Digits and Characters Recognition with Convolutional Neural Networks

COMP5329 Deep Learning - Assignment 2

Quang Trung NGUYEN *470518197,* ***Zhehao ZHANG*** *470490653*
Youyou CHEN *308012798*

Abstract *Text recognition is a challenging problem that has received much attention in recent years. Convolution Neural Network (CNN) is a prevalent technique in addressing image related classification problems. This study applied three different CNN architectures to complete a 'digits and character classification' task: modified LeNet, ZFNet and VGG. The models were trained and evaluated according to their calculation cost, model complexity and prediction accuracy on a separate validation dataset. Each model is tuned to achieve its optimal state. VGG model with transfer learning generated the highest validation accuracy of 92.37%, followed by ZFNet (91.06%) and LeNet (88.45%). Data augmentation has also been employed in the final VGG model. It indicated that the VGG architecture is the optimal model to be applied in this study due to its relevantly short running time due to the application of transfer learning, relevantly high validation set and relevantly low validation loss.*

1. Introduction

As one of the most advanced tools in machine learning field, Convolutional Neural Network (CNN) architectures have been used and continue to yield great success in completing complex computer vision tasks like text and image recognition, video analysis, etc. (Sinha, 1999; Tay et al, 2001). Although using graphics processing unit in recent years has greatly speeded up training of conventional fully connected but deeper networks, CNN-based approaches take the advantage of local spatial coherence in the input (such as images), which allows them to have fewer weights than plain artificial neural networks as some parameters are shared. This process, taking the form of convolutions, makes them especially well-suited to extract relevant information in images at a relatively lower computation cost (LeCun et al, 1998).

In this study, we attempted to complete a digits and characters recognition task using CNN. We firstly introduced some basic information regarding the dataset at hand in Section 1.1.
We then continued to describe a generic CNN architecture and its internal layer operations in Section 2.1. We then investigated some related work and several established benchmark CNN architectures (e.g. *LeNet, ZFNet, VGG* etc.) and their applications in real life cases in Section 2.2. In Section 3, we experimented the implementations and optimization of few models (inspired by the benchmark models described earlier) on our dataset. The performances of different architectures were also compared and summarized in terms of accuracy, computation cost, etc. The details for our design choice model are then further discussed with comparisons to other models applied in Section 4 with the final conclusion in Section 5. Instructions to running the code are also provided in Appendix 1 of this report.

1.1 Dataset Information

The dataset provided for this project is a labeled binary (black and white) image dataset which contains 6262 test samples and 37882 training samples of English alphabet and Arabic numerals in different fonts. There are a total of 62 labels or classes, including 26 letters, each with an uppercase (A-Z) and a lowercase (a-z) form and 10 numeric digits (0-9). Both of the testing and training set has a balanced distribution with 101 instances and 611 instances in each class respectively. Each image is in 128 pixels by 128 pixels. Each pixel is represented by the value of 0 or 255 with 0 being black and 255 being white.

1.2 Approach Overview

Our first goal was to follow the similar architecture *LeNet* used for training and evaluating the implied dataset. The simple structure established a baseline for comparison with other more complicated architectures we implemented later in this study. Next, we chose to try a deeper *ZFNet* which consists of five convolutional layers, some of which are followed by max-pooling layers, and three fully connected layers with a final *softmax* layer. VGG architecture was also tested on our dataset in this study where eight more convolutional layers were incorporated in comparing with *ZFNet*.

Due to the data dimensions and complexity variations associated with different datasets, the hyper-parameters and other training details used for the benchmark models in previous studies were likely to be not optimal for our dataset in this study. So in the attempt to optimize model performance for this project specifically, we tuned the filter numbers and sizes, learning rate, momentum, dropout percentages, data augmentation, batch normalization on different layers of each of these CNN models. Finally, we came to the conclusion that the optimal model to be applied in this study followed the *VGG* architecture with certain degree of variation. Details of this model will be introduced in Section 3 & 4.

2. Convolutional Neural Network

While the normal multilayer neural network performs well for simpler classification tasks, it has several disadvantages. First, the number of trainable parameters becomes extremely large for high dimensional inputs because each layer is fully connected to its adjacent layers. Secondly, it offers little or no invariance to shifting, scaling, and other forms of distortion (LeCun et al, 1998). Third, the geometrical properties and spatial relations within each input sample is completely omitted, yielding similar training results even when numbers within the input vector is rearranged. To overcome these drawbacks, a hand-crafted feature extractor should be put in place which is often a heuristic procedure (LeCun et al, 1998).

2.1. Basic structure

CNN is a special multilayer neural network with a supervised deep learning structure with two basic components: an automatic feature extractor and a trainable classifier (Niu & Suen, 2012). Although there are many different CNN architectures in the literature, the basic components in all designs are very similar whereas the feature extractor contains feature map and retrieves discriminating features from the raw images via two operations: convolutional filtering and down sampling.

Convolutional Layer
Convolutional layer is considered to be the core building block of a CNN architecture which focuses on the exploitation of local substructure within the input such as images because pixels that are closer together in the image (e.g., adjacent pixels) tend to have stronger correlation than pixels that are farther apart within the same image (Lowe, 2003). The convolutional filtering kernels on feature maps have the size of n by n pixels and neurons that belong to the same feature map share the same weights (also called filters or kernels) in that layer. Thus in CNN architectures, substructures are captured by constraining each neuron to depend only on a local receptive field rather than a global receptive field in the previous layer.

Pooling Layer
It is common to periodically apply a pooling layer between successive convolutional layers in CNN architectures (Bui & Chang, 2016). Down sampling operations are often performed to progressively reduce the dimension of spatial resolution of the feature map as well as to reduce the amount of parameters and computation in the network, and hence to also control over-fitting (Lee et al, 2015). The pooling layer operates independently on every depth slice of the input and resizes it spatially. A common form is a max pooling layer with filters of size 2x2 applied with a stride of 2 in both width and height, discarding 75% of the activations by taking the maximum of 4 numbers (in the 2x2 region in each depth slice) so that depth dimension remains unchanged after pooling operations. It is worth noting that pooling sizes with larger receptive fields can be too destructive (Bui & Chang, 2016).

Ultimately, these layers were usually pursued by fully connected layers and output layer. Like normal feedforward neural network operations, the output of the previous layer is taken by the next layer as the input. The classifier and the weights learned in the feature extractor are trained by a back-propagation algorithm.

2.2 Related Work

In this section, we will summarize some important developments and benchmark models developed in the field of computer vision and convolutional neural network in recent years.

LeNet or *LeNet5* is a pioneering CNN architecture developed by LeCun et al in 1998 to recognize isolated hand-written digits (0-9) in 32x32 pixel images. It produced state-of-art performance at the time. This architecture is of two convolutional layers, two pooling layers, two fully connected layers and one output layer deep, meant to recognize digits. However, the ability to process higher resolution images usually requires larger and more convolutional layers, so this technique is somewhat constrained by the availability of computing resources.

In the 2012 ILSVRC (ImageNet Large-Scale Visual Recognition Challenge), *AlexNet* significantly outperformed over other competitors and won the classification challenge with 1000 possible categories by reducing the top-5 error from 26% to 15.3%. The network had a very similar architecture as *LeNet* but was deeper, with more filters per layer, and with stacked convolutional layers. It consisted of 5 convolutional layers with different sizes (11x11, 5x5, 3x3), max pooling, dropout, data augmentation, *ReLU* activations, SGD with momentum. It attached *ReLU* activations after every convolutional and fully-connected layer. In the following year of ILSVRC, an improved hyper-parameters model over *AlexNet* called the *ZFNet* further decreased the error rate by around 5%.

For ILSVRC14, *GoogLeNet*, a 22 layers deep network, used a new variant of convolutional neural network called "Inception" for classification (Szegedy et al, 2014). It trained a much smaller neural network with no fully connected layers (except for output layer) and obtained significant better results (6.7% top 5 error rate) compared to results obtained with convolutional neural networks in the previous year's challenges. It designed a good local network topology (network within a network) and then stacked these modules on top of each other.

In 2015, the 152-layer model named *ResNet* achieved a 3.57% top 5 error rate in the same challenge. It also had stacked residual blocks like *GoogLeNet* where every residual block has two 3x3 convolutional layers. Periodically, double number of filters and spatially down-sampling stride 2x2 are applied. Additional convolutional layer is also added at the beginning. For deeper networks (*ResNet-50+*), similar to *GoogLeNet*, bottleneck layers are applied to improve efficiency.

3. Experiment and Results

The design of our experiment could be divided into three stages. Firstly, three architectures of CNN were used to fit the dataset at hand during the experiments (Section 3.2 to 3.4). These models were developed based on the basic designs of *LeNet, ZFNet* and *VGG*, respectively, with different degrees of variation. The reason we designed our experiment to adopt these established and generative models as our initial or base models is to save computation cost and improve productivity for us comparing with building and testing a model from scratch based on random guesses. Then, it discussed how each model is tailored to the task for this study specifically. In stage two of the experiment, that is, during model optimization and parameter tuning stage, to further lower the experiment frequency and shorten experiment time, we used the following methods:

(1) Trial-and-error method: to select experiment parameters based on instinct with minimal data analysis. This is not a systematic approach however as Baker et al (2017) suggested, it is indeed an essential and inevitable step during the development of a neural network structure due to the network's intuitive nature;
(2) One-factor-at-a-time method: to change only one hyper-parameter in each experiment and other factors maintaining at the original level until a satisfying condition appears. This fine-tuning experimental treatment could to some extend helps us identify the impact of each parameter to the experiment result comparing with other factors (Berger & Maurer, 2002).

We designed a similar tuning process with certain variations in process for each model which will be tabled and explained in details in Section 3.2 -3.4.

Stage three involved the choice of final architecture. It applied some further fine-tuning techniques to improve the performance of the optimal model.

3.1 Preparation

All the images are read by *misc.imread()* module of *scipy* package in Python, and thus converted into channel format, to be specific, 'width x height x channels' format. Different reading modes could lead to different array format of the output. For example, in cases

where we read the data in binary scheme (*mode = 'L'*), the depth slice or channel for the input data equals to 1. Hence each sample image can be then depicted by an array of 128x128 or a flattened vector of 16,384 dimensions whereas in three RGB channel scenario (*mode = 'RGB'*), each sample is depicted by three arrays of 128x128 pixels.

It should be noted that different extracting methods were applied during different models constructing. For our experiment, only 1 channel data, grey scale data, is used by *LeNet* and *ZFNet*, while three RGB channels data is required when constructing the *VGG* model for the purpose of applying transferring learning technique into the model.

All the models are implemented using *keras 2.1.6* with Tensorflow backend, which can be run in python 3 environments. Since image processing and the convolution neural network require tremendous computational power to run, it is therefore essential to apply the GPU to speed up computation and save running time. Hence, the Google Colaborary is chosen and utilised for improving speeding up model training efficient with the help of the GPU. The model constructing job could have done without GPU, but it will take much longer time.

The preprocessing technique applied in this study involves augmenting the existing data-set with perturbed versions of the existing images. Scaling, rotations and other affine transformations are typical. This is done to expose the neural network to a wide variety of variations. This should, in theory, make it less likely for the neural network to recognize unwanted characteristics in the dataset. Resizing of the original sample is also applied in LeNet which will be introduced in Section 3.2 below.

3. 2 Modified *LeNet*

Variants of the *LeNet* design are prevalent in the image classification literature and have yielded good results on datasets with 32x32 pixel samples such as MNIST and CIFAR (He et al, 2016). The first model we built for this study followed this structure to take the advantage of its efficiency and simplicity so that a baseline could be established for further experiments.

3.2.1 Architecture Overview

Sparse, convolutional layers and max-pooling are at the heart of the LeNet family models. In the original model (LeCun et al, 1988), the input of 32x32x1 samples are feed to a convolutional layer (C1) with 6 5x5 size filters. A max pooling layer (S2) with filter of 2x2 grid and 2x2 stride is then applied. A convolutional layer (C3) is then followed with 5x5 size filters and depth size of 16, then following by another sub-sampling layer (S4) with same hyper-parameters with S2. It should be noted that each unit in C3 is only connected to several receptive fields with S2. The grid size of the output after S4 is 5x5 so a convolution layer (C5) of the same grid size filter of 120 in depth is applied to transfer the grid size to 1x1, hence the output becomes a flattened layer. The fully-connected layers followed which correspond to a traditional MLP with one hidden layer (F6) of size 64 and output layer of size 10.

3.2.2 Experiment Design and Techniques

For the dataset used in this study, the dimension of each sample is different to (higher than) the one used to develop the *LeNet* architecture. As elaborated earlier, convolution layers play the role of extracting the feature maps from the input layer by repeatedly applying a certain filter of a given size along with the height and width of the input image. Differences in dimension of initial input, filter sizes, number of filters could significantly influence model performance.

Because we are already aware of the fact that *LeNet* performs generally well in 32x32 pixel image classifications, we firstly attempted to resize the samples in our dataset to 32x32 too in order to established the baseline more quickly with minimum trial-and-error attempts.

In the attempt to further improve the original model, we tried to add a *ReLU* activation function after each convolutional layer and applied dropout for the percentage of 0.25 and batch normalization after each pooling and fully connected layer. We also replaced layer C5 with a simpler flatten layer while connections between S2 and C3 are set to be fully connected. We also scaled the number of nodes in F6 to 256 proportionally.

3.2.3 Resizing & Initial Set-ups

The first resize approach is through the use of a max pooling layer in CNN. In *keras*, a *max pooling layer* with 4x4 filer size and 4x4 stride simply takes the maximum value of the 16 values within that grid and reduces the size of the input by 16 times. Such max pooling layer is applied to the CNN before any convolutional layer or fully connected layer, the dimension of the sample is reduced to 32x32 before it was feed to the original *LeNet* architecture.

As discussed earlier, a larger stride in the pooling layer produces a small volume in the output, but also causes a larger degree of information loss in the process of down-sampling. Under this approach, the loss of information is relatively large (1:16) which could significantly deteriorate model performance. To overcome this issue, we attempted the second approach for resizing which is to use an existing resize library under *scipy.misc.imresize()* module.

We compared the performances of the two approaches by applying the same subsequent *LeNet* architecture as introduced earlier with different weight update back-propagation method. Under a one-factor-at-a-time experiment approach, because we merely want to establish a simple comparison between the two resizing approaches, only results from one optimization method ('*rmsprop*' using default hyper-parameters in *keras* library) were summarized here (Table 1). Other weight update methods such as '*sgd*' and '*adam*' are also tested and produced similar results. It could be observed that the model converged after around 10 to 15 epochs so we set number of epoch to 20. Each epoch only takes around 4-6 seconds to run on average which makes it a very efficient model with relatively low computation cost. Cross validation is not applied in this case. We simply used model performance on the validation set as an approximate proxy to evaluate to save computation cost. Same applies to most other experiments in this study for grid search.

Table 1. Performance of Different Resize Approaches using LeNet (with variations)

	Validation Accuracy	**Validation Loss**	**Total Run-times (mins)**
Resize through conv layer	85.12%	45.08%	1.78
Resize through scipy library	**88.15%**	**32.29%**	1.45

One possible reason the second resize approach produced a better validation accuracy could be for the fact that the algorithm behind *scipy.misc.imresize()* module applied scale transformation and interpolation to the original image. According *scipy.misc.imresize()* documentation, it uses inverse mapping to determine where each point in the output grid maps in the input grid. It then interpolates within the input grid to determine the output grid

values. By default, *imresize* uses bi-cubic interpolation. That is, it applies a scale geometric transformation to the original samples. For these reasons, the second resize approach with lower degree of information loss thus performed better in this experiment than the first and sparser resize approach. We therefore used the more favorable second resize approach for further experiments next.

3.2.4 Parameters Fine-tuning & Experiment Results

In this subsection, the results of three parameters tuning experiments will be extensively discussed. We adopted a one-factor-at-a-time experiment approach for tuning of different optimizers, learning rate batch size and a trail-and-error approach for tuning of different filter sizes, depth, activation functions to further explore the possibility of further improvement on the existing model. Similar experiment processes described in this subsection will be adopted for later models introduced in Section 3.3 and 3.4 as well.

Firstly, as developed in subsection 3.2.3, the model we tried to adopt modified LeNet architecture with additional *ReLU* activation function, dropout and batch normalization. We firstly want to investigate the effect of each addition on the original model. This explorative experiment is only applied for architecture used in this section (not in deeper models introduced in Section 3.3 and Section 3.4) because the fast run-time nature of this underlying simple *LeNet* structure. In table 2 below, we test 8 scenarios where different combinations of these additions were put in place where a tick (√) indicates the relevant optimization method was incorporated and a cross (x) indicates otherwise. Last column of the table showed the validation accuracies after 20 epochs for each combination with the default '*rmsprop*' backpropagation hyper-parameters and batch size of 128 in *keras*.

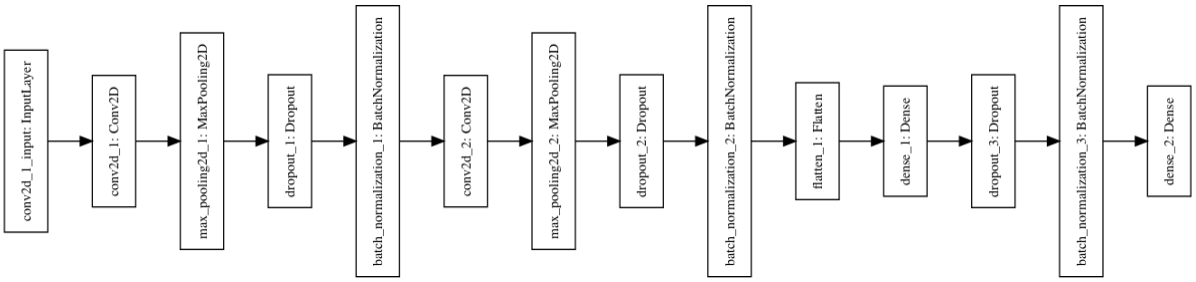
Table 2. Performance of Different Combinations of Additional Optimization Techniques

Scenario Index	ReLU Activation	Dropout at 25%	Batch Normalization	Validation Accuracy
1	x	x	x	32.40%
2	√	x	x	84.97%
3	x	√	x	31.76%
4	x	x	√	**88.18%**
5	√	√	x	1.61%
6	√	x	√	**88.45%**
7	x	√	√	86.12%
8	√	√	√	**88.15%**

It could be summarized from above that adding batch normalization (see four highlighted rows) is essential for the application of *LeNet*-like architectures in this study and could significantly increase the stability of the model by normalizing the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation. Hence, it reduces the amount by what the hidden unit values shift around (covariance shift) (Ioffe & Szegedy, 2015) and accelerates training (He et al, 2016). The variation in validation accuracy of the four highlighted combinations is not significant, which could be caused by different initialization values etc., thus we continued our experiments by using all three optimization techniques (scenario 8). There are many other different scenarios we could explore such as applying different optimization techniques for different layers, but due the

focus and scope of this study, we will not go further in this aspect. The model architecture used for further optimization is plotted as Figure 1.

Figure 1. Architecture of Modified LeNet Model



Next, we conducted experiments on applying different parameters tuning such as varying learning rates (lr), optimizers and batch sizes. We firstly fixed batch size to 128 and varying learning rates for both '*sgd*' and '*adam*'. Under 'sgd' approach, we set *nesterov momentum* to be 0.9 with decay. Under '*adam*' approach, we set $\beta 1$ and $\beta 2$ to be 0.9 and 0.999 respectively with decay and no *amsgrad*. Early stopping technique is also applied here for saving training time. Specifically, the training process stops if there is no obvious increase in validation scores for some amount of time (Goodfellow, Bengio & Courville, 2016), and then the model stores the optimal parameters combinations. Small learning rates have larger number of epochs during training to reach for convergences. Results are summarized in Table 3 below.

Table 3. Performance of Different Hyper-Parameters Tuning Approaches Under Modified LeNet

		Validation Accuracy	Validation Loss	No of Epochs	Total Run-times (mins)
SGD	**lr=0.0001**	77.29%	88.42%	40	2.53
SGD	**lr=0.001**	85.13%	46.29%	35	2.23
SGD	**lr=0.01**	87.80%	33.64%	20	1.32
Adam	**lr=0.0001**	86.59%	40.53%	40	3.03
Adam	**lr=0.001**	**89.03%**	**28.83%**	35	2.88
Adam	**lr=0.01**	87.21%	33.14%	20	1.43

The experiment is a result-driven process. The optimal solution is sought from validation accuracy, validation loss, total run time and how epochs it needs perspectives. It could be seen that learning rate of 0.0001 for '*sgd*' is too slow for model to reach convergence during training while the process is faster under same learning rate using '*adam*'. Under '*sgd*', 0.01 is the better learning rate where under '*adam*', 0.001 is. The latter optimizer performs slightly better in this study (highlighted row) so we will use this combination for finding the optimal batch size.

For finding optimal batch size, we used simple grid search of 64, 128 and 256. Results are shown in Table 4. It could be shown that different batch sizes did not lead to material differences in validation accuracy or loss but indeed made some differences in training time. We chose batch size of 256 in this case (highlighted row) to save running time.

Table 4. Performance of Different Batch Size Under Modified LeNet

	Validation Accuracy	Validation Loss	No of Epochs	Total Run-times (mins)
Batch size = 64	88.01%	30.39%	35	5.68
Batch size = 128	88.73%	30.38%	35	2.97
Batch size = 256	**88.63%**	**29.16%**	35	1.77

To summarize, it could be concluded that a modified *LeNet* architecture (See Figure 1 for details) with the '*adam*' back-propagation in weight updates with the learning rate of 0.001 and batch size of 256 is the optimal model in this section. It could also be noted that this modified *LeNet* model generalized well under this study, achieving an accuracy rate of approximately 88% in the validation set.

3.3 *ZFNet*

ZFNet, created by Zeiler and Fergus (2013), and was adopted as the second model to address this classification problem. Essentially, *ZFNet* is a variation of *AlexNet* which was introduced by Krizhevsky et al (2012). Thus, the architectures of *ZFNet* and *AlexNet* are quite similar but with minor modifications in *ZFNet,* which has some significant improvements on performance comparing with *AlexNet*. In addition to develop new ideas to improve model performance, visualising the features maps is another great improvement within the *ZFNet* framework (Zeiler & Fergus, 2013). The architecture of *ZFNet* is described in details next.

3.3.1 Architecture Overview

As introduced earlier, *ZFNet* consists of total 8 layers, including 5 convolution layers and 3 fully connected layers. One difference of *ZFNet* from the previous model in Section 3.2 is the addition of zero padding which in many cases reduce noise effect in image classification tasks (Bacha et al 2015). Padding is related to how pad the input picture by 'zeros' in *keras* package. There are "*valid*" and "*same*" padding where the former means no padding added to the picture (default setting) and the latter stands for padding the picture as many zeros as needed in order to ensure the output size to be the same as the input size.

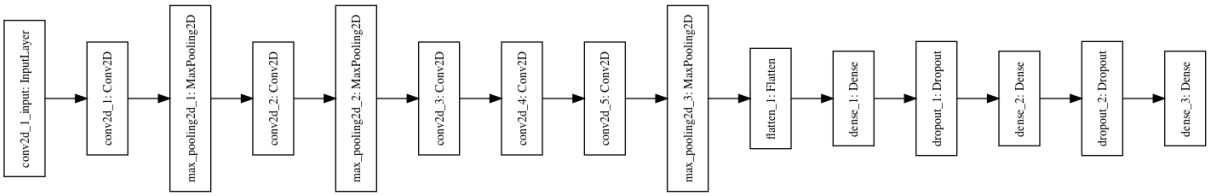
The *ReLU* activation function is utilised for every convolutional layers in this *ZFNet* for this case, which can reduce the likelihood to vanish the gradient. The number of filters for each layer in *ZFNet* is 96, 254, 384, 384 and 256 respectively. The filter size in the first convolution is 7x7 instead of 11x11 (which is the *AlexNet* parameters setting), which is the main improvement. The idea behind of decreasing filter size is in order to keep more original pixel information when processing the input pictures (Deshpande 2018). The filter size of 11x11 has been identified to be missing some useful information. Thus, this modification has been proved that will lead to better performance (Deshpande 2018). Another improvement is introducing visualizing features maps, but it was not used in this project (Zeiler & Fergus, 2013). The discussion for visualizing features maps is therefore omitted.

3.3.2 Experiment Design and Techniques

First of all, the overall architecture of *ZFNet* can be seen in Figure 2, including 5 convolution layers and 3 fully connected layers. The number of filters is 96, 254, 384, 384 and 256 respectively. The filter size for the first convolution layer is 7x7, followed by 5x5 filter size in the second convolution layer, the rest of them keep same filter size as 3x3. In order to keep output size is same as input size and prevent the output dimension becomes too small, the

'*same*' padding technique is applied for every convolution layer expect the first convolution layer. Next, flatten layer is added before the fully connected layer for to preserving weight ordering (Zeiler & Fergus, 2013).

Figure 2. Architecture of ZFNET Model



On top of that, dropouts are added after the each fully connected layer expects the last layer for keeping away from over fitting issue. It should be noted that there is an alternative technique, global average pooling (Lin, Chen & Yan, 2013), can be utilized for replacement of the flatten layer and dropouts for reducing the number of parameters and increasing model performance. Nevertheless, global average pooling is not employed in the end because the performance of model is not improved when it is applied into the model. In other words, the team made attempts to employ various techniques for improving performance, which is a result-driven process. Thus, the techniques that are unable to improve results will be not used in the final model. In the last fully connected layer, a *softmax* function is used for multi-class classification problem.

Overall, this ZF net produces 30,193,662 trainable parameters in this case (Appendix 4), which is significant less than the number of parameters produced by Alex net, 256,695,442 (Appendix 5) that needs to train. Therefore, the computational cost is drastically reduced because of less trainable parameters. In general, 5 to 10 epochs are reasonable waiting time for a deep learning model. Therefore, 8 epochs method has been chosen as early stopping setting in *ZFNet* for this case.

3.3.3 Parameters Fine-tuning & Experiment Results

In this section, the results of three parameters tuning experiments under *ZFNet* will be extensively discussed. Results from applying different learning, optimizers and batch size could be seen in Table 5 and Table 6.

Table 5. Performance of Different Hyper-Parameters Tuning Approaches Under ZFNet

		Validation Accuracy	Validation Loss	No of Epochs	Total Run-times (mins)
SGD	**lr=0.0001**	89.38%	31.63%	40	36.02
SGD	**lr=0.001**	**91.54%**	**26.52%**	35	31.28
SGD	**lr=0.01**	1.61%	1585.81%	9	8.07
Adam	**lr=0.0001**	**91.06%**	**27.16%**	20	20.42
Adam	**lr=0.001**	1.61%	1585.81%	9	8.85
Adam	**lr=0.01**	1.61%	1585.81%	9	9

As described in earlier sections, different the number of epochs is applied through the use of early stopping technique in the Table 5. It can be seen that the ZF net with two optimizers are able to achieve very close accuracy if right learning rate chosen, with 91.54% and 91.06 respectively. The appropriate learning rate for '*adam*' is smaller than that for '*sgd'*. Because the criterion of early stopping is same when the experiments are conducted. Various the number of epochs stand for how fast the model can be converged. Additionally, the corresponding time is to measure the similar thing. In the other words, the less epochs need, the better results. According to results between *ZFNet* with ''*sgd'* and 'adam' optimizer. The model with Adam optimizer can be converged faster than that with '*sgd'* optimizer, although the validation accuracy is slightly lower and validation loss is higher. Taking into running time and computational cost, the ZF net with Adam optimizer is more cost effective. It is not worth spending 10 more minutes and 15 more epochs to gain the extra 0.5% validation accuracy increases. Therefore, *ZFNet* along with '*adam*' optimizer is the best model can be achieved at this stage. Next, the batch size experiment results will be discussed.

In same methodology as described in subsection 3.2.3, the comparisons focus on validation loss and accuracy, epochs and time perspectives. The batch size tuning results can be seen in table 6.

Table 6. Performance of Different Batch Size Under Modified ZFNet

	Validation Accuracy	**Validation Loss**	**No of Epochs**	**Total Run-times (mins)**
Batch size = 64	90.99%	29.34%	18	23.13
Batch size = 128	91.06%	**26.52%**	20	20.42
Batch size = 256	**91.11%**	28.47%	22	19.97

Basically, the validation accuracy is close and similar, but validation loss is higher when batch size is 256 and 64, which is not expected. Therefore, 128 dominates amongst them and the running time is feasible as well.

To summarize, the *ZFNet* along with '*adam'* optimizer function, learning rate of 0.0001 batch size of 128 is the optimal combinations in the architecture tested in this section.

3.4 VGG

VGG is a newly developed convolutional neural network (CNN) architecture published in 2014 by Visual Geometry Group that won them the top prizes in the ImageNet Challenge that year (Simonyan & Zisserman, 2015). Our third model tested in this study followed a VGG architecture with certain variations from the original model.

3.4.1 Architecture Overview

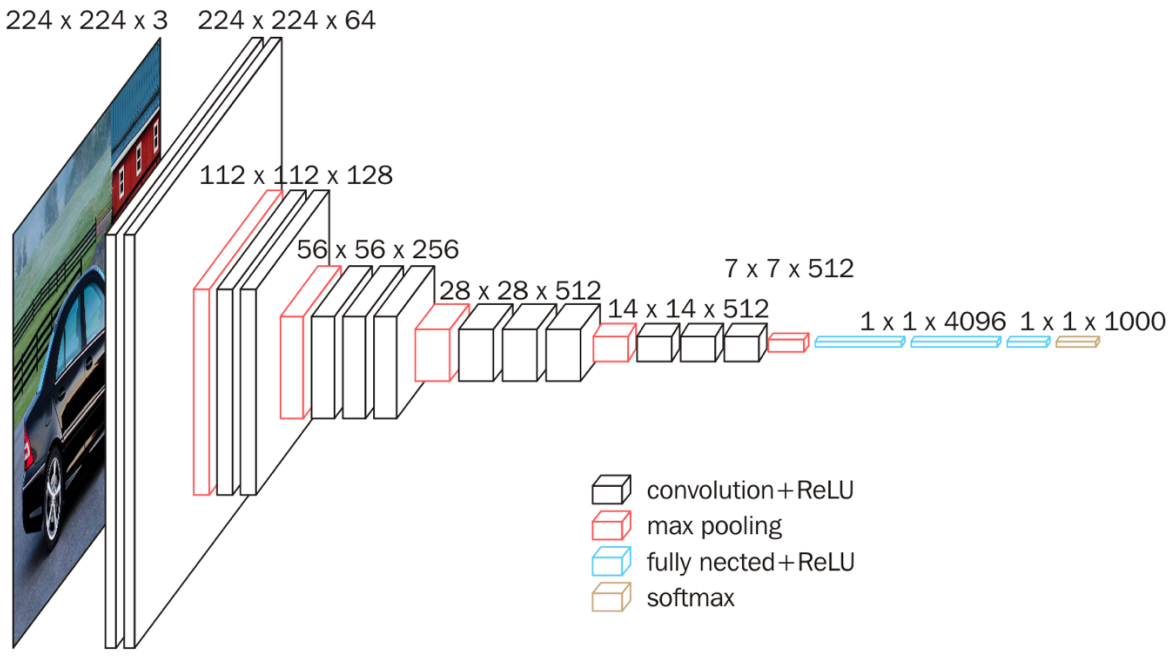
VGG's architecture aims to expand the depth of the traditional CNN architecture – *ConvNets*, developed by Krizhevsky, winner of ImageNet Challenge in 2012, hence it is often referred as 'deep' CNN. The architecture of *VGG* contains six different nets (A-E) that vary in depth from 16 to 19 layers which can be found in Appendix 6. Our study focuses on *VGG16*, a 16-layered CNN, which was one of their two most successful models; the other one is *VGG19* – a 19 layered CNN. Figure 3 shows the overall configuration of *VGG16*. It contains 5 blocks of total 16 convolution layers where the layers in each block have different widths (numbers of channels) ranging from 64 to 512 – the width is double from each block. Each block is

finished by a max-pooling layer of 2x2 pixel window. These convolution layer blocks are topped with 3 fully-connected layers where the first two layers have 4096 channels using *ReLU* activation, and the last layer has 1000 channels using *Soft-max* activation. The 1000 channels were for 1000 classes of the ImageNet dataset. Even though *VGG's* architecture is deeper, its number of parameters is similar to the traditional ConvNets architecture because it uses very small filter size of only 3x3 in each convolution layer. The 3x3 size's purpose is to capture the 3 characteristics of an image: centre, left right, up down. The model was trained using mini-batch stochastic gradient descent (*SGD*) with batch size 256, momentum 0.9 and weight decay 5e-4, initial learning rate 10-2. In the fully-connected layers, dropout regularization was set at ratio 0.5. The original image size in the training set was 224 x 224 pixels, and the input has 3 channels for RGB values for each pixel. The model took 74 epochs to train until it converged. Despite of its depth, *VGG* took less epochs for convergence compared to *ConvNets* because it uses more aggressive regularization.

The *VGG* model submitted to the 2014 ImageNet Challenge is a multi-net model in which it combined the outputs of different nets (A-E) then use this output to fine-tune only the fully-connected layers of net D (Simonyan & Zisserman, 2015). The model gave better accuracy and lower test error than older CNN models such as the 2012 and 2013 winners *ConvNets* and *Clarifai*. *GoogLeNet* is another state-of-the-art example of deep CNN with 22 layers that also uses similarly small convolution filter sizes. Its submitted multi-net model achieved a slightly better test error than the *VGG* model. However, when evaluating single-net models, *VGG*'s net E with 19 layers outperformed the *GoogleLeNet* and got the first place in the competition. As a result, it has been shown that deeper CNNs are able to perform better on image classification tasks. Moreover, *VGG* is able to perform well not only on the ImageNet dataset, but also can be applied to other similar image datasets.

3.4.2 Experiment Design and Techniques

Figure 3. Architecture of *VGG16* (config D) model (Sugata & Yang 2017)



In this study, we decided to apply transfer learning by using *VGG16* on our image classification task. Design of *VGG16* showing different layers and filter sizes can be found in Figure 3. Overall, *VGG16* contains 13 convolution layers and 3 fully-connected layers. In general, transfer learning refers to the application of learned knowledge from one dataset, usually a larger one, to a smaller dataset and is often used with CNNs (Huang, Pan & Lei 2017). As CNNs are designed for training considerably large dataset, they are often not used directly to train small datasets from scratch. Most often, CNNs are first trained on large datasets like ImageNet as in the case of *VGG*, then these pre-trained models are then applied on smaller but similar image datasets (Singh & Garzon, 2016). This is usually done by freezing the pre-trained weights of the CNN model's convolution layers, and only train its fully-connected layers on the smaller dataset. By doing so, the model is able to pick up general features of the images in the new dataset using its pre-trained weights, and these newly learned features can be used to train the fully-connected layers such that they are customized to the new dataset. Since *VGG16* has learned extensively the features of many different images of 1000 classes from the ImageNet dataset in which many of those contain digits and alphabets like our dataset, using its pre-trained weights the model should be able to pick up the features of our images quite well.

Table 7. *VGG16* and our model's Fully-connected Layers

***VGG16's* fully-connected layers**			**Our model**		
Layer	Output Shape	Param #	Layer	Output Shape	Param #
flatten (Flatten)	(None, 25088)	0	flatten (Flatten)	(None, 8192)	0
fc1 (Dense)	(None, 4096)	102764544	dense_3 (Dense)	(None, 4096)	33558528
dropout_1 (Dropout)	(None, 4096)	0	dropout_4 (Dropout)	(None, 4096)	0
fc2 (Dense)	(None, 4096)	16781312	dense_4 (Dense)	(None, 4096)	16781312
dropout_2 (Dropout)	(None, 4096)	0	dropout_5 (Dropout)	(None, 4096)	0
predictions (Dense)	(None, 1000)	4097000	dense_5 (Dense)	(None, 62)	254014

Based on this framework, we design our model such that: (1) using *VGG16's* convolution layers and their pre-trained weights to fit our images through only once, then (2) using the outputs from these convolution layers to train newly defined fully-connected layers. Our new fully-connected layers are similar to those of *VGG16*: the first two layers have 4096 channels each and use *ReLU* activation with 0.5 dropout, the only main difference is in the third layer which also uses *Soft-max* activation but has only 62 channels for 62 classes in our dataset instead of 1000 classes as the in the original model. Table 7 shows the similarity between our new model and *VGG16's* fully-connected layers.

Due to extensive parameter tuning approaches adopted here, we will spend the next subsection 3.4.3 solely on these techniques and subsection 3.4.4 on model evaluation and final results. The format of comparison tables is also slightly different.

3.4.3 Parameters Fine-tuning

We will evaluate the performance of our model using validation accuracy and loss, execution time and over-fitting issues. To fine-tune our model, we use the following hyper-parameters:

optimizer function (‘*sgd’*, ‘a*dam’*), learning rate, regularization technique (weight decay, dropout), and batch size.

(a) Optimizer function

Table 8. Performance of *SGD* Optimizer

Learning rate	Validation accuracy	Validation loss	Training accuracy	Training loss	Epoch
0.0001	80.95	71.79	75.25	95.11	26
0.001	87.48	36.12	85.89	40.31	28
0.01	**87.78**	**32.01**	85.10	39.02	29
0.1	1.61	1558.6	1.68	1523.3	1

The original model uses mini-batch ‘*sgd’* for training (Simonyan & Zisserman, 2015), so we test our model first with ‘*sgd’* then compare its performance with a more complex optimizer function, ‘*adam’*. For each optimizer, we test with four different learning rates and fix the epoch at 20 and batch size at 128. For *‘sgd’*, we use weight decay of 1e-6 and momentum of 0.9. We record the highest validation accuracy that the model reaches within 20 epochs, the epoch number that it reaches, its respective validation loss, training accuracy and training loss. Table 8 shows the performance of *‘sgd’* on our model. Learning rate 0.01 gave the highest validation accuracy as well as the lowest loss, in contrast, learning rate 0.1 has significantly worse performance compared to the rest. For *SGD*, the model is able to perform better with higher learning rate, however, with a learning rate too high such as 0.1, it starts to suffer from vanishing gradients and in this case very early from epoch 1.

Table 9. Performance of *Adam* Optimizer

Learning rate	Validation accuracy	Validation loss	Training accuracy	Training loss	Epoch
0.00001	89.62	30.13	89.66	29.41	29
0.0001	**91.09**	**24.39**	93.11	16.85	28
0.001	88.55	30.10	86.75	36.80	25
0.01	1.62	1556.2	1.63	1522.2	1

Table 9 shows the performance of ‘*adam’* on our model with four different learning rates. We also use a similar weight decay of 1e-6, the only difference is that we start with a smaller learning rate of 0.0001 for *Adam* instead of 0.001 for *sgd*. Learning rate 0.0001 achieved the highest validation accuracy of 91.09% and also the lowest loss for Adam. Using ‘*sgd’*, the model runs 20 epochs in about 5 minutes while using ‘*adam’* takes slightly longer at 7 minutes. However, ‘*adam’* is able to reach a 3.31% higher validation accuracy, and a 7.62% lower loss than *sgd*. Similar to *‘sgd’*, with a too high learning rate, in this case for ‘*adam’* is 0.01, the model suffers from vanishing gradients. It is interesting to note that ‘*adam’* begins suffering from this issue with a smaller learning rate compared to ‘*sgd’* which indicates smaller learning rates tend to work better for ‘*adam’*. Based on these results, we can conclude that our model performs better with ‘*adam’* despite its slightly longer execution time. We will use ‘*adam’* as the optimizer function to fine-tune the other hyper-parameters of our model.

(b) Batch size

Table 10. Performance of Different Batch Sizes Under VGG

Batch size	Validation accuracy	Validation loss	Runtime
128	**91.58**	24.37	19 minutes
256	91.57	23.57	11 minutes
512	91.07	23.99	8 minutes
1024	91.33	23.78	6 minutes
2048	**91.58**	**22.46**	**5 minutes**

Next, we increased the batch size of our model from 128 to 2048 to see if larger batch size helps the model to converge faster. Table 10 shows the performance of different batch sizes within 35 epochs. Overall, the model is able to run faster with increasing batch size since more data are fit to the model at the same time in each epoch. Batch size 128 and 2048 yield the same best validation accuracy of 91.58%, however using batch size 2048 the model can finish 35 epochs in significantly less time. Moreover, with larger batch size, the model is able to reduce the validation loss further within the same number of epochs which indicates larger batch size does lead to faster convergence in our model. Not only that, with faster runtime, we will be able to increase the number of epochs to train our model further and see if the performance continues to improve. As a result, we conclude that 2048 is the optimal batch size for our model, we will use this batch size to fine-tune other hyper-parameters.

(c) Regularization

Table 11. Performance of Different Dropout Ratios Under VGG

Dropout ratio	Validation accuracy	Validation loss	Training accuracy	Training loss
0.3	91.33	24.76	96.58	8.60
0.5	91.38	23.00	95.34	11.82
0.7	**91.39**	**22.89**	92.83	18.33

The original model uses dropout of ratio 0.5 in the first two fully-connected layers (Simonyan & Zisserman 2015) and we have been testing our model only with this ratio. We would like to see how different regularization levels impact the performance, so we tested our model with three different dropout ratios of 0.3, 0.5 and 0.7. Table 11 shows the performance of our model with 3 different dropout ratios within 50 epochs using batch size 2048. *Dropout* 0.7 achieves the best validation accuracy while *Dropout* 0.5 follows closely behind, however, the gap between training and validation accuracy is smaller for *Dropout* 0.7. Moreover, the model tends to reach lower validation loss with higher dropout rate. Overall, as stronger regularization is applied to our model, in this case increasing the rate of dropout, the model generalises better from the training set to the validation set. Although the original VGG16 model uses dropout 0.5, we decided to use dropout 0.7 as it helps our model reduce overfitting while maintaining similar validation accuracy and reaching lower loss.

3.4.4 Experiment Results

After having fine-tuned the hyper-parameters of our model, we have finalised the set of parameters that give the optimal performance: optimizer function *Adam* (with learning rate

0.0001, weight decay 1e-6), *dropout* ratio *0.7*, and *batch size 2048*. We then train the model in 200 epochs to see how it performs with more iterations, and to record the model's weights when it reaches its maximum validation accuracy. A call-back function from *keras*, *ModelCheckpoint()*, was used to save the model into a h5 file whenever it reaches a new maximum value for validation accuracy. In the end, the model achieves the highest validation accuracy of **92.37%** after 168 epochs, with a validation loss of **21.06%**. Our training accuracy is 94.87% and training loss is 19.81%, just differing slightly from our validation results, which indicates the model is generalising well on our validation set. Plots of the model's accuracy and loss are shown in figure 4 and 5 below. Despite training accuracy starts to increase above the validation accuracy after 80 epochs, the model still has a good fit on our dataset as both accuracies are still increasing and the gap between them is small. In terms of loss, validation loss starts off lower than training loss until epoch 80 where training loss begins to decrease faster, however, their gap is minimal. The total execution time of our model for 200 epochs is around 20 minutes, however, it converges after 168 epochs so in practice it should take less time. We run our model again with 500 epochs to see if the performance continues to increase after 200 epochs, and the model was able to reach a higher maximum validation accuracy of 92.53% at epoch 465. However, by this point the validation loss reaches 29.83% and has been increasing since around epoch 230. This indicates the model is over-fitting as the training loss is still decreasing while the validation loss is increasing. As a result, we decided that 200 is the maximum number of epochs that our model should run for it to converge.

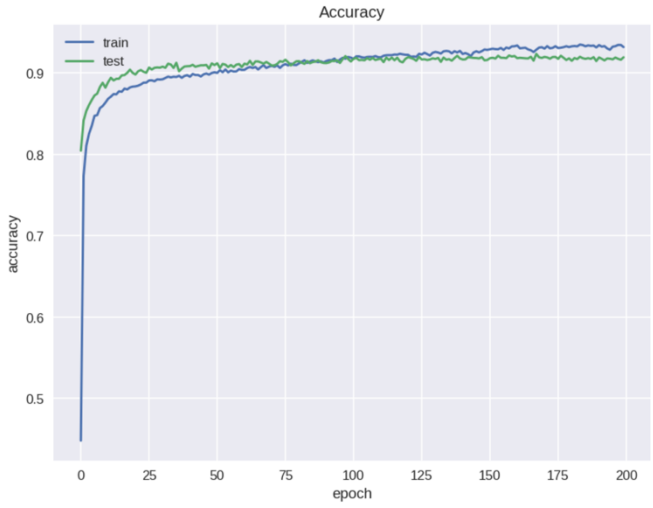


Figure 4. Training & validation accuracy of VGG16

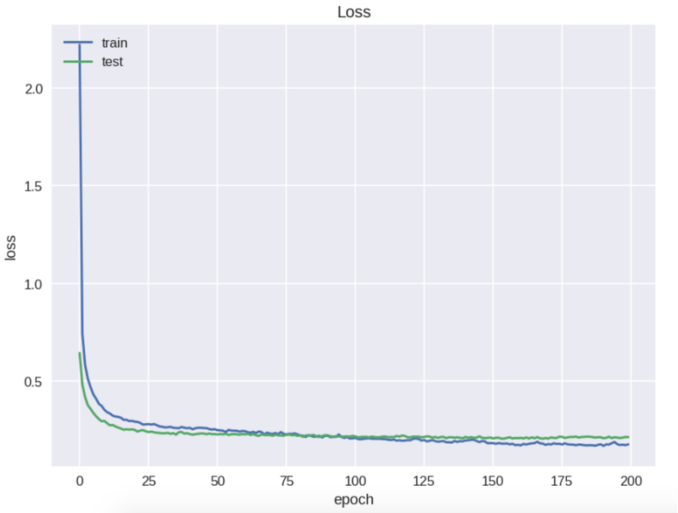


Figure 5. Training & validation loss of VGG16

4. Models Comparison

Figure 6 below compares the performance of our 3 models. Overall, our *VGG16* model was able to achieve the best validation accuracy of 92.37%, this is followed by our *ZFNet* and *LeNet* models with the accuracy of 91.54% and 88.63% respectively. Noticeably, our *VGG16* model also has the lowest validation loss of 21.06% and the smallest gap between training and validation results which indicates it has a good fit on our dataset. Regarding execution time, our *VGG16* model has the shortest running time of 10 seconds per epoch, if GPU is utilised, benefited from transfer learning and its significantly larger batch size compared to other models. For *ZFNet* and *LeNet* models, the average running time per epoch is 1 minute. As a result, we have selected *VGG16* as our final model that will be used for predicting the labels of the test dataset.

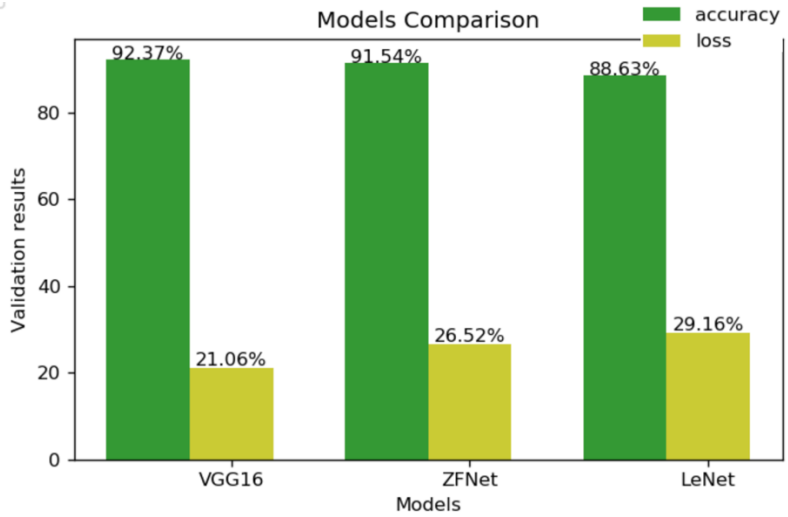


Figure 6. Performance of VGG16, ZFNet and LeNet

5. Conclusion and Future Work

Overall, the team has managed to construct three different convolution neural nets, including *LeNet, ZFNet*, and *VGG16* with the help of *Keras*. After parameters fine-tuning, our *VGG16* model outperforms among three models, with 92.37% validation accuracy with the help of transfer learning that significantly improves the results. Our *VGG16* model also has the most feasible running time per epoch - 10 seconds while it is around 1 minute per epoch for the other two models. On top of that, we tested and discovered that data augmentation did not improve the accuracy in our datasets for all 3 models.

In the future, more advanced data augmentation techniques, for instance, dimension reduction by *ZCA* whitening is worth making attempts. In addition, more parameter tuning can be explored such as other optimisers like *AdaDelta*. In addition to this, visualising feature maps though *ZFNet* is another insight could be acquired. One limitation of our *VGG16* model is that using its pre-trained weights to train the model on our dataset might not be the most suitable solution. The original model has learned features from 1000 different types of images from the ImageNet dataset in which many of them are not relevant to our dataset consisted of only 62 types of digits and alphabets. We could attempt to train the *VGG16* model on our dataset from scratch to see if its performance improves.

Appendix 1 – Instructions on how to run the code

Our submission Google Drive folder 470518197_470490653_308012798 can be accessed at **https://drive.google.com/open?id=1yh3Rp8H8fmLikGQoxe9DrTbxfpprM8SU**

Our folder contains the following files and sub-folders:

- Report.pdf
- Code
    - Algorithm
    - Input
    - Output

Code/Algorithm subfolder has the following Google Colab notebooks:

1. Models Implementation
    - **Final_Model_VGG16.ipynb** – VGG16 Implementation
    - **ZFNet.ipynb** – ZFNet implementation
    - **LeNet.ipynb** – LeNet implementation
2. Predicting labels for test data
    - **Prediction.ipynb –** generate outputs to test.txt

To run each notebook, please open them via Google Colab and double check the notebook setting is for **Python 3** and **GPU**. Please run each cell separately in order of their appearance.

Appendix 2 – Predicted labels for test dataset

The labels for test dataset can be found in file **test.txt** under the subfolder **Code/Output**

The first 10 predicted labels are

img000000.png 36

img000001.png 23

img000002.png 0

img000003.png 24

img000004.png 10

img000005.png 1

img000006.png 6

img000007.png 22

img000008.png 54

img000009.png 12

Appendix 3 – Hardware and software specifications

- Hardware
  The machine we are using is MacBook Pro 13" 2017,
  Processor: 3.5 GHz Intel Core i7
  Memory :16 GB 2133 MHz LPDDR3
  Graphics: Intel Iris Plus Graphics 650 1536 MB

- Software
  Google Colab

Appendix 4 - ZFNet Configurations & Parameters

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 61, 61, 96)	4800
max_pooling2d_4 (MaxPooling2	(None, 30, 30, 96)	0
conv2d_7 (Conv2D)	(None, 15, 15, 256)	614656
max_pooling2d_5 (MaxPooling2	(None, 7, 7, 256)	0
conv2d_8 (Conv2D)	(None, 7, 7, 384)	885120
conv2d_9 (Conv2D)	(None, 7, 7, 384)	1327488
conv2d_10 (Conv2D)	(None, 7, 7, 256)	884992
max_pooling2d_6 (MaxPooling2	(None, 3, 3, 256)	0
flatten_2 (Flatten)	(None, 2304)	0
dense_4 (Dense)	(None, 4096)	9441280
dropout_3 (Dropout)	(None, 4096)	0
dense_5 (Dense)	(None, 4096)	16781312
dropout_4 (Dropout)	(None, 4096)	0
dense_6 (Dense)	(None, 62)	254014

Total params: 30,193,662
Trainable params: 30,193,662
Non-trainable params: 0

Appendix 5 - AlexNet Configurations & Parameters

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 128, 128, 96)	11712
max_pooling2d_1 (MaxPooling2	(None, 64, 64, 96)	0
batch_normalization_1 (Batch	(None, 64, 64, 96)	384
conv2d_2 (Conv2D)	(None, 64, 64, 256)	614656
max_pooling2d_2 (MaxPooling2	(None, 31, 31, 256)	0
batch_normalization_2 (Batch	(None, 31, 31, 256)	1024
conv2d_3 (Conv2D)	(None, 31, 31, 384)	885120
conv2d_4 (Conv2D)	(None, 31, 31, 384)	1327488
conv2d_5 (Conv2D)	(None, 31, 31, 256)	884992
max_pooling2d_3 (MaxPooling2	(None, 15, 15, 256)	0
batch_normalization_3 (Batch	(None, 15, 15, 256)	1024
flatten_1 (Flatten)	(None, 57600)	0
dense_1 (Dense)	(None, 4096)	235933696
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 4096)	16781312
dropout_2 (Dropout)	(None, 4096)	0
dense_3 (Dense)	(None, 62)	254014

Total params: 256,695,422
Trainable params: 256,694,206
Non-trainable params: 1,216

Appendix 6 – *VGG* Configurations & Parameters (Simonyan & Zisserman 2015)

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

References

Bacha, M.R.A., Oukebdane, A., Hafid Belbachir, A., 2016, Pattern Recognit. Image Anal. 26: 817. https://doi.org/10.1134/S1054661816040143.

Baker, B., Gupta, O., Naik, N., Raskar, R., 2016, Designing neural network architectures using reinforcement learning, arXiv preprint arXiv:1611.02167.

Berger, P.D., Maurer, R.E., 2002, Experimental Design with Applications in Management, Engineering, and the Sciences.

Bui, V., Chang L.C., 2016, Deep learning architectures for hard character classification, Proc. Int. Conf. Art if. Int., pp. 108.

CS231n Convolutional Neural Networks for Visual Recognition [Internet]. Cs231n.github.io. 2018 [cited 2 June 2018]. Available from: http://cs231n.github.io/convolutional-networks/

Deshpande, A., The 9 Deep Learning Papers You Need To Know About (Understanding CNNs Part 3) [Internet]. Adeshpande3.github.io. 2018 [cited 3 June 2018]. Available from: https://adeshpande3.github.io/The-9-Deep-Learning-Papers-You-Need-To-Know-About.html

Goodfellow, I., Bengio, Y., Courville, A., 2016, Deep learning, Cambridge: MIT press, Nov 18.

He, K., Zhang, X., Ren, S., Sun, J., 2016, Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778.

Huang, Z., Pan, Z., Lei, B., 2017, 'Transfer Learning with Deep Convolutional Neural Network for SAR Target Classification with Limited Labeled Data', MDPI, pp. 1-21.

LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., 1998, Gradient-based learning applied to document recognition, Proceedings of the IEEE, v. 86, pp. 2278- 2324.

Ioffe, S., Szegedy, C., 2015, Batch normalization: Accelerating deep network training by reducing internal covariate shift, ICML.

Lee, C., Gallagher, P.W., Tu, Z., 2015, Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree, arXiv preprint arXiv:1509.08985.

Krizhevsky, A., Sutskever, I., Hinton, G.E., 2012, Imagenet classification with deep convolutional neural networks, Advances in neural information processing systems, pp. 1097-1105.

Lin, M., Chen, Q., Yan, S., 2013, Network in network. arXiv preprint arXiv:1312.4400.

Lowe, D., 2003, Distinctive image features from scale-invariant keypoints, IJCV, volume 20, pp. 91–110.

Niu, X., Suen, C., 2012, Novel hybrid CNN–SVM classifier for recognizing handwritten digits, Pattern Recognition, vol. 45, pp. 1318-1325.

Singh, D., Garzon, P., 2016, 'Using Convolutional Neural Networks and Transfer Learning to Perform Yelp Restaurant Photo Classification', Stanford University, pp. 4321-4329.

Sinha, A., 1999, An Improved Recognition Module for the Identification of Handwritten Digits, M.S. Thesis, MIT.

Simonyan, K., Zisserman, A, 2015, Very deep convolutional networks for large-scale image recognition, ICLR, pp. 1-8, arXiv preprint arXiv:1409.1556.

Sugata, T,.L.I., Yang, C.K., 2017, 'Leaf App: Leaf recognition with deep convolutional neural networks', InCITE, pp. 1-6.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., Rabinovich, A., 2014, Going Deeper with Convolutions, arXiv preprint arXiv:1409.4842.

Tay, Y., Lallican, P., Khalid, M., Viard-Gaudin, C., Knerr, S., 2001, An Offline Cursive Handwriting Word Recognition System, Proc. IEEE Region 10 Conf.

Zeiler, M.D., Fergus, R., 2014, Visualizing and understanding convolutional networks, European conference on computer vision, Springer, Cham., Sep 6, pp. 818-833.