Implementing a TensorFlow-Slim based Android app for image classification

Maximilian Jokel
Implementing a TensorFlow-Slim based Android app for image classification

Implementierung einer Android-App zur Bilderkennung mittels TensorFlow-Slim

Author: Maximilian Jokel
1st examiner: Univ.-Prof. Dr. Hans-Joachim Bungartz
Assistant advisor: M. Sc. Severin Reiz
Submission Date: October 15th, 2020
I confirm that this bachelor’s thesis is my own work and I have documented all sources and material used.

October 15th, 2020  
Maximilian Jokel
Abstract

In this bachelor’s thesis I explore how convolutional neural networks for image classification tasks can be deployed to mobile devices for evaluation purposes. I focused on models defined and trained with the popular TensorFlow-Slim library. The result of this project is an Android app that allows you to deploy such models without making changes to the source code. Moreover, I describe the differences between convolutional neural networks and regular neural networks with regards to a core task in the field of computer vision, image classification. I then briefly outline the potential of transfer learning in the context of image classification. Finally, I give a detailed description on the design goals, development process and architecture of the app.
# Contents

Abstract vii

I. Introduction and Background Theory 1

1. Introduction 3
   1.1. Deep Learning Revolution 3
   1.2. Motivation and Structure 3

2. Computer Vision 5
   2.0.1. Tasks in Computer Vision 5
   2.0.2. Features 5

3. Convolutional Neural Networks 7
   3.1. Motivation 7
   3.2. Introduction to Convolutional Neural Networks 8
   3.3. Convolutional Operations 8
   3.4. Architecture of Convolutional Neural Networks 10
      3.4.1. Types of Layers 10
   3.5. Regularization 14
   3.6. Concluding on Convolutional Neural Networks 15
      3.6.1. Design Patterns 15
      3.6.2. Feature Hierarchy 16

4. Transfer Learning 17
   4.1. Transfer Learning in Image Classification Tasks 17
   4.2. Fixed Feature Extractor 18
   4.3. Fine-Tuning 18
   4.4. Concluding on Transfer Learning 18

II. Body: What was done for the thesis 19

5. App Development 21
   5.1. Use Case, Approach and Design Goals 21
      5.1.1. Use Case 21
## Contents

5.1.2. Approach .................................................. 21  
5.1.3. Design Goals ............................................... 22  

5.2. Android Development and External Libraries .................. 22  
5.2.1. Android Development ..................................... 22  
5.2.2. CameraX .................................................... 23  
5.2.3. TensorFlow Framework .................................... 25  
5.2.4. TensorFlow-Slim .......................................... 26  
5.2.5. TensorFlow-Lite ........................................... 27  

5.3. App Architecture ............................................ 31  
5.3.1. Architecture: Activities and Classes ..................... 31  
5.3.2. Other notable Classes .................................... 34  

5.4. Image Classification Flow ..................................... 35  

III. Results and Conclusion .................................... 39  

6. Conclusion .................................................... 41  
6.1. Result ....................................................... 41  
6.2. Future Work ................................................ 41  

Appendix .......................................................... 45  
A. TensorFlow-Slim Model Definition ............................. 45  
B. App Screenshots ............................................... 47  

Bibliography ...................................................... 51
Part I.

Introduction and Background Theory
1. Introduction

1.1. Deep Learning Revolution

Ever since the AlexNet convolutional neural network won the ILSVRC 2012 by a considerable margin, there has been a big push into this field of research [40, 28]. However, we find that AlexNet’s architecture is very similar to other convolutional neural networks that were introduced years before in the late 1990s.

How does it come, that the so called deep learning revolution has only started so recently? *Scale* is considered to be the main driver of this development, with particular attribution to two aspects: *data* and *computational power*. We live in a world where the internet has made it easier than ever to collect large amounts of useful data. And with the advent of dedicated GPUs, it is became possible to efficiently train deeper networks on these large amounts of data. Recent advances in algorithmic design have additionally contributed to improved efficiency, allowing us to build even deeper and more complex networks.

1.2. Motivation and Structure

It is for these reasons that convolutional neural networks have become an important part in the field of computer vision. With the emergence of comprehensive frameworks such as TensorFlow or PyTorch, developing and training new networks on visual tasks became increasingly more accessible.

However, we find that evaluating those models on real-world data can pose a challenge. For this reason I have created an Android app that acts as a front end for locally deployed image classification models [8]. This bachelor’s thesis gives an introduction into background theory and documents the development process.

The thesis is structured as follows: first, I give a swift overview of the field computer vision. This is followed by a chapter on convolutional neural networks where I describe the differences to regular neural networks. The section concludes with an introduction into the field of transfer learning with focus on image classification tasks.

In the second part of the thesis I focus on the aspects of app development. First, I outline the use case and overall design goal. Next, I introduce the external libraries used in the app, TensorFlow-Lite and CameraX. This is followed by a detailed description of the app’s architecture.

The thesis concludes with a brief look at the resulting app and possible future work. Screenshots of the app’s activities can be found in the appendix.
2. Computer Vision

Computer vision (CV) is a field of research that deals with how visual data can be automatically analyzed and interpreted by machines [51]. The field is highly interdisciplinary and touches many different areas in computer science, mathematics and engineering from a research and applied point of view. Digital images are made up out of individual pixels that are aligned in a grid-like structure. Depending on the color channels, we typically store them in $n$-dimensional integer arrays or serialize them into vectors, where each value represents one individual pixel. However, this results in a so called semantic gap between the semantic idea of the object(s) in the picture and the digital representation [31]. External influences can further intensify this issue. While it is apparent that an object’s semantics stay unaltered when the camera perspective changes, this is not the case for the digital representation where the pixel values may differ significantly. Other examples for such influences include changes in a scene’s illumination and when the object is deformed or partly occluded.

We typically approach tasks in CV by creating statistical models. Given the semantic gap described above, we find that such a model must be both, reasonable robust towards potential negative external influences, as well as reasonable efficient in terms of runtime and computational resources.

2.0.1. Tasks in Computer Vision

Image classification or category recognition is one of the core problems in CV. It is the task of assigning a label from a fixed set of classes to the overall contents of an image [31, 51]. We usually assume that the image contains only a single semantic object.

As it turns out, many different tasks in CV can actually be broken down into image classification problems. For example, in object detection we try to recognize and locate one or multiple objects of interest within rectangular areas of an image. In image segmentation we partition an image into (semantically) related regions, like foreground and background. Figure 2.1 provides a visual overview of these tasks. However, we will focus on the task of image classification from here on.

2.0.2. Features

In the field of CV, we refer to features as any kind of abstract information in an image [51]. Features allow us to characterize certain image regions based on specific structures, such
as edges, corners or blobs of certain colors. We refer to these as low-level features as they are essentially ubiquitous in any image.

By combining multiple features we can eventually gain a high(er)-level understanding of an image’s contents. The process of finding relationships between specific features that indicate certain abstract objects is referred to feature engineering. Up until recently, this highly complex process depended largely on manual labour. The term feature detection refers to the typically automated process of finding and extracting features in images. These processes are essential building blocks in any CV related tasks.

Figure 2.1.: Common tasks in computer vision. Figure based on [41].
3. Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of neural networks (NNs) that have proven to work well in perceptual tasks. In this section, we look at the reasons behind that by showing the key differences between regular NNs and CNNs.

3.1. Motivation

Regular NNs don’t work well with structured data, such as images. To understand why, we take a look at the following example. Consider that we want to build a NN for an image classification problem. The model will process small RGB images that are, for example, 224 by 224 pixels wide and high and return a vector \( s \) that holds the individual class scores. Assume that we are given a set of training data with \( n \) labeled images across \( m \) classes:

\[
D = \{(x_i, y_i) \mid 0 \leq i \leq n\}
\] (3.1)

\( D \) is a finite set of tuples. A tuple’s \( x \) component is a column-vector that holds a serialized representation of image \( i \). The \( y \) component is an integer that resembles the associated class index.

![Figure 3.1.: Example data set based on MNIST data [26].](image)

We use backpropagation to train the model on the examples in \( D \). This process allows us to iteratively update the weights \( W \), resulting in improved prediction accuracy over time. It is based on computing the gradients of the so called loss function, however we skip a detailed explanation as this would go beyond the scope of this thesis.

The prediction is based on a linear score function \( f(x, W) = Wx \) that maps the pixel values in \( x \) to feature space by applying the dot product with the weights specified in matrix \( W \). By chaining this score function, we create a basic NN with two layers:

\[
s = W_2 \ast \max(0, W_1 \ast x)
\] (3.2)

Let’s break this down. At the very first stage, we compute the dot product of weights matrix \( W_1 \) and input vector \( x \). This results in an intermediate activation vector. Up next is \( \max(0, -) \), a so called activation function or non-linearity. This non-linear function is applied
to each element of the vector, resulting in an intermediate output vector whose individual elements are at least 0. By applying the dot-product with $W_2$, we finally obtain the output vector $s$ that holds the $m$ individual class scores. We interpret the index of the largest value in $s$ as the associated class index.

It is important to note that the layers of this simple network are fully-connected. This means that the intermediate output vectors or activations of a layer resemble the inputs of each neuron in of following layer. It is this type of connection that almost all regular NNs use. We will now see, why that poses an issue when considering images as input data.

Consider that vector $x$, which we use to store a serialized RGB image, holds $224 \times 224 \times 3 = 150,528$ individual pixel values. Due to the fully-connected layers in our NN, every neuron in the first hidden layer alone has $150,528$ inputs, that are each associated with a separate weight parameter. Moreover, for such a complex task as image classification, our network would likely need multiple layers with numerous neurons.

We can clearly see that approaching this problem with a NN based on fully-connected layers will not scale to larger architectures due to the extremely large number of parameters.

### 3.2. Introduction to Convolutional Neural Networks

CNNs are specifically designed around data that has a known grid-like structure, such as digital images [36]. We recall that images are typically stored in multidimensional arrays, often called tensors, where the individual elements represent the pixel values. CNNs make use of that by arranging the neurons in their layers in grid-like structures as well. The dimensions of these neuron-grids depend to some extend on the structure of the inputs. This is in contrast to regular NNs, where the neurons are aligned in a single column per layer. That leads us to the first conclusion: CNNs work over input volumes while regular NNs work on vectors. Figure 3.2 visualizes the differences in neuron alignment in the two network types.

Another key difference is that, while neurons in regular NNs apply matrix multiplications, the neurons in CNNs apply convolutional operations. This significantly reduces the number of learnable parameters in the network, resulting in decreased computational expense and ultimately allowing us to build larger, more complex network architectures.

### 3.3. Convolutional Operations

A convolution is a linear operation that, given two functions $f$ and $g$, will return a third function $h = f * g$ [36]. This means that each $f(x)$ is replaced by the mean over its surrounding function values, that is weighted by $g$. Since we are in the context of images, we can formalize the discrete convolutional operator like so:

$$ (f * g)(n) = \sum f(k)g(n - k) \quad (3.3) $$
3.3. Convolutional Operations

It is a common convention in NN terminology to refer to the first argument as input, while the second argument of the convolutional operation is typically called kernel or filter. The output is often called feature map or activation map \([36, 51]\). Intuitively, a convolutional operation applies the kernel to the input and produces a feature map. However, images are at least two-dimensional input volumes. Consequently, we usually apply convolutional operations to multiple input dimensions at the same time. For example, given a two-dimensional input image \(I\), we apply a two-dimensional kernel \(K\) like so:

\[
(I \ast K)(i, j) = \sum_{m} \sum_{n} I(m, n)K(i - m, j - n)
\]  

(3.4)

This means that we essentially slide \(K\) over the width and height dimension of \(I\). As a result we obtain a representation of the input which is a weighted combination, i.e. dot product, of the input pixels and the kernel. The exact details of the combination depend on the kernel.

In practice, we find that many libraries for building CNNs actually implement the cross-correlation where we flip input and kernel based on the commutative properties of the convolutional operation \([36]\). This achieves the same result, but is considered to be more straightforward to implement since there is usually less variation in the ranges of \(m\) and \(n\).

\[
(K \ast I)(i, j) = \sum_{m} \sum_{n} I(i + m, j + n)K(m, n)
\]  

(3.5)

Moreover, we find that kernels are usually small and have square dimensions. Common sizes for width and height are three to seven pixels, but we rarely use even numbers. It is important to note that the remaining dimensions of the kernel must always match those of the input data. For example, kernels for greyscale images need a depth of one, while RGB color space requires a depth of three.
Figure 3.3 presents an example for a two-dimensional convolutional operation. We use a 8x8 greyscale image as input and apply a 3x3 kernel. This operation will result in the image on the right.

<table>
<thead>
<tr>
<th>45</th>
<th>60</th>
<th>98</th>
<th>127</th>
<th>132</th>
<th>133</th>
<th>137</th>
<th>133</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>65</td>
<td>98</td>
<td>123</td>
<td>126</td>
<td>128</td>
<td>131</td>
<td>133</td>
</tr>
<tr>
<td>47</td>
<td>65</td>
<td>96</td>
<td>115</td>
<td>119</td>
<td>123</td>
<td>135</td>
<td>137</td>
</tr>
<tr>
<td>47</td>
<td>63</td>
<td>91</td>
<td>107</td>
<td>113</td>
<td>122</td>
<td>128</td>
<td>134</td>
</tr>
<tr>
<td>50</td>
<td>59</td>
<td>80</td>
<td>97</td>
<td>110</td>
<td>123</td>
<td>133</td>
<td>134</td>
</tr>
<tr>
<td>49</td>
<td>53</td>
<td>68</td>
<td>83</td>
<td>97</td>
<td>113</td>
<td>128</td>
<td>133</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>58</td>
<td>70</td>
<td>84</td>
<td>102</td>
<td>116</td>
<td>126</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>52</td>
<td>58</td>
<td>69</td>
<td>86</td>
<td>101</td>
<td>120</td>
</tr>
</tbody>
</table>

\[ \begin{array}{cccccccc} 
0.1 & 0.1 & 0.1 \\
0.1 & 0.2 & 0.1 \\
0.1 & 0.1 & 0.1 \\
\end{array} \] \times \begin{array}{cccccccc} 
45 & 60 & 98 & 127 & 132 & 133 & 137 & 133 \\
46 & 65 & 98 & 123 & 126 & 128 & 131 & 133 \\
47 & 65 & 96 & 115 & 119 & 123 & 135 & 137 \\
47 & 63 & 91 & 107 & 113 & 122 & 128 & 134 \\
50 & 59 & 80 & 97  & 110 & 123 & 133 & 134 \\
49 & 53 & 68 & 83  & 97  & 113 & 128 & 133 \\
50 & 50 & 58 & 70  & 84  & 102 & 116 & 126 \\
50 & 50 & 52 & 58  & 69  & 86  & 101 & 120 \\
\end{array} \]

\[ = \begin{array}{cccccccc} 
69 & 95 & 116 & 125 & 129 & 132 \\
68 & 92 & 110 & 120 & 126 & 132 \\
66 & 86 & 104 & 114 & 124 & 132 \\
62 & 78 & 94  & 108 & 120 & 129 \\
57 & 69 & 83  & 98  & 112 & 124 \\
53 & 60 & 71  & 85  & 100 & 114 \\
\end{array} \]

3.4. Architecture of Convolutional Neural Networks

We recall that regular NNs are essentially acyclic graphs where neurons make up the nodes, that are arranged in a layer-like structure. This allows us to execute the individual computations in a feed-forward way, making the process less complex and computationally expensive. A NN receives a single input vector that it is transformed step by step by the hidden layers. It is important to note that all of these transformations happen independently as each neuron has its own associated set of learnable weights.

While CNNs also consist out of successive layers, we have already stated that their neurons are aligned in a grid-like structure since they work over volumes instead of vectors. We will now discuss the benefits of this topology as well as the other types of layers found in CNNs in the context of a model for an image classification task.

3.4.1. Types of Layers

Input Layer

A CNNs’ input layer holds the individual pixel values of an input image. We assume for the rest of this section that the model processes RGB images, meaning that it works over
three-dimensional volumes. Note that it is a common convention to not include the input layer when counting the number of layers in a network.

**Convolutional Layer**

Convolutional layers transform their input volumes by applying multiple kernels in separate convolutional operations [36, 29]. We save the outcomes of these operations in two-dimensional feature maps or activation maps that we arrange one after the other, resulting in a new three-dimensional volume.

Note that each element of a given feature map was computed with the same kernel. The kernels are specific to each convolutional layer since they are learnable parameters that are obtained in the training process. We refer to this property as parameter sharing. As a result, CNNs have significantly less parameters than regular NNs. This, in turn, reduces storage requirements and computational expense in training, marking an important difference to regular NNs, where each neuron is associated with an individual weight parameter that isn’t shared.

Each convolutional layer comes with a number of hyperparameters. The number of kernels per layer controls the depth of the resulting feature map of that layer. Its width and height are controlled by the kernels’ spatial dimensions and their stride of application. Lastly, there is a hyperparameter that allows us to apply padding to the input. The output of a convolution is in general slightly smaller as the input because the values along the borders are not factored in. By applying padding to the inputs we can avoid this additional spatial downsampling.

Figure 3.4 allows us to gain a more intuitive understanding of the operations in convolutional layers. We see that, despite their simply structure, kernels are useful for finding (low-level) features such as edge or corners, to sharpen an image or to remove noise by applying blurs.

![Identity Edge detection Sharpen Box blur 3x3 Gaussian Blur](Figure 3.4.: Kernel convolutions. Figure based on [25]).
3. Convolutional Neural Networks

Activation layers

Activation layers apply a non-linear activation function to each individual element of a feature map [29]. Unlike convolutional layers, activation layers have no learnable parameters or hyperparameters. They simply implement a fixed, non-linear function that does not alter the dimensions of the input volume.

There are a number of different activation functions, such as sigmoid. Until recently, it was a common choice largely due to historic reasons and its intuitive interpretability. The core property of the sigmoid function is that it squashes its inputs into the range of $[0, 1]$:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (3.6)$$

However, almost all modern network architectures use rectified linear units (ReLUs) instead. They are associated with strong performance improvements due to their much simpler definition that does not involve computationally expensive exponentials:

$$f(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  0 & \text{otherwise}
\end{cases} = \max(0, x) \quad (3.7)$$

A ReLU activation function will threshold all activations at 0, additionally decreasing computational expense and thus accelerating the training process substantially [40]. ReLUs are considered among the few recent, significant contributions to algorithm improvement [38].

However, working with RELUs in practice can be difficult as they turn out to be fragile in backpropagation. A large gradient flow can cause weights to update in such a way, that future activation signals will become negative. If this happens across all inputs of a neuron, it gets irreversibly deactivated because the ReLU activation function no longer returns any values $> 0$. Setting an appropriate learning rate in backpropagation can help to avoid this issue [32]. A leaky ReLU activation function approaches this problem by giving the activation to a small negative slope instead of returning 0 for negative inputs [42, 37]:

$$f(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  \alpha x & \text{otherwise}
\end{cases} \quad (3.8)$$

The value $\alpha \geq 0$, e.g. $\alpha = 0.01$, can either be hard coded or set as a hyperparameter.

Pooling layers

We use pooling layers to reduce the spatial dimensions of volumes. The core idea behind pooling is creating a systematic summary that only holds the most relevant information. It allows us to decrease the number of parameters, which is first of all beneficial regarding the risk of overfitting. In addition, it reduces the number of computations and the memory requirements for storing a model while it increases a network’s capacity at the same time.
3.4. Architecture of Convolutional Neural Networks

Moreover, we find that pooling over the course of multiple layer helps the model to become increasingly invariant to changes in a feature’s location. This is helpful in the context of image classification, where we want to recognize an object of interest independently of its position in the input image.

There are a number of ways to do perform pooling, but the max pooling technique is among the most popular ones \([36, 29]\). We can think of max pooling as a \(2 \times 2\) kernel that is applied along an input’s width and height dimension with a stride of 2, where we only pick the largest among the four values. See figure 3.6 for a detailed example. We see, that 75% of the activations in a feature map are discarded with this particular configuration.

However, max pooling layers usually come with hyperparameters that allow us to adjust the kernel and its stride. Note that pooling layers in general have no learnable parameters and only affect an input’s width and height, but not its depth dimension.

![Figure 3.5.: Activation functions [24].](image)

![Figure 3.6.: Example max pooling operation with a stride of 2. Figure based on [29, 51].](image)
### 3. Convolutional Neural Networks

**Fully-Connected layers**

We typically find *fully-connected* layers near the end of a CNN’s architecture [29]. Just like in regular NNs, its neurons are aligned in a single layer-like structure while being fully connected to all activations of the preceding hidden layer. The neurons of fully-connected layers are associated with weights, that are, again, learned during training.

If a fully-connected layer is the very last layer of a CNN, its number of neurons usually corresponds to the $m$ classes in the training data. By applying a linear classifier, such as a softmax function or a support vector machine (SVM), we finally obtain a *feature vector* with the individual class scores. Note that the network has transformed the input volume into a single column vector at this point.

#### 3.5. Regularization

Regularization is another aspect in which modern CNNs differ from regular NNs. When working with image data, it is very likely that *overfitting* occurs. This is due to the reason that there is often simply not enough labeled training data available [33]. A model that is overfitted to its training data will generalize in an unpredictable way [36].

We approach this issue with a method called *regularization*, where model complexity is penalized in favor of more general, simple structures. While regular NNs have relied on regularization methods such as $L1$ or $L2$ norm [36], we find that there is another technique with a different approach that comes with significant performance benefits.

Regularization by *dropout* is considered one of the driving factors in CNN performance [48, 33, 38]. The core idea of dropout is to randomly remove neurons across all layers along with their connections in each round of training. See figure 3.7 for a visualization.

---

Figure 3.7.: Visualization of the effect of dropout on (temporary) network structure. Figures taken from [48].
3.6. Concluding on Convolutional Neural Networks

3.6.1. Design Patterns

We find that many state of the art CNNs have a deep structure in which certain groups of layers are repeated several times. A simple architecture for an image classification task might look something like this [29]:

\[
\text{IN} \rightarrow ([\text{CONV} \rightarrow \text{RELU}] \ast k \rightarrow \text{POOL}? \ast m \rightarrow [\text{FC} \rightarrow \text{RELU}] \ast n \rightarrow \text{FC}
\]

For \(k = 2\), \(m = 3\) and \(n = 1\), this expression nearly resembles the architecture of renowned VGGNet16 that came second in ILSVRC 2014 [47, 1] (the real network uses \(k = 3\) for \(m > 2\)). The network is known for its simple architecture, as can be seen in listing A.1, and considered strong evidence that deep structures and model performance are linked [47].

Figure 3.8.: Effect of dropout on a single neuron. Figures taken from [48].

In order to develop an intuition for this technique, we focus on a single neuron as can be seen in figure 3.8. In a regime where any neuron can be eliminated at random, a neuron can not rely too much on single inputs. To become invariant to such effects, its weights need to be spread across all if its incoming connections instead. We find that this archives a very similar effect to a \(L2\) norm based regularization: the weights shrink and become more diffuse. The probability \(p\) for keeping a neuron active is set as a hyperparameter. It is possible to introduce one parameter per layer, which gives us the opportunity to specifically focus on highly interconnected layers, that are more likely to be affected by overfitting.

We find that regularization by dropout accelerates the training of a network by a considerable amount, since it essentially reduces a model’s size during that process [48].
3. Convolutional Neural Networks

3.6.2. Feature Hierarchy

Earlier, we found that feature engineering and feature detection are time-consuming and highly complex tasks that, until recently, relied largely on manual labour [51]. Despite the expertise and domain knowledge that got incorporated into the process, the performance of these hand-crafted feature extractors was insufficient.

CNNs and deep learning introduce a new approach to this by making it possible to systematically learn the relevant features to extract. This is partly attributed to the so called feature hierarchy that gets formed across the multiple layers of a network [54, 53].

It was found that the activation functions in the early layers get excited by low-level features, such as corners, edge or color conjunctions. With increasing depth, the complexity of those features increases, too. For example, an eight layer model similar to prestigious [40] will respond to specific textures or patterns in layer 3. Its 4th layer is specific to higher-level features, such as any objects with a round shape. Finally, layer 5 is able to recognize full semantic objects. The network is at this stage invariant to interclass variation and changes in camera perspective.
4. Transfer Learning

Recent advancements in algorithms, such as dropout or ReLUs, and the ever growing amount of data clearly favor the machine learning based approach to core tasks in CV, such as image classification. Given the feature hierarchy in CNNs, a new use case opens up: transfer learning.

4.1. Transfer Learning in Image Classification Tasks

Transfer learning (TL) deals with applying already gained knowledge for generalization to a different, but related domain [34]. Creating a separate, labeled dataset of sufficient size for a specific task of interest in the context of image classification is a time-consuming and resource-intensive process. Consequently, we find ourselves working with sets of training data that are significantly smaller than other renowned datasets, such as CIFAR and ImageNet [39, 49]. Moreover, the training process itself is time-consuming too and relies on dedicated hardware. Since modern CNNs take around 2-3 weeks to train on ImageNet in a professional environment, starting this process from scratch for every single model is hardly efficient.

The idea of TL is that by building upon a pre-trained model, we can leverage its knowledge for our task of interest. This allows us to cut down on the cost for aggregating data and model training [34]. We are now going to look at the two most common techniques for applying TL in the context of image classification. For that we assume that there is a given model that was pre-trained on the ImageNet database.

Figure 4.1.: Visualization of the transfer learning approach. Figure based on [44].
4. Transfer Learning

4.2. Fixed Feature Extractor

We recall that the last layer of such a model is typically fully-connected and returns the output vector that resembles the 1000 class scores. By removing this layer, while leaving the remaining hidden layers unaltered, we create a fixed feature extractor \[34, 45\]. Because the model’s feature hierarchy is preserved, we can then apply this extractor to the training set for our task of interest. This gives us a multidimensional vector that holds the so-called CNN codes, i.e. the activations of the network’s final hidden layer, for each input image. After iterating over all training examples and applying a ReLU activation function to the resulting CNN codes, we train a new linear classifier, such as softmax or SVM, that replaces the previously removed output layer.

To summarize, the fixed feature extractor approach allows us to leverage a pre-trained model’s weights for extracting features in an off the shelf manner, that we then use to train a classifier for our task of interest.

4.3. Fine-Tuning

Fine-tuning allows us to adjust our new model to a new domain by using the pre-trained weights and biases as initialization for the backpropagation process \[34, 53\]. The benefit of this technique is that it allows us to selectively update the parameters, meaning that we can adjust the generalization capabilities of a models feature hierarchy. We usually want to keep the generic, low-level features, but update the higher-level ones to the new domain. However, while fine-tuning might result in better prediction accuracy, it depends on the size and similarity of the new training data to the initial dataset. It is clear that this technique is not preferable for small datasets in general, due to the high risk of overfitting.

4.4. Concluding on Transfer Learning

TL allows us to create complex models for specific tasks with limited amounts of labeled training data and is therefore considered a driving factor for the field of deep learning \[43\]. We find that both TL techniques, fixed feature extractors and fine-tuning, have proven to work well across a number of different visual tasks \[45, 54, 53\]. Since it is a common practice in ML to share own contributions with the community, there is a broad variety of pre-trained models available that can be leveraged for TF. However, it should be noted that the underlying model must be suitable for the new task of interest when applying TL.
Part II.

Implementing a TensorFlow-Slim based Android App for Image Classification
5. App Development

As part of this thesis I implemented an Android app that allows the user to classify images from the camera view finder and gallery. It is the intention of this section to describe and explain the approaches and design considerations behind the implementation.

We start with the app’s use case, followed by the approach and the design goals. I then introduce the Android operating system and the two external libraries that the app relies on. Lastly, we have a detailed look at the app’s architecture and the classification workflow.

5.1. Use Case, Approach and Design Goals

5.1.1. Use Case

In the section on transfer learning I briefly outlined the potential of this technique in the context of CNN based image classification. We found that TL is widely used today as it is easy to obtain resources, like pre-trained model weights, in such a collaboration-friendly community. However, we find that evaluating a model’s prediction accuracy on real-world data, beyond using the test images from the training dataset, isn’t all that straightforward. This step might be limited to running the model locally on previously downloaded images since model deployment for test purposes is apparently not yet a use case. This leads us to the following question:

Wouldn’t it be nice if there was an app that simply runs a model on your phone’s camera and displays the classification results?

5.1.2. Approach

For an online approach we need to deploy a model to a backend server. The app becomes hence a pure front end, which only captures images and sends them to the server over a network connection. The server then runs the model on the image and returns the classification results to the app, that displays them to the user.

This online approach has the benefit that all complex computational aspects are outsourced to the backend, allowing a rather simple app architecture. However, the network connection will affect inference latency, which makes us doubt a live classification use case. Offline use is not possible anyway. Lastly, it should be noted that such a backend causes additional costs and maintenance efforts.
5. App Development

This is why I decided to focus on an on device approach. Since there is no need for a backend, the network connection can be also omitted which enables offline and privacy sensitive use cases as no data needs to leave the device. Inference latency will benefit as well. However, this presupposes that the device offers enough disk space for storing a model as well as sufficient computational resources for running it.

Clearly, this approach requires a more complex app architecture. Yet given the computation power of modern smartphones, we should be able to run inference at a decent latency. And as it turns out, typical image classification models can be compressed so that take up about 20 MB in storage which appears, again, decent for what they are capable.

5.1.3. Design Goals

The overall design goal is to keep of the app as simple and straightforward as possible so that others can work with it easily. This includes, first of all, using only an absolute minimum of external libraries. Second, a modular structure will avoid code reuse and should make code comprehension and maintenance easier. Third, a user interface (UI) based on Google’s Material Design ensures a consistent and appealing user experience. Ultimately, the app should work ‘just out of the box’, meaning that the process of deploying a model for evaluation should require minimal effort. This specifically includes that the source code does not have to be changed as re-building the app will be sufficient. I will refer to this as dynamic approach.

The app will provide two use cases for image classification. The default procedure uses the camera’s view finder as input, resulting in a live or per frame classification of the objects the user points their phone at. The second use case is meant for classifying single images that the user can pick from their phone’s gallery. A simple UI allows the user to choose from different models and to switch between cameras.

As I am new to the field of Android development, I referred to [35] for best practices. With regard to the scientific standard of this thesis, I have documented the ideas and approaches that I have adopted from others in the implementation.

5.2. Android Development and External Libraries

5.2.1. Android Development

Android is an open-source operating system (OS) for mobile devices that is based around the Linux kernel. Its development is primarily driven by Google. While it is difficult to obtain exact numbers, Android is considered to be the most popular OS today and has dominated the mobile device market over the past decade [7]. Its most recent version is Android 11 (API level 30). However, this app has been developed and tested on a device running Android 10 (API level 29). For that, I used Android Studio which is the official IDE that is based on the popular IntelliJ IDE. The app is implemented in Java, as there was more information available than on programming with Kotlin.
5.2. Android Development and External Libraries

Activities

At its core, an Android app is made up out of multiple Java classes that are associated with specific views. The user interacts with an application via the various UI elements on screen. In Android, we refer to this concept as activity. Switching between different views usually means launching and terminating activities.

When an activity is launched, its onCreate() function is called. We use that function to assign the activity to its layout file that holds the UI. If the UI features buttons or other elements for interaction, we usually set up their event listeners here as well. We find that an activity in Android consists out of one .xml layout file and one associated Java class, that acts as controller. See listing 5.1 for an example.

Fragments

Fragments in Android are a very similar concept to activities since they are made up out of a Java class and layout file as well. We typically use fragments to modularize an application. This means that we can use multiple fragments in order to build a larger activity. The benefit of a modular architecture is that we can reuse components across multiple activities in the app.

5.2.2. CameraX

In order to provide a reliable experience across different devices, it seems reasonable to rely on renowned external libraries for handling critical features. Given the heterogenous landscape of OEMs and device hardware, handling cameras in Android poses a challenge for developers.

As part of the Android Jetpack collection, CameraX addresses this issue by providing a high-level cross-device API to control camera functions and reliably use essential features in an use-case oriented paradigm \[5\]. The library, that is currently in beta stage, was introduced at Google IO 2019 and works with devices that run Android 5.0 or higher \[52, 5\].

We include the CameraX library by adding androidx.camera:camera-core to the app’s build.gradle file. To get started we need to initialize a CameraSelector, that allows us to access a phone’s front- and rear-facing cameras. We can now set up individual use cases and bind them to the camera.

In the app, we want to display the view finder of the selected camera to the user. For that, we set up a CameraX Preview object that streams the view finder’s frames to a PreviewView layout element in the UI. In order to run inference on such a frame, we need to convert it first. Android encodes Images and ImageProxs in YUV color space while most models rely on RGB images. We use an ImageAnalysis object that allows us to transform the camera frames arbitrarily. Once the camera captures a frame, it is passed on to all analysis use cases, that are bound to the camera’s lifecycle. A benefit of this approach is that it allows us to run the computationally expensive tasks in separate threads. It is
5. App Development

```java
// [...] 
public class ViewFinder extends AppCompatActivity{

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // set layout
        setContentView(R.layout.activity_view_finder);

        // set up event listener for button
        findViewById(R.id.button).setOnClickListener(
            new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    // TODO
                }
            });
    }
    // [...] 
}

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".ViewFinder">
    <TextView
        android:text="Hello, world!"/>
</LinearLayout>
```

Source Code 5.1.: Example activity implemented in Java and XML.
5.2. Android Development and External Libraries

Source Code 5.2.: Setup for CameraX preview use case (simplified).

important to note that analysis use cases are conceptually processing pipelines, meaning that we have to explicitly close() an image to indicate that we are done with it and ready to receive the next frame for processing.

The code for above examples is shown in listings 5.2 and 5.3. In summary we find that working with CameraX is indeed straightforward as it offers with Preview and Analysis two interfaces that are well suited for the needs of the app.

5.2.3. TensorFlow Framework

TensorFlow (TF) is a comprehensive high-level framework for designing and training NNs that was initially developed for internal use at Google [13]. It was publicly released in 2017 while the upgrade to the most recent version 2 followed in 2019 [2, 23]. TF is maintained by Google and the open source community.

The framework aims at providing an end-to-end approach, meaning that a developer can build, train and finally deploy models solely with TF. It provides a high-level Python API for defining model architectures. Another strength of TF is its support for distributed training. This makes it possible to delegate the computationally expensive training process to dedicated external hardware such as GPU clusters or remote servers. In addition, TF offers integrations for languages such as C++, Java and JavaScript. This allows for model deployment across different platforms, including mobile devices as we will see below.

TF is widely adapted in the ML community and has evolved into a large ecosystem over time. Today, it bundles the capabilities several formerly independent libraries and provides compatibility with other popular libraries. One example for this is TF-Slim, which we will look at in more detail in the section below.
5. App Development

```java
// init analysis object for processing last frame
analyzer = new ImageAnalysis.Builder()
    .setTargetResolution(new Size(previewDimX, previewDimY))
    .setBackpressureStrategy(ImageAnalysis.STRATEGY_KEEP_ONLY_LATEST)
    .build();

// source computation out to new thread
ExecutorService service = Executors.newSingleThreadExecutor();

// set up transformation
analyzer.setAnalyzer(service, new ImageAnalysis.Analyzer() {
    public void analyze(@NonNull ImageProxy image) {
        // TODO
        image.close();
    }
});

// bind analysis use case to CameraX lifecycle
cameraProvider.bindToLifecycle(this, [...], analyzer);
```

Source Code 5.3.: Setup for CameraX analysis use case (simplified).

5.2.4. TensorFlow-Slim

TensorFlow-Slim (TF-Slim) is a standalone library that aims at streamlining the process of developing complex models for computer vision [16]. Being initially developed at Google for internal use, it later became part of the main TF framework [21]. However, with TF version 2, `tf.contrib.slim` is now deprecated as most of its features have been integrated into the framework’s core library [3]. Yet, as of October 2020, the TF-Slim repository is still actively maintained by one of its original developers.

TF-Slim was developed at a time when the main TF library was lacking structure and still work in progress. Specifically, there was no uniform way to define models, nor were there routines for training and model evaluation. TF-Slim encounters this by providing a high-level Python API that makes it easy to define models by omitting boilerplate code. The library itself has a modular design which ensures flexibility in the development process. For example, we can use TF-Slim for defining the model structure, but use a custom routine for training.

TF-Slim also provides central `Image Models Library` (within its repository) that hosts model definitions and pre-trained weights and biases in the form of checkpoint files [16]. The intention behind this is to reduce code duplication and to accelerates workflows. For example, it gives developers the chance to train proven model architectures on their own data sets, or to use pre-trained models for fine-tuning and transfer learning. The library
5.2. Android Development and External Libraries

```python
python export_inference_graph.py
--model_name=inception_v1
--output_file=/graph_def.pb
```

Source Code 5.4.: Exporting the inference graph for a TF-Slim designed model.

```python
freeze_graph
--input_graph=/graph_def.pb
--input_checkpoint=/training.ckpt
--input_binary=true
--output_graph=/tmp/frozen_graph.pb
--output_node_names=InceptionV1/Logits/Predictions/Reshape_1
```

Source Code 5.5.: Freezing a GraphDef and checkpoints file.

includes well-known models such as Inception V3 or VGGNet, that were developed in TF-Slim [21, 18].

To illustrate TF-Slim’s straightforward approach for defining models, we look at an example. The code segment of listing A.1 is taken from the definition of the 19-layers VGG net, a model that is known for its simple architecture. It became second in the 2014 ILSVRC image classification task with an error rate of 7.3% [46].

Tools for Exporting and Freezing Graphs

TF-Slim provides a simple API for exporting inference graphs from a model’s definition file [17]. This allows us to create a so-called GraphDef, which is a binary representation of a model’s architecture. To do so, we run TF-Slim’s export_inference_graph.py as seen in listing 5.4.

We use the resulting protocol buffer file (.pb) to freeze the graph. Freezing refers to the process of merging a model definition (GraphDef) and a set of checkpoints into a serialized binary file. For that, we run TF’s freeze_graph command as seen in listing 5.5. The resulting .pb file forms the basis for deploying the model on mobile devices, as we will see in the following section.

5.2.5. TensorFlow-Lite

The heterogeneous landscape of mobile and edge devices poses major challenge for developers who want to deploy their TF models in such environments. TensorFlow-Lite (TF-Lite) is an open source library that approaches this issue. It makes it possible to efficiently run TF models locally on Android, iOS and other mobile and edge devices. TF-Lite was initially announced in 2017 and features a high-level Java and Kotlin API [22, 14].
5. **App Development**

```java
try {
    tflite = new Interpreter(loadModelFile());
} catch ...

// loads .tflite-Model from '/assets' directory as memory mapped buffer
private MappedByteBuffer loadModelFile()
{
    ...
}

// this is where the model is actually applied
public void doInference(int i){
    tflite.run(inputImage, outputBuffer);
}
```

Source Code 5.6.: Conceptual setup for a TF-Lite classifier.

I did consider other approaches towards running TF models on Android, such as [4], first. However, after running into problems due to missing execution permissions for binaries or deprecated libraries, I came to the conclusion that TF-Lite would provide the best and most reliable solution.

To get started, we import the library by adding org.tensorflow:tensorflow-lite to the app’s build.gradle file. We furthermore add noCompress "tflite" to the aaptOptions section in order to prevent the TF-Lite model files from being compressed when the app is built. The code snippet in listing 5.6 shows the setup for an Interpreter object for running a model in TF-Lite.

### TF-Lite Converter

Before a TF model can be deployed and run on a mobile device with TF-Lite, we first need to convert it into a TF-Lite model. The core part of this process is serializing the model into a buffer that is then stored as .tflite file. Serialization is a process where structured data is transformed into a sequential representation, that can be mapped directly into a device’s main memory. Because of the sequential structure of RAM, programs can access this data very efficiently [6]. TF-Lite utilizes Google’s FlatBuffers serialization library for that.

The TF-Lite converter offers a Python API that works with a number of TF models, in particular with models that are defined and trained with TF-Slim. The conversion process is relatively simple and involves three steps, that are visualized in figure 5.1. Given a TF-Slim model definition in model.py and a corresponding set of weights in training.ckpt, we first export the inference graph and then freeze it with the checkpoints, just like described above. To convert the resulting frozen_graph.pb into a TF-Lite model, we run the Python script in listing 5.7.
5.2. Android Development and External Libraries

TF-Lite Optimization

In TF-Lite there are a number of options that allow us to optimize models and model inference specifically for use on mobile devices.

Quantization is the process of mapping inputs from a large, often continuous set to values in a smaller set, that usually has a finite number of elements. In the context of NNs, quantization refers to reducing the precision of a model’s weights and parameters. TF models use 32-bit floating point numbers for its parameters by default. With quantization, we usually reduce precision to 16 bit float or 8 bit integer [9]. Quantization reduces first of all the memory required to store a model. Moreover, as there is less data to move, processors can execute operations with less precise numbers much faster. However, while
import tensorflow.compat.v1 as tf
import tf_slim as slim

# Convert the model.
converter = tf.compat.v1.lite.TFLiteConverter.from_frozen_graph(
    graph_def_file='/frozen_graph.pb',
    input_arrays=['input'],
    input_shapes={'input' : [1, 224, 224, 3]},
    output_arrays=['InceptionV1/Logits/Predictions/Reshape_1'])
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

# Save the model.
with open('model.tflite', 'wb') as f:
    f.write(tflite_model)

Source Code 5.7.: convert.py: Python script for converting a frozen TF-Slim model into the TF-Lite format.

Quantized models are generally low in inference latency, they come with a considerable trade-off in model accuracy.

There are two approaches to this trade-off: quantization aware training and post-training quantization. Quantization aware training is considered the best trade-off between file size and model accuracy [11]. As the name suggests, the quantization aspect is already taken into account in model training. In the TF-Slim context, we have to adjust the classifier training routine in train_image_classifier.py by adding the following line before the section on optimizations: tf.contrib.quantize.create_training_graph(delay=n) [12]. This command allows us to specify the number of steps the training process is performed the usual way, based on 32 bit floating point numbers. After n steps, the quantization process starts where model weights and parameters are reduced to integer precision. Alternatively, we can use the command without parameter and load a checkpoint file instead, in order to fine-tune an already trained model with respect to quantization. In some cases we might have to re-iterate the model design as well.

The TF-Lite converter API additionally includes post-training quantization techniques for reducing precision to 16 bit float or 8 bit integer, or a dynamic range [10]. In general, the latter method has the most significant effect on model accuracy, while the former preserves accuracy at a larger file size. The advantage of post-training quantization is that it is essentially an independent process that we can apply to virtually any model, even pre-trained ones. Note line 11 in listing 5.7 that sets post-training quantization to dynamic range in the model conversion process.
5.3. App Architecture

Hardware Acceleration

However, we find that due to TF-Lite’s support for running models with hardware acceleration, we might not even need to specifically optimize float models. The system-on-a-chip unit of many modern phones includes GPUs and other dedicated circuitry that accelerate the computational operations common in machine learning tasks \cite{19, 20}. As of Android 8.1 (API level 27), the operating system exposes these capabilities with the Android Neural Network API \cite{27}.

Based on this API, TF-Lite tries to leverage device specific hardware in order to accelerate model inference. It can delegate parts or the full interpreter execution from the CPU to GPUs and neural accelerators. That, in turn, enables a use case for high-in-accuracy float models, that run on dedicated hardware, such as GPUs, much more efficiently. However, it should be noted that performance will obviously vary from device to device.

Over the course of developing and testing the app, I found that even large, unconverted float models such as Inception V1 \cite{50} would run on the CPU with decent inference latency in the low hundred milliseconds when using a small number of threads. Smaller models can easily be run below 30ms per image when using a decent number of threads. Performance was even better when the computational unit was switched to GPU or NNAPI. Since I considered the overall performance as sufficient, I decided to focus on hardware based optimization within the app, that also aligns well with the design goal of making model deployment as simple as possible. However, I am aware that there are other use cases where inference latency is absolutely critical. TF-Lite’s comprehensive approach to model deployment is therefore greatly appreciated.

5.3. App Architecture

This section is dedicated to the architecture of the app. We will have a detailed look at the individual activities and classes. The app was initially based on a demo \cite{15} but has since then been re-built from ground up to meet the design goals described above. This refers in particular to the dynamic approach that allows for model deployment without adding or adapting the Java source code. Instead, the nets.json file in the app’s /assets is used to specify a new TF-Lite model that was added to the directory.

5.3.1. Architecture: Activities and Classes

The app essentially consists out of four activities and a number of additional helper classes. In the first part of this section I am going to introduce the four activities in detail. Refer to figure 5.2 for a schematic overview of the app’s architecture with respect to the activities. The helper classes are presented in the second part.
StartScreen and PermissionDenied Activities

Whenever the app is started, it launches into the StartScreen activity. Its job is to check whether the user has granted the necessary permissions for accessing the camera and gallery since Android restricts access to these components by default. If this is the case, it directly starts the ViewFinder activity, meaning that its associated view is not rendered to the screen.

However, if one or both permissions are missing, which should usually only be the case when the app is first launched, the activity will display its UI where the reasons behind the access requests are stated. The UI features a button that, when pressed by the user, prompts two individual Android permission request menus for accessing the camera and for reading from the device’s storage. These prompts must be explicitly triggered because as of Android 6 (API level 23), the OS does no longer do this automatically. If the user accepts those requests the ViewFinder activity will be started.

The app switches to the PermissionDenied activity if the user declines one or more permissions. The activity’s UI gives a more elaborate explanation for why the app needs access to the camera and gallery. It is considered a bad practice to prompt the permission request menu a second time to the user. The UI features therefore two buttons that allow either quit the app or jump to the devices settings app, where the user can ultimately grant access to the camera and gallery. Both activities described above are depicted in figure B.3.
5.3. App Architecture

ViewFinder Activity

The ViewFinder activity can be considered the app’s core activity. Its UI features a CameraX PreviewView object that takes up the majority of the screen and is used as live output for the camera’s view finder. The circle located in the middle of the view finder indicates the region to the user that the camera automatically focuses on. Note that the app starts classifying the view finder’s frames by default. Double-tapping anywhere on the PreviewView will freeze its current frame and pause the classification which might be helpful to evaluate the classification results in more detail.

At the bottom end of the layout we find a so called BottomSheet which is a Material Design component that is useful for elegantly displaying supplementary information and content to the user. It is by default in its collapsed state that only displays the top 3 classification results over the last 10 frames. If the user would like to access additional information, for example on the per frame classification results, they can easily expand the BottomSheet by swiping up. The component holds furthermore the app’s settings and a button that allows the user to switch to the CameraRoll activity that is described below. The model selector, camera controls as well as the other UI elements in the BottomSheet are implemented as Fragments.

The RadioGroup in the ModelSelectorFragment allows the user to choose between different models. Once a change in configuration is picked up by the event listener in the fragment’s onCreate() method, the new configuration is persistently saved to the app’s SharedPreferences object and the ViewFinder activity gets notified. The camera controls, where the user can switch between the front- and rear-facing lenses and toggle the flash, as well as the settings for the threads and processing unit used for running inference are implemented in a very similar, Fragment based, way.

CameraRoll.java

The CameraRoll activity is the counterpart to the ViewFinder activity. Here, the user can select an image from their devices gallery that is subsequently classified by the currently selected model. The activity’s UI features a BottomSheet as well where the classification results are displayed. The user can return to the ViewFinder activity by pressing the return button on their device. Figure B.2 shows the activity before and after an image was selected from the gallery and classified.

However, the image classification process from obtaining an image to the top-5 predicted classes does involve a number of additional aspects, that are not necessarily related to the ViewFinder or CameraRoll activities. We will look into that process in more detail in the following section.
5. App Development

5.3.2. Other notable Classes

The app relies on a number of classes that are related to the UI or provide other helper functionalities for the dynamic approach of the app. I introduce the most relevant ones in this section.

ModelConfig.java

We use this class as an internal representation of the objects specified in nets.json in the app’s /assets directory. To create a ModelConfig object, we simply call its constructor and pass it a json object from the nets array. The class additionally offers getter and setter routines that are, for example, used when a classifier configuration is saved to SharedPreferences.

Please note that any additional "." in a model’s filename besides the one used for separating name and extension may result in the model being compressed for some unknown reason when the app is built. As a result the app will likely crash when the model gets selected.

ListSingleton.java

As the name suggests, this class is built around the popular singleton pattern that restricts its instantiation to a single instance. We use the ListSingleton as a single source of truth for the models specified in nets.json. The instance is created at app launch. It parses the json file and creates a ModelConfig object for each element in the json’s nets array, that it stores in a list like so:

```java
private List<ModelConfig> list = newListFromJSON();
```

This is an essential component to the dynamic approach of the app. The singleton pattern allows us to use the list of models from different modules. For example, it is used to dynamically create the RadioGroup of available models in the ModelSelectorFragment. Note that the app permanently contains the Float MobileNet v1 model to guarantee a use case with minimal functionality, even when there are no custom models.

Classifier.java

The Classifier class can be considered the app’s backend. Here, we create a new TF-Lite Interpreter object that eventually runs the model on an image by calling the public constructor of the class.

However, for that we first obtain the number of threads and the processing unit currently specified by the user. We then load the ModelConfig of the currently selected model, that allows us to obtain the associated labels file and to eventually map the corresponding
model from the /assets directory to memory. Note that this additional step is necessary due to the dynamic list of available models. A regular implementation that has a fixed number of models would typically use subclasses for that. Finally, we create the Interpreter like so:

```java
Interpreter tflite = new Interpreter(tfliteModel, tfliteOptions);
```

FreezeAnalyzer.java

This class handles freezing the PreviewView in the ViewFinder activity when the user pauses the classification by double-tapping the screen. There are numerous approaches to this, yet by utilizing a dedicated CameraX analysis use case this should be the most reliable solution. We use a custom Interface for the communication between ViewFinder.java and this class. When the activity detects a double tap, it notifies the FreezeAnalyzer class, where a flag is set.

When the flag is not set, the CameraX analysis use case, that runs in a separate thread, does nothing but image.close() to keep the processing pipeline open. However if the flag is set, the last frame is converted to a RGB bitmap and passed back to the ViewFinder activity, where we hide the PreviewView and display a ImageView with the bitmap instead. The reset procedure is analogous.

5.4. Image Classification Flow

We finally conclude this section with a detailed look on the process of how an image is processed so that we eventually obtain the associated class scores. Since the procedure is very similar for images from the user’s gallery, we will focus on the use case of the ViewFinder activity. Figure 5.3 visualizes the five steps of the process. Please note that the code snippets below are often simplified and would not actually run.

We find that the majority of image classification models expect images in the RGB color space as input. However, as mentioned before, CameraX and Android in general use the YUV format as standard for encoding images that were either captured by a device’s camera or obtained from the gallery.

To get started, we register a CameraX ImageAnalysis object to the camera’s lifecycle in the ViewFinder activity. Note that this use case is run in a separate thread to reduce the load on the main thread that runs the activity. The void analyze(ImageProxy image){...} method of the use case is called for each captured view finder frame. We first convert the ImageProxy into an Android YUV Image by calling:

```java
Image img = image.getImage();
```
Next, we crop the YUV Image into a square and convert it into a RGB Bitmap by applying a lossy JPEG compression algorithm. This allows us to reduce the memory overhead.

```java
yuvImage.compressToJpeg(new Rect(a, b, c, d), quality, outputByteBuffer);
```

The resulting RGB Bitmap can now be passed on to the classifier, that will eventually run the model on it.

```java
[...] classifier.recognizeImage(rgbBitmap);
```

Here, the Bitmap is transformed a second time. Since the Classifier class has access to the currently selected model, it uses this information to scale the image to the appropriate width and height, typically $224 \times 224$ pixels. Next, we serialized the sampled-down image into a buffer that then is mapped to memory.

```java
inputImageBuffer = loadImage(bitmap);
```

Before we pass the image buffer on to the TF-Lite Interpreter, we need to set up an additional buffer for the model’s output.

```java
TensorBuffer outBuffer = TensorBuffer.createFixedSize([1, 1000], probDataType);
```

Finally, we run the Interpreter:

```java
tflite.run(inputImageBuffer.getBuffer(), outBuffer.getBuffer().rewind());
```

We now have to transform the output buffer into a map structure and aggregate the returned class scores with their associated labels.

```java
Map<String, Float> labeledProb = new TensorLabel(labels, outBuffer).getMap();
```

We then sort that structure by prediction confidence and return a list that holds the top $n$ predictions to the caller. Back in the ViewFinder activity, we pass the list on to the Fragments that display the classification results. Here, we finally render the classification results as well as the inference latency to the screen.

```java
predictionsFragment.showRecognitionResults(results, startTimestamp);
```
Figure 5.3.: Visualization of the five step image classification process.
Part III.

Conclusion
6. Conclusion

The focus of this thesis project was on the applied end of computer science where I implemented an Android application that allows you to quickly deploy a TF model for image classification to a mobile device for evaluation purposes. I approached the project by making myself familiar with the basics of programming in Java for Android and created a first prof of concept app that could access a device’s camera and process the view finder’s frames. Then, I got myself into the field of machine learning based computer vision, where I largely relied on the excellent resources from [30]. At the same time, I started to experiment with the TF-Lite library that I discovered during my research on designing and creating neural networks.

The project gave me the opportunity to work on mobile development and computer vision, topics that I have always been interested in, yet didn’t had the chance to look into during my previous studies.

6.1. Result

The result of this project is the TUM Lens app that allows you to deploy a TF-Lite model to your Android device in three simple steps:

1. Place a converted model in the /assets directory of the app.
2. Open the nets.json file and add your model to the "nets" array.
3. Build the app with Android Studio and install it on your device.

You find the app in the project repository [8]. Please refer to the readme.md file for a more elaborate description of the model conversion and deployment process.

6.2. Future Work

In the future I would like to explore a way to deploy TF-Lite models to the app over the air so that the re-building step can be omitted. Moreover, given that TF-Lite supports multiple platforms, I think that creating apps with a similar use cases for other operating systems, such as Apple iOS, would pose interesting and fun thesis projects as well.
Appendix
A. TensorFlow-Slim Model Definition
import tensorflow.compat.v1 as tf
import tf_slim as slim

with tf.variable_scope(
    'vgg_16', [inputs], reuse=reuse) as sc:
    end_points_collection = sc.original_name_scope + '_end_points'
    with slim.arg_scope([slim.conv2d, slim.fully_connected, slim.max_pool2d],
        outputs_collections=end_points_collection):
        net = slim.repeat(inputs, 2, slim.conv2d, 64, [3, 3], scope='conv1')
        net = slim.max_pool2d(net, [2, 2], scope='pool1')
        net = slim.repeat(net, 2, slim.conv2d, 128, [3, 3], scope='conv2')
        net = slim.max_pool2d(net, [2, 2], scope='pool2')
        net = slim.repeat(net, 3, slim.conv2d, 256, [3, 3], scope='conv3')
        net = slim.max_pool2d(net, [2, 2], scope='pool3')
        net = slim.repeat(net, 3, slim.conv2d, 512, [3, 3], scope='conv4')
        net = slim.max_pool2d(net, [2, 2], scope='pool4')
        net = slim.repeat(net, 3, slim.conv2d, 512, [3, 3], scope='conv5')
    # Use conv2d instead of fully_connected layers.
    net = slim.conv2d(net, 4096, [7, 7], padding=fc_conv_padding, scope='fc6')
    net = slim.dropout(net, dropout_keep_prob, is_training=is_training, scope='dropout6')
    net = slim.conv2d(net, 4096, [1, 1], scope='fc7')
    # Convert end_points_collection into a end_point dict.
    end_points = slim.utils.convert_collection_to_dict(end_points_collection)
    if global_pool:
        net = tf.reduce_mean(input_tensor=net, axis=[1, 2], keepdims=True, name='global_pool')
        end_points['global_pool'] = net
    if num_classes:
        net = slim.dropout(net, dropout_keep_prob, is_training=is_training, scope='dropout7')
        net = slim.conv2d(net, num_classes, [1, 1],
            activation_fn=None,
            normalizer_fn=None,
            scope='fc8')
        if spatial_squeeze:
            net = tf.squeeze(net, [1, 2], name='fc8/squeezed')
        end_points[sc.name + '/fc8'] = net
    return net, end_points

vgg_16.default_image_size = 224

Source Code A.1.: Section of the VGG16 network defined in TF-Slim [1].
B. App Screenshots
Figure B.1.: Screenshots of the ViewFinder activity with collapsed and expanded BottomSheet.
Figure B.2.: Screenshots of the CameraRoll activity before and after selecting an image for classification.
Figure B.3.: Screenshots of the StartScreen activity (a,b). (c) shows the explanation in the PermissionDenied activity.
Bibliography


Bibliography


