CARLETON UNIVERSITY

SCHOOL OF
MATHEMATICS AND STATISTICS

HONOURS PROJECT

TITLE: Image Recognition by Convolutional Neural Networks

AUTHOR: Dicong Zhao

SUPERVISOR: Emmanuel Lorin

DATE: May 6th, 2020

Abstract

With the rapid development of technology, Machine Learning, which is a study of mathematical model that computers can use to make predictions, has been an important role in our daily life. In particular, Deep Learning, which is one of the main categories in Machine Learning but based on Neural Networks (NN), provides effective algorithms in the new technology such as self-driving car or intelligent robots. In this report, we show one of the main challenges when developing such new technology, which is Image Recognition. Convolutional Neural Network (CNN), the most common use method in visual imaging, is demonstrated with one of the most recent image recognition libraries, AlexNet, by analyzing how the layers are constructed and how the data are getting trained in mathematical views. We also implement an application that is trained by CNN in MATLAB by using one of the open-source training data, and mathematically explain the application. The paper is concluded with a wide discussion of questions related to the Convolution Neural Network for future investigation.

Acknowledgement

I would like to express my deepest appreciation to my supervisor Professor Emmanuel Lorin, who is a patient and friendly professor, always showing me practical suggestions and extensive knowledge throughout the duration of this project. Thanks again for helping to maintain a good pace of the project during the whole term even some extreme conditions happened.

Table of Contents

List of Figures

1 Introduction

Since 2012, AlexNet, a Convolutional Neural Networks (CNN) designed by Alex Krizhevsky *et al.*, has leaded image recognition to another century. AlexNet trains a network with 1000 different classes by using 1.2 million images [3]. Although the accuracy of the model may not be the highest anymore, it provides a better approach to image recognition for future researchers, which results in a more concrete model such as VGGNET (2014), GoogLeNet (2014) and Residual Network (ResNet in 2015). Let us notice that they all contain at least one convolutional layer which is an indication of CNN. Since the introducing of AlexNet, CNN has been proven to be the most effective algorithm in image recognition [5].

Current thesis reports to CNN are summarized in a theoretical manner, which provides an overview of each layer without mathematical proof [2, 3]. To begin with, some of the basic mathematical definitions and properties are displayed in the first section. The next section, Convolutional Neural Networks, introduces some of the important layers in CNN with detailed explanations. Furthermore, the third section, training CNN, provides additional knowledge which is essential in training a CNN followed by a simple application. A more complex example is given in the fourth section which demonstrates the overall training and prediction process in more details.

1.1 Machine Learning Overview

At a very high level, Machine Learning (ML) is a process of teaching a computer to learn and predict from data. It is also an algorithm or a mathematical model which provides a computer with the ability to perform tasks without explicit instructions. There are several fundamental parts that every ML algorithm follows such as gathering data, preprocessing data, training model and evaluating (or testing). *Gathering data* is the process of collecting the data from the outside world, where the majority of the data, known as *training data*, are used to train the mathematical model. The rest of them, *testing data*, are used during the evaluation process. Training data is made up of the form (x, y), where x is the input data and y is the corresponding label for that data. For example, x can be a cat image and y can be a correct label for that data, such as $y = 1$. *Preprocessing data* is the process of generalizing the data in a more standard form. For example, given a huge amount of image data, the size of each individual image should be fixed in a way that is suitable for the training. *Training model*, the main component of ML, is, informally, an operation that provides the ML algorithm with training data to learn from. Formally, *training* is a loop

procedure that repeats until all the *weight* and all the *bias* are determined (See below). *Model* is a mathematical artifact generated from the training process. Finally, *Evaluation*, also known as *testing*, is an operation which measures the accuracy of the generated model from the training process given the testing data. It is noticed that the goal is to predict the label of new data.

Neural Networks (NN) is a set of ML algorithms that are designed to uncover the hidden pattern loosely modeling how human beings take decisions. *Artificial Neural Networks (ANN)* is a particular type of NN but made up with *artificial neurons*, which loosely model neurons in a biological brain. Each *artificial neuron* (or *layer*) consists of a non-linear function (or *activation function*), *weight* and *bias*. A single *artificial neuron* is typically defined as

$$f(x) = g(Wx + b) \tag{1.1}$$

where $x \in \mathbb{R}^n$ is the input data, $f(x) \in \mathbb{R}^m$, $W \in \mathbb{R}^{m \times n}$ refers to the *weight* matrix, $b \in \mathbb{R}^m$ refers to a bias vector and g is an *activation* function (Figure 1.1). *Weight, W,* is a *learnable parameter* inside the neuron function, which mainly transforms the input x into the next neuron. *Bias, b,* is also a *learnable parameter* which represents how far off the predictions are, from their intended value. Note that a *learnable parameter* is a variable that updated during the training process in order to obtain the optimal prediction result. There are three main types of common activation functions, the *sigmoid* function, the *hyperbolic tangent* function and the *rectified linear unit (ReLU)* function. The *Sigmoid* function is defined as

$$g(x) = \frac{1}{1 + e^{-x}}. \tag{1.2}$$

The *Hyperbolic tangent* function is defined as

$$g(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{1.3}$$

and the *ReLU* function is defined as

$$g(x) = \max(0, x). \tag{1.4}$$

Furthermore, completed ANN involves multiple artificial neurons. For example, if an ANN has two neurons (or two layers), it is defined as the following equation

$$f(x) = g(W_2\, g(W_1\, x + b_1) + \, b_2) \tag{1.5}$$

where $x \in \mathbb{R}^n, W_1 \in \mathbb{R}^{m_1 \times n}$, $b_k \in \mathbb{R}^{m_1}, W_2 \in \mathbb{R}^{m_2 \times m_1}, b_2 \in \mathbb{R}^{m_2}$, $f(x) \in \mathbb{R}^{m_2}$. It is noted that W_1 and b_1 represent the weight and bias for the inner neurons (or layer 1) and W_2 and b_2 refer to the weight and bias for the outer neurons (or layer 2). In general, the output of the last neurons function is treated as an input to the next neurons function.

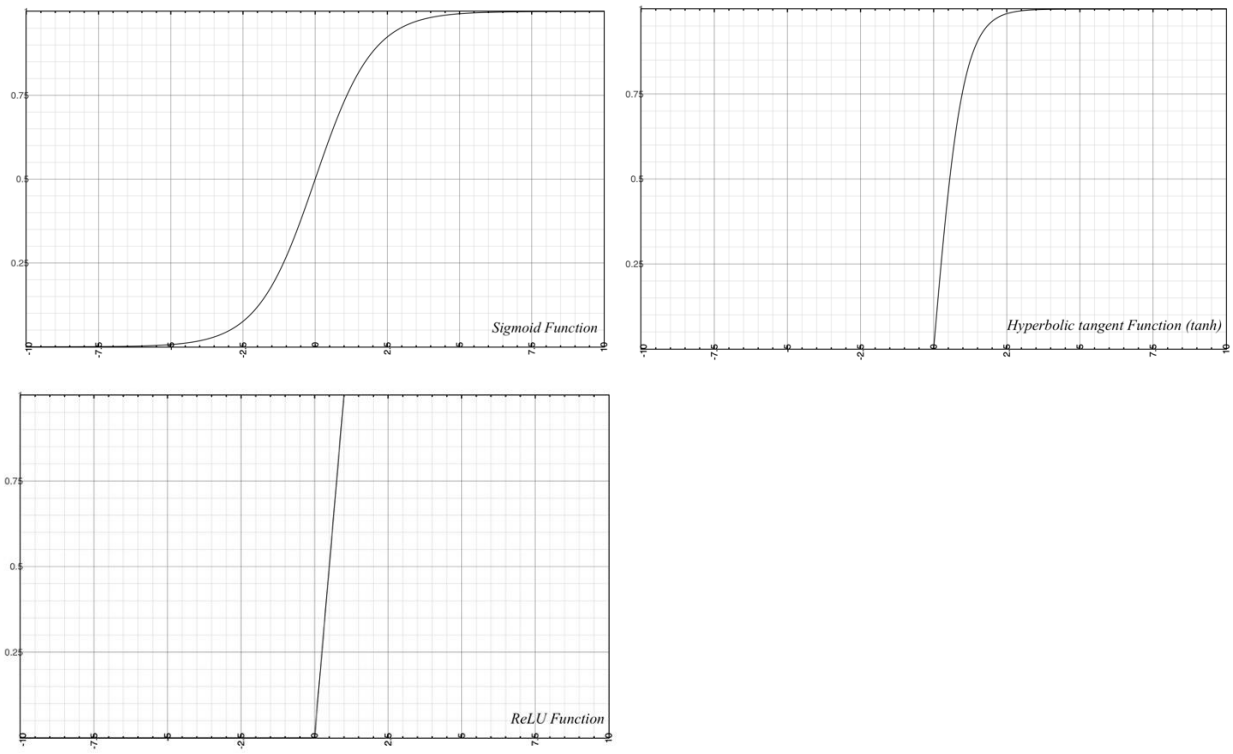

Figure 1.1 Sample Monotonic (or Activation) Functions: sigmoid function (top left), hyperbolic tangent (top right) and ReLU function (bottom left).

1.2 Bayes' Rule

The probability distribution of a discrete random variable is called the *probability function*, which is a function of X, P: $\mathbb{R} \rightarrow [0, 1]$, given by

$$P(X = x)\,, \tag{1.6}$$

where X is a set of possible outcomes and each outcome ("$X = x$") is called an *event*.
One of the most important theorems used in ML is the *Bayes' Rule* (or *Bayes' Theorem*) which describes the probability of an event given that another event happened, and states as follow

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)} \tag{1.7}$$

where A and B are events and $P(B) \neq 0$.

In Machine Learning (ML), Bayes' Rule is used to determine the value from the probability distribution function given that we have knowledge about the data. In general, we can transform the Bayes' Rule into a more readable form in ML [1],

$$P(model \mid data) = \frac{P(data \mid model)\, P\,(model)}{P(data)}, \tag{1.8}$$

where

i) $P(model \mid data)$ is called the *posterior* probability, which is the probability of a hypothesis (model) given the observed evidence (data).
ii) $P(data \mid model)$ is called the *likelihood,* which is the probability of data given the model.
iii) $P(model)$ is called the *prior* probability, which is our belief of the model before investigating the data.
iv) $P(data)$ is called the *evidence*, which is the same for all the hypotheses. Note that, in general, the evidence is not in the process of training the network.

In general, the goal of Bays' rule is to use the posterior probability to predict any new data, which is the probability of the new data given we have a knowledge of trained data. Moreover, if we know the

distribution of the training data, we can assume the likelihood ($P(data \mid model)$) to be the probability distribution function and compute the optimal parameters by maximum likelihood method [3]. For example, if the data is modeled with Gaussian distribution, we can assume the likelihood function of the form (Equation 1.9) where $G(\mu, \Sigma)$ denotes the Gaussian distribution with mean μ and standard variance Σ.

$$P(x \mid C_i) = G(\mu_i, \Sigma_i) \tag{1.9}$$

where C_i is the i^{th} class that the data belongs to, μ_i and Σ_i are the mean and standard variance for the i^{th} class, and x is the input data [3].

However, the distribution of the data may not be easily estimated if the training dataset is huge. In this case, we perform the learning from the conditional probability of the data given the model directly [3]. If we assume

$$P(C_i \mid x) = \frac{1}{1 + e^{-a(x)}} = g\big(a(x)\big) = \frac{1}{1 + e^{-w^T x}} \tag{1.10}$$

where $P(C_i \mid x)$ is a posterior function and g is an activation function. Note that such posterior function is used to create a model called the Logistic Regression. Furthermore, if we replace it with Equation 1.5, then the posterior function is called an ANN. In other words, the posterior function is interchangeable depending on the requirement of the problem.

2 Convolutional Neural Networks (CNN)

CNN, which is a type of deep neural networks (DNN), is widely used in image classification and was originally presented by Fukushima in 1988 [8]. Due to computing hardware restriction, CNN had not been found as useful as other DNN until Alex, Ilya and Geoffrey developed AlexNet (Figure 2.1) with efficient GPU implementation [2]. 1.2 million images were retrieved to train the network with 1000 distinct classes, which only had the top-5 test error rate of 15.3% [2]. It is worth mentioning that the training was done within a week by using 2 GPUs in 2012 [2]. Compared with the other DNN, there are several benefits of using CNN. For example, CNN is much closer to the way that human beings do visual processing. Feature extraction, one of the most important aspects of the networks, can be done effectively in CNN. Furthermore, the max pooling layer (see below) of CNN is powerful in reducing the dimension of the image without affecting the feature [8].

Figure 2.1 shows the architecture of AlexNet, which demonstrates a general structure of CNN. Each of the small section, which contains certain tasks, is called a *layer*. Firstly, there are three main types of layers in CNN, which are *convolutional layer*, *max pooling layer* and *fully connected layer*. Secondly, layers inside the network are connected with each other and use the output of the previous as its input so as neuron behavior. Furthermore, the main task for the first two layers, convolutional layer and max pooling layer, is to achieve feature extraction, while the last layer is considered as classification which normally has one or more fully connected layer [2, 4]. In particular, the last fully connected layer maps the features into an actual object with probability output. Lastly, similar to AlexNet, a classic CNN consists of blocks of layers, where each block contains several sub-layers such as convolutional layer, ReLU layer or max pooling layer.

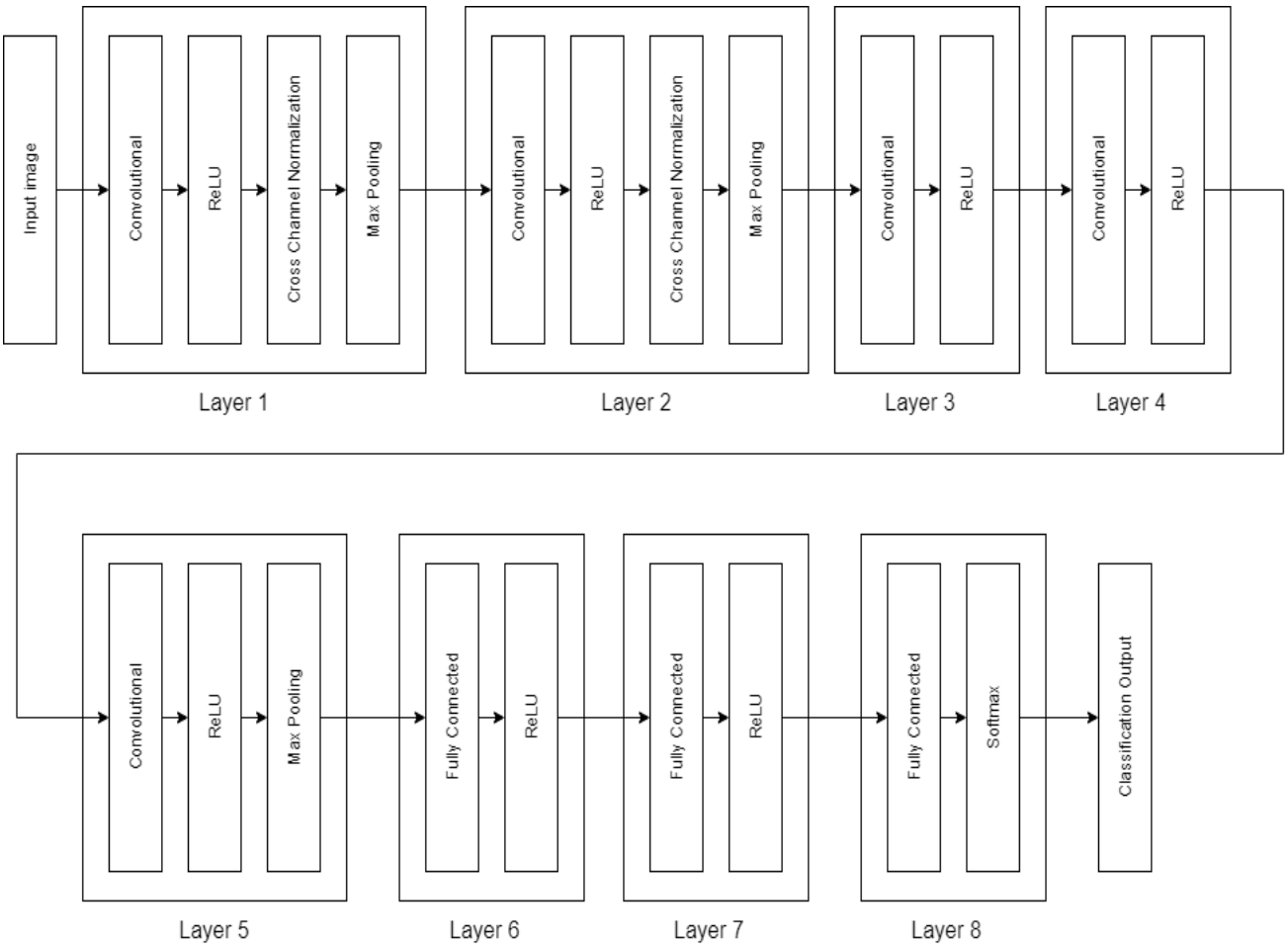

Figure 2.1 Architecture of AlexNet with 8 layers, where the first five blocks of layers are convolutional layers and the rest are fully connected layers.

2.1 Input Layer

In a general CNN, like AlexNet, the input to the first layer is not the image itself, but a so-called *channel* [6]. *Channel* refers to a certain component of an image. For example, an RGB image has three channels red, green, and blue. There are $F_0 = 3$ channels to the first layer, where each channel corresponds to each of the RGB colors (Figure 2.2). Each of the RGB channel, together, forms the first input layer.

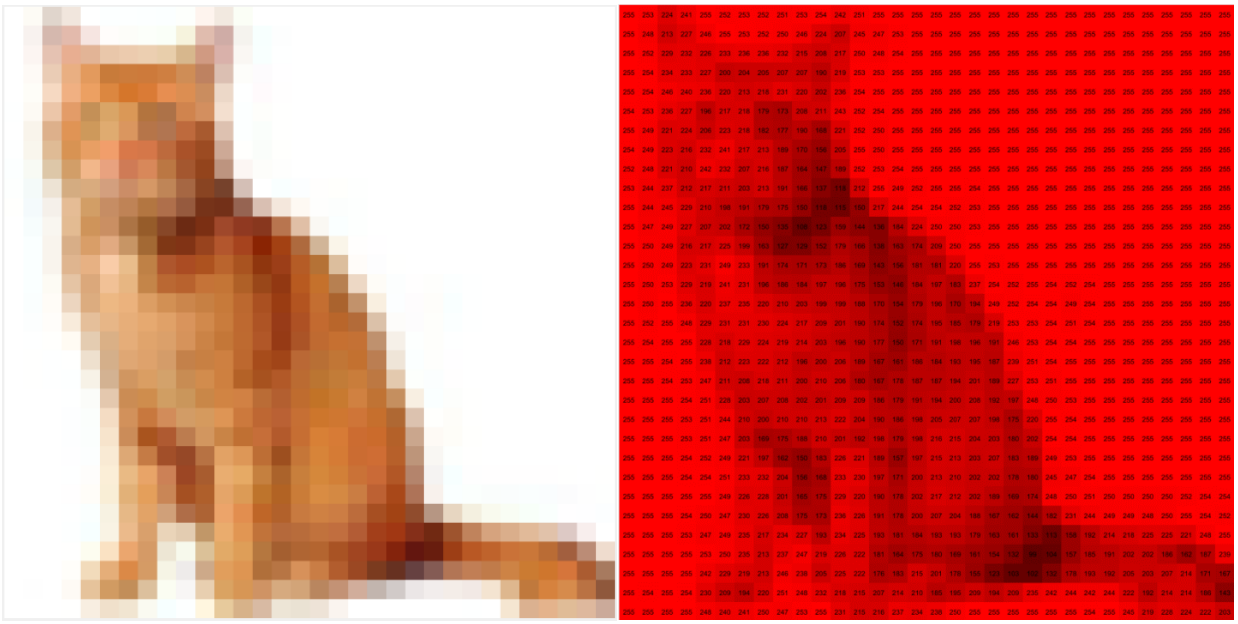

Figure 2.2 Sample input layer: A cat images with size 32×32 on the left and the corresponding red channel of the image with pixel value on the image.

2.2 Convolutional Layer

Convolutional layer, one of the most important layers in CNN, provides the essential functionality to feature extraction. Like its name the operation convolutes a feature of an input layer with two learnable parameters, weight and bias, to generate the output *feature map* [6]. The operation is done by a mathematical linear operation, called convolution product. Let us recall the convolution of two integrable functions f and g, denoted as $f * g$, is defined as the following,

$$(f * g)(t) \triangleq \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau \,, \tag{2.1}$$

where t is an input variable in $\mathbb{R}$.

In order to obtain the most appropriate output *feature map*, the value of the weight and bias are what need to be discovered in the later optimization process. *Feature map* refers to the output feature for one given *filter* (see below).

First, all the variables are under real number field ($\mathbb{R}$), and we define some of the important terminologies below. *Filter* (or *kernel*) is a three-dimensional matrix of size $K \times K \times F$, where $K \times K$ refers to the single aspect of a channel and F refers to the number of channels. The channel is also known as the *depth of the image* since the number of channels in a filter is equal to the number of channels for the input image. The *depth of the filters* is the number of filters, which also refers to the number of output feature maps from the convolutional operation. For example, in the first convolutional layer of the AlexNet, given the $227 \times 227 \times 3$ input image, the filter is of size $11 \times 11 \times 3$ with the depth of the filters equal to 96. Note that the number of channels in the filter is equal to the number of channels for the input image. Then, the convolutional layer produces 96 output feature maps ($55 \times 55 \times 96$). The output dimension is computed via Equation 2.8 below.

It is worth mentioning that the convolutional operation is a combination of four summations. From inner to outer, the first two sums over one channel of a filter ($K \times K$), the third one sums over the *channel* (F), and the last one sums over the *depth of the filters* (D). Regarding how the operation applies to an image, we first consider $F = 1$, $D = 1$, and start from an image of size 5×5 and a filter (or kernel) of size 3×3 (Figure 2.3). The convolution is computed by sliding the filter over the image, starting from the top left corner. Then, the first value of the output matric is obtained by summing the product of the corresponding pixel value and the filter value. Similarly, the final output is computed by moving the filter from left to right and from top to bottom.

I_{11}	I_{12}	I_{13}	I_{14}	I_{15}
I_{21}	I_{22}	I_{23}	I_{24}	I_{25}
I_{31}	I_{32}	I_{33}	I_{34}	I_{35}
I_{41}	I_{42}	I_{43}	I_{44}	I_{45}
I_{51}	I_{52}	I_{53}	I_{54}	I_{55}

K_{11}	K_{12}	K_{13}
K_{21}	K_{22}	K_{23}
K_{31}	K_{32}	K_{33}

Figure 2.3 Demonstration of input map and a filter: A 5×5 images with a single channel on the left and a 3×3 filter on the right. Note that each value from the image or filter is a pixel value.

In other words, we can define the operation mathematically for one single channel as [7]

$$O(i,j) = \sum_{k=1}^{K}\sum_{l=1}^{K} I(i+k-1, j+l-1)K(k,l) \tag{2.2}$$

where $K \times K$ is the size of the filter, O denotes output, and $O(i,j)$ is the value of the i^{th} row and j^{th} column of the output matrix. Furthermore, i is running from 1 to $W - K + 1$ and j is running from 1 to $H - K + 1$ where $W \times H$ is the size of the image [7]. For example, the $O(1,1)$ value from the above example is

$$O(1,1) = \sum_{k=1}^{3}\sum_{l=1}^{3} I(1+k-1, 1+l-1)K(k,l) \tag{2.3}$$

$$\begin{aligned} &= I_{11}K_{11} + I_{12}K_{12} + I_{13}K_{13} + I_{21}K_{21} + I_{22}K_{22} \\ &\quad + I_{23}K_{23} + I_{31}K_{31} + I_{32}K_{32} + I_{33}K_{33} \end{aligned} \tag{2.4}$$

where we have set $I_{ij} = I(i,j)$ for $1 \leq i,j \leq 5$ and $K_{kl} = K(k,l)$ for $1 \leq k,l \leq 3$.

As a result, a 4×4 matrix is produced from the above example. Now, if we consider a normal image with RGB colors, which means $F = 3, D = 1$, the final matrix is formed by summing all the corresponding value from each of the channel output matrices. The result can be defined as

$$O'(i,j) = \sum_{v=1}^{F} O_v(i,j) \tag{2.5}$$

where O_v denotes the output matrix from the v^{th} channels, F is the number of channels and O' is the output feature map.

2.2.1 Padding and Stride

As we can notice, the outer pixel value of an image is only involved once in one of the operations, whereas the inner pixel value of an image is involved multiple times. Moreover, compared to the original image, the dimension of the output feature map has been reduced. For example, in the above example, the

original image has size 5×5, while the output has size only 4×4. Padding (P) is added around the images with value 0 to preserve the width and height of the original image [6] (Figure 2.4).

If the original input has size $W \times H$, then, after adding the padding, the output has a size

$$(P + W + P) \times (P + H + P) \tag{2.6}$$

where each P corresponds to the padding size in each direction of the input image. The formula is simplified as

$$(W + 2P) \times (H + 2P)\,. \tag{2.7}$$

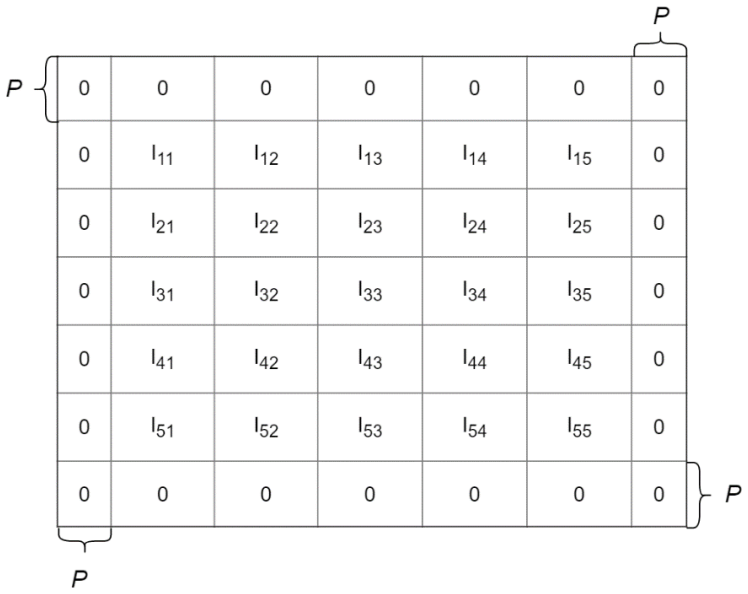

Figure 2.4 Demonstration of padding: A matrix (Or image) with padding equal to 1 ($P = 1$). The zeros are added once to each direction of the image.

Another important term to be introduced here is called *stride* (*S*), which is the distance between the movement of the filter (Figure 2.5). In general, the stride value for the convolutional layer is 1 and it is widely used in the max pooling layer (see Section 2.3) [4].

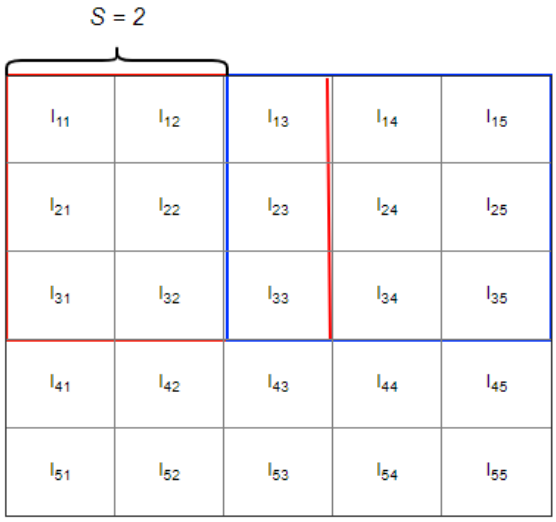

Figure 2.5 Demonstration of stride: The graph represents a movement of a filter with stride ($S = 2$) from the first position (red area) to the next position (blue area).

In conclusion, if the size of the filter is $K \times K$, then the output width (W_o) and height (H_o) are

$$W_o = \frac{W + 2P - K}{S} + 1, \qquad H_o = \frac{H + 2P - K}{S} + 1 \tag{2.8}$$

Also, in order to preserve the width and the height of the original image, common padding is

$$P = \frac{K - 1}{2} \tag{2.9}$$

where K is the dimension of the filter.

Figure 2.6 shows a general convolution operation applied to a real image with size 32×32, a filter of size 3×3, no padding and 1 be the stride value. In realistic, there is more than one filter applied to the input to extract more feature. For example, in AlexNet, the first convolution layer applies 96 filters with size $11 \times 11 \times 3$ and a stride of 4 to the input image with size $227 \times 227 \times 3$, where the 3 represents the RGB colors [3].

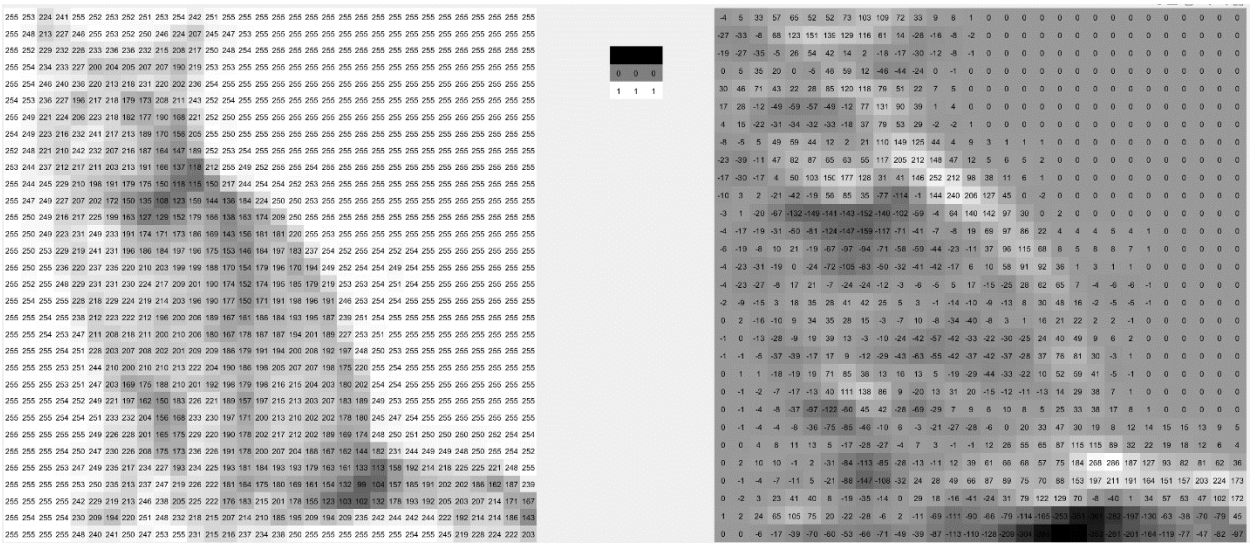

Figure 2.6 Sample convolution operation on an image: A convolution operation is applied to the red channel of the cat image (left) with a 3 × 3 filter (middle), no padding and a stride of 1 resulted in an output feature map on the right.

2.2.2 Parameters

In general, the convolutional operation is a matrix multiplication which can be expressed as a single neuron function without an activation function

$$f(x) = W'x + b \tag{2.10}$$

where $x \in \mathbb{R}^n, f(x) \in \mathbb{R}^m,\ W' \in \mathbb{R}^{m \times n}$ is a weight matrix and $b \in \mathbb{R}^m$ is a bias. The high dimensional filter is reshaped into a 2D weight matrix so as the image input. A simple example is used here to demonstrate the intuition behind it. Consider an image with the following settings, $W = 4, H = 4, K = 2, F = 3, D = 2,$ and $S = 1$. In other words, the input RGB image is of size $4 \times 4 \times 3$ ($W \times H \times F$), the filter is of size $2 \times 2 \times 3$ ($K \times K \times F$) with the depth of the filters (D) equal to 2, and the stride (S) is 1 (Figure 2.7).

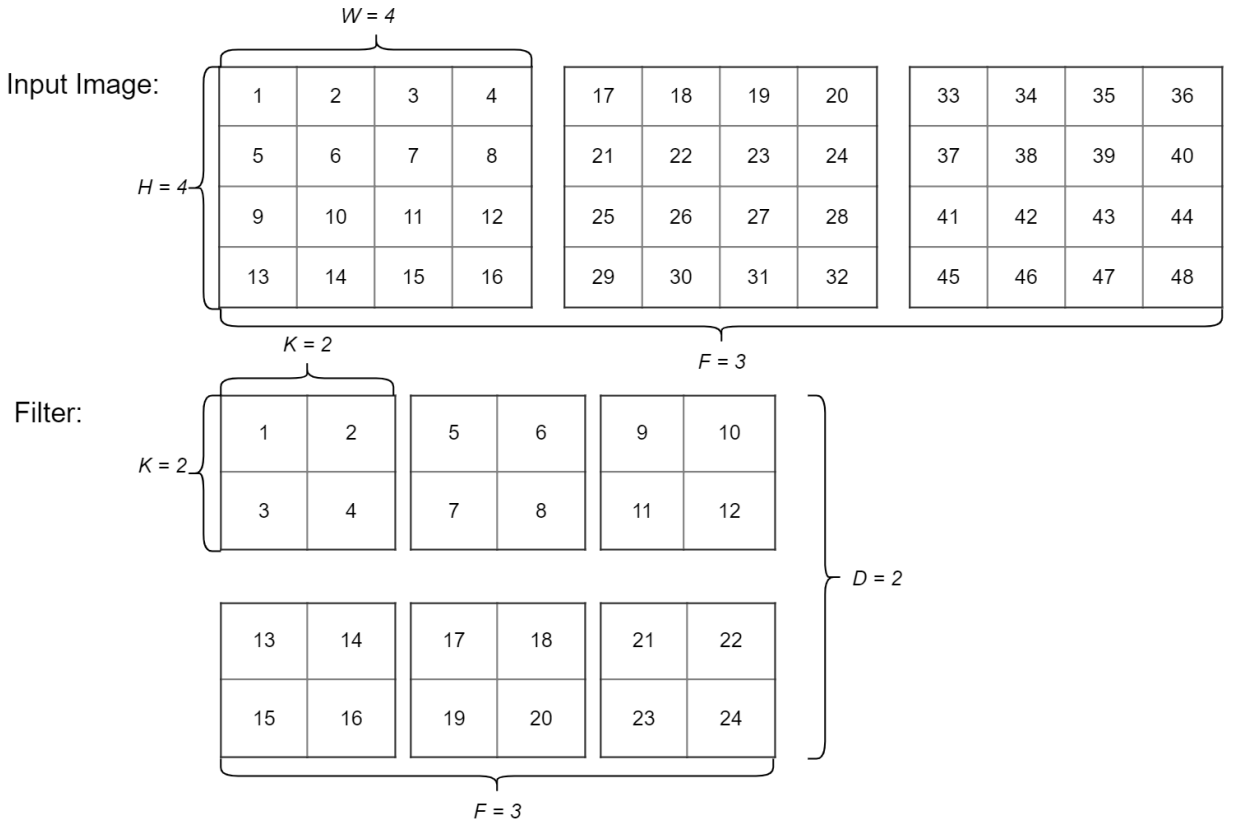

Figure 2.7 Demonstration of input image and filter: The input image has size $4 \times 4 \times 3$, and the filter has size $2 \times 2 \times 3$ with the depth of the filter equal to 2.

The conversion result of the filter and the input image is provided in Figure 2.8. The first column of the reshaped x is formed by merging the filter area from each channel. In other words, the first filtered areas for the input image are

$$\begin{bmatrix}1 & 2\\5 & 6\end{bmatrix},\begin{bmatrix}17 & 18\\21 & 22\end{bmatrix},\begin{bmatrix}33 & 34\\37 & 38\end{bmatrix} \tag{2.11}$$

from channel 1, channel 2, and channel 3, respectively. Then we flatten the matrix by row into the first column of reshaped x.

$$[1\ 2\ 5\ 6]^T\ [17\ 18\ 21\ 22]^T\ [33\ 34\ 37\ 38]^T \tag{2.12}$$

$$[1\ 2\ 5\ 6\ 17\ 18\ 21\ 22\ 33\ 34\ 37\ 38]^T \tag{2.13}$$

Later, we combine every separated vector (Equation 2.12) into one vector (Equation 2.13) to be the first column of the output matrix. The same method applies to the rest of the columns. Note that since the

stride is 1 ($S = 1$), we have 9 columns in total for the reshaped x by taking the filter from left to right and top to bottom. Similarly, the new weight matrix is obtained from the filter where the number of rows is the depth of the filters ($D = 2$). Each row is formed by concentrating the flattening form of the individual channel matrix.

W'

1	2	3	4	5	6	7	8	9	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24

x

1	2	3	5	6	7	9	10	11
2	3	4	6	7	8	10	11	12
5	6	7	9	10	11	13	14	15
6	7	8	10	11	12	14	15	16
17	18	19	21	22	23	25	26	27
18	19	20	22	23	24	26	27	28
21	22	23	25	26	27	29	30	31
22	23	24	26	27	28	30	31	32
33	34	35	37	38	39	41	42	43
34	35	36	38	39	40	42	43	44
37	38	39	41	42	43	45	46	47
38	39	40	42	43	44	46	47	48

Figure 2.8 Result of reshaping: The red area refers to the first filter area in Channel 1 of the input image, the green area refers to the first filter area in Channel 2 of the input image. Similarly, the blue area is coming from the first filter area of Channel 3 of the image.

The total number of values from all the learnable parameters is a critical measure of the complexity of a CNN [2]. The number of values in this learnable parameter, W', is

$$Number\ of\ values = (K \times K \times\ F + 1) \times D \tag{2.14}$$

where K is the size of the individual channel, F is the number of channels, D is the depth of the filters and 1 refers to a single bias for the current filter [9]. For example, the first convolution layer in AlexNet contains

$$Number\ of\ values = (11 \times 11 \times 3 + 1) \times 96 = 34944. \tag{2.15}$$

Moreover, AlexNet architecture has around 60 million values to be determined at the end of the networks, which is trained in two GPUs [3].

2.3 Pooling Layer

The pooling operation, or sub-sampling layer, integrates a down sampled operation to the input feature map, which is also called dimension reduction operation [2]. It is worth pointing out that the pooling operation does not require to train more parameters. In general, there are two kinds of pooling operations, one is called average pooling, and the other one is called max pooling. Similar to convolutional layer, the pooling layer also involves hyperparameters including filter size (R), stride (S) and padding (P) [4].

Max pooling, one of the most populate types of pooling operation, is essentially a max function, which filters the input map and takes the maximum value from the filtered region (Figure 2.9). The operation is defined by

$$O(i,j) = max_{i \le l \le i+R-1, j \le k \le j+R-1}\{I(l,k)\} \tag{2.16}$$

where $O(i,j)$ is the value of the i^{th} row and j^{th} column of the output matrix. As an example (Figure 2.9), with a filter size equal to 2 ($R = 2$) and stride equal to 1 ($S = 1$), an output feature matrix of size 4×4 is computed by taking the maximum value inside the filtered space. The first value is computed by

$$O_{11} = \max\{I_{11}, I_{12}, I_{21}, I_{22}\} \tag{2.17}$$

where we have set $O_{ij} = O(i,j)$ and $I_{lk} = I(l,k)$. Compared to the input map, the dimension of the output map is reduced by 1.

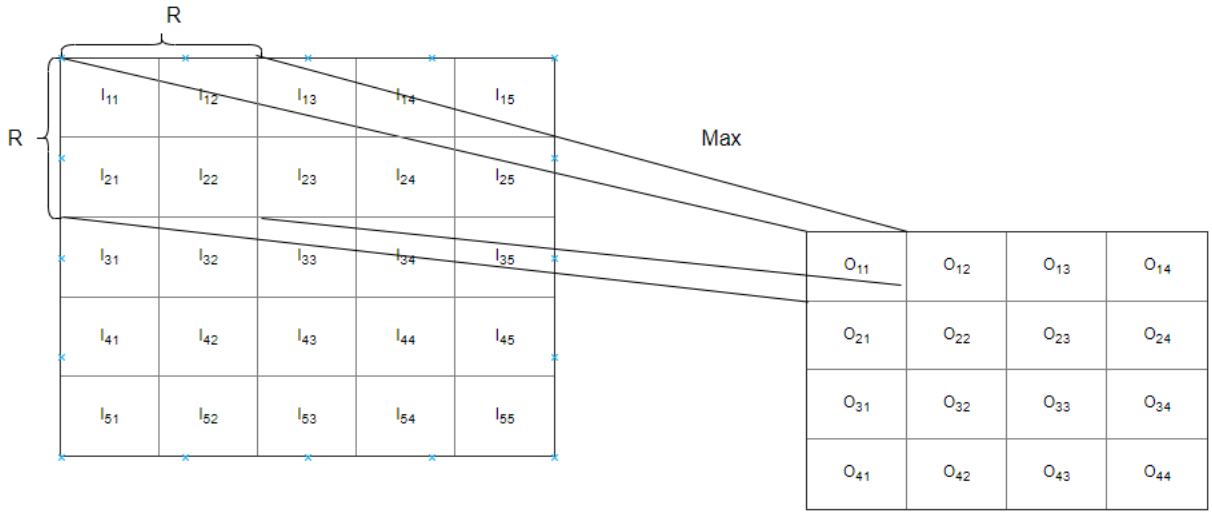

Figure 2.9 Demonstration of max pooling operation: A sample max pooling operation of an input feature map of size 5×5 with filter size ($R = 2$), stride ($S = 1$) and no padding resulted in an output of size 4×4.

2.4 Activation Function (ReLU)

Rectified linear unit (ReLU), a nonlinear activation function, acts mathematically as a biological neuron[4]. Although the traditional activation functions, such as sigmoid or hyperbolic tangent (tanh), also perform the same task, it has been observed that a CNN with ReLU as an activation function trains much faster than the network with hyperbolic tangent as its [3]. Furthermore, ReLU is a very simple function (Figure 1.1), which is defined by

$$f(x) = \max(0, x) \tag{2.18}$$

where the function is linear for $x > 0$ and 0 when x is negative (Figure 2.10). In other words, the function changes all the negative value with 0 and the positive value remains the same. Note that same as pooling layer, there are no parameters needed to be trained.

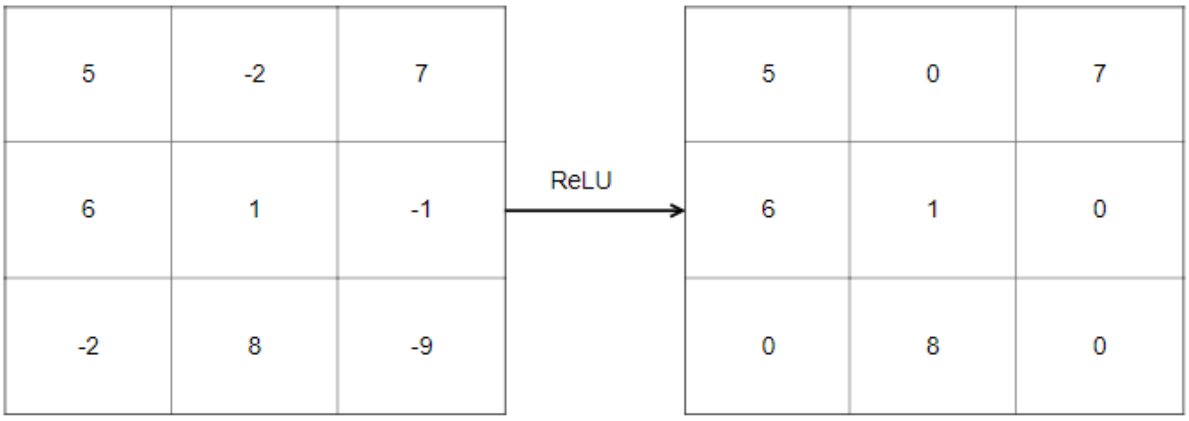

Figure 2.10 Demonstration of ReLU operation: A ReLU operation on a sample matrix with input on the left and output on the right.

2.5 Fully Connected Layer

After the previous operation (pooling or convolution), fully connected layer, like its name, connect all the input feature maps from the previous layers to a larger feature map. In other words, the operation transforms the multi-dimensional input matrix, possibly of size $K \times K \times F$, from the previous layer to a vector (one-dimensional array) [4]. Note that there are more than one fully connected layers in general and each of the layers is followed by an activation function. In particular, the last fully connected layer inside a CNN connects all the features to the actual predicted objects (such as cat or dog), which will provide a score for each object.

The operation inside this layer is relatively simple as well, which simply flatters the output (such as input map of size $6 \times 6 \times 256$) from the previous layer as input to a one-dimensional vector (a 9216×1 vector) [3]. Note that for the first fully connected layer in CNN, the input feature map is of size $K \times K \times F$, which is transformed into one huge vector by taking it row by row. The transformation is similar to the conversion of filters into a weight matrix in the convolutional layer. Then, the vector is multiplied by a weight matrix (W) plus a bias (b). The operation is the same as a neuron function in Equation 1.1 with a weight matrix, a bias, and an activation function.

Considering the first fully connected layer in AlexNet with a transformed vector of size 9216×1, we initialize a weight matrix of size 4096×9216, which would have an output vector of size 4096×1[3].

Then, the output is treated as an input to the next fully connected layer, which is the last fully connected layer in AlexNet and maps to 1000 distinct classes.

2.6 SoftMax Layer

In this layer, the input one-dimensional array is translated into a vector with probability for each class via a SoftMax function. The SoftMax function is defined as

$$f_j(x) = \frac{e^{y_j}}{\sum_{i=1}^{N} e^{y_i}} \tag{2.19}$$

where K is the number of classes, $\boldsymbol{y} = (y_1, \dots, y_j, \dots, y_N) \in \mathbb{R}^N$ and j refers to the j^{th} index in the vector $\boldsymbol{y}$. Note that

$$\sum_{j=1}^{N} f_j(x) = 1 \tag{2.20}$$

which makes sense since the sum of all the probability is 1. For example, if the input is a vector of form $[1, -2, -1]$ and there are 3 classes in total, then the output is

$$\left[\frac{e}{e + e^{-2} + e^{-1}}, \frac{e^{-2}}{e + e^{-2} + e^{-1}}, \frac{e^{-1}}{e + e^{-2} + e^{-1}}\right], \tag{2.11}$$

which is

$$[0.844, 0.042, 0.114]\,. \tag{2.22}$$

3 Training CNN

The ambition of CNN is to classify new data into proper classes with a high score or high probability. As a result, a strong accurate model is vital, which is able to identify all the training datasets with a low error rate. Intuitively, minimizing the differences between output result and given true label would be one of the options, which consists of minimizing the mean square error (loss function) between the predicted result and the actual label. However, since the classification output is a probability result from SoftMax layer in AlexNet, the other common method is introduced instead, which is to minimize the cross-entropy equation (loss function).

In CNN, training is a process of finding the optimal parameters in the convolutional layers and the fully connected layers. Training is accomplished by applying a gradient descent method, one of the techniques to solve the minimization problem and is commonly used in neural networks [2,3,4]. In particular, backpropagation is used in the gradient descent method to find the partial derivative with respect to the learnable parameters since, generally, there are multiple layers inside a network, which means that chain rules are applied multiple times when computing the gradients (see Section 3.2).

One of the important measures of the generated model is the *loss value*, which is a measure of the distance between the predicted scores (or probability) and the truth labels. It is also a measure of the quality of the solution to the optimization problem. We want to minimize the loss as closer to zero as possible. Training is completed by multiple iterations, while the learnable parameters are updated during each iteration to minimize the loss value. Before the first iteration of the training, weight initialization is done by providing small non-zero random numbers to the weight matrix to ensure the parameters get updated correctly [10]. Each input data is passed into the network and output a loss value from the last layer, which is called forward propagation [4]. After updating the result with the correct value, backpropagation is used to find the partial derivation of each of the weight matrix. Finally, parameters are updated according to the learning rate (hyperparameter) and the derivative of its, which finishes the current iteration [10]. Then, the process enters the next iteration with the new parameter (or weight matrix) [10]. Before discussing how to implement the network, several definitions are required.

3.1 Loss function

In ML, the *loss function*, also known as *cost function*, is a function which outputs a *loss value* to measure how well the model is. Two types of loss functions are commonly adopted. One is the mean square error, which is typically employed to a continued value such as linear regression, and the other one is cross-entropy frequently addressed in multiclass classification because of its probability output [4]. In ML, we define the loss function as

$$\mathcal{L}(f, \boldsymbol{y}) \tag{3.1}$$

where f is the output vector from the last layer in the network, and $\boldsymbol{y} \in \mathbb{R}^N$, defined as follows

$$y_i = \begin{cases} 1, & \text{if the given data} \in \text{ the } i^{\text{th}} \text{ class} \\ 0, & \text{otherwise} \end{cases}, \qquad \text{for } 1 \leq i \leq \text{N}, \tag{3.2}$$

is the true label vector [5]. The *cross-entropy* of the discrete probability distribution q relative to a correct discrete probability distribution p is defined by [10]

$$H(p, q) = -\sum_x p(x) \log q(x) \tag{3.3}$$

where $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$ is an input to the probability distribution function. When applying the cross-entropy to a loss function, the *cross-entropy loss* has the form [10]

$$\mathcal{L}_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right), \qquad \text{or equivalently } \mathcal{L}_i = -f_{y_i} + \log \sum_j e^{f_j} \tag{3.4}$$

where f_j denotes the j^{th} element of the output vector f from the SoftMax layer. Note that the formula is derived from Equation 3.3 where

$$q = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}} \tag{3.5}$$

which is the predicted probability coming from the SoftMax layer, and

$$p = 1 \tag{3.6}$$

which means the correct probability for this j^{th} element is 1 [10].

Figure 3.1 describes an extremely simple example of a network with only two layers. One is the fully connected layer and the other one is the SoftMax layer. Given an input data x_i with only three values, an intermediate weight matrix *W* and a bias *b*, an output vector is produced from the layer and is set as an input to the SoftMax layer, which outputs a probability for each class in a vector form. After that, the cross-entropy loss is calculated by the natural log and based on the given true predicted label ($y_i = 1$), which is equal to 0.03. Note that the goal is to leave the loss as smaller as possible, then the probability would close to 1.

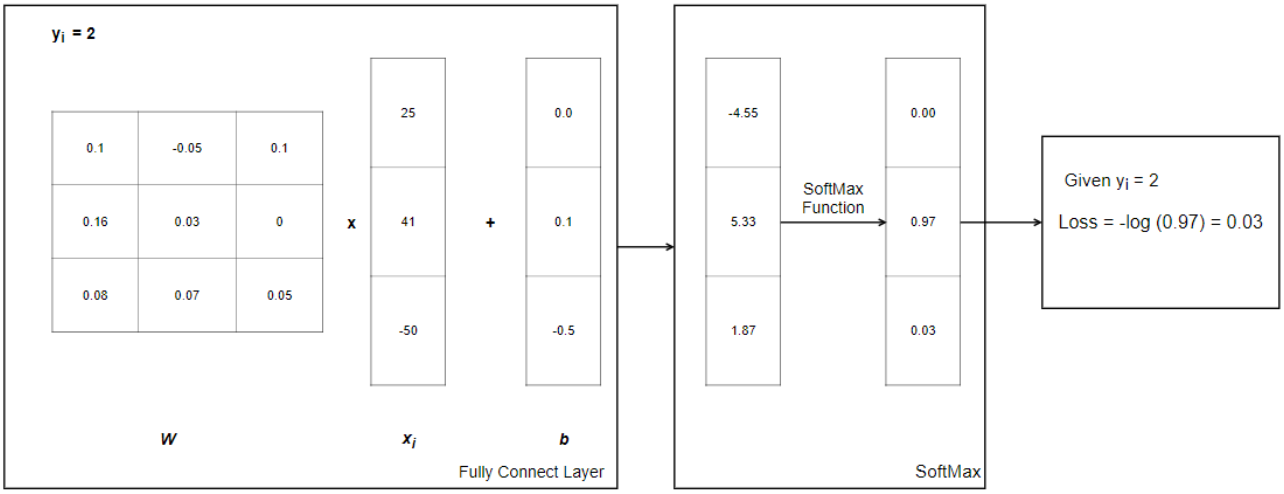

Figure 3.1 Example of loss computation: A demonstration of how the loss being computed from an input x_i of size 3×1.

In general, given a training set of n data (such as images) and m number of actual classes, then the general loss function for a multiclassification problem is [5]

$$\mathcal{L} = \sum_{i=1}^{n} \mathcal{L}_i \tag{3.7}$$

where

$$\mathcal{L}_i = -\sum_{k=1}^{m} (y_{ik} \log(f_{ik})) \,. \tag{3.8}$$

Note that $\mathcal{L}_i$ is essentially the cross-entropy loss defined before, but in a more general form, and f_{ik} is the output from the SoftMax function, and

$$y_{ik} = \begin{cases} 1, & if\ x\ \in\ the\ k^{th}\ class \\ 0, & otherwise \end{cases} . \tag{3.9}$$

In particular, when $K = 2$, we have [1, 5]

$$\mathcal{L}_i = -\sum_{k=1}^{m} (y_{ik} \log(f_{ik}) + (1 - y_{ik}) \log(1 - f_{ik})) \tag{3.10}$$

in which the conditional distribution of the target classes given input data x is a Bernoulli distribution of the form [5]

$$p(y_i \,|x_i, w) = f_i(x_i)^{y_i} \big(1 - f_i(x_i)\big)^{1-y_i} \tag{3.11}$$

where the result is 1 if $f_i \geq 0.5$ and 0 if $f_i < 0.5$. Then, the loss function is simply the negative log of the conditional distribution given above, which makes sense since we want to maximize the above conditional probability which is equivalent to minimize the negative log of the function. To minimize such general form of the loss function, we apply the optimization algorithm called Stochastic Gradient Descent (SGD) [2, 4].

3.2 Backpropagation

Backpropagation is an algorithm for efficiently finding the gradient of a function with respect to each learnable parameter especially in NN since each neural network model can be represented by computation graphs [2]. With that graph, the result can be deduced from the top layer to the bottom very quickly because of the chain rule [2].

First, we define the function

$$y = f(x) = g\,(W^L \dots g(W^2 g(W^1 x + b^1) + b^2) + \cdots + b^L) \tag{3.12}$$

where $g(x)$ is the activation function, each L refers to the layer number, W^i is the weight matrix and b^i refers to bias for $1 \leq i \leq L$ [2]. Note that the above structure is a typical form of the output from a NN. To demonstrate how the backpropagation process, we consider the composite function

$$y = f\big(g(x)\big)\,. \tag{3.13}$$

Then, if we calculate the derivative with respect to the x, by the chain rule, we have

$$\frac{\partial y}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x}\,. \tag{3.14}$$

In order to find $\frac{\partial y}{\partial x}$, we first calculate $\frac{\partial f}{\partial g}$ with the value g we know from the forward direction, Then, we obtain the result by multiplying it with $\frac{\partial g}{\partial x}$.

For the value of

$$f(x) = x^3 \tag{3.15}$$

$$g(x) = x^2 \tag{3.16}$$

$$x = 2\,, \tag{3.17}$$

we can find the derivative of the f with respect to x by applying the backpropagation algorithm (Figure 3.2).

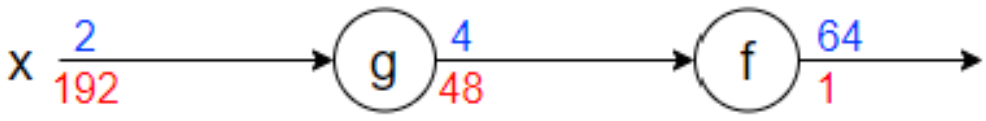

Figure 3.2 Example of backpropagation process: A example of finding $\frac{\partial y}{\partial x}(2)$ by applying the backpropagation algorithm. The blue number indicates the forward direction and the red number indicates the backward direction started from the end to the beginning.

After the corresponding values in the forward direction are successfully calculated, backpropagation is used to compute the derivative. Since $\frac{\partial f}{\partial f} = 1$, the last value is always 1. We then have

$$\frac{\partial f}{\partial g} = 3g^2, g = 4 \tag{3.18}$$

$$\frac{\partial f}{\partial g} = 48 \tag{3.19}$$

$$\frac{\partial y}{\partial x}(2) = 192 \tag{3.20}$$

Equation 3.19 is followed by substituting g back into the derivative function. Similarly, Equation 3.20 follows the same way.

Another example of the backpropagation process is demonstrated below, which is done in a NN with only four layers. There are two sets of fully connected layer and ReLU layer, one set followed the other set. Then, we can represent the network model by

$$f(x) = g(W^2 g(W^1 x + b^1) + b^2) \tag{3.21}$$

where the inner layer $g(W^1 x + b^1)$ represents the first fully connected layer with weight matrix W^1 and bias b^1, and the outer layer $g(W^2 x + b^2)$ represents the second fully connected layer with weight and bias, W^2 and b^2 respectively. Note that this is a composition function for two layers and there is no learnable parameter from the ReLU layers. Consider the case that $W^1 = 1, W^2 = -1,\ b^1 = 1, b^2 = 5$ and $x = 3$, Figure 3.3 illustrates the computation graph in a forward direction and the corresponding derivation in the backward direction. In particularly, for the ReLU function, $f(x) = \max(0, x)$, the derivative function, denoted as $F(x)$, is defined as follows

$$F(x) = \begin{cases} \frac{\partial f}{\partial x}, & if\ x \neq 0 \\ 0, & if\ x = 0 \end{cases} \tag{3.22}$$

since f is not differentiable at $x = 0$.

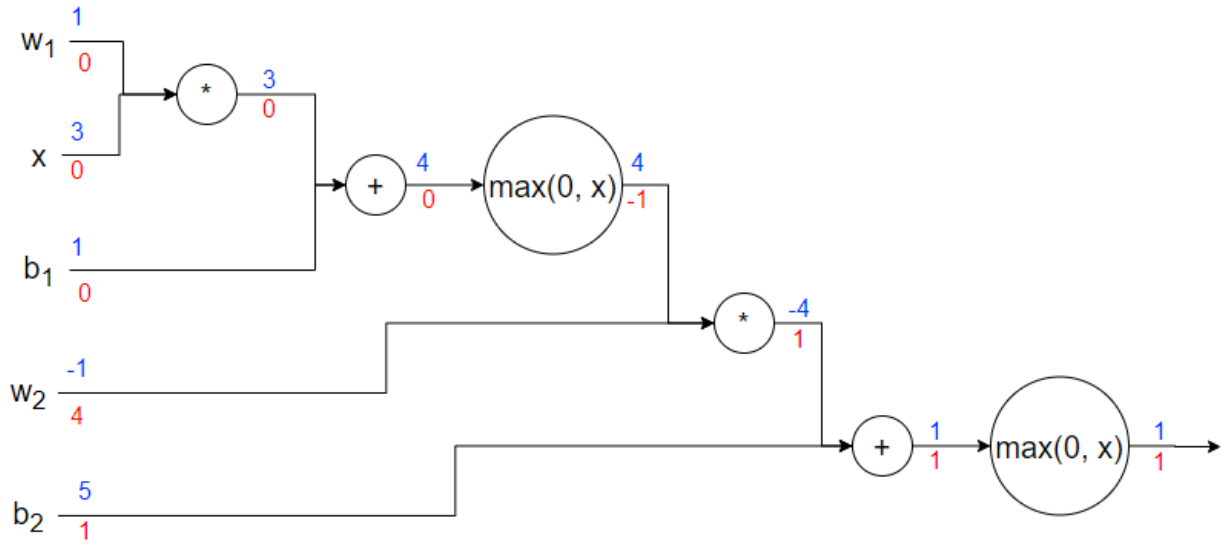

Figure 3.3 Example of backpropagation process in neural network: A acyclic computation graph of the network model $f(x) = g(W^2 g(W^1 x + b^1) + b^2)$, where g is a ReLU function ($f(x) = \max(0, x)$), $W = [1 - 1], b = [1, 5]$ and $x = 3$. The forward propagation computes values from input to output (blue) and the backpropagation is performed from the end to the beginning by applying the chain rule and the derivative rules (red).

In general, the backpropagation is done by the following pseudocode [2].

Input: A network with L layers, the activation function g_l, the output hidden layer $f_l = g_l(W f_{l-1} + b_l)$

Compute the gradient $\delta = \frac{\partial \mathcal{L}_i}{\partial y}$

For $i = L$ *to* 0 **do**

Calculate gradient for present layer

$\frac{\partial \mathcal{L}_i}{\partial W_l} = \frac{\partial \mathcal{L}_i}{\partial g_l} \frac{\partial g_l}{\partial W_l} = \delta \frac{\partial g_l}{\partial W_l}$

$\frac{\partial \mathcal{L}_i}{\partial b_l} = \frac{\partial \mathcal{L}_i}{\partial g_l} \frac{\partial g_l}{\partial b_l} = \delta \frac{\partial g_l}{\partial b_l}$

Apply gradient descent using $\frac{\partial \mathcal{L}_i}{\partial W_l}$ *and* $\frac{\partial \mathcal{L}_i}{\partial b_l}$

Back-propagate gradient to the lower layer

$\delta = \frac{\partial \mathcal{L}_i}{\partial g_l} \frac{\partial g_l}{\partial g_{l-1}} = \delta \frac{\partial g_l}{\partial g_{l-1}}$

End

3.3 Stochastic Gradient Descent (SGD)

SGD, one common optimization algorithm, is dedicated to finding the local minimum of an objective function. Compared with the standard gradient descent method, it saves a large amount of time in training without affecting the result since the training data are correlated in the same class [10]. Instead of computing the full loss function over the entire training set, SGD randomly selects a batch of training data for each target class, and updates the parameters based on the result of that batch. In particular, the size of the batch is a power of 2 such as 32, 128 or 256 because of a fast operation in many vectorized operation implementations [10].

The learnable parameter for the k^{th} iteration is updated in the following way,

$$w_{k+1} = w_k - \eta \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}_i}{\partial w_k}, \tag{3.23}$$

where N is the size of the batch, η, a hyperparameter, is the learning rate and $\mathcal{L}_i$ is the loss function. Note that $\frac{1}{N}\sum_{i=1}^{N}\frac{\partial \mathcal{L}_i}{\partial w}$ is the average derivative over a batch of size N of the loss function with respect to w, which is computed by the previous backpropagation method.

In general, the SGD is defined in the following pseudocode [2],

Input: Loss function $\mathcal{L}_i$, learning rate η, dataset X, y and the network model $f(w, x)$, where $x \in X$
Output: Optimum w which minimizes $\mathcal{L}_i$

Repeat until converge:
  Shuffle X, y;
  Create K batches as $B = [B_1, \dots, B_K] = X$
  For each batch B_j of x, y_i in X, y **do**
    $w = w - \eta \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}_i\,(f(w,x), y_i)}{\partial w}$
  End

It is noticed that for each iteration, the loss value is recorded or shown inside a graph to measure the accuracy of the model generated from the training. Since the number of iterations is huge generally, we

use *epoch*, which is a certain fixed amount of iterations, to record the overall behaviour of the loss value. Note that η, learning rate or step size, is one of the most important hyperparameters when training the network. As its name suggested, the learning rate is a control of the speed of the training, which accelerates the training process when the rate is large and make the process slower when the rate is small (Figure 3.4). Notice that if the rate is too larger, the loss would diverge instead of converging. On the contrary, if the rate is too small, although the network would eventually converge, it requires more time in training [2]. Generally, the learning rate is addressed by the step function [2]

$$\eta_t = \eta_0 \beta^{\left\lfloor \frac{t}{\epsilon} \right\rfloor} \tag{3.24}$$

where η_t is the learning rate for the t^{th} round, η_0 is the initial learning rate, ϵ is a constant and β is the decay factor. In practice, $\beta = 0.1$ and the learning rate is reduced by a factor of 10 every round [2].

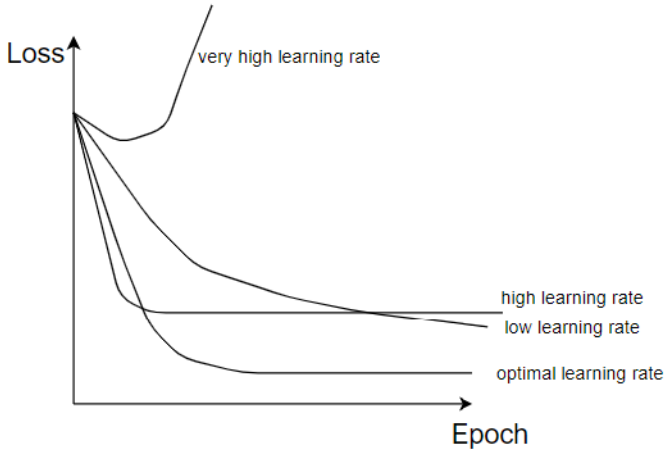

Figure 3.4 Demonstration of impact of different learning rates: A example with different learning rates with respect to the loss and epoch. Note that the optimal learning rate is the one we want to approach which is not too high and not too low.

4 Training in CIFAR-10 Dataset

In this section, a non-trivial image classification example of training a CNN is demonstrated based on the code from MathWorks [11]. The CNN example, which has a similar structure as AlexNet, trains the model with respect to 10 categories from the CIFAR-10 dataset and a result of 72% accuracy for the new data.

The whole example is fully detailed in this section. We start by introducing the training dataset, CIFAR-10 dataset. Then, thorough layer settings are analyzed before setting the hyperparameters of the training. Finally, a simple random test is performed and followed by a complete accuracy check of the test dataset.

4.1 Dataset

CIFAR-10, one of an open-source dataset, is created by Alex, Vinod and Geoffrey which contains 60000 color images of size 32×32 in 10 distinct classes: airplanes, automobiles, birds, cats, deer, dogs, frogs, horses, ships, and trucks (Figure 4.1). There are 6000 images per class including 5000 training images and 1000 test images.

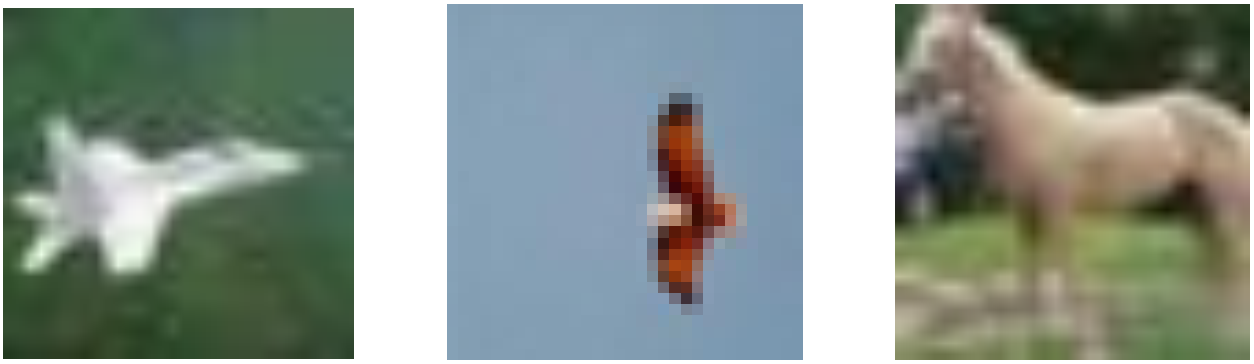

Figure 4.1 Sample images from CIFAR-10: Three sample 32×32 images from the CIFAR-10 dataset, which are plane (left), bird (middle) and horse (right).

4.2 Layers

The layer setting is presented in Figure 4.2, which consists of 15 layers as described below. The output size of each layer is shown in the activation column and the learnable column indicates the parameters which need to be trained with the corresponding size. Furthermore, the hyperparameters (stride, filter, padding) for each layer are described under the name field. Note that the average pooling layer is used twice under the example, which is simply an operation similar to the max pooling but returning the average value of the filter instead of the maximum value inside that filter. Since the network model is meant to classify ten categories, the output of the last fully connected layer is a vector of size 10 (or $1 \times 1 \times 10$). The calculation of the output size refers to Section 2.2.2.

It is worth mentioning that the first fully connected layer, Layer 11 (Figure 4.2), shows that the output is a vector of size 64, which is obtained by multiplying the weight matrix (of size 64×576) with the result converted from the previous layer. In other words, the input to the layer ($1 \times 1 \times 64$) is converted from the output (of size $3 \times 3 \times 64$) of the previous layer. Moreover, the padding for each of the convolution layer is set to preserve the output image size as the same as the input image size, but not the channels. For example, the output size of Layer 8 is still 7×7, which is the same as the input size. Finally, the ReLU layer is followed by every layer which has the learnable parameters. Note that switching the order of the max pooling layer and the ReLU layer does not affect the training. The full MATLAB code is provided in Appendix A.

ANALYSIS RESULT

	Name	Type	Activations	Learnables
1	imageinput 32x32x3 images with 'zerocenter' normalization	Image Input	32×32×3	-
2	conv_1 32 5x5x3 convolutions with stride [1 1] and padding [2 2 2 2]	Convolution	32×32×32	Weights 5×5×3×32 Bias 1×1×32
3	maxpool 3x3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	15×15×32	-
4	relu_1 ReLU	ReLU	15×15×32	-
5	conv_2 32 5x5x32 convolutions with stride [1 1] and padding [2 2 2 2]	Convolution	15×15×32	Weights 5×5×32×32 Bias 1×1×32
6	relu_2 ReLU	ReLU	15×15×32	-
7	avgpool2d_1 3x3 average pooling with stride [2 2] and padding [0 0 0 0]	Average Pooling	7×7×32	-
8	conv_3 64 5x5x32 convolutions with stride [1 1] and padding [2 2 2 2]	Convolution	7×7×64	Weights 5×5×32×64 Bias 1×1×64
9	relu_3 ReLU	ReLU	7×7×64	-
10	avgpool2d_2 3x3 average pooling with stride [2 2] and padding [0 0 0 0]	Average Pooling	3×3×64	-
11	fc_1 64 fully connected layer	Fully Connected	1×1×64	Weights 64×576 Bias 64×1
12	relu_4 ReLU	ReLU	1×1×64	-
13	fc_2 10 fully connected layer	Fully Connected	1×1×10	Weights 10×64 Bias 10×1
14	softmax softmax	Softmax	1×1×10	-
15	classoutput crossentropyex	Classification Output	-	-

Figure 4.2 Demonstration of layers: A CNN with 15 layers with the hyperparameters, learnable parameter and also the output size in activations column

4.3 Training

One can show that the output before the SoftMax layer is defined as the following,

$$f(x) = g\,(W^5 g(W^4 g(W^3 g(W^2 g(W^1 x + b^1) + b^2) + b^3) + b^4) + b^5) \tag{4.1}$$

where input x is of size $32 \times 32 \times 3$, g is the ReLU operation ($g(x) = \max(0, x)$), W^1 and b^1 refer to weight and bias of the first convolution layer in Layer 2 (Figure 4.2), while W^5 and b^5 refer to weight and bias of the last fully connected layer in Layer 13 (Figure 4.2). Notice that there are a total of 5 neuron functions corresponding to the layers which have the learnable parameters.

Then, the loss function for some input x, y is,

$$\mathcal{L}_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \tag{4.2}$$

where $y \in (0,0,\dots,1,\dots,0) \in \mathbb{R}^{10}$, y_i is the index for the value 1 in vector y, and f_{y_i} is the output of the last fully connected layer as defined previously (See Equation 2.18). Consequently, the general form of the loss function is

$$\mathcal{L} = \sum_{i=1}^{10} \mathcal{L}_i. \tag{4.3}$$

Finally, the SGD is performed to update the parameters. In particular, after the forward propagation, the scores (or probabilities) of the output are updated first in order to backpropagate to the beginning. Note that given $y_i = k$, the partial derivative of $\mathcal{L}_i$ with respect to f_k is calculated by [10]

$$\frac{\partial \mathcal{L}_i}{\partial f_k} = \frac{\partial \mathcal{L}_i}{\partial p_k}\frac{\partial p_k}{\partial f_k} = \frac{\partial}{\partial p_k}(-\log(p_k))\frac{\partial p_k}{\partial f_k}, \qquad \text{where } p_k = \frac{e^{f_k}}{\sum_j e^{f_j}} \tag{4.4}$$

$$= -\frac{1}{p_k}\frac{\partial p_k}{\partial f_k} \tag{4.5}$$

$$= -\frac{\sum_j e^{f_j}}{e^{f_k}}\frac{\partial}{\partial f_k}\left(\frac{e^{f_k}}{\sum_j e^{f_j}}\right) \tag{4.6}$$

$$= -\frac{\sum_j e^{f_j}}{e^{f_k}}\frac{e^{f_k}\sum_j e^{f_j} - e^{f_k}e^{f_k}}{\left(\sum_j e^{f_j}\right)^2} \tag{4.7}$$

$$= -\frac{\sum_j e^{f_j}}{e^{f_k}}\frac{e^{f_k}\left(\sum_j e^{f_j} - e^{f_k}\right)}{\left(\sum_j e^{f_j}\right)^2} \tag{4.8}$$

$$= -\frac{\sum_j e^{f_j} - e^{f_k}}{\sum_j e^{f_j}} \tag{4.9}$$

$$= p_k - 1\,, \tag{4.10}$$

which is a very simple term. Therefore, only the y_i^{th} element from the score vector is modified by -1 (Equation 4.10). For example, continuing from the example in Figure 3.1, given that the score vector is $[0.00\ 0.97\ 0.03]^T$ and $y_i = 2$, we have $\frac{\partial \mathcal{L}_2}{\partial f_2} = [0.00\ -0.03\ 0.03]^T$ before processing to the next step in backpropagation. Ultimately, after updating the parameters, the current iteration is completed, and the process continues to the next iteration.

The training progress is shown in Figure 4.3, which takes around 2 minutes and 21 seconds to run. Note that the process is separated into 15 epochs and each epoch requires 390 iterations. That is a total of 5850 iterations. The full logger is provided in Appendix A. The graph indicates that the data loss (in red)

approaches to 0 and the accuracy (in blue) is close to 1. It is important to notice that the accuracy line during the training process refers to the training data, while the final accuracy referred to the test dataset is 73.11%.

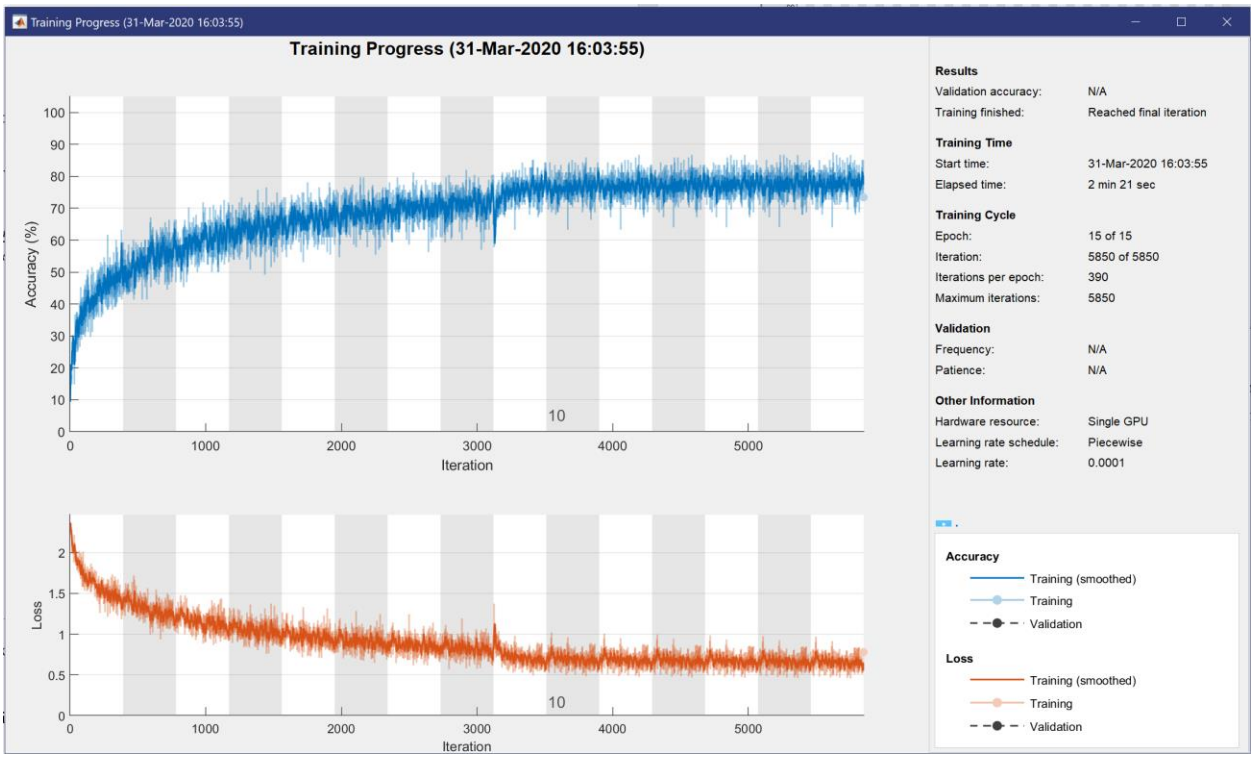

Figure 4.3 Training progress for CIFAR-10 dataset

4.4 Hyperparameter

Hyperparameters, which are the parameters that need to be manually set up before the training, play a fundamental role during the whole process. The initial learning rate (η) is set to 0.001, which is empirically chosen. If the learning rate is too high, such as 0.1, the loss value never converges to a small value, and it diverges in the beginning (Figure 4.4). The learning rate is updated following a step function (Equation 3.24), where η_0 is equal to 0.001, β is equal to 0.1, and ϵ is equal to 8. In other words, the learning rate is reduced by 0.1 every 8 epochs, where each epoch is set to 390 iterations. Furthermore, the batch size for SGD is set to 128, and the maximum number of epochs is 15.

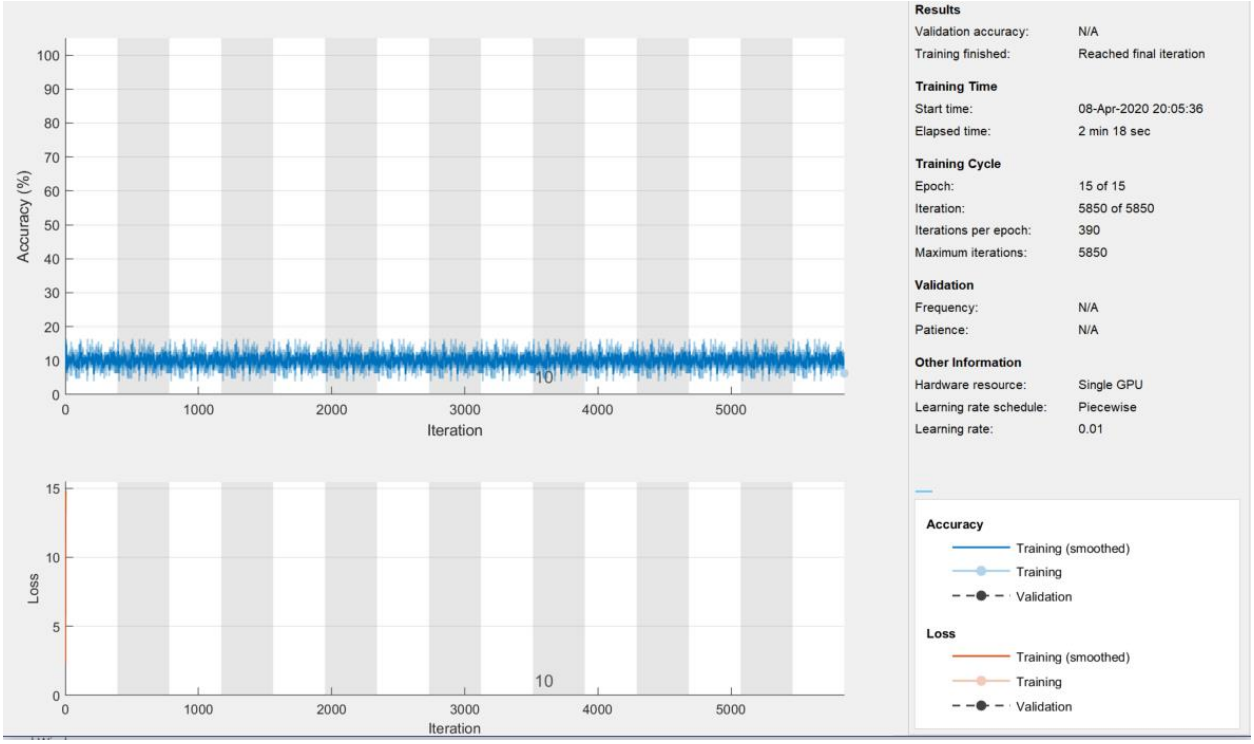

Figure 4.4 A divergence training progress

It is very critical that the initial weight for the first convolution layer is set to some random number between 0.0001 and 0.001 for each value inside the matrix, which totally alters the final accuracy result and the training time. For example, if the initial weigh is set by the default setting, the accuracy reduces to 67.39%. The same subtraction occurs to the other two fully connected layers where the weight matrix is set to some random number between 0.1 and 1.

All in all, the result of the network model is approximately 72% under the current CNN structure, while MathWorks people has shown that by using residual network (ResNet), the model can have accuracy as higher as 95% [12].

5 Conclusion

Recent CNN has gone far beyond AlexNet, which has a more complicated layer setting, such as GoogLeNet (2014), VGG (2014) and Dense CNN (2017) [2]. Notably, a technique, called transfer learning, has been developed to train on a small dataset which makes good use of the *pretrained network model* [2, 4]. *Pretrained network model* refers to the mode that has already generated from a big dataset, such as the model from AlexNet or GoogLeNet.

Convolutional neural networks (CNN) is practicable in image detection and object recognition, which has been shown by Alex, Ilya and Geoffrey in 2012 [3]. In this report, we have demonstrated the basic knowledge required to work with CNN. First, the structure of the layers is shown in a higher level with different figures that interpret its meaning. Second, we have described the mathematical view of the training in details including the loss function, backpropagation and stochastic gradient descent method. In addition, we have concluded with a small neural network example which is implemented with two layers. Lastly, a solid real-world application for a CNN training is illustrated explicitly with the explanation to each layer and the output accuracy to the network.

Appendix A

Demo Code

An example of training a CNN by using the CIFAR-10 Dataset in MATLAB

```
% Deep Learning Example: Training from scratch using CIFAR-10 Dataset
%
%  Example is originally came from the MathWorks
%  Cite as MathWorks Deep Learning Toolbox Team (2020). Deep Learning
%         Tutorial Series (https://www.mathworks.com/matlabcentral/fileexchange/
%         62990-deep-learning-tutorial-series), MATLAB Central File
%         Exchange. Retrieved March 31, 2020.


% Load training data the local directory with the following categary
categories = {'Airplane', 'Automobile', 'Bird', 'Horse', 'Ship', 'Truck','Deer','Dog','Frog','Cat'};
rootFolder = 'cifar10Train';
imds = imageDatastore(fullfile(rootFolder, categories), ...
    'LabelSource', 'foldernames');

% Conside augmenting the image dataset if necessary (Optional)
imageSize = [32 32 3];
pixelRange = [-4 4];
imageAugmenter = imageDataAugmenter( ...
    'RandXReflection',true, ...
    'RandXTranslation',pixelRange, ...
    'RandYTranslation',pixelRange);
augimdsTrain = augmentedImageDatastore(imageSize,
imds,''DataAugmentation'',imageAugmenter,'OutputSizeMode','randcrop');

% Define the layers as the following

% convolution network with filter size be 5 x 5 x 32 and padding is 2
conv1 = convolution2dLayer(5,32,'Padding',2,'BiasLearnRateFactor',2);

% The initial weight from the orignial network save a lot of time in
% training
conv1.Weights = gpuArray(single(randn([5 5 3 32])*0.0001));

% Setting the initial weight for 2 fully connected layers
fc1 = fullyConnectedLayer(64,'BiasLearnRateFactor',2);
fc1.Weights = gpuArray(single(randn([64 576])*0.1));
fc2 = fullyConnectedLayer(10,'BiasLearnRateFactor',2);
fc2.Weights = gpuArray(single(randn([10 64])*0.1));

layers = [
    imageInputLayer([32 32 3]);
    conv1;
```

```
    %max pooling layer with filter size be 3x3 and stride be 2
    maxPooling2dLayer(3,'Stride',2);

    %ReLU layer (f(x) = max(0,x)
    reluLayer();

    % convolution network with filter size be 5 x 5 x 32 and padding is 2
    convolution2dLayer(5,32,'Padding',2);

    %ReLU layer (f(x) = max(0,x)
    reluLayer();

    %max pooling layer with filter size be 3x3 and stride be 2
    maxPooling2dLayer(3,'Stride',2);

    % convolution network with filter size be 5 x 5 x 64 and padding is 2
    convolution2dLayer(5,64,'Padding',2);

    %ReLU layer (f(x) = max(0,x)
    reluLayer();

    %max pooling layer with filter size be 3x3 and stride be 2
    maxPooling2dLayer(3,'Stride',2);

    % fully connected layer with output be a vector of size 64x1
    fc1;

    %ReLU layer (f(x) = max(0,x)
    reluLayer();

    % fully connected layer with output be a vector of size 10x1
    fc2;

    softmaxLayer()
    classificationLayer()];

% Define training options

% Using SGD method
opts = trainingOptions('sgdm', ...
    'InitialLearnRate', 0.001, ... % Define initial learning rate
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropFactor', 0.1, ... % Define the rate drop factor
    'LearnRateDropPeriod', 8, ...   % Define the drop period
    'L2Regularization', 0.004, ...
    'MaxEpochs', 15, ...            % Define maximum epochs, each epoch is 390 iteration
    'MiniBatchSize', 128, ...       % Define the batch size
    'Verbose', true ,...            % Set the output to verbose
    'Plots','training-progress');   % Plot the progress when training
```

```
% Training

[net, info] = trainNetwork(imds, layers, opts);

% Load test data

rootFolder = 'cifar10Test';
imds_test = imageDatastore(fullfile(rootFolder, categories), ...
    'LabelSource', 'foldernames');

% Test one at a time
% If the title of the image is green, this is a correct
% prediction. If the title is red, the prediction is incorrect.

labels = classify(net, imds_test);

ii = randi(4000);
im = imread(imds_test.Files{ii});
imshow(im);
if labels(ii) == imds_test.Labels(ii)
   colorText = 'g';
else
    colorText = 'r';
end
title(char(labels(ii)),'Color',colorText);

% Calculate the accuracy of the model by comparing the predicted label with
% the actual label for all the testing data

% This could take a while if you are not using a GPU
confMat = confusionmat(imds_test.Labels, labels);
confMat = confMat./sum(confMat,2);
mean(diag(confMat))
```

Training Log

```
Training on single GPU.
Initializing input data normalization.
|=========================================================================|
|  Epoch  |  Iteration  |  Time Elapsed  |  Mini-batch  |  Mini-batch  |  Base Learning  |
|         |             |   (hh:mm:ss)   |   Accuracy   |     Loss     |      Rate       |
|=========================================================================|
|       1 |           1 |       00:00:00 |       9.38% |       2.3041 |          0.0010 |
|       1 |          50 |       00:00:01 |      31.25% |       1.8950 |          0.0010 |
|       1 |         100 |       00:00:03 |      32.81% |       1.7555 |          0.0010 |
|       1 |         150 |       00:00:04 |      40.63% |       1.7073 |          0.0010 |
|       1 |         200 |       00:00:05 |      46.88% |       1.5022 |          0.0010 |
|       1 |         250 |       00:00:06 |      52.34% |       1.4170 |          0.0010 |
```

1	300	00:00:07	52.34%	1.4381	0.0010
1	350	00:00:09	56.25%	1.2239	0.0010
2	400	00:00:10	48.44%	1.4591	0.0010
2	450	00:00:11	49.22%	1.3739	0.0010
2	500	00:00:12	48.44%	1.4089	0.0010
2	550	00:00:13	55.47%	1.3429	0.0010
2	600	00:00:14	61.72%	1.1354	0.0010
2	650	00:00:16	58.59%	1.2735	0.0010
2	700	00:00:17	46.09%	1.2796	0.0010
2	750	00:00:18	60.94%	1.1779	0.0010
3	800	00:00:19	51.56%	1.3215	0.0010
3	850	00:00:20	60.16%	1.1186	0.0010
3	900	00:00:22	65.63%	0.9637	0.0010
3	950	00:00:23	64.06%	1.0780	0.0010
3	1000	00:00:24	64.84%	0.9055	0.0010
3	1050	00:00:25	65.63%	0.9865	0.0010
3	1100	00:00:26	60.94%	1.1539	0.0010
3	1150	00:00:27	60.16%	1.0755	0.0010
4	1200	00:00:29	64.84%	1.1390	0.0010
4	1250	00:00:30	53.91%	1.2925	0.0010
4	1300	00:00:31	67.19%	0.9870	0.0010
4	1350	00:00:32	66.41%	0.9462	0.0010
4	1400	00:00:33	67.19%	1.0291	0.0010
4	1450	00:00:35	64.06%	0.9878	0.0010
4	1500	00:00:36	61.72%	1.1300	0.0010
4	1550	00:00:37	71.09%	0.8468	0.0010
5	1600	00:00:38	60.94%	1.0958	0.0010
5	1650	00:00:39	67.19%	0.9362	0.0010
5	1700	00:00:41	67.97%	0.8921	0.0010
5	1750	00:00:42	60.94%	1.1187	0.0010
5	1800	00:00:43	67.97%	0.8929	0.0010
5	1850	00:00:44	71.88%	0.8670	0.0010
5	1900	00:00:45	74.22%	0.8198	0.0010
5	1950	00:00:47	58.59%	1.0432	0.0010
6	2000	00:00:48	66.41%	0.9486	0.0010
6	2050	00:00:49	74.22%	0.8797	0.0010
6	2100	00:00:50	70.31%	0.8271	0.0010
6	2150	00:00:51	63.28%	1.0102	0.0010
6	2200	00:00:53	67.97%	0.8891	0.0010
6	2250	00:00:54	69.53%	0.8060	0.0010
6	2300	00:00:55	71.09%	0.7400	0.0010
7	2350	00:00:56	71.09%	0.8306	0.0010
7	2400	00:00:58	70.31%	0.8431	0.0010
7	2450	00:00:59	74.22%	0.8123	0.0010
7	2500	00:01:00	70.31%	0.8546	0.0010
7	2550	00:01:01	69.53%	0.8081	0.0010
7	2600	00:01:02	73.44%	0.8862	0.0010
7	2650	00:01:04	70.31%	0.8763	0.0010
7	2700	00:01:05	67.19%	0.8693	0.0010
8	2750	00:01:06	69.53%	0.9257	0.0010
8	2800	00:01:07	75.00%	0.7387	0.0010

8	2850	00:01:08	77.34%	0.6422	0.0010
8	2900	00:01:10	68.75%	0.8543	0.0010
8	2950	00:01:11	73.44%	0.7827	0.0010
8	3000	00:01:12	76.56%	0.5753	0.0010
8	3050	00:01:13	74.22%	0.8650	0.0010
8	3100	00:01:14	74.22%	0.7660	0.0010
9	3150	00:01:16	67.19%	0.7791	0.0001
9	3200	00:01:17	69.53%	0.8847	0.0001
9	3250	00:01:18	75.00%	0.7158	0.0001
9	3300	00:01:19	79.69%	0.7020	0.0001
9	3350	00:01:21	78.91%	0.7504	0.0001
9	3400	00:01:22	75.78%	0.7063	0.0001
9	3450	00:01:23	67.19%	0.8060	0.0001
9	3500	00:01:24	84.38%	0.5283	0.0001
10	3550	00:01:25	69.53%	0.8234	0.0001
10	3600	00:01:27	78.91%	0.6741	0.0001
10	3650	00:01:28	80.47%	0.5917	0.0001
10	3700	00:01:29	71.88%	0.8218	0.0001
10	3750	00:01:30	79.69%	0.6604	0.0001
10	3800	00:01:31	84.38%	0.6023	0.0001
10	3850	00:01:33	79.69%	0.5431	0.0001
10	3900	00:01:34	72.66%	0.7882	0.0001
11	3950	00:01:35	78.13%	0.6981	0.0001
11	4000	00:01:36	77.34%	0.6937	0.0001
11	4050	00:01:37	76.56%	0.6019	0.0001
11	4100	00:01:39	75.78%	0.7333	0.0001
11	4150	00:01:40	75.78%	0.6284	0.0001
11	4200	00:01:41	81.25%	0.5588	0.0001
11	4250	00:01:42	80.47%	0.5406	0.0001
12	4300	00:01:43	80.47%	0.6346	0.0001
12	4350	00:01:45	72.66%	0.6203	0.0001
12	4400	00:01:46	75.78%	0.6715	0.0001
12	4450	00:01:47	78.13%	0.6456	0.0001
12	4500	00:01:48	82.03%	0.5559	0.0001
12	4550	00:01:49	75.00%	0.6634	0.0001
12	4600	00:01:51	78.91%	0.6633	0.0001
12	4650	00:01:52	79.69%	0.6545	0.0001
13	4700	00:01:53	76.56%	0.8147	0.0001
13	4750	00:01:54	75.78%	0.7036	0.0001
13	4800	00:01:55	82.03%	0.5454	0.0001
13	4850	00:01:57	73.44%	0.7132	0.0001
13	4900	00:01:58	75.00%	0.6296	0.0001
13	4950	00:01:59	85.16%	0.4730	0.0001
13	5000	00:02:00	75.00%	0.7270	0.0001
13	5050	00:02:01	76.56%	0.6324	0.0001
14	5100	00:02:03	75.00%	0.6144	0.0001
14	5150	00:02:04	73.44%	0.7643	0.0001
14	5200	00:02:05	76.56%	0.6827	0.0001
14	5250	00:02:06	81.25%	0.6610	0.0001
14	5300	00:02:07	81.25%	0.6775	0.0001
14	5350	00:02:09	78.91%	0.6632	0.0001

14	5400	00:02:10	67.97%	0.7963	0.0001
14	5450	00:02:11	85.16%	0.5152	0.0001
15	5500	00:02:12	73.44%	0.7608	0.0001
15	5550	00:02:13	77.34%	0.6421	0.0001
15	5600	00:02:15	85.16%	0.5467	0.0001
15	5650	00:02:16	72.66%	0.7798	0.0001
15	5700	00:02:17	79.69%	0.6229	0.0001
15	5750	00:02:18	83.59%	0.5751	0.0001
15	5800	00:02:19	83.59%	0.5293	0.0001
15	5850	00:02:21	73.44%	0.7792	0.0001
==========

Appendix B

Glossary of Terms and Definitions

Bayes' Rule
It describes the probability of an event given that another event has happened which is defined as

$$P(A \mid B) = \frac{P(B \mid A)\, P(A)}{P(B)}.$$

Channel
Channel refers to a certain component of an image. For example, an RGB image has three channels which are red, green, and blue.
Convolution
Convolution refers to convolution product in mathematics or the convolutional layer in CNN.
Cross-entropy
The cross-entropy of the distribution q relative to a correct distribution p is defined as

$$H(p, q) = -\sum_{x} p(x) \log\big(q(x)\big).$$

Depth
Depth is the same as the channel described above but in a more general form. We can say that the depth of an RPG image is 3. In this report, we use depth to denote the number of filters (or the number of output feature maps) and channel to denote the number of the input feature maps.
Filter (or Kernel or Receptive field)
Filter or Kernel is a three-dimensional array of size ($K \times K \times F$), which refers to the weight matrix in the convolutional layer.
Learnable Parameter
It refers to the weight matrix and the bias in the convolutional layer and the fully connected layer, which means that they are being updated after each iteration.
Likelihood
It is the probability of data given the model

$$P(data \mid model).$$

Loss (or Cost)
It is a measure of the distance between the predicted scores (or probability) and the truth labels. The value is better if close to 0.
Neuron
A neuron is defined as $f(x) = wx + b$, where $x \in \mathbb{R}^n$ is the input data, $w \in \mathbb{R}^{m \times n}$ is a weight matrix, and $b \in \mathbb{R}^n$ is a bias.
Model
Model is a mathematical artifact which is generated from the training process.
Pooling
Pooling refers to the pooling layer in CNN, whose main purpose is to reduce the dimension.
Posterior
It is the probability of a hypothesis given the observed evident

$$\mathrm{P}(\text{model} \mid \text{data}) .$$

Pretrained network model

It refers to the mode that has already generated from a big dataset, such as the model from AlexNet or GoogLeNet.

Prior

It is our belief of the model before investigating the data.

ReLU (Rectified linear unit)

It refers to the ReLU layer in CNN, which is defined as

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

Sigmoid Function

The sigmoid function is defined as

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

SoftMax

SoftMax refers to the SoftMax layer in CNN. In mathematics, the SoftMax function is defined as

$$f(x) = \frac{e^{x}}{\sum_{j} e^{j}} .$$

Training model

Informally, it is an operation that provides the ML algorithm with training data to learn from. Formally, training is a loop procedure that repeats until all the weight and all the bias are determined.

References

[1] A. Hertzmann, Lecture Notes, Topic: "Machine Learning and Data Mining Lecture Notes" CSC411/D11, Computer Science Department, University of Toronto, Toronto, February 6, 2012.

[2] M. Z. Alom, T. M. Taha, C. Yakopcic, S. Westberg, P. Sidike, M. S. Nasrin, B. C. V. Essen, A. A. S. Awwal, and V. K. Asari, "The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches," arXiv.org, 12-Sep-2018. [Online]. Available: https://arxiv.org/abs/1803.01164. [Accessed: 24-Mar-2020].

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural ...," 2012. [Online]. Available: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf. [Accessed: 24-Mar-2020].

[4] Yamashita, R., Nishio, M., Do, R.K.G. et al. Convolutional neural networks: an overview and application in radiology. Insights Imaging 9, 611–629 (2018). https://doi.org/10.1007/s13244-018-0639-9

[5] F. Farhadi, "Learning Activation Functions in Deep Neural Networks," PolyPublie, 01-Dec-2017. [Online]. Available: https://publications.polymtl.ca/2945/. [Accessed: 24-Mar-2020].

[6] T. Epelbaum, "Deep learning: Technical introduction," arXiv.org, 11-Sep-2017. [Online]. Available: https://arxiv.org/abs/1709.01412. [Accessed: 25-Mar-2020].

[7] R. Fisher, S. Perkins, A. Walker, and E. Wolfart, "Convolution," Glossary - Convolution, 2003. [Online]. Available: https://homepages.inf.ed.ac.uk/rbf/HIPR2/convolve.htm. [Accessed: 25-Mar-2020].

[8] Fukushima, Kunihiko. "Neocognitron: A hierarchical neural network capable of visual pattern recognition." Neural networks 1.2 (1988): 119-130.

[9] S. Mallick and S. Nayak, "Number of Parameters and Tensor Sizes in a Convolutional Neural Network (CNN)," Learn OpenCV, 22-May-2018. [Online]. Available: https://www.learnopencv.com/number-of-parameters-and-tensor-sizes-in-convolutional-neural-network/. [Accessed: 27-Mar-2020].

[10] A. Karpathy, Lecture Notes, Topic: "Convolutional Neural Networks for Visual Recognition" CS231, Computer Science Department, University of Toronto, Toronto, January, 2019.

[11] MathWorks Deep Learning Toolbox Team (2020). Deep Learning Tutorial Series (https://www.mathworks.com/matlabcentral/fileexchange/62990-deep-learning-tutorial-series), MATLAB Central File Exchange. Retrieved March 31, 2020.

[12] The MathWorks, Inc., "Train Residual Network for Image Classification," MATLAB & Simulink. [Online]. Available: https://www.mathworks.com/help/deeplearning/ug/train-residual-network-for-image-classification.html. [Accessed: 31-Mar-2020].