Building a CNN for Rock-Paper-Scissors Classification

University of Milan, MSc in Computer Science

Statistical Methods for Machine Learning

Omar Masri (53148A)

omar.masri@studenti.unimi.it

Contents

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

1 Introduction

The aim of this project is to design and implement a series of Convolutional Neural Networks (CNNs) capable of predicting hand gestures from to the "Rock, Paper, Scissors" game. This involves designing multiple neural network architectures of varying complexity, training them, and systematically evaluating their performance. In addition we'll also investigate the models ability to generalize to a completely new set of images.

For the sake of brevity we'll assume that the reader is already familiar with the main theoretical framework behind CNNs and image classification in general, and therefore this report will NOT cover those concepts in detail. As a general reference for these topics, i relied on [6], particularly Chapter 5.

2 Data Exploration and Preprocessing

2.1 Overview

The dataset that we're going to use is a collection of hand gesture images from the classic game "Rock, Paper, Scissors" [1]. The images are separated in three sub-folders one for each gesture, giving us an implicit labeling.

The dataset is comprised of 2188 images in PNG format, each with a dimension of 300×200 pixels. The distribution of gestures is relatively balanced, with 726 Rock images, 710 Paper images, and 752 Scissors images. All images feature a consistent green background with similar lighting conditions and have been white balanced.

2.2 Preprocessing

2.2.1 Image Processing

The images required no image preprocessing due their relative high quality. White balance correction has already been applied and the images demonstrate an adequate contrast and brightness, negating the need for histogram equalization or smoothing/filtering.

2.2.2 Data Preparation

One notable characteristic common across almost all images is that the hand gesture is oriented consistently, with the hand positioned from left to right across the image, moreover all hands have roughly the same scale, meaning that they cover a similar proportion of the image. This issue is somewhat problematic since we want our model to be robust enough to work on

completely new images of hands with varying positions, scales and orientations. We'll deal with this issue in our data preparation transformation by using data augmentation. A more subtle, and not completely solvable, issue is the very limited variability in the hand gestures, for example there are only 2 images out of 2188 showing the hand palm up and other variations such as scissors with the thumb extended are wholly not represented.

So the data preparation transformation works as follows. First, all images are resized to a 255×255 pixels image using bicubic interpolation to ensure a consistent input size. Data augmentation techniques are then applied such as random horizontal flipping, random rotation on a $90°$ degree range and random scaling between 85% and 115%. This should address the aforementioned uniformity issue by introducing some much needed variability in hand positioning and orientation. Subsequently we apply a color jittering transformation (brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1), which should simulate different lighting conditions and slight hue variations, the hue jittering was specifically added to hopefully make the model more robust to changes in background but as you can see from Fig. 2.1 this alone wasn't much able to change the background color. Following, pixel values are converted to tensors and scaled from the original $[0, 255]$ range to $[0, 1]$, then we normalized (standardized) using the dataset's mean and standard deviation per each channel, resulting in each channel having approximately zero mean and unit standard deviation. We got this idea from the ResNet architecture [4] [3], and after testing it, we found better accuracy and a small but noticeable improvement in training time compared to the original $[0, 1]$ range.

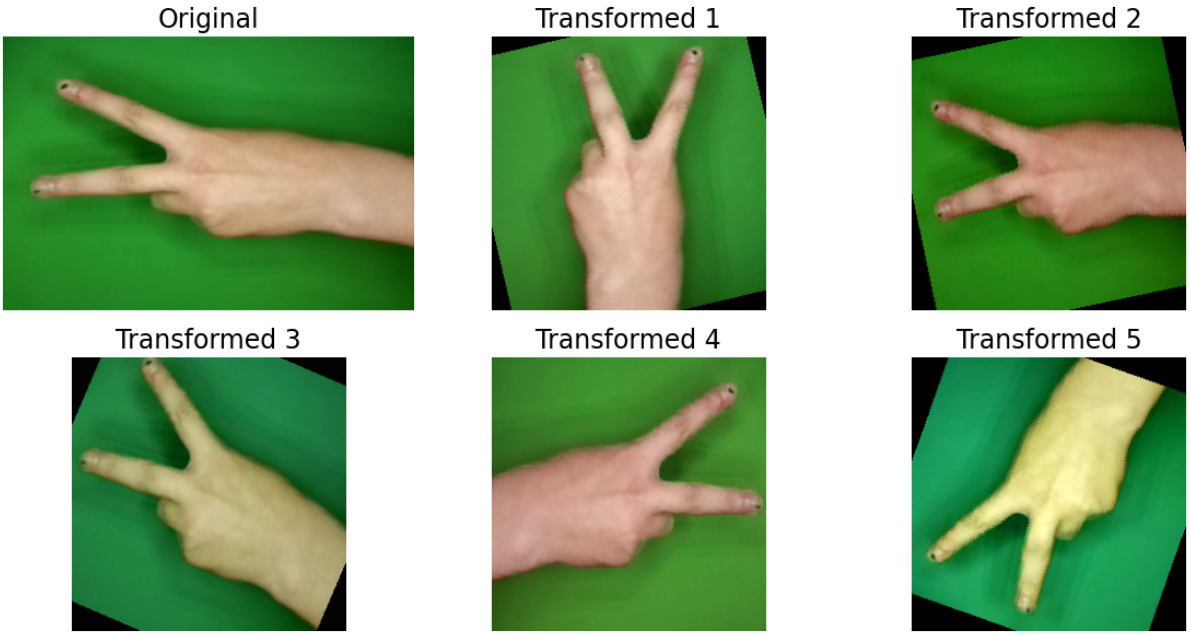


Figure 2.1: Examples of our data preparation transformation applied to a single image from the dataset. The top-left image is the original, while the others show random transformations.

3 Architectures and Hyperparameter Tuning

3.1 Architectures

We decided to design and implement four different CNN architectures of increasing complexity using the PyTorch library [5].

1. **First Architecture**: The input image (which can be seen as a 3D matrix with 3 channels) is passed through three convolutional layers with 3×3 kernels and with increasing output channels $(32 \to 64 \to 128)$, each followed by a ReLU activation and max pooling with kernel size 2 and stride 2 to progressively extract more abstract and higher level features. After this convolutional section the resulting intermediate features are flattened and passed through two fully connected layers, the first with 256 units and a ReLU activation, and the second producing the final classification outputs (logits). This is the simplest architecture we implemented (Fig. A.1).

2. **Second Architecture (+Batch normalization +Dropout)**: Builds upon the First Architecture by adding Batch Normalization after each convolutional layer, which stabilizes and accelerates the training. Spatial dropout, which randomly drops an ENTIRE feature map, with a rate of 0.1 is applied between convolutional layers to reduce overfitting, and a dropout, which randomly drops individual neurons, with rate 0.25 is applied in the fully connected layers for the same reason. These modifications should make the network slightly faster in training time and more robust to overfitting (Fig. A.2).

3. **Third Architecture (+Deeper convolutional and fully connected layers)**: Builds upon the Second Architecture by adding a fourth convolutional layer with 256 output channels. The fully connected part is also deepened, with three layers rather than two, the first with 1024 units, the second with 256 units and the last producing the final classification outputs, with each hidden layer followed by ReLU and dropout just as the Second Architecture (Fig. A.3).

4. **Fourth Architecture (VGG-Like)**: Is a deeper VGG-inspired ([6] page 300) architecture composed of five convolutional blocks. Each block contains two convolutional layers with 3×3 kernels, where each convolutional layer is followed by batch normalization and ReLU activation. The number of output channels progressively doubles across blocks $(32 \to 64 \to 128 \to 256 \to 512)$, after each block max pooling and spatial dropout (0.1) is applied just like in the Third Architecture, furthermore the fully connected layers are exactly the same as the ones found in the Third Architecture. This is the last and most architecturally complex design we implemented (Fig. A.4).

For each architecture, we decided to use the cross entropy loss as the loss function. cross entropy is particularly suitable for classification tasks in convolutional neural networks. Mathematically, the cross entropy loss for a single input is defined as

$$\ell(y, \hat{y}) = -\sum_{i=1}^{N} y_i \log(\widehat{y_i})$$

where

- N is the number of classes (which in our case is 3)
- y_i is 1 if the sample belongs to class i and 0 otherwise
- $\widehat{y_i}$ is the output for class i given by the CNN

The total cross entropy loss over the dataset is simply the sum of the individual losses:

3.2 Hyperparameter Tuning

After the preprocessing phase we divided as usual the dataset into training and test subsets with a 85/15 split. To ensure a balance in classes in both subsets, a stratified split was employed, which approximately maintains the same proportions for each class.
To tune the model hyperparameters we used a k-fold cross validation with $k = 5$ on the training subset. The hyperparameters we decided to consider were

- Learning rate
- Batch size
- Number of epochs

We decided not to consider the optimizer as a tunable hyperparameter, since further extending the hyperparameter search space would have significantly increased the computational cost of training, which was already very time intensive. Adam was used as the sole optimizer all throughout the experiments.

3.2.1 Hyperparameter Search Space

Since hyperparameters can influence each other, it is not sufficient to tune them independently, all possible combinations need to be considered. We therefore applied a grid search approach where the hyperparameter search space Θ is defined as the cartesian product of the chosen values for each individual hyperparameter. Formally, we define

$$\Theta = \mathcal{L} \times \mathcal{B} \times \mathcal{E}$$

where

- $\mathcal{L} = \{10^{-4}, 5 \times 10^{-4}, 10^{-3}\}$ denotes the set of learning rates under consideration
- $\mathcal{B} = \{16, 32, 64\}$ denotes the set of batch sizes under consideration
- $\mathcal{E} = \{10, 20, 30\}$ denotes the set of epoch values under consideration

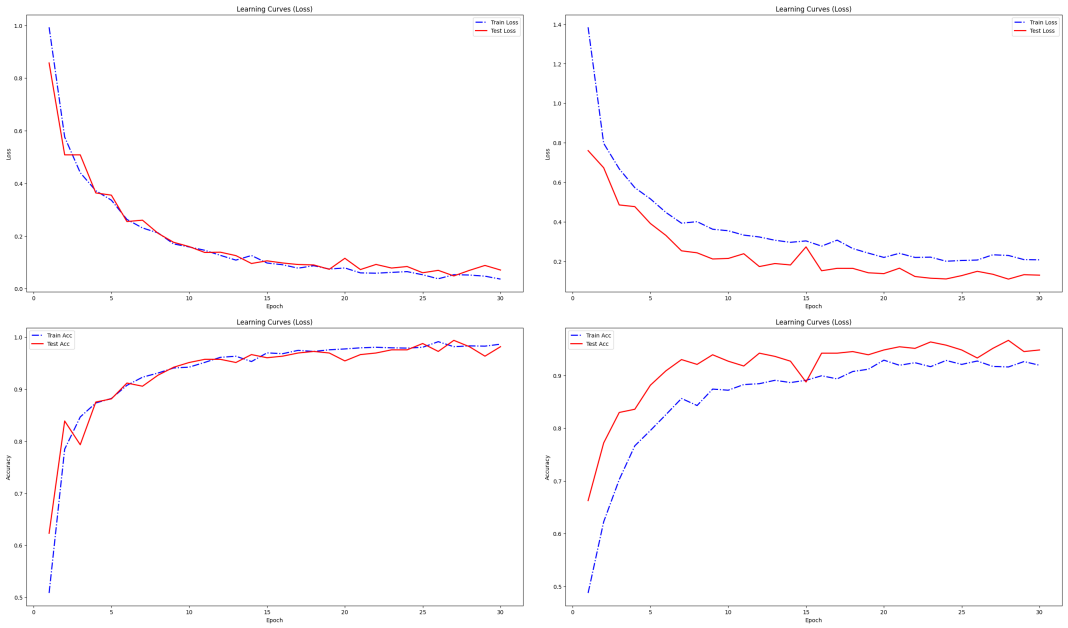
Each element $\theta \in \Theta$ therefore corresponds to a unique hyperparameter configuration (e.g. $(10^{-4}, 16, 10)$). The performance was measured in terms of the cross validation esimate ℓ^{cv}, precision, recall, F_1-score and accuracy across all five folds, these last metrics were calculated using the confusion matrix obtained on the validation set of each fold. The configuration achieving the lowest average cross validation estimate across all folds was selected as the best hyperparameter configuration.

We could have used a better estimator, such as the nested cross validation, but we decided against it due to the significantly increased computational cost, longer training time, and diminishing returns.
It is worthy of note that the grid search for hyperparameter tuning was performed ONLY on the First Architecture and the best performing hyperparameter configuration obtained from this search was subsequently used for the other architectures. The reason was that performing a full grid search for each architecture would have taken too much time, for reference, hyperparameter tuning just for the first architecture took 4 hours and 52 minutes. The best configuration found was $(10^{-4}, 16, 30)$.

4 Training And Evaluation

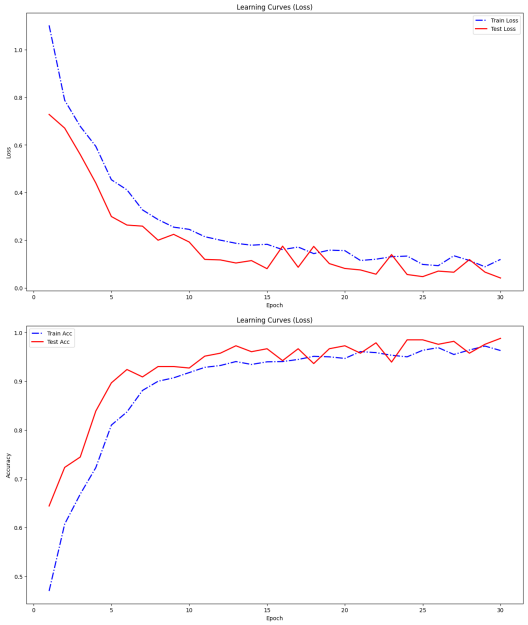
All experiments were performed on a laptop running MANJARO 6.12.44 with a AMD Ryzen 7 5800H CPU, 16 GB of RAM and a NVIDIA GeForce RTX 3050 Ti with CUDA 12.9.
During the hyperparameter tuning phase, for each configuration of hyperparameters and each fold, we printed all the evaluation metrics we talked about earlier and we plotted the corresponding training and validation loss curves, furthermore we computed and printed the average of these metrics across all folds for each configuration. The complete output and plots are available in the file `output_tuning.org`.

After finishing the hyperparameter tuning phase, the model was trained on the whole training subset using the best performing hyperparameters found earlier, as i said before the hyperparameter tuning phase was done only for the First Architecture and the best configuration we found there was then reused for the other architectures to save time.
The final model was then evaluated on the test subset. Predictions for each test data point were obviously obtained by selecting the class with the maximum output produced by the final layer of the network.

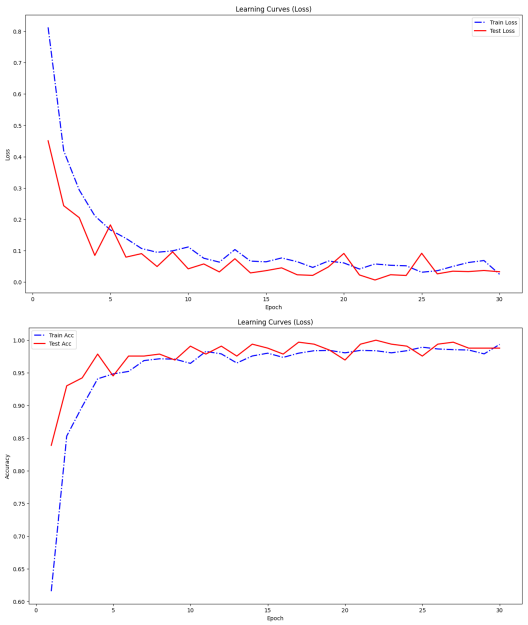


(1) First Architecture

(2) Second Architecture



(3) Third Architecture



(4) Fourth Architecture

Figure 4.1: Learning curves (loss and accuracy) for all four architectures. Each subfigure shows training and test loss (top) and training and test accuracy (bottom) over epochs.

Architecture	Loss	Precision	Recall	F_1-score	Accuracy
First Architecture	0.0763	0.9732	0.9724	0.9727	0.9726
Second Architecture	0.1161	0.9511	0.9478	0.9489	0.9483
Third Architecture	0.0425	0.9881	0.9875	0.9877	0.9878
Fourth Architecture	0.0201	0.9921	0.9909	0.9912	0.9915

Table 4.1: Evaluation metrics of the final model on the test set across all architectures (values are trucated).

Architecture	Loss (avg ± std)	F_1-score (avg ± std)
First Architecture	0.1064 ± 0.0425	0.9674 ± 0.0149
Second Architecture	0.1303 ± 0.0406	0.9574 ± 0.0191
Third Architecture	0.0783 ± 0.0267	0.9754 ± 0.0063
Fourth Architecture	0.0395 ± 0.0256	0.9912 ± 0.0047

Table 4.2: Average and standard deviation of loss and F_1-score across all five folds for all architectures using the best (or fixed) hyperparameter configuration.

4.1 Evaluation

We can now report and discuss the results obtained by the different architectures. For each architecture, we'll examine their learning curves (Fig. 4.1), which show the evolution of both training and test loss as well as training and test accuracy across all epochs. We'll also report the performance of the final model on the test set using all the aforementioned metrics (Table 4.1). In addition to gauge the stability of the results we'll present the average and standard deviation for the loss and F_1-score on the validation set of each fold, which we computed using the best (or fixed) hyperparameter configuration (Table 4.2).
We are not going to discuss each architecture in detail, as the results do not warrant it, instead we'll focus on the overall findings and we'll point out the differences between architectures when they occur, along with their possible causes.

Let's start from the learning curves (Fig. 4.1), all accuracy and loss curves show the training and test accuracies converging to nearly identical values, furthermore after a set amount of epochs all curves stabilize and reach a plateau, as expected the more complex architectures tend to stablize earlier than the simpler ones. All curves roughly follow each other in tandem and we dont see a conspicuous gap between training and test curves, more importantly we dont see the training and test curve starting to diverge from each other, like in the typical case where the training curve continues improving while the test curve degrades, which would indicate some kind of overfitting. So all things considered the learning curves suggest that no architecture exhibits overfitting.

On the other hand from Table 4.1 we can see how the increasing architectural complexity roughly correlates with increasing performance but with a notable outlier. It seems like the Second Architecture doesn't perform better then the first one, this can be reasonably justified by the fact that Second Architecture is essentially the first one but with dropout and batch normalization, and we know that dropout randomly deactivates a fraction of neurons during training, preventing the model from relying too much on specific activations therefore reducing overfitting, but since the First Architecture doesn't overfit to begin with, it is natural for the Second Architecture to perform worse.

Finally, Table 4.2 provides further evidence that the results in Table 4.1 are not are not a fluke, we can very explicitly observe how the trends in performance are somewhat stable

across different data splits, this suggests that the observed trend is a genuine behavior rather than being an artifact of a particular train/test split.

The results are exceptionally good, and better than we expected, but since we have a very small number of misclassified images across all architectures it is difficult to discern any consistent pattern among them, even after thoroughly examining all misclassifications and analyzing the confusion matrices on all architectures i couldn't provide any insight into any model weakness.

4.2 Feature Extraction

We attempted to gain some sort of intuition into the feature extraction process by visualizing convolutional filters and activation maps. However since the convolutional kernels are only 3×3 they were way too small to show any meaningful pattern, and the activation maps did not yield any interpretable features either. More sophisticated visualization techniques exist but they are outside the scope of this course.

5 Generalization

All architectures, especially the fourth one, seem to perform incredibly well, but does this generalize to images outside the dataset? to answer this question we took 33 images of my hand, with equal proportions of rock, paper and scissors and used a trained model on the fourth architecture to predict them. I intentionally tried to take those images as varied as possible, including different proportions, hand orientations, palm up or down, backgrounds (green and grey) and whether the arm was visible or not. Unfortunately i was explicitly prohibited from publishing these images to preserve privacy.

The results are quite interesting and give us some nice insights on the limitations of our model and our dataset. All paper images were consistently classified correctly with high confidence. Meanwhile the rock images were only correctly predicted when the hand was palm down, more specifically all palm up rock hands were generally misclassified as paper with high confidence, this can be reasonably attributed to the fact that the dataset contains only one single example of a palm up rock. As for the scissors images, those with the thumb extended or palm up were often misclassified as paper, this can be similarly justified by the complete lack of training examples featuring a thumb out hand and only a single palm up hand in the dataset. To finish off, another noteworthy observation is that all images with a green background were generally correctly predicted and with noticeably higher confidence compared to those with a grey background, which makes me believe that even after the data augmentation the model retains some kind of dependence on the background. These patterns of misclassification are clearly illustrated in the confusion matrix shown in Fig. 5.1.

In my opinion, these results suggest that better generalization appears to be predominantly constrained by the dataset itself rather than the architectures per se, more novel and varied

images with possibly different backgrounds, palm orientations, and variations of gestures would likely be needed.

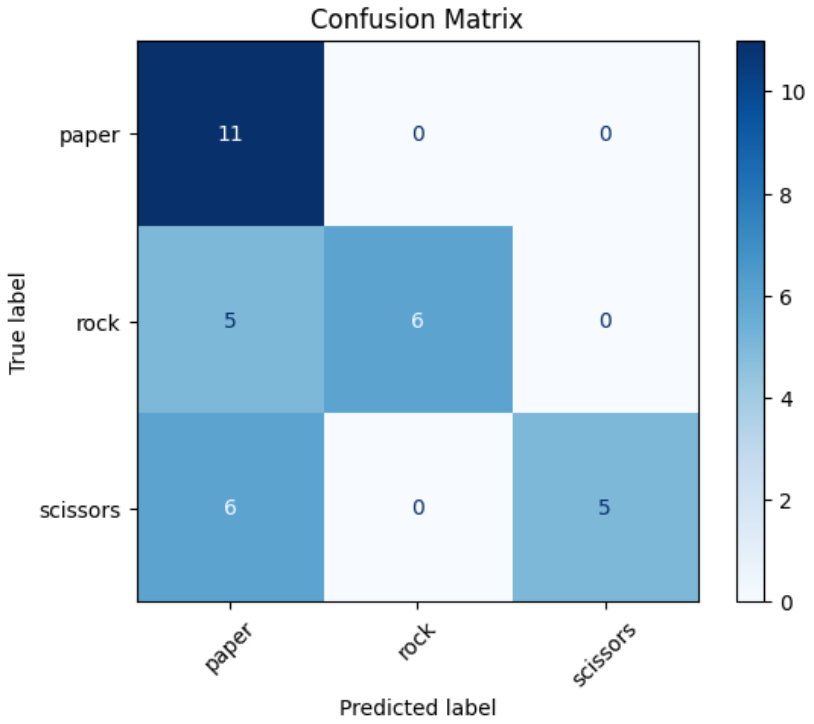


Figure 5.1: Confusion matrix on the generalization dataset

6 Conclusion

So in the end all proposed architectures achieved very strong results on the test set, with no clear signs of overfitting, and the more complex models generally converged faster and with better performance metrics.

However, our generalization experiment revealed important limitations, the model's ability to generalize was severly constrained when exposed to new images with characteristics not well represented in the training set, such as palm up hand, variations in hand geastures, or different backgrounds. Therefore, in my opinion, that the dataset itself rather than the model architecture is the main culprit, limiting further improvements.

Possible future work could be, collecting a more diverse dataset along the lines previously discussed. Additional augmentation strategies could have been tried such as background replacement, this could have been implemented either through a dedicated background segmentation model or with classical image processing techniques like those described in Chapter 10 of [2], such as Otsu's method, edge detection combined with flood filling or even, and probably the most promising, some type of color-based segmentation, these approaches however fall outside the scope of this course, and therefore were not implemented.

A Appendix

Figure A.5: All four CNN architectures side by side.

Bibliography

[1] DrGFreeman. *Rock-Paper-Scissors Dataset*. https://www.kaggle.com/datasets/drgfreeman/rockpaperscissors/data. 2018.

[2] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing (4th Edition)*. 2018.

[3] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[4] Yann LeCun et al. "Efficient BackProp". In: *Neural Networks: Tricks of the Trade*. Ed. by Genevieve B. Orr and Klaus-Robert Müller. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 9–50. ISBN: 978-3-540-49430-0.

[5] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[6] Richard Szeliski. *Computer Vision: Algorithms and Applications 2nd Edition*. 2021.