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

- Issues with learning
- Bias and variance, as well as regularization tools

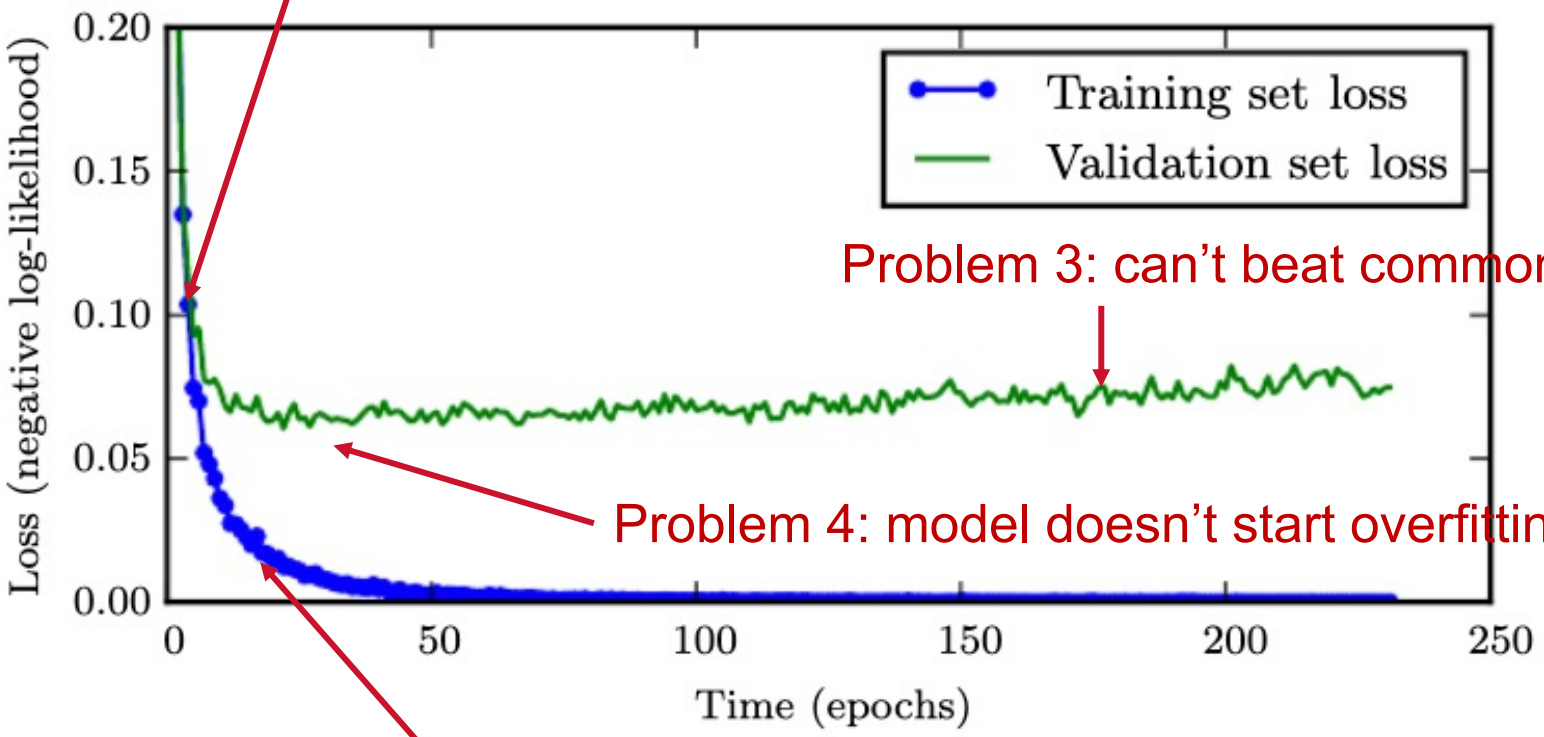
How will we do this?

- 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

Improving learning

A typical training process and possible issues

Problem 1: training loss doesn't go down



Problem 3: can't beat common-sense baseline

Problem 4: model doesn't start overfitting

Problem 2: the loss goes down, but slowly

General issues we might come across in training a neural network

- Problem 1: No training
  - → Learning rate
  - → Batch size
  - → Algorithm choice
- Problem 2: Very slow training
  - → Data normalization
  - → Batch normalization
  - → Initialization
  - → Learning-rate scheduling
  - → Reusing pretrained layers
- Problem 3: Can't beat baseline
  - → Different type of input data
  - → Different type of model
- Problem 4: Model doesn't overfit
  - → Run longer
  - → Increase capacity (e.g., layers, size of layers, etc.)

Improving learning – no training (Problem 1)

Let’s see this in Python

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Problem and solution

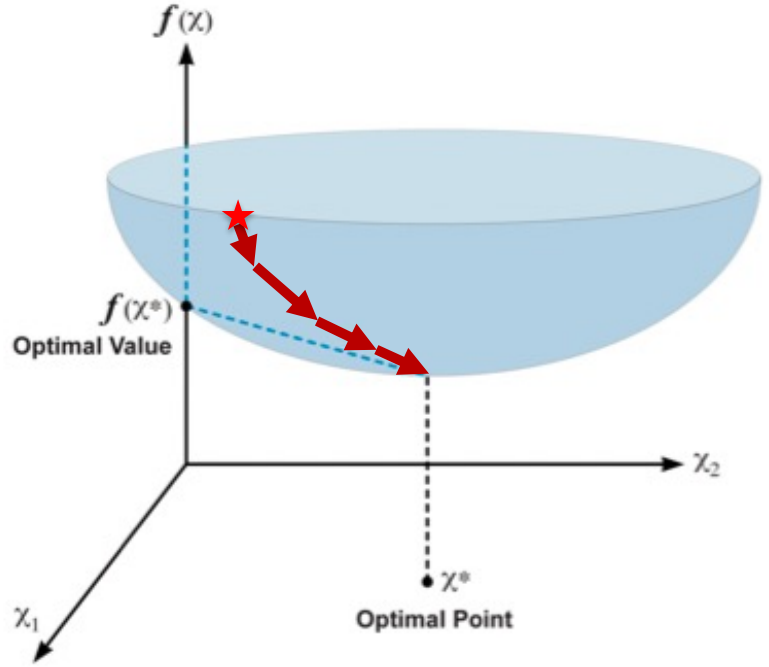
Training doesn't get started or stalls early

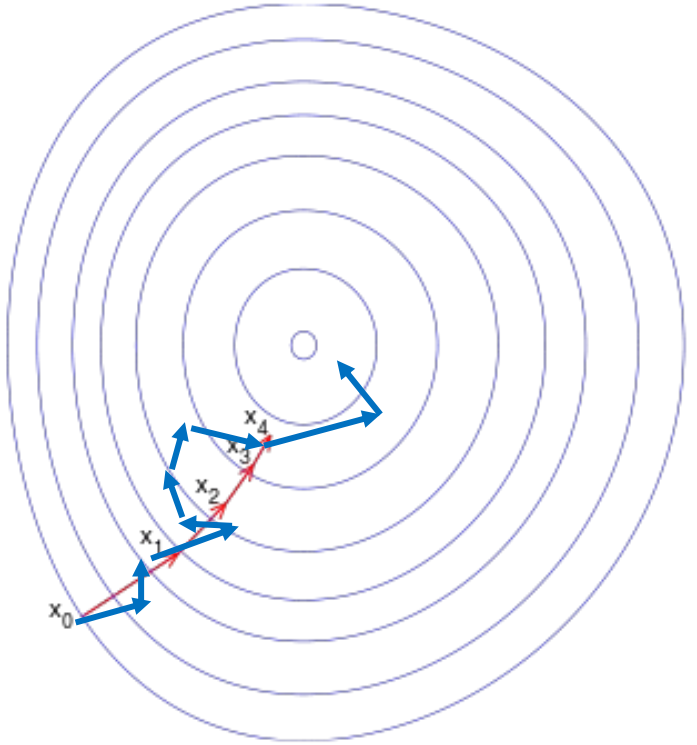
This can always be fixed (you could even learn from random data by simply overfitting)!

The problem lies with the gradient descent process:

- Learning rate
- Batch size
- Choice of optimizer
- Initial values (random initialization)

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

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 $\boldsymbol{\theta} := \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} L^{(i)}$

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

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 32, 64, 128, 256, 512, 1024 …

Gradient descent with momentum

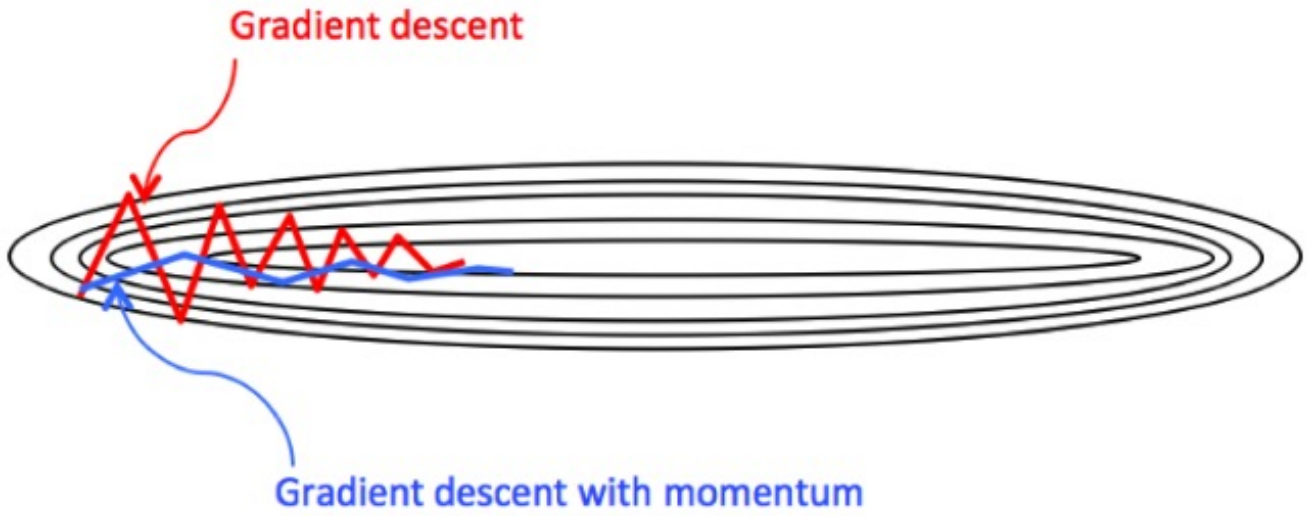
- In gradient descent, we take small, regular steps down a slope
- But if you think of a ball rolling down a slope, it will start slowly, but build up speed (and, thus, momentum) → the "steps" depend not just on the current slope, but on the slope so far!

Gradient descent:

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

Momentum optimization:

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



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

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

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

- Used to be the standard algorithm, but many use Adam instead today

Adam ("Adaptive moment estimation")

- Combining momentum and RMSprop

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

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

$$\hat{m} \coloneqq \frac{m}{1 - \beta_1^t}$$

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

$$\theta \coloneqq \theta + \alpha \frac{\hat{m}}{\sqrt{\hat{s} + \varepsilon}}$$

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”)

Try it out in Python

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Algorithm overview

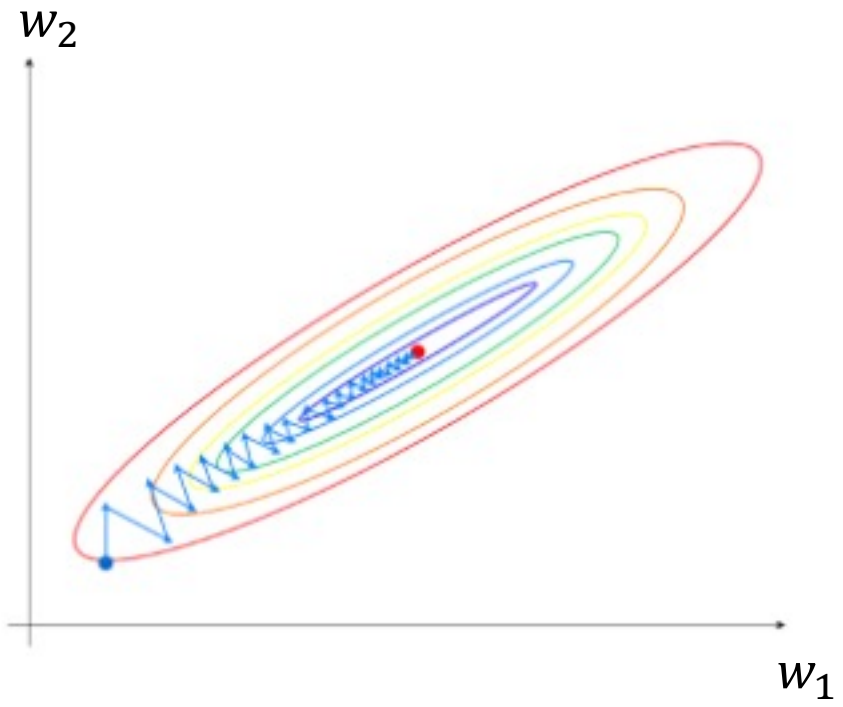
Algorithm	Convergence speed	Convergence quality
SGD	bad	good
SGD with momentum	okay	good
RMSprop	good	medium-good
Adam	good	medium-good

Source: Adapted from Géron

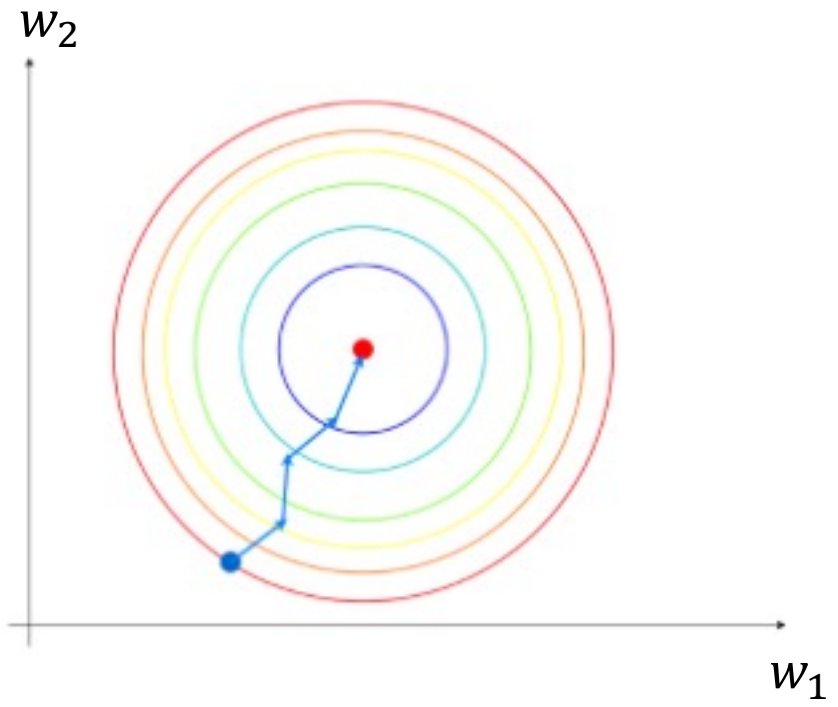
Improving learning – very slow training (Problem 2)

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)

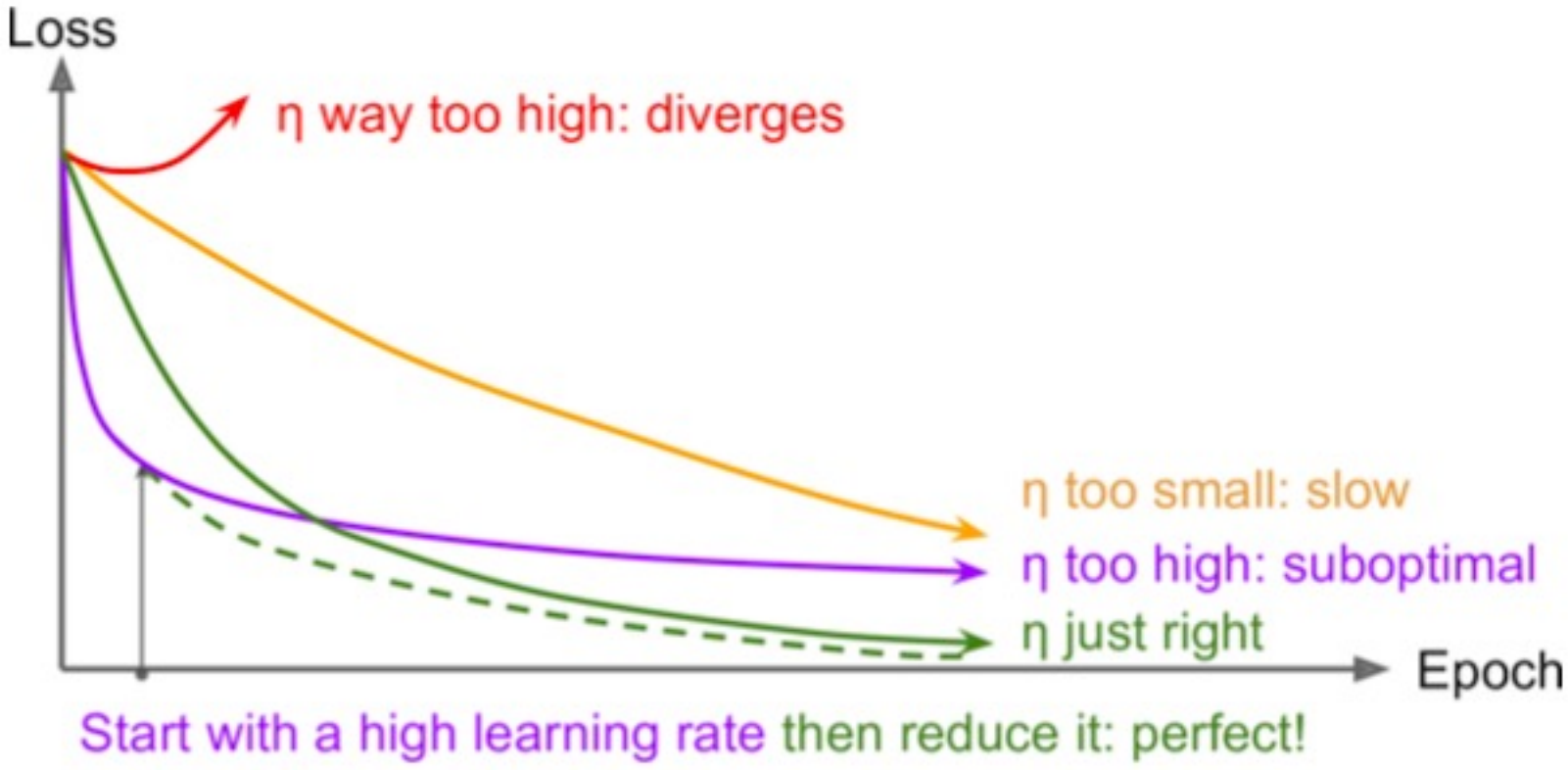
- Overall, for each layer that is batch-normalized, we have $4 \times neurons$ additional parameters

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

Try it out in Python

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

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 $0.1, 0.01, 0.001, \ldots$

Try it out in Python

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

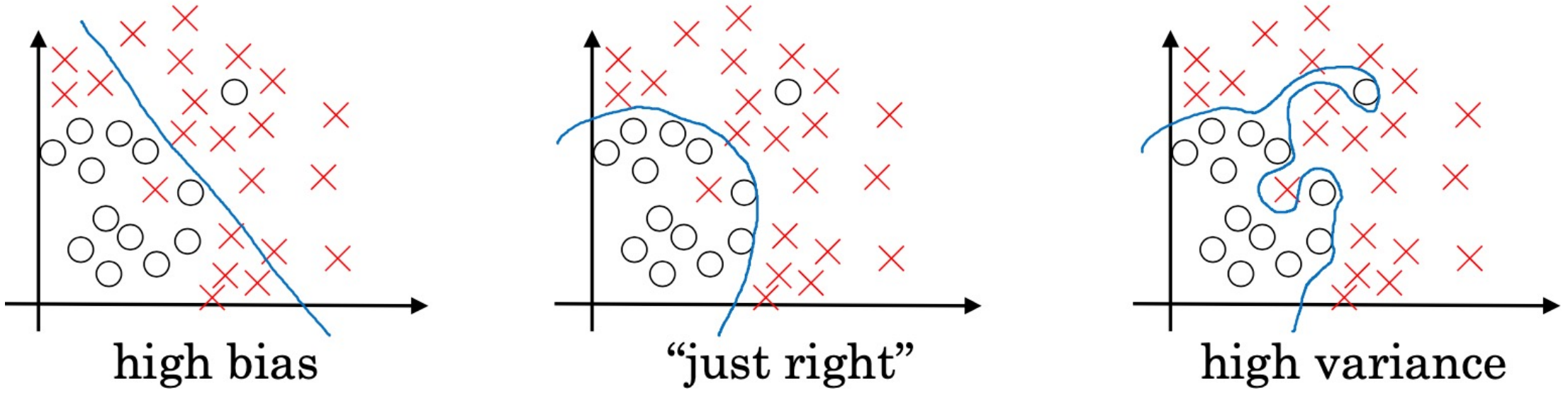
Improving learning – no overfitting occurs (Problem 4)

Let’s see this in Python

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON


Say my model trains, but…

Bias and variance



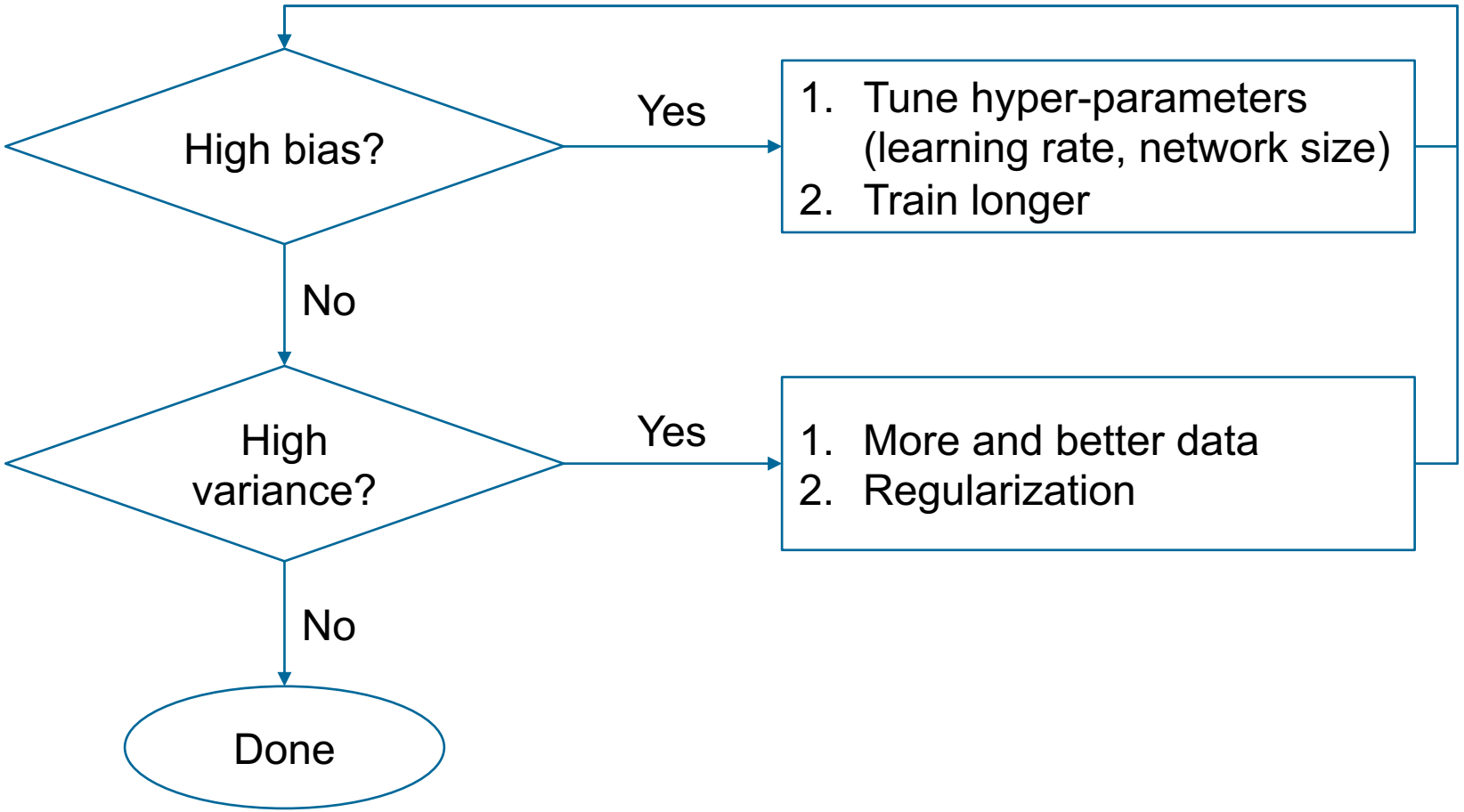
Source: DeeplearningAI

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

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

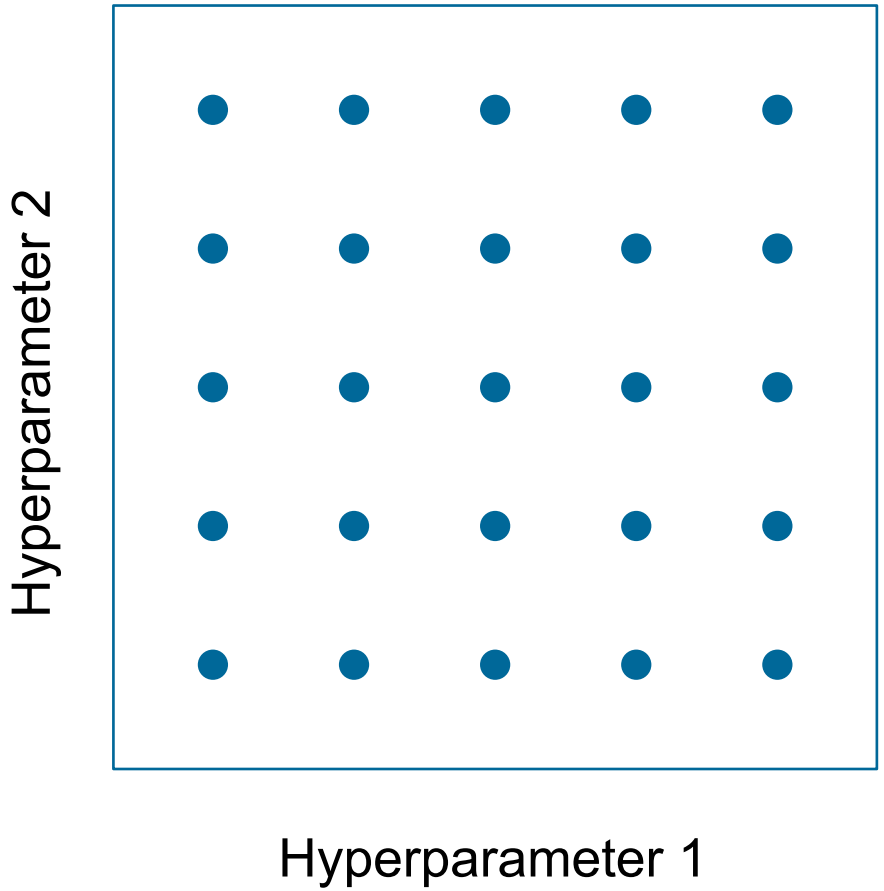
How to deal with bias and variance

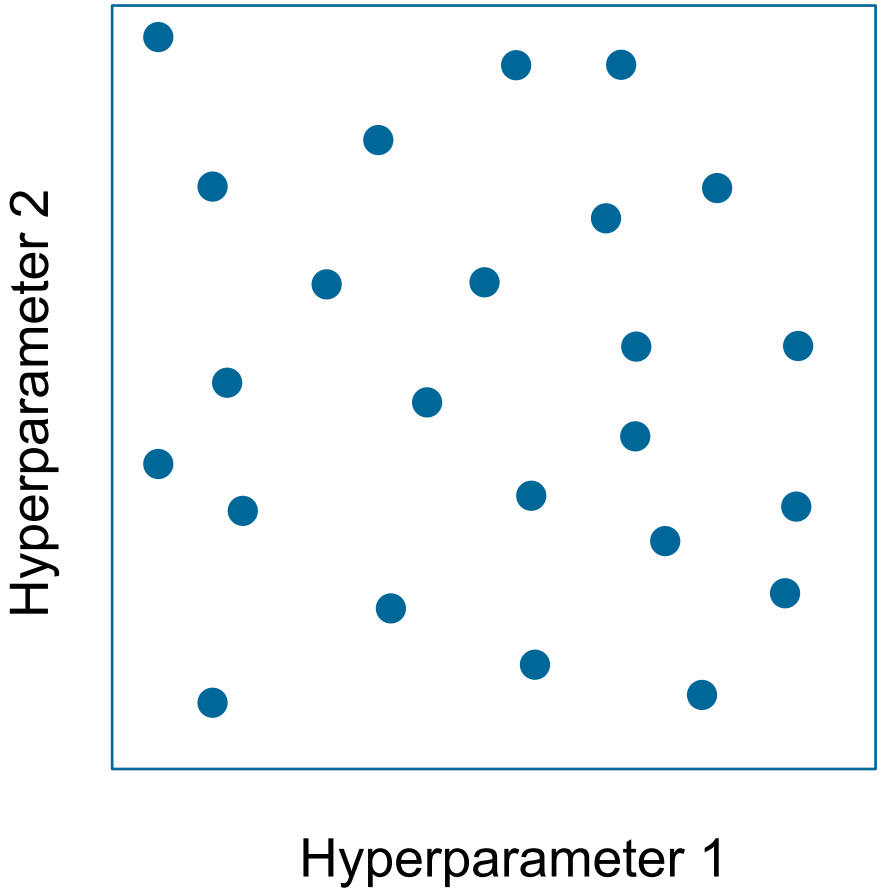


Source: DeeplearningAI

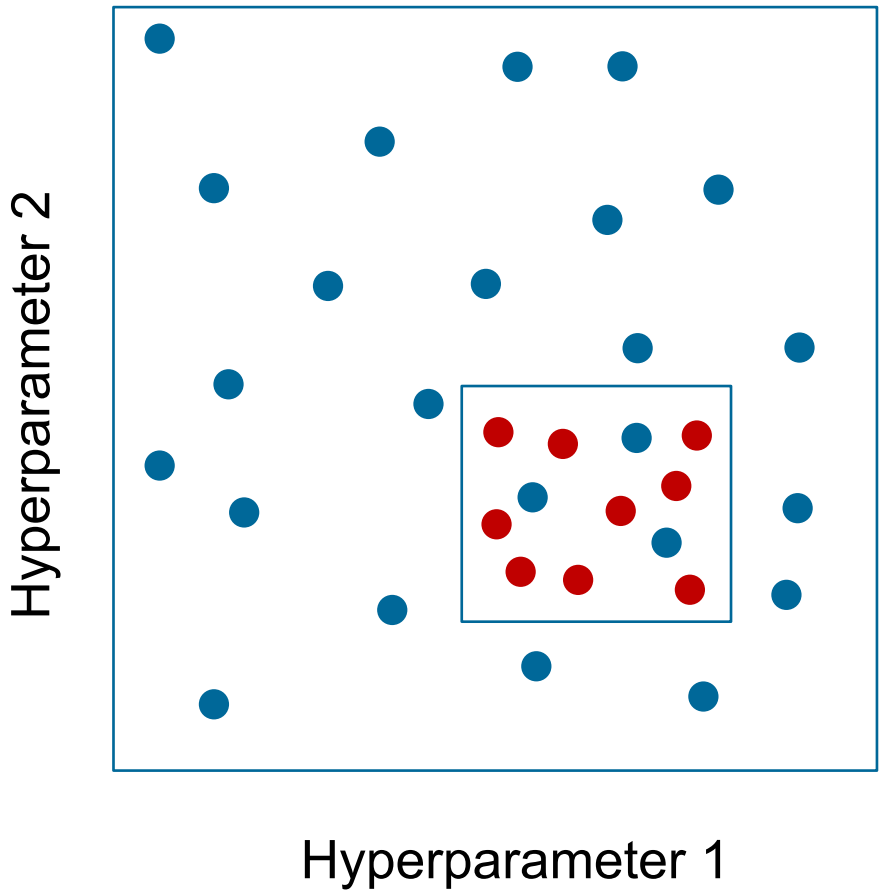
High bias – hyperparameter tuning

Hyperparameter tuning process rule 1: Choose random combinations





Hyperparameter tuning process rule 2: Go from coarse to fine



BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

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 $0.001, \ldots, 1$
- 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
  → $r = -3 \times np.random.rand(0,1)$
  → $\alpha = 10^r$

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)

Try it out in Python

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

High variance – data and dataset augmentation

Some overfitting issues caused by the data

- Noise
  - E.g., measuring errors
  - The model may try to fit to the observed data, rather than what the data should be

- Data with natural ambiguities
  - E.g., trying to predict if an employee's performance will be "bad", "intermediate", or "good"
  - The model may be overconfident, based on the ambiguity realizations in the training data, e.g., if all data comes from the same manager's assessments

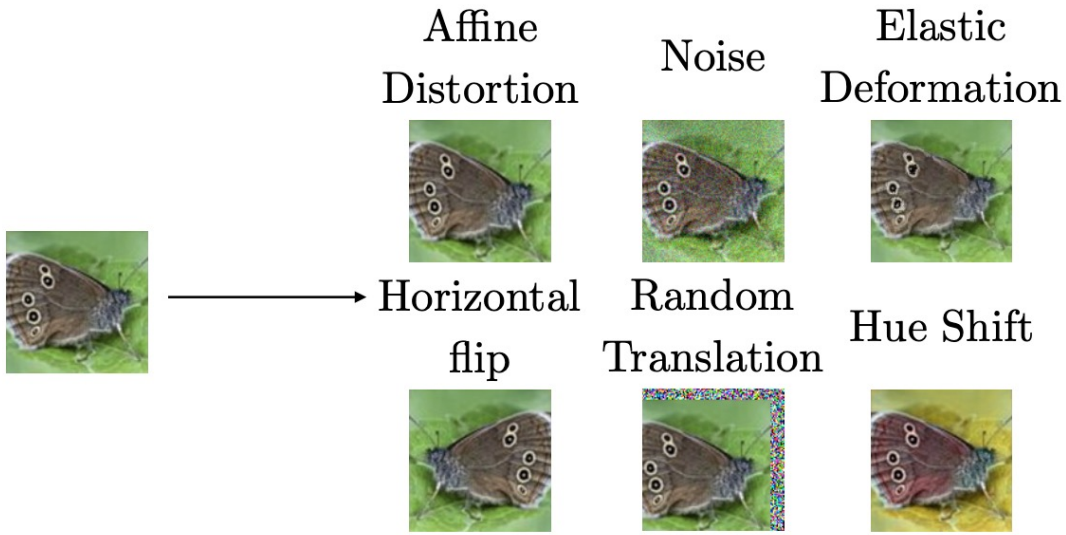
- Rare features and spurious correlations
  - E.g., 5% of stadium visitors are known hooligans. In your data set of 10,000 fans, there are three from Liechtenstein, one is a known hooligans.
  - The model will likely predict that fans from Liechtenstein are much more likely to be hooligans than the average.

Let’s see spurious correlations in action

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

The importance of good data

- The best way to improve our models is to use more and better data!
- We can often make the dataset better: minimize labeling errors, clean data, deal with missing values, select features, engineer simpler / more generalizable features
- However, we usually cannot (easily) get more data. Then, we **augment our data set**
- This doesn't give more information to the model, but it helps our model extract more of the information already available



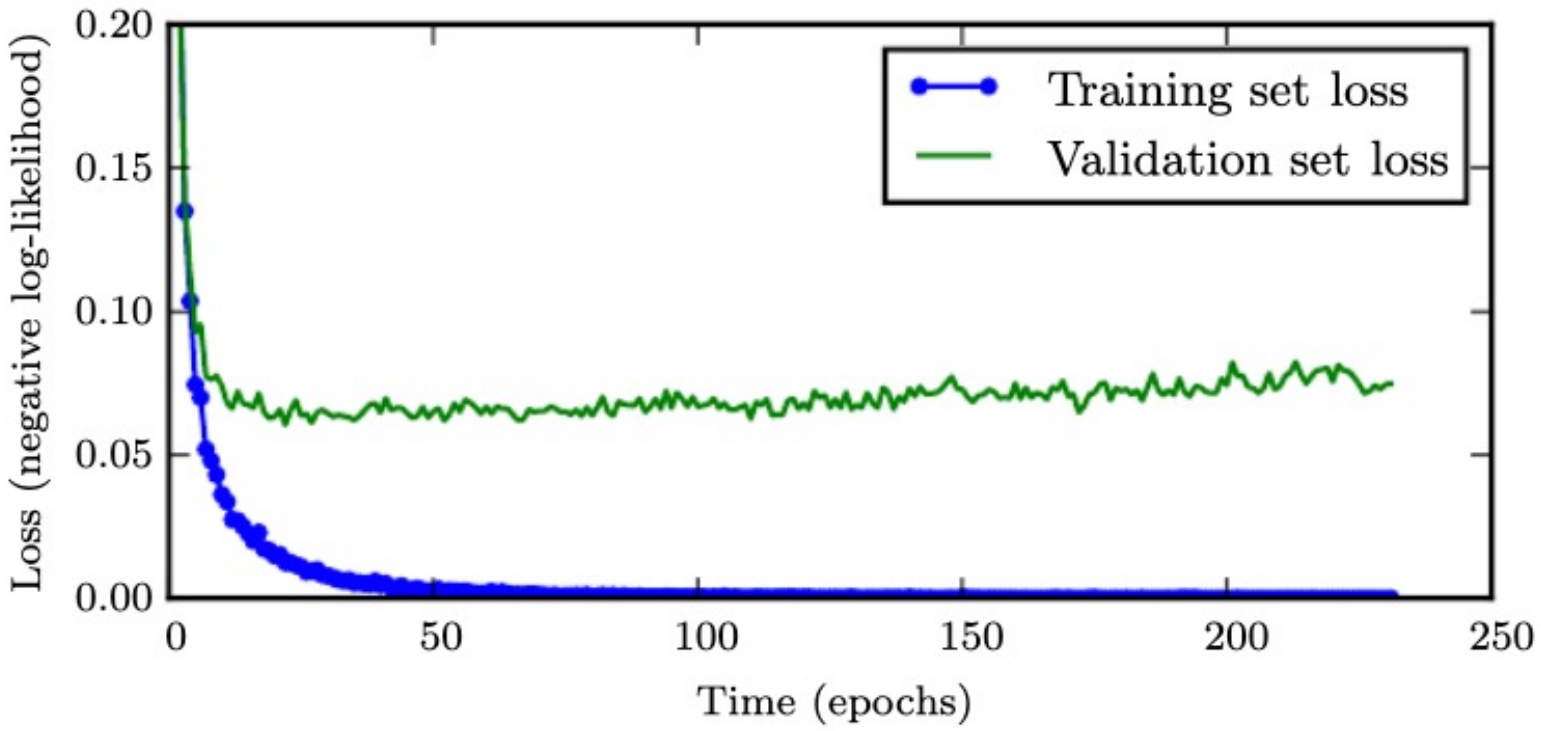
Source: Goodfellow

High variance – regularization

If we don’t have enough data to get the full picture, we focus on the prominent features

- Often, we cannot get as much data as we would like to have. Also, data augmentation only helps within limits
- Then, we want to make sure that our model focuses on the most important patterns, which are more likely to generalize
- This is the idea behind regularization

Early stopping



Source: Goodfellow

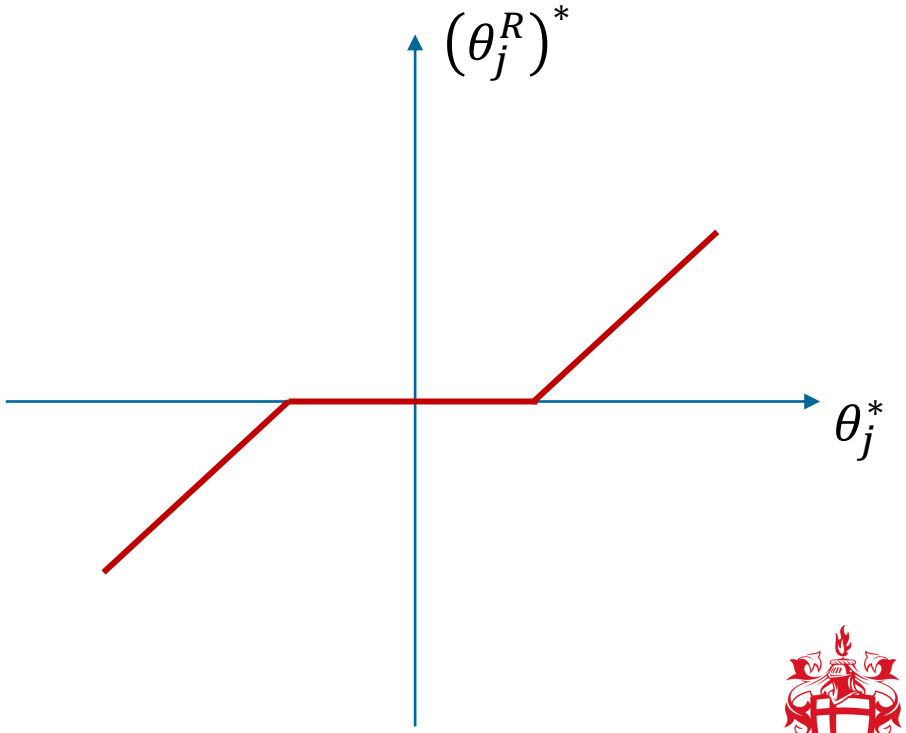
L1-regularization (“Lasso regression”)

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

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

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

→ “sparsity”



BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

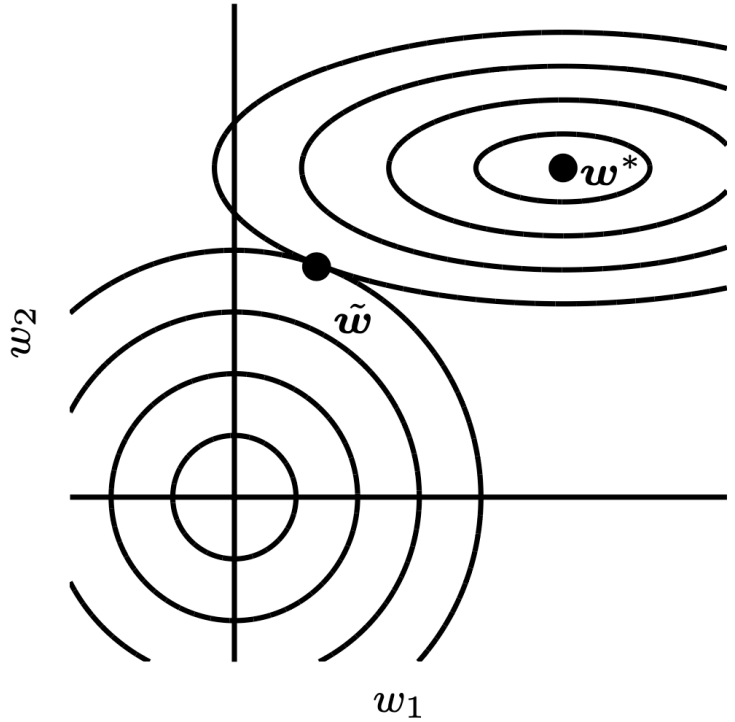
L2-regularization ("Ridge regression")

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

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

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

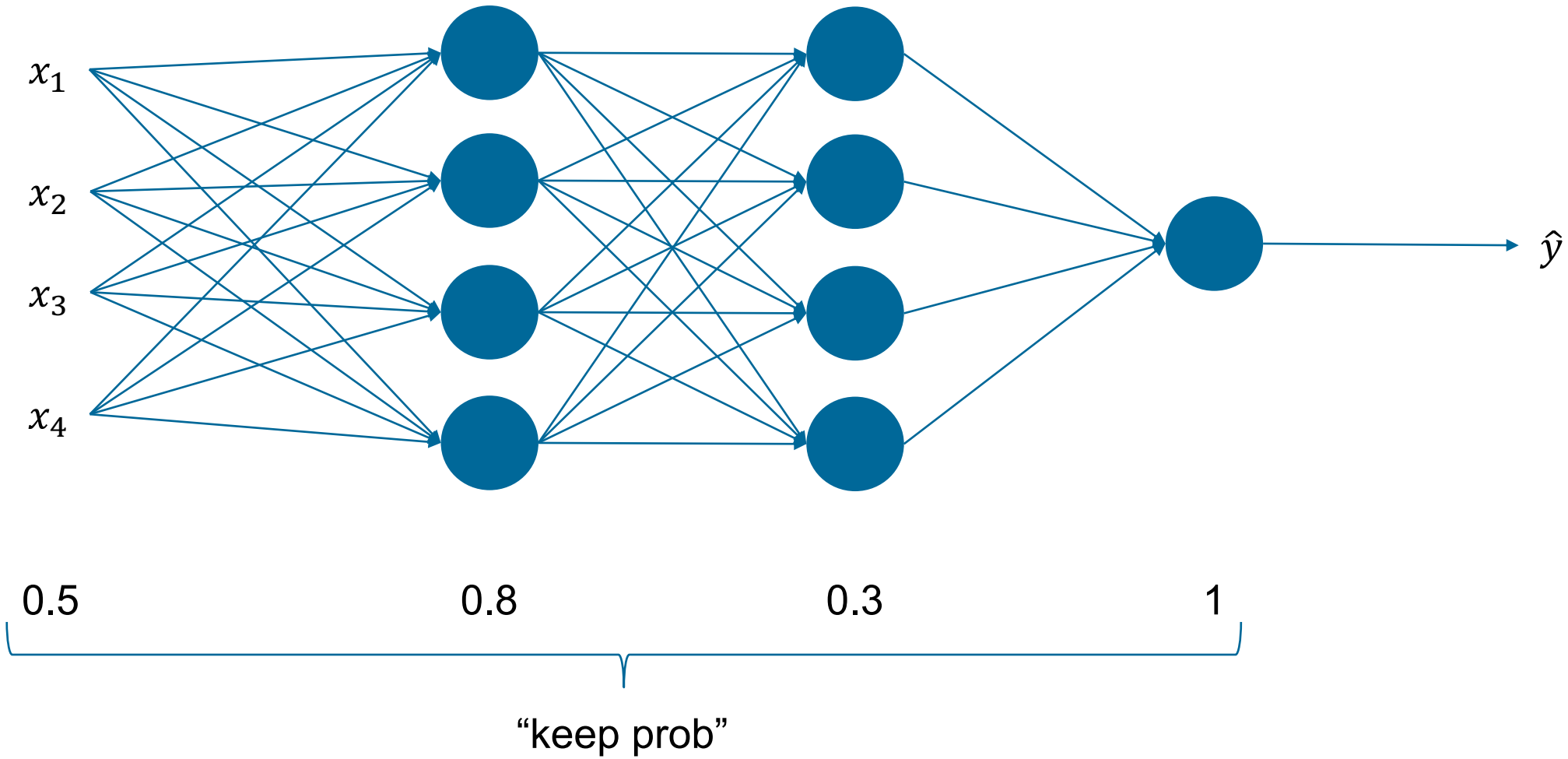
→ "weight decay"



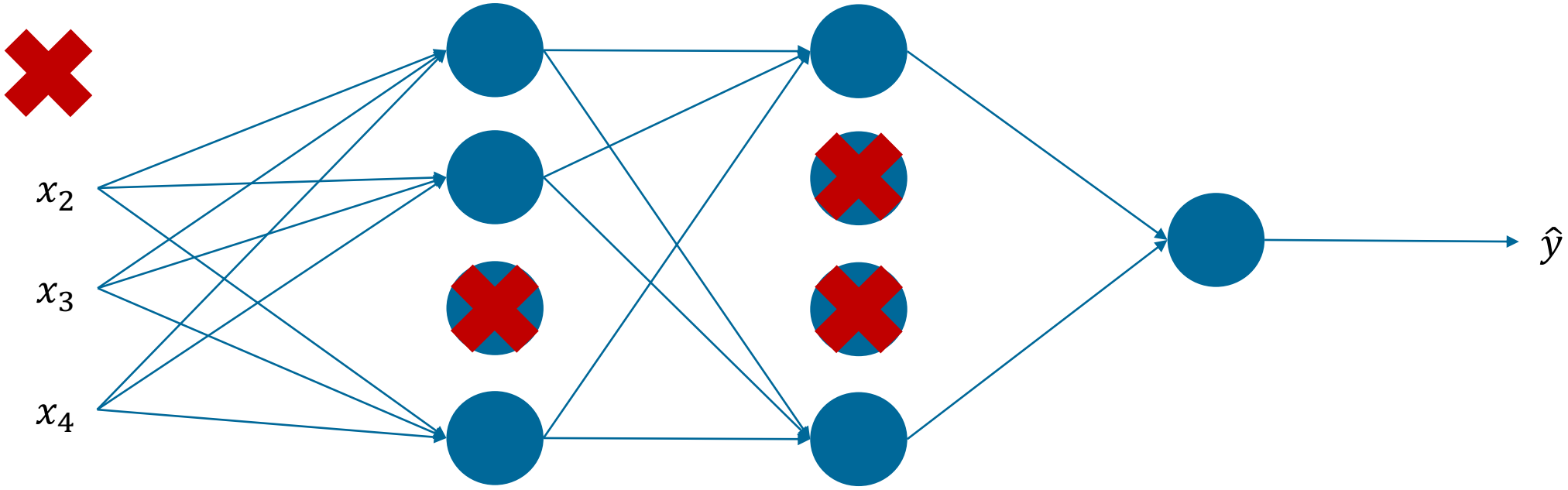
Source: Goodfellow

BAYES BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

Dropout regularization



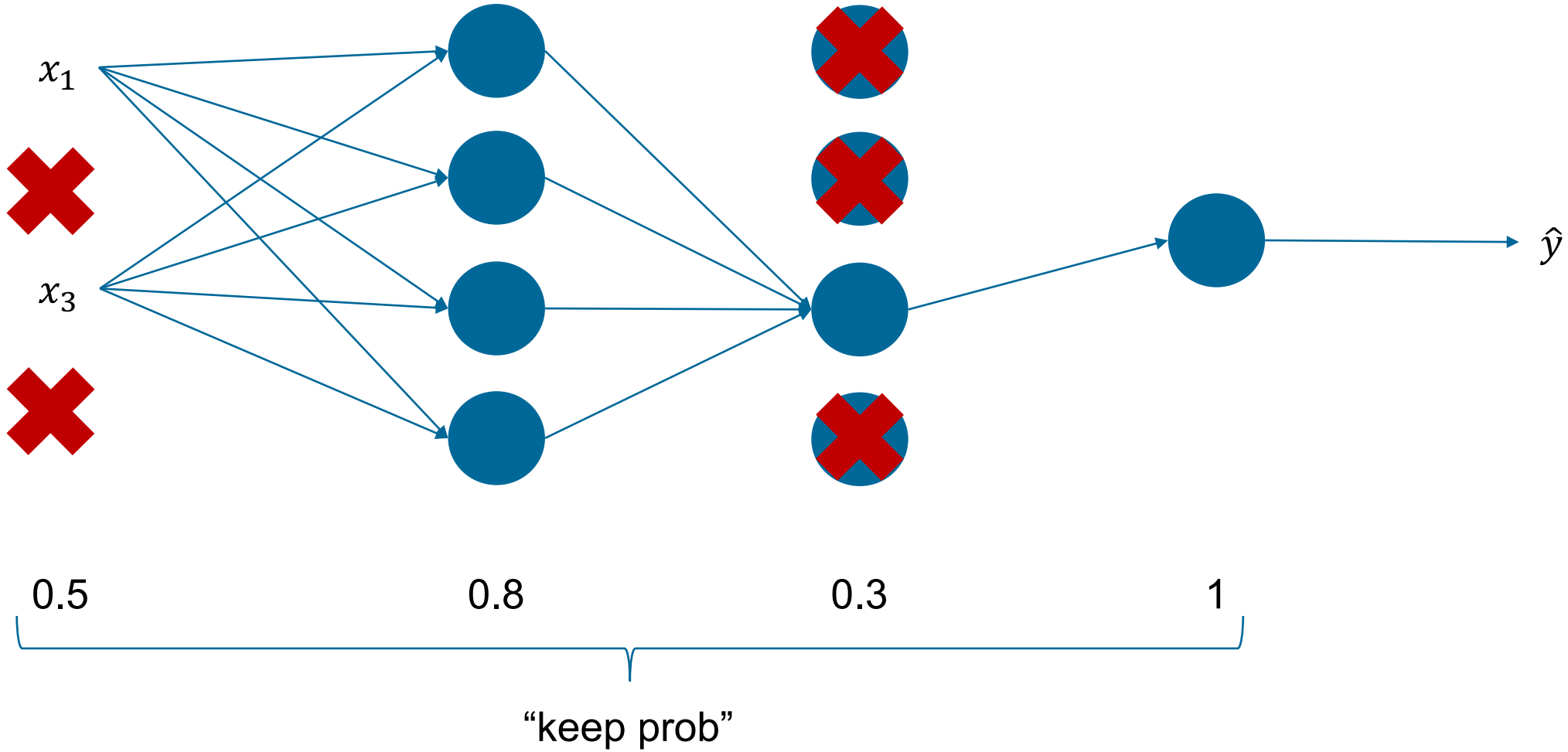
Dropout regularization



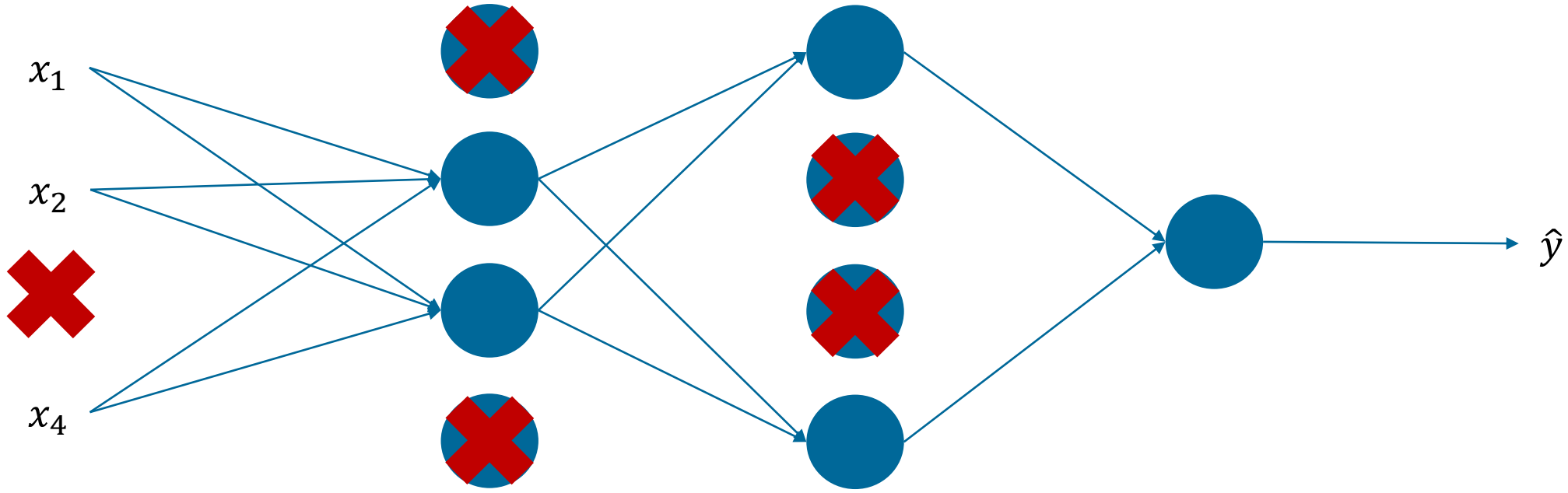
0.5 0.8 0.3 1

"keep prob"

Dropout regularization



Dropout regularization



0.5 0.8 0.3 1

"keep prob"

Try it out in Python

BAYES
BUSINESS SCHOOL
CITY, UNIVERSITY OF LONDON

The bigger picture: building a neural network as part of an ML project

The universal workflow of machine learning projects

1. Define the task
   - What is the problem? Why is that even a problem?
   - What data is available?
   - What type of algorithms and models are useful for this?
   - Are there existing solutions already?
   - What are constraints?
2. Collect the data
   - The most arduous, time-consuming part
   - This is usually where your time is best spent
   - Clean your data set, create relevant features
3. Develop a model
   - Choose a measure of success and a non-trivial baseline, as well as a strategy for evaluation
   - Beat the baseline with a small model
   - Develop a bigger model that overfits
   - Tune and regularize your model
4. Deploy the model

Source: Chollet (adjusted)


See you tomorrow!

Sources

- Bhaskhar, 2021, Introduction to Deep Learning: https://cs229.stanford.edu/syllabus.html
- Chollet, 2021, Deep Learning with Python (2nd edition)
- 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