Universitat Politècnica de Catalunya

Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

A Deep Learning Based Approach to Automated App Testing

David Llàcer Giner

Advisors: Rubèn Tous Liesa, Xavier Guàrdia Latorre

A thesis submitted in fulfillment of the requirements for the Master in Telecommunications Engineering

Barcelona, September 2020

Abstract

Mobile applications are worldwide extended. We use them for everything, from texting friends to managing our money. This boom has led to the emergence of companies dedicated exclusively to the development of mobile applications. Also, the mobile industry, made up by millions of apps and billions of users, has been growing at an unprecedented speed and has been incredibly successful.

One of the most important steps in the process of application development is testing. Once a first version of the application is released, the testing team is in charge of verifying its correct functioning and that it meets all the requirements set by the design team. Normally, tests are based on following a flow through the app. A person navigates through the different screens tapping buttons and checking everything works fine. There are some tools and software that automates this process, but they use brute force and random algorithms, that lead to inefficient testing.

The objective of this project is to design and implement a prototype of an artificial intelligence able to navigate through a mobile application mimicking the behavior of a real user. The burgeoning of deep learning and neural networks allows this kind of tasks to be learned by machines from experience.

The system consists of a deep learning architecture able to predict where to tap on the screen and which type of action to perform at this location. It consists of a Convolutional Neural Network to encode the images and recognize elements on the screen, and a Long Short-Term Memory to learn where to tap based on previous screens. Then, we have 2 outputs, one generated by a Deconvolutional Neural Network that predicts the location of the tap, and a Linear Neural Network that predicts the action type.

Overall, this work presents how mobile application testing can be automated using deep learning. Moreover, it shows the design and training process of a model to perform this task.

Key words: Deep learning, App testing.

Resum

Les aplicacions mòbils estàn àmpliament esteses pel món. Les fem servir per a tot, des d'enviar missatges als amics fins andministrar els nostres diners. Aquest auge a conduït a l'aparició de companyies dedicades exclusivament al desenvolupament d'aplicacions. També, la indústria mòbil, formada per milions d'aplicacions i bilions d'usuaris, ha estat creixent a una velocitat sense precedents i ha aconseguit gran èxit.

Un dels passos més importants en el procés de desenvolupament d'aplicacions és el *testing*. Un cop la primera versió de l'aplicació és llançada, l'equip de *testing* s'ha de fer càrrec de verificar el seu correcte funcionament i de que complisca els requisits demandats per l'equip de disseny. Normalment, els tests estàn basats en seguir un fluxe a través de l'aplicació. Una persona navega a través de diferents pantalles prement botons i comprovant que tot funcione correctament. N'hi han algunes ferramentes i *softwares* que automatitzen aquest procés, però utilitzen algoritmes de força bruta o aleatoris que comporten un *testing* ineficient.

L'objectiu d'aquest projecte és implementar una sol·lució capaç de navegar a través d'una aplicació mòbil imitant el comportament d'un usuari real. El creixement de l'aprenentatge profund i les xarxes neuronals permet que aquest tipus de tasques puguen ser apreses per una màquina gràcies a l'experiència.

El sistema està format per una arquitectura d'aprenentatge profund capaç de predir on prémer en la pantalla i quin tipus de gest realitzar en aquesta localització. Està format per una xarxa neuronal Convolucional per a extraure característiques de les pantalles i reconéixer elements en ella, seguida d'una part LSTM per a aprendre informació sequencial, és a dir, on prémer basant-se en interaccions anteriors. Per últim, la xarxa té dos ixides, una generada per una xarxa Deconvolucional que prediu la localització de la pulsació., i l'altra generada per una xarxa linial que prediu el tipus de gest.

En general, aquest treball presenta com el *testing* d'aplicacions mòbils pot ser automatitzat fent servir aprenentatge profund. A més, demostra el disseny i el procés d'entrenament de model per dur a terme aquesta tasca.

Paraules clau: Aprenentatge profund, *Testing* d'aplicacions.

Resumen

Las aplicaciones móviles están ampliamente extendidas en el mundo. Las utilizamos para todo, desde enviar mensajes a amigos hasta administrar nuestro dinero. Este auge ha llevado a la aparición de compañías dedicadas exclusivamente al desarrollo de aplicaciones móviles. También, la industria móvil, formada por millones de aplicaciones y billones de usuarios, ha estado creciendo a una velocidad sin precedentes y ha conseguido un gran éxito.

Uno de los pasos más importantes en el proceso de desarrollo de aplicaciones es el testeo. Una vez la primera versión de la aplicación es lanzada, el equipo de testeo está a cargo de verificar su correcto funcionamiento y de que cumpla todos los requisitos demandados por el equipo de diseño. Normalmente, los tests están basados en seguir un flujo a través de la aplicación. Una persona navega a través de diferentes pantallas pulsando botones y comprobando que todo funcione como es debido. Hay algunas herramientas y software que automatiza este proceso, pero utilizan algoritmos de fuerza bruta o aleatorios que conllevan a un testeo ineficiente.

El objetivo de este proyecto es implementar una solución capaz de navegar a través de una aplicación móvil imitando el comportamiento de un usuario real. El crecimiento del aprendizaje profundo y las redes neurales permite que este tipo de tareas puedan ser aprendidas por una máquina gracias a la experiencia.

El sistema está formado por una arcquitectura de aprendizaje profundo capaz de predecir dónde pulsar en la pantalla y qué tipo de gesto realizar en esa localización. Está formado por una red neuronal Convolucional para extraer características de las pantallas y reconocer elementos en ellas, seguida de una parte LSTM para aprender informacióin secuencial, es decir, dónde pulsar basándose en interacciones previas. Por último, la red tiene dos salidas, una generada por una red Deconvolucional que predice la locaclización de la pulsación, y la otra generada por una parte Lineal que predice el tipo de gesto.

En general, este trabajo presenta como el testeo de aplicaciones móviles puede ser automatizado utilizando aprendizaje profundo. Además, demuestra el diseño y el proceso de entrenamiento del modelo para llevar a cabo esta tarea.

Palabras clave: Aprendizaje profundo, Testeo de aplicaciones.

Acknowledgements

First of all, I want to thank my advisor Rubèn Tous for guiding me during the project development, helping me, solving my questions and providing me with needed resources.

This project emerged from *Soft For You* (SFY), a software development company, and they trusted me to lead it and develop it from scratch. This thesis proposal was presented to give an answer to the question: "Can we develop something to test mobile apps and find errors using artificial intelligence, in order to ease and optimize the testing process?", so this thesis is the first step of the trip. I am very grateful with SFY, Joaquín Custodio and Xavier Guàrdia for giving me this opportunity.

There are many people who have contributed to this thesis through their professional advice and work, such as Jaume Corbí, iOS developer at SFY, and Alfons González, Android developer at SFY. Thank you all!

Revision history and approval record

Revision	Date	Purpose
0	02/07/2020	Document creation
1	25/08/2020	Document revision
2	31/08/2020	Document approbation

DOCUMENT DISTRIBUTION LIST

Name	e-mail
David Llàcer Giner	david.llacer@estudiant.upc.edu
Rubèn Tous Liesa	rtous@ac.upc.edu
Xavier Guàrdia Latorre	mmoran@sfy.com

Written by:		Reviewed and approved by:		Reviewed and approved by:	
Date	31/08/2020	**Date**	31/08/2020	**Date**	30/08/2020
Name	David Llàcer Giner	**Name**	Rubèn Tous Liesa	**Name**	Xavier Guàrdia Latorre
Position	Project Author	**Position**	Project Supervisor	**Position**	Project Supervisor

Contents

List of Figures

List of Abbreviations

AI	**A**rtificial **I**ntelligence
App	Mobile **App**lication
UI	**U**ser **I**nterface
GUI	**G**raphic **U**ser **I**nterface
NN	**N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
RNN	**R**ecurrent **N**eural **N**etwork
LSTM	**L**ong **S**hort-**T**erm **M**emory
GPU	**G**raphics **P**rocessing **U**nit
MCA	**M**ódulo **C**omún **A**plicativo
QA	**Q**uality **A**ssurance

Chapter 1

Introduction

1.1 Motivation

Mobile applications have become one of the first consumer goods in today's society. There are different types and they offer different services, from communication to social media, going through entertainment, video games, fitness to bank account management, among others. To develop these apps it is usual to follow a common path, normally it begins with an idea that is presented to the design team, who is in charge of designing both the user experience and the user interface architectures. Then, with a design, we can start developing making this design real. There used to be two teams in charge of this step: front-end and back-end. Front-end team develops the user interface and user experience part, while back-end team takes care of databases, API and server issues. Once we release a first version of our app, we have to test it to check that everything works fine and it follows the requirements of the established design. To take care of this task, there is the testing team or Quality Assurance (QA) team. They have to use the app as if they were users and verify that the user experience is correct and there are neither bugs nor errors, and if there are, report them to the development team. Due to the fast releasing cycle and limited human resources, it is difficult to manually create test cases in a short period of time. As a result, automated test input generators have been extensively developed. There are some tools or software that already allow us to test apps automatically such as [1], Monkey [12], DynoDroid [27], Espresso [2] and more.

The key to success for an automated test is to choose the correct interaction for a given UI, being able to reach new relevant UI states. But choosing the correct button to click or do a scroll and understanding the GUI can be very difficult for a machine. That is why most of automated test generators [3], [6], [12], [27] ignore different types of GUI elements and just apply random strategy or brute force to choose the one to interact with. In this way, human testers are way better that automated test input generators, because they can easily identify GUI elements that are worth interacting with.

This research project is commissioned by a leading company in the banking sector in Spain, that is interested in speed up and improve its applications by improving the testing process applying new technologies, in this case deep learning and artificial intelligence. On the other hand, although this is a private project, it can have repercussion and make contributions to the community. In this work, we design and train a prototype of a deep learning model that tries to mimic how humans interact with mobile apps.

1.2 Goals

The main goal of this work is to start the development of a system for automatic test input generation for mobile applications. We want to create a first prototype to study the projection that deep learning applied to input test generation can have and the possibilities that this kind of model can offer to us.

1.3 Hardware and Software Resources

This project was developed using the *Asterix* server, that is part of the BSC-UPC NVIDIA GPU Center of Excellence at Universitat Politècnica de Catalunya (UPC). The server consists on two nodes. Each node has 12GB of RAM, four Intel(R) Xeon(R) E5620 @2.4 GHz 4-core processors and 4 NVidia Tesla K40c GPUs (12GB memory and 2880 cores each). We also used Jupyter Notebooks at Google Colab.

The algorithm was implemented with Pytorch[1] version 1.1.0, using CUDA 9.0 and cuDNN to use GPU acceleration. We chose this framework because it is a well known Python-based open source deep learning library. For the image processing and database creation, we used OpenCV[2], an open source computer vision library widely used in image processing projects.

1.4 Work Plan

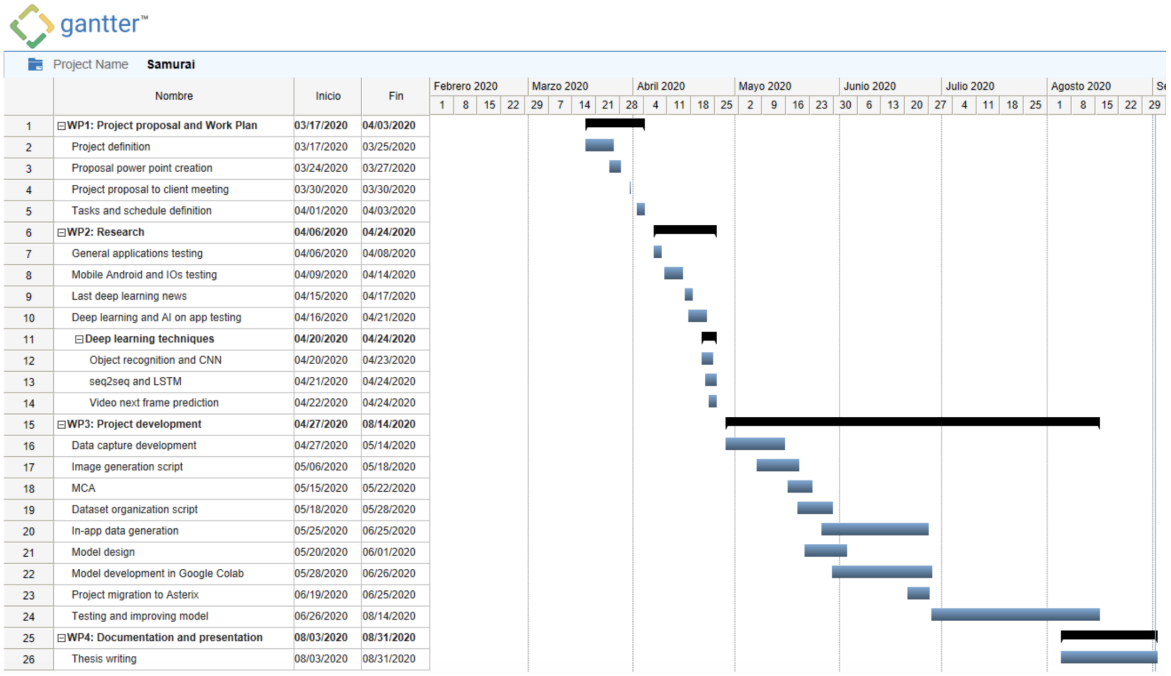


Figure 1.1: Gantt diagram

[1] https://www.pytorch.org/
[2] https://opencv.org

Chapter 2

State of the art

This chapter aims to introduce the reader to the basic and relevant concepts about deep learning, but also review some literature about specific models and app testing tools that helped the development of this work.

2.1 Neural Networks

Artificial Intelligence appears from the aim of creating machines that simulate human intelligence. Software able to do things that human beings can do, such as understanding a book, recognize faces, speak, translate, or even drive vehicles, to help us in our day-to-day life and improve it, but also to speed up and simplify industry tasks. It is known that computers are faster and more precise than human beings solving mathematical problems or even playing shogi, chess or even Atari games [34]. But when it tries to solve other tasks that we do intuitively such as recognizing a face, it becomes more complicated to solve. Deep Learning presents a method to solve these more intuitive problems by allowing machines to learn from experience.

Neural networks have become the main solution for many Machine Learning problems. They were proposed time ago [22] [32], but it has not been until now, with the evolution and improvement of the computational capacity of devices and the large amount of data we store allowing the creation of large data sets, that this solution has been considered a reliable approach. In fact, it has become really popular and widely used for lots of different tasks such as translation [7], face recognition [20], text generation [21] and more.

From a mathematical point of view, a neural network can be seen as a function approximator that learns how to map some inputs to their respective outputs. It consists of a set of non-linear operations whose parameters are trained in order to minimize a cost function. A layer in a neural network is modeled as:

$$h = f(W_h x + b_h) \tag{2.1}$$

where x is the input, f is a non-linear function (e.g. ReLU) and W_h and b_h are trainable weights and biases, respectively. Neural networks use to concatenate more of these layers to obtain a more complex system with more capacity. These are called Feed-Forward neural networks.

There are different NNs architectures. Another extensively used one is the Convolutional Neural Network model. These networks are commonly used in tasks that involve images such as object recognition [9] or face detection[31]. They consist of a set of convolutional layers that are able to extract and learn image features at different resolution levels. Each layer has a filter with its own kernel size, that is convoluted with the image, and each kernel is connected to neuron which has the learnable parameters to optimize the system.

The two neural networks architectures explained before do not care about previous inputs, they just learn sample by sample. But, what if we need to learn something whose main characteristic is that it is a sequence? Things like text, speech or video can be described as sequences of

letters, words or images. Recurrent Neural Networks extend the feed-forward model by adding a recurrent connection in time:

$$h = f(W_h x_t + U_h h_{t-1} + b_h) \tag{2.2}$$

where U_h operates on the hidden state in the previous time step, h_{t-1}, allowing information to persist along time dimension. Some examples of these kind of architectures are developed in [15] and [35]. The main problem of RNNs is that it is difficult for them to learn long sequences [8]. This led to the development of *gated* units and the so called LSTM architecture [17].

LSTM stands for Long Short-Term Memory and they main implementation concept is *cell state*, allowing to retain information along time and not suffering from same problem as RNNs. LSTM equations are described as follows:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{2.3}$$

$$i_t = \sigma((W_i x_t + U_i h_{t-1} + b_i) \tag{2.4}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{2.5}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{2.6}$$

$$h_t = o_t \odot tanh(c_t) \tag{2.7}$$

where i_t is the input gate, f_t is the forget gate and o_t the output gate, all of them at time t. σ is the sigmoid function, $\odot$ is the element-wise multiplication and W and b are weights and bias trainable matrices respectively. The *cell state* at time t is represented by c_t and the hidden state at each time state is represented by h_t.

The other fundamental part of deep learning and training neural networks is the data and the training process. A set of labeled samples is needed so that the network can learn from it. We give the network train and validation inputs and their respective labels, but the behavior of the layer is not defined. The layer parameters have to be tuned in order to optimize a loss function, that indicates how well the network is predicting, in other words, how close to the expected outputs these predictions are.

2.2 Automated test input generation

Since the appearance of mobile applications, automated GUI test generation has become an active research area. As you can guess, manually writing test cases can be hard, time consuming and error prone. Automated test generation approaches are normally designed with a particular objective, such as achieving high coverage, uncovering the largest amount of bugs, reducing testing scenarios or generating test scenarios that mimic representative use cases of an application.

Automated input generation techniques use to be divided into three approaches [25]:

- Random based input generation [12], [27].
- Semantic input generation [30], [4], [5].
- Model based input generation [29], [19], [28].

Recent work [10] illustrated the relative ineffectiveness of many research tools when comparing program coverage metrics against a naive random approach and highlighted many unsolved challenges including generation of system events, the need for manually specified inputs for certain complex app interactions, adverse side affects between different runs and a need fro reproducible cases among others.

Regarding GUI, it is one of the most important parts of an app. It allows the user to understand and know how to interact with it. There are two lines of research in this area. First is understand the problem from the software engineering point of view; and the second is to study the human-device interaction point of view to analyze the UI design. Many automated testing software uses GUI model based approaches to guide input generation. They analyse the information of current UI state and not only the transitions between UI states [18], [33], [23].

Last but not least, we want to talk about deep learning in the field of automated input test generation. There is no much research about introducing deep learning into automated testing. There is just a very recent article, from 2019, where they implement this concept. It was published by Li, Yang, Guo and Chen [24], in which they develop a system called Humanoid to generate test inputs using artificial intelligence. They trained a neural network with an Android user-app interactions dataset called Rico database [11]. Our work tries to follow their research line and is based on this paper, trying to reproduce it but with some model modifications and a custom dataset.

Chapter 3

Methodology

3.1 Design

In this section we explain the main structure of the system. Our main goal is to create a bot able to navigate through an app. For this, we have to train a neural network that consumes a sequence of images, that represents the actual screen and the three previous screens, and outputs two probability distributions: one corresponding to the location of the next touch and the second corresponding to the type of action. We can see a representation of the system in Figure 3.1.

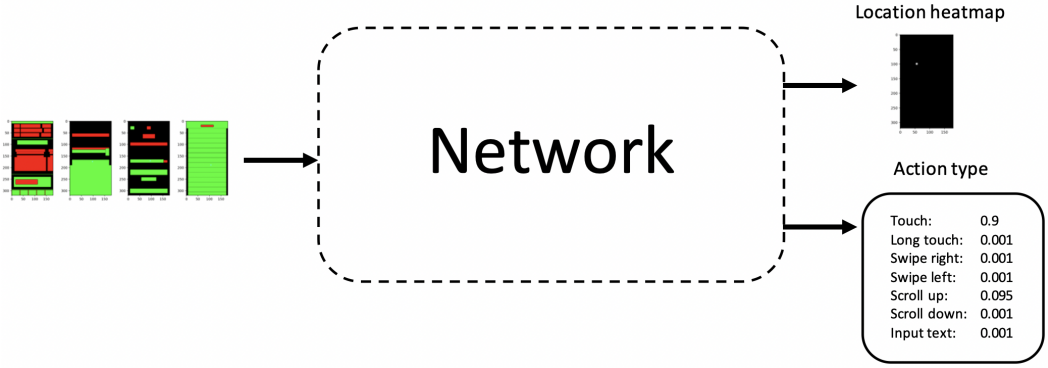


Figure 3.1: Scheme of the system

3.2 Dataset

Now that we know the structure of the project, in this section we are going to explain the steps we followed to generate the data that we used to create the dataset.

In general terms, the dataset consists of 5504 user interactions. Each sample is a sequence of 4 images representing the actual and previous screens. This will allow the networks to learn how to navigate through the app thanks to the context we are giving to it.

If we get a sample of the dataset we will get the following data:

- Sequence of four RGB images. It is a tensor of dimensions `[4, 3, Height, Width]`
- Location label. A tensor of dimensions `[4, 1, Height, Width]`
- Interaction label. An integer in the range [0,6] representing the seven action types.

As it is a custom dataset, we also made a custom dataset class in order to manage it. This class inherits from the PyTorch's class Dataset[1].

[1] https://pytorch.org/docs/stable/data.html#torch.utils.data.Dataset

3.2.1 Data acquisition, the MCA module

The first approach was to make a screen shot of the actual screen in order to have an image of the GUI, but then we realized that this would take a big computational and storage cost. On one hand, it can affect the smooth operation of the app slowing it down, and on the other hand, storing a high-resolution image of the screen would take up a lot of storage space. That is why we decided to capture only the hierarchy of each screen, which is a kind of dictionary that stores all the elements that compose the interface. This allows us to save in computational time and storage up to 40%.

We obtained our data from a private bank account and money managing app. In order to extract the data, we implemented and introduced an MCA within the app. MCA refers to "Módulo Común Aplicativo", it is a module developed by the company business department that is transversal to several applications in the company. In this case, our MCA is in charge of capturing:

- The hierarchy of the actual screen where the user taps. This gives information about all the elements that was present on the screen in the moment the user made an interaction, such as buttons, labels, images, text boxes and others.
- The location where the user has tapped, represented with two coordinates (x,y).
- Information about the tap such as start and end location of the tap, start and end time of the tap. This information allows us to know if this was a regular tap or a long tap, but also a scroll or a swipe (up or down).
- Information about the keyboard activation. For privacy reasons we were not able to capture the text input of the user, but we need to know at least if the keyboard is active because in those cases the user can also tap buttons that are out of it.

With this module, we were able to generate data just using the app and navigating through it. This data was stored in *json* files that we will call traces, and will be used later to generate the image data. As the generated files are stored inside the mobile phone, we coded the key words in order to minimize the storage space they occupy. The average size per file is about 8KB.

3.2.2 Images

From the information stored in the traces, we can "draw" our screens and generate a simpler representation of them. We made our representation based on a flag common to all the elements of the GUI: "*interaction_enabled*". If this parameter is set to *True*, it means this element is interactive, in other words, if you touch it, you cause a change in the GUI such as change to another screen or make a message appear. Examples of interactive elements could be: buttons, sliders or text boxes. On the other hand, non-interactive elements are those that if you interact with them nothing happens, such as texts, images, titles or labels, among others. Based on this flag, we will represent interactive elements with the color green and non-interactive elements with the color red. The process of data transformation is shown in Figure 3.2.

Figure 3.2: Data conversion from the app hierarchy to the final image.

In origin, images are stored with size *667x375*, that is the size of the screen in the mobile app; and the image size file is between 4 and 7 KB. We can see more examples of app screens and how, after processing, we obtain the representative image in Figure 3.3.

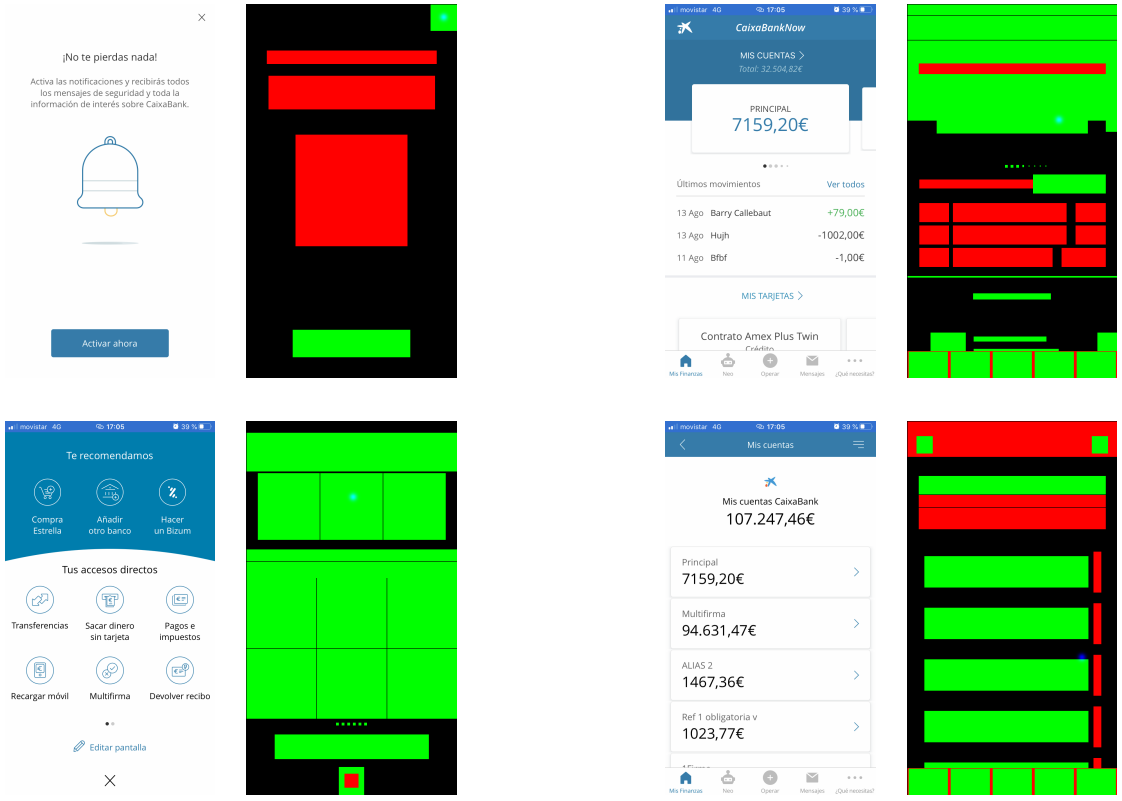


Figure 3.3: Examples of app screens and their equivalent simplification.

3.2.3 Interactions

We define interaction with two concepts: location and type of action. In this section we explain what they are and how we obtain them.

Location stands for the coordinates where the user performed the interaction on the screen. We obtain this coordinates from the traces. To represent the location, we decided to code the coordinates in a heat map that will be the same size of the screen image (see Figure 3.4). It represents a distribution in which each pixel corresponds to the probability of this pixel to be the target location. As the raw coordinates are highly non-linear and difficult to learn, we decided to represent the location with a gaussian distribution around the target location that is easier to learn.

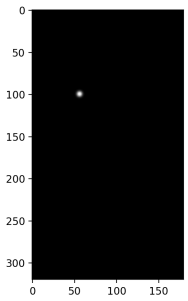


Figure 3.4: Example of heat map location.

Action type is self explanatory, it represents the type of the action the user performed. We also refer to it as gesture. In this work we considered 7 types, including `touch`, `long touch`, `scroll up/down`, `swipe left/right` and `input text`. In Table 3.1 we can observe each action type with its corresponding class value. This classes will be used later in the classifier part of the network.

In order to know which type of action it is, we make use of the start and end coordinates and the start and end time of the stored in the traces. We established a set of heuristic rules based on [13] to determine the interaction type. Those rules are showed in Table 3.1.

Interaction	**Class**	**Rules**
Touch	0	$\|loc_{end}$ - $loc_{start}\| < 48px$ and $time_{end}$ - $time_{start} < 500ms$
Long touch	1	$\|loc_{end}$ - $loc_{start}\| < 48px$ and $time_{end}$ - $time_{start} > 500ms$
Swipe right	2	$\|loc_{end}$ - $loc_{start}\| < 48px$ and loc_{end} is on top / right / bottom / left from the loc_{start}
Swipe left	3	
Scroll up	4	
Scroll down	5	
Input text	6	keyboard activated flag

Table 3.1: Interaction types with their corresponding class and the heuristic conditions.

At the same time we generate the images from the traces, we also generate a *csv* file where we associate each image to its action type. In this way, just knowing the image filename we also know which action type was performed.

3.2.4 Context

To achieve the network to learn how to use the app mimicking a real user, we need to introduce the concept of context.

The context refers to the sequence of actions that led the user to the current screen, that is: previous screens and previous interactions. This is important because when a user navigates through an app, normally it has an intention. For example, if the user wants to check one of its bank account, it will follow a concrete path to get the account screen, but if the user wants to change the password of its account, it will follow a different path in order to arrive to the corresponding screen.

Our actual screen is s_i and the action to be performed on this screen is a_i, so for the actual screen we have the interaction (s_i, a_i). If we apply this to the context concept, the context of the actual screen is $c_i = <(s_i), (s_{i-1}, a_{i-1}), (s_{i-2}, a_{i-2}), (s_{i-3}, a_{i-3})>$. Notice that the action taken in the actual screen does not belong to its context, because this is the thing that we want to predict.

As a result, the samples of our database are a sequence of images and not single images.

In this project we considered a context of 3 previous screens with its 3 corresponding previous interactions. It means that each sample in the dataset will be formed by:

- 3 images for the previous screens.
- 1 image for the actual screen.

3.2.5 Inputs and Labels

In our dataset, each sample is formed by 4 images. Each image is represented in RGB format, so we have 3 channels per image: Green, Red and Blue. This is perfect for our purpose, because we can store all the information relative to screen and interaction in a single image as we can observe in 3.5. At the end, in each image channel we have:

- *R* channel: Non-interactive elements.
- *G* channel: Interactive elements.
- *B* channel: Location of the interaction.

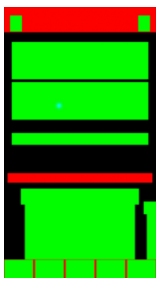


Figure 3.5: Single screen with its action location.

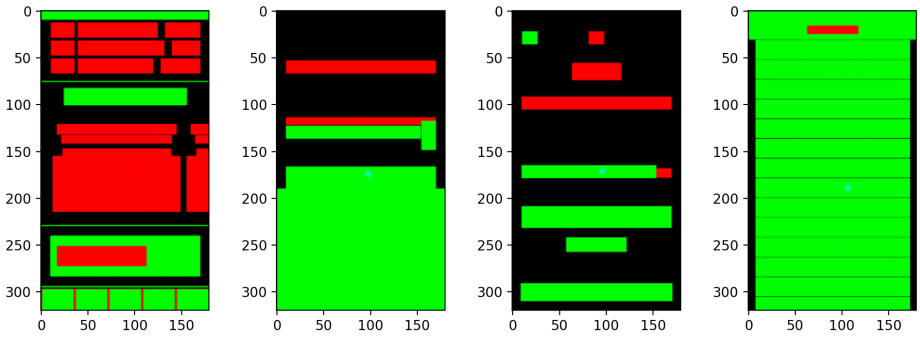


Figure 3.6: Sample.

In Figure 3.6, we can observe the 4 screens sequence. First image (left) is the actual screen and the other three images are the previous screens. They are ordered temporarily from right to left. Intuitively, first 3 traces do not have a complete context of previous images. In this case, we just pad these previous states with zeros.

In a sequence, the images belonging to the context have their corresponding location heat map in the *B* channel, but the image of the actual screen does not. This is because the location of the action the user took on the actual screen is the label of the sequence, it is the ground truth we use to compare with the network prediction. In that way, once we take the *B* channel from the actual screen image, we fill it with zeros in order to keep the three channel format.

In order to save storage space, we store images as single image, and not as sequences, each one with its own action location in *B* channel. To form the sequences, we use the time information we get from the traces *json* files, and then generate a *csv* file where we organise the images in sequences. This file is structured in 4 columns, as sequences are 4 images long, and each column has the path to the corresponding image file.

Summarizing, the **input** of the network is a sequence of 4 images: the actual screen and the 3 past screens and their corresponding interaction locations. On the other hand, the **labels** are: the heat map, that represents the probability distribution of the screen pixels to be the target location; and the action type class. The set of training components are shown in Figure 3.7.

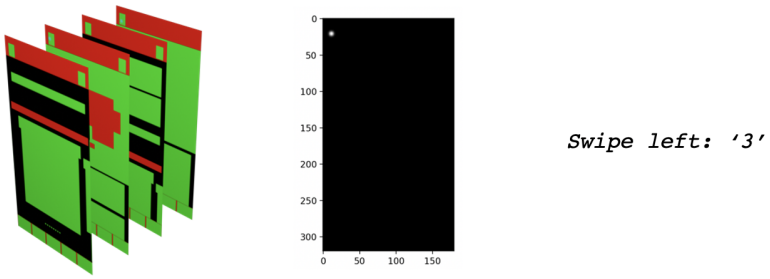


Figure 3.7: Final set of input and labels.

3.3 Model

In this section we describe the neural network we programmed, its parts and why we decided to choose this structure. This model is inspired by the one presented in [24] but with some differences such as: it is simpler, having one less convolutional layer and consequently it has one less LSTM layer. It also has one Linear layer to adjust sizes at the en of the location part, and also one more Linear layer in the action type part.

The network follows a *Convolutional Encoder-Decoder* structure with two *LSTM* layers in the middle after the bottleneck. It has two outputs: one is the *De-Convolutional* part of the *Encoder-Decoder* to calculate the heat map and the other is a *Linear* or *Fully Connected* part, in charge of predicting the action type.

As we said in 3.1, we need a model to predict two conditional distributions:

- $p_{type}(type \mid c_i)$ with $type \in \{\texttt{touch}, \texttt{long touch}, \texttt{scroll up}, \texttt{swipe right}...\}$ It is the probability distribution of the type of the next action a_i given the current context c_i.
- $p_{location}(x, y \mid c_i)$, where $0 < x < screen\_width$ and $0 < y < screen\_height$. It is the probability distribution of the target location of the next action a_i given the current context c_i.

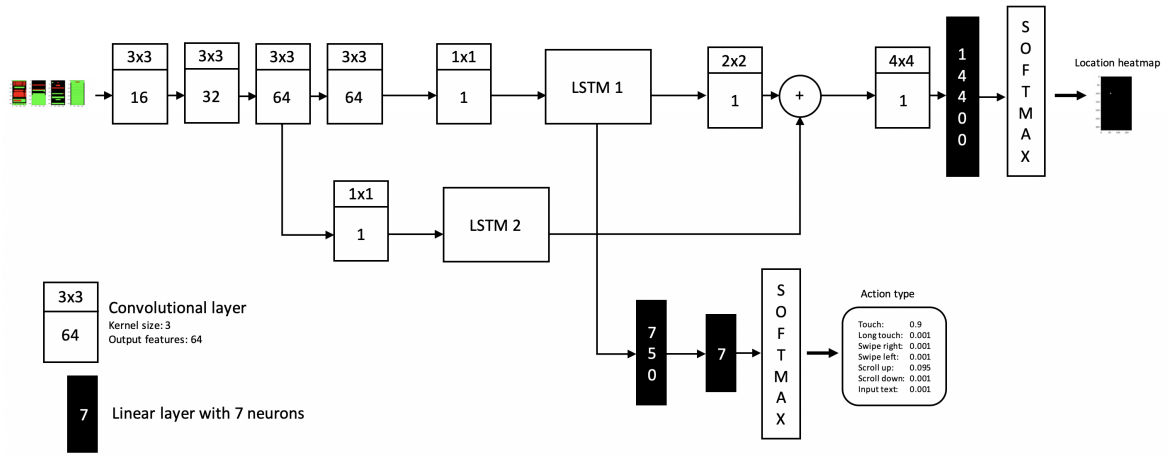


Figure 3.8: Model architecture.

In Figure 3.8 we can see the full model architecture. The first part is formed by **Convolutional** layers. Convolutional networks have become the most popular approach for image feature extraction as they are very powerful in computer vision tasks [36]. Our model has 4 convolutional layers with ReLU activations to perform the feature extraction from the GUI representations and location heat maps. After each convolutional layer, there is a max-pooling layer with stride-2 that reduces the width and height of its input to half, and a batch normalization layer that allows each layer to learn by itself a little bit more independently of other layers, but also helps to prevent overfitting because it has slight regularization effects.

Next part are the **LSTM** modules. LSTM (Long Short-Term Memory) networks are used for sequence-to-sequence problems such as translation or video next frame prediction. We can see

the internal structure if an LSTM gate and how they are connected to from a chain in Figure 3.9. LSTM networks deal with exploding and vanishing gradient problems that can appear when training traditional RNN by incorporating gates to regulate the information flow. We put residual LSTM modules after each of the 2 last convolutional layers to extract features at different levels of resolution. The residual LSTM adds the last dimension of the input and the output of the regular LSTM together, this makes the network easier to optimize [16] and gives hint that the location of an action lies inside a GUI element. In order to decrease model complexity, we added a convolutional layer before each LSTM just to reduce the number of features from 64 to 1.

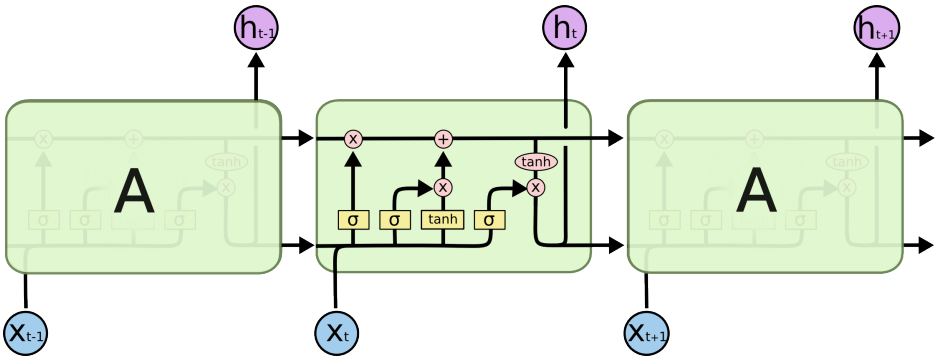


Figure 3.9: LSTM chain and LSTM gate.

Now the network splits into two parts, to output both probability distribution. The **De-Convolutional** layers are used to get a get back to the input size and get a high resolution probability distribution from the low resolution output of the LSTM modules. Features of different resolution levels are combined to improve the quality of the heat map [26]. As the output of the De-Convolutional layers is not exactly the same as the input of the network, we added a Linear layer to adjust the size to [160x90]. At the end, we use a **Softmax** layer to normalize the output so that all pixels sum to 1, as it has to be a probability distribution.

The other output consists of 2 **Linear** or Fully connected layers with a **Softmax** to predict the probability distribution of the action types.

Regarding activation functions, we tried some of them like Sigmoid, ReLU and Tanh. As our targets are probabilities and have values between 0 and 1, the best options were Sigmoid and ReLU, because they both put negative values to zero. Sigmoid is a good option for our objective, but tends to have the gradient vanishing problem due to the small values the sigmoid derivative has. This is solved using the ReLU activation, because when it's derivative is back-propagated there will be no degradation of the error. So we used ReLU as the activation function along the entire network.

3.4 Training

In this section we explain how we proceeded to prepare the data and get it ready for training.

First of all, we applied a **resize** of the images because their original size is 667x375. This size is too large to handle in memory during training, and as the images are quite simple, we can afford a resizing to a smaller size like 160x90 keeping all details.

Second step is to **normalize** and **standardize** the data. Normalization is the process of rescaling data in the range $[0, 1]$ or $[-1, 1]$ operating as follows respectively:

$$x' = \frac{x - min(x)}{max(x) - min(x)} \quad (3.1) \qquad x' = \frac{x - mean(x)}{max(x) - min(x)} \quad (3.2)$$

In our case, as the pixels of images are already zeros and ones, it is not necessary to normalize them.

Standardization is used to rescale the data to zero-mean and unit-variance.

$$x' = \frac{x - \mu}{\sigma} \quad (3.3)$$

where μ stands for the expectation of x and σ is its standard deviation. We calculated both parameters along each image channel obtaining `mean:[0.1976, 0.4471, 0.0008]` and `std:[0.3848, 0.4826, 0.0196]`.

The complete dataset has 5504 samples, but for the training process, we split it into training and validation set. We gave 80% for the training set (4403 samples) and 20% for the validation set (1101 samples).

To train the network we need a **cost function** to obtain a measure of how good the network predicting our target labels is. In our case, we have two outputs, so we need two loss functions. As both *heat map* and *action type* outputs are probability distributions, the most suitable cost function we found is the Cross Entropy Loss. This function measures the performance of a classification model whose output is a probability value between 0 and 1. It is calculated as follows:

$$H(p, q) = -\sum_{\forall x} p(x) log(q(x)) \quad (3.4)$$

In PyTorch, *CorssEntropyLoss*[2] just accepts as labels the corresponding target class. For example, imagine we have a classification problem with 4 classes, so classes belong to `[0,3]`; if our input corresponds to class 2, the label will be '2', and not a one hot encoded like `[0,0,1,0]` nor a probability distribution like `[0.1, 0.1, 0.6, 0.2]`. This helps us with the *action type* part of the network because the *action type* labels follows this format. But the *heat map* labels are probability distributions, also called *soft* labels. So in order to be able to use Cross Entropy loss we had to implement a custom soft cross entropy function in order to accept these kind of labels.

In order to updates the network parameters, we need something called **optimizer**. This is an algorithm or method to change the attributes of the network such as weights and biases in order to reduce the loss values. Regarding the *optimizer*, we trained the network with two optimizers:

- *Stochastic Gradient Descend* (SGD) with momentum. It is a variant of the basic Gradient Descent algorithm. It changes the model parameters more frequently, specifically after loss computation in each train sample. One of the disadvantages of SGD is the high variance

[2] https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.htmltorch.nn.CrossEntropyLoss

in model parameters. To solve that, we added a *momentum* parameter that accelerates the convergence towards the relevant direction and reduces the fluctuation to the irrelevant direction.

- *Adaptive Moment Estimation* (Adam). It works with momentums of first and second order. The idea behind Adam is not to roll so fast just because we can avoid a minimum; we want to decrease the velocity a little for a careful search. The main advantages of Adam are that it converges in a fast way and rectifies vanishing learning rate avoiding high variance. On the other hand, it is computationally expensive.

In both optimizers, we added an L2 Regularization of the weights. It is a penalty added in the weight update process and avoids very big weight values smoothing the oscillations of the loss curve.

To check the **accuracy** of the system we employed bounding boxes around heat map locations in order to allow a certain degree of error, since a button has a number of pixels that can be clicked, so there is no difference if you click some pixels away. We used `10x10` bounding boxes. We applied a widely used measure in object detection called Intersection Over Union (`IoU`) [37]. This measure calculates the overlapping between both prediction and ground truth bounding boxes. In Figure 3.10 we can see an example of both bounding boxes on a real sample. Then, taking into account the percentage of overlapping area, we determine an acceptable threshold to say if it is a hit or not. The most common thresholds are 50% and 75%. We decided to use the second because is more restrictive, as we are already allowing some degree of freedom when using the bounding box method.

$$IoU = \frac{Area_{Intersection}}{Area_{Union}} \tag{3.5}$$

The threshold we put to consider the prediction is a `hit` is of $\texttt{IoU} \geq \texttt{0.75}$. It means that, if the overlapping between both bounding boxes is lower than 75%, it will be considered as a `fail`.

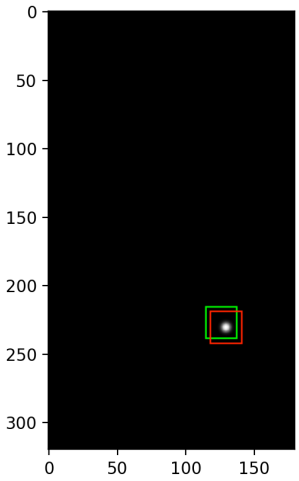


Figure 3.10: Bounding boxes example. Green bounding_box: ground truth. Red bounding_box: network prediction

Chapter 4

Results

In this section we present and describe some of the trainings we performed and the respective training results we obtained.

To optimize training process, it is recommended to initialize network parameters. Initializing the network with the right weights can be the difference between the network converging in a reasonable amount of time and the network loss function not going anywhere. The initialization we chose is Xavier initialization, proposed in [14] using a normal distribution.

As the network has a multi-output structure, we have to calculate two losses: heat map and action type. But we can only use one loss value to perform the backward pass. To do that, we just added both losses and used the result to update the weights like:

$$loss = loss_{heatmap} + loss_{actiontype}.$$

From all the training experiments carried out, we have chosen only the most relevant to show them in this report. Their results can be seen in Tables 4.1 and 4.2. We present 5 trainings with different parameter configurations changing the values of learning rate, weight decay, momentum and number of epochs. Also, as we said in 3.4, we tried two different optimizers.

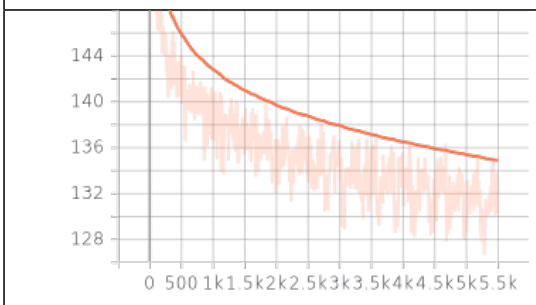
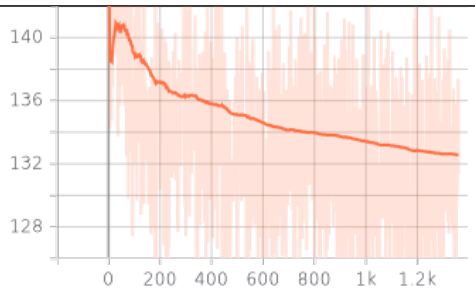
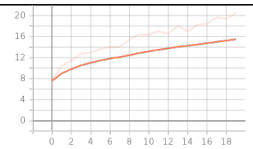
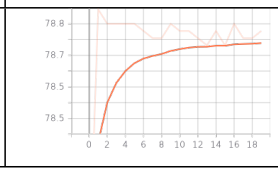
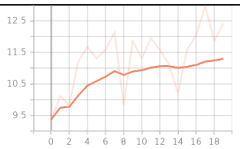
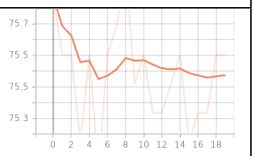
The first result to comment is that action type prediction almost always gets an accuracy of 75%. It means the classifier part of the network works pretty well. It seems to be the part that most depends on changing parameters is the location predictor part. It could be due to the fact that making a classification could be easier than predict a bigger probability distribution.

We tried different learning rates, but we got that the one which gives the best result is `0.01`. A learning rate of `0.001` is too low and delays the training a lot. On the other hand, a higher learning rate of `0.1` is too high and leads to very weird and static loss curves as we can see in Table 4.3.

Regarding optimizers, Adam was expected to perform better but surprisingly SGD performed better in our experiments. Adam is able to reach a higher training accuracy as we can see in Training 2, getting almost 35%, but validation accuracy is very bad with 11% in front of 14% of Training 3. Training 2 gets better results than Training 1 in general because it trained for 50 epochs.

Focusing on Trainings 3 and 4, here we introduce the concept of momentum that is supposed to help reaching the goal in less steps. We tried values in the range `[0.7, 0.9]` and we presented the values with better results `0.85` and `0.88`. Momentum is a parameter that changes a lot how fast we get the minimum of the cost function when using Stochastic Gradient Search. As you can see, we got better validation accuracy in Training 3 in only 12 epochs in front of the 20 epochs of Training 4.

The best result was obtained for Training 3 as it got the highest location accuracy of 14%, even though it got a 74% of action type accuracy.

Training 1					
Epochs: 20	BS: 16	Optim: Adam	LR: 0.01	wd: 0.001	momentum: -
Train Loss			Validation Loss		
					
Train HM acc.: 20%		Train AT acc.: 78%	Val. HM acc.: 12%		Val. AT acc.: 75%
					

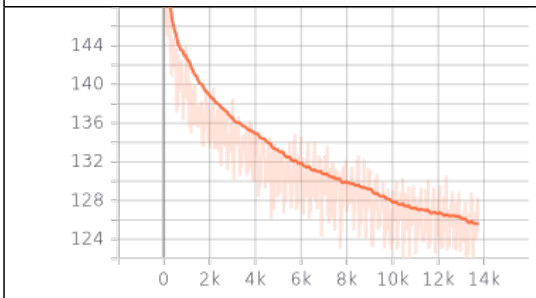
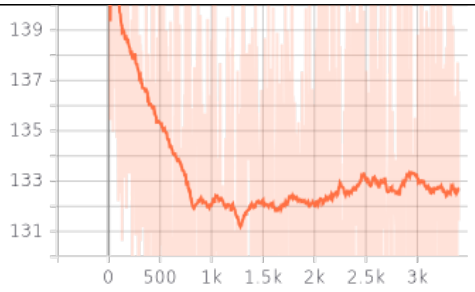
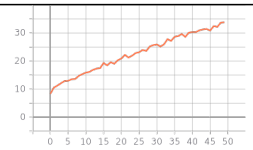
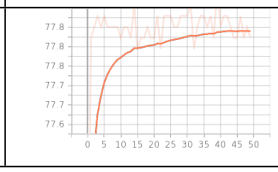
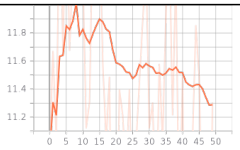
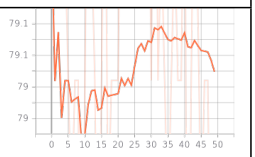
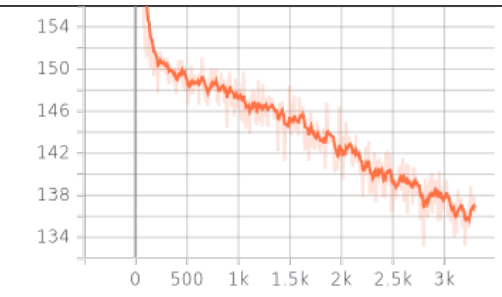
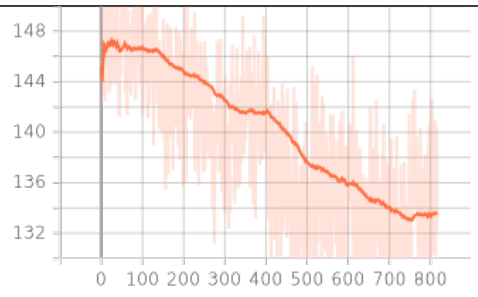
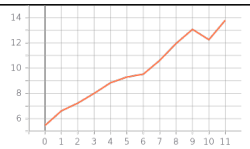
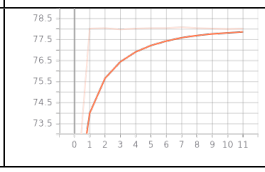
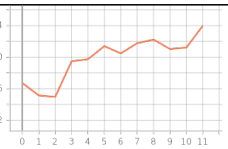
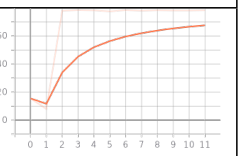
Training 2					
Epochs: 50	BS: 16	Optim: Adam	LR: 0.01	wd: 0.001	momentum: -
Train Loss			Validation Loss		
					
Train HM acc.: 34%		Train AT acc.: 77.8%	Val. HM acc.: 11.5%		Val. AT acc.: 79%
					

Table 4.1: Training 1 and 2 results. Epochs: Number of epochs, BS: Batch Size, Optim: Optimizer, LR: Learning Rate, wd: Weight Decay, momentum: Momentum.

Training 3					
Epochs: 12	BS: 16	Optim: SGD	LR: 0.01	wd: 0	momentum: 0.88
Train Loss			Validation Loss		
					
Train HM acc.: 14%	Train AT acc.: 77%		Val. HM acc.: 14%	Val. AT acc.: 74%	
					

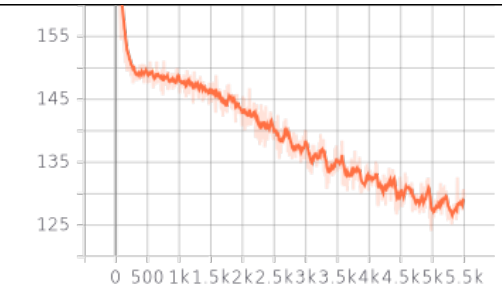
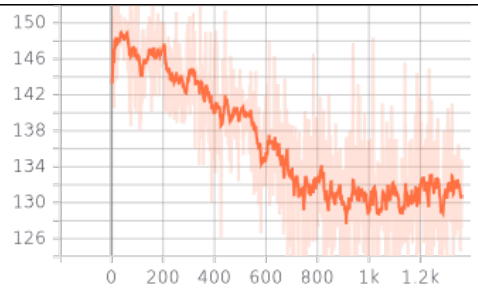
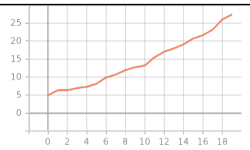
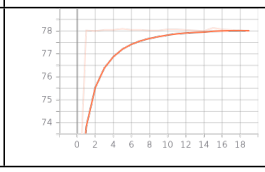
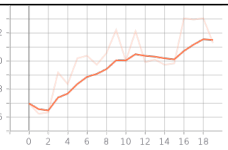
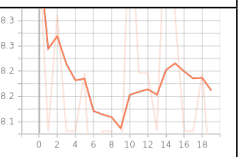
Training 4					
Epochs: 20	BS: 16	Optim: SGD	LR: 0.01	wd: 0.001	momentum: 0.85
Train Loss			Validation Loss		
					
Train HM acc.: 26%	Train AT acc.: 78%		Val. HM acc.: 12%	Val. AT acc.: 78%	
					

Table 4.2: Training 3 and 4 results. Epochs: Number of epochs, BS: Batch Size, Optim: Optimizer, LR: Learning Rate, wd: Weight Decay, momentum: Momentum.

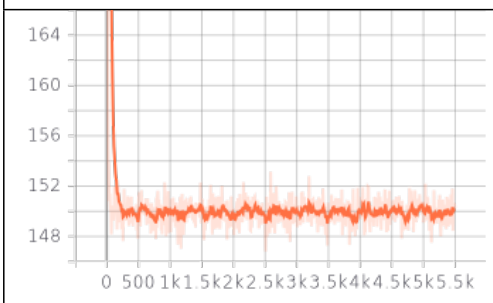
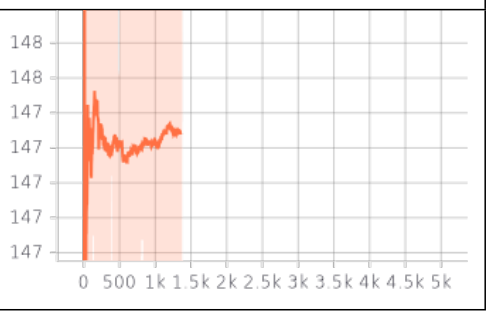
<table>
<tr><td colspan="6">Training 5</td></tr>
<tr><td>Epochs: 20</td><td>BS: 16</td><td>Optim: Adam</td><td>LR: 0.1</td><td>wd: 0.001</td><td>momentum: -</td></tr>
<tr><td colspan="3">Train Loss</td><td colspan="3">Validation Loss</td></tr>
<tr><td colspan="3"> 
</td><td colspan="3"> 
</td></tr>
</table>

Table 4.3: Training 5 results. To show the bad performance of too high learning rate. Epochs: Number of epochs, BS: Batch Size, Optim: Optimizer, LR: Learning Rate, wd: Weight Decay, momentum: Momentum.

Chapter 5

Budget

The hardware resources needed for the project were a MacBook Pro laptop and the Asterix server. The GPU was used during 1 month for the development of the model, which adds to 400 hours of computation. We compute the computation cost based on Amazon Web Services (AWS) rates[1] for *p2.xlarge* instances with one NVIDIA K40c. Software employed was Atom[2] that is license free.

The main costs of this project come from the salary of the researches and the time spent in it. The team for the development of this thesis is formed by two senior app developers as the developers of the MCA and myself as a junior engineer. The length of the project was 20 weeks, as presented in the Gantt diagram. Assuming a commitment of 30 weekly hours and that each advisor spent an average of 1h per week on meetings, the complete costs for the project are the following:

	Amount	**Cost/hour**	**Time**	**Total**
GPU *TESLA* K40c	1	0,90 €	400h	360 €
Junior engineer	1	10,00 €	700h	7.000 €
Senior developer	2	30,00 €	40h	1.200 €
Other equipment	-	-	-	3.000 €
			Total	12.760 €

Table 5.1: Cost of the project. *Other equipment* includes office and campus services and employed laptop.

[1] https://aws.amazon.com/ec2/instance-types/p2/?nc1=h_ls
[2] https://atom.io

Chapter 6

Conclusion and Future Work

We presented the design and training processes of a deep learning prototype model that tries to learn how to navigate through a mobile application mimicking the human behavior. For that, we created a dataset from scratch. We implemented a software module (MCA) in order to extract the user data from the application: location of the touch and type of action; and then process this information to generate the images that will be the inputs to feed our model. These images are simpler representations of the application GUI screens.

Once we had our dataset prepared, we designed and programmed a neural network formed mainly by a convolutional part to recognise features from the GUI screen, and an LSTM part to learn the sequence information of the user interactions.

The best experiments obtained a validation accuracy of 14% in the location prediction (the accuracy for a random classifier (random guess) is 0.35%) and 79% in the action type prediction. These results are highly improvable, but for a prototype it is a good starting point.

We did try different network configurations, but as future work, we want to try way more in order to improve the results. Furthermore, we want to try longer training time, generate a larger dataset or try to use transfer learning to train first one part of the network and then the other. Another possibility would be to train the network with a bigger public dataset and then use transfer learning method to train it over our custom dataset. The next step of this project, after improving the results, is to connect the output of the network to a testing software to use the model as an automatic input generator and check how many screens it can go through. Once the prototype is ready to navigate, the idea is to try to train it to find bugs and glitches over the app using methods like reinforcement learning, rewarding it every time it finds an error.

Bibliography

[1] Appium testing framework, http://appium.io.

[2] Espresso testing framework, https://developer.android.com/training/testing/espresso.

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Gennaro Imparato. A toolset for gui testing of android applications. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 650–653. IEEE, 2012.

[4] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[5] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 641–660, 2013.

[6] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, page 238–249, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[8] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[9] Yongqiang Cao, Yang Chen, and Deepak Khosla. Spiking deep convolutional neural networks for energy-efficient object recognition. *International Journal of Computer Vision*, 113(1):54–66, 2015.

[10] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. Automated test input generation for android: Are we there yet?(e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440. IEEE, 2015.

[11] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, pages 845–854, 2017.

[12] A. Developers. Ui/application exerciser monkey,, 2012.

[13] Zac Dickerson. Size matters! accessibility and touch targets, 2018. Last accessed 19 June 2020.

[14] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[15] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[18] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1036–1046, 2014.

[19] Casper S Jensen, Mukul R Prasad, and Anders Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67–77, 2013.

[20] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[21] Rémi Lebret, David Grangier, and Michael Auli. Neural text generation from structured data with application to the biography domain. *arXiv preprint arXiv:1603.07771*, 2016.

[22] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[23] Yuanchun Li, Yao Guo, and Xiangqun Chen. Peruim: Understanding mobile application privacy with permission-ui mapping. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 682–693, 2016.

[24] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. A deep learning based approach to automated android app testing. *arXiv preprint arXiv:1901.02633*, 2019.

[25] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 399–410, 2017.

[26] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[27] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234, 2013.

[28] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609, 2014.

[29] Ke Mao, Mark Harman, and Yue Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[30] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. Automatically discovering, reporting and reproducing android application crashes. In *2016 IEEE international conference on software testing, verification and validation (icst)*, pages 33–44. IEEE, 2016.

[31] Omkar M Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep face recognition. 2015.

[32] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[33] Julia Rubin, Michael I Gordon, Nguyen Nguyen, and Martin Rinard. Covert communication in mobile applications (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 647–657. IEEE, 2015.

[34] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265*, 2019.

[35] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *ICML*, 2011.

[36] Jonathan J Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *Advances in neural information processing systems*, pages 1799–1807, 2014.

[37] Dingfu Zhou, Jin Fang, Xibin Song, Chenye Guan, Junbo Yin, Yuchao Dai, and Ruigang Yang. Iou loss for 2d/3d object detection. In *2019 International Conference on 3D Vision (3DV)*, pages 85–94. IEEE, 2019.