Neural Networks for Data Science Applications

Master's Degree in Data Science

Lecture 4: Fully-connected neural networks

Lecturer: S. Scardapane

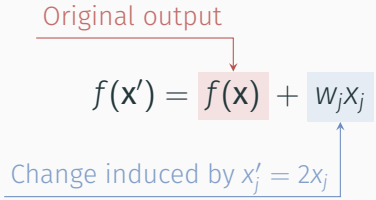
SAPIENZA
UNIVERSITÀ DI ROMA

Feedforward neural networks

Limitations of linear models

To understand the limitations of a linear model $f(\mathbf{x})$, consider an input $\widehat{\mathbf{x}}$ equal to $\mathbf{x}$ but for a single feature $\widehat{x}_i = 2x_i$ (e.g., a client with double income).

The output on the two inputs is related by:



Effects are linearly super-imposed: it is impossible to model some form of interaction between two features (e.g., 'a client with 10k income is not trustworthy, *unless* he is very young').

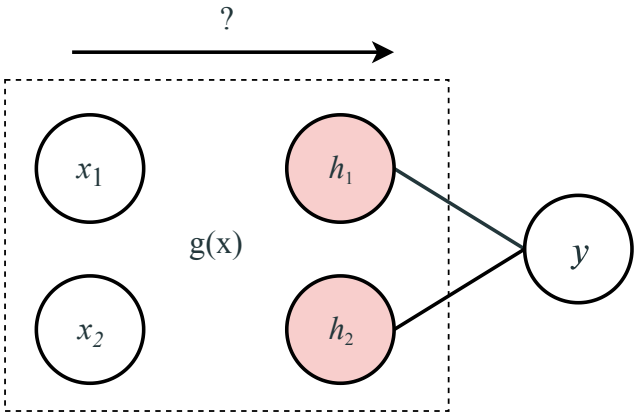


Figure 1: How can we define intermediate computations?

Feedforward neural networks

Fully-connected layers

Can we layer two linear models?

$$\mathbf{h} = \mathbf{W}\mathbf{x}\,, \tag{1}$$
$$f(\mathbf{h}) = \mathbf{w}^\top \mathbf{h}\,. \tag{2}$$

We could, but this is equivalent to a single linear model:

$$f(\mathbf{x}) = \left(\mathbf{w}^\top \mathbf{W}\right)\mathbf{x}\,. \tag{3}$$

We need some way of separating the two linear operations.

We can avoid the 'collapse' of the two linear models by interleaving them with some element-wise nonlinearity ϕ:

$$\mathbf{h} = \phi(\mathbf{W}\mathbf{x}), \tag{4}$$

$$f(\mathbf{h}) = \mathbf{w}^\top \mathbf{h}. \tag{5}$$

This is the prototype of a **fully-connected** (**FC**) neural network, sometimes known as a **multilayer perceptron** (**MLP**).

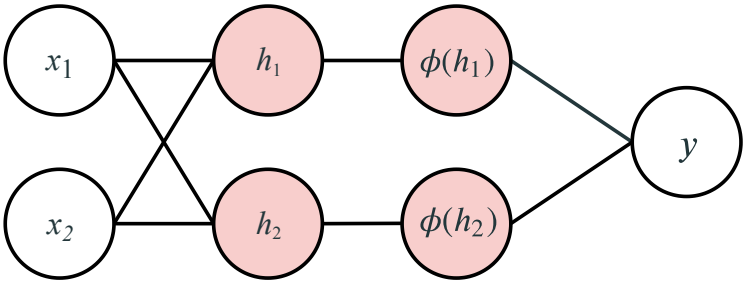


Figure 2: Visualization of a simple feedforward NN. In general, we will never show the nonlinearities explicitly in the graphs from now on.

Feedforward neural networks

Training and complexity considerations

Training of the network can be proceed similarly to a linear model. For example, given a dataset $\{\mathbf{x}_i, y_i\}_{i=1}^n$ for regression, we can minimize the LS loss (note: always remember we are ignoring the biases in the notation):

$$\mathbf{W}^*, \mathbf{w}^* = \arg\min \frac{1}{n} \sum_{i=1}^{n} \left(y_i - f(\mathbf{x}_i)\right)^2. \tag{6}$$

For classification, we can wrap the output in a sigmoid and minimize a cross-entropy loss.

Theorem 1.1: Cybenko (1989) - Hornik (1991) - Leshno (1993)

Let $h(\mathbf{x})$ be a continuous function defined on a compact subset $S \subset \mathbb{R}^d$ and $\varepsilon > 0$. For a sufficiently large p, there exists an $f(\mathbf{x})$ as in Eq. (6)-(7) with p hidden units such that:

$$|h(\mathbf{x}) - f(\mathbf{x})| < \varepsilon, \ \forall \mathbf{x} \in S\,. \tag{7}$$

This holds for any non-constant, bounded, continuous ϕ.

This is a **universal approximation** theorem. It does not tell us about the *feasibility* for a given problem (e.g., how large should p be?).

Because the NN can be highly non-convex, its optimization problem has multiple local minimum and/or saddle points.

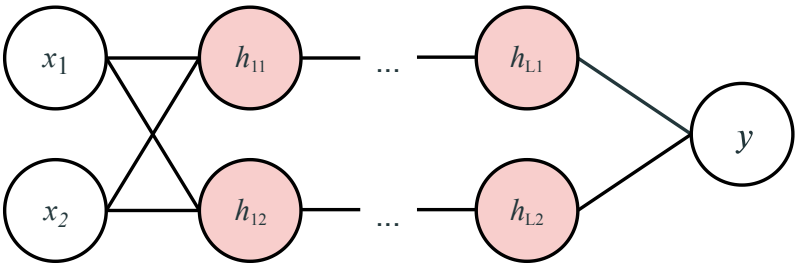
In fact, training a neural network *optimally* is **NP-hard**, even for very simple architectures, and highly dependent on a good initialization.

Finding the global optimum requires running GD from almost *everywhere*: this is similar to an exhaustive search.

Blum, A. and Rivest, R.L., 1989. Training a 3-node neural network is NP-complete. In Advances in neural information processing systems (pp. 494-501).

Nothing prevents us from adding *additional* 'hidden' (intermediate) layers:

$$f(\mathbf{x}) = \mathbf{w}^T \cdot \overbrace{\phi\big(\mathbf{Z} \cdot \underbrace{\phi\,(\mathbf{W} \cdot \mathbf{x})}_{\mathbf{h}^1} + \mathbf{c}\big)}^{\mathbf{h}^2} \tag{8}$$



Differently from a linear model, a NN has several design choices that we have freedom on:

- The nonlinearity (sometimes called the **activation function**);
- The dimensionality of the hidden layers;
- Several others that we will introduce in the next lectures.

We call these **hyper-parameters** to differentiate them from parameters (weights) to be trained via GD.

Choosing the correct set of hyper-parameters is called the **model selection** / **hyper-parameter optimization** problem.

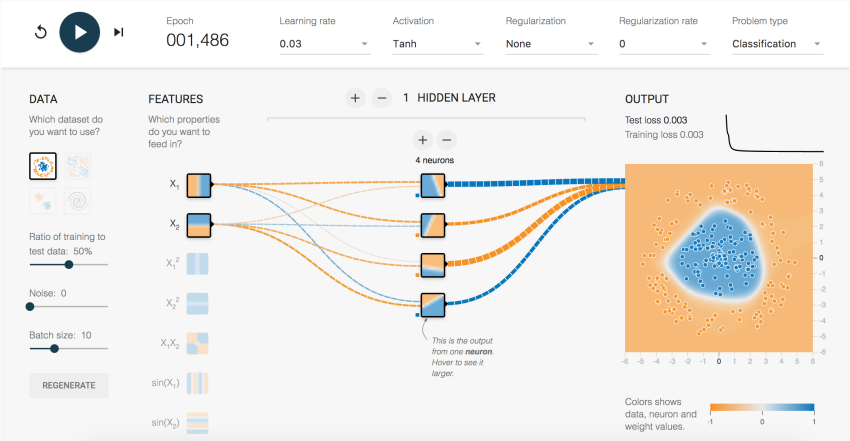


Figure 3: https://playground.tensorflow.org/.

Apart from the sigmoid, what kind of non-linearities can we use?

1. Hyperbolic tangent (**tanh**) $\tanh(s) = 2\sigma(s) - 1$, which is just a sigmoid scaled in $[-1, +1]$.
2. Polynomials $\phi(s) = s^p$ are a very bad idea unless a lot of care is taken due to numerical instabilities.
3. We will see that the **rectified linear unit** (ReLU) $\text{ReLU}(s) = \max(0, s)$ is a good default choice. Note that, if we consider the bias of the previous layer part of the activation function, $\text{ReLU}(s) = \max(0, s + b)$ is a threshold function with a trainable threshold.

We call the size of the hidden layers the **width** of the network, and the number of layers its **depth**. *Narrow* networks with potentially infinite layers are also universal approximators.[1]

UA theorems are not constructive; there exist families of functions that are efficiently learnable with deep networks as opposed to shallow networks, but it is unclear whether this also holds in practice.[2]

[1] Kidger, P. and Lyons, T., 2020. **Universal approximation with deep narrow networks**. In COLT (pp. 2306-2327). PMLR.

[2] Nye, M. and Saxe, A., 2018. **Are efficient deep representations learnable?**. arXiv preprint arXiv:1807.06399.

The distribution of NNs with infinite width converges to a mathematical object known as a **Gaussian Process** (GP).

In fact, under a first-order linearization, the predictions of any NN $f(\mathbf{x})$ can be described by:

$$(\mathbf{y}_t - \mathbf{y}) = \left(\mathbf{I} - \eta \underbrace{[\partial f(\mathbf{x})]^\top \partial f(\mathbf{x})}_{\mathbf{K} \in \mathbb{R}^{n \times n}} \right) (\mathbf{y}_{t-1} - \mathbf{y}) \ . \tag{9}$$

The elements $[\mathbf{K}]_{i,j}$ are called the **neural tangent kernel** (NTK), a fundamental recent tool in the analysis of deep neural networks.

Jacot, A., Gabriel, F. and Hongler, C., 2018. **Neural tangent kernel: Convergence and generalization in neural networks**. NeurIPS, 31.

Training of the networks

Stochastic optimization

Consider the steps needed for a single iteration of GD:

1. Computing the output of the NN $f(\mathbf{x})$ on *all* examples;
2. Computing the gradient of the cost function with respect to the weights.

Both these operations scale *linearly* in the number of examples, which is unfeasible when we have 10^5 or even more examples.

In practice, to train NNs we use **stochastic** versions of GD, that approximate the real gradient from smaller amounts of data.

The key observation is that the training problem in NNs is an expectation with respect to all data ($\boldsymbol{\theta}$ is the set of weights):

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} \left(y_i - f(\mathbf{x}_i) \right)^2 . \tag{10}$$

To limit the complexity, we can use only a **mini-batch** $\mathcal{B}$ of M examples from the full dataset:

$$J(\boldsymbol{\theta}) \approx \widetilde{J}(\boldsymbol{\theta}) = \frac{1}{M} \sum_{i \in \mathcal{B}} \left(y_i - f(\mathbf{x}_i) \right)^2 . \tag{11}$$

We can use the gradient from the approximated function to perform an iteration of GD.

The previous algorithm is called **stochastic gradient descent** (SGD).

The computational complexity of an iteration of SGD is fixed with respect to M (the **batch size**) and does not depend on the size of the dataset.

Because we assumed that samples are i.i.d., we can prove SGD also converges to a stationary point in average, albeit *with noisy steps*.

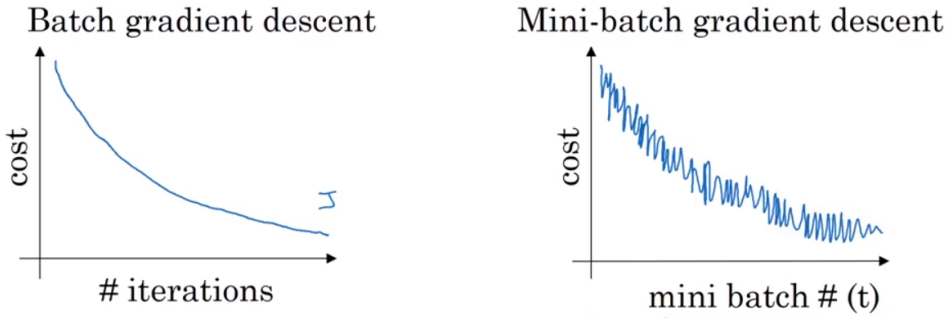


Figure 4: SGD will converge on average to a stationary point, although in an apparently noisy fashion (Source: EngMRK).

Instead of randomly sampling batches from the training dataset at each iteration, we generally apply the following procedure:

1. Shuffle the full dataset;
2. Split the dataset into blocks of M elements and process them sequentially, batch-by-batch;
3. After the last block, return to point (1) and iterate.

A full pass over the dataset is called an **epoch**. This is efficient because the majority of the time we deal with data stored *contigously* in memory. In practice, even the shuffling operation can be approximated.

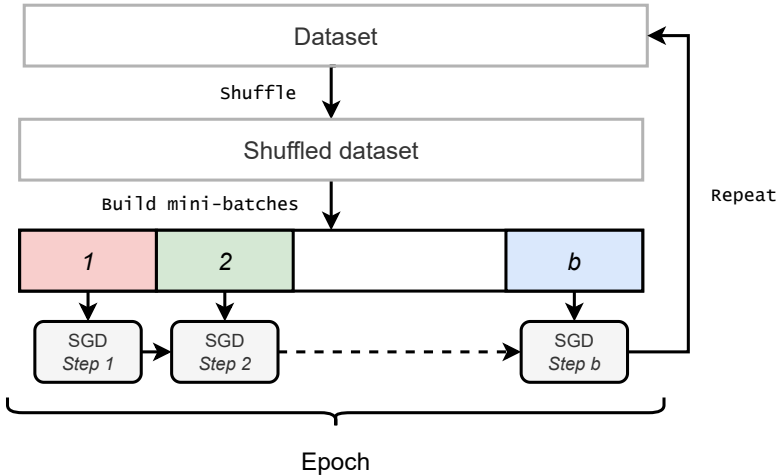


Figure 5: Dividing the optimization process into epochs is also helpful to define metrics and stopping criteria.

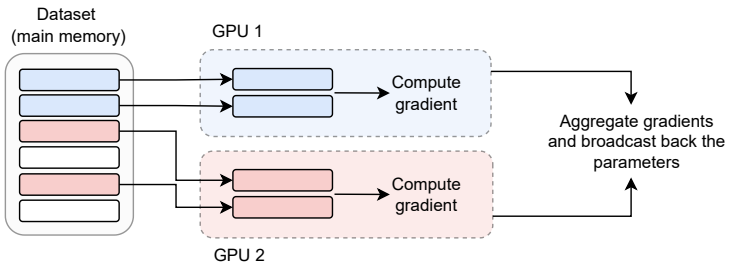


Figure 6: A simple form of distributed stochastic optimization: we process one mini-batch per available machine or GPU (by replicating the weights on each of them) and sum or average the corresponding gradients before broadcasting back the result (which is valid due to the linearity of the gradient operation). This requires a synchronization mechanism across the machines or the GPUs.

Running SGD requires selecting the size of the mini-batch, which is an additional hyper-parameter:

- Smaller mini-batches are faster, but the network might require more iterations to converge because the gradient is noisier.
- Larger mini-batches provide a more reliable estimation of the gradient, but are slower.

In general, it is typical to choose power-of-two sizes (32, 64, 128, ...) depending on the hardware configuration and the total memory available.

In the limit $M = 1$ we obtain an **online** (streaming) optimization.

Other topics

Design variants

1. **LeakyReLU** replaces the negative quadrant of ReLU with a small slope (e.g., $\alpha = 0.01$):

$$\text{LeakyReLU}(s) = \begin{cases} s & \text{if } s > 0 \\ \alpha s & \text{otherwise} \end{cases} \tag{12}$$

2. If you want a smoother version of the ReLU, you can use the **exponential linear unit** (ELU):

$$\text{ELU}(s) = \begin{cases} s & \text{if } s > 0 \\ \alpha(\exp s - 1) & \text{otherwise} \end{cases} \tag{13}$$

Viewing the ReLU as a kind of *gating* function $\mathrm{ReLU}(s) = s1_{x \geq 0}$, we can obtain smoother variants as:

$$\phi(s) = s\psi(s)\,, \tag{14}$$

where:

- If $\psi(s)$ is the standard Gaussian cumulative distribution function, we obtain the **Gaussian Exponential Linear Unit** (GELU).
- If $\psi(s) = \sigma(s)$ we obtain the **Sigmoid linear unit** (SiLU) or **Swish**.

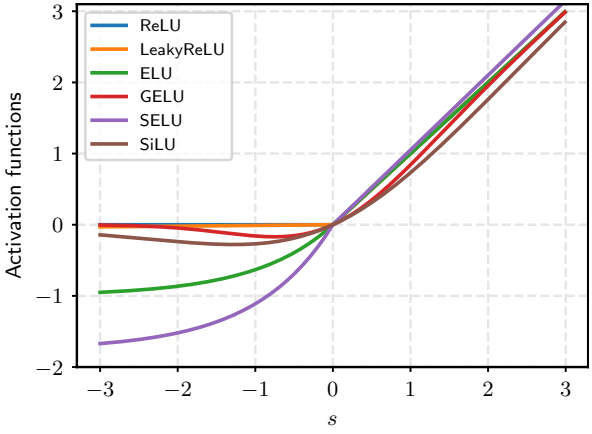


Figure 7: Common activation functions plotted side-by-side.

Activation functions can also have trainable parameters that vary for each unit. For example, the LeakyReLU with trainable α becomes the **parametric ReLU** (PReLU). Or, for a more convoluted example:

$$\text{Trainable-Swish}(s) = \sigma(as + b)(cs + d)\,, \tag{15}$$

with 4 trainable parameters per unit. Variants of this have become popular with the recent LLaMA class of models.

Shazeer, N., 2020. **GLU variants improve transformer**. *arXiv preprint arXiv:2002.05202*.

We can parametrize a generic trainable AF as a small (one hidden layer) MLP:

$$\phi(s) = \sum_i \alpha_i \psi_i(s) \tag{16}$$

where $\psi_1, \ldots, \psi_n$ are (typically fixed) **basis functions**. We can use splines, kernels, radial-basis functions, etc. to obtain multiple variants. However, it is generally difficult to find a good trade-off between number of parameters, speed, and lack of overfitting.

Apicella, A., et al., 2021. **A survey on modern trainable activation functions**. *Neural Networks*, 138, pp. 14-32.

Not every layer fits into the framework of *linear projections* and *element-wise* non-linearities. For example, the **gated linear unit** (GLU) is composed of multiplicative (Hadamard) interactions between two blocks:

$$f(\mathbf{x}) = \sigma(\mathbf{W}_1\mathbf{x}) \odot (\mathbf{W}_2\mathbf{x}) \tag{17}$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are trained. This has become extremely popular in recent LLM models, such as the Llama family.

Inverting the order of linear and non-linear operations, we can write a linear layer as:

$$h_i = \sum_j W_{ij}\phi(x_j) \tag{18}$$

If we parameterize the activation functions (e.g., the trainable Swish) we can have different AFs for each (i, j) pair:

$$h_i = \sum_j W_{ij}\phi_{ij}(x_j) \tag{19}$$

This has become popular under the name of **Kolmogorov-Arnold networks** (KANs).

Liu, Z., et al., 2024. **KAN: Kolmogorov-Arnold networks**. *arXiv preprint arXiv:2404.19756*.

- **Chapter 5** from the book.