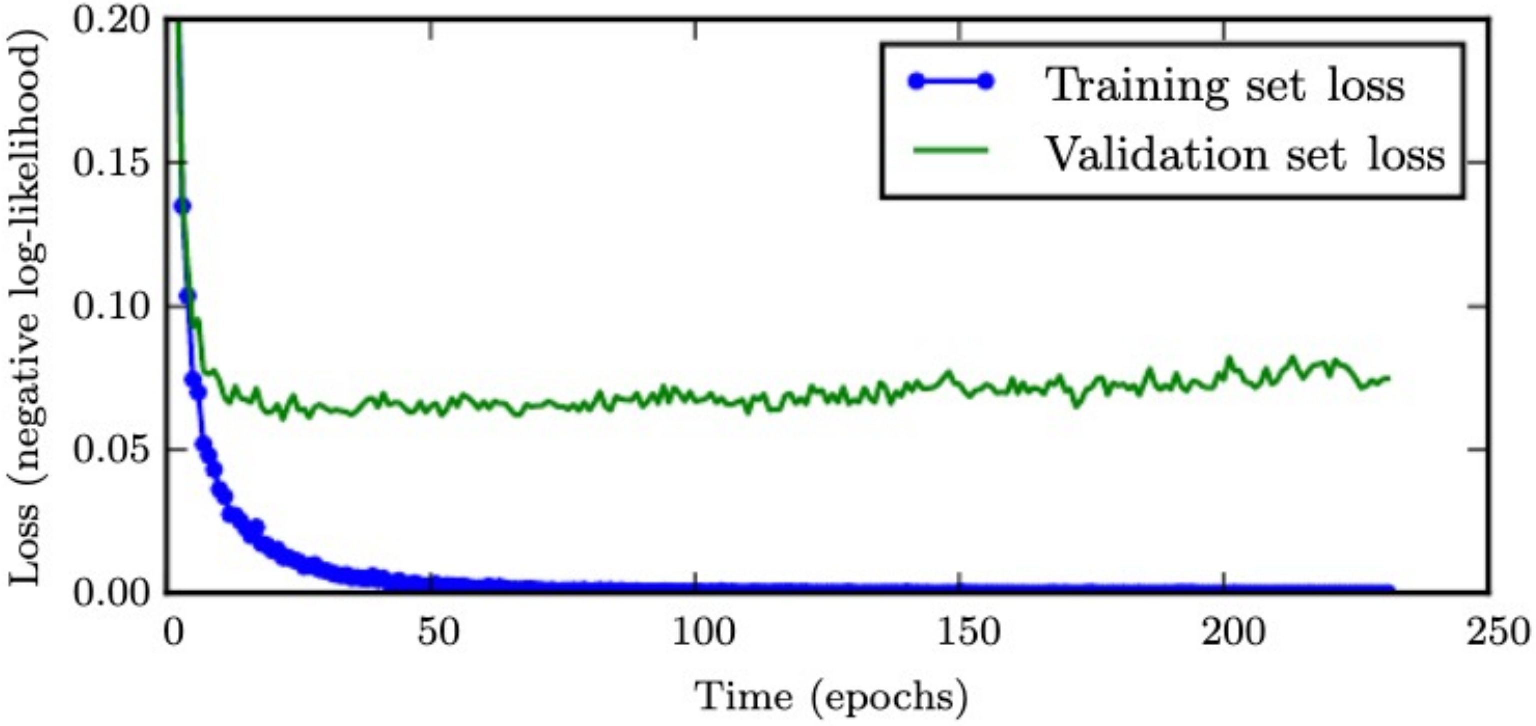
Source: DeeplearningAI

Dataset augmentation



Source: Goodfellow

Early stopping



Source: Goodfellow

Try it out in Python

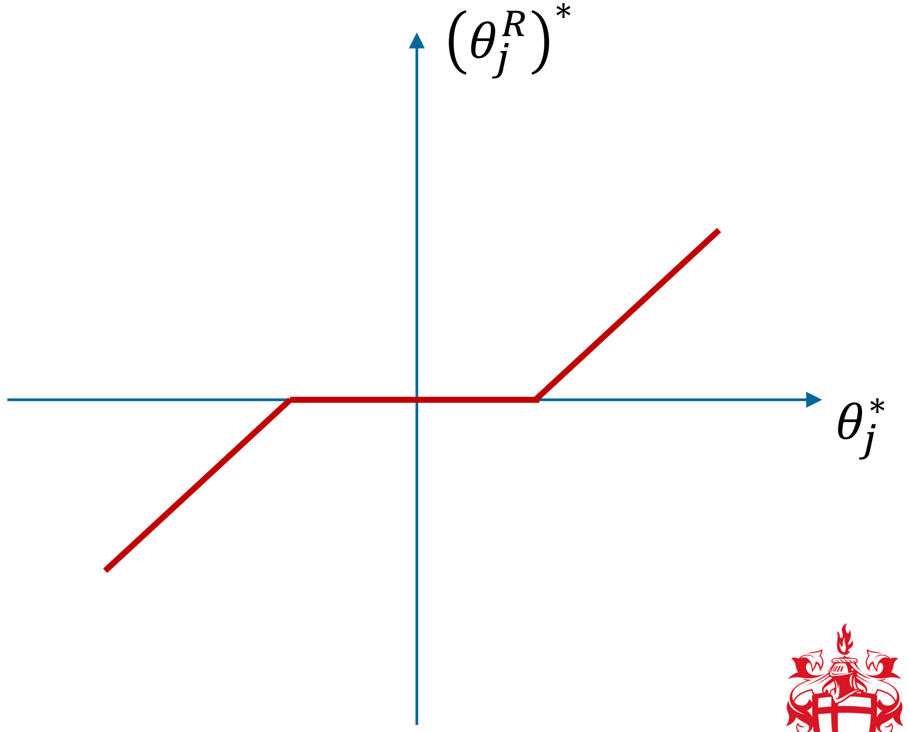
L1-regularization ("Lasso regression")

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L^{(i)} \quad \Longleftrightarrow \quad J_R(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L^{(i)} + \lambda \|\boldsymbol{\theta}\|_1 = \frac{1}{n}\sum_{i=1}^{n} L^{(i)} + \lambda \sum_{j=1}^{m} |\theta_j|$$

- Gradient: $\nabla_\theta J_R(\boldsymbol{\theta}) = \nabla_\theta J(\boldsymbol{\theta}) + \lambda\, sign(\boldsymbol{\theta})$
- Gradient descent update:

$$\begin{aligned}\boldsymbol{\theta} &:= \boldsymbol{\theta} - \alpha \nabla_\theta J_R \\ &= \boldsymbol{\theta} - \alpha \nabla_\theta J(\boldsymbol{\theta}) - \alpha\lambda\, sign(\boldsymbol{\theta})\end{aligned}$$

→ "sparsity"



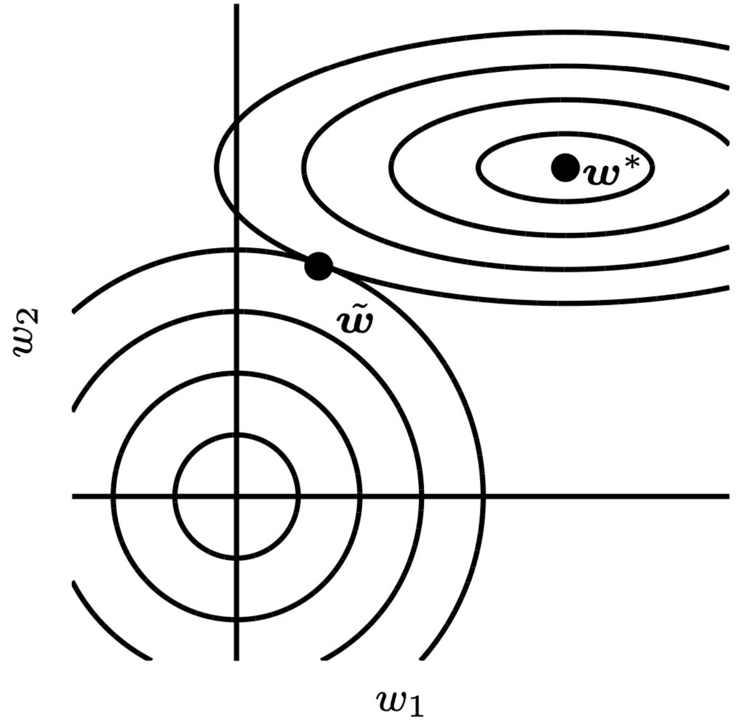
L2-regularization ("Ridge regression")

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L^{(i)} \quad \Longleftrightarrow \quad J_R(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L^{(i)} + \frac{\lambda}{2}\|\boldsymbol{\theta}\|_2 = \frac{1}{n}\sum_{i=1}^{n} L^{(i)} + \frac{\lambda}{2}\sqrt{\theta_1^2 + \theta_2^2 + \cdots + \theta_m^2}$$

- Gradient: $\nabla_\theta J_R(\boldsymbol{\theta}) = \nabla_\theta J(\boldsymbol{\theta}) + \lambda\boldsymbol{\theta}$
- Gradient descent update:

$$\begin{aligned} \boldsymbol{\theta} &:= \boldsymbol{\theta} - \alpha\nabla_\theta J_R \\ &= \boldsymbol{\theta} - \alpha\nabla_\theta J(\boldsymbol{\theta}) - \alpha\lambda\boldsymbol{\theta} \\ &= (1-\alpha\lambda)\boldsymbol{\theta} - \alpha\nabla_\theta J(\boldsymbol{\theta}) \end{aligned}$$

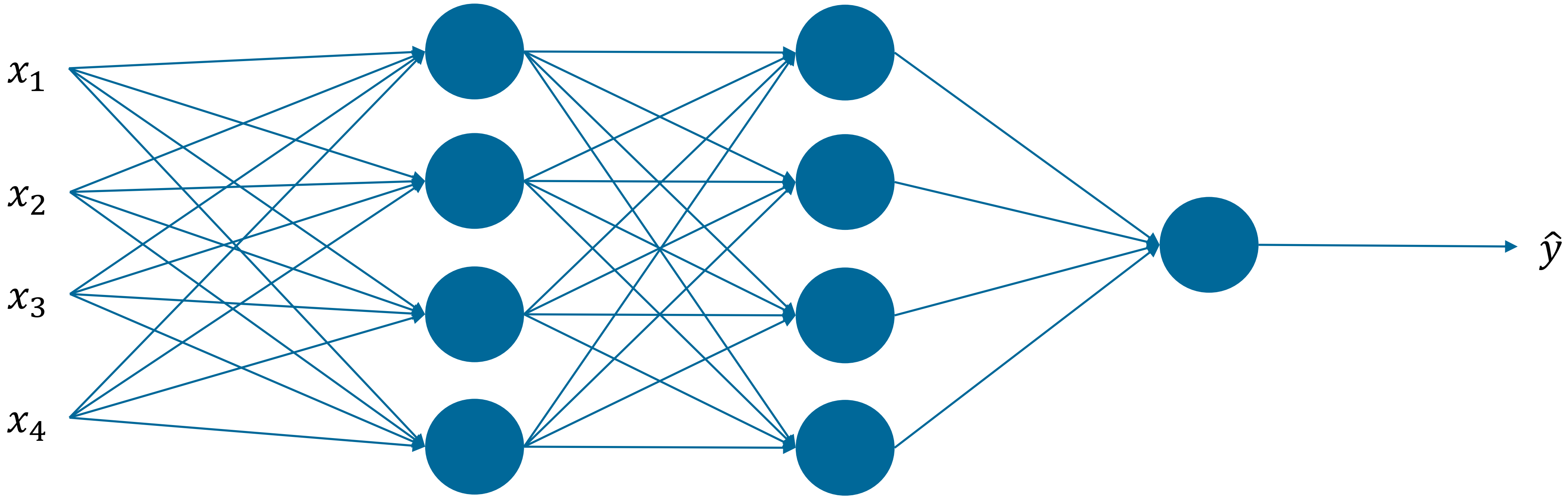
→ "weight decay"



Source: Goodfellow

Try it out in Python

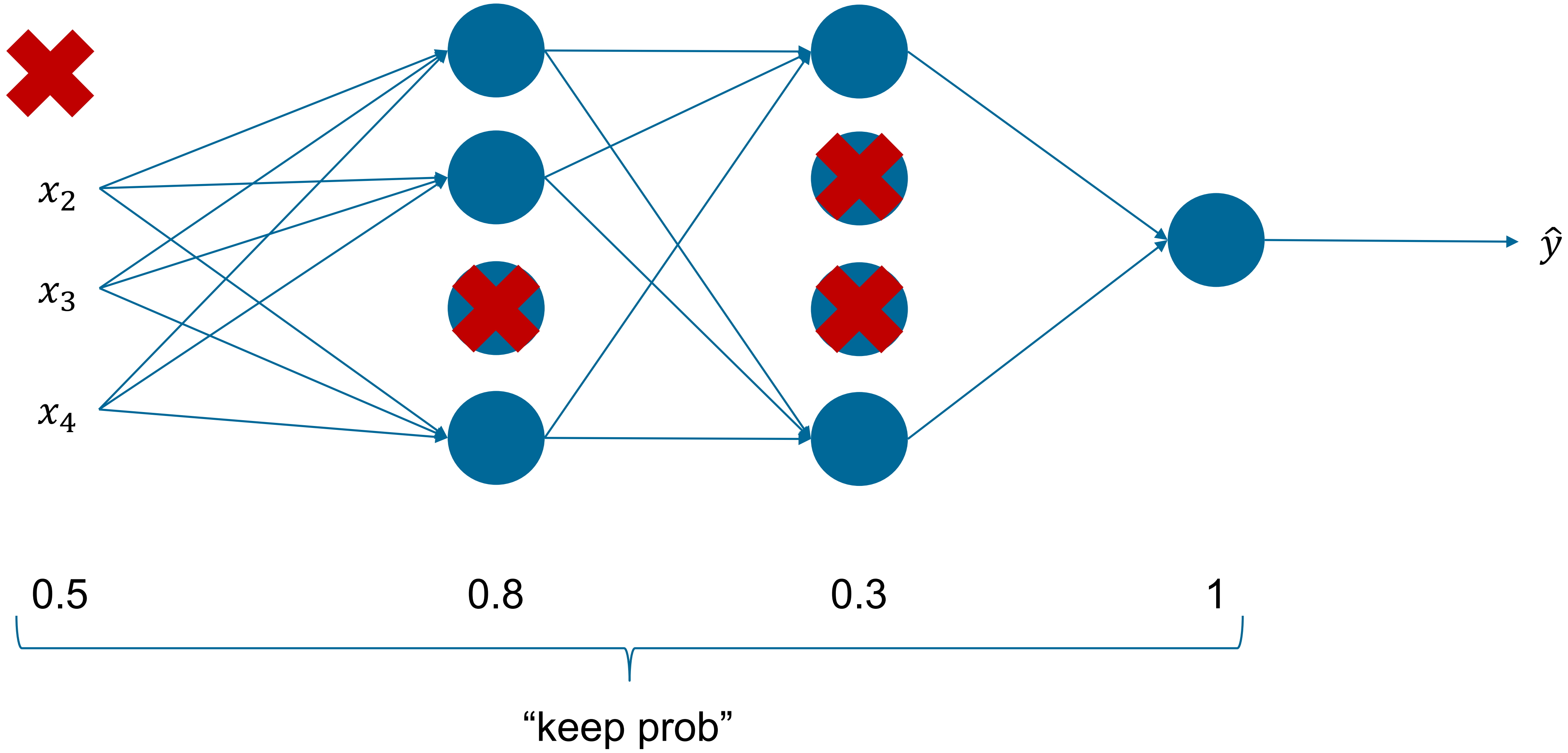
Dropout regularization



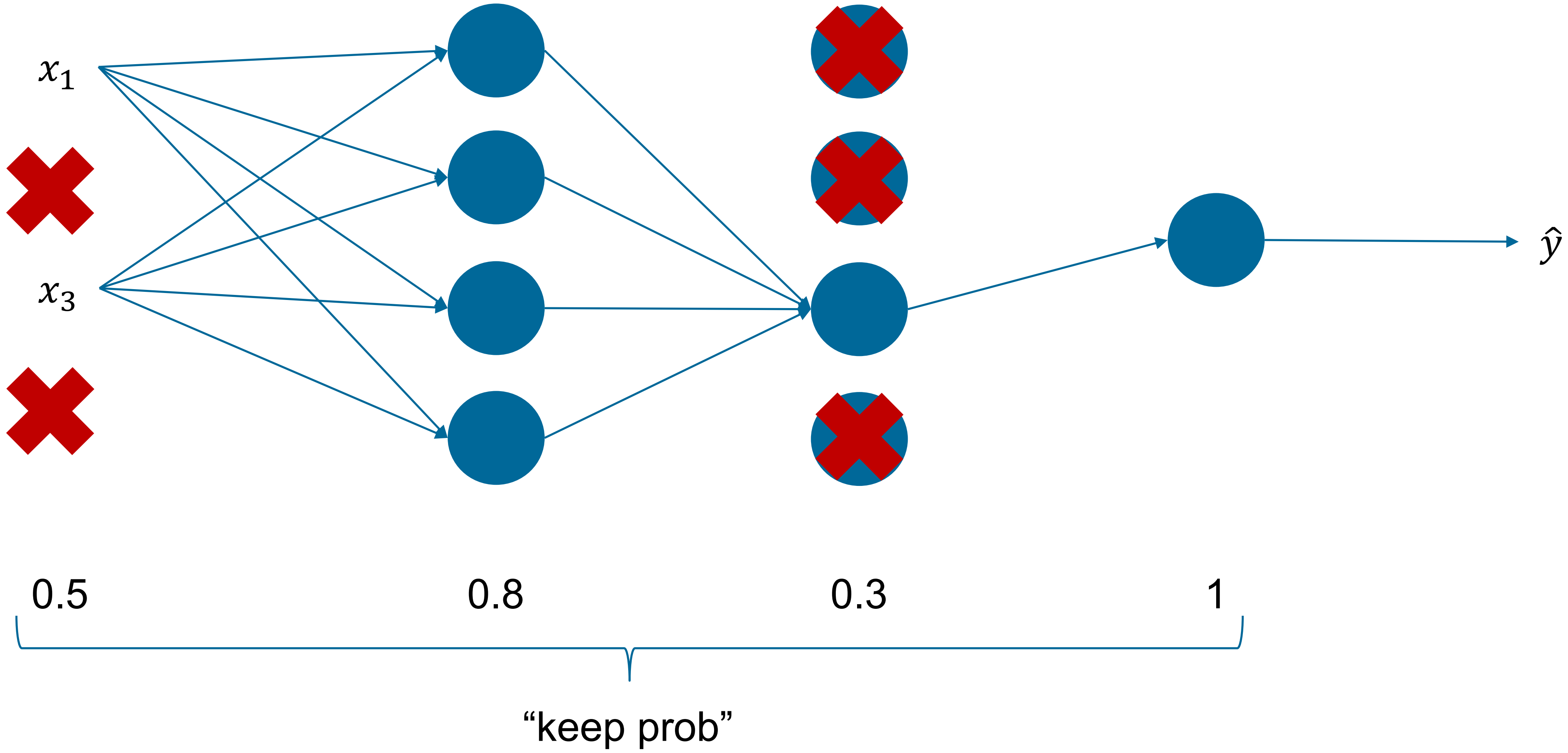
0.5 0.8 0.3 1

$\underbrace{\qquad\qquad\qquad\qquad}_{\text{"keep prob"}}$

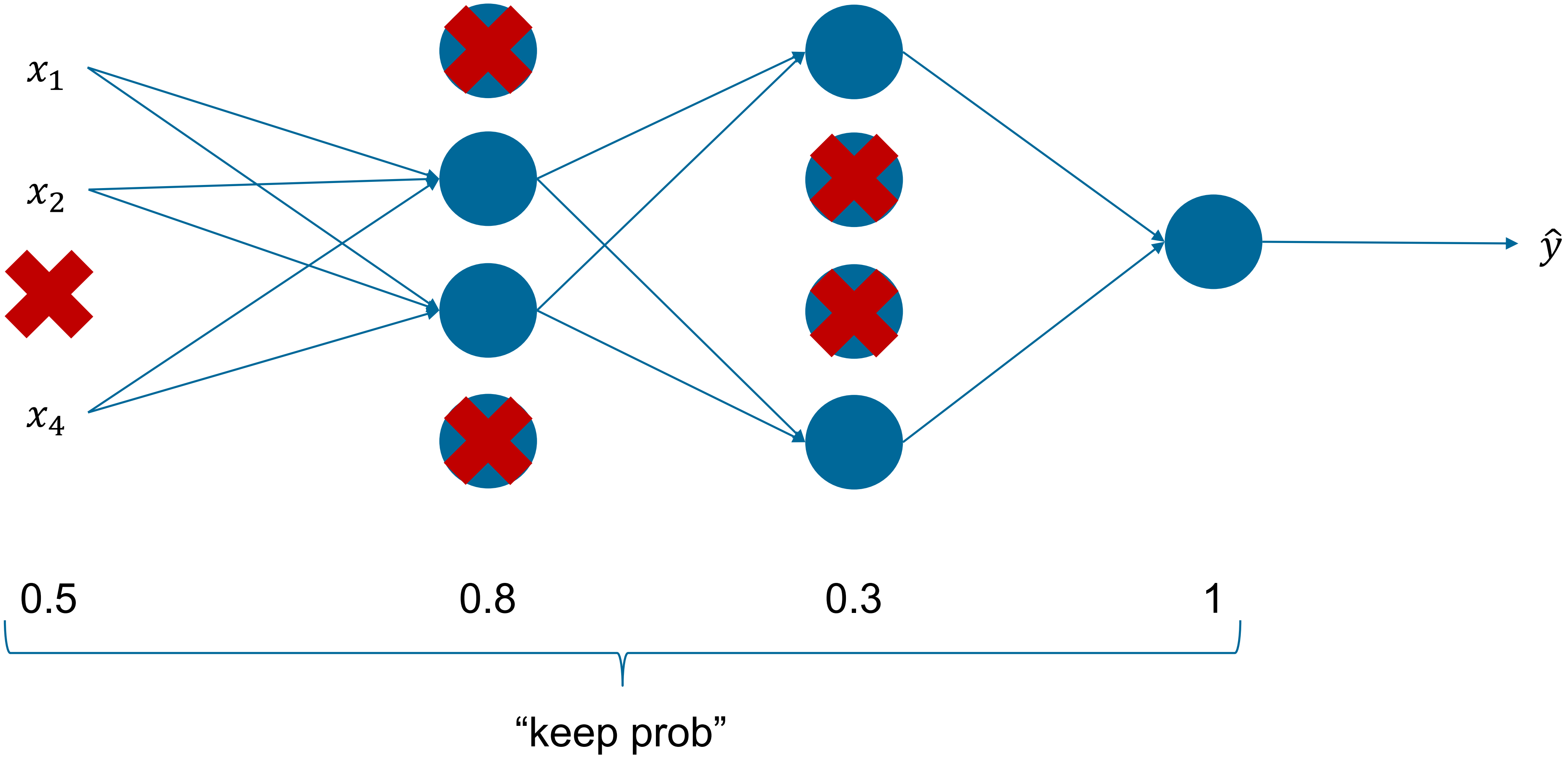
Dropout regularization



Dropout regularization



Dropout regularization



Try it out in Python

Improving learning

Other possible issues we might come across in training

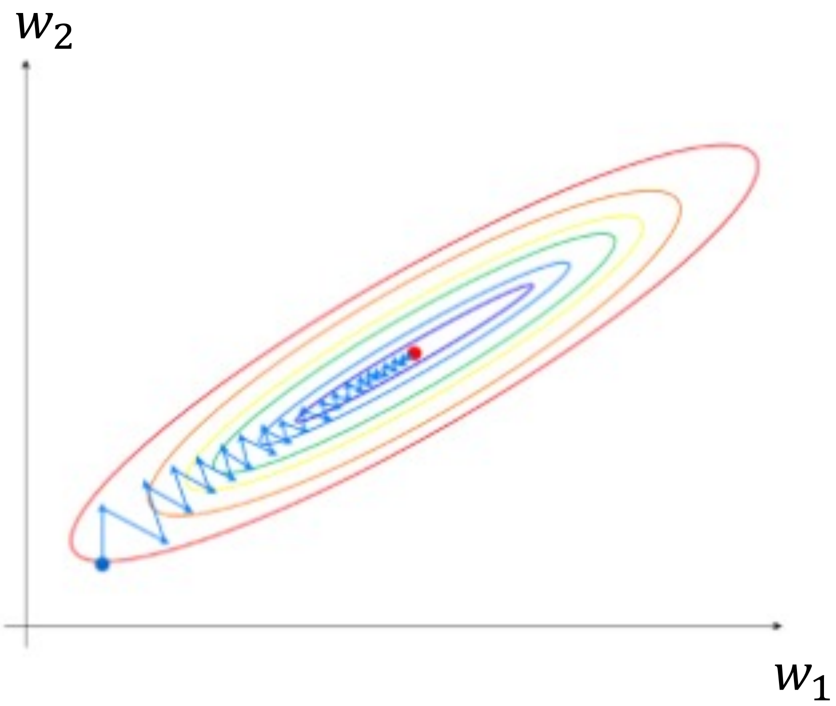
- Vanishing and exploding gradients
  → Data normalization
  → Batch normalization
  → Initialization
  → Activation functions

- Very slow training
  → Mini-batch gradient descent
  → Momentum optimization
  → Learning-rate scheduling
  → Reusing pretrained layers

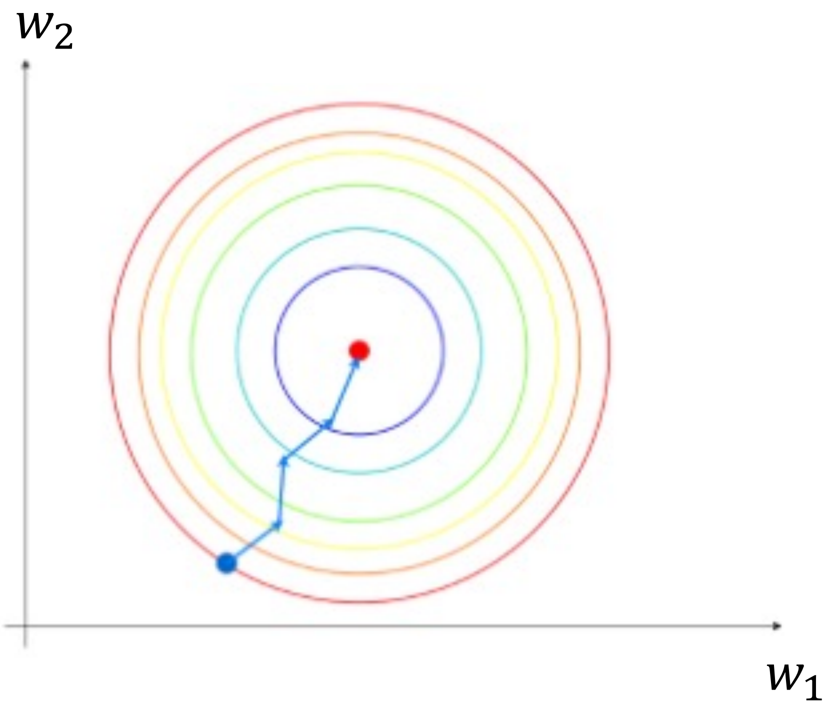
Improving learning – vanishing and exploding gradients

Finally: normalizing the inputs

E.g., $x_1 \in (0,100)$
$x_2 \in (0,1)$



E.g., $x_1 \in (0,1)$
$x_2 \in (0,1)$



Source: Erdem

Normalizing the inputs also for deeper layers: batch normalization

- Add an operation before/after activation function. For each input:
    - Standardize it (zero-centering, and division by standard deviation)
    - Then, scale it by a parameter γ and add a shift β (we **learn** these parameters)

- When we use the neural network to compute predictions, we don't necessarily have means and standard deviations, however (or the new observations might not be independent, …)

  → also keep track of a running mean μ and variance σ (we keep track of a moving average, but we are not learning, so these are **non-trainable** parameters)

- Overall, for each layer that is batch-normalized, we have $4 \times neurons$ additional parameters

- Batch normalization tends to make the network less sensitive to the initialization, and also helps to regularize it, but adds to the runtime

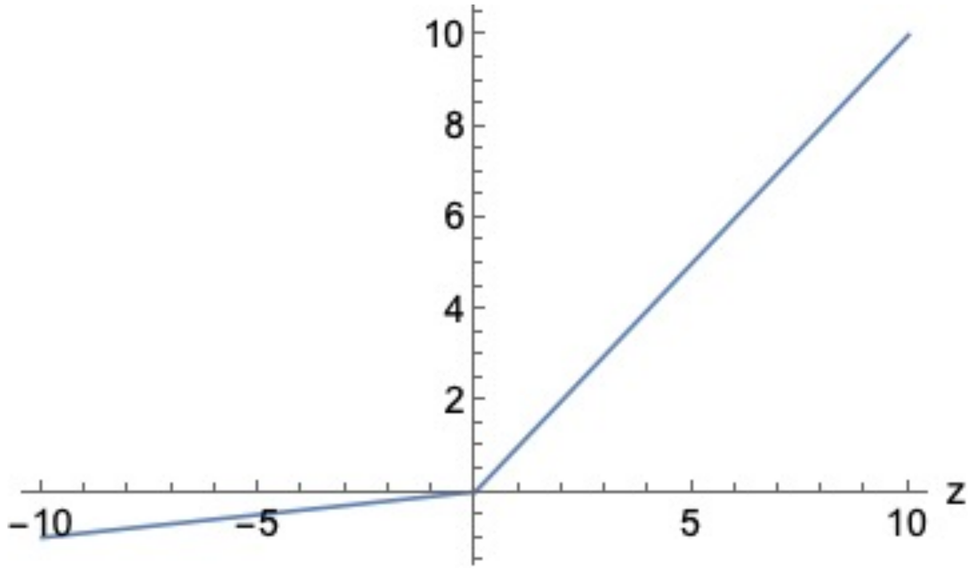
BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Try it out in Python

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

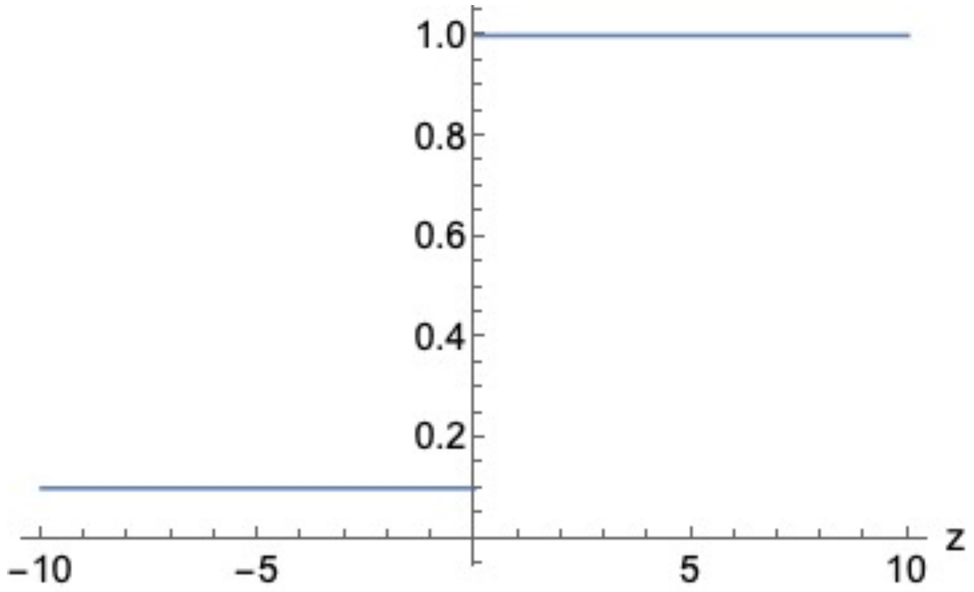
A possible counter: non-saturating activation functions

Leaky ReLU



$$f(z) = \max\{0.1z, z\}$$

"Derivative"



$$f'(z) = \begin{cases} 0.1, if\ z < 0 \\ 1, if\ z > 0 \end{cases}$$

Consider also: ELU

Another possible counter: the "right" initialization

- Glorot-initialization (assuming logistic sigmoid, tanh, softmax activation):
  - variance of inputs ≈ variance of outputs
  - variance of gradients before layer ≈ variance of gradients after layer

  - Idea: given a layer with in inputs and out neurons, distribute weights either
    - normally, with mean 0 and variance $\frac{1}{\frac{in+out}{2}}$, or (kernel_initializer="glorot_normal")
    - uniformly between $\left[-\sqrt{\frac{3}{\frac{in+out}{2}}}, \sqrt{\frac{3}{\frac{in+out}{2}}}\right]$ (default)

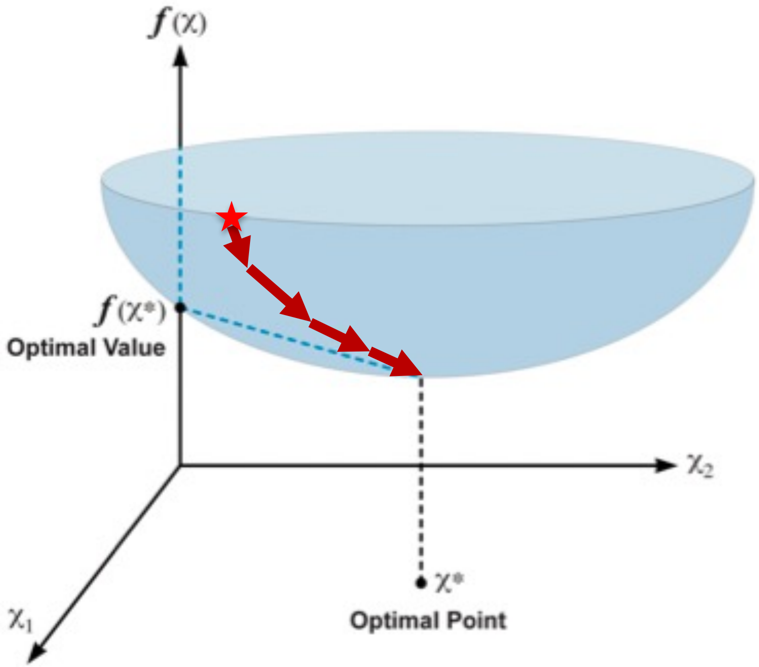
- He-initialization (assuming ReLU and its variants):
  - Idea: given a layer with in inputs and out neurons, distribute weights either
    - normally, with mean 0 and variance $\frac{2}{in}$, or (kernel_initializer="he_normal")
    - uniformly between $\left[-\sqrt{\frac{6}{in}}, \sqrt{\frac{6}{in}}\right]$ (kernel_initializer="he_uniform")

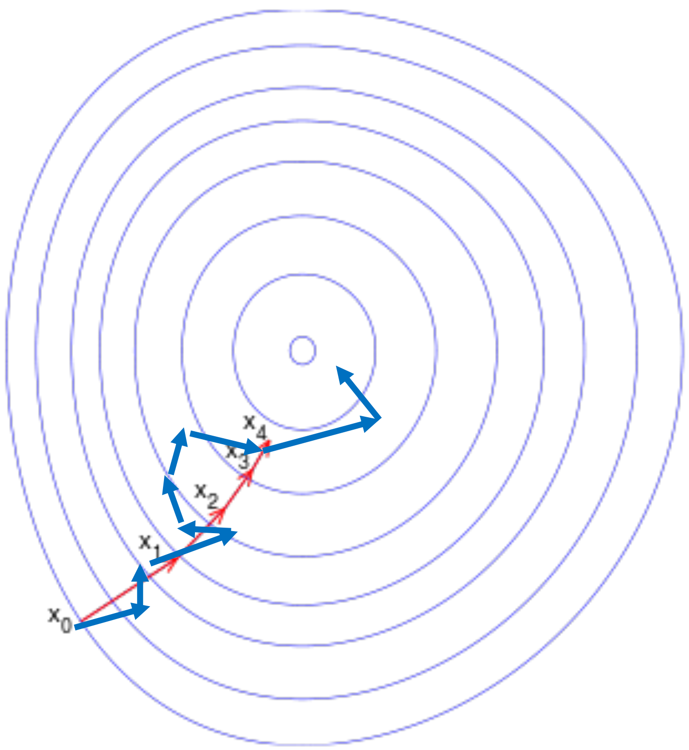
BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Try it out in Python

Improving learning – slow training

Gradient descent versus stochastic gradient descent





1. Decide a "learning rate" α
2. Start with some parameters $\boldsymbol{\theta}$
3. For a certain number of iterations
   - Compute $J(\boldsymbol{\theta})$
   - Compute $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} \left(\frac{1}{n} \sum_{i=1}^{n} L^{(i)} \right)$
   - Let $\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

1. Decide a "learning rate" α
2. Start with some parameters $\boldsymbol{\theta}$
3. For a certain number of iterations
   - Compute $J(\boldsymbol{\theta})$
   - Compute $\nabla_{\boldsymbol{\theta}} L^{(i)}$ for a random (i)
   - Let $\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L^{(i)}$

Mini-batch gradient descent

- Core idea: don’t take the derivative over all observations, but over a bit more than just one

- Trading off between normal gradient descent (“batch gradient descent”) and stochastic gradient descent
  - Batch gradient descent: too slow per iteration
  - Stochastic gradient descent: loses benefits of vectorization

- For small training sets: batch gradient descent
  - Otherwise, use typical batch sizes such as 64, 128, 256, 512, 1024 …

Try it out in Python

Gradient descent with momentum

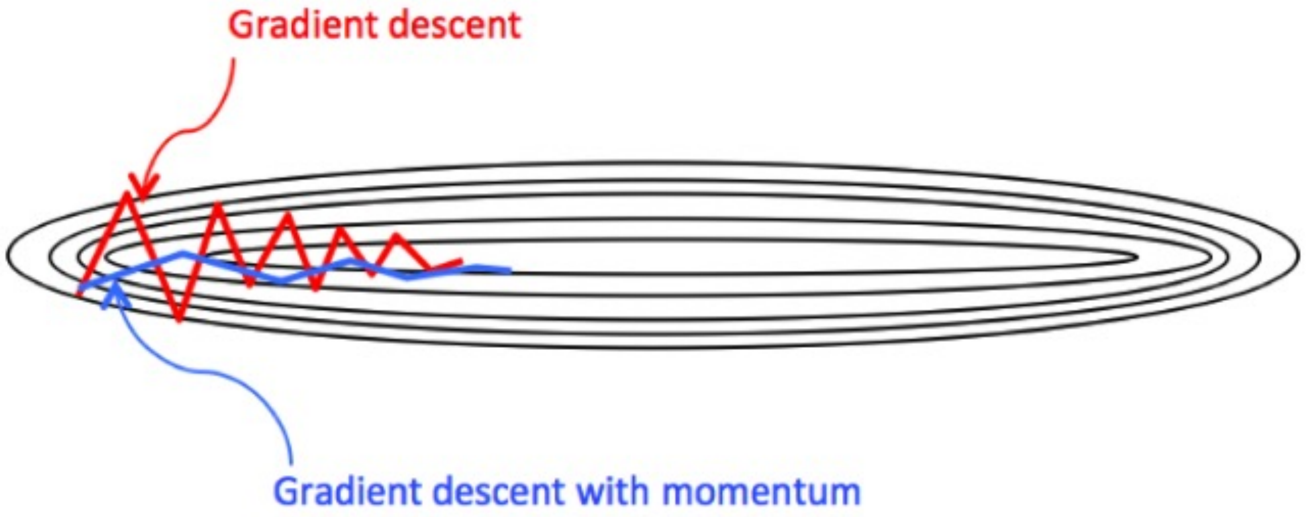
- In gradient descent, we take small, regular steps down a slope
- But if you think of a ball rolling down a slope, it will start slowly, but build up speed (and, thus, momentum) → the "steps" depend not just on the current slope, but on the slope so far!

Gradient descent:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

Momentum optimization:

$$\boldsymbol{m} := \beta \boldsymbol{m} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$
$$\boldsymbol{\theta} := \boldsymbol{\theta} + \boldsymbol{m}$$



Source: Trehan

RMSprop ("Root mean square prop")

- We normalize the gradient (using the moving average of the square of the gradients)
  - If there is a direction with a lot of oscillation, we penalize the update in this direction
  - If there is a direction with little oscillation, we help along the update in this direction

$$s \coloneqq \beta s + (1-\beta)\left(\frac{\partial J}{\partial \theta}\right)^2$$

$$\theta \coloneqq \theta - \alpha \frac{\frac{\partial J}{\partial \theta}}{\sqrt{s+\varepsilon}}$$

- Used to be the standard algorithm to use until Adam

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Adam ("Adaptive moment estimation")

- Combining momentum and RMSprop

$$m := \beta_1 m - (1 - \beta_1)\frac{\partial J}{\partial \theta}$$

$$s := \beta_2 s + (1 - \beta_2)\left(\frac{\partial J}{\partial \theta}\right)^2$$

$$\widehat{m} := \frac{m}{1 - \beta_1^t}$$

$$\hat{s} := \frac{s}{1 - \beta_2^t}$$

$$\theta := \theta + \alpha\frac{\widehat{m}}{\sqrt{\hat{s} + \varepsilon}}$$

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Try it out in Python

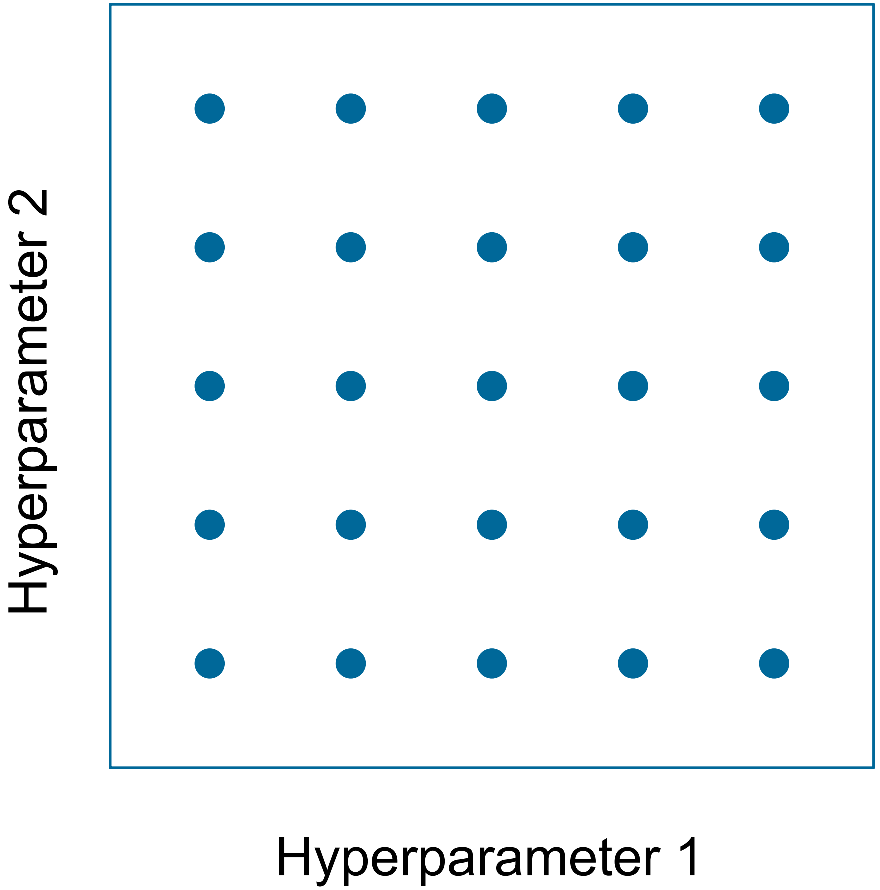
Algorithm overview

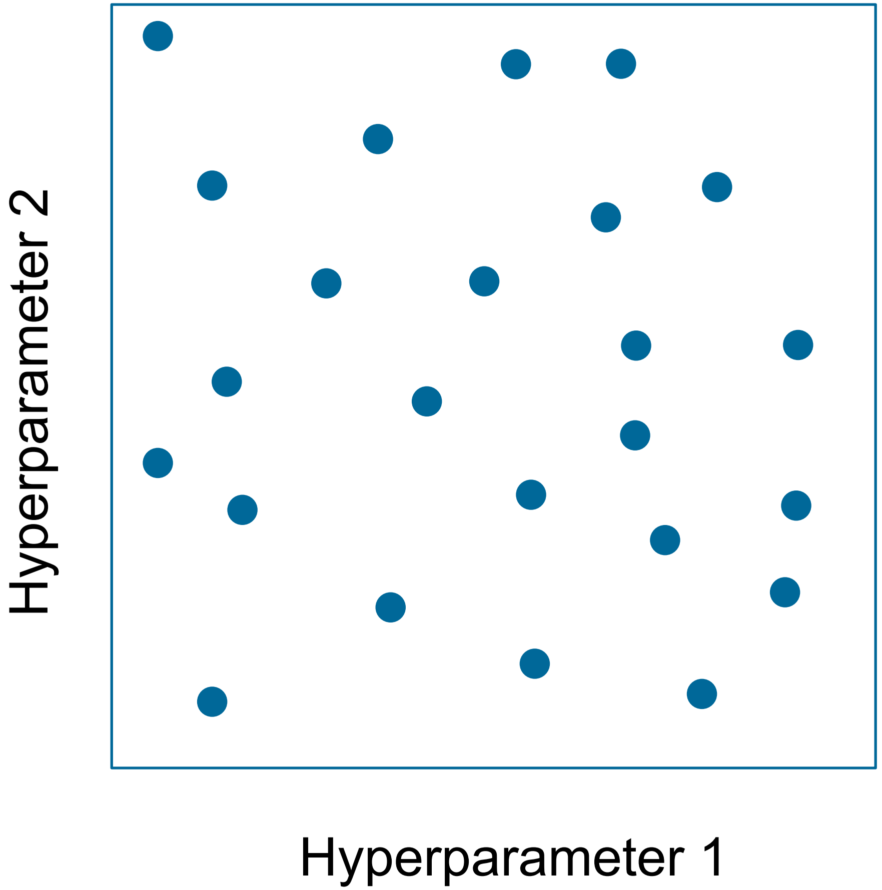
Algorithm	Convergence speed	Convergence quality
SGD	bad	good
SGD with momentum	okay	good
RMSprop	good	medium-good
Adam	good	medium-good

Source: Adapted from Géron

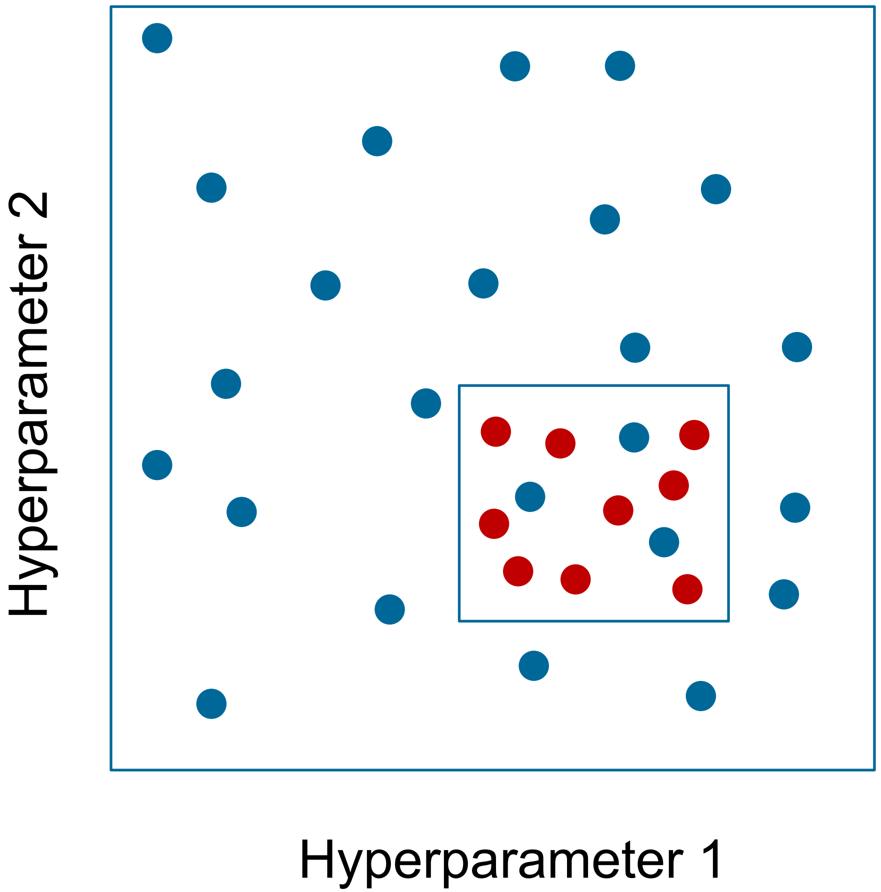
Hyperparameter tuning

Hyperparameter tuning process rule 1: Choose random combinations





Hyperparameter tuning process rule 2: Go from coarse to fine



Hyperparameter tuning process rule 3: Use purpose-built libraries

- Hyperopt
- Keras Tuner
- Scikit-Optimize
- …

Hyperparameter tuning process rule 4: Pick the right scale

- Say you want to set hyperparameter α in the range $0.001, \dots, 1$
- You can try out your model 5 times

- The naïve option: uniform distribution between 0.001 and 1
  → $\alpha = np.random.rand(0.001{,}1)$

- The smarter option: logarithmic spacing
  → $r = -3 \times np.random.rand(0{,}1)$
  → $\alpha = 10^{r}$

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Hyperparameter tuning process rule 5: Prioritize

A typical (but no way always optimal) prioritization:

1. Learning rate
2. Mini-batch size
3. Regularization parameters
4. Number of hidden units (mostly, the same number per layer works just fine – with some exceptions, such as a larger first hidden layer)
5. Number of hidden layers (usually, start with just a few hidden layers, unless you are dealing with complex tasks such as image classification. But then, you usually don't train your own model)
6. Learning rate decay
7. Other algorithm parameters (but the defaults often work fine)

Learning rate scheduling and decay

Source: Géron

Typical learning rate schedules

Power scheduling

- $\alpha_t = \frac{\alpha_0}{1+\frac{t}{s}}$, where t is the epoch and s is the number of epochs until we reach $\frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \ldots$

Exponential scheduling

- $\alpha_t = \alpha_0 0.1^{t/s}$, so now we indicate with s the number of epochs until we reach $0.1, 0.01, 0.001, \ldots$

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Try it out in Python

Autoencoders

What is an autoencoder

- A neural network that predicts its own inputs
  → why can this be useful?

- Since we are not predicting a label (at least in the base-case), what do we call the task?
  → unsupervised learning!

Recall that a neural network learns representations

x

Learn $f(\cdot)$ → $f(x)$ → Learn $g(\cdot)$ → $y \approx g\big(f(x)\big)$

E.g., $y = 1$, if it's a cat
$y = 0$, if it's a BA student

Recall that a neural network learns representations

x

Learn $f(\cdot)$

$f(x)$

Learn $g(\cdot)$

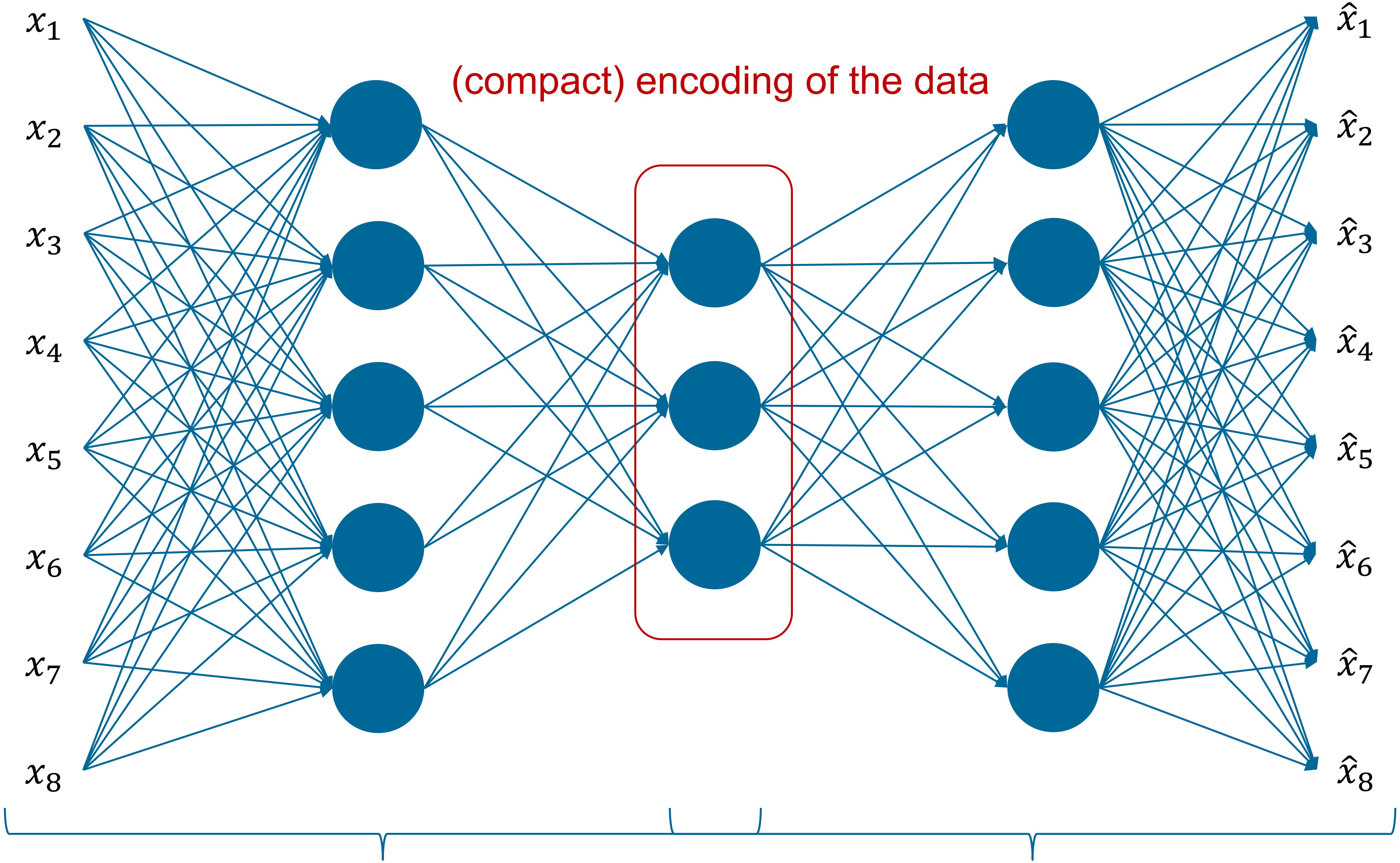
$x \approx g\big(f(x)\big)$

Why we want to “copy” the input

- We don’t care about the copy itself (which should be good, nevertheless)

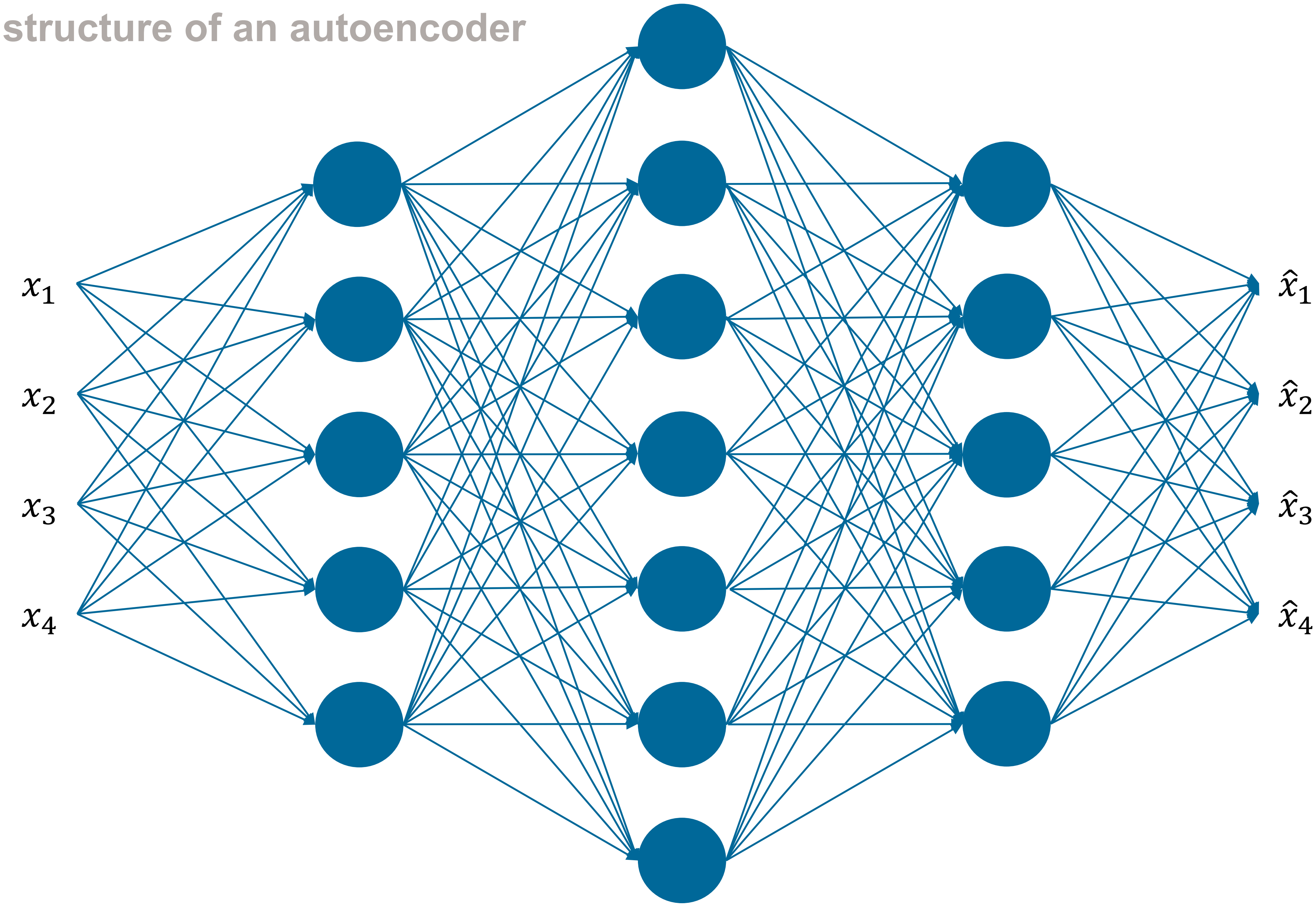
- What we care about is the representation of the copy, $f(x)$
  - Imagine your data has 4096 dimensions (e.g., number of pixels)
  - What if some hidden layer in our network only has 10 dimensions?

The structure of an autoencoder



"Encoder": find a representation $f(x)$ "Decoder": unpack $x = g(f(x))$

NOT the structure of an autoencoder



Uses of autoencoders

- Dimensionality reduction (“advanced PCA”)
- Denoising: train to “recover” data, after artificial noise has been added
- Anomaly detection: train to represent normal data. When data cannot be predicted well, it is likely to be “anormal”
- Generate new content (such as images): variational autoencoders

Remarks on the group assignment

Setting

- Trying to uncover insurance fraud in a car insurance (note: this is real data)
- We have a dataset, but with a key issue: it is "unbalanced"
  → only about 1% of cases are frauds
- Your tasks:
  - A bit of pre-processing
  - Trying out classification using a neural network and identifying the difficulties
  - Explore two approaches to deal with the unbalanced dataset in a better way
    - Synthetically creating new datapoints to make the dataset more balanced
    - Treating the fraud data as anormal and using an autoencoder to detect anormal data
  - A small discussion on transparency of neural network-based approaches
    - Plus, a bonus task

Hints

- Don’t overdo the initial neural network. The point is for you to identify the problem you will run into when using such an unbalanced dataset

- This is an exploratory task. It is more important to show you tried the different approaches and have some intuition on what is happening, than it is to have a fantastic prediction

- The synthetic data generation process is new, but you have all the relevant code available

- Autoencoders are also new, but we will see an example in tomorrow’s tutorial (although not in anomaly detection, so you will need to do some transfer learning)

Good luck with the assignment!

Sources

- Bhaskhar, 2021, Introduction to Deep Learning: https://cs229.stanford.edu/syllabus.html
- DeepLearning.AI, n.d.: deeplearning.ai
- Erdem, 2020, DengAI — Data preprocessing: https://towardsdatascience.com/dengai-data-preprocessing-28fc541c9470
- Géron, 2019, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow
- Goodfellow, Bengio, Courville, 2016, The Deep Learning Book: http://www.deeplearningbook.org
- Liang, 2016, Introduction to Deep Learning: https://www.cs.princeton.edu/courses/archive/spring16/cos495/
- Trehan, 2020, Gradient Descent Explained: https://towardsdatascience.com/gradient-descent-explained-9b953fc0d2c

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON