CHAPTER 2

Convolutional Neural Network

LEARNING OBJECTIVES

After reading this chapter, you will be able to

- Understand the basics of convolutional neural network and various components in its architecture.
- Analyze the effect of different activation functions of a CNN unit.
- Understand CNN properties, architectural variants and their applications.

2.1 | INTRODUCTION

A Convolutional Neural Network (CNN), or ConvNet, is a class of deep, feed-forward artificial neural networks, most commonly applied to analyze visual imagery. It uses comparatively little preprocessing compared to other image classification algorithms. Convolutional networks were inspired by biological processes as the connectivity pattern between neurons looks like the organization of an animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the *receptive field*. The receptive fields of different neurons partially overlap such that they cover the entire visual field.

CNNs, like neural networks, are made up of neurons with learnable weights and biases. Each neuron receives numerous inputs, takes a weighted sum over them, passes it through an activation function, and responds with an output. CNN showed the most efficacies in image and video recognition, recommender systems, image classification, medical image analysis, and natural language processing. Thus, CNNs assure promising applications in deep learning systems. Computer vision through CNNs has several applications such as self-driving cars and robotics.

The main operation in a CNN system is mathematical convolution. Convolution is a simple mathematical operation; it means to roll together multiple data or functions that represent data. Convolution is a measure of the overlapping of two functions.

Like other neural network architectures, a CNN also has an input layer, number of hidden layers, and an output layer, some of which are convolutional layers using a mathematical model to pass on the results to successive layers.

The first layer distinguishes basic attributes like lines and curves. The next layer takes in more complex mathematical features. At higher levels, the brain recognizes that the image considered is an object with the configuration of edges and colors. Similarly, a CNN processes an image using weight matrix (also called filters or features) that detect specific attributes of the image such as

diagonal edges, vertical edges, curves, etc. As the image progresses through each layer, the filters recognize more complex attributes as well. In the case of a CNN, the convolution is performed on the input data with the use of a filter or kernel to then produce a feature map.

We execute a convolution by sliding the filter over the input. At every location, a matrix multiplication is performed and the result is summed onto the feature map. The input image is given as a matrix or as an tensor to the CNN. Since the input image cannot be fed directly into a CNN system, suitable processing has to be done on the image. We first extract the features from an image. These features are then provided as input to the CNN system. On further downsampling, a list of kernels or filters are applied by convoluting such a filter with the image.

Figure 2.1 shows the convolved image after applying filters such as edge detection, smoothing, Gaussian blur with 15 × 15 kernel, Median blur with 15 × 15 kernel, sharpening using blur, and Laplacian filter.

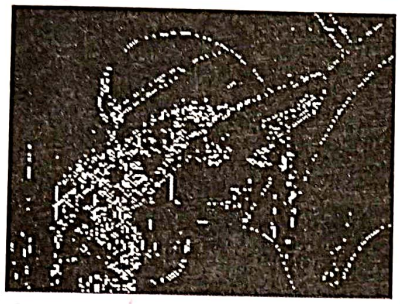
2.2 | COMPONENTS OF CNN ARCHITECTURE

A CNN is shaped as a hierarchical structure for fast feature extraction and classification. The main objective is to extract the input image volume and convert it into an output volume that holds class scores. A differential function is used for image processing. CNN consists of a stack of convolutional and subsampling layers, followed by a series of fully connected layers. Figure 2.2 represents the structure of a CNN.

The various layers composing a CNN are as follows:

1. **Convolutional Layer:** This layer is used for feature extraction, obtaining original features from the volume map, extracting data, and creating feature maps.
2. **Pooling or Downsampling Layer:** This layer reduces the number of weights and controls overfitting.
3. **Flattening Layer:** This layer prepares the CNN output to be fed to a fully connected neural network. It should be noted that CNN is not fully connected like a traditional neural network.
4. **Fully Connected Layer:** These are the layers at the top of CNN hierarchy. As mentioned above, neurons in the other layers in a CNN are not *fully connected*, that is, each neuron is connected with all activations in the previous layers. The fully connected layers are responsible to sum up the all the detections made in previous layers. They detect global characteristics of the input using the features detected in the lower layers.

A basic CNN is usually made of these four layers, but there is no restriction on the total number of layers. The main concern when dealing with CNN is to find the right kernel or right features needed. Convolution techniques are very efficient in finding the features of images if the right kernel is used. The usual method takes in every pixel of the image as a feature and thus also as an input node. The result from each convolution is placed in the next hidden layer node. A typical network consists of four-layered convolution network followed by a regular neural network which is provided into a logistic processor.

Operation	Convolved Image
Original image: Lenna	
Edge detection	
Smoothing using 5 × 5 kernel	
Gaussian blur with 15 × 15 kernel	

Operation	Convolved Image
Median blur with 15 × 15 kernel	
Sharpening using blur	
Laplacian filter	

FIGURE 2.1 Convolved image after applying filters.

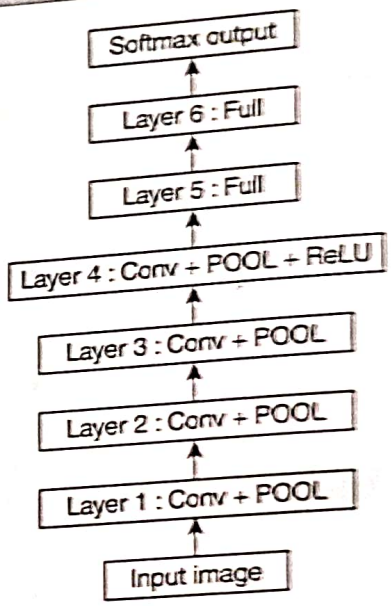


FIGURE 2.2 Structure of a CNN.

2.2.1 | Convolution Layer

The convolution layer is the basic layer that builds a CNN. Convolution is a mathematical operation on two functions to produce a third function. Filters or kernels are the base units in this system. These layers consist of a series of learnable filters, which have a small receptive field, but extend through the full depth of the input volume. During the forward pass, each filter is convolved across the input volume. The input volume contains the width and height. Convolution is done by computing the dot product between the entries of the filter and the volume. Thus, the networks acquire the properties of the filter and they can detect a specific type of feature at a particular position in the input. Let us understand the process of convolution with an example.

2.1 EXAMPLE

Consider the input matrix in Fig. 2.3. The input matrix is convolved with the filter to produce a feature map. Since the shape of the filter is 3 × 3, this convolution is called 3 × 3 convolution. Similarly, it can be 5 × 5 or 1 × 1 convolution, depending on the application.

0	1	1	0	0
0	0	1	0	0
0	1	1	0	0
0	0	1	0	0
0	1	1	0	0

(a)

0	1	1
0	0	1
0	1	1

(b)

FIGURE 2.3 (a) Input matrix; (b) filter or kernel.

The following are a few parameters that can be adjusted for a kernel:

1. **Size:** The size of the filter (e.g., 5 × 5).

2.1 EXAMPLE (Continued)

2. **Stride:** The rate at which kernel passes (a stride of 2 will move the kernel by 2 increments).
3. **Padding:** Zero-padding on the outside of the image to make sure that kernel pass is perfect on the edges.
4. **Output layers:** Number of kernels applied.

All 3 × 3 subsets of the input are convolved with the 3 × 3 filter to produce a feature map. The first 3 × 3 subset of the input matrix is matrix-multiplied with the filter and summed up to get the first value of the feature map.

The blue [grey in Fig. 2.4(a)] area is the current subset of the input feature being convolved. This is the receptive field. Once the first value is obtained, the filter is slided to the right for further computation.

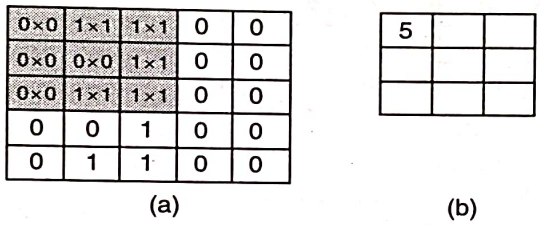


FIGURE 2.4 (a) First subset of the input feature; (b) first value of the feature map.

In Fig. 2.5, the receptive field is changed to the next 3 × 3 subset and convolved to get the next value of the feature map. Once the second value is obtained, the filter is again slided to the right.

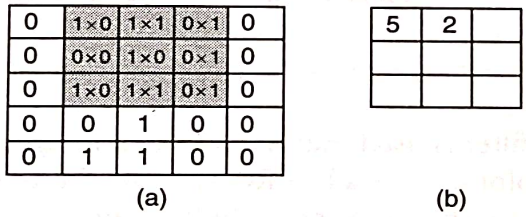


FIGURE 2.5 (a) Second subset of the input feature; (b) second value of the feature map.

Again, the convolution of the third receptive field and filter produces the third value of the feature map. By repeating these steps, the big input feature is reduced to a small feature map which implies the properties of the input. In the above example, we use a 3 × 3 filter that can be placed over the 5 × 5 input in 9 ways. So, we represent the features of the 5 × 5 matrix in a 3 × 3 feature map. In a different example, if we use a 2 × 2 filter, it can be placed over 16 different ways over the input matrix. So the 5 × 5 input matrix is reduced to 4 × 4 matrix in this case. Figure 2.6 shows the step-by-step visualization of the entire convolution.

2.1 EXAMPLE (Continued)

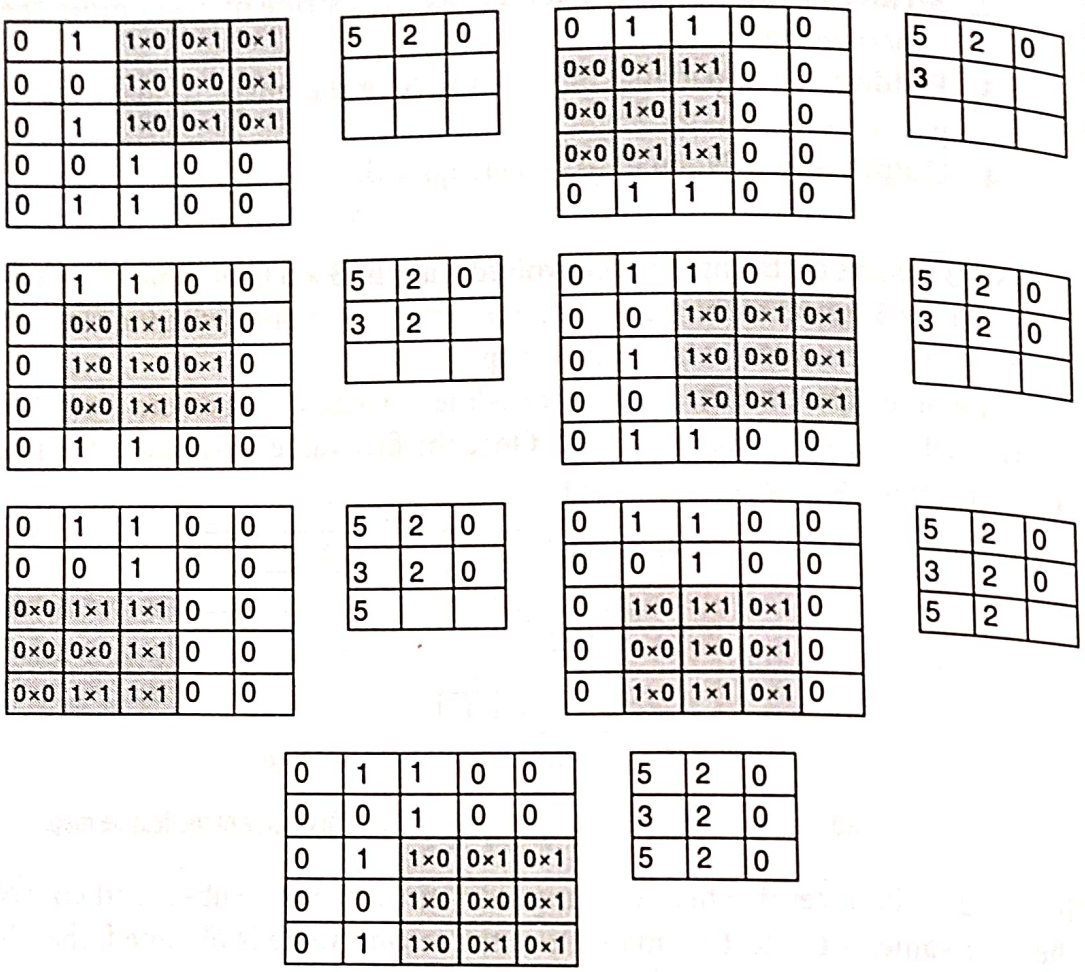


FIGURE 2.6 Complete process of convolution.

In this example, a 2D 3 × 3 filter is used. But in reality, an image is represented as a 3D matrix with width, height, and depth (color channels like RGB) as the dimensions; hence the filter used should also be 3D (i.e., the filter depth should be equal to the input channel size). Usually, many filters are applied on the input image which results in multiple distinct feature maps that are stacked together. These feature maps become the output of the convolution layer as shown in Fig. 2.6. Multiple convolutions are performed independently to get disjoint maps as represented in Fig. 2.7. As seen in the figure, the first layer filters convolve around the input matrix, and tries to identify a specific feature. When the feature is found, it is recorded in a feature map. Likewise, independent disjoint maps are created for each unique feature.

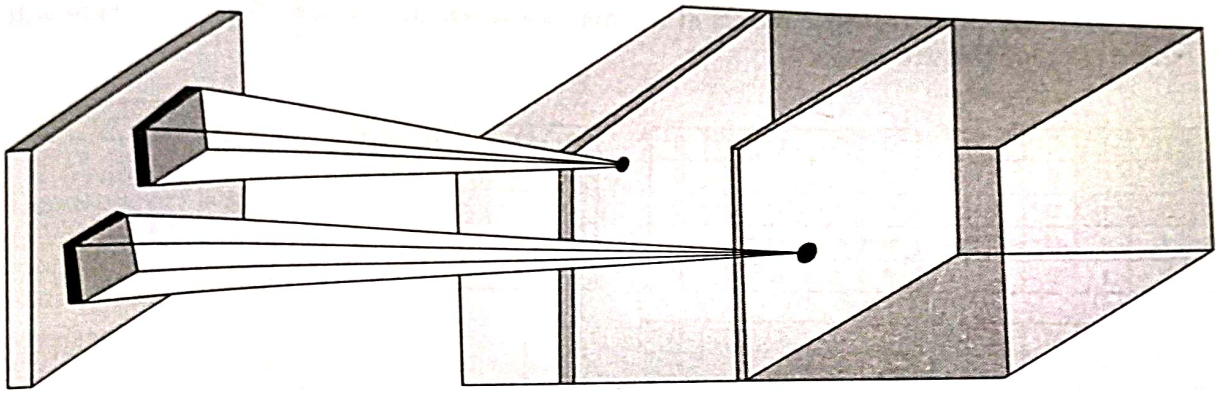


FIGURE 2.7 Independent convolutions producing disjoint maps.

2.2.1.1 Receptive Field

If there is high dimensional input image, then it is impossible to connect all neurons with all possible regions of the input size. This would result in the need to train too many weights resulting in high computational complexity. As an alternative, each neuron is connected only to a local region of the input volume. The spatial extent of this connectivity is a hyperparameter called the receptive field of the neuron. Consider the following example: the size of input image is [32 × 32 × 3]. If the filter size is 5 × 5, then each neuron in the convolution layer will need to be trained for 5 × 5 × 3 = 75 weights (and +1 bias parameter). The extent of the connectivity along the depth axis must be 3, since this is the depth of the input volume. The number of output features in each dimension is calculated based on the number of input features and the convolution properties using the following formula:

$$n_{out} = \left[\frac{n_{in} + 2p - k}{s}\right] + 1$$

where n_{in} is the number of input features, n_{out} is the number of output features, k is the convolution kernel size, p is the convolution padding size, and s is the convolution stride size.

2.2.1.2 Feature Map

The size of the feature map (convolved feature) is controlled by three parameters, namely, depth, stride, and zero-padding. These parameters have to be decided before the convolution step is performed:

1. **Depth:** Depth corresponds to the number of filters used for the convolution operation. If convolution is performed on an original image using n distinct filters, then it produces n different feature maps. Thus, the depth of the feature map would be n.
2. **Stride:** Stride is the number of pixels by which the filter matrix is slided over the input matrix. When the stride is 1 then the filters are moved one pixel at a time. When the stride

is 2, then the filters jump 2 pixels at a time as shown in Fig. 2.8. A larger stride will produce smaller feature maps.

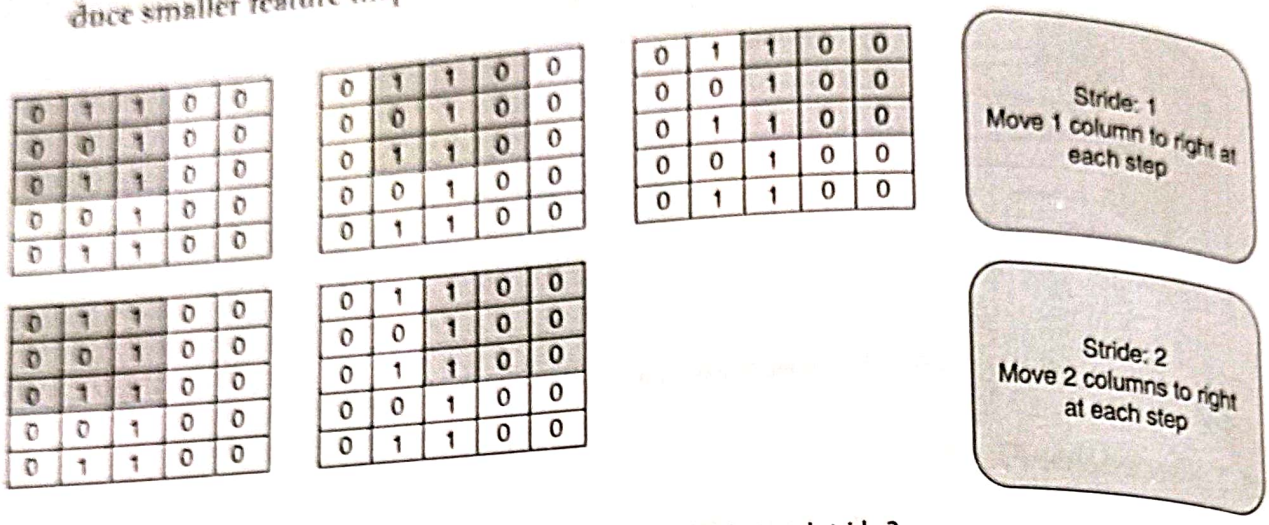


FIGURE 2.8 Representation of stride 1 and stride 2.

3. **Zero-padding:** Zero-padding is the process of adjusting the input size as per the requirement by adding zeros to the input matrix. It is mostly used in designing the CNN layers when the dimensions of the input volume need to be preserved in the output volume. Sometimes filter does not perfectly fit the input image. In that case, we need to either pad the picture with zeros so that it fits or drop the part of the image where the filter did not fit. This is called *valid padding* which keeps only the valid part of the image. When we add zero-padding, convolution is called *wide convolution*, and when zero-padding is not added, it is called *narrow convolution*. The representation of an input matrix with zero-padding is shown in Fig. 2.9.

2.2.2 | Pooling or Downsampling Layer

It is common to periodically insert a pooling layer in-between successive convolutional layers in a CNN architecture. Its function is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network, and hence to also control overfitting. The pooling layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2×2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. In this case, every MAX operation would be taking at max over 4 numbers (little 2×2 region in some depth slice). The depth dimension remains unchanged. The following are general features of the pooling layer:

1. Accepts a volume of size $W_1 \times H_1 \times D_1$.
2. Requires two hyperparameters – spatial extent F and stride S.
3. Produces a volume of size $W_2 \times H_2 \times D_2$ where

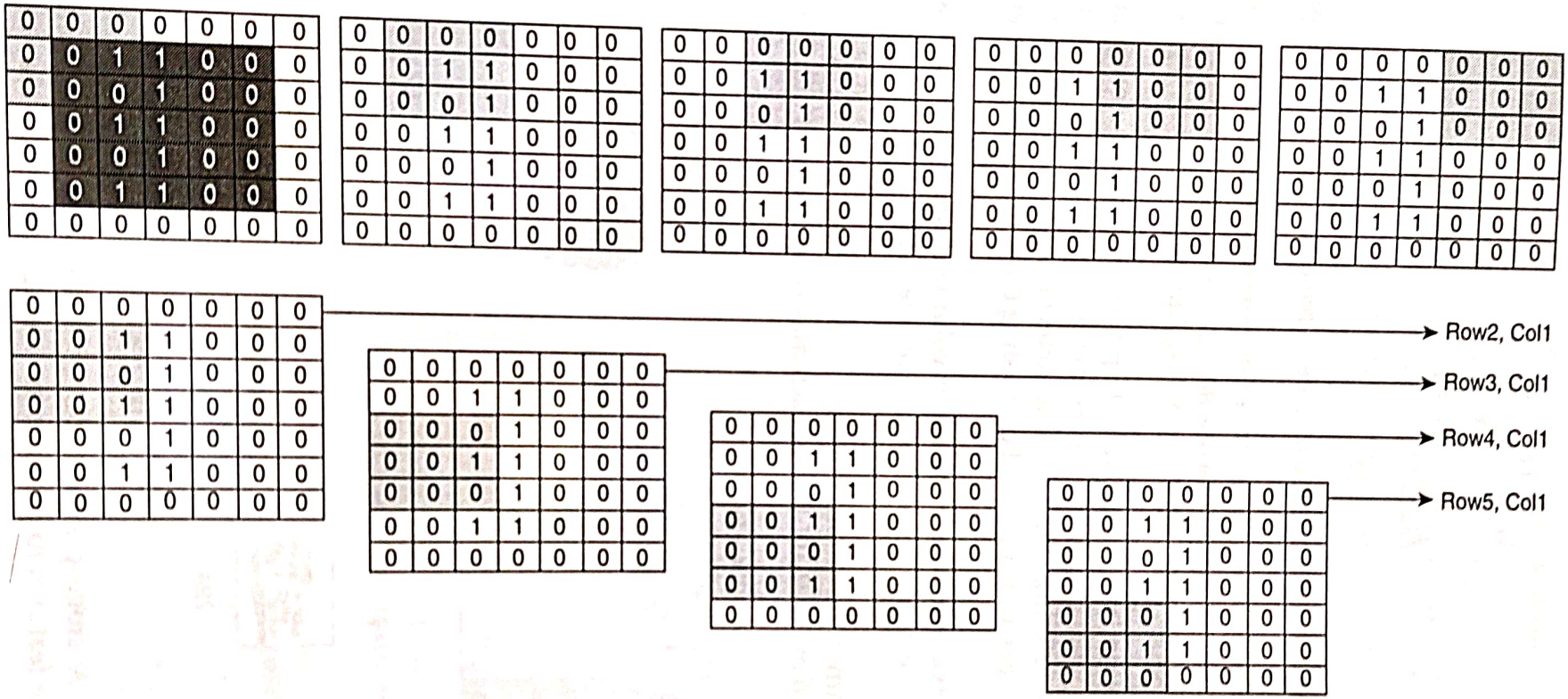


FIGURE 2.9 Representation of zero-padding.

$$W_2 = \frac{(W_1 - F)}{S} + 1$$

$$H_2 = \frac{(H_1 - F)}{S} + 1$$

$$D_2 = D_1$$

4. Introduces zero parameters since it computes a fixed function of the input.

Pooling layers section would reduce the number of parameters when the images are too large. Spatial pooling is called *subsampling* or *downsampling*; it reduces the dimensionality of each map but retains the important information. Spatial pooling can be of different types:

1. **Max-Pooling:** Pooling layers are generally placed in between convolution layers. Pooling layers reduce the spatial size of the representation to reduce the computation required and puts a check on overfitting. Max pooling determines the important features of the input image, like sharp edges and curves. Max-pooling takes the largest element from the rectified feature map.

   Consider the example where the input volume of size [224 × 224 × 64] is pooled with filter size 2 and stride 2 into output volume of size [112 × 112 × 64]. Notice that the volume depth is preserved.

   The most common downsampling operation is max, giving rise to max-pooling with a stride of 1, as shown in Fig. 2.10. That is, each max is taken over 4 numbers (little 2 × 2 square).

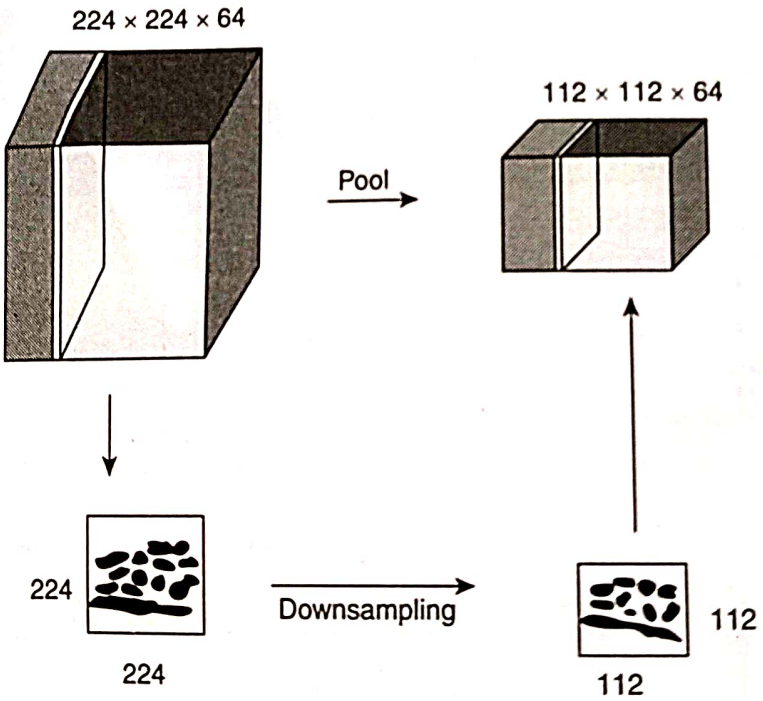


FIGURE 2.10 Max-pooling.

2. **Average Pooling:** Average pooling identifies the features in a smooth fashion. It takes into account all the features to produce an average value that may or may not detect sharp

features. Average pooling works well for straight lines and smaller curves, but cannot detect extreme features like sharp edges.

3. **Sum Pooling:** The sum of all elements in the feature map is called sum pooling. Max pooling takes the maximum value from the input region, whereas sum pooling uses the sum of the values in the input region. Both max pooling and sum pooling decrease the dimension of the input data while efficiently retaining the features. On the other hand, average pooling may or may not detect all the features.

2.2.2.1 *Getting Rid of Pooling*

Many people dislike pooling operation and try to work without it. To reduce the size of the representation without using a pooling layer, people may want to use a larger stride in convolutional layer once in a while. Discarding pooling layers has also been found to be important in training good generative models, such as Variational AutoEncoder (VAE) and Generative Adversarial Network (GAN). It seems likely that future architectures will feature very few to no pooling layers.

2.2.3 | Flattening Layer

Having extracted the features from the convolutional layer, and reduced the dimension by the pooling layer (optionally), it is the time for classification. The fully-connected layers cannot process multi-dimensional data though so that data should be reduced down to single dimension (flattened) for each channel before processing.

2.2.4 | Fully Connected Layer

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular neural networks. Hence, their activations can be computed with a matrix multiplication followed by a bias offset.

The fully connected layer is a traditional MultiLayer Perceptron (MLP) that uses a softmax activation function in the output layer. The term 'fully-connected' implies that every neuron in the previous layer is connected to every neuron on the next layer.

The output from the convolutional and pooling layers represent high-level features of the input image. The purpose of the fully connected layer is to use these features for classifying the input image into various classes based on the training dataset.

Apart from classification, adding a fully connected layer is also a (usually) cheap way of learning non-linear combinations of these features. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better.

The sum of output probabilities from the fully connected layer is 1. This is ensured by using the softmax as the activation function in the output layer of the fully connected layer. The softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one.

2.3 | RECTIFIED LINEAR UNIT (ReLU) LAYER

Activation layer is applied immediately after each convolution layer to introduce non linearity in the convolution layers. Non-linear functions like tanh and sigmoid were used initially to induce non-linearity. Now Rectified Linear Units (ReLU) are used, as they are found to be faster in training the network without compromising with accuracy. It also alleviates the vanishing gradient problem, which is the issue where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers. The ReLU layer applies the function

$$f(x) = \max(0, x)$$

to all of the values in the input volume. Basically, ReLU just changes all the negative activations to 0. It introduces the non-linear property in the overall network without affecting the receptive fields of the convolution layer.

ReLU is the most commonly used activation function for the outputs of CNN neurons. For negative input, the function returns 0, but for any positive value x it returns that value back.

This function can be used by neurons just like any other activation function. A node using the rectifier activation function is called ReLU node. The purpose of ReLU is to introduce non-linearity in CNN. There are other non-linear functions such as tanh or sigmoid, but performance-wise ReLU is better than the other two without making a significant difference to generalization accuracy. ReLU is important because it does not saturate; the gradient is always high (equal to 1) if the neuron activates. As long as it is not a dead neuron, successive updates are fairly effective. ReLU is also very quick to evaluate.

2.3.1 | Leaky ReLU and Randomized ReLU

Leaky ReLu helps increase the range of the ReLU function. Usually, the activation value a is 0.01 or so. When a is not 0.01 then it is called *randomized ReLU*. Therefore, the range of the leaky ReLU is $-\infty$ to ∞. Both leaky and randomized ReLU functions are monotonic in nature. Also, their derivatives also monotonic in nature.

Leaky ReLU has a small slope for negative values, instead of altogether zero. For example, leaky ReLU may have $y = 0.01x$ when $x<0$. Leaky ReLU is not always superior to plain ReLU, and should be considered only as an alternative, because the result is not always consistent.

Leaky ReLU has two benefits:

1. It fixes the "dying ReLU" problem, as it does not have zero-slope parts.
2. It speeds up training. Unlike ReLU, leaky ReLU is more "balanced," and may therefore learn faster.

2.4 | EXPONENTIAL LINEAR UNIT (ELU, OR SELU)

Exponential linear units (ELUs) try to make the mean activations closer to zero which speeds up learning. It has been shown that ELUs can obtain higher classification accuracy than ReLUs. ELU has a small slope for negative values, which is similar to leaky ReLU. Instead of a straight line, it uses a log curve as shown in Fig. 2.11.

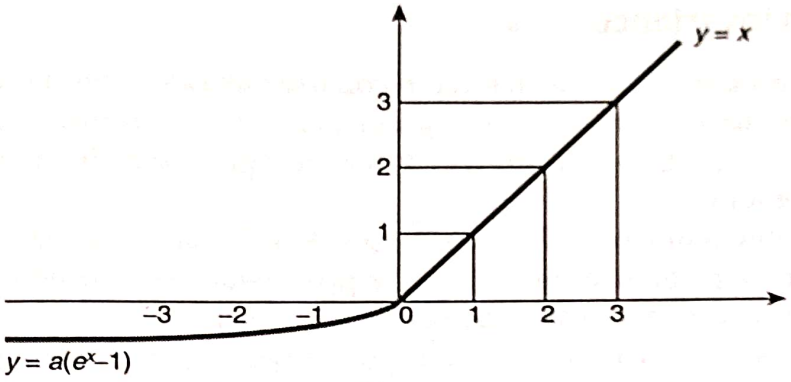


FIGURE 2.11 Exponential linear unit.

It is designed to combine the good parts of ReLU and leaky ReLU. While it does not have the dying ReLU problem, it saturates for large negative values, allowing them to be essentially inactive. It is sometimes called Scaled ELU (SELU) due to the constant factor a.

2.4.1 | Maxout

The Maxout neuron computes the function $\max(w_1 Tx + b_1, w_2 Tx + b_2)$. Notice that both ReLU and leaky ReLU are a special case of this form (e.g., for ReLU we have $w_1, b_1 = 0, w_2, b_2 = 0$). The Maxout neuron therefore enjoys all the benefits of a ReLU unit (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU). However, unlike the ReLU neurons it doubles the number of parameters for every single neuron, leading to a high total number of parameters.

2.5 | UNIQUE PROPERTIES OF CNN

2.5.1 | Weight Sharing

A property of convolutional neural networks is to share the same weight vectors across the receptive fields in a particular layer as shown in Fig. 2.12. It is known that a filter is applied to all the receptive fields in a layer to produce a feature map. By sharing the weights, the filter is learned irrespective of the position in the receptive field. Weight sharing is advantageous in some of the following ways:

1. It reduces the number of parameters to be learned or optimized.
2. Model converges faster.
3. It avoids overfitting since the neurons share the same weights.

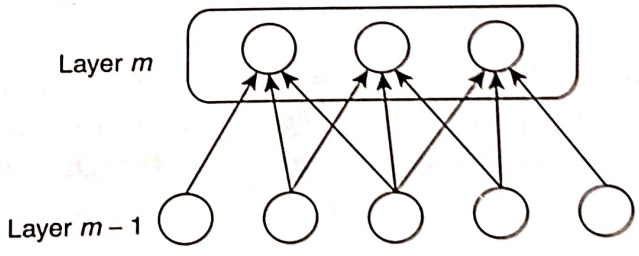


FIGURE 2.12 Neurons sharing same weights.

SOURCE: *Convolutional Neural Networks (LeNet);* http://deeplearning.net/tutorial/lenet.htm

2.5.2 | Translation Invariance

Once the human eye recognizes an object, it can recognize that same object under any circumstance (even if it is moved, rotated, under different lighting, and so on). Translation invariance provides CNN with one similar property, which allows CNN to recognize the identified object even when it is moved to any different place.

In Fig. 2.13, the statue is moved to different places. A well-trained neural network can recognize this statue in all the three pictures even though the pixel values are moved to different places. The output of the model remains the same irrespective of shifts in the input image. For example, face detector module in OpenCV identifies the face from any image regardless of the position of the face (it may differ from the reference shape predictor image that OpenCV uses) and says "FACE FOUND".

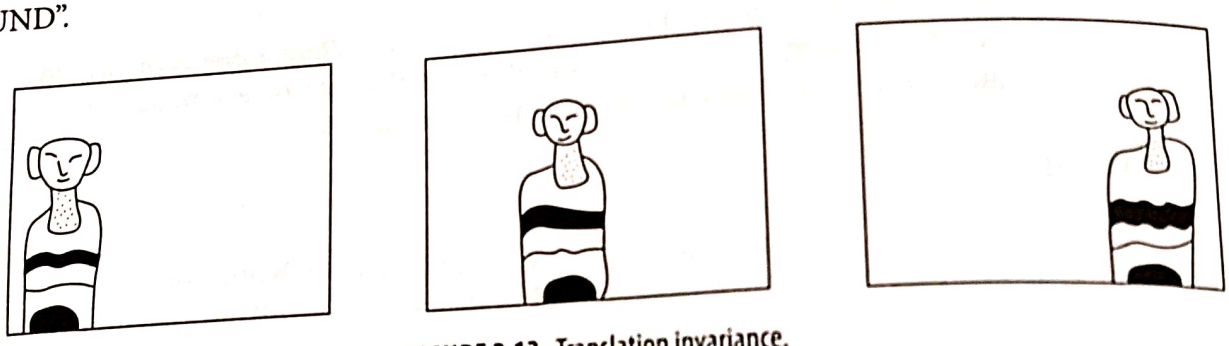


FIGURE 2.13 Translation invariance.

SOURCE: *Stack Exchange;* https://stats.stackexchange.com/questions/208936/what-is-translation-invariance-in-computer-vision-and-convolutional-neural-network

2.6 | ARCHITECTURES OF CNN

CNNs are special type of networks specifically designed to identify patterns from images. There are several CNN architectures designed to solve a particular image processing problem such as optical character recognition (OCR), object detection, face recognition, etc. The architectures differ in the number of parameters, number of layers, and type of layers. The error rate decreases as the architecture gets deeper. But some architectures, such as GoogLeNet, achieved minimal error rates with reduced number of parameters.

2.6.1 | LeNet

Introduced in 1998 by LeCun *et al.*, LeNet was the first deep convolutional architecture. It was used to perform OCR by several banks to recognize handwritten numbers on cheques digitized in 32 × 32 grayscale images. Though its ability to solve was admirable, it was constrained by the availability of computing resources at that time because of its high computational costs. Its architecture is shown in Fig. 2.14.

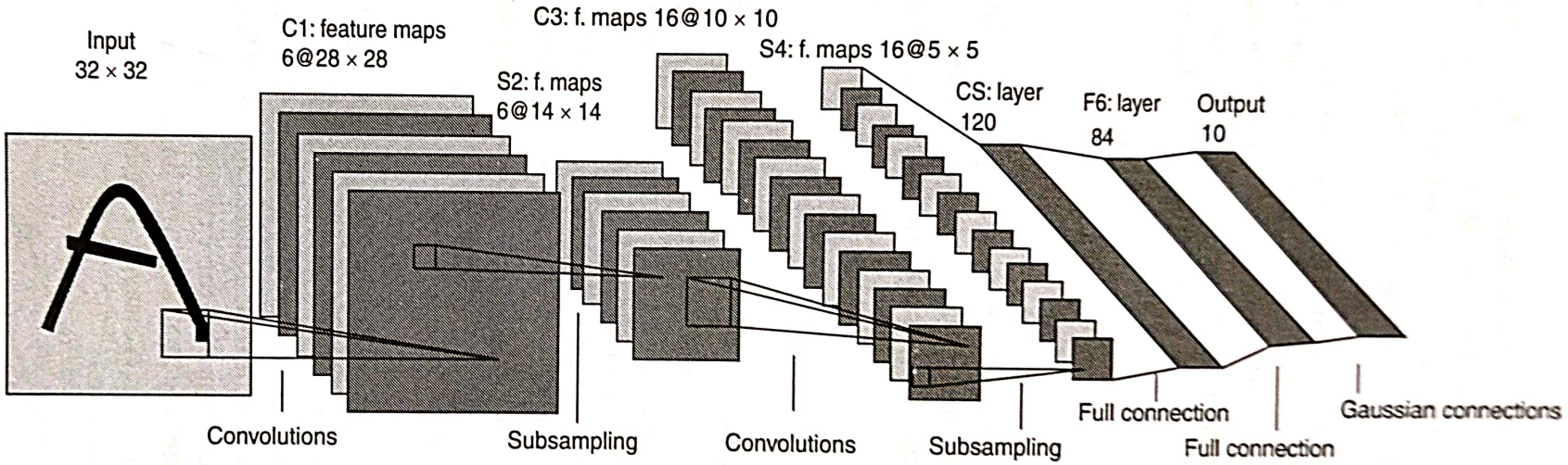


FIGURE 2.14 Architecture of LeNet.

SOURCE: LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. (1998) Gradient-Based Learning Applied to Document Recognition. *Proceedings of IEEE.*

2.6.2 | AlexNet

AlexNet was designed by the SuperVision group consisting of Alex Krizhevsky, Geoffrey Hinton, and Ilya Sutskever. It was the first CNN architecture that triumphed in the ImageNet Large Scale Visual Recognition Competition (ILSVRC) 2012 contest with error rates half that of its nearest competitors (from 26% to 15.3%). This victory dramatically stimulated the trend toward deep learning architectures in computer vision.

From Fig. 2.15, we can see that it has a similar architecture as LeNet but deeper and with more stacked convolutional layers. It has 5 convolutional layers and 3 fully connected layers summing up to 8 layers in total. ReLU was applied after each convolutional and fully connected layer whereas dropout was applied before the first and second fully connected layers. It was trained for 6 consecutive days on two NVIDIA Geforce GTX 580 GPUs and so the processing was split into two pipelines.

2.6.3 | ZFNet

ZFNet is the winner of the ILSVRC 2013. It achieved a top-5 error rate of 14.8% which is already a significant improvement over AlexNet. It was mostly an achievement by tweaking the hyper-parameters of AlexNet while maintaining the same structure with additional deep learning elements as shown in Fig. 2.16.

2.6.4 | GoogLeNet

Since 2012, CNN architectures have been coming out with flying colors in the ILSVRC contest. GoogLeNet (also known as Inception) is the winner of the ILSVRC 2014 contest. It achieved an error rate of 6.67% with its 22 layers, which was very close to the human level performance.

The architecture shown in Fig. 2.17 used a CNN inspired from LeNet but introduced 9 inception modules. They used batch normalization, image distortions and RMSprop, and were based on several but very small convolutions. The idea was to cover a bigger area but to keep a fine resolution on the images. So it convolved different sizes from 1×1 to 5×5 in parallel.

2.6.5 | VGGNet

The runner-up at the ILSVRC 2014 competition was dubbed VGGNet by the community. It was developed by Simonyan and Zisserman. VGGNet consists of 16 convolutional layers as shown in Fig. 2.18 and is very appealing because of its very uniform architecture. Similar to AlexNet, it has only 3×3 convolutions, with lots of filters. Trained on 4 GPUs for 2–3 weeks, it is currently the most preferred choice in the community for extracting features from images. The weight configuration of the VGGNet is publicly available and has been used in many applications and challenges as a baseline feature extractor. However, VGGNet consists of 138 million parameters, which can be a bit challenging to handle.

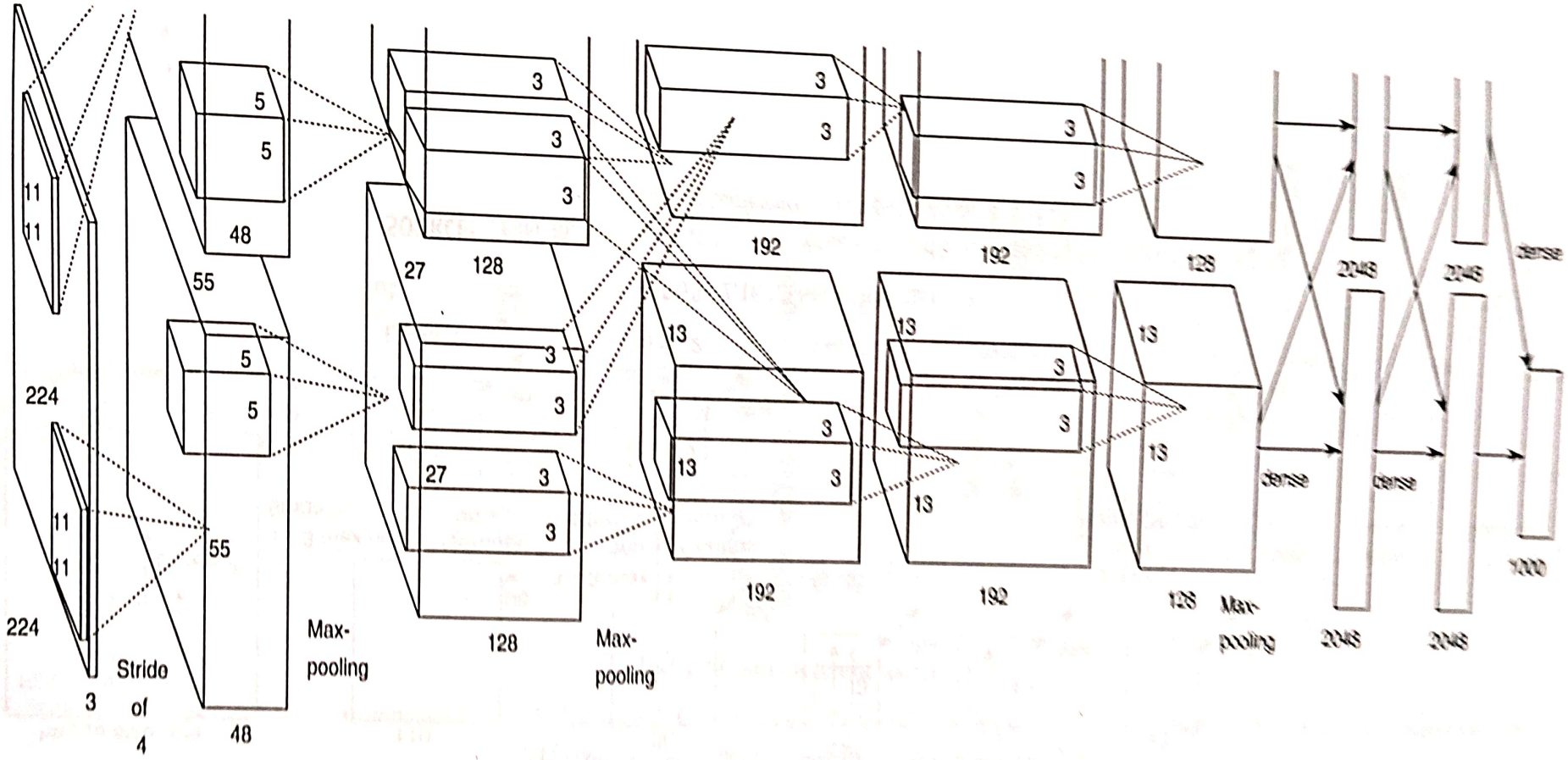


FIGURE 2.15 AlexNet architecture.

SOURCE: Krizhevsky, A., Sutskever, I. and Hinton, G. E. (2012) Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 1097–1105.

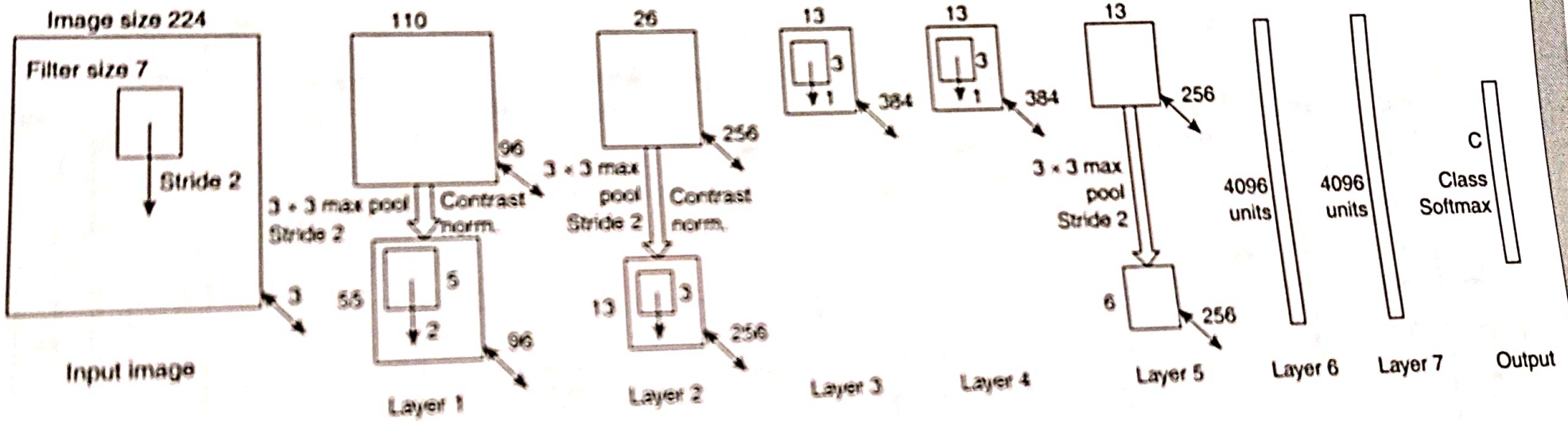


FIGURE 2.16 ZFNet architecture.

SOURCE: Zeiler, M. D. and Fergus, R. (2014) Visualizing and Understanding Convolutional Networks. *European Conference on Computer Vision*, 818–833.

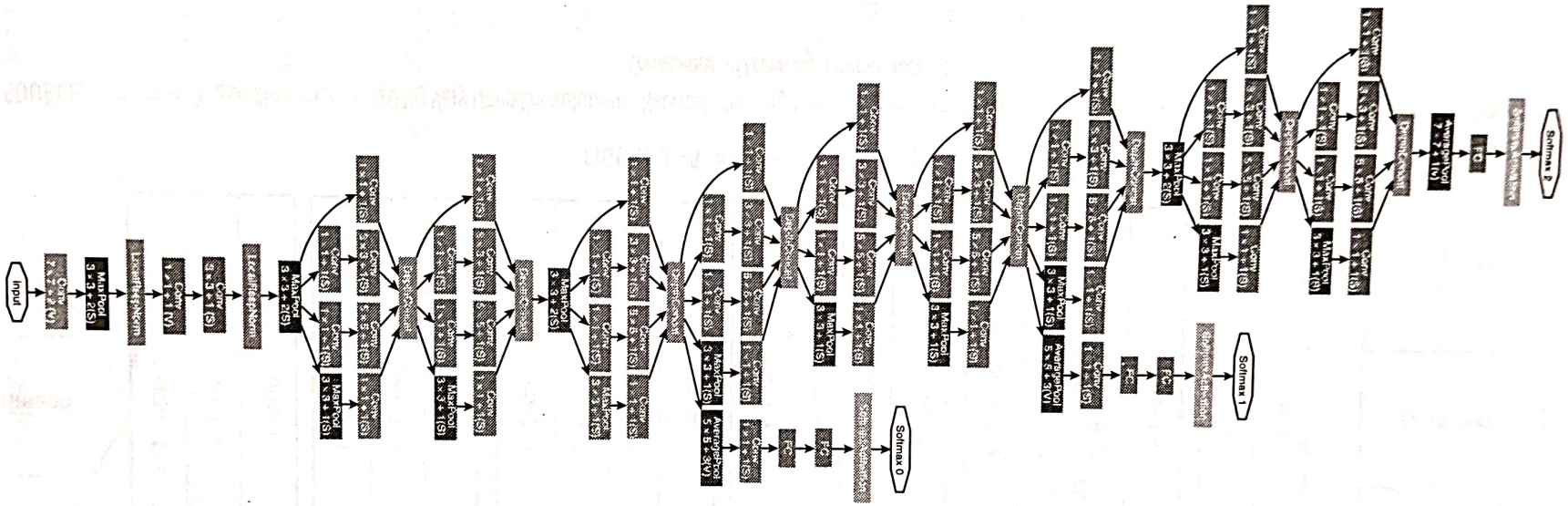


FIGURE 2.17 Architecture of GoogLeNet.

SOURCE: Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V. and Rabinovich, A. (2015) Going deeper with convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1–9.

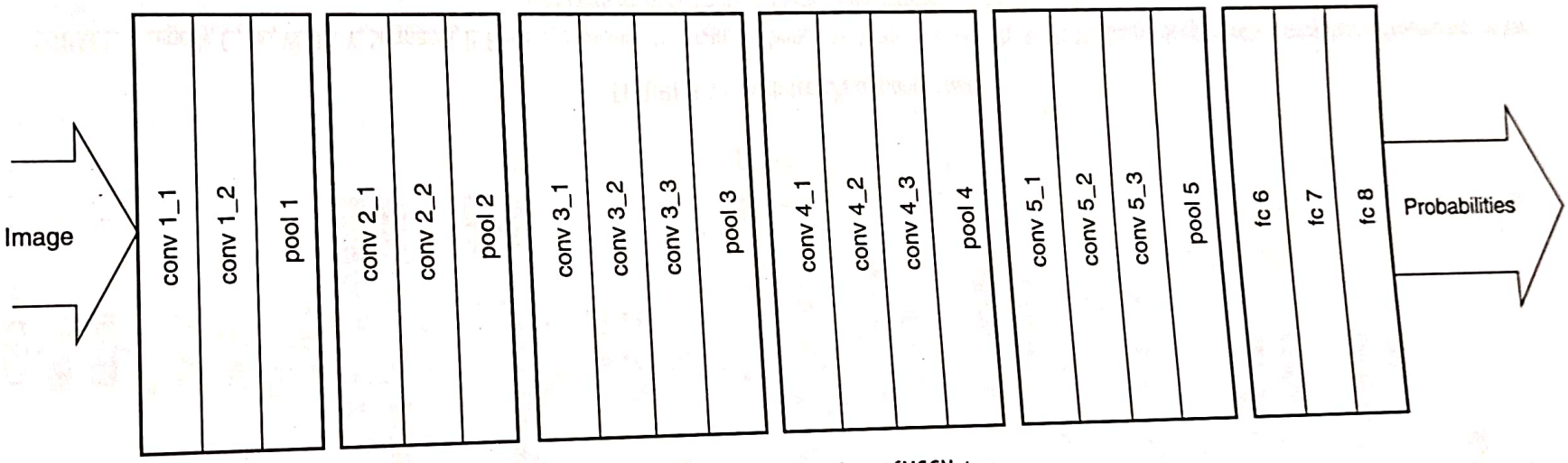


FIGURE 2.18 Architecture of VGGNet.

SOURCE: Simonyan, K. and Zisserman, A. (2015) Very Deep Convolutional Networks for Large-Scale Image Recognition. *International Conference on Learning Representations, International Conference on Learning Representations*.

2.6.6 | ResNet

ResNet was designed by Kaiming He *et al.* It is the 2015 winner of ILSVRC. It introduced 'identity shortcut connection' that skipped one or more layers, which were called 'skip connections', as shown in Fig. 2.19. The vanishing gradient problem, the main problem of deep networks, was solved by the use of skip connections. ResNet featured heavy batch normalization, and with 152 layers, it reduced the error rate to 3.57% which beat human performance. Residual network boosted the performance of many computer vision applications like object detection, face recognition, etc. Due to its success, many research communities started finding out the reason for its effectiveness and many refinements were made like ResNeXt, DenseNet, etc.

2.6.7 | DenseNet

The idea behind dense convolutional networks may be useful to reference feature maps from earlier in the network. Thus, each layer's feature map is concatenated to the input of every successive layer within a dense block. This allows later layers within the network to directly leverage the features from earlier layers, encouraging feature reuse within the network, which thereby improves the efficiency. The architecture of DenseNet is shown in Fig. 2.20.

2.7 | APPLICATIONS OF CNN

CNNs are multistage architectures with convolution, pooling, and fully connected layers. They have become an integral part of computer vision which aims at imitating the functionality of the human eye and its brain insights by a machine to understand and process images. Its applications include object detection, action recognition, scene labelling, character or handwriting recognition, etc. Some important applications are discussed in the following sub-sections.

2.7.1 | Object Detection

Object detection is a technology that deals with detecting real-world objects from a given scene. Technically, it detects the instances of a particular class like animals, birds, cars, etc. from the given image. As CNNs are getting deeper, many complex computer vision problems can be solved using these deep CNNs. All applications of CNN are built on top of object detection. Figure 2.21 shows the detection of dogs from an input image.

2.7.2 | Face Recognition

Face recognition is a problem that predicts whether there is a match between the face that is input and those available in the database. The common facial features include eyes, nose, mouth, and chin but sometimes the background of the image is also taken into account. Face recognition is affected by many problems, including the following:

1. Identifying all faces possible.
2. Focus on each face regardless of lighting and perspective.

FIGURE 2.19 Architecture of ResNet.

SOURCE: He, K., Zhang, X., Ren, S. and Sun, J. (2015) *Deep Residual Learning for Image Recognition*. [arXiv preprint arXiv:1512.03385].

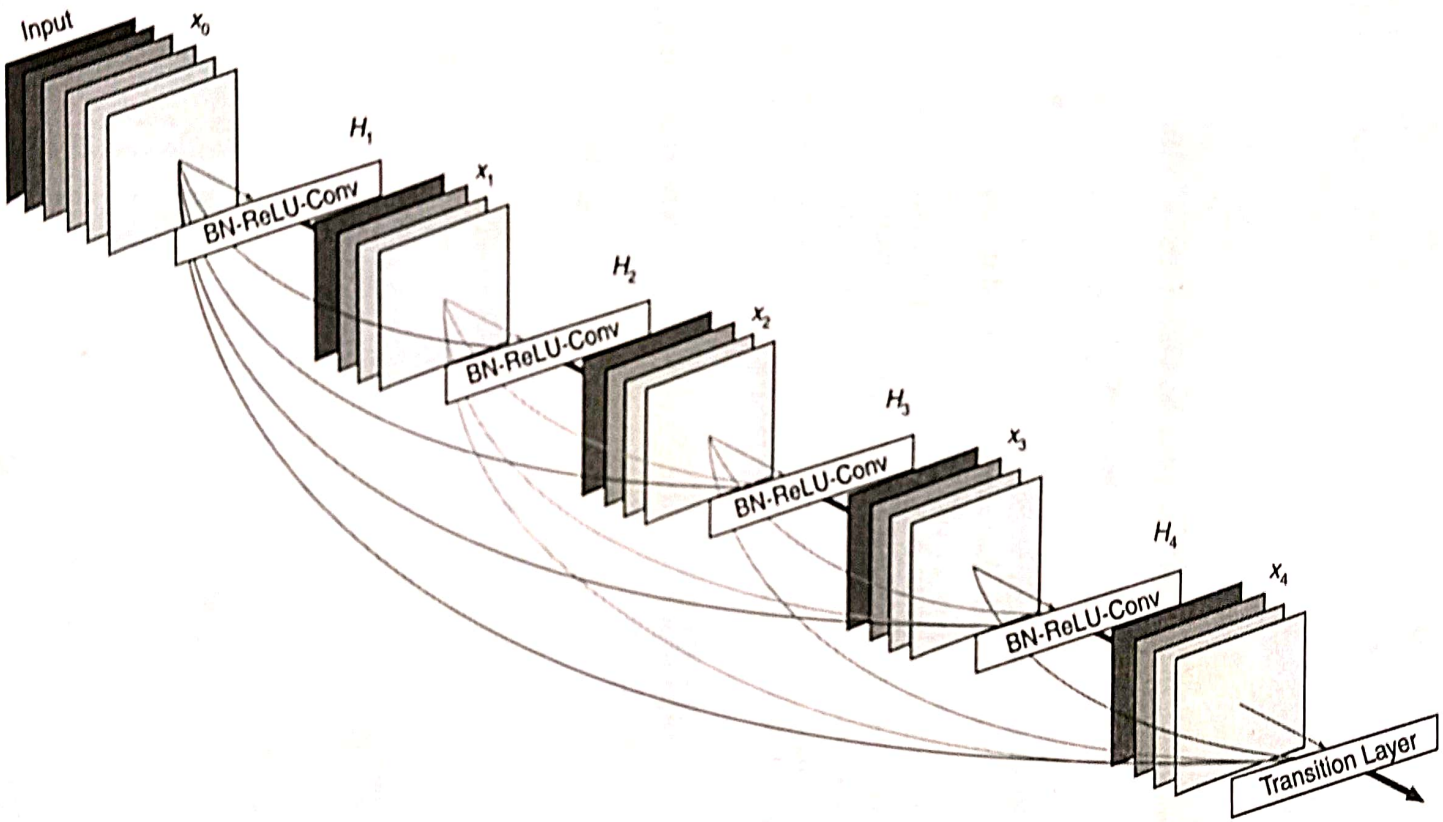


FIGURE 2.20 Architecture of DenseNet.

SOURCE: Huang, G., Liu, Z., van der Maaten, L. and Weinberger, K. Q. (2017) Densely Connected Convolutional Networks. *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR).*

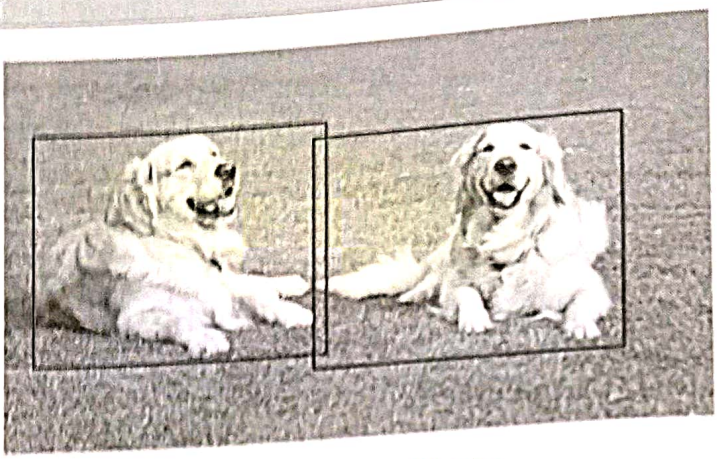


FIGURE 2.21 Object detection.

3. Finding unique features to a face.
4. Comparing identified features with that available in the database.

Usually, face recognition turns out to be an *N*-way classification problem where it is able to recognize *N* identities. However, it is also applied for recognizing single or two identities depending on the application. There are many datasets available for face recognition such as Labeled Faces in the Wild (LFW) dataset with 13,233 images of 5,749 identities, and YouTube Faces (YTF) dataset with 3,425 videos of 1,595 identities. An overview of the steps of CNN based face recognition are as follows:

1. Detect and crop faces on the input to feed aligned faces to CNN improving performance.
2. Extract vector representation of faces called embeddings.
3. Compare input vector embeddings to labeled vector embeddings in the dataset using a classifier.

Note that CNN is only used to extract facial features and only a classifier recognizes class labels. The cropped and aligned face of Jacques Chirac from LFW dataset is shown in Fig. 2.22.

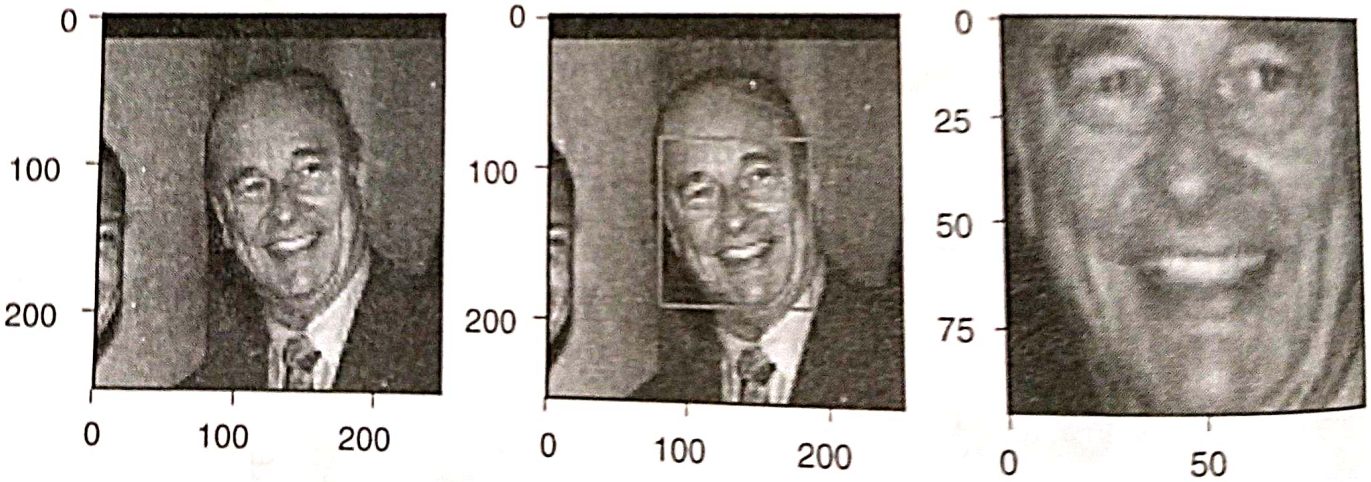


FIGURE 2.22 Cropped and aligned face of Jacques Chirac from LFW dataset.

2.7.3 | Scene Labeling

Scene labeling is the process of labeling every pixel in the given image with the category of the object it belongs. Let us consider the image in Fig. 2.23, which depicts the idea of scene labeling. The task is to take each pixel in the image and categorize it to the object of the class it belongs. Each pixel holds information about the object it is part of.

State-of-the-art systems incorporate CNN with a recurrent architecture. A recurrent system means the network has its networks share the same set of weights. The network is made to self-correct. It learns to smoothen the predicted values. As the size of the system increases, the system becomes more self-correcting.

An algorithm proposed by UCB used R-CNN performed 30% better, but yet was simple and scalable.

Recent developments in deep neural networks prove to be beneficial for machine learning applications. Likewise, deep learning convolution networks have greatly improved the performance of computer systems on problems in image classification. Although the solution is not end-to-end, this promises a better outcome.

The end-to-end, pixels-to-pixels trained CNNs overcome the shortcomings of previous approaches used for semantic segmentation where each pixel was labelled with the class of either the encoding object or region. Fully convolutional networks such as AlexNet, GoogleNet, and VGG Net achieve the state-of-the-art segmentation of PASCAL VOC, NYUDv2, and SIFT Flow datasets (20% relative improvement), where inference takes less than one-fifth of a second for a particular image.

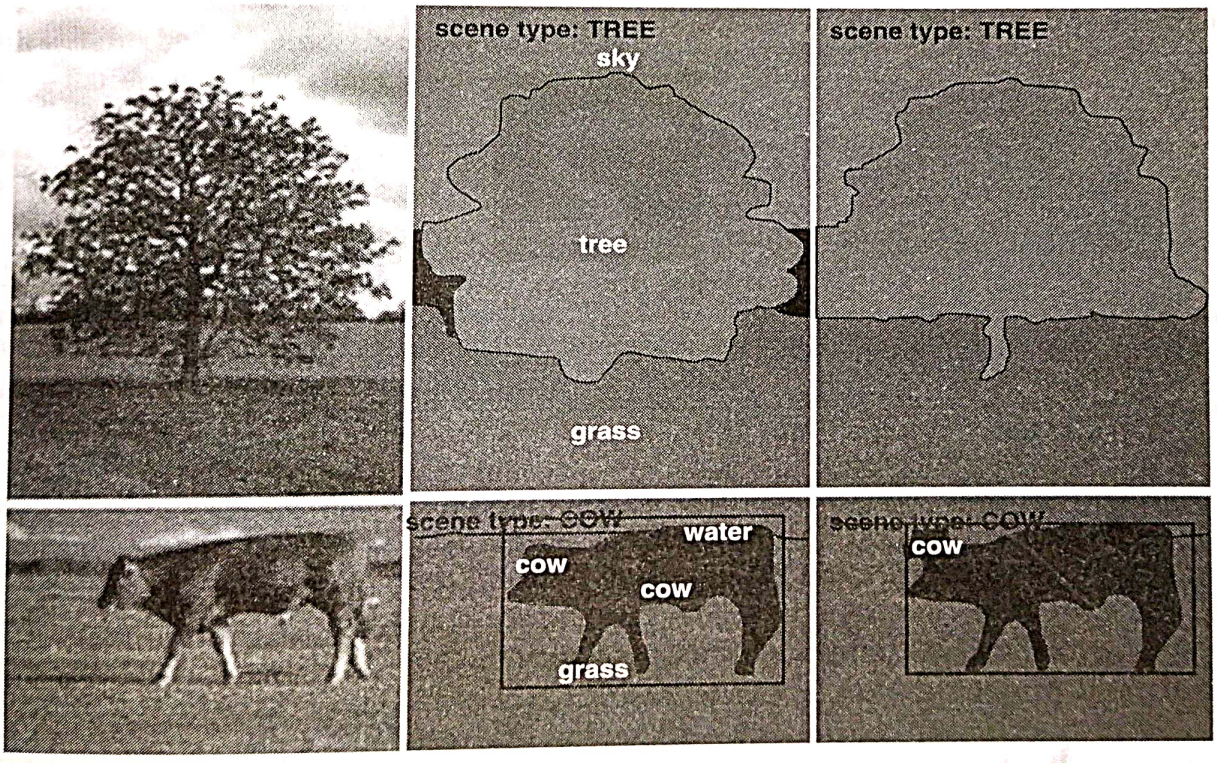


FIGURE 2.23 Scene labelling.

2.7.4 | Optical Character Recognition (OCR)

OCR is one of the domains where CNN gives the best result. Traditional systems rely on lengthy methodologies which need a large amount of training knowledge. But a system that uses multilayer neural networks with CNN can be used to design highly accurate text detectors and character modelers.

A simple system can provide extraction of text from scanned documents. However, we need a system that can recognize text in unconstrained images, where characters can be found anywhere on the image randomly with different formats (e.g., characters may be rotated through different degrees, or have different pixel density, or have different foreground and background color, or no restriction on noise level). For such cases, CNNs have proven to provide higher accuracy than most traditional approaches and other neural networks.

2.7.5 | Handwritten Digit Recognition

Handwritten digit recognition problem is one of the object recognition problems. A popular dataset that is available for this problem is the Modified National Institute of Standards and Technology (MNIST) dataset. It is a modified version of the NIST dataset in the sense that the digits are size-normalized and centered in the image of fixed size. It contains about 60,000 train examples and 10,000 test examples. This problem is like the "Hello World" problem for object detection. As it is a classification problem, regular machine learning classifiers can also be used. However, CNN proves its utmost efficacy in this problem by reducing the error rate to around 0.2%. Being a digit recognition problem, it becomes a 10-class (digits 0–9) classification problem. Figure 2.24 shows an example image of digit 2 available in MNIST dataset.

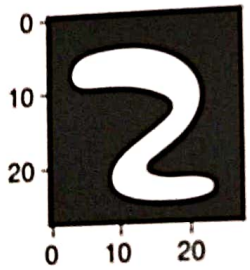


FIGURE 2.24 A sample from MNIST dataset corresponding to digit 2.

SUMMARY

This chapter has introduced the concept of CNN followed by a detailed description of CNN architecture, various components of CNN, unique CNN properties, CNN variants and a few of its infinite number of applications. Section 2.2 discussed in detail about each component (convolutional layer, pooling layer, fully connected layer and the activations) in a CNN architecture. Sections 2.3 and 2.4 discussed the various activation functions that could be used in CNN hierarchy. Section 2.5 discussed the two most important properties of any CNN architecture: weight sharing and translation invariance. The chapter concludes by discussing few real-time applications of CNN such as object detection, handwritten OCR, face recognition, scene labeling. The effectiveness of CNN in

image detection and processing is mentioned repeatedly throughout the chapter with explanation for some famous architecture including LeNet, AlexNet, VGGNet.

REVIEW QUESTIONS

1. What is the purpose of using filters?
2. What does the dimension of receptive field depend on?
3. How do the negative activations influence the gradient?
4. How can a CNN identify an object even when it is transformed?

ASSIGNMENT PROBLEMS

1. Given the input matrix and the kernel, perform convolution with stride being 1 and 2.

1	0	1	1	0
0	0	0	1	1
1	0	0	0	1
0	1	1	1	0
1	1	0	1	0

Input matrix

1	0	0
0	0	1
1	1	0

Kernel

2. Apply max-pooling, average-pooling, and sum-pooling to the results from above convolutions.
3. Visualize the flattened version of the pooled feature maps from Question 2.
4. With weights being randomly chosen and bias being 1, learn the weights and bias over an epoch for the flattened arrays from Question 3.

REFERENCES

1. He K., Zhang X., Ren S., and Sun J. 2015. Deep residual learning for image recognition. [arXiv preprint arXiv:1512.03385]
2. Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. *International Conference on Learning Representations, International Conference on Learning Representations (ICLR)*
3. Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks, *European Conference on Computer Vision (ECCV)*, 8689: 818–833
4. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 1097–1105
5. Szegedy C., Liu W., Jia Y., Sermanet P., Reed S., Anguelov D., Erhan D., Vanhoucke V., and Rabinovich, A. 2015. Going deeper with convolutions. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1–9
6. Yann LeCun, Leon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based Learning Applied to Document Recognition, *Procedure of IEEE*

CHAPTER 3

Recurrent Neural Network: Basic Concepts

LEARNING OBJECTIVES

After reading this chapter, you will be able to

- Know the basics of recurrent neural network and its architecture.
- Understand the concept of recurrent neural network.
- Apply recurrent neural network in the real-time applications.

3.1 | INTRODUCTION

Recurrent Neural Networks (RNNs) are a class of neural network architectures that received a lot of focus in research and development areas during the 1990s. Artificial neural networks are described as computational structures built from weighted connections among simple and functionally similar non-linear processing units or nodes, denoted using artificial neurons. A recurrent network is simply an artificial neural network with feedback (closed loop) connections. Examples include bidirectional associative memory, Hopfield Boltzmann machine, and recurrent backpropagation nets. These RNNs are designed to learn sequential or time-varying patterns. One of the reasons for the achievements of deep learning is the implementation of RNN algorithms A salient feature of RNN is that it can retain information of its input, due to internal memory, and can figure out a much deeper realization of a sequence and its context, compared to other algorithms. Thus, problems that involve sequential data such as text, financial data, time series, speech, audio, video, weather, etc. can be solved using RNNs. They are widely used in the software for Google Translate and Siri of Apple.

Few applications that widely use RNNs are as follows:

1. **Speech Recognition:** Identifies words and phrases spoken and converts them into machine readable format. For example, an audio clip X is taken as input and it is mapped to text transcript Y.
2. **Music Generation:** Here, the input may be the genre of music to be generated and the output is music sequence.
3. **Machine Translation:** Conversion of a sentence from one language to another.
4. **DNA Sequence Analysis:** DNA is represented by the alphabets A, C, G, and T. A DNA sequence is given as input and RNN labels the protein DNA sequence.

5. **Video Activity Recognition:** Identifies activity from the sequence of video frames.
6. **Sentiment Classification:** Classification of text data according to the sentimental polarities of views contained in it.

3.1.1 | RNN versus CNN

A main limitation of convolutional neural networks (CNNs) (and also vanilla neural networks, covered in Chapter 2) is that their API is much constrained, that is, they admit a predetermined-sized vector as input (e.g., an image) and generate a predetermined-sized vector as output (e.g., probabilities of different classes). This input–output mapping is performed using a fixed number of computational steps (e.g., number of layers in the model). The main reason that recurrent nets are more stimulating is that they permit us to function over sequences of vectors (i.e., sequences in the input, the output, or both) (Nicholson and Gibson, 2001).

3.1.2 | Feedforward Neural Network versus RNN

In a feedforward neural network, the sequential data moves from the input layer to the output layer through the hidden layer, that is, the information flows in only one straight direction through the network. Thus, the information never reaches the same node twice.

Feedforward neural networks are inadequate in forecasting the future inputs as they have no memory of the input they received in the past. They consider only the present input and have no conception of order in time. They can retain information only about their training and cannot retain any other past information. In short, feedforward networks have forgetfulness regarding their recent past; they can remember nostalgically only the shaping moments of training.

In an RNN, the information goes through a loop. When it formulates a decision, it takes the current input and the learning from past inputs into consideration. The dissimilarities in the information flow between an RNN and a feedforward neural network are illustrated in Fig. 3.1 (Nicholson and Gibson, 2001).

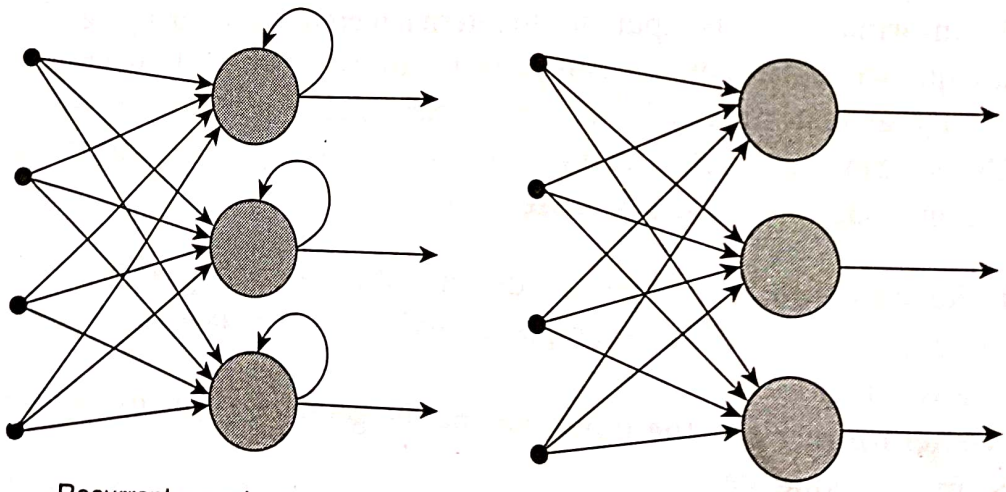


FIGURE 3.1 Recurrent neural network and a feedforward neural network flow.

3.2 | SIMPLE RECURRENT NEURAL NETWORK

A recurrent neural network takes the current input and also the previous observation as input. The Elman network was introduced by Elman in 1990 .The Elman network has connections from the hidden layer to a context layer and also has connections from input to the hidden layer, as shown in Fig. 3.2. The role of the context units is to provide dynamic memory so as to encode the information contained in the sequence of phonemes necessary for prediction. It stores the past output.

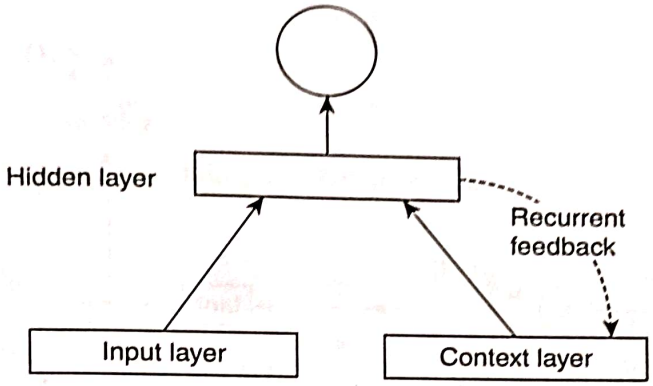


FIGURE 3.2 Simple recurrent net proposed by Elman.

The decision a simple recurrent net takes at tth time instant is affected by the decision taken at $(t-1)$th instant. So, recurrent networks have two input sources – the present and the recent past – which jointly determine how they respond to new data.

Recurrent networks are differentiated from feedforward networks by the feedback loop which is connected to their past decisions, receiving their own outputs moment after the moment as input has been repeatedly said that recurrent networks have the memory which takes information in the sequence, and recurrent nets utilize it to perform functions that cannot be performed in feedforward networks.

This sequential information is maintained in the recurrent network's hidden state, which handles to span many time steps as it cascades forward to influence the processing of each new example. It finds correlations among events parted by several moments, and these correlations are called *long-term dependencies*, because an event downstream in time is dependent on, and is a function of, one or more events that occurred before. RNN is a technique to consider that shares weights over time.

The process of transmitting memory forward is described mathematically as

$$h_t = \phi(Wx_t + Uh_{t-1}) \tag{3.1}$$

Preferably,

$$h_t = \tanh(Wx_t + Uh_{t-1}) \tag{3.2}$$

At a time step t, the hidden state is represented as h_t. It is a non-linear mathematical function of the input X_t, multiplied by a weight matrix W (like the one that was used for feedforward nets) added to the hidden state of the past time step h_{t-1} multiplied by its own hidden-state-to-hidden-state matrix U. The matrix U is also known as *transition matrix*. The choice of tanh over other non-linearities has to do with its second derivative decaying very gradually to zero. This helps to resolve the vanishing gradient problem as it keeps the gradients in the linear region of activation function.

The output vector y_t at time t is the product of the weight matrix V and the hidden state h_t, with softmax, a function that takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities applied to the product so that the resulting vector is a set of output probabilities:

$$\hat{y}_t = \text{softmax}(Vh_t)$$

An RNN can be seen as the repetition of a single cell. The computation for a single time step of an RNN cell is shown in Fig. 3.3.

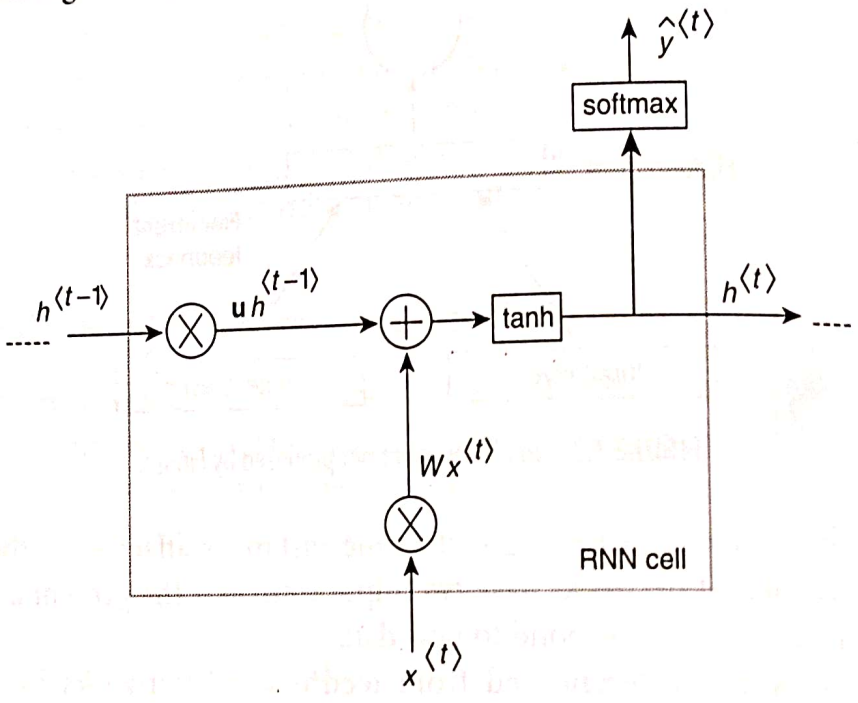


FIGURE 3.3 Single time step of an RNN cell.

To implement a single RNN cell, the following pseudo-code is used:

1. Compute the hidden state h_t with tanh activation.
2. Using your new hidden state h_t compute the prediction $\hat{y}_t$.
3. Store h_t, h_{t-1}, $x(t)$ and parameters.
4. Return h_t and $\hat{y}_t$.

After implementation of a single RNN cell, a basic RNN network is built as shown in Fig. 3.4.

We can also represent the RNN cell graphically as shown in Fig. 3.5 (left). At time t, the cell has an input x_t, and an output $\hat{y}_t$. Fraction of the output $\hat{y}_t$ (the hidden state h_t) is fed back into the cell for use at a later time step at $t + 1$. The parameters of RNNs are given by three weighted matrices just as a traditional neural network's parameters are contained in its weight matrix. These matrices are U, V, and W, corresponding to the input, output, and hidden state, respectively.

Another way to look at an RNN is to unroll it, as shown in Fig. 3.5 (right). By unrolling we mean that we draw the network for the complete sequence. The network shown here is a three-layer RNN suitable for processing three-element sequences. Note that the weight matrices U, V, and W are shared across the steps. This is because we are applying the same operation on different inputs at each time step. Being able to share these weight vectors across all the time steps greatly reduces the number of parameters that the RNN needs to learn.

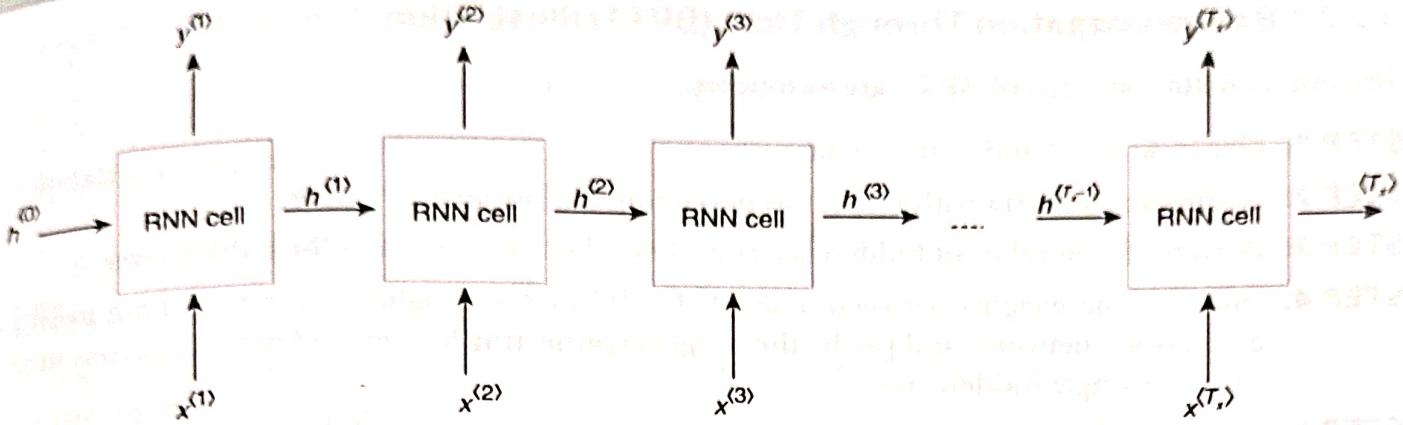


FIGURE 3.4 Basic RNN model.

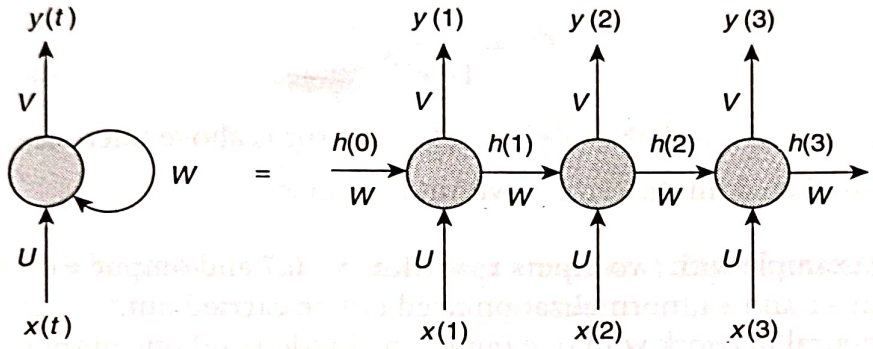


FIGURE 3.5 Unrolled RNN.

3.2.1 | Training an RNN

Training an RNN is comparable to training a CNN. The backpropagation algorithm is utilized, but with a slight twist. The gradient at all output relies not only on the computations of the current time step but also on the previous time steps as the parameters in the network are shared across all time instances.

For example, in order to determine the gradient at $t = 6$, we would need to backpropagate five steps and sum up the gradients. This is called *backpropagation through time* (BPTT). Thus, RNNs trained with BPTT algorithm have difficulties in learning long-term dependencies (i.e., dependencies among steps that are far apart). This complication arises because of vanishing/exploding gradient problems. These problems are dealt by a few technologies, and certain types of RNNs [like long short-term memory (LSTM)] were precisely designed to get around them.

In a simple (fully connected) neural network, backpropagation is used to compute the derivatives with respect to the cost function J to update the parameters W, U. Similarly, in RNN the derivatives are calculated with respect to the cost in order to update the parameters (i.e., the derivatives of the cost function J backpropagate through the RNN by following the chain rule from calculus to update the weights W, U).

3.2.2 | Backpropagation Through Time (BPTT) Illustration

The implementation steps of BPTT are as follows:

STEP 1: Generate input and output data.

STEP 2: Normalize the data with respect to maximum and minimum values.

STEP 3: Assume the number of hidden layers and number of neurons in the hidden layer.

STEP 4: Initialize the weights between 0 and 1. Let $[v]$ be the weights connecting input neurons and hidden neurons, and $[w]$ be the weights connecting hidden and output neurons in the case of a single hidden layer.

STEP 5: Compute the input and output at every neuron present in each layer. Let the last hidden layer compute the output using sigmoidal activation function given by.

$$O^h = \frac{1}{1+e^{-I^h}}$$

STEP 6: Find the error and check the tolerance. If the error is above tolerance, update the weights.

STEP 7: Repeat from Step 4 till the error is within tolerance.

Consider a training example with two inputs $x_1 = 0.4$, $x_2 = -0.7$ and output = 0.1. Since the input and output data is within −1 and +1, normalization need not be carried out.

Considering a neural network with one input, one hidden and one output layer, let the weights be initialized as

$$[v]^0 = \begin{bmatrix} 0.1 & 0.3 \\ -0.2 & 0.1 \end{bmatrix}$$

$$[w]^0 = \begin{bmatrix} 0.2 \\ -0.4 \end{bmatrix}$$

Computing the input to hidden layer, we get

$$I^h = [v]^T \{O\}^i = \begin{bmatrix} 0.1 & -0.2 \\ 0.3 & 0.1 \end{bmatrix} \begin{bmatrix} 0.4 \\ -0.7 \end{bmatrix} = \begin{bmatrix} 0.18 \\ 0.05 \end{bmatrix}$$

Computing the output of the hidden layer using sigmoid function, we get

$$O^h = \begin{bmatrix} 0.5448 \\ 0.5125 \end{bmatrix}$$

Similarly, calculating input and output of the output layers, we get

$$I^o = -0.09604 \text{ and } O^o = 0.476$$

The actual output of the training example is 0.1. Thus,

$$\text{Error} = (0.1 - 0.476)^2 = 0.141376$$

To adjust the weights, we need partial derivatives of error:

$$d = (\text{True output} - O^0)(O^0)(1 - O^0) = -0.09378$$

To update the weights,

$$[\Delta w]^{t+1} = \alpha[\Delta w]^t + \phi[y]$$

where

$$[y] = O^h \times d$$

Let us assume momentum term, $\alpha = 0.5$ and learning rate, $\phi = 0.6$. Then substituting the values, we get

$$[\Delta w]^1 = \begin{bmatrix} -0.03065 \\ -0.02884 \end{bmatrix}$$

The updated weights are

$$[w]^1 = [w]^0 + [\Delta w]^1 = \begin{bmatrix} -0.16935 \\ -0.42884 \end{bmatrix}$$

Similarly, update the weights of $[v]$. The iterations are carried out until error comes under tolerance range. Once these iterations are over, the network can predict test data.

3.2.3 | RNN Topology

RNNs do not have the limitation of performing the transformation from input to output in a constant number of steps given by the constant number of layers in the model. Sequences in the input, the output, or both are possible. This means that RNNs can be organized in various ways to resolve specific problems. This property of being able to work with sequences gives rise to a number of common topologies, some of which are shown in Fig. 3.6.

Consider each rectangle as a vector and arrows as functions (e.g., matrix multiply). Let us study the RNN topologies, from left to right:

1. **One-to-One** represents vanilla neural network (CNN) of processing without recurrent nets, from constant-sized input to constant-sized output (e.g., image classification).
2. **One-to-Many** represents a sequence output (e.g., image captioning acquires an image as input and outputs a sentence of words).
3. **Many-to-One** denotes sequence input (e.g., sentiment analysis where a known sentence is classified as stating positive or negative sentiment).
4. **Many-to-Many for Sequence Input and Sequence Output** (e.g., language translation: an RNN examines a sentence in English and then outputs it in French).
5. **Many-to-Many for Synced Sequence Input and Output** (e.g., video classification where we want to label every frame). It should be noted that there are no specific constraints on the lengths of sequences because the recurrent transformation (green) is not fixed and can be applied as many times as we want.

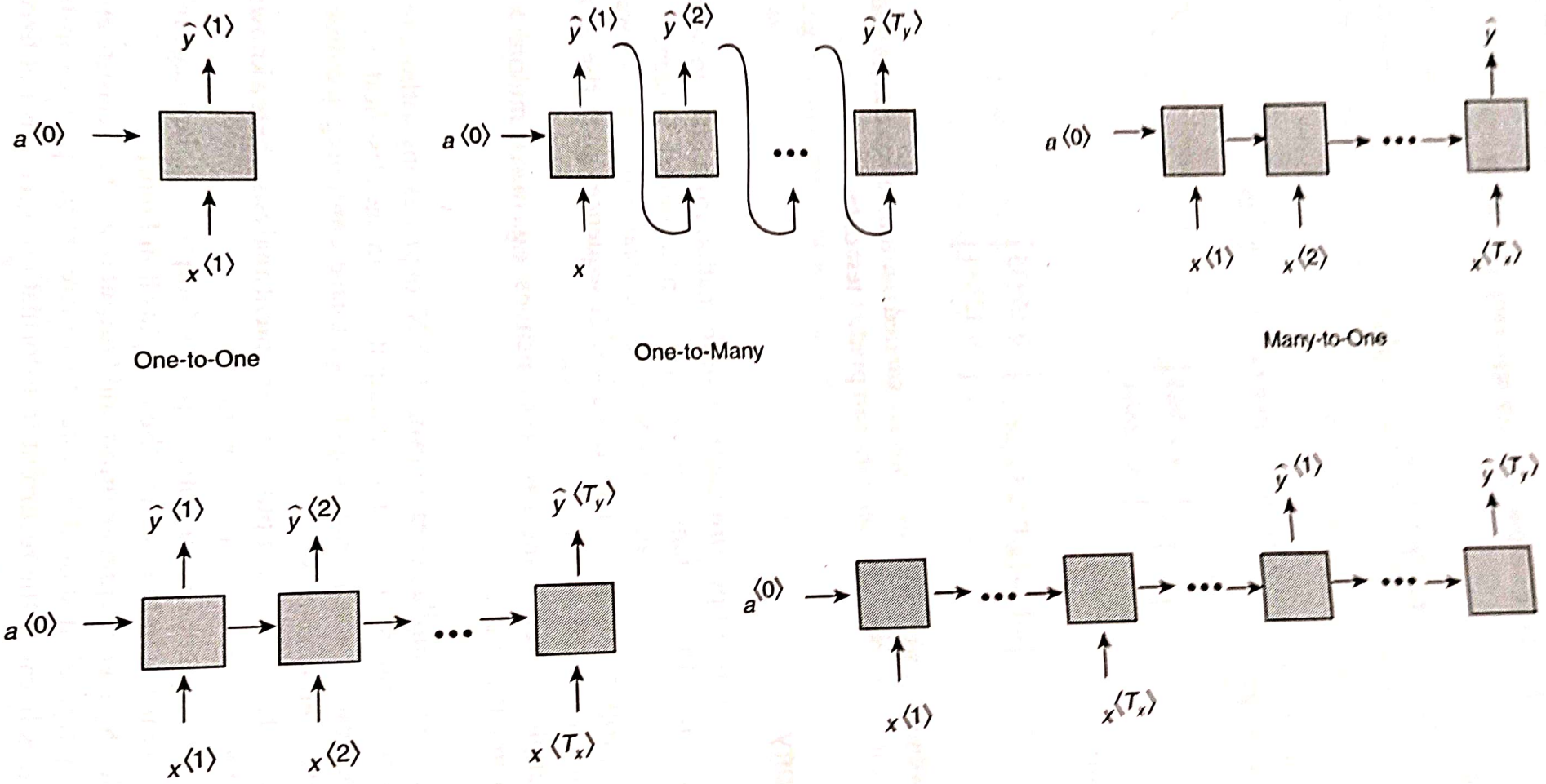


FIGURE 3.6 RNN topologies.

3.2.4 | Challenges with Vanishing Gradients

By the early 1990s, the *vanishing gradient problem* appeared as a major difficulty in recurrent net performance. Just as a straight line conveys a change in x alongside a change in y, the gradient conveys the change in all weights with respect to the change in error. If the gradient is unknown, weights cannot be tuned in a direction that will decrease error, and the network stops to learn.

Recurrent nets that are looking to establish connections between a final output and events many time steps before were shambled because of the complexity involved to know how much significance to accord to remote inputs. This is partly because the information flowing via the neural nets passes through several stages of multiplication.

For example, let us consider an example from language modelling. Note the two sentences: 'The **dog** that ate a bunch of apples, pears, carrots and bananas **was** full' and 'The **dogs** that ate a bunch of apples, pears, carrots and bananas **were** full'. This is an example of when language can have very long-term dependencies, but basic RNN is not very good at capturing very long-term dependencies because of vanishing gradient problems.

People who have learnt compound interest know that any quantity multiplied recurrently by a value slightly greater than 1 can become infinitely large (indeed, this simple mathematics emphasizes inevitable social inequalities and network effects). Its inverse, that is, multiplying by a value less than 1 will consequently vanish, is also true. For example, gamblers lose all their money when they win just 97 paise on each rupee they put in a game.

Because the time steps and layers of deep neural networks communicate to each other via multiplication, derivatives may vanish or explode in most scenarios.

The gradients may also increase exponentially with the number of hidden layers; this is called *exploding gradient problem*. It turns out that vanishing gradients tends to be a bigger problem with training RNNs, although when exploding gradients happens, it leads to exponentially large gradients which makes the neural network parameters get really large or NaN.

To solve these issues, exploding gradient problem can be trimmed or squashed and are thus relatively easy to solve. Similarly, vanishing gradients can evolve too tiny for computers to work with or for networks to learn. Thus, it is tedious to solve vanishing gradients problems.

Figure 3.7 is helpful in observing the effects of applying a sigmoidal function over and over again. The data is levelled unless, for large stretches, the slope is not detectable. This resembles gradient vanishing as it passes through many layers.

Just as in traditional neural networks, training the RNN involves backpropagation as well. The only difference is that the parameters are shared across all time steps. The gradient at every output depends not only on the present time step, but also on the previous ones. As already mentioned in Section 3.2.1, this procedure is called backpropagation through time, or BPTT in short.

Consider the small three-layered RNN shown in Fig. 3.8. During the forward propagation (shown in solid lines), the network creates predictions that are compared to the labels to calculate a loss L_t at each time step. During backpropagation (shown in dotted lines), the gradients of the loss with regard to the parameters U, V, and W are calculated at every time step and the parameters are updated with the summation of the gradients.

Equation (3.3) shows the gradient of the loss with respect to W, the matrix that encodes weights for long-term dependencies. This part of the update has to be focused because it is the reason for the

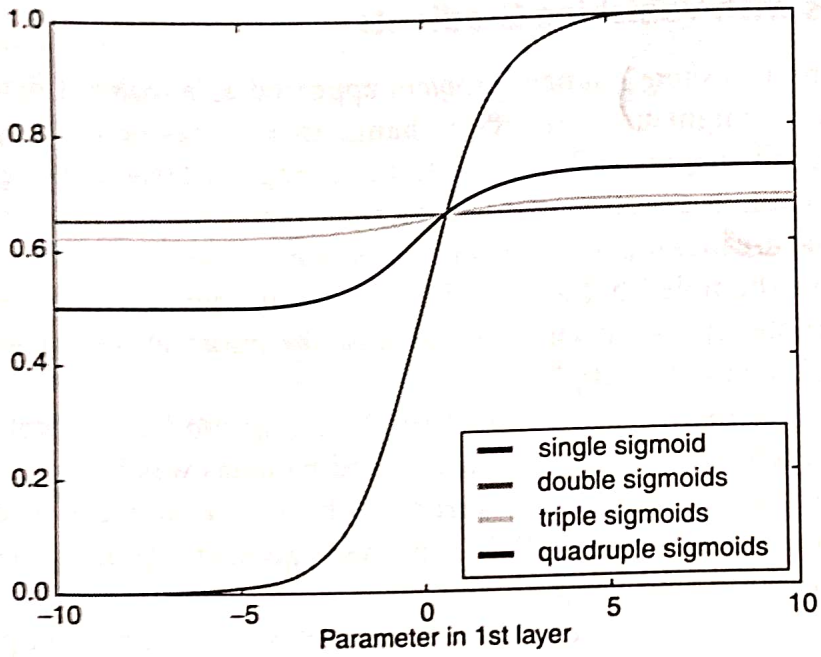


FIGURE 3.7 Sigmoid function applied multiple times.

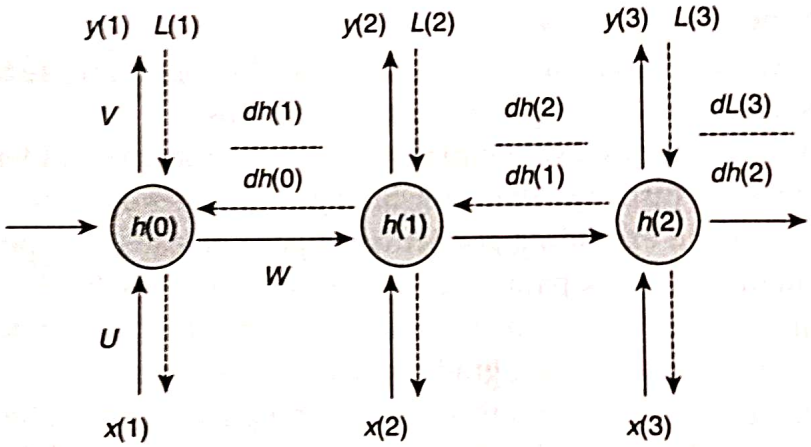


FIGURE 3.8 Three-layered RNN.

vanishing and exploding gradient problems. The other two gradients of loss with respect to matrices U and V are also summed up across all time steps in a similar way:

$$\frac{\partial L}{\partial W} = \sum_t \frac{\partial L_W}{\partial t} \tag{3.3}$$

Let us now look at what happens to the gradient of loss at the last time step ($t = 3$). As seen, this gradient can be decomposed to a product of three subgradients using the chain rule. The gradient of the hidden state h_2 with regards to W can still be decomposed as the summation of the gradient of every hidden state with regard to the previous one. Finally, each gradient of the hidden state with regard to the previous one can be further decomposed as the product of gradients of the present hidden state against the previous one.

$$\begin{aligned}\frac{\partial L_3}{\partial W} &= \frac{\partial L_3}{\partial y_3}\cdot\frac{\partial y_3}{\partial h_2}\cdot\frac{\partial h_2}{\partial W} \\ &= \sum_{t=0}^{2}\frac{\partial L_3}{\partial y_3}\cdot\frac{\partial y_3}{\partial h_2}\cdot\frac{\partial h_2}{\partial h_t}\cdot\frac{\partial h_t}{\partial W} \\ &= \sum_{t=0}^{2}\frac{\partial L_3}{\partial y_3}\cdot\frac{\partial y_3}{\partial h_2}\cdot\left(\prod_{j=t+1}^{2}\frac{\partial h_j}{\partial h_{j-1}}\right)\cdot\frac{\partial h_t}{\partial W}\end{aligned} \tag{3.4}$$

Similar calculations are made to compute the gradient of losses L_1 and L_2 (at time steps 1 and 2, respectively) with regard to W and to add them into the gradient update for W. The last form of the gradient given by Eq. (3.4) tells us why RNNs have the drawback of vanishing and exploding gradients. Consider the case where the individual gradient of a hidden state with regard to the previous one is less than one. As we backpropagate across several time steps, the product of gradients gets smaller, causing the problem of vanishing gradients. Likewise, if the gradients are greater than one, the products get larger, causing the problem of exploding gradients.

The consequence of vanishing gradients is that the gradients from steps that are at a distant do not contribute anything to the learning process. Thus, the RNN does not learn long-range dependencies. Vanishing gradients might happen for traditional neural networks as well; it is just more visible in case of RNNs, as they tend to have many more layers (time steps) over which backpropagation occurs.

Exploding gradients are more effortlessly detectable; the gradients will become very big and then turn into NaN and the training process will collapse. Exploding gradients can be controlled by cutting them at a predefined threshold. While there are a few approaches to diminish the problem of vanishing gradients, such as using ReLU instead of tanh layers, proper initialization of W matrix, and pre-training the layers using unsupervised methods, the most accepted solution is the use of LSTM or gated recurrent unit (GRU) architectures. These architectures have been designed to treat the vanishing gradient problem and to learn long-term dependencies more effectively. These architectures are discussed in subsequent sections in this chapter.

3.2.5 | Bidirectional and Stateful RNNs

At time step t, the output of the RNN is dependent on the outputs of all previous $t - n$ time steps where $n = 1, 2, ..., n - 1$. However, it is also possible that the output is reliant on future outputs as well. This is very true for applications related to Neuro-Linguistic Programming, where the attributes of the word or phrase we are trying to predict may be dependent on the context given by the entire enclosing sentence, and not only the words that come before it.

Bidirectional RNNs also help the network architecture to provide equal emphasis on the start and end of the sequence and increase the data accessible for training. Bidirectional RNNs are nothing but two RNNs stacked on top of each other, which interpret the input in opposite directions (Fig. 3.9). For example, one RNN will interpret the words right to left and the other will interpret the words left to right. The hidden state of both RNNs will decide the output at every time step.

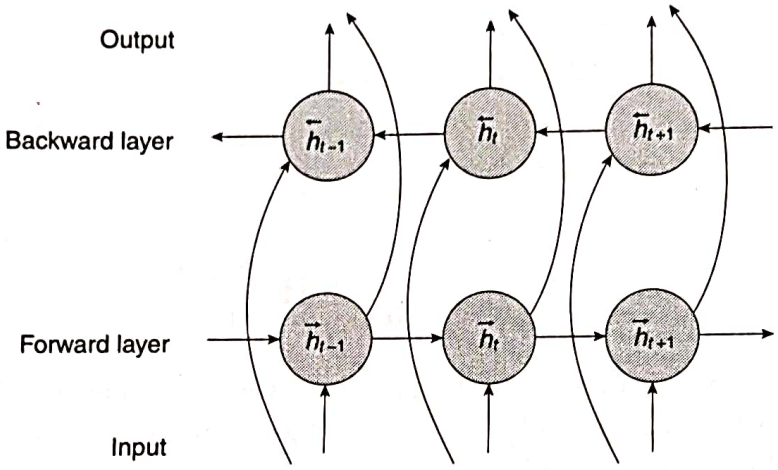


FIGURE 3.9 Bidirectional RNN.

RNNs may be stateful. This means that they can maintain state across various batches while training. The hidden state calculated for a batch of training data shall be used as the initial hidden state for the subsequent batch of training data. However, this has to be explicitly set, as RNNs are stateless by default and resets the state after every batch.

Making an RNN to be stateful means that it can assemble a state across its training sequence and even preserve that state while doing predictions. The benefits of utilizing stateful RNNs are lower training time and/or smaller network sizes. The disadvantage of stateful RNNs is that network has to be trained with a batch size that replicates the periodicity of the data, and the state is reset after each epoch. In addition, data must not be jumbled up while training the network, since the order in which the data is offered is relevant for stateful networks.

3.2.6 | Long Short-Term Memory (LSTM)

A variation of the recurrent net with LSTM units was projected by German researchers Sepp Hochreiter and Juergen Schmidhuber in the mid-1990s. LSTM serves as a remedy to the vanishing gradient problem.

LSTM helps to store up the error that can be backpropagated through time and layers. By preserving a steady constant error, recurrent nets are allowed to continue to learn over several time steps (over 1000), thereby opening a channel to connect causes and effects remotely. This is one of the vital challenges to artificial intelligence and machine learning since algorithms are often confronted by environments where reward signals are sparse and delayed, such as life itself. (Religious thinkers have handled this equivalent problem with ideas of karma or divine reward, theorizing invisible and distant cost to our actions).

LSTMs hold information in a gated cell which is outside the normal flow of the recurrent network. Information can be accumulated in, written to, or read from a cell, much like data in a computer's memory. The cell makes its decisions on what to accumulate, and when to allow reads, writes and deletes, through gates that open and close. Generally, storage in computers is in digital mode. However, these gates are analog in nature, executed with element-wise multiplication by sigmoids that are all in the range between 0 and 1. The advantage of being analog is that it is differentiable, and therefore suitable for backpropagation.

The gates act upon the signals they receive, and like neural network's nodes, they block or pass on information based on its strength and import, which are filtered with their own sets of weights. Those weights, like the weights that modulate input and hidden states, are altered via the recurrent network's learning process. That is, the cells learn when to permit the data to leave, enter or be deleted via the iterative process of assuming guesses, backpropagating errors, and altering the weights by gradient descent.

Figure 3.10 demonstrates how data flows through a memory cell and is controlled by means of its gates.

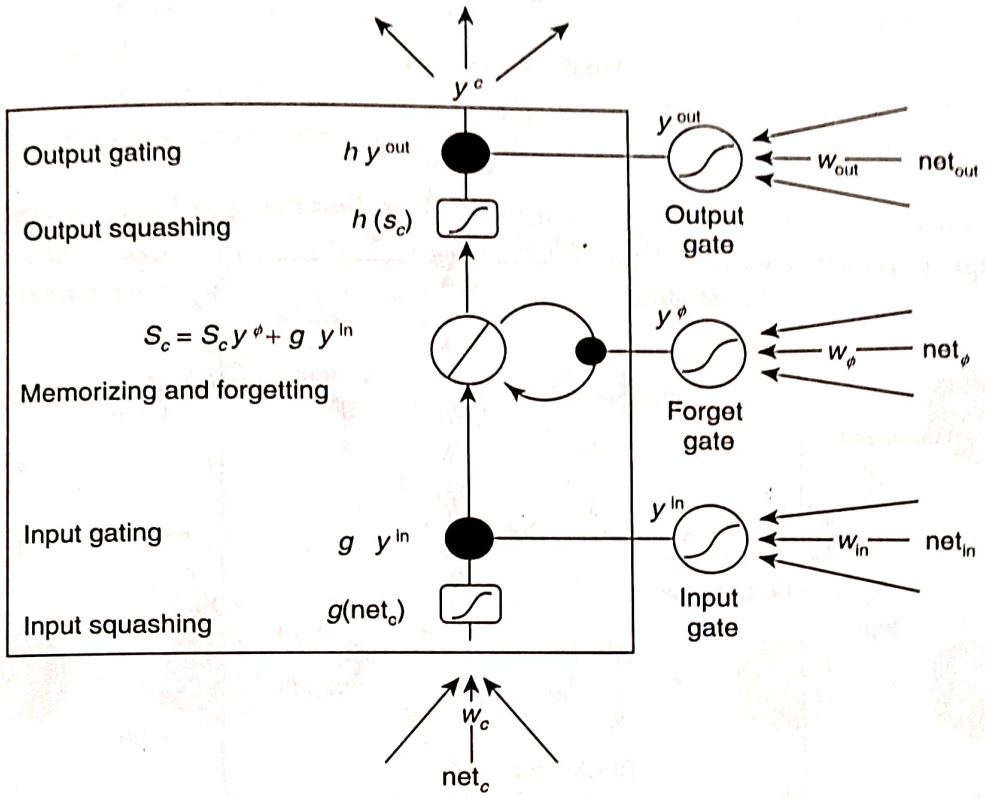
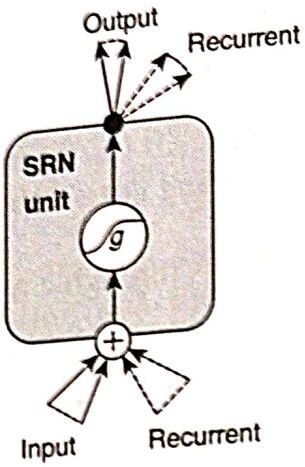


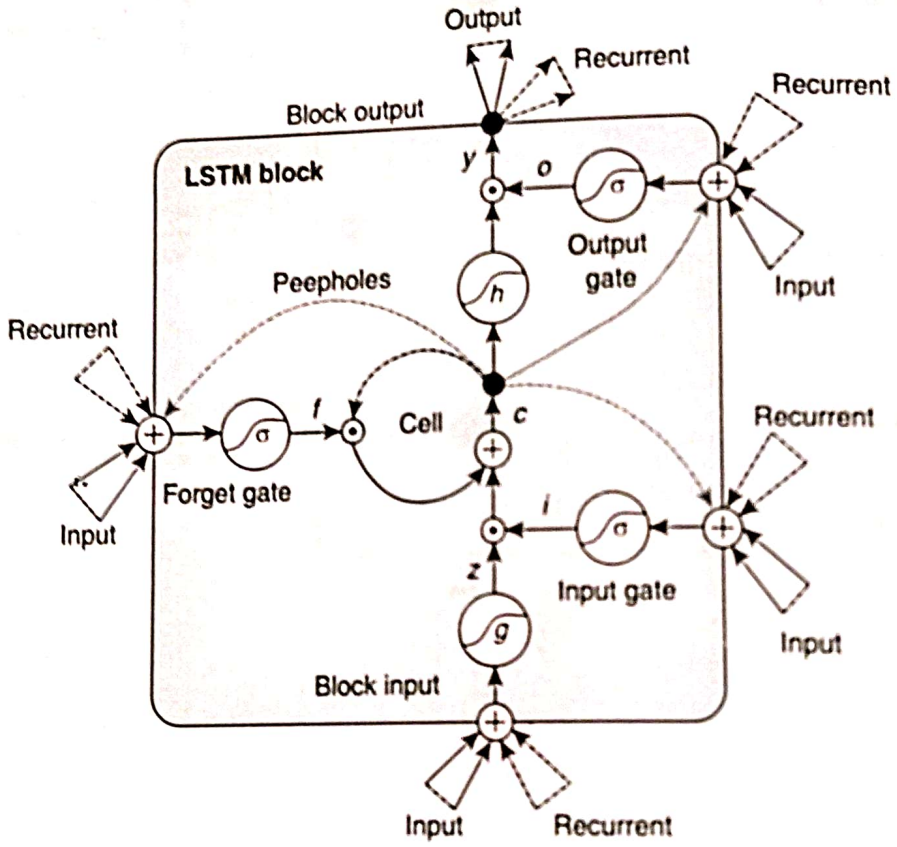
FIGURE 3.10 Data flow through a memory cell.

In Fig. 3.10, the three arrows at the bottom indicate where information flows into the cell at multiple points. That arrangement of present input and past cell state is fed not only to the cell itself, but also to all the three gates, which will decide on handling inputs. The black dots represent the gates themselves. These analog gates decide whether to permit new input in, remove the present cell state, and/or let that state impact the network's output at the current time step. S_c denotes the current state of the memory cell, and gy_{in} denotes the current input to it. It should be noted that a gate might be shut or open, and memory will recombine their shut and open states at every step. The cell can fail to remember its state or not; be written to or not; and be read from or not, at each time step. These flows are shown in Fig. 3.11.

Figure 3.11 compares a simple recurrent network [Fig. 3.11 (a)] to an LSTM cell [Fig. 3.11(b)]. The lines coming to summing points can be ignored for now.



(a)



(b)

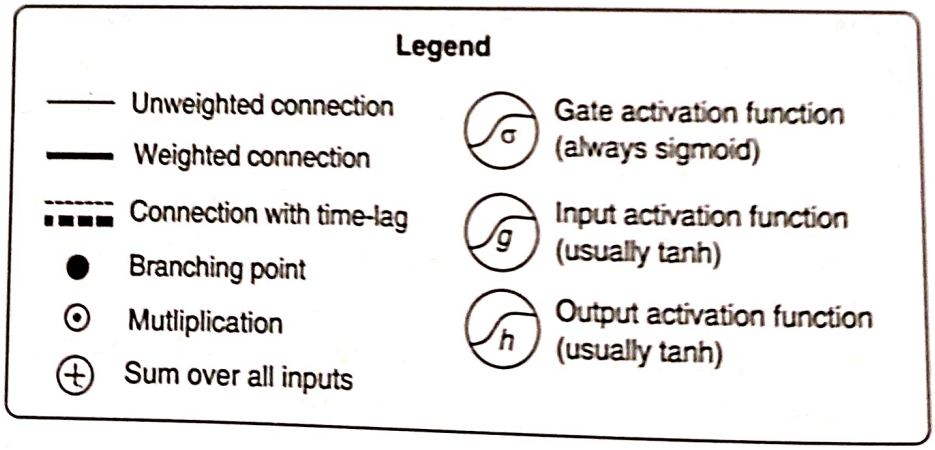


FIGURE 3.11 Detailed schematic of (a) simple RNN and (b) LSTM.

It is essential to note that the memory cells of LSTMs give different roles to multiplication and addition in the transformation of input. The central plus sign (+) in both figures is essentially the secret of LSTMs. This simple basic change helps them maintain a constant error when it has to be backpropagated at depth. As a substitute for figuring the next cell state by multiplying its current state with new input, they sum the current state and new input. This makes a lot of difference. Many different sets of weights filter the input for input, output and forgetting. A linear identity function is used to characterize the forget gate. This is because when the gate is open, the present state of the memory cell is just multiplied by 1 to transmit forward one more time step.

Moreover, while we are on the topic of simple hacks, including a bias of 1 to the forget gate of each LSTM cell is also shown to improvise performance.

In Fig. 3.12, you can observe the gates at work, with straight lines characterizing closed gates and blank circles characterizing open ones. The forget gates are the circles and lines that run below the hidden layer.

In general, feedforward networks map one input to one output. On the other hand, recurrent nets map one-to-many, as mentioned in Fig. 3.12 one image to many words in a caption), many-to-many (language translation), or many-to-one (segregating a voice).

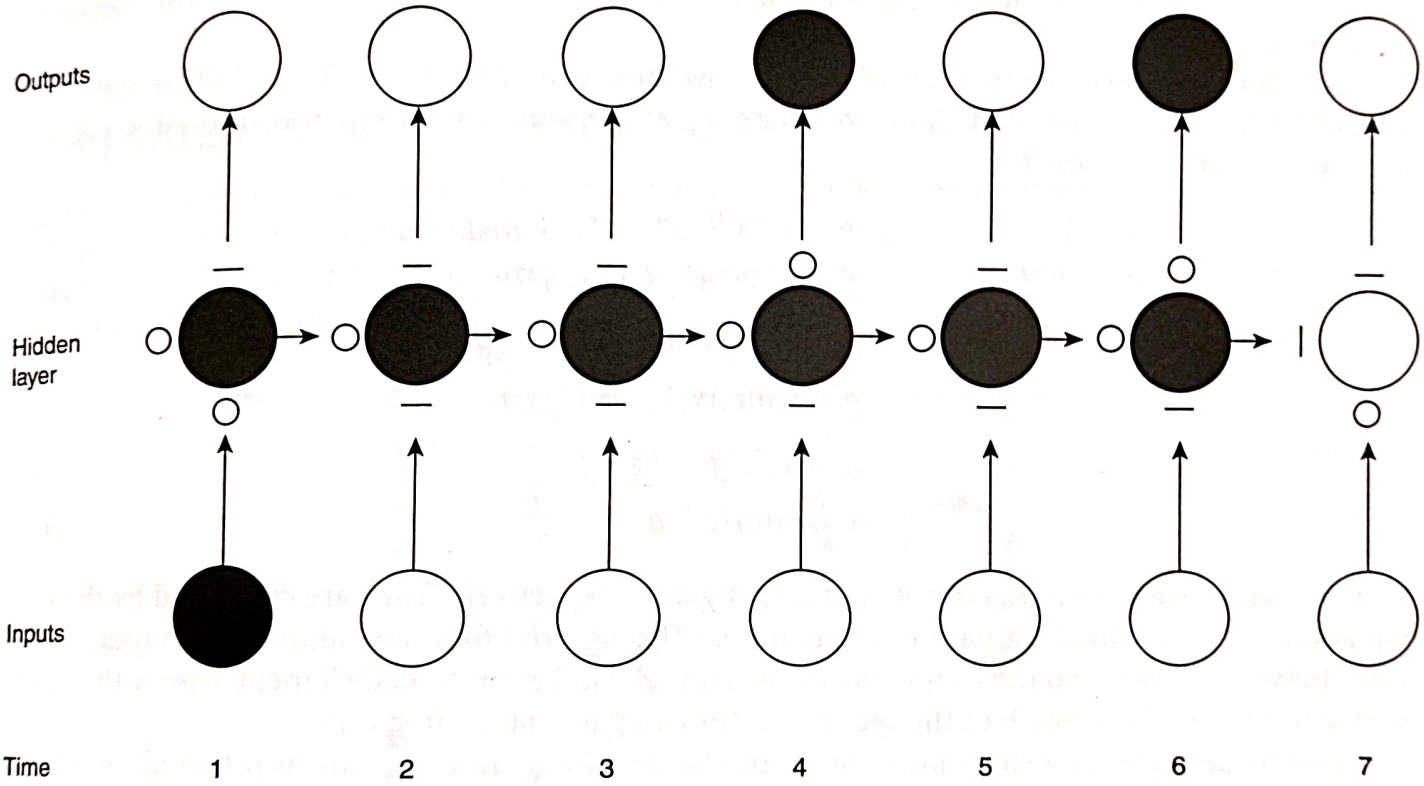


FIGURE 3.12 Input, output and forget gates.

A simple RNN utilizes the hidden state from the previous time step and the present input in a tanh layer to execute recurrence. Similar to simple RNN, LSTMs also execute recurrence. But instead of a single tanh layer, there are four layers acting together in a particular way. The transformations that are applied on the hidden state at time step t are illustrated in Fig. 3.13.

The line across the top of the figure is the cell state c; it signifies the internal memory of the unit. The line across the bottom is the hidden state, and the i, f, o, and g gates are the means by which the

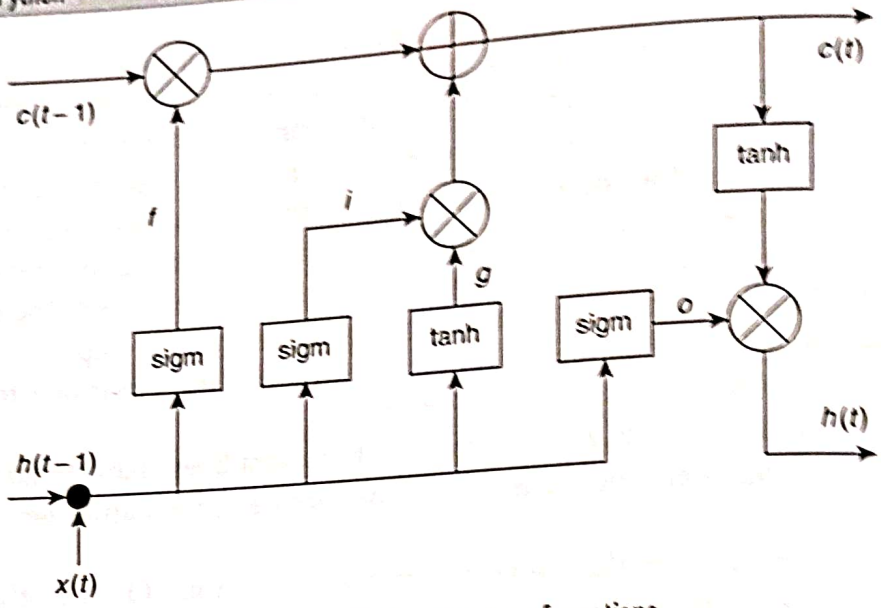


FIGURE 3.13 Hidden state transformations.

LSTM works on the vanishing gradient problem. During training, the LSTM trains the parameters for these gates.

In order to achieve a deeper knowledge of how these gates adjust the LSTM's hidden state, let us consider Eqs. (3.5)–(3.10) that show how state h_{t-1} at the previous time step is utilized to calculate the hidden state h_t at time t:

$$i = \sigma(W_i h_{t-1} + U_i x_t) \tag{3.5}$$

$$f = \sigma(W_f h_{t-1} + U_f x_t) \tag{3.6}$$

$$o = \sigma(W_o h_{t-1} + U_o x_t) \tag{3.7}$$

$$g = \tanh(W_g h_{t-1} + U_g x_t) \tag{3.8}$$

$$c_t = (c_{t-1} \circ f) + (g * i) \tag{3.9}$$

$$h_t = \tanh(c_t) \circ o \tag{3.10}$$

Here i, o and f are the input, output, and forget gates, respectively. They are calculated by the same equations but with different parameter matrices. The sigmoid function adjusts the output of these gates between 0 and 1, so the output vector generated can be multiplied element-wise with another vector to identify how much of the second vector can pass via the first one.

The forget gate describes how much of the previous state h_{t-1} you want to allow to pass through. The input gate describes how much of the newly calculated state for the present input x_t you wish to let through. The output gate describes how much of the internal state you wish to render to the next layer.

The present input x_t and the previously hidden state h_{t-1} is utilized to calculate the internally hidden state g. Note that the equation for g is similar to that for the simple RNN cell, but in this case, we will adjust the output by the output of the input gate i.

Given i, f, o and g, the cell state c_t is computed at time t in terms of c_{t-1} at time $(t-1)$ multiplied by the forget gate and the state g multiplied by the input gate i. So, this is basically a method to merge

the previous memory and the new input; resting the forget gate to 0 disregards the old memory and resting the input gate to 0 disregards the newly computed state. Finally, the hidden state h_t at time t is calculated by multiplying the memory c_t with the output gate.

3.3 | LSTM IMPLEMENTATION

The implementation steps of LSTM are as follows:

STEP 1: Implement a single LSTM cell by concatenating h_{t-1} and x_t matrices and by computing Eqs. (3.5)–(3.10) given above.

STEP 2: Compute the prediction at single time step $y(t)$ using softmax function.

STEP 3: Iterate this single LSTM cell over and over using a for-loop to process a sequence of T_x inputs.

We need to note that an LSTM is a drop-in substitute for a simple RNN cell, the only difference being LSTMs are resistant to the vanishing gradient problem. An RNN cell in a network can be replaced with an LSTM without being concerned about any side effects. Better results can be seen along with longer training times.

3.4 | GATED RECURRENT UNIT (GRU)

The GRU is a variant of the LSTM and was put forth by K. Cho. It preserves the LSTM's resistance to the vanishing gradient problem, but its internal structure is simpler. Thus, it can be trained more rapidly, since fewer calculations are required to make updates to its hidden state. Figure 3.14 represents the gates for a GRU cell.

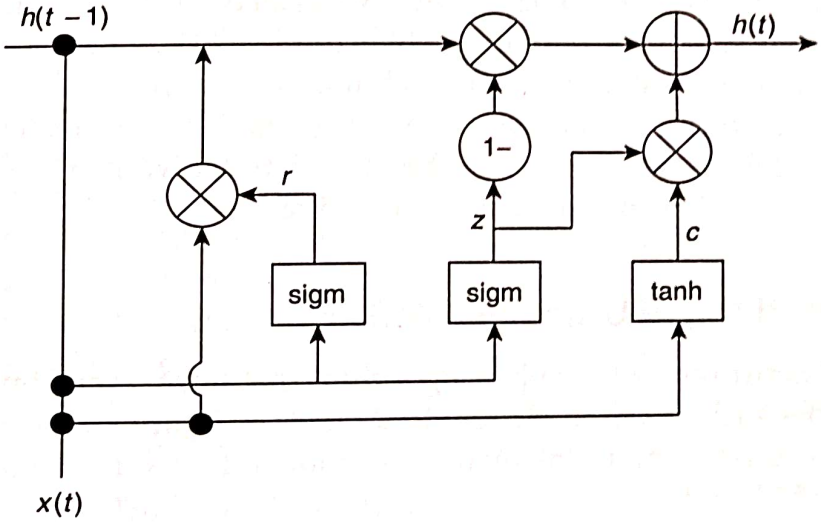


FIGURE 3.14 Gates for a gated recurrent unit (GRU) cell.

It is known that LSTM has three gates: input, forget, and output gates. The GRU cell has two gates: a reset gate r and an update gate z. The update gate describes how much previous memory to maintain around and the reset gate describes how to merge the new input with the previous memory.

GRU has no persistent cell state different from the hidden state as in LSTM. The gating mechanism in GRU is described by Eqs. (3.11)–(3.14).

$$z = \sigma(W_z h_{t-1} + U_z x_t) \tag{3.11}$$

$$r = \sigma(W_r h_{t-1} + U_r x_t) \tag{3.12}$$

$$c = \tanh[W_c(h_{t-1} \circ r) + U_c x_t] \tag{3.13}$$

$$h_t = (z \circ c) + [(1-z) \circ h_{t-1}] \tag{3.14}$$

GRU and LSTM have equivalent performance and there is no easy way to recommend one or the other for a specific task. While GRUs are quicker to train and need fewer data to generalize, in situations where there is enough data, an LSTM's better expressive power might lead to enhanced results. Like LSTMs, GRUs are drop-in alternatives for the simple RNN cell.

The general idea of utilizing a gating mechanism to learn long-term dependencies is almost the same as in a LSTM, but a few key differences are as follows:

1. GRU has two gates (reset gate r and update gate z), while LSTM has three gates (input, forget, and output gates).
2. GRUs do not have an internal memory (c_t) which is different from the exposed hidden state. They do not include the output gate that is present in LSTMs.
3. An update gate z is the combination of input and forget gates and the reset gate r is applied directly to the previous hidden state. Thus, the responsibility of the reset gate in GRU is really split up into both r and z. A second non-linearity is not applied for output computation.

Thus, the vanishing gradient problem can be cracked either by using LSTM or GRU. GRUs are quite recent (2014), and their effects have not been fully investigated yet. There is no clear winner between GRU and LSTM. In many situations, both the architectures yield comparable performances. Tuning hyperparameters like layer size is probably of much importance than picking an ideal architecture. Since GRUs have less parameters compared to LSTMs (U and W are smaller), they may train a bit quicker or need lesser data to generalize. On the other hand, if you have enough data, the superior expressive power of LSTMs may lead to improved results.

3.5 | DEEP RECURRENT NEURAL NETWORK

Deep artificial neural networks are a group of algorithms that solve several important problems with much higher accuracies such as sound recognition, image recognition, recommender systems, etc. *Deep* is a technical word referring to the quantity of hidden layers in a neural network. A shallow network has only one hidden layer, while a deep recurrent neural network has more than one.

Depth is an important parameter that differentiates deep learning networks from the more common single-hidden-layer neural networks. Depth refers to the number of node layers via which data travels in a multistep process of pattern recognition. Former versions of neural networks such as the first perceptrons were shallow, comprising single input and single output layer, and with a maximum of one hidden layer in between. Any network that has greater than

three layers (including input and output) qualifies as 'deep' learning. So, the technical term 'deep' refers to more than one hidden layer (Simon, 2008).

In deep neural networks, the previous layer's output is used to train every layer of nodes on a unique set of features. As we advance into a neural net, the nodes can recognize more complex features since they cumulate and recombine features from the previous layer. This is called *feature hierarchy* – it is a hierarchy of rising complexity and abstraction. Feature hierarchy enables deep learning neural networks capable of handling huge, multidimensional datasets with millions of parameters passing along non-linear functions.

Majority of data generated in the world are categorized under unlabeled, unstructured data. Unstructured data is also termed as *raw media*; these include texts, pictures, video and audio recordings. Processing and clustering of the world's raw, unlabeled media, identifying the similarities and anomalies in data that no human can organize in a relational database can be done by utilizing deep networks. These deep networks are skilled in discovering patterns or structures within such datasets (Simon, 2008).

For example, deep learning can acquire a million images, and cluster them based on their similarities: dogs in one cluster, ice breakers in another, and in a third all the photos of your grandfather. This is the basis of smart photo albums.

Deep learning can cluster raw text such as news articles or emails. Emails can be classified as spam/not spam or emails filled with angry complaints from unhappy customers may be clustered in one corner of the vector space, while satisfied customers, or spambot messages, may be clustered in another vector space. This is the foundation of several messaging filters and can be utilized in customer relationship management (CRM). Voice messages can also be clustered using equivalent principle. With time series, data might cluster around healthy/normal behavior and dangerous/anomalous behavior. If the time-series data is being produced from a smartphone, it will give an insight to the user's habits and health; if it is being produced by a vehicle, it might be used to prevent catastrophic breakdowns.

Deep learning neural network performs automatic feature extraction without human interventions, unlike all traditional machine-learning methods. Given that feature extraction is a task that can take teams of data scientists years to finish, deep learning is a mode to circumvent the chokepoint that limits experts. It enhances the powers of small data science teams, which by their nature do not scale.

When training on raw data, every node layer in a deep network learns features automatically by constantly trying to rebuild the input from which it draws the samples, trying to minimize the error between the network's estimation and the probability distribution of the input data itself. An example of such reconstruction happens in restricted Boltzmann machines (RBMs).

In this process, these networks are trained to recognize correlations between some relevant features and optimal results – they illustrate connections between feature signals and what those features represent, whether it be a full reconstruction or has labeled data.

A deep learning network trained on labeled data can then be implemented on unstructured data, providing its access to much more input than machine-learning nets. This is a recipe for enhanced performance: the more data a net can train upon, the more precise it is likely to be. (Bad algorithms trained on lots of data can do better than good algorithms trained on very little data). Deep learning's capability to process and learn from vast quantities of unlabeled data gives it a unique advantage over previous algorithms.

The last layer of deep neural network is an output layer which assigns a logistic, or softmax, classifier that assigns likelihood to a particular label or outcome. We call that predictive, but it is predictive in a broad sense. Given raw data in an image form, a deep-learning network may decide, for example, that the input data is 90% likely to represent a person. Computational intense is one of the attributes of deep learning, and it is one cause why GPUs are in need to train deep learning models.

An RNN remembers each and every information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called long short term memory (LSTM). RNNs are even used with convolution layers to extend the effective pixel neighborhood. RNN converts the independent activations into dependent activations by providing the same weights and biases to all the layers, thus reducing the complexity of increasing parameters and memorizing each previous outputs by giving each output as input to the next hidden layer.

SUMMARY

This chapter introduces the basic architecture of RNN and how it differs from the previously known ANN and CNN architectures. The backbone behind the RNN architecture, namely, BPTT, is explained in detail with an example. Other known variants like stateful RNN, LSTM and GRU have also been explained to help the reader analyze the pros and cons of each architecture.

REVIEW QUESTIONS

1. How is RNN different from other artificial neural network architectures?
2. What are the different RNN topologies, and why do we need them?
3. Name the major problem that hinders the performance of an RNN, and show how it can be overcome.
4. Find few applications where LSTM implementation gives better results when compared to other models and mention how LSTM performs better for that application.

ASSIGNMENT PROBLEMS

1. Given the truth table for XOR logic, use a simple RNN with weights and architecture same as that explained in Section 3.2.2, and calculate the output of the hidden layer.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

2. Apply sigmoid activation to the output of the hidden layer and find the output.
3. Compute error and repeat the process until the model converges.

REFERENCES

1. Britz, D. (2016). *Recurrent Neural Network Tutorial, Part 4 – Implementing a GRU/LSTM RNN with Python and Theano*. Available: http://www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano.
2. Gulli, A. and Pal, S. (2017). *Deep Learning with Keras*. Packt Books. Available: https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-keras.
3. Jain, C. and Medsker, L. (2000). *Recurrent Neural Networks: Design and Applications*. Boca Raton, Florida: CRC Press.
4. Nicholson, C. V. and Gibson, A. (2001). *A Beginner's Guide to Recurrent Networks and LSTMs*. Available: https://deeplearning4j.org/lstm.
5. Nicholson, C. V. and Gibson, A. (2002). *Introduction to Deep Neural Networks (Deep Learning)*. Available: https://deeplearning4j.org/neuralnet-overview.
6. "Recurrent Neural Networks TensorFlow," TensorFlow. Available: https://www.tensorflow.org/tutorials/recurrent.
7. Simon, H. (2008). *Neural Networks: A Comprehensive Foundation*. New Delhi: Prentice-Hall of India.