Deep Learning: A Concise Review

Michael Savastio

Abstract

We provide a concise review of deep learning. It is intended that with primarily this review (and, to a lesser extent, references therein) one should be able to build any of the most common deep learning constructs completely from scratch.

1 Introduction and Overview

An "artificial neural network" (ANN) is a construct for fitting a function to data (i.e. using empirical risk minimization) consisting of a set of "composed" basis function methods which are fit simultaneously.

Suppose we have some data X and y, in general arbitrary rank tensors. The goal is to learn a function $F : X \to \hat{y}$ which approximates the conditional probability distribution $P(y|X)$. At its simplest, deep learning takes F to be of the form

$$\begin{aligned} F(X) = \hat{y} &= f_{(n)}(W^{(n)} \cdot y^{(n-1)} + b^{(n)}) \\ y^{(n-1)} &= f_{(n-1)}(W^{(n-1)} \cdot y^{(n-2)} + b^{(n-1)}) \\ &\dots \\ y^{(1)} &= f_{(1)}(W^{(1)} \cdot X + b^{(1)}) \end{aligned} \tag{1}$$

where the $f_{(n)}$ are (typically non-linear) element-wise functions. The $W^{(n)}$ and $b^{(n)}$ are referred to as weight and bias tensors, respectively, their rank and dimensions are determined by X and depend on the implementation. They are typically randomly initialized according to a simple distribution, which we will address in a future section. Note that also more elaborate function topologies (ways of inputting the $y^{(n)}$'s to different basis functions) are possible and even common. We will discuss these in future sections but they are hard to express in the general case. Note that typically the $f_{(n)}$ have no parameters of their own, so that the W's and the b's constitute all of the parameters of the model. Since these are typically tensors of rank 2 or higher, it is common to have ANN's with millions of parameters. It is standard practice not to concern oneself with the number of parameters in the model, and one may even choose underdetermined models. This is essentially the key novelty of "machine learning" (ML) which we will briefly discuss momentarily.

The loss function depends on the implementation, but in most cases these are something simple like L_2 (in the continuous case) or cross-entropy (in the discrete case). Optimization is usually performed using some variant of stochastic gradient descent SGD). Of course symbolically computing the gradient $\nabla_\theta F(X;\theta)$ is trivial using the chain rule, but one must be careful of the computational implementation since a naive approach can lead to a combinatoric explosion of terms and hence time complexity.

1.1 Novelties of ML

The principle innovation of ML is that attempting to optimize loss functions following from models like those introduced above using stochastic gradient descent often leads to reasonable approximations of "real-world" $P(X, y)$. Understanding why this should be the case would require understanding the underlying structure of the distributions being fit. This is a daunting task since ML is effective for a wide variety of different regression and classification tasks. A small minority of the literature pursues this question [1, 2].

Even from an optimistic perspective one might expect the gradient descent to become stuck in one of a huge number of (spurious) local minima. There are a number of reasons why this problem is not prohibitive in practice. For one, the parameter space of $F(X;\theta)$ is typically of very high dimensionality ($\gtrsim 10^6$) so the vast majority of critical points are saddle points, where of course SGD does not get stuck. The second, and much more surprising reason is that typically the structure of the loss function is such that many of the local minima in the space actually correspond to excellent generalizations of the data (i.e. functions which are a good fit to $P(X, y)$. This is quite surprising, and indeed nobody currently understands why this should be the case in such a wide variety of different applications.

It also bears mentioning that the presence of so many *desirable* local minima (i.e. that correspond to good generalizations, typically found in the vicinity of the true minimum) makes it possible to train underdetermined models without severely overfitting them. This may seem remarkable a priori, but follows naturally from the above surprising facts. Note however that even though this is the case, often methods of preventing overfitting are employed (even when this involves simply discriminating between local minima) and will be discussed in a subsequent section.

2 Backpropogation

As discussed, these models are typically fit using some form of SGD. Differentiating the neural network $F(X; W, b)$ is straightforward, but in practice one should be careful to make sure the implementation remains computationally efficient. The standard implementation for computing derivatives of neural networks is known as back-propagation. Some new notation will be needed to simplify this. Throughout we will use Einstein notation. Let the output of layer ℓ be

$$o_i^{(\ell)} = f_{(\ell)}(w_{ij}^{(\ell)} o_j^{(\ell-1)}) \tag{2}$$

where we have absorbed the bias into the weights $w_{ij}^{(\ell)}$ (requiring the last component of each layer to always be 1). Then, quite trivially we have

$$\frac{\partial o_k^{(\ell)}}{\partial w_{ij}^{(\ell)}} = f'_{(\ell)} \delta_i^k o_j^{(\ell-1)} \tag{3}$$

where f' is the first derivative of f with respect to its argument and $o_j^{(1)} = x_j$ is the input.

We can use this to compute the gradient of an arbitrary loss function

$$\frac{\partial L}{\partial w_{ij}^{(\ell)}} = \frac{\partial L}{\partial o_k^{(\ell)}} \frac{\partial o_k^{(\ell)}}{\partial w_{ij}^{(\ell)}} = \frac{\partial L}{\partial o_i^{(\ell)}} f'_{(\ell)} o_j^{(\ell-1)} \tag{4}$$

Note also that

$$\frac{\partial L}{\partial o_i^{(\ell)}} = \frac{\partial L}{\partial o_{k_1}^{(M)}} \frac{\partial o_{k_1}^{(M)}}{\partial o_{k_2}^{(M-1)}} \frac{\partial o_{k_2}^{(M-1)}}{\partial o_{k_3}^{(M-2)}} \cdots \frac{\partial o_{k_n}^{(\ell+1)}}{\partial o_i^{(\ell)}} \tag{5}$$

where M is the total number of layers. At this point there are a few observations that simplify things considerably. For one, we could have written the above recursively

$$\frac{\partial L}{\partial o_i^{(\ell)}} = \frac{\partial L}{\partial o_k^{(\ell+1)}} \frac{\partial o_k^{(\ell+1)}}{\partial o_i^{(\ell)}} \tag{6}$$

We could write similar equations for any ℓ. So, once we have a particular $\frac{\partial L}{\partial o}$, we can use it in our computation of the rest rather than re-calculating $\frac{\partial o_i}{\partial o_j}$ every time. Furthermore, each of the intermediate derivatives takes an extremely simple form

$$\frac{\partial o_k^{(\ell+1)}}{\partial o_i^{(\ell)}} = f'_{(\ell+1)} w_{ki}^{(\ell+1)} \tag{7}$$

Then, for neural networks with trivial topologies, backpropagation reduces to taking a simple matrix product. Schematically, we have

$$\frac{\partial L}{\partial W^{(\ell)}} \sim (f')^{M-L} W^{(M)} \cdot W^{(M-1)} \cdots W^{(\ell+1)} \cdot W^{(\ell)} \cdot o^{(\ell-1)} \tag{8}$$

Of course Eq. 8 is only a rough cartoon since the factors of f' are all different (and evaluated at different points) but it makes obvious some salient features of neural network gradients.

References

[1] P. Mehta and D. J. Schwab, "An exact mapping between the Variational Renormalization Group and Deep Learning," *ArXiv e-prints* (Oct., 2014) , `arXiv:1410.3831 [stat.ML]`.

[2] H. W. Lin, M. Tegmark, and D. Rolnick, "Why does deep and cheap learning work so well?," *ArXiv e-prints* (Aug., 2016) , `arXiv:1608.08225 [cond-mat.dis-nn]`.