Chapter 3
Training Supervised Deep Learning Networks

3.1 Introduction

Training supervised deep learning networks involves obtaining model parameters using labeled dataset to allow the network to map an input data to a class label. The labeled dataset consists of training examples, where each example is a pair of an input data and a desired class label. The deep model parameters allow the network to correctly determine the class labels for unseen instances. This requires the model to generalize from the training dataset to unseen instances.

Many supervised deep learning models have been used by researchers. Convolutional neural network is one of the commonly used supervised deep architectures. This chapter will discuss the training of convolutional neural networks.

3.2 Training Convolution Neural Networks

Training supervised deep neural network is formulated in terms of minimizing a loss function. In this context, training a supervised deep neural network means searching a set of values of parameters (or weights) of the network at which the loss function has minimum value. Gradient descent is an optimization technique which is used to minimize the error by calculating gradients necessary to update the values of the parameters of the network.

The most common and successful learning algorithm for deep learning models is gradient descent-based backpropagation in which error is propagated backward from last layer to the first layer. In this learning technique, all the weights of a neural network are either initialized randomly or initialized by using probability distribution. An input is fed through the network to get the output. The obtained output and the desired output are then used to calculate the error using some cost function (error function). To understand the working of backpropagation, consider a small Convolution Neural Network (CNN) model shown in Fig. 3.1. The CNN model

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_3

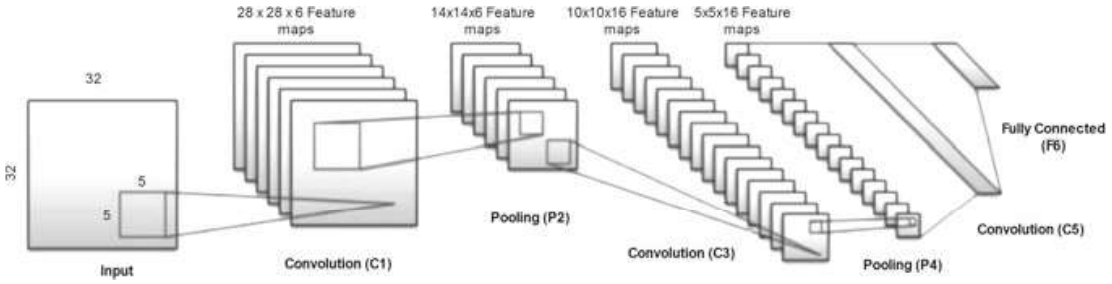

Fig. 3.1 CNN architecture

is similar to LeNet but uses ReLU (which has become a standard in deep networks) as activation function, max-pooling, instead of average pooling. The CNN model consists of three convolution layers, two subsampling layers, and one fully connected layer producing 10 outputs. The first convolution layer convolves input of size 32×32 with six 5×5 filters producing six 28×28 feature maps. Next, pooling layer down-samples these six 28×28 feature maps to size 14×14. Second convolution layer convolves the down-sampled feature maps with 5×5 filters producing 16 feature maps of size 10×10. Second pooling layer down-samples the input producing 16 feature maps of size 5×5. The third convolution layer has the input of size 5×5 and the filter of size 5×5, so no stride happens. We can say that this third layer has 120 filters of size 5×5 fully connected to each of the 16 feature maps of size 5×5. Last layer in the architecture is a fully connected layer containing 10 units for 10 classes. The output of the fully connected layer is fed to cost function (Softmax) to calculate the error. Let us describe the CNN architecture in detail including its mathematical operations.

Layer 1 (*C1*) is a convolution layer which takes input of size 32×32 and convolves it with six filters of size 5×5 producing six feature maps of size 28×28.

Mathematically, the above operation can be given as

$$C1_{i,j}^{k} = \left(\sum_{m=0}^{4} \sum_{n=0}^{4} w_{m,n}^{k} * I_{i+m,j+n} + b^{k} \right) \tag{3.1}$$

where $C1^{k}$ represent six output feature maps of convolution layer $C1$, k represents filter number, (m, n) are the indices of kth filter and (i, j) are the indices of output.

Equation 3.1 is the mathematical form of convolution operation. The output of convolution operation is fed to an activation function to make the output of linear operation nonlinear. In deep networks, the most successful activation function is ReLU given by

$$\sigma(x) = \max(0, x) \tag{3.2}$$

Using Eq. 3.1, we get

$$C1_{i,j}^{k} = \sigma\left(\sum_{m=0}^{4}\sum_{m=0}^{4} w_{m,n}^{k} * I_{i+m,j+n} + b^{k}\right) \tag{3.3}$$

Equation 3.3 is the output of layer $C1$.

Layer 2(*P2*) is a max-pooling layer. The output of the convolution layer $C1$ is fed to max-pooling layer. The max-pooling layer takes six feature maps from $C1$ and performs max-pooling operation on each of them.

The max-pooling operation on $C1^{k}$ is given as

$$P2^{k} = \text{MaxPool}\left(C1^{k}\right)$$

For each feature map in $C1^{k}$, max-pooling performs the following operation:

$$P2_{i,j}^{k} = \max\begin{pmatrix} C1_{(2i,2j)}^{k}, C1_{(2i+1,2j)}^{k} \\ C1_{(2i,2j+1)}^{k}, C1_{(2i+1,2j+1)}^{k} \end{pmatrix} \tag{3.4}$$

where (i, j) are the indices of kth feature map of output, and k is the feature map index.

Layer 3 (*C3*) is the second convolution layer which produces 16 feature maps of size 10×10 and is given by

$$C3_{i,j}^{k} = \sigma\left(\sum_{d=0}^{5}\sum_{m=0}^{4}\sum_{n=0}^{4} w_{m,n}^{k,d} * P2_{i+m,j+n}^{d} + b^{k}\right)$$

where $C3^{k}$ represents the 16 output feature maps of convolution layer $C3$, k is the index of the output feature map, (m, n) are the indices of filter weights, (i, j) are the indices of output, and d is the index of the number of channels in the input.

Layer 4 (*P4*) is a max-pooling layer which produces 16 feature maps $P4^{k}$ of size 5×5.

The max-pooling operation is given as

$$P4^{k} = \text{MaxPool}\left(C3^{k}\right)$$

$$P4_{i,j}^{k} = \max\begin{pmatrix} C3_{(2i,2j)}^{k}, C3_{(2i+1,2j)}^{k} \\ C3_{(2i,2j+1)}^{k}, C3_{(2i+1,2j+1)}^{k} \end{pmatrix}$$

where (i, j) are the indices of kth feature map of output, and k is the feature map index.

Layer 5 (*C5*) is third convolution layer that produces 120 output feature maps and is given by

$$C5_{i,j}^{k} = \sigma\left(\sum_{d=0}^{15}\sum_{m=0}^{4}\sum_{n=0}^{4} w_{m,n}^{k,d} * P4_{i+m,j+n}^{d} + b^{k}\right)$$

where $C5^{k}$ represents the 120 output feature maps of convolution layer $C5$ of size 1×1, k is the index of the output feature map, (m, n) are the indices of filter weights, d is the index of the number of channels in input and (i, j) are the indices of output, since output is only 1×1, the index (i, j) remains (0, 0) for each filter. This formula can be simplified as the filter size is equal to the size of input, so no convolution stride happens

$$C5^{k} = \sigma\left(\sum_{d=0}^{15}\sum_{m=0}^{4}\sum_{n=0}^{4} w_{m,n}^{k,d} * P4_{m,n}^{d} + b^{k}\right)$$

Layer 6 (*F6*) is a fully connected layer. It consists of 10 neurons for 10 classes and is mathematically defined as

$$F6^{k} = \sum_{i=1}^{120} w_{i}^{k} * C5^{i} \tag{3.5}$$

In last layer, for each neuron, the activation function used is softmax function given as

$$Z^{k} = \text{softmax}\left(F6^{k}\right)$$

The softmax function is defined as

$$Z^{k} = \text{softmax}\left(F6^{k}\right) = \frac{\mathrm{e}^{F6^{k}}}{\sum_{i=1}^{10} \mathrm{e}^{F6^{i}}} \tag{3.6}$$

The softmax activation function produces the final output of the neurons in the range [0, 1] and all outputs add up to 1. Each of the output represents the probability of the input belonging to a particular class.

Here, Z^{k} is the vector of size 10 containing final output of the network.

Backward Pass

Loss Layer

During training, an input is fed to the network and output is obtained. The obtained output is compared against the actual output to calculate the error or loss. The calculated error is then used to update the weights of the network. The process is repeated until the error is minimized. The function which is used to calculate the error or loss is called cost/loss function, and there are quite a few loss functions available and for softmax layer the two commonly used loss functions are Mean Squared Error (MSE)

and cross-entropy loss function. The Mean Squared Error (MSE) of the CNN model can be given as

$$\text{loss} = E(Z, \text{target})$$

The loss function E is defined as

$$E(Z, \text{target}) = \frac{1}{10}\sum_{k=1}^{10}\left(Z^k - \text{target}^k\right)^2 \tag{3.7}$$

Z^k is the kth output (in this case the network will produce 10 outputs representing class probabilities) generated by the CNN and target^k is the ground truth of the input.

$E(Z, \text{target})$ is the error/loss which represents how far the prediction of the network is from the actual target.

The purpose of training is to minimize the loss function and to do so the weights of the network are continuously updated until the minimum value is achieved. After calculating the error, the gradient of the loss function with respect to each weight of the neural network is calculated. The weights of the network are then updated with their respective gradients. This process is continued until the total error is minimized.

To minimize the loss function E, derivative of E is calculated w.r.t weight w^i. Since the loss function E is defined as composition of functions in series as

$$E = \text{loss}(\text{softmax}(F6(C5(P4(C3(P2(C1(input))))))))$$

That is, the loss function composes of the softmax activation function, which is a function of the fully connected layer $F6$ which in turn is a function of previous layer $C5$ and so on.

The loss function E is differentiable and can be differentiated w.r.t any weight at any layer using chain rule. In backpropagation, the weights of the last layer are updated first followed by second last layer and so on. To start with, let us find the derivative of the cost function *w.r.t* weight w_i^k at last layer $F6$ using chain rule.

$$\frac{\partial E}{\partial w_i^k} = \frac{\partial E}{\partial Z^k} * \frac{\partial Z^k}{\partial F6^k} * \frac{\partial F6^k}{\partial w_i^k} \tag{3.8}$$

- E is the cost function.
- Z^k represents the output of softmax function.
- $F6^k$ is the output of the last layer.
- k is the output layer neuron index.
- w_i^k is the ith weight of kth neuron of last layer.

After finding the derivative $\frac{\partial E}{\partial w_i^k}$, the weight w_i^k is then updated as

$$w_i^k = w_i^k - \mu\frac{\partial E}{\partial w_i^k} \tag{3.9}$$

where w_i^k is the updated weight and μ is the learning rate.

To solve Eq. 3.8, we first need to find the individual derivatives $\frac{\partial E}{\partial Z^k}$, $\frac{\partial Z^k}{\partial F6^k}$ and $\frac{\partial F6^k}{\partial w_i^k}$. which are given as

(i) $\frac{\partial E}{\partial Z^k} = \frac{\partial \frac{1}{10}\sum_{k=1}^{10}\left(Z^k - \text{target}^k\right)^2}{\partial Z^k} = \frac{1}{5}\left(Z^k - \text{target}^k\right)$

(ii) $\frac{\partial Z^k}{\partial F6^k} = Z^k * \left(1 - Z^k\right)$
Here, Z^k is the output of softmax function and the derivative of softmax function $f(x)$ is given as $f(x) * (1 - f(x))$ as shown in Sect. 3.3.3.

(iii) $\frac{\partial F6^k}{\partial w_i^k} = C5^i$

From Eq. 3.5

$$F6^k = \sum_{i=1}^{120} w_i^k * C5^i = \left(w_1^k * C5^1 + \cdots + w_i^k * C5^i + \cdots + w_{120}^k * C5^{120}\right)$$

The derivative of above function is given as

$$\frac{\partial F6^k}{\partial w_i^k} = \left(0 + \cdots + C5^i + \cdots + 0\right) = C5^i$$

Therefore,

$$\frac{\partial E}{\partial w_i^k} = \frac{1}{5}\left(Z^k - \text{target}^k\right) * Z^k * \left(1 - Z^k\right) * C5^i$$

Here, k is the neuron number in the last layer and i is the index of the weights to that neuron connecting input $C5^i$. For simplification in the backward pass and for propagation of error to previous layers, we calculate the deltas for this layer which is represented by

$$\delta F6^k = \frac{1}{5}\left(Z^k - \text{target}^k\right) * Z^k * \left(1 - Z^k\right)$$

These delta values enable the calculation of the gradient as well as the process of propagation of this error value to previous layers. The gradient then becomes as

$$\frac{\partial E}{\partial w_i^k} = \delta F6^k * C5^i$$

Here, i is the index of the weight to be updated and k is the neuron/filter number of layer $F6$.

The gradient $\frac{\partial E}{\partial w_i^k}$ calculated above can now be used to update the weight w_i^k using the formula

$$w_i^k = w_i^k - \mu \frac{\partial E}{\partial w_i^k}$$

Hidden Layer ($C5$)

The backward pass will continue to update the weights in hidden layer $C5$. The equations during forward pass at the layer are given as

$$C5^k = \sigma\left(\sum_{d=0}^{15}\sum_{m=0}^{4}\sum_{n=0}^{4} w_{m,n}^{k,d} * P4_{m,n}^{d} + b^k\right)$$

where $C5^k$ represents the 120 feature maps of convolution layer $C5$, (m, n) are the indices of kth filter and d is the channel number.

In order to update the weight $w_{m,n}^{k,d}$, we first have to backpropagate the error from the layer $F6$ to layer $C5$ and then calculate the deltas of layer $C5$. The error backpropagated at this layer $C5$ from layer $F6$ is given by this

$$\mathrm{e}^k = \sum_{l=1}^{10} \delta F6^l * w_k^l$$

Here, w_k^l is the weight of the $F6$ layer connecting output of the layer $C5$ to the layer $F6$; k is the index of the 120 neurons in layer $C5$, and l is the index of delta vector at layer $F6$.

The deltas for this layer are given by the formula

$$\delta C5^k = \mathrm{e}^k * \mathrm{ReLU}'\left(x^k\right)$$

$$\mathrm{ReLU}'\left(x^k\right) = \left\{0 \text{ if } x^k < 0, 1 \text{ if } x^k \text{ otherwise}\right\}$$

Here, x^k is the summation of the inputs to the neuron multiplied with the weights of this layer; it is the input to the neuron before the activation function during the forward pass, and k is the index of the number of filter at this layer.

The gradient of any weight in this layer $C5$ is given by

$$\frac{\partial E}{\partial w_{m,n}^{k,d}} = \delta C5^k * P4_{m,n}^{d}$$

Here, d is the channel number and (m, n) are the indices of filter weights; k is the filter number in this layer $C5$.

Hidden Layer ($C3$)

To calculate the gradient of any of the weights in this layer, we have to backpropagate the error from layer $C5$ through pooling layer $P4$ to layer $C3$.

$$e_{i,j}^{k} = \sum_{l=1}^{120} \delta C5^{l} * w_{i,j}^{l,k}$$

Here, $w_{i,j}^{l,k}$ is the weights of the next layer $C5$, l represents the filter number at that layer, k is the channel number/filter number connecting the output of layer $C3$ to $C5$, and (i, j) are the indices of the filter weights at $C5$, as well as the indices of the error matrix at $C3$ layer. Since we propagate error from $C5$ to $C3$ through $P4$ pooling layer, the error value is calculated for only those features which are selected during the max-pooling operation in forward pass.

This is used to calculate the delta values for this layer $C3$. These delta values are then used to calculate the gradient for any weight in this layer $C3$

$$\delta C3_{i,j}^{k} = e_{i,j}^{k} * \text{ReLU}'\left(x_{i,j}^{k}\right)$$

Here, $x_{i,j}^{k}$ is the summation of the summation of the inputs to the neuron multiplied with the weights of this layer; it is the result that is input to the activation function during the forward pass.

The gradient for the weights in this layer is calculated using the formula

$$\frac{\partial E}{\partial w_{m,n}^{k,d}} = \sum_{i=0}^{9} \sum_{j=0}^{9} \delta C3_{i,j}^{k} * P2_{m+i,n+j}^{d}$$

Here, $w_{m,n}^{k,d}$ is the weights of this layer, k is the filter number, d is the channel number, (m, n) are the indices of the weights and (i, j) are the indices of the delta matrix.

Hidden Layer ($C1$)

To calculate the gradient of any of the weights in this layer, we have to backpropagate the error from layer $C3$ through pooling layer $P2$ to layer $C1$. This operation is implemented by full convolution of the delta matrix with the 180° flipped weight matrix.

$$e^{k} = \sum_{l=1}^{16} \delta C3^{l} * w^{l,k}(\text{flipped } 180°)$$

Here, e^{k} is the error matrix for each of the k filters; k goes from 1 to 6 as there are six filters in layer $C1$. The size of this error matrix is equal to the size of output of this layer, which is 28×28. The indices (i, j) for each error value of every matrix e^{k} go from 0 to 27. This is used to calculate the delta values for this layer $C1$. These delta values are then used to calculate the gradient for any weight in this layer $C1$

$$\delta C1_{i,j}^{k} = e_{i,j}^{k} * \text{ReLU}'\left(x_{i,j}^{k}\right)$$

Here, $x_{i,j}^{k}$ is the summation of the inputs to the neuron multiplied with the weights of this layer; it is the result that is input to the activation function during the forward pass, and (i, j) is the index of the delta matrix which goes from 0 to 27.

The gradient for the weights in this layer is calculated using the formula

$$\frac{\partial E}{\partial w_{m,n}^{k}} = \sum_{i=0}^{27} \sum_{j=0}^{27} \delta C1_{i,j}^{k} * \text{input image}_{m+i,n+j}$$

Here, $w_{m,n}^{k}$ is the weights of this layer, k is the filter number, (m, n) are the indices of the weights and (i, j) are the indices of the delta matrix.

3.3 Loss Functions and Softmax Classifier

A loss function is used to calculate the error (difference between prediction and ground truth label) during the training process of a deep network.

Depending upon the application, there are many loss functions that can be used in deep learning networks. For example, mean squared error ($L2$) loss, cross-entropy loss, and hinge loss are commonly used in classification problem. Absolute deviation error ($L1$) loss is suitable for regression problem. Some of the commonly used loss functions are discussed below.

3.3.1 Mean Squared Error (L2) Loss

The most commonly used loss function in machine learning is Mean Squared Error (MSE) loss function. The MSE function, also known as $L2$ loss function, calculates the squared average error E of all the individual errors, and is given by

$$E = \frac{1}{n} \sum_{i=1}^{n} \mathrm{e}_{i}^{2}$$

where e_i represents the individual error of ith output neuron which is given by

$$e_i = \text{target}(i) - \text{output}(i)$$

During training process, a loss function is used at the output layer to calculate the error and its derivative (gradient) is propagated in the backward direction of the network. The weights of the network are then updated with their respective gradients.

3.3.2 Cross-Entropy Loss

Cross-entropy loss is another loss function mostly used in regression and classification problems. Cross-entropy loss is given by

$$H(y) = -\sum_i y_i' \log(y_i)$$

where y_i' is the target label, and y_i is the output of the classifier. Cross-entropy loss function is used when the output is a probability distribution, and thus it is preferred loss function for softmax classifier.

3.3.3 Softmax Classifier

Softmax classifier is a mathematical function which takes an input vector and produces output vector in range (0–1), where the elements of the output vector add up to 1. That is, the sum of all the outputs of softmax function is 1. Softmax function is given by

$$S(y_i) = \frac{e^{y_i}}{\sum_j e^{y_j}}$$

Since softmax function outputs probability distribution, it is useful in the final layer of deep neural networks for multiclass classification.

During backpropagation, the derivative of this loss function is calculated using quotient rule

$$\frac{dS(y_i)}{dy_i} = \frac{\left(e^{y_i} \cdot \sum_{j=1}^{n} e^{y_j}\right) - (e^{y_i} \cdot e^{y_i})}{\left(\sum_{j=1}^{n} e^{y_j}\right)^2}$$

$$\frac{dS(y_i)}{dy_i} = \frac{\left(e^{y_i} \cdot \sum_{i}^{n} e^{y_j}\right)}{\left(\sum_{j=1}^{n} e^{y_j}\right)^2} - \frac{(e^{y_i} \cdot e^{y_i})}{\left(\sum_{j=1}^{n} e^{y_j}\right)^2}$$

$$=> \frac{dS(y_i)}{dy_i} = \frac{(e^{y_i})}{\sum_{j=1}^{n} e^{y_j}} - \frac{(e^{y_i} \cdot e^{y_i})}{\left(\sum_{j=1}^{n} e^{y_j}\right)^2}$$

$$=> \frac{dS(y_i)}{dy_i} = \frac{(e^{y_i})}{\sum_{j=1}^{n} e^{y_j}} - \left(\frac{e^{y_i}}{\left(\sum_{j=1}^{n} e^{y_j}\right)}\right)^2$$

$$=> \frac{dS(y_i)}{dy_i} = S(y_i) - (S(y_i))^2$$

$$=> \frac{dS(y_i)}{dy_i} = S(y_i) \cdot (1 - (S(y_i))$$

$$\text{since } \frac{(e^{y_i})}{\sum_{j=1}^{n} e^{y_j}} = S(y_i)$$

Similarly, derivative of $S(y_i)$ w.r.t y_k is given as

$$\frac{dS(y_i)}{dy_k} = \frac{\left(0 \cdot \sum_{j=1}^{n} e^{y_j}\right) - (e^{y_i} \cdot e^{y_k})}{\left(\sum_{j=1}^{n} e^{y_j}\right)^2}$$

since $\frac{de^{y_i}}{dy_k} = 0$ as e^{y_i} is constant here.

$$\frac{dS(y_i)}{dy_k} = -\frac{(e^{y_i} \cdot e^{y_k})}{\left(\sum_{j=1}^{n} e^{y_j}\right)^2}$$

$$\frac{dS(y_i)}{dy_k} = -\frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}} \cdot \frac{e^{y_k}}{\sum_{j=1}^{n} e^{y_j}}$$

$$\frac{dS(y_i)}{dy_k} = -S(y_i) \cdot S(y_k)$$

The derivative of $S(y_i)$ is then used in Eq. 3.9 above to update the weights.

The gradient calculations are used to update the weights in such a way that the total error is minimized. The simplest way to minimize the error is to use various gradient-based optimization techniques which are briefly discussed in the next section.

3.4 Gradient Descent-Based Optimization Techniques

Gradient descent is an optimization technique used to minimize/maximize the cost function by calculating gradients necessary to update the values of the parameters of the network. Various variants of this optimizing technique define how to calculate the parameter updates using these gradients.

3.4.1 Gradient Descent Variants

There are three commonly used Gradient Descent (GD) variants. These variants differ in how many training examples are used to compute the gradient. The three variants are explained as follows.

3.4.1.1 Batch Gradient Descent (GD)

In traditional Gradient Descent (GD), also known as batch gradient descent, error gradient with respect to weight parameter w is computed for the entire training set followed by updating the weight parameter as shown below:

$$w = w - \mu \cdot \nabla \mathbf{E}(\mathbf{w})$$

where $\nabla \mathbf{E}(\mathbf{w})$ is the error gradient with respect to weight w and $\boldsymbol{\mu}$ is the learning rate that defines the step size to take along the gradient. The learning rate is a hyperparameter which cannot be too high or too low. Large value of learning rate can miss the optimum value, and too low learning rate will result in slow training time.

The training set often contains hundreds and thousands of examples that may demand huge memory which makes it difficult to fit in the memory. As a result, computing the error gradient can be very slow.

3.4.1.2 Stochastic Gradient Descent (SGD)

The above problem can be rectified by using Stochastic Gradient Descent (SGD), also known as incremental gradient descent, where gradient is computed for one training example at a time followed by updating of parameter values. It is usually much faster than standard gradient descent as it performs one update at a time.

$$w = w - \mu . \nabla \mathbf{E}(\mathbf{w}; \mathbf{x(i)}; \mathbf{y(i)})$$

where $\nabla \mathbf{E}(\mathbf{w}; \mathbf{x(i)}; \mathbf{y(i)})$ is the gradient of loss function—$E(w)$ w.r.t parameters w, for the training example $\{x(i), y(i)\}$.

In SGD, the one example based on updation of the parameter values causes the loss function to fluctuate frequently.

3.4.1.3 Mini-batch Gradient Descent

Mini-batch gradient descent also known as mini-batch SGD is a combination of both standard gradient descent and SGD techniques. Mini-batch SGD divides the entire training set into mini-batches of n training examples and performs the updating of parameter values for each mini-batch. This type of gradient descent technique takes advantage of both standard gradient descent and SGD techniques, and is commonly used optimization technique in deep learning.

$$w = w - \mu \cdot \nabla \mathbf{E}(\mathbf{w}; \mathbf{x}(\mathbf{i} : \mathbf{i} + \mathbf{n}); \mathbf{y}(\mathbf{i} : \mathbf{i} + \mathbf{n}))$$

Typical mini-batch size varies from 50 to 256 and should also be chosen sensibly according to the following factors:

- Large batch sizes provide more accurate gradients but have high memory requirements.
- Small batch sizes can offer a regularizing effect but require a small learning rate to maintain stability owing to the high variance in the estimate of the gradient. This in turn increases the training time because of the reduced learning rate.

3.4.2 Improving Gradient Descent for Faster Convergence

The main objective of optimization is to minimize the cost/loss or objective function. There are many methods available that help an optimization algorithm to converge faster. Some of the commonly used methods are discussed below.

3.4.2.1 AdaGrad

In SGD, the learning rate is set independently of gradients which may sometimes cause problems. For example, if the gradient is large, large learning rate would result in large step size, which means it may not achieve the optimum value as it may keep oscillating around the optimum value, and if the magnitude of gradient is small, a small learning rate may result in slow convergence. The problem can be resolved by using some adaptive approach for setting the learning rate. AdaGrad is one such adaptive model which uses adaptive learning rate by adding squared norms of previous gradients and dividing the learning rate by the square root of this sum.

$$w_{t+1,i} = w_{t,i} - \frac{\mu}{\sqrt{G_i}} \cdot \nabla_{t,i}$$

where $\nabla_{t,i}$ is the gradient of loss function with respect to parameter w_i and $G_i = \sum_{\tau=1}^{t} \nabla_\tau^2$ and ∇_τ is gradient at iteration τ.

In this way, parameters with high gradients will have small effective learning rate and parameters with small gradients receive increased effective learning rate.

The main advantage of AdaGrad is that learning rate is automatically adjusted, and there is no need to manually tune it. However, the sum in the denominator keeps on increasing which gradually causes the learning rate to decay. This decaying learning rate can slow down the learning or stop the learning completely.

3.4.2.2 AdaDelta

AdaDelta is a modified version of AdaGrad which overcomes the problem of decaying leaning rate. AdaDelta limits the number of previous gradients to some fixed size x and then the average of these past gradients is stored for efficiency. The average value $\text{Avg}(\nabla_t^2)$ at time t only depends on previous average and current gradient. The parameter update is then made as

$$w_{t+1} = w_t - \frac{\mu}{\sqrt{\text{Avg}(\nabla_t^2)}} \cdot \nabla_t$$

Since the denominator is just the Root Mean Square (RMS) of the parameters

$$w_{t+1} = w_t - \frac{\mu}{\text{RMS}(\nabla_t)} \cdot \nabla_t$$

3.4.2.3 RMSProp

The vanishing learning rate in AdaGrad can be rectified by using RMSProp. It is a modified version of AdaGrad which discards history from the distant past by introducing exponentially weighted moving average. RMSProp uses sign of the gradient instead of the magnitude of the gradient to update the weights. The working of RMSProp optimizer is as follows:

(a) Set same magnitude of updates for all weights. Set maximum and minimum allowable weight updates to $\Delta_{\max}$ and $\Delta_{\min}$, respectively.
(b) At each iteration, if signs of current gradient and previous gradient are same, then increase learning rate by a factor of 1.2, i.e. $\eta = \eta + 1.2$.
Therefore, the update Δ_{ij}^{t+1} becomes

$$\Delta_{ij}^{t+1} = \min\left(\eta + \Delta_{ij}^{t}, \Delta\text{max}\right)$$

(c) If signs of current gradient and previous gradient are different, then decrease the learning rate by a factor of 0.5, i.e., $\eta = \eta - 0.5$

$$\Delta_{ij}^{t+1} = \max\left(\eta - \Delta_{ij}^{t}, \Delta_{\min}\right)$$

3.4.2.4 Adam

Adaptive Moment Estimation (Adam) is an adaptive optimization technique that takes advantages of AdaGrad and RMSProp. Like AdaDelta and RMSProp, Adam saves an exponentially decaying average of previous squared gradients v_t. In addition to that, Adam also computes the average of the second moments of the gradients m_t.

m_t and v_t which are values of the mean and uncentered variance, respectively, are given as

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\boldsymbol{g}t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\boldsymbol{g}t^2$$

Adam updates exponential moving averages of the gradient and the squared gradient where the hyperparameters $\beta_1, \beta_2 \in [0, 1]$ control the decay rates of these moving averages.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

The final formula for update is given as

$$w_{t+1} = w_t - \frac{\mu}{\sqrt{\hat{v}_t}} \cdot \hat{m}_t$$

Adam slightly performs better than other adaptive techniques and converges very fast. It also overcomes the problems faced by other optimization techniques such as decaying learning rate, high variance in updates, and slow convergence.

Note that a smoothing term 'e' is usually added in the denominator of the above fast converging methods to avoid divide by 0.

3.5 Challenges in Training Deep Networks

Training a deep neural network is a challenging task, and some of the prominent challenges in training deep models are discussed below.

3.5.1 Vanishing Gradient

Any deep neural network with activation function like sigmoid, tan*h,* etc. and training through backpropagation suffers from *vanishing gradient* problem. Vanishing gradient makes it very hard to train and update the parameters of the initial layers in the network. This problem worsens as the number of layers in the network increases. The aim of backpropagation in neural networks is to update the parameters such that the error of the network is minimized and actual output gets closer to the target output. During backpropagation, the weights are updated using gradient descent (rate

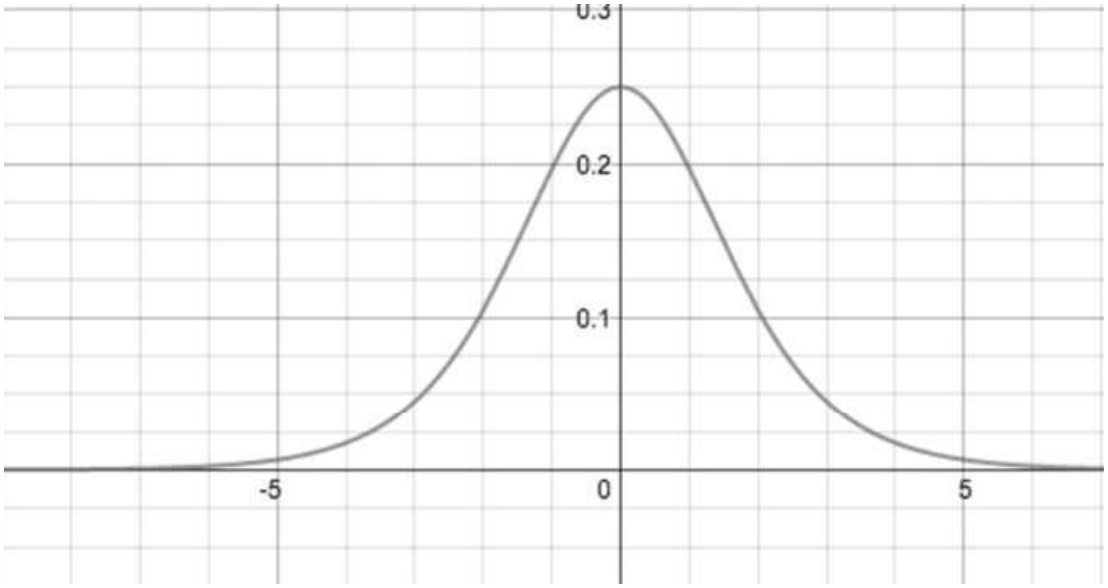

Fig. 3.2 Graph of derivative of sigmoid function

of change in total error E with respect to any weight w). In deep networks, these gradients determine how much each weight should change. The gradients become smaller as they propagate through many layers. The sigmoid function is given by

$$f(x) = \frac{1}{1 + e^{-x}}$$

The derivative of this sigmoid function is given as

$$f'(x) = \frac{1}{1 + e^{-x}}\left(1 - \frac{1}{1 + e^{-x}}\right)$$

The graph of the above equation is given in Fig. 3.2. It is evident from the graph that the maximum point of the function is 0.25 implying that the output of the derivative of the cost function will always lie between 0 and 0.25. In other words, the errors will be squeezed to the range 0 and 0.25 at each layer. Therefore, the gradients become smaller and smaller after each layer and finally vanish leaving top layers untrained.

The vanishing gradient is the primary reason that makes sigmoid or tanh activations unsuitable for deep networks, and this is where Rectified Linear Units (ReLUs) come to the rescue. ReLU activation function does not suffer from vanishing gradient because there is no squeezing of inputs as the derivative is always 1 for positive inputs. A Rectified Linear Unit (ReLU) outputs 0 for input less than 0 and raw output otherwise. That is, if the input x is less than 0, then the output is 0 and if x is greater than 0, the output is equal to the input x and its derivative is 1.

That is $\boldsymbol{f(x) = x}$ and $\mathbf{f'(x) = 1}$ for $\mathbf{x > 0}$.

3.5.2 Training Data Size

Deep networks use training data for learning and are capable of learning complex nonlinear relationships between input data and output label. Deep networks require a large number of parameters to be learnt before it can be utilized to deliver the desired result. The number of parameters in deep models is large. More complex models mean more powerful abstraction, and more parameters which require more data. So, the size of training data is an important factor which can influence the success of deep models. In fact, all the successful deep models have been trained on some very large dataset. For example, AlexNet, GoogleNet, VGG, ResNet, etc. all have been trained on a vast dataset of images called ImageNet. ImageNet is an image dataset which contains around 1.2 million labeled images distributed over 1000 classes. However, one can argue that deep models for object recognition and detection require large number of parameters to deal with different variations, different poses, different variations in color, etc., and thus require vast size dataset for training. On the other hand, less complex problems (like classification of medical images) where the variations are very small as compared to variations mentioned above can be solved using less complex models which do not require huge training datasets. The claim is true to some extent, but model complexity alone cannot decide the size of the data required for training. Training data quality also plays an import role in it. Noisy data means low Signal-to-Noise Ratio (SNR) in the data and lower SNR means more data is required for convergence. Therefore, the size of dataset really depends on the complexity of the problem being studied, and the quality of data.

Irrespective of the complexity of the task, large training data size can significantly improve the performance of deep models. That is, the larger the training data size, the better the accuracy. But the question "How much data is enough?" still remains unanswered, and there is no rule of thumb that can define exact number of examples required to train a particular deep model.

3.5.3 Overfitting and Underfitting

Once a model is trained on a training dataset, it is expected to perform well on new, previously unseen data which was not present during learning. The ability of a machine learning model to perform well on new and unseen data is called **generalization**. Generalization is one of the objectives of a good deep learning model. To estimate the generalization ability of a deep learning model, it is tested on data collected separately from the training set.

Deep learning models can suffer from two problems, viz. overfitting and underfitting and both can lead to poor model performance.

Overfitting occurs when a model is trained, and it performs so well on training data that it is unable to generalize to new data. That is, the model has low training

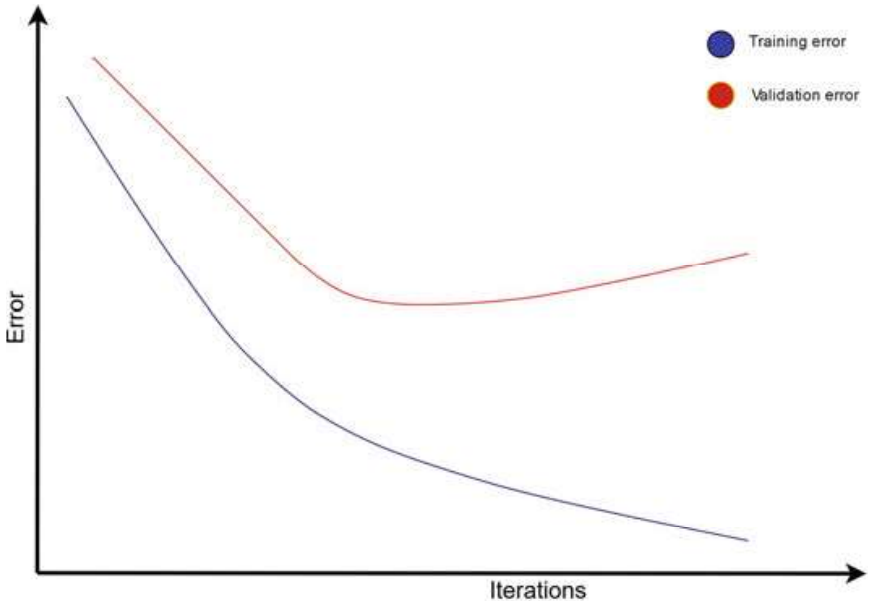

Fig. 3.3 Overfitting: training error (blue), validation error (red) as a function of the number of iterations

error but is unable to achieve low test error. In this case, the model is memorizing the data instead of learning. See Fig. 3.3.

Underfitting occurs when a model is not able to learn properly and its performance on the training set is poor.

The most common problem in deep learning is overfitting. Deep models like Convolutional Neural Network (ConvNet) have a large number of learnable parameters that must be learnt before they can be utilized to perform the task of interest. In order to train these models, extensively large training data with a specific end goal is required to accomplish the desired performance. If the training data is too small compared to the number of weights to be learned, then the network suffers from overfitting.

Overfitting is a common problem in deep networks, however, there are few techniques available that can be used in deep learning models to limit overfitting:

(a) Increase the training dataset.
(b) Reducing network size.
(c) Data augmentation: Modifying the current training data in a random way (Scaling, zooming, translation, etc.) to generate more training data.
(d) Interpolate weight penalties like $L1$ and $L2$ regularization and soft weight sharing.
(e) Dropout: The most popular technique to reduce overfitting is *Dropout*. Dropout refers to dropping out neurons/units in a neural network during training. Dropping a unit means temporarily detaching it from the network including all its

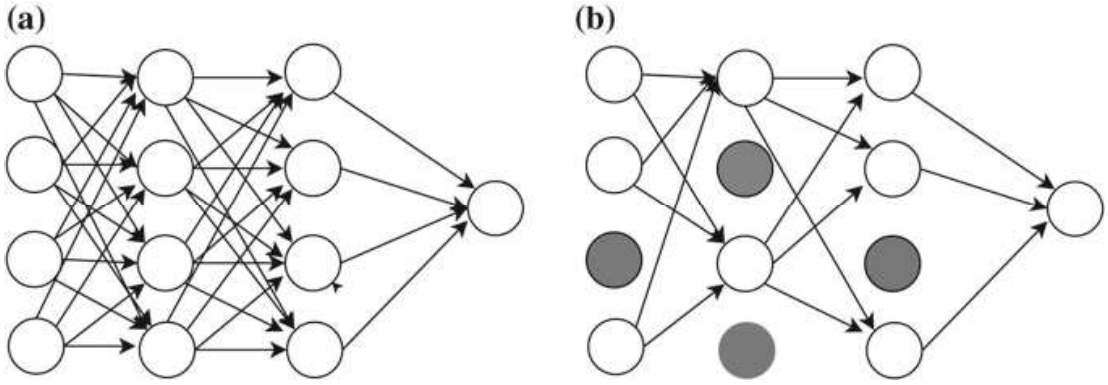

Fig. 3.4 **a** A simple neural network, **b** neural network after dropout

inward and outward connections (Fig. 3.4). The dropped-out neurons neither contribute to the forward pass nor do they contribute in backward pass. By using dropout, the network is forced to learn more robust features as network architecture changes with each input.

3.5.4 High-Performance Hardware

Training deep models on huge datasets require machines with sufficient processing power and memory. In order to get high efficiency and quick training time, it is highly recommended to use multi-core high-performance Graphics Processing Units (GPUs). These high-performance machines, GPUs, and memory are very costly and consume a lot of energy. Therefore, adopting deep learning solutions to real world becomes expensive and energy-consuming task.

3.6 Weight Initialization Techniques

One of the first tasks after designing a deep model is weight/parameter initialization. The weights which are learnt during training must have some initial values to start the training. Weight initialization is an important step which can have an intense effect on both the convergence rate and final accuracy of a network. Arbitrary initialization can lead to slow convergence or can completely freeze the learning process. A flawed or imperfect initialization of weights can impede the learning of a nonlinear system. Therefore, it is important to choose a robust technique to initialize the weights of a deep model. There are quite a few weight initialization techniques available each with its own weak features. Some initialization techniques are discussed below.

3.6.1 Initialize All Weights to 0

A simple way is to start from zero-valued weights and update these weights during training. This seems a sound idea but it has a problem associated with it. When all weights are initialized to 0, their derivative with respect to the loss function will be same for all weights. Thus, all the weights will have same value after successive iteration. This continues for all the training iterations making the model equivalent to a linear model.

3.6.2 Random Initialization

Another way is to initialize the weights with some random values (normal distribution) so that all weights are unique and compute distinct updates. This, however, suffers from two problems:

(a) Vanishing Gradients: If the weights are too small, their gradients will get smaller and smaller and finally vanish as they flow through different layers during back-propagation. This results in slow convergence and in worst case can freeze the learning process completely.
(b) Exploding Gradients: This is the opposite case of vanishing gradients. When the weights are too large, their gradients will be large as well causing large updates in the network weights. This results in unstable network as the weights keep oscillating around the minima. In worst case, the weights can become so large that overflow occurs and the output becomes NaN.

3.6.3 Random Weights from Probability Distribution

One good way to initialize weights is to assign the weights from a Gaussian distribution with zero mean and finite variance. With finite variance, the problem of vanishing and exploding gradients can be avoided. This type of technique in which weights are initialized such that variance remains same is called **Xavier initialization**.

For weight initialization, pick weights from a Gaussian distribution with zero mean and a variance of $1/N$, where N is the number of input neurons.

Xavier initialization is extensively used in neural networks with sigmoid activation function. Xavier initialization does not work well with ReLU activation function as it faces difficulties to converge. A modified version that uses

$$\text{var}(w_i) = \frac{2}{N}, \text{ instead of } \text{var}(w_i) = \frac{1}{N}$$

works well with ReLU activation function.

3.6.4 Transfer Learning

Another idea to initialize weights of a network is to transfer learnt weights from a trained model into the target model. In this way, no initialization actually takes place and the model gets previously learnt weights. This process of reusing or transferring weights learnt for one task into another similar task is called transfer learning. *Transfer learning thus refers to extracting the learnt weights from a trained base network* (*pretrained model*) *and transferring it to another untrained target network instead of training this target network from scratch*. In this way, features learned in one network are transferred and reused in another network designed to perform similar task. Transfer learning has gained popularity in deep learning especially on convolutional neural networks. It effectively reduces training time and also improves accuracy of models designed for tasks with minimum or inadequate training data. Transfer learning can be used in the following ways:

(a) *Pretrained model as fixed feature extractor*: In this scenario, the last fully connected layer (classifier layer) is replaced with a new linear classifier and this last layer is then trained on new dataset. In this way, the feature extraction layers remain fixed and only the classifier gets fine-tuned. This strategy is best suited when the new dataset is insufficient but similar to the original dataset.
(b) *Fine-tune whole Model*: Take a pretrained model, replace its last fully connected layer (classifier layer) with new fully connected layer and retrain the whole network on new dataset by continuing backpropagation up to the top layers. In this way, all the weights are fine-tuned for new task.

Transfer learning has many advantages and few drawbacks as well. Transfer learning is only possible when the two tasks have some amount of similarity. For totally different tasks, transfer learning may or may not work well. Furthermore, in order to use transfer learning, the two models should be compatible, i.e., base model and target model should have similar architecture.

3.7 Challenges and Future Research Direction

Many researchers have used deep networks and achieved good results in a wide range of applications. Despite this, application of deep networks in a given application poses many challenges and one of the challenges is training and optimization. Training deep models is not an easy task; just throwing raw training data and expecting that deep models will eventually learn by itself is not correct. Given right type of data and hyperparameters, a moderately simple model may perform well. But in general, learning an optimal deep model for a given application depends on various issues that need to be addressed carefully. Activation function, learning rates, weight initialization, data normalization, regularization, learning model structure, etc. all play an important role in the training process and to make appropriate selection or to

choose an optimal value of a parameter is a challenging task. In addition, optimizing a cost function in general is still a difficult task. There are a number of challenges associated with neural network optimization problem and among these vanishing gradients, exploding gradients, local minima, flat regions, and local versus global structure are most common. In addition, optimization techniques are said to have many theoretical limitations as well which also needs to be explored.

Bibliography

Cho, J., Lee, K., Shin, E., Choy, G., Do, S.: How much data is needed to train a medical image deep learning system to achieve necessary high accuracy? arXiv preprint arXiv:1511.06348. 19 Nov 2015

Dauphin, Y.N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., Bengio, Y.: Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In: Advances in Neural Information Processing Systems, pp. 2933–2941 (2014)

Erickson, B.J., Korfiatis, P., Kline, T.L., Akkus, Z., Philbrick, K., Weston, A.D.: Deep learning in radiology: does one size fit all? J. Am. Coll. Radiol. **15**(3), 521–526 (2018)

Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. Book in preparation for MIT Press. URL http://www.deeplearningbook.org (2016)

He, K., Zhang, X., Ren, S., Sun, J.: Delving deep into rectifiers: surpassing human-level performance on imagenet classification. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1026–1034 (2015)

Kingma, D.P., Adam, J.B.: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. 22 Dec 2014

Kumar, S.K.: On weight initialization in deep neural networks. arXiv preprint arXiv:1704.08863. 28 Apr 2017

Nowlan, S.J., Hinton, G.E.: Simplifying neural networks by soft weight-sharing. Neural Comput. **4**(4), 473–493 (1992)

Sussillo, D., Abbott, L.F.: Random walk initialization for training very deep feedforward networks. arXiv preprint arXiv:1412.6558. 19 Dec 2014

Wilson, D.R., Martinez, T.R.: The general inefficiency of batch training for gradient descent learning. Neural Netw. **16**(10), 1429–1451 (2003)

Chapter 4
Supervised Deep Learning Architectures

4.1 Introduction

Many supervised deep learning architectures have evolved over the last few years, achieving top scores on many tasks. Deep learning architectures can achieve high accuracy; sometimes, it can exceed human-level performance. Supervised training of convolutional neural networks, which contain many layers, is done by using a large set of labeled data. Some of the supervised CNN architectures proposed by researchers include LeNet-5, AlexNet, ZFNet, VGGNet, GoogleNet, ResNet, DenseNet, and CapsNet. These architectures are briefly discussed in this chapter.

4.2 LeNet-5

LeNet-5 is composed of seven layers which are fed by an input layer. The size of the input image used in LeNet-5 is 32 × 32 pixels. The values of the input pixels are normalized; as a result of this, the mean of the input tends to zero and variance roughly one, which accelerates the learning process. The architecture diagram of LeNet-5 is given in Fig. 4.1, and details of various layers are given in Table 4.1.

The first layer of LeNet-5 is the convolutional layer that produces six feature maps of size 28 × 28. There are 156 trainable parameters and 122,304 connections in the first convolutional layer.

The layer 2 performs down-sampling. There are 12 trainable parameters and 5880 connections in this layer. The third layer is a convolutional layer which produces 16 feature maps of size 10 × 10. The layer 4 is a subsampling layer that has 32 trainable parameters and 2000 connections.

The fifth layer is also a convolutional layer that produces 120 feature maps of size 1 × 1, and there are 48,120 connections in this layer. The layer 6 is a Fully Connected Layer (FCL) and has 84 outputs and 10,164 trainable parameters. The output layer

M. A. Wani et al., *Advances in Deep Learning*, Studies in Big Data 57,
https://doi.org/10.1007/978-981-13-6794-6_4

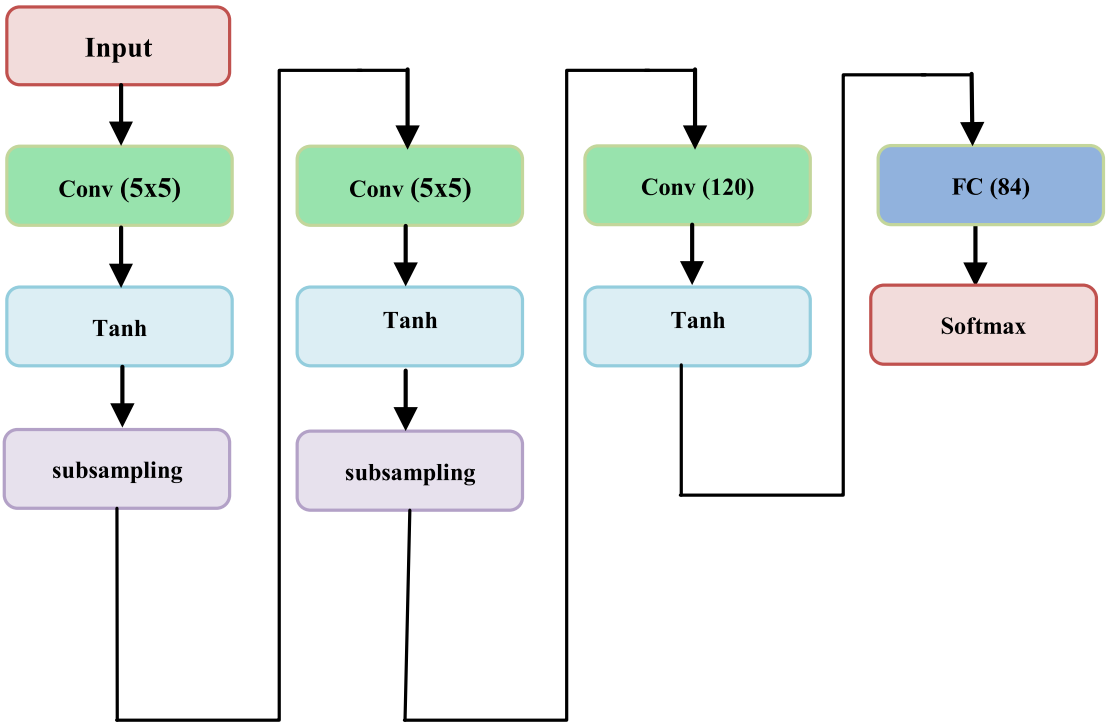

Fig. 4.1 Architecture diagram of LeNet-5

Table 4.1 Details of various layers of LeNet-5

Layer name	Input size	Filter size	Window size	# Filters	Stride	Padding	Output size	# Feature maps
Conv 1	32 × 32	5 × 5	–	6	1	0	28 × 28	6
Subsampling-1	28 × 28	–	2 × 2	–	2	0	14 × 14	6
Conv 2	14 × 14	5 × 5	–	16	1	0	10 × 10	16
Subsampling-2	10 × 10		2 × 2	16	2	0	5 × 5	16
Conv 3	5 × 5	5 × 5	–	120	1	0	1 × 1	120
Fully connected	120	–	–	–	–	–	1 × 1	84
Softmax	84	–	–	–	–	–	1 × 1	10

has radial basis function (Euclidean), one for each of the 10 output classes, and each output class is connected with 84 inputs.

The convolutional layers and the subsampling layers perform the task of feature extraction from the given image and the fully connected layers perform the task of classification. Figure 4.2 shows the handwritten digit "7" as input image. The target probability is 1 for the digit "7" and 0 for all the remaining nine digits. Therefore, the value of the target vector for digit "7" is given as [0, 0, 0, 0, 0, 0, 1, 0, 0]. Backpropagation is used to calculate the gradient of the loss function relating to all the weights in all layers of the CNN network.

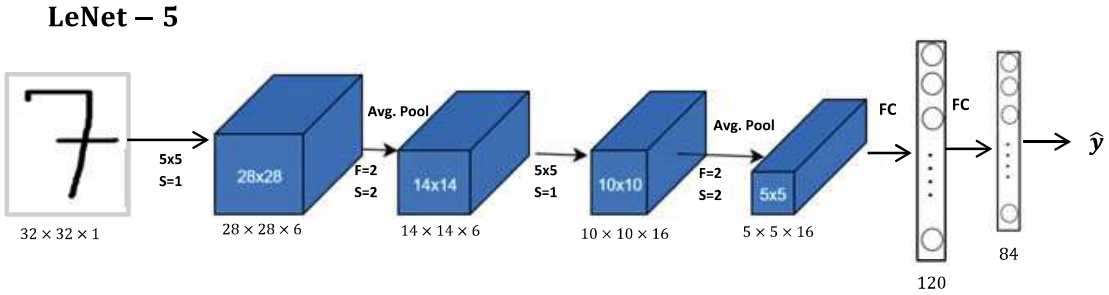

Fig. 4.2 Feature maps at various layers of LeNet-5

Training of LeNet-5:

Step 1: Random values are used for initialization of all filter parameters and weights.
Step 2: During this step, the input image goes through various layers, i.e., convolutional layers, subsampling layers, and fully connected layers. This step performs forward propagation which finds the output probabilities for all the classes in the network.
Step 3: The total error between output probabilities and target probabilities is calculated at the output layer of the network during this step.
Step 4: During this step, the error gradients with respect to weights in the network are calculated and gradient descent algorithm is used to update all weights and filter parameters to minimize the output error. The weights are adjusted proportionately depending on their contribution to the total error. Only the values of the connection weights and filter matrix are updated. The hyper-parameters like filter sizes and number of filters of the network are fixed and do not change during the training process.
Step 5: Steps 2–4 are repeated with all images present in the training set.

As shown in Table 4.1, the input image of size 32×32 is convolved with a filter of size 5×5 to generate six feature maps of size 28×28. Similarly, pooling is applied at layer 2 which down-samples the feature map of the first layer to 14×14. These feature maps of size 14×14 are again convolved with 16 filters of size 5×5 resulting in 16 feature maps of size 10×10. The result is again down-sampled to feature maps of size 5×5, and these feature maps are passed to a convolutional layer having a filter of size 5×5, which generates 120 feature maps of size 1×1. The output is then passed to a fully connected layer having 84 output neurons. The final classification layer in the network has ten neurons for ten classes.

4.3 AlexNet

One of the problems associated with training a deep neural network is vanishing gradient problem. This problem can be addressed if an activation function like Rectified Linear Unit (ReLU) is used. AlexNet is the first network which has used Rectified Linear Unit (ReLU) activation function.

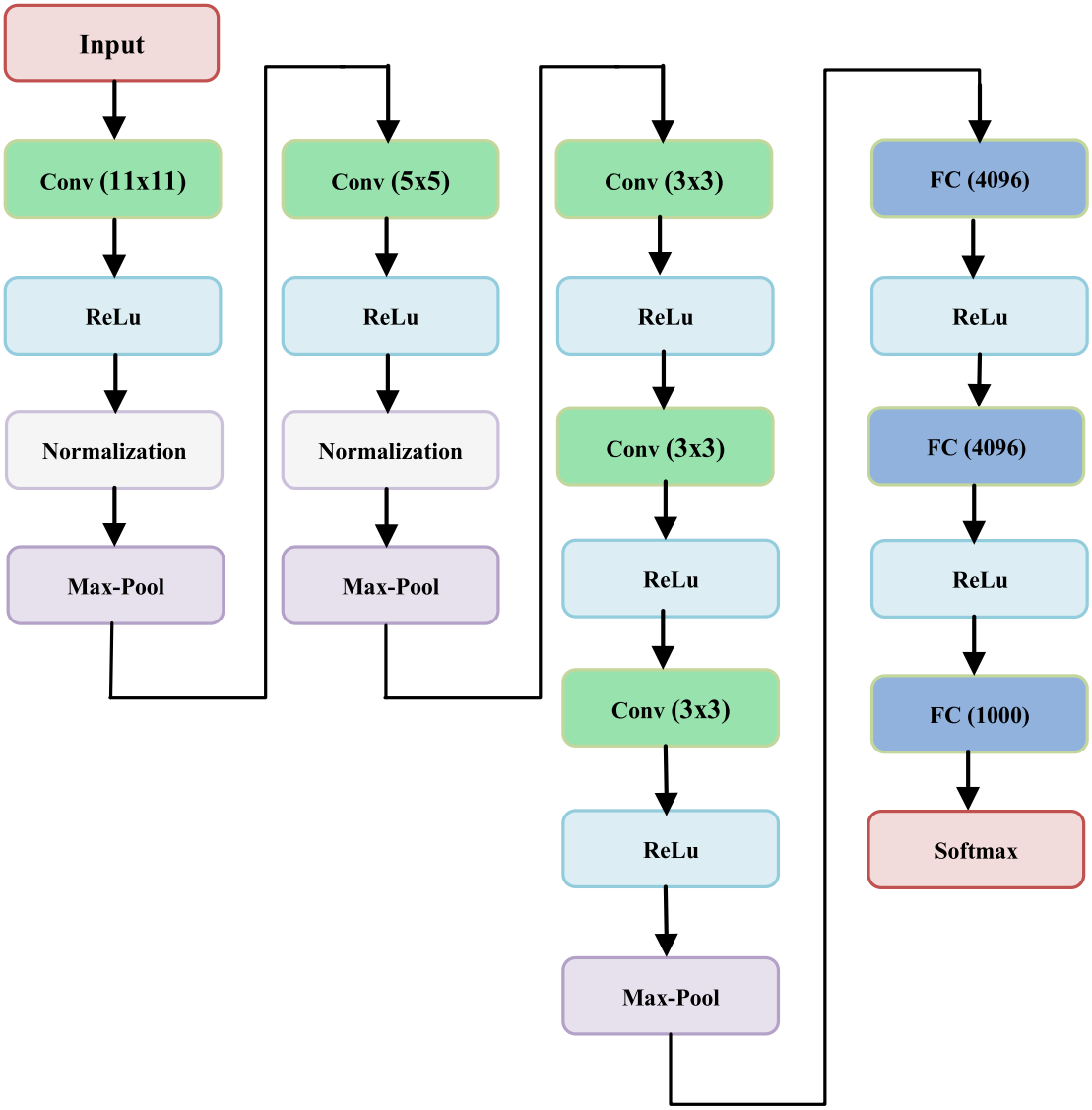

Fig. 4.3 Architecture diagram of AlexNet

The performance of AlexNet has been evaluated on ImageNet database. This database contains around 15 million high-resolution-labeled images of 22,000 subjects. AlexNet was trained on a subset of ImageNet database of around 1.2 million images of 1000 classes and tested on 150,000 images belonging to 1000 classes. The network was trained on GTX 580 GPU with 3 GB memory.

AlexNet is composed of many layers of convolutional, max-pooling, and fully connected layers. The first layer is convolutional layer, and the output of this layer is 96 images of size 55×55. The second layer is a max-pooling layer with output of size $27 \times 27 \times 96$ produced by using a window of size 3×3 with a stride of 2. The third layer is a convolutional layer with 256 filters of size 5×5 used with a stride of 1 and padding of 2. Layer 4 is a down-sampling layer also called max-pooling which produces an output of size $13 \times 13 \times 256$ by using a window of size 3×3 with a stride of 2. The layer 5 of this network is a convolutional layer that outputs

Table 4.2 Details of various layers of AlexNet

Layer name	Input size	Filter size	Window size	# Filters	Stride	Padding	Output size	# Feature maps
Conv 1	224 × 224	11 × 11	–	96	4	1	55 × 55	96
Max-pooling 1	55 × 55	–	3 × 3	–	2	0	27 × 27	96
Conv 2	27 × 27	5 × 5	–	256	1	2	27 × 27	256
Max-pooling 2	27 × 27	–	3 × 3	–	2	0	13 × 13	256
Conv 3	13 × 13	3 × 3	–	384	1	1	13 × 13	384
Conv 4	13 × 13	3 × 3	–	384	1	1	13 × 13	384
Conv 5	13 × 13	3 × 3	–	256	1	1	13 × 13	256
Max-pooling 3	13 × 13	–	3 × 3	–	2	0	6 × 6	256
Fully connected 1	4096 neurons							
Fully connected 2	4096 neurons							
Fully connected 3	1000 neurons							
Softmax	1000 Classes							

the feature maps of size 13 × 13 × 384 by using 384 filters of size 3 × 3 with a stride of 1 and a padding of 1. The sixth layer of this network is a convolutional layer that uses 384 filters of size 3 × 3 with a stride of 1 and padding of 1. Layer 7 is a convolutional layer that uses 256 filters of size 3 × 3, with a stride of 1, and padding of 1. The eighth layer is a down-sampling/max-pooling layer that uses a window of size 3 × 3 with a stride of 2. Layer 9 is the first fully connected layer with 4096 neurons. Layer 10 is a second fully connected layer with 4096 neurons. The eleventh layer of this network is a third fully connected layer with 1000 neurons. Final layer is a softmax layer. Figure 4.3 shows the architecture diagram of AlexNet, and the details of various layers of the AlexNet are given in Table 4.2.

AlexNet takes the input image of size 224 × 224, and the same is passed through five convolving layers, three fully connected layers, and then finally through softmax