Machine learning - Master ICFP 2019-2020

Neural Networks

Lénaïc Chizat

March 13, 2020

In today's class, we introduce neural networks which is a broad class of models. We present how to train them, and some of their theoretical aspects. This course is taken from Francis Bach and Gabriel Peyré's lecture notes.

1 Introduction

This is just an introductory class on neural networks, to go deeper, please look at the following references:

- a reference book `https://www.deeplearningbook.org/`
- practice driven course `https://mlelarge.github.io/dataflowr-web/`
- observe dynamical behavior on simple toy tasks `https://playground.tensorflow.org/`

In this course, the main focus has been on methods to learn from n observations $(x_i, y_i), i = 1, \dots, n$, with $x_i \in \mathcal{X}$ (input space) and $y_i \in \mathcal{Y}$ (output/label space).

A large class of methods relies on minimizing a regularized empirical risk with respect to a function $f : \mathcal{X} \to \mathbb{R}$ where the following cost function is minimized:

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

where $\ell : \mathcal{Y} \times \mathbb{R} \to \mathbb{R}$ is a loss function, and $\Omega(f)$ is a regularization term. Typical examples were:

- **Regression:** $\mathcal{Y} = \mathbb{R}$ and $\ell(y_i, f(x_i)) = \frac{1}{2}(y_i - f(x_i))^2$.
- **Classification:** $\mathcal{Y} = \{-1, 1\}$ and $\ell(y_i, f(x_i)) = \phi(y_i f(x_i)))$ where ϕ is convex, e.g. $\phi(u) = \max\{1 - u, 0\}$ (hinge loss leading to the support vector machine) or $\phi(u) = \log(1 + \exp(-u))$ (leading to logistic regression).

The class of functions we have considered so far were:

- **Affine functions:** when $\mathcal{X} = \mathbb{R}^d$, we consider $f(x) = w^\top x + b$, with parameters $(w, b) \in \mathbb{R}^{d+1}$.
  Pros: simple to implement, convex optimization (gradient descent). Complexity in $O(nd)$.
  Cons: only applies to vector spaces, only linear.
- **Non-linear functions through kernel methods:** requires (implicitly) a feature vector $\Phi(x) \in \mathcal{F}$ (feature space), accessed through a kernel $k(x, x') = \langle \Phi(x), \Phi(x') \rangle$.
  Pros: non-linear predictions, simple to implement, convex optimization.
  Cons: complexity in $O(n^2)$.

The goal of this lecture is to explore another class of functions for non-linear predictions, namely neural networks.

2 A single neuron

2.1 A bit of history

- **Artificial neuron model** (McCulloch and Pitts, 1943) [1]: $f(x) = \sigma(w^\top x + b)$, with σ non-linear (typically non-decreasing) function. Loose connection with actual biological neurons.
- **Perceptron** (Rosenblatt, 1958) [2]: learning by stochastic gradient descent update rule.
- **Activation functions**:
  - sigmoid $\sigma(u) = \frac{1}{1+e^{-u}}$,
  - step $\sigma(u) = \mathbf{1}_{u>0}$,
  - rectified linear unit (ReLU) $\sigma(u) = (u)_+ = \max\{u, 0\}$,
  - hyperbolic tangent $\sigma(u) = \tanh(u)$.

2.2 Sigmoid activation with cross-entropy loss is logistic regression

- For least-squares, the loss for a single pair (x, y) is

  $$\ell(y, f(x)) = \frac{1}{2}(y - f(x))^2 = \frac{1}{2}(y - \sigma(w^\top x + b))^2,$$

  and is in general not convex in (w, b) unless σ is linear.
- For classification, the loss for a single pair (x, y) is

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

  and is in general not convex in (w, b) (unless σ is linear).
- For the sigmoid function $f(x) = \sigma(w^\top x + b) \in \,]0, 1[$ can be seen as a probability and thus it is natural to consider the model on $Y \in \{-1, +1\}$, where $p(Y = 1 \mid x) = \sigma(w^\top x + b)$, and thus

$p(y \mid x) = \sigma(y(w^\top x + b))$. If the cross-entropy (a.k.a. log-loss in maximum likelihood) loss is used, that is,

$$-\mathbf{1}_{y=1} \log p(Y = 1 \mid x) - \mathbf{1}_{y=-1} \log p(Y = -1 \mid x)$$

then the loss to minimize is exactly

$$-\log \sigma(y(w^\top x + b)) = \log(1 + \exp(-y(w^\top x + b))),$$

which is exactly logistic regression.

2.3 Multiclass logistic loss

- Logistic regression can be interpreted as using the cross-entropy/log loss on the predicted probability distribution $(1/(e^z + 1), e^z/(e^z + 1))$ on $\{-1, +1\}$ for an output $z \in \mathbb{R}$ of the model.
- For a k-class classification task i.e. $y \in \{1, \ldots, k\}$, we can consider a model that outputs a k-vector $z \in \mathbb{R}^k$ and use the cross-entropy loss on the predicted probability $(e^{z_i}/\sum_{i=1}^k e^{z_i})_{i=1}^k$. Using the "one-hot" encoding $\tilde{y} = \delta_i \in \mathbb{R}^k$ when $y = i$, this gives the *multiclass logistic loss*

  $$\ell(y, z) = \log\left(\sum_{i=1}^k e^{z_i}\right) - \tilde{y}^\top z$$

  which is convex.

3 One hidden layer

3.1 Definition

- Limitations of single neuron: (a) only linear classification boundary (if σ monotonous) (b) classical XOR problem.
- Consider $f(x) = y^{(2)} \in \mathbb{R}$ with $x = x^{(0)} \in \mathbb{R}^d$, $y^{(1)} \in \mathbb{R}^m$

  $$\begin{aligned} y^{(1)} &= (W^{(1)})^\top x + B^{(1)} \\ x^{(1)} &= \sigma(y^{(1)}) \\ y^{(2)} &= (W^2)^\top x^{(1)} + B^{(2)} \end{aligned}$$

  with $W^{(1)} \in \mathbb{R}^{d \times m}$, $B^{(1)} \in \mathbb{R}^m$, $W^{(2)} \in \mathbb{R}^{m \times 1}$ and $B^{(2)} \in \mathbb{R}^1$.
- we could also add a bias and apply the non-linearity again to the output.
- Compute the output y is called computing a *forward* pass

3.2 Approximation properties

Clearly, this class of function solves the XOR problem. In fact, it can approximate any continuous function given sufficiently many hidden neurons.

Proposition 1 *Assume that the activation function σ is continuous and not polynomial. For any continuous function f^* on $[0,1]^d$ and any $\epsilon > 0$, there exists a one-hidden layer neural network f such that* $\sup_{x\in[0,1]^d} |f(x) - f^*(x)| < \epsilon$.

Proof Graphical proof for the case where $\sigma(u) = (u)_+ = \max\{0, u\}$ and $d = 1$. ■

3.3 Link with kernel methods

- A one-hidden layer neural network corresponds to a linear classifier with feature vector

$$\Phi(x) = \frac{1}{\sqrt{m}}\sigma((W^{(1)})^\top x + B^{(1)})$$

  parameterized by $W^{(1)}$ and $B^{(1)}$, with kernel

$$k(x, x') = \frac{1}{m}\sum_{i=1}^{m} \sigma((W_i^{(1)})^\top x + B_i^{(1)})\sigma((W_i^{(1)})^\top x' + B_i^{(1)}).$$

- Most important aspect: the feature vector is of finite dimension and learned from data.
- With random independent and identically distributed weights $W_i^{(1)} \in \mathbb{R}^m$ and $B_i^{(1)} \in \mathbb{R}$,

$$k(x, x') \to \mathbb{E}\Big[\sigma((W^{(1)})^\top x + B^{(1)})\sigma((W^{(1)})^\top x' + B^{(1)})\Big].$$

- This kernel can be computed in closed form for simple activations and distributions of weight [3].
- An infinite number of random input weights lead to a kernel method. With a finite number m, this is called a *random feature* approximation of the kernel.

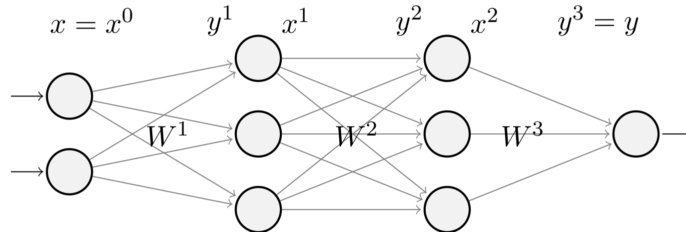
4 Multilayer neural networks

4.1 Multilayer perceptron

The most simple form of deep neural networks is a multilayer fully-connected neural network. It is of the form $f(x^{(0)}) = y^{(L)}$ with input $x^{(0)}$ and output $y^{(L)}$ given, ignoring intercept/bias terms:

$$y^{(\ell)} = (W^{(\ell)})^\top x^{(\ell-1)}$$
$$x^{(\ell)} = \sigma(y^{(\ell)})$$

- Approximation properties: can approximate any function from differentiable activation function
- Link with kernel methods: similar as before, but more complex.



4.2 Convolutional neural networks

- In order to be able to tackle data of large size and to improve preformances, it is important to leverage the prior knowledge about the structure of the typical data to process. For instance, for signal, images or videos, it is important to take into account the translation invariance (up to boundary issues) of the domain.
- Convolutional neural networks are obtained by considering that the manipulated vectors $x^{(\ell)}$ at depth ℓ in the network are of the form $x^{(\ell)} \in \mathbb{R}^{n_\ell \times d_\ell}$, where n_ℓ is the number of "spatial" positions (typically along a 1-D, 2-D, or 3-D grid) and d_ℓ is the number of "channels". For instance, for color images, one starts with n_0 being the number of pixels, and $d_0 = 3$.
- The linear operator $L^{(\ell)} : \mathbb{R}^{n_\ell \times d_\ell} \to \mathbb{R}^{n_\ell \times d_{\ell+1}}$ is chosen translation invariant (up to boundary artefacts) and hence a convolution along each channel (note that the number of channels can change between layers). It is thus parameterized by a set of filters $(\psi_{r,s}^{(\ell)})$ for $r = 1, \ldots, d_\ell$ and $s = 1, \ldots, d_{\ell+1}$. Denoting $x^{(\ell)} = (x_{s,\cdot}^{(\ell)})_{s=1}^{d_\ell}$ the different layers composing $x^{(\ell)}$, the linear map reads

$$(L^{(\ell)} x^{(\ell)})_{r,\cdot} = \sum_{s=1}^{d_\ell} \psi_{r,s}^{(\ell)} * x_{s,\cdot}^{(\ell)}$$

  and the bias term $b^{(\ell)} \in \mathbb{R}$ is cosntant (to maintain translation invariance).
- The non-linear maps across layers serve two purposes: as before a pointwise non-linearity is applied, and then a sub-sampling helps to reduce the size of the next layers. Denoting by m_k the amount of down-sampling, where usually $m_k = 1$ (no reduction) or $m_k = 2$ (reduction by a factor two in each direction). One has

$$\tilde{\sigma}^\ell(u) = (\sigma(u_{s,m_k \cdot}))_{s=1,\ldots,d_{\ell+1}}.$$

- The intuition behind such model is that as one moves forward through the layers, the neurons are receptive to larger areas in the image domain. Using an increasing number of channels helps to define different classes of "detectors" (for the first layer, they detect simple patterns such as edges and corner, and progressively capture more elaborated shapes).
- In practice, the last few layers (2 or 3) of such a CNN architectures are chosen to be fully connected. This is possible because, thanks to the sub-sampling, the dimension of these layers are small.

5 Training with back-propagation

5.1 Reverse differentiation for the composition of functions

- While the objective function for the empirical risk minimization with neural networks is non-convex, (stochastic) gradient descent is a good heuristic to train neural networks. The main computational cost is the evaluation of gradients on a single sample, i.e. given a sample (x, y) and current parameters w, to compute:

$$\nabla_w[\ell(y, h(w, x))].$$

- For concreteness, suppose that the function to be differentiated as the form

$$R(w) = \mathcal{L} \circ f_L \circ \cdots \circ f_1(w)$$

where $f_\ell : \mathbb{R}^{n_\ell} \to \mathbb{R}^{n_{\ell+1}}$ and $\mathcal{L} : \mathbb{R}^{n_L} \to \mathbb{R}$. This is often called a feedforward model.

- Then one can compute the gradient $\nabla R(w)$ in two steps:
  - a *forward pass*, where one evaluates the functions itself and keeps track of all the intermediate computations, i.e. initializing $w_1 = w$, one computes

$$w_{\ell+1} = f_\ell(w_\ell) \quad \text{and} \quad R(w) = \mathcal{L}(w_L).$$

  - a *backward pass*, where one uses the chain rule together with the Jacobian transposition

$$\nabla R(w) = [\partial f_1(w_1)]^\top \circ \cdots \circ [\partial f_L(w_L)]^\top \nabla \mathcal{L}(w_{L+1}),$$

  to define the backward recursion, initialized with $g_L = \nabla \mathcal{L}(w_L)$, and then

$$g_{\ell-1} = [\partial f_\ell(w_\ell)]^\top (h_\ell) \quad \text{and} \quad \nabla R(w) = g_1.$$

- This computation should be compared to the complexity of *forward* accumulation

$$\nabla R(w)^\top = \nabla \mathcal{L}(w_L)^\top [\partial f_L(w_{L-1})] \dots [\partial f_1(w_1)].$$

- This idea of *backward-propagation* can be generalized to any computational graph: evaluating the gradient of a function $R : \mathbb{R}^n \to \mathbb{R}$ built from a sequence of elementary operations has (within a constant factor) the same complexity as evaluating the function.

5.2 Application to fully connected neural networks

Assume that we have already computed the forward pass and stored the variables $x^{(\ell)}, y^{(\ell)}$. Let $\delta y^{(L)} = \nabla \ell(y, y^{(L)})$. Then the back-propagation equations are

$$\begin{aligned} \delta x^{(\ell-1)} &= W^{(\ell)} \delta y^{(\ell)} \\ \delta y^{(\ell)} &= \sigma'(y^{(\ell)}) \delta x^{(\ell)} \\ \delta W^{(\ell)} &= x^{(\ell-1)} [\delta y^{(\ell)}]^\top. \end{aligned}$$

where $\delta W^{(\ell)}$ are the gradients used in the (stochastic) gradient descent algorithm.

Other remarks:

- T he parameters are initialized at random. The variance and mean has to be chosen appropriately for the signal magnitude to be preserved through the forward and backward passes.
- While stochastic gradient descent remains an algorithm of choice, several tricks have been observed to lead to better stability and performance: specific step-size decay schedules, momentum, batch-normalization,...

6 Research questions

Neural networks are not yet well understood from a theoretical perspective. Some active research questions are the following:

- The objective function is non-convex. Why do gradient-based methods perform well in practice?
- Neural networks have a large number of parameters and are typically not explicitly regularized. When and why do they not overfit?
- In the first lecture, we have seen the no-free-lunch theorem, which tells that a machine learning method can only work well on a restricted class of problems. On what kind of problem structures are neural networks efficient?

References

[1] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943.

[2] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[3] Youngmin Cho and Lawrence K Saul. Kernel methods for deep learning. In *Advances in neural information processing systems*, pages 342–350, 2009.