Deep Neural Networks

Contents

References

- Chapters 6, 7, and 8 of https://www.deeplearningbook.org/
- These notes.

These notes are not exhaustive. They merely provide a summary. Please consult the book chapters for the complete details.

Deep neural networks as function approximators

Deep neural networks (DNN) are function approximators that express information in a hierarchical, layered fashion. They can be used to approximate a function of d inputs to q outputs using some parameters θ. We, typicaly write $\mathbf{y} = f(\mathbf{x};\theta)$. Here $f(\mathbf{x};\theta)$ is the DNN and θ are its parameters. Both these concepts will be clarified below.

Mathematically, deep neural networks can be expressed as compositions of simpler one-layer neural neteworks:

$$f(\mathbf{x};\theta) = (f_L \circ f_{L-1} \circ \cdots \circ f_1)(\mathbf{x}).$$

In the simplest setting, the layers f_ls are a composition of an elementwise nonlinearity with a linear transformation:

$$f_i(\mathbf{z}) = h^{(i)}(\mathbf{W}^{(i)}\mathbf{z} + \mathbf{b}^{(i)}),$$

where, $\mathbf{W}^{(i)}$ is a matrix of parameters, $\mathbf{b}^{(i)}$ is a vector of parameters, and $h^{(i)}$ is a nonlinear function applied in an elementwise fashion (i.e., applied separately to each one of the inputs that are provided to it). A DNN with this structure is called a *fully-connected* DNN.

In deep learning parlance the matrix $\mathbf{W}^{(i)}$ is referred to as a *weight* matrix, the vector $\mathbf{b}^{(i)}$ is referred to as a *bias*. The function $h^{(i)}(\cdot)$ is called the *activation* function. It is typical for all but the last layer of a DNN to have the same activation function.

At the final layer, the dimensionality of the output and the choice of the activation function are dictated by constraints on the final output of the function f. For example:

1. If the output from f is a real number with no constraints, the output diemnsions is $d_L = 1$ and $h^{(L)}(\mathbf{z}) = 1$.
2. If the output from f is a positive real, $n^{(L)} = q = 1$ and $\sigma_L(x) = \exp(x)$.
3. If the output from f is a probability mass function on K categories, $n^{(L)} = q = K$ and $h^{(L)}(\mathbf{z}) = \frac{\exp(z_i)}{\sum_{j=1}^{K}\exp(z_j)}, i = 1, 2, \ldots, K$. Don't try to memorize this. We will revisit it in the next lecture.

Different ways of constructing the compositional structure of f lead to different *architectures* such as fully connected networks (shown above), *recurrent neural networks*, *convolutional neural networks*, *autoencoders*, *residual networks* etc.

Activation functions

The most common activation functions include the rectified Linear Units or ReLU (and variants), sigmoid functions, hyperolic tangents, sinusoids, step functions etc. We will visualize them in the hands-on activity.

Universal theorem for neural networks

The universal approximation theorem guarantees that DNNs are really good function approximators. In plain English, the (original) theorem states that if you take any decent activation function and build with it a dense neural network you can approximate any continuous function (defined on a compact input domain) arbitrarily well if you keep increasing the number of neurons you use. Recently, researchers have proven similar theorems for deep neural networks. In general, you can rest assured that if you grow your network by adding neurons and layers it can approximate pretty much anything you may need. That's one of the reasons why deep neural networks have been (re)gaining momentum recently.

Training regression networks - Loss function view

Assume that you want to solve a regression problem. You have input data:

$$\mathbf{x}_{1:n} = (\mathbf{x}_1, \ldots, \mathbf{x}_n),$$

and output data:

$$\mathbf{y}_{1:n} = (y_1, \ldots, y_n).$$

You want to use them to find the map between $\mathbf{x}$ and y using DNNs.

Well, you start by using a DNN $y = f(\mathbf{x}; \theta)$ to represent the map from $\mathbf{x}$ to y. Here θ are the parameters of the network (the weights and biases of all the layrs). Your problem is to fit θ to the available data.

The simplest way forward is to follow a least-squares approach. First, define a, so-called, loss function:

$$L(\theta) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(\mathbf{x}_i; \theta))^2.$$

This loss function is the sum of the squares of the prediction error of the DNN for a given θ. Once you have the loss function, you can fit θ by minimizing it:

$$\theta^* = \arg\min L(\theta).$$

However, this minimization problem does not have an analytical solution. Neither does it have a unique solution. It is a non-linear, non-convex optimization problem. It requries special treatement. We will talk about it in a while.

Training regression networks - Probabilistic view

Sometimes is it not clear how to come up with loss functions. In such situations we can employ a probabilistic view. We need to come up with a likelihood function that helps us connect the model to the observed data. So, in general, we need to come up with: $p(y_{1:n}|\mathbf{x}_{1:n}, \theta)$. Then we can fit the parameters by maximizing the log-likelihood, which is the same as minimizing the "loss" function:

$$L(\theta) = -\log p(y_{1:n}|\mathbf{x}_{1:n}, \theta).$$

This approach is going to give you the same thing as a the classic approach under the following assumptions:

- the observations are independent (conditional on the model)
- the measurement noise is Gaussian with mean given by the DNN and a constant variance.

Let's show this. Take:

\begin{split} p(y_i|\mathbf{x}_i,\theta) &= N(y_i | f(\mathbf{x}_i;\theta), \sigma^2)\ &= \frac{1}{\sqrt{2\pi}\sigma}\exp\left{-\frac{\left(y_i-f(\mathbf{x}_i;\theta)\right)^2}{2\sigma^2}\right}, \end{split}

where σ^2 is the measurement noise variance. Then, from independence, we have:

$$p(y_{1:n}|\mathbf{x}_{1:n}, \theta) = \prod^{n} p(y_i|\mathbf{x}_i, \theta).$$

$i=1$

So, we should be minimizing:

\begin{split} L'(\theta) &= -\log p(y_{1:n}|\mathbf{x}*{1:n},\theta)\ &= -\sum*{i=1}^n\log p(y_i|\mathbf{x}*i,\theta)\ &= \frac{1}{2\sigma^2}\sum*{i=1}^n\left(y_i-f(\mathbf{x}_i;\theta)\right)^2 + \text{const}. \end{split}

Well, that's the same (up to an additive constant) as the $L(\theta)$ we had before. The benefit of the probabilistic approach is that it allows you to be more flexible with the way you model the measurement process.

The minimization problem as a stochastic optimization problem

As I mentioned earlier, $L(\theta)$ is non-linear and non-convex. Classic, gradient-based, optimization techniques do not work well on it. They tend to get trapped in bad local minima. Adding a little bit of stochasticity in the optimization algorithm helps it avoid these bad local minima. Such *stochastic optimization algorithms* are still finding local minima, but they are better ones!

Another potential problem is that $L(\theta)$ may involve a summation over millions of observations (in the case of big data). In this regime, gradient-based optimization algorithms are also computationally inefficient. Stochastic optimization algorithms subsample the available data allowing you to break them down into computationally digestible *batches*.

Let's first say what a stochastic optimization problem is. Then we are going to show how we can recast a typical $\min L(\theta)$ problema as a stochastic optimization problem. A stochastic optimization problem, is a problem of the form:

$$\min_\theta \mathbb{E}_Z[\ell(\theta;Z)],$$

where $\ell(\theta;Z)$ is some scalar function of θ and the random vector Z. The expectation is over Z. Basically, you just want to minimize the expectation over Z of $\ell(\theta;Z)$. That's it.

Okay. Back to our original problem. Take:

$$L(\theta) = \frac{1}{n}\sum_{i=1}^n (y_i - f(\mathbf{x}_i;\theta))^2.$$

We need to write this as an expectation of something. But an expectation of what? Well, it is going to be an expectation over randomly selected batches of the observed data. This is by no means the only choice. But it is a very useful choice. Let's see how we can do this.

First, let's visit the observations one by one. Take I to be a Categorical random variable that picks with equal probability the index of one of the n observations, i.e.,

$$I \sim \text{Categorical}\left(\frac{1}{n},\ldots,\frac{1}{n}\right).$$

Take:

$$\ell(\theta;I) = (y_I - f(\mathbf{x}_I;\theta))^2.$$

So, here $Z = I$. Let's take the expectation over I and see what it is going to give us:

\begin{split} \mathbb{E}*I[\ell(\theta;I)] &= \sum*{i=1}^np(I=i)\ell(\theta;i)\ &= \sum_{i=1}^n\frac{1}{n}\left(y_i-f(\mathbf{x}*i;\theta)\right)^2\ &= \frac{1}{n}\sum*{i=1}^n\left(y_i-f(\mathbf{x}*i;\theta)\right)^2\ &= L(\theta). \end{split}

Great! Minimizing $L(\theta)$ is the same as minimizing $\mathbb{E}_I[\ell(\theta;I)]$.

Let's now do it again, but using an m-sized randomly selected batch from the observed data. Take $I_1, I_2, \ldots, I_m$ to be independent and identically distributed Categoricals that pick with equal probability an index from 1 to n. Then define:

$$\ell_m(\theta;I_{1:m}) = \frac{1}{m}\sum_{j=1}^m \left(y_{I_j} - f(\mathbf{x}_{I_j};\theta)\right)^2.$$

So, here $Z = (I_1,\ldots,I_m)$. Now take the expectation of this over the I's:

\begin{split} \mathbb{E}[\ell_m(\theta;I_{1:m})] &= \mathbb{E}\left[\frac{1}{m}\sum_{j=1}^m\left(y_{I_j}-f(\mathbf{x}*{I_j};\theta)\right)^2\right]\ &= \frac{1}{m}\sum*{j=1}^m\mathbb{E}\left[\left(y_{I_j}-f(\mathbf{x}*{I_j};\theta)\right)^2\right]\ &= \frac{1}{m}\sum*{j=1}^m L(\theta)\ &= \frac{m}{m}L(\theta)\ &= L(\theta), \end{split}

where we have used that $\mathbb{E}\left[\left(y_{I_j} - f(\mathbf{x}_{I_j};\theta)\right)^2\right] = L(\theta)$ since it follows from our previous analysis. Therefore, minimizing $L(\theta)$ is the same as minimizing the expectation of $\ell_m(\theta; I_{1:m})$.

The Robbins-Monro algorithm

We reached the point where we can discuss the simplest variant of a stochastic optimization algorithm. It is known as the *stochastic gradient descent* or the Robbins-Monro algorithm. It goes as follows. Take the stochastic optimization problem:

$$\min_\theta \mathbb{E}_Z[\ell(\theta; Z)].$$

And consider the RM algorithm:

- initialize θ to θ_0
- Iterate:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta \ell(\theta_t, z_t),$$

where z_t are independent samples of Z.

In this algorithm, θ_t is gradually evolved following a noisy gradient signal. The sequence α_t is known as the *learning rate* and it is our choice. The Robbins-Monro theorem gaurantees that the RM algorithm converges to a local minimum of the expectation $\mathbb{E}[\ell(\theta, Z)]$ if the learning rate satisfies the following properties:

$$\sum_{t=1}^\infty \alpha_t = +\infty,$$

and

$$\sum_{t=1}^\infty \alpha_t^2 < +\infty.$$

Intuitively, these properties say that the learning rate should converge to zero (this is an implication of the convergence of the second series) but not too fast (an implication of the divergence of the first series). There are many sequencies of learning rates that satisfy these constraints. Here is a very commonly used one:

$$\alpha_t = \frac{A}{(Bt + C)^\rho},$$

with ρ a number between 0.5 and 1 (exclusive).

Application of the Robbins-Monro algorithm to training regression networks

The algorithm for training regression networks becomes:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla_\theta \frac{1}{m}\sum_{j=1}^m \left(y_{i_{tj}} - f(\mathbf{x}_{i_{tj}};\theta_t)\right)^2,$$

where $i_{t1}, \ldots, i_{tm}$ are randomly selected indices of the observation data. Using properties of the gradient, you can also write this as:

$$\theta_{t+1} = \theta_t - 2\alpha_t \frac{1}{m}\sum_{j=1}^m \left(y_{i_{tj}} - f(\mathbf{x}_{i_{tj}};\theta_t)\right)\nabla_\theta f(\mathbf{x}_{i_{tj}};\theta_t).$$

That's pretty much it...

Notice that to carry out the algorithm, we need to $\nabla_\theta f(\mathbf{x}_{i_{tj}};\theta_t)$, i.e., the gradient of the neural network output with respect to the parameters (weights and biases). This is done using the chain rule. The algorithm is known as the back-propagation algorithm. We are not going to cover it. Nowadays, you don't have to worry about derivatives. Software like PyTorch, TensorFlow and JAX can find the derivatives for you. In the hands-on activity, I will introduce you to PyTorch.

Advanced variations of stochastic gradient descent

The RM algorithm is the simplest stochastic optimization algorithm that I could explain in a lecture. It works, but it is not the most commonly used. There are more powerful algorithms like *stochastic gradient descent with momentum*, *AdaGrad*, or *Adam* (adaptive moment estimation). I will show you in the hands-on activities how you can use these algorithms as implemented in PyTorch, but I am not going to explain their details. If you want to know the details, please read Chapter 8 of the deep learning book referenced at the very beginning of this document.

By Ilias Bilionis (ibilion[at]purdue.edu)