



# Deep Forward Networks - Optimization

Thursday  
08h00-09h00

Géraldine Schaller, Matthew Vowels, Bern Winter School on Machine Learning 2024, Muerren

# Machine Learning Definition

*« A computer program is said to learn from experience  $E$  with respect to some class of tasks  $T$  and performance measure  $P$ , if its performance at tasks in  $T$ , as measured by  $P$ , improves with experience  $E$ . »*

Tom M. Mitchell (1997)



how well the algorithm  
performs on the “walking” task

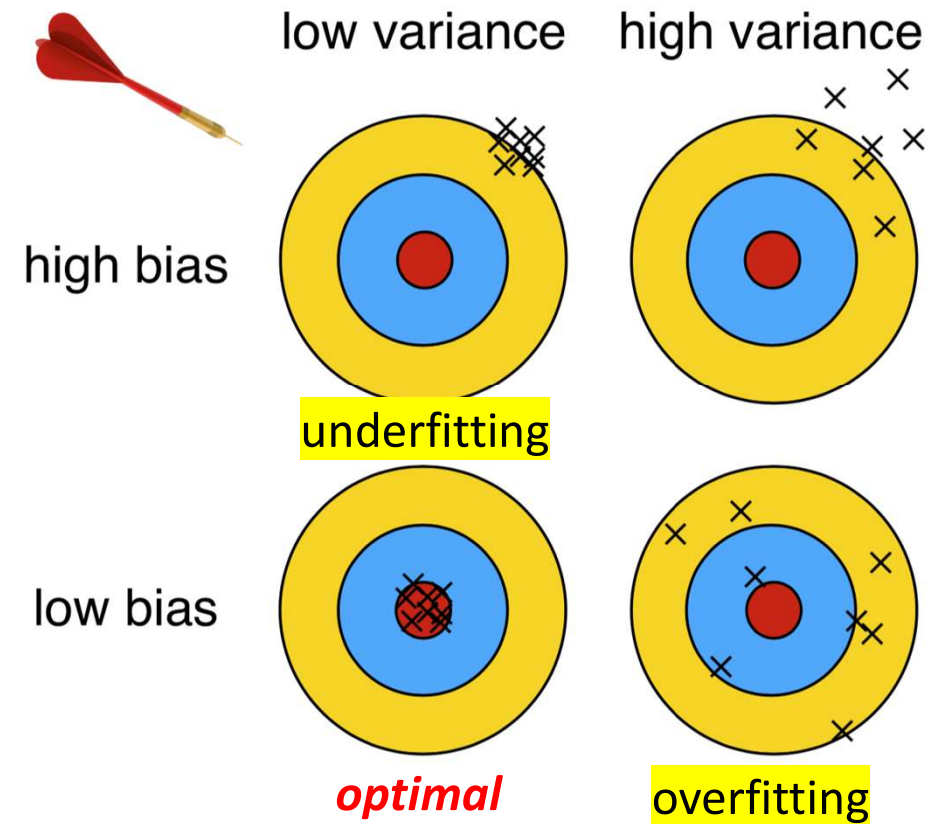
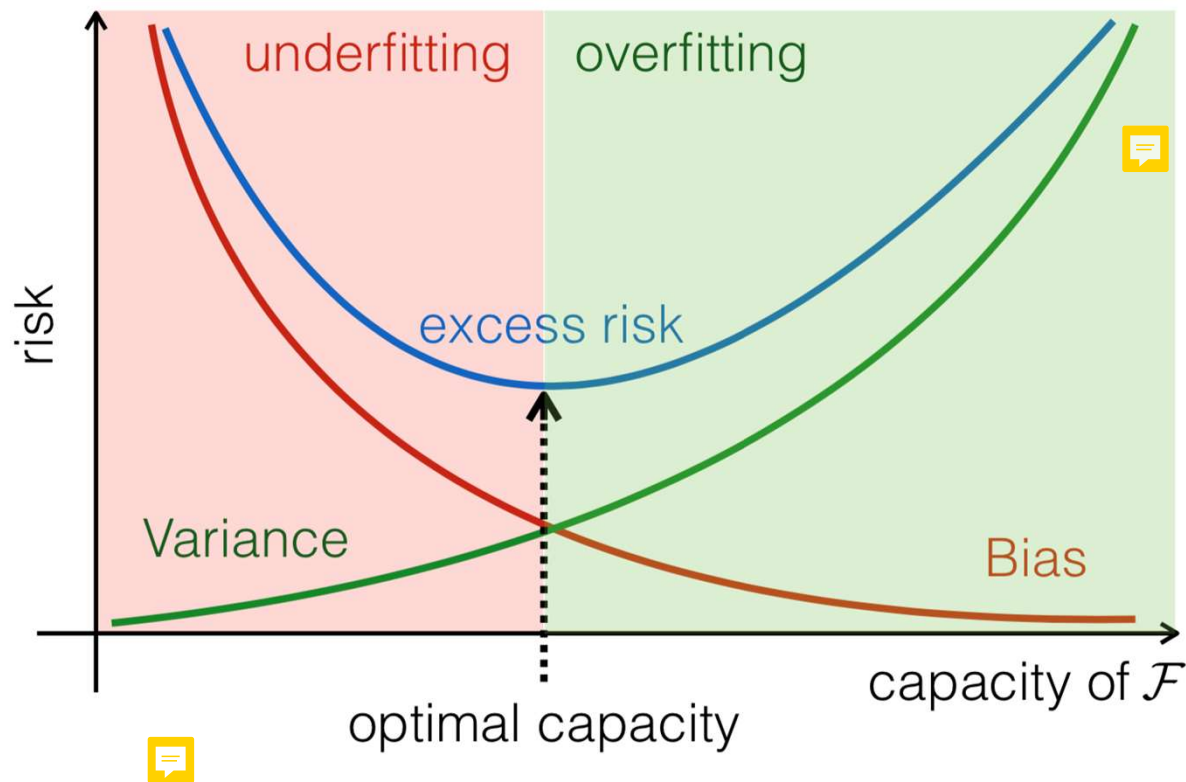
# Performance Measure P

Estimate the ML algorithm performance on task T using the validation set

# Introduction

- To evaluate a ML algorithm, we need a way to measure **how well it performs on the task**
- It is measured **on a separate set** (test set) from what we use to build the function  $f$  (training set)
- **Examples :**
  - Classification accuracy (portion of correct answers)
  - Error rate (portion of incorrect answers)
  - Regression accuracy (e.g. least squares errors)

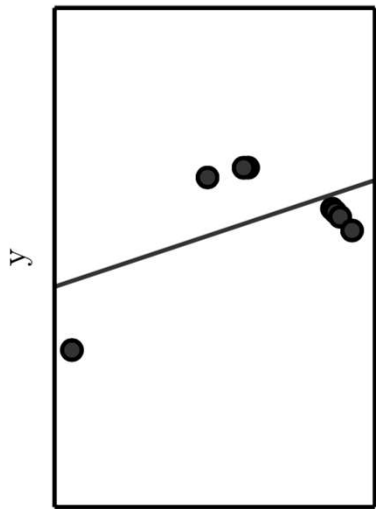
# Bias and Variance - Overfitting and Underfitting





# Overfitting and Underfitting

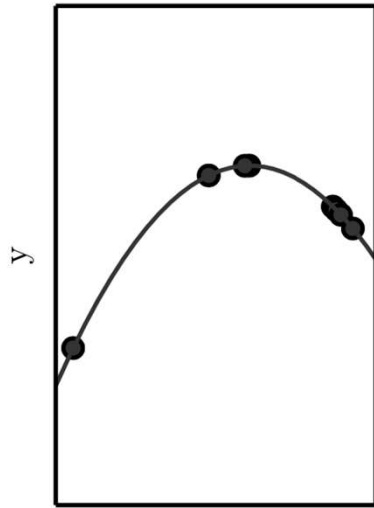
Underfitting



$x_0$

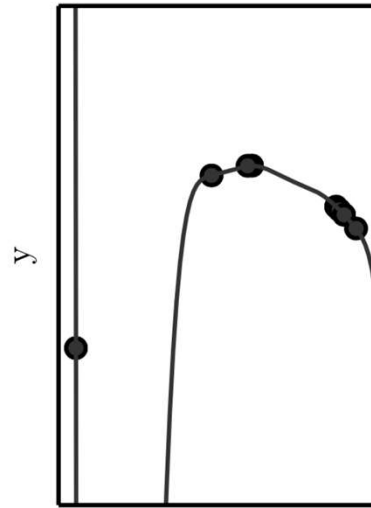
*High bias*

Appropriate capacity



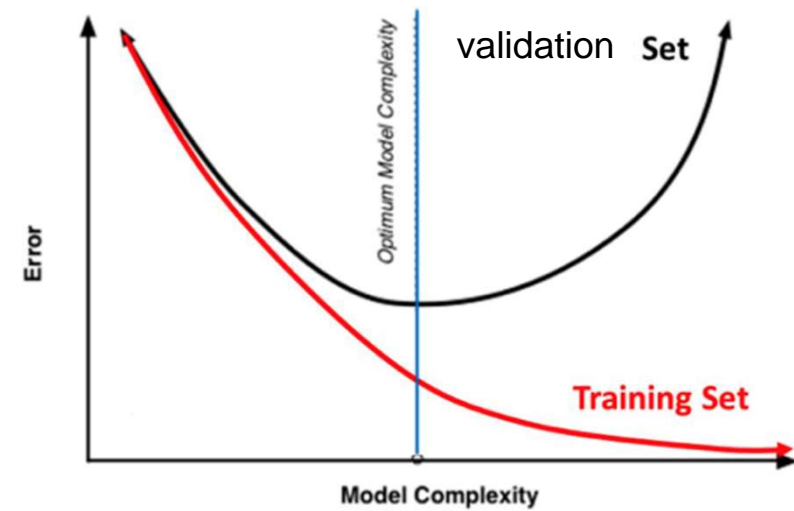
$x_0$

Overfitting



$x_0$

*High variance*



# Optimization

In terms of performance and time





# 1. Performance

- Bias reduction techniques
- Variance reduction techniques



# Case

- You want to find cats in images



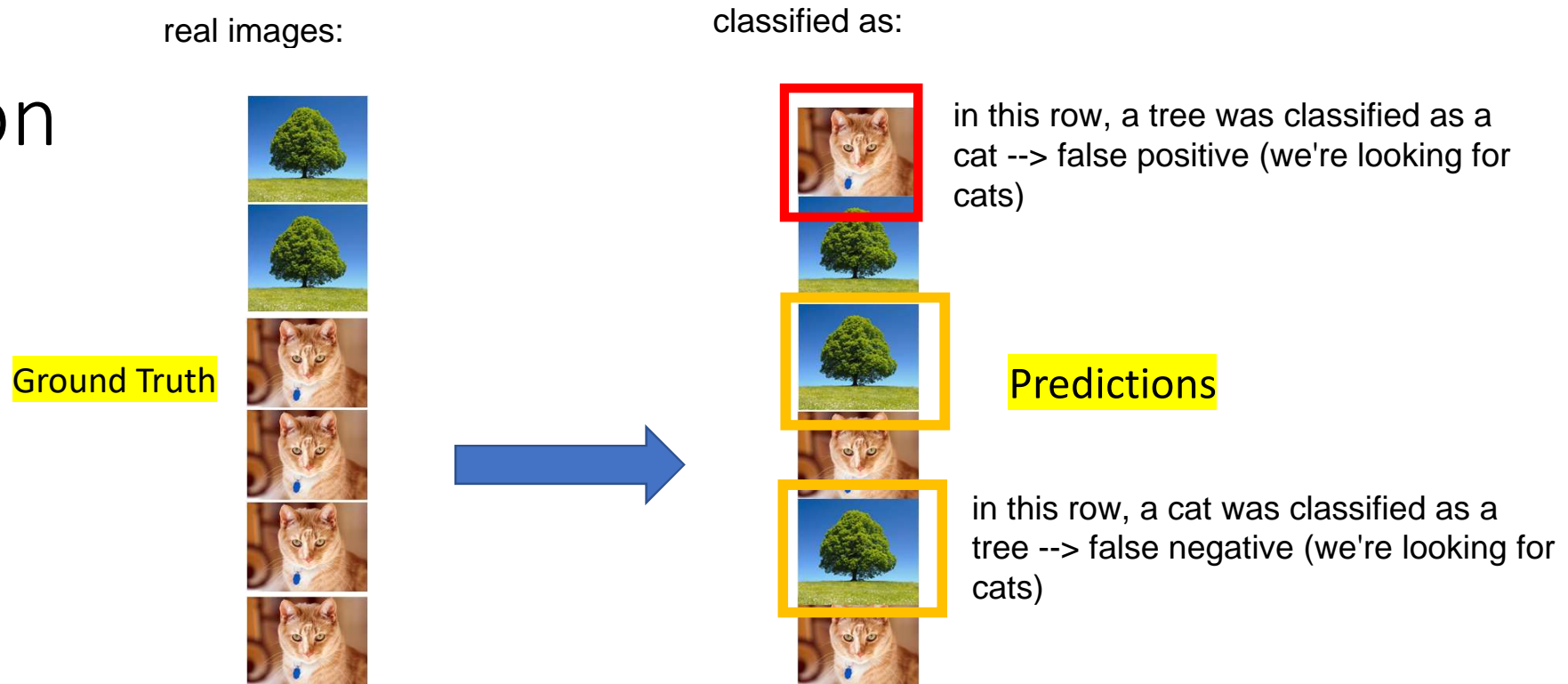
- **Classification error** (portion of wrong answers) used as an **evaluation metric**

Algorithm	Classification error (%)
<b>A</b>	<b>3%</b>
<b>B</b>	<b>5%</b>

➤ *Which one is best ?*



# Evaluation metrics



$$\text{Precision (\%)} = \frac{\text{True positive}}{\text{Number of predicted positive}} \times 100 = \frac{\text{True positive}}{(\text{True positive} + \text{False positive})} \times 100$$

$$\frac{2}{2 + 1} \times 100 = 66\%$$

$$\text{Recall (\%)} = \frac{\text{True positive}}{\text{Number of predicted actually positive}} \times 100 = \frac{\text{True positive}}{(\text{True positive} + \text{False negative})} \times 100$$

See also Sensitivity (same as recall) and Specificity

$$\frac{2}{2 + 2} \times 100 = 50\%$$

# F1-score

- F1-score is a harmonic mean combining p and r

$$\text{F1-Score} = \frac{2}{\frac{1}{p} + \frac{1}{r}}$$

	<u>Precision</u>	<u>Recall</u>	<u>F1Score</u>
Algo 1 →	0.5	0.4	0.444 ✓
Algo 2 →	0.7	0.1	0.175
Algo 3 →	0.02	1.0	0.0392

# First of all, understand your data !

- Carry out manual **error analysis**
  - Look at *mislabeled development set* examples (*do not look at test set*)
  - For example : check by hand 500 pictures (incorrect labels ? Foggy pictures ? Other causes ? )
- Clean up **incorrectly labeled** data
  - Apply same process to your dev and test sets !



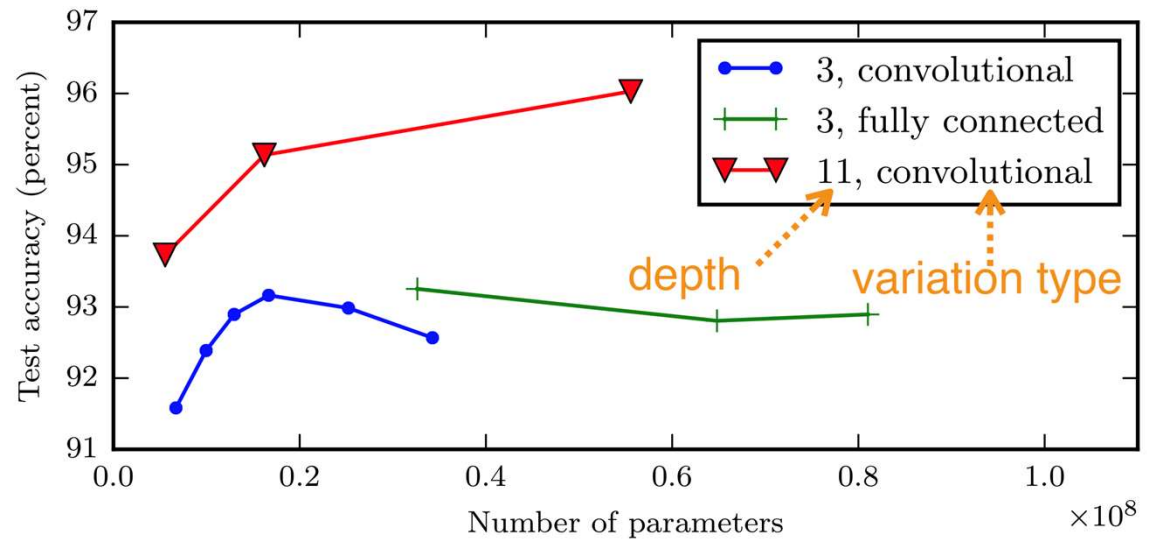
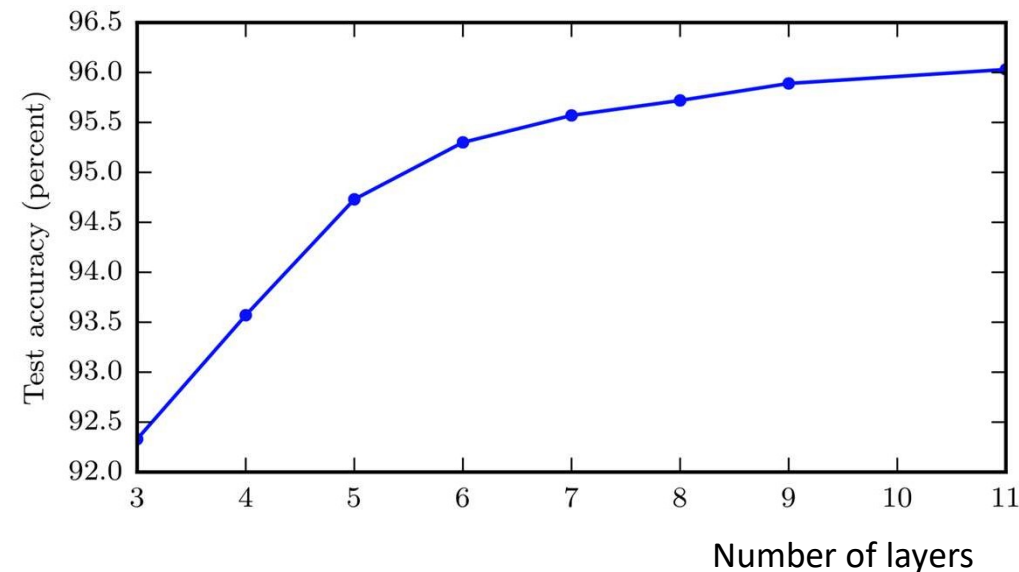


# Bias Reduction techniques

- Hyperparameter tuning
- Model tuning
- Optimization algorithm

# Hyperparameters : number of hidden layers/units

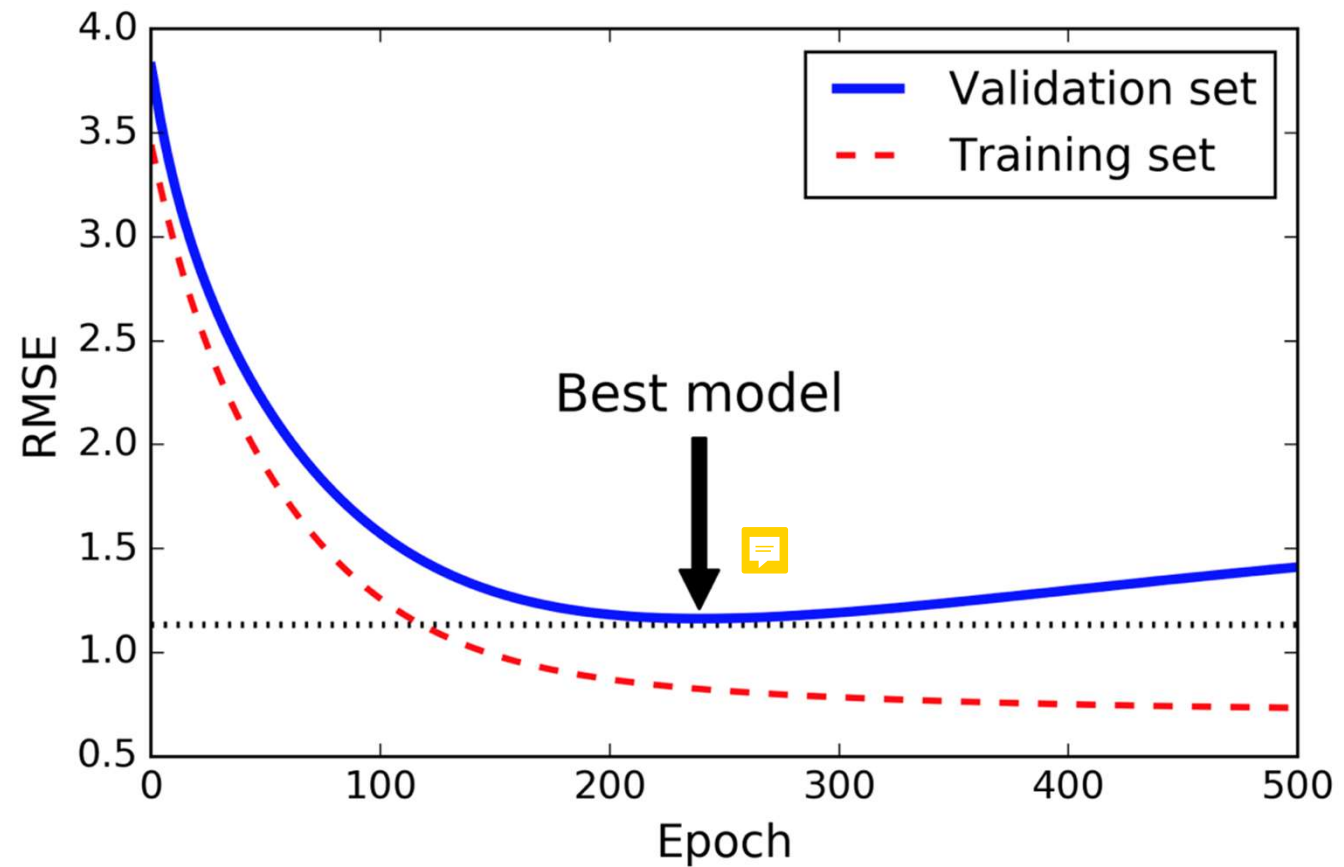
- To go **deeper** helps generalization (but depends on application)
- *better to have many simple layers than few highly complex ones*





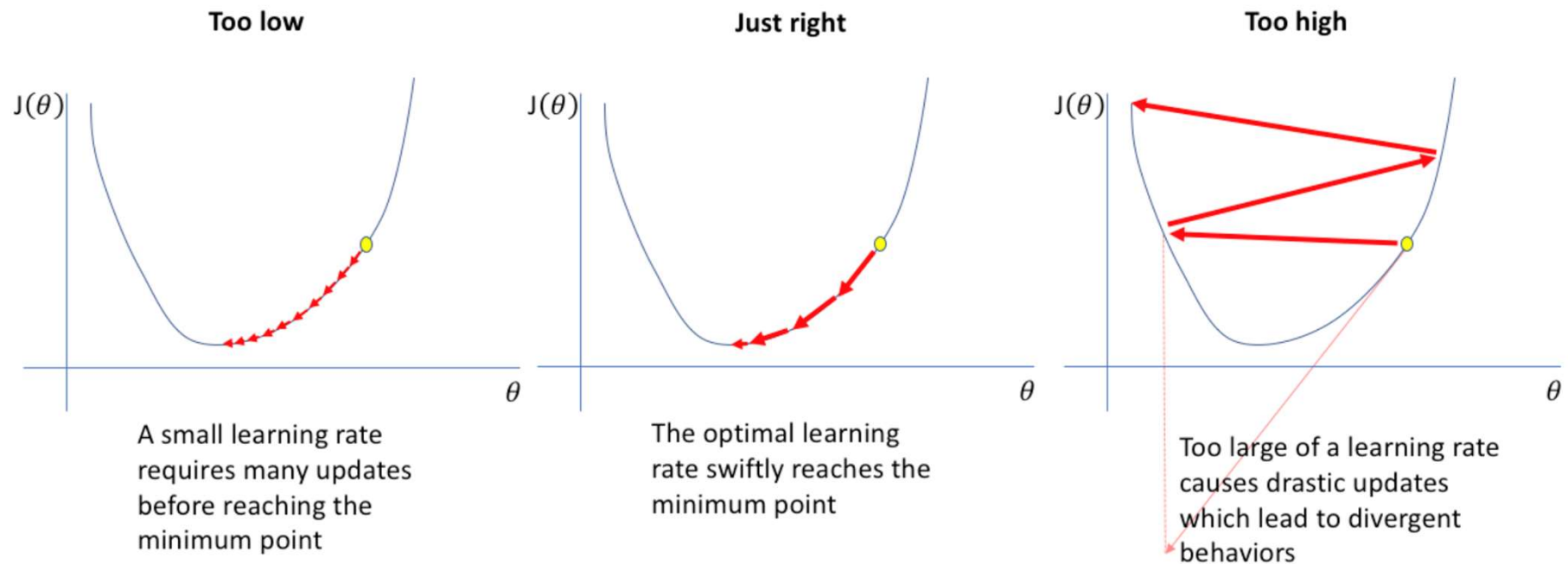
# Hyperparameters : epochs

- Train *longer*



# Hyperparameters : learning rate $\alpha$

- Has a significant impact on the model performance, while being **one of the most difficult parameters** to set



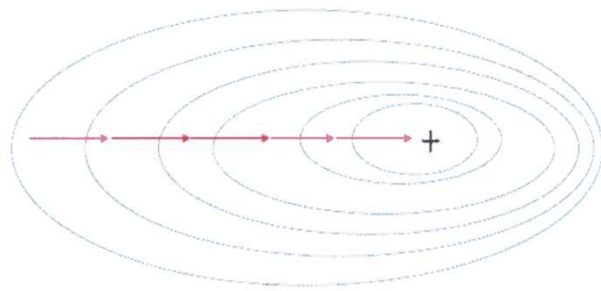
- We can also design a scheduler for the learning rate



# Hyperparameters : batch size

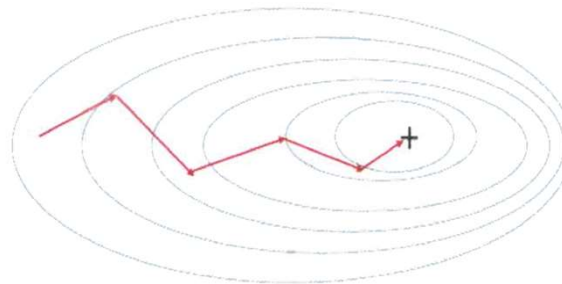
- At each iteration :

Gradient descent (GD)



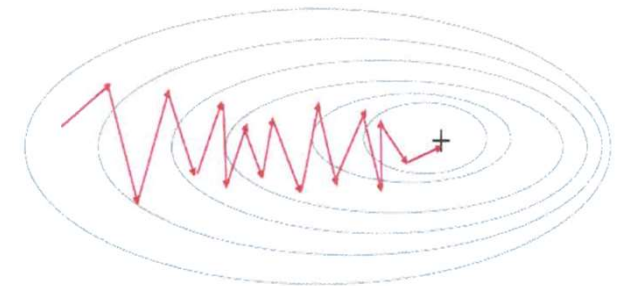
the *whole* training set

Mini-batch gradient descent



a *batch* of samples

Stochastic gradient descent (SGD)



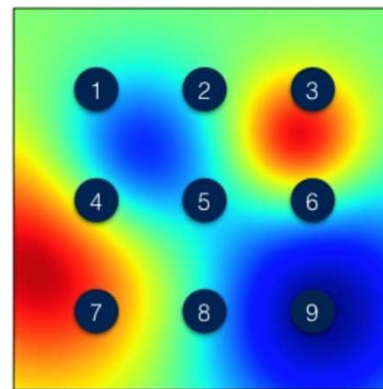
*1 sample*

- Batch size choice *typically 32,64,128,256,512..., etc.*

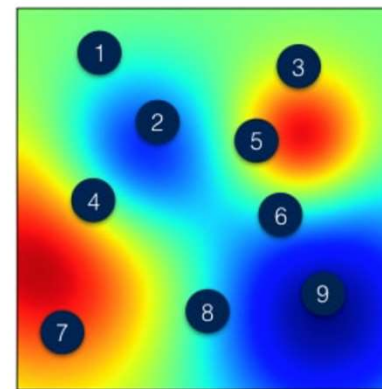
...typically use a power of 2

# Hyperparameters : Global Search

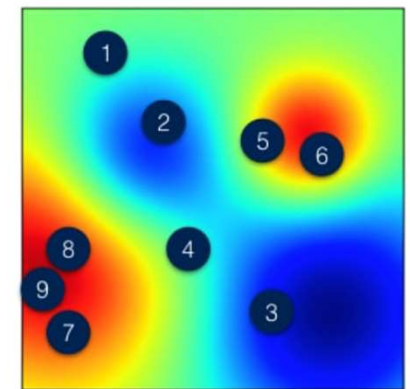
- List :
  - $\alpha$  (**0.0001 – 1**)
  - number of hidden layers
  - number of hidden units
  - learning rate decay
  - mini-batch size
  - ...
- Advice is to use **random values**



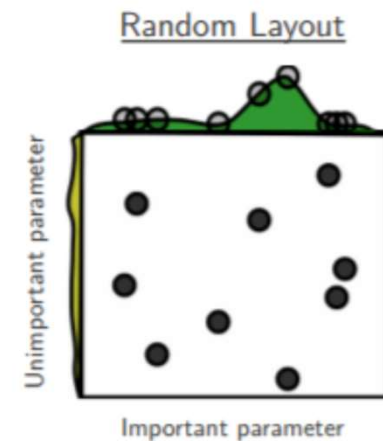
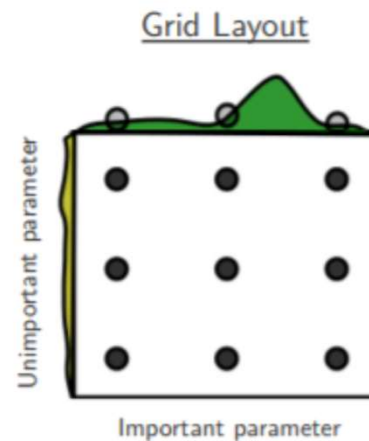
Grid Search



Random Search

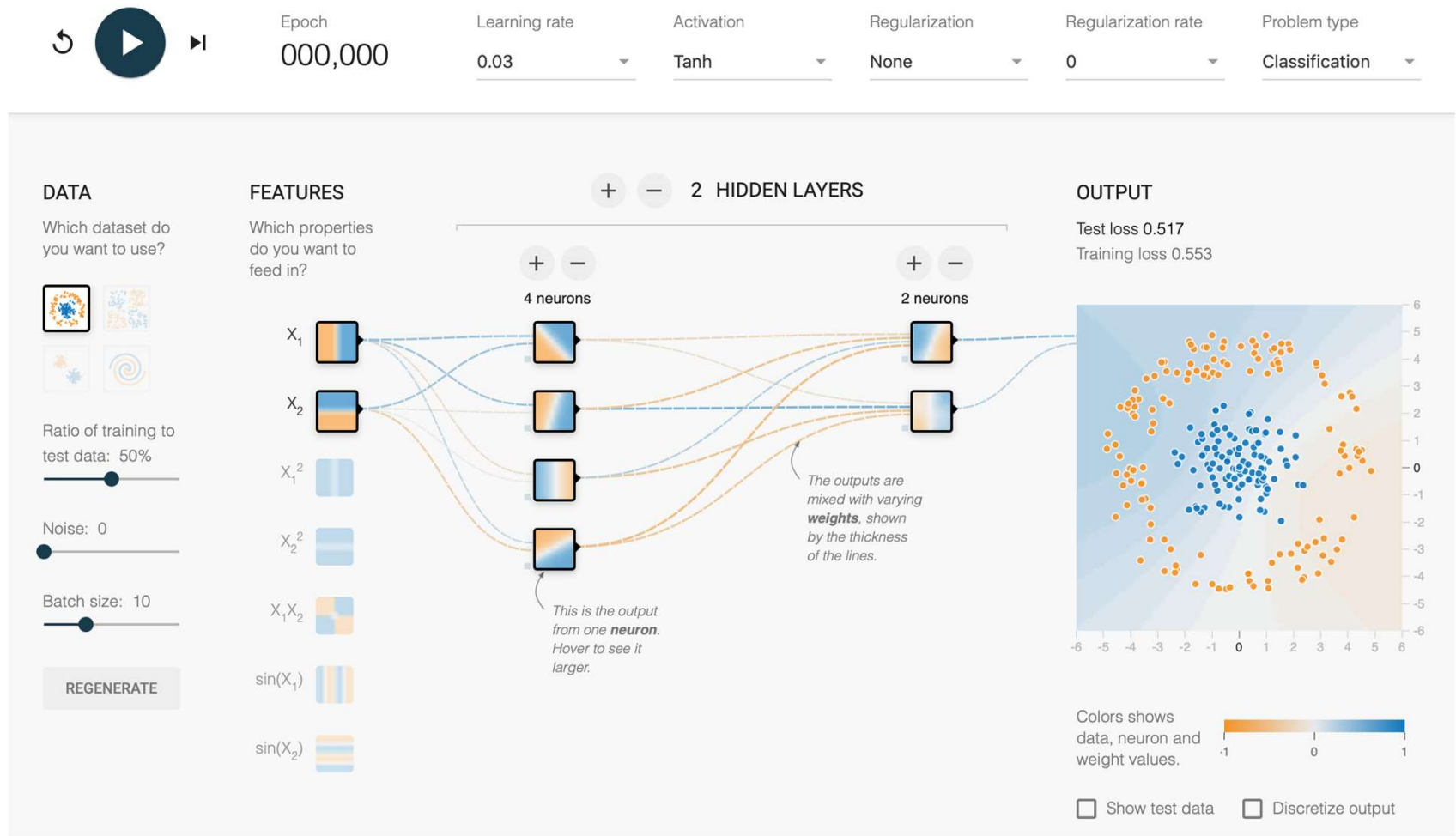


Adaptive Selection




# Hyperparameters : Global Search

## Demo



# Model : Weight initialization

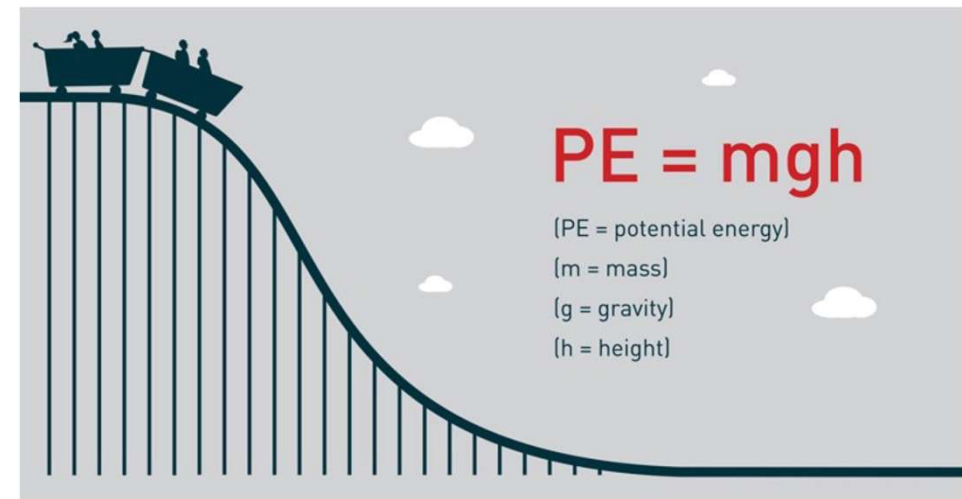
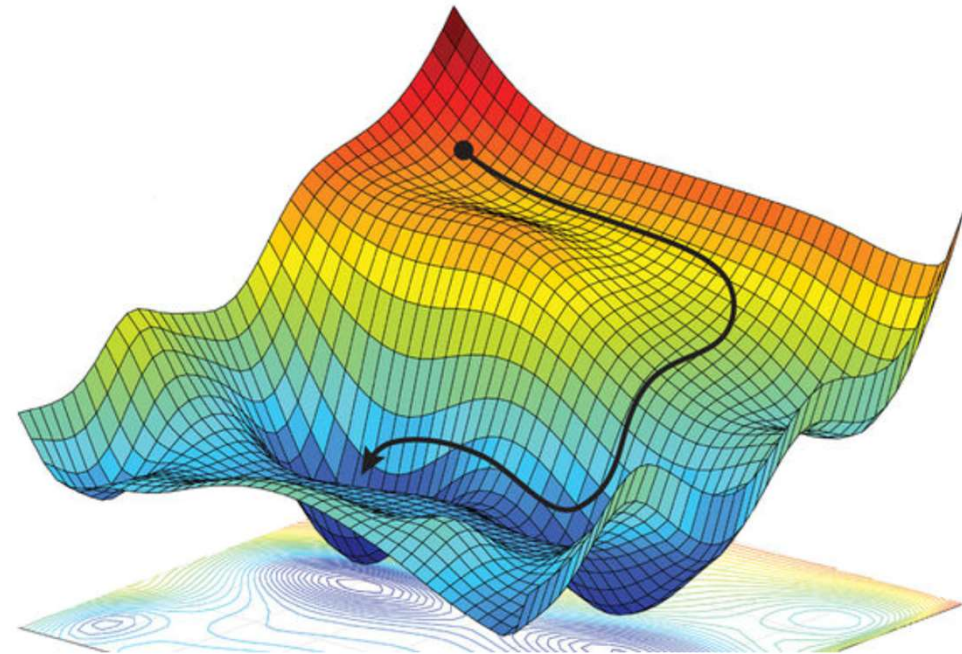
- The initial parameters need to **break the symmetry** between different units 
- Use *random weights* from a Gaussian or Uniform distribution. Alternatively, use *He* or *Xavier weights*
- Another strategy is to initialize weights by **transferring weights** learnt via an unsupervised learning method (method also called **fine-tuning**)



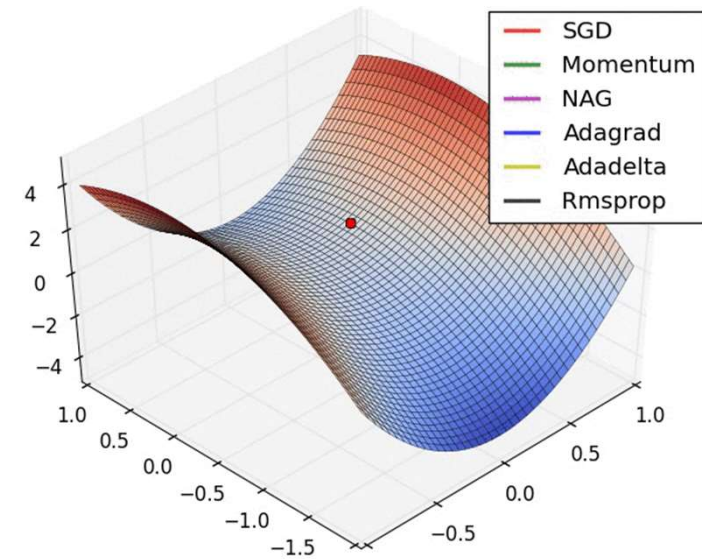
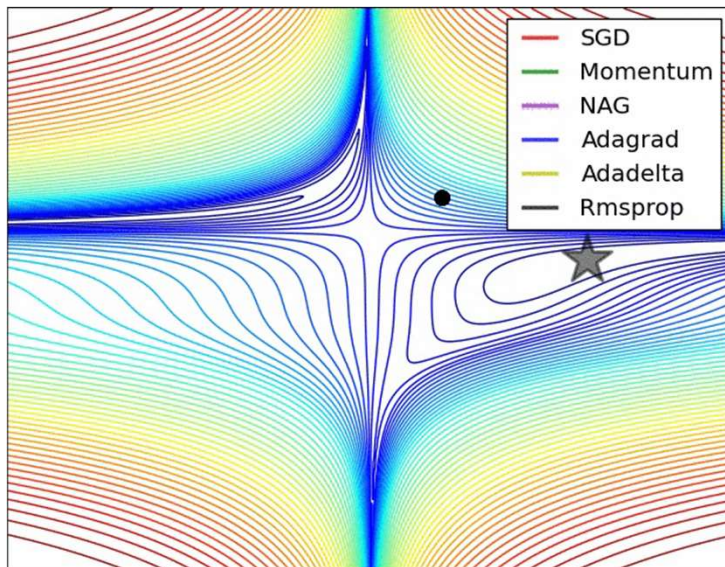


# Optimization algorithm

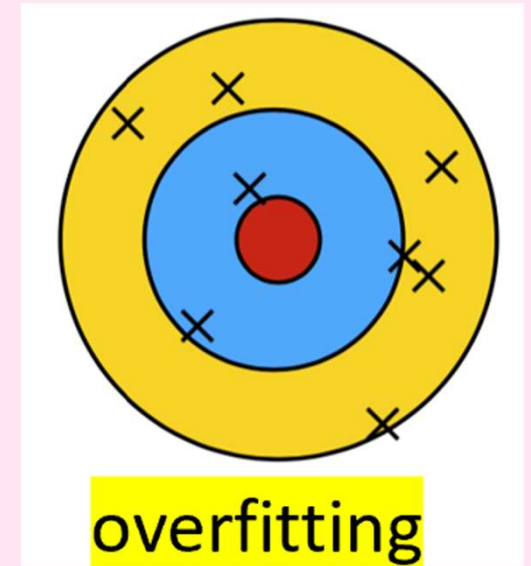
- No consensus on what algorithm performs best
- Most popular choices :
  - SGD (mini-batch gradient descent)
  - SGD + Momentum
  - RMSProp
  - RMSProp + Momentum
  - Adam
- Strategy : *pick one and get familiar* with the tuning



# Gradient Descent Variations



<https://ruder.io/optimizing-gradient-descent/>



# Variance Reduction techniques

- Bigger training set
- Regularization

# Regularization

- Different strategies :
  - Dataset (division, augmentation,...)
  - Model (dropout, L2-, ...)
  - Training (early stopping)
- Use cases : if few data or if model has more than 50 layers (CNN)

# Regularization (*Dataset*) : Division

- Divide the data into a **training**, **validation** and **test** sets
  - **Training set** to define the optimal predictor
  - **Validation set** to choose the capacity
  - **Test set** to evaluate the performance



# Regularization (*Dataset*) : Augmentation

- Apply **realistic transformations** to data to create new synthetic samples, with same label



original



affine  
distortion



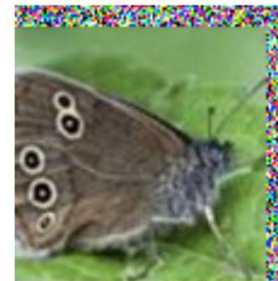
noise



elastic  
deformation



horizontal  
flip



random  
translation

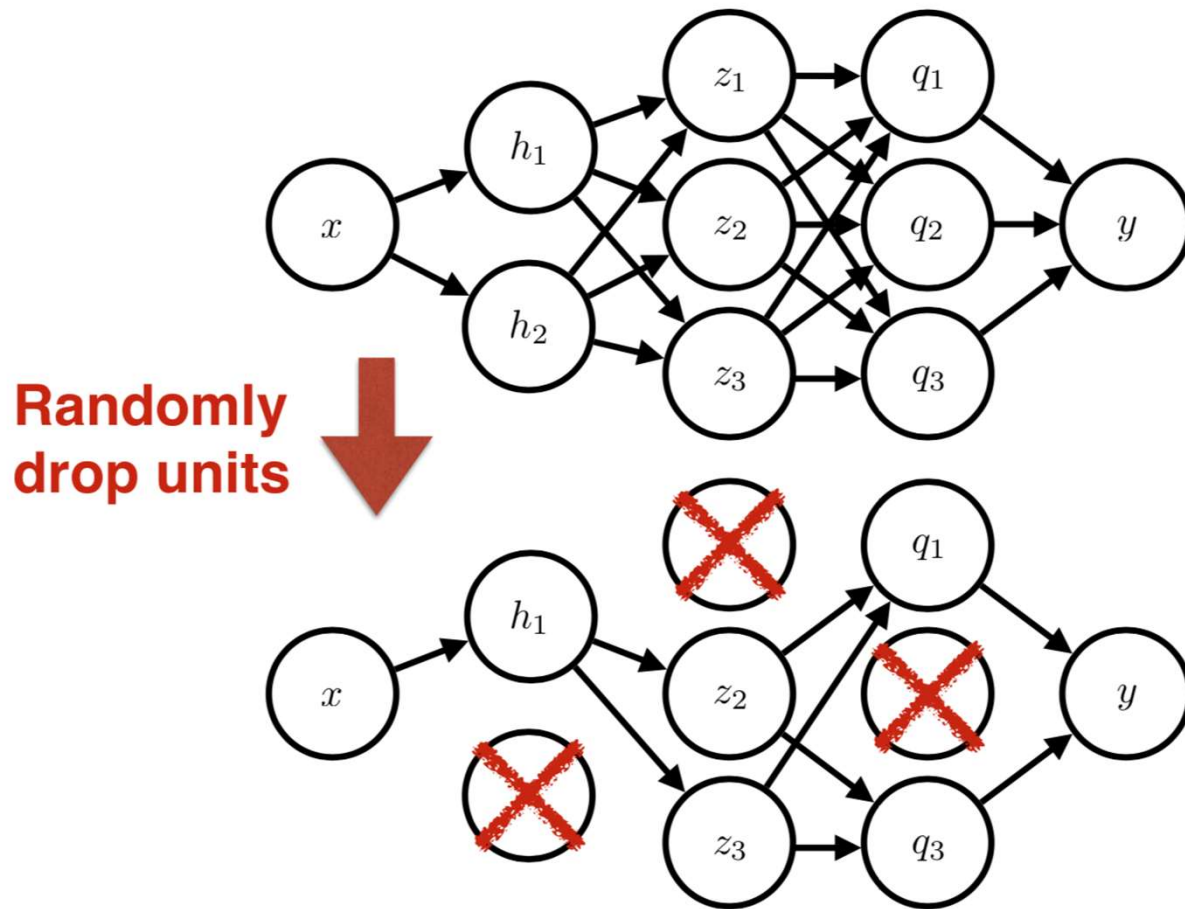


hue shift

- Process also called **jittering**



# Regularization (*Model*) : Dropout



- Apply it both in forward and backward propagations
- *BUT* use it only in the training phase !

# Regularization (*Model*) : L2-regularization

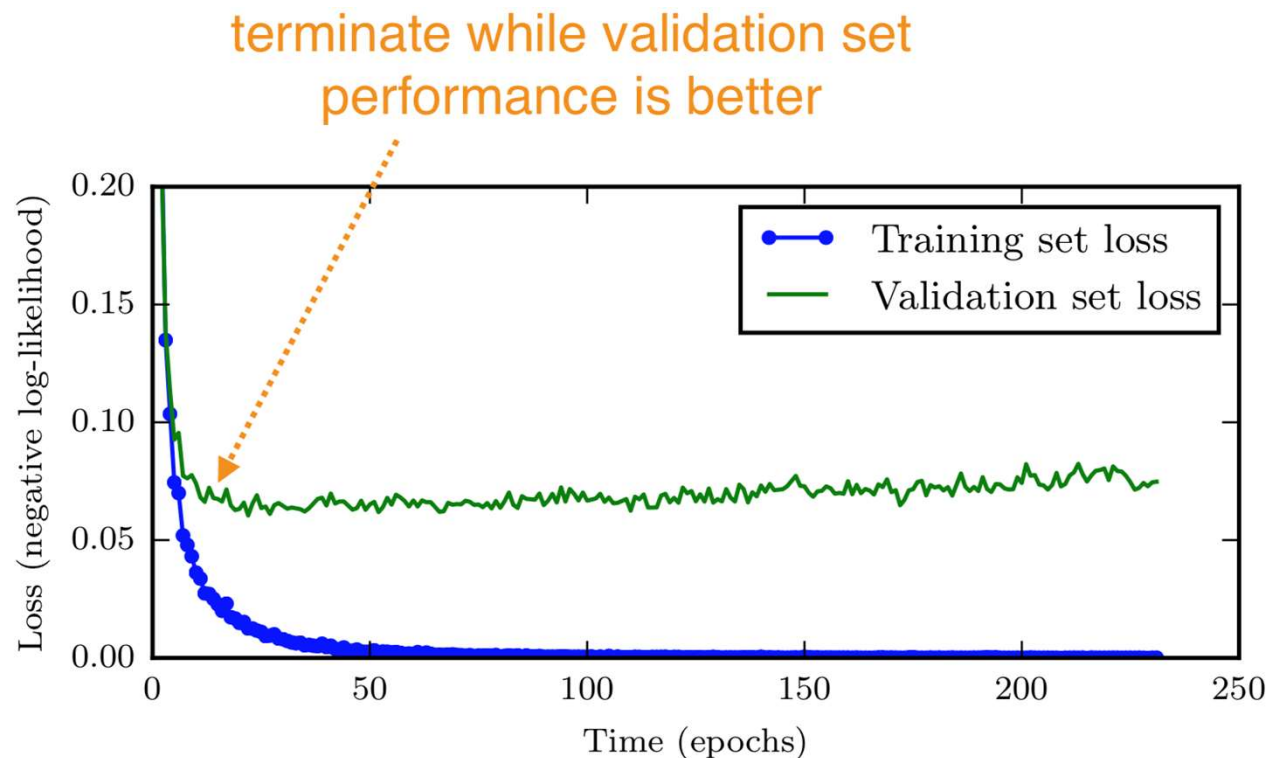
- Modify the cost function (add **soft constraint**) :
  - L2-regularization cost =  $\lambda * \sum(w^2)$

Cost function (regularized) = cost function + L2-regularization cost

- $\lambda$  regulates the complexity/capacity of the predictors. It **smoothens the decision boundary**
  - weights pushed to **smaller values**, preventing any weight from becoming too large and dominating the learning process
- Typical values : logarithmic scale between 0 and 0.1, such as **0.1, 0.001, 0.0001** (tuned using validation set)
  - If  $\lambda$  is too large, it can “oversmooth”, resulting in a model with high bias

# Regularization (*Training*) : Early stopping

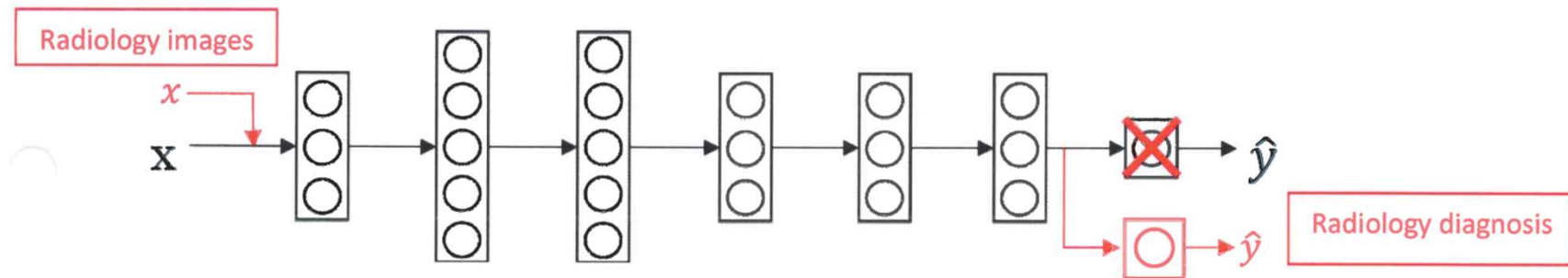
- Limit the number of iterations



*Stop the training when dev set error starts increasing again*

# Transfer Learning

- Use weights that **have been previously trained for another task**
- **Use cases :**
  - Tasks A and B have the same input  $X$
  - A lot more data for Task A than Task B
  - Low level features from Task A could be helpful for Task B



**Pre-training** : training on cat images

**Fine-tuning** : update the weights for radiology

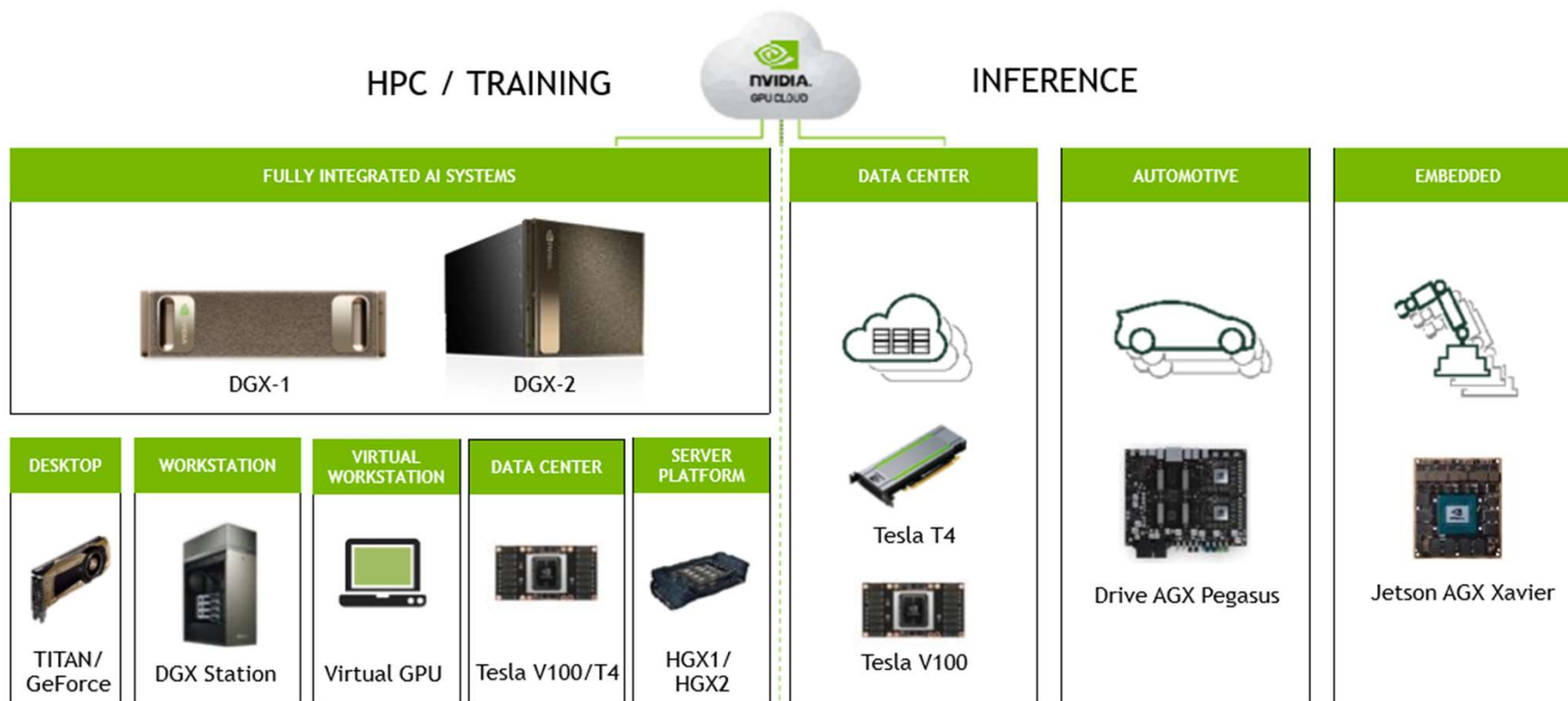
## 2. Time

How to improve time consumption when critical to get results



# Material Acceleration (GPUs)

## END-TO-END PRODUCT FAMILY





# Tutorial / Practical

