The problem of over-fitting

Over-fitting
How to detect it
How to fight it

Over-fitting (1/3)

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

- But only after the **hyper-parameters** are fixed…
  - L ➜ *Number of layers in the neural network*
  - M_l ➜ *Number of units in each layer*
  - $g^{(l)}$ ➜ *Activation function for each layer*
  - … *(and many others)*

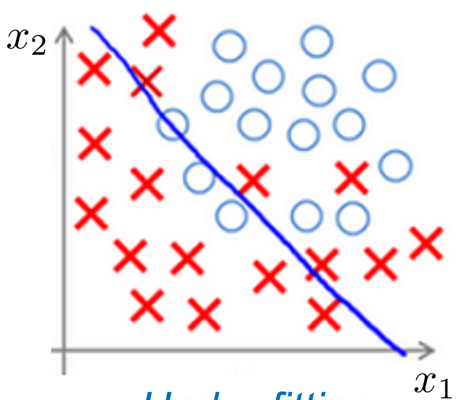
Network architecture

- Hyper-parameters are difficult to guess on the first attempt

Over-fitting (2/3)

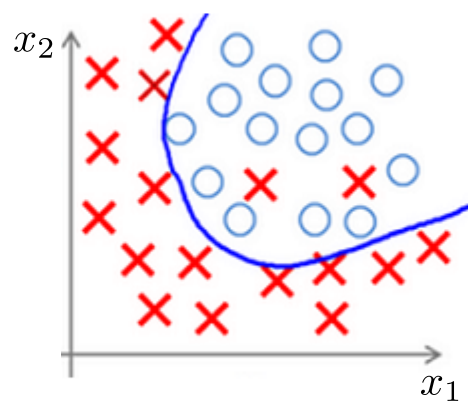
- What is the impact of hyper-parameters on learning ?
  - ***Under-fitting*** **➜** *The prediction is **<u>too far</u>** from the training data*
  - ***Over-fitting*** **➜** *The prediction is **<u>too close</u>** to the training data*

<u>Small network</u>



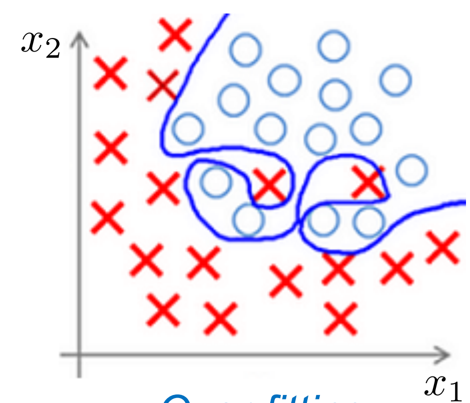
Under-fitting
(high bias)

<u>Medium network</u>



"Just right"

<u>Big network</u>

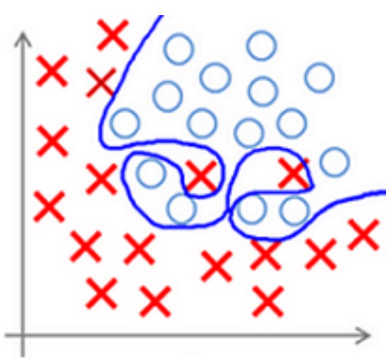


Over-fitting
(high variance)

Over-fitting (3/3)

- Learning aims at achieving 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.

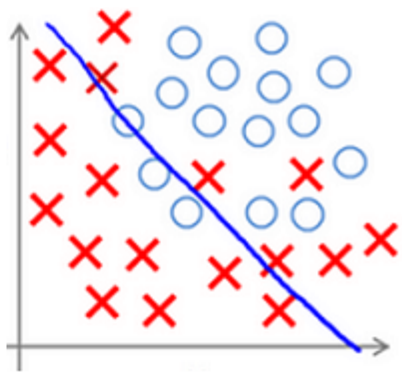
How to detect over-fitting (1/4)

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

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

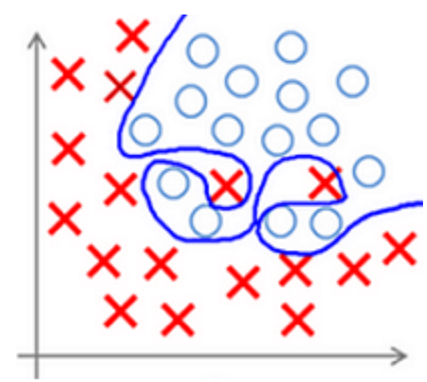
  - ***Warning*** **➜** *This estimate is biased toward* ***over-fitting*** *!!!*

Under-fitting



high ← $\mathbf{J_{Train}}$ → *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 detecting over-fitting*

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

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

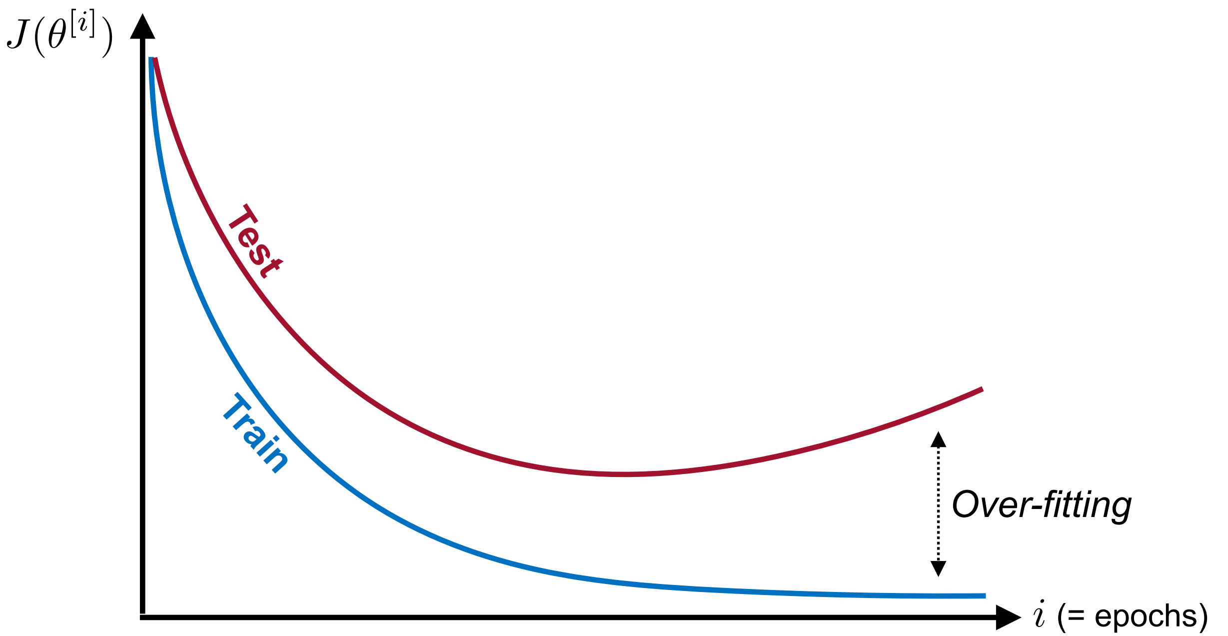
How to detect over-fitting (3/4)

- 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 %**	*Small gap in performance*
High Variance (over-fitting)	Err_{Train} = **1.0 %** Err_{Test} = **19.3 %**	Err_{Train} = **15.0 %** Err_{Test} = **30.0 %**	*Big gap in performance*
	Small error on training	*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*



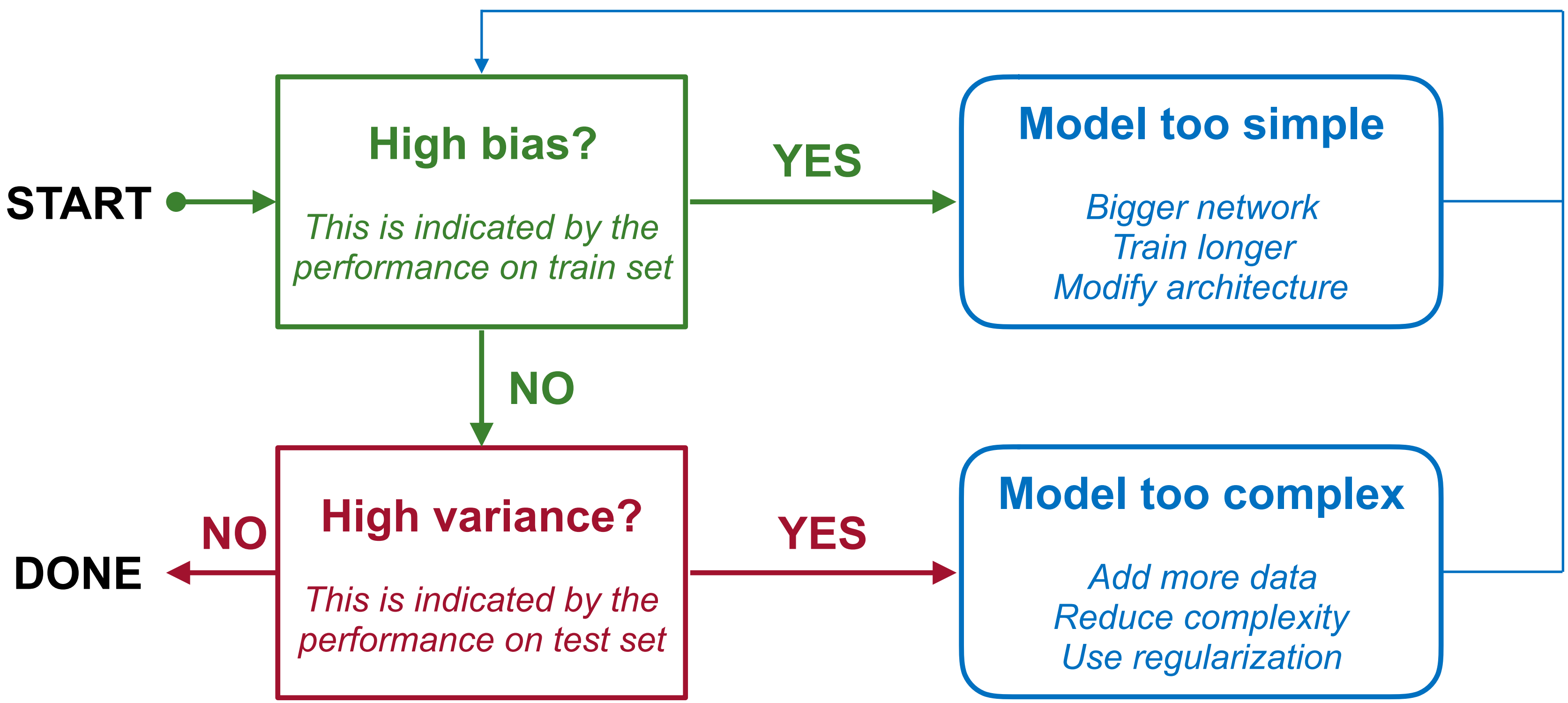
How to fight over-fitting (1/3)

- 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*

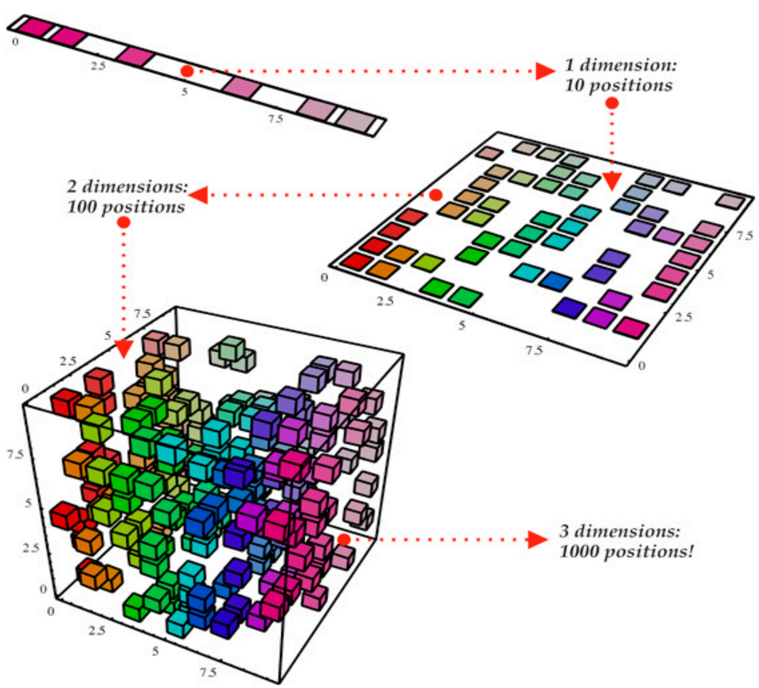
How to fight over-fitting (2/3)

- 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*



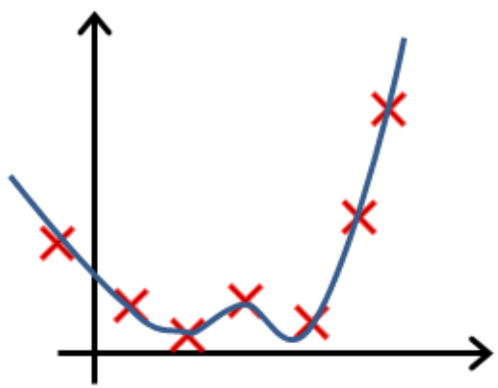
Exponential growth

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

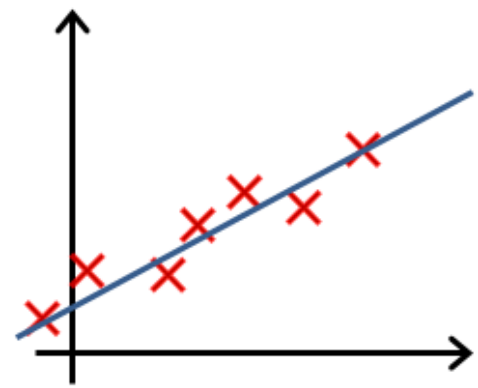
Quiz (1/3)

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

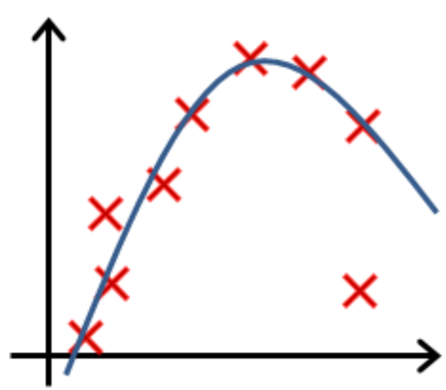
1.



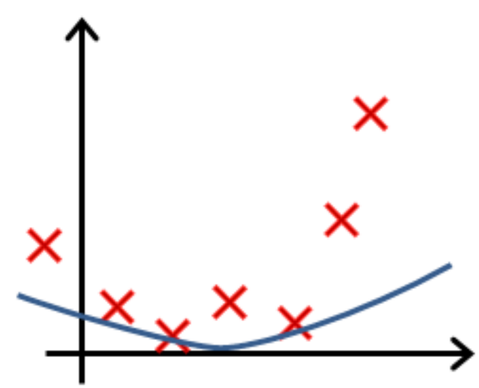
2.



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.*

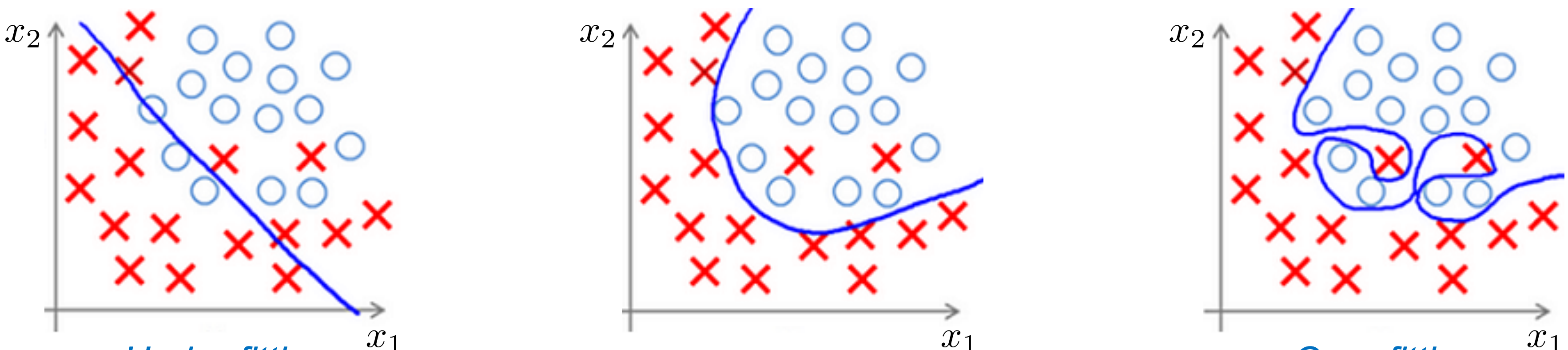
Quiz (3/3)

- 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*

What we have seen 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

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 $\boldsymbol{\theta_1,\ldots,\theta_M}$
  - *The cost function is modified as follows:*

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

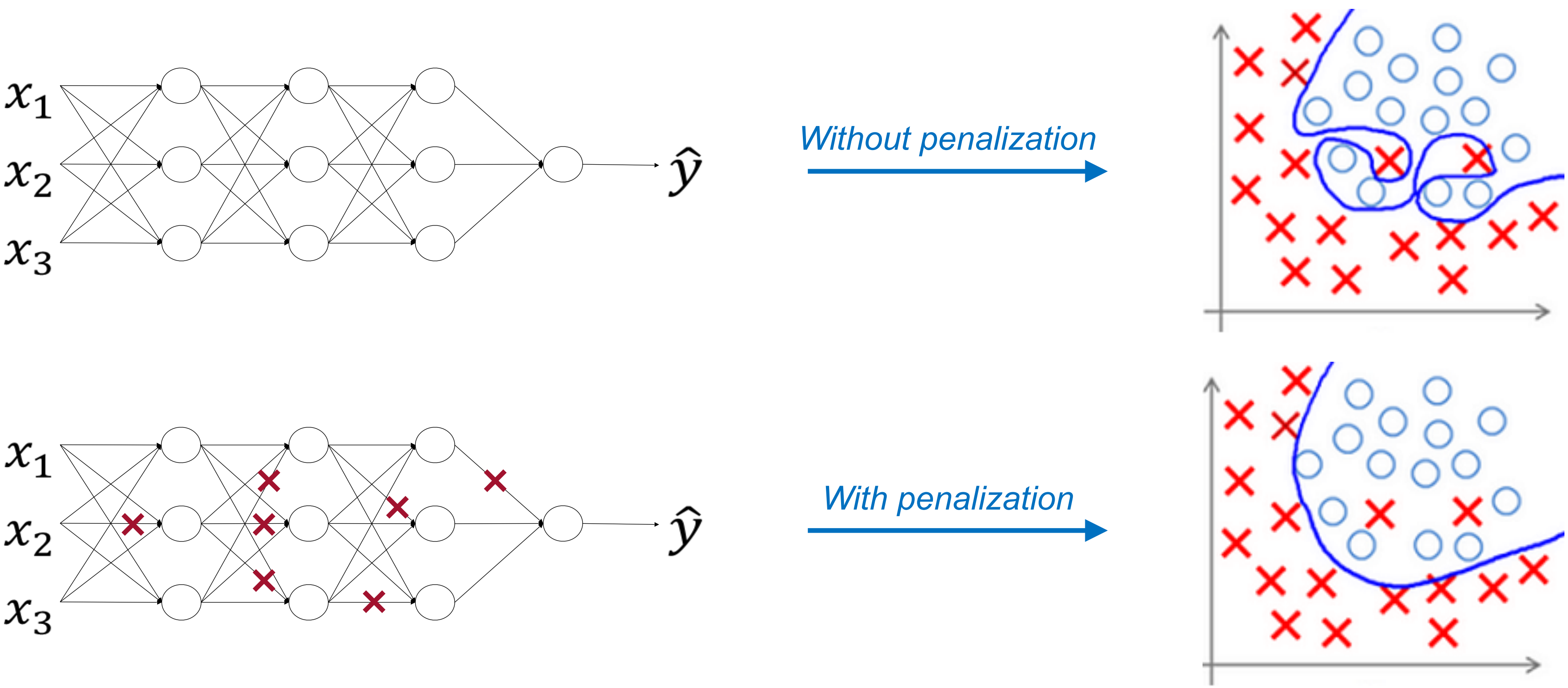
  - *Now, the cost function is minimized for smaller values of* $\boldsymbol{\theta_1,\ldots,\theta_M}$

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

  - *Small values for* $\boldsymbol{\theta_1,\ldots,\theta_M}$ *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*



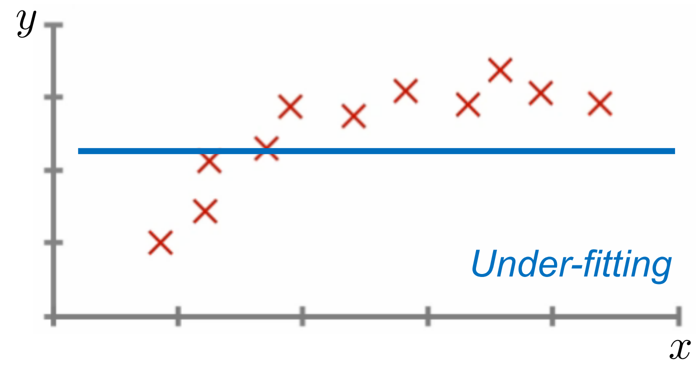
Norm penalization (3/3)

- 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*

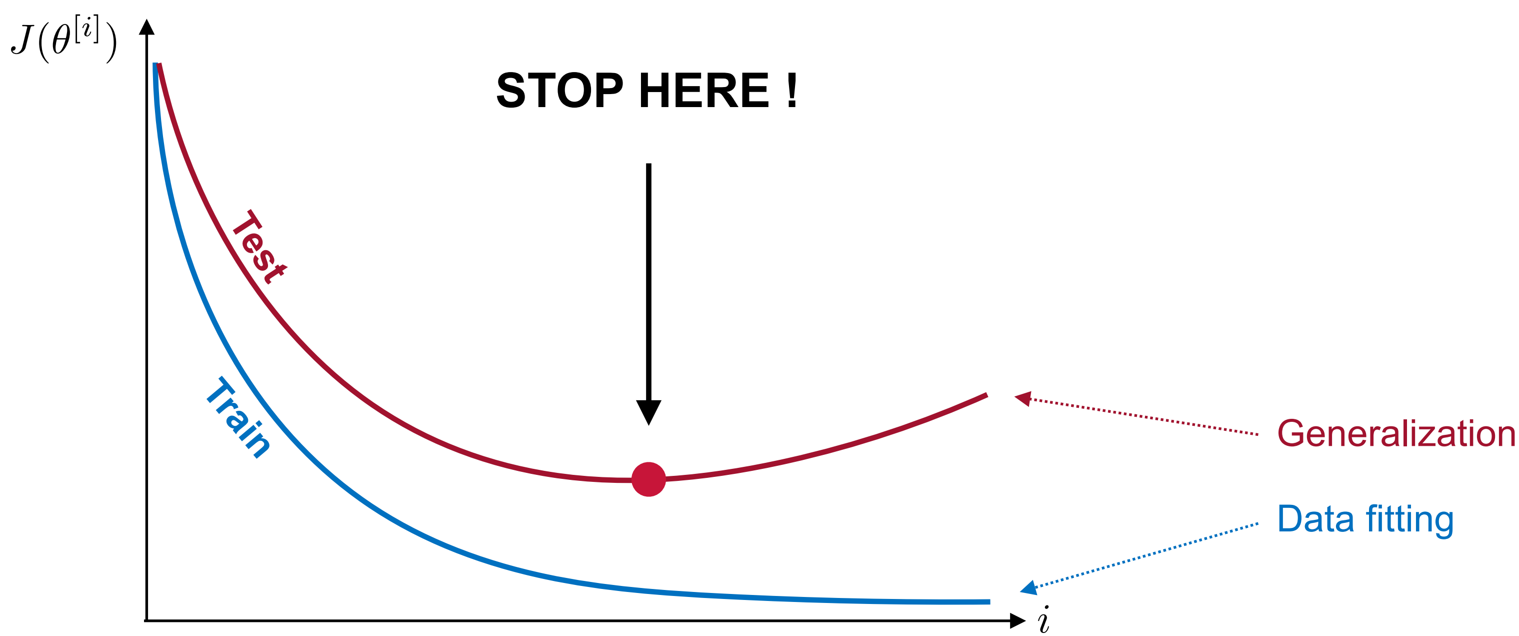
$$\lambda \to +\infty \qquad \Rightarrow \qquad \theta_1 \approx 0, \dots, \theta_M \approx 0$$

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



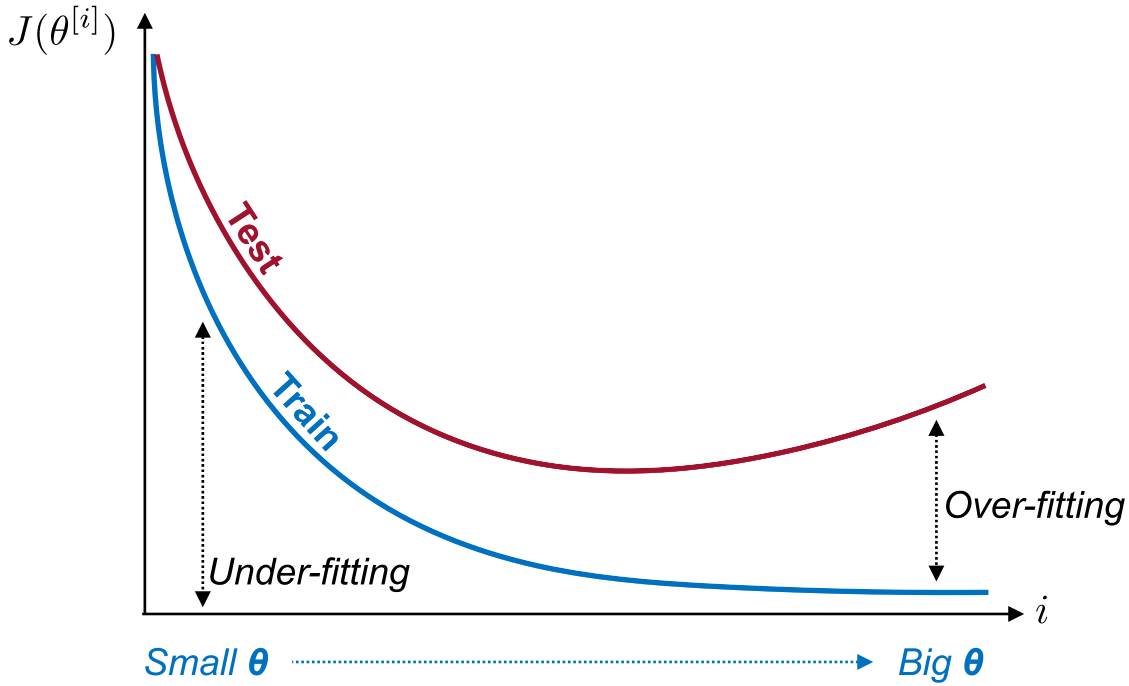
Early stopping (1/2)

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



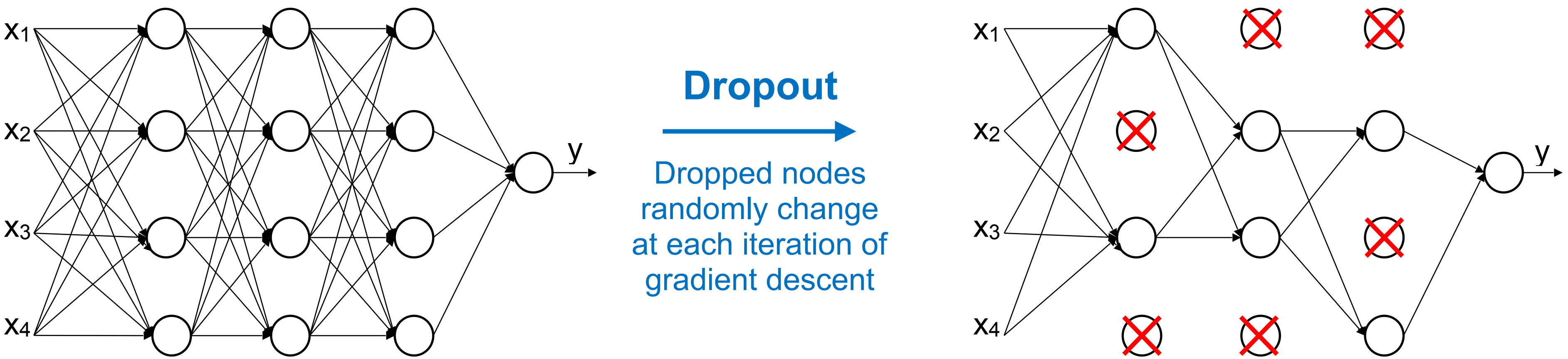
Early stopping (2/2)

- The magnitude of $\boldsymbol{\theta}_1,\ldots,\boldsymbol{\theta}_M$ increases during training
  - ***At the beginning*** ➜ $\boldsymbol{\theta}_1,\ldots,\boldsymbol{\theta}_M$ *are just initialized to small values*
  - ***Toward the end*** ➜ $\boldsymbol{\theta}_1,\ldots,\boldsymbol{\theta}_M$ *get bigger and bigger to fit the training data*



Dropout

- **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*



Quiz

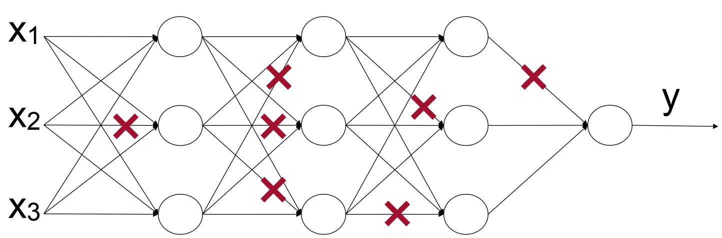
- What happens when you increase the hyper-parameter **λ**?
  1) *Weights are pushed toward becoming smaller (closer to 0)*
  2) *Weights are pushed toward becoming bigger (further from 0)*
  3) *Doubling lambda should roughly result in doubling the weights*
  4) *Gradient descent taking bigger steps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

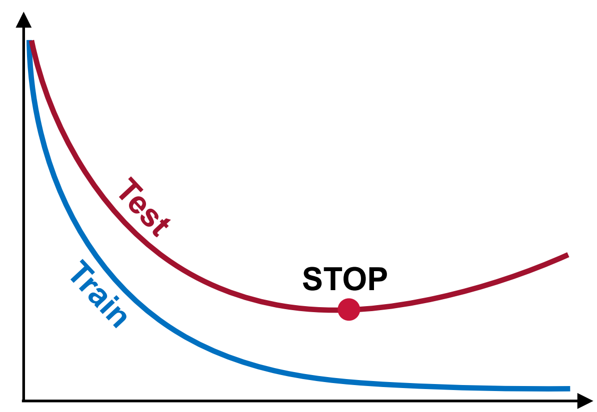
What we have seen so far…

- Three types of regularization

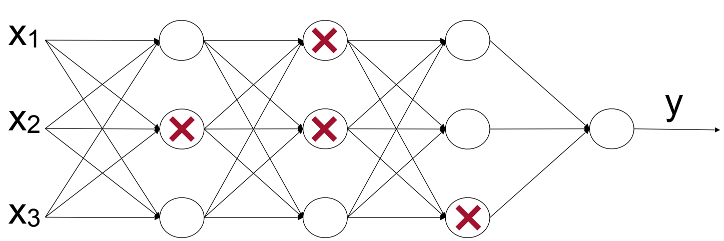
Norm penalization



Early stopping



Dropout



Hyper-parameter tuning

Hyper-parameters
Cross-validation
Sampling strategies

Hyper-parameters (1/2)

- Firstly, the **hyper-parameters** must be fixed…
  - ***L*** ➜ *Number of layers in the neural network*
  - ***M_l*** ➜ *Number of units in each layer*
  - ***$g^{(l)}$*** ➜ *Activation function for each layer*
  - ***λ*** ➜ *Regularization*

Network

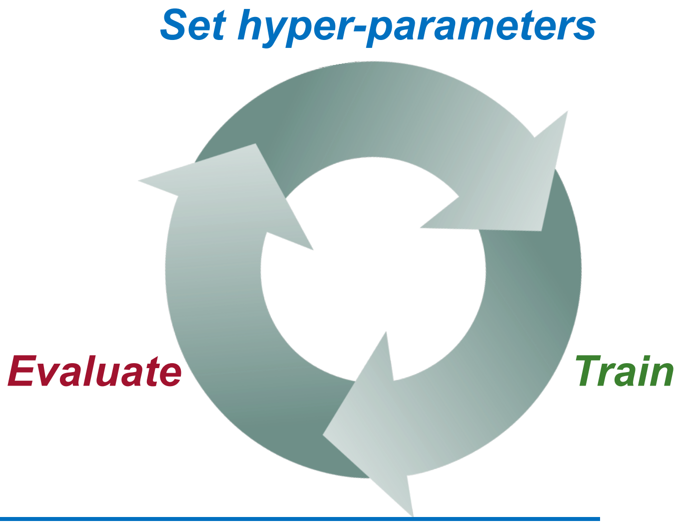
  - ***α_i*** ➜ *Step-size in gradient descent*
  - ***I_{max}*** ➜ *Iterations in gradient descent*
  - *… (and many others)*

Optimization

- 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) *Fix a set of values*
    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 ones*



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

Hold-out

2104	400
1600	330
2400	369
1416	232

Train set → ***Train score***

3000	540
1985	300

Valid set → ***Valid score***

1534	315
1427	199

Test set → ***Test score***

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*

Dataset

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

K-fold

Split 1

2104	400
1600	330
2400	369
1416	232
3000	540
1985	300

Score 1

Split 2

2104	400
1600	330
2400	369
1416	232
3000	540
1985	300

Score 2

Split 3

2104	400
1600	330
2400	369
1416	232
3000	540
1985	300

Score 3

Valid score *(average)*

Test set

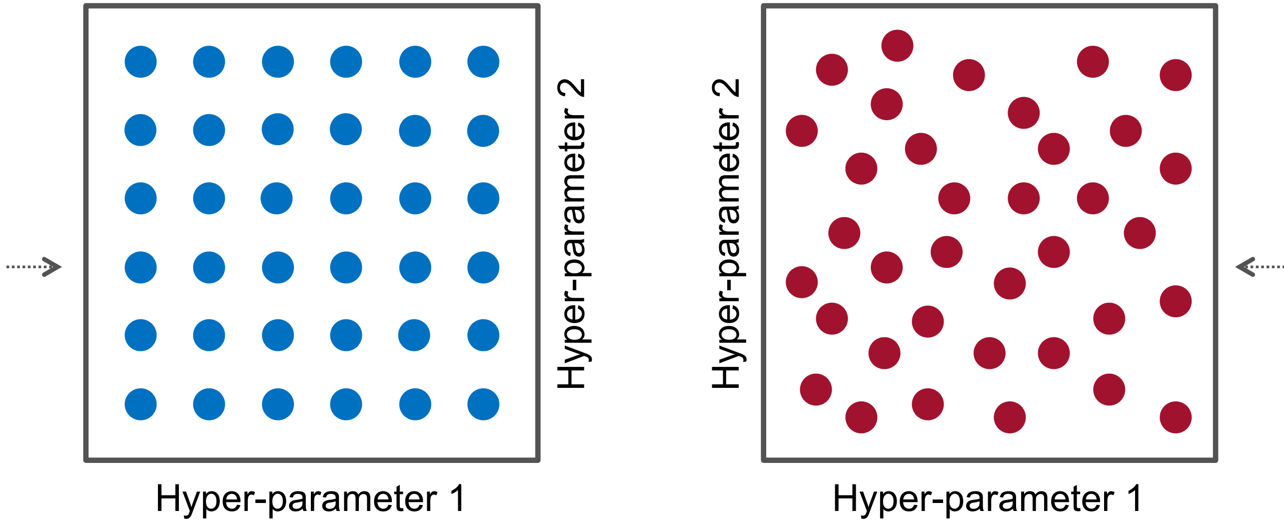
1534	315
1427	199

Test score

Hyper-parameter sampling (1/3)

- How to select a set of values to explore ?
  - ***Uniform sampling*** ➜ *Use a regular grid of points*
  - ***Random sampling*** ➜ *Choose points at random (in a given range)*

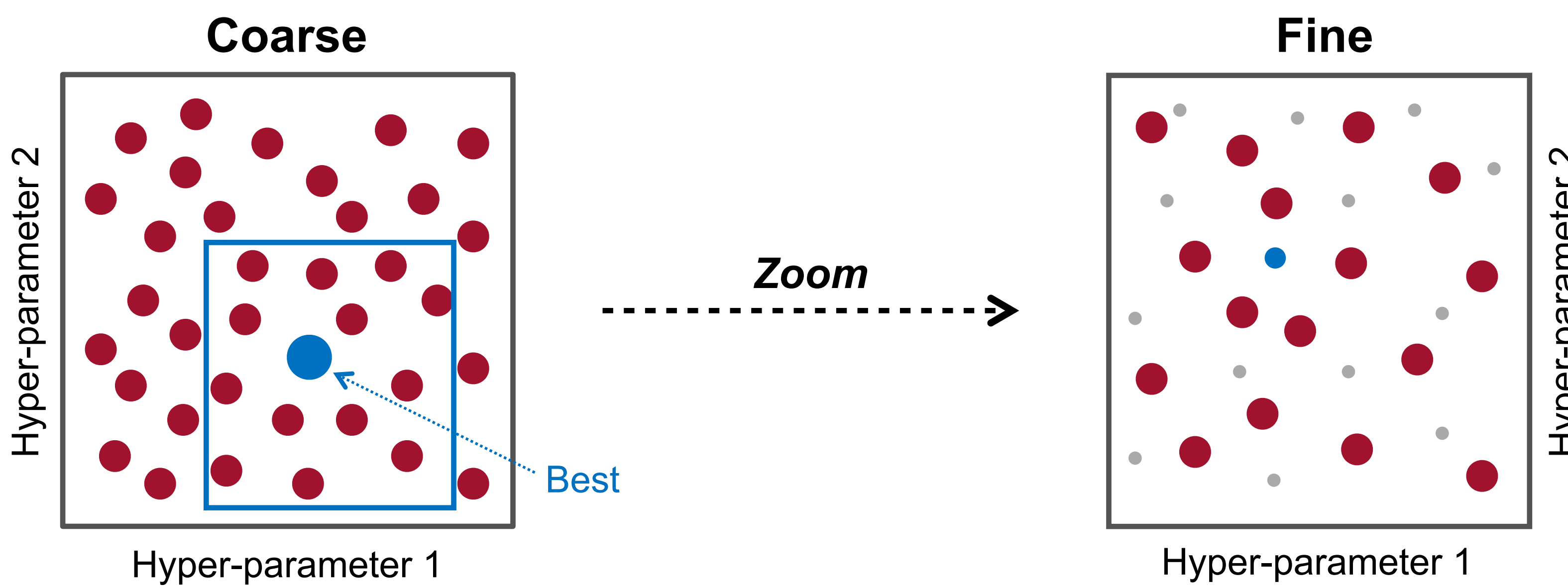
This allows you to try only **6** values for each hyper-parameter



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

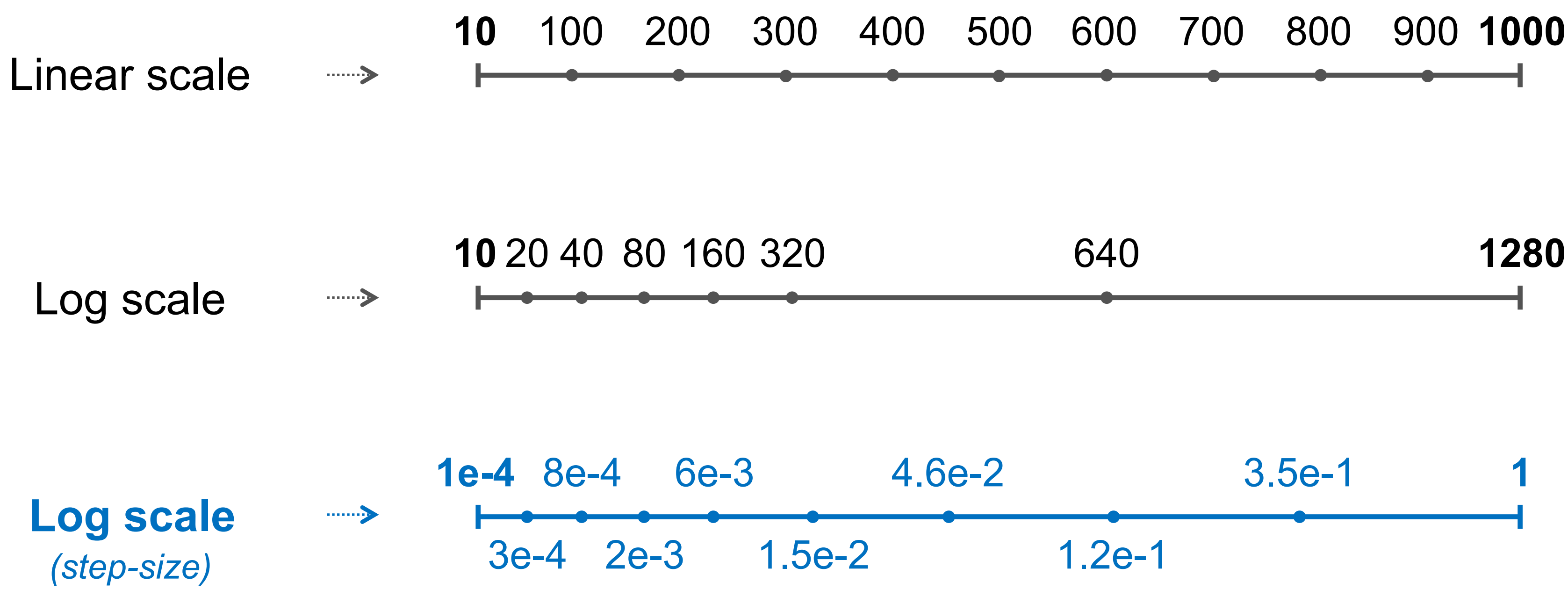
Hyper-parameter sampling (2/3)

- **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) *If searching among a large number of hyper-parameters, you should try values in a grid rather than random values, so that you can carry out the search more systematically and not rely on chance.*

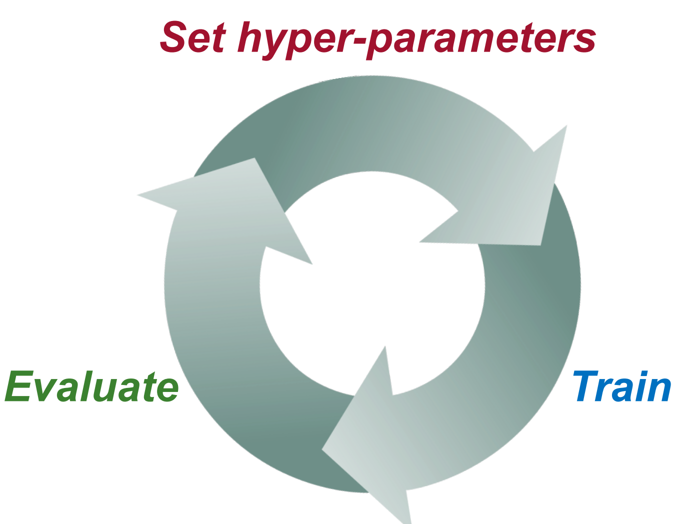
  2) *Every hyper-parameter, if set poorly, can have a huge negative impact on training, and so all of them are about equally important to tune well.*

  3) *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.*

  4) *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.*

What we have seen 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
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
1427	199

Train set

Valid set

Test set

Advanced optimization

Stochastic gradient descent
Normalized gradient descent
State-of-the-art

Stochastic gradient descent (1/4)

- **Standard gradient descent**
  - *The loss function contains a term for every single example* $\boldsymbol{(x^{(n)},y^{(n)})}$

$$J(\theta) = \sum_{n=1}^{N} \mathcal{C}\Big(f_\theta(\mathrm{x}^{(n)}), \mathrm{y}^{(n)}\Big)$$

All data

Training set

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

  - *This can be **a lot to compute** for gradient descent, as it needs to go through all data at each iteration*

$$\theta^{[i+1]} = \theta^{[i]} - \alpha_i \sum_{n=1}^{N} \nabla\mathcal{C}\Big(f_{\theta^{[i]}}(\mathrm{x}^{(n)}), \mathrm{y}^{(n)}\Big)$$

Stochastic gradient descent (2/4)

▪ Stochastic gradient descent

- *At each iteration, select a block of training data*

$$J^{[i]}(\theta) = \sum_{n \in \mathbb{L}_i \subset \{1,\dots,N\}} \mathcal{C}\Big(f_\theta(\mathrm{x}^{(n)}), y^{(n)}\Big)$$

Select

Training set

Block 1	$x^{(1)}$	$y^{(1)}$
	$x^{(2)}$	$y^{(2)}$
Block 2	$x^{(3)}$	$y^{(3)}$
	$x^{(4)}$	$y^{(4)}$
	…	…
	…	…
Block B	$x^{(N-1)}$	$y^{(N-1)}$
	$x^{(N)}$	$y^{(N)}$

- *Then, compute the gradient w.r.t. the selected block*

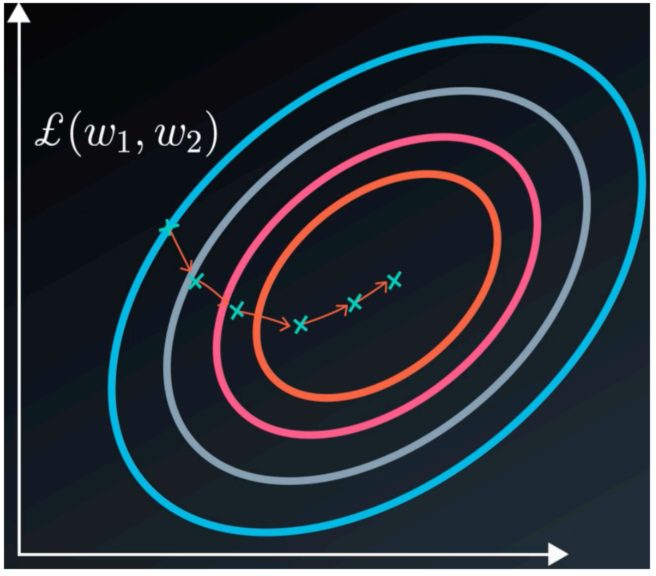
$$\boxed{\theta^{[i+1]} = \theta^{[i]} - \alpha_i \nabla J^{[i]}\big(\theta^{[i]}\big)}$$

- ***Important*** **➜** *After a complete sweep, randomly shuffle the training set*

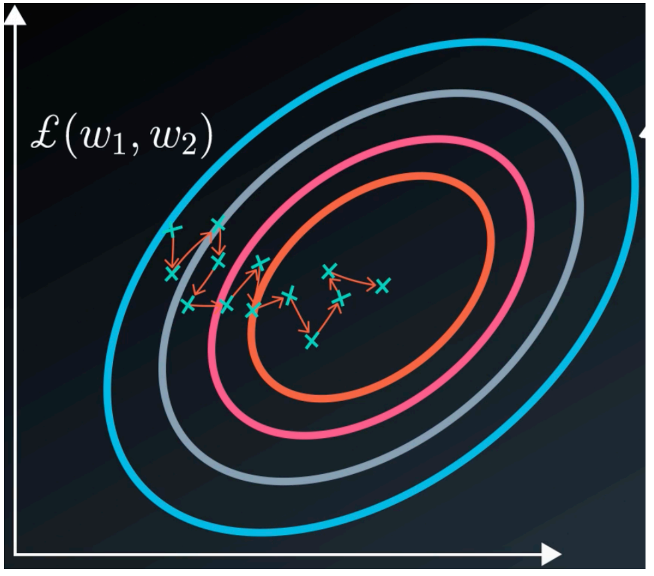
Stochastic gradient descent (3/4)

- Stochastic gradient **approximates** the "true" gradient
  - *Hence, it does not indicate the right descent direction*
  - *We compensate by taking many smaller steps (instead of few large ones)*

Gradient descent

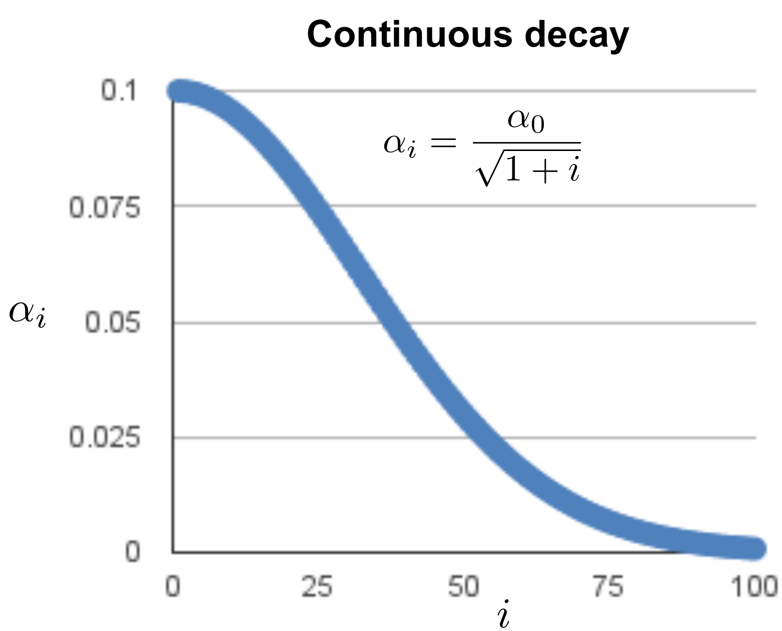


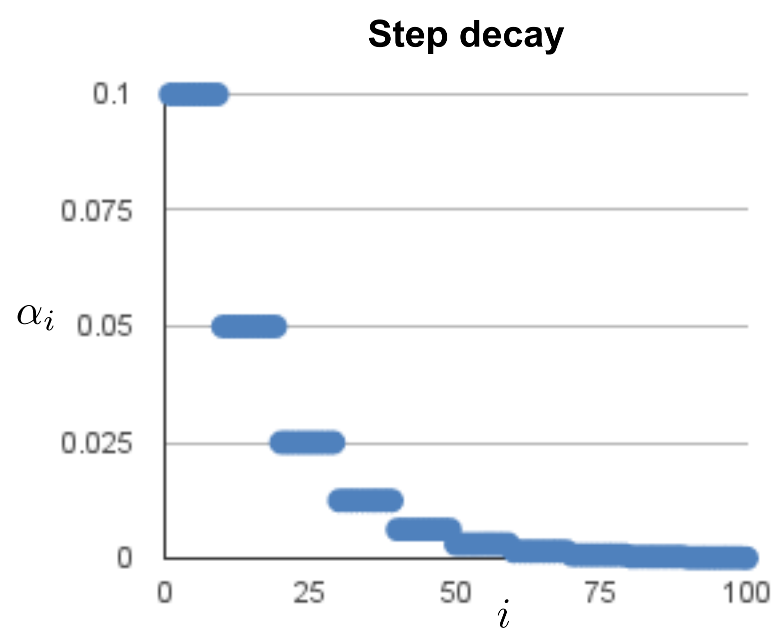
Stochastic gradient descent



Stochastic gradient descent (4/4)

- SGD needs to take many steps to ensure convergence
  - ***Advice 1*** ➜ *Decrease the step-size over time*
  - ***Advice 2*** ➜ *The initial step-size* $\boldsymbol{\alpha_0}$ *can be larger*

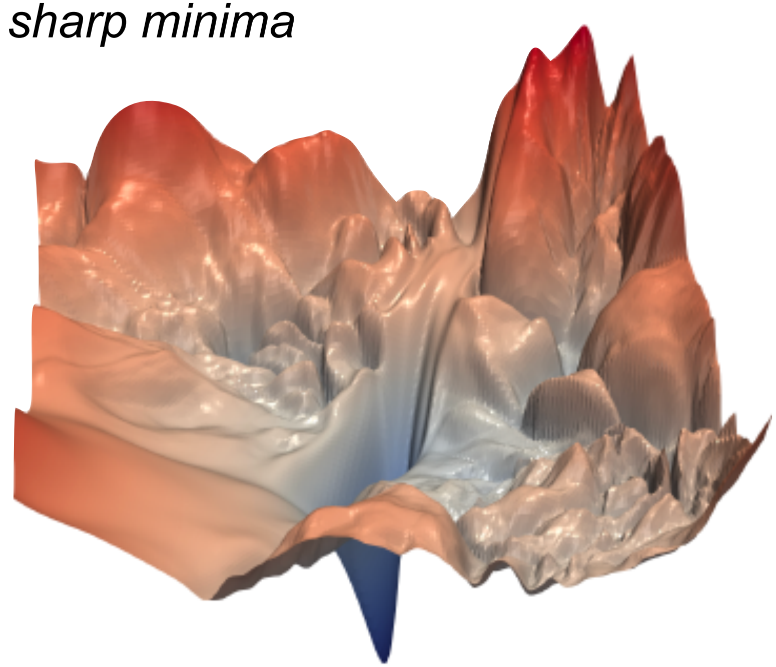




Saddle points and plateaus (1/3)

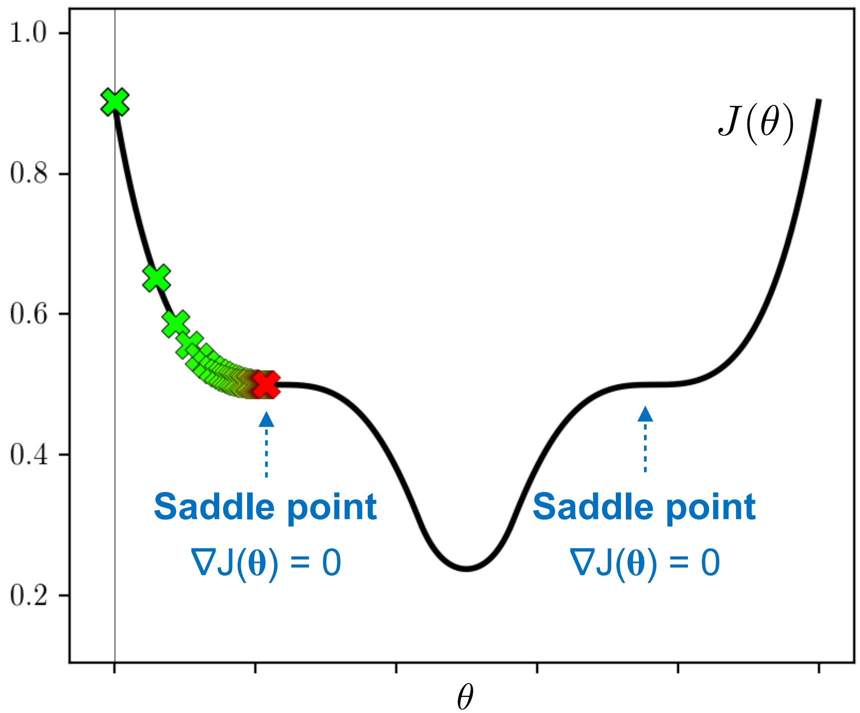
- Neural network cost function is **non-convex**
  - ***Local minima*** *dominate in shallow networks*
  - ***Saddle points*** *dominate in deep networks*
  - *Most local minima are* ***close to the bottom*** *(i.e., the global minimum)*
  - ***Flat minima*** *generalize better than sharp minima*

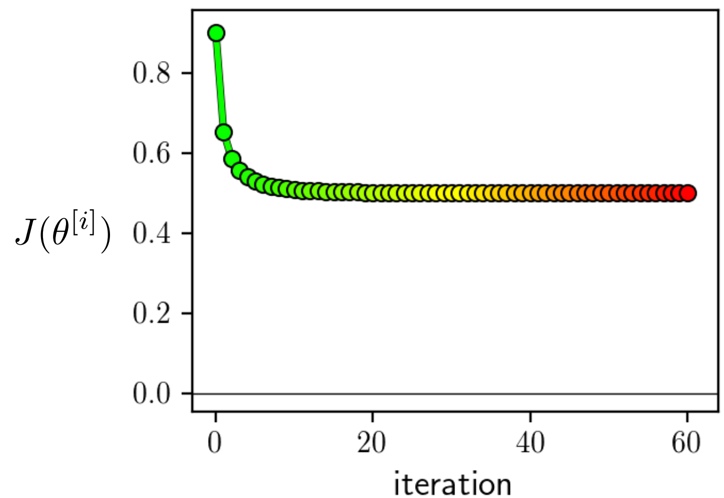
Pictorial representation of a neural network cost function



Saddle points and plateaus (2/3)

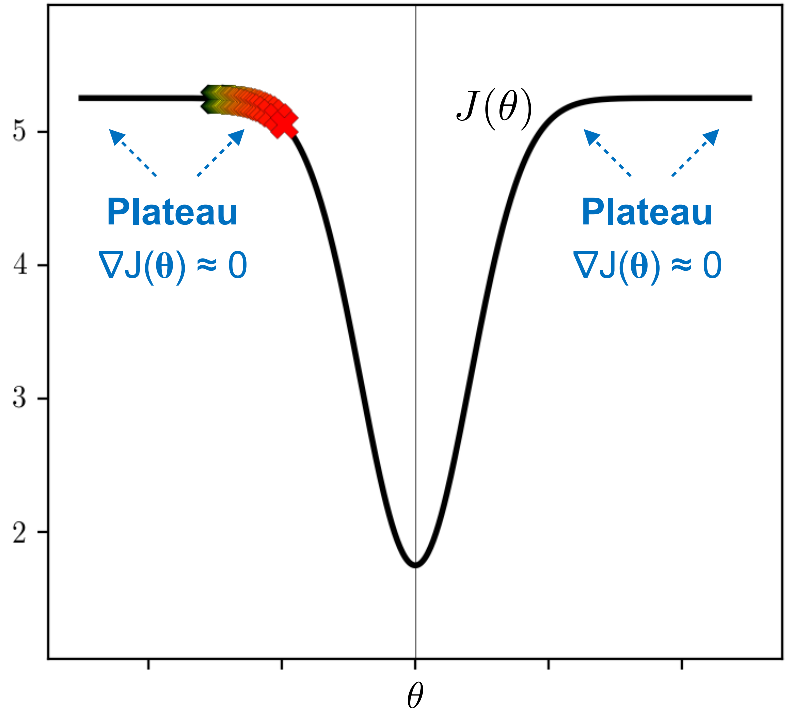
- Gradient descent **gets stuck** in saddle points

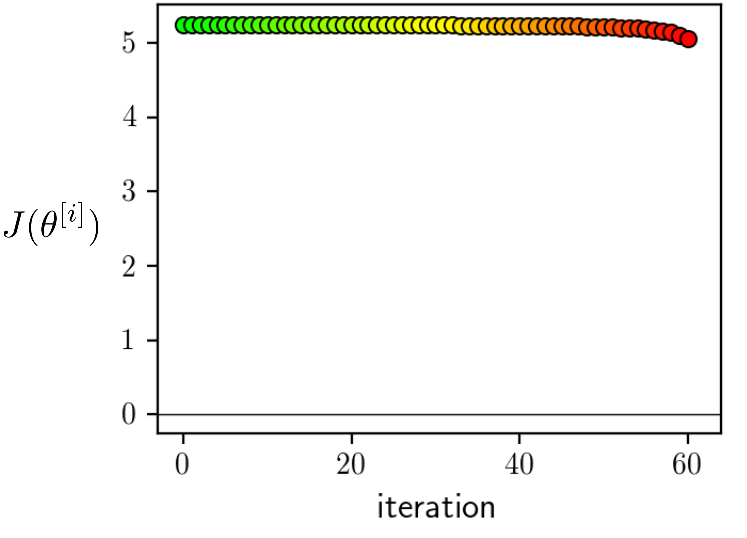




Saddle points and plateaus (3/3)

- Gradient descent **slows down** on plateaus



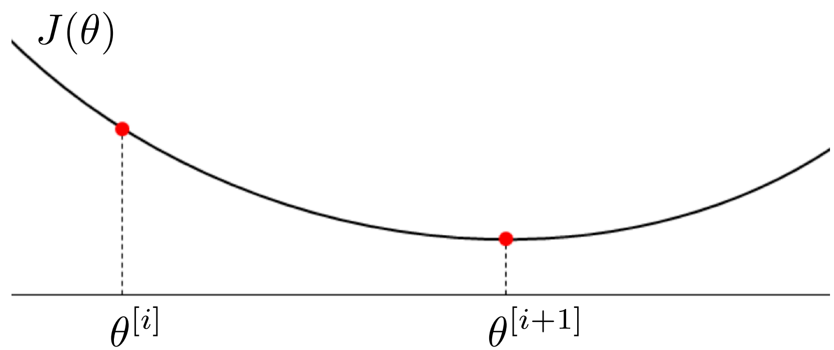


Normalized gradient descent (1/5)

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

Step-size

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



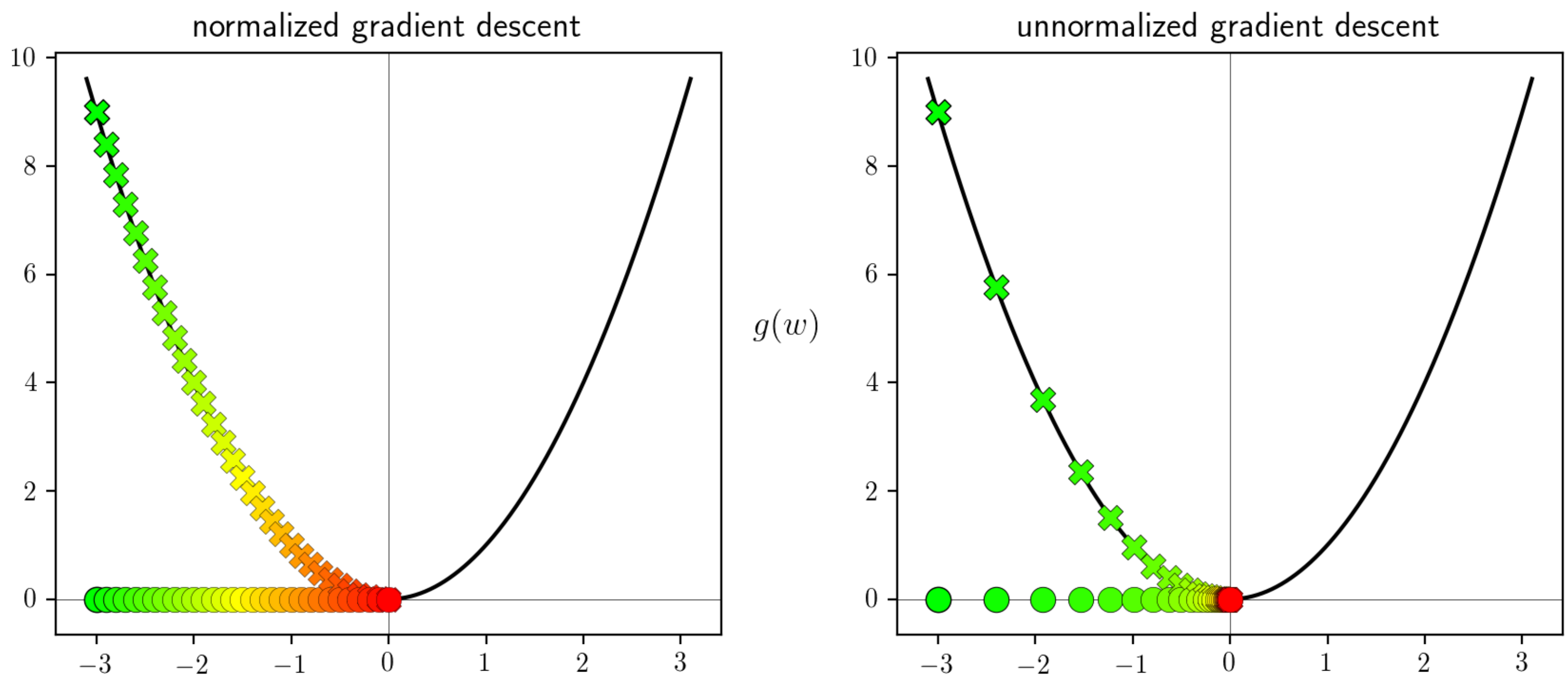
$$\left\|\theta^{[i+1]} - \theta^{[i]}\right\|^2 = \alpha_i$$

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.

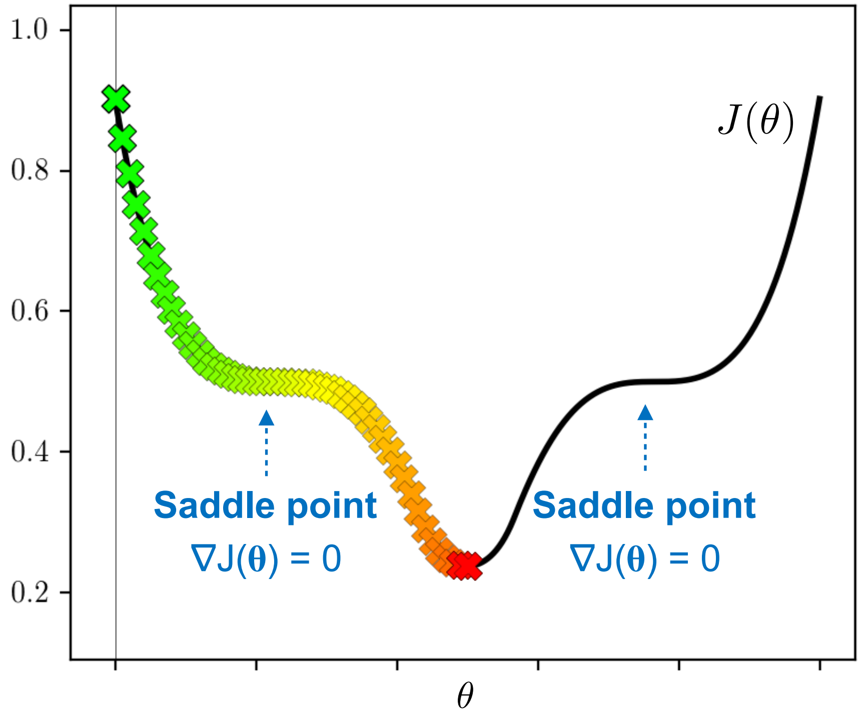
Normalized gradient descent (2/5)

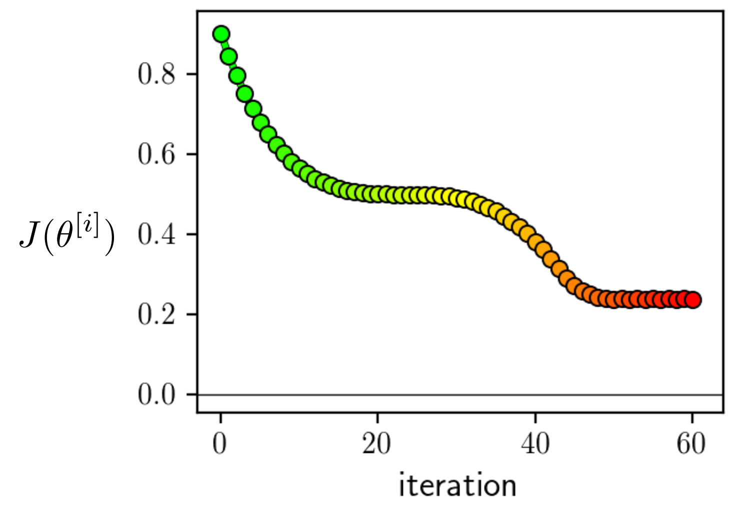
- Gradient descent ➜ **Normalized** **vs** **Standard**
  - *Normalized GD performs fixed-length updates*
  - *Standard GD performs (decreasing) variable-length updates*



Normalized gradient descent (3/5)

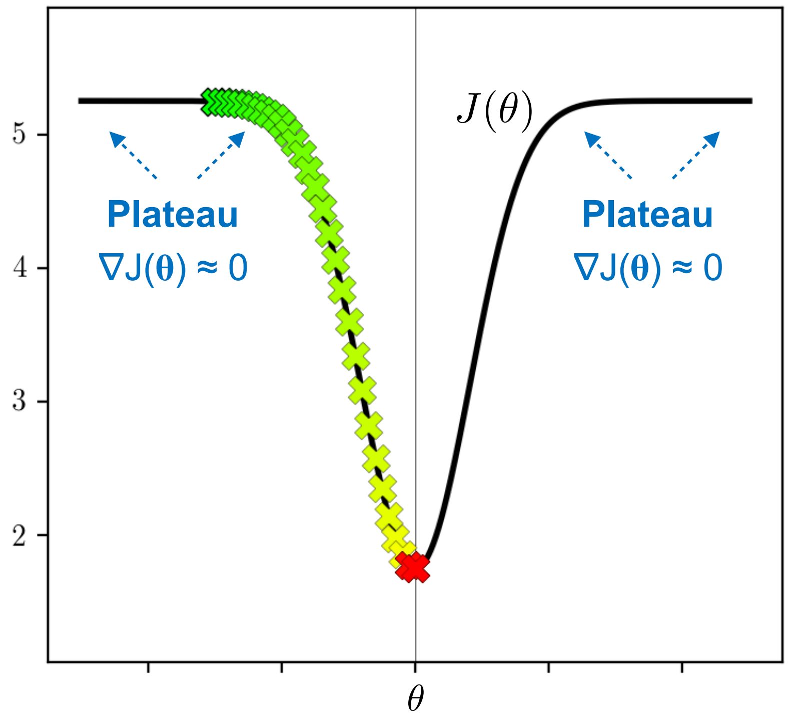
- Normalized gradient descent **goes through** saddle points

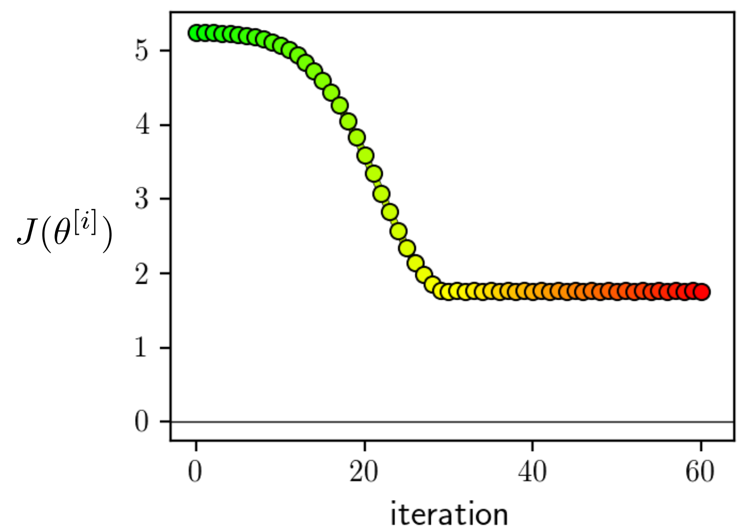




Normalized gradient descent (4/5)

- Normalized gradient descent **goes through** plateaus

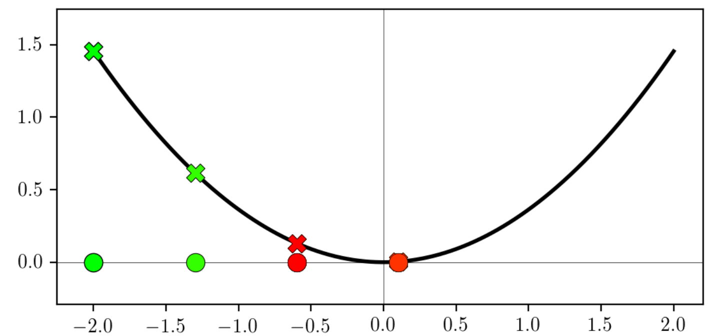


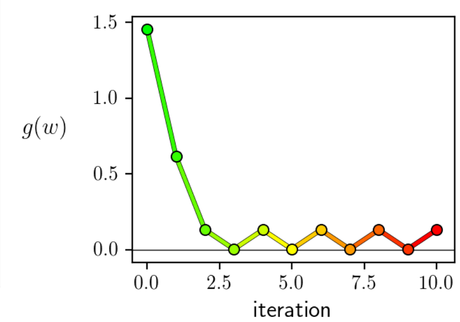


Normalized gradient descent (5/5)

- 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*

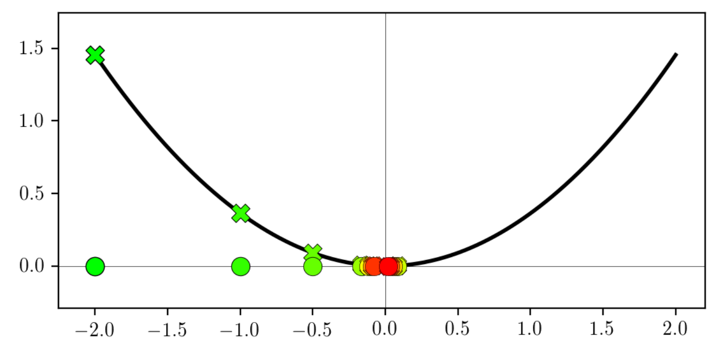
Constant step-size

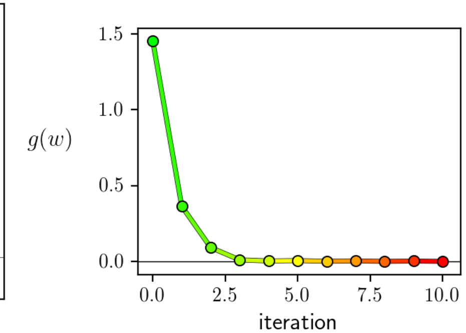




Decreasing step-size

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





State-of-the-art: ADAM

- Modern algorithms for neural network training
  - *First-order optimization + Stochastic + Normalization + Momentum*
  - *Example ➜ **ADAM** (2015)*

$$g^{[i]} = \nabla J^{[i]}\big(\theta^{[i]}\big)$$ Stochastic gradient

$$m^{[i+1]} = \beta_1 m^{[i]} + (1-\beta_1) g^{[i]}$$ Momentum

$$v^{[i+1]} = \beta_2 v^{[i]} + (1-\beta_2)\big(g^{[i]}\big)^2$$ Adaptive estimation

$$\theta^{[i+1]} = \theta^{[i]} - \alpha_i \frac{m^{[i+1]}}{\sqrt{v^{[i+1]}} + \epsilon}$$ Normalization (element-wise)

Quiz

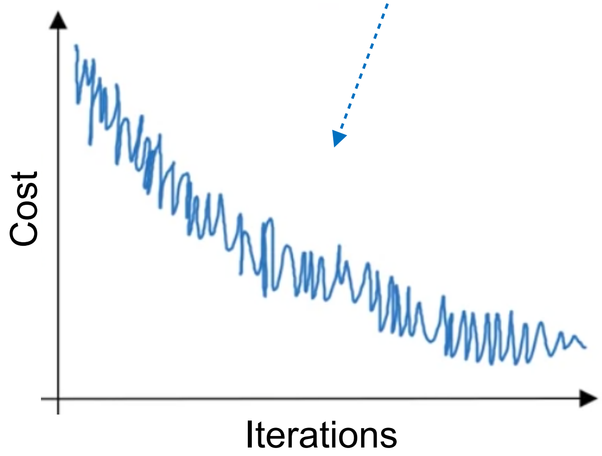
- Assume you tracked the cost function **J(θ)** during training, and the plot versus the number of iterations looks like this.

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.*



What we have seen so far…

- Accelerated gradient descent

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

Adaptive step-size

- Additional hyper-parameters
  - *Mini-batch size*
  - *Optimization (Adagrad, RMSProp, ADAM, …)*
  - *Decaying schedule for step-size*
  - *…*

Numerical optimization

Other best practices

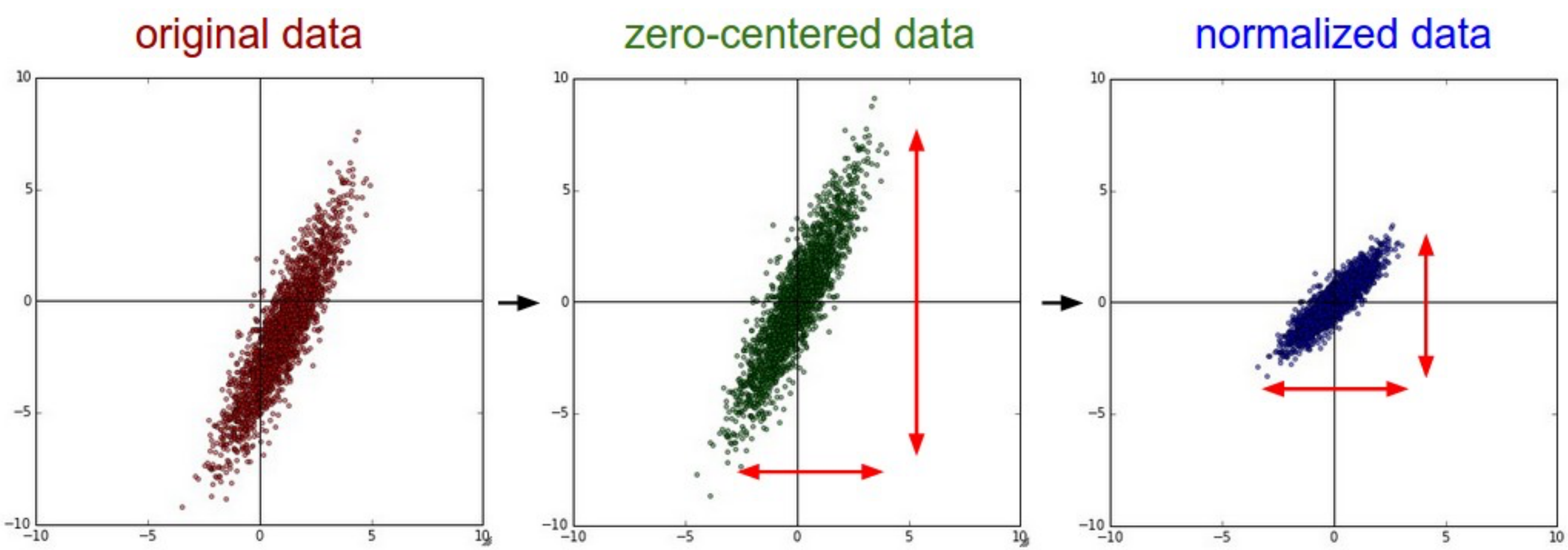
Data preprocessing

Batch normalization

Ensemble of networks

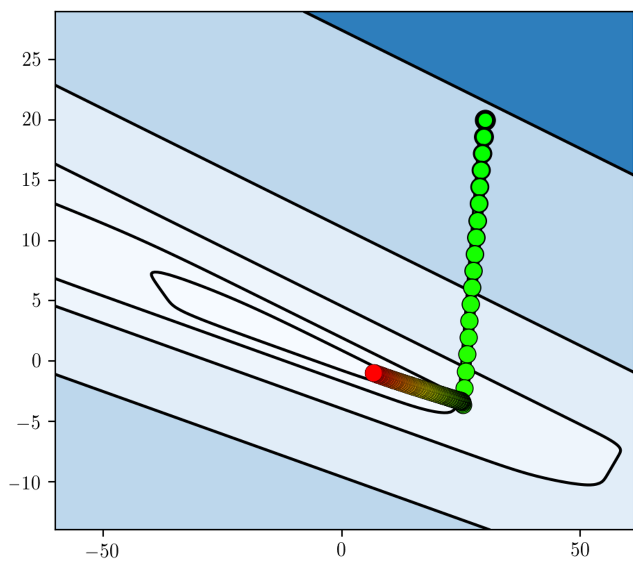
Data preprocessing (1/2)

- **Advice ➜** Normalize data at the network's input
  1) *Subtract the mean across every individual feature in the data*
  2) *Divide each feature by its standard deviation (after mean subtraction)*



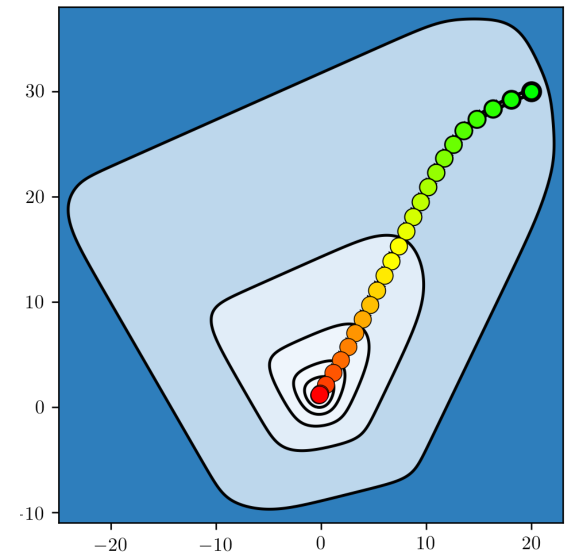
Data preprocessing (2/2)

- Input normalization can help **training go faster**
  - *The cost function is "strongly" elliptical*
  - *Normalization makes the cost function "more circular"*
  - *This transformation speeds up the optimization process*



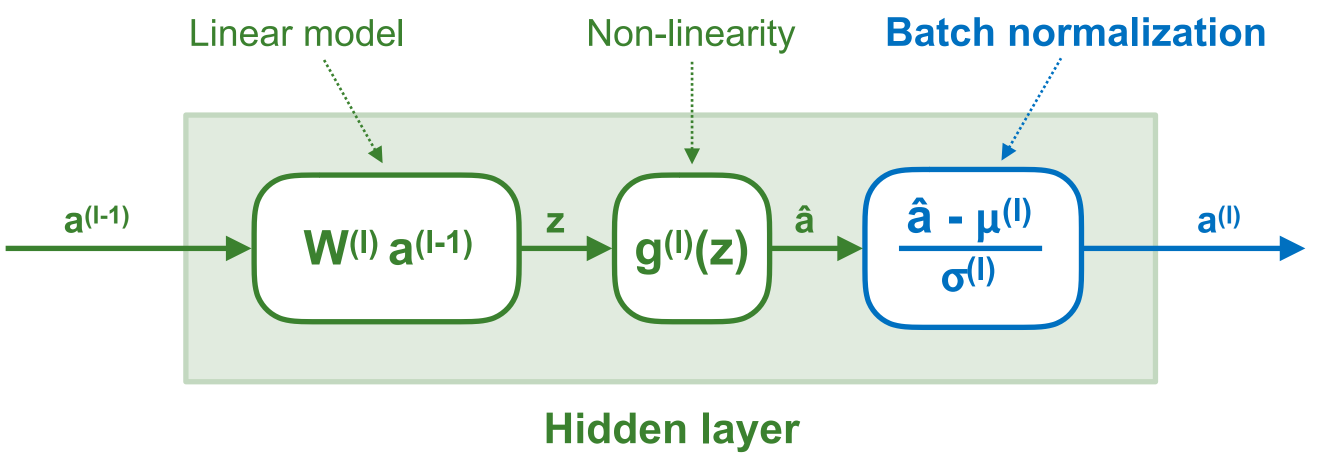
Normalization

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



Batch normalization (1/2)

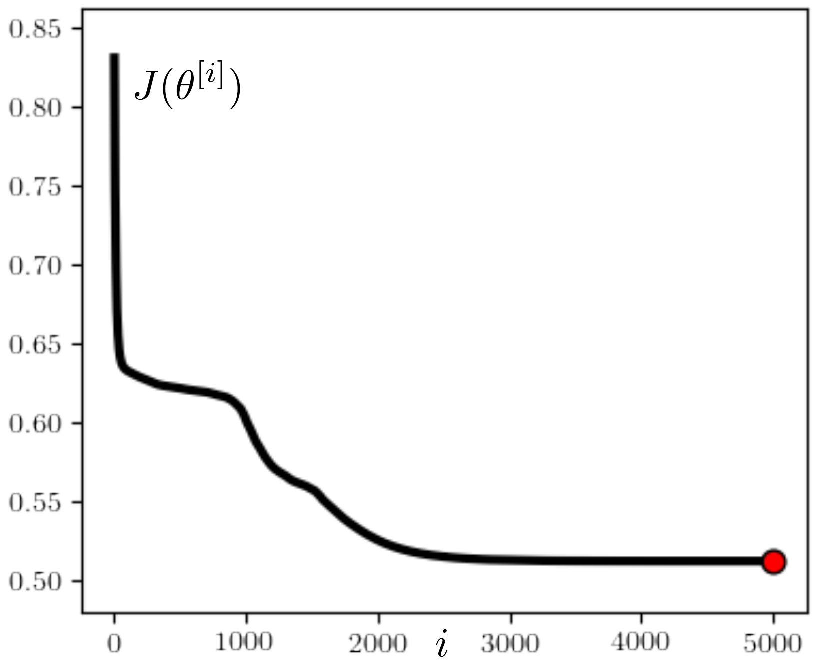
- Normalization can be also applied to hidden layers
  - ***Training*** ➜ *Parameters* $\boldsymbol{\mu}^{(l)}$ and $\boldsymbol{\sigma}^{(l)}$ *are learned*
  - ***Testing*** ➜ *Parameters* $\boldsymbol{\mu}^{(l)}$ and $\boldsymbol{\sigma}^{(l)}$ *are kept fixed*



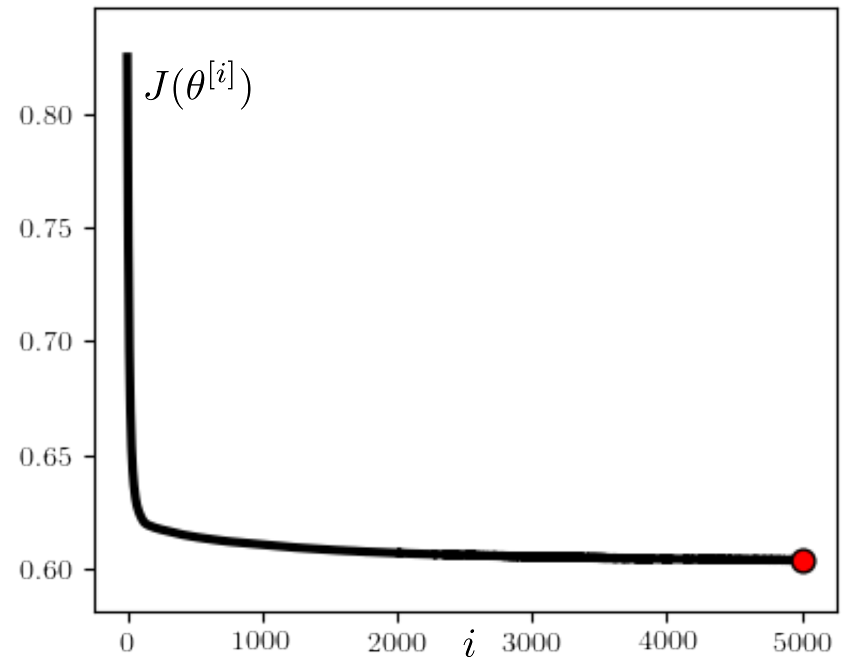
Batch normalization (2/2)

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

Without batch-normalization



With batch-normalization



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.

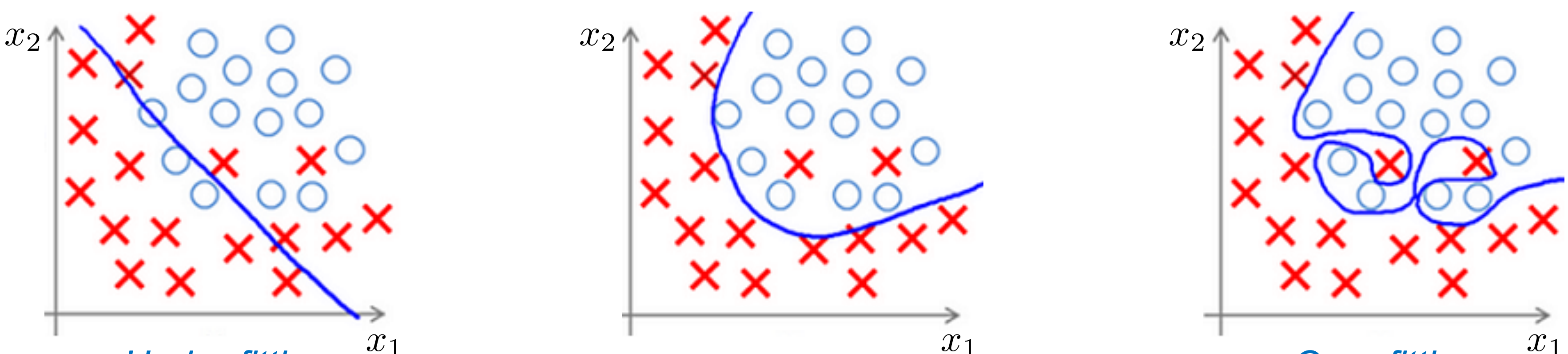
3) ***Different checkpoints of a single model.***
   - If training is very expensive, take different checkpoints of a single network over time. For example, pick a network after a fixed number of epochs. Alternatively, start with a large step-size and a decaying schedule, train the network for a fixed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

Conclusion

Over-fitting
Regularization
Hyper-parameters

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)

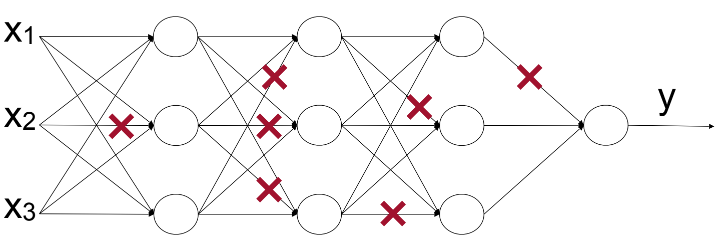
“Just right”

Over-fitting (high variance)

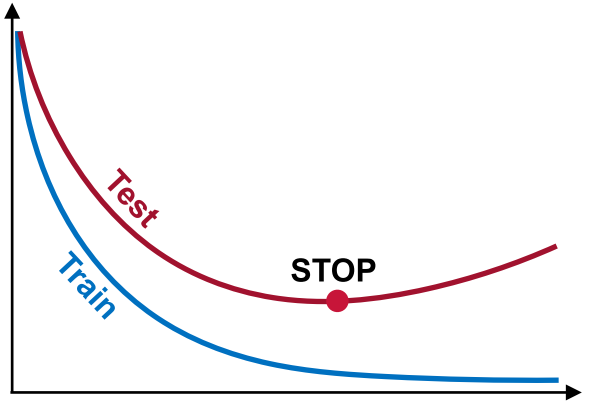
Regularization

- **Effective ways to reduce overfitting**

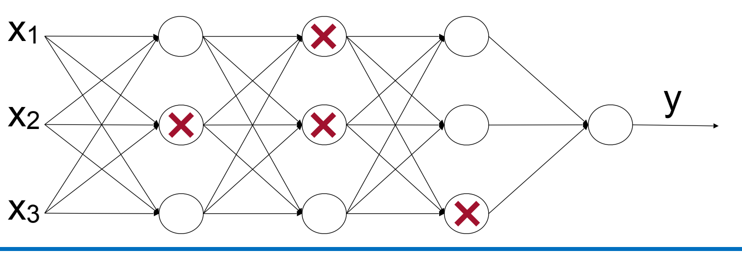
Norm penalization



Early stopping

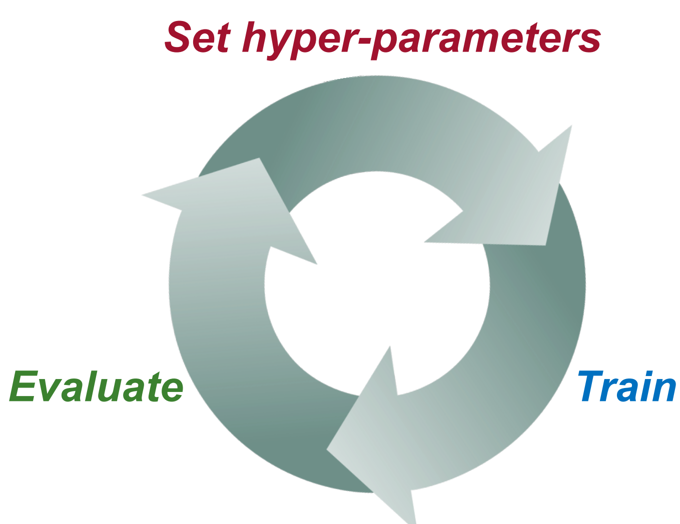


Dropout



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
1600	330
2400	369
1416	232
3000	540
1985	300
1534	315
1427	199

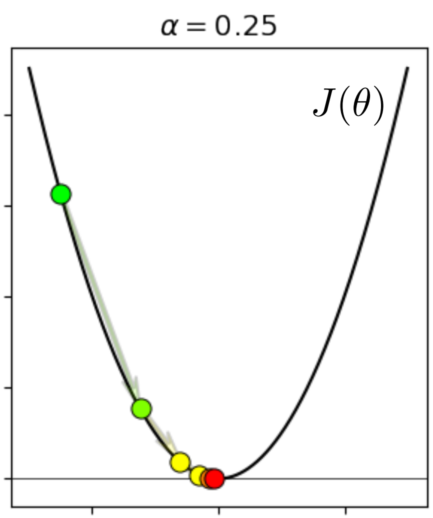
Train set (rows 2104–1416)

Valid set (rows 3000–1985)

Test set (rows 1534–1427)

Optimization

- **Accelerated gradient descent** for neural net training
  - *The choice of **step-size** is **still** critical to ensure fast convergence*



Current solution

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

Step-size

Gradient in current solution

Updated solution