Engineering Degree Project

Performance analysis: CNN model on smartphones versus on cloud
- *With focus on accuracy and execution time*

*Authors:*
Edwin Johansson &
Klas Stegmayr

*Supervisor: Moa Andersson*

*Lnu Supervisor: Neda Maleki*

*Semester: Spring 2023*

*Subject: Computer Science*
Abstract

In the modern digital landscape, mobile devices serve as crucial data generators. Their usage spans from simple communication to various applications such as user behavior analysis and intelligent applications. However, privacy concerns associated with data collection are persistent. Deep learning technologies, specifically Convolutional Neural Networks, have been increasingly integrated into mobile applications as a promising solution. In this study, we evaluated the performance of a CNN implemented on iOS smartphones using the CIFAR-10 data set, comparing the model’s accuracy and execution time before and after conversion for on-device deployment. The overarching objective was not to design the most accurate model but to investigate the feasibility of deploying machine learning models on-device while retaining their accuracy. The results revealed that both on-cloud and on-device models yielded high accuracy (93.3% and 93.25%, respectively). However, a significant difference was observed in the total execution time, with the on-device model requiring a considerably longer duration (45.64 seconds) than the cloud-based model (4.55 seconds). This study provides insights into the performance of deep learning models on iOS smartphones, aiding in understanding their practical applications and limitations.

Keywords: CNN, Deep learning, iOS, Core ML, CIFAR-10
Preface

This project’s journey has been a mix of challenges and moments of joy. Delving into interesting topics with a close friend and being committed to making things work has been a fun experience. Despite encountering obstacles, and not everything turning out as anticipated, we are content with the outcomes. We thank our supervisors, Moa and Neda, for their constant help throughout this project. Also, we sincerely thank the entire team at Bontouch for their hospitality and priceless assistance. In particular, we acknowledge Andreas, Frida, Christoffer, and Jimmy for their considerable help and support during the project. Further, we would like to thank Tobias Ohlsson for valuable insights guiding us in the right direction throughout the thesis project.
Contents

1 Introduction

1.1 Background ................................................. 1
1.2 Related work ............................................. 1
1.3 Problem formulation ...................................... 3
1.4 Motivation ............................................... 3
1.5 Milestones ............................................... 4
1.6 Scope/Limitation ........................................ 4
1.7 Target group ............................................. 5
1.8 Outline .................................................. 5

2 Theory

2.1 Software .................................................. 7
   2.1.1 TensorFlow and Keras .............................. 7
   2.1.2 Core ML ............................................. 7
   2.1.3 Vision .............................................. 7
   2.1.4 Protocol Buffers .................................. 8
2.2 Convolutional Neural Network .......................... 8
   2.2.1 Convolutional Layer ................................ 8
   2.2.2 Activation Layer .................................. 13
   2.2.3 Pool Layer ......................................... 14
   2.2.4 Fully Connected Layer ............................ 15
   2.2.5 Output Layer ...................................... 15
   2.2.6 Dropout ............................................ 16
   2.2.7 Visual Geometry Group ........................... 16
2.3 Data ...................................................... 16
   2.3.1 CIFAR-10 Data set ................................. 17
2.3.2 Accuracy ................................................. 17
2.3.3 Categorical Cross-Entropy Loss ..................... 17

3 Method .................................................. 19
  3.1 Research Project ................................... 19
  3.2 Methodology ....................................... 19
  3.3 Reliability and Validity .............................. 20
    3.3.1 Reliability .................................... 20
    3.3.2 Validity ...................................... 20
  3.4 Ethical considerations .............................. 20

4 Implementation ....................................... 22
  4.1 Model implementation .............................. 22
  4.2 Application Development ......................... 23
    4.2.1 Utilizing Vision ............................. 24
    4.2.2 On-device implementation .................. 24
    4.2.3 On-cloud implementation .................. 26

5 Experimental Setup and Results ..................... 27
  5.1 Hardware ......................................... 27
  5.2 Software and systems ............................ 27
  5.3 Results ......................................... 28
    5.3.1 Results of training the model ............... 28
    5.3.2 On-device ................................... 31
    5.3.3 On-cloud ................................... 33

6 Analysis ............................................... 35

7 Discussion .......................................... 37
  7.1 Challenges and Solutions ......................... 38
1 Introduction

This chapter presents the foundation of the thesis, outlining its value and reviewing similar studies previously undertaken in the field. The problem formulation, along with the research questions, are introduced. Additionally, the motivation for the thesis is explained, along with a discussion of how it contributes to the field. The chapter also outlines the scope and limitations of the research. At the end of the chapter is an overview of each subsequent section of the report.

1.1 Background

In our digital age, mobile devices have evolved beyond their original communication purpose to become powerful generators of vast amounts of data, including user behavior logs and sensor data. This explosion of data has garnered attention from various sectors, particularly in the domain of social media and intelligent applications, where such data can drastically improve usability and user experience. However, collecting and utilizing this data carries significant privacy and legal issues, as it typically involves sensitive personal information [1].

Deep learning technologies have been progressively integrated into mobile applications in response to this challenge, offering promising solutions. Unlike traditional computing paradigms that rely on mobile sensing and cloud computing, implementing deep learning directly on mobile devices provides many advantages. Lower communication bandwidth requirements, reduced demand for cloud computing resources, quicker response times, and improved data privacy are among the most notable benefits [2].

Convolutional Neural Networks (CNNs) are deep learning methods important for image recognition and detection. They are complex systems inspired by how the human brain works; they are especially good at managing large amounts of data. CNNs help reduce the amount of computer power needed by processing 2D images with the help of filters, also called kernels, to produce results. They can work on many levels simultaneously, making them better than simpler systems. CNNs have been used successfully in many areas, such as recognizing faces and emotions, helping self-driving cars, and in robotics [3]. We will explain more about how CNNs work in Section 2.2.

1.2 Related work

In the field of machine learning on mobile devices, several studies have already been conducted. Wang proposed a cooperative machine learning approach called CO-OP, which allows mobile devices to collaboratively learn a shared model while preserving data privacy. This work demonstrates the potential of machine learning and deep learning on smartphones while explaining how decentralized machine learning on mobile devices could be beneficial [1].

Another related study by Yunbin provides a comprehensive review of deep learning techniques on mobile devices. The author discusses various aspects, such as model com-
pression, hardware acceleration, and energy efficiency, which are essential for implementing deep learning models on resource-constrained mobile devices [2]. It is strictly related to our work since our thesis aims to evaluate how compression affects the model’s accuracy and total execution time.

A study by Lili Zhu and Petros Spachos investigated smartphone image classification and evaluated various approaches, including deep learning, traditional machine learning, and transfer learning models. They aimed to determine the best choice for an Android application. Although their study is similar to our project, some differences exist, the main one being working with different operating systems. However, their research offers valuable insights that can benefit our current project [4].

Another related research study, titled "A smartphone-based skin disease classification using mobilenet cnn", was focused on the application of the MobileNet CNN architecture for Android smartphones [5]. This study explored various methodologies to enhance the on-device model’s accuracy, including data preprocessing and data augmentation techniques. The relevance of this study to our research is that it explores porting a CNN to a smartphone, which can offer insights into how deep learning operates on such devices. However, it’s important to note that it employs a different architecture and is targeted explicitly at Android platforms.

The paper examines the advancements in AI applications on mobile devices, mainly focusing on Android. The progress of mobile AI accelerators has enabled mobile devices to achieve capabilities comparable to certain Nvidia GPUs, making it possible to run complex AI models on smartphones. The widespread adoption of deep learning models on smartphones has resulted in various applications. However, the early challenges faced were addressed by developing optimized libraries like TensorFlow Mobile and TensorFlow Lite, specifically designed for mobile inference. The paper concentrates on Android platforms and discusses tools such as the Android Neural Networks API and TFLite delegates. Nevertheless, the paper does not explore the advancements or applications of deep learning on iOS, missing an opportunity to investigate potential differences or make comparisons within this particular context [6].

The distinction of our study lies in its focus on utilizing these methodologies within iOS, which contrasts with the Android-focused approach of research works like those conducted by Lili Zhu and Petros Spachos [4] and the smartphone-based skin disease classification study using MobileNet CNN [5].

A related study, in terms of focusing on iOS, conducted by Owais Qayyum and Melike Şah, evaluates other deep learning architectures when deployed on iOS [7]. It could be useful to compare how different architectures behave when deployed.

These studies provide valuable insights into the challenges and opportunities of deploying machine learning models on mobile devices. Our project will build upon these findings and investigate the performance trade-offs between on-device and on-cloud machine learning for a specific classification task.
1.3 Problem formulation

In this study, we aim to evaluate the implementation of deep learning models, specifically a CNN, on iOS smartphones using the CIFAR-10 data set [8]. The main goal is not to create the most accurate model, but rather assessing the difference in accuracy and overall execution time pre-conversion compared to post-conversion.

We want to explore the feasibility of deploying machine learning models on-device while trying to maintain their accuracy. Our objective is to address the gap in existing research related to machine learning and deep learning models on mobile devices, focusing on the challenges presented by deployment on iOS smartphones. The thesis builds upon previous works, particularly those focusing on smartphone image classification [4, 1, 2].

The evaluation is done by studying the difference of both accuracy and total execution time when the model is run locally on the smartphone compared to it being run in the cloud. In this project the cloud is simulated through an environment on the local computer.

Potential limitations due to hardware and software of smartphones are acknowledged. We hypothesize that on-cloud models will outperform on-device models in terms of accuracy and total execution time. However, we also recognize that data transmission time could impact the total execution time for on-cloud models.

The key questions that guide our study are:

RQ1: How does the performance, in terms of accuracy and execution time, of TensorFlow-based deep learning models differ when deployed on-device (specifically on iOS smartphones) versus when run in a cloud-based environment, considering potential data transmission times?

RQ2: With the advancements in AI technology and the widespread use of powerful smartphones, is it feasible and practical to implement deep learning models on iOS devices?

1.4 Motivation

This project aims to explore the viability of deploying deep learning models directly on mobile devices, a concept often referred to as on-device. The primary motivation for this study is to investigate alternatives to cloud-based model implementations, which traditionally involve transmitting user data to large-scale cloud infrastructures such as AWS or Azure.

Bontouch, the company this project is directed towards, has expressed a keen interest in this investigation. This is driven by a growing demand among businesses to retain control over their data and avoid unnecessary distribution of sensitive information, such as medical records, to third parties.

Despite the surge in mobile machine learning, there is a gap in the existing research regarding comparative evaluations of on-device and on-cloud machine learning implementations. Moreover, there is a lack of academic reports and research focusing on the
iOS platform and Apple’s proprietary libraries. Therefore, it is important to conduct further studies in this area.

Apart from providing privacy and control benefits, on-device machine learning has several other advantages. It eliminates the need for data transmission to the cloud, which can cause delays, thus enabling faster, real-time operations. Additionally, it supports offline processing and does not require an active internet connection. Furthermore, leveraging the high processing power of modern smartphones, companies can develop intelligent apps that prioritize privacy.

Nevertheless, our focus is not solely on these advantages or security aspects. Instead, we aim to provide a balanced view, assessing both the benefits and the challenges associated with on-device deep learning. This study will contribute to a more comprehensive understanding of how on-device machine learning compares to on-cloud implementations, potentially paving the way for the accelerated development of smartphone-based machine learning applications.

1.5 Milestones

Inside Table 1.1 the milestones for the project are presented.

<table>
<thead>
<tr>
<th>M1</th>
<th>Investigate and familiarize ourselves with CNNs and the associated libraries to be utilized in the project.</th>
</tr>
</thead>
<tbody>
<tr>
<td>M2</td>
<td>Implement and run a pre-trained CNN-model with the data set on the laptop.</td>
</tr>
<tr>
<td>M3</td>
<td>Optimize and fine-tune the model.</td>
</tr>
<tr>
<td>M4</td>
<td>Prepare the model for the cloud-environment and connect to Amazon Web Services (AWS).</td>
</tr>
<tr>
<td>M5</td>
<td>Run the model on-cloud, evaluate and analyze the results.</td>
</tr>
<tr>
<td>M6</td>
<td>Compress and prepare the model for the iOS environment.</td>
</tr>
<tr>
<td>M7</td>
<td>Run and test the model on-device, evaluate and analyze the results.</td>
</tr>
<tr>
<td>M8</td>
<td>Analyze all the results, summarize and visualize.</td>
</tr>
</tbody>
</table>

1.6 Scope/Limitation

This study aims to assess the performance of machine learning (ML) models in cloud-based environments as compared to their operation on iOS devices. In this scenario, the Android platform has been excluded from the analysis due to the limited timeframe available, excluding a comprehensive evaluation of both platforms. Additionally, the research employs TensorFlow as the ML library of choice rather than another prominent library, such as PyTorch. Investigating the efficiency of different libraries in terms of compression and execution on smartphones would be a valuable area of insight. However, this is beyond the scope of the current study due to time constraints.

For a more comprehensive understanding, it would be intriguing to evaluate alterna-
tive deep learning architectures, such as MobileNet or SqueezeNet, which are specifically designed for on-device deep learning and machine learning. Incorporating these architectures alongside VGG16 could provide a broader perspective.

Another limitation of the study is that the cloud is simulated using a local computer instead of an actual cloud service. Although this approach may not provide optimal reliability for the results, it was chosen because it replicates the process of transmitting a request.

Due to time constraints within the scope of this thesis, the evaluation of alternative architectures and the utilization of an actual cloud service have been deliberately excluded. These areas have potential for future exploration.

1.7 Target group

This research aims to offer valuable insights and direction for software engineers and organizations interested in implementing machine learning (ML) or deep learning (DL) techniques in either cloud-based or on-device environments. By examining each approach’s benefits, constraints, and obstacles, the study aims to provide knowledge for informed decision-making for individuals and companies. Additionally, the outcome of this investigation seeks to provide lessons learned, helping stakeholders to reduce potential errors and missteps as they seek to implement their ML or DL model.

1.8 Outline

This report is organized into the following chapters:

Chapter 2: Theory - This chapter is intended to provide a thorough understanding of the field and is needed to understand the rest of the report.

Chapter 3: Method - This chapter explains the methodology and how the experiments are carried out.

Chapter 4: Implementation - This chapter aims to give an understanding of the application implementation, model implementation, and cloud implementation. It focuses on the most important code in each of the implementations.

Chapter 5: Experimental Setup and Results - This chapter seeks to inform the reader about the hardware and software (operating system, frameworks, and libraries) that was used. Further, it also presents the results in tables and visualizations.

Chapter 6: Analysis - In this chapter, the results are explained. Here, conclusions are drawn and compared to the hypothesis.

Chapter 7: Discussion - In this chapter, the results and analysis are further explained and discussed, here struggles and challenges are also presented in order for the reader to understand what needed to be changed and thought about during the thesis.
Chapter 8: Conclusion - Here, the thesis project is summarized, and future work is presented.
2 Theory

This chapter gives a detailed explanation of all the theories behind the thesis implementation. This chapter is vital for acquiring the necessary knowledge to understand the rest of the report.

2.1 Software

In order to implement a Convolutional Neural Network model, many different libraries and frameworks are used; these will be explained and presented in this section.

2.1.1 TensorFlow and Keras

To implement deep learning, the TensorFlow library will be utilized. This library is well-regarded for its abstraction and ease of use when developing various machine learning and deep learning models. In the context of this project, a CNN will be constructed. TensorFlow has integrated Keras, a high-level API that simplifies the development of deep learning architectures [9, 10].

Apple has good support for porting machine learning models from third-party libraries (e.g. TensorFlow and PyTorch), which further enables the employment of TensorFlow. Transferring a TensorFlow model to iOS is heavily interconnected with Apple’s machine learning framework called Core ML.

2.1.2 Core ML

In an iOS environment, TensorFlow and Keras models must be converted to .mlmodel format using the Core ML Tools library to be compatible with XCode, the IDE for iOS applications. Core ML is a versatile framework that supports various machine learning models, making it ideal for testing and developing models that run locally on Apple’s smartphones. The framework optimizes the performance of machine learning models by utilizing various smartphone hardware components such as the CPU, GPU, and Apple Neural Engine (ANE). The library, written in Python, is the primary means for converting models for compatibility with iOS devices. By using Core ML Tools, developers can efficiently integrate their TensorFlow models into iOS applications, allowing seamless deployment and enhanced user privacy [11].

2.1.3 Vision

Apple’s Vision framework is a toolset designed to simplify the integration of machine learning tasks into applications, particularly for image analysis. It enables tasks such as image classification, object detection and face recognition. It is compatible with Core ML, enabling use of custom machine learning models for image classification and recognition.
The framework also automates essential image processing tasks, such as resizing and formatting [12].

The result of the classification task is encapsulated in VNClassificationObservation objects. Every observation offers an identifier that is essentially the predicted label, and a confidence score demonstrating the model’s assurance level about the prediction. These outcomes become the foundation for evaluating the performance and precision of the image classification process [12].

2.1.4 Protocol Buffers

Protocol Buffers (Protobuf) is a language-neutral serialization method for structured data, providing an efficient and versatile alternative to formats such as JSON. This project will convert a TensorFlow and Keras model into Apple’s Core ML format, which uses Protobuf for data representation. Protobuf offers many benefits, such as smaller size, faster processing, and compatibility with different programming languages.

In this scenario, we will compress the TensorFlow and Keras model using Core ML, producing a model as a Protocol Buffer (.protobuf) file. This format is language-neutral, making integrating the model into various programming languages simple. In this scenario, we are specifically using Swift.

Upon integrating the Core ML model (with the .mlmodel file extension) into XCode, the Protobuf data generates Swift code that allows interaction with the model. Consequently, the application can execute tasks designated by the model and produce corresponding outputs/predictions [13].

2.2 Convolutional Neural Network

CNNs represent a specialized type of deep learning model within the broader realm of machine learning. These models draw inspiration from the human visual cortex, mimicking how humans perceive complex data patterns. By employing multiple layers and capitalizing on the inherent structure of the input data, CNNs excel in identifying patterns and extracting pertinent features. This sophisticated approach enables CNNs to approximate human perception across various domains, including image recognition, speech processing, and natural language processing. As such, the development and application of CNNs contribute significantly to the ongoing pursuit of artificial intelligence [14, 15]. Figure 2.1 illustrates the basic structure of a CNN that generates three outputs, each with associated probabilities.

2.2.1 Convolutional Layer

In this layer, a matrix known as a kernel or a filter is moved across the input, which in this project is an image. Stride is a parameter in CNNs that controls the step size taken when moving the filter across the input image. It is a numerical value that specifies how many pixels the filter should be slid after each feature map is calculated, see figure 2.2.
Figure 2.1: Shows the architecture of a convolutional neuron network [16]. In the initial convolutional layers, the kernel identifies abstract features such as edges and shapes. As the input progresses through successive convolutional layers, the kernels applied search for increasingly complex features [16]. This process results in the formation of a feature map, which essentially represents the detected features. The computation behind generating a feature map involves element-wise multiplication of the matrices [17, 18].

Figure 2.2: A visualisation of stride 1

A filter can vary in sizes, the particular one used in this project is 3x3, thus the examples are applying that. The first steps in the procedure of producing a feature map can be seen in figures 2.3, 2.4a, 2.4b, and 2.4c and 2.4d [17].
Figure 2.3: The first filter in the procedure and what location it is compared to.
(a) The first calculation of producing the feature map

(b) The second calculation of producing the feature map

(c) The third calculation of producing the feature map

(d) The fourth calculation of producing the feature map

Figure 2.4: Comparison of the feature maps
The scenario above is a perfect map, i.e. that the filter matches the current location in the input that it is being compared to. This will generate an exact copy of the filter inside the feature map, as seen in the matrix below.

\[
\begin{bmatrix}
1 & -1 & -1 \\
-1 & 1 & -1 \\
-1 & -1 & 1 \\
\end{bmatrix}
\]

In a situation where the filter does not perfectly align with the current location in the image, an imprecise match will be generated. A visualisation of this is shown in figures 2.5a, 2.5b, 2.5c, and 2.5d. The entire input image is intentionally retained here, in order to better illustrate the systematic traversal of the filter across current location in the image.

Figure 2.5: Comparison of the feature maps

The result of the calculations above produces a feature map resembling the matrix shown below; which obviously is not a perfect match.

\[
\begin{bmatrix}
-1 & 1 & 1 \\
-1 & -1 & 1 \\
1 & -1 & 1 \\
\end{bmatrix}
\]
In the scenarios illustrated in the previous figure, an issue that may arise is the loss of information around the image edges. This occurs because the convolutional filter has fewer calculations near the edges compared to the center of the image. This information loss can negatively impact the model’s performance and accuracy. One technique to address this problem is padding. In this project, zero-padding is utilized, which involves adding a layer of pixels around the image borders. This allows the filter to perform additional calculations close to the edges. Another advantage of using this technique is that the image dimensions do not shrink too quickly when the feature map is passed on to the pooling layer, which could help mitigate the loss of some features, e.g. an edge that could be important for classifying an image correctly [16]. Pooling layer will be explained in 2.2.3. Zero-padding is one of several padding techniques that can be employed. When the objective of padding is to preserve the input dimensions in the output of a layer, it is referred to as same-padding. Thus, zero-padding is a type of same-padding [19, 3].

In addition to the important convolutional layers, several other components contribute to achieving high accuracy and fast execution time. A few of these are activation functions, pooling, flattening layer and the fully connected layer. These concepts will be discussed in detail in the following sections.

2.2.2 Activation Layer

In a CNN, the activation layer is applied to the feature map generated by the preceding convolution layer, aiming to introduce non-linearity by transforming the input values using a specific activation function. This transformation can enhance the network’s ability to capture complex patterns and relationships within the input data. This non-linear layer, also known as the activation layer, is critical in inducing saturation or imposing output constraints, thereby improving the overall performance of the CNN for a given task.

There are three common types of activation functions used in CNNs, sigmoid, tanh, and Rectified Linear Unit (ReLU). While sigmoid and tanh were widely used for many years, ReLU has gained popularity in recent times due to its superior performance. ReLU is preferred over sigmoid and tanh for several reasons, including its simplicity, reduced likelihood of gradient vanishing, and improved computational efficiency [16]. ReLU is an activation function that operates linearly over a specific range. It directly outputs the input value when it is positive, but returns zero for negative inputs. The mathematical expression of the ReLU function is:

\[ f(x) = \max(x, 0) \]
2.2.3 Pool Layer

The pool layer is a key component of CNNs that plays a critical role in feature selection. It works by down-sampling the local feature maps obtained from the preceding convolutional layer, while retaining the most important features. This technique helps to decrease the total parameters of the network, simplify the model’s complexity, lower the likelihood of overfitting issues during training, and enhance the model’s robustness. Simultaneously, the model’s computation speed experiences an acceleration. This process is accomplished by employing one of various aggregation techniques, e.g., mean-pooling or max pooling, to the local feature maps. The mean-pooling method helps lessen the effect of image noise, though it also harms the image’s structure information. On the other hand, max pooling helps reduce mistakes from convolution while keeping the image’s structure information intact, making it one of the most widely used pooling methods and will be used for this project [16, 18, 15].

Max pooling divides the image into rectangular sub-regions, retaining only the highest value within each sub-region. A frequently employed size for max-pooling is 2x2 [16]. As illustrated in Figure 2.7, when pooling is executed on the top-left 2x2 blocks, it shifts by 2 and concentrates on the top-right portion.

![Figure 2.6: ReLU activation function](image)
2.2.4 Fully Connected Layer

The fully connected layer resembles the arrangement of neurons in a conventional neural network. As a result, every node in such this layer has direct connections to all nodes in both the preceding and the previous layer. The fully connected layer serves as a feature vector, which is formed by sorting deep layer features following the feature extraction procedure that involves multiple convolution and pooling layers. Fully connected layers can also be referenced as Dense layers, as in figure 2.8.

2.2.5 Output Layer

The Output layer is the classifier function to associate the high-level features present in the fully connected layer with the input image’s category probabilities, ultimately producing the classification outcome. This is done with a softmax function, it turns the predictions into non-negative values and normalizes them to get a probability distribution over classes [15].

Figure 2.7: Max Pooling with 2x2 size
2.2.6 Dropout

Dropout, in the context of neural networks, acts as a form of regularization that aids in preventing the model from overfitting to the training data. Overfitting occurs when the model performs exceptionally well on the training data but struggles to generalize on unseen, or test, data. This is often due to the model learning noise or random fluctuations in the training data. The process of dropout involves randomly "dropping" or deactivating a proportion of neurons in a layer during each training step. In effect, this creates a version of the network that is missing some neurons. Since the dropped neurons change randomly from step to step, the learning process can be seen as training a collection of "thinned" networks, with each network having a different architecture. When making predictions, all neurons are used, but their outputs are scaled down by the dropout rate used during training. This scaling down is necessary because during prediction, more neurons are active than during training. The technique of dropout effectively reduces complex co-adaptations of neurons, as it provides a way of making sure that no neuron or group of neurons can exclusively carry significant information. In other words, dropout encourages the distribution of learned features across many neurons, thereby making the model more robust. This is because the model is forced to learn more generic, reusable features, which improves its ability to generalize to new, unseen data. The dropout method has demonstrated its effectiveness in various studies and practical applications, thereby gaining popularity in the field of deep learning.[15, 20]

2.2.7 Visual Geometry Group

Visual Geometry Group (VGG) is a very deep CNN that were proposed by K. Simonyan and A. Zisserman in 2013. VGG-16 is a CNN model composed of 16 layers, encompassing 13 convolutional layers, 5 max-pooling layers and 3 fully connected layers, as seen in figure 2.8. The convolutional layers utilize 3 x 3 filters with a stride of 1 pixel, and a ReLU activation function is applied after each convolution layer to introduce non-linearity. Max-pooling layers, working over a 2 x 2 pixel filter with a stride of 2 pixels, serve to reduce dimensionality. The fully connected layers comprise 4096 channels each, with the final layer designed for a 10-way classification using a softmax activation function [21].

![Figure 2.8: The VGG16 architecture](image)

2.3 Data

In this section, the data set will be explained, including the features, classes and how it has been divided into training and test. Further, the different metrics will also be explained in context of the dataset used.
2.3.1 CIFAR-10 Data set

CIFAR-10 is a widely used data set in the field of machine learning and computer vision for image classification tasks. It was created by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton from the University of Toronto. The data set consists of 60,000 32x32 color images, divided into 10 classes, with 6000 images per class. There are 50000 training images and 10,000 test images [8].

Here are the main features of the CIFAR-10 data set:

1. The images in the CIFAR-10 data set are small, with dimensions of 32x32 pixels. This makes it computationally efficient for training and testing machine learning models, especially for deep learning models like CNNs.

2. The images in the data set are RGB color images, with each pixel having three channels (red, green, and blue).

3. There are 10 classes in the data set, which are:
   - Airplane
   - Automobile
   - Bird
   - Cat
   - Deer
   - Dog
   - Frog
   - Horse
   - Ship
   - Truck

4. The data set is balanced, with an equal number of images (6000) in each class. This helps in training machine learning models that can generalize well across all the classes.

5. The data set comes pre-split into a training set (50000 images) and a test set (10,000 images). This makes it easy to train and evaluate machine learning models using the standard practices of splitting data into training and testing subsets.

6. The images in the data set are real-world images, which have been preprocessed and downsampled to a smaller size. This makes the data set a suitable benchmark for image classification tasks on real-world data.

2.3.2 Accuracy

In the context of machine learning, accuracy is a measurement of a models performance, expressed as the percentage or ratio of correct predictions made by the model relative to the total number of predictions it has made. A higher accuracy score indicates better performance by the model in correctly predicting the outcomes [22].

2.3.3 Categorical Cross-Entropy Loss

Categorical Cross-Entropy Loss is a loss function used in machine learning for multi-class classification problems. This function measures the dissimilarity between the model’s
predicted probability distribution across the classes, and the actual distribution. In essence, it calculates how well the model’s confidence about the classes aligns with the true outcome [23].

For a given input, the model will output a set of probabilities - one for each class. The Cross-Entropy Loss is then computed by taking the negative logarithm of the probability predicted for the correct class. This means the loss is lower when the model’s confidence in the correct class is higher, and vice versa. Therefore, this function pushes the model to make correct predictions with higher confidence [23].
3 Method

This section explains our research methodology, primarily a controlled experiment preceded by a literature review. The literature review forms the initial phase, laying the groundwork for the experiment by offering a comprehensive understanding of the field. The controlled experiment then examines the impact of different platforms, on-cloud vs. on-device, on the model’s performance. The results will be presented and visually displayed.

3.1 Research Project

Our research project aims to investigate the performance of CNNs on-cloud and on-device using a controlled experimental approach. The objective is to understand how the environmental change impacts the performance indicators model accuracy and total execution time.

To lay the groundwork for the controlled experiment, we initially reviewed the literature using specific keywords related to our research topic. These keywords, listed in Table 3.2, were chosen due to their relevance to the project’s techniques and goal.

Table 3.2 contains estimated search results from Google Scholar, IEEE Xplore, and OneSearch. We chose articles based mainly on their publication year and impact factors. We focused on articles published after 2017, with a minimum impact factor of 1.5. However, due to the specificity of our research topic, it was occasionally challenging to find articles that fulfilled these criteria. In such instances, we also considered articles with lower impact factors to ensure a comprehensive understanding of the field. This approach ensured that our research was grounded in relevant and impactful studies. To establish a comprehensive search, we utilized various search engines, including Google Scholar, IEEE Xplore and OneSearch. Using these search engines, we could access multiple articles and publications relevant to our research topic.

Table 3.2: Table of keywords and hits

<table>
<thead>
<tr>
<th>Keyword</th>
<th>≈Number of hits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutional Neural Network</td>
<td>1 040 000</td>
</tr>
<tr>
<td>Deep learning smartphones</td>
<td>273 000</td>
</tr>
<tr>
<td>Image classification CNN</td>
<td>7 880</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>210 000</td>
</tr>
<tr>
<td>Keras</td>
<td>672 000</td>
</tr>
<tr>
<td>Protocol buffers</td>
<td>554 000</td>
</tr>
<tr>
<td>VGG16</td>
<td>60 100</td>
</tr>
<tr>
<td>Cifar-10</td>
<td>41 900</td>
</tr>
</tbody>
</table>

3.2 Methodology

This project’s primary method will be a controlled experiment, utilizing knowledge gained from a literature review conducted during the initial phases. This literature review will
allow for a better understanding of CNNs and associated libraries. It will also provide valuable insight for preparing the model for on-cloud and on-device.

The controlled experiment will involve alternating the running environment as an independent variable (cloud and iOS) while measuring the impact on dependent variables such as model accuracy and total execution time. The experimental process will include the following:

- Implementing and running the model.
- Optimizing it.
- Preparing and testing it in the different environments (iOS and cloud).
- Evaluating and analyzing the results.

In the final stages of the project, we will summarize our findings and present them through visual representations. By following this approach, we ensure that our project is based on extensive research and provides a comprehensive understanding of the methods we used.

3.3 Reliability and Validity

In this subsection, we address the concerns of reliability, validity and ethical considerations related to our research project. We outline our expectations for the consistency of results, the authenticity of the data generated and the ethical implications of our research process.

3.3.1 Reliability

In this project, we anticipate that other individuals will achieve consistent accuracy when running it on devices with equivalent computational power. Furthermore, using the same model is crucial to ensure the reproduction of identical results.

3.3.2 Validity

In this project, the results and discussions will be based on the numerical data generated by the running models. It ensures that the presented numbers are authentic and have not been manipulated. Additionally, graphs and figures will support and substantiate claims made within the results and discussions. It will help the readers to better grasp the results.

3.4 Ethical considerations

The project requires no major ethical considerations since it involves no personal information. The CIFAR-10 data set, which is publicly accessible, is maintained by the University
of Toronto 2.2.7. The considerations that are taken into account is to credit the creator of the open source model that is used and referring all the works from which information and knowledge has been retrieved.
4 Implementation

This chapter explains the steps for implementing the model, the iOS application, and the cloud environment.

4.1 Model implementation

In this project, we implemented a CNN based on the VGG16 architecture with minor modifications. Figure 4.9 and Figure 4.10 highlight the most significant changes. We integrated Dropout, as shown in Figure 4.9, which deviates from the conventional VGG16 architecture. We introduced this modification as it features in the open-source repository that serves as the foundation for this project [20]. Furthermore, we slightly modified the CNN’s final part - we included only two Dense layers instead of the traditional three found in the VGG16 architecture. This adjustment is evident in Figure 4.10, where only two Dense layers are visible.

The first layer is displayed in Figure 4.9. Here, the first line of the code introduces a 2D convolution layer to the network with 64 outputs, which are feature maps, each having a size of 3x3, and these are produced by filters striding over the input. The layer takes an input image with the shape defined by self.x_shape. Padding is applied to keep the output size the same as the input. The layer also utilizes L2 regularization with the penalty term determined by the weight_decay parameter.

Next, a ReLU activation function is added. The ReLU function introduces non-linearity to the model by outputting the input directly if it is positive, and zero otherwise. Following the activation function, Batch Normalization is applied. This technique standardizes the inputs to a layer for each mini-batch, which stabilizes the learning process and reduces the number of training epochs required. A Dropout layer is then added, which randomly sets a fraction of the input units to 0 at each update during training time, helping to prevent overfitting. In this instance, 30% of the input units will be dropped out. A second 2D convolution layer identical to the first is then added to the model. This is again followed by a ReLU activation function and Batch Normalization. Finally, a 2D max pooling layer is added. Max pooling is a process that aims to reduce the dimensionality of the input, which helps to speed up computation and also assists in reducing overfitting.
model.add(Conv2D(64, (3, 3), padding='same',
        input_shape=self.x_shape,
        kernel_regularizer=regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Dropout(0.3))

model.add(Conv2D(64, (3, 3), padding='same', kernel_regularizer=
        regularizers.l2(weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2, 2)))

Figure 4.9: The first convolutional layer followed by the pooling layer.

model.add(Flatten())
model.add(Dense(512, kernel_regularizer=regularizers.l2( 
        weight_decay)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(self.num_classes))
model.add(Activation('softmax'))

Figure 4.10: The fully connected layer.

4.2 Application Development

In the project, an iOS application was developed in order to run the model on-device and call the model on-cloud. The application was built in Xcode [24] with Swift. The app was supposed to be simple and minimalistic since the aim of the project was evaluation of the CNN. The UI is two separate start-buttons, one to start the model on-device and one to call the on-cloud model, as shown in Figure 4.11.
4.2.1 Utilizing Vision

To implement image classification models within the Vision environment, a VNCoreML-Request object needs to be established. This object packages the Core ML model, preparing it for Vision-oriented image classification tasks. Once the model has been added to the request object, it is passed onto a handler to carry out the classification task [12]. The VNImageRequestHandler class is responsible for the execution of the Vision request on a single image. The handler takes the request, which includes the model and the image and orchestrates the classification process [12].

4.2.2 On-device implementation

The test data set is placed inside the Xcode project, and the images are fetched before the model starts classifying. In order to process and categorize images, it is necessary to instantiate the Core ML model as a Vision model, as shown in Figure 4.12. This code firstly creates a Core ML model called cifar of the class model300, which is the class generated from the conversion to a Core ML file of the TensorFlow and Keras model. This function returns a Vision instance of the Core ML-model called imageClassifierVisionModel.
In Figure 4.13, the on-device image classification logic is presented. The function parameters are a tuple array, where each tuple consists of an image and its actual label, and a VNCoreMLRequest object. The function goes through each image, classifying it using the provided Core ML request. The VNCoreMLRequest is a predefined class that wraps a Core ML model for use with Vision requests. For every image, a VNImageRequestHandler instance is created. This class is a helper that performs a Vision request on a single image, i.e. a classification. The request is performed on the image and the results are obtained, expected to be of type VNClassificationObservation. Each observation includes an identifier that represents the predicted label for the image. If this predicted label matches the actual label of the image, the count of correct predictions is incremented.

```swift
func createImageClassifier() -> VNCoreMLModel {
    var cifar: model300?
    do {
        cifar = try model300(configuration: MLModelConfiguration())
        print(cifar?.model.modelDescription)
    } catch {
        print("Failed to load model: \(error)"")
    }

    guard let imageClassifier = cifar else {
        fatalError("App failed to create an image classifier model instance.")
    }

    // Get the underlying model instance.
    let imageClassifierModel = imageClassifier.model

    // Create a Vision instance using the image classifier’s model instance.
    guard let imageClassifierVisionModel = try? VNCoreMLModel(for: imageClassifierModel) else {
        fatalError("App failed to create a 'VNCoreMLModel' instance.")
    }

    return imageClassifierVisionModel
}
```

Figure 4.12: Function for creating a Vision instance of the Core ML-model.
func classify(imagesAndLabels: [(image: UIImage, label: String)], request: VNCoreMLRequest) {
    var correctPredictions = 0
    let totalImages = imagesAndLabels.count

    for (image, trueLabel) in imagesAndLabels {
        let handler = VNImageRequestHandler(cgImage: image.cgImage!)
        do {
            try handler.perform([request])
            if let observation = request.results?.first as? VNClassificationObservation {
                let predictedLabel = observation.identifier
                if predictedLabel == trueLabel {
                    correctPredictions += 1
                }
                print("\(trueLabel),\(predictedLabel),\(observation.confidence)"
            }
        } catch {
            print("Error performing classification request: ",
                  error)
        }
    }

    let accuracy = Double(correctPredictions) / Double(totalImages)
    print("Accuracy: \(accuracy * 100)%")
}

Figure 4.13: Function for classification on-device.

4.2.3 On-cloud implementation

The cloud is an HTTP server created using Flask, a web framework written in Python. It offers a restful API, providing an interface that receives POST requests at the endpoint. Upon receiving a POST request with a file, the server validates the request ensuring that the file part exists and a file is attached. Provided that the file is valid, the server anticipates it to be a ZIP archive consisting of images. It extracts and processes these images, preparing them for classification for our TensorFlow model. Once all images are classified, the server computes the accuracy of the classification by comparing the model’s predictions to the actual labels. It then returns a summary of the classification accuracy to the client.
5 Experimental Setup and Results

This chapter explains the experimental setup, containing hardware specifications, libraries, frameworks and additional software tools. Furthermore, it presents the results obtained throughout the thesis.

5.1 Hardware

We carried out all development and training tasks for this project on an Apple MacBook. This machine was equipped with an M1 processor, 64GB of RAM, a 1TB SSD, and ran on the Ventura 13.3 macOS. Due to the hardware architecture of the MacBook, the conventional installation of TensorFlow and Keras was not immediately possible. As a result, we had to establish a dedicated Conda environment. We achieved this setup by following an online guide [25]. Based on this experience, we predict that all users of Apple MacBook models with an M1 processor or newer will need to utilize this guide for their respective environment setups.

In addition to the MacBook, we employed an iPhone 14 as a part of our experimental setup. It allowed us to test the functionality and performance of our machine learning model on-device, thereby giving us a more comprehensive understanding of its capabilities across the different environments.

We have opted not to provide an step-by-step guide here as its necessity will depend on individual hardware configurations, and the linked guide is thorough, well-structured and provides clear instructions for those who require it [25].

5.2 Software and systems

We employed Python with TensorFlow and Keras for model development within the Integrated Development Environment (IDE) of VSCode. To create the iOS application, we leveraged Swift within Apple's IDE, Xcode. The integrated libraries in Xcode, including Vision and SwiftUI, were instrumental in the development process. Furthermore, we utilized Python and Pandas, Matplotlib and Seaborn for result visualization. We performed all configurations on a MacBook running the Mac Operating System version Ventura 13.3. To establish a REST API and locally simulate a cloud environment on the computer, we employed Flask. Table 5.3 displays all mentioned libraries, frameworks, and their corresponding versions.
Table 5.3: List of utilized Software and Libraries with their versions

<table>
<thead>
<tr>
<th>Libraries and Frameworks</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>pip</td>
<td>22.3.1</td>
</tr>
<tr>
<td>TensorFlow</td>
<td>2.12.0</td>
</tr>
<tr>
<td>Keras</td>
<td>2.12.0</td>
</tr>
<tr>
<td>NumPy</td>
<td>1.23.0</td>
</tr>
<tr>
<td>Coremltools</td>
<td>6.3.0</td>
</tr>
<tr>
<td>Seaborn</td>
<td>0.12.2</td>
</tr>
<tr>
<td>Pandas</td>
<td>2.0.0</td>
</tr>
<tr>
<td>Matplotlib</td>
<td>3.7.1</td>
</tr>
<tr>
<td>Flask</td>
<td>2.3.2</td>
</tr>
</tbody>
</table>

5.3 Results

This section shows the performance of models in the different environments: Subsection 5.3.1 for TensorFlow and Keras on the computer, 5.3.2 for the Core ML model on-device and 5.3.3 for TensorFlow and Keras in the cloud.

5.3.1 Results of training the model

The MacBook is used for the entire model development and training process, serving as the starting point for evaluation. This setup is practical as it enables overseeing the model’s training progress, parameter adjustments, and iterative performance testing. Real-time access to these development stages facilitates quick iterations and changes.

The performance of the TensorFlow and Keras models was observed and analyzed over various numbers of training epochs. The model was initially run on the test set for 100 epochs, yielding an accuracy of 90.44%. This accuracy served as a baseline measure of the model’s ability to classify the data from the given data set correctly. Afterward, the training was extended to 300 epochs, and the model slightly improved its accuracy rate, achieving 93.3%. These accuracies indicate that the model’s accuracy could be enhanced with additional training, as can be seen of the result in the confusion matrices 5.14a and Figure 5.14b.
During the training process, we made a significant observation around the 175th epoch, where the model’s accuracy rate noticeably declined, leading to stagnation. It can be seen in Figure 5.15a and Figure 5.16a. The categorical cross-entropy loss is shown in Figure 5.15b and Figure 5.16b. We noticed that the loss stabilized around the 130th
epoch, implying that the model had achieved its optimal state. This observation led us to conclude that running the model for only 100 epochs would be insufficient to minimize the categorical cross-entropy loss.

![Accuracy Curves](image1.png)

(a) Accuracy after 100 epochs

![Loss Curves](image2.png)

(b) Loss after 100 epochs

Figure 5.15: Accuracy and loss for 100 epochs
5.3.2 On-device

The confusion matrix depicted in Figure 5.17 represents the outcome of executing the Core ML model on the iOS smartphone. It achieves 93.25% accuracy and takes a total average of 45.64 seconds for the whole data set, as seen in Table 5.4. It displays the outcomes of ten trial runs, which compute the mean execution time of the model on various subsets of the test set. The subsets were taken at 10% intervals, beginning at 10% and increasing to 100%.
Figure 5.17: Confusion matrix of the 300 epochs model, post conversion, when run locally on the iOS smartphone

Table 5.4: Execution time (seconds) for runs on-device

<table>
<thead>
<tr>
<th>#Run</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.78</td>
<td>5.61</td>
<td>7.74</td>
<td>10.42</td>
<td>12.24</td>
<td>20.72</td>
<td>25.02</td>
<td>30.03</td>
<td>36.03</td>
<td>44.50</td>
</tr>
<tr>
<td>2</td>
<td>3.18</td>
<td>5.43</td>
<td>7.46</td>
<td>10.54</td>
<td>13.06</td>
<td>21.24</td>
<td>25.64</td>
<td>31.30</td>
<td>36.15</td>
<td>46.92</td>
</tr>
<tr>
<td>3</td>
<td>3.03</td>
<td>5.08</td>
<td>8.45</td>
<td>9.95</td>
<td>13.30</td>
<td>20.28</td>
<td>26.67</td>
<td>30.01</td>
<td>36.33</td>
<td>46.36</td>
</tr>
<tr>
<td>4</td>
<td>3.16</td>
<td>5.03</td>
<td>8.66</td>
<td>10.81</td>
<td>13.70</td>
<td>20.13</td>
<td>25.57</td>
<td>30.76</td>
<td>37.34</td>
<td>45.57</td>
</tr>
<tr>
<td>5</td>
<td>3.03</td>
<td>5.07</td>
<td>8.73</td>
<td>10.98</td>
<td>13.96</td>
<td>17.85</td>
<td>26.50</td>
<td>30.69</td>
<td>36.86</td>
<td>45.56</td>
</tr>
<tr>
<td>6</td>
<td>3.02</td>
<td>5.03</td>
<td>8.65</td>
<td>9.77</td>
<td>13.81</td>
<td>17.33</td>
<td>26.94</td>
<td>31.08</td>
<td>36.77</td>
<td>45.37</td>
</tr>
<tr>
<td>7</td>
<td>3.16</td>
<td>5.03</td>
<td>8.45</td>
<td>10.78</td>
<td>13.61</td>
<td>17.34</td>
<td>26.86</td>
<td>30.19</td>
<td>35.55</td>
<td>44.91</td>
</tr>
<tr>
<td>8</td>
<td>3.04</td>
<td>5.03</td>
<td>8.68</td>
<td>10.81</td>
<td>13.67</td>
<td>18.39</td>
<td>26.84</td>
<td>31.23</td>
<td>37.20</td>
<td>45.70</td>
</tr>
<tr>
<td>9</td>
<td>2.94</td>
<td>5.10</td>
<td>8.83</td>
<td>10.87</td>
<td>13.71</td>
<td>18.19</td>
<td>26.74</td>
<td>30.05</td>
<td>37.05</td>
<td>45.45</td>
</tr>
<tr>
<td>10</td>
<td>2.91</td>
<td>5.38</td>
<td>7.48</td>
<td>10.02</td>
<td>13.35</td>
<td>18.42</td>
<td>26.03</td>
<td>31.00</td>
<td>37.89</td>
<td>45.51</td>
</tr>
<tr>
<td></td>
<td><strong>Average</strong></td>
<td><strong>5.18</strong></td>
<td><strong>8.32</strong></td>
<td><strong>10.50</strong></td>
<td><strong>13.34</strong></td>
<td><strong>19.05</strong></td>
<td><strong>25.98</strong></td>
<td><strong>30.54</strong></td>
<td><strong>36.12</strong></td>
<td><strong>45.64</strong></td>
</tr>
</tbody>
</table>
5.3.3 On-cloud

The confusion matrix in Figure 5.18 represents the outcome of executing the TensorFlow model on the cloud. It achieves 93.3% accuracy and takes a total average of 4.55 seconds to give an response for the whole data set.

![Confusion Matrix Image]

Figure 5.18: Confusion matrix of the 300 epochs model, before conversion, when run on the cloud

Table 5.5 shows the response time for ten different runs on the cloud environment. Each run has measurements for response time. The values in the Table 5.5 represent the time it took for the application to get a response at each subset of the test set. For example, when the system classified a subset of 10% from the test set, it took 0.77 seconds for the application to receive a response.

The second Table 5.6 presents similar information but for execution time, which refers to the time it takes for the model on the cloud to classify all the images of the request.
Table 5.5: Response time for runs on cloud

<table>
<thead>
<tr>
<th>#Run</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.77</td>
<td>1.18</td>
<td>1.57</td>
<td>2.04</td>
<td>2.94</td>
<td>3.34</td>
<td>3.65</td>
<td>5.03</td>
<td>4.45</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.61</td>
<td>1.23</td>
<td>1.49</td>
<td>1.82</td>
<td>2.12</td>
<td>2.61</td>
<td>3.07</td>
<td>3.45</td>
<td>3.94</td>
<td>4.04</td>
</tr>
<tr>
<td>3</td>
<td>0.86</td>
<td>0.99</td>
<td>1.22</td>
<td>1.93</td>
<td>2.42</td>
<td>2.51</td>
<td>2.87</td>
<td>4.54</td>
<td>4.03</td>
<td>4.68</td>
</tr>
<tr>
<td>4</td>
<td>0.56</td>
<td>0.97</td>
<td>1.42</td>
<td>1.80</td>
<td>2.47</td>
<td>2.66</td>
<td>3.01</td>
<td>4.16</td>
<td>4.92</td>
<td>4.73</td>
</tr>
<tr>
<td>5</td>
<td>0.58</td>
<td>1.00</td>
<td>1.52</td>
<td>2.12</td>
<td>2.37</td>
<td>2.56</td>
<td>3.17</td>
<td>4.10</td>
<td>4.14</td>
<td>4.42</td>
</tr>
<tr>
<td>6</td>
<td>0.57</td>
<td>1.35</td>
<td>1.81</td>
<td>1.85</td>
<td>2.29</td>
<td>2.46</td>
<td>3.37</td>
<td>3.49</td>
<td>3.73</td>
<td>4.36</td>
</tr>
<tr>
<td>7</td>
<td>0.79</td>
<td>0.99</td>
<td>1.40</td>
<td>2.30</td>
<td>2.22</td>
<td>2.53</td>
<td>2.97</td>
<td>3.34</td>
<td>4.01</td>
<td>4.90</td>
</tr>
<tr>
<td>8</td>
<td>0.52</td>
<td>1.03</td>
<td>1.40</td>
<td>1.85</td>
<td>2.45</td>
<td>2.55</td>
<td>3.80</td>
<td>3.35</td>
<td>4.03</td>
<td>5.07</td>
</tr>
<tr>
<td>9</td>
<td>0.56</td>
<td>1.27</td>
<td>1.88</td>
<td>2.29</td>
<td>2.44</td>
<td>2.41</td>
<td>2.91</td>
<td>3.33</td>
<td>4.79</td>
<td>4.29</td>
</tr>
<tr>
<td>10</td>
<td>0.60</td>
<td>1.22</td>
<td>1.40</td>
<td>2.34</td>
<td>2.47</td>
<td>2.68</td>
<td>2.97</td>
<td>3.37</td>
<td>3.95</td>
<td>4.53</td>
</tr>
<tr>
<td>Average</td>
<td>0.60</td>
<td>1.15</td>
<td>1.41</td>
<td>2.03</td>
<td>2.31</td>
<td>2.54</td>
<td>3.05</td>
<td>3.62</td>
<td>4.26</td>
<td>4.55</td>
</tr>
</tbody>
</table>

Table 5.6: Execution time for runs on cloud

<table>
<thead>
<tr>
<th>#Run</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.44</td>
<td>0.82</td>
<td>1.18</td>
<td>1.47</td>
<td>2.02</td>
<td>2.34</td>
<td>2.75</td>
<td>2.98</td>
<td>3.48</td>
<td>3.66</td>
</tr>
<tr>
<td>2</td>
<td>0.42</td>
<td>0.75</td>
<td>1.12</td>
<td>1.53</td>
<td>1.74</td>
<td>2.21</td>
<td>2.58</td>
<td>2.85</td>
<td>3.13</td>
<td>3.55</td>
</tr>
<tr>
<td>3</td>
<td>0.46</td>
<td>0.74</td>
<td>1.02</td>
<td>1.49</td>
<td>1.78</td>
<td>2.13</td>
<td>2.46</td>
<td>2.91</td>
<td>3.15</td>
<td>3.49</td>
</tr>
<tr>
<td>4</td>
<td>0.39</td>
<td>0.75</td>
<td>1.13</td>
<td>1.48</td>
<td>1.80</td>
<td>2.27</td>
<td>2.54</td>
<td>2.89</td>
<td>3.25</td>
<td>3.57</td>
</tr>
<tr>
<td>5</td>
<td>0.42</td>
<td>0.75</td>
<td>1.08</td>
<td>1.43</td>
<td>1.79</td>
<td>2.19</td>
<td>2.54</td>
<td>2.84</td>
<td>3.09</td>
<td>3.53</td>
</tr>
<tr>
<td>6</td>
<td>0.41</td>
<td>0.74</td>
<td>1.11</td>
<td>1.49</td>
<td>1.86</td>
<td>2.15</td>
<td>2.51</td>
<td>2.91</td>
<td>3.13</td>
<td>3.61</td>
</tr>
<tr>
<td>7</td>
<td>0.44</td>
<td>0.75</td>
<td>1.15</td>
<td>1.43</td>
<td>1.82</td>
<td>2.14</td>
<td>2.44</td>
<td>2.93</td>
<td>3.14</td>
<td>3.62</td>
</tr>
<tr>
<td>8</td>
<td>0.42</td>
<td>0.74</td>
<td>1.09</td>
<td>1.48</td>
<td>1.85</td>
<td>2.13</td>
<td>2.53</td>
<td>2.83</td>
<td>3.19</td>
<td>3.60</td>
</tr>
<tr>
<td>9</td>
<td>0.39</td>
<td>0.74</td>
<td>1.08</td>
<td>1.44</td>
<td>1.74</td>
<td>2.08</td>
<td>2.51</td>
<td>2.84</td>
<td>3.23</td>
<td>3.44</td>
</tr>
<tr>
<td>10</td>
<td>0.40</td>
<td>0.74</td>
<td>1.12</td>
<td>1.48</td>
<td>1.78</td>
<td>2.09</td>
<td>2.56</td>
<td>2.90</td>
<td>3.15</td>
<td>3.55</td>
</tr>
<tr>
<td>Average</td>
<td>0.41</td>
<td>0.77</td>
<td>1.11</td>
<td>1.48</td>
<td>1.82</td>
<td>2.17</td>
<td>2.55</td>
<td>2.86</td>
<td>3.22</td>
<td>3.52</td>
</tr>
</tbody>
</table>
6 Analysis

We observed high accuracy on-cloud and on-device; they reached 93.3%, respectively 93.25%. These high accuracy scores indicate that the VGG16 model effectively learned and generalized from the data set. As expected, the accuracy of model aligns with our predictions, given the widespread use of the data set. However, we noted a significant difference in execution time between the cloud and on-device environments. The average total execution time on the cloud is 4.55 seconds, whereas the on-device time averaged 45.64 seconds for the whole data set. We recognize several potential factors that could account for these differences:

- **Hardware Differences**: The advanced hardware in the MacBook hosting the cloud environment is likely more powerful than the hardware in smartphones. The processing power, memory, and storage capabilities could impact the execution time.

- **Software Optimization**: Differences in software optimization between TensorFlow (used in the cloud) and Core ML (used on-device) could also affect the speed of execution. Optimization techniques used by TensorFlow may not be fully compatible or effective with Core ML, leading to slower performance on-device.

- **Model Conversion**: Converting the model from TensorFlow to Core ML may not preserve all optimizations. This could potentially lead to inefficiencies that increase the total execution time on-device.

- **Computational Resources**: Despite network latency, the computational resources available in a cloud environment could result in faster execution times. This is because cloud platforms can allocate more resources to the task, allowing for parallel processing and quicker completion of tasks.

Figure 6.19 displays the total execution time for the on-device and the on-cloud models when tasked with running different test set sizes. The models were evaluated on subsets of the test set at intervals of 10%, starting from 10% and progressing up to 100%. Interestingly, the graph exhibits a non-linear relationship between the total execution time and the size of the test set. Around 5000 images, the curve becomes noticeably steeper, suggesting potential factors influencing the observed behavior. One possible explanation is the strain on the smartphone’s memory as the model handles larger amounts of data. Resource-constrained devices like smartphones typically have limited memory capacity, and as the test set size increases, the model may require more memory to process and store intermediate results. This increased memory usage can contribute to longer execution times. Another factor contributing to the steeper curve is the model’s optimization for on-device execution. If the model has not been specifically optimized to handle larger data set efficiently, it may experience performance degradation when confronted with many images. In order to gain a deeper understanding of the underlying causes, further analysis should be done. It would be useful to investigate the memory usage patterns of the on-device model as the test set size increases. Identifying memory bottlenecks or exploring memory optimization techniques could help relieve the impact of memory constraints on execution time. Examining the model architecture and exploring ways to enhance its efficiency for on-device execution might improve performance. Techniques
like model compression, quantization, or specialized hardware acceleration could also be explored to optimize the model’s resource utilization and reduce execution time. Delving into these aspects and considering memory limitations and model optimization could help improve, understand, and address the observed non-linear relationship between the total execution time and the test set size on the device.

Figure 6.19: Comparison between on-devices and on-clouds total execution time on different amount of images
7 Discussion

Our results prove the viability of deploying deep learning models on smartphones. iOS and Android can deploy deep learning models without sacrificing accuracy, as demonstrated by ours and previous works [4, 5].

Moreover, we recognize that the use case of deep learning on a smartphone typically involves fewer images than 10,000. However, this project aims to assess the feasibility of deploying a typically complex cloud-based deep learning model on an iOS smartphone. Based on our research, it is viable to utilize this approach as long as there is no critical need for instant results. In scenarios where large volumes of sensitive data need to be processed and the overall execution time is unimportant, utilizing an on-device deep CNN could be a viable solution. On-device models could eliminate the need to transfer data to companies like Amazon, Microsoft, Google, or other major corporations. The privacy of individuals will be improved as there is no need to transmit their information to on-cloud models.

Building on the previous discussions, several industries can benefit from this technology. It not only guarantees data privacy but also supports on-device processing.

1. Healthcare: The field of medical imaging could utilize machine learning models trained to detect anomalies such as tumors. On-device data processing can uphold patient privacy and prove beneficial in remote areas with unstable internet connectivity.

2. Finance and Banking: Using fraud detection models on mobile devices can improve security by reducing the risks of transmitting sensitive data. It is crucial for mobile banking apps, where protecting user privacy is essential.

3. Education: Personalized learning can leverage deep learning to tailor educational content to individual student needs. Implementing this on-device can improve student privacy.

4. Agriculture: Farmers could utilize on-device deep learning to identify plant disease or pest infestation, offering quick results and functioning efficiently in areas with limited internet access.

5. Defense and Security: On-device facial or object recognition software can be employed for surveillance, ensuring data security and privacy while providing recognition results.

6. Retail: In physical stores, deep learning models can be used to analyze customer behavior and optimize store layouts or personalize shopping experiences. On-device processing can help maintain customer privacy.

These are simply a selection of sectors where on-device deep learning could leverage to potentially enhance. Naturally, the specific accuracy of the employed model must be considered in each scenario, as some procedures demand high accuracy. For instance, errors could have severe consequences in the Defense and Security sector.
Ultimately, our two research questions asked in beginning of the report can be answered.

How does the performance, in terms of accuracy and execution time, of TensorFlow-based deep learning models differ when deployed on-device (specifically on iOS smartphones) versus when run in a cloud-based environment, considering potential data transmission times?

With the advancements in AI technology and the widespread use of powerful smartphones, is it feasible and practical to implement deep learning models on iOS devices?

The accuracy of a deep learning model, when deployed on iOS, is nearly comparable, with a slight difference of just 0.5% post-implementation and rigorous experimentation. However, the overall execution time tends to escalate, although the precise extent is challenging to predict. One certainty is that processing large data volumes will be slower.

In conclusion, our research demonstrates the viability of implementing deep learning models, specifically CNNs, on iOS smartphones, given certain conditions. While the potential of these on-device models is indeed considerable, their practical applications are significantly influenced by the project’s specific goals. The on-device approach might encounter limitations in real-time data processing, where large volumes of data are expected to be analyzed instantaneously. Due to smartphones’ hardware and software constraints, the processing speed cannot match that of powerful cloud servers. Thus, relying on on-device deep learning might pose a challenge for real-time applications where speed is of the essence, such as certain types of interactive AI apps or real-time video processing. On the other hand, deploying deep learning models on iOS smartphones can be highly advantageous in scenarios involving large amounts of sensitive data. The primary reason is that the data processed on-device doesn’t need to leave the user’s device, offering enhanced privacy protection. It circumvents the necessity of transmitting potentially sensitive data to a cloud server, reducing the risk of data breaches or misuse of personal information. It could be crucial in healthcare applications, individual finance management apps, and other services where users’ sensitive data is processed. Furthermore, on-device deep learning models offer potential benefits such as lower data transmission costs and improved operation in environments with low or unstable internet connectivity. However, these benefits must be balanced with the limitations of on-device processing, such as potentially increased battery consumption. Therefore, the decision to employ on-device deep learning must be made on a case-by-case basis, considering the specific needs and constraints of the project. Future research may explore ways to optimize these models and further reduce the execution time, improving the feasibility of their widespread adoption.

7.1 Challenges and Solutions

While developing the model using TensorFlow and Keras, we experienced minimal complications. The abstraction provided by these libraries greatly simplified the task of implementing and optimizing deep learning models. The conversion of our model to Core ML also proved to be straightforward, with a few issues encountered. However, upon integrating and executing the model within Swift and Xcode, we noticed a significant decrease in
accuracy. Aside from the difficulties we faced in getting the the model to operate within Swift and Xcode, we also encountered challenges in setting up a functional Swift project. These struggles stemmed from our unfamiliarity with Swift as a language and Xcode as a development environment.

On closer inspection, we identified the problem associated with data normalization in the TensorFlow and Keras model. This procedure involved subtracting the mean RGB value from each pixel. However, due to limitations in parameters, this normalization step posed challenges when converting to Core ML. In hindsight, it became evident that the issue arose because the Core ML model was not processing its inputs like during the training phase in the TensorFlow and Keras model. This discrepancy in data handling between the training and application stages led to the observed performance difference.

The script in Figure 7.20 represents the finalized method of normalizing data during training. In contrast, Figure 7.21 illustrates how we instruct the Core ML model to handle input data. It is the same way of normalizing, which we found to help the Core ML model produce the expected results. The script in Figure 7.22 shows the initial normalization process of the training data. It was the root of the struggles to reproduce it during conversion to coreml.

```python
def predict(self,x,normalize=True,batch_size=50):
    if normalize:
        x = x / 255.0
    return self.model.predict(x,batch_size)
```

Figure 7.20: The final way of normalizing.

```python
# Convert the model
coreml_model = ct.converters.convert(tf_model,
    inputs=[ct.ImageType(shape=(1, 32, 32, 3), color_layout=ct.colorlayout.RGB, scale=1/255.0,)],
    classifier_config = ct.ClassifierConfig(class_labels))
```

Figure 7.21: The conversion to Core ML.
def normalize(self, X_train, X_test):
    # this function normalize inputs for zero mean and unit
    # variance
    # it is used when training a model.
    # Input: training set and test set
    # Output: normalized training set and test set according to
    # the training set statistics.
    mean = np.mean(X_train, axis=(0, 1, 2, 3))
    std = np.std(X_train, axis=(0, 1, 2, 3))
    X_train = (X_train - mean) / (std + 1e-7)
    X_test = (X_test - mean) / (std + 1e-7)
    return X_train, X_test

Figure 7.22: The initial way of normalization.

During the implementation phase, we faced a significant challenge while deploying the
TensorFlow and Keras model on AWS. We initially attempted to utilize their SageMaker
service to simplify this process. Despite slow progress, we successfully deployed the
model and established an endpoint. However, we encountered another problem when
we attempted to send a request to the endpoint. An error occurred while processing the
image, even though our prediction script could handle images. Identifying the root cause
of this error proved difficult, mainly due to insufficient documentation and a pressing time
constraint. We ultimately had to explore alternative options due to the time constraint and
online documentation. While resolving this issue would have strengthened our results, we
mitigated this setback by setting up a cloud environment on the local computer to simulate
a cloud.

7.2 Summary

At the start of our project, we had a clear plan of the steps we wanted to take. Unfortu-
nately, we had limited experience with essential tools like Xcode, Swift, and AWS. This,
along with time constraints, made things difficult. Our lack of experience slowed our
progress and made it hard to stay on track with the timeline and milestones we set in our
project plan. You can see a visual representation of our working process in Figure 7.23.
Figure 7.23: Flowchart of the work progress
8 Conclusion and Future Work

The initial thing that can be further improved in future work is implementing the model on-cloud and seeing how it performs. It could be interesting to see how it acts when deployed on different cloud services, e.g. AWS, Azure and Google Cloud.

We recognize that a side-by-side analysis of TensorFlow and PyTorch could have contributed to unique insights, particularly regarding model behavior post-compression. Moreover, while the scope of this study did not include assessing the model’s performance on Android, it would be intriguing to explore this in the future. Additionally, contrasting the performance of a simpler model architecture with the more complex VGG16 CNN used in this study could have shed light on the influence of model complexity, depth, and different layers on the smartphone’s capabilities. This aspect presents an area for deeper exploration in future studies. An exciting area to explore would be to experiment with training a model within Create ML, Apple’s tool for training and creating Core ML models. It could provide insights into whether these native models are equally effective and more optimized. It could also reveal the struggles of experimenting with third-party libraries, such as TensorFlow when deploying models on iOS smartphones. One possible approach to enhancing the efficiency of the Core ML model’s processing and classification of test images from the data set could involve its configuration to manage multiple input images simultaneously. This strategy might reduce the overall execution duration while maintaining accuracy. However, this method could strain the smartphone, posing a risk of being unable to handle the demand.

For future research, an enhancement could be to conduct a comparative analysis of smaller CNNs, like MobileNet or Squeezenet architectures. It could shed light on how model complexity impacts the total execution time when run directly on a mobile device and how it correlates with accuracy.
References


A Appendix 1

The source code and data that has been used during this project can be found https://github.com/Edjo6/thesis.