

MACHINE LEARNING FOR COMPUTER VISION IN AI COURSE

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



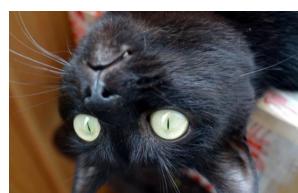
Output
Choose one among a set of classes

Dog
Cat
Bird
Frog
Person

A very challenging task !



Intra-class variations



Viewpoint variations



Occlusions



Illumination changes



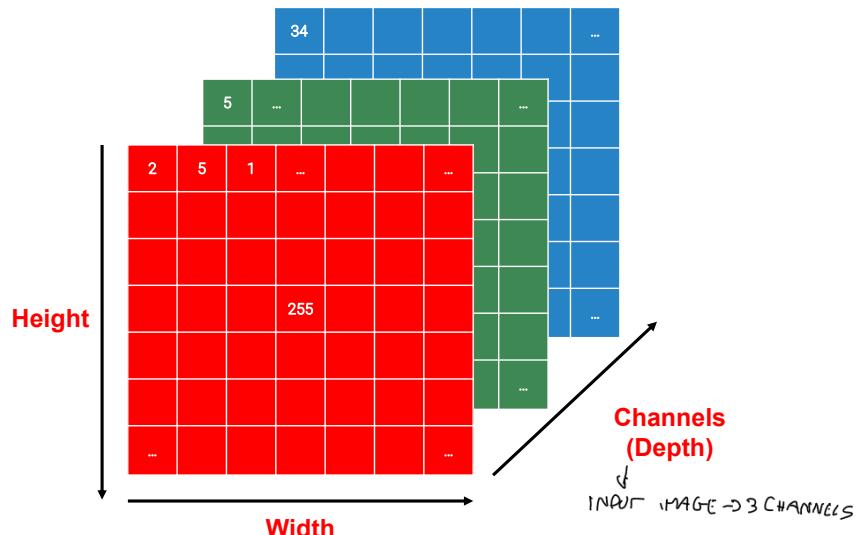
Background clutter



Weirdness of the world...

Colour images are tensors in computers

↳ MULTIDIMENSIONAL MATRICES



Classes as numbers (labels)



$$f(\text{bird image}) = 2$$

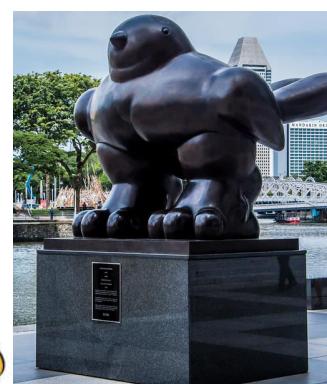


- 0 -> Dog
- 1 -> Cat
- 2 -> Bird**
- 3 -> Frog
- 4 -> Person

The output is a categorical value (?)

INPUT PIXEL, OUTPUT VALUE. With classical computer vision is impossible because there is too much variability in an image of a class, so we can't build a model.

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



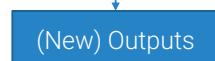
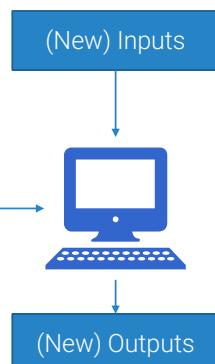
We give or input the image and the label to a training algorithm.

It's an optimization model, but the model works well also for other data.

In general an optimization model is built to work only with some data.



Testing (aka inference, prediction)



Learning the classification function is possible



Dog



Cat



$$f(\text{bird}) = 2$$

Bird



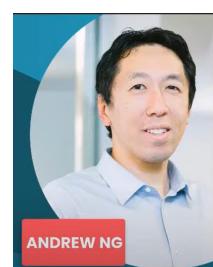
Why is possible to build this function, that is our training algorithm?

A data-centric paradigm

they must be a lot and good



- 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)



ANDREW NG

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

MLOps → STANDARDIZE OPERATIONS FOR MACHINE LEARNING

Modified NIST (MNIST)



TRAINING IMAGES → used for training



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

[MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges](#)

↳ TURING AWARD (NOBEL IN COMP. SCIENCE)

CIFAR 10



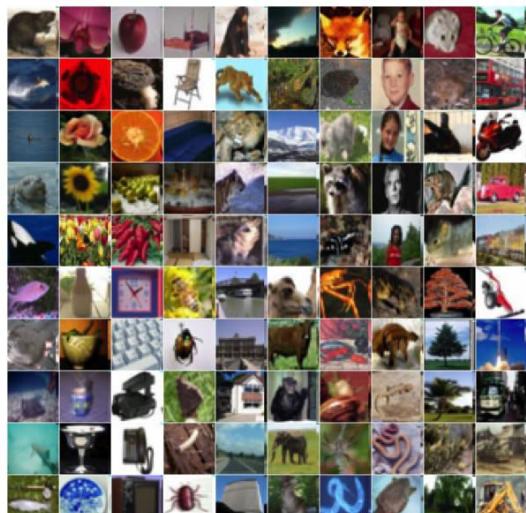
| | |
|------------|--|
| airplane | |
| automobile | |
| bird | |
| cat | |
| deer | |
| dog | |
| frog | |
| horse | |
| ship | |
| truck | |

- 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>

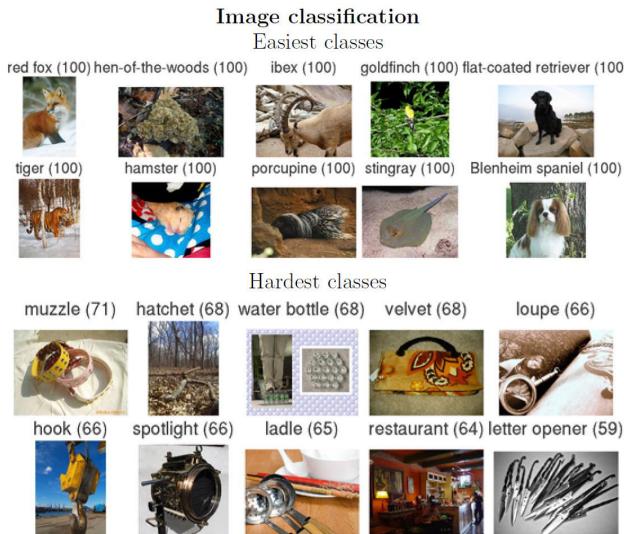
CIFAR 100



- 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.](#)



- **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 \(image-net.org\)](https://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/imagenet/>
Russakovsky et al, "ImageNet Large Scale Visual Recognition Challenge", IJCV 2015

And many more....

Benchmarks

<https://paperswithcode.com/sota>

[Add a Result](#)

| TREND | DATASET | BEST METHOD | PAPER TITLE | PAPER | CODE | COMPARE |
|-------|-----------|---|--|-------|------|-------------------------|
| | ImageNet | 🏆 FixEfficientNet-L2 | Fixing the train-test resolution discrepancy: FixEfficientNet | | | See all |
| | CIFAR-10 | 🏆 BiT-L (ResNet) | Big Transfer (BiT): General Visual Representation Learning | | | See all |
| | CIFAR-100 | 🏆 BiT-L (ResNet) | Big Transfer (BiT): General Visual Representation Learning | | | See all |
| | MNIST | 🏆 Branching/Merging CNN + Homogeneous Filter Capsules | A Branching and Merging Convolutional Network with Homogeneous Filter Capsules | | | See all |
| | SVHN | 🏆 WideResNet-28-10 | RandAugment: Practical automated data augmentation with a reduced search space | | | See all |
| | STL-10 | 🏆 NAT-M4 | Neural Architecture Transfer | | | See all |

What Machine Learning is about ?

- In Machine Learning we are given:

INPUT → IMAGE → OUTPUT → LABEL

- a training set, $D^{train} = \{(\mathbf{x}^{(i)}, y^{(i)}) \mid i = 1, \dots, N\}$ → GROUND TRUTH

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

↳ THE WORD, THE FIELD WITH WHICH WE APPLY MACHINE LEARNING

where $\mathbf{x}^{(i)} \in \mathbb{R}^f$, 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 $p_{data}(\mathbf{x}, y)$
- The objective of a machine learning model is to make use of the samples in D^{train} in order to find a function, $y = f(\mathbf{x})$, that can output correct predictions on **unseen** input data, i.e. those in D^{test} (those that don't belong to training set).
- Typically, $f(\mathbf{x})$, is assumed to belong to a chosen **parametric family of functions**, $f(\mathbf{x}, \boldsymbol{\theta})$, 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 an **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 chooses an objective function, known 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 consists in finding the parameter values, θ^* , that minimize the Loss Function on the training set:

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

- 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:

$$L(\theta, D^{train}) = \frac{1}{N} \sum_i L(\theta, (x^{(i)}, y^{(i)}))$$

parameters are known learning.
What kind of function to use is
on a priori choose. we have to
choose the best parameters.

NOT A GLOBAL FUNCTION, IT CAN BE
SEEN AS THE SUM OF INVOLVED
LOSSES

ALL THE LOSS IS THE AVERAGE
OF THAT

FOR AN IMAGE

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, $f(x, \theta)$, with $\theta \in \Theta$. 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 D^{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.

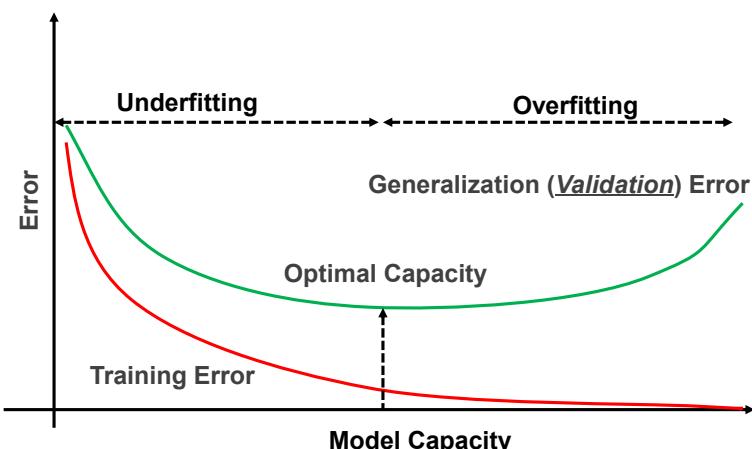
SUBSET THAT WE DON'T USE FOR TRAINING, WE USE IT TO COMPARE 2 MODELS
ALREADY CHOSEN AND TRAINED WITH TRAIN SET.

Underfitting and Overfitting

- When varying model complexity, the training error (red) and the generalization (validation) error (green) usually follow the trend shown in the Figure.

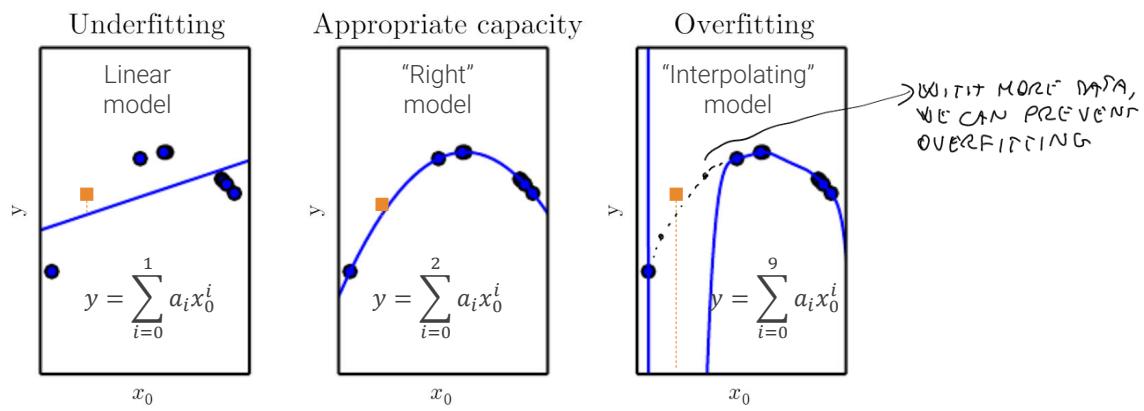
- To obtain the best generalization we want the model to work in the “sweet spot” where the training error is small and the gap between the training and validation error is also small.

- 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 algorithm is **overfitting** the training data.



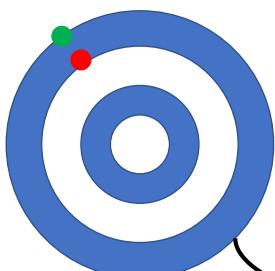
Example: Polynomial Regression

We randomly generate N data points computed by a quadratic function $y = (ax_0^2 + bx_0 + c)$ and fit them with different polynomials of degree K ($y = \sum_{i=0}^K a_i x_0^i = a_0 + a_1 x_0 + a_2 x_0^2 + a_3 x_0^3 + \dots$) by minimizing an MSE (Mean Squared Error) Loss.

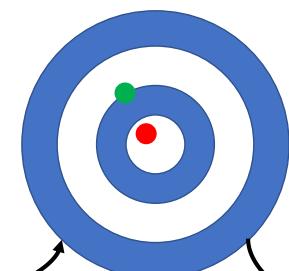


The path to the sweet spot

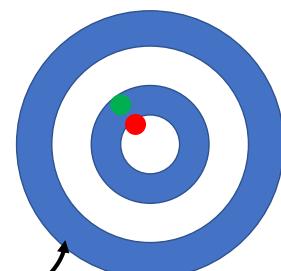
High Training Error
High Generalization Error



Low Training Error
High Generalization Error



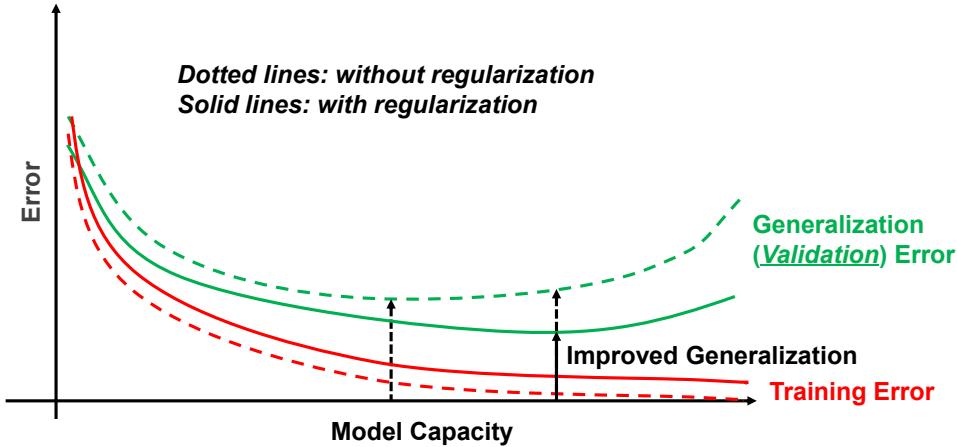
Low Training Error
Low Generalization Error



Increase Capacity
(deeper/wider network)
Train Longer

More Data
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

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

WE VALIDATE IT ON THE VALIDATION SET
Regularizer

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

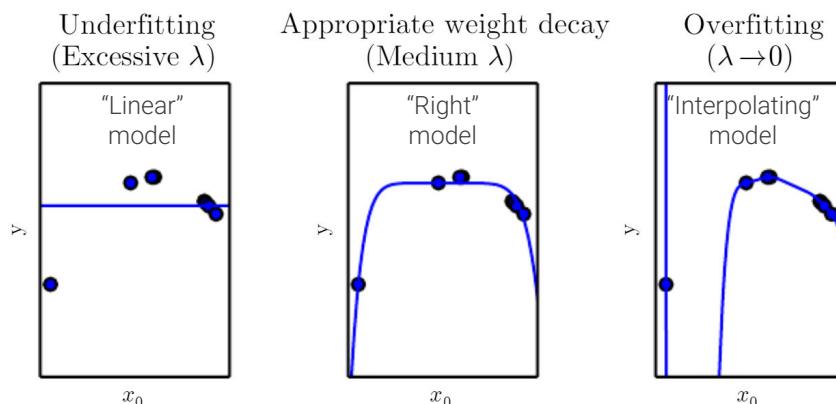
$$L^{reg}(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_2^2 = \sum_i \theta_i^2 \quad (\text{L}_2 \text{ Regularization, Weight Decay})$$

$$L^{reg}(\boldsymbol{\theta}) = \|\boldsymbol{\theta}\|_1 = \sum_i |\theta_i| \quad (\text{L}_1 \text{ Regularization})$$

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 .



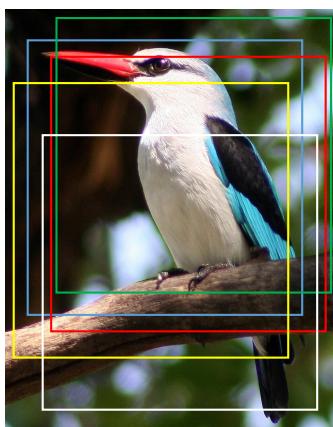
Data Augmentation (1) works very well in CV



- 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.

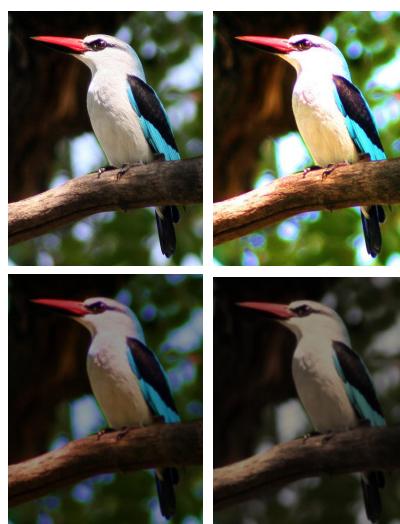


Data Augmentation (2)

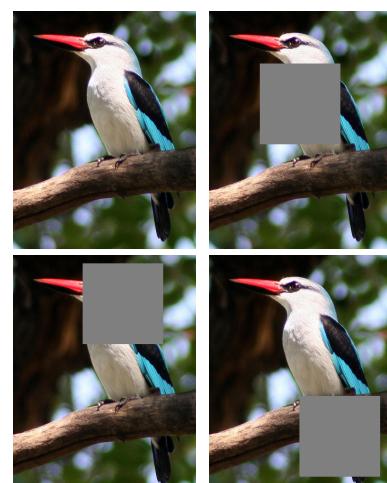


Random Crop (Multi-Scale)

Different augmentations can be combined together.



Colour Jittering



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

How to design a *linear classifier* ?



Dog



Cat



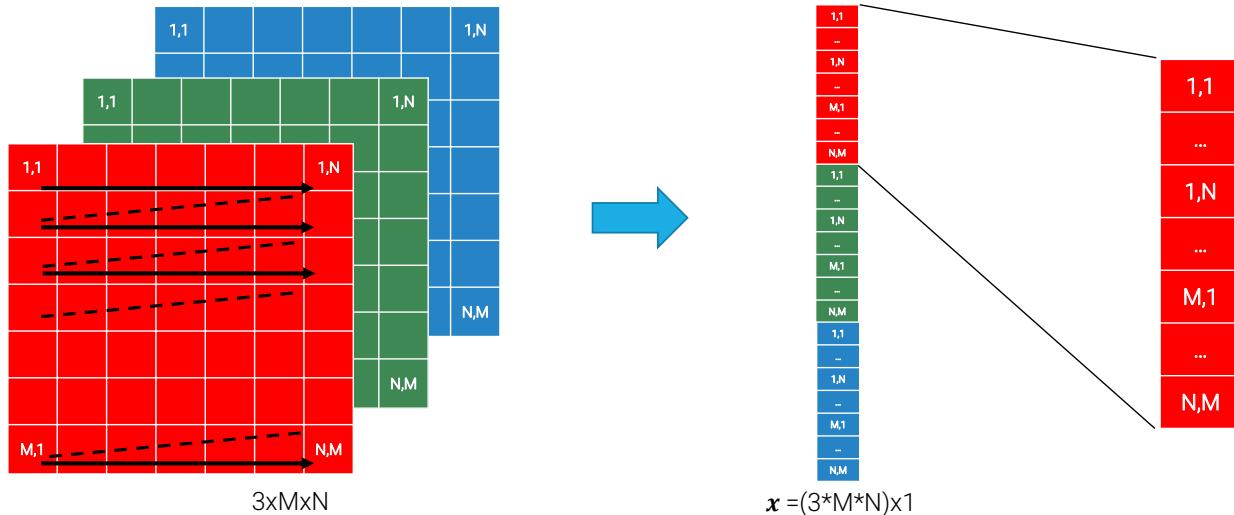
Bird



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



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



A bad linear classifier

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



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 $\text{cat}=3$ and we get 2.5 out of f , it does not mean that x depicts a fantastic beast looking half-bird and half-cat).

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

$32 \times 32 \times 3 = 3072 \times 1$ CIFAR image

1×3072

1×1 (Scalar)

A better one

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



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

$\xrightarrow{\arg\max} 2$

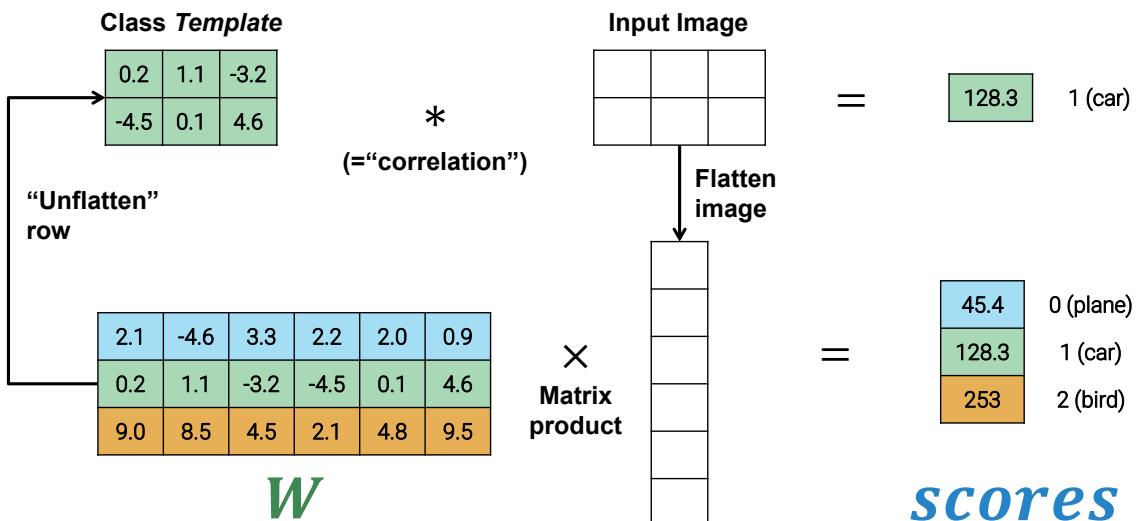
$$f(x; W) = Wx = \text{scores}$$

$32 \times 32 \times 3 = 3072 \times 1$ CIFAR image

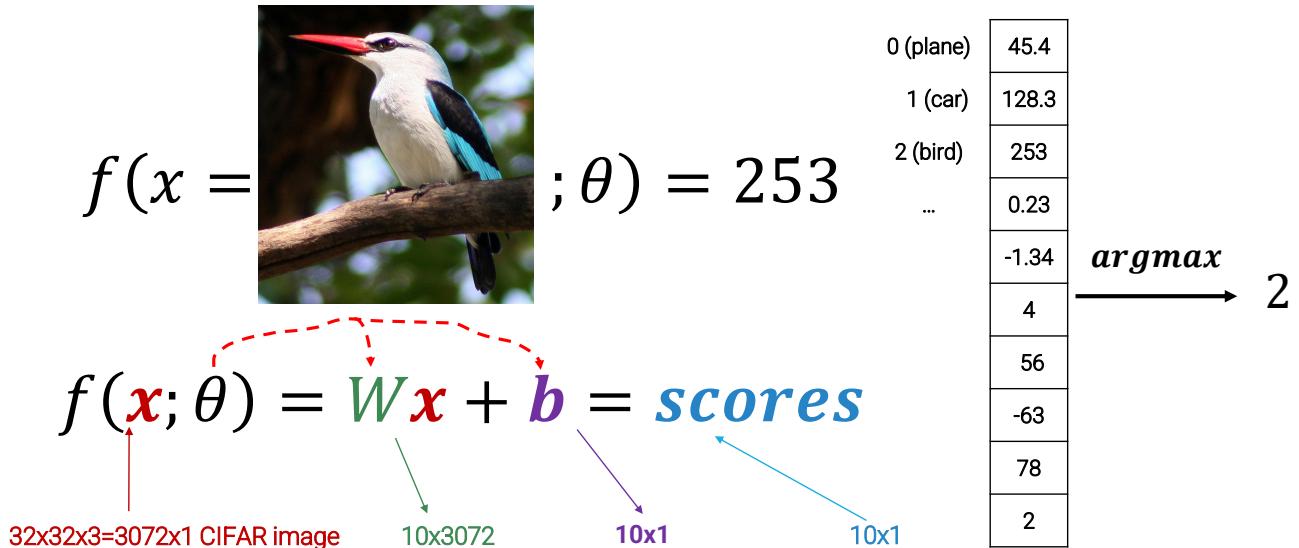
10×3072

10×1

Linear Classification as Template Matching



Linear means affine, actually.



Back to Loss Functions

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

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

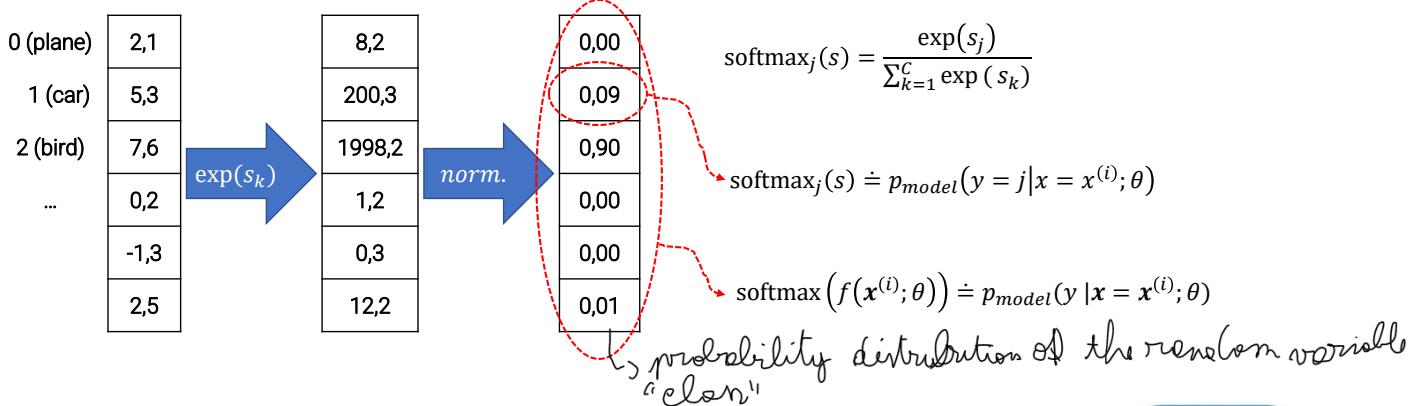
typically obtained as a sum (average) of terms computed on the individual data samples:

$$L(\theta, D^{train}) = \sum_i L(\theta, (\mathbf{x}^{(i)}, \mathbf{y}^{(i)}))$$

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. $(x^{(i)}, y^{(i)} = j) \in D^{\text{train}}$?

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

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

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

(Cross-entropy Loss)

$$\mathbb{I}(y^{(i)}) = \begin{cases} 0 \\ 0 \\ 0 \\ \vdots \\ j \\ 0 \\ 0 \end{cases}$$

One-hot Encoding

| | |
|-----------|------|
| 0 (plane) | 0,00 |
| 1 (car) | 0,09 |
| 2 (bird) | 0,90 |
| ... | 0,00 |
| | 0,00 |
| | 0,01 |

$$-\log(0.09) = 2.4 \text{ (high)}$$

$$-\log(0.9) = 0.1 \text{ (low)}$$

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.

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

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

$$= \arg \max_{\theta} \prod_{i=1}^N p_{\text{model}}(y = y^{(i)} | x = x^{(i)}; \theta) \quad (\text{i.i.d.})$$

$$= \arg \max_{\theta} \sum_{i=1}^N \log p_{\text{model}}(y = y^{(i)} | x = x^{(i)}; \theta) \quad (\text{avoid the curse of products})$$

$$= \arg \min_{\theta} \sum_{i=1}^N -\log p_{\text{model}}(y = y^{(i)} | x = x^{(i)}; \theta) \quad (\text{Cross-entropy Loss})$$

Why Cross-entropy ?

N J



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:

$$\begin{aligned} H(\mathbb{I}(y^{(i)}), p_{model}(y|x^{(i)}; \theta)) &= - \sum_{k=1}^C \mathbb{I}(y^{(i)})_k \log p_{model}(y=k|x^{(i)}; \theta) \\ &= -\log p_{model}(y=y^{(i)}|x^{(i)}; \theta) \quad \text{(Per-sample Cross-entropy Loss)} \end{aligned}$$

Putting all together

N J



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

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

and the per-sample loss is given by

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

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

due to $\exp(s_k)$ 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

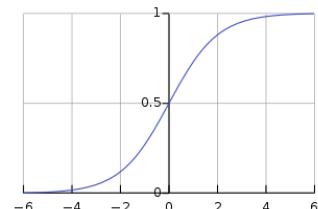
N Q



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, $\sigma(s)$.

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

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



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

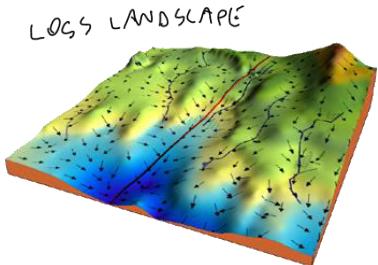
$$L(\theta, (x^{(i)}, y^{(i)})) = - \left(y^{(i)} \log p_{model}(y=y^{(i)}|x=x^{(i)}; \theta) + (1-y^{(i)}) \log (1 - p_{model}(y=y^{(i)}|x=x^{(i)}; \theta)) \right)$$

(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:

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

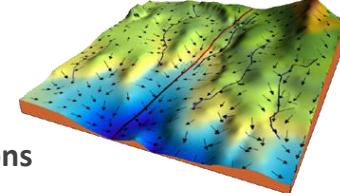


One could compute the Loss for many randomly chosen parameter values and then pick the choice yielding the lowest loss. However, this “brute force” approach would be rather inefficient, especially in high-dimensional parameter spaces. A much better strategy consist in starting from a random choice and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

Gradient Descent

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

Gradient of the Loss
wrt
the parameters



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

for $e = 1, \dots, E$ epochs

1. Forward pass: classify all the training data to get the predictions

$\hat{y}^{(i)} = f(x^{(i)}; \theta^{(e-1)})$ and the loss $L(\theta^{(e-1)}, D^{\text{train}})$

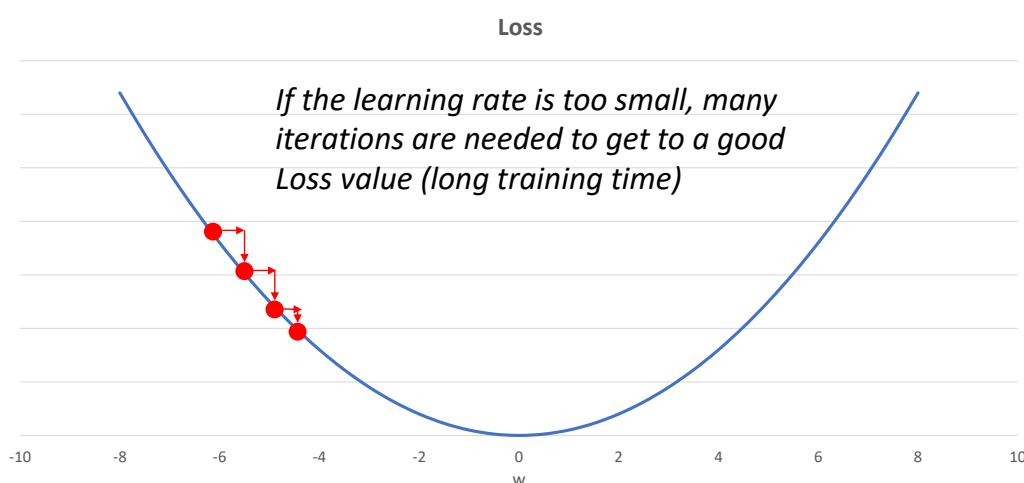
2. Backward pass: compute the gradient $\nabla L = \nabla L(\theta^{(e-1)}, D^{\text{train}})$ (Backpropagation Algorithm)

3. Parameters Update Step: $\theta^{(e)} = \theta^{(e-1)} - \alpha \nabla L$

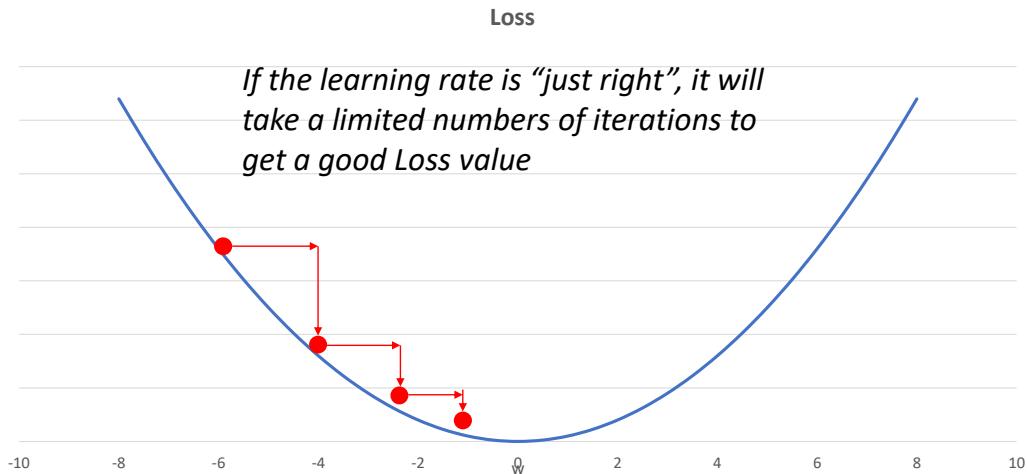
• a single update set requires processing all the Training set.

Learning Rate \rightarrow hyper parameter, critical.
if it's small \rightarrow Time $\uparrow\uparrow$
large \rightarrow get stuck

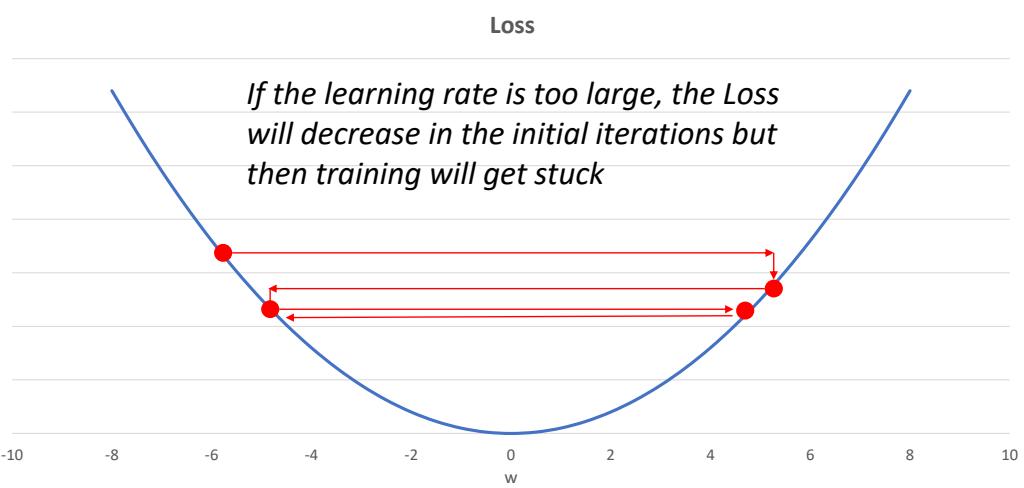
Learning Rate and Gradient Descent (1)



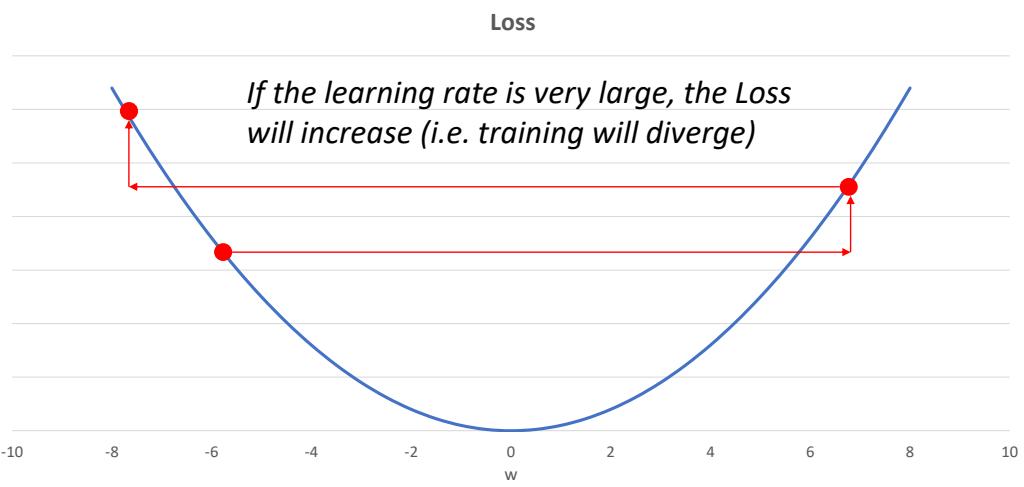
Learning Rate and Gradient Descent (2)

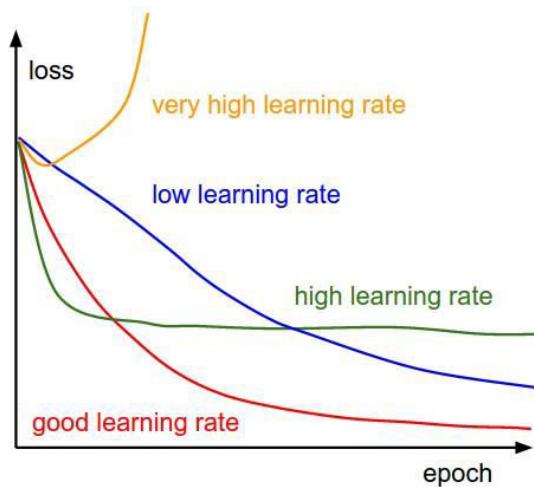


Learning Rate and Gradient Descent (3)



Learning Rate and Gradient Descent (4)





Limits of (Batch) Gradient Descent

$$L(\theta, D^{train}) = \sum_i L(\theta, (x^{(i)}, y^{(i)})) \quad \rightarrow \quad \nabla L(\theta, D^{train}) = \sum_i \nabla L(\theta, (x^{(i)}, y^{(i)}))$$

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 **Stochastic 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 D^{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: $U = \left\lceil \frac{N}{B} \right\rceil$.

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

for $e = 0, \dots, E - 1$ epochs

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

for $u = 0, \dots, U - 1$ *minibatches*

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

3. Backward pass: compute the gradient $\nabla L = \nabla L(\theta^{(e*U+u)}, (X, \hat{Y}))$

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

Larger batches provide smoother approximations of the gradient at the cost of higher memory requirements. Typical sizes are (powers of 2): $B=16\dots256$.

Beyond “plain vanilla” SGD

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

$$\beta \in [0,1], v^{(0)} = 0$$
$$\begin{cases} v^{(t+1)} = \beta v^{(t)} - \alpha \nabla L(\theta^{(t)}) \\ \theta^{(t+1)} = \theta^{(t)} + v^{(t+1)} \end{cases}$$
$$(0) \begin{cases} v^{(1)} = -\alpha \nabla L(\theta^{(0)}) \\ \theta^{(1)} = \theta^{(0)} - \alpha \nabla L(\theta^{(0)}) \end{cases}$$
$$(1) \begin{cases} v^{(2)} = -\beta \alpha \nabla L(\theta^{(0)}) - \alpha \nabla L(\theta^{(1)}) \\ \theta^{(2)} = \theta^{(1)} - \beta \alpha \nabla L(\theta^{(0)}) - \alpha \nabla L(\theta^{(1)}) \end{cases}$$
$$(2) \begin{cases} v^{(3)} = -\beta^2 \alpha \nabla L(\theta^{(0)}) - \beta \alpha \nabla L(\theta^{(1)}) - \alpha \nabla L(\theta^{(2)}) \\ \theta^{(3)} = \theta^{(2)} - \beta^2 \alpha \nabla L(\theta^{(0)}) - \beta \alpha \nabla L(\theta^{(1)}) - \alpha \nabla L(\theta^{(2)}) \end{cases}$$

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.

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

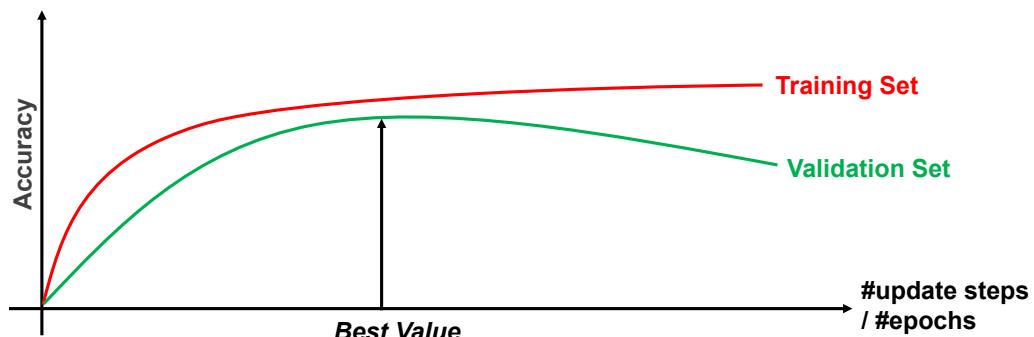
- ADAM (ADaptive Moments) proposes to maintain also a running average of the gradients.

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

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

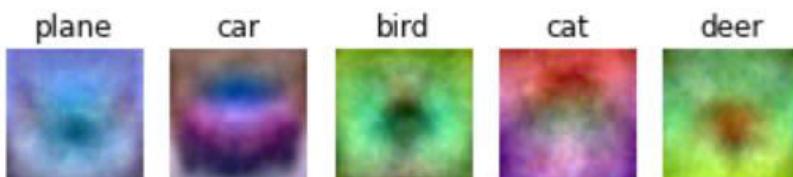
Back to Regularization – Early Stopping

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



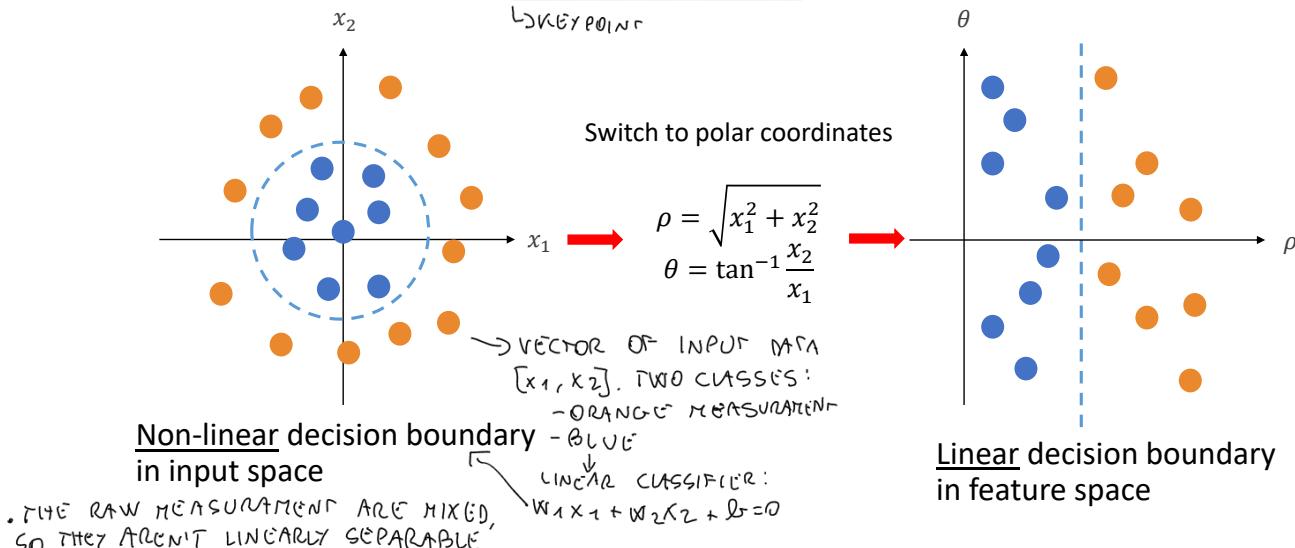
For several classes the most important feature seems to be the background color.

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

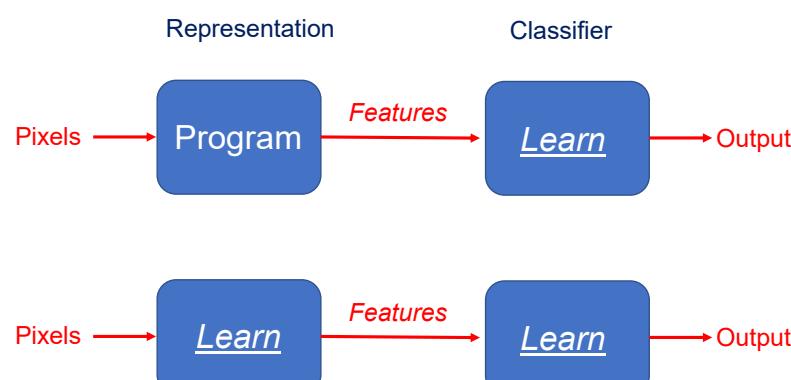
Classifying images using as features the raw pixel data turns out to be a very difficult task !

Representation is important

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

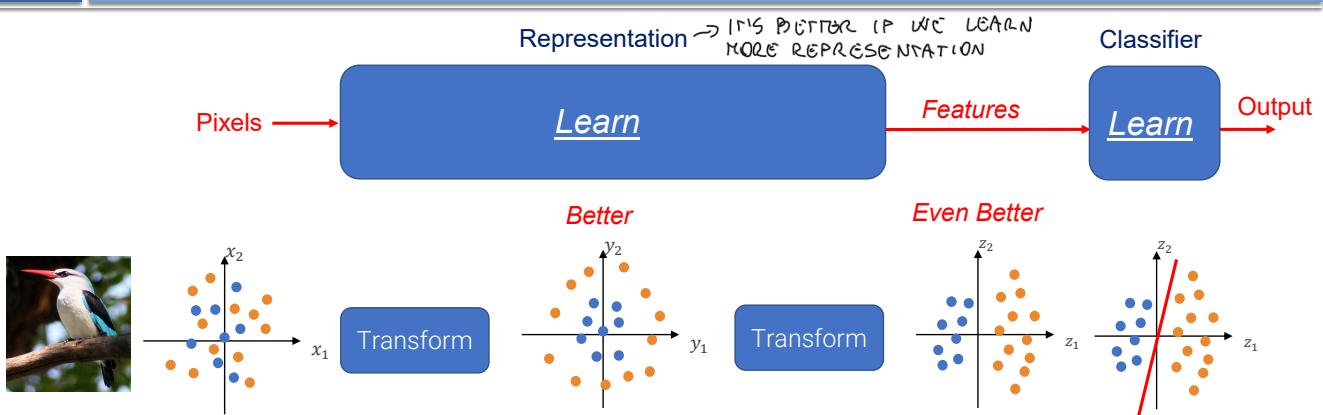


Machine (Shallow) Learning vs. Deep Learning



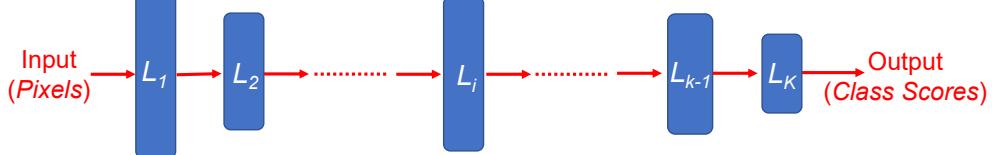
Deep learning ≈ Representation Learning

→ 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



Non-linear function (Activation) $\phi(Wx + b)$

Linear transformation

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

$$(W_2, b_2, W_1, b_1) \quad f(x, \theta) = W_2 \phi(W_1 x + b_1) + b_2 \quad (2 \text{ LAYERS})$$

3072x1 10xD Dx3072 3072x1 Dx1 10x1

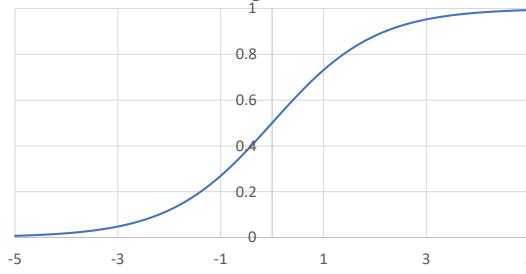
WE MIGHT THINK AS THE LINEAR CLASSIFIER

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

Activation Functions

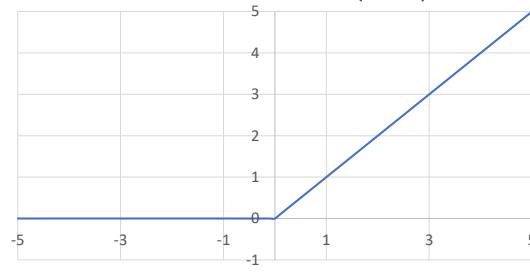
A non-linear function applied element-wise to its input tensor

Sigmoid



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

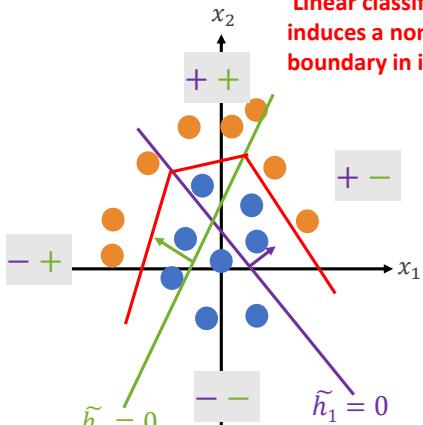
Rectified Linear Unit (ReLU)



$$\phi(a) = \max(0, a)$$

ReLU at work

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



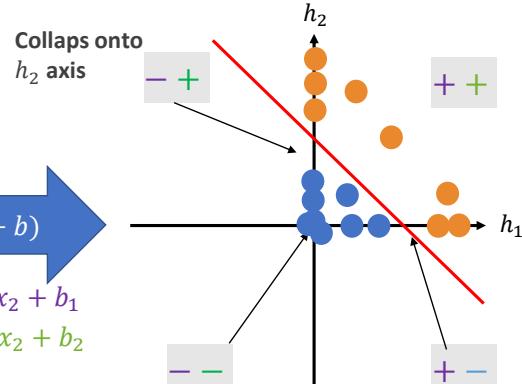
$$h = \max(0, Wx + b)$$

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

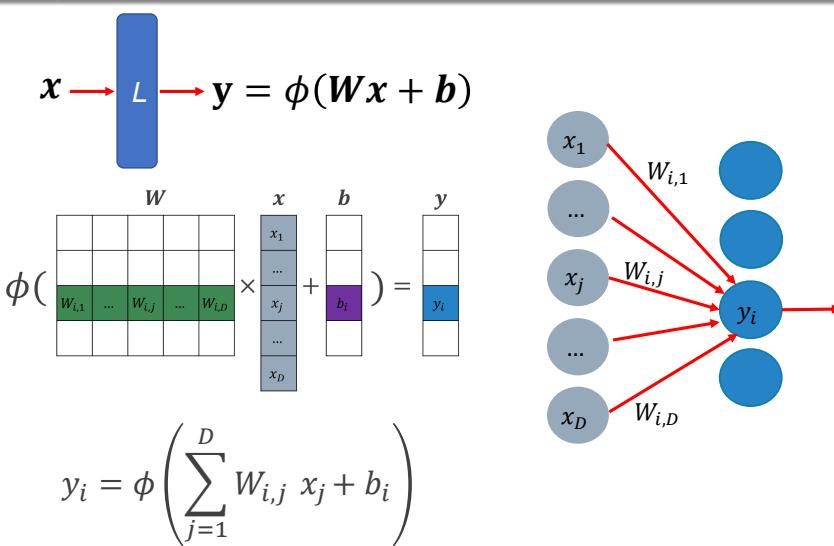
Data linearly separable in hidden space

Collaps onto h_2 axis

Collaps onto the origin



Fully Connected Layers & MLP



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.

BECAUSE IT
WEIGHS ALL
THE INPUT
UNITS

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