ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Deep Learning for Computer Vision

Luigi Di Stefano (luigi.distefano@unibo.it)

Department of Computer Science and Engineering (DISI)
University of Bologna, Italy

Image Classification

Input

Output

Choose one among a set of classes

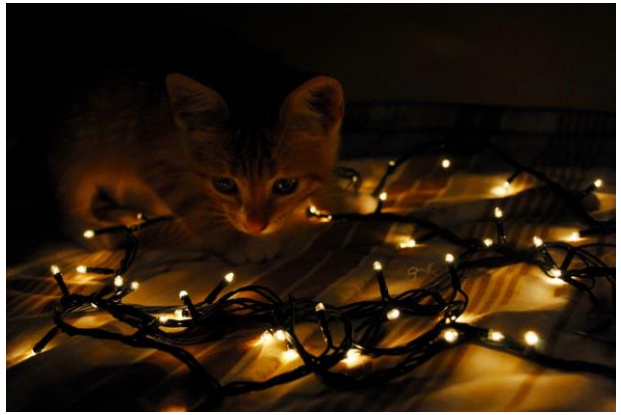
Dog
Cat
Bird
Frog
Person

A very challenging task !

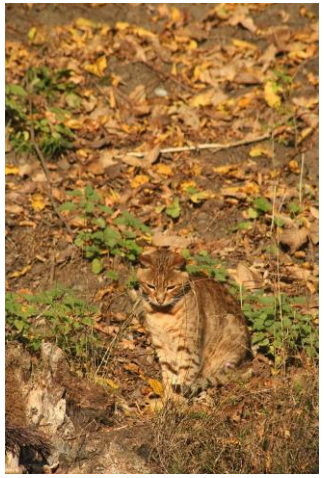
Intra-class variations

Viewpoint variations

Occlusions



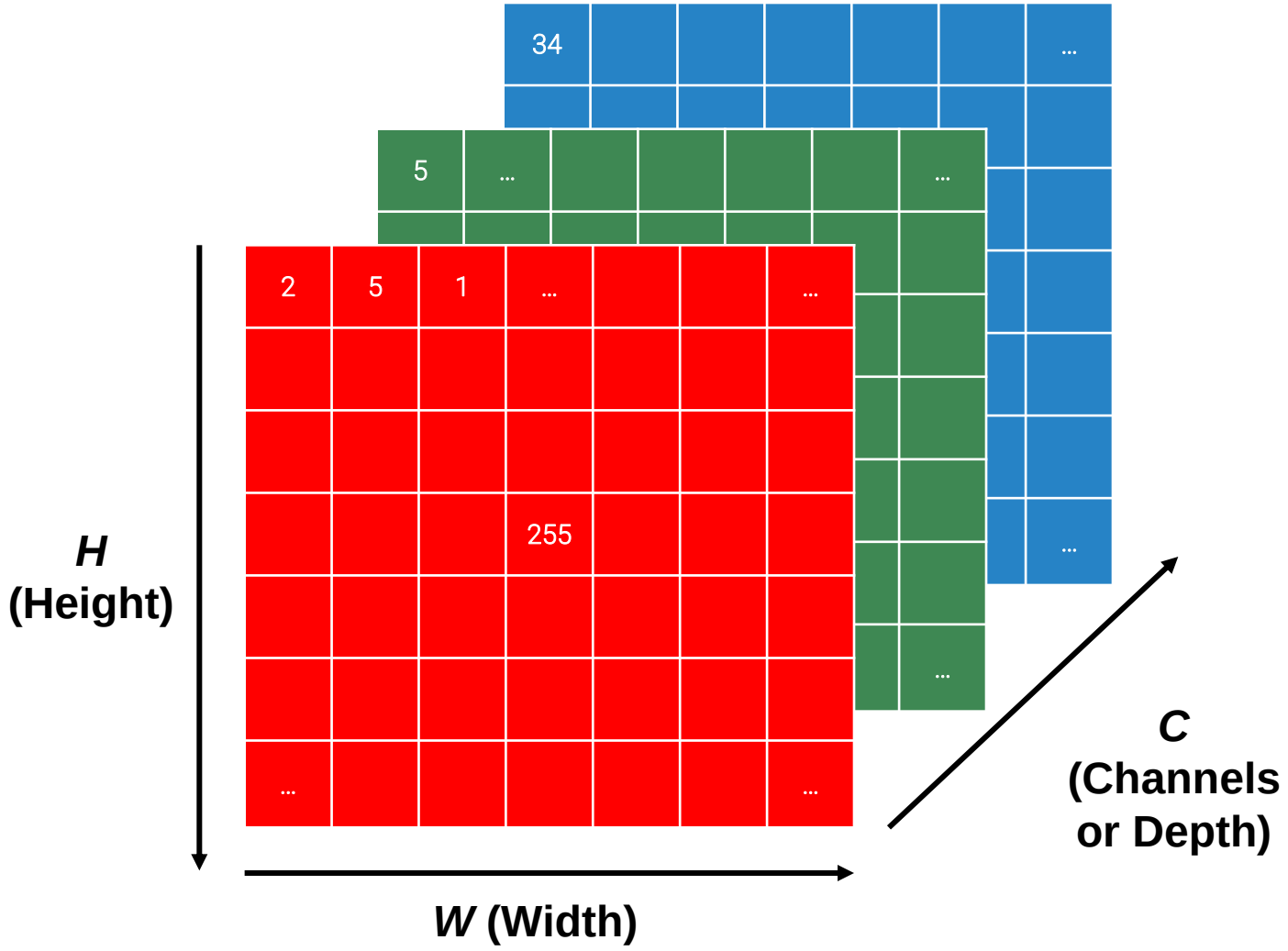
Illumination changes



Background clutter

Weirdness of the world…

Colour images are *tensors* in computers

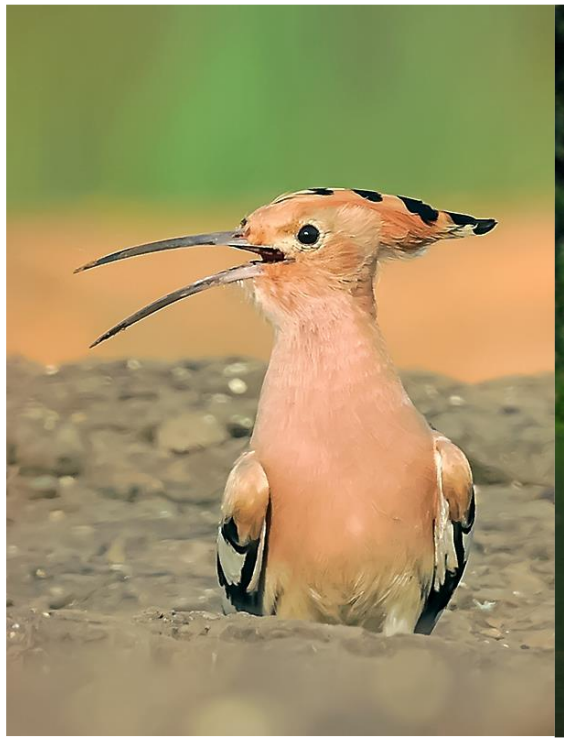
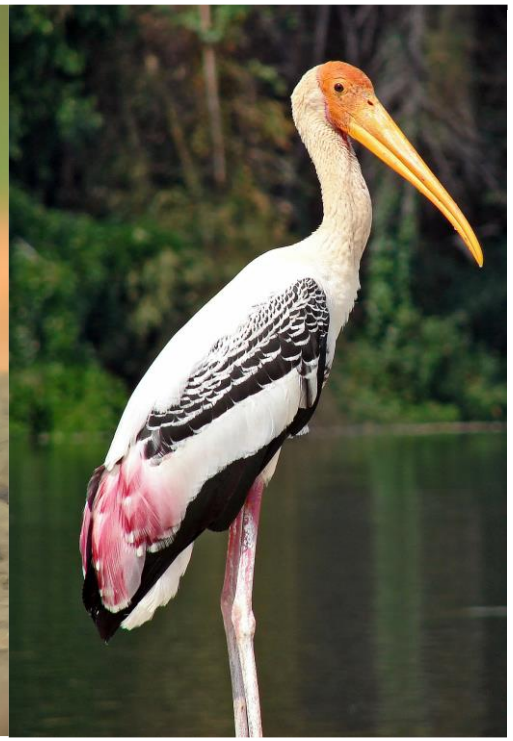


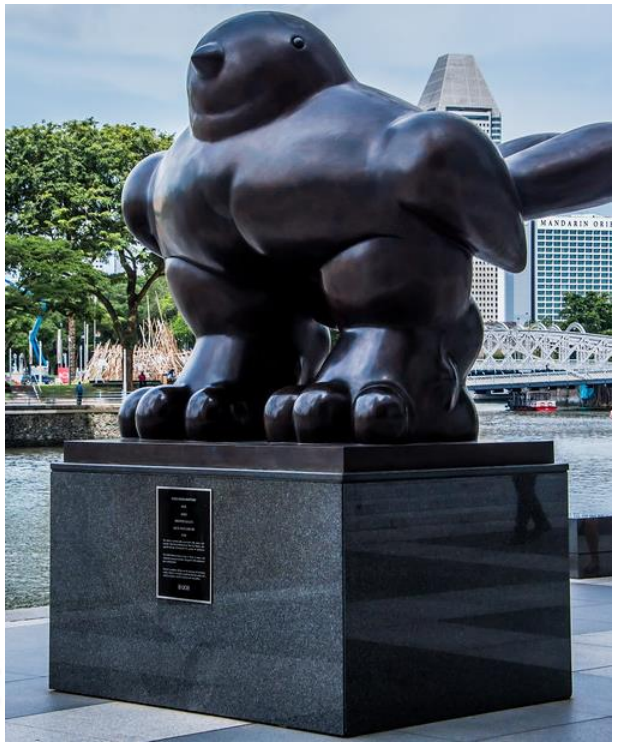
Classes as numbers (labels)

$$f(\quad) = 2$$

0 -> Dog
1 -> Cat
2 -> Bird
3 -> Frog
4 -> Person

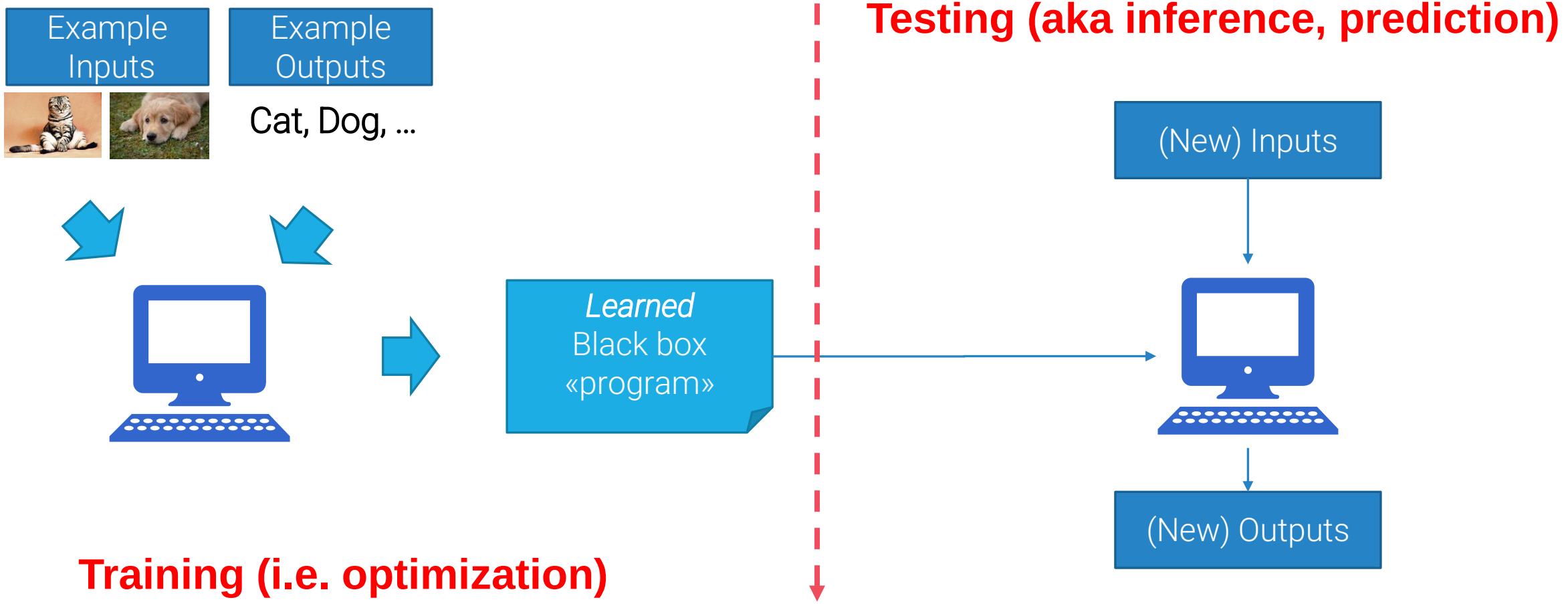
Designing the classification function is impossible



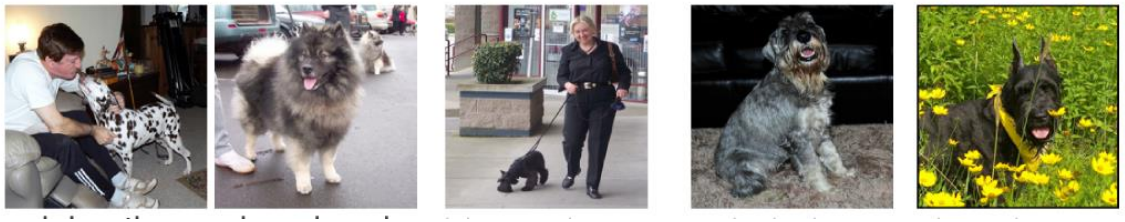
Classical (aka *model-based*) computer vision cannot handle such a high degree of variability, which makes it impossible to "handcraft" the decision rules.

Machine Learning to the rescue



Learning the classification function is possible

Dog



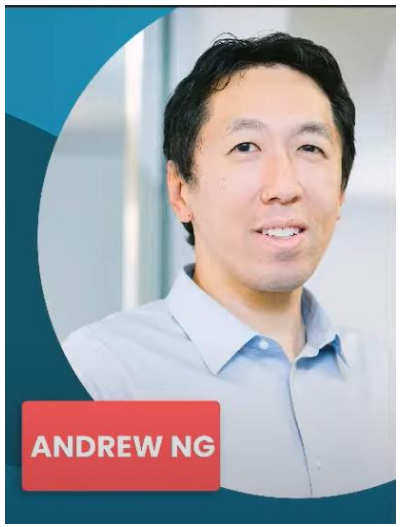
Cat

Bird

$$f\left(\text{  }\right) = 2$$

A data-centric paradigm

- **Data and datasets are crucial in this paradigm, even more important than the learning algorithm.**
- **Large datasets are needed to tackle problems characterized by high-variability.**
- **The quality of data is utterly important (e.g. images must be labelled consistently)**



"If 80% of machine learning is data preparation, we should invest more in it and be more systematic about it"

MLOps

https://www.youtube.com/watch?v=06-AZXmwHjo

Modified NIST (MNIST)

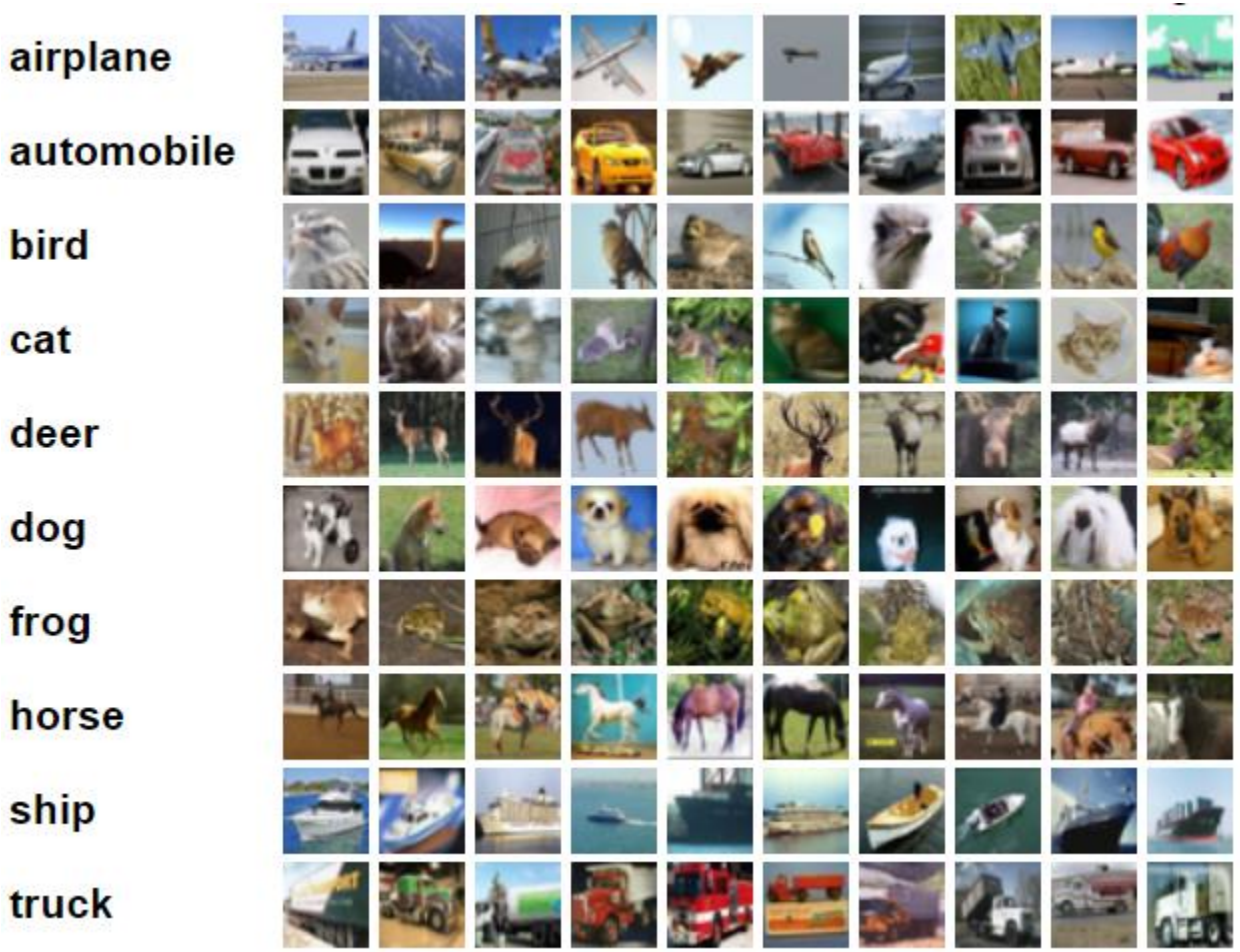
- **10** classes: handwritten digits from 0 to 9
- **50k** training images
- **10k** test images
- **28x28** grayscale images

LeCun @ICLR 2024

MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges

https://www.kaggle.com/datasets/hojjatk/mnist-dataset

CIFAR 10



- **10** classes (see the picture)
- **50k** training images
- **10k** test images
- **32x32** RGB images

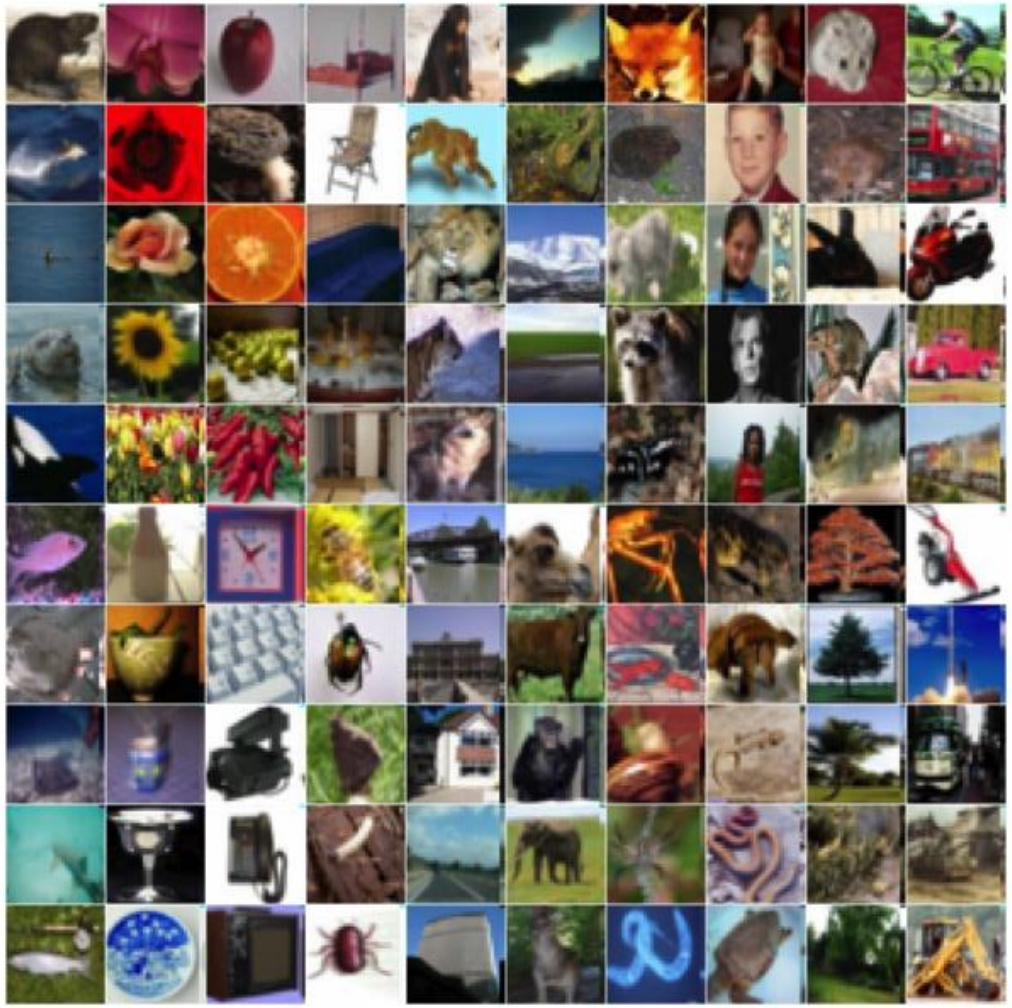
Subset of the 80 million Tiny Images dataset

https://www.cs.toronto.edu/~kriz/cifar.html

https://groups.csail.mit.edu/vision/TinyImages/

Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

CIFAR 100

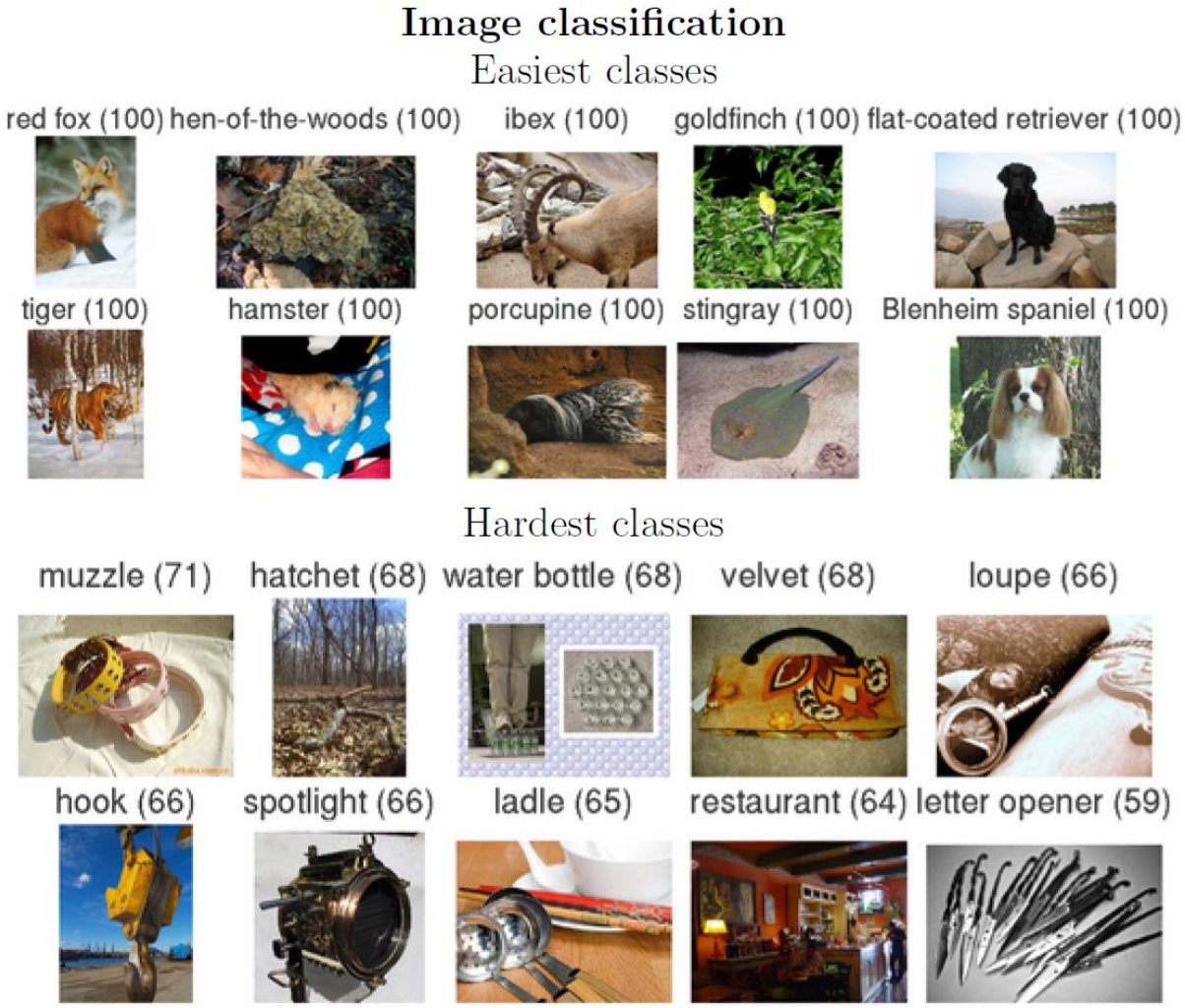


- **100** classes
- **50k** training images (500 per class)
- **10k** test images (100 per class)
- **32x32** RGB images
- Hierarchical structure: 20 super-classes with 5 subclasses each

Another subset of the 80 million Tiny Images dataset

Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

ILSVRC *aka* ImageNet /ImageNet1K



- **1000** classes
- **1.3M** training images (~1300 per class)
- **50k** validation images (50 per class)
- **100K** test images (100 per class)
- Variable resolution RGB images typically resized to **256x256**.

ImageNet (image-net.org)

Due to the ambiguity of assigning a single label to each image, performance is usually measured in terms of Top-5 Accuracy.

https://cs.stanford.edu/people/karpathy/ilsvrc/

Russakovsky et al, "ImageNet Large Scale Visual Recognition Challenge", IJCV 2015

And many more….

https://paperswithcode.com/sota

Trend	Dataset	Best Model	Paper	Code	Compare
	ImageNet	BASIC-L (Lion, fine-tuned)			See all
	CIFAR-10	ViT-H/14			See all
	CIFAR-100	EffNet-L2 (SAM)			See all
	STL-10	VIT-L/16 (Background, Spinal FC)			See all
	ObjectNet	CoCa			See all
	MNIST	Branching/Merging CNN + Homogeneous Vector Capsules			See all
	SVHN	WRN28-10 (SAM)			See all

What Machine Learning is about ?

- In Machine Learning we are given:

  - a training set, $D^{train} = \{ (\boldsymbol{x}^{(i)}, y^{(i)}) \mid i = 1, \dots, N \}$
  - a test set, $D^{test} = \{ (\boldsymbol{x}^{(i)}, y^{(i)}) \mid i = 1, \dots, M \}$

where $\boldsymbol{x}^{(i)} \in \mathbb{R}^f$, are (the features representing) the input data (e.g. images in computer vision), and $y^{(i)}$ are the outputs we want to predict for the inputs (e.g. class labels in image classification).

- We assume that the two sets contain independent and identically distributed (i.i.d.) samples from the same unknown data-generating distribution $p_{data}(\boldsymbol{x}, y)$
- The objective of a machine learning model is to make use of the samples in D^{train} in order to find a function, $y = f(\boldsymbol{x})$, that can output correct predictions on *unseen* input data, i.e. those in D^{test}.
- Typically, $f(\boldsymbol{x})$, is assumed to belong to a chosen parametric family of functions, $f(\boldsymbol{x}, \boldsymbol{\theta})$, with $\boldsymbol{\theta} \in \Theta$, such that achieving the above objective consists in finding an instance of the parameter values $\boldsymbol{\theta}^*$

Learning as Optimization – Loss Function

- How to find a parametric function that can make correct predictions on D^{test} ? At training time we have access only to D^{train}, not to D^{test}, so what we can do is finding a function that gives good predictions on D^{train}.
- Thus, learning is formulated as on optimization problem whereby one seeks to optimize an objective function that measures how good are the predictions on the data belonging to D^{train}.
- Typically, one choses an objective function, know as Loss Function (Error Function, Cost Function) such that "*the lower the better*", that is the lower the value of the Loss Function the better are the predictions provided by the machine learning model. Thus, learning consist in finding the parameter values, $\boldsymbol{\theta}^*$, that minimize the Loss Function on the training set:

$$\boldsymbol{\theta}^* = \mathbf{argmin}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \boldsymbol{L}(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}})$$

- It is common practice to work with Loss Functions such that the value computed on D^{train} is the average (or sum) of those computed on the individual data samples:

$$\boldsymbol{L}\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right) = \frac{\mathbf{1}}{\boldsymbol{N}} \sum_{\boldsymbol{i}} \boldsymbol{L}\left(\boldsymbol{\theta}, \left(\boldsymbol{x}^{(\boldsymbol{i})}, \boldsymbol{y}^{(\boldsymbol{i})}\right)\right)$$

Per-sample Loss

Hyper-parameters and Model Capacity

- Defining a machine learning model involves making some "manual" choices before training starts. In the previous formulation, such choice consists in the adopted parametric family of functions, $f(\boldsymbol{x}, \boldsymbol{\theta})$, with $\boldsymbol{\theta} \in \Theta$. When using neural networks, for example, the network architecture (number of layers, number of units in each layer,..) must be manually chosen. The choices made manually before training starts, i.e. not learnt from the training set, are referred to as hyper-parameters.
- Model Capacity may be defined informally as the "power" of a model, its ability to approximate accurately complex hidden functions. Capacity is related, among other factors, to the number of learnable parameters, i.e. in our formulation, the dimensionality of the parameter space Θ. The higher the number of parameters, the higher is the capacity. With neural networks, one may choose architectures having more/less capacity (number of layers, number of units in each layer, …). Machine learning models with higher capacity can make better predictions on $\boldsymbol{D}^{\boldsymbol{train}}$, i.e. they provide a lower training error.
- But what we actually care about is the error on unseen inputs, the so called generalization (or test) error: how to choose the model which will work best on unseen data ?

Validation Set and Model Selection

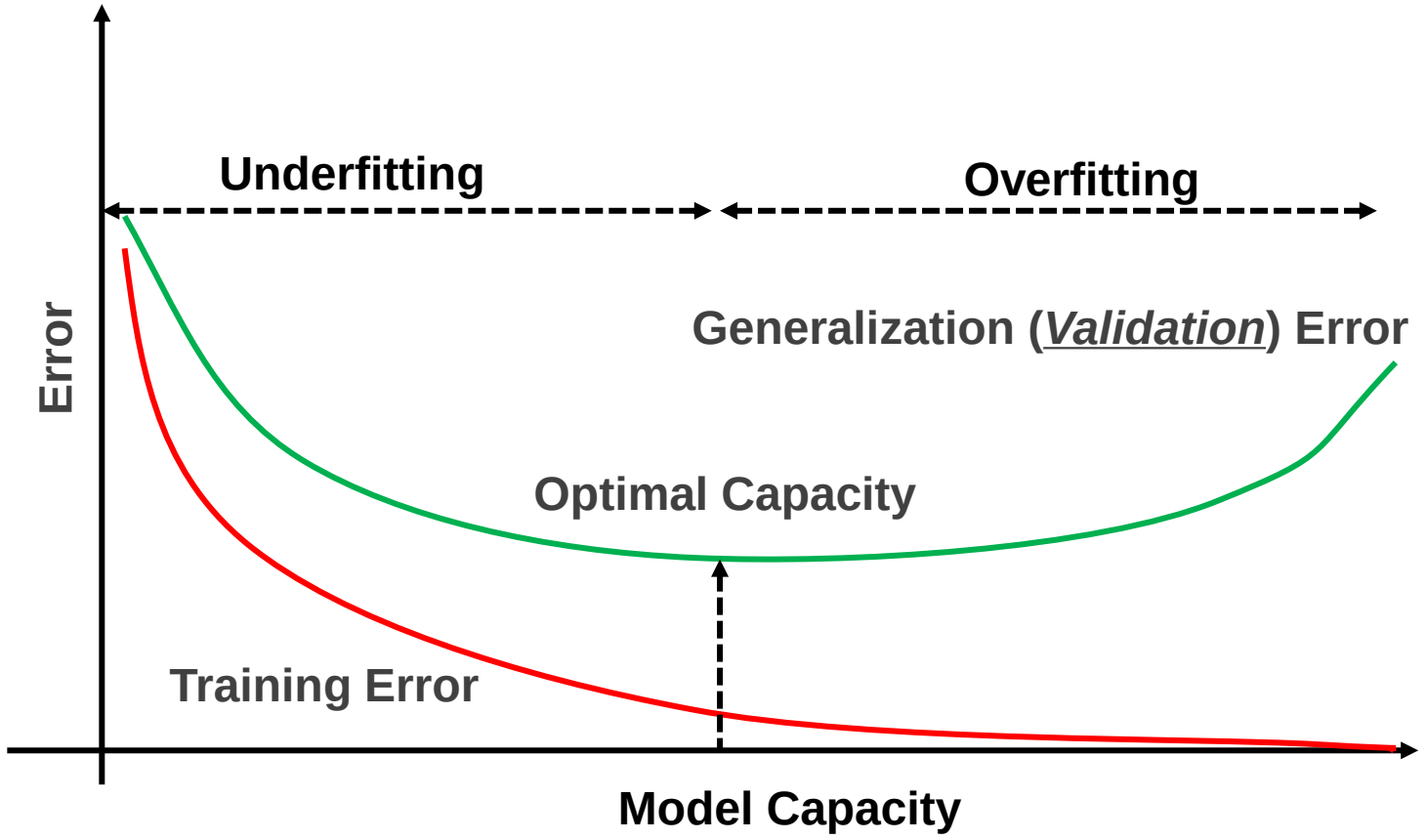
1) We held out a small part of the training set, which we call validation set.

2) We use the hyper-parameter combination performing best on the validation set.

3) We will finally run this best model (once) on the test set to estimate the generalization error.

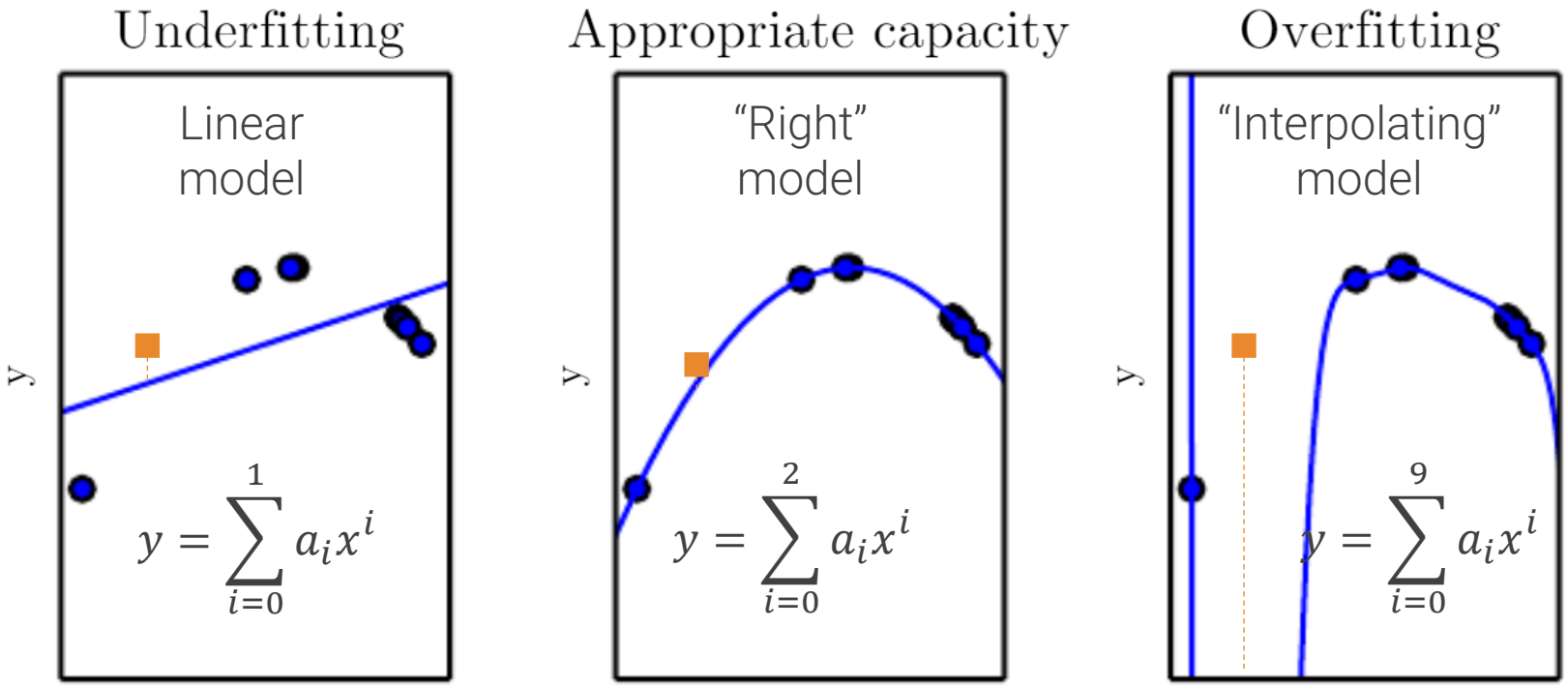
Underfitting and Overfitting

- When varying model capacity, the training error (red) and the generalization (validation) error (green) usually follow the trend shown in the Figure.
- To obtain the best generalization we want the model to work in the "sweet spot" where the training error is small and the gap between the training and validation error is also small.
- When the training error is large, the model is underfitting the training data.
- When the gap between the training error and the validation error is large, the model is overfitting the training data.



Example: Polynomial Regression

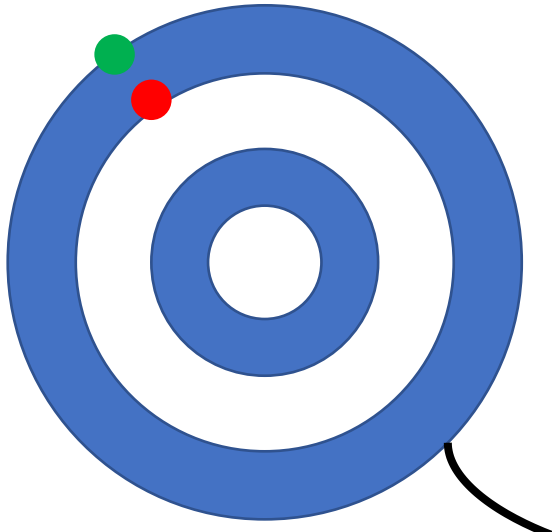
We randomly generate N data points computed by a quadratic function $y = (ax^2 + bx + c)$ and fit them with different polynomials of degree K ($y = \sum_{i=0}^{K} a_i x^i = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$) by minimizing an MSE (Mean Squared Error) Loss.



http://www.deeplearningbook.org/contents/ml.html

The path to the sweet spot

High Training Error
High Generalization Error

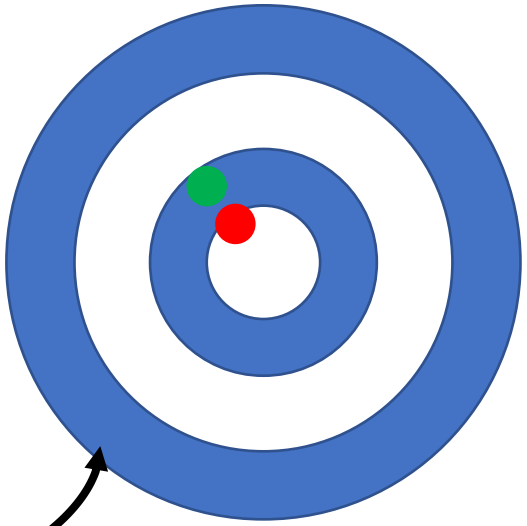


Increase Capacity
(deeper/wider network)
Train Longer

Low Training Error
High Generalization Error

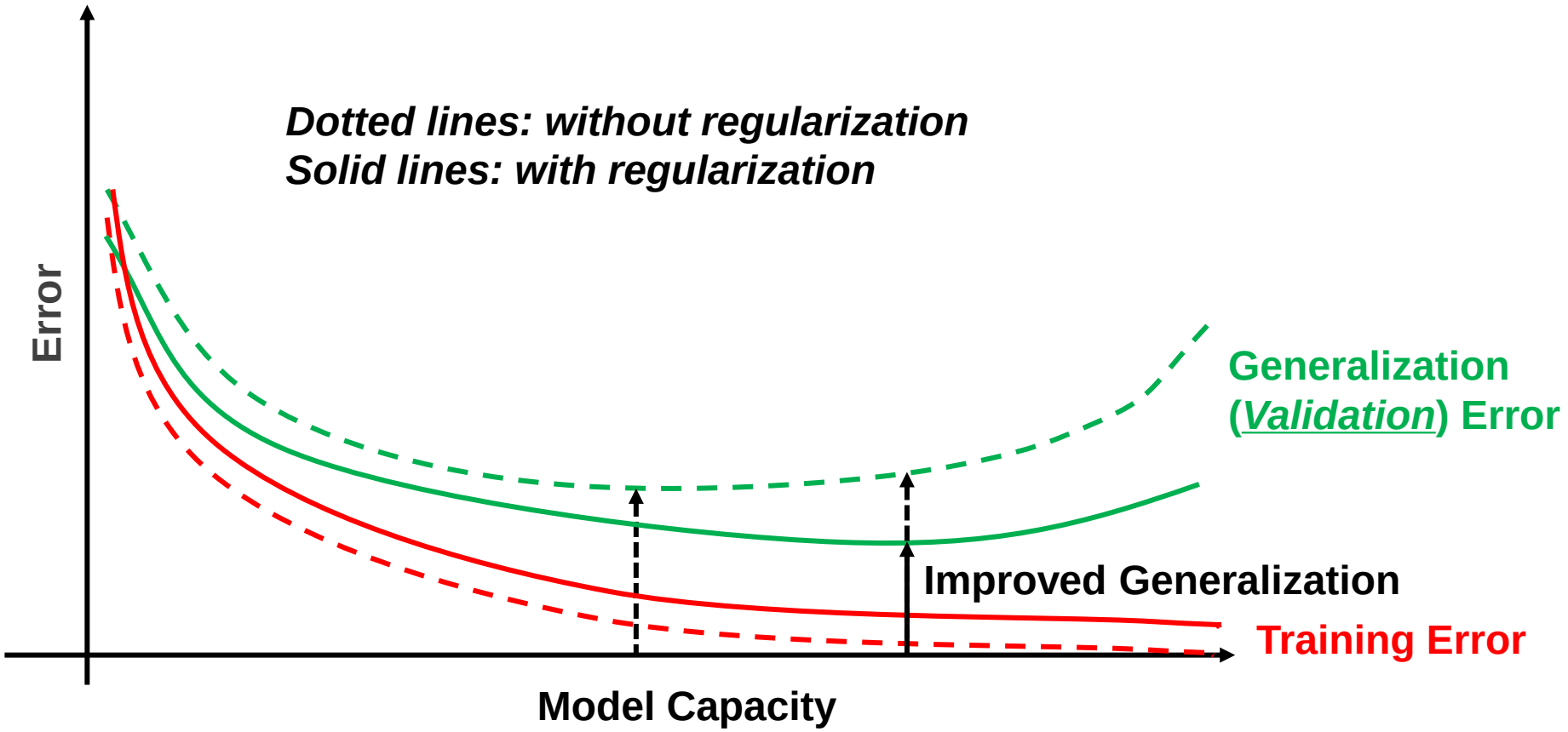
More Data
Regularization

Low Training Error
Low Generalization Error



Regularization

Regularization is any modification we make to a learning algorithm that is intended to reduce its generalization error but not its training error.



Parameter Norm Penalties

We add a term (*regularizer*) to the Loss which realizes a preference for solutions with smaller parameter values

Regularizer

$$\tilde{L}\left(\boldsymbol{\theta}; D^{train}\right) = L\left(\boldsymbol{\theta}; D^{train}\right) + \lambda\, L^{reg}(\boldsymbol{\theta})$$

The relative contribution of the two terms in the final Loss is controlled by the hyper-parameter λ, with popular regularizers being:

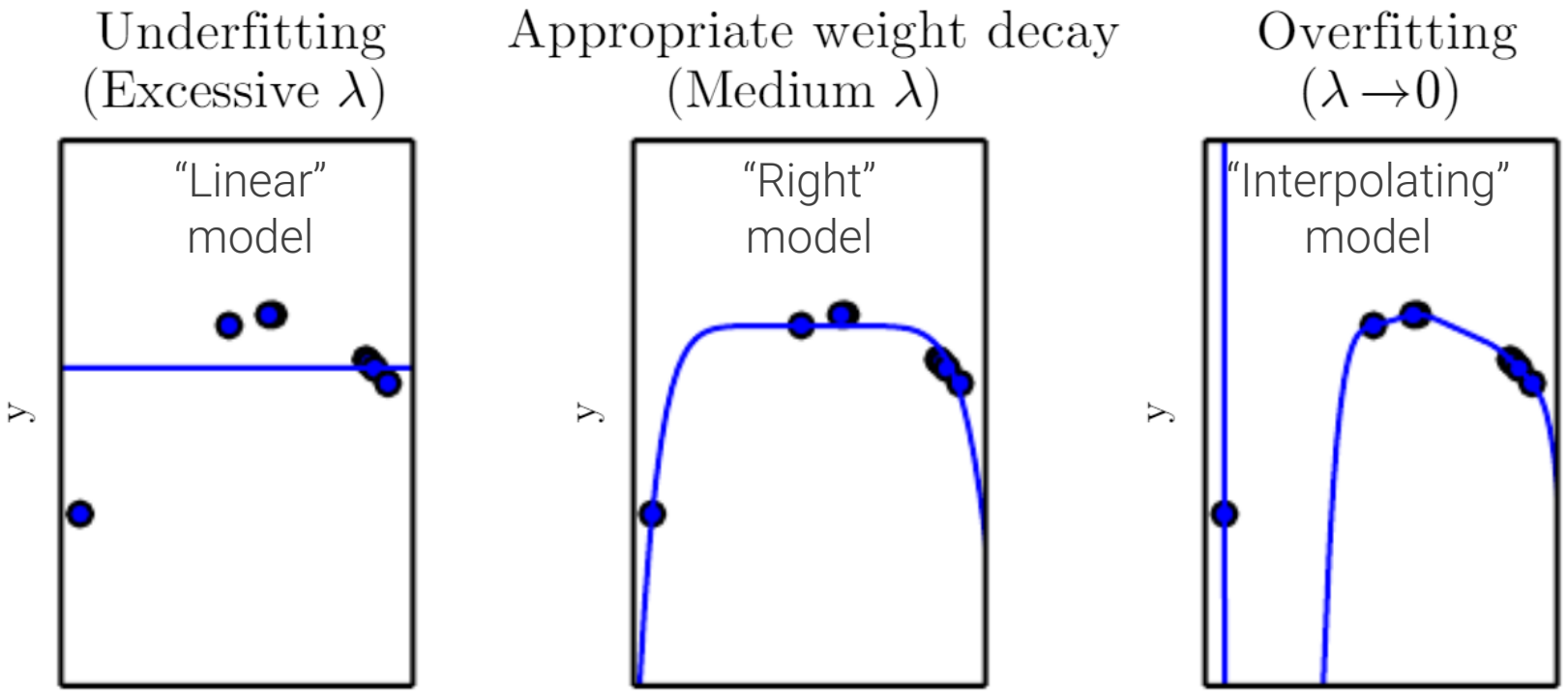
$L^{reg}(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_2^2 = \sum_i \theta_i^2$ (L_2 Regularization, Weight Decay)

$L^{reg}(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1 = \sum_i |\boldsymbol{\theta}_i|$ (L_1 Regularization)

Models with smaller parameters are “simpler” and thus tend to overfit less.

Example: Polynomial Regression with Weight Decay

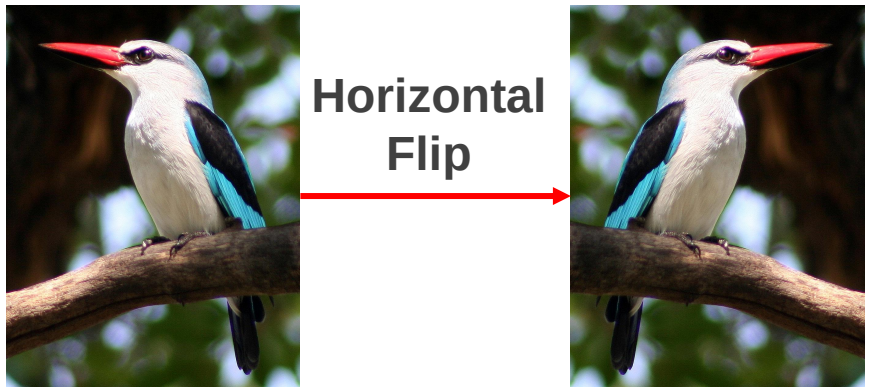
We fit the same N data points as before with 9th degree polynomials by adding an L_2 penalty to the MSE cost function with different values of λ.

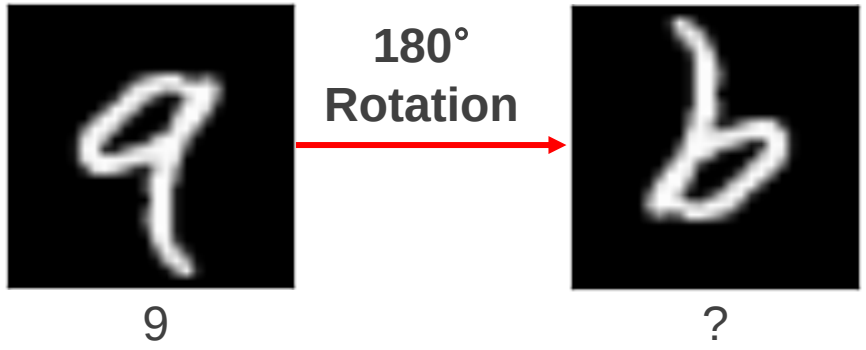


http://www.deeplearningbook.org/contents/ml.html

Data Augmentation (1)

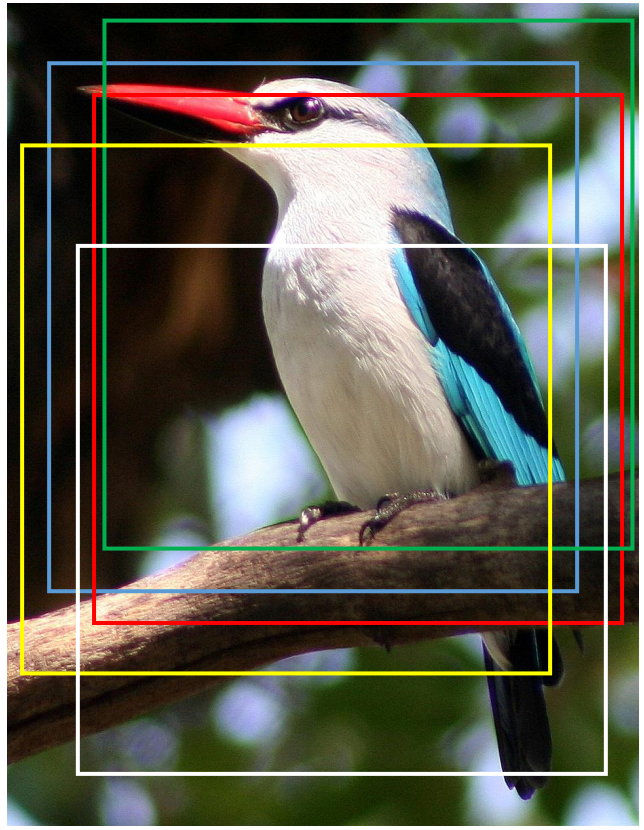
- As already discussed, training with more data improves generalization but, in practice, the amount of data we have is limited. We can get around this issue by creating “fake” data and adding them to the training set.
- This regularization (why ?) technique, known as data augmentation, is implemented by applying label-preserving transformations to the data in D^{train} so as to create a new, much larger training set.



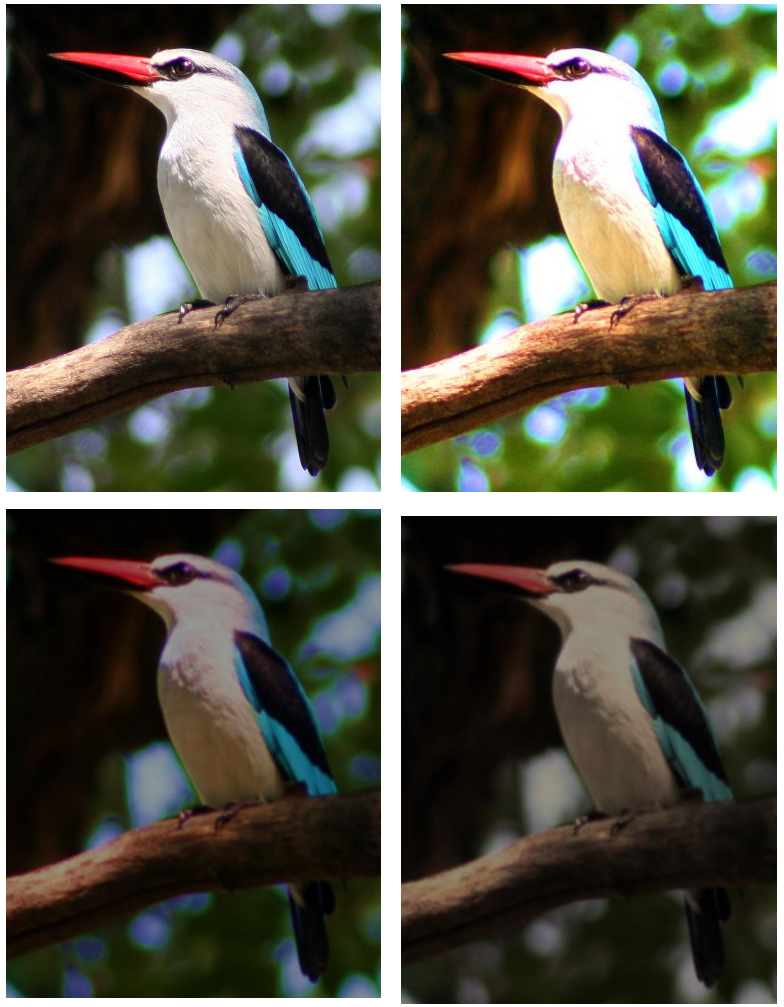


Invalid Transformation for MNIST

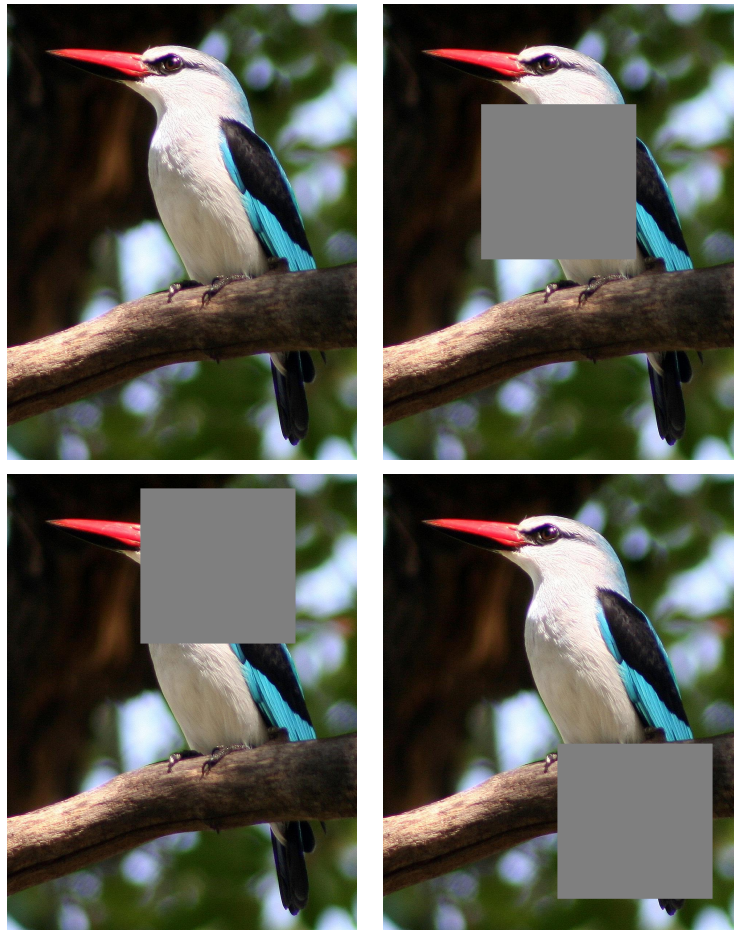
Data Augmentation (2)



Random Crop (Multi-Scale)



Colour Jittering



Cutout (remove a random square from the input image with 50% probability)

Different augmentations can be combined together.

How to design a *linear* classifier ?

Dog

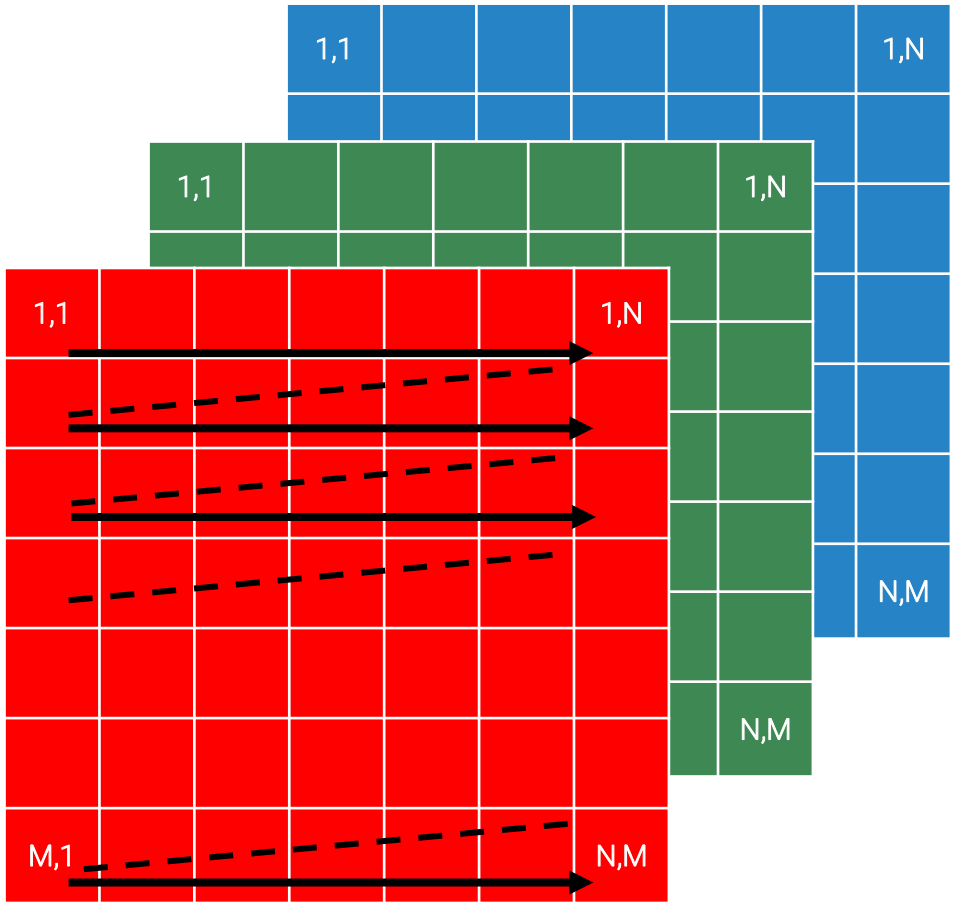
Cat

Bird

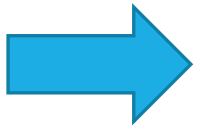
$$f(x = \text{  }; \theta) = 2$$

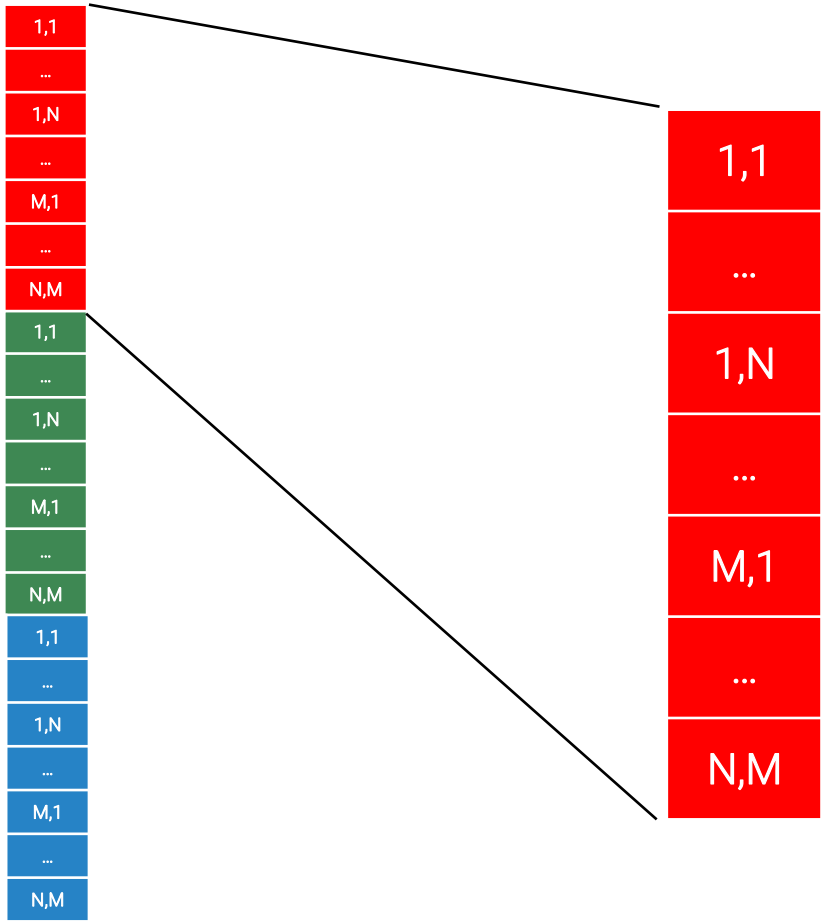
$$f(\boldsymbol{x}; W) = W\boldsymbol{x} = \boldsymbol{label}$$

Image Flattening



3xMxN





$\boldsymbol{x}$ =(3*M*N)x1

A bad *linear* classifier

$$f(x = \text{  }; \theta) = 2$$

$$f(\boldsymbol{x}; W) = W\boldsymbol{x} = \boldsymbol{label}$$

32x32x3=3072x1 CIFAR image

1x3072

1x1 (Scalar)

A class label is a categorical not a numerical variable !

Close or distant values do not imply that the corresponding classes are visually similar or dissimilar (in other words, if cat=3 and we get 2.5 out of f, it does not mean that x depicts a fantastic beast looking half-bird and half-cat).

A better one

$$f(x = \text{  } ; \theta) = 253$$

0 (plane)	45.4
1 (car)	128.3
2 (bird)	253
...	0.23
	-1.34
	4
	56
	-63
	78
	2

$\boldsymbol{argmax}$ → 2

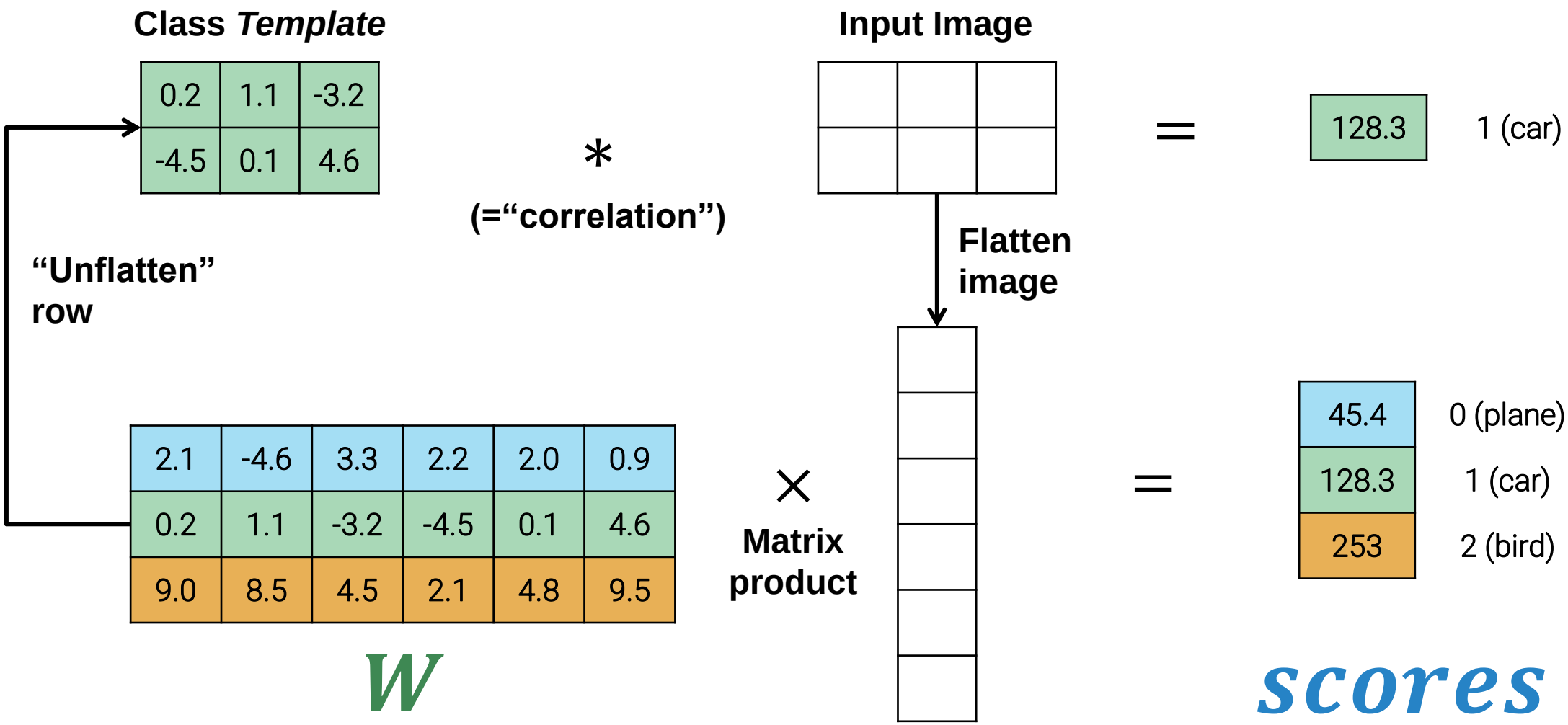
$$f(\boldsymbol{x}; W) = W\boldsymbol{x} = \boldsymbol{scores}$$

32x32x3=3072x1 CIFAR image

10x3072

10x1

Linear Classification as *Template Matching*



Linear means *affine*, actually.

$$f(x = \quad ; \theta) = 253$$

0 (plane)	45.4
1 (car)	128.3
2 (bird)	253
...	0.23
	-1.34
	4
	56
	-63
	78
	2

$\boldsymbol{argmax}$ → 2

$$f(\boldsymbol{x}; \theta) = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} = \boldsymbol{scores}$$

32x32x3=3072x1 CIFAR image

10x3072

10x1

10x1

Back to Loss Functions

Learning consists in finding the parameter values that minimize a Loss Function

$$\boldsymbol{\theta}^* = \mathbf{argmin}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \boldsymbol{L}(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}})$$

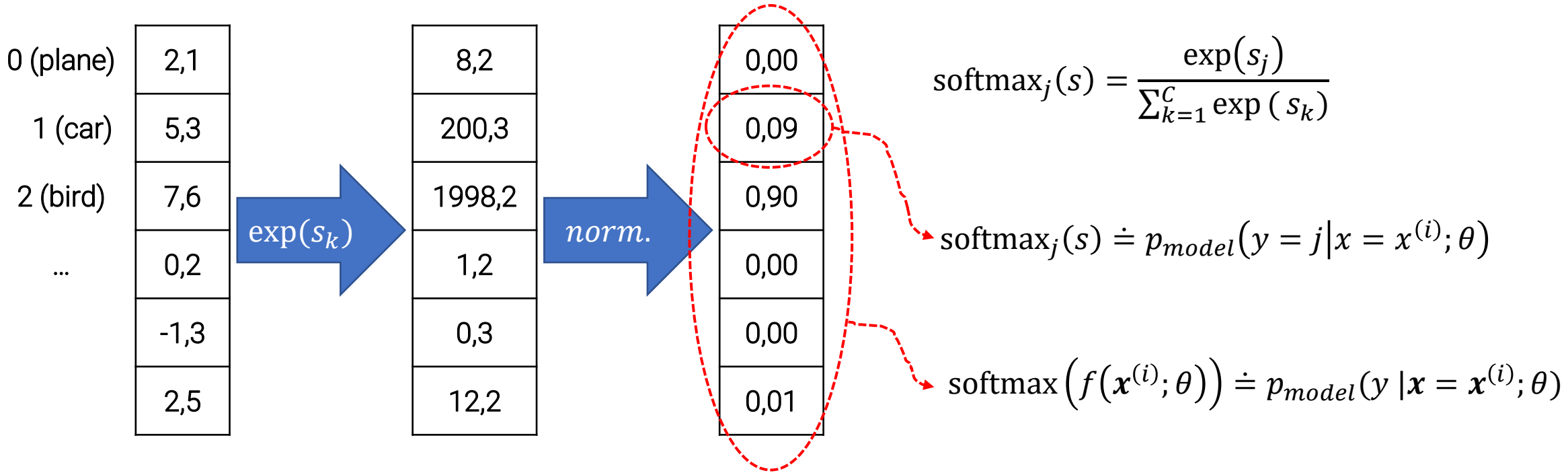
typically obtained as a sum (average) of terms computed on the individual data samples:

$$\boldsymbol{L}\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right) = \sum_{\boldsymbol{i}} \boldsymbol{L}\left(\boldsymbol{\theta}, \left(\boldsymbol{x}^{(\boldsymbol{i})}, \boldsymbol{y}^{(\boldsymbol{i})}\right)\right)$$

Which Loss can we use to train a linear classifier ?

Scores as probabilities: the *softmax* function

The modern approach to train linear (and non-linear, e.g. realized as neural networks) classifiers entails transforming the class scores into probabilities. This is achieved by processing the scores by the *softmax* function and it allows for interpreting the j^{th} output as the *conditional probability of class j* given the input image. Accordingly, the whole output vector can be interpreted as the *probability mass function* of the class given the input image.



Cross-entropy Loss

Given that the model outputs probabilities, how should we represent the "true" label of a training sample, e.g. $(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} = \boldsymbol{j}) \in \boldsymbol{D}^{\boldsymbol{train}}$?

$$\mathbb{I}(\boldsymbol{y}^{(i)}) =$$

One-hot Encoding

0
0
0
j → 1
0
0

The *per-sample* loss must be a function that gets lower as the probability given by the model to the true class gets higher:

$$\boldsymbol{L}\left(\boldsymbol{\theta}, (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\right) = -\log p_{model}\left(y = y^{(i)} \middle| \boldsymbol{x} = \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right)$$

0 (plane)	0,00	
1 (car)	0,09	$-\log(0.09) = 2.4\ (high)$
2 (bird)	0,90	$-\log(0.9) = 0.1\ (\ low)$
	0,00	
	0,00	
	0,01	

$$\boldsymbol{L}\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right) = \sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} -\log p_{model}\left(y = y^{(i)} \middle| \boldsymbol{x} = \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right)$$

(Cross-entropy Loss)

Maximum Likelihood Estimation & Cross-entropy

Maximum Likelihood Estimation of a parametric model consists in finding the parameter values that make the observations on the unknown data generating distribution contained in the training set most likely, in other words finding the parameters that best explain the training set.

$$\theta^* = \arg\max_{\theta} p_{model}\left(\boldsymbol{D}^{\boldsymbol{train}}; \boldsymbol{\theta}\right)$$

$$\theta^* = \arg\max_{\theta} p_{model}\left(y^{(1)}, \dots, y^{(N)} \middle| \boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(N)}; \boldsymbol{\theta}\right)$$

$$= \arg\max_{\theta} \prod_{i=1}^{N} p_{model}\left(y = y^{(i)} \middle| \boldsymbol{x} = \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \qquad \textbf{(i.i.d.)}$$

$$= \arg\max_{\theta} \sum_{i=1}^{N} \log p_{model}\left(y = y^{(i)} \middle| \boldsymbol{x} = \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \qquad \textbf{(avoid the curse of products)}$$

$$= \arg\min_{\theta} \sum_{i=1}^{N} -\log p_{model}\left(y = y^{(i)} \middle| \boldsymbol{x} = \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \qquad \textbf{(Cross-entropy Loss)}$$

Why Cross-entropy ?

Let's consider two discrete distributions (i.e. probability mass functions), p and q defined on the same set of events $\mathbb{X}$. The cross-entropy of p with respect to q is defined as

$$H(p,q) = -\sum_{x \in \mathbb{X}} p(x) \log q\,(x)$$

and represents a measure of the "difference" between them (the lower, the more similar).

In our classification problem we may think of $\mathbb{X}$ as to the set of classes to be assigned to the given input image and compute $H(p,q)$ with p being the "true" distribution, as defined by the one-hot encoding of the label, and q that computed by the model:

$$H\left(\mathbb{I}(y^{(i)}), p_{model}\left(y \middle| x^{(i)}; \theta\right)\right) = -\sum_{k=1}^{C} \mathbb{I}\left(y^{(i)}\right)_k \log p_{model}\left(y = k \middle| x^{(i)}; \theta\right)$$

$$= -\log\, p_{model}\left(y = y^{(i)} \middle| x^{(i)}; \theta\right)$$

(Per-sample Cross-entropy Loss)

Putting all together

Given the training sample $\left(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}\right)$, for each $j \in \{1..k\}$ the classifier outputs

$$p_{model}\left(y = j \middle| x = x^{(i)}; \theta\right) = \frac{\exp\left(s_j\right)}{\sum_{k=1}^{C} \exp\left(s_k\right)}$$

and the per-sample loss is given by

$$-\log p_{model}\left(y = y^{(i)} \middle| x^{(i)}; \theta\right)$$

$$= -\log\left(\frac{\exp\left(s_{y^{(i)}}\right)}{\sum_{k=1}^{C} \exp\left(s_k\right)}\right) = -s_{y^{(i)}} + \log\left(\sum_{k=1}^{C} \exp\left(s_k\right)\right) \approx -s_{y^{(i)}} + \max_k s_k$$

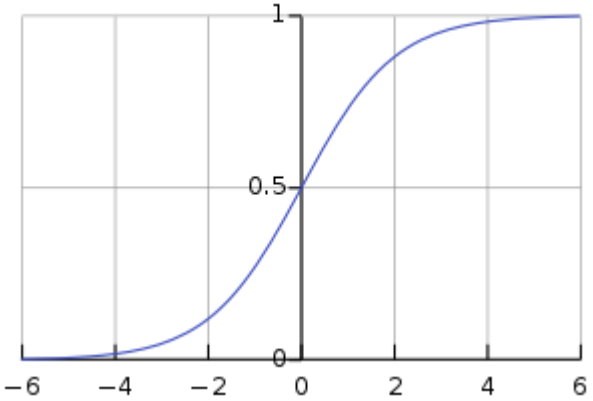
$\approx \max_k s_k$ due to $\exp\left(s_k\right)$ being insignificant for any s_k noticeably less than $\max_k s_k$

When the maximum score is given to the correct class ($\max_k s_k = s_{y^{(i)}}$) the loss is very small, otherwise it strongly penalizes the most active incorrect prediction.

Special case: binary classifier

When the classes are two, we can design a classifier that outputs the probability of only one of the two classes. Purposely, the score computed for the *chosen* class, *s*, can be converted into the corresponding probability by the *Sigmoid* function, $\sigma(s)$.

$$\sigma(s) = \frac{1}{1 + \exp -(s)}$$



$$\begin{cases} \sigma(s) \doteq p_{model}\left(y = chosen\ class \middle| x = x^{(i)}; \theta\right) \\ 1{-}\sigma(s) = p_{model}\left(y = other\ class \middle| x = x^{(i)}; \theta\right) \end{cases}$$

Moreover, we can conveniently represent the labels of the training samples by the values of a binary variable, i.e. $\boldsymbol{y^{(i)} = 1/0}$ depending on whether the i_{th} sample belongs to the *chosen* or *other* class, and deploy the so-called **Binary Cross-entropy Loss**:

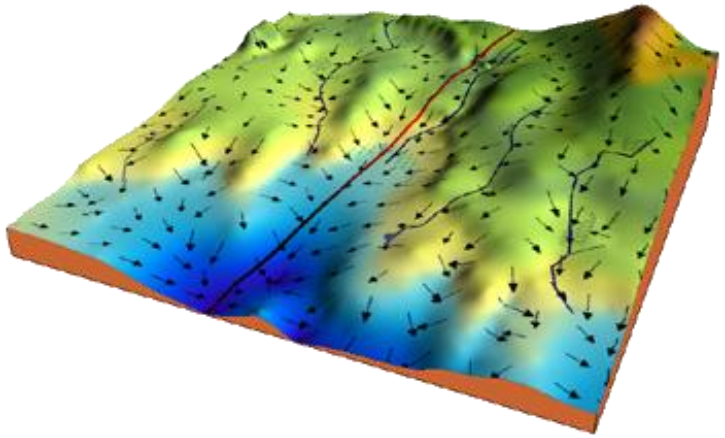
$$\boldsymbol{L\left(\theta, (x^{(i)}, y^{(i)})\right) = -\left(y^{(i)} \log p_{model}\left(y = y^{(i)} \middle| \boldsymbol{x} = \boldsymbol{x^{(i)}}; \boldsymbol{\theta}\right) + \left(\mathbf{1} - y^{(i)}\right) \log\left(1 - p_{model}\left(y = y^{(i)} \middle| \boldsymbol{x} = \boldsymbol{x^{(i)}}; \boldsymbol{\theta}\right)\right)\right)}$$

(Per-sample Binary Cross-entropy Loss)

How to minimize the Loss ?

The Loss captures preferences about the parameters of a machine learning model: we prefer those yielding a lower loss. To perform training, thus, we treat the Loss as a multivariate function, *the variables being the parameters of the model*, and try to find the values that yield a low (possibly the lowest) Loss:

$$\boldsymbol{\theta}^* = \mathbf{argmin}_{\boldsymbol{\theta} \in \boldsymbol{\Theta}} \boldsymbol{L}\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right)$$

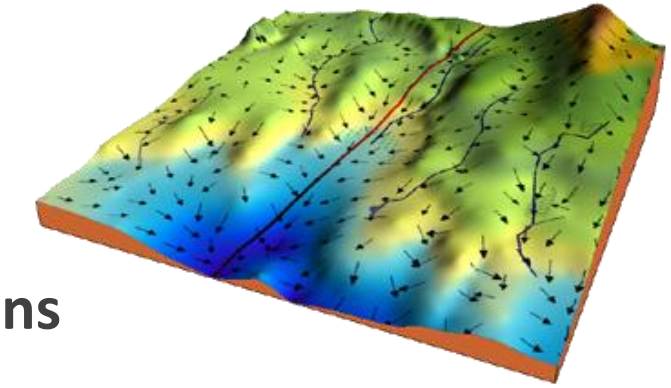


One could compute the Loss for many randomly chosen parameter values and then pick the choice yielding the lowest loss. However, this “brute force” approach would be rather inefficient, especially in high-dimensional parameter spaces. A much better strategy consist in starting from a random choice and then following the direction of the gradient of the Loss *wrt* the parameters to pick a better one (Gradient Descent).

Gradient Descent

$$L\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right) \rightarrow \nabla L\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right) = \begin{bmatrix} \frac{\partial L\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right)}{\partial \theta_1} \\ \vdots \\ \frac{\partial L\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right)}{\partial \theta_k} \end{bmatrix}$$

Gradient of the Loss
wrt
the parameters



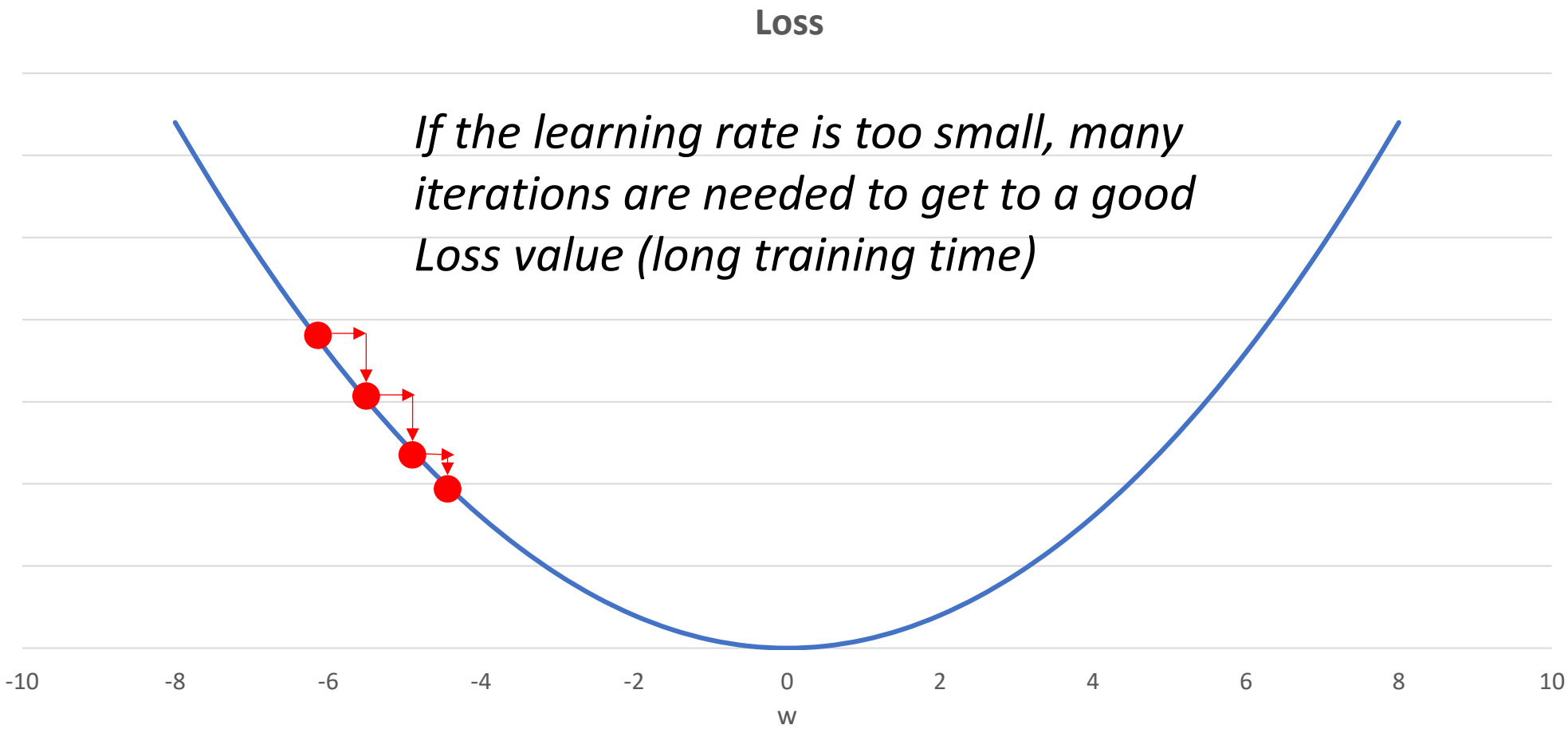
0. Randomly initialize $\boldsymbol{\theta}^{(0)}$

for $e = 1, .., E$ epochs

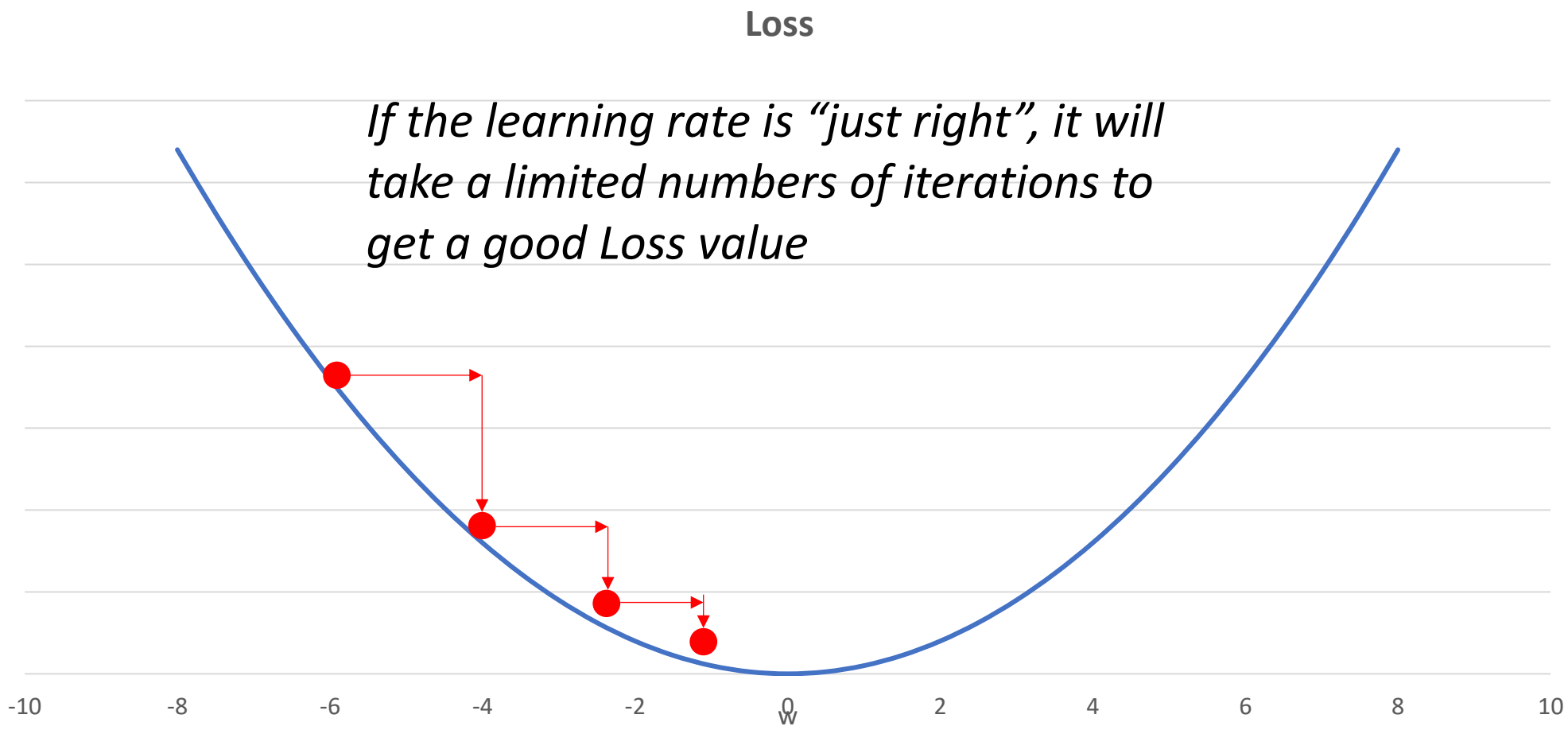
1. Forward pass: classify all the training data to get the predictions $\hat{\boldsymbol{y}}^{(i)} = \mathbf{f}\left(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}^{(e-1)}\right)$ and the loss $L\left(\boldsymbol{\theta}^{(e-1)}, \boldsymbol{D}^{\boldsymbol{train}}\right)$
2. Backward pass: compute the gradient $\nabla \boldsymbol{L} = \nabla \boldsymbol{L}\left(\boldsymbol{\theta}^{(e-1)}, \boldsymbol{D}^{\boldsymbol{train}}\right)$ (Backpropagation Algorithm)
3. Parameters Update Step: $\boldsymbol{\theta}^{(e)} = \boldsymbol{\theta}^{(e-1)} - \alpha \nabla \boldsymbol{L}$

Learning Rate

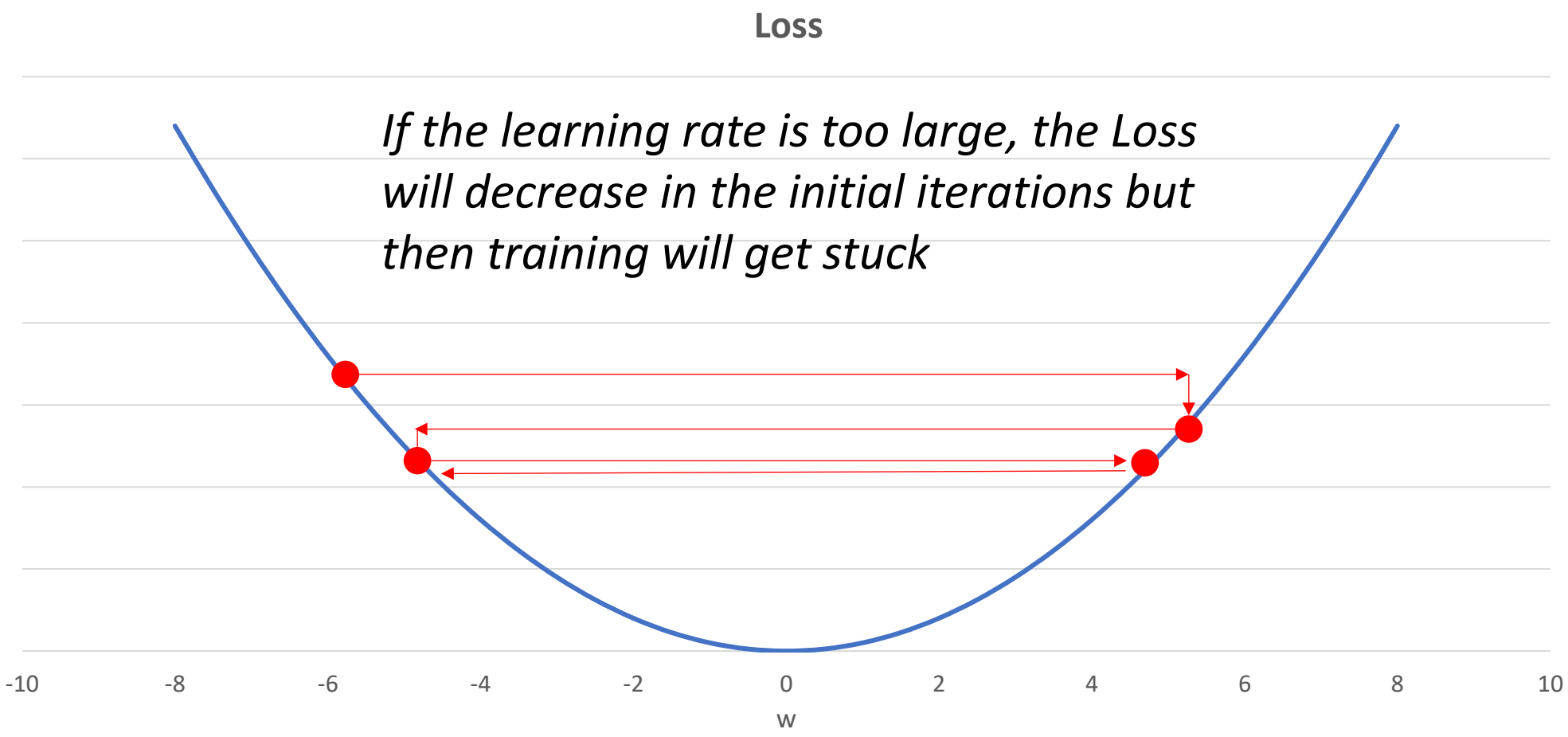
Learning Rate and Gradient Descent (1)



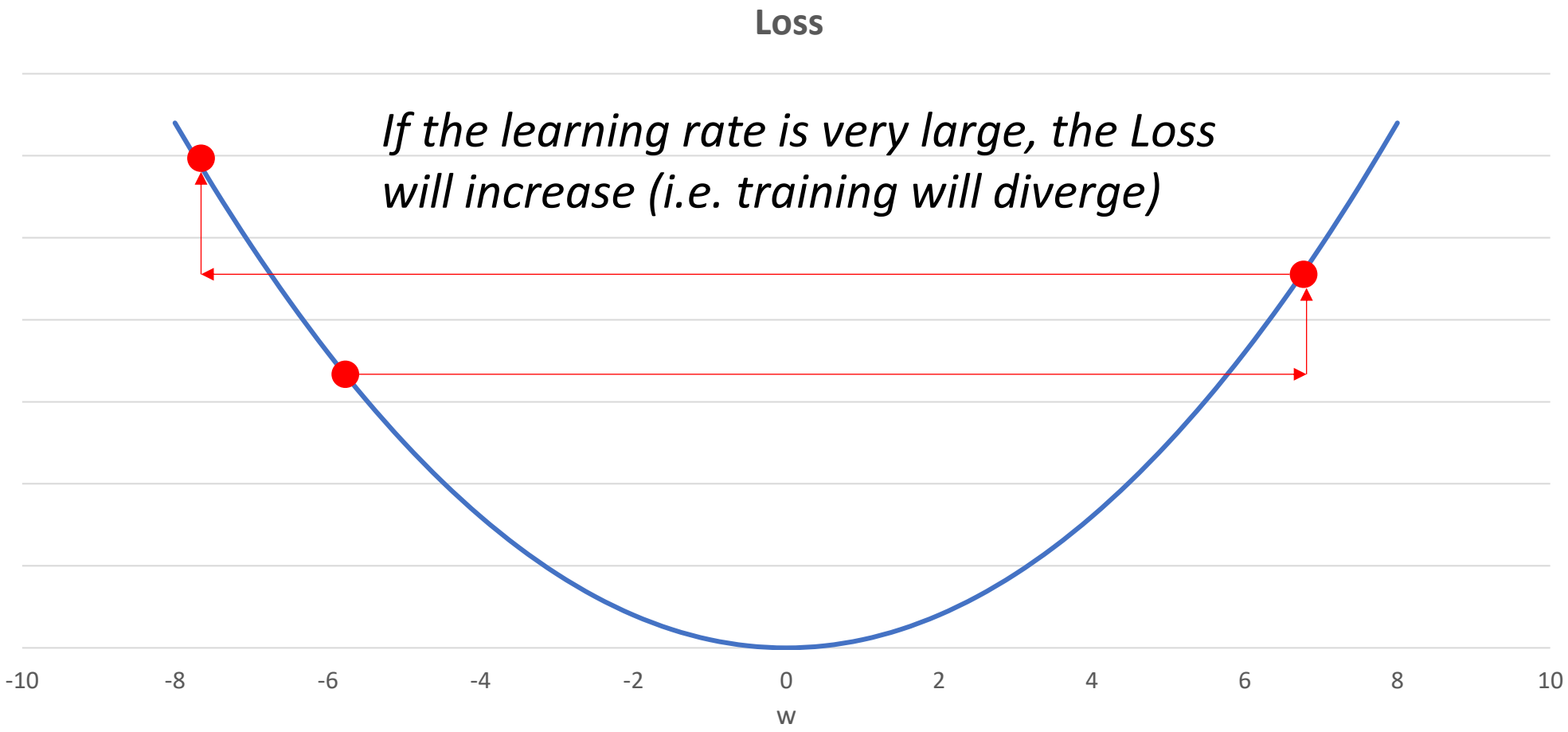
Learning Rate and Gradient Descent (2)



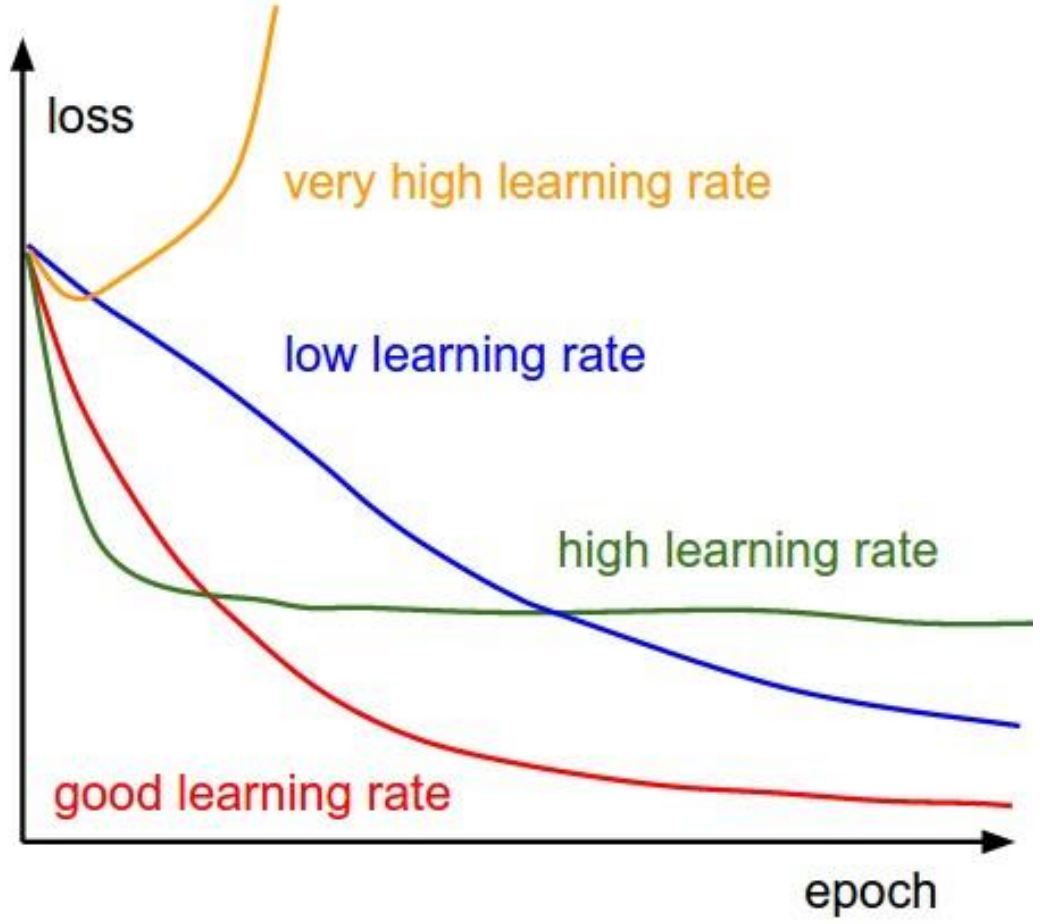
Learning Rate and Gradient Descent (3)



Learning Rate and Gradient Descent (4)



Learning Rate is a key hyper-parameter



Limits of (*Batch*) Gradient Descent

$$\boldsymbol{L}\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right) = \sum_{\boldsymbol{i}} \boldsymbol{L}\left(\boldsymbol{\theta}, \left(\boldsymbol{x}^{(\boldsymbol{i})}, \boldsymbol{y}^{(\boldsymbol{i})}\right)\right) \quad \Rightarrow \quad \nabla \boldsymbol{L}\left(\boldsymbol{\theta}, \boldsymbol{D}^{\boldsymbol{train}}\right) = \sum_{\boldsymbol{i}} \nabla \boldsymbol{L}\left(\boldsymbol{\theta}, \left(\boldsymbol{x}^{(\boldsymbol{i})}, \boldsymbol{y}^{(\boldsymbol{i})}\right)\right)$$

As the total Loss is the sum of all the per-sample losses, the total gradient is the sum of all the gradients of the per-sample losses associated with the individual training samples. Thus, to perform a single (usually tiny) parameter update step one needs to process ALL the training samples. If the training set is large, this is computationally inefficient (oftentimes infeasible) in terms of run-time and/or memory occupancy.

An alternative approach, known as Stocastic Gradient Descent (SGD) or *Online Gradient Descent*, consists in updating the parameters based on the gradient computed on each per-sample loss, i.e. after processing each individual training sample. The samples in $\boldsymbol{D}^{\boldsymbol{train}}$ are randomly shuffled before each epoch. SGD provides a computationally more efficient (a single update step takes less time and one does not need to store in memory the whole training set) though noisy approximation of *batch* Gradient Descent.

SGD with Minibatches

An effective trade-off deals with computing the gradient required to perform an update step based on a *mini-batch* consisting of B (*batch size*) training samples. Thus, the number of update steps in each epoch is given by: $U = \left\lceil \frac{N}{B} \right\rceil$.

Larger batches provide smoother approximations of the gradient at the cost of higher memory requirements. Typical sizes are (powers of 2): B=16…256.

0. **Randomly initialize $\theta^{(0)}$**

for $e = 0, .., E-1$ epochs

1. **Randomly shuffle the samples in D^{train}**

for $u = 0, .., U-1$ *minibatches*

2. **<u>Forward pass</u>: classify the samples in $X = \{x^{(Bu)}, \dots, x^{(B(u+1)-1)}\}$ to get the predictions $\hat{Y} = \{\hat{y}^{(Bu)}, \dots, \hat{y}^{(B(u+1)-1)}\} = f\left(X; \theta^{(e*U+u)}\right)$ and the loss $L\left(\theta^{(eU+u)}, \left(X, \hat{Y}\right)\right)$**

3. **<u>Backward pass</u>: compute the gradient $\nabla L = \nabla L\left(\theta^{(eU+u)}, \left(X, \hat{Y}\right)\right)$**

4. **Parameters Update Step: $\theta^{(eU+u+1)} = \theta^{(eU+u)} - \alpha \nabla L$**

Beyond “plain vanilla” SGD

- To obtain a smother convergence by dampening the oscillations due to the variance of the stochastic gradient we can add a “momentum” term in the update step:

$$\beta \in [0,1),\ v^{(0)} = 0$$

$$\begin{cases} v^{(t+1)} = \beta v^{(t)} - \alpha\, \nabla L\left(\theta^{(t)}\right) \\ \theta^{(t+1)} = \theta^{(t)} + v^{(t+1)} \end{cases}$$

$$(0)\ \begin{cases} v^{(1)} = -\alpha\, \nabla L\left(\theta^{(0)}\right) \\ \theta^{(1)} = \theta^{(0)} - \alpha\, \nabla L\left(\theta^{(0)}\right) \end{cases}$$

$$(1)\ \begin{cases} v^{(2)} = -\beta\alpha\, \nabla L\left(\theta^{(0)}\right) - \alpha\, \nabla L\left(\theta^{(1)}\right) \\ \theta^{(2)} = \theta^{(1)} - \beta\alpha\, \nabla L\left(\theta^{(0)}\right) - \alpha\, \nabla L\left(\theta^{(1)}\right) \end{cases}$$

$$(2)\ \begin{cases} v^{(3)} = -\beta^2\alpha\, \nabla L\left(\theta^{(0)}\right) - \beta\alpha\, \nabla L\left(\theta^{(1)}\right) - \alpha\, \nabla L\left(\theta^{(2)}\right) \\ \theta^{(3)} = \theta^{(2)} - \beta^2\alpha\, \nabla L\left(\theta^{(0)}\right) - \beta\alpha\, \nabla L\left(\theta^{(1)}\right) - \alpha\, \nabla L\left(\theta^{(2)}\right) \end{cases}$$

Thus, the actual update at each steps turns out to be a running average of the previous ones, with more weight given to more recent updates (exponentially decaying average).

Polyak, B.T. Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics, 4(5):1−17, 1964.
Ilya Sutskever et al., On the importance of initialization and momentum in deep learning, ICML 2013

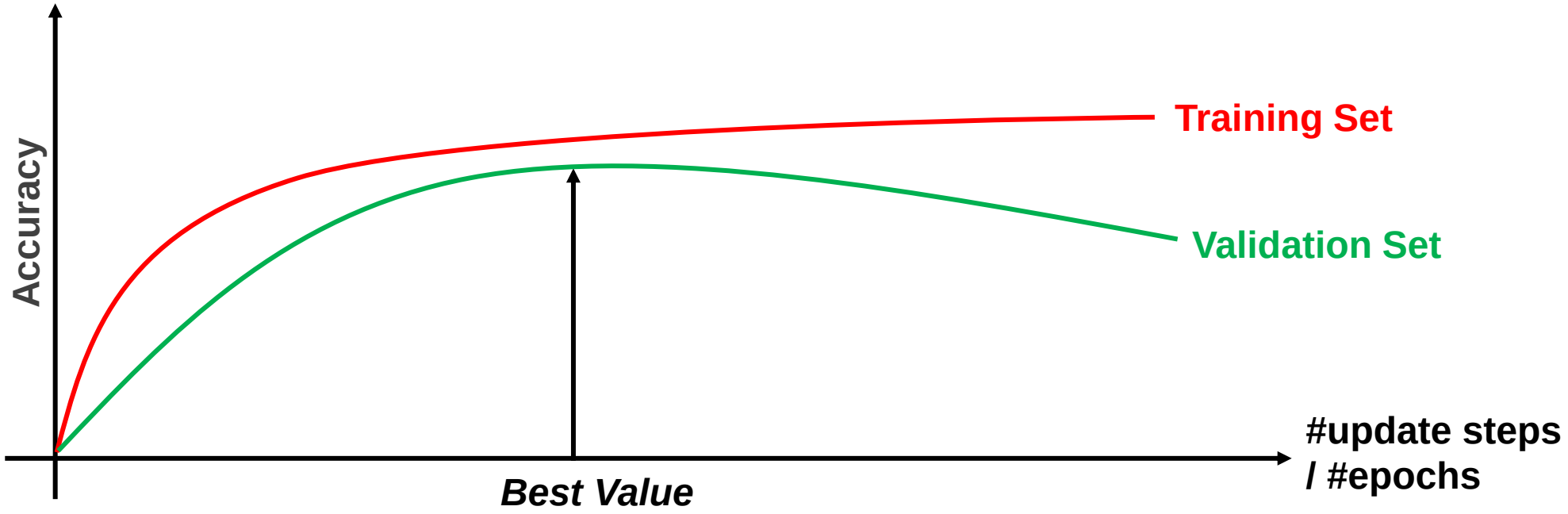
- ADAM (ADAptive Moments) adjusts learning rates dynamically for each parameter by keeping exponentially decaying averages of gradients and squared gradients.

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

The optimizer (and its parameters) are hyper-parameters !

Back to Regularization – Early Stopping

Training time (*i.e.* number update steps) is a hyper-parameter controlling the actual capacity of the model. By selecting the model which performs best on the validation set we are indeed "tuning" the best value for this hyperparameter.



http://www.deeplearningbook.org/contents/regularization.html

Limits of “shallow” classifiers

“Templates” learnt by a linear classifier

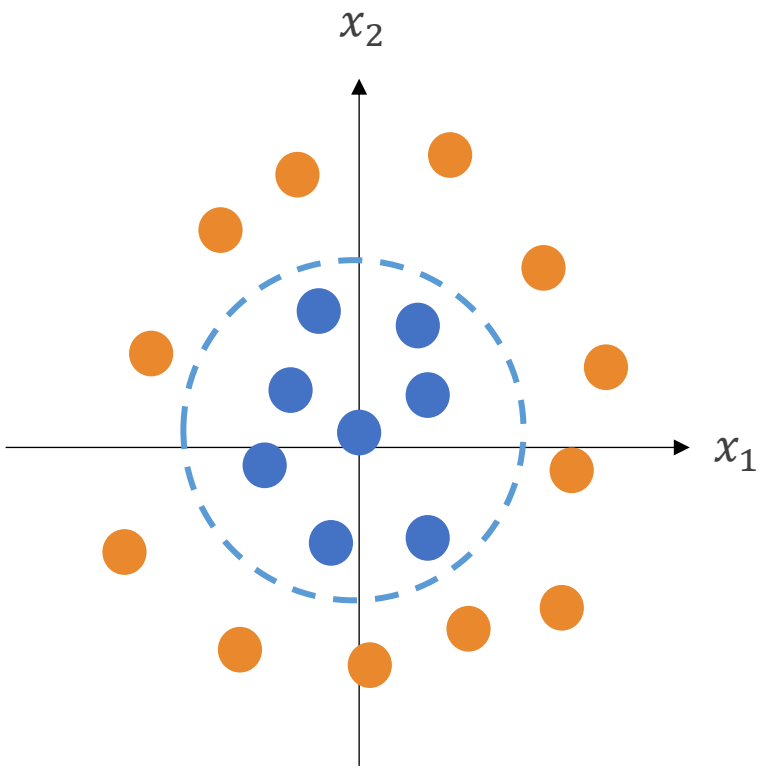
For several classes the most important feature seems to be the background color.

A single template may hardly capture a large intra-class variability.

Classifying images using as features the raw pixel data turns out to be a very difficult task !

Representation is important

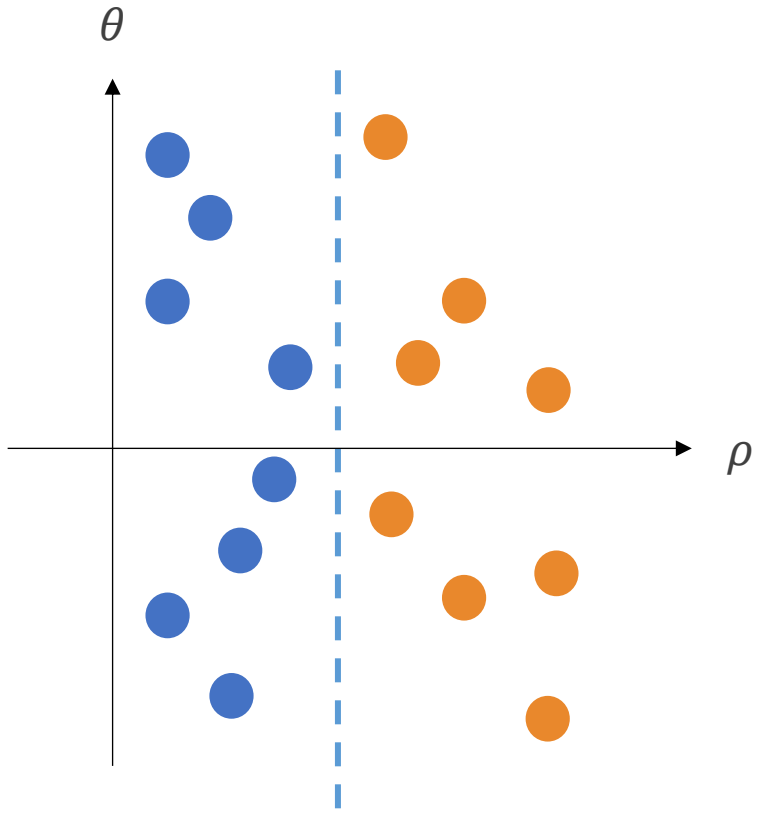
Transforming the data into a better representation can make a classification task easier



Switch to polar coordinates

$$\rho = \sqrt{x_1^2 + x_2^2}$$

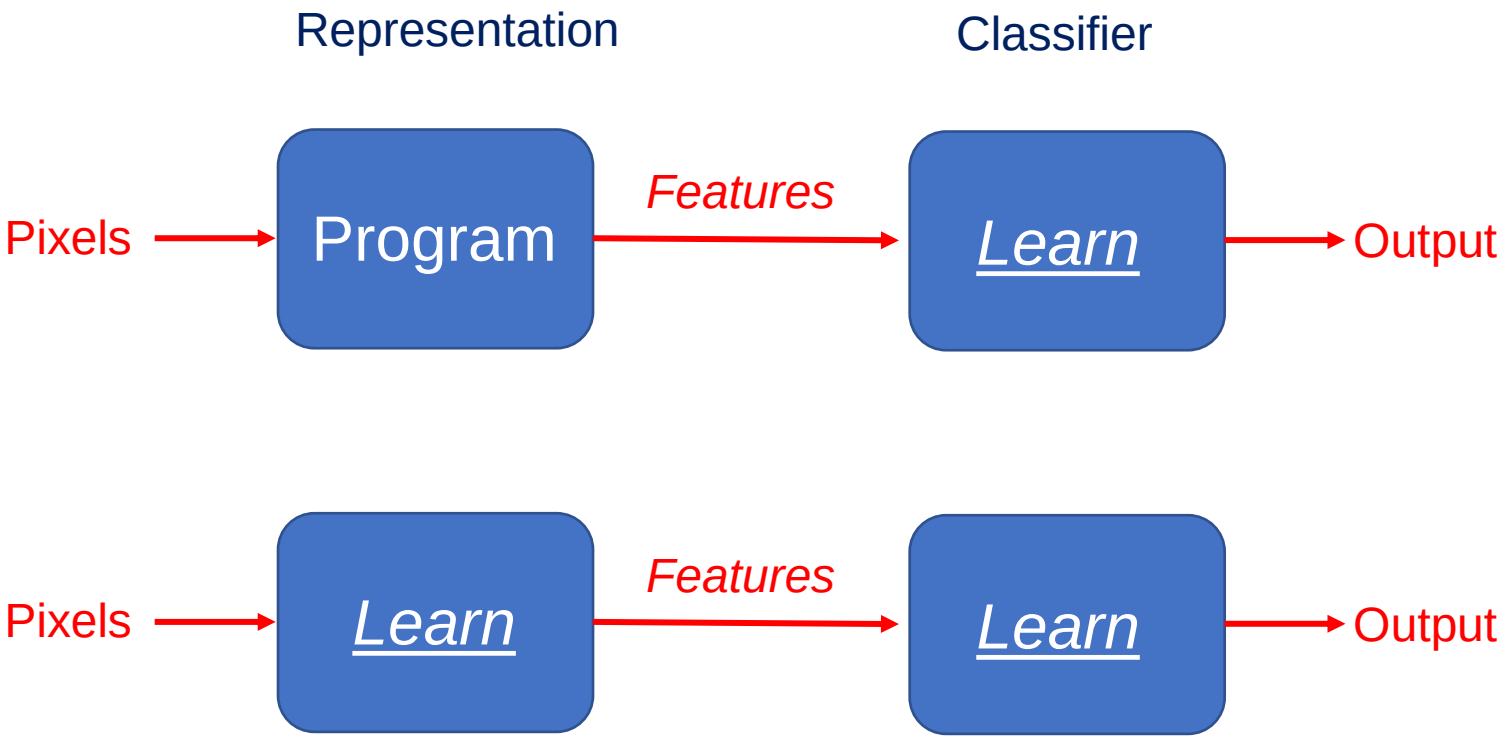
$$\theta = \tan^{-1}\frac{x_2}{x_1}$$



Non-linear decision boundary in input space

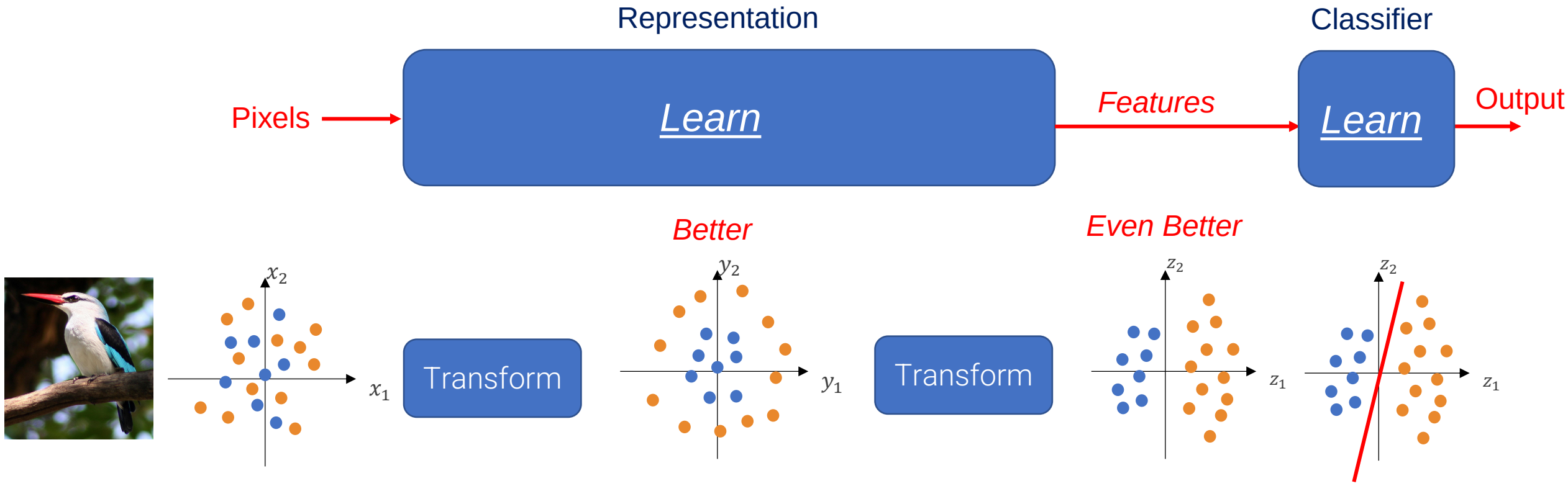
Linear decision boundary in feature space

Machine (Shallow) Learning vs. Deep Learning



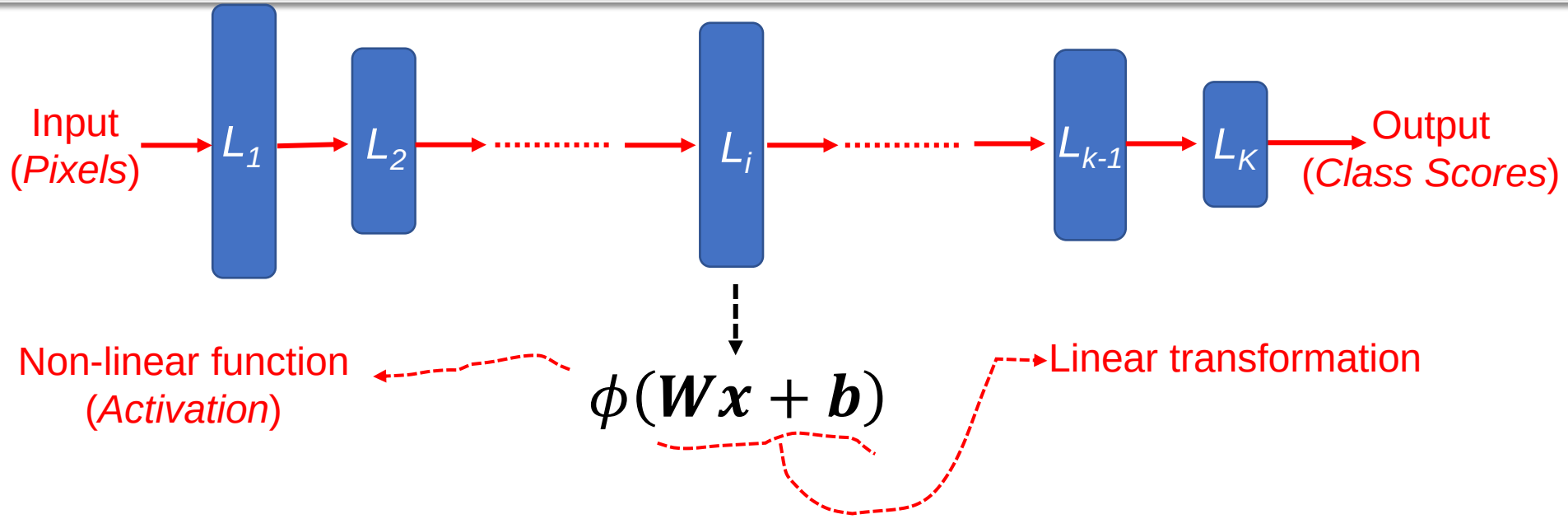
Deep learning ≈ Representation Learning

→ a hierarchy of representations

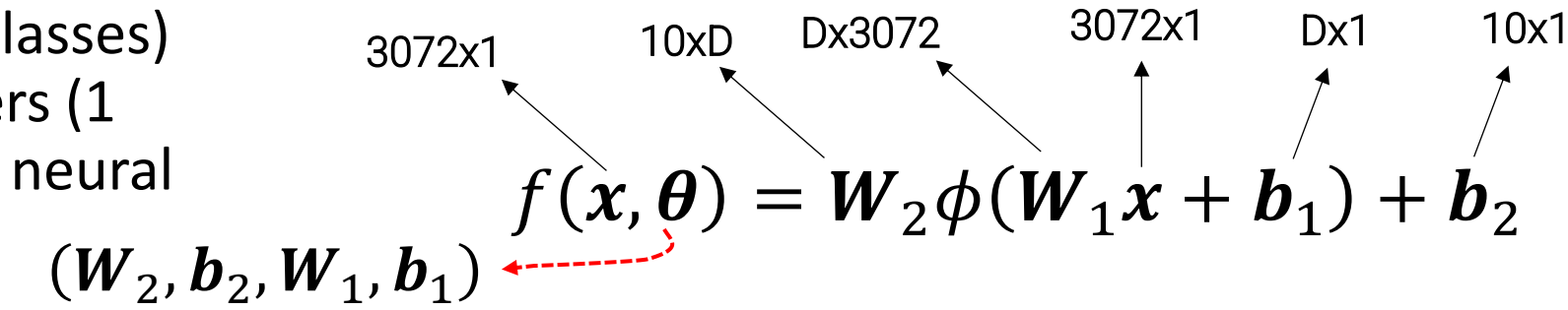


Can we realize the transforms as *linear layers* ? A chain of linear layers is equivalent to a single linear layer (i.e. a shallow classifier !). Thus, to realize representation learning effectively, we need to introduce non-linear computations.

Neural Networks



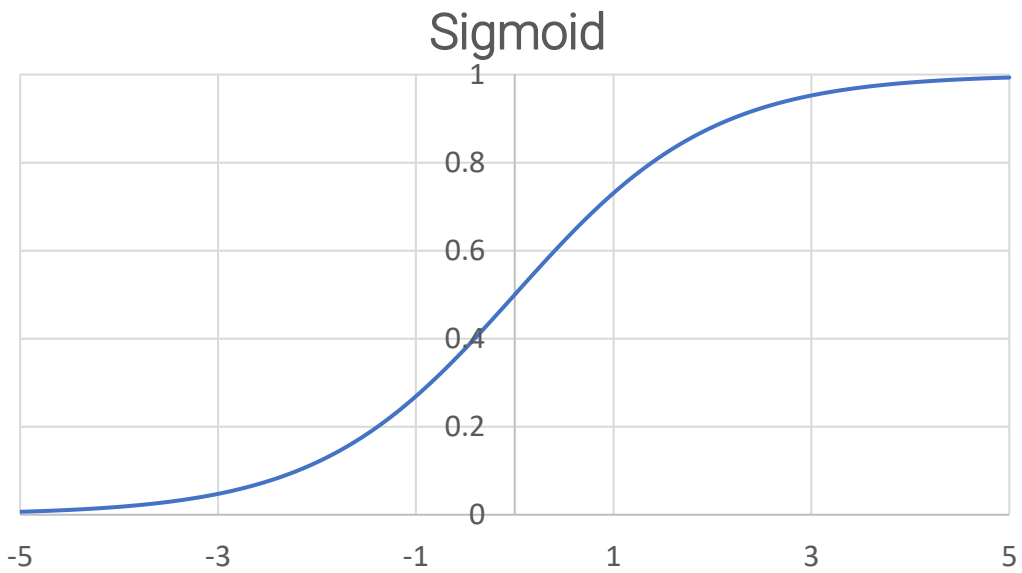
e.g. our image classifier (10 classes) may be realized as a two layers (1 hidden layer, 1 output layer) neural network:



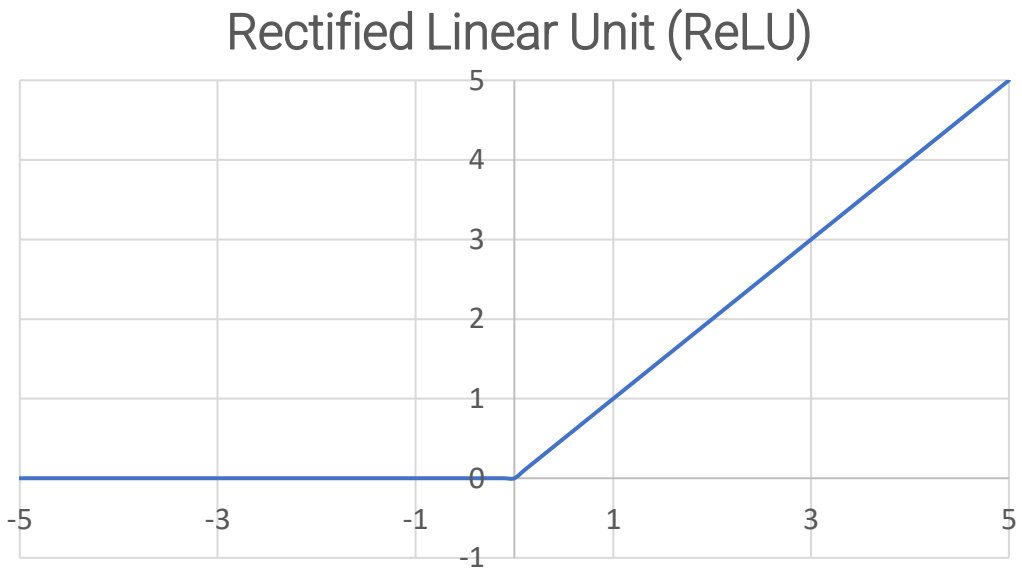
The size (D) of the hidden layer (*representation*) is a hyper-parameter

Activation Functions

A non-linear function applied element-wise to its input tensor

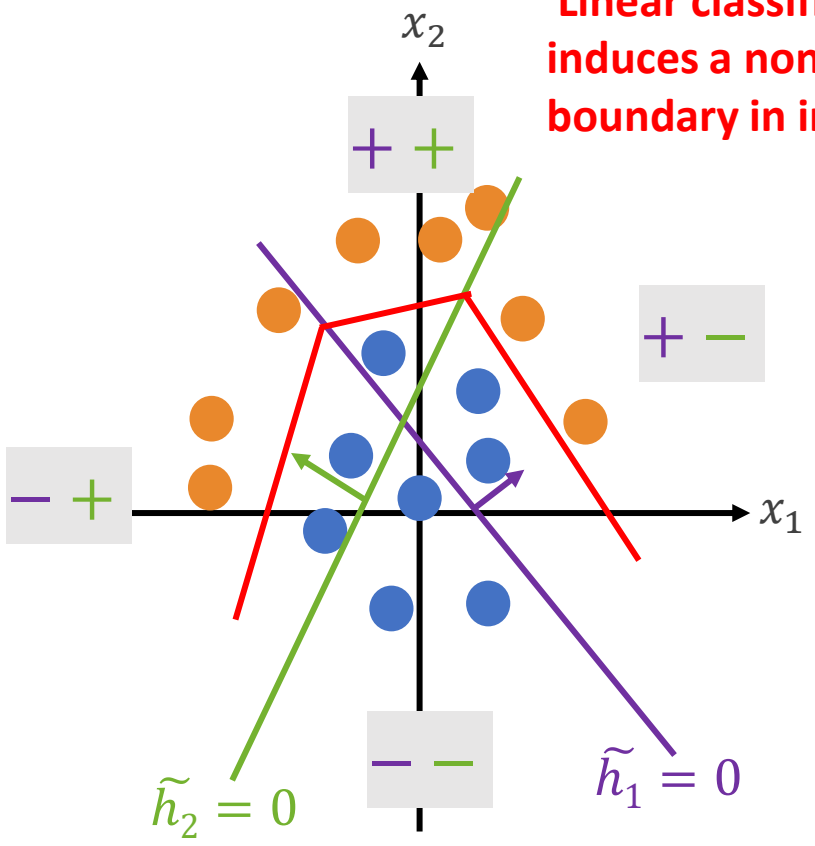


$$\boldsymbol{\phi}(\boldsymbol{a}) = \frac{\mathbf{1}}{\mathbf{1} + \mathbf{exp}(-\boldsymbol{a})}$$



$$\boldsymbol{\phi}(\boldsymbol{a}) = \mathbf{max}(\mathbf{0}, \boldsymbol{a})$$

ReLU at work

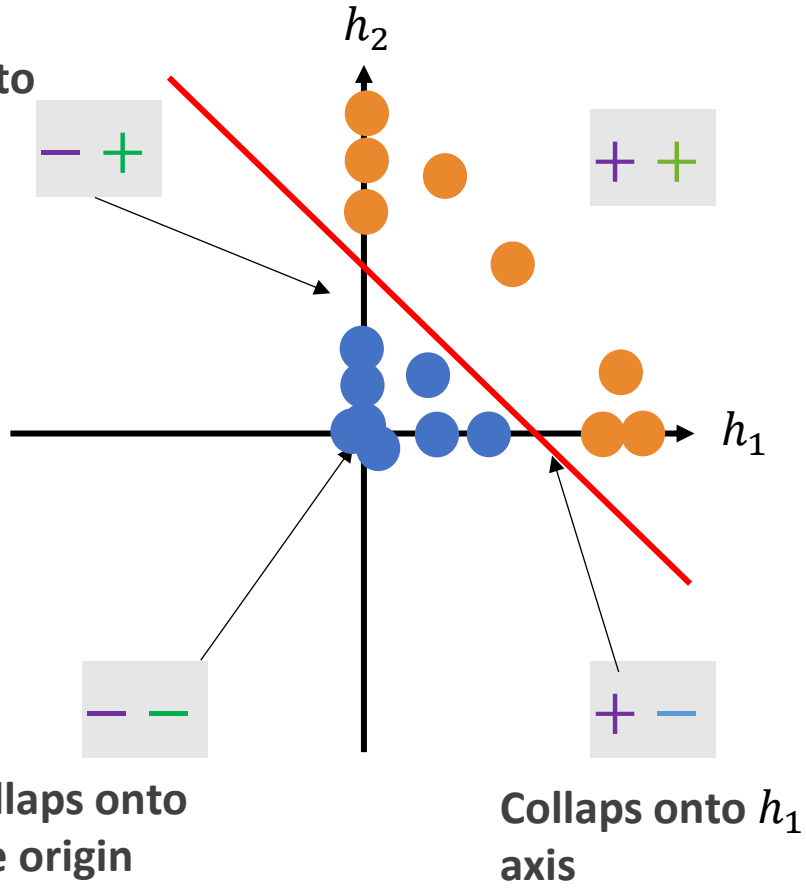


Linear classifier in hidden space induces a non-linear decision boundary in input space.

$$h = \max(0, Wx + b)$$

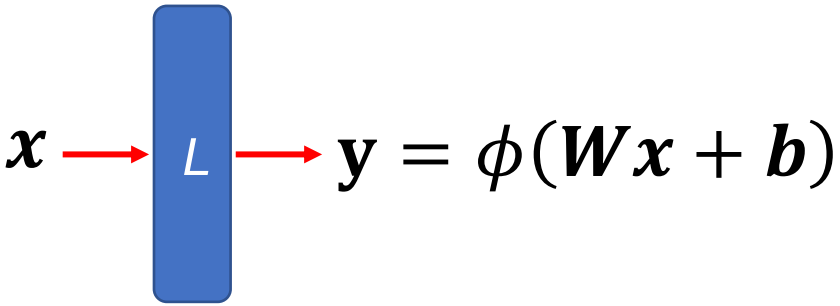
$$\tilde{h}_1 = w_{11}x_1 + w_{12}x_2 + b_1$$
$$\tilde{h}_2 = w_{21}x_1 + w_{22}x_2 + b_2$$

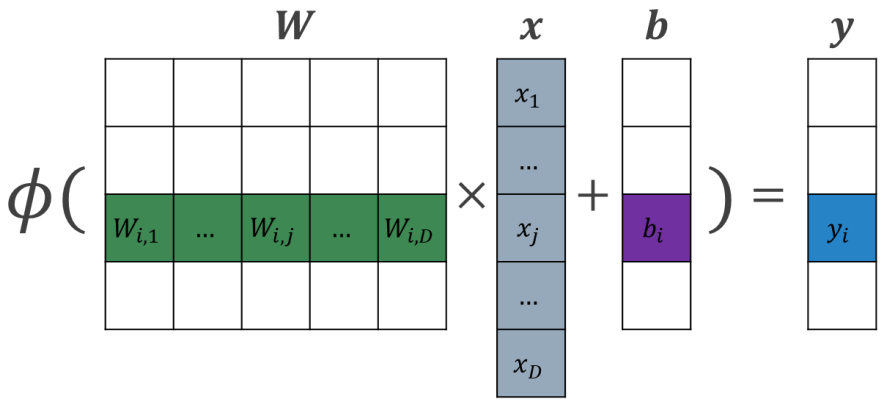
Data linearly separable in hidden space



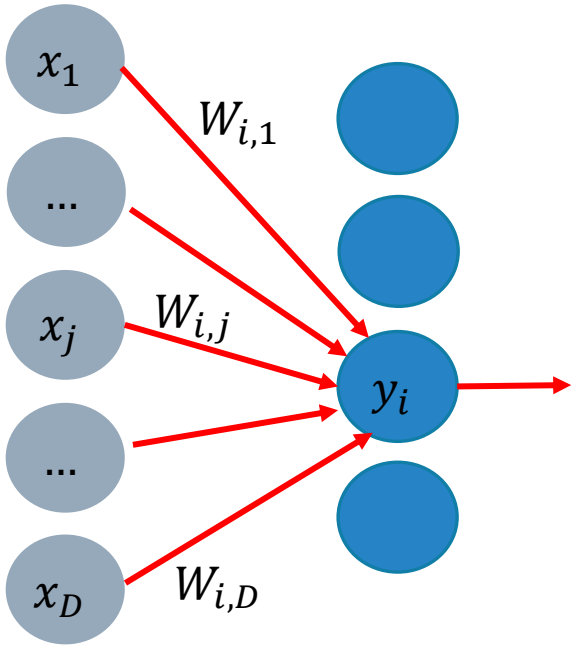
Data non-linearly separable in input space

Fully Connected Layers & MLP





$$y_i = \phi\left(\sum_{j=1}^{D} W_{i,j}\ x_j + b_i\right)$$



Each input unit is connected to each output unit: Fully-Connected (FC) layer.

Units are called *neurons* because they loosely model those present in a biological brain.

A neural network consisting of two or more FC layers is usually referred to as a Multi-Layer Perceptron (MLP).