Deep learning

Class 1: Introduction and key concepts

Hédi Hadiji

October 27th 2025

Short Introduction

Hédi Hadiji: *hedi.hadiji@l2s.centralesupelec.fr*

- Assistant Professor at L2S of CS since 2022

Research area: Mathematician, work in bandits, online optimization and learning theory. At the intersection of Machine Learning, Statistics and Optimization.

Goal of the Class

Give a theoretical and practical introduction to deep learning, standard architectures and practices.

Today: Introduce the setting and vocabulary. Give some definitions.

Class information

Edunao will be updated

Project information to come

Table of Contents

Supervised learning

Principle of Supervised Learning

Learning by examples:
predict the response Y to some input X, based on examples.

Examples of Supervised Learning tasks

- X temperature + cloud positions today; Y temperature tomorrow
- X first words in an English sentence; Y next word
- X image; Y whether it contains a cat or not
- X sequence of amino acids; Y protein shape

Supervised Learning: Formal Framework

A learner has access to a sample S of n data points $(X_i, Y_i)_{1 \leqslant i \leqslant n}$.

- $X_i \in \mathcal{X}$ are called *features*
- $Y_i \in \mathcal{Y}$ are called *responses* (or labels when $\mathcal{Y}$ is finite)

The data are i.i.d. from an unknown distribution $(X_i, Y_i) \sim \mathcal{D}$ over $\mathcal{X} \times \mathcal{Y}$.

The goal of the learner is to output a prediction $\widehat{Y}$ of the response Y for new features X not in the sample.

The quality of prediction is measured by a *loss function* $\ell(\cdot, \cdot) : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$.

Objective of Supervised Learning

Given a sample S, find a hypothesis $h_S : \mathcal{X} \to \mathcal{Y}$ such that the risk

$$R(h_S) = \mathbb{E}_{(X,Y)\sim\mathcal{D}}\left[\ell(h_S(X), Y)\right]$$

is small with high probability.

(The risk is a random variable because h_S depends on the sample.)

Empirical Risk Minimization

Perhaps the most natural idea in supervised learning is to look for a hypothesis that minimizes the **empirical risk**

$$h \in \arg\min_{?} \frac{1}{n} \sum_{i=1}^{n} \ell(h(X_i), Y_i)\,.$$

This raises questions:

0 How should one choose the hypothesis space $\mathcal{H}$ over which to perform the minimization?

1 **Optimisation:** how would we compute this minimiser?

2 **Generalization:** why would this be good?

Terminology: we denote the empirical risk a.k.a the train loss as

$$\mathcal{L}(h) = \frac{1}{n} \sum_{i=1}^{n} \ell(h(X_i), Y_i)$$

Supervised learning: Least-squares linear regression

Given a **feature map** $\Phi : \mathcal{X} \to \mathbb{R}^p$, do a linear prediction

- Linear model:
$$f_{w,b}(x) = w^\top \Phi(x) + b$$
- Empirical (squared) loss:
$$\mathcal{L}(w) = \frac{1}{n} \sum_{i=1}^{n} \left(f_{w,b}(x_i) - y_i\right)^2$$
- Ordinary least squares (if $X^\top X$ invertible):
$$(w^\star, b^\star) = (X^\top X)^{-1} X^\top y$$

  where X is the matrix obtained by concatenating the feature vectors.

Supervised learning: Logistic regression

- Binary probabilistic model : $y \in \{0,1\}$

$$p(y=1 \mid x) = \sigma\left(w^\top \Phi(x) + b\right), \qquad \sigma(z) = \frac{1}{1+e^{-z}}.$$

  Decision rule: $\widehat{y} = \mathbf{1}\{w^\top \Phi(x) + b > 0\} = \arg\max_y p(y \mid x)$.
- Loss = negative log-likelihood (binary cross-entropy):

$$\mathcal{L}(w,b) = -\frac{1}{n}\sum_{i=1}^{n}\Big[y_i \log p_i + (1-y_i)\log(1-p_i)\Big], \quad p_i = p(y=1 \mid x_i).$$

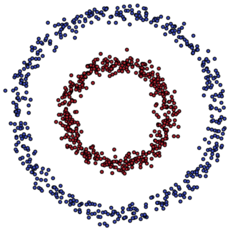
- Multiclass generalization : $y \in \{1,\dots,K\}$:

$$p(y=k \mid x) = \frac{\exp(w_k^\top \Phi(x) + b_k)}{\sum_j \exp(w_j^\top \Phi(x) + b_j)}, \quad \mathcal{L} = -\frac{1}{n}\sum_i \log p(y_i \mid x_i).$$

- No closed-form solution: optimize the loss with (stochastic) gradient descent or second-order methods.

(Generalized) linear models II

- Linear models are linear *in the parameters w*.
- The feature map is fixed.
- Linear models are good when you have a clear idea of what an adapted feature map is. e.g. if your data looks like



then $\Phi((x_1, x_2)) = x_1^2 + x_2^2$ is a good feature map. (See also kernel methods: infinite-dimensional feature maps.)

- Images $d \propto N_{\text{pixels}}$. Designing the correct feature map is hard and very task-dependent.

Table of Contents

Single-Layer Perceptron

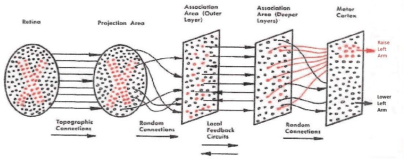


FIG. 1 — Organization of a biological brain. (Red areas indicate active cells, responding to the letter X.)

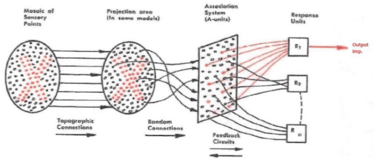
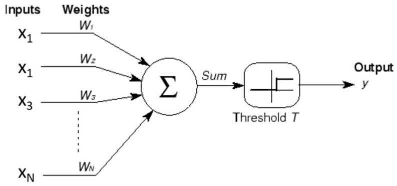
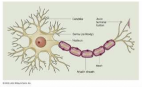


FIG. 2 — Organization of a perceptron.

Invented by Rosenblatt (ca. 1957), before general-purpose computers

Single-Layer Perceptron





Individual neurons receive $\sim$ binary inputs from neighboring neurons, and fire if the **weighted** sum of their inputs is above a certain **threshold**.

The (stylized) brain learns to suppress of amplify activation by modifying those weights, depending on whether its predictions were correct or not.

Multi-Layer Perceptron: Compositional Representation

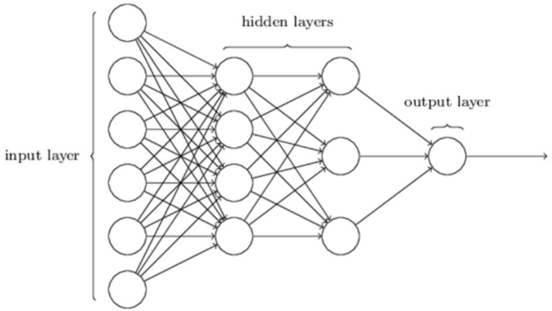
Let $d = h_0, h_1, \ldots, h_L \in \mathbb{N}$ be hidden dimensions.
Compose neurons together: let $W_i \in \mathbb{R}^{h_i \times h_{i-1}}$ and $b_i \in \mathbb{R}^{h_i}$.

$$\begin{aligned} a^0 &= x \\ a^1 &= \sigma^1(W^1 a^0 + b^1) \\ a^2 &= \sigma^2(W^2 a^1 + b^2) \\ &\ldots \\ a^L &= \sigma^{L-1}(W^{L-1} a^{L-1} + b^{L-1}) \\ z &= \sigma^{\text{out}}(a^L) \end{aligned}$$

- $a^i = \sigma^i(W^i x + b^i)$ is a layer (in $\mathbb{R}^h$) of hidden activations
- Output layer maps hidden features to predictions.
- Nonlinearity $\sigma(\cdot)$ allows complex, non-linear functions. Usually a simple element-wise operation.

MLP

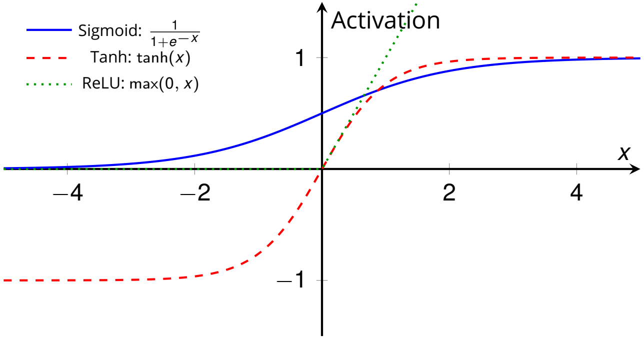
Successive layers of individual perceptrons (neurons)



- Each edge represents a component of the weight matrix
- Biases are added at every node

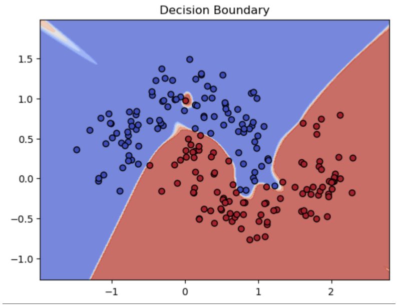
Common Activation Functions

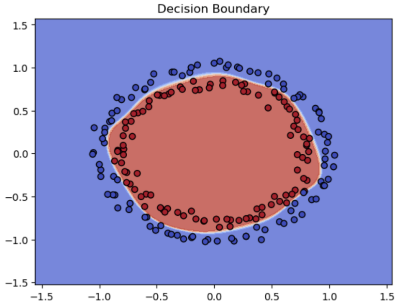
- Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$
- Tanh: $\mathtt{tanh}(x)$
- ReLU: $\mathrm{ReLU}(x) = \max(0, x)$
- Leaky ReLU, GELU, etc.



Observation: Sigmoid and Tanh saturate for large $|x|$, slowing learning. ReLU avoids saturation in the positive region and is widely used in deep networks.

Decision Boundaries





- With appropriately chosen weights MLPs can represent complex nonlinear decision boundaries.
- For any (continuous) function on a bounded domain, there is a wide enough neural network that can approximate it (see Barron’s approximation theorem).
- Hidden units act as learned feature detectors.

Table of Contents

(Stochastic) gradient descent

To find neural network with small loss, we perform *gradient descent*. Let W denote the vector containing all the weights and biases of our network. Using gradient descent consists in performing the updates:

Gradient Descent

Given learning rate (or step-size) $\eta > 0$,

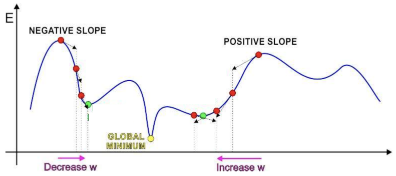
$$W_{t+1} = W_t - \eta \nabla_W \mathcal{L}(W_t)$$

where

$$\mathcal{L}(W_t) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(h(X_i; W_t), y_i)$$

In practice, the full dataset does not fit into memory, and we use instead a smaller *batch* of data, and go through the dataset, making *passes*: at every iteration take b data points, either at random or in a predefined order, and compute the average loss over the batch.

Intuition: Gradient Descent Dynamics



- Learning rate η:
  - Large η: fast progress, but risk of overshooting or divergence.
  - Small η: stable, but slow convergence and may get stuck.
- Landscape:
  - Sharp minima: large gradients, can cause instability.
  - Flat minima: small gradients, slow progress.
  - Smooth loss (e.g. quadratic) is easier for GD than non-smooth (e.g. absolute value).
- Because of non-convexity
  - Can get stuck at saddle points or plateaus.
  - Sensitive to initialization and hyperparameters.

In practice: set $\mathrm{LR} = 0.01$ or look at reference implementations.

Momentum

A practically succesful trick is to add *momentum* to the gradients, i.e.,

$$v_{t+1} = \mu v_t - \eta \nabla_w \mathcal{L}(w_t), \qquad w_{t+1} = w_t + v_{t+1}$$

- $\mu \in [0, 1)$ is the momentum coefficient (typical: 0.9–0.99); η is the learning rate.
- Intuition: v is a velocity that accumulates consistent gradient directions, accelerating progress along shallow directions and damping oscillations across steep ones.

Momentum illustration

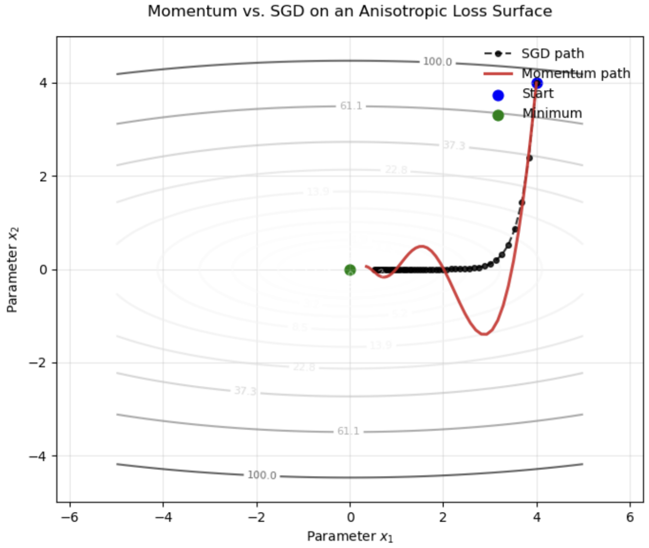


Table of Contents

Backpropagation

Gradient computation

We want to compute the partial derivatives of the final loss w.r.t. respect to all the components of $\mathbf{W} = (W^1, \dots, W^L)$ of a function of the form

$$f(W^1, \dots, W^L) = f^L(W^L, f^{L-1}(W^{L-1}, f^{L-2}(W^{L-2}, \dots, f^2(W^2, f^1(W^1)) \dots)))$$

Then

$$\partial_{W^L} f(\mathbf{W}) = J_{f^L,1}(W^L, a^{L-1})$$

$$\partial_{W^{L-1}} f(W) = J_{f^L,2}(W^L, a^{L-1})\ J_{f^{L-1},1}(W^{L-1}, a^{L-2})$$

$$\dots$$

$$\partial_{W^1} f(W) = J_{f^L,2}(W^L, a^{L-1})\ J_{f^{L-1},2}(W^{L-1}, a^{L-2}) \dots J_{f^1}(W^1)$$

We can compute the gradient w.r.t layer ℓ given

- the gradients at deeper layers,
- the activations at lower layers

Backprop in the MLP

Let us ignore biases here

$$\partial_{W^i} f(W) = J_{f^L,2}(W^L, a^{L-1})\ J_{f^{L-1},2}(W^{L-1}, a^{L-2}) \dots J_{f^i,1}(W^i, a^{i-1})$$

In our case, at the *j*-th layer,

$$f^j(W, a) = \sigma(Wa) = \begin{pmatrix} \sigma((Wa)_1) \\ \dots \\ \sigma((Wa)_{h^j}) \end{pmatrix}$$

so

$$J_{f^j,2}(W, a) = \operatorname{diag}(\sigma'(Wa))\ W,$$

and (remember the first input of a f^j is a matrix so the Jacobian can be seen as tensor of size (h_{j-1}, h_j, h_j)), i.e., mathematically for $\Delta W \in \mathbb{R}^{h_j \times h_{j-1}}$

$$J_{f^j,1}(W, a).\Delta W = \operatorname{diag}(\sigma'(Wa))\ (\Delta W)\, a.$$

Better parameterization

Parameterizing with pre-activations instead:

$$\partial_{W^i} g(W) = J_{g^L,2}(W^L, z^{L-1})\, J_{g^{L-1},2}(W^{L-1}, z^{L-2}) \dots J_{g^i,1}(W^i, z^{i-1})$$

In our case, at the *j*-th layer,

$$g^j(W, z) = W\sigma(z) = \begin{pmatrix} (W\sigma(z))_1 \\ \vdots \\ (W\sigma(z))_{h^j} \end{pmatrix}$$

so

$$J_{g^j,2}(W, z) = W \operatorname{diag}(\sigma'(z)),$$

and (remember the first input of a g^j is a matrix so the Jacobian can be seen as a tensor of size (h_{j-1}, h_j, h_j)), i.e., mathematically for $\Delta W \in \mathbb{R}^{h_j \times h_{j-1}}$

$$J_{g^j,1}(W, z).\Delta W = (\Delta W)\,\sigma(z).$$

Implementing Backprop in an MLP

Let $\odot$ denote element-wise product between arrays of the same size.

Algorithm 1 Backward pass for an L-layer network

```
1: $\delta_L \leftarrow \nabla_{a_L} \mathcal{L}(a_L, y) \odot \sigma_L'(z_L)$
2: **for** $l = L - 1$ **down to** 1 **do**
3:     $\delta_\ell \leftarrow (W_{\ell+1}^\top \delta_{\ell+1}) \odot \sigma_\ell'(z_\ell)$
```

Theorem

For any $\ell \in [1, L]$,

$$\partial_{z_\ell} \mathcal{L}(W) = \delta_\ell \quad \textit{and} \quad \partial_{W^\ell} \mathcal{L}(W) = \delta_\ell a_{\ell-1}^\top$$

Proof.

Computation on last slide + induction. □

Training Procedure

1. Initialize weights $W^{(l)}$, biases $b^{(l)}$.
2. For each batch:
    1. Forward pass: compute predictions.
    2. Compute loss.
    3. Backward pass: compute/propagate gradients.
    4. Update parameters: $W \leftarrow W - \eta \frac{\partial L}{\partial W}$

- Implementable in any directed acyclic graph of computation, beyond MLPs.
- Efficiently implemented in all DL frameworks (autograd).

Key Takeaways

- MLP = composition of affine + nonlinear transformations.
- Backpropagation efficiently computes gradients.
- Choice of activation and depth affects representational power.
- Foundation for CNNs, RNNs, Transformers.

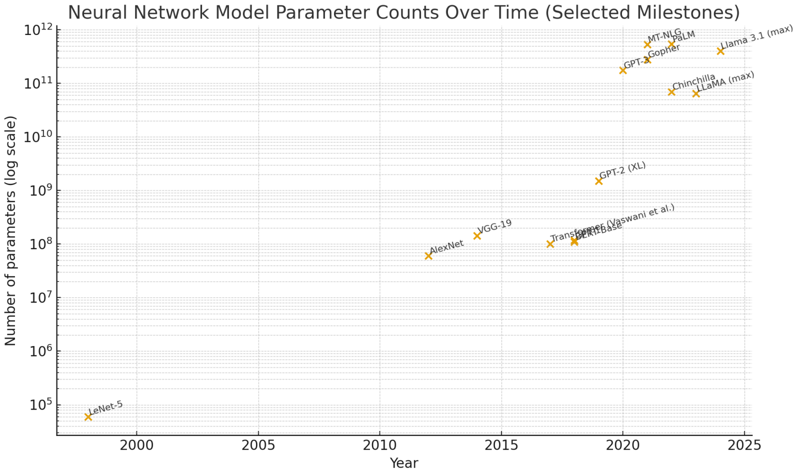
Why deep learning works?

Theoretical mystery, but there are high-level insights

- Compositionality: stacking simple nonlinear modules yields rich function classes able to express complex mappings.
- Learns hierarchical representations: lower layers extract simple features, higher layers compose them into abstract concepts.
- Parallelizability: NNs involve matrix multiplication and element-wise functions which are easy to parallelize on GPU.
- Structured inductive biases (convolutions, attention, recurrence) can incorporate domain knowledge/data structure in a principled manner.
- Scales with data and compute: larger models + more data often yield better performance (pretraining + fine-tuning).

Neural networks have been the main framework organize the massive scaling of data and computational power.

Deep learning



Limits of deep learning

The universal use of deep learning raises issues

- Black boxes: many parameters imply hard to understand and justify the output. Ethical concerns for algorithmic decision-making, safety concerns for robots/self-driving cars
- Massive scaling means massive energy consumption

How to fix or handle these questions poses major scientific and political questions.

Table of Contents

Supervised learning: Some practical considerations

There are no theoretically justified ways to optimally tune hyperparameters (learning rate, number of hidden layers, etc.)
Ideally:

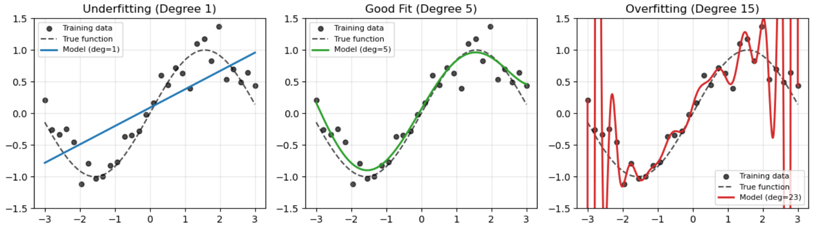
- Split the dataset into three disjoint parts:

$$\mathcal{D} = \mathcal{D}_{\text{train}} \sqcup \mathcal{D}_{\text{val}} \sqcup \mathcal{D}_{\text{test}},$$

- Roles:
    - **Train**: fit model parameters (weights).
    - **Validation**: tune hyperparameters, select models/checkpoints, early stopping.
    - **Test**: final unbiased estimate of generalization performance — used only once after all choices are fixed.
- Practical tips:
    - Typical splits: 60/20/20 or 80/10/10; for small datasets prefer cross-validation.
    - Never peek at the test set during development — this causes leakage and optimistic bias.
    - After hyperparameter selection you may retrain on train+val before the final test evaluation.

Table of Contents

Overfitting vs. Underfitting



Richer model classes have greater expressive power and can fit a wider range of data distributions. If a model is too flexible it can ”memorize” the training set and fail to generalize — this is called overfitting. Conversely, a model that is too simple will underfit.

Example: if a feature map embeds the data into $\mathbb{R}^d$ with $d > n$, then generically there exists a linear model that interpolates the n training points.

Intuitively there is an optimal level of complexity that balances approximation error and estimation error (the bias-variance trade-off). **But deep neural nets are very expressive.**

Regularization

Common ways to prevent overfitting:

- SGD with square loss does implicit regularization by natural going to small norm solutions
- Early stopping: stop training before the training loss becomes excessively small.
- Weight decay (L2 regularization): add $\frac{\lambda}{2}\|w\|_2^2$ to the loss.
- Dropout and other stochastic regularizers.
- Data augmentation: increase effective dataset size by transforming inputs (rotations, crops, noise, etc.).
- Model selection with a validation set: choose architecture and hyperparameters using held-out validation data.

Beyond supervised learning

Supervised learning is the most conceptually clean framework to introduce neural networks, but they are used in all areas of machine learning and modern engineering.
Other variants of learning in which Deep nets have had impact

- Reinforcement Learning: goal-based learning
- Generative: learn to reproduce data that is similar to your dataset
- Transfer learning: learn in a way that can be useful for other types of data

(Notice these problems are harder to formalize than SL).

TP1:

Write your own implementation of backprop.