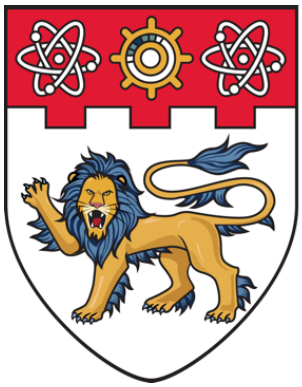


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

AI6121 Computer Vision

Course Project

MNIST Handwritten Digit Recognition

authored by

Ong Jia Hui
jong119@e.ntu.edu.sg
G1903467L

Lim Zhi Xuan
zhixuan003@e.ntu.edu.sg
G2003466A

Tan Mengxuan
mtan099@e.ntu.edu.sg
G1903482L

November 27, 2020

Abstract

MNIST is a classic dataset of hand-written digit images, released in 1999 by LeCun, Yann, et al. [7]. It is often used as a basis for many image classification tasks in Machine Learning. Thos dataset contains 60,000 training images of handwritten digits from zero to nine and 10,000 images for testing.

In this course project report, we will first perform an exploratory data analysis (EDA) to analyze the MNIST dataset at hand. In Section 3, we will document the baseline model, LeNet5 [6], which will be used to compare with our proposed model named AdjustableCNN (AJCNN). We hypothesize that for a dataset with small-sized images like MNIST, we do not need a very deep convolutional network like many of the modern image classification models. Hence, we developed a "tune-able" architecture with a convolutional feature extractor block and a linear classifier block such that the layers and feature map sizes are fully configurable. After some preliminary tuning, we found that our AJCNN8 variant performs the best in our experiments.

After which, we described the various optimizers, schedulers and loss functions we have explored and explained the cross validation process for hyperparameter tuning. In Section 6, we detailed the our experiment results and the final configuration of our model.

To improve our model further, we reviewed and benchmarked against the current state-of-the-art single model for MNIST. Our team also performed error analysis on our proposed model to evaluate the constraints and limitations of our models. As suggested by the state-of-the-art paper, we explored various data augmentation techniques to "expand" the training set so as to improve performance on our final model.

Contents

1 Introduction

MNIST stands for "Modified National Institute of Standards and Technology" is a classic dataset of hand-written digit images used as a basis for many image classification tasks in Machine Learning. It was released in 1999 by LeCun, Yann, et al. [7]. MNIST is a subset of a larger dataset, NIST. However, all of the digits in both train and test set of MNIST have been size-normalized and centered into a fixed-size 28 by 28 image.

2 Exploratory Data Analysis

In this project, we will be using the MNIST dataset from torchvision package. This dataset consists of 60,000 digits and 10,000 digits for the training and testing dataset respectively.

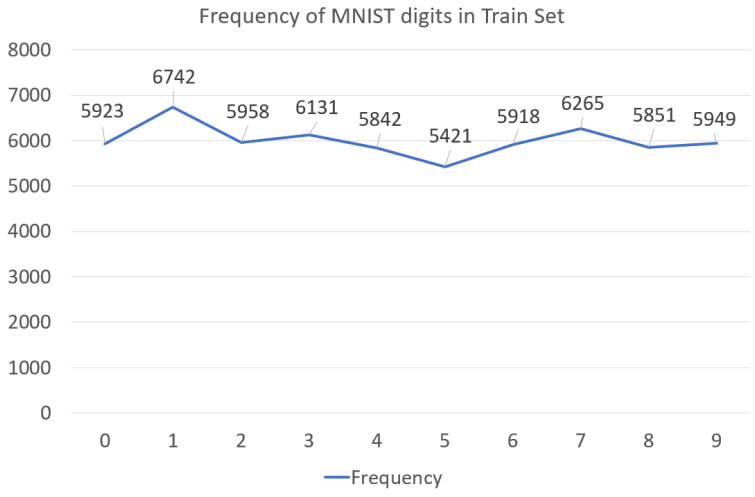


Figure 1: Frequency of MNIST digits in Train Set

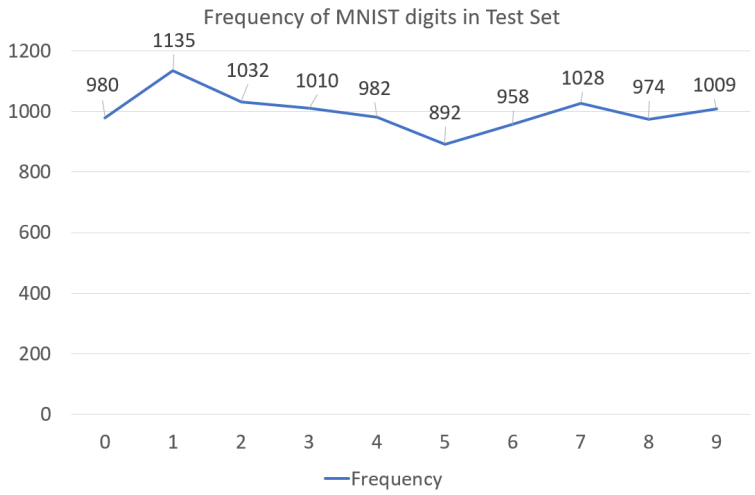
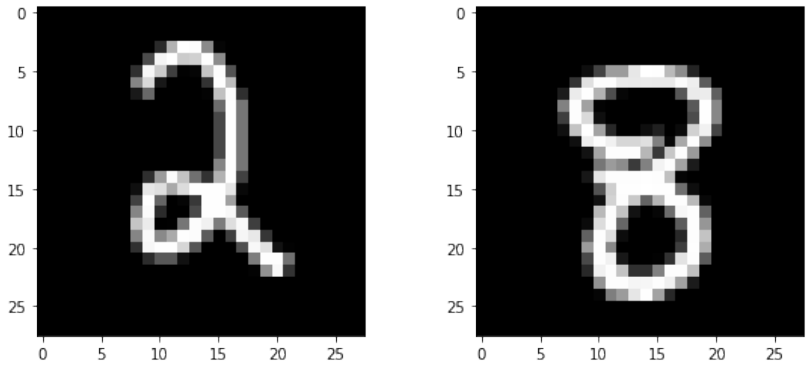


Figure 2: Frequency of MNIST digits in Test Set

The number of samples for respective digits in the training and testing datasets are tabulated and shown in two diagrams in Figure 1 and Figure 2. It can be seen that the training and testing datasets are more or less balanced with slight variations in the number of digits for each class. Specifically, we can see that digit '1' and digit '5' have the highest and lowest frequency respectively in both the train and test set.



(a) Dataset Sample - Label 2 (b) Dataset Sample - Label 8

Figure 3: Samples from MNIST dataset

Each sample in the training and testing dataset has a size of 28 by 28 pixels, which implies that there are 784 features in total. A sample of the digits from the MNIST dataset is shown in Figure 3.

As handwritten digits come in different sizes and have varying number of pixels, it will be apt to first take a look at the average intensity of each pixel in each class - 0 to 9. It is computed by getting the total sum of pixel intensity divided by the total number of the pixels that belong to the same class. The chart in Figure 4 and 5 show the average intensity of each pixel in each class for train and test set respectively. Both figures show that there are differences in intensity across the categories. The digit 1 has the lowest intensity while digit 0 has the highest intensity. Thus, this infers that there are correlations between the digits and the pixel intensity.

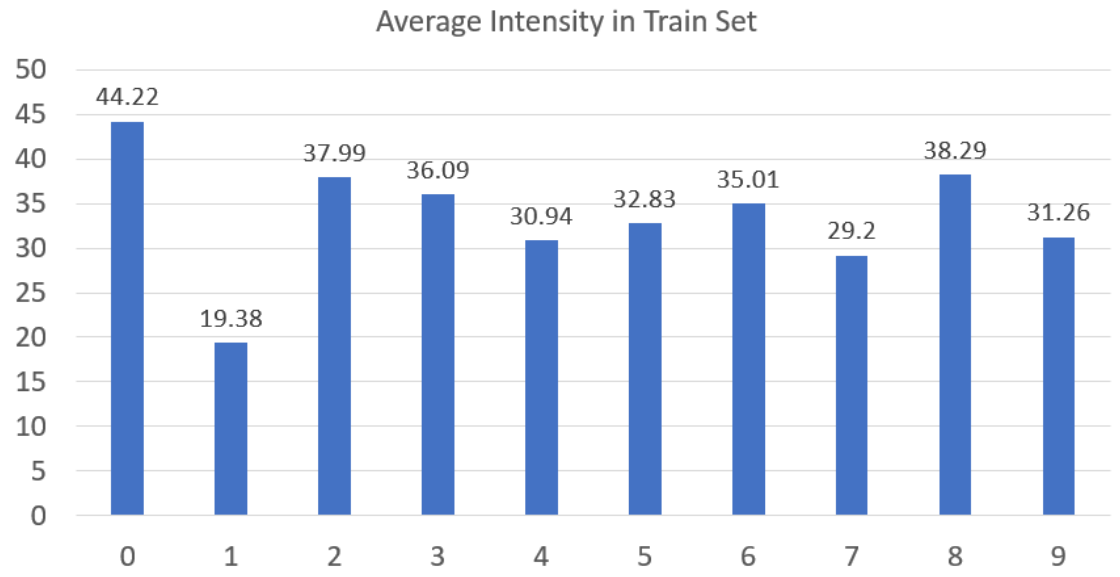


Figure 4: Average Intensity of MNIST digits in Train Set

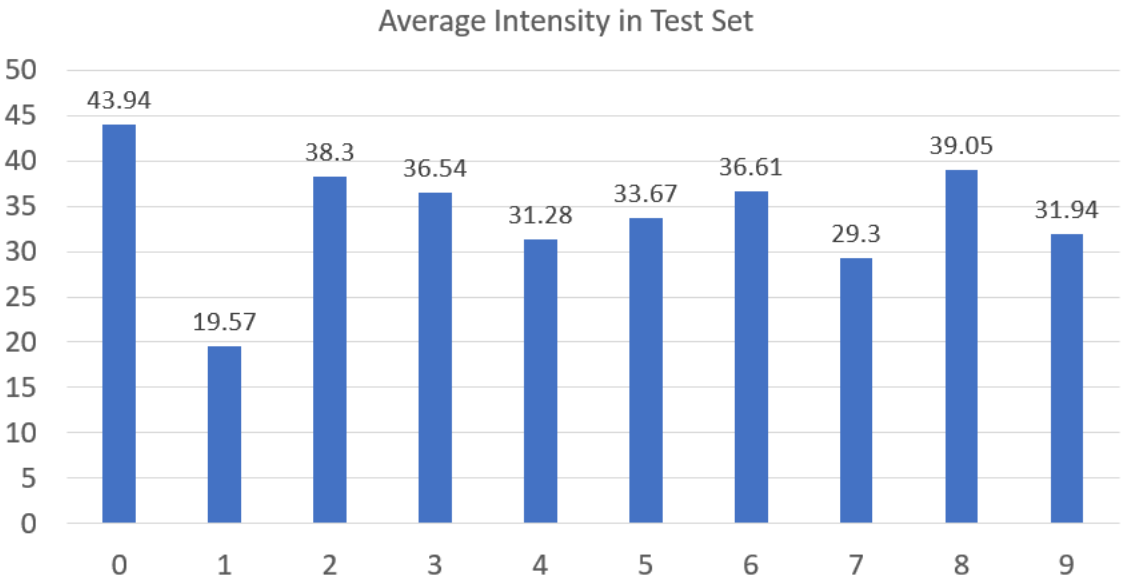


Figure 5: Average Intensity of MNIST digits in Test Set

3 Model

This section will describe the architectures of the networks that are being used for our experiments. We will first describe a baseline model (i.e. LeNet5 [6]) followed by other deeper model architectures proposed by us. After which, we conduct a preliminary round of model selection to select the best model architecture which will be used for further hyper-parameter tuning in the subsequent sections.

3.1 Baseline Model

As our baseline model, we choose to use the classic LeNet5 [6] architecture as it is one of the first deep learning models that was used for the MNIST digits classification task.

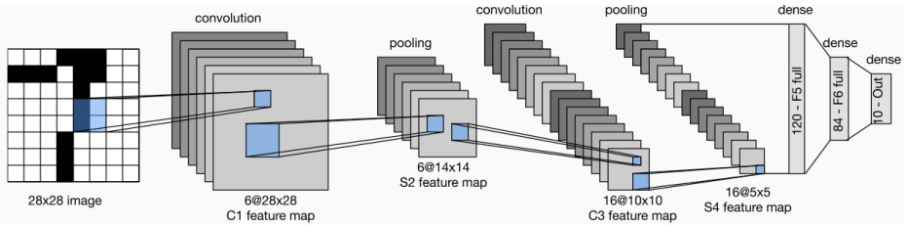


Figure 6: Network architecture of LeNet5.

The architecture of LeNet5 is displayed in Figure 6. We implemented both the original LeNet5 architecture and an improved version with modifications made to improve the dated network. For the original LeNet5 network, it uses average pooling after every convolutional layer and Sigmoid as its activation function. In our modified version of LeNet5, which we termed as "M-LeNet5", we use two sets of alternating 5x5 convolutional layers and 2x2 max pooling layers, followed by three fully connected layers at the end that give 10 outputs which correspond to the output probabilities of the 10 handwritten digits. The rectified linear activation function (ReLU) replaces the Sigmoid activations, after each convolutional and fully connected layer and average pooling is being replaced by max pooling.

3.2 Our Model - AJCNN

3.3 Motivation

We named our model architecture, "AdjustableCNN" or "AJCNN" for short. This name is representative of its design philosophy to create an easily tune-able convolutional network that can be adjusted to be as deep as it can be, while not being overly complex when extracting the image features for the MNIST dataset. As our target dataset consists of relatively small-sized images, using the state-of-the-arts deep image classification networks like ResNet50 [2] may not be suitable as it can result in unnecessary long training times and overfitting issues.

3.4 Model Design

AJCNN is designed with flexibility in mind. It is made up of two major blocks, namely the feature extractor block and the classifier block. The unique point of this architecture is that the number of layers, the feature map sizes and the intertwining layers for both blocks can be easily configured by changing the configuration dictionary. The configuration dictionary has variant name as key and a list of two arrays as value. The first array in the list determines the layers in the feature extractor block, while the values in second array adjusts the classifier block.

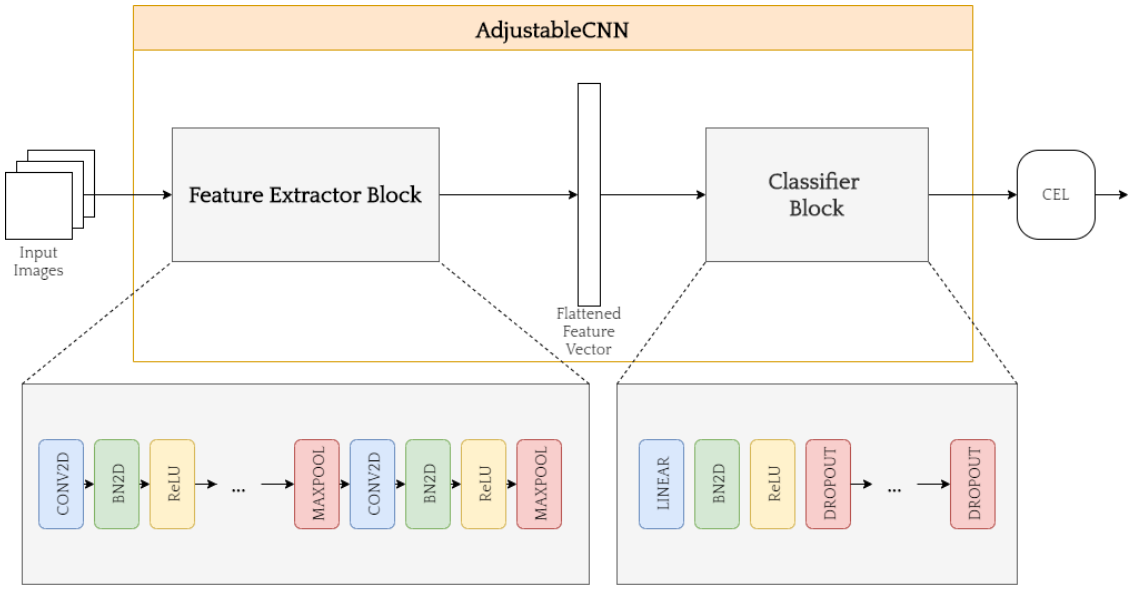


Figure 7: Network architecture of our model, AJCNN.

The feature extractor block consists of configurable 2D convolutional layers, followed by BatchNorm and ReLU activation. Max Pooling layers can be added by appending a "M" value instead of a numerical value into the feature extractor configuration array. On the other hand, the classifier block adds a linear layer with every numerical value set as output size and a Dropout layer with probability of 0.5 with every "D" in the classifier configuration array. Like most CNN architectures, the outputs of the convolutional layers are flattened before passing into the linear layers. The AJCNN was coded to automatically calculate the expected flattened tensor size based on the input image size and the configured layers in the feature extractor block. This tensor size value is passed as input size for the classifier block. This removes the need to manually calculate the input size whenever the feature extractor block is altered, allowing AJCNN to be truly adjustable via the configuration dictionary.

Unlike the baseline model LeNet5, AJCNN is able to stack deeper layers due to its use of Batch Normalization (BN) [4] after every convolutional layer. These BN layers have a regularizing effect and they reduces "internal covariate shift", which helps the network learn better. Furthermore, we added Dropout [10] after every linear layers in the classifier block to prevent overfitting. Dropout randomly sets some neurons to zero during forward pass and forces the network to learn redundant representation of the features. In our AJCNN design, we also carefully defined our weights initialization for our network as we wanted to ensure smooth convergence.

3.5 Model Architecture Tuning

Table 1: Different variants of AJCNN experimented

Variant	Feature Extractor Configuration	Classifier Configuration
AJCNN4	[32, 32, 'M', 64, 64, 'M']	['D', 512, 'D', 512, 'D']
AJCNN6	[64, 64, 128, 'M', 128, 192, 'M', 192, 'M']	['D', 512, 'D', 256, 'D']
AJCNN8	[32, 32, 64, 'M', 64, 128, 128, 'M', 192, 192, 'M']	['D', 128, 'D', 128, 'D']
AJCNN10	[32, 32, 64, 'M', 64, 128, 128, 'M', 192, 192, 256, 'M', 256]	['D', 128, 'D', 128, 'D']

With this flexible architecture developed, we can now experiment with different depths and structure for the feature extractor and classifier blocks. Table 1 shows the some of the AJCNN variants we have designed by changing the configuration dictionary. We named the variants based on the number of convolutional layers by convention, for instance AJCNN4 stands for having a four convolutional layers in the feature extractor block.

Table 2: Comparing different model architectures using default training configurations.

Model	**Optimizer**	**Initial LR**	**Scheduler**	**Loss Function**	**Test Acc**
LeNet5	SGD	0.01	StepLR	Cross Entropy	97.004
M-LeNet5	SGD	0.01	StepLR	Cross Entropy	99.055
AJCNN4	SGD	0.01	StepLR	Cross Entropy	99.363
AJCNN6	SGD	0.01	StepLR	Cross Entropy	99.562
AJCNN8	SGD	0.01	StepLR	Cross Entropy	**99.642**
AJCNN10	SGD	0.01	StepLR	Cross Entropy	99.622

Based on our non-exhaustive experiments as shown in Table 2, AJCNN significantly outperform the baseline LeNet5 model and the modified LeNet5 ("M-LeNet5"). The deeper model AJCNN8 performs better than the shallower variants like AJCNN4 and AJCNN6, while adding additional layers has resulted in poor generalization performance on the test set. Adding additional convolutional layers in AJCNN10 did not improve performance and also led to longer training times. Hence, our final model used for other hyperparameter tuning is termed as AJCNN8.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 28, 28]             320
       BatchNorm2d-2          [-1, 32, 28, 28]              64
              ReLU-3          [-1, 32, 28, 28]               0
            Conv2d-4          [-1, 32, 28, 28]           9,248
       BatchNorm2d-5          [-1, 32, 28, 28]              64
              ReLU-6          [-1, 32, 28, 28]               0
            Conv2d-7          [-1, 64, 28, 28]          18,496
       BatchNorm2d-8          [-1, 64, 28, 28]             128
              ReLU-9          [-1, 64, 28, 28]               0
         MaxPool2d-10          [-1, 64, 14, 14]               0
           Conv2d-11          [-1, 64, 14, 14]          36,928
      BatchNorm2d-12          [-1, 64, 14, 14]             128
             ReLU-13          [-1, 64, 14, 14]               0
           Conv2d-14         [-1, 128, 14, 14]          73,856
      BatchNorm2d-15         [-1, 128, 14, 14]             256
             ReLU-16         [-1, 128, 14, 14]               0
           Conv2d-17         [-1, 128, 14, 14]         147,584
      BatchNorm2d-18         [-1, 128, 14, 14]             256
             ReLU-19         [-1, 128, 14, 14]               0
         MaxPool2d-20           [-1, 128, 7, 7]               0
           Conv2d-21           [-1, 192, 7, 7]         221,376
      BatchNorm2d-22           [-1, 192, 7, 7]             384
             ReLU-23           [-1, 192, 7, 7]               0
           Conv2d-24           [-1, 192, 7, 7]         331,968
      BatchNorm2d-25           [-1, 192, 7, 7]             384
             ReLU-26           [-1, 192, 7, 7]               0
         MaxPool2d-27           [-1, 192, 3, 3]               0
           Dropout-28                [-1, 1728]               0
            Linear-29                 [-1, 128]         221,312
      BatchNorm1d-30                 [-1, 128]             256
             ReLU-31                 [-1, 128]               0
           Dropout-32                 [-1, 128]               0
            Linear-33                 [-1, 128]          16,512
      BatchNorm1d-34                 [-1, 128]             256
             ReLU-35                 [-1, 128]               0
           Dropout-36                 [-1, 128]               0
            Linear-37                  [-1, 10]           1,290
================================================================
Total params: 1,081,066
Trainable params: 1,081,066
Non-trainable params: 0
```

Figure 8: Number of layers and parameters in AJCNN8 variant.
[[32, 32, 64, 'M', 64, 128, 128, 'M', 192, 192, 'M'], ['D', 128, 'D', 128, 'D']]

4 Optimizers, Schedulers, Loss functions

After finalizing the model's architecture as AJCNN8, we may now explore other design options that may further improve the performance of this network. Many optimizers, schedulers and loss functions have been introduced in the recent years to enhance deep learning models' performances and convergence rate. We will select a few of them as our design options for hyper-parameters tuning and later show experiments on how they affect the performances of AJCNN8 in Section 6. In this section, we will describe the chosen optimizers, schedulers and loss functions and the motivations behind using them.

4.1 Optimizers

Optimizers are important in training deep learning models as they ensure the models' training stability and they converge at reasonable rate. The optimizers that are used in our experiments are:

- **Stochastic Gradient Descent (SGD)**: The most commonly used algorithm for optimizing neural networks where the parameters of the models (denoted by θ) are updated in the direction of their gradients with respect to the loss function (denoted by g_t) at each training step for a minibatch of training data. Equation 1 illustrates how this is done. α is the learning rate as it controls how large a step to take in the direction of the gradients when the parameters are being updated.

$$\theta_{t+1} = \theta_t - \alpha g_t \tag{1}$$

- **RMSProp**: It stands for the Root Mean Square Propagation. It tries to resolves diminishing learning rates during updates by using a moving average of the squared gradient.

$$\theta_{t+1} = \theta_t - \frac{\alpha g_t}{\sqrt{(1-\gamma)g_{t-1}^2 + \gamma g_t + \epsilon}} \tag{2}$$

  Equation 2 shows the parameters updating process using RMSProp. γ is the decay term that takes value from 0 to 1. The denominator represents the exponentially weighted average of the squares of the gradients. Hence learning rate in RMSProp gets adjusted automatically given the gradients in the previous step.

- **Adam**: Adam [5] is currently the most popular adaptive optimization algorithm for deep learning. It first updates the exponential moving averages of the gradient(m_t) and the squared gradient(v_t) which is the estimates of the first and second moments. The updates equations are shown below:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{3}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{4}$$

  m_t and v_t are estimates of the first and second moments respectively while β_1 and β_2 are both hyper-parameters between 0 to 1. Bias corrections are also being performed on m_t and v_t to prevent moment estimates to be biased around 0 especially during the initial timesteps:

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{5}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{6}$$

  $\hat{m}_t$ and $\hat{v}_t$ are bias corrected estimates of the first and second moments respectively. Finally the model parameters are updated as follows:

$$\theta_{t+1} = \theta_t - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \tag{7}$$

  Other than its effective updating scheme, Adam is also computationally efficient and has very little memory requirement.

4.2 Schedulers

Although several optimizers such as RMSProp and Adam mentioned in Section 4.1 can adaptively adjust the learning rate during training based on the gradients of model's parameters, we can make our learning rates more adaptive by further adjusting them based on the number of **current epochs** and **validation measurements**. This can be done via the use of schedulers in our experiments. Below are some schedulers that will be experimented:

- **StepLR**: This scheduler decays the learning rate of model parameters by a factor of γ after every *step_size* epochs where γ and *step_size* are predefined hyper-parameters.

- **ReduceLROnPlateau**: ReduceLROnPlateau monitors the model's performance on the validation set during training before adjusting the learning rate. If the validation performance does not improve after *patience* number of epochs, the current learning rate will be reduced by a factor of γ. Both *patience* and γ are predefined hyper-parameters.

- **CosineAnnealingLR**: The Cosine Annealing [8] is a type of learning rate schedule that has the effect of starting with a large learning rate that is relatively rapidly decreased to a minimum value before being increased rapidly again.

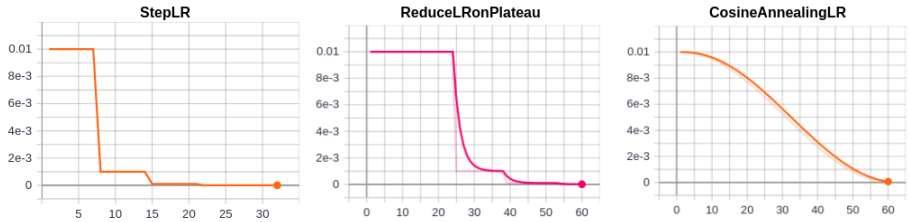


Figure 9: Visualisations on how the learning rate decay when the different schedulers are used. Vertical axis displays the learning rate and horizontal axis displays the training epoch.

Figure 9 illustrates how the learning rate decay when the different schedulers are used, with an initial starting learning rate of 0.01. In general, StepLR and ReduceLROnPlateau may cause sudden drops in learning rate while CosineAnnealingLR will result in a much smoother decay.

4.3 Loss Functions

The loss function defines how the network compute the error the model is making during the training process and the optimizer aims to minimise this error during training. It can also be used to control how parameters are updated and how gradients are backpropagated in the network. We select two loss functions to experiment with when training our models:

- **Cross Entropy Loss**: It is a commonly used loss function for multi-class classification problem. Minimising the Cross Entropy loss in equivalent to maximising the log likelihood of the data. The loss function is defined as:

$$L = -\frac{1}{m}\sum_{i=1}^{m} y_i \log(\hat{y}_i) \tag{8}$$

  where y_i is the one-hot representation of the ground-truth label for the i^{th} training sample and $\hat{y}_i$ is the model prediction probabilities of each class for the same sample. m is the total number of available training samples.

- **Label Smoothing Cross Entropy Loss**: Although the distribution of classes in the MNIST train and test sets is more or less balanced (Figure 1 and 2), there is still a gap between the number of samples from several classes. For instance, the number of samples for digit '1' is 20% more than that of digit '5'. This may lead to a tendency for the model to be overconfident in predicting frequent classes and less confident in predicting other classes where the ground-truth labels are less seen. Therefore we want to use an advanced technique known as Label Smoothing [9] to regularize the model for outputting high probability scores when predicting frequent target classes, and on the other hand, encourages the model to learn to identify digits with fewer samples. This technique ultimately creates a **new** set of ground-truth labels for computing the Cross Entropy loss. The equation for creating the new ground-truths are as follow:

$$y_i^{new} = (1 - \alpha) * y_i + \frac{\alpha}{K} \tag{9}$$

Equation 9 shows how we can obtain the smoothed label y_i^{new} from the original label y_i for a training example. α is a hyper-parameter that determines the amount of smoothing. If $\alpha = 0$, we obtain back the original one-hot encoded label. When $\alpha = 1$, we get a uniform distribution across all target classes. K represents the total number of available target classes, in our case is 10. After obtaining y_i^{new}, we replace y_i of Equation 8 with y_i^{new} to calculate the label smoothing version of the Cross Entropy loss.

5 Cross Validation for Hyper-parameters Selection

Moving on, we now briefly describe the process of hyper-parameters selection given the different design options shown in Section 4.

We mainly adopted cross validations for hyper-parameters selection. First, the original MNIST training dataset is split into both 'Train' and 'Valid' datasets where 'Train' contains 48000 samples and 'Valid' has 12000. Next, during the model training phase, we tune the hyper-parameters or use different optimizers and schedulers for our models and monitor their performances on the 'Valid' set. The models' hyper-parameters that result in the best accuracy score on the 'Valid' set are saved. Finally, we use the best set of hyper-parameters to train the model on the entire MNIST training dataset of 60000 images and then evaluate the model on the unseen MNIST testing dataset of 10000 images. Section 6 will report the model's performance on the unseen testing dataset with various hyper-parameters and design options.

6 Experiment Results

This section will display the results of model selection with the various hyper-parameters and design options. Given the relatively large number of design options described in Section 4 and the high number of possible hyper-parameters, it will be extremely time-consuming to iterate through all different combinations of design options and hyper-parameters. Therefore, we propose a greedy approach in analysing and selecting design options and hyper-parameters. The greedy approach is outlined as follows:

- Step 1: Using the Cross Entropy Loss as described in Section 4.3 but **without** any scheduler, we compare the performances of models with the various optimizers shown in Section 4.1 and different initial LR (learning rate).
- Step 2: Next, we compare the effects of various schedulers (Section 4.2) on performances by fixing the optimizer and initial LR to those that perform the best in Step 1.
- Step 3: Finally, with the best combination of optimizer, initial LR and scheduler found in Step 1 and 2, we compare the model's performances on the two different loss functions mentioned in Section 4.3.

At the end of this greedy approach, we will use the design option and hyper-parameters that give the best performance to benchmark against the State-of-the-art and further improvement that will be shown in the later sections. The results of each step from this greedy approach will be shown in the following 3 subsections.

Table 3: Comparing effects of various optimizers and initial LR on model's performances .

Model	Optimizer	Initial LR	Scheduler	Loss Function	Test Acc
AJCNN8	SGD	0.001	None	Cross Entropy	99.463
AJCNN8	RMSProp	0.001	None	Cross Entropy	98.935
AJCNN8	Adam	0.001	None	Cross Entropy	99.214
AJCNN8	**SGD**	**0.01**	None	Cross Entropy	**99.562**
AJCNN8	RMSProp	0.01	None	Cross Entropy	95.830
AJCNN8	Adam	0.01	None	Cross Entropy	98.806

6.1 Comparing Optimizers and Initial LR

Table 3 displays the effects of various optimizers and initial LR on model's performances. It can be observed that for the adaptive optimizers (i.e., RMSProp, Adam), models that are initialized with a LR of 0.001 performs better than when the LR is initialized with 0.01. And Adam performs better than RMSProp for both learning rates. However, we see the opposite behaviour for the non adaptive optimizer: SGD. Interestingly, the model performs better when SGD is used together with an initial LR of 0.01 as compared to when an initial LR of 0.001 is used. Furthermore, using SGD and an initial LR of 0.01 gives the best performance amongst the other settings.

6.2 Comparing Schedulers

Table 4: Comparing effects of various schedulers on model's performances .

Model	Optimizer	Initial LR	Scheduler	Loss Function	Test Acc
AJCNN8	**SGD**	**0.01**	**StepLR**	Cross Entropy	**99.642**
AJCNN8	SGD	0.01	ReduceLROnPlateau	Cross Entropy	99.631
AJCNN8	SGD	0.01	CosineAnnealingLR	Cross Entropy	99.622

Using the best setting of optimizer (i.e., SGD) and initial LR (i.e., 0.01) obtained from the previous step of our greedy approach in Section 6.1, we now compare the effects of the different schedulers. The results are shown in Table 4. In general, we can see that using either of the three schedulers will lead to an improvement in model's performances. Using the StepLR gives the best improvement.

6.3 Comparing Loss Functions

Table 5: Comparing effects of various loss functions on model's performances .

Model	Optimizer	Initial LR	Scheduler	Loss Function	Test Acc
AJCNN8	SGD	0.01	StepLR	Cross Entropy	99.672
AJCNN8	**SGD**	**0.01**	**StepLR**	**Label Smoothing Cross Entropy**	**99.682**

Finally, we can now compare the model's performances when different loss functions are used by fixing the optimizer, initial LR and scheduler at their best settings which are obtained in the previous two steps shown in Section 6.1 and Section 6.2. From Table 5, we can observe that using the Label Smoothing Cross Entropy loss function helps to improve the model's performance slightly by

introducing certain regularization effect as described in Section 4.3. We set α of Equation 9 to 0.001 to achieve this result.

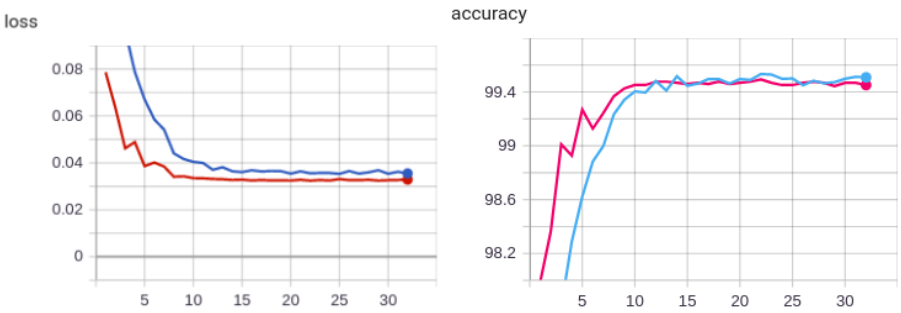


Figure 10: The loss and accuracy curves on both the train (blue) and validation (red) when AJCNN8 is trained using SGD, initial LR of 0.01, the StepLR scheduler and Label Smoothing Cross Entropy Loss.

Figure 10 shows the loss and accuracy curves on both the train and validation sets when AJCNN8 is trained using the best settings of optimizer, initial LR, scheduler and loss function. In the subsequent sections, we will be using this setting to benchmark against the state-of-the-art (Section 7), error analysis (Section 8) and possible improvements (Section 9).

7 Benchmark with the State-of-the-Art

In this section, we will look into the current state-of-the-art approach for MNIST digit predictions from 'A branching and merging convolutional network with homogeneous filter capsules' (we called it BMCNN in short) [1] by A. Byerly, T. Kalganova, I. Dear which was published recently on 13 July 2020. First, we study its network architecture and unique features of the model that enable it to achieve the highest accuracy to date. Next, we look into some data augmentation techniques proposed to improve the model's performance, this also serve as a hint for us to improve our AJCNN8 performance in Section 9. Finally, we report the performances of BMCNN and compare it with that of our AJCNN8 model.

7.1 Unique Features of the Network Architecture

In total, there are 3 unique characteristics of the state-of-the-art architecture in which the authors have done it differently compared to other conventional approaches.

1. The author deliberately avoided using any pooling operations which was usually found in many convolutional neural networks. The reason is that pooling is a down-sampling technique which removes information and it should only be used when there is insufficient computational power [3]. In the case of MNIST dataset, the size of the images is relatively small – 28 by 28 pixels. Thus, there is adequate computational power for images of this size. Instead of using downsampling, the network architecture does not use any zero-padding operations and thus, this reduces the horizontal and vertical dimensions by 2 pixels.

2. Another unique feature is that the network architecture of the state-of-the-art uses multiple branches which is shown in the figure below.

As seen in Figure 11 on the next page, there is a total of 3 branches in the network and each branch consists of different number of 3 x 3 convolutions and filters. The number of convolutions, filters and effective receptive field of the original image pixels are displayed in the table below.

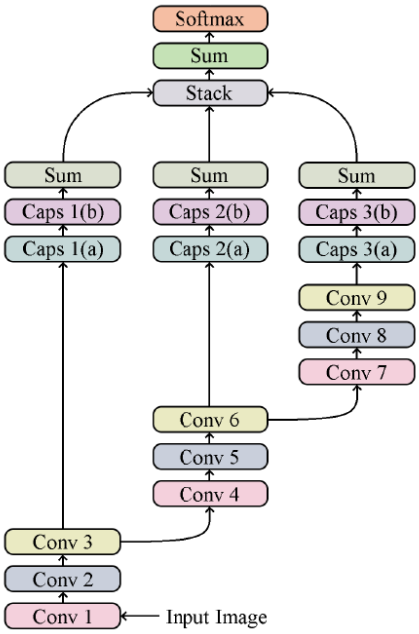


Figure 11: Overall Network Architecture of State-of-the-Art.

Table 6: Hyperparameters of Network Model.

	No. of 3 x 3 Convolutions	**No. of Filters**	**Effective Receptive Field**
First Branch	3	64	7
Second Branch	6	112	11
Third Branch	9	160	15

3. Unlike many convolutional neural networks which use fully connected layer to flatten the output of the convolutions, the state-of-the-art network architecture uses the concept of homogenous vector capsules which is known as "Caps" block in Figure 11. It transforms each filter into a vector and then performs element-wise multiplication of every vector with a set of weight vectors of the same length. Thus, a weight vector which consists of n filters by m vector capsules is obtained and summed across the filters to form the second capsule. Subsequently, the components of the vector are summed to reduce each vector to a single value per class in the form of output logits. An illustration is shown in Figure 12 on the next page.

7.2 Data Augmentation

Besides working on the network architecture, the authors also focus on the using data augmentation strategy. In total, there are 4 data augmentation methods that was used to train the images.

- **Rotation**. The training images are randomly rotated between -30 and 30 degrees. To obtain the angle of rotation, a value is drawn from a random normal distribution with mean 0

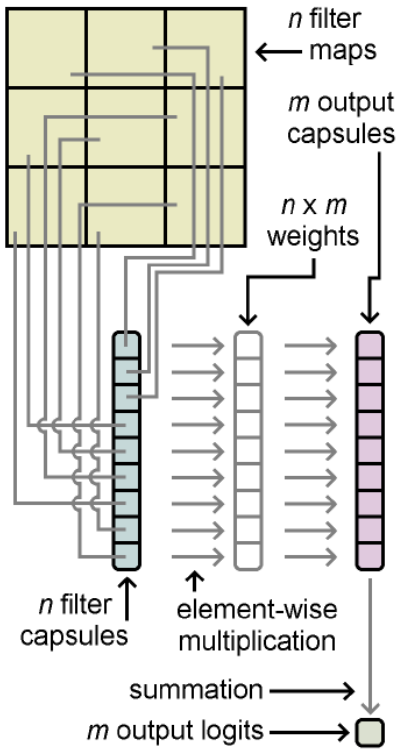


Figure 12: Transformation from filter maps through homogeneous filter capsules to the output logits.

and standard deviation 0.33, with a minimum of -1 and maximum of 1 and then multiplied by 30.

- **Translation**. A translation operation is used to shift the training image by up to 2 pixels in horizontal, vertical or both directions. The maximum value of 2 pixels is set so that the white portion in the image which consists of the important information of the digits is not being cut off.

- **Width**. The author also randomly adjusted the width of the training images by compressing the width of each image and then added equal zero padding on both side of the image so to imitate the handwriting of different people.

- **Random Erasure**. The intensity values of a 4 by 4-pixel grid in the centre 20 by 20 grid of the training images are randomly set to 0. This is to expose the model to more variations of connectedness within the strokes that form the digits.

7.3 Training of State-of-the-Art Model

For the training of state-of-the-model, the authors picked a starting learning rate of 0.001 and then apply an exponential decay rate of 0.98 per epoch with a minimum learning rate set at 1e-6. In addition, it is trained with the Adam optimizer.

7.4 State-of-the-Art Results

The authors conducted two tests for two models - single model and ensemble model and achieved the best accuracy of 99.79% and 99.84% respectively. Three experiments were then conducted for each model to observe the impact of the branch weights on the test accuracy. The three experiments conducted are equal weighting for the three branches, learnable with randomly initialized branch weights and learnable with branch weights initialized to one.

The highest accuracy for state-of-the-art (single model) is obtained by using learnable with randomly initialized branch weights. For the case of the ensemble model, it is achieved by using learnable branch weights that are initialized to one.

Table 7: Benchmarking our model AJCNN8 with the State-of-the-Art.

Model	**Test Acc**
BMCNN (Single)	99.79
BMCNN (Ensemble)	99.84
AJCNN8 - *ours*	99.68

Table 7 summaries the performances of BMCNN together with our best AJCNN8 model's performance. There is still some gap in terms of performance between our model and the state-of-the-art. For instance, our AJCNN8 model makes about 11 more wrong predictions as compared to the single BMCNN model. Therefore, in the next two sections, we will look into the constraints of our model and how we can improve on it further.

8 Constraints and Error Analysis

To understand the constraints of our models, we performed a thorough error analysis to evaluate the reasons for our model's failed predictions. We started by analyzing the performance of the model for each of the digits.

Table 8: AJCNN8 MNIST test set accuracy by digits

Digit	Accuracy (%)
0	99.7959
1	100
2	99.8062
3	99.7029
4	99.6945
5	99.4394
6	99.3736
7	99.5136
8	99.7946
9	99.2071

$$\textbf{Acc} = \frac{\text{Number of correct predictions}}{\text{Number of predictions}} \tag{10}$$

Table 8 shows the accuracy of each digit computed by tabulating their true positive scores. The accuracy of each digit is computed by tabulating the number of predictions that matches its label divided by the number of predictions made for the label as shown in Equation 10. Digit "1" has the highest accuracy rate of 100%, which means that our model did not make any mistake when predicting digit "1". On the other hand digit "9" has the lowest accuracy rate of 99.2%.

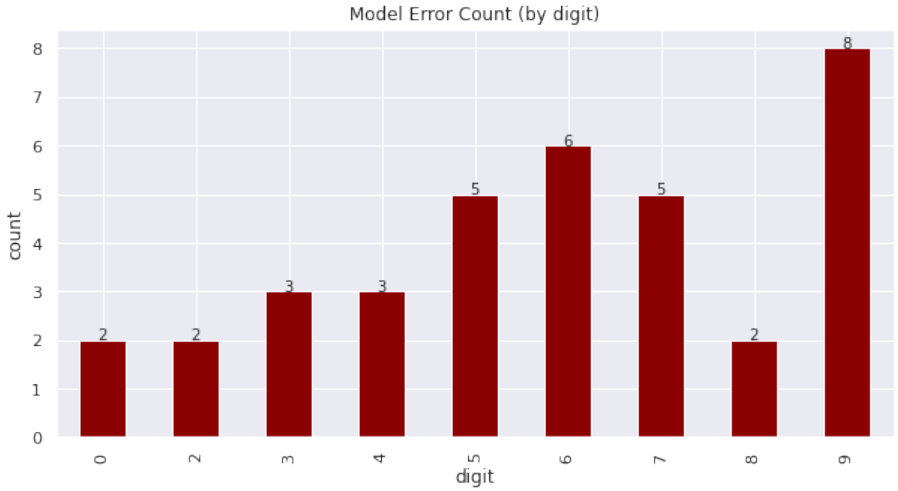


Figure 13: Testset error count (by digits)

To further analyze our model's poor performance on the digit "9", we also plot the distribution of prediction error count made by our model on the MNIST test set as shown in Figure 13. As digit

"1" has a 100% accuracy prediction rate, it is not shown in the figure.

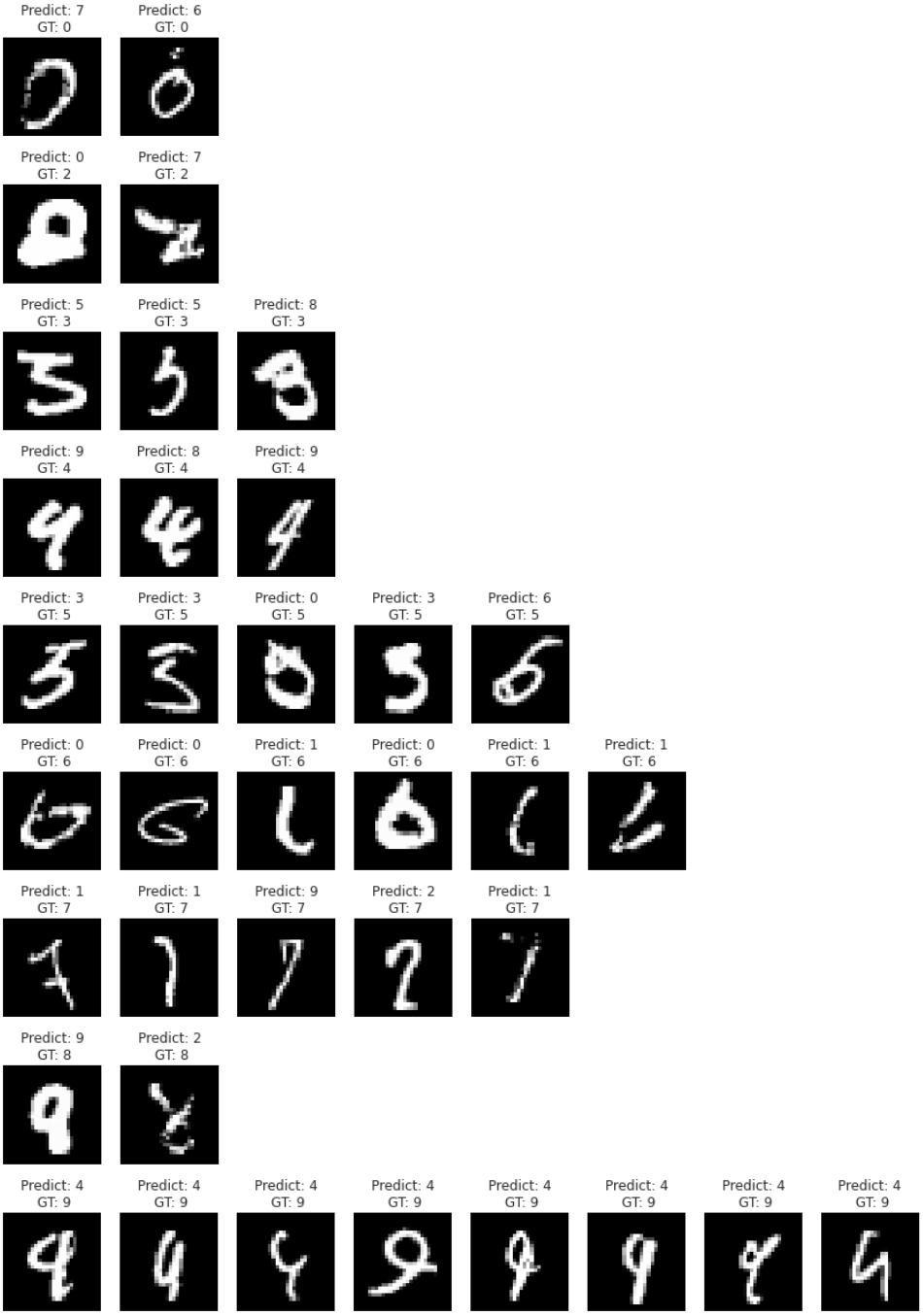


Figure 14: Misclassified examples in MNIST testset

As Digit "9" has made a total of 8 false predictions, we wanted to visualize the reasons behind our model's difficulty in making the right prediction for digit "9". We plotted out our misclassified examples in Figure 14, which showed that in many of the failed cases, our model fails to differentiate

between “4” and “9”.

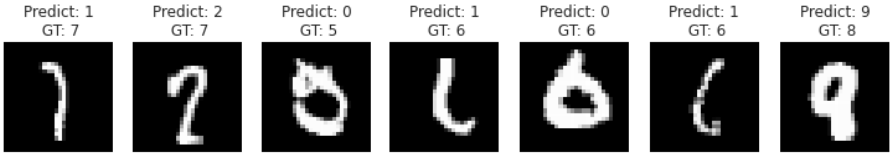


Figure 15: Some hard examples

However, we do find many of the failed examples are hard examples that may even fail human evaluation. We extracted some of them and plotted them in Figure 15. For instance, the first subimage with a ground truth of “7” as an additional tail, which makes it looked like the predicted “2” and the last subimage with a ground truth of “8”, which was predicted as “9” as it is missing one out of the distinctive two holes in the image.

9 Possible Improvements

After analysing the constraints and errors made by our models, we have understood that the model needs to perform better on more difficult and ambiguous samples in order to further improve its overall accuracy. Hence, inspired by [1], we will describe several techniques to train our model with augmented data samples in order to improve its robustness.

9.1 Data Augmentation

Data Augmentation is a common technique in the field of Computer Vision to increase the diversity of the training dataset. By doing so, it helps to boost the test accuracy of the dataset. In the subsequent sections, different data augmentations techniques are being experimented to investigate the effect of each technique on the testing accuracy. The testing accuracy of each data augmentation technique is consolidated and put in section 9.2.

9.1.1 Random Scale

In each image, pixels with high intensity, which form up the respective digits are usually centered. Thus, random scale while keeping centre invariant can be applied on the training set images. In this case, we set the random scale to be between 0.9 and 1.1 as shown in the diagram below. The 2 figures on the left show the original images while the 2 figures on the right display the training images after random scale is applied on the training images.

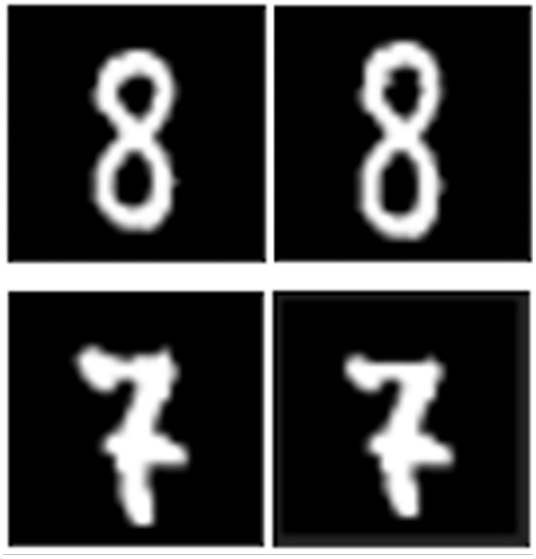


Figure 16: Original Image (left) and Image with Random Scale (right)

9.1.2 Random Rotation

The next data augmentation technique that we have experimented is random rotation. In this project, we set the hyperparameter – range of rotation angles to be between -10 degrees and 10 degrees. Figure 17 on the next page shows two examples which are extracted from the training dataset before and after random rotation is applied on the respective images.

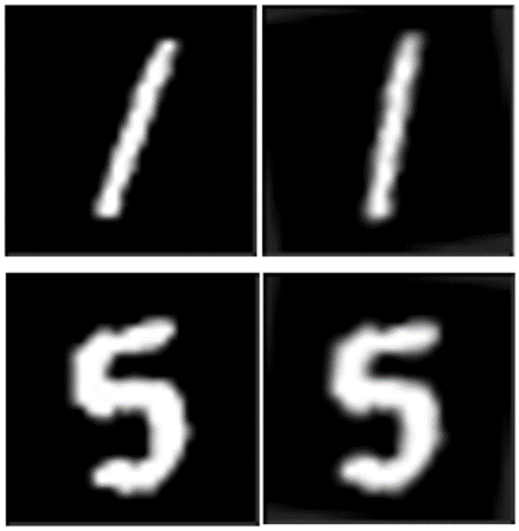


Figure 17: Original Image (left) and Image with Random Rotation (right)

9.1.3 Random Translation

Random Translation is another technique that was experimented for the training dataset. Both horizontal and vertical directions are used for the translation of the training images. The hyperparameter - maximum absolute fraction which determines the range of shift is set at 0.1 of the height and width of the images in both directions. Two images of the training dataset before and after the technique is applied are shown in Figure 18.

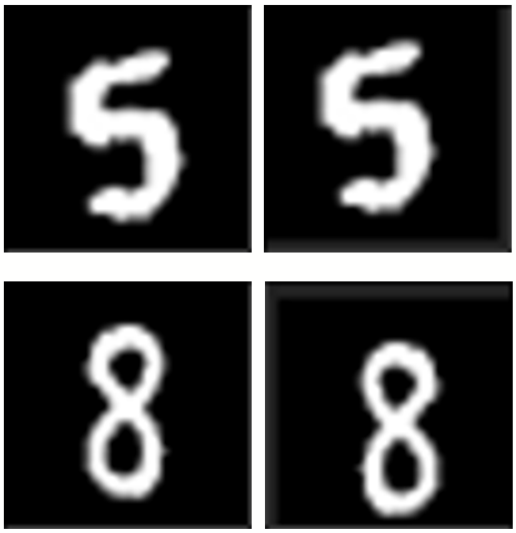


Figure 18: Original Image (left) and Image with Random Translation (right)

9.1.4 Random Crop and Resize

As mentioned previously, the bright pixels which form up the digits are usually at the centre of the images. Thus, we can make use of this property and apply random cropping on the image followed by resizing the image to 28 by 28 which is the size of the original image given a probability. The hyperparameter - size of the cropped image is set at 24 while the probability is 0.3.The figure below shows a comparison between the original image and image after the technique is applied on the original image.

Figure 19: Original Image (left) and Image with Random Crop and Resize (right)

9.1.5 Centre Crop and Resize

Beside using random crop, we also experimented with centre crop and resize since both techniques are similar. For the hyperparameters of centre crop and resize, it is experimented with the same hyperparameters that was mentioned in section 9.1.4. As seen in the figure below, the size of the digits increases with respect to the centre of the image.

Figure 20: Original Image (left) and Image with Centre Crop and Resize (right)

9.1.6 Random Perspective

Random Perspective performs perspective transformation to the images randomly given a probability. For this project, the probability is set at 0.3. As seen in Figure 21, the images are tilted at a given angle.

Figure 21: Original Image (left) and Image with Random Perspective (right)

9.1.7 Gaussian Blur

Gaussian Blur is another common data augmentation technique that is used in Computer Vision and it is also experimented in the MNIST training dataset. In the Gaussian Blur library function, it consists of 2 hyperparameters - kernel size and and the range of standard deviation which are used to create kernel for blurring. The kernel size is set at 5 while the standard deviation is set at the range of between 0.5 and 1. The image of the training images before and after Gaussian Blur is applied is shown in the figure below.

Figure 22: Original Image (left) and Image with Gaussian Blur (right)

9.2 Test Accuracy of Data Augmentation

In general, the test accuracy for the respective data augmentation techniques are compiled and shown in Table 9. Among the data augmentation techniques, random rotation is the best augmentation technique performed on AJCNN8 as it achieved a test accuracy of 99.684%.

Table 9: Compilation of Test Accuracy Results

Data Augmentation Techniques	**Test Acc**
Random Scale	99.672
Random Rotation	**99.684**
Random Translation	99.632
Random Crop and Resize	99.622
Centre Crop and Resize	99.602
Random Perspective	99.652
Gaussian Blur	99.592

10 Final Model Results

From Table 9, the best performance of 99.684% is only slightly higher than the result of 99.682% shown in Table 5 of Section 4.3 when no data augmentation is being applied. This improvement is insignificant to conclude that data augmentation is effective. Therefore, we explore other possible ways to improve the performance of the model further. In particular, we look at tuning the value of α of Equation 9 which represents the label smoothing factor. Table 10 summaries the results of our models using different values of α.

Table 10: Final model result after tuning smoothing factor α of Label Smoothing loss function.

Model	**Augmentation**	**Loss Function**	**Smoothing Factor α**	**Test Acc**
AJCNN8	None	Label Smoothing Cross Entropy	0.001	99.682
AJCNN8	Random Rotation	Label Smoothing Cross Entropy	0.001	99.684
AJCNN8	Random Rotation	Label Smoothing Cross Entropy	0.0005	99.632
AJCNN8	**Random Rotation**	**Label Smoothing Cross Entropy**	**0.003**	**99.701**

From Table 10, we can see the effect of α on the model's performance when data augmentation is being used. Recall the meaning of α described in Section 4.3, it represents how aggressive we smooth our ground-truth labels. A higher imbalance class distribution may require a higher value of α. In Table 10, we observe that increasing the α leads to a better performance. This tells us that data augmentation might have introduced more samples of a certain class due to its randomness when we apply it during training, therefore a higher value of α may be needed to regularize this effect.

In summary, our best trained model makes use of the AJCNN8 network architecture with random rotation performed on the train set as data augmentation, using SGD optimizer, StepLR scheduler of an initial LR of 0.01 and the Label Smoothing Cross Entropy loss function with an smoothing factor α of 0.003.

10.1 Final Model Error Analysis

We also evaluated our final trained model, AJCNN8 and its incorrect predictions to visualize the differences after performing data augmentations and by comparing it with the error analysis done in Section 8.

Table 11: AJCNN8 MNIST test set accuracy after training with data augmentations. The second column's values are obtained from Table 8 of Section 8 when no data augmentation is applied.

Digit	AJCNN8 Acc	Aft Augmentations Acc
0	99.7959	99.7959
1	100	100
2	99.8062	**99.9031**
3	99.7029	99.7029
4	99.6945	**99.8981**
5	99.4394	99.4394
6	99.3736	99.3736
7	99.5136	**99.7082**
8	**99.7946**	99.6920
9	99.2071	**99.4054**

Table 11 compares the accuracy improvement after training with random augmentations and increasing in label smoothing factor. The final model achieved better scores than previous model in digits "2", "4", "6" and "9". However, our model made one additional mistake for digit "8".

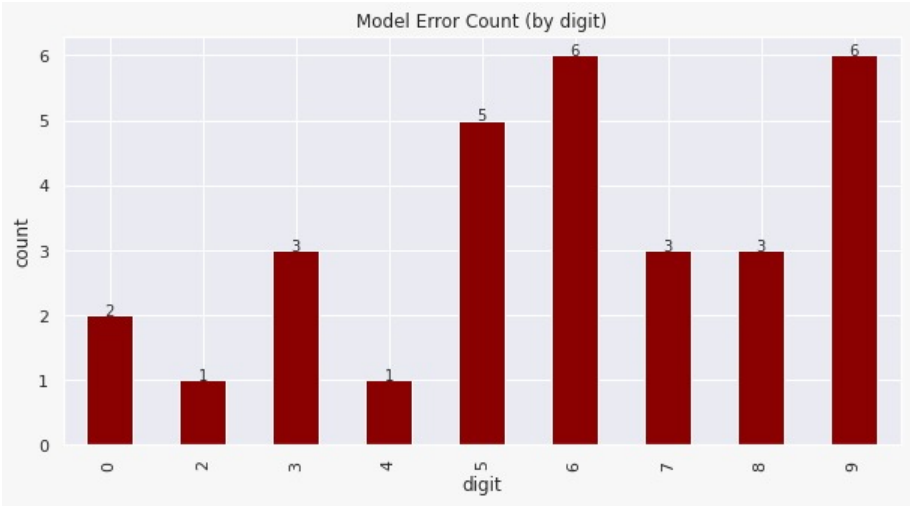


Figure 23: Testset error count (by digits) on our final model

As compared to Figure 13, Figure 23 shows a general reduction in number of misclassified digits.

Figure 24 shows all the incorrect predictions our final model has made. We found that our model improved in its ability to differentiate between "4" and "9" tremendously after training with the augmented images. Furthermore, our final model also managed to correctly predict some of the hard examples we have extracted out in Figure 14.

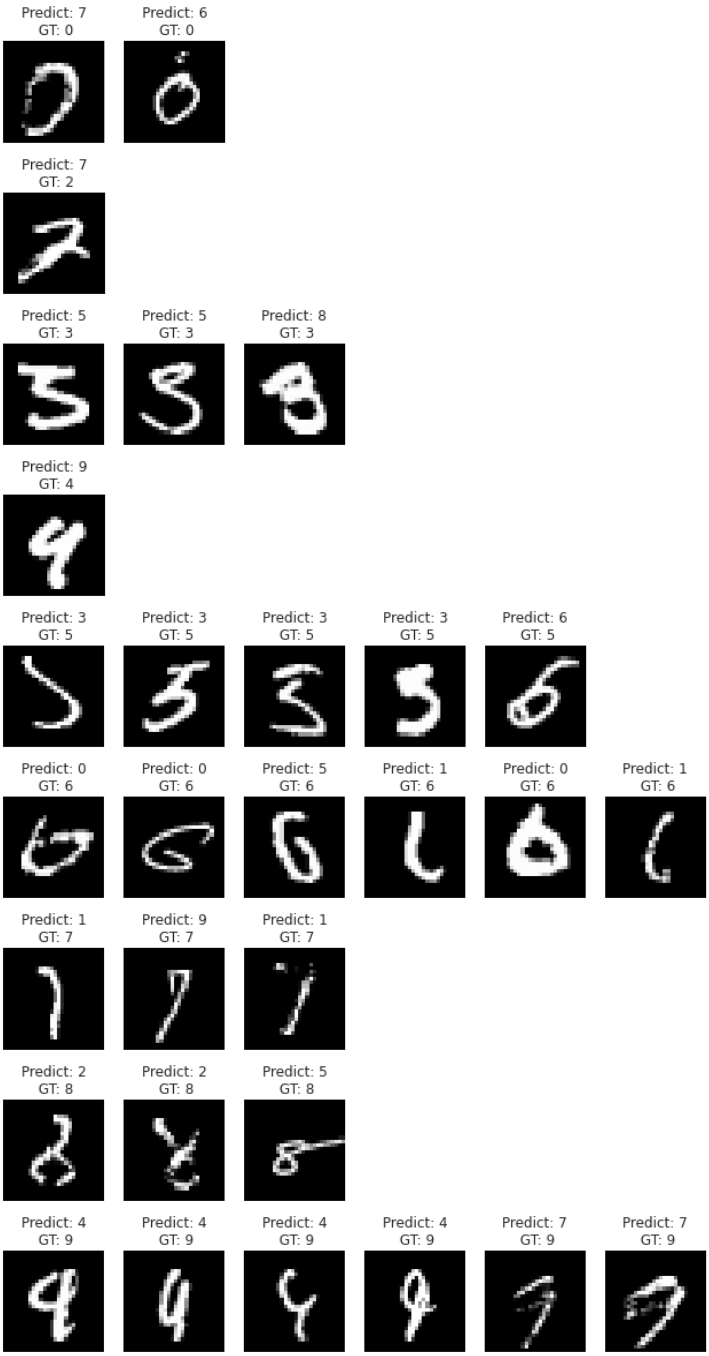


Figure 24: Misclassified examples in MNIST testset on Final Model

11 Conclusion

In this course project report, we gave a brief introduction on the MNIST dataset and performed EDA to understand the dataset better. We found out that the dataset is slightly imbalanced with lesser instances of digit "5" than digit "1".

We then described our baseline model, LeNet5 [6], which is a well-known architecture for the MNIST dataset. To ensure fair comparison, we also implemented a modified version of LeNet5 (M-LeNet5) by replacing average pooling with max pooling and Sigmoid activation functions with ReLU functions. M-LeNet5 is shown to have a great improvement over LeNet5 in terms of test accuracy.

Next, we illustrated our proposed model, AJCNN, short for AdjustableCNN. This model's layers and feature map sizes can be easily configured using a configuration dictionary. The motivation behind this design is to create a tune-able model that is not too deep, which tends to overfit and contains sufficient layers to be able to generate good feature representation and matching for small-sized MNIST images. By using a set of default configurations, AJCNN is able to outperform the two baseline models, LeNet5 and M-LeNet5.

After we selected our best model variant, AJCNN8, we proceed to experiment with the various optimizers like SGD, RMSProp and Adam to train the model. We also explored three different schedulers, namely StepLR, ReduceLROnPlateu and CosineAnnealingLR. Furthermore, we trained our models using Cross Entropy Loss and its variant with Label Smoothing applied. After numerous rounds of cross validations by splitting our train set to 80% train and 20% validation, we found that using SGD optimizer with an initial learning rate of 0.01 adjusted by StepLR scheduler and Label Smoothing Cross Entropy Loss on AJCNN8 is able to obtain an accuracy of 99.68% on the MNIST test set.

To improve our model, we analyzed the state-of-the-art model, BMCNN [1], which has a single-model score of 99.79%, to learn from their model features and training ideas. Furthermore, we also performed a thorough error analysis on the incorrect predictions made by our model through visualization. We realized that our model needs to perform better on more difficult and ambiguous samples. Hence, we decided to explore the various data augmentation strategies as introduced by the authors of BMCNN. We experimented with various augmentations methods such as Random Scale and Random Rotation and conclude that Random Rotation with a range between -10° and 10° is the best and most suitable data augmentation technique since it is able to output the highest test accuracy. Hence for our final model as discussed in Section 10, by increasing the label smoothing factor from 0.001 to 0.003 with Random Rotation has improved our test accuracy from 99.68% to 99.70%.

Appendices

A LeNet5 Model

Listing 1: LeNet5 Implementation

```
class LeNet5(nn.Module):
    def __init__(self, kernel=5, pad=2,
                 activation='sigmoid', pool='avg',
                 num_filter1=6, num_filter2=16, linear1=400):
        super(LeNet5, self).__init__()
        self.name = 'LeNet5'
        self.conv1 = nn.Conv2d(1,   num_filter1,  kernel_size=kernel, padding=pad)
        self.conv2 = nn.Conv2d(num_filter1,   num_filter2,  kernel_size=kernel)
        self.linear1 = nn.Linear(linear1, 120)
        self.linear2 = nn.Linear(120, 84)
        self.linear3 = nn.Linear(84, 10)
        self.dropout = nn.Dropout(0.001)

        if activation == 'sigmoid':
            self.activation = nn.Sigmoid()
        elif activation == 'relu':
            self.activation = nn.ReLU()
        if pool == 'avg':
            self.pool = nn.AvgPool2d(2, 2)
        elif pool == 'max':
            self.pool = nn.MaxPool2d(2,2)

    def forward(self, x):
        x = self.conv1(x)
        x = self.activation(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.activation(x)
        x = self.pool(x)
        x = x.view(-1, x.shape[1]*x.shape[2]*x.shape[3])
        x = self.linear1(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.linear2(x)
        x = F.relu(x)
        x = self.dropout(x)
        x = self.linear3(x)

        return x
```

B AJCNN Model

Listing 2: AJCNN Implementation

```
AJCNN_cfgs = {
  "AJCNN4": [[32, 32, 'M', 64, 64, 'M'],['D', 512, 'D', 512, 'D']],
  "AJCNN6": [[64, 64, 128, 'M', 128, 192, 'M', 192, 'M'],['D', 512, 'D', 256, 'D']],
  "AJCNN8": [[32, 32, 64, 'M', 64, 128, 128, 'M', 192, 192, 'M'],['D', 128, 'D', 128,
  ↪  'D']],
  "AJCNN10": [[32, 32, 64, 'M', 64, 128, 128, 'M', 192, 192, 256, 'M', 256],['D', 128,
  ↪  'D', 128, 'D']]

}

def make_conv_layers(cfg, in_channels, batch_norm=True):
    layers = []
    in_channels = in_channels
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        elif type(v) == int:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)

def make_linear_layers(cfg, in_feats, out_feats, batch_norm=True):
    layers = []
    in_feats = in_feats
    for v in cfg:
        if v == 'D':
            layers += [nn.Dropout(p=0.5)]
        elif type(v) == int:
            linear = nn.Linear(in_feats, v)
            if batch_norm:
                layers += [linear, nn.BatchNorm1d(v), nn.ReLU(inplace=True)]
            else:
                layers += [linear, nn.ReLU(inplace=True)]
            in_feats = v
    layers += [nn.Linear(in_feats, out_feats)]
    return nn.Sequential(*layers)
```

```
class AJCNN(nn.Module):
    """
    Our proposed model: Adjustable CNN

    Creates an adjustable CONV feature block from first array in configuration.
    Creates an adjustable Linear classifier block from second array in configuration.

    Automatically calculate flattened output from CONV feature block.

    CONV feature block appends convolutional layers in a configurable manner.
      If numerical: Adds a Conv2D layer, followed by optional BatchNorm2d, and ReLU
      If 'M': Adds a MaxPooling layer

    Linear classifer block appends linear layers in the following configurable manner:
      If numerical: Adds a linear layer, followed by optional BatchNom1d, and ReLU
      If 'D': Adds a Dropout layer with p=0.5
    """
    def __init__(self, in_channels: int = 1,
        n_classes: int = 10,
        input_dim = (1,28,28),
        variant:str = 'AJCNN8'):
        super(AJCNN, self).__init__()
        self.name = variant
        self.features = make_conv_layers(AJCNN_cfgs[variant][0], in_channels)
        num_feats_aft_conv = functools.reduce(operator.mul,
        ↪  list(self.features(torch.rand(1, *input_dim)).shape))
        self.classifier = make_linear_layers(AJCNN_cfgs[variant][1], num_feats_aft_conv,
        ↪  n_classes)
        self._initialize_weights()

    def _initialize_weights(self):
        for m in self.features.children():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
                if m.bias is not None:
                   nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

        for m in self.classifier.children():
            if isinstance(m, nn.Linear):
                nn.init.xavier_uniform_(m.weight)
            elif isinstance(m, nn.BatchNorm1d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)

        return x
```

C Label Smoothing Cross Entropy Loss

Listing 3: Label Smoothing Cross Entropy Implementation

```
class SmoothCrossEntropyLoss(nn.Module):
    def __init__(self, smoothing=0.0):
        super(SmoothCrossEntropyLoss, self).__init__()
        self.smoothing = smoothing

    def forward(self, input, target):
        log_prob = F.log_softmax(input, dim=-1)
        weight = input.new_ones(input.size()) * \
            self.smoothing / (input.size(-1) - 1.)
        weight.scatter_(-1, target.unsqueeze(-1), (1. - self.smoothing))
        loss = (-weight * log_prob).sum(dim=-1).mean()
        return loss
```

D Data Augmentation

Listing 4: Data Augmentations Implementation

```
def data_augmentations(augments=None):
    normalize = transforms.Normalize((0.1307,), (0.3081,))
    data_transforms = {
        'train': transforms.Compose([
            transforms.ToTensor(),
            normalize
        ]),
        'valid': transforms.Compose([
            transforms.ToTensor(),
            normalize
        ]),
        'test': transforms.Compose([
            transforms.ToTensor(),
            normalize
        ])
    }
    if augments is not None:
        all_augs = [transforms.ToTensor(), normalize]

        if 'RandomRotation' in augments:
            all_augs.append(transforms.RandomRotation(degrees=10,
                                                      resample=Image.BILINEAR))

        if 'RandomPerspective' in augments:
            all_augs.append(transforms.RandomPerspective(distortion_scale=0.2, p=0.5,
                                                         interpolation=Image.BILINEAR))

        if 'RandomResizedCrop' in augments:
            all_augs.append(transforms.RandomResizedCrop(size=(28,28)))

        if 'RandomErasing' in augments: # only random erasing is after to tensor
            all_augs.append(transforms.RandomErasing(p=0.3, scale=(0.02, 0.33),
                                                     ratio=(0.3, 3.3), value=0))
        if 'RandomCropAndResize' in augments:
            all_augs.append(transforms.RandomApply(torch.nn.ModuleList(
                [ transforms.RandomCrop(24), transforms.Resize(28)]), p=0.5))

        if 'RandomScale' in augments:
            all_augs.append(torchvision.transforms.RandomAffine(
            ↪  transforms.RandomAffine(degrees=0, scale=(0.9, 1.1))))

        if 'RandomPerspective' in augments:
            all_augs.append(transforms.RandomPerspective(0.3))

        if 'CentreCropandResize' in augments:
             all_augs.append(transforms.Compose([
            transforms.RandomApply(torch.nn.ModuleList([
            ↪  transforms.CenterCrop(24),transforms.Resize(28)]),p=0.3)]))

        data_transforms['train'] = transforms.Compose(all_augs)

    return data_transforms
```

References

[1] Adam Byerly, Tatiana Kalganova **and** Ian Dear. "A Branching and Merging Convolutional Network with Homogeneous Filter Capsules". **in**: 2020. eprint: 2001.09136.

[2] Kaiming He **andothers**. "Deep residual learning for image recognition". **in**: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, **pages** 770–778.

[3] Geoffrey E. Hinton. *What's wrong with convolutional nets?* MIT Tech TV. 2018. URL: URL: https://techtv.mit.edu/collections/bcs/videos/30698-what-s-wrong-with-convolutional-nets.

[4] Sergey Ioffe **and** Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". **in**: *arXiv preprint arXiv:1502.03167* (2015).

[5] Diederik P Kingma **and** Jimmy Ba. "Adam: A method for stochastic optimization". **in**: *arXiv preprint arXiv:1412.6980* (2014).

[6] Yann LeCun **andothers**. "LeNet-5, convolutional neural networks". **in**: *URL: http://yann. lecun. com/exdb/lenet* 20.5 (2015), **page** 14.

[7] Yann LeCun **andothers**. "Gradient-based learning applied to document recognition". **in**: *Proceedings of the IEEE* 86.11 (1998), **pages** 2278–2324.

[8] Ilya Loshchilov **and** Frank Hutter. "Sgdr: Stochastic gradient descent with warm restarts". **in**: *arXiv preprint arXiv:1608.03983* (2016).

[9] Rafael Müller, Simon Kornblith **and** Geoffrey E Hinton. "When does label smoothing help?" **in**: *Advances in Neural Information Processing Systems*. 2019, **pages** 4694–4703.

[10] Nitish Srivastava **andothers**. "Dropout: a simple way to prevent neural networks from overfitting". **in**: *The journal of machine learning research* 15.1 (2014), **pages** 1929–1958.