USAII™ UNITED STATES ARTIFICIAL INTELLIGENCE INSTITUTE

PERSONALIZED OFFICIAL USAII EXAM-PREPARATION RESOURCE

02

STUDY BOOK - 2

ADVANCED ARTIFICIAL INTELLIGENCE ESSENTIAL

FOR CERTIFIED ARTIFICIAL INTELLIGENCE ENGINEER - CAIE™

CAIE™

Certified Artificial Intelligence Engineer

Packt

USAII

Executive Edition: March 2021.

Published by: Packt

ISBN: 978-1-80181-967-1

Preface

Artificial Intelligence is around since the 1950s, and many attempts have been made in the past to bring in revolutionary changes with multiple tools and technologies for the benefit of the human race. However, it couldn't kick off well due to funding issues, though in 1970, the American computer scientist Marvin Minsky said, "from three to eight years, we will have a machine with the general intelligence of an average human being." Well, even with the basic proof of principle that was available, there was still a long way to go before the end goals of natural language processing, abstract thinking, and self-recognition could be achieved. There was also another problem with the AI projects back then – the DATA. The data creation and storage were less due to inadequate computer systems and machines due to which the process of data mining and analysis was not possible for the best results.

Eventually, in the 1990s and 2000s, a lot was achieved – world chess champion the grandmaster Gary Kasparov was defeated by IBM's Deep Blue, a chess-playing computer system. Similar to Deep Blue, speech recognition programs, *Kismet* – a robot, were developed in the same era.

Everything is changed now, and the struggle AI has gone through has actually become to reality with a lot of awareness created by businesses and experts. Globally, almost everyone is using AI in some form or the other – say your smartphone. However, there is still a lot to achieve, and organizations are bracing up pretty quickly for their AI transformation journey. But, this journey has a problem – **shortage of talent force**. Many organizations globally are facing issues to take the AI projects forward since they do not have the right talent to bring pace to these projects. Professionals with the right AI skills can help the industry grow at a lightning-fast speed. To build in the top AI skills, the pedagogy of AI learning must be better with the latest trends and topics, use cases, and practice projects.

The United States Artificial Intelligence Institute (USAII™) provides the best-in-class certification programs to upskill and reskill the AI talent force in the world. This Study-Book will help you gain the foundational knowledge and concepts of Artificial Intelligence. Every topic in this book is looked for the relevant industry trend and has been matched with the USAII's knowledge framework to give you the best learning possible.

This Study-Book covers the AI essentials under eight modules that include – **Big Data and AI, Artificial Intelligence on the Cloud, Python for Coding, Machine Learning Pipeline, Chatbots, Essential Machine Learning Concepts, Supervised and Unsupervised Learning, Deep Learning Foundations, Tensor Flow, NLP Fundamentals, Computer Vision and the Raspberry Pi, GANs, Reinforcement Learning, Deep Reinforcement Learning.**

Candidates who sign-up with the United States Artificial Intelligence Institute (USAII™) under Certified Artificial Intelligence Engineer (CAIE™) program must know that the contents of this book have been exclusively chosen by our content partner Packt's best-selling database. The content is further carefully vetted and verified by 15 most prominent SMEs of the **USAII's Artificial Intelligence Advisory Board** – these subject matter experts are the influencers of the artificial intelligence domain and are CIOs, CTOs, Data Scientists, technology leaders from Fortune 500 companies. Hence, we are sure that that the content is apt and is future-ready.

Please note, if you are a candidate registered under the CAIE™ program by USAII™, you will be provided with a self-help study kit, which will help you prepare for the CAIE™ certification examination. This study kit includes three books in the eLearning format and can be downloaded from myControlPanel dashboard. Below are the tri-series Study Books:

1. Study Book 1: Artificial Intelligence Essential
2. Study Book 2: Advanced Artificial Intelligence Essential
3. Study Book 3: Artificial Intelligence Essential Reading

The study kit will also include certain other eLearning materials that can be accessed separately as per instructions. These study materials include the latest case studies, workshops, and other important AI events that will help you crack the CAIE™ certification assessment.

Packt is the worldwide official content partner for USAII™ and has produced three world-class CAIE™ certification study books for international use under exclusive permission from and under arrangement with USAII™. Assessment for CAIE™ certification is on-demand and can be taken almost anywhere in the world.

We hope you will find reading this book a great learning experience, and soon, we shall see you adding to the rapidly expanding list of USAII™ certified professionals internationally.

With best wishes,

Research & Authoring Team

United States Artificial Intelligence Institute Editorial Desk

Table of Contents

Module 4
Advanced Deep Learning

Chapter 20 Deep Learning Foundations

Deep learning is a subset of machine learning and it is all about neural networks. Deep learning has been around for a decade, but the reason it is so popular right now is because of the computational advancements and availability of huge volumes of data. With this huge volume of data, deep learning algorithms can outperform classic machine learning algorithms.

We will start off the chapter by understanding what biological and artificial neurons are, and then we will learn about Artificial Neural Networks (ANNs) and how to implement them. Moving forward, we will learn about several interesting deep learning algorithms such as the Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), Convolutional Neural Network (CNN), and Generative Adversarial Network (GAN).

Let's begin the chapter by understanding how biological and artificial neurons work.

Biological and artificial neurons

Before going ahead, first, we will explore what neurons are and how neurons in our brain actually work, and then we will learn about artificial neurons.

A neuron can be defined as the basic computational unit of the human brain. Neurons are the fundamental units of our brain and nervous system. Our brain encompasses approximately 100 billion neurons. Each and every neuron is connected to one another through a structure called a synapse, which is accountable for receiving input from the external environment via sensory organs, for sending motor instructions to our muscles, and for performing other activities.

A neuron can also receive inputs from other neurons through a branchlike structure called a dendrite. These inputs are strengthened or weakened; that is, they are weighted according to their importance and then they are summed together in the cell body called the soma. From the cell body, these summed inputs are processed and move through the axons and are sent to the other neurons.

Figure 20.1 shows a basic single biological neuron:

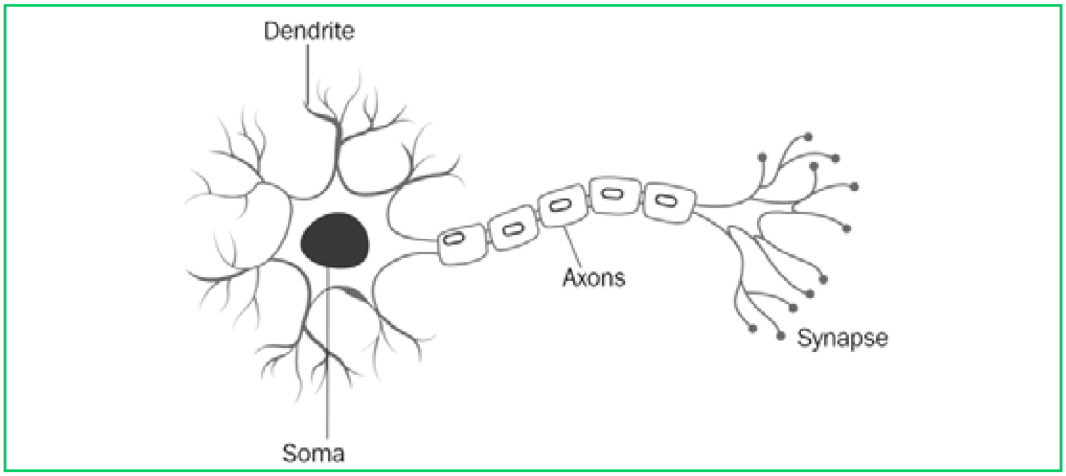


Figure 20.1: Biological neuron

Now, let's see how artificial neurons work. Let's suppose we have three inputs x_1, x_2, and x_3, to predict the output y. These inputs are multiplied by weights w_1, w_2, and w_3 and are summed together as follows:

$$x_1.w_1 + x_2.w_2 + x_3.w_3$$

But why are we multiplying these inputs by weights? Because all of the inputs are not equally important in calculating the output y. Let's say that x_2 is more important in calculating the output compared to the other two inputs. Then, we assign a higher value to w_2 than the other two weights. So, upon multiplying weights with inputs, x_2 will have a higher value than the other two inputs. In simple terms, weights are used for strengthening the inputs. After multiplying inputs with the weights, we sum them together and we add a value called bias, b:

$$z = (x_1.w_1 + x_2.w_2 + x_3.w_3) + b$$

If you look at the preceding equation closely, it may look familiar. Doesn't z look like the equation of linear regression? Isn't it just the equation of a straight line? We know that the equation of a straight line is given as:

$$z = mx + b$$

Here, m is the weights (coefficients), x is the input, and b is the bias (intercept).

Well, yes. Then, what is the difference between neurons and linear regression? In neurons, we introduce non-linearity to the result, *z*, by applying a function *f*(.) called the activation or transfer function. Thus, our output becomes:

$$y = f(z)$$

Figure 20.2 shows a single artificial neuron:

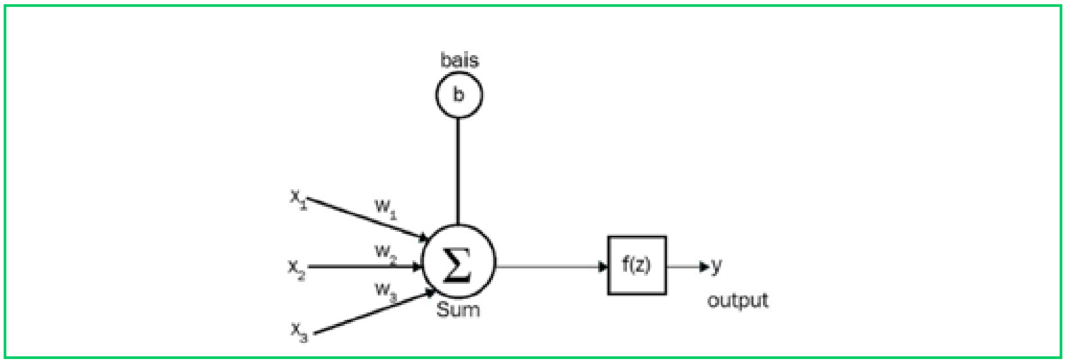


Figure 20.2: Artificial neuron

So, a neuron takes the input, *x*, multiples it by weights, *w*, and adds bias, *b*, forms *z*, and then we apply the activation function on *z* and get the output, *y*.

ANN and its layers

While neurons are really cool, we cannot just use a single neuron to perform complex tasks. This is the reason our brain has billions of neurons, stacked in layers, forming a network. Similarly, artificial neurons are arranged in layers. Each and every layer will be connected in such a way that information is passed from one layer to another.

A typical ANN consists of the following layers:

- Input layer
- Hidden layer
- Output layer

Each layer has a collection of neurons, and the neurons in one layer interact with all the neurons in the other layers. However, neurons in the same layer will not interact with one another. This is simply because neurons from the adjacent layers have connections or edges between them; however, neurons in the same layer do not have any connections. We use the term nodes or units to represent the neurons in the ANN.

Figure 20.3 shows a typical ANN:

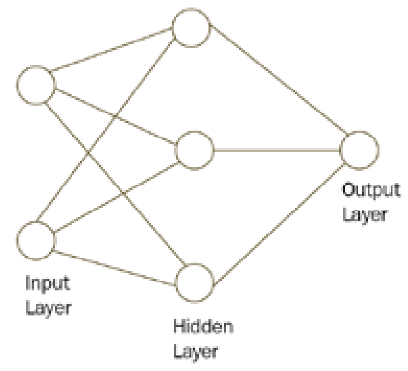


Figure 20.3: ANN

Input layer

The input layer is where we feed input to the network. The number of neurons in the input layer is the number of inputs we feed to the network. Each input will have some influence on predicting the output. However, no computation is performed in the input layer; it is just used for passing information from the outside world to the network.

Hidden layer

Any layer between the input layer and the output layer is called a hidden layer. It processes the input received from the input layer. The hidden layer is responsible for deriving complex relationships between input and output. That is, the hidden layer identifies the pattern in the dataset. It is majorly responsible for learning the data representation and for extracting the features.

There can be any number of hidden layers; however, we have to choose a number of hidden layers according to our use case. For a very simple problem, we can just use one hidden layer, but while performing complex tasks such as image recognition, we use many hidden layers, where each layer is responsible for extracting important features. The network is called a deep neural network when we have many hidden layers.

Output layer

After processing the input, the hidden layer sends its result to the output layer. As the name suggests, the output layer emits the output. The number of neurons in the output layer is based on the type of problem we want our network to solve.

If it is a binary classification, then the number of neurons in the output layer is one, and it tells us which class the input belongs to. If it is a multi-class classification say, with five classes, and if we want to get the probability of each class as an output, then the number of neurons in the output layer is five, each emitting the probability. If it is a regression problem, then we have one neuron in the output layer.

Exploring activation functions

An activation function, also known as a transfer function, plays a vital role in neural networks. It is used to introduce non-linearity in neural networks. As we learned before, we apply the activation function to the input, which is multiplied by weights and added to the bias, that is, *f*(*z*), where *z* = *(input * weights)* + *bias* and *f*(.) is the activation function.

If we do not apply the activation function, then a neuron simply resembles the linear regression. The aim of the activation function is to introduce a non-linear transformation to learn the complex underlying patterns in the data.

The sigmoid function

The sigmoid function is one of the most commonly used activation functions. It scales the value between 0 and 1. The sigmoid function can be defined as follows:

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

It is an S-shaped curve shown in *Figure 20.4*:

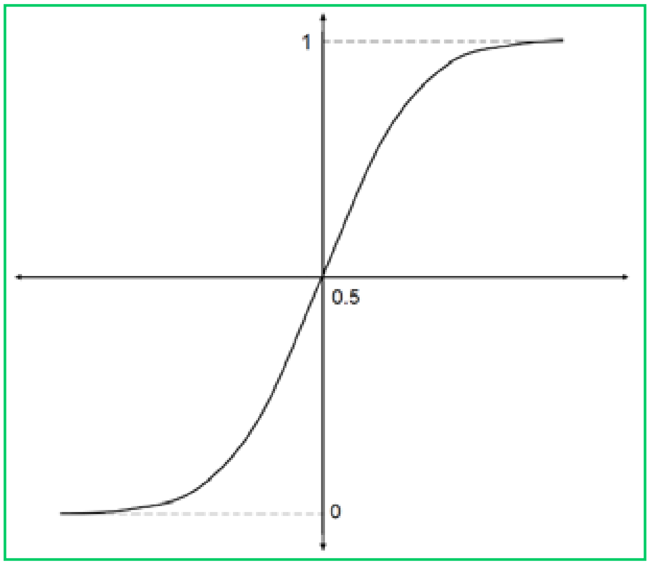


Figure 20.4: Sigmoid function

It is differentiable, meaning that we can find the slope of the curve at any two points. It is monotonic, which implies it is either entirely non-increasing or non-decreasing. The sigmoid function is also known as a logistic function. As we know that probability lies between 0 and 1, and since the sigmoid function squashes the value between 0 and 1, it is used for predicting the probability of output.

The tanh function

A hyperbolic tangent (tanh) function outputs the value between -1 to +1 and is expressed as follows:

$$f(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

It also resembles the S-shaped curve. Unlike the sigmoid function, which is centered on 0.5, the tanh function is 0-centered, as shown in the following diagram:

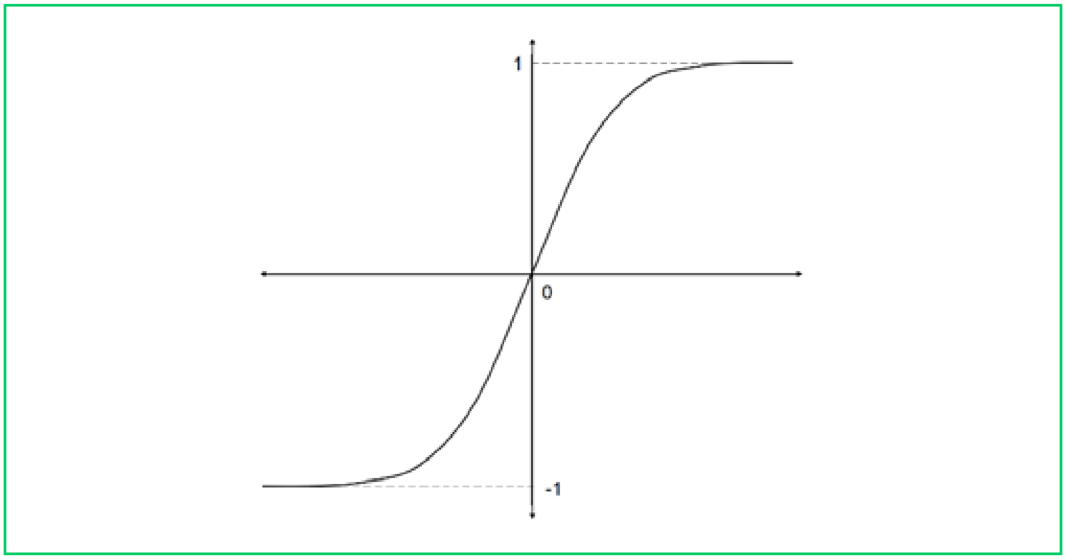


Figure 20.5: tanh function

The Rectified Linear Unit function

The Rectified Linear Unit (ReLU) function is another one of the most commonly used activation functions. It outputs a value from zero to infinity. It is basically a piecewise function and can be expressed as follows:

$$f(x) = \begin{cases} 0 & for\ x < 0 \\ x & for\ x \geq 0 \end{cases}$$

That is, $f(x)$ returns zero when the value of x is less than zero and $f(x)$ returns x when the value of x is greater than or equal to zero. It can also be expressed as follows:

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

Figure 20.6 shows the ReLU function:

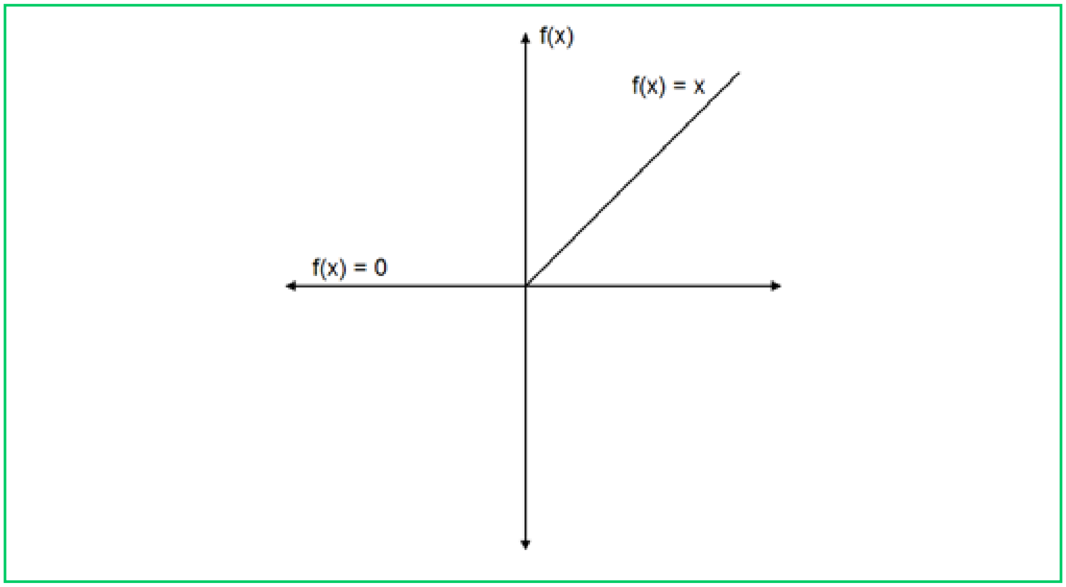


Figure 20.6: ReLU function

As we can see in the preceding diagram, when we feed any negative input to the ReLU function, it converts the negative input to zero.

The softmax function

The softmax function is basically the generalization of the sigmoid function. It is usually applied to the final layer of the network and while performing multi-class classification tasks. It gives the probabilities of each class for being output and thus, the sum of softmax values will always equal 1.

It can be represented as follows:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

As shown in the *Figure 20.7*, the softmax function converts its inputs to probabilities:

$$\begin{bmatrix}0.5\\1.3\\1.1\end{bmatrix} \rightarrow \boxed{\text{Softmax}} \rightarrow \begin{bmatrix}0.198\\0.440\\0.360\end{bmatrix}$$

Figure 20.7: Softmax function

Now that we have learned about different activation functions, in the next section, we will learn about forward propagation in ANNs.

Forward propagation in ANNs

In this section, we will see how an ANN learns where neurons are stacked up in layers. The number of layers in a network is equal to the number of hidden layers plus the number of output layers. We don't take the input layer into account when calculating the number of layers in a network. Consider a two-layer neural network with one input layer, x, one hidden layer, h, and one output layer, y, as shown in the following diagram:

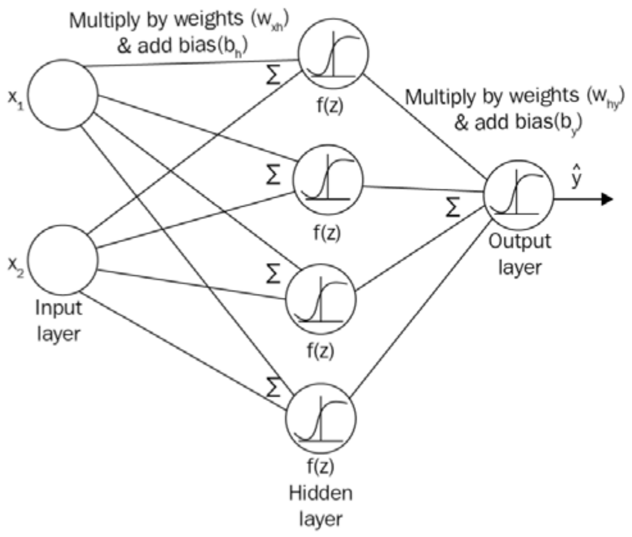

Multiply by weights (w_{xh}) & add bias(b_h)
x_1
x_2
Input layer
Σ
f(z)
Hidden layer
Multiply by weights (w_{hy}) & add bias(b_y)
$\hat{y}$
Output layer

Figure 20.8: Forward propagation in ANN

Let's consider we have two inputs, x_1 and x_2, and we have to predict the output, $\hat{y}$. Since we have two inputs, the number of neurons in the input layer is two. We set the number of neurons in the hidden layer to four, and the number of neurons in the output layer to one. Now, the inputs are multiplied by weights, and then we add bias and propagate the resultant value to the hidden layer where the activation function is applied.

Before that, we need to initialize the weight matrix. In the real world, we don't know which input is more important than the other so that we can weight them and compute the output. Therefore, we randomly initialize the weights and bias value. The weight and the bias value between the input to the hidden layer are represented by W_{xh} and b_h, respectively. What about the dimensions of the weight matrix? The dimensions of the weight matrix must be *the number of neurons in the current layer* x *the number of neurons in the next layer*. Why is that?

Because it is a basic matrix multiplication rule. To multiply any two matrices, AB, the number of columns in matrix A must be equal to the number of rows in matrix B. So, the dimension of the weight matrix, W_{xh}, should be *the number of neurons in the input layer* x *the number of neurons in the hidden layer*, that is, 2 x 4:

$$z_1 = XW_{xh} + b_h$$

The preceding equation represents, $z_1 = \text{input} \times \text{weights} + \text{bias}$. Now, this is passed to the hidden layer. In the hidden layer, we apply an activation function to z_1. Let's use the sigmoid σ activation function. Then, we can write:

$$a_1 = \sigma(z_1)$$

After applying the activation function, we again multiply result a_1 by a new weight matrix and add a new bias value that is flowing between the hidden layer and the output layer. We can denote this weight matrix and bias as W_{hy} and b_y, respectively. The dimension of the weight matrix, W_{hy}, will be *the number of neurons in the hidden layer* x *the number of neurons in the output layer*. Since we have four neurons in the hidden layer and one neuron in the output layer, the W_{hy} matrix dimension will be 4 x 1. So, we multiply a_1 by the weight matrix, W_{hy}, and add bias, b_y, and pass the result z_2 to the next layer, which is the output layer:

$$z_2 = a_1W_{hy} + b_y$$

Now, in the output layer, we apply the sigmoid function to z_2, which will result in an output value:

$$\hat{y} = \sigma(z_2)$$

This whole process from the input layer to the output layer is known as forward propagation. Thus, in order to predict the output value, inputs are propagated from the input layer to the output layer. During this propagation, they are multiplied by their respective weights on each layer and an activation function is applied on top of them. The complete forward propagation steps are given as follows:

$$z_1 = XW_{xh} + b_h$$

$$a_1 = \sigma(z_1)$$

$$z_2 = a_1 W_{hy} + b_y$$

$$\hat{y} = \sigma(z_2)$$

The preceding forward propagation steps can be implemented in Python as follows:

```
def forward_prop(X):
    z1 = np.dot(X,Wxh) + bh
    a1 = sigmoid(z1)
    z2 = np.dot(a1,Why) + by
    y_hat = sigmoid(z2)
    return y_hat
```

Forward propagation is cool, isn't it? But how do we know whether the output generated by the neural network is correct? We define a new function called the cost function (J), also known as the loss function (L), which tells us how well our neural network is performing. There are many different cost functions. We will use the mean squared error as a cost function, which can be defined as the mean of the squared difference between the actual output and the predicted output:

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Here, n is the number of training samples, y is the actual output, and $\hat{y}$ is the predicted output.

Okay, so we learned that a cost function is used for assessing our neural network; that is, it tells us how good our neural network is at predicting the output. But the question is where is our network actually learning? In forward propagation, the network is just trying to predict the output. But how does it learn to predict the correct output? In the next section, we will examine this.

How does an ANN learn?

If the cost or loss is very high, then it means that our network is not predicting the correct output. So, our objective is to minimize the cost function so that our neural network predictions will be better. How can we minimize the cost function? That is, how can we minimize the loss/cost? We learned that the neural network makes predictions using forward propagation. So, if we can change some values in the forward propagation, we can predict the correct output and minimize the loss. But what values can we change in the forward propagation? Obviously, we can't change input and output. We are now left with weights and bias values. Remember that we just initialized weight matrices randomly. Since the weights are random, they are not going to be perfect. Now, we will update these weight matrices (W_{xh} and W_{hy}) in such a way that our neural network gives a correct output. How do we update these weight matrices? Here comes a new technique called gradient descent.

With gradient descent, the neural network learns the optimal values of the randomly initialized weight matrices. With the optimal values of weights, our network can predict the correct output and minimize the loss.

Now, we will explore how the optimal values of weights are learned using gradient descent. Gradient descent is one of the most commonly used optimization algorithms. It is used for minimizing the cost function, which allows us to minimize the error and obtain the lowest possible error value. But how does gradient descent find the optimal weights? Let's begin with an analogy.

Imagine we are on top of a hill, as shown in the following diagram, and we want to reach the lowest point on the hill. There could be many regions that look like the lowest points on the hill, but we have to reach the point that is actually the lowest of all.

That is, we should not be stuck at a point believing it is the lowest point when the global lowest point exists:

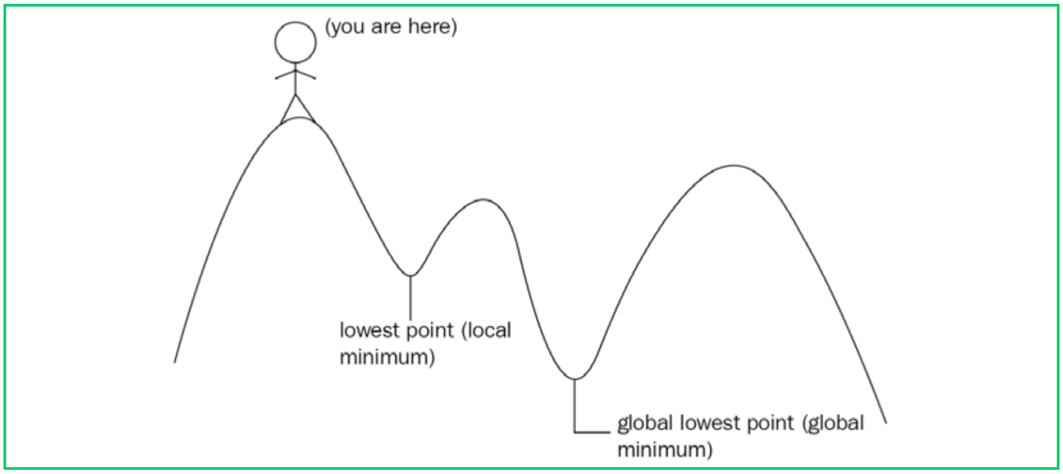

(you are here)
lowest point (local minimum)
global lowest point (global minimum)

Figure 20.9: Analogy of gradient descent

Similarly, we can represent our cost function as follows. It is a plot of cost against weights. Our objective is to minimize the cost function. That is, we have to reach the lowest point where the cost is the minimum. The solid dark point in the following diagram shows the randomly initialized weights. If we move this point downward, then we can reach the point where the cost is the minimum:

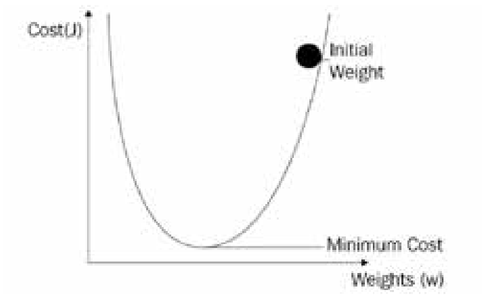


Figure 20.10: Gradient descent

But how can we move this point (initial weight) downward? How can we descend and reach the lowest point? Gradients are used for moving from one point to another. So, we can move this point (initial weight) by calculating a gradient of the cost function with respect to that point (initial weights), which is $\frac{\partial J}{\partial W}$.

Gradients are the derivatives that are actually the slope of a tangent line, as illustrated in the following diagram. So, by calculating the gradient, we descend (move downward) and reach the lowest point where the cost is the minimum. Gradient descent is a first-order optimization algorithm, which means we only take into account the first derivative when performing the updates:

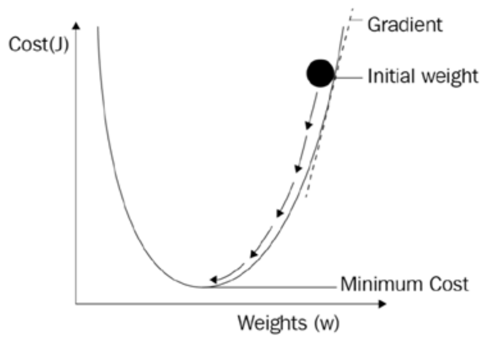


Figure 20.11: Gradient descent

Thus, with gradient descent, we move our weights to a position where the cost is minimum. But still, how do we update the weights?

As a result of forward propagation, we are in the output layer. We will now backpropagate the network from the output layer to the input layer and calculate the gradient of the cost function with respect to all the weights between the output and the input layer so that we can minimize the error. After calculating gradients, we update our old weights using the weight update rule:

$$W = W - \alpha \frac{\partial J}{\partial W}$$

This implies *weights* = *weights* -*a* x *gradients*.

What is α? It is called the learning rate. As shown in the following diagram, if the learning rate is small, then we take a small step downward and our gradient descent can be slow.

If the learning rate is large, then we take a large step and our gradient descent will be fast, but we might fail to reach the global minimum and become stuck at a local minimum. So, the learning rate should be chosen optimally:

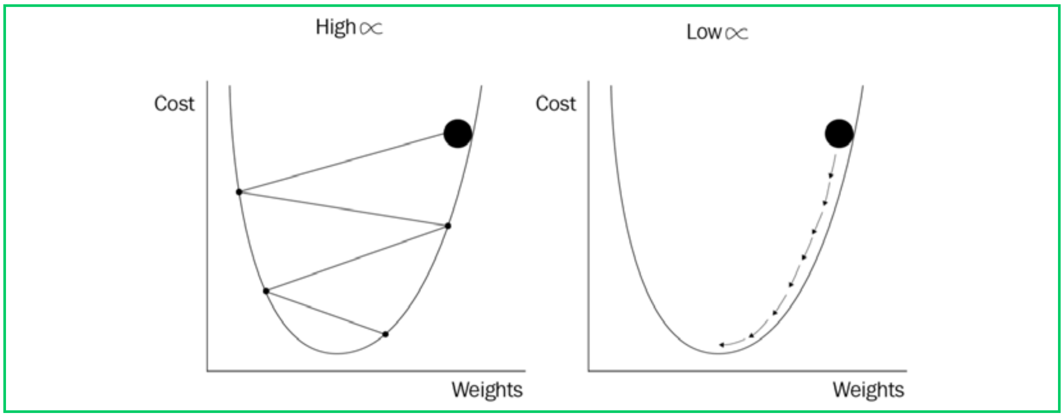


Figure 20.12: Effect of learning rate

This whole process of backpropagating the network from the output layer to the input layer and updating the weights of the network using gradient descent to minimize the loss is called backpropagation. Now that we have a basic understanding of backpropagation, we will strengthen our understanding by learning about this in detail, step by step. We are going to look at some interesting math, so put on your calculus hats and follow the steps.

So, we have two weights, one W_{xh}, which is the input to hidden layer weights, and the other W_{hy}, which is the hidden to output layer weights. We need to find the optimal values for these two weights that will give us the fewest errors. So, we need to calculate the derivative of the cost function J with respect to these weights. Since we are backpropagating, that is, going from the output layer to the input layer, our first weight will be W_{hy}. So, now we need to calculate the derivative of J with respect to W_{hy}. How do we calculate the derivative? First, let's recall our cost function, J:

$$J = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

We cannot calculate the derivative directly from the preceding equation since there is no W_{hy} term. So, instead of calculating the derivative directly, we calculate the partial derivative. Let's recall our forward propagation equation:

$$\hat{y} = \sigma(z_2)$$

$$z_2 = a_1 W_{hy} + b_y$$

First, we will calculate a partial derivative with respect to $\hat{y}$, and then from $\hat{y}$ we will calculate the partial derivative with respect to z_2. From z_2, we can directly calculate our derivative W_{hy}. It is basically the chain rule. So, the derivative of J with respect to W_{hy} becomes as follows:

$$\frac{\partial J}{\partial W_{hy}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{dz_2}{dW_{hy}} \qquad (1)$$

Now, we will compute each of the terms in the preceding equation:

$$\frac{\partial J}{\partial \hat{y}} = (y - \hat{y})$$

$$\frac{\partial \hat{y}}{\partial z_2} = \sigma'(z_2)$$

Here, σ' is the derivative of our sigmoid activation function. We know that the sigmoid function is $\sigma(z) = \frac{1}{1+e^{-z}}$, so the derivative of the sigmoid function would be $\sigma'(z) = \frac{e^z}{(1+e^{-z})^2}$.

Next we have:

$$\frac{dz_2}{dW_{hy}} = a_1$$

Thus, substituting all the preceding terms in equation *(1)* we can write:

$$\frac{\partial J}{dW_{hy}} = (y - \hat{y}).\, \sigma'(z_2).\, a_1 \qquad (2)$$

Now we need to compute a derivative of J with respect to our next weight, W_{xh}.

Similarly, we cannot calculate the derivative of W_{xh} directly from J as we don't have any W_{xh} terms in J. So, we need to use the chain rule. Let's recall the forward propagation steps again:

$$\hat{y} = \sigma(z_2)$$

$$z_2 = a_1 W_{hy} + b_y$$

$$a_1 = \sigma(z_1)$$

$$z_1 = XW_{xh} + b_h$$

Now, according to the chain rule, the derivative of J with respect to W_{xh} is given as:

$$\frac{\partial J}{\partial W_{xh}} = \frac{\partial J}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_2} \cdot \frac{dz_2}{\partial a_1} \cdot \frac{\partial a_1}{\partial z_1} \cdot \frac{dz_1}{dW_{xh}} \qquad (3)$$

We have already seen how to compute the first two terms in the preceding equation; now, we will see how to compute the rest of the terms:

$$\frac{dz_2}{\partial a_1} = W_{hy}$$

$$\frac{\partial a_1}{\partial z_1} = \sigma'(z_1)$$

$$\frac{dz_1}{dW_{xh}} = X$$

Thus, substituting all the preceding terms in equation *(3)*, we can write:

$$\frac{\partial J}{\partial W_{xh}} = (y - \hat{y}).\, \sigma'(z_2).\, W_{hy} \,.\, \sigma'(z_1) \,.\, x \qquad (4)$$

After we have computed gradients for both weights, W_{hy} and W_{xh}, we will update our initial weights according to the weight update rule:

$$W_{hy} = W_{hy} - \alpha \frac{\partial J}{\partial W_{hy}} \qquad (5)$$

$$W_{xy} = W_{xy} - \alpha \frac{\partial J}{\partial W_{xh}} \qquad (6)$$

That's it! This is how we update the weights of a network and minimize the loss. Now, let's see how to implement the backpropagation algorithm in Python.

In both the equations *(2)* and *(4)*, we have the term $(y - \hat{y}).\sigma'(z_2)$, so instead of computing them again and again, we just call them `delta2`:

```
delta2 = np.multiply(-(y-yHat),sigmoidPrime(z2))
```

Now, we compute the gradient with respect to W_{hy}. Refer to equation *(2)*:

```
dJ_dWhy = np.dot(a1.T,delta2)
```

We compute the gradient with respect to W_{xh}. Refer to equation *(4)*:

```
delta1 = np.dot(delta2,Why.T)*sigmoidPrime(z1)
dJ_dWxh = np.dot(X.T,delta1)
```

We will update the weights according to our weight update rule equation *(5)* and *(6)* as follows:

```
Wxh = Wxh - alpha * dJ_dWhy
Why = Why - alpha * dJ_dWxh
```

The complete code for the backpropagation is given as follows:

```
def backword_prop(y_hat, z1, a1, z2):
    delta2 = np.multiply(-(y-y_hat),sigmoid_derivative(z2))
    dJ_dWhy = np.dot(a1.T, delta2)
    delta1 = np.dot(delta2,Why.T)*sigmoid_derivative(z1)
    dJ_dWxh = np.dot(X.T, delta1)
    Wxh = Wxh - alpha * dJ_dWhy
    Why = Why - alpha * dJ_dWxh
    return Wxh,Why
```

That's it. Apart from this, there are different variants of gradient descent methods such as stochastic gradient descent, mini-batch gradient descent, Adam, RMSprop, and more.

Before moving on, let's familiarize ourselves with some of the frequently used terminology in neural networks:

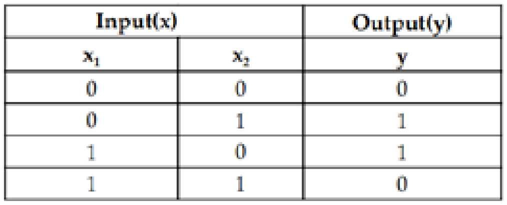
- **Forward pass**: Forward pass implies forward propagating from the input layer to the output layer.

- **Backward pass**: Backward pass implies backpropagating from the output layer to the input layer.
- **Epoch**: The epoch specifies the number of times the neural network sees our whole training data. So, we can say one epoch is equal to one forward pass and one backward pass for all training samples.
- **Batch size**: The batch size specifies the number of training samples we use in one forward pass and one backward pass.
- **Number of iterations**: The number of iterations implies the number of passes where *one pass* = *one forward pass* + *one backward pass*.

Say that we have 12,000 training samples and our batch size is 6,000. Then it will take us two iterations to complete one epoch. That is, in the first iteration, we pass the first 6,000 samples and perform a forward pass and a backward pass; in the second iteration, we pass the next 6,000 samples and perform a forward pass and a backward pass. After two iterations, our neural network will see the whole 12,000 training samples, which makes it one epoch.

Putting it all together

Putting all the concepts we have learned so far together, we will see how to build a neural network from scratch. We will understand how the neural network learns to perform the XOR gate operation. The XOR gate returns 1 only when exactly only one of its inputs is 1, else it returns 0, as shown in *Table 20.1*:



Input(x)		Output(y)
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Table 20.1: XOR operation

Building a neural network from scratch

To perform the XOR gate operation, we build a simple two-layer neural network, as shown in the following diagram. As you can see, we have an input layer with two nodes, a hidden layer with five nodes and an output layer comprising one node:

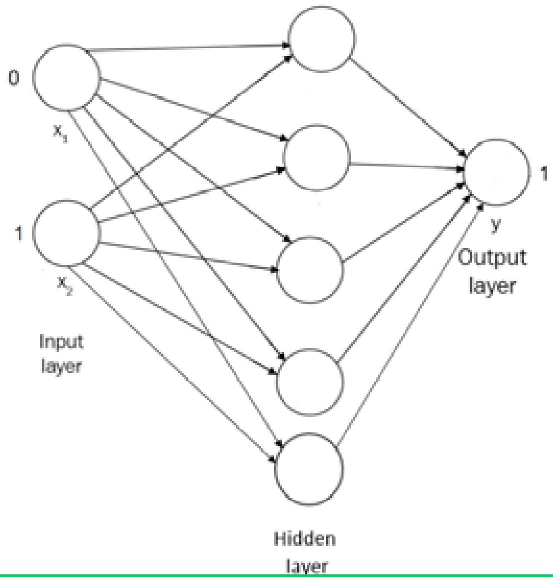


Figure 20.13: ANN

We will understand step-by-step how a neural network learns the XOR logic:

1. First, import the libraries:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
```

2. Prepare the data as shown in the preceding XOR table:

```
X = np.array([ [0, 1], [1, 0], [1, 1],[0, 0] ])
y = np.array([ [1], [1], [0], [0]])
```

3. Define the number of nodes in each layer:

```
num_input = 2
num_hidden = 5
num_output = 1
```

4. Initialize the weights and bias randomly. First, we initialize the input to hidden layer weights:

```
Wxh = np.random.randn(num_input,num_hidden)
bh = np.zeros((1,num_hidden))
```

5. Now, we initialize the hidden to output layer weights:

```
Why = np.random.randn (num_hidden,num_output)
by = np.zeros((1,num_output))
```

6. Define the sigmoid activation function:

```
def sigmoid(z):
    return 1 / (1+np.exp(-z))
```

7. Define the derivative of the sigmoid function:

```
def sigmoid_derivative(z):
     return np.exp(-z)/((1+np.exp(-z))**2)
```

8. Define the forward propagation:

```
def forward_prop(x,Wxh,Why):
    z1 = np.dot(x,Wxh) + bh
    a1 = sigmoid(z1)
    z2 = np.dot(a1,Why) + by
    y_hat = sigmoid(z2)
    return z1,a1,z2,y_hat
```

9. Define the backward propagation:

```
def backword_prop(y_hat, z1, a1, z2):
    delta2 = np.multiply(-(y-y_hat),sigmoid_derivative(z2))
    dJ_dWhy = np.dot(a1.T, delta2)
    delta1 = np.dot(delta2,Why.T)*sigmoid_derivative(z1)
    dJ_dWxh = np.dot(x.T, delta1)
    return dJ_dWxh, dJ_dWhy
```

10. Define the cost function:

```
def cost_function(y, y_hat):
    J = 0.5*sum((y-y_hat)**2)
    return J
```

11. Set the learning rate and the number of training iterations:

```
alpha = 0.01
num_iterations = 5000
```

12. Now, let's start training the network with the following code:

```
cost =[]
for i in range(num_iterations):
    z1,a1,z2,y_hat = forward_prop(X,Wxh,Why)
    dJ_dWxh, dJ_dWhy = backword_prop(y_hat, z1, a1, z2)
    #update weights
    Wxh = Wxh -alpha * dJ_dWxh
    Why = Why -alpha * dJ_dWhy
    #compute cost
    c = cost_function(y, y_hat)
        cost.append(c)
```

13. Plot the cost function:

```
plt.grid()
plt.plot(range(num_iterations),cost)
plt.title('Cost Function')
plt.xlabel('Training Iterations')
plt.ylabel('Cost')
```

As you can observe in the following plot, the loss decreases over the training iterations:

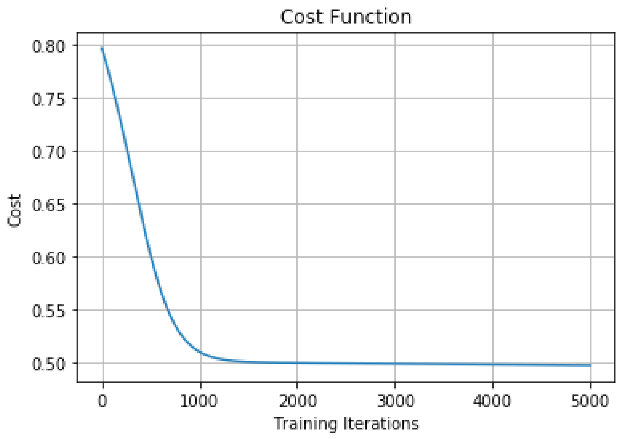


Figure 20.14: Cost function

Thus, we have an overall understanding of ANNs and how they learn.

Recurrent Neural Networks

The sun rises in the ____.

If we were asked to predict the blank term in the preceding sentence, we would probably say east. Why would we predict that the word east would be the right word here? Because we read the whole sentence, understood the context, and predicted that the word east would be an appropriate word to complete the sentence.

If we use a feedforward neural network (the one we learned in the previous section) to predict the blank, it would not predict the right word. This is due to the fact that in feedforward networks, each input is independent of other input and they make predictions based only on the current input, and they don't remember previous input.

Thus, the input to the network will just be the word preceding the blank, which is the word *the*. With this word alone as an input, our network cannot predict the correct word, because it doesn't know the context of the sentence, which means that it doesn't know the previous set of words to understand the context of the sentence and to predict an appropriate next word.

Here is where we use Recurrent Neural Networks (RNNs). They predict output not only based on the current input, but also on the previous hidden state. Why do they have to predict the output based on the current input and the previous hidden state? Why can't they just use the current input and the previous input?

This is because the previous input will only store information about the previous word, while the previous hidden state will capture the contextual information about all the words in the sentence that the network has seen so far. Basically, the previous hidden state acts like memory, and it captures the context of the sentence. With this context and the current input, we can predict the relevant word.

For instance, let's take the same sentence, *The sun rises in the ____*. As shown in the following figure, we first pass the word *the* as an input, and then we pass the next word, *sun*, as input; but along with this, we also pass the previous hidden state, h_0. So, every time we pass the input word, we also pass a previous hidden state as an input.

In the final step, we pass the word *the*, and also the previous hidden state h_3, which captures the contextual information about the sequence of words that the network has seen so far. Thus, h_3 acts as the memory and stores information about all the previous words that the network has seen. With h_3 and the current input word (*the*), we can predict the relevant next word:

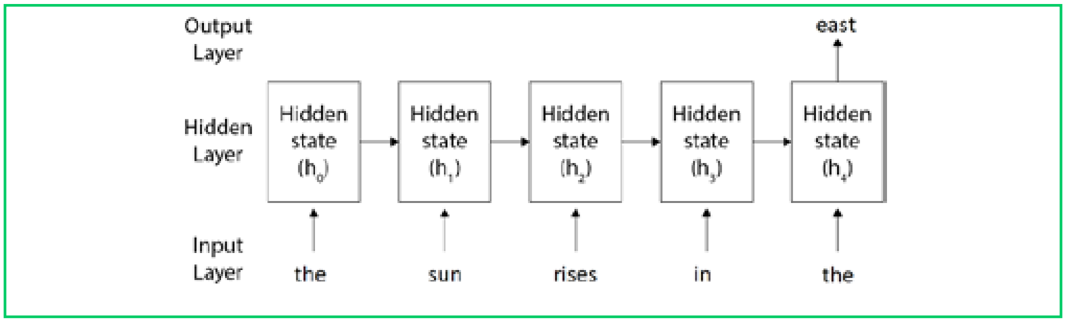


Figure 20.15: RNN

In a nutshell, an RNN uses the previous hidden state as memory, which captures and stores the contextual information (input) that the network has seen so far.

RNNs are widely applied for use cases that involve sequential data, such as time series, text, audio, speech, video, weather, and much more. They have been greatly used in various natural language processing (NLP) tasks, such as language translation, sentiment analysis, text generation, and so on.

The difference between feedforward networks and RNNs

A comparison between an RNN and a feedforward network is shown in the *Figure 20.16*:

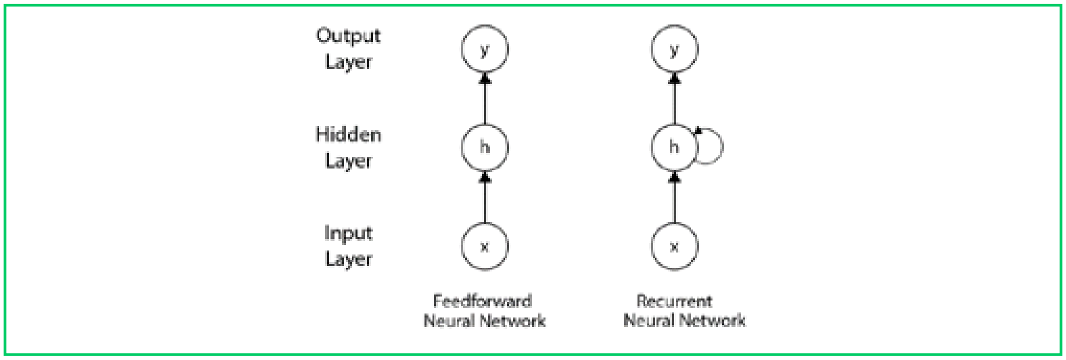


Figure 20.16: Difference between feedforward network and RNN

As you can observe in the preceding diagram, the RNN contains a looped connection in the hidden layer, which implies that we use the previous hidden state along with the input to predict the output.

Still confused? Let's look at the following unrolled version of an RNN. But wait; what is the unrolled version of an RNN?

It means that we roll out the network for a complete sequence. Let's suppose that we have an input sentence with T words; then, we will have 0 to $T—1$ layers, one for each word, as shown in *Figure 20.17*:

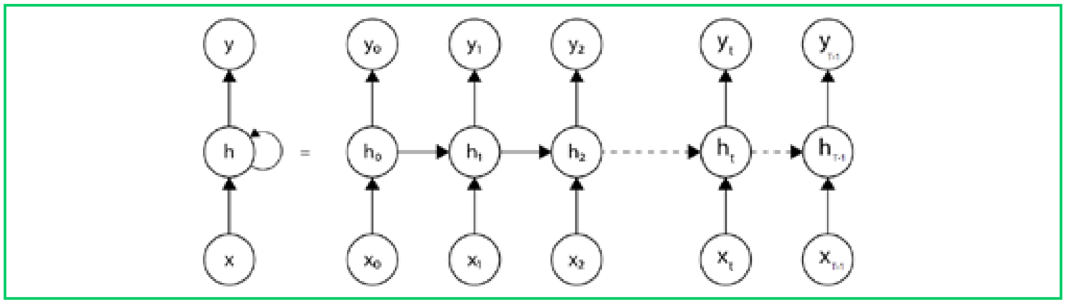


Figure 20.17: Unrolled RNN

As you can see in *Figure 20.17*, at the time step $t = 1$, the output y_1 is predicted based on the current input x_1 and the previous hidden state h_0. Similarly, at time step $t = 2$, y_2 is predicted using the current input x_2 and the previous hidden state h_1. This is how an RNN works; it takes the current input and the previous hidden state to predict the output.

Forward propagation in RNNs

Let's look at how an RNN uses forward propagation to predict the output; but before we jump right in, let's get familiar with the notations:

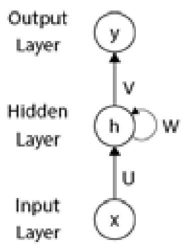


Figure 20.18: Forward propagation in RNN

The preceding figure illustrates the following:

- U represents the input to hidden layer weight matrix
- W represents the hidden to hidden layer weight matrix
- V represents the hidden to output layer weight matrix

The hidden state h at a time step t can be computed as follows:

$$h_t = \tanh\,(Ux_t + Wh_{t-1})$$

That is, *the hidden state at a time step, t = tanh([input to hidden layer weight x input] + [hidden to hidden layer weight x previous hidden state])*.

The output at a time step t can be computed as follows:

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

That is, *the output at a time step, t = softmax (hidden to output layer weight* x *hidden state at a time t)*.

We can also represent RNNs as shown in the following figure. As you can see, the hidden layer is represented by an RNN block, which implies that our network is an RNN, and previous hidden states are used in predicting the output:

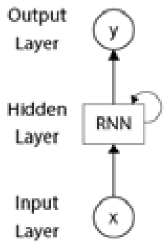


Figure 20.19: Forward propagation in an RNN

Figure 20.20 shows how forward propagation works in an unrolled version of an RNN:

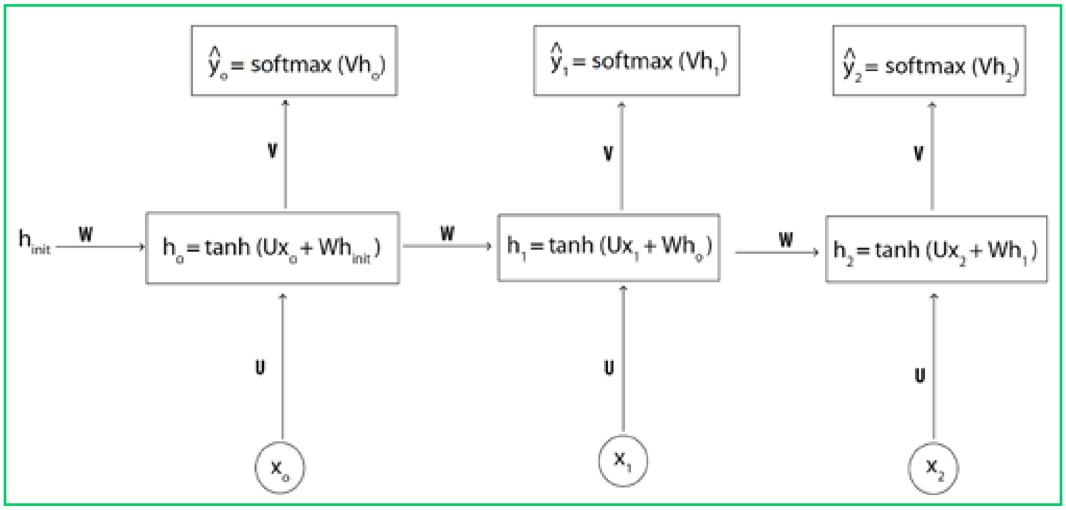

$\hat{y}_0$ = softmax (Vh_0)
$\hat{y}_1$ = softmax (Vh_1)
$\hat{y}_2$ = softmax (Vh_2)
V
V
V
h_{init}
W
h_0 = tanh (Ux_0 + Wh_{init})
W
h_1 = tanh (Ux_1 + Wh_0)
W
h_2 = tanh (Ux_2 + Wh_1)
U
U
U
x_0
x_1
x_2

Figure 20.20: Unrolled version – forward propagation in an RNN

We initialize the initial hidden state h_{init} with random values. As you can see in the preceding figure, the output, $\hat{y}_0$, is predicted based on the current input, x_0 and the previous hidden state, which is an initial hidden state, h_{init}, using the following formula:

$$h_0 = \tanh\,(Ux_0 + Wh_{init})$$

$$\hat{y}_0 = \mathrm{softmax}(Vh_0)$$

Similarly, look at how the output, $\hat{y}_1$, is computed. It takes the current input, x_1, and the previous hidden state, h_0:

$$h_1 = \tanh\,(Ux_1 + Wh_0)$$

$$\hat{y}_1 = \mathrm{softmax}(Vh_1)$$

Thus, in forward propagation to predict the output, RNN uses the current input and the previous hidden state.

Backpropagating through time

We just learned how forward propagation works in RNNs and how it predicts the output. Now, we compute the loss, L, at each time step, t, to determine how well the RNN has predicted the output. We use the cross-entropy loss as our loss function. The loss L at a time step t can be given as follows:

$$L_t = -y_t \log(\hat{y}_t)$$

Here, y_t is the actual output, and $\hat{y}_t$ is the predicted output at a time step t.

The final loss is a sum of the loss at all the time steps. Suppose that we have T - 1 layers; then, the final loss can be given as follows:

$$L = \sum_{j=0}^{T-1} L_j$$

Figure 20.21 shows that the final loss is obtained by the sum of loss at all the time steps:

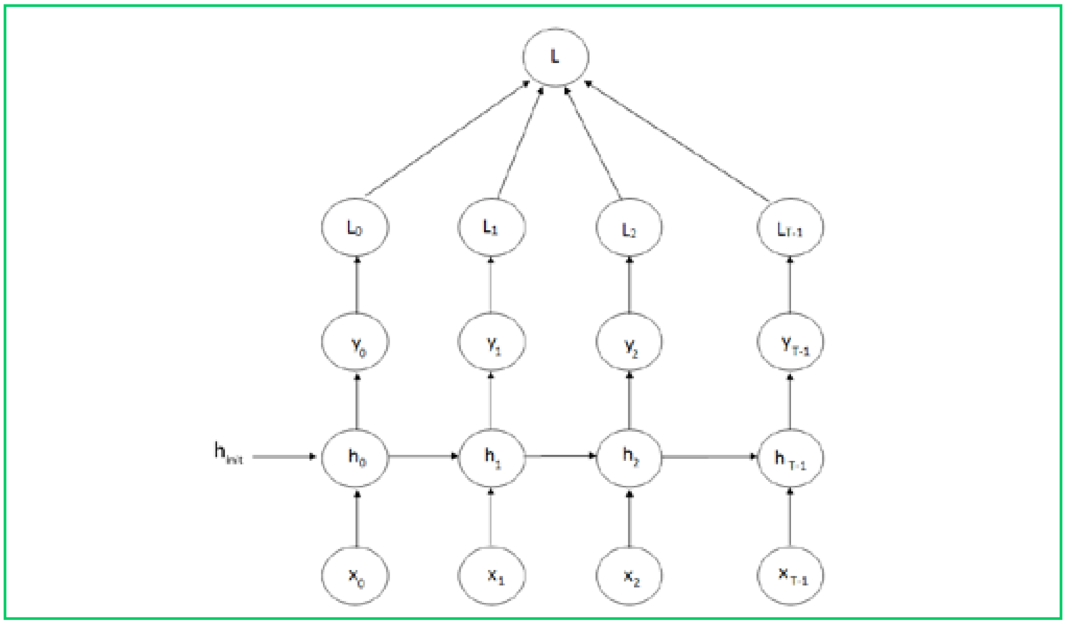


Figure 20.21: Backpropagation in an RNN

We computed the loss, now our goal is to minimize the loss. How can we minimize the loss? We can minimize the loss by finding the optimal weights of the RNN. As we learned, we have three weights in RNNs: input to hidden, U, hidden to hidden, W, and hidden to output, V.

We need to find optimal values for all of these three weights to minimize the loss. We can use our favorite gradient descent algorithm to find the optimal weights. We begin by calculating the gradients of the loss function with respect to all the weights; then, we update the weights according to the weight update rule as follows:

$$V = V - \alpha \frac{\partial L}{\partial V}$$

$$W = W - \alpha \frac{\partial L}{\partial W}$$

$$U = U - \alpha \frac{\partial L}{\partial U}$$

However, we have a problem with the RNN. The gradient calculation involves calculating the gradient with respect to the activation function. When we calculate the gradient with respect to the sigmoid or tanh function, the gradient will become very small. When we further backpropagate the network over many time steps and multiply the gradients, the gradients will tend to get smaller and smaller. This is called a vanishing gradient problem.

Since the gradient vanishes over time, we cannot learn information about long-term dependencies, that is, RNNs cannot retain information for a long time in the memory. The vanishing gradient problem occurs not only in RNNs but also in other deep networks where we have many hidden layers and when we use sigmoid/tanh functions.

One solution to avoid vanishing gradient problem is to use ReLU as an activation function. However, we have a variant of the RNN called Long Short-Term Memory (LSTM), which can solve the vanishing gradient problem effectively. We will see how it works in the upcoming section.

LSTM to the rescue

While backpropagating an RNN, we learned about a problem called vanishing gradients. Due to the vanishing gradient problem, we cannot train the network properly, and this causes the RNN to not retain long sequences in the memory. To understand what we mean by this, let's consider a small sentence:

The sky is __.

An RNN can easily predict the blank as *blue* based on the information it has seen, but it cannot cover the long-term dependencies. What does that mean? Let's consider the following sentence to understand the problem better:

Archie lived in China for 13 years. He loves listening to good music. He is a fan of comics. He is fluent in ____.

Now, if we were asked to predict the missing word in the preceding sentence, we would predict it as *Chinese*, but how did we predict that? We simply remembered the previous sentences and understood that Archie lived for 13 years in China. This led us to the conclusion that Archie might be fluent in Chinese. An RNN, on the other hand, cannot retain all of this information in its memory to say that Archie is fluent in Chinese.

Due to the vanishing gradient problem, it cannot recollect/remember information for a long time in its memory. That is, when the input sequence is long, the RNN memory (hidden state) cannot hold all the information. To alleviate this, we use an LSTM cell.

LSTM is a variant of an RNN that resolves the vanishing gradient problem and retains information in the memory as long as it is required. Basically, RNN cells are replaced with LSTM cells in the hidden units, as shown in *Figure 20.22*:

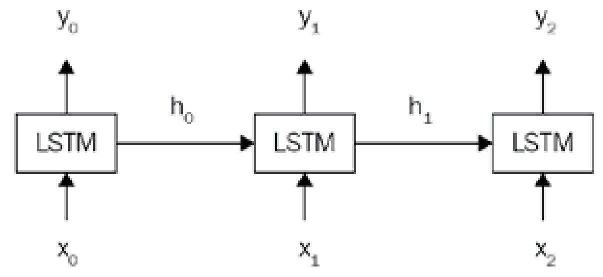


Figure 20.22: LSTM network

In the next section, we will understand how the LSTM cells works.

Understanding the LSTM cell

What makes LSTM cells so special? How do LSTM cells achieve long-term dependency? How does it know what information to keep and what information to discard from the memory?

This is all achieved by special structures called gates. As shown in the following diagram, a typical LSTM cell consists of three special gates called the input gate, output gate, and forget gate:

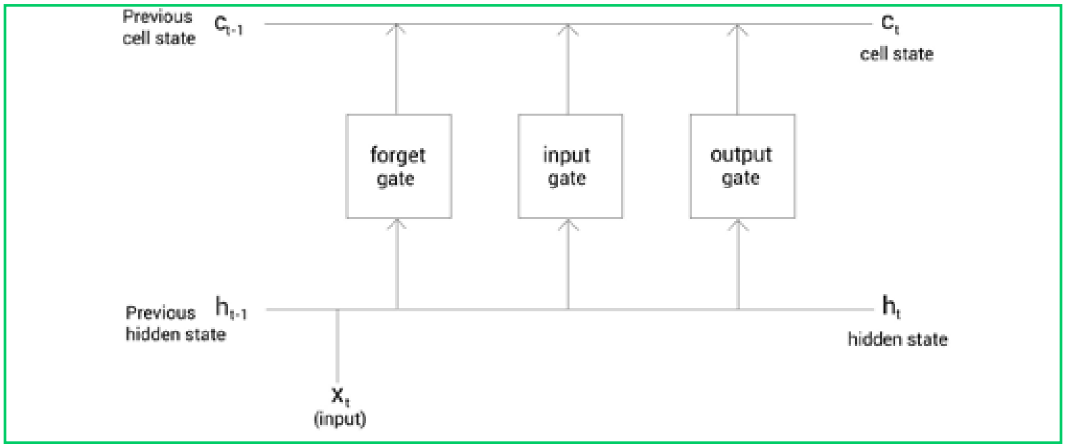


Figure 20.23: LSTM gates

These three gates are responsible for deciding what information to add, output, and forget from the memory. With these gates, an LSTM cell effectively keeps information in the memory only as long as required. *Figure 20.24* shows a typical LSTM cell:

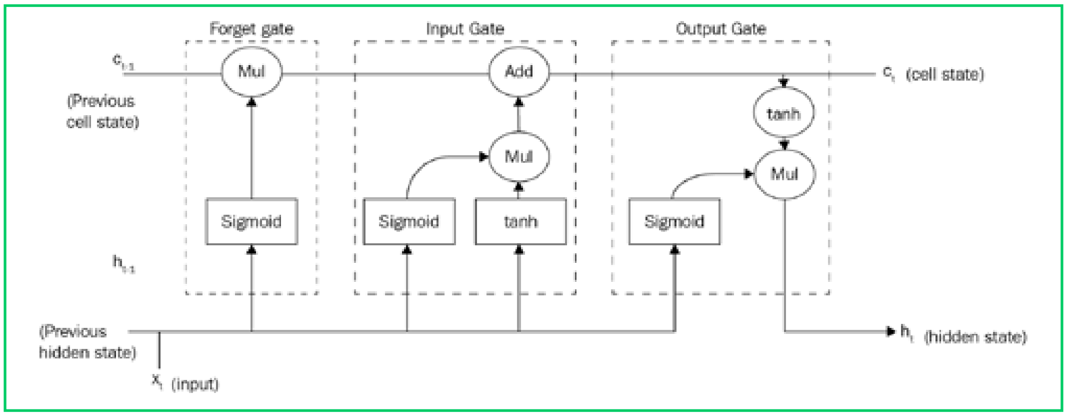


Figure 20.24: LSTM cell

If you look at the LSTM cell, the top horizontal line is called the cell state. It is where the information flows. Information on the cell state will be constantly updated by LSTM gates. Now, we will see the function of these gates:

Forget gate: The forget gate is responsible for deciding what information should not be in the cell state. Look at the following statement:

Harry is a good singer. He lives in New York. Zayn is also a good singer.

As soon as we start talking about Zayn, the network will understand that the subject has been changed from Harry to Zayn and the information about Harry is no longer required. Now, the forget gate will remove/forget information about Harry from the cell state.

Input gate: The input gate is responsible for deciding what information should be stored in the memory. Let's consider the same example:

Harry is a good singer. He lives in New York. Zayn is also a good singer.

So, after the forget gate removes information from the cell state, the input gate decides what information has to be in the memory. Here, since the information about Harry is removed from the cell state by the forget gate, the input gate decides to update the cell state with the information about Zayn.

Output gate: The output gate is responsible for deciding what information should be shown from the cell state at a time, *t*. Now, consider the following sentence:

Zayn's debut album was a huge success. Congrats ____.

Here, congrats is an adjective which is used to describe a noun. The output layer will predict Zayn (noun), to fill in the blank.

Thus, using LSTM, we can overcome the vanishing gradient problem faced in RNN. In the next section, we will learn another interesting algorithm called the Convolutional Neural Network (CNN).

What are CNNs?

A CNN, also known as a ConvNet, is one of the most widely used deep learning algorithms for computer vision tasks. Let's say we are performing an image-recognition task. Consider the following image.

We want our CNN to recognize that it contains a horse:

Figure 20.25: Image containing a horse

How can we do that? When we feed the image to a computer, it basically converts it into a matrix of pixel values. The pixel values range from 0 to 255, and the dimensions of this matrix will be of [*image width* x *image height* x *number of channels*]. A grayscale image has one channel, and colored images have three channels, red, green, and blue (RGB).

Let's say we have a colored input image with a width of 11 and a height of 11, that is 11 x 11, then our matrix dimension would be *[11 x 11 x 3]*. As you can see in *[11 x 11 x 3]*, 11 x 11 represents the image width and height and 3 represents the channel number, as we have a colored image. So, we will have a 3D matrix.

But it is hard to visualize a 3D matrix, so, for the sake of understanding, let's consider a grayscale image as our input. Since the grayscale image has only one channel, we will get a 2D matrix.

As shown in the following diagram, the input grayscale image will be converted into a matrix of pixel values ranging from 0 to 255, with the pixel values representing the intensity of pixels at that point:

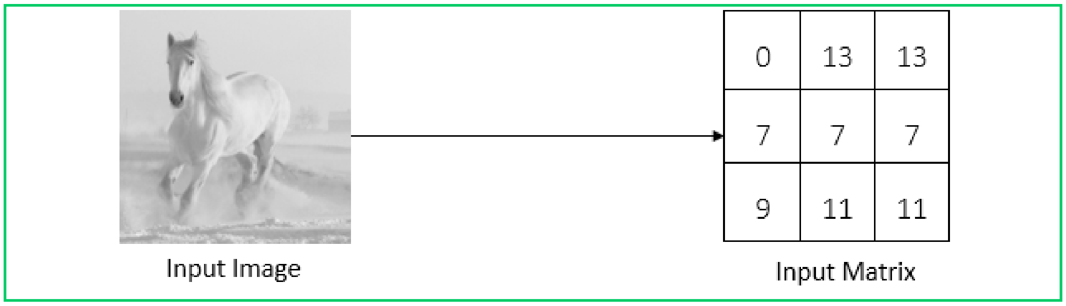


Figure 20.26: Input image is converted to matrix of pixel values

The values given in the input matrix are just arbitrary values for our understanding.

Okay, now we have an input matrix of pixel values. What happens next? How does the CNN come to understand that the image contains a horse? CNNs consists of the following three important layers:

- The convolutional layer
- The pooling layer
- The fully connected layer

With the help of these three layers, the CNN recognizes that the image contains a horse. Now we will explore each of these layers in detail.

Convolutional layers

The convolutional layer is the first and core layer of the CNN. It is one of the building blocks of a CNN and is used for extracting important features from the image.

We have an image of a horse. What do you think are the features that will help us to understand that this is an image of a horse? We can say body structure, face, legs, tail, and so on. But how does the CNN understand these features? This is where we use a convolution operation that will extract all the important features from the image that characterize the horse. So, the convolution operation helps us to understand what the image is all about.

Okay, what exactly is this convolution operation? How it is performed? How does it extract the important features? Let's look at this in detail.

As we know, every input image is represented by a matrix of pixel values. Apart from the input matrix, we also have another matrix called the filter matrix.

The filter matrix is also known as a kernel, or simply a filter, as shown in the *Figure 20.27*:

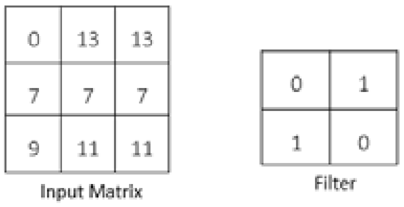


Figure 20.27: Input and filter matrix

We take the filter matrix, slide it over the input matrix by one pixel, perform element-wise multiplication, sum the results, and produce a single number. That's pretty confusing, isn't it? Let's understand this better with the aid of the following diagram:

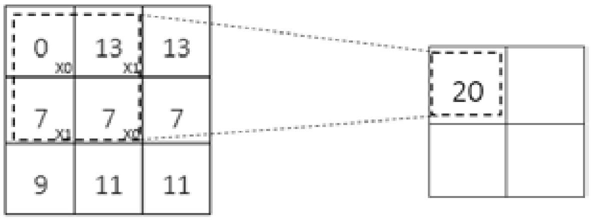


Figure 20.28: Convolution operation

As you can see in the previous diagram, we took the filter matrix and placed it on top of the input matrix, performed element-wise multiplication, summed their results, and produced the single number. This is demonstrated as follows:

$$(0 * 0) + (13 * 1) + (7 * 1) + (7 * 0) = 20$$

Now, we slide the filter over the input matrix by one pixel and perform the same steps, as shown in *Figure 20.29*:

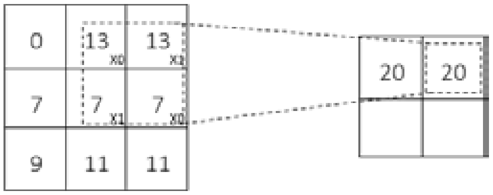


Figure 20.29: Convolution operation

This is demonstrated as follows:

$$(13 * 0) + (13 * 1) + (7 * 1) + (7 * 0) = 20$$

Again, we slide the filter matrix by one pixel and perform the same operation, as shown in *Figure 20.30*:

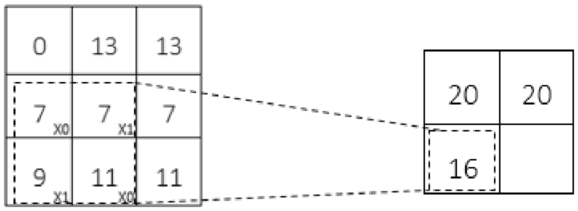


Figure 20.30: Convolution operation

This is demonstrated as follows:

$$(7 * 0) + (7 * 1) + (9 * 1) + (11 * 0) = 16$$

Now, again, we slide the filter matrix over the input matrix by one pixel and perform the same operation, as shown in *Figure 20.31*:

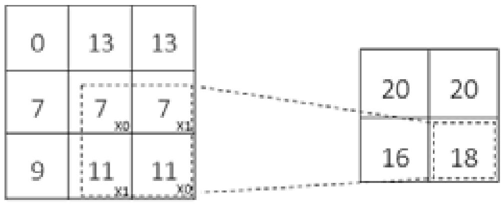


Figure 20.31: Convolution operation

That is:

$$(7 * 0) + (7 * 1) + (11 * 1) + (11 * 0) = 18$$

Okay. What are we doing here? We are basically sliding the filter matrix over the entire input matrix by one pixel, performing element-wise multiplication and summing their results, which creates a new matrix called a feature map or activation map. This is called the convolution operation.

As we've learned, the convolution operation is used to extract features, and the new matrix, that is, the feature maps, represents the extracted features. If we plot the feature maps, then we can see the features extracted by the convolution operation.

Figure 20.32 shows the actual image (the input image) and the convolved image (the feature map). We can see that our filter has detected the edges from the actual image as a feature:

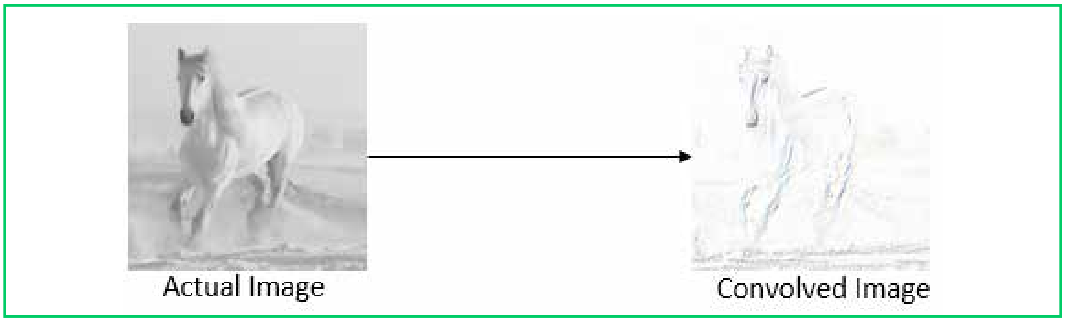


Figure 20.32: Conversion of actual image to convolved image

Various filters are used for extracting different features from the image. For instance, if we use a sharpen filter, $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & 1 & 0 \end{bmatrix}$, then it will sharpen our image, as shown in the following figure:

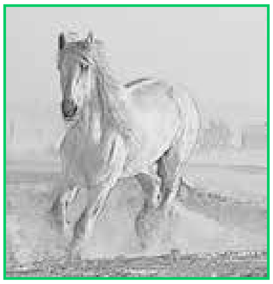


Figure 20.33: Sharpened image

Thus, we have learned that with filters, we can extract important features from the image using the convolution operation. So, instead of using one filter, we can use multiple filters to extract different features from the image and produce multiple feature maps. So, the depth of the feature map will be the number of filters. If we use seven filters to extract different features from the image, then the depth of our feature map will be seven:

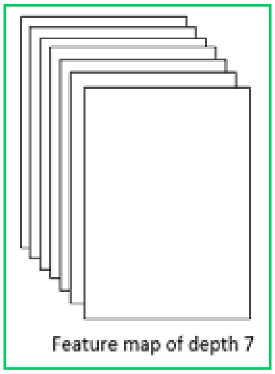


Figure 20.34: Feature maps

Okay, we have learned that different filters extract different features from the image. But the question is, how can we set the correct values for the filter matrix so that we can extract the important features from the image? Worry not! We just initialize the filter matrix randomly, and the optimal values of the filter matrix, with which we can extract the important features from the images, will be learned through backpropagation. However, we just need to specify the size of the filter and the number of filters we want to use.

Strides

We have just learned how a convolution operation works. We slide over the input matrix with the filter matrix by one pixel and perform the convolution operation. But we don't have to only slide over the input matrix by one pixel, we can also slide over the input matrix by any number of pixels.

The number of pixels we slide over the input matrix by the filter matrix is called a stride.

If we set the stride to 2, then we slide over the input matrix with the filter matrix by two pixels. *Figure 20.35* shows a convolution operation with a stride of 2:

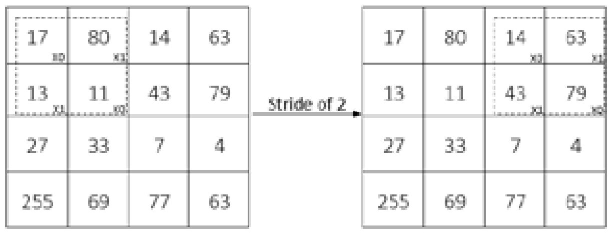


Figure 20.35: Stride operation

But how do we choose the stride number? We just learned that a stride is the number of pixels along that we move our filter matrix. So, when the stride is set to a small number, we can encode a more detailed representation of the image than when the stride is set to a large number. However, a stride with a high value takes less time to compute than one with a low value.

Padding

With the convolution operation, we are sliding over the input matrix with a filter matrix. But in some cases, the filter does not perfectly fit the input matrix. What do we mean by that? For example, let's say we are performing a convolution operation with a stride of 2. There exists a situation where, when we move our filter matrix by two pixels, it reaches the border and the filter matrix does not fit the input matrix. That is, some part of our filter matrix is outside the input matrix, as shown in the following diagram:

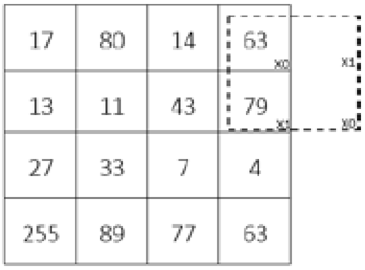


Figure 20.36: Padding operation

In this case, we perform padding. We can simply pad the input matrix with zeros so that the filter can fit the input matrix, as shown in *Figure 20.37*. Padding with zeros on the input matrix is called same padding or zero padding:

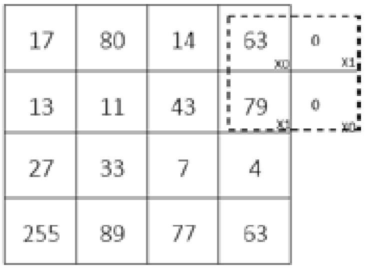


Figure 20.37: Same padding

Instead of padding them with zeros, we can also simply discard the region of the input matrix where the filter doesn't fit in. This is called valid padding:

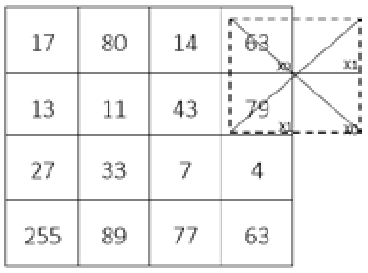


Figure 20.38: Valid padding

Pooling layers

Okay. Now, we are done with the convolution operation. As a result of the convolution operation, we have some feature maps. But the feature maps are too large in dimension. In order to reduce the dimensions of feature maps, we perform a pooling operation. This reduces the dimensions of the feature maps and keeps only the necessary details so that the amount of computation can be reduced.

For example, to recognize a horse from the image, we need to extract and keep only the features of the horse; we can simply discard unwanted features, such as the background of the image and more. A pooling operation is also called a downsampling or subsampling operation, and it makes the CNN translation invariant. Thus, the pooling layer reduces spatial dimensions by keeping only the important features.

There are different types of pooling operations, including max pooling, average pooling, and sum pooling.

In max pooling, we slide over the filter on the input matrix and simply take the maximum value from the filter window, as *Figure 20.39* shows:

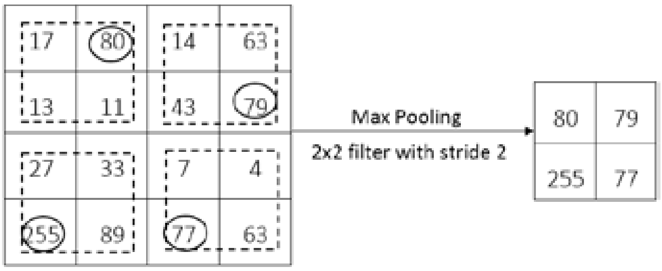


Figure 20.39: Max pooling

In average pooling, we take the average value of the input matrix within the filter window, and in sum pooling, we sum all the values of the input matrix within the filter window.

Fully connected layers

So far, we've learned how convolutional and pooling layers work. A CNN can have multiple convolutional layers and pooling layers. However, these layers will only extract features from the input image and produce the feature map; that is, they are just the feature extractors.

Given any image, convolutional layers extract features from the image and produce a feature map. Now, we need to classify these extracted features. So, we need an algorithm that can classify these extracted features and tell us whether the extracted features are the features of a horse, or something else. In order to make this classification, we use a feedforward neural network. We flatten the feature map and convert it into a vector, and feed it as an input to the feedforward network.

The feedforward network takes this flattened feature map as an input, applies an activation function, such as sigmoid, and returns the output, stating whether the image contains a horse or not; this is called a fully connected layer and is shown in the following diagram:

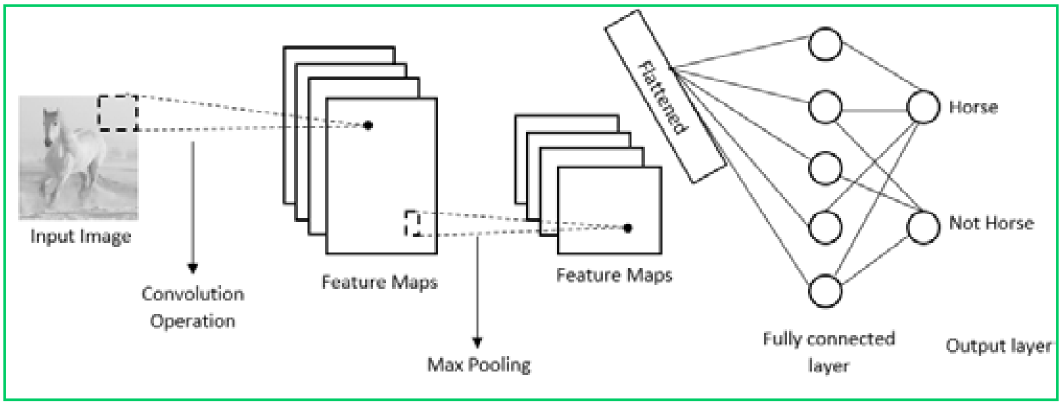


Figure 20.40: Fully connected layer

Let's see how all this fits together.