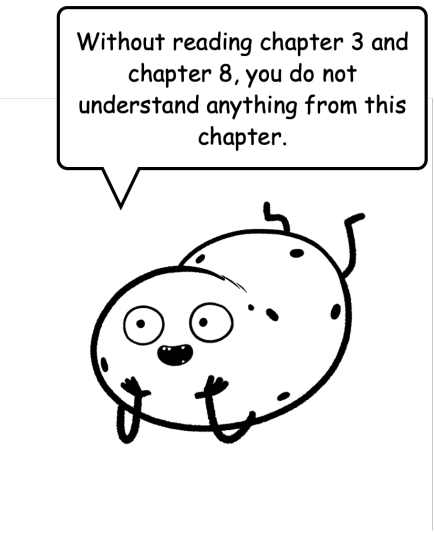
Chapter 10: Neural Networks and Deep Learning

Due to the enormous success of deep learning, you might not read the book from the beginning and start it from here. That is not good, but it should not be a big problem. You should listen to the Good News Potato feedback before you start reading a chapter, and be sure that you have read all required chapters before starting this chapter.

Deep learning has revolutionized many scientific disciplines. People whose research works sound funny and too theoretical before 2012 are now celebrities in academia, not just in computer science but also in other disciplines.



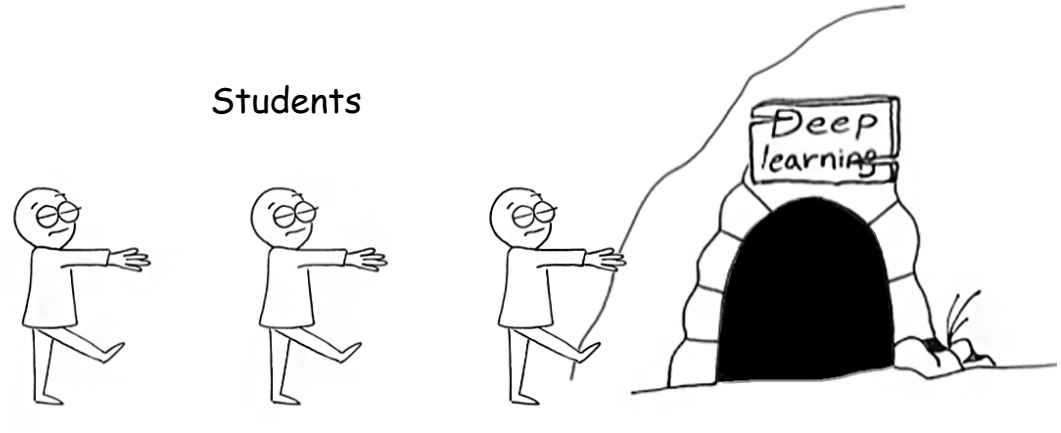
In this chapter, we start by explaining the rationale behind deep learning and then the function of a biological brain. It helps us to explain artificial neural networks more easily later. Next, we move our discussion from the traditional ANN method, including Perceptron and Multilayer Perceptron. Then, we explain activation functions, cost functions, and neural network optimizers. Afterward, two common deep learning models and architectures, including CNN and RNN, are explained.

At the time of writing this Chapter (late 2020 and revising in 2024), we are living in the AI boom, and one of the hardest jobs of academic supervisors is to lead motivated students and research workers from focusing too much on neural networks and deep learning, dedicate their time to learning other concepts and algorithms as well. There are many other important things that we should learn before starting to learn deep learning, which is why we postponed the deep learning and neural network to Chapter 10 and later chapters in this book.

Why is deep learning popular and powerful?

There are two main reasons why deep learning is successful and popular: superior accuracy and no need for feature engineering

Superior accuracy: The first reason is its superior accuracy while dealing with unstructured data (not tabular data). Other algorithms that we have described all require to have structured data. From tabular formats like CSV to a time series and signals. On the other hand, deep learning works very well for unstructured data, including image, audio, and textual data. Since most of the data we acquire is in unstructured format and conversion to structured format is either



infeasible or very expensive, deep learning algorithms are gaining tremendous popularity working with these datasets.

No need for feature engineering: The second reason that deep learning is superior to other machine learning algorithms is that there is no need for feature engineering in these algorithms. We dedicate one chapter (Chapter 6), at least in this book, to describing feature engineering and selecting the appropriate features for the algorithm, which is one of the most important challenges of data science. Feature engineering consumes a significant amount of time to prepare data for the algorithm. Deep learning algorithms, in particular, have hidden layers that handle all features. If a feature does not contribute to the prediction, its weight will be reduced, and this feature will be ineffective in the final model. It is very exciting for data scientists, especially for those who do not have the domain knowledge to do feature engineering. The hassle of preparing data for the machine learning algorithm is removed. Nevertheless, in many cases, we still do some data engineering, such as resizing the input data (e.g., images for computer vision models), augmenting the data, and increasing the dataset size because deep learning models are trained on large datasets.

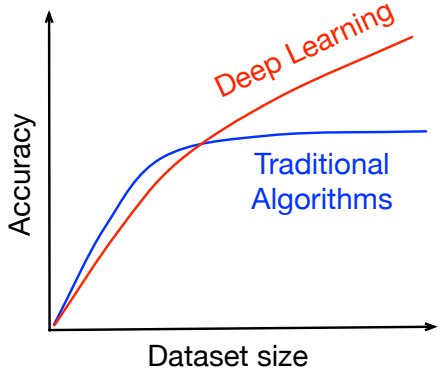


Figure 10-1: Differences between deep learning algorithms and traditional machine learning algorithms

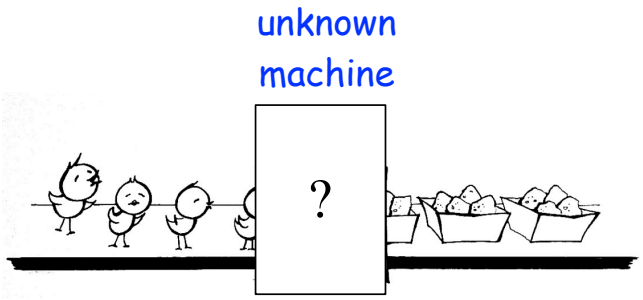
However, there is a "no free lunch" theorem, and the process of automatic feature extraction is associated with a huge cost, which is intensive computation. Therefore, scientists move their calculations from CPU to GPU (Graphical Processing Unit) for matrix multiplication. The popularity of GPU led NVIDIA to launch the first platform for coding for GPU, i.e., CUDA, and later, similar tools came into the market, such as OpenCL[1].

[1] https://www.khronos.org/opencl

Although neural networks, especially deep learning algorithms, outperform all other machine learning algorithms, they operate under the conditions that we can feed them large amounts of data. Figure 10-1 presents an abstract overview of the accuracy between traditional algorithms and deep learning. This figure is inspired by the drawing of Andrew Ng's deep learning course and Aggrawal's book on neural networks [Aggarwal '18].

Universal Approximation Theory

Supervised machine learning can be interpreted as a function approximation problem. We have some data, and there is an *underlying function that is not known.* We try to understand that function. Take a look again at the Figure we have explained in Chapter 1. We have a machine that receives chicken as input and nuggets as output. The process of supervised machine learning is to identify what this unknown machine (function) is.



This technique is also called *function approximation*. The name of this unknown function is often called the *target function*. We do not know this function, and therefore, we use machine learning to approximate this function. We can refer to deep learning models as *non-linear function approximation methods*. We will learn more about non-linearity later.

The Universal Approximation theorem states that a neural network can approximate any continuous function with a high degree of accuracy (given the right configuration). This theorem is often cited to explain the potential of neural networks to capture complex patterns within data. Deep learning, which involves neural networks with multiple hidden layers (we will explain hidden layers shortly), allows for the modeling of highly complex functions.

However, there are practical limitations to consider, such as the computational resources required as network complexity increases. Although, theoretically, neural networks can approximate any functions, in practice, the feasibility of constructing such models depends on having sufficient data, as well as the computational capacity to train large networks. Therefore, while neural networks hold a universality in function approximation, this does not imply they can model absolutely everything without constraints.

Biological Neural Network

The brains of animals and humans are responsible for thinking, decision making, and learning new things. The brain is composed of neurons that are connected to each other; the process of thinking and decision making is done by sending electrical pulses between neurons. Figure 10-2

shows a simplified biological neuron, which is composed of three components: dendrite, axon, and terminals.

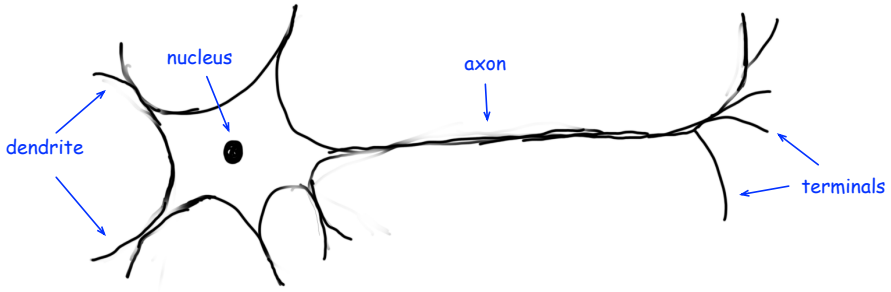


Figure 10-2: A simplified shape of a biological neuron and its components.

Neurons transmit an electrical signal from dendrites to terminals through axons. Then, the signal passes between neurons with the same structure. Based on this explanation, a simplified brain structure could be shown as a set of neurons that pass the electrical signals along with each other. Figure 10-3 presents several neurons where signals are transferred between them.

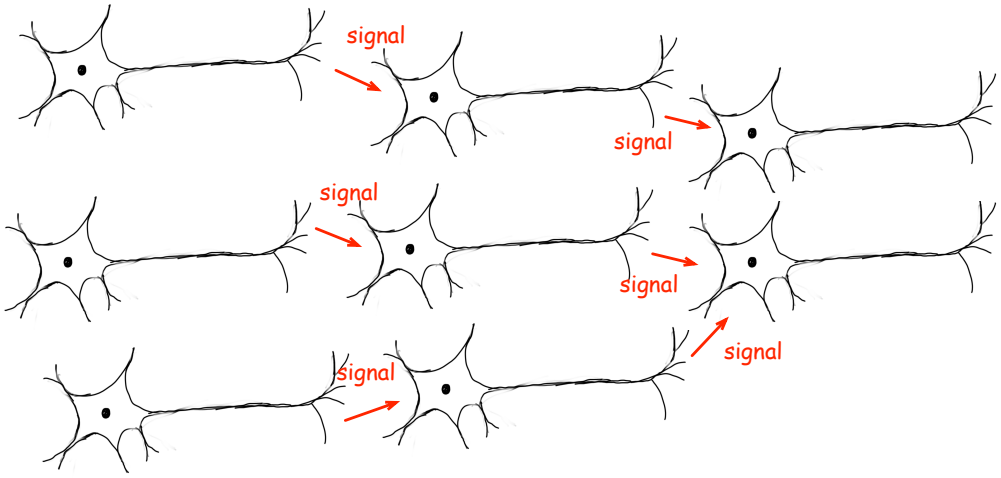


Figure 10-3: A simplified version of brain and how neurons transfer electrical signal.

The process of transferring electrical signals through neurons is called *firing*. A question might arise: when does a neuron fire a signal? Scientific evidence suggests that the nucleus of a neuron should reach a *threshold* of receiving an electrical signal, and it fires the signal *only if that threshold is reached*. If the nucleus does not reach that threshold, it does not fire the signal. It makes sense because noises, which are signals less than the threshold, are not transferred, and only information that is useful will be transferred. To summarize, a biological neuron gets a set of signals and processes them, and if the output reaches a threshold, fires another signal.

Figure 10-4 presents a single function that shows how a neuron fires a signal. It looks like a function with a binary output of 1 (fire) or 0 (not fire). The function that is presented in this figure is called the *step function,* and the function that is used to activate a neuron is called the *threshold function*. The step function has a zero value until x reaches a specific threshold, and then the value of y changes to 1.

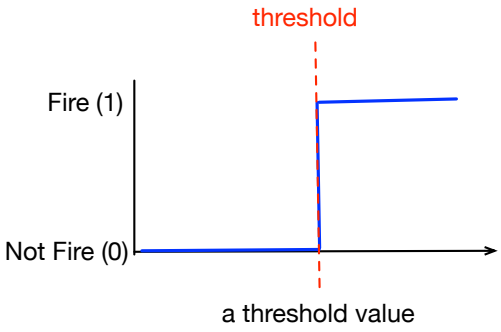


Figure 10-4: A neuron will not fire any signal until the signal reaches a threshold. Then it will get fired. This is called threshold function as well.

However, note that processing information inside the brain is in *parallel* and *fuzzy* (not just binary). Parallel processing and fuzziness are two prominent features of the biological brain.

Now that we have learned a simplified version of how the brain of animals works, we can go and stay in front of the mirror and grant ourselves a full professorship in Neuroscience. Please accept our humble congratulations on your achievement, and continue reading the rest of this chapter.

Artificial Neural Network

The concept of the first digital neuron, that is inspired by biological neurons, was proposed by McCulloch and Pitts in 1943 and is known as the MCP neuron [McCulloch '43], and it is known as the MCP neuron. Nevertheless, it was too limited, and thus it is not used anymore. Later, the Perceptron algorithm was created in 1958 [Rosenblatt '58], and it is the simplest artificial neural network (ANN).

Figure 10-5 shows a simplified biological neuron and a simple artificial neuron known as a perceptron. Intuitively, we can observe and realize the similarity of the perceptron to the biological neuron. An artificial neuron is a mathematical model with a set of inputs (similar to biological dendrites); each input is associated with a weight and a bias as parameters of a neuron, similar to regression parameters (βs) we have described in Chapter 8. Thus, we can write them as $x_1w_1 + x_2w_2 + \ldots + b$. Then, a function called the activation function is applied and sends the result to the output. Activation function could be written as a, and thus, we have $a(x_1w_1 + x_2w_2 + \ldots)$. A neuron output[2] can be written as $y = a(w_1x_1 + w_2x_2 + \ldots)$.

Weight: We have explained that each signal inside a biological neuronal network could have a different amplitude (e.g., micro-voltage), and these will be simulated in artificial neurons by weights. To better understand the need for weight, consider the prediction error, written as *error = predicted value - actual value.* If the *error* is non-zero, the perceptron algorithm changes the w of input variables and redoes the process to reduce the prediction error. Weights are very similar to coefficients in regressions, as we explained in Chapter 8.

[2] You might think it is nothing that multi-linear regression inside an activation function; that is correct a simple artificial neuron (perceptron) can be interpreted as a multi-linear regression inside an activation function.

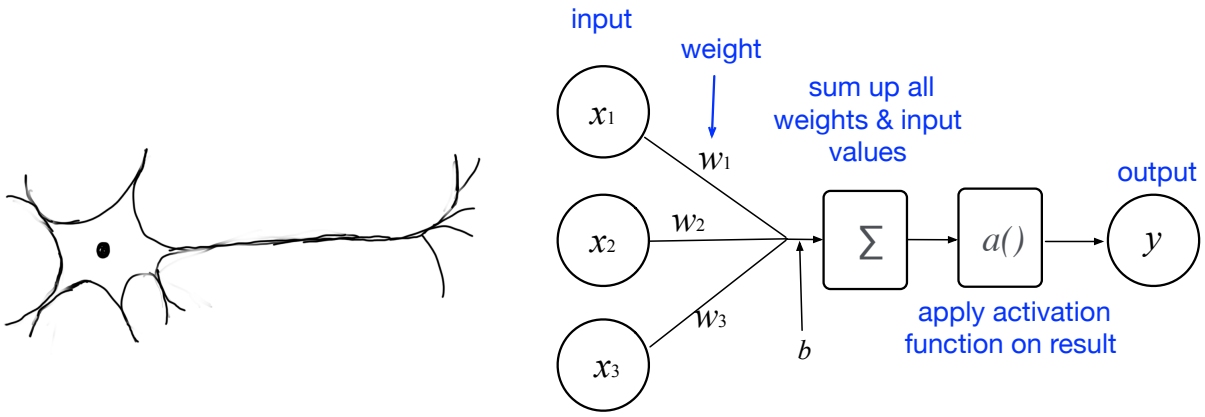


Figure 10-5: (left) a biological neuron, (right) a simple perceptron which its architecture is inspired by biological neuron.

Usually, weights are initialized with a random value, e.g., 0.1 to 0.9. There are different weight initialization methods (at random, with all zeroes, with small values close to zero, and so forth), and the user should decide on the weight. In other words, initial weights could be considered hyperparameters. Increasing the weight increases the complexity of the network, and thus, it is better to keep weights small. Usually, weights are initialized close to zero values or randomly assigned, but there are other approaches that we will explain later in this chapter.

Bias: Similar to linear regression, which has a constant value, each neuron also has a bias, presented as b. Weights adjust the strength of the connections between neurons. Biases, on the other hand, adjust the output of a neuron before it is passed through the activation function.

Activation function: It is a function that receives the sum of weighted inputs and calculates a threshold value (take a look at Figure 10-4). Based on the result of the threshold value, the activation function decides the output value. This function could be linear or non-linear. Linear here means that a hyperplane exists to model with an algebraic equation, and this hyperplane can classify the dataset (separate data points). However, non-linear activation functions such as the Sigmoid function (Check Chapter 8 for explanation about Sigmoid) are more popular. There are different choices for activation functions, which we will describe later.

We could say that an artificial neuron is nothing more than linear regression and that its result is fed into an activation function, i.e., $y = a(wx + b)$, and the activation function imposes non-linearity on this linear regression.

A neural Network is a set of connected neurons in which the output of its neurons is the input of other neurons, see Figure 10-6. All neural networks have three layers: input, hidden, and output.

Input layer: The input layer is the input variable(s) we feed into the neural network. Often, each feature is presented as one input neuron. For example, a table with 10 numerical columns will have 10 input neurons, and they are independent variables, e.g., one row of a table presents one set of input, and each row is composed of n columns (features). Therefore, the input layer has n features (one neuron for each feature). Besides, remember that the only input type we can use for

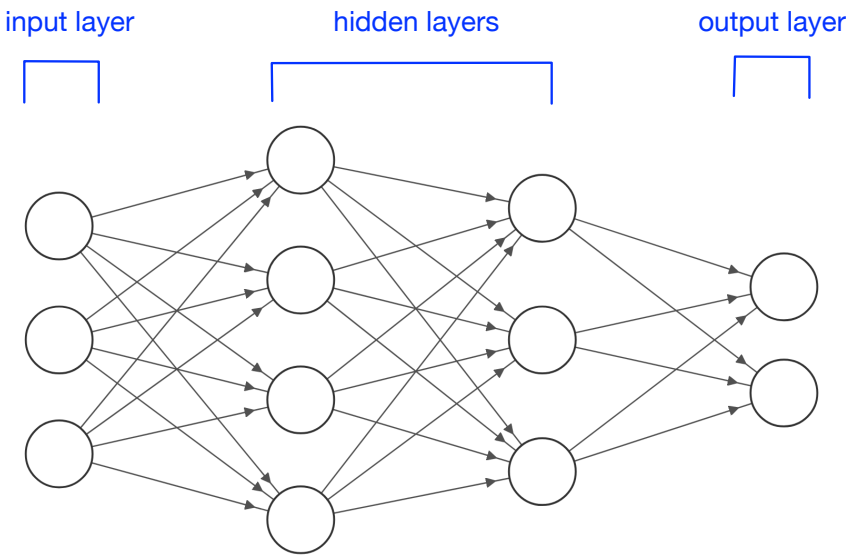


Figure 10-6: A simple presentation of neural network and its layers. You can realize it is also look a like Figure 10-2 but with artificial signal.

the neural network is *numeric,* and other data types, such as text, image, audio, etc., should be converted into numeric data types. For example, pixel data from images can be scaled (e.g., between 0 and 1), standardized, and then fed into a neural network, or words (in a textual format) should be converted into a vector of numbers and then fed into the neural network.

Hidden layers: They are layers that are located between the input layer and the output layer. The simplest neural network has one neuron as a hidden layer. Deep learning algorithms have two or more hidden layers, and because of that, they are called *deep neural networks*. Hidden layers are connected and construct the neural network, along with input and output layers. Each layer is responsible for transforming the received data for the next layer. Since they transform the data and different from the known input or output, we cannot interpret hidden layers. Therefore, deep neural networks are *black box*[3] models. In other words, a layer takes the input as a set of tensors (Check Chapter 1 to recall tensors), and the output of each layer is also a set of tensors, which is a numerical dataset. However, we can not interpret the tensor transformations between hidden layers.

Output layer: It presents the result of the neural network model. Similar to the input, the output of a neural network is also a number, i.e., tensor, which we can convert back into its original value by decoding.

Three common technical tasks could be done with an artificial neural network: binary classification, multi-class classification, and scalar regression. Therefore, the type and number of neurons in the output layer depend on the problem we intend to solve. A regression neural network can have one single neuron as the output layer. A binary classification neural network

[3] There are methods for interpreting the hidden layers, such as activation maps, feature visualization, etc. We describe in Chapter 16, briefly some common tools for data interpretation.

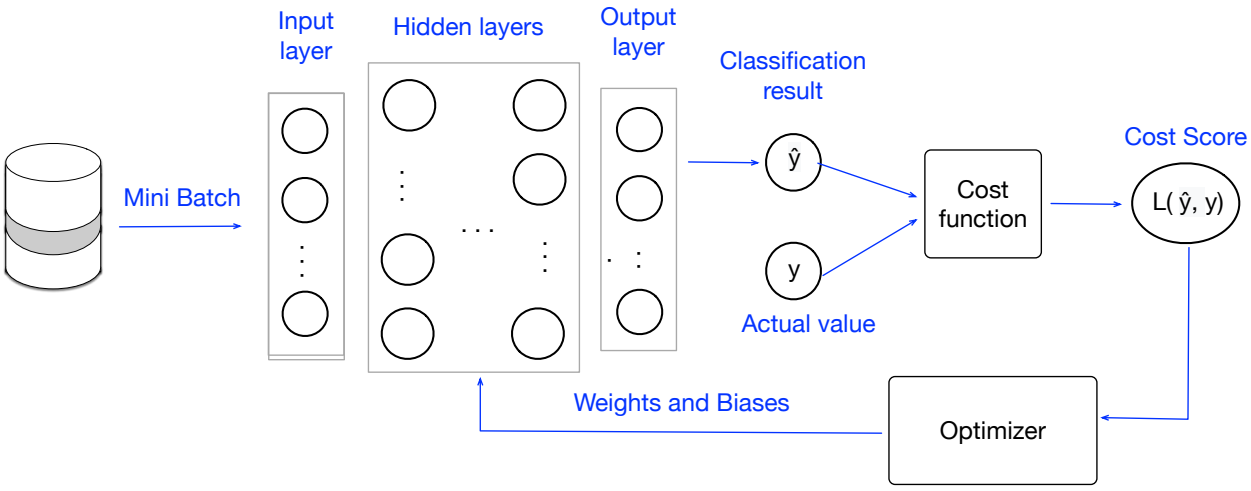


Figure 10-7: Artificial Neural Network flow of operation. The output value will be analyzed by the cost function and the loss score is calculated, then based on the loss score, the optimizer reconfigure weights of the neural network.

can have a single output, which provides zero or one or a probability. The result is determined from the probability, e.g., less than 0.5 is zero, and larger than 0.5 is one. A multi-class classification neural network has multiple output neurons.

Now that we have learned the neurons and layers of the neural network let's discuss the objective of a neural network algorithm. The objective of a neural network is to assign a correct label to the unlabelled data or make predictions or decisions based on the input data. This objective will be achieved by assigning proper values to weights and biases, testing the network, then reconfiguring weights and biases, and reevaluating the network. This process occurs in several epochs until the neural network provides satisfactory accuracy or a maximum number of iterations is reached.

Cost (objective) function: To measure neural network accuracy, we measure how different the machine-assigned labels (predicted values) are from the correct labels (actual values). This will be identified by the cost (objective) function. We have described the cost function in Chapter 8, and use that definition here as well. To review: the cost function is used to measure loss in the neural network. The neural network algorithm identifies a loss score for the output layer. Then, after each iteration, it reconfigures the network's weights and biases to reduce the loss score.

Optimizer: It is the algorithm that is responsible for reconfiguring the network's weights and biases, such as SGD, which we have explained in Chapter 8. In other words, the optimizer uses the loss value to update weights in the network and thus reduces the loss value for the next round (epoch).

Based on this explanation, a neural network has three core components: *hidden layers*, *cost function*, and the *optimizer algorithm*. Figure 10-7 describes the process of how a neural network works and its components. The design of this figure was inspired by an excellent explanation provided in Chollet's book [Chollet '18].

Learning in the context of a neural network means finding weights and biases that minimize the cost function result (cost score) for the given dataset. Updating the weights toward improving the accuracy of the neural network is the task of the optimizer. The process of changing weight and calculating the cost continues iteratively until the number of specified epochs reaches the neural network. We have explained in Chapter 8 that epoch refers to one round of scanning the dataset, and it is usually given to the neural network by the user as a hyperparameter.

Now, we understand that a neural network algorithm assigns the weights, runs the network, measures the cost, and then updates the weights to reduce costs via optimizer.

NOTE:

* It is uncommon to report the computational complexity of neural network algorithms anymore because they are known to be very complex and require lots of resources, especially for training them.
* The format of data that is transferred between layers in the neural network is a tensor, and a tensor is a multidimensional array of numbers (scalar is a 0D tensor, a vector is a 1D tensor, a matrix is a 2D tensor, and Rank 3 tensor is 3D tensor). For example, an image could be a 3D tensor (x, y coordinates, and one value for color channel, i.e., RGB value), or a video could be a 5D tensor (images' data plus the timestamp of each frame, and audio wave of each frame).
* A neural network does not feed all members of the training dataset once into the network; it either feeds input data one by one or a set of input data as mini-batches into the neural network.
* The neural networks' input and output variables present the same observation, but the output has a label assigned to the same information.
* We can say a single neuron network is nothing more than a linear regression ($y = \beta_0 x + \beta_1$, but here we use b and w instead of β_0 and β_1) that its output will be fed into an activation function. Therefore, the input will be a value for x, the neural network assigned w , and b to that identify a y. This y will be fed into an activation function, and the result will be the output. Therefore, considering the content of the bracket is what is happening inside a neuron, we can formalize a single-neuron network as follows: $x(input) \rightarrow [z = wx + b \rightarrow \sigma(z)] \rightarrow y(output)$

Perceptron Algorithm

Perceptron [Rosenblatt '58] is one of the simplest and easiest neural network algorithms, which performs a binary classification. It receives a vector of input values and calculates a linear combination of input variables' values. If the results are greater than a threshold, the output will be equal to 1, otherwise, the output will be equal to -1. Assuming our input tensor has a d number of features ($x_1, x_2, \ldots x_d$), the following equations formalize the binary classification of the perceptron algorithm.

$$\hat{y} = a(\sum_{i=1}^{d} x_i w_i + b)$$

Why do we use weight (w) for each feature? Because $w$eights determine the contribution of each input feature to the output ($\hat{y}$). In other words, they specify the importance of the neuron. However, if a particular x value is zero in some instances, the weight impact will be zero as well. To handle this problem, a bias b_i will be added to the input as well. Bias can be interpreted as offset, which assists the $x_i w_i$ reaches a specific threshold, and it has an impact on the output variable.

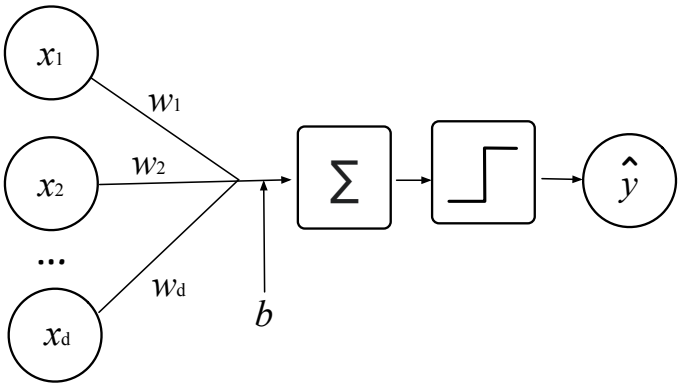


Figure 10-8: A single perceptron that receives a *d* dimensional input data. *b* presents the bias which is added to the result of summation. The first square is summing all weights and inputs together. The second square presents the activation function.

Figure 10-8 shows the simplest form of the perceptron. Σ is presenting a summation of input variables along with their weights and biases, i.e., $\Sigma_{i=1}^{d} x_i w_i + b$. The other rectangular box, with a shape similar to S, presents the activation function, i.e., $a(\Sigma_{i=1}^{d} x_i w_i + b)$.

Now, we have realized that the content of the equation in $a(.)$ function is just a simple linear regression, but perceptron is not only a linear regression; we need to have a constraint for the output to be able to classify each output value and assign a label to it. For example, all outputs should be either 0 or 1. This is the job of the activation function $a(.)$. We can conclude that the *activation function* is used to set constraints on the output values of neurons. If we use the Sigmoid function (check Chapter 8 to recall the Sigmoid), the activation function is a logistic regression, and a single perceptron is just a logistic regression.

Single perceptron is not very useful, and we use Multilayer perceptron, which we will explain later. Perceptron (similar to logistic regression algorithms) operates based on the assumption that there exists a hyperplane, which separates two sides of the dataset. It starts with random weights that are close to zero. Then, it classifies the dataset. Next, it goes back and checks the data points that have been misclassified by analyzing the cost of the predicted value ($\hat{y}$), which is acquired by comparing it to the actual value (y). The cost function of Perceptron is very simple, and it simply uses the step function (we will explain it later). It means it uses a rule-based update: if the output is correct, weights remain the same; if not, weights are adjusted. Then, the neural network changes the weight of misclassified data points to reduce the cost score, and again performs the

classification, then checks the new result. This process continues until it reaches a specific number of iterations, or weights cannot be further changed.

In general, the process of going back to the network and changing the value of weights to improve the accuracy of output is called *backpropagation* or *(Back-Prop)*, which will be explained later in detail. However, Perceptron is much simpler and doesn't use backpropagation.

Multilayer Perceptron

A Multi-Layer Perceptron (MLP) is an artificial neural network, consisting of fully connected neurons with a nonlinear activation function, organized in at least three layers, notable for being able to distinguish data that is not linearly separable.

Before we explain Multilayer Perceptron, we should review binary logic, which we learned back in high school. Table 10-1 presets possible values of two binary variables A and B, which can get binary values (0 or 1) and the result of applying a logical operation on these two variables. To recall them from school time, OR is a logical operation that outputs true or 1 when at least one of the input values is true or 1, while AND is a logical operation that outputs true or 1 only when both input values are true or 1. XOR (exclusive OR) is a logical operation that outputs true or 1 only when the two input values are different.

A	B	Not A	Not B	A AND B	A NAND B	A OR B	A XOR B
0	0	1	1	0	1	0	0
0	1	1	0	0	1	1	1
1	0	0	1	0	1	1	1
1	1	0	0	1	0	1	0

Table 10-1: Some basic binary logic operation between two variables.

Logical reasoning plays an important role in electrical engineering, computer science, and circuit design, but for our needs, it is enough to know these operators.

We have explained that a single layer perceptron is good for classifying linearly separable datasets. In the context of the logical operator, the perceptron algorithm can be used to represent boolean AND, OR, and NAND functions. However, a single layer perceptron cannot represent the XOR[4] boolean operator. In other words, it is impossible to use it for a non-linearly separable dataset [Minsky '69] with a single perceptron.

It is possible to model XOR with multiple perceptrons (a network of perceptrons). To understand this problem, let's use some visualizations with sample data points. Figure 10-9 present four data

[4] XOR means that both operands should be different, i.e. true and false. For example, (P XOR ~P) = true

points for the A and B variables described in Table 10-1. At the bottom of each plot, there is the perception that can model the distinction between the blue and red areas of its plot on top.

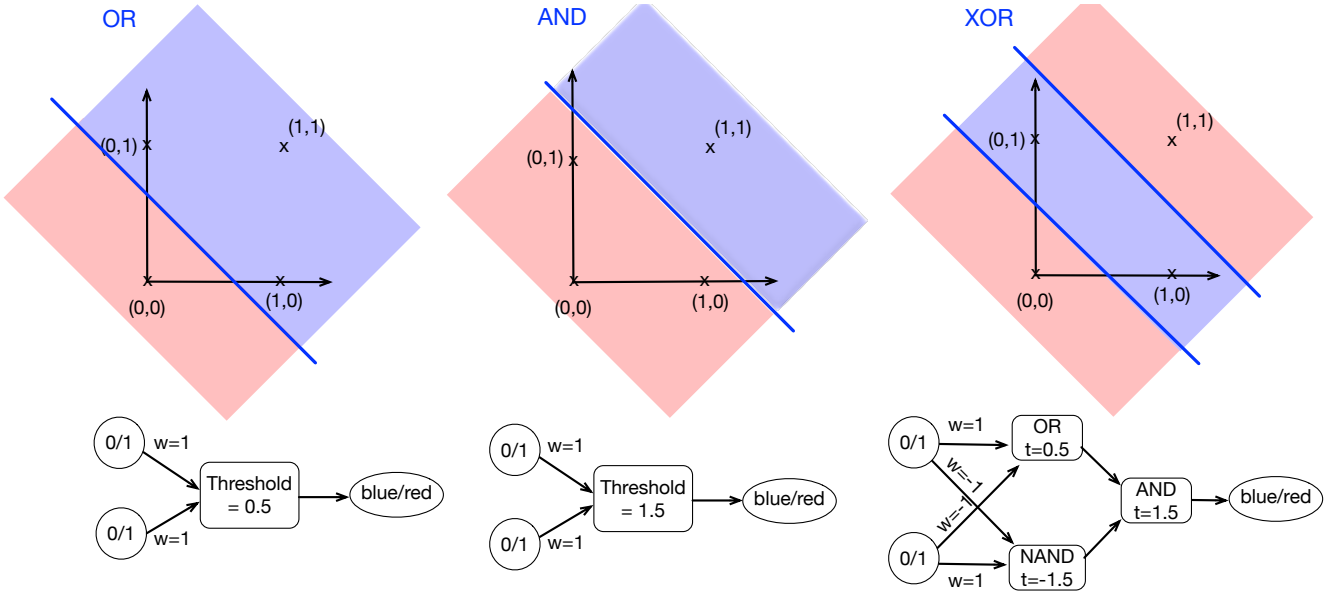
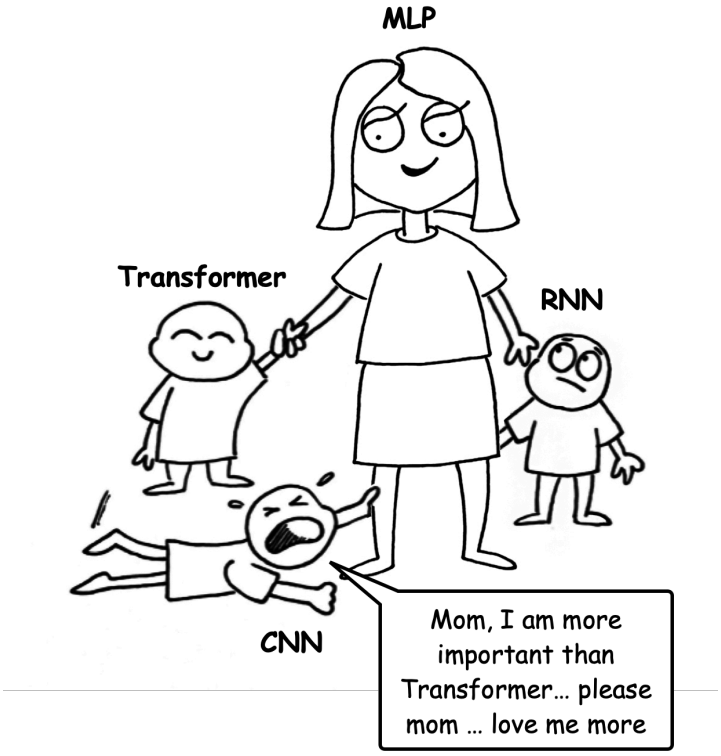


Figure 10-9: (Top) Four dots represents different status of A and B. AND and OR could be separated with linear hyperplane but XOR is not separable with a liner hyperplane. In the right diagram, we need some non-linear function to separate the blue area from the rest. (Bottom) Both AND and OR can be implemented as a perception with the step function.

All these perceptrons use a *step function* as their activation function. As we have explained, a single perceptron can linearly classify the data point. In the context of logical operators, AND, OR are linearly separable. Nevertheless, the XOR operator cannot be implemented with a single perceptron, and we can see that the XOR area is not linearly separable, unlike AND or OR, which can be separated with a single line. Linear separation can be implemented with a single perceptron and threshold activation function, as shown for AND and OR. XOR cannot work with one layer and requires at least two connected layers of perceptrons. It means the output of one layer will be fed as the input of another layer. This neural network has more than one perceptron and it is called a 'Multilayer Perceptron (MLP)'. An MLP can handle non-linear separable data as well.

In other words, MLP, which is also called a *feed-forward neural network*, is a solution to separate non-linearly separable datasets, such as the situation we have observed for XOR. As we have explained, layers between the input layer and output layers are called hidden layers, and deep learning algorithms have lots of hidden layers. Therefore, it is not wrong to say that the MLP algorithm is the mother of the deep learning algorithm.

The more layers we add, the more complexity we can handle by the algorithm. It doesn't mean that too many layers are always good. Too many hidden layers impose a huge cost on resources and could cause overfitting, which we will discuss it later.



To be sure you leave this section with a Ph.D degree in MLP, take a look at Figure 10-10 (a) as another example. Here, we have blue and red dots in two-dimensional space, and we would like to classify them. By using a single perceptron, we have a step function, and it can fit these data points, as shown in Figure 10-10 (b). The result can classify blue dots as 1 and red dots 0, but two red dots are misclassified. By using two step functions, which are possible through MLP, all data points can be classified correctly. We can see the result in Figure 10-10 (c) that blue dots are one, and red dots are 0.

Every layer in the MLP architecture, except the output layer, is fully connected to the next layer. This is known as the *fully connected layer*. This means that every neuron is connected to all other neurons in the next layer, as shown in Figure 10-11. Also, keep in mind that every layer, except

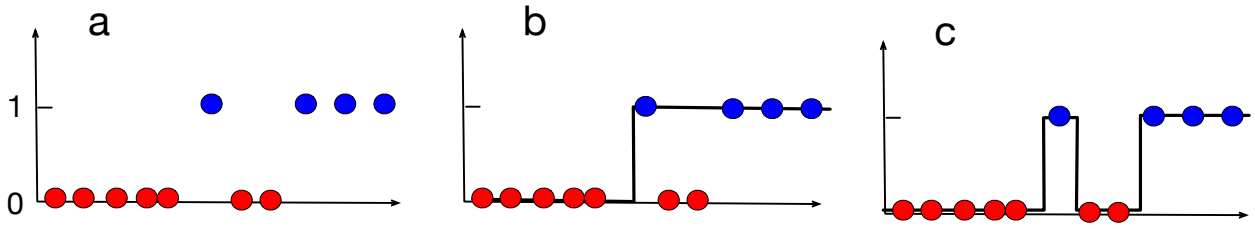


Figure 10-10: (a) original dataset (b) Using one single perceptron with a threshold activation function we can classify most data points correctly. However, two red dots are misclassified. (c) By using MLP with threshold activation function, all data points were classified correctly.

the output layer, includes biases as well. Figure 10-11 presents how we write weights. We write them on the top of each *w* its layer number, and at the bottom of it, its index.

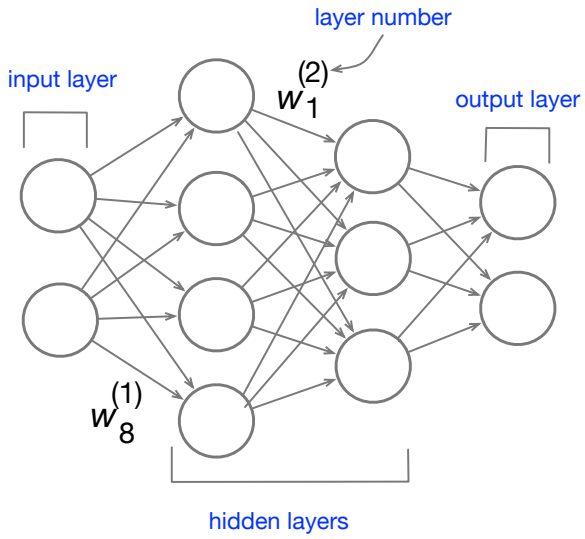


Figure 10-11: Sample MLP network, which all neurons are fully connected to all other neurons in the next layer.

Activation Functions

Activation functions are mathematical equations used to set boundaries on each neuron and decide the output value of a neuron. We can write the *output of each neuron* as $z = wx + b$, and $a(z)$ is the activation function that changes z to the neuron's output.

There are many activation functions available. Some activation functions focus on binary decisions, such as the step function, which we have explained. Some activation functions, such as softmax, can handle multi-classes, e.g., objects that appear inside a photo (cat, dog, human, etc.). In the following, we list some common activation functions, and they are visualized in Figure 10-12.

Step function: The simplest form of binary class activation function is the step function, used for binary classification. It has a similar shape to the function presented in Figure 10-4, but the threshold is z, and if it is greater than z, the signal is fired ($\hat{y} = 1$). Otherwise, it will be 0 ($\hat{y} = 0$).

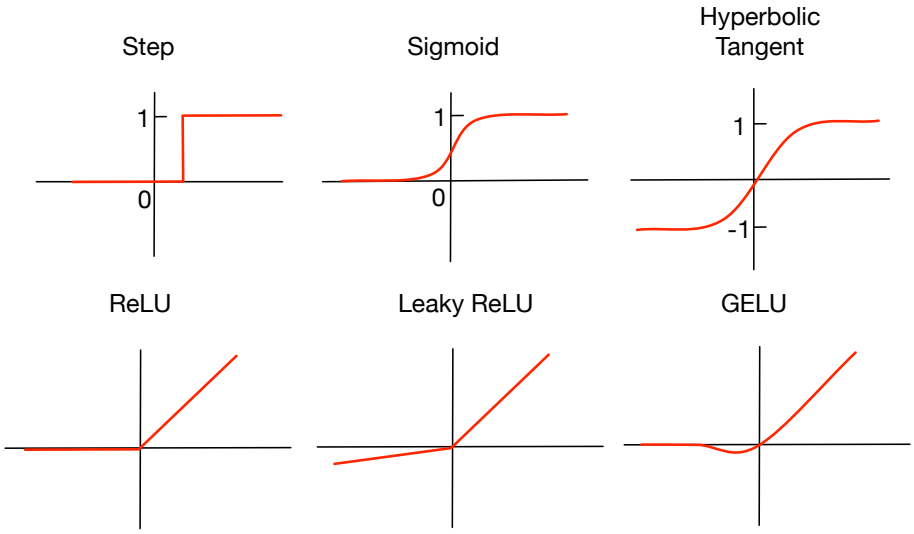


Figure 10-12: Some popular Activation functions.

Sigmoid function: It is smoother than the step function and uses the Sigmoid equation, which we have explained in Chapter 8, for logistic regression. The Sigmoid function is written as follows:

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

Sigmoid is more popular than step function because it is slightly more sensitive to small changes in the output value, and it is better suited to report probabilities because it gives a value between 0 and 1, step function only gives either 0 or 1, and no other numbers in between. The sigmoid function involves exponential calculations, which can be computationally expensive and slow down the training process, especially for large networks.

Hyperbolic Tangent function: it is another activation function, and its equation is written as follows:

$$a(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

It calculates a ratio between hyperbolic sine and hyperbolic cosine, i.e., the ratio of the half-difference and half-sum of two exponential functions in the points z and $-z$. This activation function is very similar to the Sigmoid function, but its range is from -1 to 1. It is used when we do not want to have zero as the lower boundary of the activation function.

Rectified Linear Unit (ReLU) function: ReLU [Nair '10] is a very common activation function (probably the most popular binary activation function), and we can easily write it as $max(0,z)$. It means if the output value is less than zero, the RELU activation function considers it as 0; otherwise, the value of the output is the actual output value. ReLU is a useful activation function while dealing with the vanishing gradient problem, which will be explained later in detail. The following equation formalizes the ReLU.

$$a(z) = \begin{cases} 0 & for\ z < 0 \\ z & for\ z \geq 0 \end{cases}$$

ReLU function has a specific characteristic, and it can be used in networks with many layers.

Leaky RELU (LReLU): LReLU [Maas '13] is another common activation function that allows a very small negative value to have a non-zero variable. Not having zero is useful because it mitigates the challenge of vanishing gradient better than ReLU, which is called *dying ReLU*. The LReLU equation is written as follows.

$$a(z) = \begin{cases} 0.01z & for\ z < 0 \\ z & for\ z \geq 0 \end{cases}$$

Having zero gradients can also result in slow learning, and using ReLU and LReLU resolves this. In other words, The derivative of the ReLU is 1 in the positive part and 0 in the negative part. The derivative of the LReLU is 1 in the positive part, and it is a small fraction in the negative part. If this explanation does not make sense now to you, be patient; after you have learned backpropagation and chain rule, it might make more sense.

Gaussian Error Linear Units (GELU): The GELU is a nonlinear activation function based on the standard Gaussian cumulative distribution function [Hendrycks '16]. Later in this chapter, we learn to regularize a neural network. Sometimes, we set some weights randomly to zero, i.e., *dropout*. In the context of GELU, the function smoothly varies between 0 and 1, modulating the output based on the Gaussian CDF, which gives it a probabilistic interpretation somewhat similar to dropout. In particular, assuming the $\Phi(x)$ is the standard Gaussian cumulative distribution function (CDF if you recall from Chapter 3), each data point x will be multiplied by this value, $x\Phi(x)$, and is the output of GELU activation.

This activation function is used in several transformer-based models, GPT-3, BERT, and wav2vec 2, which we will describe later in Chapter 12.

Softmax: It is an activation function used for multiclass output values. We cannot visualize it in Figure 10-12 because it has multi-dimensions (equal to the number of output classes). Assuming we have *k* number of classes, the softmax is presented with $\sigma(\,.\,)$ and it is formalized as follows:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad for \quad i = 1,...,k$$

Softmax activation function calculates probability distributions of the particular output over *k* different labels. For example, we have a neural network to recognize if the given image includes a human, cat, or dog, then our *k=3*. Softmax calculates the probability of each target class over all possible target classes.

To understand how Softmax works, we use another example. Assuming we are creating an algorithm to identify infection in microscopic blood samples, we have developed a neural network algorithm to identify specific types of infection in the blood sample. There are three types of infection. X, Y, and Z. First, our algorithm analyzes an image, and then probabilities for infection probabilities are calculated using Softmax as follows: $X : 0.3$, Y:0.6, and $Z : 0.1$. We can see that their sum is equal to one, the output of Softmax is a probability function, and thus the sum of the label values is equal to 1. In summary, based on the number of possible outputs, Softmax assigns each of the outputs a probability value, and the one that has the highest probability is the correct output.

Let's review why we need an activation function. Neural networks are linear transformations, and chaining many linear transformations ends up having a linear transformation again. This means the neural network cannot solve complex problems by just having a linear transformation. For example, if we have $f(x) = x - 1$ and $g(x) = 2x + 1$, chaining them will be $f(g(x)) = (2x + 1) - 1 = 2x$, which is still a linear function. Therefore, activation functions are used to enable the neural network to perform a non-linear transformation.

SiLu (Swish): Swish activation function is smooth, non-monotonic, and unbounded above, allowing for more complex representations in neural networks and it combines the benefits of both linear and non-linear functions. The Swish activation function, is defined as follows:

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

A variant of Swish includes a learnable parameter β, and its computed as follows:

$$Swish(x) = x \times sigmoid(\beta x)$$

Swish approximates a linear function for positive inputs and ReLU for negative inputs, which helps mitigate the vanishing gradient problem in deep networks.

Neural Network Cost Functions

We have explained that a neural network has three core components, and the cost function is one of them, as is shown in Figure 10-7. We have also introduced and explained Chapter 8 about the motivation and use of cost functions. In the context of a neural network, a cost function uses the given input to measure how accurate the output is. In other words, after the neural networks construct the output, the algorithm uses the cost function to report the loss score $L(\hat{y}, y)$. Basically, the loss score is calculated by the cost function and presents how far the algorithm output is from the true value.

A cost function for a single neuron can be formalized as $C = (W, B, S_r, E_r)$, in which C presents the cost value, W presents weights, B presents biases, S_r presents the input of a single training sample (e.g., one record of data), and E_r presents the desired output of the given input training sample (S_r). For the entire network, which has n layers and we have m numbers of weights in the last layer, we will have a cost function that depends on all of the weights $C(w_1^{(1)}, w_2^{(1)}, \ldots, w_m^{(n)})$.

Let us remind you, that when we report something specific to a layer, we use superscript. For example, $a^{(2)}(x)$ presents the value of the activation function at the second layer.

Here, we list four cost functions that are common for neural networks. Some of them have been explained in Chapter 8 and Chapter 3, but here we repeat them because they are widely used for neural networks. We can also define our cost function based on our needs for the neural network and incorporate the domain knowledge into the customized cost function.

Quadratic Cost (Mean Squared Error)

We use this cost function for regression as well, and it is known as "root mean square error". Assuming $y(x)$ presents the actual value of output, a_i^L or y presents the actual value in the last layer (L presents the last layer), and $E_{r(i)}$ (or $\hat{y}$) presents the output value. Assuming the test set has n data points, and i is the index of each data point, the quadratic cost equation is written as follows:

$$C = \frac{1}{n} \sum_{i=1}^{n} (a_i^L - E_{r(i)})^2$$

Using simply y and $\hat{y}$ is much easier to understand and memorize, but other literatures use this notation, and it is better to get familiar with this style of writing as well; we follow the common approach while describing these equations.

The quadratic cost function is usually used for neural networks that try to resolve a regression problem. The loss function used in the training phase of MLP is usually Mean Square Error (MSE), if there are many outliers, then it is recommended to use Mean Absolute Error (MAE) instead.

Cross Entropy

Cross entropy is a mathematical function that measures the differences between two statistical distributions. Assuming *P(x)* is one distribution and *Q(x)* is the second distribution, their cross entropy $H(P, Q)$ is written as follows:

$$H(P, Q) = -\sum_{x} P(x) \,.\, log\ Q(x)$$

For classification tasks of neural networks where we deal with labels, we use cross entropy cost function. This function provides a probability distribution for each class label. For example, an algorithm that recognizes your mood from facial expressions after you wake up from bed can provide a list of three probabilities as follows: *{ I bite, don't talk with me: 0.7, full of energy: 0.1, immediately to the restroom: 0.2}*.

Assuming the data points in the test dataset are specified i index, the binary cross entropy equation is written as follows.

$$C = -\sum_{i} [E_{r(i)} \,.\, ln(a_i^L) + (1 - E_{r(i)}) ln(1 - (a_i^L))]$$

Respectively, we can write the multiclass cross entropy as follows:

$$C = -\sum_{i} \sum_{j} [E_{r(i),j} \,.\, ln(a_{i,j}^L)$$

Here, i is the index of samples, and j is the index of classes. The description of other variables in this equation is exactly the same as the one we have explained for the quadratic cost function.

Kullback-Leibler Divergence

Back in Chapter 3, we explained the KL-Divergence function is used to quantify the differences (or measuring similarity) between two probability distributions, which is written as follows:

$$D_{KL}(P, Q) = \sum_{i} P(i) ln \frac{P(i)}{Q(i)}$$

In the context of neural networks, we need a cost function to compare the distribution of output values with the actual values. Therefore, it is written as follows:

$$D_{KL}(E_r, a^L) = \sum_{i} E_{r(i)}\ log \frac{E_{r(i)}}{a_i^L}$$

KL divergence and cross entropy are very similar, and KL divergence is also called relative entropy. Therefore, similar to cross entropy, we can also use it for classification tasks.

Hellinger (Bhattacharyya) Distance

Hellinger distance [Hellinger '09] is also used to quantify the similarity between two probability distributions. Therefore, outside the context of the cost function, we can formalize the Hellinger

distance between two distributions, P and Q, as the L_2 norm (Euclidean norm) used for probability distributions:

$$H(P,Q) = \frac{1}{\sqrt{2}} || \sqrt{P} - \sqrt{Q} ||_2 = \frac{1}{\sqrt{2}} \sqrt{\sum_i (\sqrt{P_i} - \sqrt{Q_i})^2}$$

However, differentiating a square root inside a square root (used by Backpropagation) is computationally complex, and this distance can be relaxed by removing the square root as follows:

$$H^2 = \frac{1}{2} \sum_i (\sqrt{a_i^L} - \sqrt{E_{r(i)}})^2$$

If we intend to ensure that the cost function result will stay between 0 and 1, we can use Hellinger distance and not its squared version.

To summarize these cost functions, cross entropy and all other cost functions we described here, except quadratic cost, measure the differences between two probabilities, in which one probability presents the actual value and the other presents the output value. To review where we use which cost function, check the following:

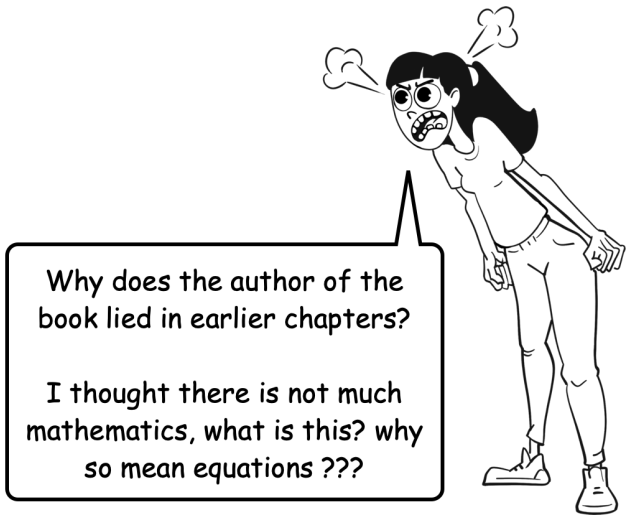
Quadratic cost: Typically used in regression problems where the output is a continuous variable.

Cross Entropy: Commonly used in classification problems, particularly multi-class classification.

Binary Cross Entropy: Specifically used in binary classification problems.

KL-Divergence: Also used in classification tasks, KL-Divergence measures how one probability distribution diverges from a second reference probability distribution.

Hellinger Distance: Used in classification problems as well, especially when a bounded metric between 0 and 1 is preferred.



NOTE:

* Since a neural network includes many neurons and many activation functions used to combine these neurons, activation functions should be computationally efficient. Similar to the similarity metrics that are required to be computationally efficient while we are doing clustering, the same issue existed for the activation function as well.
* Among the equations we described for the cost function, the quadratic cost function is more popular. It is squared to eliminate negative values and increase the impact of errors to highlight

them. Somehow, it punishes the network when the error is large by making it a larger error. If we have a multi-label classification problem, cross entropy is also popular. Nevertheless, we recommend that if you have enough resources, you experiment with all of them. You can change them in the neural network package easily as hyperparameters.

* There are more cost functions, such as the Generalized Kullback–Leibler divergence or Exponential cost function, which we did not describe.
* Some implementations of cross entropy provide one implementation for binary classification and one for multi-class classification, i.e., softmax cross entropy.
* While using gradient-based optimization, Mean Absolute Error and Mean Square Error usually do not yield good results. Therefore, they are not popular to go for neural network cost functions. However, while MSE and MAE are not typically the first choice for classification problems in neural networks, they are still popular and effective for regression tasks.

Neural Network Optimizers

Before starting to read this section, it is worth taking a look to Chapter 8 and reviewing the optimization section we have explained. In Chapter 8, we have explained that the cost function measures how far the true value is from the actual value. The optimizer's job is to reduce the cost. In other words, the goal of the optimizer is to increase the accuracy of the algorithm by changing the model parameters. In the context of the neural network algorithm, this will be achieved by minimizing the cost function through changing weight and biases. At the end of each epoch (each iteration on the dataset), the optimizer changes weights and biases in such a way that the loss score will be reduced in the next epoch. In other words, by adjusting weights and biases, the algorithm can reduce the loss score.

Now, a question arises: What is the best combination of weights that increases the accuracy of the result? To answer this question, we can try every possible combination of weights and biases and see which one is the best. We have bad news; this is not possible. To understand why, let's assume we have a fully connected neural network that has one input layer with four neurons, two hidden layers, each with five neurons, and two output neurons. In this case, we will end up with $(4 \times 5) + (5 \times 5) + (5 \times 2) = 55$ weights to configure. Besides, each neuron in the hidden layers and output layer has its bias, so this adds five (for the first hidden layer) plus five (for the second hidden layer) plus two (for the output layer), which results in 12 biases.

As we have explained, the algorithm starts by assigning a random weight. Let's say we limit the weight changes to a number that can vary between 0 to 100. Therefore, only to find the best weight configurations and not biases, we need to test 100^{67} possible combinations for experimenting and identify which combinations lead to the best minima. It is definitely not computationally possible, even with the strongest supercomputer in the world. Besides, note that neural network optimization is non-convex, and thus, it is very complex to identify good minima or global minima.

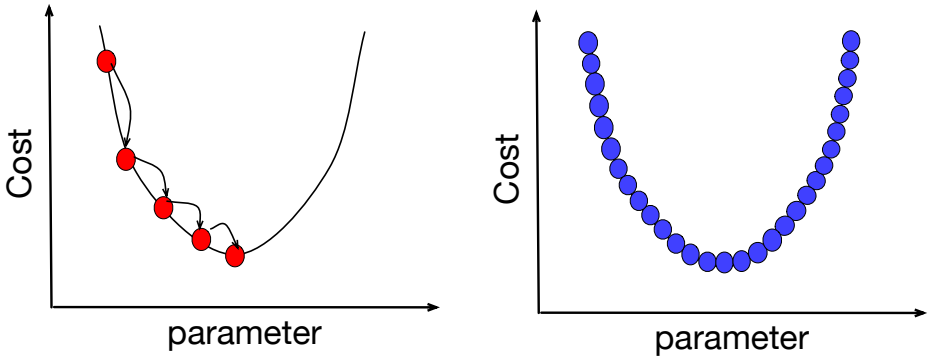


Figure 10-12: (left) Using an optimization to reach the minima and therefore very few data points will be processed. (right) Not using an optimization and thus many data points will be experimented on the function to find the minima.

We need a subtle approach to determine weights and biases. Take a look at Figure 10-12. Here, we have a very simple convex function. This figure presents the differences between using optimization and not using optimization to find the minima. By using blue dots in the right plot, we try to show that there are too many data points to experiment with, but in the left plot, there are few red dots, and instead of experimenting with any possible value, the algorithm can make some jumps toward the minima. This figure is a convex function, and we show it for the sake of simplicity; in the real world, it is much more complicated, and while working with neural networks, there is no convex shape.

Therefore, instead of experimenting with all possible points on the curve, we can experiment with some random points on the function. We use a method such as Gradient Descent to determine the next point and skip some points that are not useful. In other words, from the current data point, we need to find (i) *direction* and (ii) *step size* toward reaching the global minima and avoid experimenting with every single data point. As we have explained in Chapter 8, Gradient Descent helps us to determine the direction by using the partial derivative, which is shown on the left plot in Figure 10-12. At each point, the Gradient Descent algorithm calculates the dervative, and if the gradient sign does not change, it means that it should take the next move in this direction to move toward minima. If the sign changes (from negative to positive), it means the gradient jump is too big, and it should take the next move in the opposite direction to reach the minimum.

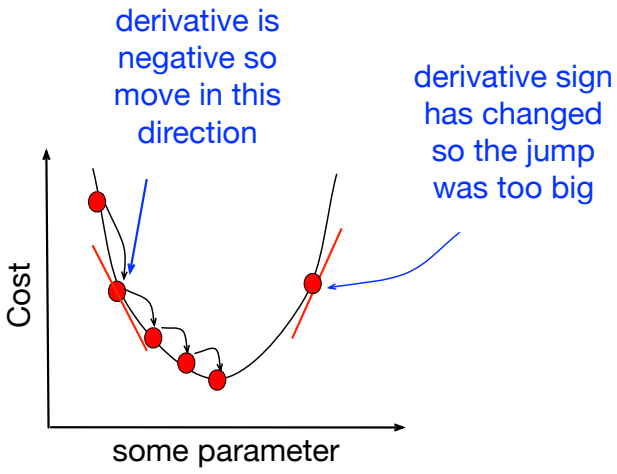


Figure 10-13: Gradient Descent decides the direction of the next point to move toward minima.

Figure 10-13 visualizes the behavior of the Gradient Descent based on the slope of the line. The *X*-axis in this figure presents a model parameter, and the *Y*-axis presents the cost of different values for the parameter. However, having constant size jumps is inefficient because if step sizes are small, it takes a long time to reach the minima, and if they are large, it might pass the minima. Also, if the jump is too big, the gradient sign will change. Therefore, a better approach is customizing the step size, and it is done through a *decaying learning rate*, which we will explain later.

In Chapter 8, we explained three types of Gradient Descent, including SGD, BGD, and Mini BGD. Gradient Descent is the most popular optimization approach for neural network optimization, but it has some limitations. It can identify the direction of the movement, but it does not specify the step size. In this section, we introduce other optimization algorithms that use SGD, and mini BGD, and improve them.

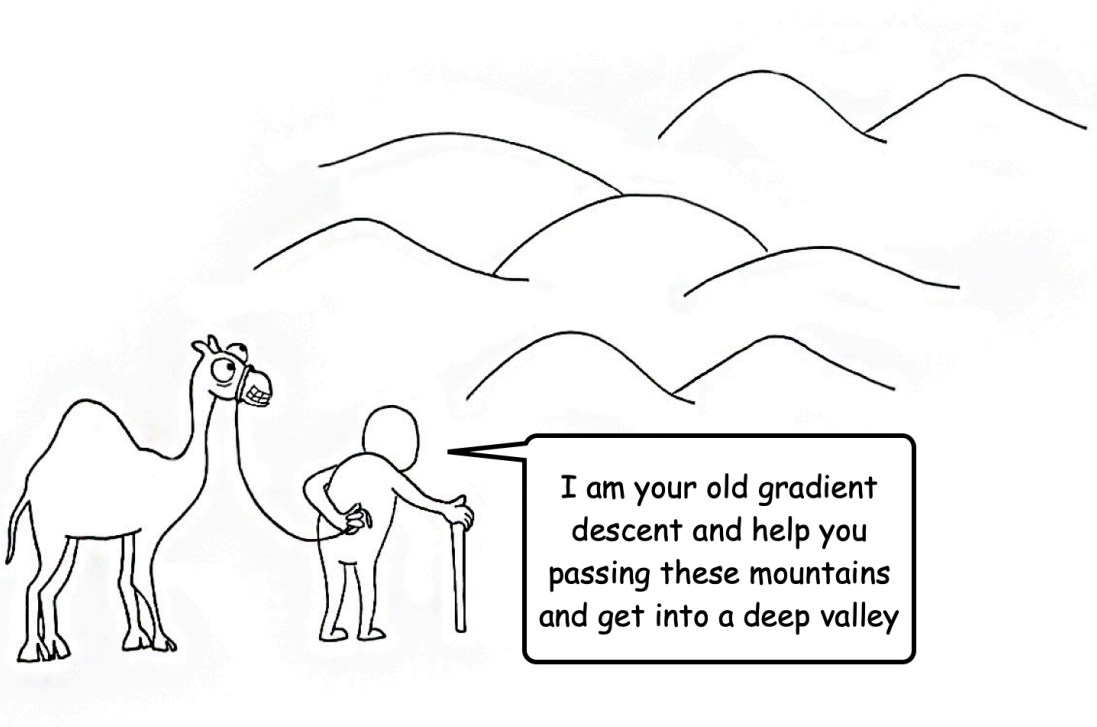
Stochastic Gradient Descent with Momentum

Although mini batch Stochastic Gradient Descent (SGD) has some limitations, it is a very popular optimization algorithm used in deep learning as well. In Chapter 8, we have explained that, while using Gradient Descent, the next value for the parameter is calculated as follows:

next point = current point − (learning rate × slope (a derivative of that particular parameter)

Assuming w_{t+1} is the weight (or any other parameter [5]) at the next epoch, w_t is the weight of the current epoch, α is the *learning rate*, and $\nabla G(.)$ is the vector of gradients. We estimate a value of w_{t+1} via the following equation:

$$w_{t+1} = w_t - \alpha \,.\, \nabla G(w_t)$$



In simple words, $\alpha \,.\, \nabla G(w_t)$ specifies the amount of the move for the next step.

There are two problems with SGD. First, we can see from the described equation that the direction is changing, but the step size is fixed. A fixed step size might trap the optimizer into a local minimum. Second, as seen in Figure 8-25 from Chapter 8, the noise of SGD is high due to its stochastic (random) behavior. This problem can be reduced by using an *exponentially weighted average (β)*. In other words, the exponentially weighted average introduces a *decay coefficient* to a series of numbers. Therefore, applying this exponentially weighted average

[5] While reading text for optimization in the context of the neural networks, some literature refers to parameters as θ or β, we use w to present weight. Nevertheless, bias b is also a parameter for the neural network. Besides, based on the literature, these parameter names change and we do comply with the name change to avoid confusing you while reading other resources as well.

reduces the impact of older data points and, thus, the noise. This type of SGD is called *SGD with momentum* [Polyak '64]. In the following, first, we describe the exponentially weighted average, and then, we explain SGD with momentum.

Exponentially Weighted Average

Weight (β) is a number between 0 and 1, usually set to 0.9. Multiplying a number less than one to another number makes it smaller. For example, let's assume we set $\beta = 0.7$, if we multiply β to something, the result value will be 70% of the original value. If we multiply β^2 by a number, the result will be 49% of the original value, etc. To understand the mathematic intuition of exponentially weighted average, assume we have a sequence of numerical data, i.e., $\{s_1, s_2, \ldots s_n\}$. Using a weighted average, we can have a sequence of transformed numerical data, i.e., $\{s'_1, s'_2, \ldots s'_n\}$ and each s' at position t is calculated as $s'_t = \beta s'_{t-1} + (1-\beta)s_t$.

For example, three sequential data in the transformed sequence with a weighted average will be written as follows.

$$s'_t = \beta s'_{t-1} + (1-\beta)s_t$$

$$s'_{t-1} = \beta s'_{t-2} + (1-\beta)s_{t-1}$$

$$s'_{t-2} = \beta s'_{t-3} + (1-\beta)s_{t-2}$$

Therefore, by combining these series and applying some mathematical simplification to them, we have the following:

$$s'_t = \ldots + \beta\beta(1-\beta)s_{t-2} + \beta(1-\beta)s_{t-1} + (1-\beta)s_t$$

As the data gets older in the series, its impact on the equation gets smaller because more numbers of βs multiplied. Note that the previous sentence is true only for β between 0 and 1.

That is enough to understand the impact of exponentially weighted average, and now we can switch back to the SGD with the momentum algorithm.

SGD with Momentum

SGD with momentum uses an exponentially weighted average to create a *velocity*. To understand velocity, assume we are rolling a ball inside the bowl shape function, and we intend to get the ball into global minima in Figure 10-14. The derivative of the ball specifies the acceleration of the moving ball, and momentums add to the velocity of the moving ball (v or *retained gradient*).

Momentum (η or *momentum coefficient*) is used to increase the velocity of the optimization ball, and thus, it will be fast enough to jump out of the local minima and move toward global minima. In

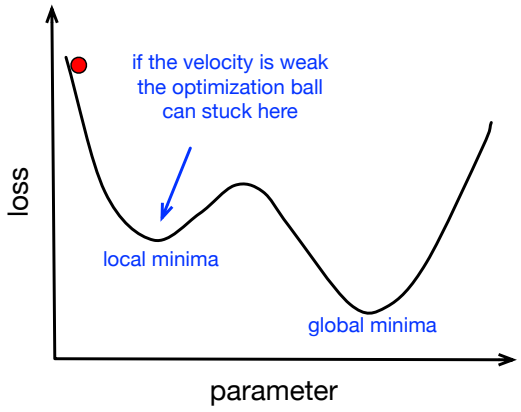


Figure 10-14: A ball with low velocity stuck in the local minima, but having high enough velocity can helps it jump out of local minima and move toward global minima.

other words, momentum reduces the oscillation of the optimizer algorithm, and thus, it can reach the minima faster. The SGD with momentum algorithm is written as follows:

$v_t = 0$ `# velocity`

$\eta = 0.9$ `# momentum coefficient`

$\alpha = 0.01$ `# learning rate`

$while\ (w_t\ \ not\ \ converged)$ {`# e.g., converged in this context: loss < 0.1`

$v_{t+1} = v_t . \eta + \nabla G(w_t)$

$w_{t+1} = w_t - \alpha . v_{t+1}$

$v_t = v_{t+1}$

$t = t + 1$

}

Don't be afraid to see these equations inside an algorithm. They are very easy to understand. $v_{t+1} = v_t . \eta + \alpha . \nabla G(w_t)$, means:

$next\_velocity = current\_velocity \times momentum + learning\_rate \times gradient$

Respectively $w_{t+1} = w_t - \alpha . v_{t+1}$, means:

$next\_weight = current\_weight - learning\ rate \times next\_velocity$

You can see that current velocity, momentum, and learning rate are all hyperparameters, and they have been given by the user of the algorithm. Nevertheless, they usually have common predefined values, and the only hassle for the user is choosing the optimization method and calling it. For example, at the time of writing this chapter in 2021, in Keras[6], we only need to write the following for the model, and the SGD is the SGD with momentum.

```
model.compile(loss='categorical_crossentropy', optimizer='SGD')
```

if we need to give some parameter value, we use the following code in Tensorflow[7]:

```
tf.keras.optimizers.SGD(
    learning_rate=0.01, momentum=0.0, nesterov=False,
name='SGD', **kwargs)
```

In pytorch[8], it is written as follows:

```
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
```

To summarize our explanation here, momentum increases the convergence speed toward global minima and also increases the chance of reaching local minima by introducing a velocity component.

---

[6] https://keras.io

[7] https://www.tensorflow.org

[8] https://pytorch.org

Nesterov Momentum

Using momentum significantly improves the convergence speed of SGD. However, using classical SGD with momentum, the gradient is always moving toward the correct direction, but momentum may not necessarily move in the correct direction. Nesterov Accelerated Gradient (NAG) or Nesterov momentum [Nesterov '83] improves the next step, and if the momentum goes in the wrong direction, then the gradient can go in the correct direction.

Take a look at Figure 10-15 to better understand the differences between the classical and Nesterov approach. In both examples, the momentum moves in the wrong direction, but in the Nesterov momentum, the gradient is started after the momentum, and thus, it moves more toward the correct direction. Therefore, the next data point is selected closer to the minima in the right figure.

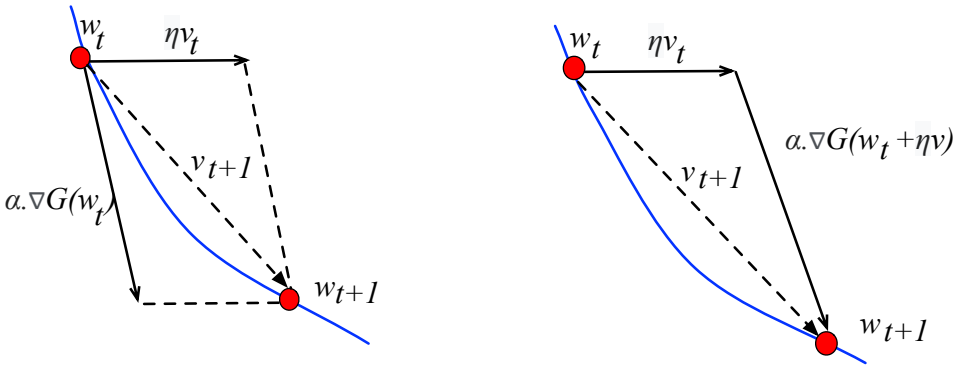


Figure 10-15: (left) Classical momentum for calculating the next parameter, (right) Nesterov momentum is used to calculate the next parameter on cost function. The blue line presents a small part of the cost function, and each new w is moving toward minima.

While using pure SGD, the next point on the cost function will be determined by the following equation:

$$w_{t+1} = w_t - \alpha \,.\, \nabla\, G(w_t)$$

By using SGD with the momentum, the next point on the cost function will be determined as:

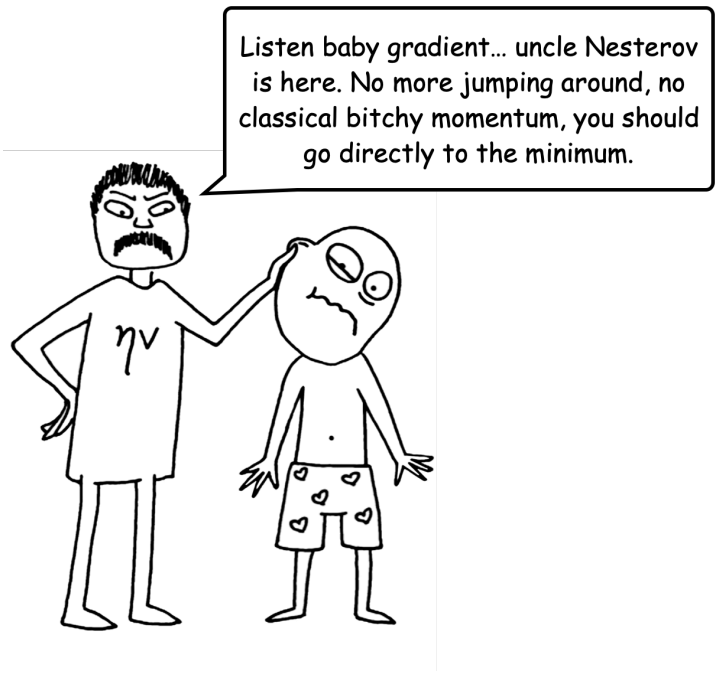
$$w_{t+1} = w_t + v_{t+1} \,.\, \eta - \alpha \,.\, \nabla\, G(w_t)$$

The Nesterov momentum calculates velocity and weight differently from classical momentum, and it uses the following equations to calculate them.

$$v_{t+1} = v_t \,.\, \eta - \alpha \,.\, \nabla\, G(w_t + \eta\, v_t)$$

$$w_{t+1} = w_t + v_{t+1} \,.\, \eta$$

Based on Figure 10-15 [Sutskever '13], we can see that by using Nesterov momentum, if the momentum goes in the wrong direction, the gradient has the chance to move in the correct direction, which results in faster convergence. It is a small difference, but it increases the



convergence speed, and in some cases, Nesterov momentum performs better than classical momentum.

Adagrad

The neural network optimization deals with a cost function that is a non-convex shape because the neural network includes many weights and biases (model parameters or dimensions). Until now, we have assumed that the learning rate is constant. A constant learning rate, a hyperparameter given by the user, could not be useful for all scenarios and all types of networks.

Experiments show that the learning rate in a multi-dimensional dataset (or features) in some dimensions (or features) is changing very fast, and in some dimensions (or features), it is changing very slowly. For example, usually, weights change more frequently than biases, which change less frequently. Therefore, employing separate learning rates for different types of parameters (e.g., one for weights and another for biases) can facilitate a more efficient approach to reaching the minimum of the cost function. It means a neural network could benefit from a dynamically changing learning rate that adapts its change pace to each feature.

One of the practical algorithms that introduce a dynamically changing learning rate is Adagrad [Duchi '11]. The Adagrad algorithm adaptively scales the learning rate for each parameter. It adapts the learning rate by scaling them inversely proportional to the square root of the sum of their historical/previous squared values.

We are sure you did not understand the previous sentence, so we explained it with the equation, which is easier to understand. Recall that SGD uses the following equation to estimate the next weight. $w_{t+1} = w_t - \alpha \,.\, \nabla\, G(w_t)$. The learning rate (shown as α) is constant in all epochs of traditional SGD, but the Adagrad algorithm uses a *dynamic learning rate* that can change based on the magnitude of the target parameter gradients until the current time.

Assuming α' is a learning rate for Adagrad, we can re-write the previous equation as $w_{t+1} = w_t - \alpha'_t \cdot \nabla G(w_t)$. Here α'_t is specified based on the previous epoch and previous weights. Therefore, we add a subscript as *t,* which presents the learning rate in t iteration (epoch). The value for α'_t will be calculated as follows:

$$\alpha'_t = \frac{\eta}{\sqrt{G_t + \epsilon}}$$

In this equation, ϵ is used to be sure the denominator will never be zero, and it is recommended to set it to 10^{-8}, which is an extremely small number. η is a constant value, and we can use a small value, such as 0.001. G_t is the variable that is performing the magic. It is the sum of squares of all previous gradients up to the current t, as it is calculated by using the following equation.

$$G_t = \sum_{i=1}^{t} \nabla G(w_i)^2$$

In other words, G_t enables Adagrad to reduce the impact of parameters with large gradients (features that are frequent in the dataset) and increases the impact of parameters with low gradients (features that are not frequent in the dataset). Because the denominator part of the equation (G_t) will be large for small gradients and small for large gradients. Recall that in primary school, we learned that having a large denominator makes the number small and vice versa.

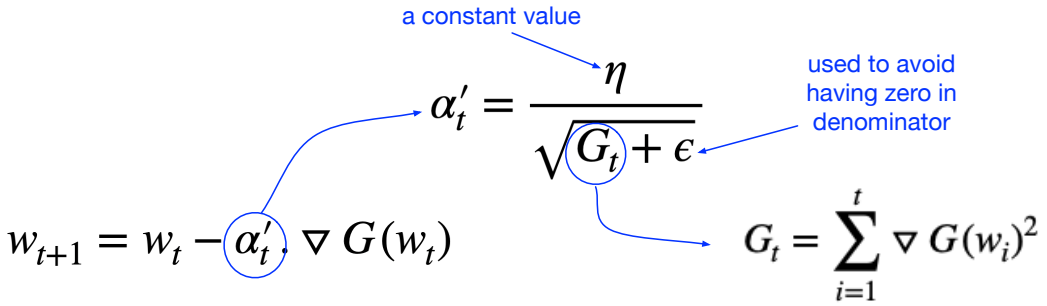


Figure 10-16: Adagrad parameter calculation equations.

To get an overview, take a look at Figure 10-16, which shows the described equation in this figure. If you still don't get the point, let's substitute the equation with some number and see the result. Assume we have a frequent feature (f_1) and its $G_t = 3.46$ and another less frequent feature (f_2) that its $G_t = 0.16$.

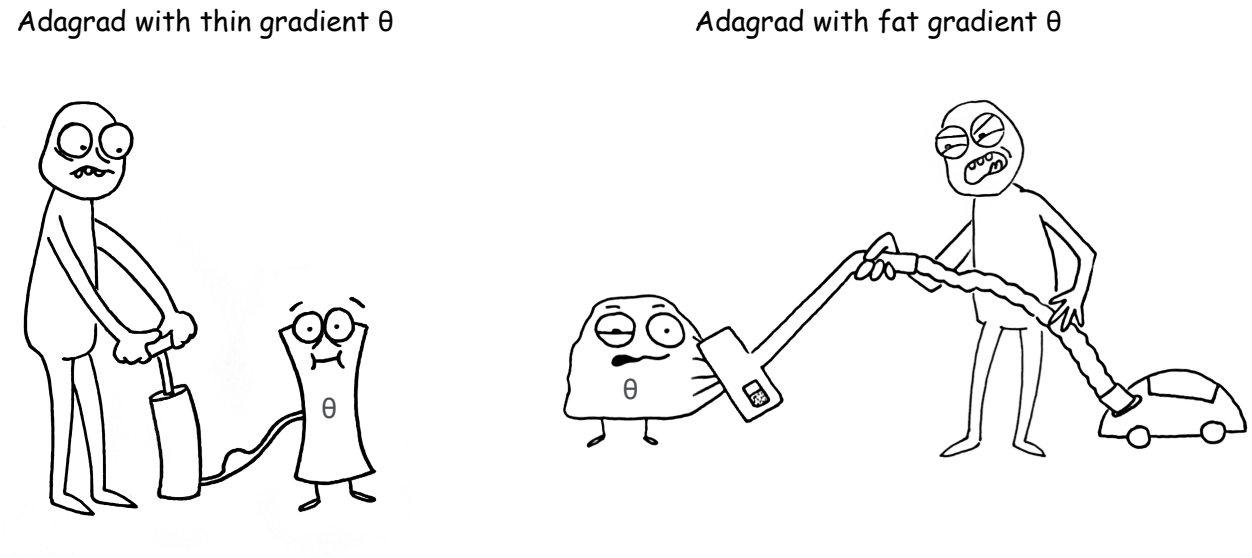
$$\alpha'_t(f_1) = \frac{0.01}{\sqrt{3.46 + 10^{-8}}} = \frac{0.01}{1.86} = 0.005$$

and

$$\alpha'_t(f_2) = \frac{0.01}{\sqrt{0.16 + 10^{-8}}} = \frac{0.01}{0.4} = 0.025$$

By comparing 0.005 and 0.025 we can realize the magic that Adagrad does is based on the sum of previous gradients of a parameter.

Adagrad works well when the dataset is sparse, which means some features include lots of zeros, and those features do not frequently appear in the dataset. For example, constructing a bag-of-words (Chapter 6) creates a sparse dataset of terms, and to process a bag-of-words with a deep learning algorithm, it is recommended to use Adagard as an optimizer.



To summarize this algorithm, make some space in your brain about Adagrad and write the following there: Adagrad reduces the focus on the parameter that is always happening by decreasing their learning rate and allows parameters that with of zeros (sparse features) to have a larger learning rate.

There is an optimized version of Adagrad, called Adadelta [Zeiler '12] and it restricts the number of past gradients to a specific window size, which is given by the user as a hyperparameter. It is late night now, and the author doesn't have the energy to explain Adadelta.

RMSprop

RMSprop (Root Mean Square propagation) is an optimization algorithm and introduced by Hinton [Hinton '12], one of the pioneers of deep learning in his online course[9]. Similar to Adagrad, RMSprop focuses on mitigating the challenge of having gradients of different sizes (too large or too small). RMSprop builds on top of the *Rprop* [Riedmiller '93], in which Rprop uses (i) the sign of gradient and (ii) adapting the step size to each gradient. To understand the RMSprop, we need to understand the Rprop first.

Rprop first checks the sign of two consecutive gradients. If the sign has been changed (similar to the two red dots in Figure 10-13), it means the jump was too large, and thus, the minima have

[9] Interestingly they did not publish any scientific paper about RMSprop

been passed and not reached. In the next step, it decreases the jump size by multiplying it by a number less than 1, e.g. $\eta^- = 0.5$. If the sign has not been changed in two consecutive gradients, the jump is correct, and we are moving in the correct direction. Therefore, the step size will increase by multiplying it by a number larger than 1, e.g., $\eta^+ = 1.2$. For example, if $\nabla G(w_t) = -1$ and $\nabla G(w_{t+1}) = -2$, it means the algorithm is moving in the correct direction, and step size can increase, and it calculates the next weight as follows: $w_{t+1} = w_t + \eta^+ \nabla G(w_t) = 1.2$.

Rprop is very good when we have a small dataset. As soon as the dataset gets large, we should go for mini batch Gradient Descent (mini BGD), and Rprop cannot handle mini BGD properly. For example, we have five mini-batches, whose gradients are as follows: -0.4, -0.3, -0.25, -0.2, -0.14, -0.1, 0.7, and 0.9. Here, Rprop increases the weight six times and decreases it only once from (-0.1 to 0.7). This means that instead of RProp coefficients canceling each other, the weights grow larger despite insignificant changes in gradient.

Instead, RMSProp performs a small improvement on the Rprop to resolve it. RMSProp benefits from using Rprop decision based on sign, but additionally, it can handle the issue existing in mini batches. RMSprop is similar to Adagard with slight differences. While using Adagrad, the next weight will be determined by the following equation:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{G_t + \epsilon}} . \nabla G(w_t)$$

While using RMSprop, the next weight will be determined by the following equation:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{E[G_t^2] + \epsilon}} \nabla G(w_t)$$

In this equation $E[G_t^2]$ presents the squared of exponentially weighted average gradient and is calculated as follows: $E[G_t^2] = \beta E(G_{t-1}^2) + (1 - \beta) \nabla G(w_t)^2$.

β is the exponentially weighted average parameter that is assigned by the user, and it is usually 0.9. $\nabla G(w_t)$ (the gradient of the cost function with respect to w_t), $\eta = 0.01$ and $\epsilon = 10^{-8}$ are similar to Adam's equation and hyperparameters, but they usually have default values.

In summary, RMSprop divides the learning rate by an exponentially decaying average of squared gradient and benefits from the sign-based decision of Rprop.

Adam

Adam (adaptive Gradient Descent) [Kingma '14] is another popular optimization algorithm that combines the advantages of both SGD with momentum and RMSprop. It operates by taking large jumps at the beginning, and as the slope gets closer to zero, it starts to take smaller jumps.

Adam introduces two β parameters (β_1, β_2), which are used to *control the decay rate* of exponentially weighted averages of the (i) gradient (used in momentum) and (ii) squared gradients (used in RMSProp).

The following algorithm describes Adam. While reading this algorithm, note the # sign is used for commenting and is written in blue.

$\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

$while\ (w_t \quad not \quad converged)\ \{$

$g_t = \nabla G(w_t)$

$m_t = \beta_1 . m_{t-1} + (1 - \beta_1) g_t$ `#first moment estimate (momentum)`

$v_t = \beta_2 . v_{t-1} + (1 - \beta_2) g_t^2$ `#second moment estimate (RMSProp)`

$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$ `# corrected first moment estimate`

$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$ `# corrected second moment estimate`

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

$\}$

In this algorithm, m_t presents the first moment estimate (to support the functionality of momentum) and v_t presents the second moment estimate (to support the functionality of RMSProp). β_1^t and β_2^t mean β_1, β_2 to the power of t. Other hyperparameter values were recommended by the authors of the Adam algorithm. We can see that a squared gradient is used to scale the learning rate like RMSprop, and a moving average of the gradient is used instead of the gradient itself, as it is done in SGD with momentum.

There is not much to explain about Adam, and it seems that in the golden era of Gradient optimizer algorithms (approximately from 2012 to 2015), authors of these algorithms, with small changes, made a big improvement in algorithm accuracy.

Optimizers Summary

Congratulations! Now, you are an optimizer expert and know these mathematical definitions. Since this section was very exciting, has no theoretical discussion and mathematics, and is full of real-world examples, let's do a summary before you completely fall asleep.

SGD will change the model parameter per sample data point, which is too frequent, and this could cause a lot of oscillation on the optimizer. To handle this problem, we can use mini-BGD, which updates mode parameters after a few samples. Nevertheless, still, a few examples could change the model parameters, and this leads to being stuck in a local minima. Therefore, we use classical momentum to reduce this noise.

Since the noise is reduced with classical momentum, the optimizer could make a wrong move and get away from minima, and the Nesterov momentum can mitigate this challenge by introducing an acceleration term into the model parameter. In simple words, if momentum makes the optimizer too fast, the Nesterov momentum slows it down. With these approaches, the

learning rate for all model parameters is the same, but Adagrad and RMSprop use a history of the gradient to find a different learning rate for each model parameter.

Adagrad tunes the learning rate per model parameter, which makes it a good choice for sparse datasets. RMSProp also tunes the learning rate per model parameter, which makes it a good choice for noisy data and non-stationary data (check Chapter 8 to recall the meaning of non-stationary time series).

RMSprop and Adagrad do not update the momentum, Adam updates the momentum (in addition to the learning rate) for each model parameter as well[10].

Only 150 more optimizers remaining that you can learn on your own [Schmidt '20]. All jokes aside, it was the madness of making better optimizers, and many students are struggling to make impactful optimizers. For example, add a decaying coefficient on the momentum parameter as well, called demon (decaying momentum) DemonAdam [Chen '19], or develop an automatic tuning approach for the momentum, such as YellowFin [Zhang '17]. We skip to explain them, and you really don't need to learn them unless you want to research optimizers.

NOTE:

* There is no best optimizer that can perform perfectly in all problems. However, some background knowledge might be helpful to decide about the optimizer. There are some platforms used to benchmark different optimizers, such as DeepOBS [Schneider '19], which is open source[11], and you can install and use them to decide on the optimizer based on your dataset.
* If you have a mathematic or algorithmic background and love this topic, invest in making a good optimizer that is not based on Gradient Descent; think about it twice. To date, many optimization algorithms, especially genetic algorithms, have been proposed, and none have been as good as the Gradient Descent ones. You can search online and see that an optimizer algorithm has been developed for every single animal or insect on this planet. However, gradient descent algorithms are performing better than any other algorithm, but there might be an opportunity to explore and identify a new one. At the time of writing this part in 2021, genetic algorithms lost badly the optimization war to gradient descent ones.
* Convex optimization problems are easier to solve than non-convex problems. If a concave problem can be reformulated into an equivalent convex problem, it may enable the use of more efficient convex optimization algorithms.

[10] Alec Radford has an amazing visualization of some of these optimizers in a gif file and we highly recommend you to check his animation, here: https://imgur.com/a/Hqolp.

[11] https://deepobs.readthedocs.io/en/stable/index.html

Backpropagation

The Backpropagation (Backprop) algorithm was introduced in 1960 [Kelley '60, Bryson '62], and later, in 1986, it was generalized and popularized by [Rumelhart '86]. At the time of writing this part, it is the backbone of deep learning. It is not a complex algorithm to understand, and we will try our best to explain it clearly here, so do not worry if you don't understand it the first time. Try reading this section more than once. Backpropagation is nothing more than applying a chain rule on the neural network neurons to change weights to reduce the loss score.

We have explained that the objective of the cost function is to change weight and biases toward improving the output's accuracy (reducing the loss score) and thus making a better prediction. How does the neural network change weight and biases to improve its accuracy? Starting from the output, it *goes back* to the network after each epoch and then reconfigures weight and biases to reduce the loss score.

Before beginning the Backpropagation explanation, take a look at Figure 10-17 (a). We did not add hidden layers for the sake of simplicity. In this Figure, we have two input neurons and one output neuron. The output of this network provides a loss score, i.e., error. The next epoch should reduce this error by changing the weights w_1 and w_2 values. Now, the question is, which one of these input nodes contributes to the error: input A or input B, or both?

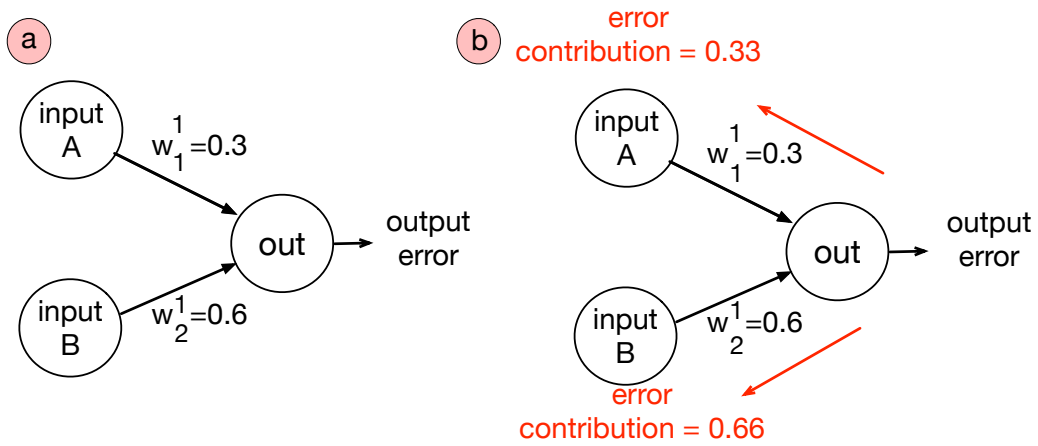


Figure 10-17: (a) Very simple network with two input and one output. (b) Based on the weight of each node in the previous layer, the contribution of each node to the output has been identified.

We could say both *A* and *B* contribute to the error, but the amount of their contributions is estimated based on their weights. In other words, neurons' contributions to the error will be measured by their weights. For example, in Figure 10-17 (a), the contribution of input neuron *A*

to the error can be calculated as $\frac{0.3}{0.3+0.6} = 0.33$ and the contribution of input neuron B will be calculated as $\frac{0.6}{0.3+0.6} = 0.66$.

Figure 10-17 (b) presents the contribution of each neuron to the error (based on their weight) in red color. We can see in Figure 10-17 (a) that we have used weights to forward the signal to the output layer, this process is referred to as the *forward step*. Next, after the output error has been identified, a neural network uses the weight to *propagate* back the signal from the output layer to the input layer, as shown in Figure 10-17 (b). This process is called *Backpropagation* or *Backprop*. To summarize, the Backprop algorithm splits the error of output neurons across the previous neurons proportional to the incoming weights to this neuron.

Note that while the Backpropagation algorithm is propagating back the error, input values do not change, and the only thing that will be changed is the weights used in the hidden layers. In this example, for the sake of simplicity, we show only two neurons and hidden layers are not separated from input layers. The network of Figure 10-17 is too simple, and we can extend the described approach to hidden layers as well.

The Backpropagation algorithm assumes the *error in a neuron (hidden or output) is the sum of the splits errors in all other previous nodes linked to this neuron*. For example, the blue lines in Figure 10-18 visualize the propagation of the error from the blue output neuron to the first layer neurons. If there is more than one hidden layer (as the case in Figure 10-18), *the errors of hidden layers split again proportional across all previous links between input and hidden layers connected to that particular neuron*.

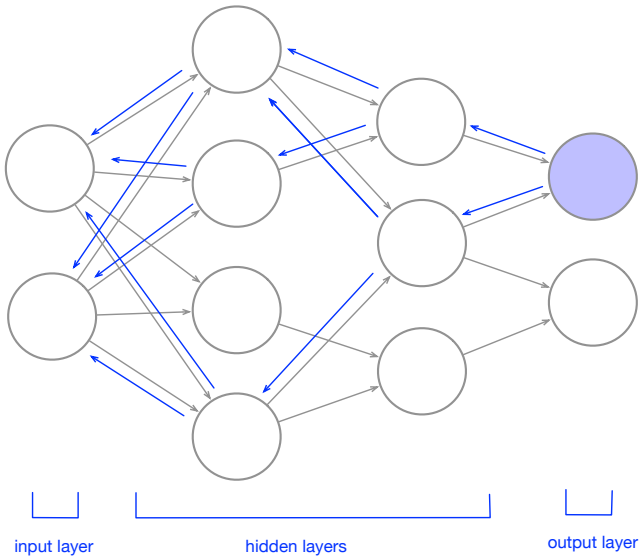


Figure 10-18: Propagation of error from the blue output neuron to the input neurons is shown in blue color.

The weights of all links are known, and with the same mathematical approach, we can calculate the contribution of each input neuron to the output's neuron error. If you believe it helps you to understand the Backpropagation better, take some pen and paper, assign random weights to each node and a random error to the end node, then try to calculate each node's contribution to the error until you reach an input layer.

In summary, the Backpropagation, specifies the contribution of each neuron to the error, and in the next epoch, weights will be changed to reduce the output error.

To implement Backpropagation, matrix multiplication is used[12]. Therefore, the errors of layer n could be written as a matrix (e_n), which is the multiplication of the transpose of weight matrix (w^T) time matrix of errors for the next layer (e_{n+1}), as follows: $e_n = w_{n+1}^T \cdot e_{n+1}$

For example, we have a simple network like Figure 10-19, the output layer errors are $e\ out_1$, $e\ out_2$, which are specidied at the end of each epoch, and thus, the errors of the hidden layer (e_h) can be calculated as follows:

$$e_h = \begin{bmatrix} e\ h1 \\ e\ h2 \end{bmatrix} = \begin{bmatrix} w_1^{(2)} & w_2^{(2)} \\ w_3^{(2)} & w_4^{(2)} \end{bmatrix} \times \begin{bmatrix} e\ out_1 \\ e\ out_2 \end{bmatrix}$$

Respectively, errors of the input (e_{in}) layer are calculated as follows:

$$e_{in} = \begin{bmatrix} e\ in1 \\ e\ in2 \end{bmatrix} = \begin{bmatrix} w_1^{(1)} & w_2^{(1)} \\ w_3^{(1)} & w_4^{(1)} \end{bmatrix} \times \begin{bmatrix} e\ h_1 \\ e\ h_2 \end{bmatrix}$$

We have learned that the Backpropagation error is presented and calculated as matrix multiplication. Now, that big fat question arises again: How do we update weights to reduce the error (loss score)?

By looking at Figure 10-18, we can see many neurons are connected to a single output. Therefore, it is not a trivial mathematical process to identify a better weight assignment for the next epoch. We need to go deeper into mathematical explanation, but first, we should review some conventions and concepts.

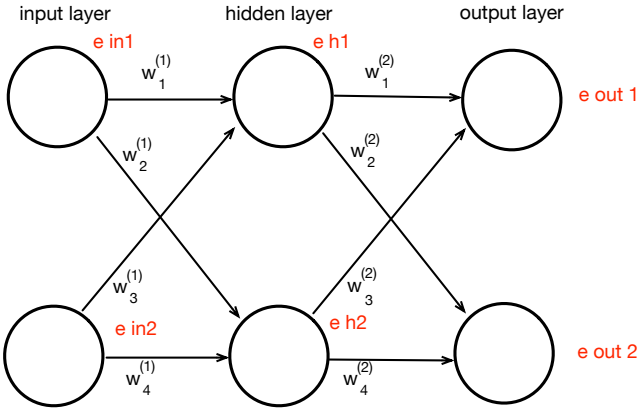


Figure 10-19: A simple neural network with one hidden layer.

The output of the neuron in the first layer is written as $z = wx + b$. Here x is the input, but in the other neurons after the input layer, we do not have x. Instead of input neuron data, we only have the output of the activation function. Therefore, we refer to the neuron output as z. In other words, $z^{(l)}$ is defined by weight and biases at level l for the given input that comes from the previous layer.

In the context of a neural network, the predicted value presents the *activation function result* in layer l of neuron number j, which is presented as $a_j^{(l)}$. We can generalize it (removing layer and node information) and write it as $a = \sigma(z)$, here, z *presents the output of the previous layer,* σ presents the activation function, and thus we can say $z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}$. For example, if we

[12] Matrix multiplication is also implemented with inner loops, but this is very resource-intensive. There are other approaches that make it more efficient to implement it, and if somebody discovers a new matrix multiplication the resources used by GPU will be drastically reduced.

refer to the very last layer of the network as L, and for the last layer we have $z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$, we know that $a^{(L)} = \sigma(z^{(L)})$.

We have explained that *error = predicted — actual*. Assuming the actual value of neuron j is presented as y_j, and the error is presented as E we can write the following equation for an error of neuron j and the last layer (output layer) L: $E_j^{(L)} = a_j^{(L)} - y_j$ also note that $\hat{y}_j = a_j^{(L)}$. We consider them as a matrix (generalizing it), and thus remove the jth parameter, and end up with $E^{(L)} = a^{(L)} - y$, to calculate the error matrix in the last layer L.

We hope you recall from Chapter 8 the partial derivative concept. To recall that, check Figure 10-20, which presents a concept called *Computation Graph*. It is used to visualize the chain rule with a partial derivative. Assume we have $y = g(f(x))$, and by using the chain rule, we can present the partial derivate of y over x, as follows:

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial g} \times \frac{\partial g}{\partial f} \times \frac{\partial f}{\partial x}$$

Now that we have learned enough conventions and concepts let's get back to what we need to understand for Backpropagation. We should understand *ho*w much loss score (or error) changes as weight and biases change. Or how sensitive is the loss score to changes in w and b?

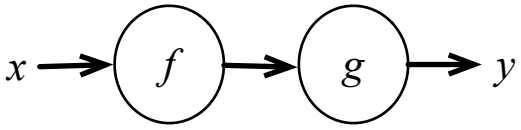


Figure 10-20: a very simple computation graph for a nested function $y = g(f(x))$

We can write the error equation as a partial derivative (check Chapter 8 to recall partial derivative) of the error to the weight at layer l, i.e., $\frac{\partial E}{\partial w^{(l)}}$. In other words, we are calculating the error with respect to the weights at layer l. Based on the mathematical notation we have described above, and by using the "chain rule" of derivative (again, say our hello to Chapter 8), we can rewrite $\frac{\partial E}{\partial w^{(l)}}$ with the partial derivative as follows:

$$\frac{\partial E}{\partial w^{(l)}} = \frac{\partial E}{\partial a^{(l)}} \times \frac{\partial a^{(l)}}{\partial z^{(l)}} \times \frac{\partial z^{(l)}}{\partial w^{(l)}}$$

If you take a look at Figure 10-20, this equation seems easy to understand. It just applies chain rules and simply uses z and a to find how sensitive the loss score (E) is to changes in weight (w).

However, in this equation, we only considered weights; for biases, we can use the same equation as follows:

$$\frac{\partial E}{\partial b^{(l)}} = \frac{\partial E}{\partial a^{(l)}} \times \frac{\partial a^{(l)}}{\partial z^{(l)}} \times \frac{\partial z^{(l)}}{\partial b^{(l)}}$$

By using a Hadamard product (it is denoted with ⊙ sign and you can check Chapter 7 to recall it), these two equations will be written for one single layer, and we can do this for the entire network. These equations use gradients to go back through the network and adjust weight and biases to minimize the output error at the output layer. By output error, we mean a vector of errors (or loss score).

Assuming our network has L layers and the training dataset has m data points, the following pseudocode describes the Backpropagation algorithm. While reading this algorithm, note that the # sign is used for commenting and is written in blue.

#------Step 1 (initialing parameters)-----

This step initializes the learning rate, weights, and biases, and also, the threshold for stopping criteria will be specified.

initialize w, b, α & stop_criteria # α is learning rate

for i = 1 to m { # m is the number of data points in the training set

#------Step 2 (forward propagation)------

This step computes the activation for all layers.

a is the predicted value extracted from the activation function and x is the input variable.

for j = 1 to L { # L is the number of layers

if (j=1) then $a^{(1)} = x^{(i)}$ # Since the activation function does not exist at the first layer, which is the input layer, at this layer $z = wx + b$ and $a^{(1)}$ is the input.

else $a^{(j)} = \sigma(z^j)$ # Now there is no x available (because we are talking about layer 2 and other layers) for each layer, the algorithm computes z *and* a, recall that $z^l = w^l a^{l-1} + b^l$

}

------Step 3 (Calculate error vectors)------

$E^{(L)} = a^{(L)} - y^{(i)}$ # In the last layer, we know y, which is the actual output.

for k = (L-1) to 2 {# this loop computes other error vectors $(E^{(L-1)}, E^{(L-2)}, \ldots E^{(2)})$ for other layers (except the last one) until it reaches layer 2, there is no error for layer 1, because the input layer does not have an error.

$E^{(k)} = (w^{(k+1)})^T \times E^{k+1} \odot \sigma'(z^{(k)})$ # $\odot$ presents a Hadamard product of two matrices. $(w^{(k+1)})^T$ is the transpose of weights for the next layer (to prepare them for matrix operation, they are transposed). $\sigma'(z^{(k)})$ is a vector of derivatives of activation function results at layer k. $\odot \sigma'(z^{(k)})$ moves the error backward through the activation function in layer k.

}

#------ Step 4 (compute gradients and update weight and biases) -----

This step computes partial derivation of the gradient with respect to weight and biases.

for all (w and b) {

$w_{new}^{(l)} = w_{old}^{(l)} - \alpha \frac{\partial E}{\partial w_{old}^{(l)}}$ # $\frac{\partial E}{\partial w_{old}^{(l)}}$ is the partial gradient of the cost function result (loss score) with respect to w. At the first run, we assume it

```
        is 0. This gradient will be acquired using the chained rule of partial
        derivative.
```

$$b_{new}^{(l)} = b_{old}^{(l)} - \alpha \frac{\partial E}{\partial b_{old}^{(l)}}$$ # $\frac{\partial E}{\partial b_{old}^{(l)}}$ is a partial gradient of the cost function result (loss score) result with respect to b. At the first run we assume it is 0.

```
    }
} # closes 'for i = 1 to m' loop
```

We can see that the partial gradients with respect to weights and biases are computed using the chain rule. The loss score is calculated by using a cost function. For example, a cost function could be binary cross entropy, $\hat{y}$ is the predicted and y is the actual value; the loss score after one pass on the network will be calculated as follows:

$$Loss = -[\, y\, log(\hat{y}) + (1-y) log(1-\hat{y})]$$

For the sake of simplicity, we did not incorporate regularization in step 4, but regularization can be involved as well.

We can summarize that *a neural network is a chain of differentiable functions*. Differentiable functions mean we can get a derivative of it, and the parameters of these functions can be trained. The process of training involves computing the gradient of network parameters (for each batch) with respect to the cost value (for each batch). Therefore, we can say a neural network is a combination of nested functions, $F(X) = f_1(f_2(f_3(x)))$, each of the f functions could be a vector function $f_l(Z) = a(x\,w + b)$, a is the activation function, l is the layer index, and it could span from 1 to any number of layers. More about this will be explained later in the Backpropagation section.

Forward and Backward Pass Example

Some might better understand the forward and backward pass on the network with mathematical examples. If you find it too complex, feel free to skip this section.

Consider a forward pass starting from x_0 as input and three layers with activation function σ_R we can formalize the forward pass f follows $f(x_0; w) = \sigma_R(w_3 \,.\, \sigma_R(w_2 \,.\, \sigma_R(w_1 x_0 + b_1) + b_2) + b_3)$.

To perform the backward pass and compute the gradients, the Backpropagation algorithm uses the chain rule of differentiation. It starts by computing the gradient of the loss with respect to the output of the network:

$\nabla f = \nabla \sigma_R(w_3 \,.\, \sigma_R(w_2 \,.\, \sigma_R(w_1 x_0 + b_1) + b_2) + b_3)$ where ∇ denotes the gradient and σ_R is the activation function.

Next, it applies the chain rule to compute the gradient of the loss with respect to the parameters of the last layer:

$\nabla w_3 = (\nabla f) \,.\, \sigma_R'(w_3 \,.\, \sigma_R(w_2 \,.\, \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) \,.\, \sigma_R(w_2 \,.\, \sigma_R(w_1 x_0 + b_1) + b_2)$

$$\nabla b_3 = (\nabla f) . \sigma_R'(w_3 . \sigma_R(w_2 . \sigma_R(w_1 x_0 + b_1) + b_2) + b_3)$$

Here σ_R' is the derivative of the activation function. Next, it uses the chain rule again to compute the gradient of the loss with respect to the parameters of the second layer:

$$\nabla w_2 = (\nabla f) . \sigma_R'(w_3 . \sigma_R(w_2 . \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) . w_3 . \sigma_R'(w_2 . \sigma_R(w_1 x_0 + b_1) + b_2) . \sigma_R(w_1 x_0 + b1)$$

$$\nabla b_2 = (\nabla f) . \sigma_R'(w_3 . \sigma_R(w_2 . \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) . w_3 . \sigma_R'(w_2 . \sigma_R(w_1 x_0 + b_1) + b_2)$$

Finally, it computes the gradient of the loss with respect to the parameters of the first layer:

$$\nabla w_1 = (\nabla f) . \sigma_R'(w_3 . \sigma_R(w_2 . \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) . w_3 . \sigma_R'(w_2 . \sigma_R(w_1 x_0 + b_1) + b_2) . w_2 . \sigma_R'(w_1 x_0 + b1) . x_0$$

$$\nabla b_1 = (\nabla f) . \sigma_R'(w_3 . \sigma_R(w_2 . \sigma_R(w_1 x_0 + b_1) + b_2) + b_3) . w_3 . \sigma_R'(w_2 . \sigma_R(w_1 x_0 + b_1) + b_2) . w_2 . \sigma_R'(w_1 x_0 + b1)$$

Regularization in Neural Network

A successful machine learning algorithm should avoid overfitting. We can think of overfitting as just memorizing the data (not learning) and matching everything to the training dataset. Therefore, if new data arrives in the system (test data) that does not match the existing data, the algorithm can not determine a label for it.

As we have described in Chapter 8, to avoid overfitting, we use regularization. We describe popular methods that used regularization in ANN, but in addition, to resolve overfitting, they can handle vanishing and exploding gradient problems as well.

Vanishing and Exploding Gradients

We have learned that a neural network operates in three steps. First, it begins from the input layer and goes through hidden layers to reach the output. This process is known as forward pass. Second, the loss function compares the output (e.g., prediction) result with the actual label and measures the error. In simple words, the output of a loss function is an error value. In the third step, the neural network uses the error value and backpropagation algorithm to calculate the gradient for each neuron in the network. Gradients (a vector of partial derviates with respect to weights and biases) are used by the network to adjust its weights and biases to reduce error.

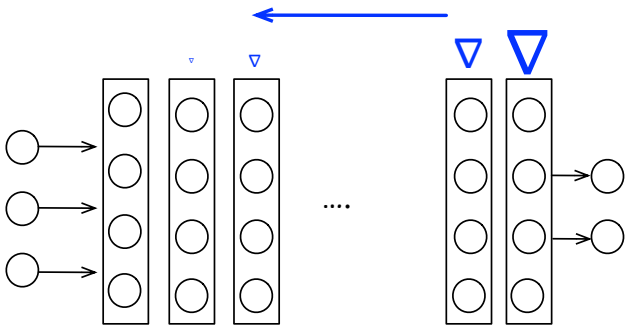


Figure 10-21: The gradient (blue triangle) is getting smaller and smaller in the back propagation that at some points get useless and weights do not change at all.

As the network moves from the output toward the input layer, the gradient gets smaller by the chain rule, and thus, the weight adjustment gets smaller. In a more technical sense, the gradient is exponentially shrinking as the backpropagation moves toward the input neuron. Figure 10-21 visualizes this phenomenon. A high gradient reveals a big adjustment, while a small gradient indicates a minor adjustment in weight.

When a network is deep (with many hidden layers), the gradient gets smaller and smaller until it vanishes, and thus, weights on the layers close to the input layer never get updated. For example, if the gradient of the last layer close to the output layer is 0.5 or less than one, as we go deep in the network, it gets smaller and smaller because it gets multiplied by some smaller number. For example, it will be $0.5 \times 0.4 \times 0.1 = 0.02$. Then the weight in the input layer will be

$w_{new}^{(l)} = w_{old}^{(l)} - \alpha \times 0.02$. Since we know a learning rate is also a small number, such as 0.01, we have $w_{new}^{(l)} = w_{old}^{(l)} - 0.0002$, which means the new weight has very insignificant differences from the old weight, this is known as *vanishing gradient*. The same thing could happen with gradients larger than one, and thus, it makes a very big gradient that does not let SGD get close to minima, which is the *exploding gradient.*

In summary, the vanishing gradient refers to a problem in deep neural networks; the gradient in backpropagation is getting too small that the weight doesn't get updated at all. This is not good because some weights, especially weights close to the input layer, didn't change by backpropagation, and only weights close to output layers get updated.

There are approaches that do not solve this problem but try to mitigate them. One approach is the use of ReLU or Leaky-ReLU as activation functions and avoiding using hyperbolic tangent as activation function, which is prone to the vanishing gradient. In the following, we briefly list common approaches.

Weight Initialization

There are two approaches that use a more subtle weight initialization and do not perform the weight initialization completely randomly at the beginning. These approaches are known as Xavier [Glorot '10] and He [He '15].

Xavier or Gorlot initialization uses random weights with a normal distribution and not completely random numbers. In particular, uses random weights drawn from a normal distribution with a mean of 0 and a variance of $2/(n_{in} + n_{out})$ where n_{in} presents the number of input neurons and n_{out} the number of output neurons.

He initialization (a.k.a Kaiming initialization) proposes a weight initialization for non-linear activation functions such as ReLU and Leaky ReLU while considering the non-linearity of non-linear activation functions. Weights are initialized to have a normal distribution, with zero mean and variance of $2/n_{in}$, and biases are initialized to zero.

Gradient Clipping

Another approach is Gradient clipping, which cuts off gradients before they reach a predefined limit. This limitation can be specified by a transformation and normalizing the range of the gradient. For example, we cut off all gradients larger than a specific threshold, e.g., 1, or smaller than a specific threshold, e.g., -1. Then, all gradient values above 1 are set to 1, and all values below -1 are set to -1.

Gradient clipping can be applied to any type of neural network where the risk of exploding gradients is a concern. However, Gradient Clipping is commonly used in RNN networks, which we will explain later.

Batch Normalization

Back in Chapter 3, we described the standardization and normalization approaches. Also, in the Clustering chapter (Chapter 4), we provide an example that all data should be in the same range to be able to cluster them. If we intend to have a neural network that handles different numerical ranges, we should normalize them. For example, we would like to have a classification algorithm to predict the happiness of our users based on their age and annual income. The age is a number between 0 and 120, but income is widely varied. For example, in the U.S., it could be from 10,000$ per year to >10,000,000,000$. Therefore, we need to bring these variables into the same range. In the neural network, Batch normalization is a bit different from the previous normalization methods we have learned.

Batch Normalization or *Batch Norm* normalizes the activations of a previous layer at each batch, meaning it applies a transformation that maintains the mean output close to zero and the output standard deviation close to one. It is done at the input layer and hidden layers after intermediate layers within the network. Batch normalization helps speed up training, reducing the sensitivity to network initialization and mitigating the vanishing/exploding gradient problem.

Batch normalization was introduced in 2015 by Loffe and Szegedy [Loffe '15]. It adjusts the outputs of the previous layer to have a mean close to 0 and a standard deviation close to 1 based on the current mini-batch. Keep in mind that *it's about the activations, not the weights directly*.

The process of Batch normalization is implemented as follows:

(i) Assuming that m is the batch size of the input and x_i is the ith example in this batch, first, mini-batch mean (μ_B) and mini-batch variance (σ_B^2) for each input batch will be calculated, independently.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2$$

(ii) Next, the output of an activation function will be substituted with the z-normalized value of it, before passing it to the next layer. In Chapter 3, we described the z-score, written as:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sigma_B}$$

(iii) After the value (output of activation function) of each neuron is normalized, then it multiplies the output by an arbitrary parameter (scale) γ and adds another arbitrary parameter (shift) β, to the result, as follows:

$$y_i = (\hat{x}_i \times \gamma) + \beta$$

Similar to weight and bias parameters, both of these parameters (scale and shift) are learned during the training process, and the optimizer configures them. Therefore, we can write the equation of batch normalization as follows:

$$y_i = \gamma \frac{\hat{x}_i - \mu_B}{\sigma_B} + \beta$$

Once again, let us remind you that the Batch norm is applied to the outputs of the input layer and the output of the hidden layers.

Now that we understand batch normalization, a question might arise: Why do we need batch normalization and not just use traditional normalization? The problem is that traditional normalization may not go far in a deep neural network, and still is prone to several issues. One of these issues is referred to as *covariate shift*, which states that the distribution of original input data as it proceeds through its hidden layers will change. In other words, input values in each layer are scaled by trainable parameters, and as parameters get turned back by the backpropagation algorithm, the original distribution of input data will change. Batch norms can mitigate this issue by updating the network's parameters, thereby stabilizing the training process and often resulting in faster convergence.

Other Normalization Techniques

In addition to Batch Normalization, there are other kinds of normalizations, including *Instance normalization* [Ulyanov '17], *Layer Normalization* [Ba '16], and *Group Normalization* [Wu '18]. Before we explain them, we should know that the input data fed into a neural network is in tensor format, and it has height, width, and depth (channel). We will learn more about input data later.

Instance Normalization is a type of batch normalization that is applied for a specific instance. For example, if the neural network applies a Batch Norm on a set of images while instance normalization applies the normalization on every single image. In other words, it treats *each input sample separately for normalization*. Therefore, mean and variance are calculated for each channel of an individual sample across both x and y (spatial) dimensions and not for batches of input samples.

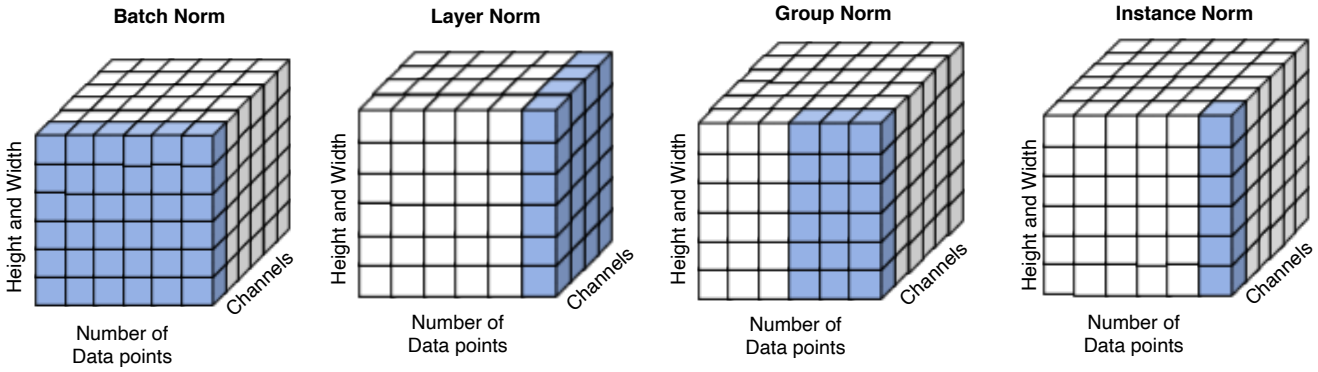


Figure 10-22: Visualization of different neural network normalization methods.

Layer Normalization tries to address the Batch Norm's dependency on the batch sizes, which is particularly challenging for RNN networks (we will explain RNN later in this Chapter). To

handle this limitation, Layer Normalization [Ba '16] introduces a *normalization across channel dimensions*. Similar to instance normalization, layer normalization normalizes the data across all the features (channels) within a layer for each individual sample rather than across different samples in a batch as in batch normalization.

Group Normalization tries to address Batch Norm's need for large memory because of a need for a large set of batches [Wu '18]. Group Normalization divides the channels into groups and computes the mean and variance of the normalization in each group, which makes it independent from batch sizes. Figure 10-22, is designed by Wu et al. [Wu '18] to visualize these normalization approaches.

Dropout

One of the most popular neural network regularization methods introduced by Hinton et al. and popularized by Srivastava [Srivastava '14] is Dropout. The idea of dropout is simple but very effective. During training, a random subset of neuron outputs in a layer is set to zero in each epoch, reducing their contribution to the next layer temporarily. It is typically implemented by adding a dropout layer to the network. For example, a network might have 50% of its neurons in the first hidden layer 'dropped out' in each iteration to prevent overfitting. While dropout rates

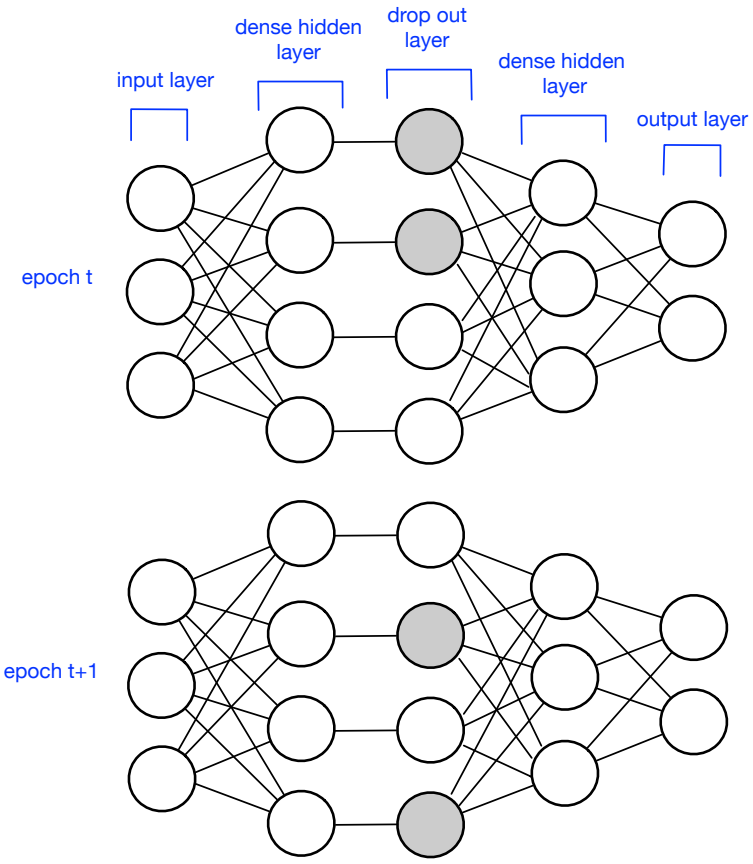


Figure 10-23: A mock example of having dropout layer after the first hidden layer. Nodes which are converted to zero are presented in grey color and this layer in each epoch turns off (make their activation function output zero) 50% of nodes.

commonly range from 20% to 50%, the optimal rate depends on the specific context and network architecture. Dropout ensures that the network's predictive performance does not rely too heavily on any single neuron, promoting more robust learning. Figure 10-23 presents a simple example of a dropout regularization. There, we have a neural network, and in every iteration, 50% of its first layer of hidden neurons are turned off (their weights will be equal to zero).

Gal and Ghahramani [Gal '16] introduced an extension called *Monte Carlo (MC) Dropout*, which applies dropout during both the training and test phases to obtain a measure of predictive uncertainty. In MC Dropout, the network makes multiple predictions for the same input with different neurons dropped out each time, and the results are averaged to estimate the final output. This approach can enhance the model's ability to estimate its uncertainty but should be used carefully in applications where accurate uncertainty estimation is critical, such as in medical diagnostics.

Early Stopping

We have explained that training a neural network is a computationally expensive process. To manage this, we employ regularization methods, which help prevent overfitting and can also indirectly reduce computational costs by promoting simpler models. During training, we monitor the loss, and once it reaches an acceptable level, we can halt the training process. One popular regularization technique is *early stopping*, which we detailed in Chapter 8. Early stopping involves terminating the training when the model's performance on the validation set starts to deteriorate, thus preventing overfitting and potentially saving computational resources. There is not much more to add here except to reiterate that early stopping can be effectively implemented in neural network training to optimize resource usage.

NOTE:

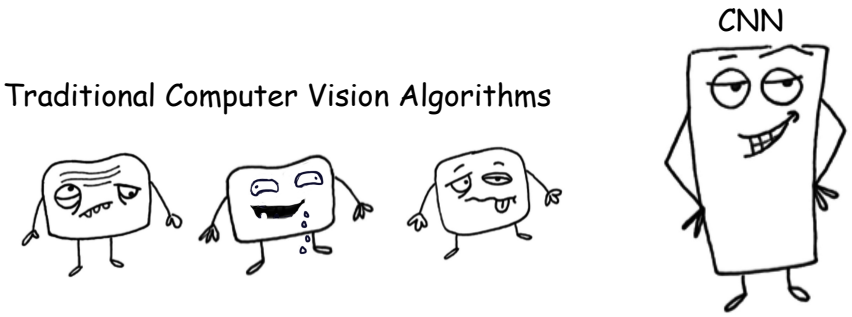
* We have explained that neural networks suffer from high computational complexity, which makes them unsuitable for settings with limited computational capacities, such as battery-powered devices. There are several different approaches to making neural networks lighter, and we will explain them in Chapter 14.
* There is no optimal way to design the number of layers and neurons, and since there is no methodological approach to doing it, the selection of neurons and layers is referred to as 'dark art'. A simple common approach is to use a large model, which is larger than our need, and use dropout to prevent overfitting. This approach is called 'stretch pants', in which we do not search for a pant that fits; we get a large pant that stretches and fits our size.
* A neural network has lots of hyperparameters to tune such as learning rate, number of hidden layers, number of neurons, cost function, types of activation function, optimization algorithm, batch size, etc. It is common to rely on some predefined settings, but you might change a parameter and make some significant advancements; nobody knows until you experiment.

Convolutional Neural Network (CNN)

We have covered the general architecture of neural networks, and now we turn our attention to specific deep learning architectures, starting with Convolutional Neural Networks (ConvNets or CNNs), which are pivotal in computer vision.

Back in 1950, Hubel and Wiesel [Huhel '59] identified that monkeys' and cats' brains have two types of cells in their brains (simple cells and complex cells) to be used for visual recognition. Later, Fukushima [Fukushima '80], inspired by the findings of Hubel and Wiesel, introduced the *Neocognitron*, a hierarchical, multilayered artificial neural network, which is considered a precursor to modern CNNs. Afterward, LeCun et al. [LeCun '89] combined the CNN architecture with the Backpropagation algorithm and experimented with it on handwritten numbers[13]. Their results showed a tremendous improvement in the accuracy of handwritten number recognition.

CNN architecture preserves the spatial structure of the input data. Preserving the spatial structure of data makes CNN very helpful for computer vision applications such as image classification, image segmentation, medical image analysis, etc. CNNs are used in a few other applications, such as natural language processing or audio analysis. Nowadays, most advances in computer vision include CNN or the use of neural networks.



CNN models are spatial invariant, which means they can recognize patterns in an image regardless of their position or orientation. They are flexible in tolerating distortion and changes in images. These features make them more accurate than the traditional image feature engineering we have described back in Chapter 6. However, CNNs are not scaling or rotation invariant.

At the time of writing this chapter (first time in 2021, and revision in 2024), most state-of-the-art image analysis algorithms are using CNN, and they even overtake human experts in some

[13] There are some very popular datasets used to evaluate computer vision algorithms. One is a handwritten number dataset called MNIST (stayed for Modified National Institute of Standards and Technology database), which to date is the most popular image dataset in use for experimenting with different types of machine learning algorithms, especially image recognition ones. There are a few popular datasets, including ImageNet [Deng '09], mtCars (vehicle data), IRIS (flower shape), CIFAR-10/CIFAR100 (tiny images of different objects), and Fashion MNIST (cloths images), which are used for experimenting or benchmarking machine learning algorithm.

computer vision applications such as medical image analysis [Javaheri ’21]. CNN operates by assigning multiple image filters to an input image, and these filters enable the algorithm to classify the image accurately.

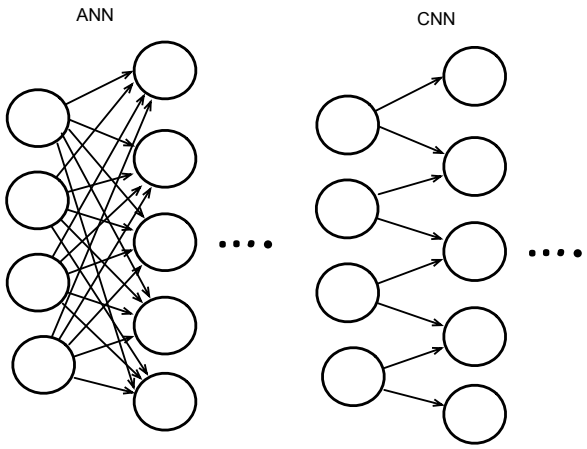


Figure 10-24: A traditional input layer and the second layer of ANN on the left and the same approach for CNN on the right, we can see the number of connections in CNN is smaller than ANN, which is fully connected.

CNN has two significant advantages over ANN. First, ANN uses densely connected layers, each neuron in CNN is connected to a limited number of neurons in the neighbor layer. Figure 10-24 shows a comparison between CNN and ANN, and we can see that all input neurons of the ANN are connected to all neurons in the next layer. Having this dense connection causes ANN to learn only *global patterns* in the image (the result of having densely connected layers where every output neuron is connected to every input neuron). However, CNN does not use densely connected layers, and by using a window over the image, it can learn the *local patterns*. The size of the window and the movement steps, or 'stride', are key factors in defining the extent of these connections, which we will explore in detail later.

To understand the importance of local patterns, let’s discuss an example in the MNIST dataset. If the handwritten number is not in the center of the image, ANN cannot classify it, but CNN can classify, even if the number is located on the side of the image, because of its local pattern learning capability.

The second advantage of CNNs is their ability to learn hierarchical features within an image. For instance, in analyzing an image of a chicken, a CNN might first identify basic features like edges and curves, then assemble these into more complex structures such as beaks, eyes, and wings, and finally recognize the overall form of the chicken. This hierarchical feature learning enables CNNs to understand complex structures and classify images more effectively, even when they vary in perspective or lighting conditions. Thus, CNNs are capable of distinguishing and classifying images in a way that is challenging for other computer vision algorithms, demonstrating their robustness across different viewing conditions.

Typically, a CNN network takes an input of a tensor, which includes image height, width, and channels (three data for red, green, and blue in RGB mode or three data for hue, saturation, and lightness in HSL mode). We can say that we work with a 3D tensor for colored images. For the sake of simplicity, we explain our example with a matrix and one value for each pixel.

CNN algorithm operates in four steps, but before starting to explain these steps, we need to learn the definition of convolution. Then, we proceed with the explanation of these four steps in a sequence.

Convolution and Cross correlation

Convolution is the process of combining two functions and, as a result, getting a new function with a different function shape than the two input functions. In the context of neural networks, convolution is the process of multiplying matrices (or tensors), and the result will be a new matrix (or tensor). Mathematically, convolution (*) is written as *the integral product of two functions after one of these functions is reversed and shifted*. The convolution specifies the amount of overlap of one function as it is shifted over another function. The convolution operation can be described using the following equation:

$$(g * f)(t) = \int_{-\infty}^{\infty} g(\tau) f(t - \tau) d_{\tau}$$

In this equation, t is the current time and τ is used to specify the size of the shift. The integral measures the overlap between $g(\tau)$ and $f(t - \tau)$ as one function slides over the other. If two functions are not overlapped, their convolution is zero. When a function overlaps with another function, then its convolution is none zero. On the top of Figure 10-25, we have two functions *f* and *g*. The bottom plots their convolution. We can see the amount of shift on top and its impact on the convolution at the bottom. The grey area on the top shows the convolution. Note that, as part of the convolution process, the function *f* is reversed starting from the second figure onward.

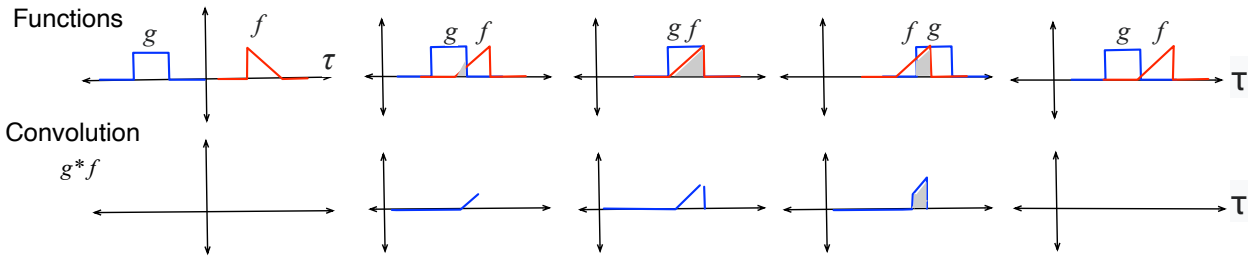


Figure 10-25: Two functions at the top and their convolution function in the bottom. The *g* function is moving along τ-axis and as it overlaps with the function *f* (the grey area is the overlapped area) the convolution starts to increase and then again decreases as the *g* is moving away from *f*. Note that the function *g* is also reversed, but its shape doesn't changed.

Cross-correlation is similar to convolution, but it measures how similar two different functions are. In other words, while both operations involve sliding one function over another and computing integrals or sums of their products, the key difference is that convolution flips one of the functions before this process. Specifically, in convolution, one of the functions is reversed along one of its dimensions (flipping means taking the negative of its indices), while cross-correlation does not perform this flipping. e.g., $f(x) = x$ and flipping it will be $f'(x) = -x$.

The following presents an equation of cross-correlation:

$$(g \otimes f)(t) = \int_{-\infty}^{\infty} g(\tau) f(t + \tau) d_{\tau}$$

Usually, when we work with the digital image, the data is considered discrete and multi-dimensional. Assuming f is one function (image), and g is the other function (kernel), the convolution equation for 2D data can be written as follows:

$$(f * g)(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} f[i-n, i-m] \,.\, g[m,n]$$

Respectively, their cross correlation will be written as:

$$(f \otimes g)(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} f[i+n, i+m] \,.\, g[m,n]$$

You don't need to learn the mathematics of convolution in this detail, but as general knowledge, they are good to be known, and you can brag about your knowledge while talking about CNN.

CNN Architecture

To implement a CNN, we should follow five steps, and we describe each step in detail.

Step 1- Convolution: The first step focuses on using kernels (filters) to construct feature maps. This step of applying filters on images is referred to as convolution. In this context, the kernel is a different concept than the kernel we explained for SVM in Chapter 9. The task of kernels or filters in the CNN is feature detection. Kernels are similar to the image filters that we use in photo editing software, such as blurring an image, sharpening, etc. Filters or kernels are small matrices of weights that slide over the input image to extract specific features, such as edges, textures, and patterns. Figure 10-26 shows some examples of filters applied to a sample image.

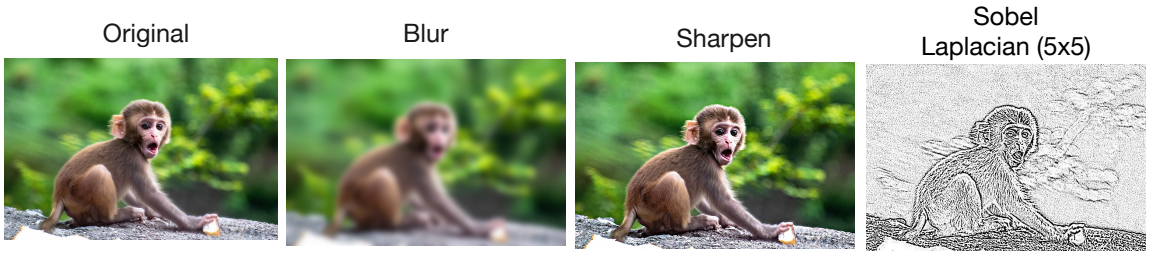


Figure 10-26: An example of image and three filters that are applied on the image.

In the context of CNN, we call these filters *convolutional kernels* (or filters), and the resulting image is a **c***onvoluted image*. Therefore, we can write the following:

*input image * kernel (feature detector) = convolved image (feature map)*

There are some common kernels, e.g., a matrix with a Gaussian distribution is a kernel used for blurring an image. A few examples are presented in Figure 10-27. We can see some filters, and intuitively, by looking at the matrices, we can realize what they are doing. For example, the Sharpen filter on the right increases the differences between the right and left center pixels. It reduces the emphasis on the other pixel, which results in sharpening the image. The convolution

process usually results in a feature map with fewer pixels than the original image; as we can see in Figure 10-27, the convolved image is smaller than the original image.

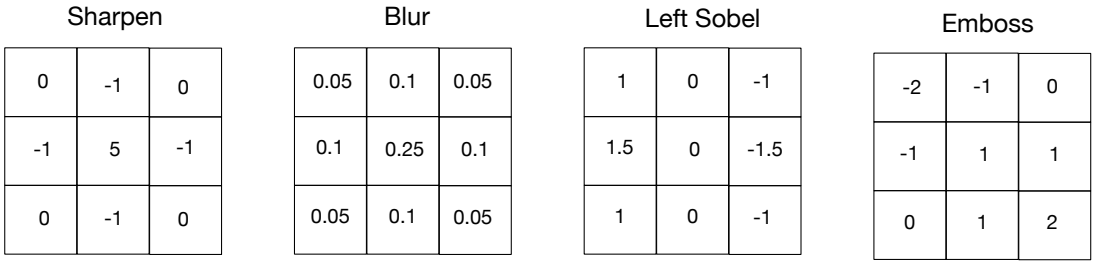


Figure 10-27: Some sample image filters. Sharpen and Blur are clear, Left Sobel is used to show differences between each pixel and its adjacent pixel on the left. Emboss creates an illusion of depth for the viewer of the image.

Take a look at Figure 10-28. We have an image, and for the sake of simplicity, we used a matrix and not a tensor; the process of convolution is a type of matrix multiplication but with a small window (or patch) on the original image. You can see the result of applying the kernel on the original matrix. Using this kernel increases the emphasis on pixels located on the left side of the kernel. The red window of Figure 10-28 is moving along the matrix and constructs pixels of the convolved matrix. The number of pixels that a window moves is called *stride*. In Figure 10-29, we move the window one pixel in the *X* and then *Y* direction. Therefore, we say the stride is equal to one. Usually, the stride is set to two pixels, and the larger the stride, the smaller the resulting convolved image becomes.

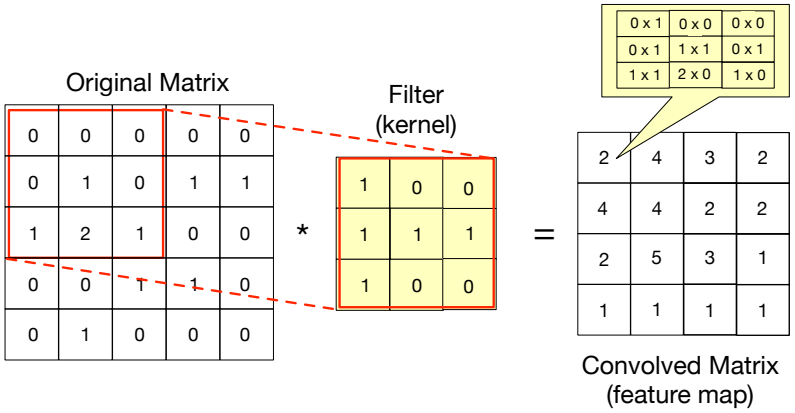


Figure 10-28: A matrix, which presents image and a filter that will be applied on the matrix. The right matrix presents the result of applying the filter on the original matrix.

The '*' operator here refers to convolution (sometimes informally referred to as element-wise multiplication in this context) and not the dot product. A filter is always smaller than the original image.

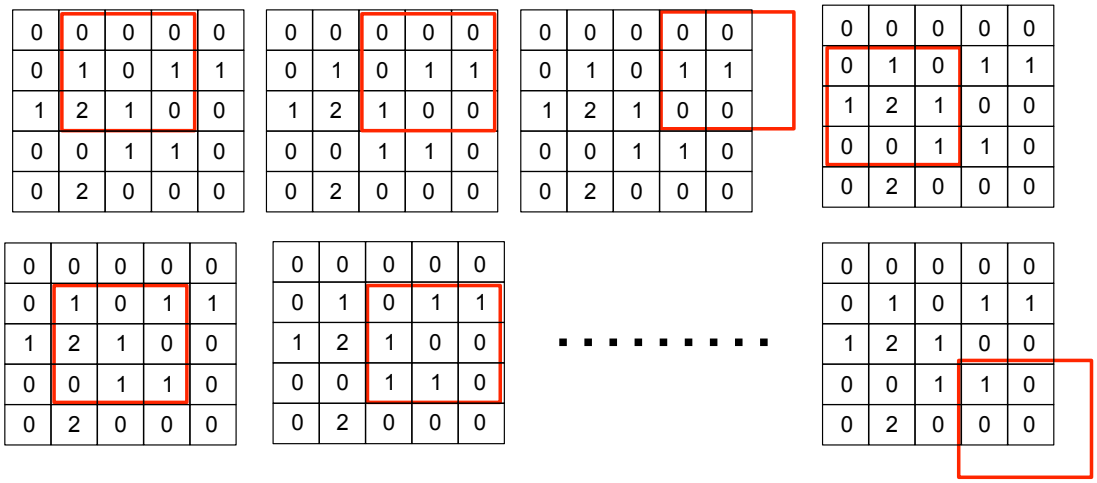


Figure 10-29: An example presented a window equal to the size of the filter (3x3) is moving along the original matrix (image) and select a window of data to multiply them to the filter values. Here, the stride is equal to one on both the X and the Y axis.

Another issue is illustrated in Figure 10-29. where the window extends beyond the image matrix, leading to a loss of the image's border in the convolved image. To avoid losing the borders, we can use a process called *padding*. Padding is the process of substituting the pixels on the edges. There are different approaches for padding, including *reflection*, *replication*, and *zero padding*. Otherwise, the convolutional filter does not go outside the image border; its padding is referred to as *valid padding*. If the output of the convolution has the same size as the input image, its padding is referred to as the *same padding*.

Zero padding (masking) is just adding a zero in the padding area. Replication padding is using the same value for the last pixel in the padding area and substituting it in the padding area. Reflection padding uses the neighbor value of the last pixel on the opposite side of this pixel and substituting it in the padding area. Valid padding happens when the filter window stays inside the image and does not go outside. Figure 10-30 presents these three padding techniques.

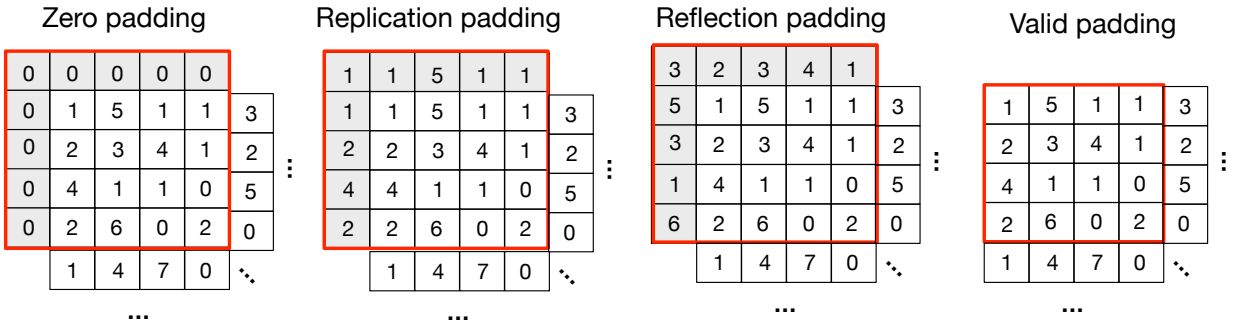


Figure 10-30: Three different approaches for padding on edge pixels. The white area in this figure presents the image pixels and grey ares is the padding area.

Applying image filters to the input image results in having a *convolutional layer* for the neural network. Note, if you encounter the term "local" in the convolutional filter, it refers to the pixels inside a window (the content of red windows in Figure 10-29). Assume that s presents the stride size, p is the padding size for $n \times n$ input matrix, and we apply $f \times f$ filter, the output feature map has the following size: $(n + p - f)/s + 1 * (n + p - f)/s + 1$.

Step 2- Perform non-linear transformation: Filters create a convolved image, which is a linear transformation of the original image. However, an image includes a set of non-linear objects. To increase the non-linearity of feature maps (convoluted images), the feature map will usually be fed into a non-linear activation function such as ReLU to get a non-linear transformation of the image.

Not having a non-linear transformation after the convolution layer results in lower classification accuracy, which this sin is equal to blasphemy in machine learning. In particular, this non-linearity of feature maps (convoluted images) makes the neural network really strong, and this step is called the 'detector step' [Goodfellow '16].

Keep in mind that at the end, the output layer should be something related to the task we expect from the neural network. For example, if it is a classification task for more than two classes, then the activation function of the output layer should be Softmax, for two classes, it should be Sigmoid (check Chapter 8 to recall Softmax and Sigmoid).

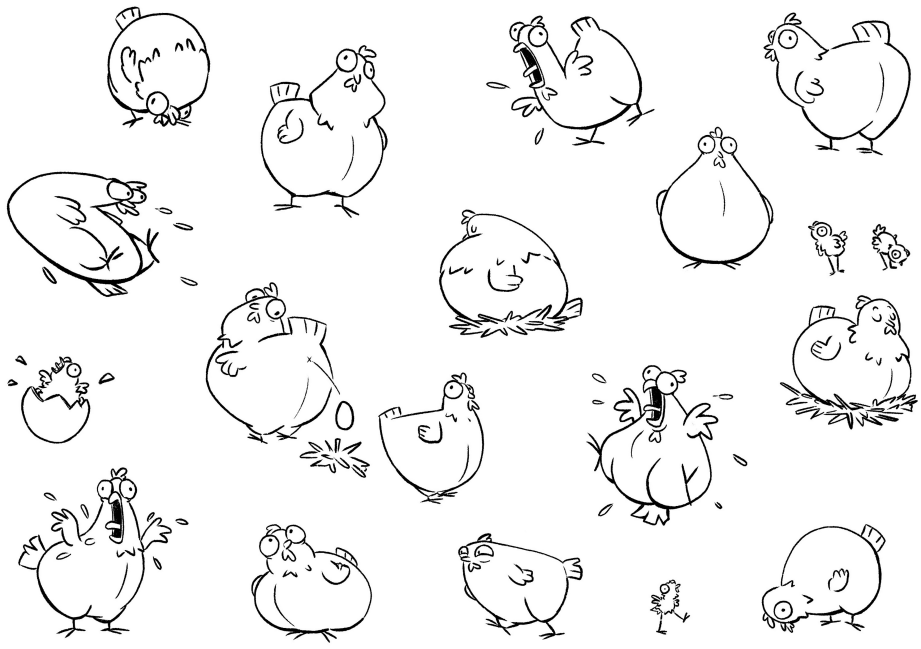


Figure 10-31: All of them are chickens but a spatial variant algorithm cannot detect them. CNN is spatial invariant and it can extract features such as body shape, peak, eyes, etc. and thus recognize them. However, it can recognize them, only if we train the algorithm with enough sample images.

Again, let us remind you that unlike ANNs, where all layers are connected to each other (fully connected), CNN neurons are only connected to a subset of neurons in the next layer, and these are filters. Figure 10-24 visualizes the difference between ANN and CNN at the input layer, which we can see in CNN neurons are not connected to all other neurons in the next layer.

Step 3- Pooling: The feature extraction, done automatically by the CNN, is spatial invariant. Spatial invariance makes CNN algorithms a very accurate and flexible algorithm for image classification. For example, traditional image feature extraction methods cannot recognize that Figure 10-31 are chickens because the shape, camera angle, etc., are different among all chickens. However, a CNN trained on many comic chickens can recognize the given chicken image despite their different visual shapes.

Each convolutional layer applies different filters on the input image for feature extraction. The number of filters is a hyperparameter and grows between convolutional layers. For example, the first layer has 32 filters, the next layers 64 filters, and so forth. Therefore, a CNN creates many convolved images (feature maps). Dealing with such a large number of data (results of applying many filters) is computationally very ineffective or impossible.

To handle this issue, the CNN downsamples (reduce the size) of convolved images while maintaining the highlighted features. The pooling functionality is used for this purpose (downsampling), similar to using window and strides, which downsample the original image into smaller images. The result of the convolutional layer, i.e., feature maps (after they have been transformed into non-linear feature maps), are sent to a pooling layer via a *tensor operation* that performs the downsampling. The pooling layer performs something similar to a kernel function (check Chapter 9 SVM algorithm explanation), kernel function downsamples, by using a linear transformation, but pooling performs downsampling by tensor operation, e.g., Figure 10-32 shows some examples of downsampling via pooling. Downsampling here refers to lowering the filter resolution while still maintaining its features. If you read Chapter 6, you might recall the

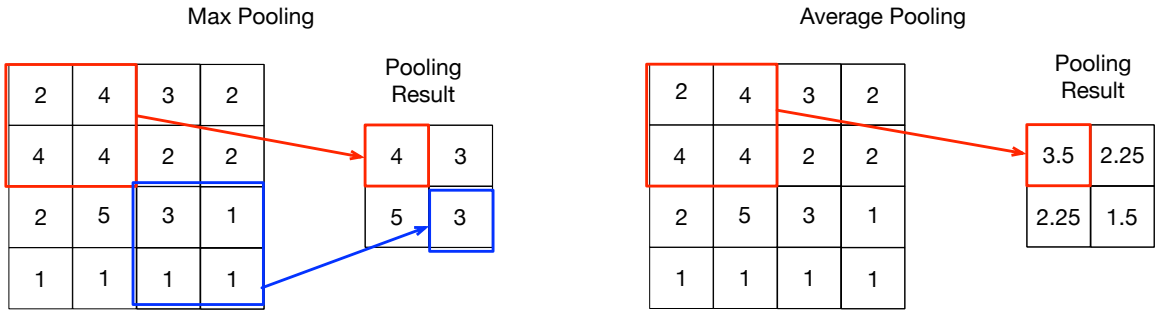


Figure 10-32: Max pooling and average pooling examples are applied to the convoluted image (feature map), and the pooled feature map is created. In these two examples, we use a stride of two pixels in each direction.

SIFT algorithm, which blurs and resizes the image for feature extraction. That process can also be called downsampling as well.

Pooling, similar to convolution, slides a window on data, and two types of pooling are commonly used: max pooling and average pooling. *Max pooling* takes the largest element of the window it defines. *Average pooling* averages the content of the window and presents it as a single pixel for the next layer. Figure 10-32 present max pooling and average pooling examples.

There are other pooling methods as well, such as calculating L_2 *norm* for all pixels inside a window, but they are not as popular as max pooling.

Pooling has another advantage: it removes irrelevant information and unnecessary features. It reduces the chance of overfitting (because of reducing the number of parameters) in addition to making the input smaller and thus making the network more resource efficient.

Step 4- Flattening: After the end of the pooling layer, there is usually a normal flattening layer. Flattening is very simple; the result of the pooling layer is a matrix, and the flattening layer converts matrices to a vector, as shown in Figure 10-33. Since we have many pooling layers, the vector size is usually too long. This process occurs before the data is passed into the fully connected layer(s).

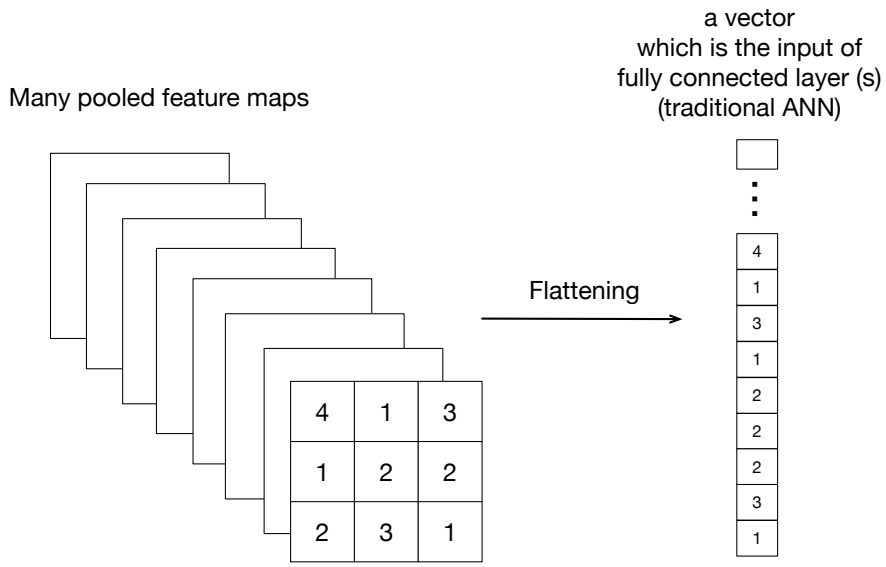


Figure 10-33: The results of max pooling functions are flattened into a large vector, which will be the input of the dense layer (traditional ANN).

Step 5- Dense Layers: After creating this very long vector, it will be fed into a traditional ANN layer. The input layer of this ANN is the result of flattening; its hidden layers are fully connected layers, and the output specifies the result of the classification or regression. The reason that hidden layers of the ANN here are called fully connected layers is because all neurons are connected, as is common with ANN architecture. We have described that CNN neurons are locally connected, but the last layers before the output layer are fully connected. In the fully connected layer, the error is calculated and also backpropagated to the neural network until the weights and biases of the networks converge (maximum iteration reached or accuracy reaches a satisfactory number).

If a feature is useless, its weights get reduced during the iterations of backpropagation. The number of neurons in the output layer is equal to the number of classes; for example, if we make a CNN distinguish whether a given image is Chicken, t-rex, or other, the output layer will have three neurons, one for chicken, one for t-rex and one for other.

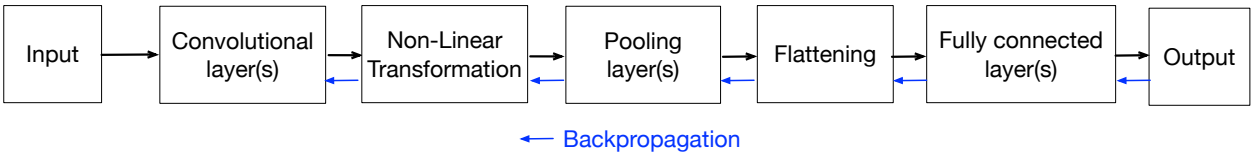


Figure 10-34: A brief summarization of CNN architecture. Here, fully connected layers refer to traditional ANN.

Figure 10-34 summarize what we have explained in the CNN architecture. However, there are usually multiple convolutional layers, multiple pooling layers, and multiple fully connected layers. We could even have several convolutional layers and pooling layers in a sequence. Note that the network learns the filters. These filters, which are small matrices of weights, start as a random set of values and are then adjusted through training iterations to help achieve the goals of the network.

A CNN architecture can be used for classification and rarely for regression. If the CNN model is designed for classification, it will have more than one neuron at the end, and for the classification, its output layer neurons are equal to the size of the classes that are used for labeling. For example, if a CNN classifier is trying to distinguish between cat, dog, and t-rex. Then, we will have three neurons. Nevertheless, the neurons in the output layer are unaware of the value of other neurons in the last layer. However, they should provide a probability, and the sum of those probabilities is equal to one. The Softmax function (described in Chapter 8) is used to convert them into a probability and assign a value to them that adds up all of the output neurons to one. In other words, CNN used for classification has a Softmax activation function (or other non-linear activation function) that outputs probabilities for a classification result. The softmax function is a generalization of the logistic function and provides a vector of values with a range between zero and one. The softmax function is usually used with cross entropy as a loss function.

It is not common to use CNN for regression, but if you intend to use a CNN for regression, there will be one output neuron, and its activation function will be linear.

The best way to design a CNN is to experiment with different combinations of layers until we get the best possible result. This can be done by measuring the accuracy and then deciding on the final architecture of the network. In Chapter 12, we will introduce some popular CNN models.

A high number of layers in the neural network leads to a high number of parameters, and thus, the network might get more accurate, but on the other hand, it gets prone to overfitting as well.

Usually, computer vision models are built by very large AI corporations, and it is not easy to generalize all of them for real-world applications developed by non-super-rich corporations.

Therefore, if you develop a model that is accurate and not affiliated with those big AI corporations, good luck in convincing the scientific community about your model.

Different Types of Convolutions

We have learned that the process of convolution is applying a filter/kernel on an image and computing the output, whose size depends on the filter size, stride, and padding. Here, we briefly introduce different types of convolution, but we do not go into detail about each convolution.

2D and 3D Convolutions: Until now, we have only used a matrix and explained convolution on a matrix. This convolution is called 2D convolution. If the original data is in 3D format, we should use 3D convolution. For example, a colored image has three color properties (red, green, and blue) along with x and y coordinates. Therefore, we should deal with a 3D tensor (three matrices with the same width and height). The third dimension is referred to as *depth* or *channel,* and the size of the channel for the original data and filter must be exactly equal. For example, in Figure 10-35, both filter and original data have the same number of channels, i.e., three.

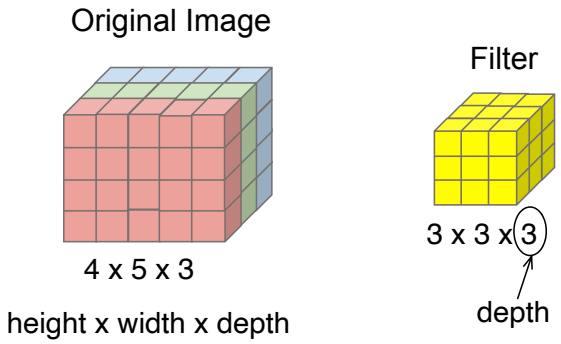


Figure 10-35: An example of a 3D Convolution with its filter. As is shown, the depth or channel in both the filter and the original image should be the same.

The dot product of a 3D filter to a 3D tensor will result in a scalar for each position where the filter is applied. As the filter moves across the tensor with a specific stride, each application produces a scalar. These scalars together construct a 2D matrix, known as the feature map or convolution output[14].

3D convolutions are typically used for volumetric images (such as MRI and CT images), where each voxel represents a point in a three-dimensional space (a voxel resembles a pixel with one additional dimension, that is depth). They are also applied to video files, where the third dimension is time, allowing the filters to capture temporal information in addition to spatial features. 2D images are generally processed with 2D convolutions, as they lack a third spatial or temporal dimension that would necessitate 3D convolution.

---

[14] A good video that visualizes this process is available under this link: https://www.youtube.com/watch?v=D0VoQDDe5zI

Dilated (Atrous) Convolution: Dilated or Atrous[15] convolution adds a gap between pixels that feds into the filter. A parameter called *dilation rate (l)* specifies the size of gaps between matrix elements (e.g., pixels). In other words, parameter l indicates how much the kernel is widened. For example, Figure 10-36 has a dilation rate of $l = 2$, 2D Convolution shown on top of this figure has $l = 1$ (no gap between pixels). As the dilation rate increases, the patch sizes increase as well because it skips several pixels.

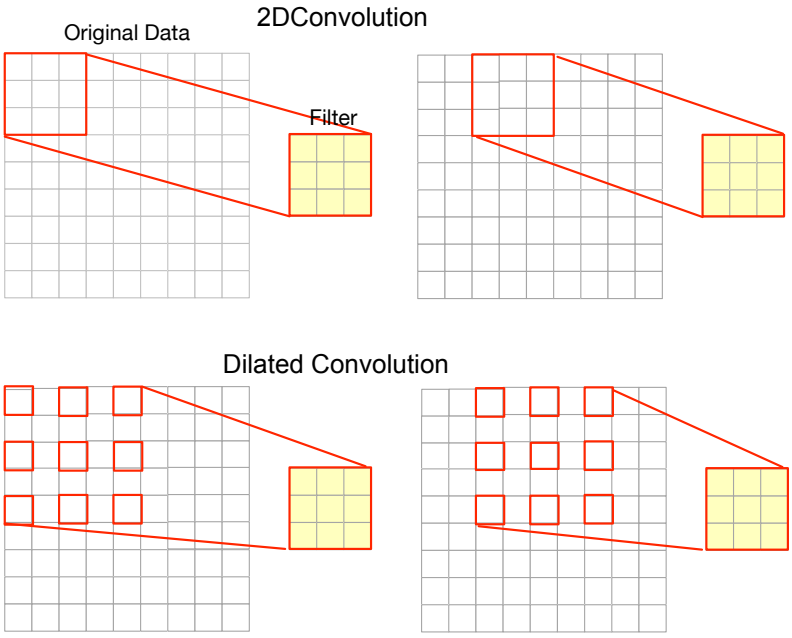


Figure 10-36: A 2D convolution on top and dilated convolution in the button. Both have zero padding and a stride size of 2.

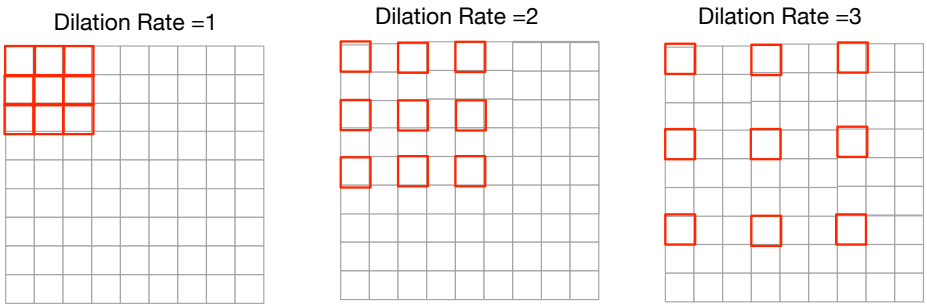


Figure 10-37: Different dilation rate specifies the size of gaps between selected pixels.

Dilation convolutions are popular approaches for image segmentation [Chen '17] because they provide an overview of the larger scope of the image. More about image segmentation will be explained in Chapter 12. Figure 10-37 shows different dilation rates (l).

[15] *trous* in French means holes

Transposed Convolution: all convolutions, except dilated convolutions, typically either preserve the size of the original data or reduce it, which is downsampling. Transpose convolution, on the other hand, is commonly used to increase the size of the output feature, a technique referred to as *upsampling*. Upsampling has many applications in image processing, such as increasing image resolution, reducing image blur, and facilitating semantic segmentation, which involves abstracting the image into a set of labeled objects. These applications will be explored in more detail in Chapter 12.

A transposed convolutional layer performs the convolution operation in a manner that reverses the spatial dimension reduction of standard convolution, effectively increasing the spatial dimensions of its output. Look at the example we provide in Figure 10-38. Here $*_{TC}$ refers to transposed convolution. In this example, we have a small input image (2×2), the stride is 1, and the padding is zero. The output will be a 3×3 matrix, which you can see the input size has been increased from 2×2 to 3×3. In particular, every cell of the input data will be multiplied by all filter cells and written in a similar position. Then, all these results will be summed in a 3×3 matrix. In Figure 10-39, we use red and blue colors to show how two cells construct the upsampled version of the input data cells. The inverse of convolution is referred to as deconvolution, and it is recommended not to call transposed convolution deconvolution.

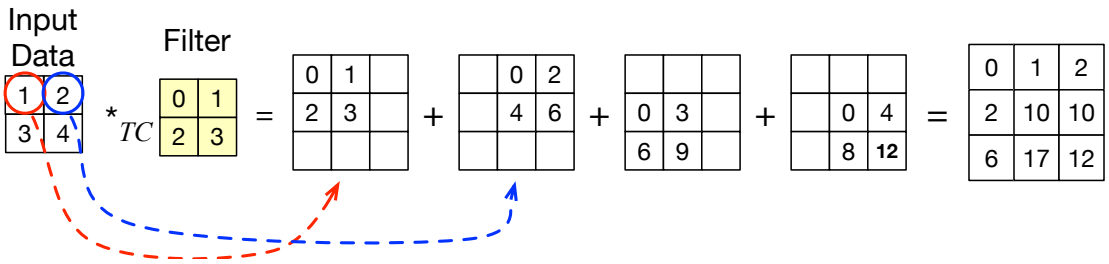


Figure 10-38: A simple example of transposed convolution, with stride size 1 and padding size 0.

NOTE:

* The CNN will learn the kernel (filter) itself, and we do not need to give the kernel manually. In particular, the algorithm will decide which kernel will get meaningful information from an image. Edge detection kernels (Sobel filters) are very common to be used by CNN algorithms.
* Often, in real-world cases, we stack several convolutional layers on top of each other. This means that the output of one convolutional layer will be an input of another convolutional layer and so forth. This causes the CNN to discover more complex patterns as it goes deeper into convolutional layers. In simple words, for example, a CNN with five convolutional layers could recognize more patterns than a CNN with three convolutional layers.
* Nowadays, we rarely need to build a CNN on our own, and most of the time, we are using an existing model to define the model that fulfills our demand, i.e., transfer learning. More about transfer learning will be explained in Chapter 14.

* Pooling layers help manage varying sizes of input images by reducing the spatial dimensions of the feature maps, thus creating more manageable and smaller representations. However, they don't directly handle varying input image sizes; rather, they contribute to making the CNN architecture more flexible and capable of dealing with the spatial hierarchy in images..
* There are different visualization techniques available to present CNN architecture. AlexNet visualization [Krizhevsky '12] is another form to visualize CCN, as shown in Figure 10-39. Here, both convolutional and pooling layers are presented as cubes, and the red cubes inside convolutional layers are filters.

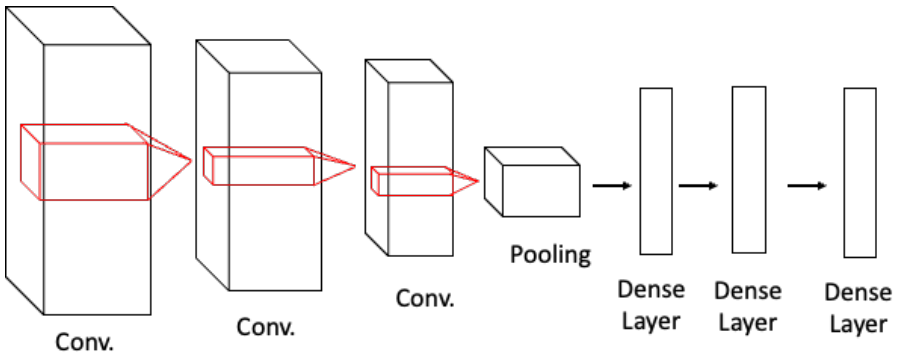


Figure 10-39: AlexNet style of presenting a CNN network architecture. Usually the sizes (height, width, channel) on the cube sides as well.

* CNN models are known to have *inductive bias*, which refers to the built-in assumptions within the model that guide its learning process. These assumptions are not directly derived from the training data but are inherent to the model's architecture, facilitating the identification of relevant patterns and features. Inductive bias enables CNN to generalize from its training data to new, unseen data, allowing it to make predictions or classifications beyond the specific examples it has been trained on. In simple terms, inductive bias is what empowers a CNN to generalize beyond its training dataset.

Recurrent Neural Network (RNN)

We have learned that neural networks, either CNN or ANN, could be used for classification and regression. Until now, our neural networks have received a fixed-size input, such as an image or vector, and the neural network output is a single output. For example, the output will be a vector of two scalars, which includes a classification result on the input image to recognize whether the picture is t-rex or chicken (0.4 for chicken and 0.7 for t-rex probabilities).

Another category of deep learning algorithms is Recurrent Neural Networks (RNN). RNNs are used for *sequential data*, such as sensor data, timestamped data, medical device data, audio, and even text (word appearances in a sentence have an order). In particular, any data with either a timestamp or implicit notion of time can be assumed as sequential data and modeled by RNN. RNNs could be used to reconstruct sequential data similar to the data we fed into it. For example, we could feed a book to RNN, and it can generate sentences similar to that book, or we could use RNN to construct music or poem, but at the time of writing this chapter (2020), they are not accurate enough for Generative AI tasks, but excellent to model sequential data.

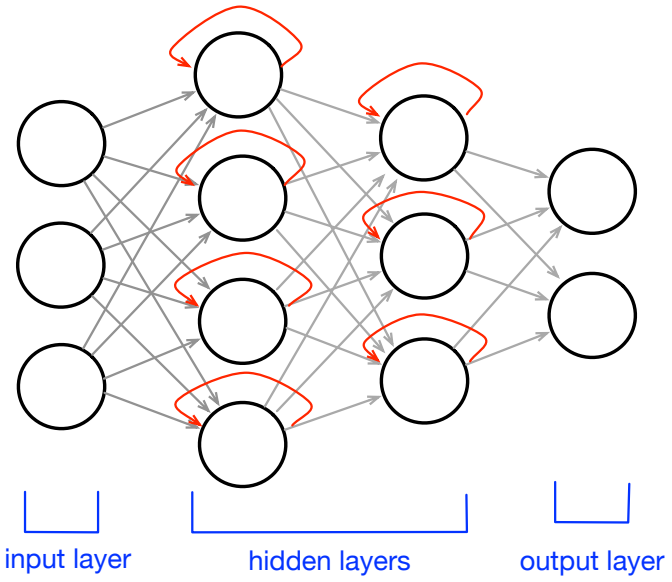


Figure 10-40: Hidden layers in RNN send their output as input in the next epoch. The red lines present the information flow in the next epoch. Here for sake of simplicity we show each neuron has a input from itself, but in fact RNN cells have input from itself, and they are different than neurons (we will explain more about this later).

As we can see from Figure 10-40, hidden layers of RNN do give output to the next layer, but they also feed back their outputs to themselves along with the input (in the next epoch). In other words, they support a temporal loop.

Figure 10-40 shows an abstract representation of RNN. Hidden layers (shown with rectangles in Figure 10-40) are connected to themselves. It means that in addition to the output for the next layer, they also feed the output of the current hidden layer to themselves as well.

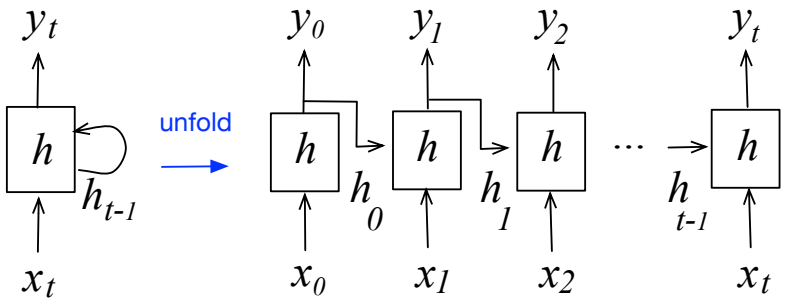


Figure 10-41: The hidden layer (h) in RNN receives input x and produces output y. However, the hidden layers have a temporal loop, which means that in addition to giving the output, it feeds the output to itself as input for the next epoch as well. Each rectangle presents a hidden layer and not a single neuron.

An unfolded hidden layer is shown in Figure 10-41. On the right side of this figure, we can see that the hidden layer at epoch (or time) t receives the output of the previous epoch $(t-1)$ as well, or the layer at a epoch $t-1$ receives the output of epoch $t-2$. Therefore, we could say that neurons have *short-term memory* and remember what has happened in the previous epoch (only a single previous epoch, not all previous epochs). To formalize this, assuming $\vec{y}$ is the output vector of hidden layer variables and $\vec{x}$ is a vector of the input variable, then we have:

$$\vec{y_t} = f(\vec{y}_{t-1}, \vec{x_t})$$

In this context, neurons functioning with inputs from previous time steps can be thought of as having *memory* capabilities, although in basic RNNs, this memory is typically short-term. This memory structure makes the neural network very attractive for some fields of machine learning, including natural language processing and text analysis, because we need temporal knowledge to predict upcoming information.

Based on the number of inputs and output, there are different types of RNN, which are presented as one-to-one, one-to-many, many-to-one, many-to-many, and one-to-one. Figure 10-42 visualizes these types of presentations and notes that rectangles here refer to a layer of neurons.

For a one-to-one example, assume a single image is given into a neural network, and the network finds its label. For a one-to-many example, suppose we give a single image into the RNN, and it generates the caption for the image. For example, in an image, a chicken weares a shirt, and the shirt has a T-rex image on it. An RNN recognizes the t-rex, the shirt, and the chicken. Then, it generates a caption that correctly describes the image. In summary, in one-to-many mode, for one input (an image), it provides many output labels (chicken, t-rex, shirt). For a many-to-one example, assume we give a sentence that includes several words (many) to a sentiment analyzer, and the RNN labels the sentence tone as positive or negative (one). For many-to-many examples,

we can think of a sentence that includes many words in Mandarin, and the RNN translates it to a Hindi sentence, which also includes many words. The position of words is important in the sentence to make an accurate translation. For example, *"I hit the ball with a bat"* is referring the *"bat"*

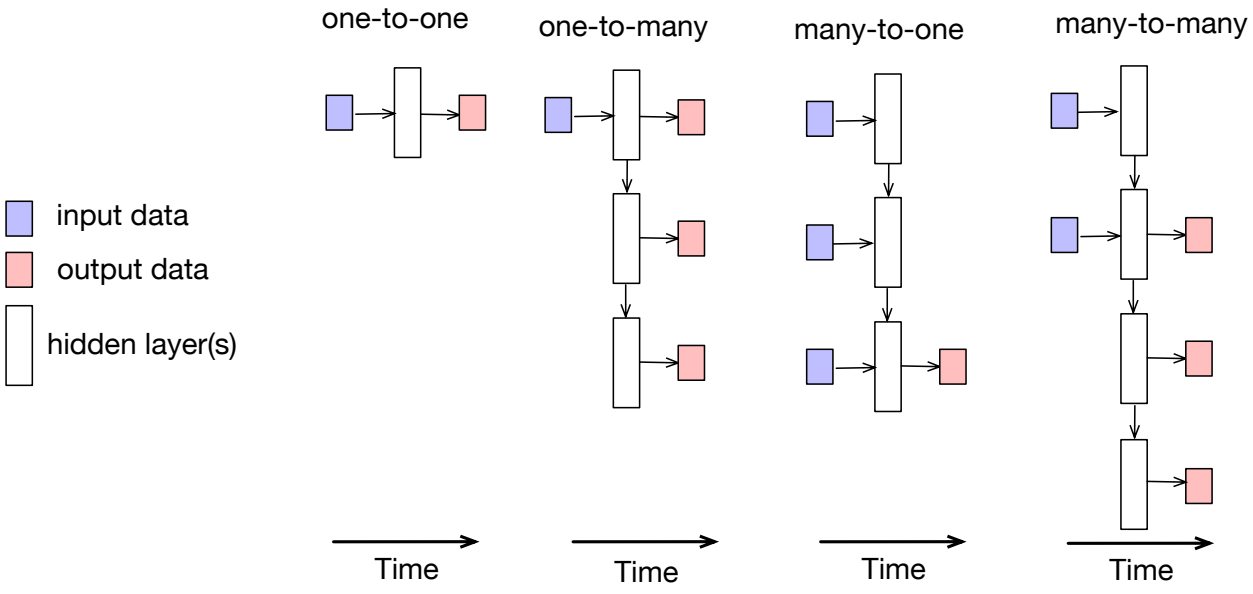


Figure 10-42: Different architectures of recurrent neural networks, you can see from left to right the time increases.

as a wooden stick and not the *"bat"* as the bird *"bat"*.

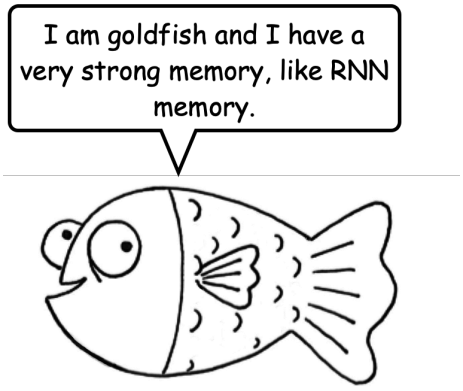
We have explained that a neuron receives an input and puts it in the linear function. Then, it applies an activation function on the result of linear regression, and produces the output.

While using RNN, we have $\overrightarrow{y_t} = f(\overrightarrow{y}_{t-1}, \overrightarrow{x_t})$, which means that the output depends on the previous outputs as well. Also, its activation function is a *hyperbolic tangent activation* function, i.e., *tanh*.

These are the basics of the RNN network. It is fantastic architecture because it has memory, but it has a goldfish memory; it only remembers the previous output, i.e., short-term memory. To mitigate this, we need an RNN that keeps tracking more previous output, i.e., long-term memory.

Long Short-Term Memory (LSTM)

Vanishing gradient is a severe issue in sequential data, and previous approaches we have explained to mitigate its risk are not very practical for sequential data, especially for long sequential data. As a result, using those methods could slow down the system significantly. Deep learning algorithms are consuming lots of resources, and they are slow; thus, applying those techniques makes things even worse. On the other hand, we have explained that the RNN is a goldfish memory; it has only one short-term memory and can't remember more than a few single previous states. For example, take a look at this sentence: *"I love money, and as a consultation fee, I get a fat check."* The word "*fat*" in this sentence refers to money, not overweight people or nutritional fat. Therefore, if we give this sentence to an RNN, which can only remember a few previous words, it cannot recognize the context of the word fat.



A well-known approach to handling time series issues and temporal data is the LSTM architecture [Hochreiter '97]. Instead of one memory, each LSTM cell has four memory components: *long-term memory*, *short-term memory*, *new long-term memory,* and *new short-term memory*. It uses the element-wise operator (Hadamard product explained in Chapter 7), sigmoid, and hyperbolic tangent activation functions to decide about the output value.

Before explaining LSTM, let's briefly review how an RNN works. It receives x_t and h_{t-1} and uses a hyperbolic tangent to calculate h_t the output, which can be written as $h_t = tanh(W[h_{t-1}, x_t] + b)$, as shown on the left side of Figure 10-43.

LSTM neurons are referred to as cells. As shown in Figure 10-43, an LSTM cell receives information called *cell state* (C_{t-1} or previous long-term memory), *previous output* (h_{t-1} or previous short-term memory), and *input vector* (x_t). The output of LSTM is a *new cell state* (C_t or new long-term memory) and the *output vector* (h_t or new short-term memory). The content of each LSTM cell has input, forget, output, sigmoid layers, hyperbolic tangent layers (tanh) gates, and point-wise operators (element-wise or Hadamard product).

Gates are responsible for deciding whether the information passes the gate or not. The Sigmoid layer outputs values close to 0 (not passing the gate) or close to 1 (passing the gate), and *tanh* brings the data into the range -1 to 1.

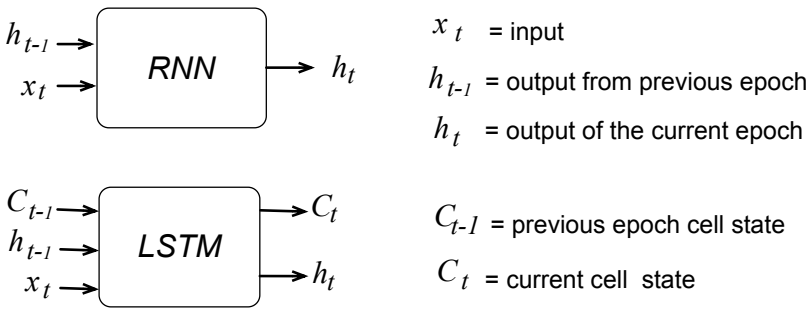


Figure 10-43: Traditional RNN neuron, which is called vanilla RNN, versus LSTM neuron. The LSTM neuron receives C_{t-1} and outputs C_t as well, which are representations of cell state that could be interpreted as long-term memory.

Now that we have a basic understanding, we can describe LSTM, step by step, as follows.

(i) The first step of the LSTM algorithm decides what information is not worth keeping and could be thrown away; this will be done through the *Forget gate*, which is a layer with the Sigmoid function. In particular, first, it receives the h_{t-1} (output of the previous time) and x_t (input), then calculates f by using the Forget gate (Sigmoid function) and outputs $f = \sigma(W_f[h_{t-1}, x_t] + b_f)$. At this step, the state is C_{t-1}, which comes from the previous cell, as shown in Figure 10-44 Step 1.

(ii) The second step decides what new information is going to be stored in the cell state. This step includes three sub-steps. First, it uses the *Input gate* to decide which values will be updated. Its equation is written as: $i = \sigma(W_i[h_{t-1}, x_t] + b_i)$, (see Figure 10-44 Step 2-1). Second, a hyperbolic tangent function creates a vector of new candidate values, i.e., $\tilde{C}_t$, that its value will be between -1 and 1. These candidate values can be added to a candidate new cell state (not the final new cell state), and the equation is written as: $\tilde{C}_t = tanh(W_C[h_{t-1}, x_t] + b_C)$, (see Figure 10-44 Step 2-2). Third, after these two sub-steps, their results should be combined to update the old cell state, C_{t-1}, into a new cell state: C_t. To construct the new cell state, the algorithm multiplies the old state by f_t, forgetting the information that is forgettable, and then decides how much a cell state is getting updated by using $i \odot \tilde{C}_t$. The third sub-step, which constructs the new cell state (C_t) is written as: $C_t = f \odot C_{t-1} + i \odot \tilde{C}_t$, (see Figure 10-44 Step 2-3)

(iii) In the final step, the algorithm decides what is going to be in the output vector, which is based on the filtered version of the C_t cell state. Here, the filter is referred to as passing through C_t to the *tanh* gate. This step has three sub-steps. In the first sub-step, h_{t-1}, and x_t sends to *Output gate* (Sigmoid gate), which decides what part of the cell state will be given as output. The equation of this sub-step is written as: $o = \sigma(W_o[h_{t-1}, x_t] + b_o)$, see Figure 10-44 Step 3-1. In the second sub-step, the cell state will be passed through a *tanh* gate (to transform its values between -1 and 1), see Figure 10-44 Step 3-2. As the third sub-step, it constructs the final output, h_t as $h_t = o \odot tanh(C_t)$, (see Figure 10-44 Step 3-3). The final output of each LSTM cell is *o,* along with its states (h_t and C_t).

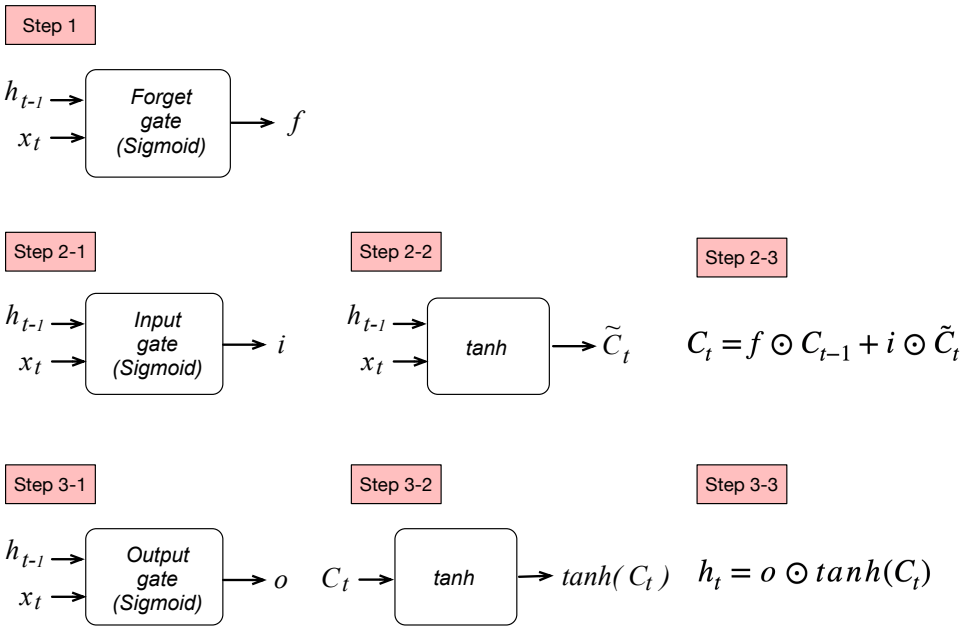


Figure 10-44: Summarization of the LSTM steps to construct short-term memory and long-term memory. Results of steps 3-2 and 3-3 are returned as outputs of the LSTM cell.

Figure 10-44 summarizes the process of the LSTM cell[16]. It is an easy-to-remember summarization of the LSTM algorithm. We did not explain each W and b, weight, and biases. There is another form of visualizing LSTM, which is presented in Figure 10-45.To be sure we understand them correctly, let's summarize their equations:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$

$$\tilde{C}_t = tanh(W_{i\tilde{c}}x_t + b_{i\tilde{c}} + W_{h\tilde{c}}h_{t-1} + b_{h\tilde{c}})$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}$$

$$h_t = o_t \odot tanh(c_t)$$

While using LSTM, if the states change in different layers (based on temporal changes), they are not significantly different, and thus, the gradient is unlikely to change drastically. LSTM can resolve the vanishing gradient problem, but it is still prone to exploding gradient.

[16] There are other nice visualizations, especially the one from Christopher Olah existed online as well. Links will be provided at the end of this chapter.

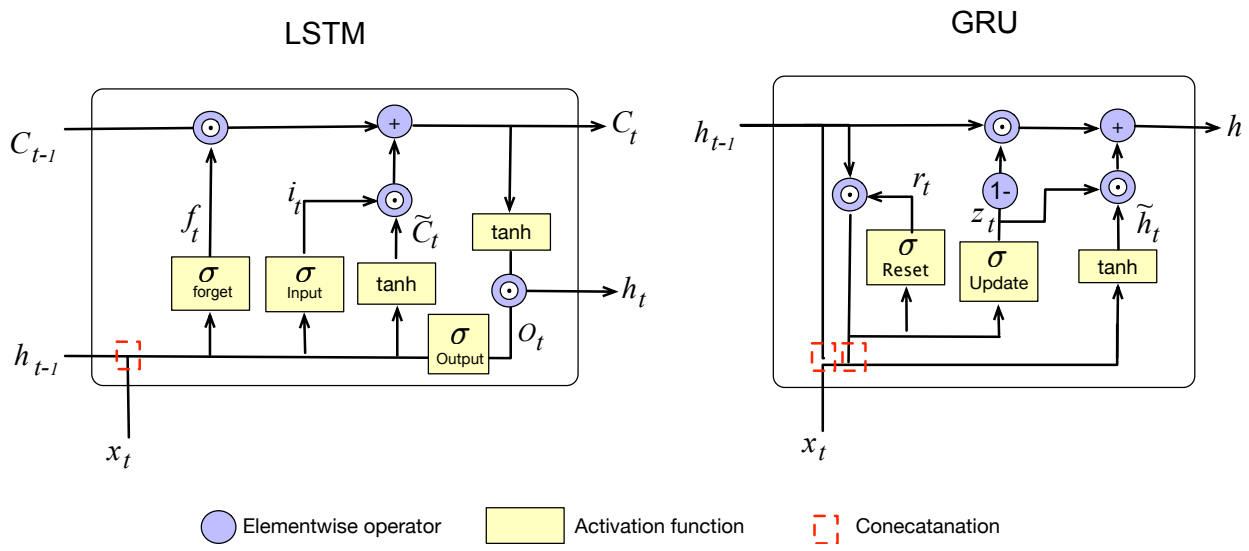


Figure 10-45: Visualization of LSTM and GRU architecture based on their gates and operators. In GRU output and new hidden state (h_t) are the same.

Gated Recurrent Unit (GRU)

Gated recurrent unit [Chung '14] is the more recent version of mitigating the vanishing gradient problem of RNN. GRU is less complex than LSTM and does not have a cell state. Since there is no cell state, its input (x_t, h_{t-1}) and output (h_t) parameters are the same as simple RNN.

GRU has two gates that use the Sigmoid function: *reset gate* (r) and *update gate* (z). The reset gate decides how much of the previous state to remember. The update gate is similar to the forget gate and input gate of LSTM, and it decides *how much information to throw away*, or in other words, how much new information is a copy of the old information. A reset gate is responsible for *short-term* dependencies of a given sequence, and an update gate is responsible for *long-term* dependencies of the given sequence. A GRU cell operates as follows.

(i) The input x_t and output of the previous time h_{t-1} are used to calculate the value for update and reset gates. They both use the Sigmoid function, and their equations are written as follows:

$$z = \sigma(x_t W_{xz} + h_{t-1} W_{hz} + b_z)$$

$$r = \sigma(x_t W_{xr} + h_{t-1} W_{hr} + b_r)$$

(ii) Now, with the result of both gates, we can calculate the *candidate hidden state*, but it is not the final hidden state. The candidate's hidden state should stay between -1 and 1 interval, thus, a *tanh* gate is required. Besides, it incorporates the reset gate information, which will be used as an element-wise product of the previous hidden state h_{t-1} and r. The candidate's hidden state equation is written as follows:

$$\tilde{h}_t = tanh(x_t W_{xh} + (r \odot h_{t-1}) W_{hh} + b_h)$$

(iii) After the candidate hidden value ($\tilde{h}_t$) and update gate (z) are both identified, the algorithm calculates the hidden state as $h_t = (1 - z) \odot h_{t-1} + z \odot \tilde{h}_t$. The output of GRU is only h_t , and unlike LSTM, it does not have a cell state.

Based on the above equation, if z is close to 1, the new hidden value (h_t) will be very similar to the old hidden value (h_{t-1}). This reveals that lots of the new information is a copy of old information. In contrast, z close to 0 indicates the new hidden value (h_t) will be close to the candidate's hidden state ($\tilde{h}_t$).

Figure 10-45 presents a common visualization approach used by these two algorithms, inspired by Christopher Olah's design. In the beginning, we did not describe LSTM and GRU with these visualizations because they might sound confusing, but now you can easily understand them. To

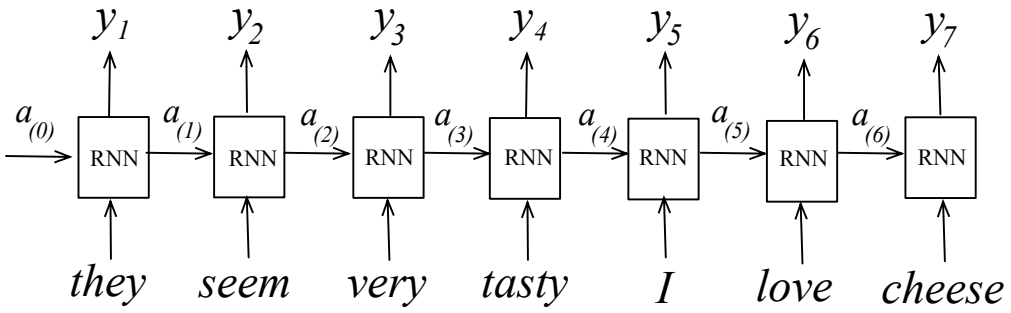


Figure 10-46: Unfold version of one RNN cell, which receives one word at each time unit. '*a*' presents activation function result.

be sure we understand it, take a look at the following summary of its equations:

$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr})$

$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{t-1} + b_{hz})$

$\tilde{h}_t = tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{t-1} + b_{hn}))$

$h_t = (1 - z) \times \tilde{h}_t + z_t \odot h_{t-1}$

Bidirectional RNN

One of the popular problems in NLP applications is referred to as *name-entity recognition*. It refers to a problem that we need to identify a word that refers to what entity (person, object, organization, etc.) in a sentence. RNN is a useful approach for named entity recognition.

For example, consider this sentence: "They seem very tasty". We do not know what "They" means. However, if we write the following: "They seem very tasty, I love cheese." Now, we can understand that "They" refers to food with cheese or cheese itself. Without that prior knowledge, we could not understand what entity the word "They" refers to. A simple RNN cannot incorporate prior knowledge before encountering the word "Cheese" because every new state got the information

from its previous state, and in this particular sentence, we realize they are referring to food with cheese later in the sentence.

Take a look at Figure 10-46, the unfolded version of one RNN cell (check Figure 10-41 to recall what we mean by unfolding). At each time unit, the RNN receives one word of the input sentence. In this figure, we present the activation result as $a_{(time-index)}$, which means that at time 3, the word "very" will be fed into the RNN, and the output of the RNN cell at that time will be y_3. Since the word, "cheese" happened at the t_7 and "They" at t_1, the RNN can not understand what "They" refers to at any time earlier than t_7.

A solution to handle this is to incorporate future information for better sequence prediction. Bidirectional RNN is designed to incorporate future data into the sequence as well. It introduces another layer with exactly the same size, but the direction of activations is in the opposite direction (from the end of the sequence to the beginning of the sequence). Figure 10-47 presents the Bidirectional RNN, and we use blue color to present the other direction. In this example, in addition to the forward RNN, the other RNN is going from the last word of the input sequence to the first word of the sequence, i.e., going backward in the sequence. In particular, the neural network of Figure 10-47 starts from $a_{(0)}$ and goes forward until it computes $a_{(6)}$; simultaneously, it starts from the $a'_{(0)}$ (blue RNN) and moves backward by computing activations until it reaches $a'_{(6)}$. Therefore, assuming our sequence has a size of T, every y_t output includes both forward (from 0 to t) and backward activation (from T to t) data (simultaneously). This structure allows the network to have information from both past and future states at any point in the sequence.

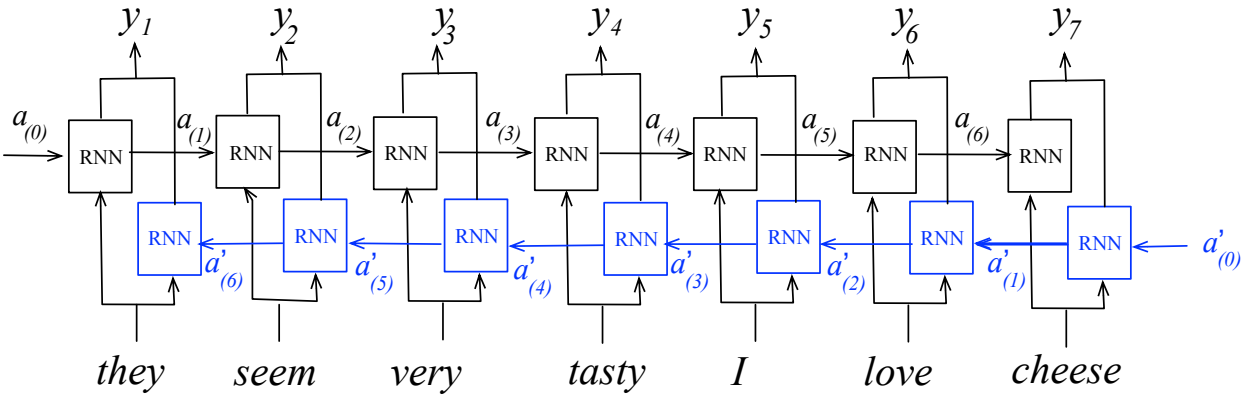


Figure 10-47: Unfold version of one BiDirectional RNN cell, which the blue layer is the same kind of RNN.

Of course, training this network is computationally intensive and slower than usual RNN, but it is a useful network to experiment with when we intend to incorporate future information into the current sequence. Another issue of Bidirectional RNN is that we need to have the entire dataset to train the model, and thus, we can not work with a stream of data (data will be available during the time).

Deep RNN

If we stack multiple layers of RNNs on top of each other, the result is a Deep RNN. This structure is employed to capture more complex patterns and dependencies in the data, which is especially useful in modeling higher levels of non-linearity. As illustrated in Figure 10-48, a Deep RNN can have several hidden layers, such as three in this example. While it is common to expect that increasing the number of layers in a neural network can improve its accuracy, this approach requires sufficient data and comes with trade-offs. Specifically, there is a limit to the

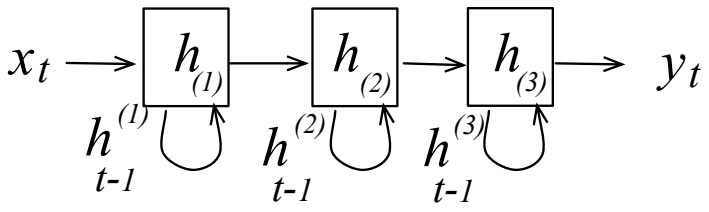


Figure 10-48: A deep RNN with three layers. Note, unlike previous figures, it has three hidden layers instead of one hidden layer that is unfolded.

number of layers that can be added before the model starts experiencing diminishing returns on accuracy, often due to overfitting or training difficulties such as the vanishing gradient problem.

RNN Examples

Unlike CNN, which we could easily understand its applications, RNN might require a bit more explanation. Therefore, we introduce a few examples that might help us better connect with the applications of RNN. More examples of popular RNN models will be explained later in Chapter 12.

We can use an RNN (LSTM, GRU, or any other model) to predict time series or other sequential information in the future. In the example presented in Figure 10-49, the blue part of the time series is fed as trained data, and the RNN constructs the red part of the time series. The given time series is a mock time series that shows the sales of a rubber duck. As we can see, the sales increase in summer and decrease in winter, when most of the planet Earth is cold. Therefore, customers are less motivated to purchase rubber ducks.

Also, RNN is used a lot in natural language processing applications such as machine translation, grammar correction, and text editing tools. For example, we can train it, and it can be used to construct sentences as well. Definitely, the larger the text size, the longer the sentence will be. Also, it could guess the rest of the sentence. For example, "I do appreciate a quick" could be followed by the word "response", which this word is guessed or constructed by an RNN. Another example is Part Of Speech (POS) tagging, which specifies the grammatical role of each word in a sentence.

As another example, take a look at the visualization proposed by Karpathy et al. [Karpathy '15]. We fed the "100 years of solitude" from Gabriel García Márquez as input, and RNN

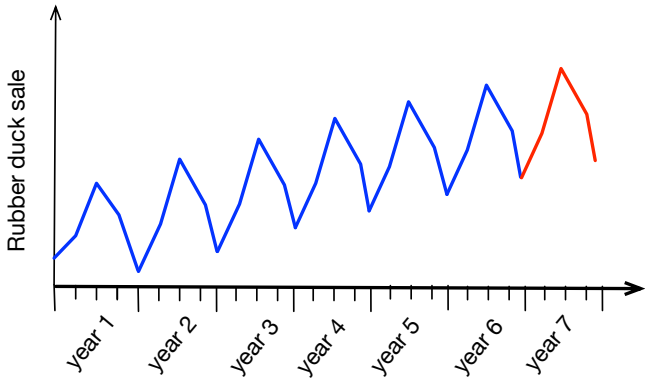


Figure 10-49: Another popular use of RNN is for time series prediction and here the red area is constructed by the algorithm, which has the blue part of the time series as its training set.

automatically learns text features, such as the beginning of a sentence, quotes, punctuations, etc. It is similar to feeding the image into CNN, and it learns image features such as edge lines. Figure 10-50 visualizes features that the LSTM identifies from the text. In this model[17], we have used Sigmoid as the activation function, and the color range from dark sky blue (0) to dark red (1) is used to visualize LSTM cell activations. Blue means it can correctly predict, and red means it has not correctly predicted. You can see this in this example of an output cell. The space and beginning characters of the next word have been predicted, especially the word "the". Thus, one application of LSTM could be a grammar checker, i.e., when the word "the" is not there, recommend adding the word "the".

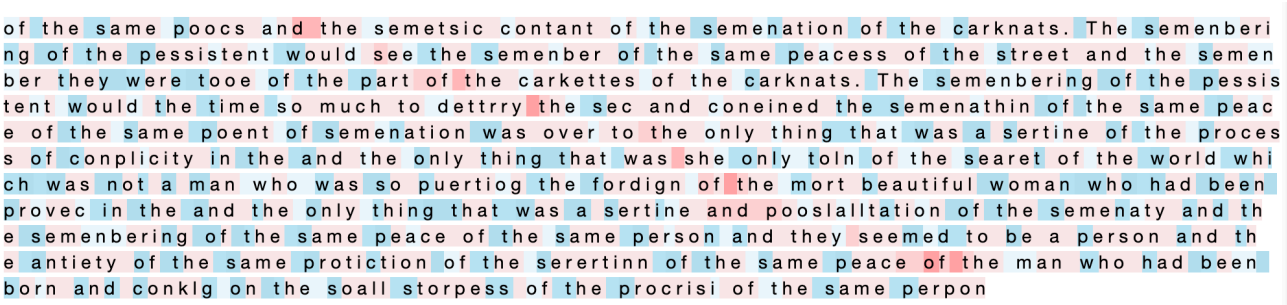


Figure 10-50: A sample feature (space between words and the character beginning of the next word) that has been detected by one output cell of the LSTM model.

Since the author of this book is poor and can not afford a strong machine with GPU, we trained it in a few epochs, but if we train it longer, we might get more human readable output. Note that there are many other features recognized by LSTM, but they are hard for us as humans to understand due to the black nature of deep neural networks.

In another example, we use a GRU to generate a text automatically by looking at the text of the "100 Years of Solitude" book and learning the patterns of the text. Following is a sample text created at the 30th epoch:

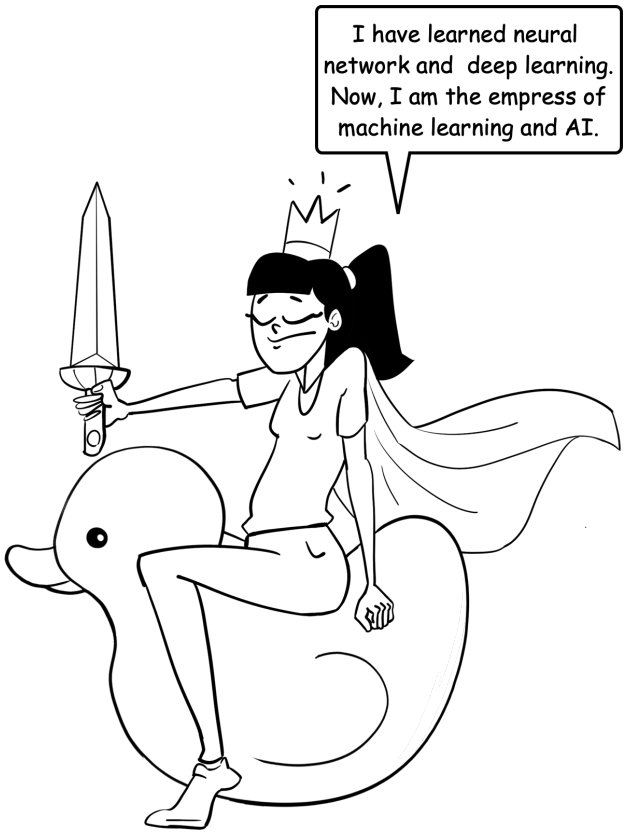
---

[17] The code is used from here https://github.com/praneet9/visualising-lstm-activations, but we apply some modications to make it run for the currecnt Keras version.

He plunged into his the one he had felt during the time he was resigned to living without a woman. He plunged into his mpose on the gone at section purbers of unerateradad difficult, and on adldo suck pestact rediract. Aureliano, and mort time to lession with phonessess had fanthoush alous the work he fate hoared with his it was a corple and troused a scorphing,s of a lecquiander some most memary and time that the cloth and that it wasred the mosting from darnesprotes. eaven who had becauged your arthioped his walk.

This text does not have a meaning, but it is very interesting, and you can see how these algorithms can be used to generate new information from existing information. Generating new information through the neural network is referred to as "Generative AI", and deep learning algorithms model the path to Generative AI. In later chapters, especially in Chapter 11 and Chapter 12, we will see more exciting examples of Generative AI.

NOTES:



* There are more variations of LSTM [Greff '16]. For example, peephole LSTM which enables *f*, *i*, and *o* to look at the cell state. We do not describe them here, and to the best of our knowledge, they are not popular frameworks.
* GRU is lighter than LSTM and thus might be more resource-efficient to use. However, whether one should choose GRU over LSTM cannot be generalized and depends on the specific task and dataset. It is recommended to experiment with both LSTM and GRU and make a decision based on comparing the accuracy of the results.
* The Bidirectional RNN and the Forward-Backward algorithm in HMMs share a functional similarity in that both incorporate future information to enhance sequence prediction. This feature allows for more accurate modeling of sequences by considering the entire context, as well as both past and future elements (refer to Chapter 5 for more on HMMs).
* While Bidirectional RNNs are computationally more intensive than unidirectional RNNs due to their processing of information in both forward and backward directions, the increased complexity can lead to better performance in tasks where understanding the entire sequence context is crucial.

Summary

In this chapter, we first describe some history and human efforts to resemble neural networks with computers. An artificial neural network is composed of three layers: input, output, and hidden layers. Each artificial neuron includes weight, bias, and an activation function that calculates its output.

Weights and biases are the foundations of neural networks. The optimizer in the neural network weights tries to find the best combination of weights that results in the lowest loss score. The objective of a cost function in a neural network algorithm is to evaluate the output correctness based on the loss score. If it is not correct, then modify its values by using an optimizer to adjust weights. As we have explained, weights are assigned randomly; they should be adjusted, and this will be done through the backpropagation algorithm, which goes back from the output of the network to the input neurons. Backpropagation splits the error, proportional to the weights back using the chain rule of the differential equation.

Next, we explained the perceptron, its limitations, and how Multi-Layer Perceptron (MLP) resolves its limitations. MLP algorithms are still in use and referred to as traditional ANN. If we increase the hidden layers of MLP algorithms, they are called deep neural networks. All neurons are connected to all neurons in the next layer of MLP. Such a connection is not mandatory in deep learning models, and if neurons of one layer are connected to all neurons of the next layer, they are called the “dense layer” or “fully connected layer”.

Afterward, we describe activation functions, which can enable the neural network to construct a non-linear model, followed by the cost functions explaination, which are mostly comparisons between the statistical distribution of predicted data and actual data.

Then, different types of Gradient Descent based optimizers have been explained. At the time of writing this book, the best optimization approach is using Backpropagation, which operates based on chain rule in derivations.

We concluded the general discussion about neural networks by regularization methods, including weight initialization, Gradient Clipping, Batch Normalization, Dropout, and Early Stopping.

CNN models are one of the most popular uses of Deep Learning because of their application in computer vision. To understand CNN models, we explained the concept of convolution and cross-correlation. Then, we described some concepts related to applying a convolution, including padding, stride, window size, etc. After the convolution, a CNN network is usually applying a pooling layer. The pooling layer is used to downsample many convolutions of the input image. The result of the pooling layer is sent into a flattening and then a dense layer for output preparation.

Since convolutions are very popular not just for computer visions but also for other tasks, we listed some of the common convolutional approaches, including 3D, dilated, and transposed convolution.

RNN is another common deep learning architecture used for sequential data modeling, such as time series, natural language processing, etc. The basic models of RNN are pruned to vanishing

gradient and exploding gradient problems. LSTM has solved the vanishing gradient issue and implements short-term and long-term memories. Each LSTM cell has four memory components**:** long-term memory, short-term memory, new long-term memory, and new short-term memory. Despite its usefulness, LSTM is complex and computationally inefficient. GRU, later, is introduced, which is less complex than LSTM. However, it is better to decide on the architecture with experiments and take into account the nature of the dataset that is used for the application.

To review LSTM and GRU operations, the following list of all their equations[18].

LSTM:

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$
$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$
$$g_t = tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg})$$
$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t \odot tanh(c_t)$$

GRU:

$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr})$$
$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{t-1} + b_{hz})$$
$$\tilde{h}_t = tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{t-1} + b_{hn}))$$
$$h_t = (1 - z) \times \tilde{h}_t + z_t \odot h_{t-1}$$

In these equations, W refers to weight and b to biases; their index is used to separate them from each other and explain the weight or bias belongs to which stage, $\odot$ presents a Hadamard product. h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t. For the LSTM, i_t is the input, f_t is the forget gate output, g_t is the cell gate output and o_t is the output. For the GRU the r_z refers to reset gate, z_t refers to update gate, and $\tilde{h}_t$ refers to candidate hidden state. To have a better overview of the differences between LSTM and GRU, you can check Figure 10-46 as well.

At the end of this chapter, we described “Bidirectional RNN”. Bidirectional RNN has the advantage of learning from both directions of a sequence, but it is slow. Nevertheless, it has useful applications such as speech recognition, part of speech tagging, etc. When we stack more than one layer of RNN cells on top of each other, this is referred to as “Deep RNN”. It also has applications in machine translation and speech recognition.

---

[18] Pytorch tutorial has a good summarization of both and we use them to build this table.

Further Reading or Watching

* Tareq Rasheed [Rasheed ’16] has a short book about neural networks and makes a very good connection between biological and artificial neural networks. It also guides the reader to build a small neural network.
* Francis Chollet [Chollet ’18] has an excellent introduction to deep learning and its components in his book. Chollet is the original author of the Keras platform[19].
* There are many explanations about the Backpropagation algorithm available, but the Aurelin Geron [Geron ’19] book provides a good explanation of this algorithm. Besides, Andrew Ng and Kian Katanforoush's notes are among the good ones that we use to understand them. http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf.
* Jay Alammar provides a very good interactive tool for learning linear regression and Gradient Descent. There, we can observe changes in the neuron (linear regression) by playing with *w* and *b* parameters. http://jalammar.github.io/visual-interactive-guide-basics-neural-networks
* If you are interested in staying updated with the Gradient Descent optimizer, John Chen has a good analysis of recent Gradient Descent algorithms on his home page https://johnchenresearch.github.io/demon. Also, Sebastian Ruder benchmarked some Gradient Descent algorithms, and you can observe the result: https://ruder.io/optimizing-gradient-descent/index.html.
* We do not go into the details of Convolution, if you are interested in learning more, Wikipedia has very good visual examples https://en.wikipedia.org/wiki/Convolution, if you are lazy, to read the content, we encourage you to check those images.
* There is a good resource by Dumoulin and Visin [Dumoulin ’16] if you are interested in learning the mathematics of Convolutions in more detail. They also have very good visualizations, which can be seen in the animation from here: https://github.com/vdumoulin/conv_arithmetic
* Christopher Olah has a fantastic and easy-to-understand, visualized explanation of LSTM. We benefited a lot from his explanation to understand LSTM. His explanation and visualization are available here: https://colah.github.io/posts/2015-08-Understanding-LSTMs
* There are courses that provide a good introduction to Deep Learning on Udemy. Two good ones are the one by Eremenko et al. (https://www.udemy.com/course/deeplearning) and Jose Portilla (https://www.udemy.com/course/complete-guide-to-tensorflow-for-deep-learning-with-python). However, the material they cover is not very broad, so consider them a start to your learning journey.

---

[19] https://github.com/keras-team/keras