1 Formalizing Deep Neural Networks

Michael Unser

1.1 Introduction

Since the various contributions presented in this book rely heavily on deep neural networks, we felt that it would be useful to include some background material on such computational structures. Our intention with this chapter is to provide a short, self-contained introduction to deep neural networks that is aimed at mathematically inclined readers. Our primary inspiration for writing it was to demonstrate the usage of a vector–matrix formalism that is well suited to the compositional structure of these networks and that facilitates the derivation and description of the backpropagation algorithm. In what follows we first develop the formalism and then present a detailed analysis of supervised learning for the two most common scenarios: (i) multivariate regression, and (ii) classification; these rely on the minimization of least squares and cross-entropy criteria, respectively. The regression setting is the most relevant one for biomedical image reconstruction; see for instance [1].

1.2 Primary Components of Neural Networks

A deep neural network (DNN) is a parameterized computational structure that implements a multidimensional map generically denoted by $\boldsymbol{f}_{\boldsymbol{\theta}} : \mathbb{R}^{N_{\text{in}}} \to \mathbb{R}^{N_{\text{out}}}$, where N_{in} and N_{out} are the dimensions of the input and output spaces, respectively. The vector $\boldsymbol{\theta}$ represents the parameters of the neural network, which are adjusted during training. A DNN results from the composition of simple computational modules: multidimensional linear (or affine) transformations and pointwise nonlinearities referred to as neuronal activations. This is often represented by a graph (see Fig. 1.1) where each node represents a neuron and where each arrow pointing to a neuron is associated with a linear (adjustable) weight.

To be more precise, a DNN is composed of L layers of neurons indexed (from left to right) by ℓ. The ℓth layer of the network has N_ℓ neurons indexed by n. In the case of a fully connected DNN, any given neuron (ℓ, n) of layer ℓ has arrows originating from all neurons of level $\ell - 1$. The architecture of a fully connected DNN is therefore uniquely specified by its node descriptor $(N_0, N_1, \ldots, N_L)$, where $N_0 = N_{\text{in}}$ and $N_L = N_{\text{out}}$.

To describe the computations performed by the DNN, we denote the intermediate values in the network at the output of layer ℓ by $\mathbf{z}_\ell = (z_{\ell,1}, \ldots, z_{\ell,N_\ell})$. If the

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press

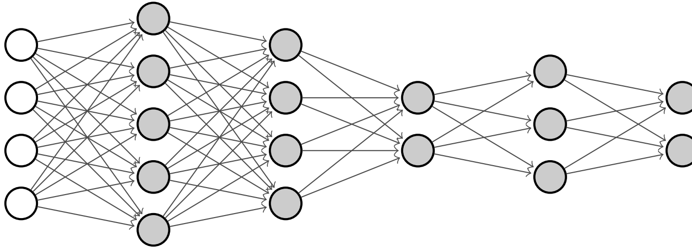

Figure 1.1 Diagram of a fully connected neural network $\mathbb{R}^4 \to \mathbb{R}^2$ with $L = 5$ layers of active neurons (gray circles) and node descriptor $(4, 5, 4, 2, 3, 2)$.

real-valued weights associated with the arrow $(\ell - 1, m) \to (\ell, n)$ are denoted by $\theta_{\ell,n,m}$ then the computation performed at neuron (n, ℓ) (a node of the graph) is simply

$$z_{\ell,n} = \sigma\left(\theta_{\ell,n} + \sum_{m=1}^{N_{\ell-1}} \theta_{\ell,n,m} z_{\ell-1,m}\right), \tag{1.1}$$

where $\sigma : \mathbb{R} \to \mathbb{R}$ is the activation function of the neuron, while $\theta_{\ell,n} \in \mathbb{R}$ is the bias parameter.

1.2.1 Vectorial Representation of a Deep Neural Network

To make the compositional structure of the DNN explicit, we shall now adopt an equivalent vectorial description. To that end, we collect the linear weights associated with layer ℓ in the matrix $\mathbf{W}_\ell = \left[(\theta_{\ell,n,m})_{n=1}^{N_\ell}, m = 1, \ldots, N_{\ell-1}\right] \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$ and the biases in the vector $\mathbf{b}_\ell = (\theta_{\ell,n})_{n=1}^{N_\ell}$. Likewise, we denote the response of the nth neuron in layer ℓ by $\sigma_{\ell,n} : \mathbb{R} \to \mathbb{R}$. This then allows us to implement the DNN with Algorithm 1.1, which is sequential.

Algorithm 1.1 Feedforward DNN

1. Input: $\mathbf{z}_0 = \mathbf{x} \in \mathbb{R}^{N_0}$.
2. Layer-to-layer propagation with intermediate input variable $\mathbf{z}_{\ell-1} \in \mathbb{R}^{N_{\ell-1}}$, output variables $\mathbf{u}_\ell, \mathbf{z}_\ell \in \mathbb{R}^{N_\ell}$, linear network parameters $\mathbf{W}_\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}$, and $\mathbf{b}_\ell \in \mathbb{R}^{N_\ell}$. For $\ell = 1, \ldots, L$, compute

$$\mathbf{u}_\ell = \mathbf{W}_\ell \mathbf{z}_{\ell-1} + \mathbf{b}_\ell, \tag{1.2}$$

$$\mathbf{z}_\ell = \boldsymbol{\sigma}_\ell(\mathbf{u}_\ell), \tag{1.3}$$

   where the vector-valued function $\boldsymbol{\sigma}_\ell = (\sigma_{\ell,1}, \ldots, \sigma_{\ell,N_\ell}) : \mathbb{R}^{N_\ell} \to \mathbb{R}^{N_\ell}$ provides a concise representation of the pointwise nonlinearities.
3. Output: $\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}) = \mathbf{z}_L \in \mathbb{R}^{N_L}$.

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press

The first processing step described by Eq. (1.2) is the part symbolized by the arrows in Fig. 1.1, which connect one layer of the network to the next. It takes the output $\mathbf{z}_{\ell-1} \in \mathbb{R}^{N_{\ell-1}}$ of the $(\ell - 1)$th layer and applies a linear transformation followed by the addition of a constant vector $\mathbf{b}_\ell$ (the bias). The linear transformation is encoded in the matrix $\mathbf{W}_\ell$ of size $N_\ell \times N_{\ell-1}$. Note that the bias can also be encoded in terms of linear weights applied to a constant input common to all layers – structurally, this is equivalent to augmenting the matrix $\mathbf{W}_\ell$ by one line and expressing $\mathbf{z}_{\ell-1}$ in homogeneous coordinates by including the (dummy) constant 1 as an additional component. Consequently, this part of the processing is intrinsically linear and parameterized by the weights $(\mathbf{W}_\ell, \mathbf{b}_\ell)$ which are learned during the training of the network.

Equation (1.3) describes the combined effect of the neuronal activations at layer ℓ, which corresponds to the nodes (gray circles) in Fig. 1.1. This module is essential since it constitutes the nonlinear part of the processing. In practice, the responses of the individual neurons are often chosen to be the same, leading to $\sigma_{\ell,n} = \sigma : \mathbb{R} \to \mathbb{R}$, with one of the most popular choices being $\sigma(x) = \mathrm{ReLU}(x) = \max(0, x)$ (a rectified linear unit).

1.3 Training

The parameters of the neural network are adjusted during the training process. This is done over the training data by minimizing some prescribed training loss with the help of a stochastic version of the steepest-descent algorithm, referred to as *stochastic gradient descent* (SGD). The latter requires the repeated calculation of the gradient of the training loss over data subsets – called batches – which are selected randomly.

As far as supervised learning is concerned, one needs to distinguish between two classes of problems. The first is *multivariate regression*, where the goal is to construct a multivariate mapping $\boldsymbol{f}_{\boldsymbol{\theta}} : \mathbb{R}^{N_0} \to \mathbb{R}^{N_L}$ such that $\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_m) \approx \mathbf{y}_m$, without overfitting, for a given set of training data $(\mathbf{x}_m, \mathbf{y}_m) \in \mathbb{R}^{N_0} \times \mathbb{R}^{N_L}$ with $m = 1 \ldots, M$. The second is *classification*, where the training data are partitioned (or labeled) into K classes $(C_1, \ldots, C_K)$ and one wishes to construct a mapping $\boldsymbol{p}_{\boldsymbol{\theta}} : \mathbb{R}^{N_0} \to [0, 1]^K$ that returns the posterior probabilities of class membership of an observed pattern $\mathbf{x} \in \mathbb{R}^{N_0}$, so that

$$\boldsymbol{p}_{\boldsymbol{\theta}}(\mathbf{x}) \approx \big(\mathrm{Prob}(C_1|\mathbf{x}), \ldots, \mathrm{Prob}(C_K|\mathbf{x})\big).$$

Here, we shall first consider the problem of (multivariate) regression, which is a refinement of classical least squares data fitting and which lends itself naturally to the derivation of the celebrated backpropagation algorithm. In Section 1.4 we then show how the underlying computational structure and training algorithm should be modified to yield a classifier.

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press

1.3.1 The Backpropagation Algorithm

The backpropagation algorithm is an efficient way to compute the partial derivatives (the gradient) of the loss function with respect to the parameters of the network. It is the workhorse of deep learning.

Appetizer: Deep Neural Network of Unit Width

To understand the procedure, it is helpful first to consider a simplified DNN with a unit width across all layers, as shown in Fig. 1.2. Such an elementary network is described recursively by the set of (scalar) equations

$$u_\ell = w_\ell z_{\ell-1} + b_\ell, \tag{1.4}$$

$$z_\ell = \sigma_\ell(u_\ell), \tag{1.5}$$

with $\ell = 1, \dots, L$, where $z_0 = x$ is the input of the network and $\sigma_\ell : \mathbb{R} \to \mathbb{R}$ is the function that describes the neuronal response at layer ℓ. Moreover, σ_ℓ is assumed to be differentiable and its derivative is denoted by σ'_ℓ. This network implements a parametric function $f_{\boldsymbol{\theta}} : \mathbb{R} \to \mathbb{R}$. The vector $\boldsymbol{\theta} = (w_1, b_1, \dots, w_L, b_L)$ collects the weights of the network, to be adjusted during training. Given a data batch $\{(x_m, y_m)\}_{m=1}^M$ and a quadratic cost

$$J(\boldsymbol{\theta}) = \sum_{m=1}^{M} J_m(\boldsymbol{\theta}) \quad \text{with} \quad J_m(\boldsymbol{\theta}) = \frac{1}{2}\big(f_{\boldsymbol{\theta}}(x_m) - y_m\big)^2, \tag{1.6}$$

the goal is now to efficiently evaluate the gradient $\nabla_{\boldsymbol{\theta}} J$ of the criterion. The gradient components are the partial derivatives $\partial J/\partial w_\ell$ and $\partial J/\partial b_\ell$ for $\ell = 1, \dots, L$. Since the cost J is additive, it is sufficient to consider the contribution to the gradient of a single data point (x_m, y_m) with associated elementary cost J_m. To that end, we first apply the neural network to x_m with the current set of parameter $\boldsymbol{\theta}$, which yields $f_{\boldsymbol{\theta}}(x_m)$ and the corresponding intermediate variables $(u_\ell, z_\ell)_{\ell=1}^N$ defined by Eqs. (1.4) and (1.5). This step is called the “forward pass.” To compute the required derivatives we then proceed backwards, starting from $\ell = L$, and propagate the differentiation within the network using the chain rule. Specifically, by substituting the equation for the last layer, we have that

$$J_m = \frac{1}{2}\big(\sigma_L(u_L) - y_m\big)^2 \quad \text{with} \quad u_L = w_L z_{L-1} + b_L \tag{1.7}$$

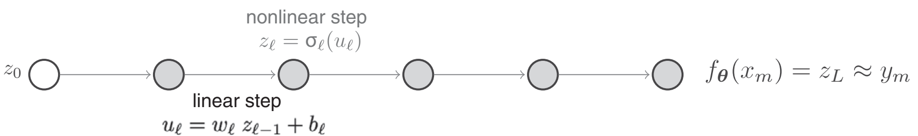

Figure 1.2 Minimal-width neural network $\mathbb{R} \to \mathbb{R}$ with $L = 5$ layers of active neurons (gray circles) and node descriptor $(1, 1, 1, 1, 1)$.

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press

which yields

$$\delta_L = \frac{\partial J_m}{\partial b_L} = \big(\sigma_L(u_L) - y_m\big)\frac{\partial \sigma_L(u_L)}{\partial u_L}\frac{\partial u_L}{\partial b_L} = \big(\sigma_L(u_L) - y_m\big)\sigma_L'(u_L) \tag{1.8}$$

$$\frac{\partial J_m}{\partial w_L} = \underbrace{\big(\sigma_L(u_L) - y_m\big)\sigma_L'(u_L)}_{\delta_L}\frac{\partial u_L}{\partial w_L} = \delta_L z_{L-1}, \tag{1.9}$$

where, for computational efficiency, we have identified the common (re-weighted) error term δ_L. By inserting $z_{L-1} = \sigma_{L-1}\big(w_{L-1}z_{L-2} + b_{L-1}\big)$ into Eq. (1.7), we then proceed with the chain rule to the next layer and obtain

$$\delta_{L-1} = \frac{\partial J_m}{\partial b_{L-1}} = \underbrace{\big(\sigma_L(u_L) - y_m\big)\sigma_L'(u_L)}_{\delta_L}\frac{\partial u_L}{\partial b_{L-1}} = \delta_L w_L \sigma_{L-1}'(u_{L-1}), \tag{1.10}$$

$$\frac{\partial J_m}{\partial w_{L-1}} = \big(\sigma_L(u_L) - y_m\big)\sigma_L'(u_L)\frac{\partial u_L}{\partial w_{L-1}} = \delta_{L-1}z_{L-2}. \tag{1.11}$$

By repeating this process for ℓ down to 1, we uncover the simple recursion

$$\delta_\ell = \frac{\partial J_m}{\partial b_\ell} = w_{\ell+1}\delta_{\ell+1}\sigma_\ell'(u_\ell), \tag{1.12}$$

$$\frac{\partial J_m}{\partial w_\ell} = \delta_\ell z_{\ell-1}, \tag{1.13}$$

which is the celebrated backpropagation algorithm. Let us also note that Eq. (1.12) is consistent with Eq. (1.8) if we set $w_{L+1} = 1$ and $\delta_{L+1} = f_{\boldsymbol{\theta}}(x_m) - y_m$. The backpropagation algorithm can therefore be interpreted as feeding the prediction error $f_{\boldsymbol{\theta}}(x_m) - y_m$ backwards in a slightly modified version of the network in Fig. 1.2, where the biases have been suppressed and the pointwise nonlinearity replaced by a simple multiplication (re-weighting) by $\sigma_\ell'(u_\ell)$. The output of every layer then yields $\delta_\ell = \partial J_m/\partial b_\ell$, which is then used to compute $\partial J_m/\partial w_\ell$.

Backpropagation in Full Generality

In practice, of course, the layers are wider so that the scalar multiplications in Eqs. (1.12) and (1.13) need to be replaced by matrix–vector multiplication.

In the interest of clarity and to highlight the parallel with the scalar scenario that has just been considered, we shall make use of the vectorial–tensor calculus formalism. Given a data batch $\{(\mathbf{x}_m, \mathbf{y}_m)\}_{m=1}^{M}$ (in the multivariate regression setting), we are now aiming at training the neural network to minimize the quadratic cost

$$J(\boldsymbol{\theta}) = \sum_{m=1}^{M} J_m(\boldsymbol{\theta}) \quad \text{with} \quad J_m(\boldsymbol{\theta}) = \frac{1}{2}\big\| \boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_m) - \mathbf{y}_m \big\|_2^2, \tag{1.14}$$

where the network parameters $\boldsymbol{\theta}$ are encoded in the weight matrices $\mathbf{W}_\ell$ and bias vectors $\mathbf{b}_\ell$ for $\ell = 1, \dots, L$.

We now focus on some particular layer ℓ and write $\mathbf{W} = \mathbf{W}_\ell$ and $\mathbf{b} = \mathbf{b}_\ell$ for better readability, while keeping all the other network parameters fixed. Under those conditions, $J(\boldsymbol{\theta}) = J(\mathbf{b}, \mathbf{W})$ is a real-valued functional that depends on the vector and matrix

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press

parameters $\mathbf{b}=(b_n)\in\mathbb{R}^N$ and $\mathbf{W}\in\mathbb{R}^{M\times N}$, with $[\mathbf{W}]_{m,n}=w_{m,n}$. It is then helpful to represent the partial derivatives of J with respect to those parameters by the following vector- and matrix-valued functions:

$$\frac{\partial J(\mathbf{b})}{\partial \mathbf{b}} = \begin{bmatrix} \partial J(\mathbf{b})/\partial b_1 \\ \vdots \\ \partial J(\mathbf{b})/\partial b_N \end{bmatrix} \tag{1.15}$$

$$\frac{\partial J(\mathbf{W})}{\partial \mathbf{W}} = \begin{bmatrix} \partial J(\mathbf{W})/\partial w_{1,1} & \dots & \partial J(\mathbf{W})/\partial w_{1,N} \\ & \vdots & \\ \partial J(\mathbf{W})/\partial w_{M,1} & \dots & \partial J(\mathbf{W})/\partial w_{M,N} \end{bmatrix}. \tag{1.16}$$

We now have all the elements needed to present the backpropagation algorithm for a generic feedforward neural network that has L layers indexed by ℓ, each of which is composed of N_ℓ neurons. Since we are dealing with partial derivatives and making use of the chain rule, it should not come as a surprise that determining this backpropagation algorithm requires a knowledge of the "derivative" map $\boldsymbol{\sigma}'_\ell = (\sigma'_{\ell,1},\dots,\sigma'_{\ell,N_\ell}) : \mathbb{R}^{N_\ell} \to \mathbb{R}^{N_\ell}$. Other than that, the sequence of computations is essentially the flow-graph transpose of Algorithm 1.1 with the recursion running backwards.

Algorithm 1.2 Gradient computation by backpropagation

1. Initialization:

$$\boldsymbol{\delta}_L = \frac{\partial J_m}{\partial \mathbf{b}_L} = \big(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_m) - \mathbf{y}_m\big) \odot \boldsymbol{\sigma}'_L(\mathbf{u}_L) \quad \in \mathbb{R}^{N_L}, \tag{1.17}$$

where the symbol $\odot$ denotes the pointwise (or Hadamard) product of two vectors.

2. Backward propagation of the error through the network: For $\ell = L-1$ down to 1, compute

$$\boldsymbol{\delta}_\ell = \frac{\partial J_m}{\partial \mathbf{b}_\ell} = \mathbf{W}^T_{\ell+1}\boldsymbol{\delta}_{\ell+1} \odot \boldsymbol{\sigma}'_\ell(\mathbf{u}_\ell) \quad \in \mathbb{R}^{N_\ell}, \tag{1.18}$$

$$\frac{\partial J_m}{\partial \mathbf{W}_\ell} = \boldsymbol{\delta}_\ell \cdot \mathbf{z}^T_{\ell-1} \quad \in \mathbb{R}^{N_\ell \times N_{\ell-1}}, \tag{1.19}$$

where (1.19) is valid for layer $\ell = L$ as well.

As in the scalar scenario, we can also extend the validity of (1.18) for $\ell = L$ by defining $\mathbf{W}_{L+1} = \mathbf{I}$ and $\boldsymbol{\delta}_{L+1} = \big(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_m) - \mathbf{y}_m\big)$. The backpropagation algorithm therefore essentially amounts to feeding the prediction error $\big(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_m) - \mathbf{y}_m\big)$ for each test datum $(\mathbf{x}_m, \mathbf{y}_m)$ backwards into the network. The only adjustment to the initial structure is that the nonlinear neuronal transformation at any given node (n,ℓ) is replaced by a pointwise multiplication of the backpropagated error with $\sigma'_{\ell,n}([\mathbf{u}_\ell]_n)$. The striking parallel between the forward and backward computations is best illustrated by the juxtaposition of Figs. 1.3 and 1.4. Practically, this translates into the determination of the gradient being essentially as fast as the application of the neural network to the

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press

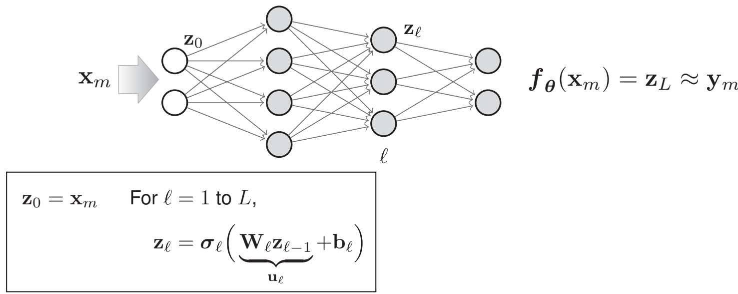

Figure 1.3 Feedforward computations in a DNN.

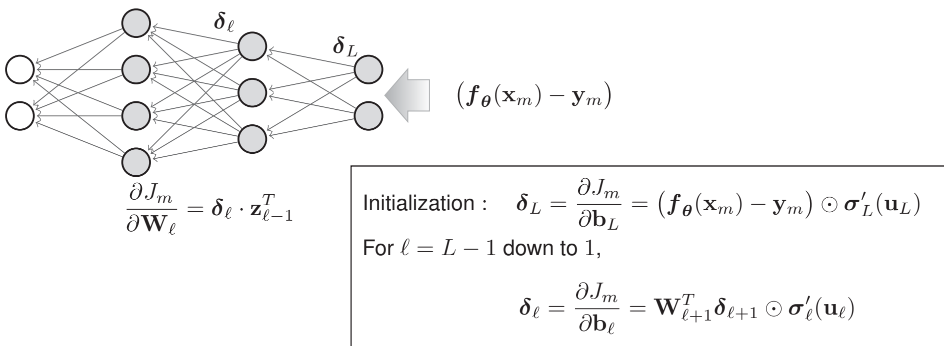

Figure 1.4 Efficient computation of the gradient of the elementary least squares term $\frac{1}{2}\|\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_m) - \mathbf{y}_m\|_2^2$ by backpropagation for the DNN of Fig. 1.3.

data. While these computations can be done very efficiently, the remaining limiting factor is the slow convergence of the gradient-descent algorithm and the necessity to process large amounts of data to ensure correct behavior. Fundamentally, it is the need for a massive number of iterations that is responsible for the large computational cost of the training of DNNs.

1.4 Categorical Loss for Classification

For cases where the DNN is being designed for a classification task, it is common to include an additional output layer that converts the real-valued output of the DNN into a set of pseudo-probabilities. This recoding is typically achieved with a softmax unit. Given the input vector $\mathbf{z} = (z_1, \ldots, z_K)$, the softmax transformation $\mathbb{R}^K \to [0,1]^K$ is defined by

$$\text{Softmax}(\mathbf{z}) = (p_1, \ldots, p_K), \quad p_k = \frac{\exp(z_k)}{\sum_{k=1}^K \exp(z_k)}, \tag{1.20}$$

the effect of which is to translate the z_k into a set of “probabilities” $\{p_k\}_{k=1}^K$, with $\sum_{k=1}^K p_k = 1$. These are intended to approximate the posterior probability distribution

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press

$\{\text{Prob}(C_k|\mathbf{x})\}_{k=1}^{K}$ of the datum $\mathbf{x}$, where C_k denotes the occurrence of the kth class. The ultimate classification output of the neural network (the hard decision) is then given by

$$k(\mathbf{x}) = \arg\max p_k = \arg\max z_k.$$

Given a representative ensemble $\big(\mathbf{x}_m, C(m)\big)_{m=1}^{M}$ of labeled instances of training data with class label $C(m) \in \{1, \ldots, K\}$, we now need to specify a suitable training criterion. The preferred choice for a categorical output is the cross-entropy

$$J = -\sum_{m=1}^{M}\sum_{k=1}^{K}[\mathbf{y}_m]_k \log\big(\text{Prob}(C_k|\mathbf{x}_m)\big) = -\sum_{m=1}^{M}\log \text{Prob}\big(C(m)|\mathbf{x}_m\big), \tag{1.21}$$

where $[\mathbf{y}_m]_k = \delta_{k,C(m)}$ is a binary variable (0 or 1) that indicates class membership and $\text{Prob}(C_k|\mathbf{x}_m) = p_k$, with p_k given by Eq. (1.20); here $\mathbf{z} = \boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_m)$ is the output of the neural network for the input $\mathbf{x}_m$. Let us note that the minimization of Eq. (1.21) is actually equivalent to the minimization of the empirical Kulback–Leibler divergence between the "true" distribution $[\mathbf{y}_m]_k = \delta_{k,C(m)}$ and the predicted distribution given by $\text{Prob}(C_k|\mathbf{x}_m)$.

In order to adapt the backpropagation scheme of Section 1.3.1 to the present scenario, we need to determine the Jacobian of the softmax transformation. Again, this is achieved by applying the chain rule. To that end we evaluate the partial derivatives of the probabilities (1.20) with respect to z_l, which yields

$$\frac{\partial p_k}{\partial z_l} = \begin{cases} p_k(1-p_l), & k = l \\ -p_k p_l, & \text{otherwise} \end{cases}, \tag{1.22}$$

$$= p_k(\delta_{k,l} - p_l). \tag{1.23}$$

Let us now consider the contribution to the cross-entropy criterion (1.21) of sample m which belongs to the class $C(m)$. It is given by

$$J_m = -\sum_{k=1}^{K} y_k \log p_k = -\log p_{C(m)}$$

with $y_k = [\mathbf{y}_m]_k$ and $p_k = \text{Prob}(C_k|\mathbf{x}_m)$. By the chain rule and using the property that $\sum_{k=1}^{K} y_k = 1$, we obtain the formula

$$\frac{\partial J_m}{\partial z_l} = -\sum_{k=1}^{K} y_k \frac{p_k(\delta_{k,l} - p_l)}{p_k} = p_l - y_l$$

which is remarkable for its simplicity. The surprising aspect is that the gradient is essentially the same as if we had applied a least squares criterion to the probabilities – the difference, of course, is that we are backpropagating the gradient with respect to z_k and not p_k!

In effect, this means that one can adapt the backpropagation procedure (Algorithm 1.2) to the case of a categorical loss by a straightforward adjustment of the initialization step. More precisely, we substitute the term $(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_m) - \mathbf{y}_m)$ in Eq. (1.17)

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press

by $(\boldsymbol{p}_{\boldsymbol{\theta}}(\mathbf{x}_m) - \mathbf{y}_m)$, with $\boldsymbol{p}_{\boldsymbol{\theta}}(\mathbf{x}_m) = \text{Softmax}\big(\boldsymbol{f}_{\boldsymbol{\theta}}(\mathbf{x}_m)\big)$, where the softmax operator is defined by Eq. (1.20).

There is also a probabilistic interpretation of criterion (1.21). It is used in what is called logistic regression. If we assume that the samples $\mathbf{x}_m$ are independent, with class probabilities $\text{Prob}(C_k|\mathbf{x}_m)$, then the global probability associated with the dataset $(\mathbf{x}_1, \ldots, \mathbf{x}_M)$ is

$$\prod_{m=1}^{M} \prod_{k=1}^{K} \text{Prob}(C_k|\mathbf{x}_m)^{[\mathbf{y}_m]_k}, \tag{1.24}$$

which, upon taking the (negative) logarithm, yields Eq. (1.21). This shows that minimization of the cross-entropy criterion is actually equivalent to the search for a maximum-likelihood solution.

1.5 Good Practice for Training DNNs

Here is a short list of practical aspects to consider when designing and training neural networks.

1. **Start from an architecture that already works**. Neural networks require a lot of heuristics. It is therefore recommended to start from models that are known to perform well for a given task – e.g., convolutional neural networks for image processing or segmentation. Also, consider incorporating components such as batch normalization [2] and skip-connections [3], whose positive effect on performance is well documented.
2. **Split the data into training, validation, and test sets**. Knowledge of the validation loss of the trained network is helpful to optimize the hyperparameters (the number of layers, number of filters, regularization weight, etc.). The test set should only be used at the very end for evaluation purposes.

   When the data is scarce – as is often the case for medical applications – perform k-fold cross-validation, which makes use of the whole dataset.
3. **Regularize the network to improve generalization or promote sparsity**. For the first use-case, the most common practice is to add weight decay (an ℓ_2-penalty on the weights); other methods include early stopping and dropout [4]. For promoting sparsity, an ℓ_1-norm regularization can be added to the loss.
4. **Use prior knowledge to guide the construction of the network**. For example, choose filter sizes that are appropriate to capture the content of the images you are classifying.
5. **Progressively reduce the learning rate (the step size of SGD) as training progresses**. One should typically begin training with a larger learning rate – in order to obtain faster convergence and avoid local minima – and then decrease it when the loss starts to converge – to prevent oscillations around a minimum.
6. **Prefer the adaptive moment estimation (ADAM) optimizer to "vanilla" gradient descent methods**. Classical gradient descent methods have been mostly

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press

replaced by optimization strategies that rely on higher-order moments of the gradient or include momentum. The most salient example is the ADAM optimizer [5], which has now become the *de facto* standard.
7. **Augment the training data to improve performance**. The training data can be extended by performing operations on the available data points which do not change their labels. Common operations of this kind which are often applied to images include flipping, cropping, and rigid-body or elastic deformations [6, 7].
8. **Consider alternatives to grid search for optimizing the hyperparameters**. Beware that there are better methods to optimize the network hyperparameters than simple intuition or grid search; these include Bayesian and evolutionary optimization [8, 9].
9. **Use tools such as Tensorboard**[1,2] to visualize the evolution of training and guide your decisions on hyperparameter optimization.

References

[1] K. H. Jin, M. T. McCann, E. Froustey, and M. Unser, "Deep convolutional neural network for inverse problems in imaging," *IEEE Transactions on Image Processing*, vol. 26, no. 9, pp. 4509–4522, 2017.
[2] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. International Conference on Machine Learning*. PMLR, 2015, pp. 448–456.
[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. 2016 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2016, pp. 770–778.
[4] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014.
[5] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. International Conference on Learning Representations*, 2015.
[6] S. Zagoruyko and N. Komodakis, "Wide residual networks," *arXiv:1605.07146*, 2016.
[7] S. Bianco, C. Cusano, F. Piccoli, and R. Schettini, "Personalized image enhancement using neural spline color transforms," *IEEE Transactions on Image Processing*, vol. 29, pp. 6223–6236, 2020.
[8] J. Bergstra, D. Yamins, and D. Cox, "Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures," in *Proc. International Conference on Machine Learning*. PMLR, 2013, pp. 115–123.
[9] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, eds., vol. 24. Curran Associates, 2011.

[1] https://www.tensorflow.org/tensorboard/
[2] https:/pytorch.org/docs/stable/tensorboard.html

https://doi.org/10.1017/9781009042529.003 Published online by Cambridge University Press