TRƯỜNG ĐẠI HỌC FPT

Residual Networks with Application to Image Colorization

Graduation Thesis Final Report

Vuong Tuan Quang
Do Trung Nghia

A thesis submitted in part fulfillment of
the degree of BSc. in Computer Science with
the supervision and moderation of **Dr. Vu Khac Ky**.

Bachelor of Computer Science
FPT University – Hoa Lac campus
December 23, 2020

Acknowledgments

Alongside our supervisor - **Dr. Vu Khac Ky** – Lecturer at Department of Mathematics, FPT University, we would love to send our warmest appreciation to **Dr. Hoang Manh Tuan** – Lecturer at Department of Mathematics, FPT University; Researcher at Institute of Information Technology, Vietnam Academy of Science and Technology (VAST) for having assisted us throughout the time!

Next, we would love to express our best regards to **Ms. Luong Thuy Chung** – Student of B.Sc. in Computer Science, FPT University, and **Ms. Do Thuy Trang** – Student of B.Sc. in Computer Science, FPT University; Artificial Intelligence Engineer at FPT Software for having transferred us incredibly valuable knowledge in our field of study.

Last but not least, we would love to send the sincere thanks to **Mr. Nguyen Dang Kien** – Student of B.Sc. in Information Assurance, FPT University; Penetration Tester at CyRadar Information Assurance Company, and **Mr. Le Tung Giang** – Student of B.Sc in Electronics and Telecommunication, Hanoi University of Science and Technology for having lent us the Microsoft Azure Virtual Machine, which is significantly vital for training the model.

Abstract

In these recent years, we have been witnessing the dramatic evolution of Artificial Intelligence, Machine Learning, and especially Deep Learning algorithms that revolutionize the definition of what a computer can do. Among the tremendous number of applications of Deep Neural Networks, a notorious breakthrough in Image Classification using Convolutional Neural Networks is undeniably **Residual Networks (ResNets)**. In the scope of this thesis, we will elucidate the architecture of Residual Networks by reviewing the very basic preliminaries and put those pieces together to reconstruct a simple ResNet from scratch. Furthermore, we will also apply the pre-trained **InceptionResNetV2** developed by Keras to the application of colorizing black-and-white images that is believed to offload the gigantic amount of work by some simple clicks.

Thesis Specification

Thesis Information

Thesis name: Residual Networks with Application to Image Colorization.
Thesis group: CST491_G1.
Timeline: From September 14, 2020 to December 23, 2020.

Personnel

Supervisor

Title	Full name	Phone Number	E-mail
Ph.D	Vu Khac Ky	+84 (0) 969-162-536	kyvk2@fe.edu.vn

Team members

Role	Full name	Roll No.	Phone Number	E-mail
Leader	Vuong Tuan Quang	HE130182	+84 (0) 869-291-124	quangvthe130182@fpt.edu.vn
Member	Do Trung Nghia	HE130296	+84 (0) 353-745-105	nghiadthe130296@fpt.edu.vn

Responsibilities

No.	Task	Executed by
1	Researching Residual Networks and its related applications	Vu Khac Ky Vuong Tuan Quang Do Trung Nghia
2	Writing chapter 1: Introduction	Vuong Tuan Quang
3	Writing chapter 2: Preliminaries	Do Trung Nghia
4	Writing chapter 3: Residual Networks (Theory)	Vuong Tuan Quang
5	Writing chapter 3: Residual Networks (Experiments)	Do Trung Nghia
6	Coding & reporting the result of MNIST Digits Classifier using ResNet34, ConvNet34	
7	Writing chapter 4: Application to Image Colorization	Vuong Tuan Quang
8	Coding & reporting the result of Image Colorization using Autoencoder and Inception-ResNet-v2	
9	Coding & reporting the result of Image Colorization using basic CNN	
10	Writing chapter 5: Conclusion & Future Work	Vuong Tuan Quang
11	Researching further applications	Vu Khac Ky Vuong Tuan Quang
12	Reviewing thesis book & presentation slide	Vu Khac Ky Vuong Tuan Quang Do Trung Nghia

The person who is responsible for executing a specific section or chapter is also responsible for reporting that section in the thesis book and the presentation slide for the thesis defense session.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Deep neural network was a revolutionary advance in image classifying problem. Deep networks fundamentally comprise of low/mid/high-level features that supply an adequate amount of information to the classifiers. Moreover, the number of stacked layers, or preferably, the depth of the network decides the "level" of features. The decisive role which network depth plays had raised an intrigued subject: *"Can networks be easily improved by stacking more layers?"* Conducted experiments in real-world models showed that a degradation problem has occurred when the network gets much deeper. In 2016, He, Zhang, Ren, and Sun [7] first proposed **Residual Networks (ResNets)** with shortcut connections that revolutionize the Deep Learning industry. Although this innovation came from a simple intuition of approximating the residual to 0 instead of approximating the identity mapping to the desired output, ResNets has proven its outstanding efficiency on image classification.

In the scope of this thesis, we will review related background knowledge and then use those to construct a simple residual network that solves the digit classification problem on MNIST dataset.

Making use of *Inception-ResNet-v2* [19] that have been trained on ImageNet dataset [10] of 1.2 million images, we implemented an image colorization model on Places365-Standard dataset [11] that could be used as a core platform for more complex tasks such as video colorization.

1.2 Thesis Outline

Our thesis contains 5 main chapters:

- Introduction
- Preliminaries
- Residual Networks
- Application to Image Colorization
- Conclusion & Future Work

Sequentially, we will walk through all the 5 chapters from the very basic ground blocks. Henceforth, building those blocks up, we hope that everybody, regardless of their majors, is able to truly understand what we are working on and which method we are using.

Preliminaries covers the background knowledge related to the topic: from the elementary neurons of a simple neural network to a deep convolutional neural network with multiple components; how a neural network is trained forwardly and backwardly.

Residual Networks is the principal chapter of our thesis. All of the other chapters revolve around Residual Networks. In this chapter, we will elucidate the architecture of a residual network from the simplest building blocks with shortcut connections to the more complex **ResNet34** model. We will also re-implement a residual network for a simple image classification problem. Our example illustrates how much a tiny primitive thought is able to help solve a giant modern problem.

Application to Image Colorization is believed to help reduce the amount of work in recovering and colorizing black-and-white images down to a few seconds by rendering colorful images with the help of **Inception-ResNet-v2** [19]. We use a simple model with encoder layers to represent a gray-scale image in latent vector space, then concatenate it with high-level features extracted from the Inception-ResNet-v2 [19], and finally decode a colorized image. Alongside our implementations of image classification and image colorization, we also monitor a traditional CNN version for the same problem to validate whether ResNet performs better on the same task.

Conclusion & Future Work does not only summarize what we have done but also mentions our future work with the video colorization problem, which is motivated by the extensive researches of image colorization. This expensive problem has largely been left behind, we hope our work in the future will make a considerable contribution to the community.

1.3 Related Works

In terms of **Image Colorization**, there are many approaches to solve this problem such as Digital Image Processing algorithms, Computer Vision techniques, Deep Learning models. We sampled a survey of image colorization using Deep Learning models from the simplest Convolutional Neural Networks (Larsson, Maire, and Shakhnarovich [12]) to the more complicated Generative Adversarial Networks such as Zhang et al. [37]; Iizuka, Simo-Serra, and Ishikawa [9]; Ozbulak [16], Lee et al. [13]. There is also an approach using the probability distribution of color proposed by Zhang, Isola, and Efros [36] which has a promising result. Finally, we decided to follow the Deep Koalarization [2], which made use of an inception Residual Network and autoencoder as our pinnacle model for the thesis.

Chapter 2

Preliminaries

2.1 Neural Networks

Artificial Intelligence (**AI**) has become more and more common thanks to increased data volumes, advanced algorithms, and improvements in hardware. The term is self-explanatory- the intelligence demonstrated by men-made products, specifically machines. There are several examples and applications of AI in use today: virtual assistants, suggestive searches, autonomously- powered self-driving vehicles. This branch of **Computer Science** has contributed a lot to the technological advance we possess today. Dive in deeper, there is **Machine Learning** (**ML**)- the study of computer algorithms that improve automatically through experience [32]. Given suitable datasets, ML algorithms (**models**) are capable of self-learning without being explicitly programmed to do so.
ML is divided into *three* kinds of learning: **supervised learning**, **unsupervised learning**. *Supervised learning* models are fit on training data comprised of both inputs and outputs, then generate predicted outputs from test sets where only the inputs are provided. The predictions are compared to the truths to see the models' performance and how to improve. *Unsupervised learning* looks for hidden patterns in the data set without knowing the outputs as well as being overseen.

Artificial neural networks (ANNs), usually called ***neural networks*** (NNs), are computing systems built to mimic biological neural networks. Neural networks can process information to gain knowledge. Based on their obtained knowledge, the neural networks' model can make inferences. Particularly, let's take the image classification problem as an example. Amongst thousands of dogs and cats' images, how will the computer distinguish the cat photos from the dog photos? By seeing the images as pixels of different values, neural networks provide us a mathematical function that transforms the pixels of a cat image to the conclusion that is an image of a cat. Of course, ourselves or the computer don't know how this function looks, so we can only make guesses until the function can efficiently perform the task.

By feeding the models images of cats and dogs along with teaching them what image is of a cat (or dog), the neural networks can give better function approximation. In the beginning, we will set all of the model's function coefficients to some default values. We test that function by inserting pictures of cats and dogs without telling the model which one is which (assuming we already know the answer) into the function, so the outputs of the function are predictions of the model deciding the animals' existence in the corresponding image 2.1. Then, the predictions will be compared to the desired outputs mathematically by calculating the numerical error between them. Ultimately, the models' primary goal is to minimize the overall error by improving the function. In ML, the overall error is the **cost function**. We repeat the procedure of trials and errors until the function performance satisfies our needs. The whole process is the **training** stage when building a model. After the training stage is finished, the model should be able to tell whether the image is of a dog or a cat.

An ANN is based on a collection of connected units or nodes called artificial ***neuron***s, which loosely resembles the neurons in a biological brain. Each connection, like the synapses in a biological brain, can transmit a *signal* to other neurons. An artificial neuron that receives a signal then processes it and can signal neurons connected to it. The "*signal*" at a connection is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs. The connections are called ***edge***s. *Neurons & edges* typically have a ***weight*** that adjusts as learning proceeds. The weight increases or decreases the strength of the signal at a connection. Neurons may have a threshold such that a signal is sent only if the aggregate signal crosses that threshold. Typically, neurons are aggregated

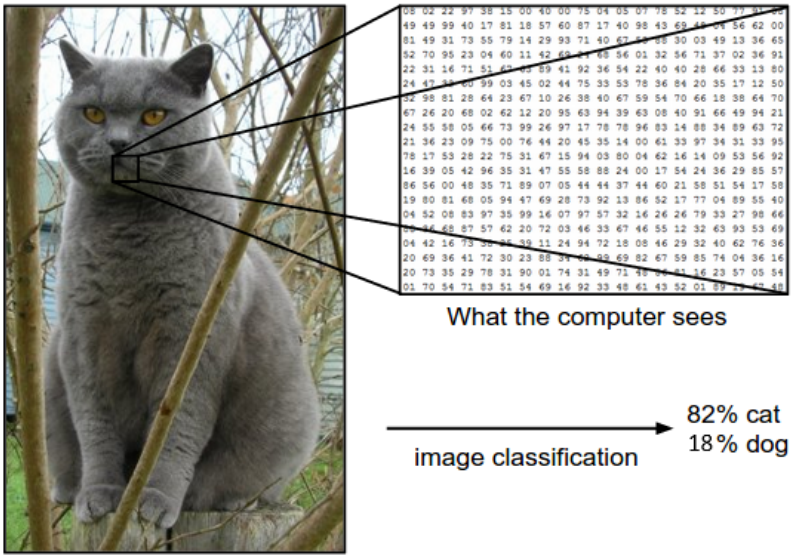


Figure 2.1: Animals pictures classification problem visualization
[5] "Image Classification," class notes for CS231n: Convolutional Neural Networks for Visual Recognition, Dept. of Comp. Sci., Stanford University, Palo Alto, CA, USA, spring 2017. [Online]. Available: https://cs231n.github.io/classification/

into ***layer***s. Different layers may perform different transformations on their inputs. Signals travel from the first layer (the *input* layer) to the last layer (the *output* layer), possibly after traversing the layers multiple times [26]. Dive in deeper, we will introduce the simplest neural network model which is the **feedforward neural network**.

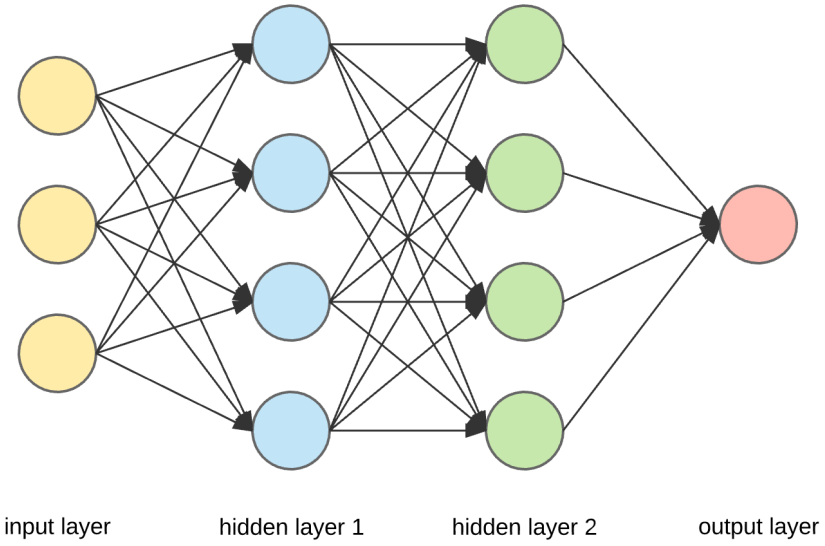


Figure 2.2: Visualization of a basic neural network

2.1.1 Feedforward Neural Networks

Feedforward neural networks, or ***multilayer perceptrons*** (MLPs), are basically artificial neural networks where connections between nodes do *not* form a cycle. The feedforward neural network was the first and simplest type of artificial neural network conducted. In this network, the information moves in only one direction—forward—from the input nodes, through the hidden nodes to the output nodes. There are no cycles or loops in the network [30]. Feedforward network is created to approximate some function f^*. Let us take the classifier as an example, $y = f^*(x)$ use the input $\mathbf{x}$ to a category y. The network defines a mapping $\mathbf{y} = f(\mathbf{x}; \theta)$ and learns the value of the parameters θ which generate the best function approximation [6].

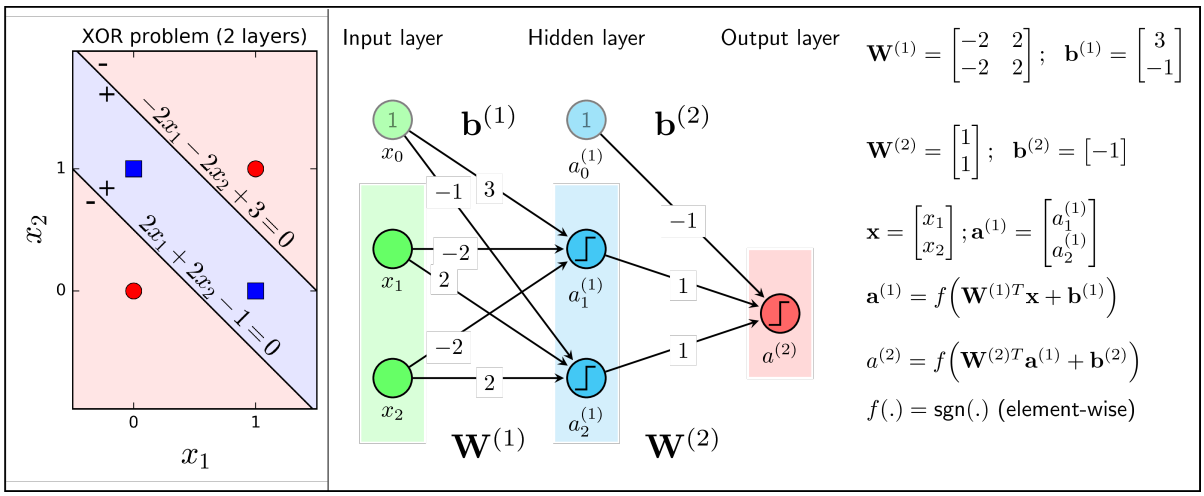


Figure 2.3: XOR function is represented by multi-layer Perceptron
Multi-layer Perceptron and Backpropagation [20]

The word *"network"* is inspired by the fact that the algorithm is typically represented by stacking many different functions together. The first layer is called the input layer, and the last one is called the output layer. For example, let there be 3 functions $f^{(1)}, f^{(2)}, f^{(3)}$ are wrapped inside each other to produce $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$. Each function from the example above represents a layer of the network. The length of the chain or the number of layers stacked together gives the *depth* of the model, when the number of layers get too **large**, ***Deep Learning*** models arise.

Training a model is the process of matching the output with the truth. The training materials forcefully command the ***output*** layer at each point x to produce a value that is close to the correct answer. Other layers' assignment is not well stated but the learning algorithm must decide how to wield those layers to produce desired outputs. Thus, training data does not guide the middle layers so we called them ***hidden layers***. Hidden layers' dimension decides the ***width*** of the model. To sum up, the layer consists of many units working in parallel to demonstrate a **vector-to-scalar** function. Each unit (or *neuron*) receives input from many other units and produces an *activation* value [6].

Perceptron Learning Algorithm (PLA)

In machine learning, the **perceptron** is a supervised learning algorithm. It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector.

Let there be two-labeled classes, finding a **linear boundary** to separate two classes from another while assuming that the boundary exists which also means two classes are *linearly separable*. This is where ***Linear Classifier*** Algorithm comes in handy.

The solution was to first randomly pick a *linear boundary*, represented in some kind of linear function. For example, the function can be:

$$f_{\mathbf{w}}(\mathbf{x}) = w_1 x_1 + \cdots + w_d x_d + w_0 x_0 = \mathbf{x}^T \mathbf{w} = 0 (x_0 = 1)$$

Where the $\mathbf{x}$ is the input, $\mathbf{w}$ represents the parameter. The mission was to figure out the best possible $\mathbf{w}$ for the linear boundary to confine the class in its space. As a linear classifier, *PLA* is the first *feedforward neural network*.

Learning XOR function

Because our data are mostly linearly inseparable, *one* perceptron can not represent the XOR function. Alternatives, e.g. *logistic regression* or *softmax regression* are proposed but essentially, those algorithms only create *linear boundaries*. Fortunately, combining perceptrons seems to work.

In Fig. 2.3, all points lie in the $(+)$ side of $-2x_1 - 2x_2 + 3 = 0$ or *line 1* to be short (we call the other line: *line 2*) produce outputs of $a_1^{(1)} = 1$, while the $(-)$ side gives us -1. The same goes for

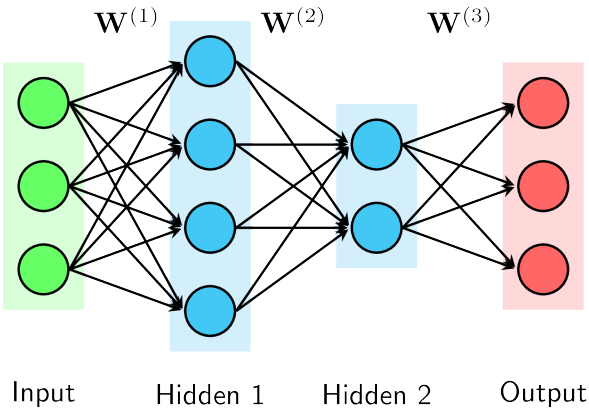


Figure 2.4: MLP with two hidden layers (hidden bias)
Multi-layer Perceptron and Backpropagation [20]

line 2. Needless to say, two lines produce *outputs* at nodes $a_1^{(1)}, a_2^{(1)}$. Now feed the outputs of those two nodes to another PLA as inputs to gain the final output or **prediction**. The predicted outputs match the ground truth completely. By using three PLAs at the same time $a_1^{(1)}, a_2^{(1)}, a^{(2)}$, we can represent the XOR function. As demonstrated in figure 2.3, those PLAs form two *layers*. The outputs of the middle layer are inputs for the last layer. The middle layer is also known as the *hidden layer*. This model is called Multi-layer Perceptron (MLP) which is the foundation of *feedforward neural networks*.

2.1.2 Main components

That's essentially what *neural networks* are doing with the data. The architecture or the design of the networks may vary though. However, they are mostly built on robust and standard components.

Layers

Besides the *input layer* and the *output layer*, a model can have many *hidden layers* in the middle. The number of layers in a model will be counted as the total amount of hidden layers plus 1, denoted as $\mathbf{L}$. In the figure below, the model's $L = 3$.

Units

One node in a layer is called one *unit*. The input layer's units are called input units, so respectively, we also have hidden units and output units. The inputs of hidden layers are denoted as z, the output of each unit is denoted as $\mathbf{a}$ (the value of *activation* function with argument z). The output of unit $\mathbf{i^{th}}$ in layer l is $a_i^{(l)}$. Let $d^{(l)}$ be the number of units in layer l (without biases). The vector shows the outputs of layer l is written as $\mathbf{a}^{(l)} \in \mathbb{R}^{d^{(l)}}$.

Weights and Biases

In a model with $\mathbf{L}$, there are $\mathbf{L}$ parameters matrices. The matrices are denoted as $\mathbf{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l-1)}}$, where $\mathbf{l} = 1, 2, \ldots, \mathbf{L}$ while $\mathbf{W}^{(l)}$ stands for the connections from layer $l-1$ to layer l (if the input layer is layer 0). Particularly, $w_{ij}^{(l)}$ symbolize the connection from unit $\mathbf{i}$ of layer $(\mathbf{l-1})$ to unit $\mathbf{j}$ of layer $(\mathbf{l})$. The biases of layer $(\mathbf{l})$ are written as $\mathbf{b^{(l)}} \in \mathbb{R}^{\mathbf{d^{(l)}}}$. A good model always requires compatible weights and biases. The collection of weights and biases are respectively denoted as $\mathbf{W}$ and $\mathbf{b}$

Activation functions

Beside input units, any output of the model is calculated as:

$$a_i^{(l)} = \mathbf{f}(\mathbf{w}_i^{(l)T}\mathbf{a}^{(l-1)} + b_i^{(l)}) \tag{2.1}$$

Whilst $f()$ is a (non-linear) *activation function*. The equality after vectorization is written as:

$$a^{(l)} = \mathbf{f}(\mathbf{W}^{(l)T}\mathbf{a}^{(l-1)} + b^{(l)}) \tag{2.2}$$

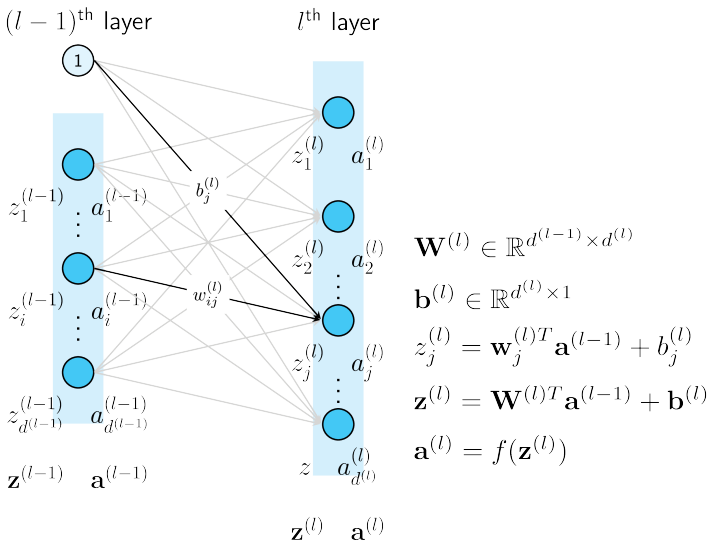


Figure 2.5: Notation used in MLP
Multi-layer Perceptron and Backpropagation [20]

Notice that the activation function is applied *element-wisely* to the matrix (or vector). The output layer may directly use the input meaning that the activation function is the identity function where the output is the same as the input. In the classification problems, the output layer normally uses a specific function like *Softmax* to calculate the probabilities of a data point fall into each class. Also, the activation function may vary for each unit, in the same network, the same activations are usually used to keep the calculation process simple.

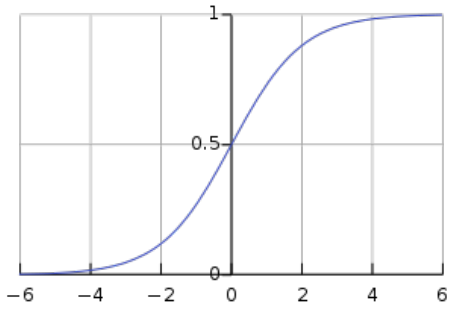


Figure 2.6: Sigmoid function
Source: Wikipedia [34]

Sigmoid & Tanh function *Sigmoid function* $\partial(z) = \frac{1}{1+e^{-z}}$ with *curve* in Fig. 2.6. If the input value is large, the function produce output approximately equals 1. While the input value is small (negative), the output will be closely 0. The function was commonly used because it is easy to differentiate:

$$\frac{d\partial}{dx} = \partial(z) \cdot (1 - \partial(x)).$$

But recently, it becomes less popular because of one fatal weakness: its saturation kill gradients. Particularly, when the inputs have significantly large absolute values, the gradient of the function is nearly 0 which means the unit's corresponding parameter will not be updated.

Tanh function $\phi(x) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$ also shares the same drawback. That's why nowadays, people start switching to different non-linearities.

ReLU Function *Rectified Linear Unit* $f(s) = max(0, s)$ is used widely because of its simplicity. Its graph, as illustrated in Fig. 2.7, looks fairly straightforward. The function significantly shortens the training duration of deep networks due to its simple calculation and fast differentiation (gradient equals 1 if the inputs are larger than 0, equals 0 if inputs are smaller than 0). Noted that $f^{'}(0) = 0$.

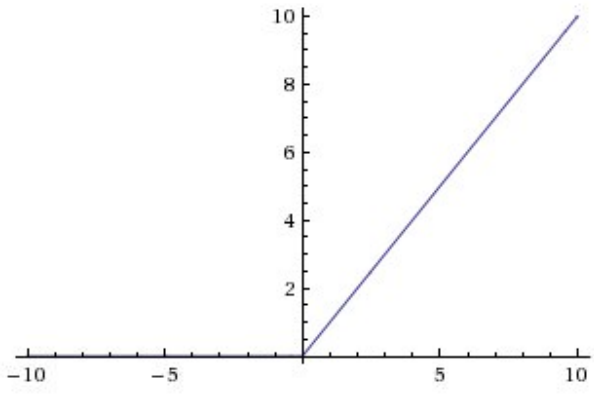


Figure 2.7: Rectified Linear Unit Function
Source: Kaggle

2.1.3 Forward Propagation

Forward propagation (or **forward pass**) refers to the measurement and storage of intermediate variables (including outputs) for the model from input to output. We operate sequentially through the mechanics of a neural network with one hidden layer. Given that the input example $\mathbf{x} \in \mathbb{R}^d$ and our hidden layer does not include bias. Here, we have our intermediate variable:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x}$$

where $\mathbf{W^{(1)}} \in \mathbb{R}^{\mathbf{h}\times\mathbf{d}}$ is the weight parameters of the hidden layer. Then we activate the intermediate variable $\mathbf{z} \in \mathbb{R}^h$ to obtain:

$$\mathbf{h} = \phi(\mathbf{z}).$$

The process is repeated as long as we need to produce a satisfactory prediction.

2.2 Convolutional Neural Networks (CNNs/ConvNets)

The word *"convolutional"* indicates that there will be a mathematics operation named **convolution**. Since the CNNs are much more complicated than regular fully connected neural networks in terms of architecture and building blocks, here are some necessary terminologies before we dive in further:

1. A **tensor** is an n-dimensional matrix where $n > 2$. A 2D RGB image can also be considered as a tensor since it is an overlap of 3 same images in 3 different channels: reg, green, and blue.

2. A **kernel** is a set of *weights* represented as a matrix that can be learned to extract features from the input that distinguish one from others.

Recently, captcha verification has been overcame by robots. So instead of using the old test, we decide to use the integrated camera to capture images of users and deep learning to identify the human face. So our problem is to examine whether there is a human face in a $64 * 64$ photo or not. As a start, we use the old-fashioned neural networks illustrated in Fig. 2.2 to do the task. Each node in all hidden layers is connected to all the nodes from the previous layer so we define that layer as a **fully connected layer**. A model that contains only this kind of layer is called a **fully connected neural network** (**FCN**). However, a $64 * 64$ image or a $64 * 64 * 3$ tensor needs 12288 nodes input layer to store each pixel of the image. If the first hidden layer has 1000 nodes then we will have $122800 * 1000$ weights between the input layer and the hidden layer, plus 1000 biases, and we get 1289000 parameters. Knowing that there are still many hidden layers in our model, we need to find a better solution. By applying the convolutional operation, we can avoid optimizing an enormous amount of parameters and still be able to extract the images' features. So we call it **Convolutional Neural Networks**.

Essentially, **Convolutional Neural Networks**, or **CNNs**, **ConvNets** are *deep neural networks* well suited for visual imagery processing. *CNNs* were inspired by biological processes. The connectivity pattern between neurons resembles the organization of the animal visual cortex. Individual cortical neurons respond to stimuli only in a restricted region of the visual field known as the *receptive field*. The receptive fields of different neurons partially overlap such that they cover the entire visual field. [29].

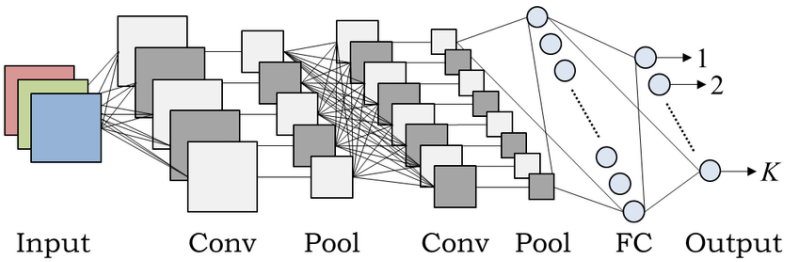


Figure 2.8: An example of CNN Architecture
Source: Consecutive Dimensionality Reduction by Canonical Correlation Analysis for Visualization of Convolutional Neural Networks (2017) by Hidaka [8]

Next, we will be explaining what is convolution. Then we will discuss what is the motivation behind the operation. After grasping the essence of CNNs, we can start familiarizing ourselves with the components that are the backbone of the networks.

2.2.1 What is convolution?

Convolution involves performing an element-wise dot product between a matrix and the unique **kernel**. By defining the **kernel** as a square matrix with the size $k*k$ while k is usually odd (1,3,5,7,9,...etc). Here is an example of a $3*3$ kernel where the operation is denoted as:

$$Y = X \otimes W$$

.

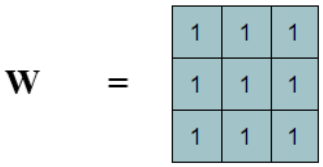


Figure 2.9: Kernel Visualization

With $m * n$ matrix $\mathbf{X}$, for each element $\mathbf{x_{i,j}}$, we extract 1 square matrix A from X which takes size from the kernel using $\mathbf{x_{i,j}}$ as the center element (this is why the kernel's size is usually odd). Then we sum all the products of corresponding pairs between A and W, then record the result to the new matrix Y.

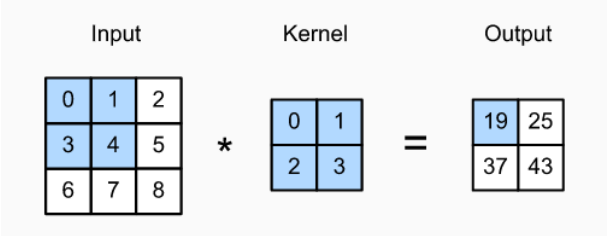


Figure 2.10: Convolution operation
The shaded portions are first output elements as well as the input and kernel tensor elements used for the output computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.
Source: Dive into Deep Learning [35]

Padding

This operation output, Y is always smaller than X in terms of size. In some cases, we want the size unchanged. So we add 0s on top of X's border. We convolve the kernel with the new X, and we get

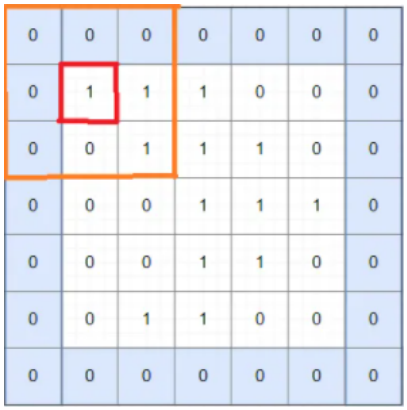


Figure 2.11: Matrix X after zero-padding
Image Processing [22]

new matrix Y with the size of old X. The operation is still *convolution*, but with **padding** $= 1$. When padding $= k$, we add k layers of 0s on top of the matrix's border.

Stride

When performing the convolution, we start with the kernel at the top-left corner of the input matrix and then slide it all over the matrix. By default, we slide one element at a time. But in many cases, we want to reduce the output size or increase computational efficiency, so we slide more than one element at a time. The number of elements traversed per slide is referred to as **stride**. Convolutional operation of

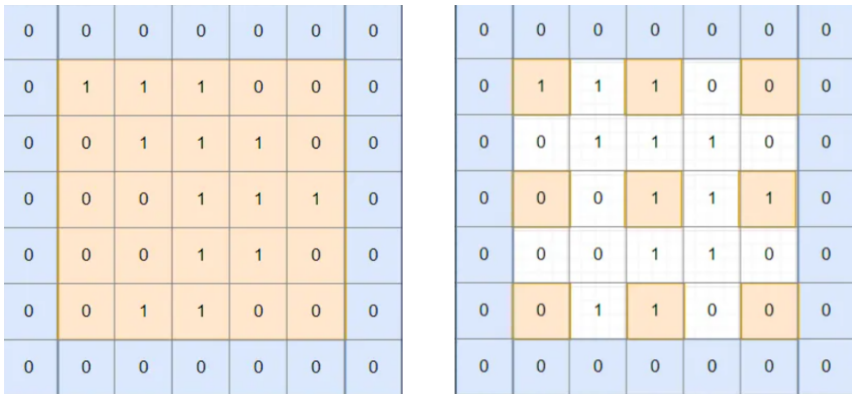


Figure 2.12: Stride
The blue shade square is the added padding and the yellow shade square marked the center element of each slide. **Left**: Convolution with stride $= 1$, padding $= 1$ produce $5 * 5$ matrix.
Right: Convolution with stride $= 2$, padding $= 1$ produce $3 * 3$ matrix. Image Processing [22]

$m * n$ matrix X with $k * k$ kernel, stride $= s$, and padding $= p$ will produce $(\frac{m-k+2p}{s} + 1) * (\frac{n-k+2p}{s} + 1)$ matrix.

Motivation

The purpose of performing convolution on images is to sharpen, blur, detecting edges,... Different kernels will give us different results. For example,

Convolution also highlights 3 important ideas that can better a model: **sparse interaction, parameter sharing,** and **equivariant representations**. In traditional neural networks, all layers are fully connected whereas ConvNets have **sparse interaction** (which is the same as **sparse connectivity** or **sparse weights**). The idea is to only calculate what is important and relevant. The deeper units will only be affected by some specific shallower units while still indirectly connected to a sufficient amount of units. This is accomplished by making the kernel smaller than the input.

Next, imagine extracting edges from a 2D image, the example will introduce the idea of **parameter sharing**. Consider the edge as a feature, using the same kernel for different regions of the image. The idea help reduce memory requirements which overall increases the model efficiency. Lastly, if the input

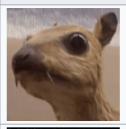
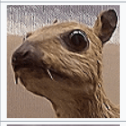
Operation	Kernel ω	Image result g(x,y)
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	

Figure 2.13: Convolution can perform many operations
Source: Wikipedia

changes and the output changes the same way then the function is equivariant. In the case of convolution, if we let g be any function that shifts the input, then the convolution function is **equivariant** to g.

2.2.2 Components of ConvNets

Convolutional Layers

Colored images are 3D tensor so our kernel in the first convolutional layer must also be a 3D tensor as well.

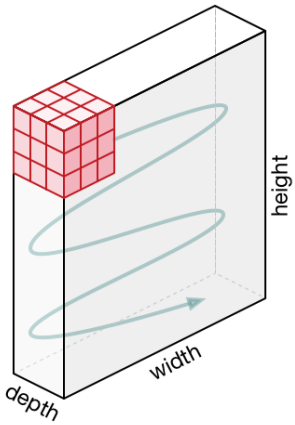


Figure 2.14: Convolution on colored image with $3 * 3 * 3$ kernel.
Source: Convolutional Neural Network [22]

Generally, the input consists of k matrices. We convolve each matrix with a kernel then sum all the results with the bias b to produce a matrix, the result matrices form a tensor of depth equals to the number of kernels.

We used multiple kernels to learn *multiple features* from the image. So let us assume that the input of a convolutional layer is a $H * W * D$ tensor. We convolve K kernels of size $F * F * D$ (where F is odd) to the

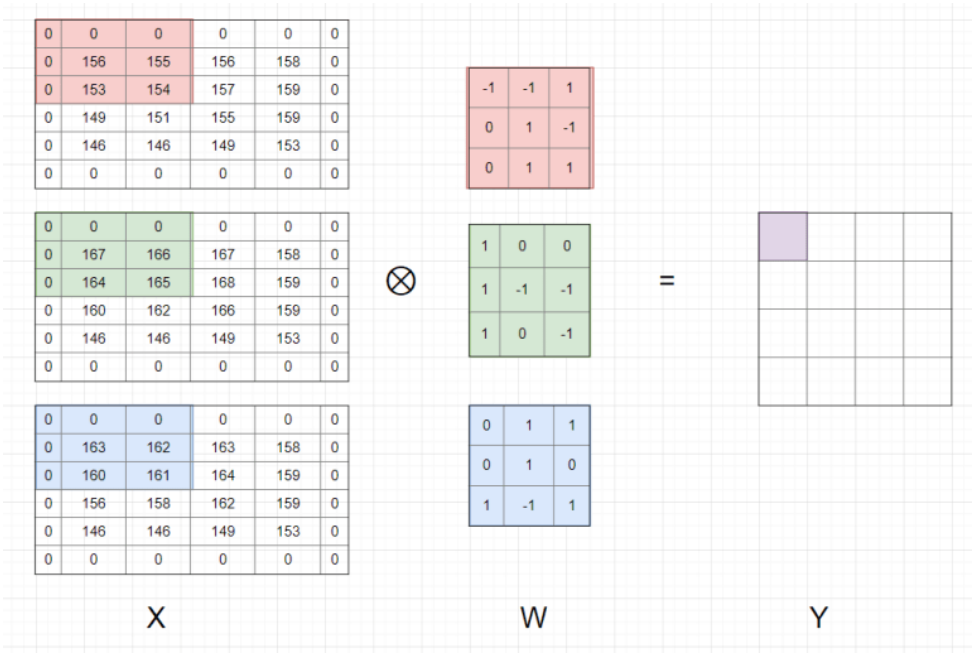


Figure 2.15: 3D Tensor convolution in matrix form
Source: Convolutional Neural Network [22]

tensor with stride S, padding P. Then we will get a $(\frac{H-F+2P}{S}+1)*(\frac{W-F+2P}{S}+1)*K$ 3D tensor. After adding the bias, the tensor is *activated* ($\mathbf{b}$ has K biases). The layer has $K*(F*F*D+1)$ parameters.

Pooling Layers

In the image, some groups of pixels can be redundant because their values are roughly the same. So we use the pooling layers to eliminate those futile pixels whilst decreasing the spatial size of the input, saving the memory.
The most common type of **pooling layer** is **Max-Pooling**. We select the *kernel size* and *stride*. The neuron will slide the kernel with the designed stride over the input whilst choosing only the *largest* value at each region the kernel hover on to yield a value for the output.

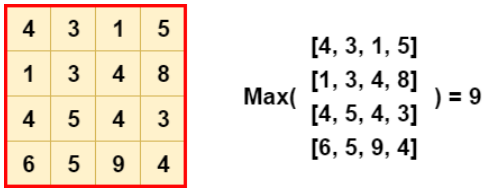


Figure 2.16: Pooling
Source: Machine Curve

Pooling layer commonly uses $2*2$ kernel with stride $= 2$ and **no** padding.

Fully Connected Layer

After lots of convolution and pooling, the output's $H*W*D$ tensor gets flatten into a single vector $\mathbf{z}$. $\mathbf{z}$ is the input of the fully connected layers, the model will continuously do forward pass (usually with softmax activation function) to measure the probability of face presence in the image. Then the model will decide whether it recognizes the human face or not.

2.3 Gradient Descent

Gradient descent is an iterative optimization algorithm for finding a local minimum of a differentiable function. To find a local minimum of a function using gradient descent, we take steps proportional to the negative of the gradient of the function at the current point [31].

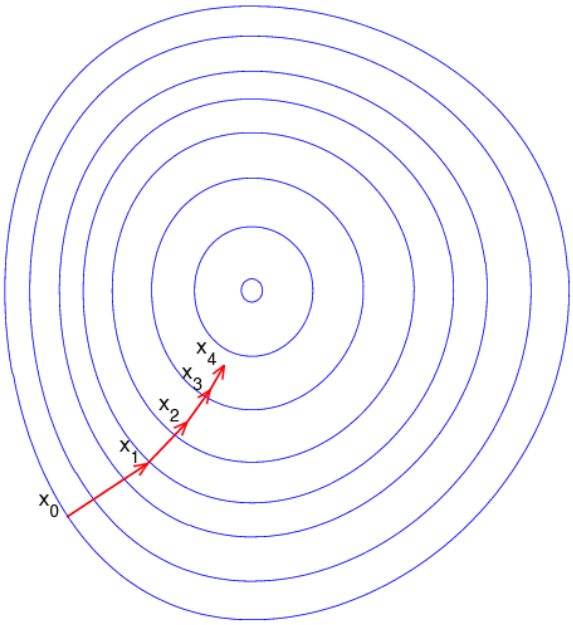


Figure 2.17: Gradient Descent Process Visualization
Source: Gradient Descent [31] – Wikipedia

Gradient descent is based on the observation that if the *multi-variable function* $\mathbf{F(x)}$ is defined and differentiable in a neighborhood of a point $\mathbf{a}$, then $\mathbf{F(x)}$ decreases *fastest* if one goes from $\mathbf{a}$, in the direction of the negative gradient of $\mathbf{F}$ at $\mathbf{a}$, $-\nabla\mathbf{F(a)}$. So we subtract the gradient of F at point a multiply with a coefficient α called the **learning rate** from a to move toward the local minimum.

$$a_{n+1} = a_n - \alpha \nabla F(a_n) \tag{2.3}$$

for $\alpha \in R_+$ small enough, then $F(a_n) \geq F(a_{n+1})$. Based on the observation, we start with a guess $\mathbf{x_0}$ for a local minimum of F, and considers the sequence $\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_2}, \ldots$ such that

$$x_{n+1} = x_n - \alpha \nabla F(x_n), n \geq 0. \tag{2.4}$$

From which we can derive a monotonic sequence

$$F(x_0) \geq F(x_1) \geq F(x_2) \geq \ldots, \tag{2.5}$$

that hopefully converges to the desired local minimum or the **global minimum** in the best case.
This process is illustrated in the figure above. Here F is assumed to be defined on the plane, and that its graph has a bowl shape. The blue curves are the contour lines, that is, the regions on which the value of F is constant. A red arrow originating at a point shows the direction of the negative gradient at that point. Note that the (negative) gradient at a point is orthogonal to the contour line going through that point. We see that gradient descent leads us to the bottom of the bowl, that is, to the point where the value of the function F is minimal [31].

Why the gradient is the direction of the steepest ascent? Each component of the gradient indicates how fast is our function changing. Imagining a ball represents the function, we would like to figure out how fast the ball rolling in some random direction. Let $\overrightarrow{v}$ be a unit vector, we can naturally project along this direction via the dot product $\text{grad}(f(a)) \cdot \overrightarrow{v}$. So in what direction is this quantity maximal? Given that

$$\text{grad}(f(a)) \cdot \overrightarrow{v} = |\text{grad}(f(a))||\overrightarrow{v}| \cos(\theta)$$

Since $\overrightarrow{v}$ is unit, we have $|\text{grad}(f(a))| \cos(\theta)$, which is maximal when $\cos(\theta) = 1$ or when $\overrightarrow{v}$ points in the same direction as $\text{grad}(f(a))$.

2.3.1 Cost Function

When optimizing models, we always hope that the difference δ between the ground truth y and the prediction $\hat{y}$ is as little as possible. The same thing applied to all pairs of *input, output* (x_i, y_i) for i

ranging from 1 to N- the amount of observed data, we want the total difference to be minimum so we defined it as the *loss function*. There are many types of loss functions, but the **MSE** (Mean Squared Error) is most commonly used:

$$MSE = \frac{\Sigma_{i=1}^{N}(y_i - \hat{y}_i)^2}{N}$$

Which means we should find the optimal $\mathbf{w}$ & $\mathbf{b}$ since $\hat{y} = active(\mathbf{w} \cdot x + \mathbf{b})$. And now we have a new function

$$J(\mathbf{W}, \mathbf{b}) = \frac{\Sigma_{i=1}^{N}(y_i - active(\mathbf{w^i} \cdot x_i + \mathbf{b^i}))^2}{N}$$

called the **cost function**, also it's our **objective function**. In reality, finding the optimal value to get the cost function to 0 is hard, so we use *gradient descent* to bring the cost function as close to 0 as possible.

2.3.2 Batch Gradient Descent

In Machine Learning, Gradient Descent, also known as **Batch Gradient Descent** or shortly GD, is used to optimize the **cost function**. We call the cost function f which takes the argument θ known as the set of parameters that needs optimization. The derivatives of the function at a point θ is denoted as $\nabla_\theta f(\theta)$. The algorithm starts with a guess θ_0, then continuously update the set of parameters by subtracting the product of the learning rate η and the derivatives from the current parameters

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta f(\theta_t)$$

Or simply put:

$$\theta := \theta - \eta \nabla_\theta f(\theta)$$

2.3.3 Stochastic Gradient Descent

In this algorithm, we calculate the derivative of the loss function based on *one* point of data $\mathbf{x_i}$ then update the weights accordingly. Let us assume that $\mathbf{f_i}(\mathbf{x})$ is the loss function of the training dataset with $\mathbf{n}$ examples, an index of $\mathbf{i}$, and parameter vector of θ, consequently we can formulate our objective function like this:

$$\mathbf{f}(\theta) = \frac{\mathbf{1}}{\mathbf{n}} \mathbf{\Sigma_{i=1}^{n}} \mathbf{f_i}(\theta).$$

The gradient of the objective function at θ is computed as

$$\nabla \mathbf{f}(\theta) = \frac{\mathbf{1}}{\mathbf{n}} \mathbf{\Sigma_{i=1}^{n}} \nabla \mathbf{f_i}(\theta)$$

If we use GD, the computing cost for each independent variable iteration is $\mathbf{O(n)}$, which grows linearly with $\mathbf{n}$. Accordingly, the performance of the model may not meet our needs. **Stochastic gradient descent (SGD)** helps reduce computational operations. In each iteration, we uniformly sample an index $\mathbf{i} \in \mathbf{1}, \ldots, \mathbf{n}$ randomly, and compute the gradient to update the weights. We can see that the time complexity for each iteration drops from $O(n)$ to the constant $O(1)$. But the trajectory of the variables in the SGD is noisier than that of GD: Even when we arrive near the minimum, we are still subjected to the

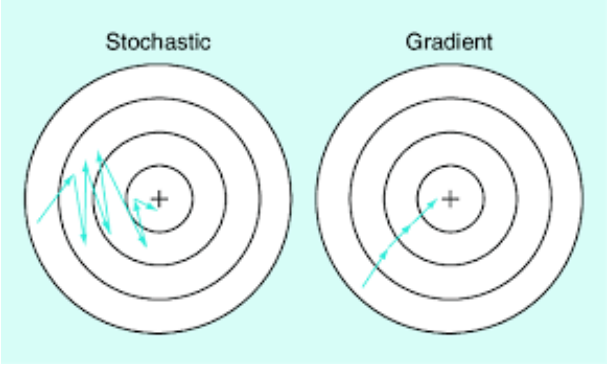


Figure 2.18: SGD compares to vanilla GD
Carpenter, Kristy & Cohen, David & Jarrell, Juliet & Huang, Xudong. (2018). Deep learning and virtual drug screening. Future Medicinal Chemistry. 10. 10.4155/fmc-2018-0314.

uncertainty injected by the instantaneous gradient $\alpha \nabla f_i(x)$ (where α is the learning rate). In the worst scenario, the gradient will not improve after additional steps. Thus, the algorithm is not computationally effective sine both GPUs and CPUs cannot utilize the full power of *vectorization*. So instead of using only one observation at a time, we use **n** observations where n is greater than 1 but still a lot smaller than the total amount of data. The gradient g_t of the small batch is calculated as

$$g_t = \partial_w \frac{1}{|B_t|} \partial_{i \in B_t} f(x_i, w)$$

where both x_t and elements of mini-batch B_t are drawn randomly from the training set. In practice, we pick the mini-batch's size large enough to offer good computational efficiency.

Changing the learning rate might be effective, but if we pick this too small, we will not make any meaningful progress initially. In contrast, if we pick it too large, the gradient might never converge. Keep its limitation in mind, reducing the learning rate *dynamically* as optimization continues is a perfect way to solve this conflict.

2.3.4 Momentum

We have found that in the case of noisy gradients, we need to be extra careful when it comes to choosing the learning rate. So we strive to explore better optimization algorithms, especially for certain types of problems mentioned before. We knew mini-batch SGD as a means for accelerating computation. It is also helpful in reducing the amount of variance. The idea is to utilize that advantage beyond averaging gradients on a mini-batch by using a **leaky average**:

$$v_t = \beta v_{t-1} + g_{t,t-1}$$

for some $\beta \in (0, 1)$. **v** is called *momentum*. It accumulates past gradients similar to how a heavy ball rolling down the objective function landscape integrates over past forces. The alternative gradient now points in the direction of a weighted average of past gradients which enables us to recognize most of the benefits of averaging over batch without the cost of actually computing the gradients on it. That is the basis for what is now known as accelerated gradient methods, such as gradients with momentum. Using v_t instead of the gradient g_t yields the following update equations:

$$v_t \leftarrow \beta v_{t-1} + g_{t,t-1},$$

$$x_t \leftarrow x_{t-1} - \alpha_t v_t$$

2.3.5 Adagrad

One of the key issues now is that the learning rate decreases at a predefined schedule of effectively $O(t^{-\frac{1}{2}})$. While this is generally appropriate for convex problems, it might not be ideal for non-convex ones. Yet, the coordinate-wise adaptability of Adagrad is highly desirable as a preconditioner. Adagrad is an optimizer with parameter-specific learning rates, which are adapted relative to how frequently a parameter gets updated during training. The more updates a parameter receives, the smaller the learning rate. Adagrad is particularly effective for sparse features where the learning rate needs to decrease more slowly for infrequently occurred terms.

2.3.6 Root Mean Square Propagation (RMSProp)

Similar to Adagrad, RMSProp also uses the square of the gradient to scale coefficients.

$$s_t \leftarrow \gamma s_{t-1} + (1 - \gamma) g_t^2,$$

$$x_t \leftarrow x_{t-1} - \frac{\alpha}{\sqrt{s_t + \epsilon}} \odot g_t$$

The constant ϵ is positive to ensure that we do not divide by zero or take large step sizes.

2.3.7 Adaptive Moment Estimation (Adam Optimizer)

We have encountered numerous optimization algorithms. SGD is more effective than GD when solving optimization problems due to its inherent resilience to redundant data. Mini-batch SGD utilizes vectorization, using larger sets of observations in one mini-batch. The momentum method adds

a mechanism for aggregating a history of past gradients to accelerate convergence. Adagrad uses per-dimension scaling allowing preconditioner efficient computation. RMSProp decouples per-coordinate scaling from a learning rate adjustment. Ultimately, **Adam Optimizer** fuses all of the mentioned techniques' essence into a single optimizer. Adam uses *leaky averaging* (exponential weighted moving averages) to gain an estimation of both the momentum and also the second moment of the gradient using state variables

$$v_t \leftarrow \beta_1 v_{t-1} + (1-\beta_1) g_t,$$
$$s_t \leftarrow s_{t-1} + (1-\beta_2) g_t^2 \odot sgn(g_t^2 - s_{t-1})$$

Here, β_1 and β_2 are non-negative weights. Usually, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. Notice that if we initialize $v_0 = s_0 = 0$ we will have a significant amount of bias initially towards smaller values. This can be addressed by using the fact that $\partial_{i=0}^{t} \beta^i = \frac{1-\beta^t}{1-\beta}$ to re-normalize terms. Accordingly, the normalized state variables are given by

$$\hat{v}_t = \frac{v_t}{1-\beta_1^t} \text{ and } \hat{s}_t = \frac{s_t}{1-\beta_2^t}.$$

With the proper estimation, we can now write out the updated equations. First, we rescale the gradient to obtain

$$g_t' = \frac{\alpha \hat{v}_t}{\sqrt{\hat{s}_t} + \epsilon}.$$

Typically we use $\epsilon = 10^{-6}$ for a good trade-off between numerical stability and fidelity. Finally, we update the objective function

$$x_t \leftarrow x_{t-1} - g_t'.$$

2.4 Backward Propagation

In short, **backpropagation** calculate the partial derivatives of a cost function regarding the parameters of our models using the **chain rule**. Let $\mathbf{Y} = \mathbf{f}(\mathbf{X})$ and $\mathbf{Z} = \mathbf{g}(\mathbf{Y})$ where $\mathbf{X}, \mathbf{Y}, \mathbf{Z}$ are tensors of random shapes. By using the chain rule, we can compute the derivative of Z respected to X via

$$\frac{\partial Z}{\partial X} = \mathbf{prod}(\frac{\partial Z}{\partial Y}, \frac{\partial Y}{\partial X}).$$

Here, *prod* operator is the multiplication of its arguments. Consider a simple neural network with one hidden layer, we denote its parameters as $\mathbf{W^{(1)}}$ and $\mathbf{W^{(2)}}$. Backprop calculates the gradients $\frac{\partial J}{\partial W^{(1)}}$ and $\frac{\partial J}{\partial W^{(2)}}$ where $\mathbf{J}$ is our objective function. To accomplish this, we apply the chain rule and calculate consecutively the gradient of each intermediate variable and parameter. The order of calculations is reversed relative to the process in the forward pass: we start with the output layer and work our way towards the parameters. The first step is to calculate the gradients of the objective function $J = L + s$ respected to the loss term L and the regularization term s.

$$\frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1.$$

Then we calculate the gradient of J respected to the variable of the output layer $\mathbf{o}$:

$$\frac{\partial J}{\partial \mathbf{o}} = \text{prod}\left(\frac{\partial J}{\partial L}, \frac{\partial L}{\partial \mathbf{o}}\right)$$

Next, we calculate the gradients of the regularization term respected to both parameters:

$$\frac{\partial s}{\partial \mathbf{W}^{(1)}} = \lambda \mathbf{W}^{(1)} \text{ and } \frac{\partial s}{\partial \mathbf{W}^{(2)}} = \lambda \mathbf{W}^{(2)}.$$

Now we can calculate the gradient $\partial J / \partial \mathbf{W}^{(2)} \in \mathbb{R}^{q \times h}$ of the model parameters closest to the output layer. Using the chain rule yields

$$\frac{\partial J}{\partial \mathbf{W}^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{W}^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(2)}}\right) = \frac{\partial J}{\partial \mathbf{o}} \mathbf{h}^\top + \lambda \mathbf{W}^{(2)}$$

The gradient respected to the hidden layer's outputs $\partial J / \partial \mathbf{h} \in \mathbb{R}^h$ is given by

$$\frac{\partial J}{\partial \mathbf{h}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{o}}, \frac{\partial \mathbf{o}}{\partial \mathbf{h}}\right) = {\mathbf{W}^{(2)}}^\top \frac{\partial J}{\partial \mathbf{o}}.$$

Since the activation function ϕ applies element-wise, calculating the gradient $\partial J/\partial \mathbf{z} \in \mathbb{R}^h$ of the intermediate variable $\mathbf{z}$ requires that we use the element-wise multiplication operator, which we denote by $\odot$:

$$\frac{\partial J}{\partial \mathbf{z}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{h}}, \frac{\partial \mathbf{h}}{\partial \mathbf{z}}\right) = \frac{\partial J}{\partial \mathbf{h}} \odot \phi'\left(\mathbf{z}\right).$$

Finally, we can obtain the gradient $\partial J/\partial \mathbf{W}^{(1)} \in \mathbb{R}^{h \times d}$ of the model parameters closest to the input layer. According to the chain rule, we get

$$\frac{\partial J}{\partial \mathbf{W}^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial \mathbf{z}}, \frac{\partial \mathbf{z}}{\partial \mathbf{W}^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}, \frac{\partial s}{\partial \mathbf{W}^{(1)}}\right) = \frac{\partial J}{\partial \mathbf{z}} \mathbf{x}^\top + \lambda \mathbf{W}^{(1)}.$$

Chapter 3

Residual Networks

3.1 Motivation

3.1.1 Vanishing Gradient in Very Deep Neural Networks

The enormous evolution of deep convolutional neural networks has led to a series of breakthroughs in image classification [7]. Recent evidence reveals that network depth is a vital element that plays a significantly important role in a neural network, many other nontrivial visual recognition tasks have also greatly benefited from very deep models. Thus, people "believe" that the deeper the model is, the better it learns. But, in fact, there is a giant obstacle when the model becomes "very deep" called ***vanishing/exploding gradient*** which hampers the convergence of the loss function.

Vanishing Gradient

While performing backward propagation (as mentioned in **??**), to compute the gradient of the first layers, we have to multiply many "less than 1" numbers. As a consequence, gradients tend to converge to 0, which makes the gradient update step become meaningless. Hence, the gradient updating step is not able to change the weights of those layers significantly, which also means that the model is slow or even unable to learn. On the other hand, if we have to multiply many "greater than 1" numbers to compute the gradient, there will be **exploding gradient** which makes the model worse and worse after each epoch.

3.1.2 Learning to 1 versus Learning to 0

In traditional neural networks, the model learns to approximate $\hat{y} \approx y$ such that $\hat{y} = Wx + b$. It means the goal of the learning process is to find the weight matrix W that approximates y as well as possible. We call this process ***learning to 1*** since in the very last layers, the weights approximately equal to 1 (identity matrix), and the biases approximately equal to 0 (zero matrix). If we keep on learning, it will become saturated and unable to update the network.

However, instead of learning $\hat{y} \approx y$, we approximate the residual $r = \hat{y} - y$ so that it converges to 0. It can be understood as: instead of trying to maximize the precision of the predicted value, we try to minimize the difference between the predicted value and the expected one. ***Learning to 0*** can easily converge to 0 by zero-initialization. The basic idea of *Residual Network* follows this intuition, it learns the *residual*, which we will explain more in the next section.

3.2 Residual Learning

Let $\mathcal{H}(x)$ denotes the formal desired underlying mapping. In *Residual Networks (ResNets)*, another mapping of $\mathcal{F}(x)$ defined by equation 3.1 is introduced [7].

$$\mathcal{F}(x) = \mathcal{H}(x) - x \tag{3.1}$$

So, in order to estimate the output, the initial mapping is recast into:

$$\mathcal{H}(x) = \mathcal{F}(x) + x \tag{3.2}$$

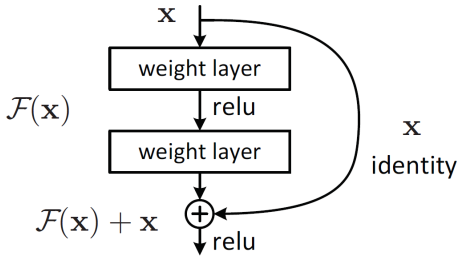


Figure 3.1: Residual learning: a building block
Source: He, Zhang, Ren, and Sun [7]

where $\mathcal{F}(x)$ is "residual". Despite the fact that the core logic of ResNets is this simple equation, it has the ability to make dramatic changes in the whole Deep Learning industry.

Following the idea of learning to 0 mentioned in section 3.1.2, He, Zhang, Ren, and Sun [7] hypothesize that it is easier to optimize the residual mapping than to optimize the original mapping. As the model gets deeper, if an identity mapping were optimal, it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers.

The formulation of $\mathcal{F}(x) + x$ is represented by a feed-forward neural network with a **shortcut connection** that skips one or more layers (shown in Fig. 3.1). In Residual Networks, these shortcut connections simply perform identity mappings, and their outputs are added to the outputs of the previous layers [7]. Identity shortcut connections do not require any extra parameters and they do not cost more computational complexity. The entire network can be easily implemented using common libraries without modifying the solvers and trained end-to-end by SGD with backpropagation [7].

The reformulation 3.1 is motivated by the counter-intuitive phenomena about the **degradation problem**: a deeper model should have a training error no greater than the shallower one. **Degradation** happens when the solvers have difficulties in approximating identity mappings by multiple nonlinear layers [7]. By applying the residual learning reformulation (equation 3.2), if identity mappings are optimized, the solvers can simply drive the weights of those nonlinear layers toward zero to approximate identity mappings.

3.3 Identity Mapping by Shortcuts

From the network architecture illustrated in Fig. 3.1, the output of the block y is defined by the equation:

$$y = \mathcal{F}(x, \{W_i\}) + x \tag{3.3}$$

where $\mathcal{F}(x, \{W_i\})$ represents the to-be-learned residual mapping, x is the identity mapping of the previous layer, and W_i indicates the weight of the i-th layer. As shown in Fig. 3.1, it can be understood that $\mathcal{F} = W_2\sigma(W_1x)$ where σ denotes ReLU activation function, W_1 and W_2 are weights of the first and second layers, and biases are omitted for a simpler notation.

The **shortcut connection** performs an element-wise addition $\mathcal{F} + x$ so that it neither require any extra parameters nor cost more computational complexity. A simple reason for this breakthrough is that it does not add any extra layers to the original network, and the element-wise addition is negligible during the computation process.

To achieve this, the dimension of $\mathcal{F}$ must be similar to the dimension of x. Or else, when changing input and output channels for instance, there must be a linear projection W_s over x so that it matches the dimension of $\mathcal{F}$:

$$y = \mathcal{F}(x, \{W_i\}) + W_sx \tag{3.4}$$

Equation 3.4 can also be applied in equation 3.3 with W_s is a square matrix. However, by experiments, He, Zhang, Ren, and Sun [7] had concluded that equation 3.3 is sufficient and more economical.

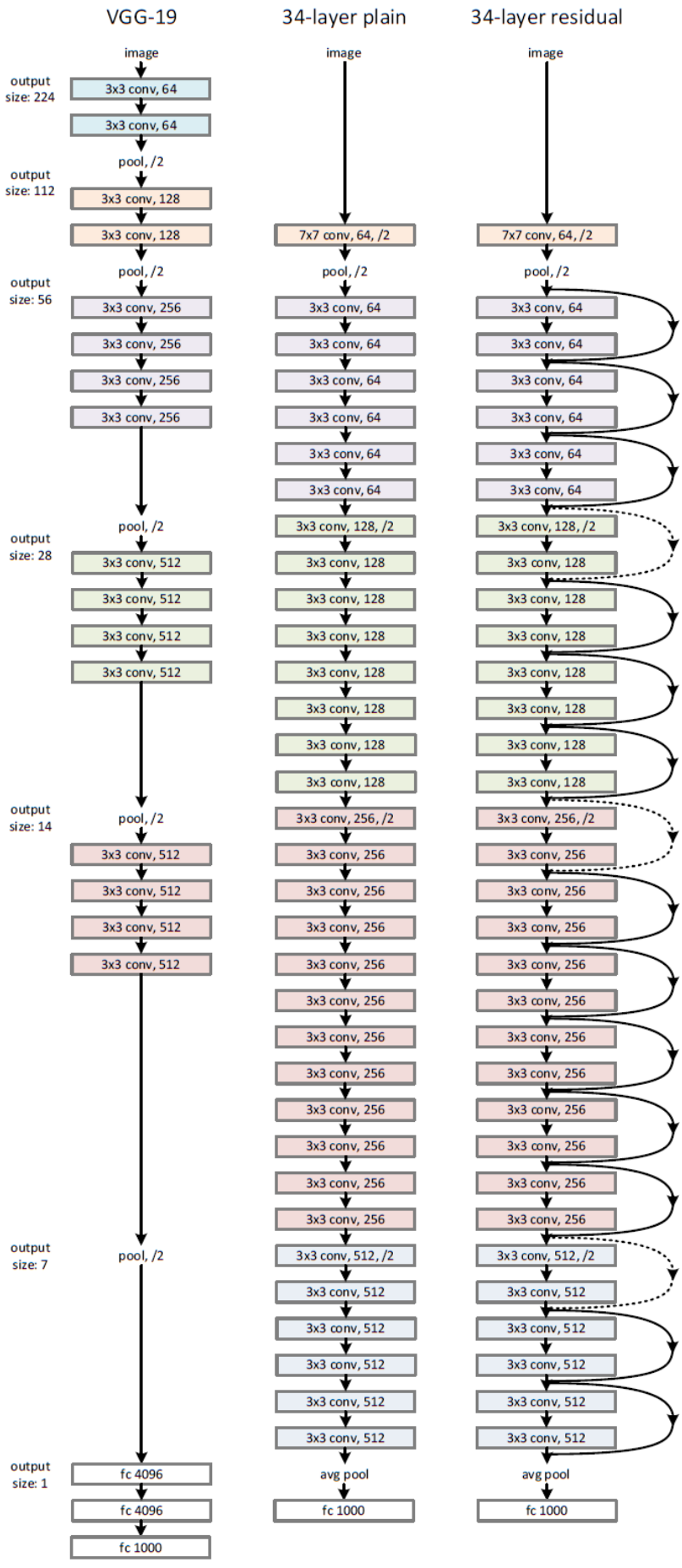


Figure 3.2: Example network architectures for ImageNet
Left: the VGG-19 model. **Middle**: a plain network with 34 parameter layers.
Right: a residual network with 34 parameter layers. The dotted shortcuts increase dimensions.
Source: He, Zhang, Ren, and Sun [7]

3.4 Network Architecture

A famous basic instance of Residual Network is a deep neural network with 34 parameter layers with **shortcut connections**, which is also known as ***ResNet34*** detailed in fig. 3.2 (right). Its target is to predict the label of a given image based on training the ImageNet dataset [10] and outputting a 1000-dimensional vector. A deeper residual network of 152 layers also proposed by He, Zhang, Ren, and Sun won the 1st place in the ILSVRC 2015 classification competition as well as ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation in ILSVRC & COCO 2015 competitions [7]. ***ResNet152*** is the deepest neural network ever presented on ImageNet, while still having lower complexity than the rival ***VGG19***. This strong evidence shows that the residual learning principle is generic [7].

Motivated by the VGG19 network, He, Zhang, Ren, and Sun [7] proposed a plain 34-layer deep neural network and then insert shortcut connections that jump over 2 layers to create a 34-layer residual network. The identity shortcuts 3.3 is applied directly when the input and output are of the same dimensions (illustrated by solid line shortcuts in Fig. 3.2). When the dimensions are increased, identity shortcuts 3.4 is applied (illustrated by dotted line shortcuts in Fig. 3.2) to match the dimensions of the input and output.

Comparing the original VGG19 network (Fig. 3.2, left) with 19.6 billion FLoating point OPerations (FLOPs), the plain neural network baseline with 34 layers shown in Fig. 3.2 (middle) has a much lower computational complexity with only 3.6 billion FLOPs. The shortcut connections do not result in any extra layers or computational resources, thus, ResNet34 and the plain network have the same number of FLOPs of 3.6 billion [7].

3.5 Backpropagation in Residual Network

Backpropagation is a general algorithm that can be applied anywhere. ResNet is no exception.
Let $y = \mathcal{F}(x) + x$. Consider our main objective here is to calculate $\frac{\partial E}{\partial x}$, without the shortcut path, we would have

$$\frac{\partial E}{\partial x} = \frac{\partial E}{\partial y} * \frac{\partial y}{\partial x} = \frac{\partial E}{\partial y} * F'(x)$$

Now with shortcut connection,

$$\begin{aligned} \frac{\partial E}{\partial x} &= \frac{\partial E}{\partial y} * \frac{\partial y}{\partial x} \\ &= \frac{\partial E}{\partial y} * (1 + F'(x)) \\ &= \frac{\partial E}{\partial y} + \frac{\partial E}{\partial y} * F'(x) \end{aligned} \tag{3.5}$$

3.6 MNIST Digits Classifiers

In this section, we train our model ResNet-34 to recognize handwritten digits using the **MNIST** dataset. The MNIST dataset contains $60,000$ training images and $10,000$ testing images which are all in greyscale and normalized to fit into a 28x28 pixel bounding box. The images are centered to reduce preprocessing. Training a classifier on this dataset can be regarded as the hello world of image recognition.
The main purpose of the experiment is to illustrate ResNet's operation. The model's architecture is illustrated in Fig. 3.2. We use **PyTorch** - a popular deep learning framework to train our model. PyTorch's strength lies in its ability to create computation graphs really fast. The first step is always to import the necessary libraries. In the experiment, we train our model with the mini-batch size of 128. We use Stochastic Gradient Descent (SGD) with the learning rate of 0.01 to optimize the model's parameters in 10 epochs. Noted that, **epoch** is a term used in training models that indicates the number of passes of the entire training dataset the algorithm has completed. Also, while we are at it, an **iteration** is a term used in machine learning that indicates the number of times the algorithm's parameters are updated. If the batch size is the same size as the training dataset then the number of epochs is the number of iterations.

Then, we use PyTorch's DataLoader to create the training and test data for our model. Also, we re-scale input images to $[0, 1]$ range for better computation. The next step is to build our network.

Figure 3.3: MNIST dataset.
Source: Wikipedia

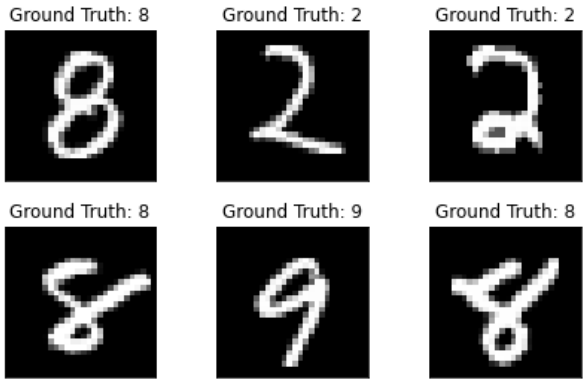


Figure 3.4: Labeled Images from the dataset

We build residual block and shortcut connection by construct a class that can stack layers while store the input of each block as the residual x.

```
class BasicBlock(nn.Module):

def __init__(self, inplanes, planes, stride=1, downsample=None):
super(BasicBlock, self).__init__()
self.conv1 = conv3x3(inplanes, planes, stride)
self.bn1 = nn.BatchNorm2d(planes)
self.relu = nn.ReLU(inplace=True)
self.conv2 = conv3x3(planes, planes)
self.bn2 = nn.BatchNorm2d(planes)
self.downsample = downsample
self.stride = stride

def forward(self, x):
residual = x

out = self.conv1(x)
out = self.bn1(out)
out = self.relu(out)

out = self.conv2(out)
out = self.bn2(out)

if self.downsample is not None:
residual = self.downsample(x)

out += residual
out = self.relu(out)

return out
```

Having the ingredients we need, we start constructing our model as a class.

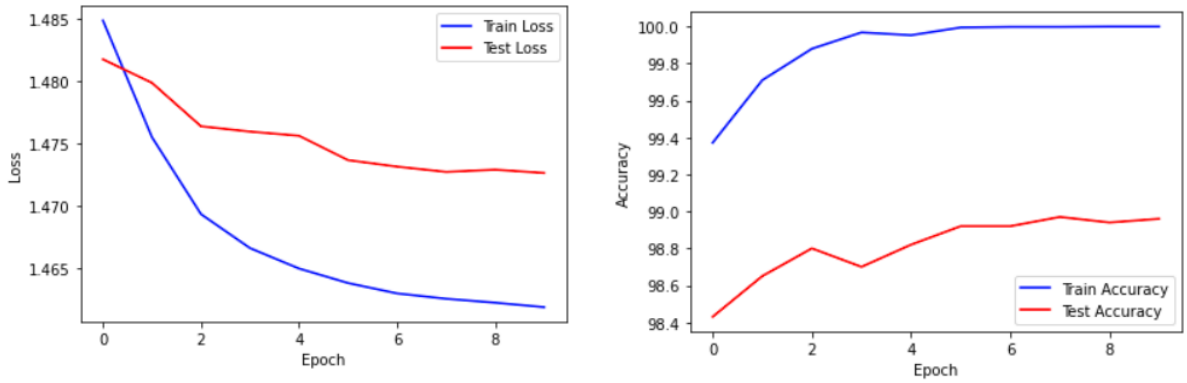
```
class ResNet(nn.Module):

def __init__(self, block, layers, num_classes, grayscale):
```

```
self.inplanes = 64
if grayscale:
in_dim = 1
else:
in_dim = 3
super(ResNet, self).__init__()
self.conv1 = nn.Conv2d(in_dim, 64, kernel_size=7, stride=2, padding=3,
bias=False)
self.bn1 = nn.BatchNorm2d(64)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
self.avgpool = nn.AvgPool2d(7, stride=1)
self.fc = nn.Linear(512 * block.expansion, num_classes)
```

Now, we just need to initialize our model and train it. After that, we visualize the process of loss descendant and accuracy ascendant to gain some better insight into ResNet's efficiency

As you can see in Fig. 3.5a, ResNet-34's loss reduces significantly after 250 seconds of training.



(a) ResNet-34's losses after each epoch.

(b) ResNet34's accuracies after each epoch.

Figure 3.5: ResNet-34's training progress visualized

We achieve near-perfect result just after a few epochs as demonstrated in Fig. 3.5b.

After the training, the result presented in Fig. 3.6 and Fig. 3.7 are good considered the fact that our model can perfectly predict the numbers represented by the images.

Nevertheless, our main purpose here is just to demonstrated how ResNet's modules are linked to each other. And it can be programmed in just a matter of seconds yet providing **state-of-the-art** image processing model. Here is the result of another experiment conducted by us using plain 34-layer CNN. Identical configurations was applied to both models but no big differences were found in models' performance. The training time takes roughly 250 seconds in total (25 seconds for each iteration). Models' accuracies are always above 98%. Naturally, ResNet was created to tackle very deep neural networks so we might not see much of a difference in this section. The vanilla CNN can also accurately recognize the number given by the images in Fig. 3.10.

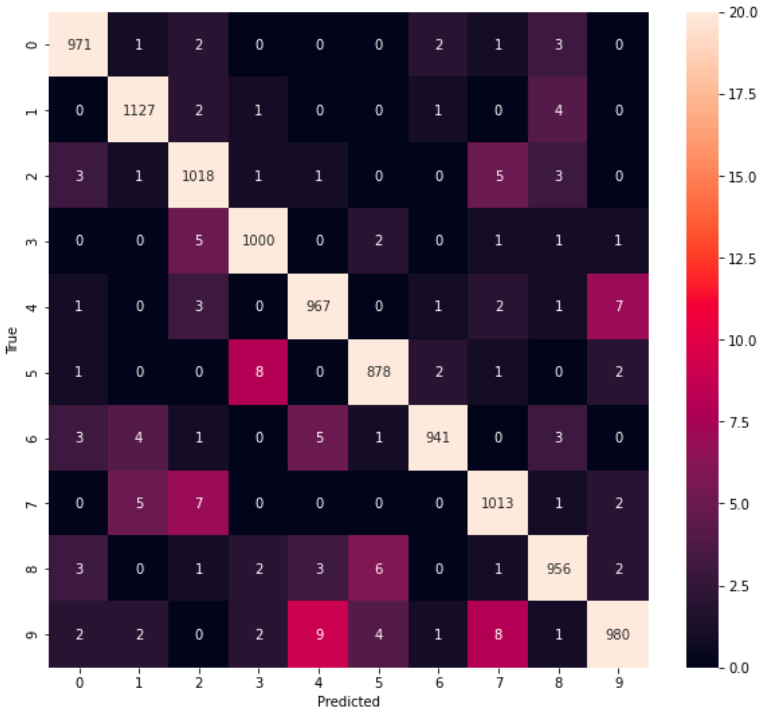


Figure 3.6: ResNet-34's confusion matrix

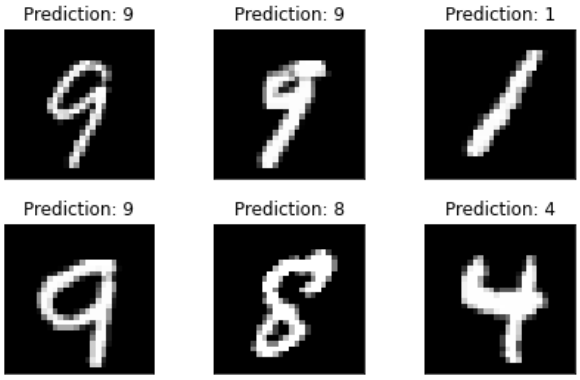


Figure 3.7: ResNet-34's predictions

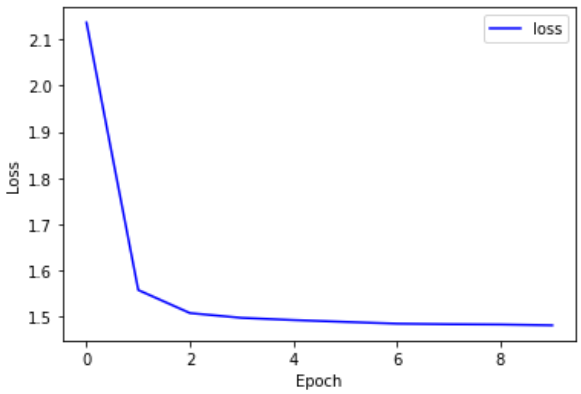


Figure 3.8: ConvNet-34's training losses.

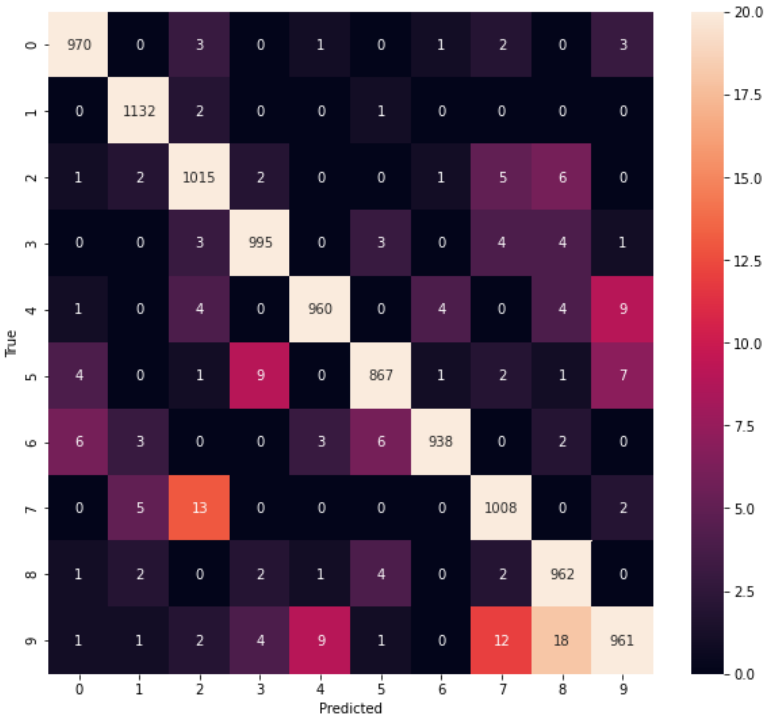


Figure 3.9: ConvNet-34's confusion matrix

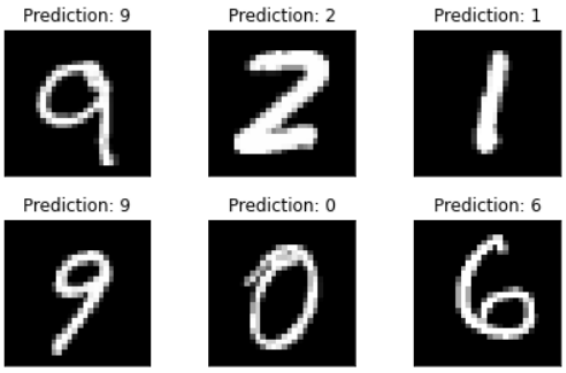


Figure 3.10: ConvNet-34's predictions

Chapter 4

Application to Image Colorization

ResNets had become the revolution of Deep Learning with a giant number of applications. The Keras' pre-trained model *Inception-ResNet-v2* [19] is the result of classifying 1.2 million images from ImageNet [10] and undoubtedly a state-of-the-art model for image classification. In 2017, Baldassarre, Morín, and Rodés-Guira [2] presented an approach that combines a deep Convolutional Neural Network with high-level features extracted from the *Inception-ResNet-v2* [19] pre-trained model. Following that idea, we had some minor modifications from the original set-ups to migrate from PyTorch to TensorFlow.

4.1 Methodology

4.1.1 Motivation

Today, colorization is usually done by hand in Adobe Photoshop where a picture alone can take up to one month to colorize. It requires extensive research and a lot of hard work behind, a face alone needs up to 20 layers of pink, green, and blue shades to get it just right [24]. What we are demonstrating here is believed to help reduce the amount of hard work for the image colorization task. Moreover, we hope that this Deep Learning approach will be applied to colorize Black and White videos as well.

4.1.2 Color space

RGB

The RGB color model is an additive color model in which red, green, and blue light are added together in various ways to reproduce a broad array of colors [33]. An image is present in all three channels, the layers not only determine color, but also brightness [24]. In gray-scale images, the value of a pixel is also determined by a $1 \times 1 \times 3$ matrix, but all 3 elements are equal to each other. A pixel with "larger" numbers means it is "brighter" or "whiter" than its neighbors.

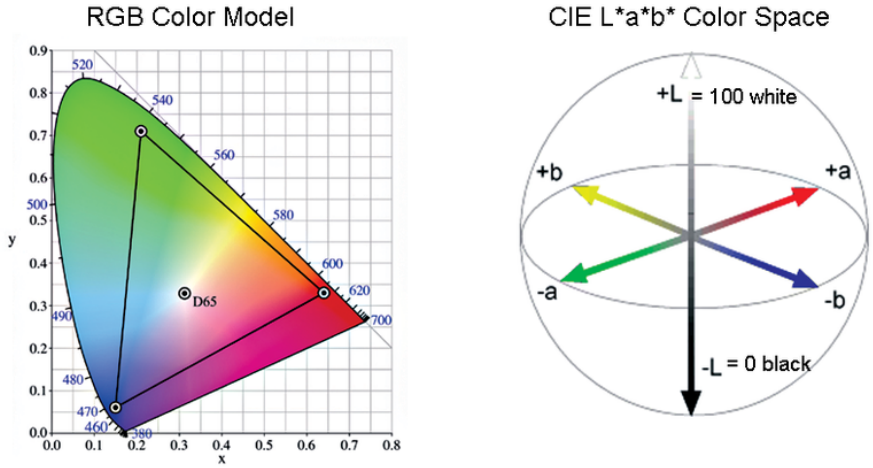


Figure 4.1: RGB and CIELAB Color Space
Source: Visscher [23]

Figure 4.2: An example of the 3 channels of an image in CIELAB color space
From left to right: **L***, **a***, **b***
Source: Wallner [24]

CIELAB

The CIELAB color space (also known as CIE L*a*b* or abbreviated simply "Lab" color space) expresses color as three values: **L*** for the lightness from black to white (0–100), **a*** from green to red (-128 – 128), and **b*** from blue to yellow (-128 – 128) [28]. To extract a gray-scale image from the original color image, we only have to extract the **L** channel and use it as a training input. We train the model to predict the **a*b*** components and then concatenate them with the gray-scale image to reproduce a colorized image.

4.1.3 Autoencoder

An autoencoder (Fig. 4.3) is an artificial neural network that is used to learn data codings in an unsupervised manner [27]. The **encoder** transforms the input data into a lower-dimensional representation (latent vector/space representation) by learning only the most important features of the data. Alongside, a reconstructing side – **decoder** generates a representation as close as possible to its original input from the encoder. An important thing to be aware of while training autoencoders is that when the latent representation is larger than the input data, they tend to memorize the input instead of learning data codings [27].

Basic Architecture

The simplest form of an autoencoder contains an input layer, an output layer, and one or more hidden layers, similar to a feedforward, non-recurrent neural network. The input layer has the same number of neurons as the output layer as its purpose is to reconstruct the inputs instead of predicting the target value for a given input. The encoder ϕ and the decoder ψ of an autoencoder can be defined as:

$$\phi : \mathcal{X} \to \mathcal{F} \tag{4.1}$$

$$\psi : \mathcal{F} \to \mathcal{X} \tag{4.2}$$

and the autoencoder goal is:

$$\phi, \psi = \underset{\phi,\psi}{\operatorname{argmin}} \|X - (\phi \circ \psi)X\|^2 \tag{4.3}$$

Normally, the encoder maps the input $\mathbf{x} \in \mathbb{R}^d = \mathcal{X}$ with the corresponding representation $\mathbf{h} \in \mathbb{R}^p = \mathcal{F}$. The image $\mathbf{h}$ is often referred as code, latent variables, or latent representation and can simply be achieved by activating a linear transformation:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{4.4}$$

Then, to retrieve the input $\mathbf{x}$ from the image $\mathbf{h}$, the decoder also use a linear transformation followed by an activation function σ':

$$\mathbf{x}' = \sigma'(\mathbf{W}'\mathbf{h} + \mathbf{b}') \tag{4.5}$$

Because the goal of an autoencoder is to approximate the input, the loss function is optimized to minimize the difference between the input $\mathbf{x}$ and the reconstructed output $\mathbf{x}'$. As mentioned before, an autoencoder is similar to a feedforward non-recurrent neural network, backward propagation can still be performed to update the entire network as usual.

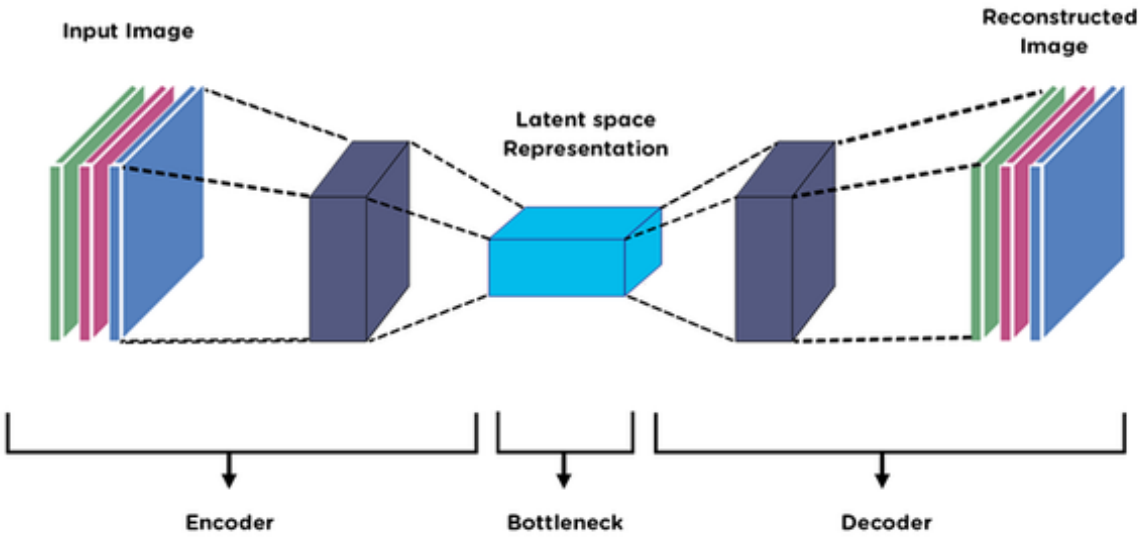


Figure 4.3: An overview of Autoencoders
Source: Birla [3]

4.1.4 Inception-ResNet-v2

The completion of *Inception-ResNet-v2* has proven how dramatically improved the introduction of residual connections leads to. It performed incredibly well, exceeding state-of-the-art performance on the ImageNet dataset [10]. Making use of that breakthrough, in 2017, Baldassarre, Morín, and Rodés-Guirao [2] has proposed an autoencoder with the help of *Inception-ResNet-v2* that solves the image colorization problem.

Architecture

Not surprisingly, *Inception-ResNet-v2* is an incredibly complicated model that contains 327 individual layers from 10 different types of modules where every single module is assigned for a specific task. As shown in Fig. 4.4, *Inception-ResNet-v2* takes an image with the size of $299 \times 299 \times 3$ as the **Input**. After having gone through the **Stem** module (detailed in Fig. 4.5), the image is convoluted into a $35 \times 35 \times 384$ space before being fetched to 5 other types of module which include 3 variations of **Inception-ResNet** modules (specified in Fig. 4.6) and 2 kinds of **Reduction** modules (specified in Fig. 4.7).

Inception-ResNet modules: There are 3 different kinds of Inception modules where *Inception-ResNet-A* module performs calculations on a 35×35 grid (Fig. 4.6a), *Inception-ResNet-B* module on a 17×17 grid (Fig. 4.6b), and *Inception-ResNet-C* module on an 8×8 grid (Fig. 4.6c). There are 5 repetitions of *Inception-ResNet-A*, 10 of *Inception-ResNet-B*, and 5 of *Inception-ResNet-C* to be used in the entire network.

For instance, in **Inception-ResNet-A** module, consecutively, there are 5 repetitions of the architecture in Fig. 4.6a where the first layer takes the output of **Stem** module with the size of $35 \times 35 \times 384$ as the input (denotes as $\mathbf{x}$). Assume $\mathcal{F}(x)$ is the output of the final convolutional layer, as the outputted space is also $35 \times 35 \times 384$, there will be a simple element-wise addition $\mathcal{F}(x) + x$ just like equation 3.2 before being activated by a ReLU function for the next layer.

Reduction modules: The *Reduction-A* module reduces the grid side from 35×35 to 17×17 (Fig. 4.7a) and acts as the input transformation from the output of *Inception-ResNet-A* to *Inception-ResNet-B*. Whereas *Reduction-B* module reduces the grid side from 17×17 to 8×8 (Fig. 4.7b) and acts as the input transformation from the output of *Inception-ResNet-B* to *Inception-ResNet-C*.

After the final activation in *Inception-ResNet-C*, the output accumulates the space of $8 \times 8 \times 1792$. It then goes through an average pooling layer and a dropout rate of 0.2 before activated by a final softmax function as take it as the global output of a 1000-dimensional vector, corresponding to 1000 given classes

of the ImageNet dataset [10]. The usage of a dropout layer follows the intuition of the human brain, if we only use a partition of our brain and still get a good result, there is no way that we will get a worse result when we use the entire brain.

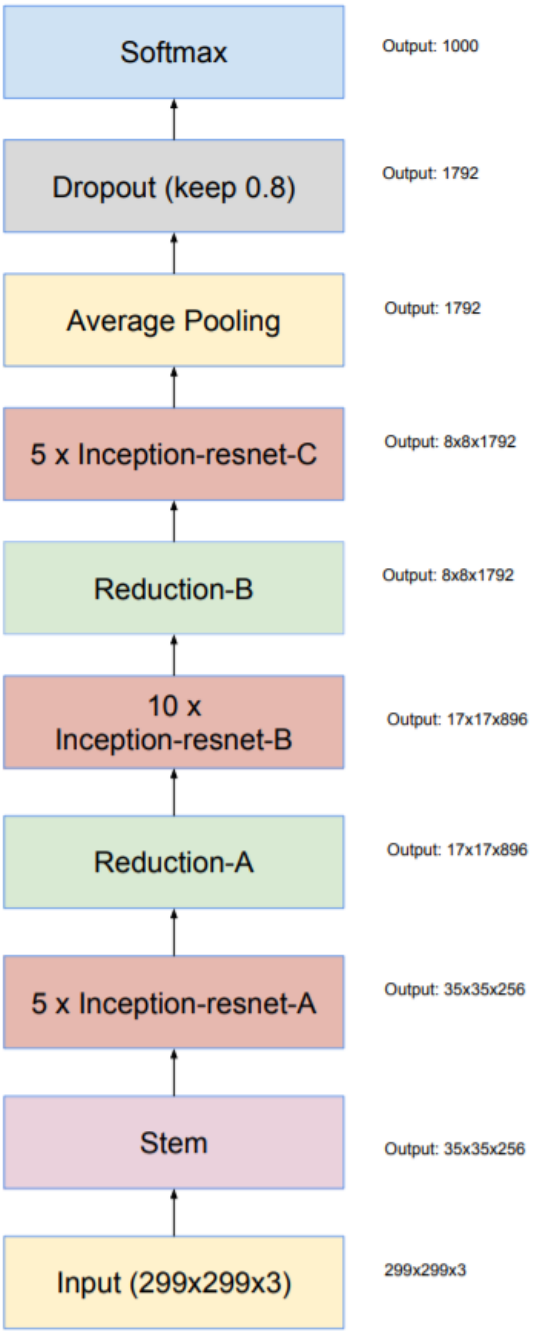


Figure 4.4: Schema for InceptionResNet-v2
Source: Szegedy, Ioffe, Vanhoucke, and Alemi [19]

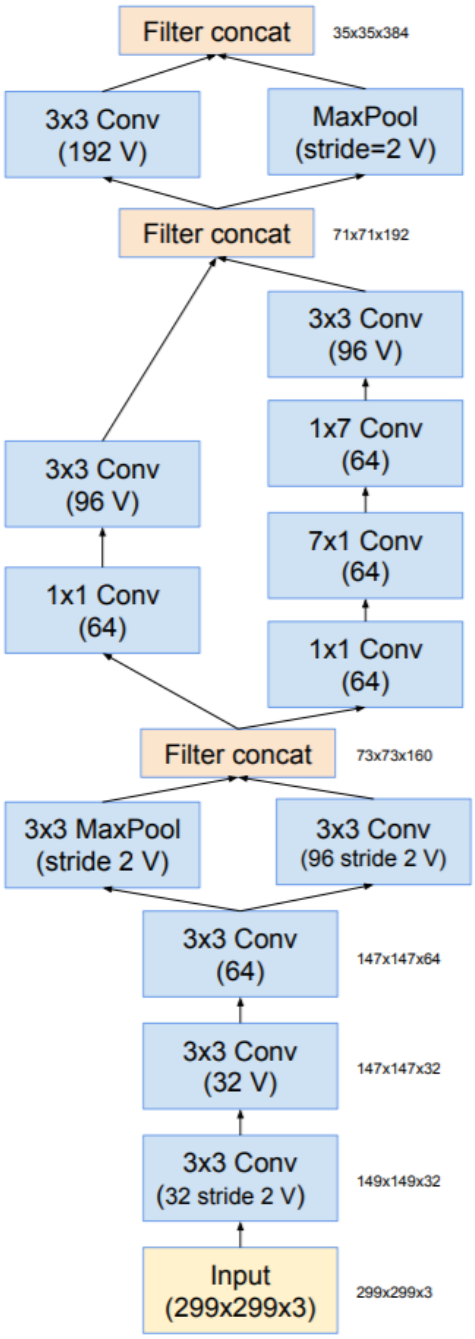
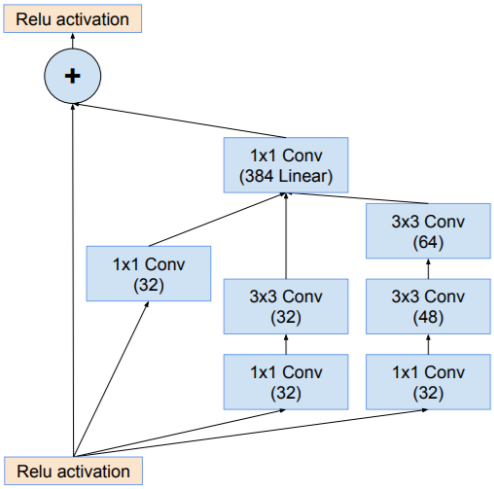


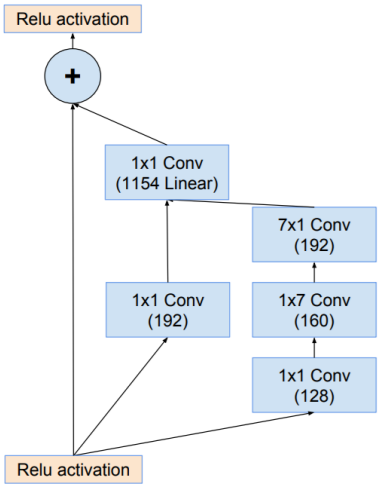
Figure 4.5: Schema for stem of Inception-ResNet-v2
Source: Szegedy, Ioffe, Vanhoucke, and Alemi [19]

Dataset & Training

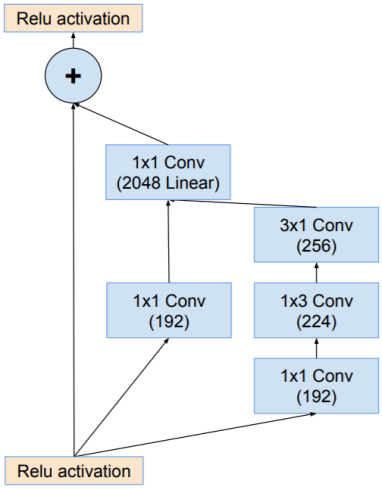
Inception-ResNet-v2 is currently distributed by Keras, it is a pre-trained model on the ImageNet dataset [10] with 1.2 million images from 1000 different classes. Keras have trained the network with TensorFlow on 20 distributed machine learning system, each on an NVidia Kepler GPU [19]. They used RMSProp with a decay of 0.9, $\epsilon = 1.0$, and a learning rate of 0.045 decayed every two epochs using an exponential rate of 0.94 [19]. Comparing to *Inception-ResNet-v1*, *Inception-ResNet-v2* has a significantly improved recognition performance as it is a costlier hybrid Inception version [19].



(a) Schema for Inception-ResNet-A module of Inception-ResNet-v2

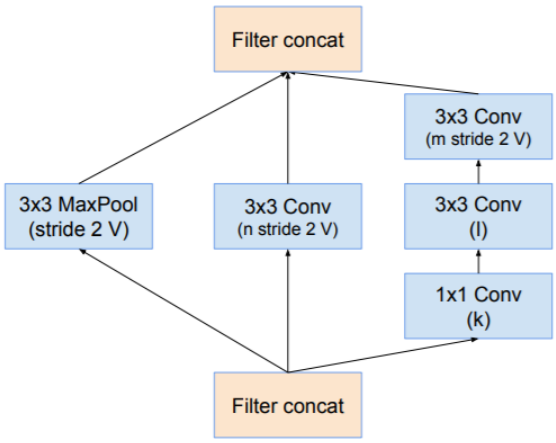


(b) Schema for Inception-ResNet-B module of Inception-ResNet-v2

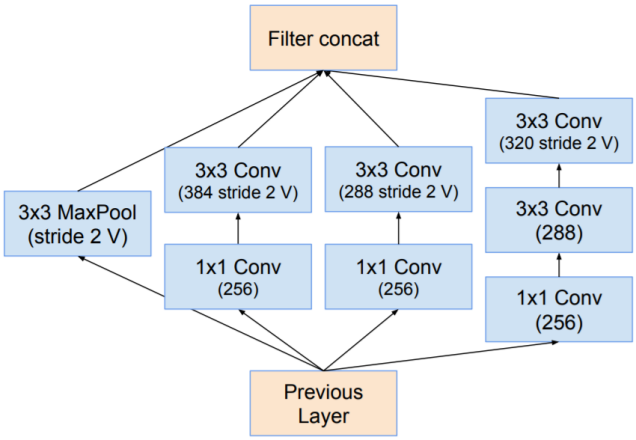


(c) Schema for Inception-ResNet-C module of Inception-ResNet-v2

Figure 4.6: Schemes for Inception-ResNet modules of Inception-ResNet-v2
Source: Szegedy, Ioffe, Vanhoucke, and Alemi [19]



(a) Schema for Reduction-A module of Inception-ResNet-v2
where $k = 256$; $l = 256$; $m = 384$; $n = 384$



(b) Schema for Reduction-B module of Inception-ResNet-v2

Figure 4.7: Schemes for Reduction modules of Inception-ResNet-v2
Source: Szegedy, Ioffe, Vanhoucke, and Alemi [19]

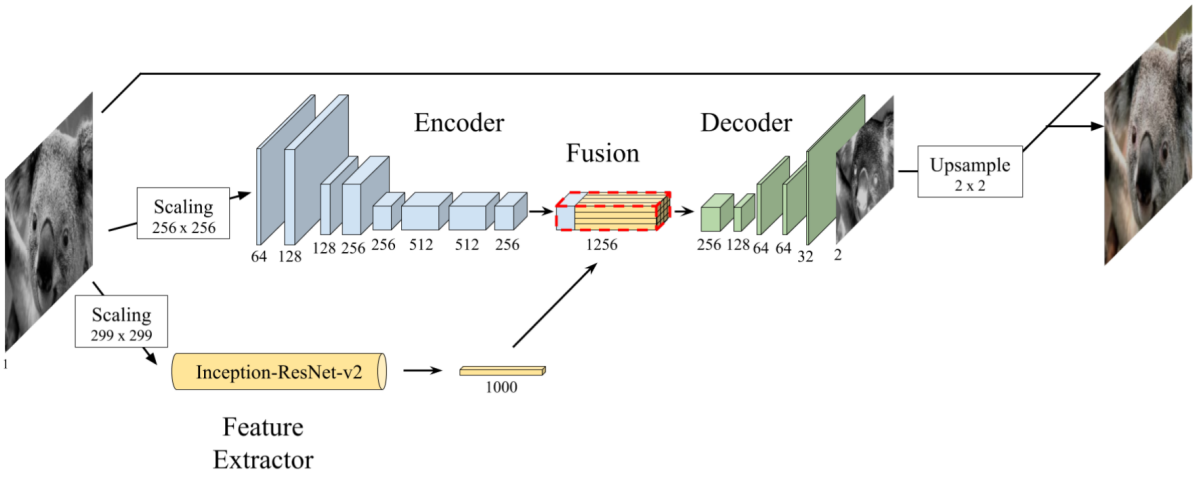


Figure 4.8: An overview of the model architecture
Modified from Baldassarre, Morín, and Rodés-Guirao [2]

Encoder Network			**Fusion Network**			**Decoder Network**		
Layer	Kernel	Stride	Layer	Kernel	Stride	Layer	Kernel	Stride
conv	$64 \times (3 \times 3)$	2×2	fusion	—	—	conv	$128 \times (3 \times 3)$	1×1
conv	$128 \times (3 \times 3)$	1×1	conv	$256 \times (1 \times 1)$	1×1	upsamp	—	—
conv	$128 \times (3 \times 3)$	2×2				conv	$64 \times (3 \times 3)$	1×1
conv	$256 \times (3 \times 3)$	1×1				conv	$64 \times (3 \times 3)$	1×1
conv	$256 \times (3 \times 3)$	2×2				upsamp	—	—
conv	$512 \times (3 \times 3)$	1×1				conv	$32 \times (3 \times 3)$	1×1
conv	$512 \times (3 \times 3)$	1×1				conv	$2 \times (3 \times 3)$	1×1
conv	$256 \times (3 \times 3)$	1×1				upsamp	—	—

Table 4.1: Model Architecture
Each convolutional layer uses a ReLu activation function,
except for the final one that employs a hyperbolic tangent (tanh) function.
The feature extraction branch has the same architecture as Inception-ResNet-v2 [19].

4.2 Model

4.2.1 Architecture

Our referred model [2] is given the luminance (**L***) component of an image, the model estimates its **a*b*** components and combines them with the input to obtain the final estimate of the colored image. Instead of training a feature extraction branch from scratch, we make use of an *Inception-ResNet-v2* [19] network (referred to as *Inception* hereafter) and retrieve an embedding of the gray-scale image from its last layer.

The network is logically divided into four main components, as shown in Fig. 4.8. The encoding and the feature extraction components obtain mid and high-level features, respectively, which are then merged in the fusion layer. Finally, the decoder uses these features to estimate the output. Table 4.1 further details the network layers.

Encoder

The Encoder processes $H \times W$ gray-scale images and outputs a $H/8 \times W/8 \times 256$ latent space feature representation. To this end, it uses 8 convolutional layers with 3×3 kernels with same padding to preserve the layer's input size. Furthermore, the first, third, and fifth layers apply a stride of 2, consequentially halving the dimension of their outputs and hence reducing the number of computations required.

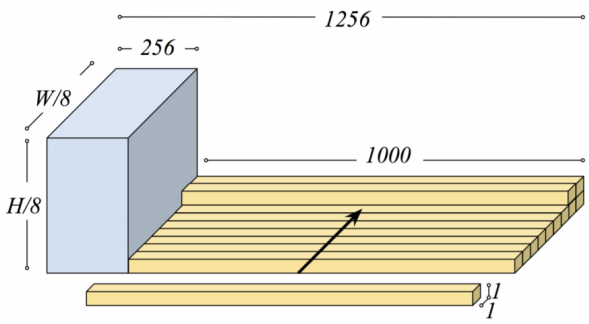


Figure 4.9: Fusing the Inception embedding with the output of the convolutional layers of the encoder
Modified from Baldassarre, Morín, and Rodés-Guirao [2]

Feature Extractor

High-level features, such as scenes (underwater, indoor,...), objects (car, dog,...) convey image information that can be used in the colorization process. To extract an image embedding we used a pre-trained *Inception-ResNet-v2* model [19]. First, we scale the input image to 299×299 and covert it to a gray-scale image by scikit-image's *rgb2gray* function [18]. Then, we use *gray2rgb* function to create a gray-scale image in RGB representation in order to satisfy Inception's dimension requirement of $299 \times 299 \times 3$. Next, we fetch the pre-processed image to the Inception network and take the final softmax function output. This results in a $1 \times 1 \times 1000$ embedding.

Fusion

The fusion layer takes the feature vector from Inception, replicates it $H/8 \times W/8$ times, and then attaches it to the feature volume outputted by the encoder along the depth axis. This approach obtains a single volume with the encoded image and the mid-level features of shape $H/8 \times W/8 \times 1256$ (Fig. 4.9). By repeating the feature vector and then concatenating it several times, it ensures that the semantic information conveyed by the feature extractor is uniformly distributed among all spatial regions of the image. Finally, 256 convolutional kernels of size 1×1 are applied to generate a feature volume of dimension $H/8 \times W/8 \times 256$.

Decoder

Finally, the decoder takes the $H/8 \times W/8 \times 256$ volume from the last layer and applies a series of convolutional and up-sampling layers in order to obtain a final layer with dimension $H/8 \times W/8 \times 2$. Here, up-sampling is used to take a nearest neighbor so that the output's height and width are twice the input's.

4.2.2 Training

We trained the autoencoder to minimize the loss function, which is a metric of the difference between the predicted output and the desired one. The model parameters are optimized by minimizing an objective loss function $\mathcal{L}$ defined by equation 4.6:

$$\mathcal{L} = MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{4.6}$$

where Mean Squared Error (***MSE***) is defined by the "mean" of "squared" difference between the i-th truth value $\boldsymbol{Y_i}$ and i-th predicted value $\boldsymbol{\hat{Y}_i}$ over $\boldsymbol{n}$ data points.

In order to minimize the loss, we calculated the Mean Squared Error between the colorized pixel in **a*b*** space and their ground truth value. While training, this loss is backpropagated to update the model parameters using Adam Optimizer with an initial learning rate of 1×10^{-3}. We also used Keras' *ReduceLROnPlateau* to reduce the previous learning rate by half but no less than 1×10^{-5}. Furthermore, we implemented a checkpoint that monitors the loss and only saves the best model regardless of fluctuations when the training process reaches the saturation point. A batch size of 20 is also configured to help reduce the computational complexity of the model. So, in an iteration, only 20 images are fetched and trained throughout the model.

4.3 Data Preparation

4.3.1 Dataset

In the scope of this thesis, we used the *Places365-Standard* data set [11], which contains 1.8 million images from 365 scene categories. Due to the limited computational power of our resources, we used the validation images in the *Small images (256 * 256)* version of the data set, which contains 36,500 images. The images in the archive have been resized to 256×256 regardless of the original aspect ratio [11]. For validating the result, we only use a portion of 100 images to test the model and try colorizing some historical images by the model.

In the meantime, we also prepared a light-weight edition dataset, suitable for training on local computers. *Landscape Pictures* dataset, shared by Arnaud Rougetet on Kaggle [17] which contains 4,319 pictures of natural landscapes from 7 research from the website Flickr:

- landscapes (900 pictures)
- landscapes mountain (900 pictures)
- landscapes desert (100 pictures)
- landscapes sea (500 pictures)
- landscapes beach (500 pictures)
- landscapes island (500 pictures)
- landscapes japan (900 pictures)

4.3.2 Pre-processing

Firstly, we filtered out gray-scale images by comparing Red, Green, and Blue channels of an RGB image. A channel is represented by a 256×256 matrix, if all the elements of the matrix equal to the two remaining channels, the image is then labeled as gray-scale and discarded to avoid miscalculations of the loss function.

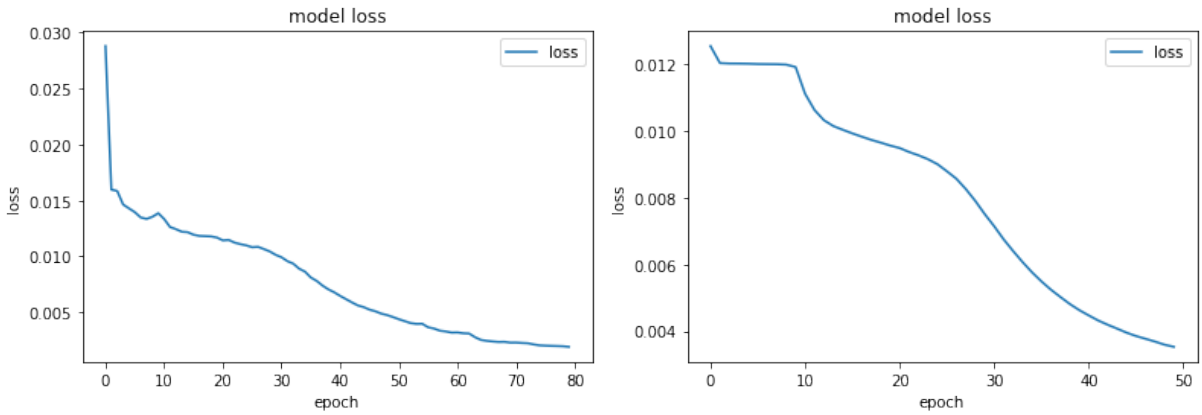
As an effort to increase the generality of the model, we augmented the training images with shearing, zooming, rotating, and flipping by applying *ImageDataGenerator* class from Keras.

To provide inputs for the model, the image is embedded differently for Encoder's input and InceptionResNet-v2's input.

- For the encoding process, the image is converted from RGB (R*G*B*) space to CIELAB (L*a*b*) space. Then, we extract the luminance component with the shape of the $256 \times 256 \times 1$ to produce a gray-scale image. The model will learn a $256 \times 256 \times 2$ matrix – green-red and blue-yellow color spectra (**a*b*** components) from the given **L***.
- For *InceptionResNet-v2*'s input, we simply convert the original image to RGB gray-scale image with the size of $299 \times 299 \times 3$. After that, the *InceptionResNet-v2* [19] extracts high-level features and outputs a $1 \times 1 \times 1000$ matrix, which will be used in the fusion layer.

To ensure the learning process, pixel values of all three image channels of the original RGB image are normalized to obtain values within the interval of $[0, 1]$ by dividing them by 255, according to their respective ranges specified in Section 4.1.2. Then, after converting to CIELAB color space, all three image components of the image will obtain values within the interval of $[-1, 1]$. By this normalization, while performing backward propagation, Gradient is actively updated because the value of the hyperbolic tan function's derivative is large in the interval of $[-1, 1]$. Finally, the predicted **a*b*** components are multiplied by 128, added with the **L*** component, and converted back to the RGB color model to reproduce a colorful image.

In terms of reducing main memory consumption, we used Keras' *flow_from_directory* function to read only 20 images per step from disk to RAM, equal to the amount of to-be fetched images as the input of the model.



(a) Landscape Pictures dataset

(b) Places365-Standard dataset

Figure 4.10: Model loss on the two datasets

4.3.3 Post-processing

In section 4.3.2, as we normalized all three image components of the **L*a*b*** image will obtain values within the interval of $[-1, 1]$, the predicted **a*b*** components are multiplied by 128 to retrieve the range of $[-128, 128]$ (as mentioned in section 4.1.2). However, to reduce the training effort, we perform an element-wise multiplication $output \times 256$ instead, as the coefficient is doubled, the number of epochs is halved. This modification is achieved by our experiments in real life, rather than trying to approximate the ground truth image, it could be better if the model is more generic to be more applicable. Henceforth, the model training process is much more time-efficient, compared to the original architecture [2].

4.4 Result

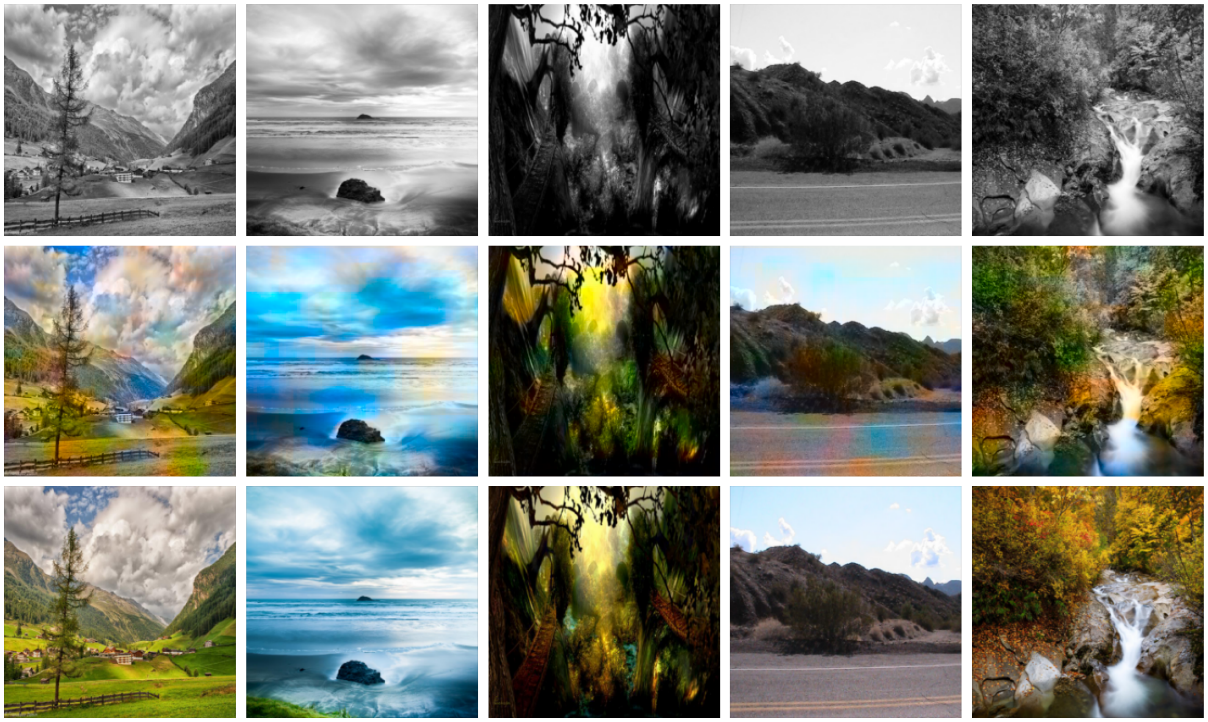
4.4.1 Landscape Pictures

We trained the *Landscape Pictures* dataset on Kaggle Kernel with the configuration of:

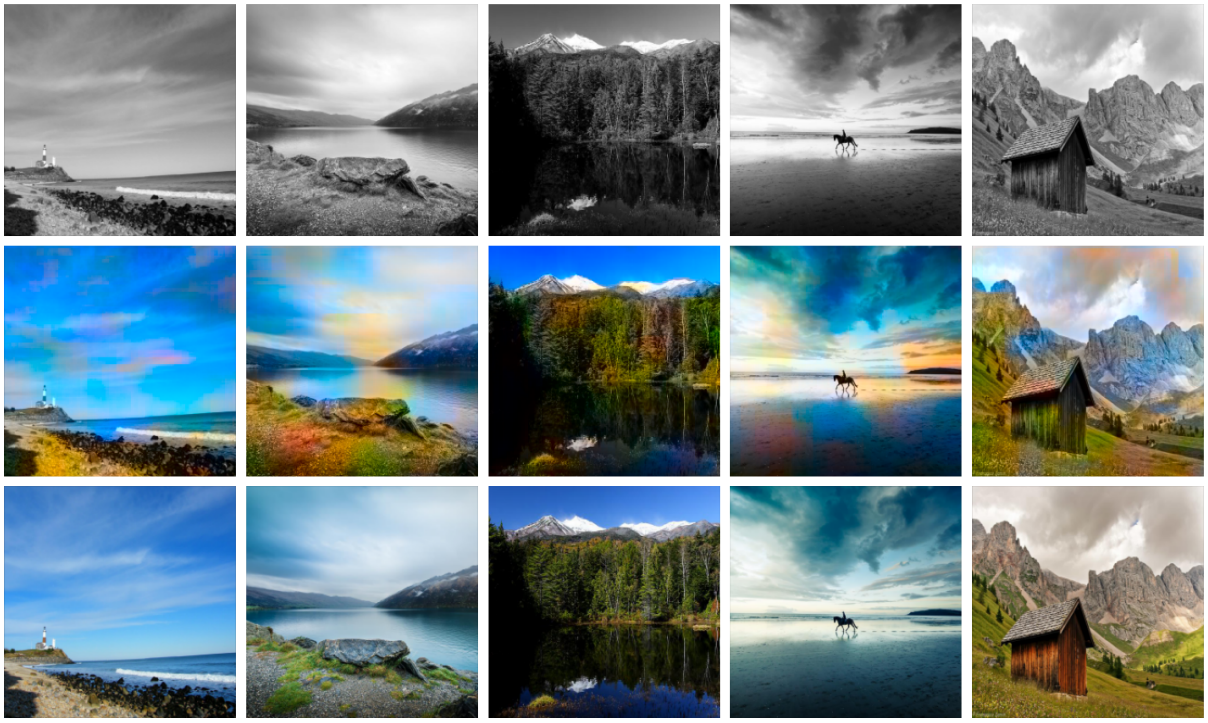
- CPU: 1x single core hyperthreaded (1 core, 2 threads) Intel(R) Xeon(R) Processors @ 2.2Ghz, 55MB Cache
- RAM: 13GB
- GPU: NVIDIA Tesla P100 PCIe 16 GB
- Disk: 20GB

After approximately 8 hours of training (365 seconds per epoch, 1.5 second per step) over 80 epochs with a batch size of 20, the model loss is shown in Fig. 4.10a, and the sample result of 10 random images is rendered in Fig. 4.11a (Image 1 – 5) and Fig. 4.11b (Image 6 – 10). The model reached a loss of 0.0019 in the 80th epoch. It can clearly be seen that our predicted image is pretty similar to the original color image. However, there are still some blue and yellow noises in some images. One more problem is that the leaves in image 5 (Fig. 4.11a, row 5, column 3) is in fact autumn leaves with a yellowish color, whereas as the model predicted (Fig. 4.11a, row 5, column 2), leaves on the tree is green so that the model paints green to the leaves as a consequence.

Due to the small size of the dataset, this result is considered to be "acceptable", and thank to this result we were more confident about the result on the *Places365-Standard* dataset that we will specify in the next section.



(a) Image 1 – 5
First row: Gray-scale image
Second row: Colorized image
Third row: Expected color image



(b) Image 6 – 10
First row: Gray-scale image
Second row: Colorized image
Third row: Expected color image

Figure 4.11: Result of training *Landscape Pictures* dataset over 80 epochs

4.4.2 Places365-Standard

We trained the *Places365-Standard* dataset [11] on Microsoft Azure's Virtual Machine [1] with the configuration of:

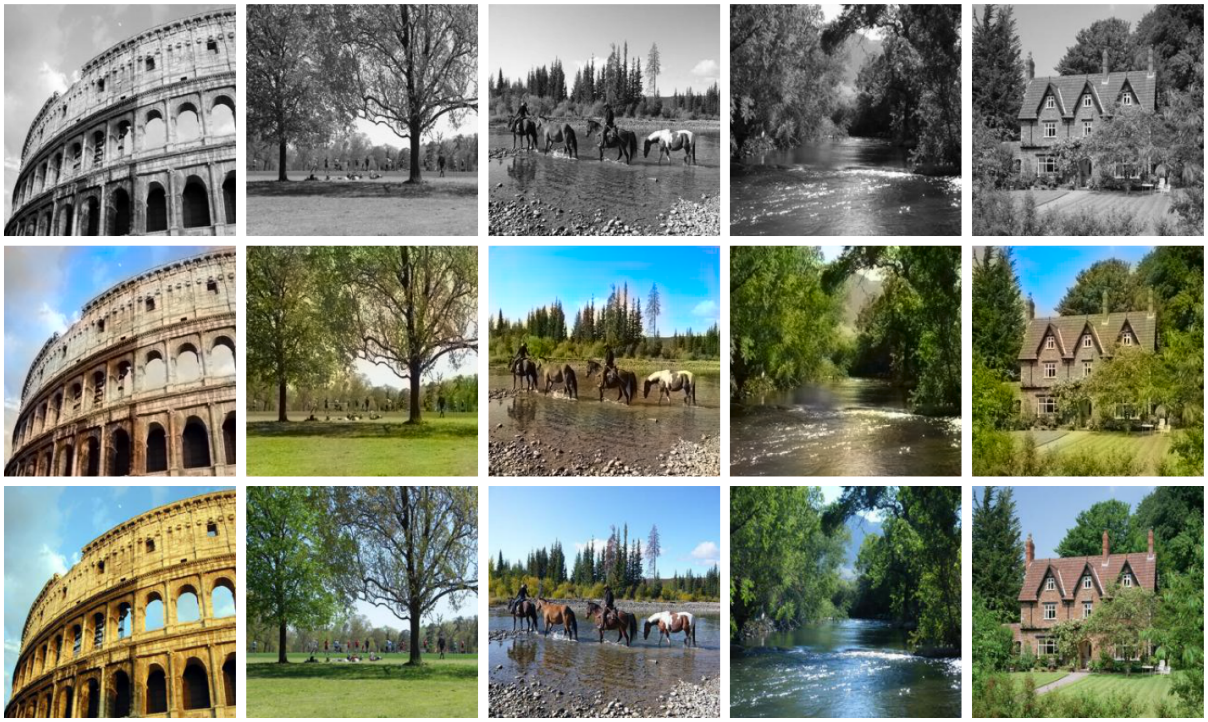
- CPU: 4x vCPU (4 core, 8 threads) Intel(R) Xeon(R) E5-2673 v3 2.4 GHz (Haswell) processors
- RAM: 16GB
- GPU: None
- Disk: 30GB
- Operating system: Linux (ubuntu 18.04)

Because our configured virtual machine does not have a GPU, so, the training speed is about 4 times slower than that of Kaggle Kernel. After approximately 10 days of training (about 5 hours per epoch, 10 seconds per step) over 50 epochs with a batch size of 20, the model loss is shown in Fig. 4.10b. We decided to reduce the number of epoch to 25 and double the output multiplication coefficient to 256 so that the model could be able to colorize similarly to the effort of 50 epochs and the coefficient of 128. The sample result of 10 random images is rendered in Fig. 4.12a (Image 1 – 5) and Fig. 4.12b (Image 6 – 10). The model reached a loss of 0.0089 in the 25th epoch and 0.0035 in the 50th epoch. However, the overfitting problem can be easily solved by reducing the multiplying coefficient of the prediction. It can clearly be seen that our predicted image is almost similar to the original color image and the color seems incredibly natural.

Actually, the image of an object captures the optical representation of that object. An image captured from a camera may be varied by many factors such as lighting, weather condition, and even the precision of the camera's sensor itself. So, a colorized image is just an item in the set of many other possible color mixtures, and what we are trying to do is to colorize an image that it "seems" natural and everybody can "feel" that it is natural. That is also the reason why we only use the loss metric but not the similarity between the colorized image and the ground truth one. We believe that there is no definition of a "natural" image but there is a "look and feel" of the human brain that decides whether an image is "natural" or not. It can be understood that we used the heuristic approach on this problem, it does not guarantee that a specific model works but it works in most cases.

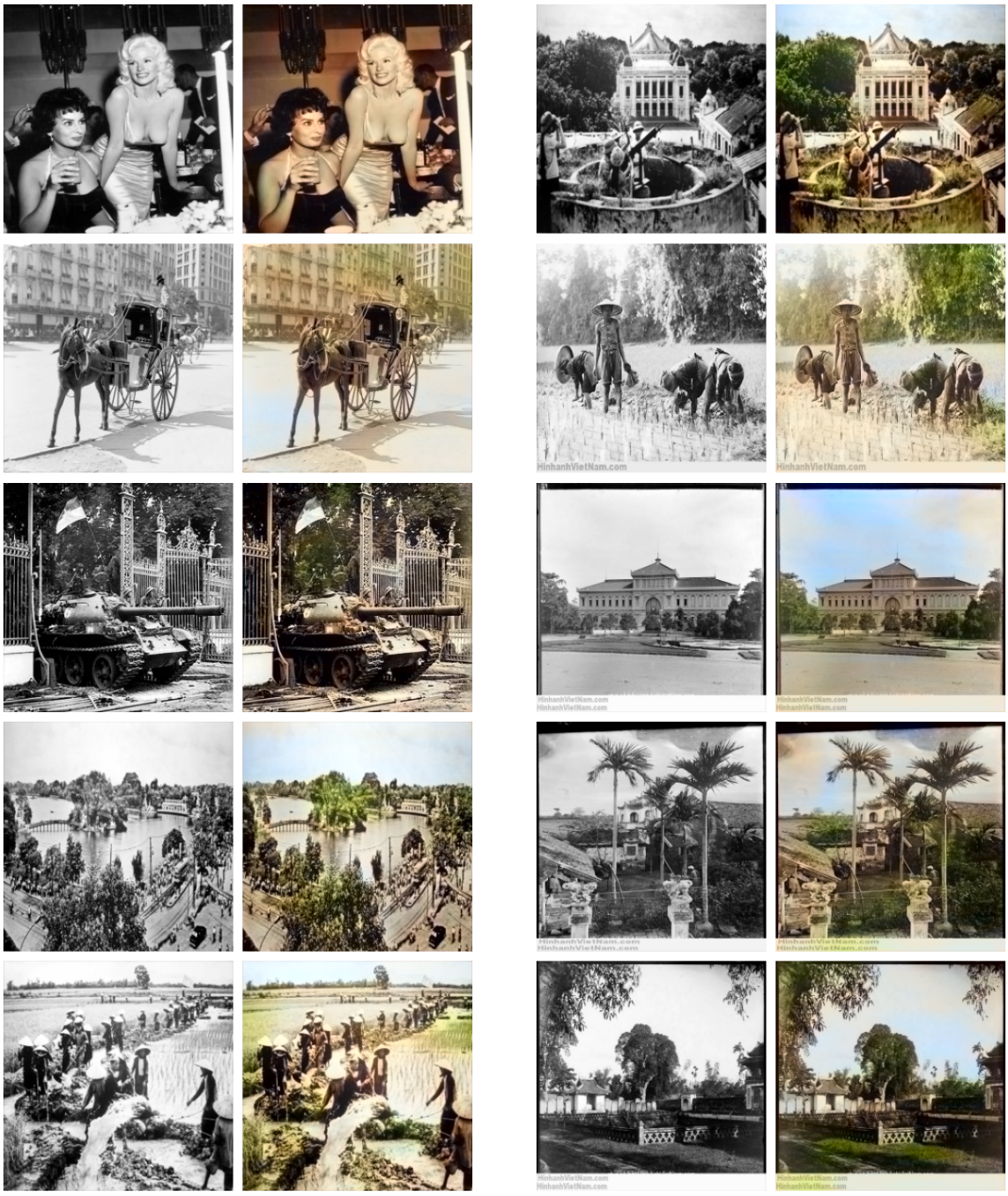
We also try our model on some famous Vietnamese historical gray-scale images retrieved from [14], [15], and some of the world's most notable images retrieved from [4]. The original one and the colorized one is illustrated in Fig. 4.13a (Image 1 – 5) and Fig. 4.13b (Image 6 – 10). The result turns out pretty great, the model does not only perform excellently on images of outdoor scenes but also product pictures with humans that look natural. Except the clothes' color might seem "incorrect" in images 5 and 7 because, in fact, a single pixel could be the representation of an array of colors, henceforth, in the context of clothes' color, it is impossible to predict precisely. Also the The Huc Bridge in image 4 is not correctly colorized because most of the bridges are not painted red, and even not everybody acknowledges that The Huc Bridge is red. However, overall, the model is generic enough because it can realize that the trees are green, the sky is blue, the cloud is white, and so on, just like the human instinct.

(a) Image 1 – 5
First row: Gray-scale image
Second row: Colorized image
Third row: Expected color image



(b) Image 6 – 10
First row: Gray-scale image
Second row: Colorized image
Third row: Expected color image

Figure 4.12: Result of training *Places365-Standard* dataset over 25 epochs



(a) Image 1 – 5
First column: Gray-scale image
Second column: Colorized image

(b) Image 6 – 10
First column: Gray-scale image
Second column: Colorized image

Figure 4.13: Restoring some famous historical images using our model

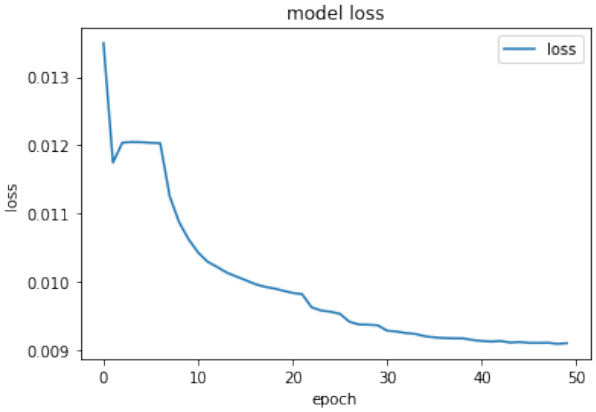


Figure 4.14: Model loss on Places365-Standard dataset

4.5 Comparison with Basic CNN

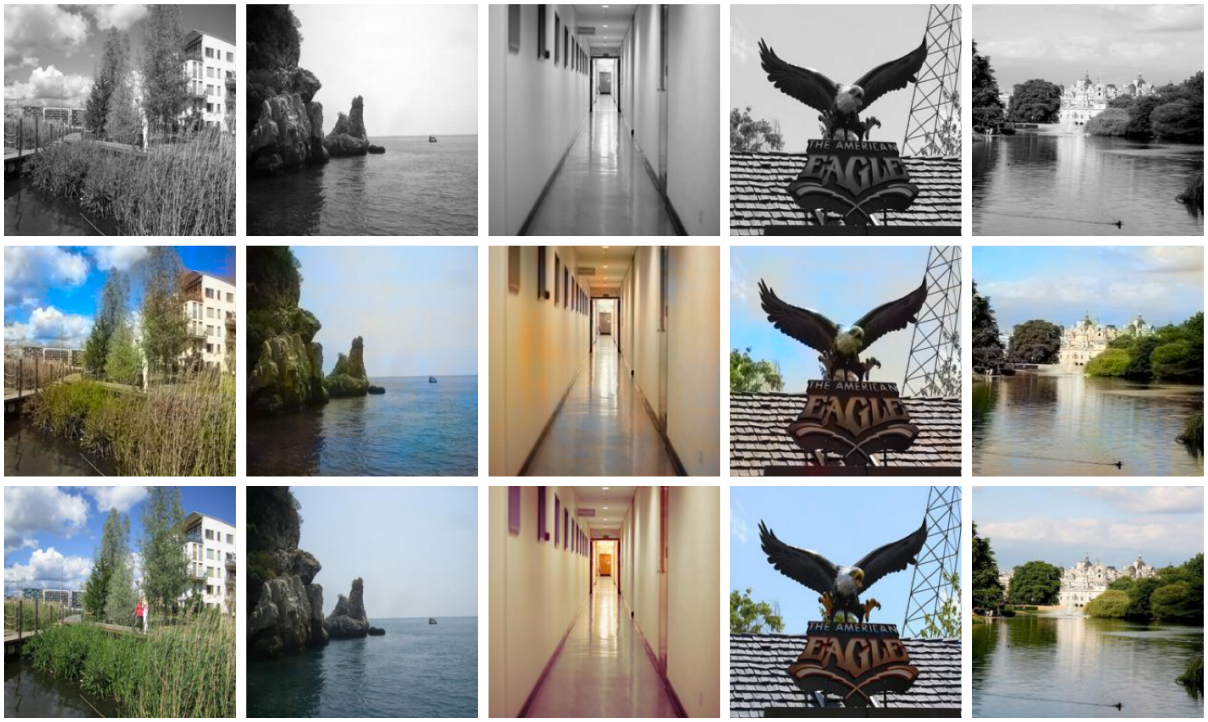
4.5.1 Model

From the original model mentioned in section 4.2.1, we simply remove the Feature Extractor and Fusion layers to create a simple Autoencoder which accumulates $H/8 \times W/8 \times 256$ latent space feature representation and immediately decodes into a final layer with dimension $H/8 \times W/8 \times 2$. We also used the same dataset and the same preprocessing as described in section 4.3.

4.5.2 Result

We trained the *Places365-Standard* dataset [11] on the same Microsoft Azure's Virtual Machine [1] mentioned in section 4.4.2. After approximately 1 week of training (about 4 hours per epoch, 5 seconds per step) over 50 epochs with a batch size of 20, the model loss is shown in Fig. 4.10b. The sample result of 10 random images is rendered in Fig. 4.15a (Image 1 – 5) and Fig. 4.15b (Image 6 – 10), the original famous historical gray-scale images and the colorized one is illustrated in Fig. 4.16a (Image 1 – 5) and Fig. 4.16b (Image 6 – 10). The model loss was saturated at the point of 0.0091 in the 50th epoch. It can clearly be seen that our predicted image is almost similar to the original color image and the color seems incredibly natural.

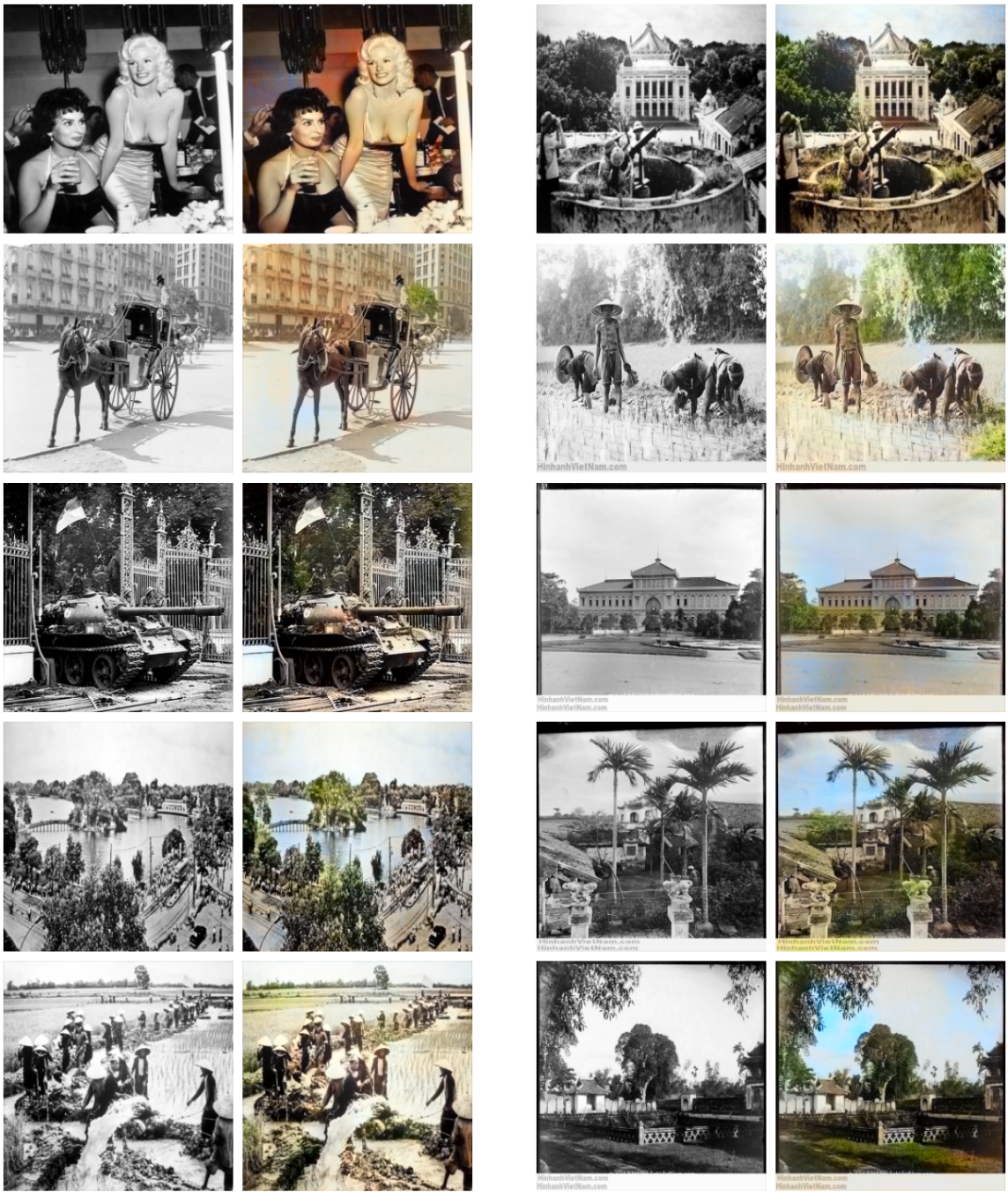
There is no significant difference between the results of our two proposed approaches. However, in the context of time efficiency, the Image Colorization model using autoencoder and *Inception-ResNet-v2* [19] (chapter 4) has achieved the same loss value in 70% the time, and half the number of epochs of the basic CNN model.



(a) Image 1 – 5
First row: Gray-scale image
Second row: Colorized image
Third row: Expected color image

(b) Image 6 – 10
First row: Gray-scale image
Second row: Colorized image
Third row: Expected color image

Figure 4.15: Result of training *Places365-Standard* dataset over 50 epochs



(a) Image 1 – 5
First column: Gray-scale image
Second column: Colorized image

(b) Image 6 – 10
First column: Gray-scale image
Second column: Colorized image

Figure 4.16: Restoring some famous historical images using our model

Chapter 5

Conclusion & Future Work

In the scope of this thesis, we reviewed related background knowledge and constructed a simple residual network that solves the image classification problem. Having made use of *Inception-ResNet-v2* [19] that have been trained on ImageNet dataset [10] of 1.2 million images, we implemented an image colorization model on Places365-Standard dataset [11] that could be used as a core platform for video colorization.

5.1 Conclusion

In terms of preliminaries, we covered the background knowledge related to our topic: from the elementary neurons of a simple neural network to a deep convolutional neural network with multiple components; how a neural network is trained forwardly and backwardly. After that, we elucidated the architecture of a residual network from the simplest building blocks with shortcut connections to the more complex **ResNet34** model. We also re-implemented a residual network for a simple image classification problem.

We have a strong belief that the application to Image Colorization will help reduce the amount of work in recovering and colorizing black-and-white images down to few seconds by rendering colorful images with the help of autoencoder and **Inception-ResNet-v2** [19]. Furthermore, our future work with the video colorization problem, which is motivated by the extensive researches of image colorization, has largely been left behind, we hope our work in the future will make a considerable contribution to the community.

5.2 Future Work

Motivation

Comparing to the image colorization problem, which has been explored quite extensively (section 1.3), but video colorizing has largely been left behind [25]. This is not surprising because of the fact that color cameras were invented before video recorder, and the cost of recording a video is an insane stack of money. However, many historical movies and documentaries are considered "masterpieces", so video colorization is a promising task that enables us to see the color of history. Video colorization could be taken as a direct extension of image colorization, where we capture a frame as an image and treat it as an image colorization task. But obviously, "temporal coherence", or simply known as coloring successive frames consistently is not guaranteed, as it would consider each frame as a separate task, ignoring the contextual connections between frames. This would result in flickering colors, and altogether unusable results [25].

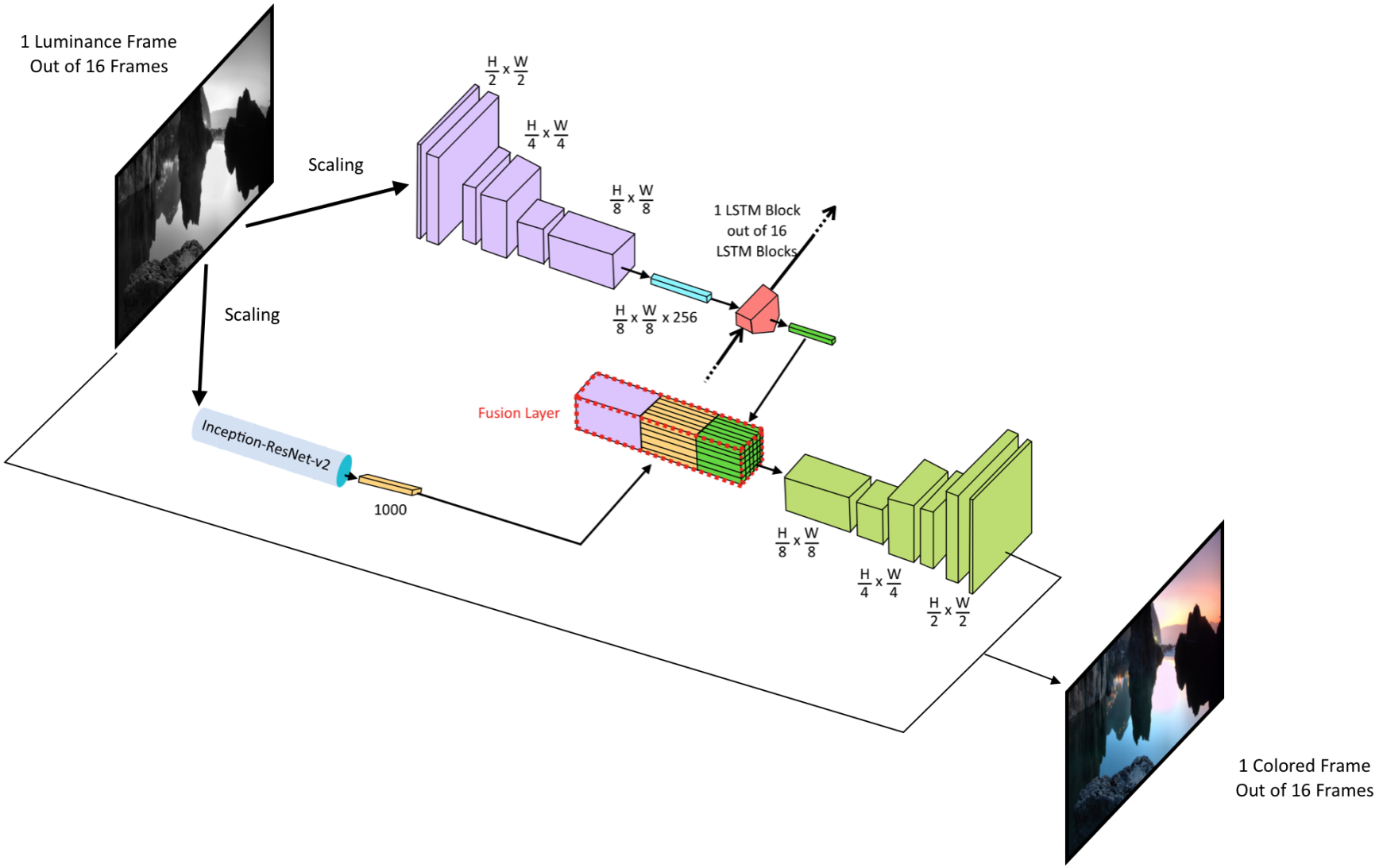


Figure 5.1: An overview of the video colorization model
Source: Wijesinghe [25]

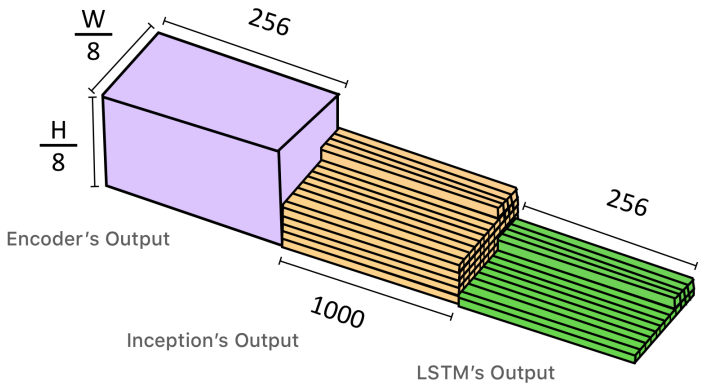


Figure 5.2: Fusion layer of the video colorization model
Source: Wijesinghe [25]

Architecture

Motivated from the model architecture in section 4.2.1, Thejan Wijesinghe [25] proposed **FlowChroma** usings autoencoders, convolutional neural networks (CNN), and long short-term memory (LSTM). The usage of LSTM will first connect contextual connections between frames because the color of a frame is now dependent on the color distribution of the predecessor frames and then result in a consistent output. The network architecture is combined from 5 basic parts and illustrated in Fig. 5.1:

- Time distributed CNN encoder
- Time distributed CNN decoder
- Fusion layer
- High-level feature extractor (*Inception-ResNet-v2*)
- LSTM to extract temporal features within frames

FlowChroma shares the same encoder layers, decoder layers, input, and output size as the model architecture in section 4.2.1. There are two main differences in this architecture is firstly the input and output of the LSTM block, and secondly the connection between LSTM blocks:

- After the final encoder layer, the model obtains a global average of the encoder output, which results in an $H/8 \times W/8 \times 256$ input for the LSTM. The LSTM output is then repeated and concatenated with the fusion layer (see Fig. 5.2), convoluted to an $H/8 \times W/8 \times 256$ vector space as the input for decoder layers.
- An LSTM block connects recurrently with 15 other LSTM blocks to create a Recurrent Convolutional Neural Network (R-CNN). Thank to this architecture, the consistency between the predecessor frame and the successor frame is increased.

The simple R-CNN using CNN and LSTM in Fig. 5.3 may help a clearer view of the model architecture.

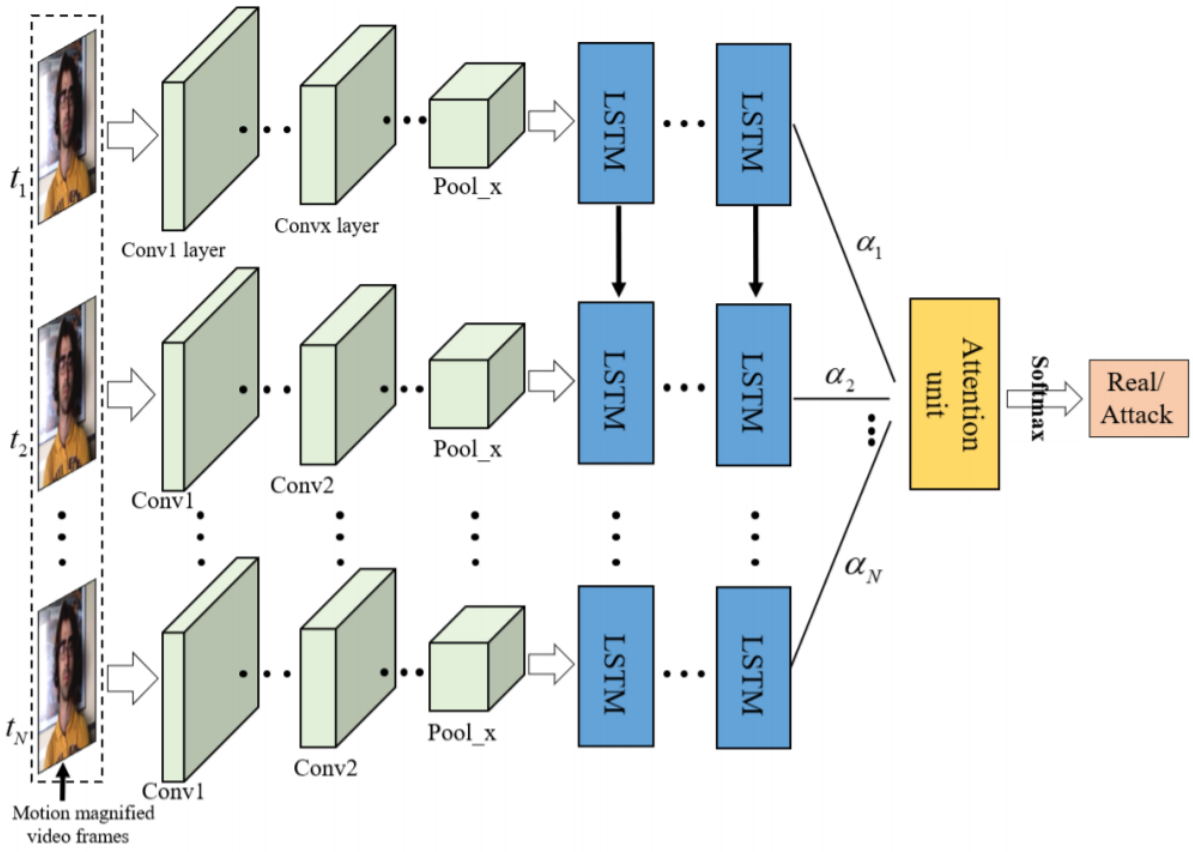


Figure 5.3: The flowchart of a simple CNN-LSTM network
Modified from: Tu et al. [21]

Bibliography

[1] Microsoft Azure. Dv3 and dsv3-series - azure virtual machines — Microsoft Docs. https://docs.microsoft.com/en-us/azure/virtual-machines/dv3-dsv3-series, 2020. [Online; accessed 18-November-2020].

[2] Federico Baldassarre, Diego González Morín, and Lucas Rodés-Guirao. Deep koalarization: Image colorization using cnns and inception-resnet-v2. *arXiv preprint arXiv:1712.03400*, 2017.

[3] Deepak Birla. Basics of autoencoders — Medium. https://medium.com/@birla.deepak26/autoencoders-76bb49ae6a8f, 2019. [Online; accessed 25-October-2020].

[4] deMilked. 21 new colorized historic photos — deMilked. demilked.com/historic-photos-colorization/, 2020. [Online; accessed 09-December-2020].

[5] Palo Alto CA USA spring 2020" "Dept. of Comp. Sci., Stanford University. *"Convolutional Neural Networks for Visual Recognition"*, "2020 (accessed November 29, 2020)".

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[8] Akinori Hidaka and Takio Kurita. Consecutive dimensionality reduction by canonical correlation analysis for visualization of convolutional neural networks. volume 2017, pages 160–167, 12 2017.

[9] Satoshi Iizuka, Edgar Simo-Serra, and Hiroshi Ishikawa. Let there be color! joint end-to-end learning of global and local image priors for automatic image colorization with simultaneous classification. *ACM Transactions on Graphics (ToG)*, 35(4):1–11, 2016.

[10] ImageNet. About imagenet — ImageNet. http://image-net.org/about-overview, 2020. [Online; accessed 09-December-2020].

[11] Aditya Khosla. A large-scale database for scene understanding — Places2. http://places2.csail.mit.edu/download.html. [Online; accessed 25-October-2020].

[12] Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. Learning representations for automatic colorization. In *European conference on computer vision*, pages 577–593. Springer, 2016.

[13] Junsoo Lee, Eungyeup Kim, Yunsung Lee, Dongjun Kim, Jaehyuk Chang, and Jaegul Choo. Reference-based sketch image colorization using augmented-self reference and dense semantic correspondence. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 5801–5810, 2020.

[14] motbahai. Nhung hinh anh lich su dac biet ve ha noi giai doan 1954 – 1975 — motbahai. http://motbahai.com/nhung-hinh-anh-lich-su-dac-biet-ve-ha-noi-giai-doan-1954-1975/, 2020. [Online; accessed 09-December-2020].

[15] Hinh Anh Viet Nam. Hinh anh xua nhat viet nam: Thap nien 1890 — Hinh Anh Viet Nam. https://hinhanhvietnam.com/nhung-hinh-anh-xua-nhat-viet-nam-nhung-nam-1890/, 2020. [Online; accessed 09-December-2020].

[16] Gokhan Ozbulak. Image colorization by capsule networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 0–0, 2019.

[17] Arnaud Rougetet. Landscape pictures — Kaggle. https://www.kaggle.com/arnaud58/landscape-pictures, 2020. [Online; accessed 18-November-2020].

[18] scikit image. scikit-image: Image processing in python — scikit-image. https://scikit-image.org/, 2020. [Online; accessed 25-October-2020].

[19] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alex Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. *arXiv preprint arXiv:1602.07261*, 2016.

[20] Vu Huu Tiep. *Machine Learning co ban*. 2018.

[21] Xiaoguang Tu, Hengsheng Zhang, Mei Xie, Yao Luo, Yuefei Zhang, and Zheng Ma. Enhance the motion cues for face anti-spoofing using cnn-lstm architecture. *arXiv preprint arXiv:1901.05635*, 2019.

[22] Nguyen Thanh Tuan. *Deep Learning co ban*. 2019.

[23] Marty Visscher. Imaging skin: Past, present and future perspectives. *Giornale italiano di dermatologia e venereologia : organo ufficiale, Società italiana di dermatologia e sifilografia*, 145:11–27, 02 2010.

[24] Emil Wallner. How to colorize black & white photos with just 100 lines of neural network code — Medium. https://medium.com/@emilwallner/colorize-b-w-photos-with-a-100-line-neural-network-53d9b4449f8d, 2017. [Online; accessed 25-October-2020].

[25] Thejan Wijesinghe. Flow chroma: A deep learning based automated video colorization framework — GitHub. https://github.com/ThejanW/FlowChroma, 2020. [Online; accessed 07-December-2020].

[26] Wikipedia contributors. Artificial neural network — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Artificial_neural_network&oldid=984752210, 2020. [Online; accessed 25-October-2020].

[27] Wikipedia contributors. Autoencoder — Wikipedia. https://en.wikipedia.org/wiki/Autoencoder, 2020. [Online; accessed 25-October-2020].

[28] Wikipedia contributors. Cielab color space — Wikipedia. https://en.wikipedia.org/wiki/CIELAB_color_space, 2020. [Online; accessed 25-October-2020].

[29] Wikipedia contributors. Convolutional neural network — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Convolutional_neural_network&oldid=984626156, 2020. [Online; accessed 1-November-2020].

[30] Wikipedia contributors. Feedforward neural network — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Feedforward_neural_network&oldid=984789558, 2020. [Online; accessed 25-October-2020].

[31] Wikipedia contributors. Gradient descent — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Gradient_descent&oldid=986763747, 2020. [Online; accessed 5-November-2020].

[32] Wikipedia contributors. Machine learning — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Machine_learning&oldid=987289349, 2020. [Online; accessed 6-November-2020].

[33] Wikipedia contributors. Rgb color model — Wikipedia. https://en.wikipedia.org/wiki/RGB_color_model, 2020. [Online; accessed 25-October-2020].

[34] Wikipedia contributors. Sigmoid function — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Sigmoid_function&oldid=980338151, 2020. [Online; accessed 9-November-2020].

[35] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. 2020. https://d2l.ai.

[36] Richard Zhang, Phillip Isola, and Alexei A Efros. Colorful image colorization. In *European conference on computer vision*, pages 649–666. Springer, 2016.

[37] Richard Zhang, Jun-Yan Zhu, Phillip Isola, Xinyang Geng, Angela S Lin, Tianhe Yu, and Alexei A Efros. Real-time user-guided image colorization with learned deep priors. *arXiv preprint arXiv:1705.02999*, 2017.

Appendix A

MNIST Digits Classifier Code

```
1
2  import os
3  import time
4
5  import numpy as np
6  import pandas as pd
7
8  import torch
9  import torch.nn as nn
10 import torch.nn.functional as F
11 from torch.utils.data import DataLoader
12
13 from torchvision import datasets
14 from torchvision import transforms
15
16 import matplotlib.pyplot as plt
17 from PIL import Image
18
19 from sklearn.metrics import confusion_matrix
20
21 ##########################
22 ### SETTINGS
23 ##########################
24
25 # Hyperparameters
26 RANDOM_SEED = 1
27 LEARNING_RATE = 0.01
28 BATCH_SIZE = 128
29 NUM_EPOCHS = 10
30
31 # Architecture
32 NUM_FEATURES = 28*28
33 NUM_CLASSES = 10
34
35 # Other
36 DEVICE = "cuda" if torch.cuda.is_available() else 'cpu'
37 GRAYSCALE = True
38
39 train_dataset = datasets.MNIST(root='data',
40 train=True,
41 transform=transforms.ToTensor(),
42 download=True)
43
44 test_dataset = datasets.MNIST(root='data',
45 train=False,
46 transform=transforms.ToTensor())
47
48 train_loader = DataLoader(dataset=train_dataset,
49 batch_size=BATCH_SIZE,
50 shuffle=True)
51
52 test_loader = DataLoader(dataset=test_dataset,
53 batch_size=BATCH_SIZE,
54 shuffle=False)
55
56 device = torch.device(DEVICE)
57 torch.manual_seed(0)
```

A.1 ResNet-34

```
##########################
### MODEL
##########################


def conv3x3(in_planes, out_planes, stride=1):
"""3x3 convolution with padding"""
return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
padding=1, bias=False)


class BasicBlock(nn.Module):
expansion = 1

def __init__(self, inplanes, planes, stride=1, downsample=None):
super(BasicBlock, self).__init__()
self.conv1 = conv3x3(inplanes, planes, stride)
self.bn1 = nn.BatchNorm2d(planes)
self.relu = nn.ReLU(inplace=True)
self.conv2 = conv3x3(planes, planes)
self.bn2 = nn.BatchNorm2d(planes)
self.downsample = downsample
self.stride = stride

def forward(self, x):
residual = x

out = self.conv1(x)
out = self.bn1(out)
out = self.relu(out)

out = self.conv2(out)
out = self.bn2(out)

if self.downsample is not None:
residual = self.downsample(x)

out += residual
out = self.relu(out)

return out

class ResNet(nn.Module):

def __init__(self, block, layers, num_classes, grayscale):
self.inplanes = 64
if grayscale:
in_dim = 1
else:
in_dim = 3
super(ResNet, self).__init__()
self.conv1 = nn.Conv2d(in_dim, 64, kernel_size=7, stride=2, padding=3,
bias=False)
self.bn1 = nn.BatchNorm2d(64)
self.relu = nn.ReLU(inplace=True)
self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
self.layer1 = self._make_layer(block, 64, layers[0])
self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
self.avgpool = nn.AvgPool2d(7, stride=1)
self.fc = nn.Linear(512 * block.expansion, num_classes)

for m in self.modules():
if isinstance(m, nn.Conv2d):
n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
m.weight.data.normal_(0, (2. / n)**.5)
elif isinstance(m, nn.BatchNorm2d):
m.weight.data.fill_(1)
m.bias.data.zero_()

def _make_layer(self, block, planes, blocks, stride=1):
downsample = None
```

```
if stride != 1 or self.inplanes != planes * block.expansion:
downsample = nn.Sequential(
nn.Conv2d(self.inplanes, planes * block.expansion,
kernel_size=1, stride=stride, bias=False),
nn.BatchNorm2d(planes * block.expansion),
)

layers = []
layers.append(block(self.inplanes, planes, stride, downsample))
self.inplanes = planes * block.expansion
for i in range(1, blocks):
layers.append(block(self.inplanes, planes))

return nn.Sequential(*layers)

def forward(self, x):
x = self.conv1(x)
x = self.bn1(x)
x = self.relu(x)
x = self.maxpool(x)

x = self.layer1(x)
x = self.layer2(x)
x = self.layer3(x)
x = self.layer4(x)
# because MNIST is already 1x1 here:
# disable avg pooling
#x = self.avgpool(x)

x = x.view(x.size(0), -1)
logits = self.fc(x)
probas = F.softmax(logits, dim=1)
return logits, probas



def resnet34(num_classes):
"""Constructs a ResNet-34 model."""
model = ResNet(block=BasicBlock,
layers=[3, 4, 6, 3],
num_classes=NUM_CLASSES,
grayscale=GRAYSCALE)
return model

torch.manual_seed(RANDOM_SEED)
model = resnet34(NUM_CLASSES)
model.to(device)

optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)

train_losses=[]
start_time = time.time()
for epoch in range(NUM_EPOCHS):

running_loss=0.0

model.train()
for batch, (images, labels) in enumerate(train_loader):
logits, probas = model(images.to(device))
loss = F.cross_entropy(probas, labels.to(device))

optimizer.zero_grad()
loss.backward()

running_loss+=loss.item()

optimizer.step()
if not batch % 100:
print ('Epoch: %03d/%03d | Batch %04d/%04d | Cost: %.4f'
%(epoch+1, NUM_EPOCHS, batch,
len(train_loader), loss))

print('Time elapsed: %d s' % (time.time() - start_time))
train_losses.append(running_loss/len(train_loader))

print('Total Training Time: %d s' % (time.time() - start_time))
```

A.2 ConvNet-34

```
1
2 def conv3x3(in_planes, out_planes, stride=1):
3 """3x3 convolution with padding"""
4 return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
5 padding=1, bias=False)
6
7
8 class BasicBlock(nn.Module):
9 expansion = 1
10
11 def __init__(self, inplanes, planes, stride=1, downsample=None):
12 super(BasicBlock, self).__init__()
13 self.conv1 = conv3x3(inplanes, planes, stride)
14 self.bn1 = nn.BatchNorm2d(planes)
15 self.relu = nn.ReLU(inplace=True)
16 self.conv2 = conv3x3(planes, planes)
17 self.bn2 = nn.BatchNorm2d(planes)
18 self.downsample = downsample
19 self.stride = stride
20
21 def forward(self, x):
22 out = self.conv1(x)
23 out = self.bn1(out)
24 out = self.relu(out)
25
26 out = self.conv2(out)
27 out = self.bn2(out)
28
29 out = self.relu(out)
30
31 return out
32
33 class ConvNet(nn.Module):
34
35 def __init__(self, block, layers, num_classes, grayscale):
36 self.inplanes = 64
37 if grayscale:
38 in_dim = 1
39 else:
40 in_dim = 3
41 super(ConvNet, self).__init__()
42 self.conv1 = nn.Conv2d(in_dim, 64, kernel_size=7, stride=2, padding=3,
43 bias=False)
44 self.bn1 = nn.BatchNorm2d(64)
45 self.relu = nn.ReLU(inplace=True)
46 self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
47 self.layer1 = self._make_layer(block, 64, layers[0])
48 self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
49 self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
50 self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
51 self.avgpool = nn.AvgPool2d(7, stride=1)
52 self.fc = nn.Linear(512 * block.expansion, num_classes)
53
54 for m in self.modules():
55 if isinstance(m, nn.Conv2d):
56 n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
57 m.weight.data.normal_(0, (2. / n)**.5)
58 elif isinstance(m, nn.BatchNorm2d):
59 m.weight.data.fill_(1)
60 m.bias.data.zero_()
61
62 def _make_layer(self, block, planes, blocks, stride=1):
63 downsample = None
64 if stride != 1 or self.inplanes != planes * block.expansion:
65 downsample = nn.Sequential(
66 nn.Conv2d(self.inplanes, planes * block.expansion,
67 kernel_size=1, stride=stride, bias=False),
68 nn.BatchNorm2d(planes * block.expansion),
69 )
70
71 layers = []
72 layers.append(block(self.inplanes, planes, stride, downsample))
73 self.inplanes = planes * block.expansion
74 for i in range(1, blocks):
```

```
layers.append(block(self.inplanes, planes))

return nn.Sequential(*layers)

def forward(self, x):
x = self.conv1(x)
x = self.bn1(x)
x = self.relu(x)
x = self.maxpool(x)

x = self.layer1(x)
x = self.layer2(x)
x = self.layer3(x)
x = self.layer4(x)
# because MNIST is already 1x1 here:
# disable avg pooling
#x = self.avgpool(x)

x = x.view(x.size(0), -1)
logits = self.fc(x)
probas = F.softmax(logits, dim=1)
return logits, probas



def convnet34(num_classes):
"""Constructs a ConvNet-34 model."""
model = ConvNet(block=BasicBlock,
layers=[3, 4, 6, 3],
num_classes=NUM_CLASSES,
grayscale=GRAYSCALE)
return model

torch.manual_seed(RANDOM_SEED)
model = convnet34(NUM_CLASSES)
model.to(DEVICE)

optimizer = torch.optim.SGD(model.parameters(), lr=LEARNING_RATE)

train_losses=[]
start_time = time.time()
for epoch in range(NUM_EPOCHS):

running_loss = 0.0

model.train()
for batch, (images, labels) in enumerate(train_loader):
logits, probas = model(images.to(device))
loss = F.cross_entropy(probas, labels.to(device))

optimizer.zero_grad()
loss.backward()

running_loss += loss.item()

optimizer.step()
if not batch % 100:
print ('Epoch: %03d/%03d | Batch %04d/%04d | Cost: %.4f'
%(epoch+1, NUM_EPOCHS, batch,
len(train_loader), loss))

print('Time elapsed: %d s' % (time.time() - start_time))
train_losses.append(running_loss/len(train_loader))

print('Total Training Time: %d' % (time.time() - start_time))
```

Appendix B

Image Colorization Code

B.1 Resnet

```
import numpy as np
import pandas as pd
from skimage.transform import resize
from skimage.color import rgb2gray, gray2rgb, rgb2lab, lab2rgb
from keras.applications.inception_resnet_v2 import InceptionResNetV2, preprocess_input
from keras.models import Model, load_model, Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Input, Dense, UpSampling2D, RepeatVector, Reshape
from keras.layers.convolutional import Conv2D, Conv2DTranspose
from keras.layers.merge import concatenate
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau

inception = InceptionResNetV2(weights=None, include_top=True)
inception.load_weights('weight/inception_resnet_v2_weights_tf_dim_ordering_tf_kernel.h5')


def Colorize():
    embed_input = Input(shape=(1000,))

    # Encoder
    encoder_input = Input(shape=(256, 256, 1,))
    encoder_output = Conv2D(64, (3, 3), activation='relu', padding='same',
                            strides=[2, 2])(encoder_input)
    encoder_output = Conv2D(128, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)
    encoder_output = Conv2D(128, (3, 3), activation='relu', padding='same',
                            strides=[2, 2])(encoder_output)
    encoder_output = Conv2D(256, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)
    encoder_output = Conv2D(256, (3, 3), activation='relu', padding='same',
                            strides=[2, 2])(encoder_output)
    encoder_output = Conv2D(512, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)
    encoder_output = Conv2D(512, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)
    encoder_output = Conv2D(256, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)

    # Fusion
    fusion_output = RepeatVector(32 * 32)(embed_input)
    fusion_output = Reshape(([32, 32, 1000]))(fusion_output)
    fusion_output = concatenate([encoder_output, fusion_output], axis=3)
    fusion_output = Conv2D(256, (1, 1), activation='relu', padding='same')(fusion_output)

    # Decoder
    decoder_output = Conv2D(128, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(fusion_output)
    decoder_output = UpSampling2D((2, 2))(decoder_output)
    decoder_output = Conv2D(64, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(decoder_output)
    decoder_output = Conv2D(64, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(decoder_output)
    decoder_output = UpSampling2D((2, 2))(decoder_output)
    decoder_output = Conv2D(32, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(decoder_output)
```

```
    decoder_output = Conv2D(2, (3, 3), activation='tanh', padding='same',
                            strides=[1, 1])(decoder_output)
    decoder_output = UpSampling2D((2, 2))(decoder_output)
    return Model(inputs=[encoder_input, embed_input], outputs=decoder_output)


model = Colorize()
model.compile(optimizer='adam', loss='mean_squared_error')
model.summary()

# Image transformer
datagen = ImageDataGenerator(
    shear_range=0,
    zoom_range=0,
    rotation_range=0,
    horizontal_flip=False,
    rescale=1. / 255)


# Create embedding
def create_inception_embedding(grayscaled_rgb):
    def resize_gray(x):
        return resize(x, (299, 299, 3), mode='constant')

    grayscaled_rgb_resized = np.array([resize_gray(x) for x in grayscaled_rgb])
    grayscaled_rgb_resized = preprocess_input(grayscaled_rgb_resized)
    embed = inception.predict(grayscaled_rgb_resized)
    return embed


# Generate training data
def image_a_b_gen(batch_size=20):
    for batch in datagen.flow_from_directory(directory="data/train", class_mode="input",
                                             batch_size=batch_size):
        X_batch = rgb2gray(np.asarray(batch)[0])
        grayscaled_rgb = gray2rgb(X_batch)
        lab_batch = rgb2lab(np.asarray(batch)[0])
        X_batch = lab_batch[:, :, :, 0]
        X_batch = X_batch.reshape(X_batch.shape + (1,))
        Y_batch = lab_batch[:, :, :, 1:] / 128
        yield [X_batch, create_inception_embedding(grayscaled_rgb)], Y_batch


# Set a learning rate annealer
learning_rate_reduction = ReduceLROnPlateau(monitor='loss',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)
filepath = "output/R25_Art_Colorization_Model.h5"
checkpoint = ModelCheckpoint(filepath,
                             save_best_only=True,
                             monitor='loss',
                             mode='min')
model_callbacks = [learning_rate_reduction, checkpoint]

# Train the Model
BATCH_SIZE = 20
history = model.fit(image_a_b_gen(BATCH_SIZE),
                    epochs=25,
                    verbose=1,
                    steps_per_epoch=36400 / 20,
                    callbacks=model_callbacks
                    )

# Save the Model
model.save(filepath)
model.save_weights("output/R25_Art_Colorization_Weights.h5")
hist_df = pd.DataFrame(history.history)
hist_csv_file = 'output/R25_history.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
```

Listing B.1: Autoencoder & Inception-ResNet-v2 model

B.2 CNN

```
import numpy as np
import pandas as pd
from skimage.transform import resize
from skimage.color import rgb2gray, gray2rgb, rgb2lab, lab2rgb
from keras.applications.inception_resnet_v2 import InceptionResNetV2, preprocess_input
from keras.models import Model, load_model, Sequential
from keras.preprocessing.image import ImageDataGenerator
from keras.layers import Input, Dense, UpSampling2D, RepeatVector, Reshape
from keras.layers.convolutional import Conv2D, Conv2DTranspose
from keras.layers.merge import concatenate
from keras.callbacks import EarlyStopping, ModelCheckpoint, ReduceLROnPlateau


def Colorize():
    # Encoder
    encoder_input = Input(shape=(256, 256, 1,))
    encoder_output = Conv2D(64, (3, 3), activation='relu', padding='same',
                            strides=[2, 2])(encoder_input)
    encoder_output = Conv2D(128, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)
    encoder_output = Conv2D(128, (3, 3), activation='relu', padding='same',
                            strides=[2, 2])(encoder_output)
    encoder_output = Conv2D(256, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)
    encoder_output = Conv2D(256, (3, 3), activation='relu', padding='same',
                            strides=[2, 2])(encoder_output)
    encoder_output = Conv2D(512, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)
    encoder_output = Conv2D(512, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)
    encoder_output = Conv2D(256, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)

    # Decoder
    decoder_output = Conv2D(128, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(encoder_output)
    decoder_output = UpSampling2D((2, 2))(decoder_output)
    decoder_output = Conv2D(64, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(decoder_output)
    decoder_output = Conv2D(64, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(decoder_output)
    decoder_output = UpSampling2D((2, 2))(decoder_output)
    decoder_output = Conv2D(32, (3, 3), activation='relu', padding='same',
                            strides=[1, 1])(decoder_output)
    decoder_output = Conv2D(2, (3, 3), activation='tanh', padding='same',
                            strides=[1, 1])(decoder_output)
    decoder_output = UpSampling2D((2, 2))(decoder_output)
    return Model(inputs=encoder_input, outputs=decoder_output)


model = Colorize()
model.compile(optimizer='adam', loss='mean_squared_error')
model.summary()

# Image transformer
datagen = ImageDataGenerator(
    shear_range=0,
    zoom_range=0,
    rotation_range=0,
    horizontal_flip=False,
    rescale=1. / 255)

# Generate training data
def image_a_b_gen(batch_size=20):
    for batch in datagen.flow_from_directory(directory="data/train", class_mode="input",
                                             batch_size=batch_size):
        lab_batch = rgb2lab(np.asarray(batch)[0])
        X_batch = lab_batch[:, :, :, 0]
        X_batch = X_batch.reshape(X_batch.shape + (1,))
        Y_batch = lab_batch[:, :, :, 1:] / 128
        yield X_batch, Y_batch


# Set a learning rate annealer
```

```
learning_rate_reduction = ReduceLROnPlateau(monitor='loss',
                                            patience=3,
                                            verbose=1,
                                            factor=0.5,
                                            min_lr=0.00001)
filepath = "output/50_Art_Colorization_Model.h5"
checkpoint = ModelCheckpoint(filepath,
                             save_best_only=True,
                             monitor='loss',
                             mode='min')
model_callbacks = [learning_rate_reduction, checkpoint]

# Train the Model
BATCH_SIZE = 20
history = model.fit(image_a_b_gen(BATCH_SIZE),
                    epochs=50,
                    verbose=1,
                    steps_per_epoch=36400 / 20,
                    callbacks=model_callbacks
                    )

# Save the Model
model.save(filepath)
model.save_weights("output/50_Art_Colorization_Weights.h5")
hist_df = pd.DataFrame(history.history)
hist_csv_file = 'output/50_history.csv'
with open(hist_csv_file, mode='w') as f:
    hist_df.to_csv(f)
```

Listing B.2: Basic CNN model

B.3 Test model

```
import os
import sys
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
from itertools import chain
from skimage.io import imread, imshow, imread_collection, concatenate_images
from skimage.transform import resize
from skimage.color import rgb2gray, gray2rgb, rgb2lab, lab2rgb
from keras.applications.inception_resnet_v2 import InceptionResNetV2, preprocess_input
from keras.models import Model, load_model, Sequential

IMG_WIDTH = 256
IMG_HEIGHT = 256
IMG_CHANNELS = 3
INPUT_SHAPE = (IMG_HEIGHT, IMG_WIDTH, 1)
TEST_PATH = 'data/test/'
test_ids = next(os.walk(TEST_PATH))[2]

X_test = np.zeros((len(test_ids), IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS), dtype=np.uint8)
missing_count = 0
print('Getting test images ... ')
sys.stdout.flush()
for n, id_ in tqdm(enumerate(test_ids), total=len(test_ids)):
    path = TEST_PATH + id_ + ''
    try:
        img = imread(path)
        img = resize(img, (IMG_HEIGHT, IMG_WIDTH), mode='constant', preserve_range=True)
        X_test[n - missing_count] = img
    except:
        # print(" Problem with: "+path)
        missing_count += 1

print("Total missing: " + str(missing_count))

inception = InceptionResNetV2(weights=None, include_top=True)
inception.load_weights('weight/inception_resnet_v2_weights_tf_dim_ordering_tf_kernel.h5')
model = load_model('output/R25_Art_Colorization_Model.h5')


# Create embedding
def create_inception_embedding(grayscaled_rgb):
```

```
    def resize_gray(x):
        return resize(x, (299, 299, 3), mode='constant')

    grayscaled_rgb_resized = np.array([resize_gray(x) for x in grayscaled_rgb])
    grayscaled_rgb_resized = preprocess_input(grayscaled_rgb_resized)
    # with inception.graph.as_default():
    embed = inception.predict(grayscaled_rgb_resized)
    return embed


sample = X_test[:5]
color_me = gray2rgb(rgb2gray(sample))
color_me_embed = create_inception_embedding(color_me)
color_me = rgb2lab(color_me)[:, :, :, 0]
color_me = color_me.reshape(color_me.shape + (1,))

output = model.predict([color_me, color_me_embed])
output = output * 256
decoded_imgs = np.zeros((len(output), 256, 256, 3))

for i in range(len(output)):
    cur = np.zeros((256, 256, 3))
    cur[:, :, 0] = color_me[i][:, :, 0]
    cur[:, :, 1:] = output[i]
    decoded_imgs[i] = lab2rgb(cur)
    cv2.imwrite("colorized_" + str(i) + ".jpg", lab2rgb(cur))

plt.figure(figsize=(20, 12))
for i in range(5):
    # grayscale
    plt.subplot(3, 5, i + 1)
    plt.imshow(rgb2gray(sample)[i].reshape(256, 256))
    plt.gray()
    plt.axis('off')

    # recolorization
    plt.subplot(3, 5, i + 1 + 5)
    plt.imshow(decoded_imgs[i].reshape(256, 256, 3))
    plt.axis('off')

    # original
    plt.subplot(3, 5, i + 1 + 10)
    plt.imshow(sample[i].reshape(256, 256, 3))
    plt.axis('off')

plt.tight_layout()
plt.show()
```

Listing B.3: Test model