Deep Learning

Lecture 2: Machine learning recap, history of neural networks and the main building blocks

Prof. Stéphane Gaïffas
https://stephanegaiffas.github.io

Machine learning recap

Supervised learning

We have data with labeled samples $(x_1, y_1), \ldots, (x_n, y_n)$

- Each $(x_i, y_i) \in \mathcal{X} \times \mathcal{Y}$ have an unknown joint distribution P
- The x_i are input features. We have often $\mathcal{X} = \mathbb{R}^d$.
- The y_i are output labels. We have often $\mathcal{Y} = \mathbb{R}$ (a regression problem, the labels do have a natural ordering) or $\mathcal{Y} = \{1, \ldots, K\}$ (a K-class classification problem, the labels are categorical and do not have a natural ordering)

The samples are supposed independent and identically distributed (i.i.d)

The training data can be of any finite size n.

In general, we do not have any prior information about P.

In most cases, x_i is a vector, but it could be an image, a piece of text, a sample of sound, a video

Supervised learning

A predictor is a measurable function

$$f : \mathcal{X} \to \widehat{\mathcal{Y}}$$

where

$$\widehat{\mathcal{Y}} = \mathcal{Y} \quad \text{or} \quad \widehat{\mathcal{Y}} = \Delta(\mathcal{Y}),$$

with $\Delta(\mathcal{Y})$ the set of distributions on $\mathcal{Y}$. Indeed, in classification problems, we often prefer to predict the distribution of y given x

$$p(y|x) = \mathbb{P}[Y = y|X = x] \in \Delta(\mathcal{Y})$$

instead of just predicting $y \in \mathcal{Y}$.

Training

Training or learning a predictor f is done using training data using the empirical risk minimization principle

Empirical risk minimization

We train the predictor by minimization of the the empirical risk or training error

$$R_n(f) = \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f(x_i))$$

over $f \in \mathcal{F}$ where

- $\mathcal{F}$ is a parametrized set of functions. In linear methods we use the set of functions $\mathcal{F} = \mathcal{F}_{\text{linear}} = \{x \mapsto x^\top w : w \in \mathbb{R}^d\}$.
- $\ell : \mathcal{Y} \times \widehat{\mathcal{Y}} \to \mathbb{R}$ is a loss function

Examples

- For linear least-squares we take $\mathcal{F} = \mathcal{F}_{\text{linear}}$ and use the square loss $\ell(y, y') = \frac{1}{2}(y - y')^2$
- For logistic regression (for binary classification where $\mathcal{Y} = \{-1, 1\}$) we take $\mathcal{F} = \mathcal{F}_{\text{linear}}$ and the logistic loss $\ell(y, y') = \log(1 + e^{-yy'})$

Generalization

The aim of a predictor is generalization. We want the generalization error

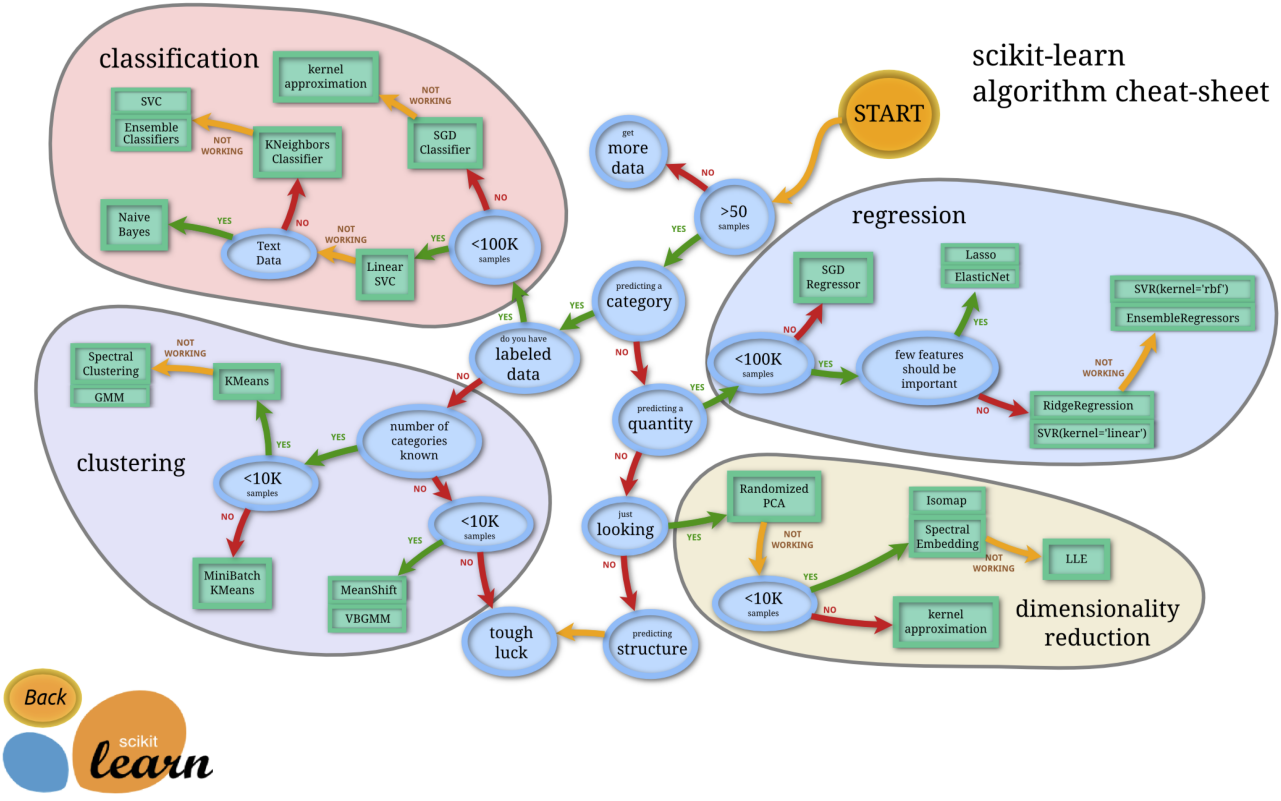
$$R(f) = \mathbb{E}_{(x,y)\sim P}\big[\ell(y, f(x))\big]$$

to be as small as possible.

- The empirical risk $R_n(f)$ is used as a way of approximating the generalization error using the training samples
- Most machine learning algorithms, including neural networks, use empirical risk minimization
- But only minimizing the training error usually leads to overfitting, most methods require cross-validation for hyper-parameters tuning

Of course, many approaches: linear methods, kernels, k-NN, ensemble methods (boosting, forests), etc. and unsupervised methods

Scikit-learn cheat-sheet



From: https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

History of neural networks

Modeling neurons

- The idea of neural networks began as a model of how neurons work in the brain.
- Called connectionism and used connected circuits to simulate intelligent behavior

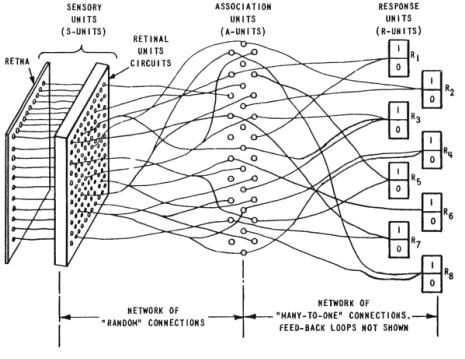


Figure 1 ORGANIZATION OF THE MARK I PERCEPTRON

- In 1943, portrayed with a simple electrical circuit by neurophysiologist Warren McCulloch and mathematician Walter Pitts.
- Donald Hebb took the idea further by proposing that neural pathways strengthen over each successive use, especially between neurons that tend to fire at the same time.

"A logical calculus of the ideas immanent in nervous activity", McCulloch and Pitts 1943
"The organization of behavior: A neuropsychological theory", Hebb 2005

Modeling neurons

- Around 50', Frank Rosenblatt, a psychologist at Cornell, was working on understanding the comparatively simpler decision systems present in the eye of a fly, which underlie and determine its flee response.
- In an attempt to understand and quantify this process, he proposed the idea of a Perceptron in 1958, calling it Mark I Perceptron.
- It was a system with a simple input output relationship, modeled on a McCulloch-Pitts neuron to explain the complex decision processes in a brain using a linear threshold gate.

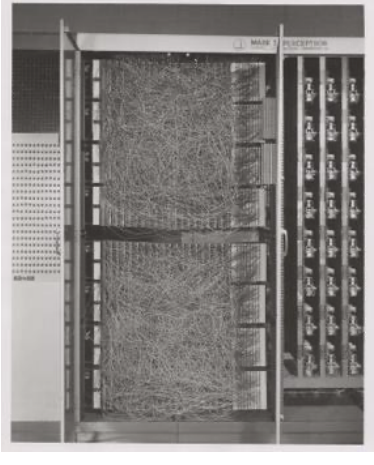
Credits: Frank Rosenblatt, Mark I Perceptron operators' manual, 1960.

First perceptron

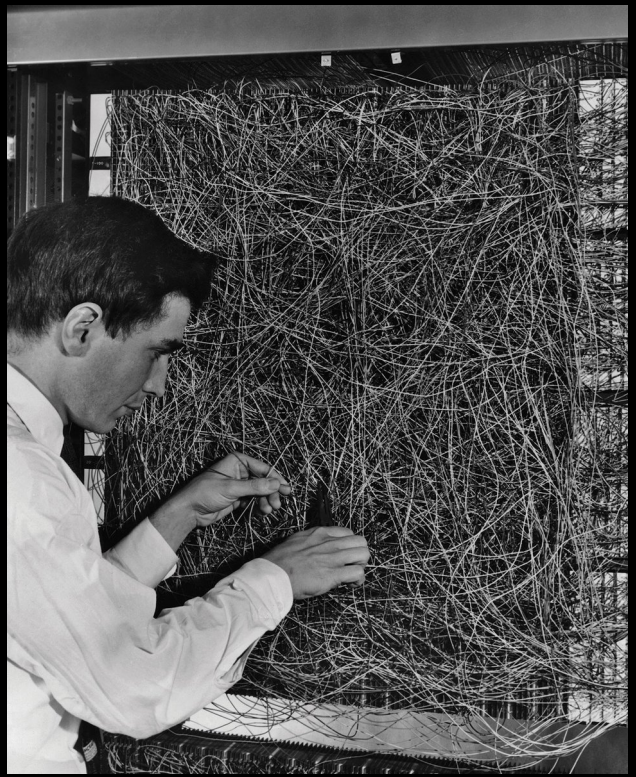
A McCulloch-Pitts neuron takes in inputs, takes a weighted sum and returns 0 if the result is below threshold and 1 otherwise.

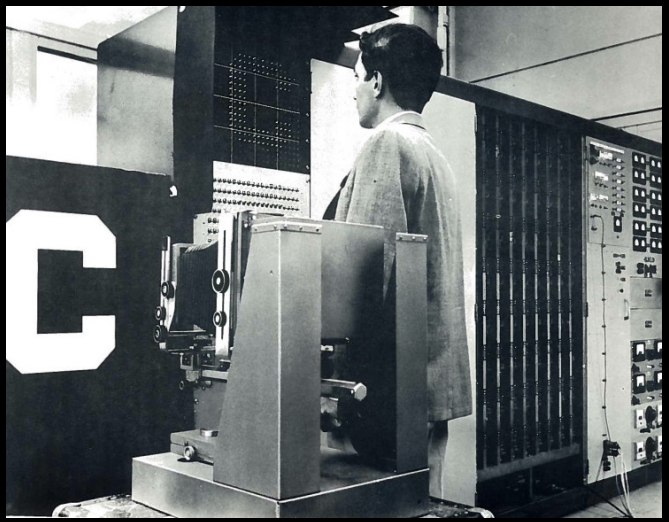
$$f(x) = \begin{cases} 1 & \text{if } \sum_i w_i x_i + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

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

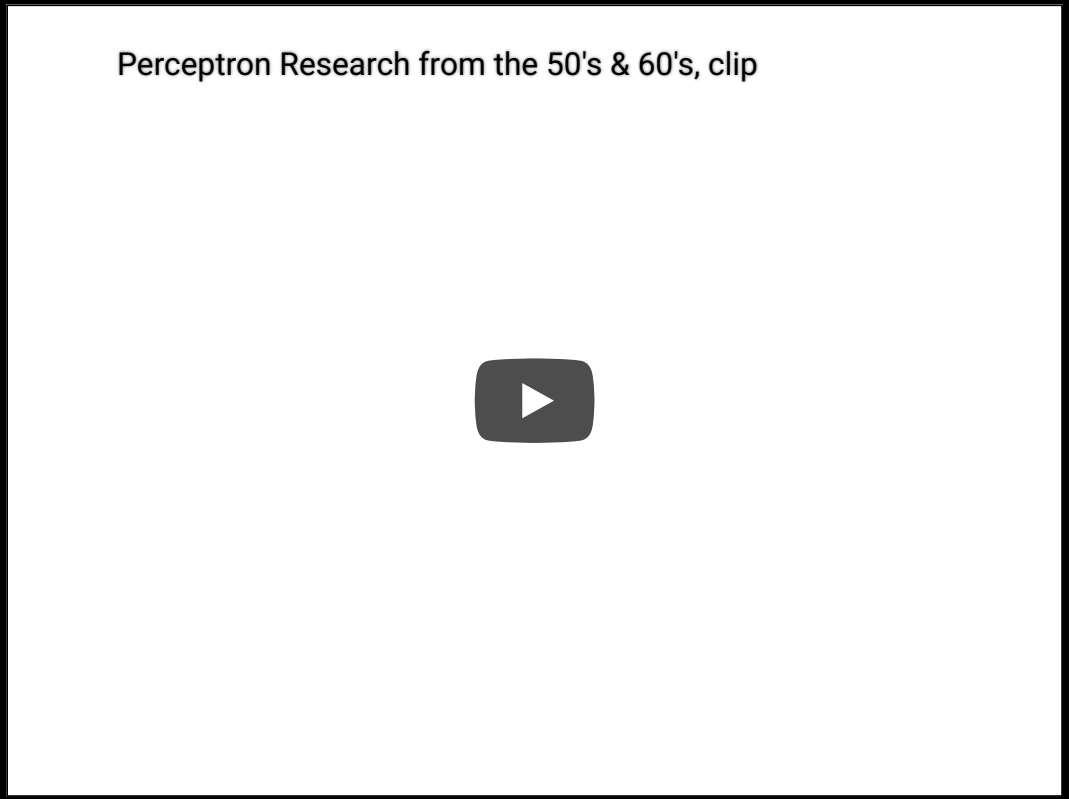


- The Mark I Perceptron was the first implementation of the perceptron algorithm.
- The machine was connected to a camera that used 20x20 photocells to produce a 400-pixel image.





The Mark I Percetron (Frank Rosenblatt).



The Perceptron

Perceptron model and algorithm

It uses the following model

$$f(x) = \sigma\Big(\sum_j w_j x_j + b\Big)$$

where x_j are inputs, w_j are weights, b is a bias and activation $\sigma(z) = \mathbf{1}(z > 0)$

- How to find (w, b) ?

The perceptron algorithm (first iterative learning algorithm)

To ease notations, put $w = [w_1 \cdots w_d \; b]$ and $x_i = [x_i \; 1]$

- Start with $w \leftarrow 0$
- Repeat over all samples:
  1. if $y_i \; x_i^\top w < 0$ put $w \leftarrow w + y_i x_i$
  2. otherwise do not modify w

Perceptron model and algorithm

- This procedure can be seen as a stochastic gradient descent method
- Indeed, a sample i is missclassified if $y_i x_i^\top w < 0$ (labels -1 or $+1$), so we can consider the goodness-of-fit

$$F(w) = \sum_{i \,:\, y_i x_i^\top w < 0} F_i(w) \quad \text{where} \quad F_i(w) = -y_i x_i^\top w$$

- But $\nabla F_i(w) = -y_i x_i$ so that a stochastic gradient descent step write

$$w \leftarrow w - \eta \nabla F_i(w) = w + \eta y_i x_i$$

  where we sample uniformly at sample i among $\{i : y_i x_i^\top w < 0\}$.

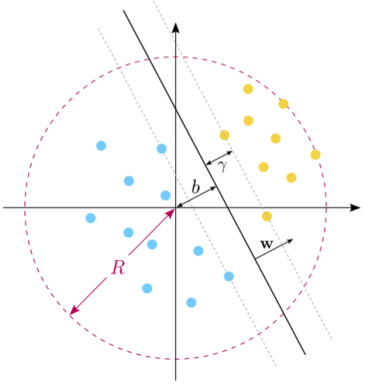
- We use $\eta = 1$ in the Perceptron algorithm

Perceptron model and algorithm

Let $R = \max_i |x_i|$ and $w^\star$ be the optimal hyperplane of margin

$$\gamma = \min_i y_i x_i^\top w^\star$$

with $|w^\star| = 1$. We have the following theorem.



Theorem (Block 1962, Novikoff 1963)

Assume that the training dataset $(x_1, y_1), \ldots, (x_n, y_n)$ is linearly separable (meaning that $\gamma > 0$). Put $w_0 \leftarrow 0$, then the number of steps in the perceptron algorithm is bounded by

$$k \leq \frac{1 + R^2}{\gamma^2}.$$

Exercice: prove it ! ($\approx$ 10 lines)

Credits: From http://image.diku.dk/%20kstensbo/notes/perceptron.png

History

- The Perceptron project led by Rosenblatt was funded by the US Office of Naval Research.

"The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence. Later perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech and writing in another language, it was predicted."

Press conference, 7 July 1958, New York Times.

- For an extensive study of the perceptron, see [Principles of neurodynamics. perceptrons and the theory of brain mechanisms, Rosenblatt 1961]

Moving forward: (M)ADALINE

- In 1959 at Stanford, Bernard Widrow and Marcian Hoff developed AdaLinE (ADAptive LINear Elements) and MAdaLinE (Multiple AdaLinE), the latter being the first network successfully applied to a real world problem

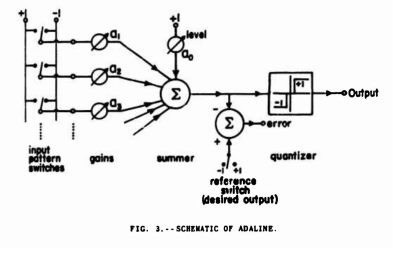


FIG. 3.--SCHEMATIC OF ADALINE.

Main differences with the perceptron

- The loss is the square difference between the sum of the weighted input and the output
- The optimization procedure is a gradient descent: all computations are trivial since the weighted sum is linear as a function of weights

Adaptive switching circuits, Bernard Widrow and Hoff 1960

MADALINE

Many Adalines: network with one hidden layer composed of many Adaline units. [1]

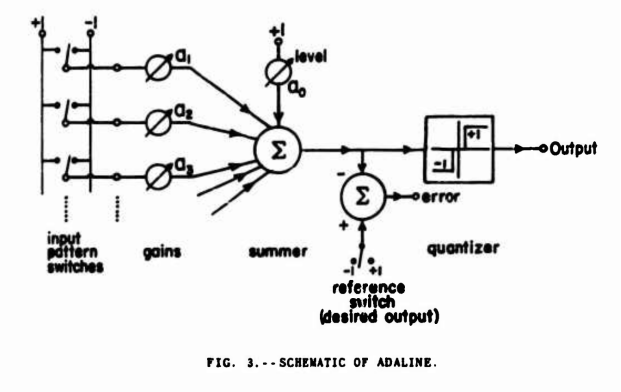


FIG. 3.--SCHEMATIC OF ADALINE.

Applications

- Speech and pattern recognition [2]
- Weather forecasting [3]
- Adaptive filtering and adaptive signal processing [4]

---

[1] "Madaline Rule II: a training algorithm for neural networks", Winter and Widrow 1988
[2] "Real-Time Adaptive Speech-Recognition System", Talbert et al. 1963
[3] "Application of the adaline system to weather forecasting", Hu 1964
[4] "Adaptive signal processing", Bernard and Samuel 1985

Further

- The kernel perceptron algorithm was already introduced by [1]
- Margin bounds for the Perceptron algorithm in the general non-separable case were proven by [2] and by [3] who extended existing results and gave new L1 bounds

---

[1] "Theoretical foundations of the potential function method in pattern recognition learning", Aizerman 1964
[2] "Large margin classification using the perceptron algorithm", Freund and Schapire 1999
[3] "Perceptron mistake bounds", Mohri and Rostamizadeh 2013

AI Winter

1969: Minsky and Papert exhibit the fact that it was difficult for perceptron to detect parity (number of activated pixels) and connectedness (are the pixels connected?). Besides, it was known that they cannot represent simple non linear function such as XOR function, see [1]

> There is no reason to suppose that any of [the virtue of perceptrons] carry over to the many-layered version. Nevertheless, we consider it to be an important research problem to elucidate (or reject) our intuitive judgement that the extension is sterile. Perhaps some powerful convergence theorem will be discovered, or some profound reason for the failure to produce an interesting "learning theorem" for the multilayered machine will be found.
>
> Minsky, Papert

- The starting point of "AI winter": a significant decline in funding of neural network research
- In [2]: more about the controversy between Rosenblatt and Minsky, Papert

---

[1] "Perceptrons: an introduction to computational geometry", Minsky and Papert 2017
[2] "A sociological study of the official history of the perceptrons controversy", Olazaran 1996.

AI Winter

Perceptron model is defined as

$$f(x) = \sigma\Big(\sum_i w_i x_i + b \geq 0\Big)$$

where $\sigma(z) = \mathbf{1}(z > 0)$ is the heavyside activation. Assuming boolean inputs $x_i \in \{0, 1\}$ it can implement

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

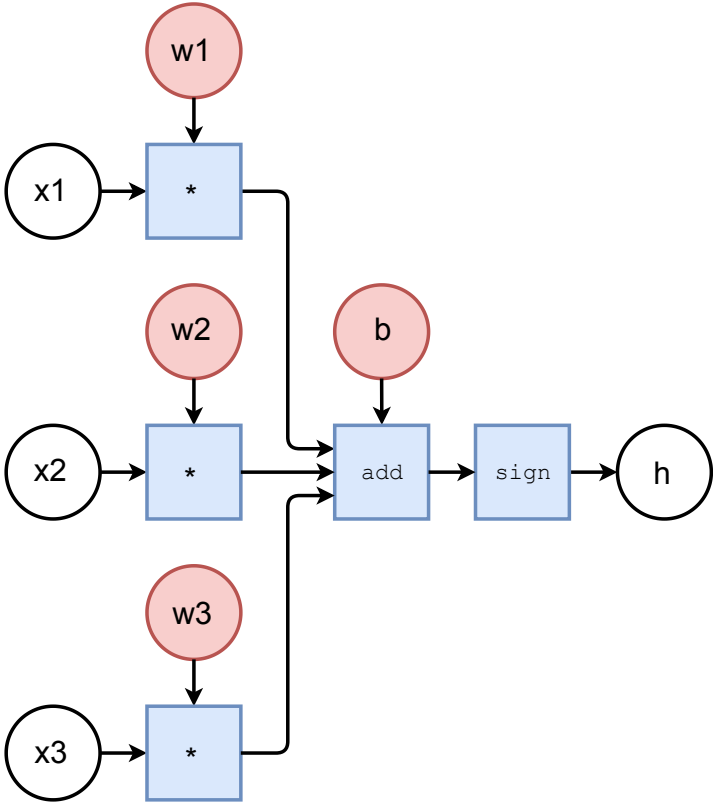
but not the XOR which is not linearly separable: $\text{xor}(a, b) = 1$ if $a = b = 0$ or if $a = b = 1$ and $\text{xor}(a, b) = 0$ otherwise.

Question: find a neural network with one hidden layer that implements the xor (it exists...)

Main building blocks

The MLP, computational graphs, SGD and backpropagation

Computational graphs



The computation of

$$f(x) = \sigma\Big(\sum_i w_i x_i + b\Big)$$

can be represented as a computational graph where

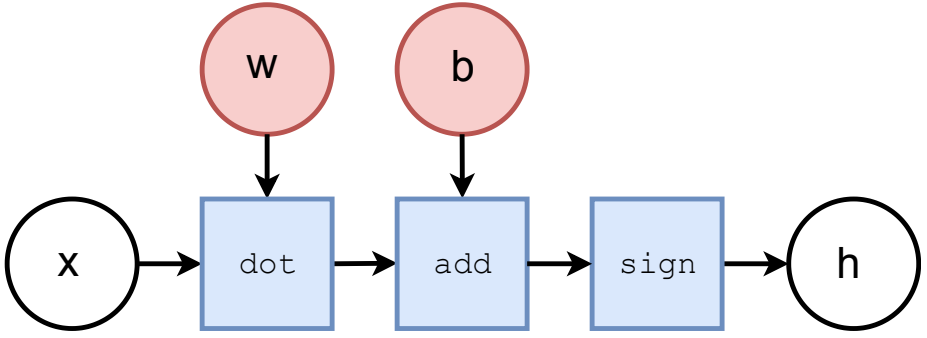
- white nodes correspond to inputs and outputs;
- red nodes correspond to model parameters;
- blue nodes correspond to intermediate operations.

Computational graphs

In terms of tensor operations, f can be rewritten as

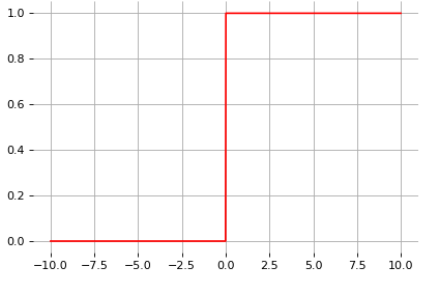
$$f(x) = \sigma(x^\top w + b),$$

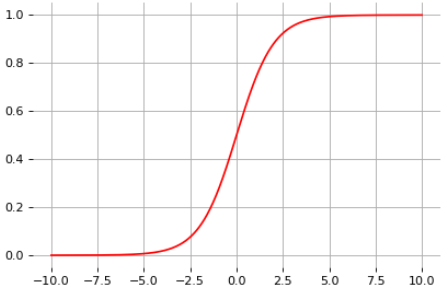
for which the corresponding computational graph of f is:



From perceptron to logistic unit

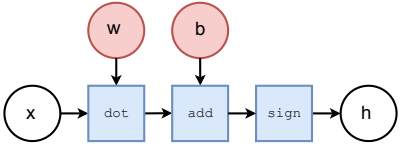
The sigmoid function looks a lot like the heavyside function





- But sigmoid has a non-flat gradient (and comes from sound statistical arguments).

Important. The $x \mapsto \sigma(x^\top w + b)$ unit is the primitive of all neural networks!



Stochastic gradient descent (SGD)

The empirical risk minimization principle leads to a goodness-of-fit of the form

$$F(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(x_i; \theta))$$

where f is the considered model (NN architecture) and θ encompasses all the trainable weights. Its gradient writes

$$\nabla F(\theta) = \frac{1}{n} \sum_{i=1}^{n} \nabla \ell(y_i, f(x_i; \theta))$$

- This is called a full gradient or batch gradient: it uses all the training data (complexity grows linearly with the size n of the dataset)
- This is baaaaad if n is large !
- Idea: this empirical risk is an approximation of the expected risk, no need to minimize it with great accuracy

Stochastic gradient descent (SGD)

Stochastic gradient descent uses instead stochastic gradients

$$\frac{1}{|B|}\sum_{i\in B}\nabla\ell(y_i, f(x_i;\theta))$$

where $B \subset \{1,\dots,n\}$ is sampled uniformly at random (at each step) with a mini-batch size $m = |B| \ll n$. One step of vanilla SGD is

$$\theta_{t+1} \leftarrow \theta_t - \eta_t \frac{1}{|B_t|}\sum_{i\in B_t}\nabla\ell(y_i, f(x_i;\theta_t))$$

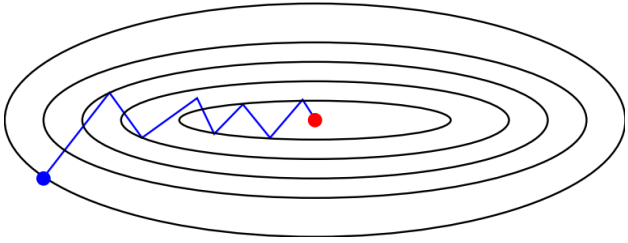
where η_t is the learning rate and B_t is the mini-batch used in the t-th iteration

- Iteration complexity is independent of n.
- The stochastic process $\{\theta_t : t = 1, 2, \dots\}$ depends on the mini-batches sampled at each iteration
- In practice, shuffle $\{1,\dots,n\}$ and use sequentially $B_t = \{1,\dots,m\}$, $B_{t+1} = \{m+1,\dots,2m\}$, etc.

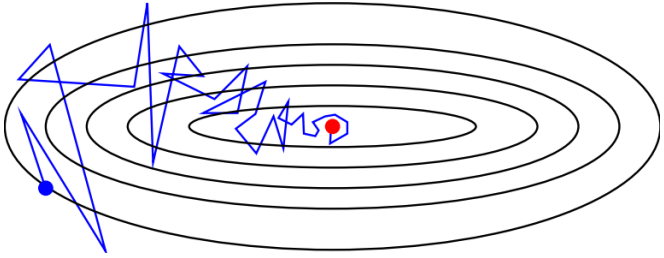
Stochastic gradient descent (SGD)

$$\theta_{t+1} \leftarrow \theta_t - \eta_t \frac{1}{|B_t|} \sum_{i \in B_t} \nabla \ell(y_i, f(x_i; \theta_t))$$

- Iteration complexity is independent of n.
- The stochastic process $\{\theta_t : t = 1, 2, \ldots\}$ depends on the mini-batches sampled at each iteration
- In practice, shuffle $\{1, \ldots, n\}$ and use sequentially $B_t = \{1, \ldots, m\}$, $B_{t+1} = \{m + 1, \ldots, 2m\}$, etc.



Batch gradient descent



Stochastic gradient descent

Stochastic gradient descent (SGD)

Why is stochastic gradient descent a good idea?

- Stochastic gradients are unbiased approximations of the batch gradient (uniform sampling of the mini-batch)
- Many guarantees on SGD and more advanced variants in the convex case
- If training is limited to single pass over the data, then SGD directly minimizes the expected risk
- Bottou and Bousquet [1]: stochastic optimization algorithms (e.g., SGD) yield the best generalization performance (in terms of excess error) despite being the worst optimization algorithms for minimizing the empirical risk

---

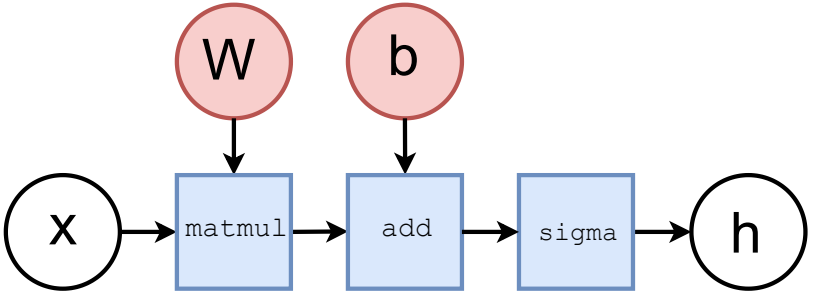
[1] Bottou, L. and Bousquet, O. (2008). "The tradeoffs of large scale learning."" In Advances in neural information processing systems (pp. 161-168)

Multi-layer perceptron

- So far we considered the logistic unit $h = \sigma(w^\top x + b)$, where $h \in \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^d, w \in \mathbb{R}^d$ and $b \in \mathbb{R}$.
- These units can be composed in parallel to form a layer with q outputs:

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

  where $h \in \mathbb{R}^q, x \in \mathbb{R}^d, \mathbf{W} \in \mathbb{R}^{d \times q}, b \in \mathbb{R}^q$ and where $\sigma(\cdot)$ is applied entry-wise.
- We also say that the width or number of units/neurons of the layer is q. It's the output size of the layer.



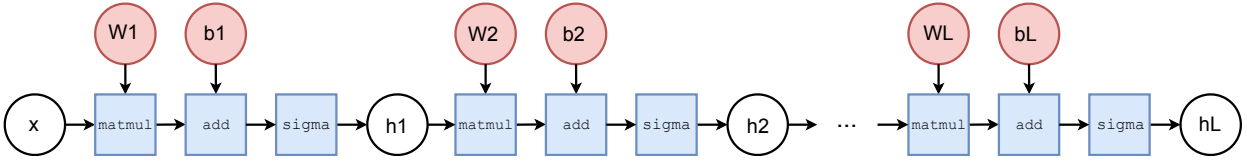
Multi-layer perceptron

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

$$
\begin{aligned}
h_0 &= x \\
h_1 &= \sigma(\mathbf{W}_1^\top h_0 + b_1) \\
&\vdots \\
h_L &= \sigma(\mathbf{W}_L^\top h_{L-1} + b_L) \\
f(\mathbf{x}; \theta) = \hat{y} &= h_L
\end{aligned}
$$

where θ denotes the model parameters $\{\mathbf{W}_k, b_k : k = 1, \ldots, L\}$.

- This model is the multi-layer perceptron
- Also known as the fully connected feedforward network.



Output layer

- For binary classification, the width of the last layer L is set to $q = 1$, which results in a single output $h_L \in [0, 1]$ that models the probability $\mathbb{P}[Y = 1|X = x]$.
- For C-class classification with $C > 2$, the activation function σ in the last layer is generalized to produce a vector $h_L \in \Delta$ where Δ is the probability simplex in $\mathbb{R}^C$ for the probability estimates $\mathbb{P}[Y = c|X = x]$ for $c = 1, \ldots, C$.

  This activation is the $\mathrm{softmax}$ function, where its i-th output is defined as

$$\mathrm{softmax}(z)_i = \frac{\exp(z_i)}{\sum_{c=1}^{C} \exp(z_c)},$$

  for $c = 1, \ldots, C$.

We need to compute gradients !

- To minimize $F(\theta)$ with SGD, given a mini-batch B, we need to compute (stochastic) gradients

$$\nabla F^B(\theta) = \frac{1}{|B|} \sum_{i \in B} \nabla \ell(y_i, f(x_i; \theta))$$

  with respect to θ

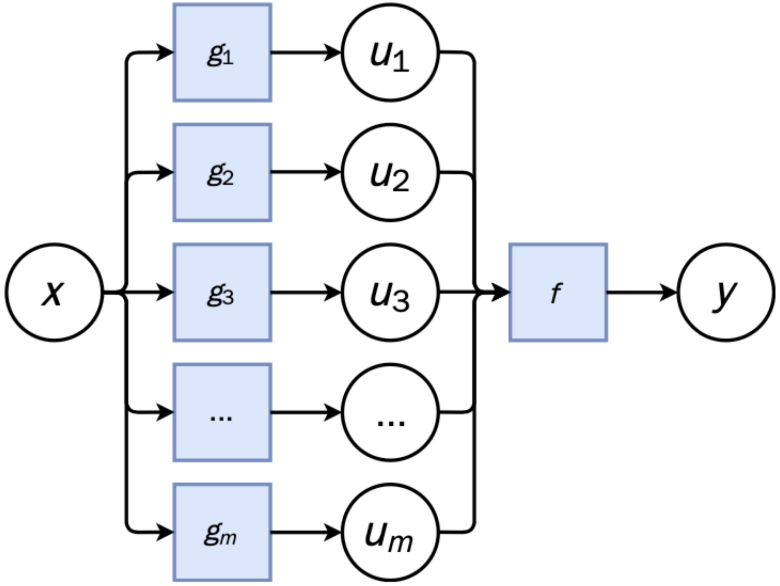
- Therefore, we require the evaluation of the (total) derivatives

$$\frac{\mathrm{d}F^B}{\mathrm{d}\mathbf{W}_k} \quad \text{and} \quad \frac{\mathrm{d}F^B}{\mathrm{d}\mathbf{b}_k}$$

  of F^B with respect to all model parameters $\mathbf{W}_k, \mathbf{b}_k$, for $k = 1, \ldots, L$

- These derivatives can be evaluated automatically from the computational graph of F using automatic differentiation

Chain rule



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

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

Chain rule

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

For the total derivative, the chain rule generalizes to

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \sum_{k=1}^{m} \frac{\partial y}{\partial u_k} \underbrace{\frac{\mathrm{d}u_k}{\mathrm{d}x}}_{\text{recursive case}}$$

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 or backpropagation

Backpropagation

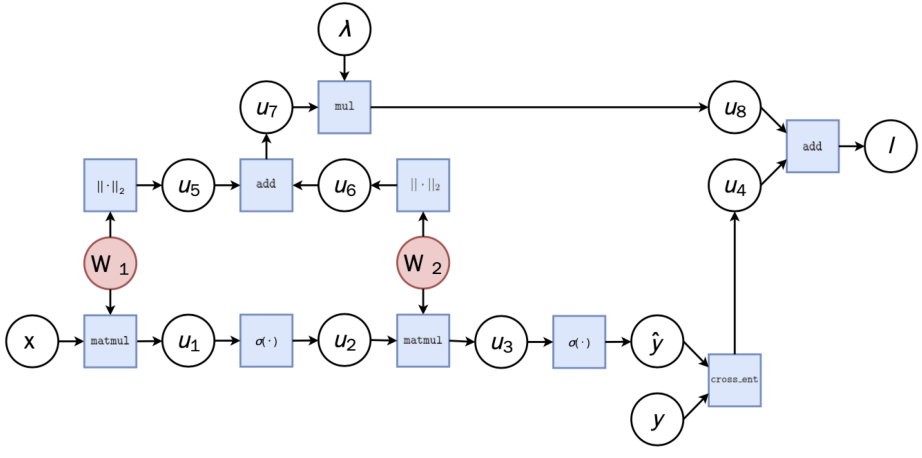
Let us consider a simplified 2-layer MLP and the following loss function:

$$F(\mathbf{W}_1, \mathbf{W}_2) = \ell(y, \hat{y}) + \lambda(\|\mathbf{W}_1\|_2^2 + \|\mathbf{W}_2\|_2^2)$$

where for $x \in \mathbb{R}^d, y \in \mathbb{R}, \mathbf{W}_1 \in \mathbb{R}^{d \times q}$ and $\mathbf{W}_2 \in \mathbb{R}^q$:

$$\hat{y} = f(x; \mathbf{W}_1, \mathbf{W}_2) = \sigma(\mathbf{W}_2^\top \sigma(\mathbf{W}_1^\top x))$$

In the forward pass, many intermediate values are computed and kept in memory from inputs to outputs, which results in the annotated computational graph below:

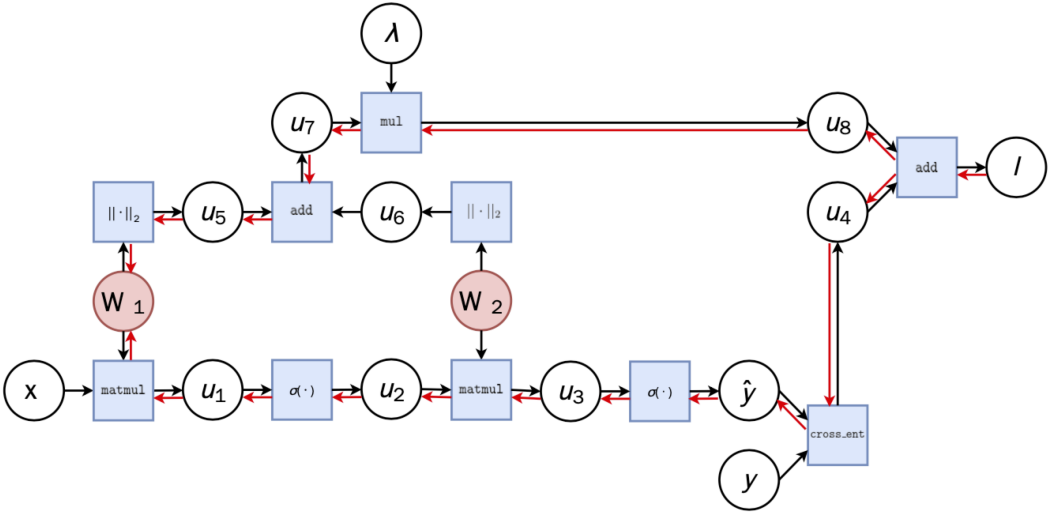


Backpropagation

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 $\frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{W}_1}$ we have:

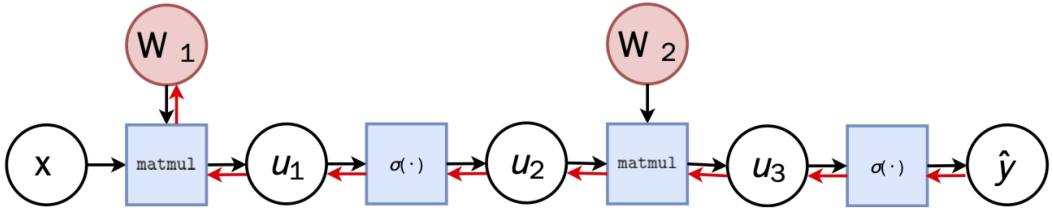
$$\frac{\mathrm{d}\ell}{\mathrm{d}\mathbf{W}_1} = \frac{\partial \ell}{\partial u_8}\frac{\mathrm{d}u_8}{\mathrm{d}\mathbf{W}_1} + \frac{\partial \ell}{\partial u_4}\frac{\mathrm{d}u_4}{\mathrm{d}\mathbf{W}_1}$$

$$\frac{\mathrm{d}u_8}{\mathrm{d}\mathbf{W}_1} = \dots$$



Backpropagation

Let us zoom in on the computation of $\hat{y}$ and 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 x, $\mathbf{W}_1$ and $\mathbf{W}_2$.
- Backward pass: by the chain rule we have

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

- Note how evaluating partial derivatives requires the intermediate values computed during the forward pass

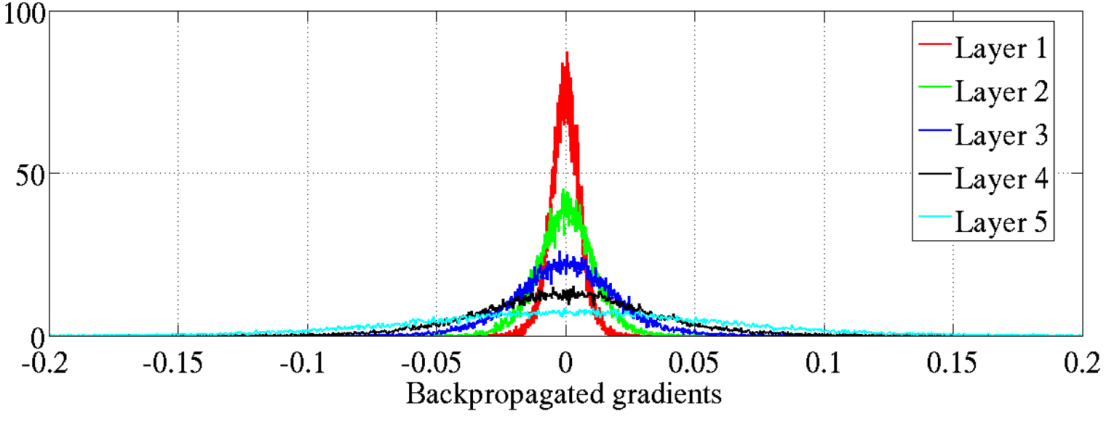
Backpropagation

- This algorithm is called automatic differentiation or backpropagation
- Often mixed up with the optimization algorithm: backpropagation does not optimize, it just helps to compute gradients
- It's the core of any deep learning library: a grammar of differentiable blocks that can be combined together
- 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 kills "learning" with stochastic gradient descent.
- Mitigated the use of deep architectures (many layer stacked)



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

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

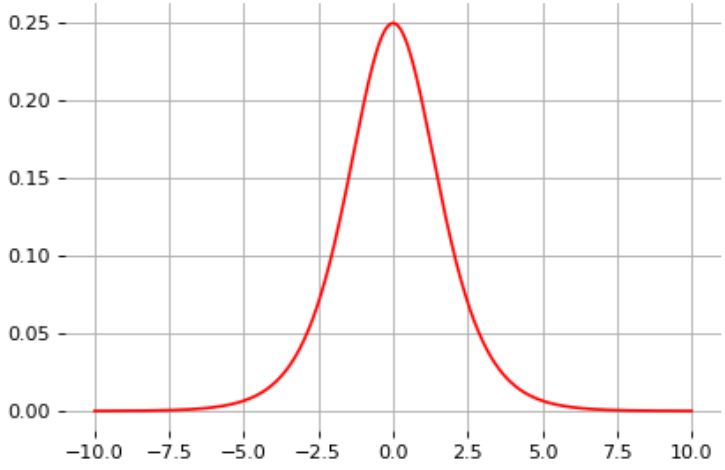
Under the hood, this would be evaluated as

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

and its derivative $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ as

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

The derivative of the sigmoid activation function σ is:



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

Notice that $0 \leq \frac{\mathrm{d}\sigma}{\mathrm{d}x}(x) \leq \frac{1}{4}$ 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,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{\leq \frac{1}{4}} \underbrace{w_3}_{\leq 1} \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{\leq \frac{1}{4}} \underbrace{w_2}_{\leq 1} \underbrace{\frac{\sigma(u_1)}{\partial u_1}}_{\leq \frac{1}{4}} x$$

This implies that the gradient $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ exponentially shrinks to zero as the number of layers in the network increases.

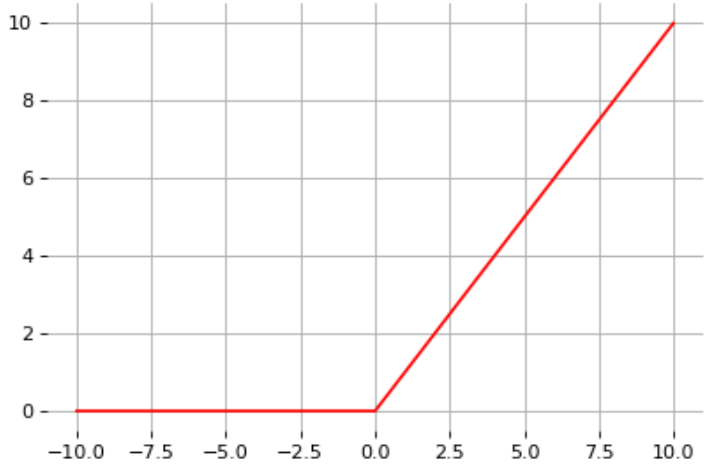
Hence the vanishing gradient problem.

- 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

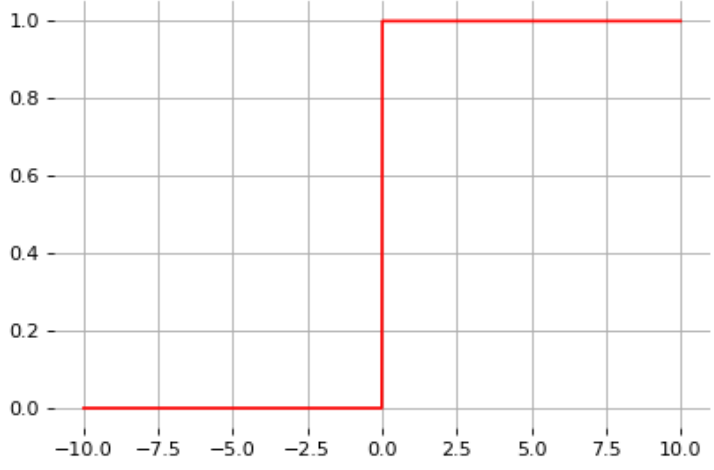
Instead of the sigmoid activation function, modern neural networks are for most based on rectified linear units (ReLU) (Glorot et al, 2011):

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



Note that the derivative of the ReLU function is

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



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

Therefore,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial\sigma(u_5)}{\partial u_5}}_{=1} w_3 \underbrace{\frac{\partial\sigma(u_3)}{\partial u_3}}_{=1} w_2 \underbrace{\frac{\partial\sigma(u_1)}{\partial u_1}}_{=1} x$$

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

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

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

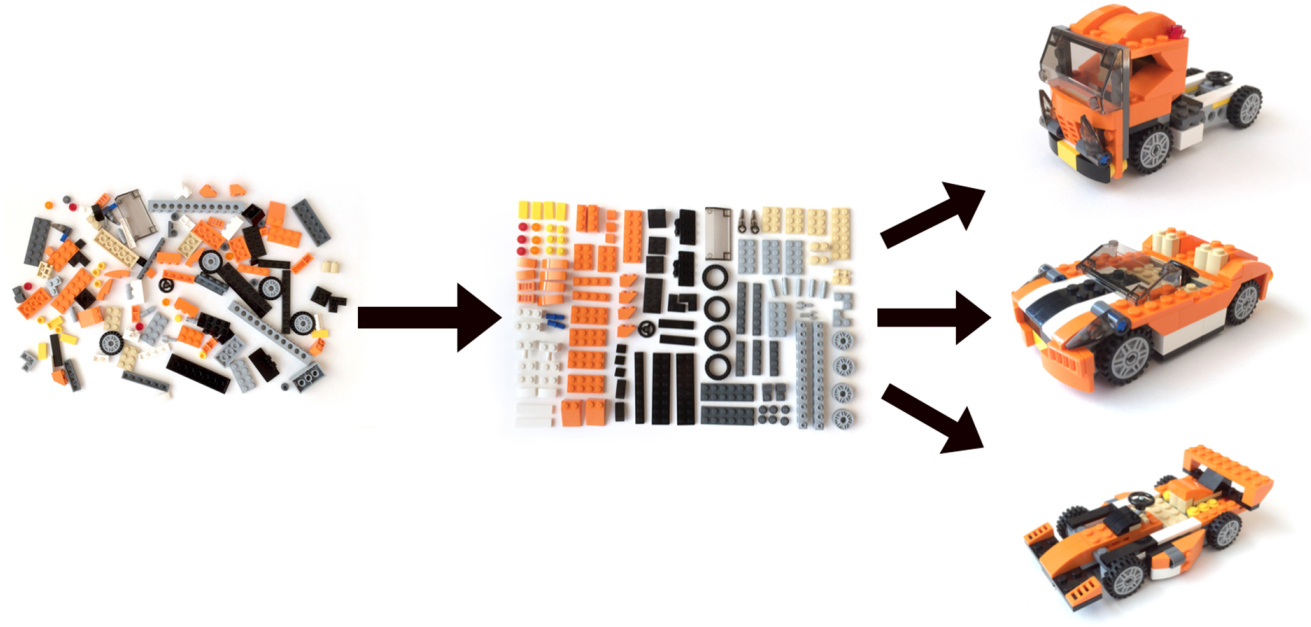
Modern deep learning is like LEGO®

LEGO® Deep Learning

*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**.*

Yann LeCun, 2018.

DL as an architectural language

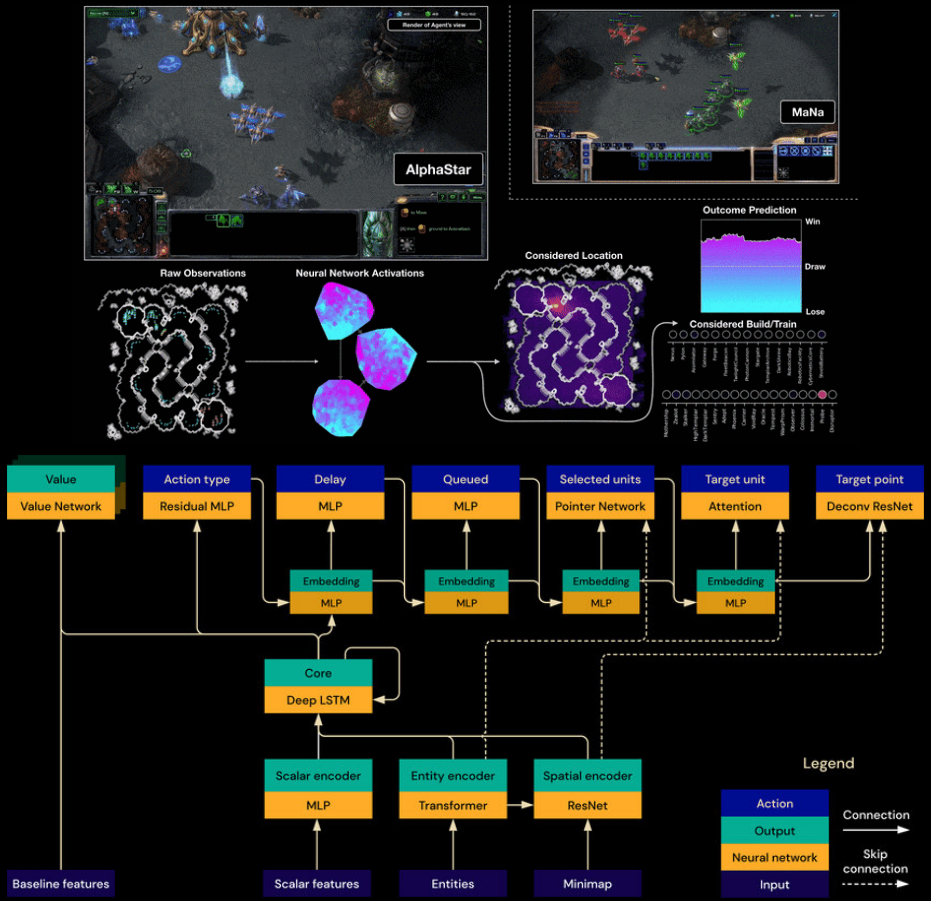


DL as an architectural language

The toolbox

Credits: Oriol Vinyals, 2020.

LEGO® Creator Expert



Credits: Vinyals et al, 2019.

Thank you !