The CLOCK Model*

CLassifier for Outdoor Images using CNNs with Keras

Gian Alix
gian.alix@gmail.com

Department of Mathematics and Statistics
York University

Project Report
April 10, 2020

Abstract. Image Analysis and Classification have become popular areas of studies being investigated by several researchers in the field of Data Science. [1] A few of the applications of these types of problems include object detection, image construction and photo segmentation. They all belong to the same class of problems under *Computer Vision*, and the approach usually deals with *Neural Networks* and *Deep Learning*, as few of its solutions among others.

In this project, we survey an image classification problem using a renowned Deep Learning technique called *Convolutional Neural Networks*. For this specific project, [2] a set of 800 images available, as detailed in [3] T. Ng's dataset technical report, was used for the training and testing of the recommended model. The goal is to come up with an algorithm or a machine learning model that has a better accuracy than [4] X. Gao's naïve logistic model[1]. It turns out that our proposed **CLOCK** Model has improved the performance of the naïve model by $\sim 10\%$, which is a terrific improvement! This paper will examine the problem in great detail, will expound on this suggested model, and will enumerate some further extensions.

Fig. 1. Object detection example - one of the applications of Image Classification. Here, the software was able to annotate all objects found in the image and give each a label. (**Image Source:** `https://medium.com/analytics-vidhya/beginners-guide-to-object-detection-algorithms-6620fb31c375`)

* The paper, source code for the R script, along with all the materials used for this project such as datasets, are found in my Github Repository: `https://github.com/techGIAN/CLOCK_Image_Classifier`, made publicly available only after April 10, 2020.

[1] X. Gao has designed a logistic model to classify the 800 images in the dataset. The goal of this paper is to come up with a model that outperforms this.

Table of Contents

1 Introduction

1.1 Problem Definition

The main task of this study is to come up with a model or a machine learning algorithm that is able to perform the image classification task, given the [2] *personal image data set* from [3] T. Ng's technical report. The dataset has labels:

- `artificial`
- `indoor-dark`
- `indoor-light`
- `natural`
- `outdoor-dawn-dusk`
- `outdoor-day`
- `outdoor-night`
- `outdoor-rain-snow`

There are a few ways we can classify the images:

I. Classify whether a given image has been labelled `outdoor-day` or not
II. Classify whether a given image is an `outdoor`-type image or not
III. Classify the label of each image (as per the above labels)

Clearly, all are image classification type problems but they get more challenging down the line (i.e. Problem II above is harder than Problem I , but Problem III is harder than both Problems I and II).

1.2 Hypothesis Statement

In [4] X. Gao's model, the algorithm is able to classify images as `outdoor-day` or not (i.e. Problem I) with a decent performance; however the proposed ***CLassifier for Outdoor Images using Convolutional Networks through Keras*** Model, or **CLOCK** for short, is able to beat this naïve logistic algorithm by solving a much more difficult problem (Problem II) and with a better performance as well.

It will be clear soon that Problem III will prove to be a tough challenge for **CLOCK**, and a few of the reasons why this is so will be explained further in the paper.

1.3 Rationale

The task of identifying objects from images, or classifying images to be of a certain kind is said to be easy for the human eyes but serves as a challenge for machines [5, p.440], as suggested by Ballard (1983). It is the goal of several machine learning researchers to be able to come up with models that are able to perform simple human tasks but are not-so straightforward for computers. One of the reasons that makes this objective difficult to accomplish is that these models are meant to be designed to have great measures of accuracies and are expected to perform well, as a human would and sometimes even better!

In the case of computer vision tasks such as image classification, the purpose of such are usually application-based. For instance, the [6] MNIST dataset as seen from **Fig. 2**, which is composed of about 70,000 images of handwritten digits from 0 through 9, is a common dataset that is used in learning and understanding convolutional neural networks in order to classify the digit in each image. However, this can be further applied into other applications. Bank apps nowadays are able to read physical copies of your cheque so that you can do so conveniently on your phone.

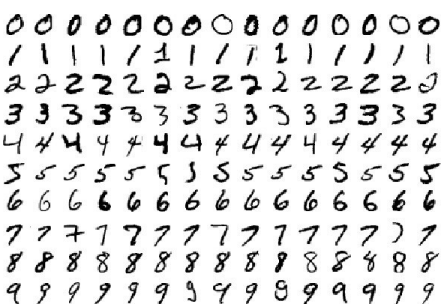


Fig. 2. The MNIST Dataset of Handwritten Digits by LeCun, Cortes and Burges (**Image Source:** `https://medium.com/@afozbek_/how-to-train-a-model-with-mnist-dataset-d79f8123ba84`)

Another example would be object detection. The task of identifying whether a particular image contains a certain common object such as a person, an animal like a dog or cat, or even an inanimate object like a ball, a car or a fire hydrant, is common in the study of computer vision. Not only does the task entail of identification of whether the target object exists in the image, but also its location on the image. Does it appear on top of the image? To the left? Is it centered? The algorithm then draws a box or a rectangle around the object and gives it a label. This is called the *annotation* and we see an example of this on the cover of the paper in **Fig. 1**. But what can this do? As an extension, this can be used for security cameras. Setting up a security camera inside a shop and applying the model onto it in order to capture a target, say a thief. The model will pre-learn and train on past footages of the face and body of the thief that the owner of the shop has suspected to be doing illegal shoplifting, and this model is said to perform well with more footages and higher accuracy. Come the next time the shoplifter has entered store, the security camera could potentially detect this person and get him caught easily.

Such examples mentioned should be able to justify the rationale of the study.

1.4 Background Literature

1.4.1 Neural Networks

Let us begin by giving context and a gentle overview of the different definitions and background knowledge before preceeding with this paper. To start, we dive into a better understanding of the notion of *Neural Networks*. Commonly known as **deep feedforward networks** (or some books would call them **feedforward neural networks** or sometimes **multi-layer perceptrons**), [5, p.163] approximates some function f given some input vector $\mathbf{x}$ and some weight vector $\mathbf{w}$. This can easily be visualized through a graph or what is known as a network, as seen from **Fig. 3**.

In its network architecture, we see that this is composed of several layers of nodes (also called *neurons*, alluding to the scientific field of Neurology). The first layer, called the **input layer**, is where we feed in different input values usually in the form of a vector. This is sometimes called **predictors** in the context

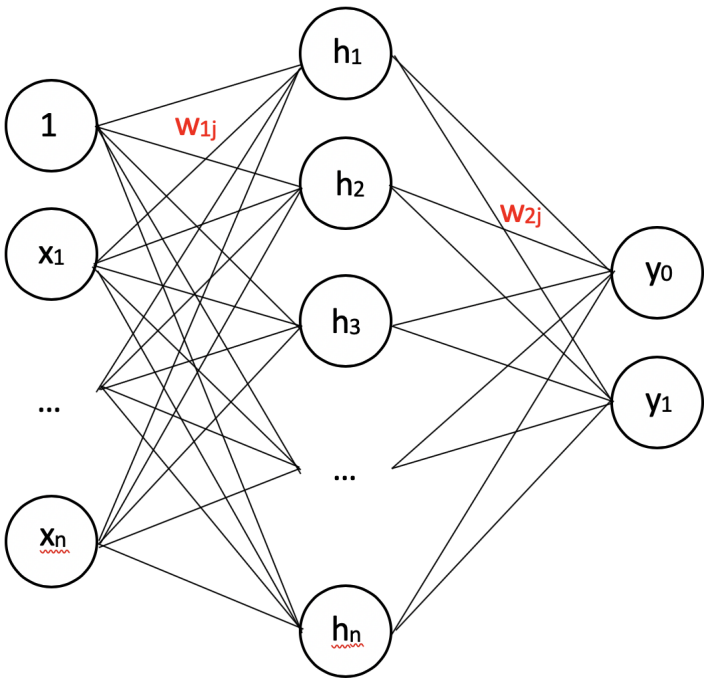


Fig. 3. An example of a feedforward, fully-connected neural network architecture with one hidden layer and n hidden units. In this example the input layer has a bias term but the hidden layer does not (usually the hidden layer has a bias node, but our example does not). For the output layer, it is composed of two nodes. We also let the weights be represented by $\{w_{ij}\}$ for the ith layer and the jth node. In some books, the activation function is included in the visual network; in our case, it should be implied that it is applied after we pass the values to the next layer.

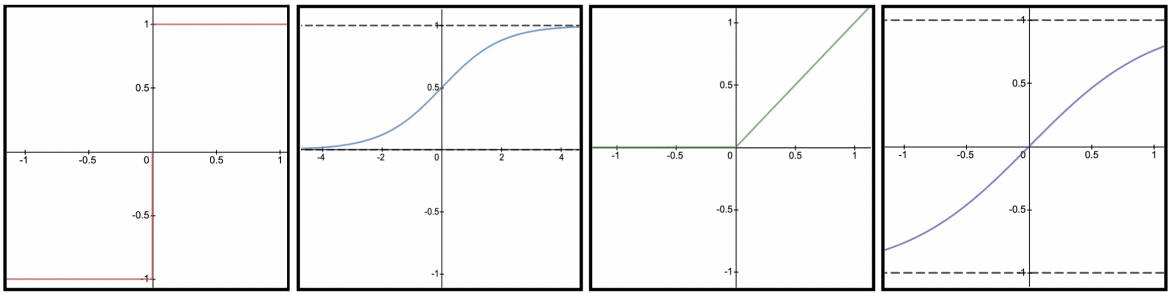


Fig. 4. Examples of activation functions (from left to right): Sign function (outputs +1 for all positive values, -1 for all negatives, and 0 otherwise), Sigmoid function (equal to $\frac{e^x}{1+e^x}$), ReLU function (the maximum of $\max\{0, x\}$) and the Tanh function (or the Hyperbolic function $\tanh(x)$). Graphs produced from the Desmos Graphing Calculator: `https://www.desmos.com/calculator`

of Statistics and its values are usually denoted by $\{x_0, x_1, x_2, ..., x_n\}$, where $x_0 = 1$ usually denotes the *bias* or the *intercept*).

These input values get passed on to the next layer, which we call **hidden layer(s)**, and each edge that connects from the input to the hidden layer represents some weight (usually denoted by $\{w_{10}, w_{11}, w_{12}, ...w_{1n}\}$), and each of these weights determine how important a particular input or predictor is. Since there could be multiple hidden layers in our architecture, then the ith index of the weight $\{w_{ij}\}$ represents the ith hidden layer in the neural network and the jth index represents the jth node of the layer. Note that there could be multiple hidden layers, as there is only one hidden layer in the above architecture.

When values get passed on to the next layer, they also undergo an **activation function**, which can be thought of as an extra function applied to the value that's being passed. This is mostly used to allow nonlinear transformations, or nonlinear mappings. Without it, it would be impossible for neural networks to learn and approximate nonlinear functions. [5, p.169] The most popular activation function used in the hidden layers of the neural network architecture is the **Rectified Linear Unit**, or the **ReLU**, defined by the function $g(x)$:

$$g(x) = \max\{0, x\} \tag{1.1}$$

We can see from **Fig. 4** the different activation functions commonly used in neural networks.

Then after the last hidden layer of the architecture, the last layer would be called the **output layer**. Depending on what the machine learning task is and what the model is trying to learn, then this output layer can range from either just a single value (an output unit y), or even perhaps a multiple neurons in the layer, $\{y_0, y_1, ..., y_n\}$.

1.4.2 Convolutional Neural Networks

Now that we have gained a clearer understanding of Neural Networks, it is time to give a brief overview of Convolutional Neural Networks, or CNNs.

Network Architecture	Accuracy (% Correct)
Net-1: Single Layer Network	80.0%
Net-2: Two Layer Network	87.0%
Net-3: Locally Connected Network	88.5%
Net-4: Constrained Network 1	94.0%
Net-5: Constrained Network 2	98.4%

Table 1. The Test Set performance of five different neural networks on a handwritten digit example (LeCun, 1989). The first three models are regular neural network architectures, while the fourth and fifth are convolutional networks. On average the convolutional networks have performed ∼10% better than the regular networks.

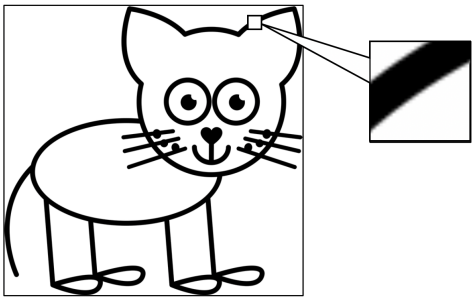


Fig. 5. A pixel on the image of this cat does not provide enough information about the given cat image. We only know that for this particular pixel, there is a stroke here, but this is not adequate for a machine to recognize that this whole image is a cat. (**Image Source:** `https://www.pinclipart.com/maxpin/xxxJR/`)

As [5, p.321] LeCun (1989) had stated that CNNs are a special type of neural networks that have been used for preprocessing data that has a known topology, similar to that of a grid. Time-series data for example is a one-dimensional grid where samples can be taken at regular time intervals; meanwhile, images are good examples of 2-D data where we can subdivide it into a grid of pixels. That said, several research work had been done on CNNs using these types of data, particularly on image sets. In fact, LeCun (1989) argued that [7, p. 567] CNNs have been proven to be good state-of-the-art models in solving problems where images have been involved. For instance, [8, p.407] in LeCun's experiment in 1989, the results from **Table 1** have shown that the convolutional networks were able to classify handwritten digits from the [6] the MNIST dataset, ∼10% much better than the use of the regular NN architecture.

The approach of these machine learning (or *deep learning* in the context of neural networks) algorithms is to break up each input image into several pixels and [5, p.321] use a simple mathematical operation called **convolution** on the networks. This is because each pixel alone does not seem to capture enough information on the given input image. It can be seen from **Fig. 5** that a machine cannot easily detect this cat image given the single pixel alone, but as all the pixels make up the image, then all can potentially be used as predictors by the convolution operation (and the neural network model) in CNNs in order to recognize that the image is indeed a cat.

[5, p. 322-23] This convolution operation as discussed can be represented mathematically as:

$$s(t) = (x * w)(t) = \int x(a)\, w(t-a)\, da \tag{1.2}$$

for some input function x, probability density function w for the weights (or **kernel** in the jargon of convolutional nets), independent variable a and parameter t [5, p. 322]. Our output s is sometimes called a **feature map**.

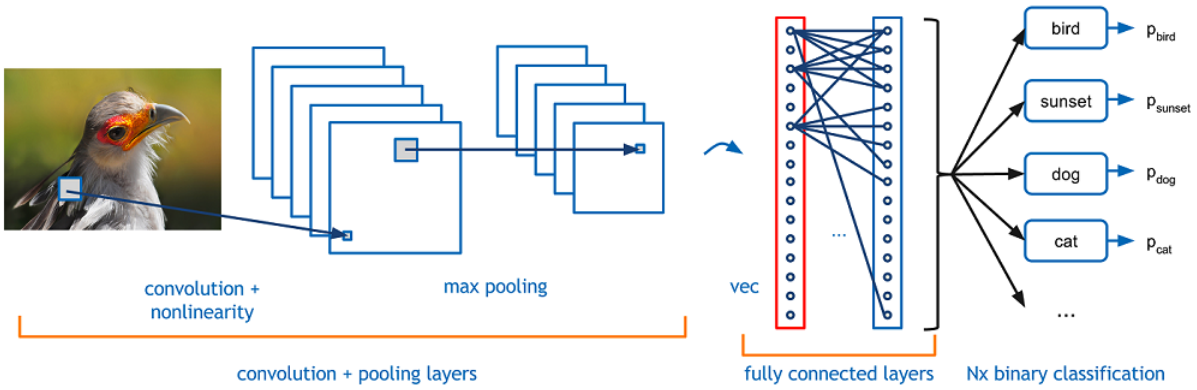


Fig. 6. A big picture of the Convolutional Neural Network Architecture, with an image as input (**Image Source:** `https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/`)

As seen from **Fig. 6** which displays the different layers of the Convolutional Network architecture, the image is divided into a grid of pixels and each pixel represents a particular value. Note that if we had a grayscale image, which implies having only one color channel (color channels briefly covered in **Sec. 1.4.4**), then the image itself is a layer that's divided into pixels; but a color image is still divided into three color channels, and hence there would be three layers and each pixel of the image is split into three different values.

A kernel, or a filter is then applied on this convolution layer and an activation function such as a ReLU is also applied in order to make values to be non-negative. This can be visually represented with a simple example. Here, a 2×2 filter is applied on a 3×3 layer[2], where each filter linearly scans the image as if it was a window and would output into another layer of size 2×2.

$$\begin{bmatrix} 1 & -2 & -1 \\ -5 & 3 & 4 \\ 3 & -2 & 7 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 2 & 1 \\ -4 & 14 \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} 2 & 1 \\ 0 & 14 \end{bmatrix}$$

[5, p. 323-24] Mathematically speaking, a convolution can be expressed as

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{1.3}$$

where I is our image, K is the kernel and S is the discretized convolution. It is important to note that due the commutative property of convolutional nets, then we can rewrite **Eq. (1.3)** as follows:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{1.4}$$

Being able to commute this is due to the fact that our kernel has been flipped. In many CNN implementations, it is in fact more complex to use the convolution operation in the architecture due to this "flipped" property; however, an equivalently-related operation called **cross-correlation** is used instead, which is defined mathematically by:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{1.5}$$

Continuing on, the next layer in the convolution line is the Pooling Layer, a few of them being `max()`, `average()`, among others. It has the same flavor as how we applied a filter onto the image layer, but with the caveat that we are applying the function (i.e. We apply the max pool layer on a window for instance). If we were to apply a max pooling layer in a 3×3 layer example below with size 2×2, then we get:

$$\begin{bmatrix} 9 & 0 & 7 \\ 3 & 7 & 8 \\ 6 & 4 & 5 \end{bmatrix} \xrightarrow{max_{2\times 2}} \begin{bmatrix} 9 & 8 \\ 7 & 8 \end{bmatrix}$$

This should make up the convolutional part of the CNN architecture. As you can imagine, this should have decreased the number of predictors or the dimensions of our original input image. And this is what CNNs are actually good at! So if you are given an image that is 28×28 pixels large, such as those in [6] the MNIST dataset, then you would be fine using a regular Neural Network with $28 \times 28 \times 1 = 784$ input nodes[3] is enough, and not much of a problem. But what about large images that have size, say 1600×900? Then $1600 \times 900 \times n$ input neurons, for n channels, is indeed infeasible. Convolutional Networks are a big help. They serve as feature reduction methods as well!

The rest of the layers that follow are simply the same layers as discussed in **Sec. 1.4.1**, called a **Fully-Connected layer** (or the FC layer).

[2] There are multiple ways the filter can be applied. For this particular filter, we simply applied element-wise multiplication then summed all values within the window.

[3] We can compute for the number of input nodes of the NN by calculating the product of the width $\times$ height of the input image $\times$ number of channels of the images. In this case of the MNIST dataset which is greyscale, then it should be multiplied by 1, for 1 channel

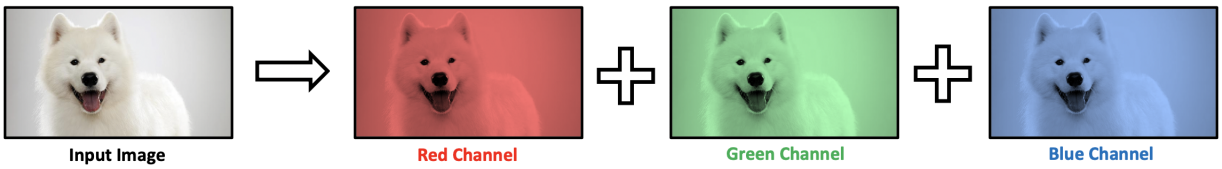


Fig. 7. A Samoyed dog image, broken down into the three RGB channels (**Image Source:** `http://www.animalplanet.com/breed-selector/dog-breeds/working/samoyed.html`)

1.4.3 The Keras Library

The project, particularly our **CLOCK Model**, would not have been possible without Keras, which is basically a neural network API that is written in the Python language. Even when we are working with R, we can still utilize Keras - we simply import the `reticulate` package into R that allows us to have Python on it (importing Keras in R imports the `reticulate` automatically since it is a prerequisite of Keras). Then Keras can then be used. The deep learning Keras library can also help us with the easy implementation of the Convolutional Networks, without having to code it from scratch. We will see in **Sec. 4** how we can use Keras to make the implementation of CNNs more straightforward. The Keras documentation can be found at `https://keras.io/`.

1.4.4 Color Channels

We have been discussing in the previous sections that images are subdivided into pixels and have channels, and depending on the color scheme of the image determines the number of channels it has. These **channels** [9] are indeed the arrays that make up the values when data of images have been imported. Basically, if we import an image and store it into some variable, it is stored as an array. Greyscale images only have one channel, so the array that you are storing for these images is one-dimensional (1D in a sense that it still has the image's height and width dimensions). On the other hand, the more common colored images are stored as three-dimensional arrays (sometimes also called **tensors**), and the reason why this is so is due to the number of channels it has - three, represeting *red*, *green* and *blue*. This is also called the **RGB Channel**, or the **RGB Color Space**. We can see an example of an image broken down into the RGB channels from **Fig. 7**.

Sometimes, we see an *A* channel after the RGB. This is called the **Alpha Channel**, which relates to the transparency of an image. However in some image formats such as JPEG formats, we usually drop this channel as transparency is not supported in these file formats, unlike other formats such as PNG formats.

There is nothing much to say about the RGB color channel that would help further our understanding and knowledge of the study, aside from the fact that storing an image as an array breaks it down into these three channels. That said, we have completed the project's prerequisite background.

1.5 Action Plan

We see on **Fig. 8** the overall picture of how the project was done. In a nutshell, it is subdivided into two major tasks:

1. Data Preprocessing
2. Machine Learning Tasks

Data Preprocessing involves cleaning the data, removing noise and formatting them to ensure that all data are consistent. Part of the data preprocessing also includes loading and importing the data. In fact, in the real world, data preprocessing takes up about ~80% of the time in doing data analyses and machine learning projects. In our case, since we have been given a clean dataset, then we do not need to spend much time on this. See **Sec. 2.1** for more details on our datasets and **Sec. 2.3** for more thorough information on our data preprocessing tasks.

The next major task is machine learning. And the first part to this is to build our model's architecture. In this case, we opt to use Convolutional Networks. Then we also need to set our *hyperparameters*, which

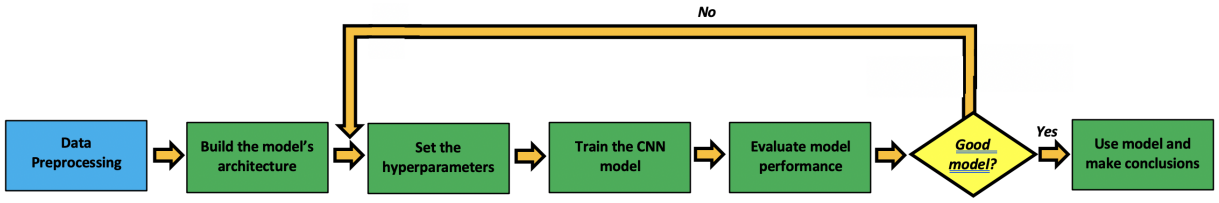


Fig. 8. General Course of Action for the Project

is simply a fancy word for the model's parameters. These include the type of optimizer used, the number of hidden layers, the number of hidden neurons in each layer, the learning rate, and so forth. This is one of the trickiest parts of the project, as there is no general rule of thumb in setting these parameters to get the best accurate model. There are some hyperparameters that are dependent on some other hyperparameters. So increasing one of them, say the learning rate, means that we may have to increase the number of training epochs as well. This is all about the guess-and-check. And then after setting the "what-we-think-is-right" hyperparameters, we can then train the model and hope for the best. We then evaluate its performance[4]. Then ask ourselves, is it a good model? Usually we can define *good* here as a model that has surpassed threshold accuracy μ, or in this case a model that has outperformed another model that we are trying to improve, which is the [4] Gao model. Thus, this means that our model can be considered good, and we may use this particular model that we have found to make any analyses and conclusions. Otherwise, we may need to go back and tweak the values of our hyperparameters in order to find a better model, which we keep on doing until we have succeeded in our task.

The **CLOCK** model that was proposed had two attempts:

(A1) It had attempted to solve Problem III as defined from **Sec. 1.1**
(A2) It had attempted to solve Problem II as defined from **Sec. 1.1**

Knowing that Problem III is more challenging than Problem II, we will let **CLOCK** take a try on solving Problem III first before Problem II, and see what results we get. Overall, we have let our proposed model run on these two problems and see **Sec. 5** for the results. Of course, since Problems II and III are different problem definitions, then additional Data Preprocessing may be required when running another type of problem, to accommodate the requirements.

[4] We can use *accuracy* as the performance metric

2 Data

2.1 The Personal Columbia Set

Image Category	Count
artificial	142
indoor-dark	68
indoor-light	74
natural	111
outdoor-dawn-dusk	31
outdoor-day	277
outdoor-night	34
outdoor-rain-snow	63
Total	800

Table 2. Image count for the categories in the *Personal Columbia* Image Dataset

The [2] Personal Columbia Image dataset (also described in [3, p. 13-17] T. Ng's technical report) is composed of 800 total images, from different settings. Each image belongs to a particular category and are labelled as such:

- `artificial`
- `indoor`
- `indoor-light`
- `natural`
- `outdoor-dawn-dusk`
- `outdoor-day`
- `outdoor-night`
- `outdoor-rain-snow`

We can see [3, p. 17] from **Table 2**, an image count for the eight different categories. We must note of this when we come to our analysis later.

There are three photographers who took the pictures, T. Ng himself, along with two other co-authors of the technical paper, J. Hsu and M. Pepeljugoksi. They have used the Canon 10D and the Nikon D70 camera for their shots. And the locations as to where they took their photos are in Boston and New York. We can see from **Fig. 9** an example image that the team has taken.

2.2 The Image Set's Metadata

Accompanying the [2] *Personal Columbia Image Set* is its metadata, in comma-separated (CSV) format, which can also be viewed here:

`https://github.com/techGIAN/CLOCK_Image_Classifier/blob/master/photoMetaData.csv`

Fig. 9. A sample Image of the *Personal Columbia Set* with filename `CRW_4786_JFR.jpg`

Name	Category	Camera	Location	Photographer
CRW_5813_JFR.jpg	artificial	Canon 10D	New York	Tian-Tsong
CRW_5499_JFR.jpg	outdoor-dawn-dusk	Canon 10D	New York	Tian-Tsong
DSC_1485.jpg	outdoor-day	Nikon D70	Boston	Tian-Tsong
DSC_0698.jpg	indoor-dark	Nikon D70	New York	Jessie
CRW_5021_JFP.jpg	outdoor-rain-snow	Canon 10D	New York	Martin
...	...	...	...	...

Table 3. Dataframe of the metadata consisting of five random rows

This is important as well when we import our image dataset, because we can know for instance what is the category or class a particular image with a particular filename belongs to. So for example, we want to know under which class an image as displayed in **Fig. 9** belongs to? It looks like it's `outdoor-day`, and if we check the metadata file then we can verify that the image above shown is indeed an image that belongs to the `outdoor-day` class.

In summary, the metadata can be imported as a dataframe and has the attributes `name`, `category`, `camera`, `location`, and `photographer`. During the Convolutional Networks architecture and training, the last three attributes mentioned may not seem significant as we will rely only on the image itself as the predictor (of course the pixels of the image). **Table 3** gives a better overview of this metadata set.

It shall be noted that as we tackle Problems II and III, we may need to add another column in this dataframe that serves as the "class" label. For instance, since Problem II deals with classifying an image as of `outdoor` category or not, then we can check if the `category` of a particular row contains the word `outdoor` in it. If so, we write `1` under a new column called `Label`; otherwise, we write `0`. It can be seen that **Table 4** is the resultant dataframe after adding the additional column from **Table 3**.

Name	Category	Camera	Location	Photographer	Label
CRW_5813_JFR.jpg	artificial	Canon 10D	New York	Tian-Tsong	0
CRW_5499_JFR.jpg	outdoor-dawn-dusk	Canon 10D	New York	Tian-Tsong	1
DSC_1485.jpg	outdoor-day	Nikon D70	Boston	Tian-Tsong	1
DSC_0698.jpg	indoor-dark	Nikon D70	New York	Jessie	0
CRW_5021_JFP.jpg	outdoor-rain-snow	Canon 10D	New York	Martin	1
...	...	...	...	...	...

Table 4. Adding an additional column from **Table 3** known as the `Label` column, where we write `1` if the particular image is an `outdoor`-type, and `0` otherwise; this is said to help us solve Problem II. The modified metadata dataframe can be found in my Github repo: `https://github.com/techGIAN/CLOCK_Image_Classifier/blob/master/photoMetadataWithLabels.csv`

Meanwhile for Problem III, the label assigned may be integers 0 through 7, indicating `artificial`, `indoor-dark`, etc. However, we will see later that there is actually a better way to represent this, which we call **one hot vectors**, or **one hot encoding**. Basically after assigning the integers 0 through 7 for each class, then for the integer label $i \in [0, 7]$, we represent it as a vector of the form as shown by **Eq. (2.1)**:

$$\mathbf{v}(i) = \big[0, 0, \ldots, 0, 1, 0, \ldots, 0, 0\big] \tag{2.1}$$
$$\uparrow$$
$$i^{\text{th}} \text{ spot}$$

This is a common technique when dealing with categorical classes (mutli-class). And so **Table 5** shows a table of the one hot vector representations of each class label. One thing to note as well is that the ith spot as represented by **Eq. (2.1)** is indexed from zero and not by one (i.e. v_1 of the hot vector is for the class label 0, v_2 for class label 1 and so forth). In general, then v_{i+1} position of the hot vector reprsents class i. And of course, for an n-class dataset, then there are n possible unique one hot vectors.

Image Category	i	One Hot Vector
artificial	0	[1,0,0,0,0,0,0,0]
indoor-dark	1	[0,1,0,0,0,0,0,0]
indoor-light	2	[0,0,1,0,0,0,0,0]
natural	3	[0,0,0,1,0,0,0,0]
outdoor-dawn-dusk	4	[0,0,0,0,1,0,0,0]
outdoor-day	5	[0,0,0,0,0,1,0,0]
outdoor-night	6	[0,0,0,0,0,0,1,0]
outdoor-rain-snow	7	[0,0,0,0,0,0,0,1]

Table 5. One hot vector encoding for our class variables, as defined by Problem III

2.3 Data Preprocessing

2.3.1 Library Imports

Although not the main heart of data preprocessing, it is important to know what libraries and packages are needed to be imported in the R file in order for certain functions, methods and algorithms to work. As an example, the `reticulate` package as mentioned from **Sec. 1.4.3** is required in order for Python to run in R, which is a prerequisite in running Keras, the neural network library. Another package known as `plyr` in R is also an essential tool in data splitting, applying and combining data; this implies that it should also be imported which would greatly help us in data preprocessing.

Code 2.1 shows the R code of all the necessary imports that we will be needing. Refer to **Table 6** for the documentation / descriptions of each of these packages.

```
================================================================================
Code 2.1: The necessary library and package imports for the project
================================================================================
1. library(plyr)
2. library(keras)
3. library(EBImage)
4. library(stringr)
5. library(pbapply)
6. library(tensorflow)
7. library(reticulate)
================================================================================
```

2.3.2 Data Loading

As per **Sec. 2.1** and **Sec. 2.2**, there will be two datasets that will be imported into R. First would be the [2] *personal columbia image dataset*, and the second being the metadata of this dataset. To import the metadata, we use the `read.csv()` function that's built-in in R, which allows us to read a comma-separated file and store it as a dataframe onto a variable. In R, this can be coded as:

```
> meta_data <- read.csv("../photoMetaData.csv")
```

Library / Package	Description
`plyr`	Tools for Splitting, Applying and Combining Data
`keras`	R Interface to 'Keras'
`EBImage`	Image Processing and Analusis Toolbox for R
`stringr`	Simple, Consistent Wrappers for Common String Operations
`pbapply`	Adding Progress Bar to 'apply' functions
`tensorflow`	R Interface to 'Tensorflow'
`reticulate`	R Interface to 'Python'

Table 6. List of the Packages that we need for the project, including their description, as what is displayed in RStudio.

Depending on where your current directory of the project is, and the location of the photo metadata file, then the paraneter of `read.csv()` can vary; you simply have to set up its path. We can always *query*[5] about this dataframe `meta_data`. For instance, if you would like to know how many instances there are in the dataframe, we can type in

```
> n_images <- nrow(meta_data)
> n_images
[1] 800
```

This means that there are 800 observations or instances, images in this case, in the metadata set; and more importantly since the metadata serves as a file that gives information on our image set, then there should also be 800 images in our *Personal Columbia image set*. Also note that we have created a new variable called `n_images`, which we will see later will be important.

We may also be interested in viewing the first few elements of the dataframe; we can do so using the `head()` command. Sometimes the purpose of such is so that we could have a glimpse of what kinds of data we are dealing with in the metadataset, as well as the attributes of the dataframe. We can always use the `attributes()` command in R but the use of the `head()` command should give more information. The usage of the said command are as follows: `head(meta_data)`, and will output the first six rows of the dataframe:

```
================================================================================
              name    category    camera location photographer
1 CRW_4786_JFR.jpg outdoor-day canon 10D new york       martin
2 CRW_4787_JFR.jpg outdoor-day canon 10D new york       martin
3 CRW_4788_JFR.jpg outdoor-day canon 10D new york       martin
4 CRW_4789_JFR.jpg outdoor-day canon 10D new york       martin
5 CRW_4790_JFR.jpg outdoor-day canon 10D new york       martin
6 CRW_4791_JFR.jpg outdoor-day canon 10D new york       martin
================================================================================
```

In the real world, it may be possible that we may deal with noisy or dirty data. For instance, if the data provided had missing values, how do we deal with that? Lucky enough for us, our metadata is clean and does not have any noise, inconsistencies or missing data. We can always do a sanity check for ensuring there are no missing values and if so, we deal with them accordingly; we use the command `sum(is.na(meta_data))`. The command `is.na()` determines whether there is a missing value and `sum()` here should count how many are missing.

```
================================================================================
Code 2.2: Loading the Sample Image
================================================================================
1. sample_IMG <- readImage("../columbiaImages/CRW_4786_JFR.jpg")
2. sample_IMG
3. display(sample_IMG)
4. meta_data[1,]
================================================================================
```

Now to load our images. Let us first consider importing a single image; suppose the first one in the above list. This is also the image as displayed in **Fig. 9**. **Code 2.2** should then show an example of importing this sample image. To briefly explain, line 1 in **Code 2.2** should read the sample image (the image path can vary depending on the project and image directories) using `EBImage`'s `readImage()` function. The function `readJPEG()` from the `JPEG` library can also do the imports but doesn't really give much information and details on our image data unlike the `EBImage` package. Line 2 should display some pertient information about the image data that is imported. We can see the result of the output of Line 2 from Code 2.2 below:

[5] A common term in Data Analysis linggo which just indicates that we may need to survey the data provided in order to gain some insights of what kinds of preprocessing and analyses we might do in order to comne up with conclusions.

```
================================================================================
Image
  colorMode    : Color
  storage.mode : double
  dim          : 737 492 3
  frames.total : 3
  frames.render: 1
================================================================================
```

This just says that the image is colored and that its values are stored as a 3-dimensional array of `double`-type values (basically not integers), has a width of 737 and a height of 492. The image, as it is colored, has 3 channels, as discussed in **Sec. 1.4.4**. There are also three frames and one frame to render. I think the most important detail to note here is the dimension of the stored array, as dimension sizes are important to notice when setting up our neural network architectures (similar to how matrices have to have the right dimensions in order to be multiplied, such as the dimension of some matrix A should be $m \times n$ and the dimension of some matrix B should $n \times p$ so the AB is defined and has dimensions $m \times p$).

```
================================================================================
imageData(object)[1:5,1:6,1]
          [,1]      [,2]      [,3]      [,4]      [,5]      [,6]
[1,] 0.4117647 0.4117647 0.4117647 0.4156863 0.4196078 0.4196078
[2,] 0.4039216 0.4039216 0.4039216 0.4117647 0.4156863 0.4196078
[3,] 0.4039216 0.4000000 0.4039216 0.4078431 0.4156863 0.4196078
[4,] 0.4039216 0.4000000 0.4039216 0.4078431 0.4196078 0.4235294
[5,] 0.4196078 0.4156863 0.4156863 0.4235294 0.4313725 0.4392157
================================================================================
```

Then Line 3 of **Code 2.2** simply displays the image as what we have seen in **Fig. 9** and Line 4 displays the first color channel (red) and a portion of the image displayed as `double`-type values in an array, as seen in the above. So for instance in the red channel, the uppermnost left cell (in Cell [1,1]) is valued 0.4117647, which indicates the *redness* of that particular pixel. All values in this channel, as well as in the green and blue channels are *probability-based* (i.e. values ranging between [0,1]), as rendered by `EBImage`).

```
================================================================================
Code 2.3: Loading the Image Set
================================================================================
 1. # new dimensions of image when scaled
 2. new_width <- 32
 3. new_height <- 32
 4. channels <- 3   # represents the RGB Channel
 5
 6. # read all images and store in a 4d tensor
 7. img_set <- array(NA, dim=c(n_images,new_height,new_width,channels))
 8. for (i in 1:n_images) {
 9.       # paste() function is used to "concatenate" strings
10.       # use the metadata frame for the filenames
11.   filename <- paste0("../columbiaImages/",meta_data\$name[i])
12.   image <- readImage(filename)
13.   img_resized <- resize(image, w = new_width, h = new_height)
14.       # adds only the resized image into the img_set
15.   img_set[i,,,] = imageData(img_resized)
16. }
================================================================================
```

There are two things to note here: (1) All the images more or less have the same size (the photos taken by the Canon camera is slightly larger than the photos taken by Nikon as seen by their number of pixels); all images are just too large, and (2) there's a lot of images to import! For the first point, this would not have been a problem for CNNs as we have talked about from **Sec. 1.4.2** that they are also known for being good techniques for feature reduction (or dimensionality reduction) so a large enough

photo shouldn't really be a problem. Furthermore, we scale down the images to 32×32 to make it more manageable and make all images standardized despite the image size differences between the two cameras used. However, the second point brings about the fact that the number of photos we are dealing with is massive. Also note, that not only do we train the model with these large images but we also have to import them first and then preprocess - this is going to take up a lot of run time! So to make things easier and simpler, we can scale down our image (square-sized) as mentioned earlier to a smaller dimension, so that training would not be too hard (although we would still be undergoing importing and preprocessing of the large images); we would not be doing this manually so we let `EBImage`'s function called `resize()` do it for us. Then as you can imagine, we can utilize a `for` loop to take each image, scale it, then store in a 4-D tensor called `img_set` of size $w \times x \times y \times z$, where w is the number of images (800), $x \times y$ is the area of the image (width $\times$ height), and z is the color channel. **Code 2.3** should complete our Data Loading stage (see comments on the code for extra details).

2.3.3 Data Annotations

Recall that Problem I only requires the model to classify whether its category belongs to `outdoor-day` or not, and we need not create another column for this in the metadata frame, as the `category` column can already serve as the class label. Problem II however solves a bigger problem: can it classify an image as `outdoor`-type or not. That means all images that have categories `outdoor-day`, `outdoor-dawn-dusk`, `outdoor-night`, and `outdoor-rain-snow` all belong to this category; and all other images do not. We do not actually have to check whether the category of a particular image is in one of these mentioned. We can simply create a new indicator column that helps us. To do so, we can reinitialize the class labels by *annotating* a 1 for all those `outdoor`-type images; and 0 for otherwise. So we simply create an additional column in the metadata frame and this will serve as the new label variable. In the code below, `grepl()` is simply a function that checks whether a particular string contains another particular string. So if a certain word, the `category` of the image in this case, has the word `outdoor` then we put 1; otherwise, we put 0.

```
> for (i in 1:n_images) {
+   meta_data$label[i] <- if(grepl("outdoor",meta_data$category[i],fixed=TRUE)) 1 else 0
+ }
```

As you can imagine, the resultant dataframe would be the one that you have seen from **Table 4**. And this is how we preprocess data when solving Problem II. For Problem III though, the only change we'll be modifying from the data preprocessing task of Problem II is that instead of 0s and 1s, we have 0 through 7 (indicating one value for each category). Then an additional preprocessing step is to convert these integers into one hot vectors (useful for Problem III, not so for Problem II).

2.3.4 Data Split

This is one of the most crucial parts of data preprocessing because we have to split our dataset into the training set, validation set and test set. There is usually no general rule of thumb to follow when splitting these. Splitting them into equal parts isn't really a good idea, since we would like our model to train with much data as it could. So we have to allocate more data to the training set. There are other models whose data split is at 50-25-25. Others at 60-20-20. I also know of the 70-10-20 split. In our case, we'll do the 80-10-10 split. This means that given our 800 images in the dataset, then 640 of those will belong to the training set, and 80 images each for the validation and test sets. Note that no data or image can belong to more than one data group. It also depends on how many total datasets you have; the more the better, and this should help you decide the ratio of how you will split the dataset.

Obviously, the purpose of the training set is used to train the model. Usually, for the supervised training (such as the task we are doing right now), then the model tries predicting the given label of an image in the training set and compares the result with the actual label. The accuracy is determined, and if it keeps on getting good accuracies (or correct predictions), then weights within the network are kept; otherwise, then they are updated. And this is why it is critical to have as much training data as possible. By how much and what is a good enough training dataset? We don't really know. It's one of those tricky aspects similar to setting the hyperparameters of a neural network model. It's all about the guessing and checking.

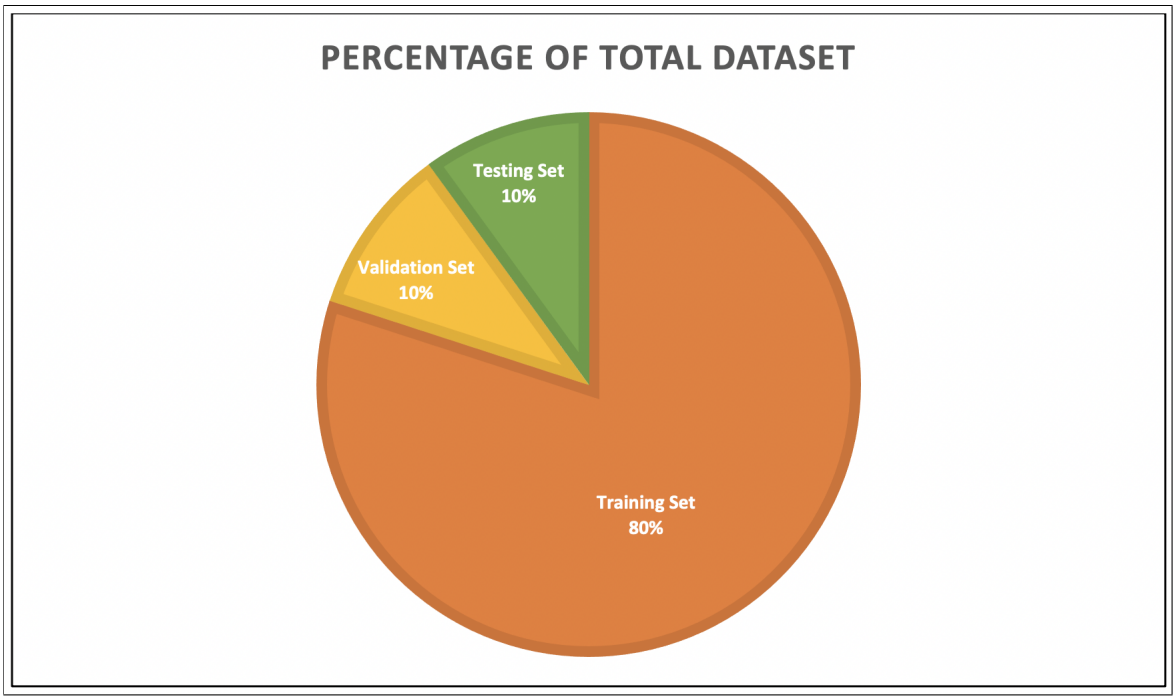


Fig. 10. Our 800 images will be split randomly into Training Set, Validation Set and Test Set (80-10-10 ratio, or 640 images for the Training Set and 80 images each for the Validation and Test Sets)

```
================================================================================
Code 2.4: Data split into Training Set, Validation Set and Testing Set
================================================================================
 1. train_split <- 0.8
 2. validation_split <- 0.5
 3. train_sample_size <- floor(train_split*n_images)
 4. arr <- 1:n_images
 5. train = sample(arr,train_sample_size)
 6. train_set_x = img_set[train,,,]
 7.
 8. not_training <- arr[!arr \%in\% train]
 9. validation_sample_size <- floor(validation_split*length(not_training))
10. validation = sample(not_training,validation_sample_size)
11. validation_set_x = img_set[validation,,,]
12.
13. not_tr_te <- not_training[!not_training \%in\% validation]
14. test_set_x = img_set[not_tr_te,,,]
15.
16. meta_train = meta_data[train,]
17. meta_validation = meta_data[validation,]
18. meta_test = meta_data[not_tr_te,]
19.
20. train_set_y_cat = as.matrix(meta_train[ncol(meta_data)])
21. validation_set_y_cat = as.matrix(meta_validation[ncol(meta_data)])
22. test_set_y_cat = as.matrix(meta_test[ncol(meta_data)])
23.
24. # use keras' built-in function for one-hot encoding
25. train_set_y <- to_categorical(train_set_y_cat,num_classes=unique_labels_count)
26. validation_set_y <- to_categorical(validation_set_y_cat,num_classes=unique_labels_count)
27. test_set_y <- to_categorical(test_set_y_cat,num_classes=unique_labels_count)
================================================================================
```

The validation set, shares some similarities with the training set, but with a few minor differences. The reason why we call it *validation set* is because we would like to verify or validate whether the training accuracies of the model during training is indeed correct and reliable. During the validation run, it will calculate the accuracy similar to the training run but it does not update the weights of the network - it is merely there for verification and validation! It should tell you whether the model will overfit, underfit, (or neither) new data that it has never seen yet. In this case, the testing set; since it will be assumed that the model has never seen yet any of the testing sets during training and validation.

And finally, we use the test set for testing obviously. We would like to know how well the model actually did (really important!) How accurate it is? The testing set will serve as new data for the model to predict, and this is where we will evaluate our model and test whether we can keep the model or tweak the model's hyperparameters.

Code 2.4 should do the splitting randomly of the 800 images into the training set, validation set and testing set. Here is a brief explanation of some parts of **Code 2.4**. 80% of the image set is training data, by Line 1. The remaining 20% is not for training data; half of those (50%) are for validation and the rest is testing (should explain 0.5 in Line 2). The reason we are using `n_images`, instead of 800, is because we may never know; what if another image set was imported instead that has more than 800 images (or less)? This helps to keep our model robust as well to any other image data that follows the same format as the one with the *Personal Columbia Image Set*. Lines 3-6 just asks R to random sample 640 numbers between 1 and 800. As each of the image is indexed from the metadata frame, then we can get the images based on the random sample. Store the images (get them from `img_set`) into `train_set_x`.

And then we want to to remove all numbers that have already been selected during the random sampling for the training set. We do so and then perform another random sample on the remaining numbers (80 numbers) and these will be for the validation set. Grab the images from `img_set` that have these indices, as what was done for the training set. Store them in a vairable called `validation_set_x`. We can see this done in Lines 8-11. The remaining images are for the testing set; we store all the remaining images in `test_set_x`. This is seen in Lines 13-14.

Lines 16-18 of **Code 2.4** are for the metadata. It should be self-explanatory from here that the metadata of each image has also been split into their respective data groups. Lines 20-22 simply converts into a matrix the last column of our metadata, which is the label. Lines 25-27 changes numerical labels to categorical, using the one-hot encoding of Keras. Refer to **Table 5** for the one-hot vector conversions. This is really helpful for Problem III. This isn't much helpful at all for binary classifications such as in Problems I and II as 0 and 1 are enough indicators for binary class problems.

2.3.5 Time Processing

It is not required or mandatory but if we would like to know how long it took for the entire data preprocessing, then we wrap the entire data preprocessing phase with the code below:

```
> t1 <- Sys.time()
+ ...       # preprocessing methods and tasks here...
+ t2 <- Sys.time()
```

This just means that we first note the time before preprocessing takes place, then we perform the necessary tasks and then after we are done, we note again the current time, which is the time after all processing has finished. The preprocessing time can be computed by subtracting time `t1` from time `t2`, then expressing it as a numeric value (i.e. `preprocessing_time <- as.numeric(t2-t1)`). This should give (in seconds) how long the entire preprocessing time took place.

Note that we can also do something similar for checking how long we were able to do the other tasks of the project (i.e. building a model, training the networks, evaluation, etc.).

For the completed data preprocessing code, refer to Lines 1-94 of the code in **Appendix C** of this paper.

3 Methodology

A visual of the methodology as described in **Sec. 3.1** can be found in **Fig. 11** on **p. 19**, or on **Appendix A**, under the Appendix section. The general course of action or action plan as detailed in **Sec. 1.5** and visually represented by **Fig. 8** will serve as a guide for the explanation of the basic framework.

3.1 The Basic Framework

We begin with the data, in this case both the image set & the metadata are considered to be the datasets for the project. It shall be used in: (1) our models, and (2) X. Gao's model, since we need to evaluate this model (serves as a benchmark of how much accuracy our models need to achieve to outperform it).

We then pick a problem. Either Problem III (our first attempt), or Problem II (our second attempt). Since there is no clear definition of what exact problem we are attempting to solve, then we will try taking a look at both. We first try tackling Problem III, as it is a harder problem than Problem II. The Gao model, upon inspection of the code, seems to have solved Problem I. We will soon discover that Problem I is too easy for **CLOCK**, and this is why Problems III and II are the problems that are designated for **CLOCK** to work on. The type of problem selected defines the data preprocessing processes and methods used, but there should only be minor tweaks and slight variations when electing a different type of problem to tackle.

After selecting a problem, we build the model. This stage includes building of the (convolutional) neural network architectures, setting the hyperparameters and training and validating the model. Note also in this phase that we can opt to select another problem, then redo data preprocessing. Again, this shouldn't be much of a tedious task as most of the data are already clean and noise-free; minor things such as formatting may be done.

The next stage after training a model[6] is to evaluate it. If we look at the diagram, the model after training and validation goes through an *evaluator machine* (standardized for all models here), some hypothetical machine that is supposed to evaluate the model M that is fed into the machine. This should be the testing phase (i.e. using the test data set). The evaluator machine should output the evaluation results of the models. For the results of **M_GAO**, it should serve as a benchmark of our final results and model; this should determine the **CLOCK** Model.

After the results have been released by the hypothetical evaluator machines (even if they are different colors, note they are all the same types of machines), we feed the model into another hypothetical machine that operates the $\textsc{best}(M)$ function (which we call the *Max-Model Evaluators*). The nature and its description is also hypothetical but it should give an essence of the process. This is similar as to how the `for` loop is linearly looking for the max integer in a given array[7]. However we do not create a bunch of models, place them in an array or set, then feed into the model. Instead we build the model, train, valiadate and evaluate. We will mark the first model created as `best` by the max-model evaluator. We go back, and create another version of the model (i.e. change of problem definition, tweaking of parameters, etc.); then repeat (i.e. build, train, validate, and evaluate). Then compare this model with the model that was marked as `best`. If the newly created model's performance (perhaps using the *accuracy* metric) turns out to outperform the model marked `best`, replace it with the newly created version. Then repeat. Otherwise, we keep `best` as is. You can imagine that **Algorithm 3.1** should help explain this process, and we can do it as many times as we want. Piecewise represented by:

$$\textsc{best}(M) = \begin{cases} M & \text{if } \texttt{best} = \varnothing \vee \text{if } \textsc{performance}(M) > \textsc{performance}(\texttt{best}) \\ \texttt{best} & \text{otherwise} \end{cases} \tag{3.1}$$

To give a better understanding of **Algorithm 3.1**, we ask ourselves from Line 2: have we generated a model yet (for the certain problem type)? If not, then it just means that `best` $= \varnothing$ and that the

[6] The model that would solve Problem III in Attempt 1 is called **M1** and the model that would solve Problem II in Attempt 2 is called **M2**. We do not call them yet the **CLOCK** Model unless they have already outperformed **M_GAO**, the model name I give for X. Gao's model.

[7] In this metaphor, the `for` loop begins to look at the first number in the array or list and marks it the maximum. It will continuously scan the array until the end linearly; replace `max` when it has found an integer greater than the current max value holder. As a result, this gives off the largest integer in the array.

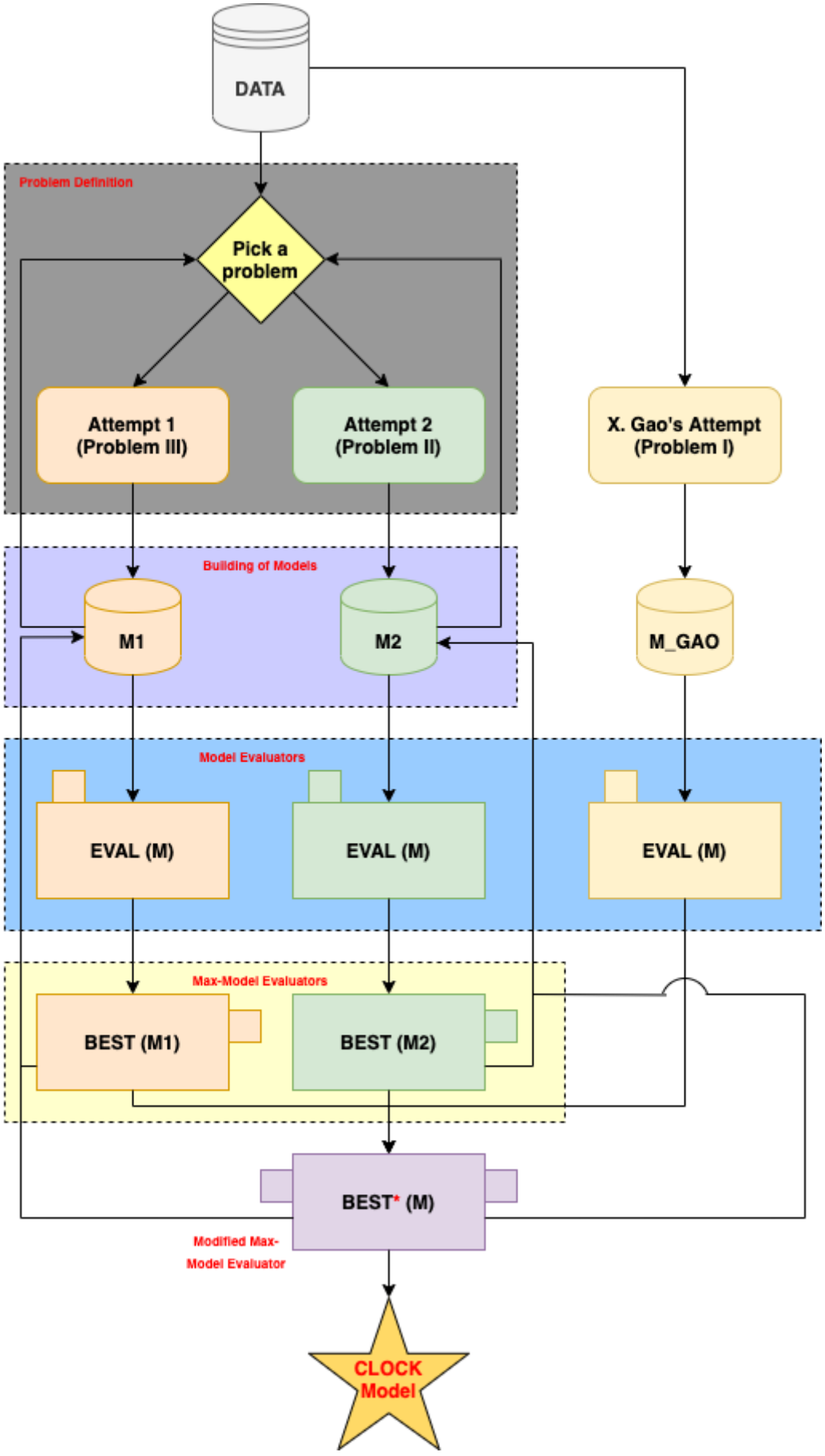


Fig. 11. The Basic Framework of the Methodology of the Project, made in `http://draw.io`

Algorithm 3.1: BEST (M)
Outputs the best model generated so far; returns the same model if no model has yet existed

Input: A model that classifies images according to the problem definition selected
Output: The model having the higher performance (we can use accuracy as a standard metric)

```
1       // if a model was created the first time, it becomes best automatically
2  if best = ∅ then
3      best ← M
4       // make best to be M if it outperforms the current best
5  if PERFORMANCE(M) > PERFORMANCE(best) then
6      best ← M
7  return best
```

first model we create becomes the `best`. Come the next models, for iteration $i = 2, 3, 4, ...$, then if the model generated turns out to be better than the current `best` performance-wise, then we replace the current `best` with this newly generated model, as suggested by Lines 5-6. So if you can imagine, the first model generated will not satisfy the condition in Line 5 (i.e. Lines 5-6 is not for the first model). And conversely, Lines 2-3 is not for the nth generated model for $n > 1$ as it is only for the first model generated. Expressed as a piecewise function in **Eq. (3.1)**, we just return the same model M if it so happens that the `best` has not yet existed (i.e. no model generated for the specific problem yet) or if the model M generated is better than the model `best` performance-wise. Otherwise, then the return is just `best`. This should be pretty much straightforward.

$$\text{BEST}^*(M) = \begin{cases} \arg\max\limits_{M} \text{ PERFORMANCE}(M) & \text{if } \left[\arg\max\limits_{M} \text{ PERFORMANCE}(M) \neq M_{GAO}\right] \\ \varnothing & \text{otherwise} \end{cases} \tag{3.2}$$

From here, we take in the best model that was used in solving Problem III (that will be the official **M1** model), the best model for solving Problem II (will be called the official **M2** model), and the **M_GAO**. We will take the best among these three but with a little caveat (using the hypothetical machine *Modified Max-Model Evaluator* that operates the BEST$^*(M)$ function as piecewise represented in **Eq. (3.2)**). If either **M1** or **M2** turns out to be the `best`, then it shall be known as the official **CLOCK** model; however, if this turned out to be **M_GAO**, then it just means that we did not beat the benchmark model and we would have to go back and either (1) change the problem, (2) rebuild the model, or (3) tweak the hyperparameters (and of course run the usual training, validation and testing/evaluation), then continue with the process until we finally find the **CLOCK** model.

Algorithm 3.2: BEST* (M)
Returns the **CLOCK** *Model, which is basically the better model between M_1 or M_2; but if it is beaten by M_{GAO} then return ∅*

Input: A set of models: $M = \{M_1, M_2, M_{GAO}\}$
Output: The model having the higher performance between M_1 and M_2 (accuracy for performance metric), but returns ∅ if M_{GAO} turns to be better than both M_1 and M_2

```
1       // initially set the clock and best_model as null
2  clock, best_model ← ∅
3       // get the better model among M1 and M2
4  if PERFORMANCE(M1) ≥ PERFORMANCE(M2) then
5      best_model ← M1
6  else
7      best_model ← M2
8       // compare the better model from M1 and M2 with M_GAO
9  if PERFORMANCE(best_model) > PERFORMANCE(M_GAO) then
10     clock ← best_model
11 return clock
```

Algorithm 3.2 gives a better idea of BEST*(M) function. We take in the three models $M = \{$**M1**, **M2**, **M_GAO**$\}$ as input set. We initially set variables `clock` and `best_model` as $\varnothing$, by Line 2. Then we compare the performance of models M_1 and M_2 to see which is better. We can see from Lines 4-7 that we intend to give some more importance for **M1** than **M2**. This is because that if there happens to be a tie between their performances (the odds of this happening is very slim), then M_1 should be model to be used since it has solved a harder problem (Problem III) than what M_2 had solved (Problem II). And now, we compare this better model with **M_GAO**. So if the better model between **M1** and **M2** has outperformed **M_GAO**[8], then this will be the official **CLOCK** model and just return it (as suggested by Lines 9-11). Otherwise, then since `clock` wasn't modified throughout the entire code, then it stays as $\varnothing$ and nothing happens. We just get `clock` $= \varnothing$ back. And if we get a $\varnothing$ model for **CLOCK**, we have to go back and do necessary modifications (i.e. switch problems, tweak hyperparameters, etc.) then redo building, training, validation, testing and evaluation, etc. of models in the process. The entire process just implies that we can keep on redoing and rebuilding our models until we have come up with a successful model that outperforms the benchmark model.

3.2 Programming

For all our programs, we will be using **R** language in an IDE called **RStudio**, which should be familiar to the reader. Consult the documentation `https://docs.rstudio.com/` if necessary.

We will also be using Keras, the neural network library in R, as discussed in **Sec. 1.4.3**.

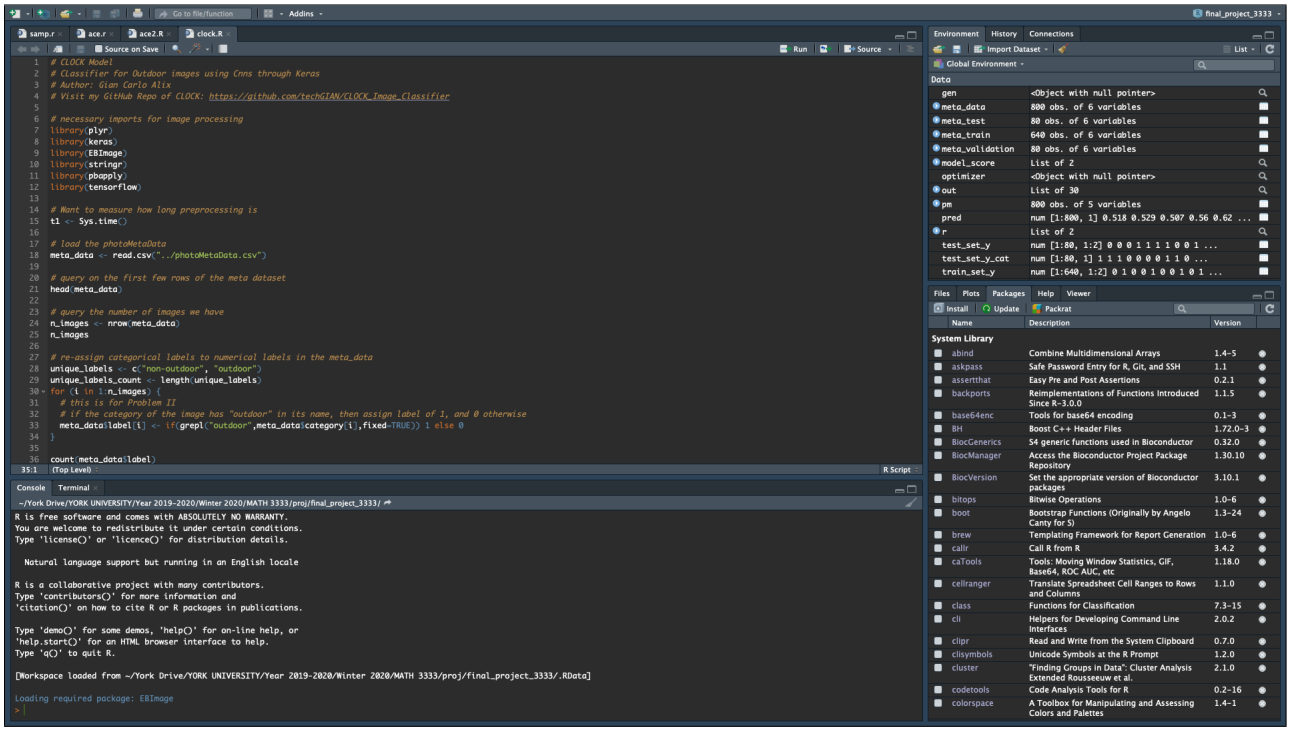


Fig. 12. A Sample RStudio Interface

[8] Either **M1** or **M2** should beat **M_GAO** performance-wise and not just be of equal performance in order to be named the **CLOCK** model.

4 The Models

4.1 The Benchmark Model

In **Sec. 1.5**, we have defined what a *good* model is. Suppose there exists some accuracy threshold μ. If our model can have a higher accuracy than μ, then we just have produced a successful model to be named as **CLOCK**. Otherwise, we try again building another model. In our case, this accuracy threshold μ turns out to be the accuracy of the Gao Model. Since we would like to outperform this model, then its accuracy serves as the accuracy threshold that our models would like to beat. Thus, the Gao Model serves as our benchmark model for producing the **CLOCK** Model.

Let us take a look into the Gao Model, which is **Code 4.1** and can be found on **p. 23** (also on **Appendix B** of this paper). All lines from Line 1-19 of the code is the preprocessing of data, which has already been detailed in **Sec. 2.3**. Hence we will no longer be going over this[9]. Now as the Gao model is based from a logistic model, then it has used R's native `glm()` function, or the *generalized linear model function*. Lines 20-23 does this and outputs the result:

```
================================================================================
[1] 5
Call:
glm(formula = y ~ X, family = binomial, subset = trainFlag)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.2744  -0.7483  -0.4592   0.8643   2.5681

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -2.1258     0.3173  -6.699 2.09e-11 ***
X1           -6.9388     2.1315  -3.255  0.00113 **
X2            2.1553     3.2821   0.657  0.51140
X3           10.6953     2.1844   4.896 9.77e-07 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 548.55  on 431  degrees of freedom
Residual deviance: 419.78  on 428  degrees of freedom
AIC: 427.78

Number of Fisher Scoring iterations: 5
================================================================================
```

To give a better perspective of the Gao Model, for each image that has been read (using the `readJPEG` library), then take the median of all pixel values of each color channel (three color channels, one each for RGB), then those three medians will serve as the three predictors (`X1`, `X2`, `X3`) of the logistic model. It's that simple, although can be long in code.

For the R output above, we get 5 as the number of iterations for the `out` variable (the `glm` variable). So Line 21 simply tells us that the predictors are stored in a vector `X` and the label in `y`. The binary classification is as specified by `family = binomial`. And the `subset=trainFlag` tells us that we should apply it only on the training set (also note that the Gao Model has partitioned the image set into training and testing sets only, without the validation; we'll see later how validation sets are actually important). Now in Line 23, we see the summary of `out`, the `glm` variable. That's the overall model. We'll look into the evaluation of benchmark model in **Sec. 5.1**.

[9] We will no longer go over the data preprocessing on the benchmark model. Also note that as the entire code is directly taken from X. Gao's website (see the comments of the code for the exact link), then the directory path of the files would have to be changed when we try importing and running it in our machines.

Code 4.1: The R Script of the Benchmark Model

```
### Professor X. Gao's Sample R Code
### Original Source can be found at https://xingao.info.yorku.ca/files/2020/01/
###                                     math3333projectsamplecode.txt
### Photograph examples

## Read in the library and metadata
library(jpeg)
pm <- read.csv("C:\\Users\\xingao\\Documents\\teaching\\winter2020\\math3333\\photoMetaData.csv")
n <- nrow(pm)

trainFlag <- (runif(n) > 0.5)
y <- as.numeric(pm$category == "outdoor-day")
X <- matrix(NA, ncol=3, nrow=n)
for (j in 1:n) {
  img <- readJPEG(paste0("C:\\Users\\xingao\\Documents\\teaching\\winter2020\\math3333\\columbia
                         Images\\columbiaImages\\",pm$name[j]))
  X[j,] <- apply(img,3,median)
  print(sprintf("%03d / %03d", j, n))
}

# build a glm model on these median values
out <- glm(y ~ X, family=binomial, subset=trainFlag)
out$iter
summary(out)

# How well did we do?
pred <- 1 / (1 + exp(-1 * cbind(1,X) %*% coef(out)))
y[order(pred)]
y[!trainFlag][order(pred[!trainFlag])]

mean((as.numeric(pred > 0.5) == y)[trainFlag])
mean((as.numeric(pred > 0.5) == y)[!trainFlag])

## ROC curve (see lecture 12)
roc <- function(y, pred) {
  alpha <- quantile(pred, seq(0,1,by=0.01))
  N <- length(alpha)

  sens <- rep(NA,N)
  spec <- rep(NA,N)
  for (i in 1:N) {
    predClass <- as.numeric(pred >= alpha[i])
    sens[i] <- sum(predClass == 1 & y == 1) / sum(y == 1)
    spec[i] <- sum(predClass == 0 & y == 0) / sum(y == 0)
  }
  return(list(fpr=1- spec, tpr=sens))
}

r <- roc(y[!trainFlag], pred[!trainFlag])
plot(r$fpr, r$tpr, xlab="false positive rate", ylab="true positive rate", type="l")
abline(0,1,lty="dashed")

# auc
auc <- function(r) {
  sum((r$fpr) * diff(c(0,r$tpr)))
}
glmAuc <- auc(r)
glmAuc
```

4.2 Attempt 1: Model 1

For building Model 1, which is the "Attempt 1", the only difference it has with Model 2 is the problem definition (and consequently the codes of the preprocessed data). All the other codes are the same (the neural networks architectures, the hyperparameters, etc.) So instead of covering the details here, I would do so in **Sec. 4.3** when covering the details of Model 2.

4.3 Attempt 2: Model 2 (The CLOCK Model)

As you guessed it, from the title of this section, Model 2 will be our official **CLOCK** Model and **Code 4.2** displays the R script of this Model. It is also displayed in Lines 97-173 of the code in the **Appendix C** of the paper. Let us cover a few of the lines of code (referencing **Code 4.2**). In line 3, we begin by initializing Keras' convolutional network through a sequence of linear layers in the form of a stack data structure. We store this in a variable called `cnn_model`.

Lines 7-23 are the tricky bits, where we set the hyperparameters of the CNN and NN. I have already done multiple testings (setting parameters → building and evaluating model → comparing with the benchmark → then re-sets the parameters → etc.) so I know that these hyperparameters are the ones that give a good accuracy. But yes, we can still keep on tweaking until we ensure that our model would be as superior as it would than the benchmark model (and perhaps even tweak it more to surpass the performance of the current **CLOCK** model).

We begin by setting the filter size to be 100 (similar to the example from **p. 7**). The kernel dimension is composed of the kernel's width × height, which we can expressed as a vector. The kernel width and height are both 3 in this case.

```
================================================================================
Code 4.2: The R Script of the CLOCK Model
================================================================================
 1. # we build the architecture of our convolutional neural network
 2. # begin by defining a sequential linear stack of layers
 3. cnn_model <- keras_model_sequential()
 4.
 5. # set the hyperparameters
 6. # this can be tweaked to improve the accuracy of the model
 7. filter_size <- 100
 8. kernel_width <- 3
 9. kernel_height <- 3
10. kernel_dim <- c(kernel_width, kernel_height)
11. input_shape <- c(new_width, new_height, 3)
12. activation_func_1 <- "relu"
13. pool_width <- 2
14. pool_height <- 2
15. pool_dim <- c(pool_width, pool_height)
16. s_strides <- 3
17. dropout_val1 <- 0.8
18. dropout_val2 <- 0.5
19. hidden_units_1 <- 1000
20. output_units <- unique_labels_count
21. activation_func_2 <- "softmax"
22. learning_rate <- 0.0001
23. learning_decay <- 10^-6
24.
25. cnn_model %>%
26.   # 1st convolutional layer
27.   layer_conv_2d(filter=filter_size, kernel_size=kernel_dim, padding="same",
                    input_shape=input_shape) %>%
```

```
28.   layer_activation(activation_func_1) %>%
29.
30.   # Use a max pool layer to dimensionally reduce the feature map to reduce
31.   # the model's complexity
32.   layer_max_pooling_2d(pool_size=pool_dim) %>%
33.
34.   # Using a dropout to avoid overfitting
35.   layer_dropout(dropout_val1) %>%
36.
37.   # need to flatten input
38.   layer_flatten() %>%
39.
40.   # create a densely-connected neural network with 1000 hidden units with the given
41.   # the given activation function relu (hidden layer) and a 0.5 dropout, then for
42.   # the output layer use softmax (to calculate cross-entropy)
43.   layer_dense(hidden_units_1) %>%
44.   layer_activation(activation_func_1) %>%
45.   layer_dropout(dropout_val2) %>%
46.   layer_dense(output_units) %>%
47.   layer_activation(activation_func_2)
48.
49. optimizer <- optimizer_adamax(lr=learning_rate, decay=learning_decay)
50.
51. cnn_model %>%
52.   compile(loss="binary_crossentropy", optimizer=optimizer, metrics="accuracy")
53.
54. # a summary of the architecture
55. summary(cnn_model)
56.
57. # We then train our model
58. data_augment <- TRUE
59. batch_size <- 100
60. n_epochs <- 100
61. if (!data_augment) {
62.   cnn_model %>%
63.     fit(train_set_x, train_set_y, batch_size=batch_size, epochs=n_epochs,
          validation_data=list(validation_set_x, validation_set_y), shuffle=TRUE)
64. } else {
65.   # Generate Images
66.   gen <- image_data_generator(featurewise_center=TRUE,
                                  featurewise_std_normalization=TRUE)
67.
68.   # Fit the image data generator to the training data
69.   gen %>% fit_image_data_generator(train_set_x)
70.
71.   # Generate batches of augmented/normalized data from image data and labels to
72.   # visualize the generated images made by the CNN Model
73.   cnn_model %>%
74.     fit_generator(flow_images_from_data(train_set_x, train_set_y, gen,
                     batch=batch_size),
                     steps_per_epoch=as.integer(train_sample_size/batch_size),
                     epochs=n_epochs,
                     validation_data=list(validation_set_x,validation_set_y))
75. }
================================================================================
```

The input shape to use is expressed as a 3D vector, with parameters `new_width` and `new_height`, and 3. The first two parameters here represent each of the image's rescaled widths and heights (32×32 as we set earlier in **Code 2.3** from Data Preprocessing), and the third hyperparameter here represents the number of color channels (3 in this case)[10].

The activation to use for our CNN and NN architectures is the Rectified Linear Unit (ReLU), defined by **Eq. (1.1)**, but we use the softmax activation function for the final layer of the neural network, defined by:

$$g(\mathbf{y}) = \frac{e^{y_i}}{\sum_{j}^{n} e^{y_j}}, \qquad \text{for } i = 1, 2, ..., n \tag{4.1}$$

which will output the probability of each class (since we only have two possible classes in Problem II, then this should output two probabilities).

And then the pooling layer's width and height were both set to 2, which would be used to form the dimensions of the pooling layer. We express this `pool_dim` in a form of a 2D vector with `pool_width` and `pool_height` as their parameters.

Then the parameter `s_strides`, describes the number of strides there are in the convolutional network. This was not mentioned in **Sec. 1.4.2**, as the default value for the number of strides is 1; however, it basically just describes the number of jumps the filter makes when it linearly scans the layer horizontally. So for an `s_strides = 1`, then the kernel would have to shift linearly to the right horizontally by 1 unit each time it needs to move. In our architecture, we set the strides to be equal to 3.

The dropout value is basically a hyperparameter that determines the percentage of the number of features that we drop during training, as they are placed at little significance (or not significant at all). So to help training, we drop them. We have two dropout values to use for the architectures, the first is 0.8, and then the next is 0.5.

Then for our hidden layer (we will only use one hidden layer for our NN architecture), this layer will have 1000 hidden units. For the number of output units, we should give it the number of unique class labels there are. Problem III has eight, but Problem II only has two. So we should have two output units. These two output units will undergo the softmax activation as discussed earlier, utilizing the formula in **Eq. (4.1)**.

For the learning rate, we will use 0.0001, which describes the rate of how fast the algorithm will be learning; the decay is an additional parameter for this and we set it to be 10^{-6}.

Now let us go over the architecture. Firstly, note that the `>%>` symbols means that it is a pipeline. In other words, given that

```
A >%> B
```

then the output of `A` becomes the input of `B`.

[10] Since all images we know will be colored anyway, we can always set the third parameter from Line 11 as fixed to be 3. Although one may opt to write another line of code before it to extract the number of channels for each of the images (given that there may be grayscale images within the image set), then placing them in a variable and using it as a third parameter. This is for code robustness purposes.

Algorithm 4.1: ADAM $(\epsilon, \rho_1, \rho_2, \delta, \boldsymbol{\theta})$
"Adaptive Moments" (Adam Algorithm) for Neural Network Optimization (Kingma & Ba, 2014)

Input:
→ Step size ϵ (default is 0.001)
→ Exponential decay rates ρ_1 and ρ_2 (default values are 0.9 and 0.999 respectively)
→ Small δ (default is 10^{-8})
→ Initial parameter vector $\boldsymbol{\theta}$

1 // initialize 1st and 2nd moment variables as zero vectors, then a time step t too
2 $\boldsymbol{s} \leftarrow \boldsymbol{0}$, $\boldsymbol{r} \leftarrow \boldsymbol{0}, t \leftarrow 0$
3 **while** ¬ stopping criterion met **do**
4 Sample a minibatch of m examples from the training $\{\boldsymbol{x}^{(1)}, ..., \boldsymbol{x}^{(m)}\}$ with targets $\{\boldsymbol{y}^{(1)}, ..., \boldsymbol{y}^{(m)}\}$
5 Compute gradient $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
6 $t \leftarrow t + 1$
7 Update the biased first moment estimate $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
8 Update the biased the second moment estimate $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
9 Correct bias in first moment $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1-\rho_1^t}$
10 Correct bias in second moment $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1-\rho_2^t}$
11 Compute update $\Delta\boldsymbol{\theta} \leftarrow -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$
12 Apply update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

Lines 27-28 creates the first layer of our `cnn_model`, which is the convolutional layer. We make use of Keras' built in `layer_conv_2d` function, and using the discussed hyperparameters to set its filter size, the kernel size, the input shape and the ReLU activation function. We also use `"same"` as its padding value. Line 32 makes use of a two-dimensional max pooling layer (using Keras' `layer_max_pooling_2d` function) with the `pool_dim` as the pool size. To reduce dimensions out of unnecessary features and to avoid overfitting[11] of data. And then to finish the CNN architecture, we flatten the layers, as seen in Line 38.

Then lines 43-47 is the architecture's NN layers. We pass on the output of the CNN architecture to that of the NN architecture using the `>%>` symbol. The function `layer_dense` in Keras creates a (hidden) layer of `hidden_units_1 = 1000` hidden units using a ReLU activation function and a 0.5 dropout value. Then the next `dense` layer is for the output layer with only two units as we only have two classes. We utilize the softmax activation function here.

Then we need to identify the optimizer for the architecture. There are several optimizers in Keras that is available for use, and this is why choosing the right optimizer is difficult and which is why it itself is considered a hyperparameter. Each optimizer has its own hyperparameters as well, and usually the learning rate and decay are the most common hyperparameters to tweak here. [5, p. 298-302] The most popular optimizers include Adam, AdaGrad, AdaDelta, Momentum, RMSProp, and Stochastic Gradient Descent (or SGD). We would not be going over each of these algorithms in detail; instead we can see from **Algorithm 4.1**, the [5, p. 301] ADAM (or "Adaptive Moments") Algorithm. In our code from **Code 4.2** in Line 49, we have seen that the optimizer ADAMAX was used instead. This is simply a modified version of the ADAM; however this one is based on the L_∞ or the max norm.

And then we compile our `cnn_mnodel` using the binary cross entropy loss (optimal for two binary variables), using the ADAMAX optimizer and take accuracy as the performance metric. Then we ask R to summarize the architecture or model for us (results of the summary is seen on **p. 28**). To give a brief interpretation of what is displayed, there are a total possible of 25,605,802 possible hyperparameters (massive!). And then the summary display also lists the different layers of the CNN and NN architectures, along with the shapes or dimensions of their output layer.

[11] Overfitting is a common machine learning linggo that describes how a particular model's training accuracy does really well but on testing and new data does poorly. This suggests that there is a better alternative model that although does not perform relatively well compared to the overfitted model in the training set; however does really well and much better in the new and testing data. This is a common issue in machine learning and that this needs to be always addressed.

```
=================================================================================
Layer (type)                                  Output Shape
=================================================================================
conv2d (Conv2D)                              (None, 32, 32, 100)
---------------------------------------------------------------------------------
activation (Activation)                      (None, 32, 32, 100)
---------------------------------------------------------------------------------
max_pooling2d (MaxPooling2D)                 (None, 16, 16, 100)
---------------------------------------------------------------------------------
dropout (Dropout)                            (None, 16, 16, 100)
---------------------------------------------------------------------------------
flatten (Flatten)                            (None, 25600)
---------------------------------------------------------------------------------
dense (Dense)                                (None, 1000)
---------------------------------------------------------------------------------
activation_1 (Activation)                      (None, 1000)
---------------------------------------------------------------------------------
dropout_1 (Dropout)                            (None, 1000)
---------------------------------------------------------------------------------
dense_1 (Dense)                                (None, 2)
---------------------------------------------------------------------------------
activation_2 (Activation)                      (None, 2)
=================================================================================
Total params: 25,605,802
Trainable params: 25,605,802
Non-trainable params: 0
=================================================================================
```

Now that we have built our neural network architectures, it is time to train our model. Since we would also like to do data augmentation, we set a variable called `data_augment` to be `TRUE` as seen in Line 58. And then in lines 59-60 are also more hyperparameters that need to be set: the *batch size* and the *number of training epochs*. The number of training epochs can simply be thought of as the number of iterations in training the model and this number should be higher if we have a higher learning rate. The more epochs there are, the more time (in terms of t time steps) the model can do learning, and this is why it needs a higher learning rate if we are to have more training epochs. In this case, we'll have 100 epochs.

For the batch size, this determines how many training samples there are per training epoch. This method of training in batches is known as **minibatch training**, and the reason we do so is [5, p. 272] because small batches offers an effect called *regularization* - which is due to the noise that they add during the learning process (Wilson and Martinez, 2003). This noise is added to the generalization or testing error, without the training error being impacted. I have not mentioned **regularization** in the introduction section, however it is sinply a common machine learning term that [10, p. 10] suggests penalizing the error loss function in order to avoid the overfitting of the model. In our case, we set the minibatch size to be 100.

And then we use Keras' `image_data_generator()` function to be able generate the data images. We have `featurewise_center` and `featurewise_std_normalization` set to `TRUE`. This just means that Keras is generating new data images from the set available and then applying some modifications and transformations to add to the set. Then Keras makes use of the `fit` and `fit_image_data_generator` functions to do the image fitting on to the model. And then finally, Keras' `fit_generator()` function generates batches of augmented and normalized data from the image data and labels to visualize the generated images made by the `cnn_model`.

Algorithm 5.1: BASIC_MODEL (M, D)
The pseudocode for a typical model for any Machine Learning Task

Input: A model M and a dataset D
Output: The testing accuracy of M, denoted by μ, along with the results of the ML task

```
1       // preprocess the dataset D; pd is composed of the train and testing sets
2 pd ← PREPROCESS(data=D)
3       // train the model given the preprocessed training set
4 TM ← TRAIN(model=M, data=pd[train_set])
5       // test the trained model on the preprocessed test set using the accuracy metric
6       // store the model's accuracy, along with any other results, in the variable mu
7 μ ← TEST(model=TM, data=pd[test_set], metric="accuracy")
8 return μ
```

5 Analysis, Results and Discussions

In **Algorithm 5.1**, we can see the pseudocode for the basic typical model for any machine learning task, given a model M and dataset D. In all models **M1**, **M2** (or **CLOCK**) and **M_GAO**, they all have basically followed this structure: data preprocessing, training (validation is sometimes included in the training) and then testing. So far we have already covered the preprocessing and training sections. Now we would like to talk about the testing part. How well did the model do? We would be talking about the performance of the models.

5.1 Testing and Performance

5.1.1 Testing

Let us begin having a look at the testing for the benchmark model. In lines 26-28 of **Code 4.1** on **p. 23**, we can see this happening when predictions for the testing set are done; and the predictions are done using the logistic model that the model used. Mathematically, it is expressed as:

$$\log\left(\frac{\mathbf{p}}{1-\mathbf{p}}\right) = \mathbf{X}^T\boldsymbol{\theta} \tag{5.1}$$

and this logarithmic function in **Eq. (5.1)** can also be rewritten exponentially as:

$$\mathbf{p} = \frac{1}{1+e^{-\mathbf{X}^T\boldsymbol{\theta}}} \tag{5.2}$$

If non-negative exponents are preferred, then rewrite **Eq. (5.2)** as:

$$\mathbf{p} = \frac{e^{\mathbf{X}^T\boldsymbol{\theta}}}{1+e^{\mathbf{X}^T\boldsymbol{\theta}}} \tag{5.3}$$

which is known as the **sigmoid function** and a sketch of its graph was seen earlier in **Fig. 4**. So in some sense, the activation function used by **M_GAO** makes use of the sigmoid function. Then the result of each p_i in $\mathbf{p}$ is the prediction that the model makes. So for instance if value of p_i turns out to be 1, then that particular image is said to have `outdoor-day` as class, for that particular i. However if it was 0, then its class is not `outdoor-day`. This was Problem I if you would recall, and this was the problem that the benchmark model had solved.

The machine makes it fast, quick and easy to do these calculations. Let's see if we could work out an example by hand. We can use **Eq. (5.2)** to calculate the particular p_i value using a specific X_i^T value. Suppose we would like to know the prediction p_i of X_1. We first query in R what are the predictors for X_1. Recall that the method applied for the Gao Model is that the median of each RGB color channels were obtained in order to serve as the three predictors. So we get:

```
================================================================================
> X[1,]
             [,1]        [,2]        [,3]
  [1,] 0.427450980 0.400000000 0.376470588
================================================================================
```

Recall the calculated $\boldsymbol{\theta}$ from **p. 22**. Hence, we compute $-X_1^T\boldsymbol{\theta}$ is equal to:

$$-X_1^T\boldsymbol{\theta} = -(1, 0.427450980, 0.400000000, 0.376470588)\begin{bmatrix}-2.1258\\-6.9388\\2.1553\\10.6953\end{bmatrix} = 0.20321098 \tag{5.4}$$

And thus finally solve for p_1:

$$p_1 = \frac{1}{1+e^{0.20321098}} = 0.449371359 \tag{5.5}$$

Lines 30-31 of the benchmark code suggests that we use a cutoff value of 0.5. This just means that all values of $p < 0.5$ gets assigned to the `0` label and `1` otherwise. As you could imagine, this is similar to the integer rounding function:

$$g(x) = \text{int-round}(x) = \begin{cases}\lceil x \rceil & \text{if } x - \lfloor x \rfloor \geq 0.5\\ \lfloor x \rfloor & \text{otherwise}\end{cases} \tag{5.6}$$

where $\lfloor x \rfloor$ represents the **floor function**[12] and $\lceil x \rceil$ represents the **ceiling function**[13]. So for instance, feeding 2.3 in this function gives 2, but feeding 2.9 gives 3. This should work even for inputs less than 0 (i.e. negatives, which can easily be tested). However since our $p_i \in (0, 1)$ [14], then **Eq. (5.6)** reduces to:

$$g(p_i) = \text{int-round}(p_i) = \begin{cases}1 & \text{if } p_i \geq 0.5\\ 0 & \text{otherwise}\end{cases} \tag{5.7}$$

Hence in our case where we calculated the predicted output by hand, since $p_1 = 0.449371359 < 0.5$, then the prediction is said to be 0.

This is basically how the model does the testing for the Gao Model. It applies the model used in training onto the testing set that was set aside during the data preprocessing. Let us now look at the testing for the **CLOCK** model.

Now due to the complexity of the architecture, it would be highly impossible to do and test an example by hand, just like how we just did it for the Gao Model. In fact, calculating one example for the benchmark model also took a while with the multiple calculations, but that was doable, just tedious. It would be a different case for the **CLOCK** Model though; for instance the fact that there are 1000 hidden neurons in the hidden layer should already say that it is infeasible for humans to compute by hand. On top of it, the several stacks of abstract layers there are in the network architecture should also suggest that testing the examples by hand should not be done.

In **Code 5.1**, we get a glimpse of testing the **CLOCK** Model. I would be covering as much as I could including the code and how it works; however I'd like to talk more about the accuracy and performance of this model in the next section under **Sec 5.1.2**.

Now, Keras has a built-in function for evaluating neural network models, called `evaluate()`. We pass on this `cnn_model` that we trained during the Training Phase onto the Keras' function `evaluate()` using the testing set. The verbose setting here just tells us if there would be an output displayed for each epoch. In our case, we would not want it to do that, so we set it to 0 for "silent". We store the result into a variable called `model_score`, as seen in Line 2.

[12] The floor function takes any number as input and outputs the greatest integer that is less than or equal to this input.

[13] The ceiling function takes any number as input and outputs the least integer that is greater than or equal to this input. As a remark, $\lceil x \rceil = \lfloor x \rfloor + 1$.

[14] As a remark, due to the formula for the logistic, we cannot have a value of $p_i = 0$ since our numerator in **Eq. (5.2)** is nonzero. Furthermore, we cannot have a value of $p_i = 1$ either as the only way for this to happen is if the denominator is equal to 1; however, this is again not possible as the exponential function is always positive and thus the denominator is always greater than 1. We can verify that this is the case with the sigmoid graph in **Fig. 4** that the function is asymptotic at $y = 0$ and $y = 1$.

```
================================================================================
Code 5.1: Testing Code for the CLOCK Model
================================================================================
 1. # Testing and Check Accuracy
 2. model_score <- cnn_model %>% evaluate(test_set_x, test_set_y, verbose=0)
 3. cat("Test Loss: ", model_score$loss, "\n")
 4. cat("Test Accuracy: ", model_score$acc, "\n")
 5.
 6. # View one test img (simple random sample)
 7. r <- sample(1:length(not_tr_te),1)
 8.
 9. img_name <- as.character(meta_test[r,]$name)
10. filename <- paste("../columbiaImages/", img_name, sep="")
11. display(readImage(filename))
12.
13. category_predictions <- cnn_model %>% predict_classes(test_set_x)
14. p <- unique_labels[category_predictions[r]+1]
15. a <- unique_labels[as.numeric(test_set_y_cat[r])+1]
16.
17. # determines whether the model was able to predict the class of the image correctly
18. cat(img_name, "\n")
19. cat("Predicted:", p, "\n")
20. cat("Actual:", a, "\n")
21. match <- p == a
22. cat("The model predicted it correctly:", match, "\n")
================================================================================
```

And then we ask Keras to display the testing accuracy and testing loss in Lines 3-4. I will distinguish these two terms, as well as relate it to the term *error* in **Sec 5.1.2**. For now, just know that accuracy and loss are inversely proportional (i.e. a higher accuracy implies a lower loss, which is desirable).

In the next lines from Lines 6-22, we want to take a random image from the testing image set. Then we shall predict using the model whether it is an `outdoor`-type or not (i.e. if it belongs to the class of `outdoor-day`, `outdoor-dawn-dusk`, `outdoor-rain-snow`, `outdoor-night`, or not) and solve Problem II. Then compare the prediction with the actual classification. In Line 7, we get a random index value; then grab the image from the test set having that particular random index. We display the image in the R Viewer to get an idea of which image was randomly selected. Then we predict the class label of this particular image using the model in Line 13 and store it in a variable `p` in Line 14. We store the actual class label in `a` as shown in Line 15. We display the results thereforth (the image filename, the predicted class label, the actual class label, and if it was correctly matched or not) as seen in Lines 18-22.

As in any science experiment, it is good to run tests more than once in order to verify that the results are indeed genuine and not just a fluke. That said, I decided to run the **CLOCK** Model at least three times (Run 1, Run 2 and Run 3, which we can shorten as R_1, R_2 and R_3). Let's take first the example done by R_1. The randomized image from the testing set turned out to be the sample image from **Fig. 9**, which we can recognize as an `outdoor`-type image, and the metadata shows that this is image is indeed so because its category is `outdoor-day`. So during R_1, we have this result:

```
================================================================================
CRW_4786_JFR.jpg
Predicted: outdoor
Actual: outdoor
The model predicted it correctly: TRUE
================================================================================
```

In R_2, another image was randomly selected, which turned out to be a `non-outdoor`-type image. We can see this image with filename `CRW_4890_JFR.jpg` in **Fig. 13**. We can verify by eye that this is indeed not in the outdoor, and from the metadata this has label `indoor-light`. The output is:

Fig. 13. An image selected randomly, with filename `CRW_4890_JFR.jpg`

```
================================================================================
CRW_4890_JFR.jpg
Predicted: non-outdoor
Actual: non-outdoor
The model predicted it correctly: TRUE
================================================================================
```

In both runs, our model was lucky enough to predict them both correctly. We can keep playing this game of testing one image after the other (R_3 for instance is an outdoor image and it was able to predict correctly). However, it is also important to note that we are only testing one image after the other. We have to test each of the images one at a time from the image set (supposedly 80 in this case) in order to get a bigger picture of how well our model did. And doing this manual testing of images one at a time isn't very efficient in terms of time. Hence, we may instead want to use different performance metrics, such as the *accuracy*, in order to know whether our model did great in classification or was it simply by chance that it was able to predict these three randomly selected images correctly? Luckily, the `evaluate()` function by Keras as mentioned before should help us in determining so, by the model's `model_score`.

5.1.2 Accuracy and Other Relevant Performance Metrics

Before we talk about the performance of our models, we first go over the basics of accuracy and other relevant performance metrics. In a nutshell, the **accuracy** of a model M is calculated as follows:

$$\text{ACCURACY}(M) = \frac{|\text{correctly predicted}|}{|D|} \tag{5.8}$$

where $|D|$ is the number of observations in the dataset D and $|\text{correctly predicted}|$ is the number of these that are correctly predicted in D. In other words, the percentage of how many observations out of D were predicted correctly by M serves as the accuracy of M.

As an example, let us take some hypothetical trained model M. We then apply this model on to a small mini sample dataset D, such as the one in **Table 7**. Now notice that the last two columns of this dataframe is the **class label** and the **predicted label**. The number of correct predictions M makes is the number of observations in this dataframe where the actual class value is equal to the predicted value of the model. We can count how many observations were correctly corrected. All rows highlighted in pink are those observations that were misclassified; there are three here. And since there are ten examples in this mini dataset, then there should be seven items in the dataset that are classified correctly. By **Eq. (5.8)**, then the accuracy of this model M is $\frac{7}{10}$ or 70%.

During machine learning tasks of training, cross-validation and testing, we usually measure a model's performance through the accuracy metric. Hence, during training, we can measure a model M's performance through the *training accuracy*. Meanwhile, during the cross-validation process, we get the *validation accuracy* to validate the training accuracy. It may be possible that the model can only be

ID	a_1	a_2	a_3	class	pred
0001	v_1	v_2	v_3	0	0
0002	v_4	v_5	v_6	0	0
0003	v_7	v_8	v_9	1	0
0004	v_{10}	v_{11}	v_{12}	1	1
0005	v_{13}	v_{14}	v_{15}	1	1
0006	v_{16}	v_{17}	v_{18}	0	0
0007	v_{19}	v_{20}	v_{21}	1	0
0008	v_{22}	v_{23}	v_{24}	0	0
0009	v_{25}	v_{26}	v_{27}	0	1
0010	v_{28}	v_{29}	v_{30}	1	1

Table 7. A small mini sample dataset D having 3 attributes and a class label; some model M makes predictions on the classes of each observation as shown. Rows highlighted in pink have misclassification prediction.

"good" in terms of performance with the dataset it saw and does horribly with dataset that it has never seen yet. Recall this problem as the *overfitting problem*, which happens when there exists some alternative model M' that attempts to solve the same machine learning task T that satisfies the condition:

$$\text{TRAIN\_ACC}(M) > \text{TRAIN\_ACC}(M') \wedge \text{TEST\_ACC}(M) < \text{TEST\_ACC}(M') \tag{5.9}$$

which just implies that M doesn't do well in terms of performance on newly seen (or generalized) data, compared to some other alternative model M' that solves the same task. And then finally, during the testing phase, the *testing accuracy* can be obtained (also called the *generalization accuracy*), as we would like to test our model M on dataset we have not yet seen before.

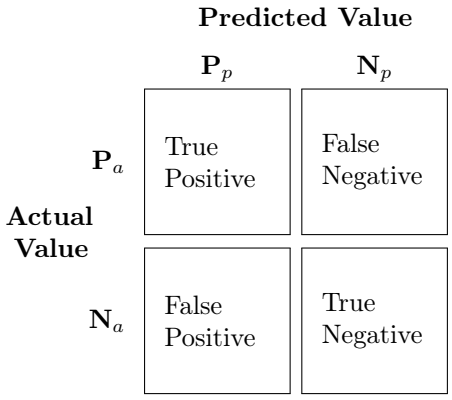
A direct performance measure that is relevant to the accuracy metric is the **error** metric, or **misclassification error**. We can easily define as such using the direct formula, given the accuracy of model M:

$$\text{ERROR}(M) = 1 - \text{ACCURACY}(M) \tag{5.10}$$

As $\text{ACCURACY}(M)$ can be expressed as a percentage (a number between 0 and 1), then the $\text{ERROR}(M)$ can also be expressed as a percentage (a number between 0 and 1) as well. As a remark, the formula from **Eq. (5.10)** can also be rewritten as:

$$\text{ERROR}(M) = 1 - \text{ACCURACY}(M) = 1 - \frac{|\text{correctly predicted}|}{|D|} = \frac{|\text{incorrectly predicted}|}{|D|} \tag{5.11}$$

where |incorrectly predicted| is the number of observations in the dataset that was misclassified by the model M. Thus, in **Table 7**, the highlighted pink rows are the misclassified observations (three); so the misclassification error of M is equal to $\frac{3}{10}$ or 30%.



From **p. 33**, we can see what is known as a **confusion matrix**, and the size of this matrix is $n \times n$, for a classification problem with n classes. In other words, we can still come up with a confusion matrix for multi-class classification problems. In our case, both the benchmark model and the **CLOCK** Model, have solved Problems I and II, which are classification problems that have two class labels. In that case then our confusion matrix will have a dimension of 2×2.

Usually in a confusion matrix, the column values depict the predicted outcomes and the row values depict the actual labels. In a 2×2 setting, we usually indicate P for the positive and N for the negative labels. These can be 1 and 0 instead of P and N; but sometimes it does not really matter and can be the other way around, as long we are consistent along the way. So for instance, if a particular machine learning problem talks about cancer cells, then the positive (or 1) can indicate the presence of cancer cells and negative (or 0) for the absence of such. However, on the flipside, another representation of such could be that a healthy human being can be represented as positive (or 1) and a human having the particular illness or disease is represented negatively (or 0). In the Gao Model, 1 has been used as an indicator of the particular image being of class `outdoor-day`, and 0 if it is not. In the **CLOCK** Model, 1 is used to indicate that a particular image is of class type `outdoor`, and 0 otherwise.

Let's have a look once more in the confusion matrix in **p. 33**. The positive P and negative N values can be further split into P_p, P_a, N_p and N_a. This just means that for a predicted value, the prediction can either be positive or negative. Similarly, the actual label on each of the observations can also be either positive or negative. We may be interested in determining what percentage of the dataset D the machine model M has predicted P on observations that were actually labelled P. This percentage is what we call the *True Positive*, or TP. This is not exactly the accuracy, as the accuracy computes for the percentage of the dataset D that was correctly classified by M and some could include correctly identified negative values. However, TP goes further than that. Among all those observations correctly classified by M, how many of them are labelled 1 (or positive)? If we were to position the predicted values as columns and actual values as rows, then the TP count (or percentage) should appear on the top left corner of the matrix. Similarly, the *True Negative*, or TN is similar to the true positive, except that these are observations predicted and actually labelled to be negative (or 0). This is located at the bottom right corner of the confusion matrix. Because M can never be perfect, it can also make mistakes (i.e. errors). *False Positives*, or FP, refers to those observations that have been predicted by the model to be positive (or 1), when in fact these observations are actually labelled as negative (or 0). We place this count or percentage at the lower left corner of the matrix. Analogously, M can predict observations to be negative (or 0) when they are in fact positively labelled (or 0). We place this count or percentage at the top right corner of the matrix, and we call this *False Negatives*, or FN.

		Predicted Value	
		1	**0**
Actual Value	**1**	TP $= \frac{3}{10}$	FN $= \frac{2}{10}$
	0	FP $= \frac{1}{10}$	TN$= \frac{4}{10}$

Suppose that on a particular evaluation of a hypothetical model M that was covered earlier, we would like to construct a confusion matrix that would represent its performance. We notice that there are 10 observations in total in this small dataset D. Then it was able to correctly classify 3 out of the 10 to have a label 1 and 4 out of the 10 to have a label 0. Since the accuracy of the model was computed to be 70%, it means that 7 out of the 10 should have been correctly classified, and this checks out. And then 1 out of the 10 were misclassified to be of having label 1 when actually it has label 0, and then 2 out of the 10 had misclassified observations with label 0 when it truly has a class label of 1. Since the misclassification error was computed as 30%, then 3 out of the 10 should have been incorrectly classified, and this we can easily verify for ourselves. We can see this particular confusion matrix above.

Code 5.2: Accuracy Measure for the Gao Model

```
# Initially set the count of all to be 0
tp <- 0
fp <- 0
tn <- 0
fn <- 0

# store the predicted labels and actuals of the testing set into a variable
predicted <- pred[!trainFlag]
actuals <- y[!trainFlag]

# determine how many tp, fp, tn, fn there are; iterate the dataset
for (i in 1:length(predicted)) {
    # apply the rounding concept of the logistic as discussed in Sec. 5.1.1
    p <- as.numeric(predicted[i] >= 0.5)
    a <- actuals[i]
    tp <- tp + as.numeric(p == 1 & a == 1)
    tn <- tn + as.numeric(p == 0 & a == 0)
    fp <- fp + as.numeric(p == 1 & a == 0)
    fn <- fn + as.numeric(p == 0 & a == 1)
}

acc <- (tp+tn)/(tp+tn+fp+fn)
```

Thus, it can easily be seen that accuracy can also be expressed in terms of these $\{TP, TN, FP, FN\}$:

$$\text{ACCURACY}(M) = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.12}$$

and consequently the misclassificaiton error rate as:

$$\text{ERROR}(M) = \frac{FP + FN}{TP + TN + FP + FN} \tag{5.13}$$

In the Gao Model, we do not exactly see where the calculation of the accuracy takes place; however as we see from traces of the code that it did somehow computed for TP and FP rates, then we can somehow write additional R code to compute for such metrics. This we can see in **Code 5.2** and we can query in `acc`, `tp`, `tn`, `fp`, and `fn` to know the performance of the Gao Model. Three runs of the benchmark model were also done initially prior to the project, similar to **CLOCK**. We see the results below:

```
> acc
[1] 0.7607053
> tp
[1] 61
> tn
[1] 241
> fp
[1] 31
> fn
[1] 64
```

Based on this, it can be interpreted that the benchmark model has a (testing) accuracy of ∼76.1%, and we can construct a confusion matrix, expressed as a count, instead of a percentage as it gives you more detail about the dataset than expressing it as a relative percentage.

		Predicted Value	
		1	**0**
Actual Value	**1**	TP = 61	FN = 64
	0	FP = 31	TN=241

Now let us analyze the performance of our **CLOCK** Model. We have a running code already that computes for the (testing) accuracy for us through the `cnn_model`'s `model_score`. We can see this code in **Code 5.1**, in Lines 2-4. I have run **CLOCK** for at least three times and here is the result of the third run, which happens to be the most superior among the other two runs (whose accuracy happens to be at 84% each):

```
================================================================================
Test Loss:  0.4514982
Test Accuracy:  0.8680556
================================================================================
```

This suggests that the testing accuracy of our CLOCK Model is at ∼86.8%, which we can say is pretty good considering the benchmark model has a general accuracy of $\mu = 76.1\%$. This is a great improvement of $\sim 10\%$! We can compute for the individual values of the TP, TN, FP, and FN of the performance of the **CLOCK** model to ensure that everything checks out. However, we may need not to do this, as this can easily be coded in R. For now, getting that sweet number there should be what is important.

We then ask ourselves how is the testing loss relevant to the testing accuracy and testing errors? The **loss function** is not exactly the same as the misclassification error. While the misclassification error computes for how much the model M has deviated its prediction of the class labels in D from their actual class labels, the loss function quantifies how much impact and consequence this misclassification error has brought onto the model's performance. Depending on whether our machine learning problem deals with a classification or regression task, then the type of loss function applied is different. For instance the popular **MSE** or *Mean Squared Error* is an example of a loss function for regression tasks:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \tag{5.14}$$

where the $\hat{y}_i$ represents predicted labels and the y_i represents the actual labels. And then we can use the **Negative Log-Likelihood Function** (commonly known as the **Cross-Entropy Loss**) as a popular loss function for classification tasks. In our **CLOCK** Model we can see this in Lines 51-52 of **Code 4.2**. The mathematical representation of the (Binary) Cross-Entropy Loss (for two classes) is given as:

$$\text{BINARY-CROSS-ENTROPY} = -y_i \log(p_i) + (y_i - 1)\log(1 - p_i) \tag{5.15}$$

So in our **CLOCK** Model, we can see that the testing loss turns out to be 0.4514982. The range of the (log) loss is somewhere between 0 and 1. A particular model would perform poorly as its loss approsches 1, but becomes better as its loss approaches 0. In the ideal setting, having a loss of 0 demonstrates a perfect model, which isn't existent in the real world. The **CLOCK** Model's loss is not too bad, not too good either; as its loss is sitting close to the middle (at 0.5). We also note that a higher accuracy should imply a lower loss value. In **Fig. 14**, we can see the training and validation accuracy and loss for the **CLOCK** Model. It is available in Keras and we cannot provide one for the benchmark model as it does not have a validation set, hence no validation accuracy and validation loss.

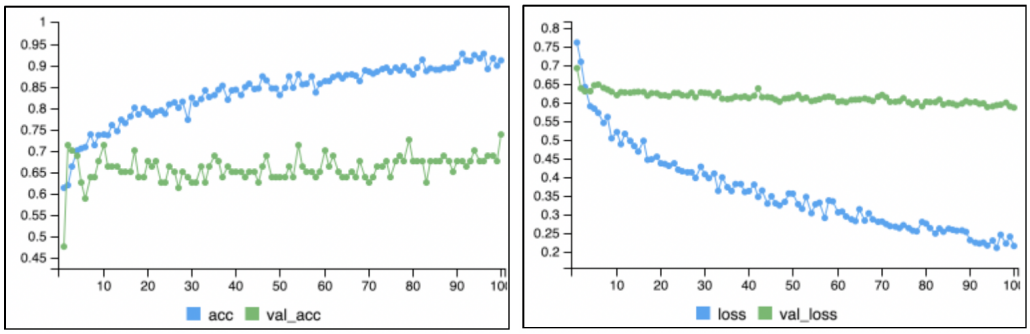


Fig. 14. The **CLOCK** Model's training and validation accuracies and losses, for 100 epochs (through Keras)

Another performance metric we can introduce relevant to the accuracy is known as the **specificity** and **sensitivity**. We can obtain the sensitivity using the formula given by:

$$\text{SENSITIVITY}(M) = \frac{TP}{TP + FN} \tag{5.16}$$

And then the specificity as:

$$\text{SPECIFICITY}(M) = \frac{TN}{TN + FP} \tag{5.17}$$

In other words, sensitivity talks about the correctly classified positive observations, while specificity indicates the correctly classified negative observations. As a remark, we should notice that a higher sensitivity implies a lower specificity, and vice-versa. This is because these metrics are inversely proportional.

From the Gao model, we can compute sensitivity as follows:

$$\text{SENSITIVITY}(\mathbf{M\_GAO}) = \frac{TP}{TP + FN} = \frac{61}{61 + 64} = 0.488 \tag{5.18}$$

and then its specificity as:

$$\text{SPECIFICITY}(\mathbf{M\_GAO}) = \frac{TN}{TN + FP} = \frac{241}{241 + 31} = 0.886029 \tag{5.19}$$

And then let us look at the **CLOCK** Model. We can query in for `tp`, `tn`, `fp` and `fn`, and that we would get $TP = 31$, $TN = 38$, $FP = 4$ and $FN = 7$ in one of its runs. According to these values, we can compute for the sensitivity of the model:

$$\text{SENSITIVITY}(\mathbf{CLOCK}) = \frac{TP}{TP + FN} = \frac{31}{31 + 7} = 0.815789 \tag{5.20}$$

And then its specificity as:

$$\text{SPECIFICITY}(\mathbf{CLOCK}) = \frac{TN}{TN + FP} = \frac{38}{38 + 4} = 0.904762 \tag{5.21}$$

We may also be interested in computing for other metrics of the model related, such as what is known as the **precision**, the **recall** and the **F1-measure** or **F1-score**.

Often confused with the term *accuracy*, the precision actually talks about the *exactness* - or basically the percentage of positively labelled observations that were in the actuallity positive. This is given by the formula:

$$\text{PRECISION}(M) = \frac{TP}{TP + FP} \tag{5.22}$$

And then there's recall, which talks about *completeness*. Here, we ask ourselves what percentage of the positively-labelled observations have been classified as positive? This is given by the formula:

$$\text{RECALL}(M) = \frac{TP}{TP + FN} \tag{5.23}$$

Notice that the recall is equivalent to the model's sensitivity, and we sometimes call this the **True Positive Rate** (or **TPR**) and the **True Negative Rate** (or **TNR**) is another term for specificity as they are equivalent.

The reason we may want to be interested in these metrics, aside from the *accuracy* metric and *misclassification error* metrics, is that data may be imbalanced. For instance, if your model has a true positive count of 998, a true negative count of 1, a false negative count of 1 and a false positive count of 0, then we have an imbalanced data. We cannot use the accuracy metric to say that the accuracy of the model is $\frac{998+1}{998+1+1+0} = 99.9\%$ and conclude that it is a good model for its high accuracy, because our data is imbalanced and skewed. There's too much positively labelled items. Usually, data is upsampled or downsampled during data preprocessing to fix the issue. However in our case, we may try using other performance metrics aside from accuracy. We can use those mentioned, or another version of the accuracy, which is a modified version which we call the **Balanced Accuracy**. We can express this using the True Positive Rate, TPR, and the True Negative Rate, TNR (or using sensitivity and specificity):

$$\textsc{balanced-accuracy}(M) = \frac{1}{2}\left[\textsc{sensitivity}(M) + \textsc{specificity}(M)\right] = \frac{TPR + TNR}{2} \tag{5.24}$$

We can notice that the classification set of dataset D turns out to be quite imbalanced, as seen in the distribution count in **Table 9**. This is for Problem III; however if we attempted Problem I or II, then the "imbalance-ness" of the data is reduced. In Problem I, 277 of the images are of type `outdoor-day`, and the rest of the 523 images are not. Meanwhile in Problem II, there are a total of 405 `outdoor`-type images and 395 non-`outdoor` images. This means that Problem II has a more "balanced" data distribution compared to the other two problems; and this is also one of the reasons why **CLOCK** performs really well!

Since **CLOCK** uses an almost perfectly balanced dataset, then it doesn't really make sense to calculated the balanced accuracy. We can verify the accuracy result with the balanced accuracy, and find out for ourselves that the balanced accuracy is at $\sim 86\%$ as well, almost close to what we got earlier. The Gao Model used a not-so balanced dataset as the count distribution of the labels in the set has ratio $\sim$1:2. We can calculate the balanced accuracy of the Gao Model, to get a more accurate value.

$$\textsc{balanced-accuracy}(M_{GAO}) = \frac{\textsc{sens}(M_{GAO}) + \textsc{spec}(M_{GAO})}{2} = \frac{0.49 + 0.89}{2} = 0.69 \tag{5.25}$$

which we can see is even less than what R had computed. As a result, overall the Gao Model can perform at an average of $\sim 70\%$ performance accuracy rate.

We can compute for the precision value of the benchmark model as:

$$\textsc{precision}(M_{GAO}) = \frac{TP}{TP + FP} = \frac{61}{61 + 31} = 0.663043 \tag{5.26}$$

and the recall as:

$$\textsc{recall}(M_{GAO}) = \frac{TP}{TP + FN} = \frac{61}{61 + 64} = 0.488 \tag{5.27}$$

We can then compute for the precision value of the **CLOCK** model as:

$$\textsc{precision}(\textbf{CLOCK}) = \frac{TP}{TP + FP} = \frac{31}{31 + 4} = 0.885714 \tag{5.28}$$

and the recall as:

$$\textsc{recall}(\textbf{CLOCK}) = \frac{TP}{TP + FN} = \frac{31}{31 + 7} = 0.815789 \tag{5.29}$$

While we talk about the precisions and recalls of our models, it should make sense to talk about the F_1**-Score** (or F_1**-Measure**), which is expressed as the harmonic mean of the precision and recall:

$$\textsc{f}_1\textsc{-score}(M) = \textsc{hm}(\textsc{precision}(M), \textsc{recall}(M)) = \frac{2\,\textsc{precision}(M)\,\textsc{recall}(M)}{\textsc{precision}(M) + \textsc{recall}(M)} \tag{5.30}$$

where $\text{HM}(a, b)$ is the harmonic mean of a and b, defined by $\frac{2ab}{a+b}$. Note that this particular metric can be used in seeking balance between precision and recall. So using the precision and recall values of the Gao Model above, we can compute its F_1-score:

$$\text{F}_1\text{-SCORE}(M_{GAO}) = \frac{2\,\text{PRECISION}(M_{GAO})\,\text{RECALL}(M_{GAO})}{\text{PRECISION}(M_{GAO}) + \text{RECALL}(M_{GAO})} = \frac{2(0.663043)(0.488)}{0.663043 + 0.488} = 0.562212 \quad (5.31)$$

And then we compute the F_1-score for the **CLOCK** Model:

$$\text{F}_1\text{-SCORE}(\textbf{CLOCK}) = \frac{2\,\text{PRECISION}(\textbf{CLOCK})\,\text{RECALL}(\textbf{CLOCK})}{\text{PRECISION}(\textbf{CLOCK}) + \text{RECALL}(\textbf{CLOCK})} = \frac{2(0.885714)(0.815789)}{0.885714 + 0.815789} = 0.849315 \quad (5.32)$$

5.2 Model Comparison

In the previous section, we have covered the different testing and evaluation methods, as well as some common performance metrics used in the world of machine learning. Now that after we have covered these metrics and analyzed them on our models, we now attempt to compare particularly the benchmark model and the **CLOCK** Models, according to:

- performance evaluation
- feasibility and complexity
- strengths and weaknesses

It would be great to summarize all the basics about each model in a table, as seen in **Table 8**, to get a sense of what encompasses each of the model.

5.2.1 Performance-based Comparison

In **Sec. 5.1.2**, we have covered the different performance metrics, such as the *accuracy*, *misclassification error*, *specificity*, *sensitivity*, *precision*, *recall* and F_1*-score*, as well as examined how each of the models performed while evaluating each of them according to such performance metrics. In this particular section, we would discuss how the **CLOCK** model compared and how it improved from the benchmark model. We shall analyze how its performance works out to be better in terms of the several performance metrics covered in the previous section. In **Table 9**, we can see a table comparing the benchmark model and the **CLOCK** Model, according to performance. We will use this as a guide to make some analyses and comparisons.

To begin, we look at accuracy and (misclassification) errors. We note how they are related, that their rates should add up to 1. This means that as we achieve a higher accuracy, we should also achieve a lower misclassification error, which is the ideal setup in any model. Also notice that there are three types of accuracy and three types of errors. One is for training, one for validation and another for testing. We have examined this in R in the previous section that depending on dataset used, we can obtain that

	GAO Model *(Naïve, Benchmark)*	**CLOCK Model** *(Proposed Improvement)*
Type of Problem Solved	Problem I (Classifying `outdoor-day` images)	Problem II (Classifying `outdoor`-type images)
Nature of Model	Generalized Linear Model (Logistic Model)	(Convolutional) Neural Network Architectures
Dataset Splitting Method	Simple Uniform Random Sample ($\sim$ 50% training set, $\sim$ 50% testing set)	Simple Random Sample ($\sim$ 80% training set, $\sim$ 10% validation set, $\sim$ 10% testing set)
R Libraries	JPEG	Plyr, Keras, EBImage, StringR, PbApply, TensorFlow

Table 8. A quick summary of the basic comparisons between the benchmark model and the **CLOCK** Model

Performance Metric		GAO Model	CLOCK Model
Accuracy	**Training**	75%	90.7%
	Validation	-	69.6%
	Testing	76.1%	86.8%
Error	**Training**	25%	9.3%
	Validation	-	30.4%
	Testing	23.9%	13.2%
Cross-Entropy Loss	**Training**	0.75	0.228233
	Validation	-	0.6001
	Testing	0.7201087	0.4514982
Confusion Matrix	**TP**	61 (15.4%)	31 (38.75%)
	TN	241 (60.7%)	38 (47.5%)
	FP	31 (7.8%)	4 (5%)
	FN	64 (16.1%)	7 (8.75%)
True Rates	**Sensitivity**	0.488	0.815789
	Specificity	0.886029	0.904762
Balanced Accuracy		68.7%	86.8%
Other Metrics	**Precision**	0.663043	0.885714
	Recall	0.488	0.815789
	F1-Score	0.562212	0.849315

Table 9. A summary of the comparisons between the **GAO** and the **CLOCK** Models, performance-wise; as a sidenote, as each of these were run multiple times for testing, the top three best runs according to accuracy were taken and averaged and the results can be seen above

particular accuracy or error (i.e. for example, if we want to determine the training accuracy of the model, we simply query in the accuracy of the model using the training set). Between the Gao and the **CLOCK** Models, only the **CLOCK** Model made use of a validation set. Hence, only this particular model has a validation accuracy and error, which happens to be 69.6% and 30.4% respectively. This doesn't look good for **CLOCK**, as it suggests that we can still improve the model even if it had outperformed the Gao Model, according to the accuracy threshold μ. Recall that accuracy threshold μ is simply the benchmark model's test accuracy, which is 76.1% and the **CLOCK** did beat it, with a test accuracy of 86.8%, by almost a good 10% improvement! In terms of training accuracy, the Gao Model achieves a 75% accuracy and the **CLOCK** Model scored 90.7%. So we can see here that in every aspect of the accuracy (and the misclassification error for that matter), **CLOCK** appears to be superior than the Gao Model.

Closely related to accuracy and error is the loss, or sometimes called a loss function. Since we are dealing with a classification problem, we would use the cross-entropy loss function as seen in **Eq. (5.15)**. This is a special case of the cross-entropy loss called the **binary cross-entropy loss**. We recall that this is not exactly the same as an error, but we should know that similar to the error, the lower it is then the better; and to achieve a lower loss entails a higher accuracy. Similar to both accuracy and error, depending on which dataset you applied the loss formula, then you should get the cross-entropy loss for that particular set (i.e. for instance, applying the formula on the training set should result into the cross-entropy training loss). For the cross-entropy training loss, the Gao Model's is 0.75 and the **CLOCK** Model's is 0.22833. For the cross-entropy testing loss, the Gao Model's is 0.7201087 while for the **CLOCK** Model's is 0.4514982. In both cases, **CLOCK** performed better for having a lower loss. Its cross-entropy validation loss is 0.6001, which again isn't too good as it implies that we can still redesign and improve the model to get better results. As the Gao Model did not have validation data during the data split, then there is no cross-entropy validation loss.

The next performance metrics to look at are the value counts or percentage of True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN), and these can easily be placed in what is called the **Confusion Matrix**. It is important to note these as they are essential in computing for other performance measures. To give a better perspective, this is only for testing, so we apply this onto the testing set only. The Gao Model has roughly half of the total dataset, 397 to be exact; while the **CLOCK** Model uses a test set that only has a tenth of the total number of images in the image set, 80 to be exact. If we were to express the counts as percentage to level the playing field, then the true positive percentages (not to be confused with the true positive rate or TPR) are 15.4% and 38.75% while

its true negative percentages (not to be confused with the true negative rate or TNR) are 60.7% and 47.5% respectively for the benchmark and **CLOCK** Models respectively. The reason why we say that the true positive (or negative) percentage should not be confused with the true positive (or negative) rate is that there is actually another separate performance metric called **True Positive Rate (TPR)**, or what is known as *sensitivity* (there is also the **True Negative Rate or TNR**; *specificity* is another term for such); and these have different formulas. Meanwhile the False Positive Percentages of the Gao and **CLOCK** Models are 7.8% and 5% respectively, while their False Negative Percentages are 16.1% and 8.75% respectively. Notice that the True Positive/Negative Percentages is the same as the testing accuracy and the False Positive/Negative Percentages is the same as the testing error. This Confusion Metric numbers are also used to give a better picture and understanding as to the accuracy percentage and error percentage. Using the counts in comparison does not seem to be effective due to the fact that the Gao Model used a larger testing set than that of the **CLOCK** Model. So using percentages to compare should make comparison levelled. Since True Positive/Negative is positively correlated with Accuracy, then the ideal model needs a higher percentage for these. **CLOCK** is able to beat the Gao Model at TP, but not in TN. We can also notice how TP compares with TN on a particular model. We see an almost equally-levelled in the **CLOCK** Model, but as we see in the benchmark model, only 15.4% for the TP and 60.7% for TN, which suggests a great imbalance of data. I would cover more on imbalance of data and how it is usually dealt with, later on. In terms of the FP/FN, the ideal model suggests that we should want a lower percentage of these as these are highly correlated to the misclassification error. We can see from our table in **Table 9** that **CLOCK** is superior to Gao Model as it has a lower FP/FN percentages.

Going back to the ideas of sensitivity and specificity as mentioned earlier, these are actually what is known as the True Rates. The True Positive Rate (TNR), or the percentage of the TPs out of the actually positive data, is what is known as the **sensitivity**. The True Positive Rate (TPR), or the percentage of the TNs out of the actually negative data,is what is known as the **specificity**. As we are talking about the percentages of correctly predicted (whether positive or negative) data, then it makes sense that we want higher values for such metrics. In this case, **CLOCK** has again outperformed the Gao Model, with its sensitivity to be 0.815789 (0.488 only for the benchmark model) and its specificity to be 0.904762 (0.886029 for the benchmark model).

Another performance metric covered in **Sec. 5.1.2** is what is known as the **balanced accuracy**, which is basically used for *imbalanced datasets*. Although there are several methods in dealing with imbalanced data during data preprocessing, in terms of performance evaluation metrics, then we can use this modified version of accuracy. The balanced accuracy is simply the mean of the True Rates (only for testing). So we grab the True Positive Rate and True Negative Rate of a particular model, then take their sum and then divide this value by 2. As the Gao Model used an imbalanced dataset as discussed in **Sec. 5.1.2**, it seems logical to use the balanced accuracy measure, as the model's balanced accuracy turns out to be 68.7%. Our **CLOCK** Model used an already balanced dataset so the balanced accuracy is simply the same as the original accuracy we have. Two remarks: (1) Even though both models began with the same dataset to begin with, the problem setup stirred an effect as to the classification distribution (i.e. Problem I uses a dataset with 277 `outdoor-day` images and 523 non-`outdoor-day` images implying an imbalanced dataset D while Problem II uses a dataset with 405 `outdoor`-type images and 395 non-`outdoor`-type images implying a more balanced dataset D), so the original imbalance of data was taken care of when we attempted Problem II in **CLOCK**. The second remark being: (2) The original accuracy that you might expect from a model that uses an imbalanced dataset can still increase or decrease; in the case of the Gao model, it has decreased.

Other metrics such as the precision and recall were also discussed, that uses $\{TP, TN, FP, FN\}$. Furthermore, as F_1-score is a related metric that can be expressed in terms of the harmonic mean of precision and recall, then we also talked about it in **Sec. 5.1.2**. What is ideal here? We want a good F_1-score, which is affected by precision and recall. And to get a perfect F_1-score (of 1) in the ideal world, then you need perfect precision and recall scores (of 1 for both). Hence, we need higher precision and recalls to get a higher F_1-score. We can see from **Table 9** the the **CLOCK** Model has a higher precision with 0.885714 than the benchmark model with 0.663043, a higher recall with 0.815789 than the benchmark model with 0.488, and a higher F_1-score with 0.849315 than the benchmark model with 0.562212.

Algorithm 5.2: CALCULATE-RUNTIME (M, D)
The pseudocode for determining runtimes of chunks of code for some model M and dataset D

Input: A model M and a dataset D
Output: The calculated runtimes of certain chunks of code

```
1       // get current time; code is innate in R
2  t1 ← Sys.time()
3       // ... insert data preprocessing code using dataset D here ...
4  t2 ← Sys.time()
5       // ... insert code for model construction of model M here ...
6  t3 ← Sys.time()
7       // ... insert code for performance evaluation of model M here ...
8  t4 ← Sys.time()
9       // display time results
10 preprocessing_time ← t2 - t1
11 model_buildtime ← t3 - t2
12 evaluation_time ← t4 - t3
13 return (preprocessing_time,model_buildtime,evaluation_time)
```

Overall, we can see how much improvement we have made from the Gao Model to the **CLOCK** Model, in terms of performance. Now, let us have a look at how feasible the models are, as well as the runtime complexities of the codes. Then we can make an overall comparison as to how both the Gao and **CLOCK** Models compared in terms of feasibility and complexity.

5.2.2 Feasibility and Complexity-based Comparison

Feasibility in this context talks about how easy and how do-able is programming the model. In the Gao Model, only one line is needed for the model to be built (note that the problem definition and data preprocessing is not included here; we only talk about model construction). This is in Line 21 in **Code 4.1**, which is found on **p. 23**. Simply use the `glm()` function and in one line, we can build the Gao Model. Clearly, the Gao Model is indeed feasible.

On the other hand, we needed 75 lines of code for the **CLOCK** Model, as seen in **Code 4.2** on **pp. 24-25**. The feasibility level of this model is somewhat subjective. In my case, given that I have worked with neural networks[15] and convolutional neural networks[16] already before (in Python), then working with NNs and CNNs in R is somehow straightforward already (from the ideas, concepts, theories, libraries used, etc.). All I have to do is to be able to translate this knowledge from Python to R (i.e. this is my first time to work on NNs and CNNs in R, but I have already done so in the past through Python, as seen in the past projects I worked on), which I think is easy now due to the `reticulate` R library that allows the Python interface (needed as the NN API Keras is written in Python). The only difference is perhaps some syntactical differences, and the guides such as [13], [14], [15], and [16] all have helped me write my own CNN Model in R. However, if the notions of Neural Networks and Convolutional Neural Networks are still new to the user, then implementing such complex architectures may be challenging, and I was hoping that my paper was able to help - from the introduction of the concepts to the discussion of the actual **CLOCK** model, which should serve as an exemplar as well on how NNs and CNNs are implemented. Hence in terms of feasibility, it seems like the Gao Model has triumphed at this comparison due to the easiness aspect of the benchmark model's implementation.

Now it's time to look at time complexities. Let us first look at the actual runtimes of these codes. Remember that to know how long a particular chunk of code is running, then we wrap it with the use of the `Sys.time()` function in R, similar to what was discussed in **Sec. 2.3.5**. Refer to **Algorithm 5.2** for the skeleton of the pseudocode. In **Table 10**, we can see here a summary of the average runtimes of data preprocessing, model construction and performance evaluation for each of the models.

[15] See one of my Neural Network projects in Python over on my GitHub page: [11], where I worked on a machine learning classifier for a modified version of the MNIST dataset.
[16] See one of CNN projects in Python over on my GitHub page: [12], where I worked on face and gender recognition on a celebrity image set.

Runtime (in secs.)	**GAO Model**	**CLOCK Model**
Data Preprocessing	49.95 secs.	20.86 secs.
Model Construction	0.04 secs.	6.16 secs.
Performance Evaluation	0.13 secs.	2.77 secs.
Total	50.12 secs.	29.79 secs.

Table 10. The average runtimes for Data Preprocessing, Model Construction, and Performance Evaluation of the **GAO** and **CLOCK** Models. Note: these runtimes were run in my computer, which only uses a regular CPU and does not have the specialized GPU or Graphics Processing Unit that enables to perform tasks with images at a faster rate

On average, it takes the Gao Model about ~ 50 seconds to preprocess data. This includes importing libraries, loading the data, and doing some data manipulations in order for the dataset to be in the "correct form" in order for a model to be constructed and used correctly. Meanwhile, the **CLOCK** model only takes less than half of the benchmark model's preprocessing time, around ~ 20 seconds! Clearly, we can see how our improved model beats the Gao Model in this aspect. However, let us examine what makes the data preprocessing take some time. We can notice in the **Table 10** that it is data preprocessing that takes the longest time, compared to the other runtimes. Even in the real world, data preprocessing takes the longest time in any machine learning or data mining task. Why is this so? Rather than talking about it in general, let us focus our attention as to why this is the case in this context. Take a look at the Gao Model, comparing it to the **CLOCK** Model. Which R library was used to import the images? The Gao Model used the `JPEG` library while the **CLOCK** Model used the `EBImage` library. Although both models initially began with the same 800 images in the image set, we can observe how much faster images were loaded in in the **CLOCK** Model. This may suggest that `EBImage` is a more advanced R library that can handle images compared to the `JPEG` library. The `EBImage` can work with images that's not of JPEG file extension and also treats images as `Objects`, which are more powerful than primitives in the world of Computer Science (`JPEG` treats images as floating point arrays, and floating numbers are considered primitive types). Also, furthermore, even though both models make use of a loop to iterate through each image upon loading, each image is stored as an array in the Gao Model; it then still gets the median of each of the arrays on each color channel - which should take longer time because behind the scenes, getting the median involves sorting the numbers in the color channel array, then selecting the middlemost number. On the other hand, the **CLOCK** Model only loads the image and no mathematical operation is applied to each of the image's color channels. Furthermore, we scale the image down to 32×32 pixels; to make training of the model faster since we are dealing with 800 many images. This should explain why data preprocessing is cut down in the **CLOCK** preprocessing.

Next would be model construction. This should include building the architecture of the model, connecting the layers together and training of the model (including validation). The Gao Model's one line code of building a GLM logistic model only took ~ 0.04 seconds which is outstandingly fast! But this should not be a surprise to us. Meanwhile, the **CLOCK** Model as it uses NNs and CNNs really does take a lot of time. My **CLOCK** Model, when run on a regular CPU (like what I used) should take about ~ 6.16 seconds, which is quite long actually! Given that we used more hidden units or more hidden layers can make the network even more complex, thus taking longer for model construction and training. Of course, less hidden units and less hidden layers can reduce the runtime of the model training and construction, as well as the model's complexity. If the **CLOCK** Model was to be run on a computer with a specialized Graphics Processing Unit (GPU), then this should accelerate any preprocessing and training of image data, and cut down runtimes!

Finally, let us have a look at performance evaluation. Anything to do with testing and evaluating the performance using the metrics that we have discussed so far are part of this. It took the Gao model 0.13 seconds while the **CLOCK** Model took 2.77 seconds. So similar to model construction and training, the **CLOCK** Model did not succeed in outperforming the Gao Model in performance evaluation runtimes. This may be partly due to the reason of my computer not containing the GPU chip, which causes such image tasks to be slower.

However in the overall, the Gao Model took 50.12 seconds to perform all tasks, with almost all work done during preprocessing. Meanwhile, the **CLOCK** Model took a little more than half of the total

runtime of the Gao Model, which is about 29.79 seconds. So the **CLOCK** Model is still superior overall in the runtime aspect.

However, even though we can see from the **Table 10** that the **CLOCK** Model outperforms the Gao Model in the total runtime aspect, we can still look at **runtime complexity**.

In the context of Computer Science, the term *complexity* here is not the same as *the time of how long it took* and it is not measured in seconds or any unit of time either. This complexity here also does not refer to "complex models" (or hard, difficult to implement), or perhaps the "complex numbers" (or the imaginary numbers). I am referring to some simple theoretical concepts of Computer Science that talks about giving an upper bound (or lower bound) to express how fast a particular algorithm runs based on some given input. See, it's not measured in some unit of time, but rather is expressed as the magnitude of the algorithm's input. This is also sometimes known as the **asymptotic analysis** or the **asymptotic (runtime) complexity**.

To illustrate with some quick, simple examples, take Algorithms A, B, and C. Let Algorithm A be an algorithm that allows us to find the product of all n numbers in a 1D list, then Algorithm B be the algorithm that allows us to sum up all elements of an $n \times n$ square matrix, and then Algorithm C that allows us to find a particular integer in a 1D list of sorted integers. We need not write a pseudo-code for these algorithms but by easily explaining in plain English, it should be clear.

For Algorithm A, if there are n numbers in some one dimensional list, then we can linearly sweep through by first taking one of the numbers then get the next one and multiplying them together, then multiply this number with the next number in the list, and so on until all numbers in the list have been accounted for. We can then say that the upper bound for this particular runtime complexity is $\mathcal{O}(n)$, which just means that given this n input then in no way would be the runtime of this algorithm run longer than that (i.e. proprotional to n). We call this the **Big-O notation** and this is how we express upper bound running time. On the other hand, another notation which expresses the lower bound is called the **Big-Omega notation**, in which Algorithm A's lower bound is $\Omega(n)$. This just says that this algorithm cannot run faster than that (i.e. proportional to n input). If an algorithm has the same upper and lower bounds, we give it what we call a **tighter bound**, or **tight bound**. This is usually expressed in **Big-Theta notation**. Thus, Algorithm A is said to have a tight bound of $\Theta(n)$. One might say that we can improve the algorithm and include a checker that checks whether a particular number that we have grabbed from the list is 0, as this automatically makes the final product 0; however, in runtime analysis we are interested in what happens at the *worst-case*. Hence, if it so happens that the list did not contain a 0 at all, then we would still have to traverse this list until all numbers have been taken into account.

Let us look quickly in Algorithm B. In order to sum the elements of a square matrix with size $n \times n$, we may do a similar strategy as Algorithm A. We add each element of the matrix into a $n \times n = n^2$-sized 1D list. And then add numbers as how we took the product of the numbers in Algorithm A. However, there is no need to do the extra step of transferring the elements of the matrix into a list. We can simply use a "nested" `for` loop to iterate through the matrix to sum all numbers up (this should involve summing first each row of the matrix then totalling up all the sums of each row). In either way, we can show that this method is $B \in \mathcal{O}(n^2) \wedge B \in \Omega(n^2) \Rightarrow B \in \Theta(n^2)$. We shall note that an n^2 runtime performs worse than an algorithm that has a runtime of n, since we consider its growth in the long run (i.e. n^2 grows much faster than n, for very large inputs of n). Note however that if I say that this matrix is of size 5×5 instead of $n \times n$, then that makes a big difference. Now that the size of the input is constant, then we know that the running times have to be constant as well. As such, we call the tight bound to be $\Theta(1)$.

Now, for Algorithm C. If a particular integer is in a given sorted 1D list of integers, then this algorithm should say "True" and "False" otherwise. We can do a simple linear scan from the beginning to the end of the list. If we find the integer that we are looking for, then return "True"; otherwise, if we have reached the end of the list and still have not yet found it, then we simply return "False". However, this is a naïve way of searching through a sorted list of integers (and any form of linear sweeping is not as good as its performance has a $\mathcal{O}(n)$ runtime). We can instead take advantage of the fact that this list is sorted, and then we can apply a popular algorithm called **Binary Search**, which runs in $\Theta(\log(n))$. This is an iterative method where I do the following: (1) grab the middlemost element in my sorted list. (2) If my search integer is less than this middlemost number, I throwaway all numbers greater than or

equal to the middlemost number in the list and keeping only all the elements less than the middlemost number (hence cutting the list into half); otherwise, then I keep all numbers greater than or equal to the middlemost number instead and throwaway all numbers less than this middlemost number instead. And (3) iterate through the steps until I am left with a single element to compare with, and decide accordingly from there if the search number is indeed in the list. One should be able to test easily that given n numbers in this list, then the number of steps at the worst case to take is $\Theta(\log(n))$ (can easily be proven).

Now that we have made a quick, easy, simple and gentle introduction to runtime complexity, let us do some runtime analysis, beginning with the benchmark model. Usually, the operations of accessing, storing into variables, and checking for conditions only have a cost of 1 (i.e. runtime of $\Theta(1)$). Another thing to note is that if an operation that runs in $\Theta(n)$ is run a constant number of times consecutively (say three times), then its total runtime is $\Theta(n) + \Theta(n) + \Theta(n) = 3 \times \Theta(n) = \Theta(3n)$, which in this case is still $\Theta(n)$. We may also deal with dataframe operations in R (for instance subtracting 1 to each value of all rows in a particular column); then we just say that its runtime is $\Theta(1)$ without loss of generality. Now refer to **Appendix B** for the Gao Model. I have broken it down into the following runtimes:

- We can see from Lines 1-13 that there are no special constructs that can be seen, other than accessing and storing simple operations. Hence the runtime for this is $\Theta(1)$.
- Now Lines 14-18 is a loop. We know that there are 800 images, so technically we should have a runtime of $\Theta(1)$, because the number of iterations would have been constant. However, for the sake of having n images instead (suppose that we want to have a robust model that can handle a variable n images input; could be more than 800 or less than 800). Usually, a single simple `for` loop has a runtime of $\Theta(n)$ if it was to run n times. However, let us consider as well what is inside the body of the loop.
  - Line 15 is simply grabbing an image and storing it, which is just constant theta time.
  - In Line 16, this is where it gets tricky. Getting the median involves sorting the numbers in a list and grabbing the middlemost number. Finding the middlemost number is straightforward constant time but sorting is a different story. Usually a quicksort algorithm is applied when we ask the program to sort some particular list, and this has $\Theta(n \log(n))$ runtime (not discussing the technical details of sorting). We are doing this for 3 channels, and since this is a constant number, it should not affect our current runtime.
  - Then printing something onto the screen in Line 17 is also constant time.
  - Hence the overall runtime in one iteration of the loop is $\Theta(n \log(n))$; and with n iterations, then the time in total for Lines 14-18 should be $\Theta(n^2 \log(n))$.
- Now, nothing much interesting is going on in Lines 19-29 as the runtime is $\Theta(1)$.
- In Line 30 (and Line 31), getting the mean involves summing and then dividing by the number of n there are (i.e. averaging); hence its runtime should be $\Theta(n)$.
- Lines 34-39 isn't interesting either ($\Theta(1)$ runtime).
- Now we'll see that Lines 40-44 is a `for` loop which iterates n times.
  - We see in Line 41 that it has a constant runtime.
  - Line 42 has a runtime of $\Theta(n)$ (sum). This is also the case for Line 43.
  - Thus overall, Lines 40-44 has a runtime of $\Theta(n * n) = \Theta(n^2)$
- Finally, the remaining lines up until Line 57 only has $\Theta(1)$ runtime.
- Hence overall, the runtime complexity of the Gao Model is: $\Theta(1)*4+\Theta(n^2 \log(n))+\Theta(n)*2+\Theta(n^2) = \Theta(n^2 \log(n))$.

In fact, we can do a similar runtime analysis for the **CLOCK** Model. Refer to **Appendix C**, which has the complete R Script of the proposed model:

- In Lines 1-29, there is nothing really going on aside from library imports, querying, and assignment operations. These are all $\Theta(1)$ constant time.
- Similar to how we handled loops in the benchmark model, we can do the "robust" way so that we do not fix our number of images to just 800. Lines 30-34 is the loop structure:
  - As Lines 31-32 are simply comments, they do not really add anything to the cost or to the count.
  - Line 33 is simply an `if-else` statement control-structure and this operation is applied on a particular column of the dataframe. As a result, this line should also be constant time $\Theta(1)$.
  - Thus, Lines 30-34 runs in $\Theta(1) * n = \Theta(n)$.
- There is nothing interesting really happening in Lines 35-50 that would change the runtime, as these lines operate in constant $\Theta(1)$ time.

- Lines 51-56 is similar to Lines 30-34 due to the loop structure. It runs n times and each of the operations within the loop body from Lines 52-55 are $\Theta(1)$ operations. Hence, Lines 51-56 run in $\Theta(1) * n = \Theta(n)$ altogether.
- Now lines 57-118 are mostly assignments. So they all run $\Theta(1)$ time.
- Now the next lines until Line 176 are all lines related to the NN and CNN's model training and architecture. This is really hard to analyze, unless we make some assumptions. Assuming that hyperparameters can easily be tweaked by the user and that it should depend on the size of the input of n images into our model, then we can either say that a $\Theta(n)$ time would be justifiable. A $\Theta(1)$ time may also be possible, but if we think about it - why do neural networks in general take a long time to process and train? If we had a smaller input size, we can get the results at a much faster rate; however if our dataset was massive, it can take a longer time due to the fact that we may opt for a bigger and more complex architecture for better accuracy. Note that $\Theta(1)$ does not suggest that the model is fast; it should suggest that no matter how complex or simple our architecture is then the operation takes the same amount of time, in proportion to the input. But this is not the case for NNs as NNs need more complex models and architectures (i.e. increase hidden units and layers for instance) to increase accuracy for instance of very big datasets. $\Theta(n)$ on the other hand suggests that increasing or decreasing the complexity of the architecture has the same impact to the operations of the model proportionally to the size of the input n (same case for any tightbounds for less than n or bigger than n). Hence, we can standardize and say $\Theta(n)$ runtime instead.
- The last few lines up until Line 198 is merely and testing and checking of accuracy, which we can conclude to have run in $\Theta(1)$ time.
- The overall runtime complexity of the **CLOCK** Model should be $\Theta(n)$ as the lines above suggest either a $\Theta(n)$ or $\Theta(1)$ operation; in this case, we pick the max.

Now that we have performed runtime complexity analysis on both models, we then compare. Basically the purpose of this analysis is to examine among the models which one performed more efficiently than the other (i.e. we are comparing the models according to efficiency). The Gao Model has a runtime of $\Theta(n^2 \log(n))$ while the **CLOCK** Model only has a runtime of $\Theta(n)$, which suggests that the proposed model is more efficient than the benchmark model - which validates the results obtained from the **Table 10**. In fact, this is an accurate representation of a naïve model vs an improved model; the naïve model seems to perform poorly in terms of efficiency compared to the improved model (analogous to the naïve LinearSearch and improved BinarySearch algorithms mentioned earlier). In this case, the poorly-performing model is less efficient and has a worse runtime compared to the more improved model.

5.2.3 Strengths and Limitations of the CLOCK Model

We come to talk about the different strengths and weaknesses of our proposed **CLOCK** Model. Along the way, we shall also be relating to the benchmark model in how they compare. Some aspects may not be directly applied to the benchmark model, but we'll still cover them here. Comparing to the Gao model as it turns out that the **CLOCK** Model is better performance-wise (accuracy metric) and is more efficient than the Gao Model (can be seen in **Table 10** as well as the runtime analysis that we have performed). This can be one advantage of the use of our proposed model compared to the benchmark model.

We can perhaps consider robustness as well. Notice that Lines 101-118 of the **CLOCK** Model in **Appendix C** are the hyperparameters of the network architecture. This may be changed anytime by anyone who has the code and it should not cause any errors at all. In fact, changing one hyperparameter changes the entire network architecture without having to change anything else in all other places multiple times. As an example of what I mean, consider these two hypothetical codes:

```
================================================================================
# Code A:
print("The conversion from US to CAD is $1.42")
print("$1 USD = $1.42 CAD")

# Code B:
usd_cad = 1.42
print("The conversion from US to CAD $" + usd_cad)
print("$1 USD = $" + usd_cad + " CAD")
================================================================================
```

Notice that both these codes will display the same result. In Code A, we see that the `$1.42` is directly affixed in the string output, while this was in the form of a variable in Code B. However, what happens if the USD to CAD conversion changes the next day? This means that for Code A, you would have to change all the lines of code here that contains `$1.42`, with the new conversion rate. This could be tedious as there could be several lines of code that has this, which is more prone to errors (either you may mistype or you may miss one or a couple of these). On the other hand, Code B allows you to change the first line with the new conversion and it should change the values of all other lines with this new conversion as it is used as a variable instead of a string constant. Code B is less tedious, error-free and more robust. In fact, Code B exhibits good programming practice. This is how we have done it in the **CLOCK** Model, as exemplified by Lines 101-117 which can be referenced in **Appendix C**. It can also make hyperparameter-tweaking much easier.

We can also talk about how the **CLOCK** model is robust in a sense that it can handle a different number of the Columbia images (could be more or less), and it should not have any errors (as long as the metadata will also be updated). We may simply update and tweak hyperparameters if necessary to preserve accuracy and performance. So in theory, we can actually use another type of dataset (say images in Toronto instead of in Columbia) and that it can in theory be able to train and learn which images are from the `outdoor` and which ones are not.

One thing to note is that because Problem II is phased in a way where the class labels turn out to be balanced (or at least close to perfectly balanced), then we did not have to face the issues of imbalanced data. This is faced by Problem I in the Gao model, as there are only 277 `outdoor-day`-type images and roughly twice than that are those belonging to the other category. As a result, the dataset was imbalanced. It is impressive that the simple change of problem definition can actually change the way that the datasets are perceived and remove the problem of this imbalance. This is the root cause of why Problem III would be a hard problem to tackle, even for **CLOCK**. Notice in the data distribution as seen in **Table 9** that there is an imbalance of distribution among the datasets (for instance the majority class is `outdoor-day` with 277 counts, and the lowest ones are the `outdoor-night` and `outdoor-dawn-dusk` with 34 and 31 counts). This would be problematic for any model trying to use this dataset. As such attempting on Problem III is a failure for **CLOCK** and this is its limitation. However, this data imbalance usually has some workarounds (upsampling and downsampling techniques) which can be done during data preprocessing.

Another weakness of **CLOCK** is its infeasibility as mentioned, as well as the difficulty to design the architecture as there is no rule of thumb for setting the correct hyperparameters. Multiple tweaking and testing is needed. As a result, it is possible that the models we get turn out to be instable. This just means that our **CLOCK** model may either be running at a very good performance during one run but then would end up not doing so well in the next run. We can say it's by chance as well; but this is more due to our model's instability as one of its core weaknesses.

To summarize all that has been mentioned in this paper, refer to **Table 11** for a final summary comparison between the benchmark naïve model and the improved, proposed model. It can be concluded that our proposed **CLOCK** Model indeed is a better classifier of the Columbia Image set, simply based on the results as shown in the table.

Criterion	GAO	CLOCK	Notes
Performance	✗	✓	**CLOCK** has improved accuracy by $\sim 10\%$
Capability	✗	✓	**CLOCK** can solve a harder problem (Problem II) than the benchmark (Problem I)
Runtime	✗	✓	**CLOCK** took roughly $\sim 60\%$ much faster than the Gao Model
Efficiency	✗	✓	**CLOCK**'s $\Theta(n)$ time beats the Gao Model's $\Theta(n^2 \log(n))$
Robustness	✓	✓	Allows to easily change model values and datasets without making errors non-tediously
Feasibility	✓	✗	The benchmark model is easier and more feasible to code (57 vs 198 lines of code)
Stability	✓	✗	Unstable **CLOCK** performance due to model's complexity

Table 11. Final comparison summary between the benchmark and the proposed models.

6 Conclusion

6.1 Summary

We have seen that the proposed **CLOCK** Model was able to improve the benchmark model in the following ways:

1. In terms of the performance metrics, in accuracy alone we have observed that the **CLOCK** Model has an improved accuracy of $\sim 10\%$, which is an impressive result!
2. Being able to solve a harder problem, **CLOCK** can solve a problem that classifies images as `outdoor`-typed or not; compared to the Gao Model which classifies only whether images are `outdoor-day` or not.
3. Runtime and efficiency-wise, **CLOCK** took $\sim 60\%$ much faster to run than the Gao Model, which when expressed in terms of runtime complexity as a function of input n, then the **CLOCK** runs $\Theta(n)$ which is much better than the $\Theta(n^2 \log(n))$ time of the benchmark model.

Although the **CLOCK** Model seemed to exhibit numerous benefits and capabilities, nowhere is it near the extremely-accurate performance of the state-of-the-art Neural Network and Convolutional Neural Network Architectures of the [8, p.407] performed experiments of LeCun, as we saw in **Table 1**. Despite the advantages that we have seen, this proposed model has its own limitations, such as:

1. Due to the complexity of the network architecture of the model, this leads to a model - one that is unstable performance-wise.
2. Its feasibility of easily coding and programming is not-so-much recognized unlike the benchmark model.

Having been able to work on the **CLOCK** Model and have used the Keras packages of NNs and CNNs, turns out to be a good experience and serves as a great stepping stone for other bigger image classification and computer vision tasks and applications.

6.2 Recommendations and Extensions

We can further extend on our study, either focusing on improving the **CLOCK** Model or perhaps even the Gao Model.

As seen in **Table 9** for example, the data distribution is very skewed and imbalanced. Usually a simple solution would be to apply additional data preprocessing steps on the imbalanced datasets. These are **upsampling** and **downsampling** techniques, which basically increases the minority-labelled data (i.e. the data which belongs to the minority class) and to decrease the majority-labelled class (i.e. the data which belongs to the majority class). The application of one of the upsampling methods on the minority-labelled data such as **Synthetic Minority Oversampling Technique (SMOTE)** as detailed in [17] is one of the ways that we could have done to improve the imbalanced dataset and produce more accurate models. If we have done so, then it would be possible that **CLOCK** may be able to solve Problem III after all.

During our process, the ultimate goal was to outperform the Gao model by outperforming its accuracy. If after we have found a model that has beaten this accuracy threshold $\mu \sim 76\%$, then we can halt our process and conclude that we have found the model **CLOCK**. Could it be possible though that there's even a better model called **CLOCK*** that is able to outperform **CLOCK**? Think about it; what if **CLOCK** is an *overfitted* model and that **CLOCK*** is a better model that does not overfit the dataset? By **Eq. (5.9)**, this suggests that even if the training accuracy of **CLOCK** is better than the training accuracy of **CLOCK*** but the testing accuracy of **CLOCK*** was actually better than the testing accuracy of **CLOCK**, then **CLOCK** is overfitted:

$$\text{TRAIN\_ACC}(\textbf{CLOCK}) > \text{TRAIN\_ACC}(\textbf{CLOCK*}) \wedge \text{TEST\_ACC}(\textbf{CLOCK}) < \text{TEST\_ACC}(\textbf{CLOCK*}) \tag{6.1}$$

What would have happened if we continued playing this game of hyperparameter-tweaking and came up with better models having higher accuracies and better performances? These new models generated should have replaced our original **CLOCK** model, and we can report models with better performance metrics. Additionally, we may come up with some new model wherein its performance stabilizes, that is to say stability is no longer an issue for **CLOCK**, and it can perform well regardless for any of its runs.

In terms of extensions, we can think of a similar problem where our model is tasked to classify images (our image set are images of different places around the world) according to the name of the city that they are in (i.e. identify where the image is taken from). So for instance, you could have images taken from perhaps Toronto, New York, Mumbai, Manila, etc. and the classifier will identify which city the image was taken from. This is similar to the past project I have made applying CNNs on face and gender recognition of celebrities as seen in [12]. Could the same (or at least a similar) model or algorithm from my past project or from **CLOCK** work to perform this task accurately?

Another extension would be object detection (humans for example but any other objects may be possible). If we can detect the humans in the images in our image set, can we "remove" them from the photo so that we are left with images without humans? And then use this as an intermediary preprocessing step in order to classify our images better? That is to say these particular objects are treated as "noise" (i.e. noise detection) and then the end goal of this is to filter out this "noise" as a preprocessing step to produce a better-performing model?

References

[1] J. Brownlee. "9 Applications of Deep Learning for Computer Vision." Internet: https://machinelearningmastery.com/applications-of-deep-learning-for-computer-vision/, Jul. 5, 2019 [Mar. 17, 2020].

[2] T. Ng, S. Chang, J. Hsu, and M. Pepelijugoksi. *Personal Columbia Set*, New York: DVMM Laboratory, Department of Electrical Engineering, Columbia University, 2005. [Online]. Available: http://www.ee.columbia.edu/ln/dvmm/downloads/PIM_PRCG_dataset/ [Feb. 28, 2020].

[3] T. Ng, S. Chang, J. Hsu, and M. Pepelijugoksi, "Columbia Photographic Images and Photorealistic Computer Graphics Dataset," ADVENT Technical Report #205-2004-5, Columbia University, Feb. 2005. Accessed on: Feb. 28, 2020. [Online]. Available: http://www.ee.columbia.edu/ln/dvmm/publications/05/ng_cgdataset_05.pdf.

[4] X. Gao (2020) MATH3333ProjectSampleCode (Version 1.0) [Source code]. https://github.com/techGIAN/CLOCK_Image_Classifier/blob/master/gao_sample.txt.

[5] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Cambridge, MA: MIT Press, 2017.

[6] Y. LeCun, C. Cortes, and C. Burges. *The MNIST Database of Handwritten Digits*, New York: Courant Institute, 1998. [Online]. Available: http://yann.lecun.com/exdb/mnist/ [Mar. 18, 2020].

[7] K. Murphy. *Machine Learning: A Probabilistic Perspective*. Cambridge, MA: MIT Press, 2012.

[8] T. Hastie, R. Tibshirani, J. Friedman *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. New York, NY: Springer, 2017.

[9] A. Thyssen. "ImageMagick - Color Basics and Channels." Internet: http://www.imagemagick.org/Usage/color_basics/#channels, Mar. 09, 2011 [Feb. 28, 2020].

[10] C. Bishop. *Pattern Recognition and Machine Learning*. New York, NY: Springer, 2006.

[11] G. Alix, NN_notMNIST, (2019), GitHub Repository, https://github.com/techGIAN/NN_notMNIST/blob/master/NN_notMNIST.ipynb.

[12] G. Alix, CNN_FaceScrub, (2019), GitHub Repository, https://github.com/techGIAN/CNN_FaceScrub/blob/master/CNN_FaceScrub.ipynb.

[13] F. Lima. "Convolutional Neural Networks in R." Internet: https://www.r-bloggers.com/convolutional-neural-networks-in-r/, Jul. 08, 2018 [Mar. 17, 2020].

[14] A.S. Walia. "How to implement Deep Learning in R using Keras and Tensorflow." Internet: https://towardsdatascience.com/how-to-implement-deep-learning-in-r-using-keras-and-tensorflow-82d135ae4889, Jun. 19, 2017 [Mar. 17, 2020].

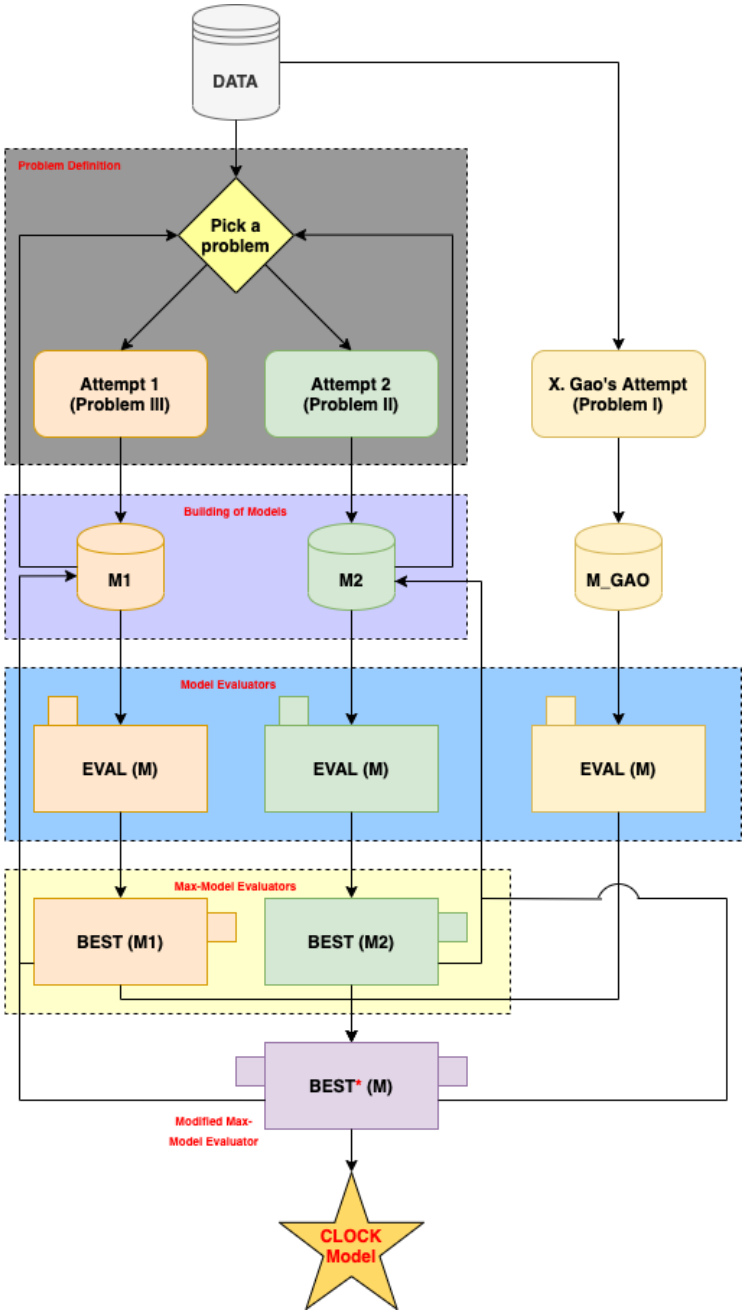
[15] F. Chollet & J.J. Allaire. "Image Classification on Small Datasets with Keras." Internet: https://blogs.rstudio.com/tensorflow/posts/2017-12-14-image-classification-on-small-datasets/, Dec. 13, 2017 [Mar. 17, 2020].

[16] J. Klopper. "Example of a convolutional neural network." Internet: https://rpubs.com/juanhklopper/example_of_a_CNN, n.d. [Mar. 17, 2020].

[17] N. Chawla, K. Bowyer, L. Hall, W.P. Kegelmeyer. (2002, June). "SMOTE: Synthetic Minority Over-sampling Technique." *Journal of Artificial Intelligence Research*. [Online]. Vol. 16, pp. 321-57. Available: https://arxiv.org/pdf/1106.1813.pdf [Apr 01, 2020].

Appendix

Appendix A: Methodology Visual



Appendix B: The R Script of the Benchmark Model

R File of the Gao Model: https://github.com/techGIAN/CLOCK_Image_Classifier/blob/master/gao_sample.txt

```
### Professor X. Gao's Sample R Code
### Original Source can be found at https://xingao.info.yorku.ca/files/2020/01/
###                                       math3333projectsamplecode.txt
### Photograph examples

## Read in the library and metadata
library(jpeg)
pm <- read.csv("C:\\Users\\xingao\\Documents\\teaching\\winter2020\\math3333\\photoMetaData.csv")
n <- nrow(pm)

trainFlag <- (runif(n) > 0.5)
y <- as.numeric(pm$category == "outdoor-day")
X <- matrix(NA, ncol=3, nrow=n)
for (j in 1:n) {
  img <- readJPEG(paste0("C:\\Users\\xingao\\Documents\\teaching\\winter2020\\math3333\\columbia
                         Images\\columbiaImages\\",pm$name[j]))
  X[j,] <- apply(img,3,median)
  print(sprintf("%03d / %03d", j, n))
}

# build a glm model on these median values
out <- glm(y ~ X, family=binomial, subset=trainFlag)
out$iter
summary(out)

# How well did we do?
pred <- 1 / (1 + exp(-1 * cbind(1,X) %*% coef(out)))
y[order(pred)]
y[!trainFlag][order(pred[!trainFlag])]

mean((as.numeric(pred > 0.5) == y)[trainFlag])
mean((as.numeric(pred > 0.5) == y)[!trainFlag])

## ROC curve (see lecture 12)
roc <- function(y, pred) {
  alpha <- quantile(pred, seq(0,1,by=0.01))
  N <- length(alpha)

  sens <- rep(NA,N)
  spec <- rep(NA,N)
  for (i in 1:N) {
    predClass <- as.numeric(pred >= alpha[i])
    sens[i] <- sum(predClass == 1 & y == 1) / sum(y == 1)
    spec[i] <- sum(predClass == 0 & y == 0) / sum(y == 0)
  }
  return(list(fpr=1- spec, tpr=sens))
}

r <- roc(y[!trainFlag], pred[!trainFlag])
plot(r$fpr, r$tpr, xlab="false positive rate", ylab="true positive rate", type="l")
abline(0,1,lty="dashed")

# auc
auc <- function(r) {
  sum((r$fpr) * diff(c(0,r$tpr)))
}
glmAuc <- auc(r)
glmAuc
```

Appendix C: The Complete R Script of the CLOCK Model

```
# CLOCK Model
# CLassifier for Outdoor images using Cnns through Keras
# Author: Gian Carlo Alix
# Visit my GitHub Repo of CLOCK: https://github.com/techGIAN/CLOCK_Image_Classifier

# necessary imports for image processing
library(plyr)
library(keras)
library(EBImage)
library(stringr)
library(pbapply)
library(tensorflow)

# Want to measure how long preprocessing is
t1 <- Sys.time()

# load the photoMetaData
meta_data <- read.csv("../photoMetaData.csv")

# query on the first few rows of the meta dataset
head(meta_data)

# query the number of images we have
n_images <- nrow(meta_data)
n_images

# re-assign categorical labels to numerical labels in the meta_data
unique_labels <- c("non-outdoor", "outdoor")
unique_labels_count <- length(unique_labels)
for (i in 1:n_images) {
  # this is for Problem II
  # if the category of the image has "outdoor" in its name, then assign label of 1 and 0 o.w.
  meta_data$label[i] <- if(grepl("outdoor",meta_data$category[i],fixed=TRUE)) 1 else 0
}

count(meta_data$label)

# view the first image in our image dataset
sample_IMG <- readImage("../columbiaImages/CRW_4786_JFR.jpg")
sample_IMG
display(sample_IMG)
meta_data[1,]

# new dimensions of image when scaled
new_width <- 32
new_height <- 32
channels <- 3   # represents the RGB Channel

# read all images and store in a 4d tensor
img_set <- array(NA, dim=c(n_images,new_height,new_width,channels))
for (i in 1:n_images) {
  filename <- paste0("../columbiaImages/",meta_data$name[i])
  image <- readImage(filename)
  img_resized <- resize(image, w = new_width, h = new_height)
  img_set[i,,,] = imageData(img_resized)
}

# split the image and meta dataset into training and testing
# set.seed(100)
train_split <- 0.8
validation_split <- 0.5
```

```
train_sample_size <- floor(train_split*n_images)
arr <- 1:n_images
train = sample(arr,train_sample_size)
train_set_x = img_set[train,,,]

not_training <- arr[!arr \%in\% train]
validation_sample_size <- floor(validation_split*length(not_training))
validation = sample(not_training,validation_sample_size)
validation_set_x = img_set[validation,,,]

not_tr_te <- not_training[!not_training \%in\% validation]
test_set_x = img_set[not_tr_te,,,]

meta_train = meta_data[train,]
meta_validation = meta_data[validation,]
meta_test = meta_data[not_tr_te,]

train_set_y_cat = as.matrix(meta_train[ncol(meta_data)])
validation_set_y_cat = as.matrix(meta_validation[ncol(meta_data)])
test_set_y_cat = as.matrix(meta_test[ncol(meta_data)])

# use keras' built-in function for one-hot encoding
train_set_y <- to_categorical(train_set_y_cat,num_classes=unique_labels_count)
validation_set_y <- to_categorical(validation_set_y_cat,num_classes=unique_labels_count)
test_set_y <- to_categorical(test_set_y_cat,num_classes=unique_labels_count)

# use keras' built-in one-hot encoding (mostly for Problem III but can also be for Problem II)
train_set_y <- to_categorical(train_set_y_cat,num_classes=unique_labels_count)
validation_set_y <- to_categorical(validation_set_y_cat,num_classes=unique_labels_count)
test_set_y <- to_categorical(test_set_y_cat,num_classes=unique_labels_count)

t2 <- Sys.time()

# we build the architecture of our convolutional neural network
# begin by defining a sequential linear stack of layers
cnn_model <- keras_model_sequential()

# set the hyperparameters
# this can be tweaked to improve the accuracy of the model
filter_size <- 100
kernel_width <- 3
kernel_height <- 3
kernel_dim <- c(kernel_width, kernel_height)
input_shape <- c(new_width, new_height, 3)
activation_func_1 <- "relu"
pool_width <- 2
pool_height <- 2
pool_dim <- c(pool_width, pool_height)
s_strides <- 3
dropout_val1 <- 0.8
dropout_val2 <- 0.5
hidden_units_1 <- 1000
output_units <- unique_labels_count
activation_func_2 <- "softmax"
learning_rate <- 0.0001
learning_decay <- 10^-6

cnn_model %>%
  # 1st convolutional layer
  layer_conv_2d(filter=filter_size, kernel_size=kernel_dim, padding="same",
                input_shape=input_shape) %>%
  layer_activation(activation_func_1) %>%

```

```
125.   # Use a max pool layer to dimensionally reduce the feature map to reduce
126.   # the model's complexity
127.   layer_max_pooling_2d(pool_size=pool_dim) %>%
128.
129.   # Using a dropout to avoid overfitting
130.   layer_dropout(dropout_val1) %>%
131.
132.   # need to flatten input
133.   layer_flatten() %>%
134.
135.   # create a densely-connected neural network with 1000 hidden units with the given
136.   # the given activation function relu (hidden layer) and a 0.5 dropout, then for
137.   # the output layer use softmax (to calculate cross-entropy)
138.   layer_dense(hidden_units_1) %>%
139.   layer_activation(activation_func_1) %>%
140.   layer_dropout(dropout_val2) %>%
141.   layer_dense(output_units) %>%
142.   layer_activation(activation_func_2)
143.
144. optimizer <- optimizer_adamax(lr=learning_rate, decay=learning_decay)
145.
146. cnn_model %>%
147.   compile(loss="binary_crossentropy", optimizer=optimizer, metrics="accuracy")
148.
149. # a summary of the architecture
150. summary(cnn_model)
151.
152. # We then train our model
153. data_augment <- TRUE
154. batch_size <- 100
155. n_epochs <- 100
156. if (!data_augment) {
157.   cnn_model %>%
158.     fit(train_set_x, train_set_y, batch_size=batch_size, epochs=n_epochs,
159.         validation_data=list(validation_set_x, validation_set_y), shuffle=TRUE)
160. } else {
161.   # Generate Images
162.   gen <- image_data_generator(featurewise_center=TRUE, featurewise_std_normalization=TRUE)
163.
164.   # Fit the image data generator to the training data
165.   gen %>% fit_image_data_generator(train_set_x)
166.
167.   # Generate batches of augmented/normalized data from image data and labels to
168.   # visualize the generated images made by the CNN Model
169.   cnn_model %>%
170.     fit_generator(flow_images_from_data(train_set_x, train_set_y, gen, batch=batch_size),
171.                   steps_per_epoch=as.integer(train_sample_size/batch_size), epochs=n_epochs,
172.                   validation_data=list(validation_set_x,validation_set_y))
173. }
174.
175. t3 <- Sys.time()
176.
177. # Testing and Check Accuracy
178. model_score <- cnn_model %>% evaluate(test_set_x, test_set_y, verbose=0)
179. cat("Test Loss: ", model_score$loss, "\n")
180. cat("Test Accuracy: ", model_score$acc, "\n")
181.
182. # View one test img (simple random sample)
183. r <- sample(1:length(not_tr_te),1)
184.
185. img_name <- as.character(meta_test[r,]$name)
186. filename <- paste("../columbiaImages/", img_name, sep="")
187. display(readImage(filename))
```

```
188.
189. category_predictions <- cnn_model %>% predict_classes(test_set_x)
190. p <- unique_labels[category_predictions[r]+1]
191. a <- unique_labels[as.numeric(test_set_y_cat[r])+1]
192.
193. # determines whether the model was able to predict the class of the image correctly
194. cat(img_name, "\n")
195. cat("Predicted:", p, "\n")
196. cat("Actual:", a, "\n")
197. match <- p == a
198. cat("The model predicted it correctly:", match, "\n")
```