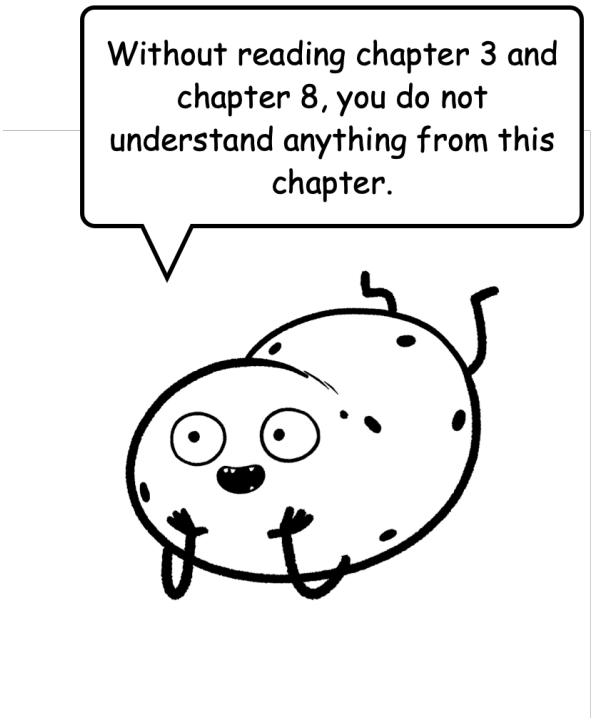
Chapter 10:

Artificial Neural Network & Deep Learning (Supervised Algorithms)

Probably, you are not passionate enough to read the book from the beginning and start it from here. That is not good, but it should not be a big problem. You need to listen to the Good News Potato feedback before you start reading a chapter and be sure that you have read all required chapters before.



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 in other disciplines.

In this chapter, we start by explaining the rationale behind deep learning and then the function of a biological brain. This helps us to explain artificial neural networks easier 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, supervised deep learning models and architectures including CNN and RNN will be explained.

Why Deep learning is both popular and powerful?

At the time of writing this Chapter (late 2020), we are in the AI hype and one of the hardest jobs of academic supervisors is to push away over motivated students and staff from focusing too much on neural networks and deep learning, dedicate their time to learning other concepts and technologies as well. This is our motivation to postpone the deep learning and neural network to Chapter 10 of this book as well. There are more important things that we should learn before starting to learn deep learning.

There are three reasons that deep learning is very popular. The first reason is its superior *accuracy*. The second reason that makes deep learning saliently superior to other machine learning algorithms, 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, their hidden layers handle all features. If a feature is not contributing to the prediction its weight will get reduced and after some iterations, this feature will be removed. This is very exciting for data scientists, especially for ones who are joining data science from other disciplines than computer science because the hassle of preparing data for the algorithm will be removed. Nevertheless, in many cases, you still end up removing unrelated features and feature engineering, because too many unrelated features cause a vanishing gradient and thus make the model unnecessary complex, and thus inaccurate. More about vanishing gradient will be described later.

However, there is "no free lunch", and the process of automatic feature extraction is associated with a huge cost, which is intensive computation. Therefore, scientistic move their calculation from CPU to GPU (Graphical Processing Unit), and the popularity of GPU, lead NVIDIA to launch the first platform for coding for GPU, i.e. CUDA[1], and later similar tools come into the market such as OpenCL[2].

The third reason is their *capability to work with unstructured 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 is working very well for unstructured data such as image, audio, and raw text 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

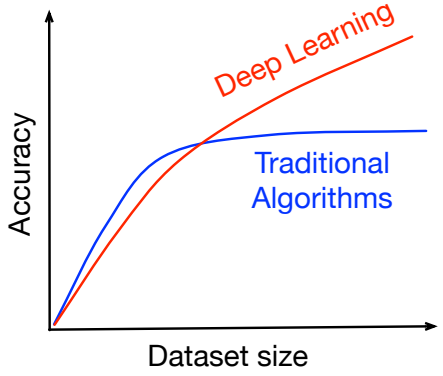
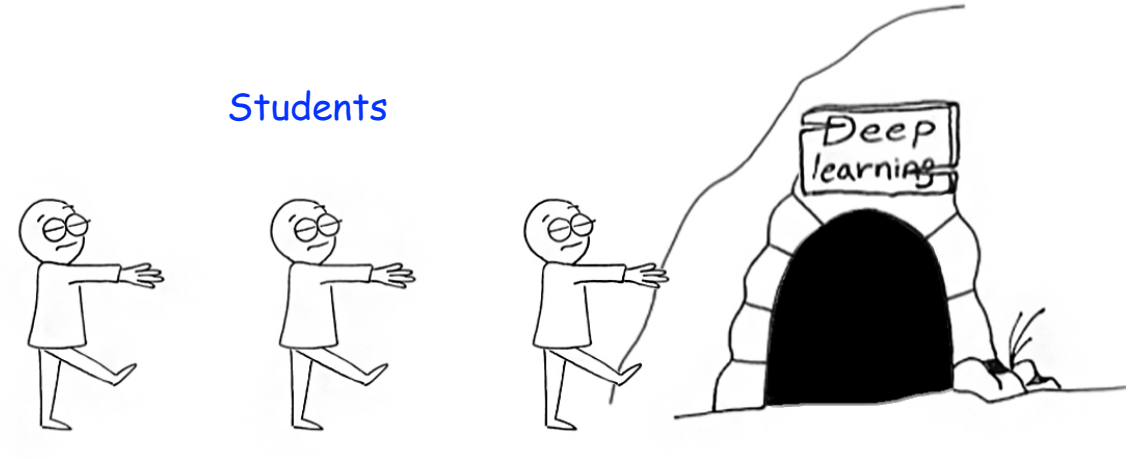


Figure 10-1: Differences between deep learning algorithms and traditional machine learning algorithms

[1] https://developer.nvidia.com/about-cuda

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

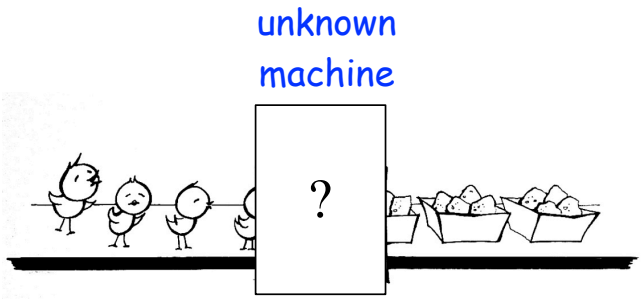
working with these datasets.



Although neural networks and especially deep learning algorithms outperform all other machine learning algorithms, they operate in a condition 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 ones. 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

There are some theories why deep learning is so powerful and provides significantly high accuracy in comparison to other algorithms?



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* and 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 produces nuggets as output. The process of supervised machine learning is to identify what is this unknown machine (function).

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 could call deep learning methods as *non-linear function approximation methods*. We will learn more about non-linearity later.

Another important concept we should be familiar with before learning neural networks is *Universal Approximation Theory.* It is a mathematical theory that indicates that any continuous function could be approximated by a neural network. In simple words, given enough amount of training data, there is a neural network that can mathematically model anything, and thus neural networks have a kind of universality. Nevertheless, the network might get too large that no computer can handle it. Keep in mind that *theoretically everything in nature could be modeled mathematically, as a combination of linear and non-linear functions*. Therefore it is not wrong to say that a neural network can model everything, but requires a significant amount of data as well.

Biological Neural Network

The brain of animals and humans is responsible for thinking and it is used to learn 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.

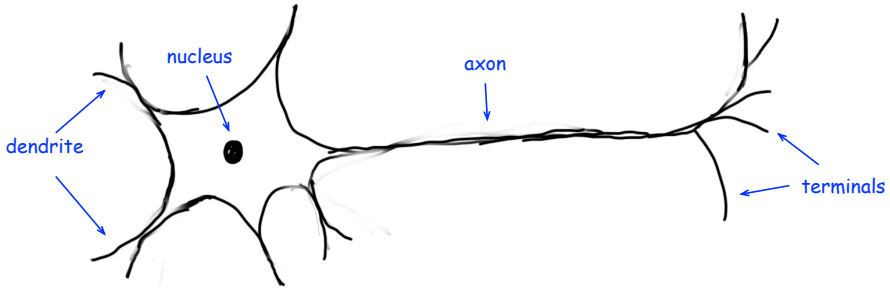


Figure 10-2: A simplified shape of a biological neuron and its components.

Neurons transmit an electrical signal from dendrite to terminals through axons. Then the signal passed between neurons with the same structure. Therefore, a simplified structure of the brain could be shown as a set of neurons that passes the electrical signal along with each other. Figure 10-3 present several neurons where signals are transferred between them.

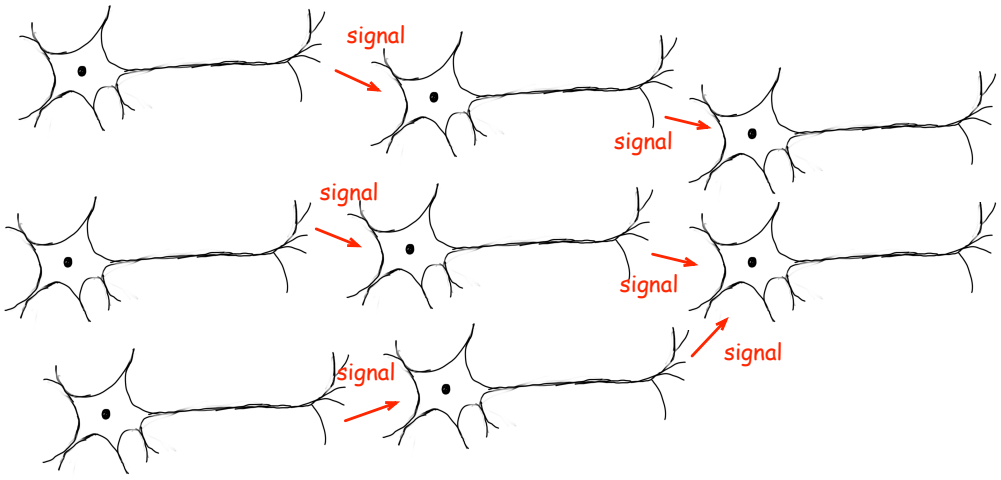


Figure 10-3: A simplified version of brain and how neurons transfer electrical signal.

The process of transferring electrical signals through neurons is called firing. When a neuron fires a signal? Scientific evidence suggests that a 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. This makes sense because noises, which are electrical signals less than the threshold, are not transferred, and only information which is useful will be transferred. Let's review what we have explained, a biological neuron gets a set of signals, processes them, and if the output reaches a threshold, fires another signal.

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

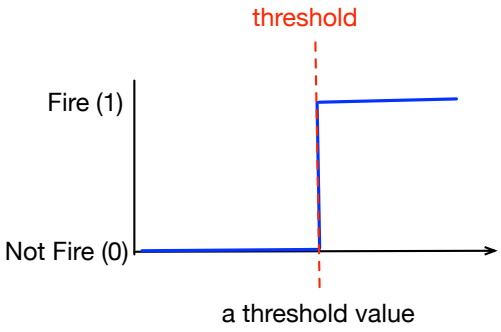


Figure 10-4: A neuron is not firing a signal until the signal reaches a threshold. Then it will get fired. This is called threshold function as well.

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

Now you have learned how the brain of animals works, and you can go and stay in front of the mirror and grant yourself a full professorship in Neuroscience. Please accept our humble congratulations on your achievement, but also continue reading the rest of this Chapter.

Artificial Neural Network

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

Figure 10-5 shows a simplified neuron and a simple artificial neuron (perceptron). Intuitively, you can observe and realize the similarity of the perceptron to the biological neuron.

An artificial neuron is a mathematical model that has a set of inputs (similar to biological dendrites), each input is associated with a weight plus a bias. Thus, we can write them as $x_1w_1 + x_2w_2 + \ldots + b$. Then, the activation function calculates the value and sends the result to the output. Activation function could be written as a, and thus we have $a(x_1w_1 + x_2w_2 + \ldots)$[3].

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 perceptron by weights. To better understand the need for weight, consider the prediction error is written as *error* =

[3] 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.

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 that have been used in regressions back in Chapter 8.

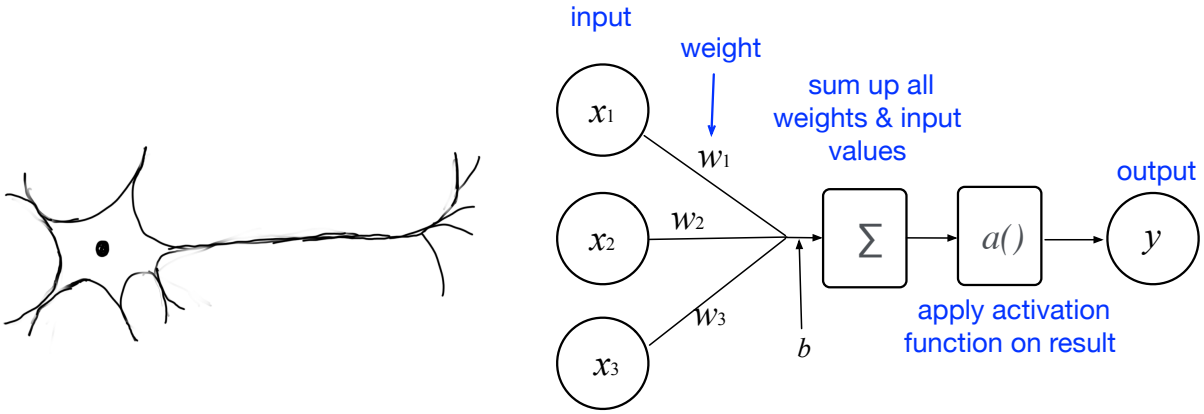


Figure 10-5: (left) a biological neuron, (right) a simple perceptron which its architecture is inspired by biological neuron.

Usually, weights are initialized with a random value from 0 to 0.3. There are different weight initialization methods (at random, with all zeroes, with small values around 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 as zero 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 are used to adjust the strength of the connections between neurons. Biases, on the other hand, are used to 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. It could be a linear or non-linear function. Linear here means that there exists a hyperplane that can be modeled 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 to recall) are more popular.

There are different choices to be used as activation functions, which we will describe later.

Now, that we have understood these concepts, we could say that an artificial neuron is nothing more than linear regression and that its result is fed into an activation function, $y = a(wx + b)$.

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

Input layer is the input variable(s) that we feed into the neural network. Often each feature is presented as one input neuron. For example, a table that has 10 numerical columns will have 10

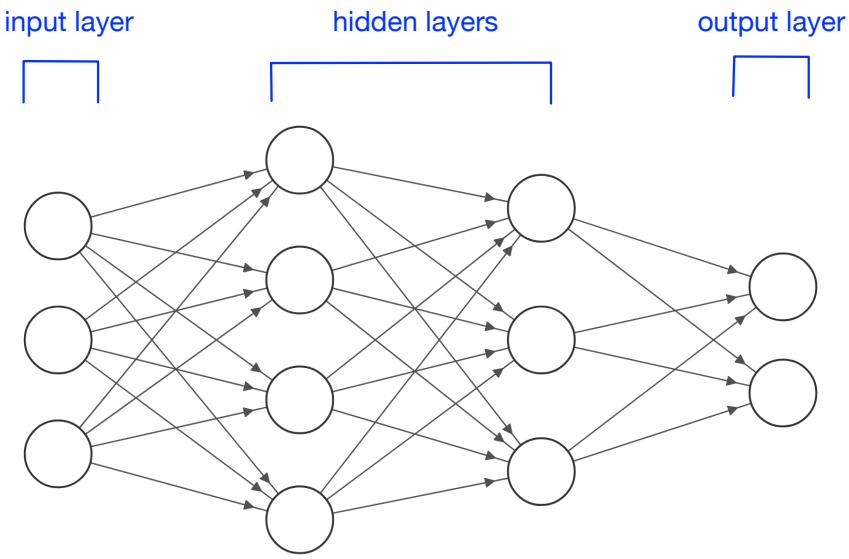


Figure 10-6: A simple presentation of neural network and its layers. You can realize it is also look a like Figure 10-2 but with artificial signal.

input neurons, and they are independent variables, e.g., one row of a table present 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, keep in mind that the only input type we can use for the neural network is *numeric,* and other data types, such as text, cannot be fed into a neural network. If the data type is different, we can use feature engineering methods we have explained in Chapter 6, such as one-hot encoding, to convert them into a numerical format. 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 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 combined, and construct the neural network. Each layer is responsible for transforming the received data for the next layer. Since they are transforming the data and are different from the input or output that are known, we cannot interpret hidden layers, therefore they are black boxes and called hidden layers. 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 that occurred in hidden layers.

Output layer is outputting the result of the neural network algorithm. 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 it.

There are three common tasks that 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 can have a single output, which provides zero or one or a probability. The result can be 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 can have multiple output neurons.

Now that we learn what is a neuron, and what the layers of the neural network are, let's turn to the rest of the explanations about neural networks. The objective of a neural network algorithm is to assign a correct label to the unlabelled data. This objective will be achieved by *assigning proper values to weights and testing the network, then, reconfiguring weights and biases and testing the network. This happens several times until the neural network provides satisfactory accuracy*.

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 measured by the **cost (objective) function**. We have described the cost function back in Chapter 8, and we use that definition here as well. To summarize: *the cost function is used to measure loss in the neural network*. The neural network algorithm identifies a loss score for each neuron. Then, after each iteration, it reconfigures the network's weights and biases to reduce the loss score of each neuron. This process of reconfiguring the network's weights and biases will be done by the **optimizer**, 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, we can say a neural network has three core components, **layers**, **cost function,** and **optimizer**. 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].

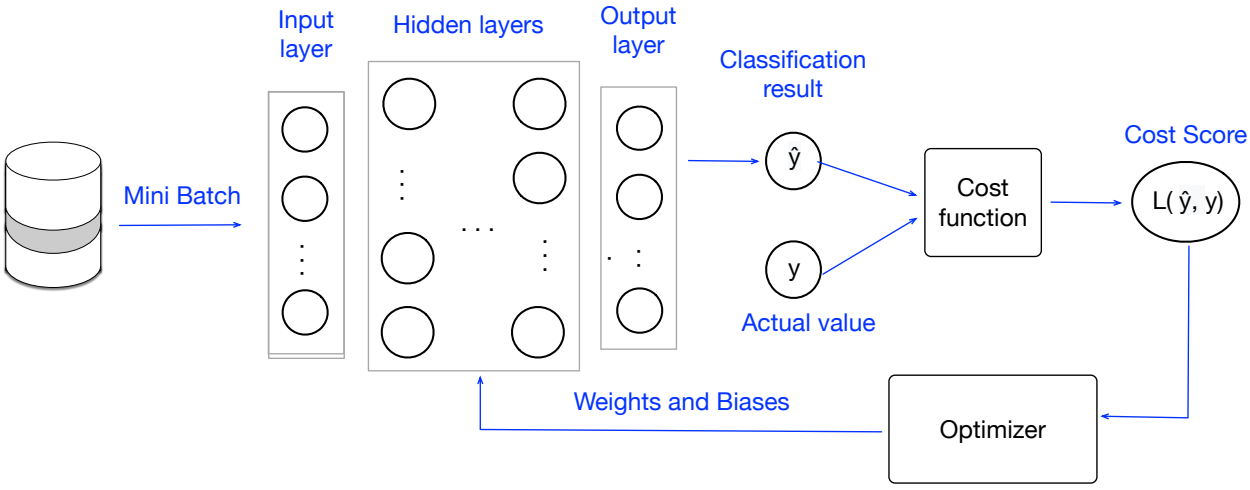


Figure 10-7: Artificial Neural Network flow of operation. The output value will be analyzed by the cost function and the loss score is calculated, then based on the loss score, the optimizer reconfigure weights of the neural network.

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.

NOTE:

* We do not report the computational complexity of neural network algorithms, 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 input variable and output variable in a neural network are presenting the same observation, but the output has a label assigned to the same information.
* A neural network is the chain of differentiable (differentiable means we can get derivative from it) tensor operations. The process of learning 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, $F(X) = f1(f2(f3(x)))$, each of the f functions could be a vector function, $f_l(Z) = a(xw + b)$, a is the activation function, l is the layer index, and it could span from 1 to any number of layers.
* 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 and identify a y. This y will be fed into an activation function 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

Now we understand that a neural network algorithm assigns the weights, runs the network, measures the cost function, and then updates the weights to reduce costs. We have explained cost function back in Chapter 8. Please be sure you are familiar with the concept of cost function before continue reading the rest of this chapter.

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 d number of features $(x_1, x_2, \ldots x_d)$, the following equations can be used to 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 *weights 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, weight impact will be zero as well. To handle this problem, a bias b_i will be added to the input as well. In other words, 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.

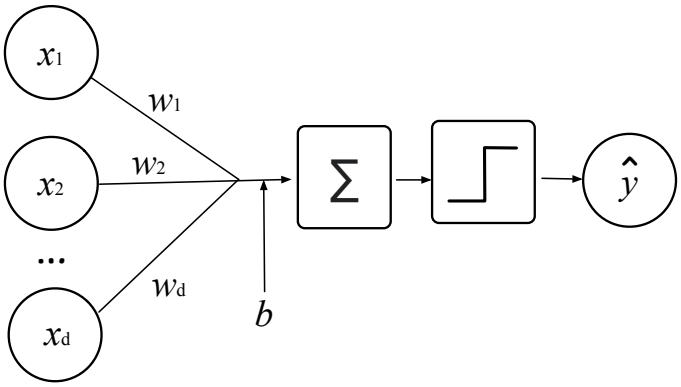


Figure 10-8: A single perceptron which receives a *d* dimensional input data. *b* presents the bias which is added to the result of summation. The first square is summing all weights and inputs together. The second square presents the activation function.

Figure 10-8 shows the simplest form of the perceptron. Σ is presenting a summation of input variables their weights, and biases ($\Sigma_{i=1}^{d} x_i w_i + b$), and the other rectangular box (with S shape figure in it) is presenting 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, $a(\ )$. In particular, the **activation function** is used to set constraints on the output values of neurons. If we use Sigmoid function (check Chapter 8 to recall Sigmoid) activation function it will be a logistic regression and a single perceptron is just a logistic regression (if its activation function is Sigmoid). Single perceptron is not very useful, and we use Multilayer perceptron, which we will explain later.

Perceptron (similar to regression algorithms) operates based on the assumption that there is a hyperplane, which separates two sides of the dataset from each other. It starts with initial weights that are zeros. Then it classifies the dataset. Next, it goes back and checks the data points that it has been misclassified by analyzing the cost of the predicted value ($\hat{y}$) which is acquired by comparing it to the actual value (y). For example, we can use J as a cost function and it is written as $J = 1/2(\hat{y} - y)^2$. Then the neural network changes the weight of misclassified data points to reduce the cost function, J, and again performs the classification and then checks the new result. This process continues until it reaches a specific number of iterations or weights cannot be further changed. 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.

Multilayer Perceptron

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. Logical reasoning plays an important role in electrical engineering, computer science, and circuit design, but for our need just knowing these operators are enough. Here we have two variables.

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 logic operation for two values

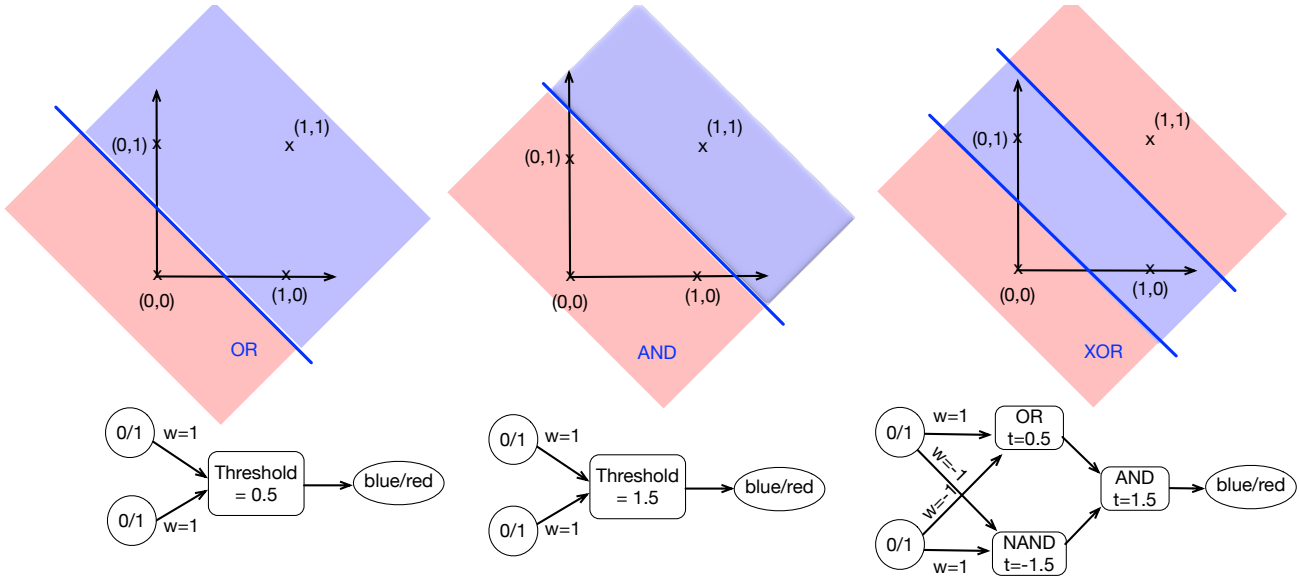


Figure 10-9: (Top) Four dots represents different status of A and B. AND and OR could be separated with linear hyperplane but XOR is not separable with a liner hyperplane. In the right diagram we need some non-linear function, to separate the blue area from the rest. (Bottom) Both AND and OR can be implemented as a perception with the step function. However, to implement XOR we need multiple layers of perceptron. Blue is 1 and red is 0.

We have explained single layer perceptron. It 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 XOR[4] boolean function. In other words, it is impossible to use it for a non-linearly separable dataset [Minsky '69] with a single perceptron. However, 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 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.

All these perceptrons use a *step function* as their activation function. 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 you 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 it requires at least two connected layers of perceptrons. This means the output of one layer will be fed as the input of another layer. This neural network which has more than one perceptron is called a multilayer perceptron, and a multilayer perceptron can handle non-linear separable data as well.

In other words, **Multilayer Perceptron (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

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

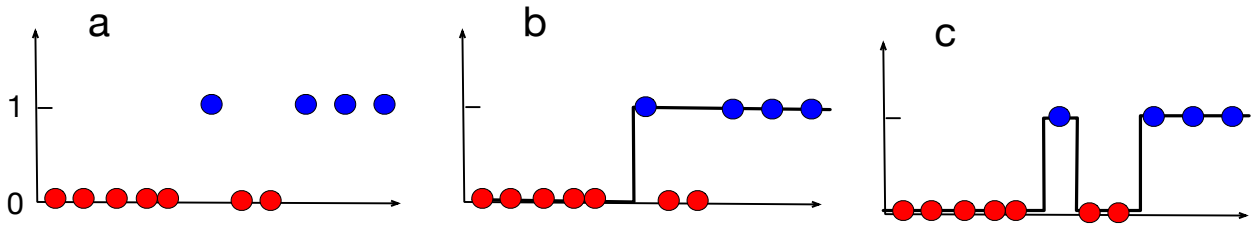


Figure 10-10: (a) original dataset (b) using one single perceptron with a threshold activation function we can classify most data points correctly. However, two red dots are misclassified. (c) By using MLP with threshold activation function, all data points were classified correctly.

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 layer we add, the more complexity can be handled by the algorithm. This doesn't mean that too many layers are always good. Too many hidden layers impose a huge cost on resources and could also cause overfitting.

To be sure you leave this section with a Ph.D. 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 one step function, and it can fit these data points as it is shown in Figure 10-10 b. The result can classify blue dots as 1 and red dots 0, but there are two red dots that are misclassified. Using two step functions, which is 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.

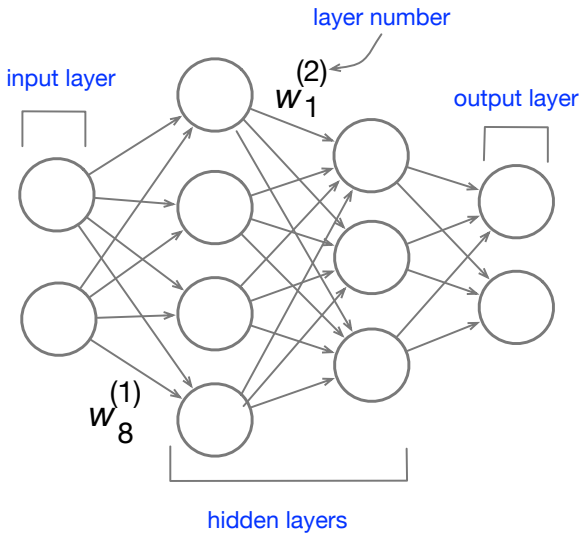
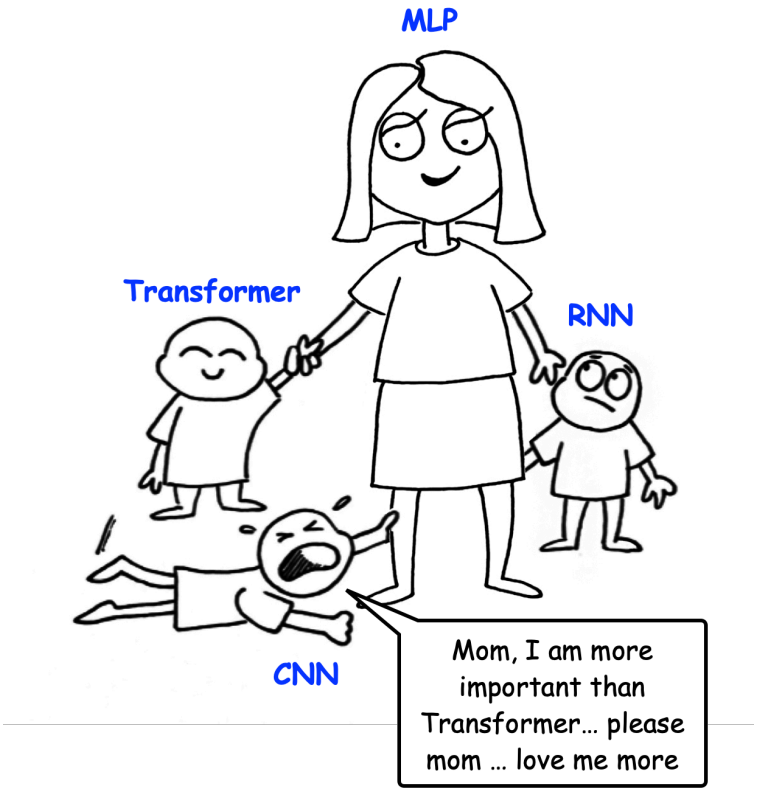


Figure 10-11: an MLP network, which all neurons are fully connected to all other neurons in the next layer.

This means that every neuron is connected to all other neurons in the next layer as it has been shown in Figure 10-11. Also keep in mind that every layer, except the output layer, includes

biases as well. Check Figure 10-11 to see how we write weights, on the top of a w its layer number is written and at the bottom, its index in that layer is written.



All other neural networks originated from MLP. Later we learn three common categories of neural networks, but they all originate from MLP.

Activation Functions

Activation functions are mathematical equations that are 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 which changes z to the final output for a neuron, i.e., $a(z)$.

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 appeared 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 which is used for binary classification. It has a similar shape to the function presented in Figure 10-4, but

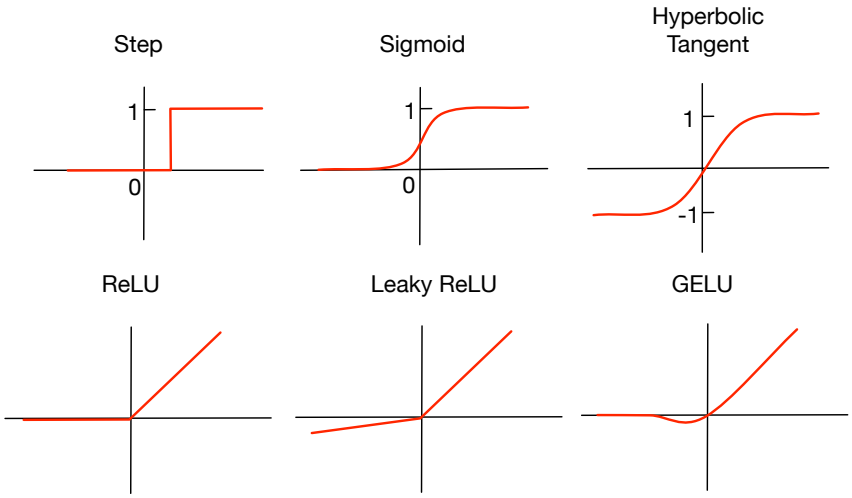


Figure 10-12: Some popular Activation functions.

the threshold is z and if it is larger than zero the signal will be fired ($\hat{y} = 1$) otherwise, it will be 0 ($\hat{y} = 0$).

Sigmoid function: It is more smooth than the step function and it 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.

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. When we do not like to have zero as the lower boundary of the activation function, this activation function can be used.

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, which 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*. 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 by 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. drop out. GELU activation has this characteristic as well and some weights will be multiplied by zero. 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 in Chapter 12.

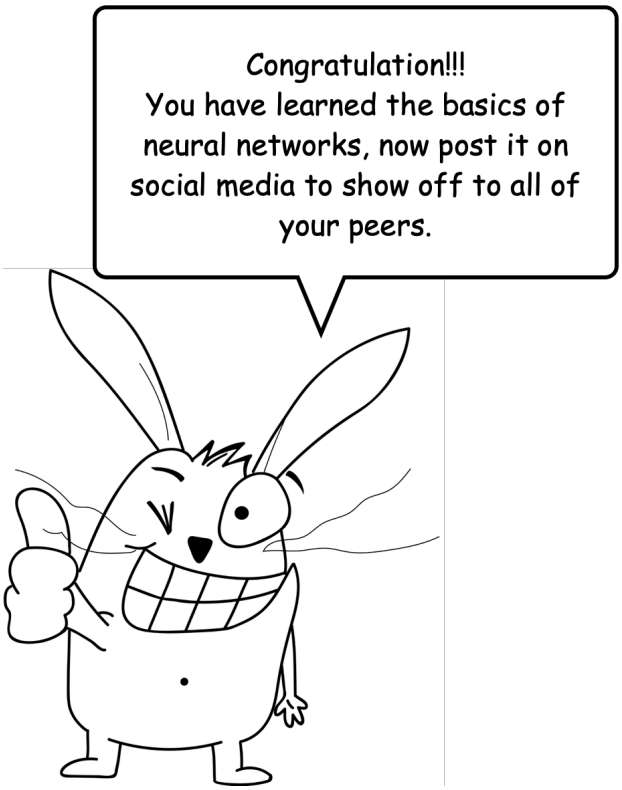
Softmax: It is an activation function that can be used for multiclass output values. We cannot visualize it in Figure 10-12, because it has multi-dimensions (equal to the number of output classes). Assuming we have k number of classes, the equation to calculate the softmax is written as follows: $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad for \quad i = 1,...,k$

Softmax activation function calculates probability distributions of the particular output over k different classes of 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. Assume 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. Our algorithm analyzes an image and provides the following output probabilities for infection: X : 0.3, Y:0.6 , and Z : 0.1 You 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 are

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.

You might ask why do we need an activation function? Neural networks are linear transformations and chaining many linear transformations ends up having a linear transformation again and by just having a linear transformation the neural network cannot solve complex problems. 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.



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 is the output*. In other words, after the neural networks construct the output the algorithm uses to 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.

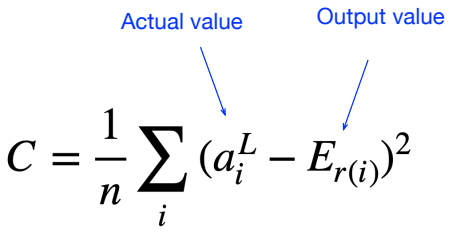
A cost function for a single neuron is written as $C = (W, B, S_r, E_r)$, in which C presents the cost value, W presents weights, B presents biases, Sr presents the input of a single training sample (e.g., one record of data), and E_r presents the desired output of the given input training sample (Sr).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

Note that when are reporting something specific to a layer we use superscript. For example, $a^{(2)}(x)$ means 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 again because they are widely in use for neural networks and it makes sense to list them here. We can also define our own cost function based on our needs and we could incorporate the domain knowledge into the cost function as well.

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 real value of output, $a^L(x)$ 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 datapoint, the quadratic cost equation is written as follows.



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

We know using simply y and $\hat{y}$ is much easier to understand and memorize but other literature do use these notations, it is better to get familiar with these notations as well, and 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 that is 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 the bed can provide a list of three probabilities as follows: {don't talk with me: 0.7, full of energy: 0.1, urgently to the restroom: 0.2}. Assuming the data points in the test dataset are specified *i* index, its equation is written as follows.

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

The description of 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 have 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:

$$C = \sum_{i} E_{r(i)}\ log \frac{E_{r(i)}}{a_i^L}$$

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 follows, which is a L_2 norm (Euclidean norm) used for probability distributions:

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

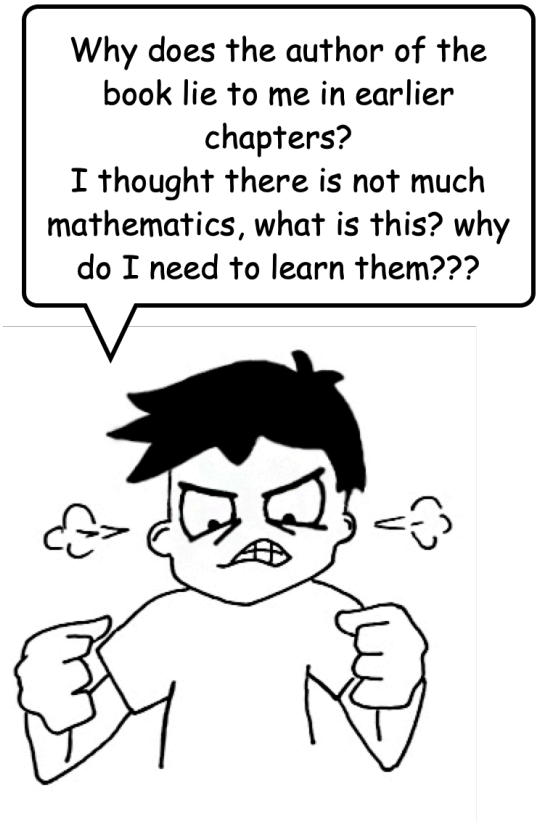
However, differentiating a square root inside a square root (used by Backpropagation) is computationally complex and this distance can be relaxed by removing the square root as follows:

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

If we intend to ensure that the cost function result will stay between 0 and 1, we can use this distance.

Cross entropy and all other cost functions we described here, except quadratic cost, are *measuring the differences between two probabilities, in which, one probability presents the actual value and the other one presents the output value*.

To summarize when to use which cost function, check the following. if we deal with a regression problem, we use the quadratic cost function. If we deal with classification, we can use Cross Entropy, KL-Divergence, or Hellinger distance. If we deal with binary classification, we can use Binary Cross Entropy.



NOTE:

* Since a neural network includes many neurons and there are too 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 equation we described for cost function, the quadratic cost function is a popular one. Its squared is done to get rid of negative values and increase the impact of errors to highlight them. Somehow, it punishes the network when the error is

huge by making it a larger error. If we have multilabel classification cross entropy is also popular. Nevertheless, we recommend if you have enough resources experiment with all of them. Your neural network software will get them as hyperparameters.

* There are more cost functions such as Generalized Kullback–Leibler divergence or Exponential cost function which we did not describe them.

* 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, usually mean absolute error and mean square error do not yield good results. Therefore, they are not popular fo be used for cost function in neural networks.

Neural Network Optimizers

Before starting to read this section, it is worth taking a look back to Chapter 8 and reviewing the optimization section we have explained. In Chapter 8 we have explained, that the cost function measures *how far is the true value 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 by 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 changing weights and biases the algorithm can reduce the loss score. Now a question arises, what is the best combination of weights that increases the result accuracy? To answer it we can try every possible combination of weight 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 have $(4 \times 5) + (5 \times 5) + (5 \times 2) = 55$ weights and biases to configure. 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^{55} possible combinations for experimenting and identify which combinations lead to the best minima. This 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.

We need a subtle approach to determine weights and biases. Take a look at Figure 10-12. There we have a very simple convex function. This figure presents the differences between using optimization and not using optimization to find the minima. We can see by using blue dots in the

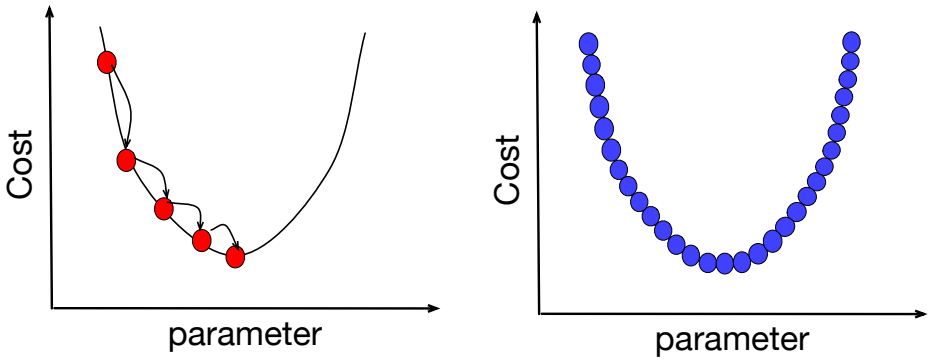


Figure 10-12: (left) using an optimization to reach the minima and therefore very few data points will be processed. (right) not using an optimization and thus many data points will be experimented on the function to find the minima.

right figure, that we try to show that there are too many data points to experiment with, but in the left figure there are few red dots and instead of experimenting with any possible value the algorithm can make some jump 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 which 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 derivative, which is shown on the left side of Figure 10-12. In particular, at each point, the Gradient Descent algorithm calculates the deviation, if the gradient sign did 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 on the opposite direction to reach the minima.

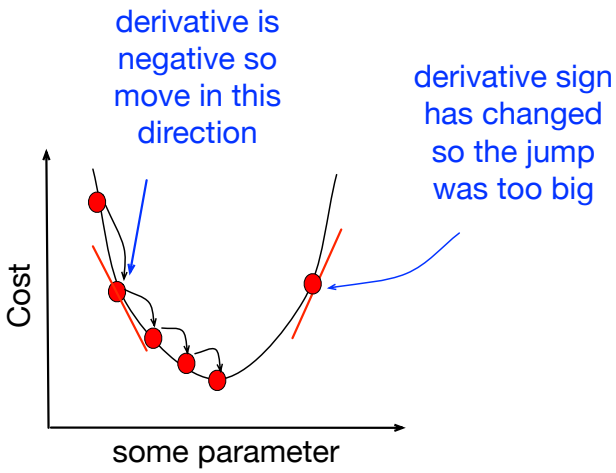


Figure 10-13: Gradient Descent decides the direction of the next point to move toward minima.

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 parameter and the *Y* axis presents the cost of different values for the parameter.

However, having constant size jumps is not efficient, 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. Therefore, a better approach is customizing the step size and this will be done through "*decaying learning rate*". Also if the jump is too big, then the gradient sign will change.

Back in Chapter 8, we have explained that there are three types of Gradient Descent including SGD, BGD and Mini BGD. Gradient Descent until now is the most popular optimization approach that is being used 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. We introduce other optimization algorithms that are using SGD, mini BGD and improve it.

Stochastic Gradient Descent with Momentum

Although mini batch Stochastic Gradient Descent (SGD) has some limitations, it is a very popular optimization algorithm that is used in deep learning as well. In Chapter 8 we have

explained that, while using Gradient Descent, the next value for the parameter will be 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 gradient function. We estimate a value of w_{t+1} via this 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 we have 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 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 first, then we describe SGD with momentum.

Exponentially Weighted Average: Weight *(β)* will be a number between 0 and 1, which is usually set to 0.9. Multiplying a number less than one to another number makes it smaller. For

[5] While reading text for optimization in the content of neural network some literature refers to parameter 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.

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 to 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., $\{s_1, s_2, \ldots s_n\}$ by 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 can have the following:

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

We can see that as the data gets older in the series, its impact on the equation gets smaller because more numbers of βs will be multiplied by it. 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 our intention is to get the ball into global minima in Figure 10-14. Derivatives present the acceleration of the moving ball, and momentums add to the velocity of the moving ball (v or *retained gradient*).

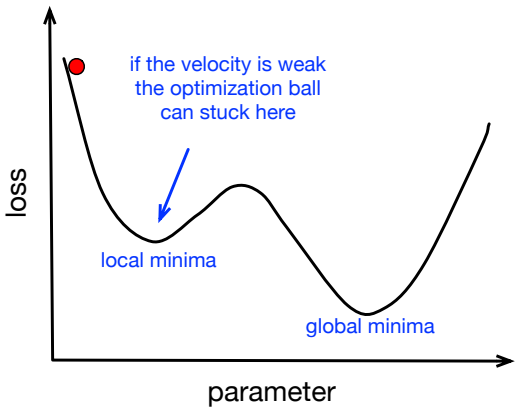


Figure 10-14: A ball with low velocity stuck in the local minima, but having high enough velocity can helps it jump out of local minima and move toward global minima.

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 other words, *momentum reduces the oscillation of the optimizer algorithm, and thus it can reach the minima faster*.

The SGD with momentum algorithm is as follows:

$v_t = 0$ `# velocity`

$\eta = 0.9$ `# momentum coefficient`

$\alpha = 0.01$ `# learning rate`

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

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

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

$v_t = v_{t+1}$

$t = t + 1$

}

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×*momentum + learning_rate*× *gradient*

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

next_weight = current_weight + next_velocity ×*momentum*

You can see that current velocity, momentum, and learning rate all are hyperparameters and they have been given by the user of the algorithm. Nevertheless, usually, they are already assigned in the algorithm, and the only hassle for the user is choosing the optimization method and calling it. For example, at the time of writing this book, in Keras,[6] you can 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 you need to give some parameter value, you can 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.

Nesterov Momentum

Using momentum significantly improves the convergence speed of SGD. However, by using classical momentum (the one we have explained), 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 toward the correct

---

[6] https://keras.io

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

[8] https://pytorch.org

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 one, the gradient is started after the momentum, and thus it moves more toward the correct direction.

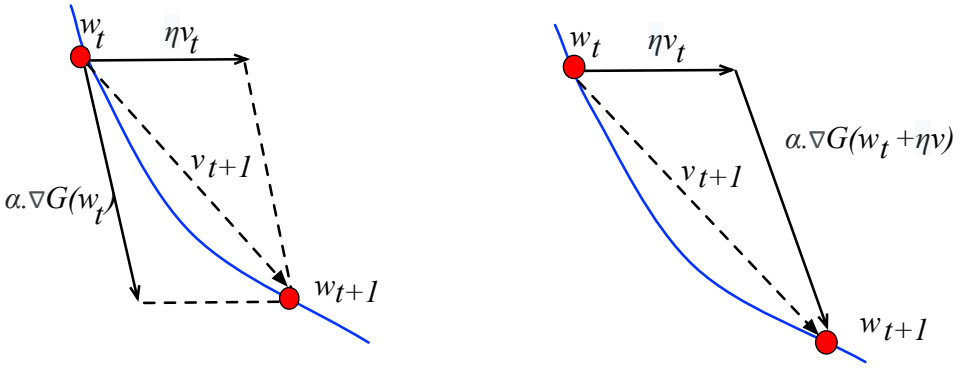
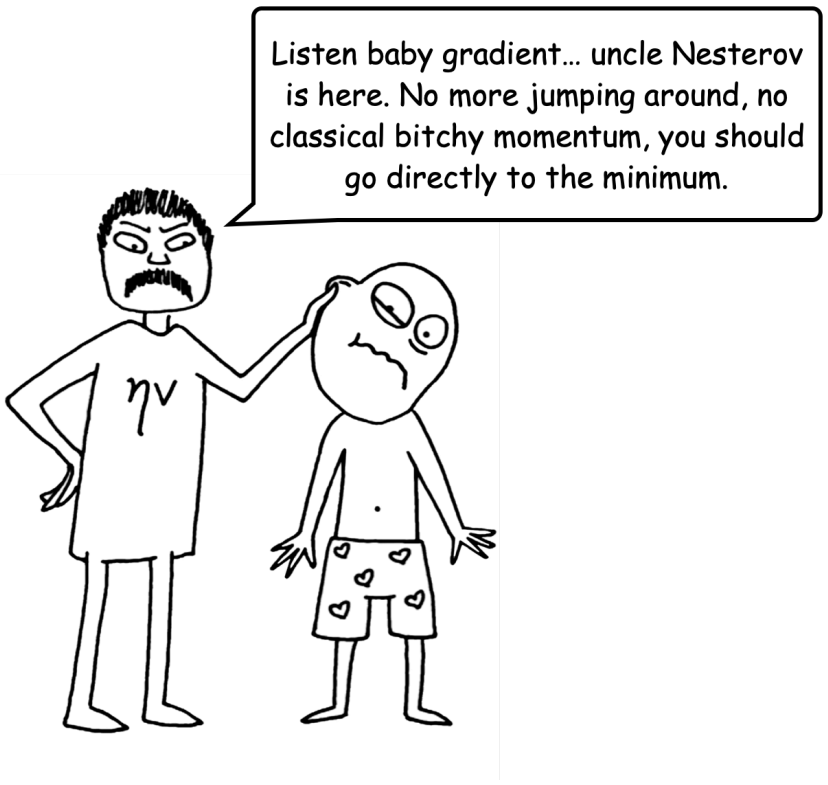


Figure 10-15: (left) classical momentum for calculating the next parameter, (right) Nesterov momentum is used to calculate the next parameter on cost function. The blue line presents a small part of the cost function and each new *w* is moving toward minima.



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:

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

Nesterov 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 by using Nesterov momentum that if the momentum goes in the wrong direction, the gradient has the chance to correct it, 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 assume that the learning rate is a constant. A constant learning rate, which is a hyperparameter given by the user, could not be useful to be used for all scenarios and all types of networks. Experiments show that a learning rate in a multi-dimensional (or features) dataset 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, having two different learning rates, one for biases and one for weights help the cost function to reach minima faster, than having a constant learning rate. This means that 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 weight.

In particular, it adapts the learning rate of all model parameters by scaling them inversely proportional to the square root of the sum of all of their historical/previous squared values. We are very sure you did not understand the previous sentence, so we explained it with the equation, which is easier to understand. Traditional 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 or η) is constant in all epochs of traditional SGD, but Adagrad algorithm tries to use 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 $w_{t+1} = w_t - \alpha'_t \,.\, \nabla\, G(w_t)$. 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 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 *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, *the use G_t enables Adagrad to reduce the impact of parameters that have large gradients (features that are frequent in the dataset) and increases the impact of parameters that*

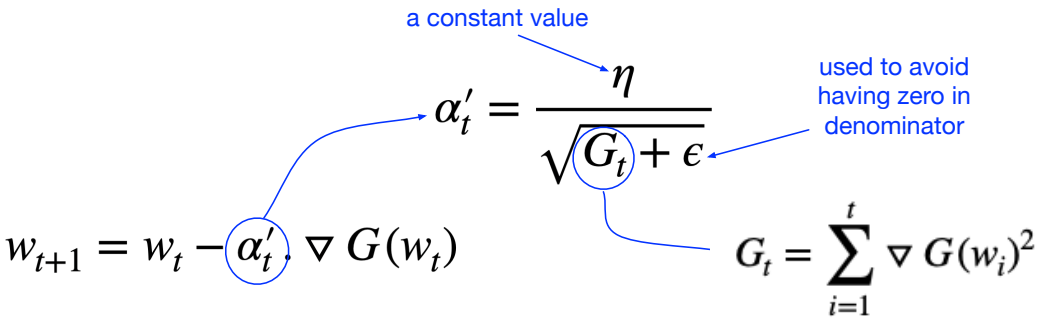


Figure 10-16: Adagrad parameter calculation equations.

have 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 having a large denominator makes the number small and vice versa.

To get an overview of what has been explained, take a look at Figure 10-16, which shows the described equation in one 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 $G_t = 0.16$.

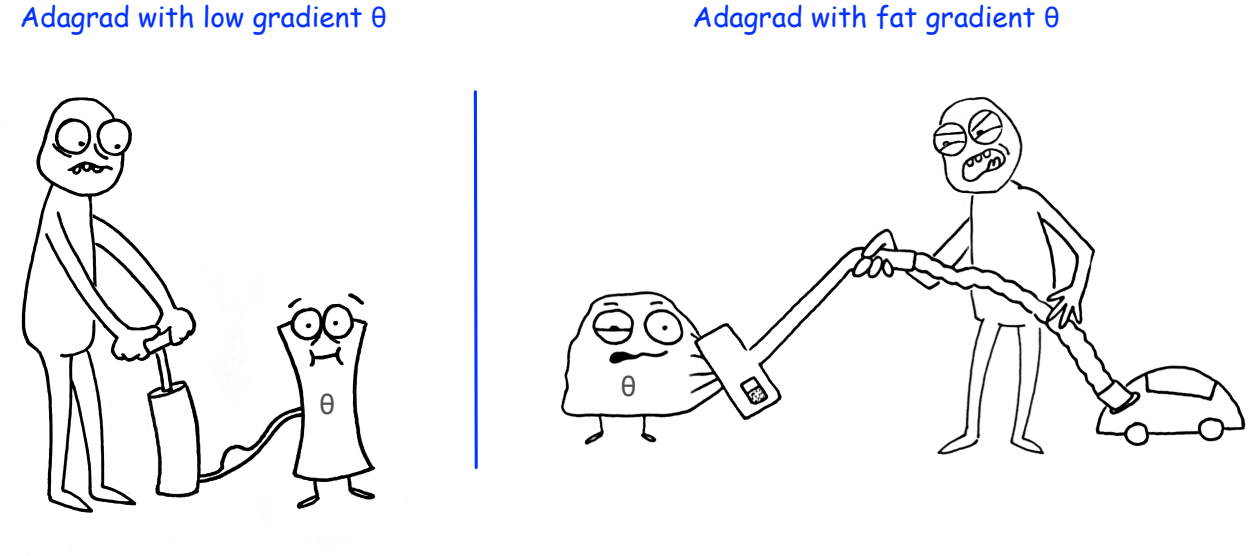
$$\alpha'_t(f_1) = \frac{0.01}{\sqrt{3.46 + 10^{-8}}} = \frac{0.01}{1.86} = 0.005$$

and

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

Now by comparing 0.005 and 0.025 we can realize that 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 have lots of zeros (sparse features) to have larger learning rate.*

There is an optimized version of Adagrad, which is 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 we don't have the energy for explaining Adadelta

RMSprop

RMSprop (Root Mean Square propagation) is an optimization algorithm that has been introduced by Hinton [Hinton '12], who is one of the pioneers of deep learning in his online course[9].

Similar to Adagrad, RMSprop focus to mitigate 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), this means the jump was too large, and thus, the minima have been passed and not reached. In the next step, it decreases the jump size, by multiplying it to a number less than 1, e.g. $\eta^- = 0.5$. If the sign has not been changed in two consecutive gradients, this means the jump was correct, and we are moving in the correct direction. Therefore, the step size could be increased, by multiplying it by a number larger than 1, e.g., $\eta^+ = 1.2$. For example,

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

if $\nabla G(w_t) = -1$ and $\nabla G(w_{t+1}) = -2$, this 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, 0.9. Here, Rprop increases the weight six times and decreases it only once from (-0.1 to 0.7). This means, 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:

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

In this equation $E(G_t)$ is the exponentially weighted average gradient, and it is calculated as follows: $E(G_t) = \beta E(G_{t-1}) + (1 - \beta) \nabla G(w_t)$.

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, we could say that *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 together. 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. 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 hyperparameters, were recommended by the authors of the Adam algorithm. 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`

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

}

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 as details of Adam, and it seems that in the golden era of Gradient optimizer algorithms, authors of these algorithms, with small changes, make a big improvement in algorithm accuracy.

Optimizer Algorithms Summary

Congratulation, now you are an optimizer expert and know these mathematical definitions. Since this section is super exciting, funny, has no theoretical discussion, and is full of real-world examples, let's do a summary before you completely fall sleep.

SGD will change the model parameter per sample datapoint, which is too frequently, and this could make lots of oscillation on the optimizer. To handle this problem we can use mini-BGD that 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 minina. 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, 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.

We explained that Adagrad tunes the learning rate per model parameter and as a result, it is a good choice for sparse datasets. RMSProp also tunes the learning rate per model parameter and as a result, it is 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 learning rate) for each model parameter as well[10].

There are only more than 150 optimizers remaining that you can learn on your own [Schmidt '20]. All jokes aside, it is the madness of making better optimizers, and many students are struggling to make impactful optimizers. Unless you like to join the optimizer algorithm maker club, there is not much need to learn the rest. Besides, we doubt if there is much that could be done in making a new optimizer, so we recommend investing your energy in another direction.

NOTE:

* There are lots of optimization algorithms that perform small updates on existing ones. 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.

* There is no best optimizer that can perform betters 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 about the best optimizer based on your dataset.

* If you have a mathematic background or algorithmic background and love this topic invest in making a good optimizer, which is not based on Gradient Descent, think about it twice. To date, there are many optimization algorithms, especially with Genetic algorithms proposed and none were as good as Gradient Descent ones. Gradient Descent algorithms are performing far better than any other algorithm, but there might be an opportunity to explore and identify a new one. Nevertheless, at the time of writing this part the war of optimizers, genetic algorithms lost badly to Gradient Descent ones.

---

[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 back 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 make the explanation clear here, do not worry if you didn't understand it the first time. Try reading this section more than once, it is nothing more than applying a chain rule on the neural network to change weights for reducing 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 *w1* and *w2* values. Now, the question is, which one of these input nodes contributes to the error, *input A* or *input B* or both?

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 $\frac{0.6}{0.3+0.6} = 0.66$.

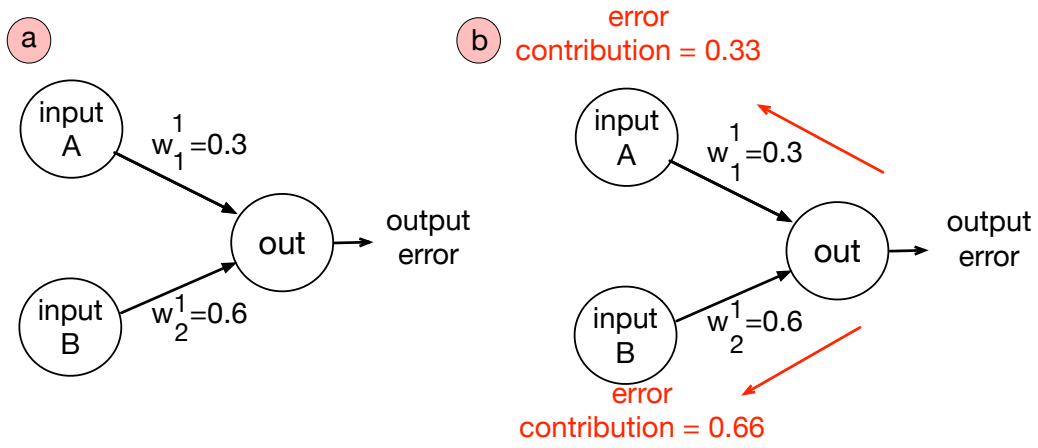


Figure 10-17: (a) Very simple network with two input and one output. (b) based on the weight of each node in the previous layer the contribution of each node into the output has been identified.

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, we have used weights to forward the signal to the output layer, this process is called 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 it is 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, the only thing that will be changed are amount of weights used in the hidden layers. In this example, for 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, we can extend the described approach to hidden layers as well.

The Backpropropagation 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 (like 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*.

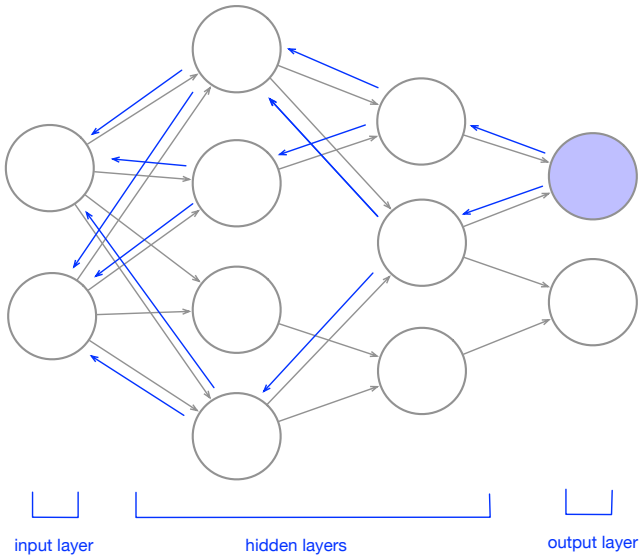


Figure 10-18: Propagation of error from the blue output neuron to the input neurons is shown in blue color.

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 we reach the input node.

Now, we understand the idea of Backpropagation, which first, 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, we can use matrix multiplication [12]. Therefore, the errors of layer n could be written as a matrix (e_n), which is the multiplication of 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, output layer errors ($e\ out_1$, $e\ out_2$) are known at the end of each epoch, and the errors of the hidden layer (e_h) could 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 will be calculated as:

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

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 how many weights and neurons are contributing 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 deep in mathematical explanation, but first, we should learn or review some conventions and concepts.

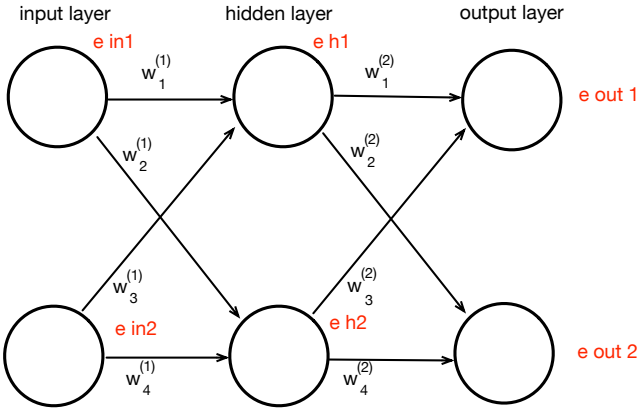


Figure 10-19: A simple neural network with one hidden layer.

The output of the neuron in the first layer can be written as $z = wx + b$, 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 implemented also with inner loops, but this is very resource intensive. there are other approaches that make it more efficient to implement it, https://en.wikipedia.org/wiki/Matrix_multiplication_algorithm

refer to the very last layer of the network as L, and for the very 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 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 last layer (output layer) L: $E_j^{(L)} = a_j^{(L)} - y_j$. 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, we learn enough conventions and concepts, let's get back to what we need to understand for Backpropagation.

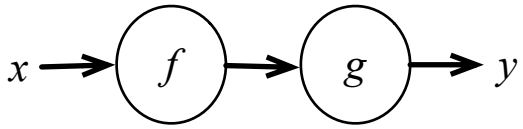


Figure 10-20: a very simple computation graph for a nested function y = g(f(x))

We should understand *how much loss score (or error) changes as weight and biases change?* Or *how sensitive is the loss score to changes in w and b?*

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 mathematic 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:

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

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

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

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

These two equations have been written for one single layer, by using a Hadamard product (check Chapter 7 to recall it), 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 here 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 the # sign is used for commenting and written in blue.

```
#------Step 1 (initialing parameters)-----
# This step initialize learning rate, weights and biases, and also the
threshold for stopping criteria will be specifies.
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) = σ(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), ... 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 error.
        E^(k) = (w^(k+1))^T × E^(k+1) ⊙ σ'(z^(k)) # ⊙ presents a Hadamard product of two
        matrices. (w^(k+1))^T is the transpose of weights for the next layer (to
        prepare the for matrix operation, they are transposed). σ'(z^(k)) is a
        vector of derivatives of activation function results at layer k.
        ⊙ σ'(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 derivate 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.

}

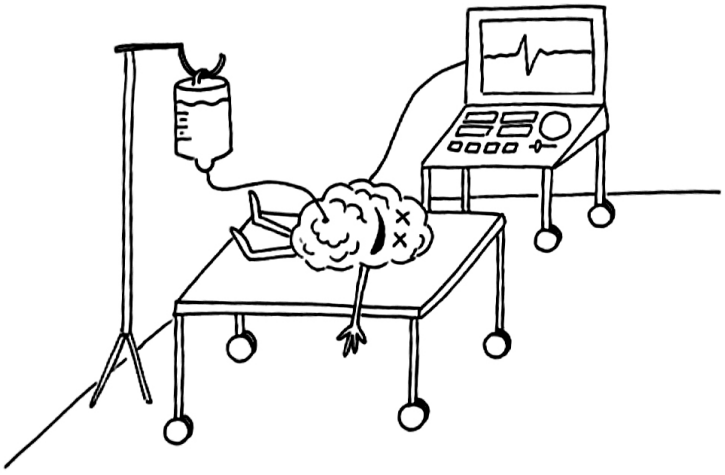
} # closes *'for i = 1 to m'* loop

We can see that the *partial gradients* with respect to weights and biases are computed by using the chain rule. The *Loss* has been calculated by using a cost function. For example, a cost function could be 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 there is also a regularization involved as well.

Your brain after learning Backpropagation Optimizer, and Activation functions



Forward and Backward Pass Example

Some might better understand the forward and backward pass on the network with mathematics. If you don't like math, 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, we need to use the chain rule of differentiation. We will start 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, we apply 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, we use 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, we compute 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 into 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 and it goes to the output and performs a prediction, i.e., a forward pass. Second, the loss function compares the prediction result with the ground truth dataset 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 (partial gradients with respect to weights and biases) are values that are used by the network to adjust its weights and biases to reduce error. As the network moves from

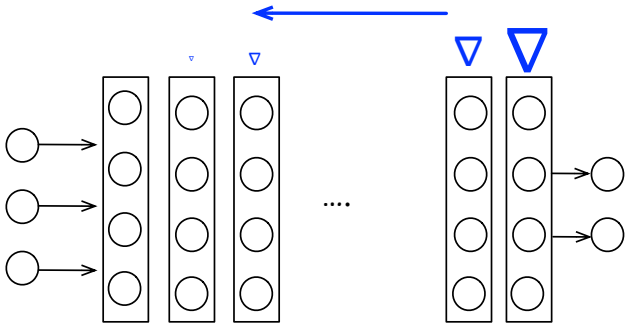


Figure 10-21: The gradient (blue triangle) is getting smaller and smaller in the back propagation that at some points get useless and weights do not change at all.

output toward the input layer, the gradient gets smaller by the chain rule, and thus the weight adjustment is getting smaller and 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 big gradient reveals a big adjustment, a small gradient reveals a small adjustment on weight.

When a network is deep (has many hidden layers), the gradient is getting 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 going to be 0.5 or less than one, as we go deep in the network it gets smaller and smaller because it gets multiplied to some smaller number, for example, it will be $0.5 \times 0.4 \times 0.1 = 0.002$. Then the weight in the input layer will be $w_{new}^{(l)} = w_{old}^{(l)} - \alpha \times 0.002$ and we know a learning rate is also a small number such

as 0.01, so we will have $w_{new}^{(l)} = w_{old}^{(l)} - 0.00002$, which means the new weight has very insignificant differences with the old weight. The same thing could happen with gradients larger than one, and thus it makes a very big gradient that does not let SGD to get close to minima, which is **exploding gradient**.

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

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 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 at the beginning completely random. These approaches are known as Xavier [Glorot '10] and He [He '15].

Xavier initialization uses random weights that have a normal distribution, and not completely random weights.

He initialization (a.k.a Kaiming initialization) proposes a weight initialization for non-linear activation functions such as ReLU and Leaky ReLU while taking into account the non-linearity of non-linear activation functions. Weights are initialized to have a normal distribution, with zero mean and standard deviation of $\sqrt{2/n^2}$ (n is the number of weights to configure), 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 using a transformation and normalizing the range of gradient. For example, we cut off all gradients that are larger than a specific threshold, e.g. 1 or smaller than a specific threshold, e.g. -1. Then do not use gradients outside this range for the backpropagation algorithm to adjust weights.

Usually, Gradient Clipping is being used in RNN networks, which we 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. The same story applies to neural networks too. 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 neural network to predict the happiness of our users based on their age and annual income. The age is a number between 0 to 120, but income is widely varied for example in the U.S. 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 context of the neural network, this process is called input **Batch Normalization** or **Batch Norm**.

Batch normalization is introduced in 2015 by Loffe and Szegedy [Loffe '15]. It brings the scale of all data between 0 and 1. Batch norm is applied on a layer basis. Not having the normalization could increase the large weights to have an exploding gradient or small weights to have a vanishing gradient. Besides, by applying batch normalization we can significantly increase the training speed.

The process of batch normalization is implemented as follows:

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

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

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

(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 and it is 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 getting optimized 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 not applied on the input layer, it is only applied to the *output of hidden layers*.

Now, that we understand batch normalization, a question might arise, why do we need batch normalization and not just using traditional normalization? The problem is traditional normalization may not go far in a deep neural network and still, the network 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 tuned back by the

backpropagation algorithm, the original distribution of input data will change. This problem is referred to as covariate shift. Batch Norm can mitigate this issue.

In summary, we use Batch Norm to speed up training (via reducing the computational overhead), decrease the importance of initial weights, and slightly regularized the model.

Other Normalization Techniques

Now that we understand Batch Normalization, we need to learn a bit more specific types of Normalizations, which include *Instance normalization* [Ulyanov '17], *Layer Normalization* [Ba '16], and *Group Normalization* [Wu '18]. Assume we have input data that we feed into a neural network. The data is in tensor format, and it has height, width, and depth (channel). We will learn later 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, 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.

Layer Normalization tries to address Batch Norm's dependency on the batch sizes, which is not possible 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, instead of batches*.

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 within each group the mean and variance for normalization, which makes it independent from batch sizes.

Figure 10-22, is designed by Wu et al. [Wu '18] to summarize these normalization approaches.

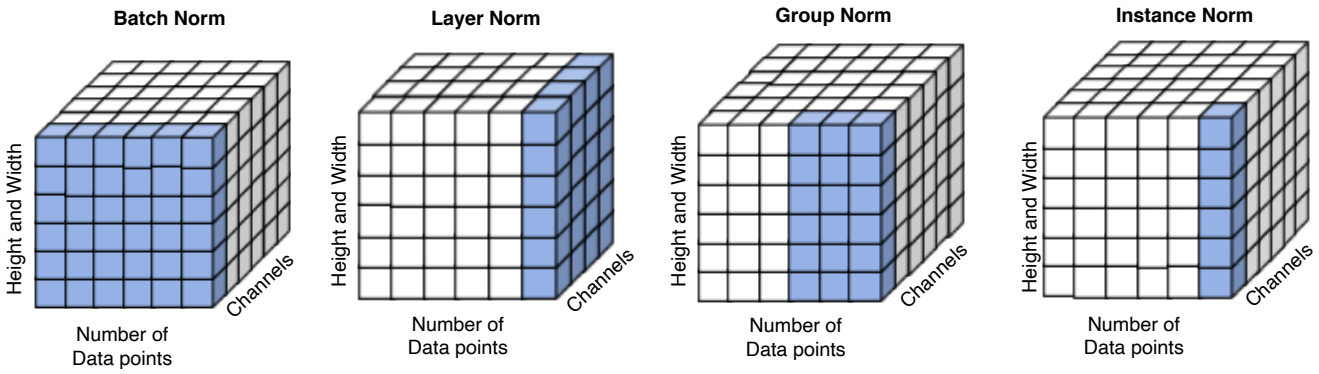


Figure 10-22: Different neural network normalization methods,

Dropout

One of the most popular neural network regularization methods that have been introduced by Hinton et al. and popularized by Srivastava [Srivastava '14] is the use of Dropout.

The idea is simple but very effective. During the training phase in each epoch, a random number of neurons output will be converted to zero and does not provide any information to the next layer. Usually, this process is performed by adding a layer that is called the **dropout** layer. 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 hidden neurons are turned off (their weights will be equal to zero). This had a tremendous impact on reducing the overfitting. Usually, the dropout rate is larger than 0, and less than 50% of neurons in each layer.

In other words, using drop out ensures that the network will be not dependent on a neuron and all neurons are participated in constructing a model, which avoids overfitting.

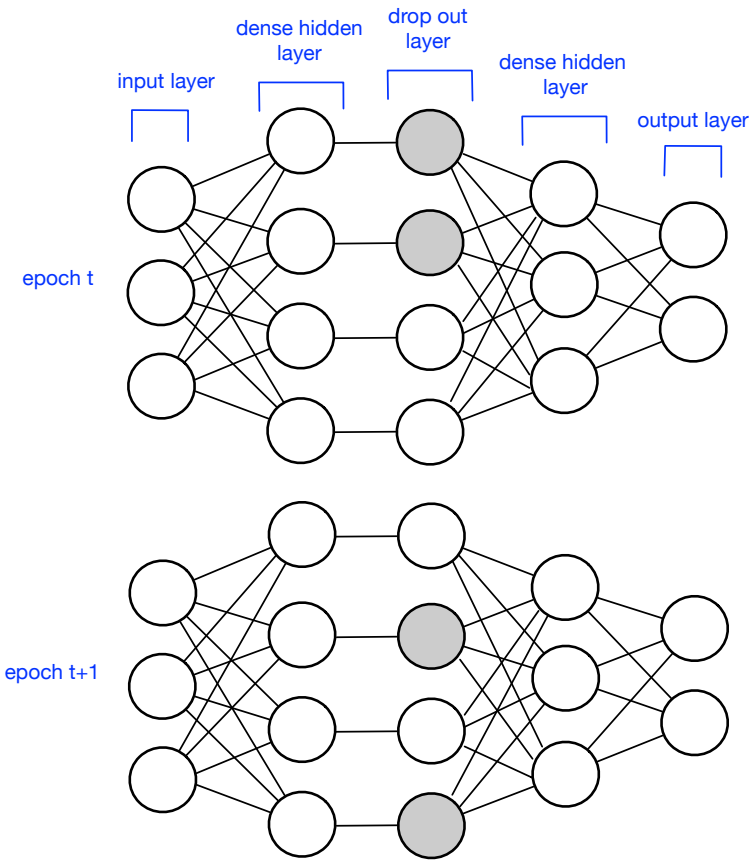


Figure 10-23: A mock example of having drop out layer after the first hidden layer. Nodes which are converted to zero are presented in grey color and this layer in each epoch turns off (make their activation function output zero) 50% of nodes.

Dropout is a very effective approach to prevent overfitting. Gal and Ghahremani [Gal '16] introduce a very simple optimization that can improve the DropOut quality even more. They call it **Monte Carlo (MC) Dropout**. It simply applies the dropout at the test phase (not the training

phase). Then, at every prediction (test) the dropout neurons change, and also the prediction result. In the end, the result (loss score) of predictions will be averaged and reported. Despite its superior efficiency, it is recommended to not use MC Dropout for applications that have lots of variance in the prediction results can not be tolerated such as medical applications.

Early Stopping

We have explained that training a neural network is an expensive process and we use regularization methods to reduce its costs. While training a neural network we measure the loss and as it reaches an acceptable stage we could stop the training. One popular approach that is used for neural network regularization is called “early stopping”, which we have described back in Chapter 8 in more detail. There is nothing much to add here, except that the same approach can be implemented on the

NOTE:

* We give up reporting the computational complexity of neural networks and there are not available as well. These are accurate but highly complex algorithms, which are not optimal for settings that have limited capacities such as wearable or other battery powered devices. Usually, in those settings we benefit from loading and using a trained model, such as Federated learning [Konečný ’16] or using the trained model on a small device, i.e. TinyML [Warden ‘19].

* There is no optimal way to design the number of layers and number of neurons, and since there is no methodological approach to doing it, the selection of neurons and layers is referred as “dark art”. A simple common approach is to use a large model, which is larger than our need, and use drop out to prevent overfitting. This approach is called “stretch pant”, in which we do not search for a pant that fits, we get a large pant that stretches and fit our size. Nevertheless, there are some common architectures that experiments show their superiority and we describe some of them in Chapter 13.

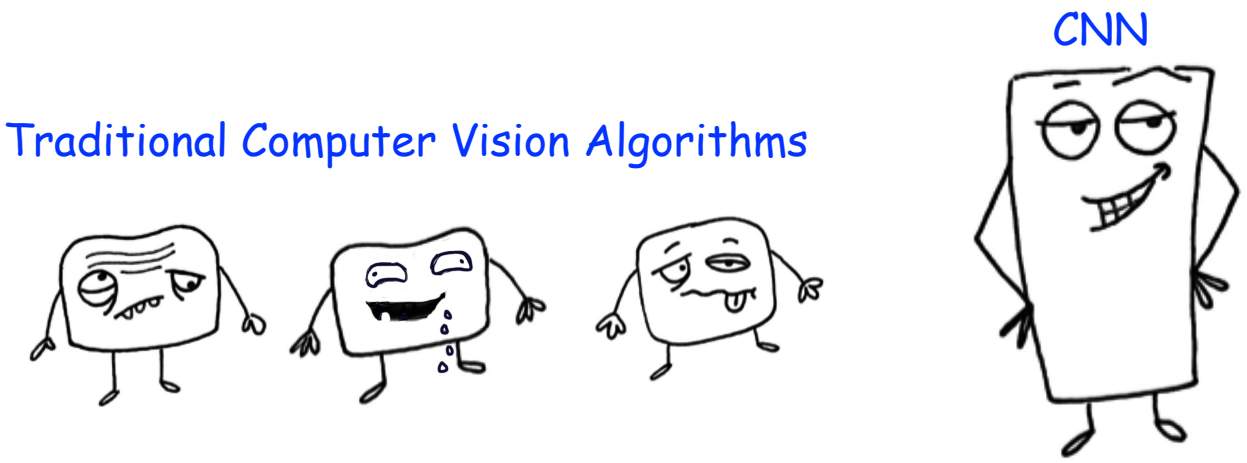
* A neural network has lots of hyperparameters to tune including, learning rate, number of hidden layers, number of neurons, cost function, types of activation function, optimization algorithm, batch size (recommends to be lower than 32), etc. At the time of writing this book, reviewers do not reject a neural network paper, based on not analyzing the hyperparameter, but this might be an expectations in the future.

Convolutional Neural Network (CNN)

We learned the general architecture of the neural network, ANN, and now we can describe a deep learning architecture, Convolutional Neural Network (ConvNet or CNN).

Back in 1950, two scientists Hubel and Wiesel [Huhel '59] identified that monkeys' and cats' brains have two types of cells in their brains (simple cells and complex cells) used for visual recognition. Later Fukushima [Fukushima '80], inspired by the findings of Hubel and Wiesel introduced the CNN architecture. Afterward, LeCun et al. [LeCun '89] combines the CNN with the Backpropagation algorithm and experimented with it on handwritten numbers[13], their results show a tremendous improvement in the accuracy of handwritten numbers recognition.

CNN architecture preserves the spatial structure of the input data, and preserving the spatial structure of data makes CNN very popular (probably the most popular) for computer vision applications such as image recognition, medical image analysis, and even CNN are used in few other applications such as natural language processing or audio analysis.



CNN models are spatial invariant (can recognize an image from a different angle) and very 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 book, most state-of-the-art image analysis algorithms are using CNN, and they even overtake human experts in some computer vision applications as medical image

---

[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) that 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.

analysis [Javaheri '21]. CNN operates by assigning multiple image filters to an input image and these filters enable the algorithm to accurately classify the image.

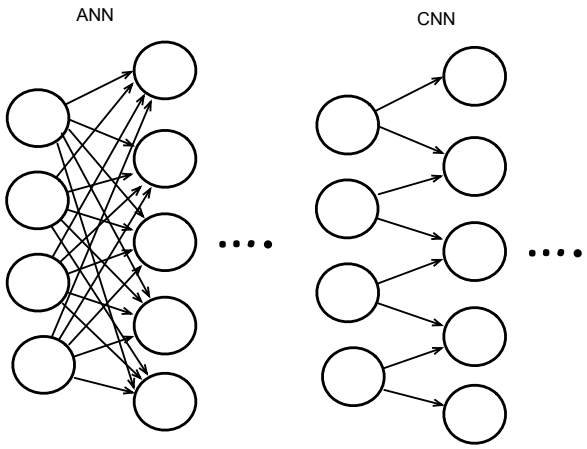


Figure 10-24: A traditional input layer and second layer of ANN on the left and the same approach for CNN on the right, you can see the number of connections in CNN is smaller than ANN, which is fully connected.

CNN has two salient advantages over ANN. First, ANN uses densely connected layers, in CNN each neuron is connected to a limited number of neurons in the neighbor layer. Figure 10-24 shows a comparison between CNN and ANN, and you can see all input neurons of the ANN are connected to all neurons in the next layer. Having all neurons connected to all other neurons of the neighbor layer cause 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 number of connections is defined by the window and stride which we will explain 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 classifies it, but CNN can classify them, even if the number is located on the side of the image, because of its local pattern learning capability.

The second salient advantage is, that CNN is learning the *hierarchical features of the image*. For example, it separates a chicken image into a peak, eyes, wings, etc. This allows the CNN to learn complex structures and be able to classify similar images, which is not possible with other algorithms. In simple words, if two images are taken from a different angle and in different light settings, still a CNN algorithm can classify them.

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 for colored images we work with a 5D tensor[14]. If we convert the data into black and white and feed it into a CNN, then we deal with a 3D tensor, one for height, one for width, and one for the color intensity. For 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.

---

[14] In practical cases usually we use one dimension for color channel and it is 3D, but we wrote 5D to enable you understand the concept behinds the dimensions.

Convolution and Cross correlation

Convolution is the process of combining two functions and, as a result getting a new function, which has 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*. In simple words, the integral specifies the amount of overlap of one function as it is shifted over another function.

It can be formalized as 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. If two functions are not overlapped, their convolution is going to be zero. As soon as a function has some overlap with the other function, then their convolution is none zero. On the top of Figure 10-25, we have

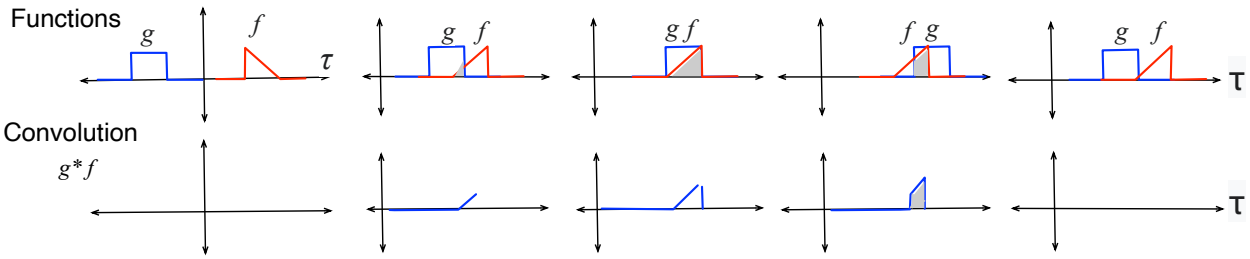


Figure 10-25: Two functions at top and a their convolution function in the bottom. The *g* function is moving along τ and as it overlaps with the *f* function (the grey area is the overlapped are) the convolution starts to increase and then again decreases as the *g* is moving away from *f*. Note that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

two functions *f* and *g* with similar shapes, 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 the *f* function is reversed from the left of the second figure.

Cross correlation is similar to convolution, but it measures how similar are two different functions. In other words, it is similar to the convolution with the difference that it does not flip one of the functions, convolution function flips one function (flipping means negative all of its variables 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:

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

Respectively their cross correlation will be written as:

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

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

How to Build a CNN network

To implement a CNN we should follow five steps, and in the following, 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.
The task of kernels or filters in the CNN is feature detection. Kernels are similar to the image filter that we are using on photo editing software such as blurring an image, sharpening an image, etc. Filters are matrices (tensors for colored images) that are multiplied by a given image and the resulting image will be the transformed image that is blurred, sharpen, etc.
Figure 10-26 shows some examples of filters that are 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 **convoluted image**, thus we can write the following:

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

There are common kernels in use. For example, a matrix with Gaussian distribution is a kernel used for blurring an image. Take a look at Figure 10-27, there 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 in the

result of applying the kernel on the original matrix. Using this kernel increase the emphasis on pixels located on the left side of the kernel.

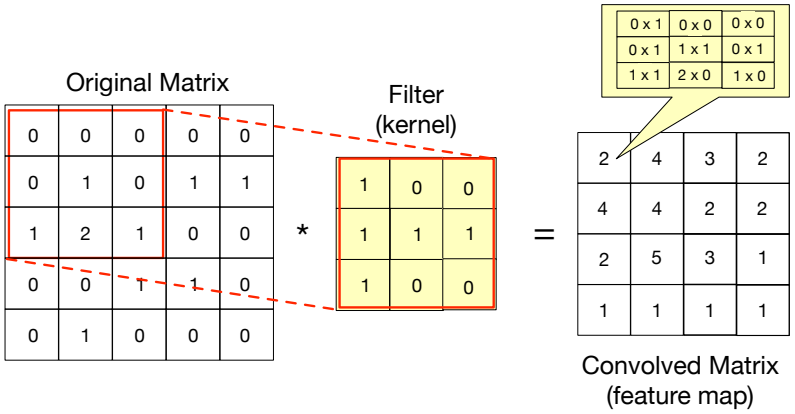


Figure 10-27: A matrix, which presents image and a filter that will be applied on the matrix. The right matrix presents the result of applying the filter on the original matrix.

Note the '*' operator here is referred to as **dot product**, which is element-wise multiplication. It is common that a filter is smaller than the original image.

The red window of Figure 10-27 is moving along the matrix and constructs pixels of the convolved matrix. The number of pixels that a window move is called **stride**. In Figure 10-28 we move the window one pixel in the X and then Y direction. Therefore, we say the stride is equal to

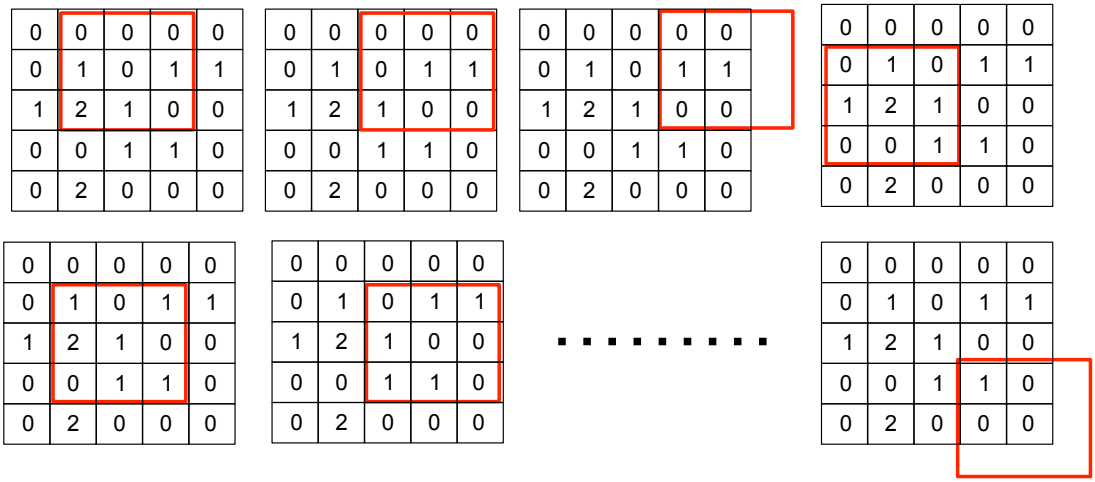


Figure 10-28: An example presented a window equal to the size of filter (3x3) is moving along the original matrix (image) and select a window of data to multiply them to the filter values. Here the stride is equal to one on both X and Y axis.

one. Usually, the stride is set to two pixels, and the larger the stride the convolved image is getting smaller.

There are many different types of kernels that are used to manipulate the input image, a few examples are presented in Figure 10-29. We can see some filters and intuitively by looking at the matrices we can realize what are they doing. For example, Sharpen filter on the right, increases the differences between the right and left center pixels and reduces the emphasis on the other pixel, thus it can be used to sharpen the image.

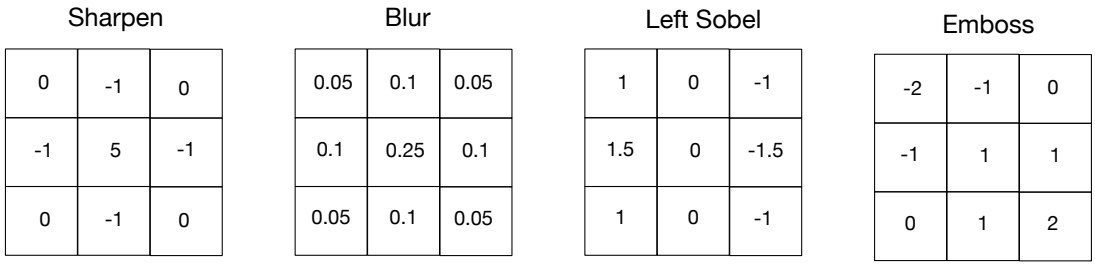


Figure 10-29: Some sample image filters. Sharpen and Blur are clear, Left Sobel is used to showing differences between each pixel and its adjacent pixel on the left. Emboss creates an illusion of depth for the viewer of the image.

During convolution, we lose some pixels, depending on the filter size. You can see in Figure 10-27 that the convolved image is smaller than the original image.

As is shown in Figure 10-28, the window is fallen outside the image matrix, and thus there is no pixel there, which causes losing the border of the image on 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 to performing padding, including **reflection padding**, **replication padding,** 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.

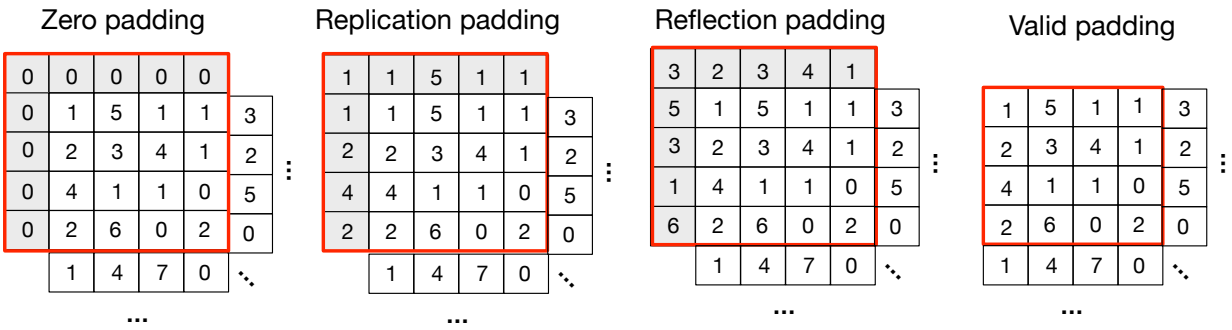


Figure 10-30: Three different approaches for padding on edge pixels. The white area in this figure presents the image pixels and grey ares is the padding area.

Reflection padding is using 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 approaches.

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 10-28).

(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), usually, the feature map will 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" as well [Goodfellow '16].

Keep in mind that at the end output layer should be something related to the task we except 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, if it is for two classes then it should be Sigmoid (check Chapter 8 to recall Softmax and Sigmoid).

Once again let us remind you, unlike ANN where all layers are connected to each other, (fully connected network) CNN neurons are only connected to a subset of neurons in the next layer, and these are filters. Figure 10-24, visualized 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 which is done automatically by the CNN is spatial invariant. Spatial invariance is the feature that makes CNN algorithms a very accurate and flexible algorithm for image classification. For example, a traditional image feature extraction, cannot recognize that the Figure 10-31 are chickens, because the shape, camera angle, etc. in all chickens are different. However, a CNN that has trained on many comic chickens can recognize the given chicken image, despite their different visual shapes.

Each convolutional layer applies many different filters on the input image for feature extraction. The number of filters is a hyperparameter, and it 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 windows and strides which downsample the original image into smaller images. The result of the convolutional layer, i.e., feature maps (after they have been

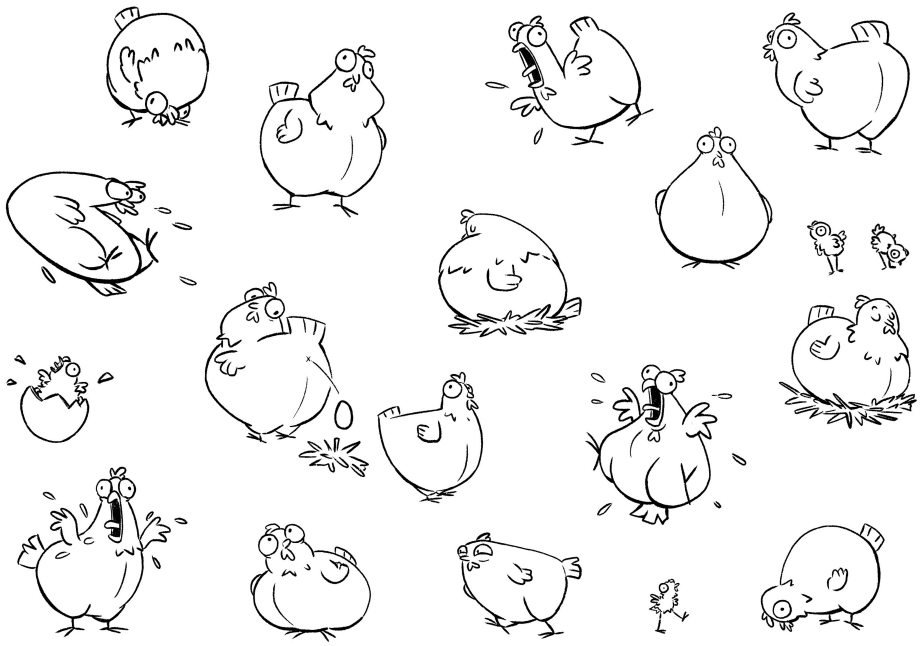


Figure 10-31: All of them are chickens but a spatial variant algorithm cannot detect them. CNN is spatial invariant and it can extract features such as body shape, peak, eyes, etc. and thus recognize them. However, it can recognize them, only if we train the algorithm with enough sample images.

transformed to non-linear feature maps) are sent to a pooling layer via a *tensor operation* it performs the downsampling. We could say it 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 10-33 shows some downsampling examples. Downsampling here refers to lowering the resolution while still maintaining the features. Perhaps you read this book from the beginning and you recall SIFT algorithm back in Chapter 6, that blurs and resizes the image for feature extraction. That process can also be called downsampling as well.

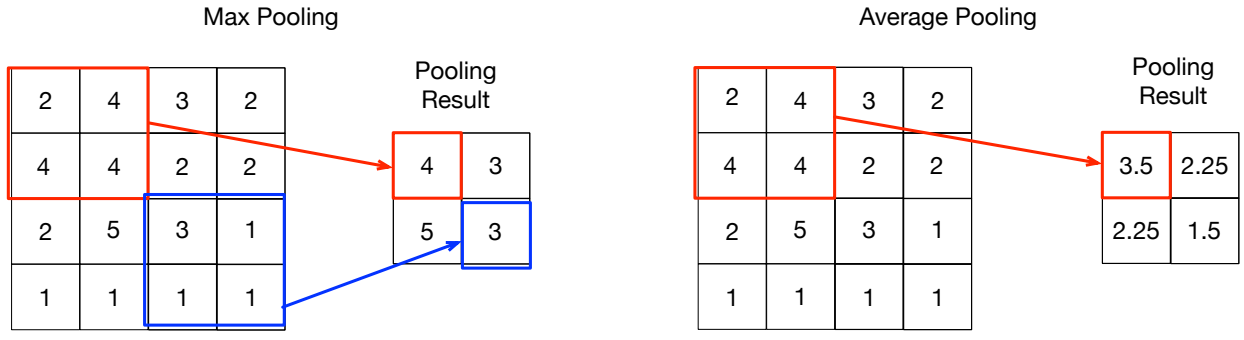


Figure 10-33: Max pooling and average pooling examples, which are applied on the convoluted image (feature map) and creates the pooled feature map. In these two examples we use stride of two pixels in each direction.

Pooling, similar to convolution uses a window of 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 them as a single pixel for the next layer. Figure 10-33 present max pooling and average pooling examples.

There are other pooling methods as well, such as calculating *L2 norm* for all pixels inside a window, but they are not as popular as max pooling.

Pooling has other advantages as well, it removes irrelevant information and unnecessary features. This reduces the chance of having 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 usually, there will be a normal flat feedforward layer. Flattening is very simple, the result of the pooling layer is a matrix, and the flattening layer converts matrices to a vector, as it is shown in Figure 10-34. Since we have many pooling layers, the vector size is usually too long.

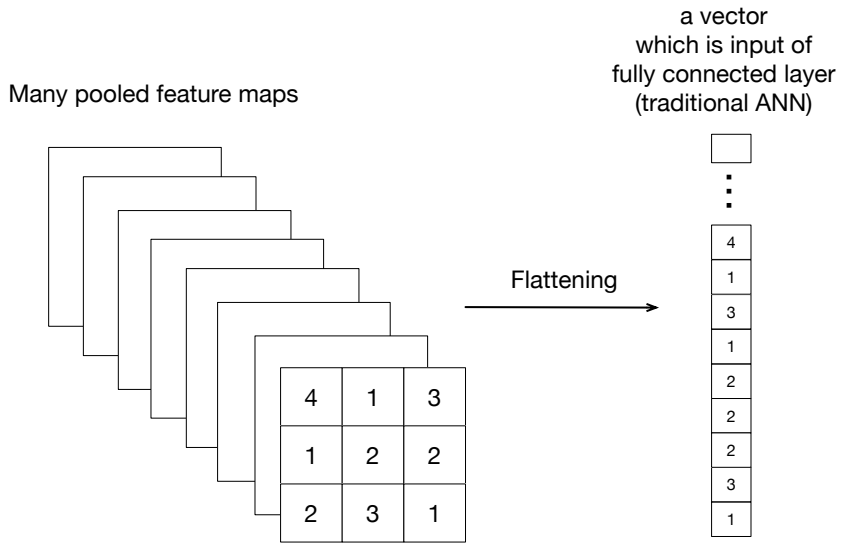


Figure 10-34: Results of max pooling functions are flattened into a large vector, which will be the input of the dense layer (traditional ANN).

(Step 5) Dense Layers

After this very long vector has been created, then this will be fed into a traditional ANN layer. The input layer of this ANN is the result of flattening, its hidden layers are called **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 in the 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 are converged (maximum iteration reached or accuracy reaches a satisfactory number).

If a feature is useless, its weights get reduced or removed entirely 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.

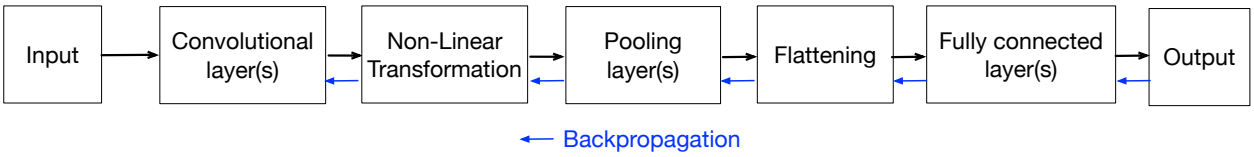


Figure 10-35: A brief summarization of CNN architecture. Here fully connected layers refer to traditional ANN.

Figure 10-35 summarize what we have explained in the CNN architecture. However, there are usually multiple convolutional layers, multiple pooling layers, and multiple fully connected layers. Even we could have several convolutional layers, then several pooling layers in a sequence.

A CNN architecture can be used for classification and regression both. If the CNN model is designed for classification it will have more than one neuron at the end, but for the classification, its output layer neurons are equal to the size of the classes that are used for labeling. If the classification is trying to distinguish between cat, dog, and t-rex. Then we will have three neurons. Nevertheless, the neurons in the output layer are not aware of the value of other neurons in the last layer. However, they should provide a probability and sum of those probabilities should be 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 to one. The softmax function is used usually 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, by measuring the accuracy and then deciding about 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 super rich AI corporations, and it is not easy to generalize all of them for real-world applications who is developed by non-super rich organizations. Therefore, if you develop a model which 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, which its 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.

3D Convolution: Until now we only use matrix and explain convolution on 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 original data and filter must be exactly equal. For example, in Figure 10-36, both Filter and original data has the same number of channel, i.e., 3.

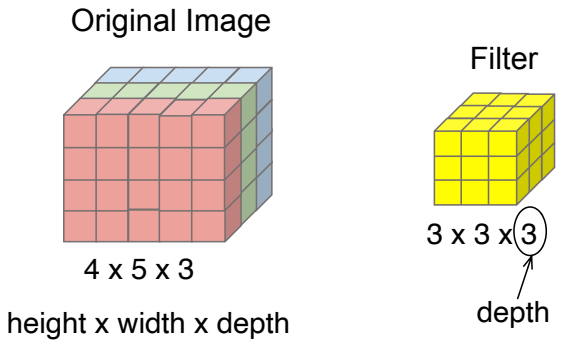


Figure 10-36: An example of a 3D Convolution with its filter. As it is shown depth or channel in both filter and original image should be the same.

The dot product of a 3D filter to a 3D filter will be a matrix. In particular, each filter multiplication results in a scalar. Then the 3D filter moves by a specific stride and another scalar is constructed. These scalars together construct a 2D matrix[15].

3D convolutions are used for common 2D images and also volumetric images (e.g. MRI, CT images), which also include voxel (a pixel with one additional dimension that is depth) or video files (the third dimension is time).

Dilated (Atrous) Convolution: Dilated 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 10-37 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

[15] A video that visualizes this process: https://www.youtube.com/watch?v=D0VoQDDe5zI

skips several pixels. Dilation convolutions are popular approaches for image segmentation [Chen '17] because it provides an overview of the larger scope of the image. More about image segmentation will be explained in Chapter 12.

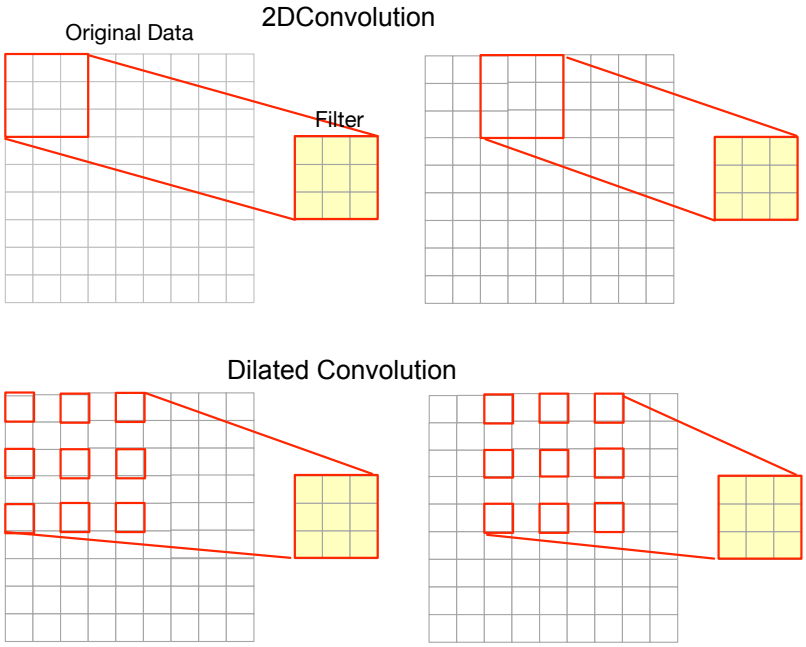


Figure 10-37: A 2D Convolution on top and Dilated convolution in the button. Both has zero padding and stride size of 2.

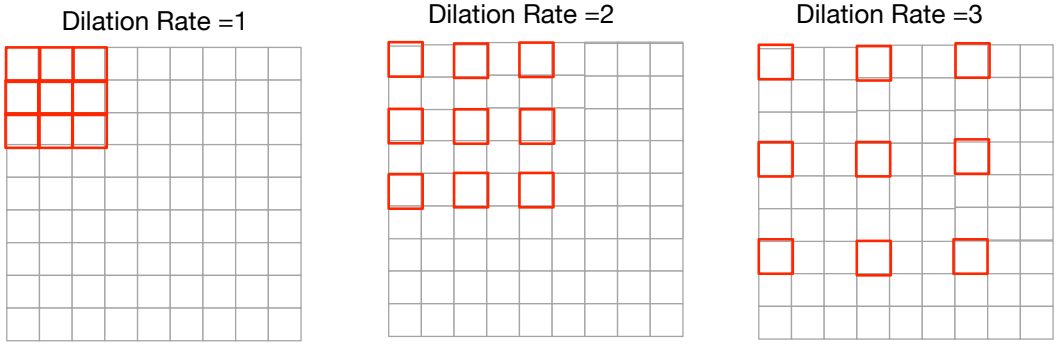


Figure 10-38: Different dilation rate specifies the size of gaps between selected pixels.

Figure 10-38 shows different dilation rates (l).

Transposed Convolution: Convolutions we explained (except dilated convolution) either keep the size of the original data or reduce its size after convolution, which is called "downsampling". The transpose convolution is usually used to increase the size of the output feature. This technique is referred to as **upsampling** process, and it has applications in image processing such as increasing the resolution of an image, reducing blurring presented in an image, and semantic

segmentation (abstracting image into a set of the labeled object), which we will describe in more detail in Chapter 12.

A transposed convolutional layer carries out regular convolution characteristics, but it reverts the spatial transformation. Take a look at the example we provide in Figure 10-39. Here $*_{TC}$ refers to transposed convolution. In this example, we have a small input image (2×2), the stride is 1 and 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.

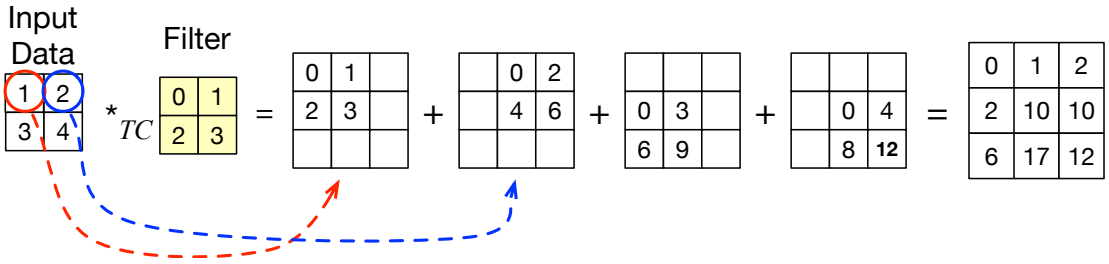


Figure 10-39: A simple example of transposed convolution, with stride size 1 and padding size 0.

The inverse of convolution is referred to as deconvolution, and it is recommended not to call transposed convolution deconvolution.

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.
* Keep in mind that we use cross entropy for classification tasks, because classifications are use Softmax function. If the task is regression means square error is recommended.
* 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 fulfill our demand, i.e., transfer learning.

* If we intend to classify images with different sizes, the pooling layer can handle this by constructing a fixed size feature map and thus making it easier for the classification.
* There are different visualization techniques available to present CNN architecture. AlexNet visualization [Krizhevsky '12] is another form to visualize CCN, and it is shown in Figure 10-40. Here, both convolutional and pooling layers are presented as cubes, and the red cubes inside convolutional layers are presenting filters.

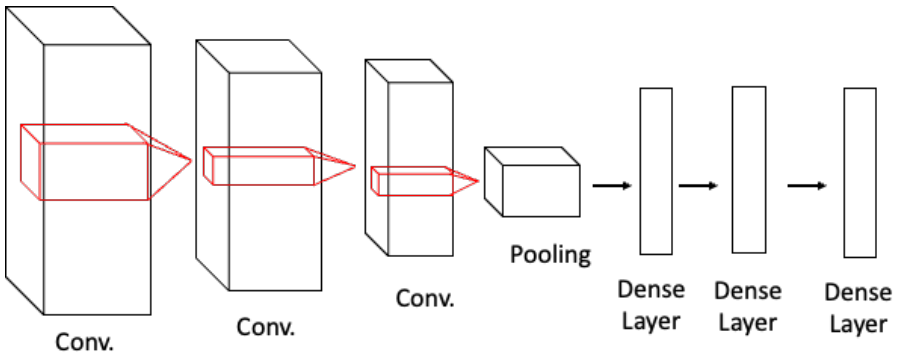


Figure 10-40: AlexNet style of presenting a CNN network architecture. Usually the sizes (height, width, channel) on the cube sides as well.

* CNN models are known to have **inductive bias.** Inductive bias refers to a set of assumptions that the algorithms made (identifying them from training data) to identify the output of the input data that it has not encountered in its training set. In simple words, generalizing beyond the training data.

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 are receiving 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).

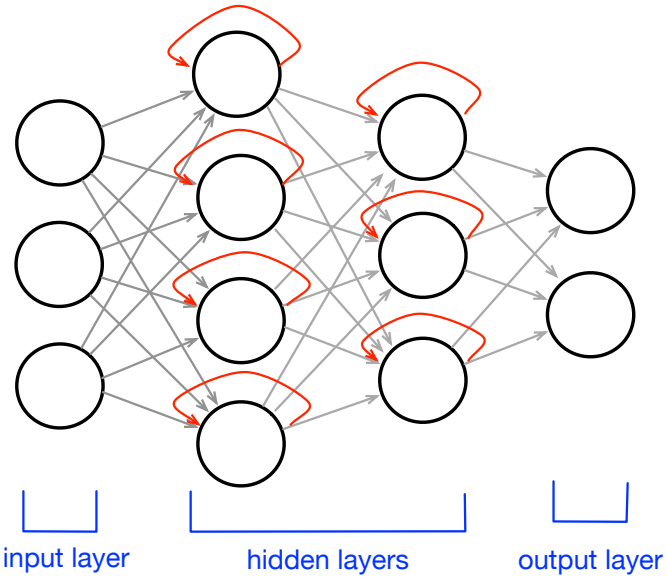


Figure 10-41: Hidden layers in RNN send their output as input in the next epoch. The red lines present the information flow in the next epoch. Here for sake of simplicity we show each neuron has a input from itself, but in fact RNN cells have input from itself, and they are different than neurons (we will explain more about this later).

Another category of deep learning algorithms is Recurrent Neural Networks (RNN). RNN is 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 that has either a timestamp or implicit notion of time or sequence, i.e., temporal data, could be modeled by RNN. Even it 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, they are not accurate enough, and we should not expect a high quality result from these algorithms.

As we can see from Figure 10-41, 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 10-41, shows a very basic representation of RNN. Hidden layers are connected to themselves, it means that in addition to the output for the next layer, they also feed the output of

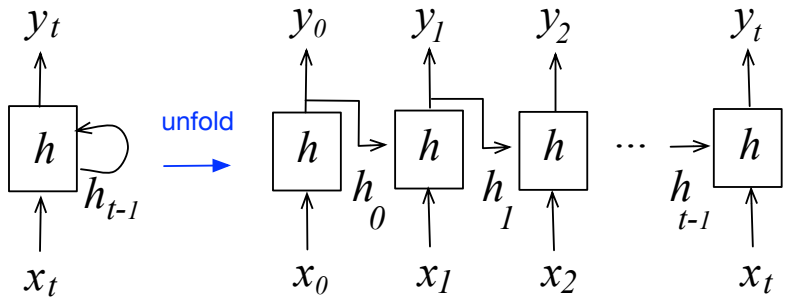


Figure 10-42: The hidden layer (*h*) in RNN receive input x and produce output y. However, the hidden layers have a temporal loop, which means that in addition to giving the output it feeds the output to itself, as input, for the next epoch as well. Each rectangle presents a hidden layer and not a single neuron

the current hidden layer to themselves as well. An unfolded hidden layer is shown in Figure 10-42.

On the right side of this figure, we can see that the hidden layer at time t receives the output of the previous epoch $(t-1)$ as well, or the layer at a time $t-1$ receives the output of time $t-2$ as well. Therefore, we could say neurons have a **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 $\bar{y}$ is the output vector of hidden layer variables and $\bar{x}$ is a vector of the input variable, then we have:

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

Neurons that are a function of inputs from the previous step are called **memory cells**.

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 the next upcoming information. For example, in the term *'Neural networ_'* you can guess that the '_' will be substituted with *'k'* because you have encountered this word before and you find it in your memory.

Based on the number of input 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 10-38 visualizes these types of presentations and notes that rectangles here are referring to a layer of neurons.

For a one-to-one example, assume a single image given into a neural network and the network finds its label.

For one-to-many example, assumes we give a single image into the RNN and it generates the caption for the image. For example, in an image, a chicken is wearing a shirt, and the shirt has a t-rex image on it. An RNN recognizes the t-rex, recognizes 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 outputs labels (chicken, t-rex, biting).

For a many-to-one example, assume we give a sentence, which 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, which includes many words in Mandarin and the algorithm translates it to a Hindi sentence, which also includes many words as

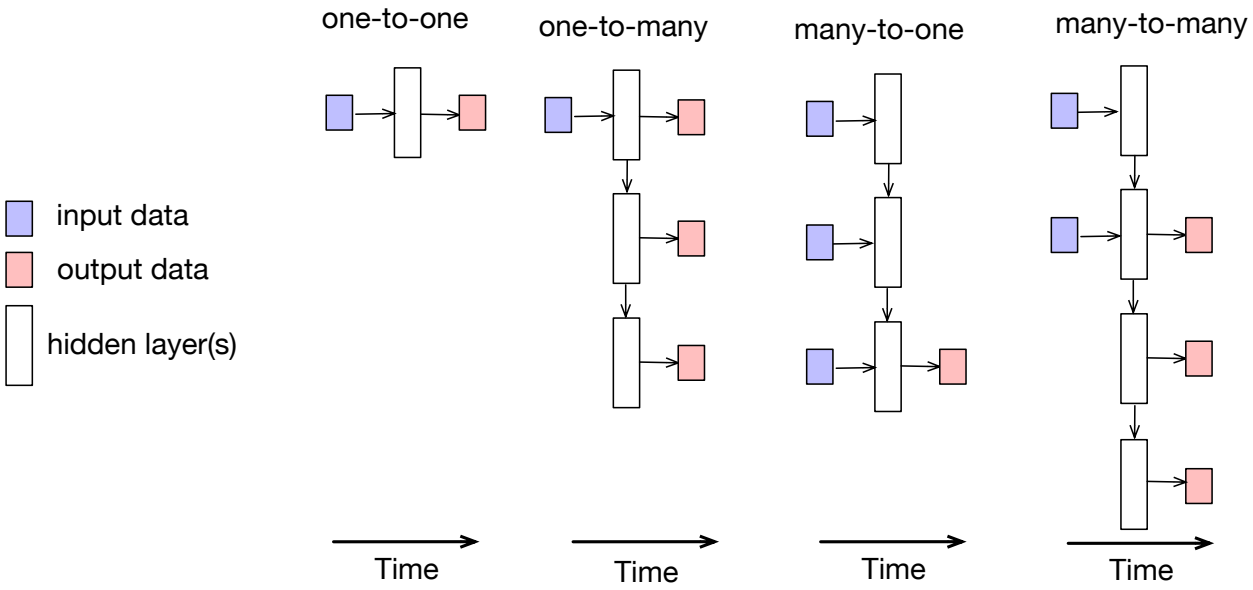


Figure 10-43: Different architectures of recurrent neural networks, you can see from left to right the time increases.

well. 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” 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, next it applies an activation function on the result of linear regression and then 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*.

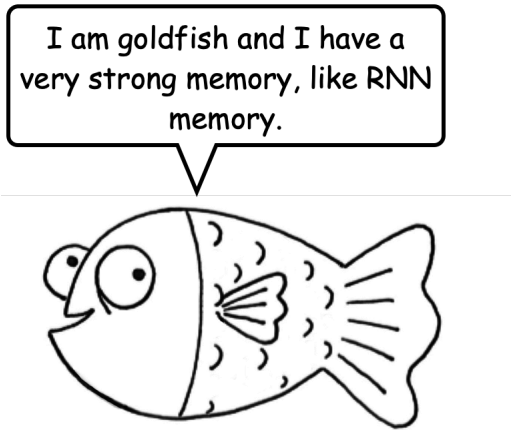
These are the basics of RNN network. It is fantastic because of having memory, but it has a goldfish memory; it only remembers the previous output, i.e., short-term memory. To mitigate this, we need an RNN that keeps tracking more previous output, i.e., long-term memory.

Long-Term 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 sequential data that are long. 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 gold-fish memory, it has only one short-term memory and can't remember more than one single previous state. 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 the previous word, it cannot recognize the context of the word fat.

A well-known approach to handling time series issues and temporal data is the LSTM algorithm [Hochreiter '97], which is a type of RNN algorithm. 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), i.e., $\odot$, sigmoid and *tanh* activation functions to decide about the output value.



Before, explaining LSTM, let's review how an RNN works. It receives x_t and h_{t-1} and uses a *tanh* to calculate h_t as output, which can be written as $h_t = tanh(W[h_{t-1}, x_t] + b)$, as it is shown on the left side of Figure 10-44.

LSTM neurons are referred as cells. As it is shown in Figure 10-44, 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) 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,

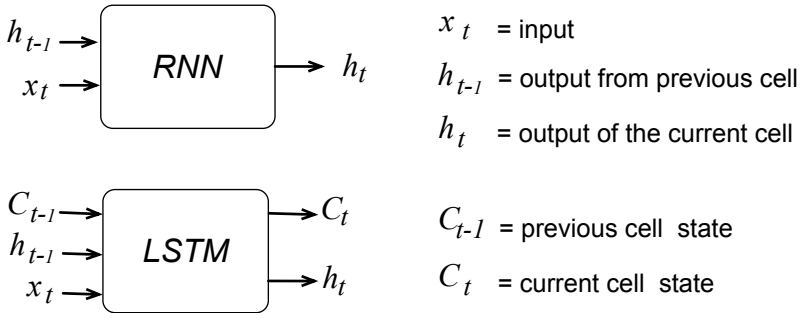


Figure 10-44: Traditional RNN neuron, which is called vanilla RNN, versus LSTM neuron. The LSTM neuron receives C_{t-1} and outputs C_t as well, which are representation of cell state that could be interpreted as long term memory.

hyperbolic tangent layers (tanh) gates, and point wise operators (Hadamard product).

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

Now, with this introduction, we can describe LSTM, step by step, as follows.

(i) The first step of 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 Sigmoid function. In particular, it receives the h_{t-1} (output of the previous time) and x_t (input), then calculate 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 (see Figure 10-45 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 10-45 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 10-45 Step 2-2). Third, after these two sub-steps, their results should be combined together 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, to forget 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 10-45 Step 2-3)

(iii) In the final step, the algorithm decides what is going to be the output vector, which is based on the filtered version of the C_t cell state. Here the filter is referred to as passing the 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**

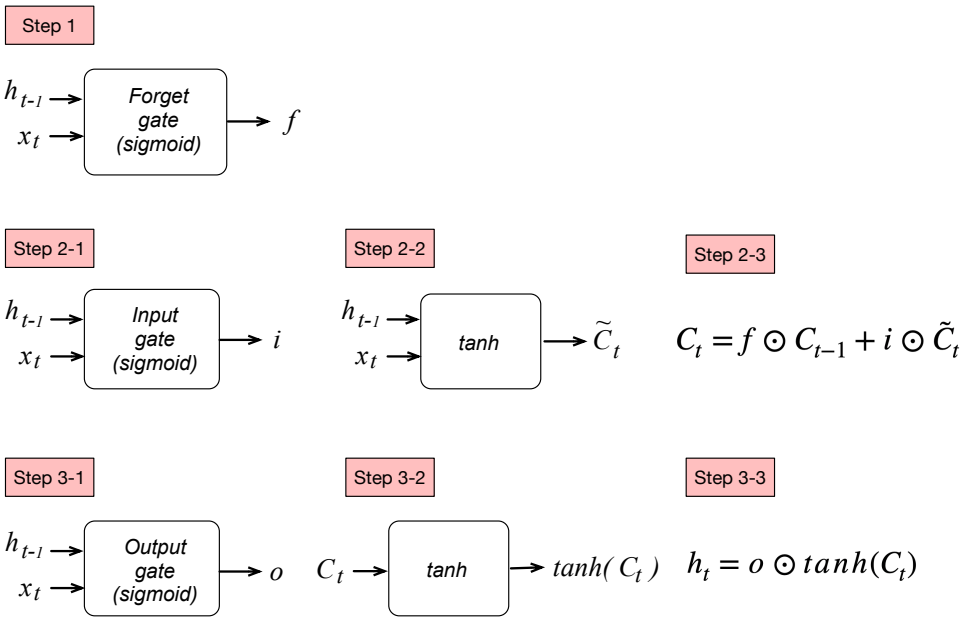


Figure 10-45: Summarization of the LSTM steps to construct short term memory and long term memory. Results of steps 3-2 and 3-3 are returned as outputs of the LSTM cell.

(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 10-45 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 10-45 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 10-45 Step 3-3). The final output of each LSTM cell is o along with its states (h_t and C_t).

Figure 10-45 summarizes the process of LSTM cell[16], it is an easy-to-remember summarization of the LSTM algorithm. We did not explain each W and b, are weight and biases. There is another form of visualizing LSTM, which is presented in Figure 10-46.

While using LSTM, if the states changes 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.

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.

---

[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.

GRU has two gates that use the Sigmoid function, **reset gate** (r) and **update gate** (z). The reset gate decides on *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} will be used to calculate the value for update and reset gates. They both use the *Sigmoid* function and their equations will be 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 having 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) 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.

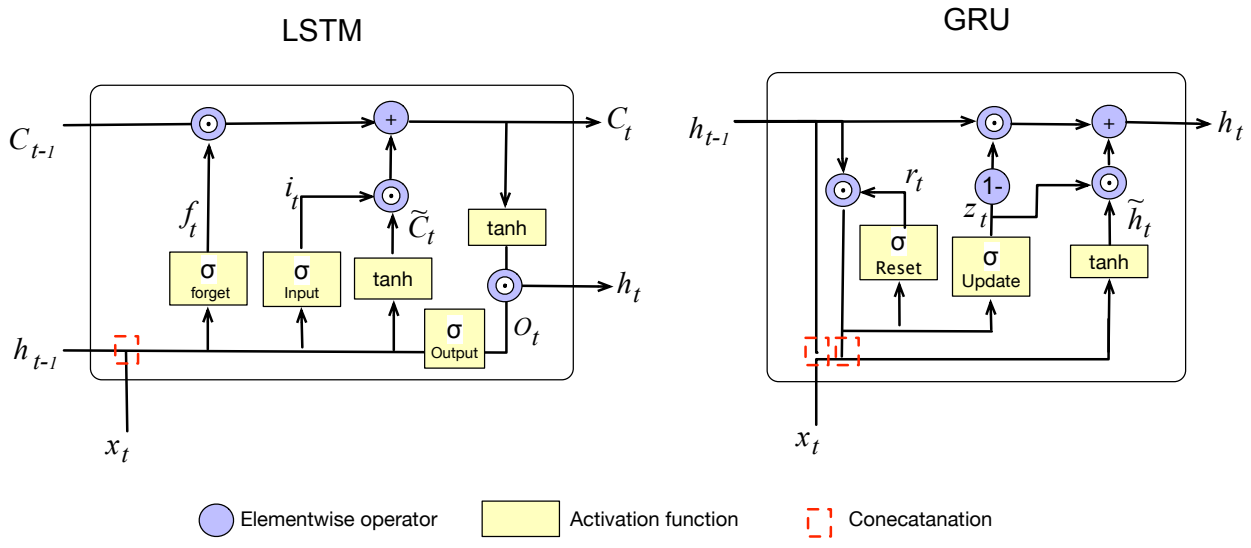


Figure 10-46: Visualization of LSTM and GRU architecture based on their gates and operators. In GRU output and new hidden state (h_t) are the same.

Based on the above equation, if z is close to 1, it means that the new hidden value (h_t) will be very similar to the old hidden value (h_{t-1}). This reveals the fact that lots of the new information

are a copy of old information. In contrast, z close to 0, indicates the new hidden value (h_t) will be close to the candidate hidden state ($\tilde{h}_t$).

Figure 10-46 presents a common visualization approach used by these two algorithms, inspired by Christopher Olah's design. In the beginning, we do not describe LSTM and GRU with these visualizations, because they might sound confusing, but now you can easily understand them.

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" mean. 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 can not 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 10-47 the unfolded version of one RNN cell (check Figure 10-42 to recall

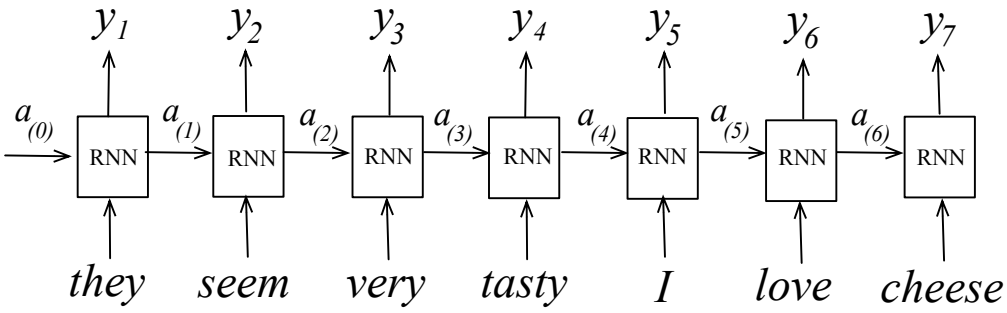


Figure 10-47: Unfold version of one RNN cell, which recieves one word at each time unit. '*a*' presents acitvation funciton result.

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 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 in the sequence for predicting data in a sequence. Bidirectional RNN is designed to *incorporate future data into the sequence* as

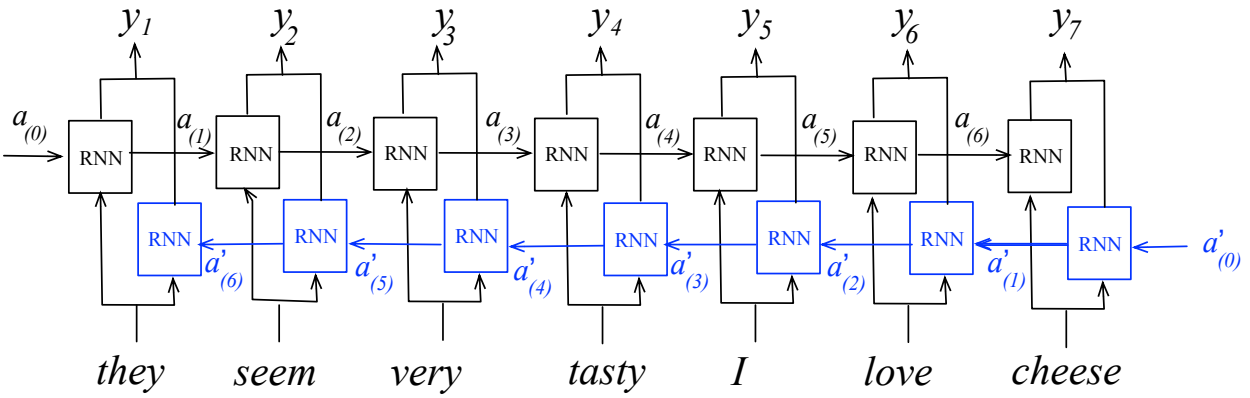


Figure 10-48: Unfold version of one BiDirectional RNN cell, which the blue layer is the same kind of RNN.

well. It introduces another layer, with exactly the same size, but the direction of activations is in opposite direction (from the end of the sequence to the beginning of the sequence). Figure 10-48 presents the Bi-directional RNN and to present the direction we use blue color to present the other RNN layer. 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 network in Figure 10-48 starts from $a_{(0)}$ and goes forward until it computes $a_{(6)}$, simultaneously it starts from the $a'_{(1)}$ (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). 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 can not work with stream of data (data will be available during the time).

Deep RNN

If we stack multiple layers of RNN on top of each other, we can call this **Deep RNN**. This model is in use when we need to model a non-linearity. Figure 10-49, presents a Deep RNN, which has three hidden layers.

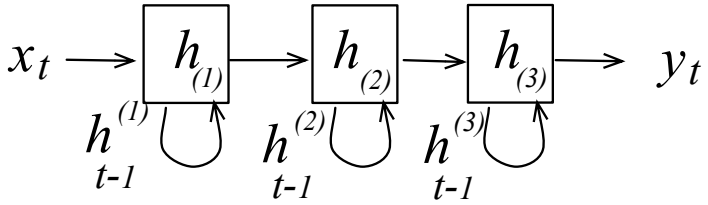


Figure 10-49: A deep RNN with three layers. Note, unlike previous figures it has three hidden layers instead of one hidden layer that is unfolded.

It is a common expectation while we work with neural networks to have enough data to increase the layers, which causes an increase in the accuracy of the model. However, it has its own tradeoff and usually, we are limited in increasing the number of layers, and after some point adding more layers causes a decrease in model accuracy.

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 about 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 10-50, 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

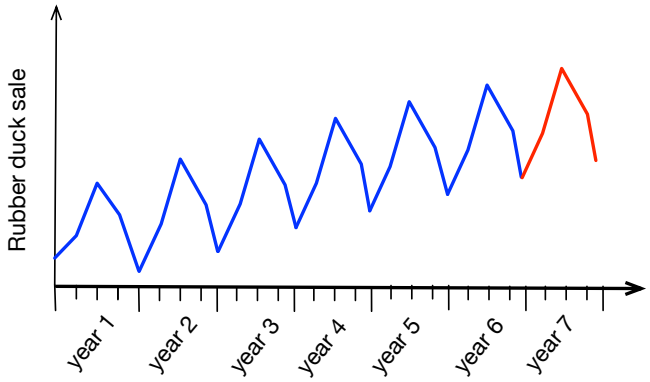


Figure 10-50: Another popular use of RNN is for time series prediction and here the red area is constructed by the algorithm, which has the blue part of the time series as its training set.

time series is a mock time series that show the sales of a rubber duck, as you can see the sale increases in summer and decrease in winter, in which 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 sentence as well. Definitely, the larger the text size the longer will be the sentence. Also it could guess the rest of the sentence. For example, “I do appreciate quick” could be followed by the word “response”, which this word is guessed or constructed by a 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 automatically learns text features, such as beginning of a sentence, quotes, punctuations, etc. It is similar that we fed the image into CNN and it learns image features such as edges, lines.

Figure 10-51, 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 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 word "the".

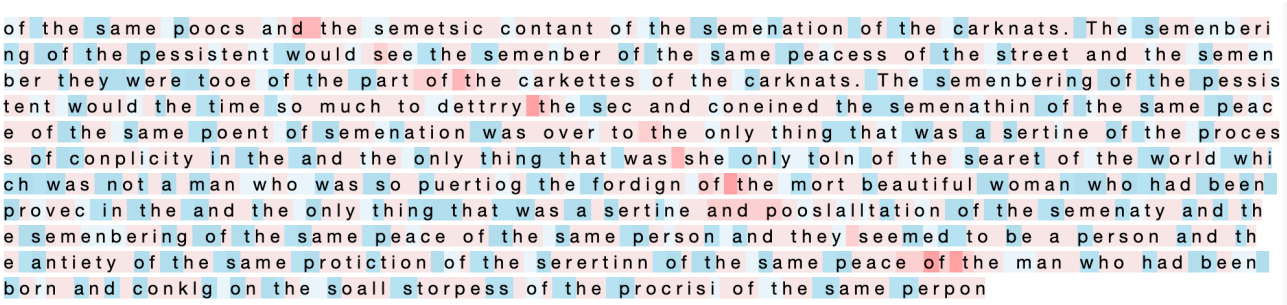


Figure 10-51: A sample feature (space between words and the character beginnng of the next word) that has been detected by one output cell of the LSTM model.

Since the author of this book is poor and not rich to train it with a server with more epochs, but we might get more human readable output if we increase the number of epochs.

Note that there are many more other features recognized by LSTM, but hard for us as humans to understand, due to the blackbox nature of deep neural networks. Perhaps if we ran the algorithm with larger epochs and stronger computational capability we were able to extract more human-understandable features.

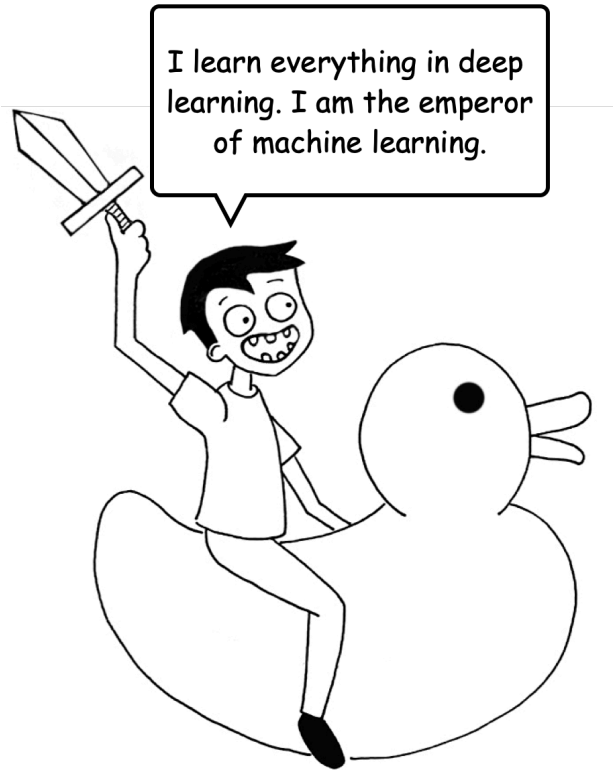
In another example, we use a GRU to generate a text automatically by looking at the text of the "100 Years of Solitude" book and learning the patterns of the text. Following is a sample text, created at the 30th epoch:

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 unerateraд 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 by the algorithm is called *Generative AI* and deep learning algorithms foster the path to Generative AI. In later chapters we will see more exciting examples of Generative AI.

NOTES:

[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.



* There are some other variation of LSTM [Greff '16]. For example, peephole LSTM, which enables *f*, *i* and *o* look at the cell state as well. 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. Nevertheless, it is recommended to experiment with both LSTM and GRU and decide based on comparing accuracies of results.
* The architecture of Bidirectional RNN has one similiarities with the Forward-Backward HMM (Check Chpater 5), because both of them incorporate future informaiton in a sequence to predict a data in a sequence.
* Bidirectional RNN are even more complex than other deep learning algorithms and enourmesly slow, because backward and forward propagation requires to operate on both directions.

Summary

In this chapter, first, we described some history and human efforts to resemble neural networks with computers. A neural network is composed of three layers, input, output, and hidden layers. An 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 have the lowest value. The objective of

a cost function in a neural network algorithm is to evaluate if the output of the network is correct based on loss score. If it is not correct, then correct 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.

Next, we explained the perceptron, its limitations, and how Multi Layer Perceptron (MLP) resolves its limitation. MLP algorithms are still in use and they are also referred to as traditional ANN. If we increase the hidden layers of MLP algorithms those are called a deep neural network. 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 “dense layer” or “fully connected layer”.

Afterward, we describe activation functions, which can enable the neural network to construct a non-linear model. Afterward, cost functions were explained 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 perhaps the most popular use of Deep Learning. To understand CNN models, we explained the concept of convolution and cross correlation. Then we explained 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 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 architecture that is 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 issue of vanishing gradient and implements both short term and long term memory. Besides, vanilla RNN has a very small memory and an LSTM cell has significantly more memory than traditional RNN. 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 that is less complex than LSTM. To review LSTM and GRU operations, let's compare all equations[18] required for LSTM and GRU cells.

---

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

LSTM:

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

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 on the differences between LSTM and GRU, you can check Figure 10-46 as well.

At the end of this chapter briefly, we described Bidirectional RNN. Bidirectional RNN has the advantage to learn from both directions of a sequence but it is very 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.

Further Reading or Watching

* Tareq Rasheed [Rasheed '16] has a short book about neural network, and make a very good connection between biological and artificial neural network. It also guides the reader to build a small neural network.
* Francis Chollet [Chollet '18] has an excellent good introduction to deep learning and its components in his book. He is the original author of the Keras platform[19] as well.
* 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

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

and Kian Katanforoush notes are among the good ones that we can understand. http://cs229.stanford.edu/notes/cs229-notes-deep_learning.pdf.

* Jay Alammar provides a very good interactive tool to learn linear regression and Gradient Descent. You 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/

* Wikipedia has a complete list of all activation functions, if you are interested in learning more about activation functions you can check this link: https://en.wikipedia.org/wiki/Activation_function

* 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 detail of Convolution if you are interested to learn 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 animation from here: https://github.com/vdumoulin/conv_arithmetic

* Christopher Olah has a fantastic and easy to understand, visualized explanation about LSTM, we benefited a lot from his explanation to understand LSTM. His explanation and visualization are probably the most used ones to describe the LSTM and are available here: https://colah.github.io/posts/2015-08-Understanding-LSTMs

* There are courses that provide good introduction to Deep Learning on Udemy, we have used the course provided 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) both are useful and good. However, the material they cover is not very broad and consider them as a start of your learning journey.