Chapter 4
Deep Neural Networks

Abstract In this chapter, we introduce deep neural networks (DNNs)—multilayer perceptrons with many hidden layers. DNNs play an important role in the modern speech recognition systems, and are the focus of the rest of the book. We depict the architecture of DNNs, describe the popular activation functions and training criteria, illustrate the famous backpropagation algorithm for learning DNN model parameters, and introduce practical tricks that make the training process robust.

4.1 The Deep Neural Network Architecture

A deep neural network (DNN)\(^1\) is a conventional multilayer perceptron (MLP) with many (often more than two) hidden layers. See a comprehensive review of the DNN in use as an acoustic model for speech recognition in [11]. Figure 4.1 depicts a DNN with a total of five layers that include an input layer, three hidden layers and an output layer. For the sake of notation simplicity, we denote the input layer as layer 0 and the output layer as layer \(L\) for an \(L + 1\)-layer DNN.

In the first \(L\) layers

\[
v^\ell = f \left( z^\ell \right) = f \left( W^\ell v^{\ell-1} + b^\ell \right), \quad \text{for } 0 < \ell < L, \tag{4.1}
\]

where \(z^\ell = W^\ell v^{\ell-1} + b^\ell \in \mathbb{R}^{N_\ell \times 1}\), \(v^\ell \in \mathbb{R}^{N_\ell \times 1}\), \(W^\ell \in \mathbb{R}^{N_\ell \times N_{\ell-1}}\), \(b^\ell \in \mathbb{R}^{N_\ell \times 1}\), and \(N_\ell \in \mathbb{R}\) are, respectively, the excitation vector, the activation vector, the weight matrix, the bias vector, and the number of neurons at layer \(\ell\). \(v^0 = o \in \mathbb{R}^{N_0 \times 1}\) is the observation (or feature) vector, \(N_0 = D\) is the feature dimension, and \(f(\cdot) : \mathbb{R}^{N_\ell \times 1} \rightarrow \mathbb{R}^{N_\ell \times 1}\) is the activation function applied to the excitation vector element-wise. In most applications, the sigmoid function

---

\(^1\) The term deep neural network first appeared in [21] in the context of speech recognition, but was coined in [5] which converted the term deep belief network in the earlier studies into the more appropriate term of deep neural network [4, 17, 24]. The term deep neural network was originally introduced to mean multilayer perceptrons with many hidden layers, but was later extended to mean any neural network with a deep structure.
Fig. 4.1 An example deep neural network with an input layer, three hidden layers, and an output layer.

\[
\sigma(z) = \frac{1}{1 + e^{-z}} \quad (4.2)
\]

or the hyperbolic tangent function

\[
\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (4.3)
\]

is used as the activation function. Since the \(\tanh(z)\) function is a rescaled version of the sigmoid function, these two activation functions have the same modeling power. However, the output range of \(\sigma(z)\) is \((0, 1)\), which encourages sparse but, at the same time, asymmetric activation values. On the other hand, the output range of \(\tanh(z)\) is \((-1, +1)\), and thus the activation value is symmetric, which was believed to help the model training [14]. Another popular activation function is the rectified linear unit (ReLU) function

\[
\text{ReLU}(z) = \max(0, z), \quad (4.4)
\]

which enforces sparse activations\(^2\) [7] and has very simple gradient as we will discuss in Sect. 4.2. Since sigmoid function is still the most popular activation functions used by the practitioners, in all the following discussions we assume that the sigmoid activation function is used unless otherwise noted.

The output layer needs to be chosen based on the tasks in hand. For the regression tasks, a linear layer

\[
v^L = z^L = W^Lv^{L-1} + b^L \quad (4.5)
\]

\(^2\) The output of the sigmoid function can be very close to 0 but cannot reach 0, while the output of the ReLU function can be exactly 0.
is typically used to generate the output vector $v^L \in \mathbb{R}^{N_L}$, where $N_L$ is the output dimension.

For the multi-class classification tasks each output neuron represents a class $i \in \{1, \ldots, C\}$, where $C = N_L$ is the number of classes. The value of the $i$th output neuron $v_i^L$ represents the probability $P_{dnn}(i|o)$ that the observation vector $o$ belongs to class $i$. To serve as a valid multinomial probability distribution, the output vector $v^L$ should satisfy the requirements $v_i^L \geq 0$ and $\sum_{i=1}^C v_i^L = 1$. This can be done by normalizing the excitation with a softmax function

$$v_i^L = P_{dnn}(i|o) = \text{softmax}_i \left( z^L \right) = \frac{e^{z_i^L}}{\sum_{j=1}^C e^{z_j^L}},$$

where $z_i^L$ is the $i$th element of the excitation vector $z^L$.

Given an observation vector $o$, the output of the DNN specified by the model parameters $(W, b) = \{W^\ell, b^\ell|0 < \ell \leq L\}$ can be calculated by computing the activation vectors with Eq. (4.1) layer by layer from layer 1 to layer $L - 1$ and with Eq. (4.5) for the regression tasks and Eq. (4.6) for the classification tasks to calculate the output of the DNN. This process is often called forward computation and is summarized in Algorithm 4.1.

---

**Algorithm 4.1 DNN Forward Computation**

1: procedure FORWARDCOMPUTATION($O$) ▶ Each column of $O$ is an observation vector 
2: $V^0 \leftarrow O$
3: for $\ell \leftarrow 1; \ell < L; \ell \leftarrow \ell + 1$ do ▶ $L$ is the total number of layers
4: $Z^\ell \leftarrow W^\ell V^{\ell-1} + B^\ell$ ▶ Each column of $B^\ell$ is $b^\ell$
5: $V^\ell \leftarrow f \left( Z^\ell \right)$ ▶ $f(.)$ can be sigmoid, tanh, ReLU, or other functions
6: end for
7: $Z^L \leftarrow W^L V^{L-1} + B^L$
8: if regression then ▶ regression task
9: $V^L \leftarrow Z^L$
10: else ▶ classification task
11: $V^L \leftarrow \text{softmax} \left( Z^L \right)$ ▶ Apply softmax column-wise
12: end if
13: Return $V^L$
14: end procedure

---

### 4.2 Parameter Estimation with Error Backpropagation

It has been known since 1980s that an MLP with a sufficiently large hidden layer is a universal approximator [12]. In other words, such an MLP can approximate any mapping $g : \mathbb{R}^D \rightarrow \mathbb{R}^C$ from the input space $\mathbb{R}^D$ to the output space $\mathbb{R}^C$. It is
obvious that a DNN, which is an MLP with more than two hidden layers, can also serve as a universal approximator.

The model parameters \(\{W, b\}\) in a DNN, however, are unknown and need to be estimated from training samples \(\mathcal{S} = \{(o^m, y^m) | 0 \leq m < M\}\) for each task, where \(M\) is the number of training samples, \(o^m\) is the \(m\)th observation vector and \(y^m\) is the corresponding desired output vector. This process is often called the training process or the parameter estimation process, which can be specified by a training criterion and a learning algorithm.

### 4.2.1 Training Criteria

The training criterion should be easy to evaluate and be highly correlated to the final goal of the task so that the improvement in the training criterion would lead to the improvement in the final evaluation score. Ideally, the model parameters should be trained to minimize the expected loss

\[
J_{EL} = \mathbb{E} \left( J(W, b; o, y) \right) = \int_{o} J(W, b; o, y) p(o) \, d(o),
\]

where \(J(W, b; o, y)\) is the loss function given the model parameters \(\{W, b\}\), the observation \(o\), and the corresponding output vector \(y\), and \(p(o)\) is the probability density function of observation \(o\). Unfortunately, \(p(o)\) is typically unknown and needs to be estimated from the training set, and \(J(W, b; o, y)\) is not well-defined (the desired output vector is unknown) for samples unseen in the training set. For this reason, the DNN model parameters are often trained to optimize the empirical criteria.

There are two popular empirical training criteria in DNN model learning. For the regression tasks, the mean square error (MSE) criterion

\[
J_{MSE}(W, b; \mathcal{S}) = \frac{1}{M} \sum_{m=1}^{M} J_{MSE}(W, b; o^m, y^m)
\]

is typically used, where

\[
J_{MSE}(W, b; o, y) = \frac{1}{2} \left\| v^L - y \right\|^2 = \frac{1}{2} \left( v^L - y \right)^T \left( v^L - y \right).
\]

For the classification tasks, \(y\) is a probability distribution and the cross-entropy (CE) criterion

\[
J_{CE}(W, b; \mathcal{S}) = \frac{1}{M} \sum_{m=1}^{M} J_{CE}(W, b; o^m, y^m)
\]

is used.
is often used, where

\[
J_{CE}(W, b; o, y) = - \sum_{i=1}^{C} y_i \log v^L_i, \tag{4.11}
\]

\( y_i = P_{\text{emp}}(i | o) \) is the empirical (observed in the training set) probability that the observation \( o \) belongs to class \( i \), and \( v^L_i = P_{\text{dnn}}(i | o) \) is the same probability estimated from the DNN. Minimizing the cross-entropy criterion is equivalent to minimizing the Kullback-Leibler divergence (KLD) between the empirical probability distribution and the probability distribution estimated from the DNN. In most cases, a hard class label is used, or \( y_i = \mathbb{I}(c = i) \), where

\[
\mathbb{I}(x) = \begin{cases} 
1, & \text{if } x \text{ is true} \\
0, & \text{otherwise}
\end{cases} \tag{4.12}
\]

is the indicator function, and \( c \) is the class label in the training set for observation \( o \). In these cases, the CE criterion specified by Eq. (4.11) is reduced to the negative log-likelihood (NLL)

\[
J_{\text{NLL}}(W, b; o, y) = - \log v^L_c. \tag{4.13}
\]

### 4.2.2 Training Algorithms

Given the training criterion, the model parameters \( \{W, b\} \) can be learned with the famous error backpropagation (BP) algorithm \([19]\), which can be derived from the chain rule used for gradient computation.\(^3\)

In its simplest form, the model parameters can be improved based on the first-order gradient information as

\[
W_{t+1}^\ell \leftarrow W_t^\ell - \varepsilon \nabla W_t^\ell J(W, b; o^m, y^m) \tag{4.14}
\]

and

\[
b_{t+1}^\ell \leftarrow b_t^\ell - \varepsilon \nabla b_t^\ell, \tag{4.15}
\]

where \( W_t^\ell \) and \( b_t^\ell \) are the weight matrix and the bias vector at layer \( \ell \) after the \( t \)th update,

\[
\nabla W_t^\ell = \frac{1}{M_b} \sum_{m=1}^{M_b} \nabla_w W_t^\ell J(W, b; o^m, y^m) \tag{4.16}
\]

---

\(^3\) Although the name backpropagation was coined in 1986 \([19]\) the algorithm itself can be traced back at least to 1969 \([3]\) as a multistage dynamic system optimization method.
\[
\Delta b_t^l = \frac{1}{M_b} \sum_{m=1}^{M_b} \nabla b_t^l J(W, b; o^m, y^m) \tag{4.17}
\]

are, respectively, the average weight matrix gradient and the average bias vector gradient at iteration \( t \) estimated from the training batch consisted of \( M_b \) samples, \( \varepsilon \) is the learning rate, and \( \nabla_x J \) is the gradient of \( J \) with regard to \( x \).

The gradient of the training criterion with regard to the top-layer weight matrix and bias vector depends on the criterion chosen. For the regression tasks, where the MSE training criterion (Eq. 4.9) and the linear output layer (Eq. 4.5) are used, the gradient with regard to the output layer weight matrix is

\[
\nabla_{W_t^L} J_{\text{MSE}} (W, b; o, y) = \nabla_{z_t^L} J_{\text{MSE}} (W, b; o, y) \frac{\partial z_t^L}{\partial W_t^L} = e_t^L \frac{\partial}{\partial W_t^L} \left( W_t^L v_{t-1}^L + b_t^L \right) = e_t^L \left( v_{t-1}^L \right)^T = \left( v_t^L - y \right) \left( v_{t-1}^L \right)^T, \tag{4.18}
\]

where we have defined the error signal at the output layer as

\[
e_t^L \triangleq \nabla_{z_t^L} J_{\text{MSE}} (W, b; o, y) = \frac{1}{2} \frac{\partial}{\partial z_t^L} (z_t^L - y)^T (z_t^L - y) = (v_t^L - y) \tag{4.19}
\]

Similarly,

\[
\nabla_{b_t^L} J_{\text{MSE}} (W, b; o, y) = \left( v_t^L - y \right). \tag{4.20}
\]

For the classification tasks, where the CE training criterion (Eq. 4.11) and the softmax output layer (Eq. 4.6) are used, the gradient with regard to the output layer weight matrix can be derived as

\[
\nabla_{W_t^L} J_{\text{CE}} (W, b; o, y) = \nabla_{z_t^L} J_{\text{CE}} (W, b; o, y) \frac{\partial z_t^L}{\partial W_t^L} = e_t^L \frac{\partial}{\partial W_t^L} \left( W_t^L v_{t-1}^L + b_t^L \right) = e_t^L \left( v_t^L - y \right) \left( v_{t-1}^L \right)^T,
\]
where we have similarly defined the error signal at the output layer as

\[
e_i^L \triangleq \nabla_{z_i^L} J_{CE} (W, b; o, y)
= -\frac{\partial}{\partial z_i^L} \sum_{j=1}^{C} y_j \log \text{softmax}_i (z_i^L)
= \frac{\partial}{\partial z_i^L} \log \sum_{j=1}^{C} e^{z_j^L} - \frac{\partial}{\partial z_i^L} \sum_{j=1}^{C} y_j e^{z_j^L}
= \frac{\partial}{\partial z_i^L} \log \sum_{j=1}^{C} e^{z_j^L} - \frac{\partial}{\partial z_i^L} \sum_{j=1}^{C} y_j e^{z_j^L}
= \begin{bmatrix}
e_1^L \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\sum_{j=1}^{C} e_j^L
\end{bmatrix}
- \begin{bmatrix}
y_1 \\
\vdots \\
y_i \\
\vdots \\
y_C \\
\sum_{j=1}^{C} e_j^L
\end{bmatrix}
= \begin{bmatrix}
e_1^L \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\sum_{j=1}^{C} e_j^L
\end{bmatrix}
- \begin{bmatrix}
y_1 \\
\vdots \\
y_i \\
\vdots \\
y_C \\
\sum_{j=1}^{C} e_j^L
\end{bmatrix}
= \begin{bmatrix}
e_1^L \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\sum_{j=1}^{C} e_j^L
\end{bmatrix}
- \begin{bmatrix}
y_1 \\
\vdots \\
y_i \\
\vdots \\
y_C \\
\sum_{j=1}^{C} e_j^L
\end{bmatrix}
= (v_i^L - y).
\]

Similarly,

\[
\nabla_{b^L} J_{CE} (W, b; o, y) = (v_i^L - y). \tag{4.22}
\]

Note that \(\nabla_{W^L} J_{CE} (W, b; o, y)\) (Eq. 4.21) appears to have the same form as \(\nabla_{W^L} J_{MSE} (W, b; o, y)\) (Eq. 4.18). However, they are actually different since in the regression case \(v_i^L = z_i^L\) and in the classification case \(v_i^L = \text{softmax} (z_i^L)\).

Note also that for \(0 < \ell < L\),

\[
\nabla_{W^L} J (W, b; o, y) = \nabla_{v_i^L} J (W, b; o, y) \frac{\partial v_i^\ell}{\partial W_i^\ell}
= \text{diag} \left( f' (z_i^\ell) \right) e_i^\ell \left( W_i^\ell v_i^{\ell-1} + b_i^\ell \right)
\]
\[ \nabla b^\ell J (W, b; o, y) = \nabla v^\ell J (W, b; o, y) \frac{\partial v^\ell}{\partial b^\ell} = \text{diag} \left( f' \left( z^\ell \right) \right) e^\ell \] 
\[ = f' \left( z^\ell \right) \cdot e^\ell, \quad (4.24) \]

where \( e^\ell \triangleq \nabla v^\ell J (W, b; o, y) \) is the error signal at layer \( \ell \), \( \cdot \) is the element-wise product, \( \text{diag}(x) \) is a square matrix whose diagonal equals to \( x \), and \( f' \left( z^\ell \right) \) is the element-wise derivative of the activation function. For the sigmoid activation function,

\[ \sigma' \left( z^\ell \right) = \left( 1 - \sigma \left( z^\ell \right) \right) \cdot \sigma \left( z^\ell \right) = \left( 1 - v^\ell \right) \cdot v^\ell. \quad (4.25) \]

Similarly, for the tanh activation function

\[ \tanh' \left( z^\ell \right) = 1 - \left[ \tanh \left( z^\ell \right) \right]^2 = 1 - \left[ v^\ell \right]^2, \quad (4.26) \]

or

\[ \tanh' \left( z^\ell \right) = 1 - v^\ell \cdot v^\ell. \quad (4.27) \]

For the ReLU activation function

\[ \text{ReLU}' \left( z^\ell \right) = \begin{cases} 
1, & \text{if } z^\ell > 0 \\
0, & \text{otherwise}
\end{cases} \quad (4.28) \]

or

\[ \text{ReLU}' \left( z^\ell \right) = \max \left( 0, \text{sgn} \left( z^\ell \right) \right), \quad (4.29) \]

where \( \text{sgn} \left( z^\ell \right) \) is the sign of \( z^\ell \) applied element-wise. The error signal can be back-propagated from top to bottom as
\[ e_{t}^{L-1} = \nabla_{v_{t}^{L-1}} J(W, b; o, y) \]
\[ = \frac{\partial z_{t}^{L}}{\partial v_{t}^{L-1}} \nabla_{z_{t}^{L}} J(W, b; o, y) \]
\[ = \frac{\partial \left( W_{t}^{f} v_{t}^{L-1} + b_{t}^{L} \right)}{\partial v_{t}^{L-1}} e_{t}^{L} \]
\[ = \left( W_{t}^{L} \right)^{T} e_{t}^{L}. \quad (4.30) \]

For \( \ell < L \),
\[ e_{t}^{\ell-1} = \nabla_{v_{t}^{\ell-1}} J(W, b; o, y) \]
\[ = \frac{\partial v_{t}^{\ell}}{\partial v_{t-1}^{\ell}} \nabla_{v_{t}^{\ell}} J(W, b; o, y) \]
\[ = \frac{\partial \left( W_{t}^{f} v_{t}^{\ell-1} + b_{t}^{\ell} \right)}{\partial v_{t-1}^{\ell}} \operatorname{diag} \left( f^{'} \left( z_{t}^{\ell} \right) \right) e_{t}^{\ell} \]
\[ = \left( W_{t}^{\ell} \right)^{T} \left[ f^{'} \left( z_{t}^{\ell} \right) \cdot e_{t}^{\ell} \right]. \quad (4.31) \]

The key steps in the backpropagation algorithm is summarized in Algorithm 4.2.

4.3 Practical Considerations

The basic backpropagation algorithm described in Sect. 4.2 is theoretically simple. However, learning a model efficiently and effectively requires taking consideration many practical issues \([1, 14]\).

4.3.1 Data Preprocessing

Data preprocessing plays an important role in many machine learning algorithms. The two most popular preprocessing techniques are per-sample feature normalization and global feature standardization.

If the mean of each sample reflects a variation that is irrelevant to the problem in hand, this mean should be subtracted from the feature to reduce the variability in the final feature fed to the DNN. For example, subtracting the mean intensity of an image can reduce the variability introduced by the brightness. In the handwriting character recognition tasks, normalizing the image center can reduce the variability caused by the shifted character position. In speech recognition, the cepstral mean
Algorithm 4.2 Backpropagation Algorithm

1: procedure Backpropagation($S = \{(o^m, y^m) \mid 0 \leq m < M\}$) 
   \hspace{1em} $\triangleright$ $S$ is the training set with $M$ samples
2: Randomly initialize $\{W_\ell^0, b_\ell^0\}$, $0 < \ell \leq L$ \hspace{1em} $\triangleright$ $L$ is the total number of layers
3: while Stopping Criterion Not Met do
   \hspace{1em} $\triangleright$ Stop if reached max iterations or the training criterion improvement is small
4: Randomly select a minibatch $O, Y$ with $M_b$ samples.
5: Call ForwardComputation($O$)
6: $E_L^t \leftarrow V_L^t - Y$ \hspace{1em} $\triangleright$ Each column of $E_L^t$ is $e_L^t$
7: $G_L^t \leftarrow E_L^t$
8: for $\ell \leftarrow L; \ell > 0; \ell \leftarrow \ell - 1$ do
9:   $\nabla W_\ell^t \leftarrow G_\ell^t \left(T_{\ell-1}^t)$ \hspace{1em} $\triangleright$ Update $W$
10:  $\nabla b_\ell^t \leftarrow G_\ell^t$ \hspace{1em} $\triangleright$ Update $b$
11:  $W_{\ell+1}^t \leftarrow W_\ell^t - \frac{\varepsilon}{M_b} \nabla W_\ell^t$ \hspace{1em} $\triangleright$ Update $W$
12:  $b_{\ell+1}^t \leftarrow b_\ell^t - \frac{\varepsilon}{M_b} \nabla b_\ell^t$
13:  $E_{\ell-1}^t \leftarrow (W_\ell^t)^T G_\ell^t$ \hspace{1em} $\triangleright$ Error backpropagation
14:  if $\ell > 1$ then
15:   $E_{\ell-1}^t \leftarrow f' \left(Z_{\ell-1}^t\right) \cdot E_{\ell-1}^t$
16: end if
17: end for
18: end while
19: Return $dnn = \{W_\ell^\ell, b_\ell^\ell\}$, $0 < \ell \leq L$
20: end procedure

normalization (CMN) [15] technique, which subtracts the per-utterance mean of the MFCC features can reduce acoustic channel distortions. Using CMN as an example, the per-sample normalization can be carried out by first estimating the per-utterance mean

\[
\bar{\mu}_i = \frac{1}{T} \sum_{t=1}^{T} o_i^t, \quad (4.32)
\]

for each dimension $i$, where $T$ is the total number of frames in the utterance, and then subtracting the mean from all frames in the utterance as

\[
\tilde{o}_i^t = o_i^t - \bar{\mu}_i. \quad (4.33)
\]

The goal of global feature standardization is to scale the data along each dimension using a global transformation so that the final data vectors lie in the similar range. For example, in image processing, we often rescale the pixel values in the range of $[0, 255]$ to the range of $[0, 1]$. For real-valued features such as the MFCC and the log filter-bank feature in speech recognition tasks, each dimension of the feature is often standardized to have zero-mean and unit-variance using a global transformation (e.g., in [5]). The global transformation in both cases is estimated from the training data only and it is then applied to both training and test sets. Given the training set $S = \{(o^m, y^m) \mid 0 \leq m < M\}$ (which may have been normalized per sample), we can
calculate the average value

\[ \mu_i = \frac{1}{M} \sum_{m=1}^{M} o_i^m \]  

(4.34)

and the standard deviation

\[ \sigma_i = \sqrt{\frac{1}{M} \sum_{m=1}^{M} (o_i^m - \mu_i)^2} \]  

(4.35)

for each dimension \( i \). All samples in both training and test sets can be standardized as

\[ \tilde{o}_i^m = \frac{o_i^m - \mu_i}{\sigma_i}. \]  

(4.36)

The global feature standardization described above is useful because the later processing steps often perform better when each dimension is scaled to a similar range of numerical values [14]. For example, in DNN training, by standardizing features we can use the same learning rate across all weight dimensions and still get a good model. Without feature standardization, the energy component or the first MFCC component \( c_0 \) would overshadow the other components and dominate the learning process unless learning algorithms (such as AdaGrad [6]), which automatically adjust learning rate for each separate dimension, are used.

### 4.3.2 Model Initialization

The learning algorithm specified in Sect. 4.2 starts from an initial model. Since the DNN is a highly nonlinear model and the training criterion with regard to the model parameters is nonconvex, the initial model can greatly affect the resulting model.

There are many heuristic tricks in initializing the DNN model. Most of these tricks are based on two considerations. First, the weights should be initialized so that each neuron operates in the linear range of the sigmoid function at the start of the learning. If weights were all very large, many neurons would saturate (close to zero or one) and the gradients would be very small according to Eq. (4.25). When the neurons operate in the linear range, instead, the gradients are large enough (close to the maximum value of 0.25) that learning can proceed effectively. Note that the excitation value depends on both the input values and the weights. When the input features are standardized as described in Sect. 4.3.1, determining the initial weights can become easier. Second, it is important to initialize the parameters randomly. This is because neurons in the DNNs are symmetric and interchangeable. If all the model parameters have identical values, all the hidden layer neurons will have the same output value and detect the same feature patterns in the lower layers. Random initialization serves the purpose of symmetry breaking.
LeCun and Bottou [14] suggested to draw values from a zero-mean Gaussian distribution with standard deviation \( \sigma_{W^{\ell+1}} = \frac{1}{\sqrt{N_\ell}} \) to initialize the weights in layer \( \ell \) as defined in Eq. (4.1), where \( N_\ell \) is the number of connections feeding into the node. For the DNNs used in speech recognition tasks, where each hidden layer has 1,000–2,000 neurons, we have found that initializing the weight matrices by drawing from a Gaussian distribution \( N(w; 0, 0.05) \) or from a uniform distribution in the range of \([-0.05, 0.05]\) performs very well. The bias vectors \( \mathbf{b}^\ell \) can be simply initialized to zero.

### 4.3.3 Weight Decay

As in many machine learning algorithms, overfitting can be a problem especially since the number of parameters in a DNN is huge compared to many other learning machines. Overfitting happens because we are interested in minimizing the expected loss \((4.7)\), but instead we actually minimize the empirical losses defined on the training set.

The simplest way to control overfitting is to regularize the training criterion so that the model parameters will not be tuned to fit the training data too well. The most widely used regularization terms include

\[
R_1 (W) = \| \text{vec} (W) \|_1 = \sum_{\ell=1}^{L} \| \text{vec} (W^\ell) \|_1 = \sum_{\ell=1}^{L} \sum_{i=1}^{N_\ell} \sum_{j=1}^{N_{\ell-1}} |W_{ij}^\ell| \tag{4.37}
\]

that is based on the \( L_1 \) norm, and

\[
R_2 (W) = \| \text{vec} (W) \|_2^2 = \sum_{\ell=1}^{L} \| \text{vec} (W^\ell) \|_2^2 = \sum_{\ell=1}^{L} \sum_{i=1}^{N_\ell} \sum_{j=1}^{N_{\ell-1}} (W_{ij}^\ell)^2 \tag{4.38}
\]

that is based on the \( L_2 \) norm, where \( W_{ij} \) is the \((i, j)\)-th value in the matrix \( W \), \text{vec} \((W^\ell) \in \mathbb{R}^{[N_\ell \times N_{\ell-1}] \times 1}\) is the vector generated by concatenating all the columns in the matrix \( W^\ell \), and \( \| \text{vec} (W^\ell) \|_2 \) equals to \( \| W^\ell \|_F \) —the Frobenius norm of the matrix \( W^\ell \). These regularization terms are often called weight decay in the neural network literature.

When regularization terms are included, the training criterion becomes

\[
\tilde{J} (W, b; \mathcal{S}) = J (W, b; \mathcal{S}) + \lambda R (W) \tag{4.39}
\]

where \( J (W, b; \mathcal{S}) \) is either \( J_{\text{MSE}} (W, b; \mathcal{S}) \) or \( J_{\text{CE}} (W, b; \mathcal{S}) \) that optimizes the empirical loss on the training set \( \mathcal{S} \), \( R (W) \) is either \( R_1 (W) \) or \( R_2 (W) \) described above, and \( \lambda \) is an interpolation weight, which is sometimes called regularization.
weight. Note that

\[
\nabla_{W^t} \tilde{J}(W, b; o, y) = \nabla_{W^t} J(W, b; o, y) + \lambda \nabla_{W^t} R(W) \tag{4.40}
\]

\[
\nabla_{b^t} \tilde{J}(W, b; o, y) = \nabla_{b^t} J(W, b; o, y) \tag{4.41}
\]

where

\[
\nabla_{W^t} R_1(W) = \text{sgn}(W^t), \tag{4.42}
\]

and

\[
\nabla_{W^t} R_2(W) = 2W^t. \tag{4.43}
\]

Weight decay often helps when the training set size is small compared to the number of parameters in the DNN. Since there are typically over one million parameters in the weight matrices in typical DNNs used in the speech recognition tasks, the interpolation weight \(\lambda\) should be small (often in the range of \(10^{-4}\)) or even 0 when the training set size is large.

### 4.3.4 Dropout

Weight decay is one way to control the overfitting. Another popular approach is dropout [10]. The basic idea of dropout is to randomly omit a certain percentage (e.g., \(\alpha\)) of the neurons in each hidden layer for each presentation of the samples during training. This means during the training each random combination of the \((1 - \alpha)\) remaining hidden neurons needs to perform well even in the absence of the omitted neurons. This requires each neuron to depend less on other neurons to detect patterns.

Alternatively, dropout can be considered a technique that adds random noise to the training data. This is because each higher-layer neuron gets input from a random collection of the lower-layer neurons. The excitation received by each neuron is different even if the same input is fed into the DNN. With dropout, DNNs need to waste some of the weights to remove the effect of the random noise introduced. As such, dropout essentially reduces the capacity of the DNN, and thus can improve generalization of the resulting model.

When a hidden neuron is dropped out, its activation is set to 0 and so no error signal will pass through it. This means that other than the random dropout operation, no other changes to the training algorithm are needed to implement this feature. At the test time, however, instead of using a random combination of the neurons at each hidden layer, we use the average of all the possible combinations. This can be easily accomplished by discounting all the weights involved in dropout training by \((1 - \alpha)\), and then using the resulting model as a normal DNN (i.e., without dropout). Thus,
dropout can also be interpreted as an efficient way of performing (geometric) model averaging (similar to bagging) in the DNN framework.

A slightly different implementation is to divide each activation by \((1 - \alpha)\) before neurons are dropped in the training. By doing so the weights are automatically discounted by \((1 - \alpha)\), and thus no weight compensation is needed when the model is used for testing. Another benefit of this approach is that we may apply different dropout rates at different epoch in the training. Our experience indicates that a dropout rate of 0.1–0.2 often helps to improve the recognition rate. An intelligently designed training schedule that uses a large dropout rate (e.g., 0.5) initially and reduces the dropout rate gradually can further improve the performance. This is because the model trained with a larger dropout rate can be considered as the seed model of that trained with a smaller dropout rate. Since the objective function associated with the larger dropout rate is smoother, it is less likely to trap into a very bad local optimum.

During the dropout training, we need to repeatedly sample a random subset of activations at each layer. This would slow down the training significantly. For this reason, the speed of the random number generation and sampling code is critical to reducing the training time. Alternatively, a fast dropout training algorithm proposed in [23] can be used. The key idea of this algorithm is to sample from or integrate a Gaussian approximation, instead of doing Monte Carlo sampling. This approximation, justified by the central limit theorem and empirical evidence, provides significant speedup and more stability. This technique can be extended to integrate out other types of noise and transformations.

### 4.3.5 Batch Size Selection

The parameter update formulas Eqs. (4.14) and (4.15) require the calculation of the empirical gradient estimated from a batch of training samples. The choice of the batch size will affect both the convergence speed and the resulting model.

The simplest and obvious choice of the batch is the whole training set. If our only goal is to minimize the empirical loss on the training set, the gradient estimated from the whole training set is the true gradient (i.e., the variance is zero). Even if our goal is to optimize the expected loss the gradient estimated from the whole training set still has smaller variance than that estimated from any subset of the training data. This approach, often referred to as batch training, has several advantages: First, the convergence property of the batch training is well-known. Second, many accelerating techniques such as conjugate gradient [9] and L-BFGS [16] work best in batch training. Third, batch training can be easily parallelized across computers. Unfortunately, batch training requires a complete pass through the entire dataset before model parameters are updated, and is thus not efficient in many large scale problems even though embarrassing parallelization is possible.

Alternatively, we can use the stochastic gradient descent (SGD) [2] technique, which is sometimes referred to as online learning in the machine learning literature. SGD updates the model parameters based on the gradients estimated from a single
training sample. If the sample is i.i.d. (independently and identically distributed),
which is easy to guarantee if we draw the sample from the training set following the
uniform distribution, we can show that

$$E(\nabla J_t(W, b; o, y)) = \frac{1}{M} \sum_{m=1}^{M} \nabla J(W, b; o^m, y^m).$$ (4.44)$$

In other words, the gradient estimated from the single sample is an unbiased es-
timation of the gradient on the whole training set. However, the variance of the
estimation is

$$\nabla(\nabla J_t(W, b; o, y)) = E\left[(x - E(x))(x - E(x))^T\right]$$
$$= E(xx^T) - E(x)E(x)^T$$
$$= \frac{1}{M} \sum_{m=1}^{M} x_m x_m^T - E(x)E(x)^T,$$ (4.45)

which is nonzero unless all samples are identical; i.e., \(\nabla J_t(W, b; o, y) = E(\nabla J_t(W, b; o, y))\), where for simplicity we have defined \(x \triangleq \nabla J_t(W, b; o, y)\).

Because this estimate of the gradient is noisy, the model parameters may not move
precisely down the gradient at each iteration. This seemingly drawback, however,
is actually an important advantage of the SGD algorithm over the batch learning
algorithm. This is because DNNs are highly nonlinear and nonconvex. The objective
function contains many local optima, many of which are very poor. Batch learning
will find the minimum of whatever basin the model parameters are initially in and
results in a model that is highly dependent on the initial model. The SGD algorithm,
however, due to the noisy gradient estimation, can jump out of the poor local optima
and enter a better basin. This property is similar to the simulated annealing [13],
which allows the model parameters to move in a direction that is inferior locally but
superior globally.

SGD is also often much faster than batch learning especially on large datasets. This
is due to two reasons. First, typically there are many similar, sometimes redundant,
samples in the large datasets. Estimating the gradient by going through the whole
dataset is thus waste of computation. Second, and more importantly, in the SGD
training, one can make quick updates after seeing each sample. The new gradient is
estimated based on the new model instead of the old model and so can move more
quickly toward finding the best model.

The SGD algorithm, however, is difficult to parallelize even on the same com-
puter. In addition, it cannot fully converge to the local minimum due to the noisy
estimation of the gradient. Instead, it will fluctuate around the minimum. The size
of the fluctuation depends on the learning rate and the amplitude of the gradient
estimation variance. Although this fluctuation can sometimes reduce overfitting, it
is not desirable in many cases.
A compromise between batch learning and the SGD algorithm is minibatch training, which estimates the gradient based on a small batch of randomly drawn training samples. It is easy to show that the gradient estimated from the minibatch is also unbiased and the variance of the estimation is smaller than that in the SGD algorithm. Minibatch training allows us to easily parallelize within the minibatch, and thus can converge faster than SGD. Since we prefer large gradient estimation variance in the early stage of the training to quickly jump out of poor local optima and smaller variance at the later stage to settle down to the minimum, we can choose smaller minibatch size initially and large ones in the later stage. In speech recognition tasks, we have found that a better model can be learned if we use 64–256 samples in early stages and 1,024–8,096 samples in later stages. An even smaller batch size is preferred at the very initial stage when a deeper network is to be trained. The minibatch size may be automatically determined based on the gradient estimation variance. Alternatively, the batch size can be determined by searching on a small subset of samples in each epoch [8, 20].

### 4.3.6 Sample Randomization

Sample randomization is irrelevant to the batch training since all samples are used to estimate the gradient. However, it is very important for SGD and minibatch training. This is because to get an unbiased estimate of the gradient the samples have to be IID. Heuristically, if successive samples are not randomly drawn from the training set (e.g., all belong to the same speaker), the model parameters will likely to move along the similar direction for too long.

If the whole training set can be loaded into the memory, sample randomization can be easily done by permuting an index array. Samples can then be drawn one by one according to the permuted index array. Since the index array is typically much smaller than the features, this would cost less than permuting the feature vectors themselves, especially if each data pass requires a different randomization order. This trick also guarantees that each sample will be presented to the training algorithm once for each data pass, and thus will not affect the data distribution. This property will guarantee that the model learned is consistent.

If the training set is huge, which is typically the case in speech recognition, we cannot load the whole training set into memory. In that case, we can load a large chunk (typically 24–48 h of speech or 8.6–17.2 M samples) of the data into memory each time using a rolling window and randomize inside the window. If the training data are from different sources (e.g., different languages), randomizing the utterance list files before feeding them into the DNN training tool also will help.
4.3 Practical Considerations

4.3.7 Momentum

It is well-known that the convergence speed can be improved if the model update is based on all the previous gradients (more global view) instead of only the current one (local view). Nesterov’s accelerated gradient algorithm [18], which is proved to be optimal for the convex condition, is an example of this trick. In the DNN training, this is typically achieved with a simple technique named momentum. When momentum is applied, Eqs. (4.16) and (4.17) are replaced with

\[
\Delta W^\ell_t = \rho \Delta W^\ell_{t-1} + (1 - \rho) \frac{1}{M_b} \sum_{m=1}^{M_b} \nabla W^\ell_t J(W, b; o^m, y^m) \tag{4.46}
\]

and

\[
\Delta b^\ell_t = \rho \Delta b^\ell_{t-1} + (1 - \rho) \frac{1}{M_b} \sum_{m=1}^{M_b} \nabla b^\ell_t J(W, b; o^m, y^m) \tag{4.47}
\]

where \( \rho \) is the momentum factor, which typically takes value of 0.9–0.99 when SGD or minibatch training is used.\(^4\) Momentum smoothes the parameter update and reduces the variance of the gradient estimation. Practically, it can reduce the oscillation problems commonly seen in the regular backpropagation algorithm when the error surface has a very narrow minimum, and thus speed up the training.

The above definition of momentum works great when the minibatch size is the same. However, sometimes we may want to use variable minibatch sizes. For example, we may want to use smaller minibatch initially, and then larger minibatch later as discussed in Sect. 4.3.5. In the sequence-discriminative training, which we will discuss in Chap. 8, each minibatch may have a different size since each utterance is of different length. Under these conditions, the above definition of momentum is no longer valid. Since the momentum can be considered as a finite impulse response (FIR) filter, we can define the momentum at the sample level as \( \rho_s \) and derive the momentum for different minibatch size \( M_b \) as

\[
\rho = \exp(M_b \rho_s). \tag{4.48}
\]

4.3.8 Learning Rate and Stopping Criterion

One of the difficulties in training a DNN is selecting an appropriate learning strategy. It has been shown in theory that when the learning rate is set to

\[^4\] In practice, we have found out that we may achieve slightly better result if we only use momentum after the first epoch.
\[ \varepsilon = \frac{c}{t}, \quad (4.49) \]

where \( t \) is the number of samples presented and \( c \) is a constant, SGD converges asymptotically \([2]\). In practice, however, this annealing scheme converges very slowly since the learning rate will quickly become very small.

Note that it is the combination of the learning rate and the batch size that affects the learning behavior. As we have discussed in Sect. 4.3.5 that a smaller batch size should be used in the first several data passes and a larger batch size should be used in the later data passes. Since the batch size is variable, we can define a per-sample learning rate

\[ \varepsilon_s = \frac{\varepsilon}{M_b}, \quad (4.50) \]

and change the model parameter update formulas to

\[ W_{\ell \ t+1} \leftarrow W_{\ell \ t} - \varepsilon_s \Delta \tilde{W}_{\ell \ t} \quad (4.51) \]

\[ b_{\ell \ t+1} \leftarrow b_{\ell \ t} - \varepsilon_s \Delta \tilde{b}_{\ell \ t} \quad (4.52) \]

where

\[ \Delta \tilde{W}_{\ell \ t} = \rho \Delta W_{\ell \ t-1} + (1 - \rho) \sum_{m=1}^{M_b} \nabla W_i \tilde{J} (W, b; o^m, y^m) \quad (4.53) \]

and

\[ \Delta \tilde{b}_{\ell \ t} = \rho \Delta b_{\ell \ t-1} + (1 - \rho) \sum_{m=1}^{M_b} \nabla b_i \tilde{J} (W, b; o^m, y^m). \quad (4.54) \]

This change also reduces one matrix division compared to using the original definition of the learning rate.

With this new update formulas, we can determine the learning strategy empirically. We first decide on a batch size and a large learning rate. We then run the training for hundreds of minibatches, which typically takes several minutes on multi-core CPUs or GPUs. We monitor the training criterion on these minibatches and reduce the batch size, learning rate, or both so that the product of \( \varepsilon_s M_b \) is halved until the training criterion is obviously improved. We then divide this learning rate by two and use it as the initial learning rate. We run through a large subset of the training set and increase \( \varepsilon_s M_b \) by four to eight folds. Note that since at this stage the model parameters are already adjusted to a relatively good location, increasing \( \varepsilon_s M_b \) will not lead to diverge but improve the training speed. This procedure can be automated as in \([8]\).

We found two strategies useful in determining the rest learning scheme. The first strategy is to double the batch size and reduce the learning rate by fourfolds whenever the training criterion fluctuates as measured on a large training subset or on a development set, and stop the training when the learning rate is smaller than
a threshold or the preset number of data passes is reached. The second strategy is to reduce the learning rate to a very small number after the training criterion fluctuates and stop the training when the fluctuation happens again on either the training or the development set. For the speech recognition tasks in which the models are learned from scratch, we found that $\epsilon_s$ of $0.8e^{-4}$ and $0.3e^{-3}$ for deep and shallow networks, respectively, for the first phase, $1.25e^{-2}$ for the second phase, and $0.8e^{-6}$ for the third phase worked very well in practice. The hyper-parameter searching can also be automated using random search techniques [1] or Bayesian optimization techniques [22].

### 4.3.9 Network Architecture

Network architecture can be considered as another hyper “parameter” that needs to be determined. Since each layer can be considered as a feature extractor of the previous layer, the number of neurons at each layer should be large enough to capture the essential patterns. This is especially important at several lower layers since the first-layer features are more variable and it requires more neurons to model the patterns than other layers. However, if the layer size is too large, it is easier to overfit to the training data. Basically, the wide, shallow models are easier to overfit and the deep, narrow models are easier to underfit. Actually, if one of the layers is small (often called bottleneck) the performance will be significantly deteriorated especially when the bottleneck layer is close to the input layer. If each layer has the same number of neurons, adding more layers may also convert the model from overfiting to underfiting. This is because additional layers impose additional constraints to the model parameters. Given this observation, we can optimize the number of neurons at each layer on a one-hidden-layer network first. We then stack more layers with the same hidden layer size. In the speech recognition tasks, we have found that 5–7 layer DNNs with 1,000–3,000 neurons at each layer worked very well. It is often easier to find a good configuration on a wide and deep model than a narrow and shallow one. This is because there are many good local optima that perform similarly on a wide and deep model.

### 4.3.10 Reproducibility and Restartability

In the DNN training, the model parameters are initialized randomly and the training samples are also fed into the trainer in random orders. This inevitably will raise the concern about reproducibility of the results of the training. If our goal is to compare two algorithms or models, we can run the experiments multiple times, each with a new random seed, and report the average result and the standard deviation. Under many other conditions, however, we want to get exactly the same model and test
result when we run the trainer twice. This can be achieved by always using the same
random seed when generating the initial model and permuting the training samples.

When the training set is large, it is often desirable to stop the training in the
middle and continue the training from the last check-point. Some mechanisms need
to be built into the training tool to guarantee that restarting from a check-point will
generate the exact same result if the training never interrupted. A simple trick is
to save all the necessary information, including model parameters, current random
number, parameter gradient, momentum, and so on in the check-point file. Another
working approach that requires saving less data is to reset all learning parameters
after each check-point.

References