Deep Learning Book: summary

May 5, 2020

Chapter 6: Deep FeedForward Networks

The goal is to approximate some function f^*. A ffn defines a mapping $\boldsymbol{y} = f(\boldsymbol{x};\boldsymbol{\theta})$and learns the values of the parameters $\boldsymbol{\theta}$ for the best function approximation. The **network** model is associated with a directed acyclic graph describing how several functions $f^{(i)}$ are composed together. These represente the **layers** of the network. The training examples specify what the output layer must do at each point $\boldsymbol{x}$, but not for the other layers. The learning algorithm must decide how to use these layers to best implement an approximation of f^*.

FF networks can be considered as extensions from linear models to represent nonlinear functions of $\boldsymbol{x}$. This is done by applying the linear model to $\phi(\boldsymbol{x})$, where ϕ is a nonlinear transformation. ϕ provides a new representation or set of features for $\boldsymbol{x}$. In DL, the strategy is to learn ϕ, with a model $y = f(\boldsymbol{x};\boldsymbol{\theta},\boldsymbol{w}) = \phi(\boldsymbol{x};\boldsymbol{\theta})^\dagger \boldsymbol{w}$; the parameters $\boldsymbol{\theta}$ are used to learn ϕ from a broad class of functions, and $\boldsymbol{w}$ map $\phi(\boldsymbol{x})$ to the output. This approach gives up on the convexity of the training problem but the benefits outweigh the harms.

0.1 Example: Learning XOR

We want a ff network to learn the XOR function, which maps $\boldsymbol{X} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\} \longmapsto \{0,1,1,0\}$. Just for this example, we treat the problem as regression with a cost function

$$J(\boldsymbol{\theta}) = \frac{1}{4}\sum_{\boldsymbol{x}\in\boldsymbol{X}} \left(f^*(\boldsymbol{x}) - f(\boldsymbol{x};\boldsymbol{\theta})\right)^2 \text{ (MSE)}$$

A linear model sharply fails, so a solution is to use a model that learns a different feature space in which a linear model can represent a solution.

We introduce a ff network with one hidden layer with two units. Thus, the model is given by $y = f^{(2)}(\boldsymbol{h};\boldsymbol{w},b)$, with the hidden units $\boldsymbol{h} = f^{(1)}(\boldsymbol{x};\boldsymbol{W},\boldsymbol{c})$. The output is a linear regression, applied to $\boldsymbol{h}$: $y = \boldsymbol{h}^\dagger\boldsymbol{w} + b$. If $f^{(1)}$ were a linear transformation, the network as a whole would remain linear, so we must use a nonlinear function to describe the features. Most nn use a learned affine transformation followed by a fixed nonlinear function (*activation*). Thus, $\boldsymbol{h} = g\left(\boldsymbol{W}^\dagger\boldsymbol{x} + \boldsymbol{c}\right)$; the activation is applied element-wise and the default choice is the ReLU. Because the function is nearly linear, it preserves many of the properties that make linear models easy to optimize with gradient-based methods.

0.2 Gradient-based learning

Unlike linear, logistic or SVM models, the nonlinearity of a nn causes loss functions to become non-convex. SGD applied to these functions has no convergence guarantee and is sensitive to parameter initial values. For ffnn, it is important to initialize all weights to small random numbers (biases to zero or small positive values).

Linear models & SVMs may be trained with SGD for large datasets; in this view, nn training is not much different. Computing the gradient is more complex but can be done exactly.

0.2.1 Cost functions

In most cases, the parametric model defines a distribution $p\left(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}\right)$ and we use maximum likelihood, thus making the cost function as the cross-entropy (see section 5.5) between training data & predictions. The total cost in training will often combine the primary cost and a regularization term.

0.2.1.1 Learning conditional distributions with Maximum Likelihood

In such conditions, the cost function is

$$J\left(\boldsymbol{\theta}\right) = -\mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim p_{\text{data}}} \underbrace{\log p_{\text{model}}\left(\boldsymbol{y} \mid \boldsymbol{x}\right)}_{\text{log likelihood}}$$

Its specific form depends on the model (by p_{model}). The expression may yield terms that do not depend on model parameters and can thus be discarded. For example, considering regression (see section 5.5.1), with $p_{\text{model}}\left(\boldsymbol{y} \mid \boldsymbol{x}\right) = \mathcal{N}\left(\boldsymbol{y}; f\left(\boldsymbol{x}; \boldsymbol{\theta}\right), \boldsymbol{I}\right)$, we recover the MSE cost

$$J\left(\boldsymbol{\theta}\right) = \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim p_{\text{data}}} \left\|\boldsymbol{y} - f\left(\boldsymbol{x}; \boldsymbol{\theta}\right)\right\|^2 + \text{const}\left(\boldsymbol{I}\right)$$

In nn design, the cost gradient should be large & predictable. Functions that saturate (*i.e.* activation) undermine this as they make the gradient small; the negative log-likelihood helps avoid this problem (saturation is usually given by an exponential, so the log undoes that)

The cross-entropy cost in MLE usually does not have a minimum (*i.e.* output probability cannot reach 0 or 1, but can come arbitrarily close to), This is solved by regularization techniques.

0.2.1.2 Learning conditional statistics

Instead of learning $p\left(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta}\right)$, we often want to learn just one conditional statistic of $\boldsymbol{y}$, given $\boldsymbol{x}$.

If we regard the nn being able to represent a large class of functions (instead of a parametric model with $\boldsymbol{\theta}$), and the cost as a functional over such space, we may optimize it with calculus of variations. By solving for

$$f^* = \underset{f}{\text{argmin}}\, \mathbb{E}_{\boldsymbol{x},\boldsymbol{y}\sim p_{\text{data}}} \left\|\boldsymbol{y} - f\left(\boldsymbol{x}\right)\right\|^2$$

yields

$$f^* = \mathbb{E}_{\boldsymbol{y}\sim p_{\text{data}}\left(\boldsymbol{y} \mid \boldsymbol{x}\right)}\left[\boldsymbol{y}\right]$$

i.e. this cost function predicts the mean of $\boldsymbol{y}$ for each $\boldsymbol{x}$. Considering the MAE cost, we obtain a predictor for the median.

0.2.2 Output units

The choice of how to represent the output determines the form of the cross-entropy. We suppose the ffnn provides $\boldsymbol{h} = f\left(\boldsymbol{x}; \boldsymbol{\theta}\right)$ and the role of the output layer is to transform these features for the task.

0.2.2.1 Linear units for gaussian output distributions (*i.e.* regression)

Affine transformation with nonlinearity: $\hat{\boldsymbol{y}} = \boldsymbol{W}^{\dagger}\boldsymbol{h} + \boldsymbol{b}$. These are used for the mean of a conditional gaussian $p\left(\boldsymbol{y} \,|\, \boldsymbol{x}\right) = \mathcal{N}\left(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I}\right)$. As we have seen, maximizing the log-likelihood is equivalent to minimizing the MSE. Also, their unbounded image goes well with GD.

0.2.2.2 Sigmoid for Bernoulli output distribution (*i.e.* binary classification)

We need to predict $P\left(y = 1 \,|\, \boldsymbol{x}\right)$. Using ReLU wouldn't work, as the gradient may be null when the training example corresponds to the negative class; a sigmoid unit is instead used: $z = \boldsymbol{h}^{\dagger}\boldsymbol{w} + b$, $\hat{y} = \sigma\left(z\right)$. We can obtain this by assuming the log likelihood is linear in the output y and z. Then, normalizing: $P\left(y\right) = \sigma\left[\left(2y - 1\right)z\right]$. Thus, the MLE cost function is

$$\begin{aligned} J\left(\boldsymbol{\theta}\right) &= \log P\left(y \mid \boldsymbol{x}\right) = -\log \sigma\left[\left(2y-1\right)z\right] \\ &= \zeta\left[\left(1-2y\right)z\right] \text{ (softplus)} \end{aligned}$$

This saturates only when $\left(1 - 2y\right)z \ll 0$ [that is, when the model already knows the correct answer for that instance]. If z has the wrong sign, then $\left(1 - 2y\right)z = \left|z\right|$, and the asymptote derivative is sign $\left(z\right)$, so the gradient does not shrink and the error can be corrected for quickly.

With other cost functions than MLE cross-entropy, such as MSE, the loss can saturate with $\sigma\left(z\right)$, thus shrinking the gradient, regardless of the actual answer. MLE is thus preferred.