Matti Mäki-Kihniä

Utilising computer vision to solve arithmetics on Android

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

7 April 2018

Metropolia

Author	Matti Mäki-Kihniä
Title	Utilising computer vision to solve arithmetics on Android
Number of Pages	55 pages + 11 appendices
Date	7 April 2018
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructor	Peter Hjort, Senior Lecturer

The objective of this thesis was to create a native Android application, which uses machine learning to solve hand-written addition equations from images captured with a device's camera. Additionally, the application was required to use a machine learning model trained with a pre-existing dataset and optimally be able to complete the given task on high end devices without noticeable latency.

A pretrained mobile first deep convolutional neural network from the MobileNets family was chosen as the machine learning model. The MobileNet model had relatively low requirements for processing power, making it possible to run inference on a mobile device in real time. The neural network was trained with an augmented MNIST handwritten digits dataset combined with a custom dataset for the addition operator class.

Image preprocessing and character extraction functionality was implemented on the client to allow the machine learning model to classify hand-written characters with a satisfactory accuracy. Most of the preprocessing steps which have been applied to the MNIST dataset are also performed in real time for all camera input by the application.

The outcome of the thesis is an Android application which can be used to scan and solve hand-written addition equations from a device's camera feed in real time. Testing on hand-written characters proved that the application could be used with reasonable accuracy.

Keywords	Android, machine learning, MNIST, MobileNets

Metropolia

Tekijä Otsikko Sivumäärä Aika	Matti Mäki-Kihniä Yhteenlaskujen ratkaiseminen Android-sovelluksella konenäköä käyttäen 55 sivua + 11 liitettä 7.4.2018
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Mobile solutions
Ohjaaja	Lehtori Peter Hjort

Insinöörityössä oli tavoitteena kehittää natiivi Android-sovellus, joka koneoppimista käyttäen kykenee ratkaisemaan käsinkirjoitettuja yhteenlaskuja mobiililaitteen kameralla otetuista kuvista. Sovelluksen käyttämä koneoppimismalli tuli kouluttaa vapaasti saatavilla olevalla tietoaineistolla, ja sen tuli kyetä johtamaan oikea luokittelu syötteestä ilman merkittävää viivettä.

Käytetyksi koneoppimismalliksi valikoitui MobileNets-koneoppimisperheen syvä konvolutionaalinen mobiili ensin -neuroverkko. Koneoppimismalli valittiin sen tarvitseman suhteellisen matalan laskentatehon takia, joka omalta osaltaan mahdollisti reaaliajassa tapahtuvan syötteen luokittelun. Neuroverkko koulutettiin tunnistamaan numerot 0−9 laajennetulla MNIST-tietoaineistolla ja yhteenlaskuoperaattori insinöörityötä varten käsitellyllä tietoaineistolla.

Jotta koneoppimismallin oli mahdollista luokitella kuva kameran syötteestä, tuli kuva ensin esiprosessoida ja siinä olevat merkit piti erotella muusta kuvasta. Sovellus esiprosessoi kameran syötteen reaaliajassa käyttäen samoja esiprosessointitekniikoita, joita on käytetty MNIST-tietoaineiston esikäsittelyssä.

Insinöörityön lopputuloksena valmistui vaatimuksien mukainen Android-sovellus, jolla käyttäjä pystyy ratkaisemaan laitteen kameralla kuvattuja käsin kirjoitettuja yhteenlaskuja reaaliajassa. Sovellusta testattiin käsin kirjoitetuilla merkeillä, ja sen todettiin toimivan kohtuullisella tarkkuudella.

Avainsanat	Android, koneoppiminen, MNIST, MobileNets


Metropolia

Contents

Appendix 0. Evaluation characters for class 0

Appendix +. Evaluation characters for class +

1 Introduction

Machine learning is not a recent invention, but it has peaked in popularity lately, bringing it to the public's attention. Use cases to which machine learning could be applied to have been long known, but due to constraints in processing power and data availability, it has not been possible to utilize machine learning to solve these issues. In the recent years however, these bottlenecks have been overcome thanks to improvements in algorithms, hardware and the data made available. All these factors have made it possible to successfully apply machine learning in fields such as medicine, city planning and in transportation.

Thanks to these successful applications, different entities have expressed growing interest in the technology. Governments and companies alike are striving to be the frontrunners in the space which Google CEO Sundar Pichai has speculated to become more impactful than the Internet or more profound than electricity (1). The interest in facilitating machine learning and making it available outside of the research community, has brought forth a wide variety of tools for developers to use.

Incorporating machine learning in mobile applications is a relatively new form of applied machine learning. Both leading mobile operating system developers, Google and Apple, have released tools to facilitate implementing machine learning on their platforms, however these are still relatively new technologies.

The objective of this thesis was to use the tools offered by Google to combine machine learning with a mobile application. The implementation was done as a native Android application, which uses a convolutional neural network to classify hand-written characters and addition equations. The aim was to use a mobile first convolutional neural network, in order to be able to run inference for multiple characters in real time without noticeable latency. It was intended to use a pre-existing data set of hand-written characters, as compiling a comprehensive dataset is not a simple task to undertake.

2 Machine learning

Machine learning is a subfield of artificial intelligence, which aims to solve tasks that would be hard to program by hand, but which are possible for a machine to learn. An example of such task would be an autonomous vehicle steering or optical character recognition. Even though machine learning and artificial intelligence have been trending topics lately, both have been around before this millennium, a well-known example being IBM's Deep Blue artificial intelligence (2). Machine learning was defined in 1959 by Arthur Samuel as follows:

> Field of study that gives computers the ability to learn without being explicitly programmed. (3)

The recent rise of machine learning can be credited partly to the increase in processing power as well as to the increase of data availability. There rarely seems to be enough of either one of the two; the more complex the algorithm, the more data it requires to be accurate and that much more processing power will be required to train the model. With the increase in data size and computing power, also the complexity of the machine learning tasks has increased. Training a deep convolutional neural network can take from days up to months, depending on the hardware being used.

The machine learning field has taken strides towards solving more complex problems in the recent years. A comparison between Deep Blue and AlphaGo can be made to highlight the improvements (4). Another example of the improvement in the field are the results of the yearly ImageNet Large Scale Visual Recognition Challenge (ILSVRC), which includes 1.4 million images from 1000 different categories. One of the challenges is the image classification task where the winning algorithm's classification error rate has decreased from 0.28 to 0.023 between 2010 and 2017. It should be noted that while the error rate has decreased dramatically, the 2017 ILSVRC challenge had 115 entries while the 2010 challenge had only 35 entries (5).

The improvement in the ILSVRC classification task's results can be accredited to deep convolutional neural networks, especially after AlexNet won the 2012 ILSVRC challenge. AlexNet brought down the error rate from 0.26 to 0.12, thus popularizing the use of deep convolutional neural networks. (6)

This thesis uses a retrained MobileNets convolutional neural network model. MobileNets is a class of CNN models, specifically designed to be used on devices with limited amount processing power, such as smart phones and other embedded devices. The use of MobileNets models facilitates fast and relatively low-cost inference performance compared to, for example, a retrained InceptionV3 model. (7)

2.1 Machine learning techniques

Machine learning can be divided into three branches: supervised learning, unsupervised learning and reinforcement learning. Supervised learning is an approach where the machine learning algorithm is provided a set of data and the expected outcome for the data. The given data could be for example, a set of images with their matching labels, in which case the algorithm tries to learn features from the images and map them to the correct labels and finally be able to label inputs that have not appeared in the training data. This is called classification and it is the algorithm that is applied in this thesis. (8)

Another way to apply supervised learning is with regression algorithms. Instead of trying to predict the label for an input, regression aims to predict a value for the input based on its properties. In the first lecture of the Stanford University's CS229 Machine Learning course Andrew Ng gives an often-used example of a regression problem, predicting the price of a house based on some properties such as the square meters of the property and number of bedrooms. (9)

Both of the above techniques apply specific algorithms, for classification the algorithms include logistic regression, support vector machines and random forests. For regression some algorithms are linear regression, Bayesian networks and decision trees. (8)

2.2 Image classification

Image classification is a problem where a machine learning model is tasked to assign a label to an input image. As per the Stanford University's CS231n course, image classification is one of the core problems in computer vision. Image classification can be applied to general computer vision tasks like detecting objects from an image and image segmentation, where an input image is segmented based on its contents. (10)

To input an image to a machine learning model for classification, the image needs to be first converted into the correct format, which is often a three-dimensional array of numbers. The three dimensions are width, height and depth. The depth dimension is the color channels, commonly red, green and blue (RGB). For example, a color image in the RGB color space with the spatial size of 224 x 224 would then have 224 x 224 x 3 numbers in the input. From the 150.000 numbers ranging between 0 and 255 the machine learning model would then derive the label for the input. (10)

There are multiple different challenges in classifying objects and due to the numerous possible variations in the input, it would be extremely difficult to manually write all the rules for image classification. The challenges include variation in the perspective of which the object is viewed, for example from the front or back. The object could also be partly blocked by some other object or it could be in a different scale than expected. Other possible variations in the image can be for example the color space and noise in the image signal. (10)

3 Data

In order for an image classification model to perform well, it needs to learn to classify inputs based on the general features which make up each class. By basing the classification on these features, the model will be able to provide correct outputs, even when variations such as illumination or rotation, are present in the input. For the model to be invariant to these differences, it needs to be provided with a wide range of training images for each class, including images with variations. During training the model will look at each of the provided training images and gradually adjust its perception of which details and features are commonly present in that class of images. The method of first gathering a training set of labelled images and then using it for training is called the data-driven approach. (10)

The set of data which will be used in training and testing the model is called a dataset. The dataset is often split into three different sets: training, validation and testing. The ratio between these subsets of the dataset differ, but a generally the portion of the dataset that is used for training should be above 50%. The validation subset is used during the training to evaluate how well the model is learning to generalize the features in of different classes. Once the training has been completed, the final accuracy of the model

is evaluated using the testing dataset. It is important to note that the testing subset should comprise only of images that the model has not seen during training in order for the testing results to be credible.

Compiling a comprehensive dataset, for example a dataset of hand-written digits, can be an expensive, time-consuming and difficult task to complete. Considerable efforts would have to be put towards compiling a dataset such as the MNIST, which contains 60.000 images of hand-written digits from approximately 250 different writers, all of which have been manually verified to have correct labels (11). The MNIST dataset is used in this thesis, because gathering such a dataset was not a feasible option in the scope of this thesis.

There are multiple well-known and publicly available datasets for image recognition, perhaps the most popular being the MNIST dataset. Examples of other datasets are the CIFAR-10 and CIFAR-100, both which include 60.000 labeled images of 10 and 100 classes respectively (12). ImageNet is an image database which links image URLs to labels which match the contents of the images (13). The aforementioned datasets are commonly used for machine learning model benchmarking or for training the initial parameters of a model before distributing it for retraining. A similar thing is done with the MobileNets pretrained models.

3.1 The MNIST dataset

The MNIST dataset consists of 60.000 images of hand-written numbers distributed among numbers from 0-9. The dataset was made available in 1998 and it is one of the most used datasets used for image recognition. Nowadays the dataset is no longer challenging enough to benchmark convolutional neural networks, as over 99% accuracies can be achieved. The MNIST dataset has been derived from the larger National Institute of Standards and Technology's (NIST) special databases 1 and 3 (11).

The images of characters in the MNIST dataset have been preprocessed, which is why they could be considered unsuitable for comparison with any images encountered in the real world. Image 1 displays the preprocessing steps which have been applied to each of the images in the MNIST dataset. Because of the heavy preprocessing, a model trained with the MNIST dataset will not be able to correctly classify images which has

not been through the same preprocessing steps. The last step of the preprocessing is especially impactful as it converts the binary image into a grayscale image using bi-cubic interpolation (14).

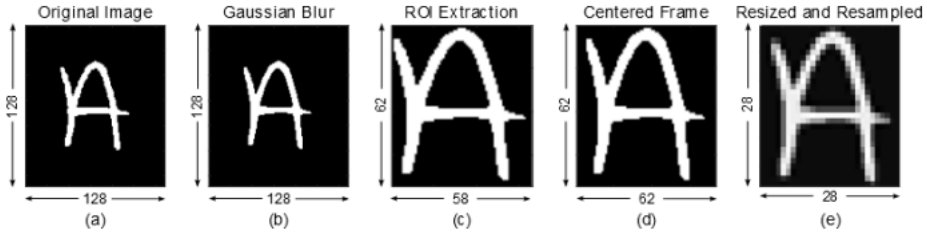


Image 1. Required preprocessing steps to convert an image from the NIST dataset to the same format as the images in the MNIST dataset (14)

This thesis applies similar preprocessing steps as displayed in the image 1, in order to be able to classify hand written numbers from an input image with a model trained on the MNIST dataset.

3.2 Data augmentation

Data augmentation aims to increase the variety of images in an existing dataset by creating new data from images already existing in the dataset, thus possibly improving the performance of the computer vision system. Andrew Ng mentions mirroring and random cropping as examples of commonly used augmentation techniques (15). The augmentation methods should be chosen keeping the task in mind, for example using mirroring or rotation to augment a dataset of animals could be considered feasible, but it might not have the desired outcome on datasets of numbers like the MNIST.

4 Convolutional neural networks

Convolutional neural networks (CNNs) have been used to achieve the best results in image classification tasks, especially after the deep convolutional neural network AlexNet was introduced in 2012 (5). Even though the algorithms used for image classification have improved significantly after 2012, using convolutions still remains one of the key building blocks for state-of-the art machine learning models, like the MobileNets, MobileNetsV2 and InceptionV3 (16, 17, 18). Even though using convolutions is currently the key to reaching the most accurate outputs in image classification, models like the

MobileNets do not implement standard convolutional layers, but instead depthwise separable convolutions (16). The use of the convolution operation remains the same however.

A simplified visualization of a convolutional neural network's layers is presented in image 2. State-of-the art convolutional neural network are significantly deeper than the one presented in the image, for example the MobileNets models have 28 layers. Despite this the image provides a rough visualization of how a feed-forward convolutional neural network works.

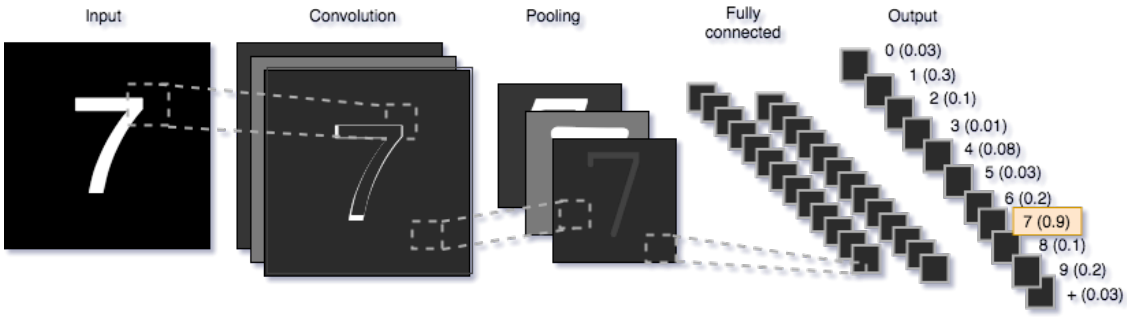


Image 2. Visualization of a CNNs layers where the input is shown on the left side and finally the output on the right.

The model definition of a neural network declares how the model is constructed, all variables in the definition are hyperparameters. Hyperparameters are those parameters of the model which the developer of the model chooses, instead of values which the network adjusts during training. Both high level decisions, such as the depth and width of the model, and low level choices, like the stride for each applicable layer, are hyperparameters.

4.1 The convolutional layer

The task of the convolutional layer is to apply a predefined amount of convolution filters on the given input. The width of the layer is defined by the number of filters in it, MobileNet's depthwise convolution layers are up to 1024 filters in width. Each filter on a convolutional layer has distinctive weights and thus each will activate on different inputs. For example, on the lower levels of the network the filters could have learnt to recognize

edges or curves while and on the higher levels the filters could be looking for cats, features of certain characters or faces, depending on the data the network has been trained on. (10)

The filters, sometimes also called a neurons or kernels, which are applied in the convolutional layer are matrices, commonly considerably smaller in size than the input matrix. The weights of the filters are iterated throughout the model's learning process in order to correctly activate on certain inputs. The output of the convolution operation is reached by convolving the filter over the input values and calculating the dot-product of the filter's and the current location's values. (10)

An example of a horizontal edge detection filter is demonstrated in image 3. The filter is a 3x3 matrix with weights that will activate when slid on vertical edges. The input and the output are both 4x4 matrices, meaning that the convolution operation is done with a stride of 1. If stride is set to 2 then every second input position is convoluted, thus resulting in smaller output. In this example the filter is applied to the area outlined with red borders on the input and with the shown equation the output value of b6 is calculated.

Besides stride each convolutional layer has another hyperparameter, padding. It defines how to pad the input on the bordering pixels, which do not necessarily have surrounding pixels on every side. For example, the border pixel a12 does not have any values for the top row of the matrix, in which case a zero-padding value can be defined so that those missing pixel values are padded as 0s. This way the input and the output sizes of the convolution will be the same with stride 1. (10)

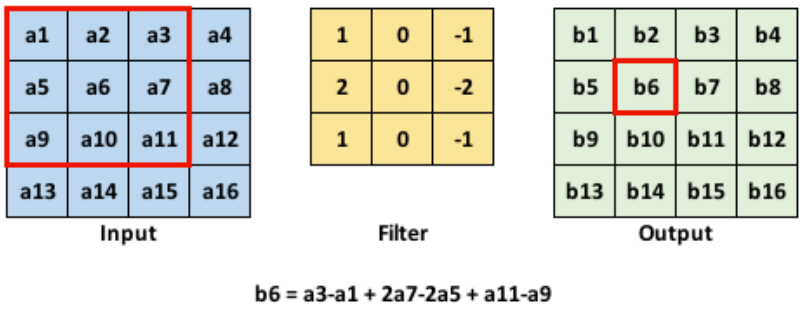


Image 3. Vertical edge detection filter applied on the area with the red borders on the input resulting in the value b6 in the output.

Convolution is followed by a detector stage. In this stage the linear output of the previous step is rectified using a nonlinear activation function (20, p. 335). One common activation function is the rectified linear unit (ReLU), which computes the function demonstrated in image 2. As can be easily seen from the graph all the activation passed to the ReLU function is simply thresholder at zero (10).

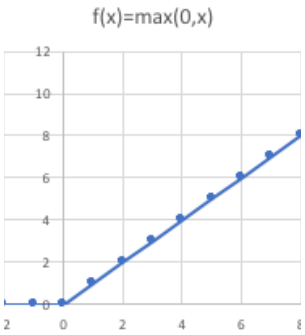


Image 4. Rectified linear unit function

4.2 Pooling

Typically, the final stage in the convolution step is pooling. Pooling takes as inputs locations in the spatial domain and outputs summary statistics of its neighborhood. As per the Deep Learning book there are some different pooling functions which are max pooling, pooling by average of a rectangular neighborhood and pooling by weighted average based on the distance from the center point. (20, p. 335-336)

An example of the max pooling function is visualized in image 4. The input for the function is displayed on the left side and the output of the function is shown on the right. In the example demonstrated in the image the pooling filter size is 2x2 and its stride is 2. In the context of pooling stride has the same intent as in convolution, meaning that it controls over how many pixels the filter is slid across over every step. When stride is set to two the values in the input will not overlap. The max pooling function filters out all but the highest value, outputting the maximum value in the neighborhood which it is slid across. (21)

Because of the pooling step small invariance in the inputs, such as noise, will be cancelled out. If the pooling function is performed with a stride of 2 the input is down-sampled by a factor of two. The effect of the down-sampling is that the number of features are

reduced and thus the parameters for the next layer of the network are reduced, as can be seen from image 4. The pooling outcome abstracts away the actual location of different activations in the input image yet keeps their location relative to other activations.

Image 5. Pooling an input with a 2x2 filter

It should be noted that pooling can be replaced by using for example a stride of 2 in the convolution step. This technique is used for example in the MobileNets models. (7)

4.3 The fully connected layer

Unlike the commonly used sparsely connected convolutional layers, the last layer or layers of a convolutional neural network are fully connected to their previous layers, hence the name fully connected layer. Generally, the purpose of the fully connected layer is to take the input from the previous convolutional layers, compute the class scores from that input and output the scores forward to the output layer. Because the previous convolutional layer's tasks have been to find features and objects from the input, the fully connected layer's task is to take that input and calculate which class best matches those activations. (21)

The fully connected layer is fully connected to the previous layer because input from all of the filters in the previous layer are needed in order to compute the class scores. The fully connected layer does not use any hyperparameters to calculate its output, the weights and biases of the layer are randomized at first and then using backpropagation and gradient decent those values are adjusted to match the output labels for the training images. (21)

4.4 Output layer

The layer following the fully connected layer is the output layer, the final layer in a neural network. This layer takes in the values calculated by the previous layer and outputs the final result. A commonly used classifier is the softmax classifier, used for example in InceptionV3 and MobileNets models, which outputs probabilities for each class in the training dataset. These output probabilities can be considered as the function's confidence about in which of the classes the input belongs to. (22)

The softmax function is used to convert the class scores provided by the last fully connected layer of the network to probability values between 0 and 1. The softmax function is used as a part of the cross-entropy loss function displayed in equation 1. The cross-entropy loss function is used to calculate how far off the output probabilities of the network are from the expected output value, the ground truth. The output of the cross-entropy loss function is then applied in the gradient decent process to adjust the network's weights in order to reduce the loss rate. It can be expected that in a standard classification task the correct output probability is a one-hot vector, meaning that one of the classes' probability value is 1 while others are 0. (22)

$$H(p,q)=-\sum_{x}p(x)\log q(x) \quad (1)$$

The cross-entropy loss function. P(x) is the expected probability, q(x) the output probability (22)

The output of the Softmax function is a vector of the same size as the layer's input, in this thesis it would be a vector of size 11, 1.

5 Training

The process of training a machine learning model often starts from randomizing all the parameters in the model. All those parameters' values then must be iteratively adjusted so that they will activate on certain inputs and thus the network can provide accurate predictions. In case of convolutional neural networks and image classifiers, the changes in the parameters are derived by inputting a batch of training images into the system and

then based on the output results, the parameters' values are moved towards a smaller error rate.

Training a convolutional neural network is a data hungry task, it is not uncommon to have tens of thousands of training images for the process. Due to the extent of the training data and the relatively small changes in the network's parameters in each iteration, the training process of networks like the InceptionV3 can take weeks on industry grade hardware. Transfer learning is an alternative to training a model from randomized weights, a pretrained model's final layer is replaced by a new layer which is then trained from randomized weights based on new training data.

5.1 Backwards propagation

The previous sections mentioned hyperparameters, weights for the different steps in a neural network that the programmer has to choose, but it did not explain how the other weights, such as values for the convolutional filters are decided. The filter values for the convolutional layer define whether the filter will activate when its input is for example a book or an edge. In machine learning these parameters are gradually adjusted by the optimization algorithm selected for training the neural network and one way to achieve this is to use backwards propagation. Backwards propagation means that the parameters of the network are gradually adjusted to minimize the output error rate, starting from the end of the network towards the start, thus improving the accuracy of the network's predictions.

One way to adjust the weights of the network is to use gradient descent. Gradient descent works by calculating the derivative of the network's output function, which can be then used to calculate how changes in the parameters of the function relate to changes in the output of the function. The optimal outcome of gradient descent optimization is to come to the global minimum loss value of the output function, however as explained in the Deep Learning book it might be difficult to achieve this point. When the global minimum cannot be reached, it is possible to settle for a local minimum point as close to the global minimum as possible. Image 5 demonstrates the minimum points of function f(x) at point x. Two different local minimum points can be observed on the curve. (20, p. 30-38)

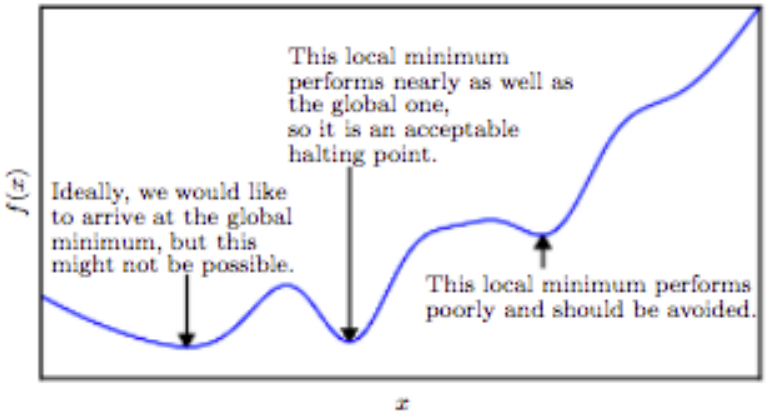


Image 6. Global minimum and local minima highlighted on function f(x). (20, p. 83)

The process of optimizing the weights happens gradually over the training process of the network. The intended result of this process is that the weights of the whole network are optimized to produce the most accurate outcome with the given training data. This is the process of changing the kernels of the convolutional layer to be able to detect points of interest in the training data.

5.2 Transfer learning

The amount of time it takes to train a convolutional neural network depends on the amount of processing power to be used, the size of the dataset, class labels and especially the depth, width and other parameters of the neural network. According to You et al. it takes 14 days to train the RestNet-50 model on the ImageNet-1k dataset on an NVidia M40 GPU (23). Even though scope of the dataset covered in this thesis is considerably lower than the ImageNet-1k, the example is given to highlight that training a complex model might not be reasonable to do with consumer grade hardware.

Transfer learning is a method with which pretrained deep learning models can be retrained using another dataset. With this approach an image recognition network, such as Google's InceptionV3, which has been trained on the ImageNet-1k dataset can be retrained to classify for example different kinds of flowers or characters with promising outcomes and reasonable training times even on consumer grade hardware (24). The reduction in the time it takes to retrain a model, compared to training a model from randomized weights, is because the only a selected part of the model must be retrained. The lower layers, ones closer to the input, have already learnt to activate on high level concepts in the inputs such as edges or colors, which means that they can be applied to

different kinds of tasks, not just classifying the ImageNet-1k dataset, thus they do not have to be retrained.

There are different ways in which transfer learning can be done. Andrew Ng mentions two different ways in his video about transfer learning, first option is fine tuning the network's existing weights by retraining the whole network, this approach does not require the weights of the network to be randomized, but instead they are fine-tuned to be more accurate with the new dataset. The second approach is removing the final output layer and replacing it with one or more layers trained with the new data. The first option is recommended if the new dataset is relatively large and the latter if the new dataset is narrow. For both of these approaches the output layer needs to be replaced and retrained for it to be able to classify the labels of the new dataset. Transfer learning has the best outcomes when the dataset has been initially trained with a dataset larger than the one to which the network is transferred to, however as mentioned in the video some improvement might be achieved through retraining a dataset. (25)

6 MobileNets models

MobileNets is a selection of open source computer vision models released by Google mid-2017. As the name suggests the MobileNets models were designed to run on devices with constrained processing power such as smart phones or embedded devices while requiring only a relatively small amount of space. Even when running on mobile devices the MobileNets models can achieve low latency and satisfactory accuracy (6). For these reasons a MobileNets model was chosen to be used in this thesis.

The MobileNets class includes the model definition, a Python file with which the model can be trained from initial weights, as well as 16 different models trained on the ImageNets dataset. The model checkpoints for different model definitions were released to allow relatively easy and fast way to make use of the MobileNets. (6) A user can choose the model which seems most suitable for the task, the MobileNets checkpoints have different number of parameters which reflect on the model's size, accuracy and latency, and then retrain the model with the desired dataset. Due to the availability of the model definition, it is also possible to train the desired model from the beginning as well as to customize the architecture.

The two key hyperparameters for the MobileNets models are the height and width multipliers. These are the only hyperparameters on which the selection of the model is based on, and they directly effect on the size and performance of the network. (6)

6.1 Depthwise separable convolutions

The MobileNets only uses one standard convolutional layer which is the first layer, all lower layers of the network use depthwise separable convolutions instead. Depthwise separable convolutions can be understood as a standard convolution factorized into two separate convolutions. The two convolutions are a depthwise convolution, a convolution which applies a filter on all input channels, and a pointwise convolution, a convolution with a kernel size of 1x1, which combines the outputs of the depthwise convolution. A standard convolution both applies a filter on all channels and then combines those into an output in a single step. (7)

The use of depthwise separable convolutions allows MobileNets to perform with significantly less latency and use less parameters compared to standard convolutions. The 3 x 3 depthwise separable convolutions in MobileNets use 8 to 9 time less computations compared to standard convolutions due having fewer parameters and thus less to compute. With the reduced number of parameters comes also the reduction in the model size, though the MobileNets models are optimized for latency, not for small size. (7)

The computational cost of a standard convolutional layer can be calculated with the formula presented in equation 2. DK is the spatial dimension, the width and height on the input, of the kernel. M is the number of input channels, N is the number of output channels, DF is the spatial dimension of the input. (7)

$$DK \cdot DK \cdot M \cdot N \cdot DF \cdot DF \quad (2)$$

The function to calculate the cost of a standard convolution (7)

Since the depthwise separable convolution is the factorization of the above, it can be broken into two parts. The computational cost for first part, the depthwise convolution, is presented in equation 3. The parameters here refer to the same ones as used in the standard convolution function, but the difference between the two is that the depthwise

convolution's cost function does not involve the output channels N, because combining the outputs of applying the filter DK on M inputs is not done in this step. (7)

$$DK \cdot DK \cdot M \cdot DF \cdot DF \quad (3)$$

The function to calculate the cost of a depthwise convolution (7)

Creation of new features happens when the outputs of the convolutional filters are combined together on a pointwise convolution layer. A pointwise convolution means applying a 1 x 1 filter on the input. The function to calculate the computational cost of a depthwise separable convolution, depthwise convolution combined with a pointwise convolution, is presented in equation 4. Once again, the parameters are the same as the ones used in the previous examples. (7)

$$DK \cdot DK \cdot M \cdot DF \cdot DF + M \cdot N \cdot DF \cdot DF \quad (4)$$

The function to calculate the cost of a depthwise convolution and a pointwise convolution combined, the depthwise separable convolution. (7)

Image 6 illustrates the difference between a standard convolution and the depthwise separable convolution and presented the parameters for each of them. The addition between depthwise convolution and pointwise convolution is not illustrated.

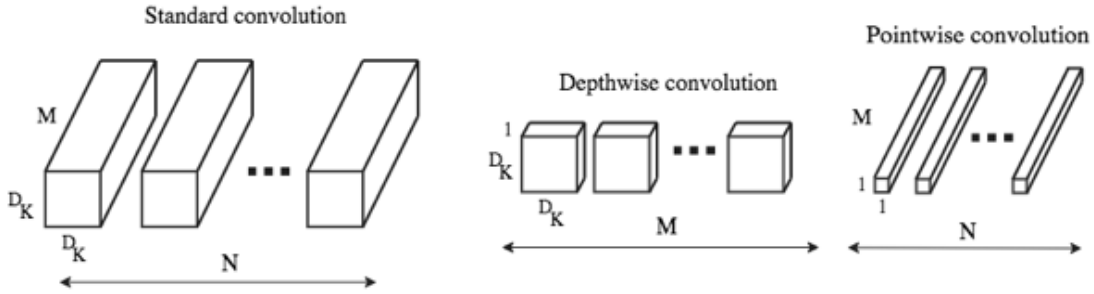


Image 7. Illustrations of standard deviation and the two convolutions which make depthwise separable convolution, the depthwise convolution and the pointwise convolution.

6.2 Architecture

As per the MobileNets paper, nearly all of the computation done by the model is done in the pointwise convolutions. Doing most of the computation this way is made possible by

the highly optimized general matrix-multiplication functions, which according to the paper, are one of the most optimized linear algebra algorithms. (7) The breakdown of the usage of computational resources can be seen in table 1. It can be seen that the MobileNets model uses nearly 95 % of the computation time in for the pointwise convolutions.

Table 1. Breakdown of the resource usage per layer type. (7)

Layer type	**Mult-adds**	**Parameters**
Pointwise convolution	94,86 %	74,59 %
Depthwise 3x3 convolution	3,06 %	1,06 %
Standard 3x3 convolution	1,19 %	0,02 %
Fully connected	0,18 %	24,33 %

The complete MobileNets architecture is displayed in table 2 where the layers, as well as their input and filter sizes are shown. After each layer a batch normalization function and a ReLU function are applied to the output, except for the fully connected layer which feeds directly to the output Softmax layer (7). It can be also noticed that the MobileNets structure has only one pooling layer because downsampling is achieved by using a stride of 2 in some of the depthwise convolutional layers.

The last depthwise convolution's stride has been changed from the original MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications paper. In the paper the layer's stride was marked as 2, which is not possible seeing that the next layer's input size is the same as that layers, which means that the filter is applied with stride 1. This can be confirmed from the MobileNets source code on GitHub where all the convolutional layers are defined and added to the _CONV_DEFS variable (26).

Table 2. MobileNets architecture for 224 x 224 input. (7)

Layer type / stride	**Filter-shape**	**Input size**
Standard convolution / s2	3 × 3 × 3 × 32	224 × 224 × 3
Depthwise convolution / s1	3 × 3 × 32	112 × 112 × 3
Pointwise convolution / s1	1 × 1 × 32 × 64	112 × 112 × 3

Depthwise convolution / s2	3 × 3 × 64	112 × 112 × 64
Pointwise convolution / s1	1 × 1 × 64 × 128	56 × 56 × 64
Depthwise convolution / s1	3 × 3 × 128	56 × 56 × 128
Pointwise convolution / s1	1 × 1 × 128 × 128	56 × 56 × 128
Depthwise convolution / s2	3 × 3 × 128	56 × 56 × 128
Pointwise convolution / s1	1 × 1 × 128 × 256	28 × 28 × 128
Depthwise convolution / s1	3 × 3 × 256	28 × 28 × 256
Pointwise convolution / s1	1 × 1 × 256 × 256	28 × 28 × 256
Depthwise convolution / s2	3 × 3 × 256	28 × 28 × 256
Pointwise convolution / s1	1 × 1 × 256 × 512	14 × 14 × 256
5 × Depthwise convolution / s1, pointwise convolution / s1	3 × 3 × 512 1 × 1 × 256 × 512	14 × 14 × 512 14 × 14 × 512
Depthwise convolution / s2	3 × 3 × 512	14 × 14 × 512
Pointwise convolution / s1	1 × 1 × 512 × 1024	7 × 7 × 512
Depthwise convolution / **s1**	3 × 3 × 1024	7 × 7 × 1024
Pointwise convolution / s1	1 × 1 × 1024 × 1024	7 × 7 × 1024
Average pooling / s1	7 × 7 pooling	7 × 7 × 1024
Fully connected / s1	1024 × 1000	1 × 1 × 1024
Softmax / s1	Classifier	1 × 1 × 1000

6.3 Hyperparameters

The selection of the correct MobileNets model depends on two hyperparameters, the width multiplier and the resolution multiplier. In total 16 different model checkpoints of MobileNets with different hyperparameter variations were released and by using the model definition it will be possible to train a MobileNets model with desired combination between these two hyperparameters. (7)

The width multiplier is a value between 0…1. It is used in thinning the network by reducing the amount of input and output channels on each layer, which directly affects the number of total parameters in the network thus making it less computationally expensive

and smaller in size. The default value for the width multiplier is 1 and those implementations where the multiplier is < 1 are considered reduced MobileNets. The equation to calculate the computational cost of a reduced MobileNet is the same as presented in equation 4 except that input channels M and output channels N are multiplied by the width multiplier. The negative result of thinning the network is reduction in accuracy, as presented in table 4. (7)

Table 3. Comparison between a reduced MobileNet model with 0.75 width multiplier and the baseline MobileNet (7)

Width multiplier	**ImageNet accuracy**	**Million parameters**
1.0 MobileNet-224	70,6 %	4,2
0.75 MobileNet-224	68,4 %	2,6

The second hyperparameter is the resolution multiplier, which is applied to the input image of the network. Looking at the parameters in table 2, it can be noticed that the input resolution is 224 × 224, which means that it is an example of a baseline MobileNet, a MobileNet model which resolution multiplier is set to 1. By lowering the resolution multiplier, it will be possible to reduce the computational cost of the model and thus decrease the inference latency. Once again, the caveat for using a reduced MobileNet model is a decrease in accuracy, as can be seen in table 4. (7)

Table 4. Comparison between a reduced MobileNet model a 0.75 width multiplier and the baseline MobileNet (7)

Width multiplier	**ImageNet accuracy**	**Million mult-adds**
1.0 MobileNet-224	70,6 %	569
1.0 MobileNet-128	64,4 %	186

It should also be noted that the resolution multiplier does not affect the parameters of the network, which is why it does not have an impact to the size of the network either. It however lowers the latency of the network by reducing the amount of multiply-accumulate operations of the network. Function 5 presents the equation with which to calculate the total computational cost of the network with the hyperparameters α as the width multiplier and ρ as the resolution multiplier. (7)

$$DK \cdot DK \cdot \alpha M \cdot \rho DF \cdot \rho DF + \alpha M \cdot \alpha N \cdot \rho DF \cdot \rho DF \quad (5)$$

The function to calculate the total computational cost of the MobileNets models with hyperparameters. (7)

6.4 Other mobile-first models

At the time of writing this, the MobileNets class could be already considered somewhat outdated, as the current machine learning space is improving rapidly. MobileNets can be considered an important milestone because they enable the usage of deep neural networks in applications running on mobile and embedded devices, and it is also the first official mobile-first computer vision model for Tensorflow (6). While newer models have improved performance compared to the MobileNets, the MobileNets are still viable to use as is demonstrated in this thesis.

Chinese research and development company Megvii released a paper about ShuffleNet, a convolutional neural network claiming to be superior to MobileNets in terms of accuracy and latency. The paper demonstrates improved accuracy over the baseline MobileNet model with reduced inference time. The ShuffleNet paper was released less than a month after the MobileNets paper. (16)

After starting this thesis, a paper about MobileNetsV2 architecture was released by some of the same research team that were behind the original MobileNets architecture. The MobileNetsV2 cannot be considered as an improved version of the original MobileNets, as it's architecture is considerably different than the original MobileNet's. The MobileNetsV2 beats both ShuffleNet and MobileNetsV1 in an ImageNet performance comparison with both the top 1 accuracy and amount of multiplication-addition operations are reduced. (17)

It should be also noted that the MobileNetsV2 paper also introduces the mobile friendly version of the feature detector Single Shot Detector (SSD) called SSDLite. SSDLite uses separable convolutions, similar to the depthwise separable convolutions explained in this chapter, in place of convolutional layers used in the standard SSD. The paper demonstrates that the SSDLite is considerably more efficient than the SSD300, SSD512 and

YOLOv2 networks while also being the most accurate one (17). The usage of the MobileNetsV2 model with the SSDLite could be considered a great improvement to the techniques used in this thesis.

7 Image preprocessing

Feature engineering is the task of modifying the input data of a machine learning model to achieve better performance which could be increased accuracy, training speed, inference latency or any combination of the aforementioned. Feature engineering used to be a common technique to use, especially in image classification, before convolutional neural networks started to be used for the task. The test data of the MNIST dataset was commonly preprocessed to reach better accuracy. The MNIST database homepage lists multiple different machine learning methods and their preprocessing steps which have been applied to the MNIST dataset (11).

Present-day neural networks do not require preprocessing to be applied to the input images in order to achieve accuracy of 99% or higher on the MNIST training set (14). In order, however, to be able to apply a neural network model trained on the MNIST data to real-life inputs, those inputs need to be preprocessed. Even though the model used in this thesis has learnt what an eight or a three looks like, it has learned those features from the MNIST training data, which is highly preprocessed. A comparison between a real-life input and an image from the MNIST training data can be seen in image 8. The model used in this thesis is not able to classify the input on the left as an eight.

Image 8. On the left an image from the client camera, on the right an image from the MNIST training data

Image preprocessing tasks described in this chapter are done in order to extract characters from images captured by the user's camera and apply the same preprocessing steps to them as has been applied on the MNIST training data. As mentioned by Prince the

choices for the selected preprocessing methods can affect the outcome and the performance of a computer vision system by at least as much as the chosen machine learning model (27, p. 323). The image preprocessing steps which are applied to the input are as follows, color-to-grayscale conversion, adaptive thresholding, smoothing, contour extraction and finally resizing using either bi-cubic or bilinear interpolation.

7.1 Color-to-grayscale conversion

Humans sense color through different wave lengths of light that is presented to our eyes, blue color is experienced by a light of 400 nm wave length while a 700 nm wave length is experienced as red. Color in computer systems is described with different color models, with the selection of the model depending on the system and its application, for example the CMYK model for print and RGB model for most displays. A source image in this thesis is an RGB image captured with the client's camera. (28, p. 745, 772)

The source image is converted from RGB to grayscale to be able to apply a contour detection algorithm on the input. While the conversion is a requirement, it is also done to reduce the amount of data that needs to be processed in the subsequent steps, thus improving the system's performance. While an RGB image is defined in the way that each of the three channels have an intensity value for each pixel, a grayscale image has only one channel for the pixel's intensity values.

Loss of information is inevitable when performing a conversion to grayscale, in this case the all information about the luminance of the image is lost as a result of the conversion. In this thesis the conversion is done with the CIE Y conversion, which simply omits all the chrominance channels of the source image and stores only the luminance for the grayscale representation. The mathematical expression for calculating the luminance is presented in equation 1. Other grayscale conversion techniques include Decolorize and Color2Gray algorithms. (29)

$$Y = 0.299 * R + 0.587 * G + 0.114 * B \quad (1)$$

The CIE Y is a simple conversion as it works by multiplying the values of each pixel's RGB channels by a predefined weight and then adds them together to provide the lumi-

nance value for that pixel. In a comparison between different color to grayscale conversion techniques conducted by Čadík (2008), the CIE Y conversion, even though simplistic, was rated at the same level as the Color2Gray conversion and better than multiple other conversion techniques (29). The multiplicands for the conversion are as defined by the International Telecommunication union in the BT.601 standard (30).

7.2 Adaptive thresholding

Thresholding is a step in the image preprocessing toolkit where an image is converted from grayscale to a binary image. Each pixel in an input image is analyzed and an intensity value is given to the pixel, depending on the defined or calculated threshold the value is often set to either 0 or 255, black or white. Thresholding can be used for example to separate objects from their background to facilitate character extraction. (31).

Creation of a binary image by the means of thresholding can be achieved by either thresholding the image with a global threshold value or by adaptive thresholding where a variable threshold value is calculated for each pixel. Both of these techniques use the same expression, the one presented in equation 2, to provide the output value. In the former technique the intensity value of each pixel in the input image is compared against a global threshold value. For example, all pixels with intensity values above the threshold could be set to white while all pixels with values below the threshold could be set to black, thus achieving a binary image as a result. Due to all pixels being compared against the same global value, this technique is prone to failure especially in cases where the source image has variable background illumination. (32)

$$dest_i = (src_i > T)?\, M : O \quad (2)$$

Threshold function for resolving output pixel value. T is the global threshold value, M is the maximum value assigned to the pixel if the condition is true, while 0 is the value set if the condition is false. (33, p. 136).

Instead of using a global threshold value for all the pixels, with adaptive thresholding the value is recalculated for each pixel. There are different ways to calculating the threshold value, for example finding the mean intensity value from a predefined sized neighborhood or the technique used in this thesis which is to calculate the weighted average of intensity values of pixels in a defined sized neighborhood surrounding the source pixel.

Unlike thresholding using a global value, adaptive thresholding can be successfully applied even when the source image's background has strong illumination gradients. (33, p. 139)

7.3 Median blur

Median blur is an image smoothing technique for which the desired outcome is an image with reduced amount of noise or camera artifacts. As the name suggests the output image will appear to be blurred, with reduced number of visible features. Median blur traverses through each of the input image's pixels and replaces their intensity value with the mean intensity value found within the source pixel's neighborhood. (33. p 109, 111)

As noted by Bradski & Kaehler (33. p 110) as opposed to blurring by mean, median blur is not as sensitive to images that are especially noisy or images with shot noise. This is due to median blurring not considering outlier values caused by for example shot noise. Other blurring techniques include Gaussian smoothing and bilateral filtering, both of which are more computationally expensive than the median blur. (33. p 111, 113)

7.4 Contour extraction

The extraction of the characters from the user input takes place in the last step of the image preprocessing. Contour extraction is a task in which contours, in other words borders, are detected from an input image. In this context input image refers to an image depicting characters, which has been preprocessed into a single channel binary image which background is black and the objects on it are white, as described in the previous chapters. The chosen contour detection algorithm traverses through all pixels in the input image and classifies them as either border or background. Once the contour detection algorithm has finished, the contours can be extracted either by simply extracting the contours by following all the pixels marked as borders or by a contour approximation method such as Freeman Chain Code of Eight Directions (33).

The contour detection algorithm used in this project is the one described by Satoshi and Abe in the whitepaper Topological Structural Analysis of Digitized Binary Images by Border Following. The authors present an algorithm which finds contours in a binary image

by the means of border following. Two variations of the same algorithm are presented, first for extracting all borders from an image, including borders inside already found borders, hole borders. The second which is used for extracting only the outer borders. This thesis uses the latter variation of the algorithm as the current scope of the thesis does not require recognition of characters inside other characters. (34)

The algorithm traverses through the pixels of the input image until it finds a pixel which satisfies the condition for either starting a new border following point or a hole border. Upon finding a new border following point the algorithm evaluates the pixel's 4-neighbor area to find the next border point to continue to. Once the algorithm encounters a new border following point it starts to assign pixels following that border a sequential number specific for that border. Once the algorithm has encountered the end of the border it continues the traversing through the pixels from where it left off, stopping once the condition for a new border following point or the end of the file is met. (34)

8 The character classifier application

As a part of this thesis an Android application, the character classifier app, was created. The hand-written number classifier has been implemented as a native Android application. The application enables users to classify numbers 0-9 and the character + written on a white paper with pen, pencil or marker as well as characters written on a whiteboard. By being able to classify the characters, the application can be used to solve addition operations. Depending on the font used, numbers printed on paper and viewed on a computer screen can be also classified with reasonable accuracies.

The implementation relies on using the OpenCV library for the input image preprocessing and character extraction and TensorFlow for input classification. The latency between capturing the input and providing a classification result depends highly on the hardware on which the application is run. The whole process can be run without almost any noticeable latency on mid to high end devices.

The application's user interface includes one screen, on which the user's device's backwards facing camera feed is displayed. A screenshot of the user interface is revealed in image 9 where points of interest are marked with numbers.

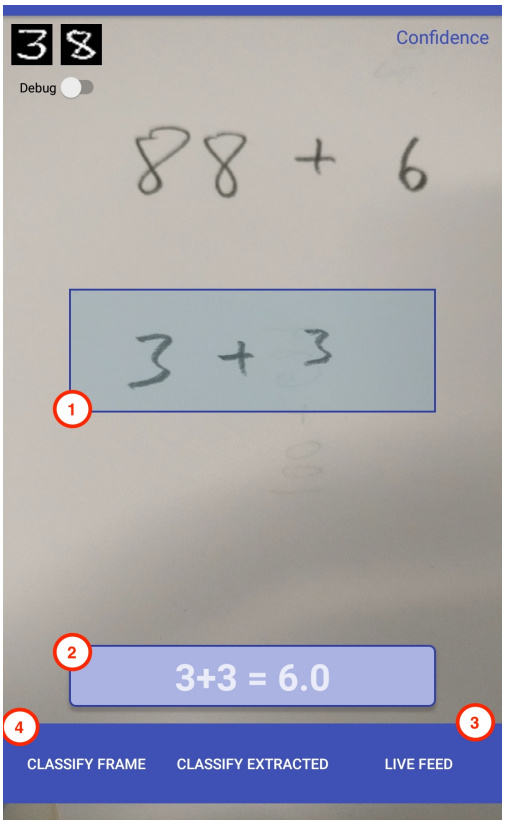


Image 9. User interface of the application

The blueish area, marked with number one, in the center of the screen is the input area. This is the area from which the input is extracted and classified, characters or objects outside of the area are not processed. The rectangle marked with number 2 is where the result of the input is presented. On the bottom is a row of buttons, the rightmost button toggles the live scanning functionality on and off. When set on, the application will constantly process the feed inside the input area. The leftmost button marked with the number 4 is used to classify the currently viewed frame.

The live processing functionality can be used without almost any noticeable latency on mid to high-end devices, where low end device users can choose to use the functionality to classify only the frame they wish.

9 Application architecture

The application has three main tasks, displaying the device's back-facing camera's feed, preprocessing the input from the camera and passing the preprocessed input to the MobileNet model for inference. All of the tasks are run in parallel, and depending on the user's device, possibly without any noticeable latency.

Besides the Android software development kit, the application has two main dependencies, OpenCV and TensorflowInferenceInterface. The OpenCV library is used in preprocessing the device's camera feed and the TensorflowInferenceInterface is used to load the machine learning model and run the inference process.

9.1 Architecture overview

The application functionality is mainly built around three different components, the camera feed received via the Android platform's Camera2 API, preprocessing done with OpenCV and finally running the inference on with the TensorflowInferenceInterface implementation. Image 9 depicts the application flow as a simplified swim lane process flow diagram.

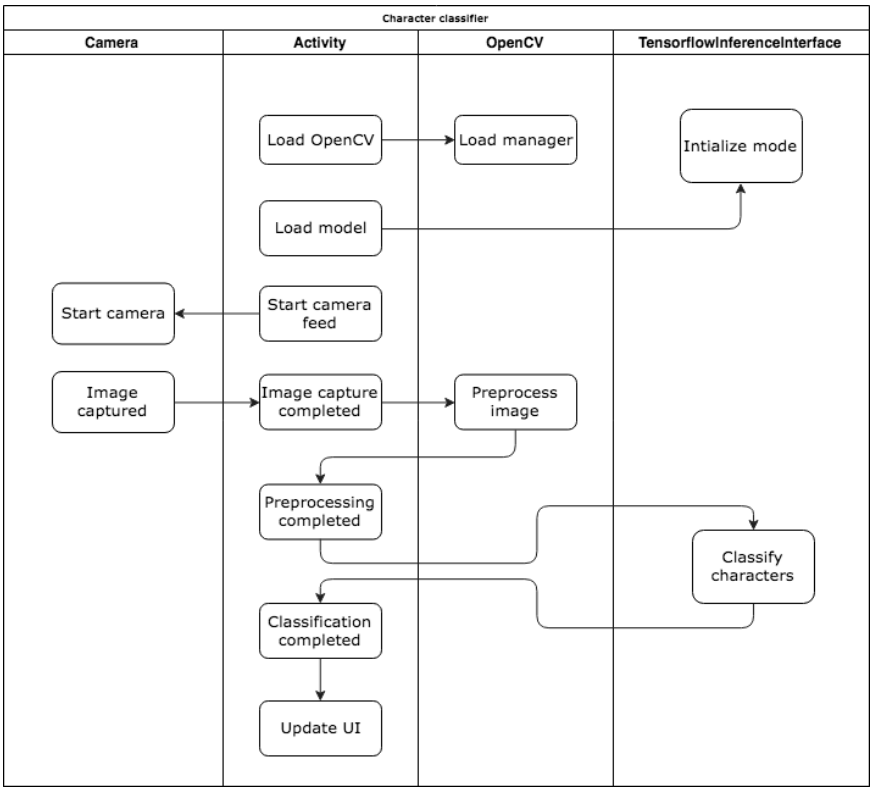


Image 10. Application process flow diagram

All function calls outside of the Activity lane are executed in separate threads, except for initializing the machine learning model. Running operations like preprocessing and inference on background threads ensures that frames are not dropped, and thus the user experience remains smooth. This is especially important as live camera feed is the main focus point of the application and any latency on its refreshing rate is easily noticeable by the user.

9.2 Threading

When an application is launched, the Android operating system creates a thread called the main thread in which that application is executed. The main thread handles tasks such as drawing the user interface and dispatching events, like the touch or swipe, to the correct user interface widgets. Because the main thread is responsible for the user interface, it should not be used for running tasks such as image preprocessing, networking or inference. Executing long running tasks on the main thread will block it from being able to draw on the user interface or propagate events. When the main thread is blocked

by a long running task and the user interface cannot be updated the application will appear to the user to be lagging. (35)

In order to prevent blocking the main thread the preprocessing and inference tasks need to be run in background threads. A background thread is similar to the main thread, except that it cannot be used to update the user interface, and it can be run in parallel to the main thread (35). Running the preprocessing and inference tasks in background threads will allow the main thread to keep refreshing the camera feed and listen to user interactions while at the same time processing the camera input.

9.3 Camera2 API

In order to receive the camera feed, the application uses the Android platform's Camera2 API which was released with Android 5.0 in 2014 (36). The Camera2 API was introduced to replace the deprecated Camera API and provides functionality lacking from the previous API, such as live camera preview. The usage of the Camera2 API comes with a caveat, which is that it can be only used on devices with Android 5.0 or greater, because it is not supported by the Android support library. In order to support devices running an Android version prior to 5.0, the application should either have to provide fallback functionality with the original camera API or the Android native development kit's native camera APIs. Fallback support has not been implemented in this application and as a result the minimum API level of the application has been set to 21.

To be able to receive the feed from the device's camera the application creates a camera capture session, which sets up the designated camera and allocates memory buffers for the captured images. The initialization of the device's camera is done asynchronously and provided callback methods will be called once the initialization has finalized. Upon successfully initializing the device's camera the onConfigured callback method will be called with an instance of the CameraCaptureRequest class as an argument. With the received camera capture request object a capture request can then be created. (36)

In order to receive the feed from the device's camera, the application creates a repeating request. As per the Android developer documentation for the Camera2 API, the setRepeatingRequest function creates an endlessly repeating request for capturing images at the maximum possible rate (37). The maximum image capture rate is sufficient

for creating a smooth live preview of the camera feed, at least on OnePlus 3, Samsung Galaxy S5 and Huawei P8 lite devices.

One of the arguments for creating the repeating request is a Handler object. A handler receives Message and Runnable objects from other threads and executes them on the thread to which it is bound to. In the implementation of this application, the handler used as an argument for the camera request is bound to a background thread called camera thread. The handler argument is used to define which thread the CaptureCallback listener should be invoked on. If the handler argument has not been provided the API will default to calling the listener on the main thread. It should be noted that listener is only used to provide a callback and some metadata about the image captured, the camera feed is drawn on the user interface on a TextureView by the main thread (37). In this application the callback is used to signal that a new frame is available and can be preprocessed and inferenced.

9.4 OpenCV

The Open Source Computer Vision Library (OpenCV) is a free computer vision library with interfaces for C++, Python and Java and support for the Android platform. The library includes functionality for a variety of computer vision tasks such as image processing, feature detection and camera preview. Unfortunately, the official documentation for Android has not been kept up to date and users have to rely on documentation created by the community in order to be able to include the library in their Android projects. An example of this is that the official documentation has instructions for setting up OpenCV for Android development for the Eclipse IDE, which has not been supported for Android development since June 2015 (38, 39).

Even though the official OpenCV documentation for Android is mostly outdated, OpenCV's general Java interface is still under active development and can be used in the Android environment. OpenCV was chosen to be used as the library for preprocessing in this thesis because it is designed for computational efficiency and real-time applications, and it contains out of the box functionality for most of the required preprocessing tasks. OpenCV is also relatively easy to learn, support resources can be found from the active community IRC channel, the comprehensive documentation and books.

In order to be able to execute OpenCV functionality in an Android application, the client device needs to either have the OpenCV Manager application installed or the application using the OpenCV library needs to bundle the required OpenCV library binaries into the application. The OpenCV manager provides all the OpenCV library binaries and thus enables applications on the user's device to share those binaries. The required binaries are not currently bundled with the character classifier application and thus in order to use the application the user will have to install the OpenCV Manager. The binaries will be bundled into the application, should the application be released (40).

9.5 TensorflowInferenceInterface

Tensorflow is an open source machine learning framework developed by the Google Brain Team, with focus on deep neural networks. Tensorflow enables defining and training machine learning models with Python, as well as running inference. The core Tensorflow is written in C++ and in order to use Tensorflow in Android the TensorflowInferenceInterface needs to be used.

The trained machine learning model is bundled with the app and it is loaded from the application's asset folder with an implementation of the TensorflowInferenceInterface. The interface is also responsible for loading the class labels matching the model and running the inference. The implementation of the interface is small and functionality for loading the model graph, feeding input to the model and fetching the output require only one method call each. (41)

An optional library for using Tensorflow on mobile is the TensorFlow Lite. TensorFlow Lite is optimized for mobile and embedded devices and thus would likely provide faster execution time than the inference interface. The library also uses a specific .tflite model file type and will be able to use the Android Neural Networks API for hardware acceleration as well as run quantized models on CPU. An additional benefit is that the library is less than 300KB, meaning that it might be possible to be used with Instant apps, at least from the perspective of size. (42)

For this application the inference interface was chosen over the TensorFlow Lite library, because the TensorFlow Lite is currently in developer preview and thus there is likely

less support material for it. The TensorflowInferenceInterface is also very easy to implement and at minimum only requires adding the dependency to the application's Gradle build file and creating a class that implements the interface.

10 Machine learning model

The machine learning model used in the character classifier application is a retrained MobileNets model. The release of the MobileNets class came with 16 pre-trained models with different hyperparameters, resolutions ranging between 128 to 224 for width multipliers 0.25, 0.50, 0.75 and 1.0, to allow for easy adoption of the models. Each of the provided models have been trained on ImageNet, meaning that the chosen model had to be retrained on the dataset used in this thesis.

The decision on which pretrained model to use is made on the basis of the two hyperparameters of the MobileNets models, resolution and width multiplier. The resolution hyperparameter reduces the latency of inference, due to reduction in the total multiplication-addition operations of the network, while the width multiplier reduces the width of the network and results in smaller networks in terms of disk space as well as increased inference latency.

As one of the goals of this thesis is to be able to run the classifier on the user's device in real time, a model with low inference latency is important. The model chosen for this thesis has a width multiplier of 0.50 and a resolution multiplier of 128. The model has 49 million multiplication-addition operations and 1.34 million parameters, which is less than 10 % of the baseline model's multiplication-addition operations and less than a third of its parameters. Comparison results between the models can be seen in table 5.

Table 5. Retraining results between different MobileNets models, the inference time is measured on a OnePlus 3 device, training accuracy is reached with a learning rate of 0.01 and a training batch size of 100 images.

Width multiplier	Resolution multiplier	Training steps	Accuracy	Inference time (ms)
0.50	128	8000	96,5%	32
1.0	224	8000	97,5%	118

The average inference time of over 100ms with the baseline MobileNets is easily noticeable by the user, while the 32ms latency of the reduced model is much harder to notice. The inference latency is likely to increase for devices with less processing power than the OnePlus 3, which was used for these tests.

10.1 Model retraining

There are different ways to approach retraining a model, for example fine-tuning all, or only a part, of the model's weights or simply removing the output layer and replacing it with a new layer retrained on the new data. The latter approach is taken in this thesis, because the approach is easier to implement and faster to experiment on with different hyperparameters. TensorFlow provides a retraining script which makes experimenting with different hyperparameters even easier (43).

Using the provided retraining script a new Softmax top layer is added to the chosen model. The added layer is then trained from randomized weights with the thesis' dataset, while retaining the existing weights for all other layers. The retraining process can be roughly split into three different steps which are choosing the hyperparameters, creating bottleneck files, training and finally testing the model. (42)

Choosing the right model is the first hyperparameter in the retraining process, the comparisons and reasoning behind the chosen model are in the previous chapter. The retraining script has multiple different hyperparameters which are used during to define the training session, such as the amount of training steps, learning rate, training, testing and validation batch sizes as well as testing and validation dataset sizes. (44)

In order to be able to achieve reliable training results the model needs to be validated and tested on data that has not been used for training the model. This is achieved by dividing the original dataset into three portions, the training, validation and testing datasets. The training process in this thesis uses the default ratio between these sets, which is 80 % for training and 10 % each for validation and testing. The data in the validation dataset is used to validate the model in between training steps, which allows the developer to notice possible problems such as overfitting early on (44). It should be noted that the script divides the dataset based on filenames, which is not optimal for

datasets like the MNIST. For MNIST the most reliable results would be received by guaranteeing that the testing dataset has characters by writers whose characters have not appeared in the training set.

The retraining process happens in steps, where a batch of random training data is passed through the model and then using backward propagation the weights being retrained are updated. One training step is defined in two different hyperparameters, the training batch size and learning rate. The batch size defines how much training data is used in one training step, and the learning rate defines by how much the weights of the layer being retrained are moved towards smaller error rate (44). The values chosen for these two hyperparameters are the script's default values, chosen through experimenting with different values.

The amount of training steps that are taken during the retraining process is also a hyperparameter. The optimal training step count can be reached by attempting to find a number at which the training no longer produces more accurate results. The ideal training step count was discovered through experimenting with values ranging from 2000 to 24000. The results for retraining with different hyperparameters can be seen in table 6.

Table 6. Retraining results with different hyperparameters on MobileNets_0.50_128 model. The testing dataset size is 8273 images.

Training steps	**Learning rate**	**Training batch size**	**Accuracy**
2000	0.010	100	94,6%
4000	0.010	100	95,6%
8000	0.010	100	96,5%
8000	0.010	50	94,2%
8000	0.005	100	96,1%
10000	0.010	100	96,5%
11000	0.010	100	96,6%
12000	0.010	100	96,9%
16000	0.010	100	96,8%
24000	0.010	100	96,7%

Based on the experiments with different hyperparameters the chosen model was trained on 12000 steps with 0.010 learning rate and the batch size of 100. This resulted in accuracy of 96,9% for the testing set of close to 8300 images.

10.2 Data augmentation

The dataset used for classes 0-9 is from the MNIST dataset while the data for the + class is from the Competition on Recognition of Online Handwritten Mathematical Expressions (CROHME) dataset converted to .jpg file format by Xai Nano (45). The MNIST dataset was augmented by applying random rotation between from +25% to -25% to a fourth of the images, while the dataset for the + class was preprocessed in multiple different steps. Example images randomly picked from each class can be seen in image 10.

Image 11. An image representing each class in the dataset from 0 on the left to the + class on the right.

As the CROHME dataset contains images drawn on a computer in one-pixel width they were not suitable to use as is. This is because the client application preprocesses all characters in the input image the same way, which will result in an output significantly different from the CROHME data. To overcome this issue multiple steps of preprocessing is applied to the images in the + class dataset.

First preprocessing step is to increase the stroke width of the characters. Increasing the stroke width is done with the scikit-image python library's morphological filtering function erosion (46). The stroke width was randomly increased between 0 and 6 pixels, resulting in + characters with varying stroke width. The result of increased stroke width can be seen in image 11 where the erosion filter has been applied to the image on the left and the output of the preprocessing is presented on it's right.

Image 12. Visualization of the preprocessing outputs for the characters in the + class. A character from the CROHME dataset on the left without any preprocessing, a character in the middle with increased stroke width and finally an image on the right with the MNIST preprocessing.

The images from the CROHME dataset are also preprocessed with the same preprocessing steps as those applied to the user's camera feed. As the preprocessing functionality had already been implemented on the character classifier application, it was decided to preprocess the + class' dataset on the OnePlus 3 device. The images were transferred to the assets of the application and complied with the application, which allowed to easily loop through all the 5000 images, apply the preprocessing steps and save them to the device's public images directory.

The outcome of these preprocessing steps can be seen in image 11, where the rightmost character is one of the characters with which the model was trained on.

The total amount of images in the dataset and their respective class distribution is shown in table 7. The amount of training data for the + class was reduced because the model's test accuracy drastically dropped when the count was at 7000 images. The reason for the reduction in the test accuracy was not discovered, however the accuracy was restored by under-sampling the + class' dataset.

Table 7. Distribution of images in the dataset between different classes.

Class label	**Image count**
0	7101
1	8100
2	7154
3	7407
4	6 983
5	6 504
6	7 111
7	7 560

8	6 966
9	7 140
+	4 005

11 Camera feed processing

Because the machine learning model has been trained on heavily preprocessed images extracted from their background, all input needs to be preprocessed the same way in order to be able to provide accurate classification results. All of the input image preprocessing is done using the OpenCV library version 3.4.0 which is available for both desktop and Android development (47). More accurately each processing function can be found from the ImgProc module's ImgProc class while utility methods such as the bitmapToMat() function can be found from the Android package's Utils class.

All of the functionality described in this chapter is implemented in the CharacterExtractor class, whose responsibility, as the name implies, is to extract the input characters from the background. It would be understandable to consider that having all the preprocessing functionality bundled together would make the CharacterExtractor a huge class, but that is not true in this case as the OpenCV team has managed to abstract most of these preprocessing steps to one line of code. Having to write very little code for each of the preprocessing steps does not make the process of developing a preprocessing system such as this one trivial.

The CharacterExtractor class has one public function, extractCharacters, which takes as an argument a Bitmap of type ARGB_8888 or RGB_565 depicting the source image. The return value of the function is an ArrayList containing the Bitmaps extracted from the source image. The most important steps in converting the input to the output are described in this chapter, with some steps like type conversions are left out.

All of the steps and their outcomes are shown in image 12. It should be noted that considerations about the preprocessing steps had to be taken from the complete preprocessing pipeline's perspective, instead just applying the same preprocessing steps to the input as have been applied to the MNIST dataset. This is because the preprocessing pipeline also includes extracting the images.

The contour extraction step is can be prone to failure in situations where it's input is noisy, or the characters do not consist of a continuous line, which are both considerations which need to be carefully taken into account when adjusting the preprocessing steps preceding contour extraction. Making the contour extraction step to work in a consistent manner could not be considered a trivial task, as it was one of the most challenging tasks in the scope of this thesis.

Some image processing functionality has not been described in this chapter, for example resizing the images, even though it might play a crucial part in the pipeline. Different resizing algorithms are used depending on if the images are upscaled or downscaled, bilinear interpolation is used when upscaling and averaging when downscaling. Bilinear interpolation is used for because it converts the extracted characters from binary images to grayscale images and averaging is to downscale as it is the fastest downscaling algorithm offered by OpenCV.

The extracted images are also centered on a black 28x28 background, once again to follow the preprocessing done to the MNIST dataset. Functionality to calculate the border size required was written specifically for this thesis, while adding the borders of given size is a function in the OpenCV library.

Preprocessing and extracting shown an image similar to the one as shown in image 12 takes on average 26ms on an OnePlus3 device.

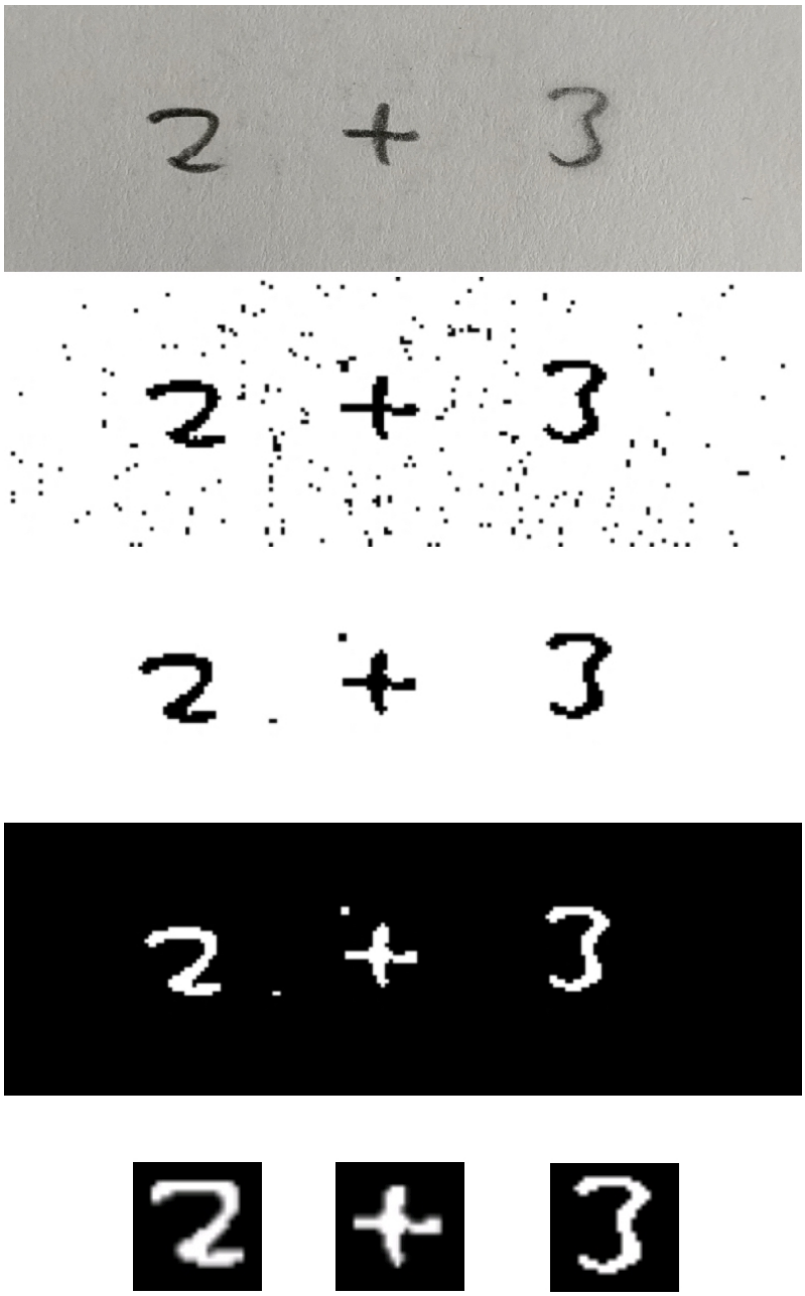


Image 13. The image preprocessing pipeline. The first image is the raw input from the device's camera, second is the input after color space conversion and adaptive threshold, third image is the output of median blurring, fourth is the blurred image with inverted pixel values which is used as input for the contour extraction and finally the images on the bottom are the output of the preprocessing function.

11.1 Color conversion to grayscale

Images received from the device's camera are in the RGB color space, but as the machine learning model has been trained on grayscale images the color information in the input is not relevant. As a result of removing the information about the input's color the

amount of data required to be processed in the following steps is reduced and the performance of the preprocessing process is increased. For these reasons converting the images from RGB color space to grayscale is done as the first step in the preprocessing pipeline. The conversion is done with the ImgProc class' cvtColor() function. The function declaration is shown in example code 1.

```
static void cvtColor(Mat src, Mat dst, int code)
```

Listing 1. ImgProc.cvtColor() function declaration (48).

An example output of the color-to-grayscale conversion is show in image 12. The image on the left side is constructed from the original source bitmap which is configured in ARGB_8888 bitmap configuration. With the ARGB_8888 configuration each pixel is stored in a total of 4 Bytes, one Byte for each of the RGBA channels. On the right side is the source image constructed from the output of the cvtColor function, this image has the RGB_565 configuration in which each pixel is stored in 2 Bytes (49).

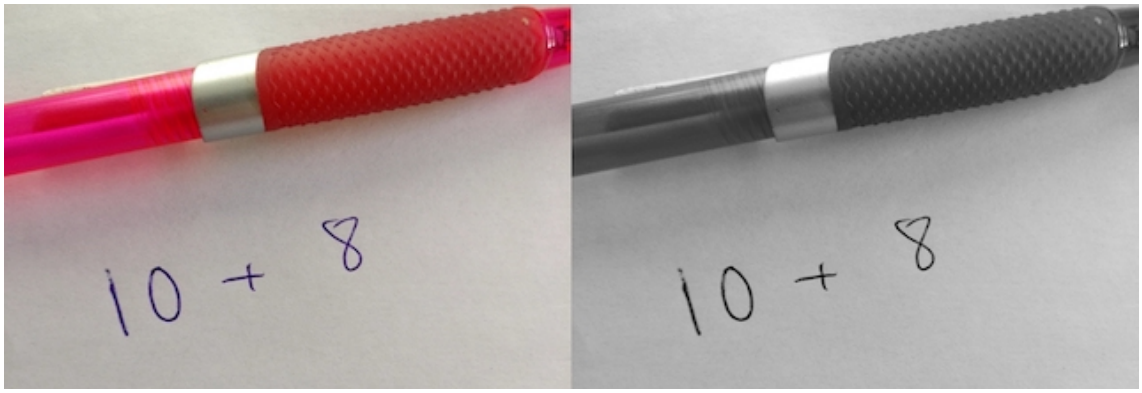


Image 14. Example of color space conversion from ARGB_8888 to grayscale to RGB_565.

11.2 Applying adaptive threshold

Adaptive threshold is an image preprocessing task which takes as input a grayscale image and converts it to a binary image. This preprocessing step is taken in order to attempt to reduce the amount of unnecessary information in the source image as well as to prepare the input image for contour recognition. An example of the output of this step can be seen in image 2. Converting an image from grayscale typically causes the loss of a great amount of information, this is demonstrated in image 13 where all information about the background as well as some information about the pen is lost. In the example the

information that was lost was not relevant and thus the outcome can be considered a success.

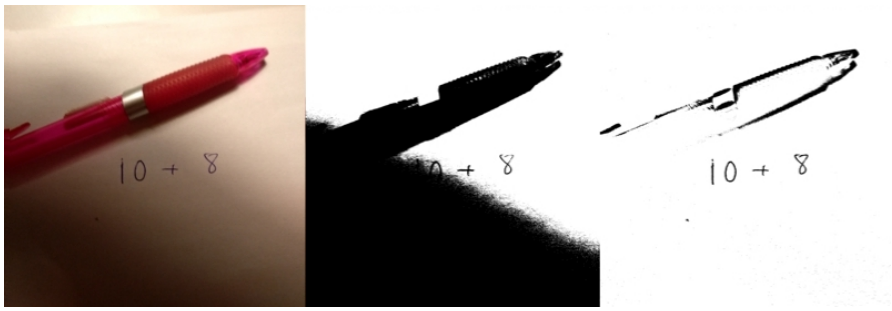


Image 15. From left to right, source image, thresholding with global threshold, adaptive thresholding with the ADAPTIVE_THRESH_GAUSSIAN_C algorithm.

The adaptiveThreshold function declaration is shown in example code 2. The input Mat file should be an 8-bit single channel image. Similarly, as the cvtColor function described in the previous chapter, the adaptiveThreshold function does not return a value but instead sets the output into the Mat variable passed as the second argument. The third parameter defines the value which is assigned to the pixels that that satisfy the thresholding condition. The next parameter defines the adaptive threshold method, the options for this parameter are explained in more detail in the next paragraph. The rest of the parameters are used to define the specifics of the thresholding. (50)

OpenCV documentation mentions that there are two different adaptive thresholding methods which are “ADAPTIVE_THRESH_MEAN_C” and "ADAPTIVE_THRESH_GAUSSIAN_C” (50). The difference between these algorithms is that the former uses a mean calculated intensity value of the neighboring pixels, while the latter calculates a weighted sum using Gaussian distribution.

```
public static void adaptiveThreshold(Mat src, Mat dst, double maxValue,
int adaptiveMethod, int thresholdType, int blockSize, double C)
```

Listing 2. Function declaration of adaptiveThreshold. (50)

Using adaptive threshold for creating a binary image is computationally more expensive than using a global threshold, but it can succeed where global thresholding cannot. One example of this is when the source image has a strong illumination gradient. A comparison of the output of between these two functions can be seen in image 13, where both thresholding functions have the same source image. In the image thresholding done with

the global threshold fails in both removing the unnecessary background illumination gradient and storing the numbers.

11.3 Median blurring

Median blur is an image smoothing technique which attempts to reduce noise from an input image. Noise in the input image can be caused for example by a high ISO setting due to low lighting. The function declaration for the medianBlur function is shown in example code 3. Similarly, to the previously described functions, medianBlur does not have a return type, but sets the output into the Mat variable passed as the second argument. The third parameter is explained in more detail in the following paragraph.

```
public static void medianBlur(Mat src, Mat dst, int ksize);
```

Listing 3. Median blur function declaration. (47)

Median blur works by replacing each pixel in the source Mat with the median pixel value in a ksize * ksize neighborhood around the center pixel. The third parameter defines the ksize, the size of the neighborhood. Median blur can be run on images with more than one channel, however at the point of reaching to step the image has already been converted to an 8-bit binary image. Image 3 shows an example output of the median blur function, where reduction of noise between the source and output images can be noticed. (33, p. 111)

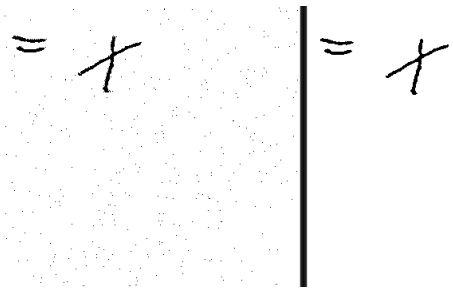


Image 16. Source image for the median blur function on the left, output on the right

3.1. Contour extraction

Following the previous preprocessing steps, the input image is sufficiently preprocessed for contour extraction. Contour extraction is done in order to be able to find external

contours of each characters in the input image, this information is then used to extract the characters from the input.

The OpenCV library offers an abstracted way to extract contours from an input image, as the complete process requires only one line of code which is calling the ImgProc class' findContours method with the required arguments. The function applies Satoshi's and Abe's contour extraction algorithm which is detailed in chapter 3. The output of the function is a list of MatOfPoint objects which are here used for finding the bounding boxes around each character and then extracting the contents of each bounding box.

One of the arguments the findContours method takes is a contour retrieval mode. The options for the contour retrieval mode include a mode for retrieving only the external borders which is the modified version of the Satoshi's and Abe's extraction algorithm detailed in chapter X and it is the one used in this project as nested contour hierarchy is not required. Other retrieval modes include retrieving the contours in different hierarchies as well as a mode for retrieving all contours without defining any hierarchy for them. Being able to retrieve the contours in a hierarchy is useful in uses cases where there exist objects inside other objects and each object needs to be retrieved separately. (33, p. 237)

The final argument is used to define the method in which the contours are approximated. As explained by Bradski and Kaehler in the book Learning OpenCV, all approximation methods besides the CV_CHAIN_CODE store the contours in sequences of points. (33, p. 236 - 238) For this project the CV_CHAIN_APPROX_SIMPLE method is chosen as it compresses the contour segments leaving only their ending points. The reasoning behind choosing this approximation mode is that only the bounding box of each character is required for the characters extraction from the source image. If it was required to extract for example a 2-dimensional geometric shape without any background, then another approximation method should be chosen.

12 Performance evaluation

As the objective of this thesis is to create an Android application which is able to correctly classify hand-written addition operations, the evaluation of the performance of the application is conducted on a set of hand-written characters representing each class as well

as addition operations. The evaluation process starts from evaluating each character class separately in order to highlight issues caused by the use of the MNIST dataset. Further evaluation is then done on addition operations of varying length and on varying surfaces.

The model's accuracy on the testing dataset is over 96%, however this accuracy does not reflect the model's accuracy on real life data. The input for the model in real life is not as uniform as the data in the MNIST dataset, even though the dataset consists of characters gathered from numerous writers, it does not cover all variations in writing styles, such as those resulting from cultural differences. Sometimes the writer's hand-writing can be so incomprehensible that they themselves will not be able to say what character they have written. The image preprocessing and extraction steps can provide suboptimal input for the model, which directly affects the model's accuracy.

12.1 Class evaluation

It is important to evaluate each character class' accuracy separately in order to be able to highlight differences between real world input and the training data. The biggest differences between misclassifications of characters are due to different writing styles. The MNIST dataset is gathered from the hand-written samples of American Census Board employees and high-school students while the evaluation inputs are gathered from Finnish adults (11). This causes a difference in the way some of the characters are written and these differences are highlighted in this chapter.

All evaluation data in this chapter has been written by one Finnish adult with a Staedtler Noris 122 HB pencil on UPM Office Copy/print A4 80g sheets. Each character is classified separately from a distance 5-7cm. Images of all the characters used for the evaluation can be found from the attachments 1-10.

Results of the evaluation can be seen in table 8 and a breakdown of the results in the following chapter.

Table 8. Evaluation of each class based on 50 hand-written characters classified with the mobile application

Class label	Correct classification percentage	Leading misclassified label
0	96 %	+ (100 %)
1	62 %	7 (80 %)
2	90 %	+ (40 %)
3	96 %	5 (100 %)
4	40 %	+ (100 %)
5	88 %	3 (80 %)
6	90 %	5 (80 %)
7	71 %	+ (85 %)
8	96 %	3 (50 %)
9	70 %	3 (66 %)
+	93 %	7 (100 %)

12.2 Breakdown of evaluation results

The average correct classification rate is 81 % between all of the classes, which cannot be considered an excellent result. The model's accuracy on the testing dataset was considerably higher than the accuracy it displayed in the evaluation. This difference is likely due to two main factors, the image preprocessing pipeline not outputting optimally preprocessed characters and the model is not being able to adequately generalize the characteristics of the classes in which it performed most poorly. The real-time performance of the application is similar to what was displayed here, as the workflows for both this evaluation and real-time classification of these are identical.

Classes whose evaluation percentage was most affected due to differences in handwriting styles are 1, 7 and 9. The evaluation data included features in some characters which are prominent in the way some Finnish people write those characters, but which are absent in the training data. Examples of this are shown in image 16. Misclassified characters from the class 1 evaluation set are shown on the left side, each of these have a common feature, the notch on the top of the character. The misclassified characters

for class 7 all have two horizontal lines, something which does not appear often in the training data.

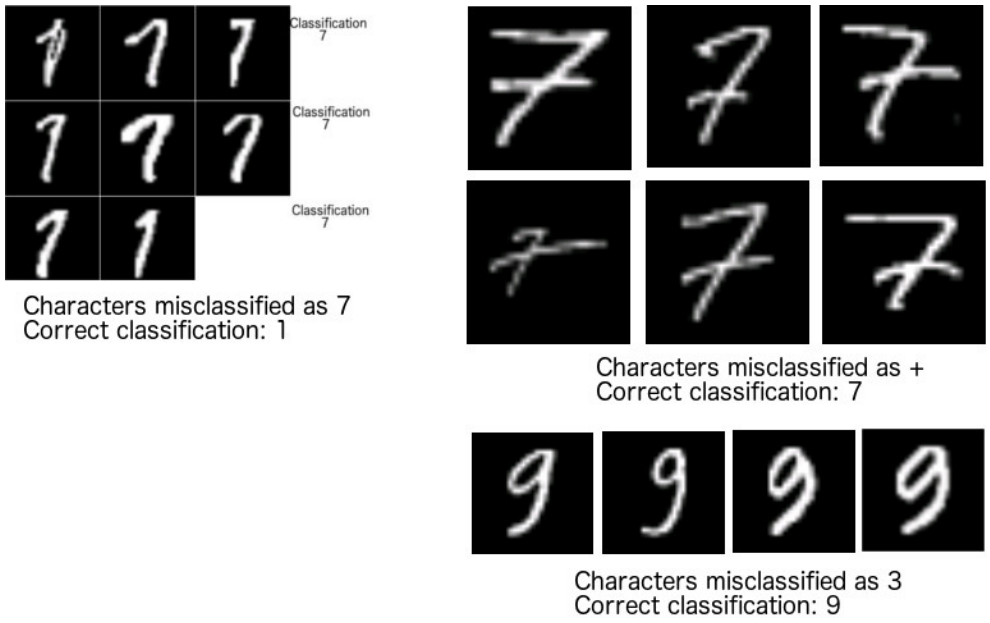


Image 17. Examples of misclassified characters from classes 1, 7 and 9.

The other main reason for misclassifications was poor performance of the machine learning model. This is the main reason for the poor evaluation performance of class 4, where the model classified most of the evaluation characters as +. A probable explanation for the misclassifications is that the 4 and + classes share some of the same features, like very distinctive vertical and horizontal lines. Some of the misclassified characters from class' evaluation set are shown in image 17, however it is not apparent from these images why they are not classified as +.

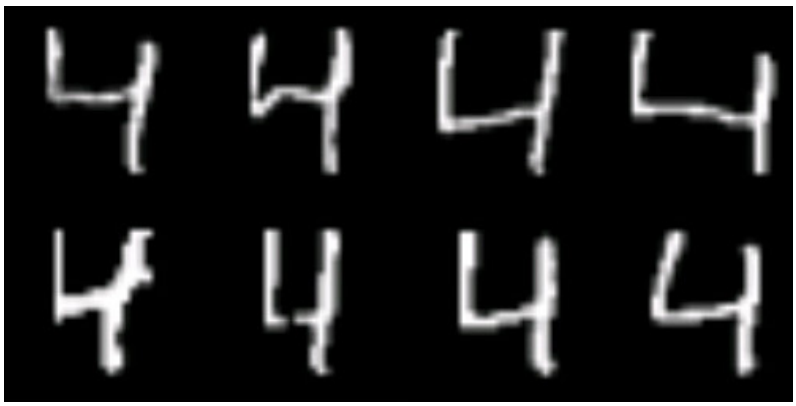


Image 18. Some of the misclassified characters for class 4.

12.3 Tolerance to rotation

Tolerance to rotation in the context of this thesis refers to the model's ability to correctly classify inputs, even when the input has been rotated. The model's tolerance for rotation can be increased for example through data augmentation by applying rotations to a subset of the training data. This technique can also be used to increase the dataset's size by keeping both the original and augmented data. As explained in the data augmentation chapter, a subset of this thesis dataset has been augmented with random rotations, which is an explanation for the satisfactory rotation tolerance.

The rotation tolerance of two classes is evaluated by applying increasing degree of clockwise rotation to one image from both classes and recording the model's classification. The classes 3 and 6 are chosen for this evaluation, as both of them have a relatively high evaluation accuracy, as can be seen from table 8.

The results for the rotation evaluation for the image of class 3 can be seen in image 18. Even when rotated 60 degrees clockwise, the model is still able to correctly classify the image with acceptable confidence. However, when increasing the rotation amount to 90 degrees, the model will no longer be able to provide correct classification and the confidence will drop considerably. In real life situations it might be expected that rotations of up to 45 degrees are encountered, rotations higher than that seem unlikely.

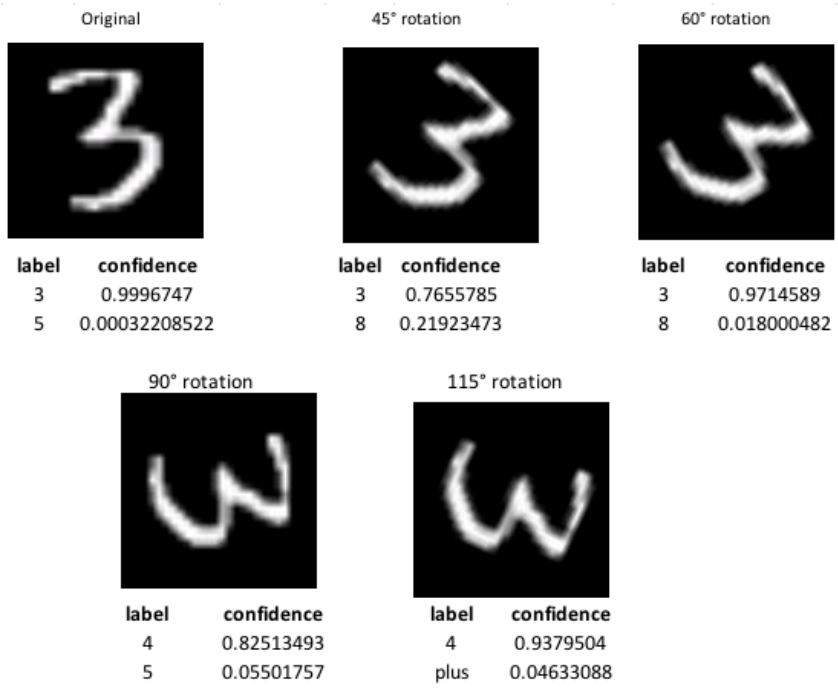


Image 19. Results for rotation evaluation for an image representing the number 3

Results for rotation evaluations of the chosen image from the class 6 are shown in image 19. Already at 45-degree rotation the first classification is incorrect, and even though the second classification is correct it's confidence is so low that the results can be considered guesses.

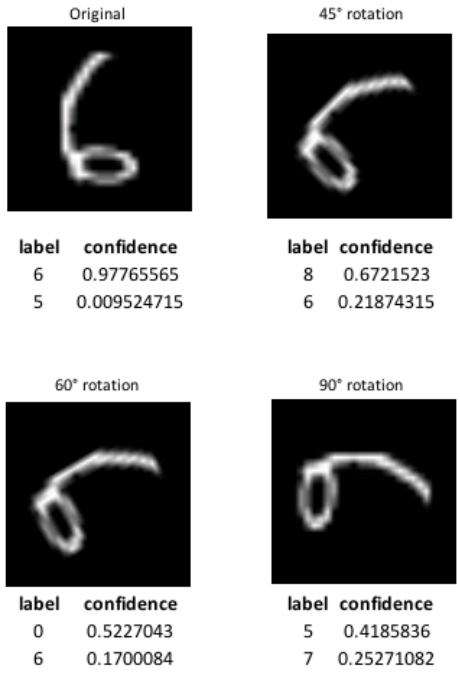


Image 20. Results for rotation evaluation for an image representing the number 6

The model's ability to classify images which are rotated could likely be increased by further augmenting the dataset with rotated images. Some of the images in the dataset used for training the model are rotated by +/- 25 degrees. By adding images with higher degree of rotation to the dataset, the model could perhaps be able to perform better in the evaluation.

12.4 Classifying addition equations

In the scope of this thesis an addition equation has been defined to have any number of characters from classes 0-9 in any order with one or more + characters in between. All characters should be on a relatively similar horizontal position and have spacing between them. The application should then technically be able to capture and classify all inputs matching these criteria.

Three example equations were captured for this evaluation, the model was able to classify all characters in each input correctly. It should be noted, that considering the results in table 8, choosing characters with poor evaluation rates should result in wrong classification results in this example as well. This chapter is thus more focused on the implementation's ability to classify multiple characters from the same input on different backgrounds with different writing tools.

The first example is an equation written on paper with a black ballpoint pen, which can be seen in image 20. The color image shown above is the input for the character extraction. Even though the input is slightly shaken, and the characters are not written with perfect lines, the application is still able to correctly extract the characters and classify all of them.

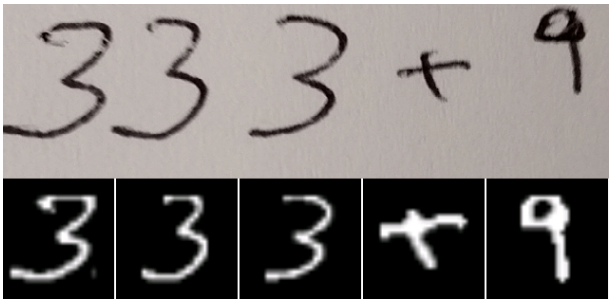


Image 21. Equation written on paper with a black ballpoint pen with each extracted character shown below.

The second example is shown in image 21. The image captured with the camera contains multiple challenges for the character extractor. First challenge is that the paper material is clearly present in the input, with close inspections this appears as small white strokes throughout the image. If the character extraction process would not be able to smooth out this noise in the image the output would be hundreds of small contours. The second challenge is the low amount of strength applied to the pencil by the writer, which appears as a relatively high number of light spots in the strokes which make up the characters 2 and 0. The extraction process is able to overcome this challenge with adaptive thresholding, which converts these pixel values correctly. The application is able to classify all the characters from this input correctly.

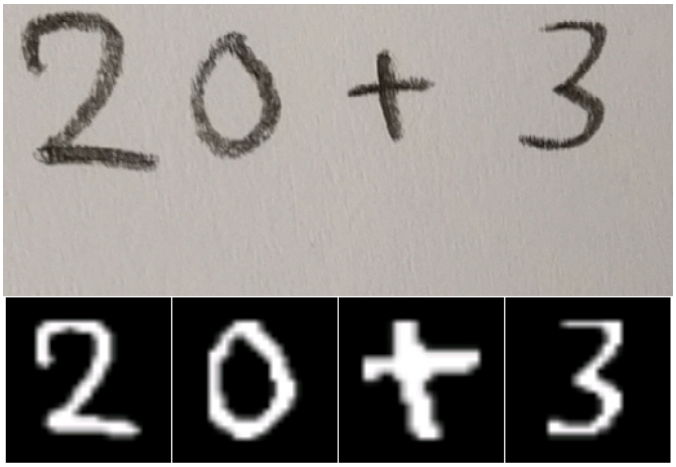


Image 22. Equation written on paper with a pencil.

The third example shown is an equation written on a on a whiteboard with a green marker pen, as can be seen in image 22. The application was used to scan the input from an estimated distance of 50 cm. The input does not provide any remarkable challenges for the character extraction process or the classifier. It should be noted that the application is able to correctly classify the characters in the image even though they are not written in black color, because the color's contrast to the background is used to find the edges contours for each character.

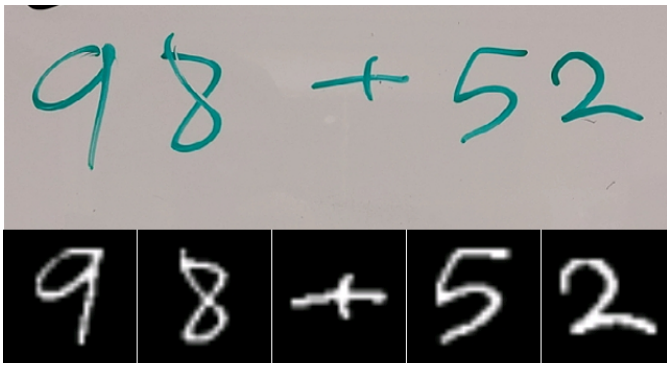


Image 23. Equation written on a whiteboard with a green marker pen.

12.5 Problematic inputs

There are some known issues in the current implementation of the character classifier application. Besides the classification issues discussed in the previous chapters, the character extraction functionality is not being able to work as intended when presented with certain inputs. Failure in correctly extracting characters from the input will most likely cause the classification to be incorrect. Because the character extraction is susceptible to failure, it increases the overall likelihood providing of great performance to decrease.

The preprocessing functionality is not able to distinct between characters and for example pens in the input and will process them the same way. This will cause the classifier to receive the extracted contours of the pen as input. In order to reduce the amount of unwanted input, a rectangle is drawn over the camera's input feed to represent the area in which the characters will be searched. A screenshot of the application is shown in image 23 where the user has positioned the camera over the input so that the pencil will not be used in the classification process.

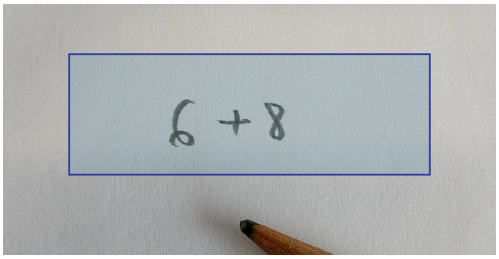


Image 24. Screenshot of the application where the input is limited by the blue rectangle.

An additional issue for the character extraction are characters written close to each other. Due to the way the contour extraction algorithm works, characters which' outlines are close to each other can be considered to consist of one contour. This will cause the extraction process to consider the two characters as one. The same issue will arise if the two characters are overlapping. An example of the issue is shown in image 24.

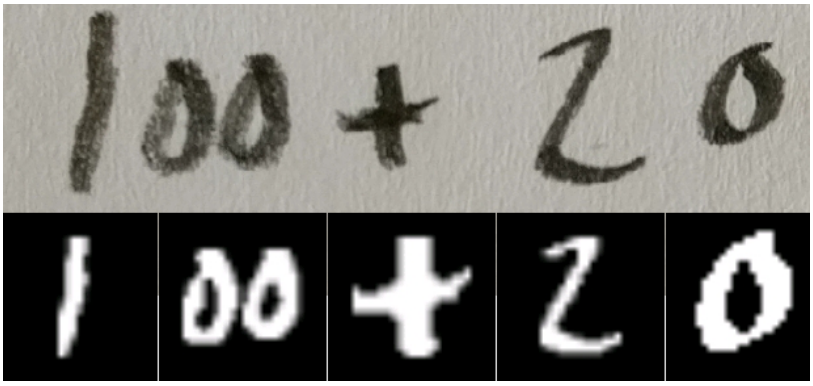


Image 25. The input shown above provides output where the second and third character are extracted as one.

13 Summary

The goal of this thesis was to create a mobile application which applies machine learning to the task of solving hand-written addition equations. Also, the aim was to use mobile first machine learning techniques to be able to complete this task with minimal latency. The MNIST data set, often used for benchmarking a machine learning model's performance, was chosen to be used as the main source of training data for the model.

The outcome of the thesis was an Android application that matches the specified requirements: it is able to process and infer the feed from the device's camera in real time and use the machine learning model's output to solve an addition equation in the camera's feed. The application is able to perform the required task with relatively satisfactory accuracy, it was tested on different inputs drawn with different writing instruments and on different backgrounds.

The performance, in terms of latency and accuracy, of both the machine learning model and the application in general could be improved. Augmenting the existing dataset, for example with the characters used in the evaluation chapter, would be a good first step to start improving the accuracy of the model.

A way to increase both the latency and the accuracy of the implementation would be to use a machine learning model which is able to detect all objects in a given input image, such as the SSDLite. This approach eliminates the need for the most error prone feature engineering task contour detection, however the usage of the MNIST dataset still comes with the caveat of having to provide the model with preprocessed inputs for inference.

Choosing the correct data is at least as impactful to the success of the outcome as selecting the correct model is. Being able to use a pre-existing dataset might be one of the dividing factors between ideas that rarely see the light of day and those which are implemented. Based on the findings in this thesis, working with the constraints resulting from using the MNIST dataset, or any other heavily preprocessed dataset, could prove to be more work than creating a similarly inclusive dataset.

References

1 Goode, Lauren. 2018. Google CEO Sundar Pichai compares impact of AI to electricity and fire. Internet source. The Verge. https://www.theverge.com/2018/1/19/16911354/google-ceo-sundar-pichai-ai-artificial-intelligence-fire-electricity-jobs-cancer> Read 7.4.2018.

2 Deep Blue. IBM100. Internet source. <https:// www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue>. Read 1.4.2018

3 Puget, Jean Francois. 2016. What Is Machine Learning? IBM developerWorks. Internet source. <https:// www.ibm.com/developerworks/community/blogs/jfp/entry/What_Is_Machine_Learning>. Read 1.4.2018

4 Metz, Cade. 2016. In a huge breakthrough, Google's AI beats a top player at the game of go. WIRED. Internet source. <https://www.wired.com/2016/01/in-a-huge-breakthrough-googles-ai-beats-a-top-player-at-the-game-of-gog>. Read 1.4.2018.

5 ILSVRC 2017 overview. 2017. Internet source. ImageNet. <http://image-net.org/challenges/talks_2017/ILSVRC2017_overview.pdf/>. Read 23.3.2018.

6 MobileNets: Open-Source Models for Efficient On-Device Vision. 2017. Internet source. Google Research Blog. <https://research.googleblog.com/2017/06/mobilenets-open-source-models-for.html>. Read 13.3.2018.

7 Howard, Andrew G. & Zhu, Menglong & Chen, Bo & Kalenichenko, Dmitry & Wang, Weijun & Weyand, Tobias & Andreetto, Marco & Adam, Hartwig. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. 2017. Internet source. eprint arXiv. <https://arxiv.org/abs/1704.04861>. Read 1.3.2018.

8 Ebert, Christof & Louridas, Panos. 2016. Machine Learning. IEEE Software. Volume: 33, Issue: 5, pages 110-115. Read 23.3.2018.

9 Ng, Andrew. 2007. CS229 - Machine Learning. Internet source. Stanford University. <https://see.stanford.edu/Course/CS229/47/>. Read 23.3.2018.

10 CS231n Convolutional Neural Networks for Visual Recognition. 2017. Internet source. Stanford University. <https://cs231n.github.io/classification/>. Read 13.3.2018.

11 THE MNIST DATABASE of handwritten digits. Internet source. <http:// yann.lecun.com/exdb/mnist>. Read 24.3.2018.

12 The CIFAR-10 and CIFAR-100 datasets. Internet source. <https://www.cs.toronto.edu/~kriz/cifar.html >. Read 24.3.2018.

13 ImageNet. Internet source. <http://www.image-net.org>. Read 24.3.2018.

14 Afshar, Saeed & Cohen, Gregory & van Schaik, André & Jonathan Tapson. 2017 EMNIST: an extension of MNIST to handwritten letters. arXiv. Internet source. <https://arxiv.org/abs/1702.05373>. Read 15.3.2018.

15 Ng, Andrew. 2017. CNN21. Data Augmentation. Internet source. <https://www.youtube.com/watch?v=AxC-L46vSmY>.

16 Lin, Mengxiao & Sun, Jian & Zhang, Xiangyu & Zhou, Xinyu. 2017. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. arXiv. <https://arxiv.org/abs/1707.01083>. Read 13.3.2018.

17 Chen, Liang-Chieh & Howard, Andrew & Sandler, Mark & Zhmoginov, Andrey & Zhu, Menglong. 2018. Inverted Residuals and Linear Bottlenecks: Mobile Networks for Classification, Detection and Segmentation. arXiv. <https://arxiv.org/abs/1801.04381v2 >. Read 13.3.2018.

18 Improving Inception and Image Classification in TensorFlow. 2016. Internet source. Google Research Blog. <https://research.googleblog.com/2016/08/improving-inception-and-image.html/>. Read 25.3.2018.

19 Yosinski, Jason. 2015 Deep Visualization Toolbox. Internet source. <https://www.youtube.com/watch?v=AgkfIQ4IGaM>

20 Goodfellow, Ian. Bengio, Yoshua. Courville, Aaron. 2016. Deep Learning. Stanford University. MIT Press. Read 13.3.2018.

21 CS231n Convolutional Neural Networks for Visual Recognition. 2017. Internet source. Stanford University. < https://cs231n.github.io/convolutional-networks/>. Read 13.3.2018.

22 CS231n Convolutional Neural Networks for Visual Recognition. 2017. Internet source. Stanford University. <https://cs231n.github.io/linear-classify/>. Read 13.3.2018.

23 You, Yang. Zhang, Zhao. Hsieh, Cho-Jui. Demmel, James. Keutzer, Kurt. ImageNet Training in Minutes. 2017. eprint arXiv:1709.05011. <https://arxiv.org/abs/1709.05011/>. Read 13.3.2018.

24 How to Retrain Inception's Final Layer for New Categories. 2018. TensorFlow Develop. Internet source. <https://www.tensorflow.org/tutorials/image_retraining>. Read 13.3.2018.

25 Ng, Andrew. 2017. Transfer Learning (C3W2L07). TensorFlow Develop. Internet source. <https://www.youtube.com/watch?v=yofjFQddwHE>. Read 13.3.2018.

26 MobileNet v1 model definition. 2017. Internet source. Tensorflow. <https://github.com/tensorflow/models/blob/master/research/slim/nets/mobilenet_v1.py/>. Read 13.3.2018.

27 Prince, Simon J.D. 2012. Computer vision: models, learning and inference. Cambridge: Cambridge University Press.

28 Hughes, John F. & van Dam, Andries & McGuire, Morgan & Sklar, David F. & Foley, James D. & Feiner, Steven K. & Akeley, Kurt. 2013. Computer graphics: principles and practice (3rd ed.). Boston: Addison-Wesley Professional.

29 Ĉadík, M. 2008. Perceptual Evaluation of Color-to-Grayscale Image Conversions. Computer Graphics Forum. Vol. 27, p. 1745 - 1754.

30 Recommendation ITU-R BT.601-7: Studio encoding parameters of digital television for standard 4:3 and wide-screen 16:9 aspect ratios. 2017. International Telecommunication Union. Geneva: ITU.

31 Sezgin, Mehmet & Sankur, Bulent. 2004. Survey over image thresholding techniques and quantitative performance evaluation. Journal of Electronic Imaging 13(1), p. 146–165.

32 Fisher, R. & Perkins, S. & Walker A. & Wolfart E. 2003. Internet source. HIPR2. <https://homepages.inf.ed.ac.uk/rbf/HIPR2/gryimage.htm>. Read 3.2.2018.

33 Bradski, Gary & Kaehler, Adria. 2008. Learning OpenCV. Sebastopol: O'Reilly Media, Inc.

34 Suzuki, Satoshi & Abe, Keiichi. 1985. Topological Structural Analysis of Digitized Binary Images by Border Following. Computer Vision Graphics and Image Processing 30 (1), p. 32–46.

35 Processes and Threads Overview. 2017. Internet source. Android Open Source Project. <https://developer.android.com/guide/components/processes-and-threads.html>. Read 29.3.2018.

36 android.hardware.camera2. 2017. Internet source. Android Open Source Project. <https developer.android.com/reference/android/hardware/camera2/package-summary.html>. Read 29.3.2018.

37 CameraCaptureSession. 2017. Internet source. Android Open Source Project. <https://developer.android.com/reference/android/hardware/camera2/CameraCaptureSession.html>. Read 28.3.2018.

38 Introduction into Android Development. 2014. Internet source. OpenCV documentation. <https://docs.opencv.org/2.4/doc/tutorials/introduction/android_binary_package/android_dev_intro.html>. Read 29.3.2018.

39 ADT Plugin (UNSUPPORTED). 2015. Internet source. Android Open Source Project. <https://developer.android.com/studio/tools/sdk/eclipse-adt.html>. Read 29.3.2018.

40 Android OpenCV Manager. 2014. Internet source. OpenCV documentation. <https://docs.opencv.org/2.4/platforms/android/service/doc/Intro.html>. Read 29.3.2018.

41 Building TensorFlow on Android. 2017. Internet source. TensorFlow. <https://www.tensorflow.org/mobile/android_build>. Read 29.3.2018.

42 Introduction to TensorFlow Lite. 2017. Internet source. TensorFlow. <https://www.tensorflow.org/mobile/tflite>. Read 29.3.2018.

43 Simple transfer learning with Inception v3 or MobileNet models. 2016. Internet source. TensorFlow. <https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/image_retraining/retrain.py>. Read 30.3.2018.

44 How to Retrain Inception's Final Layer for New Categories. 2016. Internet source. TensorFlow. <https://www.tensorflow.org/tutorials/image_retraining>. Read 31.3.2018.

45 Handwritten math symbols dataset, over 100 000 image samples. 2017. Internet source. Kaggle. <https://www.kaggle.com/xainano/handwrittenmathsymbols/version/1>. Read 29.1.2018.

46 Boulogne, François & Gouillart, Emmanuelle & Nunez-Iglesias, Juan & Schönberger, Johannes L. & van der Walt, Stéfan & Warner, Joshua D. & Yager, Neil & Yu, Tony & the scikit-image contributors. 2014. scikit-image: Image processing in Python. <http://dx.doi.org/10.7717/peerj.453>. Read 1.3.2018

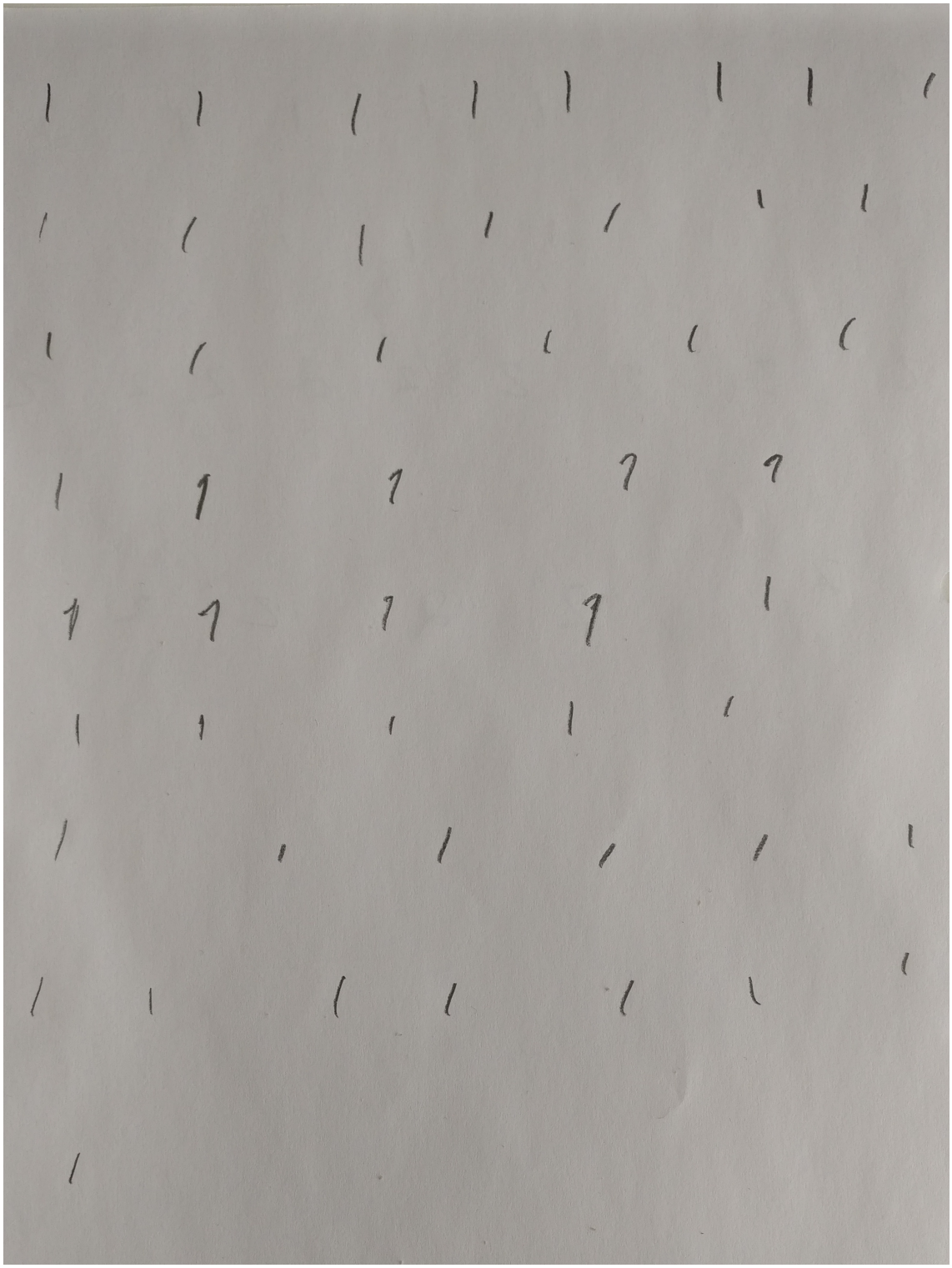
47 OpenCV documentation. 2018. Internet source. OpenCV dev team. <https://docs.opencv.org/2.4/index.html>. Read 11.2.2018.

48 Imgproc color conversions. 2015. Internet source. Open Source Computer Vision. <https://docs.opencv.org/3.1.0/de/d25/imgproc_color_conversions.html>. Read 4.2.2018.

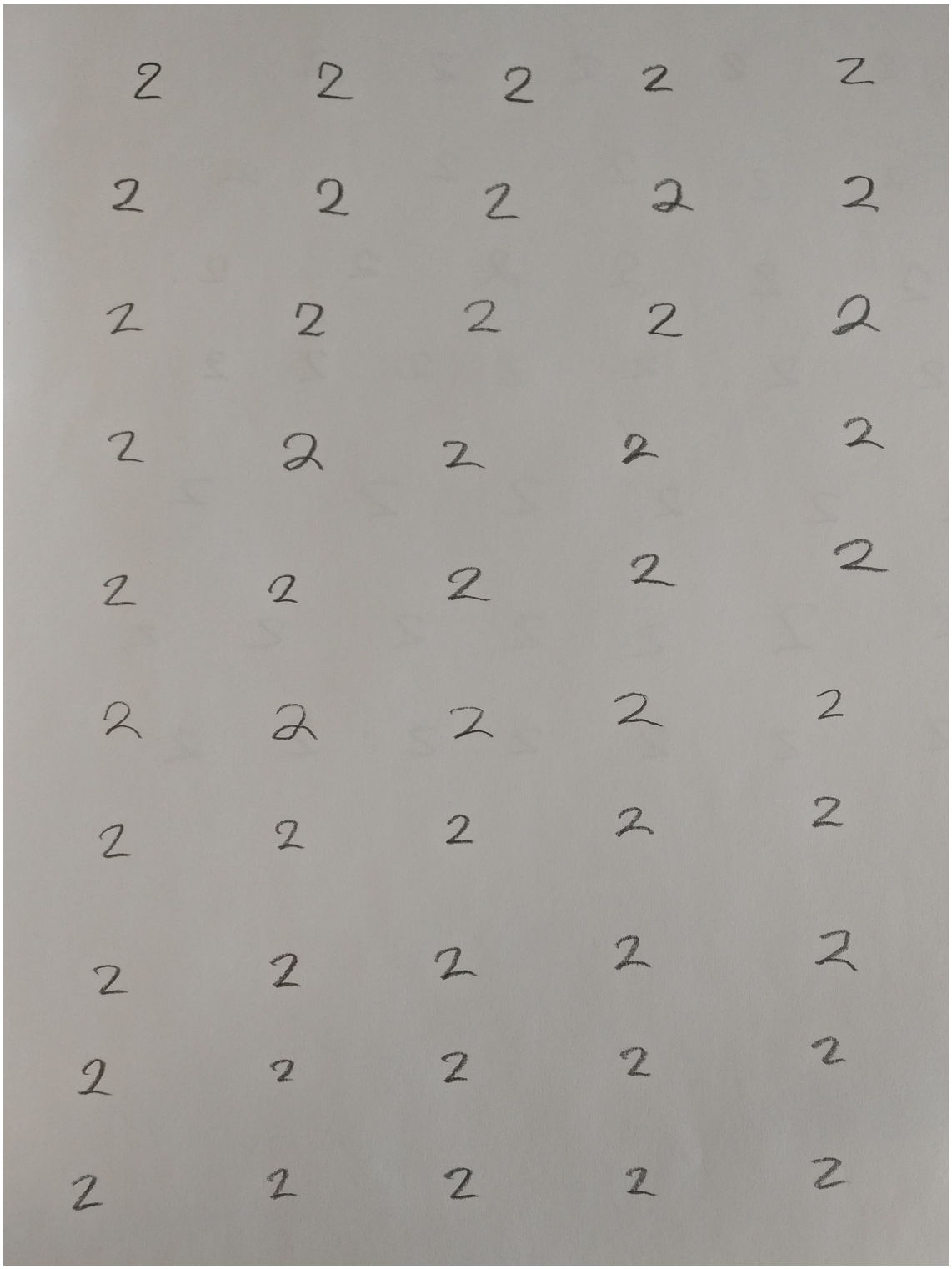
49 Bitmap.Config. 2017. Internet source. Android Open Source Project. <https://developer.android.com/reference/android/graphics/Bitmap.Config.html>. Read 4.2.2018.

50 Basic Thresholding Operations. 2018. Internet source. OpenCV dev team. < https://docs.opencv.org/2.4/doc/tutorials/imgproc/threshold/threshold.html>. Read 4.2.2018.

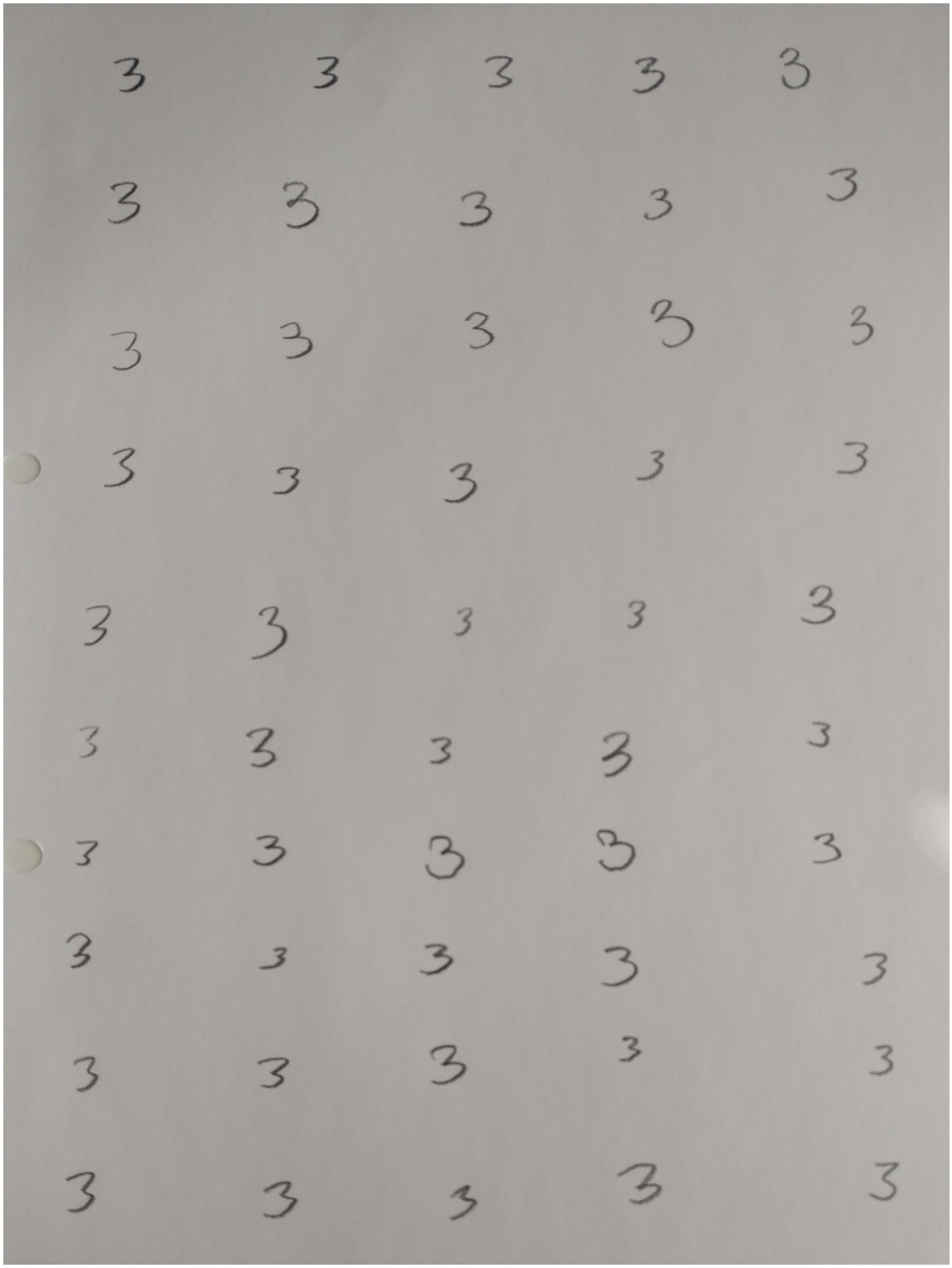
Evaluation characters for class 1



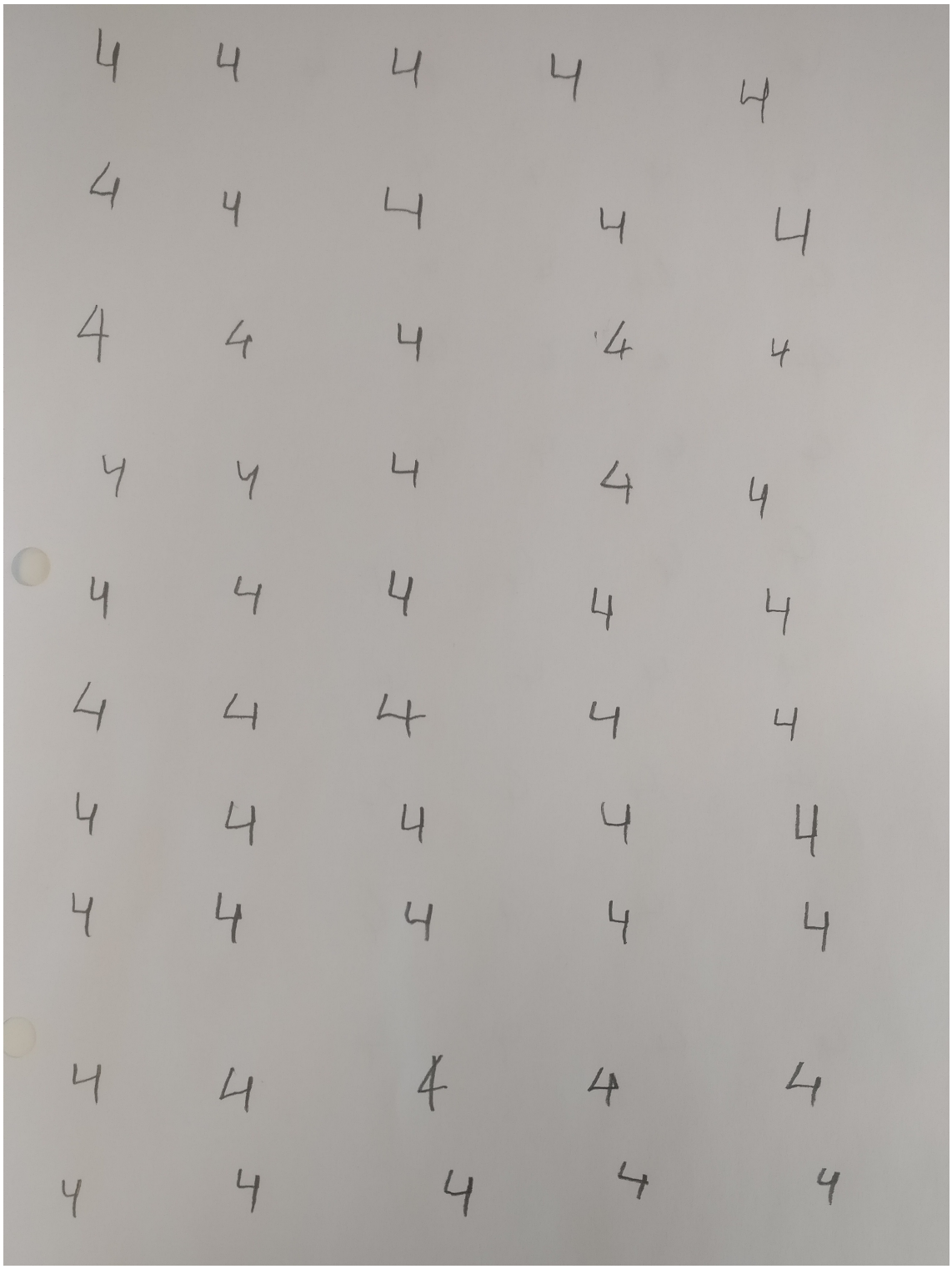
Evaluation characters for class 2



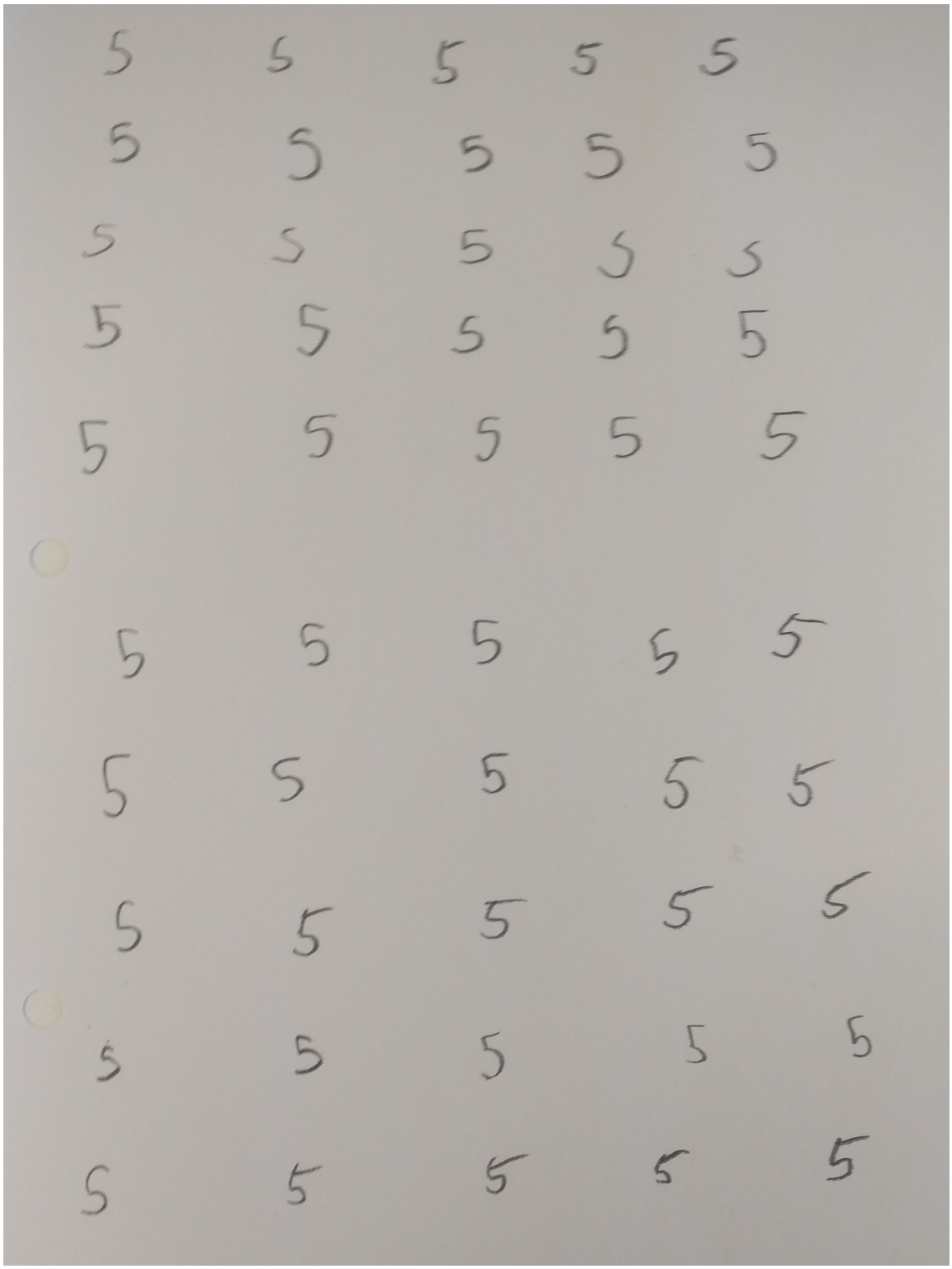
Evaluation characters for class 3



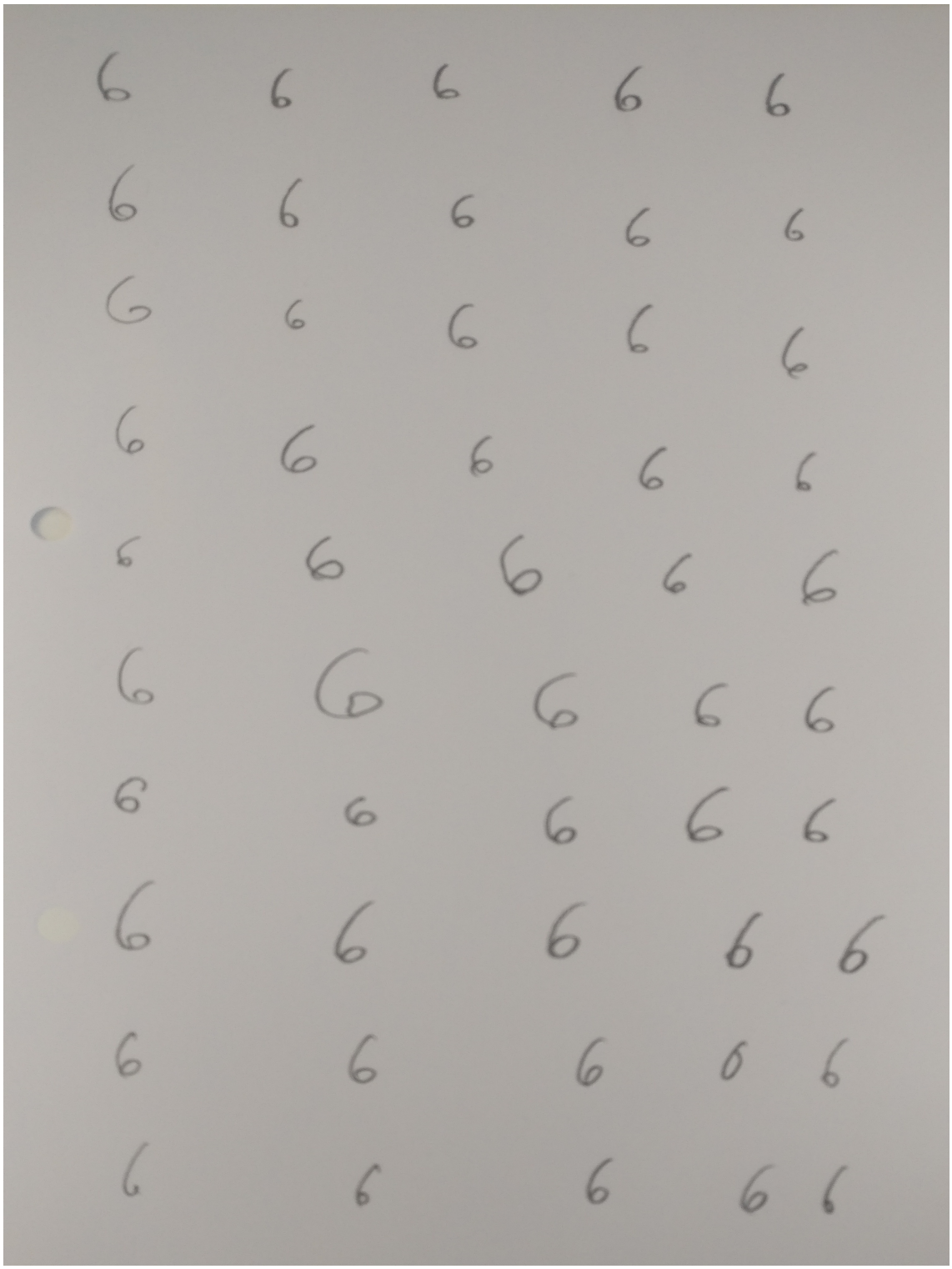
Evaluation characters for class 4



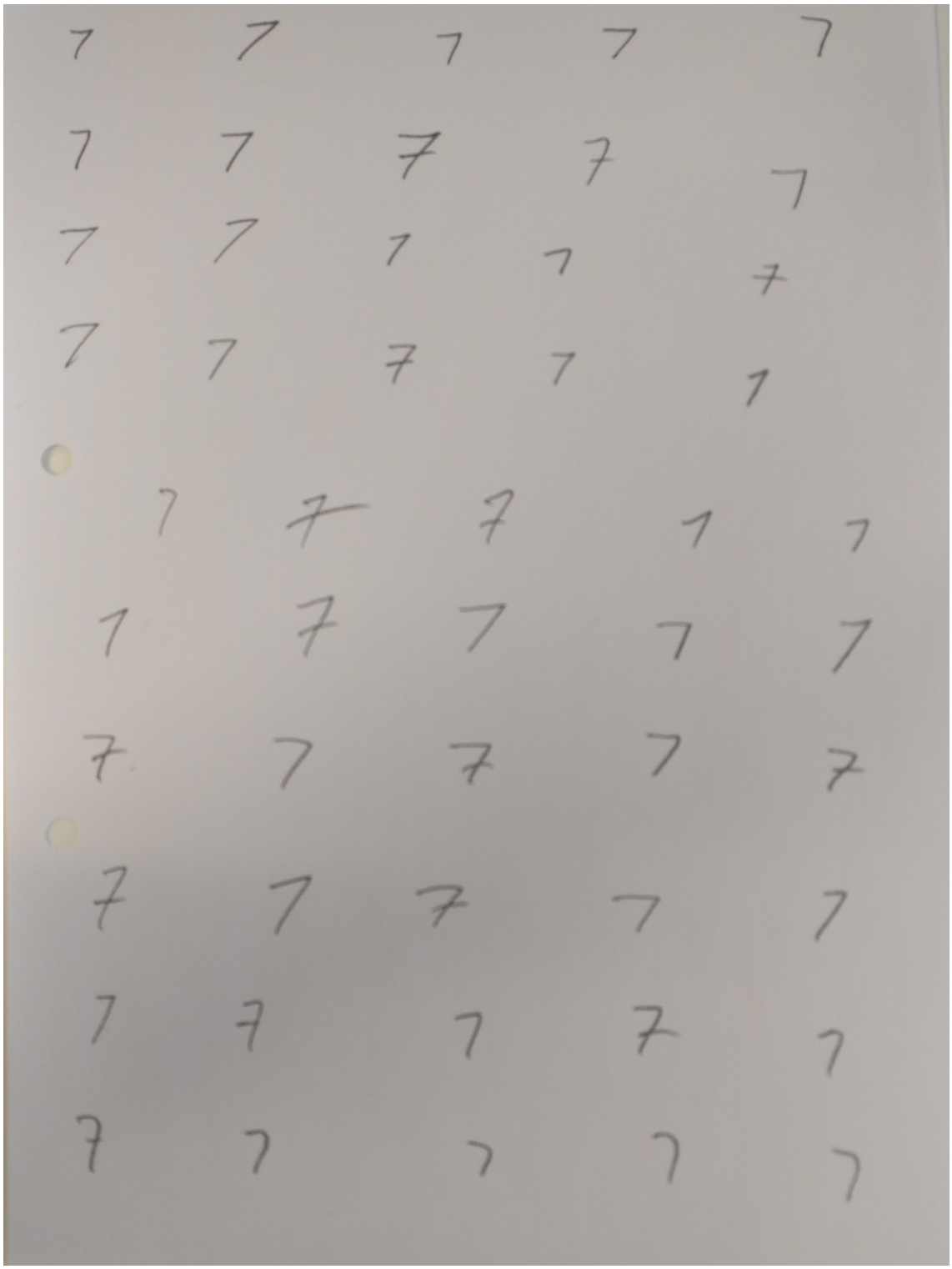
Evaluation characters for class 5



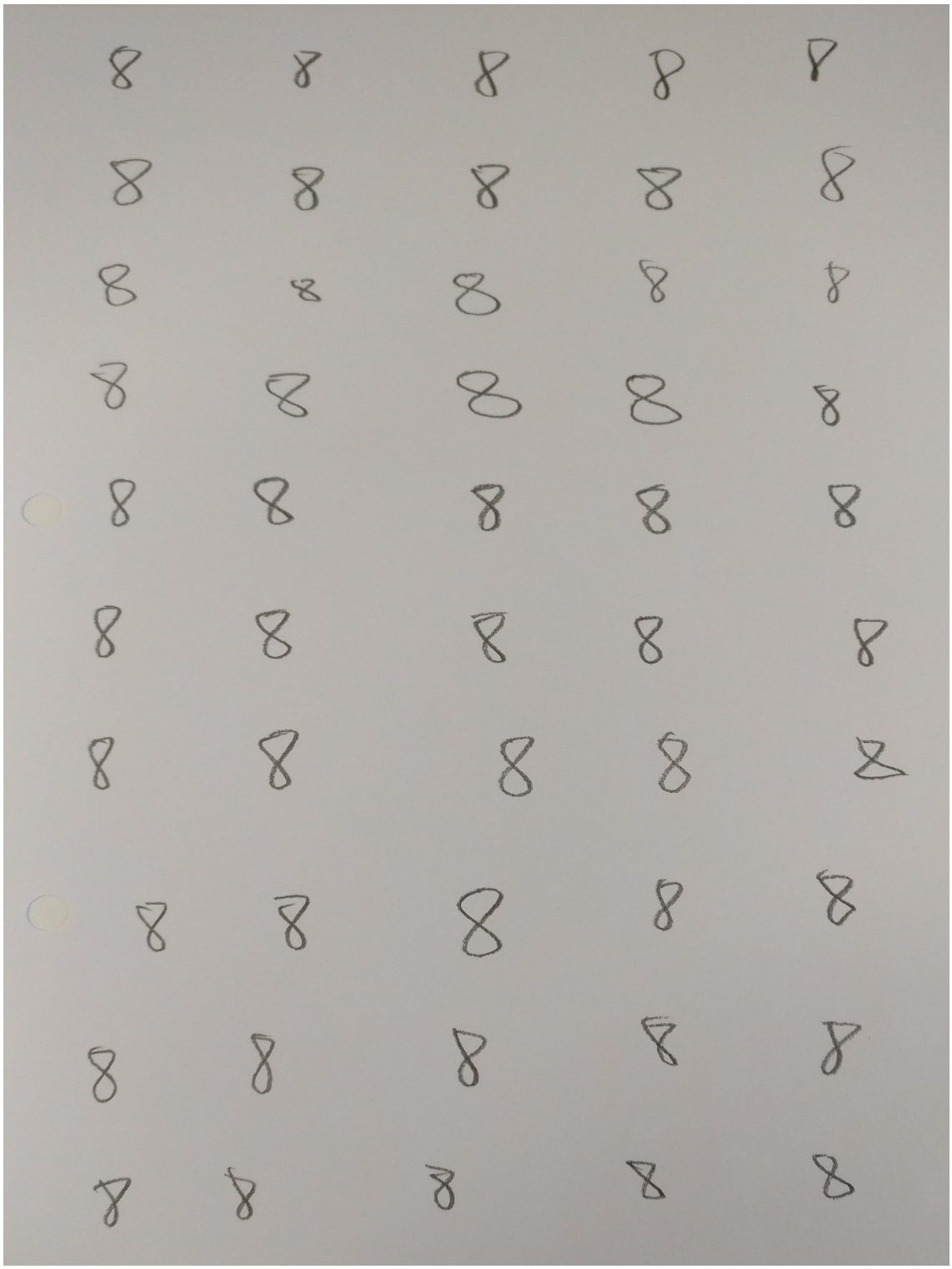
Evaluation characters for class 6



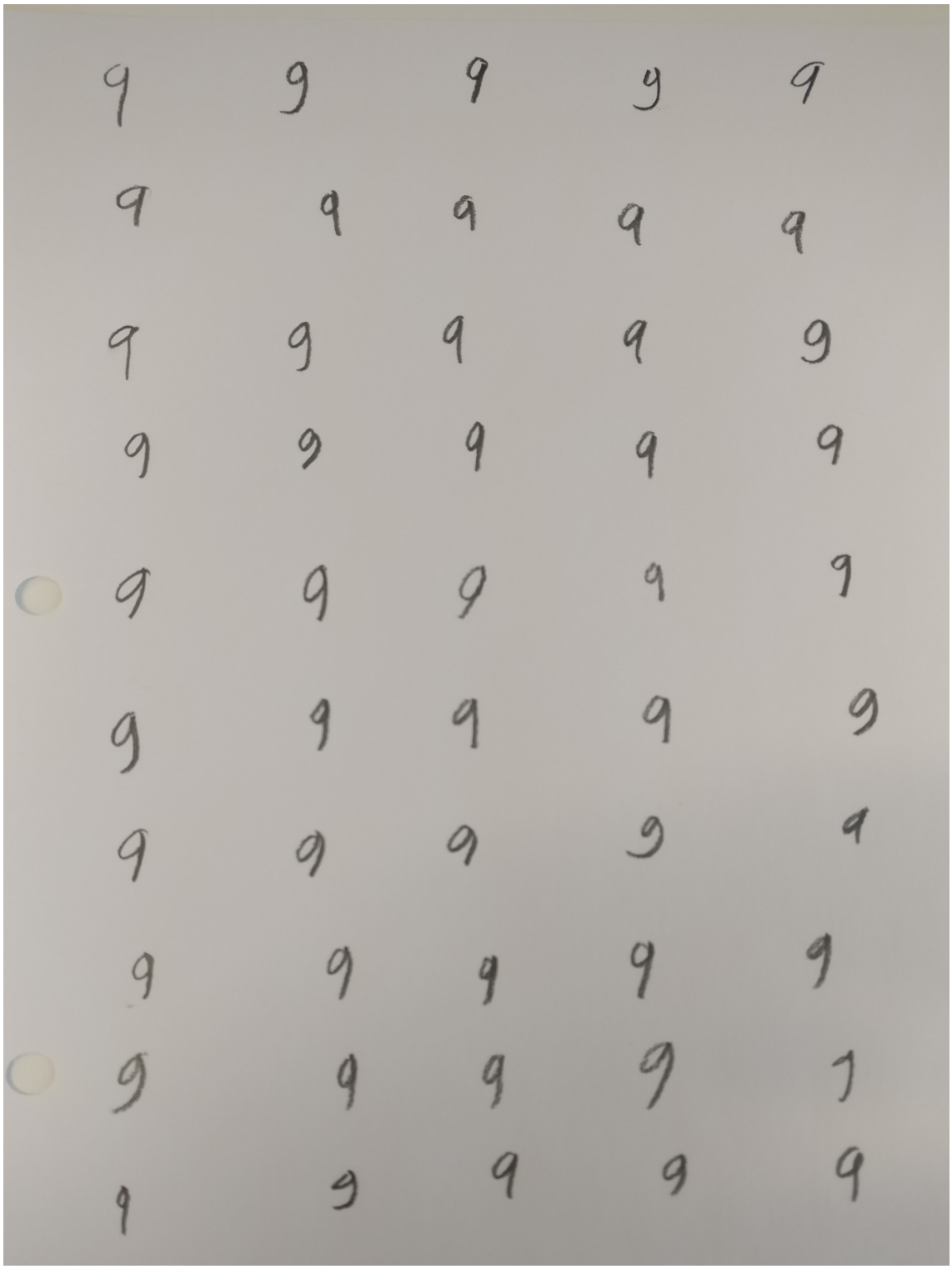
Evaluation characters for class 7



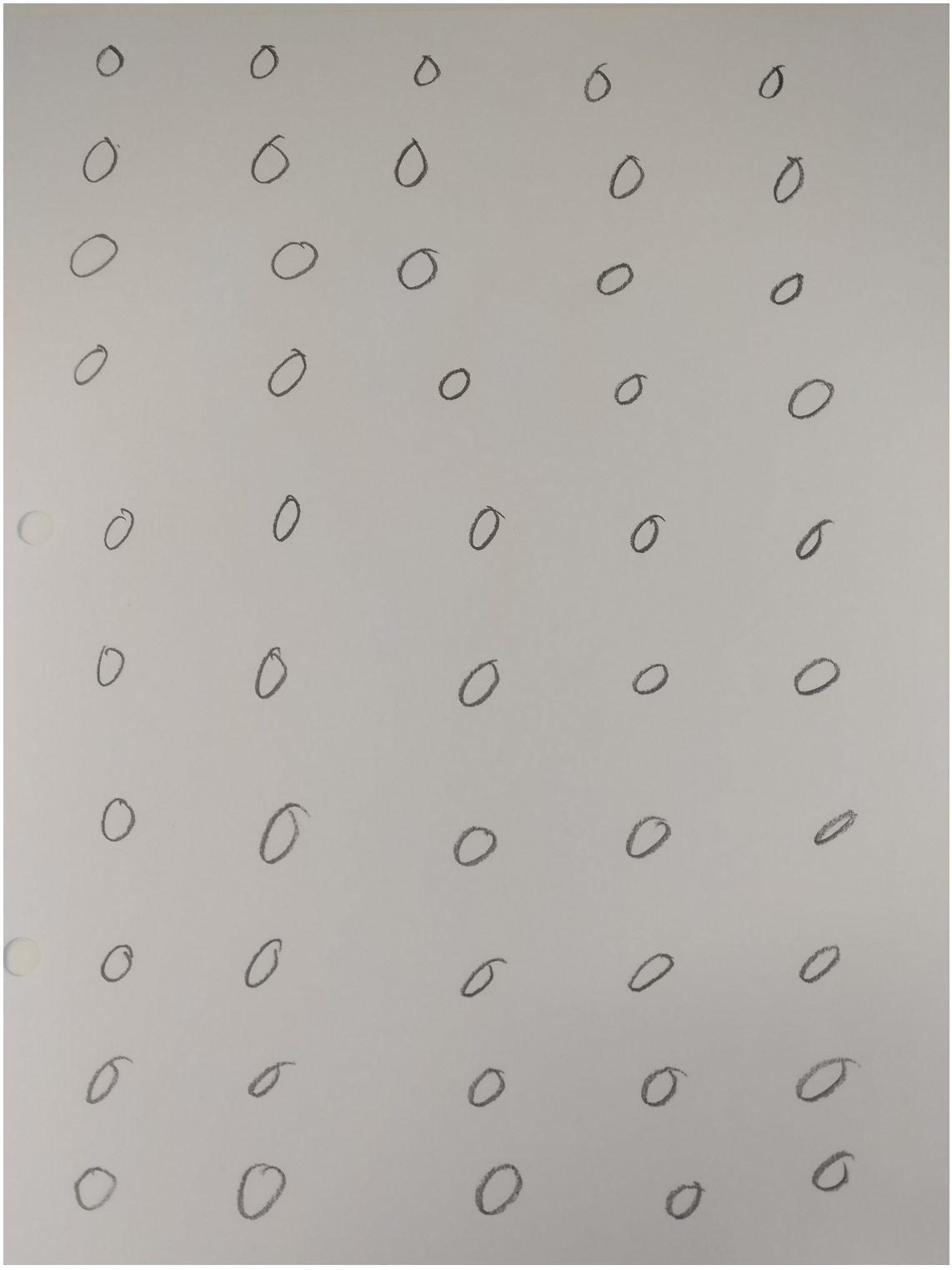
Evaluation characters for class 8



Evaluation characters for class 9



Evaluation characters for class 0



Evaluation characters for class +