Lecture 3
Image Classification

IMAGE PROCESSING AND COMPUTER VISION – PART 2

SAMUELE SALTI

Image Classification

Input

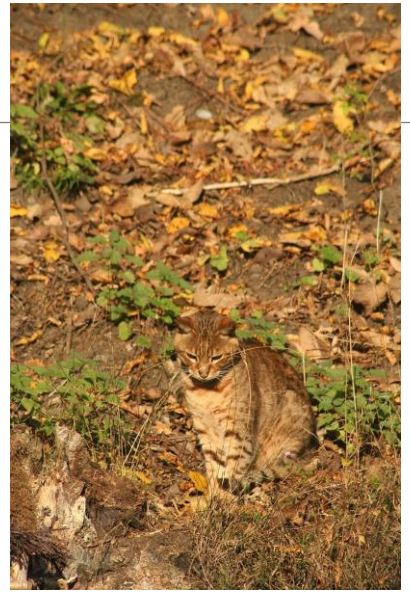
Output
Choose among
these categories

Dog
Cat
Bird
Frog
Person

Some challenges

Intraclass variations

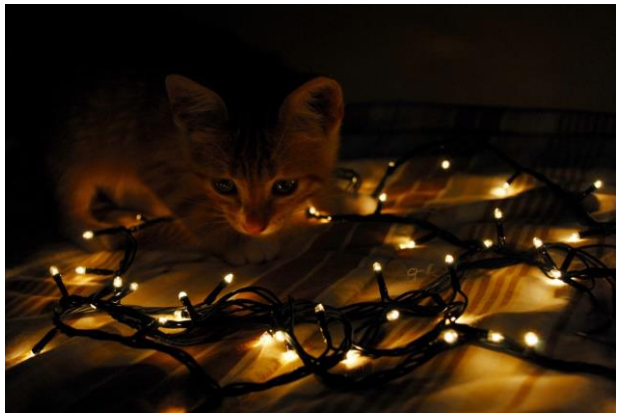
barely visible



Background clutter

Occlusions

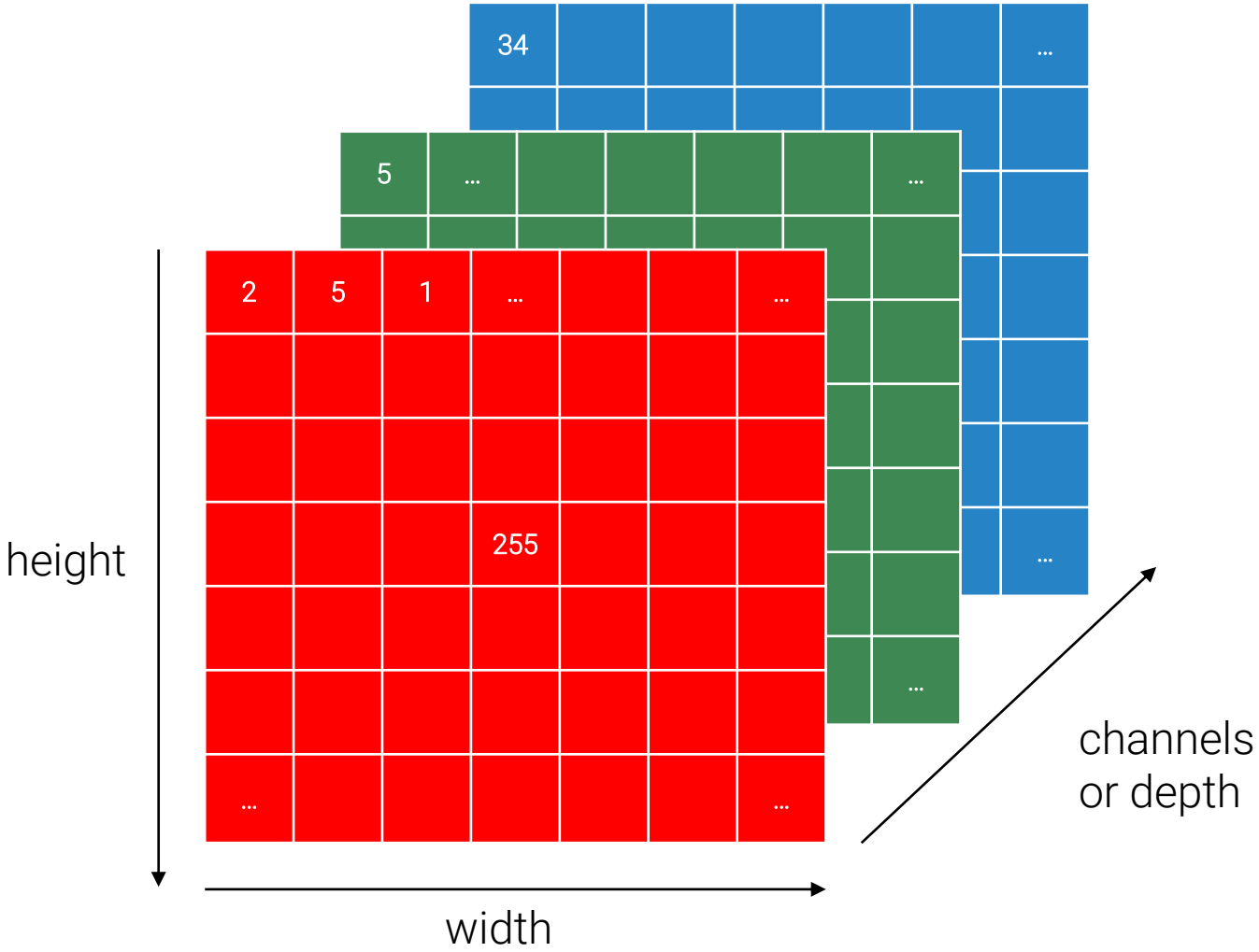
Viewpoint variations



Illumination changes

General weirdness of the world…

RGB images are tensors in a computer



Categories as numbers

$f($  $) = 2$

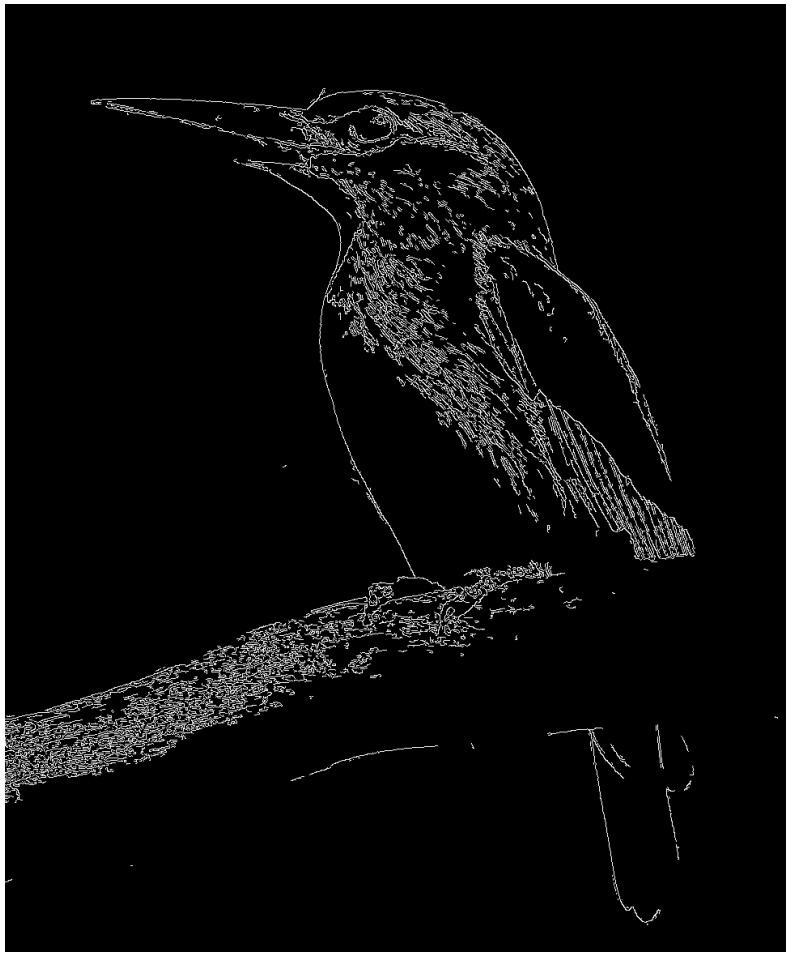
look-up table

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

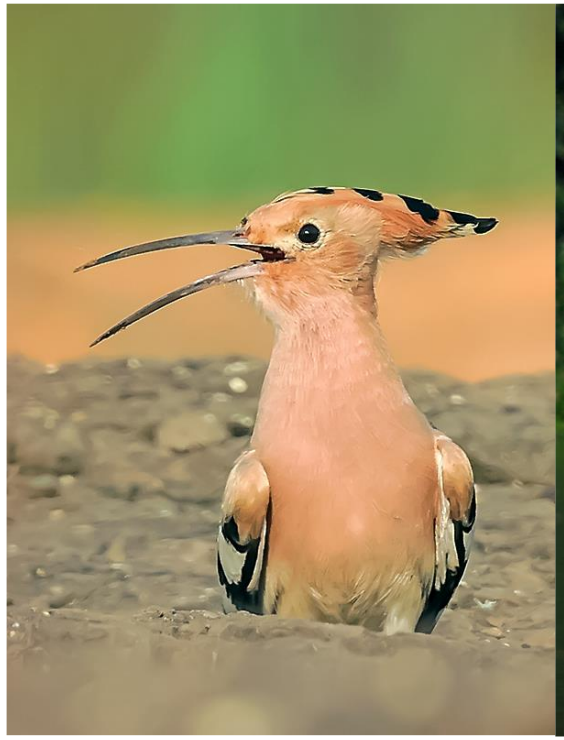
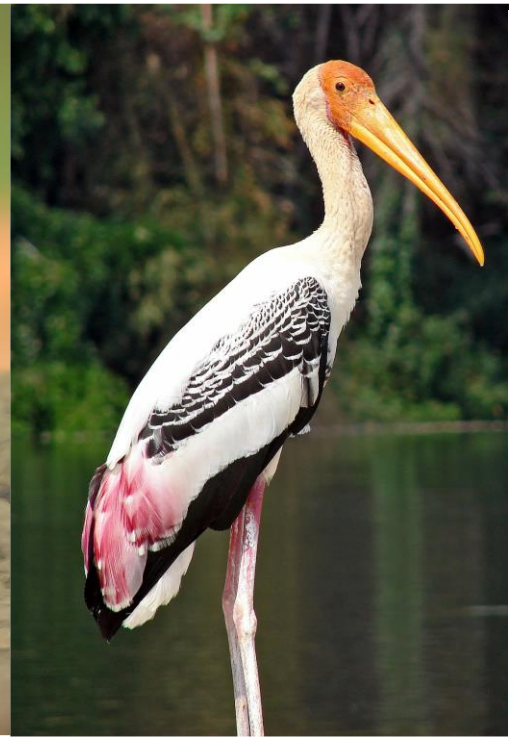
How far can «classic» computer vision bring us?

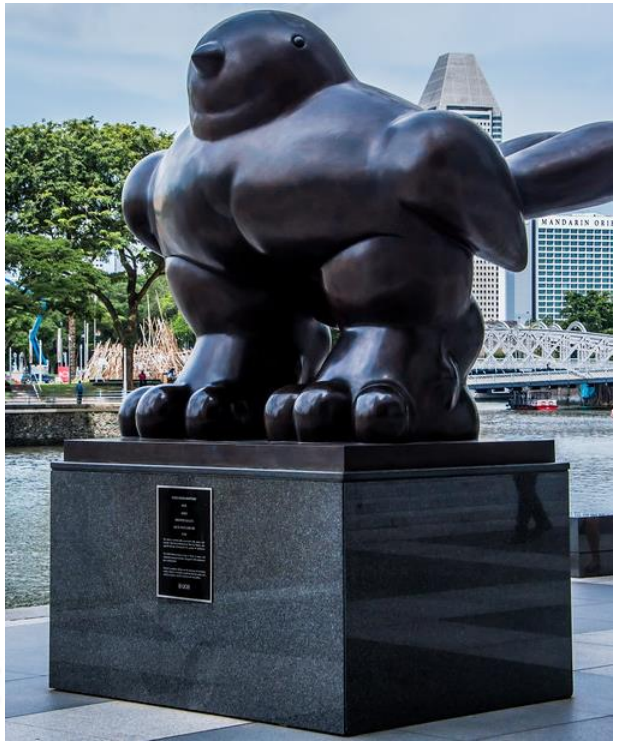
detect_edges(image)

If ???



Birds…

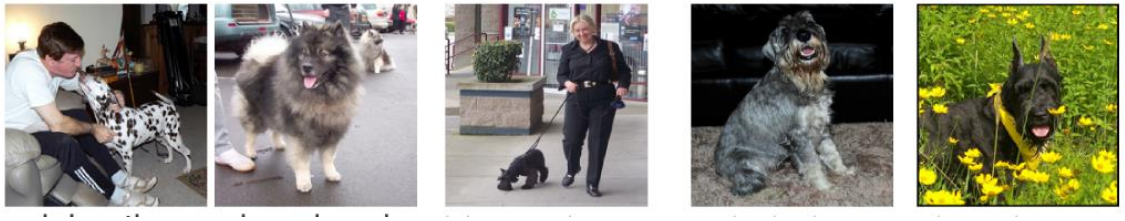





Traditional Computer Vision techniques, e.g. handcrafted rules based on edges, need **a controlled environment**, usually feasible in industrial vision applications, otherwise they are very brittle.

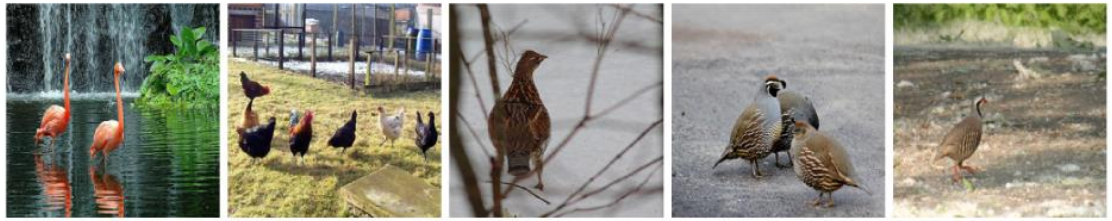
(Supervised) Machine learning to the rescue

Dog



Cat

Bird



…

…

…

$f($  $) = 2$

Training and testing dataset

When applying machine learning methods, we are given (or we create):

- 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 real word **items** we care about (i.e. images in our case), and $y^{(i)}$ are the outputs we want to predict for that item, i.e. the **label** in image classification.

We assume the two sets contains **independent and identically distributed** samples from the **same** unknown distribution $p_{data}(\boldsymbol{x}, y)$

Modified NIST (MNIST)

10 classes: handwritten digits from 0 to 9

50k train images

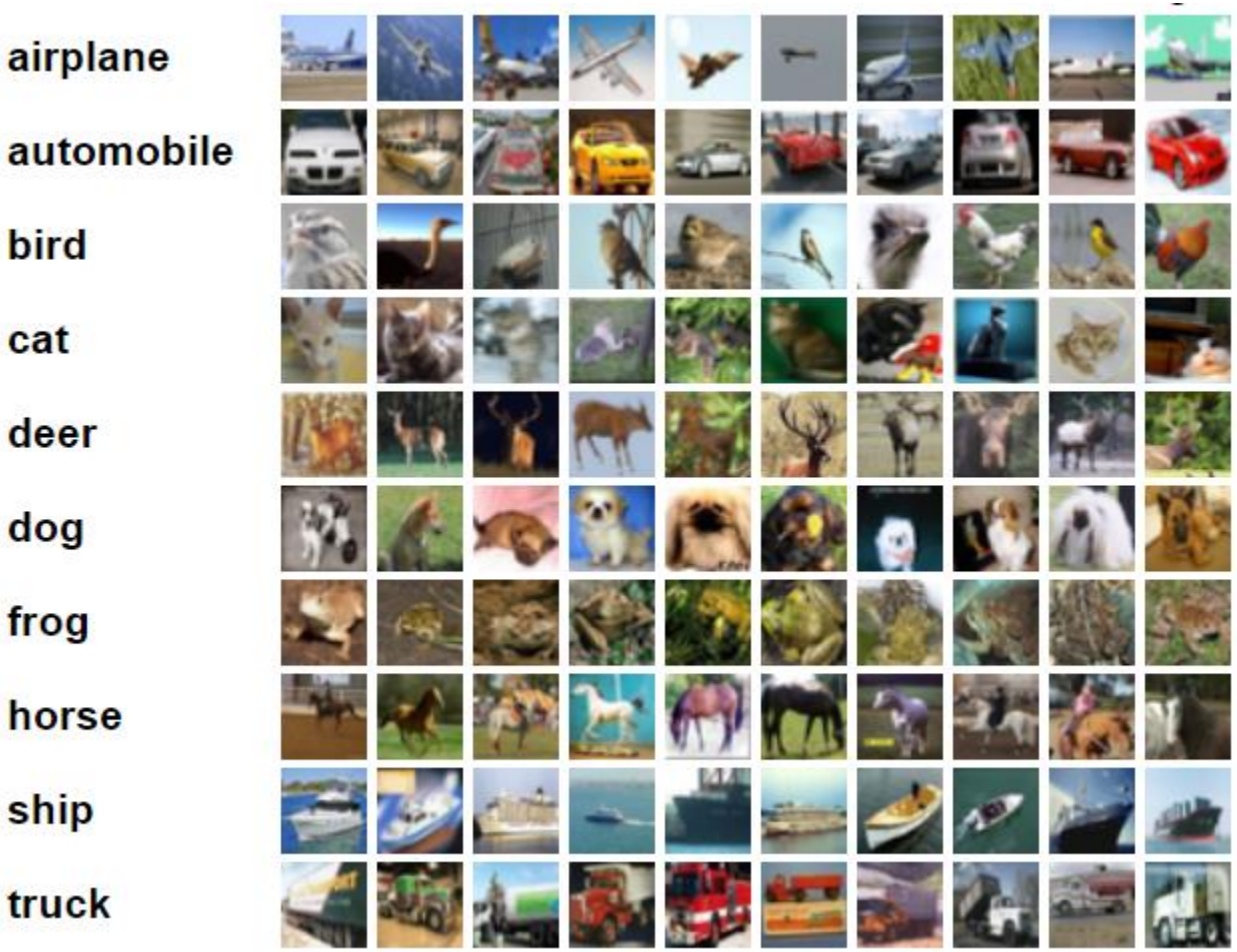
10k test images

28x28 grayscale images

Results on MNIST may not generalize to other datasets

http://yann.lecun.com/exdb/mnist/

CIFAR 10



Subset of the 80 million Tiny Images dataset

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

10 classes

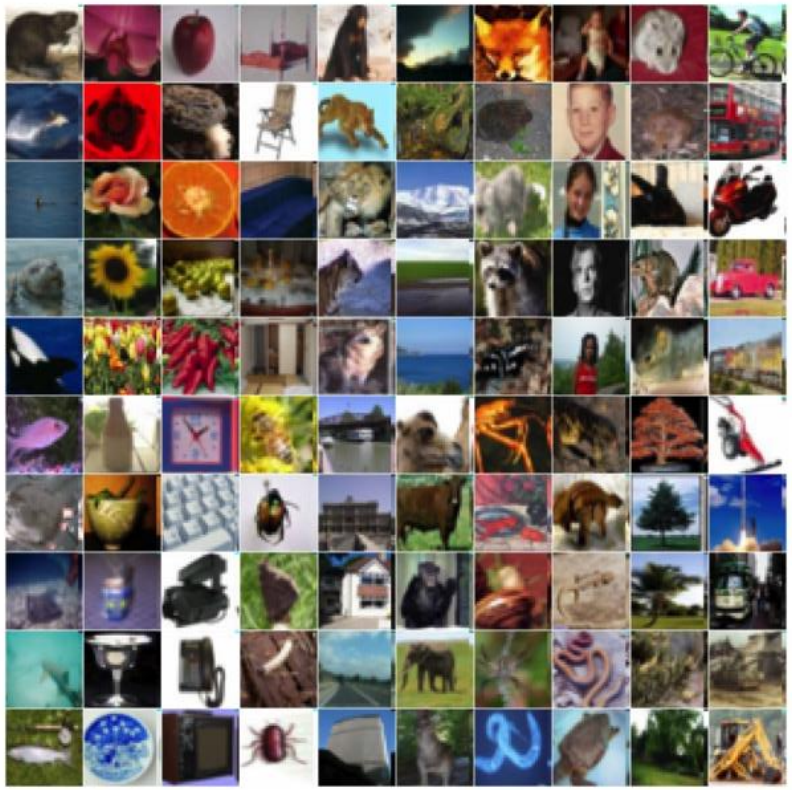
50k training images

10k testing images

32x32 RGB images

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

CIFAR 100



Another subset of the 80 million Tiny Images dataset

100 classes

50k training images (500 per class)

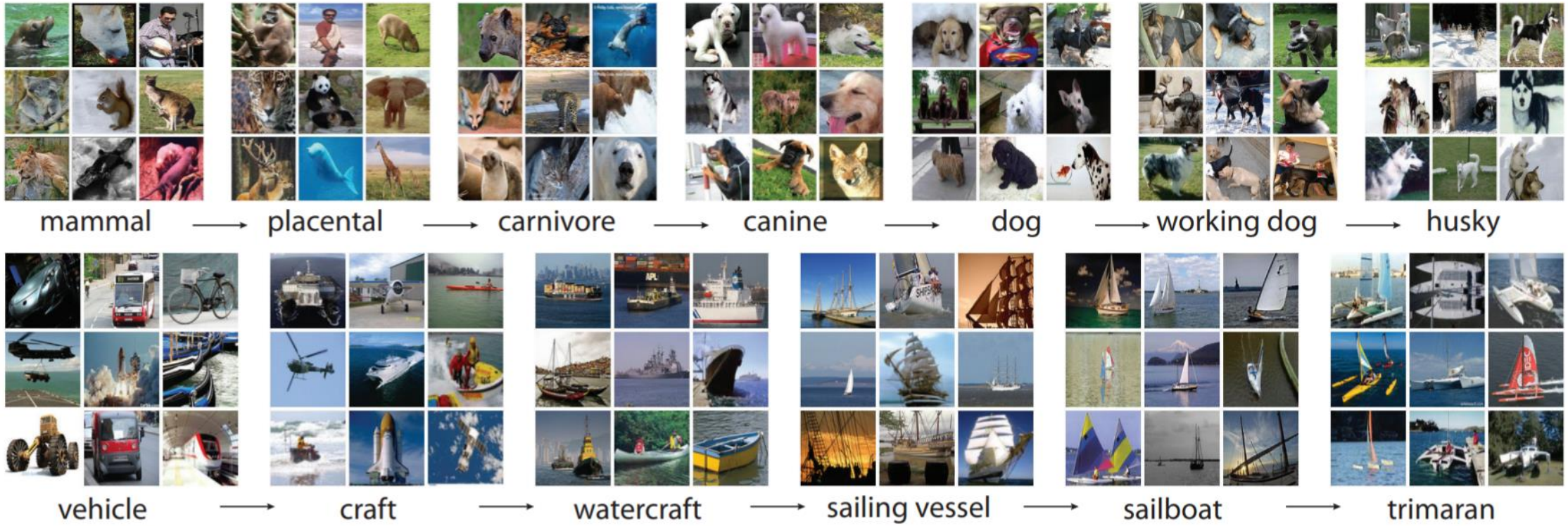
10k testing images (100 per class)

32x32 RGB images

Hierarchical structure: 20 super-classes with 5 sub-classes each

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

ImageNet / ImageNet 21k classes



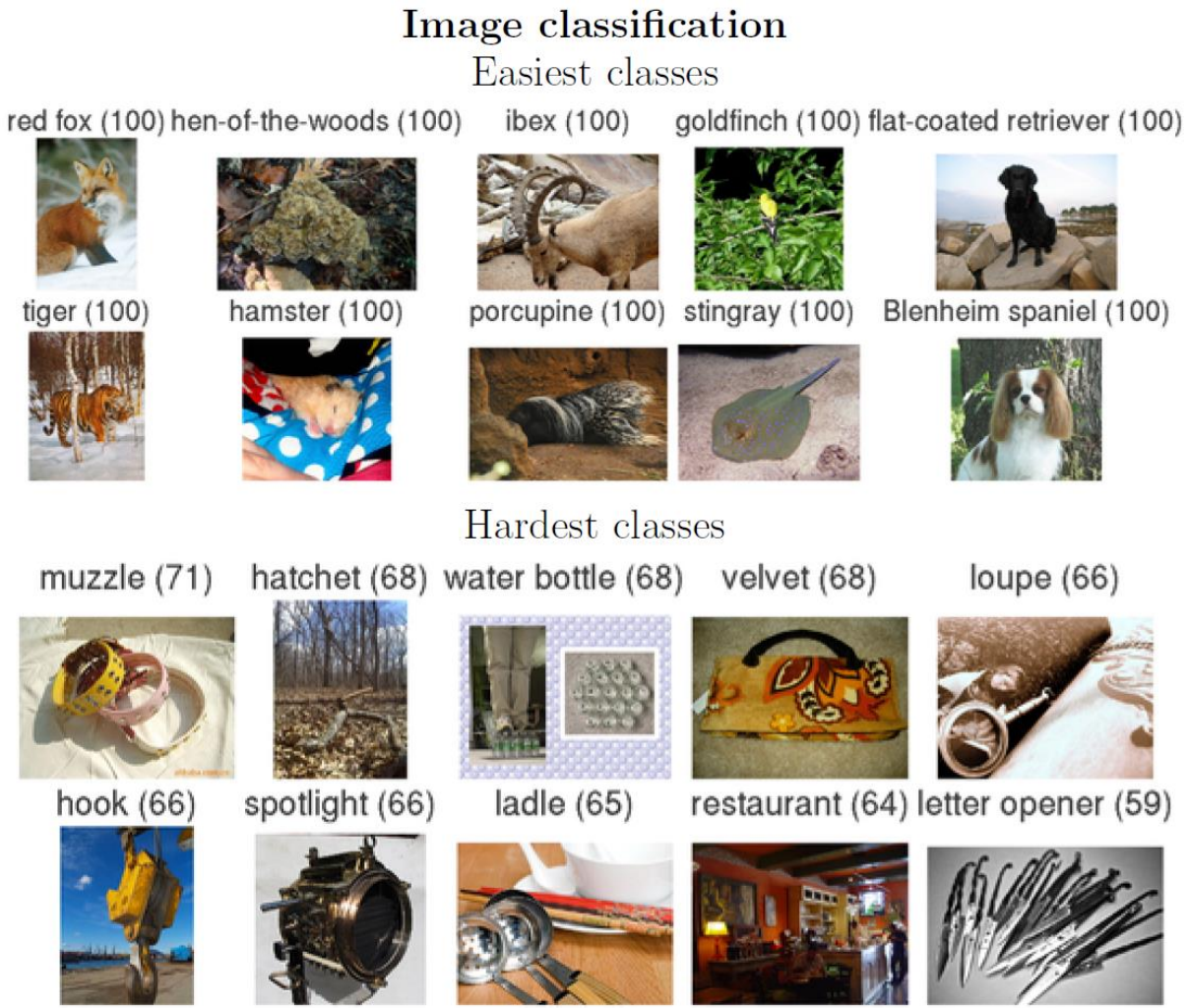
14 millions RGB images at full and variable resolution with average size about 400 × 350.

Hierarchical structure: modelled on about 21k synsets from **WordNet** (out of 50k)

groups of words that
mean the same concept

Deng et al, "ImageNet: A Large-Scale Hierarchical Image Database", CVPR 2009

ILSVRC – often referred to as ImageNet / ImageNet1k classes



1000 classes

1.3M training images (about 1300 per class)

50k validation images (50 per class)

100k test images (100 per class)

Variable resolution RGB images as in ImageNet, often resized to 256x256 for training.

because there are too many classes

Due to the inherent ambiguity of assigning only one label to each image, performance is usually reported as **top-5 accuracy**: an image is considered correctly classified if the correct label is present in 5 classes predicted by the algorithm. **Multi-label** accuracy has also been recently proposed

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

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

Vaishaal Shankar et al., Evaluating Machine Accuracy on ImageNet, ICML 2020

And many more…

A great resource to find datasets together with state-of-the-art methods is https://paperswithcode.com/sota

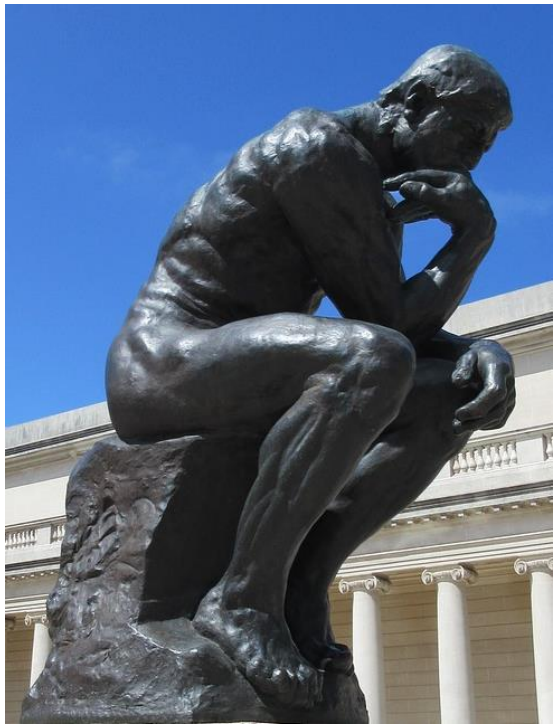
Benchmarks

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

A bird's eye view of traditional programming

Requirements (e.g. sort array)
(May include a few input/output examples, but only for clarity)



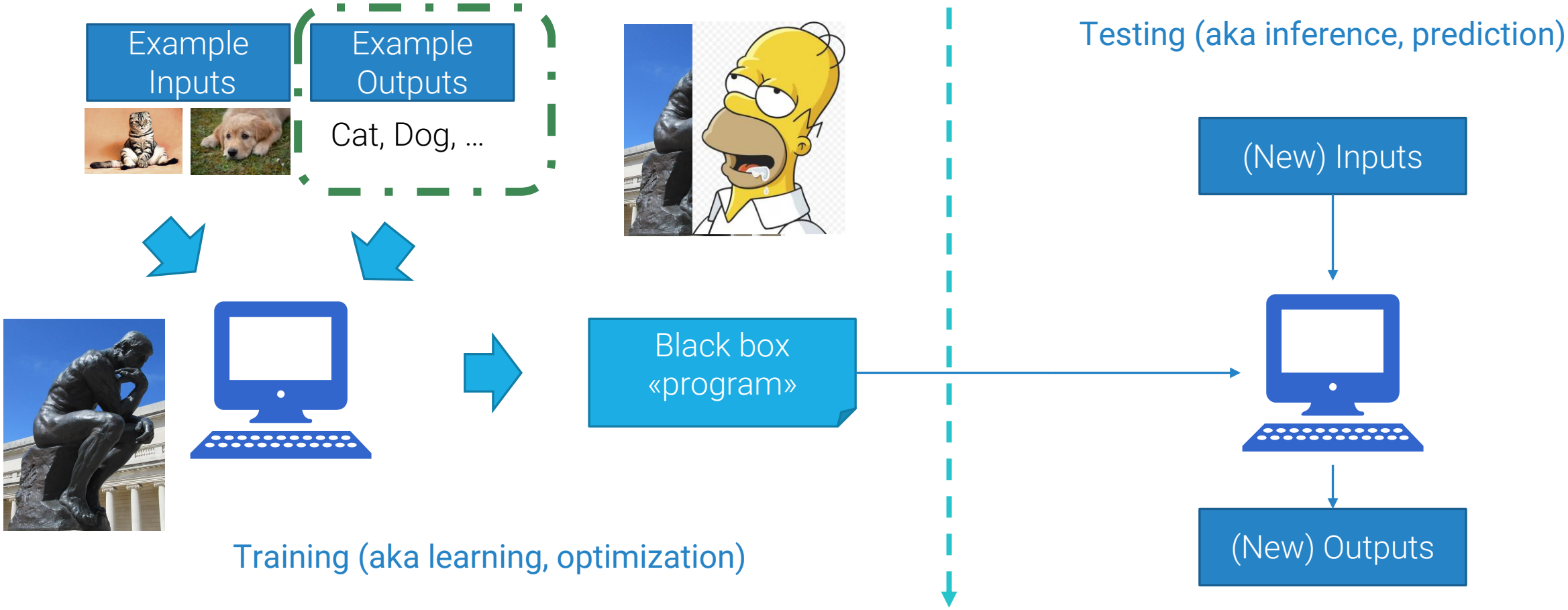
Program
for (int i=0; i <N; i++)
...

Input
[10,4, 6,1]

Output
[1,4,6,10]

Machine learning or data-driven approach

We can think of machine learning as a new way to instruct computers about what we want them to do.



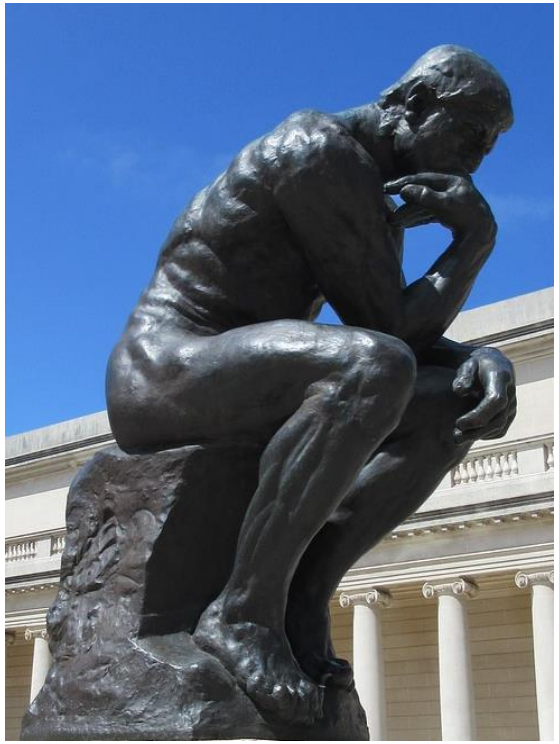
Data-driven approach: other consequences

The rise of data-driven approaches changes the relative importance of data and algorithms

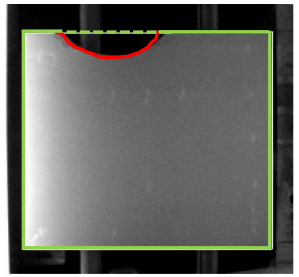
Traditional programming

for (int i = 0; ...

≈



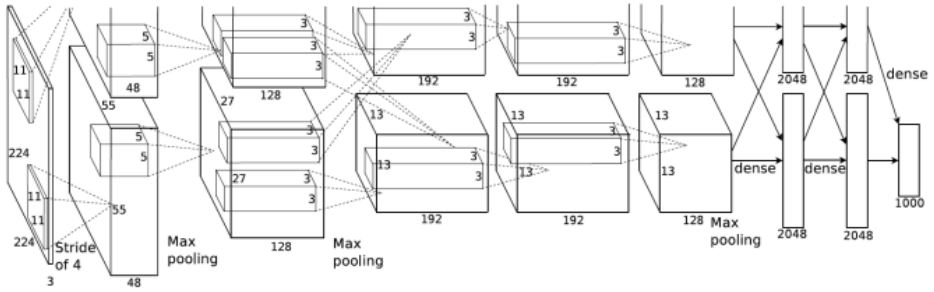
Input
[10,4, 6,1]
Output
[1,4,6,10]



Data-driven approach: other consequences

Data and datasets are crucial in this new paradigm. To know the most used ones for each vision task and to **understand the impact of metrics** on the results is therefore at least as important as it is to know the latest and greatest machine learning models.

Data-driven approach



MLOps: From Model-centric to Data-centric AI

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

https://www.youtube.com/watch?v=06-AZXmwHjo
https://worksheets.codalab.org/worksheets/0x7a8721f11e61436e93ac8f76da83f0e6

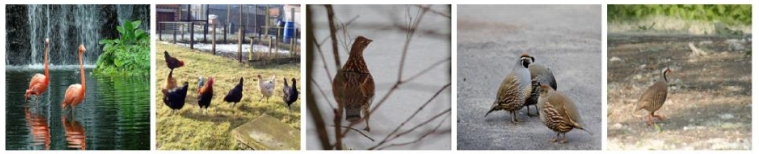
Parametric approach

$f($  $;\theta) = 2$

we will learn a set of parameters THETA that will force the output to be 2 when a picture like the one to the left, a bird, is the input of the model

array of number: theta

Cat  ...

Bird  ...

Linear classifier

cannot treat like a regression problem

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

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

32x32x3=3072x1 CIFAR image

flatten

1x3072

Scalar

Bad idea! Classes are categorical variables, their assignment to integer ids is random and it is not designed to capture the semantic of classes (i.e. if cat is class 3 and we get class=2,5 out of f, it does not mean that x depicts half a bird and half a cat; or, similarly, the fact that two ids are nearby or far away does not mean that the corresponding classes share or not visual similarities)

Linear classifier

do not output classes, instead scores, or LOGITS!!

in this way we say how this image looks like a frog, for example, independently from how it looks like a cat!

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

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

$argmax \rightarrow 2$

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

aka logits

32x32x3=3072x1 CIFAR image

classifier that has as many rows as classes and as many columns as pixels

10x3072

10x1

vector with dim = classes

Treating linear classification of images as template matching means that we can view the classification process as comparing the input image to a set of predefined templates (or patterns) corresponding to different classes

it's doing a correlation of the kernel it can learn and the input image, but JUST ONE template

Linear classifier as “template matching”

a linear classifier has to learn good templates for our classes

it's like INSTANCE DETECTION, want to recognize that exact template

but it's really limited " process"

These weights, when visualized, often resemble the average image of the respective class or emphasize the features that are most indicative of the class.

Class “template”

0.2	1.1	-3.2
-4.5	0.1	4.6

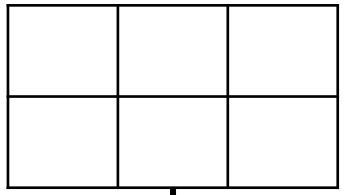
“Unflatten” row a kernel

computes dot product (which measure similarity) between kernel and portion of image

*

(=“correlation”)

Input image



=

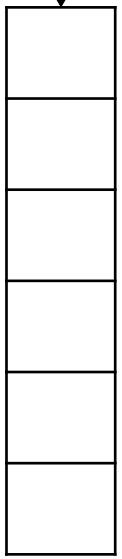
128.3

1 (car)

treating linear classification as template matching means interpreting the classification task as a comparison between the input image and a set of learned templates, where the class corresponding to the best-matching template is chosen

Flatten image

defining a order



this set of weights make our linear classifier

2.1	-4.6	3.3	2.2	2.0	0.9
0.2	1.1	-3.2	-4.5	0.1	4.6
9.0	8.5	4.5	2.1	4.8	9.5

each row creates a score for one class

3 classes

W

×

Matrix product

=

45.4	0 (plane)
128.3	1 (car)
253	2 (bird)

3x1

$scores$

In ML, linear often means affine...

it's not properly a linear function, we have to add a vector of biases also

$$f(\boldsymbol{x} =$$

$$;\theta) = 2$$

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

$argmax \rightarrow 2$

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

$$\theta = (W, \boldsymbol{b})$$

3072x1 10x3072 10x1 10x1

Learning as optimization

the space of all the possible functions my model can learn

The space of the functions that a machine learning model can produce is its **hypothesis space** $\mathbb{H}$

Learning = solve an **optimization problem** to find the "best" **function** $h \in \mathbb{H}$

$$\boldsymbol{h^* = \text{argmin}_{h \in \mathbb{H}} L(h, D^{train})}$$

the best function h* is the one which comes from the hyp space and minimizes the LOSS on the training data collected

our function h is completely specified by a set of parameters. namely each of those functions are parametrized by theta

Parametric models

Learning = solve an **optimization problem** to find the "best" **parameters** $\theta \in \Theta$

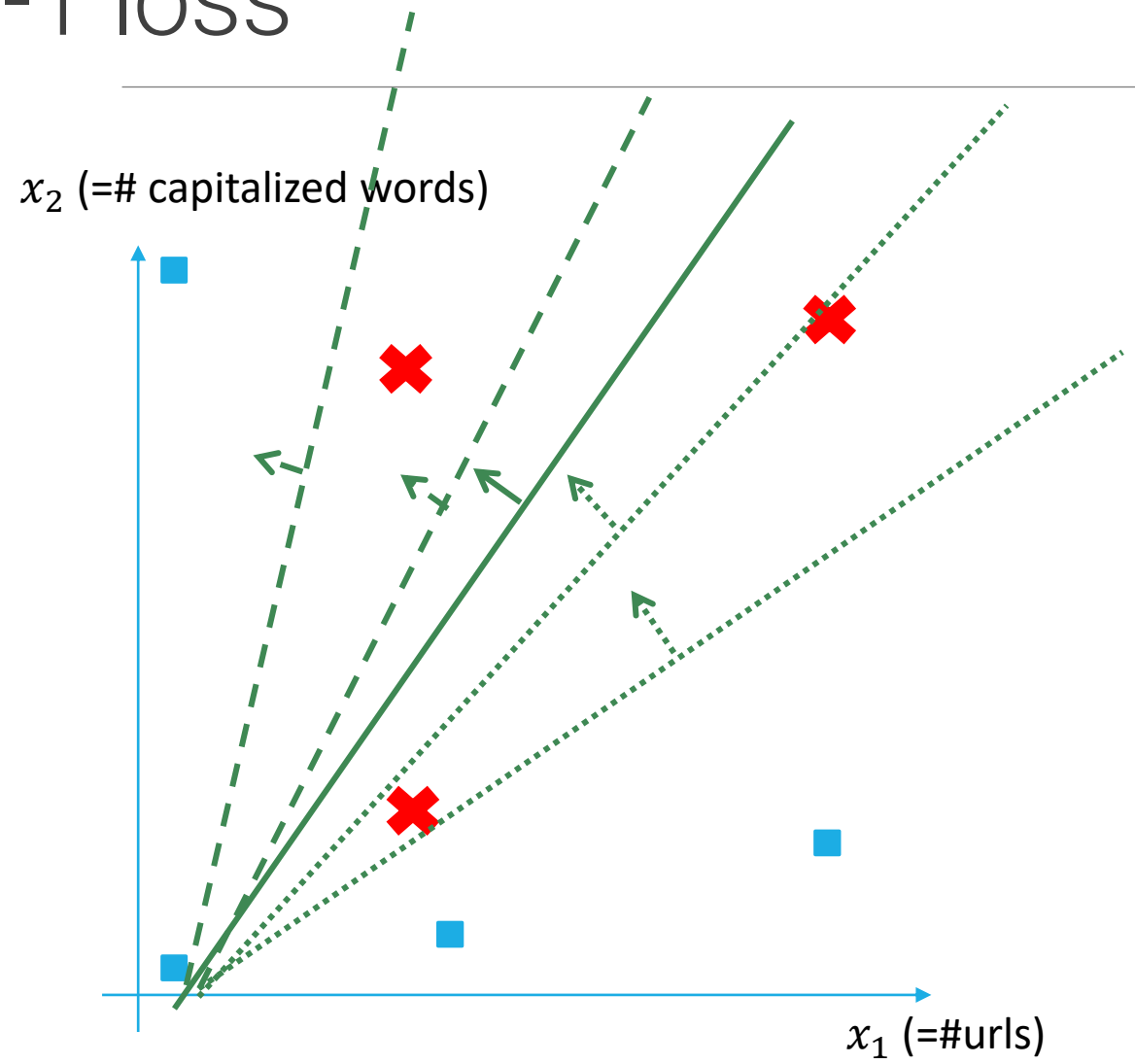
The goal of learning is to find the optimal parameters that minimizes the loss function

for each entry of the vector THETA we have one particular function

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

What function L should we minimize?

0-1 loss



$$L = \#errors?$$

This choice, known as the **0-1 loss**, results in a hard optimization problem.

Given the classifier represented by the solid line, changing it to become the dashed or the dotted line produces the same number of errors: **#errors is insensitive to small (and even large, sometimes) changes of the parameters.**

However, they are not equivalent directions of change: if we keep moving the classifier in the direction of the dotted line, we will improve it; whereas, if we keep moving in the direction of the dashed one, we will make more errors: **the error rate does not tell us if we are "moving" in the "right" direction while we change the parameters**.

We can minimize this loss as a combinatorial optimization problem, but then it does not scale well to large datasets

The loss function

as happens with 0-1 loss

Instead of directly optimizing accuracy, we then usually optimize a proxy measure, the **loss** function, that is easier to optimize but still correlated with how good our classifier is.

Loss function is also called **objective function, cost function, error function**,…

If the loss is high, our classifier is performing poorly, and we also expect low accuracy.

If the loss is low, our classifier is good, and we also expect high accuracy.

Hence, we prefer values of the parameters that minimize it on the training set

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

We will always work with losses whose value on a dataset is the average (or the sum) of the values for the single samples

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

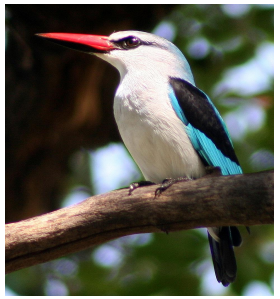
because each sample is indipendent

decompose loss over a sum of single samples

What loss can we use for the linear classifier?

Historically, the RMSE between the prediction and the training label was used

$$\boldsymbol{L}\left(\boldsymbol{\theta}, (\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\right) = \boldsymbol{L}\left(\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b}, \boldsymbol{y}^{(i)}\right) = \left\|\boldsymbol{W}\boldsymbol{x}^{(i)} + \boldsymbol{b} - \mathbb{I}(\boldsymbol{y}^{(i)})\right\|_2$$



$$f\left(\boldsymbol{x}^{(i)}; \theta\right) = W\boldsymbol{x}^{(i)} + \boldsymbol{b} =$$

0 (plane)	45.4
1 (car)	128
2 (bird)	253
...	0.23
	-1.34
	...

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

0 (plane)	0
1 (car)	0
2 (bird)	1
...	0
	0
	...

This is called one-hot encoding of class bird

Softmax function

transform output into probabiliy mass function over the possible classes

It turns out there are theoretical and practical reasons to prefer a loss that **transforms the scores computed by the classifier into probabilities** and then perform **maximum likelihood estimation** of θ.

How do we go from scores s_j to probabilities? We use the **softmax** function ($\mathrm{softmax}\colon \mathbb{R}^n \to \mathbb{R}^n$)

$$p_{model}\left(Y = j \middle| X = x^{(i)}; \theta\right) = \mathrm{softmax}_j(s) = \frac{\exp(s_j)}{\sum_{k=1}^{C} \exp\left(s_k\right)}$$

0 (plane)	2,1		8,2		0,00
1 (car)	5,3		200,3		0,09
2 (bird)	7,6	$\exp(s_k)$	1998,2	$norm.$	0,90
…	0,2		1,2		0,00
	-1,3		0,3		0,00
	2,5		12,2		0,01

Should be called "softargmax", as **it is a smooth and differentiable approximation of the one-hot encoding of the $\boldsymbol{argmax}$**

To implement it reducing numerical issues , it is useful to note that

$$\mathrm{softmax}_j(s + c) = \frac{\exp(s_j + c)}{\sum_k \exp(s_k + c)} =$$

$$\frac{\exp(s_j)\exp(c)}{\sum_k \exp(s_k)\exp(c)} = \frac{\cancel{\exp(c)}\exp(s_j)}{\cancel{\exp(c)} \sum_k \exp(s_k)} = \mathrm{softmax}_j(s)$$

subtract then the max of the scores

and then compute it as $\mathrm{softmax}\left(s - \max_k s_k\right)$

Cross-entropy loss

the softmax of the scores will be the probability that my model assigns to the classes given any image and a set of learnable parameters

Now that our linear classifier outputs «probabilities» over the classes, we can think of it as **a family of probability mass functions** over the classes given an image, indexed by the vector of parameters θ

MLE: we want the model to assign a higher probability to the true class for the given image

$$\text{softmax}\left(f\left(\boldsymbol{x}^{(i)};\theta\right)\right) \doteq p_{model}(Y \mid \boldsymbol{X} = \boldsymbol{x}^{(i)};\theta)$$

Then, **the maximum likelihood estimation** of θ is

about maximizing the probabilities of the true data

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

decomposition due to independ. assumption

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

curse of products: very small numbers

$$= \arg\max_{\theta} \sum_{i=1}^{N} \log p_{model}\left(Y = y^{(i)} \middle| \boldsymbol{X} = \boldsymbol{x}^{(i)};\boldsymbol{\theta}\right)$$

we want to minimize the negative log-likelihood

$$= \arg\min_{\theta} \sum_{i=1}^{N} -\log p_{model}\left(Y = y^{(i)} \middle| \boldsymbol{X} = \boldsymbol{x}^{(i)};\boldsymbol{\theta}\right)$$

i.i.d. data

Avoid the «curse of products»

Per-sample cross-entropy loss $\boldsymbol{L}\left(\boldsymbol{\theta},(\boldsymbol{x}^{(i)},\boldsymbol{y}^{(i)})\right)$

0 (plane)	0,00
1 (car)	0,09
2 (bird)	0,90
…	0,00
	0,00
	0,01

If the true class of this image is “bird”, then the cross-entropy loss is $-\log(0.9) = 0.1$ (low value)
If it were “car”, then the loss value would have been $-\log(0.09) = 2.4$ (high value)

check: am i doing better than random classifier

Putting everything together

Given

$$p_{model}\left(Y = j \middle| X = x^{(i)}; \theta\right) = \mathrm{softmax}_j(s) = \frac{\exp(s_j)}{\sum_{k=1}^{C} \exp\left(s_k\right)}$$

and the per-sample loss to minimize to perform maximum likelihood estimation

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

the overall expression for the per-sample loss becomes

the crossentropy is about minimizing the score of the correct class plus the logsumexp of all the other classes

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

The second term of the sum on the right is usually referred to as the `logsumexp` and numerical libraries (e.g. PyTorch) usually have numerically stable functions to compute it. It approximates the max function, hence we can think of the cross-entropy loss as approximately

the logsumexp can be approximated with the max score

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

if the score of the correct class is the largest, then this is roughly zero, meaning that we are correctyl classifying oru images

and of minimizing it as penalizing the most active incorrect prediction, i.e. a proxy to increase accuracy.

if the largest score is not the one corresponding to the correct class, in order to minimize the quantity above, we can either increase the score of the correct class or decreasing the score of the wrongly seelcted class

Losses in PyTorch

they are theoretically the same thing, they only differ for some implementation details

In PyTorch, you will find both `NLLLoss` (where NLL stands for negative log likelihood) and `CrossEntropyLoss`. Pay attention to the difference!

CLASS `torch.nn.CrossEntropyLoss(weight: Optional[torch.Tensor] = None, size_average=None, ignore_index: int = -100, reduce=None, reduction: str = 'mean')` [SOURCE]

This criterion combines `nn.LogSoftmax()` and `nn.NLLLoss()` in one single class.

It is useful when training a classification problem with *C* classes. If provided, the optional argument `weight` should be a 1D *Tensor* assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* is expected to contain raw, unnormalized scores for each class.

input has to be a Tensor of size either $(\text{minibatch}, C)$ or $(\text{minibatch}, C, d_1, d$

if my model outputs scores, use Cross Entropy Loss

use NLL Loss if outputs are already softmax outputs

CLASS `torch.nn.NLLLoss(weight: Optional[torch.Tensor] = None, size_average=None, ignore_index: int = -100, reduce=None, reduction: str = 'mean')` [SOURCE]

The negative log likelihood loss. It is useful to train a classification problem with *C* classes.

If provided, the optional argument `weight` should be a 1D Tensor assigning weight to each of the classes. This is particularly useful when you have an unbalanced training set.

The *input* given through a forward call is expected to contain log-probabilities of each class. *input* has to be a Tensor of size either $(\text{minibatch}, C)$ or $(\text{minibatch}, C, d_1, d_2, ..., d_K)$ with $K \geq 1$ for the *K*-dimensional case (described later).

Obtaining log-probabilities in a neural network is easily achieved by adding a *LogSoftmax* layer in the last layer of your network. You may use *CrossEntropyLoss* instead, if you prefer not to add an extra layer.

The *target* that this loss expects should be a class index in the range $[0, C-1]$ where *C = number of classes*; if

Where are we?

we have to define a measure of quaility fro our classifier, that cannot be accuracy (number of errors: 0/1 loss) but it is instead another scalar number, the loss

cross entropy loss: most used for classifcation

$$s^{(i)} \rightarrow \boxed{-\log\left(\frac{\exp\left(s_{y^{(i)}}\right)}{\sum_{k=1}^{C}\exp\left(s_k\right)}\right)} \leftarrow \mathbb{I}\left(y^{(i)}\right)$$

$$f\left(\boldsymbol{x}^{(i)};\theta\right) = W\boldsymbol{x} + \boldsymbol{b} =$$

$$\theta = (W, \boldsymbol{b})$$

3072x1

10x3072

10x1

0 (plane)	2,1
1 (car)	5,3
2 (bird)	7,6
...	0,2
	-1,3
	2,5
	3,5
	4,5
	5,5
	6,5

0 (plane)	0
1 (car)	0
2 (bird)	1
...	0
	0
	0
	0
	0
	0
	0

Now we can express preferences over different choices of θ with one number. How can we use this knowledge to train our model, i.e. select the "best" θ?

G D

Gradient descent

we optimize/learn parameters with GD; instead, we have to do something else for hyperparams

number of params to estimate is very large

we use the whole training set in order to do update the parameter under interest

$$\boldsymbol{\theta}^* = \boldsymbol{argmin}_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \sum_{\boldsymbol{i}} \boldsymbol{L}\left(\boldsymbol{\theta}, \left(\boldsymbol{x}^{(\boldsymbol{i})}, \boldsymbol{y}^{(\boldsymbol{i})}\right)\right)$$

standard GD: full pass through the training set

differentyl from camera calibration, where there is an initial guess

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

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

here in GD is then done an averaging of the sum of the computed gradients

1. Forward pass: classify all the training data to get the predictions $\hat{y}^{(i)} = f\left(\boldsymbol{x}^{(\boldsymbol{i})}; \theta^{(e-1)}\right)$ and the loss $L\left(\theta^{(e-1)}, D^{train}\right)$
2. Backward pass: Compute the gradient $g = \frac{\partial L}{\partial \theta}\left(\theta^{(e-1)}, D^{train}\right)$
3. Step: Update the parameters: $\theta^{(e)} = \theta^{(e-1)} - \boldsymbol{lr} * g$

keep attention to variance of data

it says in which direction i should update the parameter

Important design decisions / hyperparameters

Initialization method?

Number of epochs? Or alternative way to decide to stop?

Learning rate $\boldsymbol{lr}$?

Limits of gradient descent

when we have very huge dataset

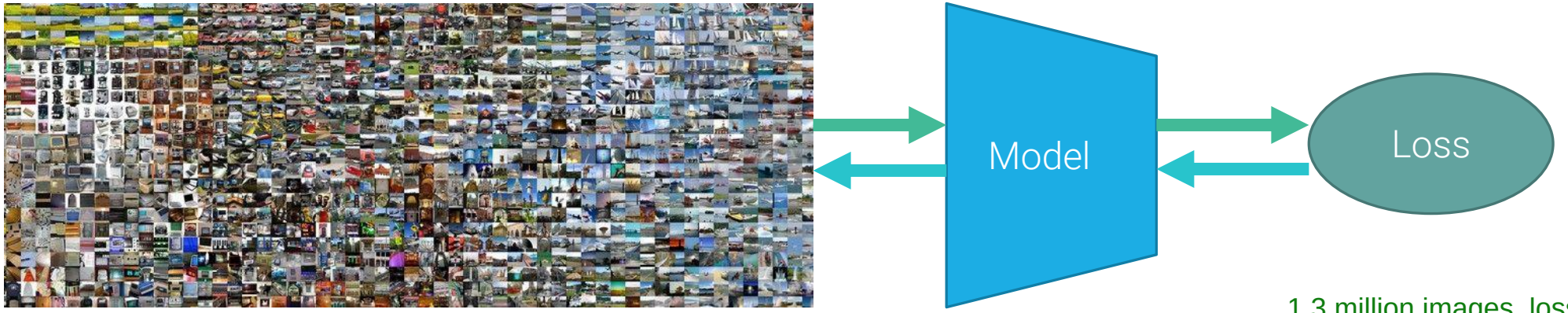
our loss is decomposed into a sum over losses for each sample

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

also the total gradient is decomposed into a sum of the gradients of the single entries

The total gradient $\nabla_\theta L$ is computed as the sum of the gradients of the single training samples.

Think about training a linear model on ImageNet (1.3M training images…).



1.3 million images, losses, gradients,..
too much expensive

1.3M forward passes and 1.3M backward passes just to perform one (usually tiny) step

we never compute the true gradient

Batch gradient descent is an approximation

ImageNet is large…
but all the possible images of its 1000 classes are a much larger set

The gradient computed by gradient descent approximates the "true" gradient, even when computed on all available training data, as they are a (large, but not full) sample of the true population (i.e., all the possible images of its classes)

SGD - Stochastic Gradient Descent

$$\boldsymbol{\theta}^* = \boldsymbol{argmin}_{\boldsymbol{\theta}\in\boldsymbol{\Theta}} \sum_{\boldsymbol{i}} \boldsymbol{L}\left(\boldsymbol{\theta}, (\boldsymbol{x}^{(\boldsymbol{i})}, \boldsymbol{y}^{(\boldsymbol{i})})\right)$$

A faster alternative to optimize a differentiable function is stochastic gradient descent.

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

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

1. Randomly shuffle examples in $\boldsymbol{D}^{\boldsymbol{train}}$

for $\boldsymbol{i} = \boldsymbol{0}, \dots, \boldsymbol{N}-\boldsymbol{1}$

2. Forward pass: classify $\boldsymbol{x}^{(\boldsymbol{i})}$ to get the predictions $\hat{y}^{(i)} = f\left(\boldsymbol{x}^{(\boldsymbol{i})}; \theta^{(e*N+i)}\right)$ and the loss $L\left(\theta^{(e*N+i)}, (\boldsymbol{x}^{(\boldsymbol{i})}, \boldsymbol{y}^{(\boldsymbol{i})})\right)$
3. Backward pass: Compute the gradient $g = \frac{\partial L}{\partial \theta}\left(\theta^{(e*N+i)}, (\boldsymbol{x}^{(\boldsymbol{i})}, \boldsymbol{y}^{(\boldsymbol{i})})\right)$
4. Step: Update the parameters: $\theta^{(e*N+i+1)} = \theta^{(e*N+i)} - \boldsymbol{lr} * g$

One sample instead of the full dataset
Sometimes this is also called "on-line" GD

Parameters updated after every forward+backward pass

SGD - Stochastic Gradient Descent with minibatches

A compromise: use a mini-batch of data of size $\boldsymbol{B}$ at each iteration instead of a single example. The number of parameter updates in each epoch will be $U = \left\lceil \frac{\boldsymbol{N}}{\boldsymbol{B}} \right\rceil$

how many times I update parameters

B samples are processed simultaneously
For $B = 1$ we recover online SGD,
for $B = N$ batch gradient descent

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

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

1. Randomly shuffle examples in $\boldsymbol{D^{train}}$

for $\boldsymbol{u} = \boldsymbol{0}, \dots, \boldsymbol{U} - \boldsymbol{1}$

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

it computes one gradient but that is the average of the samples in B

3. Backward pass: Compute the gradient $g = \frac{\partial L}{\partial \theta}\left(\theta^{(e*U+u)}, (\boldsymbol{X^{(u)}}, \boldsymbol{Y^{(u)}})\right)$
4. Step: Update the parameters: $\theta^{(e*U+u+1)} = \theta^{(e*U+u)} - \boldsymbol{lr} * g$

Design decisions / hyperparameters

Initialization method?
Number of epochs? Or how to decide to stop?
Learning rate, i.e. step size?
Mini-batch size?

Mini batches - tradeoffs

empirical observations, no theorems

I have to keep in memory all the intermediate results

it means there is saturation

- Larger batches provide smoother estimations of the gradient, but with less than linear returns.
- Larger batches better exploit parallel hardware, below a minimum size no gain in training time.
- If examples in a batch are processed in parallel, memory requirements scale linearly with batch size.
- Usually power of 2 sizes (more suited for parallel hardware), typical values on-single GPUs is 16 to 256, but modern distributed training can use values up to 8,192 or 16,384 on multiple GPUs. [Goyal et al. 2018]
- Smaller batches may have a regularization effect and results in better generalization, at the cost of longer training time. [Smith et al., 2020]

larger batches inject more noise in the model

I make the task more diffcult, wanting a better model, that generalizes better -> more training time

- Usual rule of thumb: start with the largest power of 2 size that fits in the memory of your GPU. Experiment from there, if time allows.

Goyal et al., Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour, 2018. https://arxiv.org/abs/1706.02677
Smith et al., On the Generalization Benefit of Noise in Stochastic Gradient Descent, 2020. https://arxiv.org/abs/2006.15081

How to compute gradients?

Gradients can be computed

- Numerically
  - Slow and approximate
  - But easy to implement

with finite differences

- Analytically, by exploiting the rules of calculus and, in particular, the **chain rule**.

$$\frac{d}{dx} f\big(g(x)\big) = f'\big(g(x)\big) g'(x)$$

  - Exact, but slow, tedious, and error prone. For the linear classifier, we get

$$\frac{\partial L}{\partial b_j} = \frac{\partial}{\partial b_j} - \log\left(\frac{\exp(s_j)}{\sum_{k=1}^{C} \exp\,(s_k)}\right) = \frac{\partial}{\partial b_j} \; - \log\left(\frac{\exp(w_j x + b_j)}{\sum_{k=1}^{C} \exp\,(s_k)}\right) = -\frac{\sum_{k=1}^{C} \exp\,(s_k)}{\exp(w_j x + b_j)} \left(\frac{\partial}{\partial b_j} \frac{\exp(w_j x + b_j)}{\sum_{k=1}^{C} \exp\,(s_k)}\right) = \cdots$$

- Automatically, by **automatic differentiation,** e.g. with the **backpropagation algorithm**

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986) "Learning representations by back-propagating errors." Nature, 323, 533--536.

What does a linear model learn on CIFAR10?

Accuracy about 38%

Let's use the template matching interpretation of a linear classifier to understand what the model is learning

It looks like the background color is the predominant feature used by the model

Moreover, one template cannot capture multiple appearances within one class, e.g. rotated cars, trucks, etc..

Distance between templates and images is a distance in input space.

major limiitation: you can0t do better if you remain in pixel space

here it's way more important the context rather than tje object itself

What's next?

Learn more effective representations.