DEEP LEARNING

An introduction to deep learning

Jeremy Fix

March 16, 2022

Slides made with slidemaker

INTRODUCTION

HISTORICAL PERSPECTIVE

- Hodgkin-Huxley (Hodgkin & Huxley, 1952) : giant squid axon
- Formal neuron (McCulloch & Pitts, 1943) : the community gets very excited
- Perceptron (Rosenblatt, 1958) : linear classifier
- AdaLinE (Widrow & Hoff, 1962) : linear regressor
- Minsky/Papert (Minksy & Papert, 1969) : first winter
- Convolutional Neural networks (Fukushima, 1980),(LeCun et al., 1989) : great !
- Multilayer Perceptron and **backprop** (Rumelhart, Hinton, & Williams, 1986) : great !

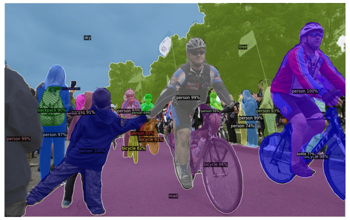
but it is hard to train (except the CNN) and the SVM comes in the play in the 1990s … : second winter

- 2006 : pretraining greatly helps
- 2012 : AlexNet on ImageNet (10% better on test than the 2nd)
- Now on : lot of SOTA neural networks

For an overview : (Schmidhuber, 2015)

SUCCESS STORIES

- Image classification : (Krizhevsky, Sutskever, & Hinton, 2012)
- Image segmentation CSAIL repo, detectron2
- Depth estimation Github
- 3D Pose estimation
- Generative Adversial Networks (e.g. NVlabs stylegan3), see also fakestylegan



- Speech synthesis/recognition (Ze, Senior, & Schuster, 2013), (J. Li et al., 2019)
- Automatic translation (Google Neural Machine Translation)
- Language models (BERT, GPT) (Devlin, Chang, Lee, & Toutanova, 2019)

Figure 1: A second of generated speech.

Wavenet

See also Deep reinforcement learning : Atari / AlphaGO / AlphaStar / AlphaChem; Graph neural networks, etc..

≡

WHY IS DEEP LEARNING WORKING “NOW”

Some of the reasons of the current success :

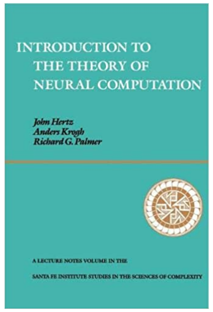
- GPU (speed of processing) / Data (regularizing)
- theoretical understandings on the difficulty of training deep networks (from 2006)

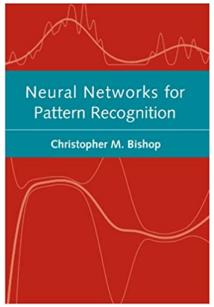
Libraries allow to easily implement/test/deploy neural networks :

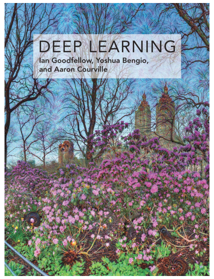
- Torch (Lua) / PyTorch(Python/C++), Caffe(C++/Python), Caffe2 (RIP 2018)
- Microsoft CNTK
- Google Tensorflow / Keras
- Theano/Lasagne (Python, RIP 2017)
- CNTK, Chainer, Matlab, Mathematica, ….

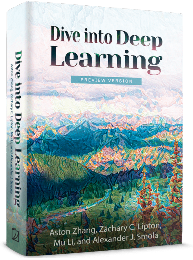
RESSOURCES

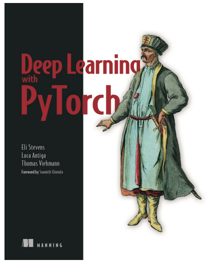
BOOKS


INTRODUCTION TO THE THEORY OF NEURAL COMPUTATION
John Hertz
Anders Krogh
Richard G. Palmer
A LECTURE NOTES VOLUME IN THE SANTA FE INSTITUTE STUDIES IN THE SCIENCES OF COMPLEXITY






Dive into Deep Learning
PREVIEW VERSION
Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola



PEOPLE AND CONFERENCES

Some of the major contributors to the field:

- N-2 : McCulloch/Pitts, Rumelhart, Rosenblatt, Hopfield,
- N-1 : Hinton, Bengio, LeCun, Schmidhuber
- N : Goodfellow, Dauphin, Graves, Sutskever, Karpathy, Krizevsky, Hochreiter

Some the most important conferences: NIPS/NeurIPS, ICLR, (ICML, ICASSP, ..)
Online ressources :
- distill.pub, blog posts (e.g. pytorch.org blog),
- FastAI lectures, CS231n, MIT S191
- awesome deep learning, Awesome deep learning papers

SYLLABUS

Lecture 1/2 (08/02): Introduction, Linear networks, RBF
Lecture 3/4 (10/02): Feedforward networks, differential programming, initialization and gradient descent

2 : 17/02 1TP : 21/02 1TP : 28/02 2 CM : 07/03 2 CM : 14/03 1 TP : 21/03

Lecture 5 (17/02): Regularization, and Convolutional neural networks architectures
Lecture 6 (17/02-07/03) : Convolutional Neural Networks : applications

Lab work 1 (21/02-28/02) : Introduction to pytorch, tensorboard, FCN, CNNs

Lecture 7 (07/03-14/03): Recurrent neural networks : architectures
Lecture 8 (14/03): Recurrent neural networks : applications

Lab work 3 (21/03): Recurrent neural networks : Seq2Seq for Speech to text

Labworks : on our GPU clusters (1080, 2080 Ti, pytorch), in pairs, remotely with VNC.

Exam (22/03): 2h paper and pen exam

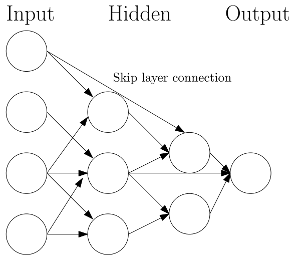
≡

WHAT IS A NEURAL NETWORK ?

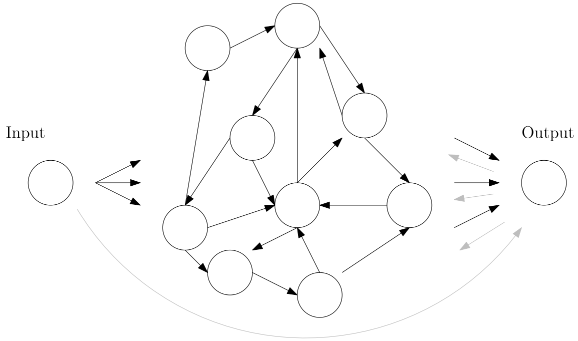
DEFINITION

A neural network is a directed graph :

- nodes : computational units
- edges : weighted connections



Feedforward neural network



Recurrent neural network

There are two types of graphs :

- no cycle : feedforward neural network
- with at least one cycle : recurrent neural networks

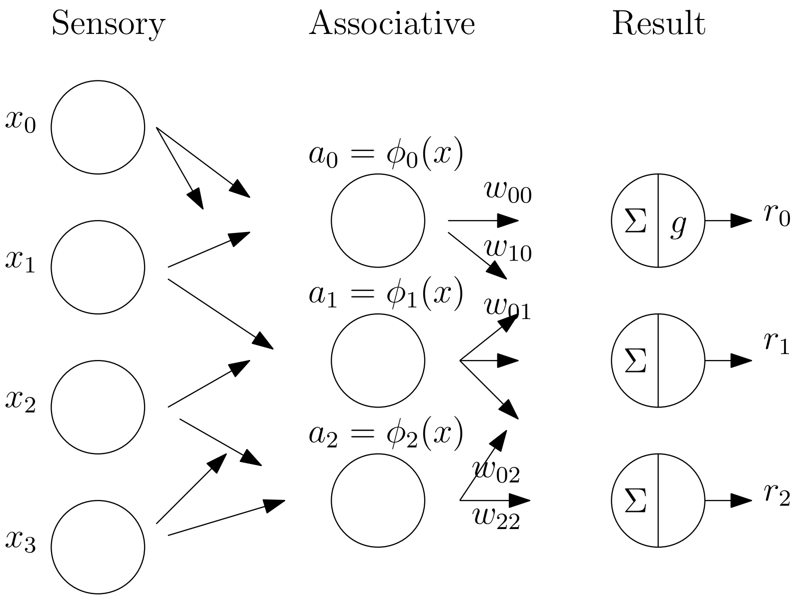
But why do we care about **convolutional neural networks** with a **softmax** output, **ReLu** hiddden activations, **cross entropy** loss, **batch normalization** layers, trained with **RMSprop** with **Nesterov momentum** regularized with **dropout** exactly ?

LINEAR NEURAL NETWORKS

PERCEPTRON (ROSENBLATT, 1958)

PERCEPTRON (ROSENBLATT, 1958)

- Classification : given $(x_i, y_i) \in \mathbb{R}^n \times \{-1, 1\}$
- Sensory - Associative - Response architecture, $\phi_j(x)$ with $\phi_0(x) = 1$
- Algorithm and geometrical interpretation



SAR Architecture

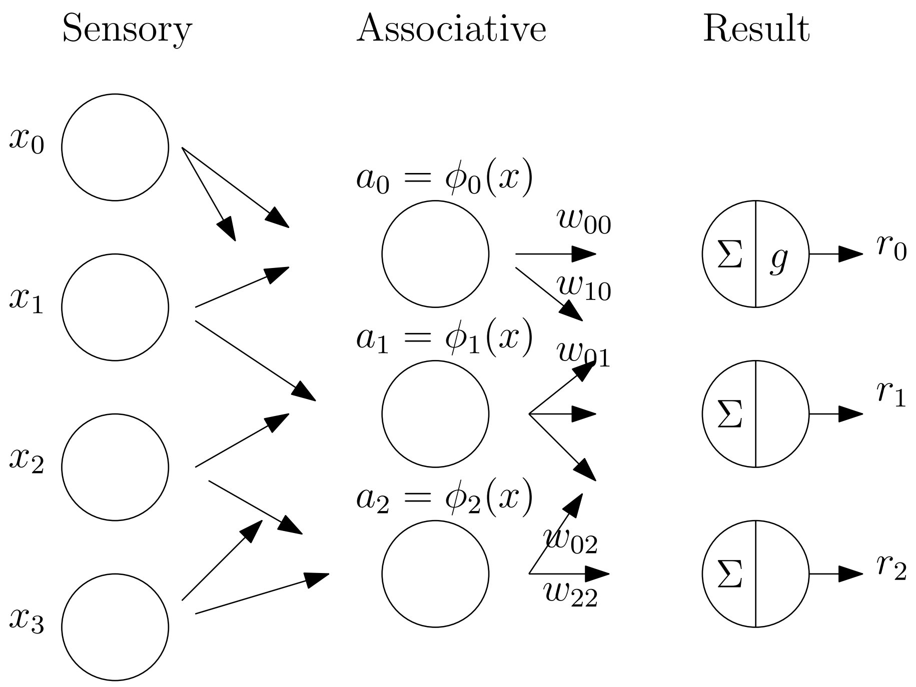
ARCHITECTURE OF THE CLASSIFIER

Given **fixed, predefined** feature functions ϕ_j, with $\phi_0(x) = 1, \forall x \in \mathbb{R}^n$, the perceptron classifies x as :

$$y = g(w^T \Phi(x)) \tag{1}$$

$$g(x) = \begin{cases} -1 & \text{if} \quad x < 0 \\ +1 & \text{if} \quad x \geq 0 \end{cases} \tag{2}$$

with $\phi(x) \in \mathbb{R}^{n_a+1}$, $\phi(x) = \begin{bmatrix} 1 \\ \phi_1(x) \\ \phi_2(x) \\ \vdots \end{bmatrix}$



SAR Architecture

ONLINE TRAINING ALGORITHM

Given $(x_i, y_i), y_i \in \{-1, 1\}$, the **perceptron learning rule** operates online:

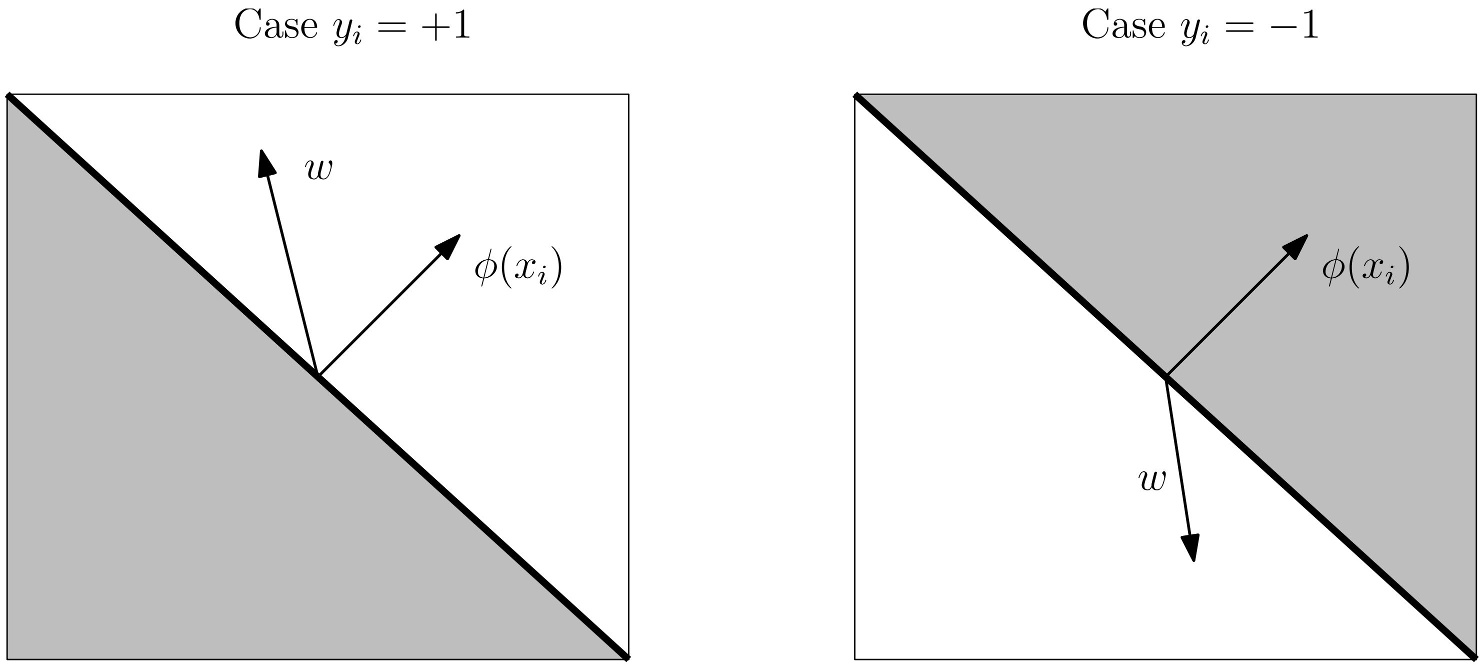
$$w = \begin{cases} w & \text{if the input is correctly classified} \\ w + \phi(x_i) & \text{if the input is incorrectly classified as -1} \\ w - \phi(x_i) & \text{if the input is incorrectly classified as } +1 \end{cases} \tag{3}$$

GEOMETRICAL INTERPRETATION : CORRECT CLASSIFICATION

Decision rule : $y = g(w^T\Phi(x))$

Algorithm:

$$w = \begin{cases} w & \text{if the input is correctly classified} \\ w + \phi(x_i) & \text{if the input is incorrectly classified as -1} \\ w - \phi(x_i) & \text{if the input is incorrectly classified as } +1 \end{cases} \tag{4}$$



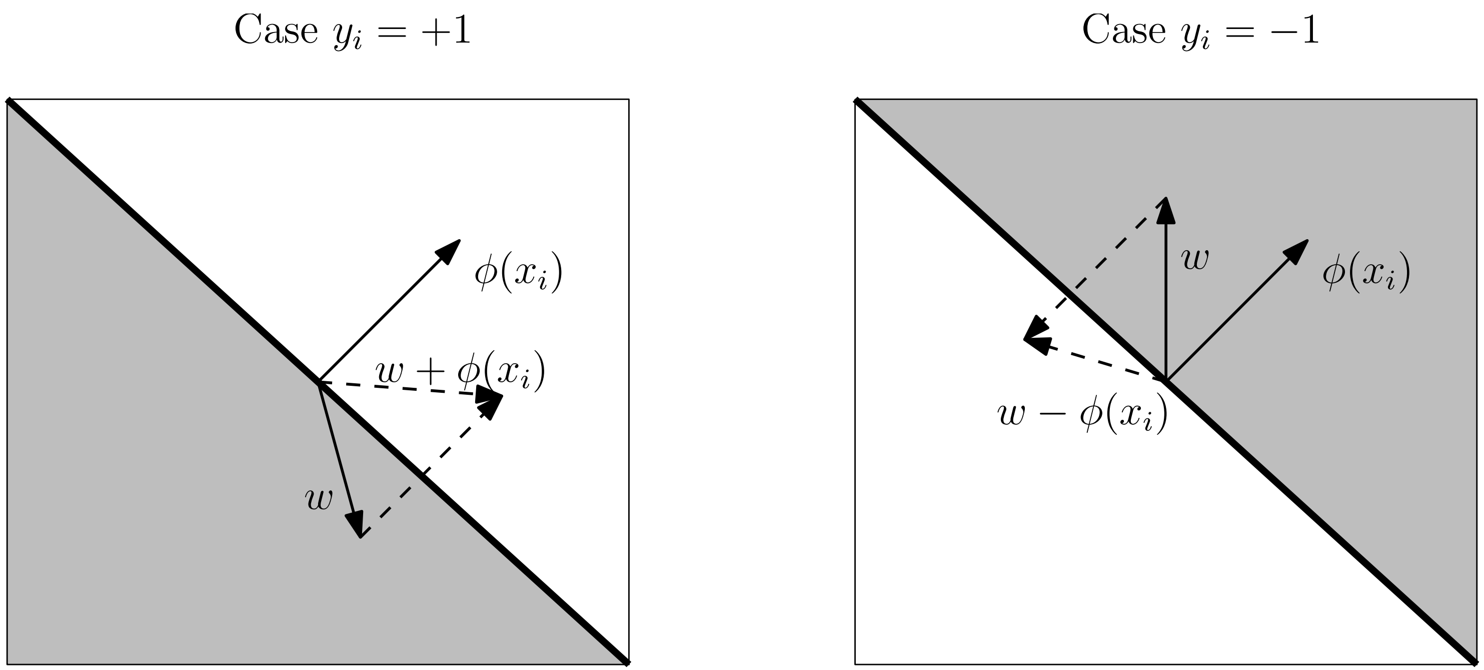
A correctly classified sample either positive or negative

GEOMETRICAL INTERPRETATION : MISCLASSIFICATION

Decision rule : $y = g(w^T \Phi(x))$

Algorithm:

$$w = \begin{cases} w & \text{if the input is correctly classified} \\ w + \phi(x_i) & \text{if the input is incorrectly classified as -1} \\ w - \phi(x_i) & \text{if the input is incorrectly classified as } +1 \end{cases} \tag{5}$$



Case $y_i = +1$

Case $y_i = -1$

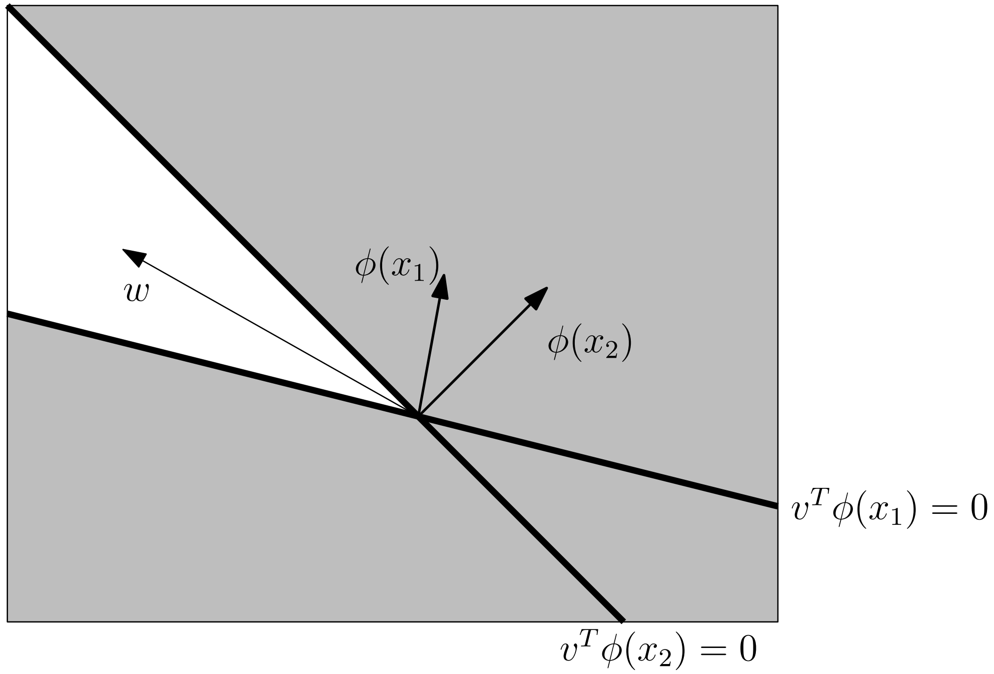
An incorrectly classified sample either positive or negative

GEOMETRICAL INTERPRETATION : MULTIPLE SAMPLES

Decision rule : $y = g(w^T \Phi(x))$

The intersection of the valid halfspaces is called the cone of feasibility (it may be empty).

Consider two samples x_1, x_2 with $y_1 = +1, y_2 = -1$



The cone of feasibility for $y_1 = +1$ and $y_2 = -1$

TOWARD A CANONICAL LEARNING RULE

Given $(x_i, y_i), y_i \in \{-1, 1\}$, the *perceptron learning rule* operates online:

$$w = \begin{cases} w & \text{if the input is correctly classified} \\ w + \phi(x_i) & \text{if the input is incorrectly classified as -1} \\ w - \phi(x_i) & \text{if the input is incorrectly classified as } +1 \end{cases} \tag{6}$$

$$w = \begin{cases} w & \text{if } g(w^T\phi(x_i)) = y_i \\ w + \phi(x_i) & \text{if } g(w^T\phi(x_i)) = -1 \text{ and } y_i = +1 \\ w - \phi(x_i) & \text{if } g(w^T\phi(x_i)) = +1 \text{ and } y_i = -1 \end{cases} \tag{7}$$

$$w = \begin{cases} w & \text{if } g(w^T\phi(x_i)) = y_i \\ w + y_i\phi(x_i) & \text{if } g(w^T\phi(x_i)) \neq y_i \end{cases}$$

TOWARD A CANONICAL LEARNING RULE

Given $(x_i, y_i), y_i \in \{-1, 1\}$, the *perceptron learning rule* operates online:

$$w = \begin{cases} w & \text{if } g(w^T\phi(x_i)) = y_i \\ w + y_i\phi(x_i) & \text{if } g(w^T\phi(x_i)) \neq y_i \end{cases}$$

$$w = w + \frac{1}{2}(y_i - \hat{y}_i)\phi(x_i)$$

with $\hat{y}_i = g(w^T\phi(x_i))$. This is called the **delta rule**.

PERCEPTRON CONVERGENCE THEOREM

Definition (Linear separability)

A binary classification problem $(x_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}, i \in [1..N]$ is said to be linearly separable if there exists $\mathbf{w} \in \mathbb{R}^d$ such that~:

$$\forall i, \text{sign}(\mathbf{w}^T x_i) = y_i$$

with $\forall x < 0, \text{sign}(x) = -1, \forall x \geq 0, \text{sign}(x) = +1.$

Theorem (Perceptron convergence theorem)

A classification problem $(x_i, y_i) \in \mathbb{R}^d \times \{-1, 1\}, i \in [1..N]$ is linearly separable if and only if the perceptron learning rule converges to an optimal solution in a finite number of steps.

$\Leftarrow$: easy; $\Rightarrow$: we upper/lower bound $|w(t)|_2^2$

VARIOUS FACTS

- $w_t = w_0 + \sum_{i \in \mathcal{I}(t)} y_i \phi(x_i)$, with $\mathcal{I}(t)$ the set of misclassified samples
- it minimizes a loss : $J(w) = \frac{1}{N} \sum_i \max(0, -y_i w^T \phi(x_i))$
- the solution can be written as

$$w_t = w_0 + \sum_i \frac{1}{2}(y_i - \hat{y}_i)\phi(x_i) \tag{8}$$

$(y_i - \hat{y}_i)$ is the prediction error

KERNEL PERCEPTRON

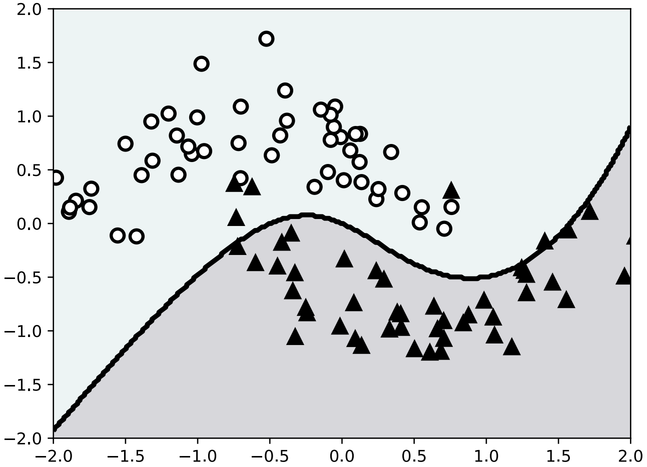
Any linear predictor involving *only scalar products* can be kernelized (**kernel trick**, cf SVM);

Decision rule : $\mathrm{sign}(< w, x >)$

Given $w(t) = w_0 + \sum_{i \in \mathcal{I}} y_i x_i$

$$< w, x > = < w_0, x > + \sum_{i \in \mathcal{I}} y_i < x_i, x >$$

$$\Rightarrow k(w, x) = k(w_0, x) + \sum_{i \in \mathcal{I}} y_i k(x_i, x)$$



Kernel perceptron

Polynomial kernel of degree $d = 3$:

$$k(x, y) = (1 + < x, y >)^3$$

Training set : 50 samples

Real risk : 92%

Code : https://github.com/rougier/ML-Recipes/blob/master/recipes/ANN/kernel-perceptron.py

≡

ADALINE (WIDROW & HOFF, 1962)

LINEAR REGRESSION ANALYTICALLY

Problem : Given (x_i, y_i), $x_i \in \mathbb{R}^{n+1}, y_i \in \mathbb{R}$, minimize

$$J(w) = \frac{1}{N} \sum_i ||y_i - w^T x_i||^2$$

We assume that $x_i[0] = 1 \forall i$ so that $w[0]$ hosts the bias term.

Analytically Introduce $X = [x_0|x_1|\ldots]$, $J(w) = \|y - X^T w\|^2$. In numerator layout (see later)

$$\nabla_w J(w) = 0 \Rightarrow \nabla_z \|z\|_2^2 (z = y - X^T w) \nabla_w (y - X^T w) = -2.(y - X^T w)^T X^T = 0 \Rightarrow XX^T w = Xy$$

- XX^T non singular : $w = (XX^T)^{-1} Xy$
- XX^T singular (e.g. points along a line in 2D), infinite nb solutions
  - One solution can be found with the regularized least square :

$$minG(w) = J(w) + \alpha w^T w$$

  - $\nabla_w G(w) = 0 \Rightarrow (XX^T + \alpha I)w = Xy$
  - as soon as $\alpha > 0$, $(XX^T + \alpha I)$ is not singular

Needs to compute XX^T, i.e. over the whole training set…

≡

LINEAR REGRESSION WITH STOCHASTIC GRADIENT DESCENT

Problem : Given $(x_i, y_i), x_i \in \mathbb{R}^{n+1}, y_i \in \mathbb{R}$, minimize

$$J(w) = \frac{1}{N} \sum_i ||y_i - w^T x_i||^2$$

We assume that $x_i[0] = 1 \forall i$ so that $w[0]$ hosts the bias term.

- start at w_0
- take each sample one after the other (online) x_i, y_i
- denote $\hat{y}_i = w^T x_i$ the prediction
- update

$$w_{t+1} = w_t - \epsilon \nabla_w J(w_t) = w_t + \epsilon (y_i - \hat{y}_i) x_i$$

- delta rule, $\delta = (y_i - \hat{y}_i)$ prediction error

$$w_{t+1} = w_t + \epsilon \delta x_i$$

GRADIENT DESCENT

BATCH GRADIENT DESCENT

$$J(w, x, y) = \frac{1}{N} \sum_{i=1}^{N} L(w, x_i, y_i)$$

e.g. $L(w, x_i, y_i) = ||y_i - w^T x_i||^2$

Batch gradient descent

- compute the gradient of the loss $J(w)$ over the whole training set
- performs one step in direction of $-\nabla_w J(w, x, y)$

$$w_{t+1} = w_t - \epsilon_t \nabla_w J(w, x, y)$$

- ϵ : learning rate

≡

STOCHASTIC GRADIENT DESCENT

$$J(w, x, y) = \frac{1}{N} \sum_{i=1}^{N} L(w, x_i, y_i)$$

e.g. $L(w, x_i, y_i) = ||y_i - w^T x_i||^2$

Stochastic gradient descent (SGD)

- one sample at a time, noisy estimate of $\nabla_w J$
- performs one step in direction of $-\nabla_w L(w, x_i, y_i)$

$$w_{t+1} = w_t - \epsilon_t \nabla_w L(w, x_i, y_i)$$

- faster to converge than gradient descent

MINIBATCH GRADIENT DESCENT

$$J(w, x, y) = \frac{1}{N} \sum_{i=1}^{N} L(w, x_i, y_i)$$

e.g. $L(w, x_i, y_i) = ||y_i - w^T x_i||^2$

Minibatch

- noisy estimate of the true gradient with M samples (e.g. $M = 64, 128$); M is the minibatch size
- Randomize $\mathcal{J}$ with $|\mathcal{J}| = M$, one set at a time

$$w_{t+1} = w_t - \epsilon_t \frac{1}{M} \sum_{j \in \mathcal{J}} \nabla_w L(w, x_j, y_j)$$

- smoother estimate than SGD
- great for parallel architectures (GPU)

If the batch size is too large, there is a generalization gap (LeCun, Bottou, Orr, & Müller, 1998), maybe due to sharp minimum (Keskar, Mudigere, Nocedal, Smelyanskiy, & Tang, 2017); see also (Hoffer, Hubara, & Soudry, 2017)

DOES IT MAKE SENSE TO USE GRADIENT DESCENT ?

Convex function A function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is convex :

1. $\iff \forall x_1, x_2 \in \mathbb{R}^n, \forall t \in [0,1]\ f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$
2. with f twice diff.,
   $\iff \forall x \in \mathbb{R}^n, H = \nabla^2 f(x)$ is positive semidefinite
   i.e. $\forall x \in \mathbb{R}^n, x^T H x \geq 0$

For a convex function f, all local minima are global minima. Our losses are lower bounded, so these minima exist. Under mild conditions, gradient descent and stochastic gradient descent converge, typically $\sum \epsilon_t = \infty, \sum \epsilon_t^2 < \infty$ (cf lectures on convex optimization).

LINEAR REGRESSION

SUMMARY

Problem : Given $(x_i, y_i), x_i \in \mathbb{R}^{n+1}, y_i \in \mathbb{R}$

- We assume that $x[0] = 1$ to encompass the bias
- **Linear model** : $\hat{y} = w^T x$
- **L2 loss** : $L(\hat{y}, y) = \|\hat{y} - y\|^2$
- by **gradient descent**

$$\nabla_w L(w, x_i, y_i) = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w} = -(y_i - \hat{y}_i) x_i$$

Other choices may also be considered (Huber loss, MAE, …).

Possibly regularized (but more on regularization latter).

LINEAR REGRESSION WITH L2 LOSS IS CONVEX

Indeed,

- Given $x_i, y_i, L(w) = \frac{1}{2}(w^T x_i - y_i)^2$ is convex:

$$\nabla_w L = (w^T x_i - y_i) x_i$$
$$\nabla_w^2 L = x_i x_i^T$$
$$\forall x \in \mathbb{R}^n x^T x_i x_i^T x = (x_i^T x)^2 \geq 0$$

- a non negative weighted sum of convex functions is convex

LINEAR CLASSIFICATION

MAXIMUM LIKELIHOOD (BINARY CLASSIFICATION)

Problem : Given $(x_i, y_i), x_i \in \mathbb{R}^n, y_i \in \{0,1\}$

Assume that $P(y=1|x) = p(x;w)$, parametrized by w, and our samples to be independent, the conditional likelihood of the labels is:

$$\mathcal{L}(w) = \prod_i P(y=y_i|x_i) = \prod_i p(x_i;w)^{y_i}(1-p(x_i;w))^{1-y_i}$$

With maximum likelihood estimation, we rather equivalently minimize the averaged negative log-likelihood :

$$J(w) = -\frac{1}{N}\log(\mathcal{L}(w)) = \frac{1}{N}\sum_i -y_i \log(p(x_i;w)) - (1-y_i)\log(1-p(x_i;w))$$

≡

LOGISTIC REGRESSION (BINARY CLASSIFICATION)

Problem : Given $(x_i, y_i), x_i \in \mathbb{R}^{n+1}, y_i \in \{0, 1\}$

- **Linear logit model** : $o(x) = w^T x$ (we still assume $x[0] = 1$ for the bias)
- **Sigmoid** transfer function : $\hat{y}(x) = p(x; w) = \sigma(o(x)) = \sigma(w^T x)$
  - $\sigma(x) = \frac{1}{1+\exp(-x)}, \sigma(x) \in [0, 1]$
  - $\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$
- Following maximum likelihood estimation, we minimize :

$$J(w) = \frac{1}{N} \sum_i -y_i \log(p(x_i; w)) - (1 - y_i) \log(1 - p(x_i; w))$$

- The loss $L(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$ is called the **cross entropy loss**, or **negative log-likelihood**
- The gradient of the cross entropy loss with $\hat{y}(x) = \sigma(x)$ is :

$$\nabla_w L(w, x_i, y_i) = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w} = -(y_i - \hat{y}_i) x_i$$

≡

LOGISTIC REGRESSION IS CONVEX

Indeed,

- Given $x_i, y_i = 1, L_1(w) = -\log(\sigma(w^T x_i) = \log(1 + \exp(-w^T x_i)),$
  $\nabla_w L_1 = -(1 - \sigma(w^T x_i))x_i$
  $\nabla_w^2 L_1 = \underbrace{\sigma(w^T x_i)(1 - \sigma(w^T x_i))}_{>0} x_i x_i^T$
- Given $x_i, y_i = 0, L_2(w) = -\log(1 - \sigma(w^T x_i))$
  $\nabla_w L_2 = \sigma(w^T x_i)x$
  $\nabla_w^2 L_2 = \underbrace{\sigma(w^T x_i)(1 - \sigma(w^T x_i))}_{>0} x_i x_i^T$
- a non negative weighted sum of convex functions is convex

≡

DO NOT USE A L2 LOSS

Compute the gradient to see why

Take L2 loss $L(\hat{y}, y) = \frac{1}{2}||\hat{y} - y||^2$

- Take the “linear” model : $\hat{y}_i = \sigma(w^T x_i)$
- Check that $\frac{d}{dx}\sigma(x) = \sigma(x)(1 - \sigma(x))$
- Compute the gradient wrt w:

$$\nabla_w L(w, x_i, y_i) = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w} = (\hat{y}_i - y_i)\sigma(w^T x_i)(1 - \sigma(w^T x_i))x_i$$

- If x_i is strongly misclassified (e.g. $y_i = 1, w^T x_i = -\infty$). Then $\sigma(w^T x_i)(1 - \sigma(w^T x_i)) \approx 0$, i.e. $\nabla_w L(w, x_i, y_i) \approx 0 \Rightarrow$ stepsize is very small while the sample is misclassified

With a cross entropy loss, $\nabla_w L(w, x_i, y_i)$ is proportional to the error

SOFTMAX REGRESSION (MULTICLASS CLASSIFICATION)

Problem : Given $(x_i, y_i), x_i \in \mathbb{R}^{n+1}, y_i \in [|0, K-1|]$

Assume that $P(y=c|x) = \frac{e^{w_c^T x}}{\sum_k e^{w_k^T x}}$, parametrized by $w_0, w_1, w_2, ..$, and our samples to be independent, the conditional likelihood of the labels is:

$$\mathcal{L}(w) = \prod_i P(y = y_i|x_i)$$

With maximum likelihood estimation, we rather equivalently minimize the averaged negative log-likelihood:

$$J(w) = -\frac{1}{N}\log(\mathcal{L}(w)) = -\frac{1}{N}\sum_i \log(P(y = y_i|x_i))$$

With a one-hot encoding of the target class (i.e. $y_i = [0, \ldots, 0, 1, 0, ..]$), it can be written as :

$$J(w) = -\frac{1}{N}\log(\mathcal{L}(w)) = -\frac{1}{N}\sum_i \sum_c y_c \log(P(y = c|x_i))$$

SOFTMAX REGRESSION (MULTICLASS CLASSIFICATION)

Problem : Given $(x_i, y_i), x_i \in \mathbb{R}^{n+1}, y_i \in [|0, K-1|]$

- Linear models for each class $o_j(x) = w_j^T x$ (we still assume $x[0] = 1$)
- Softmax transfer function : $P[y = j|x] = \hat{y}_j = \frac{\exp(o_j(x))}{\sum_k \exp(o_k(x))}$
- Generalization of the sigmoid for a vectorial output
- Following maximum likelihood estimation, we minimize

$$J(w) = -\frac{1}{N}\log(\mathcal{L}(w)) = -\frac{1}{N}\sum_i \log(P(y = y_i|x_i))$$

- The loss $L(\hat{y}, y) = -\log(\hat{y}_y)$ is called the cross-entropy loss
- by **gradient descent**:

$$\nabla_{w_j} L(w, x, y) = \sum_k \frac{\partial L}{\partial \hat{y}_k}\frac{\partial \hat{y}_k}{\partial w_j} = -(\delta_{j,y} - \hat{y}_j)x$$

Softmax regression is convex.

NUMERICAL ISSUES WITH THE SOFTMAX AND CE LOSS

Large exponentials

If you compute naïvely the softmax, you would have $\exp(..)$ which is quickly large.

Fortunately:

$$softmax(o_1, o_2, o_3, ..) = softmax(o_1 - o^\star, o_2 - o^\star, o_3 - o^\star) = \frac{\exp(o_i - o^\star)}{\sum_j \exp(o_j - o^\star)}$$

You always compute $\exp(z)$ with $z \leq 0$.

Avoiding some exponentials with the log-sum-exp trick

$\log(\sum_j \exp(o_j)) = o^\star + \log(\sum_j \exp(o_j - o^\star))$

You do not really need to compute the $\log(\hat{y}_j) = \log(softmax_j(x)))$ since :

$$\log(\hat{y}_i) = \log(\frac{\exp(o_i - o^\star)}{\sum_j \exp(o_j - o^\star)}) = o_i - o^\star - \log(\sum_j \exp(o_j - o^\star))$$

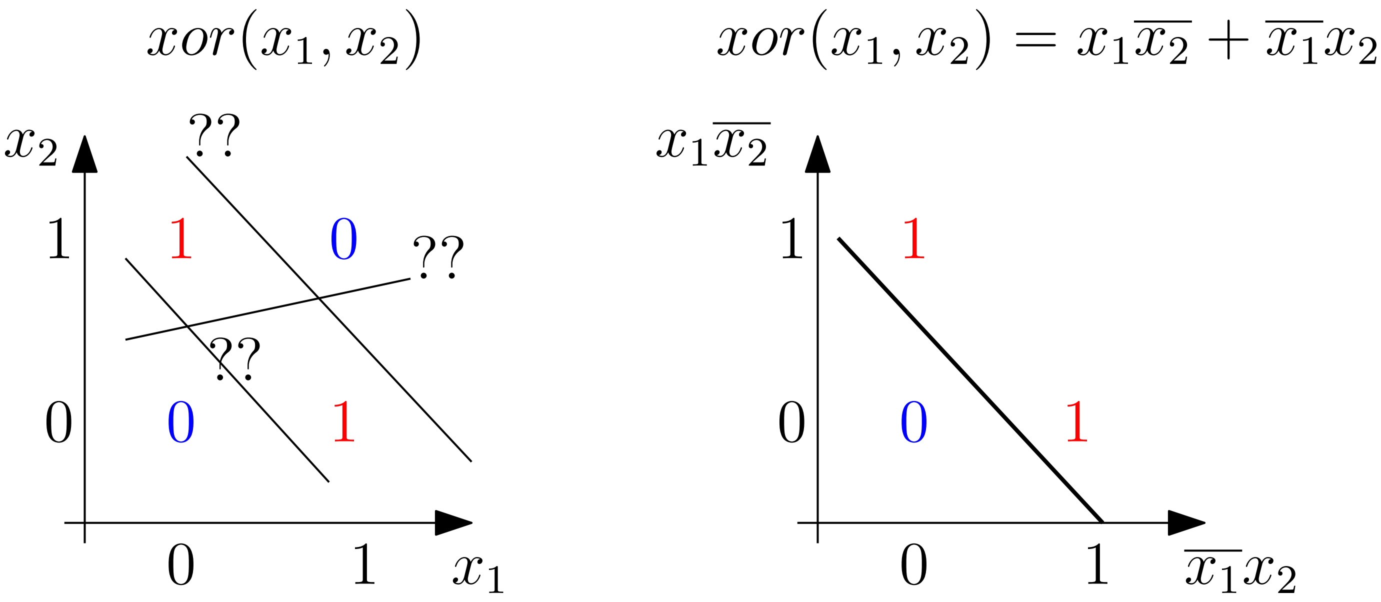
In practice that explains why we use the Cross entropy loss with logits outputs rather than Softmax + Negative log likelihood or even LogSoftMax + NLLLoss (which does not have the log… yeah confusing…)

≡

TOWARD NON LINEAR NETWORKS

LIMITS OF LINEAR CLASSIFICATION

Perceptrons and logistic regression perform linear separation in a **predefined**, **fixed** feature space.



The XOR and its transformation

What about learning these features $\phi_j(x)$?

RADIAL BASIS FUNCTION NETWORKS (RBFN) (BROOMHEAD & LOWE, 1988)

ARCHITECTURE (BROOMHEAD & LOWE, 1988)

- RBFN are **prototype based** function approximator.
- specific architecture with a single layer of learnable feature vectors with “weights” (parameters) $(\mu_j, \sigma_j)_{j \in [0..N_a-1]}$

$$\phi(x) = \begin{pmatrix} 1 \\ \exp \frac{-||x-\mu_0||^2}{2\sigma_0^2} \\ \vdots \\ \exp \frac{-||x-\mu_{N_a-1}||^2}{2\sigma_{N_a-1}^2} \end{pmatrix}$$

Regression

- identity transfer function $y = w^T \phi(x)$
- L2 loss $L(y, \hat{y}) = \|\hat{y} - y\|^2$

Binary classification

- sigmoidal transfer function $y = \sigma(w^T \phi(x))$
- CE loss

$$\begin{aligned} L(y, \hat{y}) = \quad & -y \log(\hat{y}) \\ & -(1-y) \log(1-\hat{y}) \end{aligned}$$

Multi classification

- softmax transfer function (see Lecture 1)
- CE loss (see Lecture 1)

LEARNING

- We know how to learn the weights w : minibatch gradient descent (or a variant thereof)
- What about the centers and variances ? (Schwenker, Kestler, & Palm, 2001)
  - place them uniformly, randomly, by vector quantization (K-means++(Arthur & Vassilvitskii, 2007), GNG (Fritzke, 1994))
  - two phases : fix the centers/variances, fit the weights
  - three phases : fix the centers/variances, fit the weights, fit everything ($\nabla_\mu L, \nabla_\sigma L, \nabla_w L$)

≡

UNIVERSAL APPROXIMATOR

Theorem : Universal approximation (Park & Sandberg, 1991)

Denote $\mathcal{S}$ the family of functions based on RBF in $\mathbb{R}^d$:

$$\mathcal{S} = \{g \in \mathbb{R}^d \to \mathbb{R}, g(x) = \sum_i w_i K(\frac{x - \mu_i}{\sigma}), w \in \mathbb{R}^N\}$$

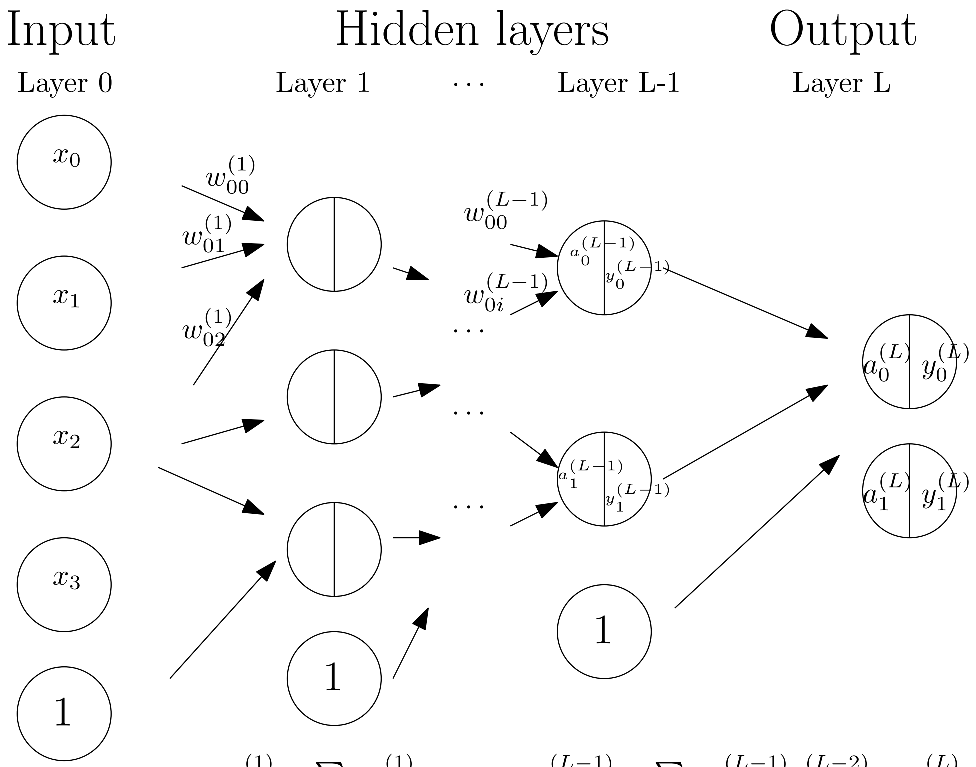
with $K : \mathbb{R}^d \to \mathbb{R}$ continuous almost everywhere and $\int_{\mathbb{R}^d} K(x)dx \neq 0$,
Then $\mathcal{S}$ is dense in $L^p(\mathbb{R})$ for every $p \in [1, \infty)$

In particular, it applies to the gaussian kernel introduced before.

EXAMPLE

FEEDFORWARD NEURAL NETWORKS (FNN)

ARCHITECTURE

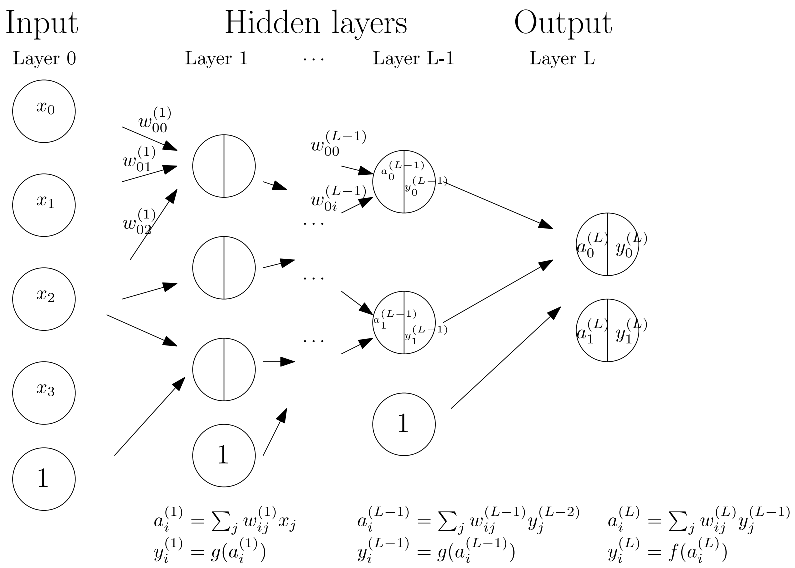


$$a_i^{(1)} = \sum_j w_{ij}^{(1)} x_j \qquad a_i^{(L-1)} = \sum_j w_{ij}^{(L-1)} y_j^{(L-2)} \qquad a_i^{(L)} = \sum_j w_{ij}^{(L)} y_j^{(L-1)}$$

$$y_i^{(1)} = g(a_i^{(1)}) \qquad y_i^{(L-1)} = g(a_i^{(L-1)}) \qquad y_i^{(L)} = f(a_i^{(L)})$$

A feedforward neural network

ARCHITECTURE



A feedforward neural network

Vocabulary

- Depth : number of weight layers
- Width : number of units per layer
- Parameters : Weights and biases for every unit
- Skip layer connections can bypass layers
- one hidden transfer function f, one task-specific output transfer function g

HIDDEN TRANSFER FUNCTION

- historically: hyperbolic tangent $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$ or sigmoid $\sigma(x) = \frac{1}{1+\exp(-x)}$
- now mainly Recitifed Linear Units (**ReLu**)(Nair & Hinton, 2010),(Krizhevsky et al., 2012) or variants :

$$\mathrm{relu}(x) = \max(x, 0)$$

ReLu are more favorable for the **gradient flow** than the saturating functions (more on that latter when discussing computational graphs and gradient computation).

≡

SOME OTHER RECENT HIDDEN TRANSFER FUNCTIONS

Relu (Nair & Hinton, 2010)

$$f(x)=\begin{cases} x & \text{if } x\geq 0 \\ 0 & \text{if } x<0 \end{cases}$$

Leaky Relu
(Maas, Hannun, & Ng, 2013)
Parametric ReLu
(He, Zhang, Ren, & Sun, 2015)

$$f(x)=\begin{cases} x & \text{if } x\geq 0 \\ \alpha x & \text{if } x<0 \end{cases}$$

Exponential Linear Unit
(Clevert, Unterthiner, & Hochreiter, 2016)

$$f(x)=\begin{cases} x & \text{if } x\geq 0 \\ \alpha(\exp(x)-1) & \text{if } x<0 \end{cases}$$

OUTPUT TRANSFER FUNCTION

Exactly as when we discussed about the RBF, this is task dependent.

Regression

- identity transfer function $y = w^T \phi(x)$
- L2 loss $L(y, \hat{y}) = \|\hat{y} - y\|^2$

Binary classification

- sigmoidal transfer function $y = \sigma(w^T \phi(x))$
- CE loss

$$\begin{aligned} L(y, \hat{y}) = \quad & -y \log(\hat{y}) \\ & -(1 - y) \log(1 - \hat{y}) \end{aligned}$$

Multi classification

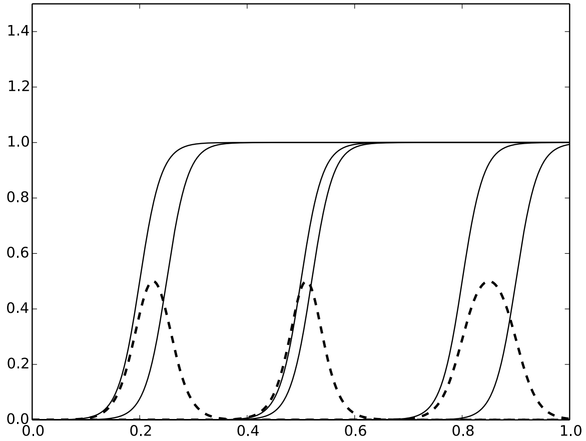
- softmax transfer function (see Lecture 1)
- CE loss (see Lecture 1)

UNIVERSAL APPROXIMATION

*Any well behaved function can be **arbitrarily** approximated with a **single layer** FNN* (Cybenko, 1989), (Hornik, 1991)

Intuition

- Transform the input with a linear transform $y = w^T x$
- Take a sigmoid transfer function $z = f(y) = \frac{1}{1+e^{-y}}$: this is the output of the hidden layer
- combine multiple activities in the $z-$layer to build up gaussian like kernels



Substracting $z-$ layer activities to produce RBF kernels

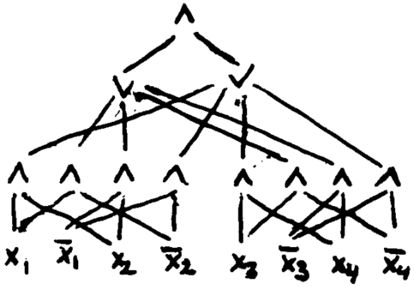
- weight such substractions and you are back to the RBF universal approximation theorem

At that point, you may wonder why we bother about deep learning, right ?

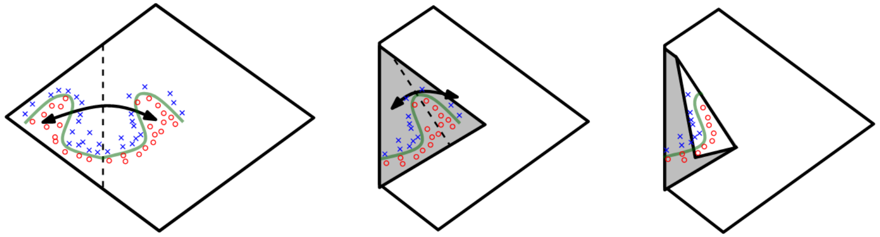
≡

WHY DO WE BOTHER ABOUT DEEP LEARNING ?

- Single hidden layer FFN are universal approximators but the hidden layer can be arbitrarily large
- a **deep network** (large number of layers) builds high level features by **composing/factoring** lower level features which can be reused by multiple units. Image analogy :
  - first layer : extract oriented contours, texture filters, ..
  - second layer : learn corners, crosses, curves, by combining contours
  - next layers : build up more and more complex features
- a **shallow network** must learn all the possibly complex filters at once, no real way to compose
- early theoretical results on logic gates circuits (Hastad, 1986). More recent works on ReLU FFN (Montufar, Pascanu, Cho, & Bengio, 2014)


x_1 $\bar{x}_1$ x_2 $\bar{x}_2$ x_3 $\bar{x}_3$ x_4 $\bar{x}_4$

A logical circuit as of studied in (Hastad, 1986)



Space folding as discussed in (Montufar et al., 2014)

TRAINING : ERROR BACKPROPAGATION

Training is performed by **gradient descent** which was popularized by (Rumelhart et al., 1986) who called it **error backpropagation** (but (Werbos, 1981) already introduced the idea, see (Schmidhuber, 2015)).

Gradient descent is an **iterative algorithm** :

- initialize the weights and biases : w_0
- at every iteration compute :

$$w \leftarrow w - \epsilon \nabla_w J$$

Remember : by minibatch gradient descent (see Lecture 1)

The question is : how do you compute $\frac{\partial J}{\partial w_i}$??

But let us first see pytorch in action.

≡

EXAMPLE ON A REGRESSION PROBLEM

Overall steps :

Training

0- Imports
1- Loading the data
2- Define the network
3- Define the loss, optimizer, callbacks, …
4- Iterate and monitor

Testing

0- Imports
1- Loading the data
2- Define the network and load the trained parameters
3- Define the loss
4- Iterate

EXAMPLE ON A REGRESSION PROBLEM

0- Imports

```
import torch
import torch.nn as nn
import torch.optim as optim

import sklearn
import sklearn.datasets
```

1- Loading the data

```
# Load the data and build up our dataloader
data = sklearn.datasets.fetch_california_housing()
# X is (20640, 8), y is (20640, )
X, y = data.data, data.target

# At least normalize the input for an easier optimization
mean, std = X.mean(axis=0), X.std(axis=0)
```

Doc: Dataset, DataLoader. Pin memory

Iterating over train_dataloader gives a pair of tensors of shape $(64, 8)$ and $(64,)$.

EXAMPLE ON A REGRESSION PROBLEM

2- Define the network

```python
if torch.cuda.is_available():
    device = torch.device('gpu')
else:
    device = torch.device('cpu')

# Build up the model
Nh = 64
```

Doc: Linear, Sequential

EXAMPLE ON A REGRESSION PROBLEM

3- Define the loss, optimizer, callbacks, …

```
# Define the loss
loss = nn.MSELoss()

# Define the gradient descent algorithm
optimizer = optim.Adam(model.parameters(),
                       lr=0.001)
scheduler = optim.lr_scheduler.StepLR(optimizer,
```

Doc: MSELoss, Adam, StepLR

EXAMPLE ON A REGRESSION PROBLEM

4- Iterate and monitor

```
for e in range(num_epochs):
    # Switch the network in train mode
    model.train()

    print(f"Epoch {e}")
    for X, y in tqdm.tqdm(train_dataloader):
        # Send the data to the GPU if necessary
```

Evaluation

```
def mseloss(loader):
    # After every epoch, compute the risk
    # on the loader
    cum_loss = 0.0
    n_samples = 0

    # Switch the network in eval mode
```

COMPUTATIONAL GRAPH AND DIFFERENTIAL PROGRAMMING

COMPUTATIONAL GRAPH

A computational graph is a directed acyclic graph where nodes are :

- variables (weights, inputs, outputs, targets, $\ldots$)
- operations (ReLu, Softmax, $w^T x + b$, losses, updates, ..)

Example graph for a linear regression $\mathbb{R}^8 \mapsto \mathbb{R}$ with minibatch (X, y)

$$J = \frac{1}{M} \sum_{i=0}^{63} (w_1^T x_i + b_1 - y_i)^2$$

COMPUTATIONAL GRAPH

Problem computing the partial derivatives with respect to the variables $\frac{\partial J}{\partial var}$.

You just need to provide the **local derivatives** of the output w.r.t the inputs.

And then apply the **chain rule**.

ex : $\frac{\partial J}{\partial w_1} \in \mathcal{M}_{1,8}(\mathbb{R})$, assuming numerator layout

COMPUTATIONAL GRAPH : THE CHAIN RULE

Numerator layout convention (otherwise, we transpose and reverse the jacobian product order):

The derivative of a scalar with respect to a vector is a row vector :

$$y \in \mathbb{R}, x \in \mathbb{R}^n, \frac{dy}{dx} \in \mathcal{M}_{1,n}(\mathbb{R})$$

More generally, the derivative of a vector valued function $y : \mathbb{R}^{n_x} \mapsto \mathbb{R}^{n_y}$ with respect to its input (the Jacobian) is a $n_y \times n_x$ matrix :

$$x \in \mathbb{R}^{n_x}, y(x) \in \mathbb{R}^{n_y}, \frac{dy}{dx}(x) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \dots & \frac{\partial y_1}{\partial x_{n_x}} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \dots & \frac{\partial y_2}{\partial x_{n_x}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_{n_y}}{\partial x_1} & \frac{\partial y_{n_y}}{\partial x_2} & \dots & \frac{\partial y_{n_y}}{\partial x_{n_x}} \end{bmatrix}(x)$$

COMPUTATIONAL GRAPH : THE CHAIN RULE

For a (**single-path**) chain $y_1 \rightarrow y_2 = f_1(y_1) \rightarrow y_3 = f_2(y_2) \cdots y_n = f_{n-1}(y_{n-1})$, of vector valued functions $y_1 \in \mathbb{R}^{n_1}, y_2 \in \mathbb{R}^{n_2}, \cdots y_n \in \mathbb{R}^{n_n}$,

$$\frac{\partial y_n}{\partial y_1} = \frac{\partial y_n}{\partial y_{n-1}} \frac{\partial y_{n-1}}{\partial y_{n-2}} \cdots \frac{\partial y_2}{\partial y_1}$$

ex : $\frac{\partial J}{\partial w_1} = \frac{\partial J}{\partial y_1} \frac{\partial y_1}{\partial w_1} = \frac{2}{M}(y_1 - y)^T X \in \mathbb{R}^8$, assuming numerator layout

For matrix variables, we should be introducing tensors. See also this and this

THE CHAIN RULE : MULTIPLE PATHS

For multiple paths, **in principle** we sum over all the paths :

$$\frac{\partial y}{\partial x} = \sum_{j=3,4} \frac{\partial y}{\partial y_i} \frac{\partial y_i}{\partial x} = y_4 \frac{\partial y_3}{\partial x} + y_3 \frac{\partial y_4}{\partial x} = y_4 f'(y_1) w_1^T + y_3 f'(y_2) w_2^T \in \mathbb{R}^8$$

THE CHAIN RULE : MULTIPLES PATHS

But this can be computationally (too) expensive :

- there can be many paths you need to identify and sum over : L layers, N units, N^L paths
- and you must repeat the process for every variable w.r.t. which you want to differentiate
- some computations can be factored (e.g. $\frac{\partial y_2}{\partial x}, \frac{\partial y_1}{\partial x}$)

≡

AUTOMATIC DIFFERENTIATION : FORWARD MODE

Let us be more efficient : **forward mode differentiation**

Idea: To compute $\frac{\partial y}{\partial x}$, forward propagate $\frac{\partial}{\partial x}$
e.g. $\frac{\partial y}{\partial x} = y_3 e^{y_1} \left[w_2^T + y_2 w_1^T\right] + y_4 e^{y_2} \left[w_1^T + y_1 w_2^T\right]$

Welcome to the field of **automatic differentiation** (AD). For more, see (Griewank, 2012), (Griewank & Walther, 2008) (see also (Olah, 2015), (Paszke et al., 2017))

≡

AUTOMATIC DIFFERENTIATION : REVERSE MODE

Let us be (sometimes) even more efficient : **reverse mode differentiation**

Idea: To compute $\frac{\partial y}{\partial x}$, backward propagate $\frac{\partial y}{\partial}$ (compute the adjoint)

e.g. $\frac{\partial y}{\partial x} = (y_4 y_1 e^{y_2} + y_3 e^{y_1}) w_2^T + (y_3 y_2 e^{y_1} + y_4 e^{y_2}) w_1^T$

Oh ! We also got $\frac{\partial y}{\partial w_2} = \frac{\partial y}{\partial y_2} \frac{\partial y_2}{\partial w_2}, \frac{\partial y}{\partial b_2} = \frac{\partial y}{\partial y_2} \frac{\partial y_2}{\partial b_2}, \dots$

This is more efficient than forward mode when we have much more inputs (n) than outputs (m) for $f : \mathbb{R}^n \to \mathbb{R}^m$, computing $\frac{df}{dx}(x)$

A Review of Automatic Differentiationand its Efficient Implementation

≡

GRADIENT ERROR BACKPROPAGATION

In (Rumelhart et al., 1986), the algorithm was called “error backpropagation” : why ?

Suppose a 2-layer multi-layer feedforward network and propagating one sample, with a scalar loss :

$$L = g(y_i, \begin{bmatrix} \\ W_2(n_2 \times n_1) \\ \\ \end{bmatrix} f(\begin{bmatrix} \\ W_1(n_1 \times n_x) \\ \\ \end{bmatrix} \begin{bmatrix} \\ x_i \\ \\ \end{bmatrix})) \in \mathbb{R}$$

g could be a squared loss for regression (with $n_2 = 1$), or CrossEntropyLoss (with logits and $n_2 = n_{class}$) for multiclass classification.

We denote $z_1 = W_1 x_i, z_2 = W_2 f(z_1)$ and $\delta_i = \frac{\partial L}{\partial z_i} \in \mathbb{R}^{n_i}$. Then :

$$\delta_2 = \frac{\partial L}{\partial z_2} = \frac{\partial g(x_1, x_2)}{\partial x_2}(y_i, z_2) \tag{9}$$

$$\delta_1 = \frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial z_1} = \begin{bmatrix} \quad \delta_2 \quad \end{bmatrix} \begin{bmatrix} \\ W_2(n_2 \times n_1) \\ \\ \end{bmatrix} \mathrm{diag}(f'(z_1)) \tag{10}$$

The errors of δ_2 are integrated back through the weight matrix that was used for the forward pass. (See also (Nielsen, 2015), chap 2).

GRADIENT DESCENT IN DEEP LEARNING

Training in two phase

- Evaluation of the outputs : forward propagation / forward pass
- Evaluation of the gradient : reverse-mode differentiation / backward pass

⚠ The reverse-mode differentiation uses the variables computed in the forward pass

$\longrightarrow$ we can apply efficiently stochastic gradient descent to optimize the parameters of our neural networks !

Note the computational graph can be extended to encompass the operations of the backward pass.

GRADIENT DESCENT IN PRACTICE

- The deep learning frameworks all compute the backward pass automatically.

```
optimizer = optim.Adam(model.parameters())

for e in range(epochs):
    for X,y in train_dataloader:
        optimizer.zero_grad()
        ...
        loss.backward()
        optimizer.step()
```

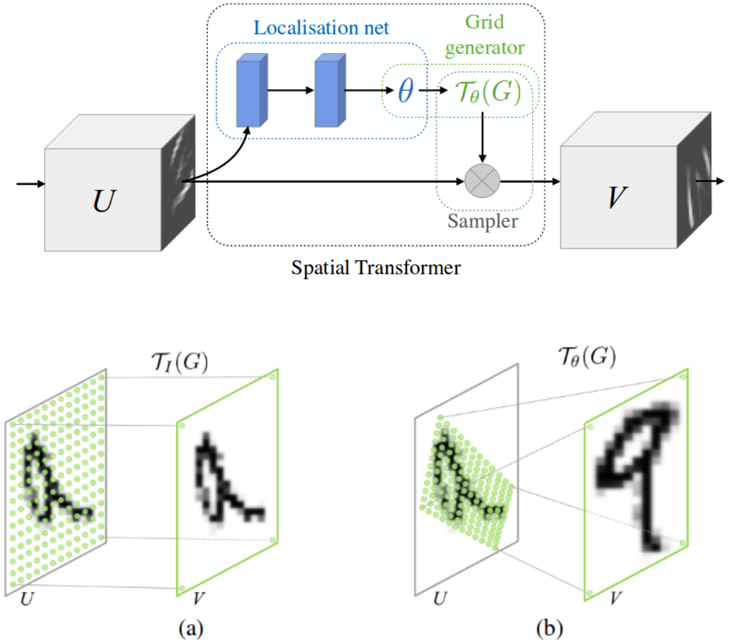
- The computational graphs can be built dynamically (eager mode) or static
- If you want/need to extend the frameworks with new operations, e.g. extending pytorch autograd

```

import torch.autograd.Function as Function

class MyFunction(Function):

    @staticmethod
    def forward(ctx, input, ..):
        ...
```

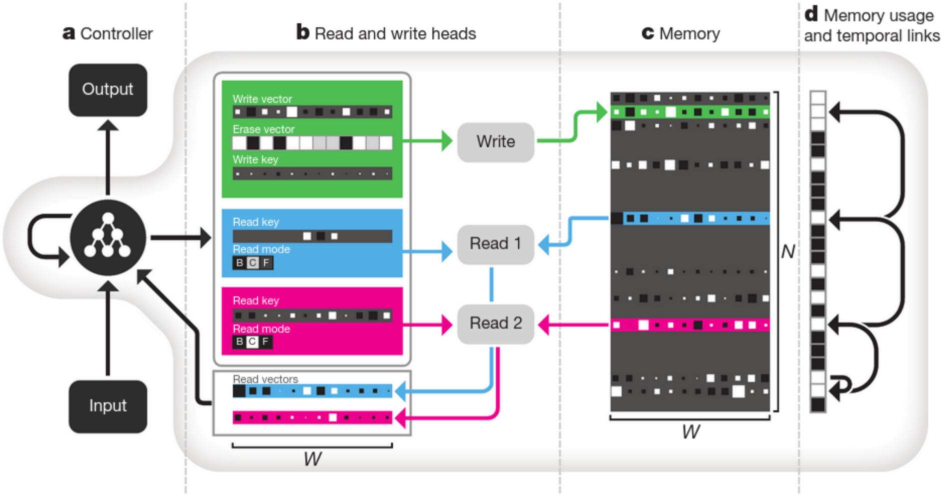
THE WAY TOWARD DIFFERENTIABLE PROGRAMMING

The **computational graph** is a central notion in modern neural networks/deep learning. Broaden the scope with **differential programming**.

In the recent years, “fancier” differentiable blocks others than $f(W f(W..))$, and that are dynamically built (eager mode vs static graph).



Spatial Transformer Networks
(Jaderberg, Simonyan, & Zisserman, 2015)



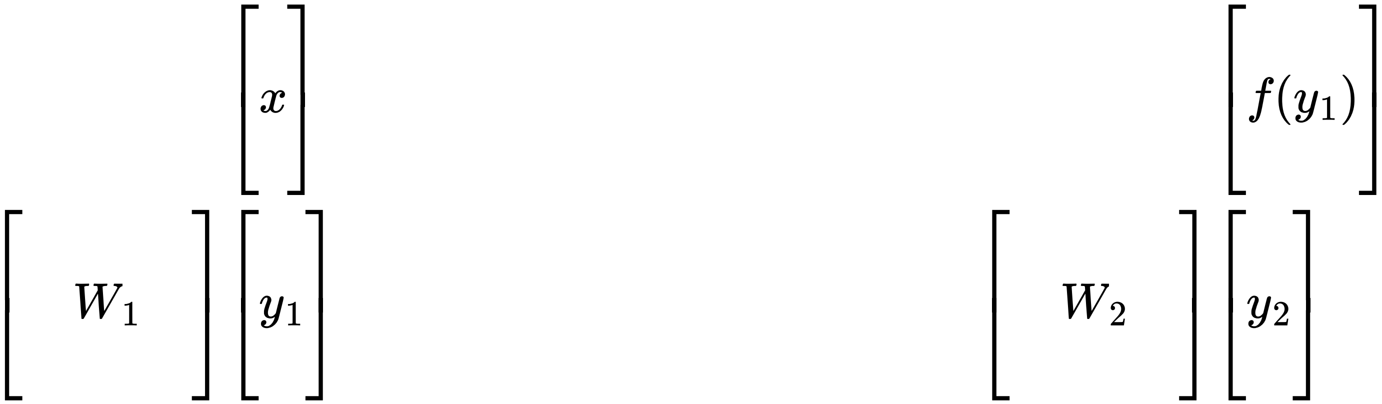
Content/Location based addressing
Neural Turing Machine / Differential Neural computer
(Graves et al., 2016)

GRADIENT DESCENT ALGORITHMS

DOES IT MAKE SENSE TO USE GRADIENT DESCENT ?

Indeed :

- we cannot do better than a **local minima**
- neural networks lead to **non convex optimization**. For example, consider a 2-layer FFN :



But empirically, most local minima are close (in performance) to the global minimum, especially with **large/deep** networks. See (Dauphin et al., 2014), (Pascanu, Dauphin, Ganguli, & Bengio, 2014), (Choromanska, Henaff, Mathieu, Arous, & LeCun, 2015). Saddle points seem to be more critical.

FIRST ORDER METHODS : MINIBATCH STOCHASTIC GRADIENT DESCENT

Algorithm

- Start at θ_0
- for every minibatch :

$$\theta(t+1) = \theta(t) - \epsilon \nabla_\theta L(\theta(t))$$
$$L(\theta) = \frac{1}{M} \sum_i J(\theta, x_i, y_i)$$

Rationale (Taylor expansion) : $L(\theta_{t+1}) \approx L(\theta_t) + (\theta_{t+1} - \theta_t)^T \nabla_\theta L(\theta_t)$

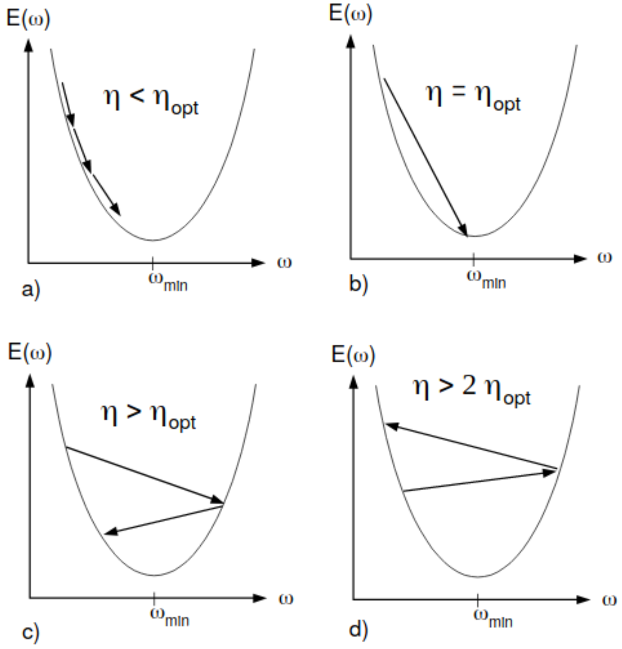
The choice of the **batch size** :

- Stochastic gradient descent (small minibatch, $M = 1$) : noisy estimate, not GPU friendly
- Batch Gradient descent ($M = N$) : More GPU friendly. But more prone to bad generalization (*generalization gap*) and to local minima (Keskar et al., 2017). Roughly speaking, we should avoid sharp minima.

The optimization may converge **slowly** or even **diverge** if the learning rate ϵ is not appropriate.

≡

CHOOSING A LEARNING RATE



The impact of the learning rate on the optimization (LeCun et al., 1998)

Bengio: “The optimal learning rate is usually close to the largest learning rate that does not cause divergence of the training criterion” (Bengio, 2012)

Karpathy “0.0003 is the best learning rate for Adam, hands down.” (Twitter, 2016)

(Note: Adam will be discussed in few slides)

See also :

- Practical Recommendations for gradient-based training of deep architectures (Bengio, 2012)
- Efficient Backprop (LeCun et al., 1998)

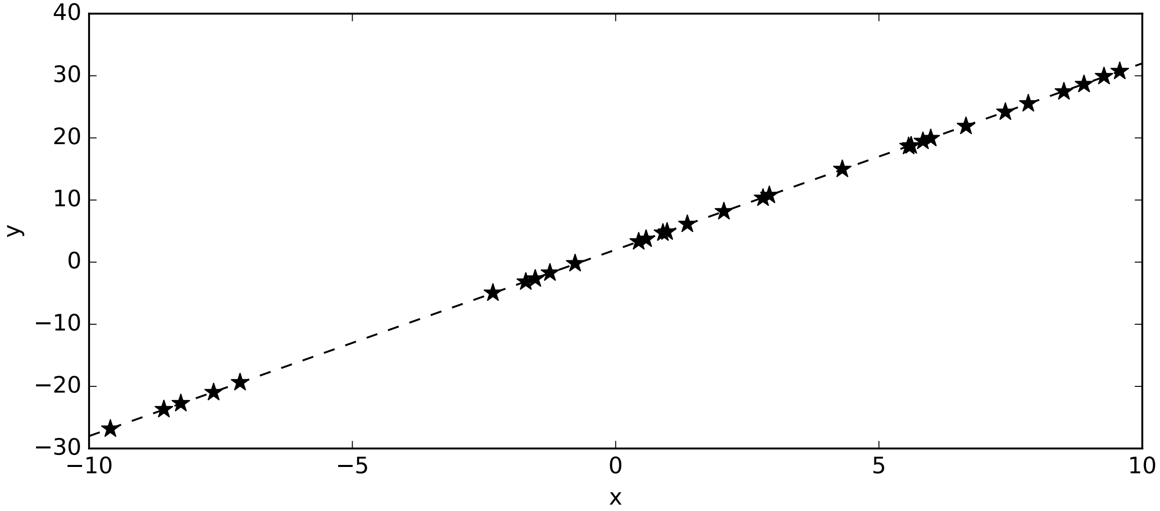
EXAMPLE REGRESSION PROBLEM

Setup

- $N = 30$ samples generated with :

$$y = 3x + 2 + \mathcal{U}(-0.1, 0.1)$$

- Model : $f_\theta(x) = \theta^T \begin{bmatrix} 1 \\ x \end{bmatrix}$,
- L2 loss : $L(y_i, f_\theta(x_i)) = (y_i - f_\theta(x_i))^2$

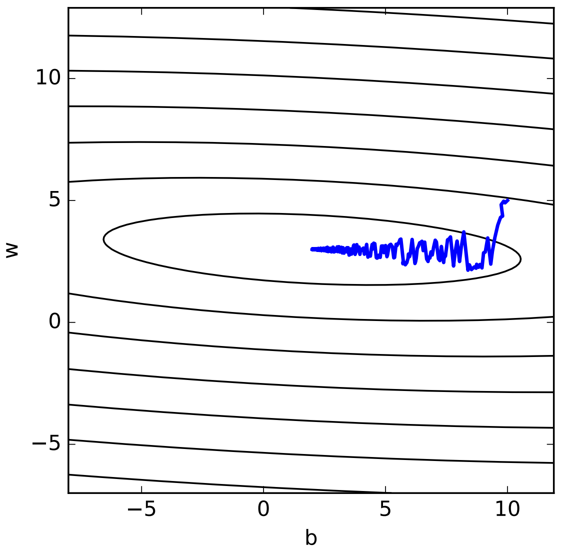


Our simple dataset

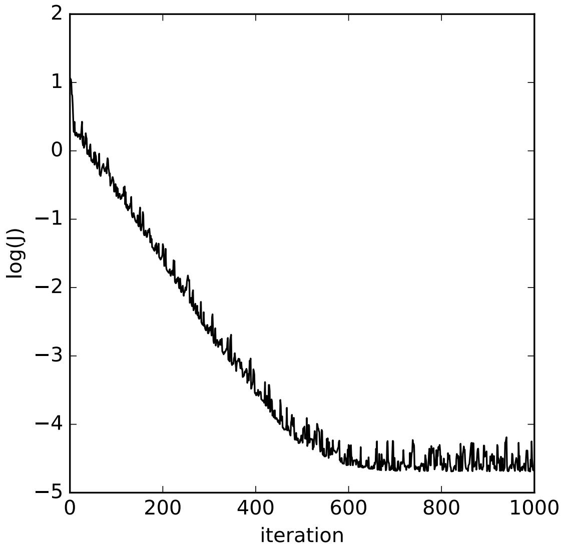
EXAMPLE USING SGD

Parameters : $\epsilon = 0.005, \theta_0 = \begin{bmatrix} 10 \\ 5 \end{bmatrix}$

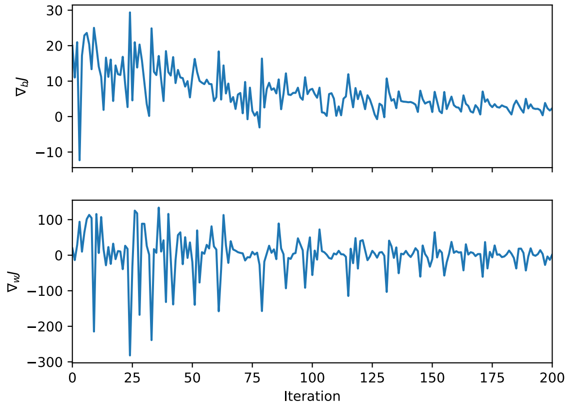
Converges to $\theta_\infty = \begin{bmatrix} 1.9882 \\ 2.9975 \end{bmatrix}$



The optimization path



The value of the function



The components of $\nabla_\theta J$

FIRST ORDER METHODS : MOMENTUM

Algorithm : Let us damp the oscillations with a low pass on ∇_θ

- Start at $\theta_0, v_0 = 0$
- for every minibatch :

$$v(t+1) = \mu v(t) - \epsilon \nabla_\theta J(\theta(t))$$
$$\theta(t+1) = \theta(t) + v(t+1)$$

Usually $\mu \approx 0.9$ or 0.99.

- as an exponential moving average, it low pass filters and therefore dampen oscillations along fast varying dimensions
- it can accelerate (increase the learning rate) in constant directions (or low curvature).
  If $\nabla_\theta J = g, v(0) = 0$

$$v(t) = -\epsilon g \sum_{i=0}^{t-1} \mu^i = -\epsilon g \frac{1-\mu^t}{1-\mu}$$

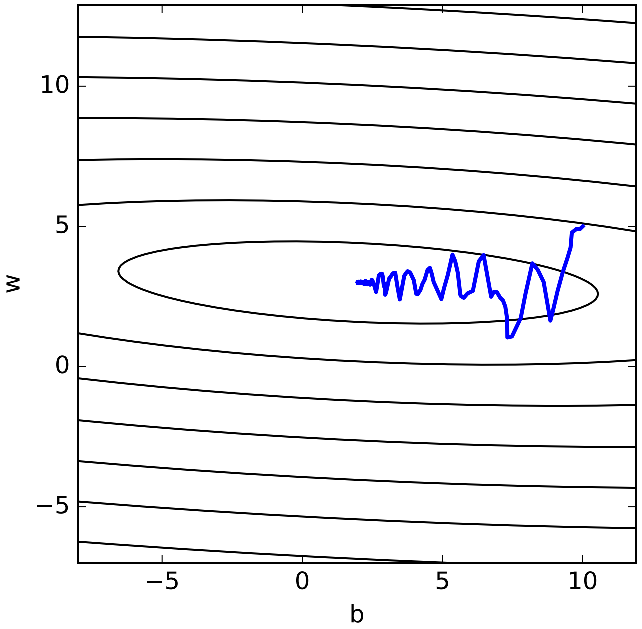
See also distill.pub. Note the frameworks may implement subtle variations.

≡

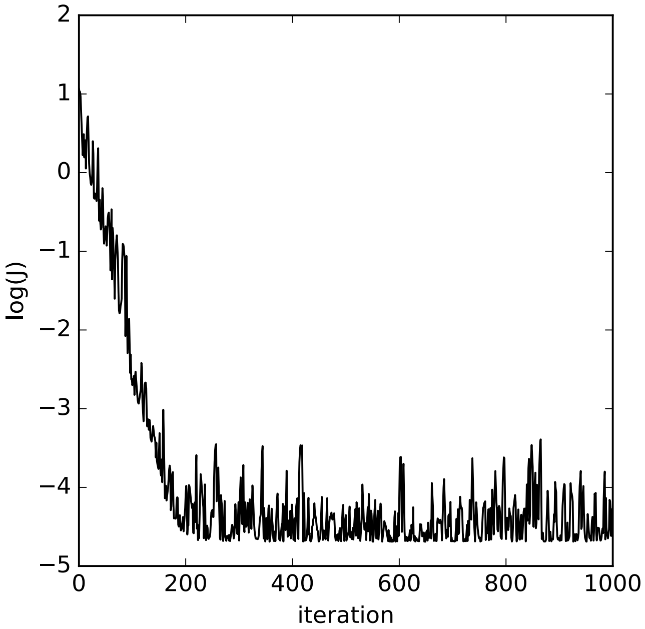
EXAMPLE USING SGD WITH MOMENTUM

Parameters : $\epsilon = 0.005, \mu = 0.6, \theta_0 = \begin{bmatrix} 10 \\ 5 \end{bmatrix}$

Converges to $\theta_\infty = \begin{bmatrix} 1.9837 \\ 2.9933 \end{bmatrix}$



The optimization path



The value of the function

≡

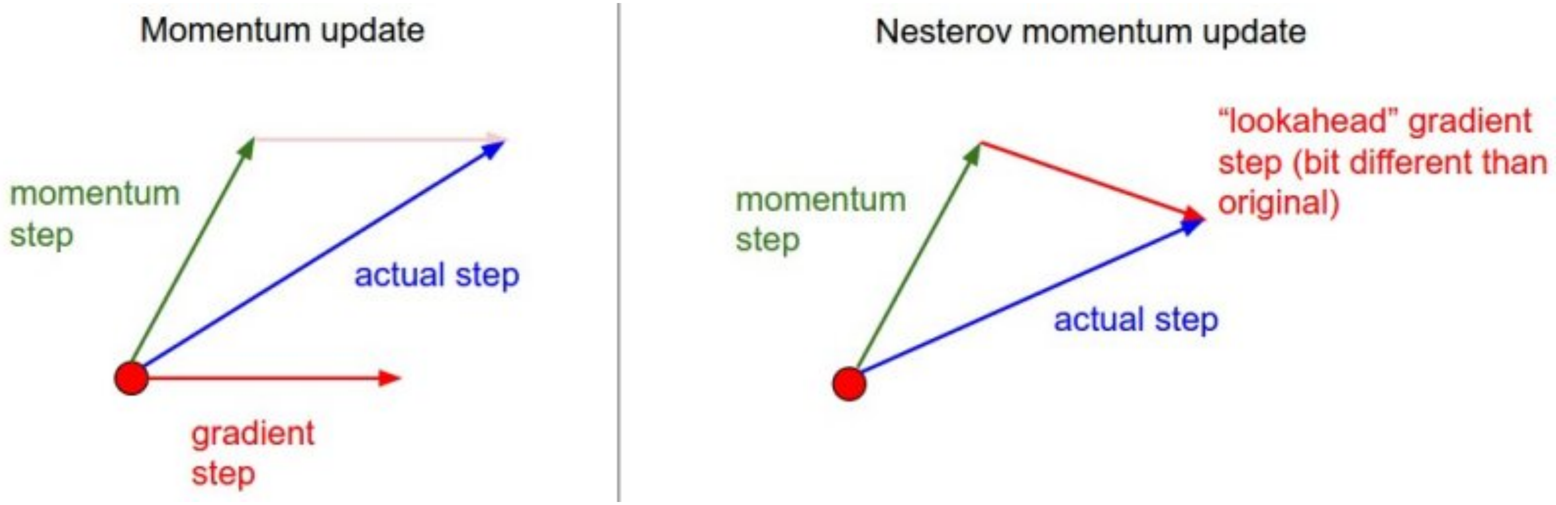
FIRST ORDER METHODS : NESTEROV MOMENTUM

Idea Look ahead to potentially correct the update. Based on Nesterov Accelerated Gradient. Formulation of (Sutskever et al., 2013)

Algorithm

- Start at θ_0
- for every minibatch :

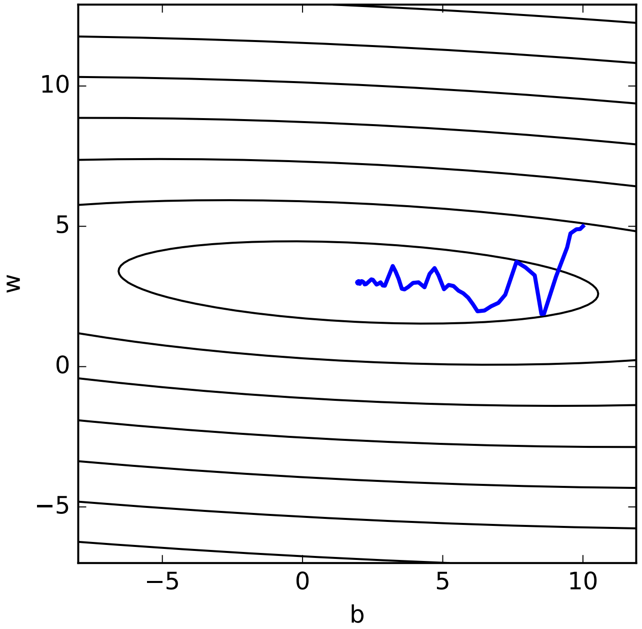
$$\begin{aligned} \overline{\theta}(t) &= \theta(t) + \mu v(t) \\ v(t+1) &= \mu v(t) - \epsilon \nabla_\theta J(\overline{\theta}(t)) \\ \theta(t+1) &= \theta(t) + v(t+1) \end{aligned}$$



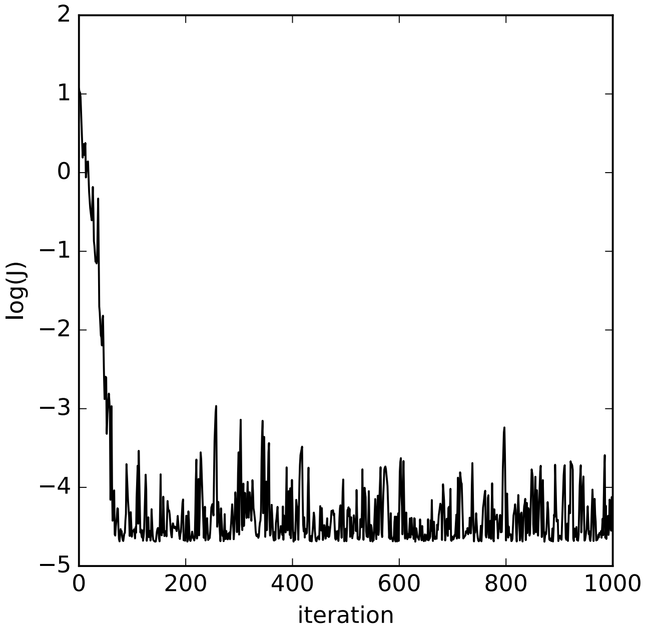
EXAMPLE USING SGD WITH NESTEROV MOMENTUM

Parameters : $\epsilon = 0.005, \mu = 0.8, \theta_0 = \begin{bmatrix} 10 \\ 5 \end{bmatrix}$

Converges to $\theta_\infty = \begin{bmatrix} 1.9738 \\ 2.9914 \end{bmatrix}$



The optimization path

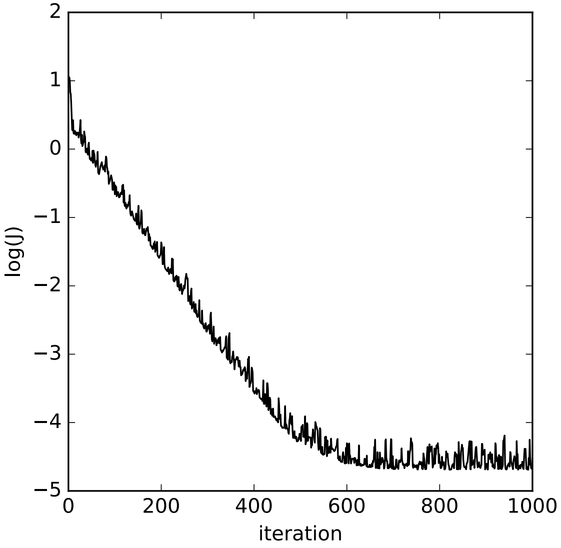


The value of the function

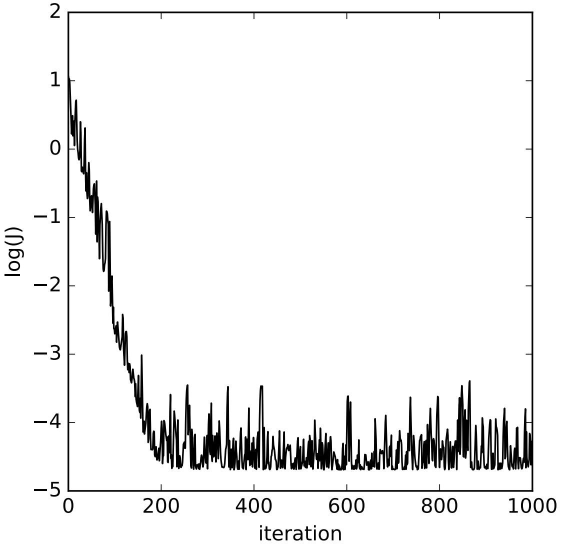
≡

COMPARISON OF THE 1ST ORDER METHODS

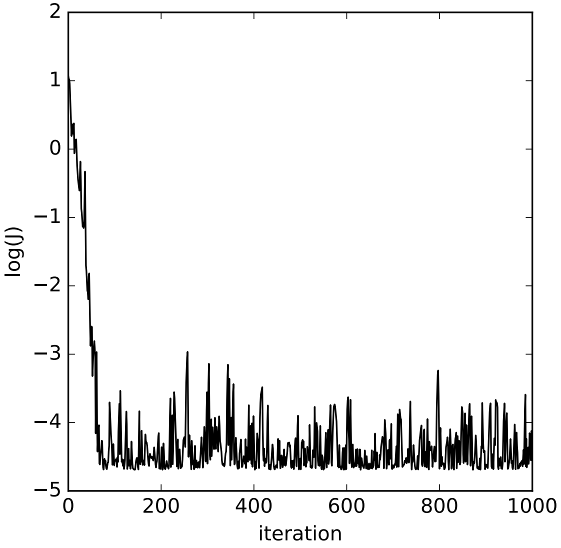
On our simple regression problem



SGD



Momentum

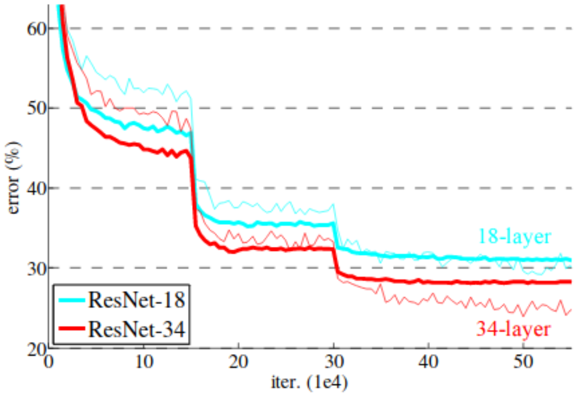


Nesterov

FIRST ORDER : ADAPTIVE LEARNING RATE

You should **always** adapt your learning rate with a learning rate scheduler

- Linear decrease from ϵ_0 downto ϵ_f
- halve the learning rate when the validation error stops improving
- halve the learning rate on a fixed schedule (every $50th$ epochs)

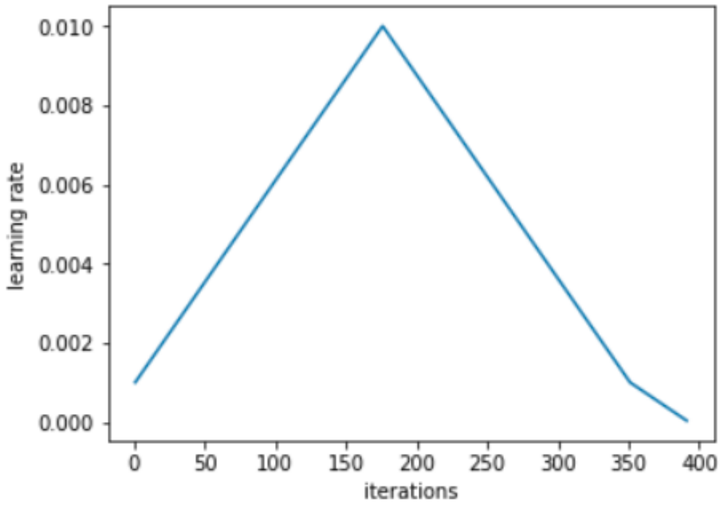


Resnet training curves. “The learning rate starts from 0.1 and is divided by 10 when the error plateaus”

≡

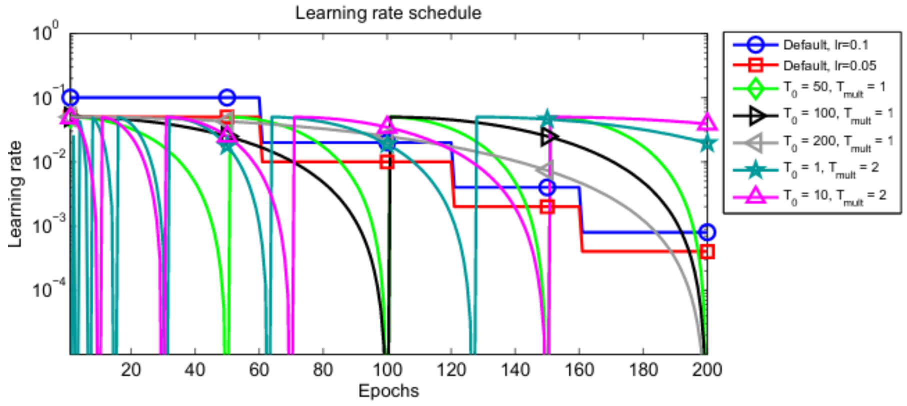
ADAPTIVE FIRST ORDER : ADAPTIVE LEARNING RATE

Some more recent approaches are changing the picture of “decreasing learning rate” (“Robbins Monro conditions”)



The 1cycle policy

See (Smith, 2018), The 1cycle policy - S. Gugger



SGDR

Stochastic Gradient Descent with Warm Restart (Loshchilov & Hutter, 2017)

The improved performances may be linked to reaching flatter minimums (i.e. with predictions less sensitive than sharper minimums). The models reached before the warm restarts can be averaged (see Snapshot ensemble).

It seems also that initial large learning rates tend to lead to better models on the long run (Y. Li et al., 2019)

ADAPTIVE FIRST ORDER : ADAGRAD

Adagrad Adaptive Gradient (Duchi, Hazan, & Singer, 2011)

- Accumulate the square of the gradient

$$r(t+1) = r(t) + \nabla_\theta J(\theta(t)) \odot \nabla_\theta J(\theta(t))$$

- Scale individually the learning rates

$$\theta(t+1) = \theta(t) - \frac{\epsilon}{\delta + \sqrt{r(t+1)}} \odot \nabla_\theta J(\theta(t))$$

The $\sqrt{.}$ is experimentally critical ; $\delta \approx [1e-8, 1e-4]$ for numerical stability.

Small gradients $\longrightarrow$ bigger learning rate for moving fast along flat directions
Big gradients $\longrightarrow$ smaller learning rate to calm down on high curvature.

But accumulation from the beginning is too aggressive. Learning rates decrease too fast.

ADAPTIVE FIRST ORDER : RMSPROP

RMSprop Hinton(unpublished, Coursera)

Idea: we should be using an exponential moving average when accumulating the gradient.

- Accumulate the square of the gradient

$$r(t+1) = \rho r(t) + (1-\rho)\nabla_\theta J(\theta(t)) \odot \nabla_\theta J(\theta(t))$$

- Scale individually the learning rates

$$\theta(t+1) = \theta(t) - \frac{\epsilon}{\delta + \sqrt{r(t+1)}} \odot \nabla_\theta J(\theta(t))$$

$\rho \approx 0.9$

ADAPTIVE FIRST ORDER : ADAM

Adaptive Moments (ADAM) (Kingma & Ba, 2015)

- Like momentum and RMSprop, store running averages of past gradients :

$$\begin{aligned} m(t+1) &= \beta_1 m(t) + (1-\beta_1)\nabla_\theta J(\theta(t) \\ v(t+1) &= \beta_2 v(t) + (1-\beta_2)\nabla_\theta J(\theta(t) \odot \nabla_\theta J(\theta(t) \end{aligned}$$

  $m(t)$ and $v(t)$ are the first moment and second (uncentered) moments of $\nabla_\theta J$. They are bias corrected $\hat{m}(t)$, $\hat{v}(t)$ and then :

$$\theta(t+1) = \theta(t) - \frac{\epsilon}{\delta + \sqrt{\hat{v}(t+1)}}\hat{m}(t+1)$$

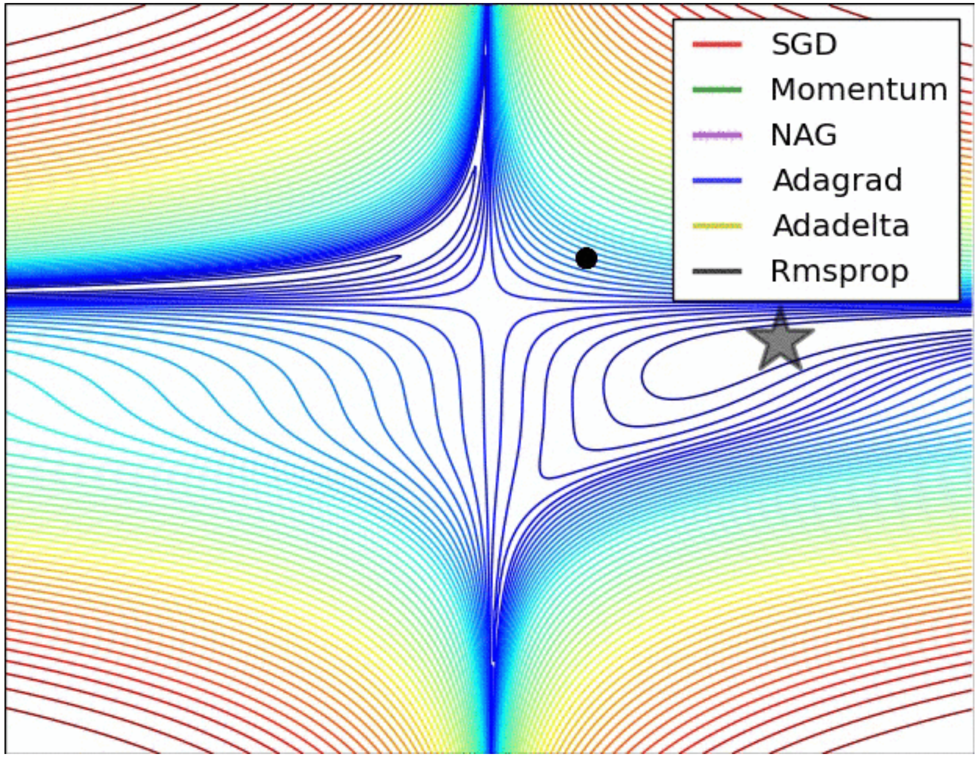
and some others : Adadelta (Zeiler, 2012), … , YellowFin (Zhang & Mitliagkas, 2018).

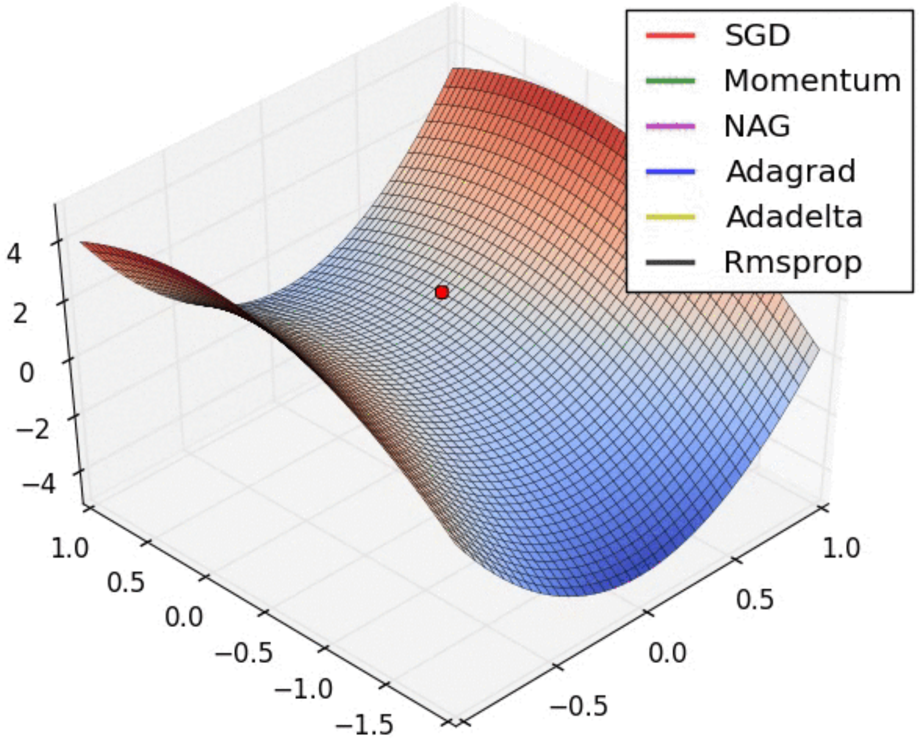
See Sebastian ruder blog post, or John Chen blog post

≡

FIRST ORDER : TO SUM UP

(Goodfellow, Bengio, & Courville, 2016) There is currently no consensus[…] **no single best** algorithm has emerged[…]the most popular and actively in use include **SGD**,**SGD with momentum**, **RMSprop**, **RMSprop with momentum**, **Adadelta** and **Adam**.





See also Chap. 8 of (Goodfellow et al., 2016)

≡

A GLIMPSE INTO SECOND ORDER METHODS

Rationale :

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T \nabla^2_\theta J(\theta_0)(\theta - \theta_0)$$

with $H = \nabla^2 J$ the Hessian matrix, a $n_\theta \times n_\theta$ matrix hosting the second derivatives of J.

The second derivates are much more noisy than the first derivative (gradient), a larger batch size is usually required to prevent instabilities.

- Conjugate gradient : using line search (or hessian) along $\nabla_\theta J(\theta_k)$
- Newton : **never** use except if you want to find critical points (Dauphin et al., 2014). Solves above for θ and find $\nabla^2_\theta J(\theta_0).(\theta - \theta_0) = -\nabla_\theta J(\theta_0)$
- Quasi Newton : BFGS (approximating H^{-1}), L-BFGS, and saddle-free versions (Dauphin et al., 2014).

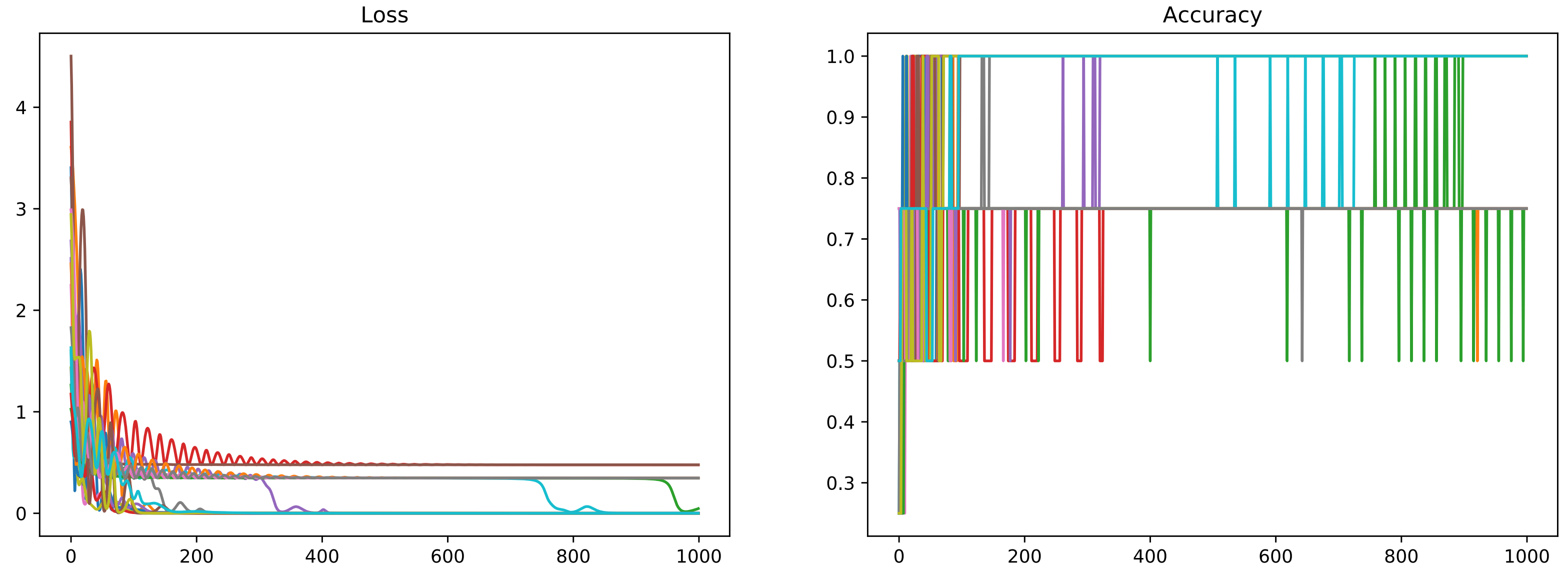
INITIALIZATION AND THE DISTRIBUTIONS OF ACTIVATIONS AND GRADIENTS

THE STARTING POINT IS IMPORTANT : XOR

XOR is easy right ?

- Model : 2-4-1, Sigmoid activations (great!); 17 parameters
- Init : $\mathcal{U}(-10, 10)$, bias=0 (hum hum)
- Loss : Binary cross entropy (great!)
- Optimizer : SGD (= 0.1, momentum=0.99)

But it fails miserably (6/20 fails). Tmax=1000



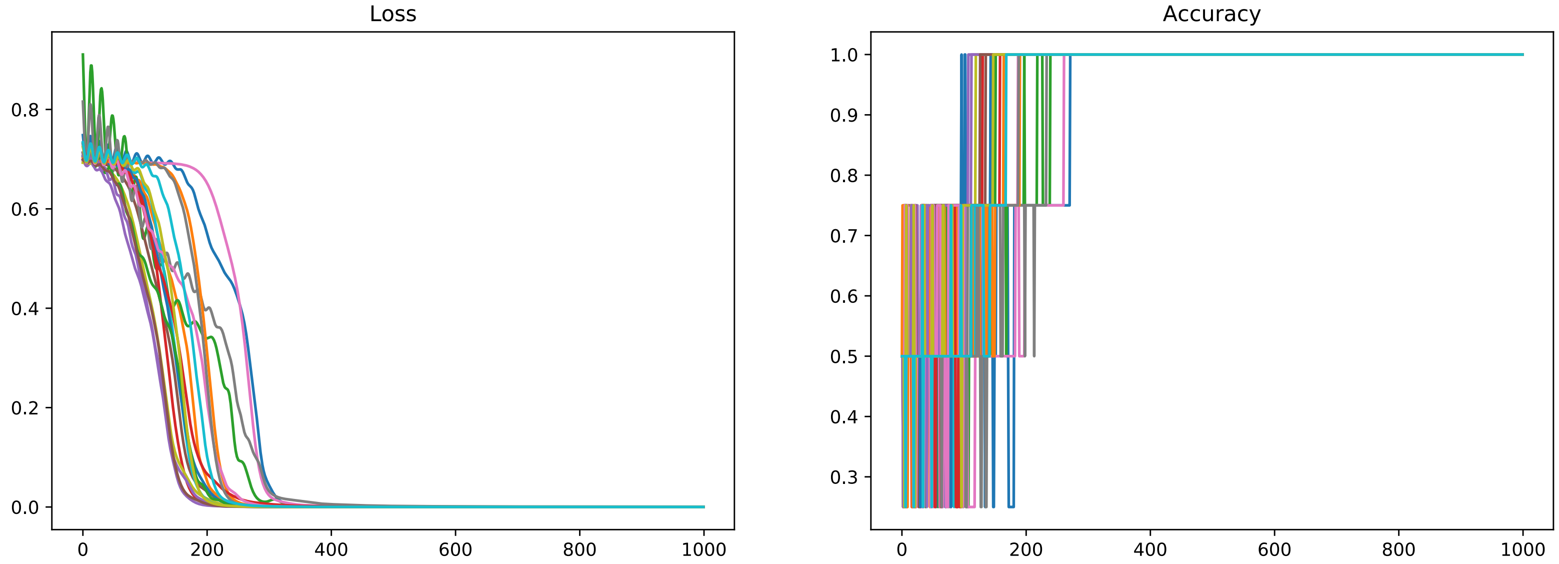
BCE Loss and accuracy on the training set

THE STARTING POINT IS IMPORTANT : XOR

XOR is easy right ?

- Model : 2-4-1, Sigmoid activations (great!); 17 parameters
- Init : $\mathcal{N}(0, \frac{1}{\sqrt{fan_{in}}})$, bias=0 (great!)
- Loss : Binary cross entropy (great!)
- Optimizer : SGD (= 0.1, momentum=0.99)

Now it is better (0/20 fails). Tmax=1000



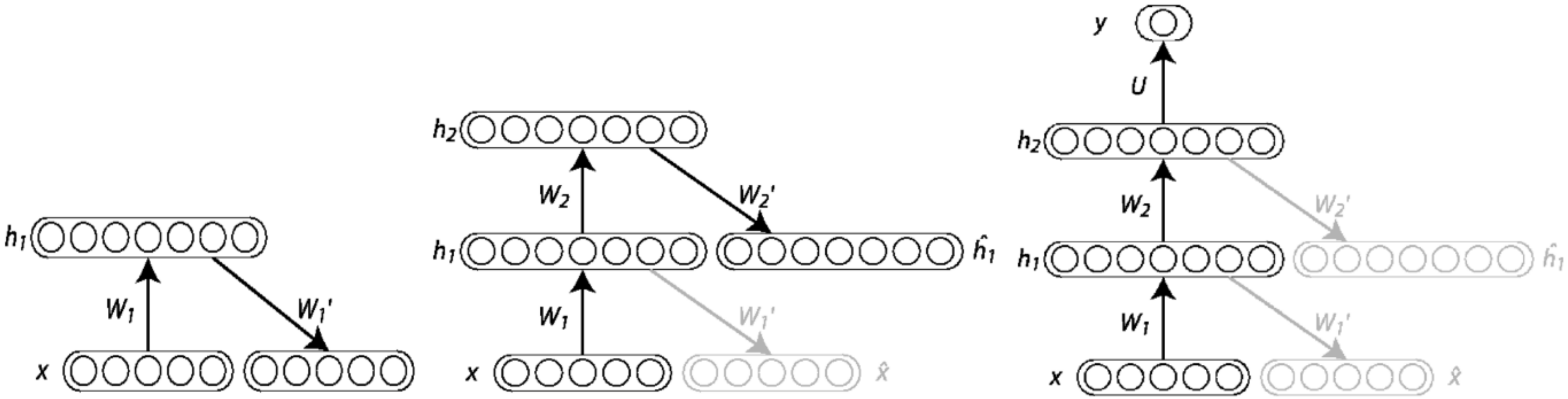
BCE Loss and accuracy on the training set

PRETRAINING

Historically, training deep FNN was known to be hard, i.e. bad generalization errors.

The starting point of a gradient descent has a dramatic impact :

- neural history compressors (Schmidhuber, 1992)
- competitive learning (Maclin & Shavlik, 1995)
- unsupervised pretraining based on Boltzman machines (Hinton, 2006)
- unsupervised pretraining based on auto-encoders (Bengio, Lamblin, Popovici, & Larochelle, 2006)



Pretraining with auto-encoders

Pretraining is no more used (because of xxRelu, Initialization schemes, ..)

STANDARDIZING YOUR INPUTS

Gradient descent converges faster if your data are normalized and decorrelated. Denote by $x_i \in \mathbb{R}^d$ your input data, $\hat{x}_i$ its normalized.

- Min-max scaling

$$\forall i, j\hat{x}_{i,j} = \frac{x_{i,j} - \min_k x_{k,j}}{\max_k x_{k,j} - \min_k x_{k,j} + \epsilon}$$

- Z-score normalization (goal: $\hat{\mu}_j = 0, \hat{\sigma}_j = 1$)

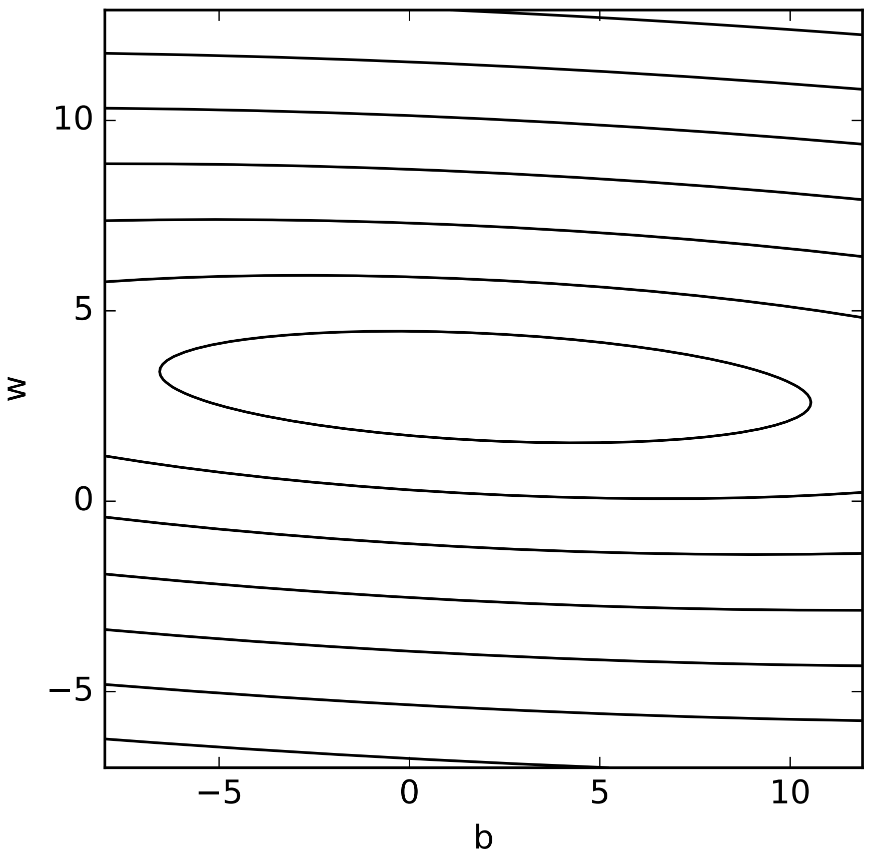
$$\forall i, j, \hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sigma_j + \epsilon}$$

- ZCA whitening (goal: $\hat{\mu}_j = 0, \hat{\sigma}_j = 1, \frac{1}{n-1}\hat{X}\hat{X}^T = I$)

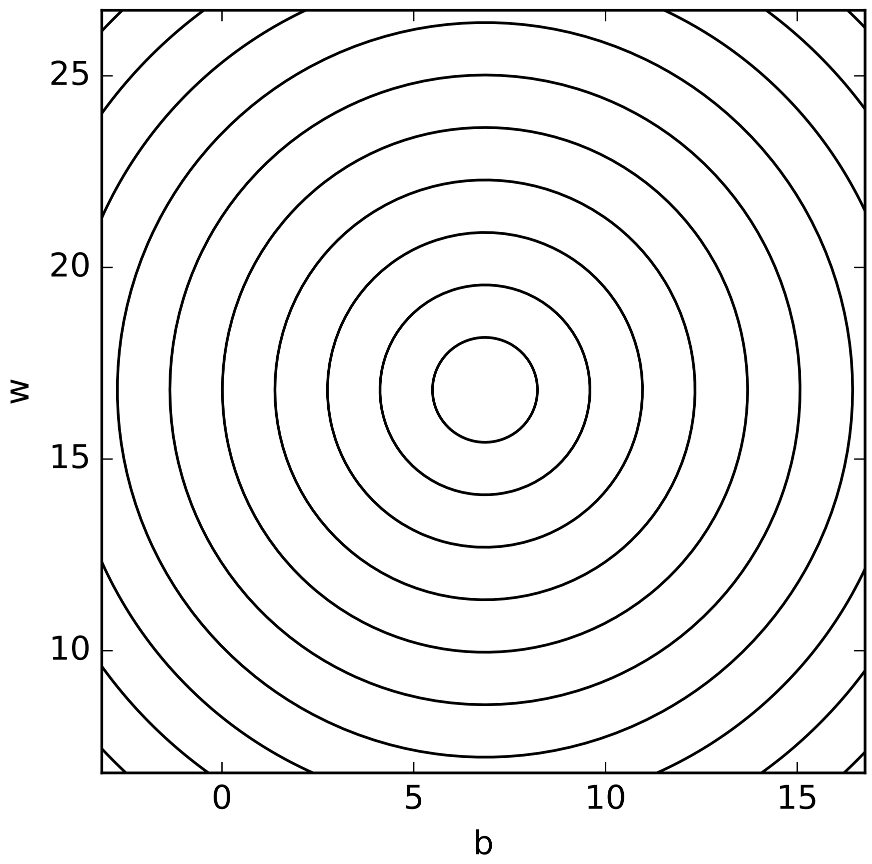
$$\hat{X} = WX, W = \frac{1}{\sqrt{n-1}}(XX^T)^{-1/2}$$

Z-SCORE NORMALIZATION / STANDARDIZING THE INPUTS

Remember our linear regression : $y = 3x + 2 + \mathcal{U}(-0.1, 0.1)$, L2 loss, 30 1D samples



Loss with raw inputs



Loss with standardized inputs

≡

GENERAL STRATEGY

- A good initialization should break the symmetry : constant initialization schemes make units learning all the same thing
- A good initialization should start optimization in a region of low capacity : linear neural network
- A good initialization scheme should preserve the distribution of the activations and gradients : exploding/vanishing gradients

THE EXPLODING AND VANISHING GRADIENT PROBLEM

The Fundamental Deep Learning Problem first observed by (Josef Hochreiter, 1991) for RNN, the gradient can either vanish or explode, especially in deep networks (RNN are very deep).

- Remember that the backpropagated gradient involves :

$$\frac{\partial J}{\partial x_l} = \frac{\partial J}{\partial y_L} W_L f'(y_l) W_{L-1} f'(y_{l-1}) \cdots$$

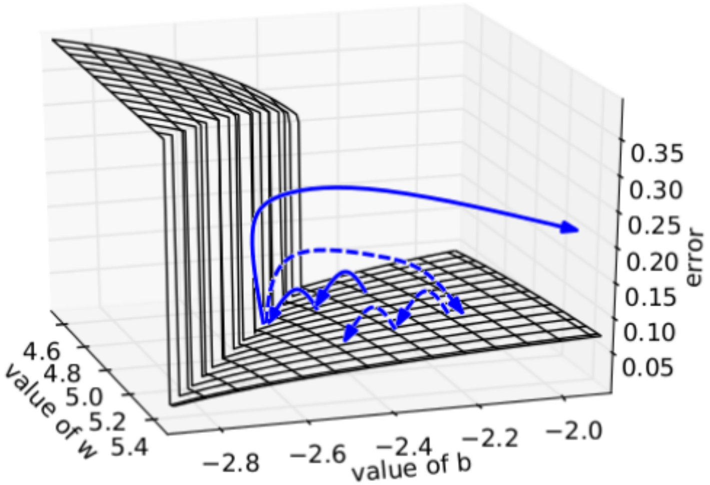
  with $y_l = W_l x_l + b, x_l = f(y_{l-1})$.

- We see a pattern like $(W.\, f')^L$ which can diverge or vanish for large L.
- especially, with the sigmoid : $f' < 1$.

With a ReLu, the positive part has $f' = 1$.

PREVENTING VANISHING/EXPLODING GRADIENT

- We must ensure a good flow of gradient :
  - using appropriate transfer functions ReLu, PreLu, etc..
  - using architectural elements :
    - ResNet (CNN) : shortcurt connections
    - LSTM (RNN): constant error caroussel
- We can prevent exploding gradient by clipping (Pascanu, Mikolov, & Bengio, 2013)



Exploding gradient and the effect of clipping. Experiment with 50 layers, single unit, sigmoid transfer function

LECUN INITIALIZATION

In (LeCun et al., 1998), Y. LeCun provided some guidelines on the design:

Aim Initialize the weights/biases to keep f in its linear part through multiple layers:

- Use a symmetric transfer function $f(x) = 1.7159 \tanh(\frac{2}{3}x), \rightarrow f(1) = 1, f(-1) = -1$

- set the biases to 0
- initialize randomly and independently from $\mathcal{N}(\mu = 0, \sigma^2 = \frac{1}{fan_{in}})$.

If $x \in \mathbb{R}^n$ is $\mathcal{N}(0, \Sigma = I)$, $w \in \mathbb{R}^n$ is $\mathcal{N}(\mu = 0, \Sigma = \frac{1}{n} I)$, then :

$$E[w^T x + b] = E[w^T x] = \sum_i E[w_i x_i] = \sum_i E[w_i] E$$

$$\begin{aligned} var[w^T x + b] &= var[w^T x] \\ &= \sum_i \sigma^2_{w_i} \sigma^2_{x_i} + \sigma^2_{w_i} \mu^2_{x_i} + \mu^2_{w_i} \sigma^2_{x_i} \\ &= \sum_i \sigma^2_{w_i} \sigma^2_{x_i} = \frac{1}{n} \sum \sigma^2_{x_i} = 1 \end{aligned}$$

x_i, w_i are all pairwise independent.

XAVIER (GLOROT) INITIALIZATION

Idea we must preserve the same distribution along the **forward** and **backward** pass (Glorot & Bengio, 2010).

This prevents:

- the saturation of saturating transfer functions (e.g. tanh, sigmoid)
- vanishing/exploding gradient

Glorot (Xavier) initialization scheme for a feedforward network $f(W_n..f(W^1f(W_0x+b_0)+b_1)...+b_n)$ with layer sizes n_i:

- the input dimensions should be centered, normalized, uncorrelated
- symmetric activation function, with $f'(0)=1$ (e.g. $f(x)=\tanh(x)$, $f(x)=4(\frac{1}{1+e^{-x}}-0.5)$)

Assuming the linear regime $f'()=1$ of the network :

$$\text{Forward propagation variance constraint :}\forall i, fan_{in_i}\sigma^2_{W_i}=1$$
$$\text{Backward propagation variance constraint :}\forall i, fan_{out_i}\sigma^2_{W_i}=1$$

Compromise : $\forall i, \frac{1}{\sigma^2_{W_i}}=\frac{fan_{in}+fan_{out}}{2}$

- Glorot (Xavier) uniform : $\mathcal{U}(-\frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}+fan_{out}}})$, b=0

- Glorot (Xavier) normal : $\mathcal{N}(0, \frac{\sqrt{2}}{\sqrt{fan_{in}+fan_{out}}})$, b=0

HE INITIALIZATION

Idea we must preserve the same distribution along the **forward** and **backward** pass for **rectifiers** (He et al., 2015). For a feedforward network $f(W_n..f(W^1f(W_0x+b_0)+b_1)...+b_n)$ with layer sizes n_i:

- the input dimensions should be centered, normalized, uncorrelated
- ReLu activation $f(x) = \max(x,0)$
- weights initialized with symmetric distribution, zero mean $\mu_{w_l} = 0$, independently. Bias set to $b = 0$
- the components of x_l are assumed i.i.d.; Note they are not centered (because of ReLu)

$$\mathbf{y}_l = \begin{bmatrix} \vdots \\ y_l \\ \vdots \end{bmatrix} = W_l\mathbf{x}_l + \mathbf{b} = W_l f(\mathbf{y}_{l-1}) + \mathbf{b}$$

$$\mu_{y_l} = E[\sum_i w_{l,i}x_{l,i}] = \mu_{w_l}\sum_i \mu_{x_{l,i}} = 0$$

$$\sigma^2_{y_l} = n_l\sigma^2_{w_l x_l} = n_l\mu_{w_l^2}\mu_{x_l^2} = n_l\sigma^2_{w_l}\mu_{x_l^2}(\text{because } \mu_{w_l} = 0)$$

$$\mu_{x_l^2} = \int_{y_{l-1}} \max(0,y_{l-1})^2 dp_{y_{l-1}} = \frac{1}{2}\mu_{y^2_{l-1}} = \frac{1}{2}\sigma^2_{y_{l-1}}(\mu_{y_{l-1}} = 0 \text{ and } y_{l-1} \text{ has symmetric distrib.})$$

So, $\sigma^2_{y_l} = \frac{1}{2}n_l\sigma^2_{w_l}\sigma^2_{y_{l-1}}$. To preserve the variance, we must guarantee $\frac{1}{2}n_l\sigma^2_{w_l} = 1$.

We used : if X and Y are independent : $\sigma^2_{X.Y} = \mu_{X^2}\mu_{Y^2} - \mu^2_X\mu^2_Y$

HE INITIALIZATION

Idea we must preserve the same distribution along the **forward** and **backward** pass for **rectifiers** (He et al., 2015). For a feedforward network $f(W_n . . f(W^1 f(W_0 x + b_0) + b_1) . . . + b_n)$ with layer sizes n_i:

- the input dimensions should be centered, normalized, uncorrelated
- ReLu activation $f(x) = \max(x, 0)$
- weights initialized with symmetric distribution, zero mean $\mu_{w_l} = 0$, independently. Bias set to $b = 0$
- the components of x_l are assumed i.i.d.; Note they are not centered (because of ReLu)

$$\text{Forward propagation variance constraint :} \forall i, \frac{1}{2} fan_{in_i} \sigma^2_{W_i} = 1$$

$$\text{Backward propagation variance constraint :} \forall i, \frac{1}{2} fan_{out_i} \sigma^2_{W_i} = 1$$

He suggests to use either one or the other, e.g. $\sigma^2_{W_i} = \frac{2}{fan_{in}}$

- He uniform : $\mathcal{U}(-\frac{\sqrt{6}}{\sqrt{fan_{in}}}, \frac{\sqrt{6}}{\sqrt{fan_{in}}})$, b=0

- He normal : $\mathcal{N}(0, \frac{\sqrt{2}}{\sqrt{fan_{in}}})$, b=0

Note: for PreLu : $\frac{1}{2}(1 + a^2) fan_{in} \sigma^2_{W_i} = 1$

WEIGHT INITIALIZATION IN PRACTICE (PYTORCH)

By default, the parameters are initialized randomly. e.g. in torch.nn.Linear :

```
class Linear(torch.nn.Module):
    def __init__(self):
        ...
        self.reset_parameters()

    def reset_parameters(self) -> None:
        # Setting a=sqrt(5) in kaiming_uniform is the same as initializing with
        # uniform(-1/sqrt(in_features), 1/sqrt(in_features)). For details, see
```

Oh, but that’s not what we should use for ReLu ?!?! Indeed you are right, see this issue. This is to avoid breaking with the way torch(lua) was initializing.

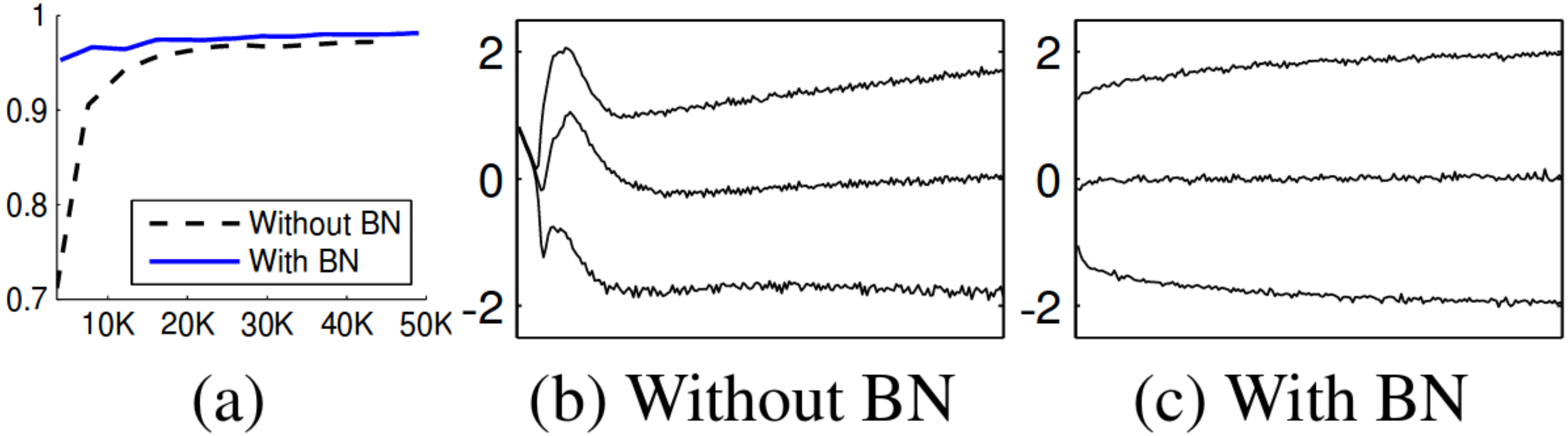
```

import torch.nn.init as init

class MyModel(torch.nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.classifier =  nn.Sequential(
            *linear_relu(input_size, 256),
```

```
def linear_relu(dim_in,
  dim_out):
    return
  [nn.Linear(dim_in,
  dim_out),

  nn.ReLU(inplace=True)]
```

INTERNAL COVARIATE SHIFT

(Ioffe & Szegedy, 2015) observed the change in distribution of network activations due to the change in network parameters during training.

Experiment 3 fully connected layers (100 units), sigmoid, softmax output, MNIST dataset



left) Test accuracy, right)Distribution of the activations of the last hidden layer during training, {15, 50, 85}th percentile

≡

BATCH NORMALIZATION

Idea standardize the activations of every layers to keep the same distributions **during training** (Ioffe & Szegedy, 2015)

- The gradient must be aware of this normalization, otherwise may get parameter explosion (see (Ioffe & Szegedy, 2015)) $\longrightarrow$ we need a differentiable normalization layer
- introduces a differentiable Batch Normalization layer :

$$z = g(Wx + b) \rightarrow z = g(BN(Wx))$$

BN operates element-wise :

$$y_i = BN_{\gamma,\beta}(x_i) = \gamma\hat{x}_i + \beta$$
$$\hat{x}_i = \frac{x_i - \mu_{\mathcal{B},i}}{\sqrt{\sigma^2_{\mathcal{B},i} + \epsilon}}$$

with $\mu_{\mathcal{B},i}$ and $\sigma_{\mathcal{B},i}$ statistics computed on the mini batch during training.

Learning faster, with better generalization.

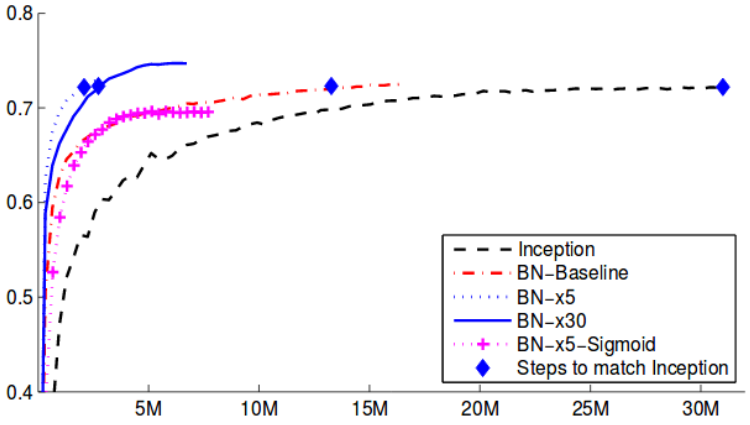


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

BATCH NORMALIZATION

During **training**

- put BN layers everywhere along the network, after the linear layer, before the ReLus
- evaluate the statistics μ, σ over the minibatches
- update an exponential moving average of the mean $\mu_{\mathcal{B}}$ and variance $\sigma_{\mathcal{B}}^2$

During **inference** (test) :

- use the running average as the statistics to standardize : this is now just a fixed affine transform.

⚠Do not forget to switch to **test mode** :

```
model = MyModel()   # a pytorch nn.Module
# For training
model.train()
# For testing
model.test()
```

Some recent works challenge the idea of covariate shift (Santurkar, Tsipras, Ilyas, & Ma, 2018), (Bjorck, Gomes, Selman, & Weinberger, 2018). The loss seems smoother allowing larger learning rates, better generalization, robustness to hyperparameters.

REGULARIZATION

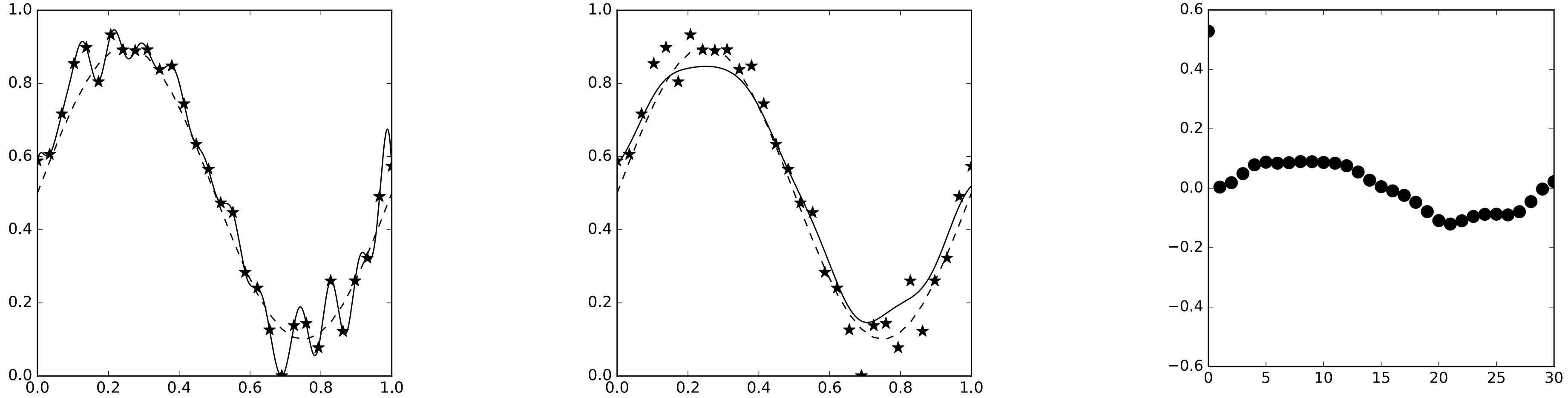
L2 PENALTY

Add a L2 penalty on the weights, $\alpha > 0$

$$\begin{aligned} J(\theta) &= L(\theta) + \frac{\alpha}{2}\|\theta\|_2^2 = L(\theta) + \frac{\alpha}{2}\theta^T\theta \\ \nabla_\theta J &= \nabla_\theta L + \alpha\theta \\ \theta &\leftarrow \theta - \epsilon\nabla_\theta J = (1-\alpha\epsilon)\theta - \epsilon\nabla_\theta L \end{aligned}$$

Called L2 regularization, Tikhonov regularization, weight decay

Example RBF, 1 kernel per sample, $N = 30$, noisy inputs,



$\alpha = 0$ $\alpha = 2$ $\theta^\star$

See chap 7 of (Goodfellow et al., 2016) for a geometrical interpretation

Intuition : for linear layers, the gradient of the function equals the weights. Small weights → small gradient → smooth function.

L2 PENALTY

In theory, regularizing the bias will cause underfitting

Example

$$J(w,b) = \frac{1}{N}\sum_{i=1}^{N} \|y_i - b - w^T x_i\|_2^2$$

$$\nabla_b J(w,b) \implies b = (\frac{1}{N}\sum_i y_i) - w^T(\frac{1}{N}\sum_i x_i)$$

If your data are centered (as they should), the optimal bias is the mean of the targets.

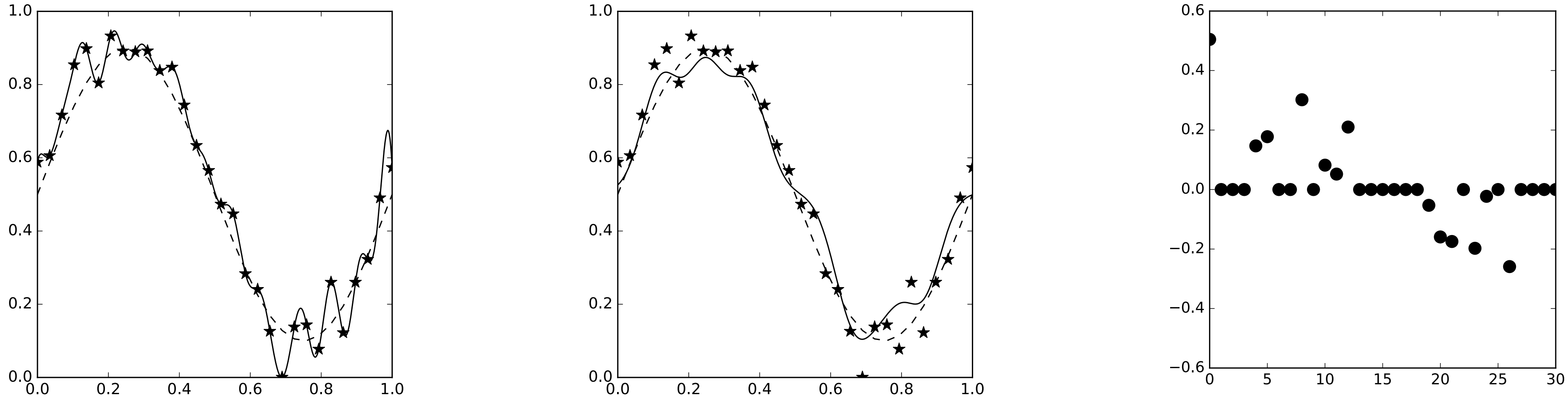
L1 PENALTY

Add a L1 penalty to the weights :

$$J(\theta) = L(\theta) + \alpha\|\theta\|_1 = L(\theta) + \alpha \sum_i |\theta_i|$$

$$\nabla_\theta J = \nabla_\theta L + \alpha \text{sign}(\theta)$$

Example RBF, 1 kernel per sample, $N = 30$, noisy inputs,



$\alpha = 0$

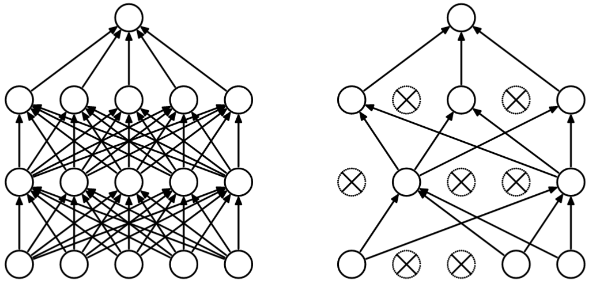
$\alpha = 0.003$

$\theta^\star$

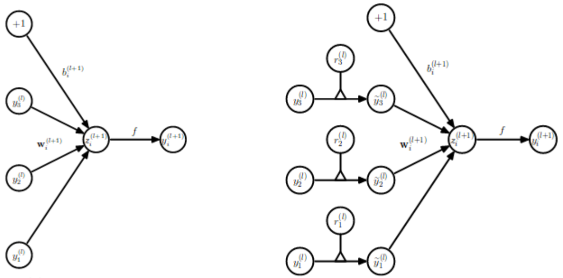
See chap 7 of (Goodfellow et al., 2016) for a mathematical explanation in a specific case. Sparsity used for feature selection with LASSO (filter/wrapper/**embedded**).

DROPOUT

Introduced in (Srivastava, Hinton, Krizhevsky, Sutskever, & Salakhutdinov, 2014):



(a) Standard Neural Net (b) After applying dropout.



(a) Standard network (b) Dropout network

Figure 3: Comparison of the basic operations of a standard and dropout network.

Idea 1 : preventing co-adaptation. A pattern is robust by itself not because of others doing part of the job.
Idea 2 : average of all the sub-networks (ensemble learning)

How :

- for every minibatch, zeroes hidden and input activations with probability p ($p = 0.5$ for hidden, $p = 0.2$ for input). At test time, multiply every activations by p
- “Inverted” dropout : multiply the kept activations by p at train time. At test time, just do a normal forward pass.

≡

DROPOUT

- Usually, after all fully connected layers (p=0.5) and input layer
- less usual on convolutional layers (because these are already regularized)

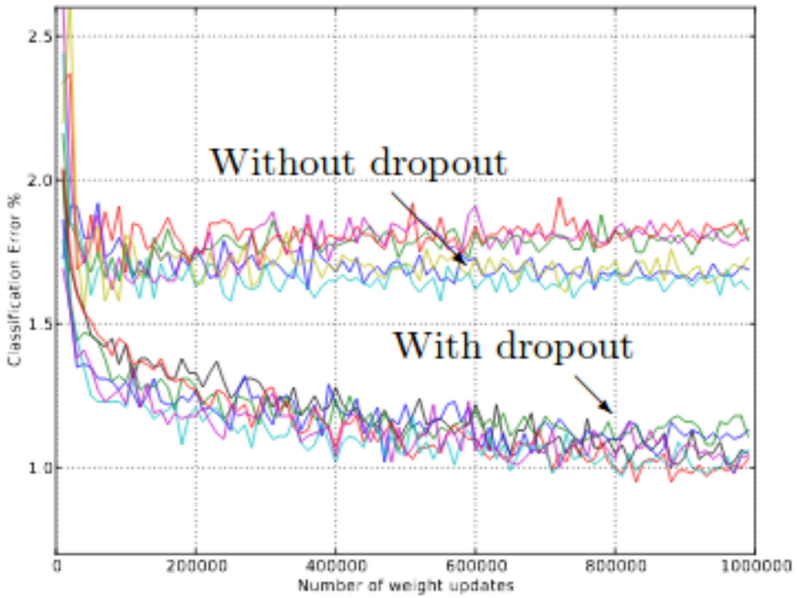


Figure 4: Test error for different architectures with and without dropout. The networks have 2 to 4 hidden layers each with 1024 to 2048 units.

Can be interpreted as if training/averaging all the possible subnetworks.

L1/L2/DROPOUT IN PYTORCH

L1/L2

```
class MyModel(nn.Module):
    def __init__(..., l2_reg, ..):
        self.lin1 = nn.Linear(784, 256)
        self.lin2 = nn.Linear(256, 256)
        self.l2_reg = l2_reg

    def penalty(self):
        return l2_reg *
```

```
def train():
    ...
    optimizer.zero_grad()
    loss.backward()
    model.penalty().backward()
```

Dropout

```

import torch.nn as nn

class MyModel(nn.Module):
    def __init__(self, ..):
        self.classifier = nn.Sequential(
            *dropout_linear_relu(784, 128,
  0.5),
```

```
def dropout_linear_relu(dim_in, dim_out,
  p_zeroed):
    return [nn.Dropout(p_zeroed),
            nn.Linear(dim_in, dim_out),
            nn.ReLU(inplace=True)]
```

EARLY STOPPING

Split your data in three sets :

- training set : for training ..
- validation set: for choosing the hyperparameters (learning rates, number of layers, layer size, momentum, …)
- test set : for estimation the generalization error

Everything can be placed in a cross validation loop.

Early stopping is about keeping the model with the **lowest validation loss**.

```
# Training over an epoch
for X, y in tqdm.tqdm(train_dataloader):
    X, y = X.to(device), y.to(device)
    ...
    optimizer.step()
# Model checkpoint
val_loss = test(model, valid_loader)
```

DATA, DATA, WE NEED DATA !

The best regularizer you may find is **data.** The more you have, the better you learn.

- you can use pretrained models on some tasks as an initialization for learning your task (but may fail due to **domain shift**) : check the Pytorch Hub
- you can use unlabeled data for pretraining your networks (as done in 2006s) with auto-encoders / RBM : unsupervised/semi-supervised learning
- you can apply **random transformations** to your data : **dataset augmentation**, see for example alubmentations.ai

LABEL SMOOTHING

Introduced in (Szegedy, Vanhoucke, Ioffe, Shlens, & Wojna, 2015) in the context of Convolutional Neural Networks.

Idea : Preventing the network to be over confident on its predictions on the training set.

Recipe : in a k-class problem, instead of using **hard targets** $\in \{0, 1\}$, use **soft targets** $\in \{\frac{\alpha}{k}, 1 - \alpha\frac{k-1}{k}\}$ (weighted average between the hard targets and uniform target). $\alpha \approx 0.1$.

See also (Müller, Kornblith, & Hinton, 2020) for several experiments.

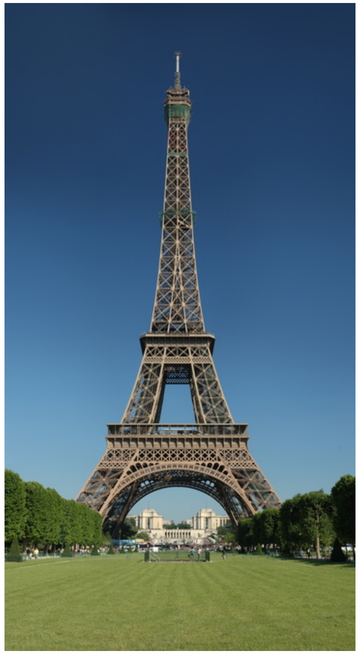
≡

CONVOLUTIONAL NEURAL NETWORKS

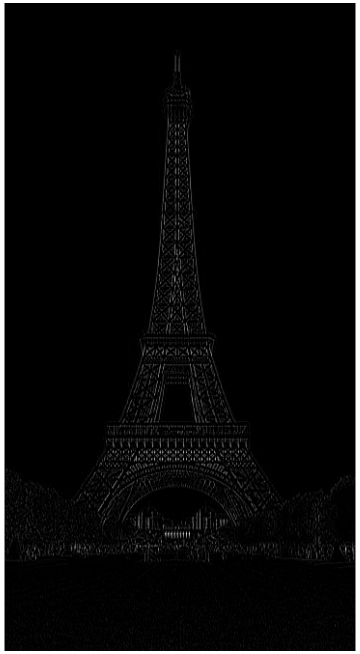
EXTRACTING FEATURES WITH CONVOLUTIONS

From data that have a spatial structure (locally correlated), features can be extracted with convolutions.

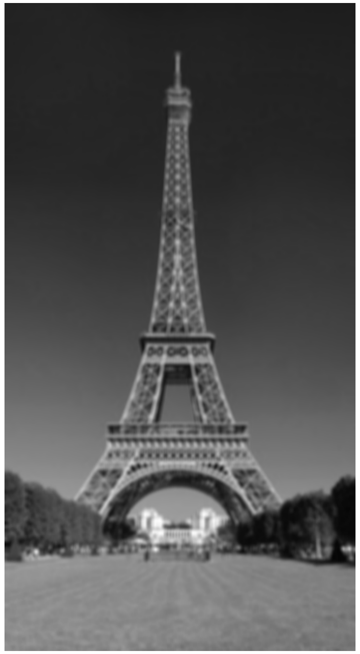
On **Images**



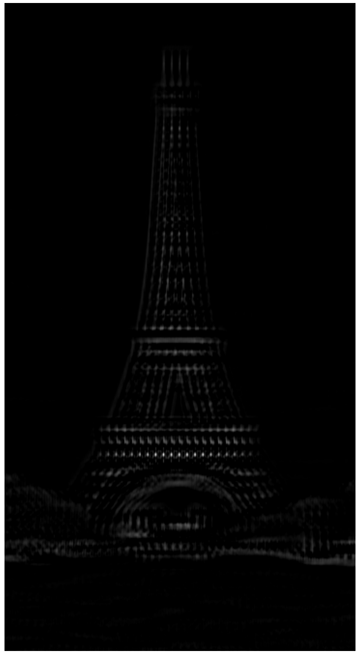
Original image



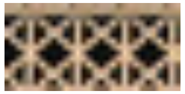
Discrete laplacian



Gaussian blur



Pattern matching



The pattern

That also makes sense for **temporal series** that have a structure in time.

A CONVOLUTION AS A SPARSE MATRIX MULTIPLY

What is a convolution : Example in 2D

Seen as a matrix multiplication

Given two 1D-vectors f, k, say $k = [c, b, a]$

$$(f * k) = \begin{bmatrix} b & c & 0 & 0 & \cdots & 0 & 0 \\ a & b & c & 0 & \cdots & 0 & 0 \\ 0 & a & b & c & \cdots & 0 & 0 \\ 0 & 0 & a & b & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & b & c \\ 0 & 0 & 0 & 0 & \cdots & a & b \end{bmatrix} \cdot \begin{bmatrix} \\ f \\ \\ \end{bmatrix}$$

COMPOSITION TO LEARN HIGHER LEVEL FEATURES

Local features can be combined to learn higher level features.

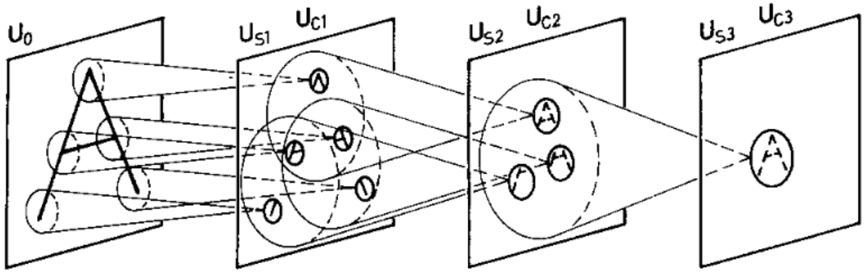
Let us build a house detector

ARCHITECTURE

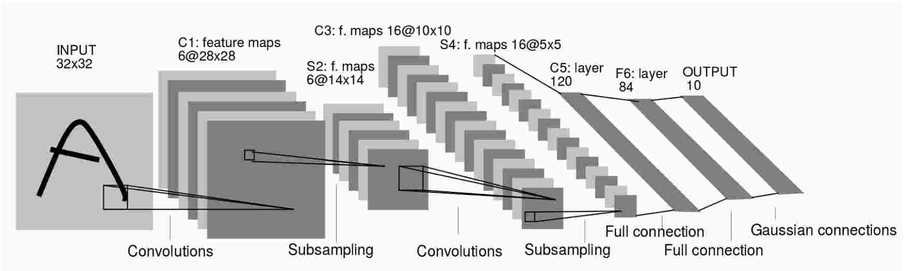
Ideas Using the structure of the inputs to limit the number of parameters without limiting the expressiveness of the network

- For inputs with spatial (or temporal) correlations, features can be extracted with **convolutions of local kernels**
- A convolution can be seen as a fully connected layer with :
  - a lot of weights set exactly to 0
  - a lot of weights **shared** across positions

$\rightarrow$ strongly regularized !



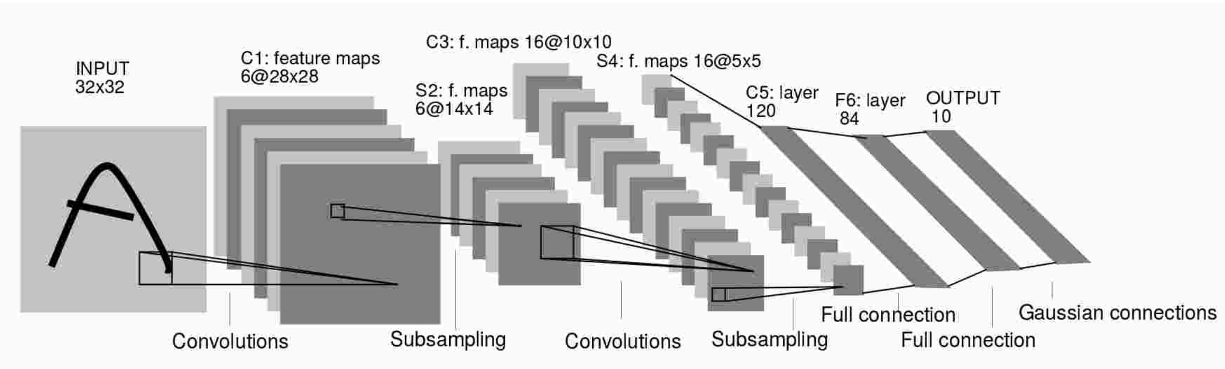
Neocognitron (Fukushima, 1980)



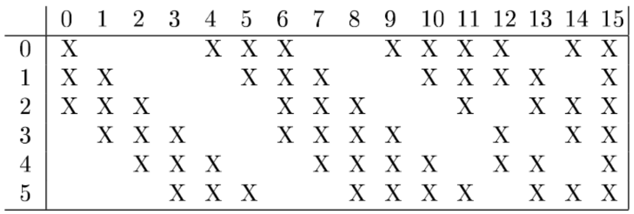
LeNet5 (LeCun et al., 1989)

VANILLA CNN OF LECUN

The architecture of LeNet-5 (LeCun et al., 1989), let’s call it the **Vanilla CNN**



LeNet5 (LeCun et al., 1989)



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

TABLE I

Each column indicates which feature map in S2 are combined by the units in a particular feature map of C3.

Architecture

Two main parts :

- convolutional part : C1 -> C5 : convolution - non-linearity - subsampling
- fully connected part : linear - non-linearity

Specificities :

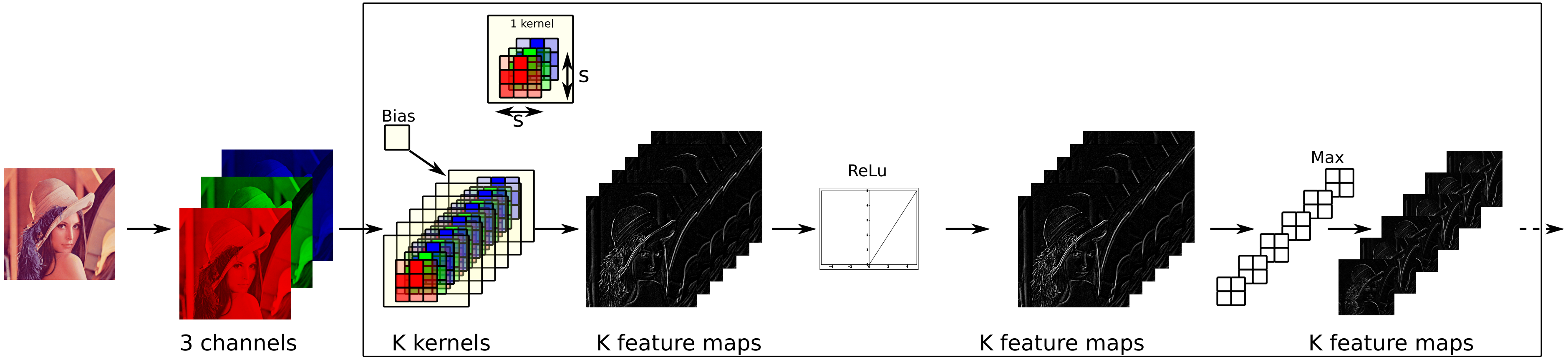
- Weighted sub-sampling
- Gaussian connections (RBF output layer)
- connectivity pattern $S_2 - C_3$ to reduce the number of weights

Number of parameters :

Layer	Parameters
C_1	156
S_2	12
C_3	1.516
S_4	32
C_5	48.120
F_6	10.164

≡

CNN VOCABULARY



The building blocks of the convolutional part of a vanilla CNN

Convolution :
- size (e.g. $3 \times 3, 5 \times 5$)
- padding (e.g. $1, 2$)
- stride (e.g. 1)

Pooling (max/average):
- size (e.g. 2×2)
- padding (e.g. 0)
- stride (e.g. 2)

We work with 4D tensors for 2D images, 3D tensors for nD temporal series (e.g. multiple simultaneous recordings), 2D tensors for 1D temporal series

In Pytorch, the tensors follow the Batch-Channel-Height-Width (**BCHW**, channel-first) convention. Other frameworks, like TensorFlow or CNTK, use BHWC (channel-last).

CNN IN PRACTICE

Pytorch code for implementing a CNN : Conv1D Conv2D, MaxPool1D MaxPool2D, AveragePooling, etc…

```
conv_model   = nn.Sequential(
    *conv_relu_maxpool(cin=3, cout=32,
                       csize=3, cstride=1,
  cpad=1,
                       msize=2, mstride=2,
  mpad=0),
    *conv_relu_maxpool(cin=3, cout=64,
```

```
def conv_relu_maxpool(cin, cout, csize,
  cstride, cpad, msize, mstride, mpad):
    return [nn.Conv2d(cin, cout, csize,
  cstride, cpad),
             nn.ReLU(inplace=True),
             nn.MaxPool2d(msize, mstride,
  mpad)
```

e.g. Conv2d(32, 64, 3, 1, 1)

```
fc_model = nn.Sequential(
    *linear_relu(output_size, 256),
    nn.Linear(256, num_classes)
)
```

How can I get the feature dimensions of conv_model output ?

```
dummy_output = conv_model(torch.zeros((1, 3, height, width)))
output_size = np.prod(dummy_output.shape[1:] )
```

CNN IN PRACTICE

All of these should fit into a nn.Module subclass :

```
class MyModel(torch.nn.Module):
    def __init__(self, ....):
        super(MyModel, self).__init__()
        self.conv_model = nn.Sequential(...)
        output_size = ...
        self.fc_model = nn.Sequential(...)
```

You can also use the recently introduced nn.Flatten layer.

TRANSPOSED CONVOLUTION

Given two 1D-vectors x_1, k, say $k = [c, b, a]$

$$y_1 = (x_1 * k) = \begin{bmatrix} b & c & 0 & 0 & \cdots & 0 & 0 \\ a & b & c & 0 & \cdots & 0 & 0 \\ 0 & a & b & c & \cdots & 0 & 0 \\ 0 & 0 & a & b & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & b & c \\ 0 & 0 & 0 & 0 & \cdots & a & b \end{bmatrix} \cdot \begin{bmatrix} \\ x_1 \\ \\ \end{bmatrix} = W_k x_1$$

If we compute the gradient of the loss, in denominator layout:

$$\frac{\partial L}{\partial x_1} = \frac{\partial y_1}{\partial x_1} \frac{\partial L}{\partial y_1} = W_k^T \frac{\partial L}{\partial y_1}$$

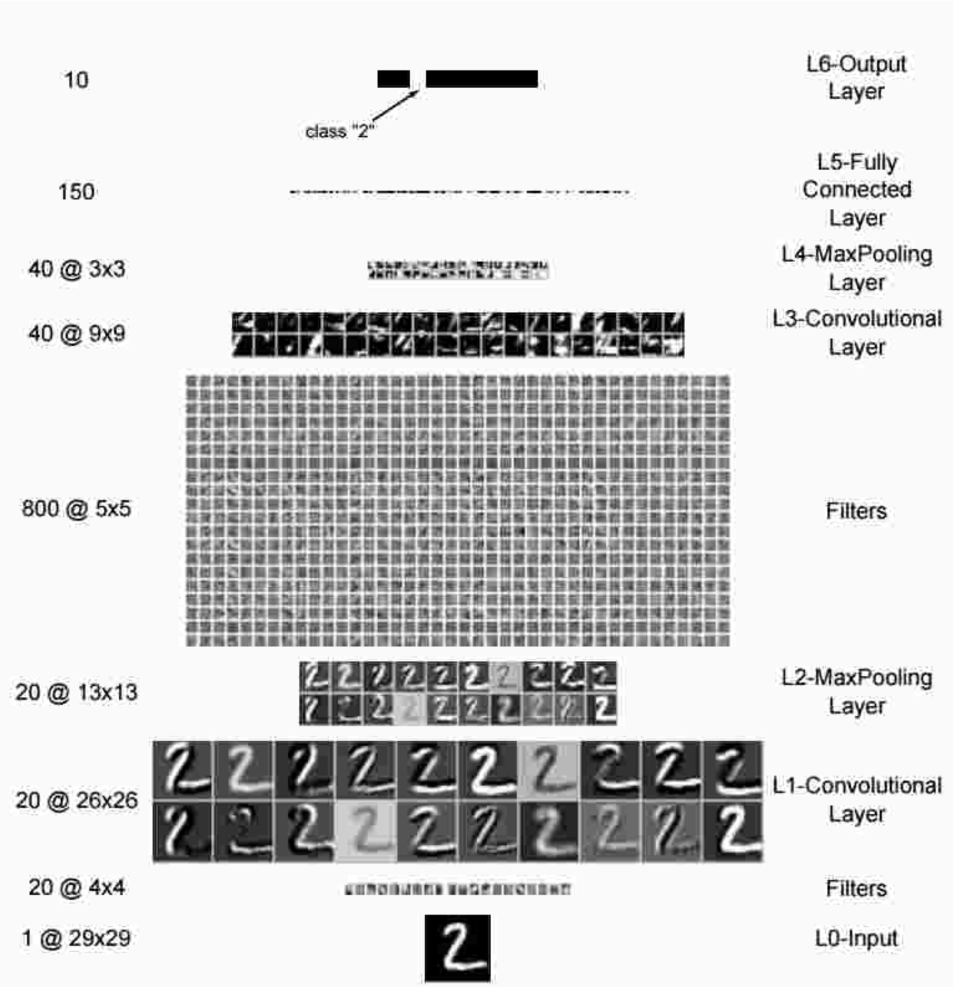
Hence, it is coined the term **transposed convolution** or **backward convolution**. This will pop up again when speaking about **deconvolution**.

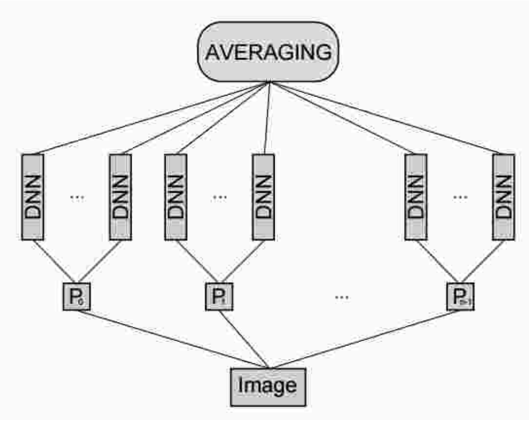
≡

10 YEARS OF CNN REVOLUTION

MULTICOLUMN CDNN

Introduced in (Ciresan, Meier, & Schmidhuber, 2012), ensemble of CNNs trained with dataset augmentation

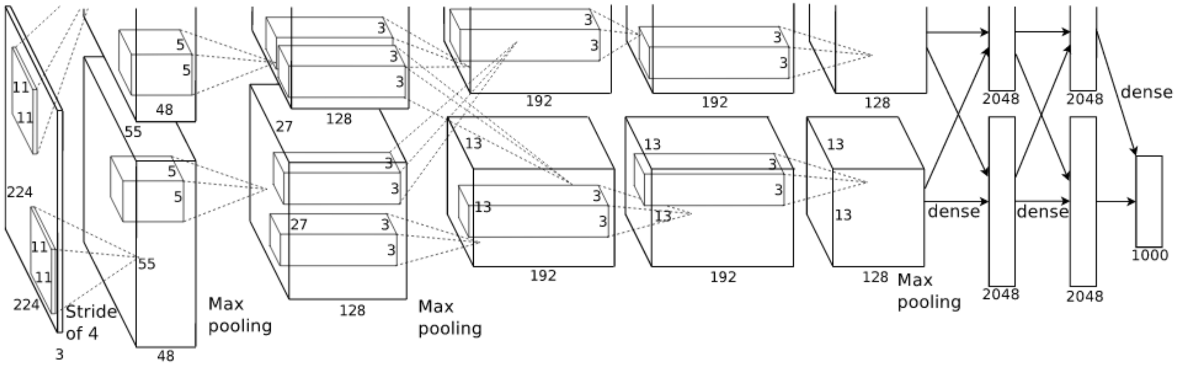


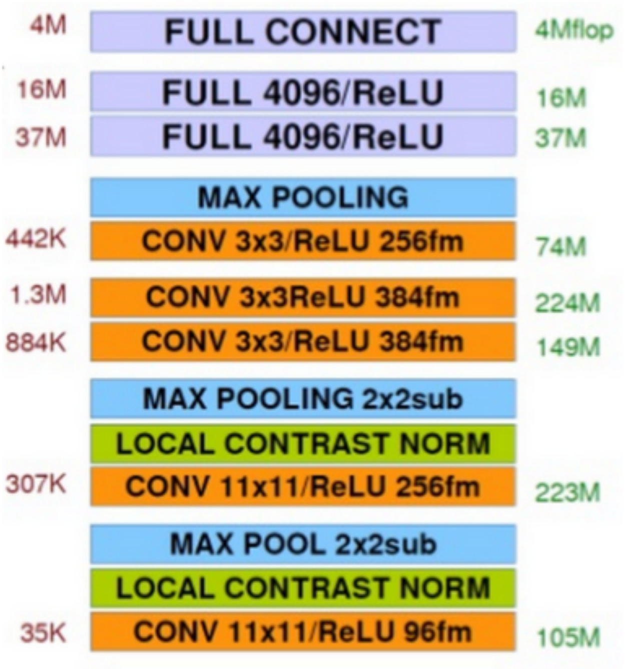


- 0.23% test misclassification on MNIST.
- 1.5 million of parameters

SUPERVISION

Introduced in (Krizhevsky et al., 2012), the “spark” giving birth to the revival of neural networks.

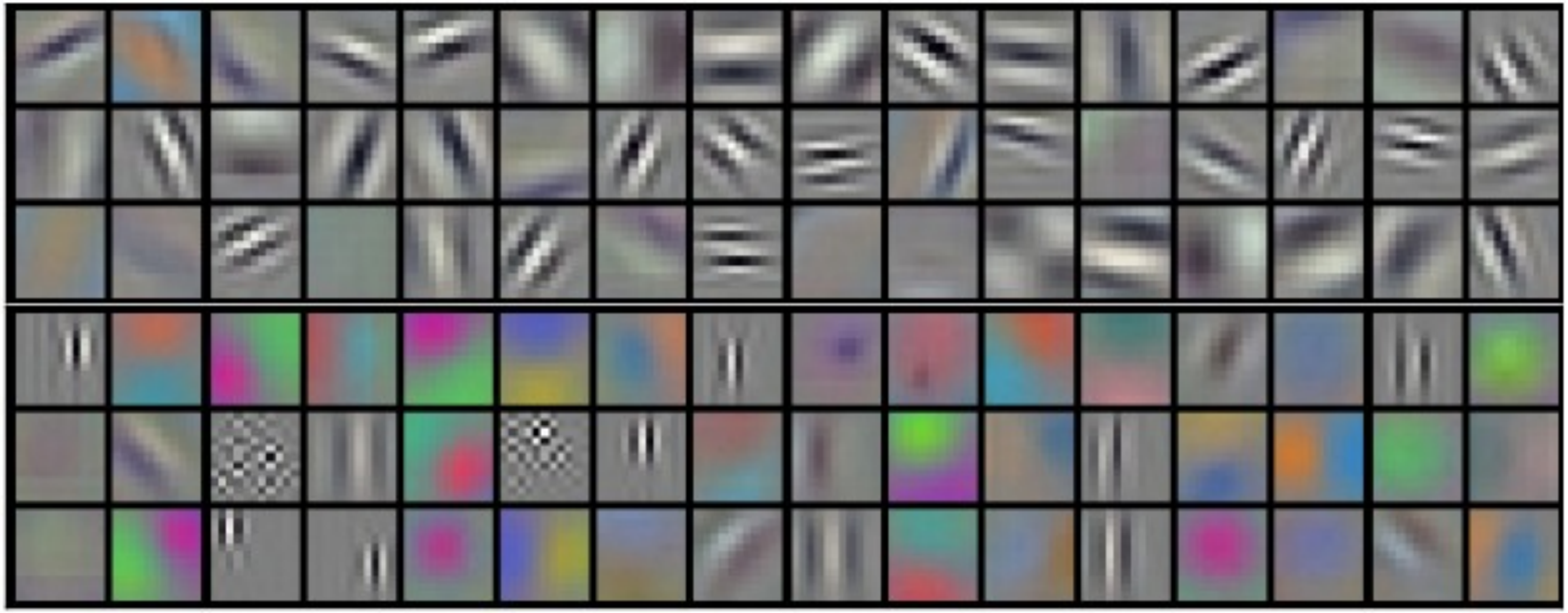




- Top 5 error of 16%, runner-up at 26%
- several convolutions stacked before pooling
- trained on 2 GPUs, for a week on ImageNet (resized to $256 \times 256 \times 3$), 1M images. (now it’s 18 minutes)
- 60 Million parameters, dropout, momentum, L2 penalty, dataset augmentation (rand crop 224×224, translation, reflections, PCA)
- Learning rate at 0.01 divided by 10 when validation error stalls
- at test time, avg probabilities on 5 crops + reflections
- The conv layers are cheap but super important

SUPERVISION

The first layer learned to extract meaningful features

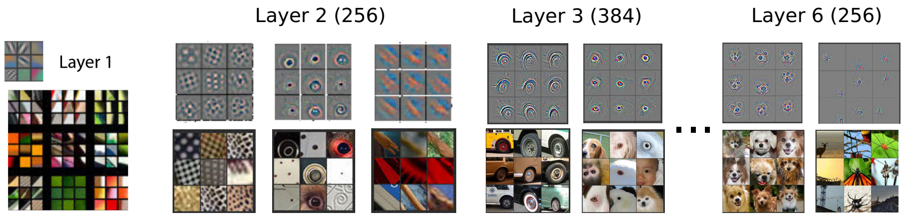


≡

ZFNET

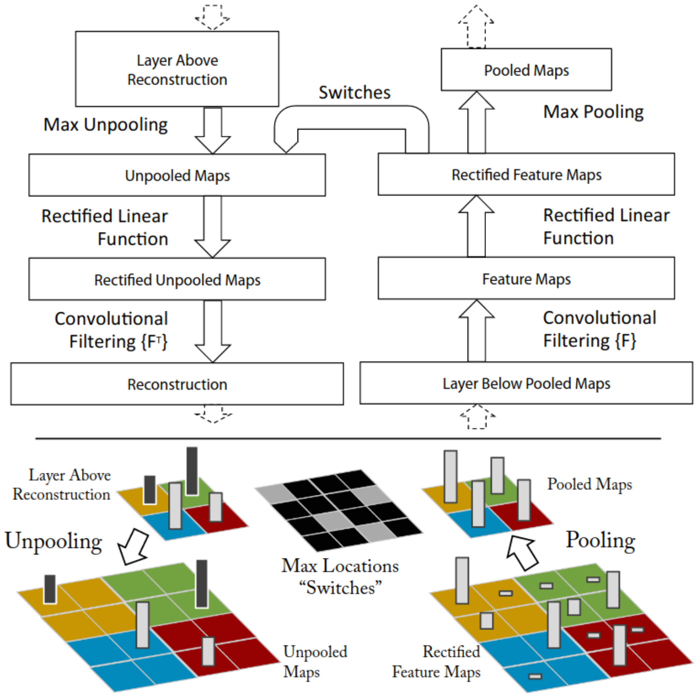
ILSVRC’13 winner. Introduced in (Zeiler & Fergus, 2014)

- Introduced visualization techniques to inspect which features are learned.



Some inputs got by deconvolution

- Ablation studies on AlexNet : the FC layers are not that important
- Introduced the idea of supervised pretraining (pretraining on ImageNet, finetune the softmax for Caltech-101, Caltech-256, Pascal 2012)
- SGD minibatch(128), momentum(0.9), learning rate (0.01) manual schedule,



Deconvnet

Deconvnet computes approximately the gradient of the loss w.r.t. the input (Simonyan, Vedaldi, & Zisserman, 2014). It differs in the way the ReLu is integrated.

VGG

ILSVRC'14 1st runner up. Introduced by (Simonyan & Zisserman, 2015).

- 16 layers : 13 convolutive, 3 fully connected
- Only 3×3 convolution, 2×2 pooling
- **Stacked 3×3 convolutions** $\equiv 5 \times 5$ convolution receptive field with less parameters
  - If $c_{in} = K, c_{out} = K, 5 \times 5$ convolution $\rightarrow 25K^2$ parameters
  - If $c_{in} = K, c_{out} = K$, 2 stacked 3×3 convolution $\rightarrow 18K^2$ parameters
- **140 million parameters**, batch size(256), Momentum(0.9), Weight decay(0.0005), Dropout(0.5) in FC, learning rate(0.01) divided 3 times by 10
- Initialization of B, C, D, E from trained A. Init of A random $\mathcal{N}(0, 10^{-2}), b = 0$. Noticed (Glorot & Bengio, 2010) after submission.
- can cope with **variable input size** changing the FC layers to conv 7×7, conv1×1.

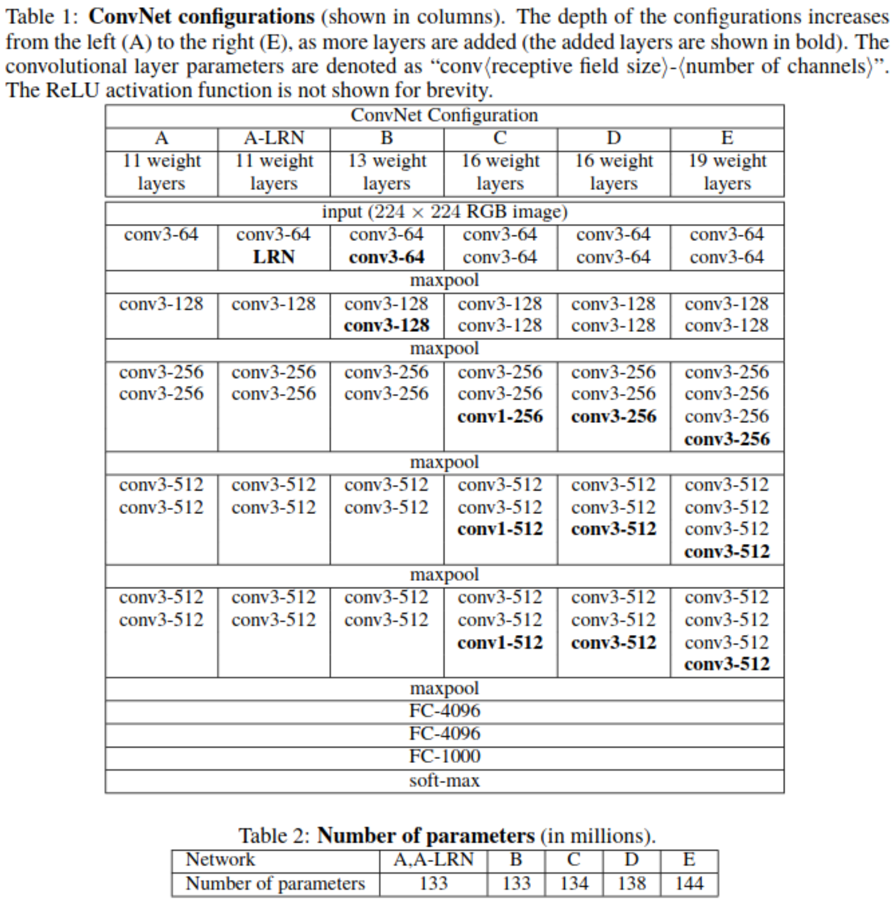


Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 2: **Number of parameters** (in millions).

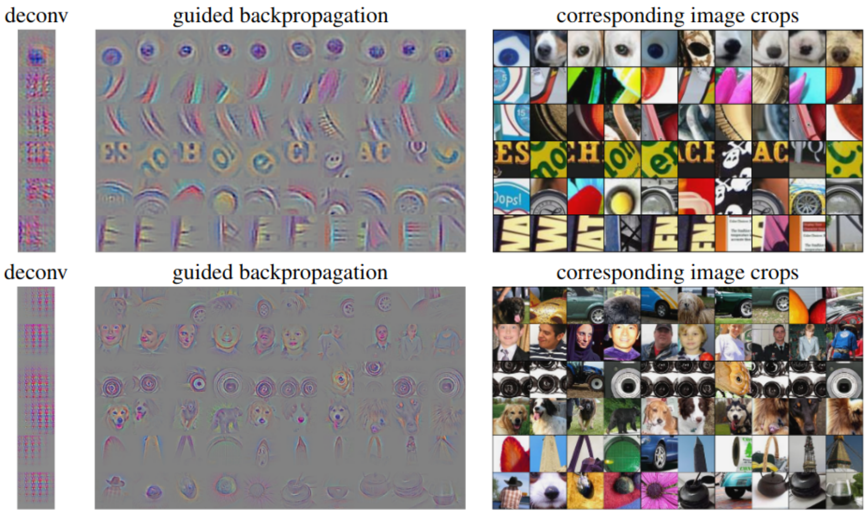
Network	A,A-LRN	B	C	D	E
Number of parameters	133	133	134	138	144

The VGG architectures

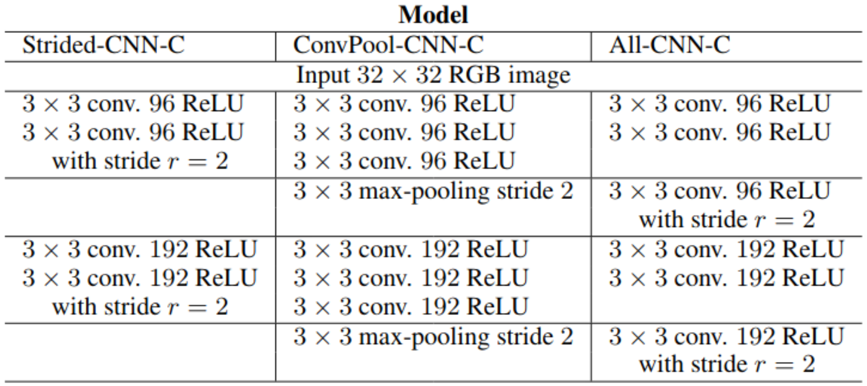
STRIVING FOR SIMPLICITY

Introduced in (Springenberg, Dosovitskiy, Brox, & Riedmiller, 2015).

- uses only convolutions, with various strides, no max pooling
- introduces “guided backpropagation” visualization



Guided backpropagation examples



	Model	
Strided-CNN-C	ConvPool-CNN-C	All-CNN-C
	Input 32×32 RGB image	
3×3 conv. 96 ReLU 3×3 conv. 96 ReLU with stride $r = 2$	3×3 conv. 96 ReLU 3×3 conv. 96 ReLU 3×3 conv. 96 ReLU	3×3 conv. 96 ReLU 3×3 conv. 96 ReLU
	3×3 max-pooling stride 2	3×3 conv. 96 ReLU with stride $r = 2$
3×3 conv. 192 ReLU 3×3 conv. 192 ReLU with stride $r = 2$	3×3 conv. 192 ReLU 3×3 conv. 192 ReLU 3×3 conv. 192 ReLU	3×3 conv. 192 ReLU 3×3 conv. 192 ReLU
	3×3 max-pooling stride 2	3×3 conv. 192 ReLU with stride $r = 2$

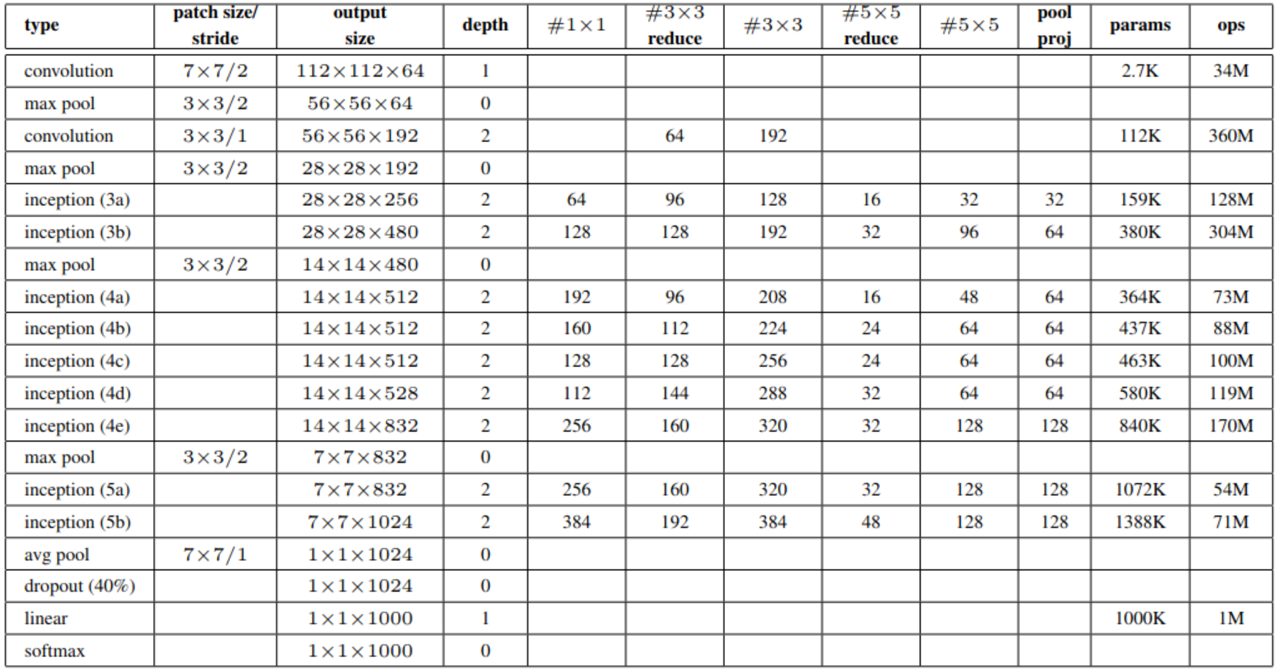
Architectures

GOOGLENET (INCEPTION V1)

ILSVR’14 winner. Introduced by (Szegedy et al., 2014).

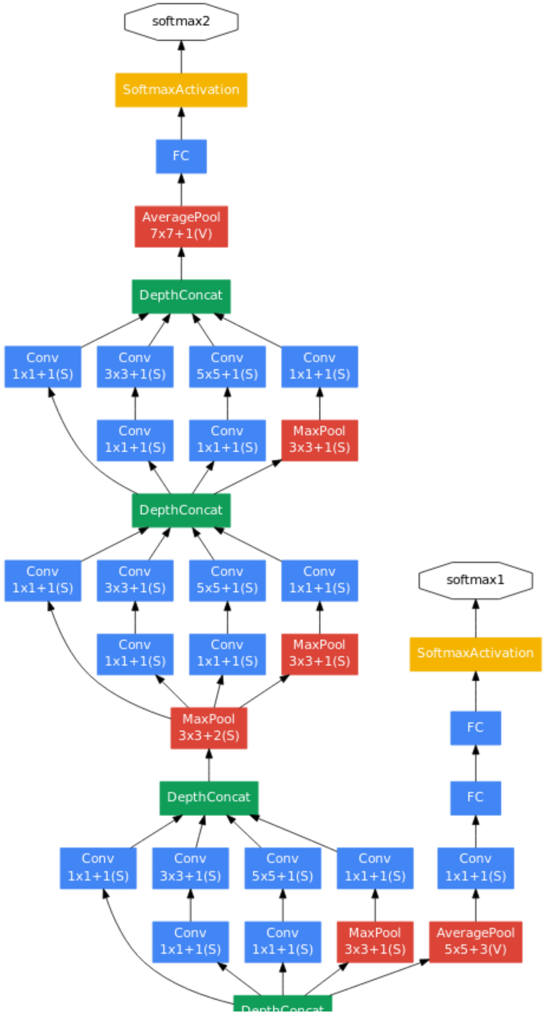
Idea Multi-scale feature detection and dimensionality reduction

- 22 layers, 6.8M parameters
- trained in parallel , asynchronous SGD, momentum(0.9), learning rate schedule (4% every 8 epochs)
- at test : polyak average and ensemble of 7 models
- auxiliary heads to mitigate vanishing gradient



type	patch size/ stride	output size	depth	$\#1\times1$	$\#3\times3$ reduce	$\#3\times3$	$\#5\times5$ reduce	$\#5\times5$	pool proj	params	ops
convolution	$7\times7/2$	$112\times112\times64$	1							2.7K	34M
max pool	$3\times3/2$	$56\times56\times64$	0								
convolution	$3\times3/1$	$56\times56\times192$	2		64	192				112K	360M
max pool	$3\times3/2$	$28\times28\times192$	0								
inception (3a)		$28\times28\times256$	2	64	96	128	16	32	32	159K	128M
inception (3b)		$28\times28\times480$	2	128	128	192	32	96	64	380K	304M
max pool	$3\times3/2$	$14\times14\times480$	0								
inception (4a)		$14\times14\times512$	2	192	96	208	16	48	64	364K	73M
inception (4b)		$14\times14\times512$	2	160	112	224	24	64	64	437K	88M
inception (4c)		$14\times14\times512$	2	128	128	256	24	64	64	463K	100M
inception (4d)		$14\times14\times528$	2	112	144	288	32	64	64	580K	119M
inception (4e)		$14\times14\times832$	2	256	160	320	32	128	128	840K	170M
max pool	$3\times3/2$	$7\times7\times832$	0								
inception (5a)		$7\times7\times832$	2	256	160	320	32	128	128	1072K	54M
inception (5b)		$7\times7\times1024$	2	384	192	384	48	128	128	1388K	71M
avg pool	$7\times7/1$	$1\times1\times1024$	0								
dropout (40%)		$1\times1\times1024$	0								
linear		$1\times1\times1000$	1							1000K	1M
softmax		$1\times1\times1000$	0								

Table 1: GoogLeNet incarnation of the Inception architecture



GoogleNet last layers

RESIDUAL NETWORKS (RESNET)

ILSVRC’15 winner. Introduced in (He et al., 2016a)

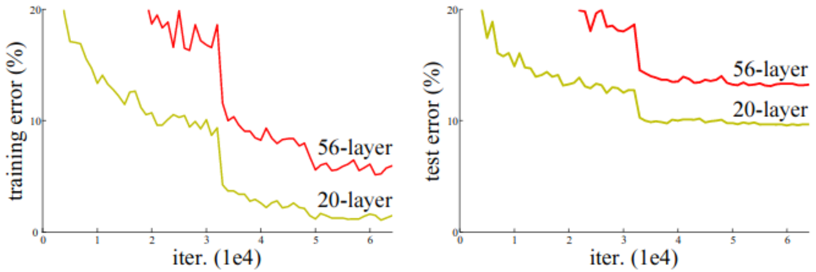
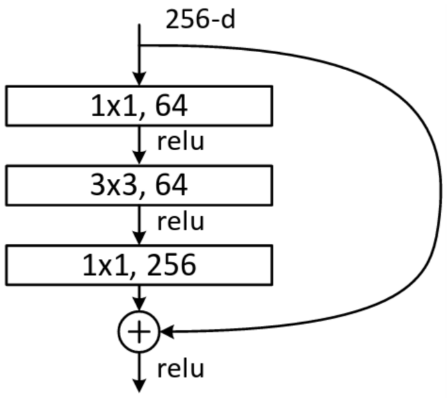


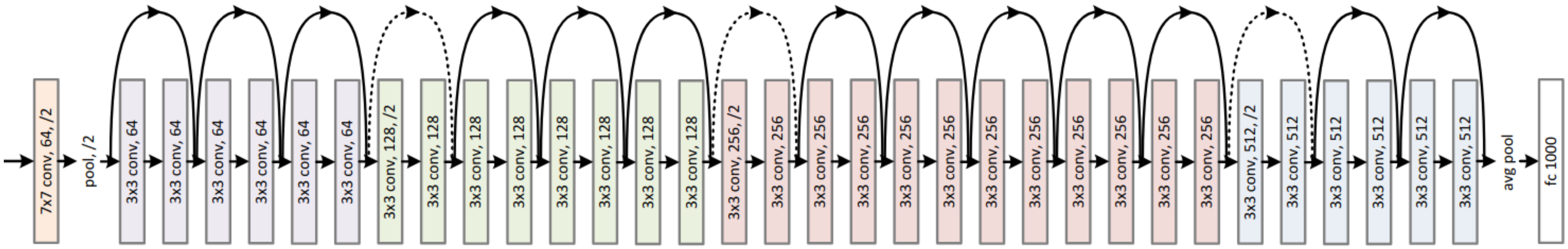
Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena on ImageNet is presented in Fig. 4.

Deeper is worse ?!

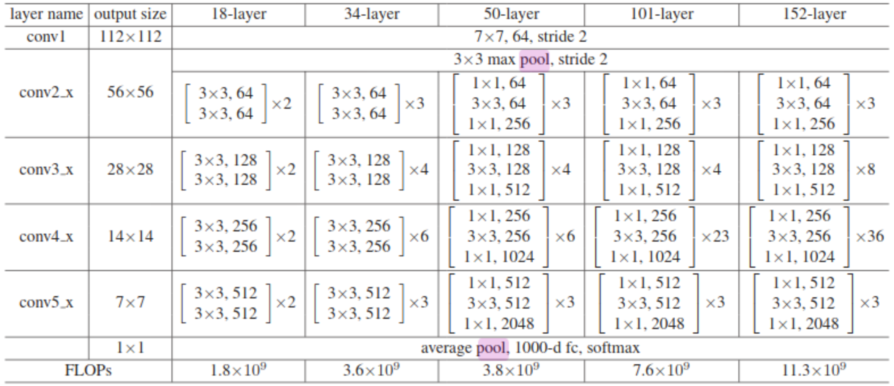


Residual block

RESIDUAL NETWORKS (RESNET)

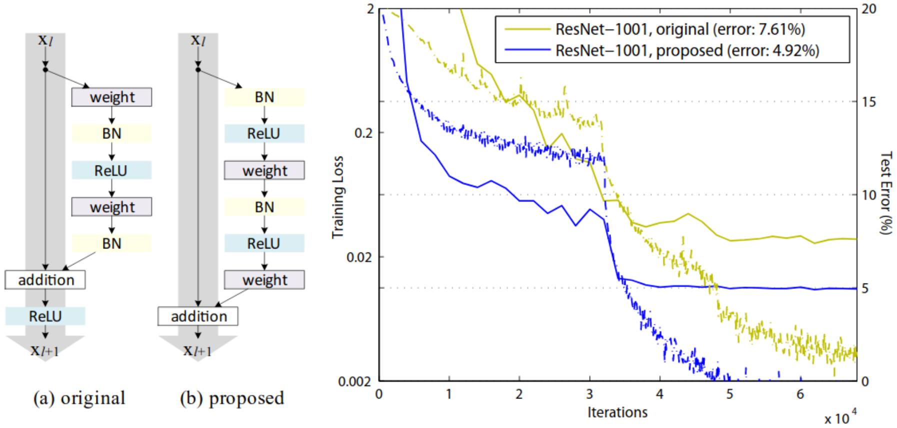


ResNet34. Dotted shortcuts and conv“/2” are stride 2 to match the spatial dimensions. Dotted shortcuts use 1×1 conv to match the depth. 0.46M parameters.



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Resnet architectures. Conv are “Conv-BN-Relu”. ResNet-50 has 23M parameters.



(a) original

(b) proposed

Conv branch variations (He et al., 2016b): BN-ReLu-conv instead of conv-BN-ReLu

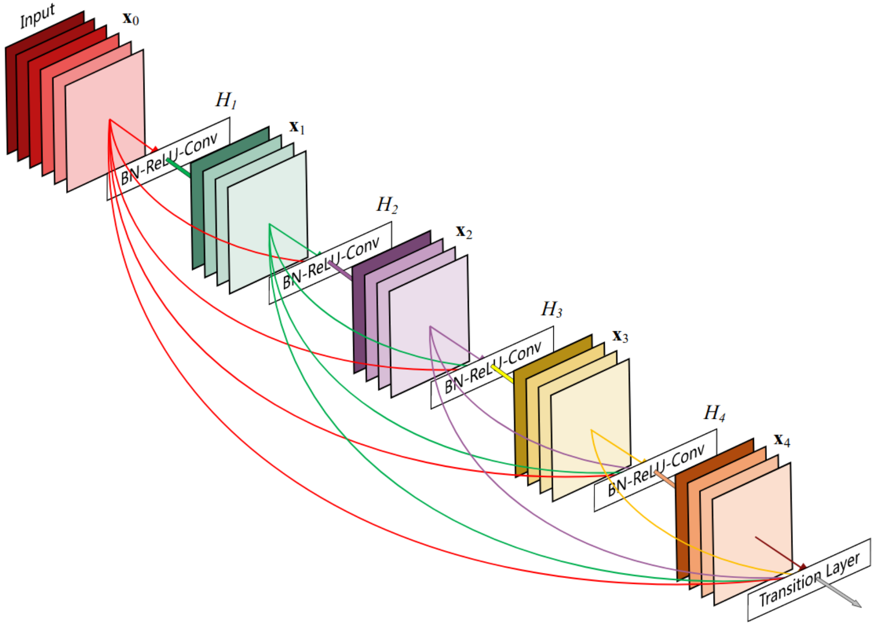
VARIATIONS AROUND SKIP LAYER CONNECTIONS

Highway Networks (Srivastava, Greff, & Schmidhuber, 2015)

- Uses “gates” (as in LSTM, see lectures on RNN) :
  - Transform gate $T(x) = \sigma(W_T x + b_T)$
  - Carry gate $C(x) = \sigma(f_c(x))$

$$y = T(x).\, H(x) + C(x).\, x$$

DenseNets



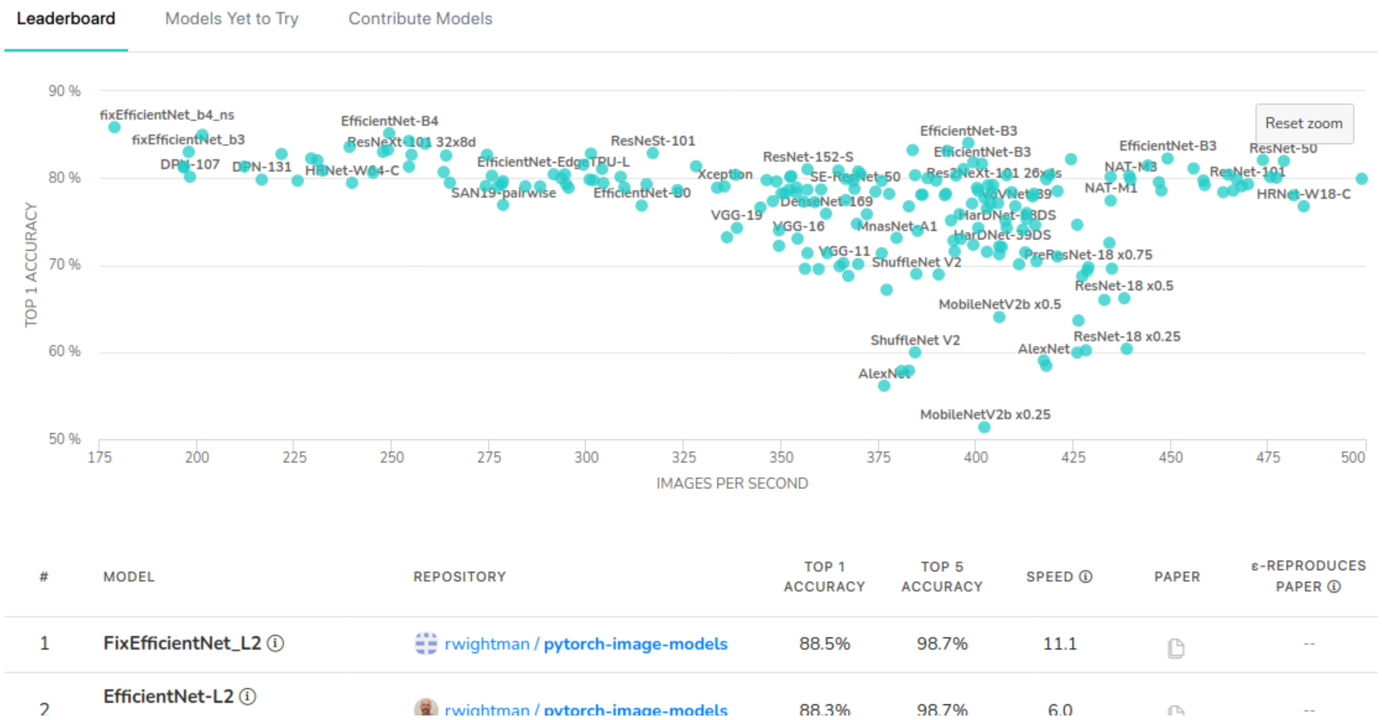
Densenets (Huang, Liu, Maaten, & Weinberger, 2018)

OTHER NETWORKS

Fitnet [Romero(2015)], Wideresnet(2017), Mobilenetv1, v2, v3 [Howard(2019)] : searching for the best architecture, EfficientNet (Tan & Le, 2020)

See also :

- Papers with code
- SOTA bench



#	MODEL	REPOSITORY	TOP 1 ACCURACY	TOP 5 ACCURACY	SPEED ⓘ	PAPER	ε-REPRODUCES PAPER ⓘ
1	FixEfficientNet_L2 ⓘ	rwightman / pytorch-image-models	88.5%	98.7%	11.1		--
2	EfficientNet-L2 ⓘ	rwightman / pytorch-image-models	88.3%	98.7%	6.0		--

CNN DESIGN PRINCIPLES

NUMBER OF FILTERS

You should increase the number of filters throughout the network :

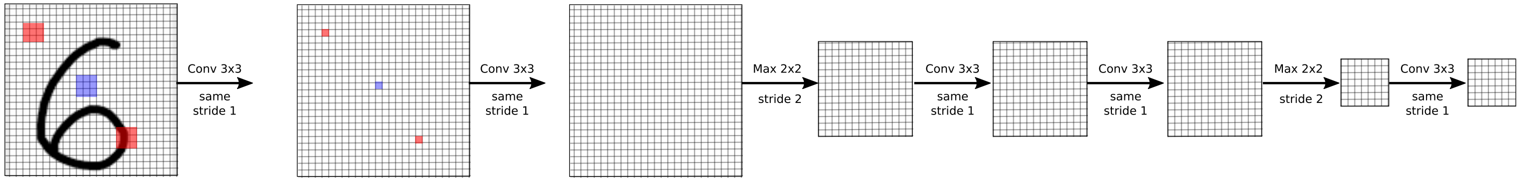
- the first layer extracts low level features
- the higher layers **compose** on the lower layer dictionary of features

Examples :

- LeNet-5 (1998) : $65 \times 5, 165 \times 5$
- AlexNet (2012) : $9611 \times 11, 2565 \times 5, 2 \times (3843 \times 3), 2563 \times 3$
- VGG (2014) : $64 - 128 - 256 - 512$, all 3×3
- ResNet (2015) : $64 - 128 - 256 - 512$, all 3×3
- Inception (2015) : $32 - 64 - 80 - 192 - 288 - 768 - 1280 - 2048, 1 \times 1, 3 \times 3,' 5 \times 5'$

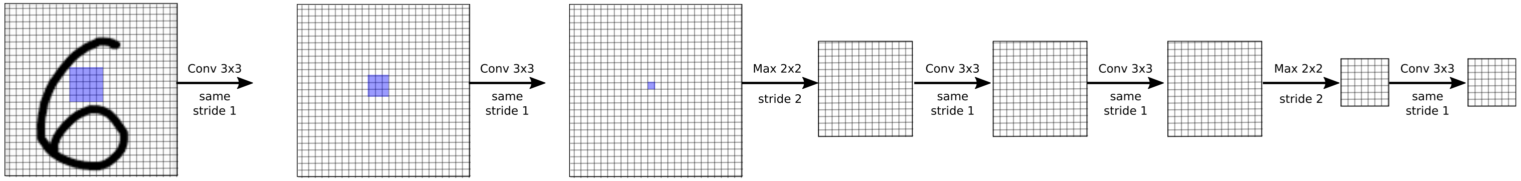
EfficientNet (Tan & Le, 2020) studies the scaling strategies of conv. models.

EFFECTIVE RECEPTIVE FIELD (1/3)



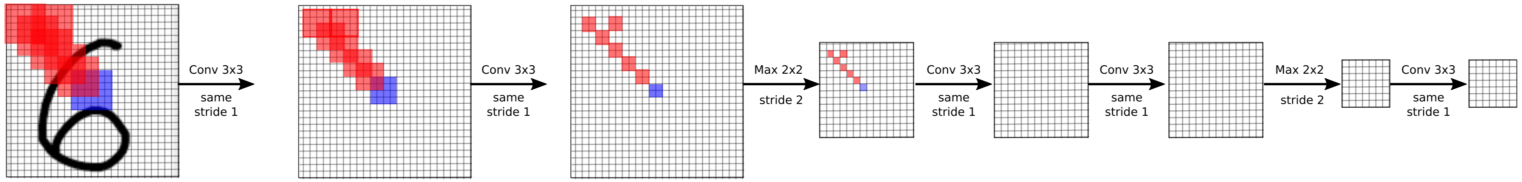
Conv 3x3 same stride 1 → Conv 3x3 same stride 1 → Max 2x2 stride 2 → Conv 3x3 same stride 1 → Conv 3x3 same stride 1 → Max 2x2 stride 2 → Conv 3x3 same stride 1

Representation Size	28x28	28x28
Input RF Size	1x1	3x3



Conv 3x3 same stride 1 → Conv 3x3 same stride 1 → Max 2x2 stride 2 → Conv 3x3 same stride 1 → Conv 3x3 same stride 1 → Max 2x2 stride 2 → Conv 3x3 same stride 1

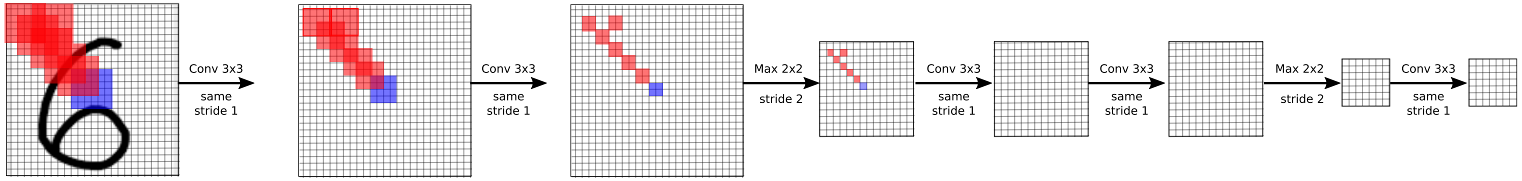
Representation Size	28x28	28x28	28x28
Input RF Size	1x1	3x3	5x5



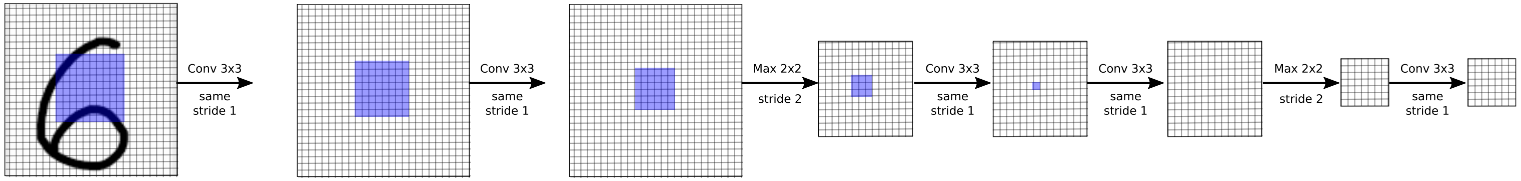
Conv 3x3 same stride 1 → Conv 3x3 same stride 1 → Max 2x2 stride 2 → Conv 3x3 same stride 1 → Conv 3x3 same stride 1 → Max 2x2 stride 2 → Conv 3x3 same stride 1

Representation Size	28x28	28x28	28x28	14x14
Input RF Size	1x1	3x3	5x5	6x6

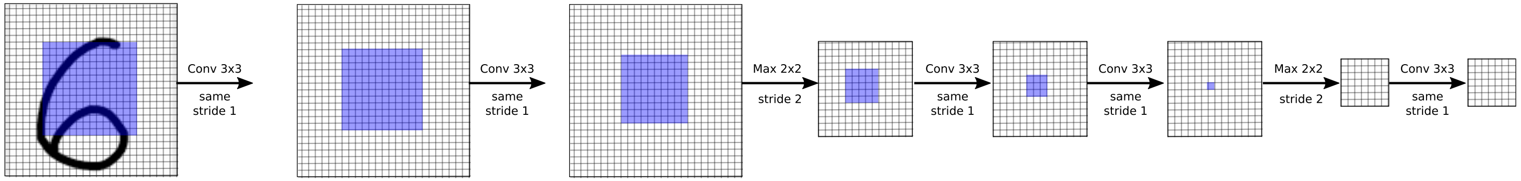
EFFECTIVE RECEPTIVE FIELD (2/3)



Representation Size	28x28	28x28	28x28	14x14
Input RF Size	1x1	3x3	5x5	6x6

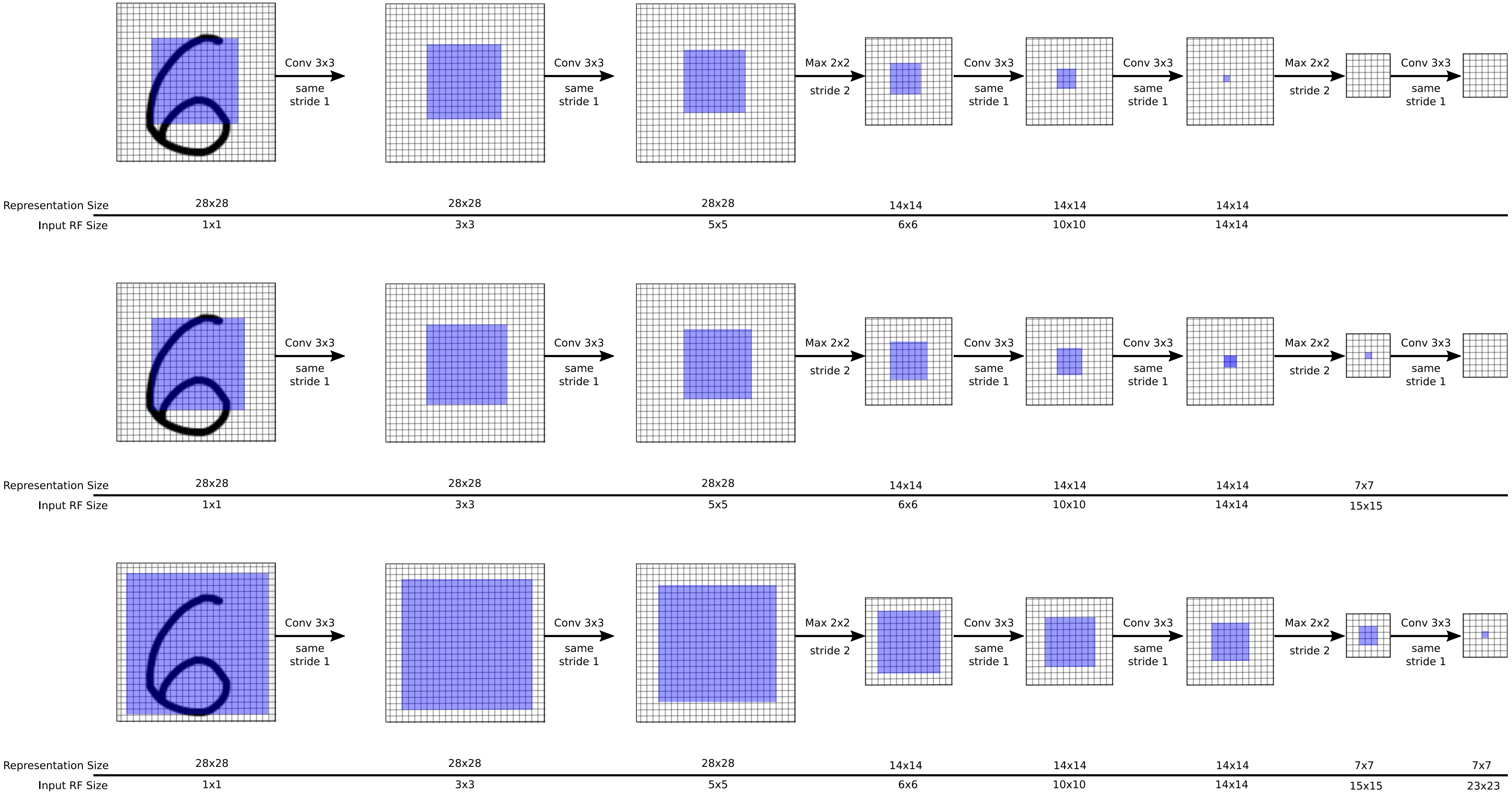


Representation Size	28x28	28x28	28x28	14x14	14x14
Input RF Size	1x1	3x3	5x5	6x6	10x10



Representation Size	28x28	28x28	28x28	14x14	14x14	14x14
Input RF Size	1x1	3x3	5x5	6x6	10x10	14x14

EFFECTIVE RECEPTIVE FIELD (3/3)



Representation Size	28x28	28x28	28x28	14x14	14x14	14x14
Input RF Size	1x1	3x3	5x5	6x6	10x10	14x14

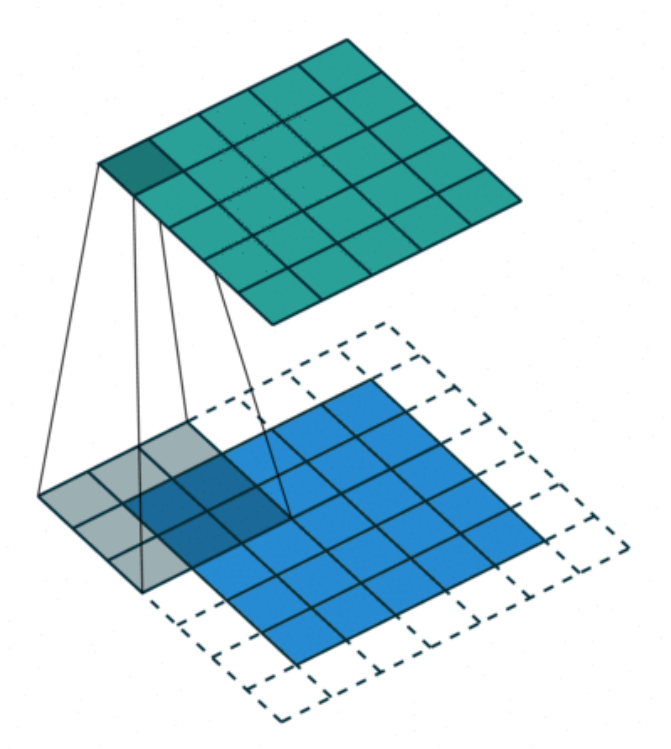
Representation Size	28x28	28x28	28x28	14x14	14x14	14x14	7x7
Input RF Size	1x1	3x3	5x5	6x6	10x10	14x14	15x15

Representation Size	28x28	28x28	28x28	14x14	14x14	14x14	7x7	7x7
Input RF Size	1x1	3x3	5x5	6x6	10x10	14x14	15x15	23x23

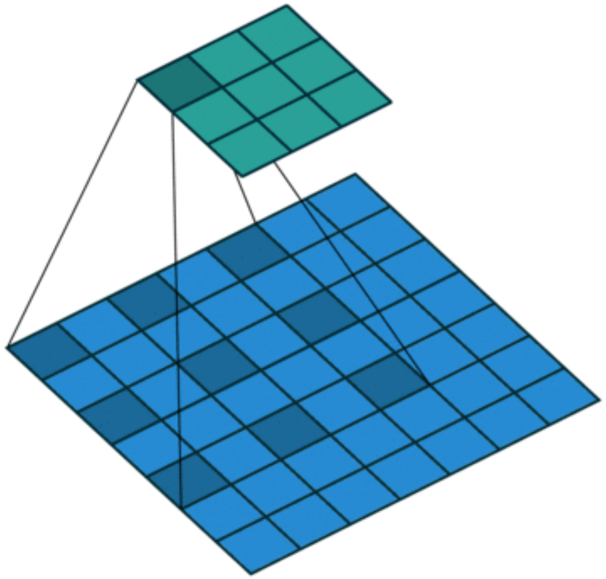
For calculating the effective receptive field size, see this guide on conv arithmetic.

A-TROU CONVOLUTIONS

Your effective receptive field can grow faster with *a-trou* convolutions (or dilated convolutions) (Yu & Koltun, 2016):



Conv 3, Pad 1, Stride 1



Conv 3, Pad 0, Stride 1, Dilated 1

Illustrations from this guide on conv arithmetic. The Conv2D object's constructor accepts a *dilation* argument.

≡

STACKING AND FACTORIZING SMALL KERNELS

Introduced in Inception v3 (Szegedy et al., 2015)

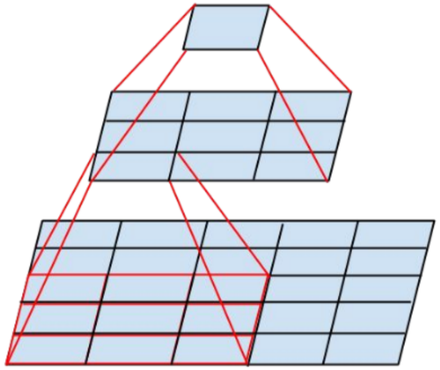


Figure 1. Mini-network replacing the 5×5 convolutions.

Stacking $2(3 \times 3)$ conv

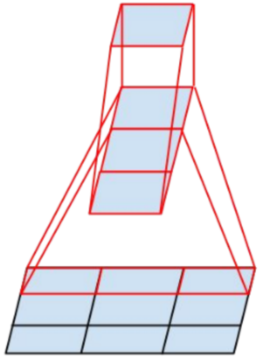


Figure 3. Mini-network replacing the 3×3 convolutions. The lower layer of this network consists of a 3×1 convolution with 3 output units.

1×3 and 3×1 conv

n input filters,αn output filters :

- $(\alpha n, 5 \times 5)$ conv : $25\alpha n^2$ params
- $(\sqrt{\alpha}n, 3 \times 3)$- $(\alpha n, 3 \times 3)$: $9\sqrt{\alpha}n^2 + 9\sqrt{\alpha}\alpha n^2$ params;

$\alpha = 2 \Rightarrow -24\%$ ($\sqrt{\alpha}$ is critical!)

n input filters,αn output filters :

- $(\alpha n, 3 \times 3)$ conv : $9\alpha n^2$ params
- $(\sqrt{\alpha}n, 1 \times 3)$ - $(\alpha n, 3 \times 1)$: $3\sqrt{\alpha}n^2 + 3\alpha\sqrt{\alpha}n^2$ params

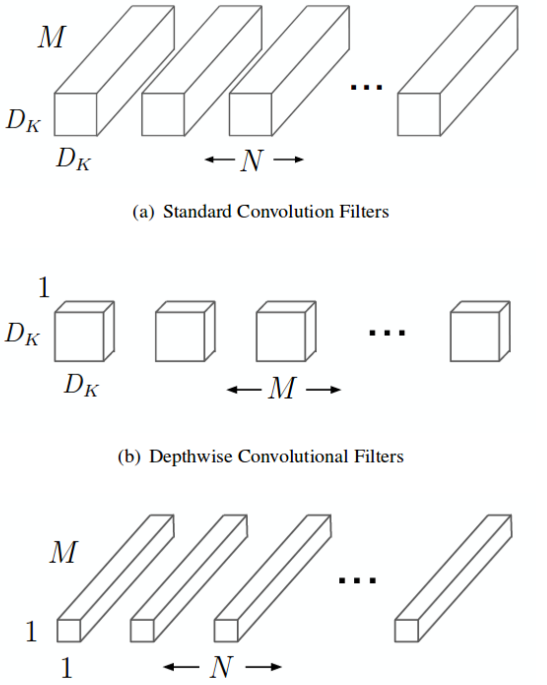
$\alpha = 2 \Rightarrow -30\%$

See also the recent work on “Rethinking Model scaling for convolutional neural networks” (Tan & Le, 2020)

DEPTHWISE SEPARABABLE CONVOLUTIONS

Inception and Xception, Mobilnets. It separates :

- feature extraction in each channel, in space : depthwise convolution
- feature combination between channels : pointwise convolution 1×1


M
D_K
D_K
$\leftarrow N \rightarrow$
1
D_K
D_K
$\leftarrow M \rightarrow$
M
1
1
$\leftarrow N \rightarrow$

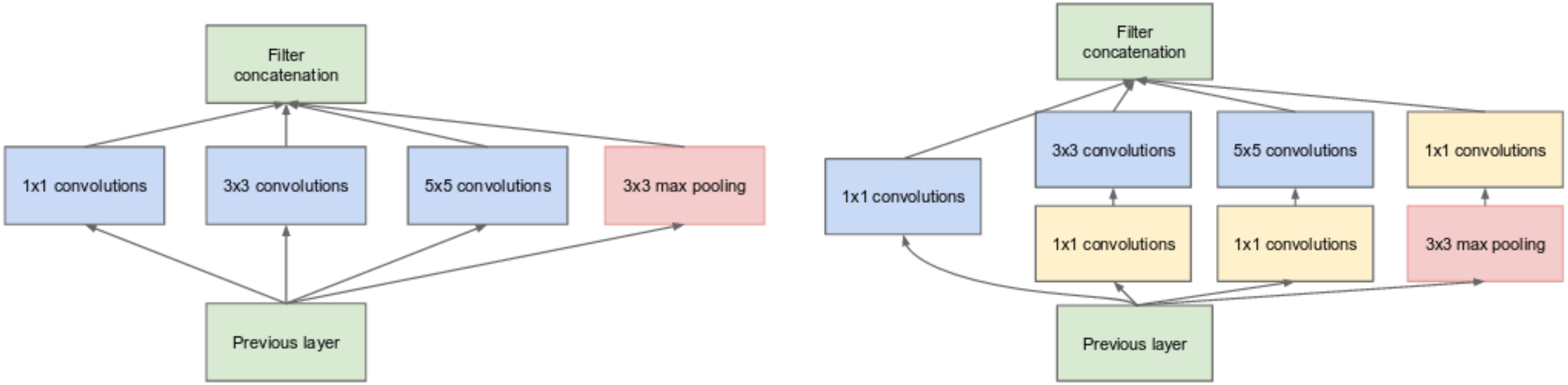
(a) Standard Convolution Filters

(b) Depthwise Convolutional Filters

(c) 1×1 Convolutional Filters called Pointwise Convolution in the context of Depthwise Separable Convolution

Depthwise and pointwise convolutions (Howard et al., 2017)

MULTI-SCALE FEATURE EXTRACTION



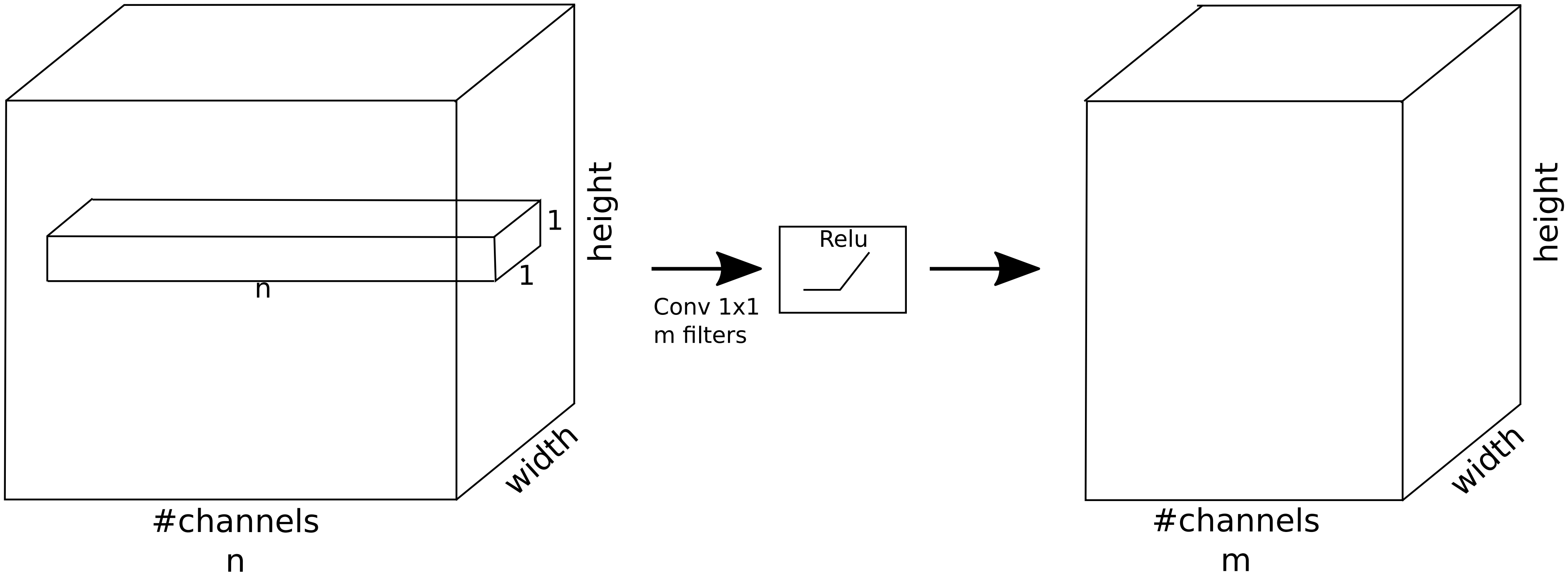
(a) Inception module, naïve version

(b) Inception module with dimension reductions

Extract features at multiple scales

See also the Feature Pyramid Networks for multi-scale features.

DIMENSIONALITY REDUCTION



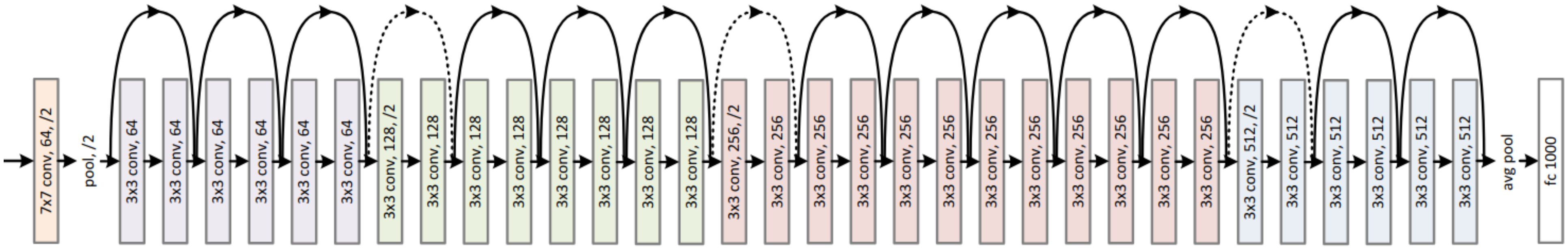
Dimensionality reduction with 1×1 conv

Trainable non-linear transformation of the channels. Network in network (Lin, Chen, & Yan, 2014)

EASING THE GRADIENT FLOW

You can check the norm of the gradient w.r.t. the first layers' parameters to diagnose **vanishing gradients**

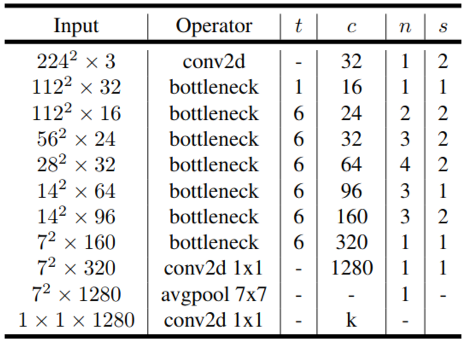
- Shortcut connections (e.g. ResNet, DenseNet, Highway)



- auxiliary heads (e.g. GoogleNet)

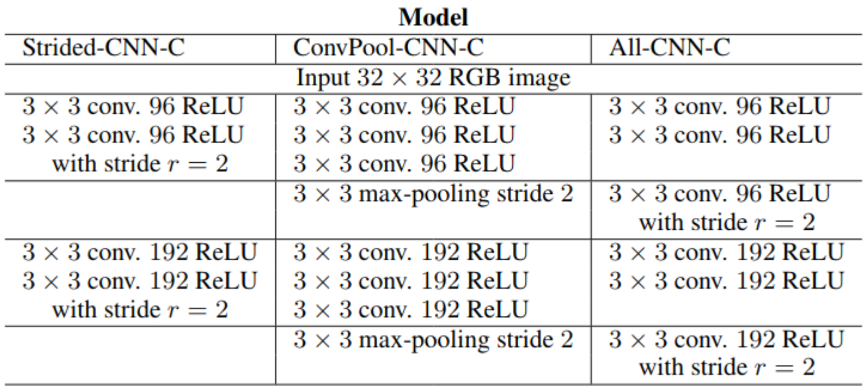
DO WE NEED MAX POOLING ?

Recent architectures remove the max pooling layers and replace them by conv(stride=2) for downsampling



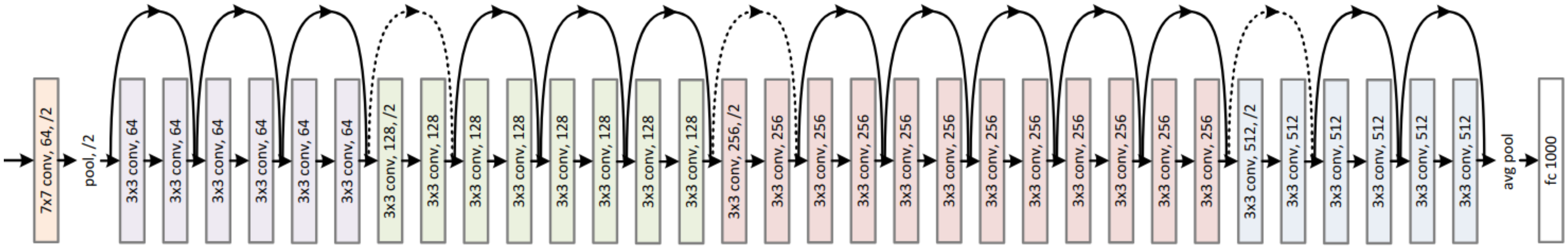
Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

MobileNetv2 (Sandler, Howard, Zhu, Zhmoginov, & Chen, 2018). Bottleneck used also in EfficientNet(2019)



	Model	
Strided-CNN-C	ConvPool-CNN-C	All-CNN-C
	Input 32×32 RGB image	
3×3 conv. 96 ReLU 3×3 conv. 96 ReLU with stride $r = 2$	3×3 conv. 96 ReLU 3×3 conv. 96 ReLU 3×3 conv. 96 ReLU	3×3 conv. 96 ReLU 3×3 conv. 96 ReLU
	3×3 max-pooling stride 2	3×3 conv. 96 ReLU with stride $r = 2$
3×3 conv. 192 ReLU 3×3 conv. 192 ReLU with stride $r = 2$	3×3 conv. 192 ReLU 3×3 conv. 192 ReLU 3×3 conv. 192 ReLU	3×3 conv. 192 ReLU 3×3 conv. 192 ReLU
	3×3 max-pooling stride 2	3×3 conv. 192 ReLU with stride $r = 2$

Striving for simplicity (Springenberg et al., 2015)

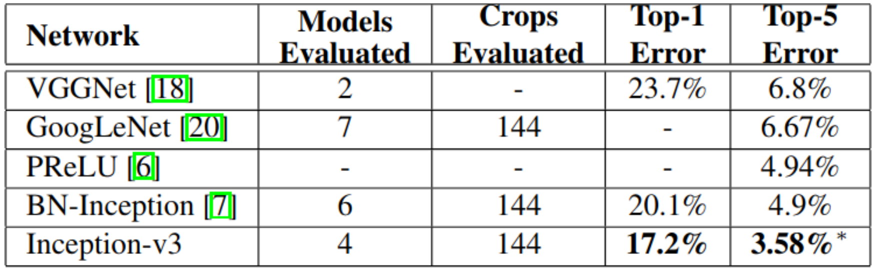


ResNet

MODEL AND WEIGHT AVERAGING

All the competitors in ImageNet do perform model averaging.

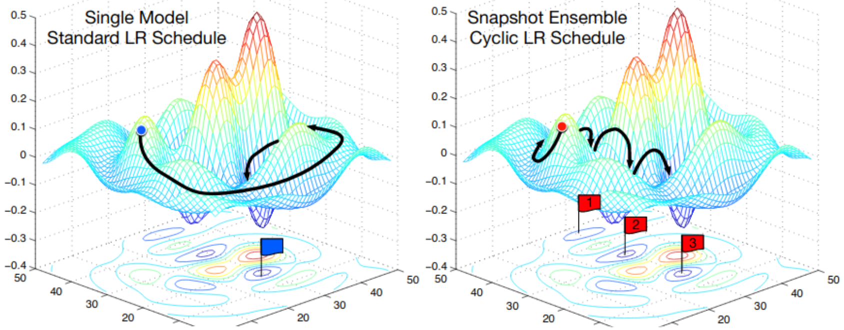
Model averaging



Network	Models Evaluated	Crops Evaluated	Top-1 Error	Top-5 Error
VGGNet [18]	2	-	23.7%	6.8%
GoogLeNet [20]	7	144	-	6.67%
PReLU [6]	-	-	-	4.94%
BN-Inception [7]	6	144	20.1%	4.9%
Inception-v3	4	144	**17.2%**	**3.58%***

Model averaging performance on ImageNet'12 with multiple models and multiple crops-scale-flips

Weight averaging



Snapshot ensembles (Huang et al., 2017)

If you worry about the increased computational complexity, see knowledge distillation (Hinton, Vinyals, & Dean, 2015) : training a light model with the soft targets (vs. the labels, i.e. the hard targets) of a computationally intensive one.

WE NEED DATA !

USING PRE-TRAINED MODELS

All the frameworks provide you with a **model zoo** of pre-trained networks. E.g. in PyTorch, for image classification. You can cut the head and finetune the softmax only.

```
import torchvision.models as models

resnet18 = models.resnet18(pretrained=True)
alexnet = models.alexnet(pretrained=True)
vgg16 = models.vgg16(pretrained=True)
inception = models.inception_v3(pretrained=True)
googlenet = models.googlenet(pretrained=True)
```

⚠ Do not forget the input normalization !

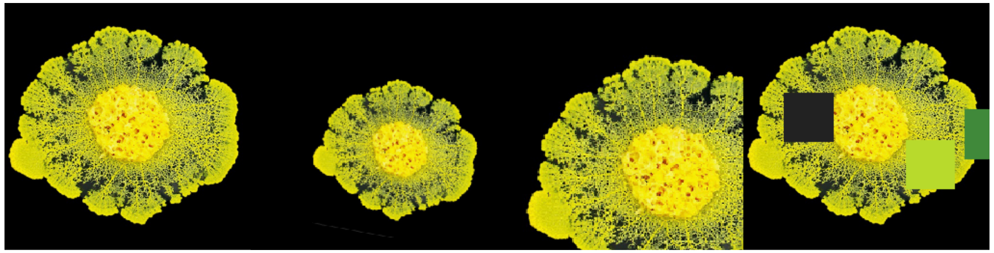
```
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225])
```

Have a look in the torchvision doc, there are pretrained for classification, detection, segmentation … See also pytorch hub and timm for very up to date image models.

DATASET AUGMENTATION

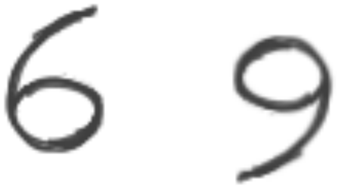
You can oversample around your training samples by applying transforms on the inputs that make predictable changes on the targets.

- color jittering, translations, reflections, rotations, PCA, …



Some images generated with imgaug. All *physarum polycephalum*, right ? Source image from the CNRS

Libraries for augmentation : albumentations, imgaug



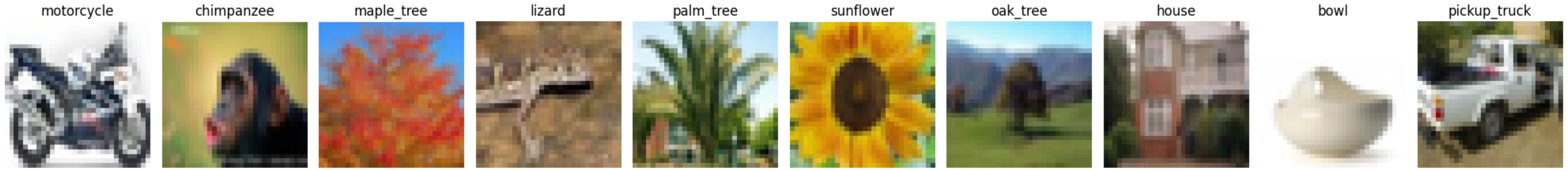
⚠ Your augmentation transforms must be well calibrated : you must be able to predict the change in label given the change of input !

See also mixup: Beyond empirical risk minimization

EXAMPLE CNN

CIFAR-100 DATASET

- The CIFAR-100 dataset is made of 100 classes with 600 images per class.
- The images are 32×32 RGB



Extract from CIFAR-100

- The training set has 500×100 images, and the test set has 100×100 images.

MODEL ARCHITECTURE AND OPTIMIZATION SETUP

Operator	Resolution	RF size	#Channels
ConvBlock	32×32	5×5	32
ConvBlock	32×32	9×9	32
Sub	16×16	15×15	32
ConvBlock	16×16	15×15	128
ConvBlock	16×16	23×23	128
Sub	8×8	31×31	128
AvgPool	1×1		128
Linear	100		

ConvBlock: 2x [Conv(1x3)-(BN)-Relu-Conv(3x1)-(BN)-Relu]
Sub : Conv(3x3, stride=2)-(BN)-Relu

Number of parameters: $\simeq 2M$
Time per epoch (1080Ti) : 17s. , 42min training time

If applied, only the weights of the convolution and linear layers are regularized (not the bias, nor the coefficients of the Batch Norm)

Common settings :

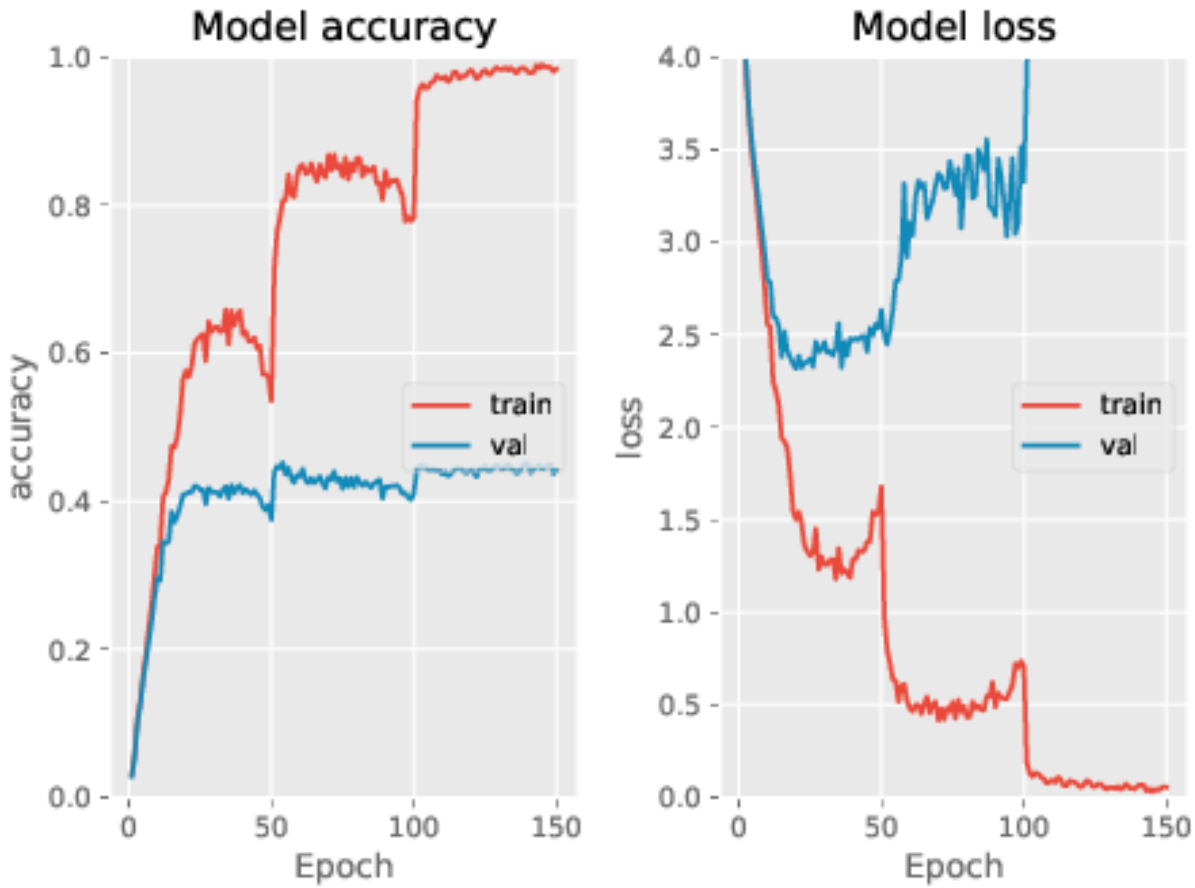
- BatchSize(32),
- SGD(lrate=0.01) with momentum(0.9)
- learning rate halved every 50 epochs
- validation on 20%, early stopping on the val loss

Different configurations :

- base
- Conv-BN-Relu or Conv-Relu
- dataset augmentation (HFlip, Trans(5pix), Scale(0.8,1.2)), CenterCrop(32)
- Dropout, L2, label smoothing

BASELINE

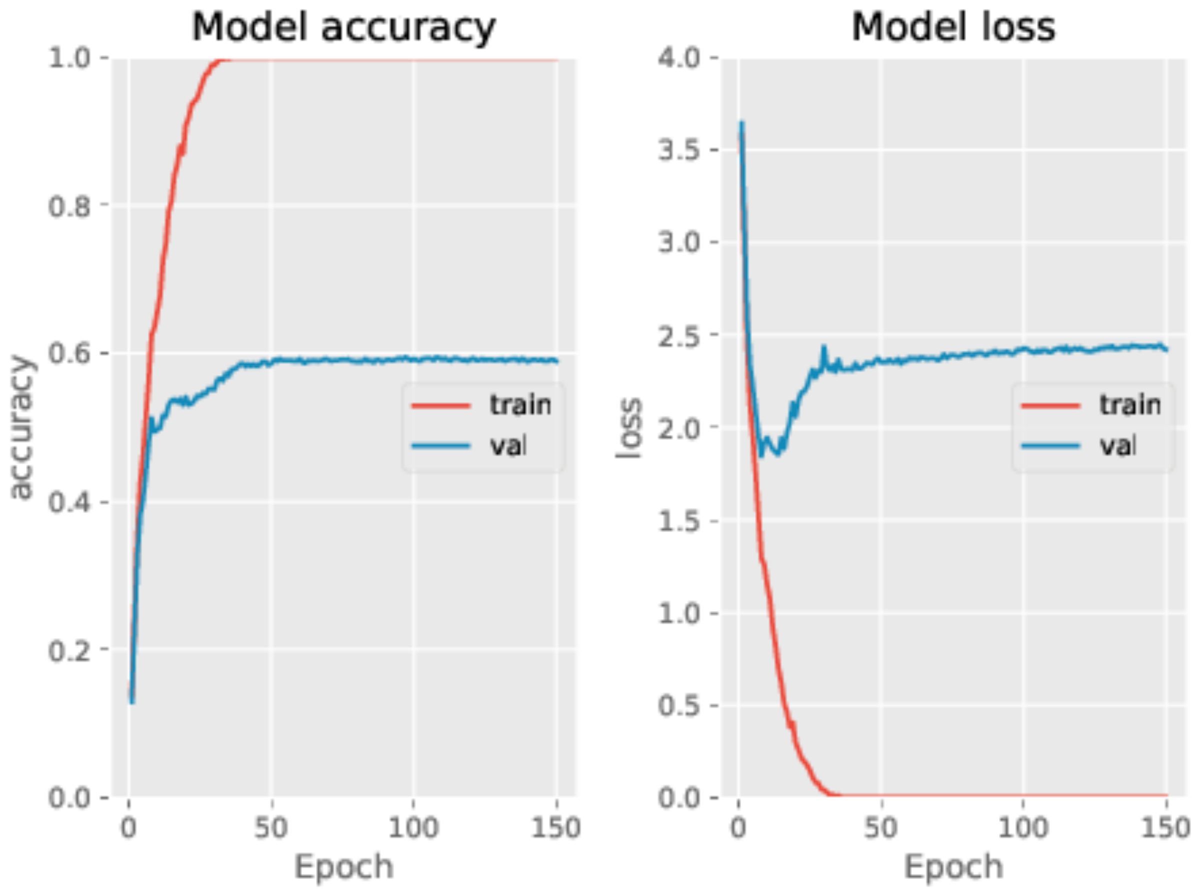
No regularization (either L2, Dropout, Label smoothing, data augmentation), No BatchNorm



≡

WITH BATCHNORM

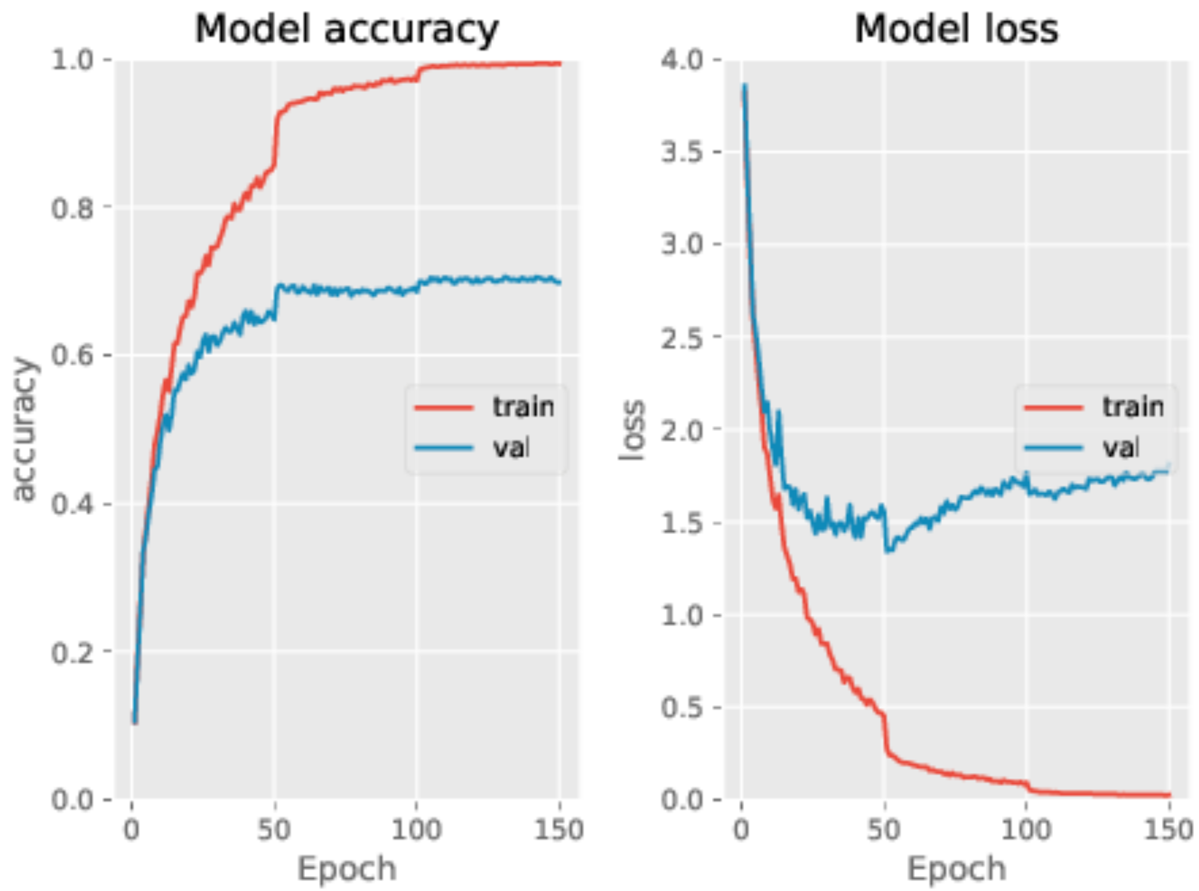
With batchnorm after every convolution (Note it is also regularizing the network)



≡

WITH DATA AUGMENTATION

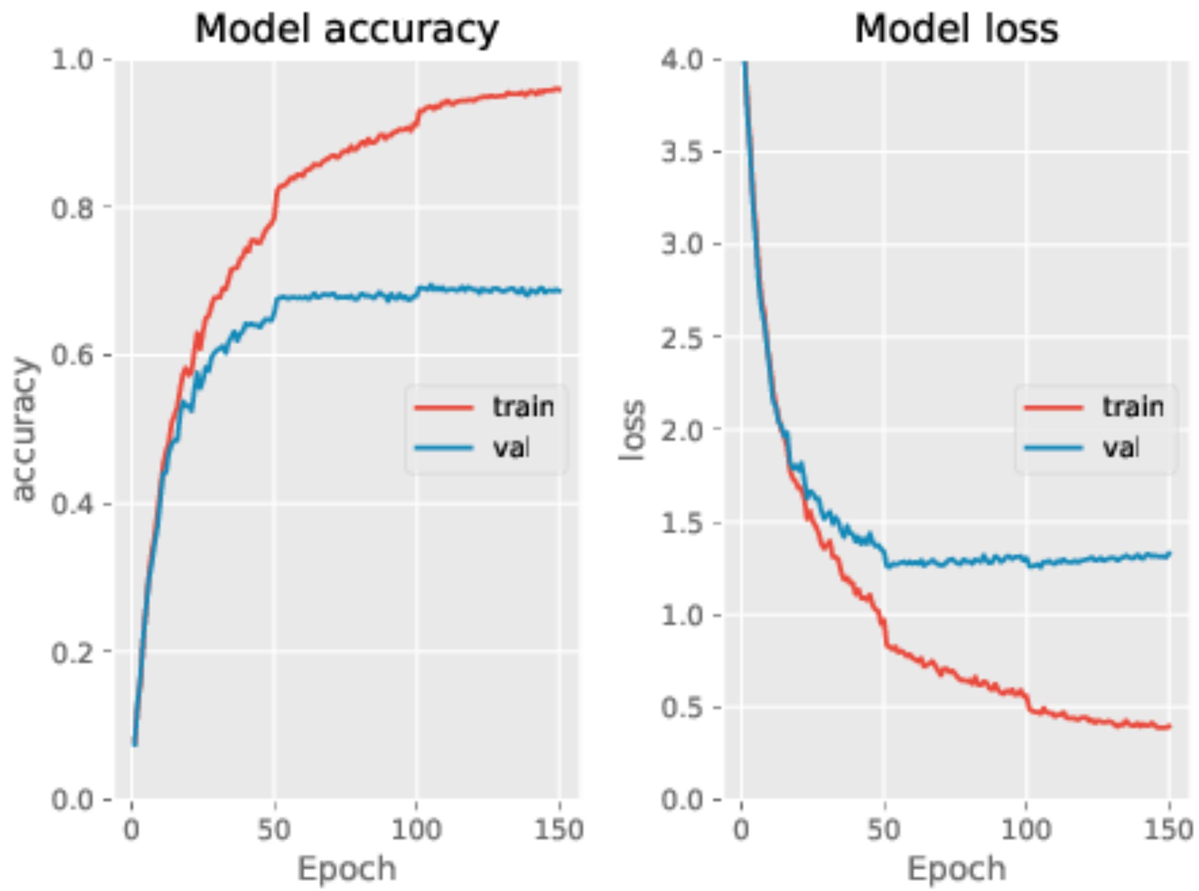
With dataset augmentation (HFlip, Scale, Trans)



≡

WITH REGULARIZATION

With regularization : L2 (0.0025), Dropout(0.5), Label smoothing(0.1)



≡

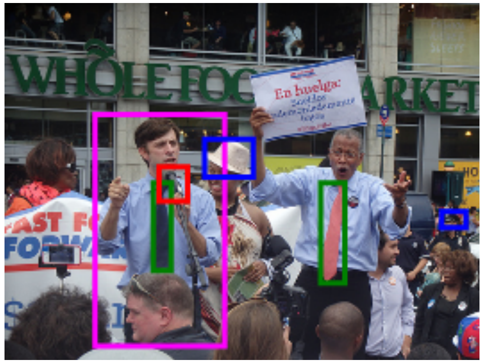
OBJECT DETECTION : INTRODUCTION

PROBLEM STATEMENT

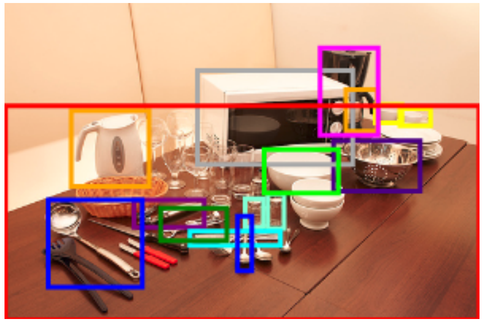
Given :

- images x_i,
- targets y_i which contains objects bounding boxes and labels

Examples from ImageNet (see here)



ILSVRC2014_train_00005559 : few objects annotated.



ILSVRC2014_train_00029372 : 12 objets with occlusions

Bounding boxes given, **in the datasets** (the predictor parametrization may differ), by : $[x, y, w, h]$, $[x_{min}, y_{min}, x_{max}, y_{max}]$, $\dots$

Datasets : Coco, ImageNet, Open Images Dataset

Recent survey : Object detection in 20 years: a survey

Open image evaluation:

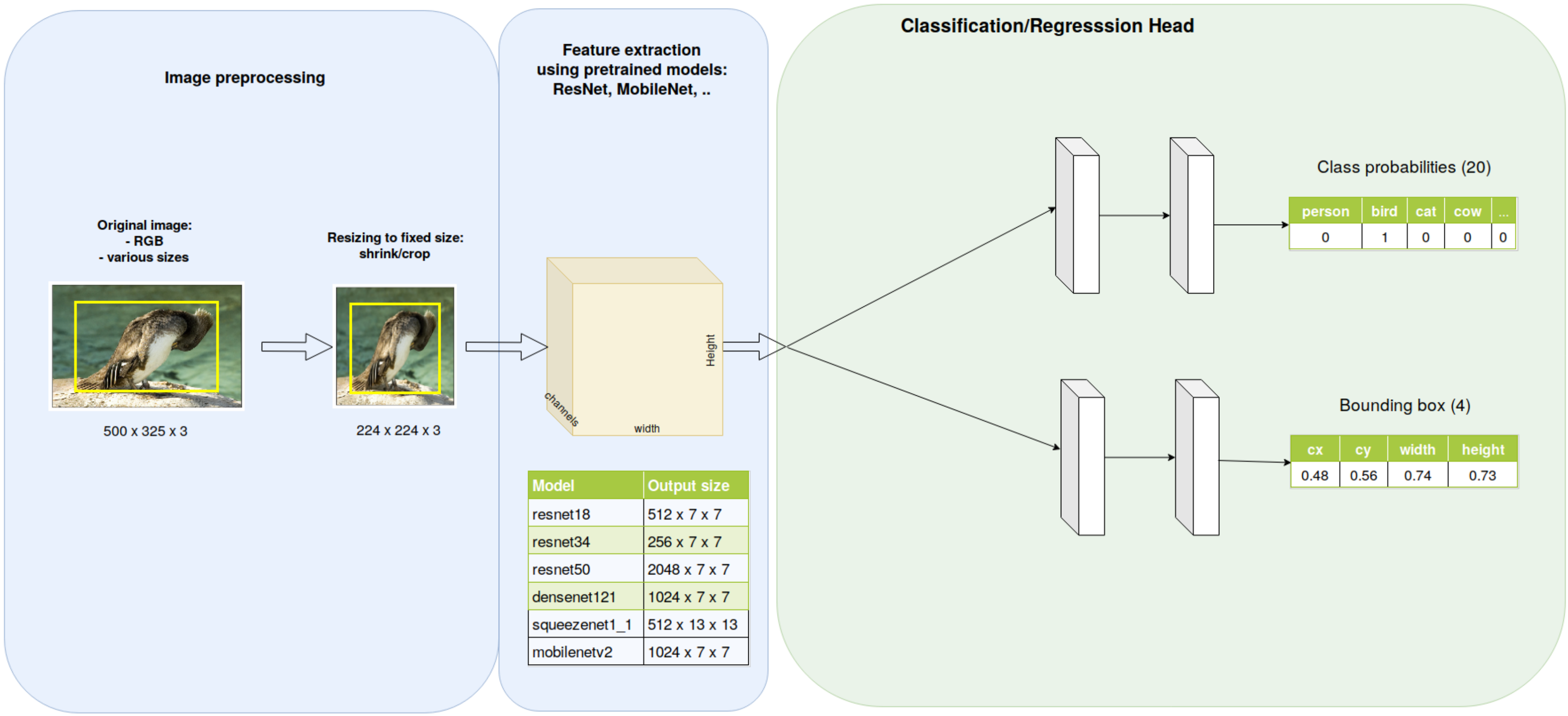
- uses a variant of VOC 2010. more details here

OBJECT LOCALIZATION

A FIRST STEP: OBJECT LOCALIZATION

Suppose you have a single object to detect, can you localize it into the image ?

Single object detection : Classification and Bounding box regression



Model	Output size
resnet18	512 x 7 x 7
resnet34	256 x 7 x 7
resnet50	2048 x 7 x 7
densenet121	1024 x 7 x 7
squeezenet1_1	512 x 13 x 13
mobilenetv2	1024 x 7 x 7

person	bird	cat	cow	...
0	1	0	0	0

cx	cy	width	height
0.48	0.56	0.74	0.73

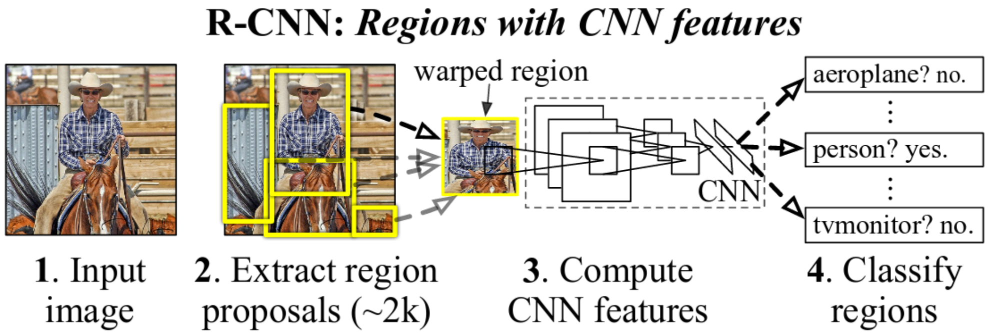
OBJECT DETECTION : STATE OF THE ART

RCNN

How can we proceed with multiple objects ? (Girshick, Donahue, Darrell, & Malik, 2014) proposed to :

- use **selective search** for proposing bounding boxes
- to classify with a SVM from the features extracted by a **pretrained** DNN.
- to optimize localization with linear bbox adaptors

Revolution in the object detection community (vs. “traditional” HOG like features).



Drawback :

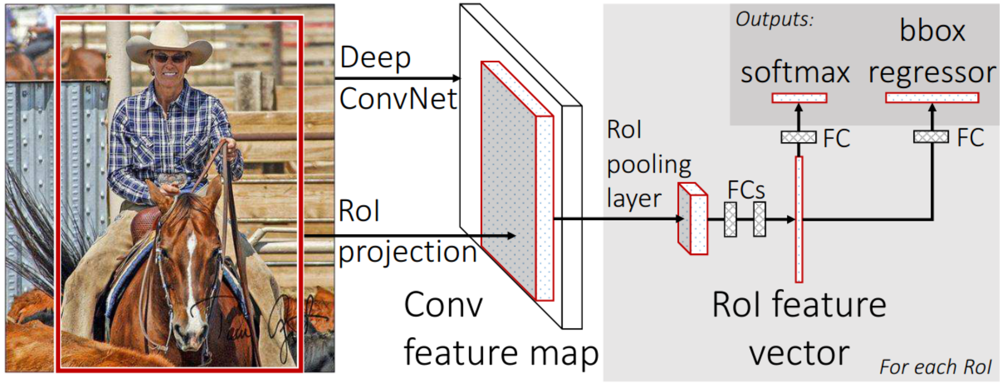
- external algorithm (not in the computational graph, not trainable)
- one forward pass per bounding box proposal ($\sim 2K$ or so) $\rightarrow$ training and test are slow ($47s.$ per image with VGG16)

Notes : pretained on ImageNet, finetuned on the considered classes with warped images. Hard negative mining (boosting).

FAST RCNN

Introduced in (Girshick, 2015). Idea:

- just one forward pass
- cropping the convolutional feature maps (e.g. $(1, 512, H/16, W/16)$ conv5 of VGG16)
- max-pool the **variable sized** crop to a **fixed-sized** (e.g. 7×7) vector before dense layers: **ROI pooling**



Drawbacks:

- external algorithm for ROI proposals: not trainable and slow
- ROIs are snapped to the grid (see here) $\longrightarrow$ ROI align

Github repository. CVPR'15 slides

Notes : pretrained VGG16 on ImageNet. Fast training with multiple ROIs per image to build the 128 mini batch from $N = 2$ images, using 64 proposals : 25% with IoU>0.5 and 75% with $IoU \in [0.1, 0.5[$. Data augmentation : horizontal flip. Per layer learning rate, SGD with momentum, etc..

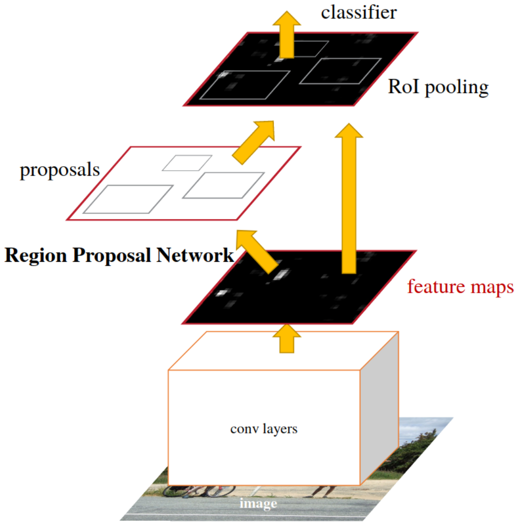
Multi task loss :

$$L(p, u, t, v) = -\log(p_u) + \lambda \text{smooth L1}(t, v)$$

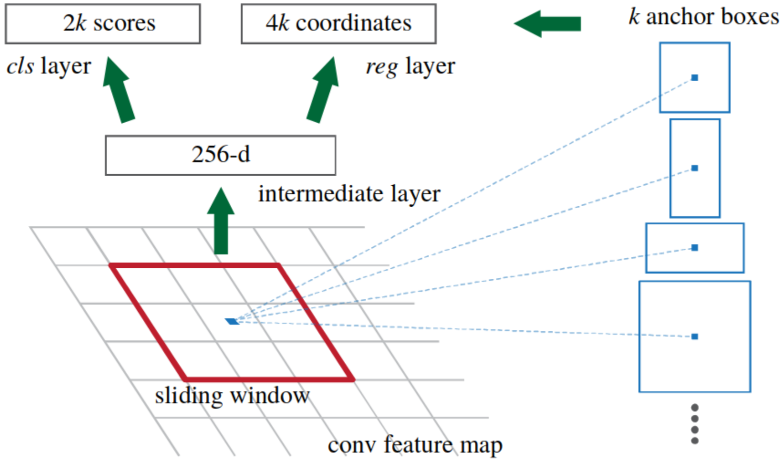
The bbox is parameterized as in (Girshick et al., 2014). Single scale is more efficient than multi-scale.

FASTER RCNN : 2-STAGES TRAINED END-TO-END

Introduced in (Ren, He, Girshick, & Sun, 2016). The first **end-to-end trainable** network. Introducing the **Region Proposal Network** (RPN). A RPN is a sliding Conv(3×3) - Conv(1×1, k + 4k) network (see here). It also introduces anchor boxes of predefined aspect ratios learned by vector quantization.



Architecture



Region proposal network with anchors at 3 scales, 3 aspect ratios

Check the paper for a lot of quantitative results. Small objects may not have a lot of features.

Github repository

Bbox parametrization identical to (Girshick et al., 2014), with smooth L1 loss. Multi-task loss for the RPN. Momentum(0.9), weight decay(0.0005), learning rate (0.001) for 60k minibatches, 0.0001 for 20k.

Multi-step training. Gradient is non-trivial due to the coordinate snapping of the boxes (see ROI align for a more continuous version)

With VGG-16, the conv5 layer is $H/16, W/16$. For an image 1000×600, there are $60 \times 40 = 2400$ anchor boxes centers.

FPN

Introduced in (Lin et al., 2017)

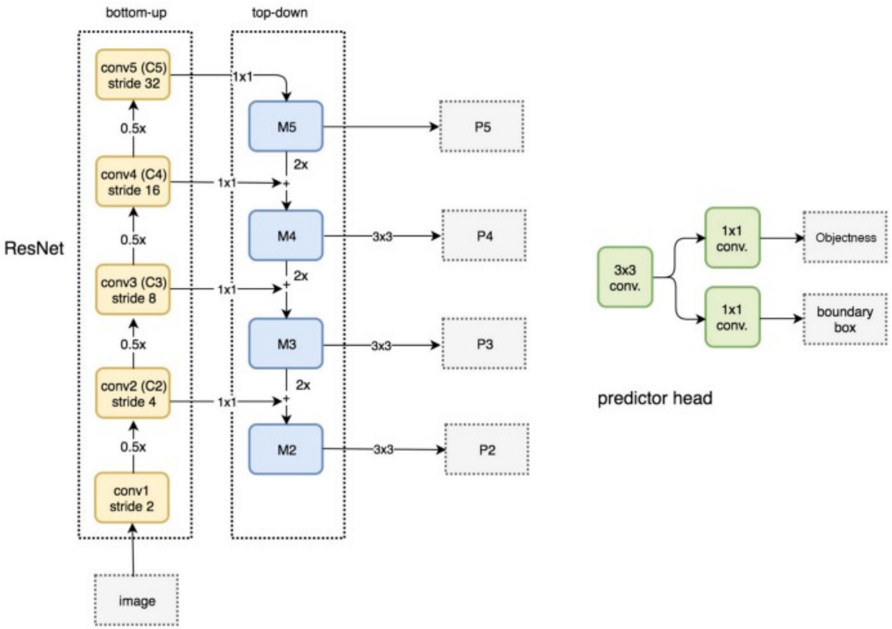


Illustration of a feature pyramid network. Credit to Jonathan Hui

Upsampling is performed by using nearest neighbors.

For object detection, a RPN is run on every scale of the pyramid P_2, P_3, P_4, P_5.

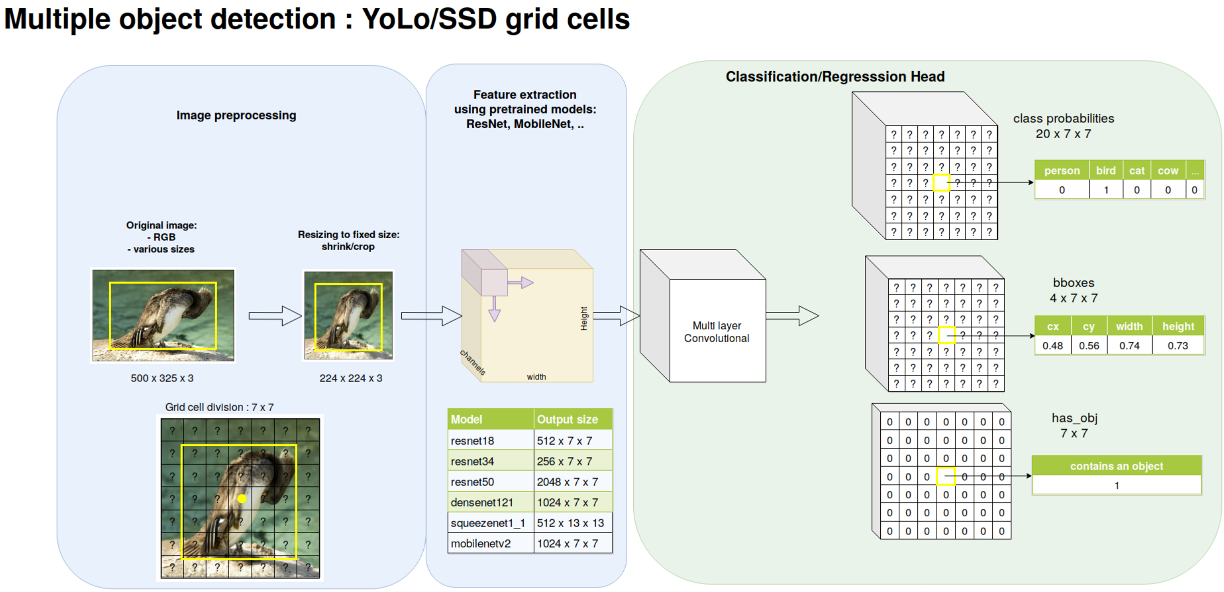
ROIPooling/Align is fed with the feature map at a scale depending on ROI size. Large ROI on small/coarse feature maps, Small ROI on large/fine feature maps

YOU ONLY LOOK ONCE (YOLO V1,V2,V3) (1/2)

The first one-stage detector. Introduced in (Redmon, Divvala, Girshick, & Farhadi, 2016). It outputs:

- B bounding boxes $(x, y, w, h, conf)$ for each cell of a $S \times S$ grid
- the probabilities over the K classes
- the output volume is $(5 \times B + K) \times (S \times S)$ in YoloV1, then $(5 + K) \times B \times (S \times S)$ from v2

Bounding box encoding:



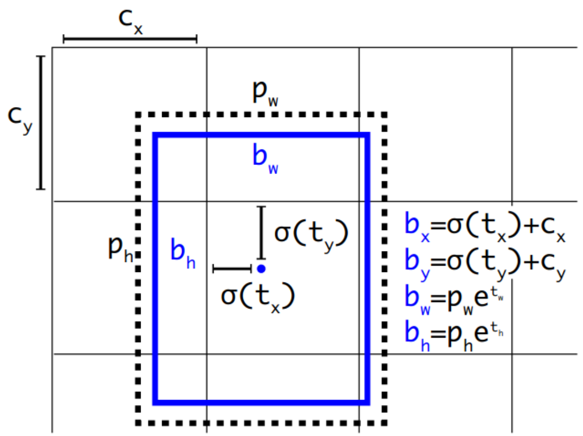
Model	Output size
resnet18	512 x 7 x 7
resnet34	256 x 7 x 7
resnet50	2048 x 7 x 7
densenet121	1024 x 7 x 7
squeezenet1_1	512 x 13 x 13
mobilenetv2	1024 x 7 x 7

person	bird	cat	cow	...
0	1	0	0	0

cx	cy	width	height
0.48	0.56	0.74	0.73

contains an object
1

YoLo v0 with one bounding box per cell $B = 1$



From Yolo v2

In YoLo v3, the network is Feature Pyramid Network (FPN) like with a downsampling and an upsampling paths, with predictions at 3 stages.

YOU ONLY LOOK ONCE (YOLO V1,V2,V3) (2/2)

The loss is multi-task with :

- a term for the regression of the transformation of the anchor boxes
- a term for detecting the presence of an object
- a term for the (possibly multi) labelling of the boxes

$$
\begin{aligned}
\mathcal{L} = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(t_x - t_x^*)^2 + (t_y - t_y^*)^2 + (t_w - t_w^*)^2 + (t_h - t_h^*)^2] \\
- \sum_{i=0}^{S^2} \sum_{j=0}^{B} BCE(\mathbb{1}_{ij}^{obj}, \text{has\_obj}_{ij}) \\
- \sum_{i=0}^{S^2} \sum_{j=0}^{B} \sum_{k=0}^{K} BCE(\text{has\_class}_{ijk}, p_{ijk})
\end{aligned}
$$

Darknet code

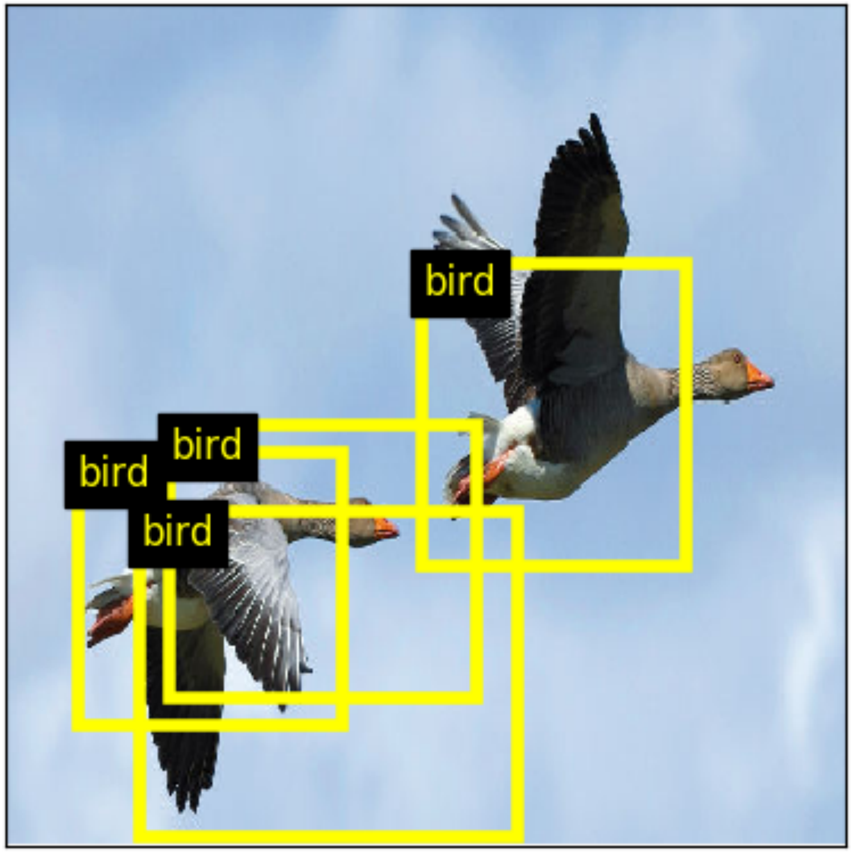
In v1 and v2, the prediction losses were L2 losses.

Multi labelling can occur in coco (e.g. women, person)

≡

NON MAXIMUM SUPPRESSION (NMS)

The object detectors may output multiple overlapping bounding box for the same object



Multiple bounding boxes for the same object

NMS algorithm :

- order the bounding boxes by decreasing confidence
- for every rank, remove every bounding box, of lower rank, with IoU higher than a threshold

NMS may suppress one of two “overlapped” objects. It hard resets the scores of overlapping bboxes.

SoftNMS (Bodla, Singh, Chellappa, & Davis, 2017):

- order the bounding boxes by decreasing confidence
- for every rank, scale the confidence of the bounding boxes of lower rank bboxes (rather than setting to 0)

SEMANTIC/INSTANCE SEGMENTATION

PROBLEM STATEMENT

Given an image,

Semantic segmentation : predict the class of every single pixel. We also call dense prediction/dense labelling.

Example image from MS Coco

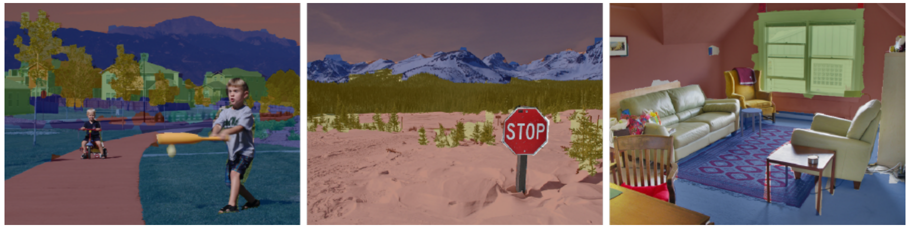


Image labeled with stuff classes

Instance segmentation : classify all the pixels belonging to the same countable objects

Example image from MS Coco

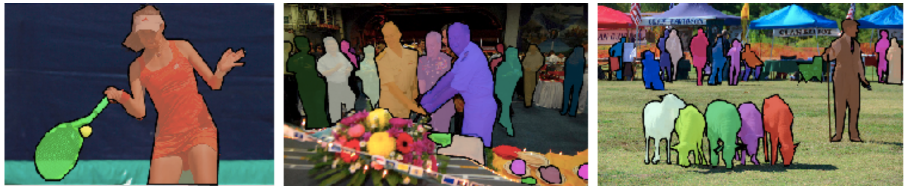


Image labeled with things classes

More recently, **panoptic segmentation** refers to instance segmentation for countable objects (e.g. people, animals, tools) and semantic segmentation for amorphous regions (grass, sky, road).

Metrics : see Coco panotpic evaluation

Some example networks : PSP-Net, U-Net, Dilated Net, ParseNet, DeepLab, Mask RCNN, …

CONCLUSION

≡

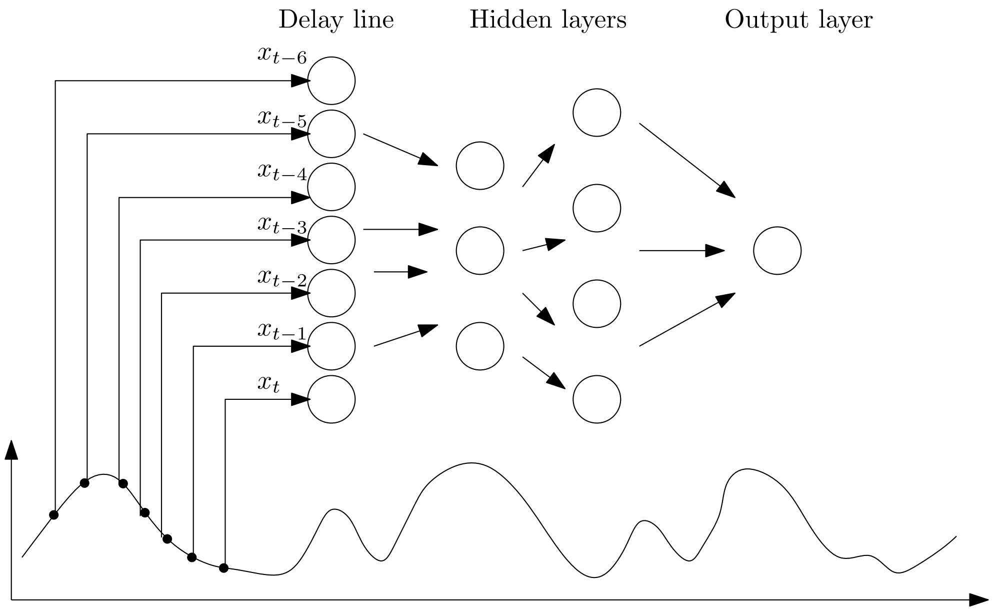
OTHER CNN RELATED PROBLEMS WE DID NOT DISCUSS

- Super resolution :
  - Learn to upscale an image
  - example networks : SRCNN, FSRCNN, VDSR, ESPCN, RED-Net
- Graph convolutions :
  - Learn to aggregate features on graph which are **structures of arbitrary sizes and arbitrary branching factor**
  - example networks : Weisfeilher lehman, see pytorch geometric, AlphaChem, ..
- Processing 3D point clouds :
  - unstructured sparse irregular 3D data
  - example networks : PointNet

DEALING WITH SEQUENTIAL DATA

SPATIALISING THE TIME : TDNN

Time delay neural networks as introduced in (Waibel, Hanazawa, Hinton, Shikano, & Lang, 1989) spatializes the time:



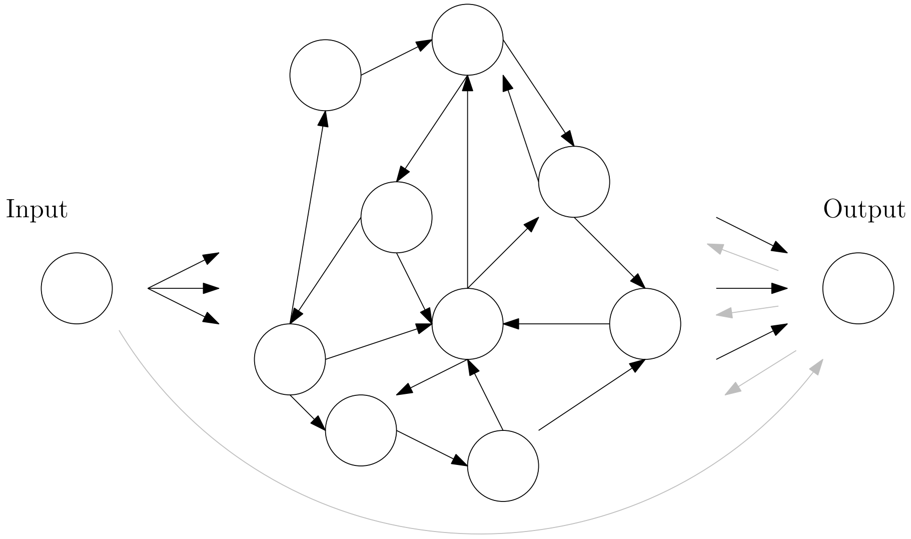
Time delay neural network

But : which size of the time window ? Must the history size always be the same ? Do we need the data over the whole time span ? How to share computations in time instead of using distinct weights per time instant?

Feedforward neural networks can still be efficient for processing sequential data, e.g. Gated ConvNet (Dauphin, Fan, Auli, & Grangier, 2017), Transformers, …

ARCHITECTURE OF A RNN

Introduced by (Elman, 1990).



Recurrent neural network

Weight matrices :

- W^{in} input to hidden
- W^{h} hidden to hidden
- W^{out} hidden to output
- W^{back} outputs to hidden

$$h(t) = f(W^{in}x(t) + W^{h}h(t-1) + W^{back}y(t-1) + b$$
$$y(t) = g(W^{out}h(t) + b_y)$$

The hidden to hidden weight matrix W_h is repeatedly applied.

Named **Elman networks** if $W^{back} = 0$, and **Jordan networks** if $W^{h} = 0$. Elman networks with a random fixed W^{h} are called **Echo State networks**.

MANY TO ONE, ONE TO MANY, MANY TO MANY

The inputs and outputs can be of variable ($1 \rightarrow T_x, 1 \rightarrow T_y$) and arbitrary sizes ($T_x \neq T_y$).

- **Many to one** example : language model, sentiment analysis : multiclass sequence classification $T_y = 1$:
  - $['this','movie','was','fantastic'] \mapsto 1$
  - $['you','should','not','spend','even','a','single','penny','watching','this','movie'] \mapsto 0$
- **Many to many** example : Neural Machine Translation
  - What is the most likely EN translation of “La croissance économique s’est ralentie ces dernières années.” : “Economic growth has slowed down in recent years.”
- **One to many** example: image captioning, language model with probabilistic sampling

start : ‘LA JUMENT ET’

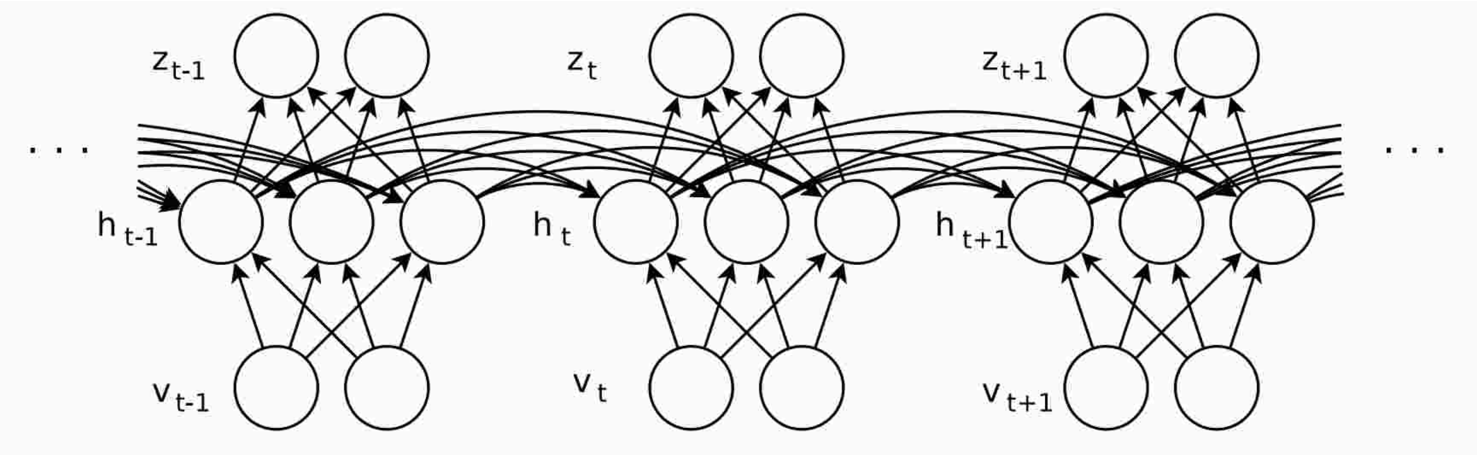
LA JUMENT ET LE RAT
ET L’huiller craignait les gens d’une mise un vers atteint:
Va c’est d’être indigne de Vénus d’aller pressez l’ame
D’une mais, dit-il, un plongeant l’avertion :
Son échangé vous refusiez-vous

start : ‘LA JUMENT ET’

LA JUMENT ET LE BULÉE
[Ésope]
Comme à part craindre déjà cet honneur à couvrir jamais
Et ses mélonces, condition tempérament.
L’autre honne alla vie.
Je ne saurais pas que d’un moutons.
Que ce choix, coquet, g

TRAINING WITH (TRUNCATED)-BPTT

Idea: unfold in time the computational graph and perform **reverse mode** differentiation (Werbos, 1990).



Backpropagation through time. Image from (Sutskever, 2013)

You must be training on truncated series to prevent a computational burden.

You can also perform **forward mode** differentiation (**Real time recurrent learning** RTTL (Williams & Peng, 1990)) with online adaptation as the inputs/targets comes in but this is computationally expensive.

≡

TRAINING A RNN CAN BE HARD

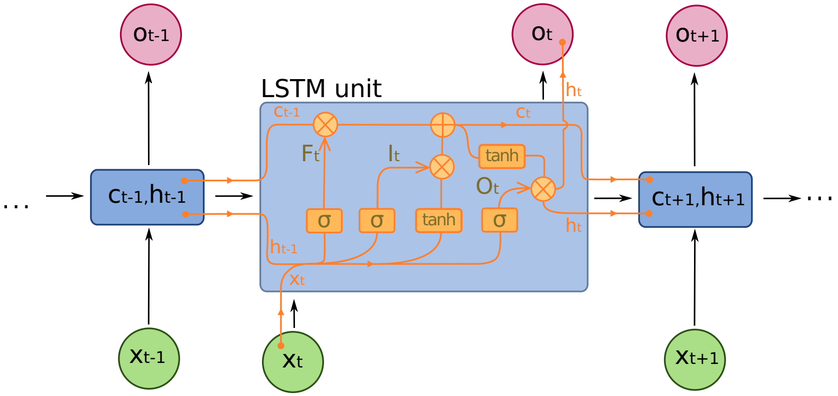
Unrolled in time, RNN appears as very deep networks → vanishing/exploding gradient

- Initialization strategies :
  - LSTM : high forget bias to favor a remember by default (Gers, Schmidhuber, & Cummins, 2000)
  - orthogonal weight initialization (Henaff, Szlam, & LeCun, 2016), to keep the hidden activities normalized $\|Wh\|_2^2 = h^T W^T W h = h^T h = \|h\|_2^2$, see also (Arjovsky, Shah, & Bengio, 2016)
  - identity recurrent weight initialization to favor a copy as-is by default for RNN (Le, Jaitly, & Hinton, 2015)
- Architecture :
  - in-lieu of Batch Normalization : Layer normalization (Ba, Kiros, & Hinton, 2016); statistics are computed independently per sample over the whole layer
- Training :
  - gradient clipping (Pascanu et al., 2013) seems to be frequently used,
  - activation clipping is sometimes considered (Hannun et al., 2014),
- Regularization:
  - Noise (Kam-Chuen Jim, Giles, & Horne, 1996), (Graves, Mohamed, & Hinton, 2013)
  - Naive dropout is not an option (LSTM/GRU memory cell states could be masked)
  - ZoneOut (Krueger et al., 2017), rnnDrop (Moon, Choi, Lee, & Song, 2015), ..

MEMORY CELLS

LONG-SHORT TERM MEMORY (LSTM)

RNNs have difficulties learning long range dependencies. The LSTM (Hochreiter & Schmidhuber, 1997) introduces **memory cells** to address that problem.



LSTM memory cell. Image from Wikipedia

Equations:

$$\begin{aligned}
I_t &= \sigma(W_i^x x_t + W_i^h h_{t-1} + b_i) && \in [0,1], \text{Input ga} \\
F_t &= \sigma(W_f^x x_t + W_f^h h_{t-1} + b_f) && \in [0,1], \text{Forget ga} \\
O_t &= \sigma(W_o^x x_t + W_o^h h_{t-1} + b_o) && \in [0,1], \text{Output ga} \\
n_t &= \tanh(W_n^x x_t + W_n^h h_{t-1} + b_z) && \text{unit's inp} \\
c_t &= F_t \odot c_{t-1} + I_t \odot n_t && \text{cell upda} \\
h_t &= O_t \odot \tanh(c_t) && \text{unit's outp}
\end{aligned}$$

Peepholes may connect the c_t to **their** gates.

The next layers integrate what is exposed by the cells, i.e. the unit's output h_t, not c_t.

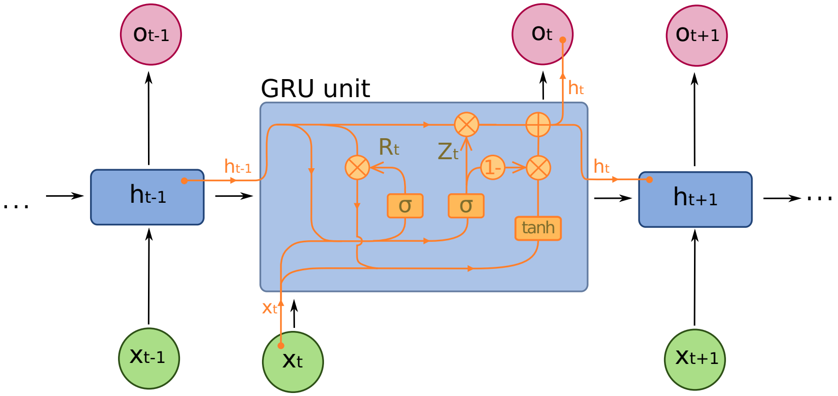
If $F_t = 1, I_t = 0$, the cell state c_t is unmodified. This is called the *constant error carrousel*.

The forget gate is introduced in (Gers et al., 2000). Variants have been investigated in a search space odyssey (Greff, Srivastava, Koutník, Steunebrink, & Schmidhuber, 2017).

See also (Le et al., 2015) which reconsiders using ReLU in LSTM given appropriate initialization of the recurrent weights to the identity to be copy by default mode.

GATED RECURRENT UNITS (GRU)

The GRU is introduced as an alternative, simpler model than LSTM. Introduced in (Cho et al., 2014).



GRU memory cell. Image from Wikipedia

Equations:

$$
\begin{aligned}
R_t &= \sigma(W_i^x x_t + W_i^h h_{t-1} + b_i) \text{ Reset gate} \\
Z_t &= \sigma(W_z^x x_t + W_z^h h_{t-1} + b_z) \text{ Update gate} \\
n_t &= \tanh(W_n^x x_t + b_{nx} + R_t \odot (W_n^h h_{t-1} + b_{nh})) \\
h_t &= Z_t \odot h_{t-1} + (1 - Z_t) \odot n_t
\end{aligned}
$$

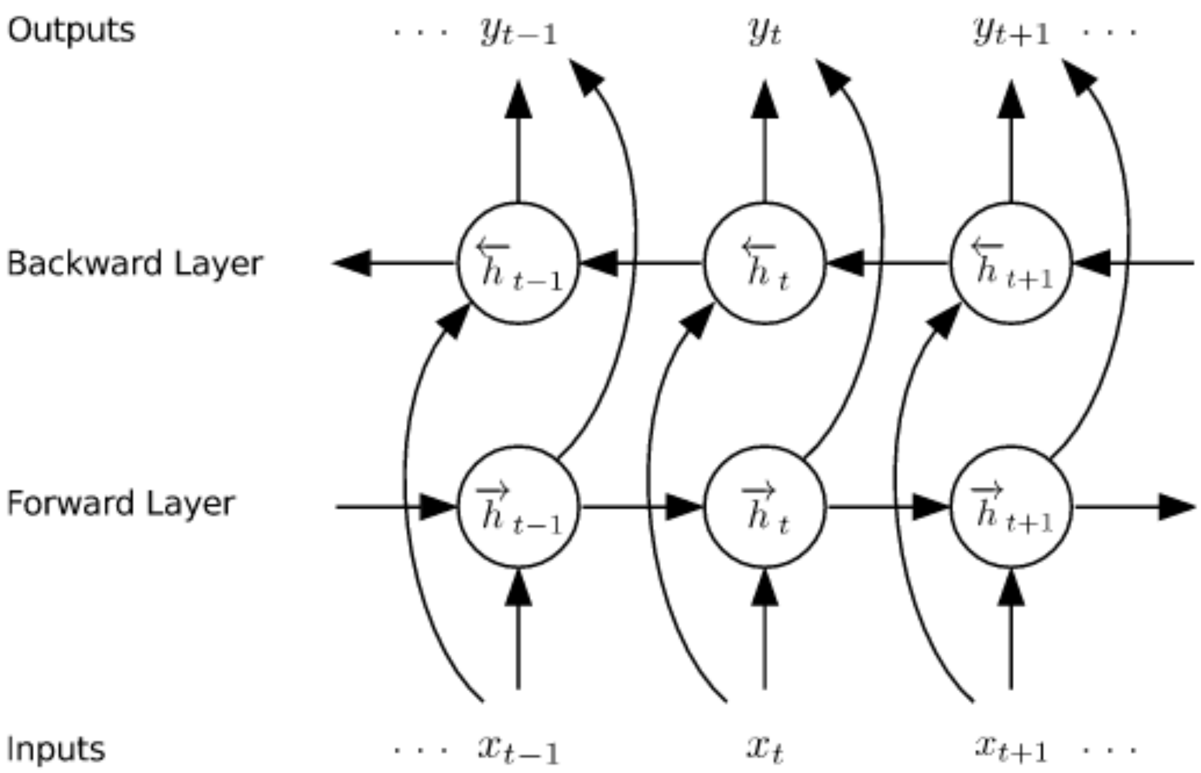
If $Z_t = 1$, the cell state h_t is not modified. If $Z_t = 0$ and $R_t = 1$, it is updated in one step.

Compared to LSTM, a GRU cell :

- unconditionally exposes its hidden state (there is no private cell state c_t)
- the hidden state is reset by getting $R_t = 0$

BIDIRECTIONAL RNN/LSTM/GRU

Idea Both past and future contexts can sometimes be required for classification at the current time step; e.g. when you speak, past and future phonemes influence the way you pronounce the current one. Introduced in (Schuster & Paliwal, 1997})



Bidirectionnal RNN. Image from (Graves et al., 2013)

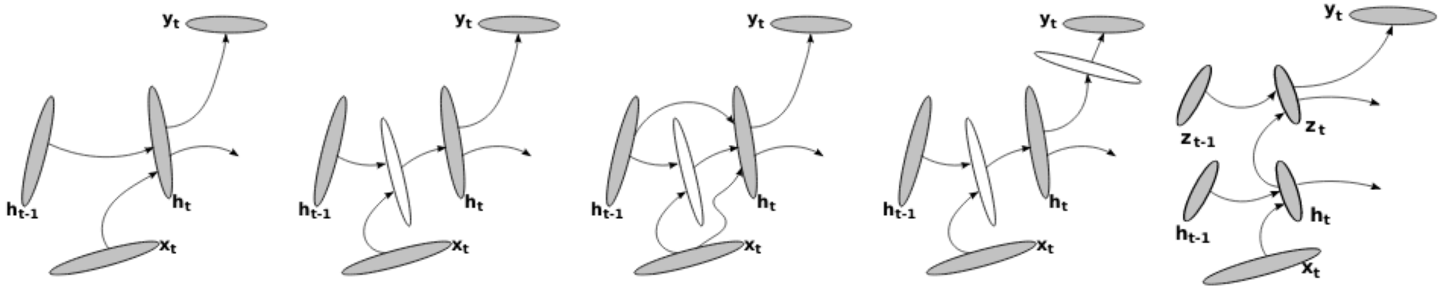
≡

DEEP RNNS

While RNN are fundamentally deep neural networks, they can still benefit from being stacked : this allows the layers to operate at increasing time scales. The lower layers can change their content at a higher rate than the higher layers.

- (Graves et al., 2013): Phoneme classification with stacked bidirectionnal LSTMs
- (Sutskever, Vinyals, & Le, 2014) : Machine translation with stacked unidirectionnal LSTMs (Seq2Seq)

In a stacked RNN, you can concatenate consecutive hidden states before feeding in the next RNN layer, e.g. Listen, Attend and Spell encoder (→ downscale time)



(a) RNN (b) DT-RNN (b*) DT(S)-RNN (c) DOT-RNN (d) Stacked RNN

Deep RNN variants. Image from (Pascanu et al., 2014)

Other variants for introducing depth in RNN is explored in (Pascanu et al., 2014). For example, the transition function from h_{t-1} to h_t is not deep, even in stacked RNNs but is deep in DT-RNN.

DEFINING RNN IN PYTORCH

Stacked bidirectional LSTM, documentation

with discrete inputs (words, characters, …) of the same time length, one prediction per time step.

```
import torch
import torch.nn as nn

seq_len = 51
batch_size = 32
vocab_size = 10
embedding_dim = 128
```

You can provide an initial state to the call function of the LSTM, in which case, you must take out the LSTM from the nn.Sequential, by default $\overrightarrow{h}_0 = \overleftarrow{h}_0 = \overleftarrow{c}_0 = \overrightarrow{c}_0 = 0$). You could learn these initial hidden states (to bias the first operations of your rnn).

All the weights and biases and initialized from LeCun like initialization $\mathcal{U}(-\sqrt{k}, \sqrt{k}), k = \frac{1}{\text{hidden\_size}}$

Some authors (Gers et al., 2000) suggest to favor either long-term dependencies or short-term dependencies by setting the bias of the forget gate accordingly (“Learning to forget”, $F_{t=0} = 1$ to remember everything by default).

See the lab work for specificities on representing variable sized sequences with pytorch PackedSequences.

DIGGING INTO PYTORCH CODE

How do you know how to access these weights ? See the doc

Note several redundant biases $b_{i\cdot}$ and $b_{h\cdot}$ (for CuDNN compatibility).

CUSTOM INITIALIZATION

Task: initialize in the “Learning to forget” regime of (Gers et al., 2000)

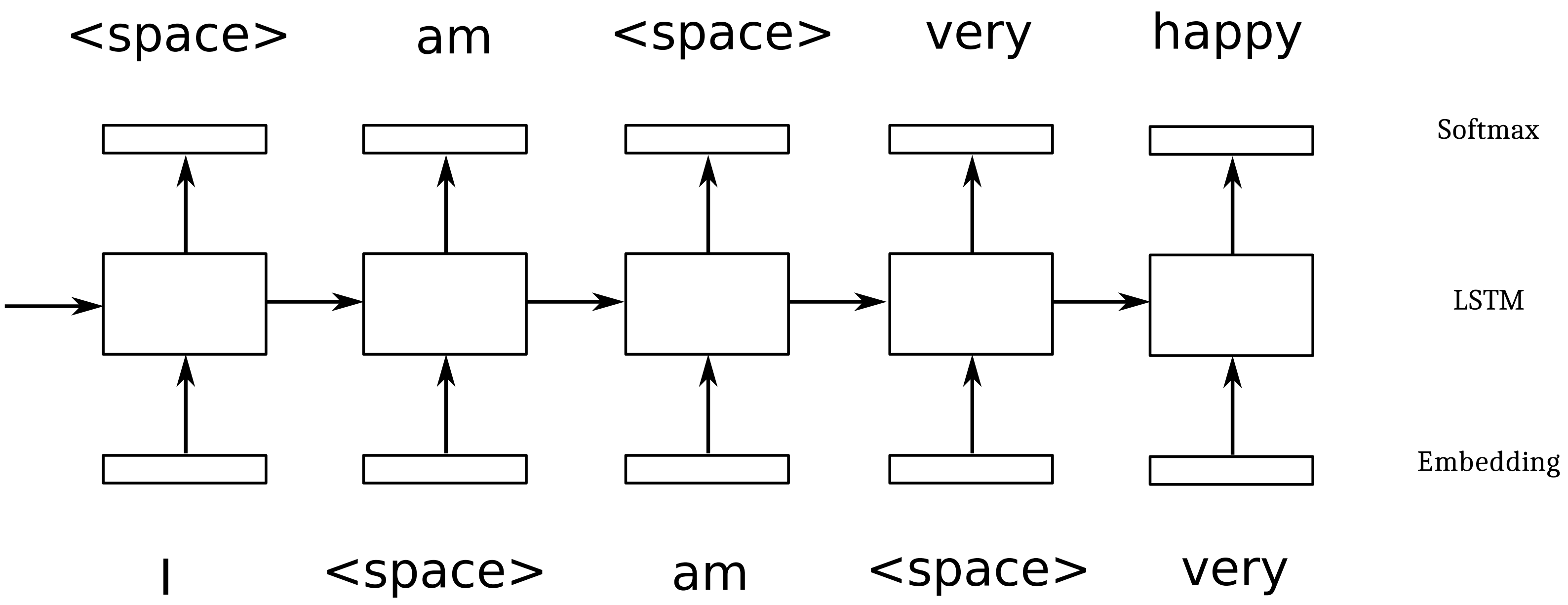
```
num_layers=3

rnn = nn.LSTM(input_size=embedding_dim,
              hidden_size=hidden_size,
              num_layers=num_layers,
              bidirectional=True)
```

The ordering of the weights/biases are inputg/forgetg/cell/outputg.

LANGUAGE MODEL : AN EXAMPLE OF ALIGNED SEQUENCES OF IDENTICAL SIZES

CHARACTER LEVEL LANGUAGE MODEL (CHAR-RNN)

Problem given fixed length chunks of sentences, predict the next word/character : $p(x_T|x_0, x_1, \ldots, x_{T-1})$



Example RNN language model

Many to many during training (teacher forcing (Williams & Peng, 1990)) but many to one for inference.

A language model can be used, e.g., to constrain the decoding of a network outputting sentences (e.g. in speech-to-text or captioning tasks)

See also The unreasonnable effectiveness of recurrent neural networks and (Sutskever, Martens, & Hinton, 2011).

TRAINING AND SAMPLING FROM THE CHARACTER RNN

Example on “Les fabulistes”

- Vocabulary of 105 elements : {‘\t’: 0, ‘\n’: 1, ’ ‘: 2,’!‘: 3,’“‘: 4, "’”: 5, ‘(’: 6, ‘)’: 7, ‘*’: 8, ‘+’: 9, ‘,’: 10, ‘-’: 11, ‘.’: 12, ‘/’: 13, ‘0’: 14, ‘1’: 15, ‘4’: 16, ‘5’: 17, ‘6’: 18, ‘8’: 19, ‘9’: 20, ‘:’: 21, ‘;’: 22, ‘<’: 23, ‘>’: 24, ‘?’: 25, ‘A’: 26, ‘B’: 27, ‘C’: 28, ‘D’: 29, ‘E’: 30, ‘F’: 31, ‘G’: 32, ‘H’: 33, ‘I’: 34, ‘J’: 35, ‘L’: 36, ‘M’: 37, ‘N’: 38, ‘O’: 39, ‘P’: 40, ‘Q’: 41, ‘R’: 42, ‘S’: 43, ‘T’: 44, ‘U’: 45, ‘V’: 46, ‘W’: 47, ‘X’: 48, ‘Y’: 49, ‘Z’: 50, ‘\[’: 51, …}
- Dataset size : 10.048 non overlapping chunks of length 60.
- Example samples :
  - Input : [2,71,67,54,70,57,10,2,56,95,71…65,57,66,72,2,70,57,55,60,57]

    " sobre, dé…ment reche"
  - Output [71,67,54,70,57,10,2,56,95,71,61…57,66,72,2,70,57,55,60,57,70]

    “sobre, dés…ent recher”
- Network : Embedding(64) , 2 × LSTM(64), 2 ×LinearRelu(128), LinearSoftmax(105), 111.657 parameters

Note we use **uni**-directional LSTM. With **bi**-directionnal LSTM, the problem is easily solved by using the backward LSTM only.

- Loss : cross-entropy averaged over batch_size × seq_len
- Training: Adam(0.01), learning rate halved every 10 steps, gradient clipping (5) (not sure it helped though)

SAMPLING FROM THE CHARACTER RNN

- After 30 epochs, validation loss of 1.45 and validation accuracy of 56%.
- To sample from the language model, you can provide it with some context sentence, e.g. [‘L’, ‘A’, ’ ‘, ’G’, ‘R’,‘E’,‘N’,‘O’,‘U’, ‘I’, ‘L’, ‘L’, ‘E’, ’ ’]

Sample of 200 chars after init

```
LA GRENOUILLE y
-Ô)asZYc5[h+IÉë?8—>Y.bèqp;ÎzÇÇ<)!|f]Lt+«-u
XûoÜ:!ïgVùb|Ceü9ùÈ«à
6)ZàÀçBJi)X:ZÛdzxQ8PcvïV]O]xPX,Înc.è’Pâs:X;ûfjBâ?X
ç’ED’fSOl*Z(È’È1SnjàvPïLoUÊêàDgùO9z8eJûRYJ?Yg
Uâp|jCbû—HxBràZBMZÛPCGuR’]ÀiÊÂSBF4D),û
```

Sample 1 of 200 chars after 30 epochs

```
LA GRENOUILLE ET MOURE ET LA RENARDIER
Quel Grâce tout mon ambassade est pris.
L’un pourtant rare,
D’une première
Qu’à partout tout en mon nommée et quelques
fleuris ;
Vous n’oserions les Fermerois, les heurs la
```

Note the upper case after the line breaks, the uppercase title, the quite existing words. The text does not make much sense but it is generated character by character !

Sample 2 of 200 chars (from the same model as before)

```
LA GRENOUILLE D’INDÉTES
[Phèdre]
Tout faire force belle, commune,
Et des arts qui, derris vôtre gouverne a rond d’une partage conclut sous besort qu’il plaît du lui dit Portune
comme un Heurant enlever bien homme,
```

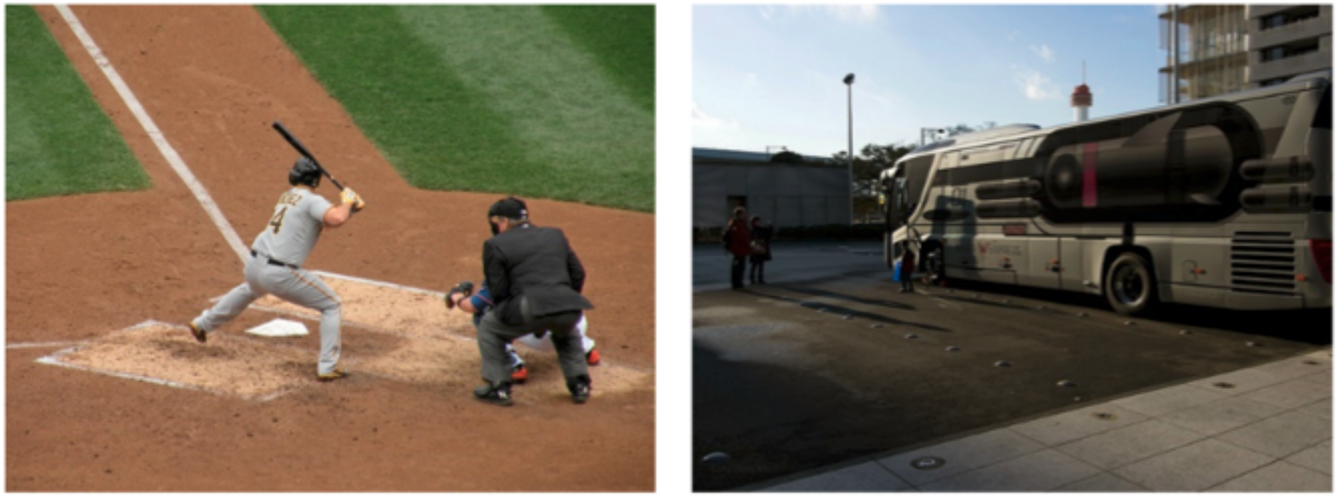
More on language modeling (metrics, models, …) in the Deep NLP lecture of Joel Legrand.

GENERATING TEXT DESCRIPTIONS FROM IMAGES (ONE TO MANY)

IMAGE CAPTIONING

Problem Given an image, generate a textual description of it.

Example datasets : Coco captions, Flickr8k, Flickr30k



The man at bat readies to swing at the pitch while the umpire looks on.

A large bus sitting next to a very tall building.

Example of captioning samples from MSCoco Image captioning 2015

Some of the first entries : (Vinyals, Toshev, Bengio, & Erhan, 2015), (Xu et al., 2016)

Difficulty: object detection with their relationship

SHOW AND TELL

Idea Use a pre-trained CNN for image embedding plugged into a RNN for generating (decoding) the sequence of words. Introduced in (Vinyals et al., 2015).

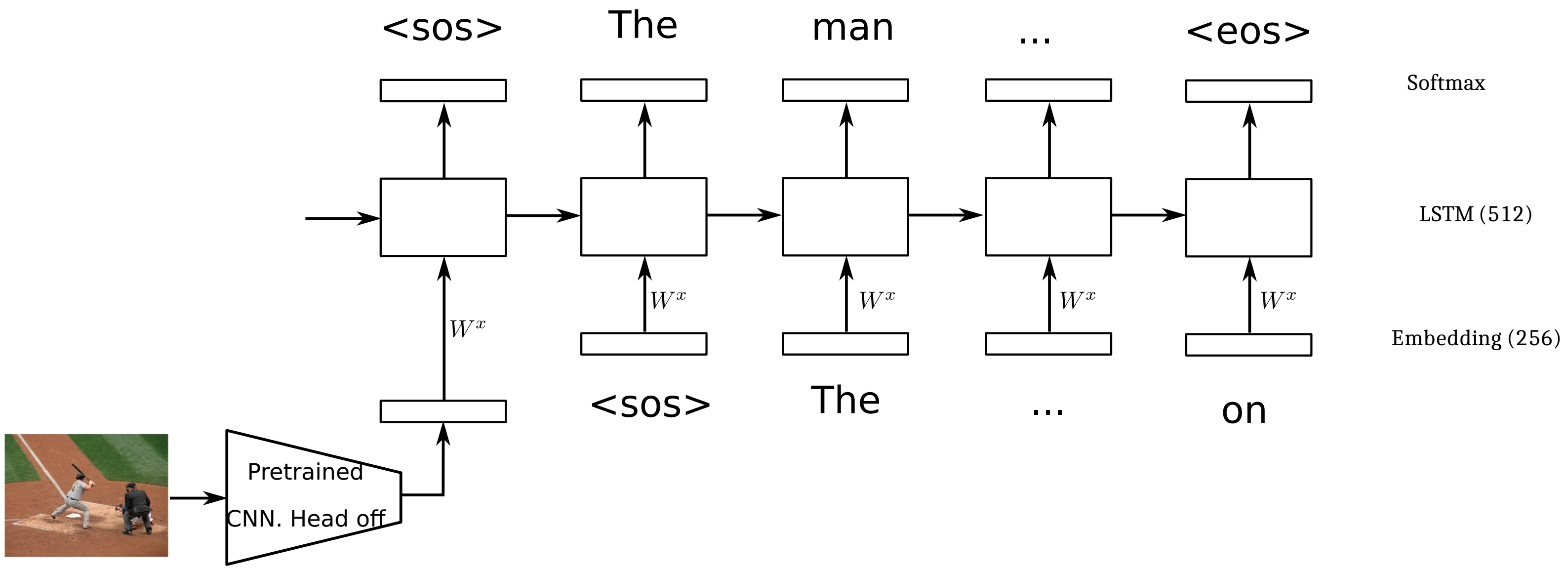
Learn a model maximizing :

$$p(S_0S_1S_2..S_T|I,\theta) = p(S_0|I,\theta)\prod_{j>0}^{T} p(S_j|S_0S_1\dots S_{j-1},I,\theta)$$

i.e. minimizing $-\log(p(S_0S_1S_2..S_T|I,\theta)) = -\sum_j \log(p(S_j|S_0\dots S_{j-1},I,\theta))$

Inspired by the Seq2Seq approach successful in machine translation (more on this later), they proposed an encoder-decoder model to *translate an image to a sentence*

SHOW AND TELL



Show and tell architecture

Training ingredients :

- GoogleNet CNN pretrained on ImageNet
- words embeddings randomly initialized (pretraining on a large news corpus did not help)
- embedding of size 512
- LSTM with 512 cells
- Stochastic gradient descent, no momentum,
- training the LSTM with frozen CNN then finetuning the whole. Too early training end-to-end fails
- scheduled sampling (otherwise, divergence between teacher forcing training and inference performances)

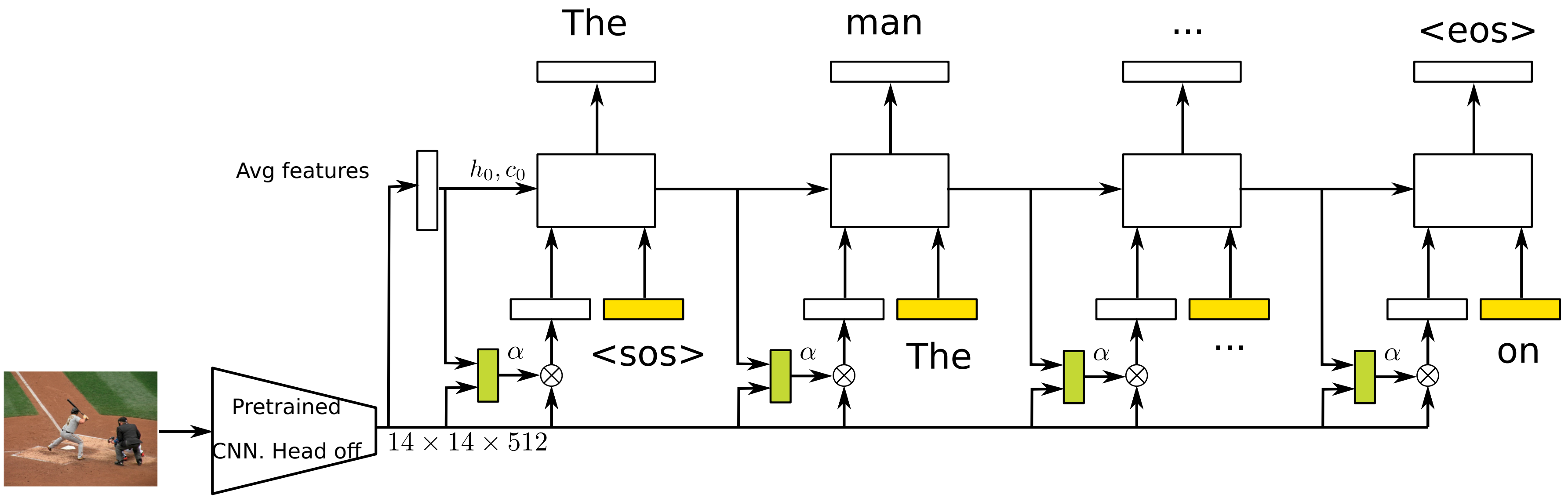
Introducing the visual convolutional features at every step did not help.

Inference :

- Decoding by beam search (beam size 20), then reduced beam size to 3 yielded, unexpectedly, better results

SHOW ATTEND AND TELL

Idea Allow the RNN to filter out/focus on CNN features during generation using an attention mechanism (Bahdanau, Cho, & Bengio, 2015). Introduced in (Xu et al., 2016). Link to theano source code



Show attend and tell architecture with soft attention. The alphas are normalized to sum to 1 (softmax).

Training:

- resnet CNN (head off)
- vocabulary of 10000 words
- Embedding (100), LSTM(1000),
- RMSProp(0.1),
- dropout for $h_0, c_0,$,
- Early stopping on the BLUE score

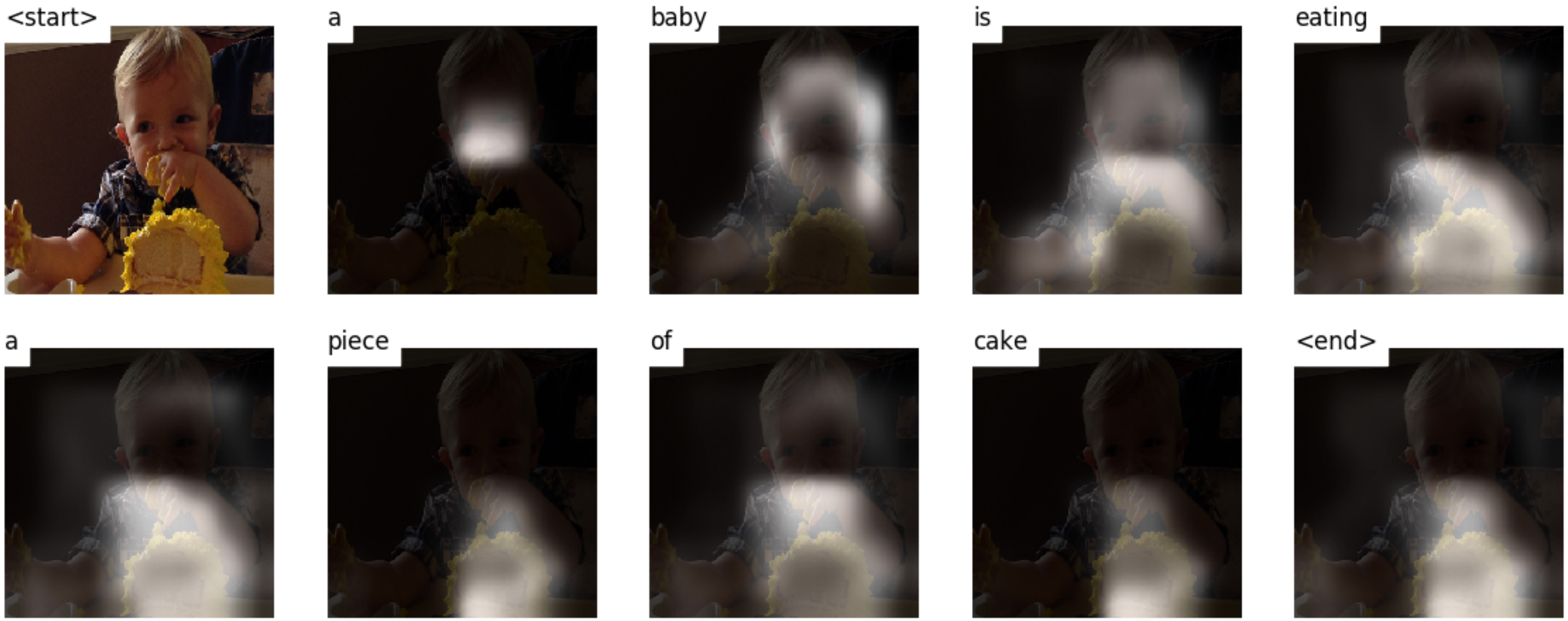
Double stochastic attention :

- by construction $\sum_i \alpha_{t,i} = 1$
- regularization $\lambda \sum_{loc} (1 - \sum_t \alpha_{t,loc})^2$ to enforce the model to pay equal attention to all the locations, norm in time for every location.

Inference:

- Decoding by beam search

SHOW ATTEND AND TELL



Example of caption with the attentional mask. Image from this nice tutorial and pytorch implementation

DEALING WITH VARIABLE SIZE UNALIGNED INPUT/OUTPUT SEQUENCES

WHERE IS THE PROBLEM

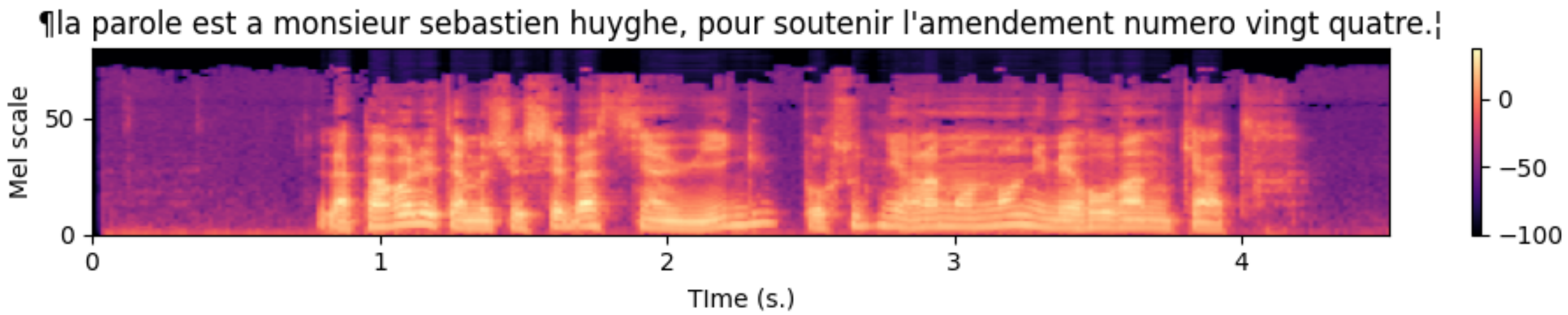
Problem In tasks such as Machine Translation (MT) or Automatic Speech Recognition (ASR), input sequences get mapped to output sequences, both can be of arbitrary sizes.

Machine translation :

The proposal will not now be implemented

Les propositions ne seront pas mises en application maintenant

Automatic speech recognition



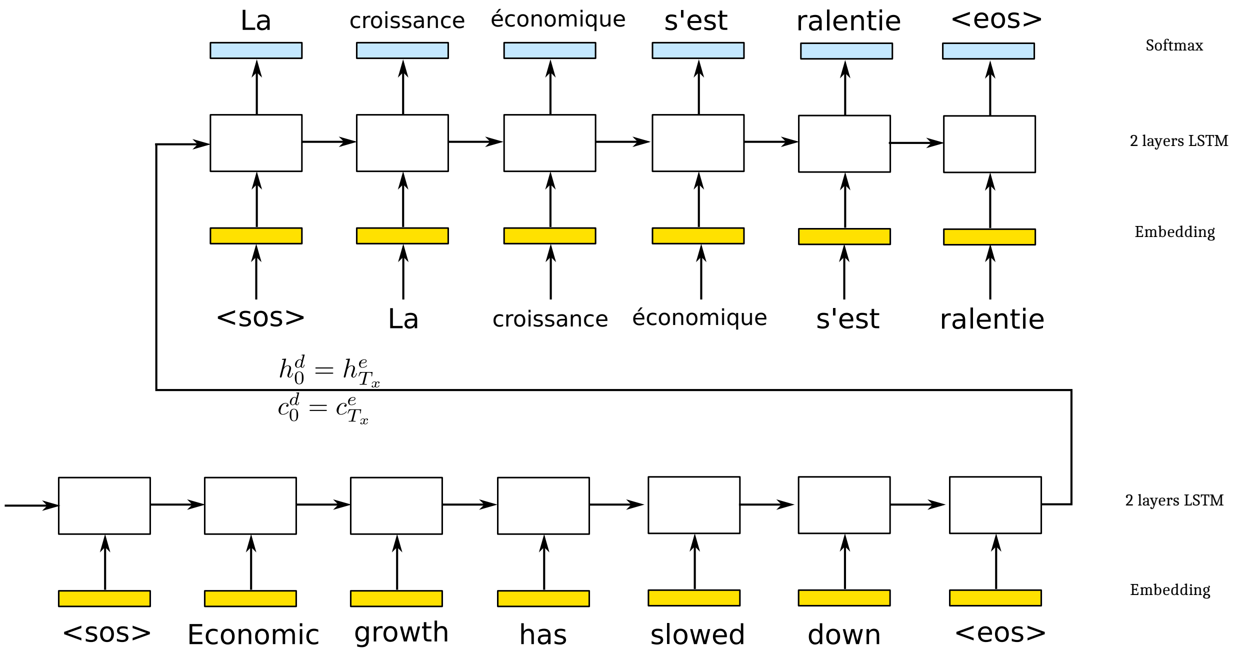
A mel-spectrogram with its expected transcript. The spectrogram is sampled at

The **alignment** can be difficult to explicit. Contrary to the language model, we may not know easily when to output what.

WHEN THE ALIGNMENT IS MISSING : SEQ2SEQ

ENCODER / DECODER ARCHITECTURES

Idea Encode/Compress the input sequence to a hidden state and decode/decompress the output sequence from there. Introduced in (Cho et al., 2014) for ranking translations and (Sutskever et al., 2014) for generating translations (NMT).



Seq2Seq architecture for Neural Machine Translation

Architecture :

- 4 layers LSTM$(1000, \mathcal{U}(-0.08, 0.08))$, Embeddings$(1000)$
- Vocabularies (in:160.000, out: 80.000)
- SGD(0.7), halved every half epoch after 5 epochs. Trained for 7.5 epochs. Batch(128)
- gradient clipping 5
- 10 days on 8 GPUs

The input sentence is fed in reverse order.

Beam search decoding. Teacher forcing for training but see also Scheduled sampling or Professor Forcing.

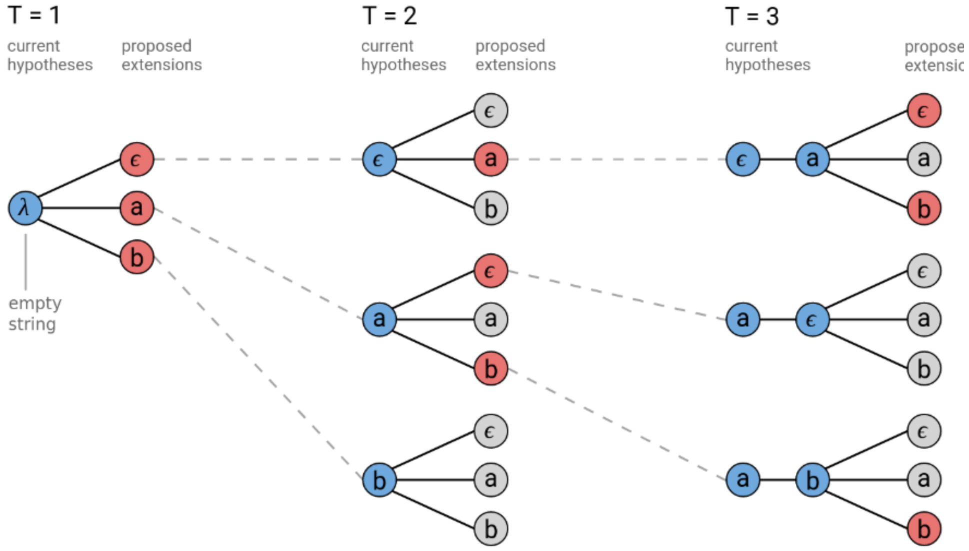
See Cho’s blog post. See this implementation in pytorch

DECODING WITH BEAM SEARCH

To get the most likely translation, you need to estimate

$$p(y|x) = p(y_0|x, \theta) \prod_t p(y_t|y_0 \ldots y_{t-1} x \theta)$$

But the probability distribution over the labels is dependent on the previously generated label (which feeds the input for the next step) $\rightarrow$ approximate search by maintaining a set of B candidates.



Beam search decoding with beam size B. Image from distill.pub

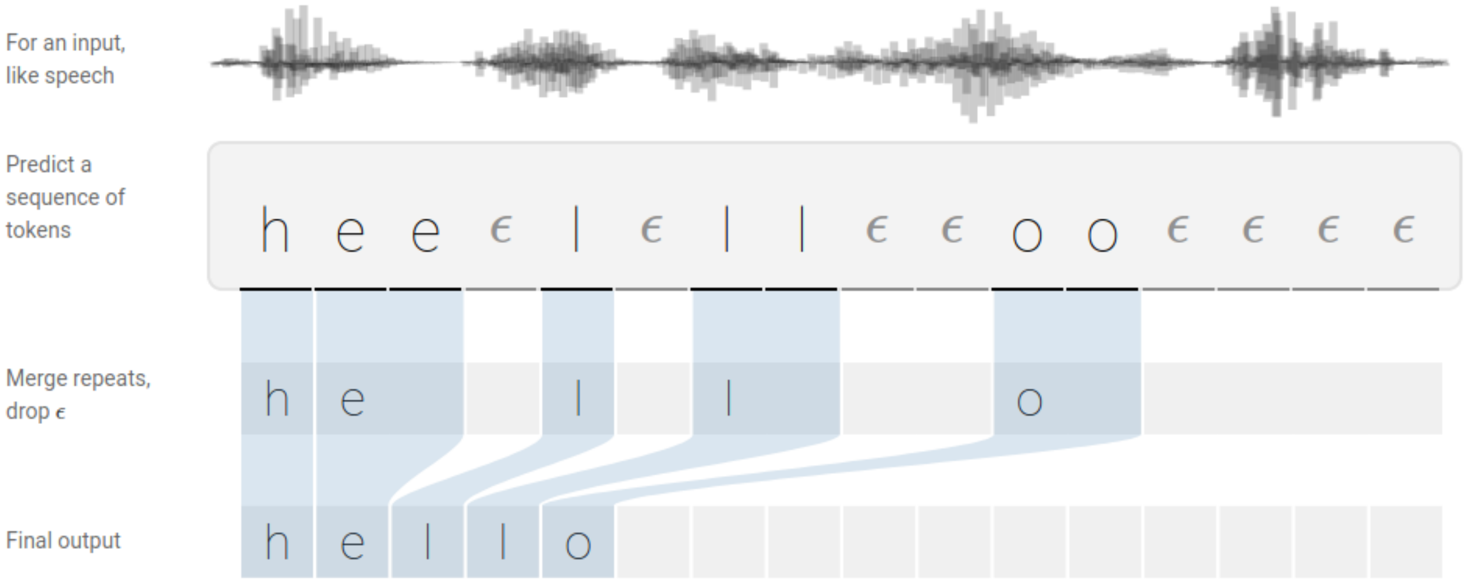
See also the modified beam search scoring of GNMT (Wu et al., 2016).

WHEN THE ALIGNMENT IS MISSING : CTC

≡

CONNECTIONIST TEMPORAL CLASSIFICATION (CTC)

Idea For problems with the output sequence length T_y is smaller than the input sequence T_x, allow a **blank** character. Introduced in (Graves, Fernández, Gomez, & Schmidhuber, 2006)


For an input, like speech
Predict a sequence of tokens
h e e ϵ l ϵ l l ϵ ϵ o o ϵ ϵ ϵ ϵ
Merge repeats, drop ϵ
h e l l o
Final output
h e l l o

CTC collapsing. Illustration from distill.pub

The collapsing many-to-one mapping $\mathcal{B}$ removes the duplicates and then the blanks.

The CTC networks learn from all the possible alignments of X with Y by adding the extra-blank character. Allows to learn from **unsegmented** sequences !

See also alternatives of the **blank** character in (Collobert, Puhrsch, & Synnaeve, 2016).

CTC TRAINING : CTC LOSS

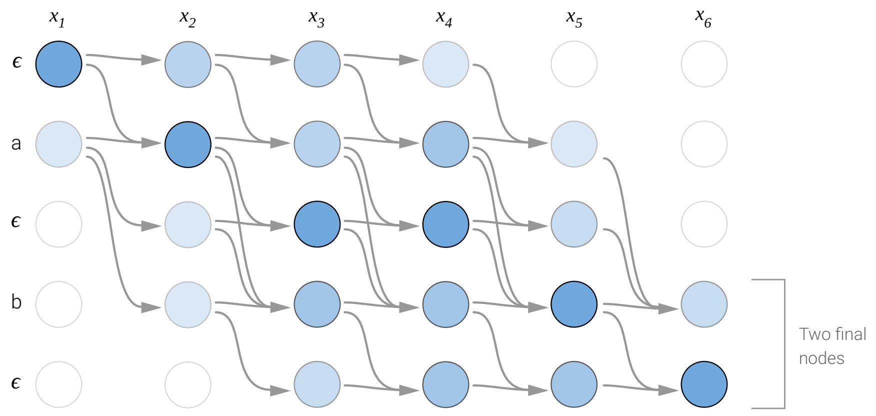
- Step: extend your model to output a **blank** character.

- Use the CTC Loss which is estimating the probability of a labeling by marginalizing over all the possible alignments. Assuming conditional independence of the outputs :

$$p(Y|X) = \sum_{\pi} p(\pi|x) \\ = \sum_{\pi} \prod_{t} p(\pi_t|x)$$

No need to sum over the possibly large number of paths π, it can be computed recursively.

Graphical representation from distill.pub



CTC cost efficient computation

Recursively compute $\alpha_{s,t}$ the probability assigned by the model at time t to the subsequence (extended with the blank) $y_{1:s}$

You end up with a computational graph through which the gradient can propagate.

CTC DECODING BY COMBINING THE ALTERNATIVES

Problem During inference, given an input x, what is the most probable **collapsed** labeling ? This is intractable.

Solution 1: best path decoding by selecting, at each time step, the output with the highest probability assigned by your model

$$\hat{y}(x) = \mathcal{B}(\text{argmax}_{\pi} p(\pi|x,\theta)) = \mathcal{B}(\text{argmax}_{\pi} \prod_t p(\pi_t|x,\theta))$$

But the same labeling can have many alignments and the probability can be spiky on one bad alignment.

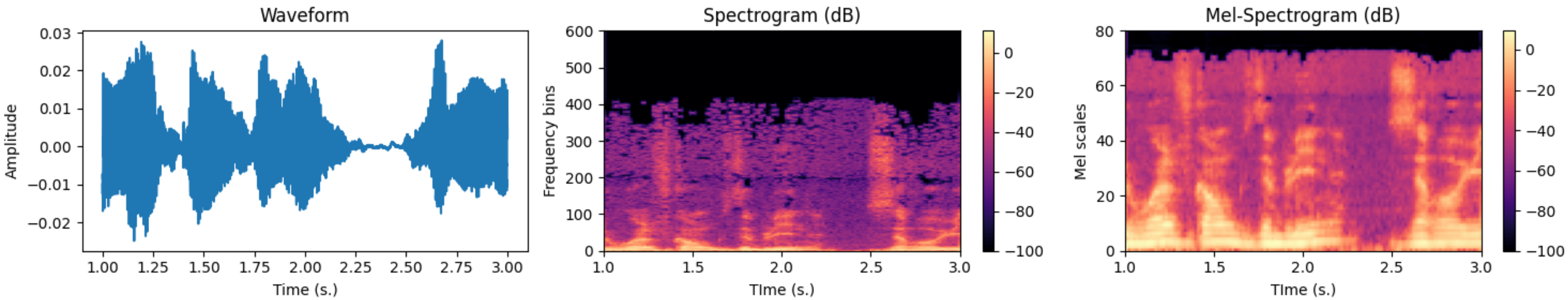
Solution 2: beam search decoding taking care of the **blank** character (multiple paths may collapse to the same final labeling)

Possibility to introduce a language model to bias the decoding in favor of plausible words. See (Hannun et al., 2014) :

$$\text{argmax}_y (p(y|x) p_{LM}(y)^{\alpha} \text{wordcount}^{\beta}(y))$$

CTC EXAMPLE ON VOICE RECOGNITION

Problem Given a waveform, produce the transcript.
Example datasets : Librispeech (English, 1000 hours, Aligned), TED (English, 450 hours, Aligned), Mozilla common voice (Multi language, 2000 hours in English, 600 hours in French, unaligned)



Preprocessing of the waveform with spectrogram using STFT(win_size=25ms, win_step=15ms), with the transcript “Rue Wolfgang Doeblin, zéro huit, six cents Givet”

Note: you can contribute the open shared common voice dataset in one of the **60 languages** by either recording or validating (Ardila et al., 2020)!

Example model : end-to-end trainable Baidu DeepSpeech (v1,v2) (Hannun et al., 2014),(Amodei et al., 2015). See also the implementation of Mozilla DeepSpeech v2.

Note some authors introduced end-to-end trainable networks from the raw waveforms (Zeghidour, Usunier, Synnaeve, Collobert, & Dupoux, 2018).

DEEPSPEECH : ASR WITH CONV-BIGRU-CTC

Introduced in (Amodei et al., 2015) on English and Mandarin.

The English architecture involves :

- mel spectrograms
- 3 2D-convolutions in time and frequency with cliped ReLu
- 7 bidirectional GRU(1280)
- 1 FC with Batch Norm
- CTC loss, decoding with beam size 500

35 M. parameters

The training :

- dataset sizes up from 120hours up to 12.000 hours
- SGD with Nesterov momentum (0.99)
- gradient clipping to 400
- learning rate ($\approx 1e-4$), downscaled by 0.8 at every epoch
- during the 1st epoch, the samples are ordered by increasing duration (SortaGrad)
- data augmentation with noise added to the speech

BIBLIOGRAPHY

REFERENCES

Rather check the full online document references.pdf

Amodei, D., Anubhai, R., Battenberg, E., Case, C., Casper, J., Catanzaro, B., … Zhu, Z. (2015). Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *arXiv:1512.02595 [Cs]*. Retrieved from http://arxiv.org/abs/1512.02595

Ardila, R., Branson, M., Davis, K., Henretty, M., Kohler, M., Meyer, J., … Weber, G. (2020). Common Voice: A Massively-Multilingual Speech Corpus. *arXiv:1912.06670 [Cs]*. Retrieved from http://arxiv.org/abs/1912.06670

Arjovsky, M., Shah, A., & Bengio, Y. (2016). Unitary Evolution Recurrent Neural Networks. *arXiv:1511.06464 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1511.06464

Arthur, D., & Vassilvitskii, S. (2007). K-means++: The advantages of careful seeding. In *In Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*.

Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer Normalization. *arXiv:1607.06450 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1607.06450

Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. *arXiv:1206.5533 [Cs]*. Retrieved from http://arxiv.org/abs/1206.5533

Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2006). Greedy Layer-Wise Training of Deep Networks. In (p. 8).

Bjorck, N., Gomes, C. P., Selman, B., & Weinberger, K. Q. (2018). Understanding Batch Normalization. In *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)* (p. 12).

Bodla, N., Singh, B., Chellappa, R., & Davis, L. S. (2017). Soft-NMS – Improving Object Detection With One Line of Code. *arXiv:1704.04503 [Cs]*. Retrieved from http://arxiv.org/abs/1704.04503

Broomhead, D., & Lowe, D. (1988). Multivariable Functional Interpolation and Adaptive Networks. *Complex Systems*, *2*, 321–355.

Cho, K., Merrienboer, B. van, Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *arXiv:1406.1078 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1406.1078

Choromanska, A., Henaff, M., Mathieu, M., Arous, G. B., & LeCun, Y. (2015). The Loss Surfaces of Multilayer Networks. In *Roceedings of the 18thInternational Con-ference on Artificial Intelligence and Statistics* (p. 13).

Ciresan, D., Meier, U., & Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition* (pp. 3642–3649). Providence, RI: IEEE. https://doi.org/10.1109/CVPR.2012.6248110

Clevert, D.-A., Unterthiner, T., & Hochreiter, S. (2016). Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *arXiv:1511.07289 [Cs]*. Retrieved from http://arxiv.org/abs/1511.07289

Collobert, R., Puhrsch, C., & Synnaeve, G. (2016). Wav2Letter: An End-to-End ConvNet-based Speech Recognition System. *arXiv:1609.03193 [Cs]*. Retrieved from http://arxiv.org/abs/1609.03193

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, *2*(4), 303–314. https://doi.org/10.1007/BF02551274

Dauphin, Y. N., Fan, A., Auli, M., & Grangier, D. (2017). Language modeling with gated convolutional networks. Retrieved from http://arxiv.org/abs/1612.08083

Dauphin, Y. N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., & Bengio, Y. (2014). Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *Advances in Neural Information Processing Systems*, *27*, 2933–2941. Retrieved from https://papers.nips.cc/paper/2014/hash/17e23e50bedc63b4095e3d8204ce063b-Abstract.html

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [Cs]*. Retrieved from http://arxiv.org/abs/1810.04805

Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization, *12*, 2121–2159.

Elman, J. L. (1990). Finding Structure in Time. *Cognitive Science*, *14*(2), 179–211. https://doi.org/10.1207/s15516709cog1402_1

Fritzke, B. (1994). A growing neural gas network learns topologies. In *Proceedings of the 7th International Conference on Neural Information Processing Systems* (pp. 625–632). Cambridge, MA, USA: MIT Press.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, *36*(4), 193–202. https://doi.org/10.1007/BF00344251

Gers, F. A., Schmidhuber, J. A., & Cummins, F. A. (2000). Learning to forget: Continual prediction with lstm. *Neural Comput.*, *12*(10), 2451–2471. https://doi.org/10.1162/089976600300015015

Girshick, R. (2015). Fast R-CNN. In *2015 IEEE International Conference on Computer Vision (ICCV)* (pp. 1440–1448). https://doi.org/10.1109/ICCV.2015.169

Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2014). Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv:1311.2524 [Cs]*. Retrieved from http://arxiv.org/abs/1311.2524

Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Roceedings of the13thInternational Conferenceon Artificial Intelligence and Statistics (AISTATS) 2010* (p. 8).

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Graves, A., Fernández, S., Gomez, F., & Schmidhuber, J. (2006). Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks. In *Proceedings of the 23rd international conference on machine learning* (pp. 369–376). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/1143844.1143891

Graves, A., Mohamed, A.-r., & Hinton, G. (2013). Speech Recognition with Deep Recurrent Neural Networks. *arXiv:1303.5778 [Cs]*. Retrieved from http://arxiv.org/abs/1303.5778

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., … Hassabis, D. (2016). Hybrid computing using a neural network with dynamic external memory. *Nature*, *538*(7626), 471–476. https://doi.org/10.1038/nature20101

Greff, K., Srivastava, R. K., Koutník, J., Steunebrink, B. R., & Schmidhuber, J. (2017). LSTM: A search space odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, *28*(10), 2222–2232. https://doi.org/10.1109/TNNLS.2016.2582924

Griewank, A. (2012). Who Invented the Reverse Mode of Differentiation? *Documenta Mathematica*, 12.

Griewank, A., & Walther, A. (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (2 édition). Philadelphia, PA: Society for Industrial; Applied Mathematics.

Hannun, A., Case, C., Casper, J., Catanzaro, B., Diamos, G., Elsen, E., … Ng, A. Y. (2014). Deep Speech: Scaling up end-to-end speech recognition. *arXiv:1412.5567 [Cs]*. Retrieved from http://arxiv.org/abs/1412.5567

Hastad, J. (1986). Almost optimal lower bounds for small depth circuits. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing* (pp. 6–20). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/12130.12132

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. *arXiv:1502.01852 [Cs]*. Retrieved from http://arxiv.org/abs/1502.01852

He, K., Zhang, X., Ren, S., & Sun, J. (2016a). Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 770–778). Las Vegas, NV, USA: IEEE. https://doi.org/10.1109/CVPR.2016.90

He, K., Zhang, X., Ren, S., & Sun, J. (2016b). Identity Mappings in Deep Residual Networks. *arXiv:1603.05027 [Cs]*. Retrieved from http://arxiv.org/abs/1603.05027

Henaff, M., Szlam, A., & LeCun, Y. (2016). Recurrent Orthogonal Networks and Long-Memory Tasks, 9.

Hinton, G. E. (2006). Reducing the Dimensionality of Data with Neural Networks. *Science*, *313*(5786), 504–507. https://doi.org/10.1126/science.1127647

Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the Knowledge in a Neural Network. *arXiv:1503.02531 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1503.02531

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, *9*(8), 1735–1780.

Hodgkin, A. L., & Huxley, A. F. (1952). A quantitative description of membrane current and its application to conduction and excitation in nerve. *The Journal of Physiology*, *117*(4), 500–544. Retrieved from https://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413/

Hoffer, E., Hubara, I., & Soudry, D. (2017). Train longer, generalize better: Closing the generalization gap in large batch training of neural networks. *arXiv:1705.08741 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1705.08741

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural Networks*, *4*(2), 251–257. https://doi.org/10.1016/0893-6080(91)90009-T

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., … Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv:1704.04861 [Cs]*. Retrieved from http://arxiv.org/abs/1704.04861

Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., & Weinberger, K. Q. (2017). Snapshot ensembles: Train 1, get m for free. In (p. 14).

Huang, G., Liu, Z., Maaten, L. van der, & Weinberger, K. Q. (2018). Densely Connected Convolutional Networks. *arXiv:1608.06993 [Cs]*. Retrieved from http://arxiv.org/abs/1608.06993

Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *International Conference on Machine Learning* (pp. 448–456). PMLR. Retrieved from http://proceedings.mlr.press/v37/ioffe15.html

Jaderberg, M., Simonyan, K., & Zisserman, A. (2015). Spatial Transformer Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems* (p. 9).

Josef Hochreiter. (1991). *Untersuchungen zu dynamischen neuronalen Netzen* (PhD thesis). Retrieved from http://people.idsia.ch/~juergen/SeppHochreiter1991ThesisAdvisorSchmidhuber.pdf

Kam-Chuen Jim, Giles, C. L., & Horne, B. G. (1996). An analysis of noise in recurrent neural networks: Convergence and generalization. *IEEE Transactions on Neural Networks*, *7*(6), 1424–1438. https://doi.org/10.1109/72.548170

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2017). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *arXiv:1609.04836 [Cs, Math]*. Retrieved from http://arxiv.org/abs/1609.04836

Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. In *arXiv:1412.6980 [cs]*. Retrieved from http://arxiv.org/abs/1412.6980

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. https://doi.org/10.1145/3065386

Krueger, D., Maharaj, T., Kramár, J., Pezeshki, M., Ballas, N., Ke, N. R., … Pal, C. (2017). ZONEOUT: REGULARIZING RNNS BY RANDOMLY PRESERVING HIDDEN ACTIVATIONS, 11.

Le, Q. V., Jaitly, N., & Hinton, G. E. (2015). A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. *arXiv:1504.00941 [Cs]*. Retrieved from http://arxiv.org/abs/1504.00941

LeCun, Y. A., Bottou, L., Orr, G. B., & Müller, K.-R. (1998). Efficient BackProp. In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural Networks: Tricks of the Trade: Second Edition* (pp. 9–48). Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-35289-8_3

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, *1*(4), 541–551. https://doi.org/10.1162/neco.1989.1.4.541

Li, J., Lavrukhin, V., Ginsburg, B., Leary, R., Kuchaiev, O., Cohen, J. M., … Gadde, R. T. (2019). Jasper: An End-to-End Convolutional Neural Acoustic Model. *arXiv:1904.03288 [Cs, Eess]*. Retrieved from http://arxiv.org/abs/1904.03288

Li, Y., Wei, C., & Ma, T. (2019). Towards Explaining the Regularization Effect of Initial Large Learning Rate in Training Neural Networks. In *NIPS 2019* (p. 12).

Lin, M., Chen, Q., & Yan, S. (2014). Network In Network. *arXiv:1312.4400 [Cs]*. Retrieved from http://arxiv.org/abs/1312.4400

Lin, T.-Y., Dollar, P., Girshick, R., He, K., Hariharan, B., & Belongie, S. (2017). Feature Pyramid Networks for Object Detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 936–944). Honolulu, HI: IEEE. https://doi.org/10.1109/CVPR.2017.106

Loshchilov, I., & Hutter, F. (2017). SGDR: Stochastic Gradient Descent with Warm Restarts. In *arXiv:1608.03983 [cs, math]*. Retrieved from http://arxiv.org/abs/1608.03983

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013). Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proceedings of the 30th International Conference on Machine Learning (ICML13)* (p. 6).

Maclin, R., & Shavlik, J. W. (1995). Combining the predictions of multiple classifiers: Using competitive learning to initialize neural networks. In *Proceedings of the 14th international joint conference on Artificial intelligence - Volume 1* (pp. 524–530). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, *5*(4), 115–133. https://doi.org/10.1007/BF02478259

Minksy, M., & Papert, S. (1969). *Perceptrons: An Introduction to Computational Geometry*.

Montufar, G. F., Pascanu, R., Cho, K., & Bengio, Y. (2014). On the Number of Linear Regions of Deep Neural Networks. In *Advances in Neural Information Processing Systems* (Vol. 27, pp. 2924–2932). Retrieved from https://papers.nips.cc/paper/2014/hash/109d2dd3608f669ca17920c511c2a41e-Abstract.html

Moon, T., Choi, H., Lee, H., & Song, I. (2015). RNNDROP: A novel dropout for rnns in asr. In *2015 ieee workshop on automatic speech recognition and understanding (asru)* (pp. 65–70). https://doi.org/10.1109/ASRU.2015.7404775

Müller, R., Kornblith, S., & Hinton, G. (2020). When Does Label Smoothing Help? *arXiv:1906.02629 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1906.02629

Nair, V., & Hinton, G. E. (2010). Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning* (pp. 807–814). Madison, WI, USA: Omnipress.

Nielsen, M. A. (2015). *Neural Networks and Deep Learning*. Retrieved from http://neuralnetworksanddeeplearning.com

Olah, C. (2015). Calculus on computational graphs: Backpropagation.

Park, J., & Sandberg, I. W. (1991). Universal Approximation Using Radial-Basis-Function Networks. *Neural Computation*, *3*(2), 246–257. https://doi.org/10.1162/neco.1991.3.2.246

Pascanu, R., Dauphin, Y. N., Ganguli, S., & Bengio, Y. (2014). On the saddle point problem for non-convex optimization. *arXiv:1405.4604 [Cs]*. Retrieved from http://arxiv.org/abs/1405.4604

Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. In *Proceedings of the 30thInternational Conference on Machine Learning* (p. 9).

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., … Lerer, A. (2017). Automatic differentiation in PyTorch. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (p. 4).

Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. *arXiv:1506.02640 [Cs]*. Retrieved from http://arxiv.org/abs/1506.02640

Ren, S., He, K., Girshick, R., & Sun, J. (2016). Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *arXiv:1506.01497 [Cs]*. Retrieved from http://arxiv.org/abs/1506.01497

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, *65*(6), 386–408. https://doi.org/10.1037/h0042519

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L.-C. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (pp. 4510–4520). Salt Lake City, UT: IEEE. https://doi.org/10.1109/CVPR.2018.00474

Santurkar, S., Tsipras, D., Ilyas, A., & Ma, A. (2018). How Does Batch Normalization Help Optimization? In *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)* (p. 11).

Schmidhuber, J. (1992). Learning Complex, Extended Sequences Using the Principle of History Compression. *Neural Computation*, *4*(2), 234–242. https://doi.org/10.1162/neco.1992.4.2.234

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

Schuster, M., & Paliwal, K. (1997). Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, *45*(11), 2673–2681. https://doi.org/10.1109/78.650093

Schwenker, F., Kestler, H. A., & Palm, G. (2001). Three learning phases for radial-basis-function networks. *Neural Networks*, *14*(4-5), 439–458. Retrieved from http://dblp.uni-trier.de/db/journals/nn/nn14.html#SchwenkerKP01

Simonyan, K., Vedaldi, A., & Zisserman, A. (2014). Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. *arXiv:1312.6034 [Cs]*. Retrieved from http://arxiv.org/abs/1312.6034

Simonyan, K., & Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. In *arXiv:1409.1556 [cs]*. Retrieved from http://arxiv.org/abs/1409.1556

Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay. *arXiv:1803.09820 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1803.09820

Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2015). Striving for Simplicity: The All Convolutional Net. *arXiv:1412.6806 [Cs]*. Retrieved from http://arxiv.org/abs/1412.6806

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, *15*(56), 1929–1958. Retrieved from http://jmlr.org/papers/v15/srivastava14a.html

Srivastava, R. K., Greff, K., & Schmidhuber, J. (2015). Training Very Deep Networks, 9.

Sutskever, I. (2013). *Training recurrent neural networks* (PhD thesis). University of Toronto, CAN.

Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *ICML* (p. 14).

Sutskever, I., Martens, J., & Hinton, G. (2011). Generating Text with Recurrent Neural Networks, 8.

Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to Sequence Learning with Neural Networks. *arXiv:1409.3215 [Cs]*. Retrieved from http://arxiv.org/abs/1409.3215

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., … Rabinovich, A. (2014). Going Deeper with Convolutions. *arXiv:1409.4842 [Cs]*. Retrieved from http://arxiv.org/abs/1409.4842

Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., & Wojna, Z. (2015). Rethinking the Inception Architecture for Computer Vision. *arXiv:1512.00567 [Cs]*. Retrieved from http://arxiv.org/abs/1512.00567

Tan, M., & Le, Q. V. (2020). EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *arXiv:1905.11946 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1905.11946

Vinyals, O., Toshev, A., Bengio, S., & Erhan, D. (2015). Show and Tell: Lessons learned from the 2015 MSCOCO Image Captioning Challenge. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *39*(4), 652–663. https://doi.org/10.1109/TPAMI.2016.2587640

Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. J. (1989). Phoneme recognition using time-delay neural networks. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *37*(3), 328–339. https://doi.org/10.1109/29.21701

Werbos, P. (1981). Application of advances in nonlinear sensitivity analysis. In *Proc. Of the 10th IFIP conference* (pp. 762–770).

Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, *78*(10), 1550–1560. Retrieved from https://www.bibsonomy.org/bibtex/25ea3485ce75778e802cd8466cd7ffa69/joachimagne

Widrow, B., & Hoff, M. E. (1962). Associative Storage and Retrieval of Digital Information in Networks of Adaptive “Neurons”. In E. E. Bernard & M. R. Kare (Eds.), *Biological Prototypes and Synthetic Systems: Volume 1 Proceedings of the Second Annual Bionics Symposium sponsored by Cornell University and the General Electric Company, Advanced Electronics Center, held at Cornell University, August 30–September 1, 1961* (pp. 160–160). Boston, MA: Springer US. https://doi.org/10.1007/978-1-4684-1716-6_25

Williams, R. J., & Peng, J. (1990). An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, *2*(4), 490–501. https://doi.org/10.1162/neco.1990.2.4.490

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., … Dean, J. (2016). Google’s Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *arXiv:1609.08144 [Cs]*. Retrieved from http://arxiv.org/abs/1609.08144

Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhutdinov, R., … Bengio, Y. (2016). Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. *arXiv:1502.03044 [Cs]*. Retrieved from http://arxiv.org/abs/1502.03044

Yu, F., & Koltun, V. (2016). Multi-Scale Context Aggregation by Dilated Convolutions. *arXiv:1511.07122 [Cs]*. Retrieved from http://arxiv.org/abs/1511.07122

Ze, H., Senior, A., & Schuster, M. (2013). Statistical parametric speech synthesis using deep neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing* (pp. 7962–7966). Vancouver, BC, Canada: IEEE. https://doi.org/10.1109/ICASSP.2013.6639215

Zeghidour, N., Usunier, N., Synnaeve, G., Collobert, R., & Dupoux, E. (2018). End-to-End Speech Recognition from the Raw Waveform. In *Interspeech 2018* (pp. 781–785). ISCA. https://doi.org/10.21437/Interspeech.2018-2414

Zeiler, M. D. (2012). ADADELTA: An Adaptive Learning Rate Method. *arXiv:1212.5701 [Cs]*. Retrieved from http://arxiv.org/abs/1212.5701

Zeiler, M. D., & Fergus, R. (2014). Visualizing and Understanding Convolutional Networks. In D. Fleet, T. Pajdla, B. Schiele, & T. Tuytelaars (Eds.), *Computer Vision – ECCV 2014* (pp. 818–833). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-10590-1_53

Zhang, J., & Mitliagkas, I. (2018). YellowFin and the Art of Momentum Tuning. *arXiv:1706.03471 [Cs, Stat]*. Retrieved from http://arxiv.org/abs/1706.03471