Deep Learning Book: summary

May 7, 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 $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ 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

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

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 $p_{\text{model}}(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x}; \boldsymbol{\theta}), \boldsymbol{I})$, we recover the MSE cost

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

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 $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$, 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

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

yields

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

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 $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$ 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 $p(\boldsymbol{y} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$. 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 $P(y = 1 \mid \boldsymbol{x})$. 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: $z = \boldsymbol{h}^\dagger \boldsymbol{w} + b$, $\hat{y} = \sigma(z)$. We can obtain this by assuming the log likelihood is linear in the output y and z. Then, normalizing: $P(y) = \sigma[(2y-1)z]$. Thus, the MLE cost function is

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

This saturates only when $(1-2y)z \ll 0$ [that is, when the model already knows the correct answer for that instance]. If z has the wrong sign, then $(1-2y)z = |z|$, and the asymptote derivative is sign (z), 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 $\sigma(z)$, thus shrinking the gradient, regardless of the actual answer. MLE is thus preferred.

0.2.2.3 Softmax for Multinoulli output distribution (*i.e.* multiclass classification)

To represent the probability distribution over a discrete variable with n possible values, we may use softmax.

We now need $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i \mid \boldsymbol{x})$ and also $\sum_i \hat{y}_i = 1$. We generalize the approach for the Bernoulli distribution: $\boldsymbol{z} = \boldsymbol{W}^\dagger \boldsymbol{h} + \boldsymbol{b}$, where $\boldsymbol{z} \propto \log P(y = i \mid \boldsymbol{x})$. Then, normalizing:

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

The log-likelihood is then

$$\log \text{softmax}(\boldsymbol{z})_i = z_i - \log \sum_j \exp(z_j)$$

the first term can never saturate, thus learning can proceed. The second term is $\sim \max_j z_j$; the intuition is that the cost always strongly penalizes the most active incorrect prediction. If the correct answer already has the largest z_i, the cost $\sim \max_j z_j - z_i \simeq 0$; this examples will contribute little to the cost, which will be dominated by the incorrect ones. Unregularized MLE will drive the model to predict

$$\text{softmax}(\boldsymbol{z}(\boldsymbol{x}; \boldsymbol{\theta}))_i = \frac{\#\text{instances with } \boldsymbol{x}^{(j)} = \boldsymbol{x} \text{ and } \boldsymbol{y}^{(j)} = i}{\#\text{instances with } \boldsymbol{x}^{(j)} = \boldsymbol{x}} \Bigg\} \to \begin{cases} \text{proportion of class } i \text{ from} \\ \text{instances with same } \boldsymbol{x} \end{cases}$$

As MLE is consistent, this will happen as ling as the model family is capable of representing the target distribution. In practice, the model will only approximate these factors.

Other cost functions than MLE don't work as well with softmax. The log from log-likelihood can prevent the exponential saturation, which diminishes the gradient. This saturation can occur even in incorrect predictions (when $z_i = \max_j z_j$ and $z_i \gg z_j \ j \neq i$). This means the model can make highly confident incorrect predictions with little loss.

We can think of the softmax as creatin competition between its inputs (*i.e.* increase in one output decreases others). The function is actually a softened version of argmax $(\boldsymbol{z})$ *(softargmax)*.

0.3 Hidden Units

The design choices of hidden units is an active area of research without many definitive guiding theoretical principles. ReLUs are an excellent default choice; the jump in their derivatives at $z = 0$ is not a problem in practice, in part because training does not usually arrive at a local minimum of cost, but rather reduces its value significantly.

0.3.1 ReLUs & their generalizations

ReLUs are easy to optimize as they are similar to linear units: their gradients are large and consistent. When initializing parameters, it can be good practice to set bias to a small positive number $b \sim 0.1$, thus making units initially active for most inputs and allowing gradients to pass through.

Some generalizations exist to address the problem that they cannot learn on examples for which their activation is zero. A **leaky ReLU** has an α_i slope when $z_i < 0$, fixed to a small value like 0.01. A **parametric ReLU** treats α_i as a learnable parameter.

Maxout units

ReLUs also help with recurrent nn, as linear activation facilitates the propagation of information through several time steps.

0.3.2 Logistic sigmoid & tanh

These were mostly used prior to ReLUs. Unlike the latter, these saturate across most of their domain; thus their use as hidden units in ffnn is now discouraged. They are more common in recurrent nn and autoencoders.

0.3.3 Other hidden units

A wide variety of differentiable functions perform perfectly well. New hidden unit types are now published only if they provide a clear improvement; otherwise they are so common as to be uninteresting.

Softplus is generally discouraged, as it empirically shows worse performance than the rectifier.

0.4 Architecture design

In layer-based ffnn, the main choices are the depth of the network and width of each layer. Deeper networks often use fewer units per layer and total parameters, and often generalize to the test set, but are also harder to optimize. The ideal architecture for a task must be found via experimentation.

0.4.1 Universal approximation properties & Depth

The **UA theorem** states that a ffnn with at least one hidden layer can approximate any Borel measurable function (on spaces of finite dimension) with any desired non-zero amount of error, given enough hidden units. However, the theorem states the nn will be able to *represent* this function, that it can be *learned* thru training is not guaranteed. Moreover, the number of hidden units needed, in worst case, is exponentially bounded.

There exist families of functions which can be efficiently approximated by a network with depth $> d$, but require a much larger model if using a lesser depth.
There are also statistical motivations for choosing a deep model. This choice encodes a general belief that the function to learn involves the composition of several simpler functions. This can be interpreted from a representation point of view that the learning problem consists of discovering a set of underlying features which in turn can be described in terms of simpler underlying features. Empirically, greater depth does seem to result in better generalization for many tasks (deeps archs. as a useful prior).

0.4.2 Other Architectural considerations

Another consideration is exactly hoe to connect a pair of layers to each other. Instead of the default fully connected, specialized networks (*i.e.* convolutional) have fewer connections. These strategies reduce the number of parameters but are highly problem-dependent.

0.5 Back-Propagation and other differentiation algorithms

The term backpropagation refers only to the method fot computing the gradient (with a simple and inexpensive procedure), which is used in SGD to perform learning. The gradient we most often require is that of the cost function with respect to the parameters, $\boldsymbol{\nabla}_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. BP can also be applied to other tasks that involve computing derivatives.

0.5.1 Computational graphs

We use each node to describe a variable. An *operation* is a simple function of one or more variables. We specify a set of allowable operations, with more complicated functions describable by composition [without loss of generality, we will work here with operations with a single return variable]. If a variable y results from the output of an operation applied to x. We draw a directed edge from x to y, labeled by the operation.

0.5.2 Chain rule of calculus

BP computes the chain rule, with a specific order of operations that is highly efficient.

$$\frac{\partial z}{\partial x_i} = \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \text{ or } \boldsymbol{\nabla}_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^{\dagger} \boldsymbol{\nabla}_{\boldsymbol{y}} z$$

This will usually be applied to tensors. We denote the gradient of z with respect to a tensor X as $\boldsymbol{\nabla}_{\mathsf{X}} z$;

we can use a single index i to represent the full tuple of indices, thus representing $(\nabla_{\mathsf{X}} z)_i = \dfrac{\partial z}{\partial \mathsf{X}_i}$. We can then write the chain rule as

$$\nabla_{\mathsf{X}} z = (\nabla_{\mathsf{X}} \mathsf{Y}_j) \frac{\partial z}{\partial \mathsf{Y}_j}$$

0.5.3 Recursively applying the chain rule to obtain backpropagation

Actually computing the gradient with the chain rule includes many subexpressions that may be repeated several times within the overall expression of the gradient. For complicated graphs, there can be exponentially many repetitions, thus creating some necessity to store them.

We now consider BP that specifies the actual gradient computation, but recursively applying the chain rule.

We consider a computational graph to compute $u^{(n)}$ (*e.g.* the loss function) with n_i inputs $u^{(i)}$ with $i = 1, ..., n_i$; we wish to compute $\dfrac{\partial u^{(n)}}{\partial u^{(i)}}$ (these inputs correspond to the model parameters).

We assume the graph lets us compute their outputs $u^{(n_I+i)}$ up to $u^{(n)}$; each associated with an operation $u^{(i)} = f^{(i)}\left(\boldsymbol{A}^{(i)}\right)$, where $\boldsymbol{A}^{(i)} = \text{Pa}\left(u^{(i)}\right)$ is the set of parent nodes to $u^{(i)}$. The forward computation defines a graph $\mathcal{G}$ of all computed nodes. To perform BP, we can extend $\mathcal{G}$ into $\mathcal{B}$, containing all nodes. Each node in $\mathcal{B}$ computes the derivative $\dfrac{\partial u^{(n)}}{\partial u^{(i)}}$ associated with the forward node $u^{(i)}$, using the chain rule

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j\in \text{Pa}\left(u^{(i)}\right)} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$$

$\mathcal{B}$ contains exactly one edge from $u^{(j)}$ to $u^{(i)}$ of $\mathcal{G}$, associated with the computation of $\dfrac{\partial u^{(i)}}{\partial u^{(j)}}$. A dot product is performed for each node, between the gradient already computed for $u^{(i)}$, children of $u^{(j)}$, and the vector of partial derivatives of $u^{(j)}$ with respect to its children $u^{(i)}$. The amount of computation scales linearly with the number of edges in $\mathcal{G}$.

By storing a gradient table, BP performs one gradient $\dfrac{\partial u^{(i)}}{\partial u^{(j)}}$ for each node, thus avoiding the exponential explosion of repeated expressions (with no regard to memory).

0.5.4 BP computation in fully-connected multilayer perceptron

We consider the specific graph of a multi-layer MLP.

Forward propagation:

$$\boldsymbol{h}^{(0)} = \boldsymbol{x}$$

$$\begin{cases} \text{for } k = 1, ..., l \text{ do} & \\ & \boldsymbol{a}^k = \boldsymbol{b}^k + \boldsymbol{W}^k \boldsymbol{h}^{k-1} \\ & \boldsymbol{h}^k = f^k\left(\boldsymbol{a}^k\right) \\ \text{end for} & \end{cases}$$

Backward propagation:

$$\boldsymbol{g} \leftarrow \boldsymbol{\nabla}_{\boldsymbol{y}} J = \boldsymbol{\nabla}_{\boldsymbol{y}} L\left(\hat{\boldsymbol{y}}, \boldsymbol{y}\right)$$

$$\begin{cases} \text{for } k = l, l-1, ..., 1 \text{ do} & \\ \text{convert gradient on output into pre-activation} & \boldsymbol{g} \leftarrow \boldsymbol{\nabla}_{\boldsymbol{a}^k} J = \boldsymbol{g} \odot f'\left(\boldsymbol{a}^k\right) \\ \text{compute gradient wrt weights and biases} & \boldsymbol{\nabla}_{\boldsymbol{b}^k} J = \boldsymbol{g} + \lambda \boldsymbol{\nabla}_{\boldsymbol{b}^k} \Omega\left(\boldsymbol{\theta}\right) \\ & \boldsymbol{\nabla}_{\boldsymbol{W}^k} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)\dagger} + \lambda \boldsymbol{\nabla}_{\boldsymbol{W}^k} \Omega\left(\boldsymbol{\theta}\right) \\ \text{propagate gradient wrt to the next layer's activationf} & \boldsymbol{g} = \boldsymbol{\nabla}_{\boldsymbol{h}^{k-1}} J = \boldsymbol{W}^{k\dagger} \boldsymbol{g} \\ \text{end for} & \end{cases}$$