Written characters recognition in Unity phone application

Martin Pajerský
PV026 - AI Project
Faculty of Informatics Masaryk University
June 17, 2024

1. INTRODUCTION

Character recognition is a well-established problem in the field of machine learning, extensively studied for decades with numerous effective solutions. This project aims to explore the task of recognizing handwritten both single letters and numbers - a challenge considered largely resolved by various machine learning methods that achieve high accuracy [1]. Specifically, the project was developed using Unity game engine and its default programming language, C#, to create an interactive interface. This interface allows users to draw a symbol on their phone screen, which is then recognized by a neural network. The inspiration for integrating this feature came during the development of a mobile game, envisioning neural networks as a potential minigame or a mechanism to enhance gameplay. Consequently, the project's purpose was not only to tackle a classic problem but also to establish a framework for implementing real-time, simple machine learning tasks on mobile devices. The neural network is designed to recognize a total of 36 classes, encompassing both the letters A-Z and the numbers 0-9, using well-recognized datasets.

2. SIMILAR PROJECTS

While Python and C++ are popular choices for implementing machine learning solutions, other programming languages, such as C#, do not enjoy the same breadth of usage and support and popularity. This section provides a concise review of existing character recognition solutions and machine learning libraries specifically designed for the Unity game engine and C#.

2.1. Unity Number Recognition project by tomislavrekic [2]

This project employs Unity and C# to create a neural network that recognizes handwritten digits from the MNIST dataset. Although the network is well-structured, it has limitations in terms of maintainability and extensibility. The architecture is hard-coded with fixed layer sizes and depth, limiting its adaptability to other tasks without significant modifications. This rigidity renders it less suitable for tasks that require different network topologies or extend beyond simple digit recognition.

2.2. Unity official ML package [3]

The ML-Agents library is primarily designed for creating and training intelligent agents in a Unity environment via simulations. It is well-suited for scenarios where agents interact with game-like environments, learning through trial and error using reinforcement learning. While it is conceivable to design a scenario in which an agent categorizes symbols within a simulated environment, using these symbols as part of the environmental state inputs, no such examples have been found online.

2.3. Microsoft ML.NET package [4]

The ML.NET library supports neural networks and provides a variety of other machine learning capabilities for creating custom ML models directly in .NET environments. Despite numerous forum threads inquiring about the possibility of integrating this library into Unity or addressing integration errors, no clear, working solution has been documented.

3. INSTALLATION AND STARTUP

Steps described in this chapter should help in setting up both the Android application and the Unity project.

3.1. Android application

Installing an “CharactersRecognition.apk” file on an Android device involves several steps. Here is a concise description of the process:

- Enabling Installation from Unknown Sources
  Access the settings menu of the Android device. Locate the option for installations from unknown sources. This setting is generally found under "Security" or "Applications" sections, though it may also appear as "Install unknown apps" within "Special access" settings depending on the device model and manufacturer. Activate this option to permit installations outside the Google Play Store.
- Transfer the APK File
  Connect the device to a computer via a USB cable. Copy the APK file from the computer to the device’s storage.
- Install the APK File
  Locate the APK file on your device’s file manager and select "Install”. Grant the necessary permissions when prompted to finalize the installation. Upon completion, the application can be accessed from the app drawer of the device.

3.2. Unity project

The application was developed using Unity version 2022.3.12f1. Newer versions should also be compatible. The project contains datasets in the Assets folder. To display scenes correctly, open the project in Unity and switch the platform to Android.

4. DATA

To address the task of recognizing both letters and digits, distinct datasets had to be merged to collectively include alphanumeric symbols. Additionally, we have collected our own set for the testing purposes. This chapter describes the process of the data collection and its analysis.

4.1. Downloaded training set

- A-Z Handwritten Alphabets (*A-Z*) [5]:
  This dataset consists of a single CSV file containing 28x28 pixel images of handwritten letters across 26 classes, totaling 372,451 data points.
- The MNIST database of handwritten digits (*0-9*) [6]:
  This well-known dataset contains 70,000 examples, also formatted into 28x28 images.

Additional datasets were considered but were not utilized due to either their small sample size or different image resolutions, which would have increased complexity of the preprocessing stage. We divided the dataset into standard training/validation and test set splits.

4.2. Data exploration

Initial analysis of the dataset revealed large variability in pen thickness and writing styles, with letters appearing both capitalized and lowercase. As depicted in Figure 1, class distribution is highly unbalanced, which illustrates the necessity of data augmentation strategies detailed later. We chose not to subsample the most populous classes (the letter “O” has up to 57,000 samples), but instead focused on augmenting the least represented ones to prevent the network from ignoring them.

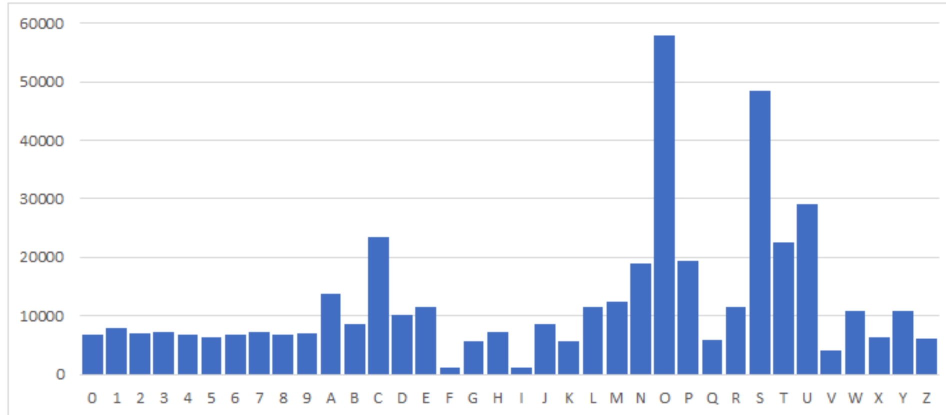

Figure 1. Class distribution on the downloaded training dataset.

4.3. Collected testing dataset

The goal of the application is to make predictions directly on the drawings on the smartphone screen. The most suitable way to test the performance is on the symbols drawn directly in that environment. We have collected 717 instances submitted by friends and used this dataset in the final evaluation. This dataset can be found alongside the downloaded ones in the Assets folder of the developed Unity application.

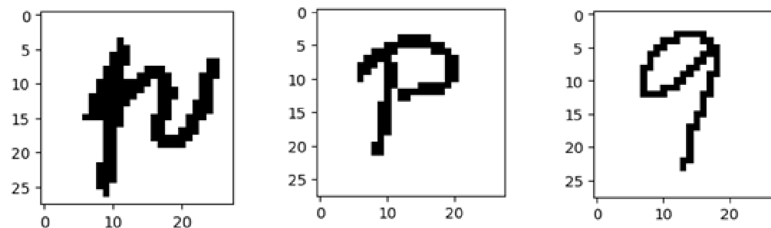

Figure 2. Example instances from the collected set

4.4. Preprocessing

The preprocessing steps were designed to address the challenges posed by integrating smartphone screen drawings with the training data. We have implemented multiple image transformation functions for this purpose.

The images in the datasets are grayscale, however in the smartphone inputs we do not know how "strong" the user pushed on the screen, the inputs are binary (0 or 1). First step in the data processing pipeline is thresholding, i.e. binarizing the input pixels of all data points.

Drawing on the screen also creates an image of arbitrary resolution. A fixed size to ensure compatibility with a network had to be chosen. In the description of the *A-Z* dataset it is stated that each image is center fitted to the 20x20 pixel box. Using such an image as input would leave almost half of the weights in the first layer inactive. Additionally, training data might be already centered, but users can freely draw in the corner of the plane, which poses a challenge to the simple MLP model and brings weak results. To address those roadblocks, each input image is first algorithmically centered based on the center of its non-zero pixels bounding box and then resized using nearest-neighbor interpolation, while preserving the original aspect ratio. The result is a 24x24 image, required by our MLP model. This transformation is also applied to user-drawn images to ensure compatibility.

4.5. Data augmentation

We developed a simple augmentation function that randomly rotates input images, creating two new data points per instance. This function was initially applied to underrepresented classes "F" and "I", and significantly enhanced their recognition by the model. Later, this approach was extended to classes where the model showed poor performance.

Other potential augmentation techniques, such as horizontal flips and shifts, were excluded as they could misrepresent the orientation of characters, and adding noise was deemed unnecessary given its absence in the input data.

5. IMPLEMENTATION AND FINAL APPLICATION

Three primary "Scenes" were created in the Unity engine. The scene used for training the network is only functional within the editor. The Drawing scene was utilized for data collection and testing methods for drawing and image transformations. Lastly, the Inference scene is included in the resulting APK file and is employed to make predictions based on user drawings. Screenshots of the final scenes are displayed in Figure 3. This chapter describes these scenes in depth as well as some implementation details.

Figure 3. Three final scenes created in Unity. Draw Scene on the left, Training in the middle and Inference on the right side.

5.1. Draw Scene

The main purpose of this scene was the collection of additional data samples, as described in Section 1.3. Users were presented with a "task" where a symbol that we wanted them to draw was displayed. Pen thickness is not adjustable in this scene and was randomly generated to ensure variability in the collected data. Two buttons are present: "Clear," to erase the drawing, and "Submit," to send the sketch to the database. The database was hosted remotely on a free version of endora.cz. Data was subsequently downloaded and processed/explored using a simple Python script.

The core part of the implementation is the ability to draw a single line. This functionality is handled in the Draw.cs script by connecting the previous position of the input (Raycast on the user's finger in this case) to the last recorded point on the screen. The actual image is captured from a fixed-size plane at the center of the screen, and a simple image processing procedure concludes the process.

For testing purposes, the class DisplayData.cs was used to visually verify that the data was being correctly captured from the screen and had the same format as the training data. After validation, there was no longer a need to use this class.

5.2. Training Scene

All training in this application was conducted inside the editor, and this scene will not function properly if built for a mobile device. It is theoretically possible to train the entire network on a mobile device, but this would likely be time-consuming and could risk other problems such as overheating; thus, this option was not explored. Initially, data must be loaded, with functionality supporting the loading and concatenation of multiple datasets simultaneously. The training procedure can then be executed. No adjustments can be made in the scene; all configurations must be edited directly in the code.

Additional functionality of this scene is for testing on external dataset, supporting different csv files and models, loaded by their file paths.

5.2.1. Network implementation

The implementation of the network is straightforward, not dependent on any external libraries, yet it is extendable, and new functionality can be added. The Network.cs class contains the definition of the Layer class, which handles both the computation and the state, among all other necessary methods. Let's describe the chosen architecture in a bit more detail. Weights and biases are initialized using the He method. Activation functions used are ReLU for hidden layers and Sigmoid for output layers. Forward and backward passes are calculated in batches, using the SGD learning algorithm and MSE loss function. Individual functions contain parallelization within, but a more robust solution might be desirable to achieve faster performance. Also, evaluation utilities are implemented, including loss calculation and accuracy assessments with a confusion matrix for deeper insights.

An important feature is the ability to save and load model parameters in JSON format, allowing for model reuse without retraining. A data management save method is triggered by a callback system to monitor validation loss, saving the model when a new best performance is achieved.

5.2.2. Training loop

A standalone class for the symbol recognition task was created. Here, all of the network's hyperparameters are defined, followed by the data pipeline, the training loop itself, and finally, the evaluation. The final model uses two layers with 400 hidden neurons. The process spanned 50 epochs, although the validation loss plateaued at around 25 epochs, as displayed in Figure 4.

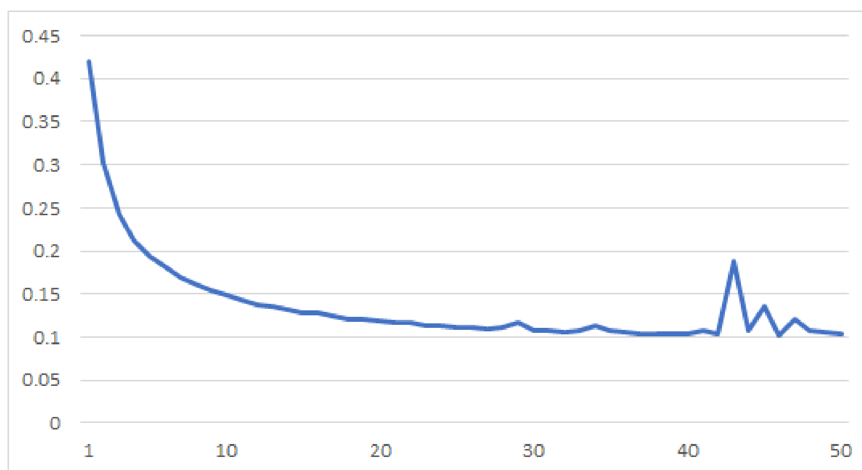

Figure 4. Validation loss during the training process

5.3. Inference scene

This is the final scene that is also built into the .apk file for Android devices. The scene contains a plane where users can draw, as described in the Draw Scene section. The thickness of the pen can be adjusted by a slider. The model is automatically loaded at startup, so whenever the user is satisfied with their drawing, they can click the "Predict" button to start the evaluation. After a second, the predictions are displayed, showing the five most probable drawing symbol classes predicted by the network. From an implementation perspective, all functionalities are reused from the previously mentioned workflows, namely, the network is loaded from storage, its forward pass is called, and the Draw.cs script is used again for user input.

6. RESULTS AND EVALUATION

6.1. Test split

The model's performance was assessed after completing training and tuning processes, which relied on the validation dataset. The test split, constituting 10% of all data samples, resulted in a total accuracy of 41,536 correct classifications out of 44,246, equating to 93.89%. Class-wise accuracies are detailed in Figure 5, with most symbols achieving recognition rates of 90% or higher. However, the digit "0" was an outlier, with a notably low accuracy of only 22%.

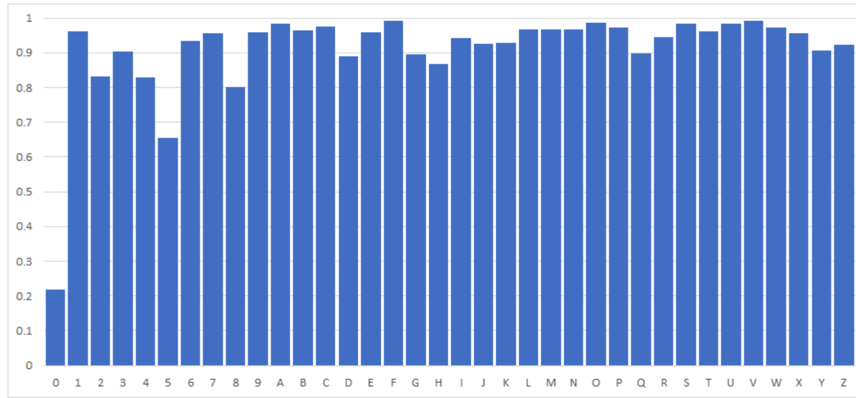

Figure 5. Class accuracies on the test data

A confusion matrix provided in Figure 6 illustrates the misclassification patterns. The x-axis identifies which symbols got misclassified into the classes represented by the columns. Similar-looking symbols posed the greatest challenge; for example, "Zero" was frequently misidentified as "O," explaining its low accuracy. Other problematic pairs included "5" and "S," as well as "2" and "Z." Due to class imbalances, characters such as "S" and "O" dominated, leading to frequent false predictions towards them. Additionally, there is no clear explanation for why symbols like "9" or "B" were common misclassifications.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
0	153	0	0	1	0	0	0	1	0	1	1	5	0	8	0	0	2	0	0	0	0	0	0	1	522	0	2	0	0	0	3	0	1	0	0	0
1	0	798	1	1	0	0	1	1	1	2	0	0	0	4	0	0	0	0	8	1	0	4	0	0	0	2	0	0	2	0	0	1	0	1	2	0
2	0	0	574	2	1	0	0	7	1	0	1	3	4	2	2	0	2	0	2	1	0	7	0	0	1	0	5	5	4	0	3	0	0	1	1	60
3	0	0	2	647	0	4	0	6	3	12	0	14	0	2	0	0	1	0	4	3	0	0	0	0	0	0	0	0	16	0	1	0	0	1	0	1
4	0	0	0	0	577	0	4	2	0	34	9	1	0	0	0	0	0	9	0	0	0	0	1	4	0	0	1	1	0	1	19	2	2	0	29	0
5	2	0	2	6	0	409	2	1	3	6	0	4	1	0	6	2	0	0	0	4	0	0	0	0	1	0	0	0	172	3	0	0	0	0	0	0
6	1	0	0	0	1	0	670	0	0	0	0	5	2	0	3	0	17	0	0	0	0	7	0	0	1	0	0	0	4	0	3	0	3	1	0	0
7	1	0	2	0	0	0	0	692	0	17	1	0	0	0	0	1	0	0	0	1	0	0	0	0	0	1	0	0	0	5	0	0	1	0	1	1
8	1	2	0	3	0	1	1	3	546	24	1	55	0	2	5	1	0	0	2	2	2	0	0	0	4	6	0	4	7	0	0	1	2	5	1	0
9	0	1	1	1	2	0	0	11	1	614	1	0	0	0	0	0	0	0	0	0	0	0	2	0	1	1	1	0	2	0	1	0	0	0	0	0
A	0	1	0	0	1	0	0	0	0	2	1416	2	0	0	0	0	0	1	0	1	0	1	4	1	0	0	1	2	0	0	1	0	4	1	0	1
B	0	0	0	1	0	1	0	0	2	4	5	866	1	0	4	0	0	0	0	0	0	0	0	1	9	2	0	1	0	0	0	0	0	0	0	1
C	1	0	0	0	1	1	3	0	0	0	1	5	2368	0	10	0	2	0	0	0	0	12	0	0	13	3	0	0	1	2	4	0	1	0	0	3
D	4	0	2	3	0	0	0	1	0	0	2	12	0	904	0	0	0	0	1	1	0	0	1	1	71	6	0	0	1	0	3	0	1	0	0	1
E	0	0	2	0	0	0	6	0	1	2	2	5	12	1	1076	0	0	0	1	0	1	0	1	1	0	2	0	2	1	0	2	0	4	0	0	1
F	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	108	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	2	0	19	0	0	3	0	8	8	0	3	2	492	0	0	2	0	0	1	0	3	0	0	0	4	1	1	0	1	0	0	0
H	0	1	2	0	11	0	0	4	0	2	17	1	0	0	2	0	0	623	0	0	0	0	8	26	0	2	0	1	3	0	5	0	9	0	0	0
I	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	116	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
J	0	3	2	1	0	5	0	4	0	0	0	0	0	2	0	0	0	0	6	809	0	0	3	0	0	0	0	0	19	11	4	1	2	0	1	0
K	0	1	0	0	0	1	0	0	2	1	1	0	2	0	1	0	0	2	0	0	532	0	1	7	0	1	0	9	2	0	3	0	3	4	0	0
L	0	8	1	0	0	0	4	0	0	0	0	0	9	1	1	0	0	1	0	1	1	1072	0	0	0	0	0	0	3	0	1	0	0	3	0	2
M	0	0	0	0	0	0	0	0	0	2	8	6	0	0	0	0	1	3	0	0	1	0	1204	7	1	0	0	3	0	0	2	0	6	0	0	0
N	0	1	0	0	1	0	0	0	0	0	11	0	0	1	0	0	0	4	0	4	0	0	3	1811	0	0	1	0	0	1	7	0	22	5	2	0
O	33	0	1	0	0	0	0	0	0	4	0	3	2	19	1	1	0	0	0	0	0	0	2	0	5615	0	2	0	0	0	5	0	2	0	0	0
P	0	1	0	0	0	0	1	9	0	14	7	1	0	7	1	2	0	0	0	0	0	0	1	2	0	1875	1	0	0	4	0	0	0	0	0	0
Q	1	0	1	0	0	0	0	2	0	26	1	1	0	1	1	0	5	0	0	0	0	0	0	1	14	2	529	2	0	0	0	0	2	0	0	0
R	0	0	3	1	1	0	0	1	0	5	21	6	5	2	4	0	1	0	0	0	3	1	3	0	0	2	3	1126	0	0	1	0	2	1	0	0
S	0	0	0	5	0	15	1	2	2	6	2	7	4	0	5	1	5	0	2	10	0	0	0	0	7	0	0	0	4737	4	0	0	3	0	0	0
T	0	1	0	0	16	0	0	44	0	3	0	1	0	0	1	2	0	2	0	1	1	0	6	0	0	0	0	0	0	2080	0	0	1	0	2	3
U	1	0	0	0	3	0	0	0	0	1	0	2	1	3	1	0	0	0	0	1	0	0	1	2	4	0	1	3	0	0	2875	2	23	2	0	0
V	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	443	1	0	1	0
W	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	1	0	0	0	0	1	0	0	18	0	0	0	0	0	0	4	3	1021	0	0	0
X	0	0	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	9	0	0	0	0	0	1	1	0	1	1	0	1	566	9	0
Y	0	7	1	0	29	1	1	13	0	20	0	0	0	0	1	0	0	0	1	4	3	0	0	0	1	1	0	0	0	3	5	5	1	8	1012	1
Z	0	0	21	2	1	0	0	5	1	2	0	0	1	0	4	0	1	0	1	0	0	2	0	1	0	0	0	1	0	0	0	0	1	4	0	580

Figure 6. Confusion matrix for the test data

6.2. Collected data set

The final model correctly classified 503 out of 717 collected samples, corresponding to an accuracy of 70.15%, which is significantly lower than that observed in the test set. During the data collection phase, participants were instructed to draw symbols in any style, not limited to simple capital block letters. This led to numerous cursive entries, which the model struggled to recognize due to the insufficient representation of such styles in the training data. The examples of this issue can be observed in Figure 7, which includes multiple cursive letters. Moreover, the variance in pen thickness was broad, with extreme values not previously encountered in the dataset, such as the thick pen line used for the "0" with index 673 in the Figure. This experience underlines the importance of an in-depth investigation into the dataset to set appropriate conditions for experiments. Failing to do so initially can lead to a hard and lengthy process of fixing mistakes later.

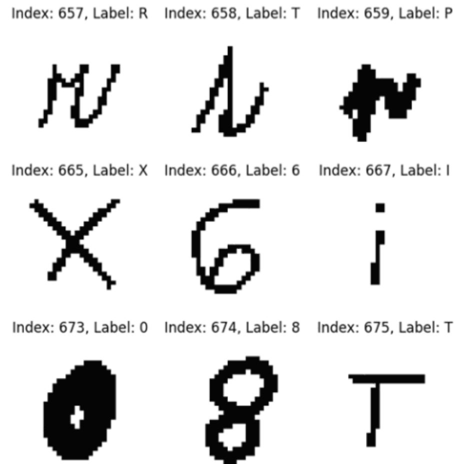

Figure 7. Sample form the collected data

References

[1] http://yann.lecun.com/exdb/mnist/

[2] https://github.com/tomislavrekic/Unity-Number-Recognition

[3] https://docs.unity3d.com/Packages/com.unity.ml-agents@2.0/manual/index.html

[4] https://learn.microsoft.com/en-us/dotnet/machine-learning/how-does-mldotnet-work

[5] https://www.kaggle.com/datasets/sachinpatel21/az-handwritten-alphabets-in-csv-format/data

[6] https://www.kaggle.com/datasets/hojjatk/mnist-dataset