Machine Learning Class - ENS
Neural networks

Francis Bach

January 11, 2019

1 Introduction

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

- `http://www.deeplearningbook.org/`
- `https://www.di.ens.fr/~lelarge/dldiy/`

In this class, 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 $\ell(y_i, f(x_i)) = \varphi(y_i f(x_i))$ where φ is convex, e.g., $\varphi(u) = \max\{1 - u, 0\}$ (hinge loss leading to the support vector machine) or $\varphi(u) = \log(1 + e^{-u})$ (leading to logistic regression).

  Note that the usual goal of binary classification is to consider the $0 - 1$ loss, which we don't consider here.

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 proportional to $O(nd)$.

  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), known through a kernel $k(x, x') = \langle \Phi(x), \Phi(x') \rangle$.

  Pros: non-linear predictions, simple to implement, convex optimization.

  Cons: complexity is at least $O(n^2)$.

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

2 A single neuron

2.1 A (tiny) bit of history / background

- **Artificial neuron model** (McCulloch and Pitts, 1942): $f(x) = \sigma(w^\top x + b)$, with σ non-linear (typically non-decreasing) function. Loose connection with actual biological neurons.
- **Perceptron** (Rosenblatt, 1958): Learning by stochastic gradient descent update rule.
- **Activation functions**:

  sigmoid $\sigma(u) = \frac{1}{1+e^{-u}}$,

  step $\sigma(u) = 1_{u>0}$,

  rectified linear unit (ReLU) $\sigma(u) = (u)_+ = \max\{u, 0\}$.

2.2 Sigmoid activation + cross-entropy loss = 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 not convex in (w, b) (unless σ is linear).

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

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

  and is 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 $Y \in \{-1, 1\}$, wher $p(Y = 1|x) = \sigma(w^\top x + b)$, and thus $p(y|x) = \sigma(y(w^\top x + b))$.

  If the cross-entropy loss is used (or equivalently maximum likelihood), that is, $-1_{y=1} \log p(Y = 1|x) - 1_{y=-1} \log p(Y = -1|x)$, then the loss to minimize is exactly $-\log \sigma(y(w^\top x + b)) = \log(1 + \exp(-y(w^\top x + b)))$, which is exacly logistic regression.

- Consequences:

  (a) neural network with no hidden layers are reduced to linear predictors,

  (b) last layers of deeper network will be treated in practice in the same way (for classification: cross-entropy loss, for regression: no activation function).

2.3 Gradient and SGD

- The gradient of $\ell(y, f(x)) = L_y(f(x, w, b)) = L_y(\sigma(w^\top x + b))$ can be computed using the chain rule, with $f(x) = \sigma(w^\top x + b)$:

$$\begin{aligned}\frac{\partial[L_y \circ f]}{\partial w} &= L'_y(f(x,w,b))\frac{\partial f}{\partial w} = L'(f(x,w,b))\sigma'(w^\top x + b)x \\ \frac{\partial[L_y \circ f]}{\partial b} &= L'_y(f(x,w,b))\frac{\partial f}{\partial b} = L'_y(f(x,w,b))\sigma'(w^\top x + b)\end{aligned}$$

- (optional) Safety check: for example, for logistic regression, with label $z = (1+y)/2 \in \{0, 1\}$, when $L_y(u) = -z \log u - (1-z)\log(1-u)$, $L'_y(u) = -\frac{z}{u} + \frac{1-z}{1-u}$, and $\sigma(u) = \frac{1}{1+e^{-u}}$, $\sigma'(u) = \sigma(u)(1-\sigma(u))$, we get

$$\begin{aligned}\frac{\partial[L \circ f]}{\partial b} &= \Big[-\frac{z}{\sigma(w^\top x + b)} + \frac{1-z}{1-\sigma(w^\top x + b)}\Big]\sigma(w^\top x + b)(1 - \sigma(w^\top x + b)) \\ &= -z(1-\sigma) + (1-z)\sigma.\end{aligned}$$

  But, we have directly $L(f(x)) = z\log(1 + e^{-w^\top x - b}) + (1-z)\log(1 + e^{w^\top x + b})$, and we can take the derivative as $-z(1-\sigma) + (1-z)\sigma$, which is the same.

- Batch gradient: $J(w, b) = \frac{1}{n}\sum_{i=1}^n L_{y_i}(\sigma(w^\top x_i + b)) + \Omega(w, b)$:

$$\frac{\partial J}{\partial w} = \frac{1}{n}\sum_{i=1}^n L'_{y_i}(f(x,w,b))\sigma'(w^\top x_i + b)x_i + \frac{\partial \Omega}{\partial w}$$

  Algorithm requiring access to the entire data set at each iteration:

$$\begin{aligned}w &\leftarrow w - \gamma\frac{\partial J}{\partial w} = w - \frac{\gamma}{n}\sum_{i=1}^n L'_{y_i}(f(x,w,b))\sigma'(w^\top x_i + b)x_i - \gamma\frac{\partial \Omega}{\partial w} \\ b &\leftarrow b - \gamma\frac{\partial J}{\partial b} = b - \frac{\gamma}{n}\sum_{i=1}^n L'_{y_i}(f(x,w,b))\sigma'(w^\top x_i + b) - \gamma\frac{\partial \Omega}{\partial b}\end{aligned}$$

- Stochastic gradient descent (with mini-batches), where I is a set of indices in $\{1, \ldots, n\}$:

$$\begin{aligned}w &\leftarrow w - \gamma\frac{\partial J}{\partial w} = w - \frac{\gamma}{|I|}\sum_{i\in I} L'_{y_i}(f(x,w,b))\sigma'(w^\top x_i + b)x_i - \gamma\frac{\partial \Omega}{\partial w} \\ b &\leftarrow b - \gamma\frac{\partial J}{\partial b} = b - \frac{\gamma}{|I|}\sum_{i\in I} L'_{y_i}(f(x,w,b))\sigma'(w^\top x_i + b) - \gamma\frac{\partial \Omega}{\partial b}\end{aligned}$$

- Convergence of SGD: (a) not convergent with constant step-size, (b) need decreasing step-size, (c) effect of averaging, (c) not a descent algorithm.

- No convergence to global optimum because of lack of convexity!

3 One-hidden layer

3.1 Definition

- Limitations of single neuron: (a) classical XOR problem, (b) only linear predictions.
- Parameterization: $x \in \mathbb{R}^d$, $h \in \mathbb{R}^m$, $y \in \mathbb{R}$:

$$\begin{aligned} h &= \sigma\big[(W^h)^\top x + B^h\big] \\ y &= \sigma\big[(w^o)^\top h + b^o\big], \end{aligned}$$

  with $W^h \in \mathbb{R}^{d \times m}$ and $B^h \in \mathbb{R}^m$, and $w^o \in \mathbb{R}^m$ and $b^o \in \mathbb{R}$. We denote by $W_i^h \in \mathbb{R}^d$ the i-th input/hidden weight.

  Computing the output y requires a *forward* pass.

3.2 Gradient through back-propagation

- We have for the hidden layer:

$$\begin{aligned} \frac{\partial h_i}{\partial W_i^h} &= \sigma'\big[(W_i^h)^\top x + B_i^h\big] x \\ \frac{\partial h_i}{\partial B_i^h} &= \sigma'\big[(W_i^h)^\top x + B_i^h\big] \end{aligned}$$

- For the output layer (and the hidden-output parameters):

$$\begin{aligned} \frac{\partial y}{\partial w^o} &= \sigma'\big[(w^o)^\top h + b^o\big] h \\ \frac{\partial y}{\partial b^o} &= \sigma'\big[(w^o)^\top h + b^o\big] \end{aligned}$$

- For the output layer (and the input-hidden parameters):

$$\begin{aligned} \frac{\partial y}{\partial W_i^h} &= \sum_{j=1}^m \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial W_i^h} = \sigma'\big[(w^o)^\top h + b^o\big](w_i^o) \frac{\partial h_i}{\partial W_i^h} \\ \frac{\partial y}{\partial B_i^h} &= \sum_{j=1}^m \frac{\partial y}{\partial h_j} \frac{\partial h_j}{\partial B_i^h} = \sigma'\big[(w^o)^\top h + b^o\big](w_i^o) \frac{\partial h_i}{\partial B_i^h} \end{aligned}$$

- Running-time complexity. Vectorized operations adapted to GPUs.

3.3 Approximation properties

- Solves the XOR problem
- Can approximate any continuous function given sufficiently many hidden neurons, from differentiable activation function. Simple graphical proof for rectified linear units in one dimension. Requires activation not to be a polynomial.

  G. Cybenko. Approximation by superposition of a sigmoidal function. Mathematics of Control, Signal and Systems, 2:303-314, 1989.

  K. Hornik. Some new results on neural network approximation. Neural Networks, 6:1060-1072, 1993.

3.4 Link with kernel methods

- When no activation is used at the output layer, we have:

$$\begin{aligned} h &= \sigma\big[(W^h)^\top x + B^h\big] \\ y &= (w^o)^\top h + b^o. \end{aligned}$$

This corresponds to a linear classifier with feature vector $\Phi(x) = \frac{1}{\sqrt{m}}\sigma\big[(W^h)^\top x + B^h\big]$, parameterized by W^h and B^h, with kernel

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

Most important aspect: feature vector of finite dimension and *learned* from data.

- With random independent and identically distributed weights $W_i^h \in \mathbb{R}^m$ and $B_i^h \in \mathbb{R}$,

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

Can be computed in closed form for simple distributions of weights (see, e.g., Cho, Y., & Saul, L. K. (2009). Kernel methods for deep learning. In Advances in neural information processing systems (pp. 342-350)) . Thus an infinite number of random input weights lead to a kernel method.

4 Multiple hidden layers

- Ignoring the constant terms, two hidden layers can be expressed as:

$$\begin{aligned} y &= \sigma(W_1^\top \sigma(W_2^\top \sigma(W_3^\top x))) = f_1 \circ f_2 \circ f_3(x) \\ y &= f_1(\theta_1, y_2) \\ y_2 &= f_2(\theta_2, y_3) \\ y_3 &= f_3(\theta_3, x) \end{aligned}$$

- Gradient through back-propagation. We get:

$$\begin{aligned} \frac{\partial y}{\partial \theta_1} &= \frac{\partial f_1}{\partial \theta_1} \\ \frac{\partial y}{\partial y_2} &= \frac{\partial f_1}{\partial y_2} \\ \frac{\partial y}{\partial \theta_2} &= \frac{\partial y}{\partial y_2}\frac{\partial y_2}{\partial \theta_2} = \frac{\partial y}{\partial y_2}\frac{\partial f_2}{\partial \theta_2} \\ \frac{\partial y}{\partial y_3} &= \frac{\partial y}{\partial y_2}\frac{\partial y_2}{\partial y_3} = \frac{\partial y}{\partial y_2}\frac{\partial f_2}{\partial y_3} \\ \frac{\partial y}{\partial \theta_3} &= \frac{\partial y}{\partial y_3}\frac{\partial y_3}{\partial \theta_3} = \frac{\partial y}{\partial y_3}\frac{\partial f_3}{\partial \theta_2} \end{aligned}$$

- Approximation properties: Can approximate any function from differentiable activation function.

  A. Lapedes and R. Farber. How neural nets work. In Anderson, editor, Neural Information Processing Systems, pages 442-456. New York, American Institute of Physics, 1987.

  G. Cybenko. Continuous valued neural networks with two hidden layers are sufficient. Technical report, Dep. of Computer Science, Tufts University, Medford, MA, 1988.

- Link with kernel methods: same as before, but more complex.

5 Extensions

5.1 Convolutional neural networks

- Working on a 512 x 512 image requires weight sharing (for both numerical and statistical reasons).
- $h_i = \sum_{j=1}^{d} W_{ij}^h x_j$. W_{ij}^h depends only $i - j$
- Often used with subsampling or pooling
- Partial invariance to translation is a good prior.

5.2 Automatic differentiation

- Finite differences and the "complex trick" (`https://blogs.mathworks.com/cleve/2013/10/14/complex-step-differentiation/`)
- No need to code everything for deep models to obtain the exact gradient.
- See `http://www.autodiff.org/`

5.3 Applications

- Computer vision (CNN)
- Speech (CNN)
- Natural language processings (recurrent neural networks)