Introduction to Convolution Neural Network

Junsheng Shi

1 Introduction: How CNN Works?

Convolution Neural Network (CNN), or ConvNet, is a biologically-inspired variant of MLP. In biological fields, the terminology "visual cortex" refers to sub-regions of the receptive visual field that contains a complex arrangement of neurons. These neurons behave as local filters over the receptive field and are capable of detecting the strong spatially correlation in images. Unlike the regular neural networks, which contain a tons of parameters due to its fully connected layers, CNNs simulate the functionality of visual cortex by convolution and shared weights. Please refer to 1.1 for further explanations of this concept.

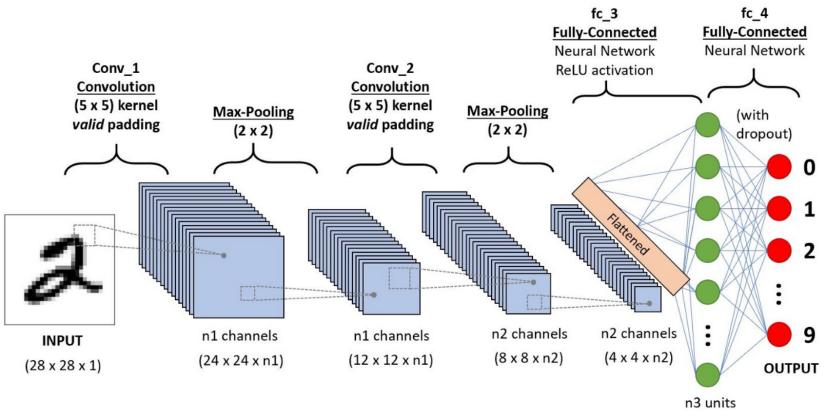

Figure 1: Convolution Neural Network

Figure 1 showcases a ConvNet architecture that is used to perform handwriting classification. We will discuss more regarding how this model may fit into our project in section 4.

The following sections provide a comprehensive review of some concepts and terminologies related to ConvNet. I will provide examples and visualizations in these sections to help understand how each part of the ConvNets works and what configurations we apply on a particular layer in our own model.

1.1 Convolution

One of the major part of CNN is, of course, the idea of convolution. The mathematical definition of convolution is stated as the integral of the product of the two functions after one is reversed and shifted. In mathematical notation,

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

The primary purpose of convolution in case of a CNN is to extract features from the input image using a filter or kernel (formal definition in section 1.4.1). As the filter is sliding, or convolving, around the input image, it is dot multiplying, or element-wise multiplying, the values in the filter with the original pixel values of the image. Each filter is replicated across the entire visual field. These replicated units share the same parameterization and form a feature map. Hence, convolution preserves the spatial relationship between pixels by learning image features using small squares of input data.

1.2 Neural Network

Neural Networks as neurons in graphs. Neural Networks are modeled as collections of neurons that are connected in an acyclic graph. In other words, the outputs of some neurons can become inputs to other neurons. Cycles are not allowed since that would imply an infinite loop in the forward pass of a network. For regular neural networks, the most common layer type is the fully-connected layer in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections. Figure 2 showcases a regular neural network and a 3D ConvNet.

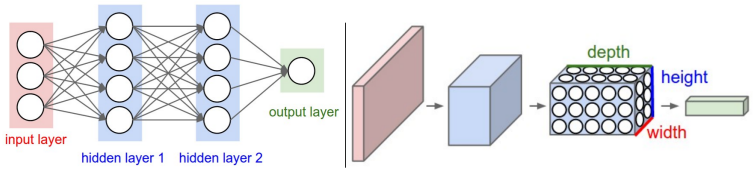

Figure 2: Neural Network

1.3 Backpropagation

Backpropagation is the essence of neural network training. It is the method of fine-tuning the weights of a neural network based on the loss function obtained in the previous epoch. Proper tuning of the weights allows to reduce loss and make the model reliable by increasing its generalization. Figure 3 presents an example of the one-neuron case, in which all layers only contains a single neuron.

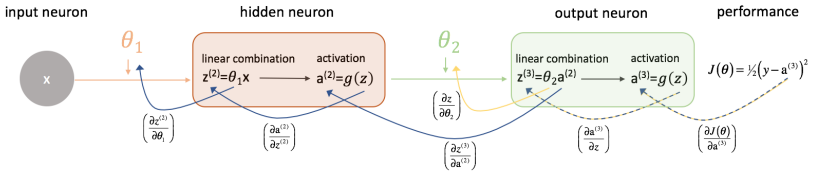

Figure 3: One-neuron Backpropagation

1.4 Convolution Layers

1.4.1 Kernel

In Convolution neural network, the kernel is nothing but a filter that is used to extract the features from the images. The kernel is a matrix that moves over the input data, performs the dot product with the sub-region of input data, and gets the output as the matrix of dot products. The kernel moves on the input data by the stride value. If the stride value is 2, then the kernel moves by 2 columns of pixels in the input matrix. In short, the kernel is used to extract high-level features like edges from the image.

1.4.2 Stride

Stride defines the step size of every move of the kernel. Figure 4 displays a convolution operation with stride of 1. That is, the kernel will move one step per time and perform convolution operation between the kernel and the corresponding submatrix, or basically dot product in this case. While applying convolution on a 5×5 matrix and a 3×3 matrix, the convolved features is a 3×3 matrix, because every time the kernel moves by 1 step and it can move 3 times vertically and horizontally within the input matrix.

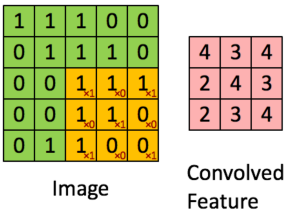

Figure 4: Convolution

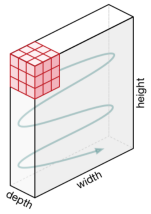

Figure 5: Movement of Kernel

Figure 5 on page 3 shows the movement of a 3×3×3 kernel in a three-dimensional matrix.

In general, the dimension of convolved feature matrix can be determined by three components: the size of the input matrix, the size of the kernel and the stride value. The mathematical expression is defined as the following:

$$\textbf{Feature size} = \textbf{((Input size - Kernel size)/Stride)} + \textbf{1}$$

1.4.3 Padding

Padding is a technique to simply add zeros around the margin of the image to increase it's dimension. Padding allows the kernel to emphasize the border pixels during convolution operation.

There are two types of padding technique: Valid Padding and Same Padding. An example of Valid Padding is Figure 4. As discussed in the previous section 1.4.2, valid padding starts from the upper left corner of the input matrix and moves to the bottom right in the pace defined by the stride value. In case that the remaining rows or columns are not enough for another move with the same stride, the algorithm drops off the remaining rows or columns and outputs the feature map. Therefore, the

size of the feature map should have been less than the input matrix, which results in dimensionality reduction.

On the other hand, Same Padding (Zero Padding) retains the dimensionality of the input matrix by adding zeroes around the margin of the original image data. Figure 6 presents a example of Same Padding and the feature map shares the same dimensionality as the image thereby.

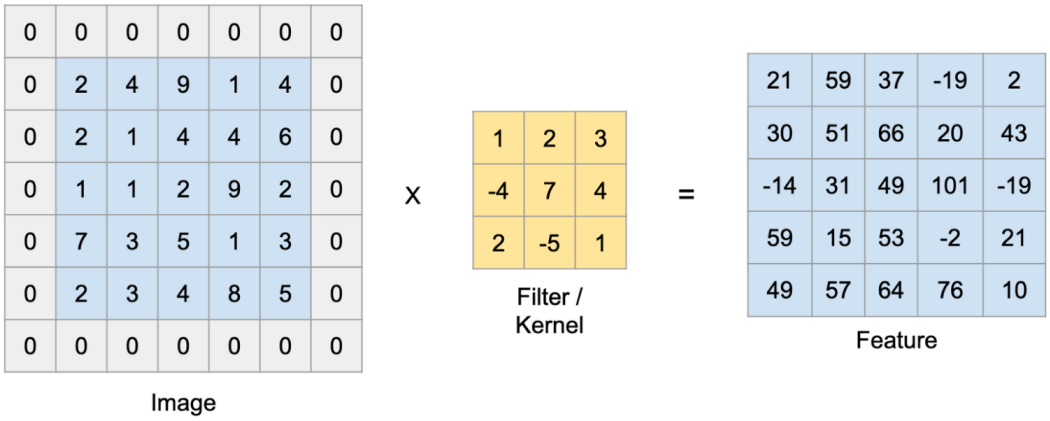

Figure 6: Convolution operation with kernel size 3, stride 1 and padding 1

Indeed, there is another type of padding, called Full Padding, which is essentially a mixture of both. Full Padding, like the Valid Padding, does not add zero borders before convolution, whereas once the kernel slides to the end of the image and the remaining rows or columns are not enough for another stride, Full Padding technique allows the kernel to slide over and complete the next stride by making up enough zero-valued rows or columns for convolution operation. Because it is a hybrid method, Full Padding is not included in the two major types.

Hence, the size of the feature map can be calculated by the following equation.

$$\textbf{Feature size} = (\textbf{(Input size} + 2\times\textbf{Padding size - Kernel size)/Stride)} + 1$$

1.4.4 Activation Function

Activation functions introduce non-linearity to the model which allows it to learn complex functional mappings between the inputs and response variables. There are quite a few different activation functions like sigmoid, tanh, ReLU, etc.

ReLU function is a piecewise linear function that outputs the input directly if is positive; otherwise, it will output zero.

$$\textbf{ReLU(x)} = \max(0, x)$$

In this project, Rectified Linear Unit (ReLU) is used in every convolution layer, mainly because (1) it is very simple to calculate, and (2) it has a derivative of either 0 or 1.

As a consequence, the usage of ReLU helps to prevent the exponential growth in the computation required to operate the neural network. ReLUs also prevent the emergence of the "vanishing gradient"

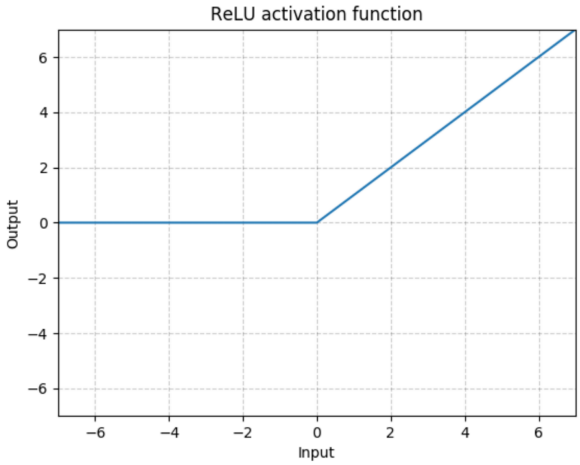

Figure 7: ReLU Activation Function

problem, which is common when using sigmoid functions. This problem refers to the tendency for the gradient of a neuron to approach zero for high values of the input.

The softmax function is used as the activation function in the output layer of neural network models for multi-class classification problems where class membership is required on more than two class labels. The mathematical definition of softmax is

$$softmax(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

1.5 Pooling Layers

A Pooling Layer is responsible for reducing the spatial size of the image features after convolution. Similar to dimensionality reduction, pooling directly leads to a decrease in computational power required to process high-dimensional image data. Furthermore, it is useful for extracting dominant features which are rotational and positional invariant, thus maintaining the process of effectively training of the model. There are two types of poolings to be considered: Max Pooling (1.5.1) and Average Pooling (1.5.2).

1.5.1 Max Pooling

Max Pooling returns the maximum value from the portion of the image covered by the kernel. For instance, in Figure 8, a 2×2 max pooling with stride of 2 returns the maximum value of each 2×2 block in the input matrix.

1.5.2 Average Pooling

Average Pooling returns the average of all the values from the portion of the image covered by the kernel. In Figure 8, for example, a 2×2 average pooling with stride of 2 moves around within the input matrix and returns the average value of every 2×2 square.

Both pooling methods achieve dimensionality reduction. However, max pooling also performs as a noise suppressant. It performs denoising by discarding noisy activation. Hence, Max Pooling performs better than Average Pooling and is deployed in this project.

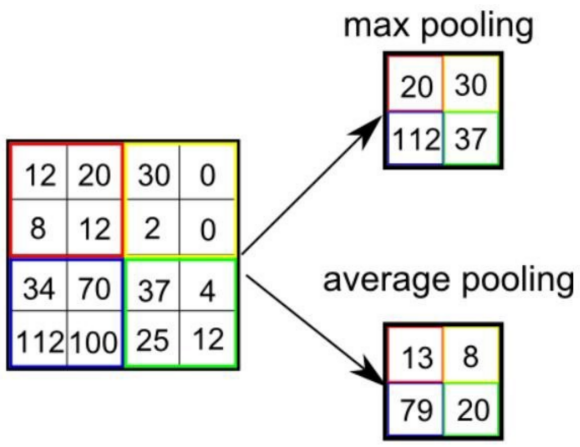

Figure 8: Pooling

1.6 Fully Connected Layers

The Fully Connected layer is configured exactly the way its name implies: it is fully connected with the output of the previous layer. Fully-connected layers are typically used in the last stages of the CNN to connect to the output layer and construct the desired number of outputs.

2 Parameters and Hyperparameters

In section 2.1, explanations of functions or classes with their parameters are provided in granular details.

In section 2.2, three major hyperparameters are discussed, in order to understand their impact on the model training.

2.1 Parameters

2.1.1 Sequential()

The main class in a CNN model is called ***Sequential***, which is a inherent class from ***Model*** in Keras library. It provides an overall architecture design for the neural network.

The input parameter of this instance is a list of ConvNet layer instances, including **Conv2D()**, **MaxPooling2D()**, **Dense()**, **Flatten()**, etc. The layers will be stacked upon each other in its given order. That is, the input parameter defines the model architecture. The layers can be classified into three categories: input layer, hidden layer and output layer. Further discussions about each layer and its parameters will be provided in the rest of this section.

2.1.2 Conv2D()

This layer creates a convolution kernel that is convolved with the layer input to produce a tensor of outputs. When using this layer as the first layer in a model, I provide the keyword argument input_shape=(32,32,3) that is the shape of the image data. If activation is not None, it is applied to the outputs as well.

Parameters:

filters: The number of output filters in the convolution.

kernel_size: An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Here I uniformly apply a 3×3 kernel across all convolution layers because its smaller size retains more features in the output map.

padding: "valid" means no padding. "same" results in padding with zeros evenly to the left/right or up/down of the input. When padding="same" and strides=1, the output has the same size as the input. Hence, I use "same" padding and default strides=(1,1) and obtain a same size feature map after every convolution layers.

strides: An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. I leave the stride as default (1,1), in order to obtain a same size feature map.

activation: I use ReLU activation functions for every convolution layers because its linearity behavior is faster in training and often performs better. Please refer to section 1.4.4 for more details.

2.1.3 MaxPooling2D()

Downsample the input along its spatial dimensions (height and width) by taking the maximum value over an input window (of size defined by pool_size) for each channel of the input.

Parameters:

pool_size: Integer or tuple of 2 integers, window size over which to take the maximum. I consider that a (2,2) window is sufficient to capture the features after each convolution operation. The reason why I choose the smallest possible window size is that I am conservative about losing information due to overly aggressive dimension reduction. I'd rather experiment on various window sizes before I reach to the best option. However, it turns out that a (2,2) window works sufficiently well and thus I did not follow up with any additional tuning attempts because of the limited computational power.

strides: Similar to **strides** in **Conv2D()**, here I leave it as default, which is the same as the **pool_size**, since this avoids the window slides into a visited region and outputs the same values in neighboring areas.

padding: Defaulted as "valid" so as to perform dimensionality reduction, as mentioned in section 1.4.3.

2.1.4 BatchNormalization()

Batch normalization applies a transformation that maintains the mean output close to 0 and the output standard deviation close to 1. No parameters are particularly considered and all are set to their default, as (1) I have complete data standardization during the data preprocessing stage (refer to section 3.1.3) and (2) the normalization will not alter the model evaluation result significantly.

2.1.5 Dense()

Dense Layer is used to classify image based on output from convolutional layers. Unlike **LeNet** using three Dense layers with sigmoidal activation, I aim to combine a **ReLU** dense layer with and **sigmoid** dense layer, for (1) it simplifies the architecture and thus (2) it saves unnecessary computational power with achieving a similar model performance. The **units** parameter for each dense layer is determined by the expected output space. The final output space should match the number of classes (i.e. 10 classes).

2.1.6 Flatten()

Flattening layers flatten the output of the convolutional layers to create a single long feature vector. And it is connected to the final classification model, which is called a fully-connected layer. I do not specify any parameters in this function.

2.1.7 ImageDataGenerator() & flow()

This image generator helps with data augmentation. It essentially creates new samples by rotation and shifting on the original copy.

Parameters:

> **rotation_range:** Degree range for random rotations. I set this parameter to 9, allowing random rotations with angle $\theta \in [0, 9]$. Any larger angles are not considered because it is just unrealistic to view a street sign from a perspective that deviates from the front view.
>
> **height_shift_range:** Set to be 0.1. The augmented image will be shifted up and down by 10% of its height.
>
> **width_shift_range:** Set to be 0.1. The augmented image will be shifted left and right by 10% of its width.
>
> **vertical_flip:** Set to be False because it does not make sense to view digits on the street signs upside down.
>
> **horizontal_flip:** Set to be False because flipping digits from left to right would help recognition.

The **flow()** method fits the data into the generator and produces an augmented dataset. It also takes a batch_size parameter along with the training data. In this case, I apply the same batch size uniformly to stay consistent.

2.1.8 EarlyStopping()

This instance stops training when a monitored metric has stopped improving. The goal of the training is to minimize the loss. With this, the metric to be monitored would be 'loss', and mode would be 'min'. I also specify the min_delta parameter to be 0.0005 and patience to be 3, which defines non-improvement as less than 0.0005 between 2 epochs and stops the training after detecting three consecutive non-improvement epochs.

2.1.9 Dropout()

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. I only concern about the **rate** parameter in this function, which defines the proportion of the input units to drop. For every hidden layers, I decide to only retain 70 percent of the input units, whereas for the flattened fully-connected layer, I only keep 60 percent of the units before the final classification. The rates are my own experimental results but other rates may be equivalently applicable if available.

2.1.10 Adam()

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

2.1.11 compile()

This method compiles the sequential model with an optimizer, loss function and metrics. Since the objective of this project is to obtain 91% testing accuracy, I use 'accuracy' as the evaluation metrics. In this project, Adam optimization algorithm is utilized to train the model. Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems. The 'categorical_crossentropy' loss function is thus used.

2.1.12 to_categorical()

This function converts a class vector (integers) to binary class matrix.

Parameters:

y: Array-like with class values to be converted into a matrix (integers from 0 to num_classes - 1). Here the input is the label matrix, which is a 1D vector containing integers from 1 to 10. Thus, we should delete the first column from the output.

num_classes: Total number of classes. If None, this would be inferred as max(y) + 1. I leave this parameter as default.

2.1.13 fit() & fit_generator()

Both functions work to fit the input data to the compiled model and return a history dictionary that contains loss values and metric scores.

Parameters:

generator: If data augmentation is required, this input will be the pre-configured **ImageDataGenerator()** instance that has been fitted on the original dataset. Otherwise, we use its default setting to fit the training dataset, which does not perform data augmentation. See 2.1.7 for explanations.

epochs: Set to be 5 for Stage I (6.1) and 10 for Stage III (6.3). More discussions can be found in section 6.

callbacks: Pass in the configured **EarlyStopping** instance. Please refer to section 2.1.8 for more details.

2.1.14 evaluate()

This function takes the testing data X and corresponding label matrix y and returns the testing loss score and testing accuracy. I also specify the batch size of 128.

2.1.15 build_model()

This method helps construct and compile a CNN model given different input parameters. It returns a pre-compiled model as specified in the *model*.

Parameters:

input_shape: A tuple of three integers. This tuple represents the shape of the input data.

num_classes: A integer variable. It indicates the number of classes in the output layer.

learning_rate: A float variable. Refer to section 2.2.3 for further discussion.

model: String from "starting", "second", "third", and "final". This parameter specifies which model to be built.

2.1.16 load()

The **load()** method takes two Boolean parameters, namely *train* and *test*, and returns a list of datasets. This helper function loads training and/or testing datasets if the corresponding parameters are set to be True.

2.2 Hyperparameters

In CNNs, the properties pertaining to the structure of layers and neurons, such spatial arrangement and receptive field values, are called hyperparameters. Hyperparameters uniquely specify layers. Because of the computational capacity, I will only consider three hyperparameters in this project: batch size

(2.2.1), epochs (2.2.2) and learning rate (2.2.3). The other hyperparameters, such as strides and padding, are discussed as regular parameters of model definition in the previous Parameters section 2.1 as well as in the final model construction section 6.1.

2.2.1 Batch size:

The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters. A training dataset can be divided into one or more batches.

Suppose the learning algorithm employs gradient descent optimization algorithm. When all training samples are used to create one batch, the learning algorithm is called batch gradient descent. When the batch is the size of one sample, the learning algorithm is called stochastic gradient descent. When the batch size is more than one sample and less than the size of the training dataset, the learning algorithm is called mini-batch gradient descent.

The size of a batch must be more than or equal to one and less than or equal to the number of samples in the training dataset. In the proposed model, the batch size is determined to be 128, which has been experimented against various alternatives and then chosen as the best performer.

2.2.2 Epochs:

The number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset. One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters. An epoch is comprised of one or more batches.

The number of epochs can be set to an integer value between one and infinity. Normally, a model should be trained with a larger epoch parameter to obtain a better predictive performance. In the proposed model, it has been shown that the epoch of 5 is already sufficient to obtain over 91% testing accuracy, as (1) the model architecture is sophisticated and well-designed enough to learn well, and (2) the dataset is of well-representative of the generic population.

Therefore, we will use 5 epochs in Stage I Architecture Design and Model Selection (section 6.1). However, epochs of 10 is considered in the final stage of model training, as we would like to see its performance and stability over a longer period of time.

2.2.3 Learning rate:

The amount that the weights are updated during training is referred to as the step size or the "learning rate." Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0.

The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck.

Because a more complex model requires longer training test as a result of its advanced architecture design, I decide not to pursue the entirety of its tuning but rather to seek proved candidates. A similar project has been found on Kaggle and the result shows, in Figure 9, that one outstanding performer is 10^{-3}. After a few rounds of experiments, the exact learning rate also suggests a good performance in my model. Hence, the learning rate is set to be 10^{-3}.

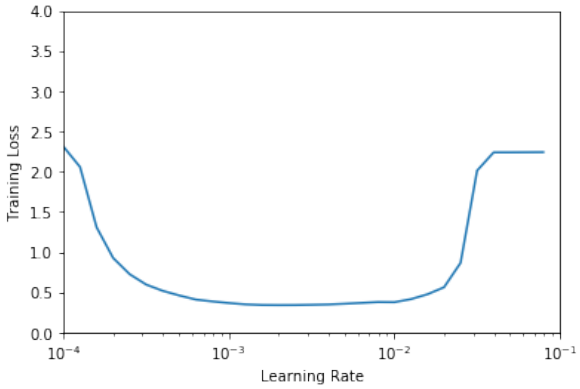

Figure 9: Hyperparameter Tuning - LR

3 Code Logic Walkthrough

The Python script consist of three major components: data preprocessing, model construction and model training and evaluation. To keep the paper well-organized and concise, the script will not be exhibited in this section; however, readers may find evident comment lines in the script as reference to each of the subsections mentioned and elaborated below. In the following subsections, the discussion will mainly focus on the empirical reasons and practice of such implementations.

3.1 Global Variables and Data Preprocessing

3.1.1 Environment Setup

At the very beginning of the project, a series of environment variables need to be determined, in order to deliver a reproducible project or experiment result. In this project, a Tensorflow backend is employed along with Keras library. Therefore, all Keras modules or functions are callable and indeed imported from the parent Tensorflow API. To assure the Tensorflow backend is setup and activated in the background, practitioners may use *keras.backend.backend()* command and expect to see "tensorflow" output in the IDE or terminal window.

Meanwhile, varying random seeds must have been predetermined, for the device always admits to a particular result during model training or other randomization processes. This involves a consistent setup at both the operating system level and the specific libraries level. Thus, setting *PYTHONHASHSEED* to 0 and *CUDA_VISIBLE_DEVICES* to a NULL character provide the control over these environment variables at the system level. On the other hand, providing numpy, random and Tensorflow library with predetermined seed values allows the output of their randomization be to consistent and thus

reproducible. The seed values can be different integers for these packages but in this project 0's are assigned invariably as a simple demonstration.

3.1.2 Global Variables

Global variables is composed of some predetermined variables that can be obtained from the pre-knowledge of the dataset, such as the path variable, the input shape and the number of classes. The path variable is determined by the directory structure of the project alone. This variable contains a placeholder so in the later stage the variable can be customized to a desired path string. The input shape and number of classes are acknowledged from the SVHN dataset. In the dataset description, it has been clearly indicated that the image data are of 32-by-32-pixel RGB format, which means each sample is in the dimension of $32 \times 32 \times 3$. It also claims that the labels are integers in the range from 1 to 10. Therefore, the input shape is set to be a tuple of (32,32,3) and the number of classes is 10.

3.1.3 Data Preprocessing

Data preprocessing and preparation includes data cleaning, data normalizing/standardizing, date transforming, or even data augmentation (refer to section 6.3). Since the SVHN dataset has been well-prepared, the data cleaning step is thus not performed. After loading the dataset, a image matrix X and a label matrix y are created to hold corresponding data for the training and testing datasets separately.

First of all, note the image data contain pixel values from 0 to 255. To save the computational power, the original data are all standardized into a range from 0 to 1 by dividing each pixel by 255. This step also converts the integer-valued pixels into float, which is the required data type in Tensorflow framework.

Second, note that the original dataset stores the image data in the order of (height, width, depth, number of samples/images), which is inconsistent with the expected ordering required by Tensorflow framework. Thus, the image matrix is transposed into the expected arrangement of (number of samples/images, height, width, depth).

Third, one-hot encoding technique helps transform the label matrix from a one-dimensional vector into a 10-column matrix, of which each column represents a class from 1 to 10 and each row represents a sample. Be aware of the fact that the classes do not start with 0, after calling the keras function, the output matrix actually consists of 11 columns, in which the first column represents a nonexistent class 0 and should have been deleted from the output.

So far, the image matrices and label matrices are well-prepared and ready for model fitting, as they meet all requirements of the Tensorflow framework.

3.2 Model Construction

In model construction stage, I define a *build_model()* function as explained in 2.1.15. Within this function, the model is built based on ***Sequential*** instance from tensorflow-backend Keras library. Multiple layers are stacked upon each other and then are added into the sequential model.

Note that all intermediate models are included in the script though, in the final submission, only the recommended final model will be trained and evaluated. A detailed discussion around the model configuration will be provided in section 4 and 6.

3.3 Model Training And Evaluation

The model training uses the entire training dataset over the pre-determined epochs time. For each epoch iteration, the model obtains a better accuracy and moves forward from there to the next epoch period. Over a few epochs, the accuracy of the model on the training dataset stabilizes and tends not to increase significantly in the next epoch period. The use of early stopping helps reduce the unnecessary training time as long as the increment in accuracy becomes trivial.

A grid-search stage may exist so as to tune the hyperparameters. In this project, the hyperparameters may include batch size, epochs and learning rate (refer to section 2.2). However, due to the extensive running time of experimenting with different values, in the final model I provide some recommended hyperparameters that has been considered empirically capable of attaining the desired testing accuracy (i.e. 91%).

When the model training is complete, the trained model will be brought to the evaluation stage. The performance evaluation will be carried out on the testing dataset, which remains intact in the model training stage. Thus, the evaluation outcome can be regarded as the model generalization on any unknown population and it may prove the predictive power of the model itself.

4 Starting Model and Methodology

The starting model built for experiment was inspired by the example of Simple MNIST convnet, which deployed a 5-layer CNN with MNIST dataset. This example provided a thorough explanation in terms of how to design a reasonable model architecture and how to tweak varying parameters to attain a specific model or achieve a learning objective. However, the example model was designed to work on a 32×32×1 dimension dataset, which represents a single color channel image dataset with 32 pixels in height and 32 pixels in length. Thus, the example model has been re-engineered to collaborate with a RGB image dataset by augmenting the single-channel model to a triple channel one. Figure 10 on page 15 illustrates the model architecture in details.

This first model provides a great experimental baseline for the subsequent, more complex models. In section 6, we will discuss how to reach to the final model from this model in details.

5 Model Evaluation

The final model evaluation will be performed once on the intact testing dataset. The goal is to achieve 91 % of testing accuracy. This step will be done after the final model has been determined.

For the intermediate stages, I employ cross-validation to evaluate the performance on the training dataset. For instance, in this project I use a 10% validation split. The model will set apart this fraction of the training data, will not train on it, and will evaluate the loss and any model metrics

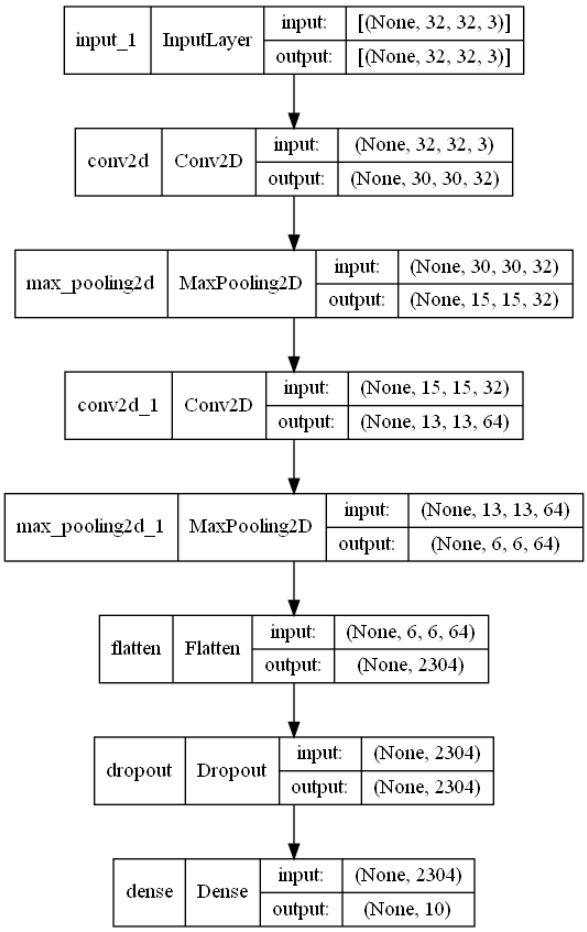

Figure 10: Starting CNN Architecture

on this data at the end of each epoch. I will use the accuracy score after each epoch and observe the growth rate and maximum value within 5 epochs to determine which model outperforms the others. Please refer to section 6.1 for evaluation discussions.

6 Final Model

In this section, I will discuss three major stages that help obtain the final model. Each stage will focus on only one aspect of the model optimization, including architecture design, hyperparameter tuning, and data augmentation.

6.1 Stage I

In Stage I, I mainly focus on the model architecture design. The objective of this experiment stage is to find a CNN architecture that can sustain a desired learning goal.

For the purpose of experiment, I only pay attention to the first five epoch periods and compare the model performance against each other. I will primary concern about two factors: (1) the number of epochs prior to stabilization and (2) the best mean cross-validation accuracy. Note that even though these two factors may positively correlate with the model performance in terms of training speed and accuracy, it is worth noting that a faster model or higher accuracy does not guarantee a model superior to another, especially for a short period of epochs.

To control the variations due to hyperparameters, I decide to use perform a backward testing, which acquire the tuned hyperparameters from Stage II and apply them uniformly on all candidate models. Thus, I set learning rate to 10^{-3}, batch size to 128 and epochs to 5, and use a 9:1 validation split in the training dataset.

I present the second (Figure 11) and third model (Figure 12) architecture on page 17. Please refer to Figure 10 in section 4 for the starting model and Figure 17 in section 6.3 for the final model.

Figure 13 and Figure 14 exhibit the trend of training improvement for each model.

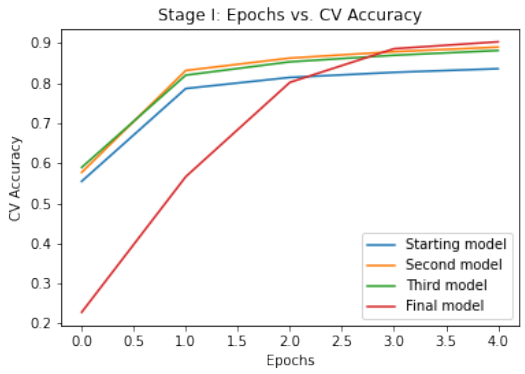

Figure 13: Stage I: Model Comparison - Accuracy

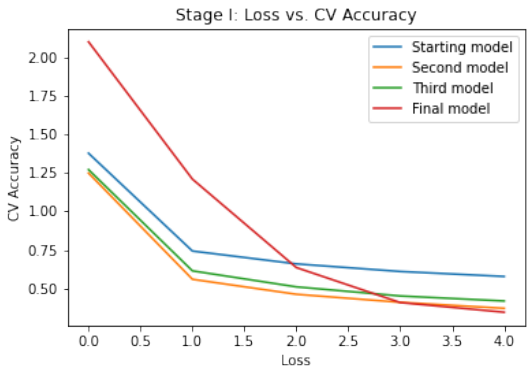

Figure 14: Stage I: Model Comparison - Loss

Observation: In Figure 13, the final model starts to outperform the other models after epoch 3.

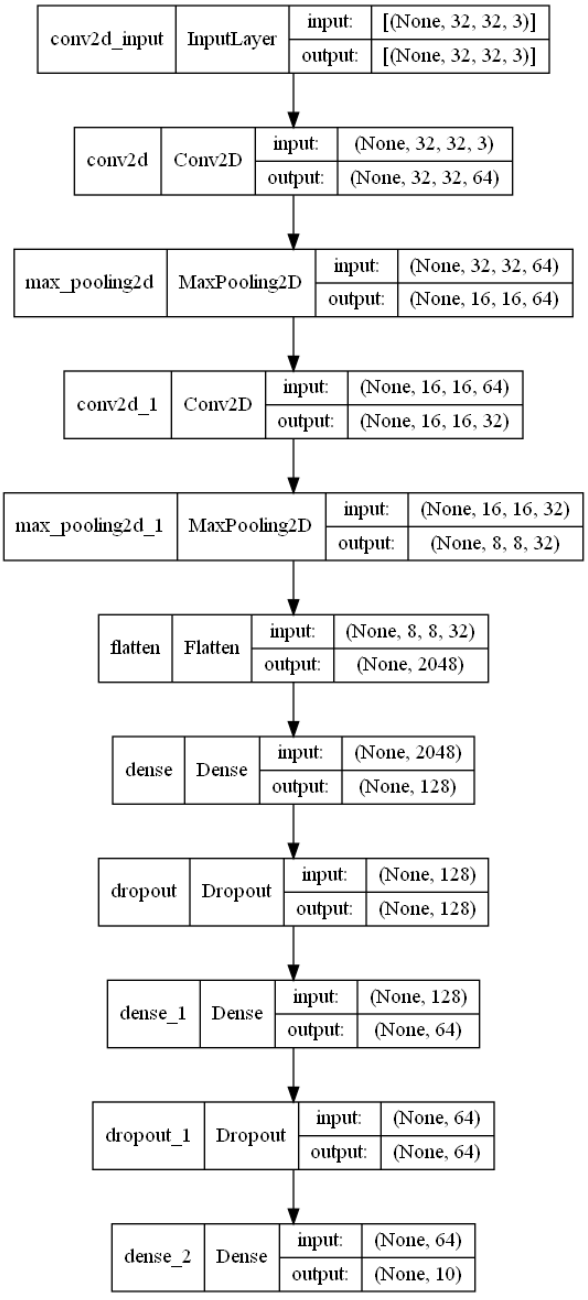

Figure 11: Second CNN Architecture

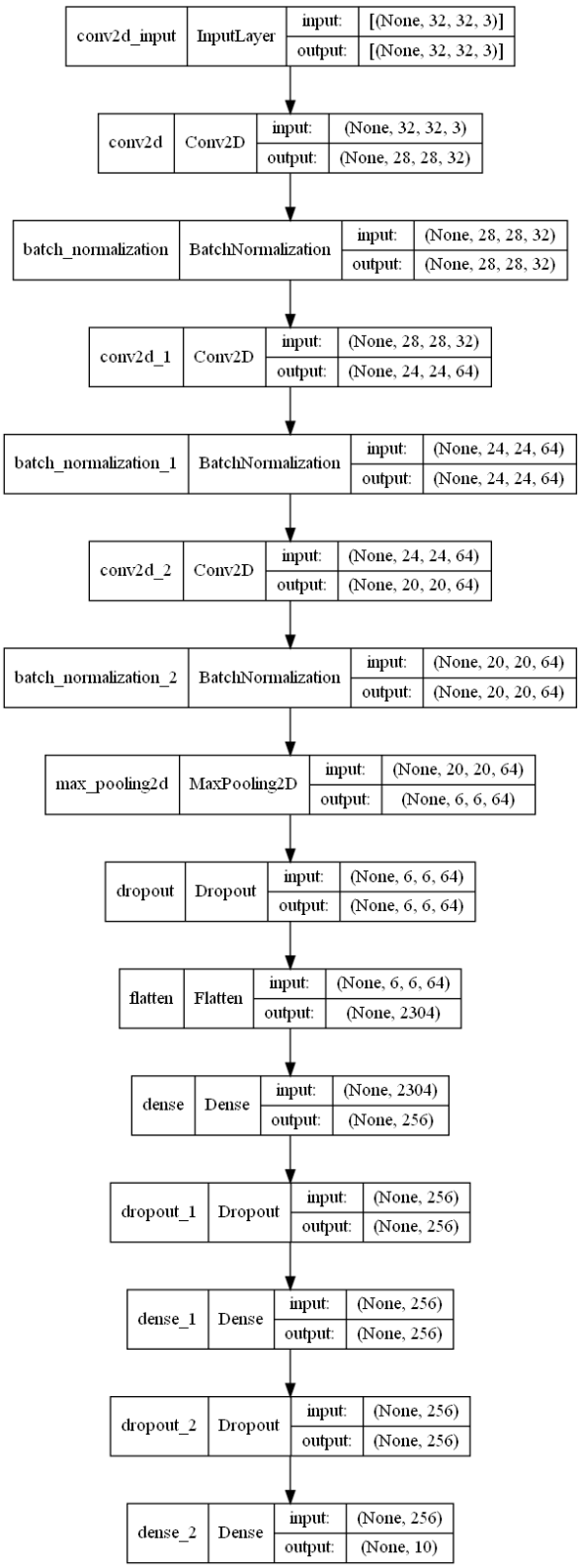

Figure 12: Third CNN Architecture

However, in the earlier periods, the final model does not indicate a rapid growth in accuracy relative to the other models. This collaborates my previous statement that the speed of converging to the stable state shall not be a single determinant of the model selection. Indeed, the final model actually achieves the highest accuracy at the end of the fifth epoch.

6.2 Stage II

In Stage II, the hyperparameters will be reviewed. In general, the set of hyperparameters may also contain the number of convolution layers, the size of the kernel and pooling window, the activation function, etc. However, since we regard the model architecture as a separate component from hyper-parameter tuning, we can assume the model that has been chosen in Stage I and use it to perform tuning. Otherwise, one should expect a very lengthy tuning process provided that every combinations of number of layers and other parameters will be cross-validated before a conclusion can be drawn.

In this stage, I take an empirical approach (as mentioned in section 2.2) to determine the learning rate parameter. Figure 9 and section 2.2.3 provide a thorough explanation about the definition of the learning rate parameter and its impact on training accuracy. As a result, I determine that 10^{-3} is the best candidate among others and it has been chosen in our model.

The choices of bath size often consist of { 32, 64, 128, 256 . . . } but less than or equal to the sample size. I have carefully discussed the decision-making process in section 2.2.1 and conclude that 128 is a good candidate in this project. Unlike the batch size, I decide to use two distinct epoch values for Stage I (6.1) and Stage III (6.3). In the earlier stage, we'd like to assign a relative small value to epochs as long as it is large enough to obtain a promising training accuracy but speeds up the training. Therefore, we let epochs equal to 5 in Stage I and indeed observed that the training accuracy mostly met our expectation of 90%. However, in Stage III, in order to make a convincing model recommendation, we'd rather train the final model over a longer period of time and see if its performance stabilizes over time. Hence, the final epoch value is set to be 10.

6.3 Stage III

In the first two stages, the model architecture and hyperparameters have been determined. The main focus of Stage III is, therefore, data augmentation. This idea is actually inspired by the fact that in the real world different aspects of viewing the street signs lead to different visual outcomes. Researchers simulate the human visual reactions due to this difference and feed some artificial data to the model and evaluate the model performance.

Figure 15 showcases the performance of the final model in two training datasets: one in which the image data are kept intact after data preprocesssing (section 3.1.3), and the other in which preprocessed training data are further augmented by randomly rotating with angle $\theta \in [0, 9]$ and shifting either vertically or horizontally within 10% of its height or width. Refer to the parameters specification in section 2.1.7.

The training outcomes indicate that the final model consistently performs better in the original training dataset over the first 10 epochs. However, the model performance in the testing dataset suggests the other way around.

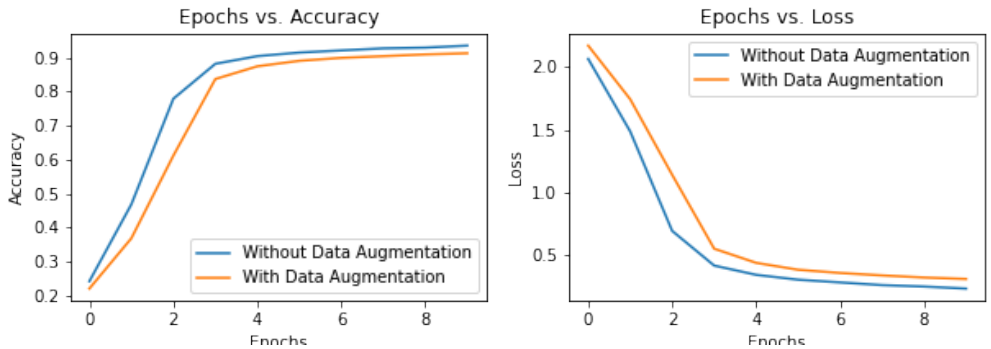

Figure 15: Stage III: Data Augmentation

```
204/204 [==============================] - 15s 76ms/step - loss: 0.1957 - accuracy: 0.9499
test loss: 0.20
test accuracy: 0.95
204/204 [==============================] - 19s 95ms/step - loss: 0.1842 - accuracy: 0.9541
test loss (data augmentation): 0.18
test accuracy (data augmentation): 0.95
```

Figure 16: Stage III: Testing Accuracy

Figure 16 on page 19 implies that data augmentation does improve the predictive accuracy in the testing dataset by supplying augmented training data, even at the expense of losing accuracy in the model training stage. But this contradictory observation actually mitigates our concern about the overfitting problem. The lower accuracy score with data augmentation implies that the regularization occurs during the training, whereas its higher testing accuracy score proves the trained model is superior to the one without data augmentation.

Hence, I recommend to employ the final model in Figure 17 on page 20 because it achieves the best perform among all candidates. Also, data augmentation should be a great addition to data preprocessing stage, as it manges to control overfitting and admits a higher testing accuracy score.

7 Challenges and Limitations

- High demand for computational power. One of the most common challenges for a fully connected neural network is that it requires very high-level computational capacity. Sometimes a sophisticated model seems to be unattainable because the more layers involved usually means the more computations the machine has to undertake. Although the complexity of a model may, unfortunately, lead to the overfitting problem, it is still worth discussing the likelihood of finding such advanced models, as these pre-trained models may eventually help with transfer learning.

- Extensive running time for hyperparameter tuning. In section 2.2, we are aware of the richness of hyperparameters but decide not to consider all of them because of the extensive time required for grid search. Alternatively, we research on some of the outstanding performers and use the knowledge of their performance in related projects to determine the ones we will use. Though it

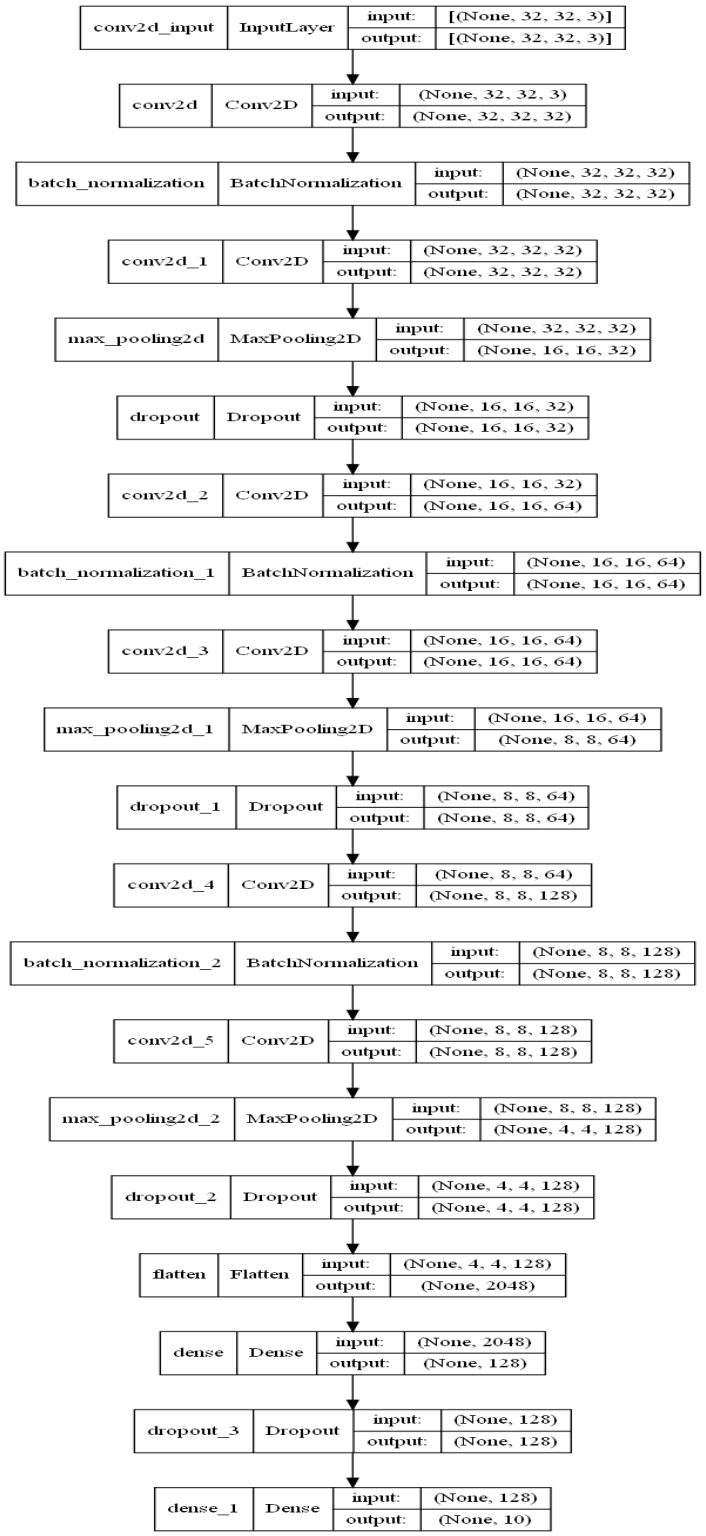

Figure 17: Final CNN Architecture

has proved that this approach can bring us with the desired outcome, a more robust tuning process is expected if possible. In section 8, we even expect to introduce some new hyperparameters and fully automate the process of fine-tuning ConvNets.

- Overfitting problem. Even though the final model successfully achieves the learning objective, we still need to pay attention to the overfitting issue and use proper regularization strategies to address this concern. In this project, data augmentation is considered in Stage III 6.3 as a complementary strategy to deal with overfitting. Using a larger dataset may also help with overfitting. Indeed, data augmentation actually provides more training samples by rotating and twisting the original image data.

8 Conclusion and Improvement

- Use of even more sophisticated CNN architecture. Though the final CNN model is capable of delivering a desirable learning objective, it is still considered simple in design and lack of innovation. Further research on model selection and architecture design is required because it is evident that we may achieve the same or higher performance but consume less computational resources through a better model architecture.

- Sensitivity analysis and comparison analysis. During Stage I (section 6.1), a sensitivity analysis and/or comparison analysis can provide us with more informative guidance regarding hyperparameters tuning and model selection. Grid Search brings about the initial hyperparameters tuning and presents, in theory, an optimal set of hyperparameters in a single model. However, as mentioned earlier in section 2.2, a more robust approach should be taken to perform a comprehensive search over all hyperparameters' domains. This may also lead to the next bullet point.

- Introduction of more hyperparameters for tuning if higher computational power permits. For instance, the model itself can be one of the hyperparameters. Some popular, advanced models include resnet, vgg, resnest, etc. Also, as I already mentioned, hyperparameters that uniquely define layers should also be tuned if the computational capacity allows. For example, we can tune the dropout rate in the dropout layer as a hyperparameter that contribute to overfitting control. This will enrich the candidate model pool by supplying more variations in model architecture.

- More discussions about the overfitting problem and its resolution. Although we have implemented a variety of strategies for reducing overfitting, such as Dropout, Early Stopping and data augmentation, additional regularization techniques should be considered. For instance, we can implement the L-2 regularization for parameters in the fully connected layers.

9 References

- A Beginner's Guide To Understanding Convolutional Neural Networks
- An Intuitive Explanation of Convolutional Neural Networks
- Convolutional Neural Networks (CNNs): An Illustrated Explanation
- Convolutional Neural Networks (CNNs / ConvNets)
- Simple MNIST convnet