

Deep Learning

Spring 2019

Prof. Gilles Louppe
g.louppe@uliege.be

Logistics

This course is given by:

- Theory: Prof. Gilles Louppe (g.louppe@uliege.be)
- Projects and guidance:
 - Joeri Hermans (joeri.hermans@doct.uliege.be)
 - Matthia Sabatelli (m.sabatelli@uliege.be)
 - Antoine Wehenkel (antoine.wehenkel@uliege.be)

Feel free to contact any of us for help!



Lectures

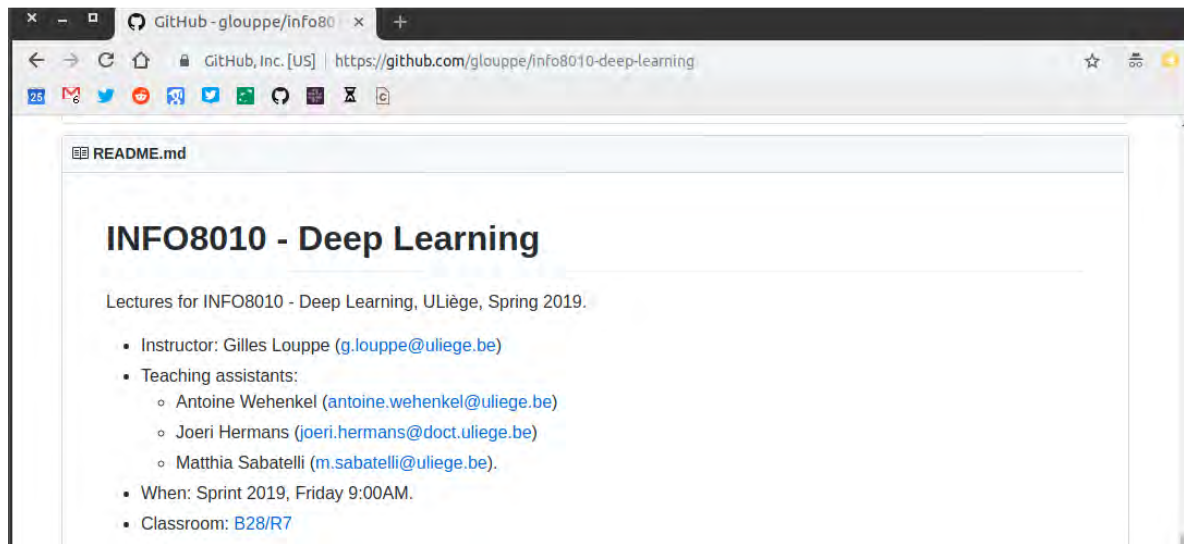
- Theoretical lectures
- Tutorials
- Q&A sessions

Materials

Slides are available at github.com/glouppe/info8010-deep-learning.

- In HTML and in PDFs.
- Posted online the day before the lesson (hopefully).

Some lessons are partially adapted from "[EE-559 Deep Learning](#)" by Francois Fleuret at EPFL.



Textbook

None!

Resources

- [Awesome Deep Learning](#)
- [Awesome Deep Learning papers](#)

AI at ULiège

This course is part of the many other courses available at ULiège and related to AI, including:

- INFO8006: Introduction to Artificial Intelligence
- ELEN0062: Introduction to Machine Learning
- **INFO8010: Deep Learning** ← you are there
- INFO8003: Optimal decision making for complex problems
- INFO8004: Advanced Machine Learning
- INFO0948: Introduction to Intelligent Robotics
- INFO0049: Knowledge representation
- ELEN0016: Computer vision
- DROI8031: Introduction to the law of robots

Outline

(Tentative and subject to change!)

- Lecture 1: Fundamentals of machine learning
- Lecture 2: Neural networks
- Lecture 3: Convolutional neural networks
- Lecture 4: Training neural networks
- Lecture 5: Recurrent neural networks
- Lecture 6: Auto-encoders and generative models
- Lecture 7: Generative adversarial networks
- Lecture 8: Uncertainty
- Lecture 9: Adversarial attacks and defenses

Philosophy

Thorough and detailed

- Understand the foundations and the landscape of deep learning.
- Be able to write from scratch, debug and run (some) deep learning algorithms.

State-of-the-art

- Introduction to materials new from research (\leq 5 years old).
- Understand some of the open questions and challenges in the field.

Practical

- Fun and challenging course project.

Projects

Reading assignment

Read, summarize and criticize a major scientific paper in deep learning.

Pick one of the following three papers:

- He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. arXiv:[1512.03385](https://arxiv.org/abs/1512.03385).
- Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., ... & De Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. arXiv:[1606.04474](https://arxiv.org/abs/1606.04474).
- Zhang, C., Bengio, S., Hardt, M., Recht, B., & Vinyals, O. (2016). Understanding deep learning requires rethinking generalization. arXiv:[1611.03530](https://arxiv.org/abs/1611.03530).

Deadline: **April 5, 2019 at 23:59**.

Project

Ambitious project of your choosing. Details to be announced soon.

Evaluation

- Exam (50%)
- Reading assignment (10%)
- Project (40%)

The reading assignment and the project are **mandatory** for presenting the exam.

Deep Learning

Lecture 1: Fundamentals of machine learning

Prof. Gilles Louppe
g.louppe@uliege.be

Today

Set the fundamentals of machine learning.

- Why learning?
- Applications and success
- Statistical learning
 - Supervised learning
 - Empirical risk minimization
 - Under-fitting and over-fitting
 - Bias-variance dilemma

Why learning?



What do you see?

How do we do that?!



Sheepdog or mop?



Chihuahua or muffin?

The automatic extraction of **semantic information** from raw signal is at the core of many applications, such as

- image recognition
- speech processing
- natural language processing
- robotic control
- ... and many others.

How can we **write a computer program** that implements that?

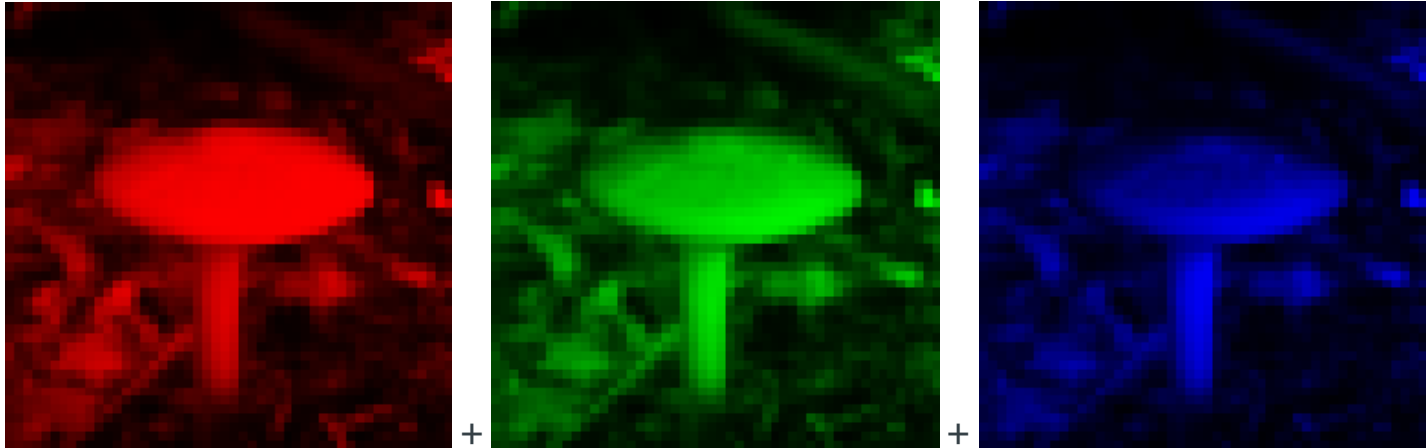
The (human) brain is so good at interpreting visual information that the **gap** between raw data and its semantic interpretation is difficult to assess intuitively:



This is a mushroom.



This is a mushroom.



This is a mushroom.

```

array([[0.03921569, 0.03529412, 0.02352941, 1.          ],
       [0.2509804 , 0.1882353 , 0.20392157, 1.          ],
       [0.4117647 , 0.34117648, 0.37254903, 1.          ],
       ...,
       [0.20392157, 0.23529412, 0.17254902, 1.          ],
       [0.16470589, 0.18039216, 0.12156863, 1.          ],
       [0.18039216, 0.18039216, 0.14117648, 1.          ]],

 [[0.1254902 , 0.11372549, 0.09411765, 1.          ],
  [0.2901961 , 0.2509804 , 0.24705882, 1.          ],
  [0.21176471, 0.2          , 0.20392157, 1.          ],
  ...,
  [0.1764706 , 0.24705882, 0.12156863, 1.          ],
  [0.10980392, 0.15686275, 0.07843138, 1.          ],
  [0.16470589, 0.20784314, 0.11764706, 1.          ]],

 [[0.14117648, 0.12941177, 0.10980392, 1.          ],
  [0.21176471, 0.1882353 , 0.16862746, 1.          ],
  [0.14117648, 0.13725491, 0.12941177, 1.          ],
  ...,
  [0.10980392, 0.15686275, 0.08627451, 1.          ],
  [0.0627451 , 0.08235294, 0.05098039, 1.          ],
  [0.14117648, 0.2          , 0.09803922, 1.          ]],

 ...,

```

This is a mushroom.

Extracting semantic information requires models of **high complexity**, which cannot be designed by hand.

However, one can write a program that **learns** the task of extracting semantic information.

Techniques used in practice consist of:

- defining a parametric model with high capacity,
- optimizing its parameters, by "making it work" on the training data.

This is similar to **biological systems** for which the model (e.g., brain structure) is DNA-encoded, and parameters (e.g., synaptic weights) are tuned through experiences.

Deep learning encompasses software technologies to **scale-up** to billions of model parameters and as many training examples.

Applications and success



YOLOv3



Watch later



Share



Real-time object detection (Redmon and Farhadi, 2018)



ICNet for Real-Time Semantic Segmentation ...



Watch later



Share



Segmentation (Hengshuang et al, 2017)



Realtime Multi-Person 2D Human Pose Estim...



Watch later



Share



Pose estimation (Cao et al, 2017)



Google DeepMind's Deep Q-learning playing A...



Watch later



Share



Reinforcement learning (Mnih et al, 2014)



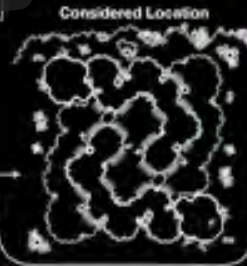
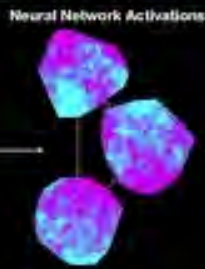
AlphaStar Agent Visualisation



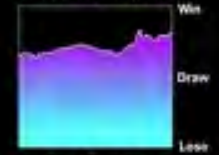
Watch later



Share



Outcome Prediction



Strategy games (Deepmind, 2016-2018)



NVIDIA Autonomous Car



Watch later



Share



Autonomous cars (NVIDIA, 2016)



So, that one change that particular breakthrough increased recognition rates by approximately thirty percent, that's a big deal.
That's the difference between going

Recognizability: 90%

Speech recognition, translation and synthesis (Microsoft, 2012)



Auto-captioning (2015)



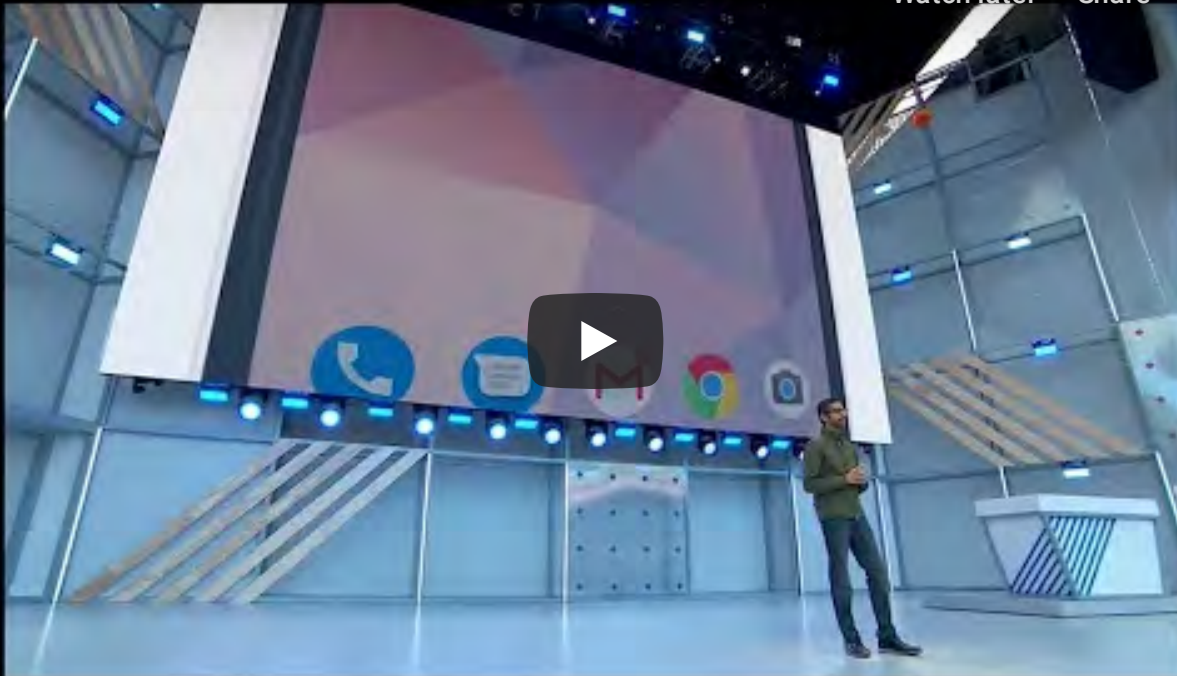
Google Assistant will soon be able to call rest...



Watch later



Share



Speech synthesis and question answering (Google, 2018)



A Style-Based Generator Architecture for Gen...



Watch later



Share

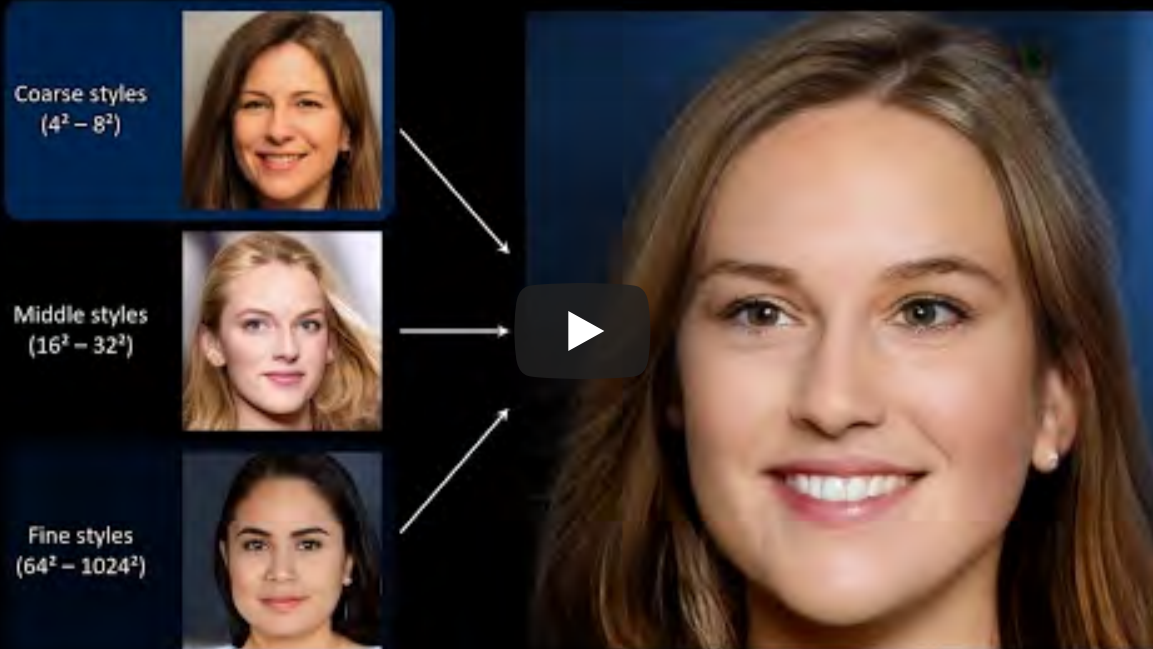


Image generation (Karras et al, 2018)



GTC Japan 2017 Part 9: AI Creates Original M...



Watch later



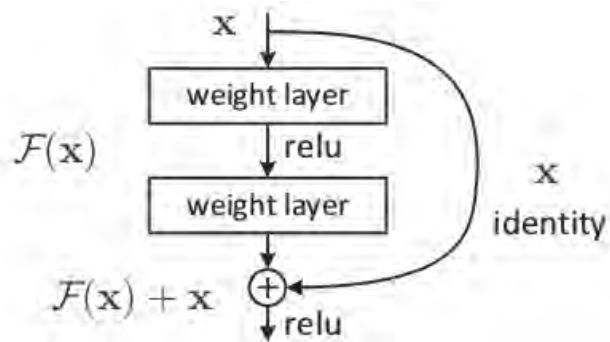
Share



Music composition (NVIDIA, 2017)

Why does it work now?

New algorithms



More data



Software



theano



PYTORCH

dmlc
mxnet



Faster compute engines



Building on the shoulders of giants

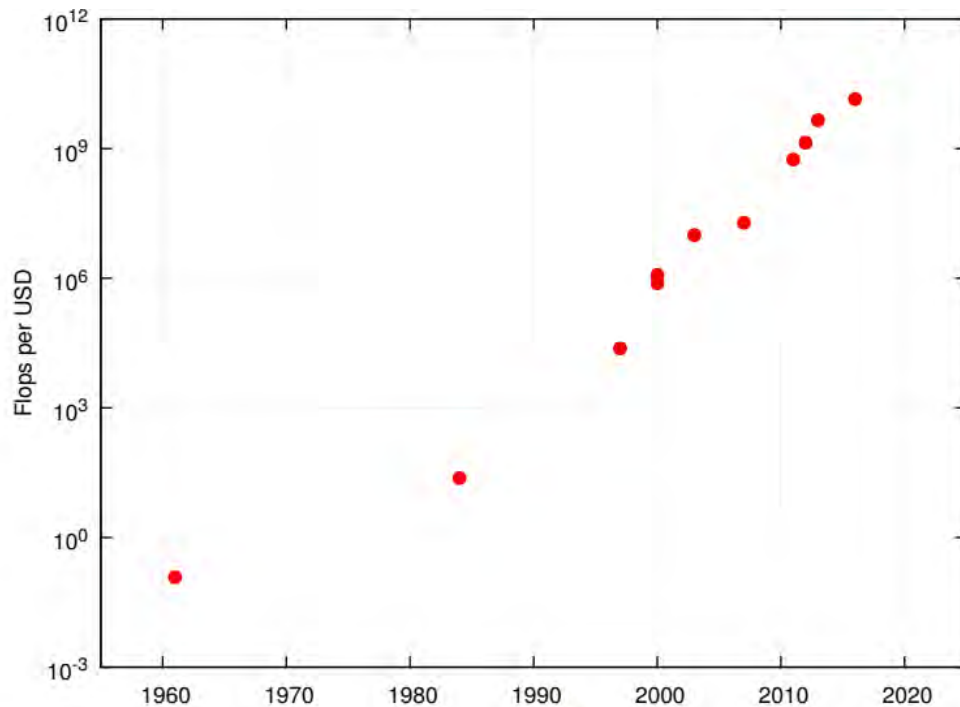
Five decades of research in machine learning provided

- a taxonomy of ML concepts (classification, generative models, clustering, kernels, linear embeddings, etc.),
- a sound statistical formalization (Bayesian estimation, PAC),
- a clear picture of fundamental issues (bias/variance dilemma, VC dimension, generalization bounds, etc.),
- a good understanding of optimization issues,
- efficient large-scale algorithms.

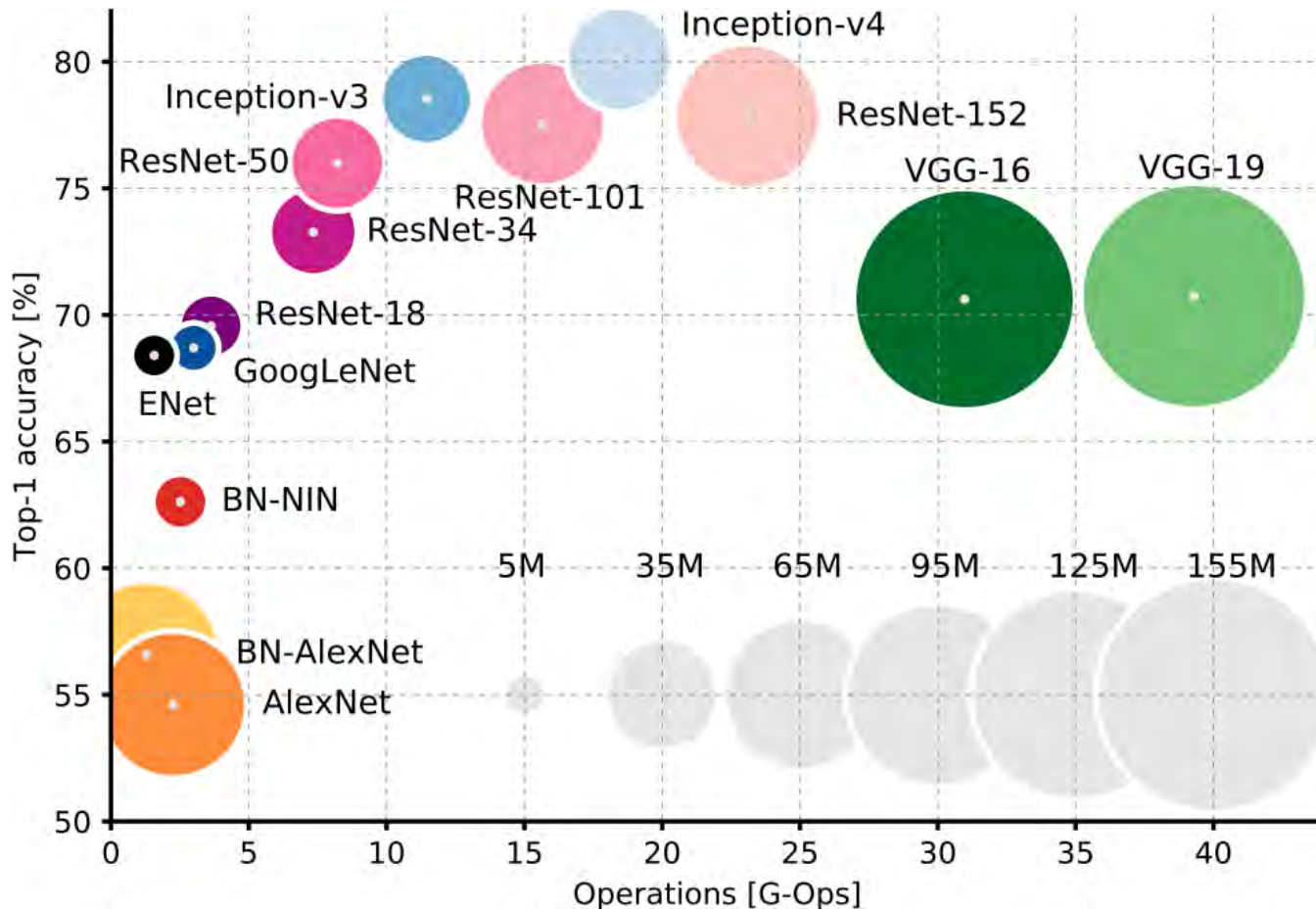
Deep learning

From a practical perspective, deep learning

- lessens the need for a deep mathematical grasp,
- makes the design of large learning architectures a system/software development task,
- allows to leverage modern hardware (clusters of GPUs),
- does not plateau when using more data,
- makes large trained networks a commodity.



	TFlops (10^{12})	Price	GFlops per \$
Intel i7-6700K	0.2	\$344	0.6
AMD Radeon R-7 240	0.5	\$55	9.1
NVIDIA GTX 750 Ti	1.3	\$105	12.3
AMD RX 480	5.2	\$239	21.6
NVIDIA GTX 1080	8.9	\$699	12.7



Statistical learning

Supervised learning

Consider an unknown joint probability distribution $P(X, Y)$.

Assume training data

$$(\mathbf{x}_i, y_i) \sim P(X, Y),$$

with $\mathbf{x}_i \in \mathcal{X}, y_i \in \mathcal{Y}, i = 1, \dots, N$.

- In most cases,
 - \mathbf{x}_i is a p -dimensional vector of features or descriptors,
 - y_i is a scalar (e.g., a category or a real value).
- The training data is generated i.i.d.
- The training data can be of any finite size N .
- In general, we do not have any prior information about $P(X, Y)$.

Inference

Supervised learning is usually concerned with the two following inference problems:

- **Classification:** Given $(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y} = \mathbb{R}^p \times \{1, \dots, C\}$, for $i = 1, \dots, N$, we want to estimate for any new \mathbf{x} ,

$$\arg \max_y P(Y = y | X = \mathbf{x}).$$

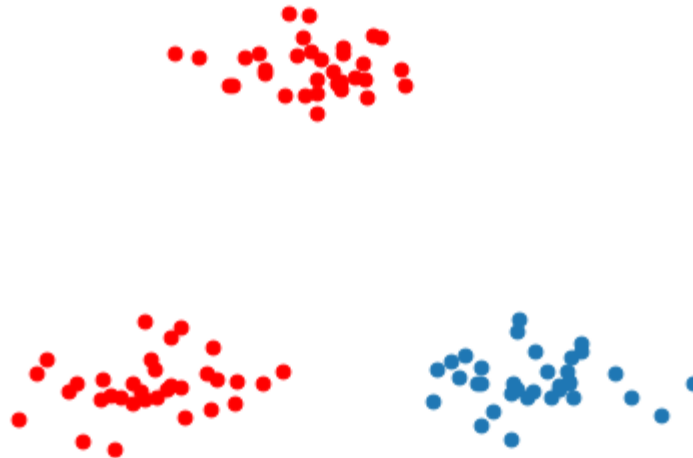
- **Regression:** Given $(\mathbf{x}_i, y_i) \in \mathcal{X} \times \mathcal{Y} = \mathbb{R}^p \times \mathbb{R}$, for $i = 1, \dots, N$, we want to estimate for any new \mathbf{x} ,

$$\mathbb{E}[Y | X = \mathbf{x}].$$

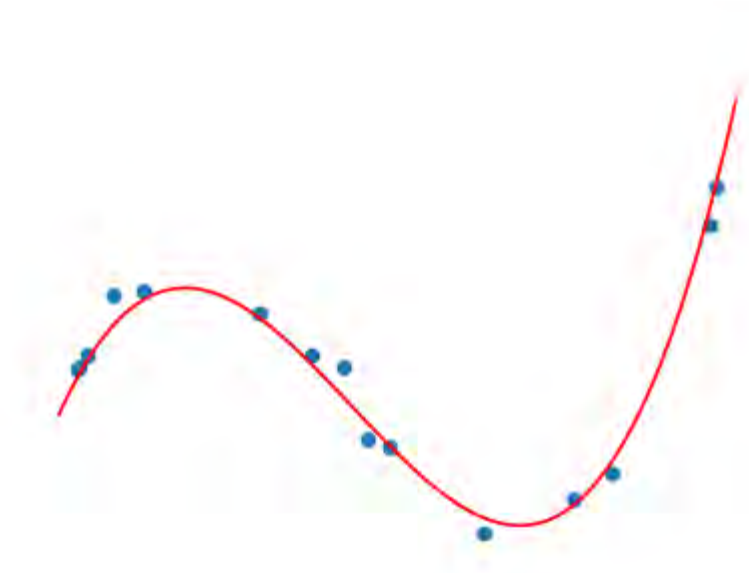
Or more generally, inference is concerned with the conditional estimation

$$P(Y = y | X = \mathbf{x})$$

for any new (\mathbf{x}, y) .



Classification consists in identifying a decision boundary between objects of distinct classes.



Regression aims at estimating relationships among (usually continuous) variables.

Empirical risk minimization

Consider a function $f : \mathcal{X} \rightarrow \mathcal{Y}$ produced by some learning algorithm. The predictions of this function can be evaluated through a loss

$$\ell : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R},$$

such that $\ell(y, f(\mathbf{x})) \geq 0$ measures how close the prediction $f(\mathbf{x})$ from y is.

Examples of loss functions

Classification: $\ell(y, f(\mathbf{x})) = \mathbf{1}_{y \neq f(\mathbf{x})}$

Regression: $\ell(y, f(\mathbf{x})) = (y - f(\mathbf{x}))^2$

Let \mathcal{F} denote the hypothesis space, i.e. the set of all functions f than can be produced by the chosen learning algorithm.

We are looking for a function $f \in \mathcal{F}$ with a small **expected risk** (or generalization error)

$$R(f) = \mathbb{E}_{(\mathbf{x}, y) \sim P(X, Y)} [\ell(y, f(\mathbf{x}))].$$

This means that for a given data generating distribution $P(X, Y)$ and for a given hypothesis space \mathcal{F} , the optimal model is

$$f_* = \arg \min_{f \in \mathcal{F}} R(f).$$

Unfortunately, since $P(X, Y)$ is unknown, the expected risk cannot be evaluated and the optimal model cannot be determined.

However, if we have i.i.d. training data $\mathbf{d} = \{(\mathbf{x}_i, y_i) | i = 1, \dots, N\}$, we can compute an estimate, the **empirical risk** (or training error)

$$\hat{R}(f, \mathbf{d}) = \frac{1}{N} \sum_{(\mathbf{x}_i, y_i) \in \mathbf{d}} \ell(y_i, f(\mathbf{x}_i)).$$

This estimate is **unbiased** and can be used for finding a good enough approximation of f_* . This results into the **empirical risk minimization principle**:

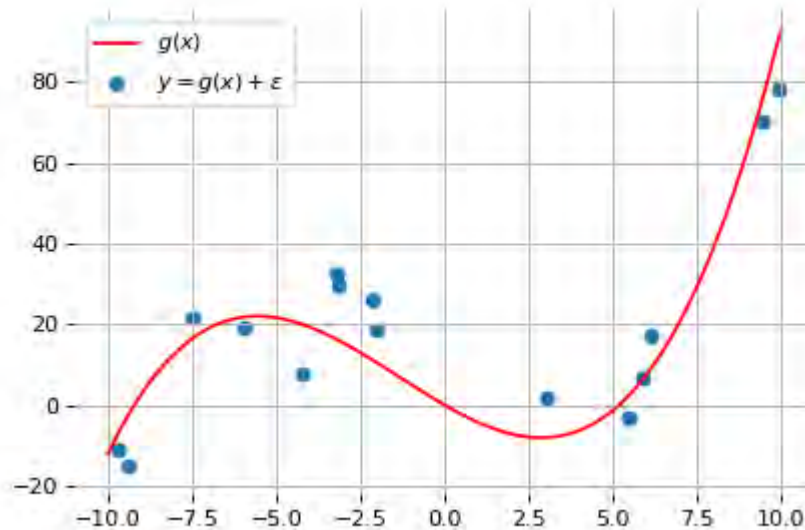
$$f_*^{\mathbf{d}} = \arg \min_{f \in \mathcal{F}} \hat{R}(f, \mathbf{d})$$

Most machine learning algorithms, including **neural networks**, implement empirical risk minimization.

Under regularity assumptions, empirical risk minimizers converge:

$$\lim_{N \rightarrow \infty} f_*^d = f_*$$

Polynomial regression



Consider the joint probability distribution $P(X, Y)$ induced by the data generating process

$$(x, y) \sim P(X, Y) \Leftrightarrow x \sim U[-10; 10], \epsilon \sim \mathcal{N}(0, \sigma^2), y = g(x) + \epsilon$$

where $x \in \mathbb{R}, y \in \mathbb{R}$ and g is an unknown polynomial of degree 3.

Our goal is to find a function f that makes good predictions on average over $P(X, Y)$.

Consider the hypothesis space $f \in \mathcal{F}$ of polynomials of degree 3 defined through their parameters $\mathbf{w} \in \mathbb{R}^4$ such that

$$\hat{y} \triangleq f(x; \mathbf{w}) = \sum_{d=0}^3 w_d x^d$$

For this regression problem, we use the squared error loss

$$\ell(y, f(x; \mathbf{w})) = (y - f(x; \mathbf{w}))^2$$

to measure how wrong the predictions are.

Therefore, our goal is to find the best value \mathbf{w}_* such

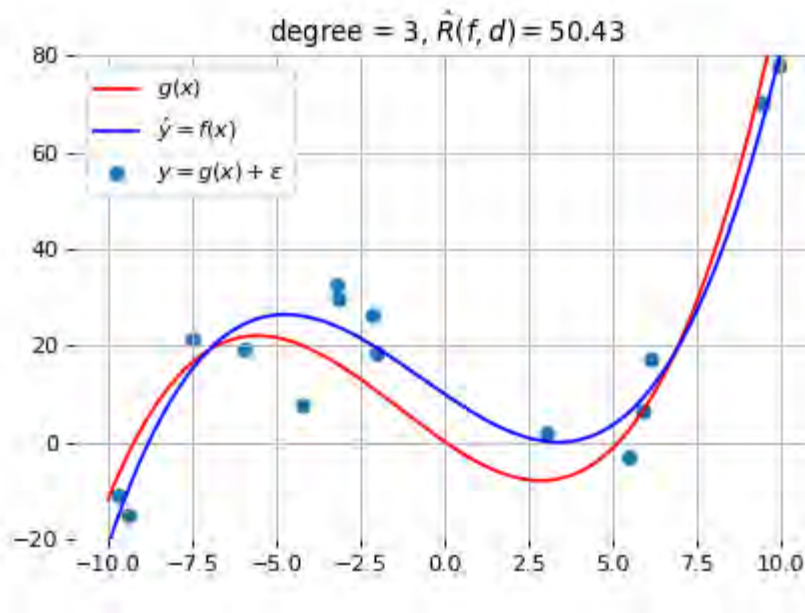
$$\begin{aligned} \mathbf{w}_* &= \arg \min_{\mathbf{w}} R(\mathbf{w}) \\ &= \arg \min_{\mathbf{w}} \mathbb{E}_{(x,y) \sim P(X,Y)} [(y - f(x; \mathbf{w}))^2] \end{aligned}$$

Given a large enough training set $\mathbf{d} = \{(x_i, y_i) | i = 1, \dots, N\}$, the empirical risk minimization principle tells us that a good estimate $\mathbf{w}_*^{\mathbf{d}}$ of \mathbf{w}_* can be found by minimizing the empirical risk:

$$\begin{aligned}
 \mathbf{w}_*^{\mathbf{d}} &= \arg \min_{\mathbf{w}} \hat{R}(\mathbf{w}, \mathbf{d}) \\
 &= \arg \min_{\mathbf{w}} \frac{1}{N} \sum_{(x_i, y_i) \in \mathbf{d}} (y_i - f(x_i; \mathbf{w}))^2 \\
 &= \arg \min_{\mathbf{w}} \frac{1}{N} \sum_{(x_i, y_i) \in \mathbf{d}} \left(y_i - \sum_{d=0}^3 w_d x_i^d \right)^2 \\
 &= \arg \min_{\mathbf{w}} \frac{1}{N} \left\| \underbrace{\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_N \end{pmatrix}}_{\mathbf{y}} - \underbrace{\begin{pmatrix} x_1^0 & \dots & x_1^3 \\ x_2^0 & \dots & x_2^3 \\ \dots & & \dots \\ x_N^0 & \dots & x_N^3 \end{pmatrix}}_{\mathbf{X}} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{pmatrix} \right\|^2
 \end{aligned}$$

This is **ordinary least squares** regression, for which the solution is known analytically:

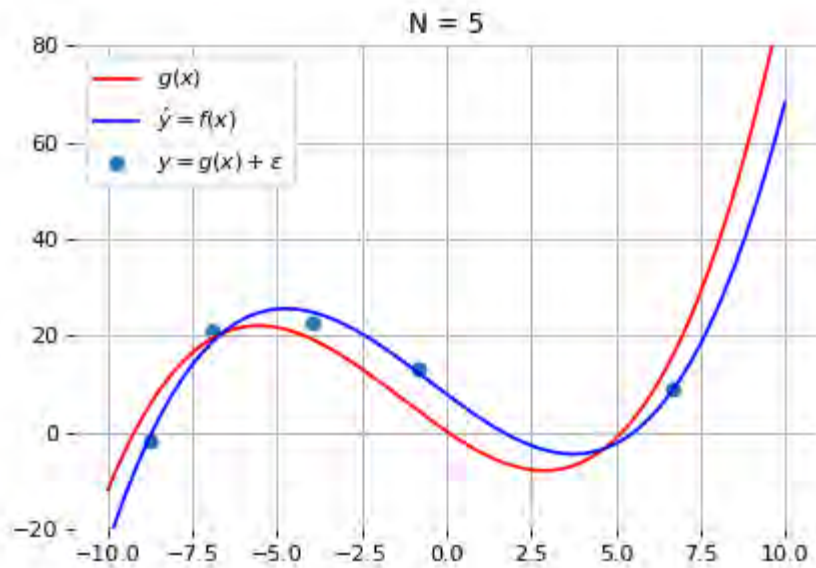
$$\mathbf{w}_*^d = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}$$

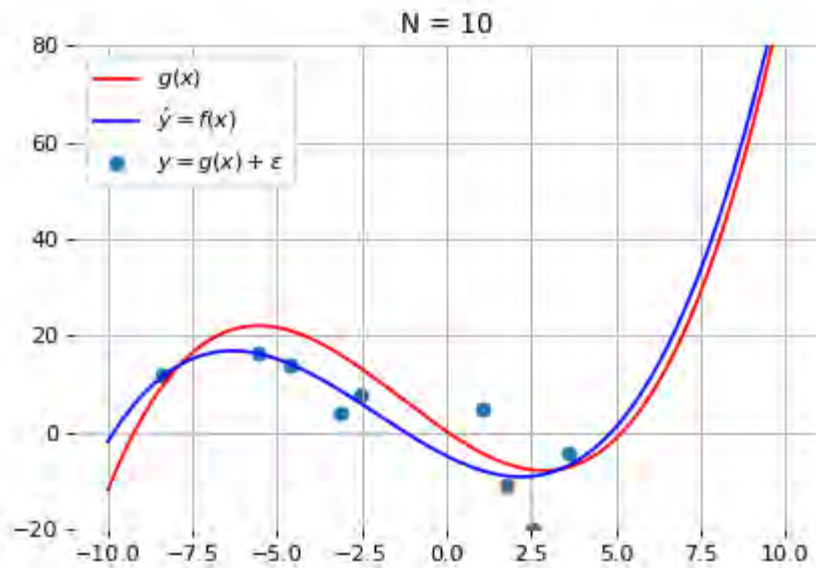


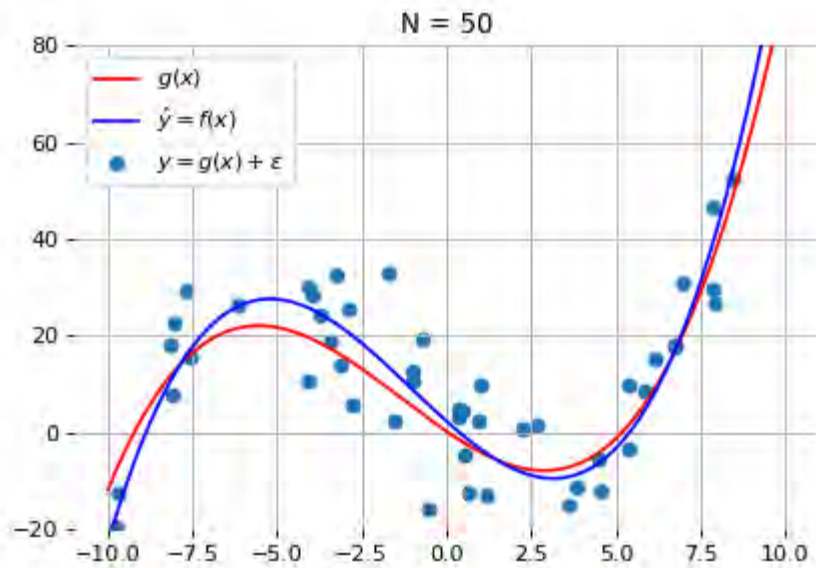
The expected risk minimizer \mathbf{w}_* within our hypothesis space is g itself.

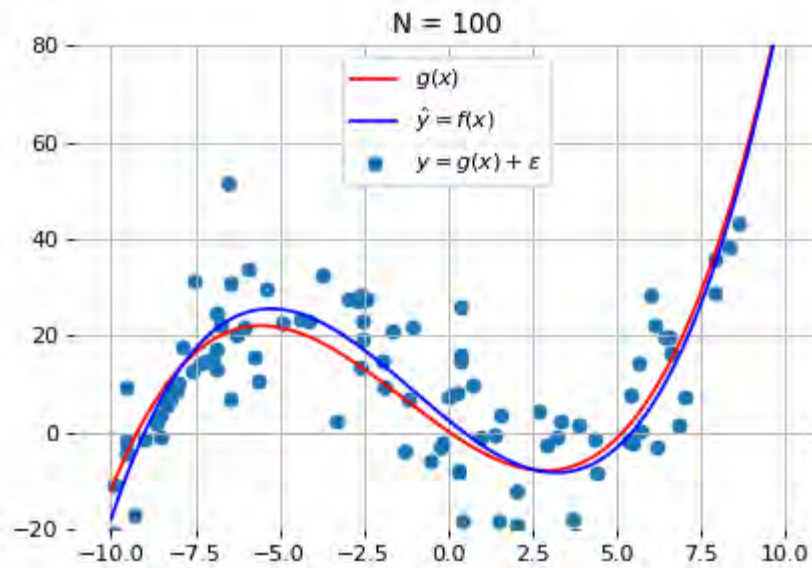
Therefore, on this toy problem, we can verify that

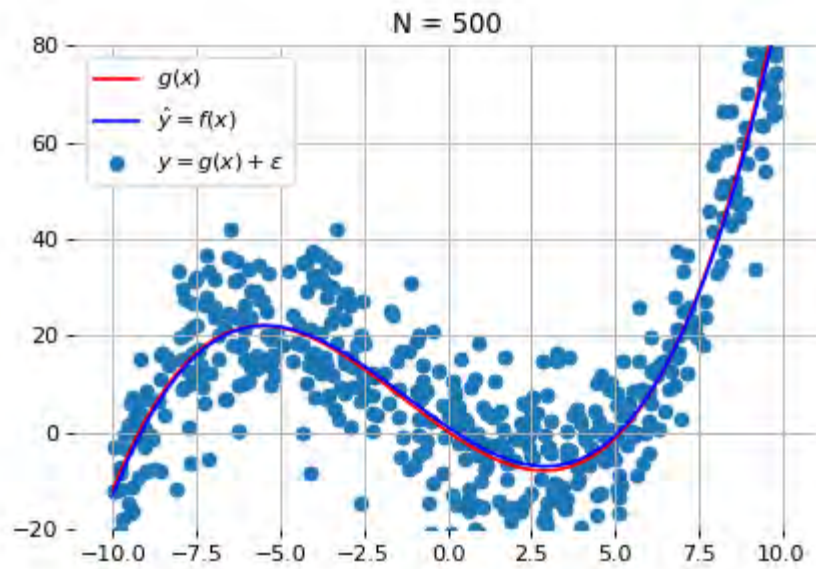
$$f(x; \mathbf{w}_*^d) \rightarrow f(x; \mathbf{w}_*) = g(x) \text{ as } N \rightarrow \infty.$$





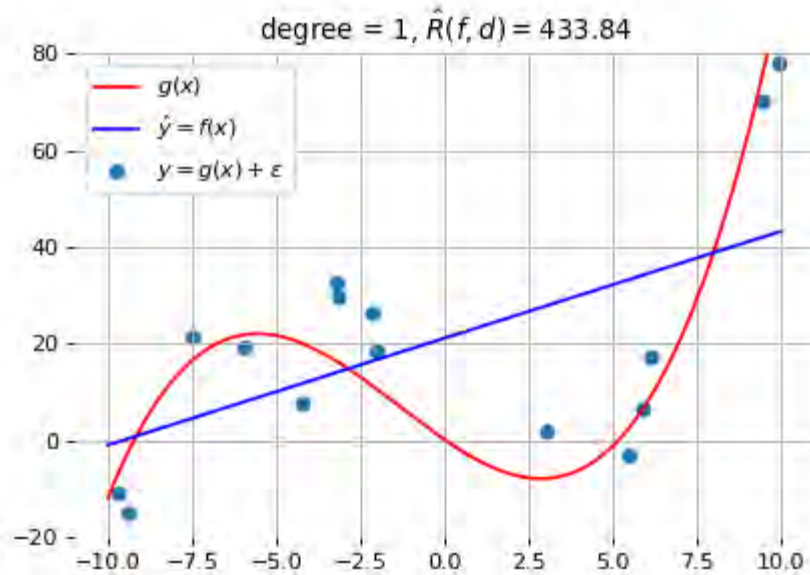




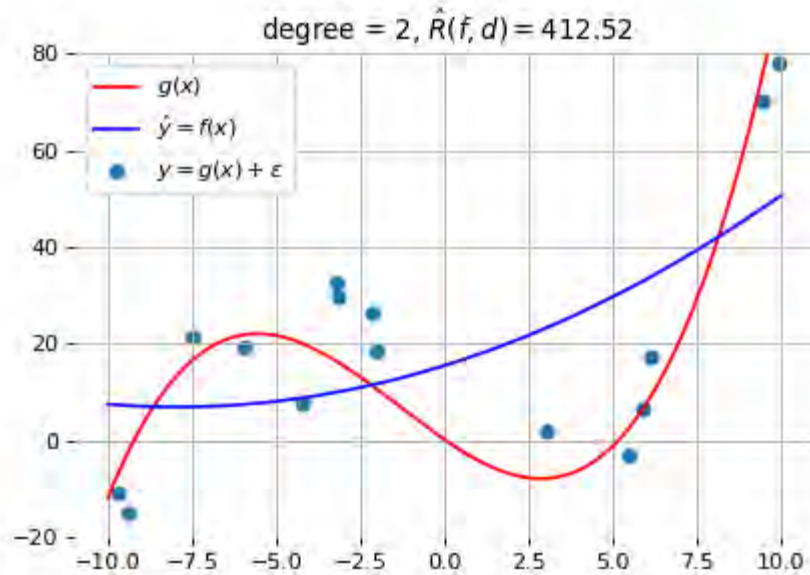


Under-fitting and over-fitting

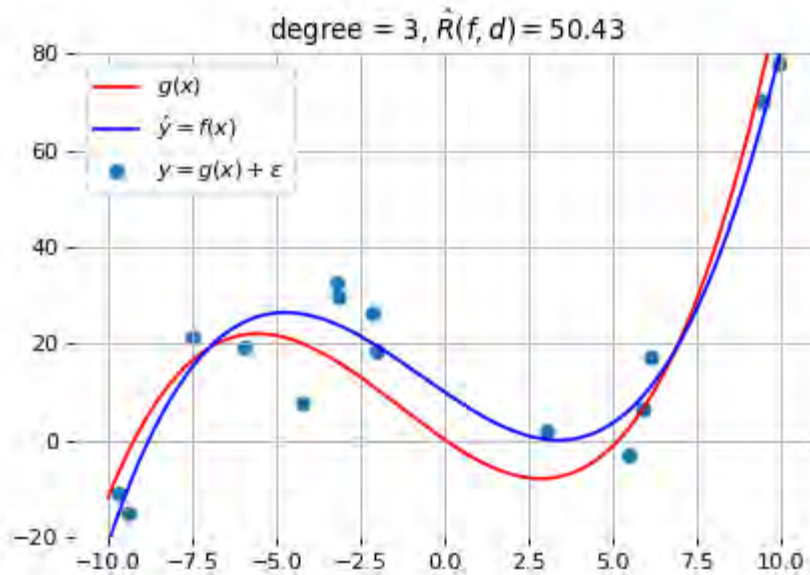
What if we consider a hypothesis space \mathcal{F} in which candidate functions f are either too "simple" or too "complex" with respect to the true data generating process?



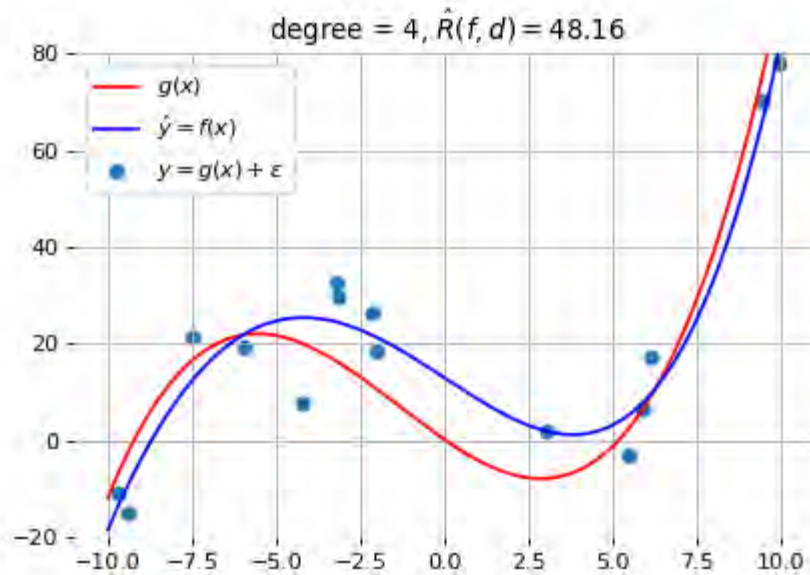
\mathcal{F} = polynomials of degree 1



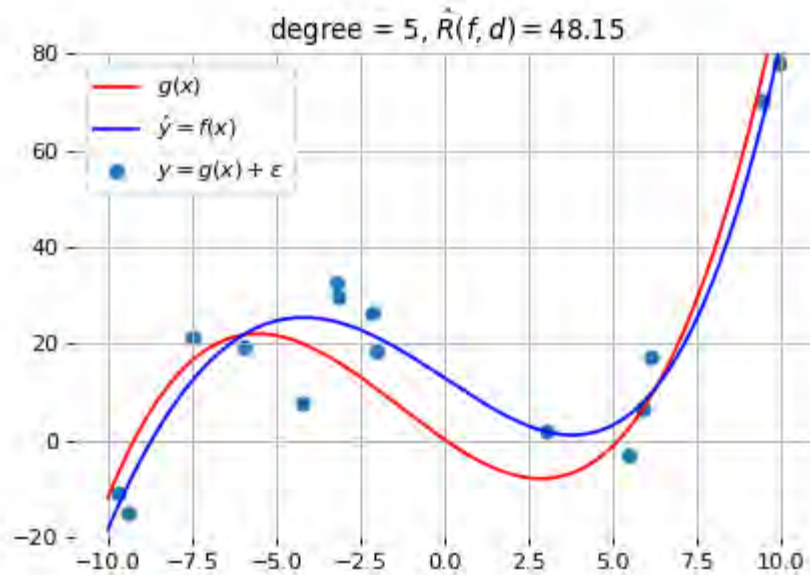
\mathcal{F} = polynomials of degree 2



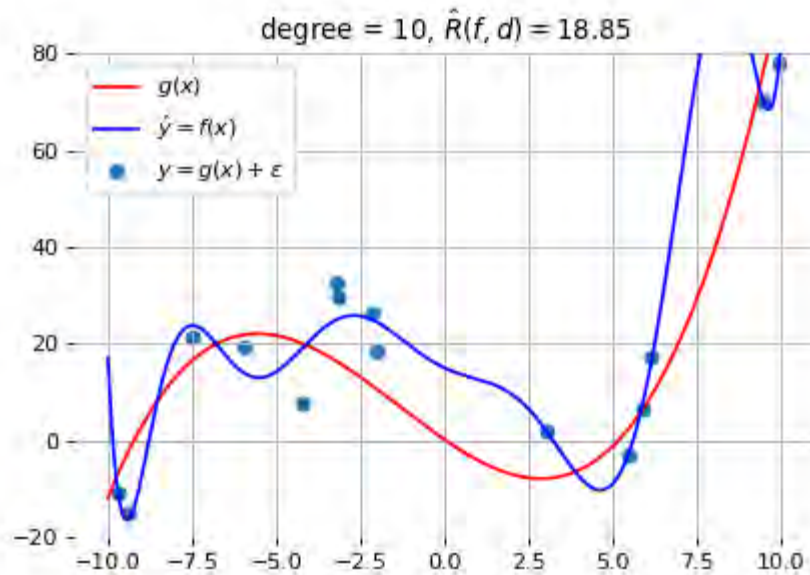
\mathcal{F} = polynomials of degree 3



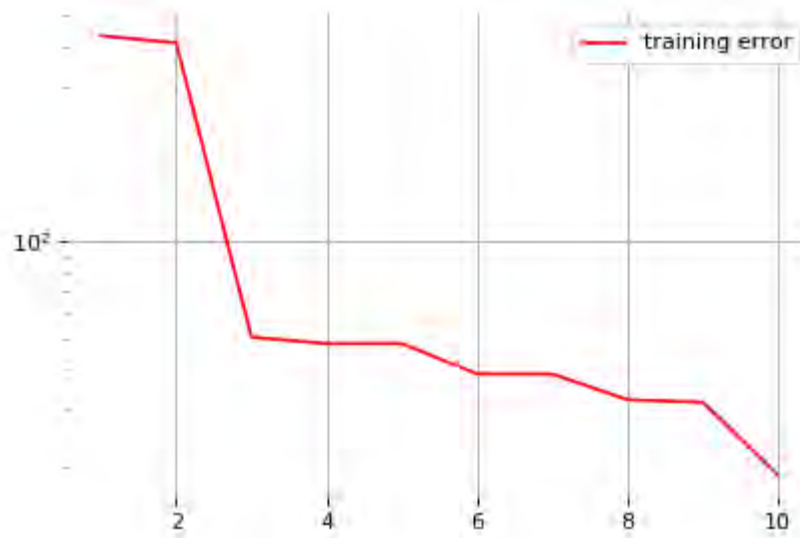
\mathcal{F} = polynomials of degree 4



\mathcal{F} = polynomials of degree 5



\mathcal{F} = polynomials of degree 10



Degree d of the polynomial VS. error.

Let $\mathcal{Y}^{\mathcal{X}}$ be the set of all functions $f : \mathcal{X} \rightarrow \mathcal{Y}$.

We define the **Bayes risk** as the minimal expected risk over all possible functions,

$$R_B = \min_{f \in \mathcal{Y}^{\mathcal{X}}} R(f),$$

and call **Bayes model** the model f_B that achieves this minimum.

No model f can perform better than f_B .

The **capacity** of an hypothesis space induced by a learning algorithm intuitively represents the ability to find a good model $f \in \mathcal{F}$ for any function, regardless of its complexity.

In practice, capacity can be controlled through hyper-parameters of the learning algorithm. For example:

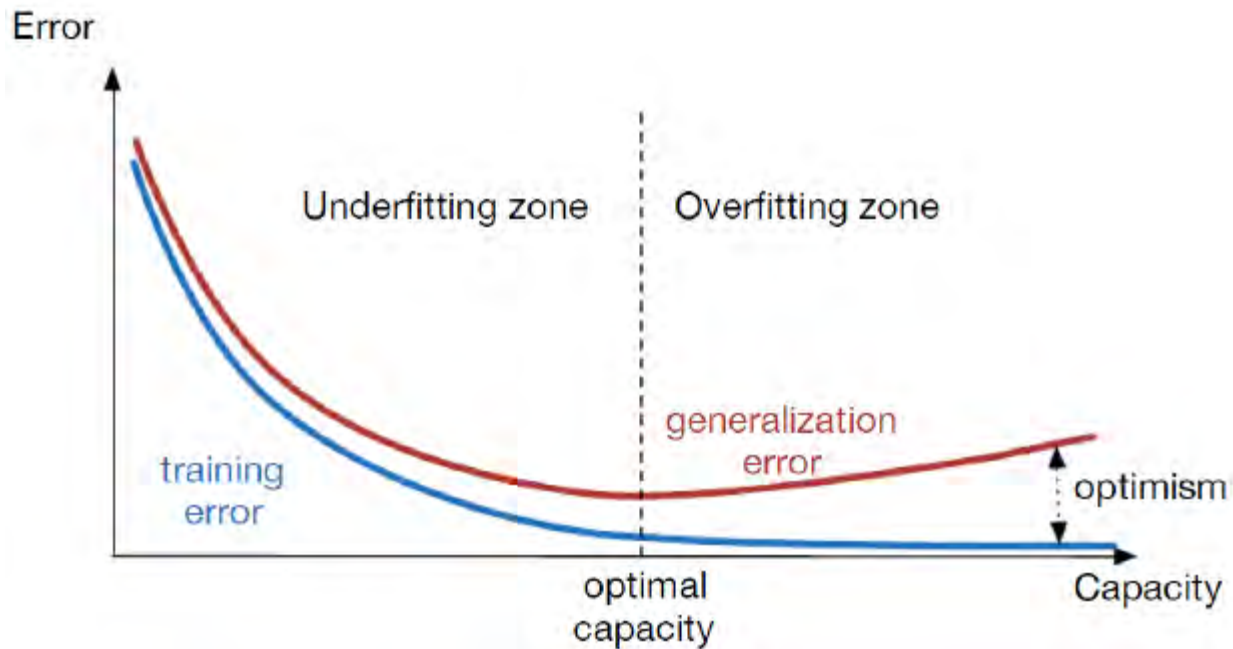
- The degree of the family of polynomials;
- The number of layers in a neural network;
- The number of training iterations;
- Regularization terms.

- If the capacity of \mathcal{F} is too low, then $f_B \notin \mathcal{F}$ and $R(f) - R_B$ is large for any $f \in \mathcal{F}$, including f_* and $f_*^{\mathbf{d}}$. Such models f are said to **underfit** the data.
- If the capacity of \mathcal{F} is too high, then $f_B \in \mathcal{F}$ or $R(f_*) - R_B$ is small. However, because of the high capacity of the hypothesis space, the empirical risk minimizer $f_*^{\mathbf{d}}$ could fit the training data arbitrarily well such that

$$R(f_*^{\mathbf{d}}) \geq R_B \geq \hat{R}(f_*^{\mathbf{d}}, \mathbf{d}) \geq 0.$$

In this situation, $f_*^{\mathbf{d}}$ becomes too specialized with respect to the true data generating process and a large reduction of the empirical risk (often) comes at the price of an increase of the expected risk of the empirical risk minimizer $R(f_*^{\mathbf{d}})$. In this situation, $f_*^{\mathbf{d}}$ is said to **overfit** the data.

Therefore, our goal is to adjust the capacity of the hypothesis space such that the expected risk of the empirical risk minimizer gets as low as possible.



When overfitting,

$$R(f_*^{\mathbf{d}}) \geq R_B \geq \hat{R}(f_*^{\mathbf{d}}, \mathbf{d}) \geq 0.$$

This indicates that the empirical risk $\hat{R}(f_*^{\mathbf{d}}, \mathbf{d})$ is a poor estimator of the expected risk $R(f_*^{\mathbf{d}})$.

Nevertheless, an unbiased estimate of the expected risk can be obtained by evaluating $f_*^{\mathbf{d}}$ on data \mathbf{d}_{test} independent from the training samples \mathbf{d} :

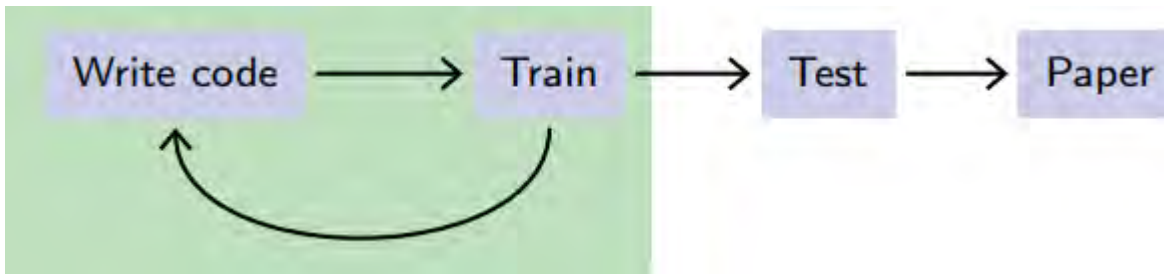
$$\hat{R}(f_*^{\mathbf{d}}, \mathbf{d}_{\text{test}}) = \frac{1}{N} \sum_{(\mathbf{x}_i, y_i) \in \mathbf{d}_{\text{test}}} \ell(y_i, f_*^{\mathbf{d}}(\mathbf{x}_i))$$

This **test error** estimate can be used to evaluate the actual performance of the model. However, it should not be used, at the same time, for model selection.

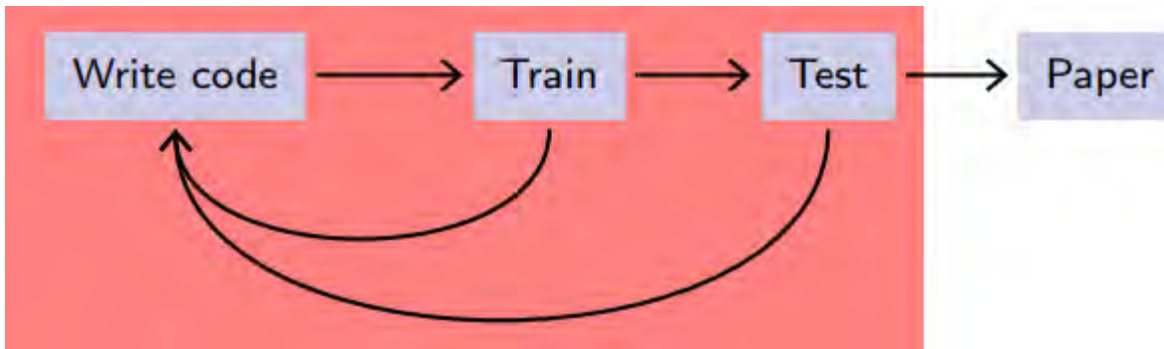


Degree d of the polynomial VS. error.

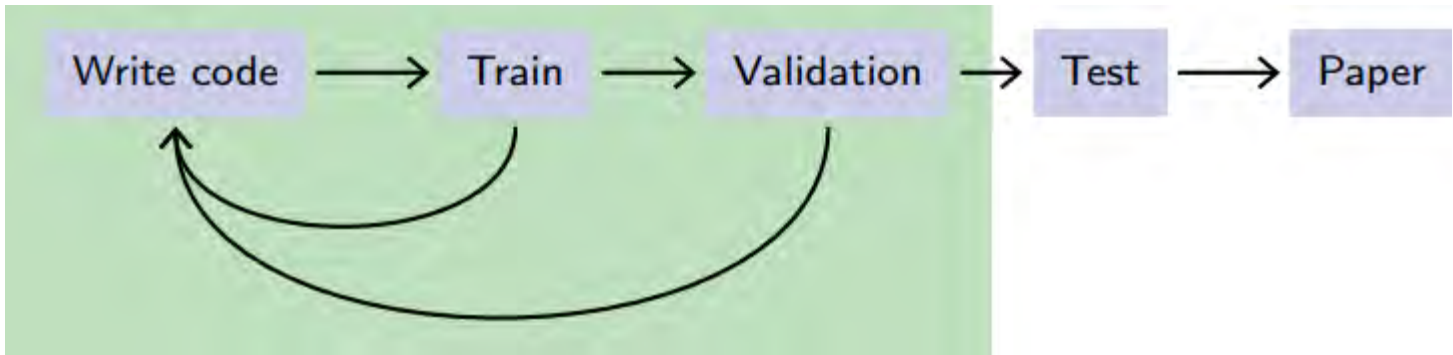
(Proper) evaluation protocol



There may be over-fitting, but it does not bias the final performance evaluation.



This should be **avoided** at all costs!



Instead, keep a separate validation set for tuning the hyper-parameters.

Bias-variance decomposition

Consider a fixed point \boldsymbol{x} and the prediction $\hat{Y} = f_*^{\mathbf{d}}(\boldsymbol{x})$ of the empirical risk minimizer at \boldsymbol{x} .

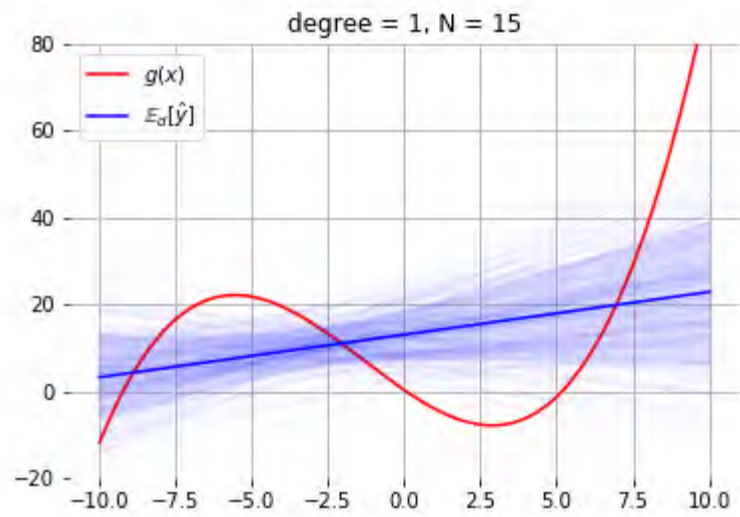
Then the local expected risk of $f_*^{\mathbf{d}}$ is

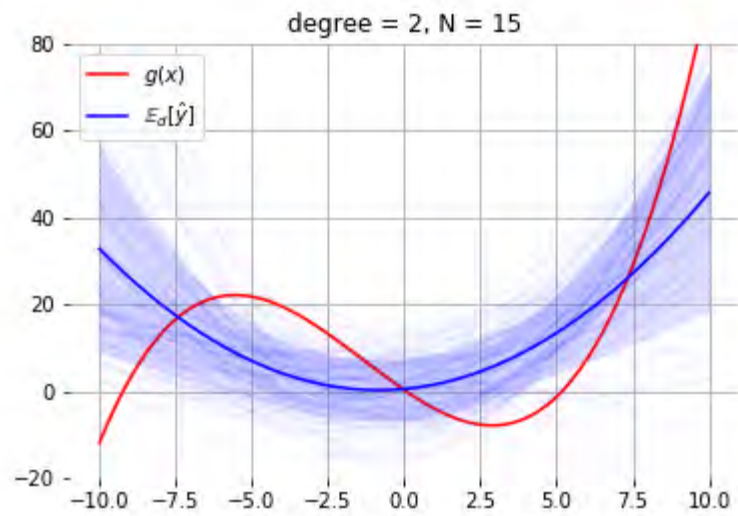
$$\begin{aligned} R(f_*^{\mathbf{d}}|\boldsymbol{x}) &= \mathbb{E}_{y \sim P(Y|\boldsymbol{x})} [(y - f_*^{\mathbf{d}}(\boldsymbol{x}))^2] \\ &= \mathbb{E}_{y \sim P(Y|\boldsymbol{x})} [(y - f_B(\boldsymbol{x}) + f_B(\boldsymbol{x}) - f_*^{\mathbf{d}}(\boldsymbol{x}))^2] \\ &= \mathbb{E}_{y \sim P(Y|\boldsymbol{x})} [(y - f_B(\boldsymbol{x}))^2] + \mathbb{E}_{y \sim P(Y|\boldsymbol{x})} [(f_B(\boldsymbol{x}) - f_*^{\mathbf{d}}(\boldsymbol{x}))^2] \\ &= R(f_B|\boldsymbol{x}) + (f_B(\boldsymbol{x}) - f_*^{\mathbf{d}}(\boldsymbol{x}))^2 \end{aligned}$$

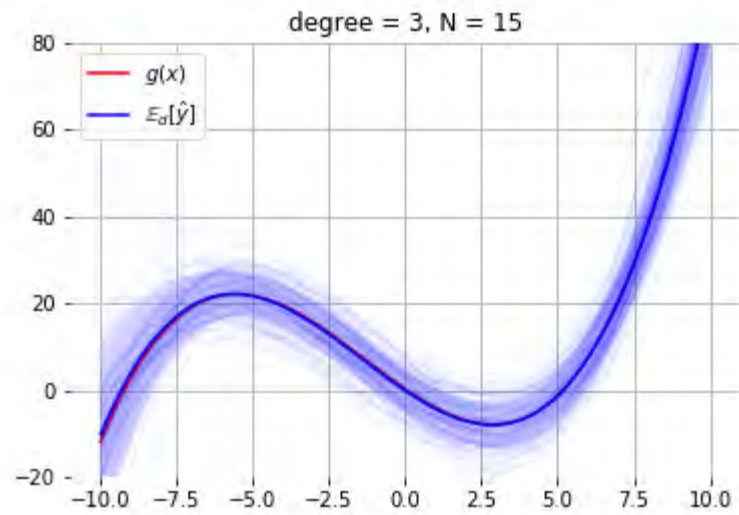
where

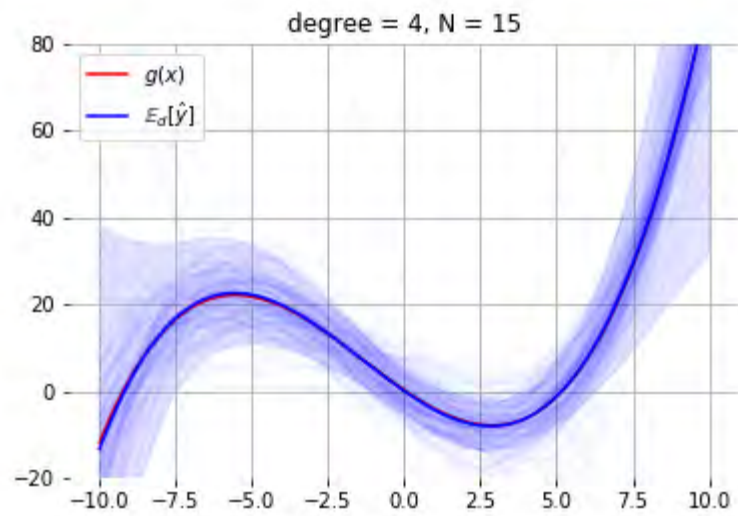
- $R(f_B|\boldsymbol{x})$ is the local expected risk of the Bayes model. This term cannot be reduced.
- $(f_B(\boldsymbol{x}) - f_*^{\mathbf{d}}(\boldsymbol{x}))^2$ represents the discrepancy between f_B and $f_*^{\mathbf{d}}$.

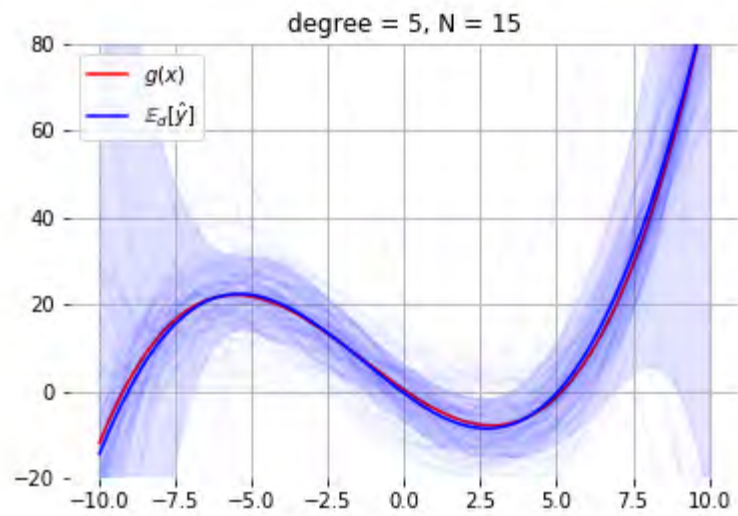
If $\mathbf{d} \sim P(X, Y)$ is itself considered as a random variable, then $f_*^{\mathbf{d}}$ is also a random variable, along with its predictions \hat{Y} .











Formally, the expected local expected risk yields to:

$$\begin{aligned} & \mathbb{E}_{\mathbf{d}} [R(f_*^{\mathbf{d}} | x)] \\ &= \mathbb{E}_{\mathbf{d}} [R(f_B | x) + (f_B(x) - f_*^{\mathbf{d}}(x))^2] \\ &= R(f_B | x) + \mathbb{E}_{\mathbf{d}} [(f_B(x) - f_*^{\mathbf{d}}(x))^2] \\ &= \underbrace{R(f_B | x)}_{\text{noise}(x)} + \underbrace{(f_B(x) - \mathbb{E}_{\mathbf{d}} [f_*^{\mathbf{d}}(x)])^2}_{\text{bias}^2(x)} + \underbrace{\mathbb{E}_{\mathbf{d}} [(\mathbb{E}_{\mathbf{d}} [f_*^{\mathbf{d}}(x)] - f_*^{\mathbf{d}}(x))^2]}_{\text{var}(x)} \end{aligned}$$

This decomposition is known as the **bias-variance** decomposition.

- The noise term quantifies the irreducible part of the expected risk.
- The bias term measures the discrepancy between the average model and the Bayes model.
- The variance term quantifies the variability of the predictions.

Bias-variance trade-off

- Reducing the capacity makes f_*^d fit the data less on average, which increases the bias term.
- Increasing the capacity makes f_*^d vary a lot with the training data, which increases the variance term.

The end.

References

- Vapnik, V. (1992). Principles of risk minimization for learning theory. In Advances in neural information processing systems (pp. 831-838).
- Louppe, G. (2014). Understanding random forests: From theory to practice. arXiv preprint arXiv:1407.7502.

Deep Learning

Lecture 2: Neural networks

Prof. Gilles Louppe
g.louppe@uliege.be

Today

Explain and motivate the basic constructs of neural networks.

- From linear discriminant analysis to logistic regression
- Stochastic gradient descent
- From logistic regression to the multi-layer perceptron
- Vanishing gradients and rectified networks
- Universal approximation theorem

Cooking recipe

- Get data (loads of them).
- Get good hardware.
- Define the neural network architecture as a composition of differentiable functions.
 - Stick to non-saturating activation function to avoid vanishing gradients.
 - Prefer deep over shallow architectures.
- Optimize with (variants of) stochastic gradient descent.
 - Evaluate gradients with automatic differentiation.

Neural networks

Threshold Logic Unit

The Threshold Logic Unit (McCulloch and Pitts, 1943) was the first mathematical model for a **neuron**. Assuming Boolean inputs and outputs, it is defined as:

$$f(\mathbf{x}) = 1_{\{\sum_i w_i x_i + b \geq 0\}}$$

This unit can implement:

- $\text{or}(a, b) = 1_{\{a+b-0.5 \geq 0\}}$
- $\text{and}(a, b) = 1_{\{a+b-1.5 \geq 0\}}$
- $\text{not}(a) = 1_{\{-a+0.5 \geq 0\}}$

Therefore, any Boolean function can be built with such units.

A Logical Calculus of Ideas Immanent in Nervous Activity

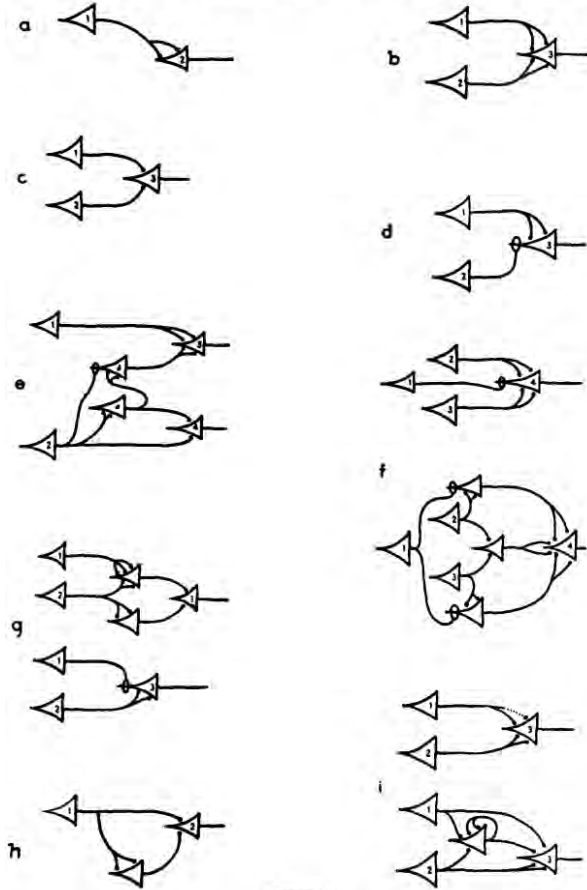


FIGURE 1

Perceptron

The perceptron (Rosenblatt, 1957) is very similar, except that the inputs are real:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

This model was originally motivated by biology, with w_i being synaptic weights and x_i and f firing rates.

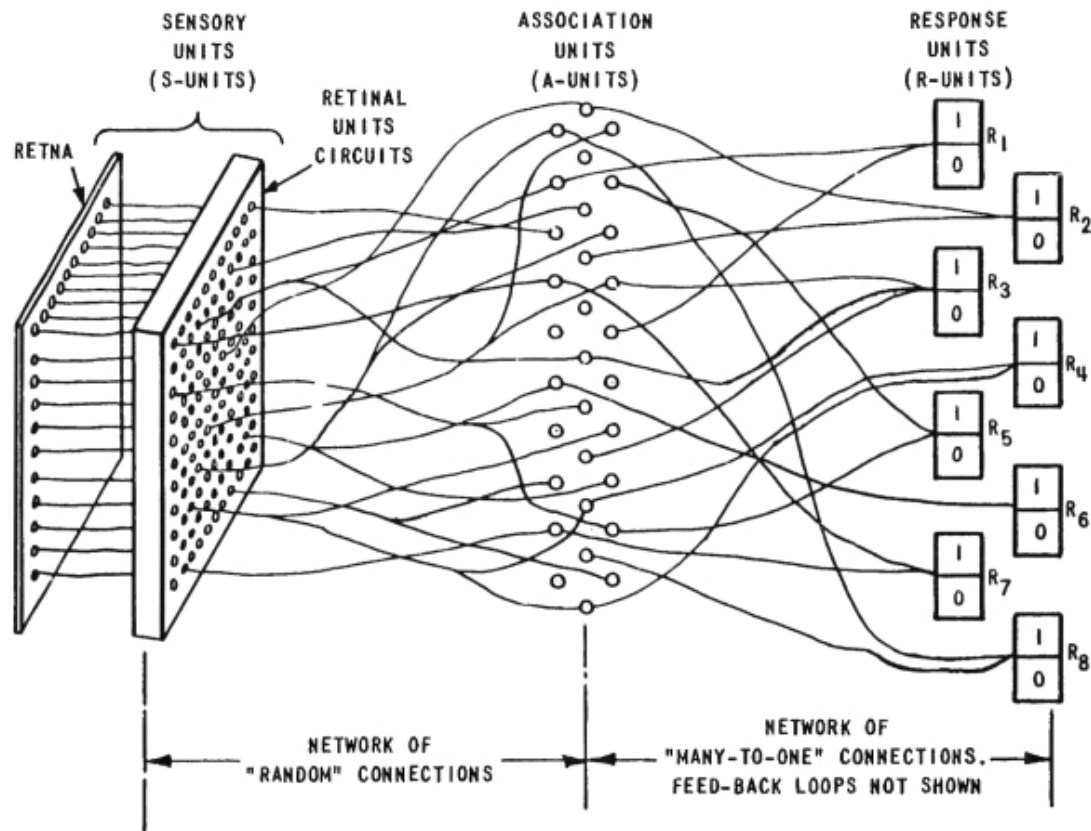
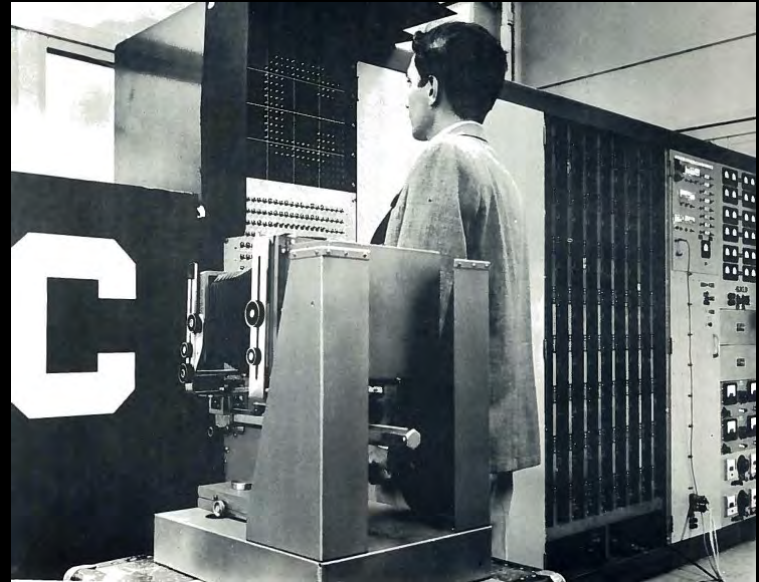
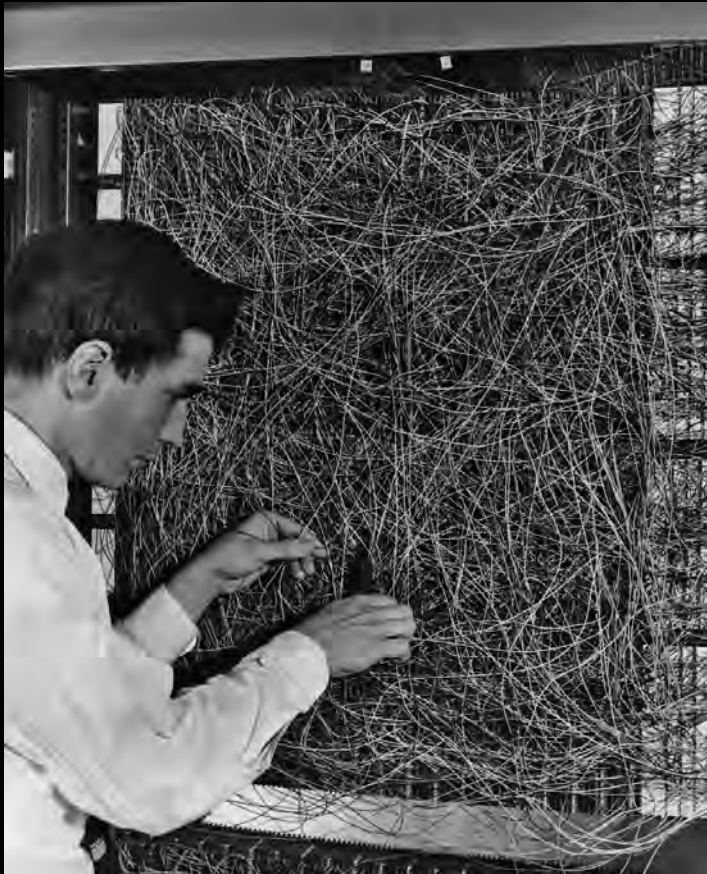


Figure 1 ORGANIZATION OF THE MARK I PERCEPTRON



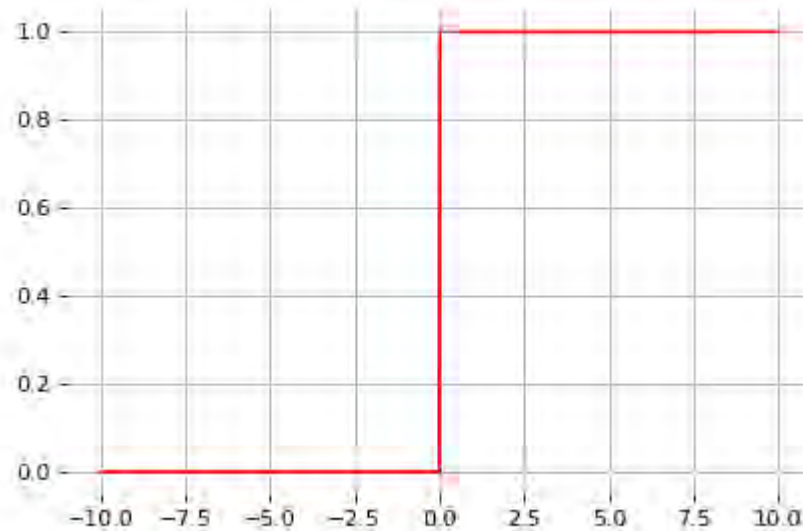
The Mark I Percetron (Frank Rosenblatt).



The Perceptron

Let us define the (non-linear) **activation** function:

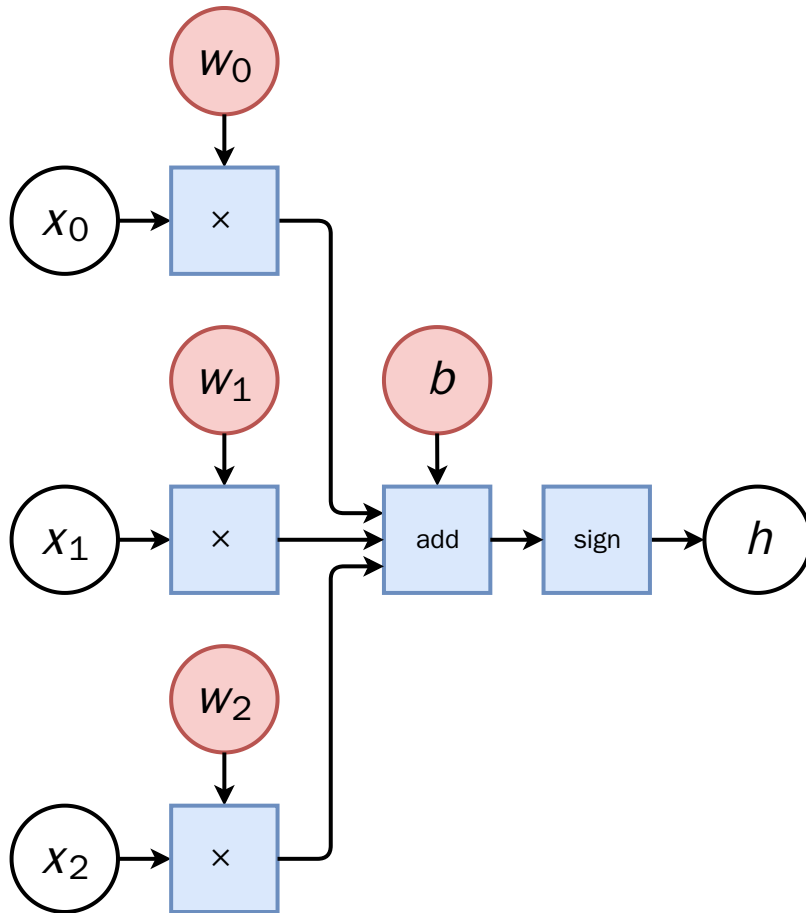
$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



The perceptron classification rule can be rewritten as

$$f(\mathbf{x}) = \text{sign}\left(\sum_i w_i x_i + b\right).$$

Computational graphs



The computation of

$$f(\mathbf{x}) = \text{sign}\left(\sum_i w_i x_i + b\right)$$

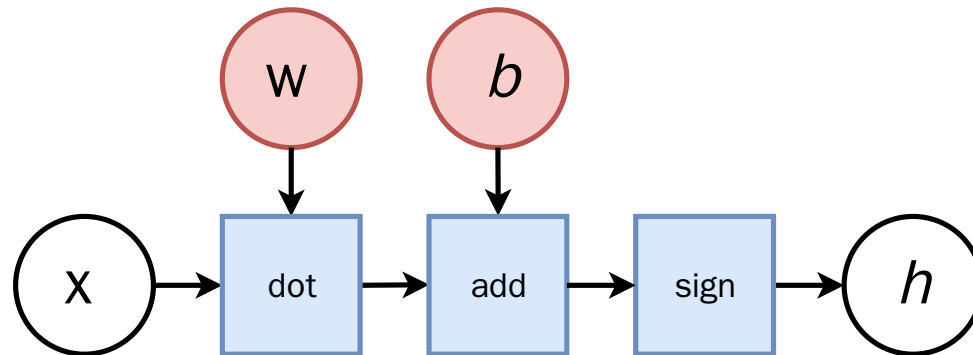
can be represented as a **computational graph** where

- white nodes correspond to inputs and outputs;
- red nodes correspond to model parameters;
- blue nodes correspond to intermediate operations.

In terms of **tensor operations**, f can be rewritten as

$$f(\mathbf{x}) = \text{sign}(\mathbf{w}^T \mathbf{x} + b),$$

for which the corresponding computational graph of f is:



Linear discriminant analysis

Consider training data $(\mathbf{x}, y) \sim P(X, Y)$, with

- $\mathbf{x} \in \mathbb{R}^p$,
- $y \in \{0, 1\}$.

Assume class populations are Gaussian, with same covariance matrix Σ (homoscedasticity):

$$P(\mathbf{x}|y) = \frac{1}{\sqrt{(2\pi)^p |\Sigma|}} \exp \left(-\frac{1}{2} (\mathbf{x} - \mu_y)^T \Sigma^{-1} (\mathbf{x} - \mu_y) \right)$$

Using the Bayes' rule, we have:

$$\begin{aligned} P(Y = 1|\mathbf{x}) &= \frac{P(\mathbf{x}|Y = 1)P(Y = 1)}{P(\mathbf{x})} \\ &= \frac{P(\mathbf{x}|Y = 1)P(Y = 1)}{P(\mathbf{x}|Y = 0)P(Y = 0) + P(\mathbf{x}|Y = 1)P(Y = 1)} \\ &= \frac{1}{1 + \frac{P(\mathbf{x}|Y=0)P(Y=0)}{P(\mathbf{x}|Y=1)P(Y=1)}}. \end{aligned}$$

Using the Bayes' rule, we have:

$$\begin{aligned} P(Y = 1|\mathbf{x}) &= \frac{P(\mathbf{x}|Y = 1)P(Y = 1)}{P(\mathbf{x})} \\ &= \frac{P(\mathbf{x}|Y = 1)P(Y = 1)}{P(\mathbf{x}|Y = 0)P(Y = 0) + P(\mathbf{x}|Y = 1)P(Y = 1)} \\ &= \frac{1}{1 + \frac{P(\mathbf{x}|Y=0)P(Y=0)}{P(\mathbf{x}|Y=1)P(Y=1)}}. \end{aligned}$$

It follows that with

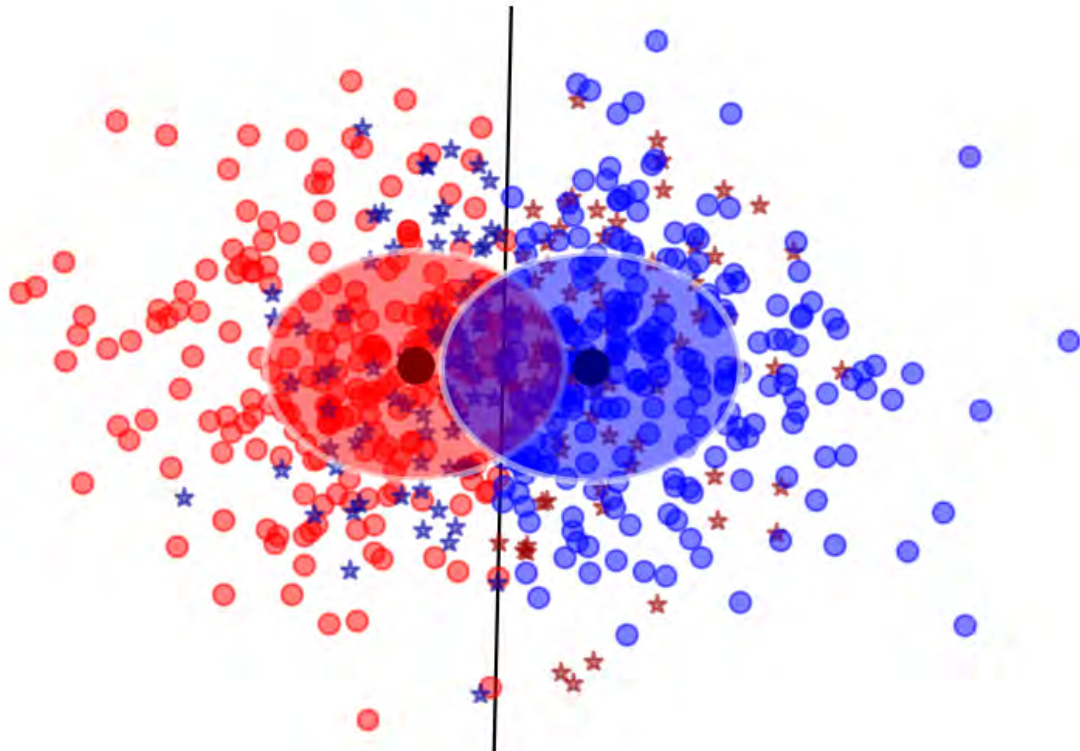
$$\sigma(x) = \frac{1}{1 + \exp(-x)},$$

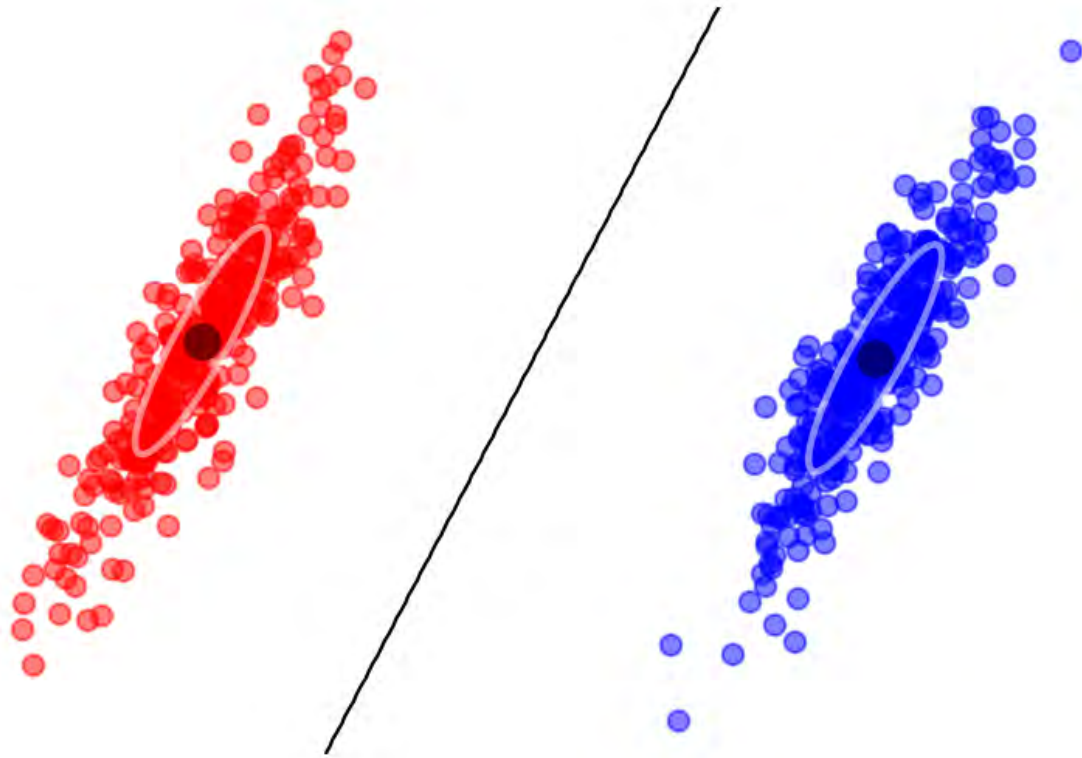
we get

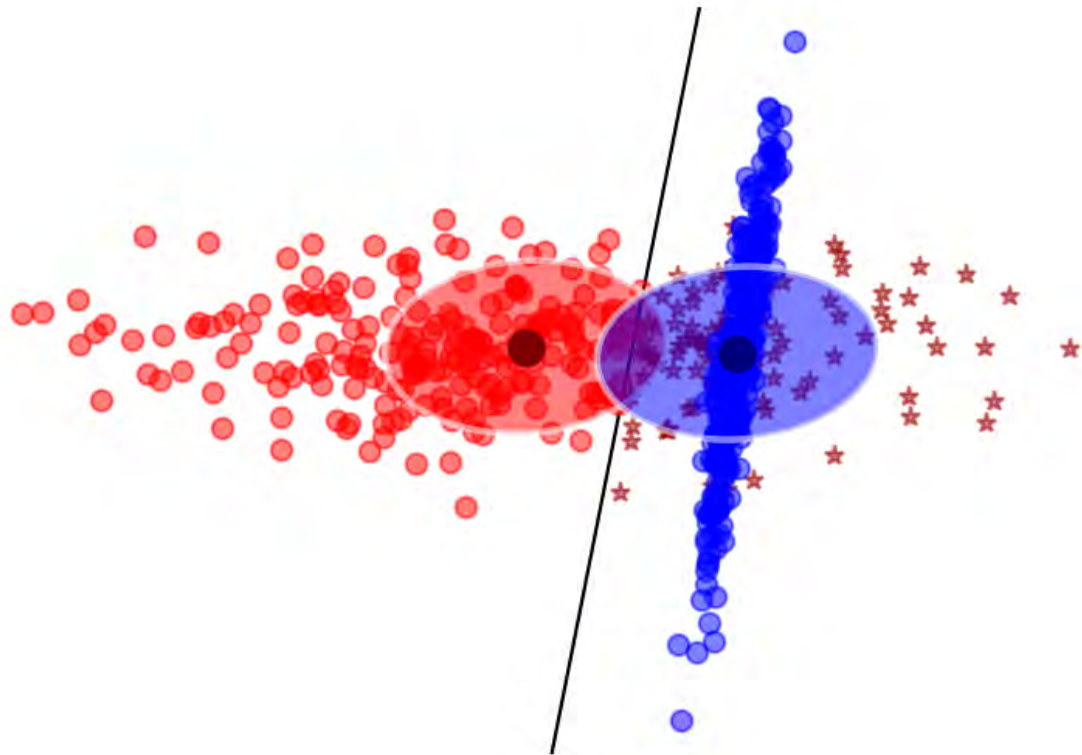
$$P(Y = 1|\mathbf{x}) = \sigma \left(\log \frac{P(\mathbf{x}|Y = 1)}{P(\mathbf{x}|Y = 0)} + \log \frac{P(Y = 1)}{P(Y = 0)} \right).$$

Therefore,

$$\begin{aligned} & P(Y = 1 | \mathbf{x}) \\ &= \sigma \left(\log \frac{P(\mathbf{x} | Y = 1)}{P(\mathbf{x} | Y = 0)} + \underbrace{\log \frac{P(Y = 1)}{P(Y = 0)}}_a \right) \\ &= \sigma (\log P(\mathbf{x} | Y = 1) - \log P(\mathbf{x} | Y = 0) + a) \\ &= \sigma \left(-\frac{1}{2}(\mathbf{x} - \mu_1)^T \Sigma^{-1} (\mathbf{x} - \mu_1) + \frac{1}{2}(\mathbf{x} - \mu_0)^T \Sigma^{-1} (\mathbf{x} - \mu_0) + a \right) \\ &= \sigma \left(\underbrace{(\mu_1 - \mu_0)^T \Sigma^{-1} \mathbf{x}}_{\mathbf{w}^T} + \underbrace{\frac{1}{2}(\mu_0^T \Sigma^{-1} \mu_0 - \mu_1^T \Sigma^{-1} \mu_1)}_b + a \right) \\ &= \sigma (\mathbf{w}^T \mathbf{x} + b) \end{aligned}$$







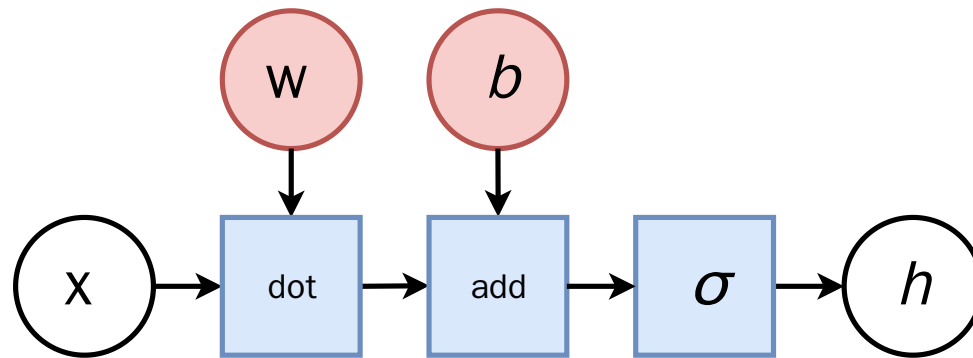
Note that the **sigmoid** function

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

looks like a soft heavyside:



Therefore, the overall model $f(\mathbf{x}; \mathbf{w}, b) = \sigma(\mathbf{w}^T \mathbf{x} + b)$ is very similar to the perceptron.



This unit is the **lego brick** of all neural networks!

Logistic regression

Same model

$$P(Y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x} + b)$$

as for linear discriminant analysis.

But,

- **ignore** model assumptions (Gaussian class populations, homoscedasticity);
- instead, find \mathbf{w} , b that maximizes the likelihood of the data.

We have,

$$\begin{aligned} & \arg \max_{\mathbf{w}, b} P(\mathbf{d} | \mathbf{w}, b) \\ &= \arg \max_{\mathbf{w}, b} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} P(Y = y_i | \mathbf{x}_i, \mathbf{w}, b) \\ &= \arg \max_{\mathbf{w}, b} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} \sigma(\mathbf{w}^T \mathbf{x}_i + b)^{y_i} (1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b))^{1-y_i} \\ &= \arg \min_{\mathbf{w}, b} \underbrace{\sum_{\mathbf{x}_i, y_i \in \mathbf{d}} -y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i + b) - (1 - y_i) \log(1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b))}_{\mathcal{L}(\mathbf{w}, b) = \sum_i \ell(y_i, \hat{y}(\mathbf{x}_i; \mathbf{w}, b))} \end{aligned}$$

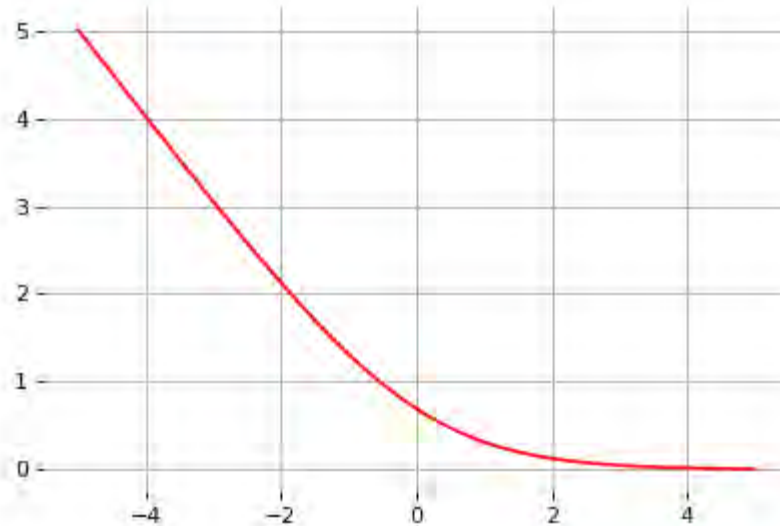
This loss is an instance of the **cross-entropy**

$$H(p, q) = \mathbb{E}_p[-\log q]$$

for $p = Y | \mathbf{x}_i$ and $q = \hat{Y} | \mathbf{x}_i$.

When Y takes values in $\{-1, 1\}$, a similar derivation yields the **logistic loss**

$$\mathcal{L}(\mathbf{w}, b) = - \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \log \sigma (y_i(\mathbf{w}^T \mathbf{x}_i + b)).$$



- In general, the cross-entropy and the logistic losses do not admit a minimizer that can be expressed analytically in closed form.
- However, a minimizer can be found numerically, using a general minimization technique such as **gradient descent**.

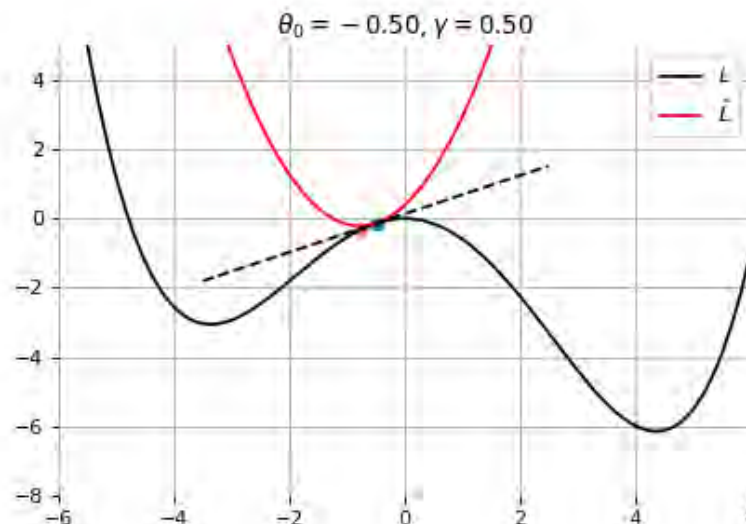
Gradient descent

Let $\mathcal{L}(\theta)$ denote a loss function defined over model parameters θ (e.g., \mathbf{w} and b).

To minimize $\mathcal{L}(\theta)$, **gradient descent** uses local linear information to iteratively move towards a (local) minimum.

For $\theta_0 \in \mathbb{R}^d$, a first-order approximation around θ_0 can be defined as

$$\hat{\mathcal{L}}(\theta_0 + \epsilon) = \mathcal{L}(\theta_0) + \epsilon^T \nabla_{\theta} \mathcal{L}(\theta_0) + \frac{1}{2\gamma} \|\epsilon\|^2.$$



A minimizer of the approximation $\hat{\mathcal{L}}(\theta_0 + \epsilon)$ is given for

$$\begin{aligned}\nabla_{\epsilon} \hat{\mathcal{L}}(\theta_0 + \epsilon) &= 0 \\ &= \nabla_{\theta} \mathcal{L}(\theta_0) + \frac{1}{\gamma} \epsilon,\end{aligned}$$

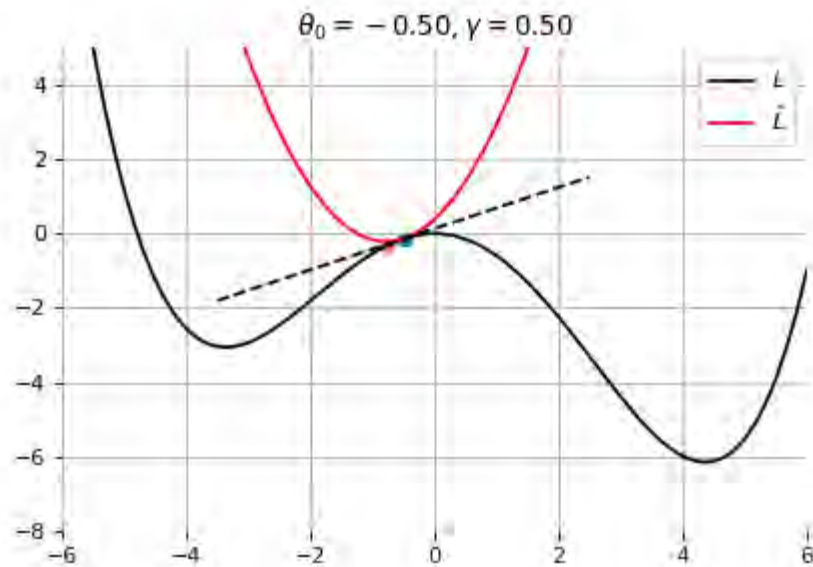
which results in the best improvement for the step $\epsilon = -\gamma \nabla_{\theta} \mathcal{L}(\theta_0)$.

Therefore, model parameters can be updated iteratively using the update rule

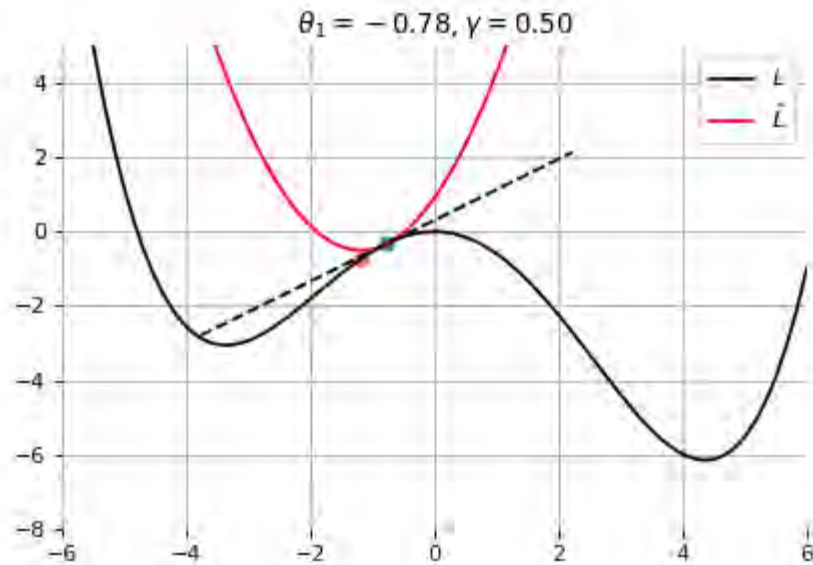
$$\theta_{t+1} = \theta_t - \gamma \nabla_{\theta} \mathcal{L}(\theta_t),$$

where

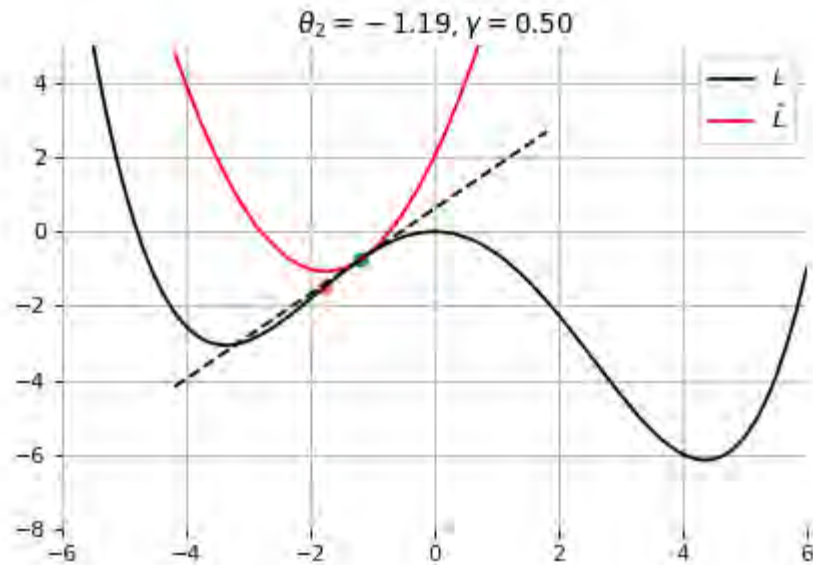
- θ_0 are the initial parameters of the model;
- γ is the **learning rate**;
- both are critical for the convergence of the update rule.



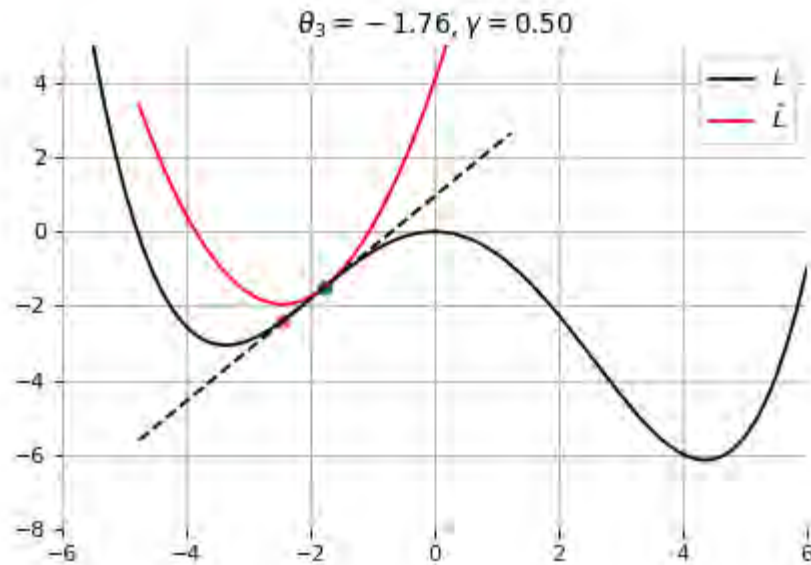
Example 1: Convergence to a local minima



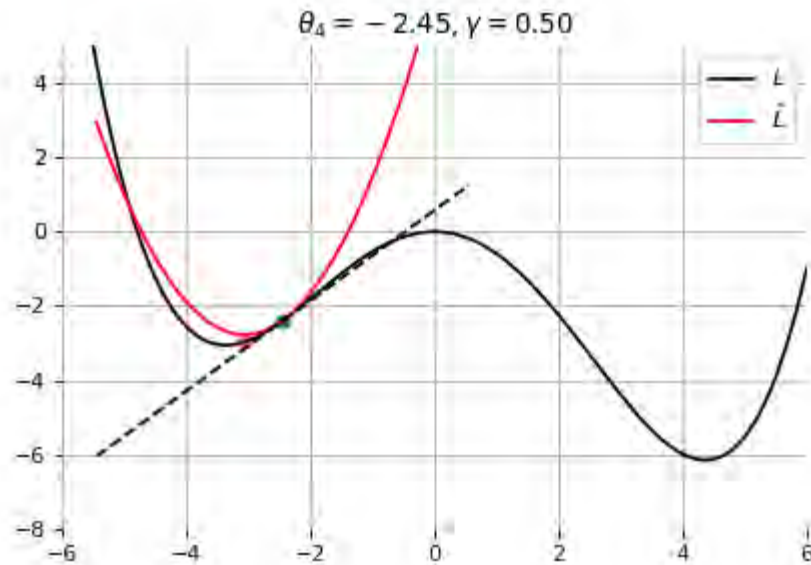
Example 1: Convergence to a local minima



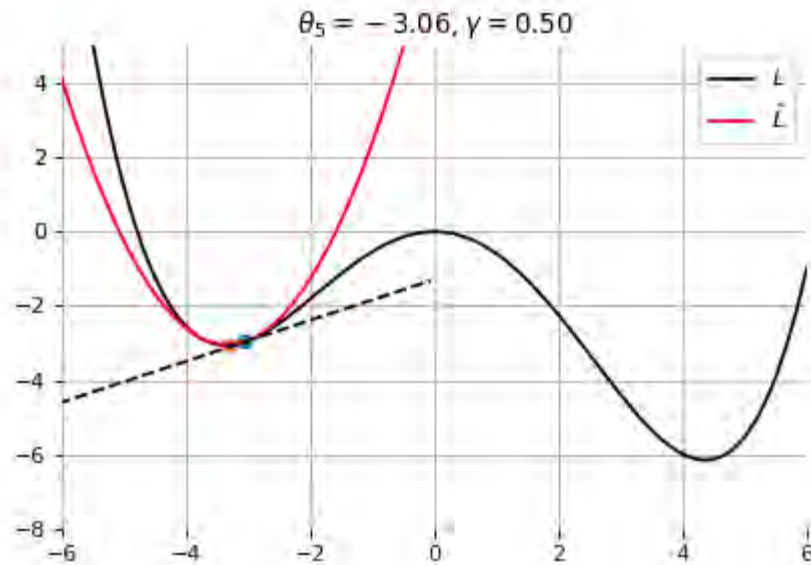
Example 1: Convergence to a local minima



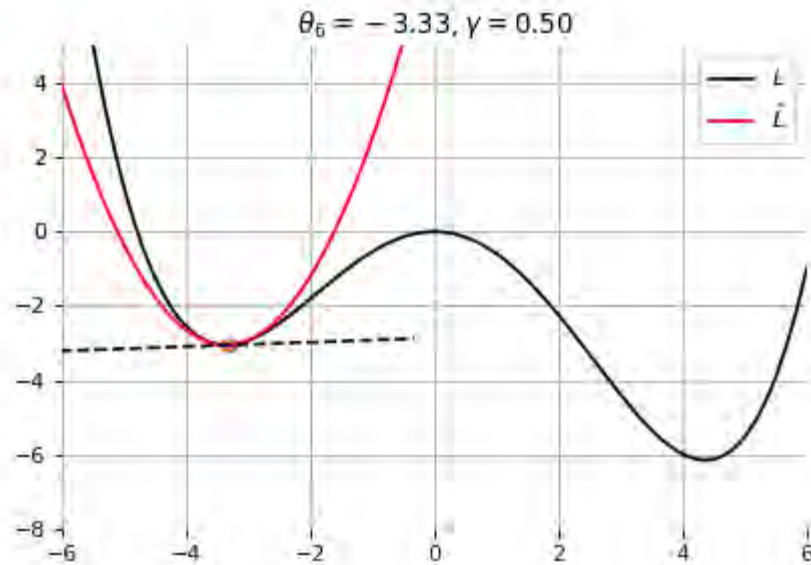
Example 1: Convergence to a local minima



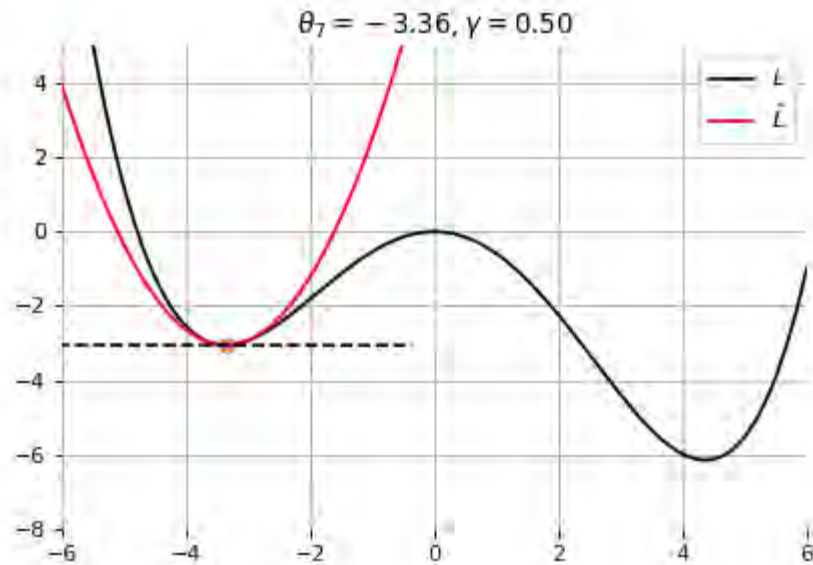
Example 1: Convergence to a local minima



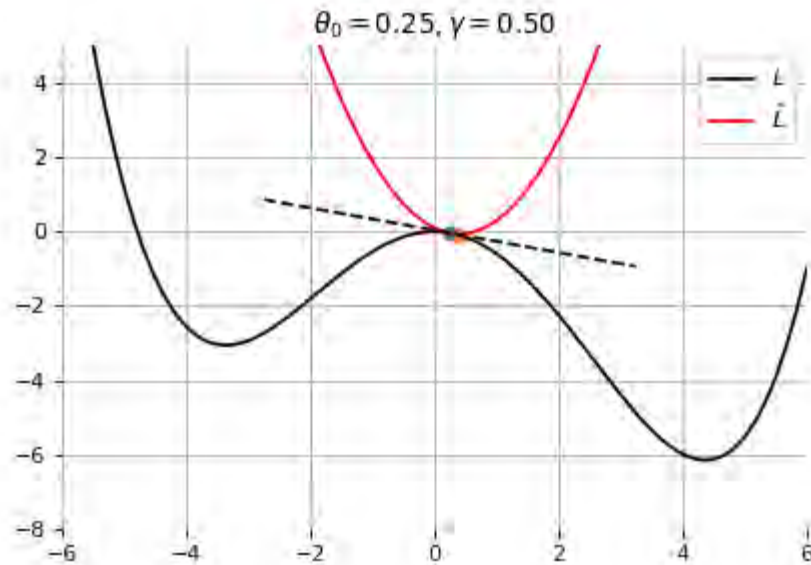
Example 1: Convergence to a local minima



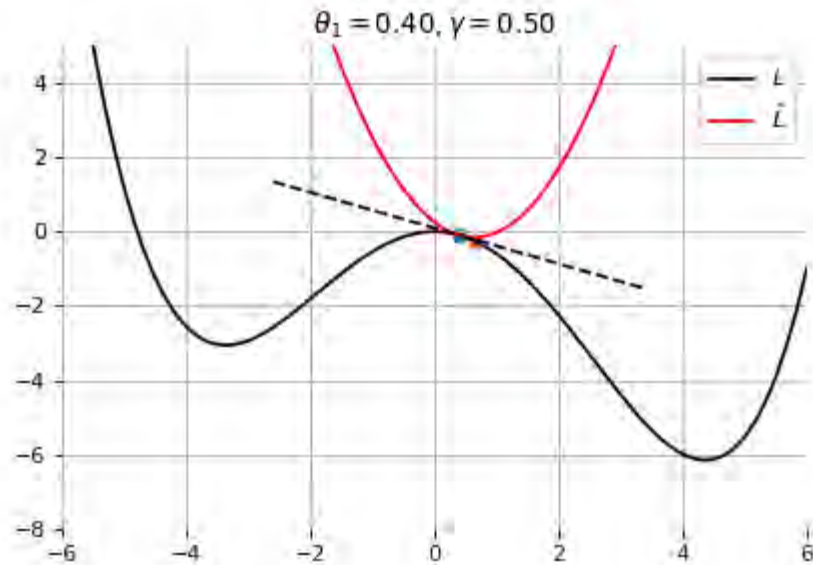
Example 1: Convergence to a local minima



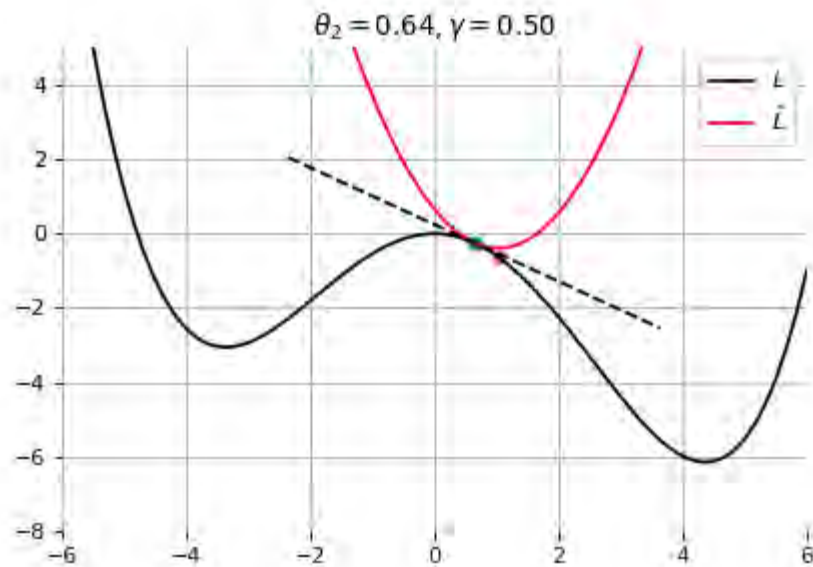
Example 1: Convergence to a local minima



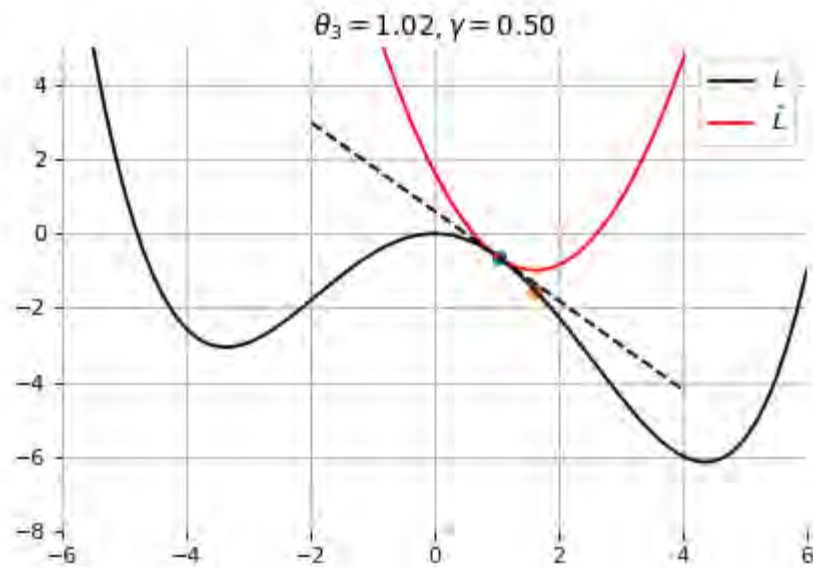
Example 2: Convergence to the global minima



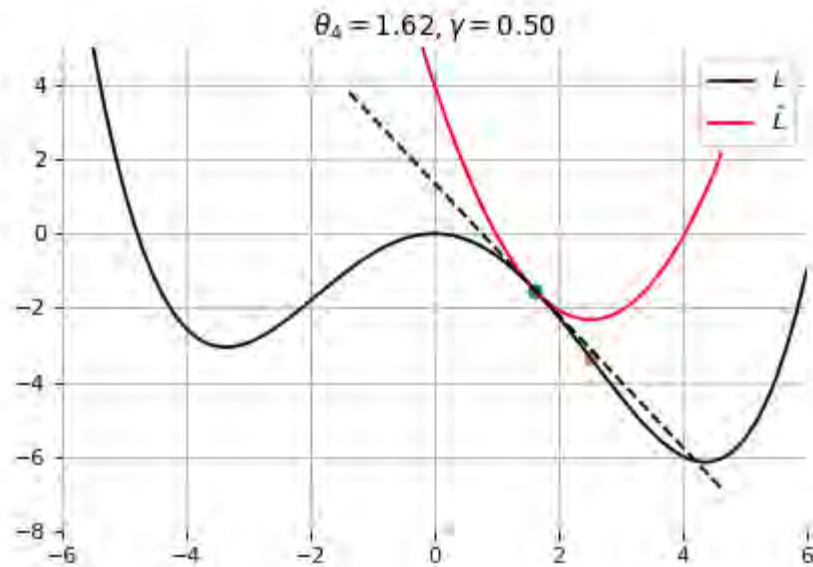
Example 2: Convergence to the global minima



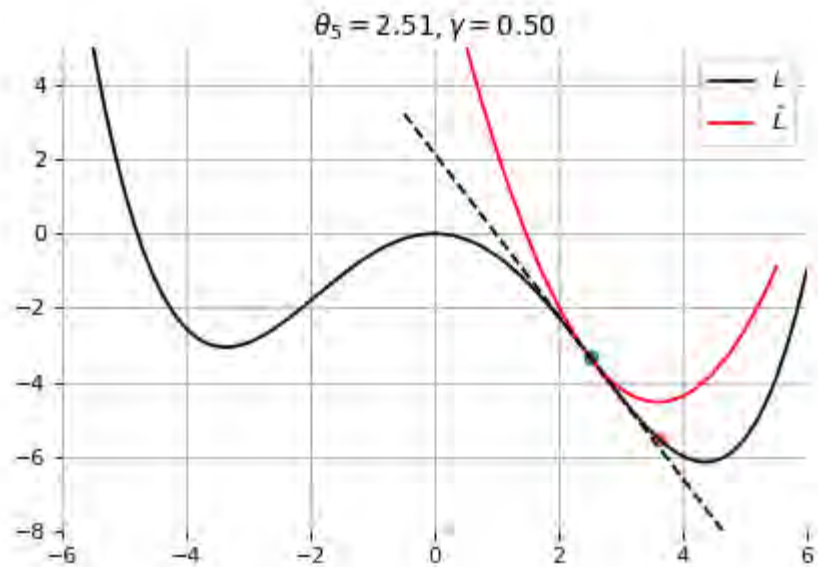
Example 2: Convergence to the global minima



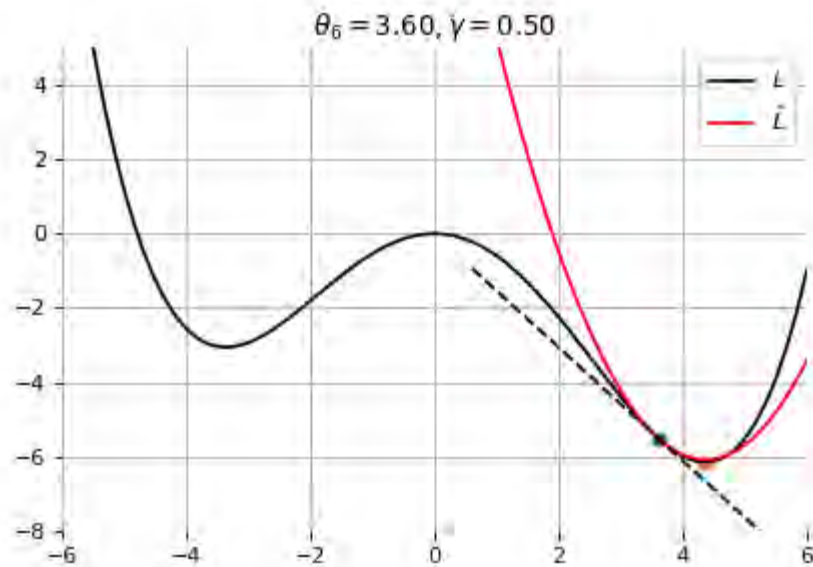
Example 2: Convergence to the global minima



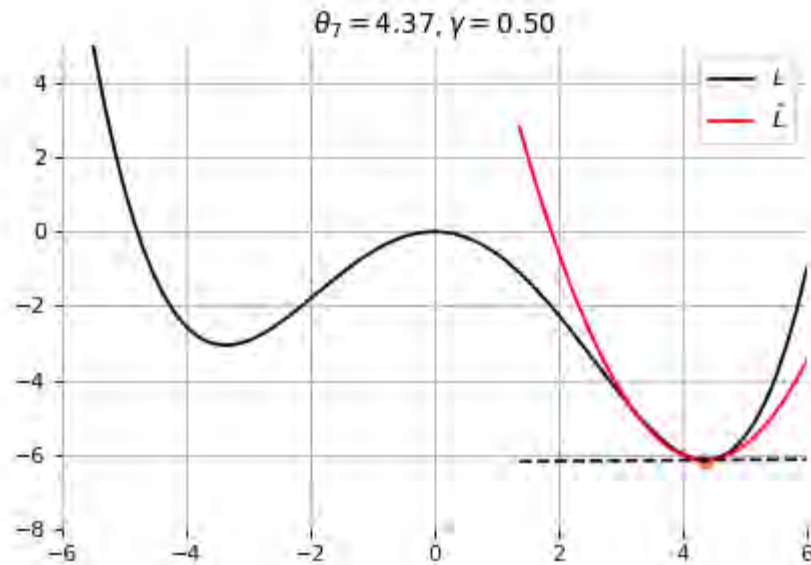
Example 2: Convergence to the global minima



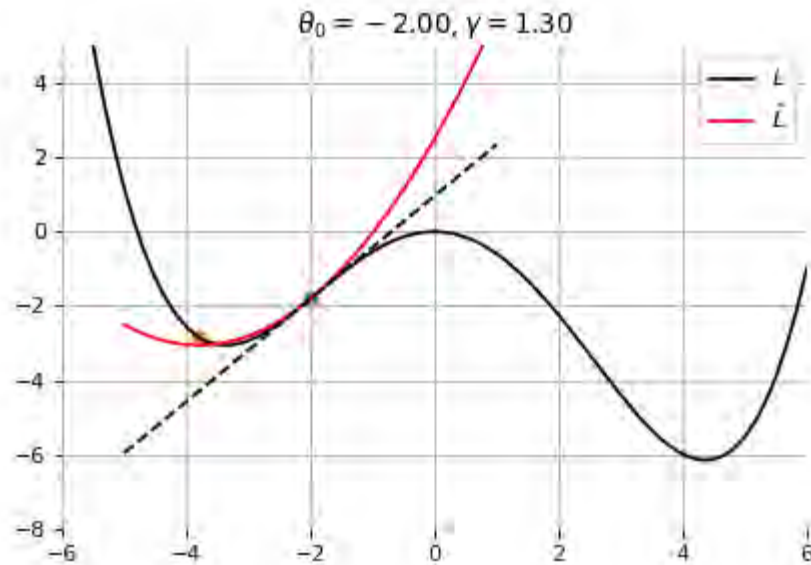
Example 2: Convergence to the global minima



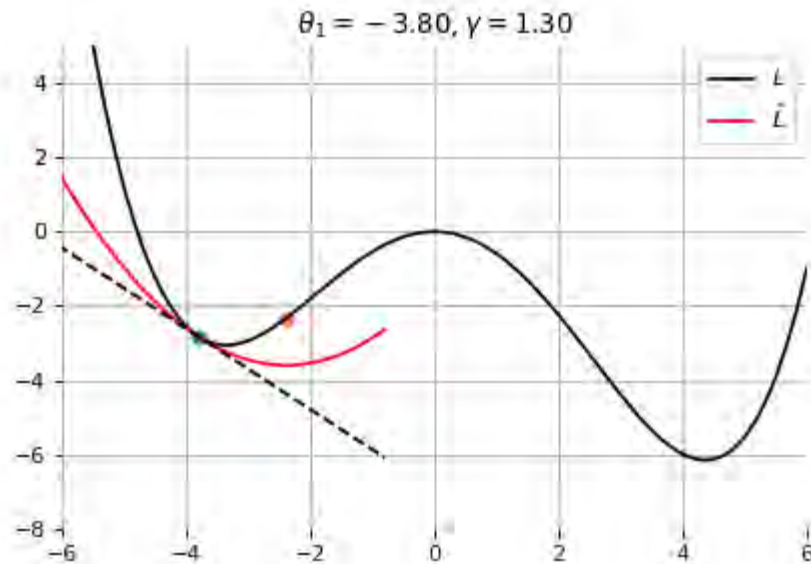
Example 2: Convergence to the global minima



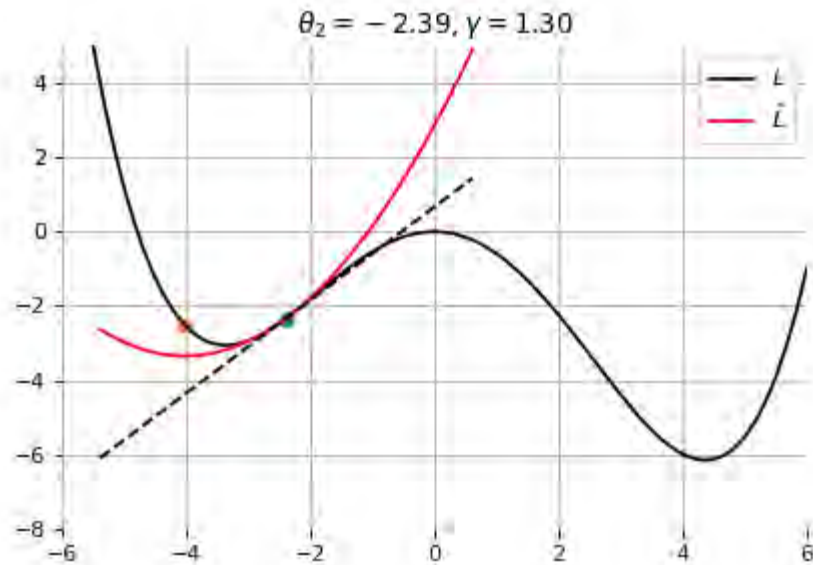
Example 2: Convergence to the global minima



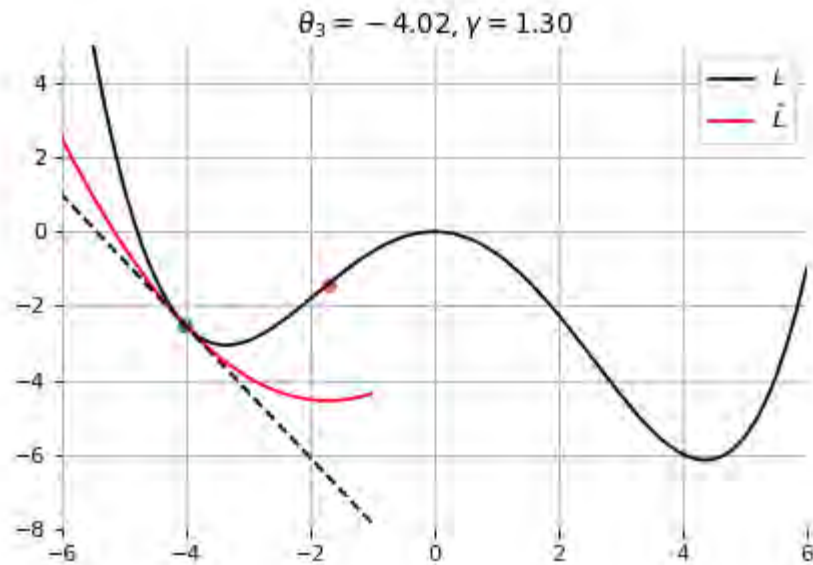
Example 3: Divergence due to a too large learning rate



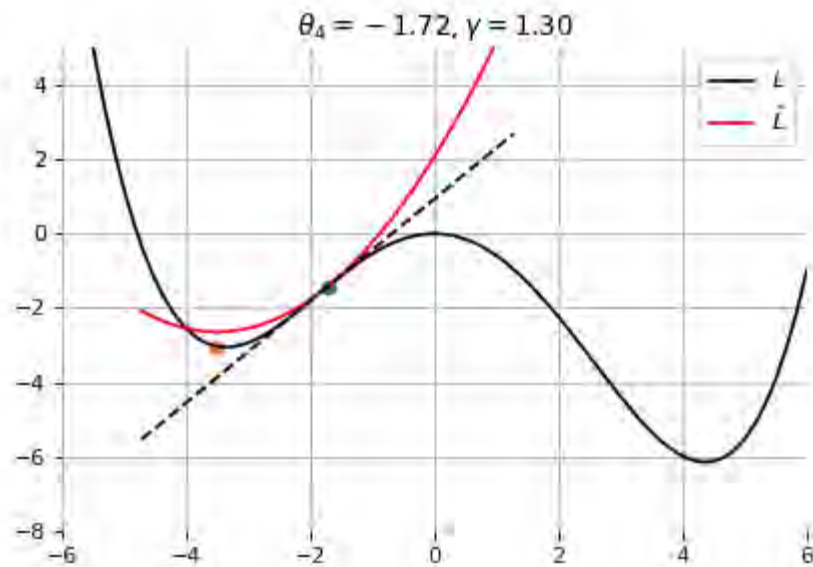
Example 3: Divergence due to a too large learning rate



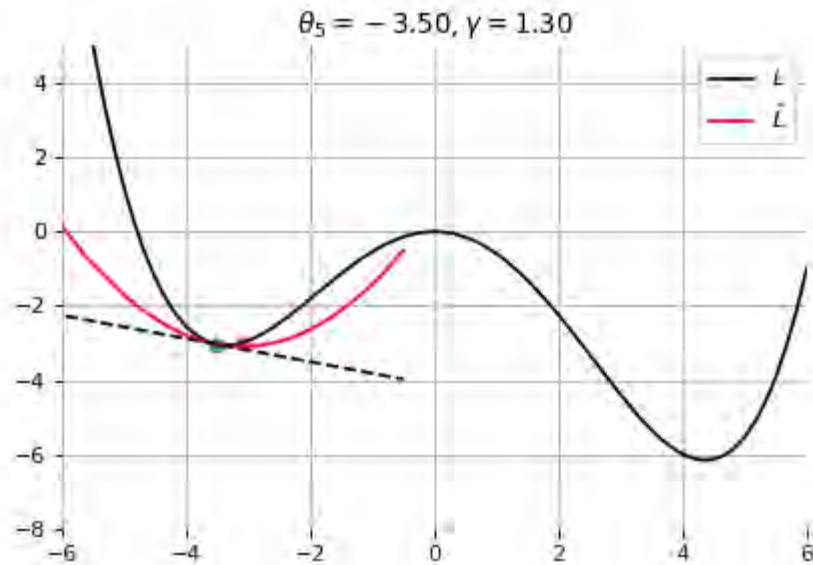
Example 3: Divergence due to a too large learning rate



Example 3: Divergence due to a too large learning rate



Example 3: Divergence due to a too large learning rate



Example 3: Divergence due to a too large learning rate

Stochastic gradient descent

In the empirical risk minimization setup, $\mathcal{L}(\theta)$ and its gradient decompose as

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \ell(y_i, f(\mathbf{x}_i; \theta))$$
$$\nabla \mathcal{L}(\theta) = \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta)).$$

Therefore, in **batch** gradient descent the complexity of an update grows linearly with the size N of the dataset.

More importantly, since the empirical risk is already an approximation of the expected risk, it should not be necessary to carry out the minimization with great accuracy.

Instead, **stochastic** gradient descent uses as update rule:

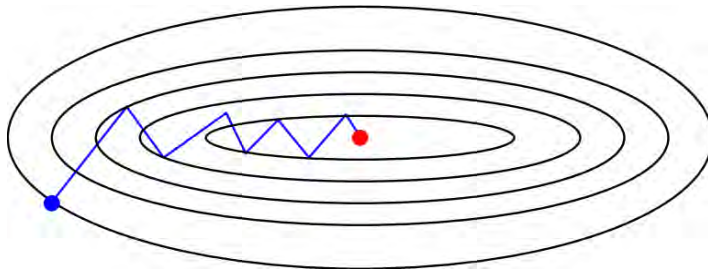
$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- Iteration complexity is independent of N .
- The stochastic process $\{\theta_t | t = 1, \dots\}$ depends on the examples $i(t)$ picked randomly at each iteration.

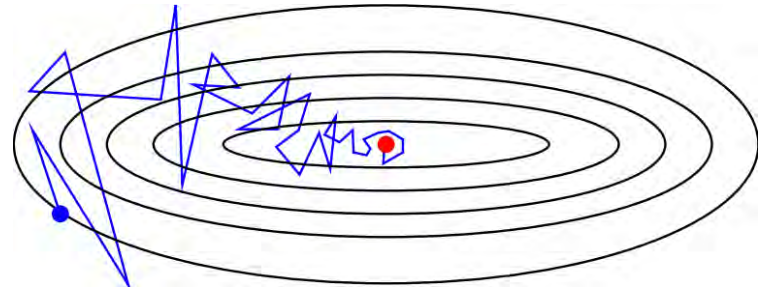
Instead, **stochastic** gradient descent uses as update rule:

$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

- Iteration complexity is independent of N .
- The stochastic process $\{\theta_t | t = 1, \dots\}$ depends on the examples $i(t)$ picked randomly at each iteration.



Batch gradient descent



Stochastic gradient descent

Why is stochastic gradient descent still a good idea?

- Informally, averaging the update

$$\theta_{t+1} = \theta_t - \gamma \nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))$$

over all choices $i(t+1)$ restores batch gradient descent.

- Formally, if the gradient estimate is **unbiased**, e.g., if

$$\begin{aligned} \mathbb{E}_{i(t+1)} [\nabla \ell(y_{i(t+1)}, f(\mathbf{x}_{i(t+1)}; \theta_t))] &= \frac{1}{N} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \nabla \ell(y_i, f(\mathbf{x}_i; \theta_t)) \\ &= \nabla \mathcal{L}(\theta_t) \end{aligned}$$

then the formal convergence of SGD can be proved, under appropriate assumptions (see references).

- Interestingly, if training examples $\mathbf{x}_i, y_i \sim P_{X,Y}$ are received and used in an online fashion, then SGD directly minimizes the **expected** risk.

When decomposing the excess error in terms of approximation, estimation and optimization errors, stochastic algorithms yield the best generalization performance (in terms of **expected** risk) despite being the worst optimization algorithms (in terms of **empirical risk**) (Bottou, 2011).

$$\begin{aligned} & \mathbb{E} \left[R(\tilde{f}_*^{\mathbf{d}}) - R(f_B) \right] \\ &= \mathbb{E} \left[R(f_*) - R(f_B) \right] + \mathbb{E} \left[R(f_*^{\mathbf{d}}) - R(f_*) \right] + \mathbb{E} \left[R(\tilde{f}_*^{\mathbf{d}}) - R(f_*^{\mathbf{d}}) \right] \\ &= \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \end{aligned}$$

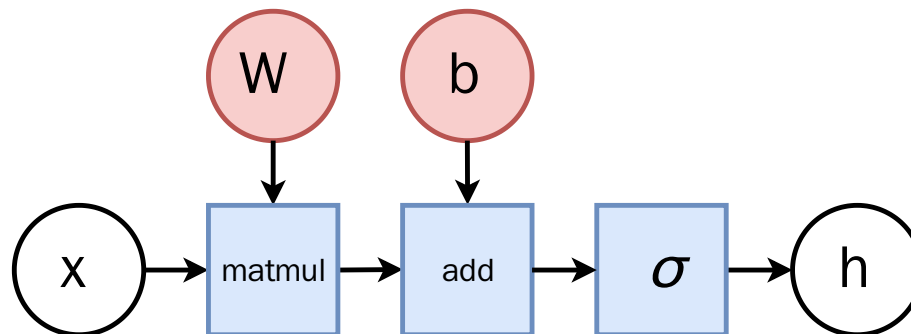
Layers

So far we considered the logistic unit $h = \sigma(\mathbf{w}^T \mathbf{x} + b)$, where $h \in \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^p$, $\mathbf{w} \in \mathbb{R}^p$ and $b \in \mathbb{R}$.

These units can be composed **in parallel** to form a **layer** with q outputs:

$$\mathbf{h} = \sigma(\mathbf{W}^T \mathbf{x} + \mathbf{b})$$

where $\mathbf{h} \in \mathbb{R}^q$, $\mathbf{x} \in \mathbb{R}^p$, $\mathbf{W} \in \mathbb{R}^{p \times q}$, $\mathbf{b} \in \mathbb{R}^q$ and where $\sigma(\cdot)$ is upgraded to the element-wise sigmoid function.



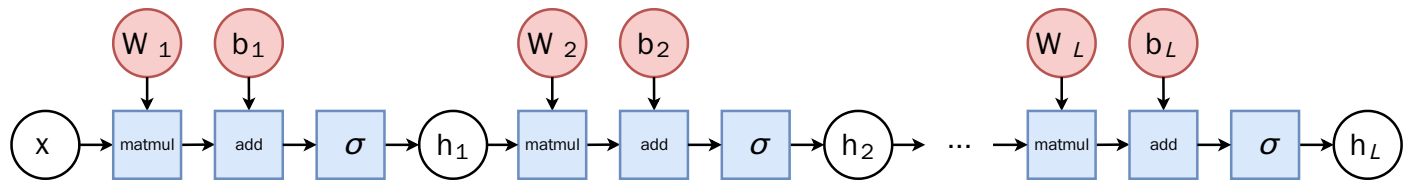
Multi-layer perceptron

Similarly, layers can be composed **in series**, such that:

$$\begin{aligned}\mathbf{h}_0 &= \mathbf{x} \\ \mathbf{h}_1 &= \sigma(\mathbf{W}_1^T \mathbf{h}_0 + \mathbf{b}_1) \\ &\dots \\ \mathbf{h}_L &= \sigma(\mathbf{W}_L^T \mathbf{h}_{L-1} + \mathbf{b}_L) \\ f(\mathbf{x}; \theta) &= \hat{y} = \mathbf{h}_L\end{aligned}$$

where θ denotes the model parameters $\{\mathbf{W}_k, \mathbf{b}_k, \dots | k = 1, \dots, L\}$.

This model is the **multi-layer perceptron**, also known as the fully connected feedforward network.



Classification

- For binary classification, the width q of the last layer L is set to 1 , which results in a single output $h_L \in [0, 1]$ that models the probability $P(Y = 1|\mathbf{x})$.
- For multi-class classification, the sigmoid action σ in the last layer can be generalized to produce a (normalized) vector $\mathbf{h}_L \in [0, 1]^C$ of probability estimates $P(Y = i|\mathbf{x})$.

This activation is the **Softmax** function, where its i -th output is defined as

$$\text{Softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)},$$

for $i = 1, \dots, C$.

Regression

The last activation σ can be skipped to produce unbounded output values $h_L \in \mathbb{R}$.

Automatic differentiation

To minimize $\mathcal{L}(\theta)$ with stochastic gradient descent, we need the gradient $\nabla_{\theta} \ell(\theta_t)$.

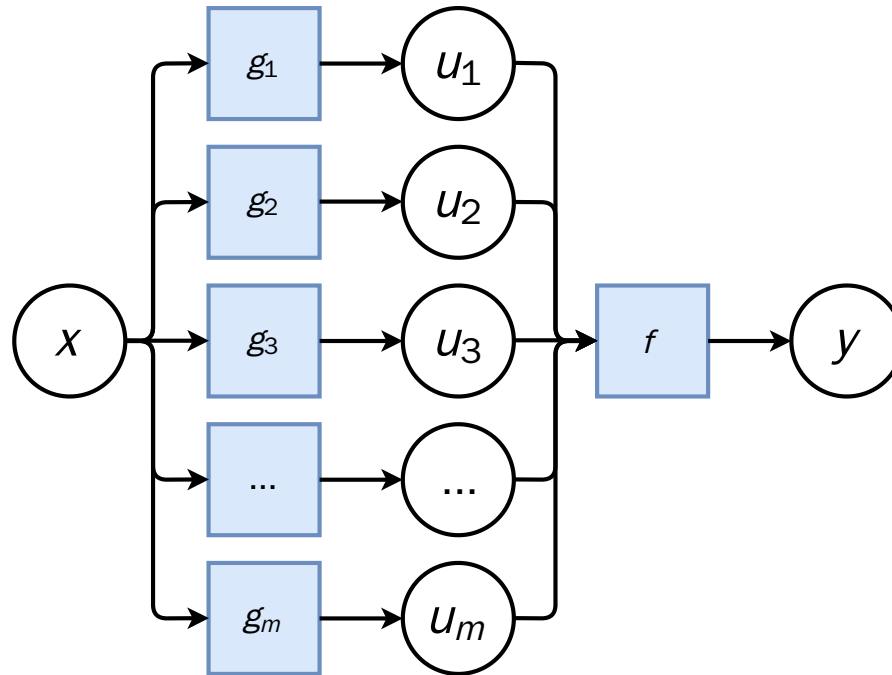
Therefore, we require the evaluation of the (total) derivatives

$$\frac{d\ell}{d\mathbf{W}_k}, \frac{d\ell}{d\mathbf{b}_k}$$

of the loss ℓ with respect to all model parameters $\mathbf{W}_k, \mathbf{b}_k$, for $k = 1, \dots, L$.

These derivatives can be evaluated automatically from the [computational graph](#) of ℓ using [automatic differentiation](#).

Chain rule



Let us consider a 1-dimensional output composition $f \circ g$, such that

$$y = f(\mathbf{u})$$

$$\mathbf{u} = g(x) = (g_1(x), \dots, g_m(x)).$$

The **chain rule** states that $(f \circ g)' = (f' \circ g)g'$.

For the total derivative, the chain rule generalizes to

$$\frac{dy}{dx} = \sum_{k=1}^m \frac{\partial y}{\partial u_k} \underbrace{\frac{du_k}{dx}}_{\text{recursive case}}$$

Reverse automatic differentiation

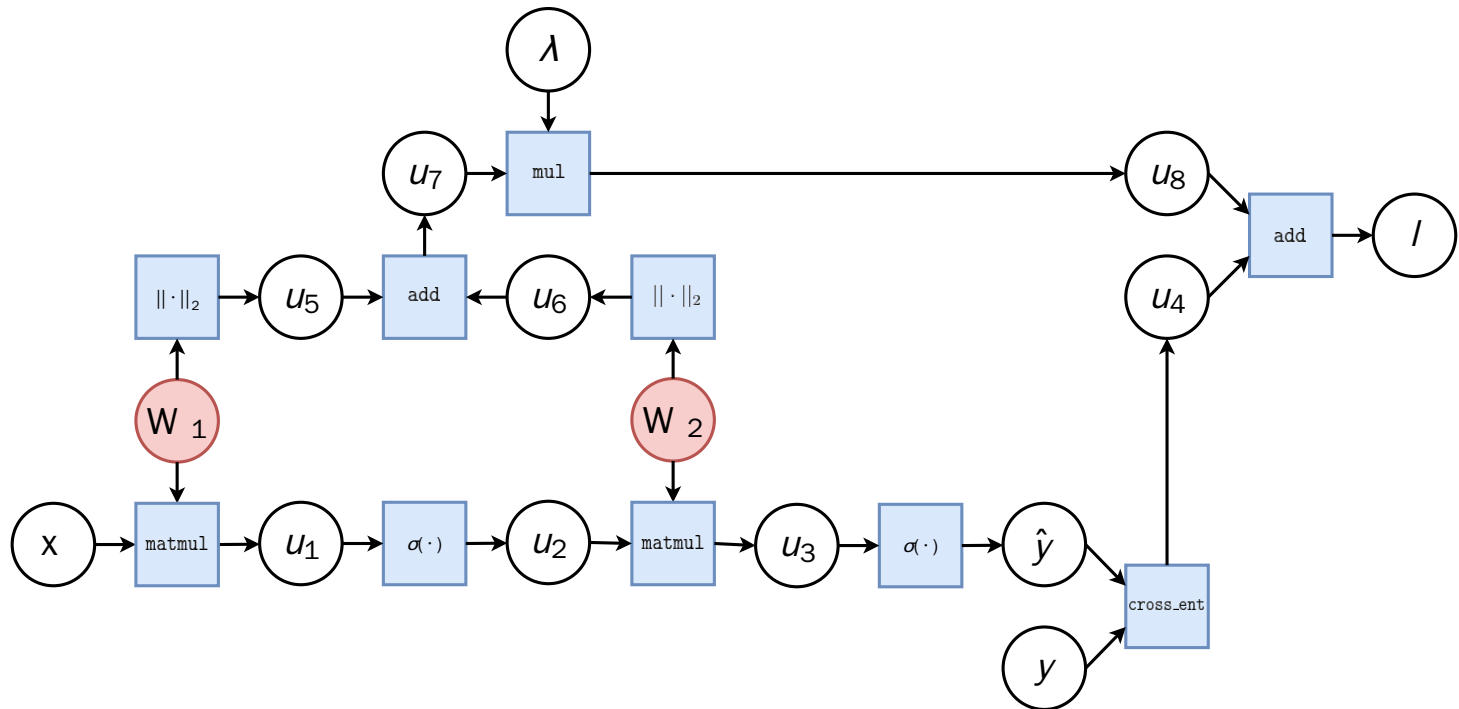
- Since a neural network is a **composition of differentiable functions**, the total derivatives of the loss can be evaluated backward, by applying the chain rule recursively over its computational graph.
- The implementation of this procedure is called reverse **automatic differentiation**.

Let us consider a simplified 2-layer MLP and the following loss function:

$$f(\mathbf{x}; \mathbf{W}_1, \mathbf{W}_2) = \sigma(\mathbf{W}_2^T \sigma(\mathbf{W}_1^T \mathbf{x}))$$
$$\ell(y, \hat{y}; \mathbf{W}_1, \mathbf{W}_2) = \text{cross_ent}(y, \hat{y}) + \lambda (\|\mathbf{W}_1\|_2 + \|\mathbf{W}_2\|_2)$$

for $\mathbf{x} \in \mathbb{R}^p$, $y \in \mathbb{R}$, $\mathbf{W}_1 \in \mathbb{R}^{p \times q}$ and $\mathbf{W}_2 \in \mathbb{R}^q$.

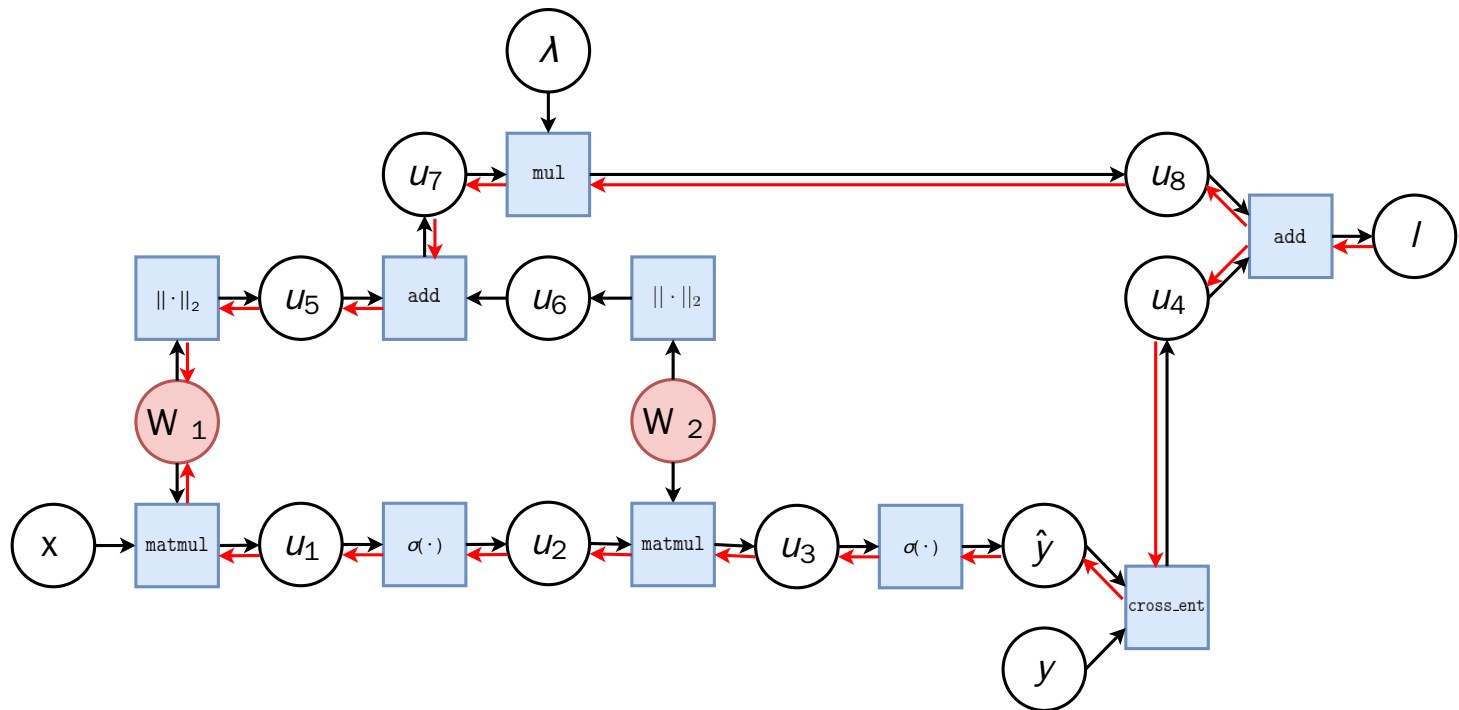
In the **forward pass**, intermediate values are all computed from inputs to outputs, which results in the annotated computational graph below:

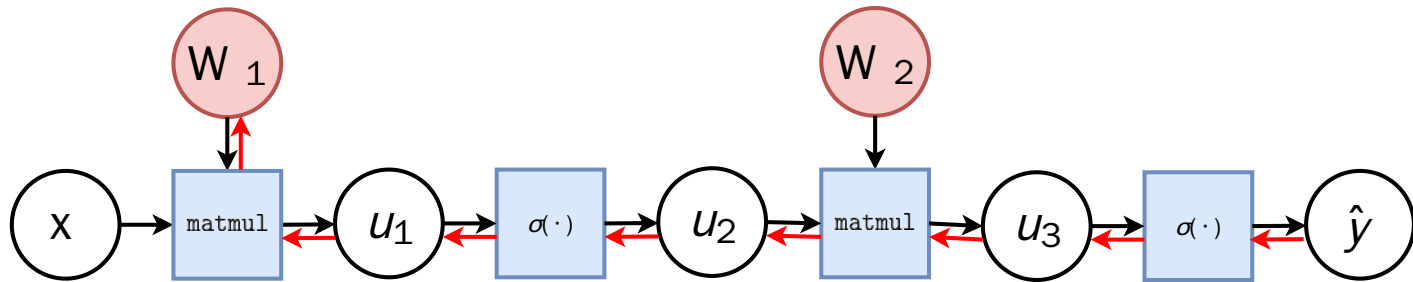


The total derivative can be computed through a **backward pass**, by walking through all paths from outputs to parameters in the computational graph and accumulating the terms. For example, for $\frac{dl}{d\mathbf{W}_1}$ we have:

$$\frac{dl}{d\mathbf{W}_1} = \frac{\partial l}{\partial u_8} \frac{du_8}{d\mathbf{W}_1} + \frac{\partial l}{\partial u_4} \frac{du_4}{d\mathbf{W}_1}$$

$$\frac{du_8}{d\mathbf{W}_1} = \dots$$





Let us zoom in on the computation of the network output \hat{y} and of its derivative with respect to \mathbf{W}_1 .

- **Forward pass:** values u_1, u_2, u_3 and \hat{y} are computed by traversing the graph from inputs to outputs given \mathbf{x}, \mathbf{W}_1 and \mathbf{W}_2 .
- **Backward pass:** by the chain rule we have

$$\begin{aligned} \frac{d\hat{y}}{d\mathbf{W}_1} &= \frac{\partial \hat{y}}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial \mathbf{W}_1} \\ &= \frac{\partial \sigma(u_3)}{\partial u_3} \frac{\partial \mathbf{W}_2^T u_2}{\partial u_2} \frac{\partial \sigma(u_1)}{\partial u_1} \frac{\partial \mathbf{W}_1^T u_1}{\partial \mathbf{W}_1} \end{aligned}$$

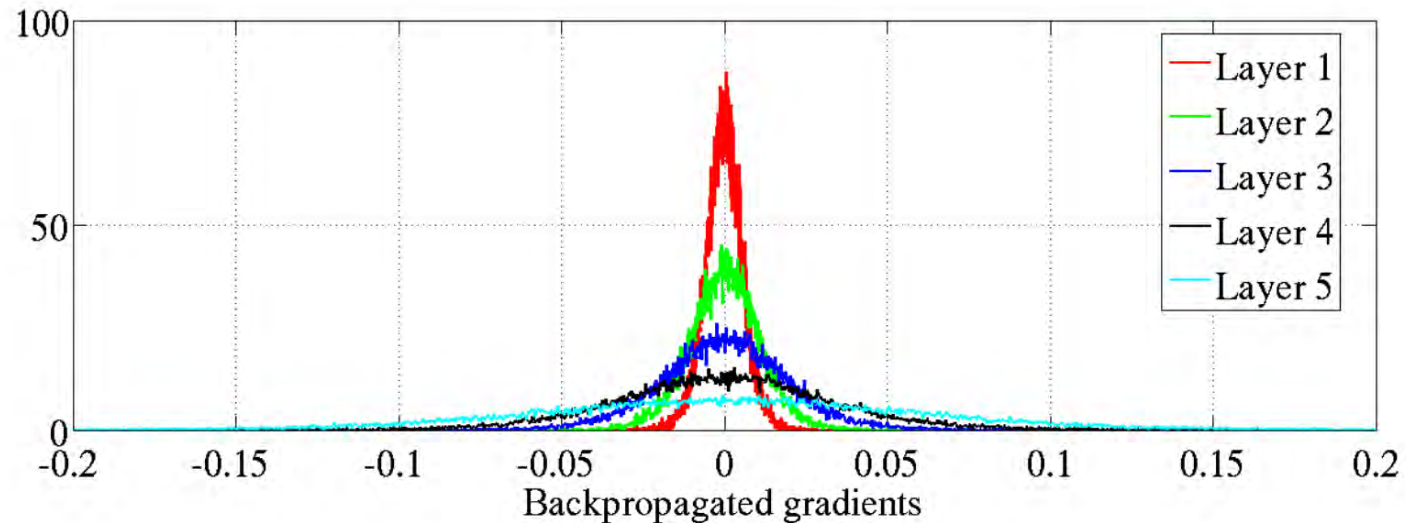
Note how evaluating the partial derivatives requires the intermediate values computed forward.

- This algorithm is also known as **backpropagation**.
- An equivalent procedure can be defined to evaluate the derivatives in **forward mode**, from inputs to outputs.
- Since differentiation is a linear operator, automatic differentiation can be implemented efficiently in terms of tensor operations.

Vanishing gradients

Training deep MLPs with many layers has for long (pre-2011) been very difficult due to the **vanishing gradient** problem.

- Small gradients slow down, and eventually block, stochastic gradient descent.
- This results in a limited capacity of learning.



*Backpropagated gradients normalized histograms (Glorot and Bengio, 2010).
Gradients for layers far from the output vanish to zero.*

Let us consider a simplified 3-layer MLP, with $x, w_1, w_2, w_3 \in \mathbb{R}$, such that

$$f(x; w_1, w_2, w_3) = \sigma(w_3 \sigma(w_2 \sigma(w_1 x))).$$

Under the hood, this would be evaluated as

$$u_1 = w_1 x$$

$$u_2 = \sigma(u_1)$$

$$u_3 = w_2 u_2$$

$$u_4 = \sigma(u_3)$$

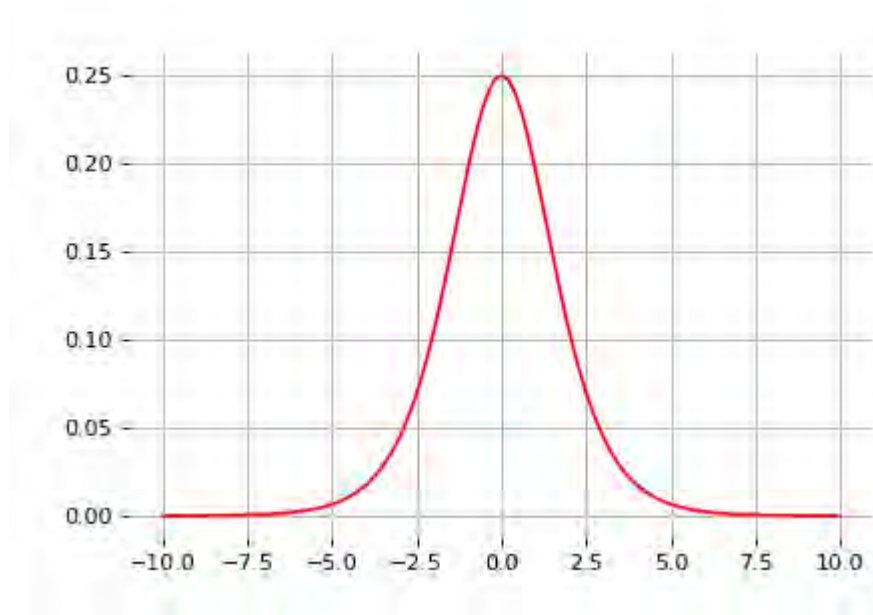
$$u_5 = w_3 u_4$$

$$\hat{y} = \sigma(u_5)$$

and its derivative $\frac{d\hat{y}}{dw_1}$ as

$$\begin{aligned} \frac{d\hat{y}}{dw_1} &= \frac{\partial \hat{y}}{\partial u_5} \frac{\partial u_5}{\partial u_4} \frac{\partial u_4}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial w_1} \\ &= \frac{\partial \sigma(u_5)}{\partial u_5} w_3 \frac{\partial \sigma(u_3)}{\partial u_3} w_2 \frac{\partial \sigma(u_1)}{\partial u_1} x \end{aligned}$$

The derivative of the sigmoid activation function σ is:



$$\frac{d\sigma}{dx}(x) = \sigma(x)(1 - \sigma(x))$$

Notice that $0 \leq \frac{d\sigma}{dx}(x) \leq \frac{1}{4}$ for all x .

Assume that weights w_1, w_2, w_3 are initialized randomly from a Gaussian with zero-mean and small variance, such that with high probability $-1 \leq w_i \leq 1$.

Then,

$$\frac{d\hat{y}}{dw_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{\leq \frac{1}{4}} \underbrace{w_3}_{\leq 1} \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{\leq \frac{1}{4}} \underbrace{w_2}_{\leq 1} \underbrace{\frac{\sigma(u_1)}{\partial u_1}}_{\leq \frac{1}{4}} x$$

This implies that the gradient $\frac{d\hat{y}}{dw_1}$ **exponentially** shrinks to zero as the number of layers in the network increases.

Hence the vanishing gradient problem.

- In general, bounded activation functions (sigmoid, tanh, etc) are prone to the vanishing gradient problem.
- Note the importance of a proper initialization scheme.

Rectified linear units

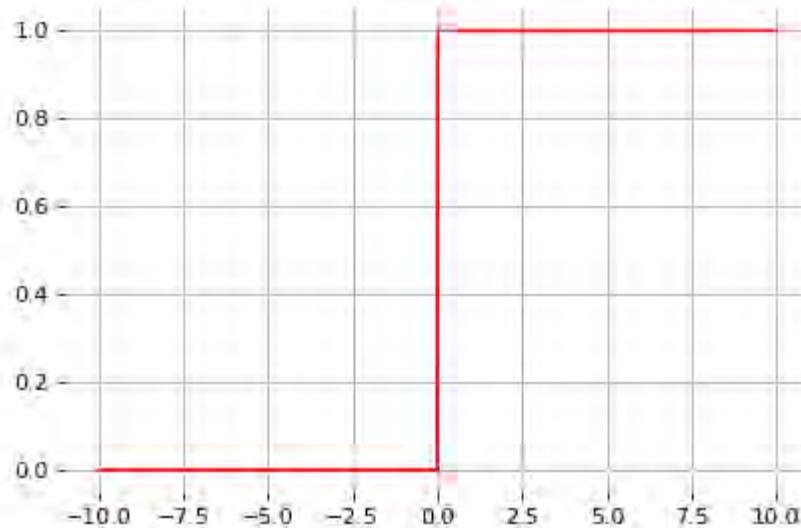
Instead of the sigmoid activation function, modern neural networks are for most based on **rectified linear units** (ReLU) (Glorot et al, 2011):

$$\text{ReLU}(x) = \max(0, x)$$



Note that the derivative of the ReLU function is

$$\frac{d}{dx}\text{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$



For $x = 0$, the derivative is undefined. In practice, it is set to zero.

Therefore,

$$\frac{d\hat{y}}{dw_1} = \underbrace{\frac{\partial\sigma(u_5)}{\partial u_5}}_{=1} w_3 \underbrace{\frac{\partial\sigma(u_3)}{\partial u_3}}_{=1} w_2 \underbrace{\frac{\partial\sigma(u_1)}{\partial u_1}}_{=1} x$$

This **solves** the vanishing gradient problem, even for deep networks! (provided proper initialization)

Note that:

- The ReLU unit dies when its input is negative, which might block gradient descent.
- This is actually a useful property to induce **sparsity**.
- This issue can also be solved using **leaky** ReLUs, defined as

$$\text{LeakyReLU}(x) = \max(\alpha x, x)$$

for a small $\alpha \in \mathbb{R}^+$ (e.g., $\alpha = 0.1$).

Universal approximation

Theorem. (Cybenko 1989; Hornik et al, 1991) Let $\sigma(\cdot)$ be a bounded, non-constant continuous function. Let I_p denote the p -dimensional hypercube, and $C(I_p)$ denote the space of continuous functions on I_p . Given any $f \in C(I_p)$ and $\epsilon > 0$, there exists $q > 0$ and $v_i, w_i, b_i, i = 1, \dots, q$ such that

$$F(x) = \sum_{i \leq q} v_i \sigma(w_i^T x + b_i)$$

satisfies

$$\sup_{x \in I_p} |f(x) - F(x)| < \epsilon.$$

- It guarantees that even a single hidden-layer network can represent any classification problem in which the boundary is locally linear (smooth);
- It does not inform about good/bad architectures, nor how they relate to the optimization procedure.
- The universal approximation theorem generalizes to any non-polynomial (possibly unbounded) activation function, including the ReLU (Leshno, 1993).

Theorem (Barron, 1992) The mean integrated square error between the estimated network \hat{F} and the target function f is bounded by

$$O\left(\frac{C_f^2}{q} + \frac{qp}{N} \log N\right)$$

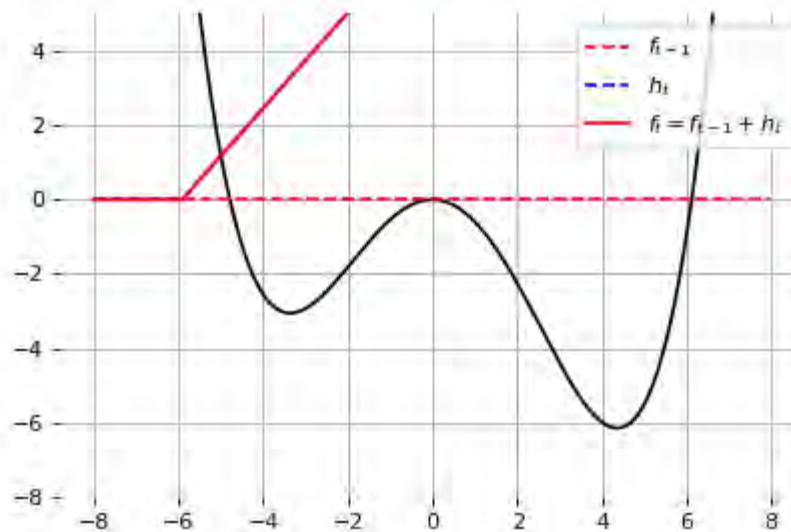
where N is the number of training points, q is the number of neurons, p is the input dimension, and C_f measures the global smoothness of f .

- Combines approximation and estimation errors.
- Provided enough data, it guarantees that adding more neurons will result in a better approximation.

Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

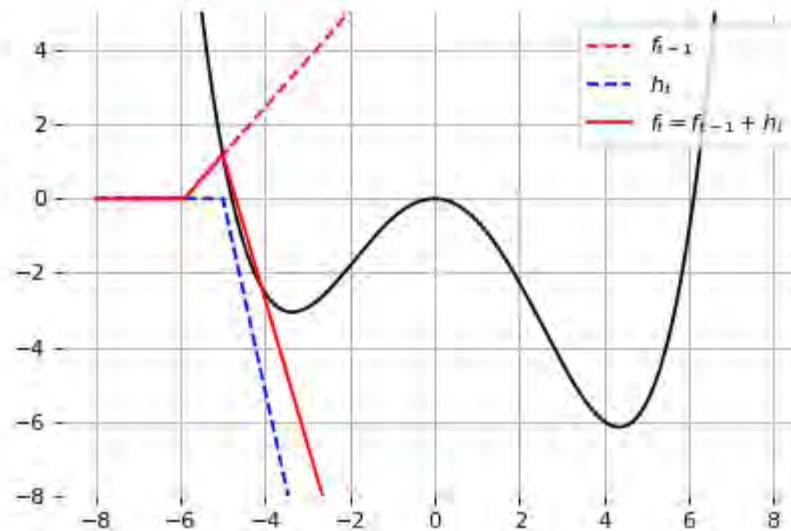
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

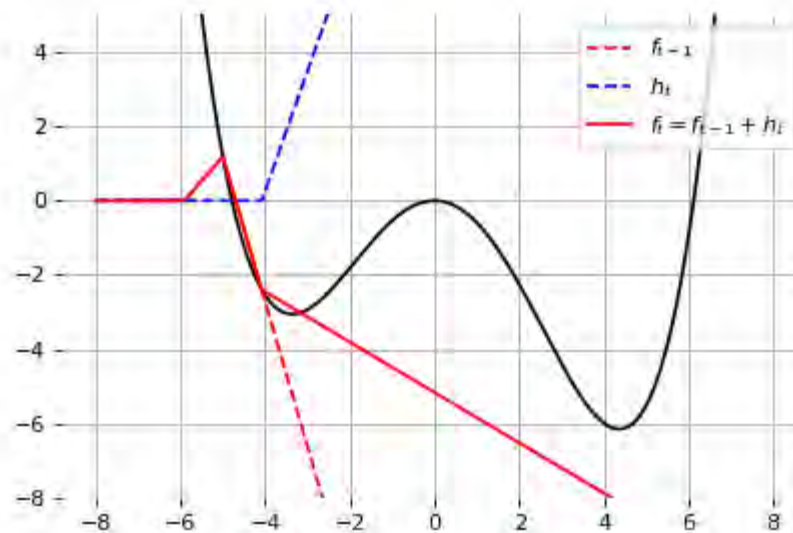
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

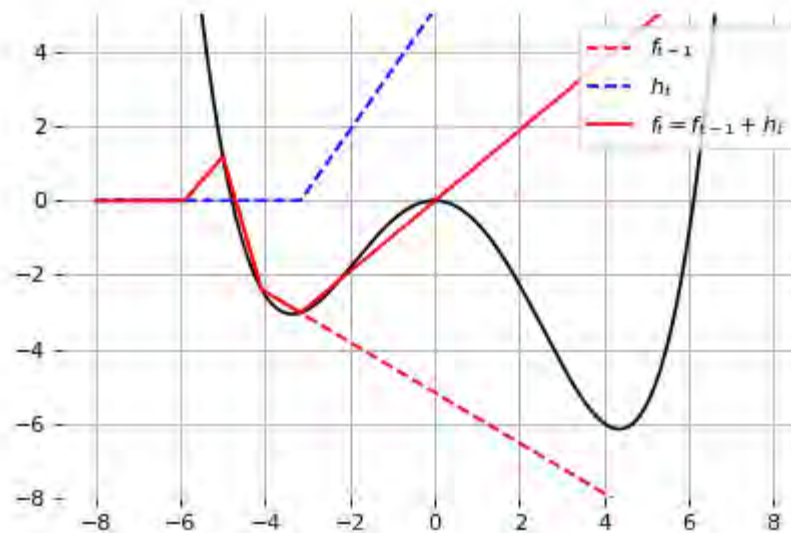
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

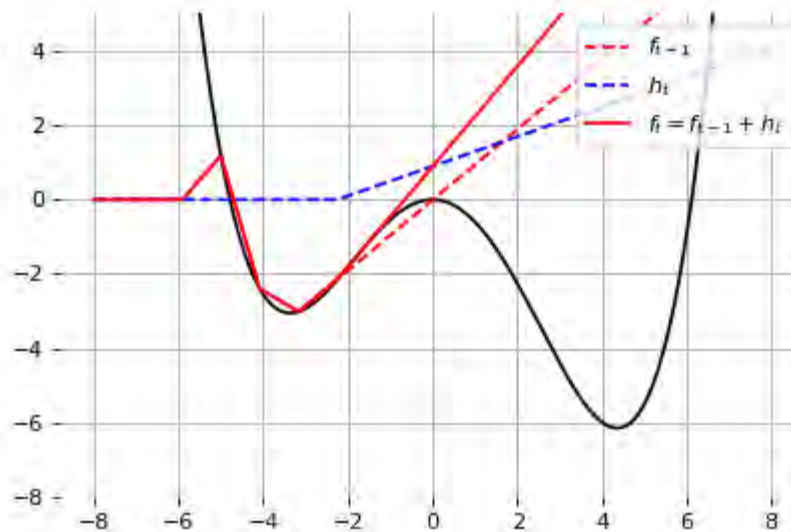
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

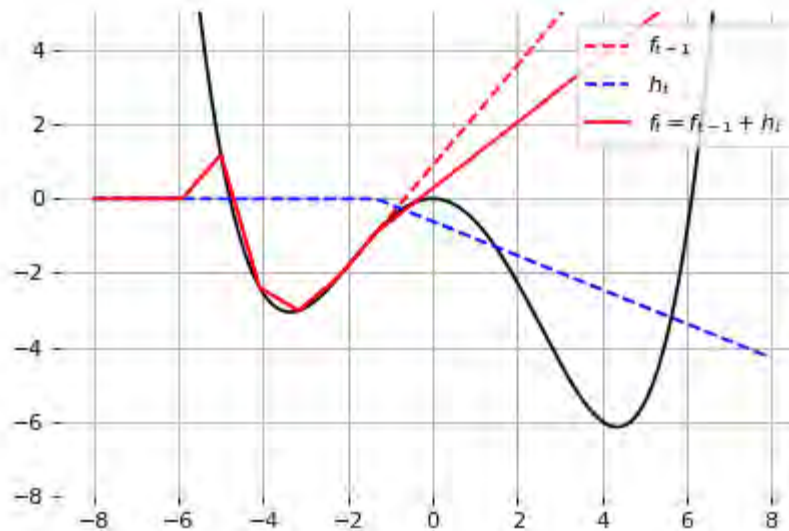
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

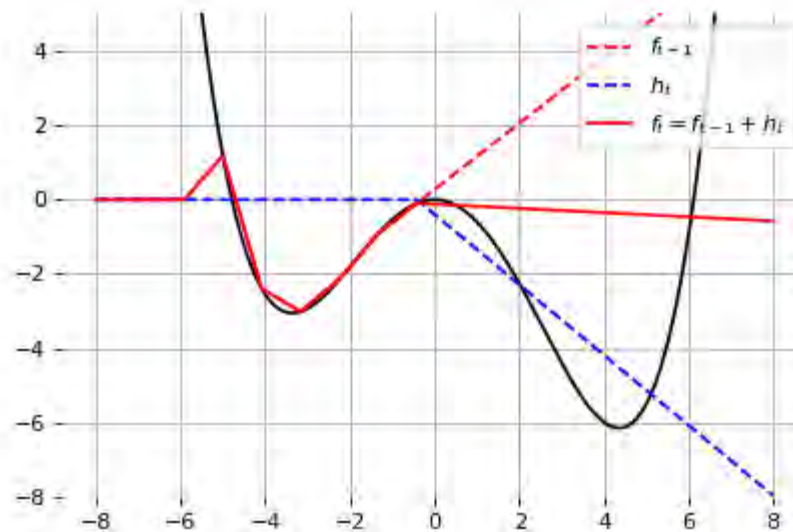
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

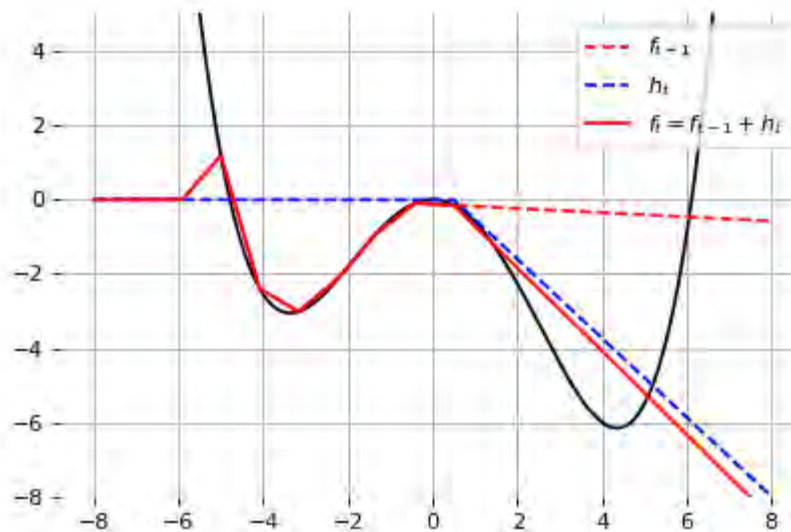
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

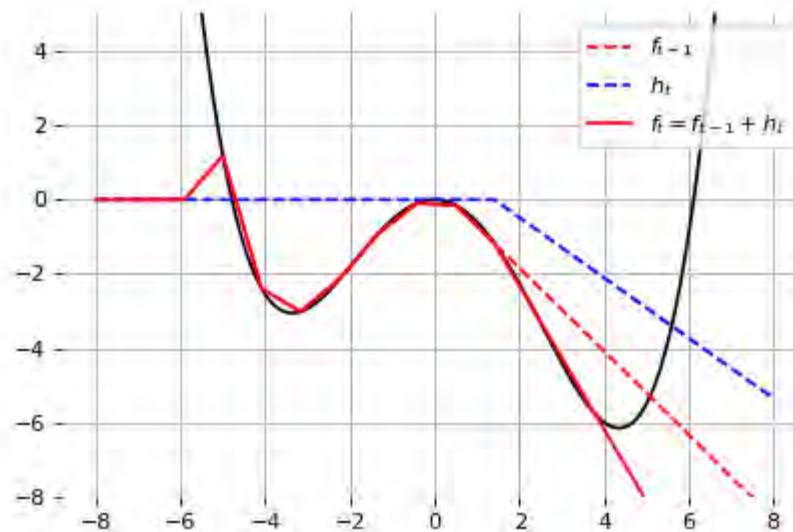
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

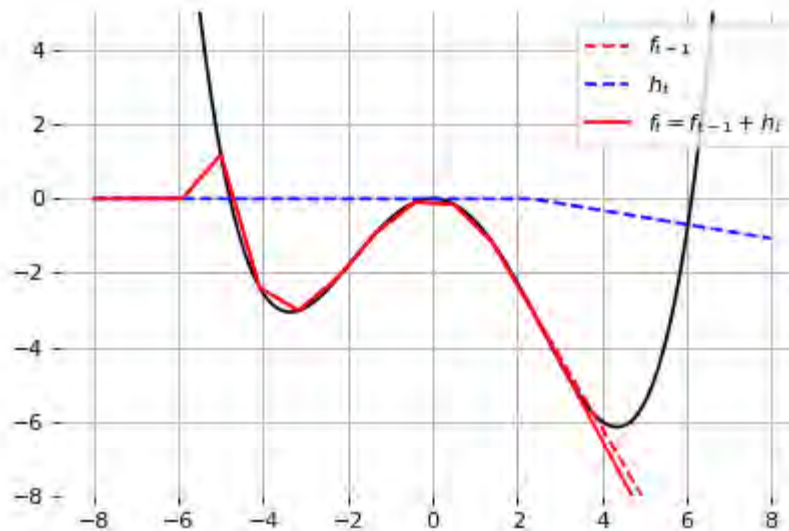
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

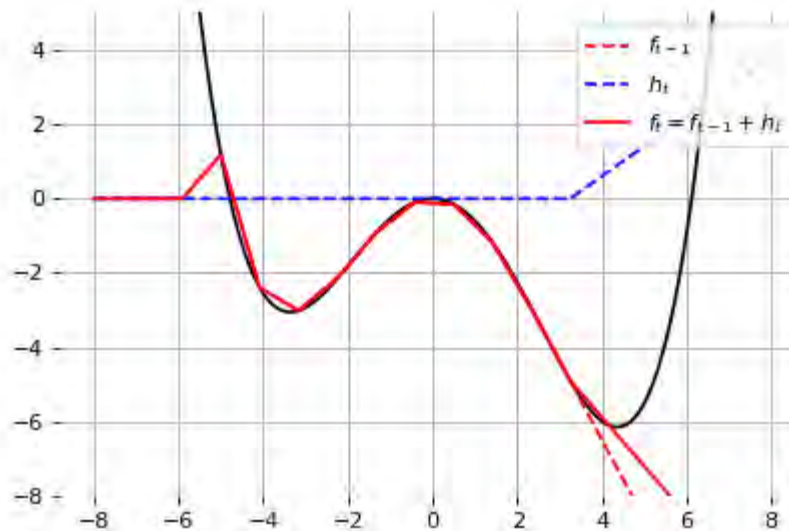
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

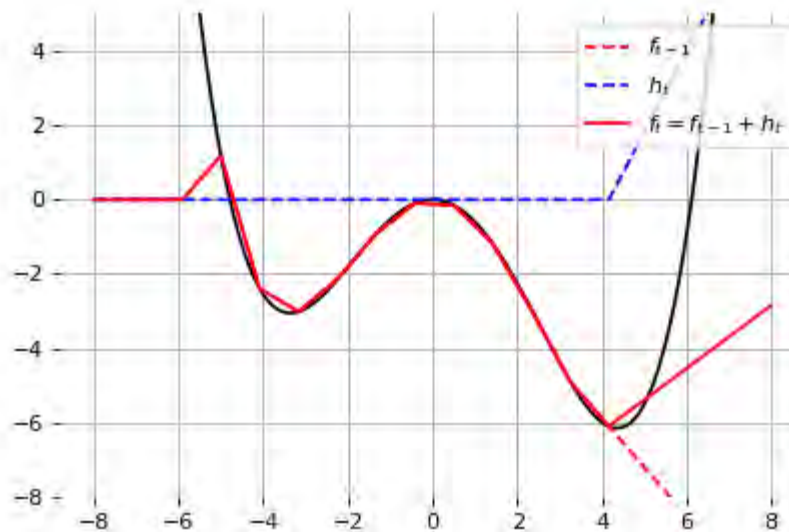
This model can approximate any smooth 1D function, provided enough hidden units.



Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

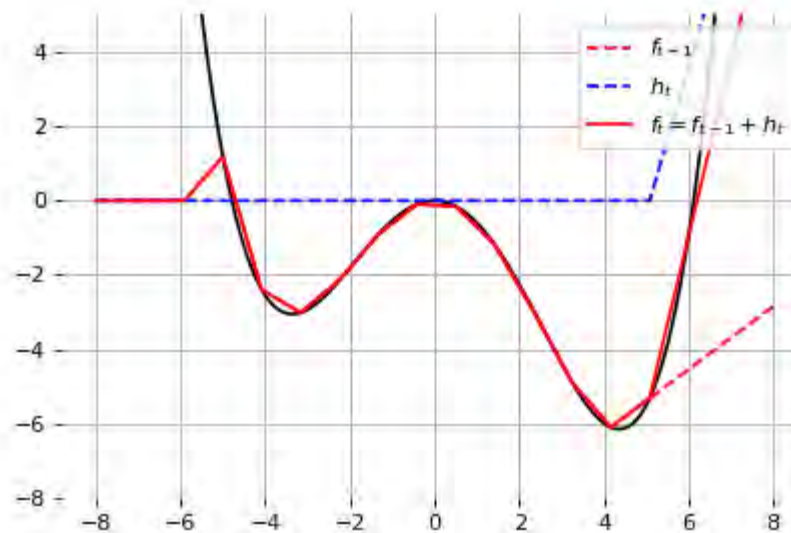
This model can approximate any smooth 1D function, provided enough hidden units.



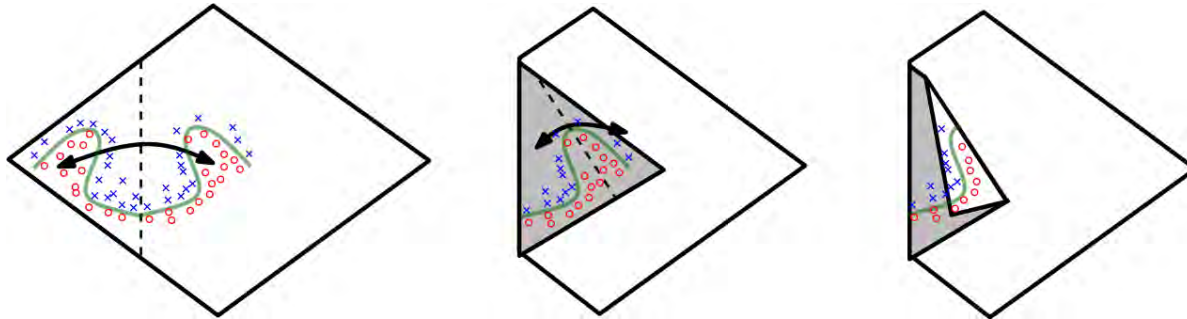
Let us consider the 1-layer MLP

$$f(x) = \sum w_i \text{ReLU}(x + b_i).$$

This model can approximate any smooth 1D function, provided enough hidden units.



Effect of depth



Theorem (Montúfar et al, 2014) A rectifier neural network with p input units and L hidden layers of width $q \geq p$ can compute functions that have $\Omega\left(\left(\frac{q}{p}\right)^{(L-1)p} q^p\right)$ linear regions.

- That is, the number of linear regions of deep models grows **exponentially** in L and polynomially in q .
- Even for small values of L and q , deep rectifier models are able to produce substantially more linear regions than shallow rectifier models.

Deep learning

Recent advances and model architectures in deep learning are built on a natural generalization of a neural network: **a graph of tensor operators**, taking advantage of

- the chain rule
- stochastic gradient descent
- convolutions
- parallel operations on GPUs.

This does not differ much from networks from the 90s, as covered in Today's lecture.

This generalization allows to **compose** and design complex networks of operators, possibly dynamically, dealing with images, sound, text, sequences, etc. and to train them **end-to-end**.

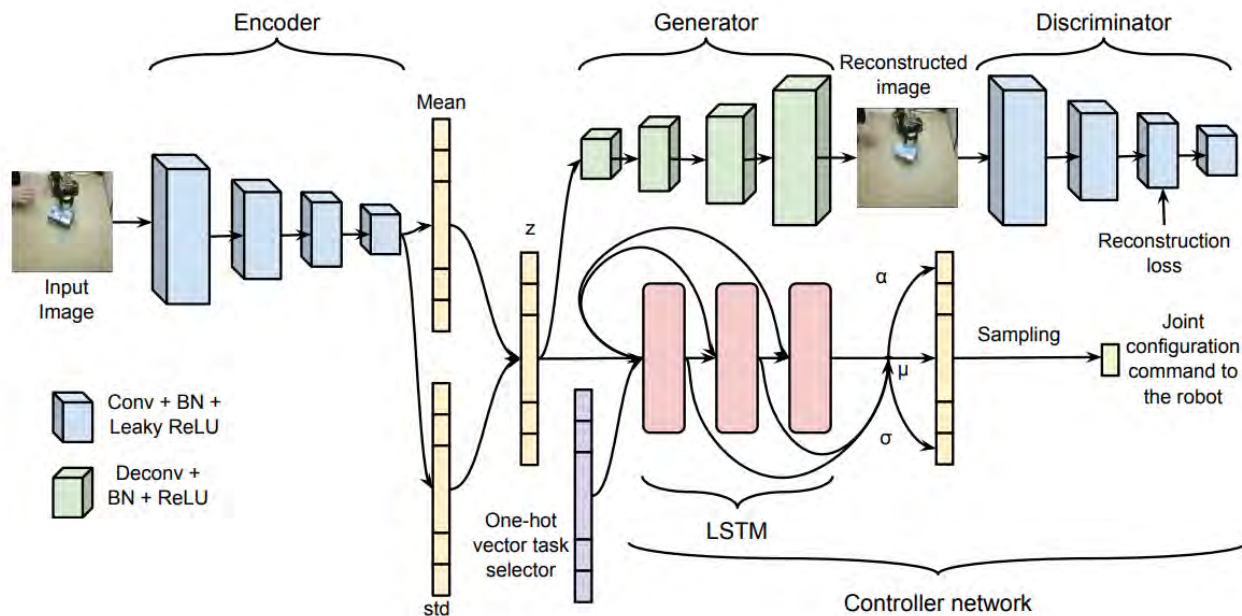


Fig. 2: Our proposed architecture for multi-task robot manipulation learning. The neural network consists of a controller network that outputs joint commands based on a multi-modal autoregressive estimator and a VAE-GAN autoencoder that reconstructs the input image. The encoder is shared between the VAE-GAN autoencoder and the controller network and extracts some shared features that will be used for two tasks (reconstruction and controlling the robot).

The end.

References

- Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6), 386.
- Bottou, L., & Bousquet, O. (2008). The tradeoffs of large scale learning. In *Advances in neural information processing systems* (pp. 161-168).
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *nature*, 323(6088), 533.
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4), 303-314.
- 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* (pp. 2924-2932).

Deep Learning

Lecture 3: Convolutional networks

Prof. Gilles Louppe
g.louppe@uliege.be

Today

How to **make neural networks see**?

- A little history
- Convolutions
- Convolutional network architectures
- What is really happening?

A little history

Adapted from Yannis Avrithis, "Lecture 1: Introduction", [Deep Learning for vision](#), 2018.

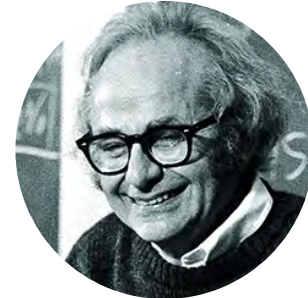
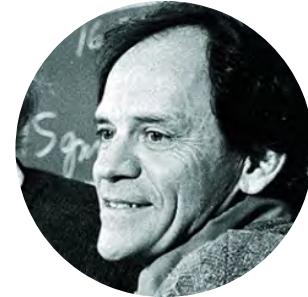
J. Physiol. (1959) 148, 574-591

RECEPTIVE FIELDS OF SINGLE NEURONES IN
THE CAT'S STRIATE CORTEX

BY D. H. HUBEL* AND T. N. WIESEL*

*From the Wilmer Institute, The Johns Hopkins Hospital and
University, Baltimore, Maryland, U.S.A.*

(Received 22 April 1959)



Visual perception (Hubel and Wiesel, 1959-1962)

- David Hubel and Torsten Wiesel discover the neural basis of **visual perception**.
- Nobel Prize of Medicine in 1981 for this work.



Hubel & Wiesel 1: Intro



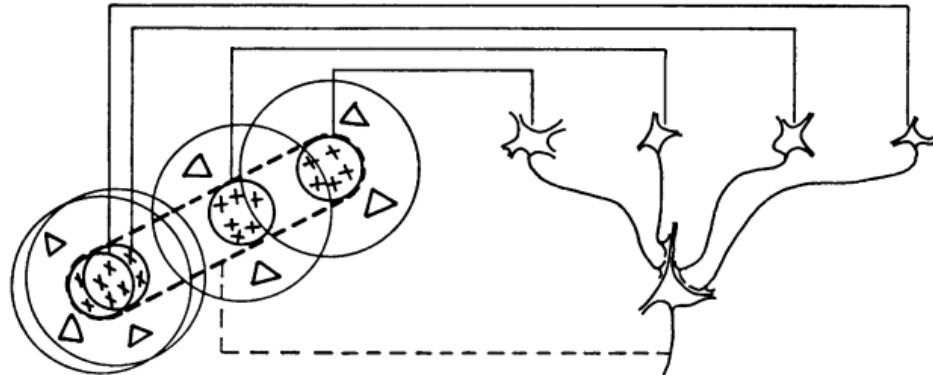
Watch later



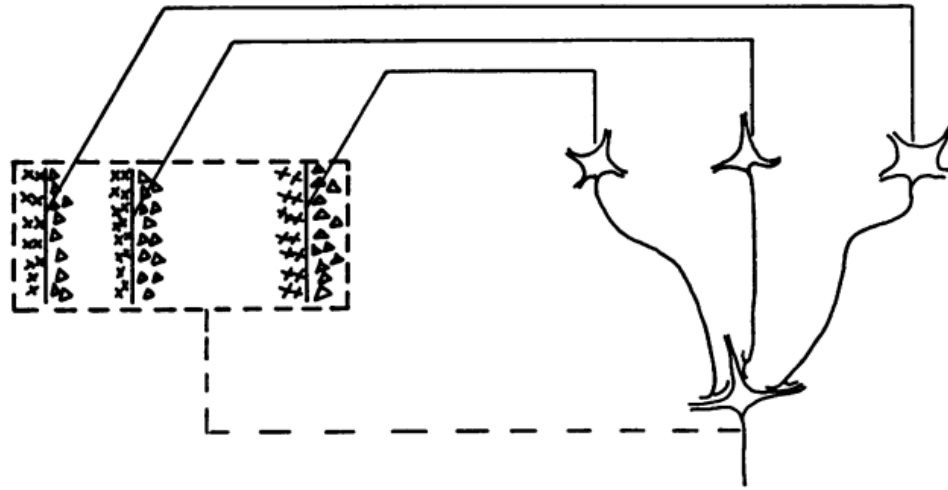
Share



Hubel and Wiesel

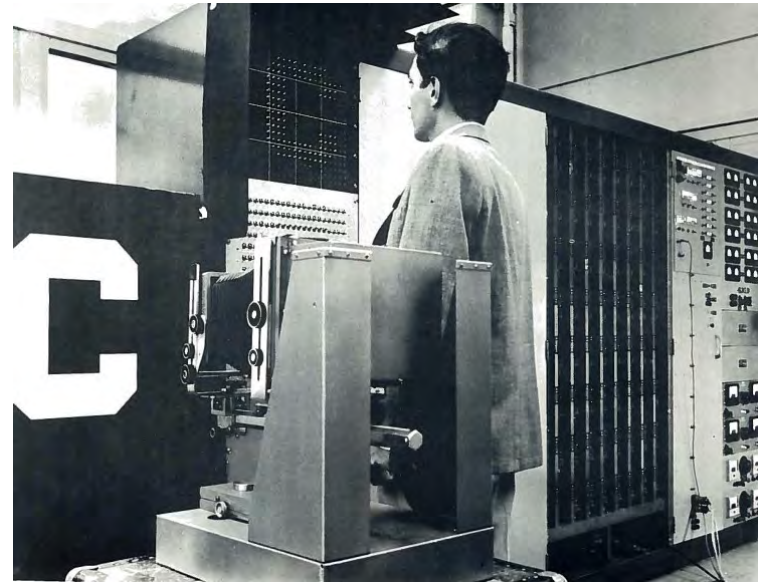
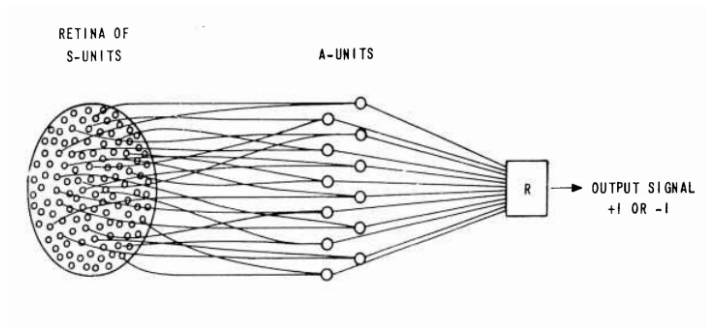


Text-fig. 19. Possible scheme for explaining the organization of simple receptive fields. A large number of lateral geniculate cells, of which four are illustrated in the upper right in the figure, have receptive fields with 'on' centres arranged along a straight line on the retina. All of these project upon a single cortical cell, and the synapses are supposed to be excitatory. The receptive field of the cortical cell will then have an elongated 'on' centre indicated by the interrupted lines in the receptive-field diagram to the left of the figure.



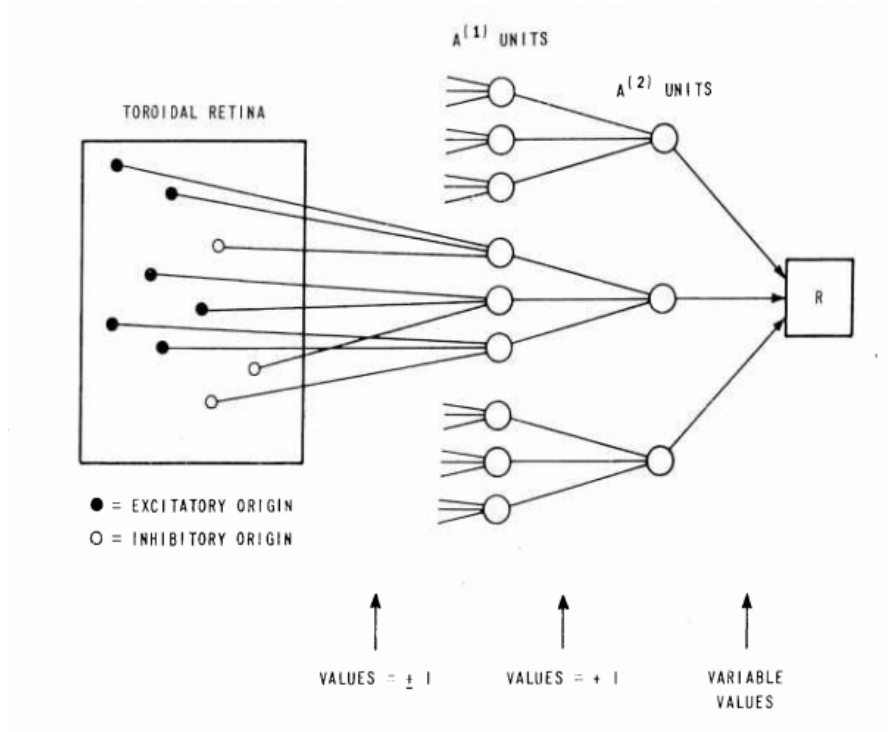
Text-fig. 20. Possible scheme for explaining the organization of complex receptive fields. A number of cells with simple fields, of which three are shown schematically, are imagined to project to a single cortical cell of higher order. Each projecting neurone has a receptive field arranged as shown to the left: an excitatory region to the left and an inhibitory region to the right of a vertical straight-line boundary. The boundaries of the fields are staggered within an area outlined by the interrupted lines. Any vertical-edge stimulus falling across this rectangle, regardless of its position, will excite some simple-field cells, leading to excitation of the higher-order cell.

Perceptron (Rosenblatt, 1959)



The Mark-1 Perceptron:

- Analog circuit implementation of a neural network,
- Parameters as potentiometers.



"If we show the perceptron a stimulus, say a square, and associate a response to that square, this response will immediately generalize perfectly to all transforms of the square under the transformation group [...]."

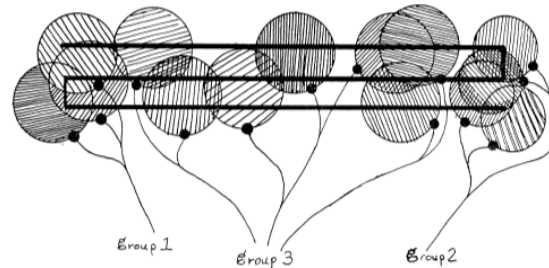
This is quite similar to Hubel and Wiesel's simple and complex cells!

Theorem 0.8: No diameter-limited perceptron can determine whether or not all the parts of any geometric figure are connected to one another! That is, no such perceptron computes $\psi_{\text{CONNECTED}}$.

The proof requires us to consider just four figures



and a diameter-limited perceptron ψ whose support sets have diameters like those indicated by the circles below:



AI winter (Minsky and Papert, 1969+)

- Minsky and Papert redefine the perceptron as a linear classifier,
- Then they prove a series of impossibility results. **AI winter** follows.

Actual Variable	Variable Number (Address)	Category	Major Source	Minor Source
$A(t+1)$	13	sum	12	11
$k_1 A(t)$	12	product	3	1
$k_2 U(t) \left(\frac{A(t)-U(t)}{A(t)+U(t)} \right)^{k_4}$	11	product	10	4
$U(t) \left(\frac{A(t)-U(t)}{A(t)+U(t)} \right)^{k_4}$	10	product	9	2
$\left(\frac{A(t)-U(t)}{A(t)+U(t)} \right)^{k_4}$	9	power	8	5
$\frac{A(t)-U(t)}{A(t)+U(t)}$	8	ratio	7	6
$A(t)-U(t)$	7	difference	1	2
$A(t)+U(t)$	6	sum	1	2
k_4	5	parameter	-	-
k_2	4	parameter	-	-
k_1	3	parameter	-	-
$U(t)$	2	given	-	-
$A(t)$	1	given	-	-

Automatic differentiation (Werbos, 1974)

- Formulate an arbitrary function as computational graph.
- Dynamic feedback: compute symbolic derivatives by dynamic programming.

Neocognitron (Fukushima, 1980)

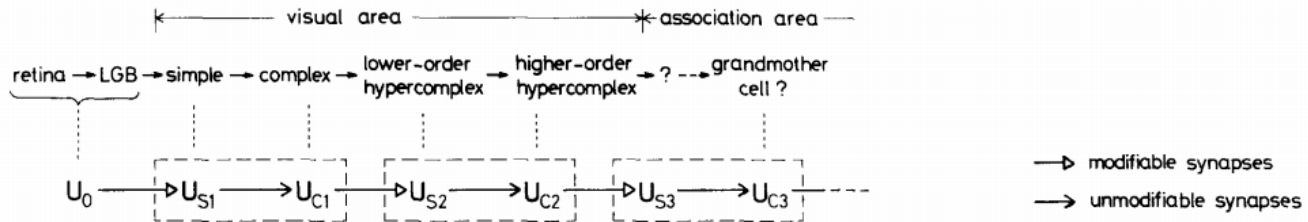


Fig. 1. Correspondence between the hierarchy model by Hubel and Wiesel, and the neural network of the neocognitron

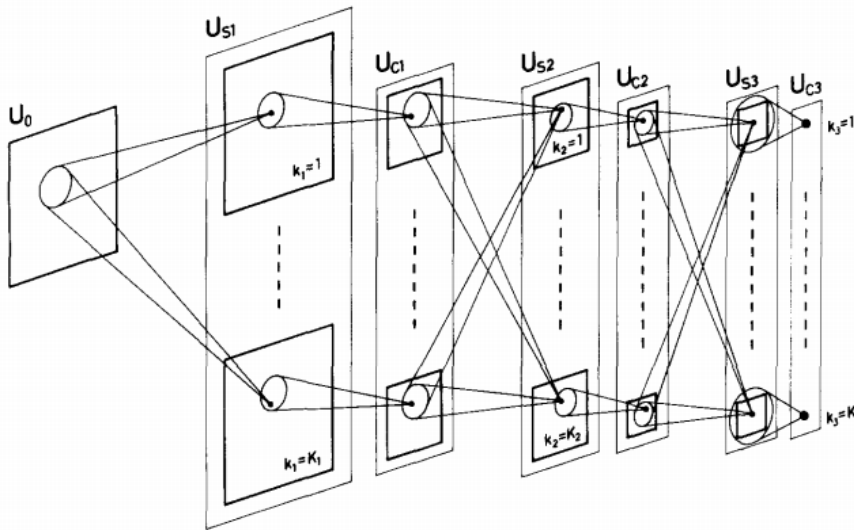
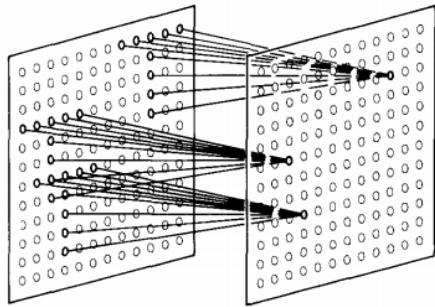
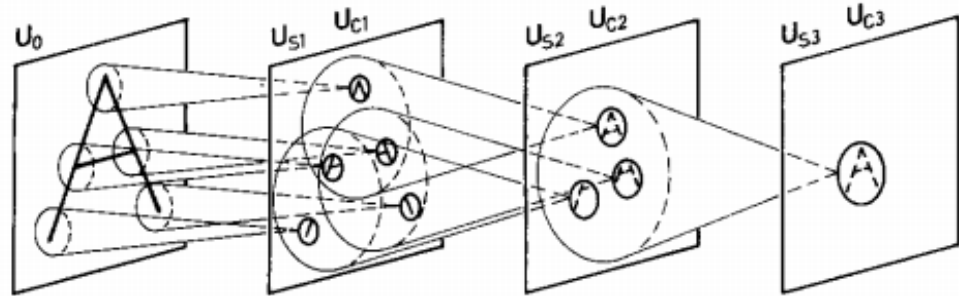


Fig. 2. Schematic diagram illustrating the interconnections between layers in the neocognitron

Fukushima proposes a direct neural network implementation of the hierarchy model of the visual nervous system of Hubel and Wiesel.



Convolutions



Feature hierarchy

- Built upon **convolutions** and enables the composition of a **feature hierarchy**.
- Biologically-inspired training algorithm, which proves to be largely inefficient.

Backpropagation (Rumelhart et al, 1986)

- Introduce **backpropagation** in multi-layer networks with sigmoid non-linearities and sum of squares loss function.
- Advocate batch gradient descent for supervised learning.
- Discuss online gradient descent, momentum and random initialization.
- Depart from **biologically plausible** training algorithms.

The backward pass starts by computing $\partial E/\partial y$ for each of the output units. Differentiating equation (3) for a particular case, c , and suppressing the index c gives

$$\partial E/\partial y_j = y_j - d_j \quad (4)$$

We can then apply the chain rule to compute $\partial E/\partial x_j$

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot dy_j/dx_j$$

Differentiating equation (2) to get the value of dy_j/dx_j and substituting gives

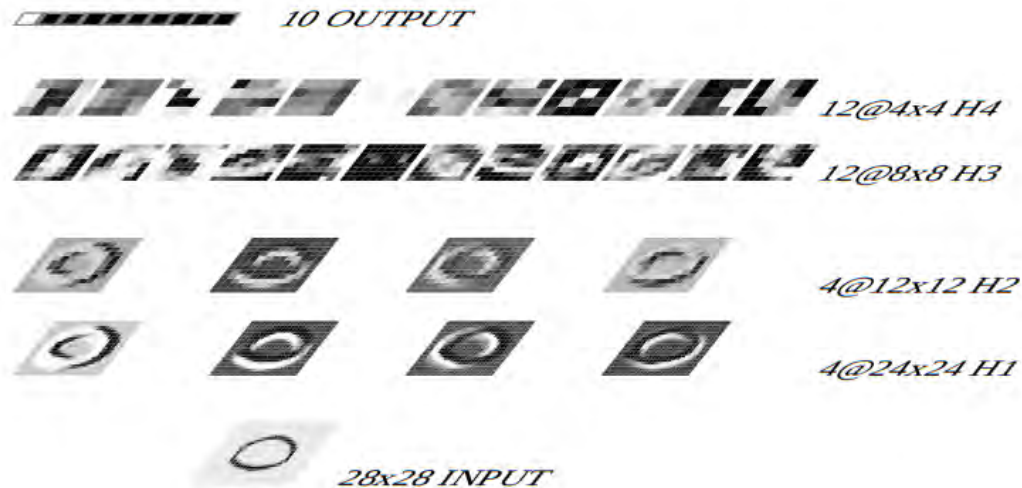
$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j(1 - y_j) \quad (5)$$

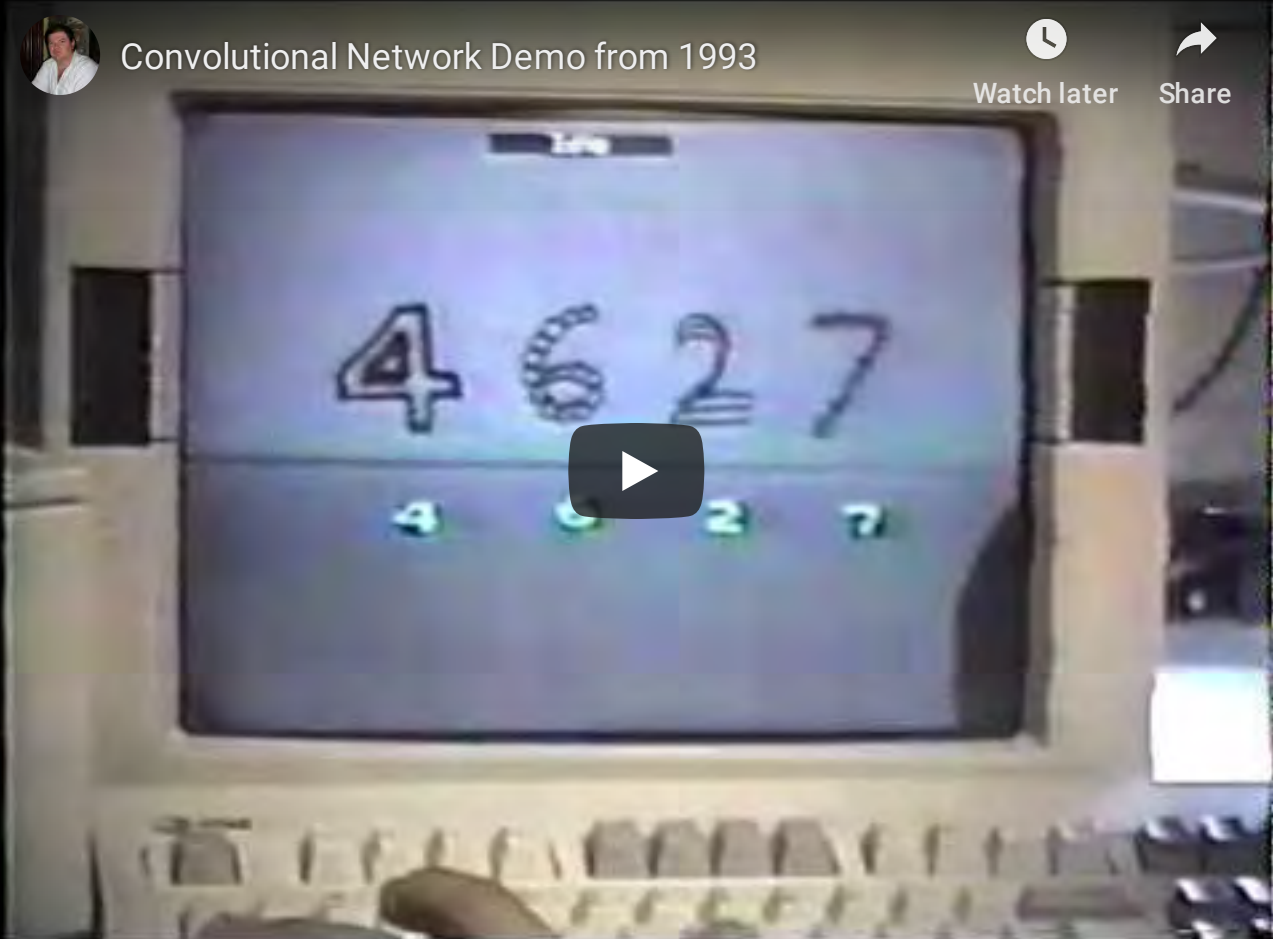
This means that we know how a change in the total input x to an output unit will affect the error. But this total input is just a linear function of the states of the lower level units and it is also a linear function of the weights on the connections, so it is easy to compute how the error will be affected by changing these states and weights. For a weight w_{ji} , from i to j the derivative is

$$\begin{aligned} \partial E/\partial w_{ji} &= \partial E/\partial x_j \cdot \partial x_j/\partial w_{ji} \\ &= \partial E/\partial x_j \cdot y_i \end{aligned} \quad (6)$$

Convolutional networks (LeCun, 1990)

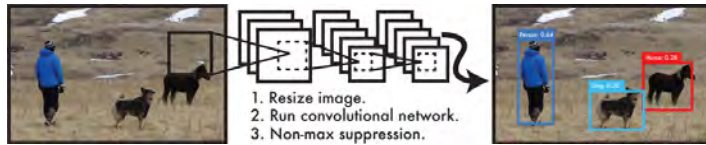
- Train a convolutional network by backpropagation.
- Advocate end-to-end feature learning for image classification.



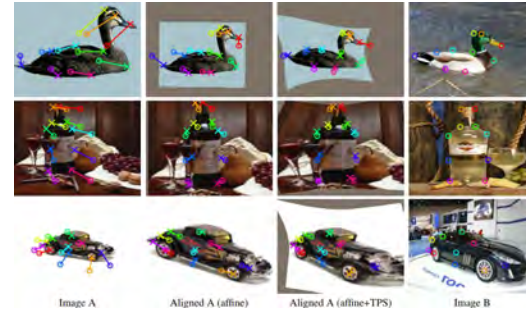


LeNet-1 (LeCun et al, 1993)

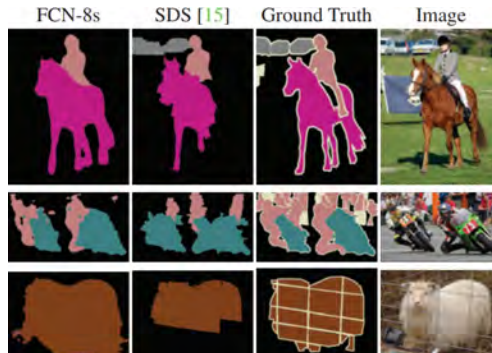
Convolutional networks are now **used everywhere in vision**.



Object detection
(Redmon et al, 2015)



Geometric matching
(Rocco et al, 2017)



Semantic segmentation
(Long et al, 2015)

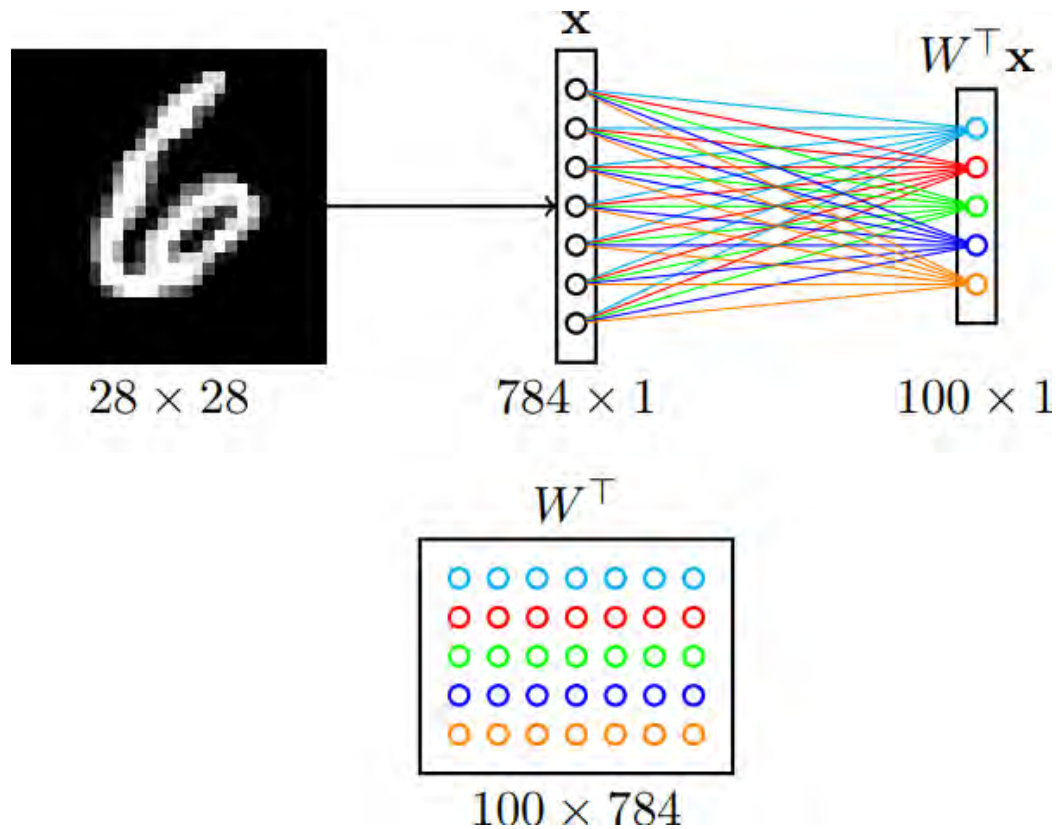


Instance segmentation
(He et al, 2017)

... but also in many other applications, including:

- speech recognition and synthesis
- natural language processing
- protein/DNA binding prediction
- or more generally, any problem **with a spatial** (or sequential) **structure**.

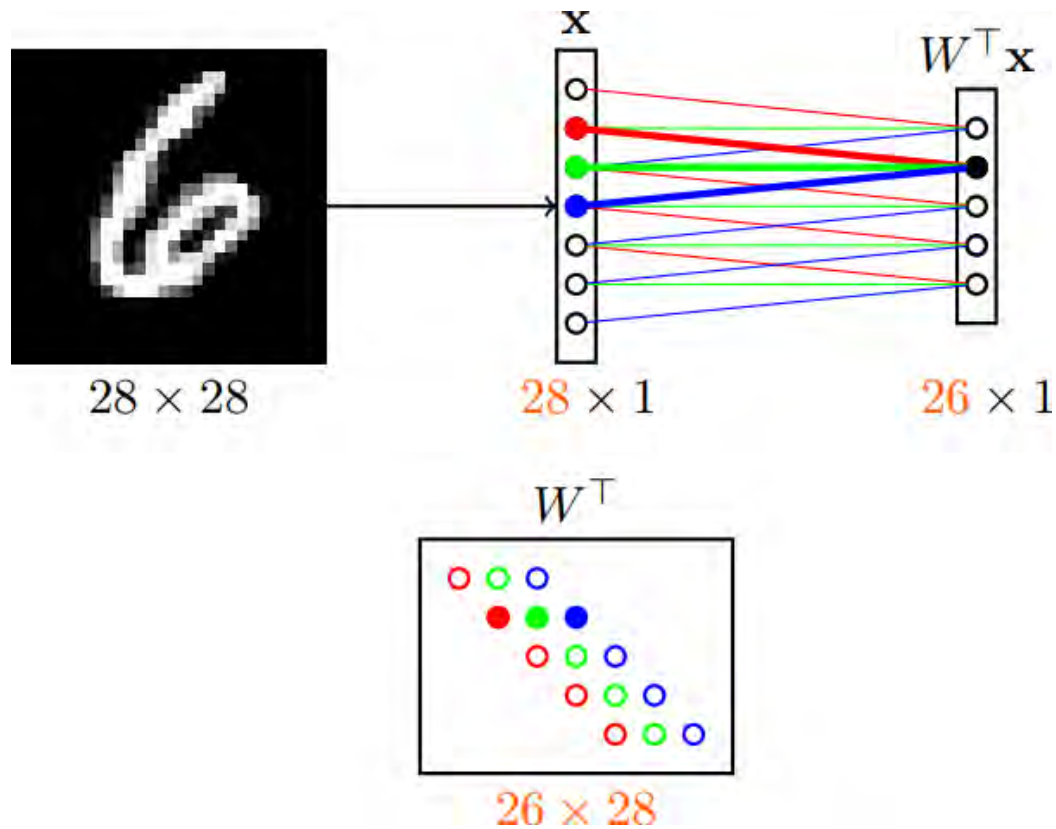
Convolutions



Let us consider the first layer of a MLP taking images as input. What are the problems with this architecture?

Issues

- Too many parameters: $100 \times 784 + 100$.
 - What if images are $640 \times 480 \times 3$?
 - What if the first layer counts 1000 units?
- Spatial organization of the input is destroyed.
- The network is not invariant to transformations (e.g., translation).



Instead, let us only keep a **sparse** set of connections, where all weights having the same color are **shared**.

- The resulting operation can be seen as **shifting** the same weight triplet (**kernel**).
- The set of inputs seen by each unit is its **receptive field**.

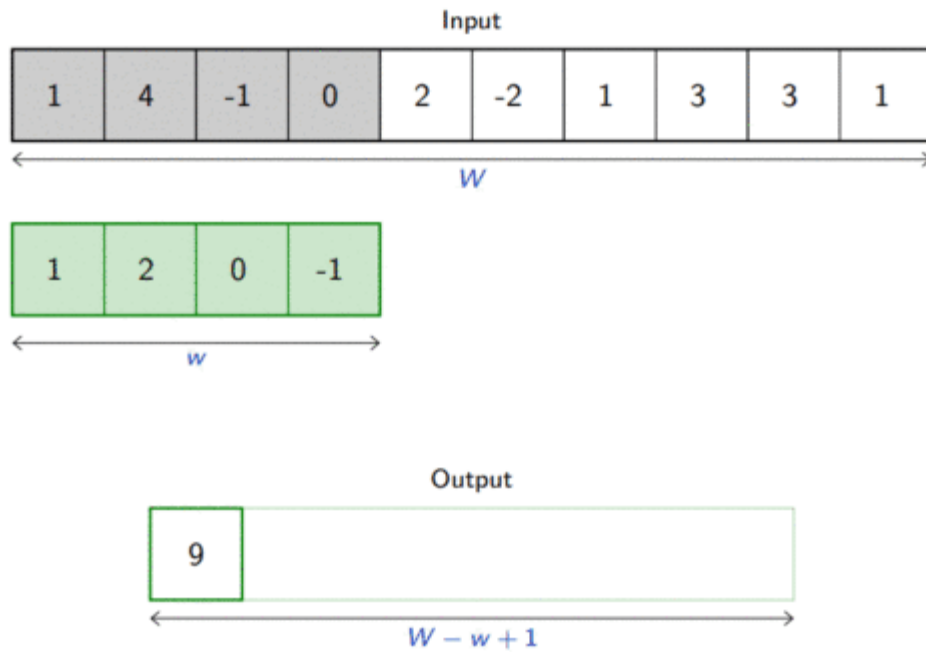
⇒ This is a 1D **convolution**, which can be generalized to more dimensions.

Convolutions

For one-dimensional tensors, given an input vector $\mathbf{x} \in \mathbb{R}^W$ and a convolutional kernel $\mathbf{u} \in \mathbb{R}^w$, the discrete **convolution** $\mathbf{u} \star \mathbf{x}$ is a vector of size $W - w + 1$ such that

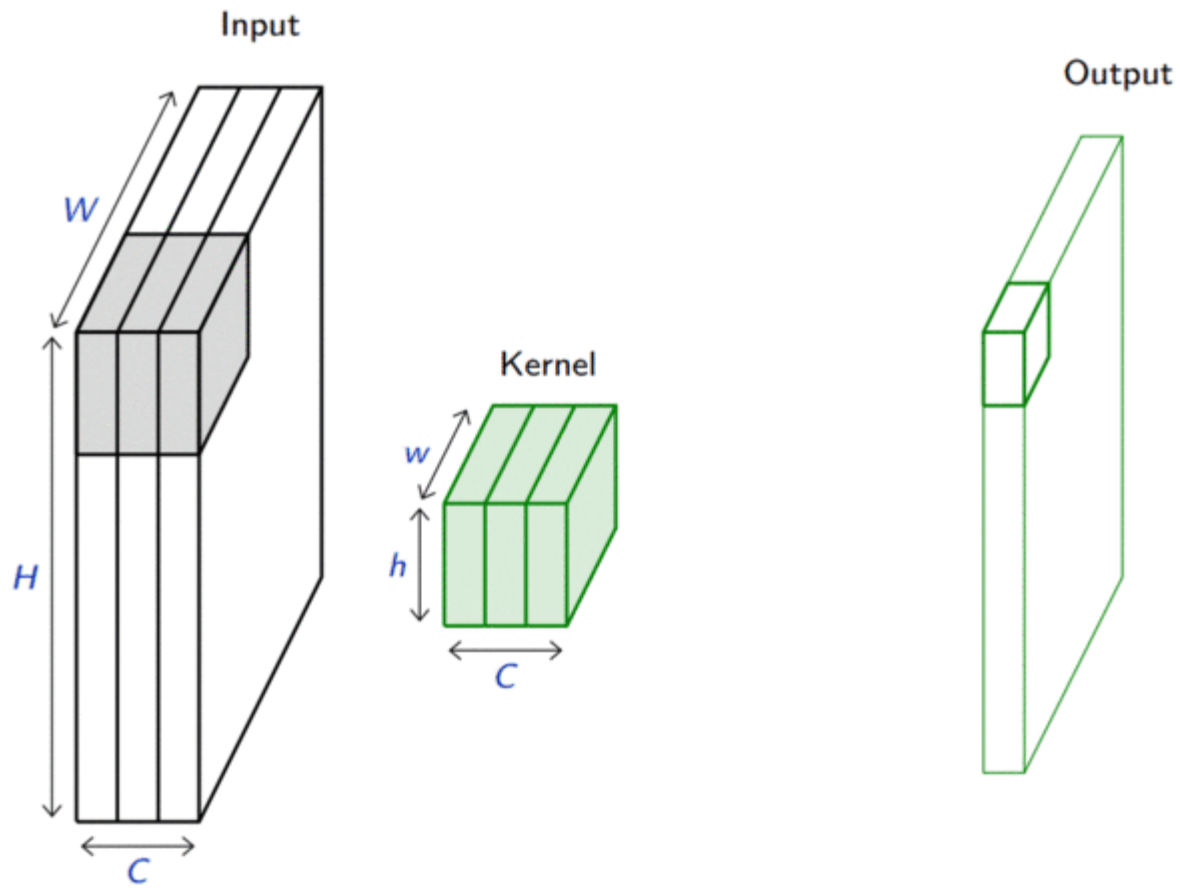
$$(\mathbf{u} \star \mathbf{x})[i] = \sum_{m=0}^{w-1} u_m x_{m+i}.$$

- Technically, \star denotes the **cross-correlation** operator.
- However, most machine learning libraries call it convolution.



Convolutions generalize to multi-dimensional tensors:

- In its most usual form, a convolution takes as input a 3D tensor $\mathbf{x} \in \mathbb{R}^{C \times H \times W}$, called the **input feature map**.
- A kernel $\mathbf{u} \in \mathbb{R}^{C \times h \times w}$ slides across the input feature map, along its height and width. The size $h \times w$ is the size of the receptive field.
- At each location, the element-wise product between the kernel and the input elements it overlaps is computed and the results are summed up.



- The final output \mathbf{o} is a 2D tensor of size $(H - h + 1) \times (W - w + 1)$ called the **output feature map** and such that:

$$\mathbf{o}_{j,i} = \mathbf{b}_{j,i} + \sum_{c=0}^{C-1} (\mathbf{u}_c \star \mathbf{x}_c)[j, i] = \mathbf{b}_{j,i} + \sum_{c=0}^{C-1} \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} \mathbf{u}_{c,n,m} \mathbf{x}_{c,n+j,m+i}$$

where \mathbf{u} and \mathbf{b} are shared parameters to learn.

- D convolutions can be applied in the same way to produce a $D \times (H - h + 1) \times (W - w + 1)$ feature map, where D is the depth.

Convolution as a matrix multiplication

As a guiding example, let us consider the convolution of single-channel tensors $\mathbf{x} \in \mathbb{R}^{4 \times 4}$ and $\mathbf{u} \in \mathbb{R}^{3 \times 3}$:

$$\mathbf{u} \star \mathbf{x} = \begin{pmatrix} 1 & 4 & 1 \\ 1 & 4 & 3 \\ 3 & 3 & 1 \end{pmatrix} \star \begin{pmatrix} 4 & 5 & 8 & 7 \\ 1 & 8 & 8 & 8 \\ 3 & 6 & 6 & 4 \\ 6 & 5 & 7 & 8 \end{pmatrix} = \begin{pmatrix} 122 & 148 \\ 126 & 134 \end{pmatrix}$$

The convolution operation can be equivalently re-expressed as a single matrix multiplication:

- the convolutional kernel \mathbf{u} is rearranged as a **sparse Toeplitz circulant matrix**, called the convolution matrix:

$$\mathbf{U} = \begin{pmatrix} 1 & 4 & 1 & 0 & 1 & 4 & 3 & 0 & 3 & 3 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 & 1 & 4 & 3 & 0 & 3 & 3 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 4 & 1 & 0 & 1 & 4 & 3 & 0 & 3 & 3 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 4 & 1 & 0 & 1 & 4 & 3 & 0 & 3 & 3 & 1 \end{pmatrix}$$

- the input \mathbf{x} is flattened row by row, from top to bottom:

$$v(\mathbf{x}) = (4 \quad 5 \quad 8 \quad 7 \quad 1 \quad 8 \quad 8 \quad 8 \quad 3 \quad 6 \quad 6 \quad 4 \quad 6 \quad 5 \quad 7 \quad 8)^T$$

Then,

$$\mathbf{U}v(\mathbf{x}) = (122 \quad 148 \quad 126 \quad 134)^T$$

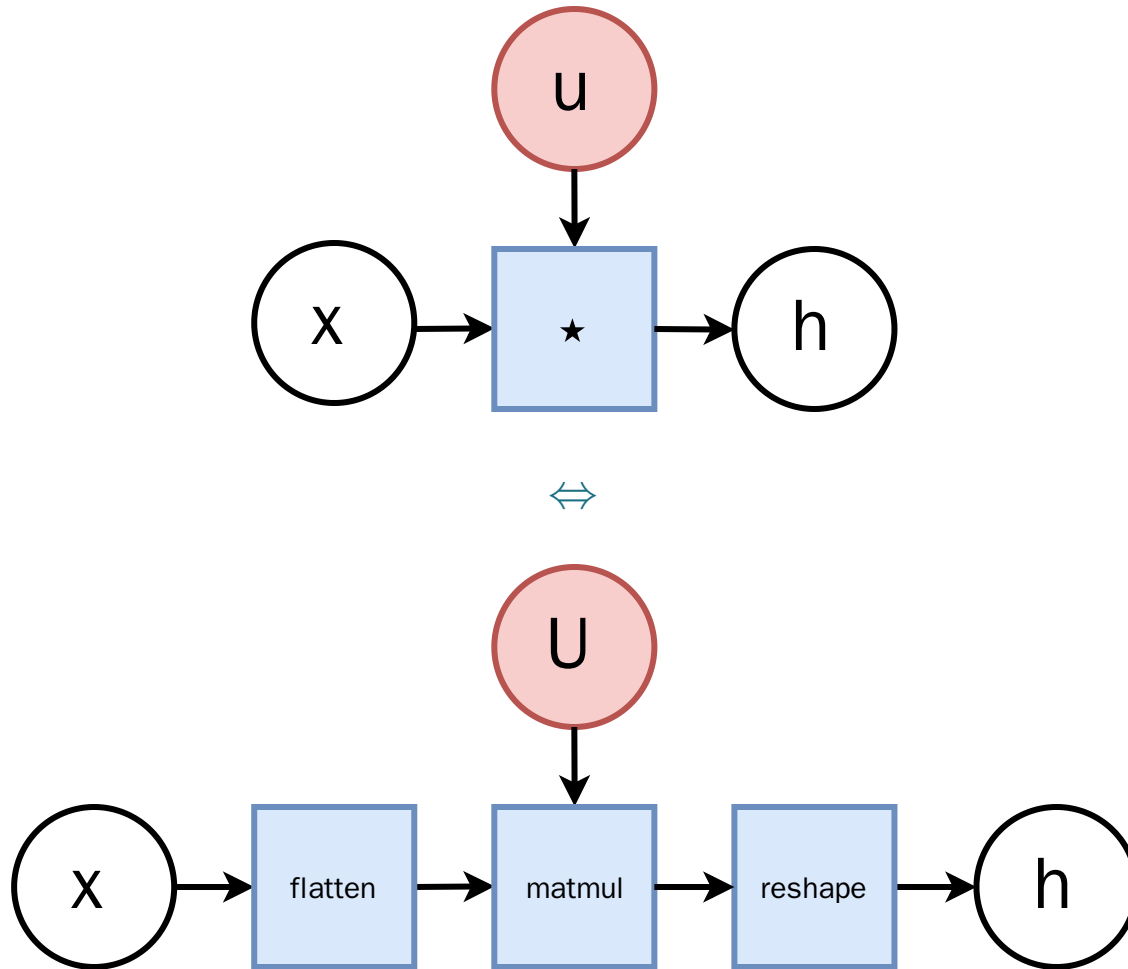
which we can reshape to a 2×2 matrix to obtain $\mathbf{u} \star \mathbf{x}$.

The same procedure generalizes to $\mathbf{x} \in \mathbb{R}^{H \times W}$ and convolutional kernel $\mathbf{u} \in \mathbb{R}^{h \times w}$, such that:

- the convolutional kernel is rearranged as a sparse Toeplitz circulant matrix \mathbf{U} of shape $(H - h + 1)(W - w + 1) \times HW$ where
 - each row i identifies an element of the output feature map,
 - each column j identifies an element of the input feature map,
 - the value $\mathbf{U}_{i,j}$ corresponds to the kernel value the element j is multiplied with in output i ;
- the input \mathbf{x} is flattened into a column vector $v(\mathbf{x})$ of shape $HW \times 1$;
- the output feature map $\mathbf{u} \star \mathbf{x}$ is obtained by reshaping the $(H - h + 1)(W - w + 1) \times 1$ column vector $\mathbf{U}v(\mathbf{x})$ as a $(H - h + 1) \times (W - w + 1)$ matrix.

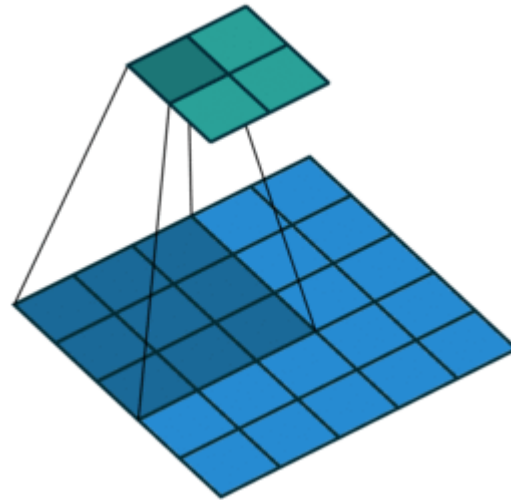
Therefore, a convolutional layer is a special case of a fully connected layer:

$$\mathbf{h} = \mathbf{u} \star \mathbf{x} \Leftrightarrow v(\mathbf{h}) = \mathbf{U}v(\mathbf{x}) \Leftrightarrow v(\mathbf{h}) = \mathbf{W}^T v(\mathbf{x})$$



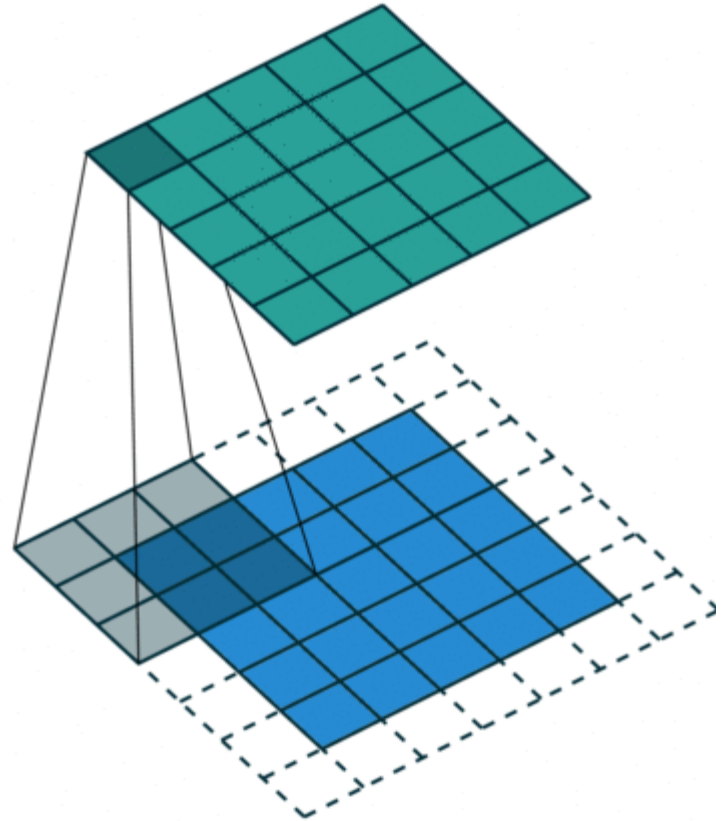
Strides

- The **stride** specifies the size of the step for the convolution operator.
- This parameter reduces the size of the output map.



Padding

- **Padding** specifies whether the input volume is padded artificially around its border.
- This parameter is useful to keep spatial dimensions constant across filters.
- Zero-padding is the default mode.



Equivariance

A function f is **equivariant** to g if $f(g(\mathbf{x})) = g(f(\mathbf{x}))$.

- Parameter sharing used in a convolutional layer causes the layer to be equivariant to translation.
- That is, if g is any function that translates the input, the convolution function is equivariant to g .



If an object moves in the input image, its representation will move the same amount in the output.

- Equivariance is useful when we know some local function is useful everywhere (e.g., edge detectors).
- Convolution is not equivariant to other operations such as change in scale or rotation.

Pooling

When the input volume is large, **pooling layers** can be used to reduce the input dimension while preserving its global structure, in a way similar to a down-scaling operation.

Consider a pooling area of size $h \times w$ and a 3D input tensor $\mathbf{x} \in \mathbb{R}^{C \times (rh) \times (sw)}$.

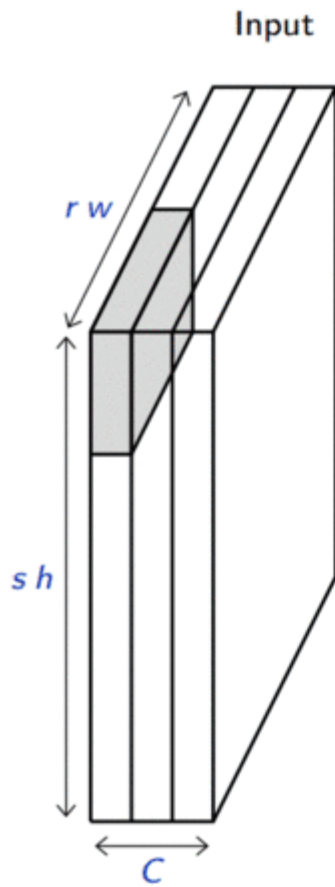
- Max-pooling produces a tensor $\mathbf{o} \in \mathbb{R}^{C \times r \times s}$ such that

$$\mathbf{o}_{c,j,i} = \max_{n < h, m < w} \mathbf{x}_{c,rj+n,si+m}.$$

- Average pooling produces a tensor $\mathbf{o} \in \mathbb{R}^{C \times r \times s}$ such that

$$\mathbf{o}_{c,j,i} = \frac{1}{hw} \sum_{n=0}^{h-1} \sum_{m=0}^{w-1} \mathbf{x}_{c,rj+n,si+m}.$$

Pooling is very similar in its formulation to convolution.



Invariance

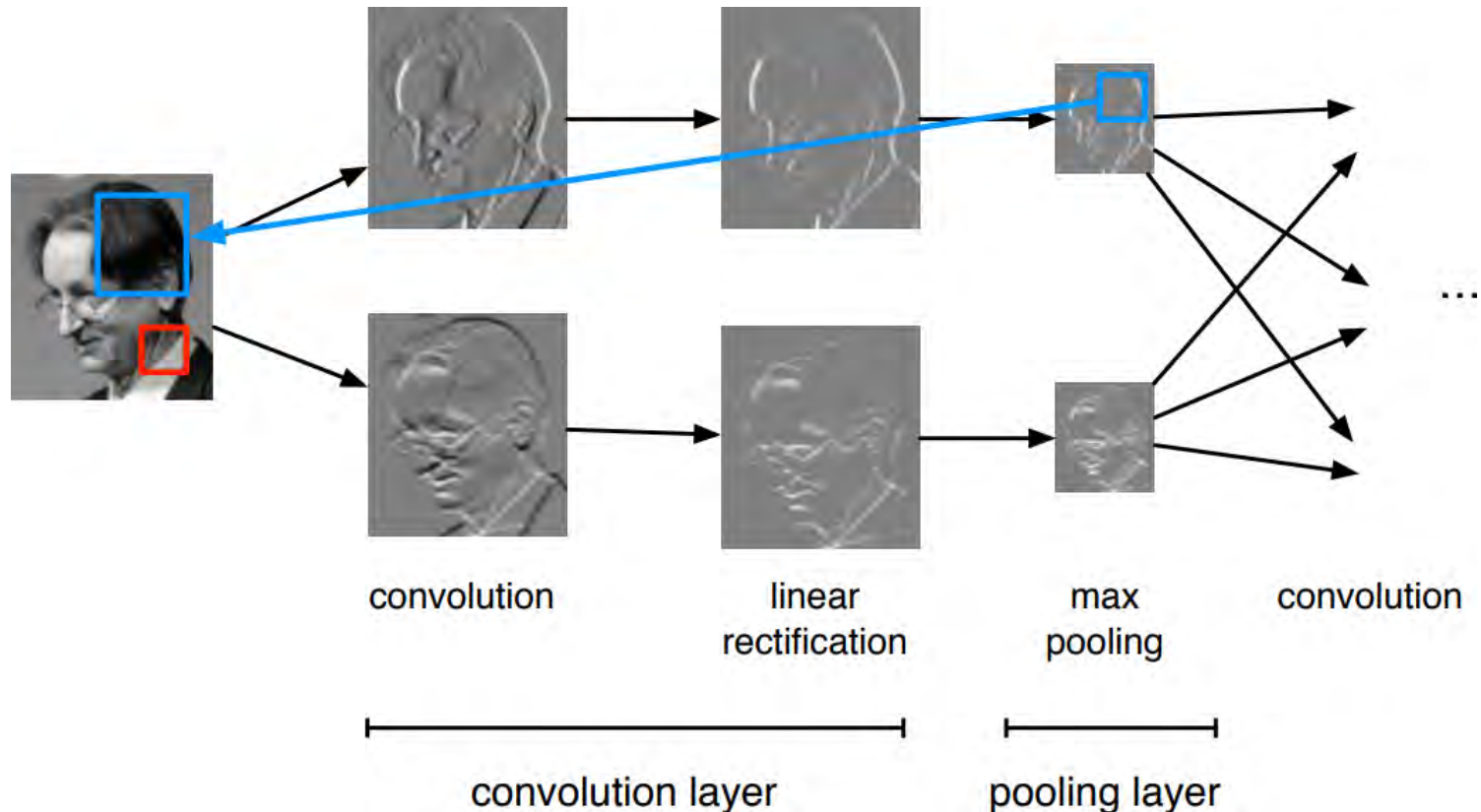
A function f is **invariant** to g if $f(g(\mathbf{x})) = f(\mathbf{x})$.

- Pooling layers can be used for building inner activations that are (slightly) invariant to small translations of the input.
- Invariance to local translation is helpful if we care more about the presence of a pattern rather than its exact position.

Architectures

Layer patterns

A **convolutional network** can often be defined as a composition of convolutional layers (**CONV**), pooling layers (**POOL**), linear rectifiers (**RELU**) and fully connected layers (**FC**).



The most common convolutional network architecture follows the pattern:

INPUT \rightarrow **[[CONV \rightarrow RELU]* N \rightarrow POOL?]* M \rightarrow [FC \rightarrow RELU]* K \rightarrow **FC****

where:

- * indicates repetition;
- **POOL?** indicates an optional pooling layer;
- $N \geq 0$ (and usually $N \leq 3$), $M \geq 0$, $K \geq 0$ (and usually $K < 3$);
- the last fully connected layer holds the output (e.g., the class scores).

Architectures

Some common architectures for convolutional networks following this pattern include:

- **INPUT** \rightarrow **FC**, which implements a linear classifier ($N = M = K = 0$).
- **INPUT** \rightarrow [**FC** \rightarrow **RELU**]* K \rightarrow **FC**, which implements a K -layer MLP.
- **INPUT** \rightarrow **CONV** \rightarrow **RELU** \rightarrow **FC**.
- **INPUT** \rightarrow [**CONV** \rightarrow **RELU** \rightarrow **POOL**]*2 \rightarrow **FC** \rightarrow **RELU** \rightarrow **FC**.
- **INPUT** \rightarrow [[**CONV** \rightarrow **RELU**]*2 \rightarrow **POOL**]*3 \rightarrow [**FC** \rightarrow **RELU**]*2 \rightarrow **FC**.



LeNet-5 (LeCun et al, 1998)

- First convolutional network to use backpropagation.
- Applied to character recognition.

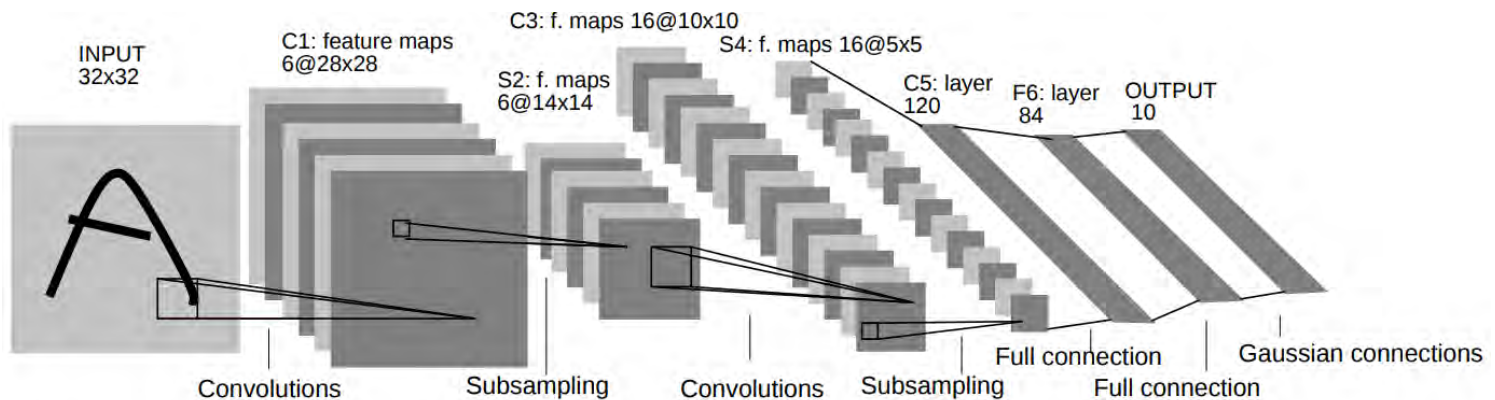


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 6, 28, 28]	156
ReLU-2	[-1, 6, 28, 28]	0
MaxPool2d-3	[-1, 6, 14, 14]	0
Conv2d-4	[-1, 16, 10, 10]	2,416
ReLU-5	[-1, 16, 10, 10]	0
MaxPool2d-6	[-1, 16, 5, 5]	0
Conv2d-7	[-1, 120, 1, 1]	48,120
ReLU-8	[-1, 120, 1, 1]	0
Linear-9	[-1, 84]	10,164
ReLU-10	[-1, 84]	0
Linear-11	[-1, 10]	850
LogSoftmax-12	[-1, 10]	0

Total params: 61,706
 Trainable params: 61,706
 Non-trainable params: 0

Input size (MB): 0.00
 Forward/backward pass size (MB): 0.11
 Params size (MB): 0.24
 Estimated Total Size (MB): 0.35

AlexNet (Krizhevsky et al, 2012)

- 16.4% top-5 error on ILSVRC'12, outperformed all by 10%.
- Implementation on two GPUs, because of memory constraints.

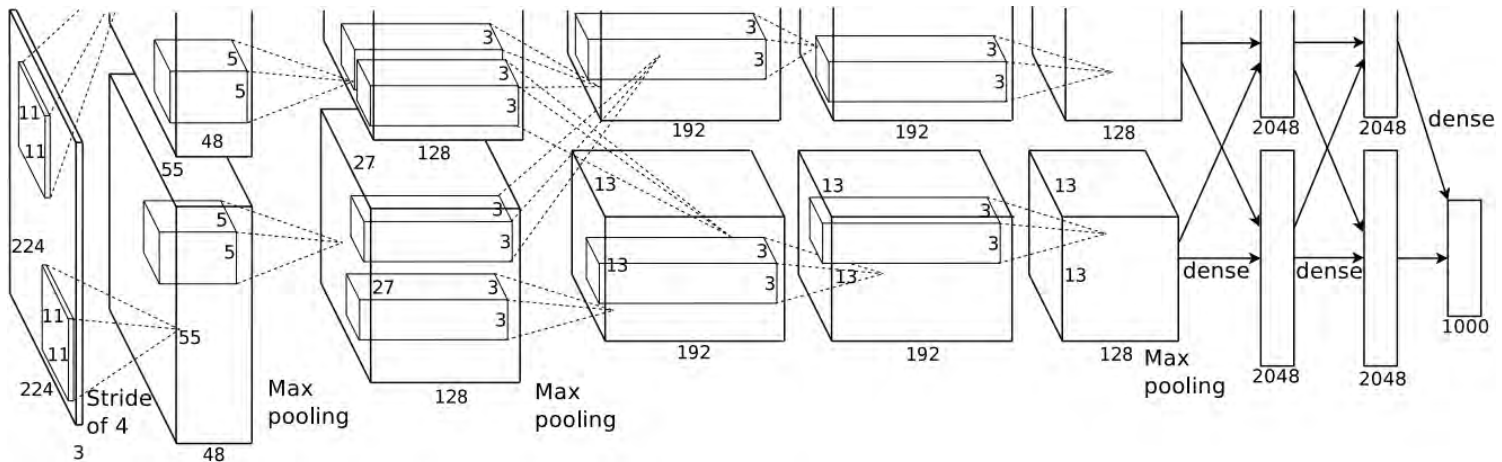


Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 55, 55]	23,296
ReLU-2	[-1, 64, 55, 55]	0
MaxPool2d-3	[-1, 64, 27, 27]	0
Conv2d-4	[-1, 192, 27, 27]	307,392
ReLU-5	[-1, 192, 27, 27]	0
MaxPool2d-6	[-1, 192, 13, 13]	0
Conv2d-7	[-1, 384, 13, 13]	663,936
ReLU-8	[-1, 384, 13, 13]	0
Conv2d-9	[-1, 256, 13, 13]	884,992
ReLU-10	[-1, 256, 13, 13]	0
Conv2d-11	[-1, 256, 13, 13]	590,080
ReLU-12	[-1, 256, 13, 13]	0
MaxPool2d-13	[-1, 256, 6, 6]	0
Dropout-14	[-1, 9216]	0
Linear-15	[-1, 4096]	37,752,832
ReLU-16	[-1, 4096]	0
Dropout-17	[-1, 4096]	0
Linear-18	[-1, 4096]	16,781,312
ReLU-19	[-1, 4096]	0
Linear-20	[-1, 1000]	4,097,000

Total params: 61,100,840

Trainable params: 61,100,840

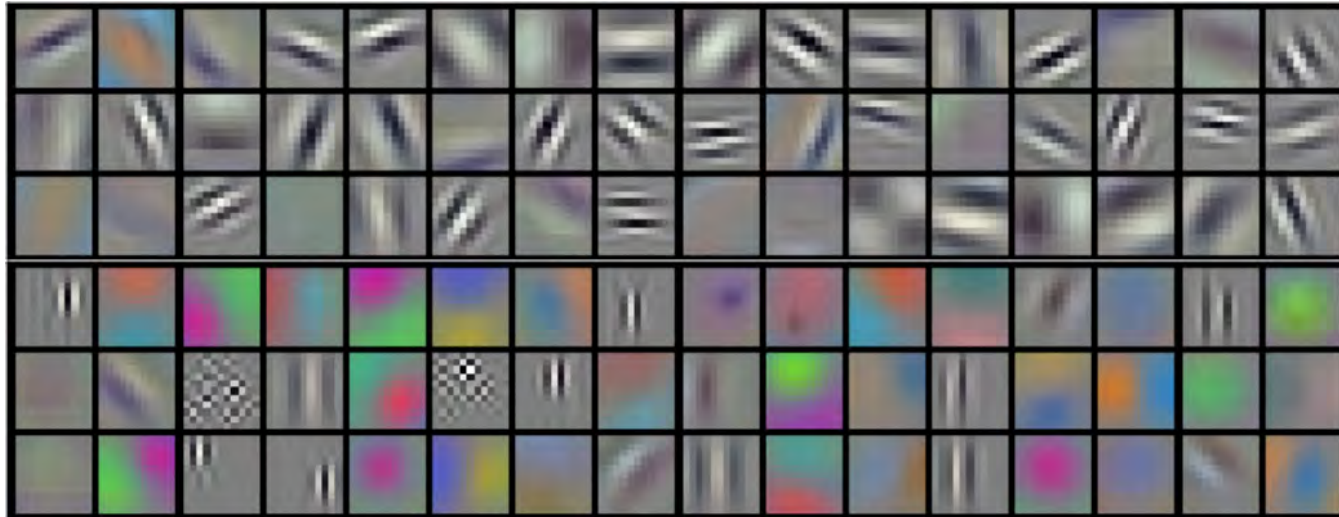
Non-trainable params: 0

Input size (MB): 0.57

Forward/backward pass size (MB): 8.31

Params size (MB): 233.08

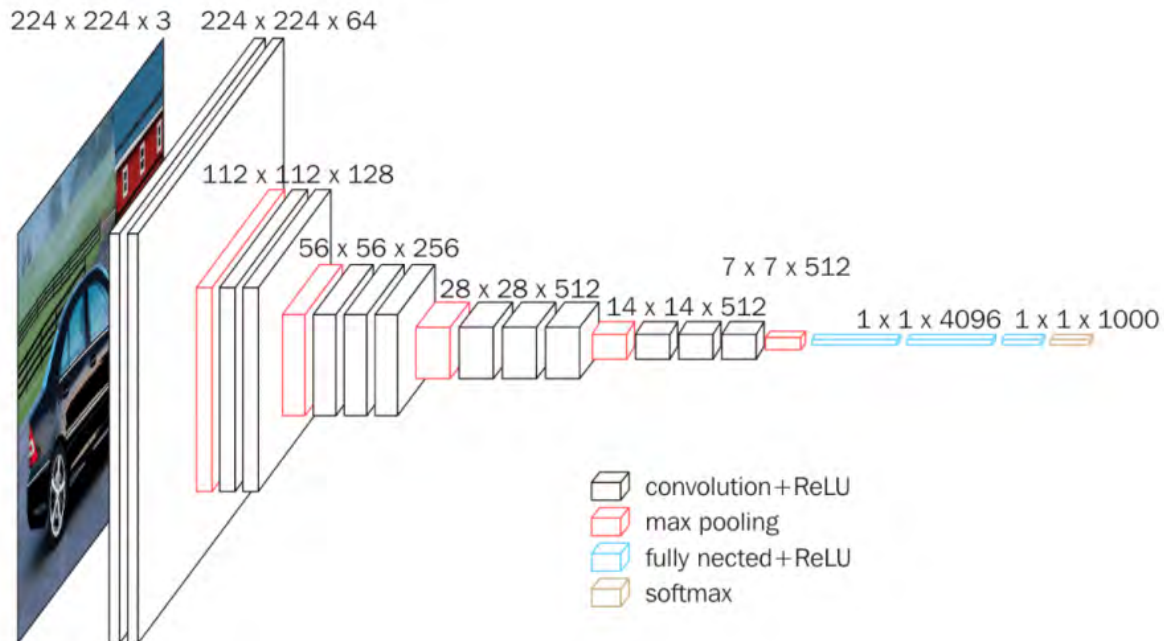
Estimated Total Size (MB): 241.96



- 96 $11 \times 11 \times 3$ kernels learned by the first convolutional layer.
- Top 48 kernels were learned on GPU1, while the bottom 48 kernels were learned on GPU 2.

VGG (Simonyan and Zisserman, 2014)

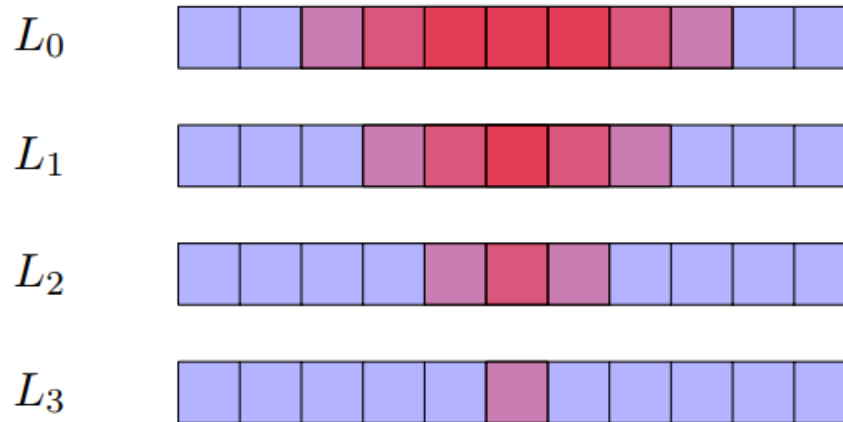
- 7.3% top-5 error on ILSVRC'14.
- Depth increased up to 19 layers, kernel sizes reduced to 3.



Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 224, 224]	1,792
ReLU-2	[-1, 64, 224, 224]	0
Conv2d-3	[-1, 64, 224, 224]	36,928
ReLU-4	[-1, 64, 224, 224]	0
MaxPool2d-5	[-1, 64, 112, 112]	0
Conv2d-6	[-1, 128, 112, 112]	73,856
ReLU-7	[-1, 128, 112, 112]	0
Conv2d-8	[-1, 128, 112, 112]	147,584
ReLU-9	[-1, 128, 112, 112]	0
MaxPool2d-10	[-1, 128, 56, 56]	0
Conv2d-11	[-1, 256, 56, 56]	295,168
ReLU-12	[-1, 256, 56, 56]	0
Conv2d-13	[-1, 256, 56, 56]	590,080
ReLU-14	[-1, 256, 56, 56]	0
Conv2d-15	[-1, 256, 56, 56]	590,080
ReLU-16	[-1, 256, 56, 56]	0
MaxPool2d-17	[-1, 256, 28, 28]	0
Conv2d-18	[-1, 512, 28, 28]	1,180,160
ReLU-19	[-1, 512, 28, 28]	0
Conv2d-20	[-1, 512, 28, 28]	2,359,808
ReLU-21	[-1, 512, 28, 28]	0
Conv2d-22	[-1, 512, 28, 28]	2,359,808
ReLU-23	[-1, 512, 28, 28]	0
MaxPool2d-24	[-1, 512, 14, 14]	0
Conv2d-25	[-1, 512, 14, 14]	2,359,808
ReLU-26	[-1, 512, 14, 14]	0
Conv2d-27	[-1, 512, 14, 14]	2,359,808
ReLU-28	[-1, 512, 14, 14]	0
Conv2d-29	[-1, 512, 14, 14]	2,359,808
ReLU-30	[-1, 512, 14, 14]	0
MaxPool2d-31	[-1, 512, 7, 7]	0
Linear-32	[-1, 4096]	102,764,544
ReLU-33	[-1, 4096]	0
Dropout-34	[-1, 4096]	0
Linear-35	[-1, 4096]	16,781,312
ReLU-36	[-1, 4096]	0
Dropout-37	[-1, 4096]	0
Linear-38	[-1, 1000]	4,097,000

Total params: 138,357,544
 Trainable params: 138,357,544
 Non-trainable params: 0

Input size (MB): 0.57
 Forward/backward pass size (MB): 218.59
 Params size (MB): 527.79
 Estimated Total Size (MB): 746.96



The **effective receptive field** is the part of the visual input that affects a given unit indirectly through previous layers.

- It grows linearly with depth.
- A stack of three 3×3 kernels of stride 1 has the same effective receptive field as a single 7×7 kernel, but fewer parameters.

ResNet (He et al, 2015)

- Even deeper models (34, 50, 101 and 152 layers)
- Skip connections.
- Resnet-50 vs. VGG:
 - 5.25% top-5 error vs. 7.1%
 - 25M vs. 138M parameters
 - 3.8B Flops vs. 15.3B Flops
 - Fully convolutional until the last layer

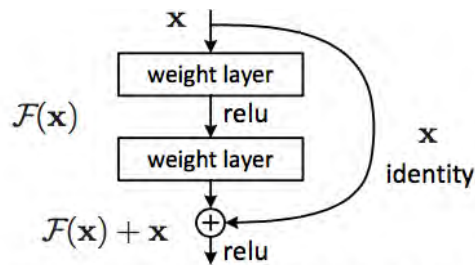
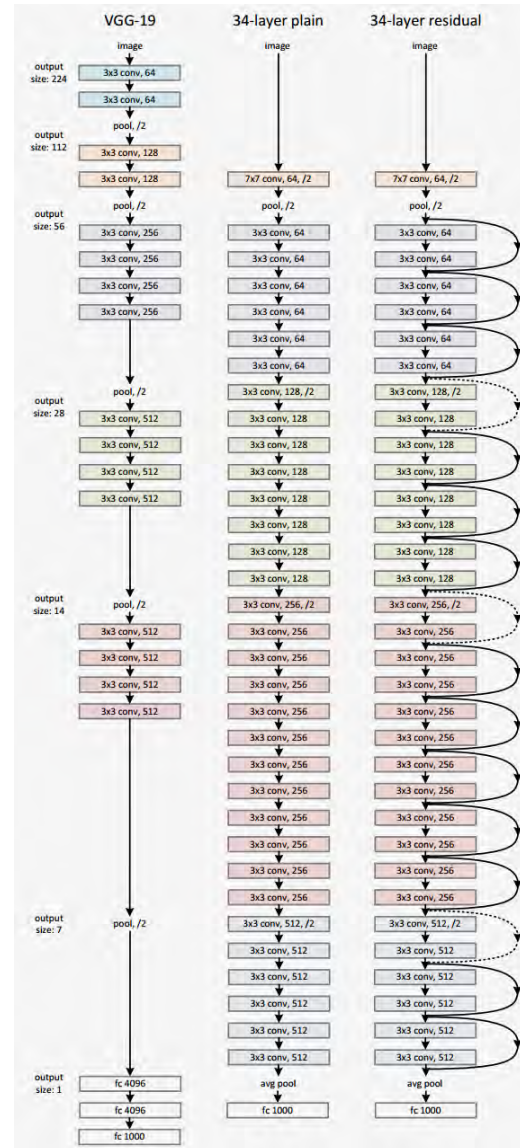


Figure 2. Residual learning: a building block.

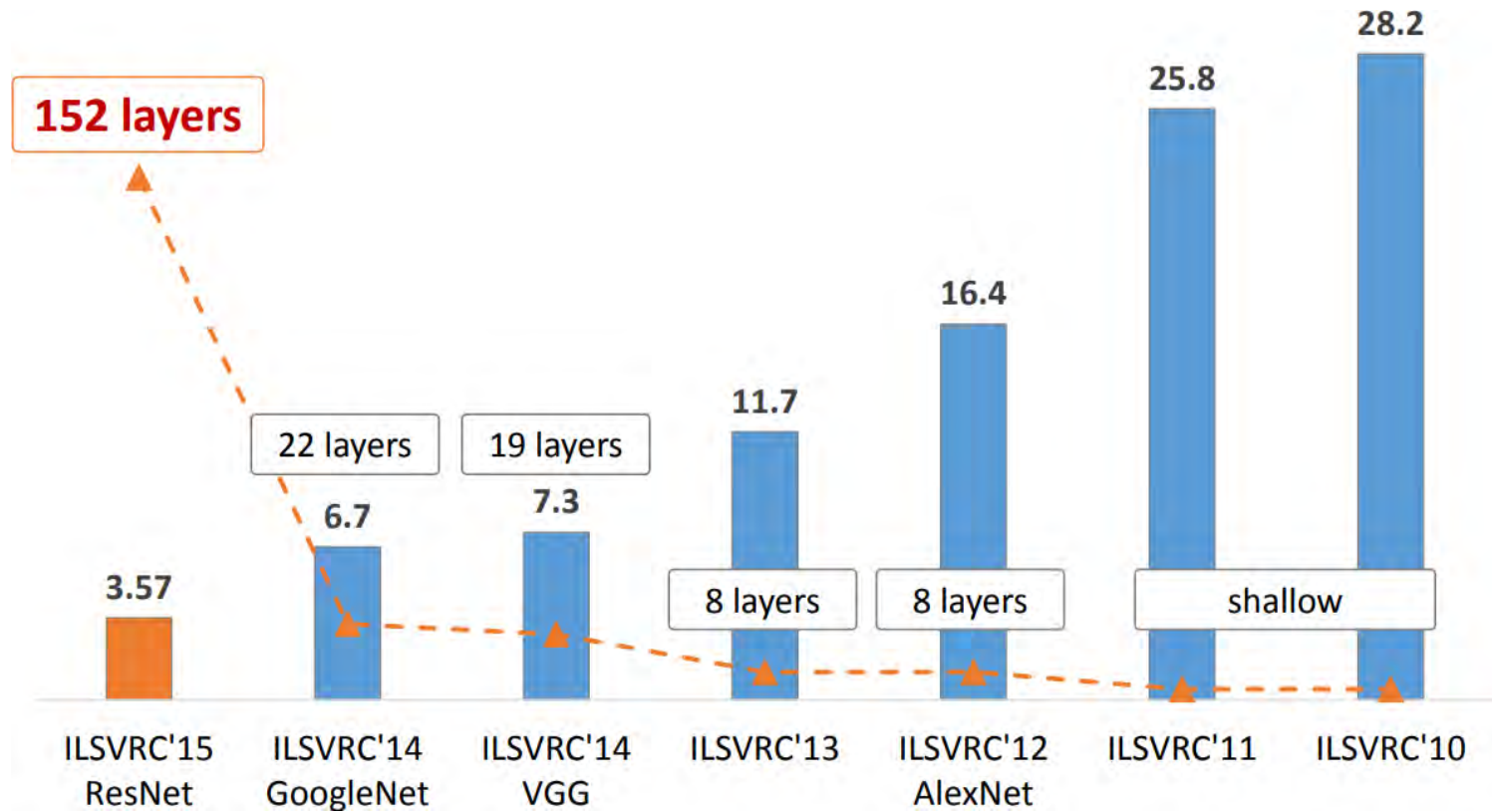


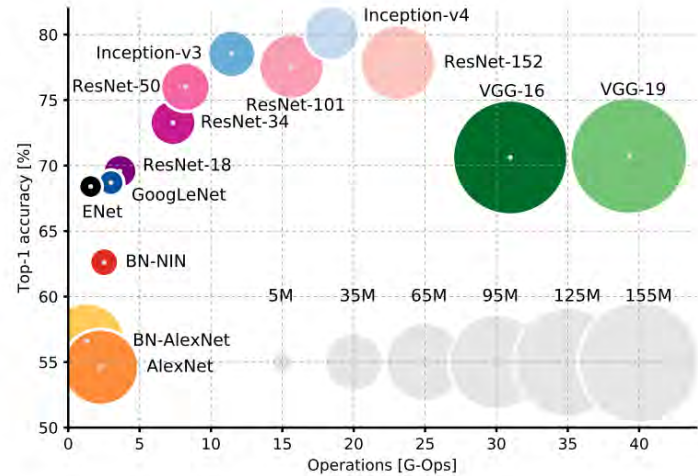
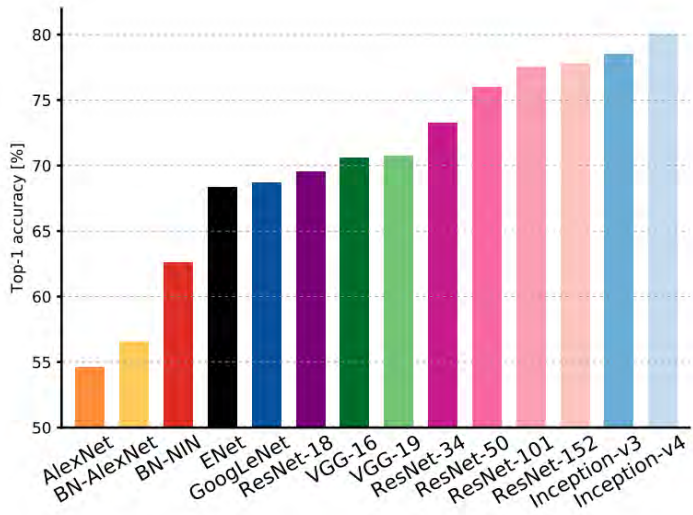
Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 112, 112]	9,408
BatchNorm2d-2	[-1, 64, 112, 112]	128
ReLU-3	[-1, 64, 112, 112]	0
MaxPool2d-4	[-1, 64, 56, 56]	0
Conv2d-5	[-1, 64, 56, 56]	4,096
BatchNorm2d-6	[-1, 64, 56, 56]	128
ReLU-7	[-1, 64, 56, 56]	0
Conv2d-8	[-1, 64, 56, 56]	36,864
BatchNorm2d-9	[-1, 64, 56, 56]	128
ReLU-10	[-1, 64, 56, 56]	0
Conv2d-11	[-1, 256, 56, 56]	16,384
BatchNorm2d-12	[-1, 256, 56, 56]	512
Conv2d-13	[-1, 256, 56, 56]	16,384
BatchNorm2d-14	[-1, 256, 56, 56]	512
ReLU-15	[-1, 256, 56, 56]	0
Bottleneck-16	[-1, 256, 56, 56]	0
Conv2d-17	[-1, 64, 56, 56]	16,384
BatchNorm2d-18	[-1, 64, 56, 56]	128
ReLU-19	[-1, 64, 56, 56]	0
Conv2d-20	[-1, 64, 56, 56]	36,864
BatchNorm2d-21	[-1, 64, 56, 56]	128
ReLU-22	[-1, 64, 56, 56]	0
Conv2d-23	[-1, 256, 56, 56]	16,384
BatchNorm2d-24	[-1, 256, 56, 56]	512
ReLU-25	[-1, 256, 56, 56]	0
Bottleneck-26	[-1, 256, 56, 56]	0
Conv2d-27	[-1, 64, 56, 56]	16,384
BatchNorm2d-28	[-1, 64, 56, 56]	128
ReLU-29	[-1, 64, 56, 56]	0
Conv2d-30	[-1, 64, 56, 56]	36,864
BatchNorm2d-31	[-1, 64, 56, 56]	128
ReLU-32	[-1, 64, 56, 56]	0
Conv2d-33	[-1, 256, 56, 56]	16,384
BatchNorm2d-34	[-1, 256, 56, 56]	512
ReLU-35	[-1, 256, 56, 56]	0
Bottleneck-36	[-1, 256, 56, 56]	0
Conv2d-37	[-1, 128, 56, 56]	32,768
BatchNorm2d-38	[-1, 128, 56, 56]	256
ReLU-39	[-1, 128, 56, 56]	0
Conv2d-40	[-1, 128, 28, 28]	147,456
BatchNorm2d-41	[-1, 128, 28, 28]	256
ReLU-42	[-1, 128, 28, 28]	0
Conv2d-43	[-1, 512, 28, 28]	65,536
BatchNorm2d-44	[-1, 512, 28, 28]	1,024
Conv2d-45	[-1, 512, 28, 28]	131,072
BatchNorm2d-46	[-1, 512, 28, 28]	1,024
ReLU-47	[-1, 512, 28, 28]	0
Bottleneck-48	[-1, 512, 28, 28]	0
Conv2d-49	[-1, 128, 28, 28]	65,536
BatchNorm2d-50	[-1, 128, 28, 28]	256
ReLU-51	[-1, 128, 28, 28]	0
Conv2d-52	[-1, 128, 28, 28]	147,456
BatchNorm2d-53	[-1, 128, 28, 28]	256

...
Bottleneck-130	[-1, 1024, 14, 14]	0
Conv2d-131	[-1, 256, 14, 14]	262,144
BatchNorm2d-132	[-1, 256, 14, 14]	512
ReLU-133	[-1, 256, 14, 14]	0
Conv2d-134	[-1, 256, 14, 14]	589,824
BatchNorm2d-135	[-1, 256, 14, 14]	512
ReLU-136	[-1, 256, 14, 14]	0
Conv2d-137	[-1, 1024, 14, 14]	262,144
BatchNorm2d-138	[-1, 1024, 14, 14]	2,048
ReLU-139	[-1, 1024, 14, 14]	0
Bottleneck-140	[-1, 1024, 14, 14]	0
Conv2d-141	[-1, 512, 14, 14]	524,288
BatchNorm2d-142	[-1, 512, 14, 14]	1,024
ReLU-143	[-1, 512, 14, 14]	0
Conv2d-144	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-145	[-1, 512, 7, 7]	1,024
ReLU-146	[-1, 512, 7, 7]	0
Conv2d-147	[-1, 2048, 7, 7]	1,048,576
BatchNorm2d-148	[-1, 2048, 7, 7]	4,096
Conv2d-149	[-1, 2048, 7, 7]	2,097,152
BatchNorm2d-150	[-1, 2048, 7, 7]	4,096
ReLU-151	[-1, 2048, 7, 7]	0
Bottleneck-152	[-1, 2048, 7, 7]	0
Conv2d-153	[-1, 512, 7, 7]	1,048,576
BatchNorm2d-154	[-1, 512, 7, 7]	1,024
ReLU-155	[-1, 512, 7, 7]	0
Conv2d-156	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-157	[-1, 512, 7, 7]	1,024
ReLU-158	[-1, 512, 7, 7]	0
Conv2d-159	[-1, 2048, 7, 7]	1,048,576
BatchNorm2d-160	[-1, 2048, 7, 7]	4,096
ReLU-161	[-1, 2048, 7, 7]	0
Bottleneck-162	[-1, 2048, 7, 7]	0
Conv2d-163	[-1, 512, 7, 7]	1,048,576
BatchNorm2d-164	[-1, 512, 7, 7]	1,024
ReLU-165	[-1, 512, 7, 7]	0
Conv2d-166	[-1, 512, 7, 7]	2,359,296
BatchNorm2d-167	[-1, 512, 7, 7]	1,024
ReLU-168	[-1, 512, 7, 7]	0
Conv2d-169	[-1, 2048, 7, 7]	1,048,576
BatchNorm2d-170	[-1, 2048, 7, 7]	4,096
ReLU-171	[-1, 2048, 7, 7]	0
Bottleneck-172	[-1, 2048, 7, 7]	0
AvgPool2d-173	[-1, 2048, 1, 1]	0
Linear-174	[-1, 1000]	2,049,000
=====		
Total params:	25,557,032	
Trainable params:	25,557,032	
Non-trainable params:	0	

Input size (MB):	0.57	
Forward/backward pass size (MB):	286.56	
Params size (MB):	97.49	
Estimated Total Size (MB):	384.62	

Deeper is better





Finding the optimal neural network architecture remains an **active area of research.**

Pre-trained models

- Training a model on natural images, from scratch, takes **days or weeks**.
- Many models trained on ImageNet are publicly available for download. These models can be used as **feature extractors** or for smart **initialization**.

Transfer learning

- Take a pre-trained network, remove the last layer(s) and then treat the rest of the the network as a **fixed** feature extractor.
- Train a model from these features on a new task.
- Often better than handcrafted feature extraction for natural images, or better than training from data of the new task only.

Fine tuning

- Same as for transfer learning, but also **fine-tune** the weights of the pre-trained network by continuing backpropagation.
- All or only some of the layers can be tuned.

In the case of models pre-trained on ImageNet, this often works even when input images for the new task are not photographs of objects or animals, such as biomedical images, satellite images or paintings.

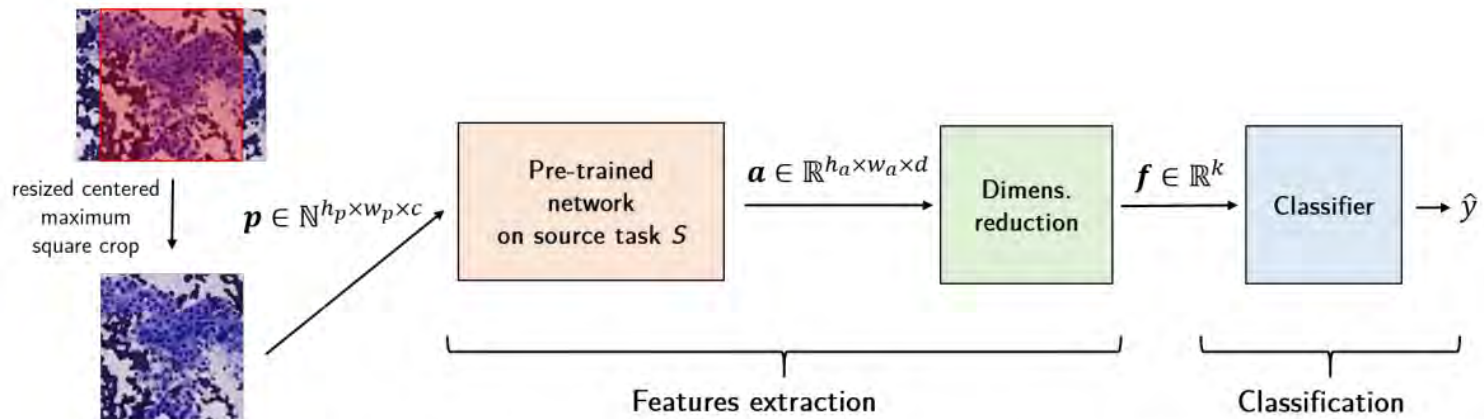


Figure 2. Feature extraction from pre-trained convolutional neural networks

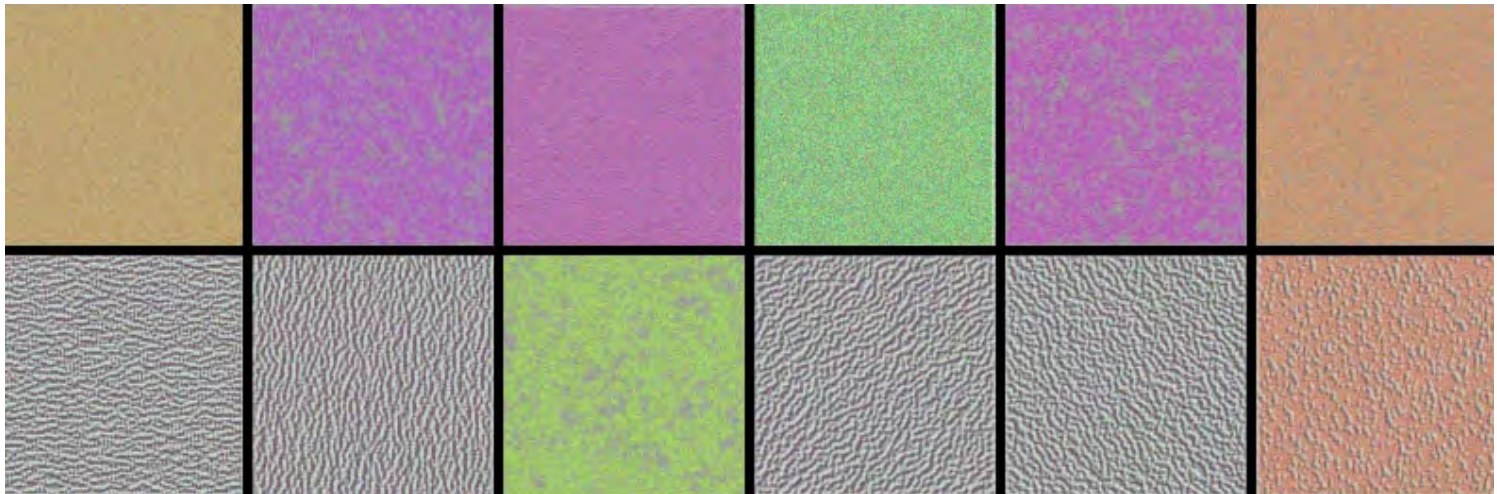
What is really happening?

Maximum response samples

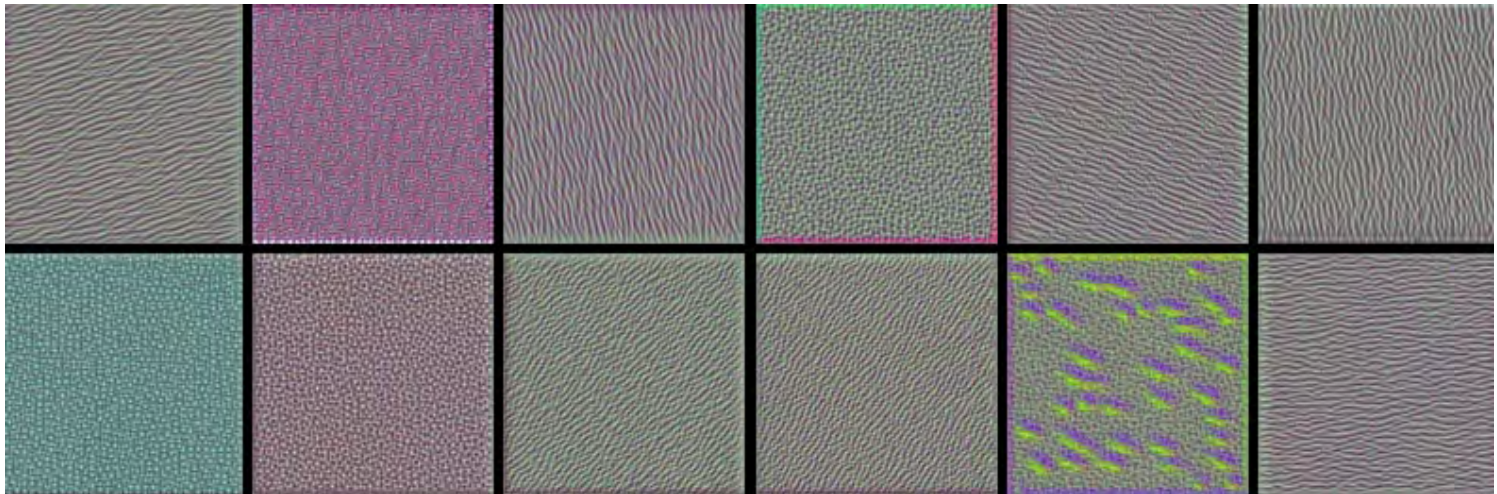
Convolutional networks can be inspected by looking for input images \mathbf{x} that maximize the activation $\mathbf{h}_{\ell,d}(\mathbf{x})$ of a chosen convolutional kernel \mathbf{u} at layer ℓ and index d in the layer filter bank.

Such images can be found by gradient ascent on the input space:

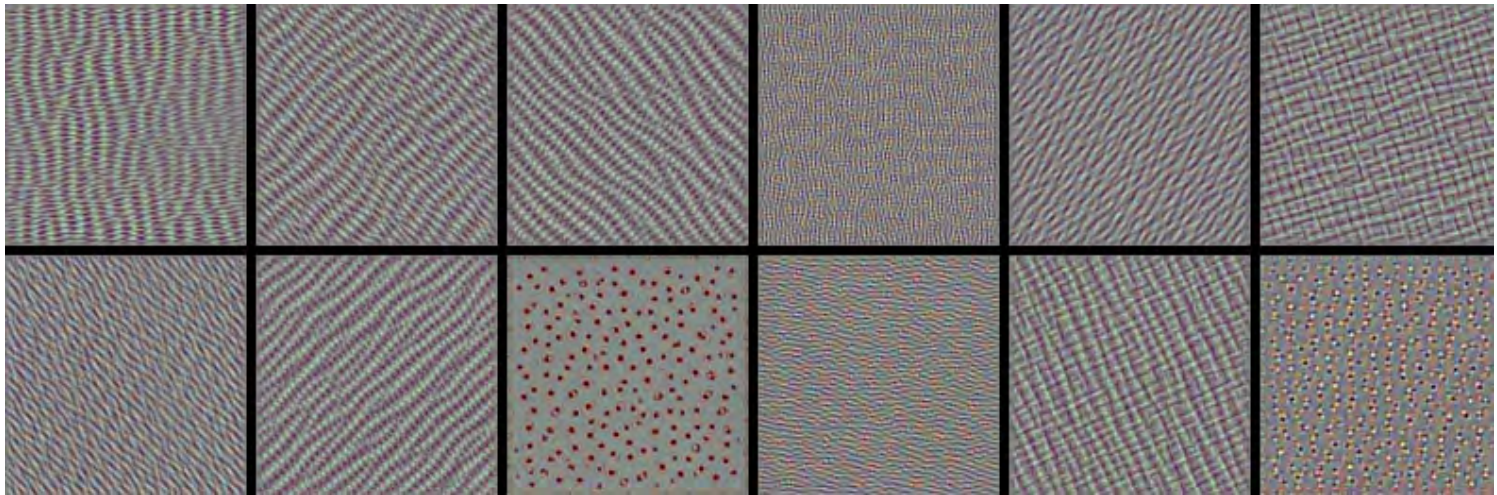
$$\begin{aligned}\mathcal{L}_{\ell,d}(\mathbf{x}) &= \|\mathbf{h}_{\ell,d}(\mathbf{x})\|_2 \\ \mathbf{x}_0 &\sim U[0, 1]^{C \times H \times W} \\ \mathbf{x}_{t+1} &= \mathbf{x}_t + \gamma \nabla_{\mathbf{x}} \mathcal{L}_{\ell,d}(\mathbf{x}_t)\end{aligned}$$



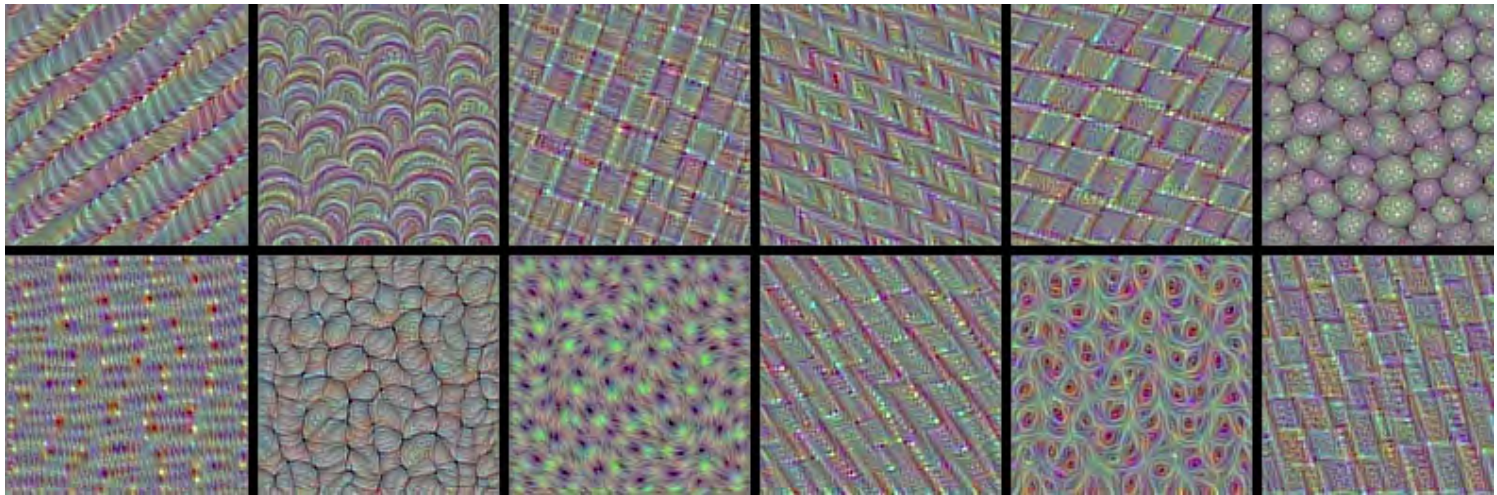
VGG-16, convolutional layer 1-1, a few of the 64 filters



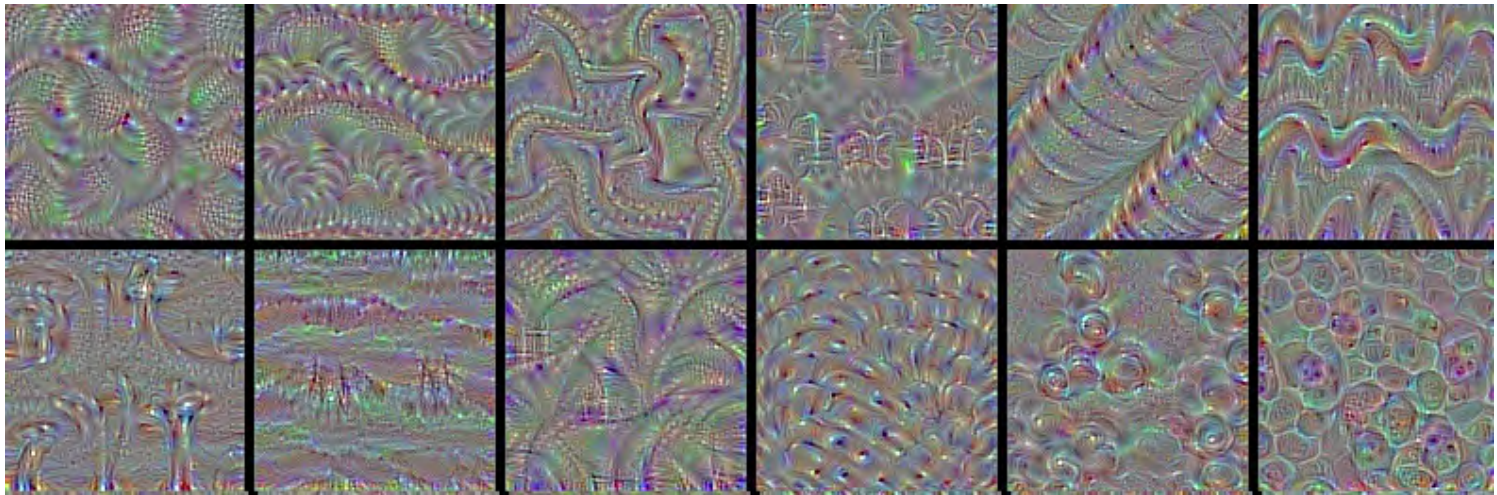
VGG-16, convolutional layer 2-1, a few of the 128 filters



VGG-16, convolutional layer 3-1, a few of the 256 filters



VGG-16, convolutional layer 4-1, a few of the 512 filters

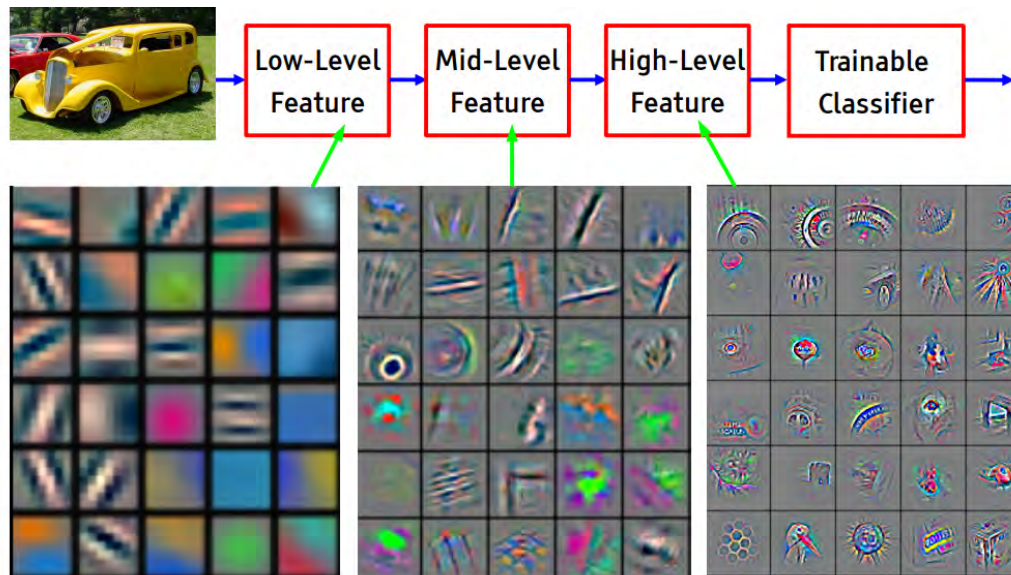


VGG-16, convolutional layer 5-1, a few of the 512 filters

Some observations:

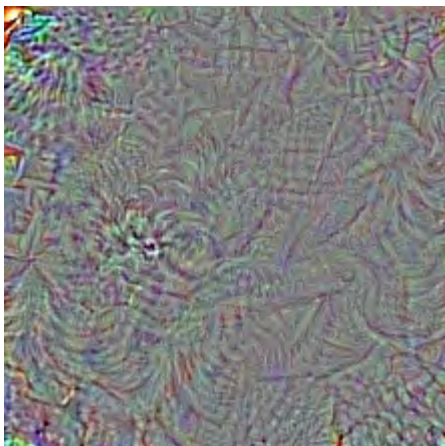
- The first layers appear to encode direction and color.
- The direction and color filters get combined into grid and spot textures.
- These textures gradually get combined into increasingly complex patterns.

In other words, the network appears to learn a hierarchical composition of patterns.



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

What if we build images that maximize the activation of a chosen class output?
The left image is predicted **with 99.9% confidence** as a magpie!



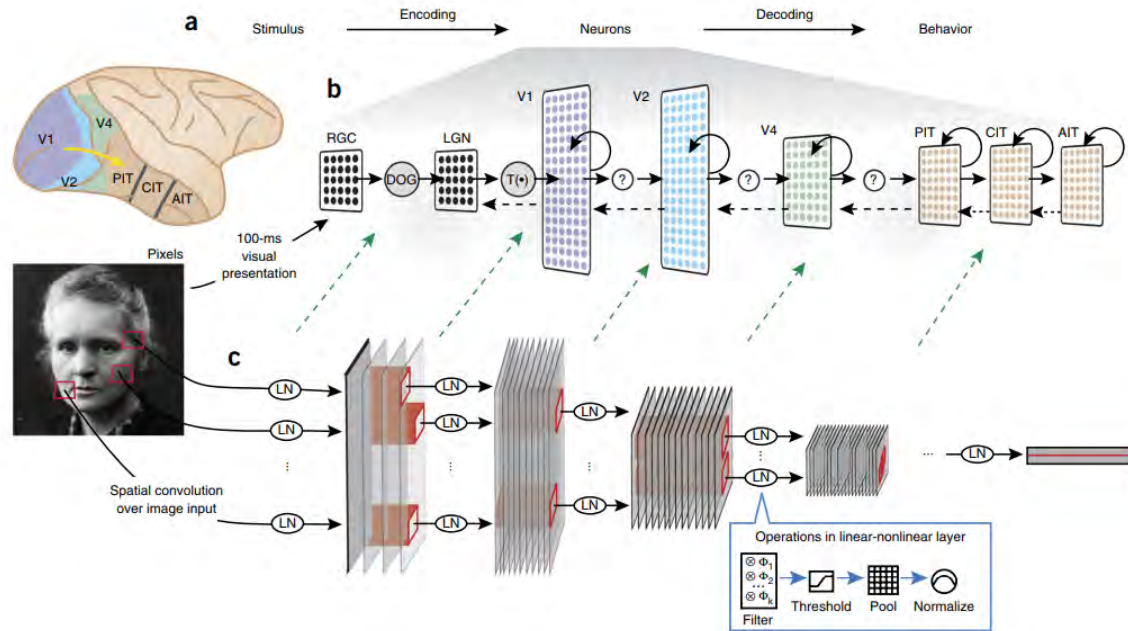


Journey on the Deep Dream



Deep Dream. Start from an image \mathbf{x}_t , offset by a random jitter, enhance some layer activation at multiple scales, zoom in, repeat on the produced image \mathbf{x}_{t+1} .

Biological plausibility



"Deep hierarchical neural networks are beginning to transform neuroscientists' ability to produce quantitatively accurate computational models of the sensory systems, especially in higher cortical areas where neural response properties had previously been enigmatic."

The end.

References

- Francois Fleuret, Deep Learning Course, [4.4. Convolutions](#), EPFL, 2018.
- Yannis Avrithis, Deep Learning for Vision, [Lecture 1: Introduction](#), University of Rennes 1, 2018.
- Yannis Avrithis, Deep Learning for Vision, [Lecture 7: Convolution and network architectures](#) , University of Rennes 1, 2018.
- Olivier Grisel and Charles Ollion, Deep Learning, [Lecture 4: Convolutional Neural Networks for Image Classification](#) , Université Paris-Saclay, 2018.

Deep Learning

Lecture 4: Training neural networks

Prof. Gilles Louppe
g.louppe@uliege.be

Today

How to **optimize parameters** efficiently?

- Optimizers
- Initialization
- Normalization

Optimizers

Gradient descent

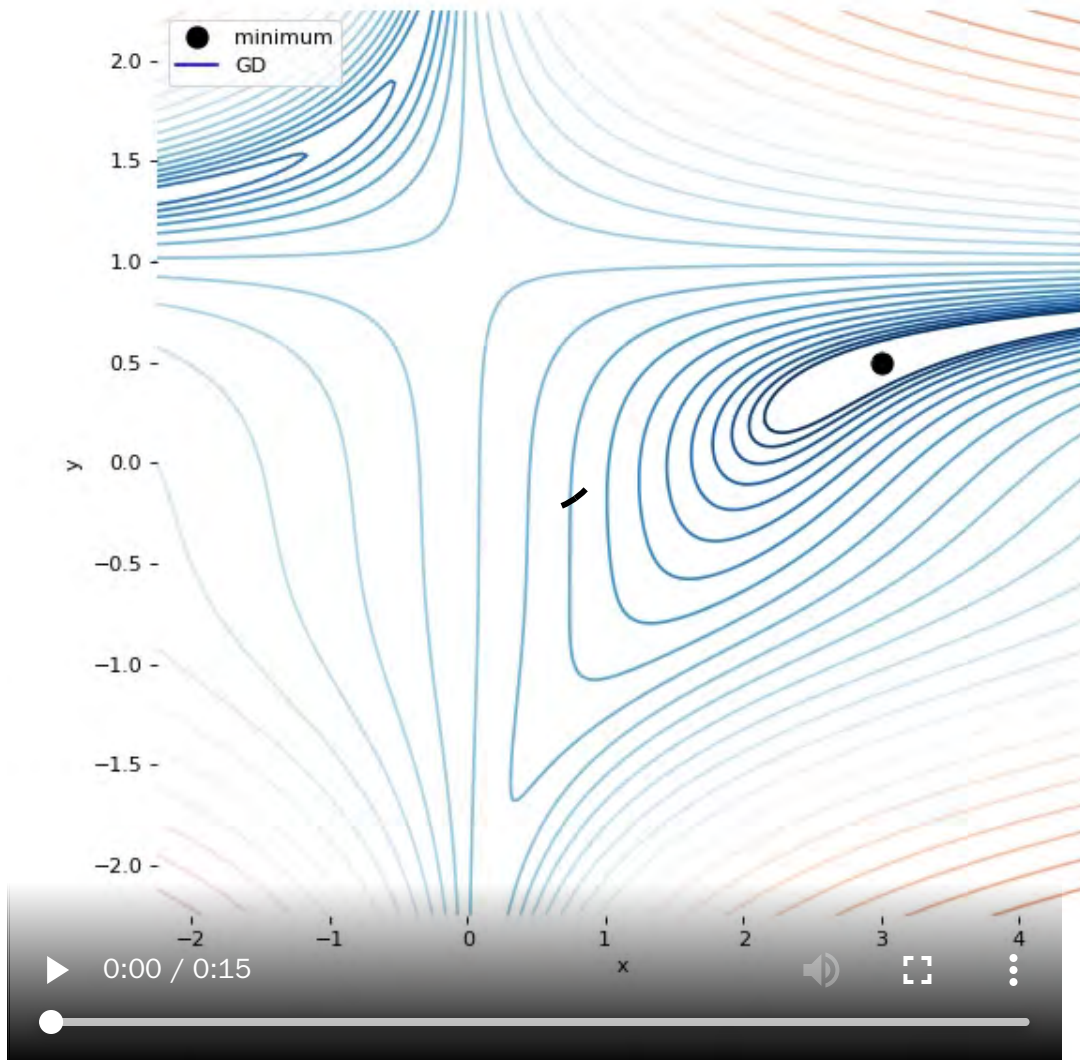
To minimize a loss $\mathcal{L}(\theta)$ of the form

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{n=1}^N \ell(y_n, f(\mathbf{x}_n; \theta)),$$

standard **batch gradient descent** (GD) consists in applying the update rule

$$g_t = \frac{1}{N} \sum_{n=1}^N \nabla_{\theta} \ell(y_n, f(\mathbf{x}_n; \theta_t))$$
$$\theta_{t+1} = \theta_t - \gamma g_t,$$

where γ is the learning rate.



While it makes sense in principle to compute the gradient exactly,

- it takes time to compute and becomes inefficient for large N ,
- it is an empirical estimation of an hidden quantity (the expected risk), and any partial sum is also an unbiased estimate, although of greater variance.

To illustrate how partial sums are good estimates, consider an ideal case where the training set is the same set of $M \ll N$ samples replicated K times. Then,

$$\begin{aligned}\mathcal{L}(\theta) &= \frac{1}{N} \sum_{i=1}^N \ell(y_i, f(\mathbf{x}_i; \theta)) \\ &= \frac{1}{N} \sum_{k=1}^K \sum_{m=1}^M \ell(y_m, f(\mathbf{x}_m; \theta)) \\ &= \frac{1}{N} K \sum_{m=1}^M \ell(y_m, f(\mathbf{x}_m; \theta)).\end{aligned}$$

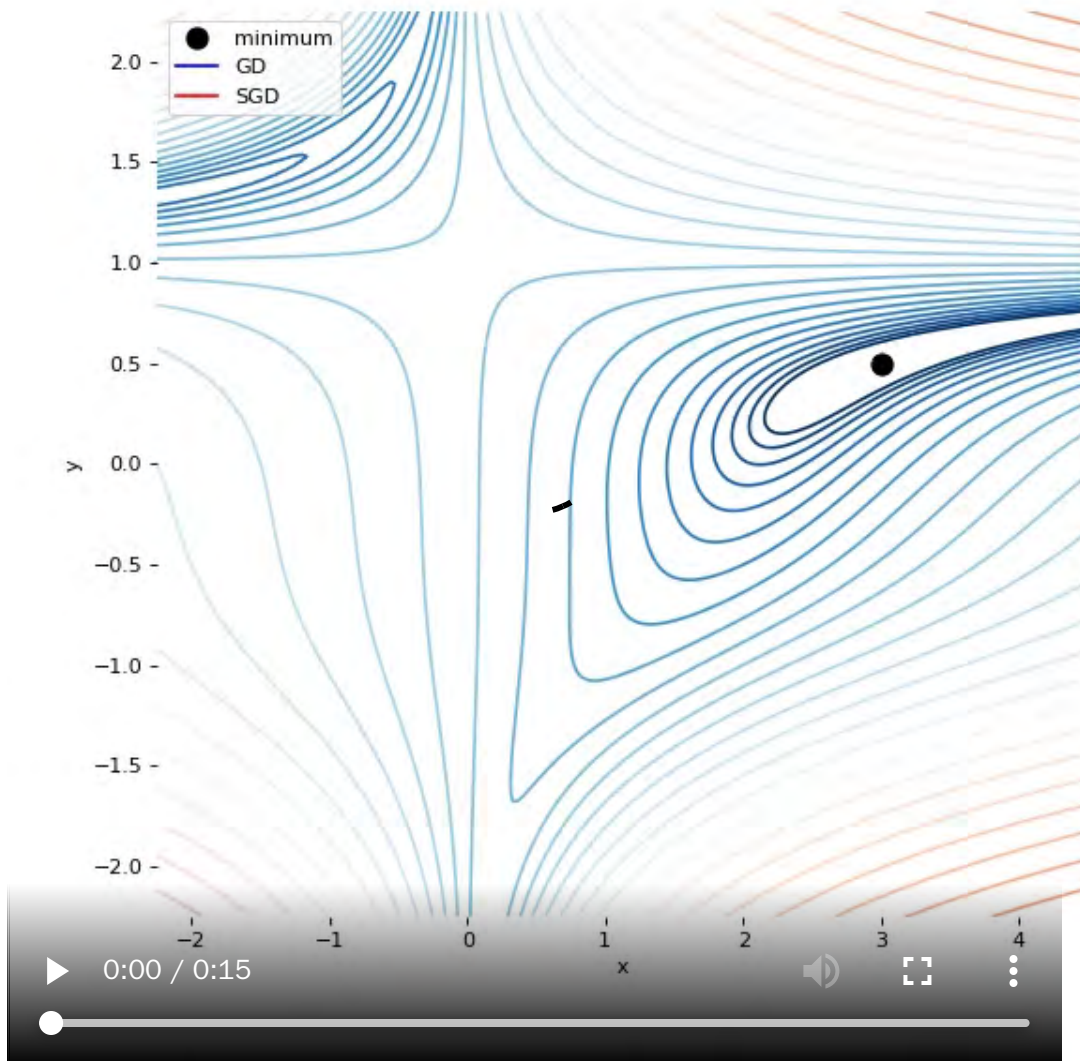
Then, instead of summing over all the samples and moving by γ , we can visit only $M = N/K$ samples and move by $K\gamma$, which would cut the computation by K .

Although this is an ideal case, there is redundancy in practice that results in similar behaviors.

Stochastic gradient descent

To reduce the computational complexity, **stochastic gradient descent** (SGD) consists in updating the parameters after every sample

$$g_t = \nabla_{\theta} \ell(y_{n(t)}, f(\mathbf{x}_{n(t)}; \theta_t))$$
$$\theta_{t+1} = \theta_t - \gamma g_t.$$



The stochastic behavior of SGD helps **evade local minima**.

While being computationally faster than batch gradient descent,

- gradient estimates used by SGD can be **very noisy**,
- SGD does not benefit from the speed-up of **batch-processing**.

Mini-batching

Instead, **mini-batch** SGD consists of visiting the samples in mini-batches and updating the parameters each time

$$g_t = \frac{1}{B} \sum_{b=1}^B \nabla_{\theta} \ell(y_{n(t,b)}, f(\mathbf{x}_{n(t,b)}; \theta_t))$$
$$\theta_{t+1} = \theta_t - \gamma g_t,$$

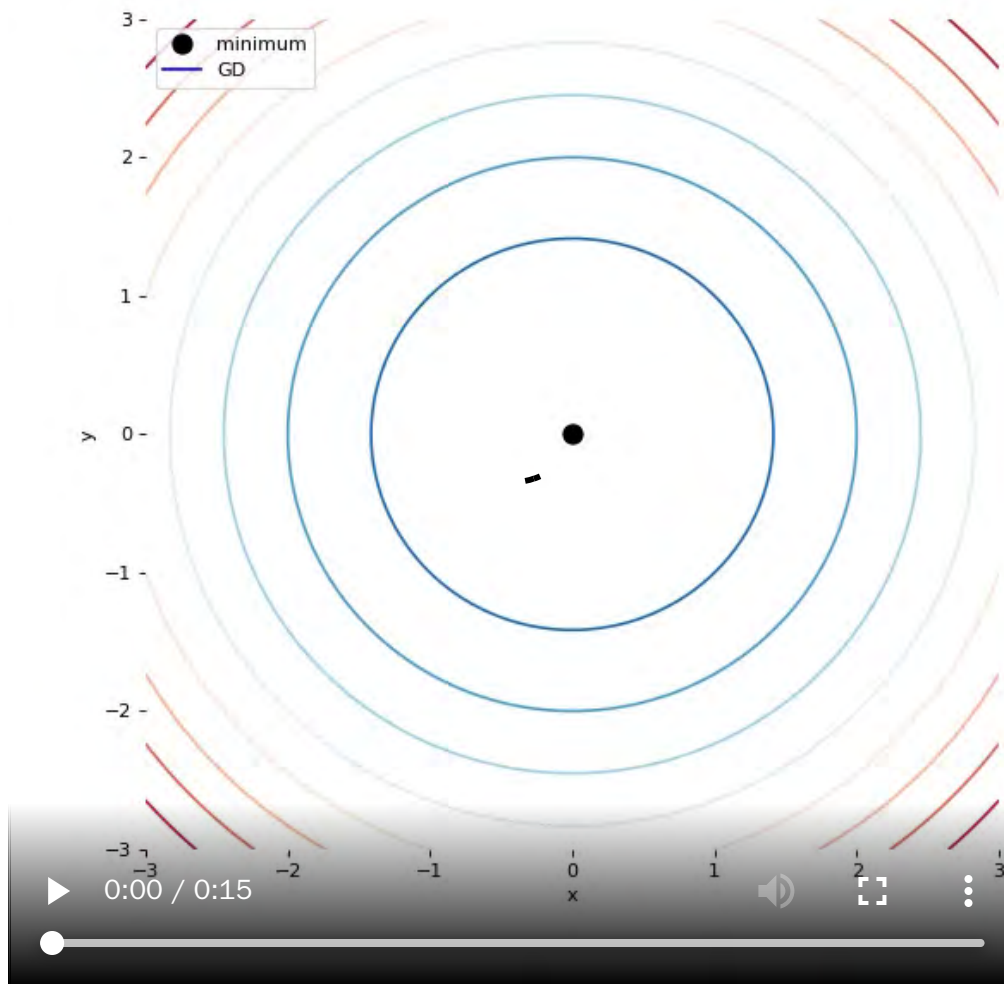
where the order $n(t, b)$ to visit the samples can be either sequential or random.

- Increasing the batch size B reduces the variance of the gradient estimates and enables the speed-up of batch processing.
- The interplay between B and γ is still unclear.

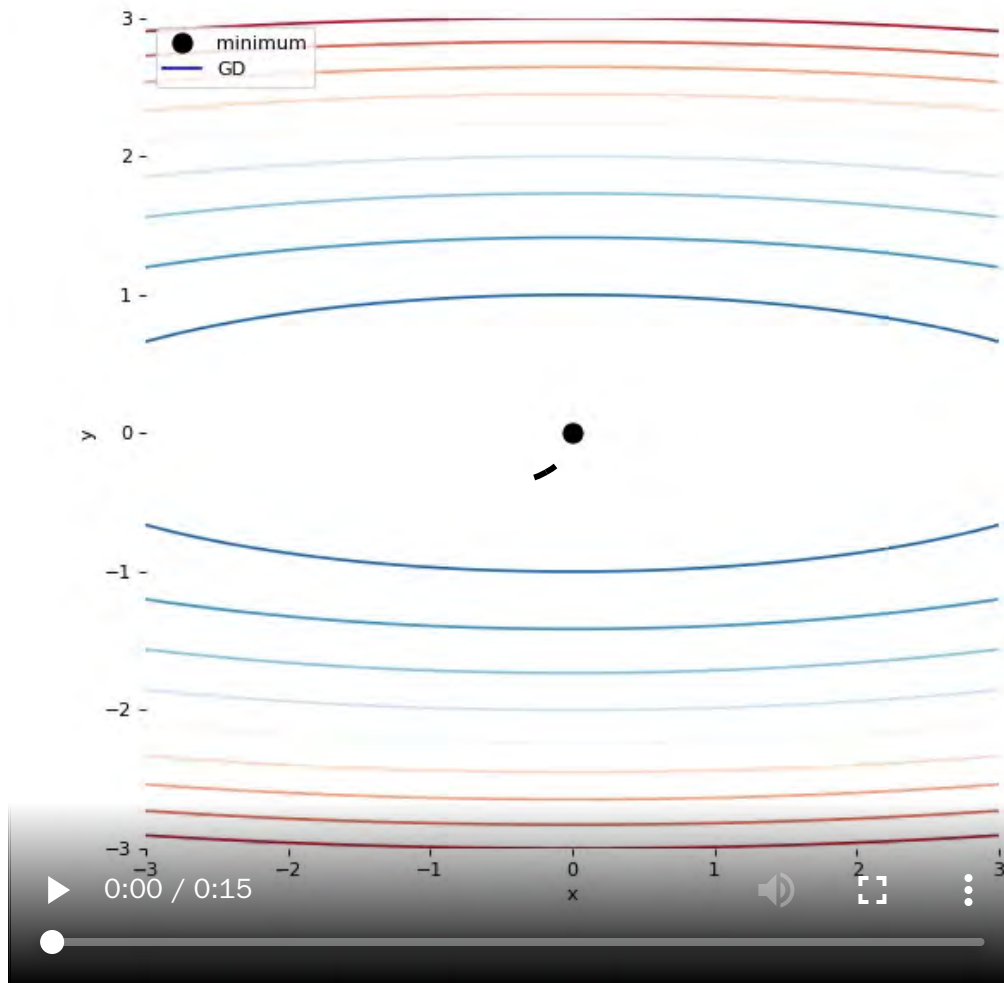
Limitations

The gradient descent method makes strong assumptions about

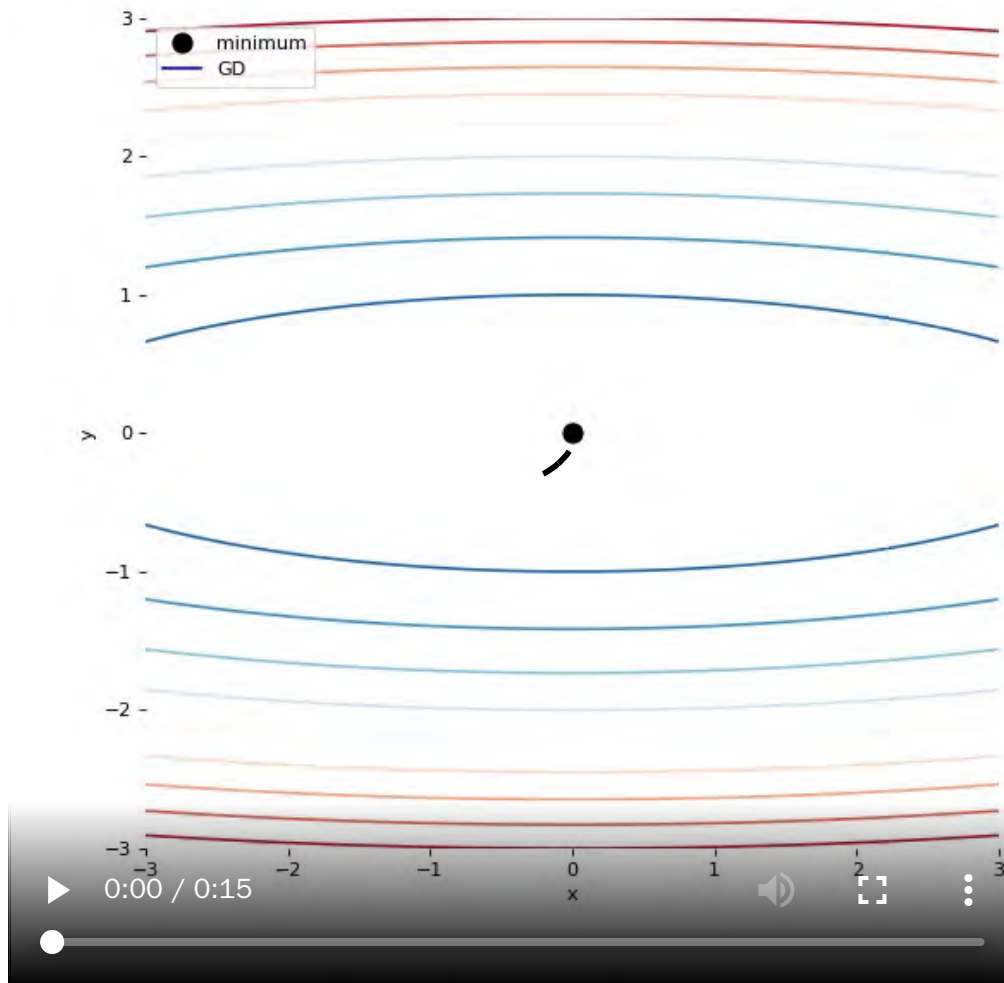
- the magnitude of the local curvature to set the step size,
- the isotropy of the curvature, so that the same step size γ makes sense in all directions.



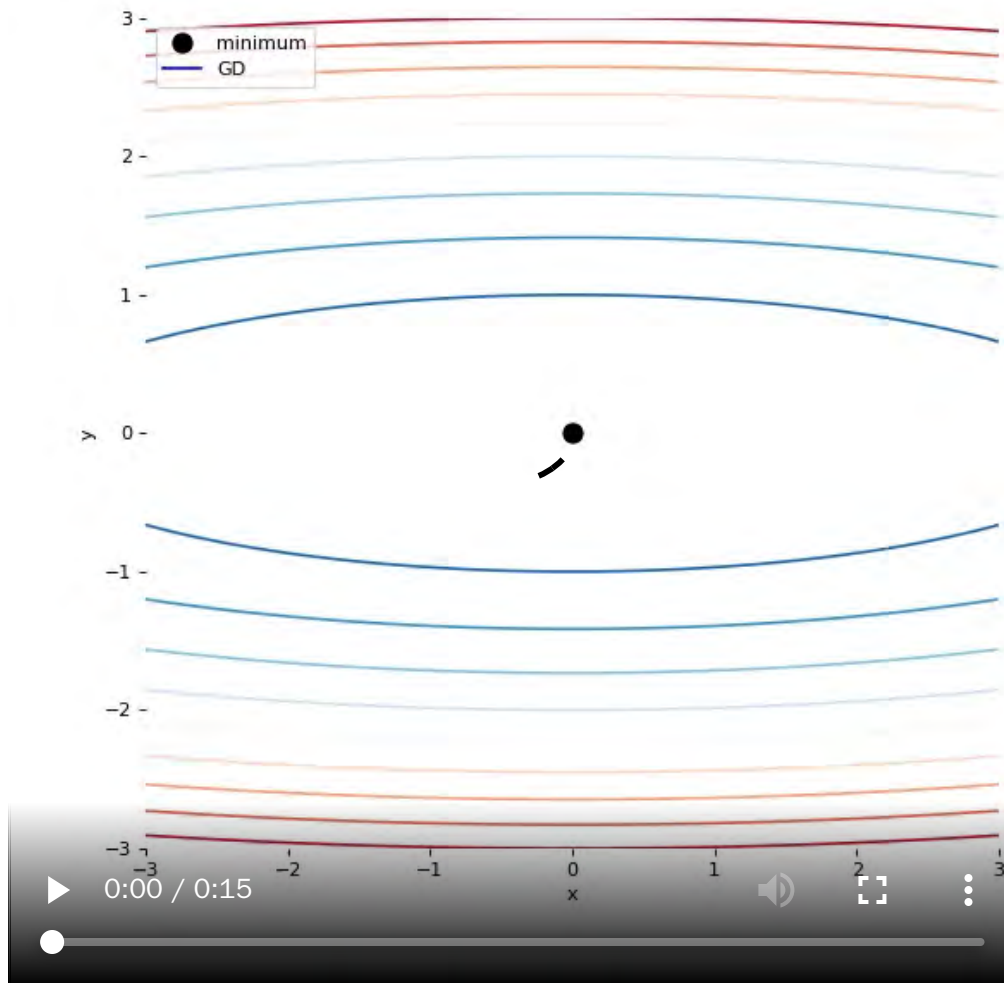
$$\gamma = 0.01$$



$$\gamma = 0.01$$



$$\gamma = 0.1$$



$$\gamma = 0.4$$

Wolfe conditions

Let us consider a function f to minimize along x , following a direction of descent p .

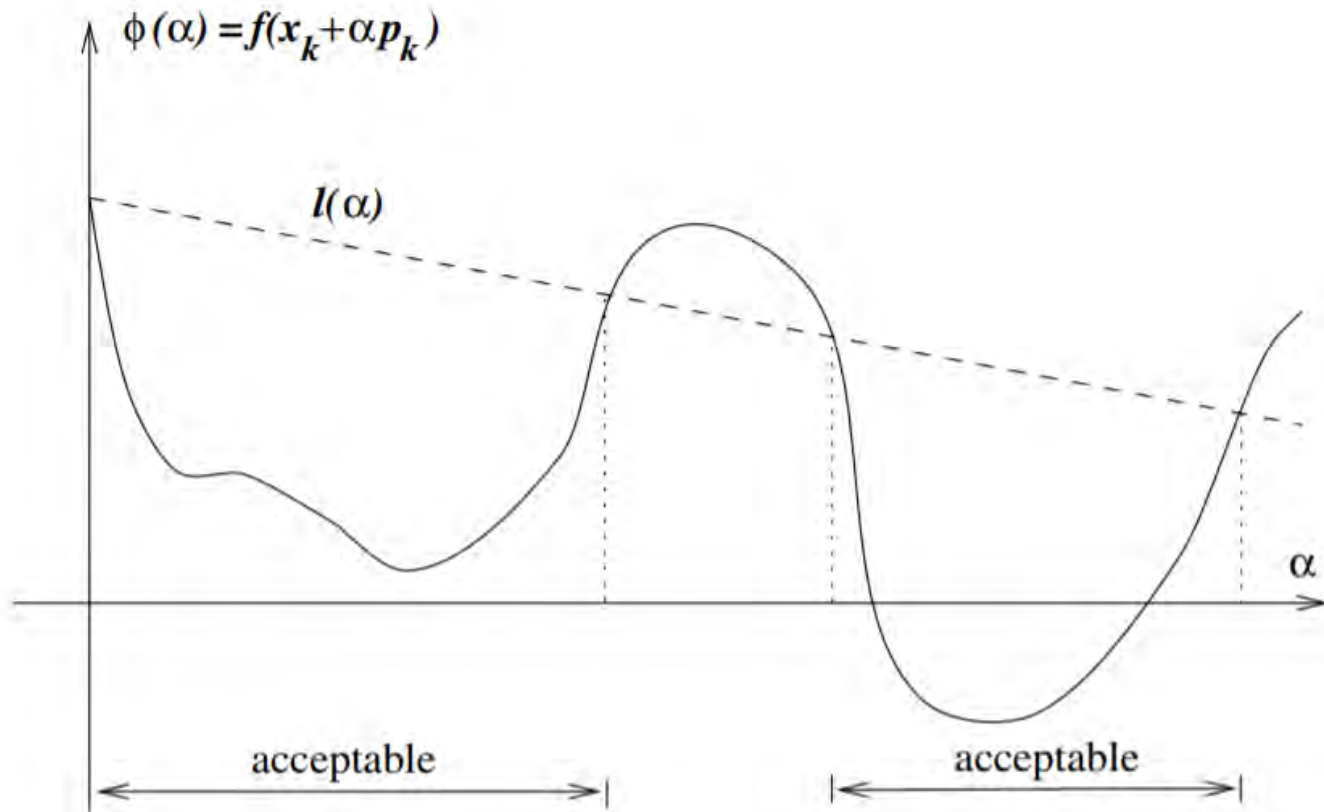
For $0 < c_1 < c_2 < 1$, the Wolfe conditions on the step size γ are as follows:

- Sufficient decrease condition:

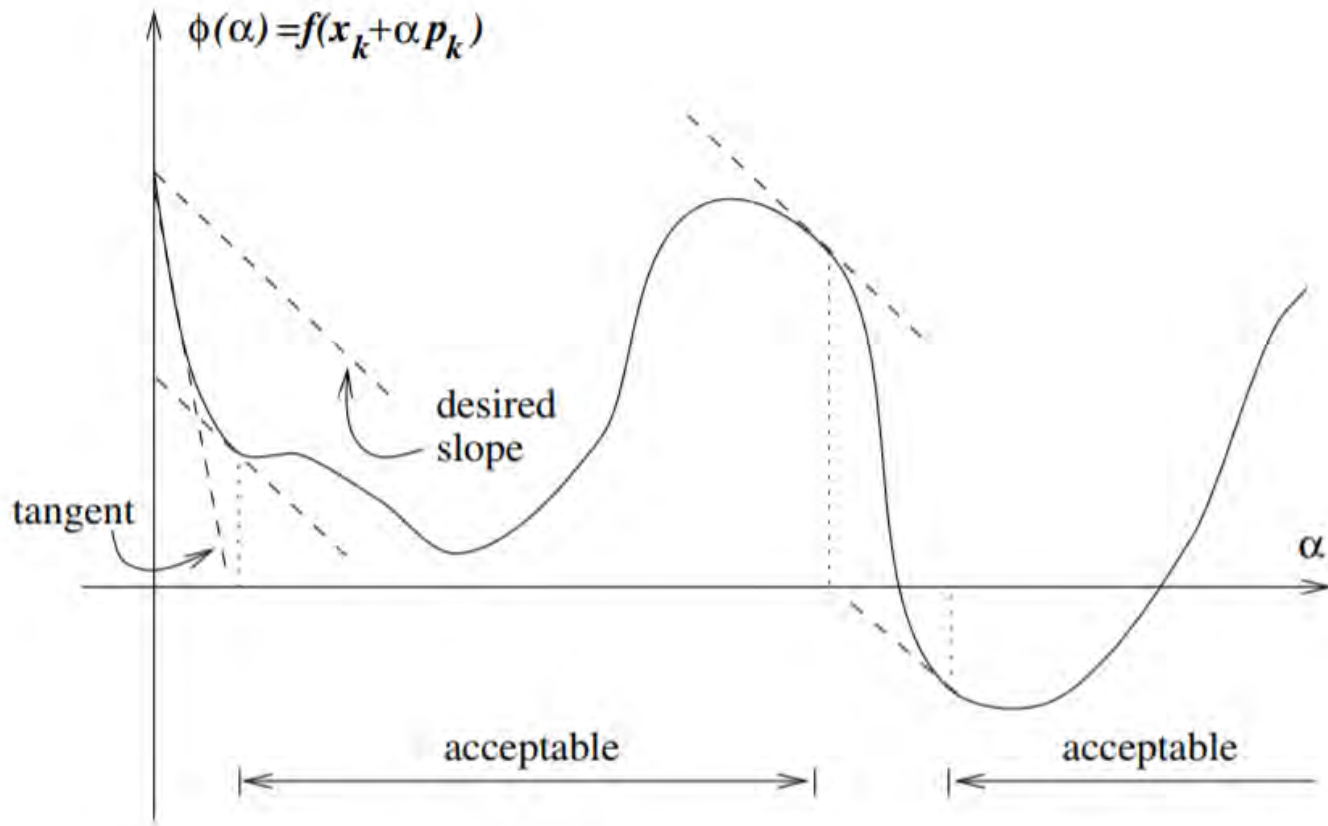
$$f(x + \gamma p) \leq f(x) + c_1 \gamma p^T \nabla f(x)$$

- Curvature condition:

$$c_2 p^T \nabla f(x) \leq p^T \nabla f(x + \gamma p)$$



The sufficient decrease condition ensures that f decreases sufficiently.
 (α is the step size.)



The curvature condition ensures that the slope has been reduced sufficiently.

The Wolfe conditions can be used to design **line search** algorithms to automatically determine a step size γ_t , hence **ensuring convergence** towards a local minima.

However, in deep learning,

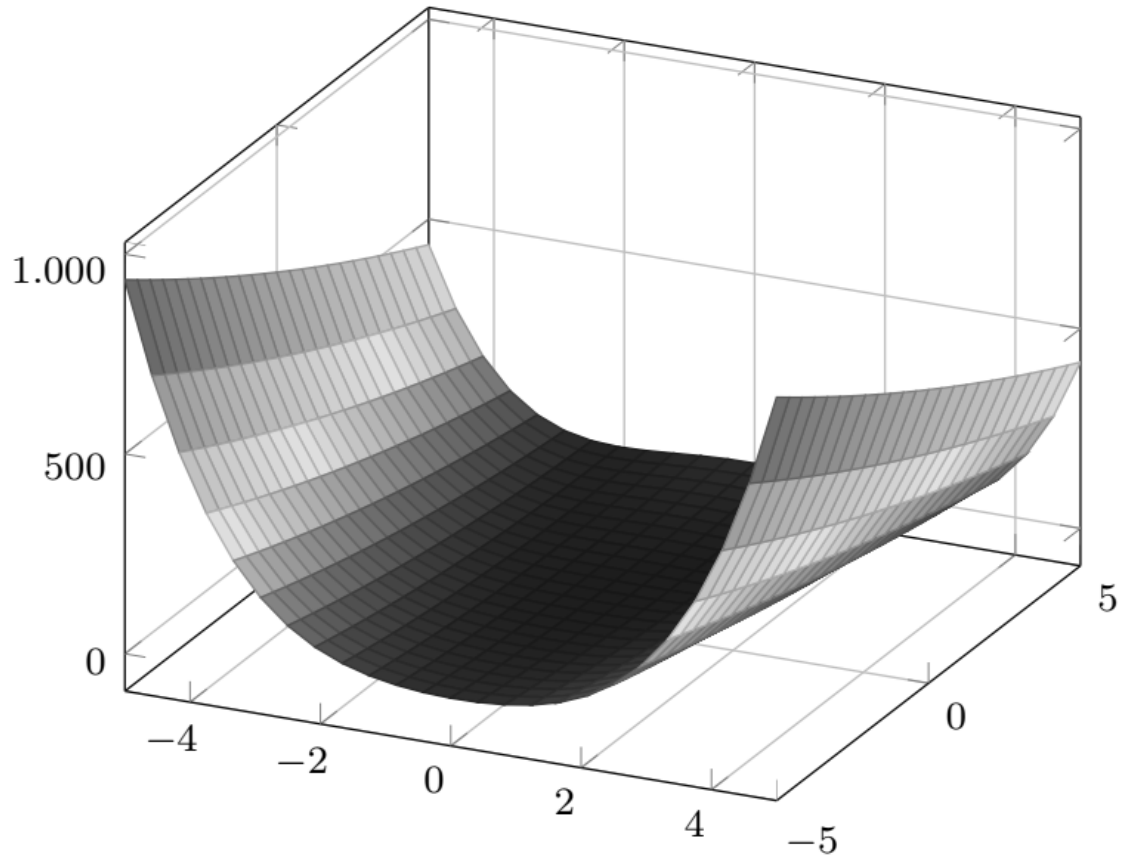
- these algorithms are impractical because of the size of the parameter space and the overhead it would induce,
- they might lead to overfitting when the empirical risk is minimized too well.

The tradeoffs of learning

When decomposing the excess error in terms of approximation, estimation and optimization errors, stochastic algorithms yield the best generalization performance (in terms of **expected** risk) despite being the worst optimization algorithms (in terms of **empirical risk**) (Bottou, 2011).

$$\begin{aligned} & \mathbb{E} \left[R(\tilde{f}_*^{\mathbf{d}}) - R(f_B) \right] \\ &= \mathbb{E} \left[R(f_*) - R(f_B) \right] + \mathbb{E} \left[R(f_*^{\mathbf{d}}) - R(f_*) \right] + \mathbb{E} \left[R(\tilde{f}_*^{\mathbf{d}}) - R(f_*^{\mathbf{d}}) \right] \\ &= \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}} \end{aligned}$$

Momentum

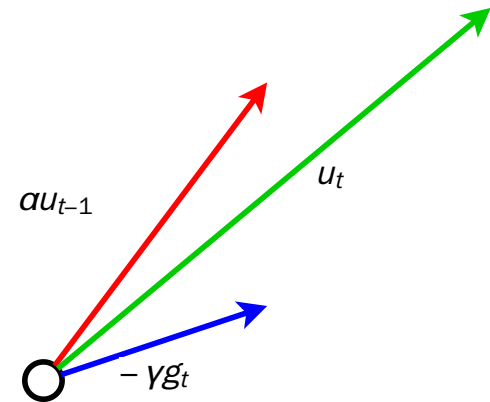


In the situation of small but consistent gradients, as through valley floors, gradient descent moves **very slowly**.

An improvement to gradient descent is to use **momentum** to add inertia in the choice of the step direction, that is

$$u_t = \alpha u_{t-1} - \gamma g_t$$
$$\theta_{t+1} = \theta_t + u_t.$$

- The new variable u_t is the velocity. It corresponds to the direction and speed by which the parameters move as the learning dynamics progresses, modeled as an exponentially decaying moving average of negative gradients.
- Gradient descent with momentum has three nice properties:
 - it can go through local barriers,
 - it accelerates if the gradient does not change much,
 - it dampens oscillations in narrow valleys.

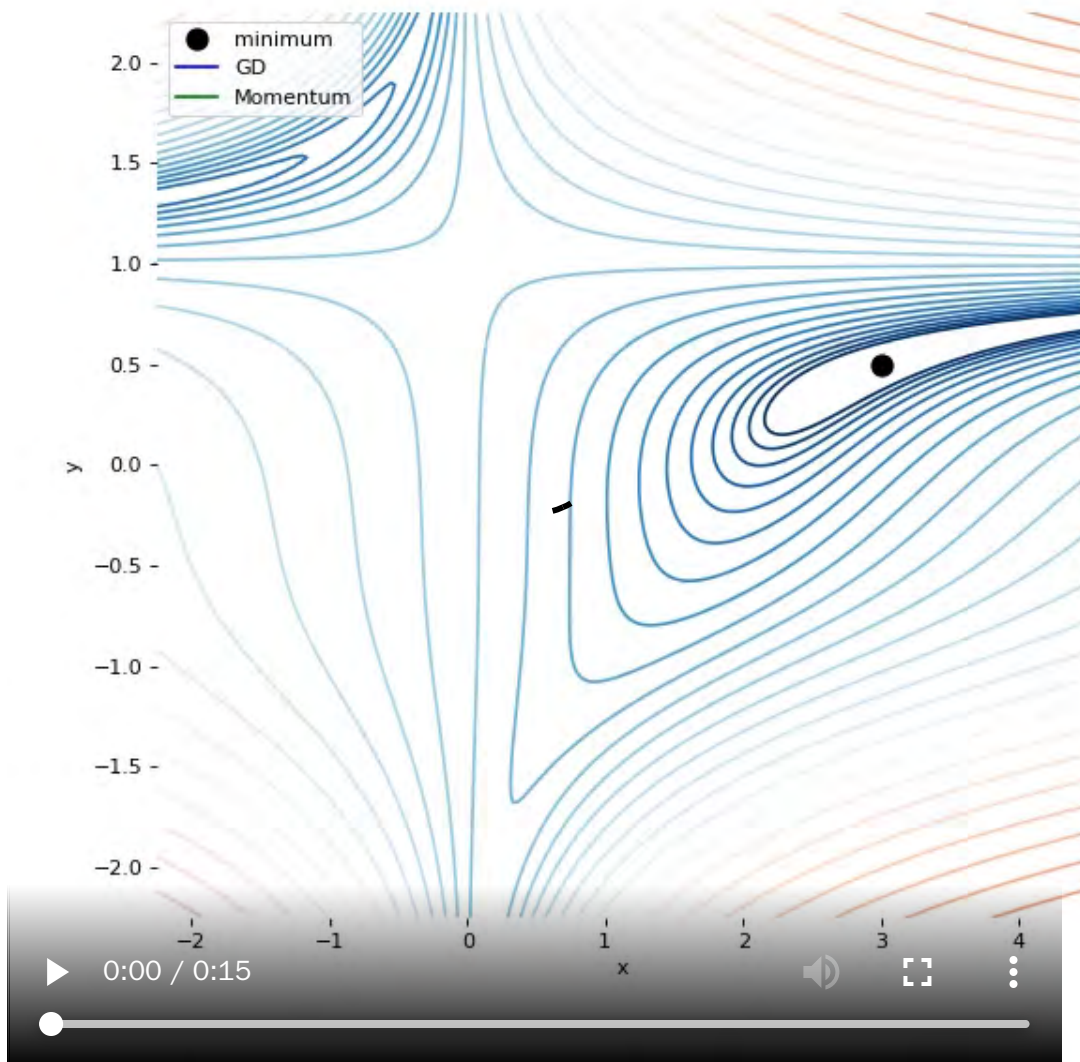


The hyper-parameter α controls how recent gradients affect the current update.

- Usually, $\alpha = 0.9$, with $\alpha > \gamma$.
- If at each update we observed g , the step would (eventually) be

$$u = -\frac{\gamma}{1 - \alpha}g.$$

- Therefore, for $\alpha = 0.9$, it is like multiplying the maximum speed by 10 relative to the current direction.



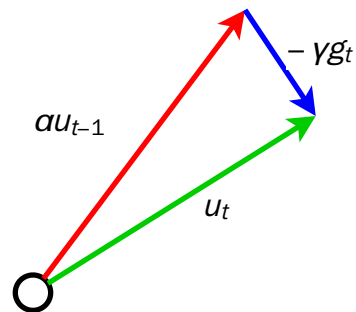
Nesterov momentum

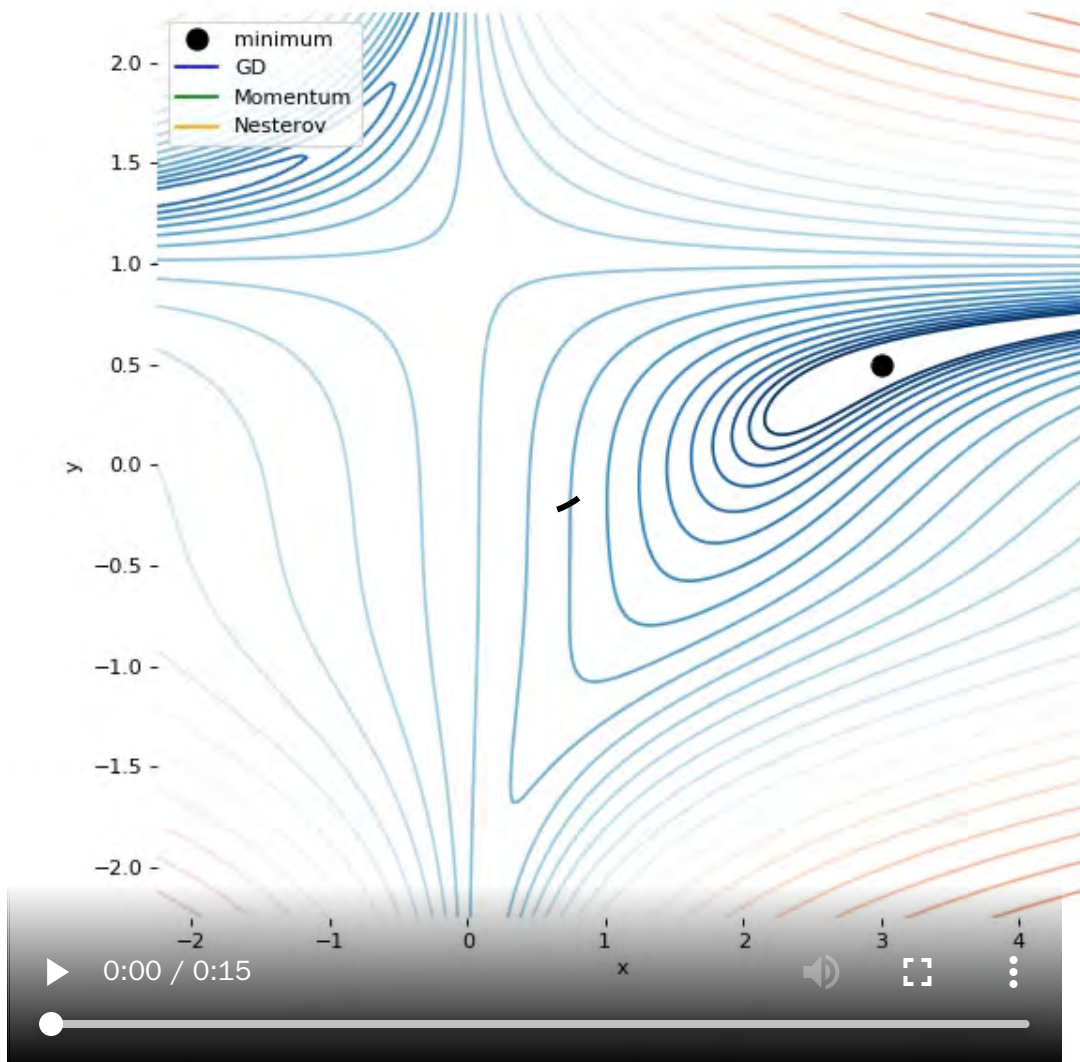
An alternative consists in simulating a step in the direction of the velocity, then calculate the gradient and make a correction.

$$g_t = \frac{1}{N} \sum_{n=1}^N \nabla_{\theta} \ell(y_n, f(\mathbf{x}_n; \theta_t + \alpha u_{t-1}))$$

$$u_t = \alpha u_{t-1} - \gamma g_t$$

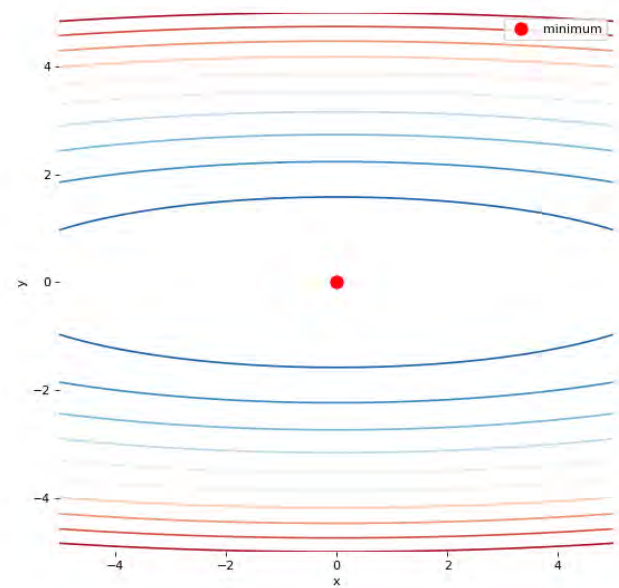
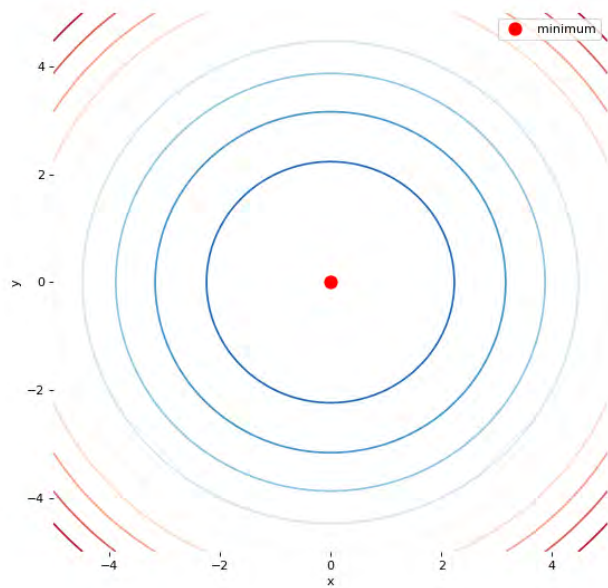
$$\theta_{t+1} = \theta_t + u_t$$





Adaptive learning rate

Vanilla gradient descent assumes the isotropy of the curvature, so that the same step size γ applies to all parameters.



Isotropic vs. Anisotropic

AdaGrad

Per-parameter downscale by square-root of sum of squares of all its historical values.

$$r_t = r_{t-1} + g_t \odot g_t$$
$$\theta_{t+1} = \theta_t - \frac{\gamma}{\delta + \sqrt{r_t}} \odot g_t.$$

- AdaGrad eliminates the need to manually tune the learning rate. Most implementation use $\gamma = 0.01$ as default.
- It is good when the objective is convex.
- r_t grows unboundedly during training, which may cause the step size to shrink and eventually become infinitesimally small.

RMSProp

Same as AdaGrad but accumulate an exponentially decaying average of the gradient.

$$r_t = \rho r_{t-1} + (1 - \rho) g_t \odot g_t$$
$$\theta_{t+1} = \theta_t - \frac{\gamma}{\delta + \sqrt{r_t}} \odot g_t.$$

- Perform better in non-convex settings.
- Does not grow unboundedly.

Adam

Similar to RMSProp with momentum, but with bias correction terms for the first and second moments.

$$s_t = \rho_1 s_{t-1} + (1 - \rho_1) g_t$$

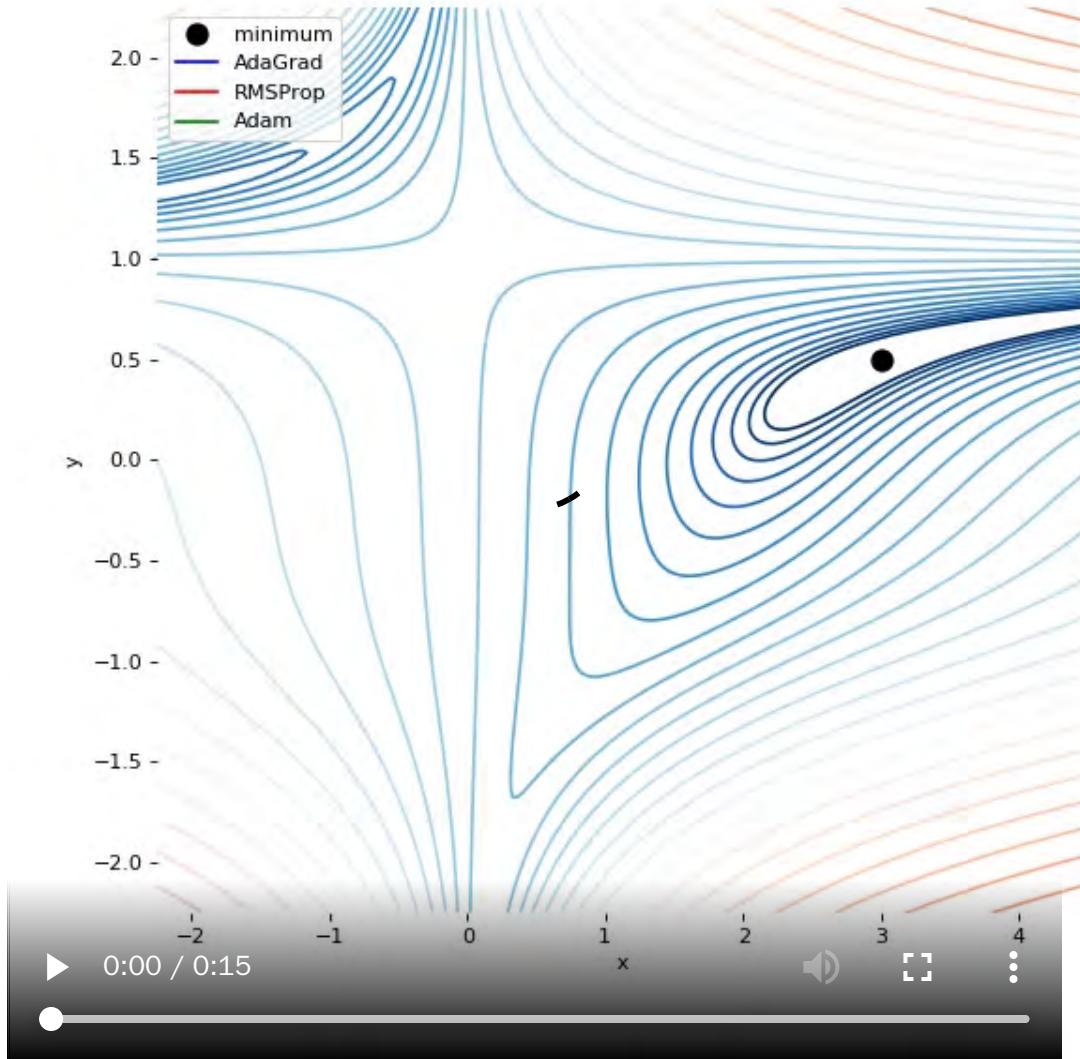
$$\hat{s}_t = \frac{s_t}{1 - \rho_1^t}$$

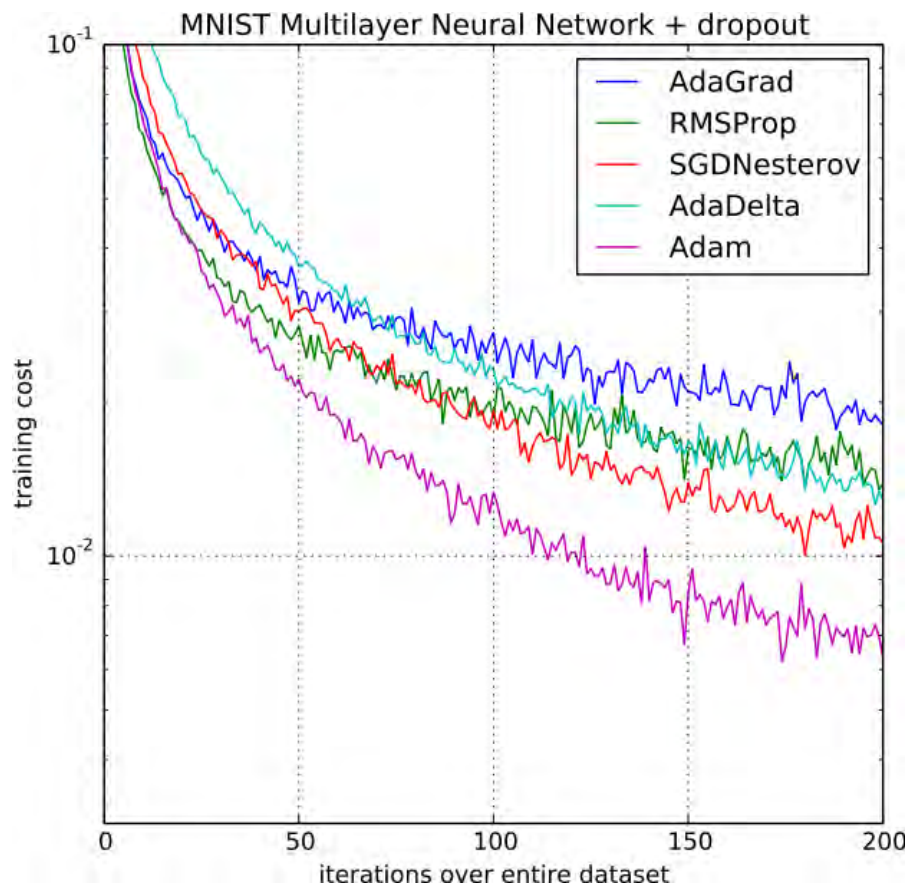
$$r_t = \rho_2 r_{t-1} + (1 - \rho_2) g_t \odot g_t$$

$$\hat{r}_t = \frac{r_t}{1 - \rho_2^t}$$

$$\theta_{t+1} = \theta_t - \gamma \frac{\hat{s}_t}{\delta + \sqrt{\hat{r}_t}}$$

- Good defaults are $\rho_1 = 0.9$ and $\rho_2 = 0.999$.
- Adam is one of the **default optimizers** in deep learning, along with SGD with momentum.

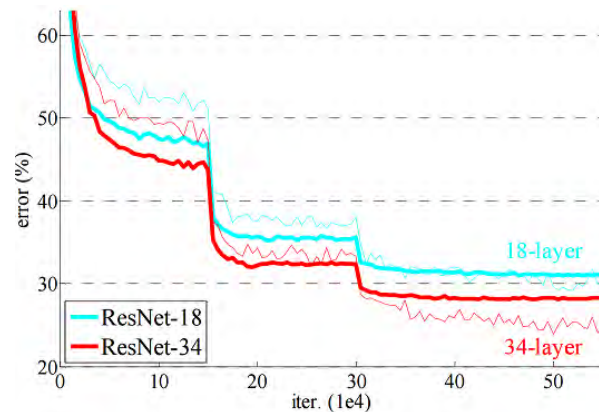




Scheduling

Despite per-parameter adaptive learning rate methods, it is usually helpful to **anneal the learning rate** γ over time.

- Step decay: reduce the learning rate by some factor every few epochs (e.g, by half every 10 epochs).
- Exponential decay: $\gamma_t = \gamma_0 \exp(-kt)$ where γ_0 and k are hyper-parameters.
- $1/t$ decay: $\gamma_t = \gamma_0 / (1 + kt)$ where γ_0 and k are hyper-parameters.



Step decay scheduling for training ResNets.

Initialization

- In convex problems, provided a good learning rate γ , convergence is guaranteed regardless of the **initial parameter values**.
- In the non-convex regime, initialization is **much more important!**
- Little is known on the mathematics of initialization strategies of neural networks.
 - What is known: initialization should break symmetry.
 - What is known: the scale of weights is important.

Controlling for the variance in the forward pass

A first strategy is to initialize the network parameters such that activations preserve the **same variance across layers**.

Intuitively, this ensures that the information keeps flowing during the **forward pass**, without reducing or magnifying the magnitude of input signals exponentially.

Let us assume that

- we are in a linear regime at initialization (e.g., the positive part of a ReLU or the middle of a sigmoid),
- weights w_{ij}^l are initialized independently,
- biases b_l are initialized to be 0,
- input feature variances are the same, which we denote as $\mathbb{V}[\mathbf{x}]$.

Then, the variance of the activation h_i^l of unit i in layer l is

$$\begin{aligned}\mathbb{V}[h_i^l] &= \mathbb{V}\left[\sum_{j=0}^{q_{l-1}-1} w_{ij}^l h_j^{l-1}\right] \\ &= \sum_{j=0}^{q_{l-1}-1} \mathbb{V}[w_{ij}^l] \mathbb{V}[h_j^{l-1}]\end{aligned}$$

where q_l is the width of layer l and $h_j^0 = x_j$ for all $j = 0, \dots, p - 1$.

If we further assume that weights w_{ij}^l at layer l share the same variance $\mathbb{V} [w^l]$ and that the variance of the activations in the previous layer are the same, then we can drop the indices and write

$$\mathbb{V} [h^l] = q_{l-1} \mathbb{V} [w^l] \mathbb{V} [h^{l-1}].$$

Therefore, the variance of the activations is preserved across layers when

$$\mathbb{V} [w^l] = \frac{1}{q_{l-1}} \quad \forall l.$$

This condition is enforced in **LeCun's uniform initialization**, which is defined as

$$w_{ij}^l \sim \mathcal{U} \left[-\sqrt{\frac{3}{q_{l-1}}}, \sqrt{\frac{3}{q_{l-1}}} \right].$$

Controlling for the variance in the backward pass

A similar idea can be applied to ensure that the gradients flow in the **backward pass** (without vanishing nor exploding), by maintaining the variance of the gradient with respect to the activations fixed across layers.

Under the same assumptions as before,

$$\begin{aligned}\mathbb{V} \left[\frac{d\hat{y}}{dh_i^l} \right] &= \mathbb{V} \left[\sum_{j=0}^{q_{l+1}-1} \frac{d\hat{y}}{dh_j^{l+1}} \frac{\partial h_j^{l+1}}{\partial h_i^l} \right] \\ &= \mathbb{V} \left[\sum_{j=0}^{q_{l+1}-1} \frac{d\hat{y}}{dh_j^{l+1}} w_{j,i}^{l+1} \right] \\ &= \sum_{j=0}^{q_{l+1}-1} \mathbb{V} \left[\frac{d\hat{y}}{dh_j^{l+1}} \right] \mathbb{V} [w_{ji}^{l+1}]\end{aligned}$$

If we further assume that

- the gradients of the activations at layer l share the same variance
- the weights at layer $l + 1$ share the same variance $\mathbb{V} [w^{l+1}]$,

then we can drop the indices and write

$$\mathbb{V} \left[\frac{d\hat{y}}{dh^l} \right] = q_{l+1} \mathbb{V} \left[\frac{d\hat{y}}{dh^{l+1}} \right] \mathbb{V} [w^{l+1}].$$

Therefore, the variance of the gradients with respect to the activations is preserved across layers when

$$\mathbb{V} [w^l] = \frac{1}{q_l} \quad \forall l.$$

Xavier initialization

We have derived two different conditions on the variance of w^l ,

- $\mathbb{V} [w^l] = \frac{1}{q_{l-1}}$
- $\mathbb{V} [w^l] = \frac{1}{q_l}$.

A compromise is the **Xavier initialization**, which initializes w^l randomly from a distribution with variance

$$\mathbb{V} [w^l] = \frac{1}{\frac{q_{l-1} + q_l}{2}} = \frac{2}{q_{l-1} + q_l}.$$

For example, **normalized initialization** is defined as

$$w_{ij}^l \sim \mathcal{U} \left[-\sqrt{\frac{6}{q_{l-1} + q_l}}, \sqrt{\frac{6}{q_{l-1} + q_l}} \right].$$

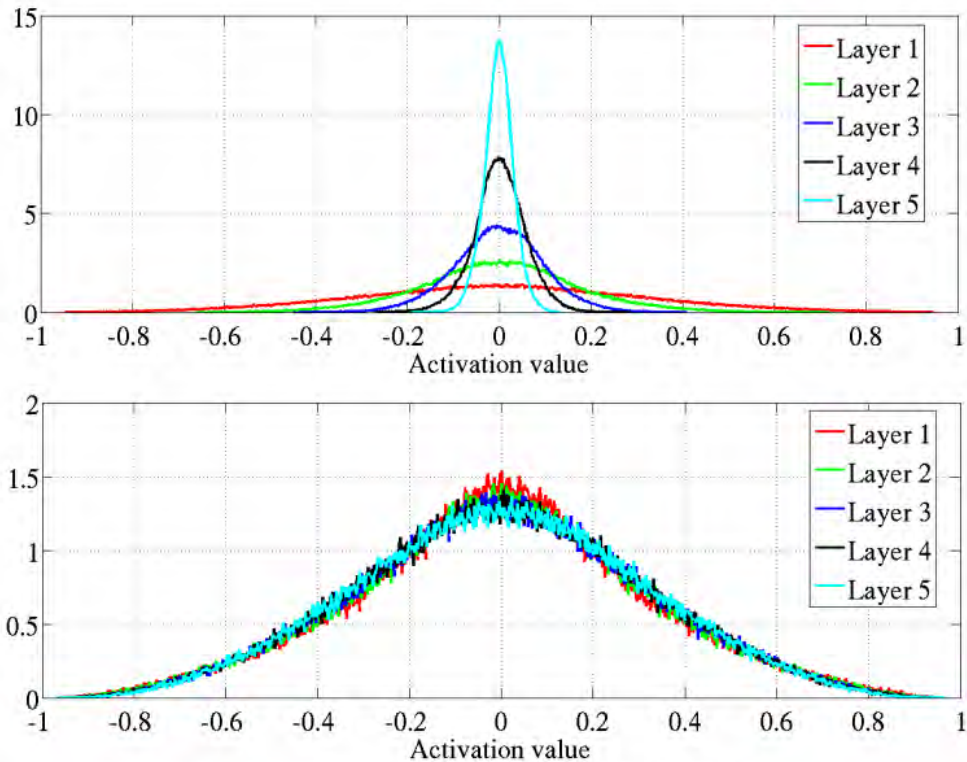


Figure 6: *Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.*

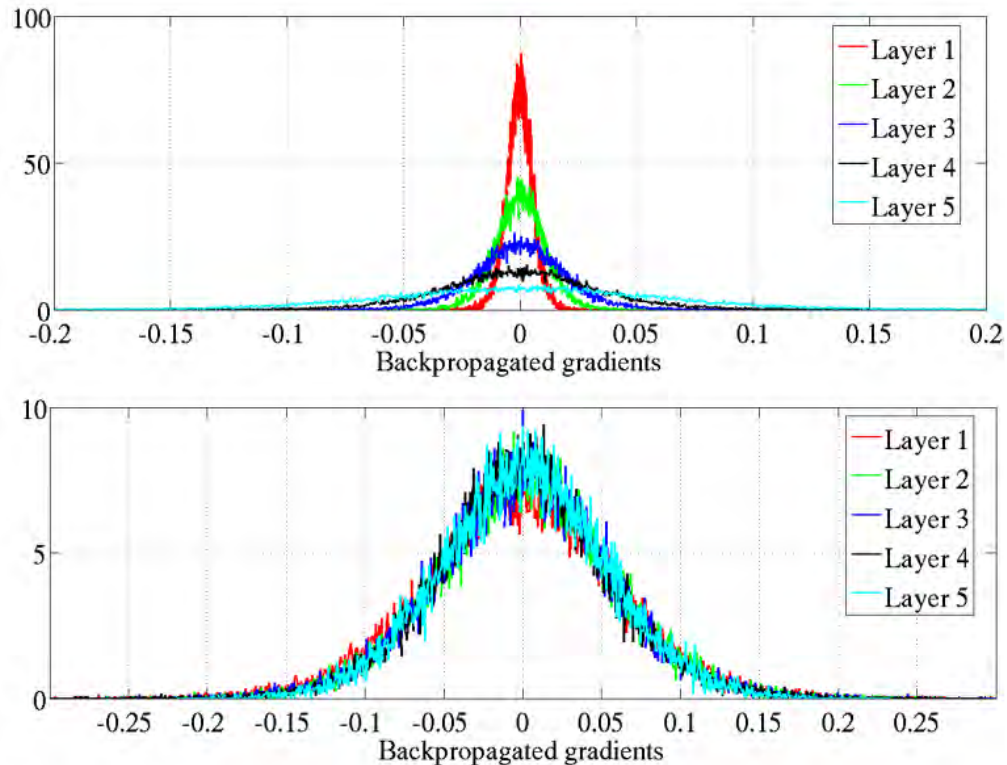


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Normalization

Data normalization

Previous weight initialization strategies rely on preserving the activation variance constant across layers, under the initial assumption that the **input feature variances are the same**.

That is,

$$\mathbb{V}[x_i] = \mathbb{V}[x_j] \triangleq \mathbb{V}[x]$$

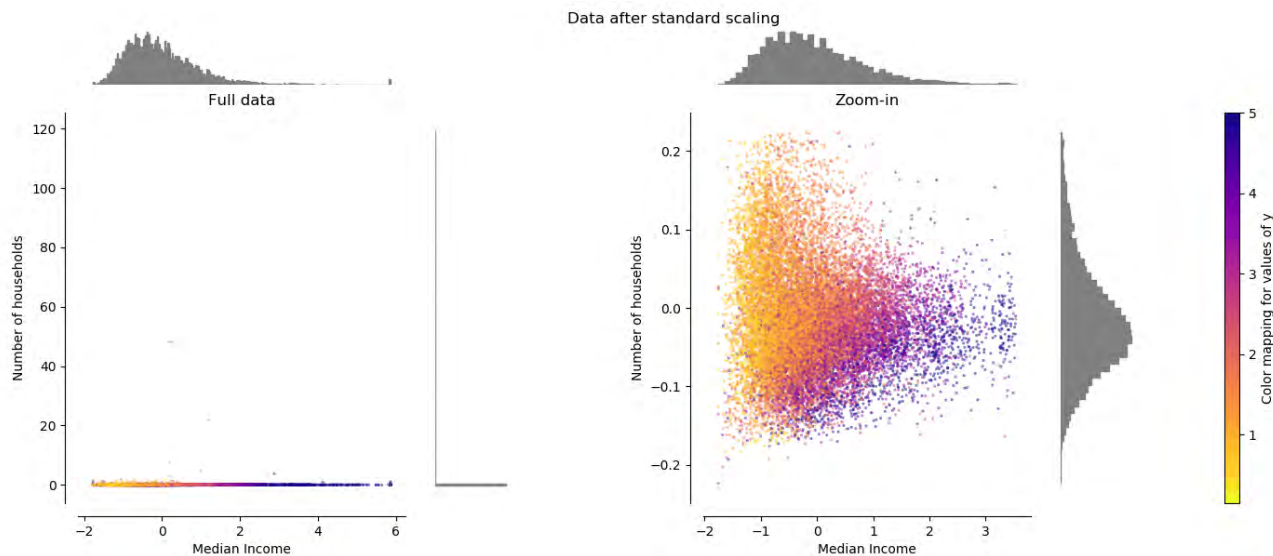
for all pairs of features i, j .

In general, this constraint is not satisfied but can be enforced by **standardizing** the input data feature-wise,

$$\mathbf{x}' = (\mathbf{x} - \hat{\mu}) \odot \frac{1}{\hat{\sigma}},$$

where

$$\hat{\mu} = \frac{1}{N} \sum_{\mathbf{x} \in \mathbf{d}} \mathbf{x} \quad \hat{\sigma}^2 = \frac{1}{N} \sum_{\mathbf{x} \in \mathbf{d}} (\mathbf{x} - \hat{\mu})^2.$$



Batch normalization

Maintaining proper statistics of the activations and derivatives is critical for training neural networks.

- This constraint can be enforced explicitly during the forward pass by re-normalizing them.
- **Batch normalization** was the first method introducing this idea.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe

Google Inc., sioffe@google.com

Christian Szegedy

Google Inc., szegedy@google.com

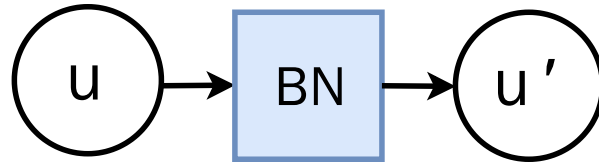
Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate*

shift. Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

While stochastic gradient is simple and effective, it

- During training, batch normalization shifts and rescales according to the mean and variance estimated on the batch.
- During test, it shifts and rescales according to the empirical moments estimated during training.



Let us consider a given minibatch of samples at training, for which $\mathbf{u}_b \in \mathbb{R}^q$, $b = 1, \dots, B$, are intermediate values computed at some location in the computational graph.

In batch normalization following the node \mathbf{u} , the per-component mean and variance are first computed on the batch

$$\hat{\mu}_{\text{batch}} = \frac{1}{B} \sum_{b=1}^B \mathbf{u}_b \quad \hat{\sigma}_{\text{batch}}^2 = \frac{1}{B} \sum_{b=1}^B (\mathbf{u}_b - \hat{\mu}_{\text{batch}})^2,$$

from which the standardized $\mathbf{u}'_b \in \mathbb{R}^q$ are computed such that

$$\mathbf{u}'_b = \gamma \odot (\mathbf{u}_b - \hat{\mu}_{\text{batch}}) \odot \frac{1}{\hat{\sigma}_{\text{batch}} + \epsilon} + \beta$$

where $\gamma, \beta \in \mathbb{R}^q$ are parameters to optimize.

Exercise: How does batch normalization combine with backpropagation?

During inference, batch normalization shifts and rescales each component according to the empirical moments estimated during training:

$$\mathbf{u}' = \gamma \odot (\mathbf{u} - \hat{\boldsymbol{\mu}}) \odot \frac{1}{\hat{\boldsymbol{\sigma}}} + \beta.$$

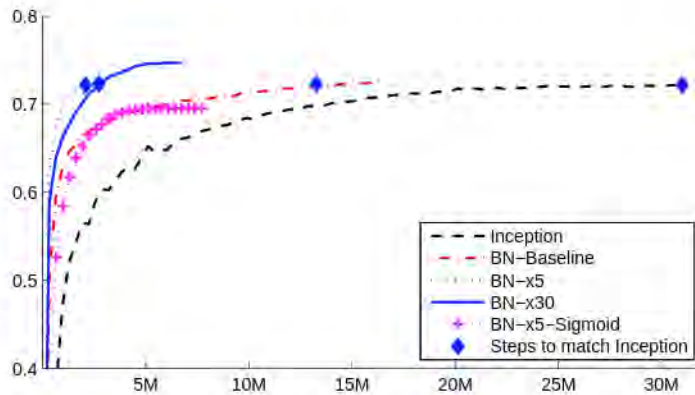


Figure 2: Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
<i>BN-Baseline</i>	$13.3 \cdot 10^6$	72.7%
<i>BN-x5</i>	$2.1 \cdot 10^6$	73.0%
<i>BN-x30</i>	$2.7 \cdot 10^6$	74.8%
<i>BN-x5-Sigmoid</i>		69.8%

Figure 3: For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.

The position of batch normalization relative to the non-linearity is not clear.

where W and b are learned parameters of the model, and $g(\cdot)$ is the nonlinearity such as sigmoid or ReLU. This formulation covers both fully-connected and convolutional layers. We add the BN transform immediately before the nonlinearity, by normalizing $x = W'u + b$. We could have also normalized the layer inputs u , but since u is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. In contrast, $W'u + b$ is more likely to have a symmetric, non-sparse distribution, that is “more Gaussian” (Hyvärinen & Oja, 2000); normalizing it is likely to produce activations with a stable distribution.

Layer normalization

Given a single input sample \mathbf{x} , a similar approach can be applied to standardize the activations \mathbf{u} across a layer instead of doing it over the batch.

The end.

Deep Learning

Lecture 5: Recurrent neural networks

Prof. Gilles Louppe
g.louppe@uliege.be

Today

How to make sense of [sequential data](#)?

- Recurrent neural networks
- Applications
- Differentiable computers

Many real-world problems require to process a signal with a **sequence** structure.

- Sequence classification:
 - sentiment analysis
 - activity/action recognition
 - DNA sequence classification
 - action selection
- Sequence synthesis:
 - text synthesis
 - music synthesis
 - motion synthesis
- Sequence-to-sequence translation:
 - speech recognition
 - text translation
 - part-of-speech tagging

Given a set \mathcal{X} , if $S(\mathcal{X})$ denotes the set of sequences of elements from \mathcal{X} ,

$$S(\mathcal{X}) = \cup_{t=1}^{\infty} \mathcal{X}^t,$$

then we formally define:

Sequence classification

$$f : S(\mathcal{X}) \rightarrow \{1, \dots, C\}$$

Sequence synthesis

$$f : \mathbb{R}^d \rightarrow S(\mathcal{X})$$

Sequence-to-sequence translation

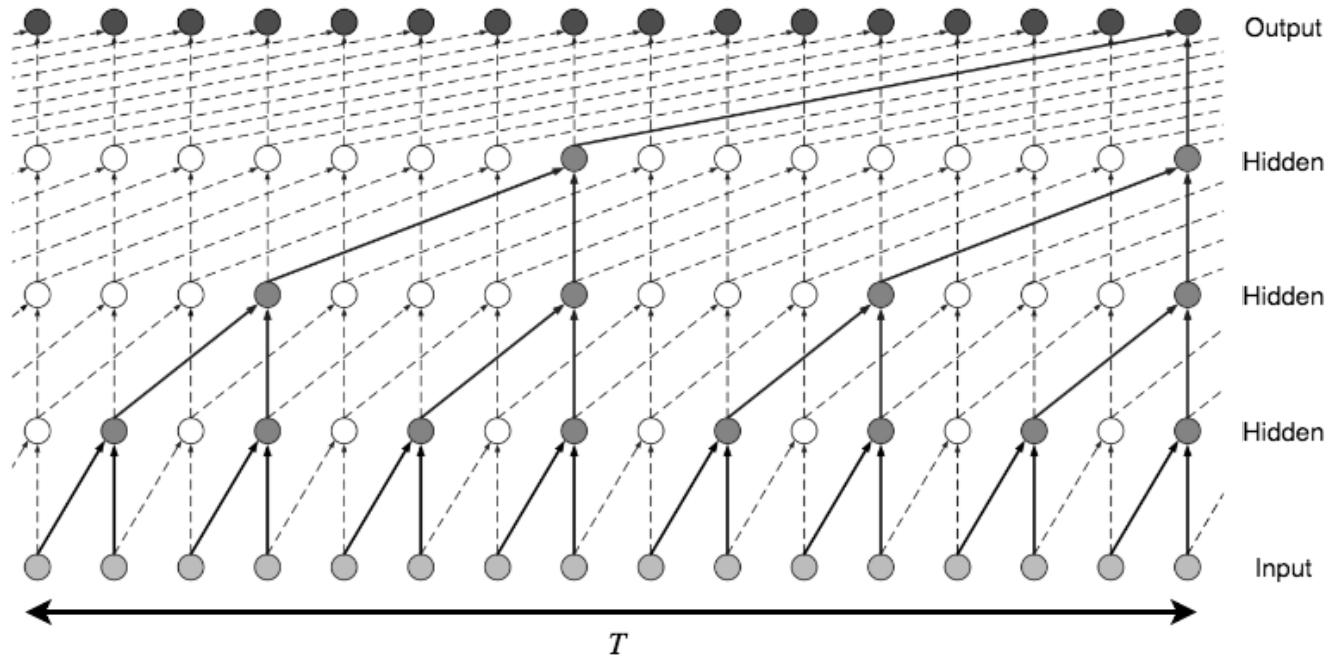
$$f : S(\mathcal{X}) \rightarrow S(\mathcal{Y})$$

In the rest of the slides, we consider only time-indexed signal, although it generalizes to arbitrary sequences.

Temporal convolutions

One of the simplest approach to sequence processing is to use **temporal convolutional networks** (TCNs).

- TCNs correspond to standard 1D convolutional networks.
- They process input sequences as fixed-size vectors of the **maximum possible length**.



Complexity:

- Increasing the window size T makes the required number of layers grow as $O(\log T)$.
- Thanks to dilated convolutions, the model size is $O(\log T)$.
- The memory footprint and computation are $O(T \log T)$.

Table 1. Evaluation of TCNs and recurrent architectures on synthetic stress tests, polyphonic music modeling, character-level language modeling, and word-level language modeling. The generic TCN architecture outperforms canonical recurrent networks across a comprehensive suite of tasks and datasets. Current state-of-the-art results are listed in the supplement. ^h means that higher is better. ^ℓ means that lower is better.

Sequence Modeling Task	Model Size (\approx)	Models			
		LSTM	GRU	RNN	TCN
Seq. MNIST (accuracy ^h)	70K	87.2	96.2	21.5	99.0
Permuted MNIST (accuracy)	70K	85.7	87.3	25.3	97.2
Adding problem $T=600$ (loss ^ℓ)	70K	0.164	5.3e-5	0.177	5.8e-5
Copy memory $T=1000$ (loss)	16K	0.0204	0.0197	0.0202	3.5e-5
Music JSB Chorales (loss)	300K	8.45	8.43	8.91	8.10
Music Nottingham (loss)	1M	3.29	3.46	4.05	3.07
Word-level PTB (perplexity ^ℓ)	13M	78.93	92.48	114.50	88.68
Word-level Wiki-103 (perplexity)	-	48.4	-	-	45.19
Word-level LAMBADA (perplexity)	-	4186	-	14725	1279
Char-level PTB (bpc ^ℓ)	3M	1.36	1.37	1.48	1.31
Char-level text8 (bpc)	5M	1.50	1.53	1.69	1.45

Recurrent neural networks

When the input is a sequence $\mathbf{x} \in \mathcal{S}(\mathbb{R}^p)$ of **variable** length $T(\mathbf{x})$, a standard approach is to use a recurrent model which maintains a **recurrent state** $\mathbf{h}_t \in \mathbb{R}^q$ updated at each time step t .

Formally, for $t = 1, \dots, T(\mathbf{x})$,

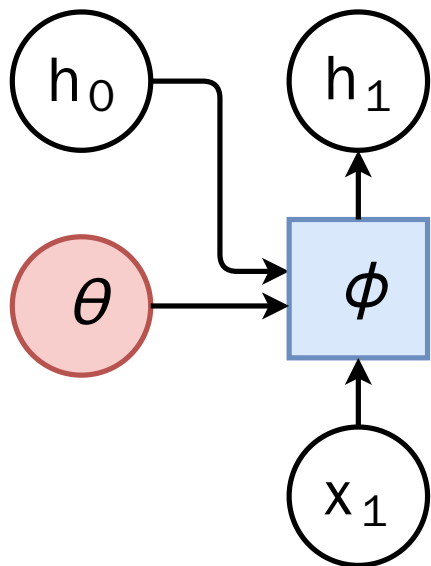
$$\mathbf{h}_t = \phi(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta),$$

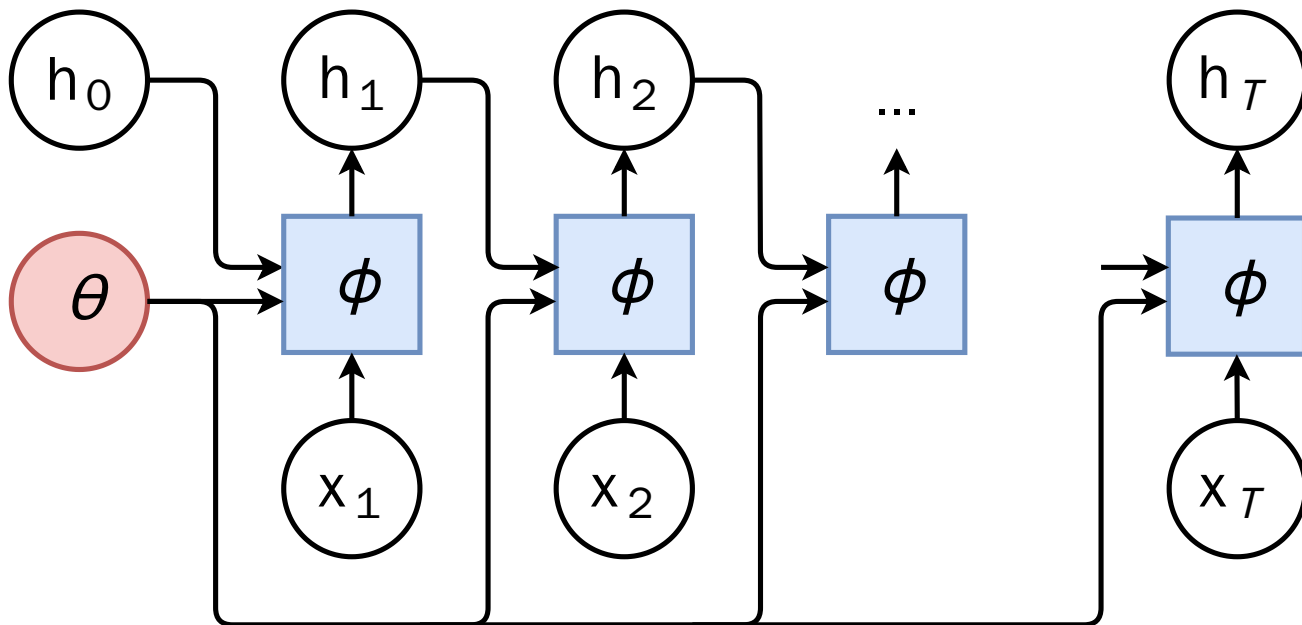
where $\phi : \mathbb{R}^p \times \mathbb{R}^q \rightarrow \mathbb{R}^q$ and $\mathbf{h}_0 \in \mathbb{R}^q$.

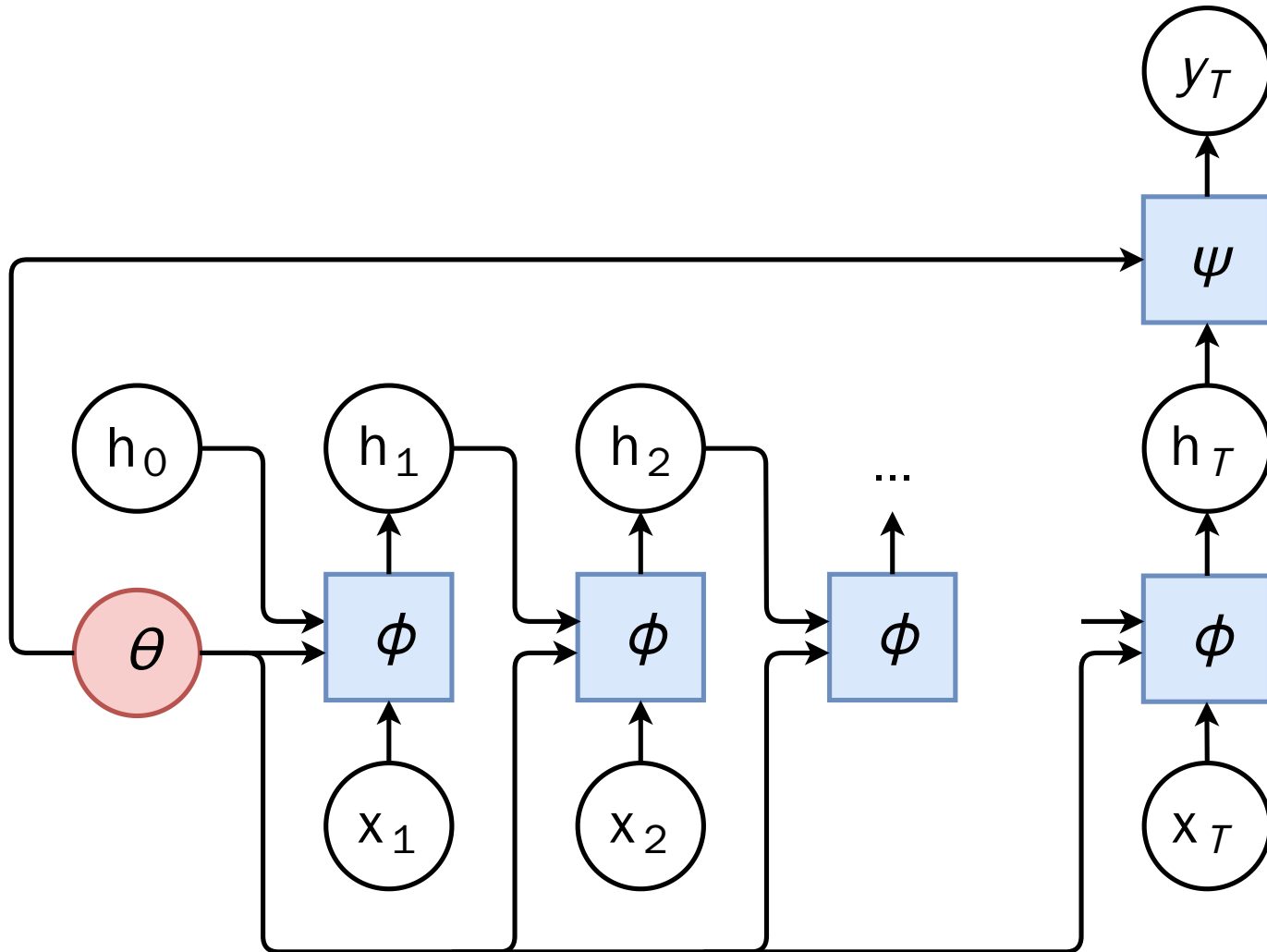
Predictions can be computed at any time step t from the recurrent state,

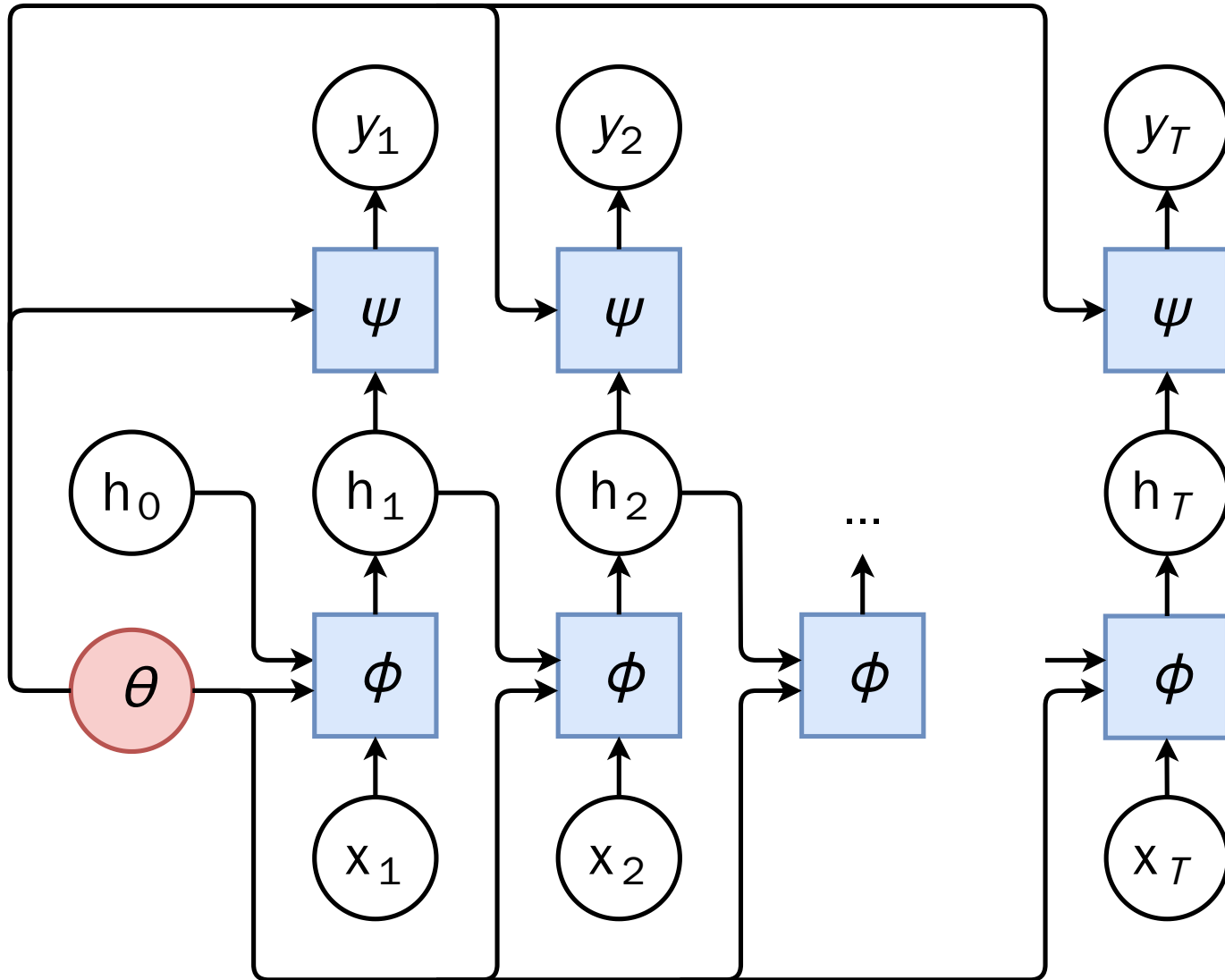
$$y_t = \psi(\mathbf{h}_t; \theta),$$

with $\psi : \mathbb{R}^q \rightarrow \mathbb{R}^C$.



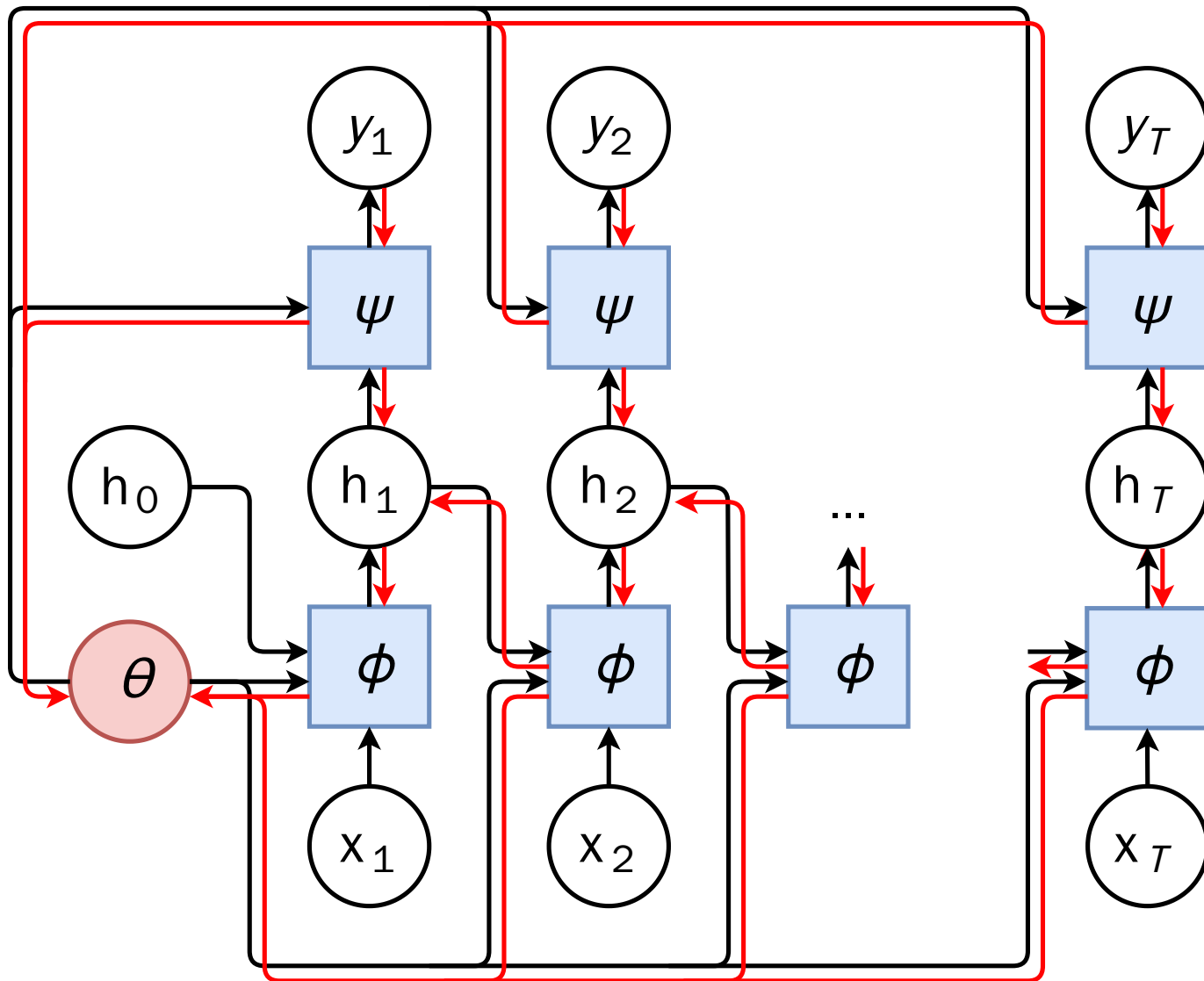






Even though the number of steps T depends on \mathbf{x} , this is a standard computational graph, and automatic differentiation can deal with it as usual.

In the case of recurrent neural networks, this is referred to as **backpropagation through time**.



Elman networks

Elman networks consist of ϕ and ψ defined as primitive neuron units, such as logistic regression units.

That is,

$$\mathbf{h}_t = \sigma_h (\mathbf{W}_{xh}^T \mathbf{x}_t + \mathbf{W}_{hh}^T \mathbf{h}_{t-1} + \mathbf{b}_h)$$

$$y_t = \sigma_y (\mathbf{W}_y^T \mathbf{h}_t + b_y)$$

$$\mathbf{W}_{xh}^T \in \mathbb{R}^{p \times q}, \mathbf{W}_{hh}^T \in \mathbb{R}^{q \times q}, \mathbf{b}_h \in \mathbb{R}^q, b_y \in \mathbb{R}, \mathbf{h}_0 = \mathbf{0}$$

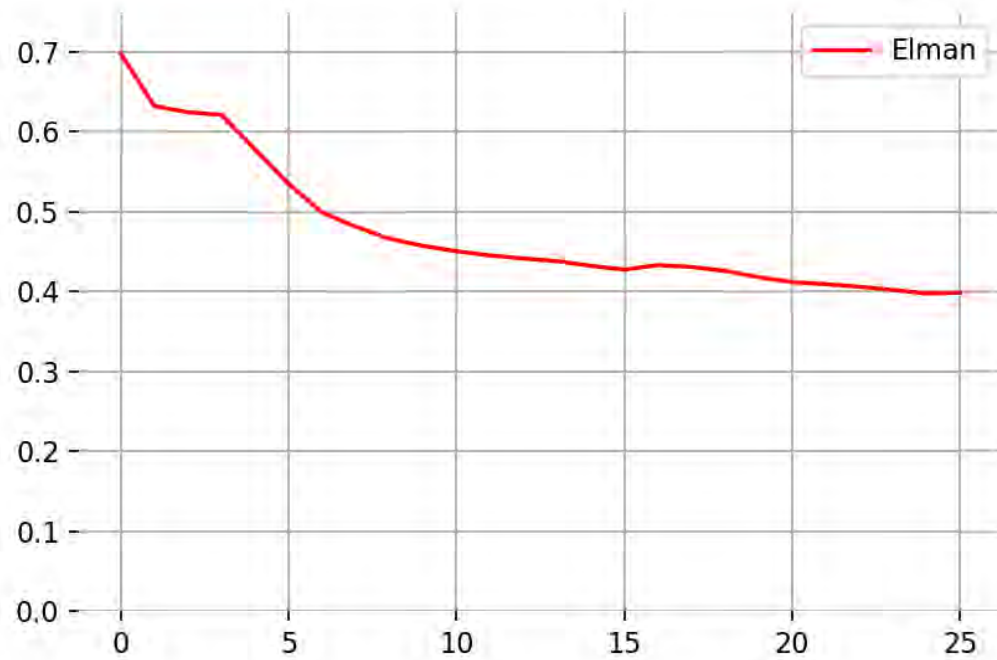
where σ_h and σ_y are non-linear activation functions, such as the sigmoid function, `tanh` or `ReLU`.

Example

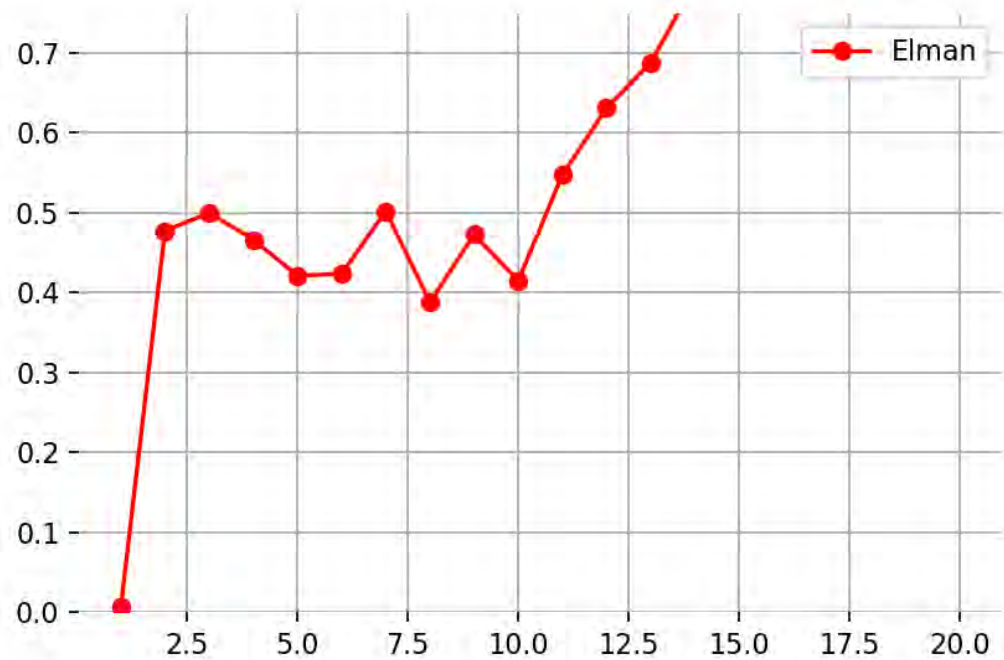
Can a recurrent network learn to tell whether a variable-length sequence is a **palindrome**?

\mathbf{x}	y
(1, 2, 3, 2, 1)	1
(2, 1, 2)	1
(3, 4, 1, 2)	0
(0)	1
(1, 4)	0

For training, we will use sequences of random sizes, from **1** to **10**.



Epoch vs. cross-entropy.



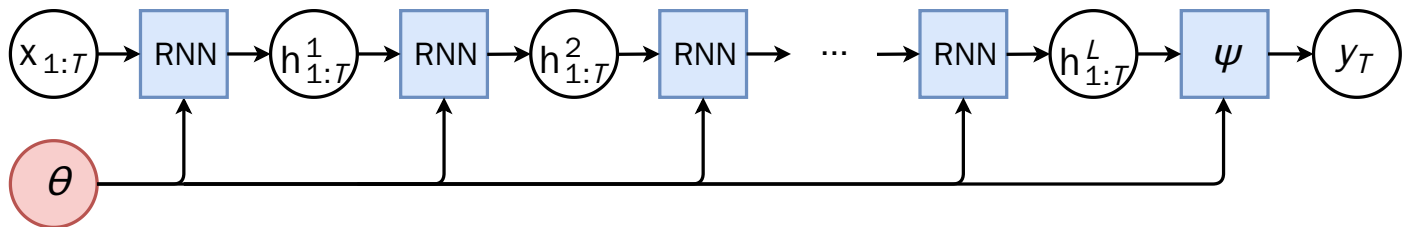
Sequence length vs. cross-entropy.

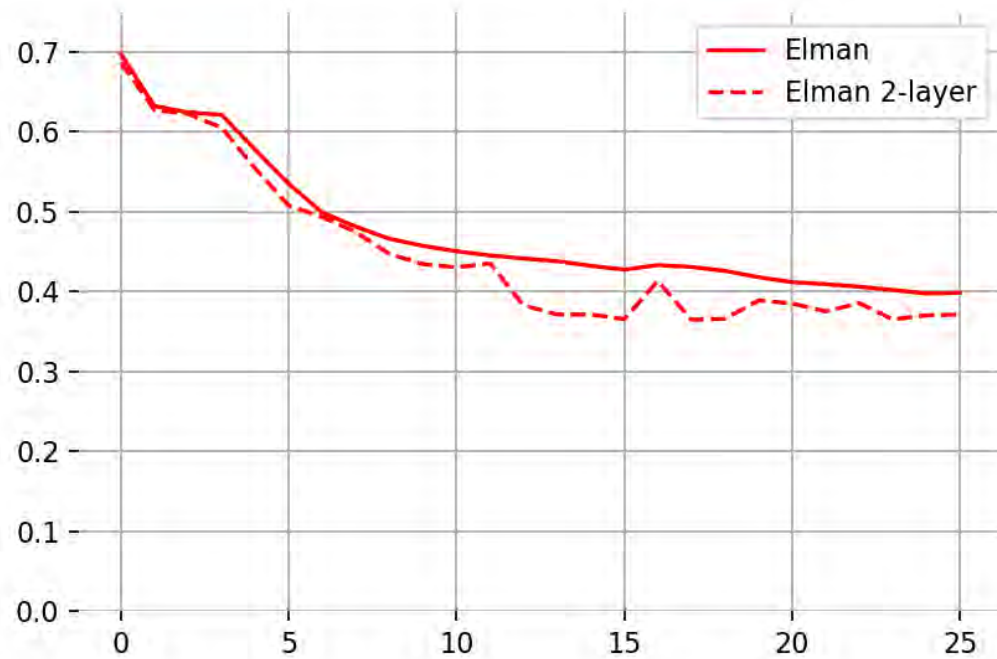
Note that the network was trained on sequences of size 10 or lower. It does not appear to generalize outside of the range.

Stacked RNNs

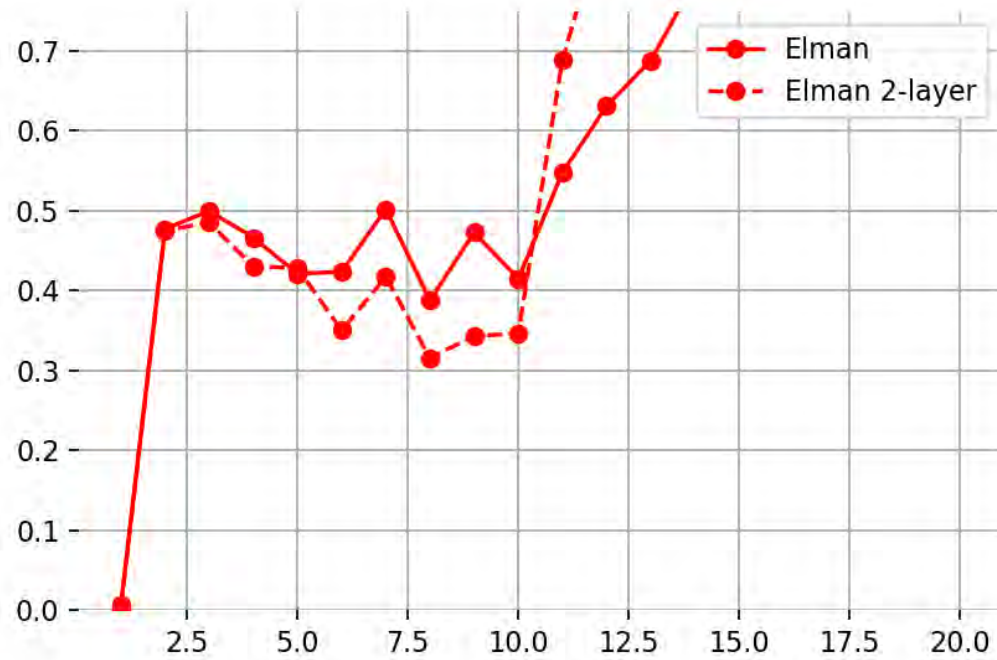
Recurrent networks can be viewed as layers producing sequences $\mathbf{h}_{1:T}^l$ of activations.

As for multi-perceptron layers, recurrent layers can be composed in series to form a **stack** of recurrent networks.





Epoch vs. cross-entropy.



Sequence length vs. cross-entropy.

Bidirectional RNNs

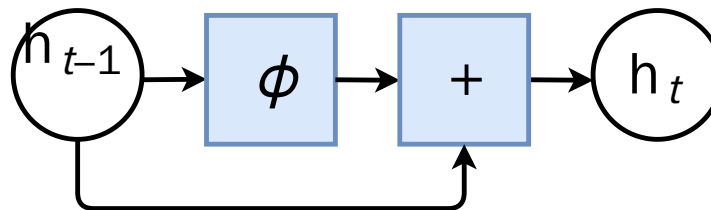
Computing the recurrent states forward in time does not make use of future input values $\mathbf{x}_{t+1:T}$, even though there are known.

- RNNs can be made **bidirectional** by consuming the sequence in both directions.
- Effectively, this amounts to run the same (single direction) RNN twice:
 - once over the original sequence $\mathbf{x}_{1:T}$,
 - once over the reversed sequence $\mathbf{x}_{T:1}$.
- The resulting recurrent states of the bidirectional RNN is the concatenation of two resulting sequences of recurrent states.

Gating

When unfolded through time, the resulting network can grow very deep, and training it involves dealing with **vanishing gradients**.

- A critical component in the design of RNN cells is to add in a **pass-through**, or additive paths, so that the recurrent state does not go repeatedly through a squashing non-linearity.
- This is very similar to skip connections in ResNets.



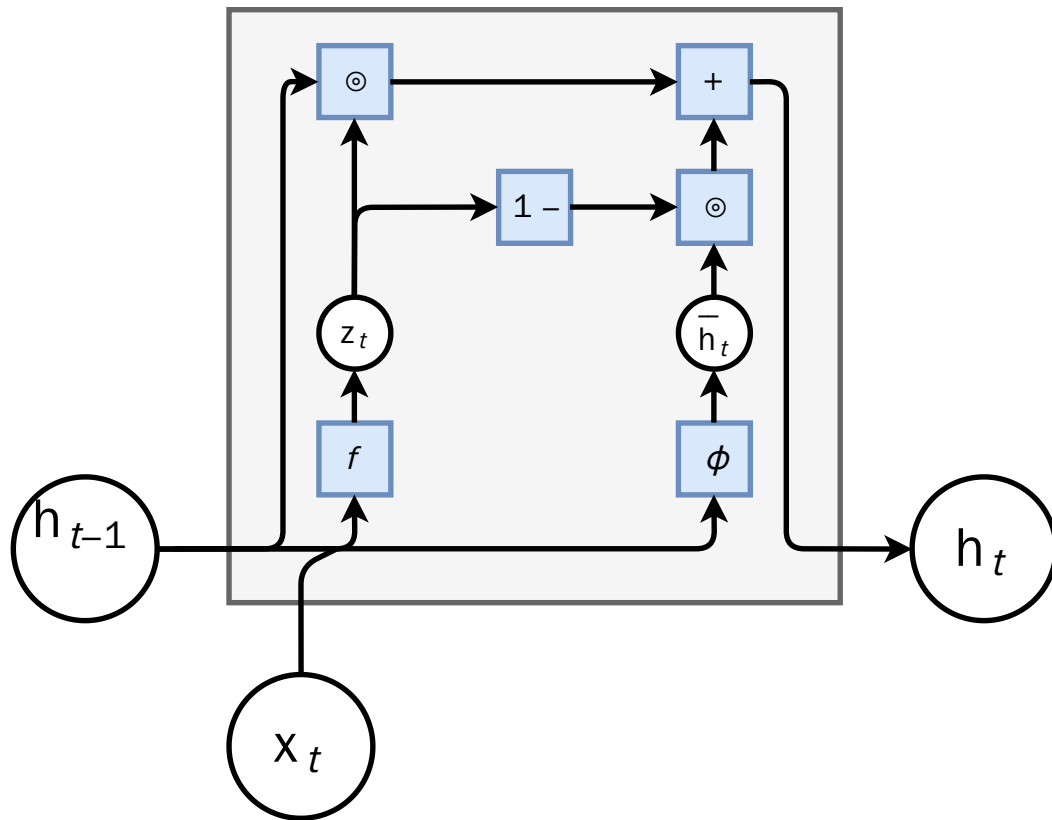
For instance, the recurrent state update can be a per-component weighted average of its previous value \mathbf{h}_{t-1} and a full update $\bar{\mathbf{h}}_t$, with the weighting \mathbf{z}_t depending on the input and the recurrent state, hence acting as a **forget gate**.

Formally,

$$\bar{\mathbf{h}}_t = \phi(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta)$$

$$\mathbf{z}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}; \theta)$$

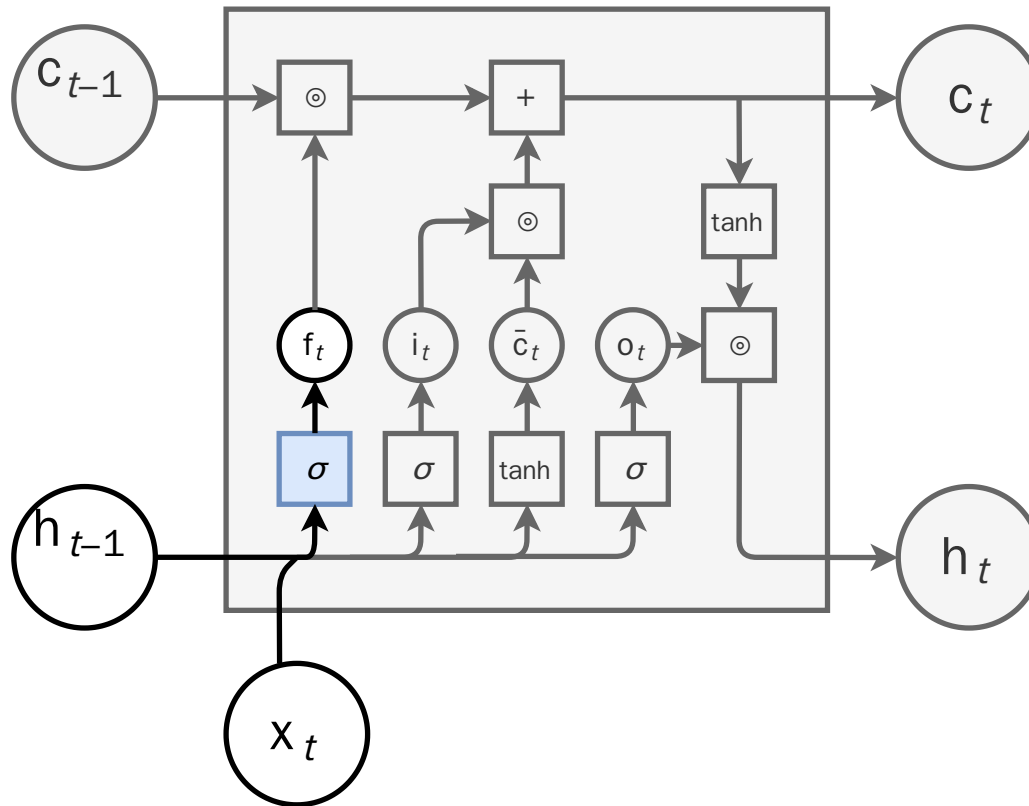
$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \bar{\mathbf{h}}_t.$$



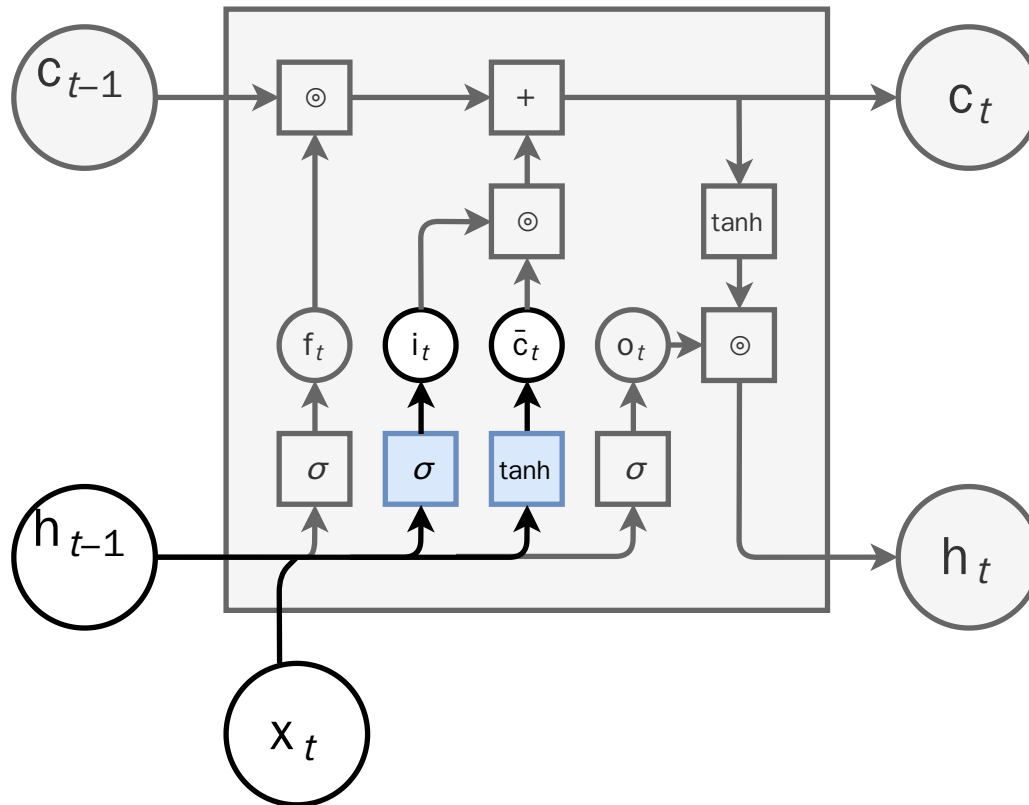
LSTM

The **long short-term memory model** (Hochreiter and Schmidhuber, 1997) is an instance of the previous gated recurrent cell, with the following changes:

- The recurrent state is split into two parts \mathbf{c}_t and \mathbf{h}_t , where
 - \mathbf{c}_t is the cell state and
 - \mathbf{h}_t is output state.
- A forget gate \mathbf{f}_t selects the cell state information to erase.
- An input gate \mathbf{i}_t selects the cell state information to update.
- An output gate \mathbf{o}_t selects the cell state information to output.

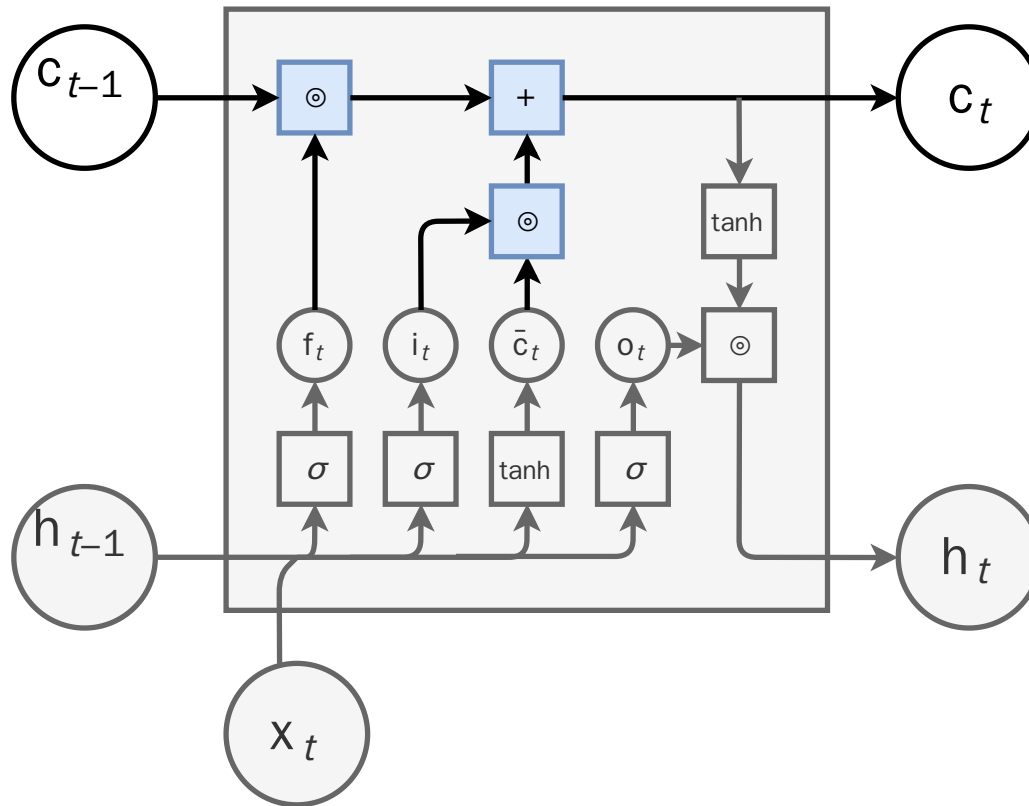


$$\mathbf{f}_t = \sigma(\mathbf{W}_f^T [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_f)$$

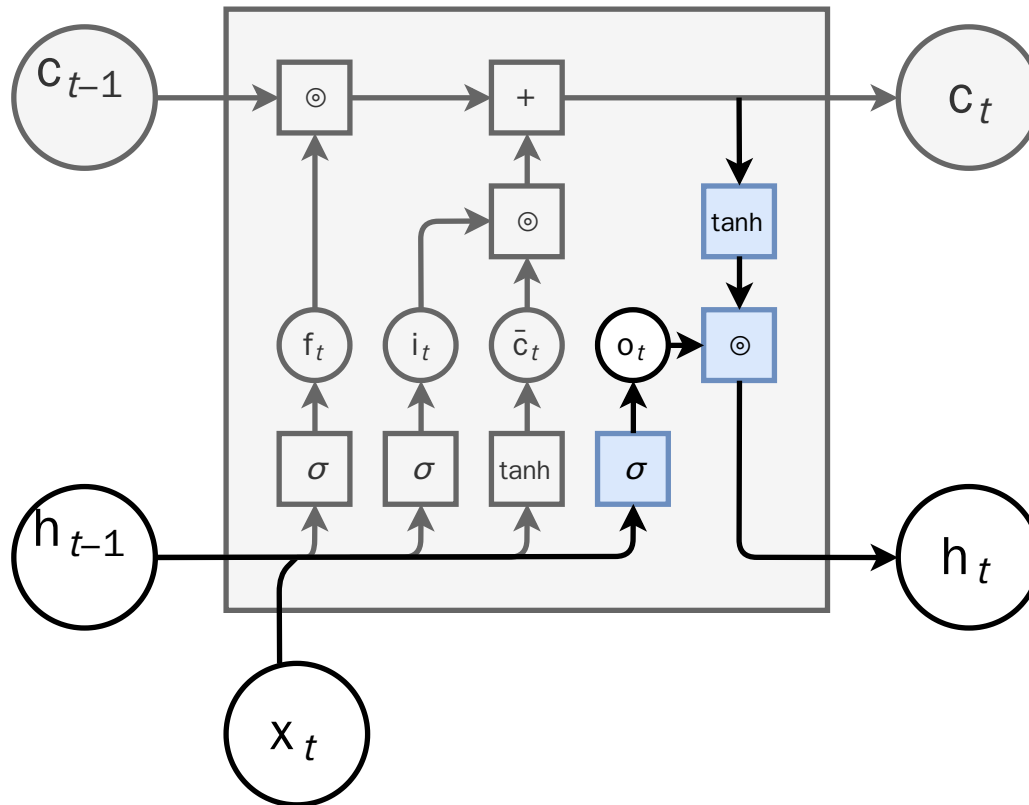


$$\mathbf{i}_t = \sigma(\mathbf{W}_i^T [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_i)$$

$$\bar{\mathbf{c}}_t = \tanh(\mathbf{W}_c^T [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_c)$$

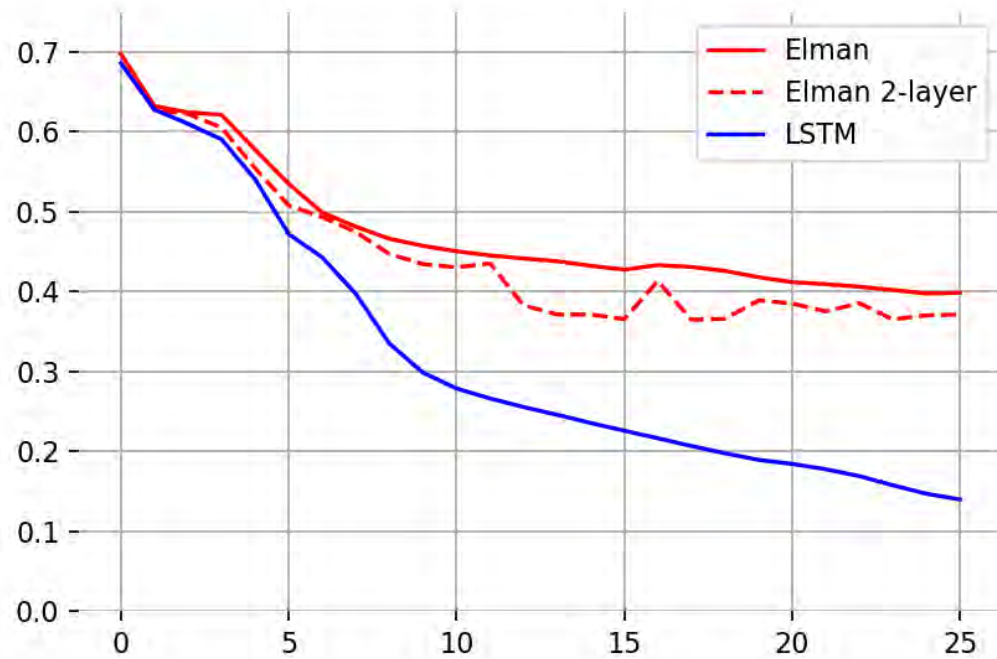


$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \bar{\mathbf{c}}_t$$

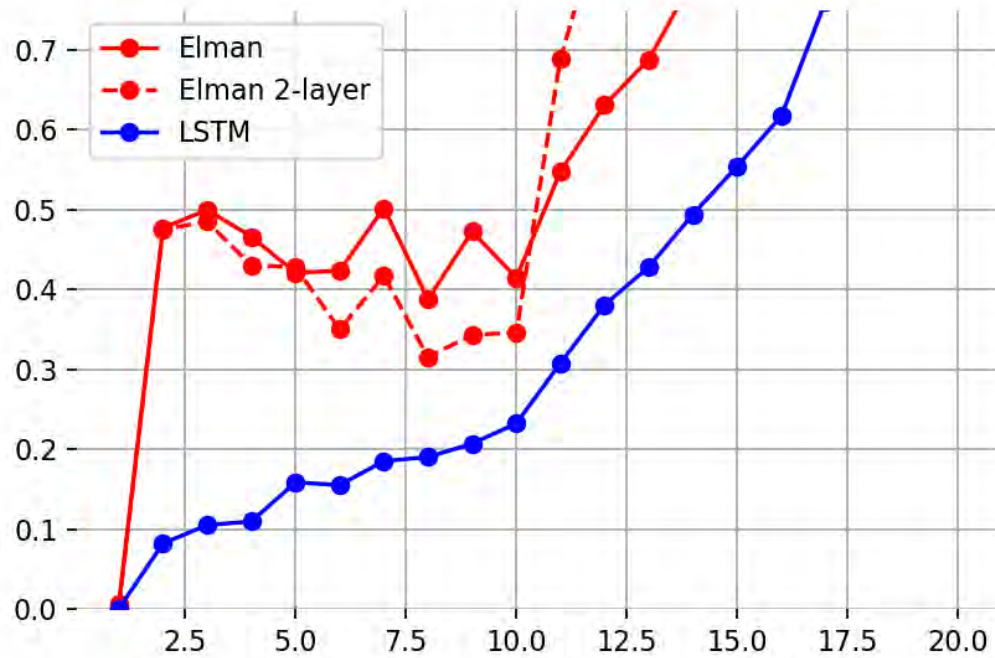


$$\mathbf{o}_t = \sigma(\mathbf{W}_o^T [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_o)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)$$



Epoch vs. cross-entropy.

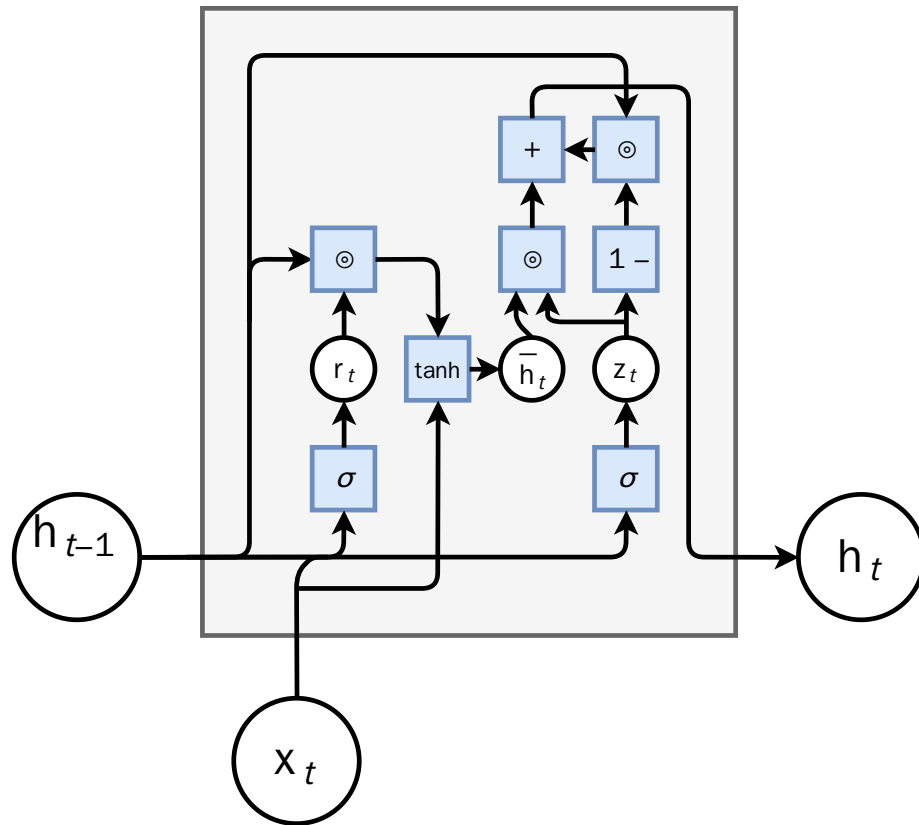


Sequence length vs. cross-entropy.

GRU

The **gated recurrent unit** (Cho et al, 2014) is another gated recurrent cell.

- It is based on two gates instead of three: an update gate \mathbf{z}_t and a reset gate \mathbf{r}_t .
- GRUs perform similarly as LSTMs for language or speech modeling sequences, but with fewer parameters.
- However, LSTMs remain strictly stronger than GRUs.

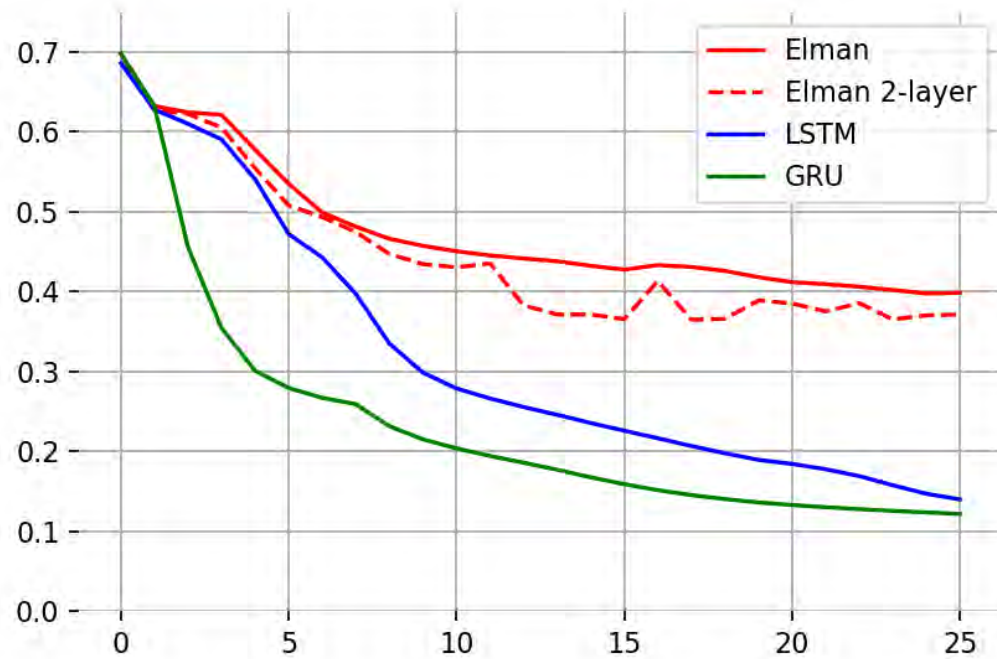


$$\mathbf{z}_t = \sigma (\mathbf{W}_z^T [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_z)$$

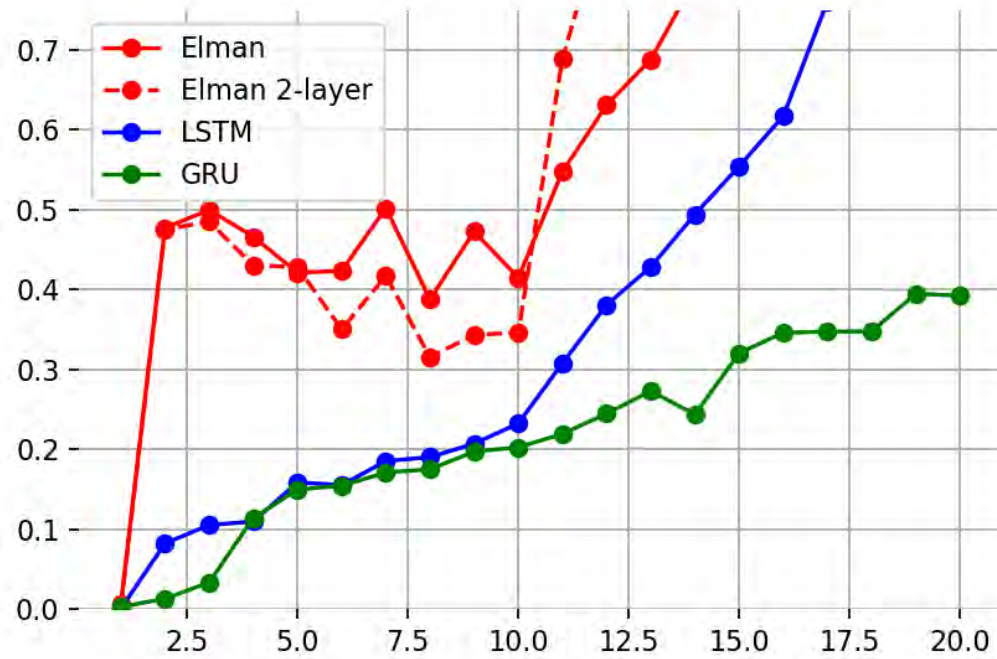
$$\mathbf{r}_t = \sigma (\mathbf{W}_r^T [\mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_r)$$

$$\bar{\mathbf{h}}_t = \tanh (\mathbf{W}_h^T [\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t] + \mathbf{b}_h)$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \bar{\mathbf{h}}_t$$



Epoch vs. cross-entropy.



Sequence length vs. cross-entropy.

Gradient clipping

Gated units prevent gradients from vanishing, but not from **exploding**.

The standard strategy to solve this issue is **gradient norm clipping**, which rescales the norm of the gradient to a fixed threshold δ when it is above:

$$\tilde{\nabla} f = \frac{\nabla f}{\|\nabla f\|} \min(\|\nabla f\|, \delta).$$

Orthogonal initialization

Let us consider a simplified RNN, with no inputs, no bias, an identity activation function σ (as in the positive part of a ReLU) and the initial recurrent state \mathbf{h}_0 set to the identity matrix.

We have,

$$\begin{aligned}\mathbf{h}_t &= \sigma(\mathbf{W}_{xh}^T \mathbf{x}_t + \mathbf{W}_{hh}^T \mathbf{h}_{t-1} + \mathbf{b}_h) \\ &= \mathbf{W}_{hh}^T \mathbf{h}_{t-1} \\ &= \mathbf{W}^T \mathbf{h}_{t-1}.\end{aligned}$$

For a sequence of size n , it comes

$$\mathbf{h}_n = \mathbf{W}(\mathbf{W}(\mathbf{W}(\dots(\mathbf{W}\mathbf{h}_0)\dots))) = \mathbf{W}^n \mathbf{h}_0 = \mathbf{W}^n I = \mathbf{W}^n.$$

Ideally, we would like \mathbf{W}^n to neither vanish nor explode as n increases.

Fibonacci digression

The Fibonacci sequence is

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, ...

It grows fast! But how fast?

In matrix form, the Fibonacci sequence is equivalently expressed as

$$\begin{pmatrix} f_{k+2} \\ f_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{k+1} \\ f_k \end{pmatrix}.$$

With $\mathbf{f}_0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, we have

$$\mathbf{f}_{k+1} = \mathbf{A}\mathbf{f}_k = \mathbf{A}^{k+1}\mathbf{f}_0.$$

The matrix \mathbf{A} can be diagonalized as

$$\mathbf{A} = \mathbf{S}\mathbf{\Lambda}\mathbf{S}^{-1},$$

where

$$\mathbf{\Lambda} = \begin{pmatrix} \varphi & 0 \\ 0 & -\varphi^{-1} \end{pmatrix}$$
$$\mathbf{S} = \begin{pmatrix} \varphi & -\varphi^{-1} \\ 1 & 1 \end{pmatrix}.$$

In particular,

$$\mathbf{A}^n = \mathbf{S}\mathbf{\Lambda}^n\mathbf{S}^{-1}.$$

Therefore, the Fibonacci sequence grows **exponentially fast** with the golden ratio φ .

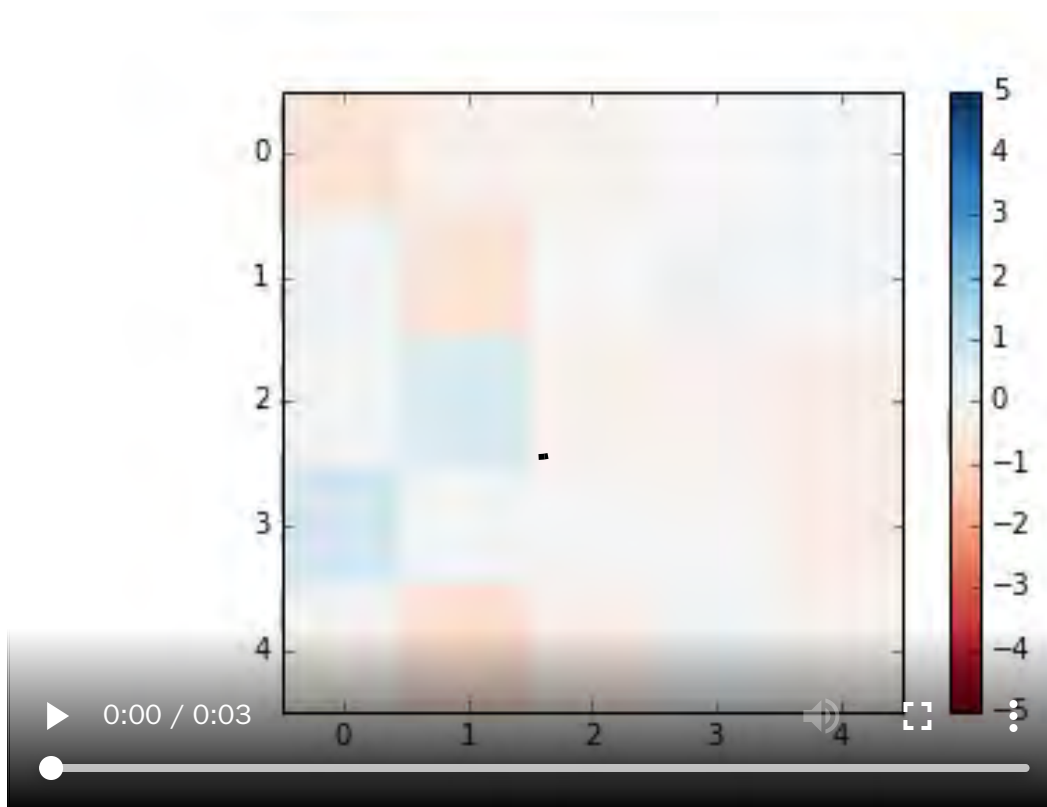
Theorem

Let $\rho(\mathbf{A})$ be the spectral radius of the matrix \mathbf{A} , defined as

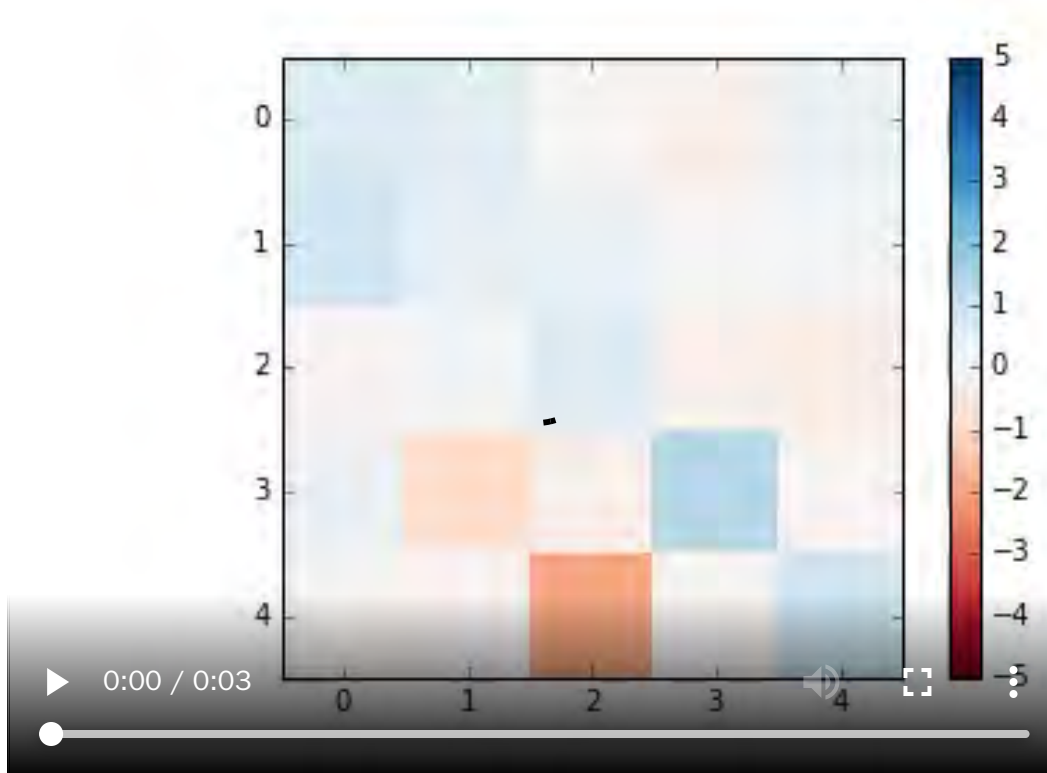
$$\rho(\mathbf{A}) = \max\{|\lambda_1|, \dots, |\lambda_d|\}.$$

We have:

- if $\rho(\mathbf{A}) < 1$ then $\lim_{n \rightarrow \infty} \|\mathbf{A}^n\| = \mathbf{0}$ (= vanishing activations),
- if $\rho(\mathbf{A}) > 1$ then $\lim_{n \rightarrow \infty} \|\mathbf{A}^n\| = \infty$ (= exploding activations).



$\rho(\mathbf{A}) < 1, \mathbf{A}^n$ vanish.



$\rho(\mathbf{A}) > 1, \mathbf{A}^n$ explode.

Orthogonal initialization

If \mathbf{A} is orthogonal, then it is diagonalizable and all its eigenvalues are equal to -1 or 1 . In this case, the norm of

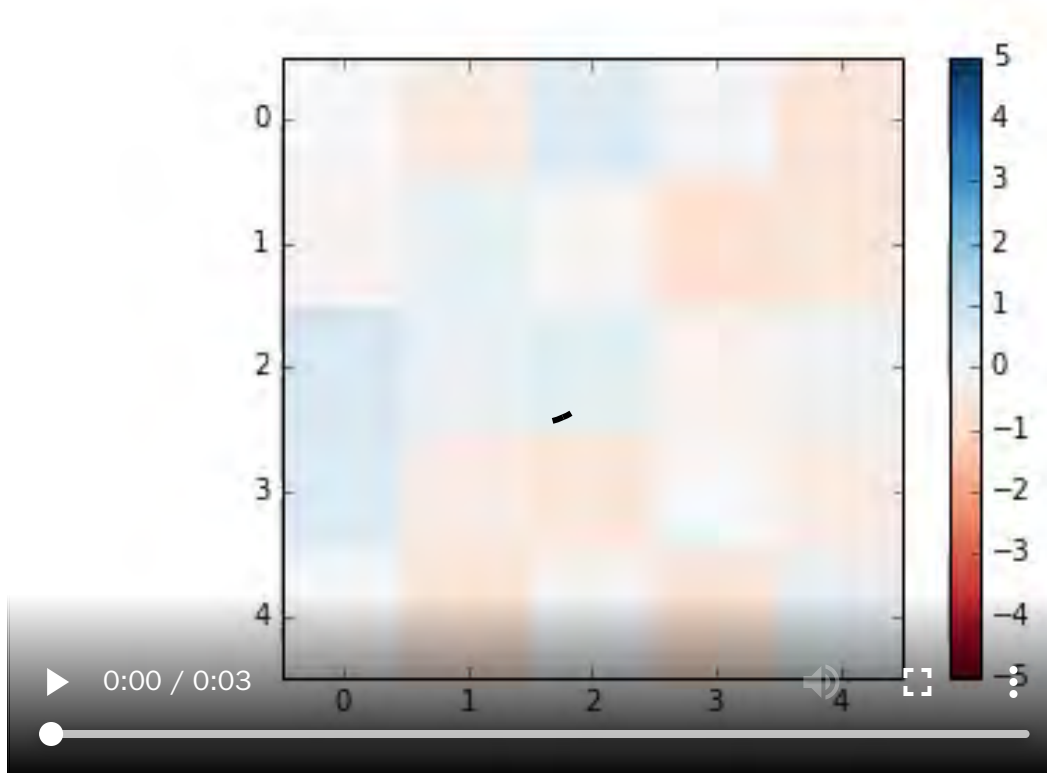
$$\mathbf{A}^n = \mathbf{S}\mathbf{\Lambda}^n\mathbf{S}^{-1}$$

remains bounded.

- Therefore, initializing \mathbf{W} as a random orthogonal matrix will guarantee that activations will neither vanish nor explode.
- In practice, a random orthogonal matrix can be found through the SVD decomposition or the QR factorization of a random matrix.
- This initialization strategy is known as **orthogonal initialization**.

In Tensorflow's Orthogonal initializer:

```
# Generate a random matrix
a = random_ops.random_normal(flat_shape, dtype=dtype, seed=self.seed)
# Compute the qr factorization
q, r = gen_linalg_ops.qr(a, full_matrices=False)
# Make Q uniform
d = array_ops.diag_part(r)
q *= math_ops.sign(d)
if num_rows < num_cols:
    q = array_ops.matrix_transpose(q)
return self.gain * array_ops.reshape(q, shape)
```



A is orthogonal.

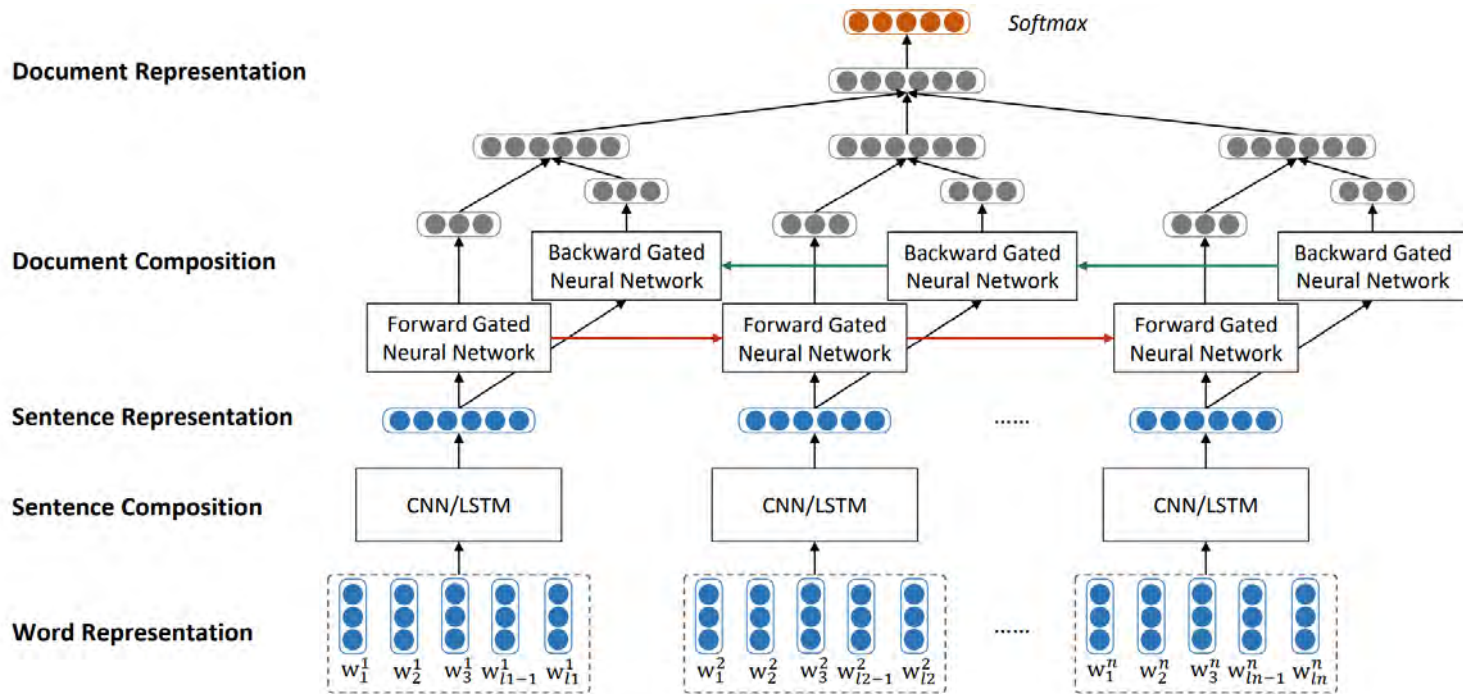
Finally, let us note that exploding activations are also the reason why squashing non-linearity functions (such as `tanh`) are preferred in RNNs.

- They avoid recurrent states from exploding by upper bounding $\|\mathbf{h}_t\|$.
- (At least when running the network forward.)

Applications

(some)

Sentiment analysis



Document-level modeling for sentiment analysis (= text classification), with stacked, bidirectional and gated recurrent networks.

Language models

Model language as a Markov chain, such that sentences are sequences of words $\mathbf{w}_{1:T}$ drawn repeatedly from

$$p(\mathbf{w}_t | \mathbf{w}_{1:t-1}).$$

This is an instance of sequence synthesis, for which predictions are computed at all time steps t .

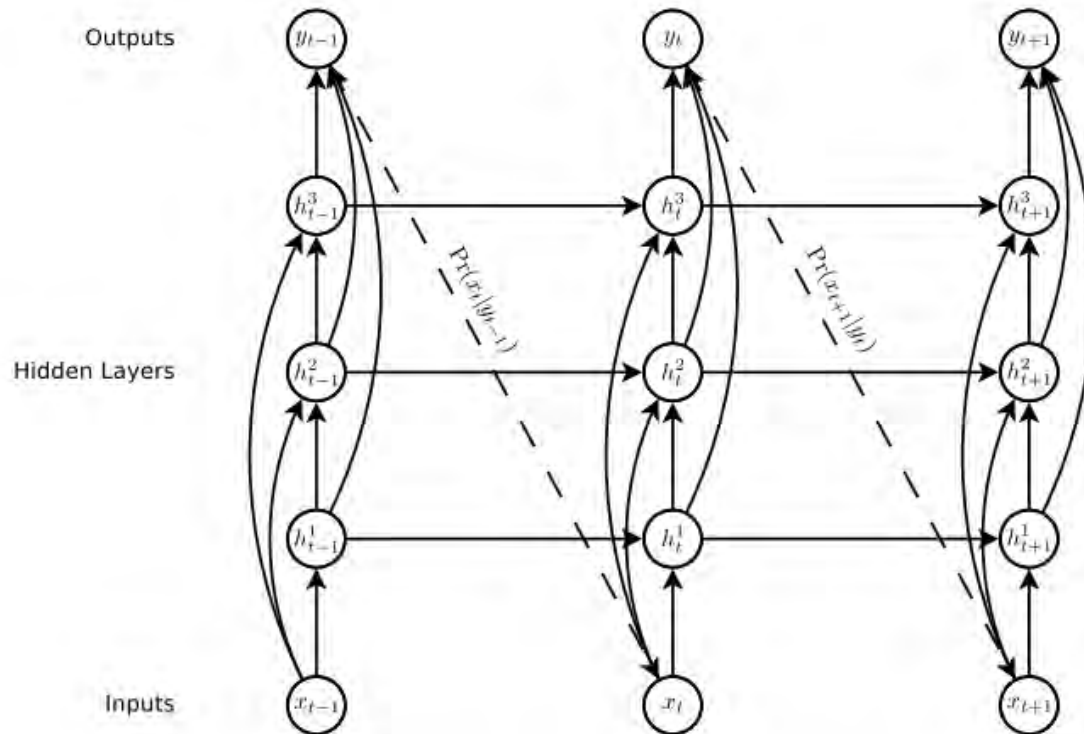


Figure 1: Deep recurrent neural network prediction architecture. The circles represent network layers, the solid lines represent weighted connections and the dashed lines represent predictions.

```
[maxs-mbp:tweet-generator maxwoolf$ python3  
Python 3.6.4 (default, Jan 6 2018, 11:51:59)  
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.39.2)] on darwin  
Type "help", "copyright", "credits" or "license" for more information.  
>>> █
```

[Open](#) in Google Colab.

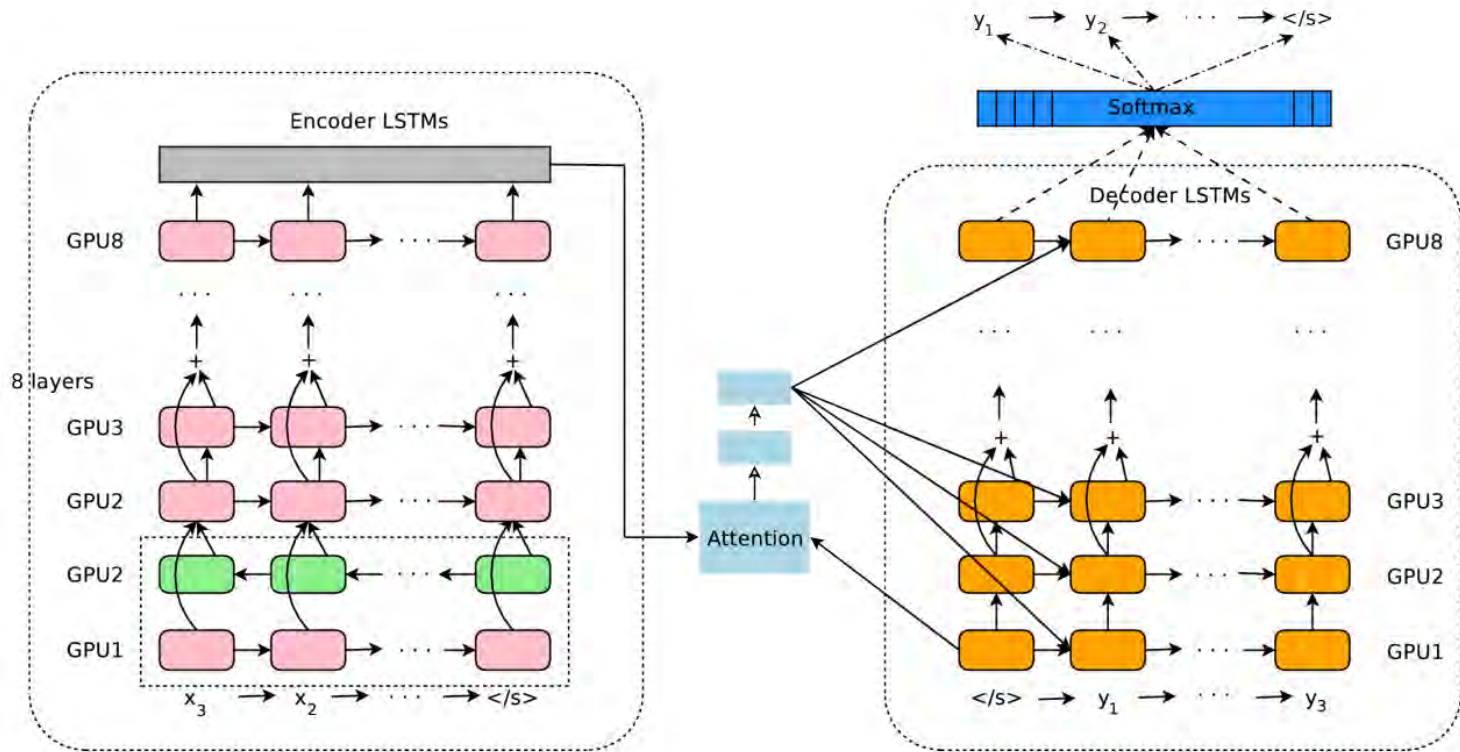
The same generative architecture applies to any kind of sequences.

Say, sketches defined as sequences of strokes?

`sketch-rnn-demo`



Neural machine translation



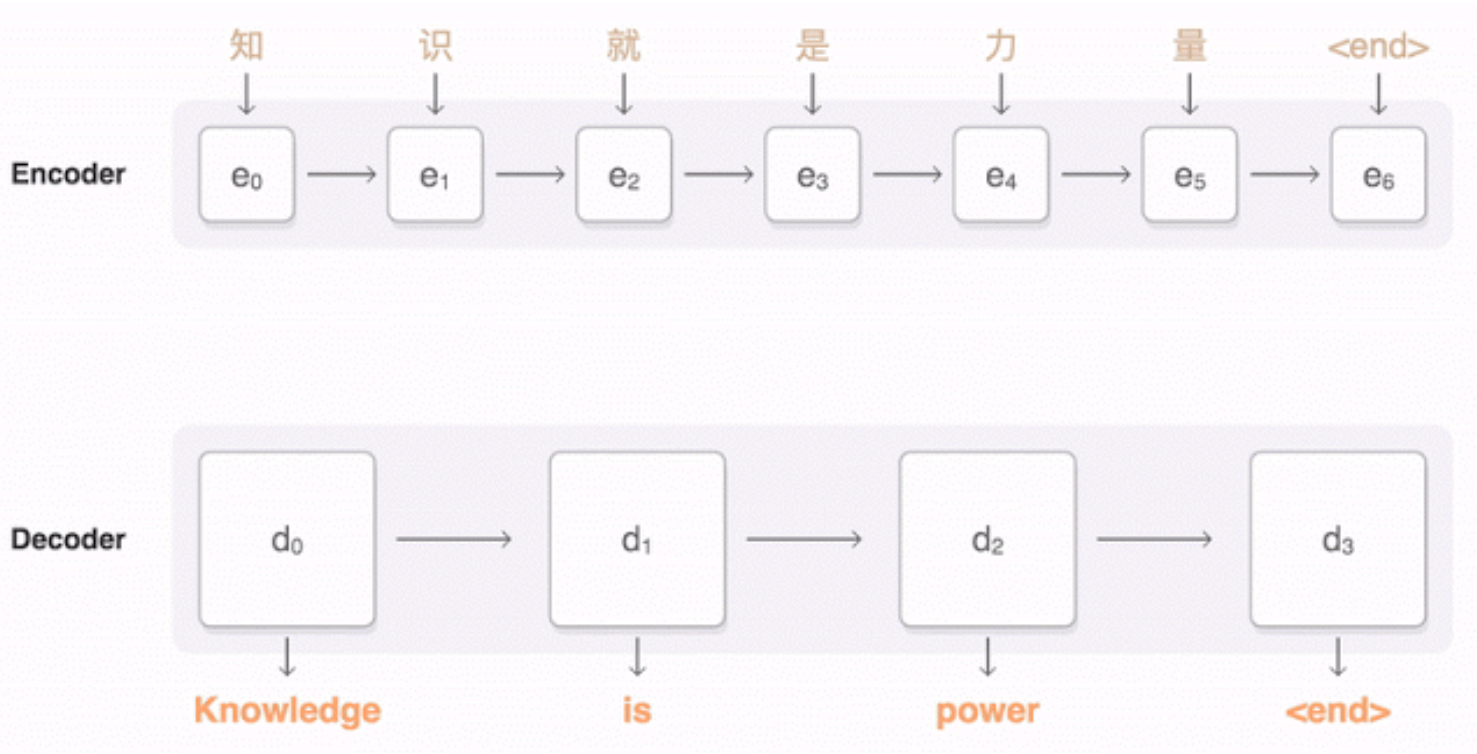
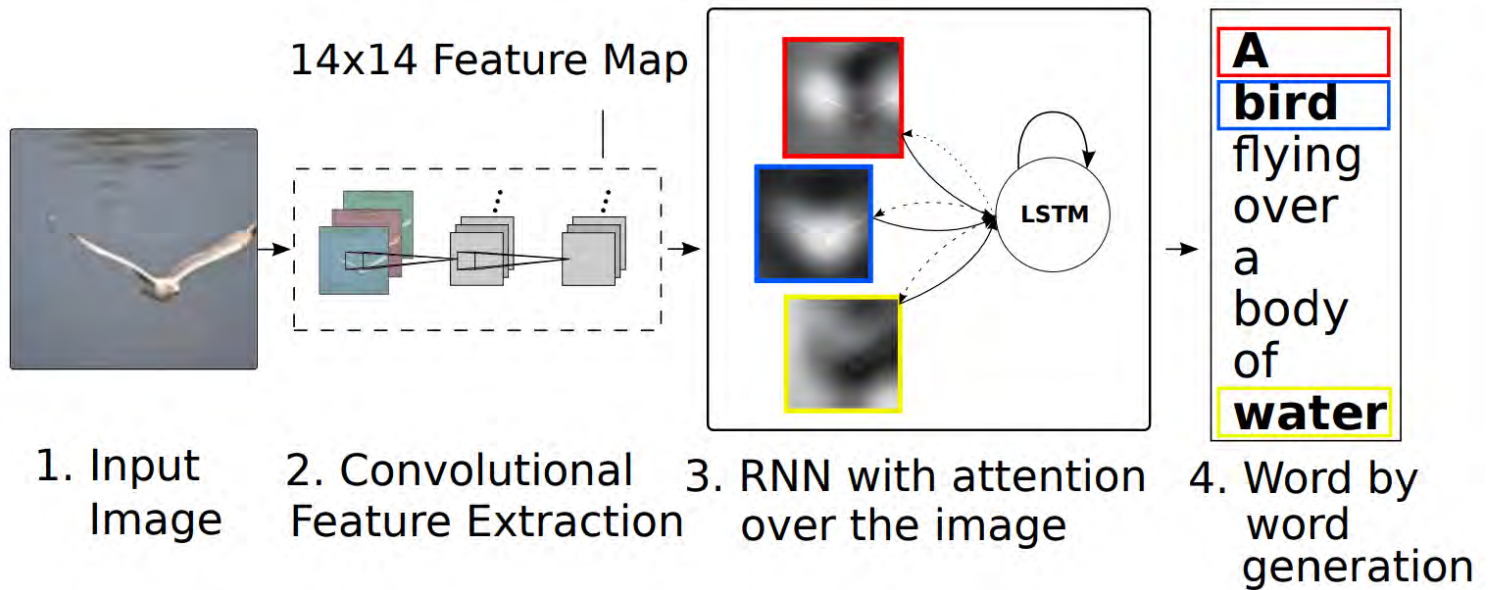
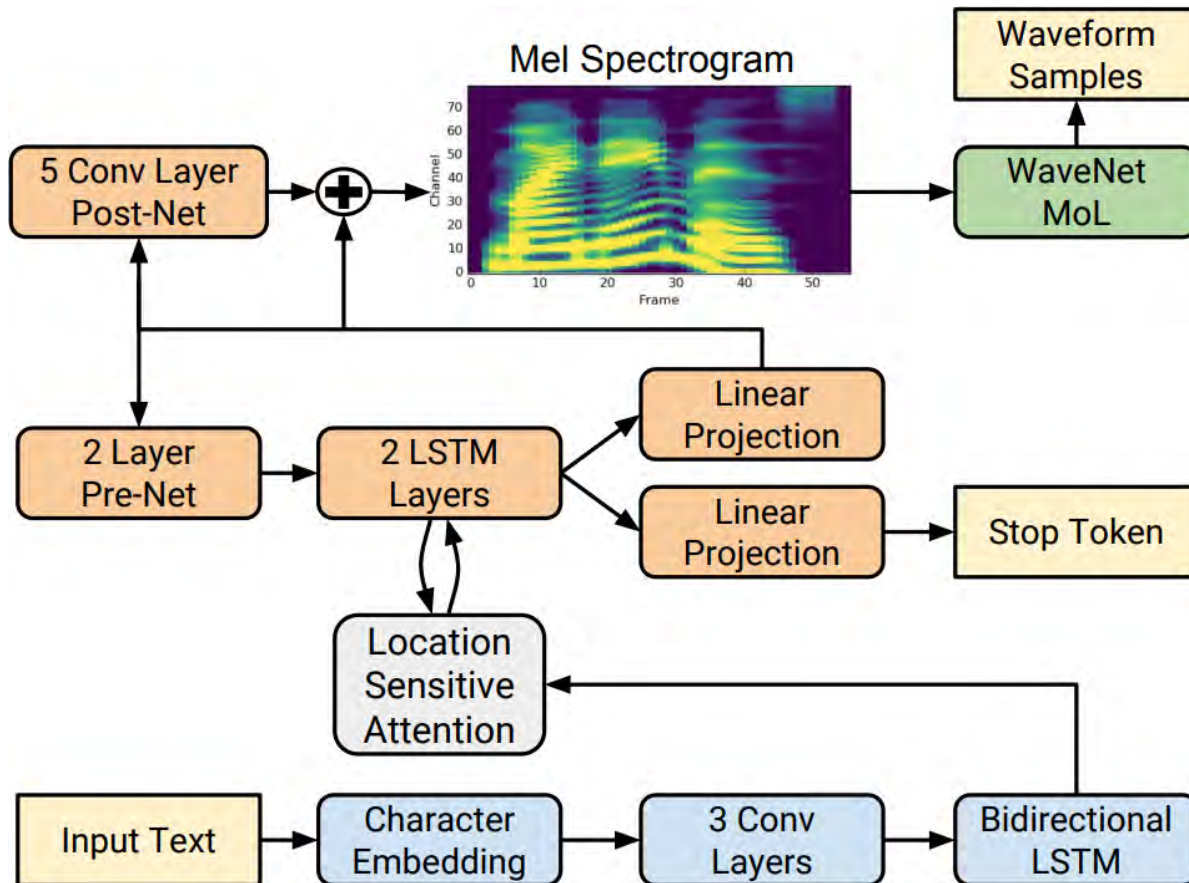


Image captioning





Text-to-speech synthesis





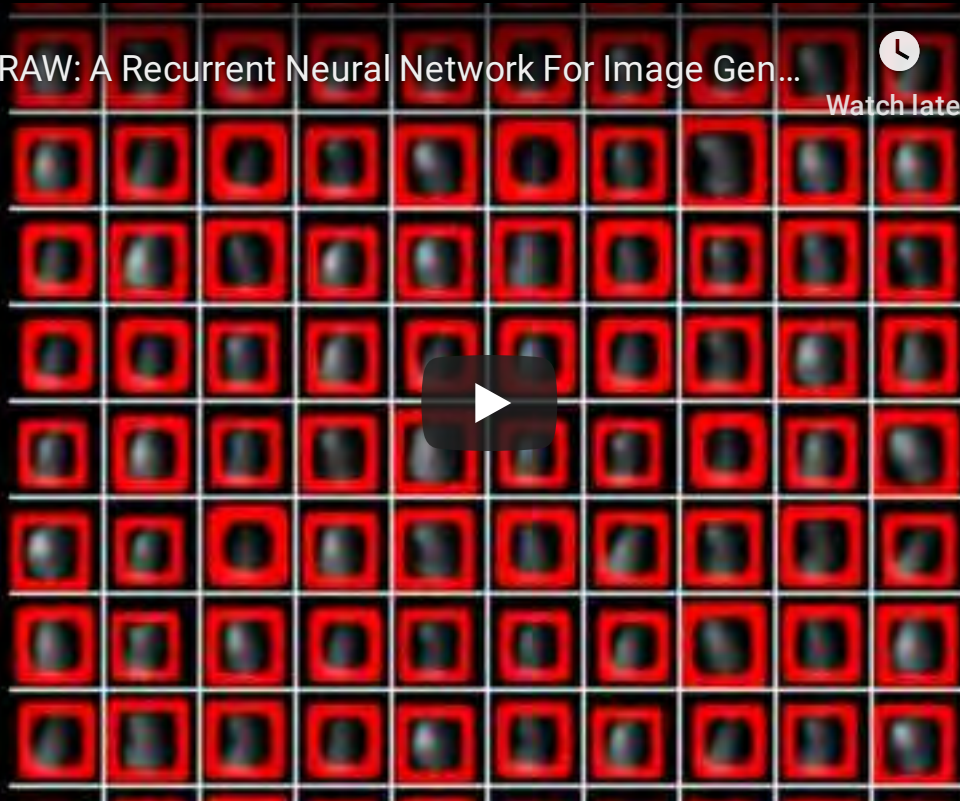
DRAW: A Recurrent Neural Network For Image Gen...



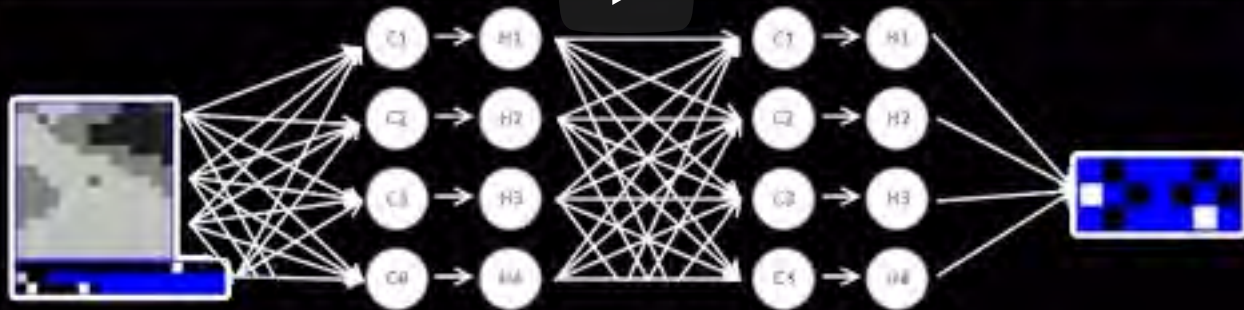
Watch later



Share



DRAW: A Recurrent Neural Network For Image Generation



A recurrent network playing Mario Kart.

Differentiable computers



People are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization.

*An increasingly large number of **people are defining the networks procedurally in a data-dependent way (with loops and conditionals)**, allowing them to change dynamically as a function of the input data fed to them. It's really **very much like a regular program, except it's parameterized.***

Yann LeCun (Director of AI Research, Facebook, 2018)

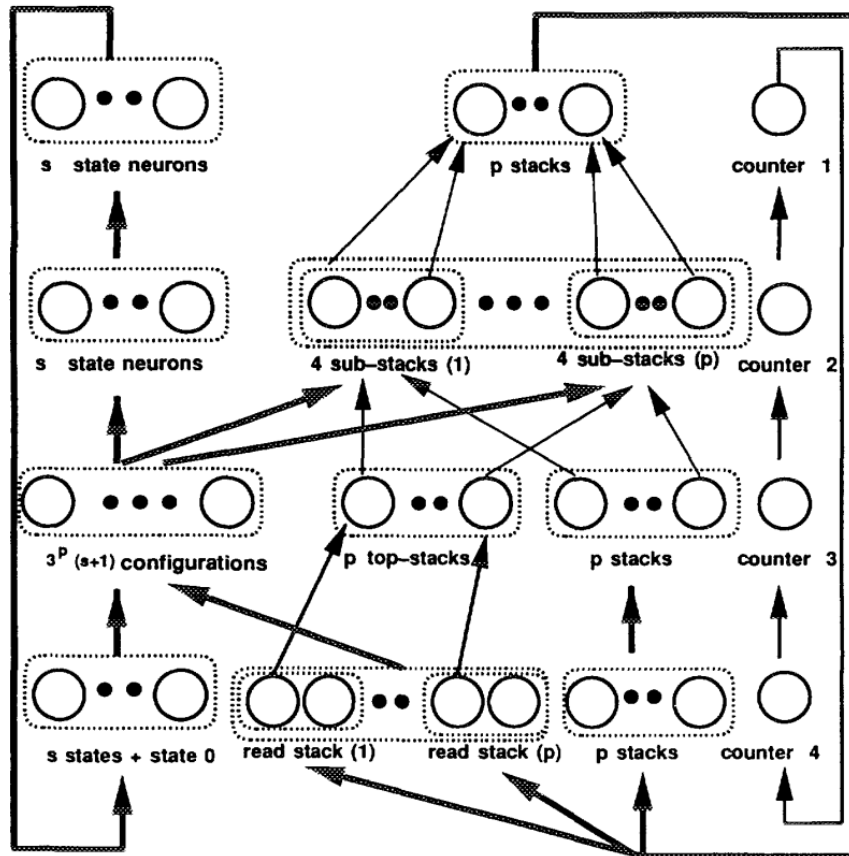
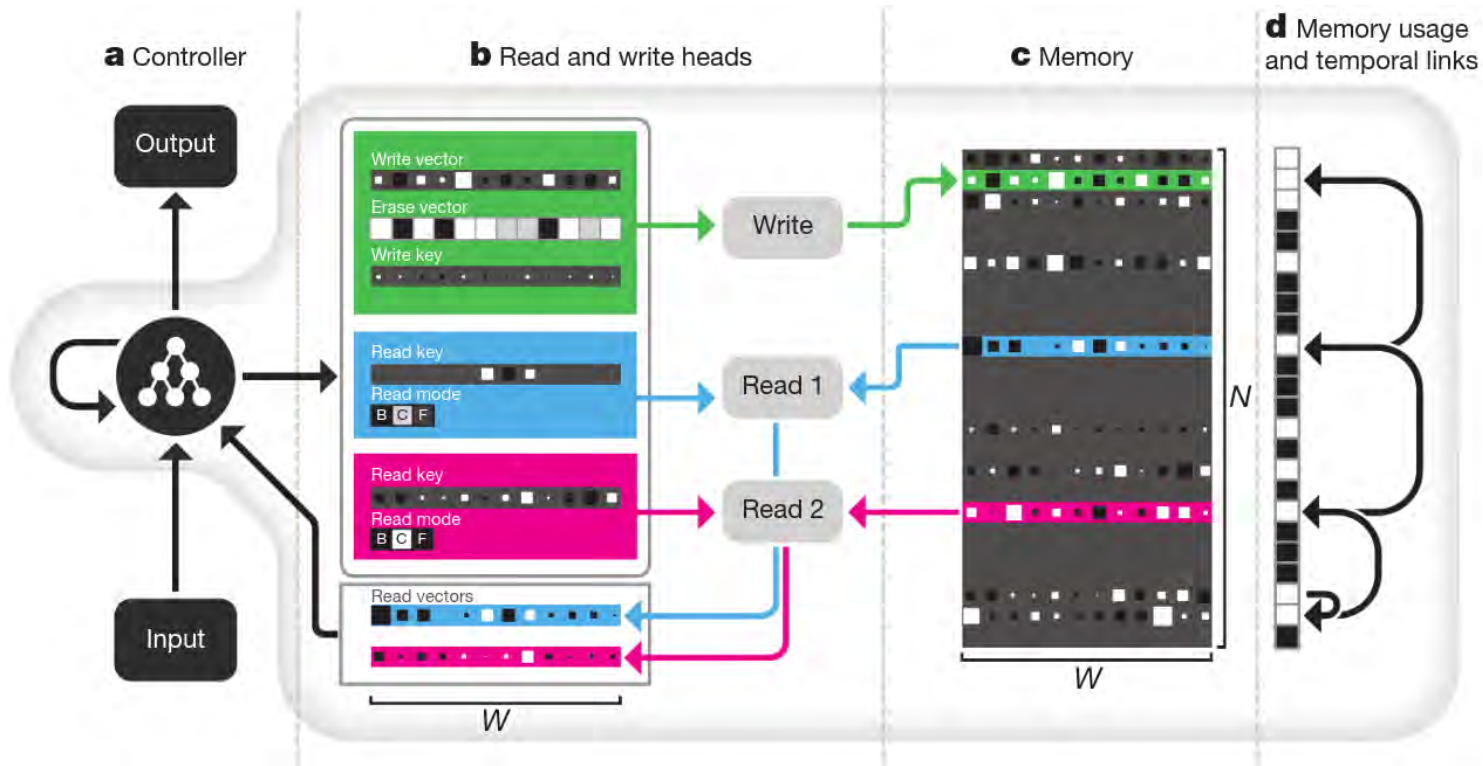
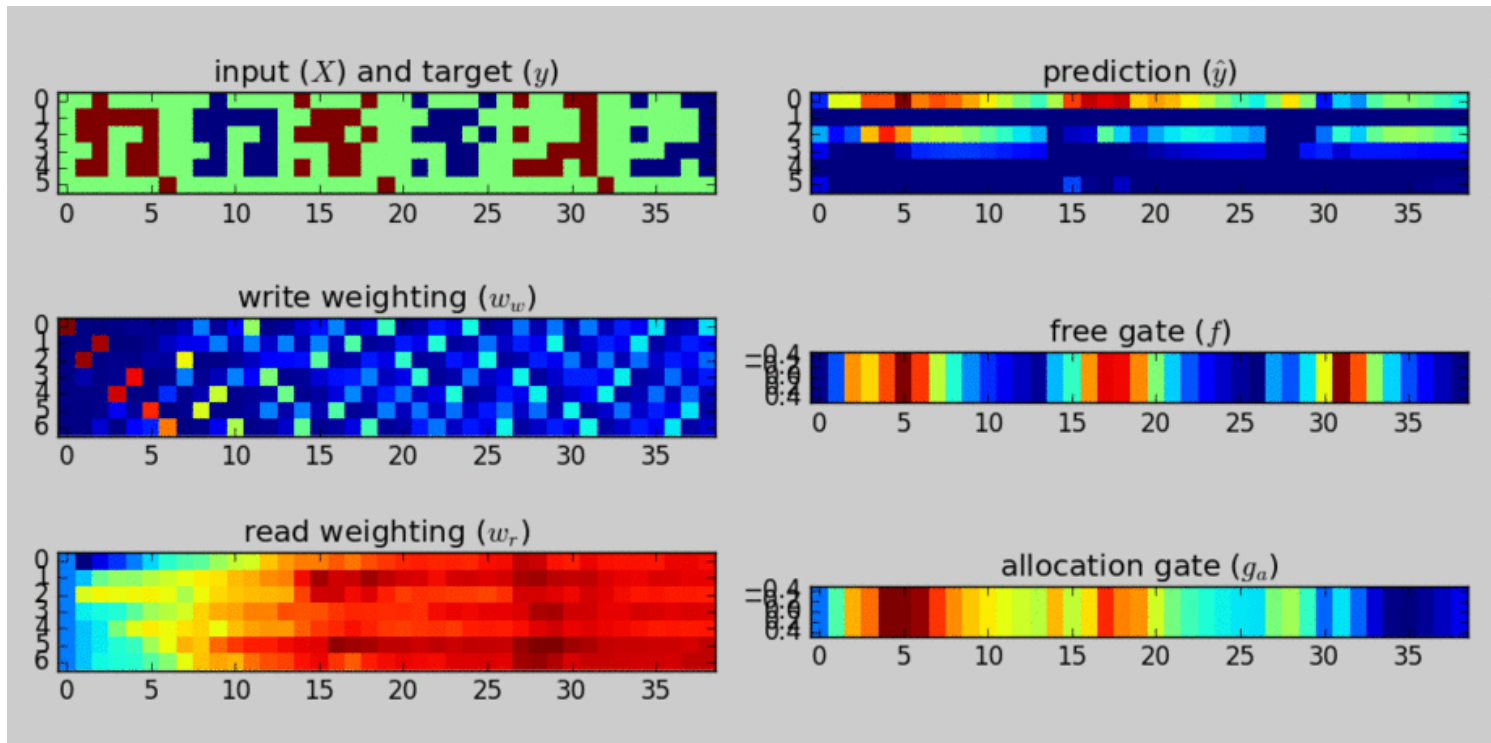


FIG. 1. The universal network.

Any Turing machine can be simulated by a recurrent neural network
(Siegelmann and Sontag, 1995)



Differentiable Neural Computer (Graves et al, 2016)



A differentiable neural computer being trained to store and recall dense binary numbers. Upper left: the input (red) and target (blue), as 5-bit words and a 1 bit interrupt signal. Upper right: the model's output

The end.

References

- Kyunghyun Cho, "Natural Language Understanding with Distributed Representation", 2015.

Deep Learning

Lecture 6: Auto-encoders and generative models

Prof. Gilles Louppe
g.louppe@uliege.be

Today

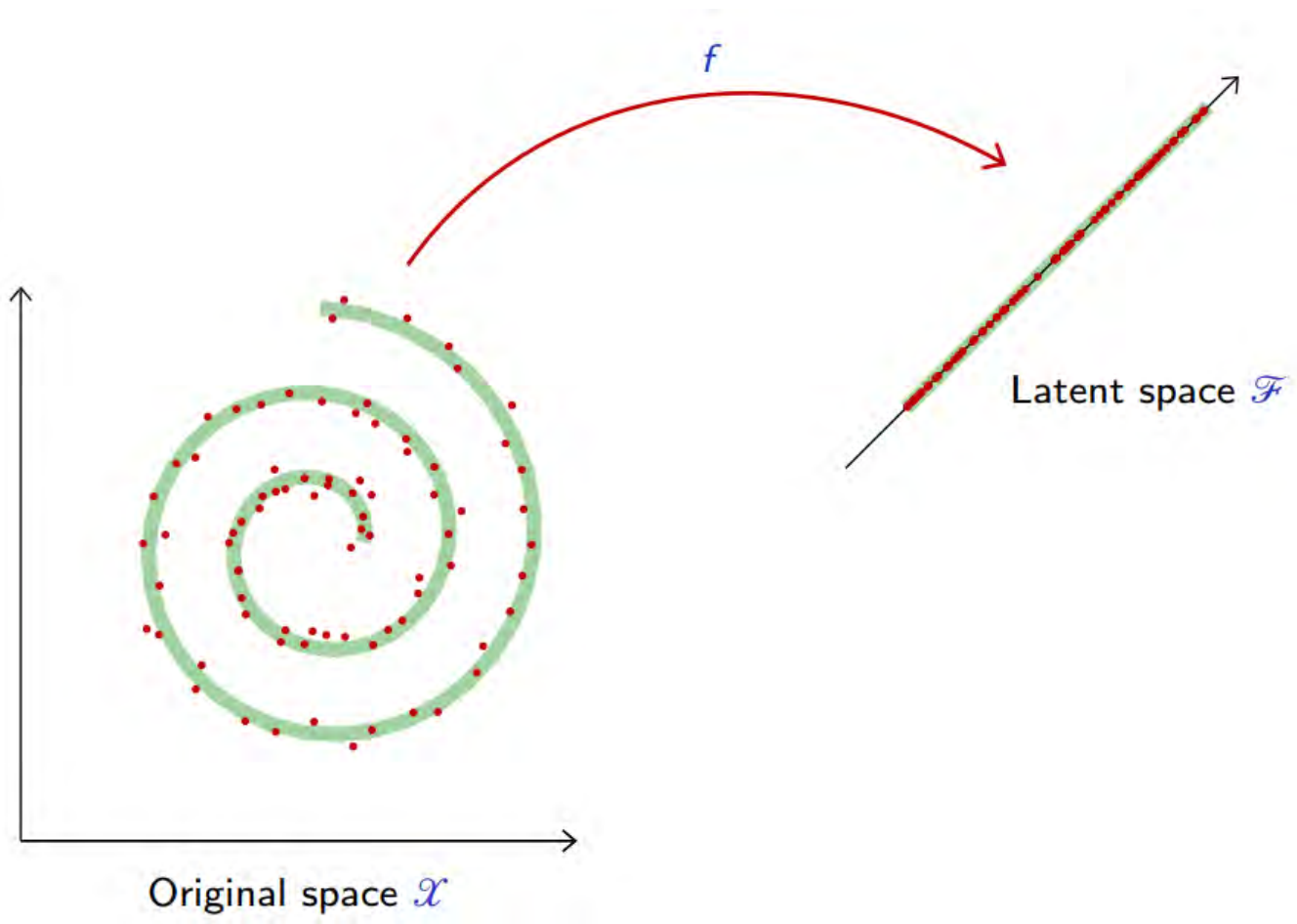
Learn a model of the data.

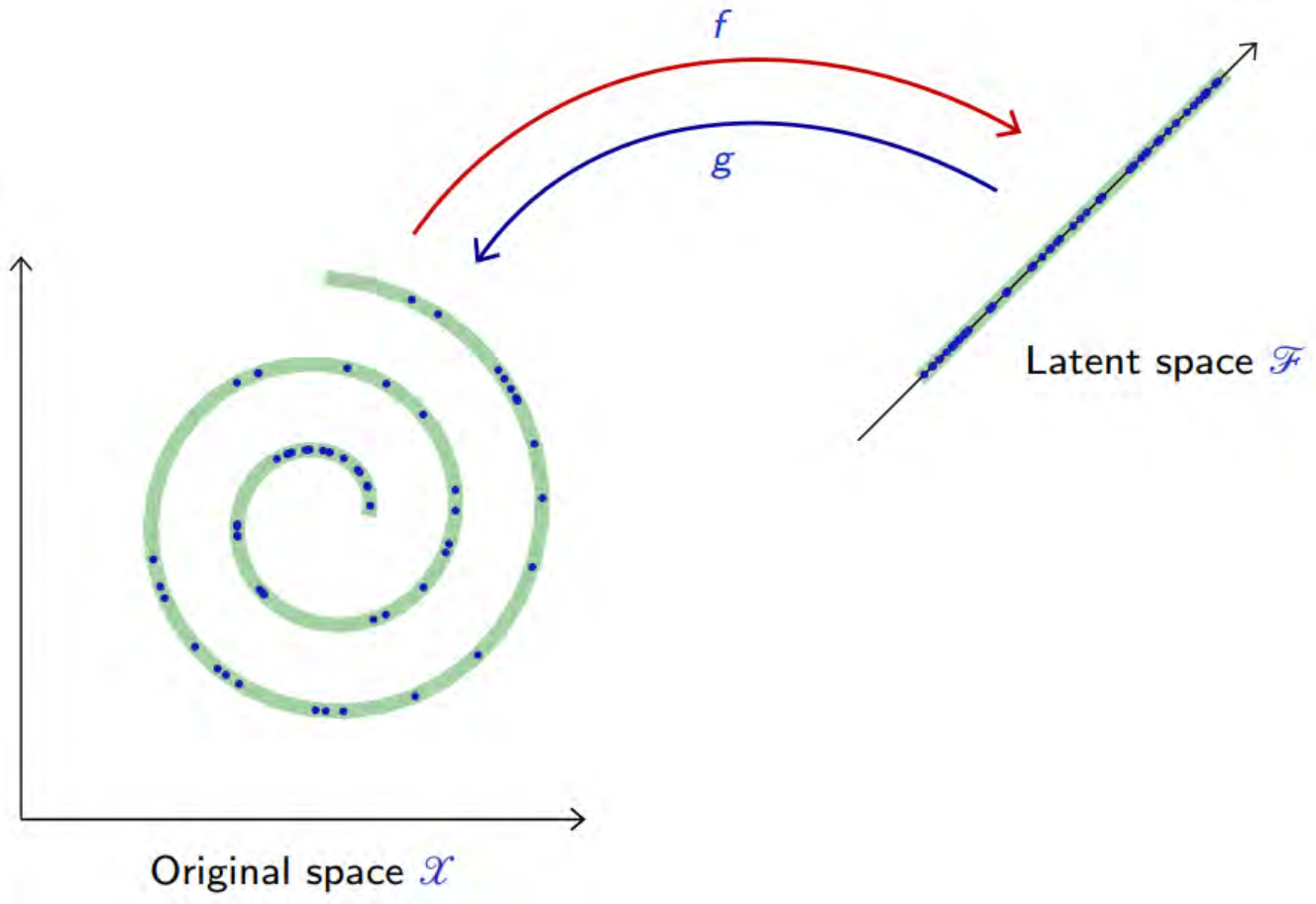
- Auto-encoders
- Generative models
- Variational inference
- Variational auto-encoders

Auto-encoders

Many applications such as image synthesis, denoising, super-resolution, speech synthesis or compression, require to [go beyond classification and regression](#) and model explicitly a high-dimensional signal.

This modeling consists of finding "*meaningful degrees of freedom*", or "*factors of variations*", that describe the signal and are of lesser dimension.



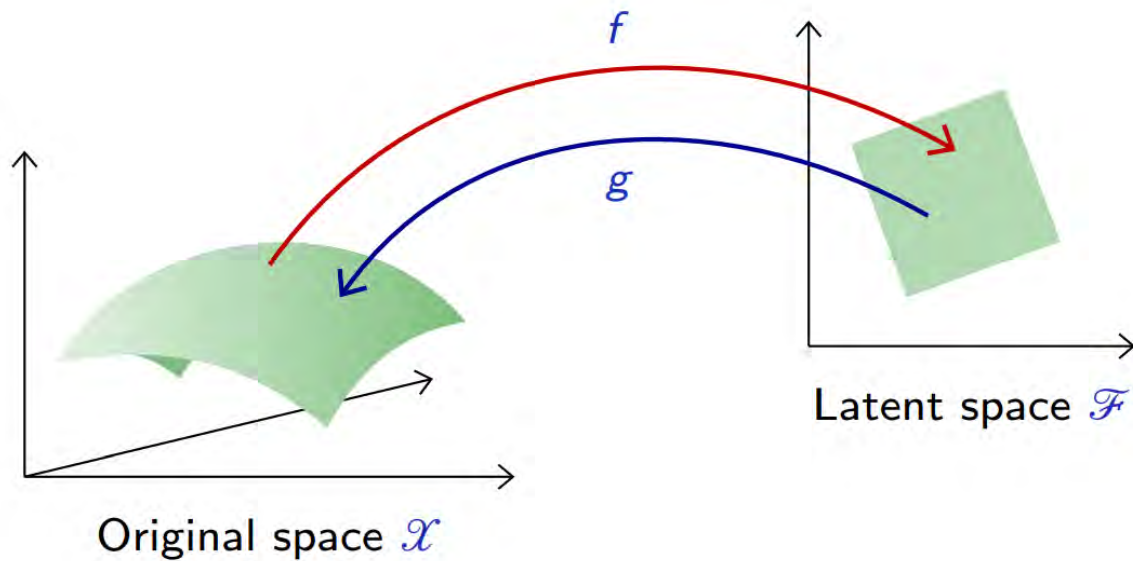


Auto-encoders

An auto-encoder is a composite function made of

- an **encoder** f from the original space \mathcal{X} to a latent space \mathcal{Z} ,
- a **decoder** g to map back to \mathcal{X} ,

such that $g \circ f$ is close to the identity on the data.



A proper auto-encoder should capture a good parameterization of the signal, and in particular the statistical dependencies between the signal components.

Let $p(\mathbf{x})$ be the data distribution over \mathcal{X} . A good auto-encoder could be characterized with the reconstruction loss

$$\mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} [\|\mathbf{x} - g \circ f(\mathbf{x})\|^2] \approx 0.$$

Given two parameterized mappings $f(\cdot; \theta_f)$ and $g(\cdot; \theta_g)$, training consists of minimizing an empirical estimate of that loss,

$$\theta = \arg \min_{\theta_f, \theta_g} \frac{1}{N} \sum_{i=1}^N \|\mathbf{x}_i - g(f(\mathbf{x}_i, \theta_f), \theta_g)\|^2.$$

For example, when the auto-encoder is linear,

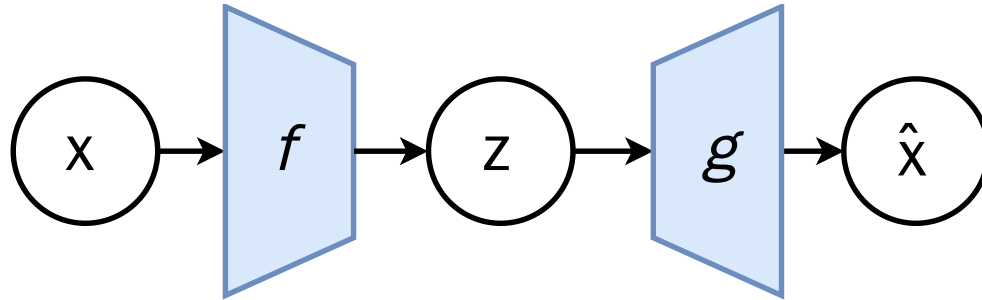
$$\begin{aligned}f : \mathbf{z} &= \mathbf{U}^T \mathbf{x} \\g : \hat{\mathbf{x}} &= \mathbf{U} \mathbf{z},\end{aligned}$$

with $\mathbf{U} \in \mathbb{R}^{p \times k}$, the reconstruction error reduces to

$$\mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} [\|\mathbf{x} - \mathbf{U} \mathbf{U}^T \mathbf{x}\|^2].$$

In this case, an optimal solution is given by PCA.

Deep auto-encoders



Better results can be achieved with more sophisticated classes of mappings than linear projections, in particular by designing f and g as deep neural networks.

For instance,

- by combining a multi-layer perceptron encoder $f : \mathbb{R}^p \rightarrow \mathbb{R}^q$ with a multi-layer perceptron decoder $g : \mathbb{R}^q \rightarrow \mathbb{R}^p$.
- by combining a convolutional network encoder $f : \mathbb{R}^{w \times h \times c} \rightarrow \mathbb{R}^q$ with a decoder $g : \mathbb{R}^q \rightarrow \mathbb{R}^{w \times h \times c}$ composed of the reciprocal transposed convolutional layers.

Deep neural decoders require layers that increase the input dimension, i.e., that map $\mathbf{z} \in \mathbb{R}^q$ to $\hat{\mathbf{x}} = g(\mathbf{z}) \in \mathbb{R}^p$, with $p \gg q$.

- This is the opposite of what we did so far with feedforward networks, in which we reduced the dimension of the input to a few values.
- Fully connected layers could be used for that purpose but would face the same limitations as before (spatial specialization, too many parameters).
- Ideally, we would like layers that implement the inverse of convolutional and pooling layers.

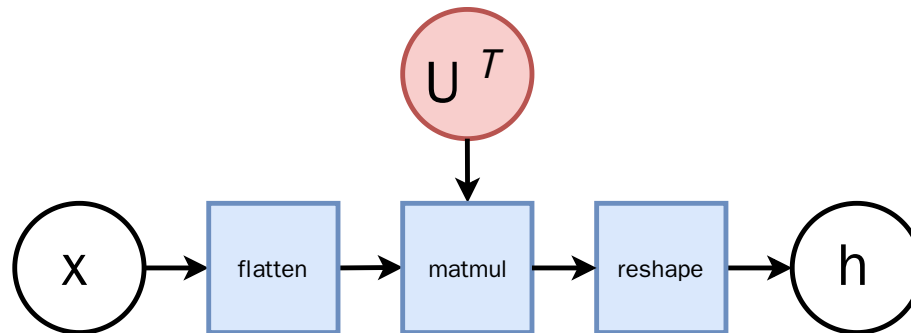
Transposed convolutions

A **transposed convolution** is a convolution where the implementation of the forward and backward passes are swapped.

Given a convolutional kernel \mathbf{u} ,

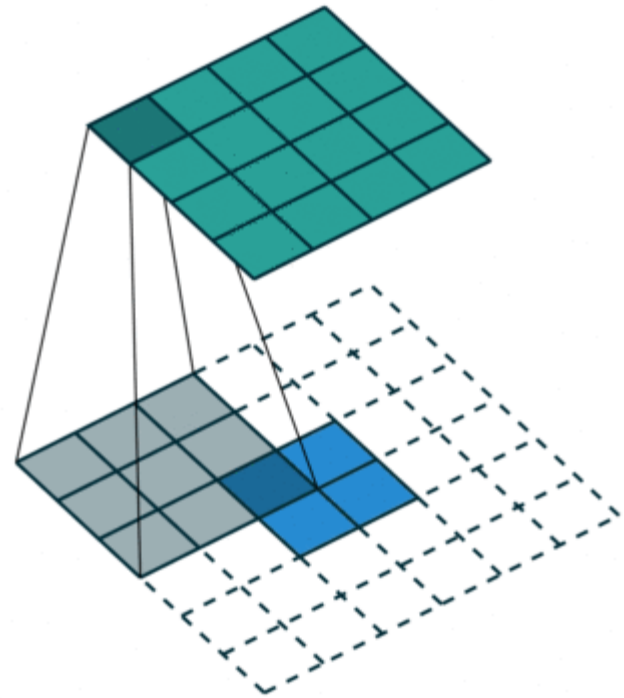
- the forward pass is implemented as $v(\mathbf{h}) = \mathbf{U}^T v(\mathbf{x})$ with appropriate reshaping, thereby effectively up-sampling an input $v(\mathbf{x})$ into a larger one;
- the backward pass is computed by multiplying the loss by \mathbf{U} instead of \mathbf{U}^T .

Transposed convolutions are also referred to as fractionally-strided convolutions or deconvolutions (mistakenly).



$$\mathbf{U}^T v(\mathbf{x}) = v(\mathbf{h})$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 4 & 1 & 0 & 0 \\ 1 & 4 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 4 & 1 & 4 & 1 \\ 3 & 4 & 1 & 4 \\ 0 & 3 & 0 & 1 \\ 3 & 0 & 1 & 0 \\ 3 & 3 & 4 & 1 \\ 1 & 3 & 3 & 4 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 3 & 3 \\ 0 & 0 & 1 & 3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 1 \\ 4 \\ 4 \end{pmatrix} = \begin{pmatrix} 2 \\ 9 \\ 6 \\ 1 \\ 6 \\ 29 \\ 30 \\ 7 \\ 10 \\ 29 \\ 33 \\ 13 \\ 12 \\ 24 \\ 16 \\ 4 \end{pmatrix}$$



X (original samples)

7 2 1 0 4 1 4 9 5 9 0 6
9 0 1 5 9 7 3 4 9 6 6 5
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(X)$ (CNN, $d = 2$)

7 2 1 0 4 1 4 9 6 9 0 6
9 0 1 5 9 7 5 9 9 6 6 5
9 0 7 4 0 1 5 1 3 6 7 2

$g \circ f(X)$ (PCA, $d = 2$)

9 3 1 0 9 1 9 9 8 9 8 8
9 0 1 3 9 9 3 9 9 8 9 8
9 0 9 9 8 1 3 1 3 8 9 8

X (original samples)

7 2 1 0 4 1 4 9 5 9 0 6
9 0 1 5 9 7 3 4 9 6 6 5
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(X)$ (CNN, $d = 8$)

7 2 1 0 4 1 4 9 5 9 0 6
9 0 1 5 9 7 3 4 9 6 6 5
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(X)$ (PCA, $d = 8$)

7 3 1 0 4 1 9 9 0 9 0 0
9 0 1 0 9 7 3 4 9 6 0 5
4 0 7 4 0 1 3 1 3 0 7 0

X (original samples)

7 2 1 0 4 1 4 9 5 9 0 6
9 0 1 5 9 7 8 4 9 6 6 5
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(X)$ (CNN, $d = 32$)

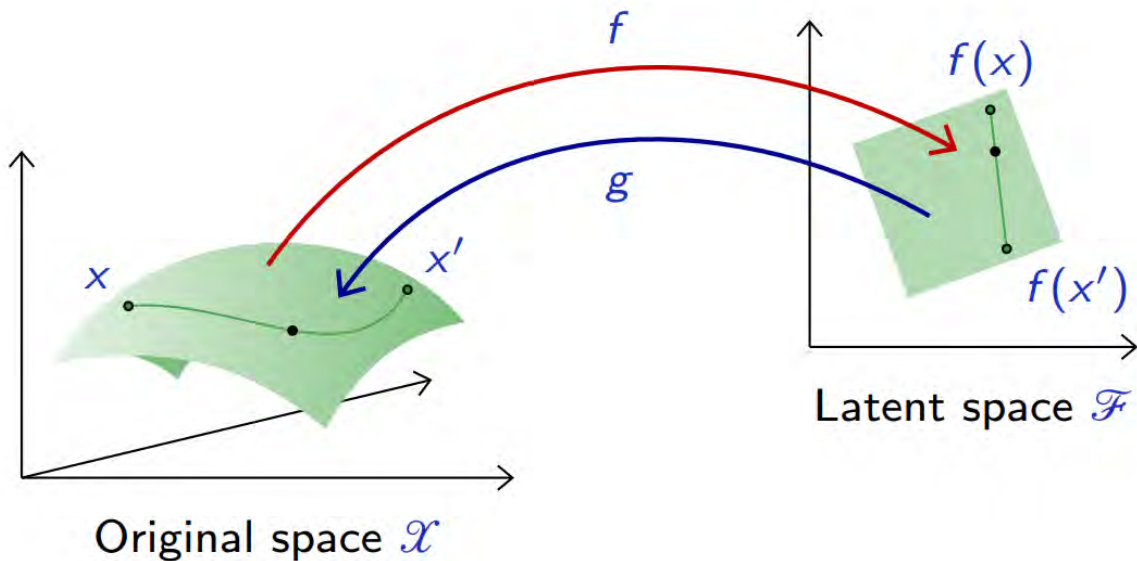
7 2 1 0 4 1 4 9 5 9 0 6
9 0 1 5 9 7 8 4 9 6 6 5
4 0 7 4 0 1 3 1 3 4 7 2

$g \circ f(X)$ (PCA, $d = 32$)

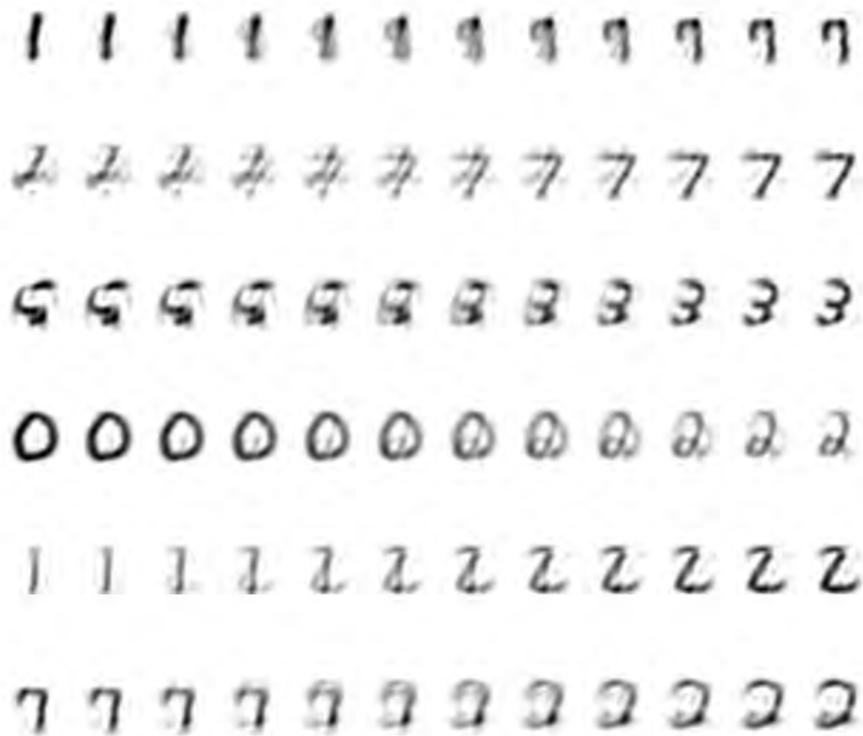
7 2 1 0 4 1 4 9 5 9 0 6
9 0 1 5 9 7 8 4 9 6 6 5
4 0 7 4 0 1 3 1 3 4 7 2

Interpolation

To get an intuition of the learned latent representation, we can pick two samples \mathbf{x} and \mathbf{x}' at random and interpolate samples along the line in the latent space.



PCA interpolation ($d = 32$)

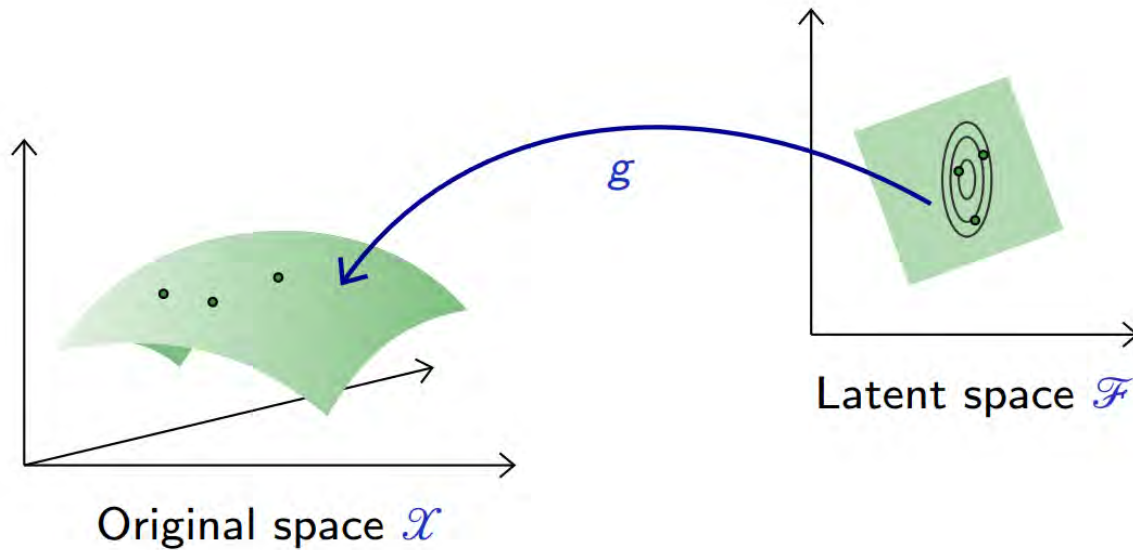


Autoencoder interpolation ($d = 32$)



Sampling from latent space

The generative capability of the decoder g can be assessed by introducing a (simple) density model q over the latent space \mathcal{Z} , sample there, and map the samples into the data space \mathcal{X} with g .



For instance, a factored Gaussian model with diagonal covariance matrix,

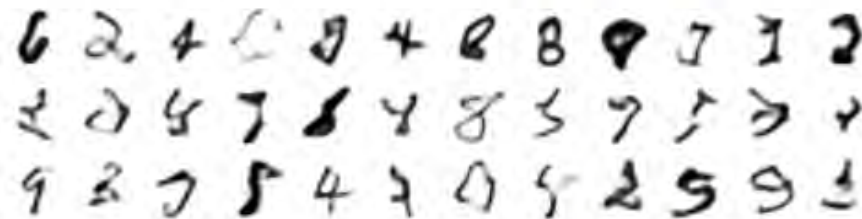
$$q(\mathbf{z}) = \mathcal{N}(\hat{\boldsymbol{\mu}}, \hat{\boldsymbol{\Sigma}}),$$

where both $\hat{\boldsymbol{\mu}}$ and $\hat{\boldsymbol{\Sigma}}$ are estimated on training data.

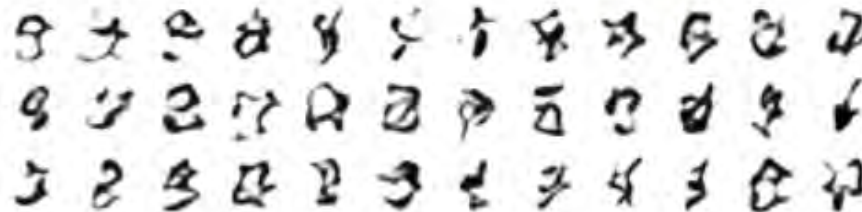
Autoencoder sampling ($d = 8$)



Autoencoder sampling ($d = 16$)



Autoencoder sampling ($d = 32$)



These results are not satisfactory because the density model on the latent space is **too simple and inadequate**.

Building a good model amounts to our original problem of modeling an empirical distribution, although it may now be in a lower dimension space.

Generative models

A **generative model** is a probabilistic model p that can be used as a **simulator of the data**. Its purpose is to generate synthetic but realistic high-dimensional data

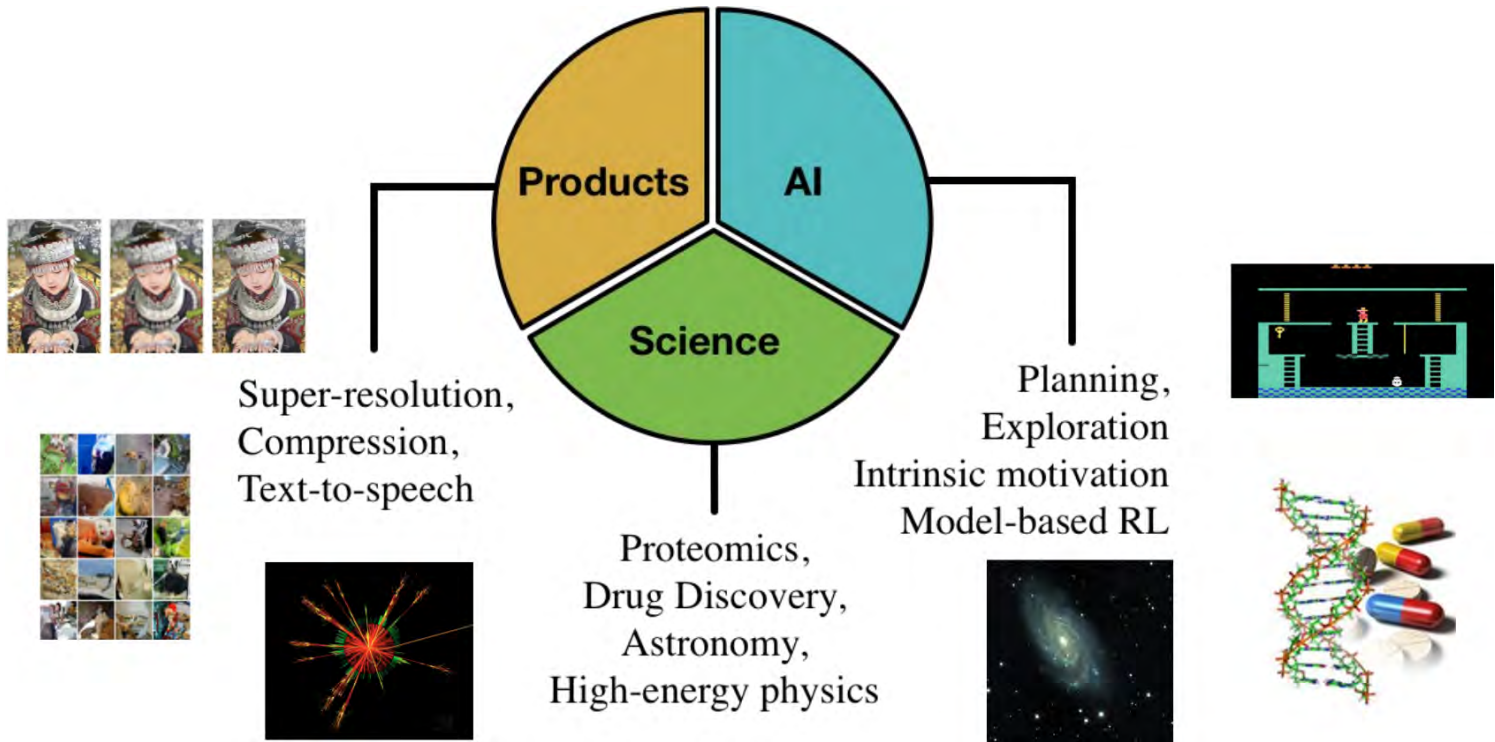
$$\mathbf{x} \sim p(\mathbf{x}; \theta),$$

that is as close as possible from the true but unknown data distribution $p(\mathbf{x})$, but for which we have empirical samples.

Motivation

Go beyond estimating $p(y|\mathbf{x})$:

- Understand and imagine how the world evolves.
- Recognize objects in the world and their factors of variation.
- Establish concepts for reasoning and decision making.



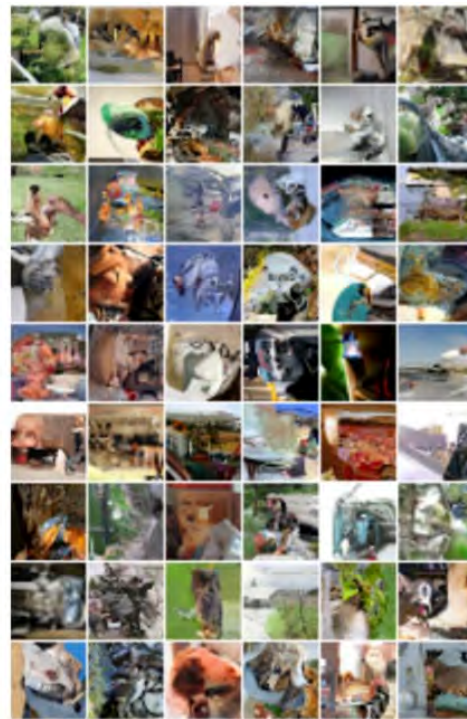
Generative models have a role in many important problems

Image and content generation

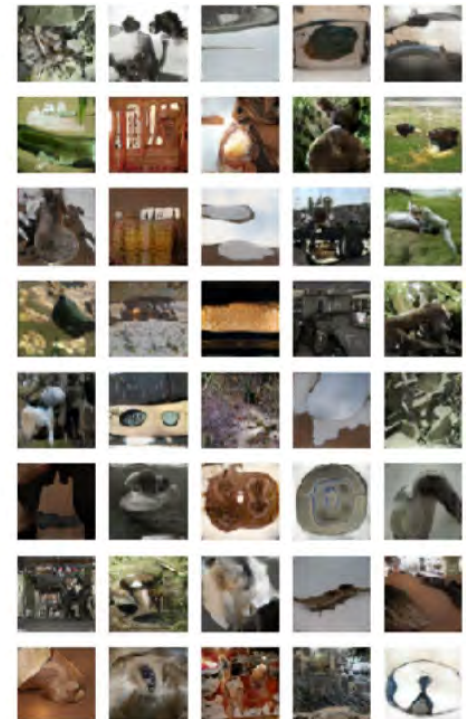
Generating images and video content.



DRAW



Pixel RNN

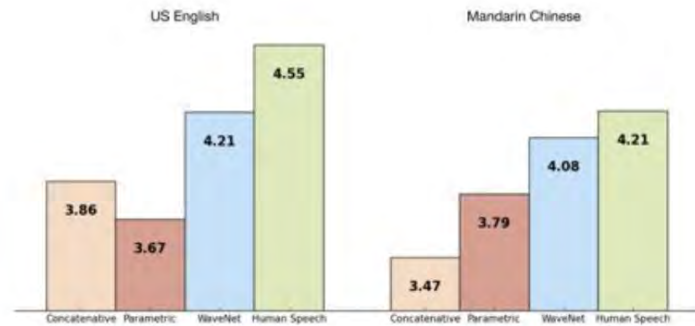
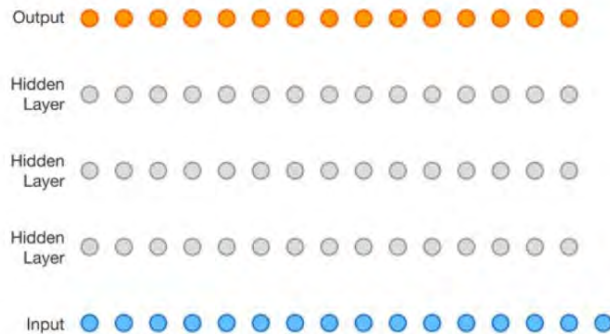


ALI

(Gregor et al, 2015; Oord et al, 2016; Dumoulin et al, 2016)

Text-to-speech synthesis

Generating audio conditioned on text.



(Oord et al, 2016)

Communication and compression

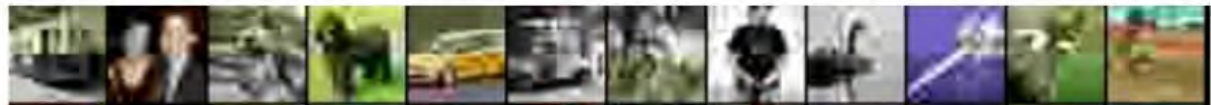
Hierarchical compression of images and other data.

Original images



Compression rate: 0.2bits/dimension

JPEG



JPEG-2000



RVAE v1



RVAE v2



(Gregor et al, 2016)

Image super-resolution

Photo-realistic single image super-resolution.

original



bicubic
(21.59dB/0.6423)



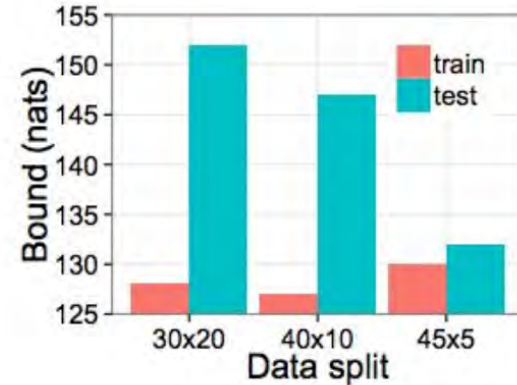
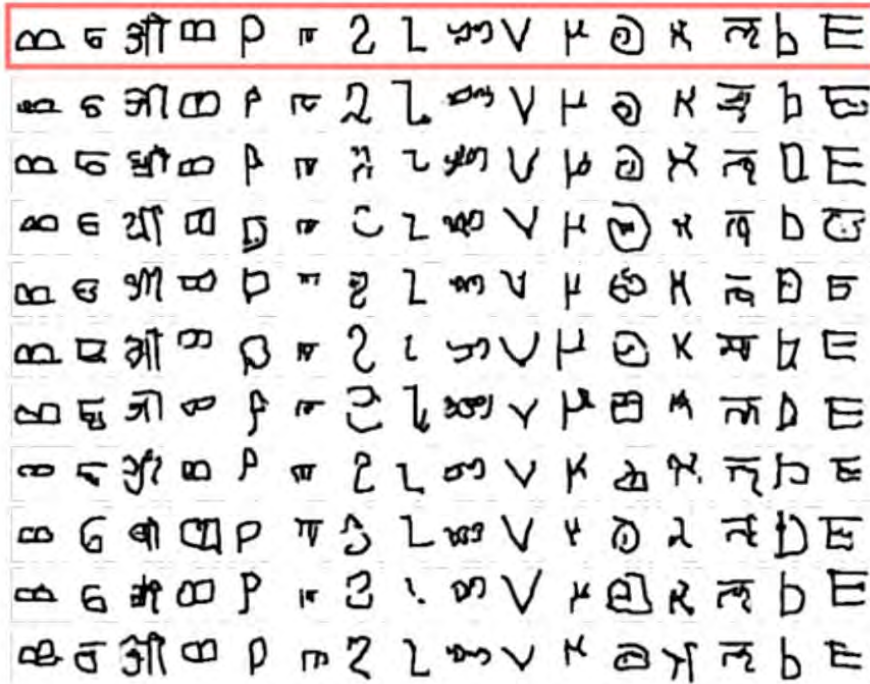
SRGAN
(20.34dB/0.6562)



(Ledig et al, 2016)

One-shot generalization

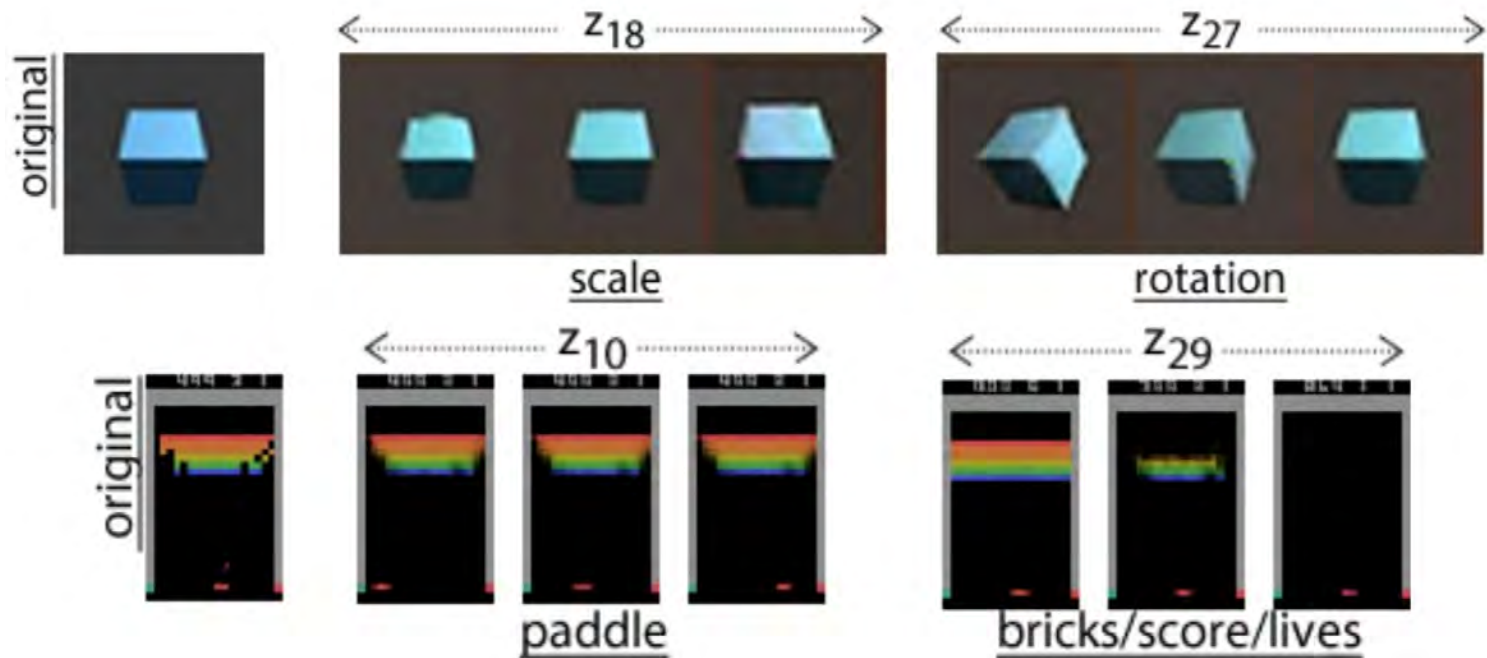
Rapid generalization of novel concepts.



(Gregor et al, 2016)

Visual concept learning

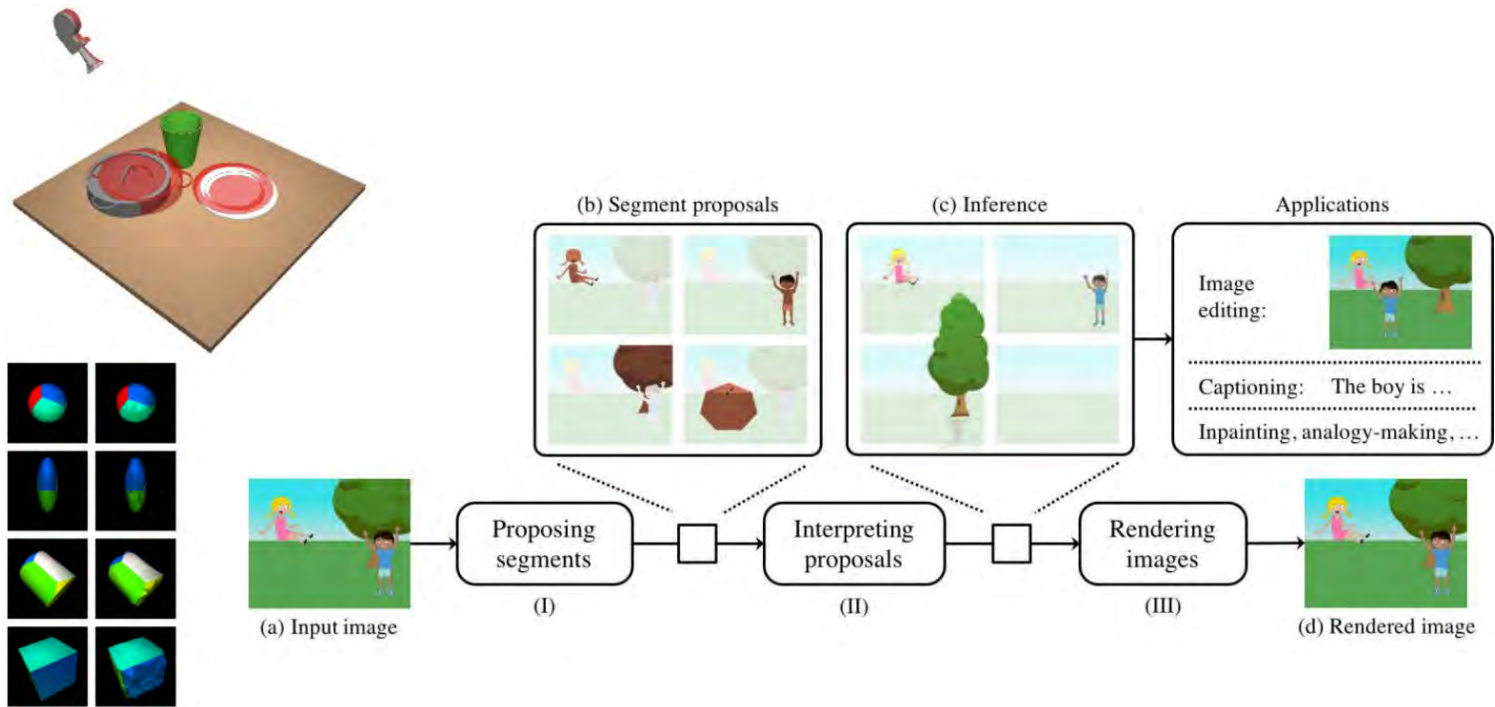
Understanding the factors of variation and invariances.



(Higgins et al, 2017)

Scene understanding

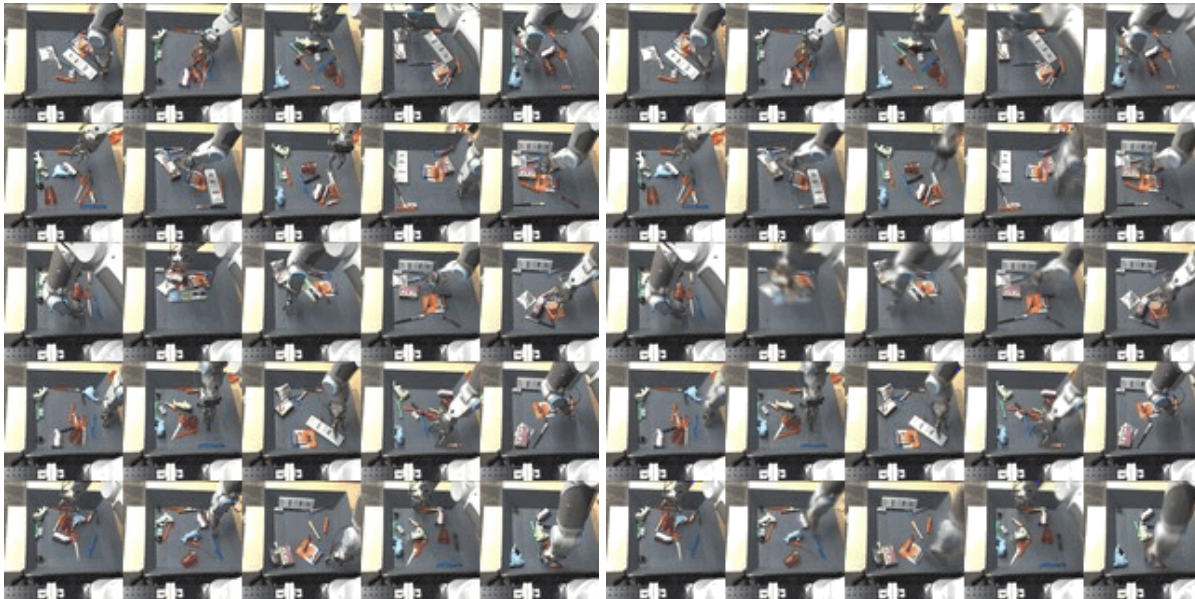
Understanding the components of scenes and their interactions.



(Wu et al, 2017)

Future simulation

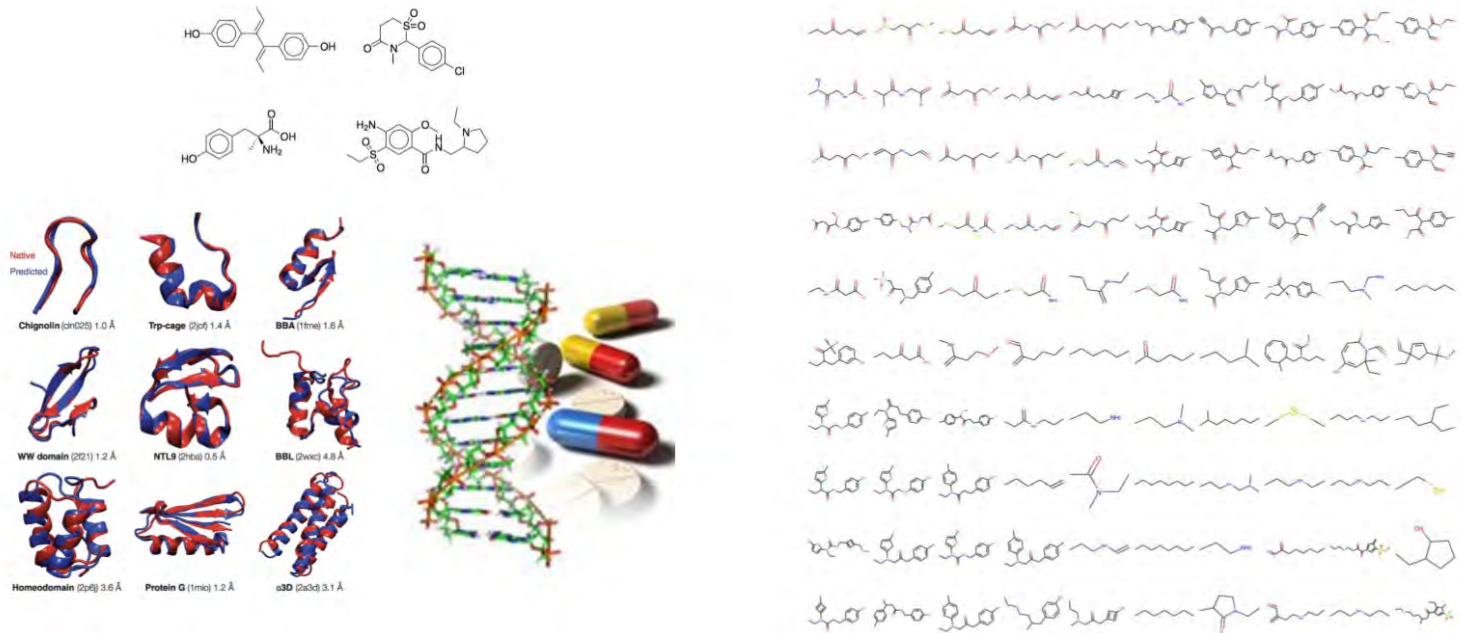
Simulate future trajectories of environments based on actions for planning.



(Finn et al, 2016)

Drug design and response prediction

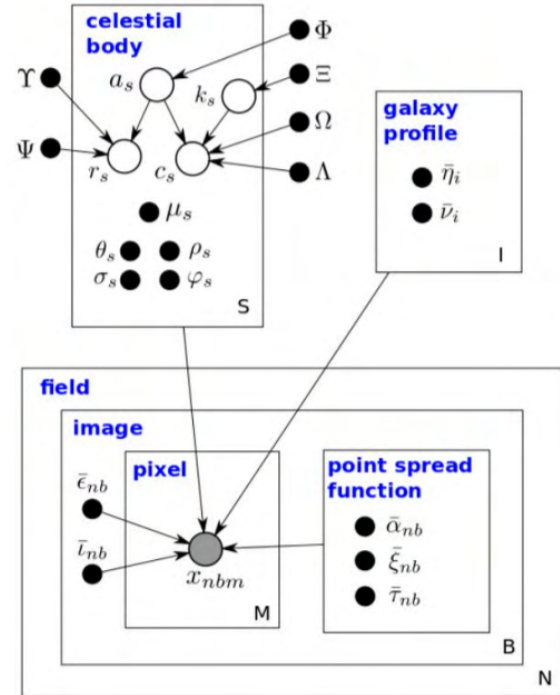
Generative models for proposing candidate molecules and for improving prediction through semi-supervised learning.



(Gomez-Bombarelli et al, 2016)

Locating celestial bodies

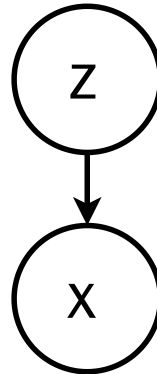
Generative models for applications in astronomy and high-energy physics.



(Regier et al, 2015)

Variational inference

Latent variable model



Consider for now a **prescribed latent variable model** that relates a set of observable variables $\mathbf{x} \in \mathcal{X}$ to a set of unobserved variables $\mathbf{z} \in \mathcal{Z}$.

The probabilistic model is given and motivated by domain knowledge assumptions.

Examples include:

- Linear discriminant analysis
- Bayesian networks
- Hidden Markov models
- Probabilistic programs

The probabilistic model defines a joint probability distribution $p(\mathbf{x}, \mathbf{z})$, which decomposes as

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{x}|\mathbf{z})p(\mathbf{z}).$$

If we interpret \mathbf{z} as causal factors for the high-dimension representations \mathbf{x} , then sampling from $p(\mathbf{x}|\mathbf{z})$ can be interpreted as a **stochastic generating process** from \mathcal{Z} to \mathcal{X} .

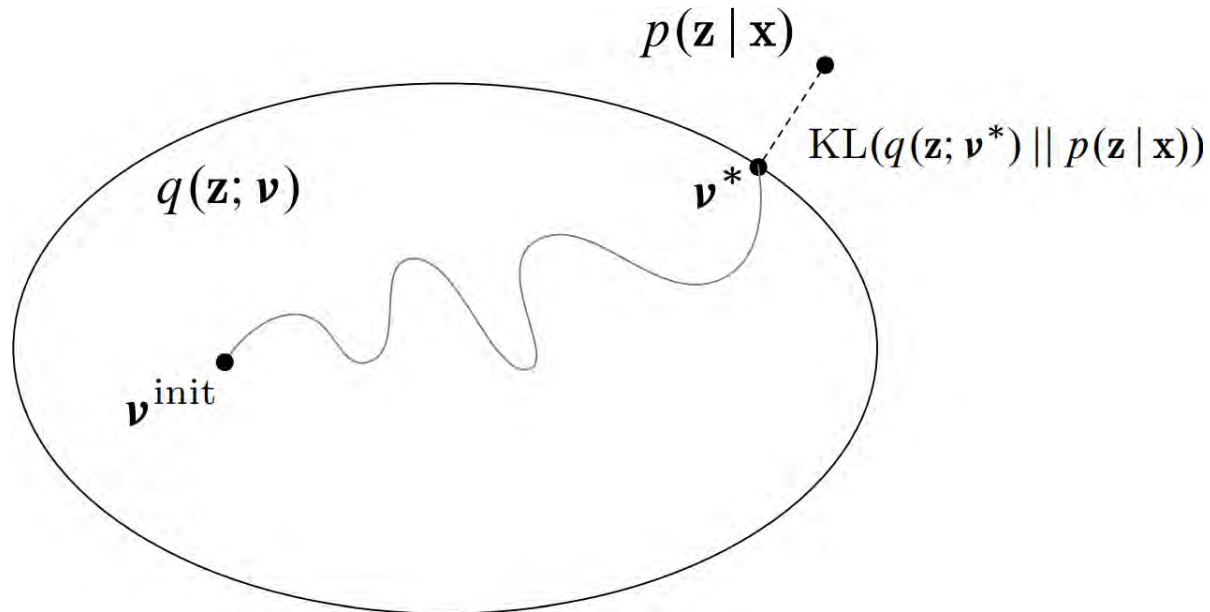
For a given model $p(\mathbf{x}, \mathbf{z})$, inference consists in computing the posterior

$$p(\mathbf{z}|\mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{z})p(\mathbf{z})}{p(\mathbf{x})}.$$

For most interesting cases, this is usually intractable since it requires evaluating the evidence

$$p(\mathbf{x}) = \int p(\mathbf{x}|\mathbf{z})p(\mathbf{z})d\mathbf{z}.$$

Variational inference



Variational inference turns posterior inference into an optimization problem.

- Consider a family of distributions $q(\mathbf{z} | \mathbf{x}; \nu)$ that approximate the posterior $p(\mathbf{z} | \mathbf{x})$, where the variational parameters ν index the family of distributions.
- The parameters ν are fit to minimize the KL divergence between $p(\mathbf{z} | \mathbf{x})$ and the approximation $q(\mathbf{z} | \mathbf{x}; \nu)$.

Formally, we want to minimize

$$\begin{aligned} KL(q(\mathbf{z}|\mathbf{x}; \nu) || p(\mathbf{z}|\mathbf{x})) &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \nu)} \left[\log \frac{q(\mathbf{z}|\mathbf{x}; \nu)}{p(\mathbf{z}|\mathbf{x})} \right] \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \nu)} [\log q(\mathbf{z}|\mathbf{x}; \nu) - \log p(\mathbf{x}, \mathbf{z})] + \log p(\mathbf{x}). \end{aligned}$$

For the same reason as before, the KL divergence cannot be directly minimized because of the $\log p(\mathbf{x})$ term.

However, we can write

$$KL(q(\mathbf{z}|\mathbf{x}; \nu) || p(\mathbf{z}|\mathbf{x})) = \log p(\mathbf{x}) - \underbrace{\mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \nu)} [\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\mathbf{x}; \nu)]}_{\text{ELBO}(\mathbf{x}; \nu)}$$

where $\text{ELBO}(\mathbf{x}; \nu)$ is called the **evidence lower bound objective**.

- Since $\log p(\mathbf{x})$ does not depend on ν , it can be considered as a constant, and minimizing the KL divergence is equivalent to maximizing the evidence lower bound, while being computationally tractable.
- Given a dataset $\mathbf{d} = \{\mathbf{x}_i | i = 1, \dots, N\}$, the final objective is the sum $\sum_{\{\mathbf{x}_i \in \mathbf{d}\}} \text{ELBO}(\mathbf{x}_i; \nu)$.

Remark that

$$\begin{aligned}\text{ELBO}(\mathbf{x}; \nu) &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \nu)} [\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\mathbf{x}; \nu)] \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \nu)} [\log p(\mathbf{x}|\mathbf{z})p(\mathbf{z}) - \log q(\mathbf{z}|\mathbf{x}; \nu)] \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \nu)} [\log p(\mathbf{x}|\mathbf{z})] - KL(q(\mathbf{z}|\mathbf{x}; \nu) || p(\mathbf{z}))\end{aligned}$$

Therefore, maximizing the ELBO:

- encourages distributions to place their mass on configurations of latent variables that explain the observed data (first term);
- encourages distributions close to the prior (second term).

Optimization

We want

$$\begin{aligned}\nu^* &= \arg \max_{\nu} \text{ELBO}(\mathbf{x}; \nu) \\ &= \arg \max_{\nu} \mathbb{E}_{q(\mathbf{z}|\mathbf{x};\nu)} [\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}|\mathbf{x}; \nu)].\end{aligned}$$

We can proceed by gradient ascent, provided we can evaluate $\nabla_{\nu} \text{ELBO}(\mathbf{x}; \nu)$.

In general, this gradient is difficult to compute because the expectation is unknown and the parameters ν are parameters of the distribution $q(\mathbf{z}|\mathbf{x}; \nu)$ we integrate over.

Variational auto-encoders

So far we assumed a prescribed probabilistic model motivated by domain knowledge. We will now directly learn a stochastic generating process with a neural network.

Variational auto-encoders

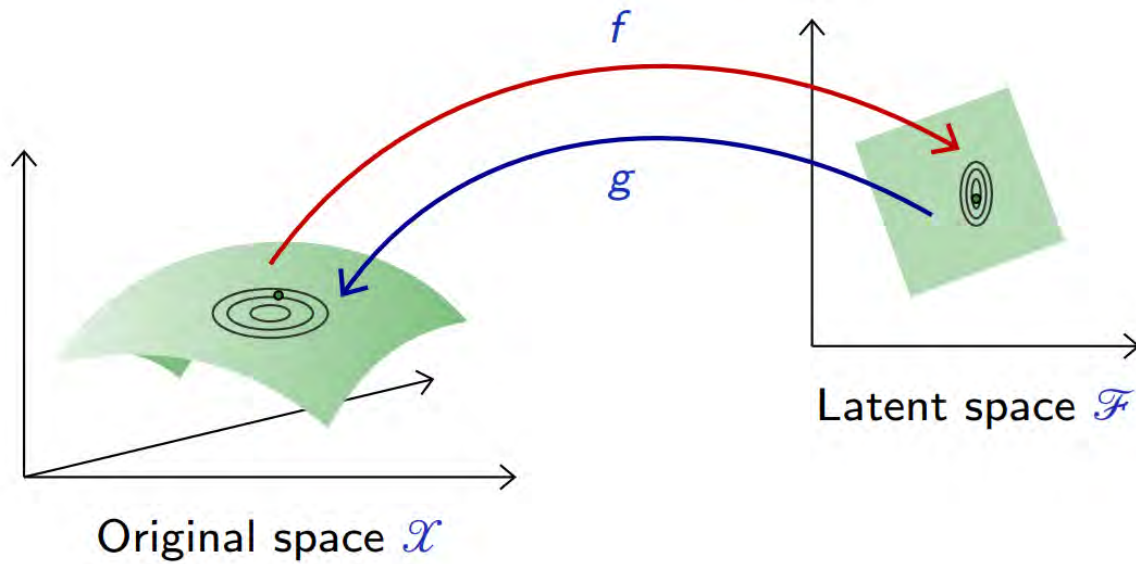
A variational auto-encoder is a deep latent variable model where:

- The likelihood $p(\mathbf{x}|\mathbf{z}; \theta)$ is parameterized with a **generative network** NN_θ (or decoder) that takes as input \mathbf{z} and outputs parameters $\phi = \text{NN}_\theta(\mathbf{z})$ to the data distribution. E.g.,

$$\begin{aligned}\mu, \sigma &= \text{NN}_\theta(\mathbf{z}) \\ p(\mathbf{x}|\mathbf{z}; \theta) &= \mathcal{N}(\mathbf{x}; \mu, \sigma^2 \mathbf{I})\end{aligned}$$

- The approximate posterior $q(\mathbf{z}|\mathbf{x}; \varphi)$ is parameterized with an **inference network** NN_φ (or encoder) that takes as input \mathbf{x} and outputs parameters $\nu = \text{NN}_\varphi(\mathbf{x})$ to the approximate posterior. E.g.,

$$\begin{aligned}\mu, \sigma &= \text{NN}_\varphi(\mathbf{x}) \\ q(\mathbf{z}|\mathbf{x}; \varphi) &= \mathcal{N}(\mathbf{z}; \mu, \sigma^2 \mathbf{I})\end{aligned}$$



As before, we can use variational inference, but to jointly optimize the generative and the inference networks parameters θ and φ .

We want

$$\begin{aligned}\theta^*, \varphi^* &= \arg \max_{\theta, \varphi} \text{ELBO}(\mathbf{x}; \theta, \varphi) \\ &= \arg \max_{\theta, \varphi} \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \varphi)} [\log p(\mathbf{x}, \mathbf{z}; \theta) - \log q(\mathbf{z}|\mathbf{x}; \varphi)] \\ &= \arg \max_{\theta, \varphi} \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \varphi)} [\log p(\mathbf{x}|\mathbf{z}; \theta)] - KL(q(\mathbf{z}|\mathbf{x}; \varphi) || p(\mathbf{z})).\end{aligned}$$

- Given some generative network θ , we want to put the mass of the latent variables, by adjusting φ , such that they explain the observed data, while remaining close to the prior.
- Given some inference network φ , we want to put the mass of the observed variables, by adjusting θ , such that they are well explained by the latent variables.

Unbiased gradients of the ELBO with respect to the generative model parameters θ are simple to obtain:

$$\begin{aligned}\nabla_{\theta} \text{ELBO}(\mathbf{x}; \theta, \varphi) &= \nabla_{\theta} \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \varphi)} [\log p(\mathbf{x}, \mathbf{z}; \theta) - \log q(\mathbf{z}|\mathbf{x}; \varphi)] \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \varphi)} [\nabla_{\theta} (\log p(\mathbf{x}, \mathbf{z}; \theta) - \log q(\mathbf{z}|\mathbf{x}; \varphi))] \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \varphi)} [\nabla_{\theta} \log p(\mathbf{x}, \mathbf{z}; \theta)],\end{aligned}$$

which can be estimated with Monte Carlo integration.

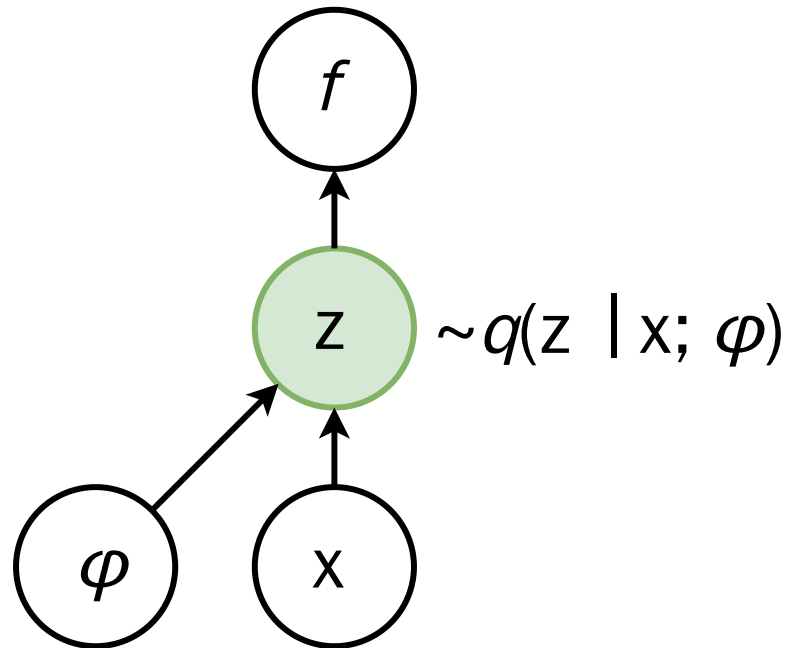
However, gradients with respect to the inference model parameters φ are more difficult to obtain:

$$\begin{aligned}\nabla_{\varphi} \text{ELBO}(\mathbf{x}; \theta, \varphi) &= \nabla_{\varphi} \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \varphi)} [\log p(\mathbf{x}, \mathbf{z}; \theta) - \log q(\mathbf{z}|\mathbf{x}; \varphi)] \\ &\neq \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \varphi)} [\nabla_{\varphi} (\log p(\mathbf{x}, \mathbf{z}; \theta) - \log q(\mathbf{z}|\mathbf{x}; \varphi))]\end{aligned}$$

Let us abbreviate

$$\begin{aligned}\text{ELBO}(\mathbf{x}; \theta, \varphi) &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x};\varphi)} [\log p(\mathbf{x}, \mathbf{z}; \theta) - \log q(\mathbf{z}|\mathbf{x}; \varphi)] \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x};\varphi)} [f(\mathbf{x}, \mathbf{z}; \varphi)].\end{aligned}$$

We have



We cannot backpropagate through the stochastic node \mathbf{z} to compute $\nabla_{\varphi} f$!

Reparameterization trick

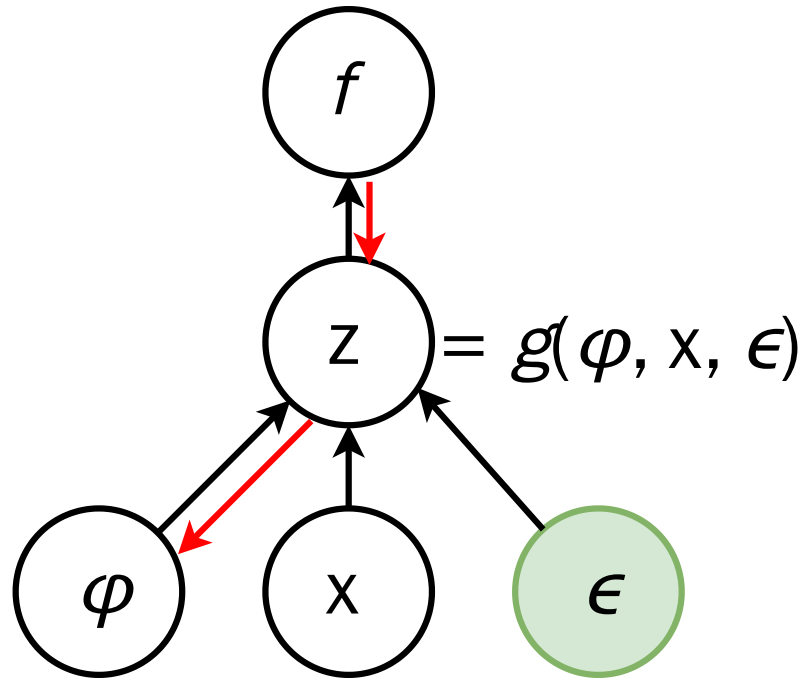
The **reparameterization trick** consists in re-expressing the variable

$$\mathbf{z} \sim q(\mathbf{z}|\mathbf{x}; \varphi)$$

as some differentiable and invertible transformation of another random variable ϵ given \mathbf{x} and φ ,

$$\mathbf{z} = g(\varphi, \mathbf{x}, \epsilon),$$

such that the distribution of ϵ is independent of \mathbf{x} or φ .



For example, if $q(\mathbf{z}|\mathbf{x}; \varphi) = \mathcal{N}(\mathbf{z}; \mu(\mathbf{x}; \varphi), \sigma^2(\mathbf{x}; \varphi))$, where $\mu(\mathbf{x}; \varphi)$ and $\sigma^2(\mathbf{x}; \varphi)$ are the outputs of the inference network NN_φ , then a common reparameterization is:

$$p(\epsilon) = \mathcal{N}(\epsilon; \mathbf{0}, \mathbf{I})$$

$$\mathbf{z} = \mu(\mathbf{x}; \varphi) + \sigma(\mathbf{x}; \varphi) \odot \epsilon$$

Given such a change of variable, the ELBO can be rewritten as:

$$\begin{aligned}\text{ELBO}(\mathbf{x}; \theta, \varphi) &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x};\varphi)} [f(\mathbf{x}, \mathbf{z}; \varphi)] \\ &= \mathbb{E}_{p(\epsilon)} [f(\mathbf{x}, g(\varphi, \mathbf{x}, \epsilon); \varphi)]\end{aligned}$$

Therefore,

$$\begin{aligned}\nabla_{\varphi} \text{ELBO}(\mathbf{x}; \theta, \varphi) &= \nabla_{\varphi} \mathbb{E}_{p(\epsilon)} [f(\mathbf{x}, g(\varphi, \mathbf{x}, \epsilon); \varphi)] \\ &= \mathbb{E}_{p(\epsilon)} [\nabla_{\varphi} f(\mathbf{x}, g(\varphi, \mathbf{x}, \epsilon); \varphi)],\end{aligned}$$

which we can now estimate with Monte Carlo integration.

The last required ingredient is the evaluation of the likelihood $q(\mathbf{z}|\mathbf{x}; \varphi)$ given the change of variable g . As long as g is invertible, we have:

$$\log q(\mathbf{z}|\mathbf{x}; \varphi) = \log p(\epsilon) - \log \left| \det \left(\frac{\partial \mathbf{z}}{\partial \epsilon} \right) \right|.$$

Example

Consider the following setup:

- Generative model:

$$\mathbf{z} \in \mathbb{R}^J$$

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{I})$$

$$p(\mathbf{x}|\mathbf{z}; \theta) = \mathcal{N}(\mathbf{x}; \mu(\mathbf{z}; \theta), \sigma^2(\mathbf{z}; \theta)\mathbf{I})$$

$$\mu(\mathbf{z}; \theta) = \mathbf{W}_2^T \mathbf{h} + \mathbf{b}_2$$

$$\log \sigma^2(\mathbf{z}; \theta) = \mathbf{W}_3^T \mathbf{h} + \mathbf{b}_3$$

$$\mathbf{h} = \text{ReLU}(\mathbf{W}_1^T \mathbf{z} + \mathbf{b}_1)$$

$$\theta = \{\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2, \mathbf{W}_3, \mathbf{b}_3\}$$

- Inference model:

$$q(\mathbf{z}|\mathbf{x}; \varphi) = \mathcal{N}(\mathbf{z}; \mu(\mathbf{x}; \varphi), \sigma^2(\mathbf{x}; \varphi)\mathbf{I})$$

$$p(\epsilon) = \mathcal{N}(\epsilon; \mathbf{0}, \mathbf{I})$$

$$\mathbf{z} = \mu(\mathbf{x}; \varphi) + \sigma(\mathbf{x}; \varphi) \odot \epsilon$$

$$\mu(\mathbf{x}; \varphi) = \mathbf{W}_5^T \mathbf{h} + \mathbf{b}_5$$

$$\log \sigma^2(\mathbf{x}; \varphi) = \mathbf{W}_6^T \mathbf{h} + \mathbf{b}_6$$

$$\mathbf{h} = \text{ReLU}(\mathbf{W}_4^T \mathbf{x} + \mathbf{b}_4)$$

$$\varphi = \{\mathbf{W}_4, \mathbf{b}_4, \mathbf{W}_5, \mathbf{b}_5, \mathbf{W}_6, \mathbf{b}_6\}$$

Note that there is no restriction on the generative and inference network architectures. They could as well be arbitrarily complex convolutional networks.

Plugging everything together, the objective can be expressed as:

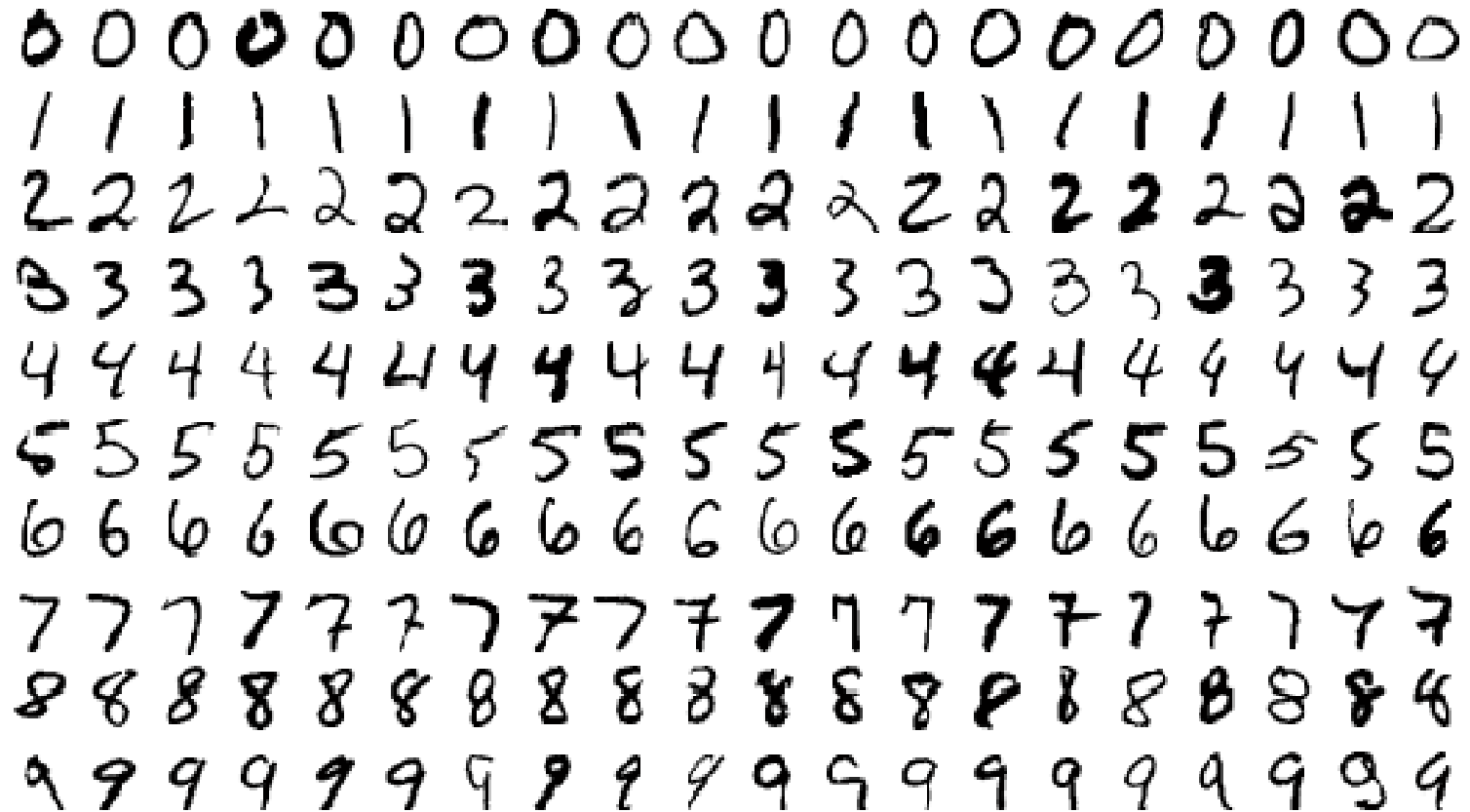
$$\begin{aligned}\text{ELBO}(\mathbf{x}; \theta, \varphi) &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \varphi)} [\log p(\mathbf{x}, \mathbf{z}; \theta) - \log q(\mathbf{z}|\mathbf{x}; \varphi)] \\ &= \mathbb{E}_{q(\mathbf{z}|\mathbf{x}; \varphi)} [\log p(\mathbf{x}|\mathbf{z}; \theta)] - KL(q(\mathbf{z}|\mathbf{x}; \varphi) || p(\mathbf{z})) \\ &= \mathbb{E}_{p(\epsilon)} [\log p(\mathbf{x}|\mathbf{z} = g(\varphi, \mathbf{x}, \epsilon); \theta)] - KL(q(\mathbf{z}|\mathbf{x}; \varphi) || p(\mathbf{z}))\end{aligned}$$

where the KL divergence can be expressed analytically as

$$KL(q(\mathbf{z}|\mathbf{x}; \varphi) || p(\mathbf{z})) = \frac{1}{2} \sum_{j=1}^J (1 + \log(\sigma_j^2(\mathbf{x}; \varphi)) - \mu_j^2(\mathbf{x}; \varphi) - \sigma_j^2(\mathbf{x}; \varphi)),$$

which allows to evaluate its derivative without approximation.

Consider as data **d** the MNIST digit dataset:



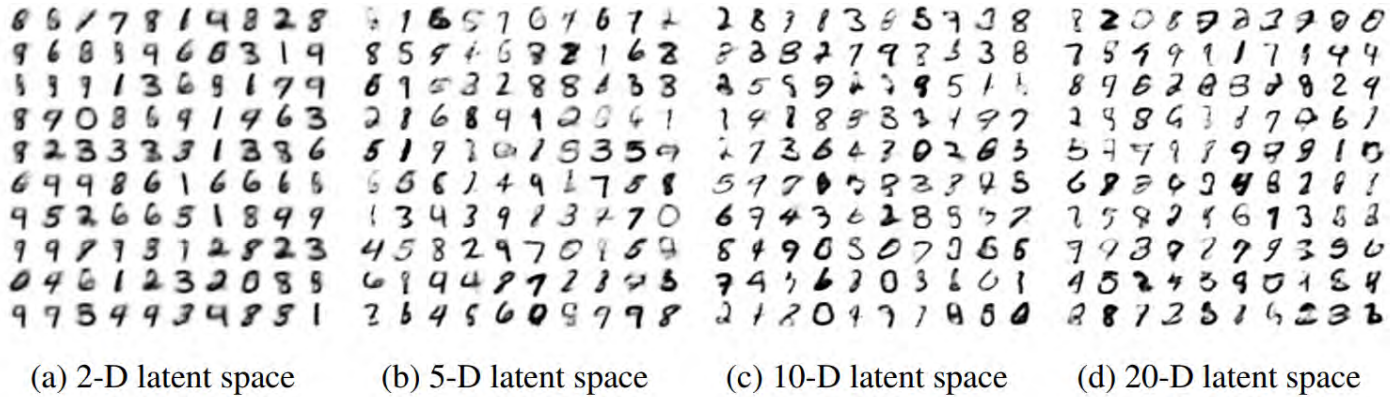
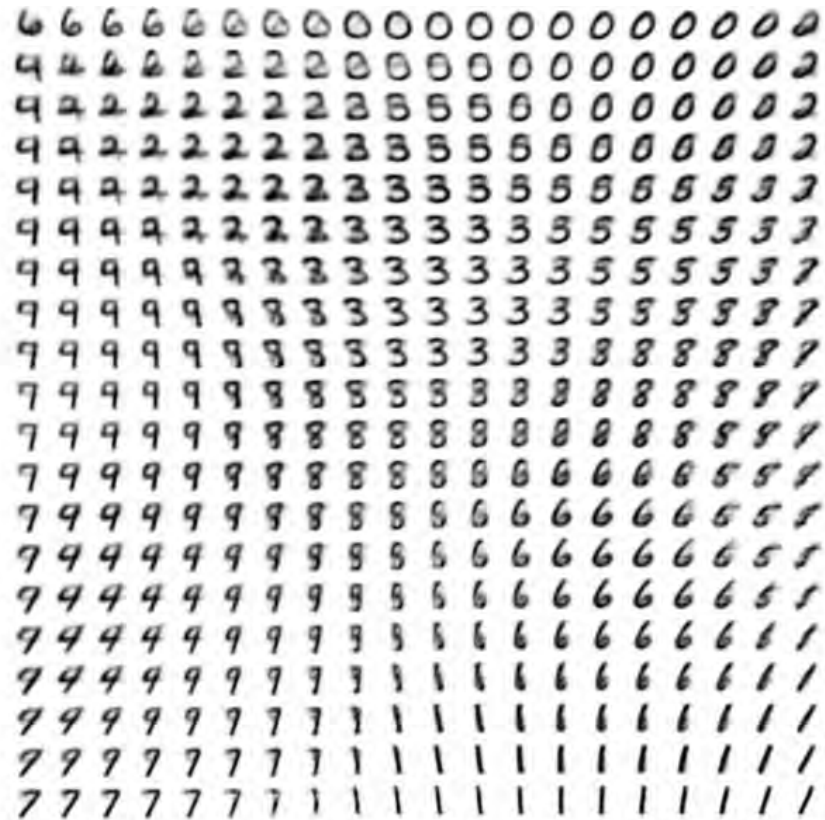


Figure 5: Random samples from learned generative models of MNIST for different dimensionalities of latent space.

(Kingma and Welling, 2013)



(a) Learned Frey Face manifold



(b) Learned MNIST manifold

Figure 4: Visualisations of learned data manifold for generative models with two-dimensional latent space, learned with AEVB. Since the prior of the latent space is Gaussian, linearly spaced coordinates on the unit square were transformed through the inverse CDF of the Gaussian to produce values of the latent variables \mathbf{z} . For each of these values \mathbf{z} , we plotted the corresponding generative $p_{\theta}(\mathbf{x}|\mathbf{z})$ with the learned parameters θ .

Applications of (variational) AEs



Face manifold from conv/deconv variational autoe...



Watch later

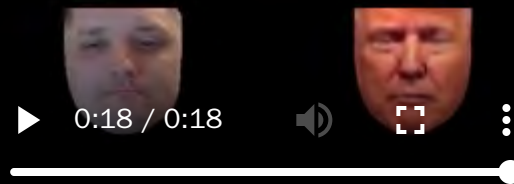


Share



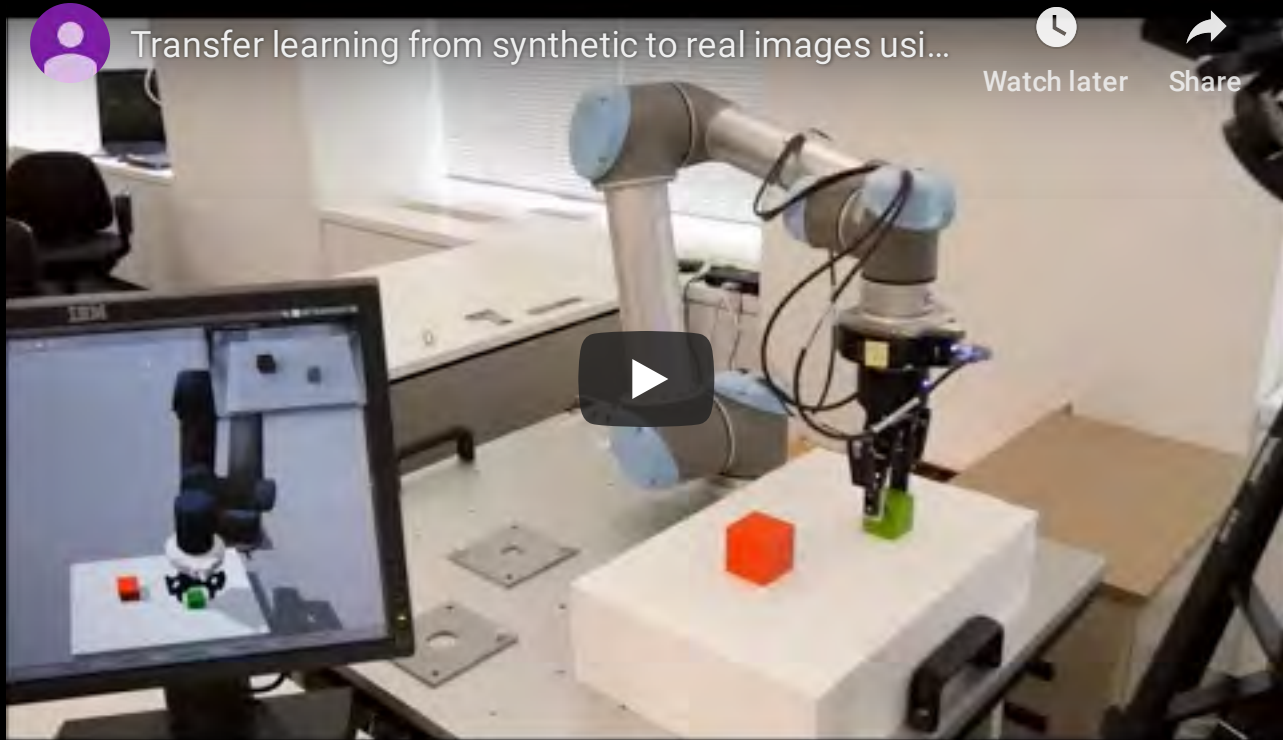
Random walks in latent space.

(Alex Radford, 2015)



Impersonation by encoding-decoding an unknown face.

(Kamil Czarnogórski, 2016)



(Inoue et al, 2017)

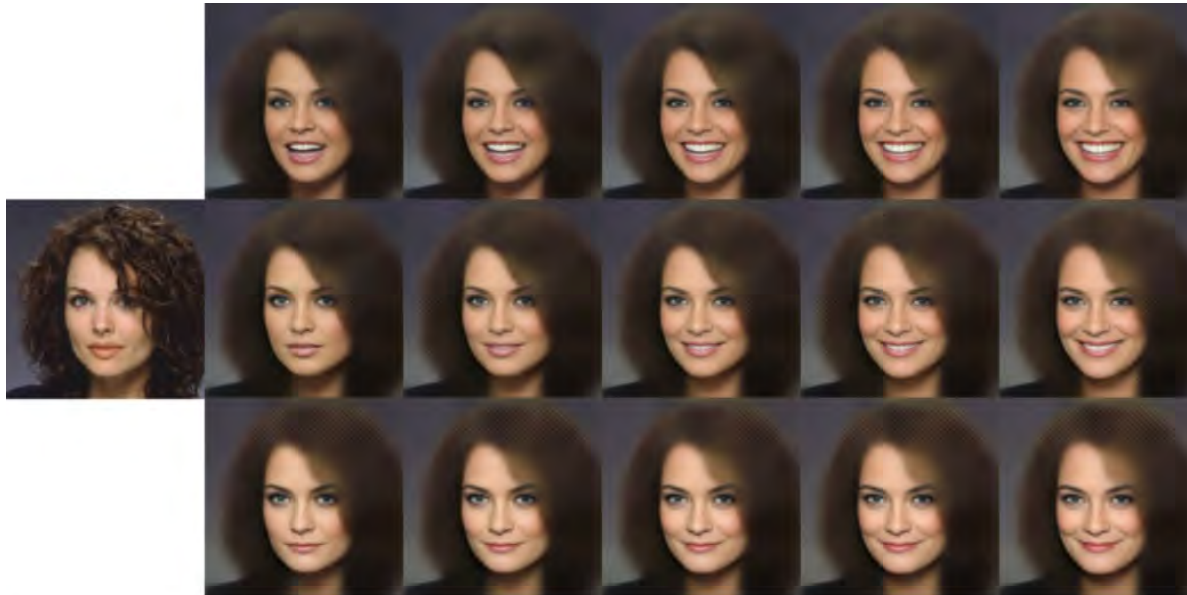


Figure 7: Decoupling attribute vectors for smiling (x-axis) and mouth open (y-axis) allows for more flexible latent space transformations. Input shown at left with reconstruction adjacent. (model: VAE from Lamb 16 on CelebA)

(Tom White, 2016)

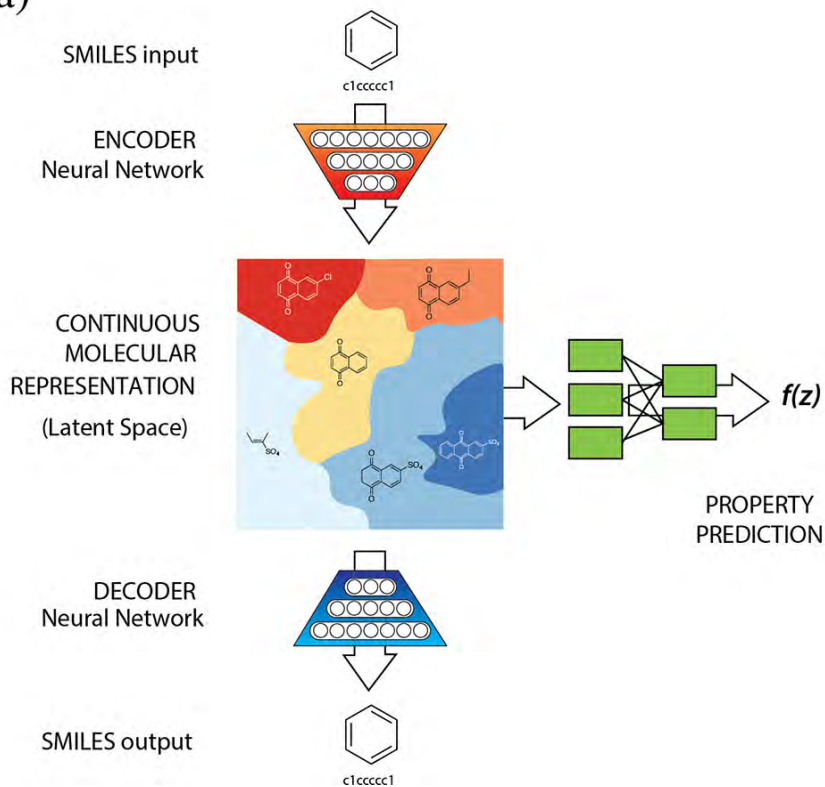
“ i want to talk to you . ”
“i want to be with you . ”
“i do n’t want to be with you . ”
i do n’t want to be with you .
she did n’t want to be with him .

he was silent for a long moment .
he was silent for a moment .
it was quiet for a moment .
it was dark and cold .
there was a pause .
it was my turn .

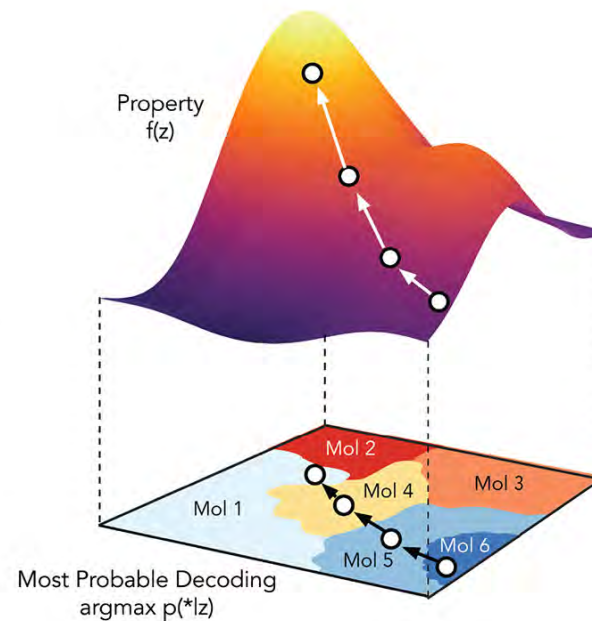
Table 8: Paths between pairs of random points in VAE space: Note that intermediate sentences are grammatical, and that topic and syntactic structure are usually locally consistent.

(Bowman et al, 2015)

(a)



(b)



Design of new molecules with desired chemical properties.
(Gomez-Bombarelli et al, 2016)

The end.

References

- Mohamed and Rezende, "[Tutorial on Deep Generative Models](#)", UAI 2017.
- Blei et al, "[Variational inference: Foundations and modern methods](#)", 2016.
- Kingma and Welling, "[Auto-Encoding Variational Bayes](#)", 2013.

Deep Learning

Lecture 7: Generative adversarial networks

Prof. Gilles Louppe
g.louppe@uliege.be

*Turing Award Won by 3
Pioneers in Artificial Intelligence*



*"ACM named **Yoshua Bengio**, **Geoffrey Hinton**, and **Yann LeCun** recipients of the **2018 ACM A.M. Turing Award** for conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing."*

Today

Learn a model of the data.

- Generative adversarial networks
- Wasserstein GANs
- Convergence of GANs
- State of the art
- Applications



"Generative adversarial networks is the coolest idea in deep learning in the last 20 years." -- Yann LeCun.

Generative adversarial networks

GANs



A two-player game

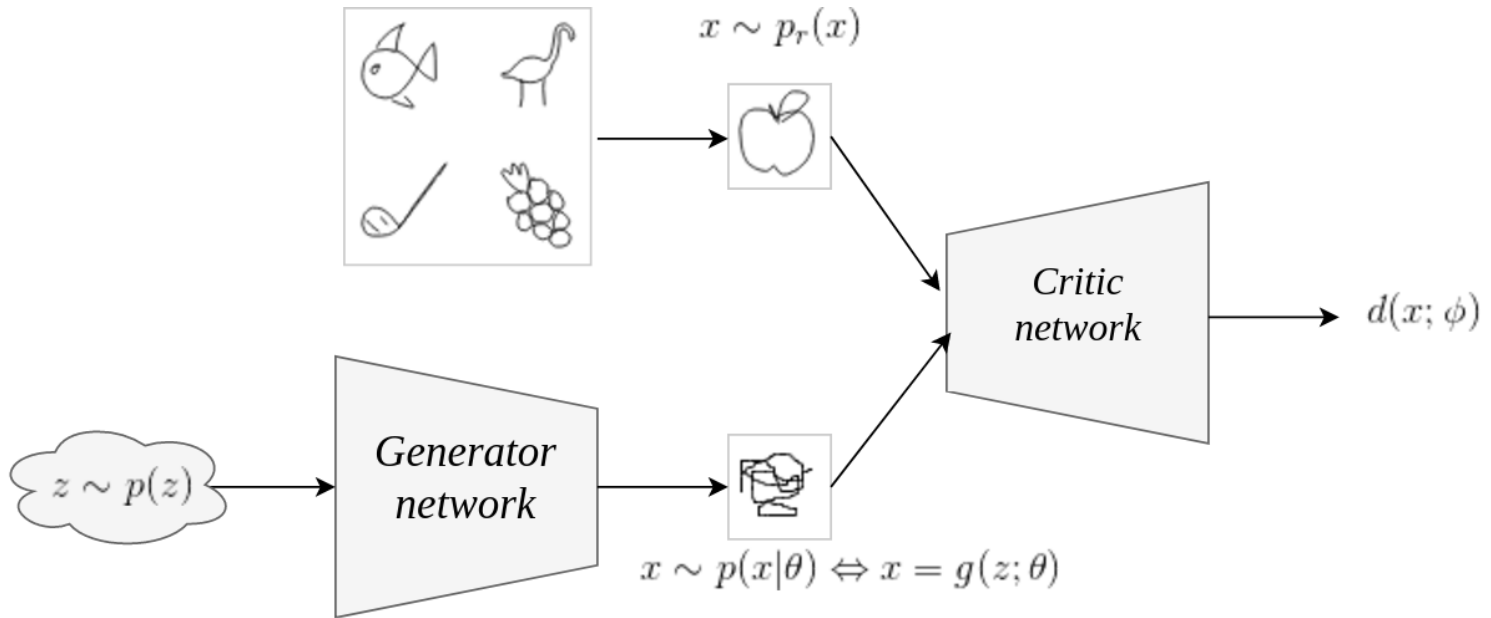
In **generative adversarial networks** (GANs), the task of learning a generative model is expressed as a two-player zero-sum game between two networks.

- The first network is a **generator** $g(\cdot; \theta) : \mathcal{Z} \rightarrow \mathcal{X}$, mapping a latent space equipped with a prior distribution $p(\mathbf{z})$ to the data space, thereby inducing a distribution

$$\mathbf{x} \sim q(\mathbf{x}; \theta) \Leftrightarrow \mathbf{z} \sim p(\mathbf{z}), \mathbf{x} = g(\mathbf{z}; \theta).$$

- The second network $d(\cdot; \phi) : \mathcal{X} \rightarrow [0, 1]$ is a **classifier** trained to distinguish between true samples $\mathbf{x} \sim p(\mathbf{x})$ and generated samples $\mathbf{x} \sim q(\mathbf{x}; \theta)$.

The central mechanism consists in using supervised learning to guide the learning of the generative model.



$$\arg \min_{\theta} \max_{\phi} \underbrace{\mathbb{E}_{x \sim p(x)} [\log d(\mathbf{x}; \phi)] + \mathbb{E}_{z \sim p(z)} [\log(1 - d(g(\mathbf{z}; \theta); \phi))]}_{V(\phi, \theta)}$$

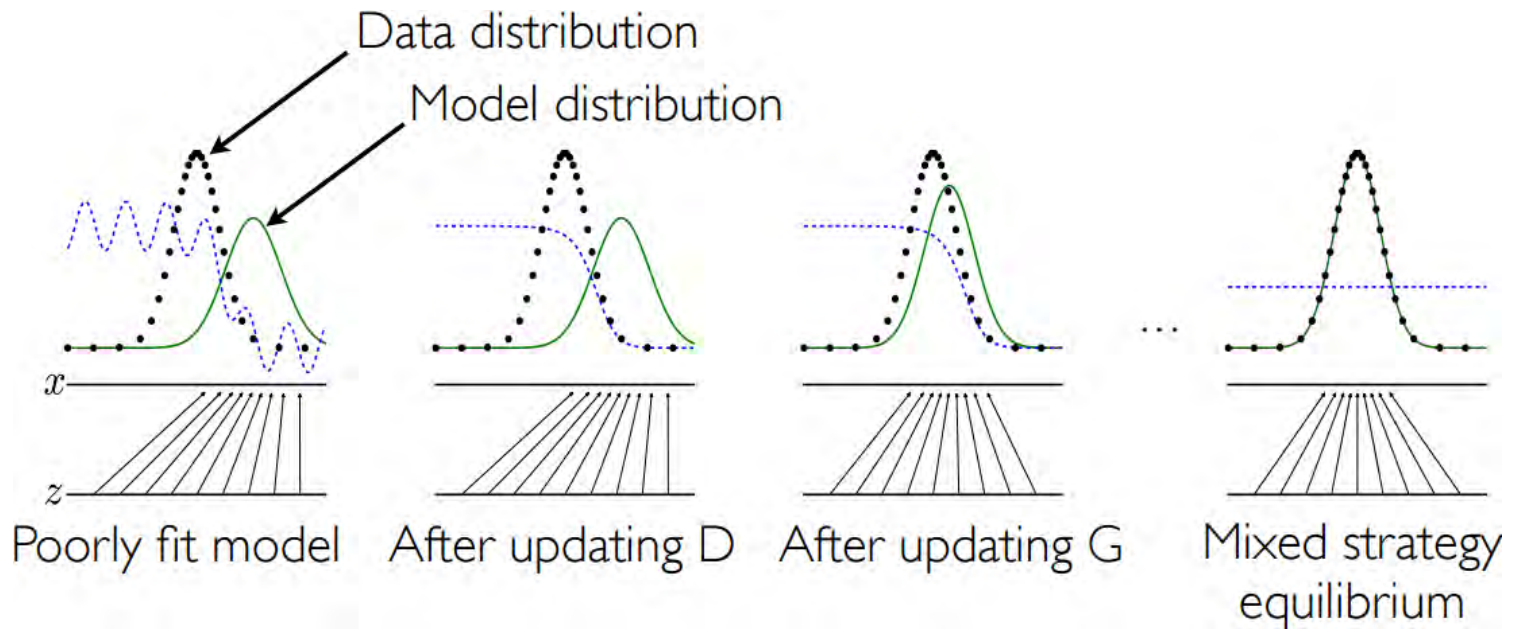
Learning process

In practice, the minimax solution is approximated using **alternating** stochastic gradient descent:

$$\begin{aligned}\theta &\leftarrow \theta - \gamma \nabla_{\theta} V(\phi, \theta) \\ \phi &\leftarrow \phi + \gamma \nabla_{\phi} V(\phi, \theta),\end{aligned}$$

where gradients are estimated with Monte Carlo integration.

- For one step on θ , we can optionally take k steps on ϕ , since we need the classifier to remain near optimal.
- Note that to compute $\nabla_{\theta} V(\phi, \theta)$, it is necessary to backprop all the way through d before computing the partial derivatives with respect to g 's internals.



(Goodfellow et al, 2014)

GAN Lab

Data Distribution

Epoch
003,786

MODEL OVERVIEW GRAPH

LAYERED DISTRIBUTIONS

Each dot is a 2D data sample: real samples fake samples.

Background colors of grid cells represent **discriminator's** classifications. Samples in **green regions** are likely to be real; those in **purple regions** likely fake.

Manifold represents **generator's** transformation results from noise space. Opacity encodes density; darker purple means more samples in smaller area.

Pink lines from fake samples represent **gradients** for generator.

This sample needs to move upper right to decrease generator's loss.

METRICS

■ Discriminator's Loss
■ Generator's Loss

■ KL Divergence (by grid)
■ JS Divergence (by grid)

Demo: GAN Lab

Game analysis

Let us consider the value function $V(\phi, \theta)$.

- For a fixed g , $V(\phi, \theta)$ is high if d is good at recognizing true from generated samples.
- If d is the best classifier given g , and if V is high, then this implies that the generator is bad at reproducing the data distribution.
- Conversely, g will be a good generative model if V is low when d is a perfect opponent.

Therefore, the ultimate goal is

$$\theta^* = \arg \min_{\theta} \max_{\phi} V(\phi, \theta).$$

For a generator g fixed at θ , the classifier d with parameters ϕ_θ^* is optimal if and only if

$$\forall \mathbf{x}, d(\mathbf{x}; \phi_\theta^*) = \frac{p(\mathbf{x})}{q(\mathbf{x}; \theta) + p(\mathbf{x})}.$$

Therefore,

$$\begin{aligned} \min_{\theta} \max_{\phi} V(\phi, \theta) &= \min_{\theta} V(\phi_{\theta}^*, \theta) \\ &= \min_{\theta} \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} \left[\log \frac{p(\mathbf{x})}{q(\mathbf{x}; \theta) + p(\mathbf{x})} \right] + \mathbb{E}_{\mathbf{x} \sim q(\mathbf{x}; \theta)} \left[\log \frac{q(\mathbf{x}; \theta)}{q(\mathbf{x}; \theta) + p(\mathbf{x})} \right] \\ &= \min_{\theta} \text{KL} \left(p(\mathbf{x}) \parallel \frac{p(\mathbf{x}) + q(\mathbf{x}; \theta)}{2} \right) \\ &\quad + \text{KL} \left(q(\mathbf{x}; \theta) \parallel \frac{p(\mathbf{x}) + q(\mathbf{x}; \theta)}{2} \right) - \log 4 \\ &= \min_{\theta} 2 \text{JSD}(p(\mathbf{x}) \parallel q(\mathbf{x}; \theta)) - \log 4 \end{aligned}$$

where **JSD** is the Jensen-Shannon divergence.

In summary,

$$\begin{aligned}\theta^* &= \arg \min_{\theta} \max_{\phi} V(\phi, \theta) \\ &= \arg \min_{\theta} \text{JSD}(p(\mathbf{x}) || q(\mathbf{x}; \theta)).\end{aligned}$$

Since $\text{JSD}(p(\mathbf{x}) || q(\mathbf{x}; \theta))$ is minimum if and only if

$$p(\mathbf{x}) = q(\mathbf{x}; \theta)$$

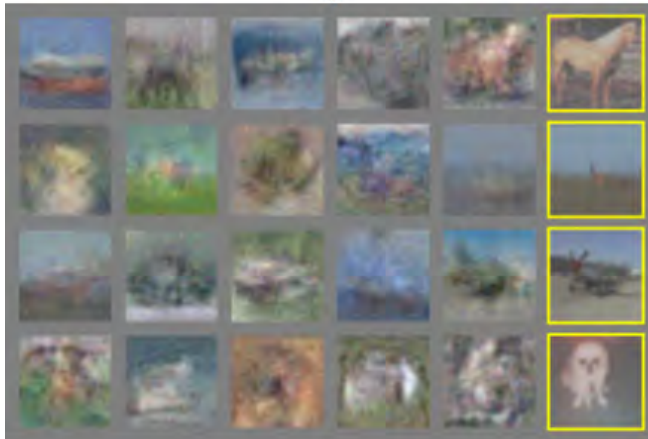
for all \mathbf{x} , this proves that the minimax solution corresponds to a generative model that perfectly reproduces the true data distribution.



a)



b)



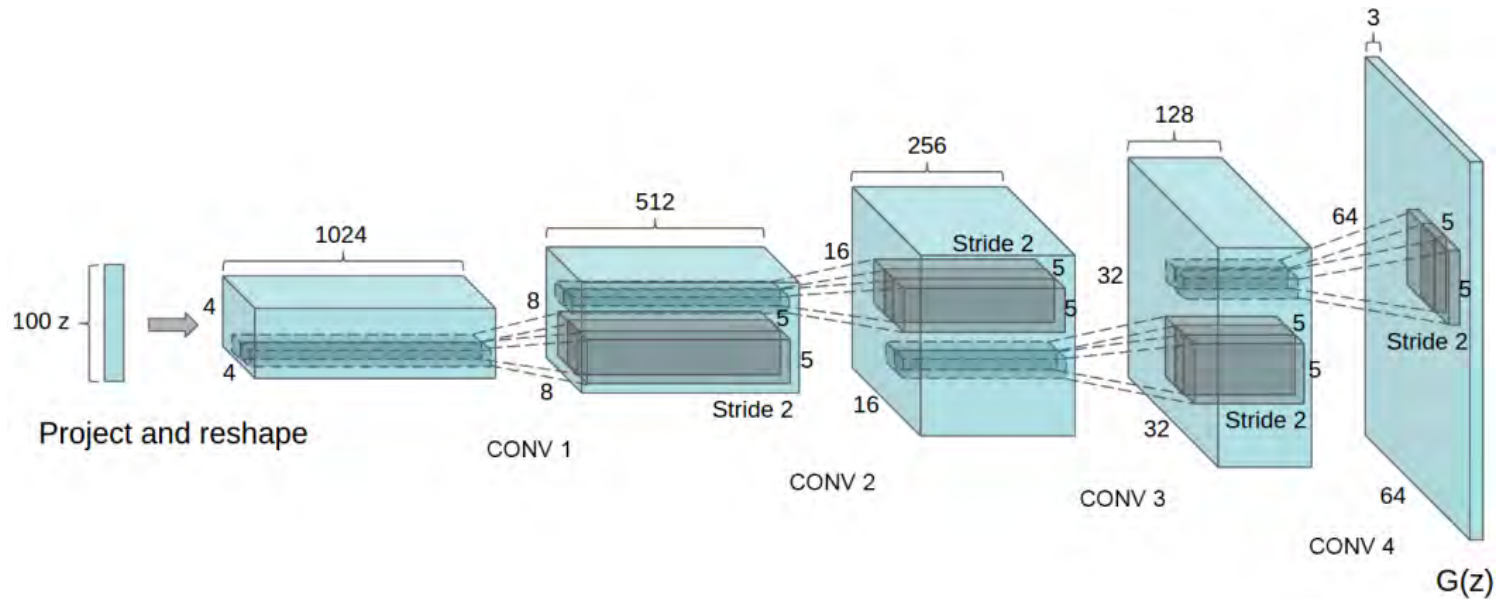
c)



d)

(Goodfellow et al, 2014)

DCGANs



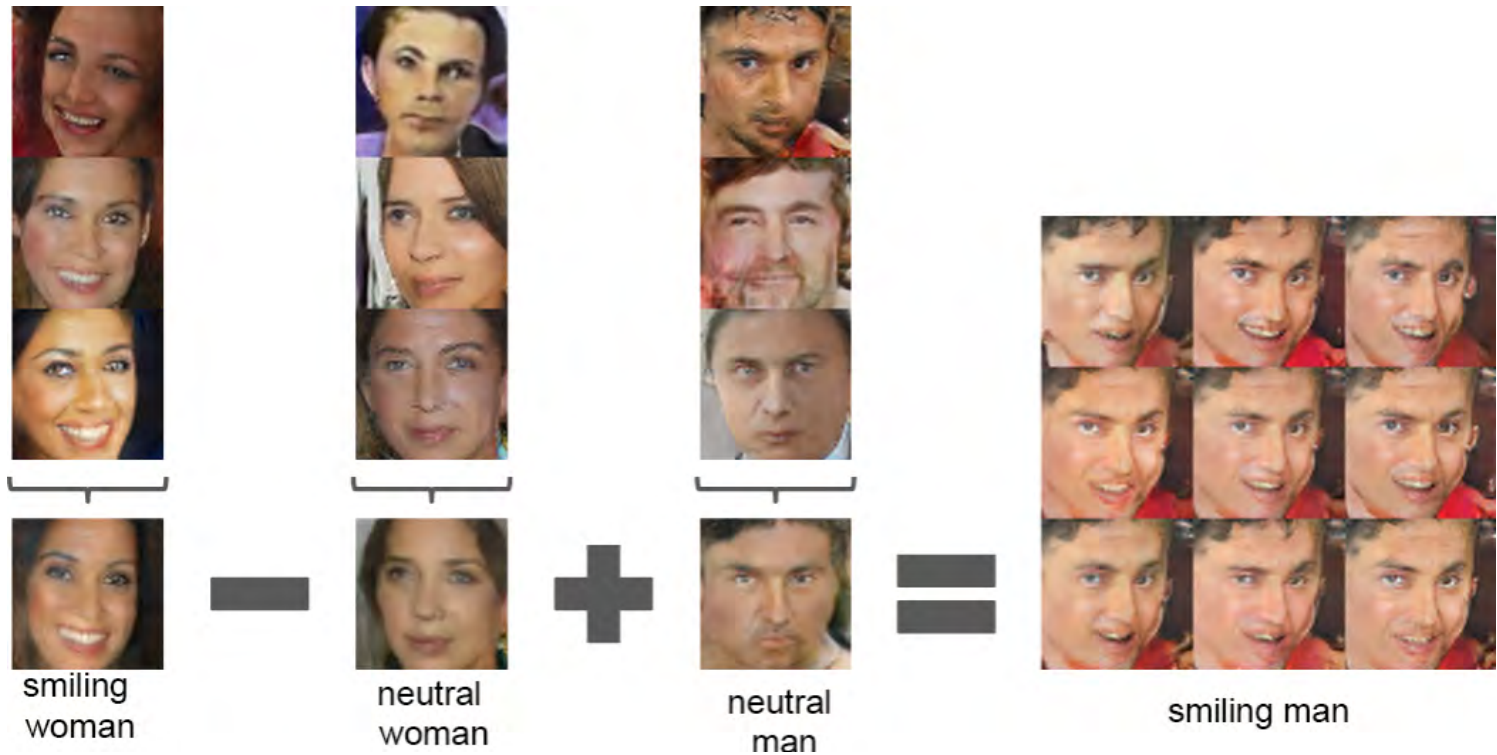
(Radford et al, 2015)



(Radford et al, 2015)



(Radford et al, 2015)

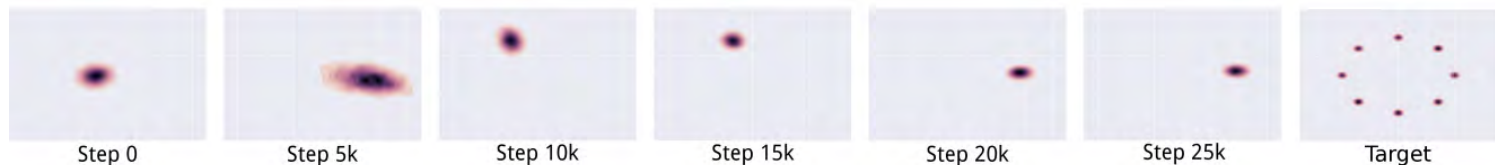


Vector arithmetic in latent space (Radford et al, 2015)

Open problems

Training a standard GAN often results in pathological behaviors:

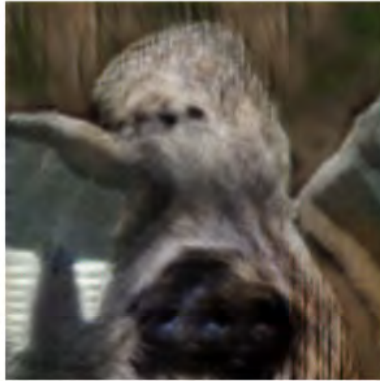
- **Oscillations** without convergence: contrary to standard loss minimization, alternating stochastic gradient descent has no guarantee of convergence.
- **Vanishing gradients**: when the classifier d is too good, the value function saturates and we end up with no gradient to update the generator.
- **Mode collapse**: the generator g models very well a small sub-population, concentrating on a few modes of the data distribution.
- Performance is also difficult to assess in practice.



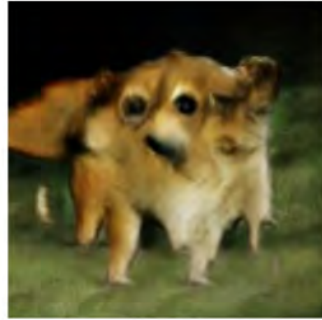
Mode collapse (Metz et al, 2016)

Cabinet of curiosities

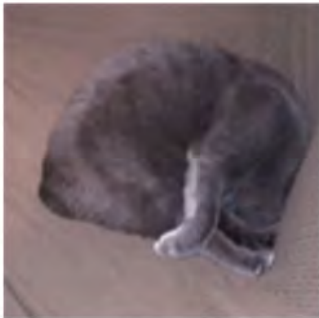
While early results (2014-2016) were already impressive, a close inspection of the fake samples distribution $q(\mathbf{x}; \theta)$ often revealed fundamental issues highlighting architectural limitations.



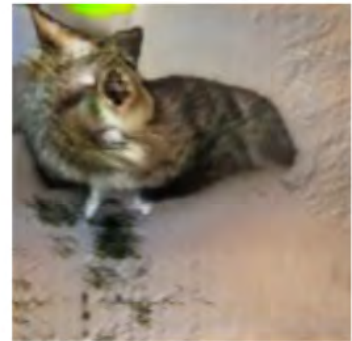
Cherry-picks (Goodfellow, 2016)



Problems with counting (Goodfellow, 2016)



Problems with perspective (Goodfellow, 2016)



Problems with global structures (Goodfellow, 2016)

Wasserstein GANs

Return of the Vanishing Gradients

For most non-toy data distributions, the fake samples $\mathbf{x} \sim q(\mathbf{x}; \theta)$ may be so bad initially that the response of d saturates.

At the limit, when d is perfect given the current generator g ,

$$\begin{aligned}d(\mathbf{x}; \phi) &= 1, \forall \mathbf{x} \sim p(\mathbf{x}), \\d(\mathbf{x}; \phi) &= 0, \forall \mathbf{x} \sim q(\mathbf{x}; \theta).\end{aligned}$$

Therefore,

$$V(\phi, \theta) = \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} [\log d(\mathbf{x}; \phi)] + \mathbb{E}_{\mathbf{z} \sim p(\mathbf{z})} [\log(1 - d(g(\mathbf{z}; \theta); \phi))] = 0$$

and $\nabla_{\theta} V(\phi, \theta) = 0$, thereby **halting** gradient descent.

Dilemma

- If d is bad, then g does not have accurate feedback and the loss function cannot represent the reality.
- If d is too good, the gradients drop to 0, thereby slowing down or even halting the optimization.

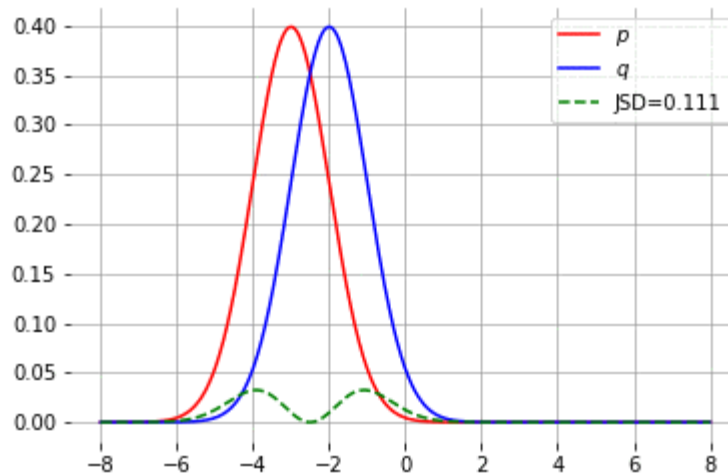
Jensen-Shannon divergence

For any two distributions p and q ,

$$0 \leq JSD(p||q) \leq \log 2,$$

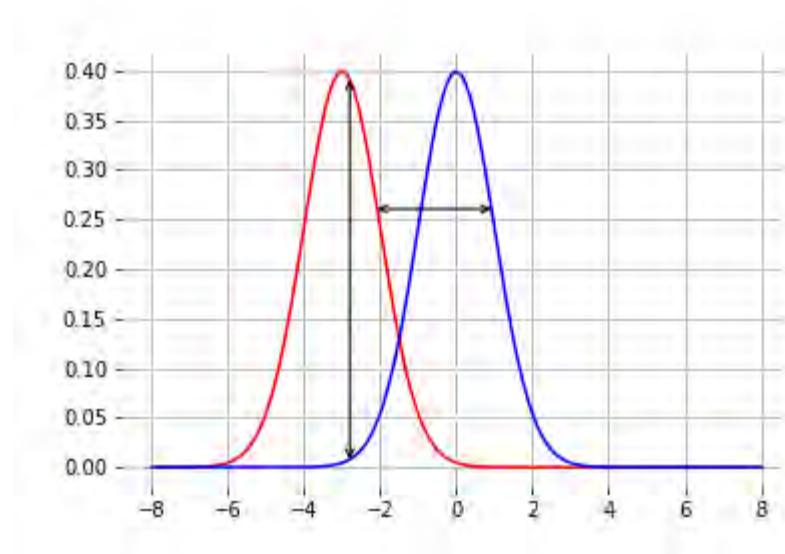
where

- $JSD(p||q) = 0$ if and only if $p = q$,
- $JSD(p||q) = \log 2$ if and only if p and q have disjoint supports.



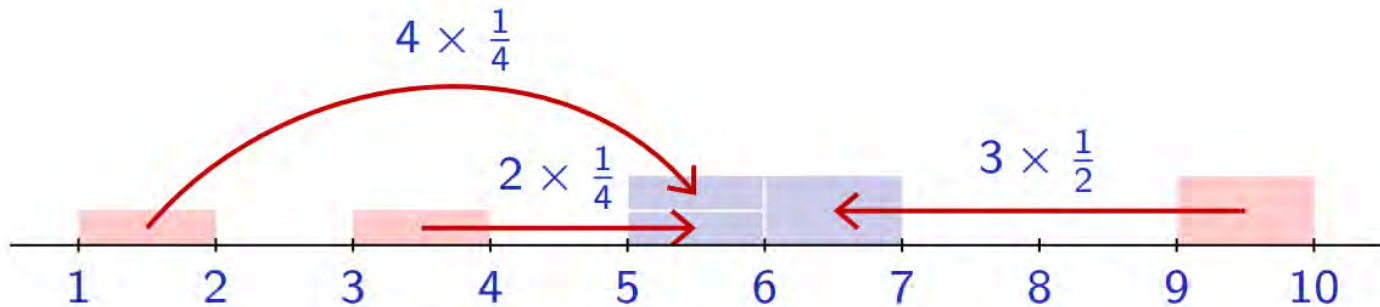
Notice how the Jensen-Shannon divergence poorly accounts for the metric structure of the space.

Intuitively, instead of comparing distributions "vertically", we would like to compare them "horizontally".



Wasserstein distance

An alternative choice is the **Earth mover's distance**, which intuitively corresponds to the minimum mass displacement to transform one distribution into the other.



- $p = \frac{1}{4}\mathbf{1}_{[1,2]} + \frac{1}{4}\mathbf{1}_{[3,4]} + \frac{1}{2}\mathbf{1}_{[9,10]}$
- $q = \mathbf{1}_{[5,7]}$

Then,

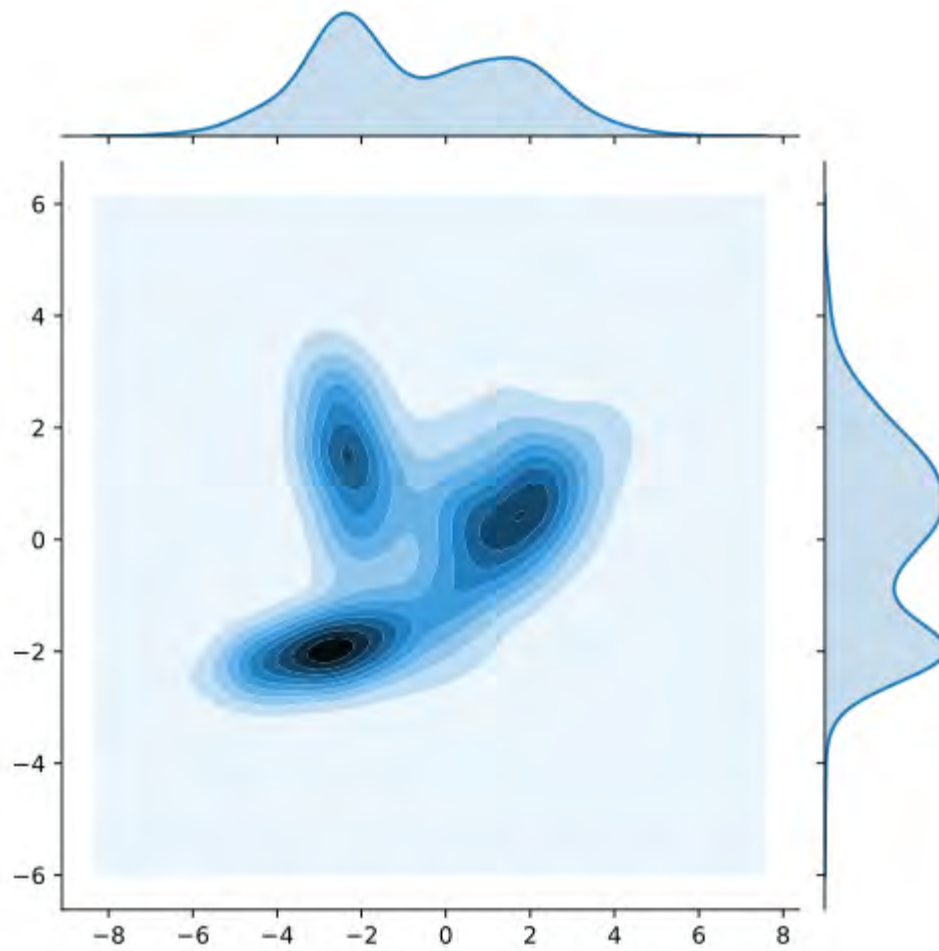
$$W_1(p, q) = 4 \times \frac{1}{4} + 2 \times \frac{1}{4} + 3 \times \frac{1}{2} = 3$$

The Earth mover's distance is also known as the Wasserstein-1 distance and is defined as:

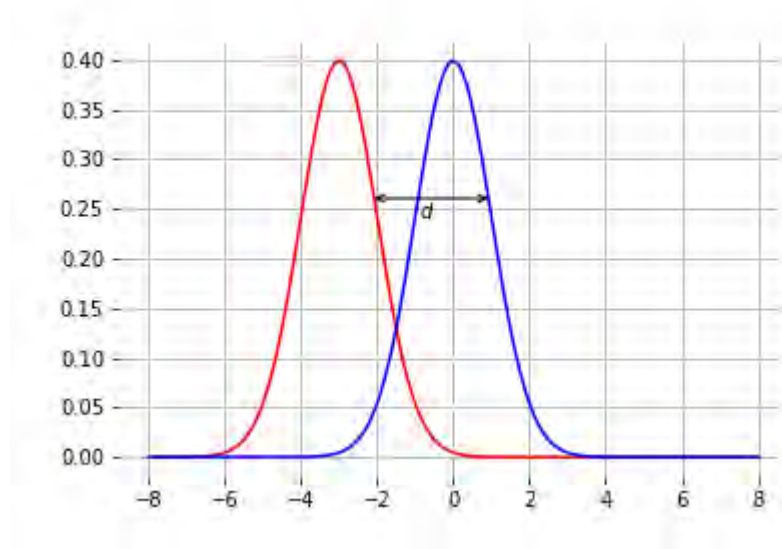
$$W_1(p, q) = \inf_{\gamma \in \Pi(p, q)} \mathbb{E}_{(x, y) \sim \gamma} [\|x - y\|]$$

where:

- $\Pi(p, q)$ denotes the set of all joint distributions $\gamma(x, y)$ whose marginals are respectively p and q ;
- $\gamma(x, y)$ indicates how much mass must be transported from x to y in order to transform the distribution p into q .
- $\|\cdot\|$ is the L1 norm and $\|x - y\|$ represents the cost of moving a unit of mass from x to y .



Notice how the W_1 distance does not saturate. Instead, it increases monotonically with the distance between modes:



$$W_1(p, q) = d$$

For any two distributions p and q ,

- $W_1(p, q) \in \mathbb{R}^+$,
- $W_1(p, q) = 0$ if and only if $p = q$.

Wasserstein GANs

Given the attractive properties of the Wasserstein-1 distance, Arjovsky et al (2017) propose to learn a generative model by solving instead:

$$\theta^* = \arg \min_{\theta} W_1(p(\mathbf{x}) || q(\mathbf{x}; \theta))$$

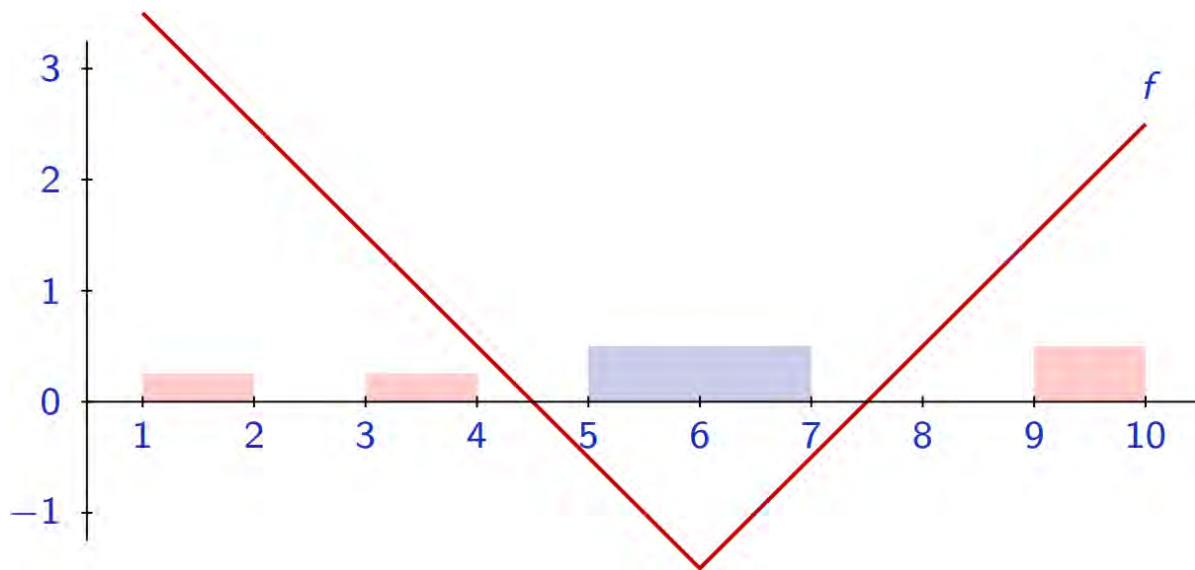
Unfortunately, the definition of W_1 does not provide with an operational way of estimating it because of the intractable \inf .

On the other hand, the Kantorovich-Rubinstein duality tells us that

$$W_1(p(\mathbf{x}) || q(\mathbf{x}; \theta)) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} [f(\mathbf{x})] - \mathbb{E}_{\mathbf{x} \sim q(\mathbf{x}; \theta)} [f(\mathbf{x})]$$

where the supremum is over all the 1-Lipschitz functions $f : \mathcal{X} \rightarrow \mathbb{R}$. That is, functions f such that

$$\|f\|_L = \max_{\mathbf{x}, \mathbf{x}'} \frac{\|f(\mathbf{x}) - f(\mathbf{x}')\|}{\|\mathbf{x} - \mathbf{x}'\|} \leq 1.$$



For $p = \frac{1}{4}\mathbf{1}_{[1,2]} + \frac{1}{4}\mathbf{1}_{[3,4]} + \frac{1}{2}\mathbf{1}_{[9,10]}$ and $q = \mathbf{1}_{[5,7]}$,

$$\begin{aligned}
 W_1(p, q) &= 4 \times \frac{1}{4} + 2 \times \frac{1}{4} + 3 \times \frac{1}{2} = 3 \\
 &= \underbrace{\left(3 \times \frac{1}{4} + 1 \times \frac{1}{4} + 2 \times \frac{1}{2} \right)}_{\mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} [f(\mathbf{x})]} - \underbrace{\left(-1 \times \frac{1}{2} - 1 \times \frac{1}{2} \right)}_{\mathbb{E}_{\mathbf{x} \sim q(\mathbf{x}; \theta)} [f(\mathbf{x})]} = 3
 \end{aligned}$$

Using this result, the Wasserstein GAN algorithm consists in solving the minimax problem:

$$\theta^* = \arg \min_{\theta} \max_{\phi: \|d(\cdot; \phi)\|_L \leq 1} \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} [d(\mathbf{x}; \phi)] - \mathbb{E}_{\mathbf{x} \sim q(\mathbf{x}; \theta)} [d(\mathbf{x}; \phi)]$$

Note that this formulation is very close to the original GANs, except that:

- The classifier $d : \mathcal{X} \rightarrow [0, 1]$ is replaced by a critic function $d : \mathcal{X} \rightarrow \mathbb{R}$ and its output is not interpreted through the cross-entropy loss;
- There is a strong regularization on the form of d . In practice, to ensure 1-Lipschitzness,
 - Arjovsky et al (2017) propose to clip the weights of the critic at each iteration;
 - Gulrajani et al (2017) add a regularization term to the loss.
- As a result, Wasserstein GANs benefit from:
 - a meaningful loss metric,
 - improved stability (no mode collapse is observed).

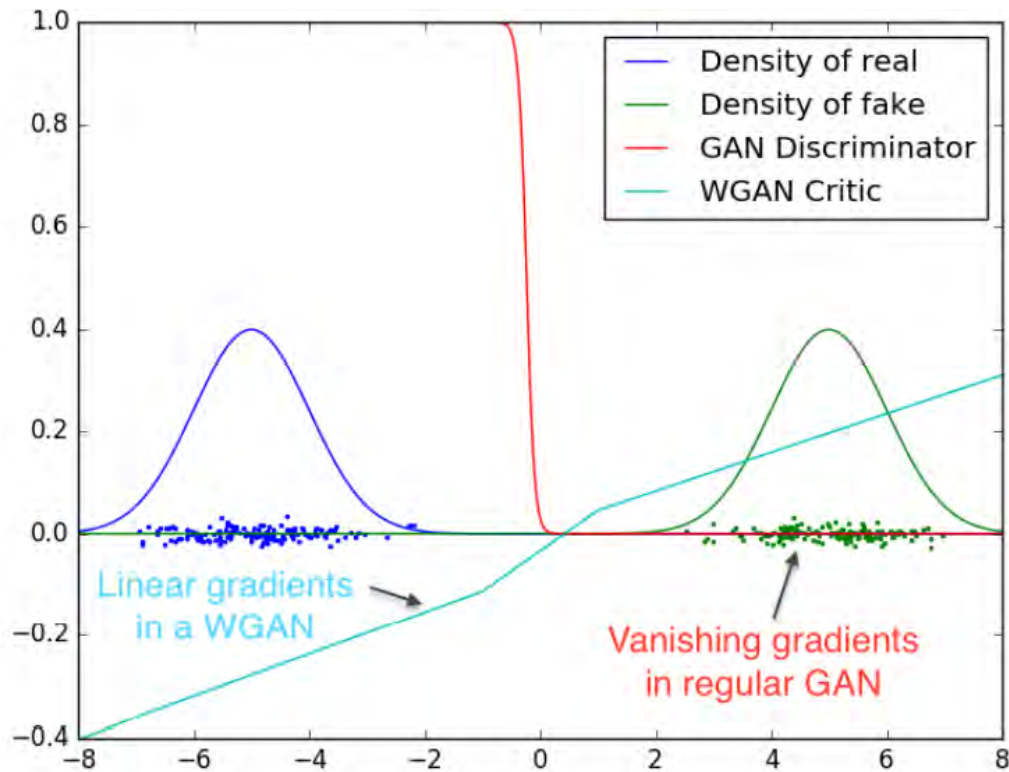
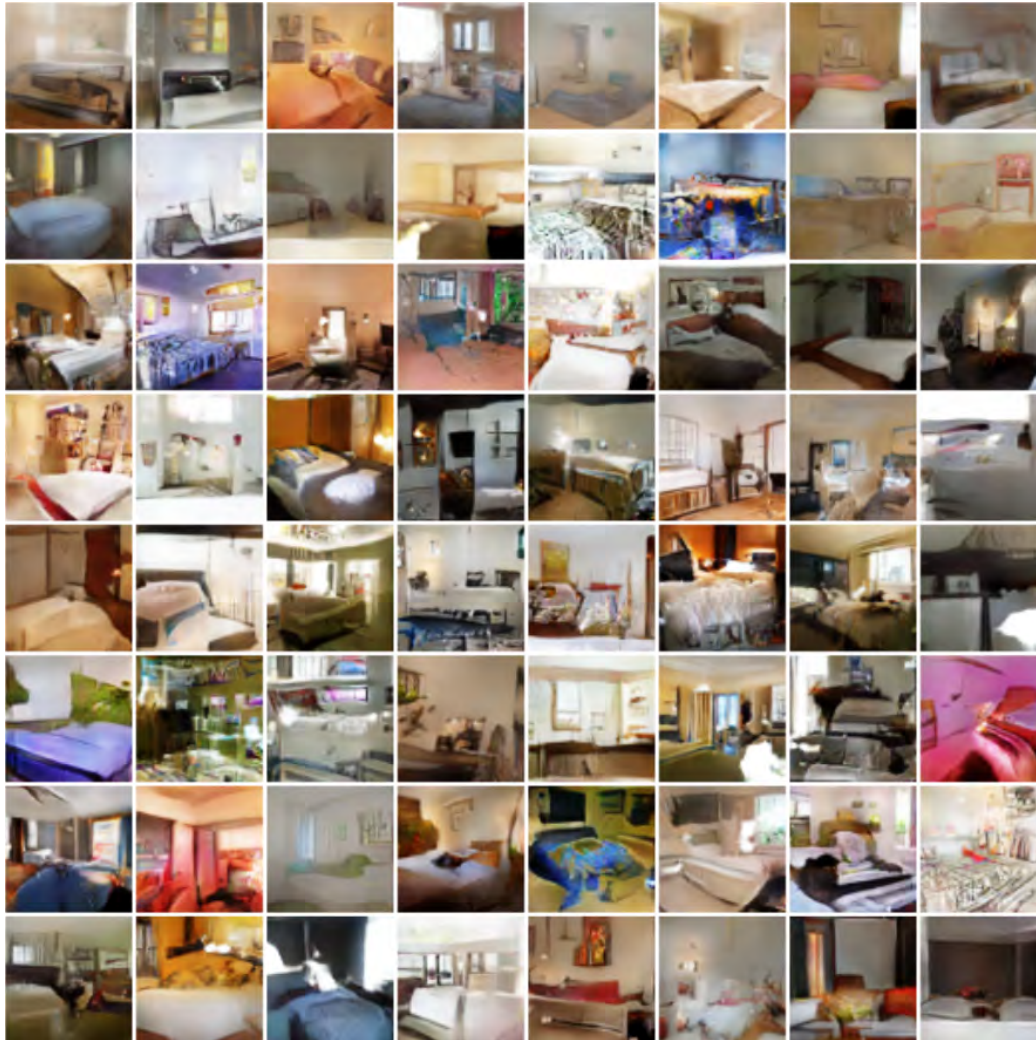


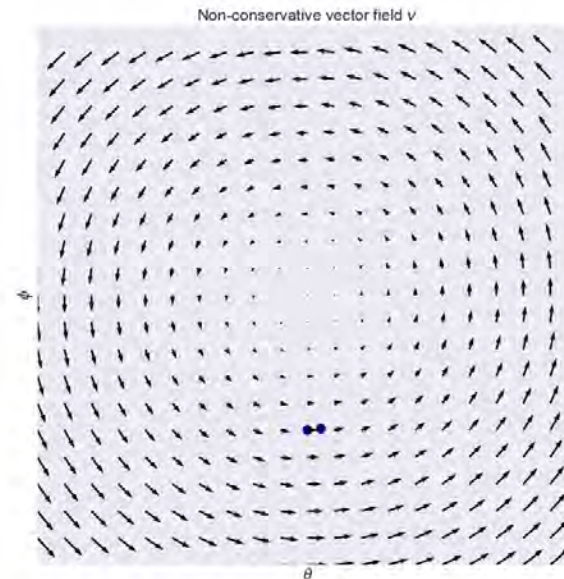
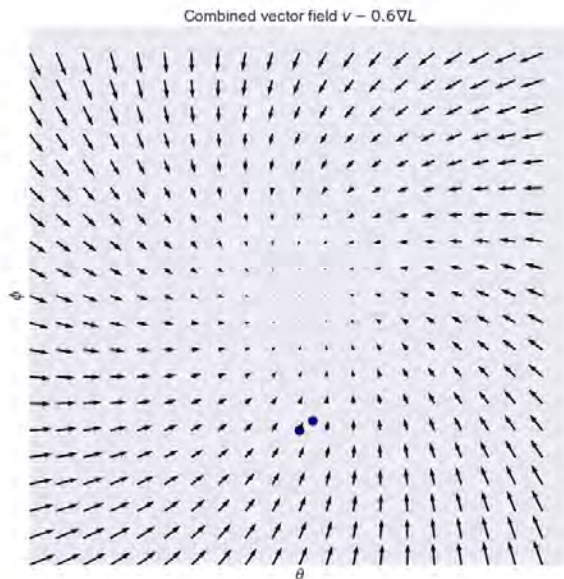
Figure 2: Optimal discriminator and critic when learning to differentiate two Gaussians. As we can see, the discriminator of a minimax GAN saturates and results in vanishing gradients. Our WGAN critic provides very clean gradients on all parts of the space.

(Arjovsky et al, 2017)



(Arjovsky et al, 2017)

Convergence of GANs



Solving for saddle points is different from gradient descent.

- Minimization problems yield **conservative** vector fields.
- Min-max saddle point problems may yield **non-conservative** vector fields.

Following the notations of Mescheder et al (2018), the training objective for the two players can be described by an objective function of the form

$$L(\theta, \phi) = \mathbb{E}_{p(\mathbf{z})} [f(d(g(\mathbf{z}; \theta); \phi))] + \mathbb{E}_{p(\mathbf{x})} [f(-d(\mathbf{x}; \phi))],$$

where the goal of the generator is to minimize the loss, whereas the discriminator tries to maximize it.

- If $f(t) = -\log(1 + \exp(-t))$, then we recover the original GAN objective.
- if $f(t) = -t$ and if we impose the Lipschitz constraint on d , then we recover Wasserstein GAN.

Training algorithms can be described as fixed points algorithms that apply some operator $F_h(\theta, \phi)$ to the parameters values (θ, ϕ) .

- For simultaneous gradient descent,

$$F_h(\theta, \phi) = (\theta, \phi) + hv(\theta, \phi)$$

where $v(\theta, \phi)$ denotes the **gradient vector field**

$$v(\theta, \phi) := \begin{pmatrix} -\nabla_{\theta}L(\theta, \phi) \\ \nabla_{\phi}L(\theta, \phi) \end{pmatrix}.$$

- Similarly, alternating gradient descent can be described by an operator $F_h = F_{2,h} \circ F_{1,h}$, where $F_{1,h}$ and $F_{2,h}$ perform an update for the generator and discriminator, respectively.

Local convergence near an equilibrium point

Let us consider the Jacobian $F'_h(\theta^*, \phi^*)$ at the equilibrium (θ^*, ϕ^*) :

- if $F'_h(\theta^*, \phi^*)$ has eigenvalues with absolute value bigger than 1, the training will generally not converge to (θ^*, ϕ^*) .
- if all eigenvalues have absolute value smaller than 1, the training will converge to (θ^*, ϕ^*) .
- if all eigenvalues values are on the unit circle, training can be convergent, divergent or neither.

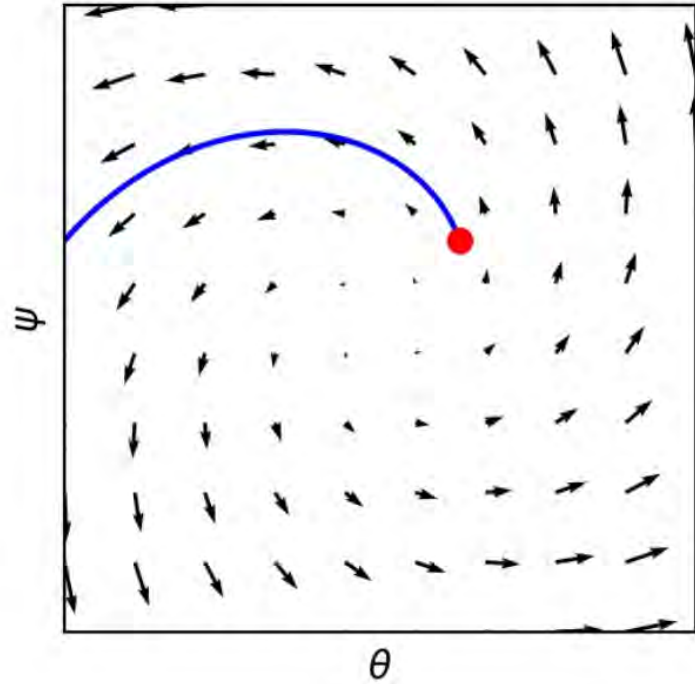
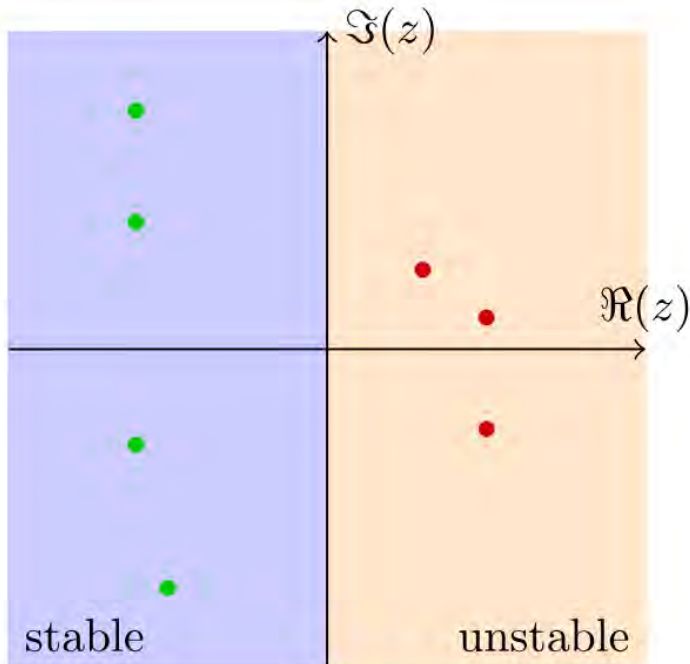
In particular, Mescheder et al (2017) show that all eigenvalues can be forced to remain within the unit ball if and only if the learning rate h is made sufficiently small.

For the (idealized) continuous system

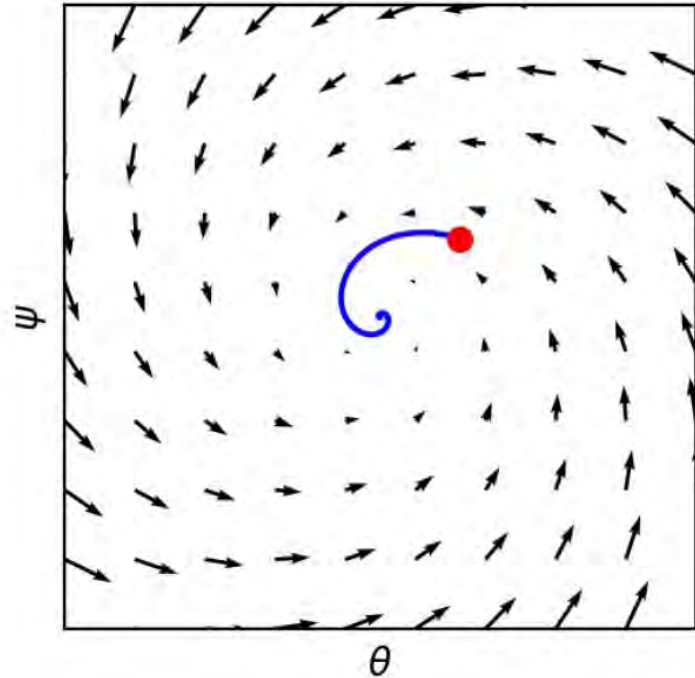
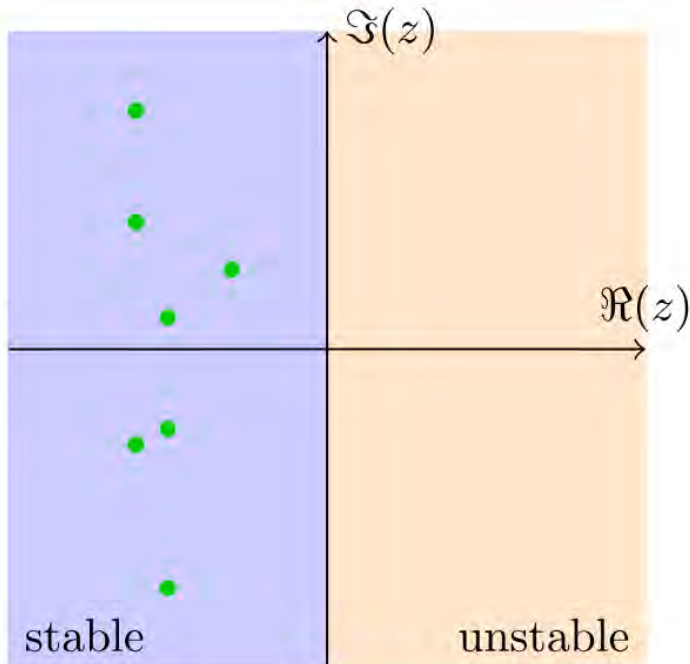
$$\begin{pmatrix} \dot{\theta}(t) \\ \dot{\phi}(t) \end{pmatrix} = \begin{pmatrix} -\nabla_{\theta} L(\theta, \phi) \\ \nabla_{\phi} L(\theta, \phi) \end{pmatrix},$$

which corresponds to training GANs with infinitely small learning rate $h \rightarrow 0$:

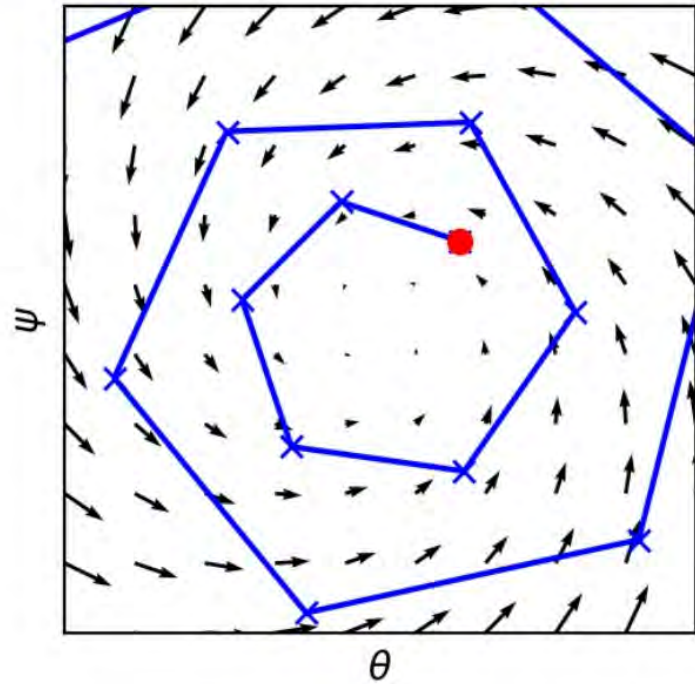
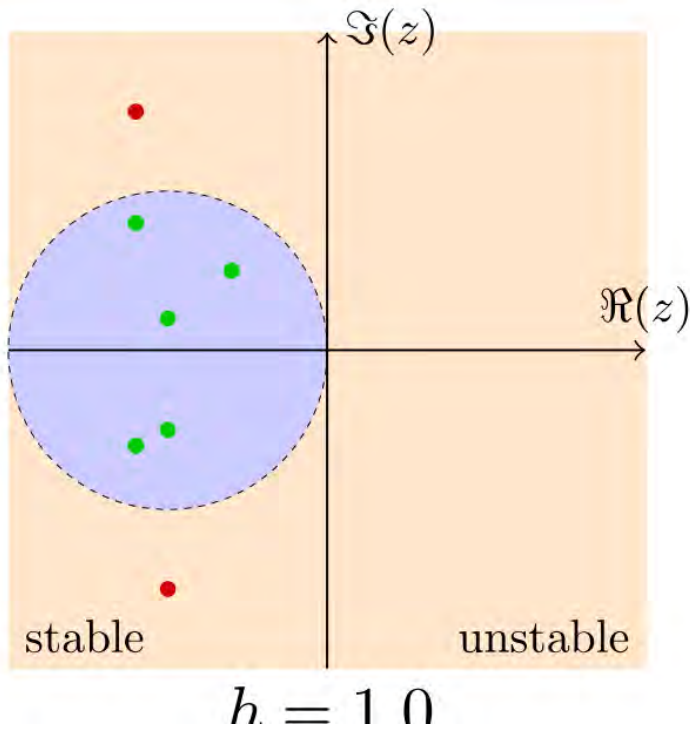
- if all eigenvalues of the Jacobian $v'(\theta^*, \phi^*)$ at a stationary point (θ^*, ϕ^*) have negative real-part, the continuous system converges locally to (θ^*, ϕ^*) ;
- if $v'(\theta^*, \phi^*)$ has eigenvalues with positive real-part, the continuous system is not locally convergent.
- if all eigenvalues have zero real-part, it can be convergent, divergent or neither.



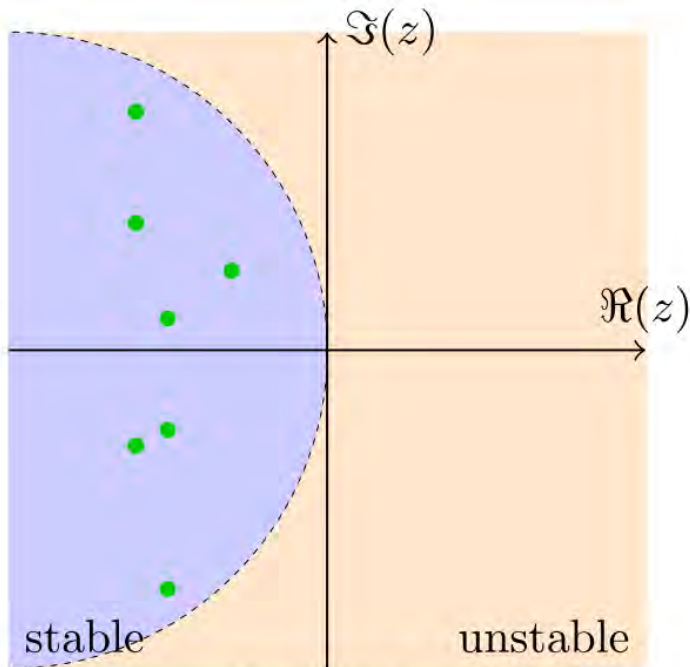
Continuous system: divergence.



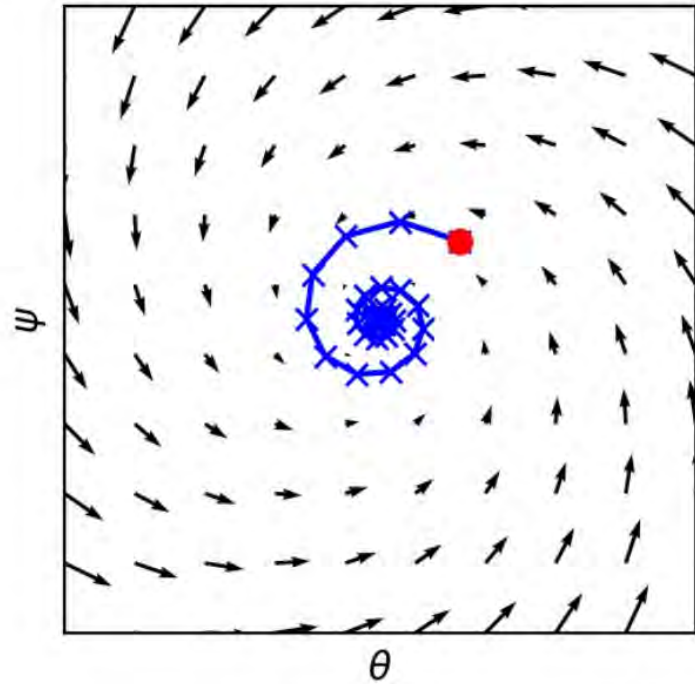
Continuous system: convergence.



Discrete system: divergence ($h = 1$, too large).

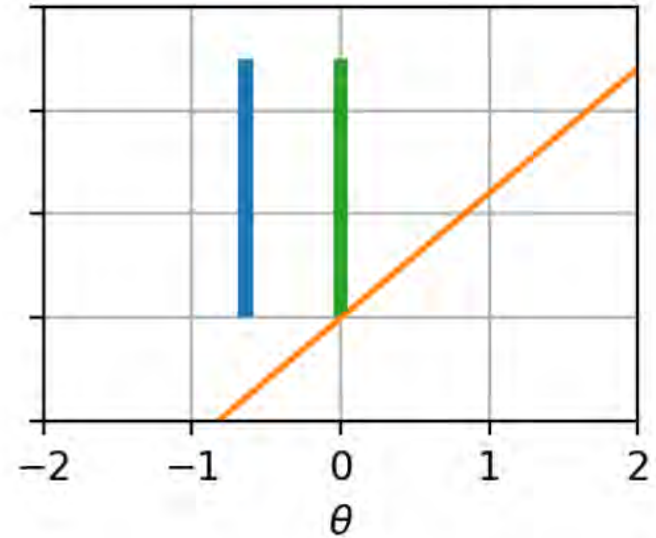
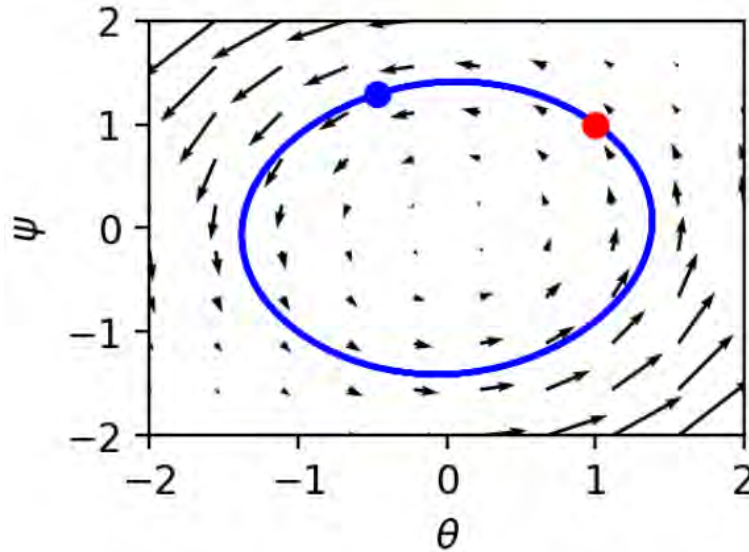


$$h = 0.5$$



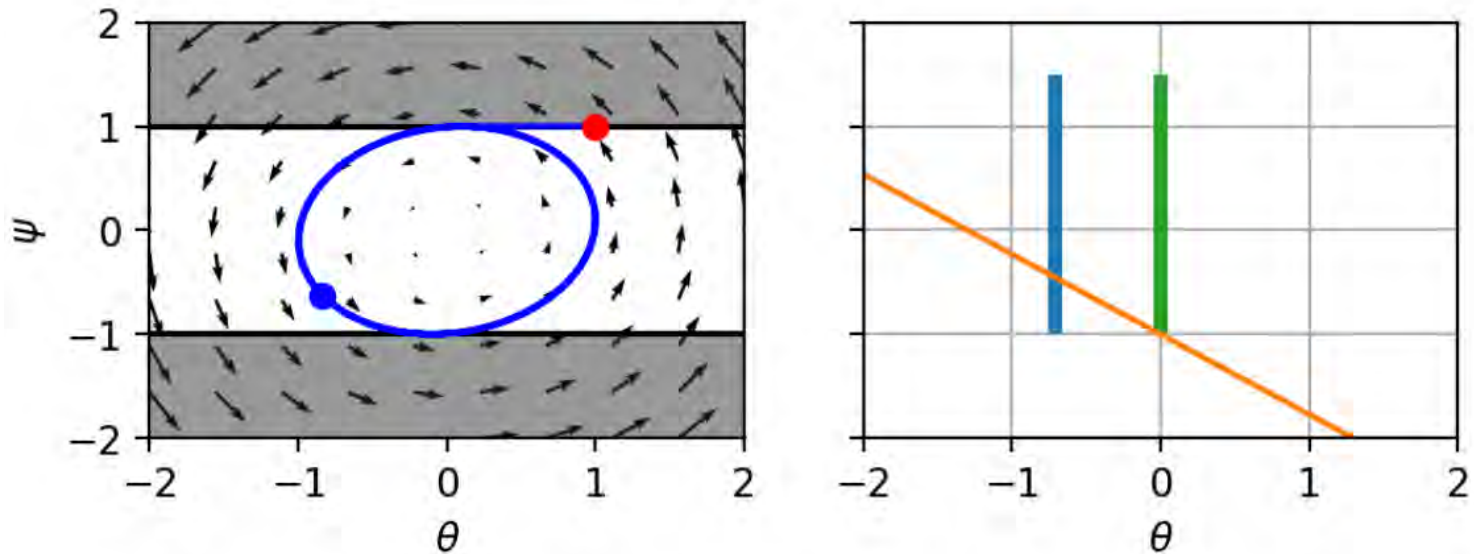
Discrete system: convergence ($h = 0.5$, small enough).

Dirac-GAN: Vanilla GANs



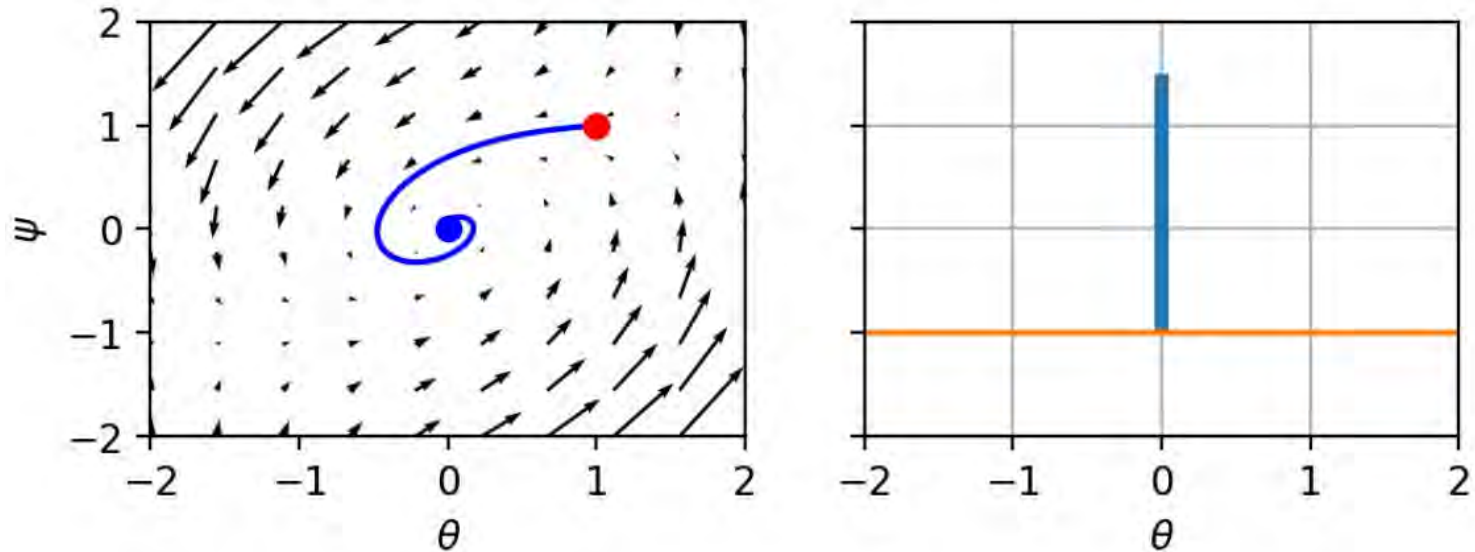
On the Dirac-GAN toy problem, eigenvalues are $\{-f'(0)i, +f'(0)i\}$. Therefore convergence is not guaranteed.

Dirac-GAN: Wasserstein GANs



Eigenvalues are $\{-i, +i\}$. Therefore convergence is not guaranteed.

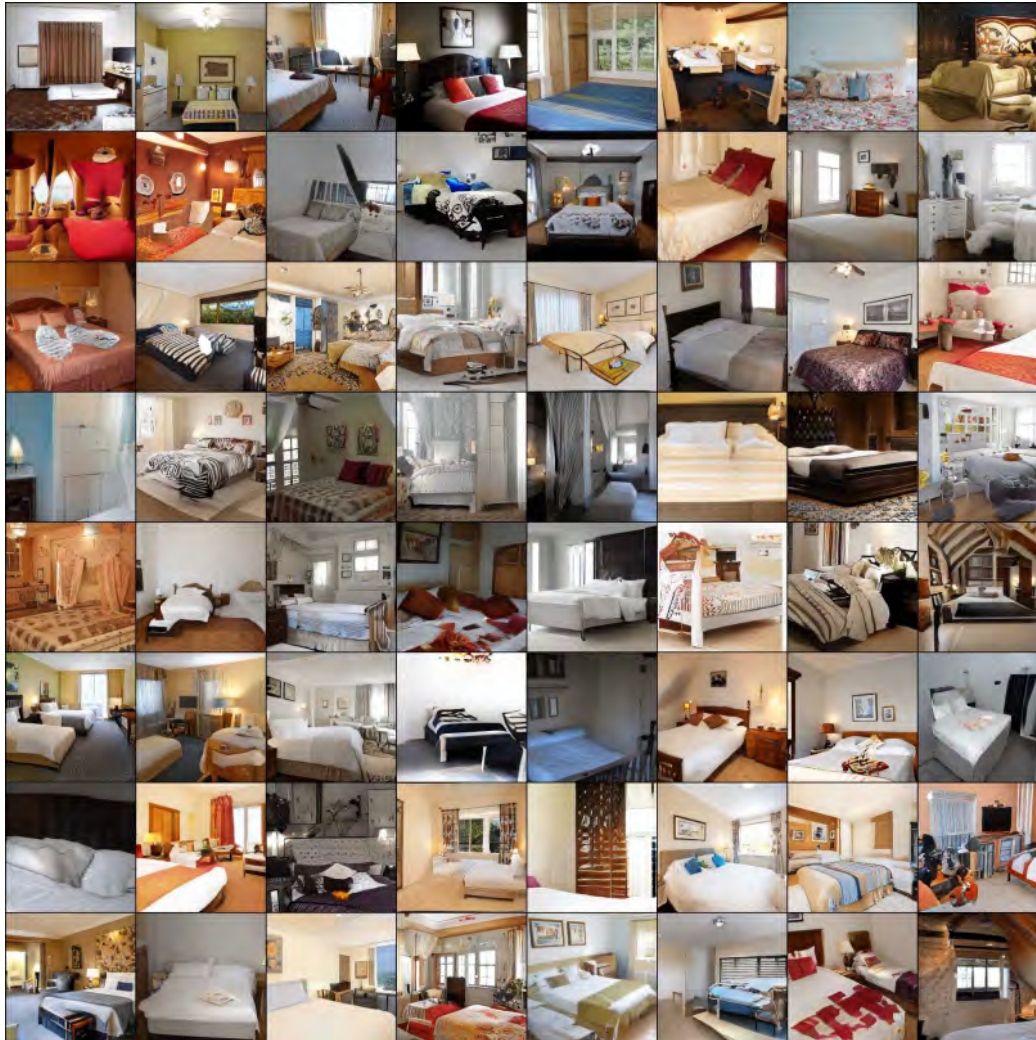
Dirac-GAN: Zero-centered gradient penalties



A penalty on the squared norm of the gradients of the discriminator results in the regularization

$$R_1(\phi) = \frac{\gamma}{2} \mathbb{E}_{\mathbf{x} \sim p(\mathbf{x})} [\|\nabla_{\mathbf{x}} d(\mathbf{x}; \phi)\|^2].$$

The resulting eigenvalues are $\{-\frac{\gamma}{2} \pm \sqrt{\frac{\gamma}{4} - f'(0)^2}\}$. Therefore, for $\gamma > 0$, all eigenvalues have negative real part, hence training is locally convergent!







State of the art



Ian Goodfellow

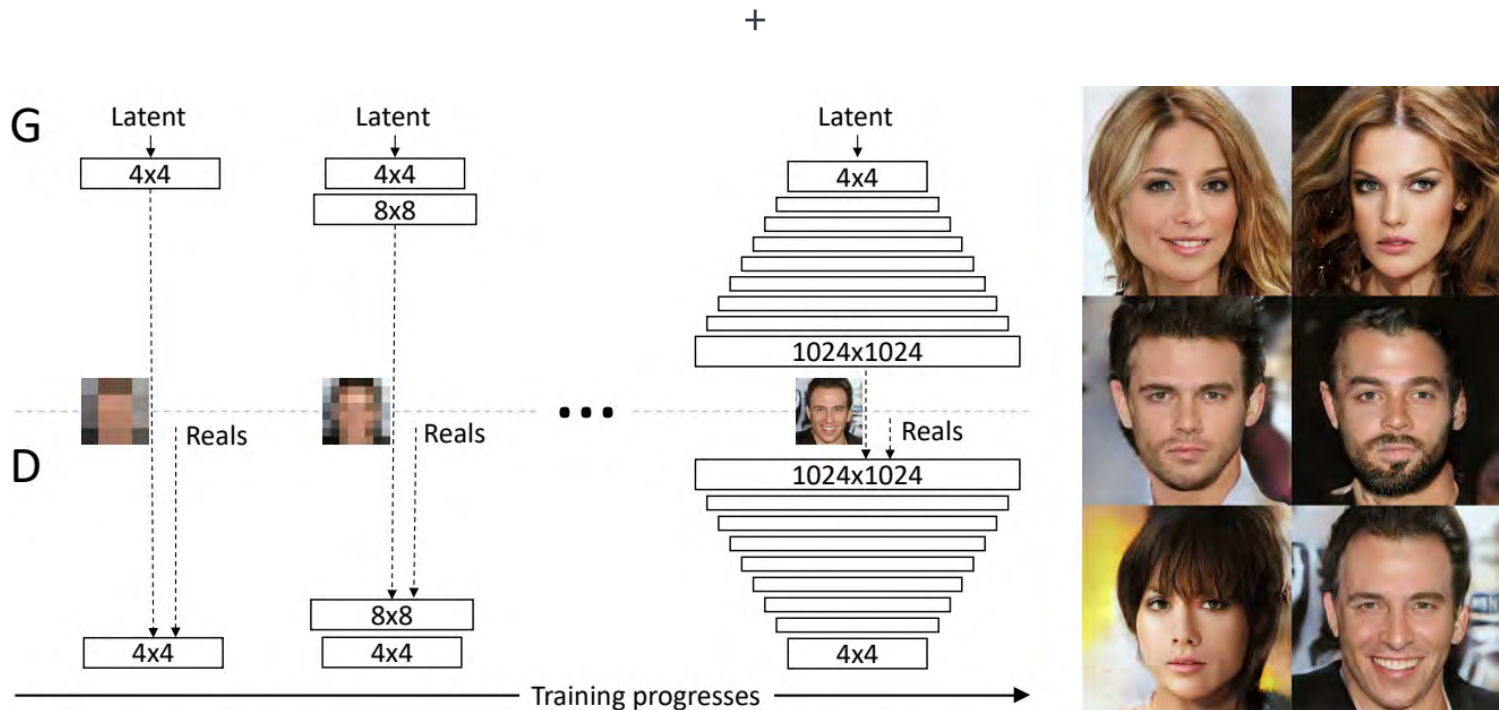
@goodfellow_ian

4.5 years of GAN progress on face generation. arxiv.org/abs/1406.2661
arxiv.org/abs/1511.06434
arxiv.org/abs/1606.07536
arxiv.org/abs/1710.10196
arxiv.org/abs/1812.04948



Progressive growing of GANs

Wasserstein GANs as baseline (Arjovsky et al, 2017) +
Gradient Penalty (Gulrajani, 2017) + (quite a few other tricks)



(Karras et al, 2017)

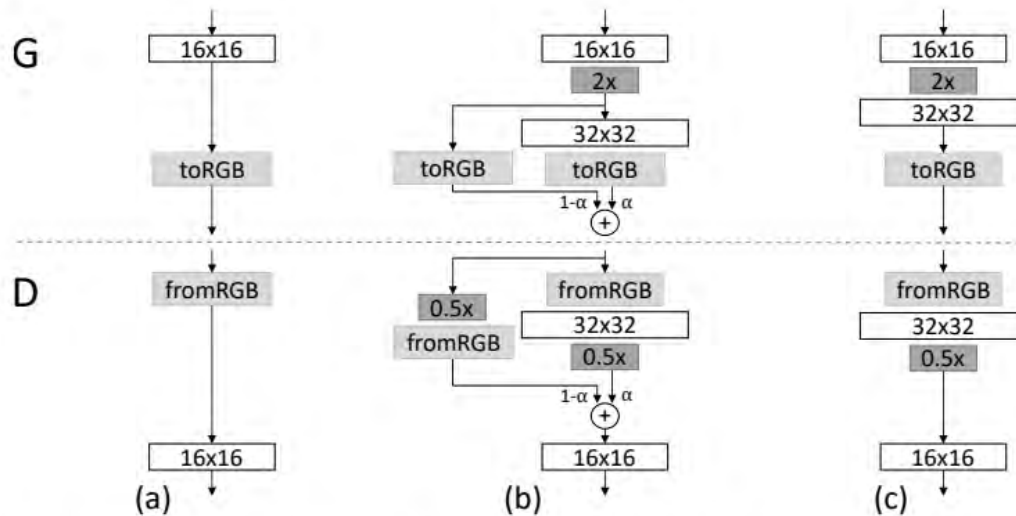


Figure 2: When doubling the resolution of the generator (G) and discriminator (D) we fade in the new layers smoothly. This example illustrates the transition from 16×16 images (a) to 32×32 images (c). During the transition (b) we treat the layers that operate on the higher resolution like a residual block, whose weight α increases linearly from 0 to 1. Here $2\times$ and $0.5\times$ refer to doubling and halving the image resolution using nearest neighbor filtering and average pooling, respectively. The `toRGB` represents a layer that projects feature vectors to RGB colors and `fromRGB` does the reverse; both use 1×1 convolutions. When training the discriminator, we feed in real images that are downsampled to match the current resolution of the network. During a resolution transition, we interpolate between two resolutions of the real images, similarly to how the generator output combines two resolutions.

(Karras et al, 2017)



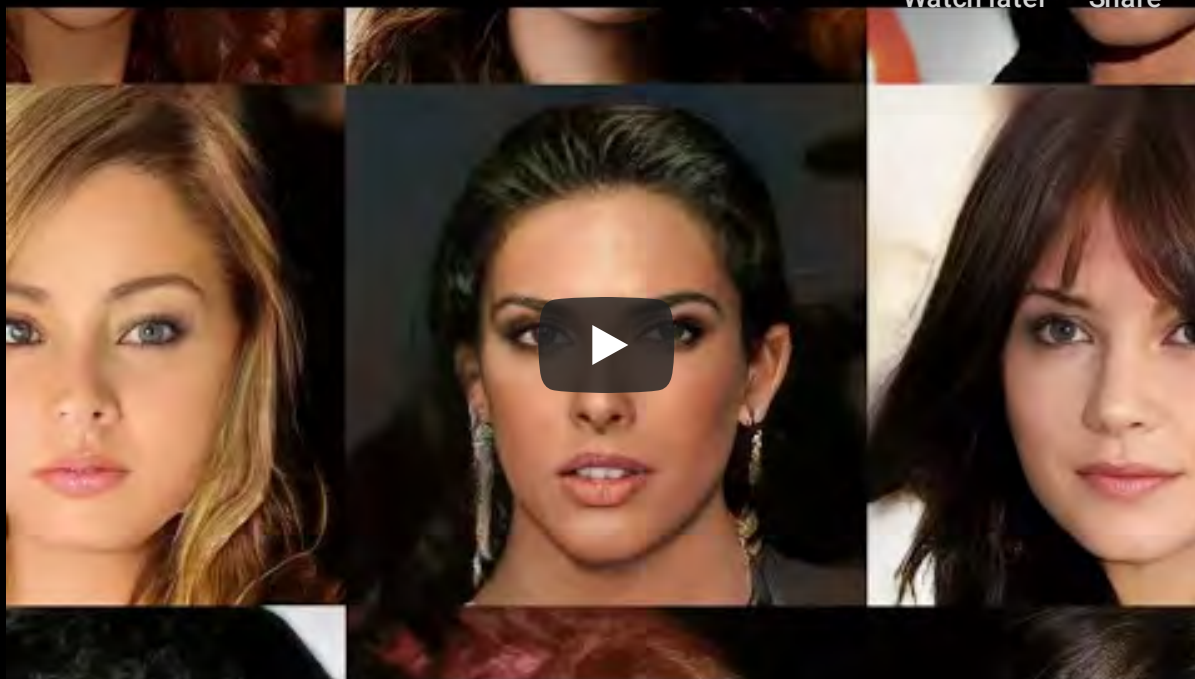
Progressive Growing of GANs for Improved Q...



Watch later



Share



(Karras et al, 2017)

BigGANs

Self-attention GANs as baseline (Zhang et al, 2018) + Hinge loss objective (Lim and Ye, 2017; Tran et al, 2017) + Class information to g with class-conditional batchnorm (de Vries et al, 2017) + Class information to d with projection (Miyato and Koyama, 2018) + Half the learning rate of SAGAN, 2 d -steps per g -step + Spectral normalization for both g and d + Orthogonal initialization (Saxe et al, 2014) + Large minibatches (2048) + Large number of convolution filters + Shared embedding and hierarchical latent spaces + Orthogonal regularization + Truncated sampling + (quite a few other tricks)



(Brock et al, 2018)



The 1000 ImageNet Categories inside of BigG...



Watch later



Share

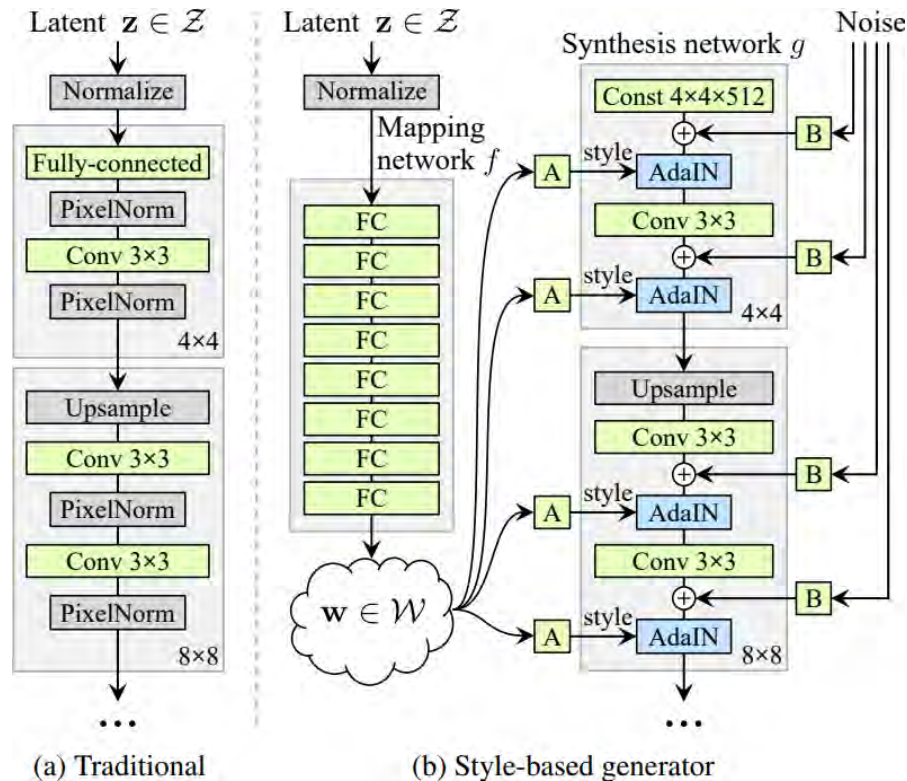


(Brock et al, 2018)

StyleGAN

Progressive GANs as baseline (Karras et al, 2017) + Non-saturating loss instead of WGAN-GP + R_1 regularization (Mescheder et al, 2018) + (quite a few other tricks)

+





A Style-Based Generator Architecture for Gen...



Watch later



Share



(Karras et al, 2018)

The StyleGAN generator g is so powerful that it can re-generate arbitrary faces.



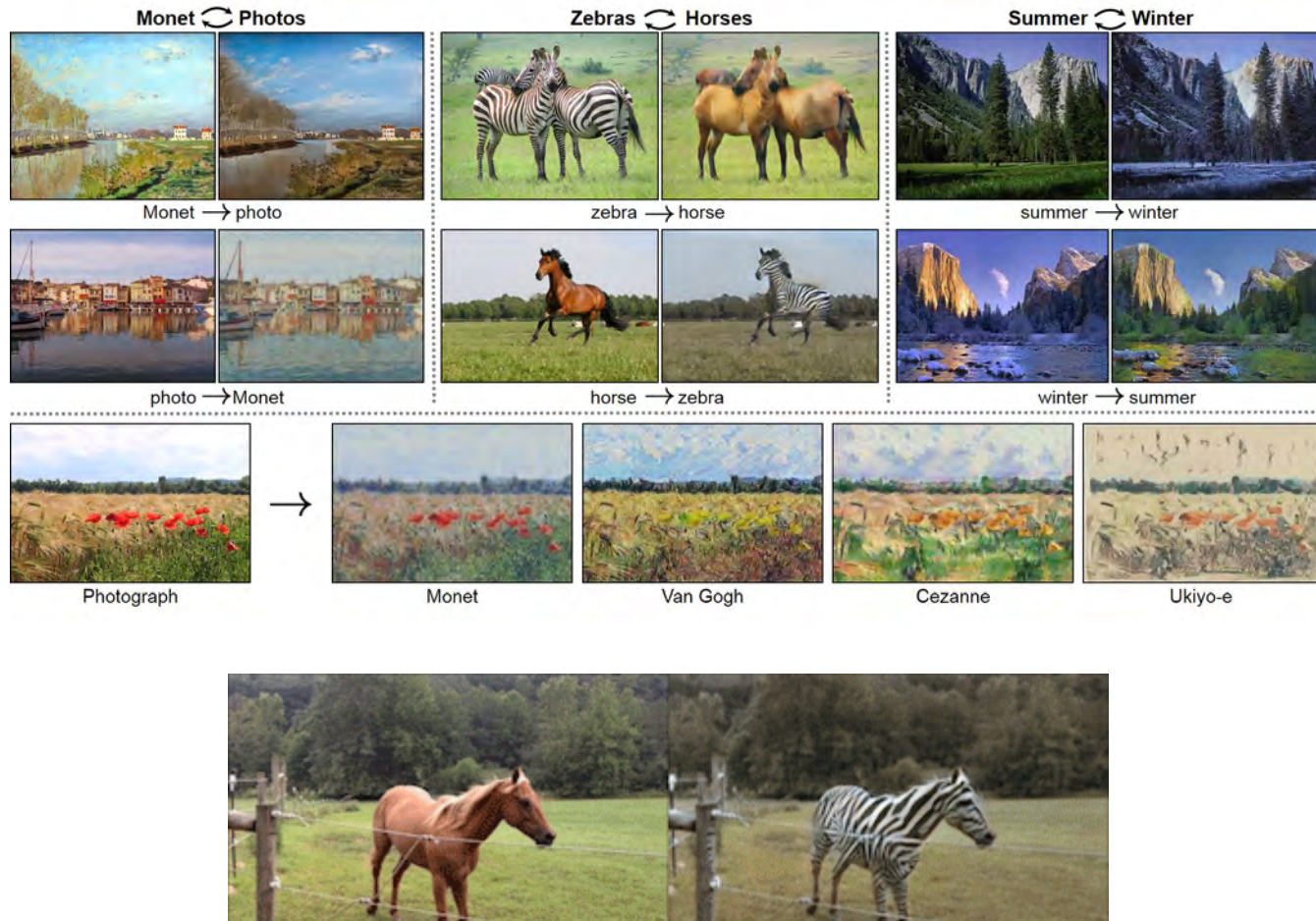




Applications

$p(\mathbf{z})$ need not be a random noise distribution.

Image-to-image translation



CycleGANs (Zhu et al, 2017)



High-Resolution Image Synthesis and Semant...



Watch later



Share



High-resolution image synthesis (Wang et al, 2017)



GauGAN: Changing Sketches into Photorealis...



Watch later



Share



GauGAN: Changing sketches into photorealistic masterpieces (NVIDIA, 2019)

Captioning



a tennis player gets ready to return a serve



two men dressed in costumes and holding tennis rackets



a tennis player hits the ball during a match



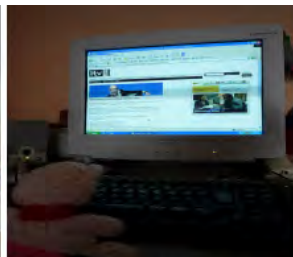
a male tennis player in action on the court



a man in white is about to serve a tennis ball



a laptop and a desktop computer sit on a desk



a person is working on a computer screen



a cup of coffee sitting next to a laptop



a laptop computer sitting on top of a desk next to a



a picture of a computer on a desk

(Shetty et al, 2017)

Text-to-image synthesis

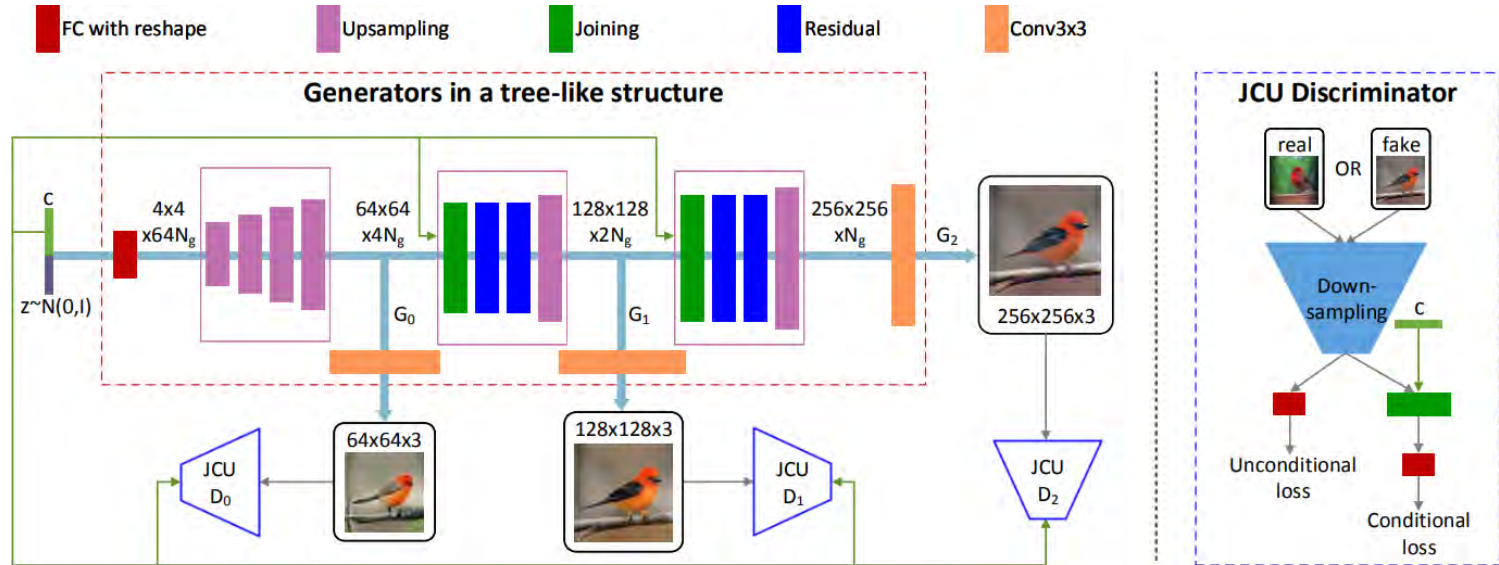


Fig. 2: The overall framework of our proposed StackGAN-v2 for the conditional image synthesis task. c is the vector of conditioning variables which can be computed from the class label, the text description, etc.. N_g and N_d are the numbers of channels of a tensor.

(Zhang et al, 2017)



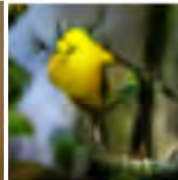



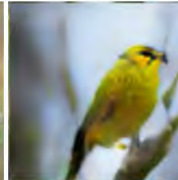
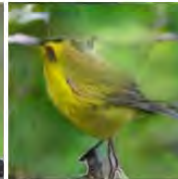

Text description	This bird is red and brown in color, with a stubby beak	The bird is short and stubby with yellow on its body	A bird with a medium orange bill white body gray wings and webbed feet	This small black bird has a short, slightly curved bill and long legs	A small bird with varying shades of brown with white under the eyes	A small yellow bird with a black crown and a short black pointed beak	This small bird has a white breast, light grey head, and black wings and tail
64x64 GAN-INT-CLS							
128x128 GAWWN							
256x256 StackGAN-v1							

Fig. 3: Example results by our StackGAN-v1, GAWWN [29], and GAN-INT-CLS [31] conditioned on text descriptions from CUB test set.

(Zhang et al, 2017)

Music generation

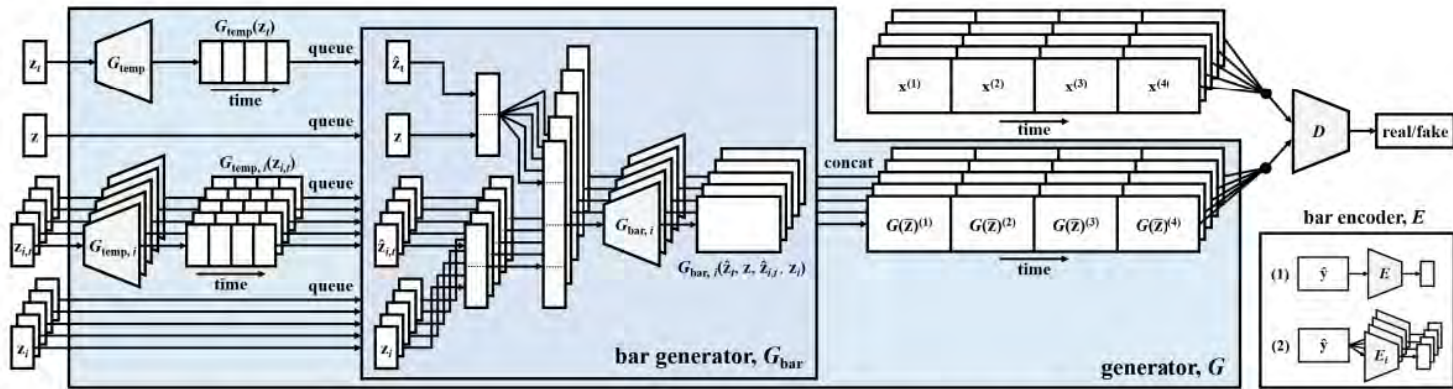


Figure 5: System diagram of the proposed MuseGAN model for multi-track sequential data generation.

▶ 0:00 / 3:15 ● 🔊

MuseGAN (Dong et al, 2018)

Accelerating scientific simulators

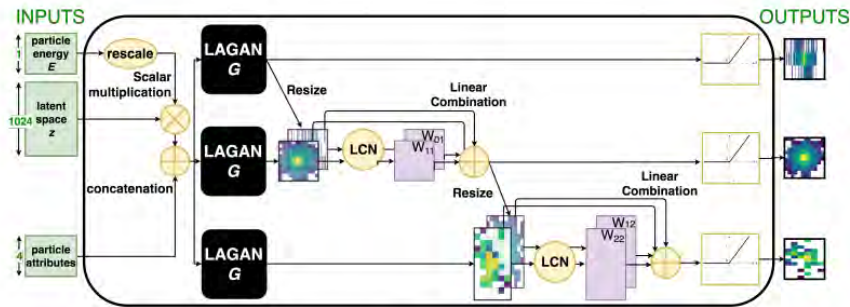


Figure 8.37: Composite conditional CaloGAN generator G , with three LAGAN-like streams connected by attentional layer-to-layer dependence.

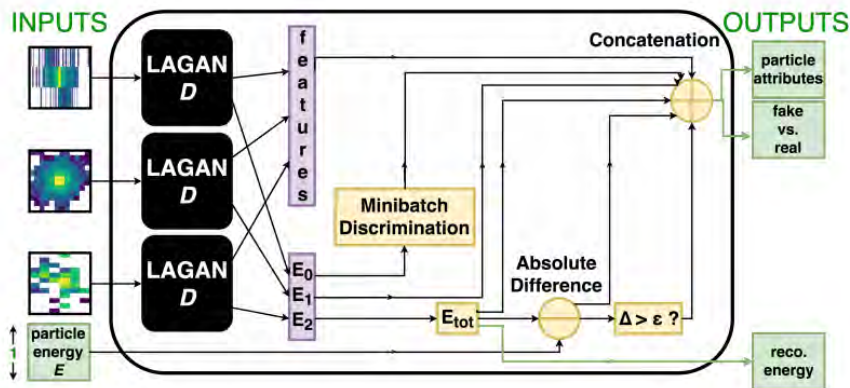
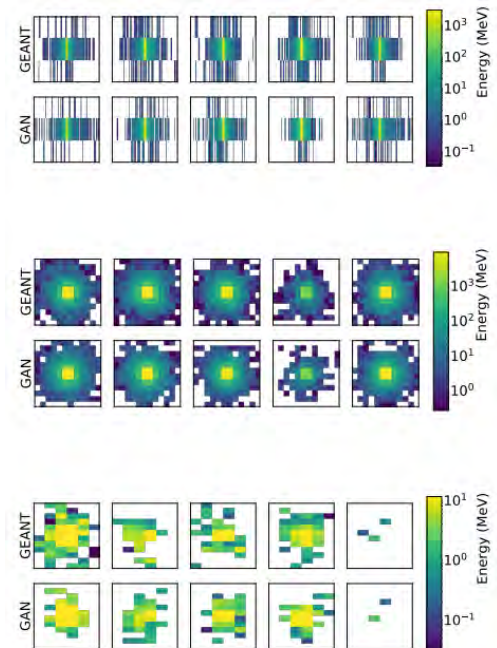


Figure 8.38: Composite conditional CaloGAN discriminator D , with three LAGAN-like streams and additional domain-specific energy calculations included in the final feature space.



Learning particle physics (Paganini et al, 2017)

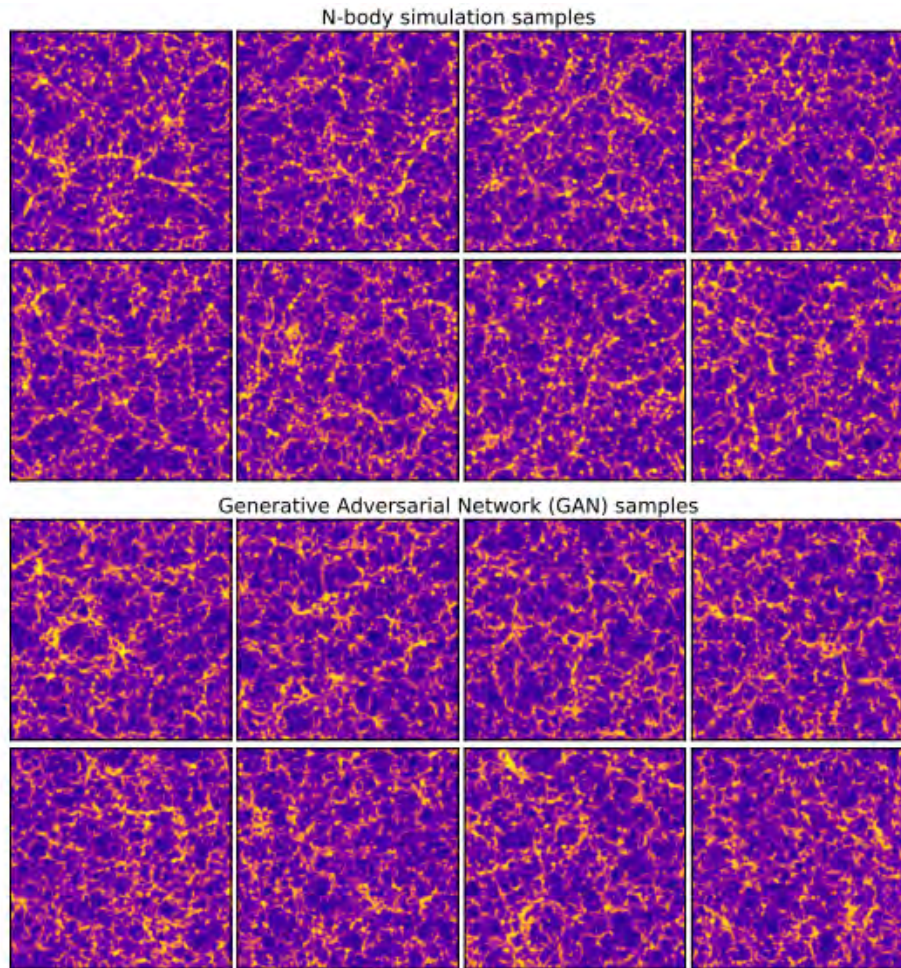
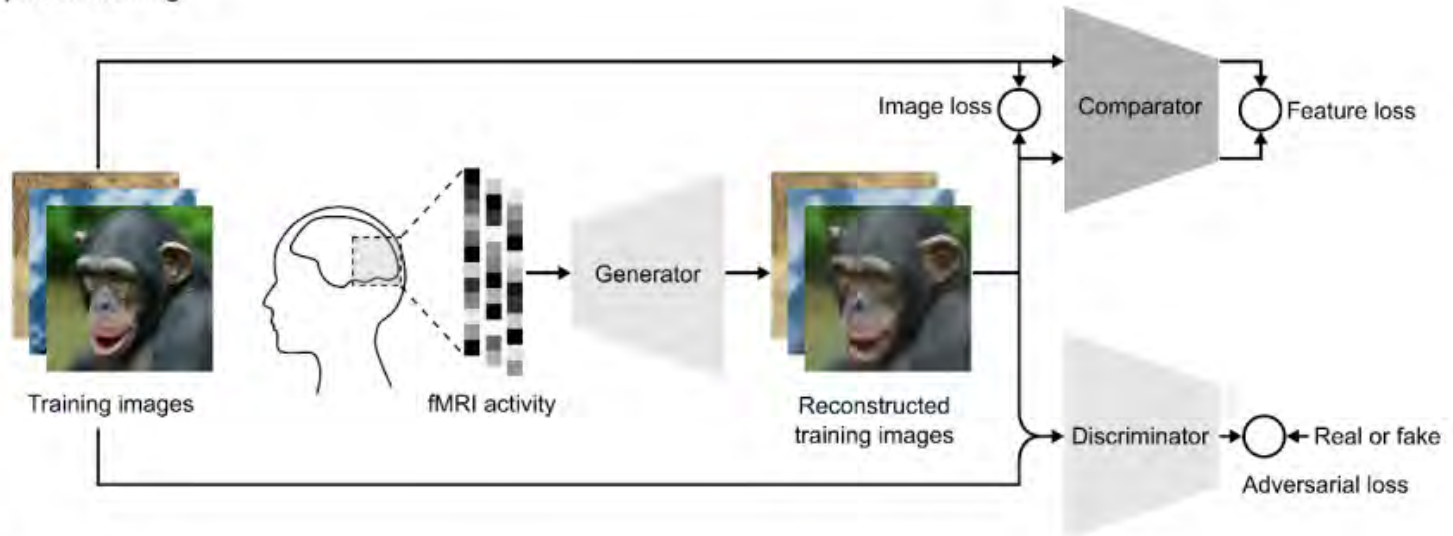


Figure 1: Samples from N-body simulation and from GAN for the box size of 500 Mpc. Note that the transformation in Equation 3.1 with $\alpha = 20$ was applied to the images shown above for better clarity.

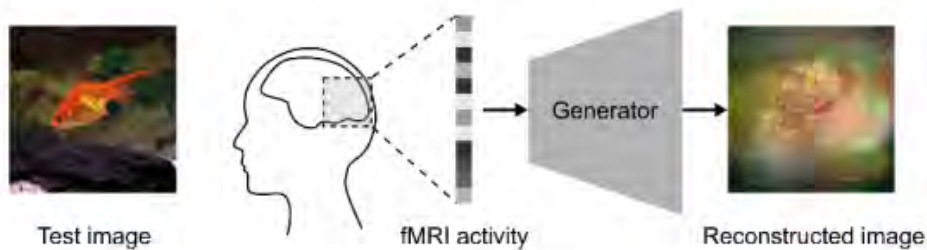
Learning cosmological models (Rodriguez et al, 2018)

Brain reading

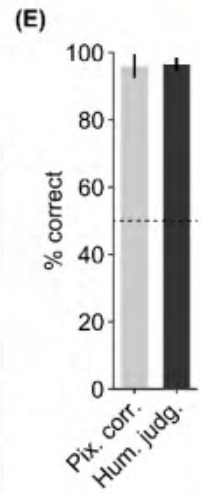
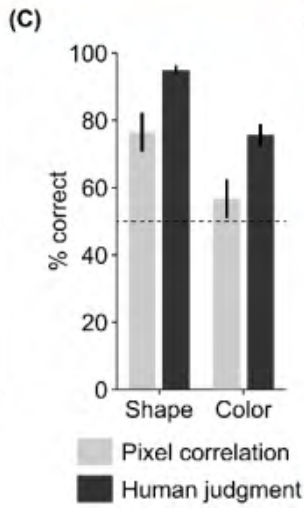
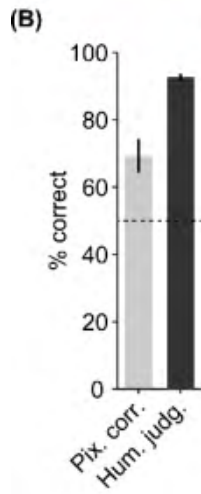
(A) Model training



(B) Model test



(Shen et al, 2018)



(Shen et al, 2018)



Deep image reconstruction: Natural images



Watch later



Share



Brain reading (Shen et al, 2018)

The end.

References

- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... & Bengio, Y. (2014). Generative adversarial nets. In Advances in neural information processing systems (pp. 2672-2680).
- Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein gan. arXiv preprint arXiv:1701.07875.
- Mescheder, L., Geiger, A., & Nowozin, S. (2018). Which training methods for GANs do actually Converge?. arXiv preprint arXiv:1801.04406.

Deep Learning

Lecture 8: Uncertainty

Prof. Gilles Louppe
g.louppe@uliege.be

Today

How to model **uncertainty** in deep learning?

- Uncertainty
- Aleatoric uncertainty
- Epistemic uncertainty

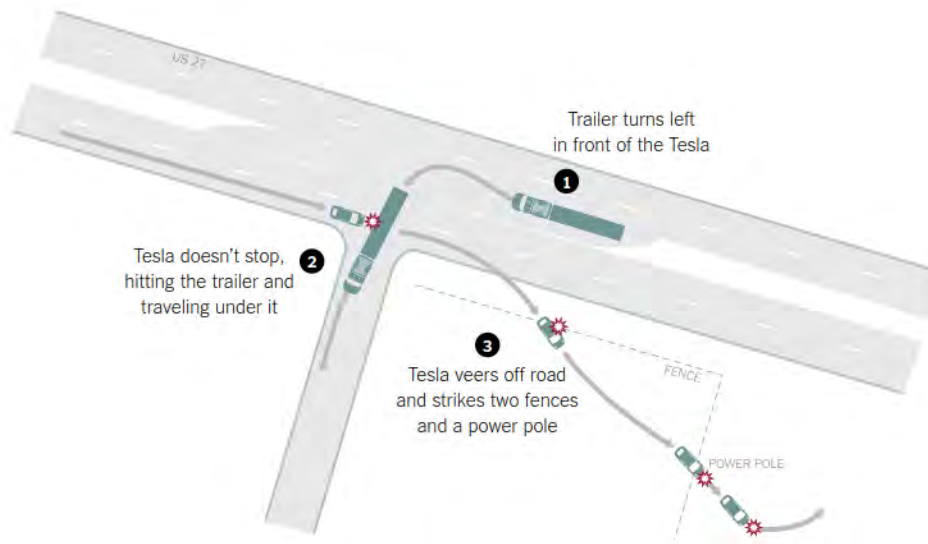


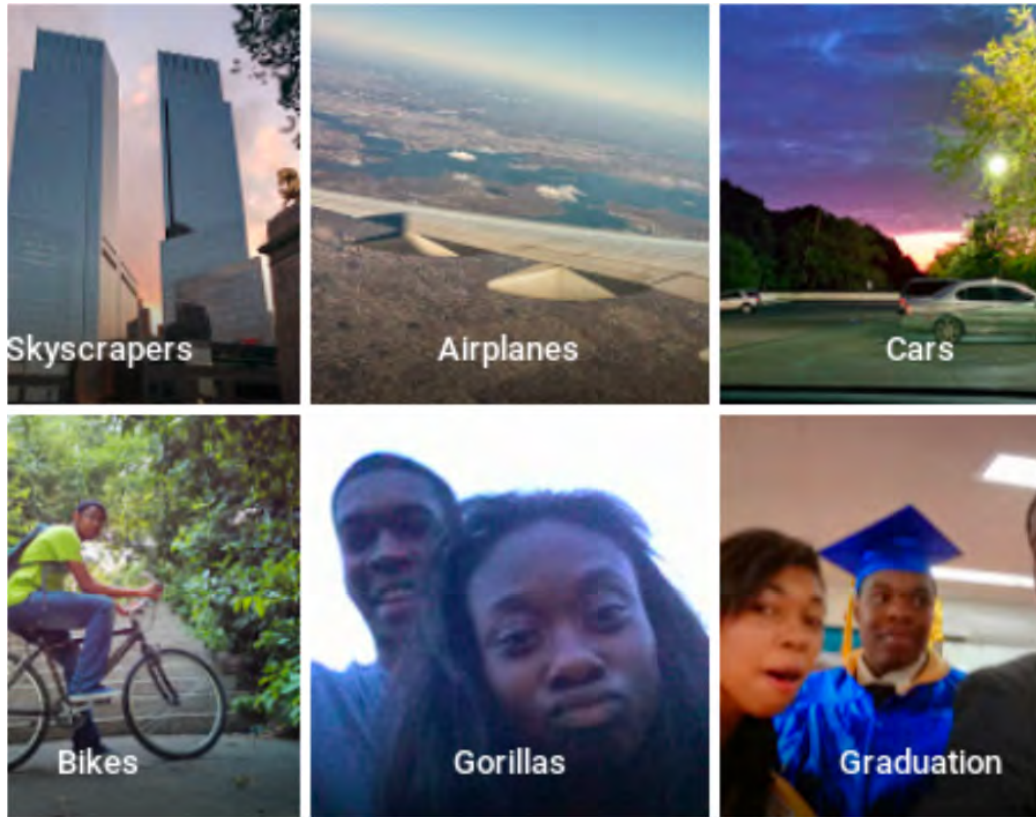
*"Every time a scientific paper presents a bit of data, it's accompanied by an **error bar** – a quiet but insistent reminder that no knowledge is complete or perfect. It's a **calibration of how much we trust what we think we know.**" — Carl Sagan.*

Uncertainty

Motivation

In May 2016, there was the **first fatality** from an assisted driving system, caused by the perception system confusing the white side of a trailer for bright sky.





An image classification system erroneously identifies two African Americans as gorillas, raising concerns of racial discrimination.

If both these algorithms were able to assign a high level of **uncertainty** to their erroneous predictions, then the system may have been able to **make better decisions**, and likely avoid disaster.

Types of uncertainty

Case 1

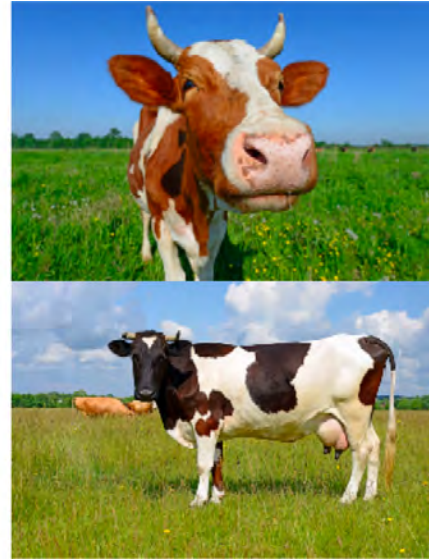
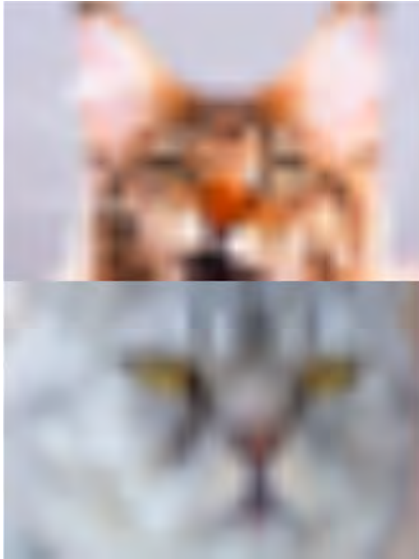
Let us consider a neural network model trained with several pictures of dog breeds.

- We ask the model to decide on a dog breed using a photo of a cat.
- What would you want the model to do?



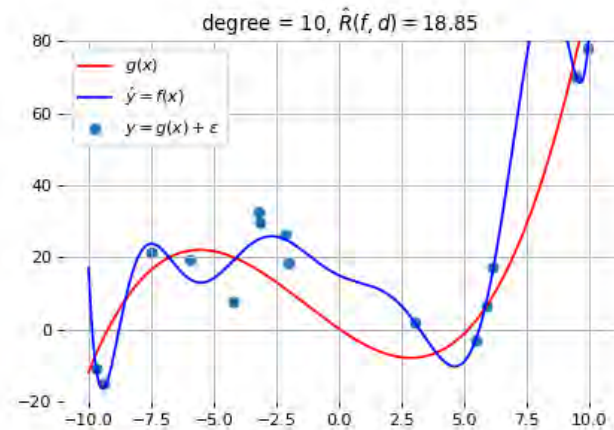
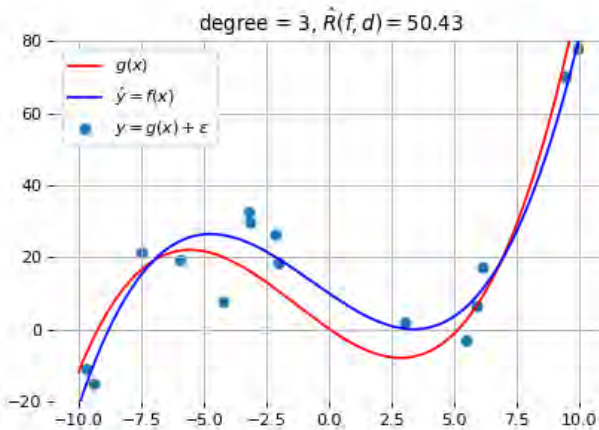
Case 2

We have three different types of images to classify, cat, dog, and cow, where only cat images are noisy.



Case 3

What is the best model parameters that best explain a given dataset? What model structure should we use?



Case 1: Given a model trained with several pictures of dog breeds. We ask the model to decide on a dog breed using a photo of a cat.

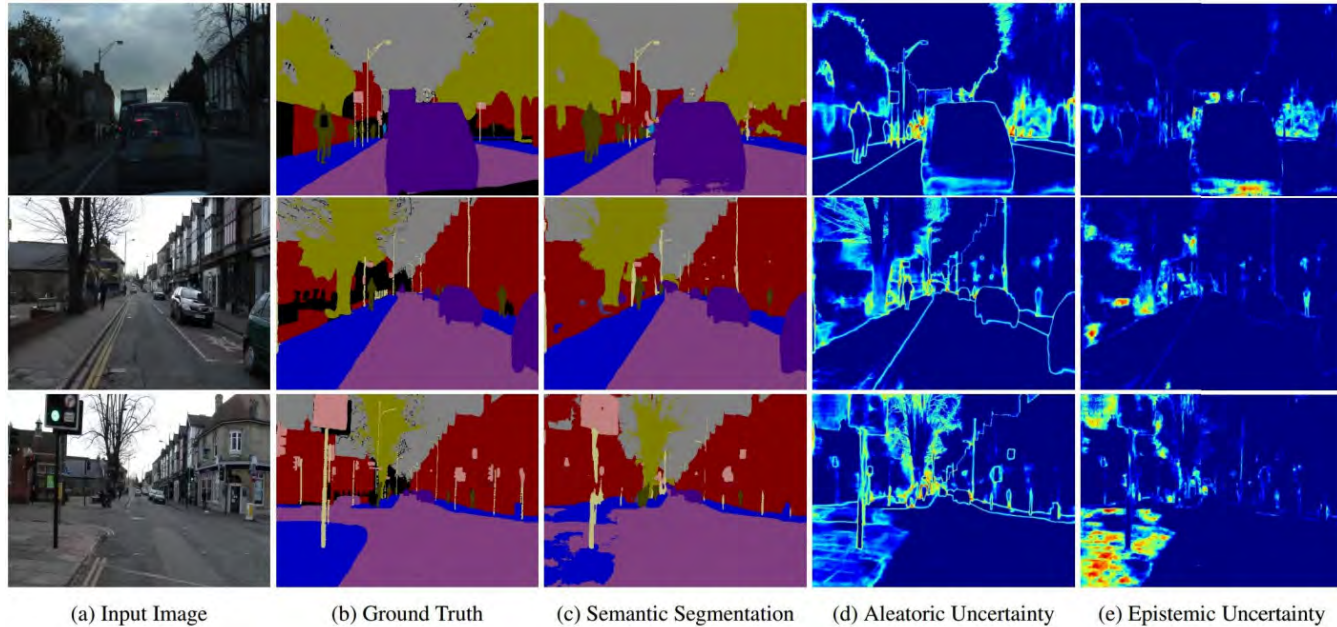
⇒ **Out of distribution test data.**

Case 2: We have three different types of images to classify, cat, dog, and cow, where only cat images are noisy.

⇒ **Aleatoric uncertainty.**

Case 3: What is the best model parameters that best explain a given dataset?
What model structure should we use?

⇒ **Epistemic uncertainty.**



*"Our model exhibits in (d) increased **aleatoric uncertainty on object boundaries and for objects far from the camera. Epistemic uncertainty accounts for our ignorance about which model generated our collected data.** In (e) our model exhibits increased epistemic uncertainty for semantically and visually challenging pixels. The bottom row shows a failure case of the segmentation model when the model fails to segment the footpath due to increased epistemic uncertainty, but not aleatoric uncertainty."*

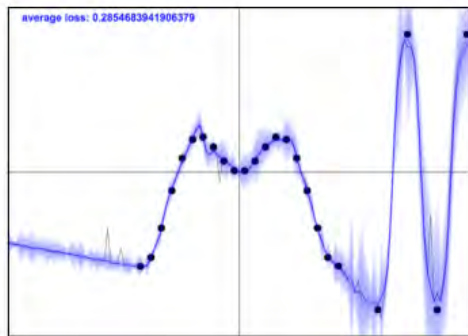
Aleatoric uncertainty

Aleatoric uncertainty captures noise inherent in the observations.

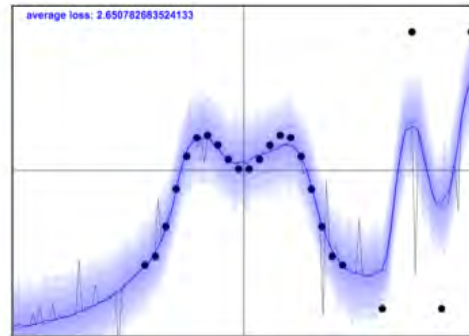
- For example, sensor noise or motion noise result in uncertainty.
- This uncertainty **cannot be reduced** with more data.
- However, aleatoric could be reduced with better measurements.

Aleatoric uncertainty can further be categorized into **homoscedastic** and **heteroscedastic** uncertainties:

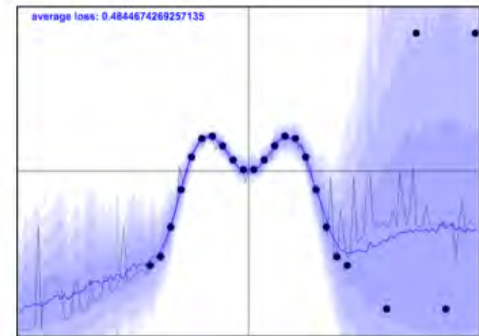
- Homoscedastic uncertainty relates to the uncertainty that a particular task might cause. It stays constant for different inputs.
- Heteroscedastic uncertainty depends on the inputs to the model, with some inputs potentially having more noisy outputs than others.



(a) Homoscedastic model with small observation noise.



(b) Homoscedastic model with large observation noise.



(c) Heteroscedastic model with data-dependent observation noise.

Regression with uncertainty

Consider training data $(\mathbf{x}, y) \sim P(X, Y)$, with

- $\mathbf{x} \in \mathbb{R}^p$,
- $y \in \mathbb{R}$.

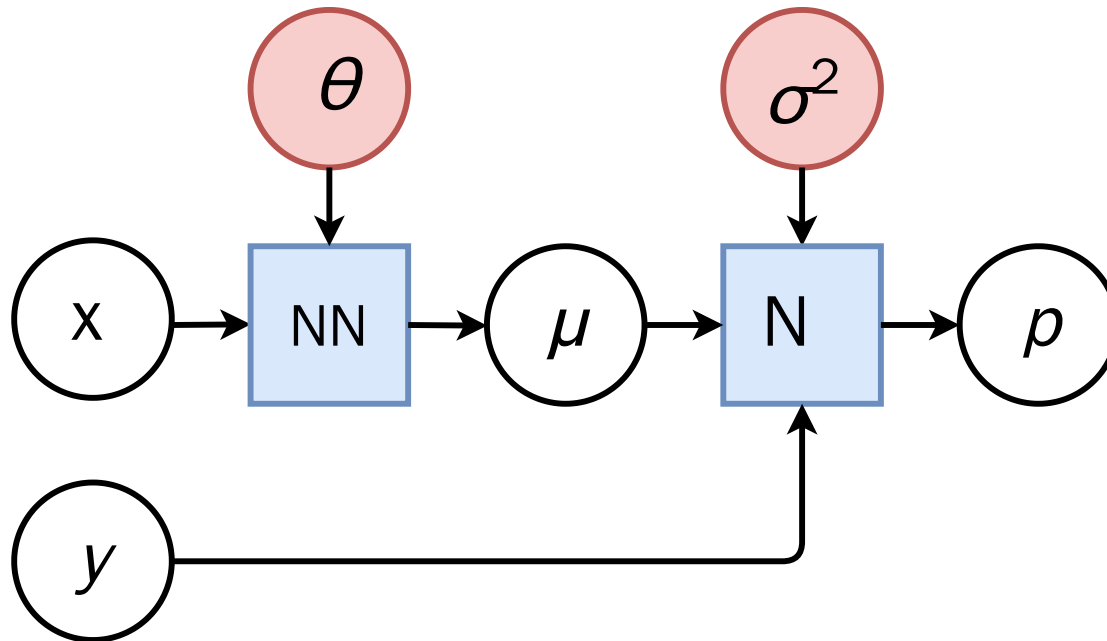
We model aleatoric uncertainty in the output by modelling the conditional distribution as a Normal distribution,

$$p(y|\mathbf{x}) = \mathcal{N}(y; \mu(\mathbf{x}), \sigma^2(\mathbf{x})),$$

where $\mu(\mathbf{x})$ and $\sigma^2(\mathbf{x})$ are parametric functions to be learned, such as neural networks.

In particular, we do not wish to learn a function $\hat{y} = f(\mathbf{x})$ that would only produce point estimates.

Homoscedastic aleatoric uncertainty

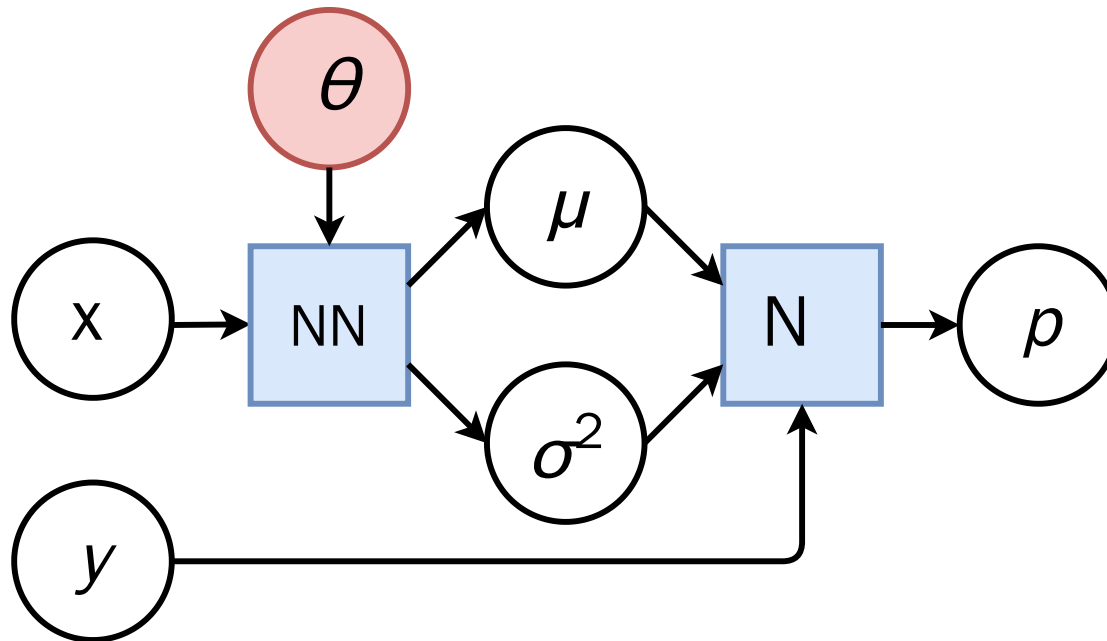


We have,

$$\begin{aligned} & \arg \max_{\theta, \sigma^2} p(\mathbf{d} | \theta, \sigma^2) \\ &= \arg \max_{\theta, \sigma^2} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} p(y_i | \mathbf{x}_i, \theta, \sigma^2) \\ &= \arg \max_{\theta, \sigma^2} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} \frac{1}{\sqrt{2\pi}\sigma} \exp\left(-\frac{(y_i - \mu(\mathbf{x}_i))^2}{2\sigma^2}\right) \\ &= \arg \min_{\theta, \sigma^2} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \frac{(y_i - \mu(\mathbf{x}_i))^2}{2\sigma^2} + \log(\sigma) + C \end{aligned}$$

[Q] What if σ^2 was fixed?

Heteroscedastic aleatoric uncertainty



Same as for the homoscedastic case, except that that σ^2 is now a function of \mathbf{x}_i :

$$\begin{aligned} & \arg \max_{\theta} p(\mathbf{d}|\theta) \\ &= \arg \max_{\theta} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} p(y_i | \mathbf{x}_i, \theta) \\ &= \arg \max_{\theta} \prod_{\mathbf{x}_i, y_i \in \mathbf{d}} \frac{1}{\sqrt{2\pi}\sigma(\mathbf{x}_i)} \exp\left(-\frac{(y_i - \mu(\mathbf{x}_i))^2}{2\sigma^2(\mathbf{x}_i)}\right) \\ &= \arg \min_{\theta} \sum_{\mathbf{x}_i, y_i \in \mathbf{d}} \frac{(y_i - \mu(\mathbf{x}_i))^2}{2\sigma^2(\mathbf{x}_i)} + \log(\sigma(\mathbf{x}_i)) + C \end{aligned}$$

- What is the role of $2\sigma^2(\mathbf{x}_i)$?
- What about $\log(\sigma(\mathbf{x}_i))$?

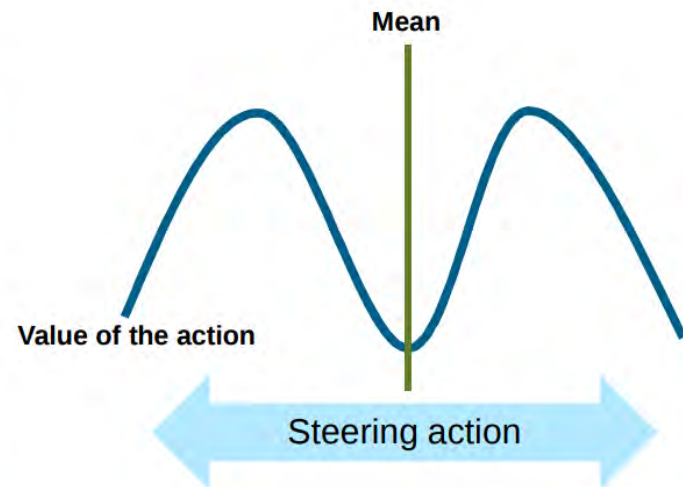
Multimodality

Modelling $p(y|\mathbf{x})$ as a unimodal Gaussian is not always a good idea!

(and it would be even worse to have only point estimates for y !)



https://commons.wikimedia.org/wiki/File:Newport_Whitepit_Lane_pot_hole.JPG

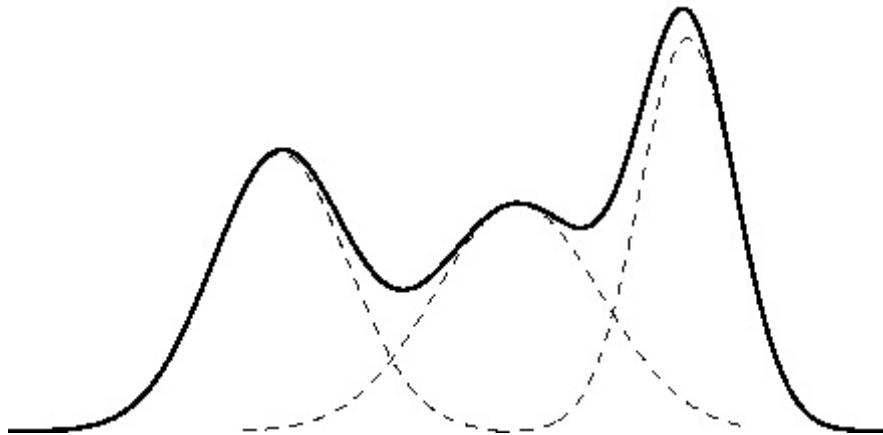


Gaussian mixture model

A **Gaussian mixture model** (GMM) defines instead $p(y|\mathbf{x})$ as a mixture of K Gaussian components,

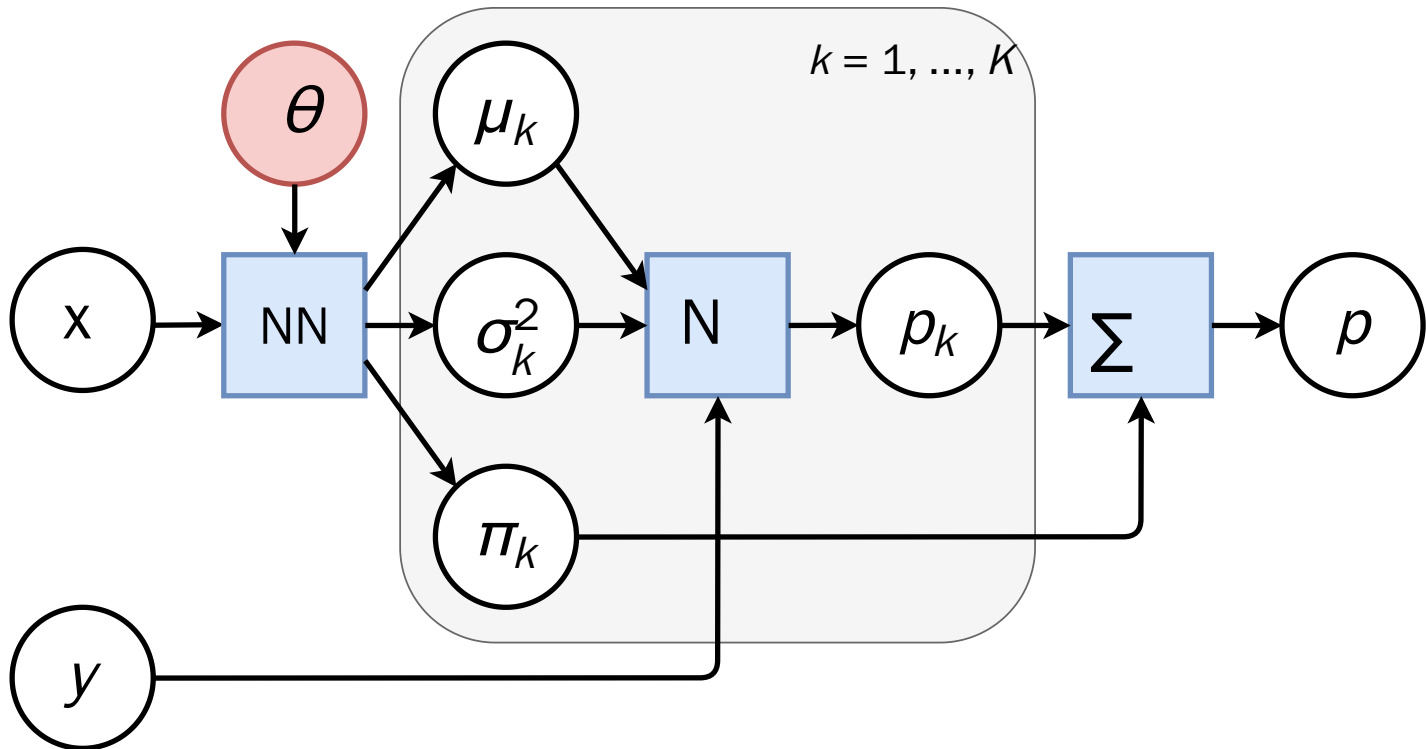
$$p(y|\mathbf{x}) = \sum_{k=1}^K \pi_k \mathcal{N}(y; \mu_k, \sigma_k^2),$$

where $0 \leq \pi_k \leq 1$ for all k and $\sum_{k=1}^K \pi_k = 1$.



Mixture density network

A **mixture density network** is a neural network implementation of the Gaussian mixture model.

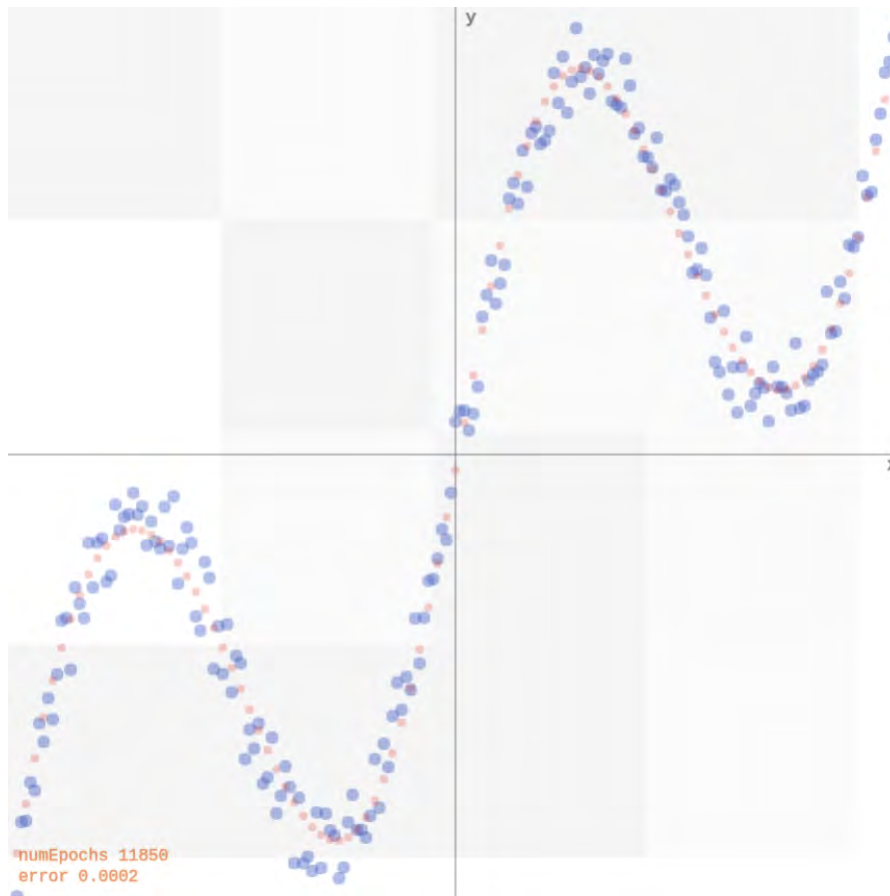


Illustration

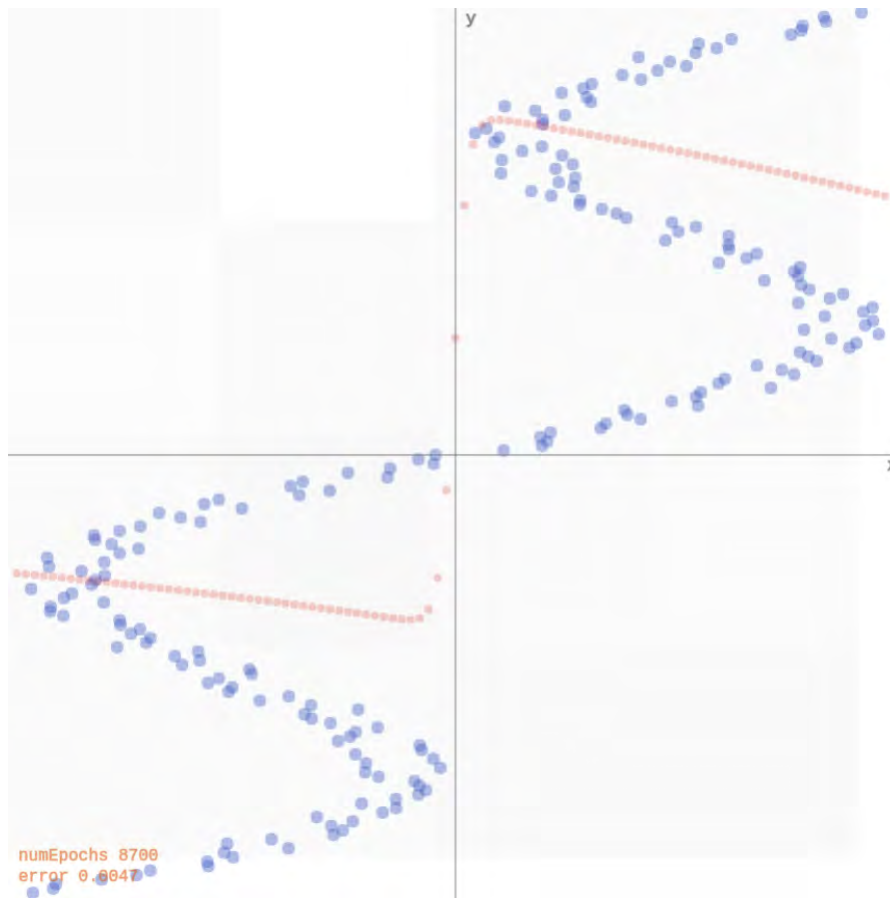
Let us consider training data generated randomly as

$$y_i = \mathbf{x}_i + 0.3 \sin(4\pi\mathbf{x}_i) + \epsilon_i$$

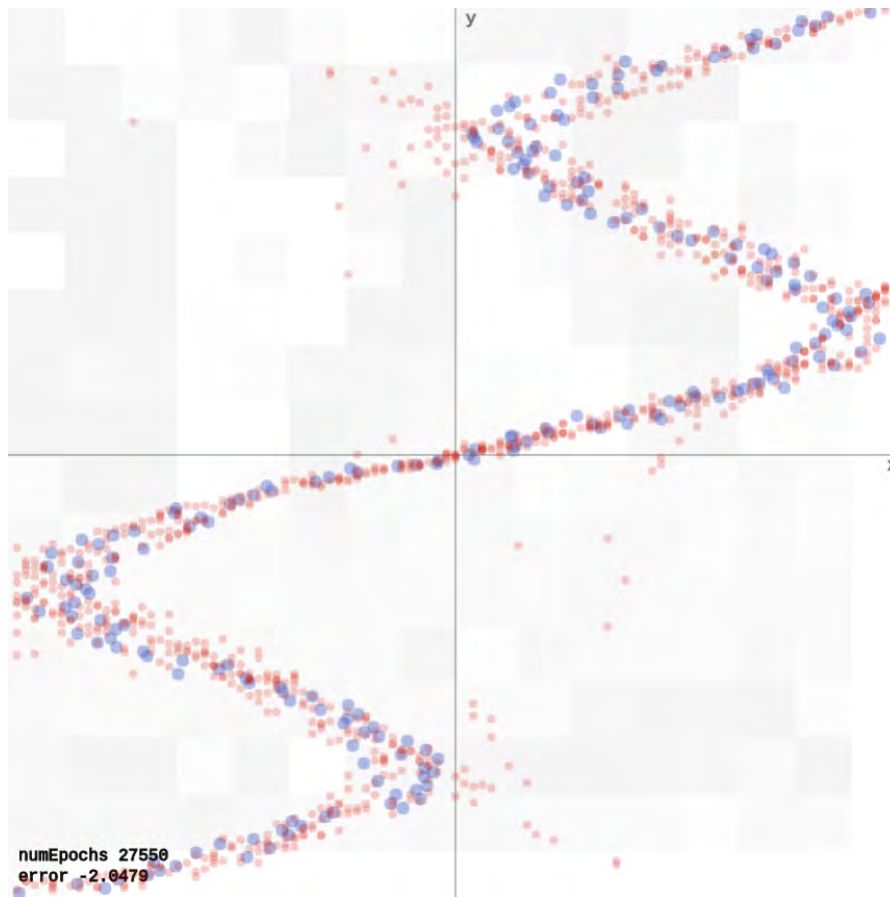
with $\epsilon_i \sim \mathcal{N}$.



The data can be fit with a 2-layer network producing point estimates for y .
[\[demo\]](#)



If we flip \mathbf{x}_i and y_i , the network faces issues since for each input, there are multiple outputs that can work. It produces some sort of average of the correct values. [[demo](#)]



A mixture density network models the data correctly, as it predicts for each input a distribution for the output, rather than a point estimate. [[demo](#)]

Epistemic uncertainty

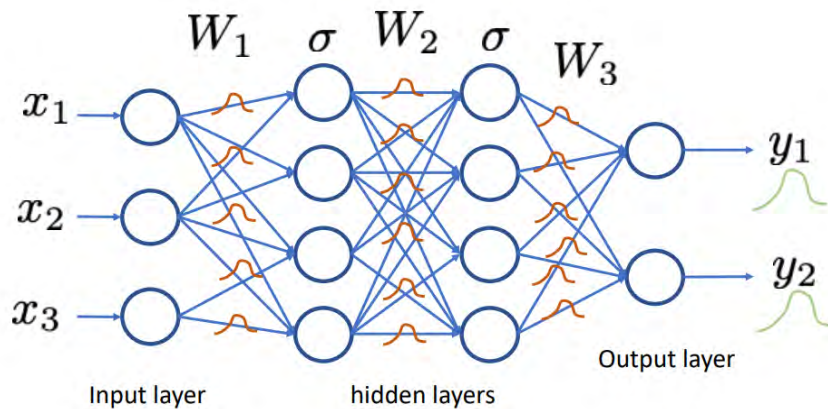
Epistemic uncertainty accounts for uncertainty in the model parameters.

- It captures our **ignorance** about which model generated the collected data.
- It can be explained away given enough data (why?).
- It is also often referred to as **model uncertainty**.

Bayesian neural networks

To capture epistemic uncertainty in a neural network, we model our ignorance with a prior distribution $p(\omega)$ over its weights.

Then we invoke Bayes for making predictions.



- The prior predictive distribution at \mathbf{x} is given by integrating over all possible weight configurations,

$$p(y|\mathbf{x}) = \int p(y|\mathbf{x}, \omega)p(\omega)d\omega.$$

- Given training data $\mathbf{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ and $\mathbf{Y} = \{y_1, \dots, y_N\}$, a Bayesian update results in the posterior

$$p(\omega|\mathbf{X}, \mathbf{Y}) = \frac{p(\mathbf{Y}|\mathbf{X}, \omega)p(\omega)}{p(\mathbf{Y}|\mathbf{X})}.$$

- The posterior predictive distribution is then given by

$$p(y|\mathbf{x}, \mathbf{X}, \mathbf{Y}) = \int p(y|\mathbf{x}, \omega)p(\omega|\mathbf{X}, \mathbf{Y})d\omega.$$

Bayesian neural networks are **easy to formulate**, but notoriously **difficult to perform inference in**.

- This stems mainly from the fact that the marginal $p(\mathbf{Y}|\mathbf{X})$ is intractable to evaluate, which results in the posterior $p(\omega|\mathbf{X}, \mathbf{Y})$ not being tractable either.
- Therefore, we must rely on approximations.

Variational inference

Variational inference can be used for building an approximation $q(\omega; \nu)$ of the posterior $p(\omega | \mathbf{X}, \mathbf{Y})$.

As before (see Lecture 6), we can show that minimizing

$$\text{KL}(q(\omega; \nu) || p(\omega | \mathbf{X}, \mathbf{Y}))$$

with respect to the variational parameters ν , is identical to maximizing the evidence lower bound objective (ELBO)

$$\text{ELBO}(\nu) = \mathbb{E}_{q(\omega; \nu)} [\log p(\mathbf{Y} | \mathbf{X}, \omega)] - \text{KL}(q(\omega; \nu) || p(\omega)).$$

The integral in the ELBO is not tractable for almost all q , but it can be minimized with stochastic gradient descent:

1. Sample $\hat{\omega} \sim q(\omega; \nu)$.
2. Do one step of maximization with respect to ν on

$$\hat{L}(\nu) = \log p(\mathbf{Y}|\mathbf{X}, \hat{\omega}) - \text{KL}(q(\omega; \nu) || p(\omega))$$

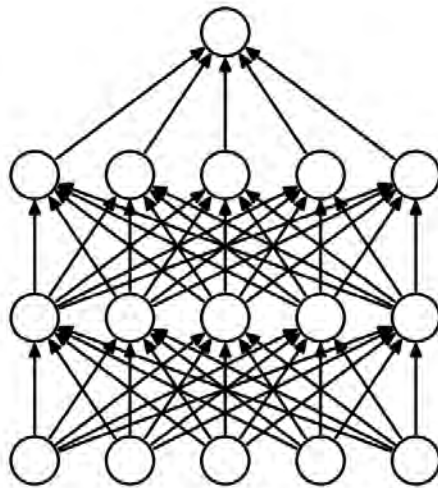
In the context of Bayesian neural networks, this procedure is also known as **Bayes by backprop** (Blundell et al, 2015).

Dropout

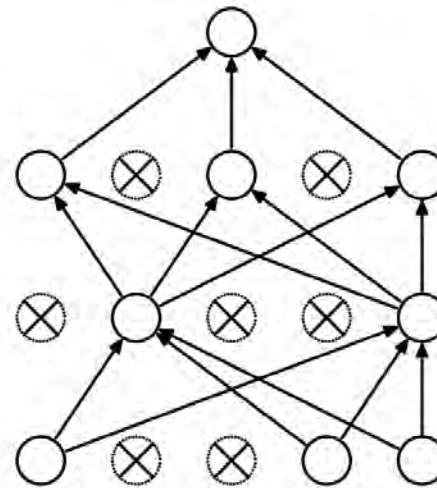
Dropout is an **empirical** technique that was first proposed to avoid overfitting in neural networks.

At **each training step** (i.e., for each sample within a mini-batch):

- Remove each node in the network with a probability p .
- Update the weights of the remaining nodes with backpropagation.



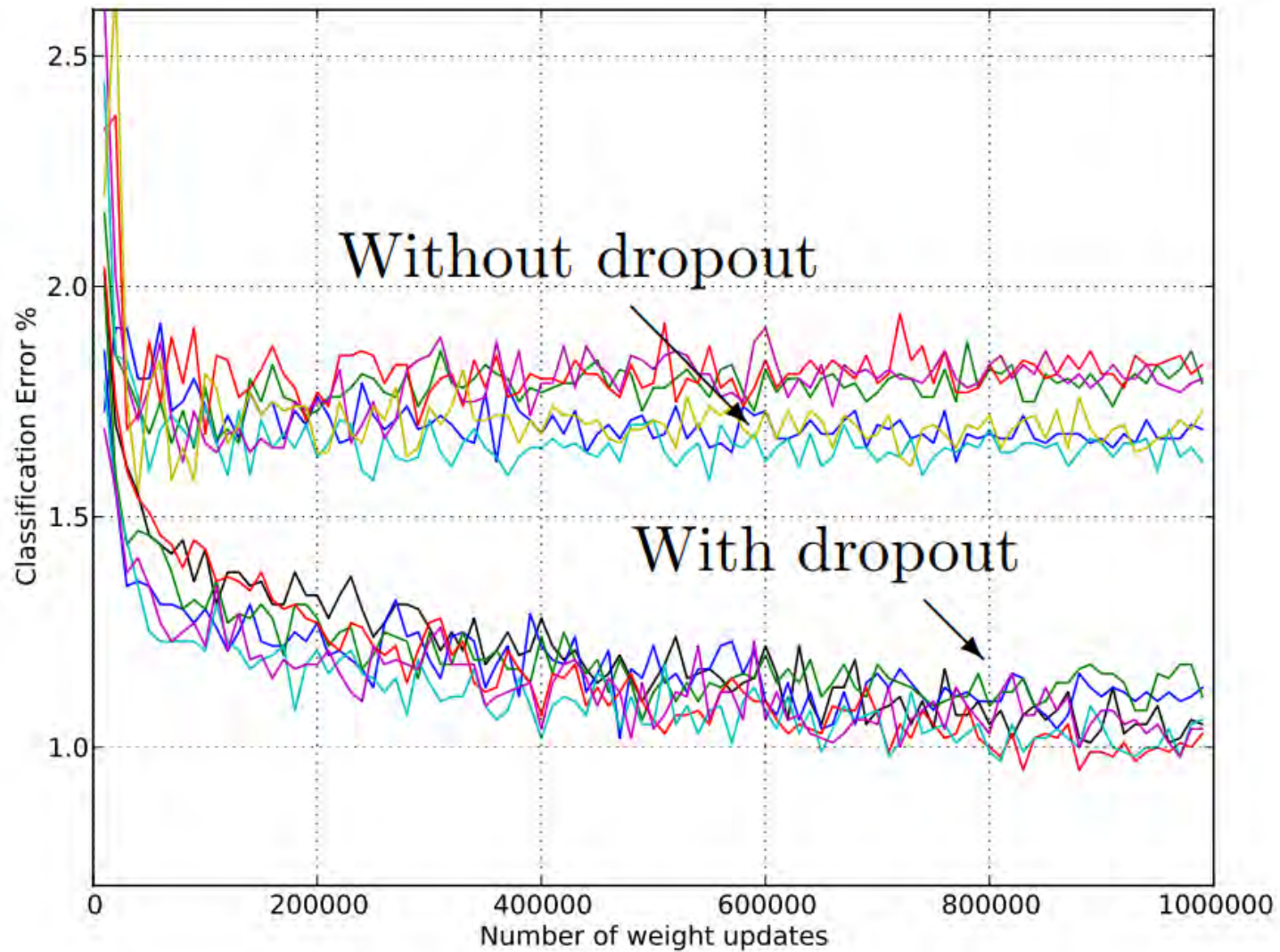
(a) Standard Neural Net



(b) After applying dropout.

At **test time**, either:

- Make predictions using the trained network **without** dropout but rescaling the weights by the dropout probability p (fast and standard).
- Sample T neural networks using dropout and average their predictions (slower but better principled).



Why does dropout work?

- It makes the learned weights of a node less sensitive to the weights of the other nodes.
- This forces the network to learn several independent representations of the patterns and thus decreases overfitting.
- It approximates **Bayesian model averaging**.

Dropout does variational inference

What variational family q would correspond to dropout?

- Let us split the weights ω per layer, $\omega = \{\mathbf{W}_1, \dots, \mathbf{W}_L\}$, where \mathbf{W}_i is further split per unit $\mathbf{W}_i = \{\mathbf{w}_{i,1}, \dots, \mathbf{w}_{i,q_i}\}$.
- Variational parameters ν are split similarly into $\nu = \{\mathbf{M}_1, \dots, \mathbf{M}_L\}$, with $\mathbf{M}_i = \{\mathbf{m}_{i,1}, \dots, \mathbf{m}_{i,q_i}\}$.
- Then, the proposed $q(\omega; \nu)$ is defined as follows:

$$q(\omega; \nu) = \prod_{i=1}^L q(\mathbf{W}_i; \mathbf{M}_i)$$
$$q(\mathbf{W}_i; \mathbf{M}_i) = \prod_{k=1}^{q_i} q(\mathbf{w}_{i,k}; \mathbf{m}_{i,k})$$
$$q(\mathbf{w}_{i,k}; \mathbf{m}_{i,k}) = p\delta_0(\mathbf{w}_{i,k}) + (1-p)\delta_{\mathbf{m}_{i,k}}(\mathbf{w}_{i,k})$$

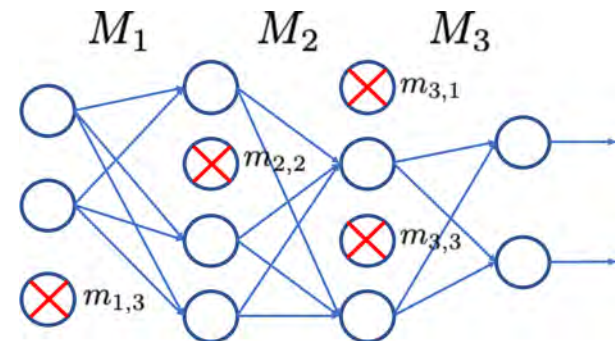
where $\delta_a(x)$ denotes a (multivariate) Dirac distribution centered at a .

Given the previous definition for q , sampling parameters $\hat{\omega} = \{\hat{\mathbf{W}}_1, \dots, \hat{\mathbf{W}}_L\}$ is done as follows:

- Draw binary $z_{i,k} \sim \text{Bernoulli}(1 - p)$ for each layer i and unit k .
- Compute $\hat{\mathbf{W}}_i = \mathbf{M}_i \cdot \text{diag}([z_{i,k}]_{k=1}^{q_i})$, where \mathbf{M}_i denotes a matrix composed of the columns $\mathbf{m}_{i,k}$.

That is, $\hat{\mathbf{W}}_i$ are obtained by setting columns of \mathbf{M}_i to zero with probability p .

This is **strictly equivalent to dropout**, i.e. removing units from the network with probability p .



Therefore, one step of stochastic gradient descent on the ELBO becomes:

1. Sample $\hat{\omega} \sim q(\omega; \nu) \Leftrightarrow$ Randomly set units of the network to zero \Leftrightarrow Dropout.
2. Do one step of maximization with respect to $\nu = \{\mathbf{M}_i\}$ on

$$\hat{L}(\nu) = \log p(\mathbf{Y}|\mathbf{X}, \hat{\omega}) - \text{KL}(q(\omega; \nu) || p(\omega)).$$

Maximizing $\hat{L}(\nu)$ is equivalent to minimizing

$$-\hat{L}(\nu) = -\log p(\mathbf{Y}|\mathbf{X}, \hat{\omega}) + \text{KL}(q(\omega; \nu)||p(\omega))$$

Is this equivalent to one minimization step of a standard classification or regression objective? **Yes!**

- The first term is the typical objective (see Lecture 2).
- The second term forces q to remain close to the prior $p(\omega)$.
 - If $p(\omega)$ is Gaussian, minimizing the KL is equivalent to ℓ_2 regularization.
 - If $p(\omega)$ is Laplacian, minimizing the KL is equivalent to ℓ_1 regularization.

Conversely, this shows that when **training a network with dropout** with a standard classification or regression objective, one **is actually implicitly doing variational inference** to match the posterior distribution of the weights.

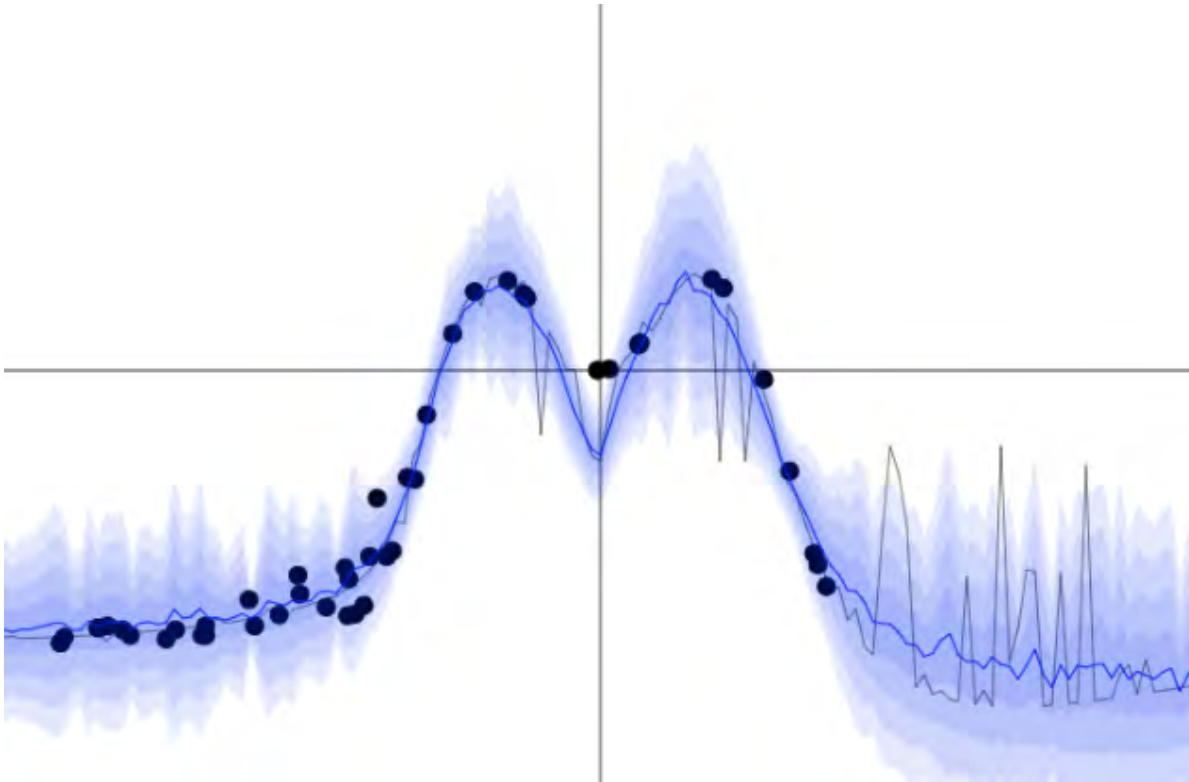
Uncertainty estimates from dropout

Proper epistemic uncertainty estimates at \mathbf{x} can be obtained in a principled way using Monte-Carlo integration:

- Draw T sets of network parameters $\hat{\omega}_t$ from $q(\omega; \nu)$.
- Compute the predictions for the T networks, $\{f(\mathbf{x}; \hat{\omega}_t)\}_{t=1}^T$.
- Approximate the predictive mean and variance as follows:

$$\mathbb{E}_{p(y|\mathbf{x}, \mathbf{X}, \mathbf{Y})} [y] \approx \frac{1}{T} \sum_{t=1}^T f(\mathbf{x}; \hat{\omega}_t)$$

$$\mathbb{V}_{p(y|\mathbf{x}, \mathbf{X}, \mathbf{Y})} [y] \approx \sigma^2 + \frac{1}{T} \sum_{t=1}^T f(\mathbf{x}; \hat{\omega}_t)^2 - \hat{\mathbb{E}} [y]^2$$



Yarin Gal's [demo](#).

Pixel-wise depth regression

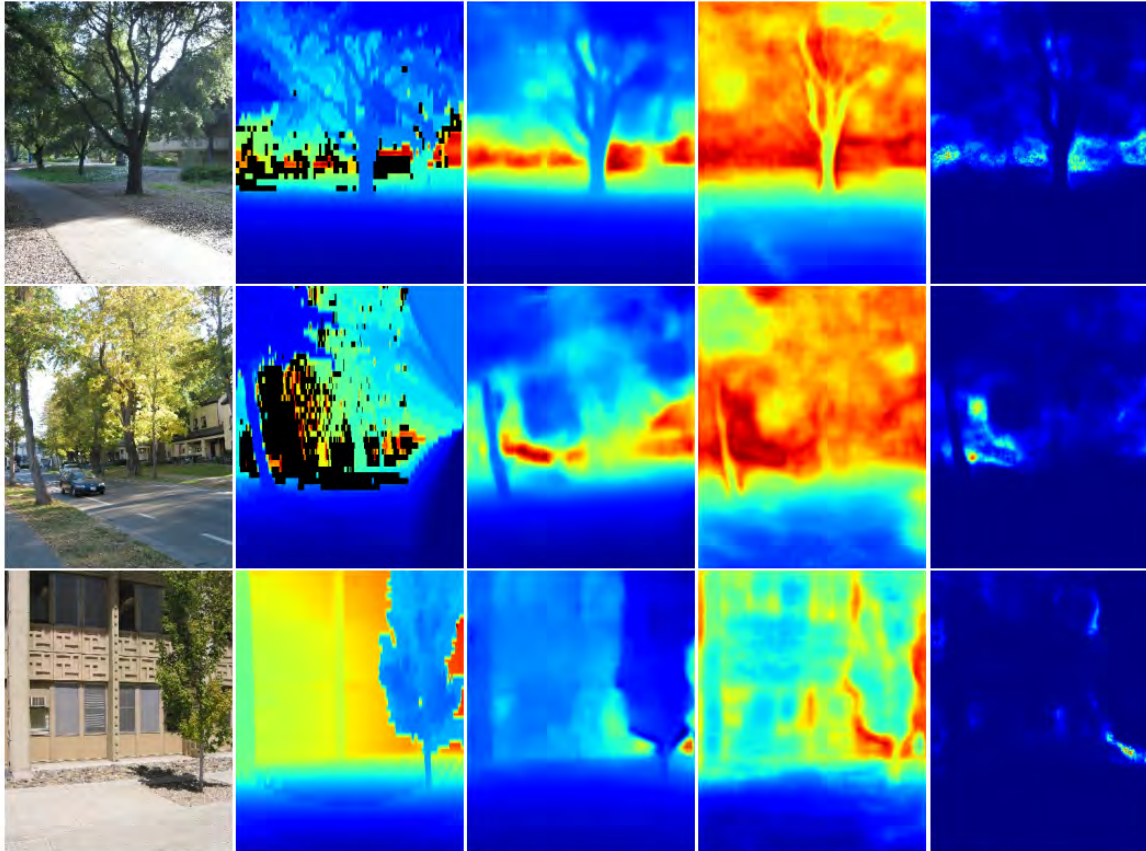


Figure 6: Qualitative results on the Make3D depth regression dataset. Left to right: input image, ground truth, depth prediction, aleatoric uncertainty, epistemic uncertainty. Make3D does not provide labels for depth greater than 70m, therefore these distances dominate the epistemic uncertainty signal. Aleatoric uncertainty is prevalent around depth edges or distant points.

Bayesian Infinite Networks

Consider the 1-layer MLP with a hidden layer of size q and a bounded activation function σ :

$$f(x) = b + \sum_{j=1}^q v_j h_j(x)$$
$$h_j(x) = \sigma \left(a_j + \sum_{i=1}^p u_{i,j} x_i \right)$$

Assume Gaussian priors $v_j \sim \mathcal{N}(0, \sigma_v^2)$, $b \sim \mathcal{N}(0, \sigma_b^2)$, $u_{i,j} \sim \mathcal{N}(0, \sigma_u^2)$ and $a_j \sim \mathcal{N}(0, \sigma_a^2)$.

For a fixed value $\mathbf{x}^{(1)}$, let us consider the prior distribution of $f(\mathbf{x}^{(1)})$ implied by the prior distributions for the weights and biases.

We have

$$\mathbb{E}[v_j h_j(\mathbf{x}^{(1)})] = \mathbb{E}[v_j] \mathbb{E}[h_j(\mathbf{x}^{(1)})] = 0,$$

since v_j and $h_j(\mathbf{x}^{(1)})$ are statistically independent and v_j has zero mean by hypothesis.

The variance of the contribution of each hidden unit h_j is

$$\begin{aligned} \mathbb{V}[v_j h_j(\mathbf{x}^{(1)})] &= \mathbb{E}[(v_j h_j(\mathbf{x}^{(1)}))^2] - \mathbb{E}[v_j h_j(\mathbf{x}^{(1)})]^2 \\ &= \mathbb{E}[v_j^2] \mathbb{E}[h_j(\mathbf{x}^{(1)})^2] \\ &= \sigma_v^2 \mathbb{E}[h_j(\mathbf{x}^{(1)})^2], \end{aligned}$$

which must be finite since h_j is bounded by its activation function.

We define $V(\mathbf{x}^{(1)}) = \mathbb{E}[h_j(\mathbf{x}^{(1)})^2]$, and is the same for all j .

What if $q \rightarrow \infty$?

By the Central Limit Theorem, as $q \rightarrow \infty$, the total contribution of the hidden units, $\sum_{j=1}^q v_j h_j(x)$, to the value of $f(x^{(1)})$ becomes a Gaussian with variance $q\sigma_v^2 V(x^{(1)})$.

The bias b is also Gaussian, of variance σ_b^2 , so for large q , the prior distribution $f(x^{(1)})$ is a Gaussian of variance $\sigma_b^2 + q\sigma_v^2 V(x^{(1)})$.

Accordingly, for $\sigma_v = \omega_v q^{-\frac{1}{2}}$, for some fixed ω_v , the prior $f(x^{(1)})$ converges to a Gaussian of mean zero and variance $\sigma_b^2 + \omega_v^2 \sigma_v^2 V(x^{(1)})$ as $q \rightarrow \infty$.

For two or more fixed values $x^{(1)}, x^{(2)}, \dots$, a similar argument shows that, as $q \rightarrow \infty$, the joint distribution of the outputs converges to a multivariate Gaussian with means of zero and covariances of

$$\begin{aligned}\mathbb{E}[f(x^{(1)})f(x^{(2)})] &= \sigma_b^2 + \sum_{j=1}^q \sigma_v^2 \mathbb{E}[h_j(x^{(1)})h_j(x^{(2)})] \\ &= \sigma_b^2 + \omega_v^2 C(x^{(1)}, x^{(2)})\end{aligned}$$

where $C(x^{(1)}, x^{(2)}) = \mathbb{E}[h_j(x^{(1)})h_j(x^{(2)})]$ and is the same for all j .

This result states that for any set of fixed points $x^{(1)}, x^{(2)}, \dots$, the joint distribution of $f(x^{(1)}), f(x^{(2)}), \dots$ is a multivariate Gaussian.

In other words, the **infinitely wide 1-layer MLP** converges towards a **Gaussian process**.

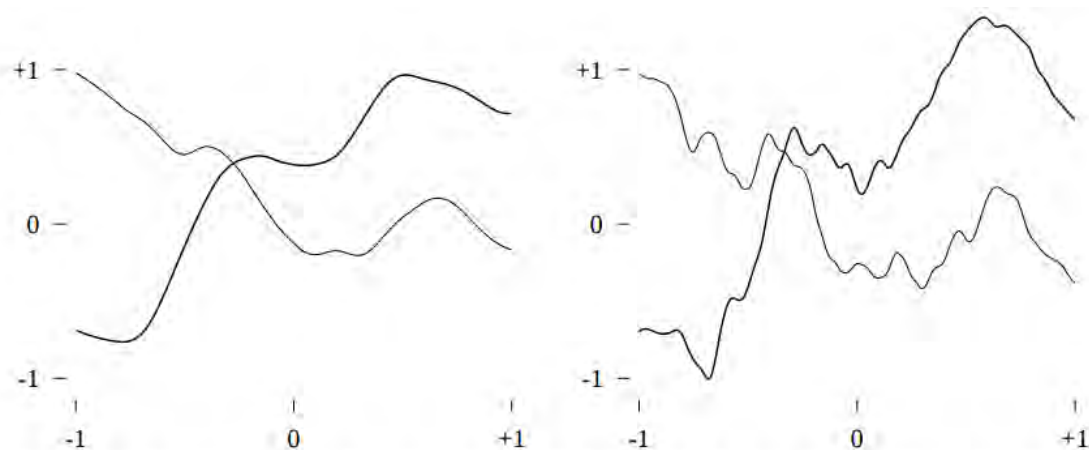


Figure 2.2: Functions drawn from Gaussian priors for a network with 10 000 tanh hidden units. Two functions drawn from a prior with $\sigma_u = 5$ are shown on the left, two from a prior with $\sigma_u = 20$ on the right. In both cases, $\sigma_a/\sigma_u = 1$ and $\sigma_b = \omega_v = 1$. The functions with different σ_u were generated using the same random number seed, the same as that used to generate the functions in the lower-right of Figure 2.1. This allows a direct evaluation of the effect of changing σ_u . (Use of a step function is equivalent to letting σ_u go to infinity, while keeping σ_a/σ_u fixed.)

(Neal, 1995)

The end.

References

- Bishop, C. M. (1994). Mixture density networks (p. 7). Technical Report NCRG/4288, Aston University, Birmingham, UK.
- Kendall, A., & Gal, Y. (2017). What uncertainties do we need in bayesian deep learning for computer vision?. In Advances in neural information processing systems (pp. 5574-5584).
- Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. The Journal of Machine Learning Research, 15(1), 1929-1958.
- Pierre Geurts, [INFO8004 Advanced Machine Learning - Lecture 1](#), 2019.

Deep Learning

Lecture 9: Adversarial attacks and defense

Prof. Gilles Louppe
g.louppe@uliege.be

Today

Can you fool neural networks?

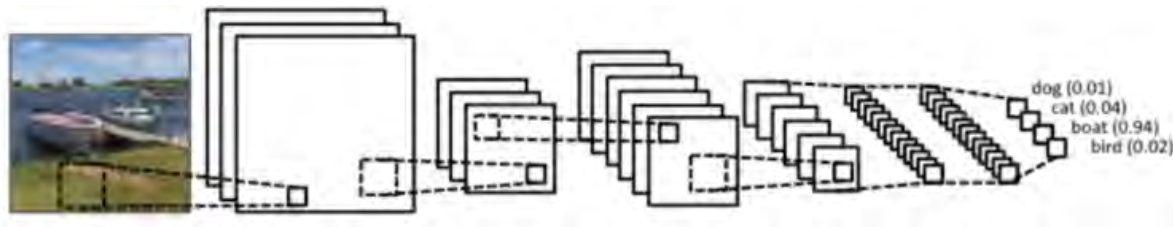
- Adversarial attacks
- Adversarial defenses

We have seen that deep networks achieve **super-human performance** on a large variety of tasks.

Soon enough, it seems like:

- neural networks will replace your doctor;
- neural networks will drive your car;
- neural networks will compose the music you listen to.

But is that the end of the story?



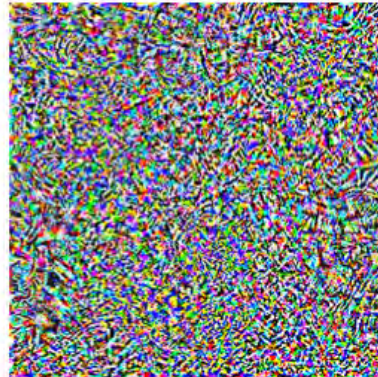
Adversarial attacks

Adversarial examples

“pig”



+ 0.005 x



=

“airliner”

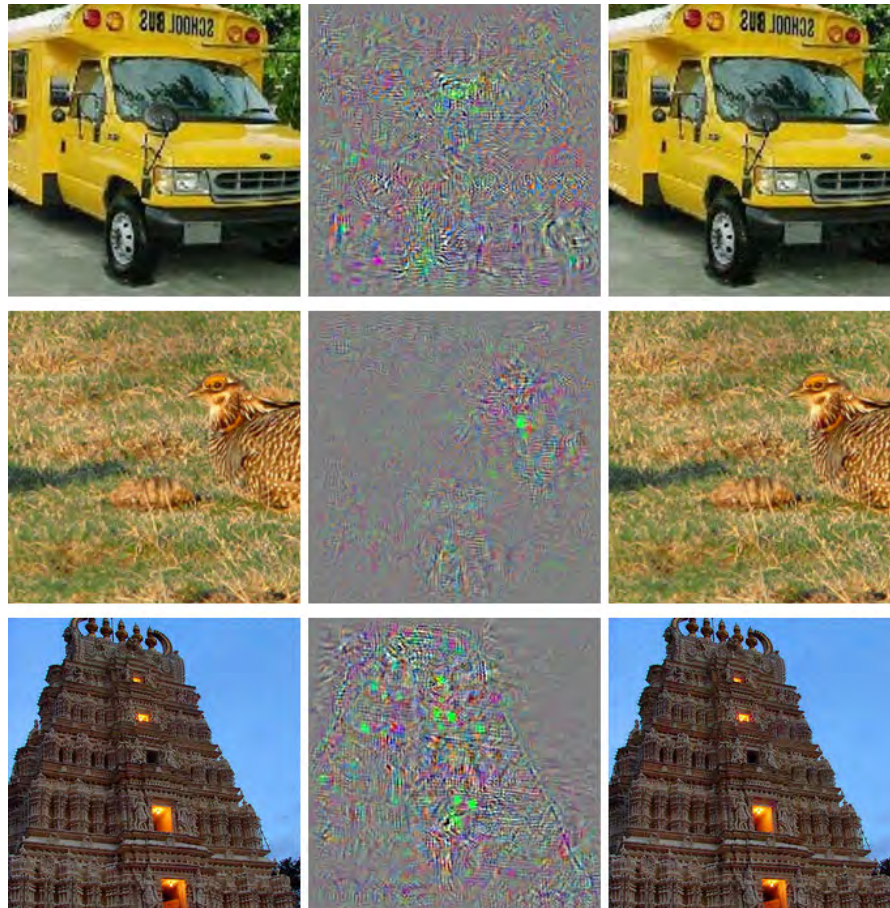


Intriguing properties of neural networks

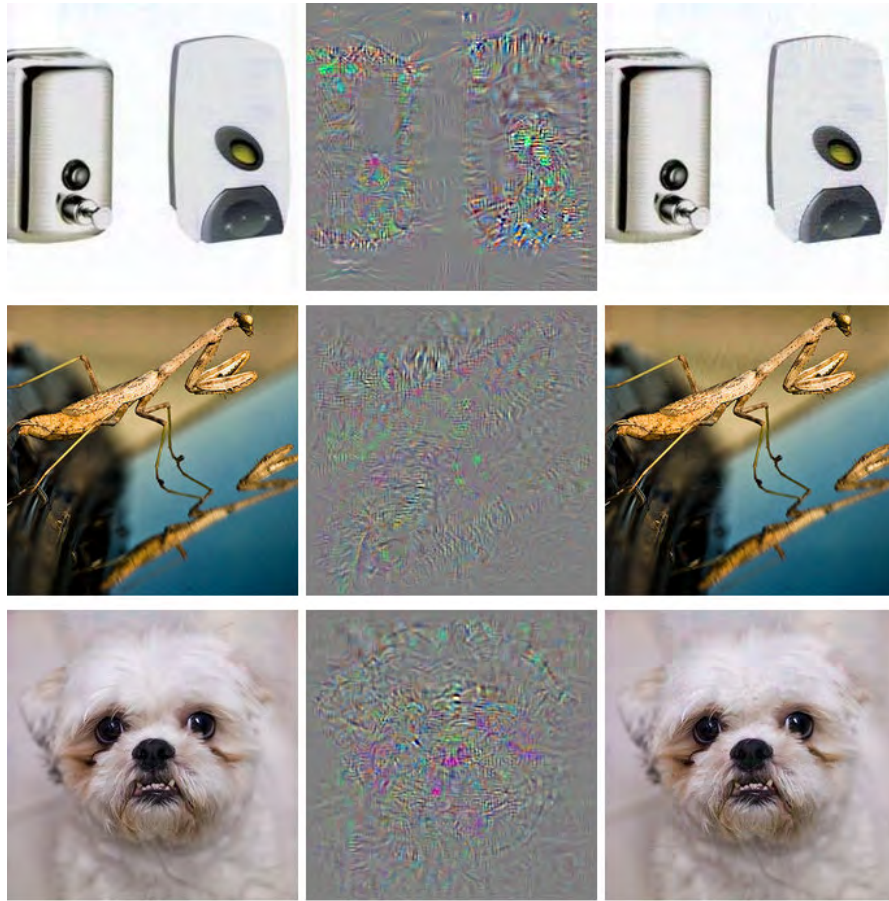
"We can cause the network to misclassify an image by applying a certain hardly perceptible perturbation, which is found by maximizing the network's prediction error. In addition, the specific nature of these perturbations is not a random artifact of learning: the same perturbation can cause a different network, that was trained on a different subset of the dataset, to misclassify the same input."

The existence of the adversarial negatives appears to be in contradiction with the network's ability to achieve high generalization performance. Indeed, if the network can generalize well, how can it be confused by these adversarial negatives, which are indistinguishable from the regular examples?"

(Szegedy et al, 2013)

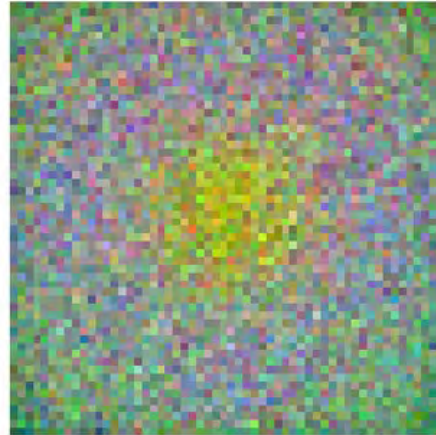


(Left) Original images. (Middle) Adversarial noise. (Right) Modified images.
All are classified as 'Ostrich'.



Fooling a logistic regression model

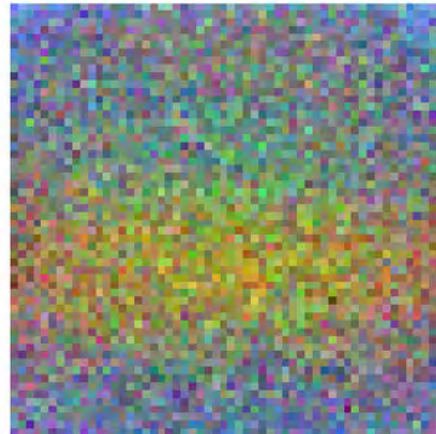
8.3% goldfish



12.5% daisy



1.0% kit fox



3.9% school bus



Many machine learning models are subject to adversarial examples, including:

- Neural networks
- Linear models
 - Logistic regression
 - Softmax regression
 - Support vector machines
- Decision trees
- Nearest neighbors

Fooling language understanding models

Article: Super Bowl 50

Paragraph: *“Peyton Manning became the first quarterback ever to lead two different teams to multiple Super Bowls. He is also the oldest quarterback ever to play in a Super Bowl at age 39. The past record was held by John Elway, who led the Broncos to victory in Super Bowl XXXIII at age 38 and is currently Denver’s Executive Vice President of Football Operations and General Manager. Quarterback Jeff Dean had jersey number 37 in Champ Bowl XXXIV.”*

Question: *“What is the name of the quarterback who was 38 in Super Bowl XXXIII?”*

Original Prediction: John Elway

Prediction under adversary: Jeff Dean

Figure 1: An example from the SQuAD dataset. The BiDAF Ensemble model originally gets the answer correct, but is fooled by the addition of an adversarial distracting sentence (in blue).

(Jia and Liang, 2017)

Fooling deep structured prediction models

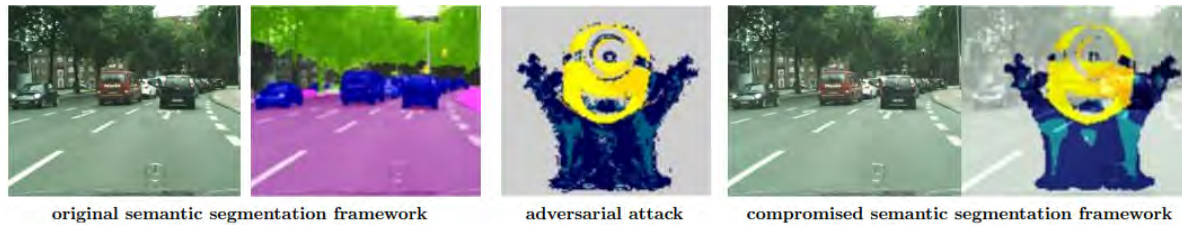


Figure 1: We cause the network to generate a *minion* as segmentation for the adversarially perturbed version of the original image. Note that the original and the perturbed image are indistinguishable.

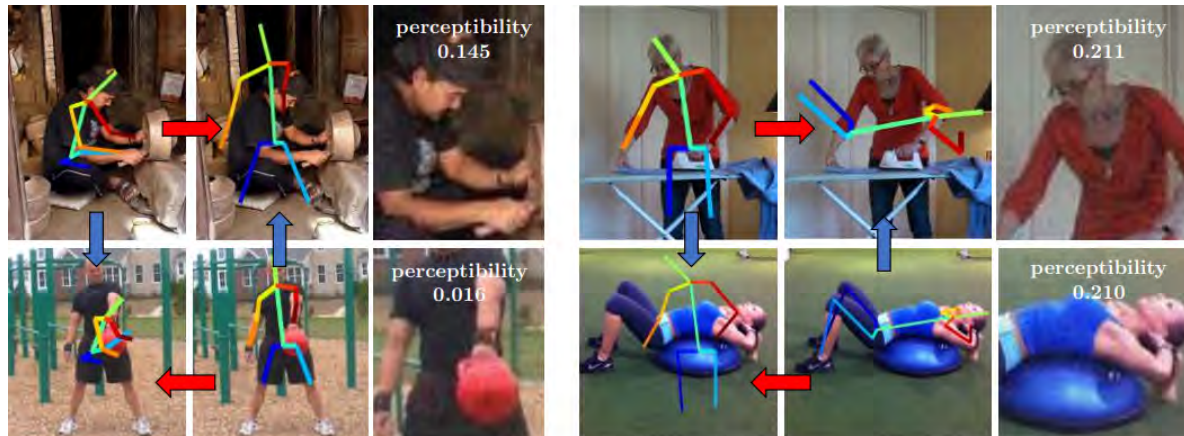
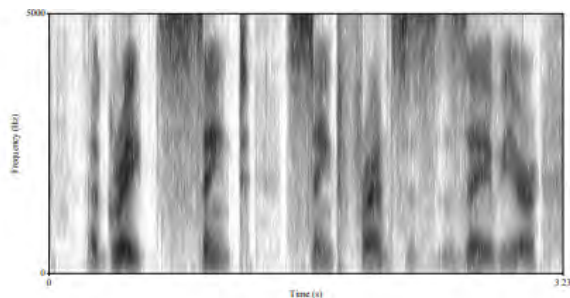
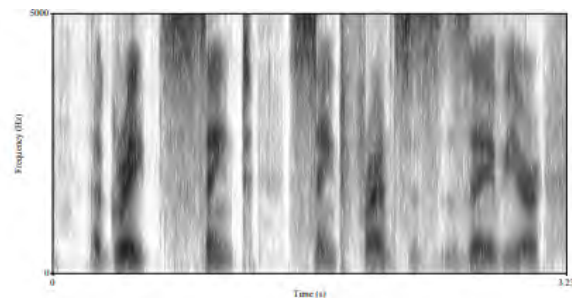


Figure 4: Examples of successful targeted attacks on a pose estimation system. Despite the important difference between the images selected, it is possible to make the network predict the wrong pose by adding an imperceptible perturbation.

(Cisse et al, 2017)



(a) a great saint saint francis zaviour

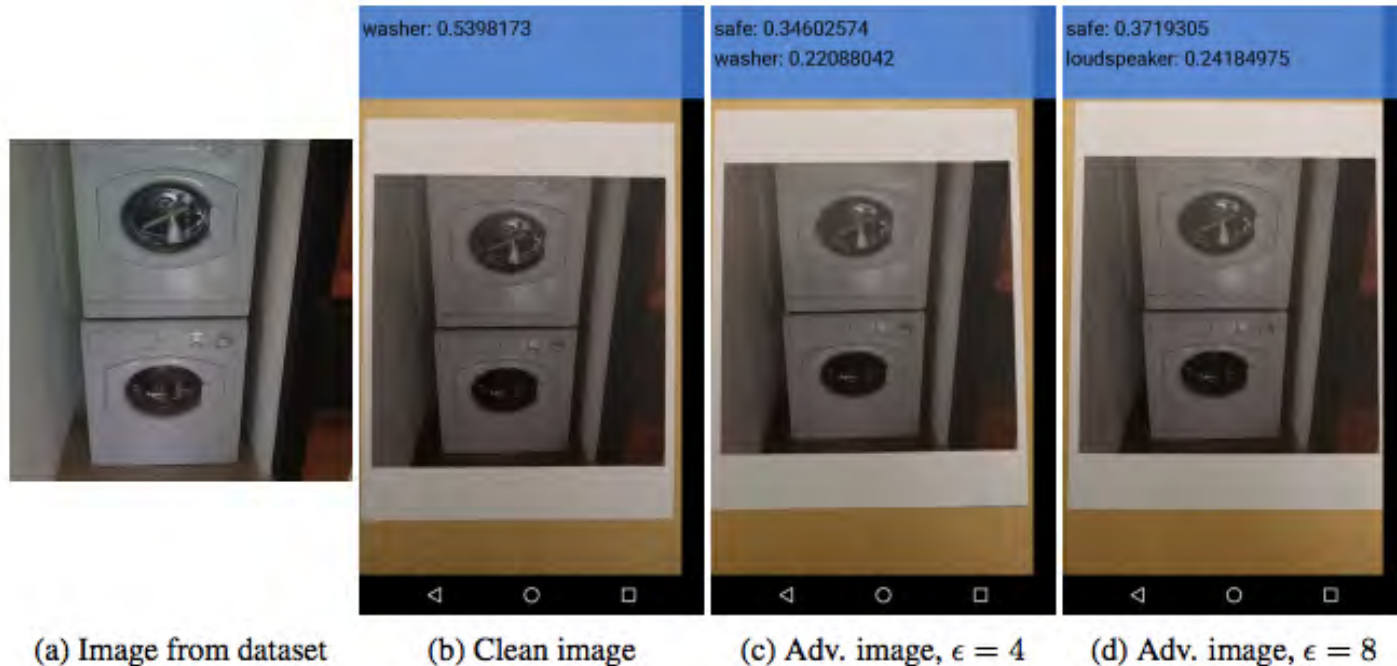


(b) i great sinkt shink t frimsuss avir

Figure 7: The model models' output for each of the spectrograms is located at the bottom of each spectrogram. The target transcription is: A Great Saint Saint Francis Xavier.

(Cisse et al, 2017)

Adversarial examples in the physical world



Adversarial examples can be printed out on normal paper and photographed with a standard resolution smartphone and still cause a classifier to, in this case, label a “washer” as a “safe”.



Adversarial Examples In The Physical World - ...



Watch later



Share

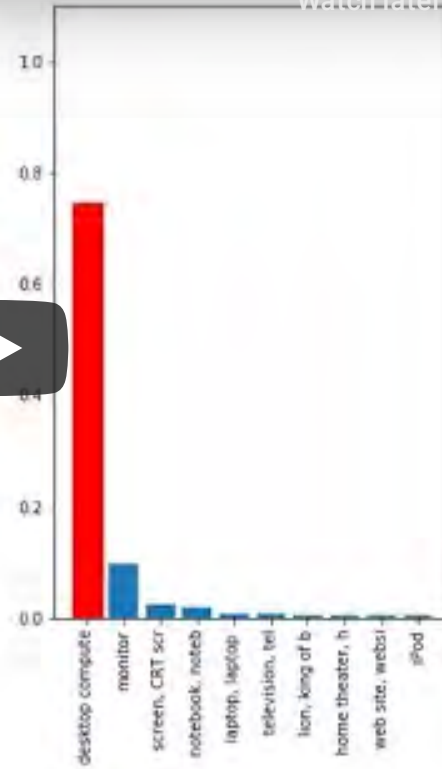




Physical Adversarial Example



Watch later Share



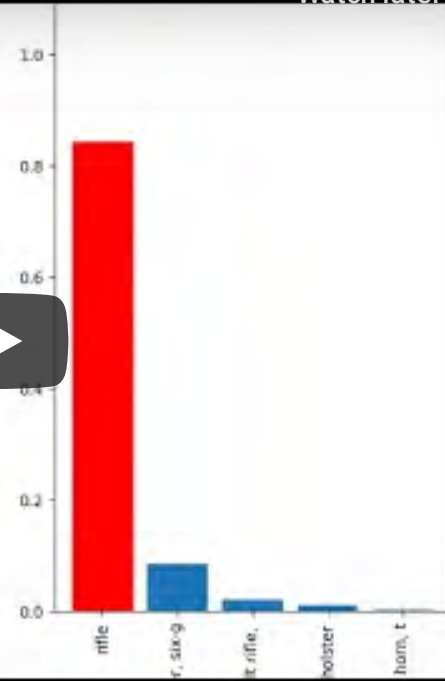
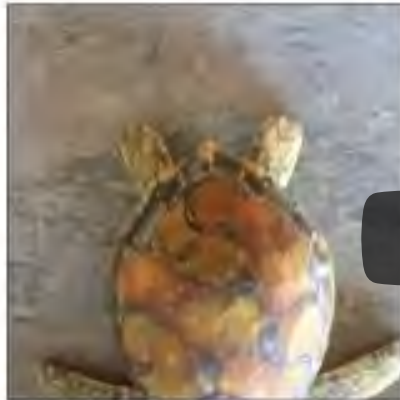


Synthesizing Robust Adversarial Examples: A...

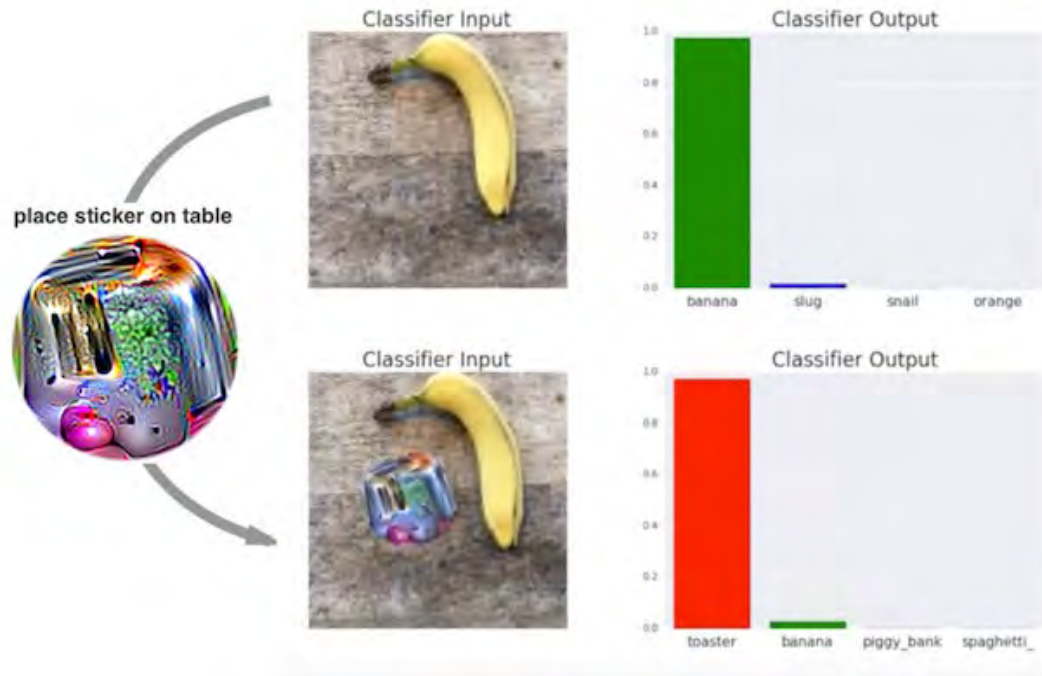


Watch later

Share



Adversarial patch



(Brown et al, 2017)

Creating adversarial examples

Locality assumption

"The deep stack of non-linear layers are a way for the model to encode a non-local generalization prior over the input space. In other words, it is assumed that is possible for the output unit to assign probabilities to regions of the input space that contain no training examples in their vicinity.

It is implicit in such arguments that local generalization—in the very proximity of the training examples—works as expected. And that in particular, for a small enough radius $\epsilon > 0$ in the vicinity of a given training input \mathbf{x} , an $\mathbf{x} + \mathbf{r}$ satisfying $\|\mathbf{r}\| < \epsilon$ will get assigned a high probability of the correct class by the model."

(Szegedy et al, 2013)

$$\min_{\mathbf{r}} \ell(y_{\text{target}}, f(\mathbf{x} + \mathbf{r}; \theta))$$

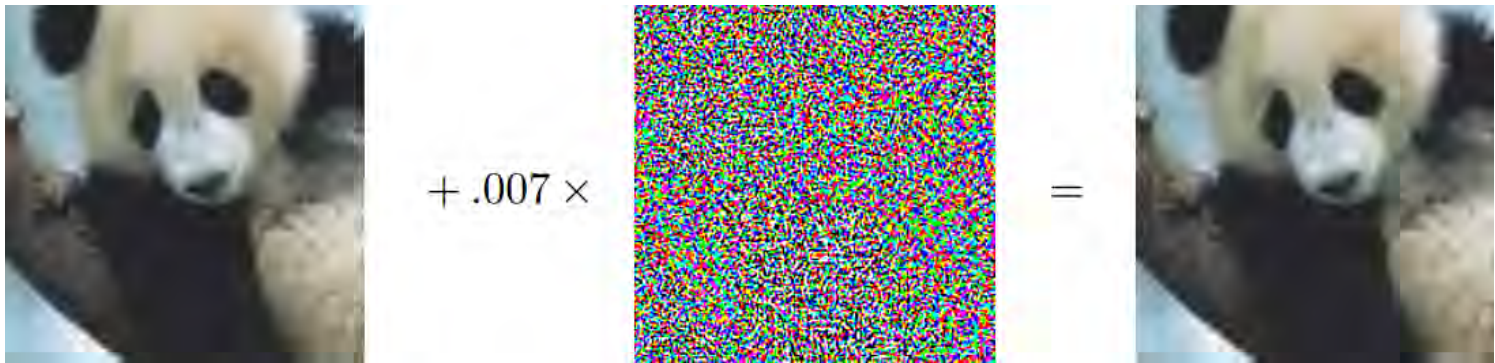
subject to $\|\mathbf{r}\| \leq L$

Fast gradient sign method

Take a step along the direction of the sign of the gradient at each pixel,

$$\mathbf{r} = \epsilon \operatorname{sign}(\nabla_{\mathbf{x}} \ell(y_{\text{target}}, f(\mathbf{x}; \theta))),$$

where ϵ is the magnitude of the perturbation.

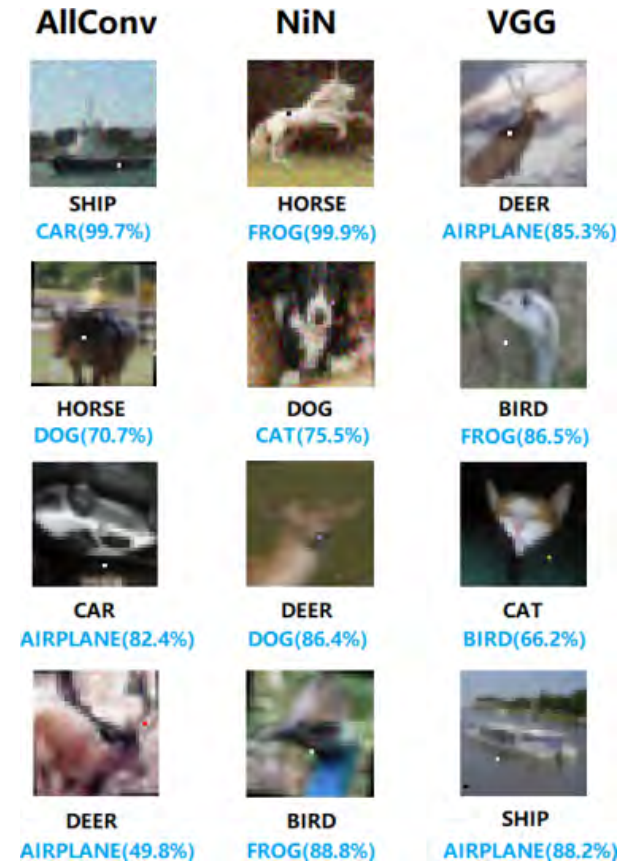


The panda on the right is classified as a 'Gibbon' (Goodfellow et al, 2014).

One pixel attacks

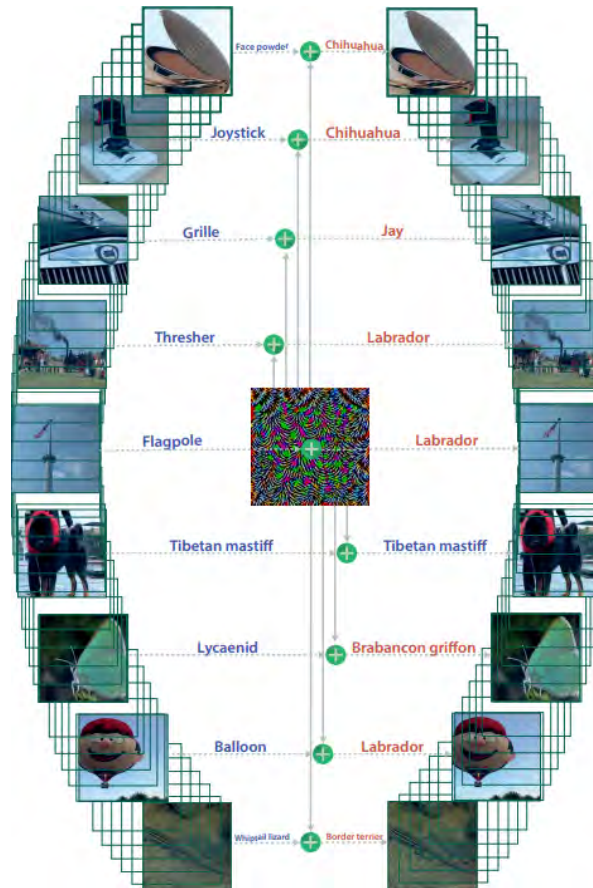
$$\min_{\mathbf{r}} \ell(y_{\text{target}}, f(\mathbf{x} + \mathbf{r}; \theta))$$

subject to $\|\mathbf{r}\|_0 \leq d$



(Su et al, 2017)

Universal adversarial perturbations



(Moosavi-Dezfooli et al, 2016)

Adversarial defenses

Security threat

Adversarial attacks pose a serious **security threat** to machine learning systems deployed in the real world.

Examples include:

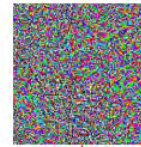
- fooling real classifiers trained by remotely hosted API (e.g., Google),
- fooling malware detector networks,
- obfuscating speech data,
- displaying adversarial examples in the physical world and fool systems that perceive them through a camera.



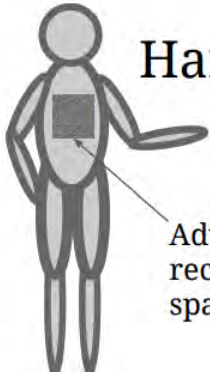
What if one puts adversarial patches on road signs?
Say, for a self-driving car?

Hypothetical attacks on self-driving cars

Denial of service



Confusing object



Harm others

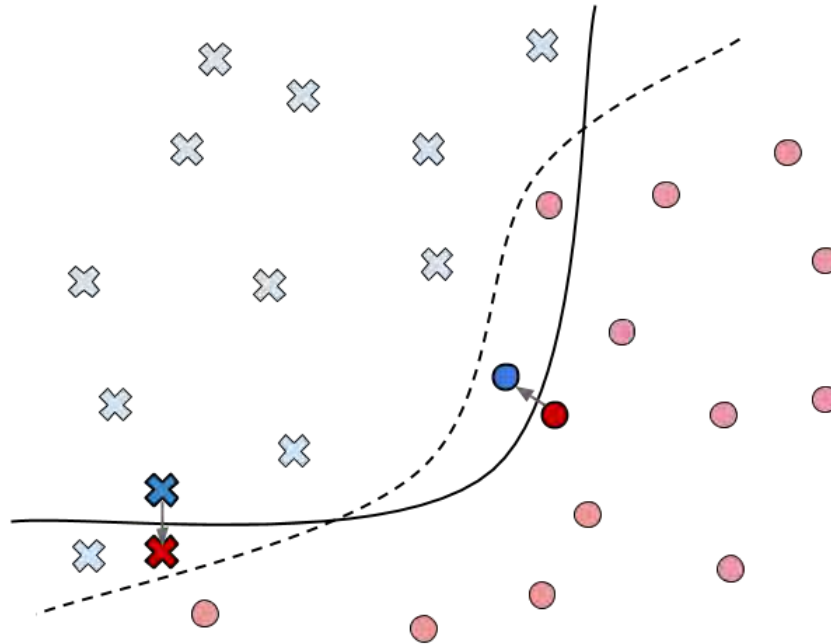
Adversarial input recognized as "open space on the road"

Harm self / passengers



Adversarial input recognized as "navigable road"

Origins of the vulnerability



----- Task decision boundary

———— Model decision boundary

⊗ Test point for class 1

⊗ Adversarial example for class 1

⊗ Training points for class 1

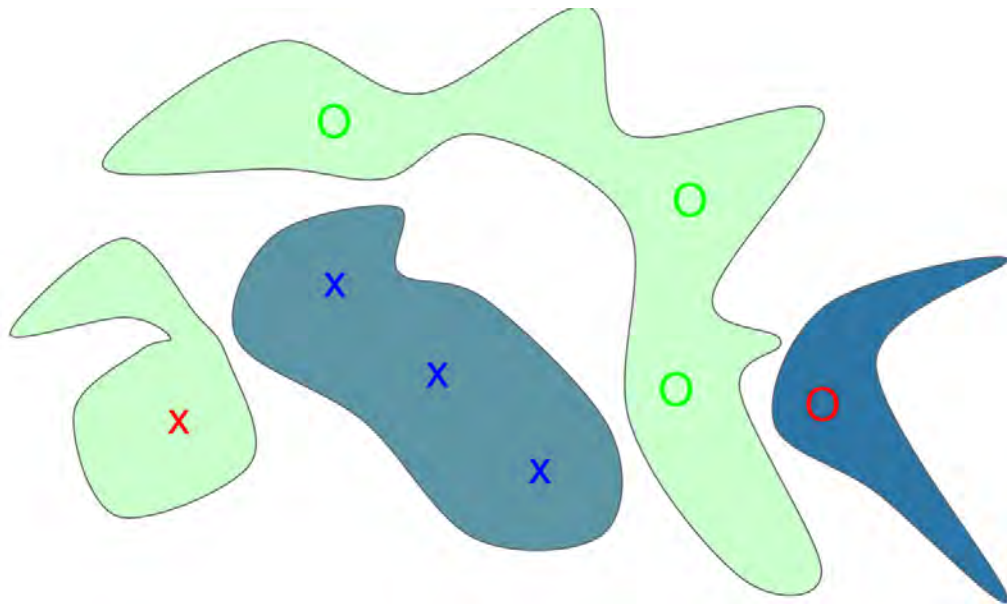
○ Training points for class 2

● Test point for class 2

● Adversarial example for class 2

Conjecture 1: Overfitting

Natural images are within the correct regions, but are also sufficiently close to the decision boundary.



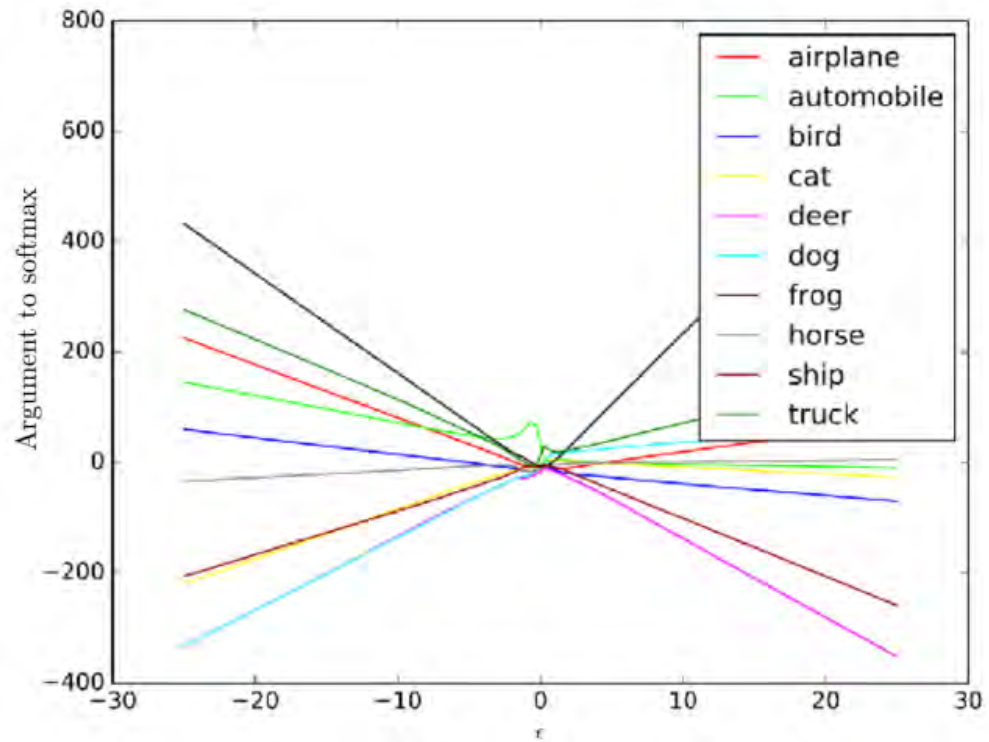
Conjecture 2: Excessive linearity

The decision boundary for most ML models, including neural networks, are near piecewise linear.

Then, for an adversarial sample $\hat{\mathbf{x}}$, its dot product with a weight vector \mathbf{w} is such that

$$\mathbf{w}^T \hat{\mathbf{x}} = \mathbf{w}^T \mathbf{x} + \mathbf{w}^T \mathbf{r}.$$

- The adversarial perturbation causes the activation to grow by $\mathbf{w}^T \mathbf{r}$.
- For $\mathbf{r} = \epsilon \text{sign}(\mathbf{w})$, if \mathbf{w} has n dimensions and the average magnitude of an element is m , then the activation will grow by ϵmn .
- Therefore, for high dimensional problems, we can make many infinitesimal changes to the input that add up to one large change to the output.



Empirical observation: neural networks produce nearly linear responses over ϵ .

Defense

- Data augmentation
- Adversarial training
- Denoising / smoothing

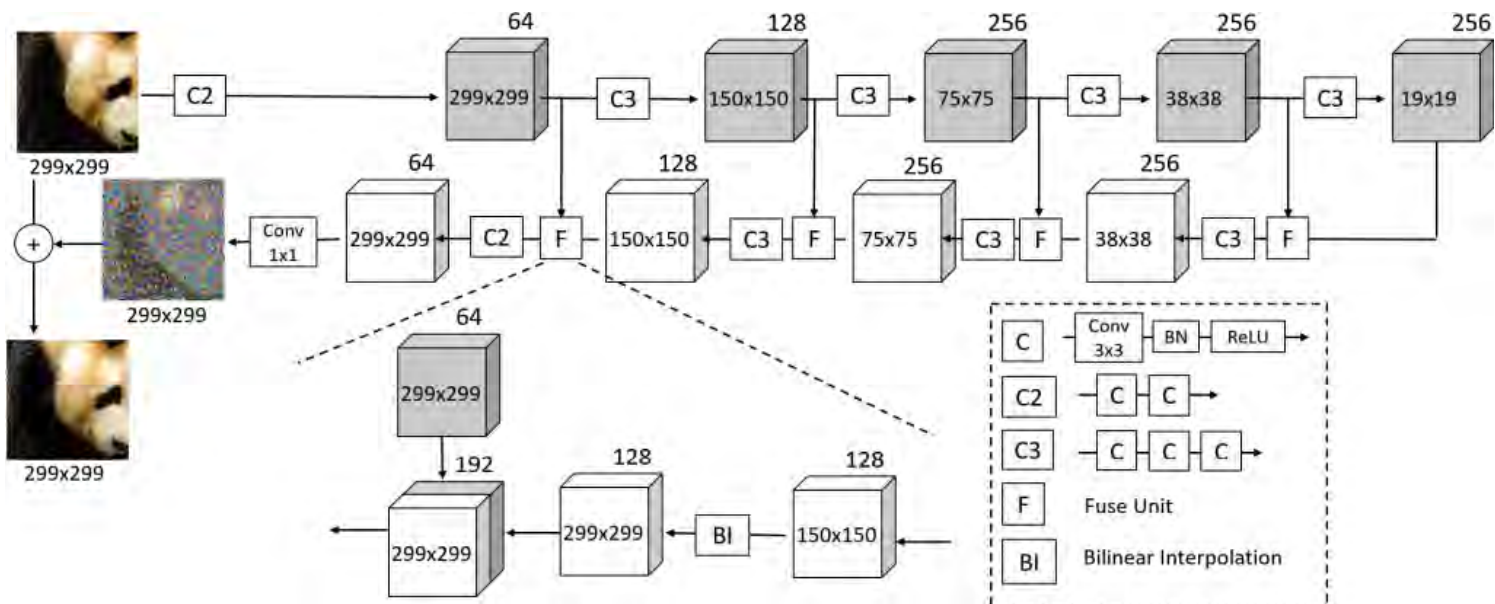
Adversarial training

Generate adversarial examples (based on a given attack) and include them as additional training data.

- **Expensive** in training time.
- Tends to **overfit the attack** used during training.

Denosing

- Train the network to remove adversarial perturbations before using the input.
- The winning team of the defense track of the NIPS 2017 competition trained a denoising U-Net to remove adversarial noise.



SHIELD  **Stochastic Local Quantization
Removes Adversarial Perturbations**

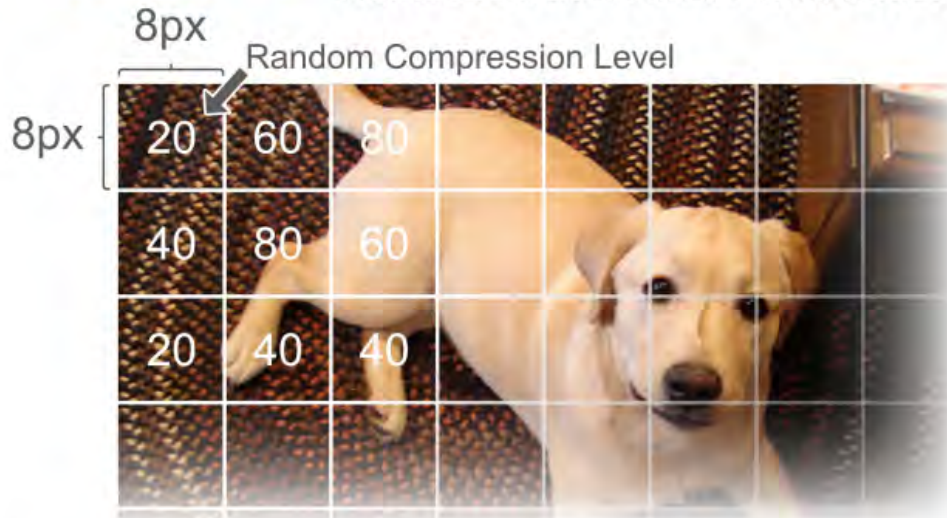
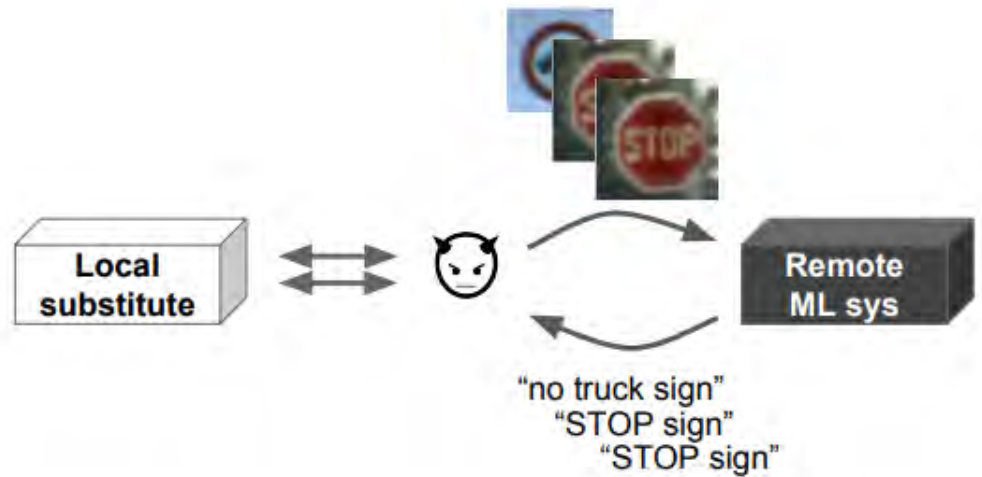


Figure 2: SHIELD uses Stochastic Local Quantization (SLQ) to remove adversarial perturbations from input images. SHIELD divides images into 8×8 blocks and applies a randomly selected JPEG compression quality (20, 40, 60 or 80) to each block to remove adversarial attacks. Note this figure is an illustration; our images are of actual size 299×299 .

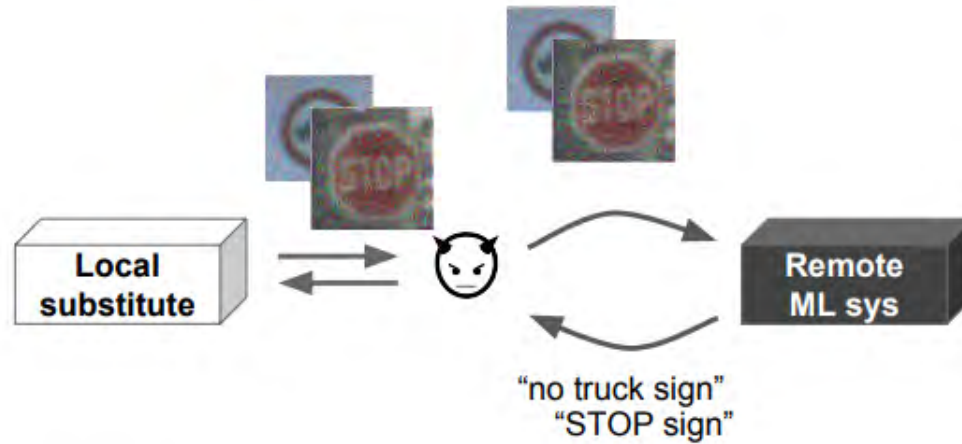
Hiding information

Attacks considered so far are **white-box** attacks, for which the attacker has full access to the model.

- What if instead the model internals remain hidden?
- Are models prone to **black-box** attacks?



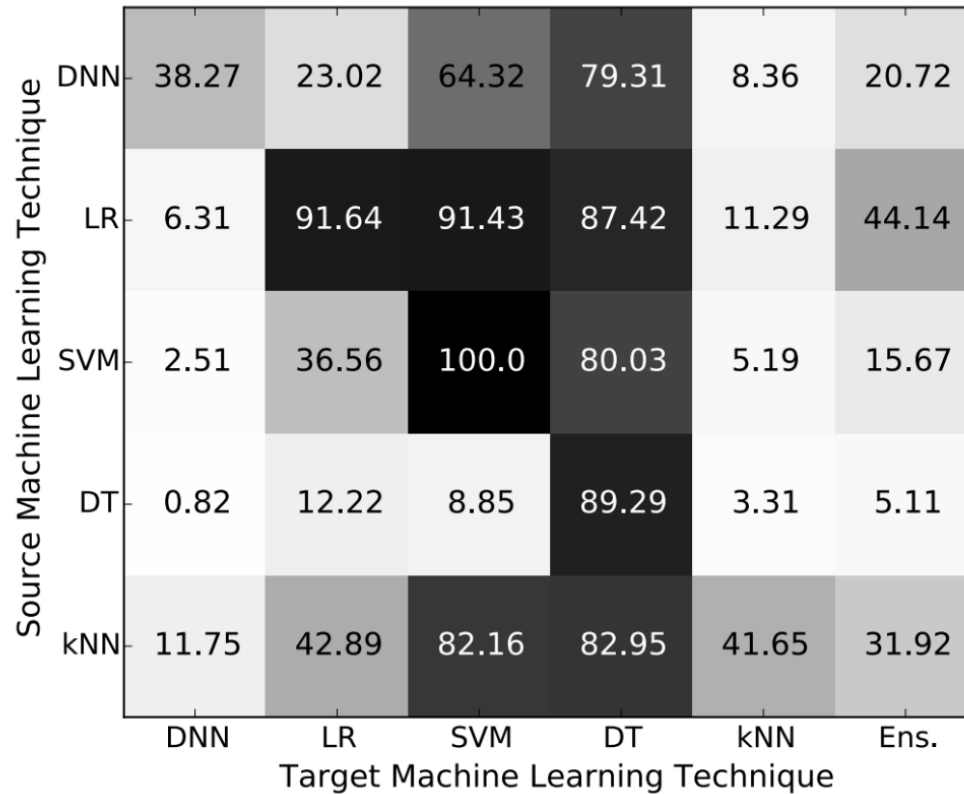
- (1) The adversary queries the target remote ML system for labels on inputs of its choice.
- (2) The adversary uses the labeled data to train a local substitute of the remote system.



(3) The adversary selects new synthetic inputs for queries to the remote ML system based on the local substitute's output surface sensitivity to input variations.

Transferrability

Adversarial examples are transferable across ML models!



Failed defenses

"In this paper we evaluate ten proposed defenses and demonstrate that none of them are able to withstand a white-box attack. We do this by constructing defense-specific loss functions that we minimize with a strong iterative attack algorithm. With these attacks, on CIFAR an adversary can create imperceptible adversarial examples for each defense.

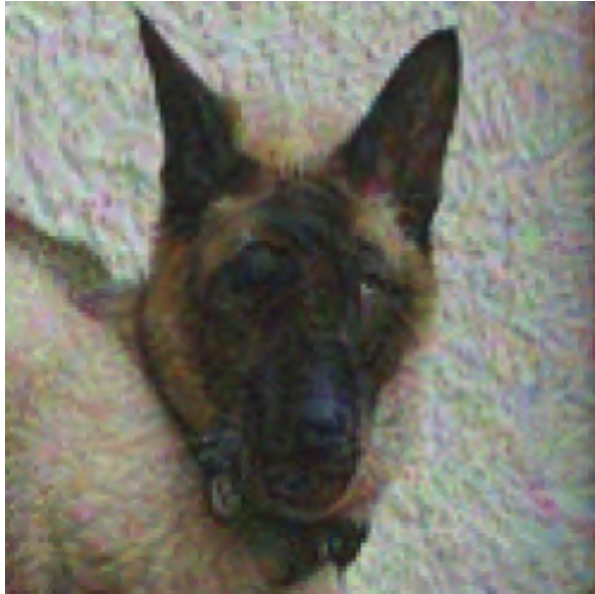
By studying these ten defenses, we have drawn two lessons: existing defenses lack thorough security evaluations, and adversarial examples are much more difficult to detect than previously recognized."

(Carlini and Wagner, 2017)

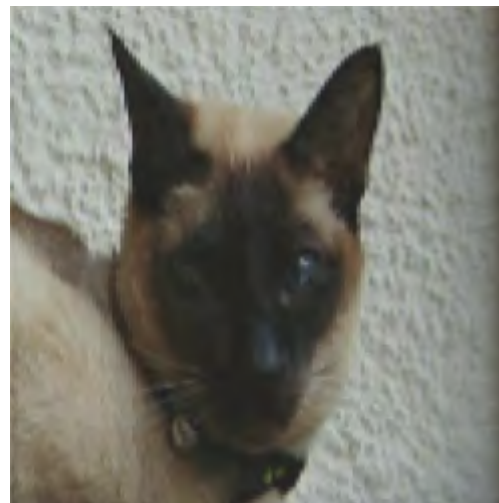
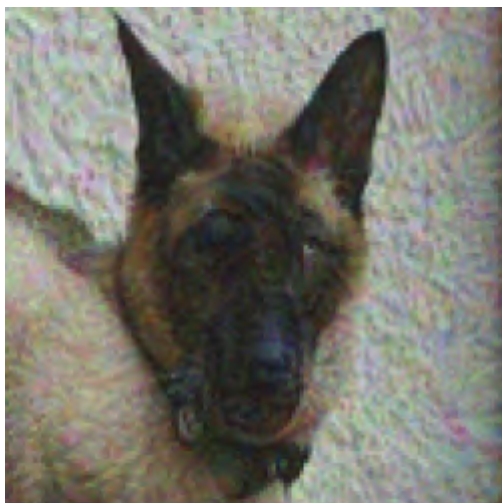
"No method of defending against adversarial examples is yet completely satisfactory. This remains a rapidly evolving research area."

(Kurakin, Goodfellow and Bengio, 2018)

Fooling both computers and humans



What do you see?



By building neural network architectures that closely match the human visual system, adversarial samples can be created to fool humans.

That's all folks!