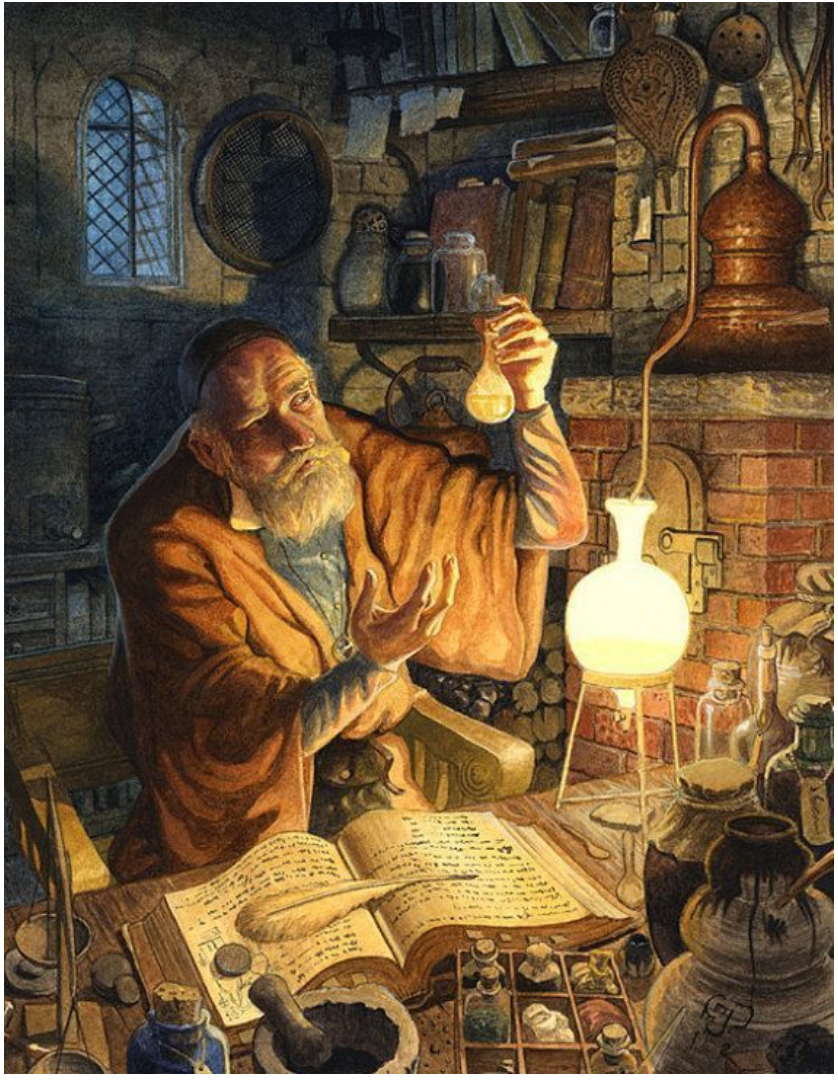


Neural Networks and image classification

M1 ARIA
Image and Video Processing

Gabriele Facciolo

Lecture 1 - 16-09-2024

Presentation of the course

The objective of this course is to present a panorama of the main modeling aspects and practical insights of neuronal networks (NN) for computer vision applications.

Page: https://gfacciol.github.io/M1_IAML_image/

Presentation of the course

The objective of this course is to present a panorama of the main modeling aspects and practical insights of neuronal networks (NN) for computer vision applications.

Page: https://gfacciol.github.io/M1_IAML_image/

Lessons:

1. – Monday 18/9 (1X74): 8h00-12h30 - Intro NN, backprop and CNN for classification
2. – Friday 22/9 (1Q07): 13h30-16h00 - Semantic segmentation
3. – Friday 29/9 (1Q07): 13h30-16h00 - Object detection
4. – Friday 6/10 (1Q07): 13h30-16h00 - Transfer learning and representation learning

Validation

Homeworks

+

Binome Project / Bibliographical study of a subject of your interest

1. Think a subject
2. Check with me if it is feasible (must include some coding)
3. Do it!
4. Write a short report + presentation

Plan

- The image classification problem
- Feedforward neural networks
  - Perceptron
  - Deep multilayer networks
  - Types of layers
  - The power of deep architectures
- Training
  - Data
  - Loss and optimization
  - Backpropagation
  - Weight initialization and evolution
- Easing the learning
  - Batch Normalization
  - Regularization
  - Skip connections / Residual learning

Image classification

Image classification

- Image classification is the prototypical computer vision problem
- A nontrivial problem:

$$\mathbf{u}_i \in \mathbb{R}^{H \times W \times 3} \longrightarrow c_i \in \mathcal{G}$$

- Difficult to craft a program to solve it in an unrestricted setting

Image classification is hard

Changes in illumination

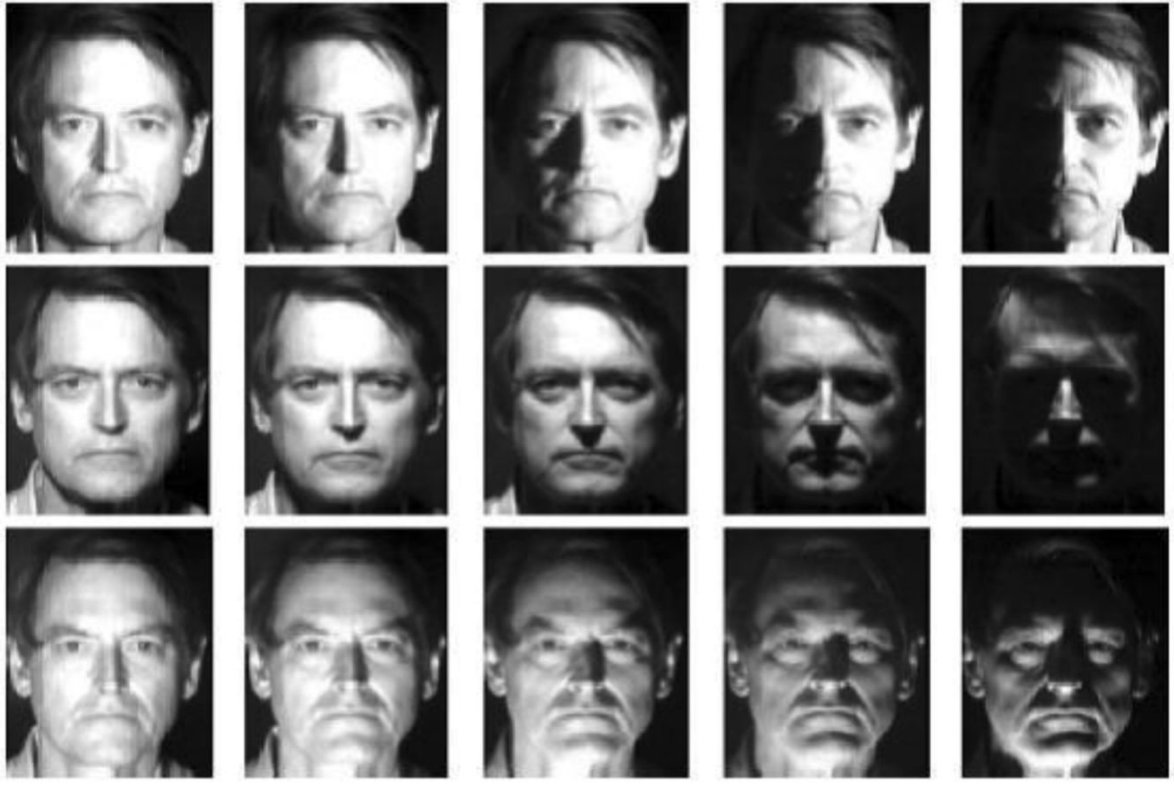


Image classification is hard

Pose and shape changes

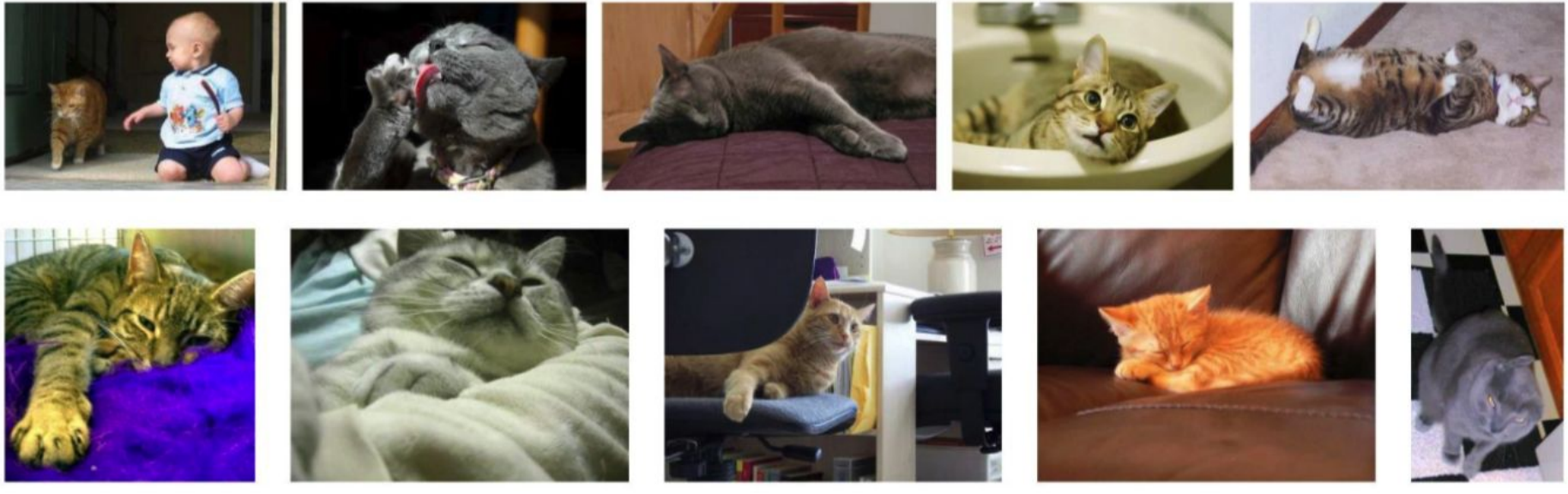
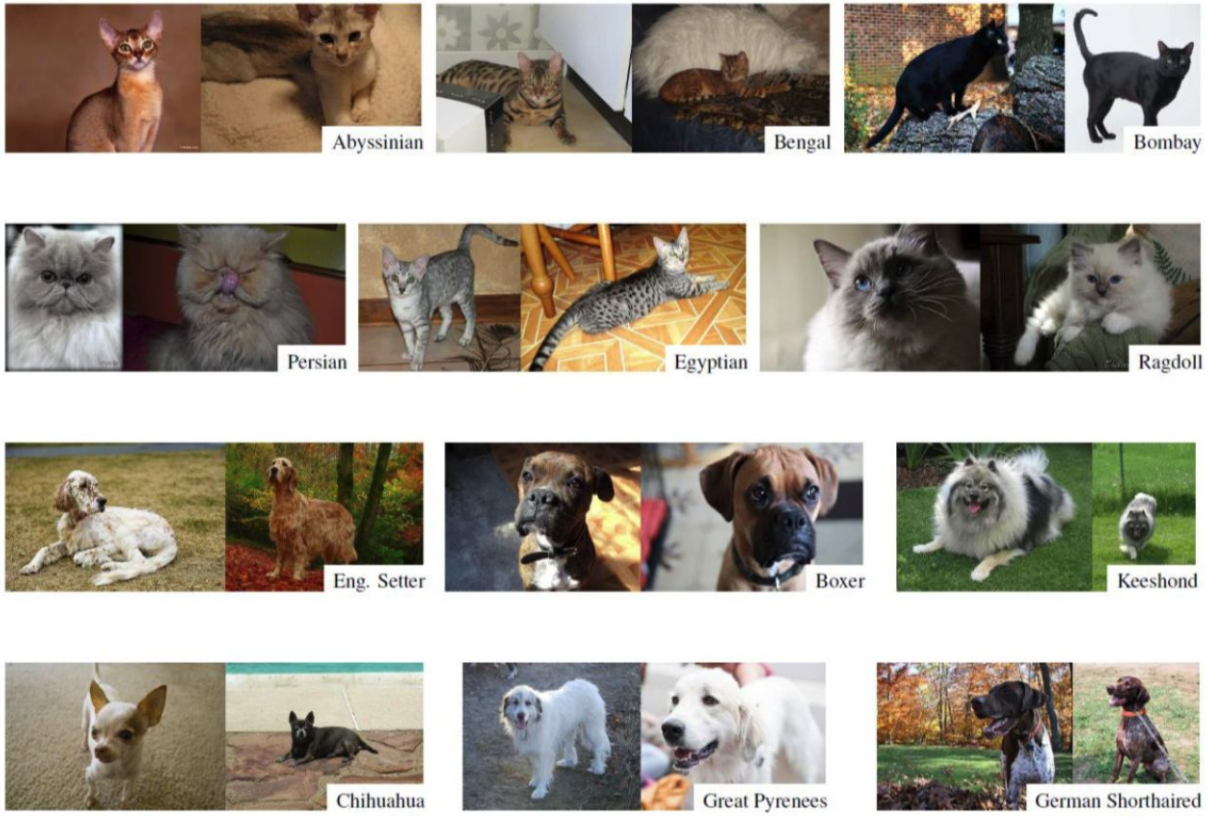


Image classification is hard

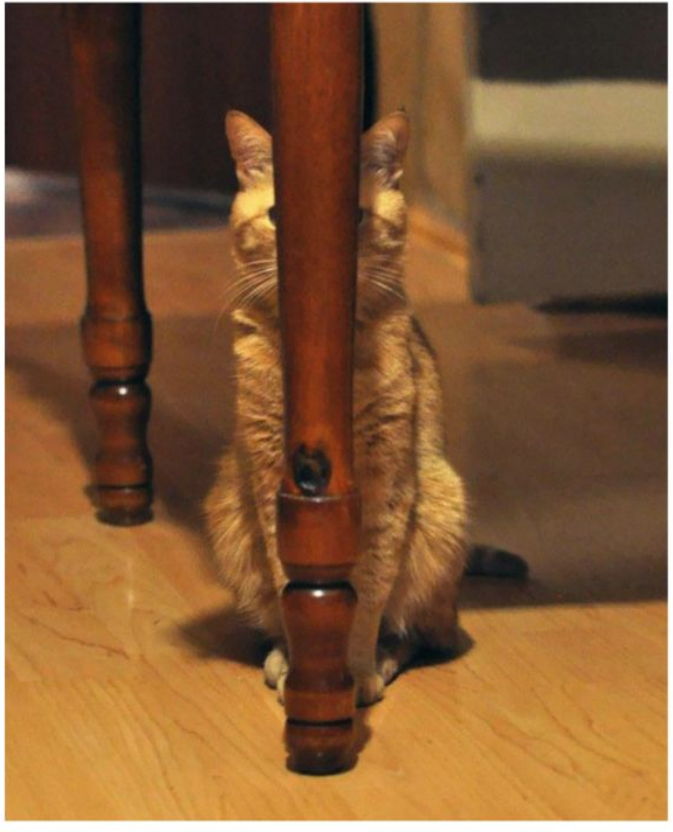
Intra-class variability



Parkhi et al. "Cats and dogs." 2012.

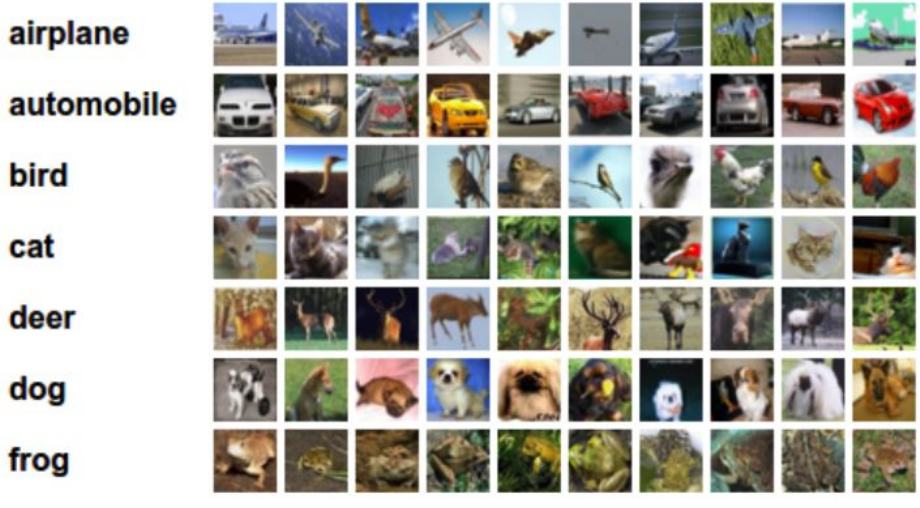
Image classification is hard

Occlusions



Data driven approaches

1. Assemble a **dataset** of labeled images
2. **Train a classifier** using the labeled examples
3. **Evaluate** the classifier on new images

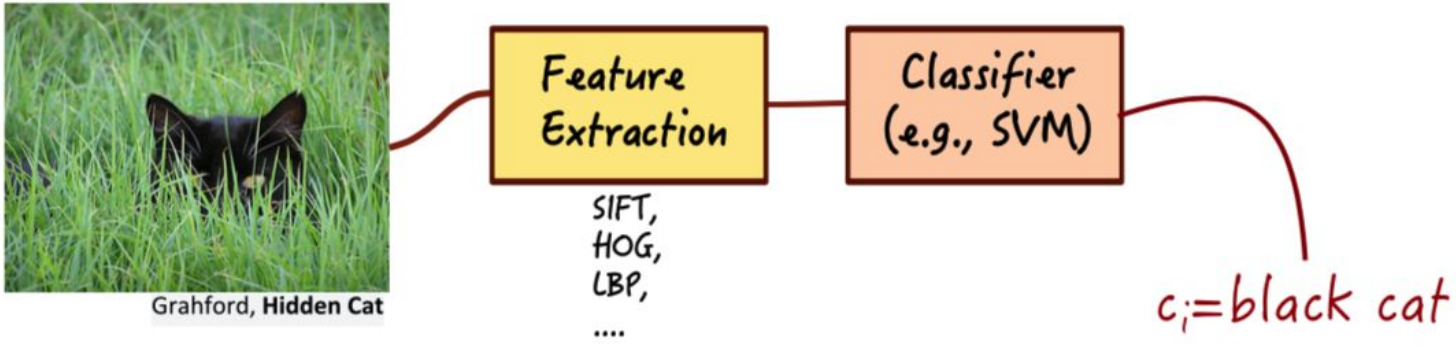


A. Krizhevsky, V. Nair, and G. Hinton "CIFAR-10"

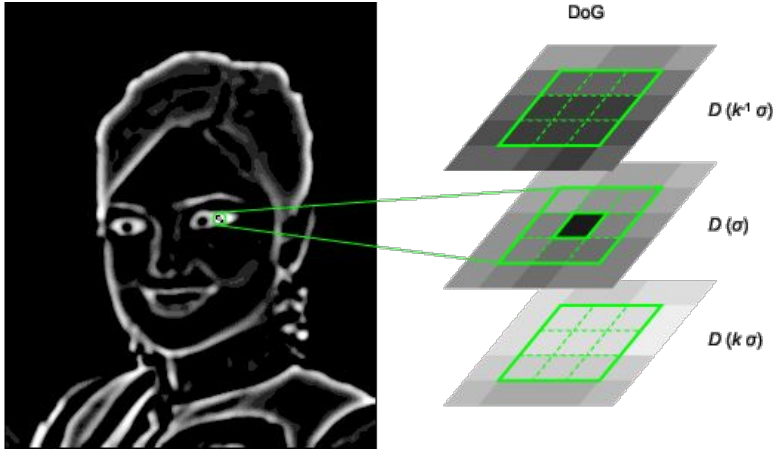
O. Russakovsky, et al. "ImageNet Large Scale Visual Recognition Challenge"

Classic approaches

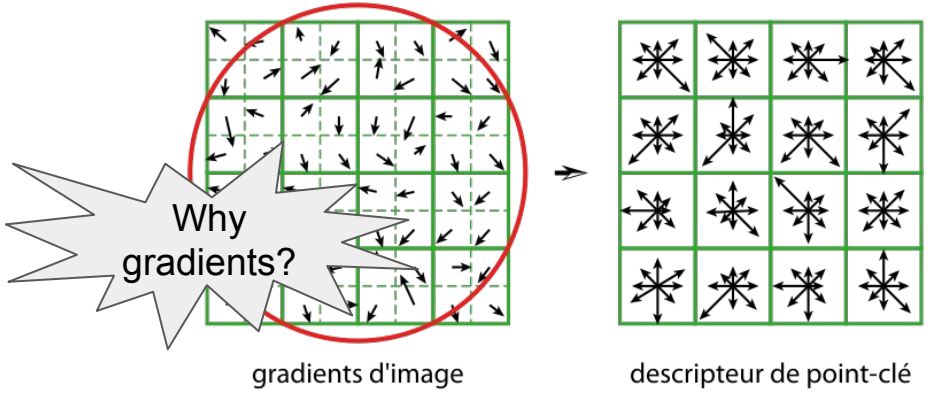
- First **extract features** (SIFT, HOG...), then feed them to a classifier
- Allows to **reduce the dimension** of the classifier
- **Features are invariant** (to rotation, translation, scale, and illumination changes) and allow to robustly classify



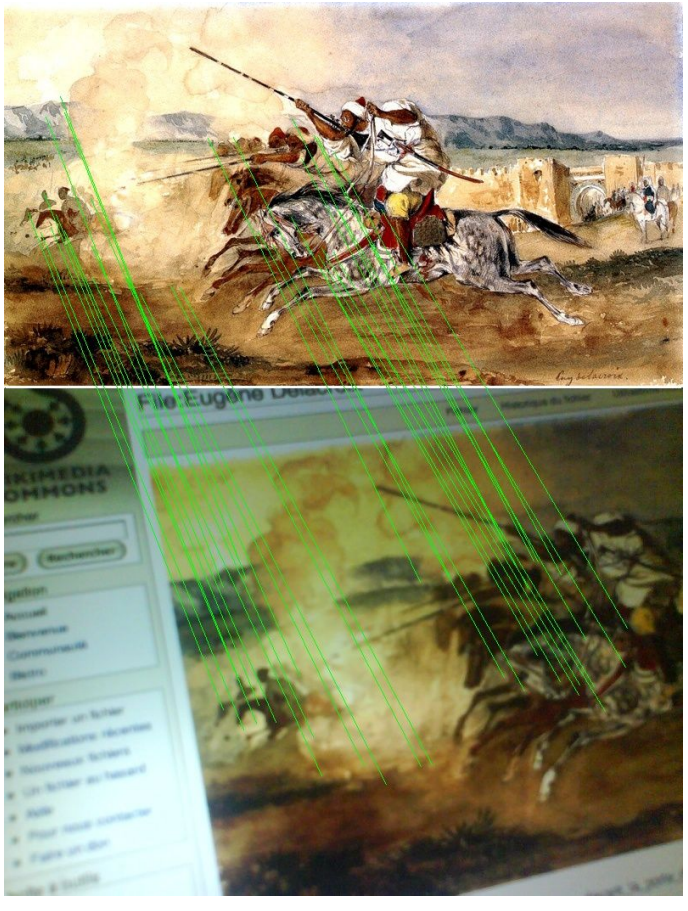
SIFT in a nutshell



1. Keypoint localization



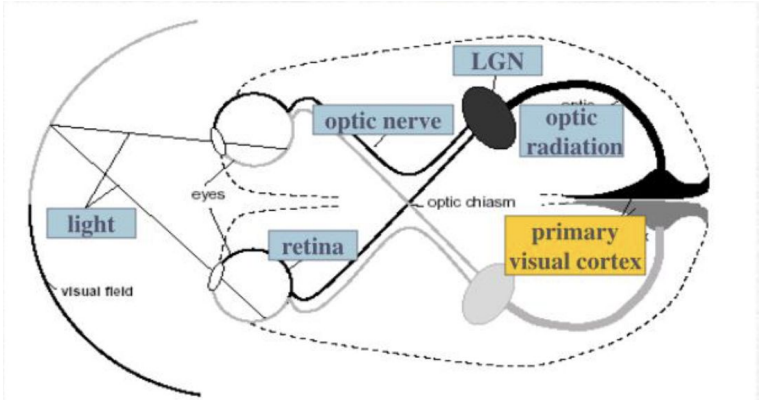
2. Keypoint descriptors



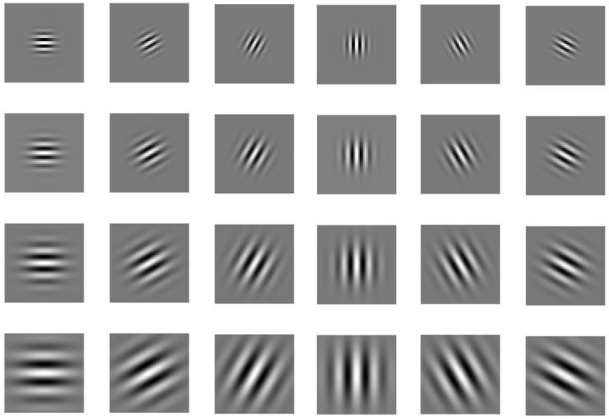
3. Keypoint matches

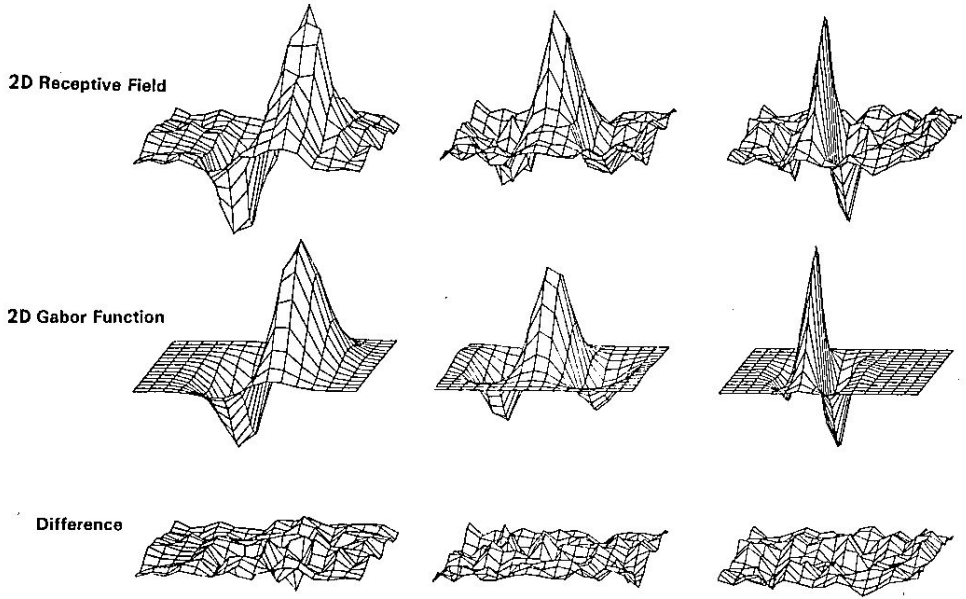
Edge-based features are physiologically plausible

- Hubel & Wiesel ‘62: visual cortex neurons have a restricted receptive field. These cells are sensitive to specific orientations in the receptive field (like linear filters)

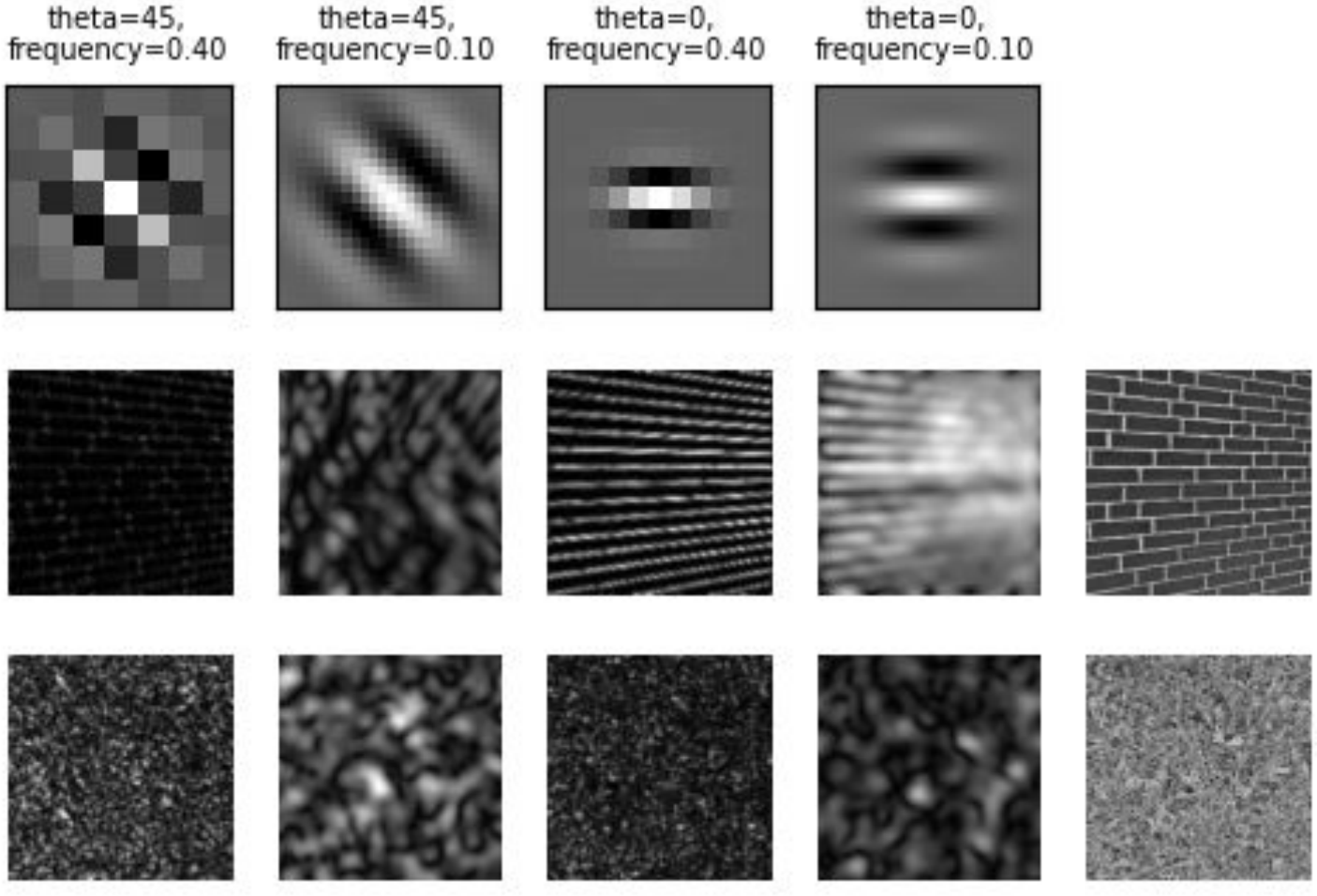


- Jones & Palmer ‘87 concluded that Gabor filters fit well these activations

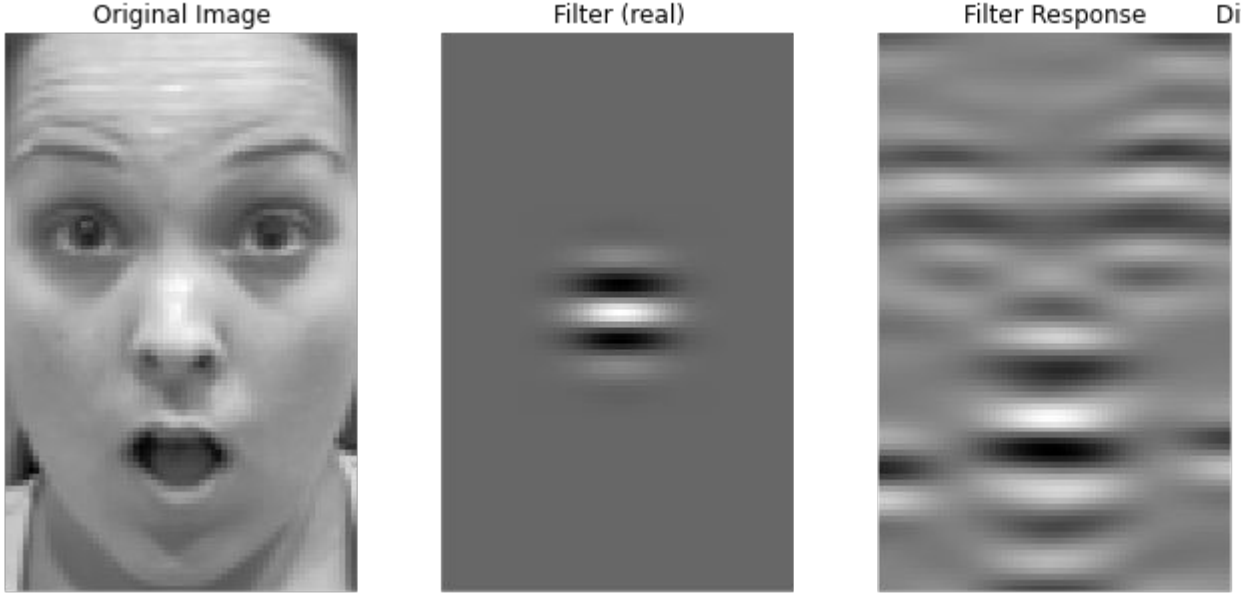




Gabor filters decompose images in useful features

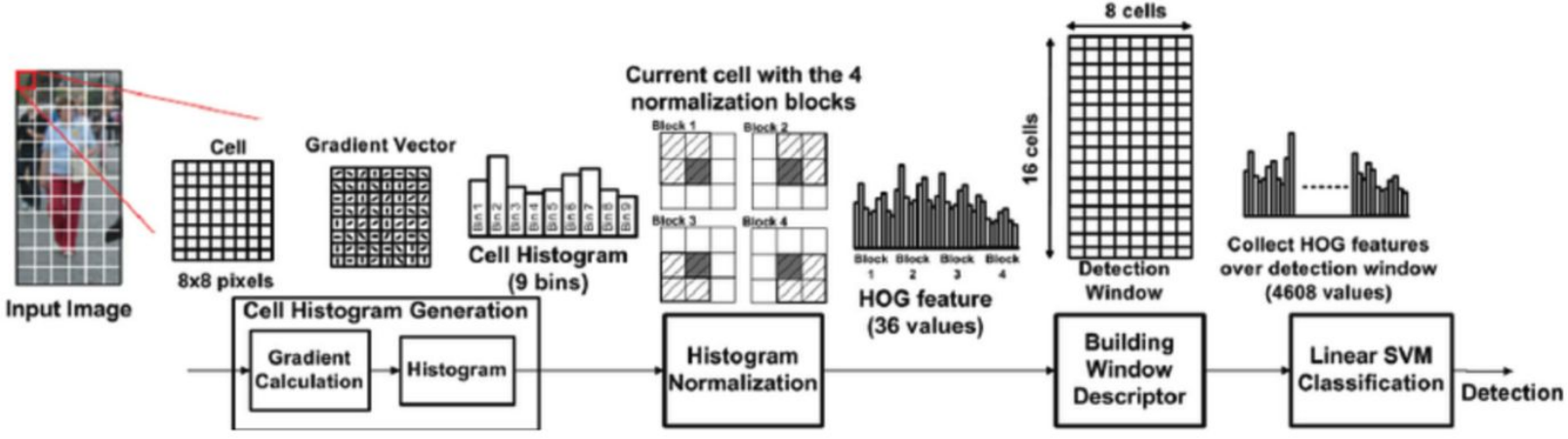


Gabor filters decompose images in useful features



Classic computer vision approaches

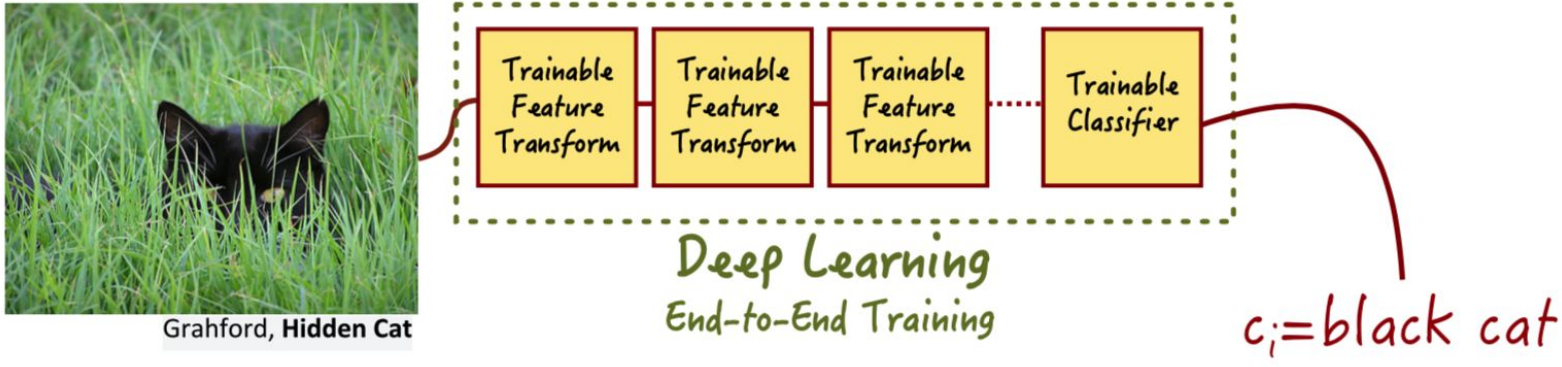
- First **extract features** (SIFT, HOG...), then feed them to a classifier
- Allows to **reduce the dimension** of the classifier
- **Features are invariant** (to rotation, translation, scale, and illumination changes) and allow to robustly classify



Suleiman, A., Sze, V. "Energy-efficient HOG-based object detection at 1080HD 60 fps with multi-scale support." SiPS, 2014.

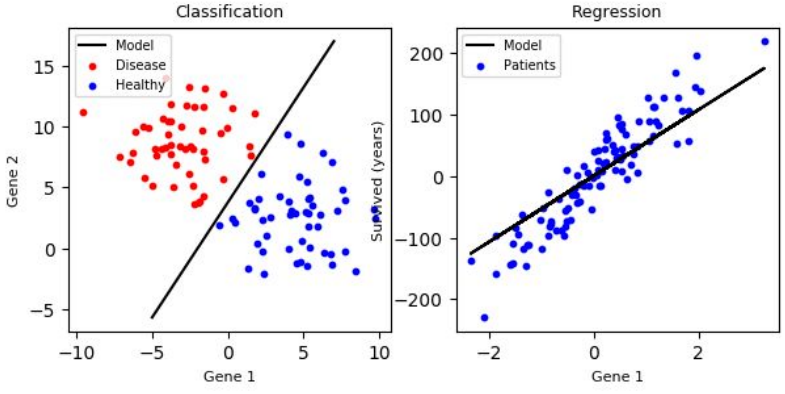
Deep learning approach

- Learn the features at the same time as the classifier
- Features and classifier are coded in the layers of a DNN
- The network is usually trained in an end-to-end way



Classification vs Regression

- Classification
  - Outputs a discrete label
  - Finds decision boundary
- Regression
  - Outputs continuous variable
  - Finds a function

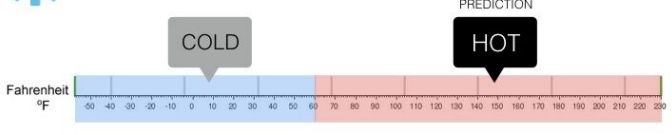


Regression

What is the temperature going to be tomorrow?

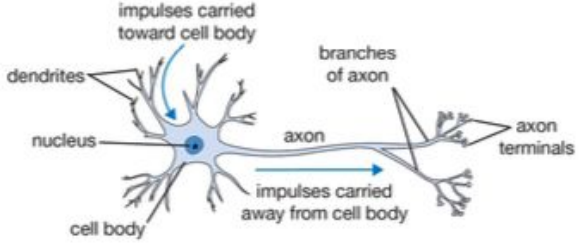
Classification

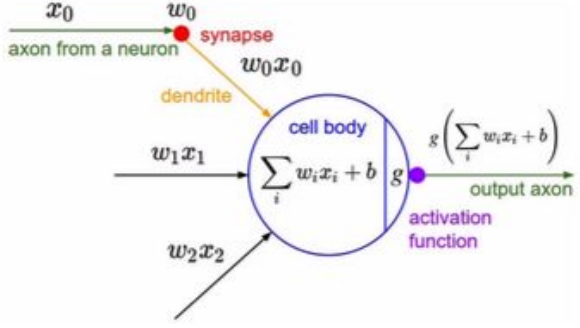
Will it be Cold or Hot tomorrow?

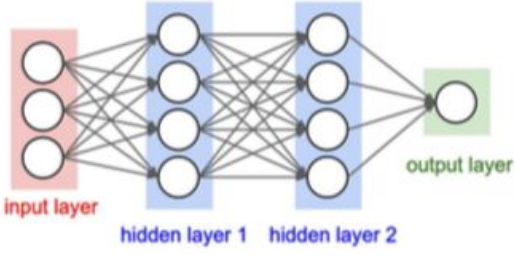


Neural networks

Feedforward Neural Networks







- Neural networks are vaguely inspired on biological neurons
- A neuron/unit is modeled as a composition of an **affine transformation** of its inputs x: $w\,x + b$ and a non-linearity g (**activation function**)

$$f(x) = g(w \cdot x + b)$$

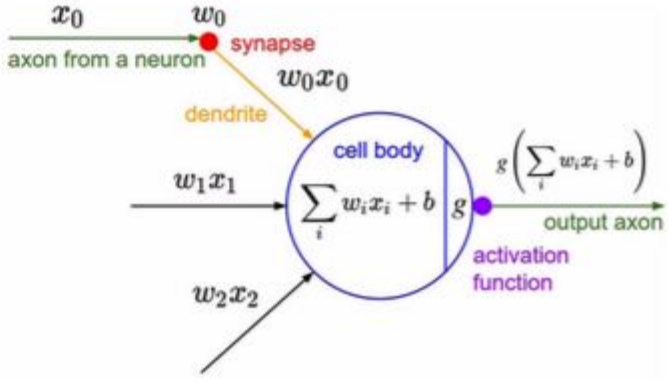
- Often are **grouped in layers**, where each unit is connected to all units from the previous layer

$$y = g_3(b_3 + W_3 \cdot g_2(b_2 + W_2 \cdot g_1(b_1 + W_1 \cdot x)))$$

Perceptron

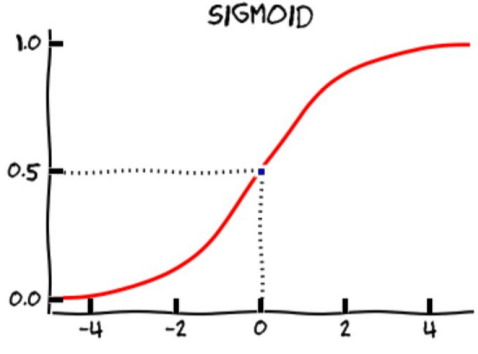
- Binary valued function of its inputs proposed in the 1950's

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0, \\ 0 & otherwise, \end{cases}$$



- The discontinuous Heaviside function makes it hard to train by gradient descent methods
- **Sigmoid** activation is a smooth approximation of Heaviside

$$g(z) = \frac{1}{1 + e^{-z}}$$

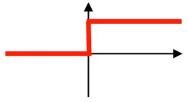
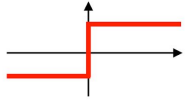
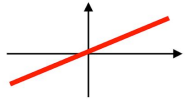
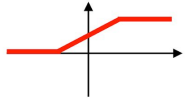
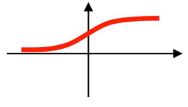
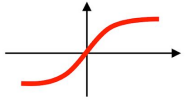
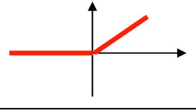
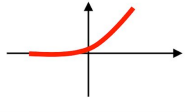


Activation Functions

- **ReLU**: the most frequently used the activation today

$$g(z) = \max(0, z)$$

- Easy to differentiate
- Enable better training of deeper networks

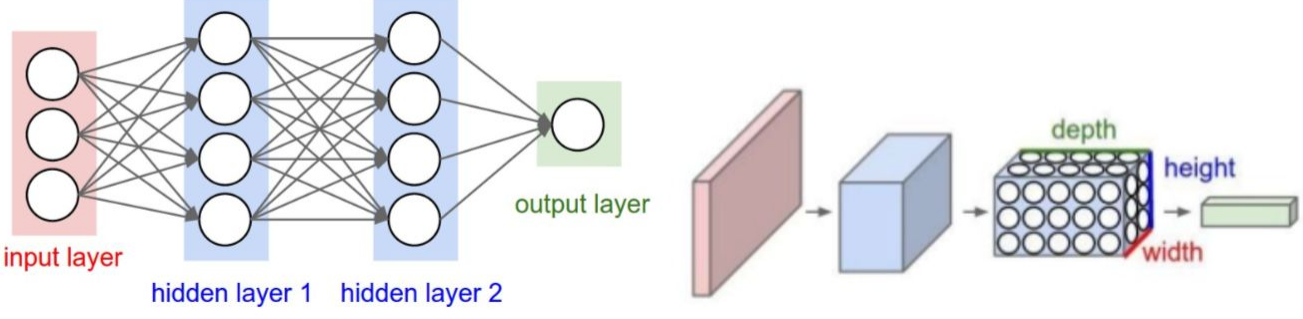
Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

Feedforward Neural Network architecture

- Feedforward networks are often organized in *“layers”*
- The architecture can be specified by an acyclic graph of layers e.g.

$$\mathcal{F}(x) = f_n(f_{n-1}(...(f_2(f_1(x))...))$$

- In image processing and computer vision applications the **input vector has shape *H x W x C*** (height, width, channel)
- ConvNets interpret a layer of neurons as a volume with dimensions (H,W,Depth), which **preserves the spatial structure of the image**



Network layers

- A layer is a map $f_i : A_i \to B_i \text{ with } B_i = A_{i+1}$

- It is customary to define a layer of neurons as the affine transformation together with the activation function. However, it is often **convenient to split the activation function in an independent layer**

- Some common “layers”:
  - Activation: *applies the same nonlinear function g to all its inputs*
  - Fully connected
  - Convolutional
  - Transposed convolution
  - Pooling
  - Batch Normalization (later)

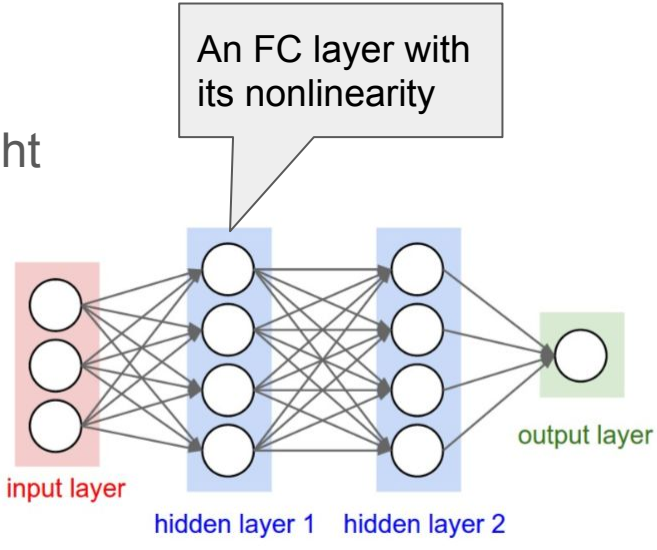
Layers: fully connected (FC)

- Compute an affine transformation of its input (in matrix notation)

$$f(x) = Wx + b$$

$$W \in \mathbb{R}^{c \times d}, b \in \mathbb{R}^{c}, x \in \mathbb{R}^{d}$$

- All possible connections between layer neurons each connection with its own weight
- Contains $c\,(d+1)$ parameters



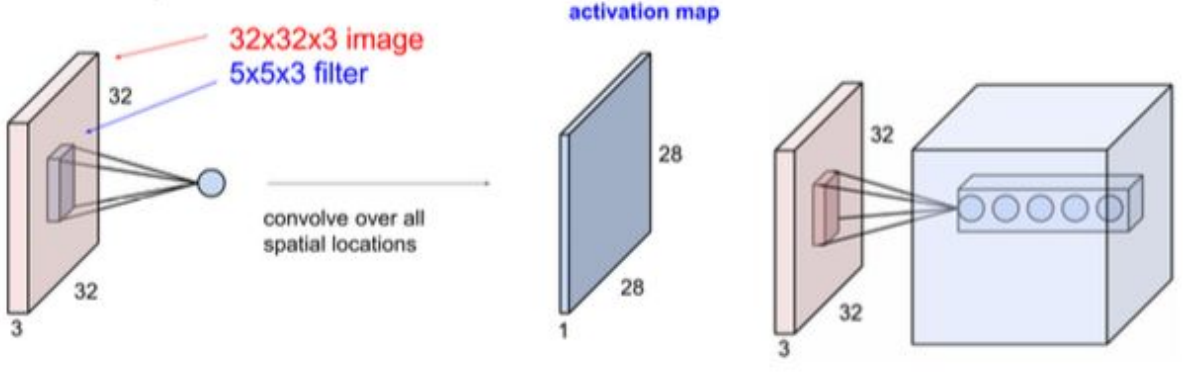
Layers: convolution (Conv)

$$\begin{pmatrix} a & b & 0 & 0 & 0 \\ c & a & b & 0 & 0 \\ 0 & c & a & b & 0 \\ 0 & 0 & c & a & b \\ 0 & 0 & 0 & c & a \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix}$$

- A particular case of FC layer

$$y(i,j,l) = b_l + \sum_{(s,t,k) \in \mathrm{supp}(w_l)} x(i+s, j+t, k) w_l(s,t,k)$$

- Each output map is result of convolving the input with a kernel w_l
- Conv layers involve many more connections than unique weights
  i.e. many connections share the same weight
- Conv layers are translation equivariant



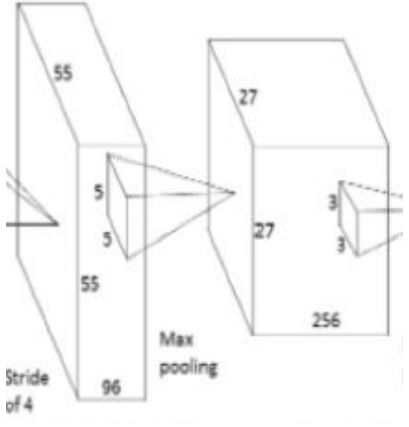
Let’s count some parameters

How many parameters are there in this Conv2D layer?

$y = W x + b$

K = 5x5
In = 96
Out = 256

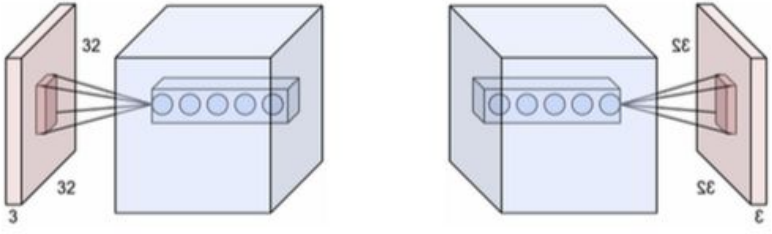
Params: 96 x 5x5 x 256 + 256



About convolutional networks (ConvNets)

Convnets are inspired by the organization of the **animal visual cortex.**
Individual cortical **neurons respond to stimuli only in a restricted region** of the visual field known as the receptive field.
Different neurons respond similarly when presented **to the same stimulus**, which implies that translations do not affect the analysis.
The receptive fields of different neurons partially overlap such that they cover the entire visual field. This particular kind of neural network assumes that we wish to learn filters, in a data-driven fashion, as a means to extract features describing the inputs.

Layers: transposed convolution



Conv Transposed Conv

- “Splats” the kernel on the output layer (similar to aggregation)
  - **Equivalent to a convolution with the rotated kernel if stride=1**
- It is the transpose of the convolution matrix

We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & z & 0 & 0 & 0 \\ 0 & 0 & x & y & z & 0 \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ bx + cy + dz \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=2, padding=1

Convolution transpose multiplies by the transpose of the same matrix:

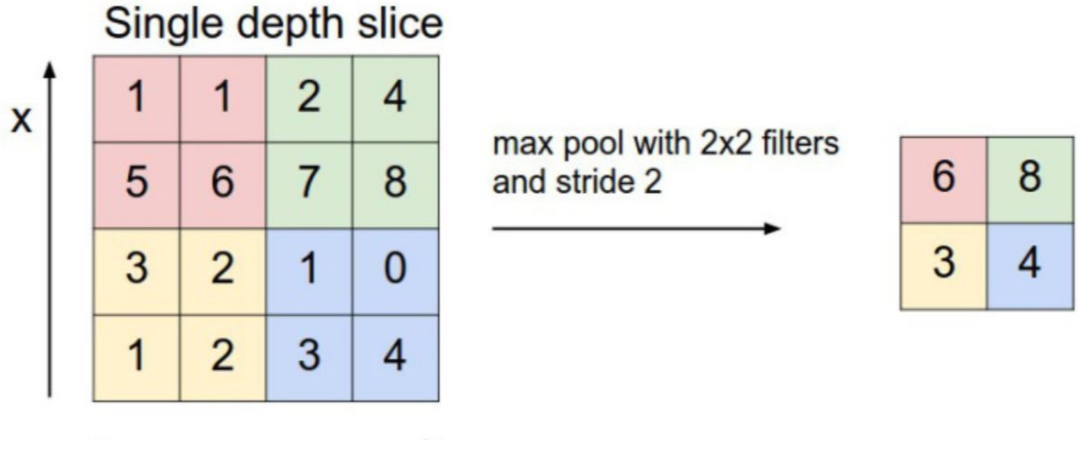
$$\vec{x} *^T \vec{a} = X^T \vec{a}$$

$$\begin{bmatrix} x & 0 \\ y & 0 \\ z & x \\ 0 & y \\ 0 & z \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az + bx \\ by \\ bz \\ 0 \end{bmatrix}$$

When stride>1, convolution transpose is no longer a normal convolution!

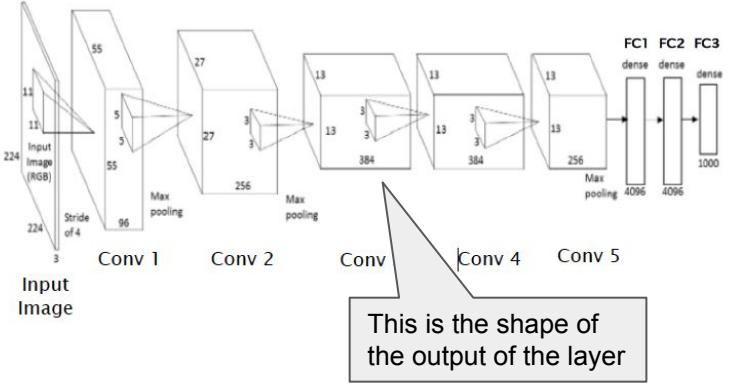
Layer: Pooling (POOL)

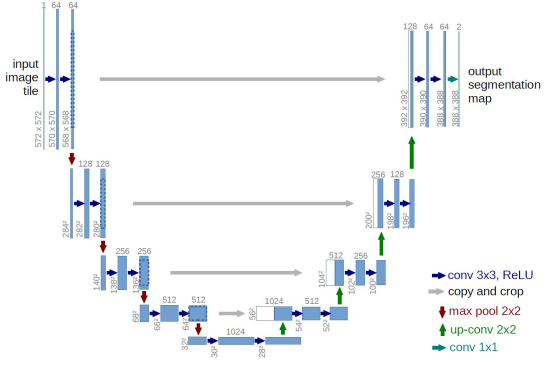
- Spatial subsampling by **binning of the input features**
- Max Pooling is the most common but average pooling also feasible
- Provides more translation invariance in the feature maps
- The current trend is to use strided convolution instead of pooling

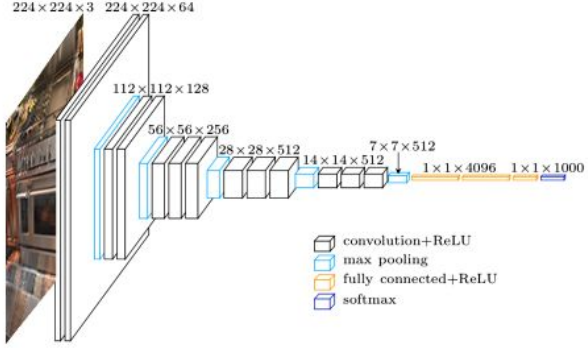


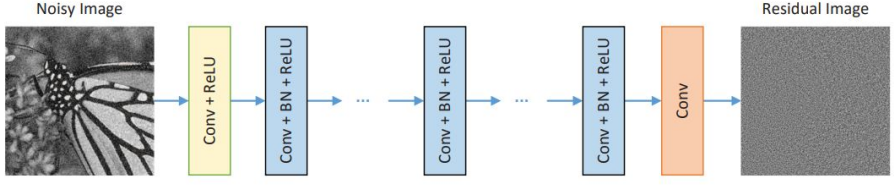
Notes on the ConvNet architectures

- ConvNet architectures vary depending on the application: Encoders, Hourglass Fully convolutional, ...
- **Depth** (hence deep) is a common trait
- Diagrams tend to omit many details

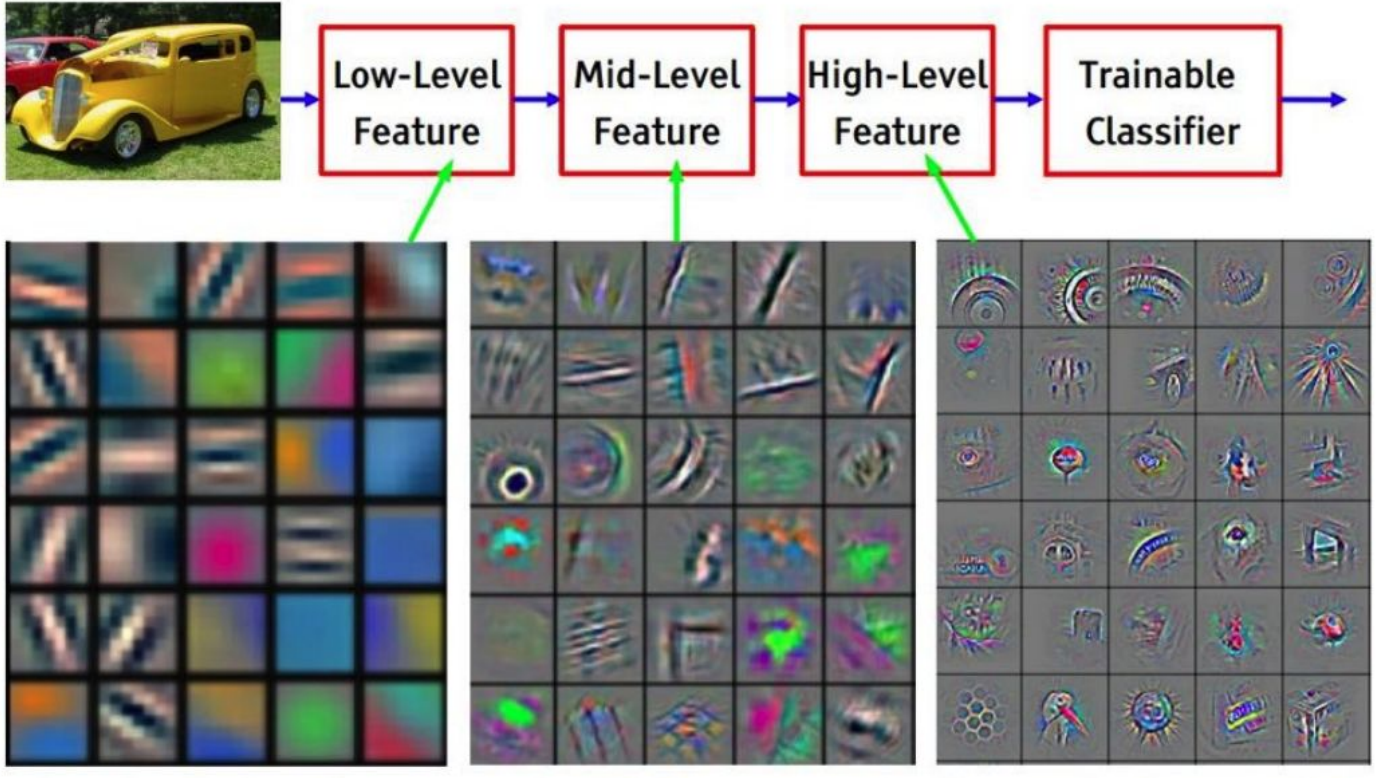








The hierarchical layer structure allows to learn hierarchical filters (features)



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

Slide credit: Yann LeCun

Representation Power of NNs

Universal Approximation theorems

- NN with one hidden layer (arbitrar width) can represent [Cybenko 1989]
  - Any bounded continuous function (to arbitrary error)
  - Any Boolean function exactly

- Arbitrary depth [Zhou Lu et al. in 2017]

Universal approximation theorem (L1 distance, ReLU activation, arbitrary depth). For any Lebesgue-integrable function $f : \mathbb{R}^n \to \mathbb{R}$ and any $\epsilon > 0$, there exists a fully-connected ReLU network $\mathcal{A}$ with width $d_m \leq n + 4$, such that the function $F_{\mathcal{A}}$ represented by this network satisfies

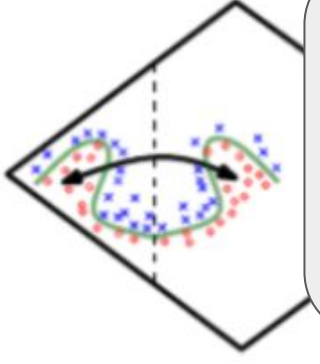
$$\int_{\mathbb{R}^n} |f(x) - F_{\mathcal{A}}(x)| \, \mathrm{d}x < \epsilon$$

- ...

Approximation power of deep architectures

Why deep architectures?

- The *universal approximation theorem* states that a network with one hidden layer can represent any function
  - But the number of neurons required may be unfeasibly large
- Using **deeper** models can exponentially **reduce the number of units required** to represent the desired function
  - An intuitive geometric explanation of this, using the absolute value nonlinearity, is:



- How do we find these models?
- Can we even find them?

Figure: Montufar et al. *On the Number of Linear Regions of Deep Neural Networks*. 2014

Reload this page

Tinker With a **Neural Network** Right Here in Your Browser.
Don't Worry, You Can't Break It. We Promise.

Epoch 000,000

Learning rate: 0.03

Activation: Tanh

Regularization: None

Regularization rate: 0

Problem type: Classification

DATA

Which dataset do you want to use?

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

FEATURES

Which properties do you want to feed in?

X_1

X_2

X_1^2

X_2^2

X_1X_2

2 HIDDEN LAYERS

4 neurons

2 neurons

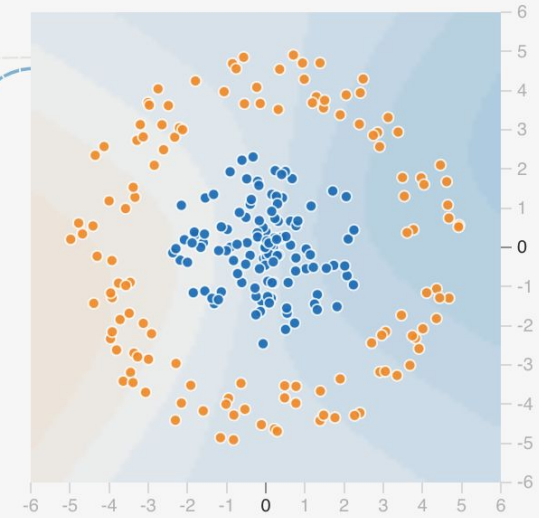
*The outputs are mixed with varying **weights**, shown by the thickness of the lines.*

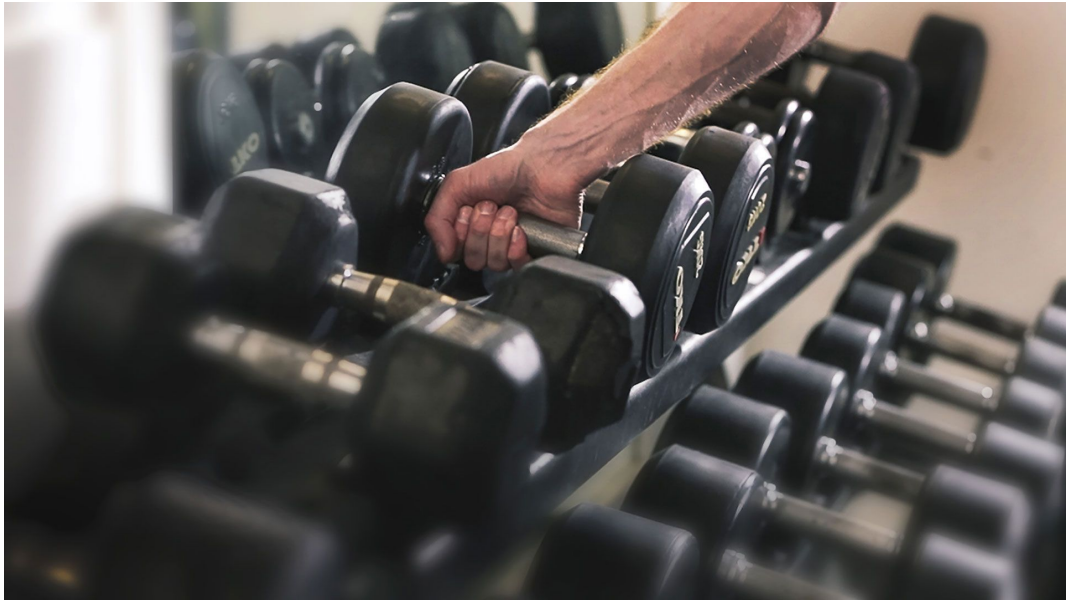
*This is the output from one **neuron**. Hover to see it larger.*

OUTPUT

Test loss 0.504

Training loss 0.512





Training

Training

- Given a large number of sample input-output pairs (x,y) of the problem
- We'd like to optimize the parameters θ of the model to **minimize the risk** (expected value of the loss)

$$R(\mathcal{F}_\theta) = \int \ell(\mathcal{F}_\theta(x), y) p(x, y) dx dy$$

- Since the density of data $p(x,y)$ is unknown, empirical risk is used instead

$$R^{\text{emp}}(\mathcal{F}_\theta) = \sum_i \ell(\mathcal{F}_\theta(x_i), y_i)$$

- The challenge is that the learned model **generalizes** well to unseen data
  - **Attention!** The distribution of the training data must be representative of the real density else we have what is called ***dataset bias***.
  - And the training dataset must be large enough

Training

Supervised learning checklist:

- **Labeled training data**: *many pairs of noisy and noiseless images*
- **A model**: *the parametric function* $\mathcal{F}_\theta$
- **A loss function**: *defines the goal of the algorithm, usually a norm between output and label*
- **An optimizer**: *that updates the model parameters so as to minimize the empirical loss computed on the training data*

Overfitting and validation

- The objective of training is to **fit the parameters of the model** to minimize the empirical risk

$$E_{train} = \sum_{i=1}^{n_{train}} \ell(\mathcal{F}_\theta(x_i), y_i)$$

while being able to **generalize to unobserved data** $E_{test} = \sum_{i=1}^{n_{test}} \ell(\mathcal{F}_\theta(x_i), y_i)$

- The problem would be to train a model that **overfits to the training set**

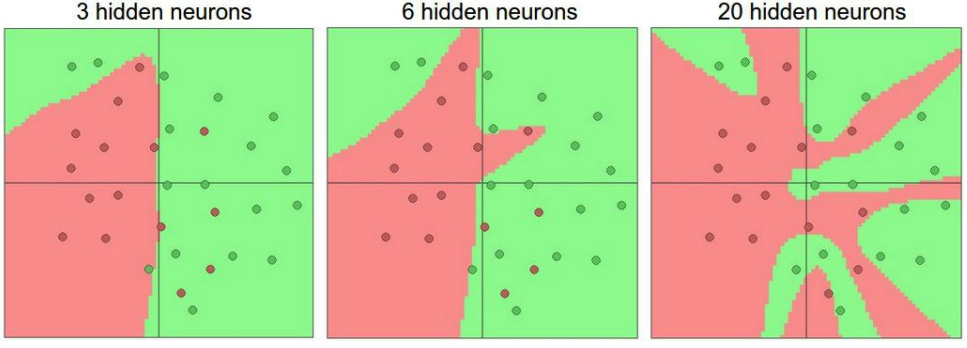
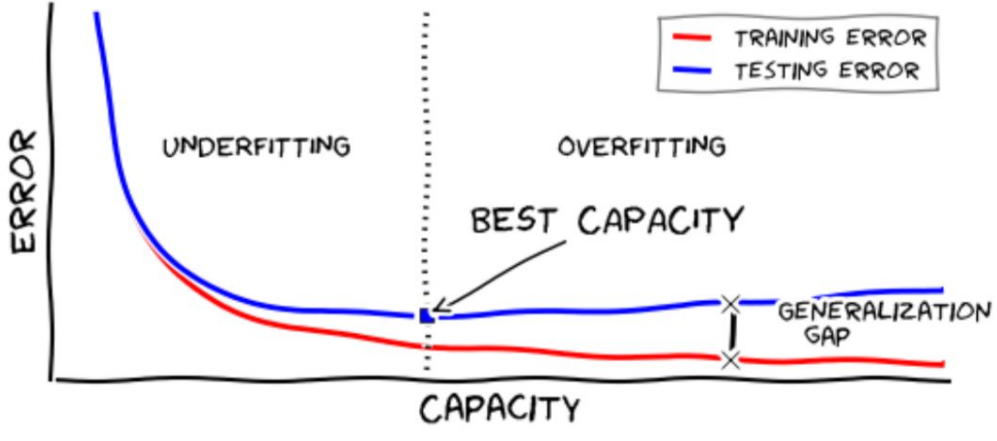


Image from the online demo by Andrej Karpathy:
http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

- ***The capacity* of the model** (ability to learn/overfit) is controlled by: function space (i.e. polynomials of degree n), regularization, and number of parameters

Overfitting and validation

- Defining and estimating the capacity of a NN is still an active research topic. **But we can detect the symptoms of overfitting**.
- The **dataset is split in training, validation, and test sets**
  - Test is used to evaluate the final network. Should only be used once for the final assessment of the performance of the model.
  - Validation is used to monitor the generalization performance during training, allowing to spot overfitting, and tune hyperparameters
  - When train and validation errors diverge too much it is probably due to overfitting



Optimization

- Stochastic gradient descent is simple
  - Approximates the gradient of the risk with a small set of training samples (**mini-batch**)
  - Computes the gradient of the mini-batch risk wrt all the parameters and updates them
  - Learning rate τ: controls the step size. It is a very delicate hyperparameter

Algorithm 24: Stochastic gradient descent.

while *stopping criterion not met* **do**
  Sample mini-batch of m samples $x_1, x_2, ..., x_m$ and corresponding targets y_i;
  Compute gradient estimate: $\Delta\theta \longleftarrow \frac{1}{m}\nabla_\theta \sum_i \ell(\mathcal{F}_\theta(x_i), y_i)$
  Update the parameters: $\theta \longleftarrow \theta - \tau \cdot \Delta\theta$

- In practice use adaptive gradient methods with momentum
  - We will use ADAM (Adaptive Moment Optimization) [Kingma, Ba 2014]
- Second order methods also exist ...

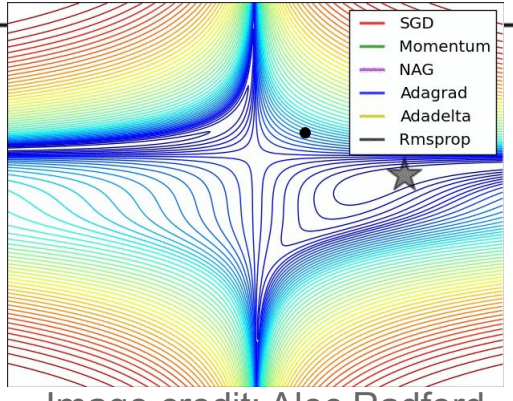
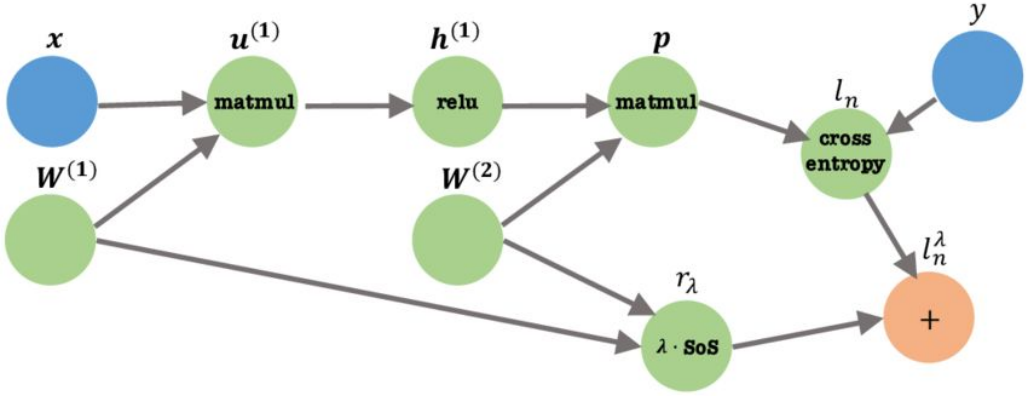


Image credit: Alec Radford

Computational graphs

- Networks and losses are coded using computational graphs
  - This is done by coding using specific packages such as *torch, tensorflow or jax* …
- Since all operations are traced in the graph they enable automatic differentiation and thus the computation of the gradients needed for SGD



Computation of the derivatives using the chain rule

- Compute the differential of a composite function $f(x) = h(g(x))$
- Chain rule says $\frac{\partial f}{\partial x} = \frac{\partial h}{\partial g}\frac{\partial g}{\partial x}$ where all terms are Jacobian matrices
- For scalar functions that is $\frac{\partial f}{\partial x} = h'(g(x))g'(x)$
- Let's compose f with k.

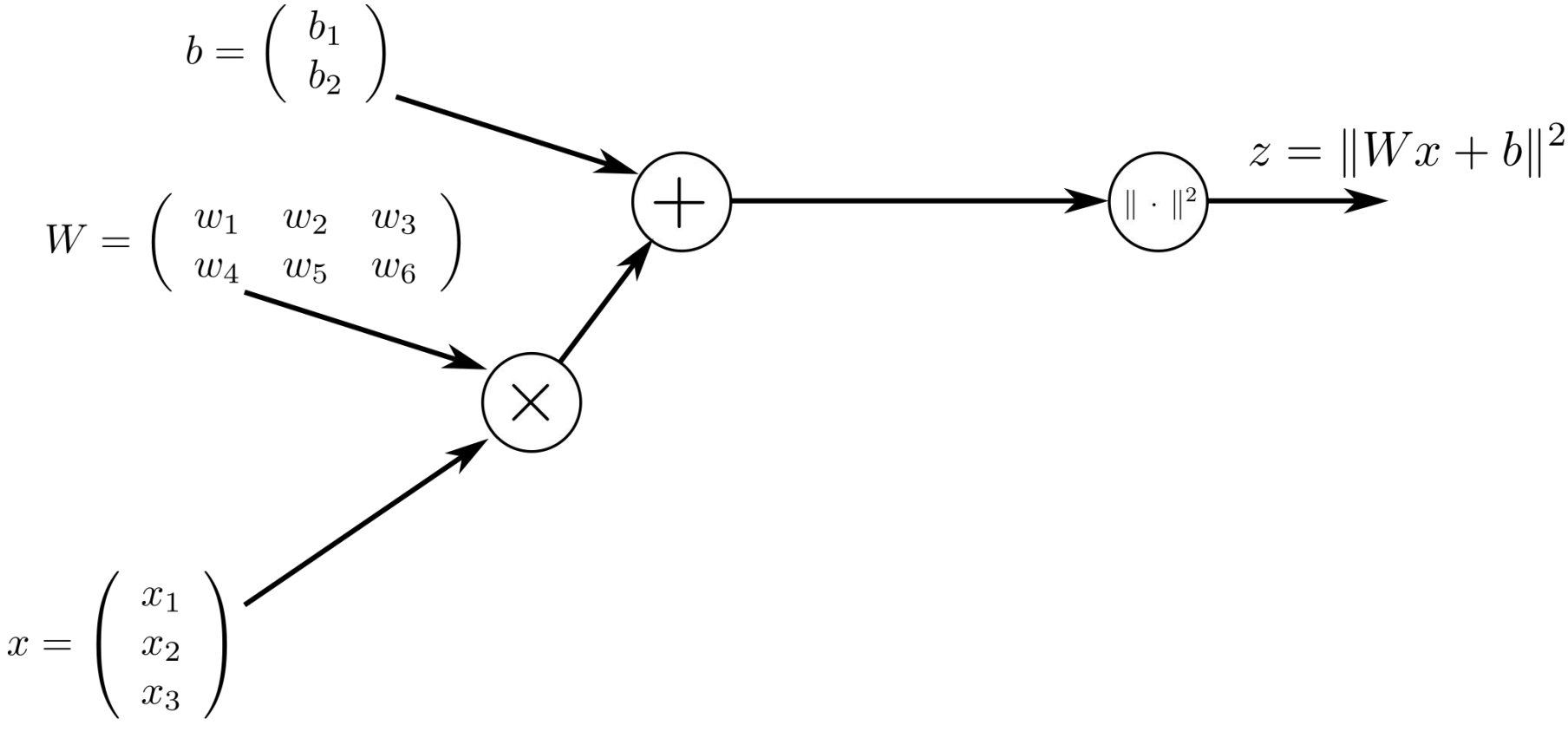
  See any pattern?

$$e(x) = k(f(x))$$

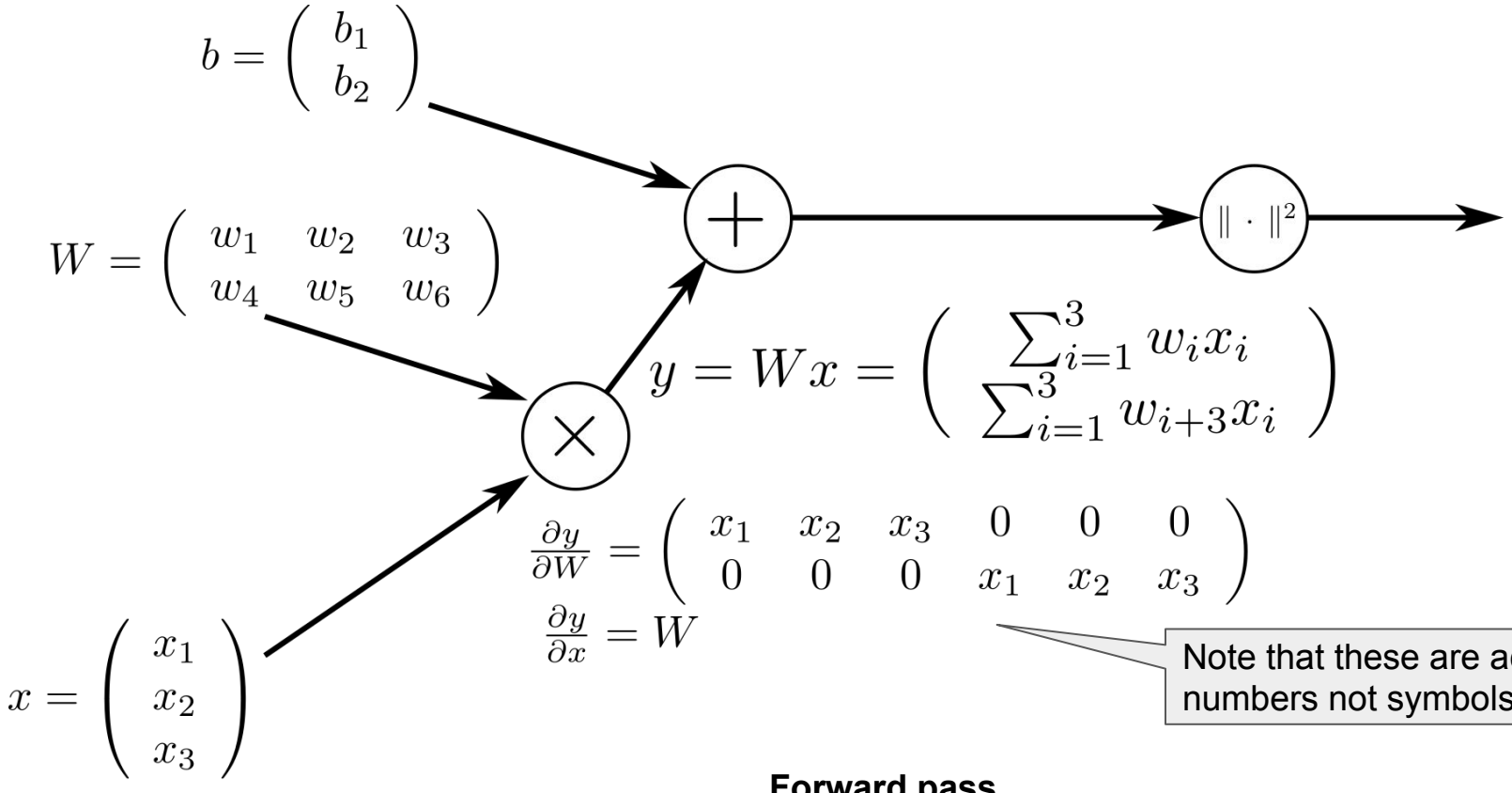
$$e(x) = k(h(g(x)))$$

$$\frac{\partial e}{\partial x} = \underbrace{k'(h(g(x)))}_{\frac{\partial e}{\partial f}} \underbrace{h'(g(x))g'(x)}_{\frac{\partial f}{\partial x}}$$

Backpropagation by example $z = \|Wx + b\|^2$

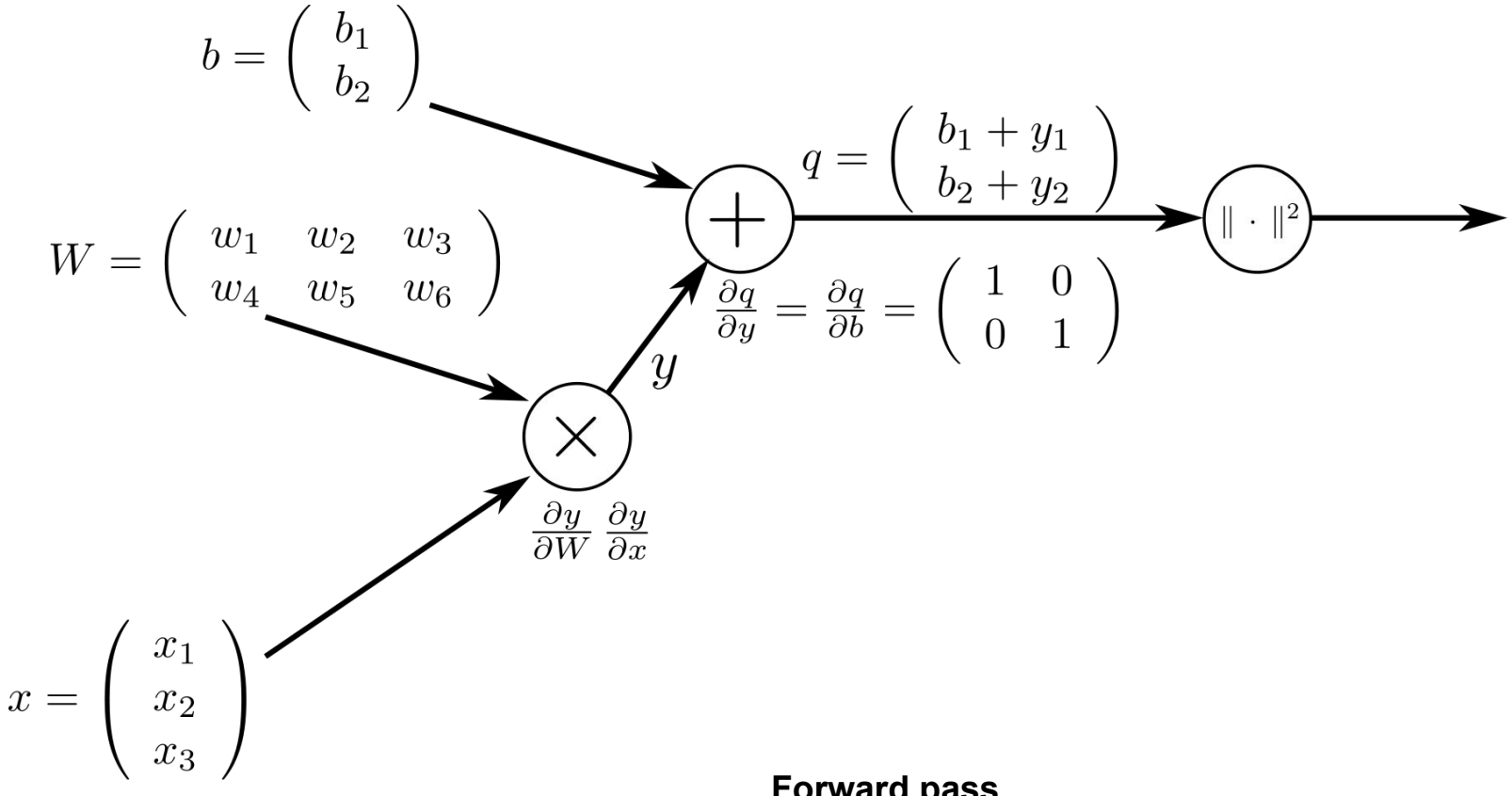


Backpropagation by example $z = \|Wx + b\|^2$



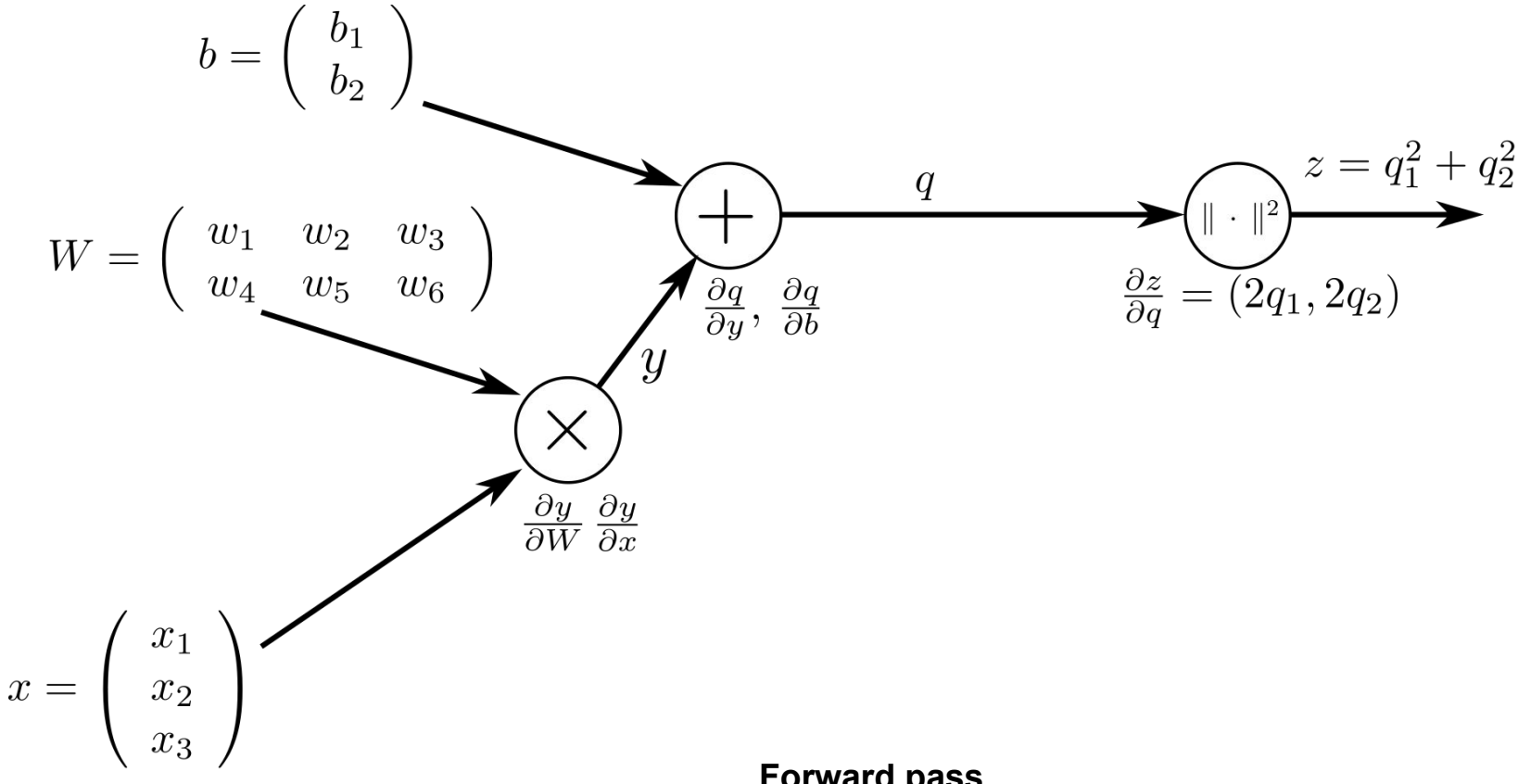
Forward pass

Backpropagation by example $z = \|Wx + b\|^2$



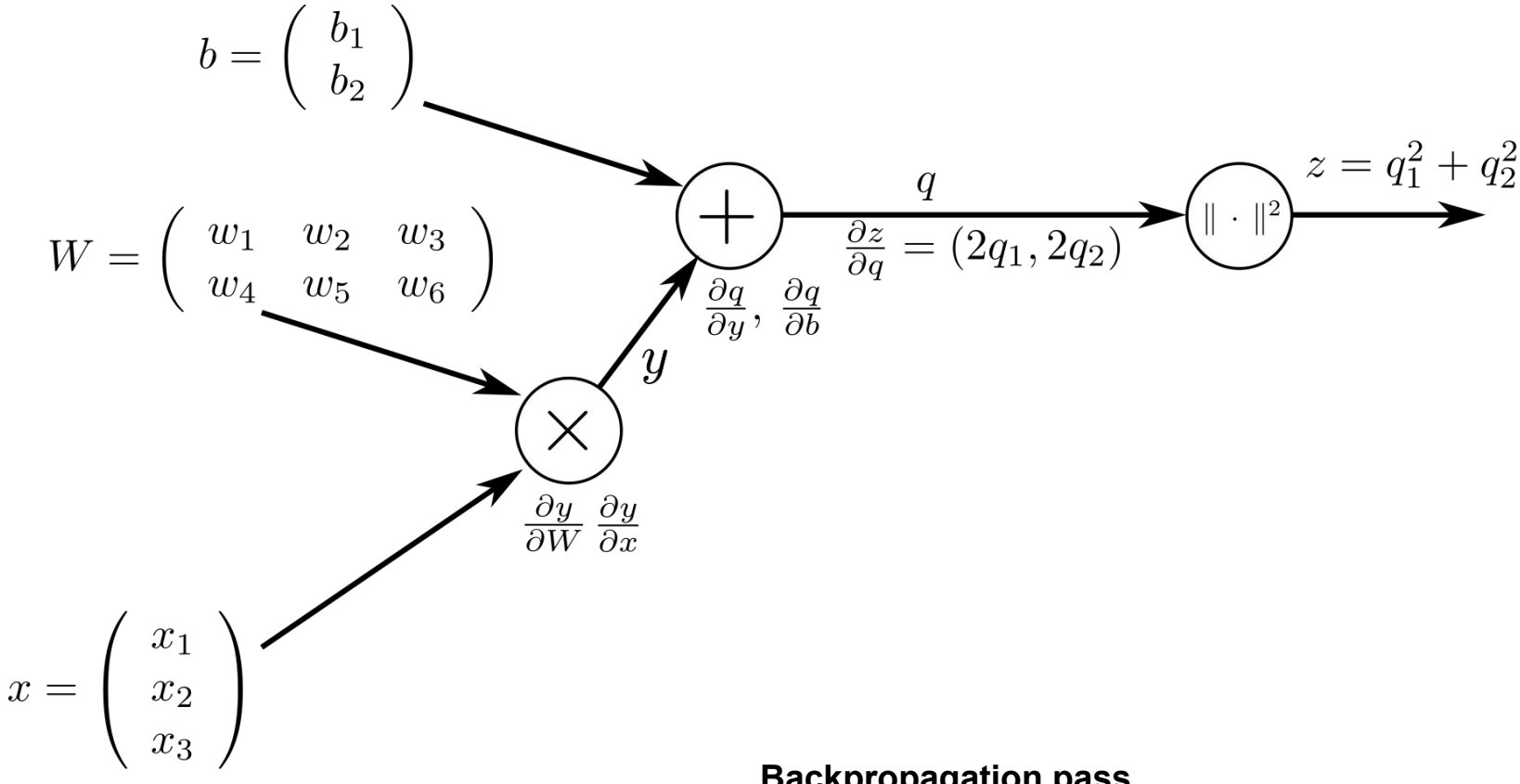
Forward pass

Backpropagation by example $z = \|Wx + b\|^2$



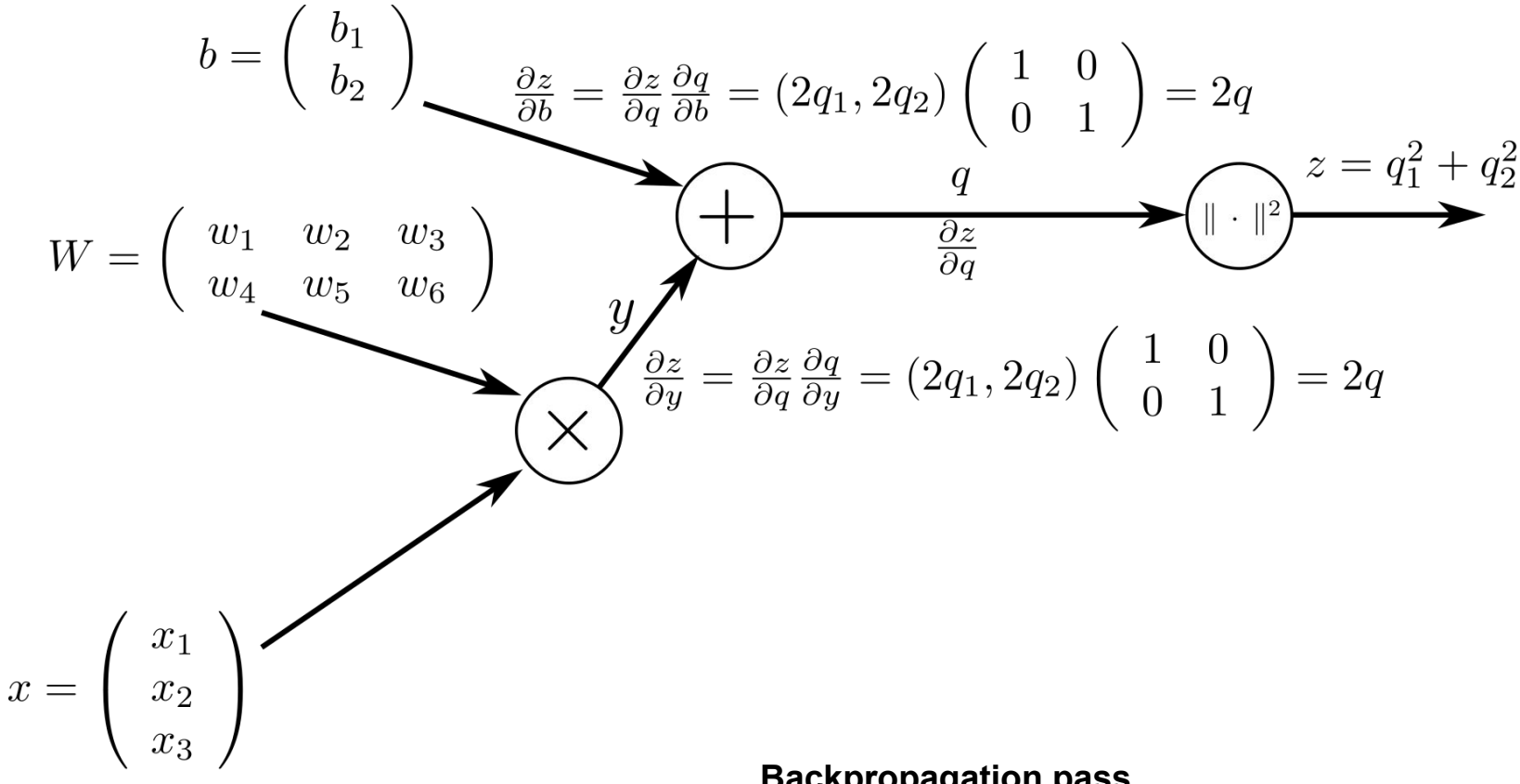
Forward pass

Backpropagation by example $z = \|Wx + b\|^2$



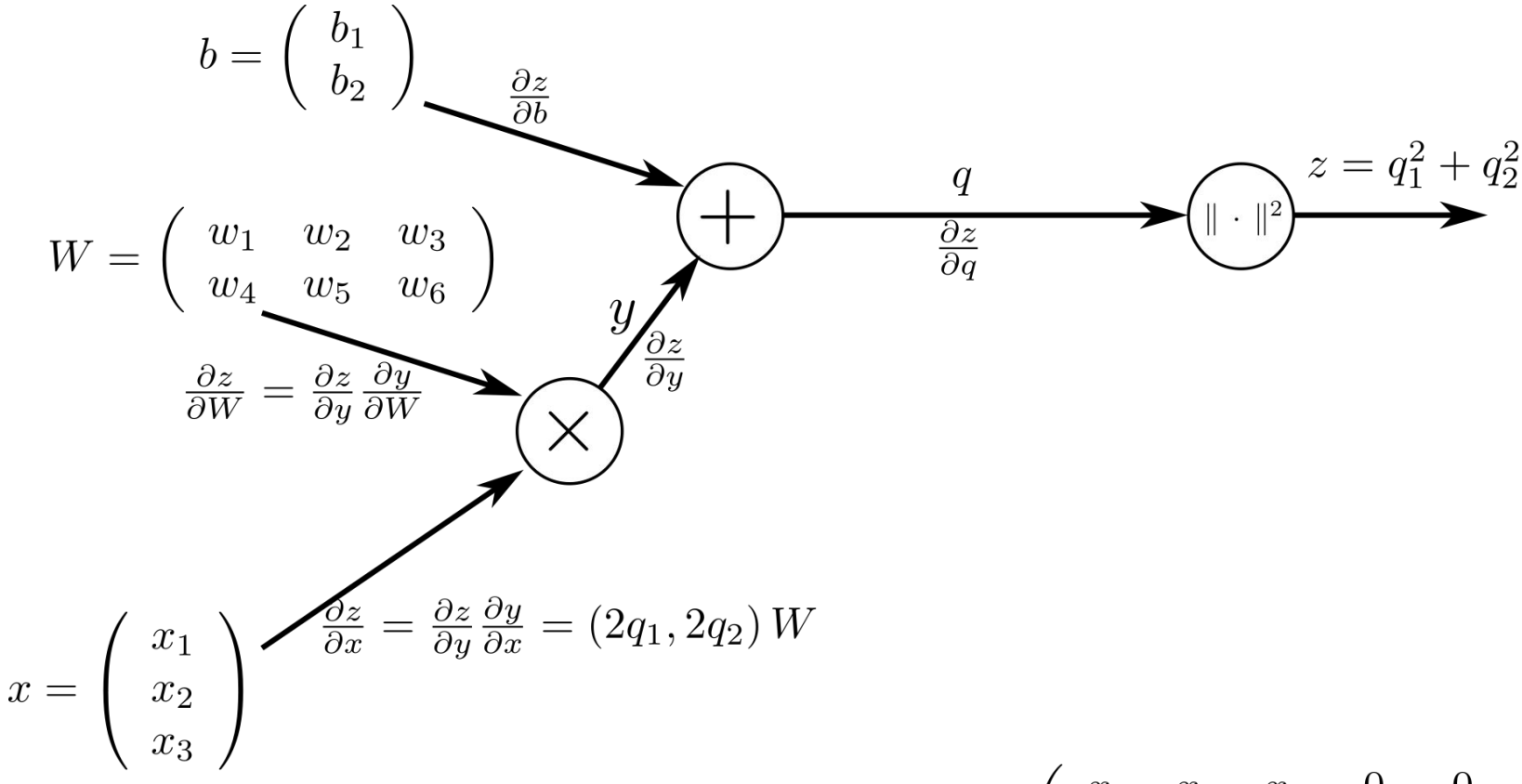
Backpropagation pass

Backpropagation by example $z = \|Wx + b\|^2$



Backpropagation pass

Backpropagation by example $z = \|Wx + b\|^2$



$$\frac{\partial z}{\partial W} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial W} = (2q_1, 2q_2) \begin{pmatrix} x_1 & x_2 & x_3 & 0 & 0 & 0 \\ 0 & 0 & 0 & x_1 & x_2 & x_3 \end{pmatrix}$$

Backpropagation example

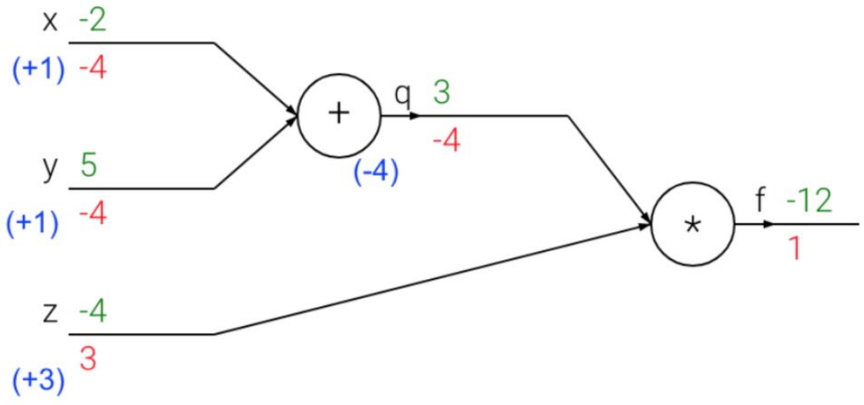


Figure 10.10: Computational graph of $f(x, y, z) = (x + y)z$. The forward pass computes values from inputs to output (shown in green). During the forward pass also the local gradients of the gates are computed (shown in blue). The backward pass then performs backpropagation which starts at the end and recursively applies the chain rule to compute the gradients (shown in red) all the way to the inputs of the circuit. The gradients can be thought of as flowing backwards through the graph. Figure reproduced from `http://cs231n.github.io/`.

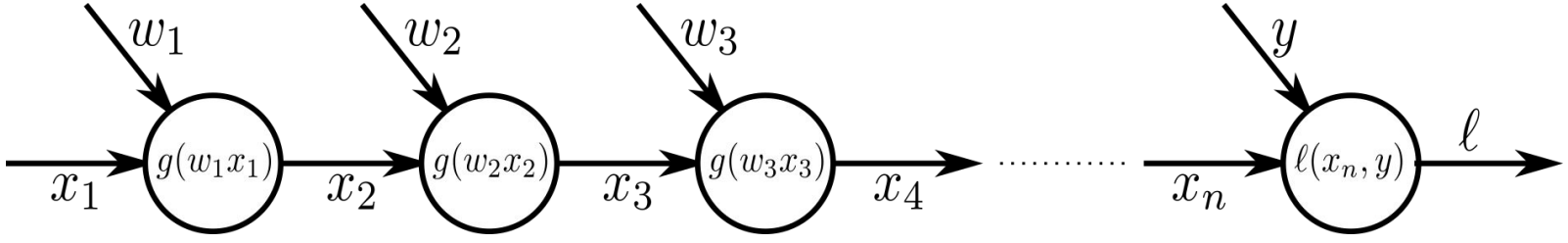
Yes you should understand backprop

Andrej Karpathy Dec 19, 2016 · 7 min read

The problem with Backpropagation is that it is a leaky abstraction.
… it's easy to believe that you can simply stack arbitrary layers together and backprop will "magically make them work" on your data.

Vanishing gradients

In deep networks the gradient vanishes during backprop

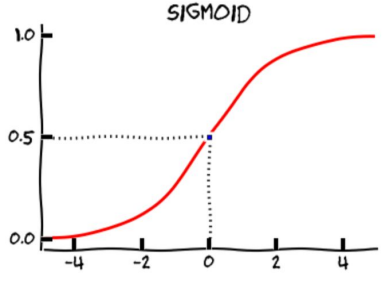


$$\frac{\partial \ell}{\partial w_2} = \frac{\partial x_3}{\partial w_2}\frac{\partial \ell}{\partial x_3}$$

$$\frac{\partial \ell}{\partial w_2} = \frac{\partial x_3}{\partial w_2}\frac{\partial x_4}{\partial x_3}\cdots\frac{\partial \ell}{\partial x_n}$$

$$\frac{\partial \ell}{\partial w_2} = g'(w_2x_2)x_2\prod_{i=3}^{n} g'(w_ix_i)w_i\frac{\partial \ell}{\partial x_n}$$

If one of these terms becomes too small then the gradient wrt w_2 will vanish

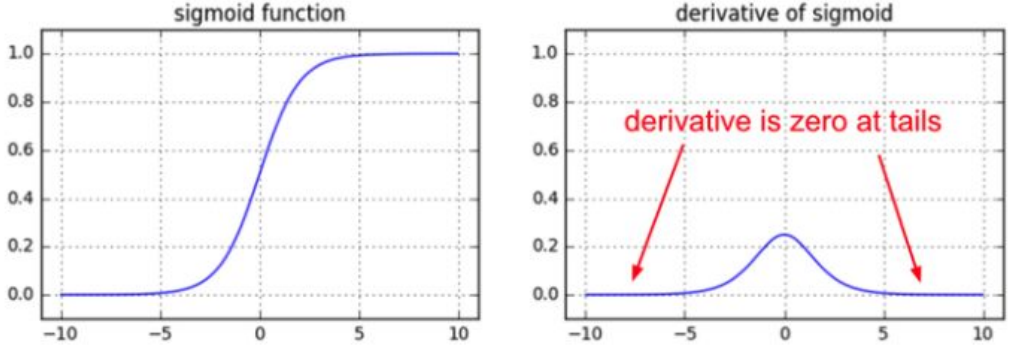


Vanishing gradients

When using sigmoid or tanh nonlinearities

```
z = 1/(1 + np.exp(-np.dot(W, x))) # forward pass sigmoid
dx = np.dot(W.T, z*(1-z)) # backward pass: local gradient for x
dW = np.outer(z*(1-z), x) # backward pass: local gradient for W
```

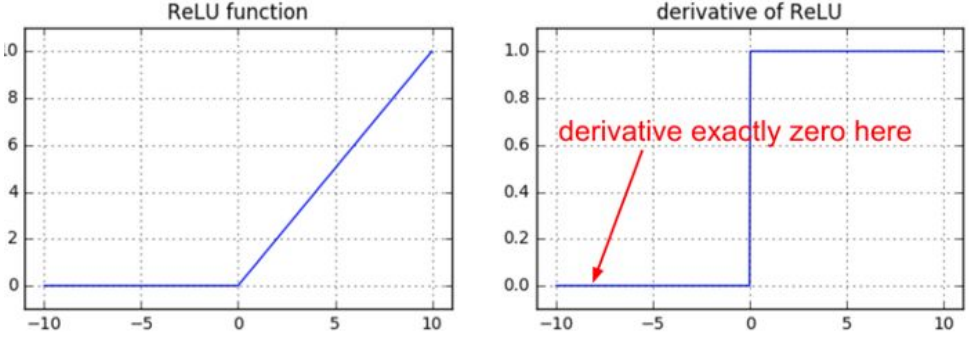
- If weights W are initialized too large the output *z* also becomes large, which **nulls all the gradients on the rest of the backward pass**
- The sigmoid local gradient *(z*(1-z))* achieves a maximum at 0.25, so stacking many sigmoids leads to a gradient that vanishes with depth



Dying ReLUs

If a ReLU neuron initialized such that it never fires then it will never change

```
z = np.maximum(0, np.dot(W, x)) # forward pass
dW = np.outer(z > 0, x) # backward pass: local gradient for W
```



Weight initialization

- Initialize weights so that **outputs of the affine layers are close to 0**, where the nonlinearity of the activation function takes place
  - Typically initialized following a Gaussian distribution of a small standard deviation
- Note that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs

$$Var(y) = Var(\sum^{n} w_i x_i) = \sum^{n} Var(w_i x_i) = \sum^{n} Var(x_i) Var(w_i) = (nVar(w)) Var(x_i)$$

  - This is a problem for deep networks
  - To ensure that the variance of the output is the same as the input the standard deviation of the initial weights could be set to $\frac{1}{\sqrt{n_{\text{in}}}}$
  - A refined analysis of the effect of the ReLU activation leads to std. $\sqrt{\frac{2}{n_{\text{in}}}}$

- Initialization has become less critical with the introduction of BatchNorm

Easing the learning

Batch normalization (BN)

Controls the mean and variance of the distribution of inputs to an activation function

- During the training of a deep network the distribution of inputs to a neuron can drift away from the nonlinearity (this is the *"internal covariance shift"*)
- **BN centers** these values on the **relevant zone of the activation** function
- It allows to train deeper network and alleviates **the vanishing gradient**

Algorithm 2: BN layer. During inference only steps 3 and 4 are applied where $\mu_B, \sigma_B^2, \gamma$, and β are those computed during the training.

input : Output values h of affine neuron over mini-batch $B = \{x_1, \cdots, x_M\}$
Target mean and variance parameters β, γ^2.

output: $\hat{h}_i = BN_{\gamma,\beta}(h_i)$

1 $\mu_B = \frac{1}{M}\sum_{i=1}^{M} h_i$ // *mini-batch mean*

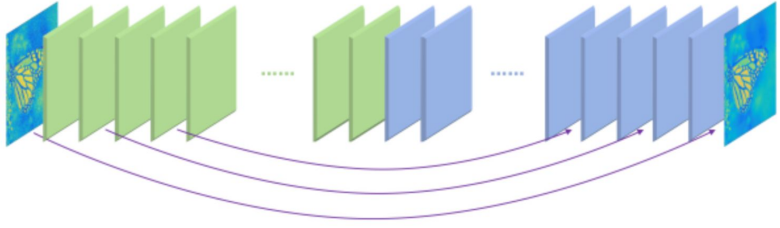
2 $\sigma_B^2 = \frac{1}{M}\sum_{i=1}^{M}(h_i - \mu_B)^2$ // *mini-batch variance*

3 $\tilde{h}_i = \frac{h_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ // *normalization*

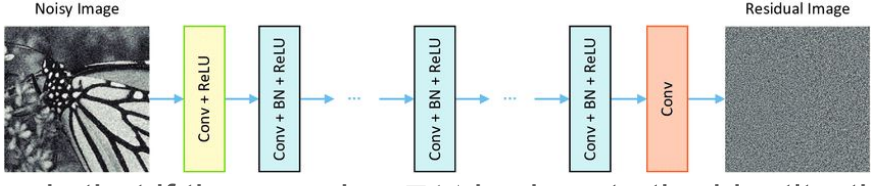
4 $\hat{h}_i = \gamma\tilde{h}_i + \beta$ // *scale and shift*

Skip connections / residual learning

- Skip connections aim at facilitating the training of deeper networks by attenuating vanishing gradient



- Residual learning amounts to creating a skip connection from the input to the output of the network i.e. $F(x) = x + Net(x)$



  - The intuition is that if the mapping $F(x)$ is close to the identity, then it is easier to learn the residual mapping $Net(x)$

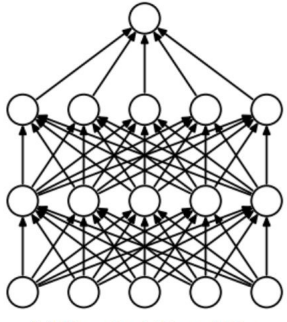
Regularization

Aims at improving generalization (prevent overfitting) by indirectly controlling the capacity of the model

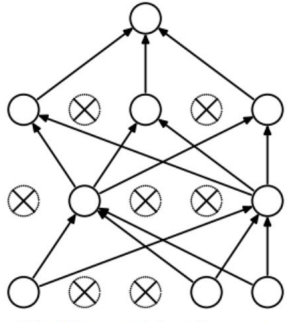
- **L2 or *weight decay***
  - Adds a term $+ \lambda \, ||W||^2$ to the empirical risk function
    Its gradient is $-W$ which reduces the weights towards 0 at each iteration

- **L1 regularization**
  - Adds a term $+ \lambda \, ||W||_1$ to the empirical risk function
    Its minimization enforces sparsity of the weights

- **Dropout**
  - Randomly removes neurons during a batch update
  - The aim is to enforce that all neurons are used



(a) Standard Neural Net

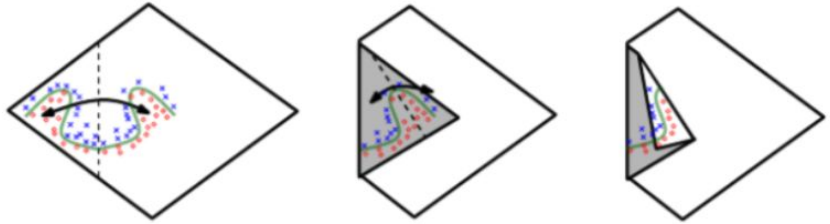


(b) After applying dropout.

How NNs even work?

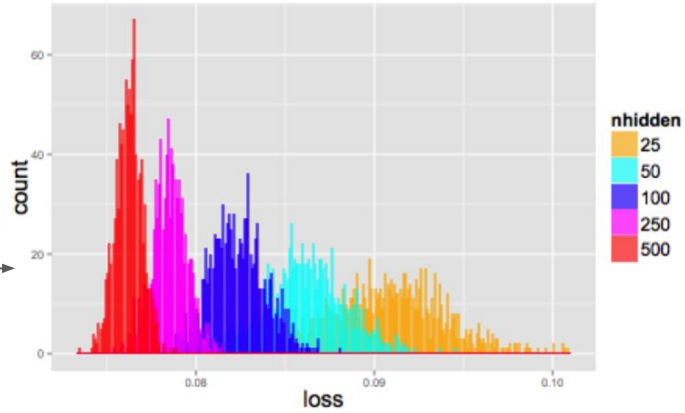
Deep thoughts about deep networks

- We have seen that deep neural networks can represent any function (representation theorem)
  - But the optimal network configuration **may be very hard to find** with a SGD algorithm



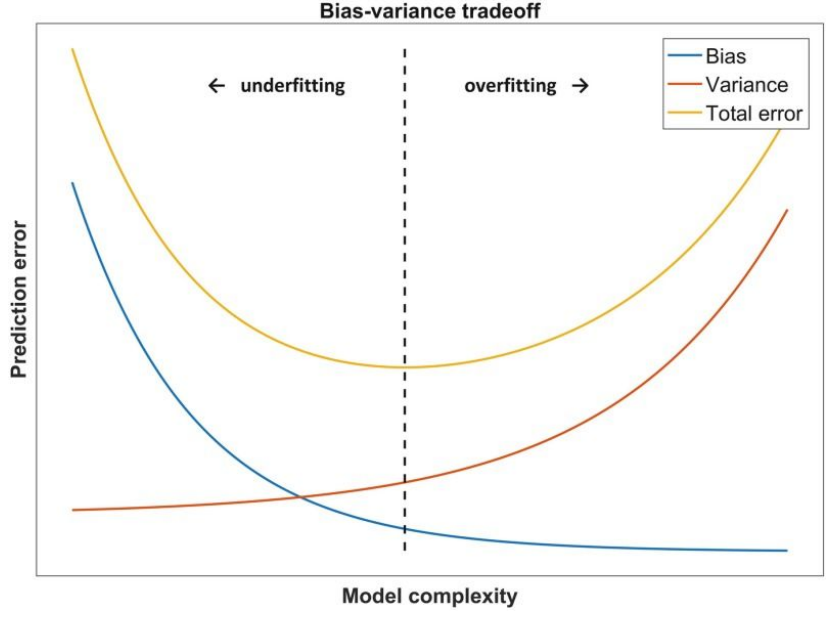
- However, a SGD training is more likely to find a “good minimum” in a **bigger network** (many good minima)

Histograms of loss values for 1000 trainings with SGD varying the number of hidden units in a network with a single hidden layer. [Choromanska et.al 2015]



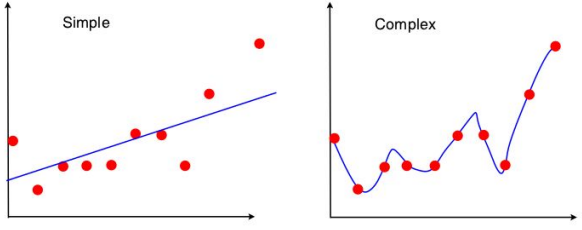
The bias-variance tradeoff

- ML tries to approximate an ideal target function *f* with a model
- Hypothesis set H: set of all possible models we consider, e.g. $y = f(x) + \varepsilon,$
  - **How good** is my hypothesis to approximate the target?
  - **Is it hard to find** a solution in the set that approximates my target function?



The bias-variance tradeoff

- Suppose that the ideal model *f* explains the data: $y = f(x) + \varepsilon$, the noise eps has variance σ^2
- We want to find a function $\hat{f}(x; D)$ of the training data *D* that approximates *f* as well as possible (in the MSE sense) $(y - \hat{f}(x; D))^2$



- The expected error on an unseen sample can be decomposed as

$$\mathrm{E}_D\left[\left(y - \hat{f}(x; D)\right)^2\right] = \left(\mathrm{Bias}_D\left[\hat{f}(x; D)\right]\right)^2 + \mathrm{Var}_D\left[\hat{f}(x; D)\right] + \sigma^2$$

with

$$\mathrm{Bias}_D\left[\hat{f}(x; D)\right] = \mathrm{E}_D\left[\hat{f}(x; D)\right] - f(x)$$

$$\mathrm{Var}_D\left[\hat{f}(x; D)\right] = \mathrm{E}_D[\left(\mathrm{E}_D[\hat{f}(x; D)] - \hat{f}(x; D)\right)^2].$$

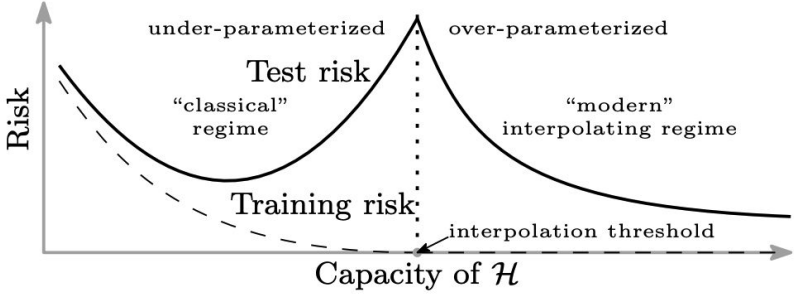
Bias-variance tradeoff and NNs

Reconciling modern machine learning practice and the bias-variance trade-off

Mikhail Belkin[a], Daniel Hsu[b], Siyuan Ma[a], and Soumik Mandal[a]

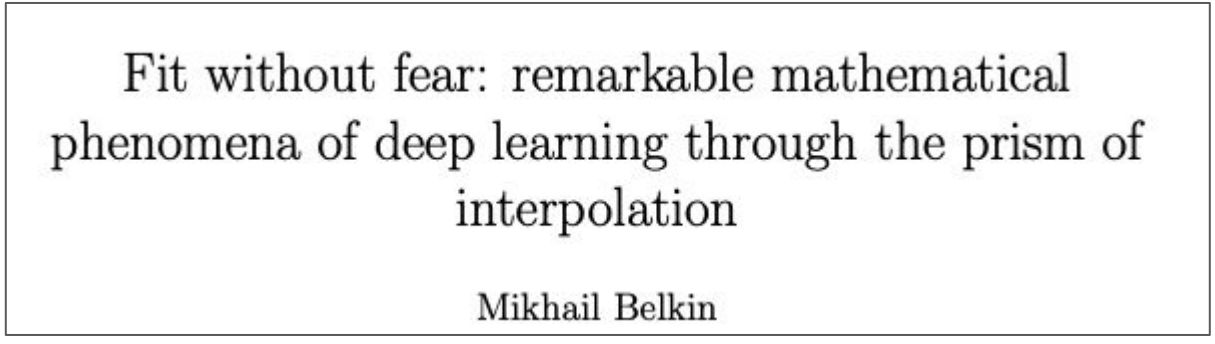
2018

- NNs have millions of parameter but they generalize extremely well!
- Observes a *"double descent"* phenomenon, i.e. test risk re-descends for over-parametrized models



- No good explanation …

Overparameterization is good



2021

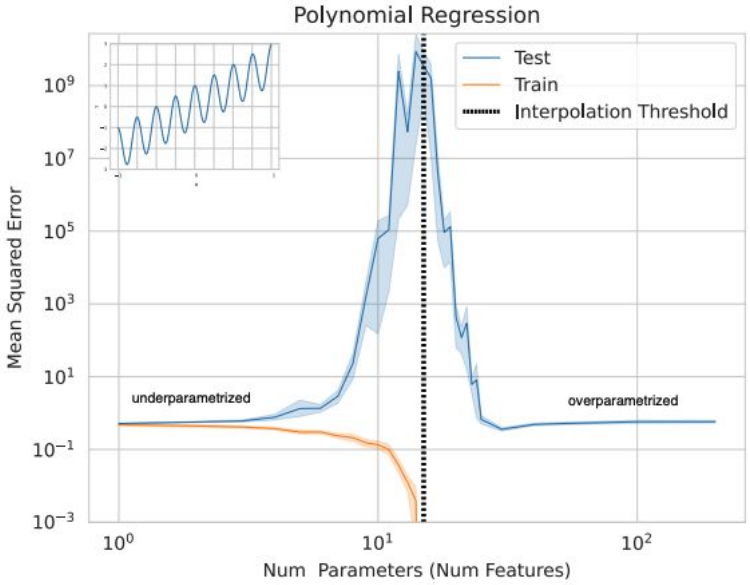
“you want to make sure you hit the zero training error. Because if you don't, you somehow waste the capacity of the model.”

Overparameterization is good

Double Descent Demystified: Identifying, Interpreting & Ablating the Sources of a Deep Learning Puzzle

Rylan Schaeffer[1], Mikail Khona[2], Zachary Robertson[1], Akhilan Boopathy[3], Kateryna Pistunova[4], Jason W. Rocks[5], Ila Rani Fiete[6], and Oluwasanmi Koyejo[1]

2023



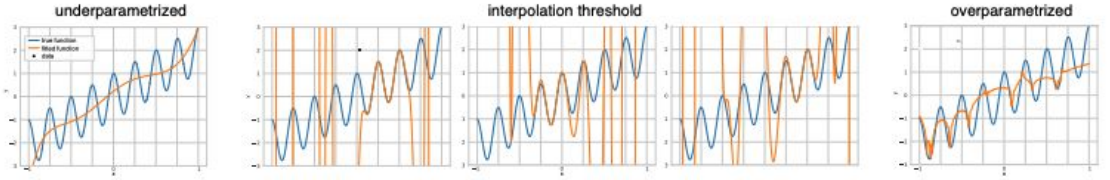
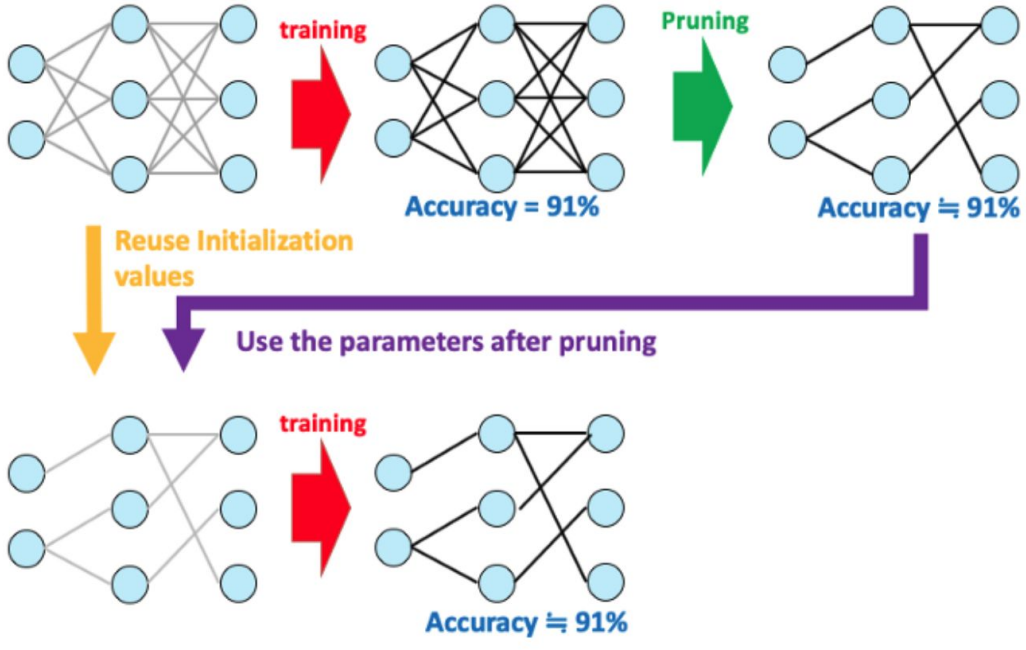


Figure 2: **Intuition for double descent from polynomial regression.** Top: Polynomial regression displays double descent. Bottom: When *underparameterized*, the model is unable to capture finer-grained features in the training data, meaning bias is large but variance is small. As the interpolation threshold is approached, the training data can be fit exactly, meaning bias is small; however, the particular realization of the training data significantly affects the learnt function, meaning variance is large. When *overparameterized*, the model can exactly fit the training data, meaning bias is again small, but the model is also regularized towards a small-norm solution, making variance small.

High dimensional representation

The Lottery Ticket Hypothesis [Frankle & Carbing 2019]

- The pruned sub-network attains at least the same accuracy as the original
- Attains it in at most the same iterations



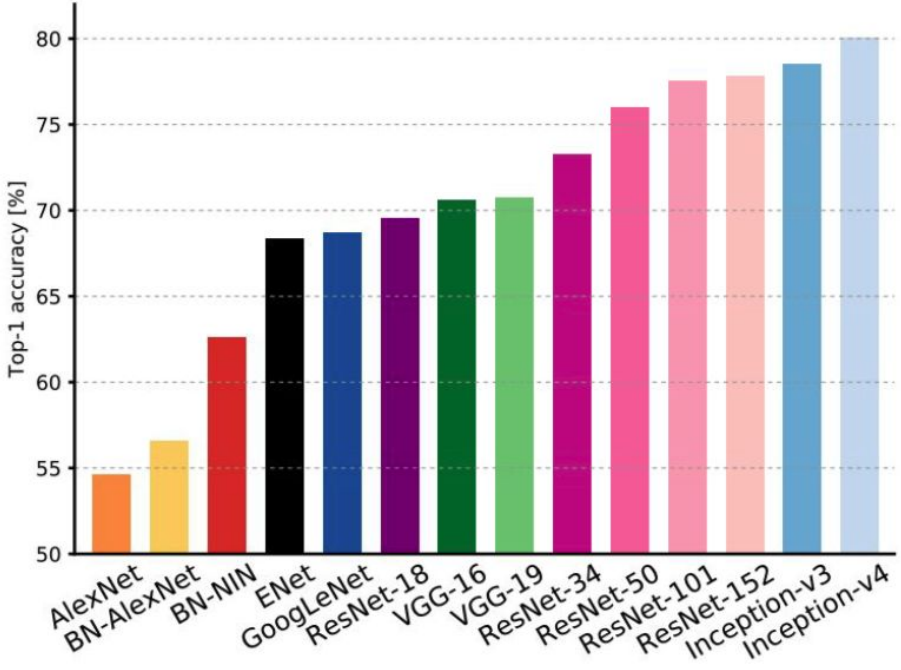
Overview of Lottery Ticket Hypothesis

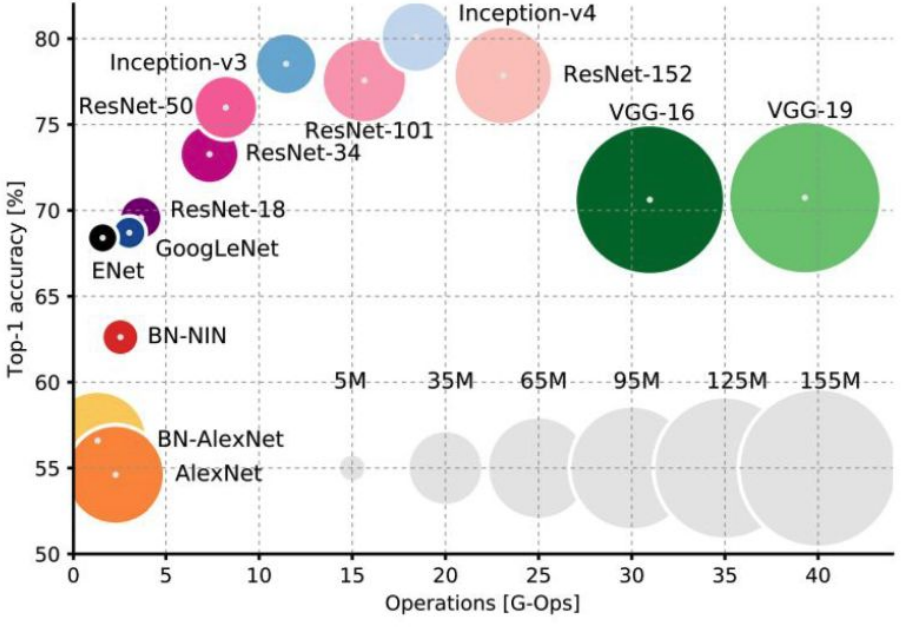
The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks, [Frankle & Carbing ICLR2019]

Classification with CNNs

A classification network

Many architectures, different properties



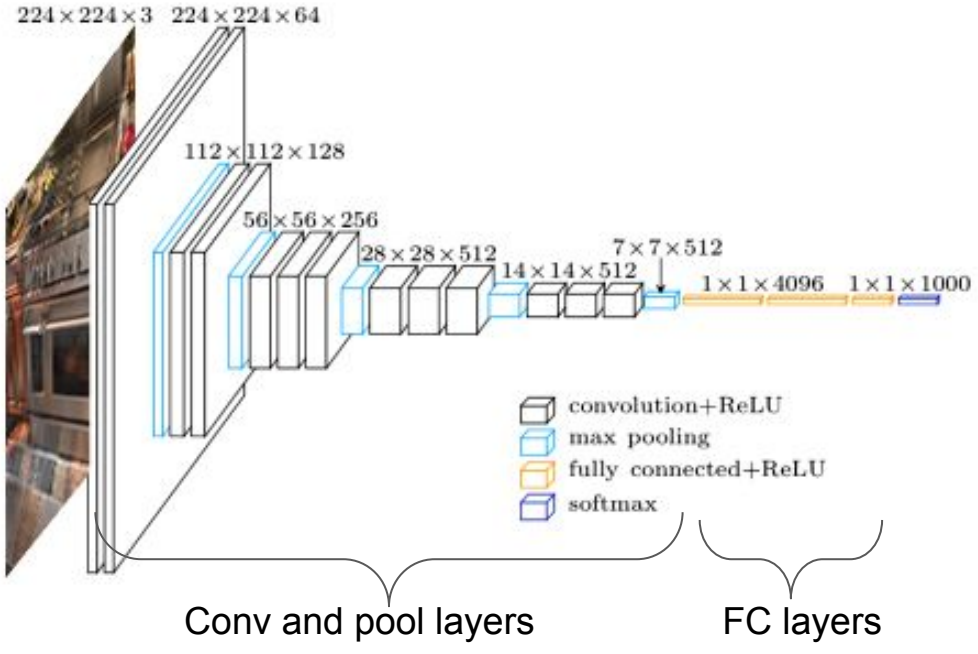


A classification network

VGG [Simonyan, Zisserman. 2014, Very deep convolutional networks for large-scale image recognition.]

- Encoder type architecture
- Final layers produce a vector of probabilities by applying **softmax**

$$P(Y = j \mid X = \mathbf{x}) = \frac{e^{s_j}}{\sum_k e^{s_k}}$$

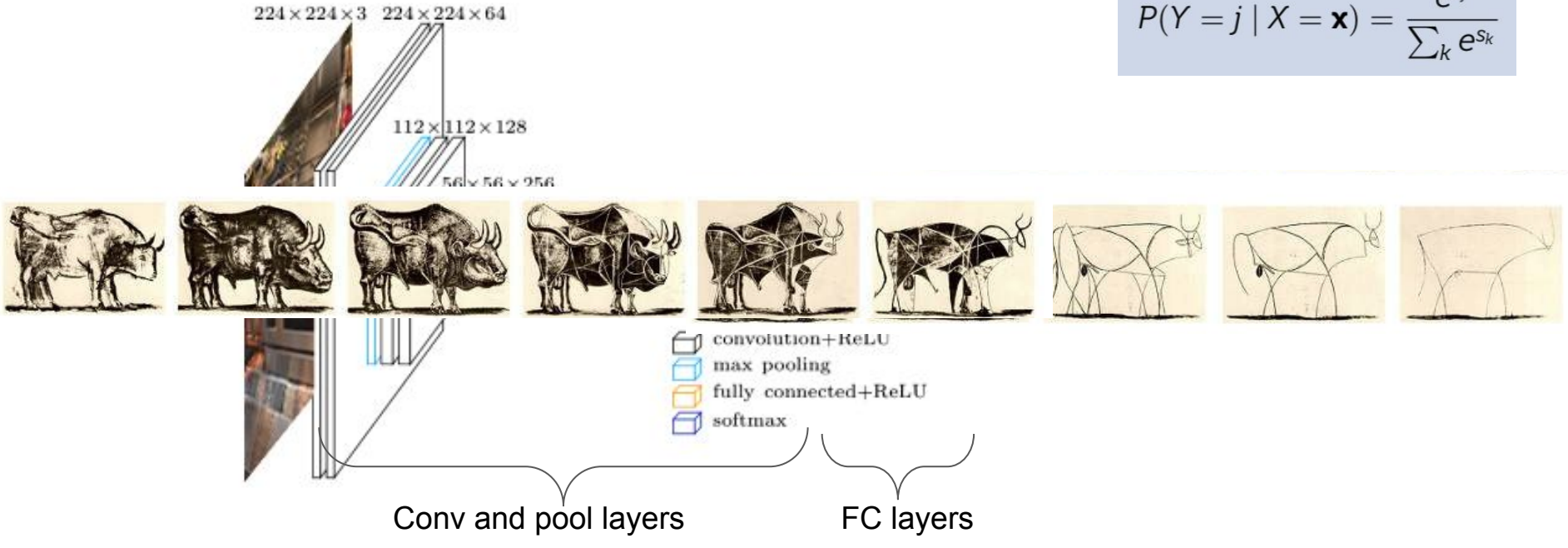


A classification network

VGG [Simonyan, Zisserman. 2014, Very deep convolutional networks for large-scale image recognition.]

- Encoder type architecture
- Final layers produce a vector of probabilities by applying **softmax**

$$P(Y = j \mid X = \mathbf{x}) = \frac{e^{s_j}}{\sum_k e^{s_k}}$$



Classification Loss: Cross-Entropy Loss

The optimizer will minimize the loss over all the training examples

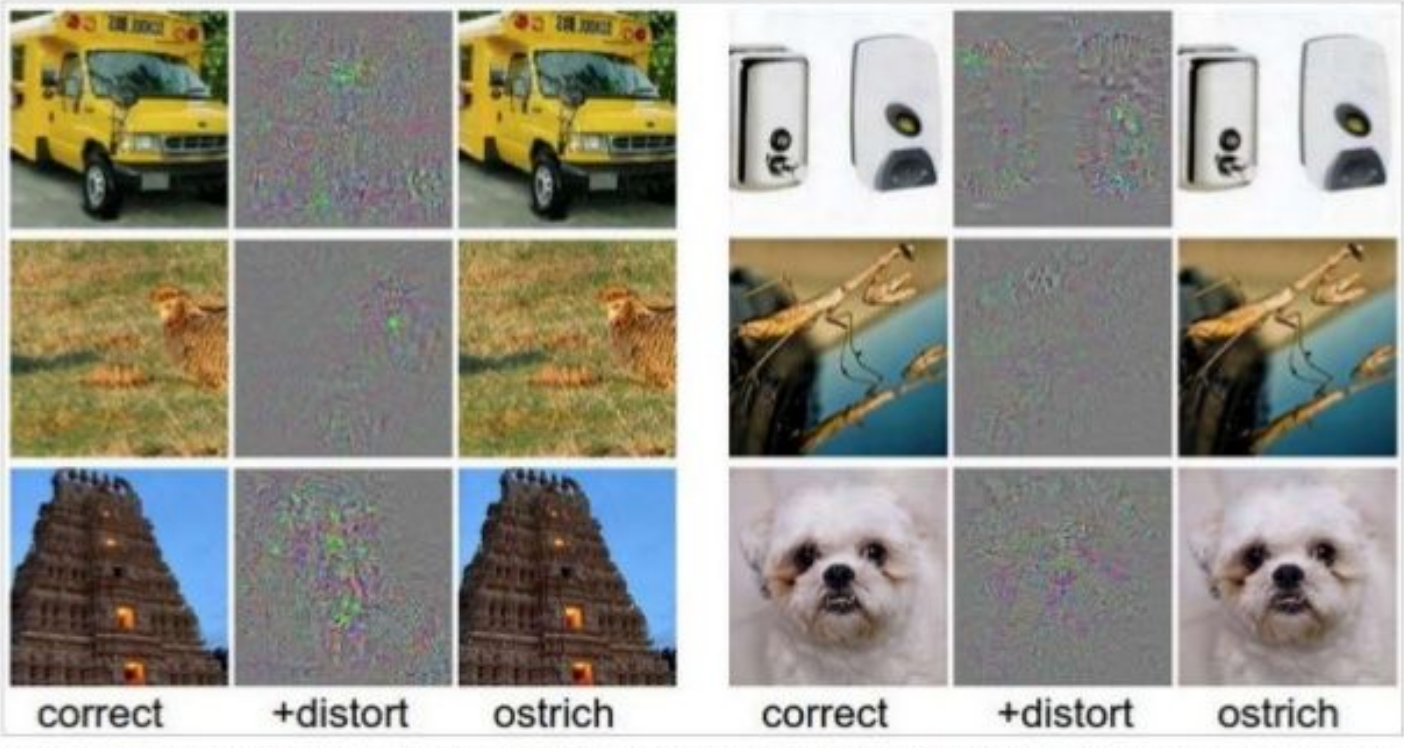
$$\min_{\theta} \sum_{(x_i, y_i) \in \text{examples}} \ell(\text{NET}_\theta(x_i), y_i).$$

The cross-entropy loss is defined as

$$\ell(\text{NET}_\theta(x_i), y_i) = -\sum_{c=1}^{C} \mathbb{1}_{y_i \in C_c} \log \underbrace{p_{model}}_{\text{NET}_\theta(x_i)} [y_i \in C_c].$$

it amounts to maximizing the predicted **probability** for the correct class

Still imperfect solution: adversarial attacks



Take a correctly classified image (left image in both columns), and add a tiny distortion (middle) to fool the ConvNet with the resulting image (right).

Intriguing properties of neural networks [Szegedy ICLR 2014]

Summary

- NNs are powerful end-to-end trainable functions
- Allow to solve complex problems better than hand made programs
- Only require data to be trained
  - Also depend on the choice of model, design of loss, and optimization algorithm
- Still many open questions

Homework single-Gaussian Splatting

Le problème consiste à trouver les paramètres d'une gaussienne a, \mu et \Sigma de sorte à bien approximer le contenu d'une image monochrome I(x). On fait ca par minimisation de la fonction g.

$$\min_{\mu,\Sigma,a} g(I,\mu,\Sigma,a)$$

$$g(I,\mu,\Sigma,a) = \sum_x |f(x,\mu,\Sigma,a) - I(x)|^2$$

I(x) est une image monochrome qu'on fournit

$$f(x,\mu,\Sigma,a) = ae^{-(x-\mu)^T\Sigma(x-\mu)} \quad with \quad \mu, x \in \mathbf{R}^2, \quad \Sigma = \begin{pmatrix} \sigma_1^{-2} & 0 \\ 0 & \sigma_2^{-2} \end{pmatrix}$$

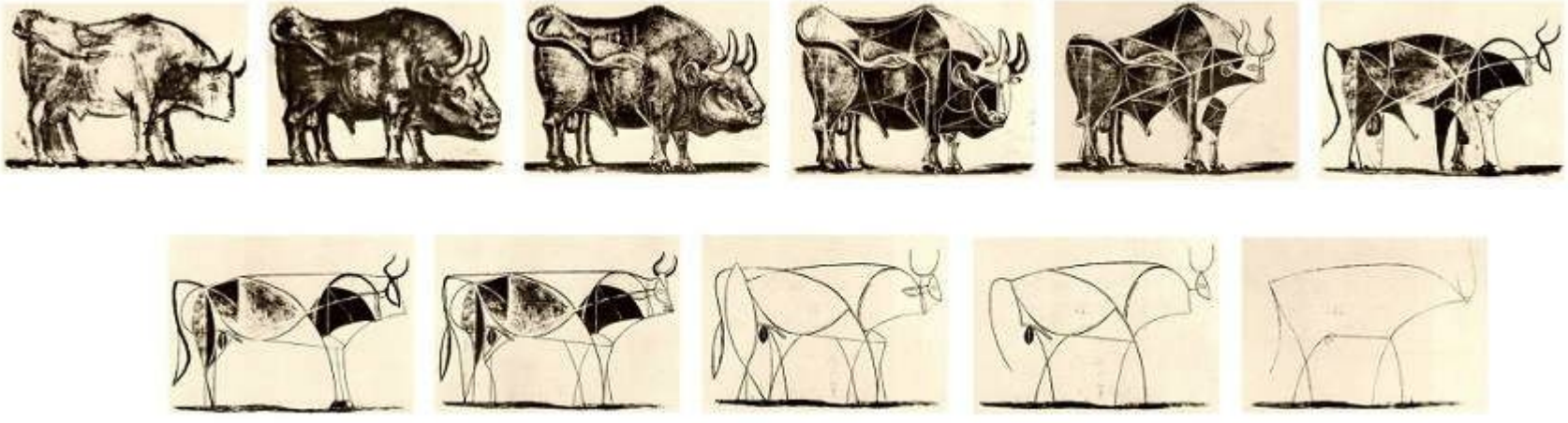
- L'objectif est de décrire le graph de calcul la minimisation de g.
- Calculer par backprop le gradient des parametres a, \mu et \Sigma.
- Puis implémenter l'optimisation sans autodiff de pytorch, et avec autodiff pour obtenir a, \mu et \Sigma.
- Bonus: Comment se comporte si on remplace on \sigma_i par exp(signa_i^{-2})?

Exemple de Gaussian Splatting

https://poly.cam/capture/52b1e099-c2cc-4eca-a0ae-e17aabece1ff

Must have NN ingredients

- ConvNets
- ReLU
- BatchNorm
- Skip connections
- Deeper networks



Rules

- ReLU
- Residual learning
- Batch normalization
- Exploit image redundancies
- Multiscale?

UNet = Multigrid