Deep Learning

AlejandroMllo

This document serves as a very brief summary of the topics covered in each chapter of the book Deep Learning [1].

Disclaimer: This document is completely extracted from [1], the author does not attribute any ownership over the material.

6 Deep Feedforward Networks

- They define a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learn the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation.
- There are no feedback connections in which outputs of the model are fed back into itself.
- The model is associated with a directed acyclic graph describing how the functions are composed together.
- Output layer: final layer (function) of the network.
- The training data provides noisy, approximate examples of $f^*(\boldsymbol{x})$ evaluated at different training points. Each example $\boldsymbol{x}$ is accompanied by a label $y \approx f^*(\boldsymbol{x})$, that specify what the output layer must do. The learning algorithm must decide how to use the hidden layers to best implement an approximation of f^* (the training data does not show a desired output for each of these layers).
- This networks require to initialize all weights to small random values.
- To apply gradient-based learning, a cost function and output representation must be chosen.
- The total cost of the network will often combine one of the primary cost functions with a regularization term.
- Most modern NN are trained using maximum likelihood.This means that the cost function is simply the negative log-likelihood:
$$J(\boldsymbol{\theta}) = -\mathbb{E}_{x,y\sim\hat{y}_{data}} \log p_{model}(\boldsymbol{y}|\boldsymbol{x})$$
The expansion of the above equation typically yields some terms that do not depend on the model parameters and may be discarded.
- The equivalence between maximum likelihood estimation and minimization of MSE holds regardless of the $f(\boldsymbol{x}; \theta)$ used to predict the mean of the Gaussian.
- The gradient of the cost function must be large and predictable enough to serve as a good guide for the learning algorithm.
- It is possible to view the cost function as a functional. A functional maps from functions to real numbers. It can have its minimum at some specified function.
- The choice of cost function is tightly coupled with the choice of output unit.
- Linear units for Gaussian distributions. The layer produces the mean of a conditional Gaussian distribution: $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$.
- Sigmoid units for Bernoulli distributions. Useful for predicting the value of a binary variable y.
- Logit: variable (usually denoted by z) defining a probability distribution based on exponentiation and normalization over binary variables.
- Softmax units for Multinoulli distributions. Useful to represent a PD over a discrete variable with n possible values.
- The objective functions that do not use a log to undo the exp of the softmax fail to learn when the argument to the exp becomes very negative.
- In general, NN represent a function $f(\boldsymbol{x}|\boldsymbol{y})$. The outputs of $f(\boldsymbol{x}|\boldsymbol{y}) = \boldsymbol{\omega}$ provide the parameters for a distribution over y. The loss function can then be interpreted as $-\log p(\mathrm{y}, \boldsymbol{\omega}(\boldsymbol{x}))$.
- Gaussian mixture: lets predict real values from a conditional distribution $p(\boldsymbol{y}|\boldsymbol{x})$ that can have several different peaks in $\boldsymbol{y}$ space for the same value of $\boldsymbol{x}$.

- The function used in the context of NN usually have defined left derivatives and defined right derivatives.
- Most hidden units can be described as accepting a vector of inputs $\boldsymbol{x}$, computing an affine transformation $\boldsymbol{z} = \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$, and then applying an element-wise nonlinear function $g(\boldsymbol{z})$.
- ReLU are an excellent default choice of hidden unit. They use the activation function $g(\boldsymbol{z}) = \max\{0, z\}$. They cannot learn via gradient-based methods on examples for which their activation is zero.
  Three generalizations of ReLU are based on using a nonzero slope α_i when $z_i < 0 : h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$.
  - Absolute value error rectification fixes $\alpha_i = -1$ to obtain $g(z) = |z|$.
  - Leaky ReLU fixes α_i to a small value like 0.01.
  - Parametric ReLU (PReLU) treats α_i as a learnable parameter.
- Maxout units divide $\boldsymbol{z}$ into groups of k values. Each maxout unit then outputs the maximum element of one of these groups: $g(\boldsymbol{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$, where $\mathbb{G}^{(i)}$ is the set of indices into the inputs for group i, $\{(i-1)k + 1, \cdots, ik\}$. They can resist a phenomenon called catastrophic forgetting, in which NN forget how to perform tasks that they were trained on in the past.
- Logistic Sigmoid activation function: $g(z) = \sigma(z)$. Sigmoidal units saturate across most of their domain. Their use as output units is compatible with the use of gradient-based learning when an appropriate cost function can undo the saturation of the sigmoid in the output layer.
- Hyperbolic Tangent activation function: $g(z) = \tanh(z) = 2\sigma(2z) - 1$. It resembles the identity function more closely, in the sense that $\tanh(0) = 0$ while $\sigma(0) = \frac{1}{2}$.
- It is acceptable for **some** layers of the NN to be purely linear.
- Radial basis function (RBF) unit: $h_i = \exp(-\frac{1}{\sigma_i^2} ||\boldsymbol{W}_{:,i} - \boldsymbol{x})||^2)$. It becomes more active as $\boldsymbol{x}$ approaches a template $\boldsymbol{W}_{:,i}$. Because it saturates to 0 for most $\boldsymbol{x}$, it can be difficult to optimize.
- Softplus units: $g(a) = \zeta(a) = \log(1 + \mathrm{e}^a)$. Its use is generally discouraged.
- Hard tanh unit: $g(a) = \max(-1, \min(1, a))$.
- Architecture of the network: how many units it should have and how these units should be connected to each other.
- Most NN are organized into groups of units called layers. Most NN architectures arrange these layers in a chain structure, with each layer being a function of the layer that preceded it. In these architectures the main considerations are choosing the depth of the network and the width of each layer. Deeper networks generalize better (most of the time).
- Universal Approximation theorem: a feedforward NN with a linear output layer and at least one hidden layer with any "squashing" activation function can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.
- Many architectures build a main layers chain but then add extra architectural features to it, such as skip connections going from layer i to layer $i+2$ or higher. These skip connections make it easier for the gradient to flow from output layers to layers nearer the input.
- During training, forward propagation can continue onward until it produces a scalar cost $J(\boldsymbol{\theta})$. The back-propagation algorithm allows the information from the cost to then flow backward through the network in order to compute the gradient.
- Back-propagation is the method for computing the gradient, while another algorithm, such as stochastic gradient descent, is used to perform learning using this gradient. In learning algorithm, the gradient that is most often required is the gradient of the cost function w.r.t. the parameters: $\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$.
- The chain rule of calculus is used to compute the derivatives of functions formed by composing other functions whose derivatives are known. Backprop computes the chain rule, with a specific order of operations that is highly efficient.
  Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n, g$ maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$, then
  $$\frac{\partial z}{\partial x_i} = \sum_i \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$
  In vector notation, this may be equivalently written as
  $$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^\top \nabla_{\boldsymbol{y}} z,$$

where $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the $n \times m$ Jacobian matrix of g.
So, the gradient of a variable $\boldsymbol{x}$ can be obtained by multiplying a Jacobian matrix $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ by a gradient $\nabla_{\boldsymbol{y}} z$. The backprop algorithm consists of performing such a Jacobian-gradient product for each operation in the graph.

- Algorithms 6.1, 6.2, 6.3, 6.4, 6.5 in the book further enhance understanding of backprop.

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.