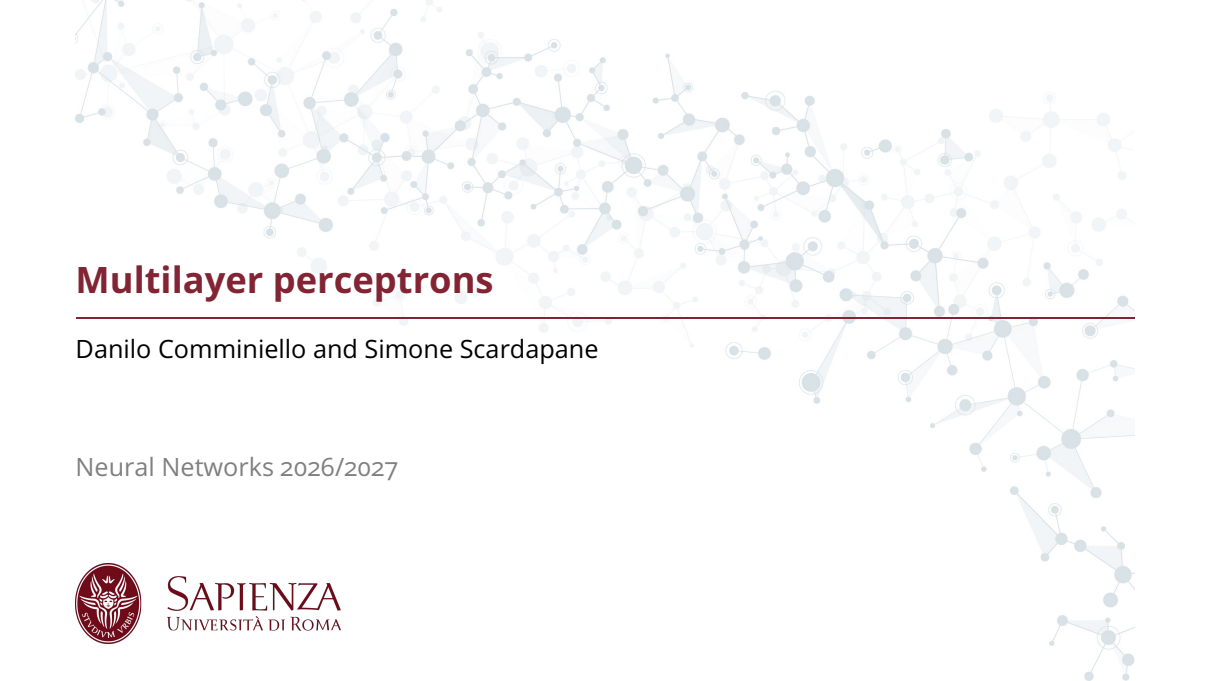




Multilayer perceptrons

Danilo Comminiello and Simone Scardapane

Neural Networks 2026/2027



SAPIENZA
UNIVERSITÀ DI ROMA

1 Multilayer perceptrons

Basic definitions

ReLU and depth

The multilayer perceptron

A **fully-connected neural network**, also known as a **multilayer perceptron (MLP)**, learns an intermediate representation of the input:

Hidden representation

$$\mathbf{h} = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$$

ϕ is an elementwise **activation**.

Linear output

$$f(\mathbf{x}) = \mathbf{w}^\top \mathbf{h} + c$$

A linear model on learned features $\mathbf{h}$.

The parameters $\mathbf{W}$, $\mathbf{b}$, $\mathbf{w}$, c are learned jointly from the data.

Without the nonlinearity ($\phi(z) = z$), the two affine maps collapse:

$$f(\mathbf{x}) = \mathbf{w}^\top (\mathbf{W}\mathbf{x} + \mathbf{b}) + c = (\mathbf{w}^\top \mathbf{W})\mathbf{x} + (\mathbf{w}^\top \mathbf{b} + c).$$

Adding hidden layers

Nothing prevents us from stacking L **hidden** layers before the linear output:



Hidden layers

$$\mathbf{h}^{(\ell)} = \phi_{\ell}(\mathbf{W}^{(\ell)}\mathbf{h}^{(\ell-1)} + \mathbf{b}^{(\ell)})$$

Each layer $\ell = 1, \dots, L$ has its own weights and activation; $\mathbf{h}^{(0)} = \mathbf{x}$.

Linear output

$$f_{\theta}(\mathbf{x}) = \mathbf{w}^{\top} \mathbf{h}^{(L)} + c$$

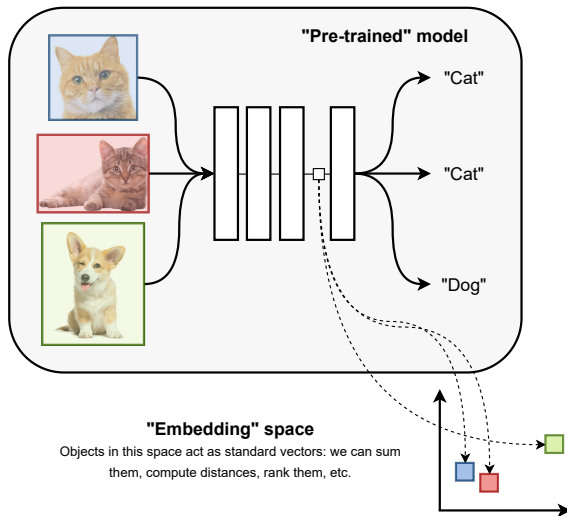
The same linear model as before, applied to the last representation.

With widths $D_0, \dots, D_L$ (D_0 is the input dimension), $\mathbf{W}^{(\ell)} \in \mathbb{R}^{D_{\ell} \times D_{\ell-1}}$ and $\mathbf{b}^{(\ell)} \in \mathbb{R}^{D_{\ell}}$: layer ℓ has $D_{\ell}(D_{\ell-1} + 1)$ parameters, and the output adds $D_L + 1$.

The parameters are $\theta = \{\mathbf{W}^{(\ell)}, \mathbf{b}^{(\ell)}\}_{\ell=1}^L \cup \{\mathbf{w}, c\}$.

Embedding the data

The hidden representation provides new coordinates for the input, in which a linear output layer can distinguish the classes.



Training the network

Given a dataset $\mathcal{S} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$, we minimize:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), y_i).$$

We use the losses introduced in the linear-model lecture, with the appropriate output for regression or classification (i.e., adding a softmax or sigmoid for multi-class or binary classification, respectively). In order to compute gradients we can use **automatic differentiation**.

Universal approximation

Let $g : K \rightarrow \mathbb{R}$ be continuous on a compact set $K \subset \mathbb{R}^d$, and let ϕ be continuous and non-polynomial.

For any $\varepsilon > 0$, there exists an MLP with **one hidden layer**, sufficiently large width, and biases such that:

$$\sup_{\mathbf{x} \in K} |g(\mathbf{x}) - f_{\theta}(\mathbf{x})| < \varepsilon.$$

This is an **existence result**. It does not tell us how much width or data we need, or whether training will find suitable parameters.

A single hidden layer can be sufficient for approximation, but it need not be an efficient choice. We will see shortly how depth can help.

Leshno, Lin, Pinkus and Schocken (1993), Neural Networks, 6(6), 861–867.

Some terminology

Representation / embedding

The feature vector $\mathbf{h}^{(\ell)}$ computed at an intermediate layer.

Width and depth

Width: $\dim \mathbf{h}^{(\ell)}$. Depth: L hidden layers ($L + 1$ with the output).

Parameters: learned

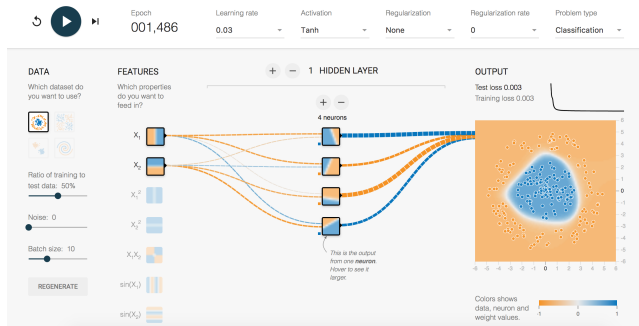
The weights and biases θ learned from the data.

Hyperparameters: chosen

Layer count, widths, activation functions, and learning rate.

Choosing hyperparameters is called **model selection** / **hyperparameter optimization**.

Playing with a neural network



playground.tensorflow.org

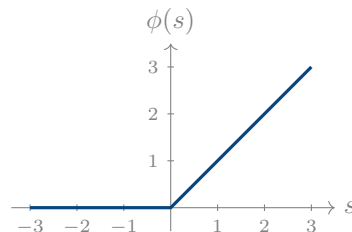
What happens to the learned representations and the decision boundary as we change the number and width of the hidden layers?

The rectified linear unit (ReLU)

The **rectified linear unit** is defined as:

$$\phi(s) = \max(0, s).$$

In words: negative inputs become zero, positive inputs pass through unchanged.



ReLU is a good default choice for MLPs: we will explain more in the automatic differentiation lecture, when we study how gradients propagate through the network.

One network, many affine maps

Each ReLU either passes its input or sets it to zero:

$$\mathbf{h} = \mathbf{D}(\mathbf{x})(\mathbf{W}\mathbf{x} + \mathbf{b}), \quad D_{ii}(\mathbf{x}) \in \{0, 1\}.$$

The diagonal entries of $\mathbf{D}$ indicate which units are active. With an affine output,

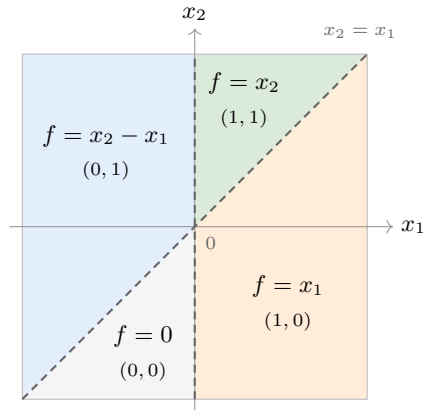
$$\begin{aligned} f(\mathbf{x}) &= \mathbf{w}^\top \mathbf{D}(\mathbf{x})(\mathbf{W}\mathbf{x} + \mathbf{b}) + c \\ &= \mathbf{a}(\mathbf{x})^\top \mathbf{x} + \beta(\mathbf{x}). \end{aligned}$$

Within each region, the gates stay fixed: the network is **exactly affine**.

This holds at any depth. Training learns both the regions and their affine maps.

For ReLU hidden layers, before any output sigmoid or softmax. Montúfar et al. (2014).

$$f(\mathbf{x}) = \text{ReLU}(x_1) + \text{ReLU}(x_2 - x_1)$$

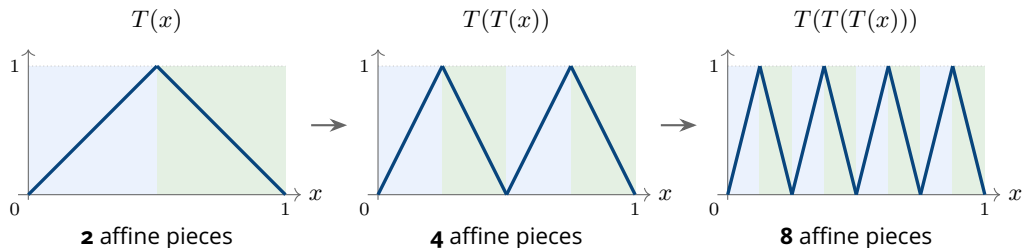


Labels: output formula and gate pattern (d_1, d_2)

Depth through composition

Three ReLUs make a triangular function. We apply the same function repeatedly:

$$T(x) = 2 \operatorname{ReLU}(x) - 4 \operatorname{ReLU}\left(x - \frac{1}{2}\right) + 2 \operatorname{ReLU}(x - 1), \quad x \in [0, 1].$$



Each composition splits every piece in two: k blocks give 2^k pieces using $3k$ ReLU units.

A single hidden layer with m ReLUs has at most $m + 1$ pieces in one dimension. **Depth can make a representation much more compact.**

An expressivity example, not a guarantee about training or generalization. Telgarsky (2016).

2 Training MLPs

Training in practice

Training can be weird

Initializing the weights

If all the weights of a layer start equal, all its hidden units stay identical during training. We break this **symmetry** by sampling the weights at random, $W_{ij} \sim \mathcal{N}(0, \sigma^2)$.

How large should σ^2 be? We follow the mean square of the activations, $m_\ell = \mathbb{E}[(h_i^{(\ell)})^2]$, through one ReLU layer with zero biases:

- The preactivation $s_i = \sum_j W_{ij} h_j$ is a sum of $D_{\ell-1}$ independent zero-mean terms, so their variances add up: $\mathbb{E}[s_i^2] = \sigma^2 D_{\ell-1} m_{\ell-1}$.
- s_i is symmetric around zero, and ReLU sets its negative half to zero: $m_\ell = \frac{1}{2} \mathbb{E}[s_i^2]$.

$$m_\ell = \frac{\sigma^2 D_{\ell-1}}{2} m_{\ell-1} \quad \implies \quad \text{the scale is preserved when } \sigma^2 = \frac{2}{D_{\ell-1}}.$$

He initialization

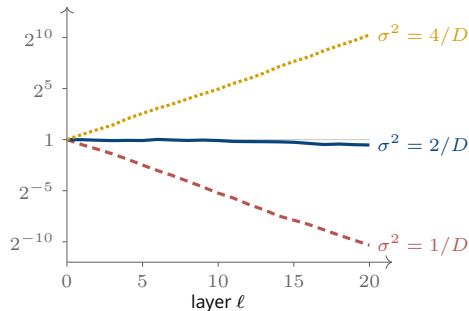
With any other σ^2 , every layer multiplies m_ℓ by the same factor, so the error compounds with depth: activations **vanish** for small weights and **explode** for large ones.

He initialization

$$W_{ij}^{(\ell)} \sim \mathcal{N}\left(0, \frac{2}{D_{\ell-1}}\right), \quad \mathbf{b}^{(\ell)} = \mathbf{0}.$$

Root mean square of $\mathbf{h}^{(\ell)}$

ReLU MLP, width $D_\ell = 512$



The same reasoning applies to gradients flowing backward, as we will see in the autodiff lecture. For other activations, see Glorot and Bengio (2010); He et al. (2015).

Weight decay

In ridge regression we penalized large weights. We can add the same penalty to the loss of any network:

$$\mathcal{L}_\lambda(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \frac{\lambda}{2} \|\boldsymbol{\theta}\|^2 \quad (\text{sum of all squared parameters}).$$

The penalty adds $\lambda\boldsymbol{\theta}$ to the gradient, so one gradient descent step becomes:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta (\nabla \mathcal{L}(\boldsymbol{\theta}) + \lambda\boldsymbol{\theta}) = (1 - \eta\lambda) \boldsymbol{\theta} - \eta \nabla \mathcal{L}(\boldsymbol{\theta}).$$

Each step first shrinks all parameters by the same factor, hence the name **weight decay**. A parameter stays large only if the loss keeps pushing it there.

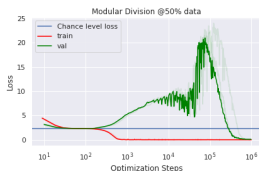
In PyTorch, it is an optimizer argument: `SGD(..., weight_decay=1e-4)`.

With Adam, the penalty and the decay differ; AdamW applies the decay directly. Loshchilov and Hutter (2019).

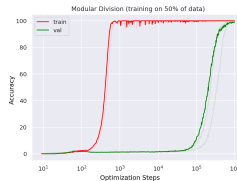
Training neural networks can be weird

Train a model (transformer, later in the course) to learn division modulo 97. What happens?

Loss over training



Accuracy in the same experiment



Definitely not intuitive: multiple descents for the validation loss, with accuracy improving long after perfect memorization (**grokking**).

Other surprising phenomena we have no time to cover: fitting random labels, double descent as model size grows, and low-loss paths connecting different solutions.

Original plots: Power et al. (2022), Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets, Figs. 1 and 4.

3 Designing the block

Gating

Sharing a block

ReLU as a gate

ReLU multiplies its input by a **gate** that depends on the input itself:

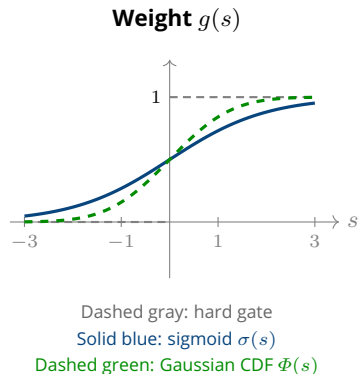
$$\text{ReLU}(s) = s g(s), \quad g(s) = \begin{cases} 0 & s < 0, \\ 1 & s \geq 0. \end{cases}$$

Changing the gate gives other activations:

- **LeakyReLU**: $g(s) = \alpha$ for $s < 0$ (e.g., $\alpha = 0.01$).
- **PReLU**, same as LeakyReLU but α is learned (per-unit).
- **Smooth gates**: any smooth non-decreasing $0 < g(s) < 1$, as shown on the right.

Kolmogorov–Arnold networks (KANs) go further: they learn a whole scalar function on every connection, at a higher cost.

He et al. (2015); Liu et al. (2024), KAN.



Smooth gates: SiLU and GELU

SiLU (Sigmoid-weighted linear unit):

$$\text{SiLU}(s) = s \sigma(s), \quad \sigma(s) = \frac{1}{1 + e^{-s}}.$$

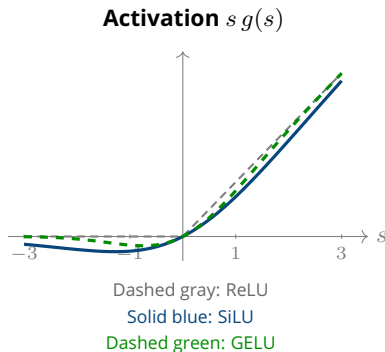
GELU (Gaussian error linear unit):

$$\text{GELU}(s) = s \Phi(s).$$

where Φ is the standard Gaussian CDF.

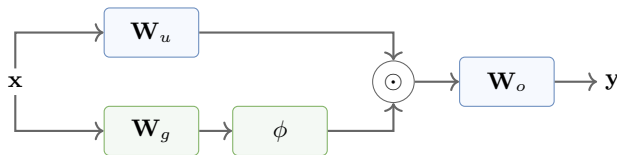
GELU is common in transformers such as BERT and GPT-2; SiLU appears inside SwiGLU.

Elfwing et al. (2017); Hendrycks and Gimpel (2016).



Gating with a second projection: GLU and SwiGLU

In ReLU and SiLU, each unit gates its own preactivation. In a **gated linear unit** (GLU), the gate comes from a second projection of the input:



$$\mathbf{y} = \mathbf{W}_o [(\mathbf{W}_u \mathbf{x}) \odot \phi(\mathbf{W}_g \mathbf{x})] .$$

With $\phi = \sigma$, the gate lies in $[0, 1]$, as in the original GLU. Replacing it with SiLU gives **SwiGLU**, used in Llama and many recent language models: its multiplier can also be negative or exceed one.

For $\mathbf{x} \in \mathbb{R}^D$, $\mathbf{W}_u, \mathbf{W}_g \in \mathbb{R}^{M \times D}$ and $\mathbf{W}_o \in \mathbb{R}^{D \times M}$. Biases are omitted.

Shazeer (2020); Touvron et al., LLaMA (2023).

Mixtures of experts

A gate can also choose between whole blocks. A **mixture of experts** (MoE) combines E **experts** $f_1, \dots, f_E$ with the same output shape, weighted by a **router**:

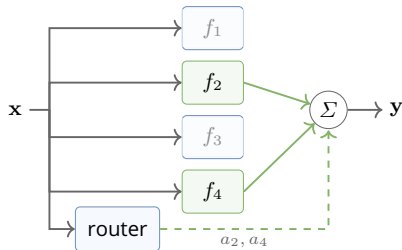
$$\mathbf{y} = \sum_{e=1}^E a_e(\mathbf{x}) f_e(\mathbf{x}), \quad \mathbf{a}(\mathbf{x}) = \text{softmax}(\mathbf{W}_r \mathbf{x}).$$

Dense MoE

All E experts run; the router reweights them.

Sparse MoE

Only the k top-scoring experts run: computation grows with k , not E .



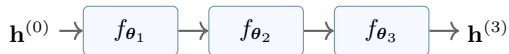
Sparse MoEs are common in recent language models. Training must keep the experts balanced, or the router may send most inputs to a few of them.

Shazeer et al. (2017); Fedus et al. (2022); Jiang et al. (2024).

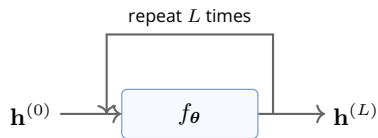
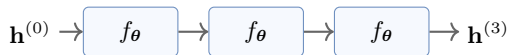
Sharing parameters and looping a block

We can use different parameters at each layer, or apply the same block repeatedly. The latter gives a **looped model**.

Separate parameters



Shared parameters



The shared block computes $\mathbf{h}^{(\ell)} = f_{\theta}(\mathbf{h}^{(\ell-1)})$, with matching input and output shapes. A block with P parameters needs P instead of LP , but still L evaluations.

We will meet looped blocks again with recurrent networks.

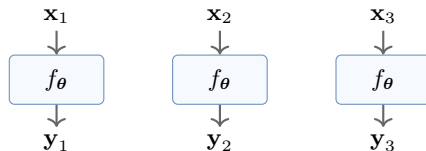
Dehghani et al., Universal Transformers (2019).

Shared MLPs on structured data

We can also share an MLP across the positions of a sequence of vectors $\mathbf{x}_1, \dots, \mathbf{x}_T \in \mathbb{R}^D$, applying it to each position independently:

$$\mathbf{y}_t = f_{\theta}(\mathbf{x}_t), \quad t = 1, \dots, T.$$

The number of parameters does not depend on the sequence length, and the same holds for the pixels of an image. Transformers use these **positionwise MLPs** in every block.



A positionwise MLP mixes the features within a position, but no information moves between positions.

To exchange information, we need another operation: convolutions over neighborhoods (the next lecture), attention, or a recurrent state.