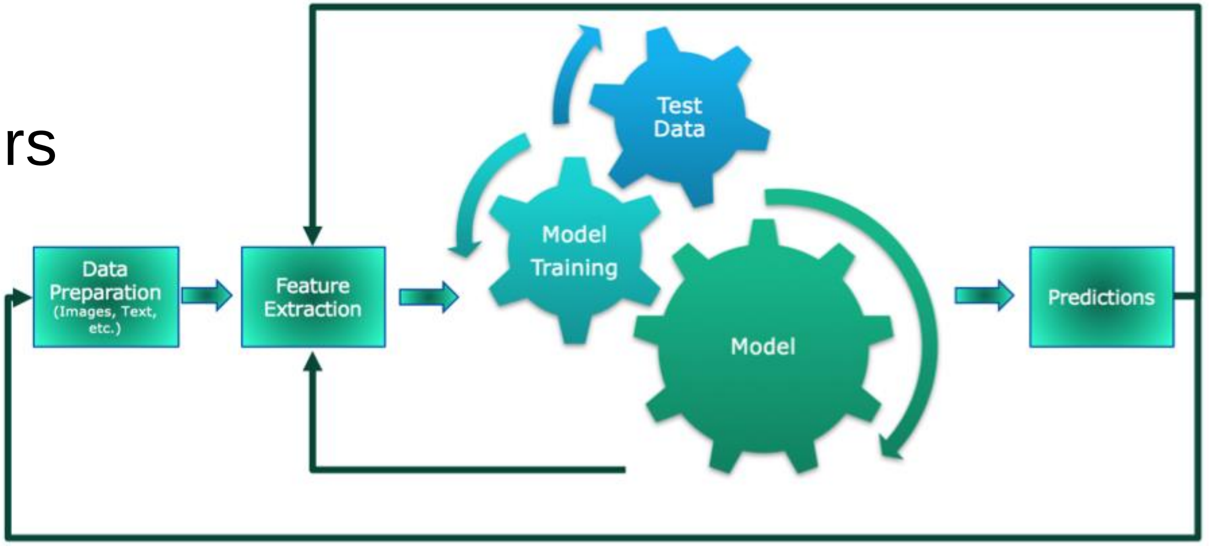
Deep Learning

Lecture 2
Best Practices

Giovanni Chierchia

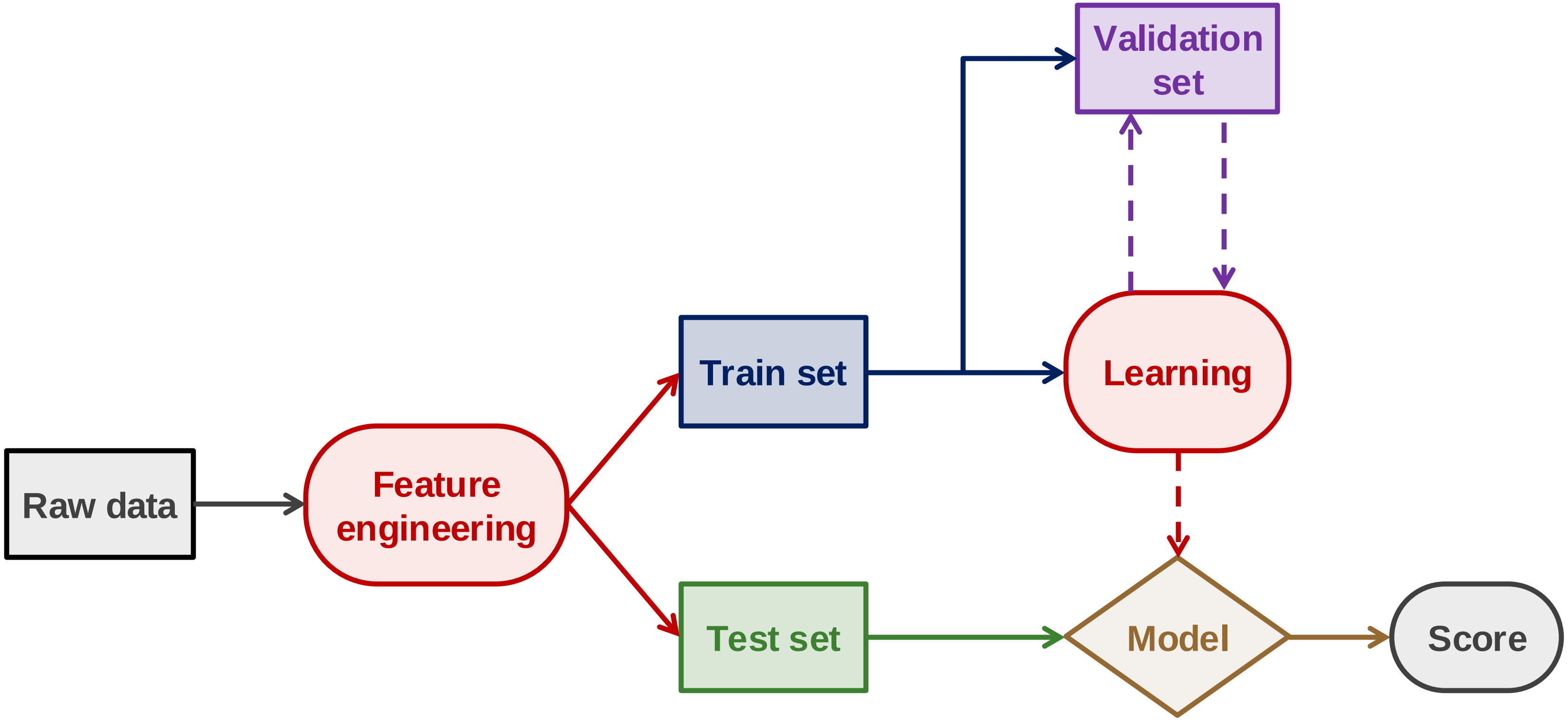
Table of contents

- Training a neural network
- Optimization algorithms
- Feature engineering
- Overfitting
- Regularization
- Hyperparameters



Machine learning system

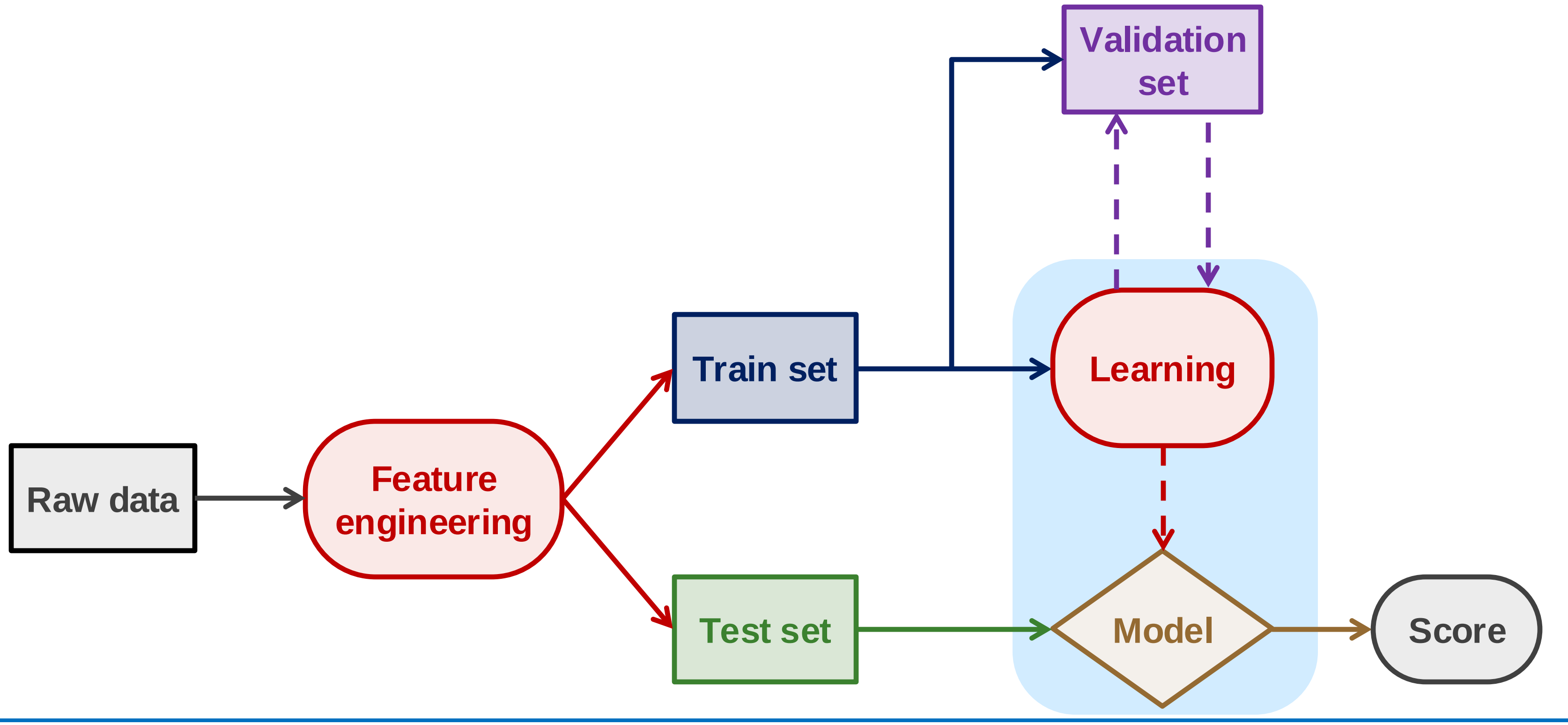
- **Training pipeline**



Neural network training

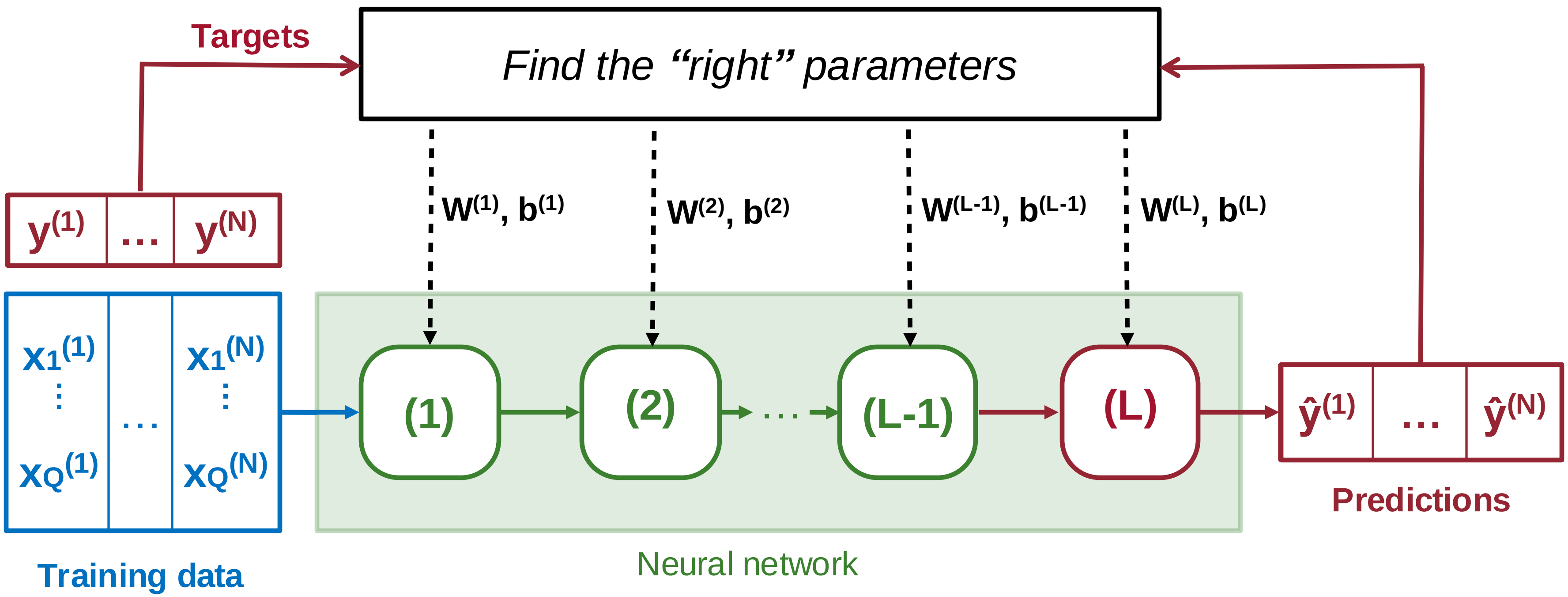
Machine learning system

- **Training pipeline**
  - How to make it work in practice?



Supervised learning

- **Goal** ➜ Train the network on the training data
  - Find the parameters that make predictions similar to targets



Training (1/2)

- How to select the **right values** for the parameters?
  - Minimize the mean error of prediction on the training data

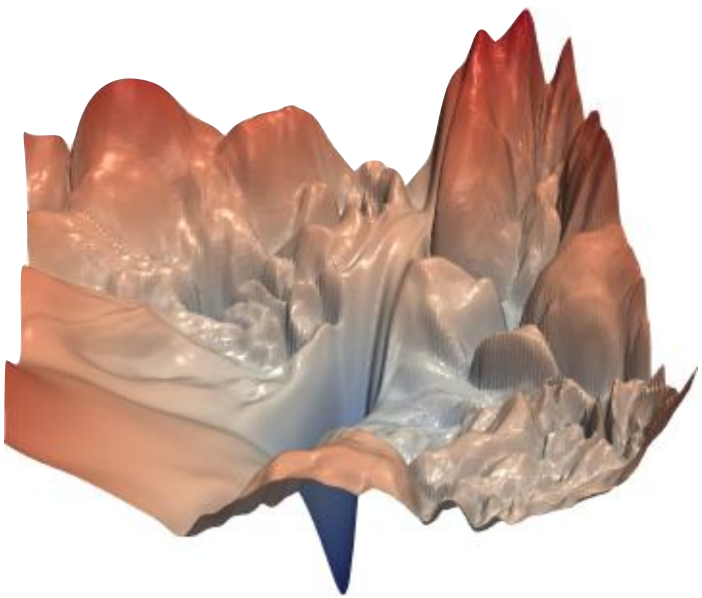
$$\underset{\theta}{\text{minimize}} \quad \frac{1}{N}\sum_{n=1}^{N} \mathcal{E}\Big(f_\theta(\mathbf{x}^{(n)}), \mathbf{y}^{(n)}\Big)$$

Parameters of the network

Output of the network

n-th sample in the training data

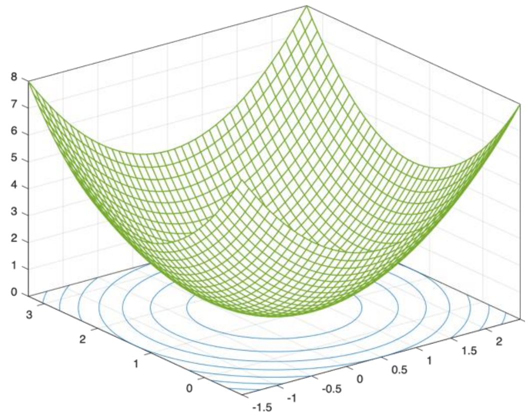
Error of prediction



Training (2/2)

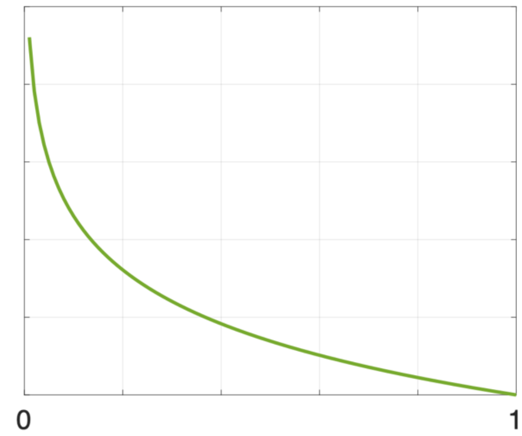
- The error of prediction is measured by a loss function
  - **Regression ➜** Euclidean distance

$$\mathcal{E}\big(f_\theta(\mathbf{x}), \mathbf{y}\big) = \big\| f_\theta(\mathbf{x}) - \mathbf{y} \big\|^2$$



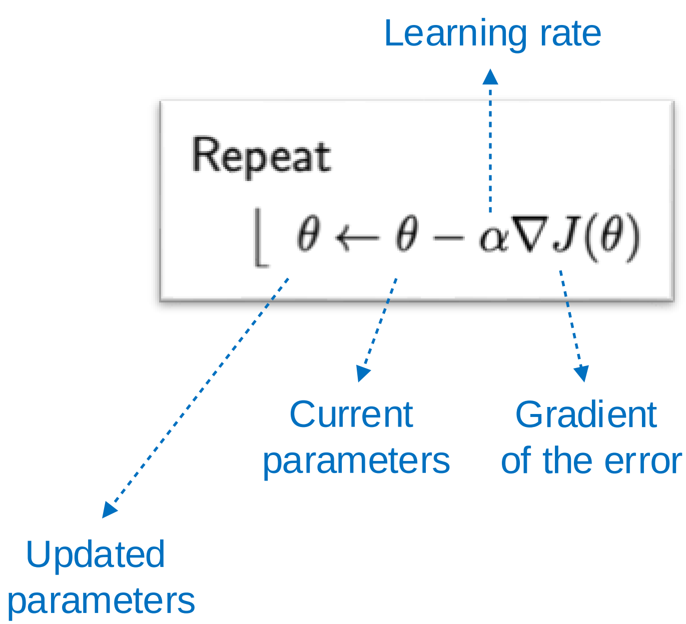
  - **Classification ➜** Cross entropy

$$\mathcal{E}\big(f_\theta(\mathbf{x}), \mathbf{y}\big) = -\mathbf{y}^\top \log\big(f_\theta(\mathbf{x})\big)$$

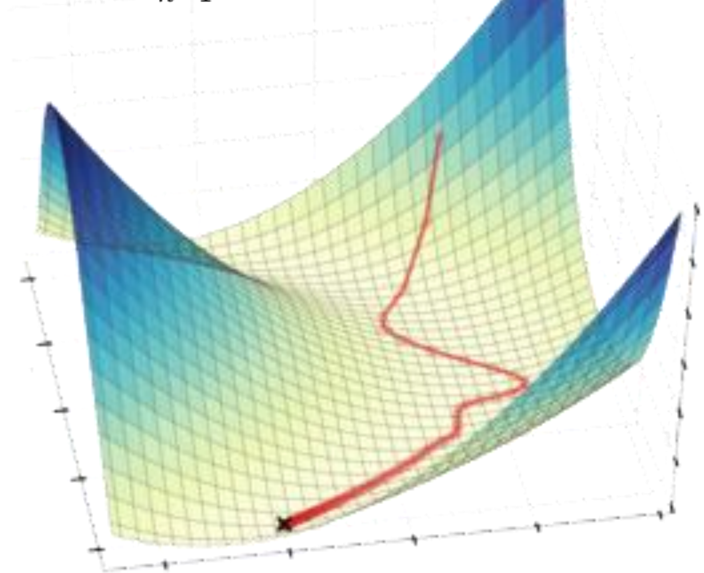


Gradient descent (1/2)

- How to minimize the mean error of prediction ?
  - By using a numerical algorithm called **gradient descent**



$$J(\theta) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}\Big(f_\theta\big(\mathbf{x}^{(n)}\big), \mathbf{y}^{(n)}\Big)$$

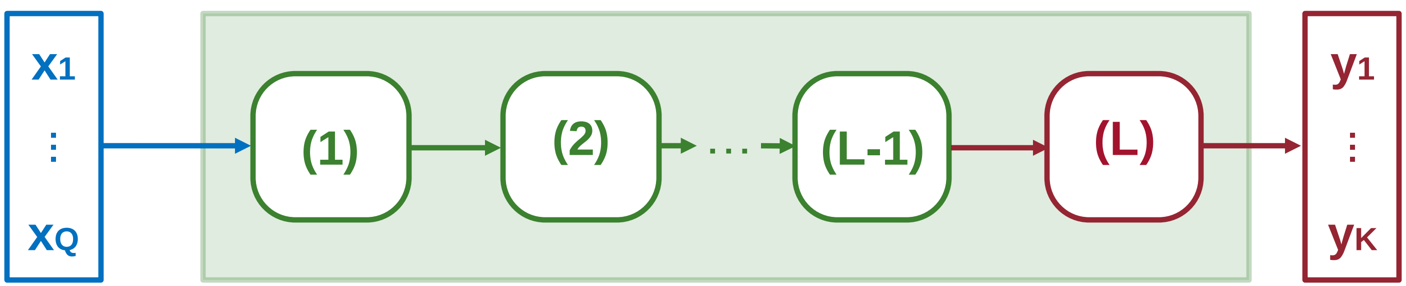


Gradient descent (2/2)

- **Be aware →** Gradient descent has multiple pitfalls !!!
  - **Choice of the learning rate**
    - *The network learns nothing if the learning rate is not sufficiently small*
  - **Convergence to local minima or saddle points**
    - *The network may not learn correctly, even if it is capable of doing so*
  - **Dependence on the data**
    - *The network learns very slowly if the data are not preprocessed*

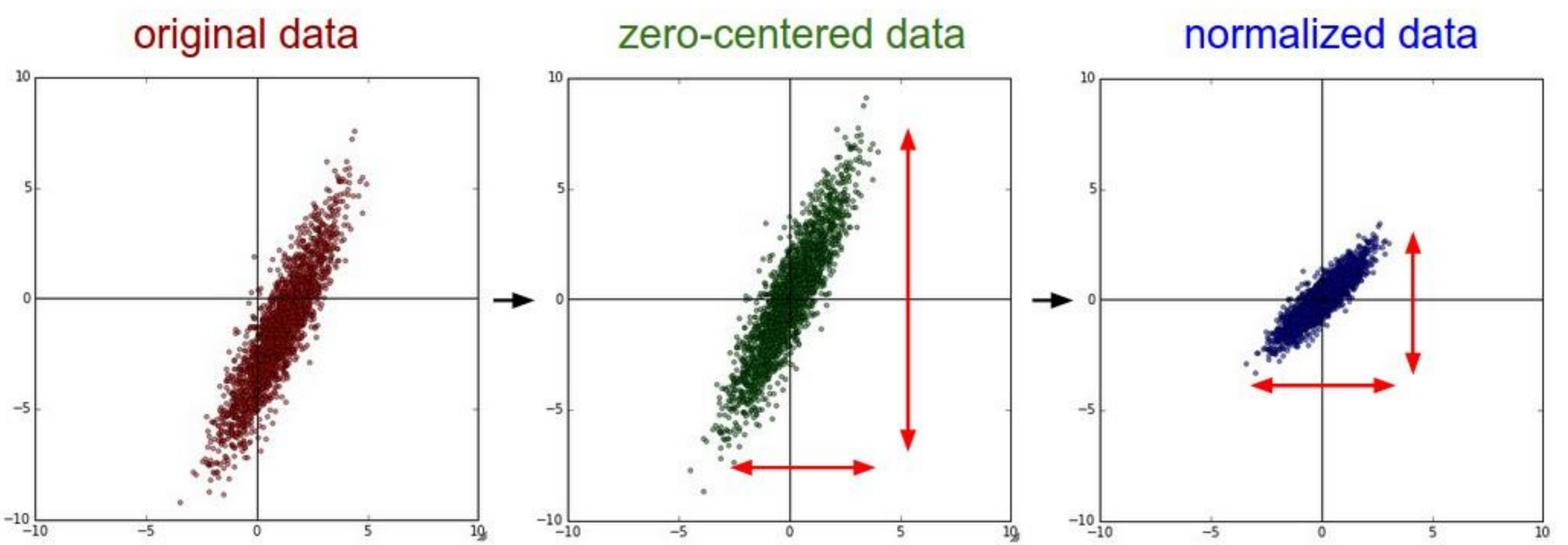
Tricks of the trade (1/3)

- The network parameters must be **randomly initialized**
  - If the parameters were initialized to zero, each neuron in the hidden layers would perform the same computation…
  - … so even after multiple iterations of gradient descent, all the neurons would be computing the same thing over and over.
  - **Note →** Random initialization introduces diversity in the ensemble



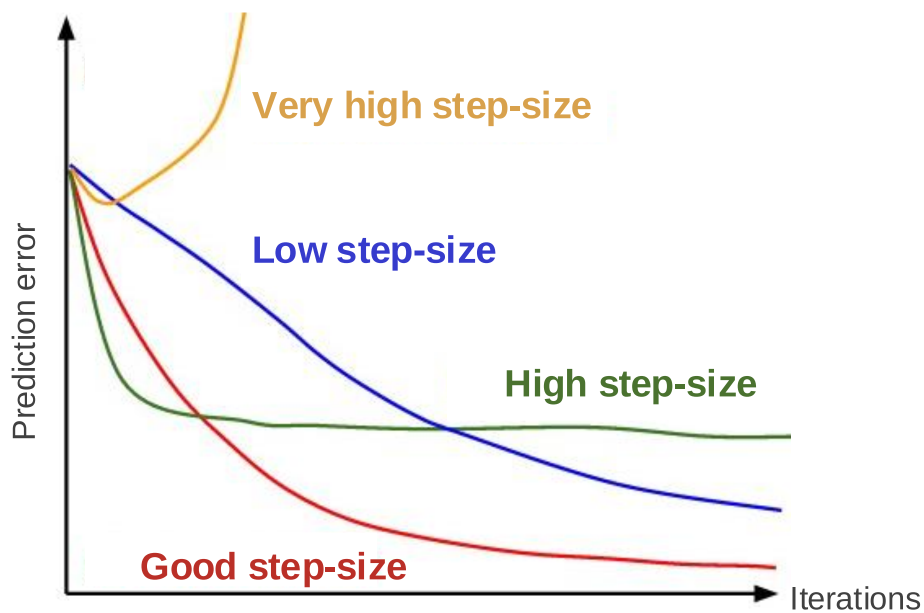
Tricks of the trade (2/3)

- Data must be normalized before the network input
  - **Standard ➔** Subtract the mean and divide by the variance
  - **Min-max ➔** Map the min-max values into the range 0-1
  - …



Tricks of the trade (3/3)

- Track the prediction error during training
  - Reduce the learning rate if the curve stagnates early or goes up
  - Increase the learning rate if the curve goes down too slowly



Training with mini-batches (1/3)

- **Gradient descent** ➜ Full batch
  - The prediction error is computed on all the training set

$$J(\theta) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{E}\Big(f_\theta\big(\mathbf{x}^{(n)}\big), \mathbf{y}^{(n)}\Big)$$

$x^{(1)}$	$y^{(1)}$
$x^{(2)}$	$y^{(2)}$
$x^{(3)}$	$y^{(3)}$
$x^{(4)}$	$y^{(4)}$
…	…
$x^{(n)}$	$y^{(n)}$
…	…
$x^{(N)}$	$y^{(N)}$

Training set

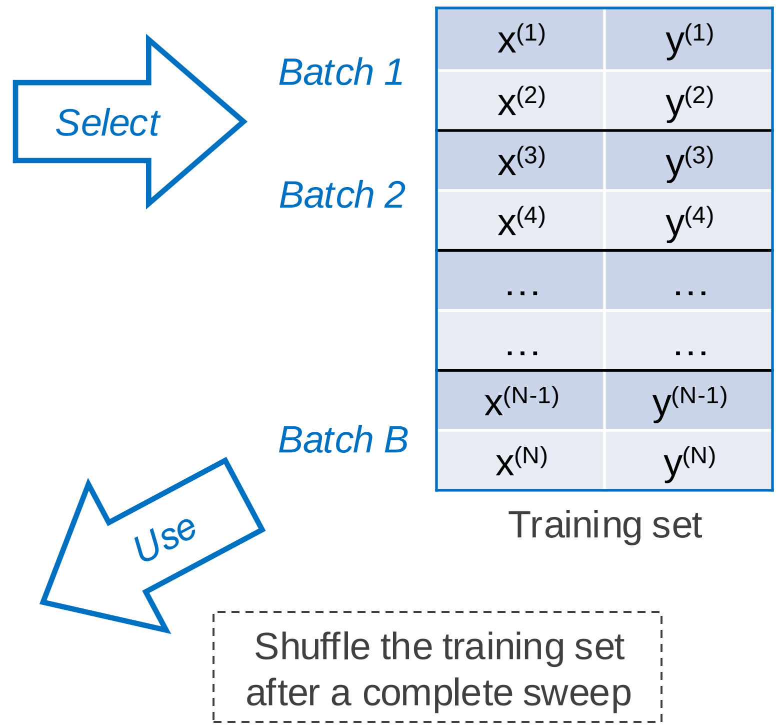
  - This requires intensive computation during training, as gradient descent must process all the training data at each iteration

$$\theta \leftarrow \theta - \frac{\alpha}{N} \sum_{n=1}^{N} \nabla \mathcal{E}\Big(f_\theta\big(\mathbf{x}^{(n)}\big), \mathbf{y}^{(n)}\Big)$$

Training with mini-batches (2/3)

- **Gradient descent ➜** Mini-batches
  - The prediction error is computed on a mini-batch of data

$$J_i(\theta) = \frac{1}{|\mathcal{N}_i|} \sum_{n \in \mathcal{N}_i} \mathcal{E}\Big(f_\theta\big(\mathbf{x}^{(n)}\big), \mathbf{y}^{(n)}\Big)$$



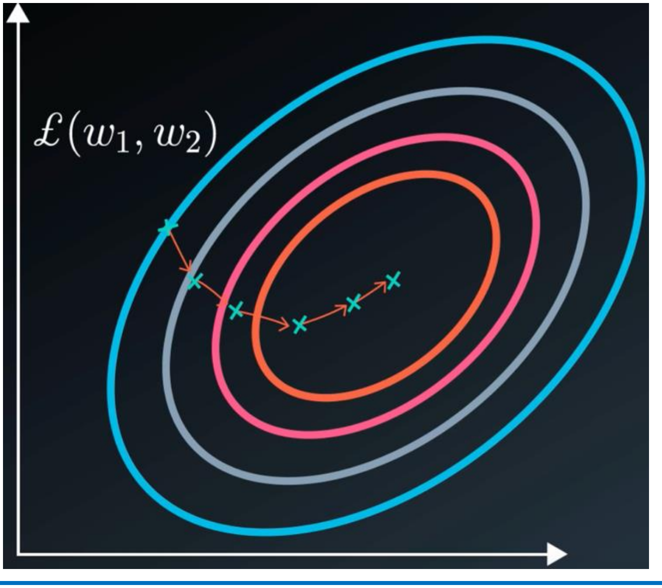
  - Use a different mini-batch at each iteration of gradient descent

$$\theta \leftarrow \theta - \frac{\alpha}{|\mathcal{N}_i|} \sum_{n \in |\mathcal{N}_i|} \nabla \mathcal{E}\Big(f_\theta\big(\mathbf{x}^{(n)}\big), \mathbf{y}^{(n)}\Big)$$

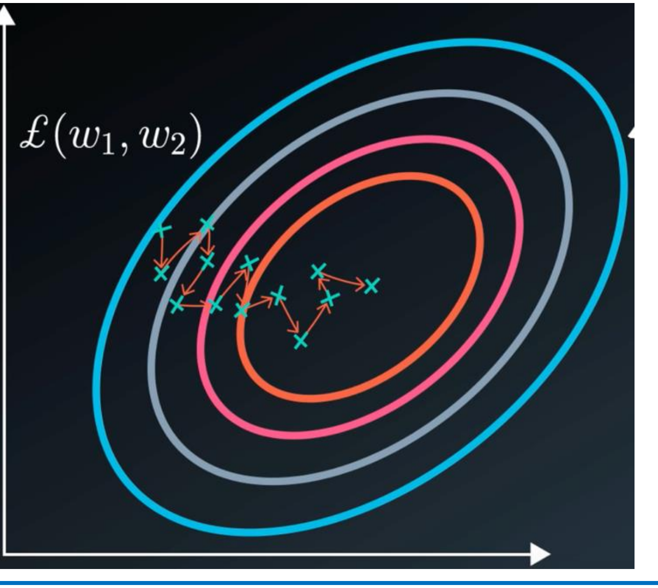
Training with mini-batches (3/3)

- Stochastic gradient **approximates** the “true” gradient
  - Hence, it does not indicate the fastest way to update parameters
  - Training must take many smaller steps (instead of few large ones)

Gradient descent – Full batch



Gradient descent – Mini batches



Quiz

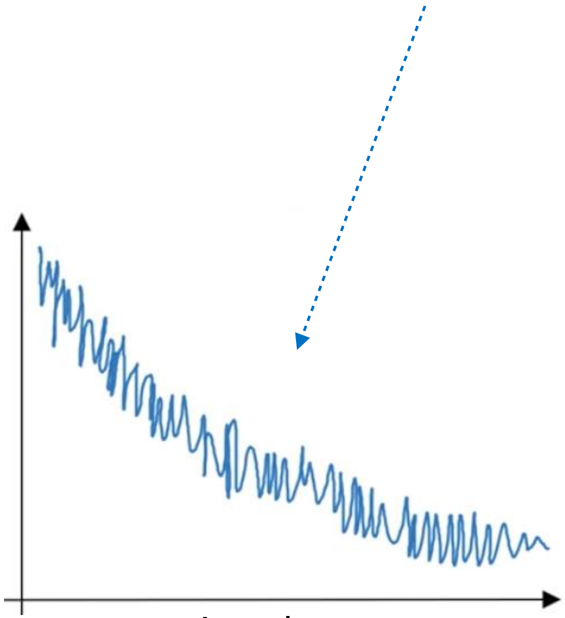
- Assume you tracked the cost function **J(θ)** during training, and the plot versus the number of iterations looks like this.

  1) *If you're using stochastic gradient descent, something is wrong. But if you're using gradient descent, this looks acceptable.*

  2) *Whether you're using standard or stochastic gradient descent, this looks acceptable.*

  3) *If you're using stochastic gradient descent, this looks acceptable. But if you're using gradient descent, something is wrong.*

  4) *Whether you're using standard or stochastic gradient descent, something is wrong.*



Summary so far

- Neural networks are trained with gradient descent

$$\text{Repeat} \quad \lfloor\ \theta \leftarrow \theta - \alpha \nabla J(\theta)$$

- Tricks of the trade

1) *Data normalization* - - - - → *Speed up the optimization*
2) *Random initialization* - - - - → *Otherwise, the network won't learn*
3) *Learning rate* - - - - → *Must be chosen small enough*
4) *Mini-batches* - - - - → *Better generalization*

Optimization algorithms

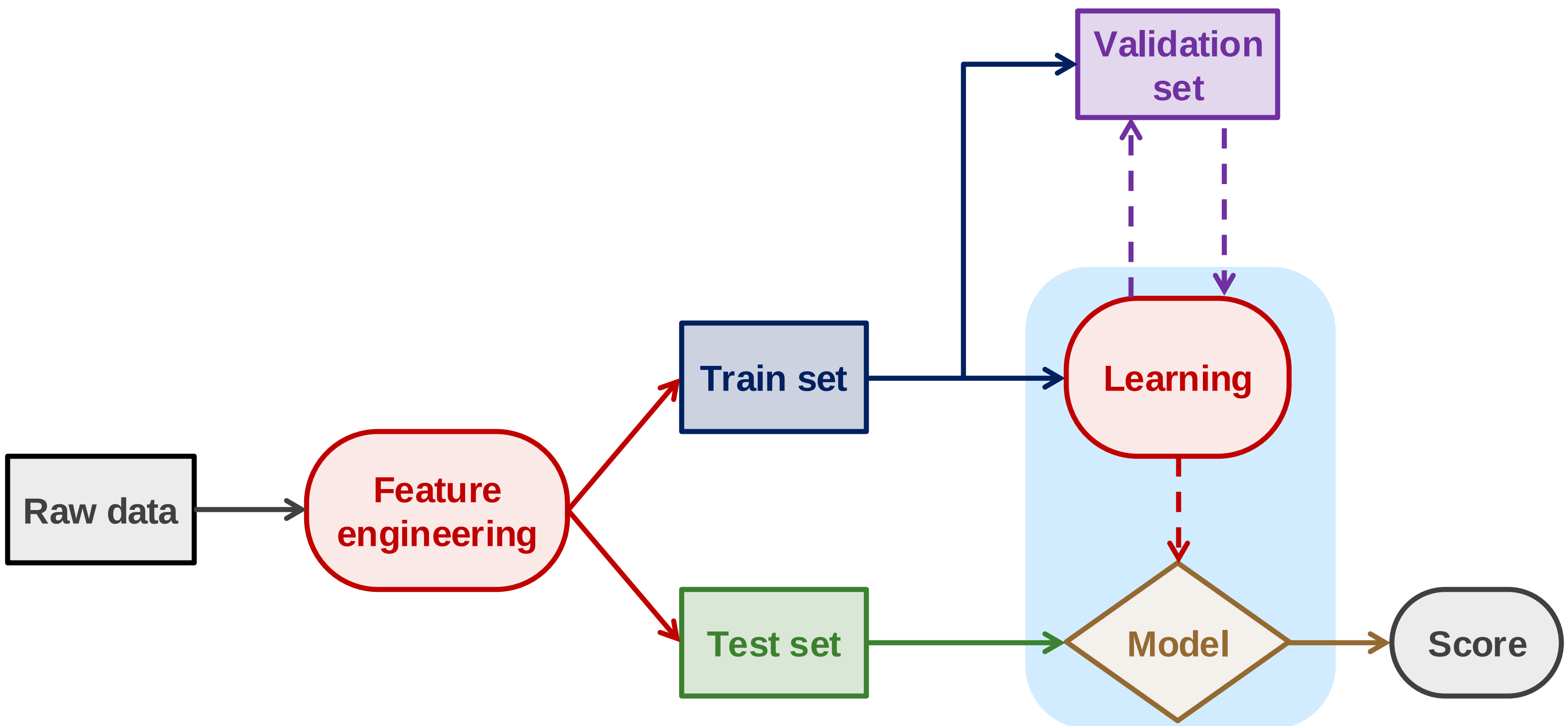
Stochastic gradient descent

Normalized gradient descent

State-of-the-art

Machine learning system

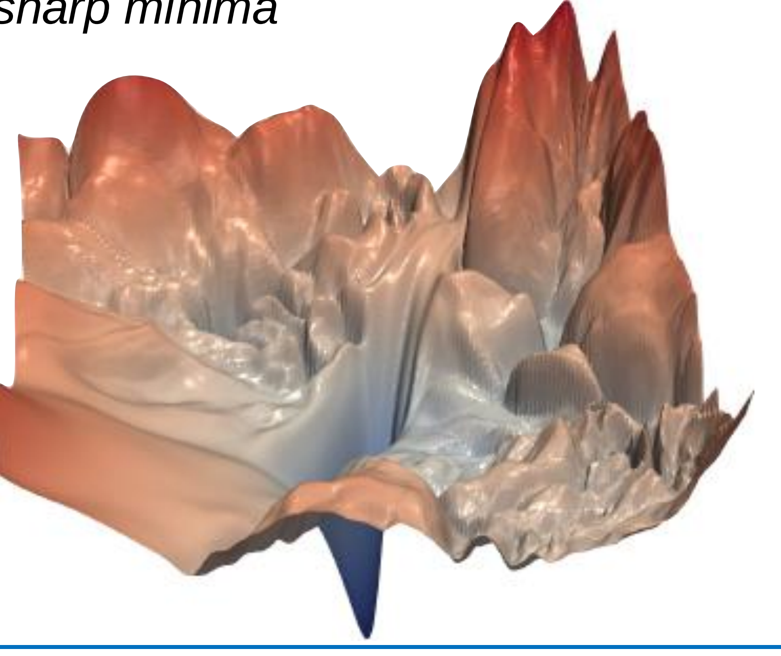
- **Training pipeline**
  - How to make it work in practice?



Saddle points and plateaus (1/3)

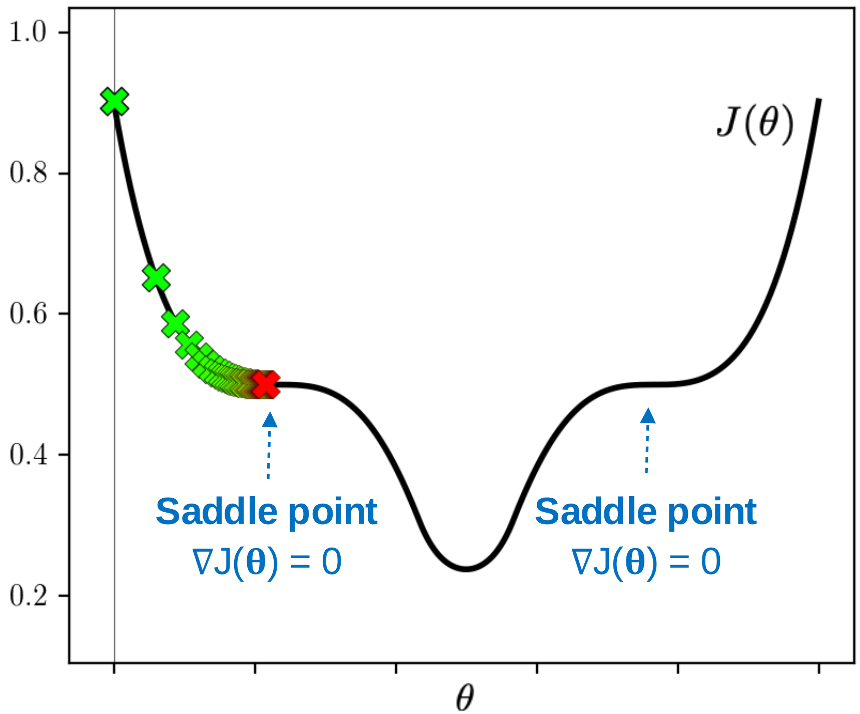
- Neural network cost function is **non-convex**
  - ***Local minima*** *dominate in shallow networks*
  - ***Saddle points*** *dominate in deep networks*
  - *Most local minima are* ***close to the bottom*** *(i.e., the global minimum)*
  - ***Flat minima*** *generalize better than sharp minima*

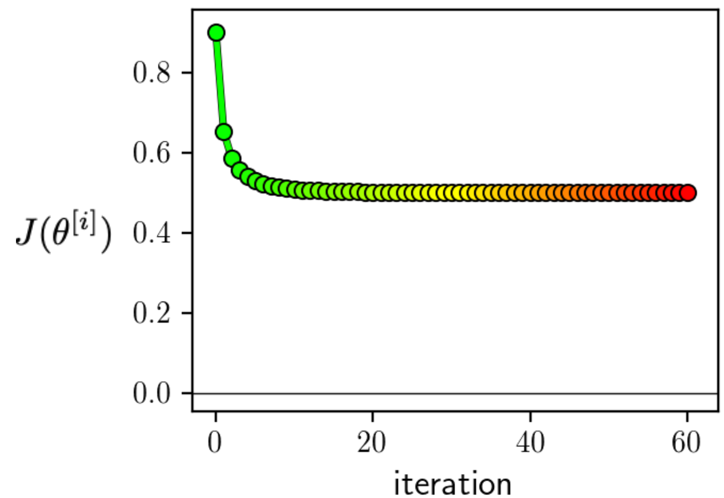
Pictorial representation of a neural network cost function



Saddle points and plateaus (2/3)

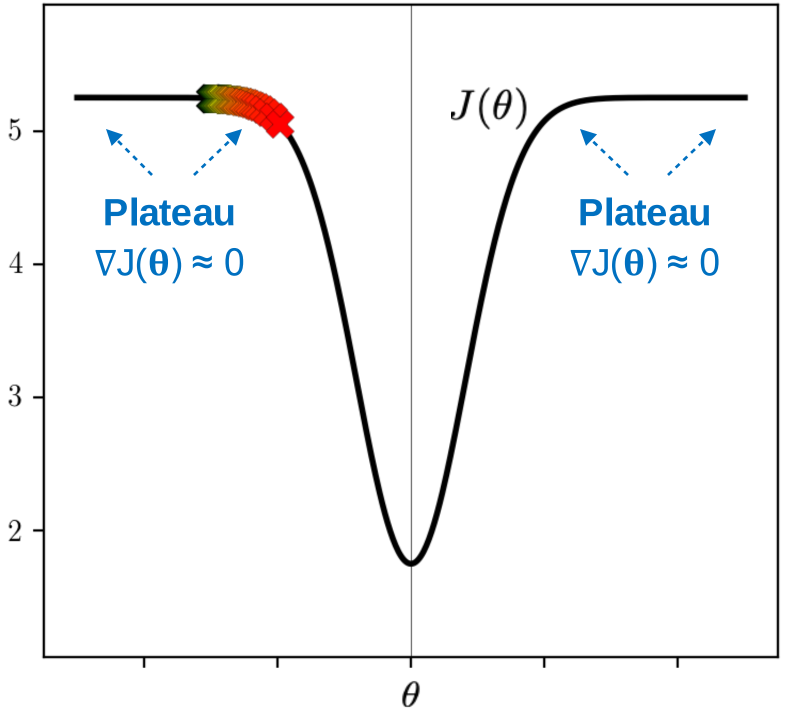
- Gradient descent **gets stuck** in saddle points

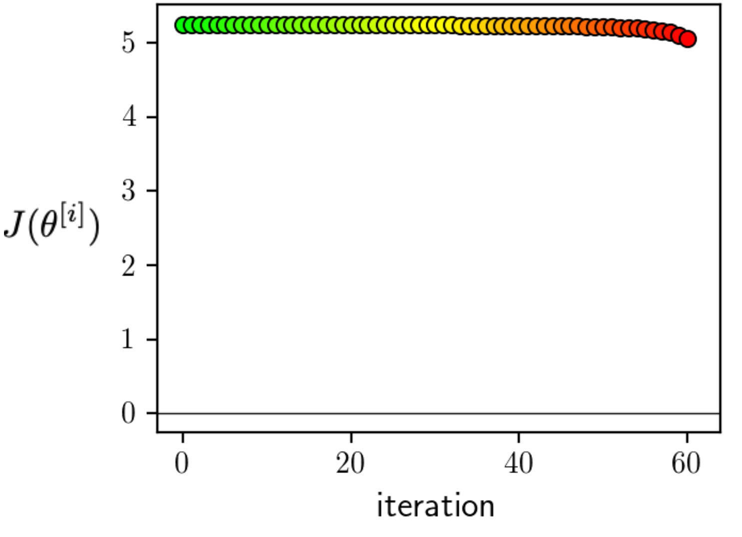




Saddle points and plateaus (3/3)

- Gradient descent **slows down** on plateaus



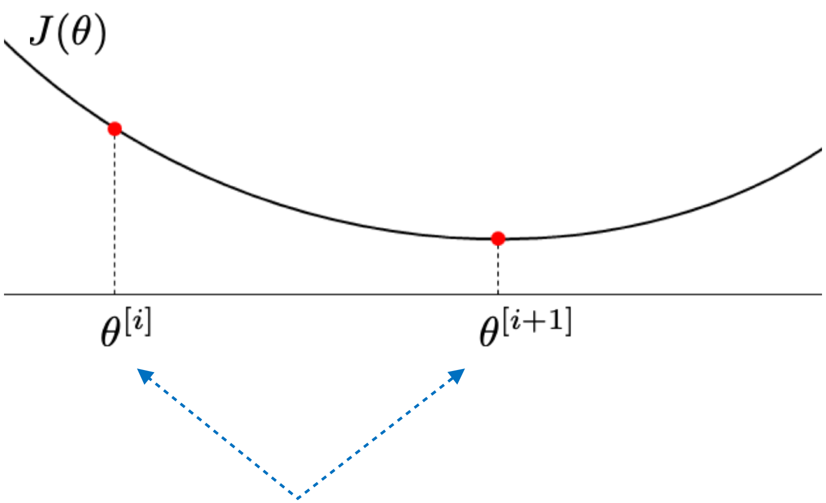


Normalized gradient descent (1/6)

- **Normalized gradient descent** uses unit-length directions
  - *The length travelled at each update is* ***constant***

Step-size

$$\theta^{[i+1]} = \theta^{[i]} - \alpha_i \frac{\nabla J\left(\theta^{[i]}\right)}{\left\|\nabla J\left(\theta^{[i]}\right)\right\|}$$



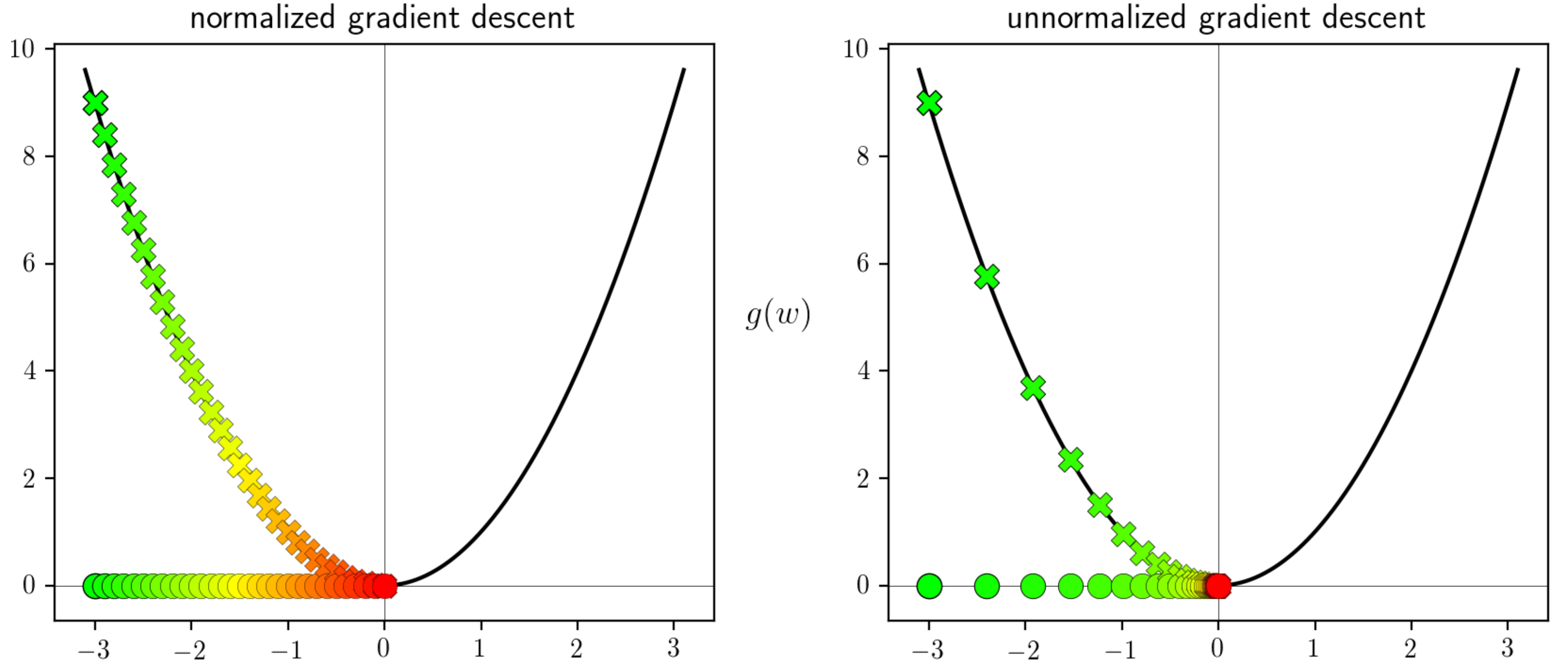
$$\left\|\theta^{[i+1]} - \theta^{[i]}\right\|^2 = \alpha_i$$

The distance travelled at each step is exactly equal to the step-size.

- **Pros.** The descent is only attracted by minima (local or global), not by saddle points.
- **Cons.** To get infinitesimally close to the solution, the step-size must decay to zero.

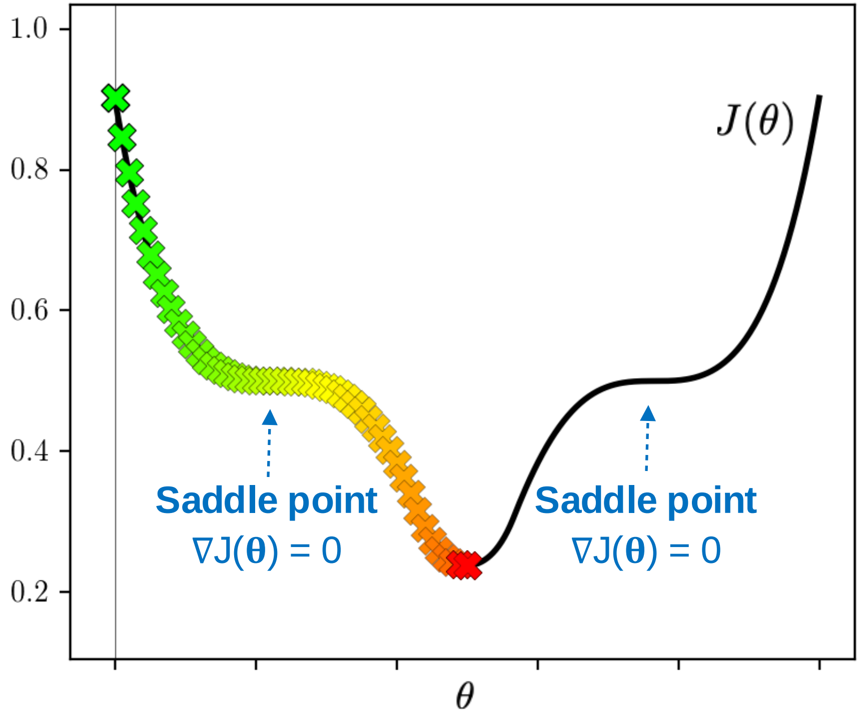
Normalized gradient descent (2/6)

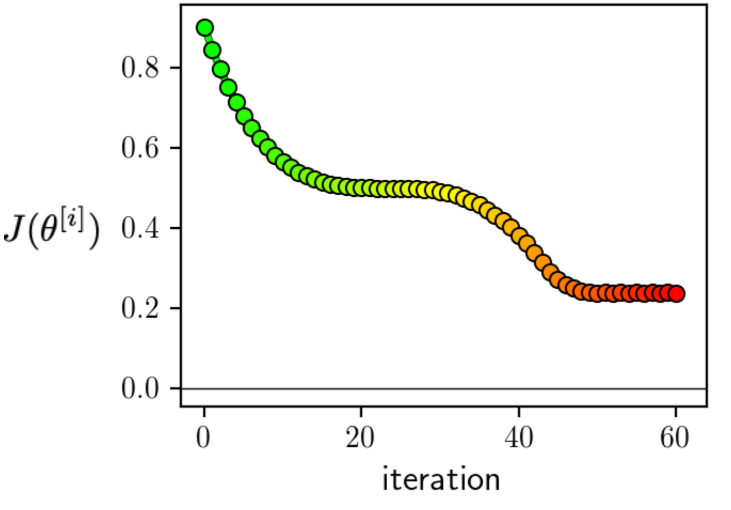
- Gradient descent → **Normalized** **vs** **Standard**
  - *Normalized GD performs fixed-length updates*
  - *Standard GD performs (decreasing) variable-length updates*



Normalized gradient descent (3/6)

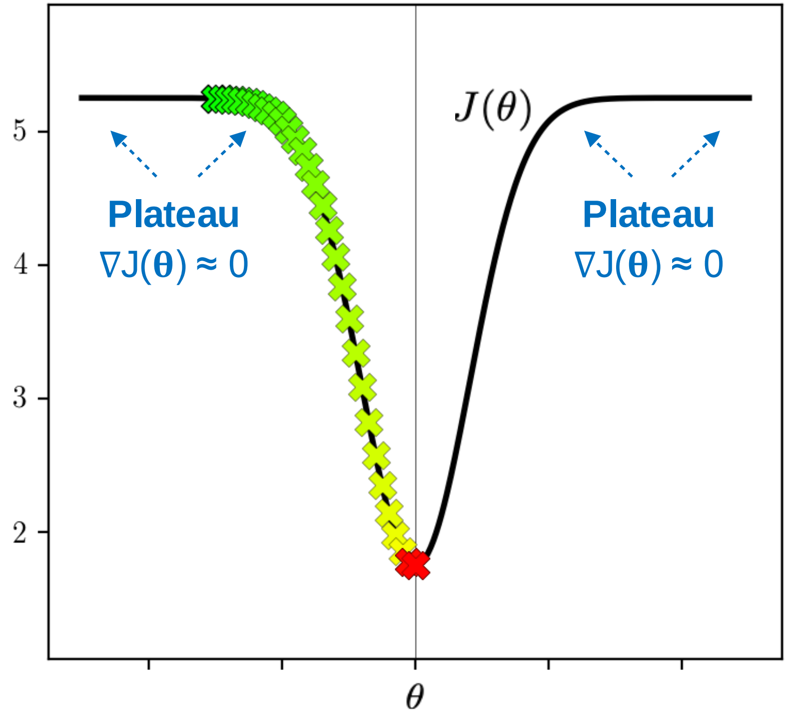
- Normalized gradient descent **overcomes** saddle points

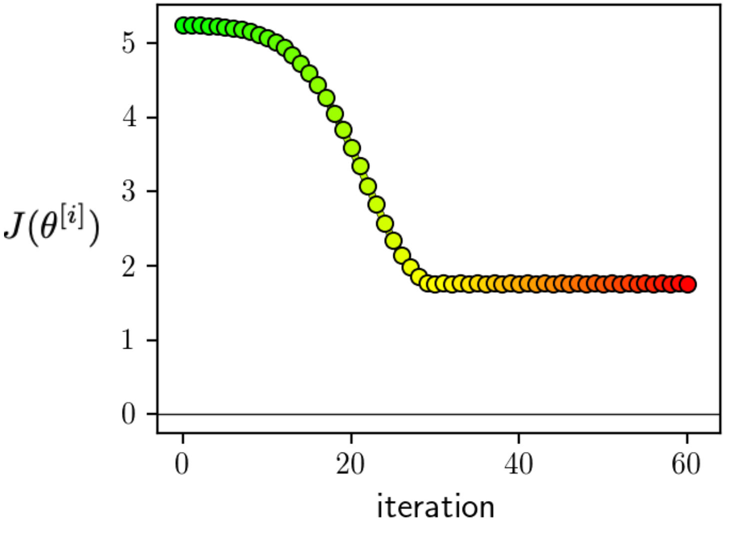




Normalized gradient descent (4/6)

- Normalized gradient descent **goes through** plateaus

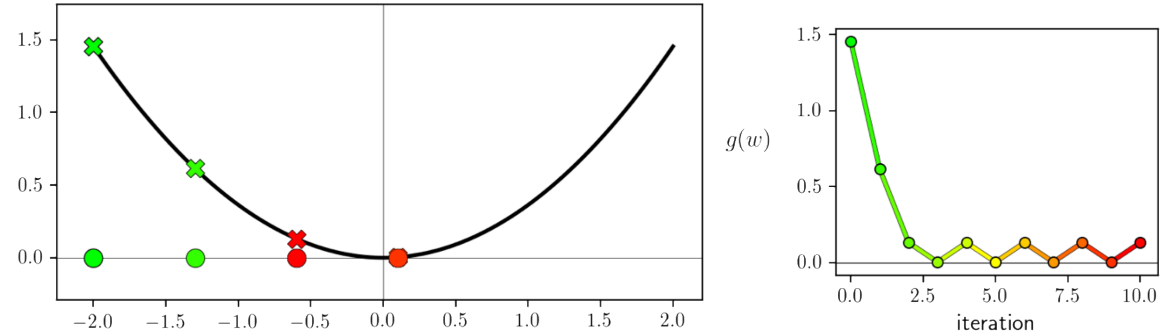




Normalized gradient descent (5/6)

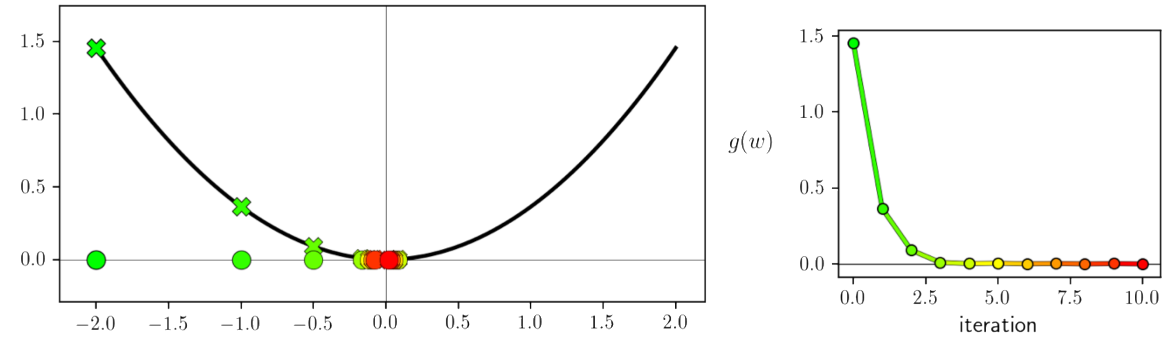
- Normalized GD **can only get so close** to a minimum
  - *The length of each step doesn't decrease while approaching a minimum*
  - ***Solution*** ➜ *Use a decreasing step-size to get arbitrary close to a minimum*

Constant step-size



Decreasing step-size

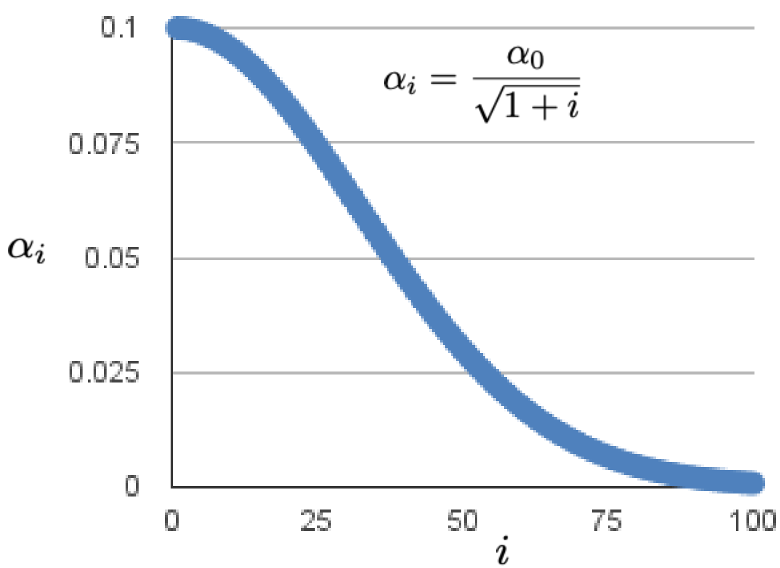
$\alpha_i = \alpha_0 / (i+1)^{0.5}$



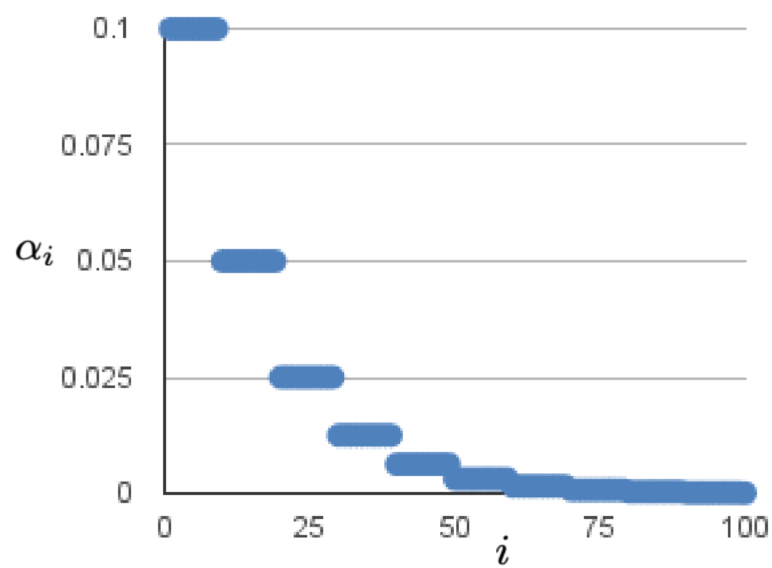
Normalized gradient descent (6/6)

- Decreasing the step-size over time
  - The initial step-size can be larger

Continuous decay

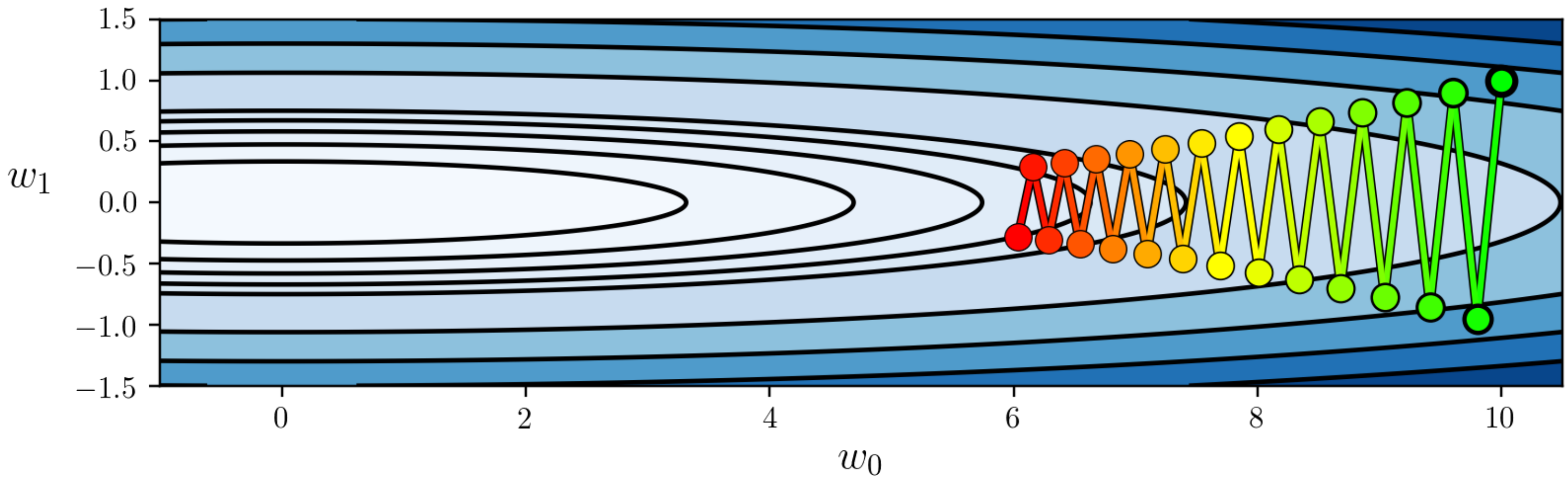


Step decay



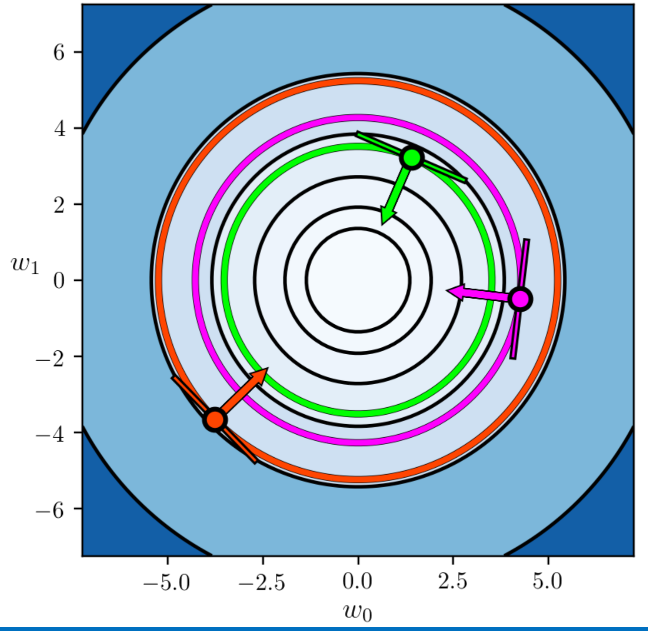
Momentum (1/4)

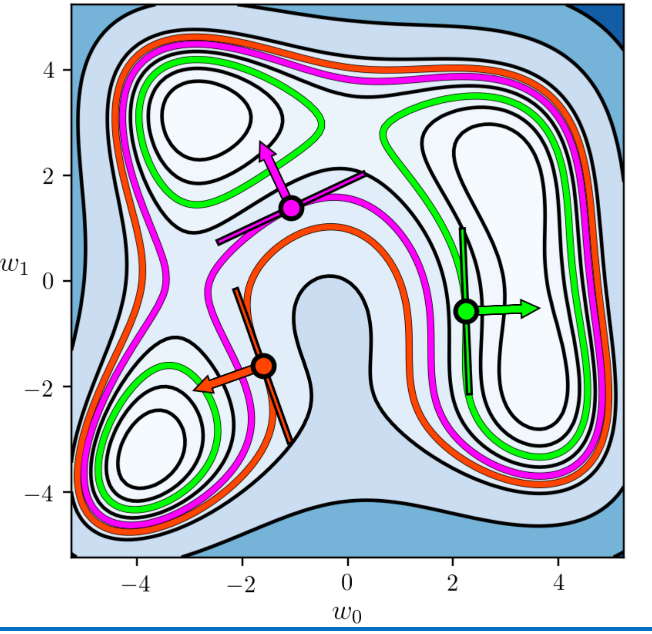
- Another issue is the “zigzagging” effect
  - Oscillations along the “steep” direction
  - Very slow progress along the “shallow” dimension



Momentum (2/4)

- Zigzagging arises when the loss function is **elliptical**
  - This is due to the very definition of gradient
  - Gradient always points perpendicular to the function contours

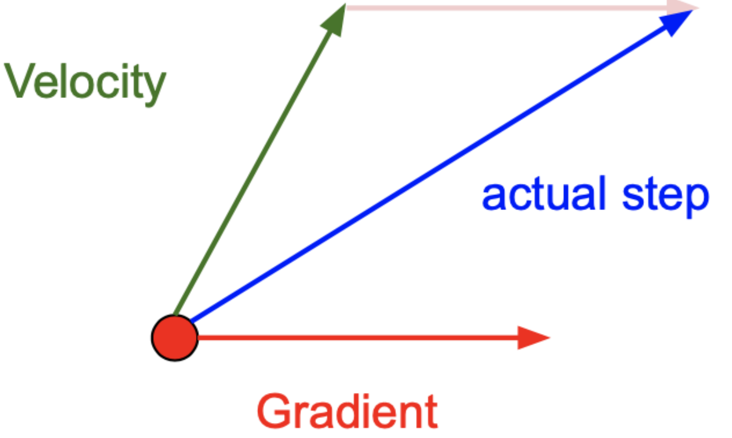




Momentum (3/4)

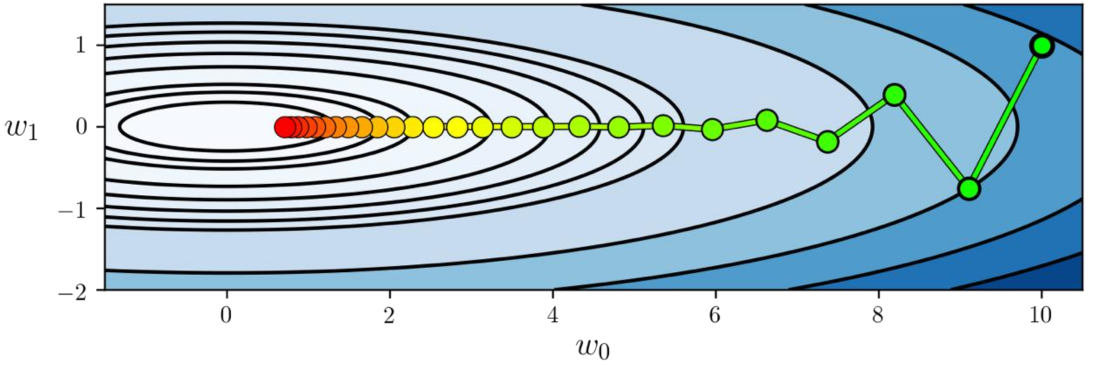
- **Solution →** Add a "momentum" term
  - Build up velocity as a running mean of gradients
  - Combine gradient with velocity to update parameters

$$\begin{cases} \mathrm{v}^{[i+1]} = \beta \mathrm{v}^{[i]} + \nabla J(\theta^{[i]}) \\ \theta^{[i+1]} = \theta^{[i]} - \alpha\, \mathrm{v}^{[i+1]} \end{cases}$$

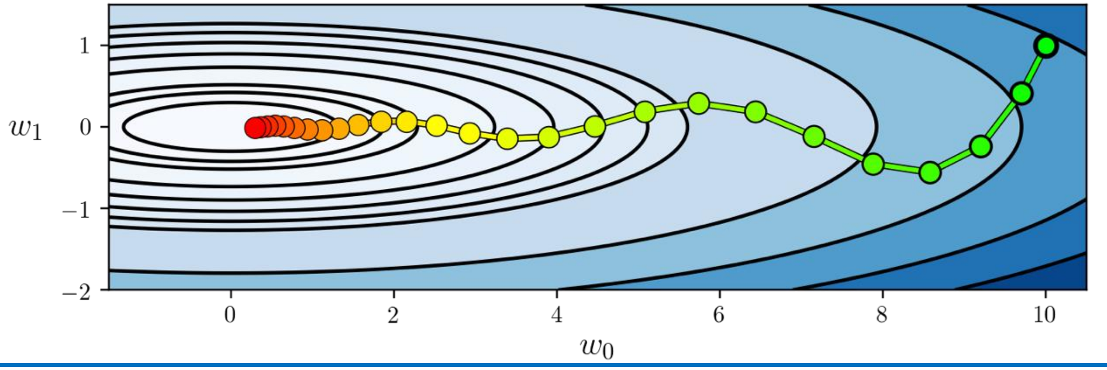


Momentum (4/4)

- The momentum term dampens the oscillations
  - It makes the trajectory reluctant to change direction



$\beta = 0.2$



$\beta = 0.7$

State-of-the-art

- **ADAM** → Modern algorithm for neural network training
  - *Gradient descent + Normalization + Momentum*

$$
\begin{cases}
g^{[i]} = \nabla J^{[i]}\left(\theta^{[i]}\right) & \text{Gradient (possibly on mini-batch)} \\
m^{[i+1]} = \beta_1 m^{[i]} + (1-\beta_1) g^{[i]} & \text{Moving average of gradients} \\
v^{[i+1]} = \beta_2 v^{[i]} + (1-\beta_2) \left(g^{[i]}\right)^2 & \text{Moving average of square gradients} \\
\theta^{[i+1]} = \theta^{[i]} - \alpha_i \dfrac{m^{[i+1]}}{\sqrt{v^{[i+1]}} + \epsilon} & \text{Normalization (element-wise)}
\end{cases}
$$

Summary so far…

- **ADAM →** Accelerated gradient descent

$$\theta^{[i+1]} = \theta^{[i]} - \alpha_i \nabla J^{[i]}(\theta^{[i]})$$

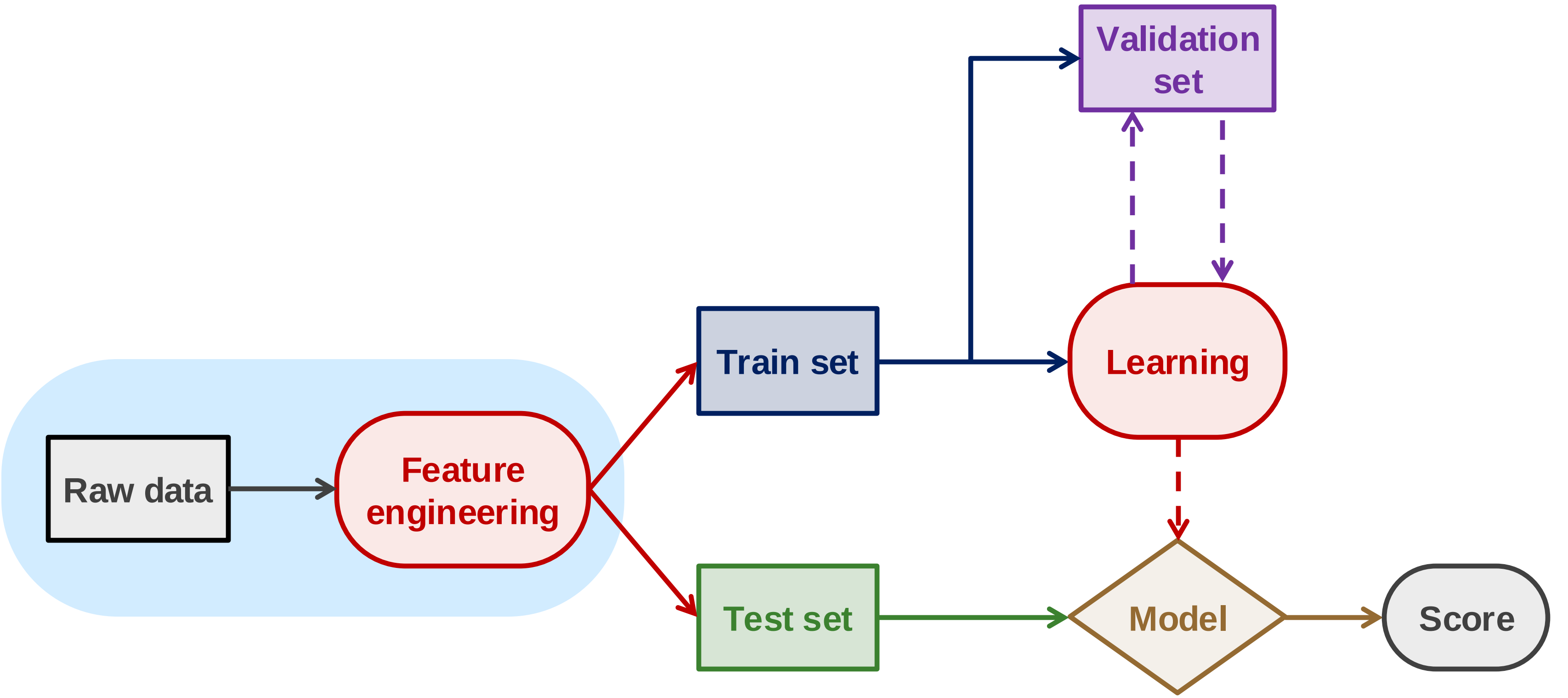
Adaptive step-size

- New hyper-parameters unlocked !!!
  - *Learning rate*
  - *Mini-batch size*
  - *Optimization (SGD, ADAM, …)*
  - *Decaying schedule for step-size*

Feature engineering

Machine learning system

- **Training pipeline**
  - How to make it work in practice?



Feature engineering (1/2)

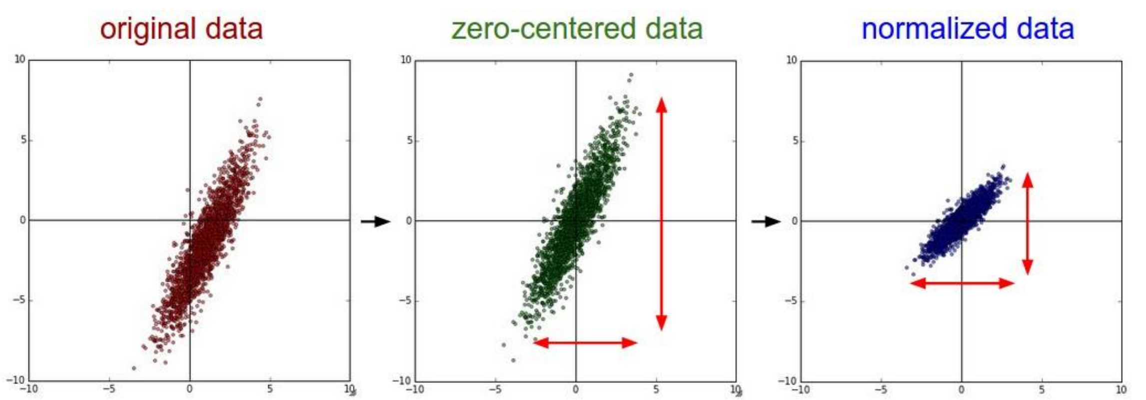
- What is **feature engineering**?
  - The process of extracting informative features from raw data
  - ***(Feature*** *= Individual measurable property of a phenomenon)*

- Examples
  - Crafting new variables from raw data
  - Numerical transformations
  - Normalization
  - Encoding
  - Cleaning & Imputation

Feature engineering (2/2)

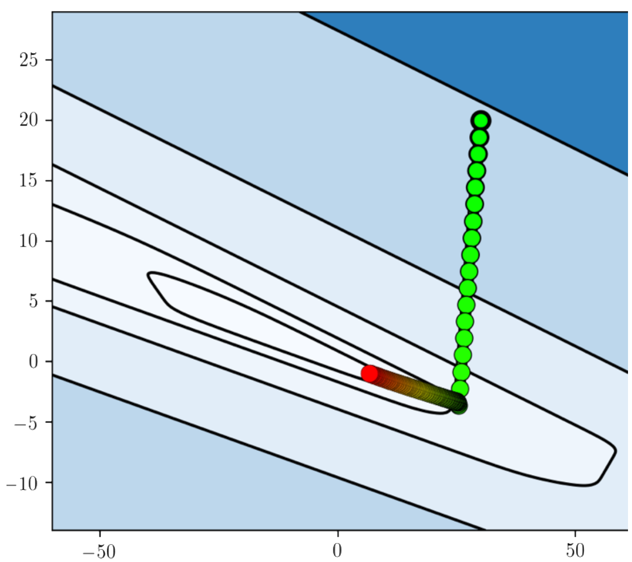
- Neural networks are capable of **feature learning**
  - Hidden layers learn how to extract informative features
  - There is no need to manually craft new variables

- Feature learning works well on numerical data
  - Remember to normalize numerical variables!



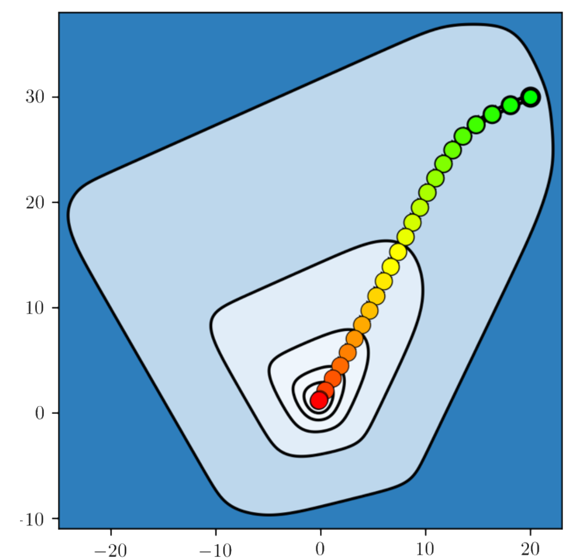
Numerical variables

- Normalization helps **training go faster**
  - *The cost function is “strongly” elliptical*
  - *Normalization makes the cost function “more circular”*
  - *This transformation speeds up the optimization process*



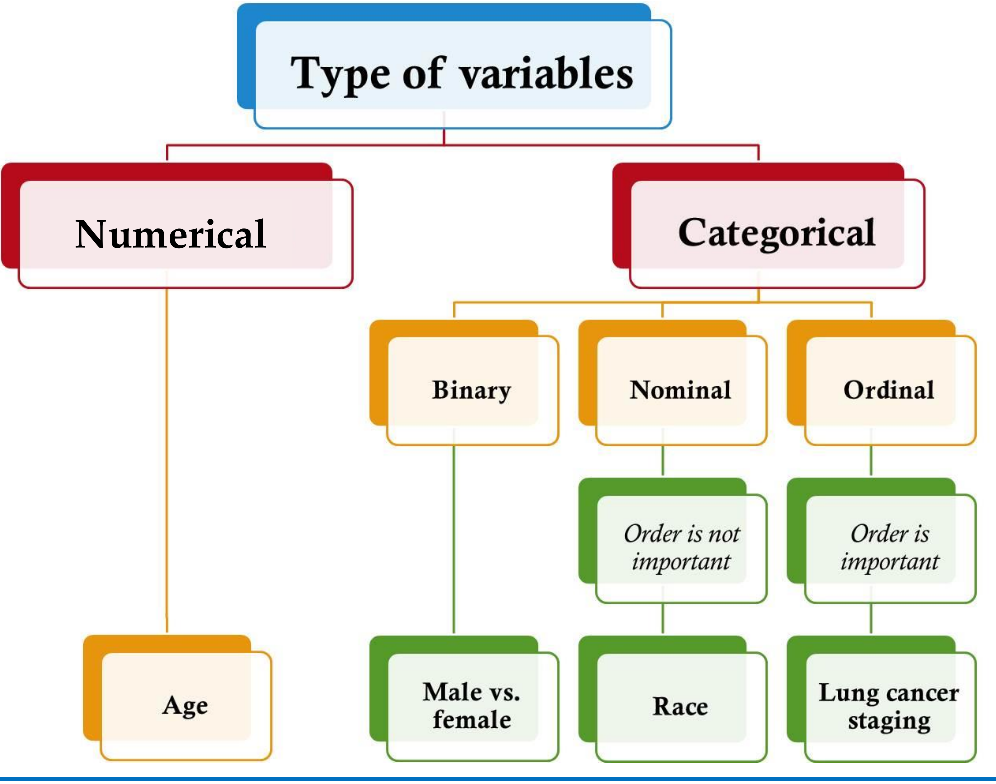
Normalization

The cost function becomes “more circular”, and thus gradient descent can reach the minimum in less steps.



Categorical variables

- Neural networks struggle with categorical data
  - Variables that can take on a fixed number of possible values



Dummy coding

- A categorical variable is replaced by binary variables
  - Use N-1 binary values to represent N categories
  - A group is encoded with the vector (0, 0, …, 0)
  - The other groups are one-hot encoded
  - **When to use?** One group is more important than the others

Nationality	C1	C2	C3
French	0	0	0
Italian	1	0	0
German	0	1	0
Other	0	0	1

← Most important or biggest group

Effects coding

- A categorical variable is replaced by binary variables
  - Use N-1 binary values to represent N categories
  - A group is encoded with the vector (-1, -1, …, -1)
  - The other groups are one-hot encoded
  - **When to use?** One group is less important than the others

Nationality	C1	C2	C3
French	1	0	0
Italian	0	1	0
German	0	0	1
Other	-1	-1	-1

← Least important or smallest group

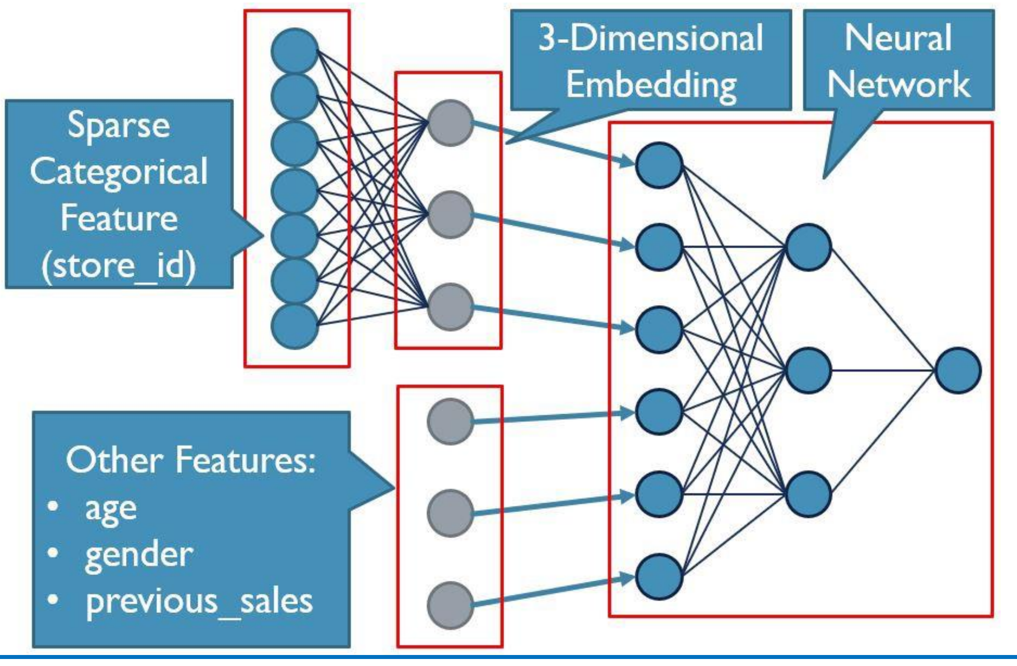
Contrast coding

- A categorical variable is replaced by numerical variables
  - Use N-1 variables to represent N categories
  - The coefficients per each variable must sum to zero
  - The difference between the sum of the positive values and the sum of the negative values per each variable should equal 1
  - The vector of coefficients per each variable must be orthogonal

Nationality	C1	C2	C3
French	0.25	0.33	0.5
Italian	0.25	0.33	-0.5
German	0.25	-0.66	0
Other	-0.75	0	0

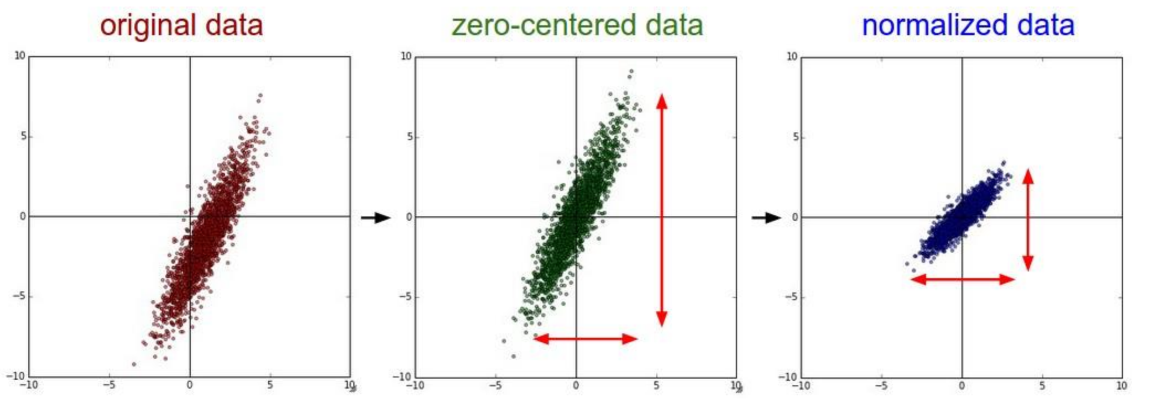
Embedding

- A special “embedding” layer is added to the network
  - This layer maps each category to a numerical vector (of arbitrary size) that is learned by the network during training



Summary so far…

- **Data preprocessing is important**
  - *Clean the dataset*
  - *Normalize the numerical variables*
  - *Replace the categorical variables*



Overfitting

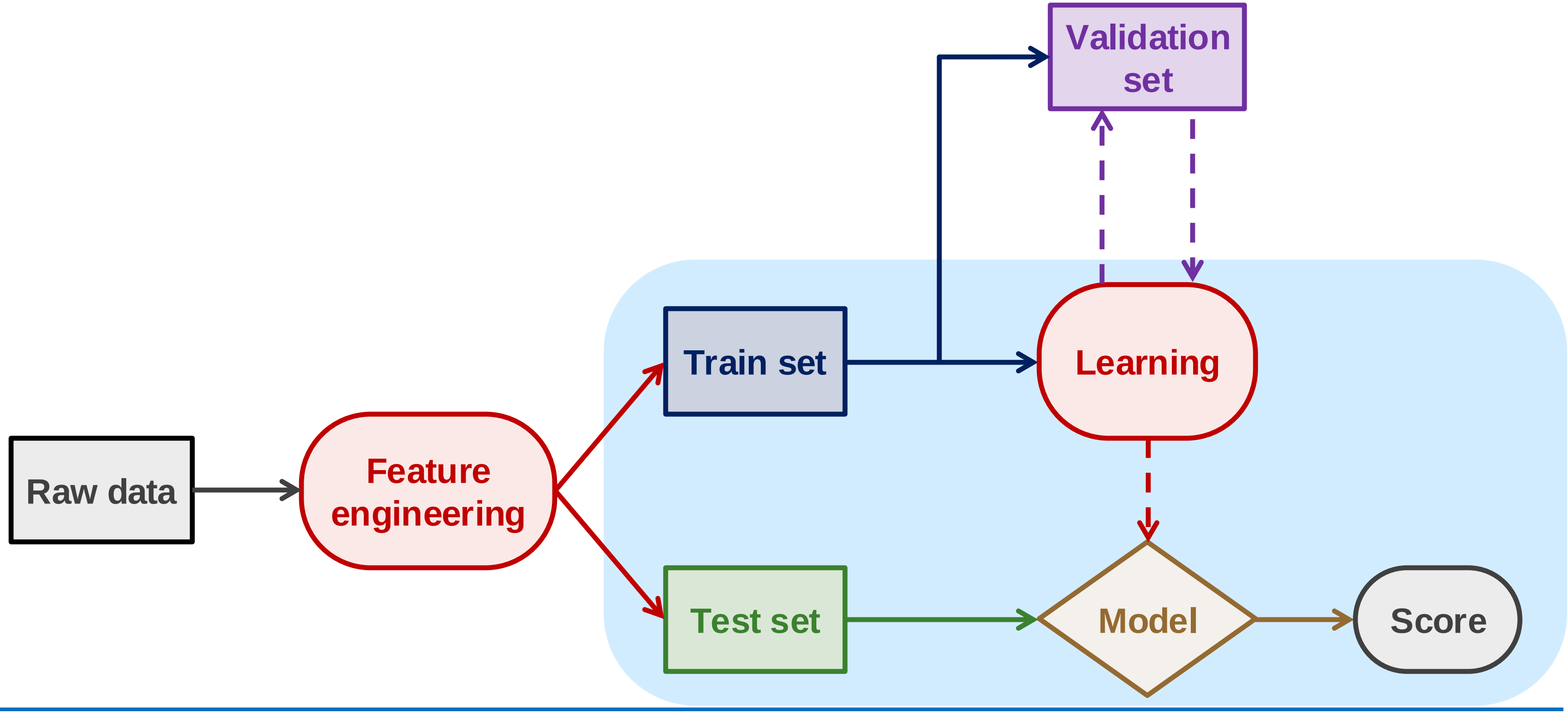
What is it?

How to detect it?

How to fight it?

Machine learning system

- **Training pipeline**
  - How to make it work in practice?



Over-fitting (1/3)

- Training allows the network to learn its **parameters**
  - $\boldsymbol{\theta} = \mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)} \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}$

- But only after the **hyper-parameters** are fixed…
  - *Number of layers in the neural network*
  - *Number of units in each layer*
  - *Activation function for each layer*
  - *… (and many others)*

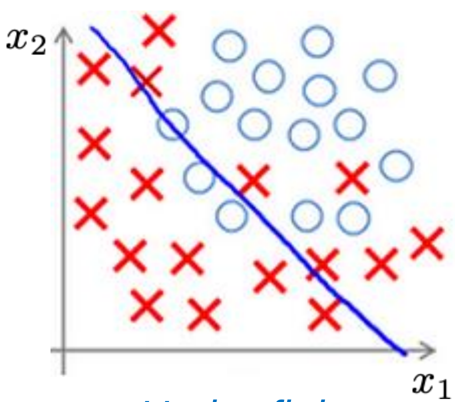
Network architecture

- Hyper-parameters affect the network predictions

Over-fitting (2/3)

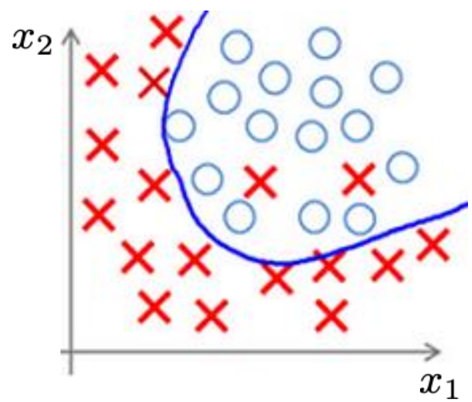
- What is the impact of hyper-parameters on learning ?
  - ***Under-fitting*** → *The predictions are* ***<u>too far</u>*** *from the expected outputs*
  - ***Over-fitting*** → *The predictions are* ***<u>too close</u>*** *to the expected outputs*

Small network



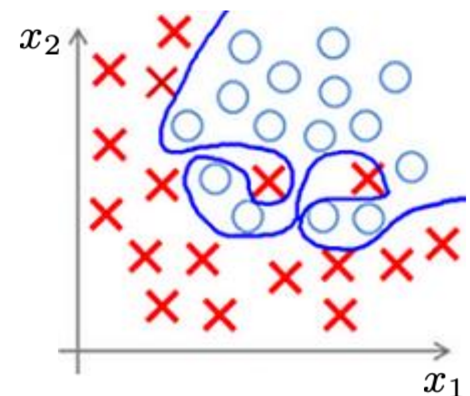
Under-fitting
(high bias)

Medium network



"Just right"

Big network

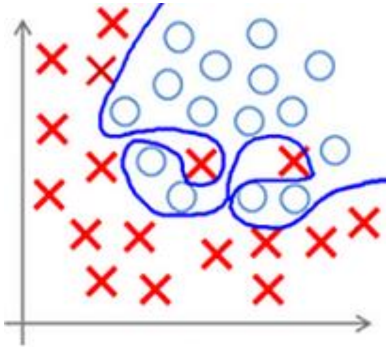


Over-fitting
(high variance)

Over-fitting (3/3)

- Learning aims to achieve a **good generalization**
  - *The model must perform well on never-before-seen data*

- Over-fitting is an obstacle to generalization
  - ***Learning*** ➜ *The model fits very well the training data…*
  - ***Prediction*** ➜ *… but it is unable to generalize to new data.*



Nothing useful is being learned here

The model is distracted by some outliers, instead of following the general trend of data.

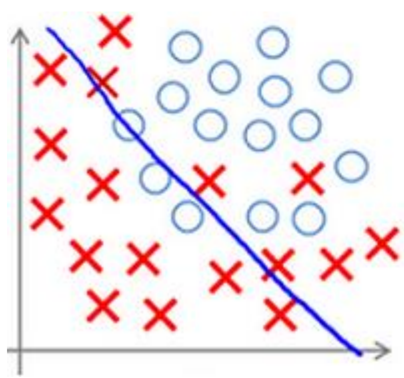
How to detect over-fitting (1/4)

- It is not advised to evaluate the model on the training data

$$J_{\text{train}}\left(\widehat{\theta}\right) = \frac{1}{N}\sum_{n=1}^{N} C\left(f_{\widehat{\theta}}(\mathrm{x}^{(n)}), y^{(n)}\right)$$

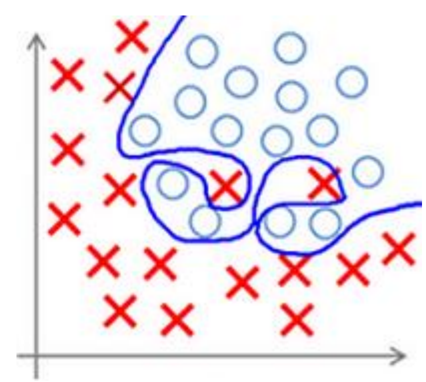
  - ***Warning*** *→ This estimate is biased toward* ***over-fitting*** *!!!*

Under-fitting



high ← J_{Train} → *low*

Over-fitting



How to detect over-fitting (2/4)

- It is better to evaluate the model on **fresh data**
  - ***Train set*** → *Used for training the model*
  - ***Test set*** → *Used for testing the model*

Dataset

Input	Output
2104	400
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
1427	199

Train set *70%*

2104	400
1600	330
2400	369
1416	232
3000	540

Score

$$J_{\text{train}}(\widehat{\theta}) = \sum_{n=1}^{N_1} C\Big(f_{\widehat{\theta}}(\mathrm{x}^{(n)}), y^{(n)}\Big)$$

Test set *30%*

1985	300
1534	315
1427	199

Score

$$J_{\text{test}}(\widehat{\theta}) = \sum_{n=1}^{N_3} C\Big(f_{\widehat{\theta}}(\mathrm{x}_{\mathrm{t}}^{(n)}), y_{\mathrm{t}}^{(n)}\Big)$$

How to detect over-fitting (3/4)

- Over-fitting can be detected on the test set
  - ***Regression*** → *Model evaluated on mean square error*
  - ***Classification*** → *Model evaluated on classification error*

	Low bias	**High bias** (under-fitting)
Low variance	Err_{Train} = 0.5 % Err_{Test} = 1.0 %	Err_{Train} = 17.0 % Err_{Test} = 18.3 %
High Variance (over-fitting)	Err_{Train} = 1.0 % Err_{Test} = 19.3 %	Err_{Train} = 15.0 % Err_{Test} = 30.0 %

Low variance → *Small gap in performance*

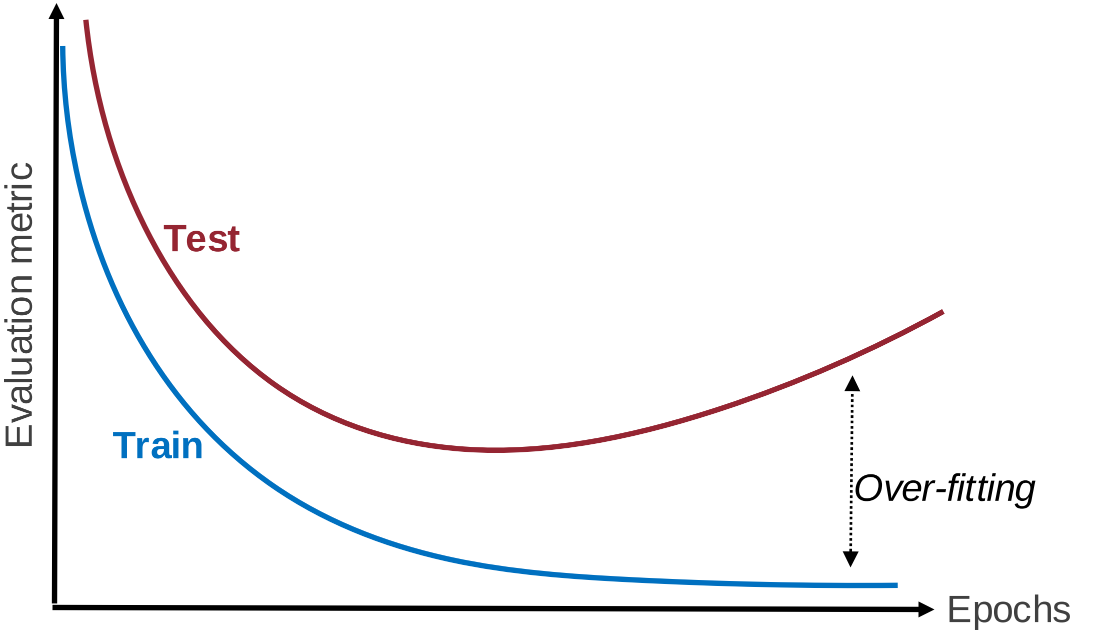
High Variance → *Big gap in performance*

Low bias → *Small error on training*

High bias → *Big error on training*

How to detect over-fitting (4/4)

- Over-fitting can be also monitored during training
  - ***Train cost*** **→** *How well the model fits the training data*
  - ***Test cost*** **→** *How well the model performs on new unseen data*



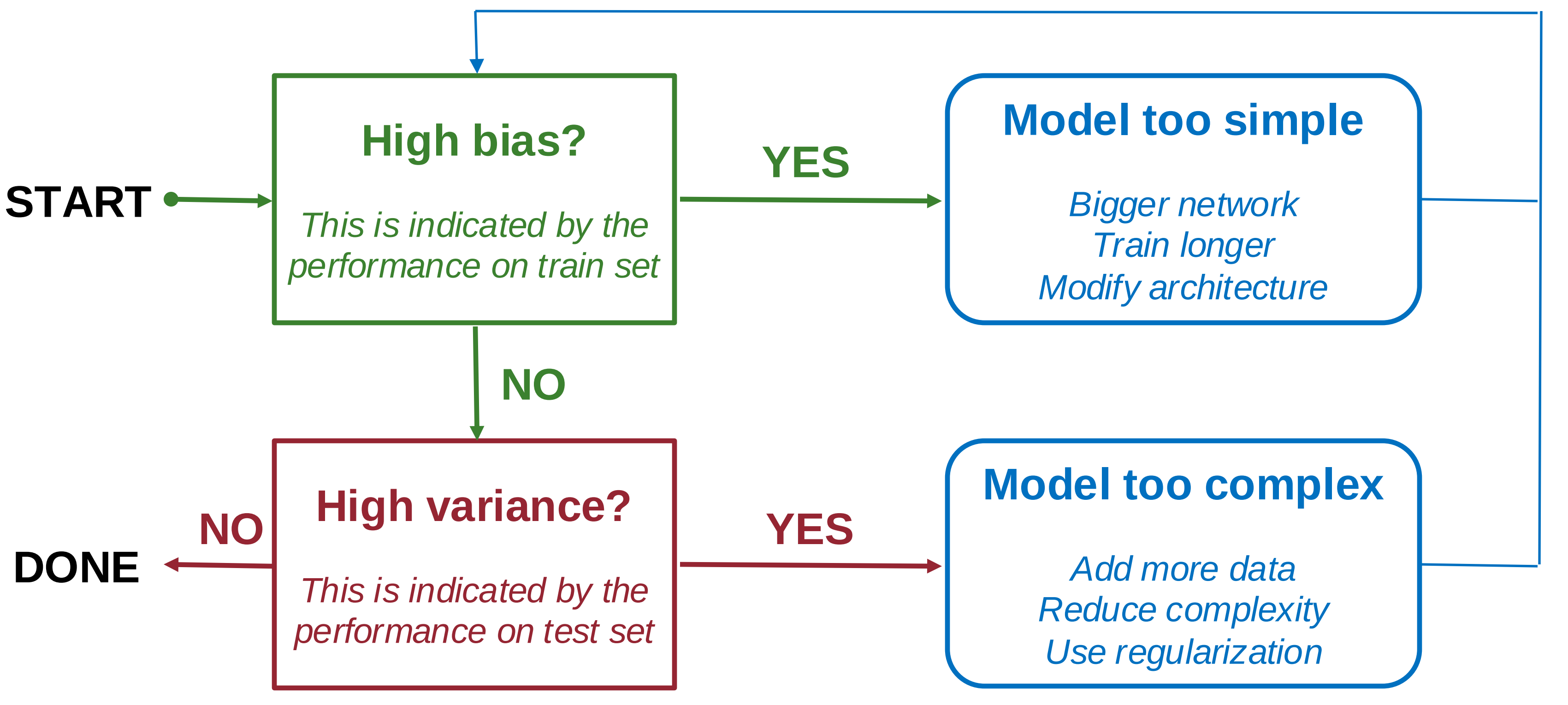
How to fight over-fitting (1/3)

- The underlying causes of **under-fitting**
  - ***Simple model*** ➜ *Prediction close to linear, few parameters, …*
  - ***Low dimension*** ➜ *Features are not enough to make a prediction*

- The underlying causes of **over-fitting**
  - ***Complex model*** ➜ *Prediction highly nonlinear, a lot of parameters, …*
  - ***High dimension*** ➜ *There are too many features*
  - ***Lack of data*** ➜ *The train set is too small w.r.t. the parameters to learn*

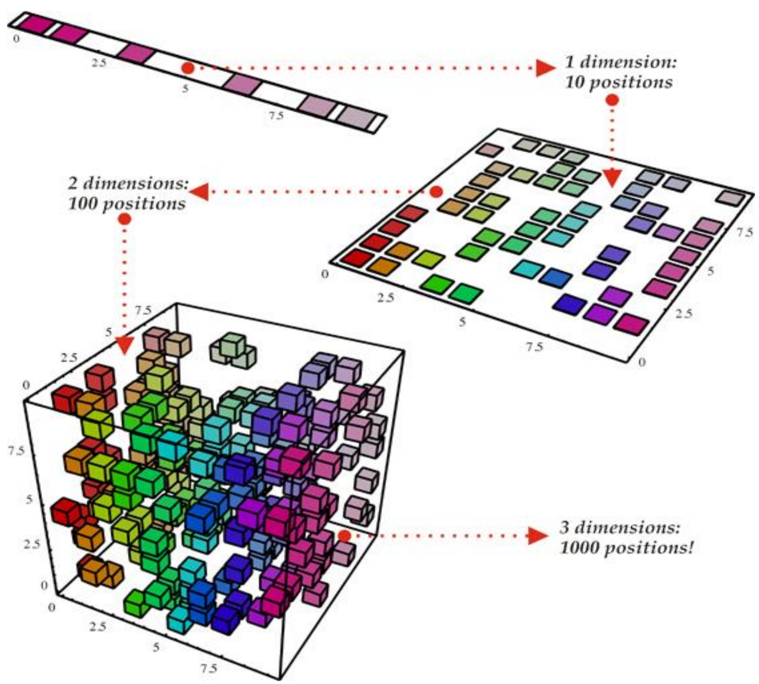
How to fight over-fitting (2/3)

- Bias and variance reduction can be tackled separately



How to fight over-fitting (3/3)

- Can we avoid over-fitting only with more training data ?
  - *The amount of data* ***grows exponentially*** *with the dimensionality*
  - *At some point, we can't add enough data to prevent over-fitting*



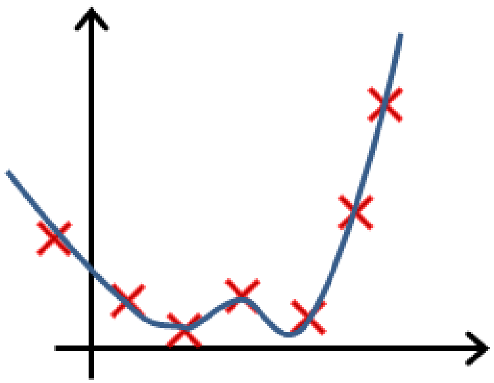
Exponential growth

- *1 feat.* → *10 samples*
- *2 feat.* → 10^2 *samples*
- *3 feat.* → 10^3 *samples*
- *…*

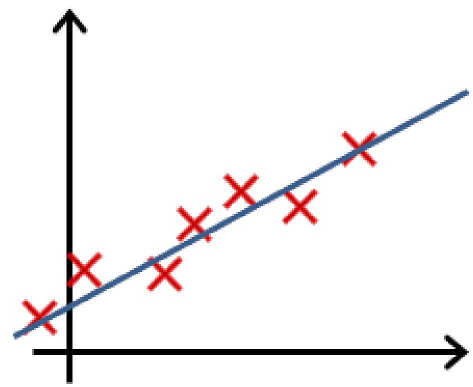
Quiz (1/3)

- In which figure the model has overfit or underfit the training set?

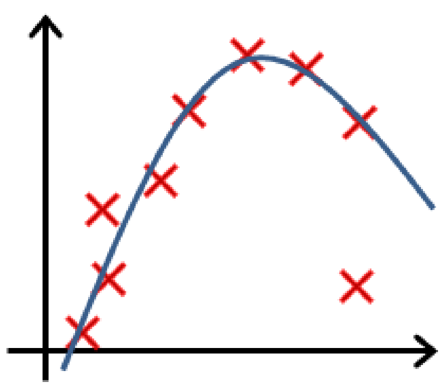
1.



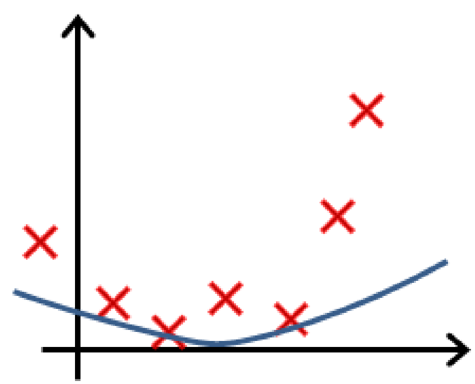
2.



3.



4.



Quiz (2/3)

- What does it mean that a model $\mathbf{f}_\theta$ has **overfit** the data ?

1. *It makes accurate predictions for examples in the training set, and generalizes well to make accurate predictions on new examples.*

2. *It doesn't makes accurate predictions for examples in the training set, but it generalizes well to make accurate predictions on new examples.*

3. *It makes accurate predictions for examples in the training set, but it doesn't generalizes well to make accurate predictions on new examples*

4. *It doesn't make accurate predictions for examples in the training set, and doesn't generalizes well to make accurate predictions on new examples.*

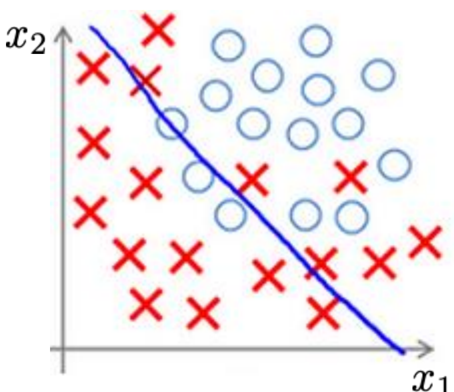
Quiz (3/3)

- Suppose your neural network obtains a train set error of 0.5%, and a test set error of 7%.
- What should you try to improve the performance?

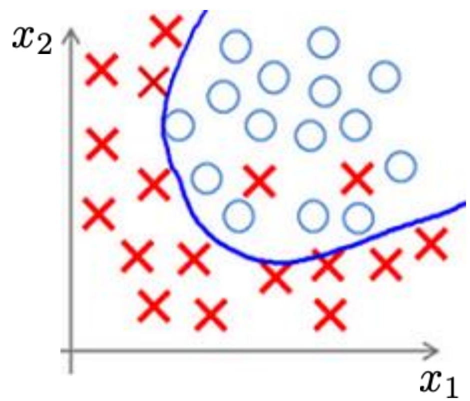
1) *Increase the number of units in each hidden layer*
2) *Add regularization*
3) *Use a deeper neural network*
4) *Get more test data*
5) *Get more training data*

Summary so far…

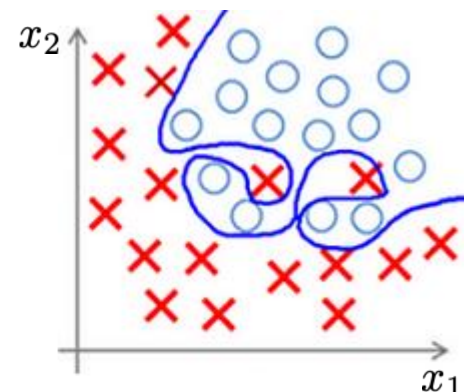
- **Bias-variance tradeoff**
  - *Over-fitting is the obstacle to generalization*
  - *Use a test set to detect over-fitting (or under-fitting)*
  - *Recipes to reduce bias and variance*



Under-fitting
(high bias)



"Just right"



Over-fitting
(high variance)

Regularization

Norm penalization
Early stopping
Dropout
Batch normalization

Over-fitting

- How to reduce over-fitting ?
  - ***Option 1*** **→ *Add more training data***
    - This is always beneficial, but it could be expensive to get more data
  - ***Option 2*** **→ *Simplify the model***
    - Reduce the network parameters by using less units and layers
    - The risk is to increase the bias
  - ***Option 3*** **→ *Apply regularization***
    - Keep the complexity, but reduce the model's degrees of freedom
    - This diminishes somewhat the capacity to fit the training data
    - A big variance reduction is traded for a small bias increase

Norm penalization (1/3)

- **Norm penalization ➜** Small values for parameters
  - *The cost function is modified as follows:*

$$J(\theta) = \sum_{n=1}^{N} C\Big(f_\theta(\mathrm{x}^{(n)}), \mathrm{y}^{(n)}\Big) + \lambda \sum_{m=1}^{M} |\theta_m|^p$$

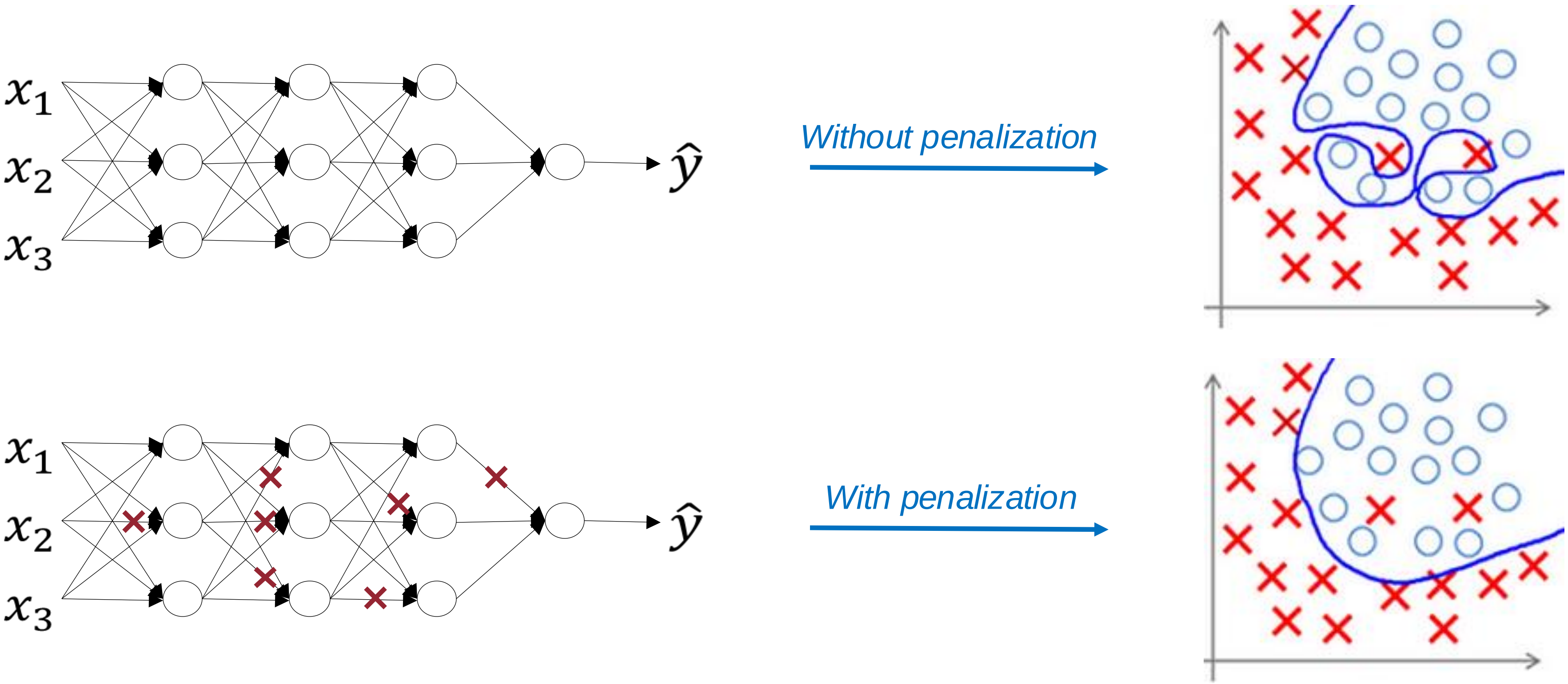
  - *Now, the cost function is minimized for smaller values of parameters*

$$J(\theta) \to 0 \qquad \Leftrightarrow \qquad \theta_1 \to 0, \ldots, \theta_M \to 0$$

  - *Small values correspond to a simpler model*
  - *A simpler model is less prone to over-fitting and more to under-fitting*

Norm penalization (2/3)

- The penalization gets rid of some **network connections**
  - *The connections to be removed are identified during training*



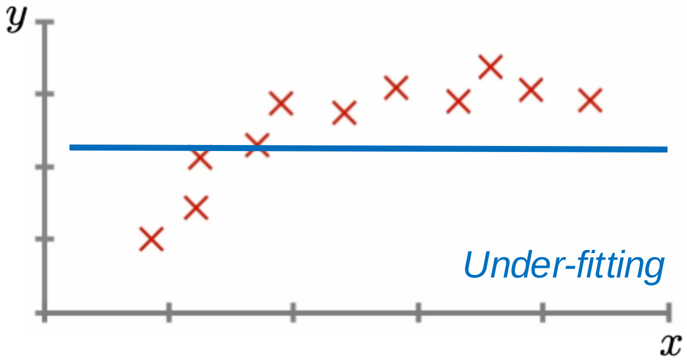
Norm penalization (3/3)

- The hyper-parameter **λ** controls the tradeoff of two goals
  - *Fitting the train set*
  - *Keeping a simple model*

- **Warning ➜** The choice of **λ** is critical
  - *If* ***λ*** *is very large, all the model parameters end up being close to zero*

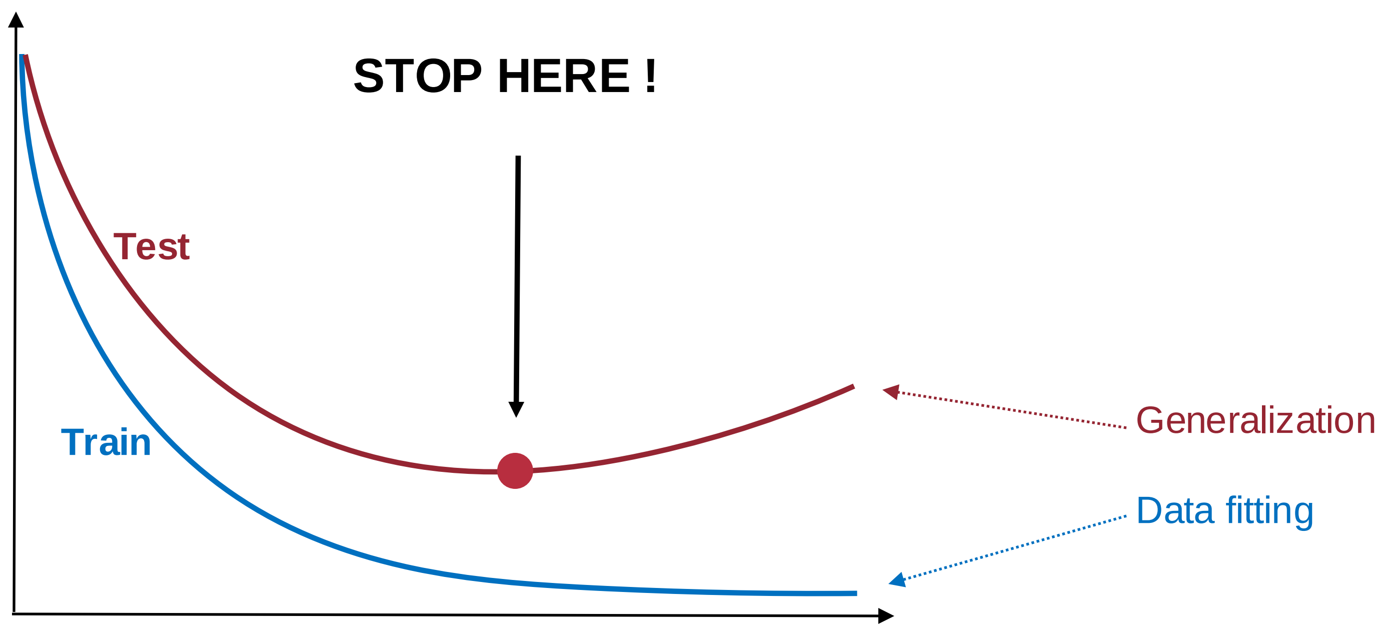
$$\lambda \to +\infty \qquad \Rightarrow \qquad \theta_1 \approx 0, \dots, \theta_M \approx 0$$

  - *In this case, the model is under-fitting, as we get rid of all the network connections*



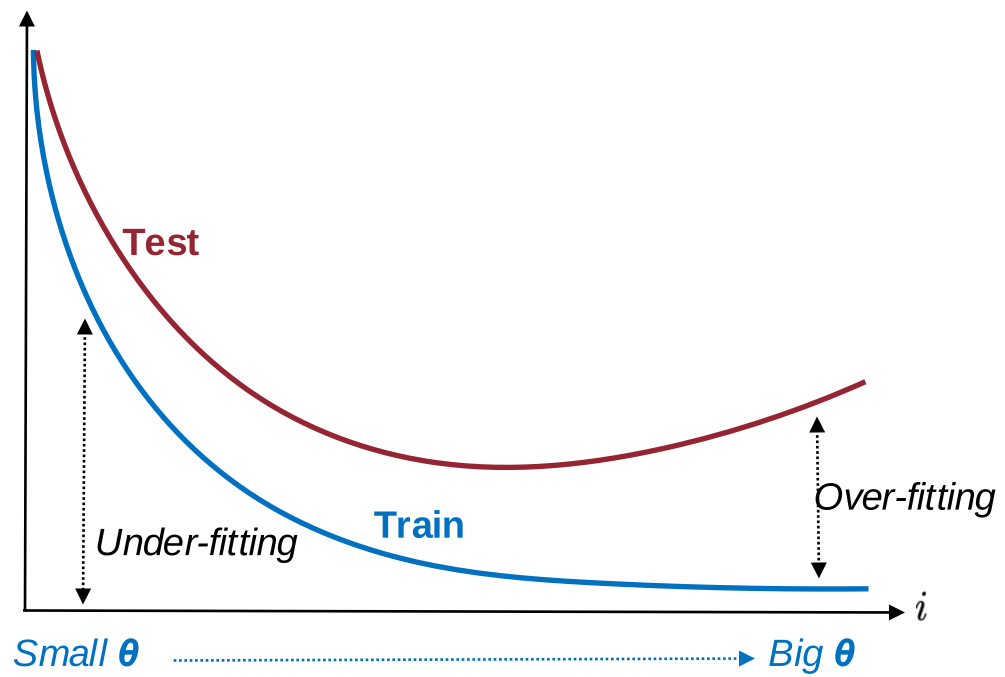
Early stopping (1/2)

- **Early stopping ➜** Halt when generalization stops improving
  - *Training is halted when the **performance on test set** begins to degrade*



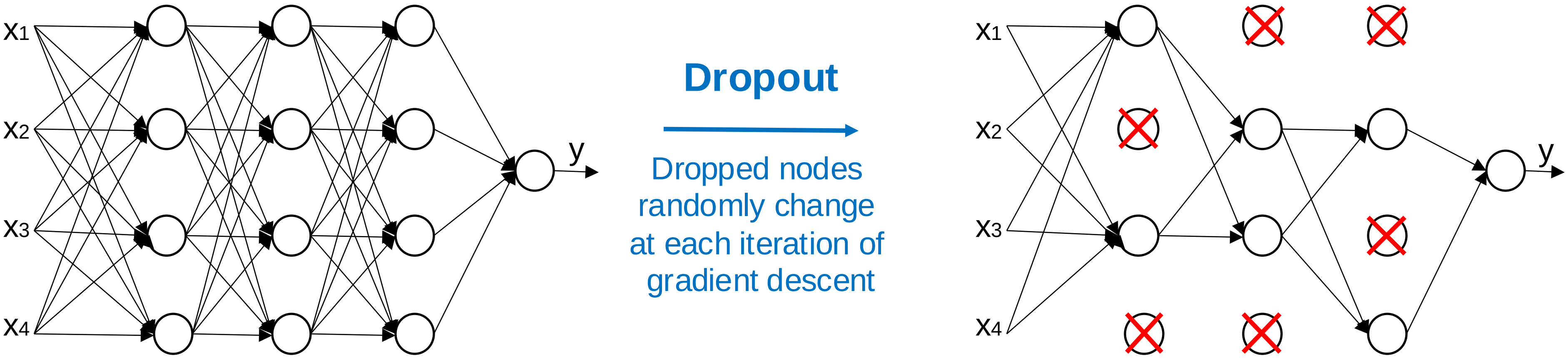
Early stopping (2/2)

- The magnitude of parameters increases during training
  - ***At the beginning*** → *Parameters are just initialized to small values*
  - ***Toward the end*** → *Parameters get bigger to fit the training data*



Dropout (1/2)

- **Dropout ➜** Nodes are randomly removed during training
  - *The output of random nodes is temporarily **set to zero** (for one iteration)*
  - *The **dropout rate** is the fraction of nodes that are zeroed out*
  - ***Why it works?** At test time, all the nodes are kept. This is equivalent to averaging the output of all the networks randomly created during training*



Dropout (2/2)

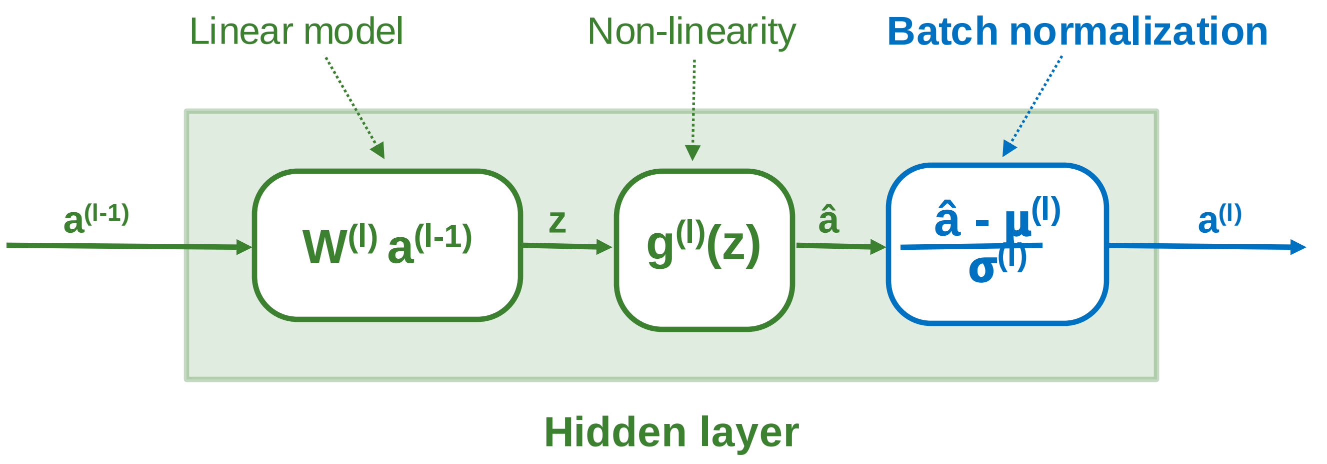
- **Inverted Dropout** (implementation)
  - Drop and scale at training time; do nothing at test time

```
p = 0.5  # prob. of keeping a unit (higher = less dropout)

def train_forward (X):
    # forward pass of 3-layer neural network at train time

    H1 = np.maximum(0, W1 @ X + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p
    H1 *= U1  # 1st dropout

    H2 = np.maximum(0, W2 @ H1 + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p
    H2 *= U2  # 2nd dropout

    out = W3 @ H2 + b3
    return out
```

```
def predict(X):
    # forward pass at test time

    H1 = np.maximum(0, W1 @ X + b1)

    H2 = np.maximum(0, W2 @ H1 + b2)

    out = W3 @ H2 + b3

    return out
```

Batch normalization (1/2)

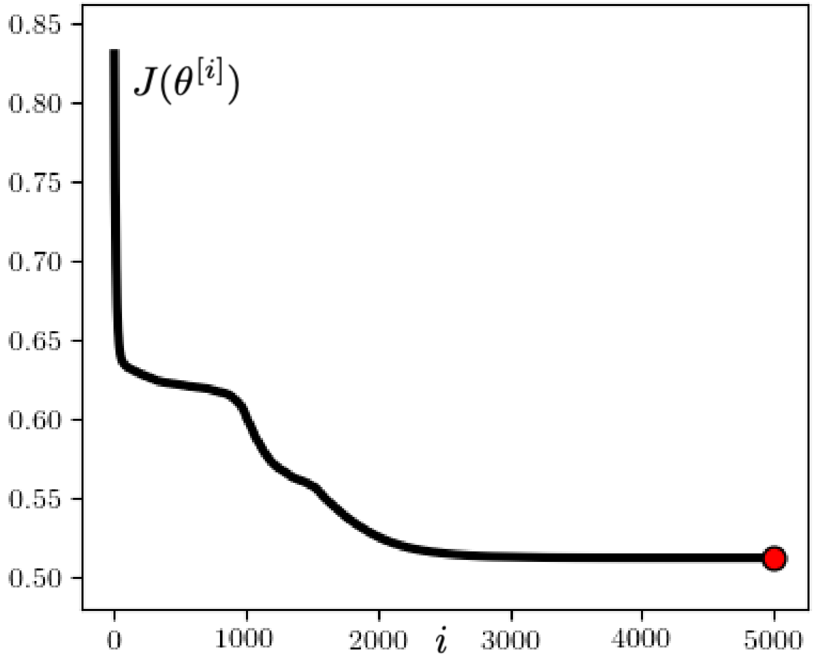
- Normalization can be also applied to hidden layers
  - ***Training*** → *Parameters* $\mu^{(l)}$ and $\sigma^{(l)}$ *are learned*
  - ***Testing*** → *Parameters* $\mu^{(l)}$ and $\sigma^{(l)}$ *are kept fixed*



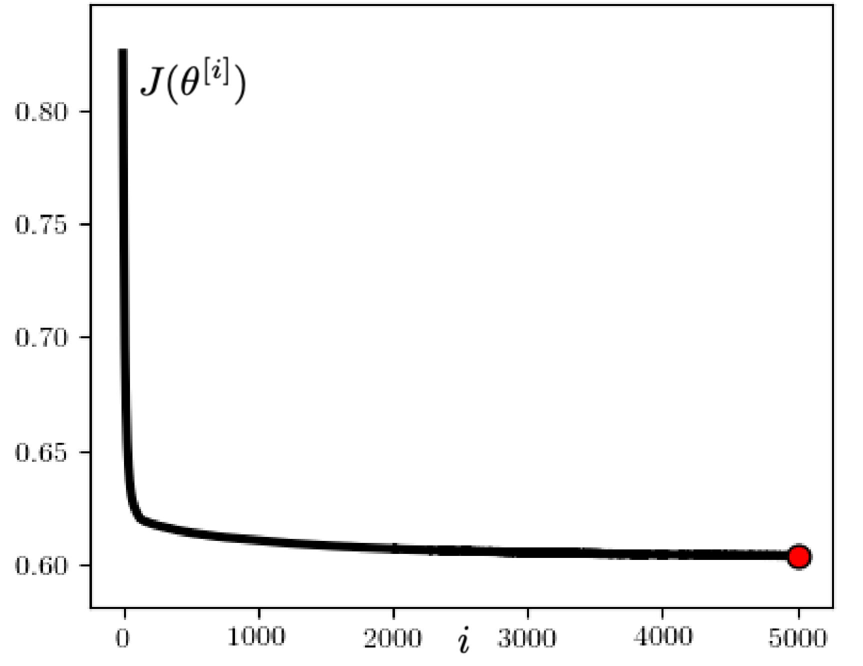
Batch normalization (2/2)

- Layer normalization **speeds up** the training process
  - *It also helps to avoid gradient explosions*

Without batch-normalization



With batch-normalization



Quiz

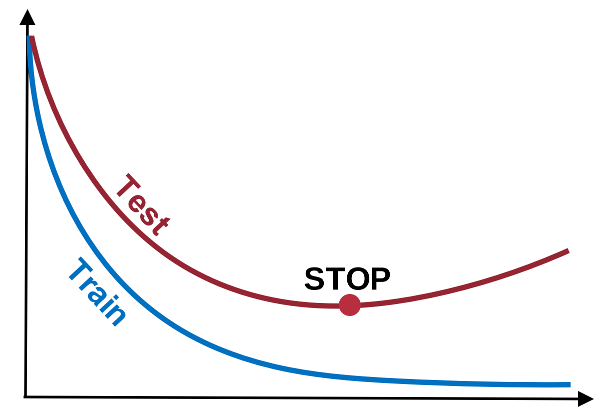
- What happens when you increase the hyper-parameter **λ**?
  1) *Weights are pushed toward becoming smaller (closer to 0)*
  2) *Weights are pushed toward becoming bigger (further from 0)*
  3) *Doubling lambda should roughly result in doubling the weights*
  4) *Gradient descent taking bigger steps with each iteration*

- What will likely happen when you increase the dropout rate?
  1) *Increasing the regularization effect*
  2) *Reducing the regularization effect*
  3) *Causing the neural network to end up with a higher training set error*
  4) *Causing the neural network to end up with a lower training set error*

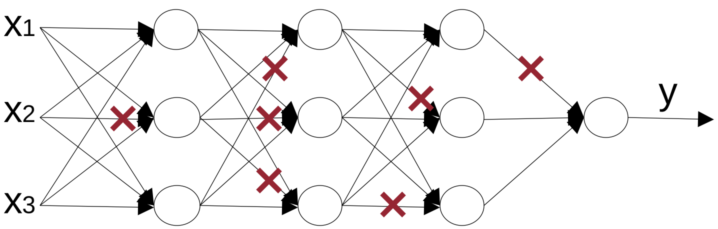
Summary so far…

- Three types of regularization

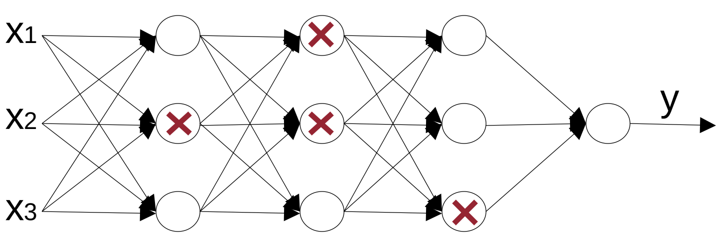
Early stopping



Norm penalization



Dropout



Hyper-parameter tuning

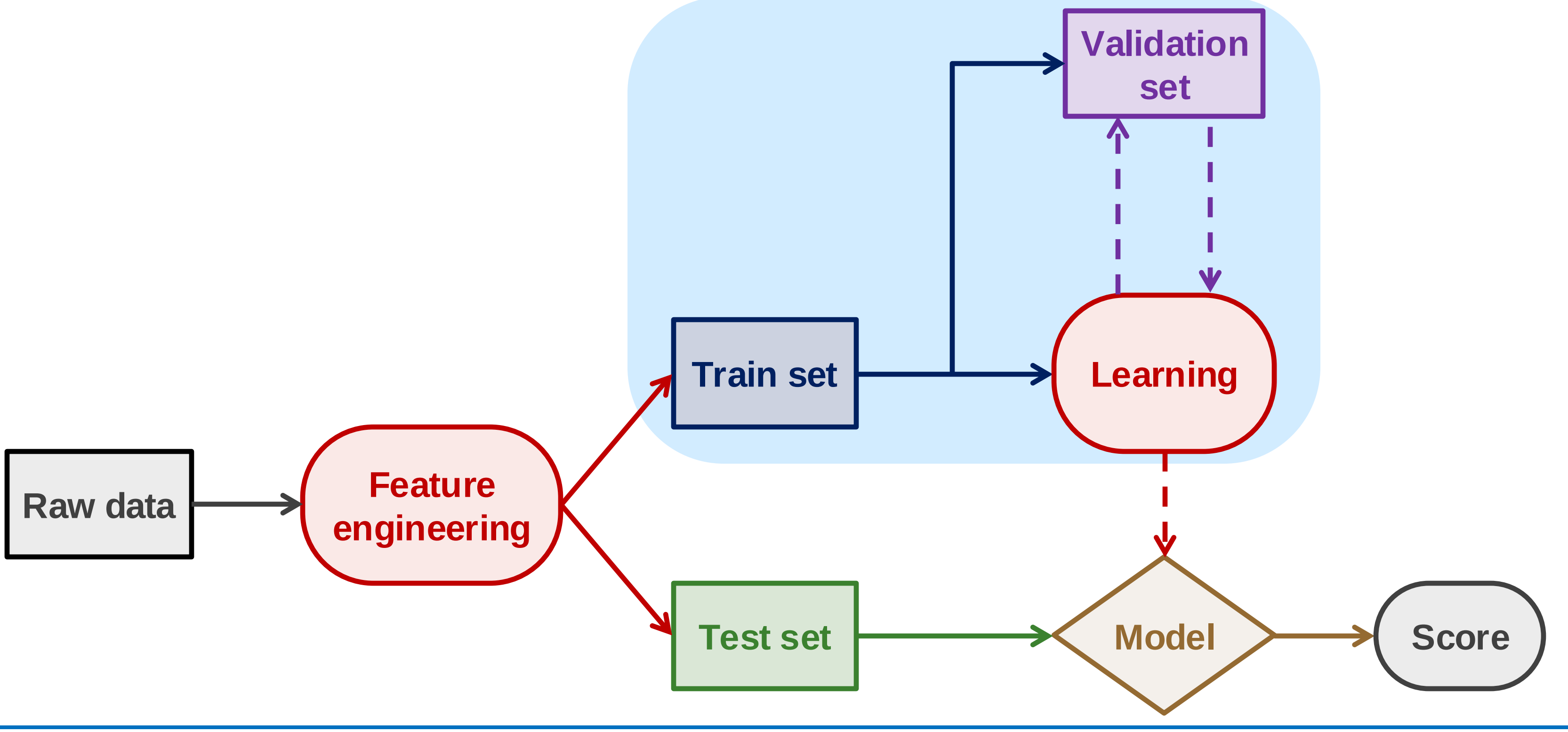
Hyper-parameters

Cross-validation

Sampling strategies

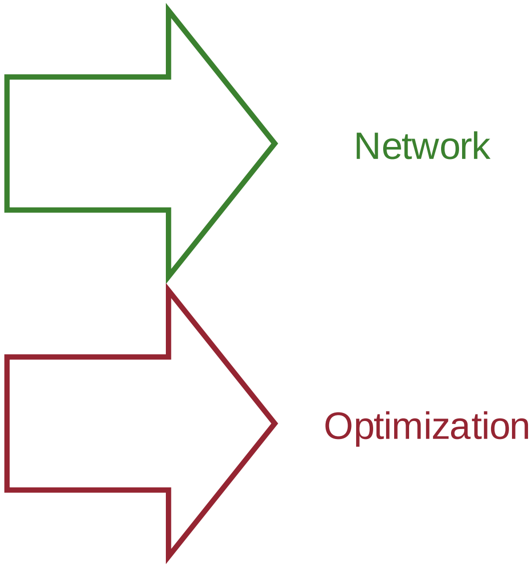
Machine learning system

- **Training pipeline**
  - How to make it work in practice?



Hyper-parameters (1/2)

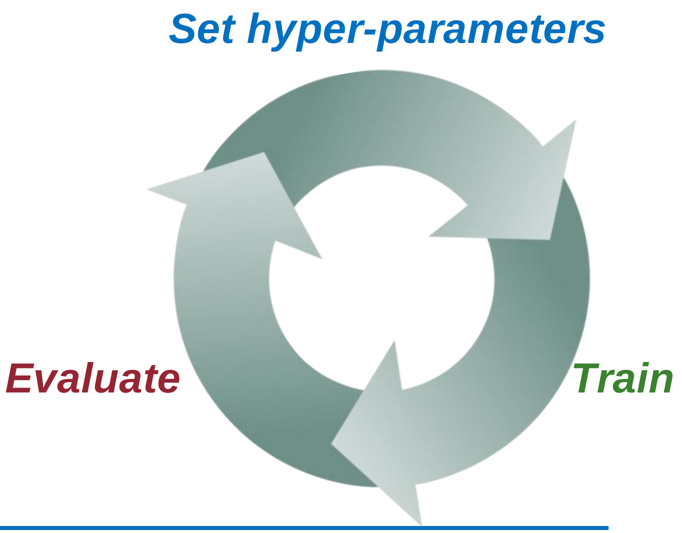
- Firstly, the **hyper-parameters** must be fixed…
  - *Number of layers in the neural network*
  - *Number of units in each layer*
  - *Activation function for each layer*
  - *Regularization*
  - *Learning rate in gradient descent*
  - *Number of iterations in gradient descent*
  - *… (and many others)*



- Then, the **parameters** can be learned via training
  - $\boldsymbol{\theta} = W^{(1)}, W^{(2)}, \ldots, W^{(L)}$

Hyper-parameters (2/2)

- How to find the best values for the hyper-parameters ?
  - *Difficult to know in advance what are the best values*
  - *Unlike parameters, they can be hardly estimated through optimization*
  - *Instead, they are found by a* ***trial-and-error*** *process*
    1) *Assign some values to hyper-parameters*
    2) *Train the network (on the train set)*
    3) *Evaluate the performance (on the valid set)*
    4) *Repeat 1-3 for different values*
    5) *Select the best values*

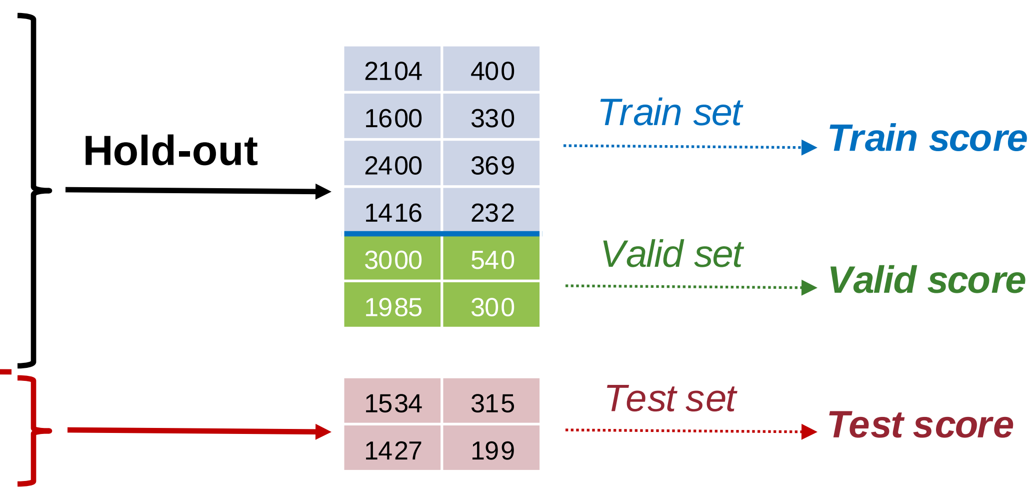


Cross-validation (1/2)

- For the evaluation, the dataset is split in three chunks
  - ***Train set*** *→ Used for training the model*
  - ***Valid set*** *→ Used for choosing the best hyper-parameters*
  - ***Test set*** *→ Used for detecting over-fitting*

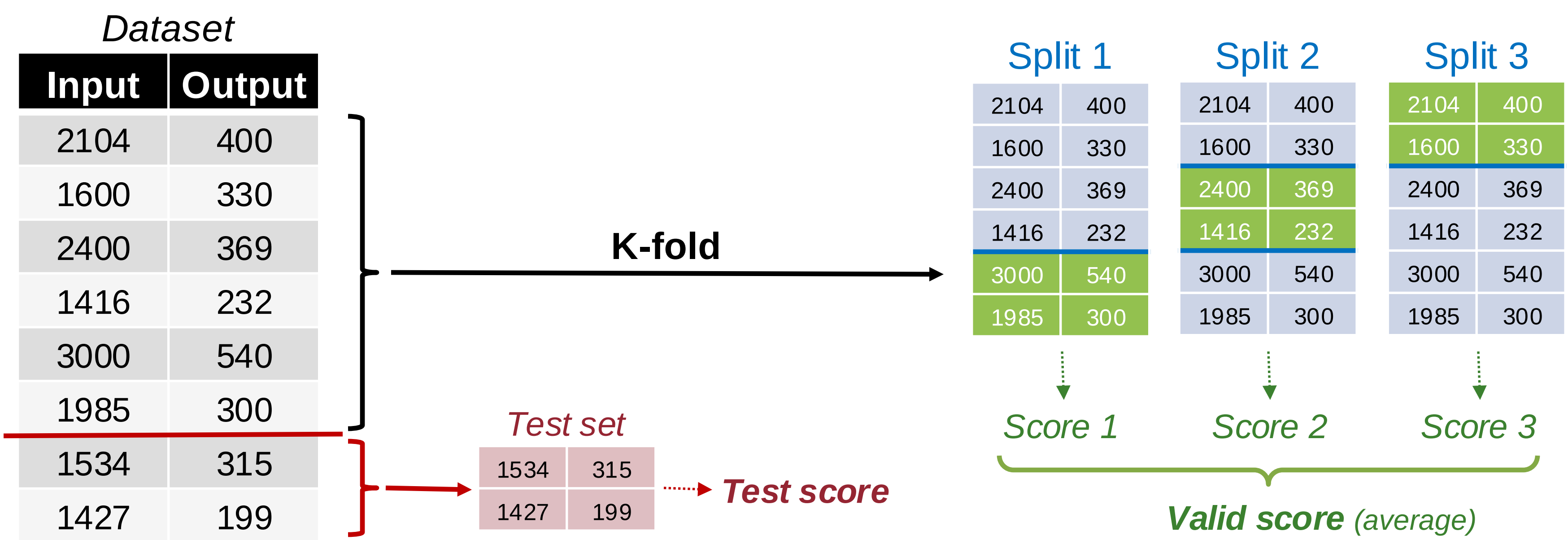
Dataset

Input	Output
2104	400
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
1427	199



Cross-validation (2/2)

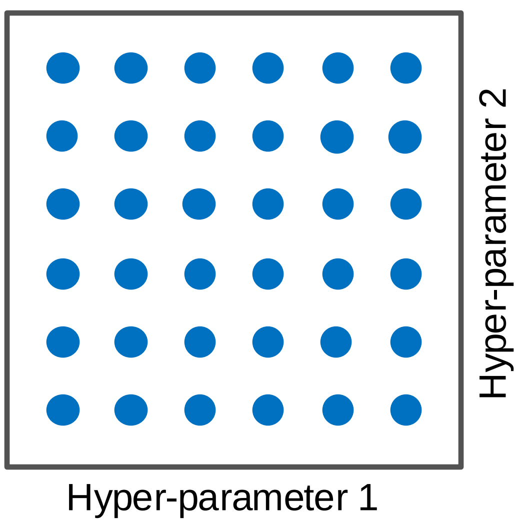
- Training data can be **shaken up** for a better evaluation
  - *Divide your data in K partitions of equal size*
  - *For each partition, use it as the valid set and the rest for training*
  - *Your final score is the average of the K scores obtained*

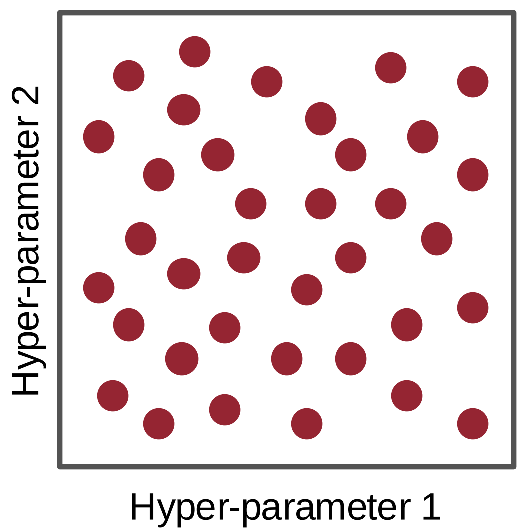


Hyper-parameter sampling (1/3)

- How to pick values for hyper-parameters ?
  - ***Uniform sampling*** → *Use a regular grid of points*
  - ***Random sampling*** → *Choose points at random (in a given range)*

This allows you to try only **6** values for each hyper-parameter

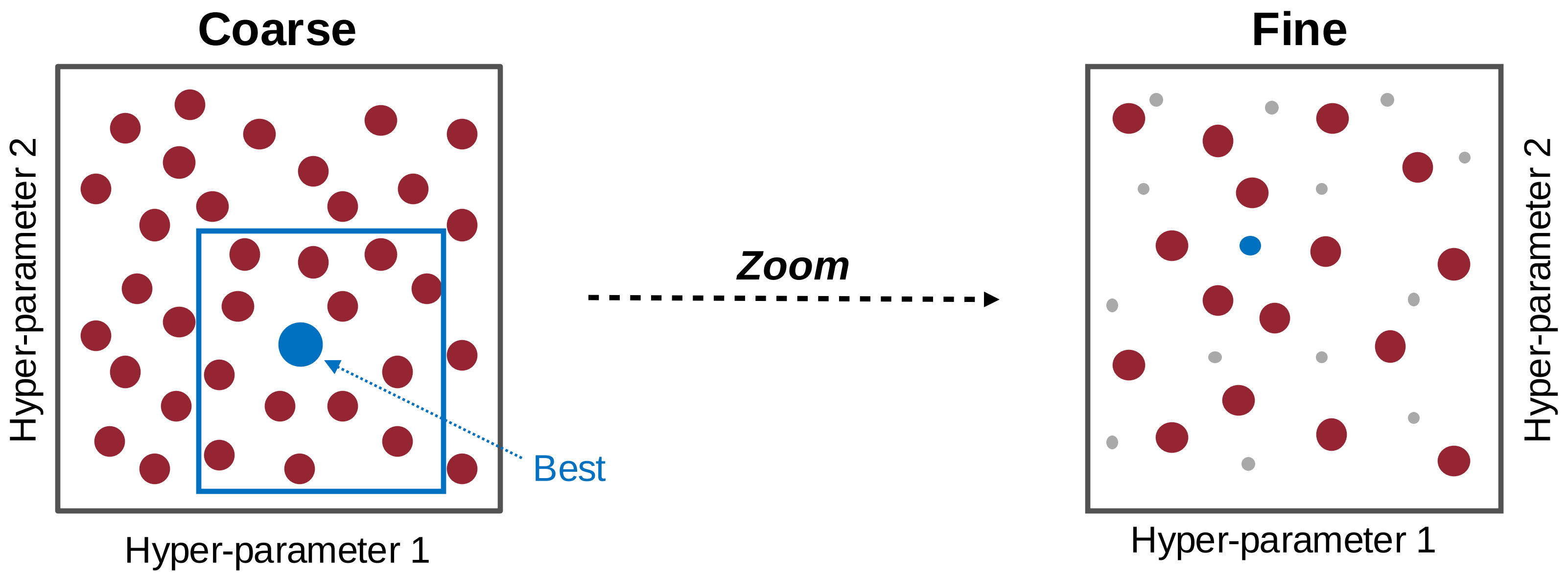




This allows you to try **6^2 = 36** values for each hyper-parameter

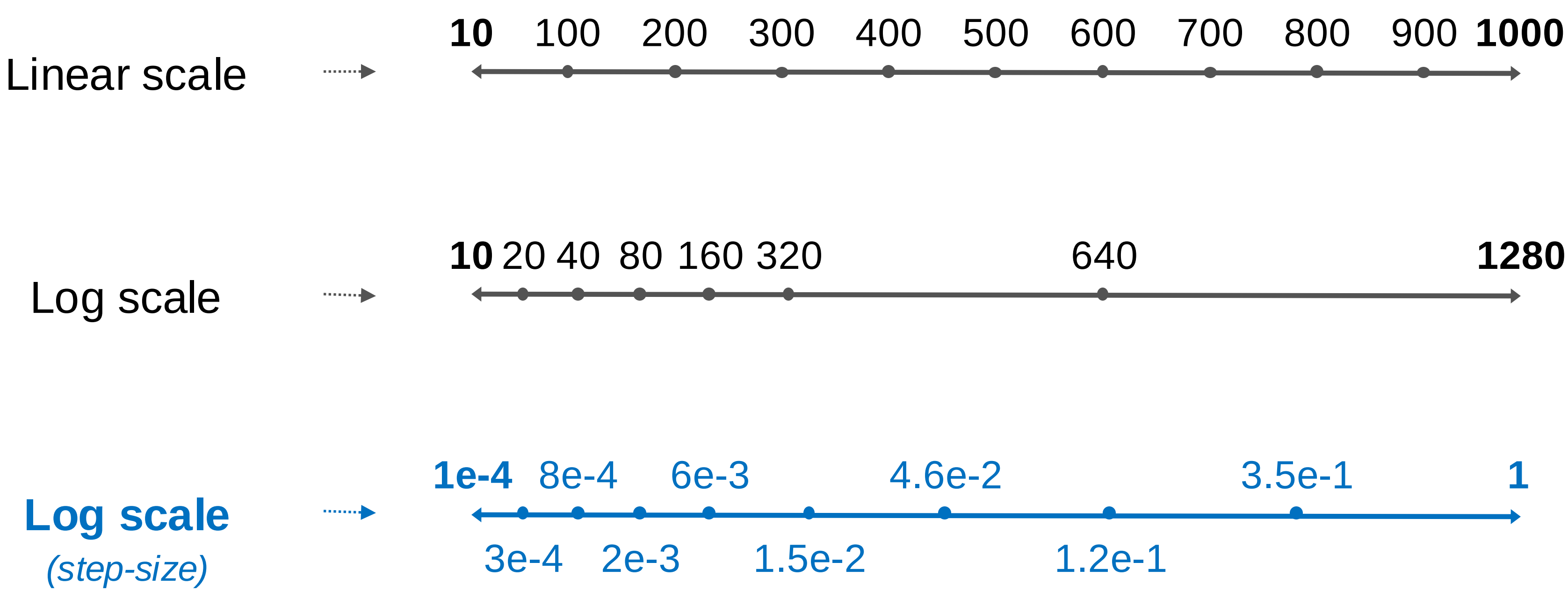
Hyper-parameter sampling (2/3)

- **Advice ➜** Use a **coarse to fine** sampling scheme



Hyper-parameter sampling (3/3)

- **Advice ➜** Consider also a **logarithmic scale** for sampling
  - *In some cases, the log scale is better than the linear one*



Quiz

- Which of the following statements are true?

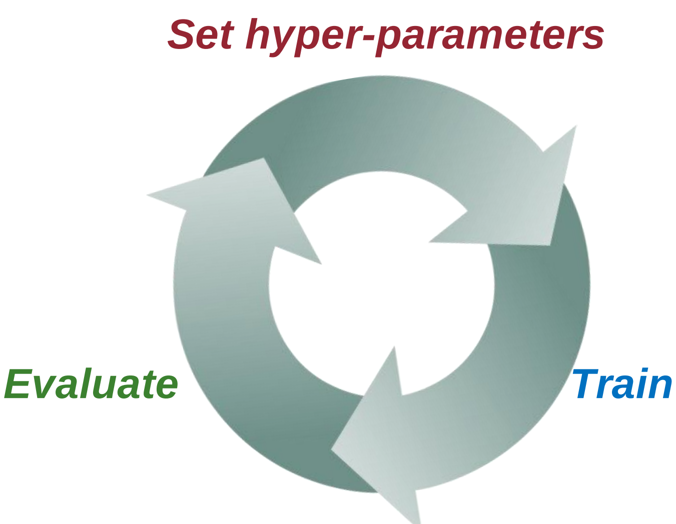
  1) *Every hyper-parameter, if set poorly, can have a huge negative impact on training, and so all of them are about equally important to tune well.*

  2) *Finding good hyper-parameter values is very time-consuming. So you should do it once at the start of the project, and try to find very good values, so that you don't ever have to revisit tuning them again.*

  3) *If you think that the step-size (hyper-parameter for gradient descent) is between 10^{-3} (= 0.001) and 10^{-1} (= 0.1), the recommended way to sample its possible values consists of using a logarithmic scale.*

Summary so far…

- **Hyper-parameter search**
  - *Use a validation set to find the best hyper-parameters*
  - *Random sampling is superior to uniform grid search*
  - *Use a logarithmic scale when it is appropriate (e.g., for step-size)*



Dataset

2104	400	Train set
1600	330	Train set
2400	369	Train set
1416	232	Train set
3000	540	Valid set
1985	300	Valid set
1534	315	Test set
1427	199	Test set

Conclusion

Training

Over-fitting

Regularization

Hyper-parameters

Training

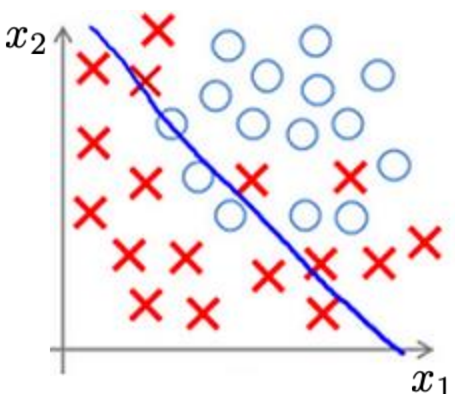
- Neural networks are trained with gradient descent

$$\text{Repeat} \quad \lfloor\ \theta \leftarrow \theta - \alpha \nabla J(\theta)$$

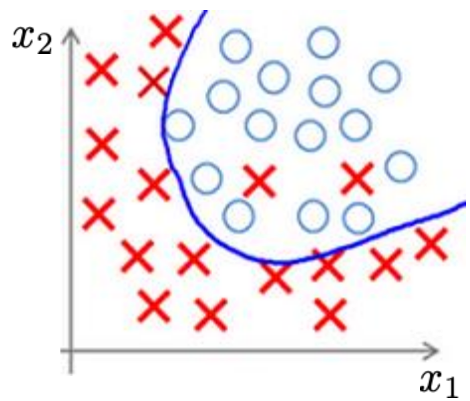
- Tricks of the trade
  1) *Data normalization* → *Speed up the optimization*
  2) *Random initialization* → *Otherwise the network won't learn*
  3) *Learning rate* → *Must be chosen small enough*
  4) *Mini-batches* → *Better generalization*

The problem of over-fitting

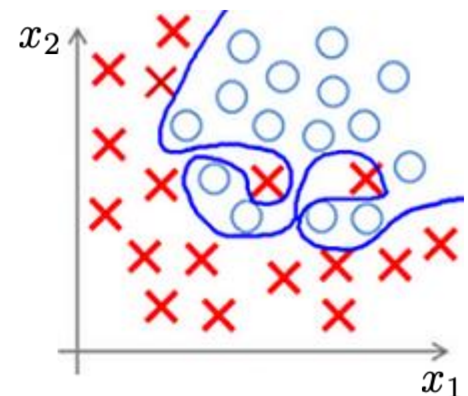
- **Bias-variance tradeoff**
  - *Over-fitting is the obstacle to generalization*
  - *Use a test set to detect over-fitting (or under-fitting)*
  - *Recipes to reduce bias and variance*



Under-fitting (high bias)



"Just right"

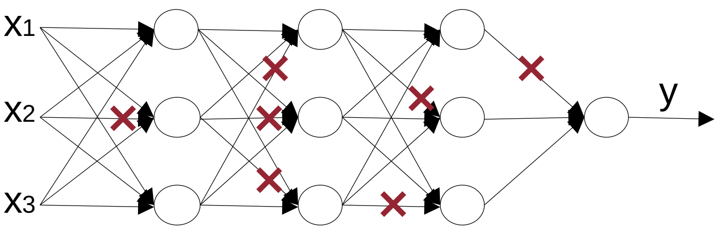


Over-fitting (high variance)

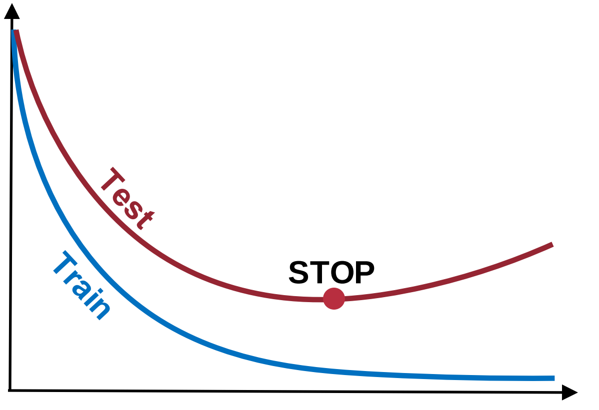
Regularization

- **Effective ways to reduce overfitting**

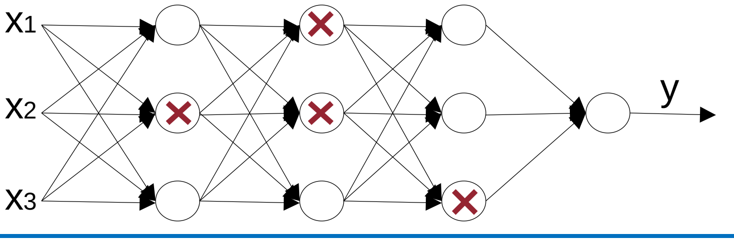
Norm penalization



Early stopping

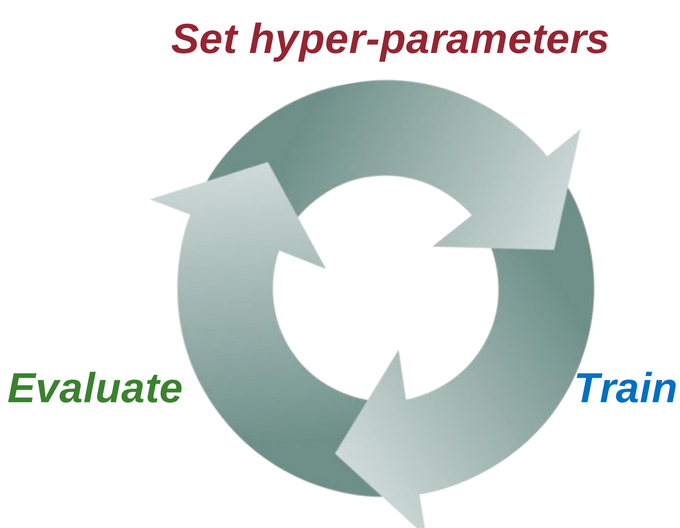


Dropout



Hyper-parameters

- **How to deal with hyper-parameters**
  - *Use a validation set to find the best hyper-parameters*
  - *Random sampling is superior to uniform grid search*
  - *Use a logarithmic scale when it is appropriate (e.g., for step-size)*



Dataset

2104	400	Train set
1600	330	Train set
2400	369	Train set
1416	232	Train set
3000	540	Valid set
1985	300	Valid set
1534	315	Test set
1427	199	Test set

Ensemble of networks

- **Advice** ➜ Train several networks and combine their outputs

1) ***Same model, different initialization.***
   - Use cross-validation to determine the best hyper-parameters, then train several models with the same hyper-parameters, but with different random initialization.

2) ***Top models discovered during cross-validation.***
   - Use cross-validation to determine the best hyper-parameters, then pick the models having the best-performing sets of hyper-parameters.

3) ***Different checkpoints of a single model.***
   - If training is very expensive, take different checkpoints of a single network over time. For example, pick a network after a fixed number of epochs. Alternatively, start with a large step-size and a decaying schedule, train the network for a fixed time, and restart with a large step-size after saving the network. Another way is to maintain a running average of network parameters during training.