BABEŞ-BOLYAI UNIVERSITY

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

COMPUTER SCIENCE SPECIALIZATION IN ENGLISH

DIPLOMA THESIS

Optimizing Convolutional Neural Networks for low-resource devices

Scientific supervisor

Prof. dr. Czibula Gabriela

Author

Rusu Cosmin-Ionuţ

2018

UNIVERSITATEA BABEŞ-BOLYAI
FACULTATEA OF MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ÎN LIMBA ENGLEZĂ

LUCRARE DE LICENŢĂ

Optimizarea reţelelor neuronale convoluţionale pentru dispozitive cu resurse limitate

Conducător ştiinţific
Prof. dr. Czibula Gabriela

Absolvent
Rusu Cosmin-Ionuţ

2018

Contents

List of publications **5**

Introduction **6**

1 Artificial Neural Networks **8**
1.1 Supervised learning 8
1.1.1 Examples of supervised learning 8
1.1.2 Training data 9
1.1.3 Feature selection 9
1.1.4 Feature scaling 9
1.1.5 Model selection 10
1.1.6 Training 10
1.1.7 Testing 10
1.2 The anatomy of a neuron 10
1.2.1 The perceptron 10
1.2.2 Adaptive Linear Neurons and the Delta Rule 12
1.2.3 Gradient Descent 12
1.2.4 Multiclass classification 14
1.3 Multi-layered networks 15
1.3.1 MNIST Classification case study 16
1.4 Convolutional Neural Networks 17
1.4.1 Architecture overview 18
1.4.2 Convolutional Layer 19
1.4.3 Pooling Layer 20
1.4.4 Fully-Connected Layer 20
1.4.5 Designing architectures 21

2 Optimizing CNNS for low-resource devices **22**
2.1 Introduction 22
2.2 Problem importance and motivation 24
2.2.1 Related work 24
2.3 Methodology 25

2.3.1 Data preprocessing 26
2.3.2 Training 26
2.3.3 Distillation 26
2.3.4 Evaluation 27
2.4 Experimental evaluation 29
2.4.1 TensorFlow Lite 29
2.4.2 Data sets 29
2.4.3 Experimental setup and description 29
2.4.4 Results 30
2.5 Discussion and comparison to related work 32
2.6 Conclusions and future work 35

3 Software Application **36**
3.1 Software Development 36
3.1.1 Problem definition and specification 36
3.1.2 Analysis and Design 36
3.1.3 Implementation 39
3.1.4 User's Manual 40
3.2 Future improvements 47

Conclusions and future work **48**

List of Figures

1.1 Linear decision boundary for binary classification 11
1.2 The perceptron 12
1.3 Gradient descent 13
1.4 Large learning rate: overshooting the minima, fail to converge 15
1.5 Small learning rate: Many iterations until convergence, trap in local minima 15
1.6 ANN architecture for handwritten digits classification 16
1.7 Softmax layer for multi-class classification problems 17
1.8 CNN architecture [cs217] 18
1.9 Downsampling the volume [cs217] 20
1.10 Single slice pooling [cs217] 20
1.11 Sequence of layers [cs217] 21

2.1 Pipeline of the proposed methodology for running experiments 26
2.2 Accuracy variation during all ten runs, for each SimpLeNet architecture. . 31
2.3 $F\text{-}measure$ vs $CompRate$ for all SimpLeNet architectures. Standard error bars are added for the results. 33
2.4 SimpLeNet architectures comparison in terms of accuracy and trainable weights 33
2.5 Variation of $F\text{-}measure$ for different values for T for different SimpLeNet architectures. 34

3.1 Samples for each category on MNIST dataset [LC10] 37
3.2 Samples for each category on CIFAR-10 dataset [KH09] 37
3.3 Architecture for running experiments with different models 38
3.4 Mobile application architecture 39
3.5 TensorFlow Lite architecture [tfl18] 40
3.6 Command line utility for the experiments pipeline 41
3.7 Training models from the command line utility 41
3.8 Fast iterations with Jupyter notebook 42
3.9 Camera permission request 43
3.10 Permission granted 43
3.11 App starts making real time inferences 43

List of Tables

2.1 Teacher model. 30
2.2 SimpLeNet models (v1 and v5). 30
2.3 Experimental results. 31
2.4 Detailed results for SimpLeNet-v1 trained with T=1 and SimpLeNet-v5 trained with T=3. A CI of 95% was used for the average. 32
2.5 Results obtained for SimpLeNet-v1 and different values for the temperature hyperparameter. 34
2.6 Improvements on using distillation 34

List of publications

[RC18] **Cosmin-Ionuţ Rusu** and Gabriela Czibula. *Optimizing Convolutional Neural Networks for low-resource devices*. Proceedings of the 14th International conference on Intelligent Computer Communication and Processing, Cluj-Napoca, Romania, 2018, under review (**ISI Proceedings**).

Introduction

The purpose of the present thesis is to demonstrate the applicability of convolutional neural networks for low resource devices and to study their performance in real life scenarios.

Given the increasing number of portable devices, the problem of creating lightweight machine learning models is of strong interest nowadays. The main challenges gravitate around the very limited environment these devices live in, such as low resources (battery, memory, CPU, GPU) and low network connectivity (low bandwidth is usually the case). The challenges persist even on high end devices. In addition, we think that we should find better ways to overcome this problem instead of just throwing more hardware components at it.

Many small and large companies already saw the widely-available smart devices as an opportunity for *on-device machine intelligence*, hence their interest in such technology is obvious. Internet of Things is one trend that can serve as use cases for lightweight machine learning models.

Concern in stronger *user data privacy* where this data does not need to leave the device is another major issue addressed with such models. Having the possibility for the machine learning models to serve offline cases, where the device does not need to be connected to a network, could impact more users (especially from countries where internet connectivity is still limited or inaccessible). For example, by integrating different features in the operating system of a mobile phone.

The main contribution of the thesis is to introduce a convolutional neural network model, called SimpLeNet, using distillation for image tagging, that can run on low-resource devices such as smartphones, smartwatches, tablets or TVs. Our major goal is that of preserving the performance of the convolutional neural network. For emphasizing the effectiveness of SimpLeNet, both in terms of model's size reduction, as well as in terms of classification accuracy, several experiments are performed on various data sets for image classification.

Our second contribution consists of designing and implementing an application capable of making real time image classification directly on the device, without accessing the internet. The application also tracks the time took in order to make inferences so we can exactly measure its performance.

The remainder of the thesis is structured as follows.

Chapter 1 presents an overview of the theory and intuition behind Supervised learning. We described the process of designing successful machine learning models, starting from training data to feature selection and testing. Then, we introduced the concept of a single

neuron and how it can be used as a linear binary classificator. Going forward, we showed how these neurons can be groups together in layers of neurons that can be stacked together in order to create an artificial neural network. Finally, we presented a new type of neural network designed especially for image recognition tasks: convolutional neural networks.

Chapter 2 proposes an approach to optimize Convolutions Neural Networks for low-resource devices. We designed new compressed architectures using a transfer learning technique called distillation [HVD]. We were able to considerably reduce the number of trainable parameters while still preserving a very good accuracy.

In Chapter 3 we demonstrate the practicability and applicability of Convolutional Neural Networks on Android-based smartphones, by implementing a real-time on-device image classificator. The application takes a FlatBuffer representation of the model and a configuration file containing the class labels. Then, it feeds the camera images into the model, showing the top 3 predictions alongside the elapsed time.

We mention that the original part of this thesis is contained in Chapter 3 and was published in the research paper [RC18].

Chapter 1

Artificial Neural Networks

This section will give an introduction into artificial neural networks, starting from the supervised learning method all the way to convolutional neural networks. We will be focusing on both the theoretical and practical concepts.

1.1 Supervised learning

Within the machine learning domain, *supervised learning* represents the problem of learning to approximate a mapping from inputs to outputs by generalizing from a set of training data consisting of labeled examples. The training examples are input-output pairs provided by an external supervisor (teacher) and map particular inputs to the desired or target response for outputs. Supervised learning was proved to be very effective in problems never thought to be solvable. The amount of unstructured data that we are able to collect and store these days are what empowers all the models that perform tasks such as object classification, facial detection and disease detection in the healthcare sector.

In a nutshell, suppose we have a list of some input X that can represent any kind of structured data (integers, vectors, images) and we know for each X its target, y.

A supervised learning will try to learn a mapping f such that $f(X)$ approximates very well the target y. How we define "very well" it's up to us, in terms of how we measure the accuracy and the error of our function - often referred to as the model.

The process of learning the mapping function f is called training. After training, the model is able to make predictions for some X values never seen before. This step is also referred to as inference.

1.1.1 Examples of supervised learning

Supervised learning problems are split into two main categories:

- **Classification** This is the case where the targets y are concrete values. Example: Classify a set of images into pictures containing a car, a truck or an airplane.

- **Regression** In this category, the y target that we want to learn is a real value. One example is determining the prices of a house given the number of rooms, bathrooms, kitchens and the number of square meters it has.

1.1.2 Training data

In order to train a supervised learning algorithm, we need to gather some labeled data. This may be timed data regarding the price of some stocks, tagged images or the prices for some houses. Mathematically, this is equivalent to creating the set

$$TrainingSet = \{(X, y) \mid X \text{ is a feature vector}\}$$

.

1.1.3 Feature selection

How we define the feature vector depends on the problem we are trying to solve. For example, if we want to detect the prices for some houses, some possible features can be the square footage, the number of kitchens or the location. All of these and even more are suitable. These features must be easy to collect (sometimes is really hard to gather all the values for the different features) and we should make sure they are linearly independent (we should not add both the house square footage and the exact values of its dimensions since they correlate). The whole process of finding a what values to consider as features is an important step in designing a good supervised learning model.

1.1.4 Feature scaling

The features do not have the same range of values. In order to overcome this problem, we will scale every feature. The easiest way to accomplish this is to map the features in the range $[0, 1]$ or $[-1, 1]$. Selecting the target range, however, depends on the nature of the data.

- **Rescaling**
$$x' = \frac{x - min(x)}{max(x) - min(x)}$$
- **Mean normalization**
$$x' = \frac{x - mean(x)}{max(x) - min(x)}$$
- **Standardization**
$$x' = \frac{x - \overline{x}}{\sigma}$$
- **Scaling to unit length**
$$x' = \frac{x}{\|x\|}$$

1.1.5 Model selection

This is yet another important part of successfully using supervised learning to solve problems. Simply put. model selection is simply defining the function f. Besides having an input X, our model function also depends (in a variety of ways) on some parameters (or weights) which is exactly what our models will learn. We can reason about this in a simple fashion: our model is simply a function which combines the weights with the inputs in some way. Since our training data is immutable by definition, we can only change the weights of our model. We will change these weights in such a way that we minimize a cost function which we'll define next.

1.1.6 Training

On the training set, we are defining a cost function. That is, we defined our optimization problem and use different algorithms to tweak our model in order to minimize the cost function. For the house pricing prediction, a simple cost function could be:

$$Cost = \sum_{(X,y) \in TrainingSet} (f(X) - y)^2$$

.

1.1.7 Testing

It is important to make sure the model generalize on unseen data. In order to ensure this, we will split our data set into two: one that will be used for training purposes and the one that will be used for testing how well our model is doing.

1.2 The anatomy of a neuron

Continuing with the function analogy, we can define a simple model, also called the perceptron or simply, a neuron. The perceptron is the first algorithmically described learning algorithm [Ros57]. It is very intuitive and represents the basic unit of an Artificial Neural Network.

1.2.1 The perceptron

Using the analogy of a human neuron, that simply fires in order to propagate some information further, a perceptron is simply a function which takes the input features, multiplies them with some weights W and return a binary value representing whether or not the neuron has fired. The learning algorithm will try to find the optimal values for the weight vector. Such algorithm could be used to determine if a data point belongs to a class or another - also called a **binary classification task**.

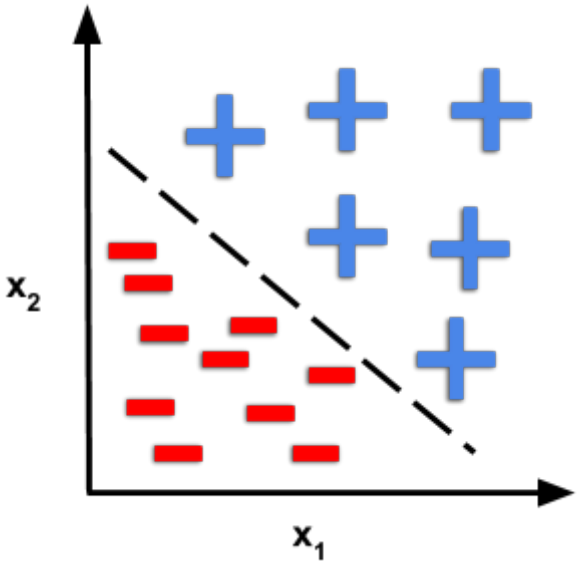

Figure 1.1: Linear decision boundary for binary classification

Let us label the positive and negative classes in our binary classification as 1 and -1, respectively. Let X be a high dimensional vector and y some binary value associated with X, the $target$. Next, we will define the activation function g that takes the linear combination of X and the weights vector W as input and predict 1 if that input is greater than or equal to some threshold θ and -1 otherwise.

$$g(z) = \begin{cases} 1 & : \text{z} \geq \theta \\ -1 & : \text{otherwise} \end{cases}$$

where

$$z = \sum_{i=1,m} w_i * x_i = W^T * X$$

We can further simplify the notation, by bringing θ to the left side of the equation and define $w_0 = -\theta$ and $x_0 = 1$, obtaining the so called **Unit step** activation layer. Therefore:

$$g(z) = \begin{cases} 1 & : \text{z} \geq 0 \\ -1 & : \text{otherwise} \end{cases}$$

where

$$z = \sum_{i=0,m} w_i * x_i = W^T * X$$

The idea behind this thresholded neuron was to imitate how a single neuron in the brain works: it either fires or not. Frank Rosenblatt's idea [Ros57] of learning the weights for the input values in order to draw a linear decision boundary that allows us to discriminate between the two linearly separable classes is what makes this a learning algorithm.

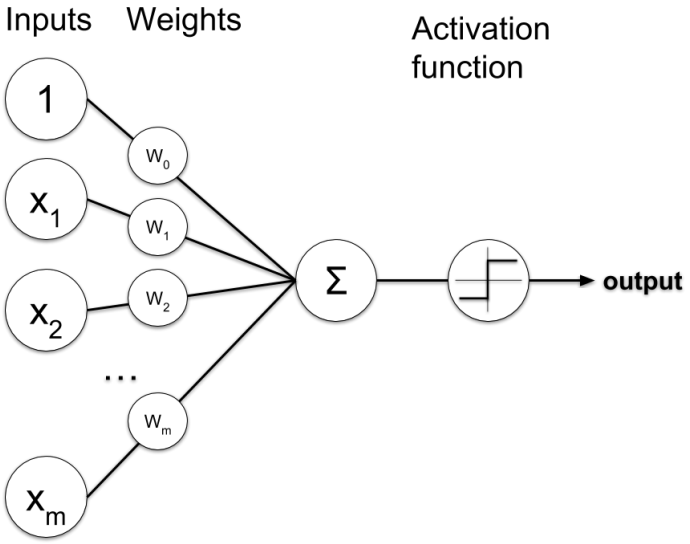

Figure 1.2: The perceptron

The learning algorithm is fairly simple and can be summarized as follows:

```
1 Initialize the weights to 0 or small random numbers.;
2 for each training sample x^(i) do
3 |   Computer the output value.
4 |   Update the weights.
5 end
```

Algorithm 1: Rosenblatt's perception rule

The output values is the class label predicted by the activation function defined earlier ($g(z)$) and the weight update can be written formally as $w_j = w_j + \delta w_j$. The value for updating the weights at each step is computed by the learning rule:

$$\Delta w_j = \eta(target^{(i)} - output^{(i)})x_j^{(i)}$$

where η is the learning rate (a constant between 0.0 and 1.0).

1.2.2 Adaptive Linear Neurons and the Delta Rule

The adaptive linear neuron takes further the idea of the perceptron and improves it. In contrast to the perceptron rule, the delta rule of adaline [W+60] updates the weights vectors based on a linear activation function rather than the unit step function. The activation function $g(z)$ is the identify function: $g(W^T * X) = W^T * x$.

1.2.3 Gradient Descent

The most important advantage of the linear activation function is the fact that it is differentiable. Let us define a cost function $J(W)$ that we want to minimize in order to update the

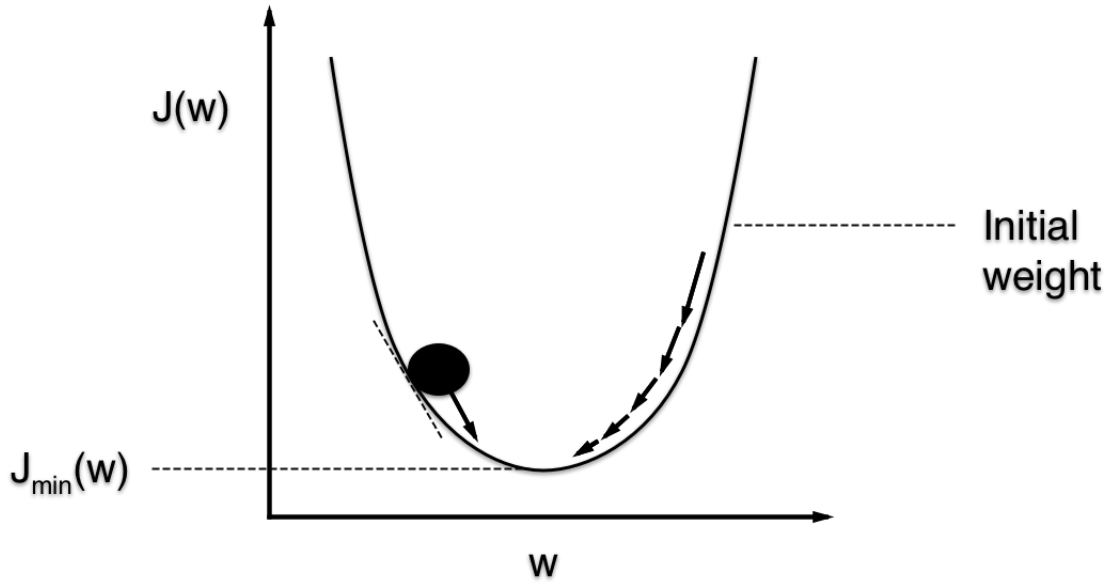

Figure 1.3: Gradient descent

weights. In the case of linear activation function, we can define $J(W)$ as the sum of squared errors as follow:

$$J(W) = \frac{1}{2} * \sum_i (target^{(i)} - output^{(i)})^2 \qquad output^{(i)} \in \mathbb{R}$$

The constant $\frac{1}{2}$ is used only for convenience to derive the gradient. Gradient descent is a simple yet powerful algorithm that is often used in machine learning to find the local minimum of linear systems.

For example, let us consider a convex cost function for one single weight values. We can imagine the principle behind gradient descent as climbing down a hill until a local or global minimum is reached. Every time we will take a step in the opposite direction of the gradient, and the step size will be determined by the value of the slope of the gradient and the learning rate.

To put it more formally, each update will be of the form $\Delta W = -\eta \nabla J(W)$. Now let us compute the partial derivative of the cost function for each weight in the weight vector: $\Delta w_j = -\eta \frac{\partial J}{\partial w_j}$

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i (t^{(i)} - o^{(i)})^2 \\
&= \frac{1}{2}\sum_i \frac{\partial}{\partial w_j}(t^{(i)} - o^{(i)})^2 \\
&= \frac{1}{2}\sum_i 2(t^{(i)} - o^{(i)})\frac{\partial}{\partial w_j}(t^{(i)} - o^{(i)})^2 \\
&= \sum_i (t^{(i)} - o^{(i)})\frac{\partial}{\partial w_j}\left(t^{(i)} - \sum_j w_j * x_j^{(i)}\right) \\
&= \sum_i (t^{(i)} - o^{(i)}) * (-x_j^{(i)})
\end{aligned} \tag{1.1}$$

where $t^{(i)}$ - the target, $o^{(i)}$ - the output

If we plug the result back to the learning rule, we get:

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = -\eta\sum_i (t^{(i)} - o^{(i)}) * (-x_j^{(i)}) = \eta\sum_i (t^{(i)} - o^{(i)}) * x_j^{(i)}$$

And the update rule is similar to the perceptron rule: $W := W + \Delta W$. However, there are two notable differences between the two:

1. Here, the output is a real value and not a class label as in the perceptron rule.
2. The weight update is computed based on all the samples in the training set, which is why is approach is often called **batch gradient descent**.

It is important to plot the learning curves, that is, how the cost is evolving over the iterations in order to avoid the two most common problems with gradient descent:

1. If the learning rate is too large, gradient descent can overshoot the minimum point and diverge.
2. If the learning rate is too small, the algorithm may require too many iterations to converge and can become trapped in local minima.

1.2.4 Multiclass classification

The single neuron is only able to discriminate between two classes. However, we can reduce the multiclass classification problems to multiple binary classification problems. Two main ideas stand out:

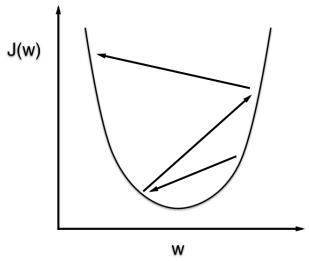

Figure 1.4: Large learning rate: overshooting the minima, fail to converge

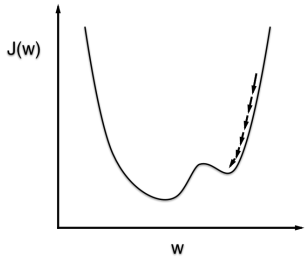

Figure 1.5: Small learning rate: Many iterations until convergence, trap in local minima

- **One vs all** involves training a classifier per class, with the samples of that class as positive and all other samples as negatives. This strategy requires each individual classifier to output a real valued confidence score.
- **One vs one** in which one trains $K(K-1)/2$ classifiers for a K-way classification problem. Each such classifier received the samples of a pair of classes and learns to distinguish between the two. At prediction, a voting algorithm is applied.

Although these two strategies are very popular, they suffer from several problems. First, the scale of confidence for the classifiers might differ. Second, the binary classifiers see unbalanced distribution because usually the set of negatives examples are much larger than the set of positives. Moreover, One vs one method suffers from ambiguity in the sense that some classes may receive the same number of votes.

1.3 Multi-layered networks

We can push the idea of the perceptron further and introduce layers of neurons. We can group together multiple neurons in layers, and stack multiple layers in order to get a **Neural Network**. The first layer would consist of the features (also called the input layer). Each neuron in the further layers will get the output of the neurons from the previous layer, combine them together using their activation function and then output another set of features.

An *artificial neural network* (ANN) [MCC18] is a supervised learning system which models a biological neural system. During its training phase, an ANN adapts its parameters in order to best fit the training data. The error composed from the difference between the desired output and the system output is fed back into the network and is used for adjusting the ANN parameters through the learning rule. The inductive learning algorithm used for training ANNs is the *backpropagation* algorithm [Mit97].

In such neural networks, besides unit and linear activation function, other activation function have been successfully used in. Notably, the sigmoid function $g(z) = \frac{1}{1+e^{-z}}$, Tanh function $g(z) = \frac{2}{1+e^{-2z}} - 1$ and ReLU (rectified linear unit) $g(z) = max(z, 0)$. The last one have

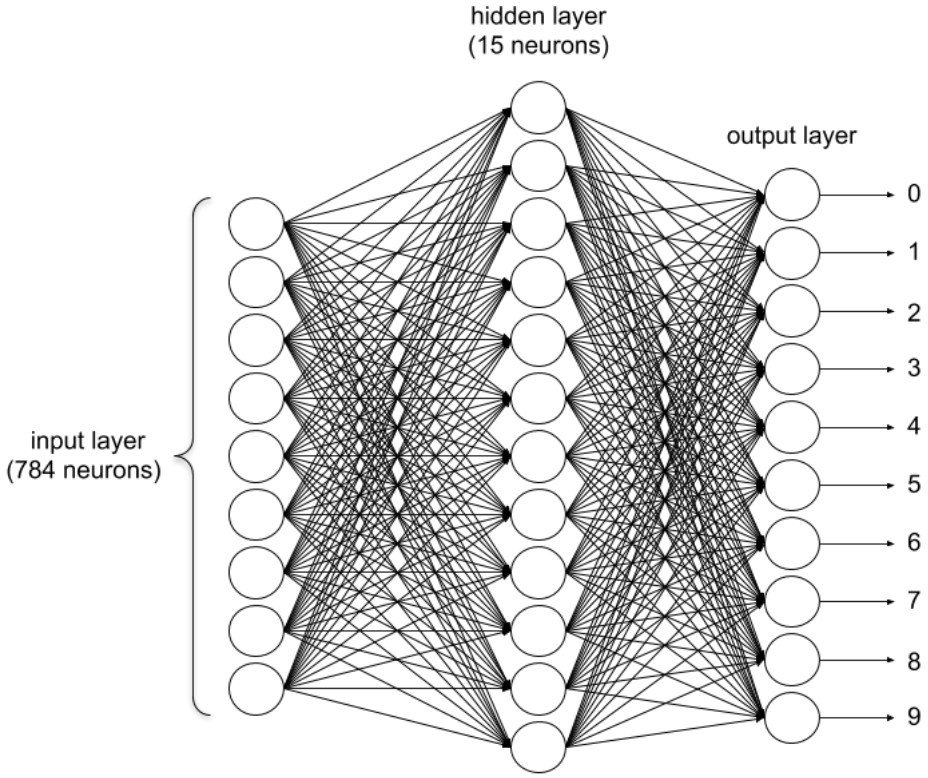

Figure 1.6: ANN architecture for handwritten digits classification

been proved to be very effective for Convolutional Neural Networks in the sense that it has lead to faster training time.

1.3.1 MNIST Classification case study

Let us describe a neural network that can be used to classify handwritten digits [LC10]. The network will have three layers of neurons: the first layer is the input layer, the second layer is also called the hidden layer and will compute higher order features, and the output layer is the output layer which will have 10 neurons (corresponding to each class). The output layer will use a softmax activation function so that the 10 real values of our neural network output is a probability distribution representing how confident we are that the sample belongs to each of the classes.

The hidden layer will use a sigmoid activation layer for both

The MNIST dataset [LC10] is a collection of greyscale 28 by 28 pixels images representing handwritten digits, having a training set of 60000 examples, and a test set of 10000 examples. The input layer for the neural network contains neurons encoding the $784 = 28 \times 28$ values of the input pixels. Since the images are greyscale, every input is going to be in the range $[0, 1]$ with 0 being white and 1 being black. The second layer of the network is a hidden layer, with 15 neurons such that each neuron is connected to the input neurons. Finally, the output layer of the network contains 10 neurons. Every neuron in the last layer represents

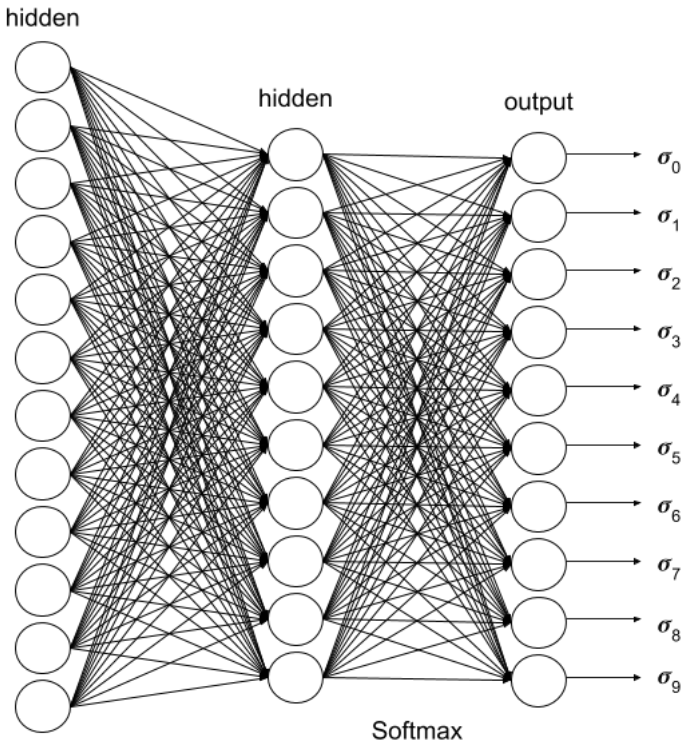

Figure 1.7: Softmax layer for multi-class classification problems

one of the 10 possible classes. This architecture is depicted in Figure 1.6.

We saw some activation function for a single neuron. For a single neuron, the sigmoid function $g(z) = \frac{1}{1+e^{-z}}$ is often used because it outputs a value between 0 and 1. For a binary classification, one can interpret that as the probability for the sample to belong to the first class. The **softmax** activation function is used in order to extend this idea to multiclass classification problems. That is, Softmax assigns probabilities to each class in a multi-class classification problem. These probabilities will add up to 1.0. This additional constraint helps training converge faster.

Softmax is added as a neural network layer just before the output layer. The Softmax layer will have the same number of nodes as the output layer. The softmax activation function is given by:

$$\sigma : \mathbb{R}^K \to \left\{ \sigma \in \mathbb{R}^K | \sigma_i > 0, \sum_{i=1}^{K} \sigma_i = 1 \right\}, \quad \sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}. \tag{1.2}$$

1.4 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [KSH12a] are a type of Artificial Neural Networks mainly specialized for input data in form of images. In general, a CNN enhances

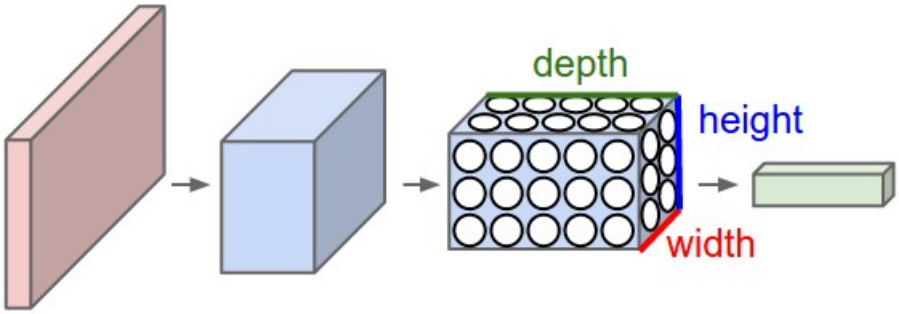

Figure 1.8: CNN architecture [cs217]

the classical ANN architecture by using several alternating convolutional and pooling layers, followed by 2-3 fully connected layers. The convolutional layers [SLJ+14] can be viewed as a trainable feature extractor, while the last traditional layers represent a trainable classifier.

When introduced, AlexNet [KSH12b] popularized convolutional neural networks by outperforming top models in tasks from computer vision and sound detection. They are widely used today in a variety of areas and they are one of the best tools that can be used out of the box for so many tasks. Research is still active in getting more insights into what features they learn during the training time. Olah, et al. [OMS17] contributed to discovering what behavior makes CNNs able to solve so complex problems, behavior that is very important in order to fight adversarial examples [SZS+13]. Capsule Network [HKW11] [SFH17] addresses some of the fundamental drawbacks CNNs have and we believe we are only seeing the tip of the iceberg with regard to these new types of architectures.

All these models are cumbersome with respect to the computational resource and power needed to both train and infer. For example, training AlexNet [KSH12b] takes from five to six days on GRX 580 3GB GPUs. The number of parameters for the same network is around 60M. Under no circumstance, we can deploy such models to low resource devices.

1.4.1 Architecture overview

Convolutional Neural Networks are similar to Artificial Neural Networks previously introduced: they are made up of neurons that have trainable weights and biases (a term that is added to the dot product of the input and weights). Each neuron takes some input, performs a dot product and optionally pipes it through an activation function. The main difference is that CNNs make the explicit assumption that the inputs are images, which allows them to encode certain properties into the architecture, vastly reducing the number of parameters in the network.

Regular Artificial Neural Networks do not scale well to full images. In CIFAR-10 [KH09], images are $32 \times 32 \times 3$ (32 pixels width, 32 pixels height, and 3 color channels RGB), so a single neuron in the first hidden layer would have $32 * 32 * 32 = 3072$ weights. This number still seems manageable, but clearly, this fully-connected structure does not scale to larger images. For example, consider an image of more respectable size, $200 \times 200 \times 3$, would

require the neurons to have $200 * 200 * 3 = 120000$ weights. Considering the fact that we have multiple neurons, the parameters would add up quickly.

3D Volumes of neurons. CNNs take advantage of the fact that the input images consist of images and the layers consist of neurons arranged in 3 dimensions: **width, height, depth**. The neurons in each layer are connected to a small region of the layer before it, instead of all the neurons in a fully-connected layer. Finally, the output layer for CIFAR-10 [KH09] have dimension $1 \times 1 \times 10$, representing a single vector of class scores arranged along the depth dimension. Figure 1.8 shows such an architecture.

As with ANNs, CNNs consist of a sequence of layers, and each layer transforms one volume of activations through a differentiable function. CNNs use three main type of layers: **Convolutional Layer**, **Pooling Layer** and **Fully-Connected Layer**. These layers are stacked together in order to form a ConvNet model.

1.4.2 Convolutional Layer

The convolutional layer is the core building block of CNNs. The convolutional layer consists of a set of learnable filters. Every filter is small spatially along the width and height but extends through the full depth of the input volume. During the forward pass, the filter convolves (slide) across the entire width and height of the input volume and compute the dot product between the input at any position and the weight entries for the filter. This rolling method will produce a 2-dimensional activation map representing the response for the responses of that filter at every spatial position.

Intuitively, the network will learn filters that activate when they encounter different visual features such as an edge, color combination on the first layer, and eventually high-level features such as head or wheel-like shapes patterns on final layers. Every convolutional layer will have an entire set of filters each of them producing separate 2-dimensional activation maps. By stacking these activation maps along the depth dimensions, the convolutional layer will produce the output volume.

Since connecting a neuron to every neuron from the previous layer is impractical when dealing with images, a neuron in a convolutional layer will be connected to only a small local region of the input volume. This spatial extent of this connectivity is a hyperparameter called the **receptive field** of the neuron, representing, in fact, the filter size. However, the size on the depth axis of the filter is always equal to the depth of the input volume.

Three other hyperparameters control the output volume of the convolutional layer: the **depth, stride** and **zero-padding**.

- The **depth** controls the number of filters we would like to use. Each learned feature will look for something different in the input. For example, the first filter could be looking for the presence of various oriented edges, or combinations of color.

- The **stride** defines how the filter is moved along the width and height on the input volume. If the stride is 1, we move the filter one pixel at a time. When the stride is 2,

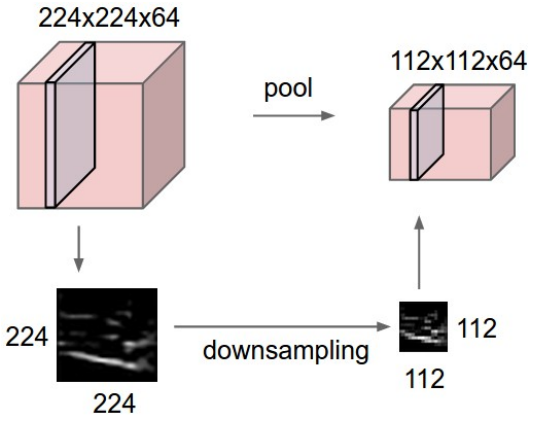

Figure 1.9: Downsampling the volume [cs217]

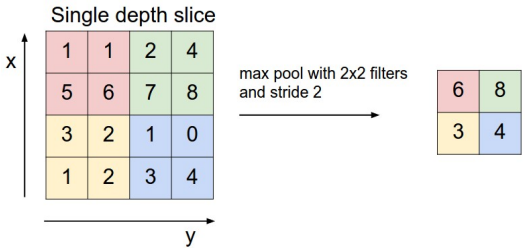

Figure 1.10: Single slice pooling [cs217]

then the filter jumps two pixels at a time as it slides around. The bigger the stride is, the smaller the output it will be.

- It is often convenient to pad the input volume with zeros around the border. This **zero-padding** hyperparameter controls the spatial size of the output volume.

Finally, we can compute the shape size of the output volume as a function of the input size W, the receptive field size F, the stride with which the filters are applied S, and the amount of zero padding used (P):

$$output(W, F, P, S) = (W - F + 2P)/S + 1.$$

1.4.3 Pooling Layer

The main role of the pooling layer is to reduce the spatial size of the input volume and, hence, the number of parameters and computation in the network. However, it was shown in [SFH17] that the max pooling layer is bad for the network because it loses a lot of information. The pooling layer operates independently on every depth slice of the input and resizes it spatially, using the max operation, hence it is also known as *max pooling layer*. The most common form is a max pooling layer with filters of size 2×2 applied with a stride of 2, downsampling every depth slice in the input by 2 along both width and height, discarding 75% of the activations.

In addition, the pooling units can also perform other functions, such as an average pooling, which was historically used, but fallen in favor of max pooling operation given the better results shown in practice.

1.4.4 Fully-Connected Layer

Neurons in a fully connected layer have connections to all the neurons in the previous layer, exactly as in regular Neural Networks. They are usually located at the end of the

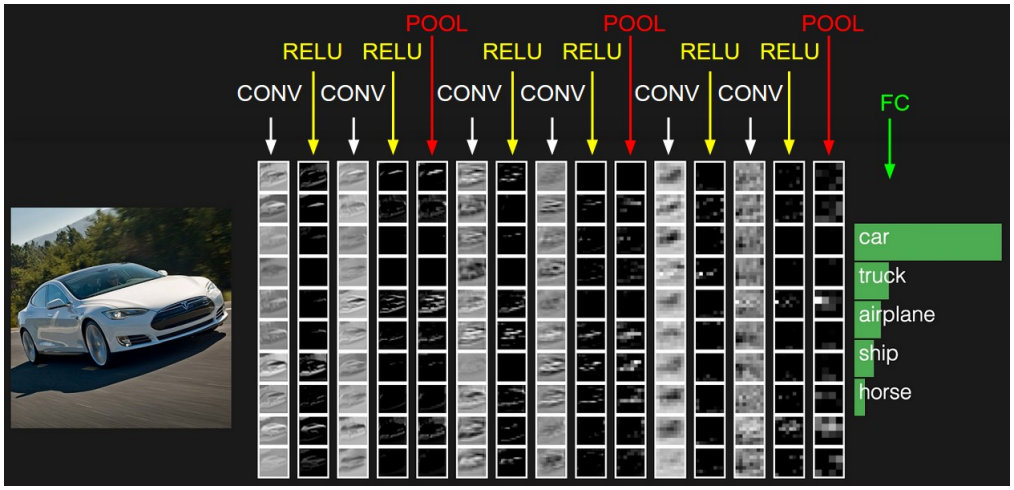

Figure 1.11: Sequence of layers [cs217]

sequence of layers, computing the final class scores.

1.4.5 Designing architectures

Convolutional Neural Networks are commonly made up of only three layer types: convolutional, pooling and fully connected. The activation function for the convolutional and fully connected layer is usually the Rectified Linear Unit (or short, ReLU), defined as $g(z) = max(0, z)$, and has been widely used in practice since it has shown that it helps the network converge faster.

Finally, the output layer is usually followed by a Softmax layer in order to force the network to output a probability distribution over the class scores.

Chapter 2

An approach for optimizing Convolutional Neural Networks for low-resource devices

Convolutional neural networks are effective supervised learning models which are widely used nowadays in various applications ranging from computer vision tasks such as image detection and classification, image captioning to video classification. Even if the convolutional models are highly performant, a major drawback is given by their computationally expensiveness from the viewpoint of the required memory, additions and multiplications operations and thus are hardly portable on limited-resource devices. The purpose of this chapter is to demonstrate the applicability of convolutional neural networks for low resource devices and to study their performance in real life scenarios. In this respect, with the major goal of preserving the performance, we proposed in the original paper [RC18] a convolutional neural network model, called SimpLeNet, using distillation for image tagging that can run on low-resource devices such as smartphones, smartwatches, tablets or TVs. Experiments performed on various data sets for image classification emphasize the effectiveness of SimpLeNet, both in terms of model's size reduction, as well as in terms of classification accuracy.

2.1 Introduction

Convolutional neural networks (CNNs) [KSH12b] and their performance in Computer Vision [SLJ+14] applications are one of the reasons Deep Learning is so popular today. CNNs have been successful in identifying faces, objects and traffic signs apart from powering vision in robots and self driving cars. Nonetheless, they do have their drawbacks and fundamental limitations. The main problem with CNNs is that they do not consider the orientational and relative spatial relationships between the learned features. Max-pooling layer [SDBR14] loses a lot of information about the image and, as a result, it is more prone to adversarial examples [SZS+13]. Professor Geoffrey Hinton addressed these problems and

proposed another deep learning model: the Capsule Network [SFH17]. Research towards this new type of architecture is at the very early stage. Their outstanding results in simple Computer Vision tasks demonstrate that they can achieve state-of-the-art performance at scale, but it requires more insights before out-performing a highly developed technology.

Distillation is a training method that is used to compress the knowledge learned by a very big or an ensemble of models into a smaller, lightweight, production-ready model, without losing accuracy. Moreover, it has been shown that the accuracy of this smaller model is increased if trained using this method compared to previous training strategies. The paper that introduced this concept [HVD] is especially relevant as we are trying to achieve the same goal: compressing the knowledge of a high performant model into one that is capable of running in low-resource conditions like smart phones or tablets.

Distillation [HVD] showed extremely good results on artificial neural networks for detecting handwritten digits on the MNIST data set, by transferring and compressing [BCNM06] the knowledge stored by a large model into a smaller one. In our approach, we are trying to use the same idea for a CNN in order to make them smaller in terms of number of computations performed on inference, with the end goal to make them portable to limited-resource devices.

Mobile Nets [HZC+17] [SHZ+18] were introduced with the goal of creating mobile-ready convolutional neural networks without impacting the overall device health (such as battery, RAM memory used, CPU and GPU usage). MobileNets changed the regular convolutional layer with a depthwise convolution that acts separately on channels followed by a pointwise convolution, hence reducing the number of additions and multiplications required to compute the output layer.

The main contribution of the current work is to develop a simpler, new convolutional neural network architecture, called SimpLeNet, for limited-resource devices such as mobile phones as well as experimenting methods for training this model. For optimizing the learning models, we will use the process called *distillation* [HVD], which refers to transferring and compressing the knowledge stored by a large model or an ensemble of models into a smaller one that is more suitable for deployment. Due to the very good performance of distilling the knowledge in neural networks [HVD], we aim to preserve the accuracy of a larger CNN model, while reducing the number of layers and nodes. The approach proposed in this chapter is new, as we tried to squeeze the architectures as much as we can, while still preserving accuracy that compare with state-of-the-art models.

To summarize, the purpose of the research conducted in this chapter [RC18] is to answer the following research questions:

RQ1 What optimizations are appropriate for increasing the performance of SimpLeNets in terms of number of weights to be trained and inference time (number of additions and multiplications)? How effective is to replace the CNN's max-pooling-layer with a convolutional filter of the same size, with increased stride?

RQ2 To what extent are the SimpLeNet architectures able to preserve the accuracy of the larger model on image classification problems?

RQ3 How do the SimpLeNet model compare to other models in terms of performance and model size?

2.2 Problem importance and motivation

Lightweight machine learning models are of strong interest nowadays given the increasing number of portable gadgets. This crucial problem poses a lot of challenges since the computational power of such devices is very limited. Even high end devices can be affected by cumbersome models in areas ranging from battery life to memory, CPU and GPU resources. We also want to make sure our models performs extremely well in low bandwidth environments which is the case with most devices. Widely-available smart devices create new opportunities for *on-device machine intelligence*. Many small and large companies are seeking these new possibilities, hence their interest in such a technology is obvious.

Concern in stronger *user data privacy* where this data does not need to leave the device is another issue that we address with such models. Moreover, another possible use case is the ability for the machine learning model to serve offline, where the device does not need to be connected to a network. This is beneficial since there are still a lot of countries where internet connection is limited and so machine learning models that can be built in the operating system of a mobile phone, for example, could impact more users.

Internet of Things is still another trend that can serve as use cases for lightweight machine learning models.

2.2.1 Related work

An efficient way to increase the performance of a learning model and to achieve a low-biased model is to train an ensemble of models and to make predictions based on the median of the ensemble's models. Unfortunately this is cumbersome and not feasible if we then want to ship this model to software that runs on limited-resources devices.

Hinton et al. introduced in [HVD] a simple way to transfer the knowledge gathered by an ensemble of models in a single significantly smaller model. The method allows the "distillation" of lots of information learned by a model in another model, a concept similar to archiving (the information is not lost, but it is reduces in terms of its size). The idea behind distillation is that the smaller model is being taught to mimic the output of the bigger model, when having the same input.

Howard et al. proposed in [HZC+17] a method of decomposing a convolutional layer in two smaller convolutional layers, considerably reducing the number operations necessary to train and make inferences. A handful of MobileNets were developed, each trading between the accuracy on the ImageNet data set and the number of additions and multiplications

needed to make an inference. One of the model achieved an accuracy of 70.6%, comparable to VGG16 [SZ14] [LD15], while being 32 times smaller.

Sabour et al. [SFH17] address some of the fundamental limitations regarding convolutional neural networks and proposed an alternative: Capsule Networks. They have successfully outperformed CNNs on the MNIST [LC10] data set. However, they are still under development and the time it takes to train and infer does not qualify them for a low resource environment.

2.3 Methodology

This section introduces our CNN based SimpLeNet architecture proposal that can run on low-resource devices such as smartphones, smartwatches, tablets or TVs, as well as the methodology employed in our research.

With the main focus of preserving the performance, the proposed SimpLeNet model uses *distillation* for transferring and compressing the knowledge stored by a large CNN model or an ensemble of CNN models into the smaller SimpLeNet model. Instead of training the smaller model to classify the MNIST [LC10] using the hard targets, we will train the model to mimic the probability distribution received from the teacher model. By using this way we transfer more information that we would if we used hard targets. As humans, we know that 1 looks like a 7 but it's slightly rotated. That's exactly the kind of information we teach the small model. SimpLeNet can be trained with unlabeled examples which are fed to the pretrained larger model to output probabilities, and then SimpLeNet is trained to match them [HVD]. The main advantage of this process is that SimpLeNet can be trained on very large data sets, without requiring human labeling for the training data, under the assumption that the larger model is a pretrained one.

The following improvements are targeted in order to optimize the models: (1) changing the max pooling layer with a convolutional layer with a stride equal to the size of the max pooling filter; (2) developing an automated pipeline for testing different models; (3) extracting and measuring performance improvements for each model in terms of accuracy and number of trainable weights in the model.

Experiments will be performed on classical data sets for image classification with the main challenge of reducing the size of the SimpLeNet model whilst preserving the accuracy of the classification process.

Figure 2.1 illustrate an overview of our proposal consisting of four main stages which will be further detailed.

1. **Data preprocessing.** This stage consists of gathering the data.

2. **Training (building the models).** This stage involves experimenting with creating simple, lightweight models and train them using distillation, the teacher being a cum-

bersome model or an ensemble of models. For simplicity, we used only one teacher and we measured it's performance in terms of accuracy.

3. **Distillation.** This step consists of distillating the knowledge from a cumbersome model to the smaller SimpLeNet model. First, we compute the soft targets for each MNIST data example and then we train our smaller models to match those targets.

4. **Evaluation (testing).** Here we evaluate the model from the perspective of the accuracy and its dimension. We also compute the compression rate that is introduced in the Section VI.

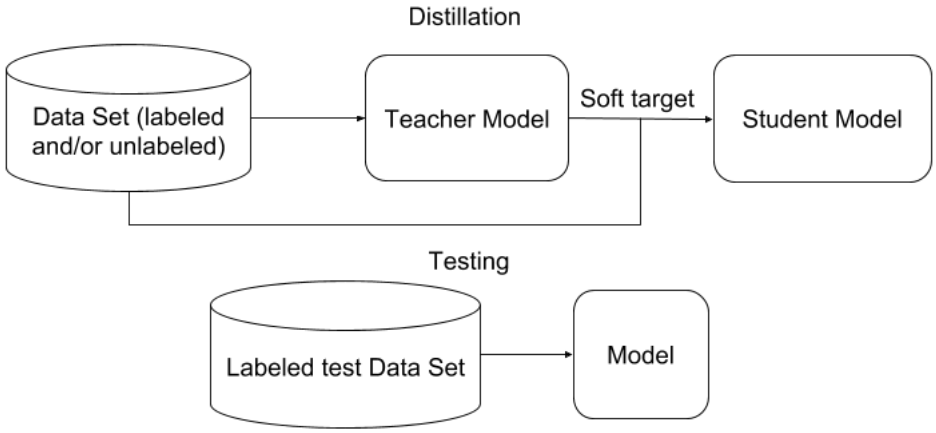

Figure 2.1: Pipeline of the proposed methodology for running experiments

2.3.1 Data preprocessing

We've chosen to use the MNIST data set and the CIFAR-10 [TFF08] for our experiments. We normalized CIFAR-10 images since the original data set is RGB. We also applied data augmentation for CIFAR-10 to achieve better results on training the teacher.

2.3.2 Training

We created a pipeline to build and train the teacher model. We then pipe all the data set through the teacher to get the soft targets. Then, we used distillation to train the student model to mimic the same output of the teacher.

2.3.3 Distillation

Distillation [HVD] showed promising results on the MNIST data set [LC10] for deep neural networks.

When we want to transfer the knowledge using distillation we're not interested in matching the weights of the model, since these are not really relevant when we have models with

different architectures, but rather we are interested in the mapping from the input to the output learned by the models.

Simply put, we can see every model as a mapping $f(x) = y$ where x and y are high-dimensional vectors. So if we have a big model or an ensemble of models, we can transfer that knowledge by trying to copy the distribution of f. Instead of training the small model on hard targets (that is, one hot vector), we are training them on soft targets - the output of the bigger model.

Distillation showed very good results when a higher temperature T was used when transforming logits z_i computed for each class into probability q_i through softmax using the following formula:

$$q_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

Clearly, if $T = 1$ the formula is exactly the softmax layer. The higher the value of T, the softer the probability distribution will be. Good results were shown using higher values of T.

The *cost function* is defined as $Cost = \sum_{i=1}^{n} (y_i - q_i)^2$, where n denotes the number of training examples, q is the vector containing the soft targets of the teacher model and y contains the output from the student model, computed using the same formula as above.

The first main *difficulty* in the distillation process is to decide when the network has converged (in most cases it's better to keep training). Secondly, it is hard to define a good temperature, a good approach would be to learn, by supervision, a good value for hyperparameter T.

2.3.4 Evaluation

For evaluating the effectiveness of the SimpLeNet model, we are measuring both the performance of the image classification task (Section 2.3.4), as well as the compression rate obtained through reducing the learning model's size (Section 2.3.4).

Accuracy, precision, recall and F-measure

The image classification tasks considered in our experiments are multiclass classification problems. Let us denote by $C_1, C_2, \ldots, C_m$ the number of considered classes. Four evaluation measures which are usually computed for estimating the performance of supervised classifiers will be further used in the experimental evaluation: *accuracy*, *precision*, *recall* and *F-measure*.

The generalized confusion matrix for the supervised image classification problem is denoted by $A = (a_{ij})_{\substack{i=\overline{1,m} \\ j=\overline{1,m}}}$ [PCB06], where a_{ij} is computed as the number of samples which are actually belonging to class C_j and were predicted in class C_i.

The *precision* for class C_i ($\forall 1 \leq i \leq m$), denoted by $Prec_i$, is defined as the proportion of instances correctly predicted as C_i out of all instances which were predicted as belonging to C_i: $Prec_i = \frac{a_{ii}}{\sum_{j=1}^{m} a_{ij}}$. The overall *precision* (*Prec*) for the classifier is computed as the average of the precision values for all m classes, i.e. $Prec = \frac{\sum_{i=1}^{m} Prec_i}{m}$. The *recall* for the class C_i ($\forall 1 \leq i \leq m$), denoted by $Recall_j$, is computed as the fraction of instances which were correctly predicted as C_i out of all instances which are actually belonging to class C_i: $Recall_i = \frac{a_{ii}}{\sum_{j=1}^{m} a_{ji}}$. The overall *recall* value for the classifier is computed as the average of the recall values for all m classes, i.e. $Recall = \frac{\sum_{j=1}^{m} Recall_j}{m}$.

Based on the overall *precision* and *recall* values, the $F\text{-}measure$ for the partition is computed as the harmonic mean between the *precision* and *recall* values, i.e. $F\text{-}measure = \frac{2 \cdot Prec \cdot Recall}{Prec + Recall}$. The overall *accuracy* for the classifier, denoted by Acc, is defined as the proportion of instances correctly classified: $Acc = \frac{\sum_{i=1}^{m} a_{ii}}{\sum_{i=1}^{m} \sum_{j=1}^{m} a_{ij}}$.

All the previously described evaluation measures values (*accuracy*, *precision*, *recall* and *F-measure*) range from 0 to 1 and should be maximized in order to obtain better classifiers.

Compression rate

When transferring the knowledge stored by a large model (LM) into a smaller model (SimpLeNet in our case), besides the accuracy of the model we are interested in measuring the reduction of the model's size. For expressing the size of the learning model we are using the number of trainable weights from the network, more exactly the weights of the convolutional and fully connected layers. Note that max pooling layer does not contain trainable weights since it simply takes the output of the previous layer and it applies a function on it.

The *compression rate* obtained through reducing LM to SimpLeNet is measured as $CompRate = \frac{D-d}{D}$, where D is the the number of trainable weights from the LM model, while d is the number of trainable weights from the SimpLeNet model. $CompRate$ ranges from 0 to 1, with larger values indicating a better compression rate.

2.4 Experimental evaluation

The experiments performed in this section are conducted in order to answer the research questions RQ1 and RQ2 stated at the beginning of the chapter. Section 2.4.2 describes the data sets used in our evaluation, then we continue with the experimental setup (Section 2.4.3) and the obtained experimental results (Section 2.3).

2.4.1 TensorFlow Lite

TensorFlow lite is TensorFlow's library for mobile and embedded devices. It provides on-device machine learning inference a small binary size and low latency. It is based on a new file model format, FlatBuffers [1] - an efficient cross platform serialization library. Originally created at Google, for game development and other performance-critical applications, it is based on protocol buffers [2], the primary difference being that it does not need a parsing step to a secondary representation before accessing the data. Moreover, the code footprint of FlatBuffers is an order of magnitude smaller than protocol buffers.

TensorFlow Lite also ships with a mobile-ready interpreter, especially designed to keep apps lean and fast. It also provides an interface to leverage hardware acceleration, if available on the devices. Although at the time of writing, TensorFlow Lite is only available as a developer preview and includes only a limited set of features, we think we will see a growth overtime as the interest to create more and more machine learning models for mobile devices is increasing.

2.4.2 Data sets

In this experiment, we used the MNIST [LC10] data set, consisting of 60000 training examples and 10000 test examples.

2.4.3 Experimental setup and description

We used TensorFlow [AAB+16] and Keras [C+15] to conduct our experiments. That is, to build, train and measure the accuracy for each and every model.

Table 2.1 illustrate the architecture used in our experiments for the *large learning model* (LM) or *teacher*. Various architectures were used for the SimpLeNet model (denoted in the following by v1-v5) by changing the number of convolutional filters, the filter size and swapping the max pooling layer with another one of increased stride. Table 2.2 describes the architectures used for the SimpLeNet models versions v1 and v5. SimpLeNet v2 has a convolutional layer of 5x5 with 3 filters, followed by a 2x2 max pooling layer and a fully connected layer. SimpLeNet v3 has only 2 filters on the convolutional layer. SimpLeNet v4 replaces v2's max pooling layer with a 2x2 convolutional layer with increased stride.

[1] https://google.github.io/flatbuffers/

[2] https://developers.google.com/protocol-buffers/

Type	Shape	Description
Input	28x28x1	Input layer, 28x28 gray scale image
Conv	28x28x32	32 5x5x1 convolution layers with a ReLU activation function
MaxPool	14x14x32	2x2 max pooling layer
Conv	14x14x64	64 14x14x32 convolution layers with a ReLU activation layer
MaxPool	7x7x64	2x2 max pooling layer
Dense	1024	Fully connected layer
Dropout	1024	
Dense	10	Fully connected layer - produces the logits
Softmax	10	Produces the probability distribution of the classes

Table 2.1: Teacher model.

Model	Type	Shape	Description
SimpLeNet-v1	Input	28x28x1	Input layer, 28x28 gray scale image
	Conv	28x28x3	3 5x5x1 convolution layers with a ReLU activation function
	MaxPool	14x14x3	2x2 max pooling layer
	Conv	14x14x6	6 14x14x3 convolution layers with a ReLU activation layer
	MaxPool	7x7x6	2x2 max pooling layer
	Dense	10	Fully connected layer - produces the logits
	Softmax	10	Produces the probability distribution of the classes
SimpLeNet-v5	Input	28x28x1	Input layer, 28x28 gray scale image
	Conv	28x28x2	2 5x5x1 convolution layers with a ReLU activation function
	Conv-MaxPool	14x14x2	2 convolution layers with a 2x2x2 stride
	Dense	10	Fully connected layer

Table 2.2: SimpLeNet models (v1 and v5).

We performed experiments with different integer values of T ranging from 1 to 20.

2.4.4 Results

For assessing the performance of the proposed SimpLeNet models, multiple experiments are performed as described in Section 2.4.3. For each experiment, ten runs were used. The obtained experimental results are depicted in Table 2.3, where the average values for the evaluation measures introduced in Section 2.3.4 are computed for both the LM and the SimpLeNet models. For the SimpLeNet models, the second column depicts the value for the temperature hyperparameter T which provided the best results, while the last column present the compression rate $CompRate$ achieved through the SimpLeNet model. For all values, a 95% confidence interval [BCD01] is used. The best performance achieved by the SimpLeNet model, both in terms of $F\text{-}measure$ and *compression rate* is highlighted.

Figure 2.2 illustrate, for each SimpLeNet architecture, the variation for the accuracy

Model	T	*Acc*	*Prec*	*Recall*	*F-measure*	**#Trainable weights**	*CompRate*
LM	-	99.281 ± 0.0427	99.279 ± 0.0432	99.271 ± 0.0432	99.275 ± 0.0431	3274634	-
SimpLeNet-v1	1	98.046 ± 0.0633	98.052 ± 0.0610	98.035 ± 0.0642	98.044 ± 0.0624	3484	99.89%
SimpLeNet-v2	1	96.497 ± 0.5839	96.491 ± 0.5984	96.473 ± 0.5968	96.482 ± 0.5976	5968	99.82%
SimpLeNet-v3	1	95.931 ± 0.4863	95.931 ± 0.4913	95.898 ± 0.4968	95.915 ± 0.4940	3982	99.88%
SimpLeNet-v4	10	96.56 ± 0.3041	96.569 ± 0.2979	96.540 ± 0.3091	96.55 ± 0.3034	6007	99.82%
SimpLeNet-v5	3	95.397 ± 0.7609	95.380 ± 0.7740	95.360 ± 0.7799	95.370 ± 0.7769	4000	99.88%

Table 2.3: Experimental results.

during all ten runs. For SimpLeNet-v1 and SimpLeNet-v5, Table 2.4 presents the values for the evaluation measures detailed for all runs. Besides SimpLeNet-v1 which has the best performance, SimpLeNet-v5 was also selected since we saw better results on a higher value of T. The last four lines from Table 2.4 depict the *minimum*, *maximum*, *standard deviation* and *average* of the values.

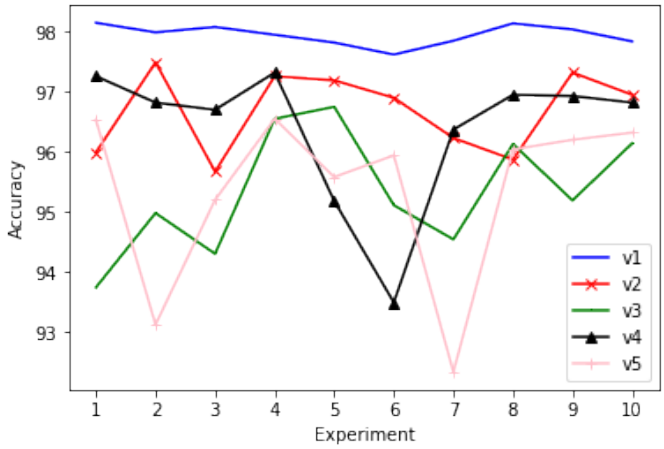

Figure 2.2: Accuracy variation during all ten runs, for each SimpLeNet architecture.

Table 2.3 and Figure 2.2 highlight that SimpLeNet-v1 obtains the best performance in terms of accuracy (**98.046**%) and compression rate (**99.89**%). We also note that the performance achieved by SimpLeNet-v1 is very close to that of LM (with only 1.271% smaller) and it significantly reduces the size of LM. The small standard deviation value provided by the SimpLeNet-v1 model highlight its effectiveness.

SimpLeNet-v4 is the same as SimpLeNet-v2, except that we changed the max pooling layer with a convolutional layer with increased stride. As shown in Table 2.3, this introduced a slight improvement on the accuracy, for the trade off of 39 more trainable parameters. To answer the question RQ1 regarding the max pooling layer, we think SimpLeNet-v4 was able to pick up a better pooling layer and hence we can achieve better results by simply using convolutional and densely connected layers. The only advantage of the max pooling layer is that it does not have any trainable weights.

Model	T	#	*Acc*	*Prec*	*Recall*	*F-measure*
SimpLeNet-v1	1	1	98.10	98.101	98.095	98.098
		2	98.18	98.188	98.171	98.179
		3	98.10	98.093	98.093	98.093
		4	98.15	98.147	98.139	98.143
		5	97.98	98.001	97.951	97.976
		6	97.83	97.838	97.829	97.834
		7	98.09	98.092	98.087	98.089
		8	97.98	97.982	97.969	97.975
		9	98.00	98.026	97.971	97.998
		10	98.05	98.055	98.043	98.049
		Min	97.83	97.838	97.829	97.834
		Max	98.18	98.188	98.171	98.179
		Stdev	0.1022	0.0985	0.1036	0.1006
		Avg	98.046 ± 0.0633	98.052 ± 00.0610	98.035 ± 0.0642	98.044 ± 0.0624
SimpLeNet-v5	3	1	95.55	95.552	95.530	95.541
		2	95.65	95.638	95.613	95.626
		3	96.6	96.621	96.583	96.602
		4	95.94	95.927	95.903	95.915
		5	95.55	95.544	95.532	95.538
		6	95.43	95.376	95.394	95.385
		7	92.22	92.161	92.100	92.131
		8	96.08	96.073	96.059	96.066
		9	94.72	94.673	94.665	94.669
		10	96.23	96.236	96.222	96.229
		Min	92.22	92.161	92.100	92.131
		Max	96.6	96.621	96.583	96.602
		Stdev	1.2277	1.2488	1.258	1.2536
		Avg	95.397 ± 0.7609	95.380 ± 0.7740	95.360 ± 0.7799	95.370 ± 0.7769

Table 2.4: Detailed results for SimpLeNet-v1 trained with T=1 and SimpLeNet-v5 trained with T=3. A CI of 95% was used for the average.

2.5 Discussion and comparison to related work

This section analyzes the experimental results described in Section 2.4 and conducts a comparison of our proposal to existent models with the aim of answering research question RQ3. The comparison will be conducted considering the performance of the models as well as their size (in number of trainable weights).

Experimental results were provided in Section 2.3 for various SimpLeNet architectures and different values for the hyperparameter T. Figure 2.3 graphically depicts the variation of $F\text{-}measure$ and $CompRate$ values with respect to different SimpLeNet architectures. A comparison of SimpLeNet architectures in terms of accuracy and trainable weights is shown in Figure 2.4.

From Figures 2.3 and 2.4 one observes, as already shown by the experimental results presented in Table 2.3, that SimpLeNet-v1 is the most effective architecture, preserving a high enough accuracy and also providing a very high compression rate.

For analyzing how the temperature hyperparameter T used in the distillation step affects the performance of the SimpLeNet models, we performed experiments for different values

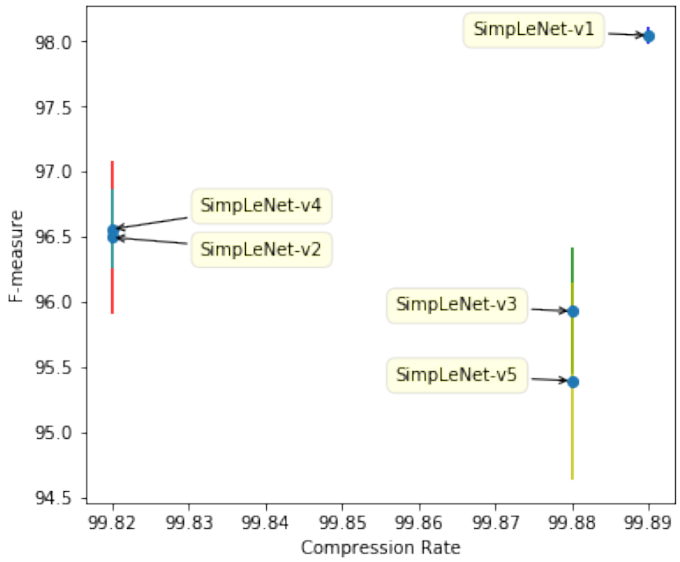

Figure 2.3: $F\text{-}measure$ vs $CompRate$ for all SimpLeNet architectures. Standard error bars are added for the results.

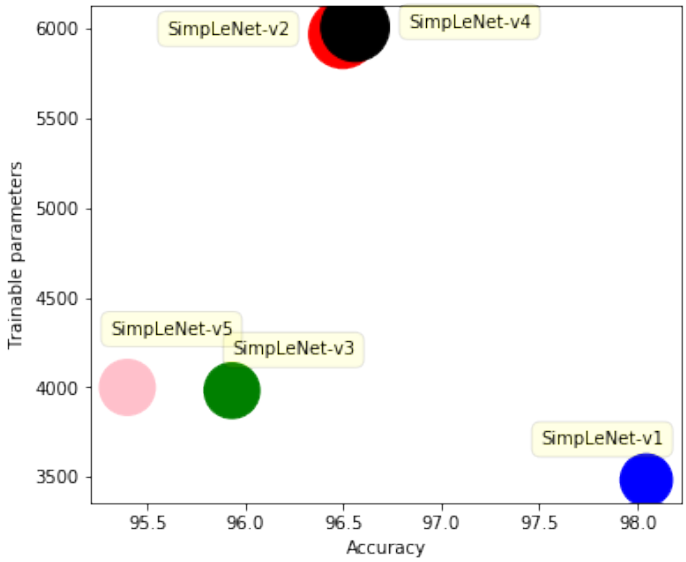

Figure 2.4: SimpLeNet architectures comparison in terms of accuracy and trainable weights

for T. For SimpLeNet-v1, Table 2.5 illustrates the obtained results and Figure 2.5 depicts the variation of the evaluation measures with respect to the value of T. From Table 2.5 and Figure 2.5 we observe that the best performance for SimpLeNet v1, v2 and v3 is achieved for $T = 1$, while SimpLeNet v4 and v5 have the best performance for $T = 10$ and $T = 3$, respectively.

To further demonstrate the effectiveness of distillation, we trained the SimpLeNet models using the cross-entropy cost function on hard targets only (without distillation). For each SimpLeNet architecture, Table 2.6 shows in the last column the differences between the accuracy achieved through classic training and distillation. Excepting SimpLeNet-v2, we observe a slight improvement obtained using the distillation process.

T	*Acc*	*Prec*	*Recall*	*F-measure*
1	98.046 ± 0.0633	98.0528 ± 0.0610	98.035 ± 0.0642	98.044 ± 0.0624
3	97.817 ± 0.1436	97.8237 ± 0.1407	97.801 ± 0.1477	97.812 ± 0.1441
6	97.975 ± 0.1201	97.9791 ± 0.1225	97.965 ± 0.1171	97.972 ± 0.1197
7	97.959 ± 0.1086	97.9630 ± 0.1075	97.945 ± 0.1098	97.954 ± 0.1085
8	97.957 ± 0.1092	97.9624 ± 0.1087	97.944 ± 0.1081	97.953 ± 0.1083
9	97.824 ± 0.1367	97.8308 ± 0.1326	97.812 ± 0.1371	97.821 ± 0.1346
10	97.917 ± 0.1010	97.9238 ± 0.0942	97.903 ± 0.1062	97.913 ± 0.1000
11	97.954 ± 0.0908	97.9558 ± 0.0946	97.942 ± 0.0884	97.949 ± 0.0914
12	97.869 ± 0.0881	97.8727 ± 0.0879	97.863 ± 0.0889	97.867 ± 0.0883
15	97.674 ± 0.1358	97.6806 ± 0.1323	97.662 ± 0.1365	97.671 ± 0.1341
20	97.703 ± 0.0955	97.7108 ± 0.0881	97.695 ± 0.0963	97.703 ± 0.0917

Table 2.5: Results obtained for SimpLeNet-v1 and different values for the temperature hyperparameter.

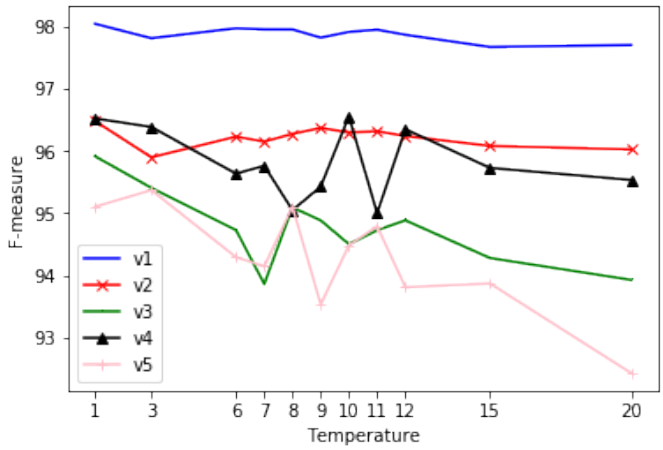

Figure 2.5: Variation of $F\text{-}measure$ for different values for T for different SimpLeNet architectures.

Model	**Classic training**	**Distillation**	**Difference**
SimpLeNet-v1	97.948	98.046	0.098
SimpLeNet-v2	96.687	96.497	-0.19
SimpLeNet-v3	95.343	95.931	0.588
SimpLeNet-v4	96.384	96.56	0.176
SimpLeNet-v5	95.381	95.397	0.016

Table 2.6: Improvements on using distillation

The SimpLeNet architecture proposed in this chapter is new, as we have not found similar existing approaches. We proposed simple architectures that are capable of compressing the knowledge gathered by way bigger models, with low impact in terms of performance. We demonstrated that distillation can successfully be used as a training method for other

architectures such as MobileNets [HZC+17] or SqueezeNet [IHM+16].

2.6 Conclusions and future work

The study conducted in this chapter was aimed to demonstrate the applicability of CNNs for low resource devices and to study their performance in real life scenarios. We introduced a CNN model, called SimpLeNet, which can run on low-resource devices such as smartphones, smartwatches, tablets or TVs. For proving the effectiveness of SimpLeNet models, experiments were conducted on various data sets for image classification, emphasizing that the accuracy of the classification process can be preserved even if the size of the SimpLeNet model is significantly reduced.

Future work will be carried out in order to extend the experimental evaluation of SimpLeNet on other image data sets, to further test its performance. We also aim to investigate methods for learning a good value for the temperature hyperparameter T used in the distillation process.

Chapter 3

Software Application

In this chapter, we are introducing the software application developed in order to demonstrate the use case of our SimpLeNet in mobile devices applications. The chapter is structured into two sections: section 3.1 offers software development related details, while section 3.2 summarizes future possible optimization and enhancements.

3.1 Software Development

3.1.1 Problem definition and specification

Our goal was to find new mobile-ready convolutional neural networks architectures. So we wanted a convolutional neural network that was able to be given as input an image and classify that image in some classes that we define before. We worked mainly with MNIST but also performed experiments on CIFAR-10 and CIFAR-100. For ImageNet, we looked into how well MobileNets [HZC$^+$17] performs in practice.

After finding those good deep learning models, we had to export that model to a mobile application. In order to test how well our model performs we measure the elapsed time for making an inference.

The mobile application should be able to make real-time inference on the images captured from the camera on the back of the phone.

3.1.2 Analysis and Design

In order to test different architectures, we developed a fully automated pipeline for running our experiments. Briefly described in Figure 3.3, the architecture was perfect for our use case: to easily plug and test different architectures.

Although the application is very minimalist in terms of UI/UX and design, the mobile application was carefully designed around testing different models as well. The architecture is depicted in figure 3.4.

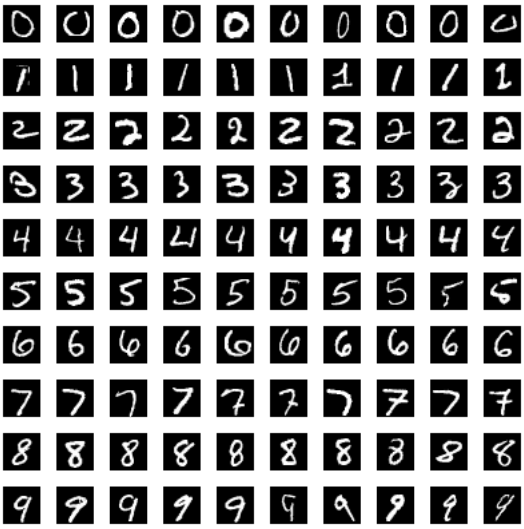

Figure 3.1: Samples for each category on MNIST dataset [LC10]

Figure 3.2: Samples for each category on CIFAR-10 dataset [KH09]

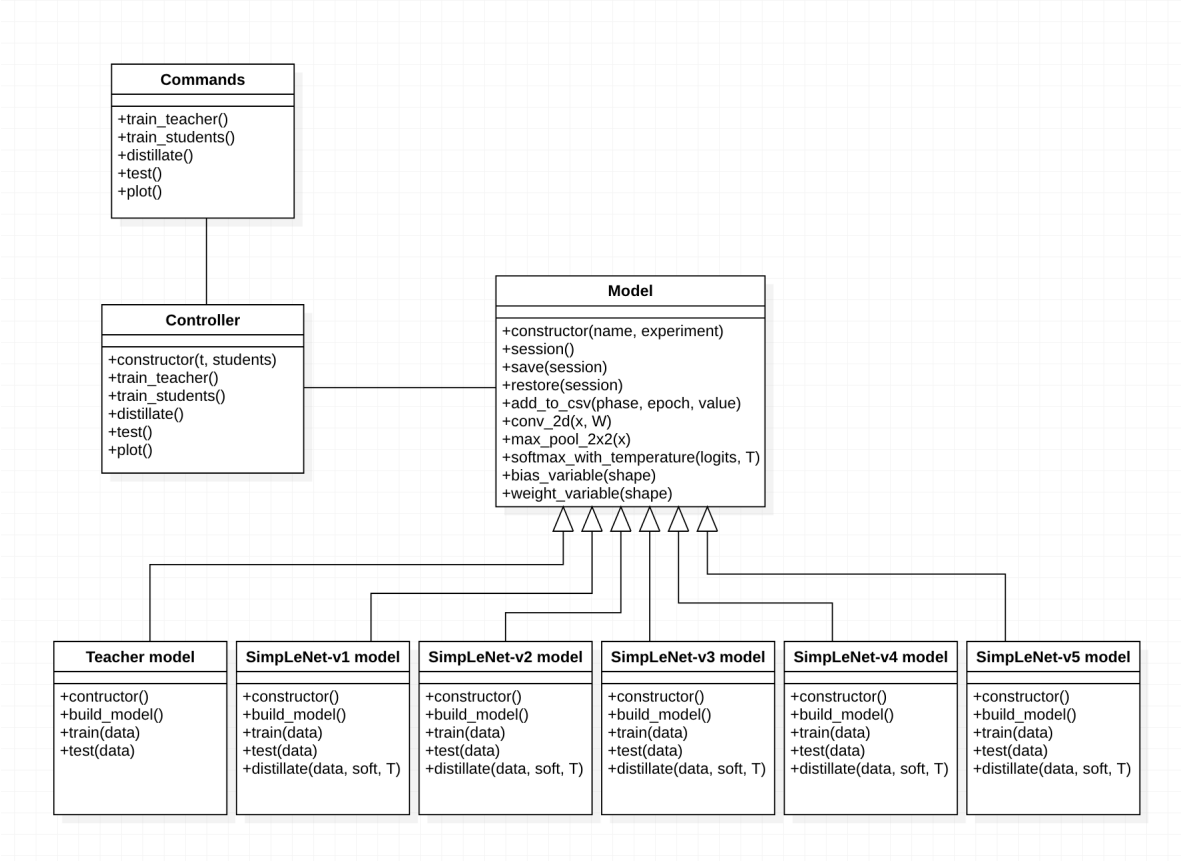

Figure 3.3: Architecture for running experiments with different models

Let's further demonstrate how easy it is with our architecture to experiment with a new model. First, we would create a new model class that inherits our base **Model** class. We then add it to the list of known models in our controller. We use the command line tool to train the teacher and distillate the knowledge to our brand new model. We carefully observe how the model learns and note its performance in terms of accuracy and compression rate. We adjust the model and repeat this feedback loop until our model reaches a mature state.

As soon after having a model that we are happy with, we use TensorFlow Lite to convert it to mobile format. Depending on the dataset the model was trained with, we create a configuration file containing the mapping from our indices to the actual classes. The two files are needed as the next class that we add in the application is going to make use of them. Concretely, we add a new class that inherits the ImageClassifier class and implement the abstract methods. We then change the ImageClassifier used by the main activity to an instance of our brand new classifier, and we can deploy to an actual device to check its performance.

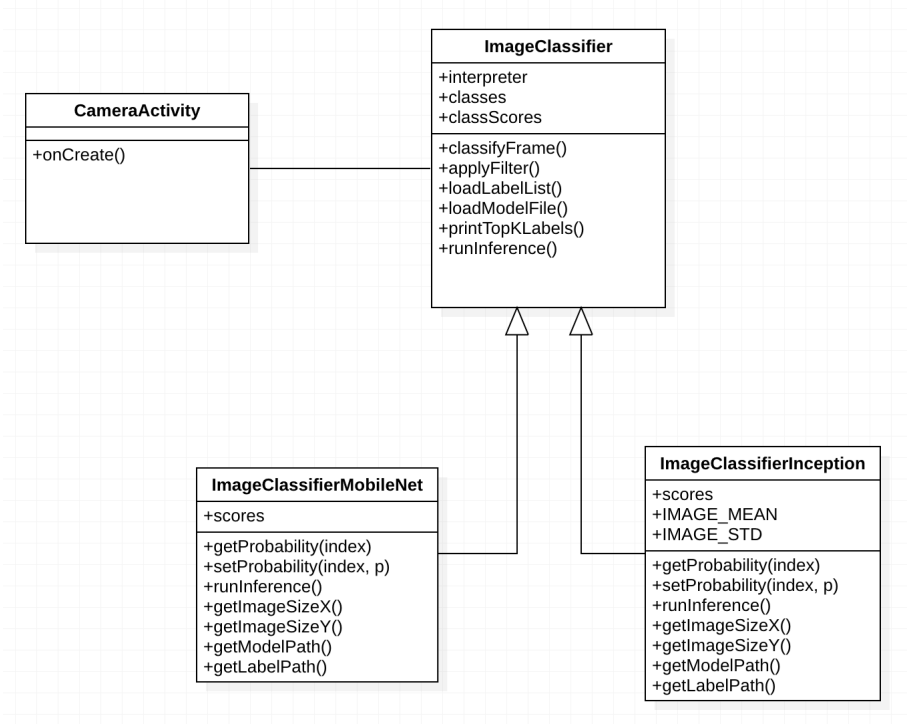

Figure 3.4: Mobile application architecture

3.1.3 Implementation

Model development

We used TensorFlow [AAB+16] and Keras [C+15] in order to build our deep learning models, and TensorFlow Lite to export our model to the mobile application. After that, we used TensorFlow Lite's Java interpreter in order to integrate the model with an Android application that looks for the camera preview and makes on-device real time inferences.

TensorFlow provides both a low level and high level API that can be used to build complex deep learning models.

We wanted to quickly be able to train and test multiple mobile-ready CNN architectures, so we added a command line utility to our pipeline. The main commands implemented were training the Teacher model, teaching the Student models, piloting useful data and saving experiment results that we can check in order to get a feedback on where and what to improve. Another very useful tool was a Jupyter notebook providing a TensorFlow-ready Python environment perfect for prototyping and getting insights early on.

Mobile development

We have chosen Android to build a proof-of-concept mobile application capable of running on-device machine learning. There are a lot of technologies available today to develop an Android application: React Native, Qt, Kotlin, Java, just to name few. We have chosen the latter mainly because of the easy integration with TensorFlow Lite interpreter.

The application has only one main activity containing a live back camera preview, the

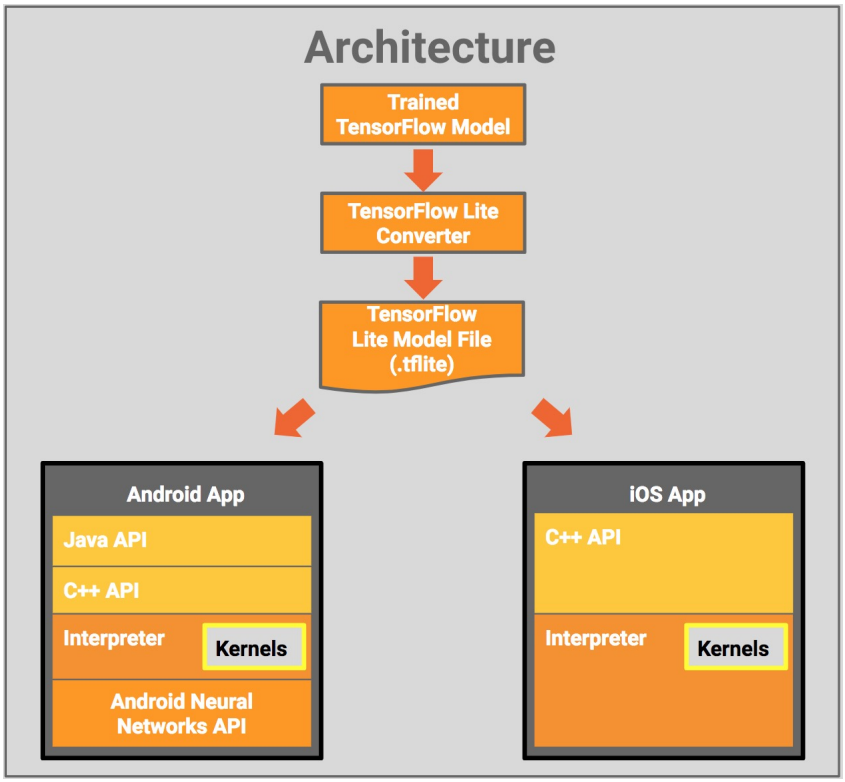

Figure 3.5: TensorFlow Lite architecture [tfl18]

class labels associated with the current image frame, and the time elapsed for our model to infer the class scores.

Since the camera previews somewhere between 30 and 60 frames per second (depending on the device), we saw a lot of noise between the class scores on each frame. Instead of making inferences once in a while, we used an algorithm that smoothens the results across the frames. We came across to the Low-Pass Filter algorithm with 3 stages and filter factor equal to 0.4. The whole classification and smoothing algorithm is shown in the Listing 3

3.1.4 User's Manual

Command line utility

We have built a command line tool for the pipeline of running model experiments. The figure 3.6 shows all the available commands along with the arguments for each. The following commands are currently supported:

- **Train Teacher** $ python3 exp1.py −t

  This command will train the teacher model and save a checkpoint for it when it finishes so that the next time we want it, we don't need to train it again.

- **Train Students** $ python3 exp1.py −s

```
cosminr@MLPrime:~/bsc/research/experiment1$ python3 exp1.py --help
usage: exp1.py [-h] [-t] [-s] [-d] [-p] [-te]

experiment1 of the research project

optional arguments:
  -h, --help           show this help message and exit
  -t, --trainTeacher   train teacher
  -s, --trainStudents  train students
  -d, --distillate     distillate
  -p, --plot           plots the data
  -te, --test          prints the accuracy and confusion matrix for the models
```

Figure 3.6: Command line utility for the experiments pipeline

```
cosminr@MLPrime:~/bsc/research/experiment1$ python3 exp1.py -t
Teacher::__init__
Student4::__init__
Student5::__init__
Student::__init__
Student2::__init__
Student3::__init__
> Loading MNIST data...
Extracting MNIST_data/train-images-idx3-ubyte.gz
Extracting MNIST_data/train-labels-idx1-ubyte.gz
Extracting MNIST_data/t10k-images-idx3-ubyte.gz
Extracting MNIST_data/t10k-labels-idx1-ubyte.gz
trainingTeacher
Teacher::train
2018-06-23 14:29:59.892249: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your CP
U supports instructions that this TensorFlow binary was not compiled to use: SSE4.1 SSE4
.2 AVX AVX2 FMA
2018-06-23 14:30:00.013750: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1030] Fou
nd device 0 with properties:
name: GeForce GTX TITAN X major: 5 minor: 2 memoryClockRate(GHz): 1.076
pciBusID: 0000:06:00.0
totalMemory: 11.92GiB freeMemory: 11.23GiB
2018-06-23 14:30:00.013778: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1120] Cre
ating TensorFlow device (/device:GPU:0) -> (device: 0, name: GeForce GTX TITAN X, pci bu
s id: 0000:06:00.0, compute capability: 5.2)
Starting training epoch 0
Epoch : 1, Loss : 0.065698, Accuracy: 0.984000, Test accuracy: 0.968900
Starting training epoch 1
Epoch : 2, Loss : 0.081868, Accuracy: 0.972000, Test accuracy: 0.977900
Starting training epoch 2
Epoch : 3, Loss : 0.065433, Accuracy: 0.980000, Test accuracy: 0.982500
Starting training epoch 3
Epoch : 4, Loss : 0.055596, Accuracy: 0.984000, Test accuracy: 0.986500
Starting training epoch 4
Epoch : 5, Loss : 0.024911, Accuracy: 0.996000, Test accuracy: 0.987800
Starting training epoch 5
Epoch : 6, Loss : 0.030701, Accuracy: 0.988000, Test accuracy: 0.989100
Starting training epoch 6
Epoch : 7, Loss : 0.017179, Accuracy: 0.992000, Test accuracy: 0.989400
```

Figure 3.7: Training models from the command line utility

In [25]:
```
# train and evaluate
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv, 1), tf.argmax(y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for i in range(2000):
        batch = mnist.train.next_batch(50)
        if i % 100 == 0:
            train_accuracy = accuracy.eval(feed_dict = {x: batch[0], y_: batch[1], keep_prob: 1.0})
            print("step %d, training accuracy %g" % (i, train_accuracy))
        train_step.run(feed_dict={x:  batch[0], y_: batch[1], keep_prob: 0.5})
    batch = mnist.test.next_batch(10000)
    print("test accuracy %g" % accuracy.eval(feed_dict = {x: batch[0], y_: batch[1], keep_prob: 1.0}))
```

```
step 0, training accuracy 0.18
step 100, training accuracy 0.86
step 200, training accuracy 0.92
step 300, training accuracy 0.88
step 400, training accuracy 1
step 500, training accuracy 0.88
step 600, training accuracy 0.98
step 700, training accuracy 1
step 800, training accuracy 0.94
step 900, training accuracy 0.94
step 1000, training accuracy 0.96
step 1100, training accuracy 1
step 1200, training accuracy 1
step 1300, training accuracy 1
step 1400, training accuracy 0.98
step 1500, training accuracy 0.98
step 1600, training accuracy 1
step 1700, training accuracy 1
step 1800, training accuracy 1
step 1900, training accuracy 0.98
test accuracy 0.9769
```

Figure 3.8: Fast iterations with Jupyter notebook

This command line will train all the students that we have on created on the controller class. It will also log useful debugging information regarding the training and at the end will save a TensorFlow checkpoint file for each of them.

- **Distillate** $ python3 exp1.py −d

  This command will load the teacher from the checkpoint (or train again) and then will train the students using distillation. The controller contains some predefined values for the temperature parameter T.

- **Plot**

  If we append the −p argument to the command, it will generate useful plots such as the learning curve.

- **Test**

  Finally, passing the −te argument, the program will print the confusion matrix, accuracy, f-measure, precision and recall for all our models. The test and plot flags were introduced because training so many models and using 10 values for the temperature hyperparameter take a very long time so we wanted the pipeline to run overnight and get all the results properly formatted at the end.

Android application

After the application has been installed, the user must simply open the application. The first time the user opens the app, a pop up will appear that requests for user permission to use

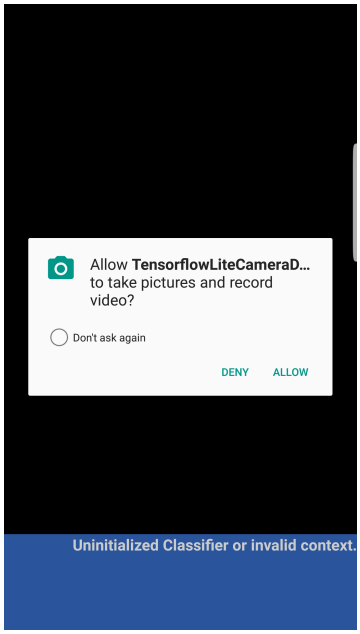

Figure 3.9: Camera permission request

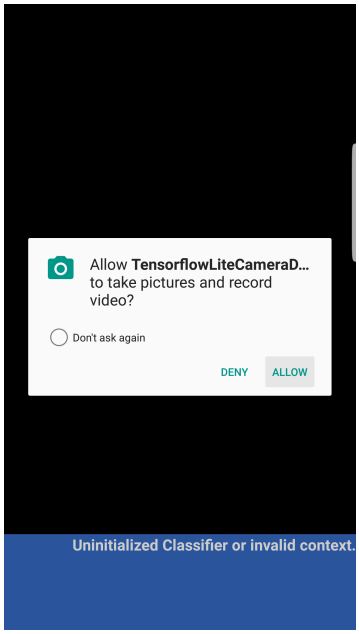

Figure 3.10: Permission granted

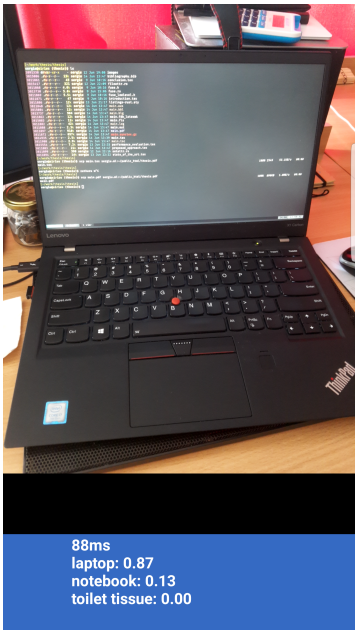

Figure 3.11: App starts making real time inferences

the camera. As soon as the user grants the permission, it starts making real time inferences on the preview images from the camera. The user can see the list of Top K classes ordered by the confidence score from our model. The parameter K can be changed, but it must satisfy the condition that $K <= N$ where N is the total number of classes.

```
class Student(Model):
  def __init__(self, name):
    print("Student::__init__")
    super().__init__(name)

    # placeholders for input and output variables in the
    # dataset (x = features, y = labels)

    # x = 28 x 28 pixels from images
    # y = one hot vector where 1 denotes the correct label
    self.x = tf.placeholder(tf.float32, shape=[None, 784])
    self.y_ = tf.placeholder(tf.float32, shape = [None, 10])

    # reshape the input to a 4d tensor
    # (-1 since we don't know how many images we have)
    # the second and third dimension is the size of the image
    # and the last dimension represents the number of color channels
    self.x_image = tf.reshape(self.x, [-1, 28, 28, 1])

    # The first convolutional layer
    self.W_conv1 = Model.weight_variable([5, 5, 1, 3])
    self.b_conv1 = Model.bias_variable([3])

    # convolve the image with a relu activation function
    self.h_conv1 = tf.nn.relu(
        Model.conv2d(self.x_image, self.W_conv1) + self.b_conv1)
    # add max_pooling layer
    self.h_pool1 = Model.max_pool_2x2(self.h_conv1)

    # create the second convolutional layer
    self.W_conv2 = Model.weight_variable([5, 5, 3, 6])
    self.b_conv2 = Model.bias_variable([6])

    # stack it on top of the first one
    self.h_conv2 = tf.nn.relu(
        Model.conv2d(self.h_pool1, self.W_conv2) + self.b_conv2)
    # add max_pooling 2x2 layer
    self.h_pool2 = Model.max_pool_2x2(self.h_conv2)

    # create the fully connected layer
    self.W_fc1 = Model.weight_variable([7 * 7 * 6, 10])
    self.b_fc1 = Model.bias_variable([10])

    # reshape the pooling layer to be flat
    self.h_pool2_flat = tf.reshape(self.h_pool2, [-1, 7 * 7 * 6])
    self.y_conv = tf.matmul(self.h_pool2_flat, self.W_fc1) + self.b_fc1
```

Listing 1: SimpLeNet-v1 architecture

```
1   /** Takes photos and classify them periodically. */
2   private Runnable periodicClassify =
3       new Runnable() {
4         @Override
5         public void run() {
6           synchronized (lock) {
7             if (runClassifier) {
8               classifyFrame();
9             }
10          }
11          backgroundHandler.post(periodicClassify);
12        }
13      };
14
15  /** Classifies a frame from the preview stream. */
16  private void classifyFrame() {
17    if (classifier == null
18        || getActivity() == null
19        || cameraDevice == null) {
20      showToast("Uninitialized Classifier or invalid context.");
21      return;
22    }
23    Bitmap bitmap = textureView.getBitmap(
24        classifier.getImageSizeX(),
25        classifier.getImageSizeY());
26    String textToShow = classifier.classifyFrame(bitmap);
27    bitmap.recycle();
28    showToast(textToShow);
29  }
```

Listing 2: Background thread performing the classification

```
/** Classifies a frame from the preview stream. */
String classifyFrame(Bitmap bitmap) {
  if (tflite == null) {
    Log.e(TAG, "Image classifier has not been initialized; Skipped.");
    return "Uninitialized Classifier.";
  }
  convertBitmapToByteBuffer(bitmap);

  long startTime = SystemClock.uptimeMillis();
  runInference();
  long endTime = SystemClock.uptimeMillis();
  Log.d(TAG, "Timecost to run model inference: " +
      Long.toString(endTime - startTime));

  // Smooth the results across frames.
  applyFilter();

  // Print the results.
  String textToShow = printTopKLabels();
  textToShow = Long.toString(endTime - startTime) + "ms" + textToShow;
  return textToShow;
}

void applyFilter() {
  int numLabels = getNumLabels();

  // Low pass filter `labelProbArray` into the first stage of the filter.
  for (int j = 0; j < numLabels; ++j) {
    filterLabelProbArray[0][j] +=
        FILTER_FACTOR * (getProbability(j) - filterLabelProbArray[0][j]);
  }
  // Low pass filter each stage into the next.
  for (int i = 1; i < FILTER_STAGES; ++i) {
    for (int j = 0; j < numLabels; ++j) {
      filterLabelProbArray[i][j] +=
          FILTER_FACTOR * (filterLabelProbArray[i - 1][j] -
              filterLabelProbArray[i][j]);
    }
  }

  // Copy the last stage filter output back to `labelProbArray`.
  for (int j = 0; j < numLabels; ++j) {
    setProbability(j, filterLabelProbArray[FILTER_STAGES - 1][j]);
  }
}
```

Listing 3: Classifying frames and Low-Pass Filter algorithm

3.2 Future improvements

The experimental evaluation can be further extended to other data sets (for example ImageNet). The automated pipeline can be further improved so that it can take a model and try to find the best compressed architecture in order to make it suitable for low-resource environments. One could use an evolutionary algorithm for example to find the best such architectures.

An iOS application is also possible to implement that use TensorFlow Lite C++ API. Other resources can also be tracked (and shown) on the mobile app interface such as the memory, CPU and GPU consumption, although these can be seen using development tools such as profiling.

A settings activity can be added for changing the number of labels displayed on the user interface, and the constants for the Low-Pass filter algorithms. The settings can also include the resolution of the images that are sent to the model for classification.

Conclusions

In this thesis, we showed how cumbersome models like Convolutional Neural Networks can be optimized to be able to make inference or event training on low resource devices such as smartphones, smartwatches, Internet of Things devices or smart TVs. We introduced a CNN model, called SimpLeNet, which can run on low-resource devices. Experiments were conducted on various data sets for image classification in order to empirically prove the effectiveness of SimpLeNet models and that the accuracy of the model can be preserved even if their size is significantly reduced.

We used TensorFlow to build, train and test our Convolutional Neural Networks, as well as TensorFlow Lite to integrate these models with an Android application.

The experimental evaluation of SimpLeNet will be further extended on other image data sets. Improvements of the proposed model are also envisioned, such as methods for learning a good value for the temperature hyperparameter used in the distillation process.

We believe the next wave of technology, Software 2.0 [Kar17], will intensively use artificial intelligence algorithms and we expect more and more innovation in this field in the upcoming years.

Bibliography

[AAB+16] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[BCD01] Lawrence D. Brown, T. Tony Cai, and Anirban Dasgupta. Interval estimation for a binomial proportion. *Statistical Science*, 16:101–133, 2001.

[BCNM06] Cristian Bucilu, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006.

[C+15] François Chollet et al. Keras. `https://keras.io`, 2015.

[cs217] Cs231n convolutional neural networks for visual recognition, 2017. [Online; accessed 14-May-2018].

[HKW11] Geoffrey E Hinton, Alex Krizhevsky, and Sida D Wang. Transforming auto-encoders. In *International Conference on Artificial Neural Networks*, pages 44–51. Springer, 2011.

[HVD] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv preprint arXiv:1503.02531*, pages 1–9.

[HZC+17] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861:1–9, 2017.

[IHM+16] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, pages 1–13, 2016.

[Kar17] Andrej Karpathy. Software 2.0, 2017. [Online; accessed 19-June-2018].

[KH09] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[KSH12a] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[KSH12b] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[LC10] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. 2010.

[LD15] Shuying Liu and Weihong Deng. Very deep convolutional neural network based image classification using small training sample size. In *Pattern Recognition (ACPR), 2015 3rd IAPR Asian Conference on*, pages 730–734. IEEE, 2015.

[MCC18] Diana-Lucia Miholca, Gabriela Czibula, and Istvan Gergely Czibula. A novel approach for software defect prediction through hybridizing gradual relational association rules with artificial neural networks. *Information Sciences*, 441:152 – 170, 2018.

[Mit97] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[OMS17] Chris Olah, Alexander Mordvintsev, and Ludwig Schubert. Feature visualization. *Distill*, 2017. https://distill.pub/2017/feature-visualization.

[PCB06] Davide Picca, Benot Curdy, and Franois Bavaud. Non-linear correspondence analysis in text retrieval: a kernel view. In *Proc. of JADT 2006 : 8es Journees internationales dAnalyse statistique des Donnees Textuelles*, 2006.

[RC18] Cosmin-Ionu¸t Rusu and Gabriela Czibula. Optimizing convolutional neural networks for low-resource devices. In *Proceedings of IEEE 14th International Conference on Intelligent Computer Communication and Processing*, ICCP 2018, page under review. IEEE Computer Society, 2018.

[Ros57] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[SDBR14] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*, 2014.

[SFH17] Sara Sabour, Nicholas Frosst, and Geoffrey E Hinton. Dynamic routing between capsules. In *Advances in Neural Information Processing Systems*, pages 3857–3867, 2017.

[SHZ+18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *arXiv preprint arXiv:1801.04381*, 2018.

[SLJ+14] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, 2014.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[SZS+13] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[TFF08] Antonio Torralba, Rob Fergus, and William T Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 30(11):1958–1970, 2008.

[tfl18] Tensorflow lite documentation, 2018. [Online; accessed 23-June-2018].

[W+60] Bernard Widrow et al. *Adaptive" adaline" Neuron Using Chemical" memistors.".* 1960.