Introduction to Deep Learning

LECTURE 1

Elad Hoffer
Nir Ailon
Ran El-Yaniv

Technion Israel Institute Of Technology,
Computer-Science department

Overview

1. Background

2. Training Neural Networks

2.1 Objective and Loss functions

2.2 Stochastic-Gradient-Descent

2.3 Backpropagation

Key aspects of the course

- The course will cover various aspects of neural network models that a currently used to solve various machine learning problems.
- It is intended as a practical, hands-on guide at using these kind of models
- As such, technical tutorials will be interleaved, employing the *Torch* scientific-computing framework
- We assume students are familiarized with basic machine learning concepts and models

Key aspects of the course

Some cautionary points

This field is currently at a point where theory is sometime lacking.

- As such, technicalities and "tricks of the trade" are important and inseparable from any real-world use of these models.
- This also means that proofs and mathematical accuracy are frequently sacrificed in favor of intuition and empirical findings.

Projects

- Students accepted to perform a project have been notified, we are not able to include any more (sorry)
- Intermediate assignments will be given as an instruction tool. Doing them will also affect final assessment.
- Those not doing registered are also encouraged to do the assignments and report their findings.

Background

What is Deep Learning?

Deep learning (deep structured learning or hierarchical learning) is a set of algorithms in machine learning that attempt to model high-level abstractions in data by using model architectures composed of multiple non-linear transformations.

Deep learning models appear as deep networks, deep Boltzman-machines, deep belief networks, deep auto-encoders and more.

But usually a deep learning model is simply a: Neural networks with MORE THAN 1 HIDDEN LAYERS

Basic models of neural networks

We will start by examining the most basic building block of a neural network - a *fully-connected* layer. This layer computes and affine function of its input: For $x \in \mathbb{R}^n$, weights $W \in \mathbb{R}^{m \times n}$ and bias vector $b \in \mathbb{R}^m$

$$f(x) = \varphi(Wx + b)$$

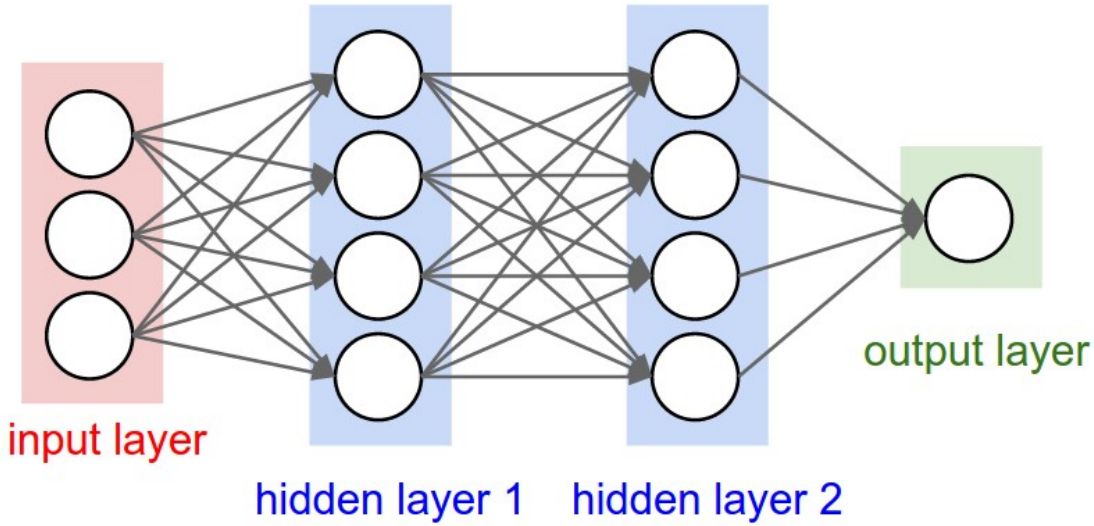
Where φ non-linear *activation function*.

Using the step-function as activation, this is the well-known *Perceptron* (Rosenblatt '57)

neural networks

We can now stack multiple layers to form a simple network.

- Networks are fed with inputs and trained to output the corresponding targets.
- Hidden layers (or hidden activations/representations), are said to *represent* the data, by applying non-linear transformations.



Why are non-linearities important?

Common non-linearities used:

- Sigmodial function - $Sigmoid(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent - $TanH(x) = \frac{1-e^{-x}}{1+e^{x}}$
- Rectified linear unit - $ReLU(x) = \max(x, 0)$

We use non-linearities to prevent our network from "collapsing" to a linear model.

VISUALIZING SIMPLE NN

Universal approximation theorem

One noteworthy aspect of neural networks is the *universal approximation theorem* (Hornik, 1991)

- Feed-forward network with a single hidden layer containing a finite number of neurons, can approximate continuous functions on compact subsets of $\mathbb{R}^n$, (under mild assumptions on the activation function).

However, this theorem does not indicate the number of neurons needed for a given error. It can be shown that, in some cases, that number grows exponentially.

Training Neural Networks

Neural network as a learning model

From the perspective of statistical learning theory, by specifying a neural network architecture (the underlying graph and the activation function) we obtain a hypothesis class, namely, the set of all prediction rules obtained by using the same network architecture while changing the weights of the network.

- Learning the class involves finding a specific set of weights, based on training examples
- We are interested in *generalizing* such that our predictor will yield good performance on unseen examples

Objective and Loss functions

We will focus on problem of the form: given training examples $\{(x^{(1)}, t^{(1)}), \ldots, (x^{(N)}, t^{(N)})\}$ such that $x^{(i)}$ are the inputs and $t^{(i)}$ are the targets, we wish to minimize an objective function between the targets, and the network output $y = F(x)$.

In order to optimize our model according to the desired objective, we will need to choose an *error function*.

For example, we can perform regression using a network with a *Mean-square-error (MSE)* - computing for the network outputs $y^{(i)}$ vs targets $t^{(i)}$

$$E_{MSE}(y, t) = \frac{1}{N} \sum_{i=1}^{N} \|y^{(i)} - t^{(i)}\|^2$$

Objective and Loss functions

A common objective is to *classify* correctly a set of examples, such that t_i is the class label. For each input x, the output of the network $y = F(x)$ indicates a probability measure over the set of classes C, and the classification is taken as

$$c = \arg\max_j y$$

where $y \in \mathbb{R}^{|C|}$, $\sum_{j=1}^{|C|} y_j = 1$, $0 \leq y \leq 1$

In order to create this normalized measure, we use another non-linearity - *SoftMax* function

$$\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}$$

which we can use to do multi-class cross entropy regression.

Objective and Loss functions

The error function for cross-entropy regression is

$$E_{\text{cross-entropy}}(y, t) = -\sum_{j \in C} t_i \log y_i$$

and when considering t as an indicator, where c_t is the correct label

$$t = \begin{cases} 1, & \text{if} \quad j = c_t \\ 0, & \text{otherwise} \end{cases}$$

We get the "negative-log-likelihood" criterion

$$E_{\text{nll}}(y, t) = -\sum_{j \in C} t_j \log y_j = -\log y_{c_t}$$

Stochastic-Gradient-Descent

- Training a neural network with a large number of parameters requires a simple optimization technique.
- Usually, a a variant of *Stochastic gradient descent (SGD)* is used, using noisy subset estimation of the gradient.
- It requires $O(n)$ number of computations and memory use, where n is the number of parameters
- The most simple update-rule is:

SGD update rule

$$w_{t+1} = w_t - \varepsilon \cdot \frac{\partial E}{\partial w_t}$$

where w is the optimized parameter (weight), E is the error (loss function estimation) and ε is the *Learning-Rate*

Calculating the gradient

We will start by differentiating a *fully-connected* layer without the non-linearity. For $x \in \mathbb{R}^n$, weights $W \in \mathbb{R}^{m \times n}$ and bias vector $b \in \mathbb{R}^m$

$$f(x) = Wx + b$$

Knowing the error gradient with regard to the output $\frac{\partial E}{\partial f}$, we can compute the gradient with regard to input

$$\frac{\partial E}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial E}{\partial f} = W^T \frac{\partial E}{\partial f}$$

and with regard to parameters (weight+bias)

$$\frac{\partial E}{\partial W} = \frac{\partial E}{\partial f} x^T \quad , \quad \frac{\partial E}{\partial b} = \frac{\partial E}{\partial f}$$

Backpropagation

This can be expanded to arbitrary graph (DAG) using the chain rule for derivatives - a.k.a *Backpropagation*

- Starting at the output and the gradient of our error function, we can backpropagate our gradients towards previous layers
- Each step is local - requiring the (saved) output of previous layer, and the gradient w.r.t to the next layer
- For each layer we calculate the gradient w.r.t to input, and gradient w.r.t weights

Additional - Universal approximation theorem

Let $\varphi(\cdot)$ be a non-constant, bounded and monotonically increasing continuous function. Let I_m denote the hypercube $[0,1]^m$. Then, given any function $f \in C(I_m)$ and $\varepsilon > 0$, there exists an integer N and real constants $v_i, b_i \in \mathbb{R}$

$$F(x) = \sum_{i=1}^{N} v_i \varphi\left(w_i^T x + b_i\right)$$

such that

$$|F(x) - f(x)| < \varepsilon$$

for all $x \in I_m$. This still holds when replacing I_m with any compact subset of $\mathbb{R}^m$.