Hand-Drawn Pictures Classifier

Ali Shobeiri - 260665549
ali.shobeiri@mail.mcgill.ca

Antonios Valkanas - 2606702034
antonios.valkanas@mail.mcgill.ca

Elie-Joe Haikal - 26068821
elie-joe.haikal@mail.mcgill.ca

I. Introduction

As part of our coursework in COMP 551 at McGill university we were introduced to a Kaggle competition which was computer vision challenge. Specifically, we were given a subset of the images used in the Google Quickdraw dataset which were placed randomly inside a 100 x 100 square with added noise and were asked to implement an algorithm that could identify the images. Our approach was to first remove the noise from the images and to then implement a baseline linear model to measure our performance. This was done using a Support Vector Machine (SVM) with Histogram of Gradients (HoG) feature descriptors. We then implemented our own Neural Network (NN) which showed marked improvement over random classification. Finally, we decided to build and test multiple Convolutional Neural Networks (CNN) following the architectures proposed by the state-of-the-art models for image recognition and of models used in competitions similar to this one. The best performance we were able to get was above 75% accuracy using our fine-tuned CNN.

II. Feature Design

For our feature design, we created a pipeline that would take as input our noisy dataset, denoise it, and generate a new dataset of images. Since the images are grayscale, we first apply a binary threshold, ensuring that pixels with intensity below a cutoff point are set to 0 and those above the cutoff are set to 255 (maximum pixel value for grayscale). Choosing a cutoff value for our thresholding procedure was crucial in determining the performance of our preprocessing, as such we tested five cutoff values and chose the best value to be the threshold value of 50 and used this for our preprocessing. These results are shown in table 1.

Table 1
Images classified as noise for various thresholds

Binary Thresh. Value	Total Images Found to Be Noise
50	364
100	554
150	994
200	3581
250	9284

We then further processed the binary thresholded image using OpenCV's *findCountours* function. Contours are a continuous curves joining points with the same color and intensity along a boundary. Using OpenCV, we extracted all the contours from the image, and removed all except the largest contour. The extraction of the largest contour is done by using a binary mask. In our use case, we made the assumption that the hand drawn image was the largest continuous contour, which was correct in about 96% of images in our original dataset. This can be inferred from the result in Table 1 as 364 images are found to contain noise after preprocessing with a threshold of 50 from a total dataset of 10,000 images. The results of our denoising method are shown in Fig. 1.

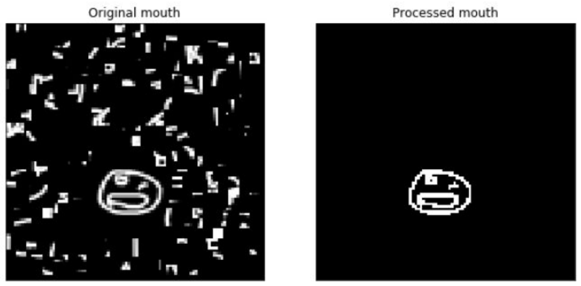

Figure 1: Training Image before denoising and after.

We then cropped the remaining contour into a 40 x 40 square. This square size was chosen after scanning through all the images and finding the largest possible contour size to be 36 pixels wide and 31 pixels tall. We chose an image size of 40 by 40 to allow for a margin of error in case the test set contained larger images. A further improvement was to resize the image after cropping, this was done to reduce the amount of black space in the image and to give our algorithms potentially more features to learn from. Therefore, the final size of the resized images was 42x42, as a result of having 40x40 pixels for the image itself along with 1 pixel of padding on all sides.

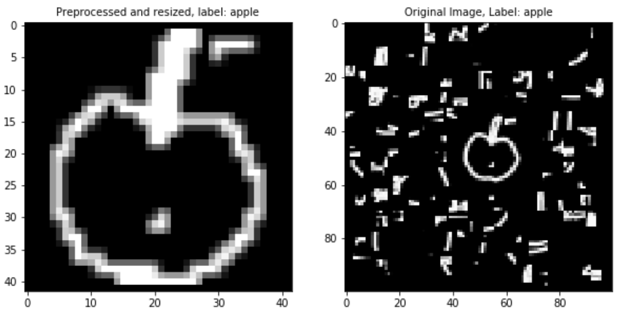
Preprocessed and resized, label: apple
Original Image, Label: apple

Figure 2: Training Image after denoising, cropping and resizing up.

As a further preprocessing step, and in hopes of improving the performance of our linear baselines, we implemented a histogram of oriented gradients (HoG) descriptor for the images. The HoG descriptor is able to determine the dominant gradient in several directions, with our particular filter being an HoG descriptor with 3 orientations, a 3x3 pixels per cell and 3x3 blocks. Generating HoG descriptors included iterating over our already processed dataset and extracting HoG features from each and saving them as a separate dataset. The result of this preprocessing step is shown in Fig.3.

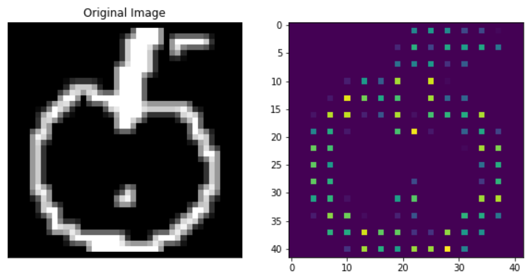

Figure 3: Histogram of Oriented gradients

III. Algorithms

A. *Linear SVM*

An SVM is a supervised learning algorithm used for both classification and regression. It attempts to find the maximum margin hyperplane that divides the set input points of different classes such that the distance between the hyperplane and the nearest point of opposing classes is maximized. The hyperparameter *C* is used to optimize the fit of the line to data and penalize the amount of samples inside the margin at the same time, with a lower C, samples inside the margin are penalized less whereas with a large C, samples are penalized more.

B. *Feedforward NN*

A feedforward NN is a fundamental deep learning model. Each layer is comprised of nodes called neurons. In a fully connected network like the one implemented, the output of each node at layer k is connected to the input of all nodes at layer k + 1. A neural network operates and learns by applying two consecutive operations, a forward propagation and a back propagation step. In the forward propagation step, inputs are fed forward in the network with outputs being generated by individual neurons. Inputs are passed through an activation function. Once an input has passed all the way through a network, the error between the generated output and the ground truth value is propagated back through the network from the output and each individual neuron updates its weights to minimize the error on the next pass.

C. *Convolution Neural Network*

A CNN extends the standard NN architecture in a manner that is powerful for image classification. An image is represented in memory as an array of pixels. CNNs are able to highlight keypoints and boundaries in an image by applying continuous convolutional filters on the image and in effect highlight and identify boundaries and simple colors on the image that are then passed on to subsequent layers to learn from. CNNs are comprised of several types of layers, some of which used in our implementation are listed below:
Layers in each CNN architecture, these layers included:

a. Convolutional Layer
The convolutional layer is the core building block of the CNN. This layer consists of a set of learnable filters where each filter is convolved against the entire image, with the kernel size of the filter being determined when the layer is created. During the forward pass of the network, a filter with the predetermined kernel size is slid over the width and height of the entire image. At each filter location, we compute dot products of all values inside the filter. This process generates an activation map that shows the response of an image to a specified filter at any point. In each convolutional layer, there may be a number of different filters that the network will learn from and use to pass values onto proceeding layers.

b. Max Pooling Layer
A max pooling layer is a functional layer that downsamples an input image. A max pooling layer works similarly to a convolutional layer in that a filter size is defined when the layer is instantiated. The filter is then slid across the entirety of the image. At each step along this sliding process, the maximum value contained in the filter window is saved to be the output of the node.

c. Dense Layer/Fully Connected Layer
A dense layer is a layer where every output from the previous layer is connected to every input in the dense layer. At each node, every node performs a non linear transform and generates one output. For example, given a Dense layer with 512 neurons, there will be exactly 512 outputs from that layer.

d. Softmax function Layer
Typically used as the final layer in a neural network based classifier. The softmax layer is able to generate the probability that its inputs correspond to a certain class. Summing all the outputs of a softmax layer gives the value of one.

IV. Methodology

For all approaches considered, we generated a stratified train-test split to account for differing frequencies in class labels. For our linear baselines, to increase the number of training data we opted for a train test split of 90/10, however in our handmade NN to more accurately gauge performance we opted for a train test split of 80/20. Similarly in our CNN, as training performance was most important and since our official test set was available on Kaggle, we opted for a train test split of 90/10 to allow for better training.

A. *Linear SVM*

For the linear SVM, we measured its performance on three different datasets. The first dataset being the original noisy dataset, the second dataset being the preprocessed and resized dataset, and the third dataset being the dataset of HoG descriptors generated from our preprocessed dataset.
We then vary the hyperparameter *C* on the best performing dataset to see if we are able to achieve better performance.

B. *Feedforward NN*

The hyperparameters for this NN are: number of hidden layers, nodes per layer, learning rate, mini-batching size, and momentum rate. We did a cross validation to determine the network architecture: it consists of a 5 80-20 splits of the entire train data, and then training a model of each split, and finally taking the average of the accuracies of the 5 models. We implemented optimization tricks: first of all, we added mini-batching to our algorithm which leads to more stable gradient descent convergence, and then momentum which accelerates this convergence and reduces oscillations. On top of that, we used minimum squared error (MSE) as our evaluation method.

C. *CNN*

To select our top performing CNN we made a comparison of three different CNN architectures on our preprocessed and resized dataset. The three networks considered were a handmade 10-layer architecture that was successful on the popular MNIST dataset, consisting of 6 convolutional layers and 2 max pooling layers, along with implementations VGG16 [1] and Densenet121 [2]. For existing networks such as DenseNet121 and VGG16, the last layers of these networks were removed and replaced with a dense layer and an activation layer that would allow classification of 31 classes. Each of the networks considered were compared on maximum validation performance over 50 epochs.

As a hyperparameter of our system, we varied different optimizers with our best performing network to measure their differences on performance, optimizers considered were Adam, RMSProp and SGD. We also experimented with varying the input dataset, passing in preprocessed and resized images and also passing in histogram of oriented gradients extracted from the image.

V. Results

A. *Linear SVM*

As it can be seen in table 2, prior to data preprocessing, we started with an initial accuracy of 3%. After denoising our images and resizing them we were able to further improve our linear SVM performance that to 24%. We then further improved our data processing algorithm by incorporating the HoG approach, the HoG approach allowed us to increase the dimensionality of our data and get several descriptors for a single pixel in the image. This improvement allowed our SVM to increase in accuracy up to 48%. The documentation of our parameter tuning for the SVM can be seen in Table 3. As shown, the optimal bias is 0.5.

Table 2
Linear SVM performance for different levels of preprocessing

Propreccessing Stack Level	Linear SVM Accuracy (%)
No Processing	3
Denoising, Resizing and Param. Tuning	24
Denoising, Resizing, and Param Tuning and HoG analysis	48

Table 3
Linear SVM parameter tuning

C Value	Linear SVM F1 (%)
0.1	45.7
0.5	47.8
2.0	44.4
5.0	43.3
50.0	43.1

B. *Feedforward NN*

Firstly, we implemented a basic NN, which accepts any number of hidden layers and neurons per layer. We used tanh as activation function. We tested the algorithm (figure 4) with two hidden layers of 2048 and 1024 neurons, learning rate of 0.00001.

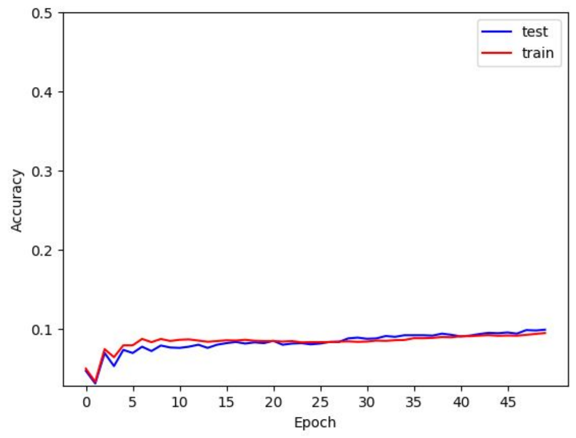

Figure 4: Train-Test accuracies of basic Neural Network

We can observe that both the training and test accuracies fluctuate for the first epochs, and then converge to an accuracy of **10%**. In this case our model does not overfit, nevertheless, does not perform well. Our first change to our NN was the activation function. We used 'leaky relu' instead of tanh. This change led to an **17.8%** increase in accuracy.

Then, we optimized our algorithm by implementing the mini-batch gradient descent (instead of full batch), and then we added momentum. To determine the network architecture, we used cross-validation. The chosen ranges of values for each parameter are:

<u>Number of hidden layers</u>: [1, 2, 3, 4]
<u>Number of Neurons per Layer</u>: [2048], [2048, 1024], [2048, 1024, 512], [2048, 1024, 512, 512]
<u>Learning rate</u>: [0.00001, 0.0001, 0.001]
<u>Momentum rate</u>: [0.0001, 0.0005, 0.001, 0.005, 0.001, 0.1]
<u>Mini-batch size</u>: [100, 200]
The table illustrates the five best accuracies achieved by our neural network using cross validation:

Table 4
Validation Performance with different hyperparameters' values

Hidden Layers	Neurons per layer	Learning Rate	Momentum Rate	Mini-batch size	Accuracy (%)
2	2048, 1024	0.00001	0.0001	200	32.84
2	2048, 1024	0.001	0.01	100	28.6
3	2048, 1024, 512	0.0001	0.0001	200	25.35
1	2048	0.001	0.0001	200	24.375
3	2048, 1024, 512	0.0001	0.0005	1000	24.23

Therefore, the best combination of hyperparameters is the one with accuracy **32.84%**. We used these hyperparameters to train our model, with and without momentum. We got the figures below:

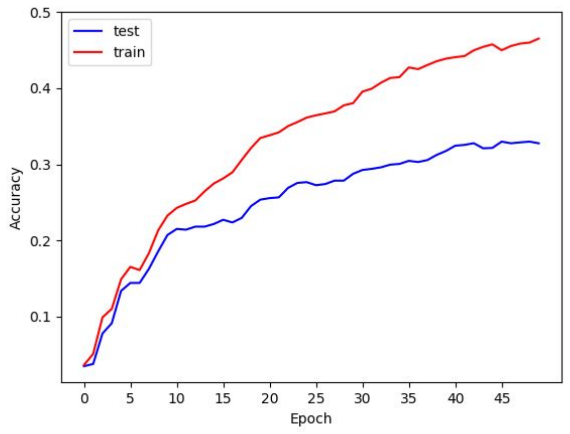

Figure 5: Model train-test accuracy without momentum

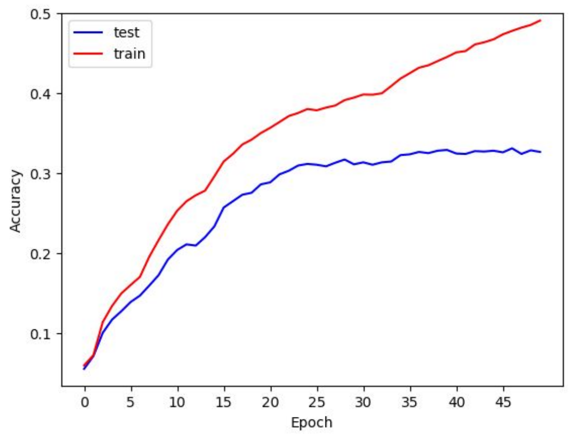

Figure 6: Model train-test accuracy with momentum

We can notice that both graphs converge to the same value (**32.84**), however, the model trained with momentum (figure 6) has a faster convergence. This validates the momentum effect. Moreover, the difference between the test and train accuracies imply that our model starts overfitting; this difference becomes more significant with higher numbers of epochs. On top of that, overfitting is more significant with momentum.

C. *Convolutional Neural Network*

The performance of all three architectures considered are seen in table 4.

Table 5
Performance for various CNN architectures

Architecture	Best Validation (%)	5-fold-Validation (%)
10-layer MNIST	63.7	63.6
DenseNet 121	69.4	69.0
VGG 16	75.4	75.3

VGG16 was measured to be the best performing model, and further improvements were performed on the VGG model. The first of which was training the VGG16 architecture with different optimizers. The optimizers considered on the VGG model as well as their validation performance is shown below, each optimizer was considered with an annealed learning rate with a minimum value of 1e-6 and trained over 50 epochs.

Table 6
Different Optimizers Validations for VGG 16

Architecture	Best Validation	5-fold-validation (%)
Adam	0.754	75.36
RMSProp	0.773	76.84
SGD	0.561	55.5

Note: We used a learning rate of 1e-6 for the three optimizers. For RMSProp we used a rho of 0.9, and for SGD we used a momentum of 0.01.

Plots of VGG16 validation accuracy and loss over the number of epochs is seen in figure 7 and figure 8 respectively. We can see that RMSProp and Adam have similar convergence speeds, with RMSProp achieving greater validation accuracy, RMSProp was the optimizer that gave us the highest test accuracy on the kaggle competition.

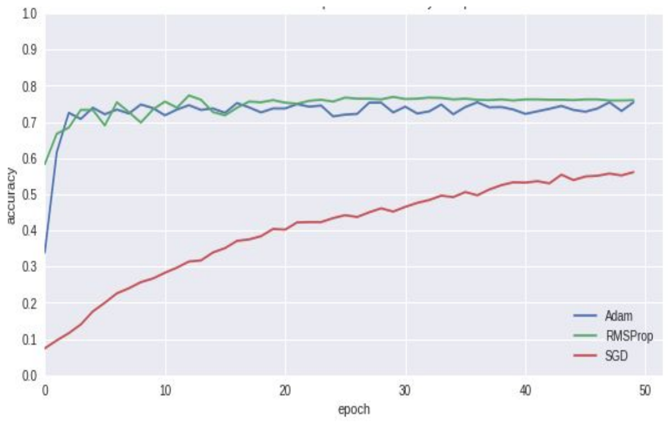

Figure 7: VGG16 Different Optimizers Accuracy vs Epocs

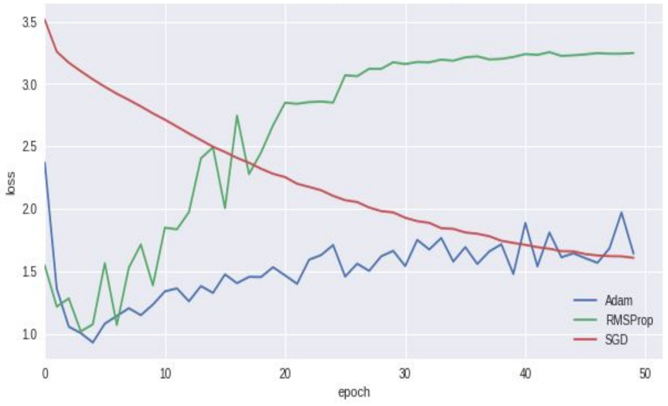

Figure 8: VGG16 Different Optimizers Loss vs Epoch

We also tested our implementation with RMSProp optimizers with a different set of input images. Instead ng in just the image, we tested our VGG16 with an image's HoG features as input, the validation and training accuracy is shown below figure 9.

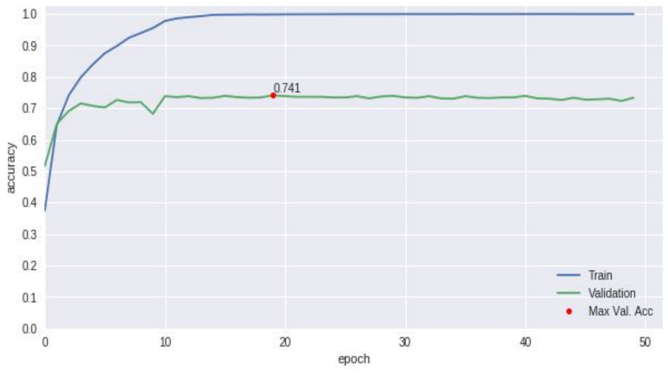

Figure 9: VGG16 HoG Features Accuracy vs Epochs

It was theorized that passing in HoG features would allow our CNN to incorporate directionality information and thus help to improve our overall performance. However, this was not the case as our overall validation accuracy was decreased. This result showed the robustness of CNNs as they are able to learn HoG features on their own without further feature extraction required.

Our final and most accurate submission was the VGG16 architecture with RMSProp as an optimizer and a learning rate of 1e-6. The training and validation accuracy as well as loss are shown in figure 10 and 11 respectively.

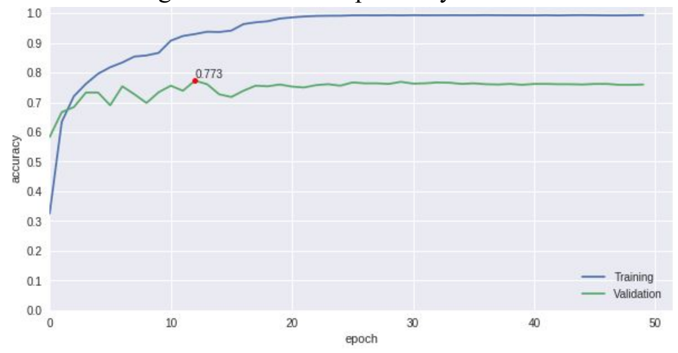

Figure 10: VGG16 Best Optimizer Accuracy vs Epochs

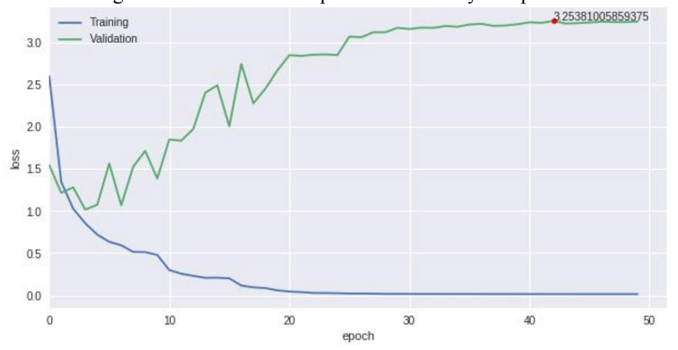

Figure 11: VGG16 Best Optimizer Loss vs Epochs

D. *Comparison of the Three Classifiers*

From our empirical results, we note that the CNN is the most powerful method when dealing with our given image data. However, we note that the SVM with HoG features was able to outperform our handmade NN architecture. This could be due to the fact that providing HoG features gives the SVM more data to make decisions about its hyperplane. In general, we conclude that CNNs seem to be superior when dealing with image based data.

VII. Discussion

A. *Methodology Pros & Cons*

Our methodology was able to maximize the effect of the linear method to close to 50% accuracy. While it is clear that the HoG processing of the images helped the linear method, it did not improve the NN and the CNN's performance. We therefore think that not all machine learning methods applied to this problem require the same preprocessing approaches to the data. Another appropriately done procedure was the hyperparameter tuning. We tested various network sizes with different momentum rates and mini-batch sizes. For the CNN we tested various state of the art architectures with various optimizers. On top of that, the three classifiers were tuned using cross validation.

In terms of the train-test splitting, using the stratified split allowed our validation test not to be biased since not all classes had the same frequency in the training set. This prevented us from non representative test sets for evaluating our models.

Our approach was using large neural networks. For the given dataset with only 10000 examples, we were bound to overfit the data even with the use of dropout layers. This was due to the fact that the number of tunable parameters of our models was far greater than the number of training examples.

Our preprocessing was able to remove the majority of noise but we had cases where images were incorrectly filtered and noise was left in the image.

B. *Future work & Improvements*

A. Preprocessing

In future, we should improve the preprocessing pipeline. While presently we do not have many examples were the preprocessing fails by allowing noise to pass through the filter, when it happens it always results in a misclassification of the image. One way to improve our preprocessing would be to use more CV2 image manipulation functions such as erosion dilation to remove noise located near the contour we select to keep for each image.

B. Feedforward Neural Network

In the future, we could better tune our neural network, by validating it on more combinations. With the time constraint, and computationally heavy task, we couldn’t validate the model on more combinations. Moreover, as noticed in figure 5 and 6, our model overfits, therefore we can use regularization to reduce overfitting within our model.
On top of that, instead of using mini-batch stochastic descent with momentum, we could optimize our network by using the Adam approach; a combination AdaGrad and RMSprop. This optimization should improve the network’s performance and increase the convergence rate.

C. Convolutional Neural Network

To improve our CNN we could potentially run our network using image augmentation which would allow us to have a much larger training dataset. As well, as our training accuracy seemed to overfit, we could add dropout layers that would regularize our learning and potentially allow our network to be more robust.

VIII. Statement of Contributions

For this project there were 5 things to do. The preprocessing, the base linear solver, the neural network, the convolutional neural network and the report. Antonios focused on the preprocessing and the baseline solver. Elie-Joe worked on the handmade neural network and Ali worked on the convolutional neural network. Finally, we all worked on the report.

IX. References

[1] K. Simonyan and A. Zisserman, "*Very Deep Convolutional Networks for Large-Scale Image Recognition*", *2014.*

[2] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, “Densely Connected Convolutional Networks,” *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.