# Reimplementing the AFFACT-Paper

Parallel Implementation of Data Augmentation for Alignment-Free Facial Attribute Classification in PyTorch

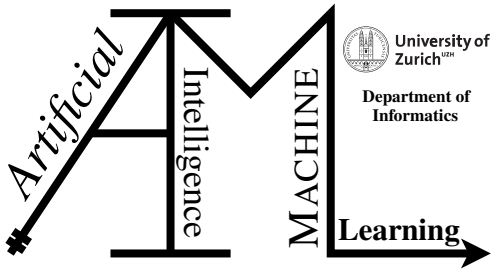
Master Project

**N. Chavannes, Y. Rutishauser**

16-701-872, 16-701-658

**Submitted at**
10.02.2021

**Thesis Supervisor**
Prof. Dr. Manuel Günther



**Master Project**

**Author:** N. Chavannes, Y. Rutishauser

**Project period:** 08.09.2020 - 10.02.2021

Artificial Intelligence and Machine Learning Group
Department of Informatics, University of Zurich

# Abstract

Facial attribute classification tasks have many potential real-world applications across different areas (e.g. surveillance, entertainment, medical treatment). To make classification models more stable toward misaligned images, a data augmentation approach called "alignment-free facial attribute classification technique" (AFFACT) was introduced by Günther et al. [18]. Since their approach was originally implemented using the nowadays outdated Caffe framework, this master project aims to reimplement this technique using the more popular PyTorch framework. Our reimplementation performs on the same level as the original AFFACT model. We further introduce the possibility to perform hyperparameter optimization on existing models and establish an extended network trained on an additional data augmentation operation.

# Zusammenfassung

Klassifizierungsaufgaben für Gesichtsattribute haben viele potenzielle Anwendungen in unterschiedlichen Bereichen (z. B. Überwachung, Unterhaltung, medizinische Behandlung). Um die Stabilität der Klassifikationsmodelle gegenüber falsch ausgerichteten Bildern zu erhöhen, wurde von Günther et al. eine Technik zur Data Augmentation eingeführt, die "Alignment-free Facial Attribute Classification Technique" (AFFACT) genannt wird [18]. Da der Ansatz ursprünglich mit dem heutzutage veralteten Caffe-Framework implementiert wurde, ist das Ziel dieses Masterprojekts, diese Technik mit dem populäreren PyTorch-Framework zu re-implementieren. Unser Nachbau erreicht das gleiche Performance-Niveau wie das ursprüngliche AFFACT-Modell. Darüber hinaus führen wir die Möglichkeit ein, Hyperparameter-Optimierung an bestehenden Modellen durchzuführen und schlagen ein erweitertes Netzwerk vor, welches mit einer zusätzlichen Data Augmentation Technik trainiert wurde.

# Contents

# List of Figures



# List of Tables

# Chapter 1

# Introduction

Nowadays, machine learning is widely used for pattern recognition problems. Especially deep learning has become a popular method within machine learning to solve difficult natural language processing and image analysis tasks that require a vast amount of training data. Usually, image processing training data is artificially increased by scaling, mirroring, and cropping the training images. This process is called data augmentation. Furthermore, data augmentation can be used to improve the robustness of a neural network by increasing the variation in the training data. In the AFFACT paper, Günther et al. [18] show that using extended data augmentation techniques such as rotation, scaling, and blurring the training faces, enabled the network to waive the otherwise required face alignment step. In the mentioned paper, the data augmentation, the network, and the training procedure are implemented in C++ using the Caffe framework [19]. Since Caffe is outdated now, this master project aims to port the existing AFFACT project to Python. To do so, we reimplemented the data augmentation and translated the code using PyTorch (a popular deep learning framework for Python) [22]. Furthermore, we organized the code in a well-structured, meaningful, and extendable way that follows a standard data science pipeline. Additionally, we added the possibility to extensively monitor the experiments using the Weights and Biases framework [12] with customized, informative training and evaluation charts.

The report is structured in the following way. First, we give some insights about our local setup and the dataset used. Then we discuss the milestones listed below:

- Milestone 1 (Chapter 3): We reimplemented the AFFACT deep facial attribute classification network by adhering to the original architecture using a pretrained ResNet-50 and added one additional fully connected layer. We trained this network on pre-aligned images provided by the CelabA dataset and implemented an evaluation method.
- Milestone 2 (Chapter 4): We implemented the image processing/data augmentation techniques introduced in the AFFACT paper [18]. This allowed us to generate a randomly perturbed face image in the desired image resolution based on an input image and its facial landmarks.
- Milestone 3 (Chapter 5): We implemented a parallel processing scheme including data shuffling, i.e. batches of randomly perturbed images are generated in parallel for several training epochs.
- Milestone 4 (Chapter 6): We trained the AFFACT network [18] and extensively evaluated it. The network is trained and is more stable to misalignment than the network from Milestone 1.

Lastly, we explain our contribution and conclude this master project.

# Chapter 2

# Setup and Dataset

## 2.1 Development Setup

We mainly spent the first week of the semester project setting up our development environment and familiarizing ourselves with the existing code base and the research problem in general.

### 2.1.1 Familiarizing With AFFACT

First, we had to familiarize ourselves with the AFFACT project. After reading and discussing the paper, we tried to understand the existing source code written in C++. As we are both not familiar with C++, we had to familiarize ourselves with C++ to understand the AFFACT code.

### 2.1.2 Local System

A UNIX-based operating system is required to use the recommended library for the image transformations (bob). Therefore we set up a dual boot on our local system using Ubuntu and Windows. The system uses a GeForce RTX 3070 (8GB) and a Ryzen 9 3950X (16C/32T), and 32GB RAM. Installing the proper drivers and libraries that allow the GPU to be accessed within PyTorch was a tedious task.

### 2.1.3 Project Structure

Our goal was to create a project structure that embraces very fast experimentation, easy reproduction, and is set up in a modular way. Experiments can easily be reproduced and simple modifications can be made using console arguments or several separate configuration files.

The project's modular structure follows a standard data science pipeline consisting of preprocessing & dataset generation, selection of the model, the training process, and the evaluation of the trained model.

We created several configuration files that hold all necessary information about the whole machine learning pipeline to achieve fast experimentation. Each step of our pipeline can be configured separately and all parameters can also be overwritten using command line arguments. This enables fast execution and adjustment of the experiment on any system. To guarantee the reproducibility of our experiments, we set seeds in the Numpy and PyTorch framework. This initializes the network's weights and shuffles the data batches in the same way when the experiment is rerun.

The experiments' results are saved in a distinct folder, marked by a timestamp and experiment name. This folder contains the training history, the training configuration file, sample images if desired and the final model weights. Furthermore, we created a Readme-File in the source code [16], which explains how the project can be used.

## 2.2 Dataset CelebA

The CelebA (CelebFaces Attributes) dataset is widely used for the following computer vision tasks: face attribute recognition, face detection, landmark (or facial part) localization, and more [20]. "CelebA is a large-scale face attributes dataset with more than 200'000 celebrity images, each with 40 attribute annotations. The images in this dataset cover large pose variations and background clutter" [20]. More specifically, CelebA covers 10'177 identities, 202'599 images, and 5 landmark locations, 40 binary attributes annotations per image [20].

We split the CelebA dataset according to the partition file which is found in the original dataset. The split results in roughly 160'000 images used for training, 20'000 used for validation and 20'000 used for evaluation. The partitioning ensures that identities that are present in the training set are not present in the test set and vice versa. The same is true for the validation set.

### 2.2.1 Cropping the images

A main challenge poses the task of cropping the image at the right position, so all attributes for classification are still visible on the picture, but background data is disregarded as much as possible. There are three possible strategies to crop the images:

- Crop the image based on a bounding box: The CelebA dataset provides bounding boxes for every image in the dataset. These bounding boxes can be used to crop the images.
- Crop the image based on a bounding box that is calculated using landmarks: The CelebA dataset provides locations of several landmarks for every image in the dataset. These landmarks can be used to calculate bounding boxes and the rotation angle based on the eye and mouth landmarks. These bounding boxes are then used to crop and align the images. When images are aligned based on these bounding boxes, facial attributes (e.g. nose, eyes, mouth) are located almost at the same position for every image/subject. This strategy is called alignment.
- Automatically detected bounding box: A face detector can be used to calculate a bounding box for each image. These bounding boxes can be used to crop the pictures.

For training all the networks, we used the second strategy of calculating bounding boxes based on landmarks. The same strategy was used by the AFFACT paper [18].

### 2.2.2 Evaluation Sets

To provide a fair comparison between the models, we evaluated the models on different crops and transformations of the test set. The datasets are defined as follows:

- Test set A (aligned with landmarks): The images are aligned according to the hand-labeled landmarks found in the dataset.

- Test set C (ten crops): The bounding boxes of the images are automatically detected with the MTCNN face detector. As these bounding boxes are narrow, we scale them by a constant factor of 2, such that the re-scaled bounding box has approximately the same size as bounding boxes calculated based on landmarks. Then, the image is cropped to the re-scaled bounding box, and the ten-crop strategy is applied, which yields ten slightly shifted images. The Ten-crop operation crops a given image into its four corners as well as into its center. Additionally, these five crops are horizontally flipped [11]. The model predicts on all ten images, and the majority guess is used as the final attribute prediction.
- Test set D (detected bounding boxes): The bounding boxes used to crop the images are detected using the MTCNN face detector from facenet-pytorch [4]. We used a scaling factor of 2 to re-scale the bounding boxes.
- Test set T (transformations): The MTCNN face detector is used to detect the bounding box (using a scaling factor of 2). Then several random transformations are applied to the images. The transformations applied are the same as discussed in Chapter 4.

In our project code, we created the file `generate_test_datasets.py` that generates these four test sets. This ensures that all models are evaluated on the same images. Even though the original images from the CelebA dataset are provided in the JPG file format, we saved all images from the test sets in the PNG file format. When saving the images as JPG files, we observed that the models' average performance decreased by approximately 0.75%, due to the compression of the JPG file format that is enabled by default.

# Chapter 3

# Milestone 1

## 3.1 Motivation

The goal of milestone 1 was to create a deep facial attribute classification network and train and evaluate it on aligned face images. The images were aligned using the landmarks provided by the CelebA dataset. We call this network ResNet-51-S. We further assumed that the model would perform better than just predicting the majority class of each attribute. To compare this model with the baseline (predicting majority classes), we had to implement a standard machine learning pipeline. The pipeline consists of preprocessing the aligned images, choosing a proper network architecture, implementing a training loop, and evaluating the results in a meaningful way. The ResNet-50 network has already been available in PyTorch. We only had to extend the network with an additional layer, which yielded the ResNet-51-S.

## 3.2 Methodology

This section covers the details of the data science pipeline we implemented in this project. First, we illustrate the process of loading the data, preprocessing the images, and transforming them so that they can be fed batch-wise into the network. Next, we show how we adapted the ResNet-50 network for our use case. Furthermore, we describe how we implemented the training loop, and lastly, we report our results.

### 3.2.1 Preprocessing

The first step of preprocessing the images was to load the labels and the landmarks from the CelebA dataset. A partition file determines the split between the training, validation, and test set. In PyTorch, one can use data loaders [8] which allow you to easily load your data during training in batches. A data loader needs an instance of a dataset and an instance of a sampler class and generates an iterable over the given dataset. We created a `CelebADataset` class that inherits from the original torch dataset class. This allowed us to specify the operations applied to each image before it was used in training. This process is visualized in Figure 3.1. To prepare the images for training, we used the transforms API from torchvision, which allowed us to write custom transformers [10]. The transformer either uses bounding boxes provided by a face detector or calculates a bounding box around a subject's face based on the dataset's landmarks. A bounding box consists of the top left corner (X and Y coordinates) as well as its width, height, and rotation angle. When calculating the bounding box based on landmarks, the center of both eye coordinates ($\vec{t_e}$)

as well as the center of both mouth corners ($\vec{t_m}$) are calculated. The formula is given in equation (3.1).

$$\vec{t_e} = \frac{\vec{t_{e_r}} - \vec{t_{e_l}}}{2},\ \vec{t_m} = \frac{\vec{t_{m_r}} - \vec{t_{m_l}}}{2} \tag{3.1}$$

The distance ($d$) between those two center points is the basis for calculating the width ($w$) and height ($h$) of the bounding box, given in equation (3.2). This distance is multiplied by a constant which yields the actual height and width of the bounding box.

$$\begin{aligned} d &= \| \vec{t_e} - \vec{t_m} \| \\ w &= h = d \cdot 5.5 \end{aligned} \tag{3.2}$$

To calculate the X and Y coordinates, the center point between both eyes is used. On the horizontal axis, the bounding box is centered. On the vertical axis the bounding box is slightly shifted upwards. Equation (3.3) shows this procedure (specifying left, right, top and bottom coordinates $x_l, x_r, y_t, y_b$).

$$\begin{aligned} x_l &= x_e - 0.50 \cdot w \\ x_r &= x_e + 0.50 \cdot w \\ y_t &= y_e - 0.45 \cdot h \\ y_b &= y_e + 0.55 \cdot h \end{aligned} \tag{3.3}$$

Additionally, the rotation angle ($\alpha$) is calculated based on the coordinates of both eye positions. The formula is given in equation (3.4). When using bounding boxes provided by a face detector, the rotation angle is set to zero.

$$\alpha = arctan \frac{y_{e_r} - y_{e_l}}{x_{e_r} - x_{e_l}} \tag{3.4}$$

After calculating a scaling factor as well as the crop center based on the new bounding box, geometric normalization is applied using the `bob.ip.base` library [5]. Geometric normalization combines rotating, scaling and cropping an image (See Fig 3.2). The algorithm is described in detail in Section 4.2.

The images yielded were converted to tensors, which changed the pixels' range from $[0, 255]$ to $[0, 1]$ by dividing each pixel by $255$. Additionally, we normalized the images by centering each channel's pixel values (RGB) with means and standard deviations derived from the original ImageNet dataset [21]. The means used for normalization were 0.485 (red), 0.456 (green), 0.406 (blue); standard deviations used were 0.229 (red), 0.224 (green), 0.225 (blue). Now the tensor was ready to be used in training.

### 3.2.2 Model Architecture

To create the ResNet-51-S model, we chose the same model architecture as the AFFACT model [18]. We took the popular ResNet-50 architecture and added a fully-connected feed-forward layer with 40 output nodes (=number of attributes). We activated the last layer by using the sigmoid function. The sigmoid function is the industry standard for multi-label problems (classes are not mutually exclusive) [3]. When combining a sigmoid activation on the last layer with a binary cross-entropy loss, PyTorch suggests using the `BCEWithLogits` loss function. `BCEWithLogitsLoss` internally combines a binary cross-entropy loss and a sigmoid activation function, which promises better results than using them separately [1]. To improve the predictions' quality, we initialized the ResNet-50 part of the network with weights that were pretrained on the ImageNet dataset

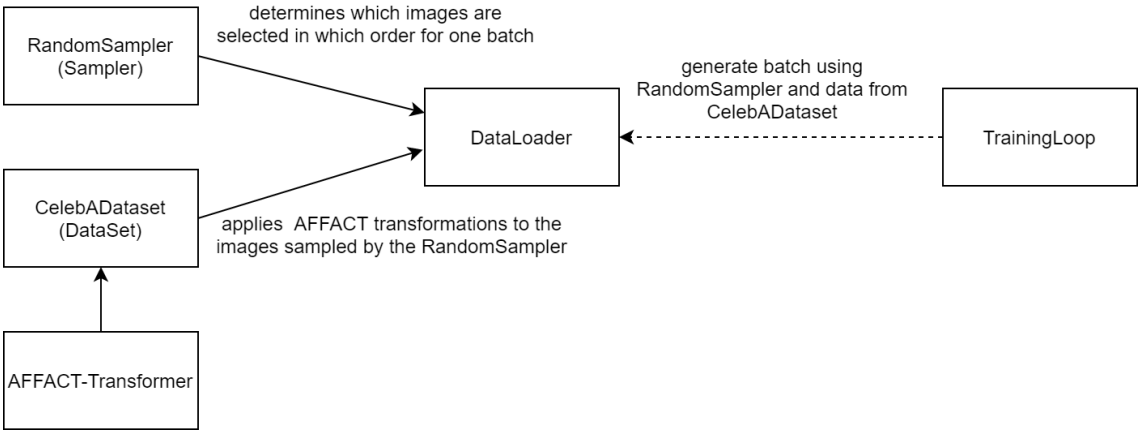



**Figure 3.1**: Simplified view of generating batches during training using a data loader.

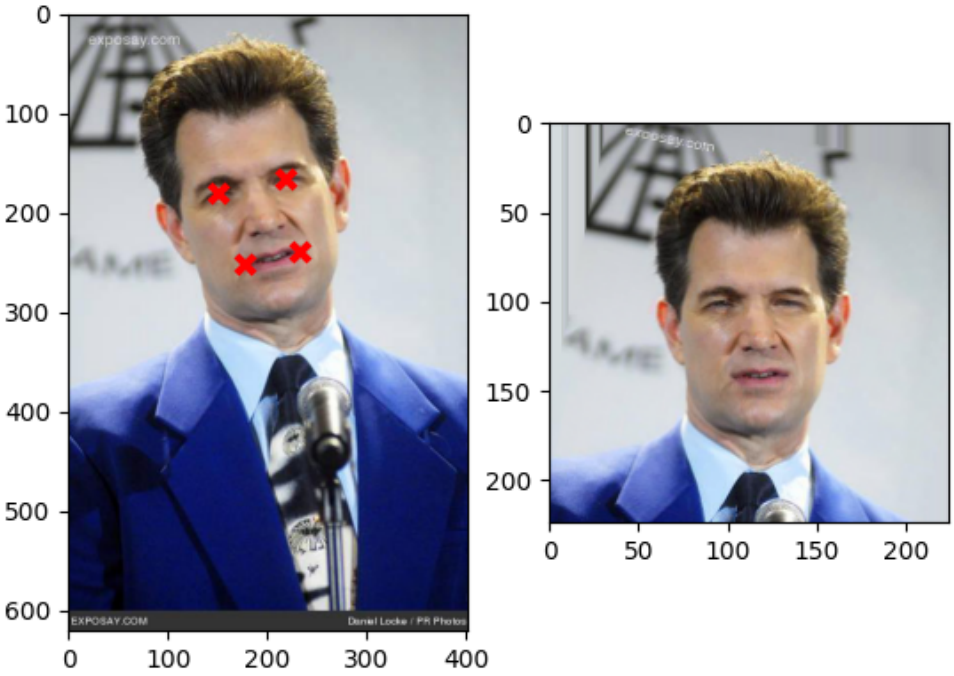



**Figure 3.2**: Example image with hand-labeled landmarks (left), image after preprocessing and applying geometric normalization.

[17]. The weights of the additional fully-connected layer were initialized based on formula (3.5), which is the default initialization method of PyTorch for linear layers [13].

$$\mathcal{U}(-\sqrt{k}, \sqrt{k}),\ k = \frac{1}{in\_features} \tag{3.5}$$

### 3.2.3 Training

To streamline the training process, we implemented a training manager which sets up the training process. This included loading the preprocessed dataset, retrieving the model, setting up the loss criterion, and the optimizer. The hyperparameters for all training related instances were loaded from the global configuration file. We used binary cross-entropy (BCE) as the loss criterion, and `RMSprop`, which is a gradient-based optimizer, to update the weights during training [7]. After the training manager had been initialized, we were able to start the training on this instance.

The goal of the training loop is to fit the model on the data. To do so, we looped through the whole dataset several times. In each epoch (= iteration), we went through two phases. In the first phase, which is called the training phase, we loaded the data in randomized batches and calculated our model's facial attribute predictions based on the input. The predictions were then compared to the ground-truth, and the BCE loss was calculated. We performed backpropagation based on the calculated loss and updated the weights of our network according to the optimizer's policies and with regards to the hyperparameters. In the second phase, which was called the validation phase, the model predicted on the validation dataset. The weights were fixed and not updated in this phase.

The number of epochs, the batch size, hyperparameters for the optimizer, the loss type and hyperparameters for the learning rate scheduler can be adjusted in the configuration file. We saved the model every five epochs and furthermore saved the best model, according to the highest accuracy on the validation set, at the end of the training. To monitor our model's performance, we computed standard metrics like accuracy and loss on the training and validation set. Furthermore, we calculated more sophisticated metrics, which are described in the next section. A simplified version of the training loop is depicted in Algorithm 1.

For training the model, we used a batch size of 64, an initial learning rate of $10^{-4}$. The model was trained using a learning rate scheduler and re-initialization technique described in more detail in Section 7.3. We used the same hyperparameters as in the AFFACT paper [18] but automated the retraining with a lower learning rate. We trained the model over 6 epochs.

## 3.3 Result

This section discusses the qualitative and quantitative analysis of the ResNet-51-S model.

### 3.3.1 Quantitative Analysis

In the quantitative analysis, we analyze the model's accuracy on the four test sets (A, C, D, T) introduced in the dataset chapter. The baseline of the comparison is given by the majority class of the attribute. For example, if most people in the training data are not bold, the baseline algorithm predicts not bold for the entire test set.

Figure 6.1 shows the overall error rate (1 - accuracy). Overall, we report an error rate of 8.15% on test set A, 10.51% on test set C, 9.22% on test set D and 13.14% on test set T, very similar to the error rates reported in the AFFACT paper [18]. Moreover, figure 6.1 shows the error rate per attribute on all test sets. For example, the baseline error rate of the attribute "arched eyebrows"

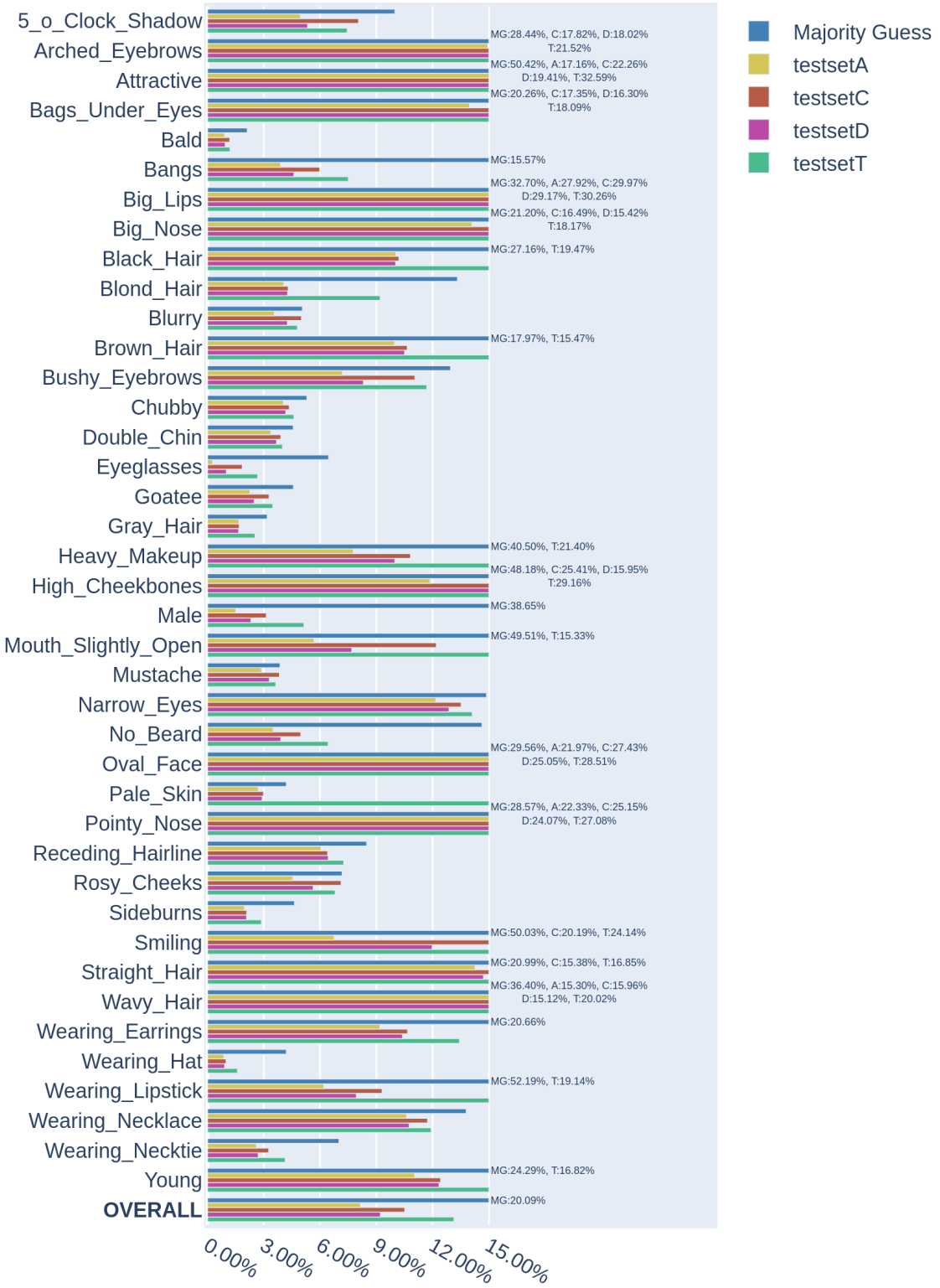



**Figure 3.3**: Attribute-wise accuracy of the ResNet-51-S model on four different test datasets. The error rates are cut off if the value exceeds 15%. In that case, the true error rate is annotated next to it. (Majority Guess = Accuracy when only guessing the majority class)

**Algorithm 1** Training Loop

**for** each epoch **do**
  **for** $input\_batch$ and $labels\_batch$ in training data **do**
    $predictions = model.predict(input\_batch)$
    $loss = BCELoss(predictions, labels\_batch)$
    Calculate backpropogation based on $loss$
    Adjust model weights based on optimizer
    Calculate accuracy based on $predictions$
  **end for**
  **for** $input\_batch$ and $labels\_batch$ in validation data **do**
    $predictions = model.predict(input\_batch)$
    $loss = BCELoss(predictions, labels\_batch)$
    Calculate accuracy based on $predictions$
  **end for**
  Adjust learning rate if needed
  Save state of best model
**end for**
Save best model

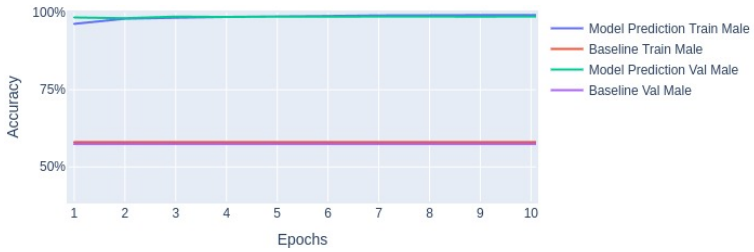



**Figure 3.4**: Accuracy of gender prediction over 10 epochs.

is 28.44% while the error rate of the trained model is 14.94% on the same attribute on test set A. On test set T, the model achieves an error rate of 21.52%. We detect a similar pattern on other attributes as well, where the model performs significantly worse on test set T than on the other test sets. Furthermore, we show that the ResNet-51-S model is not very robust to detect facial attributes on images that are not aligned. The model seems to have difficulties when the face is only cropped but not aligned using the rotation angle. For example, the model performs much better on the "smiling" attribute when the image is aligned. We deduct that the model memorizes the mouth's pixel position and is not robust when the mouth appears at a different place, which is likely on the other test sets. More examples of this behavior can be observed for "wearing lipstick" and "eyeglasses".

There are only a few attributes where the model performs similarly across all test sets. These include: "bald", "chubby" and "sideburn". We assume that the model sees a lot of variation of these attributes during training.

We further conclude that the pretrained weights of the ResNet-50 in the convolutional layers give the network a good starting point for many attributes. This observation is consistent with the data of the training history. The network starts with 90% accuracy instead of 85% (majority guess). For instance, the network is nearly immediately able to distinguish between male and female as indicated in figure 3.4.

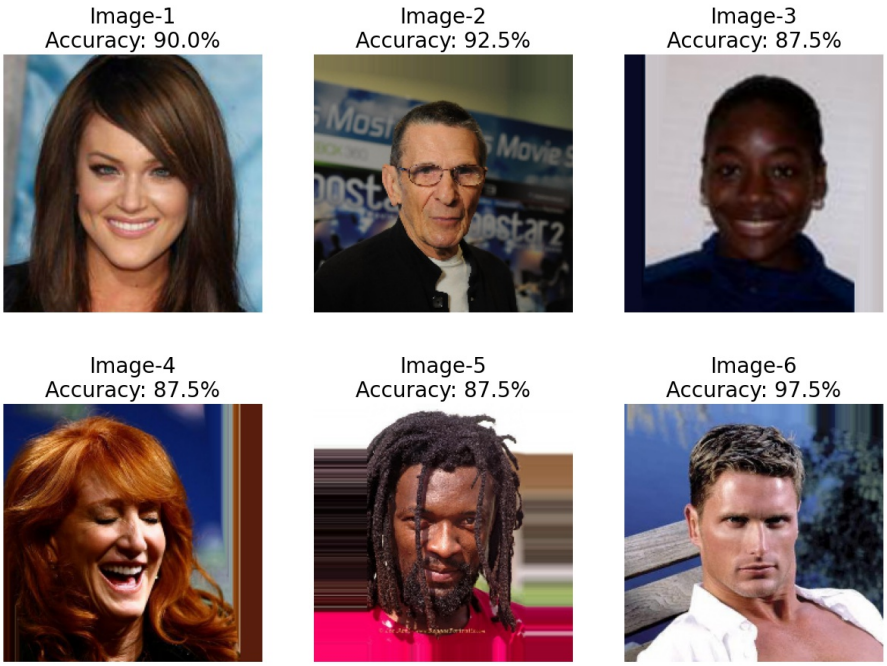



**Figure 3.5**: Sample images with prediction accuracy over all attributes. The samples are taken from test set A.

### 3.3.2 Qualitative Analysis

In the qualitative analysis, we want to look at a sample of images from the test set A and evaluate the prediction quality of the ResNet-51-S model (Figure 3.5). To do so, we took six images that are preprocessed in the same way as during the training process. The overall accuracy on the images is on average above 90%, which was to be expected when looking at the results from the quantitative analysis. In the appendix, there is a related table (Figure A.1) which looks at each of these six images individually and shows attribute wise correctness of the prediction. The "yes" or "no" in the table cells indicate the value the model predicted. The background color shows whether a prediction was correct (green) or incorrect (red). We do not observe specific attributes where the model has apparent difficulties in making an accurate classification. We notice that some attributes might be subjective. It is not easy to clearly define when a hairstyle changes from being wavy to being straight. Such attributes are difficult to always get right. But there are other attributes where the dataset is not always labeled correctly. For example, the man on image 6 clearly has no beard, even though the dataset suggests otherwise.

## 3.4 Conclusion

The results show that the performance of the ResNet-51-S is quite impressive on the aligned test set (A). The performance drops, though, as soon as the test sets' images are not aligned anymore using a rotation angle and landmarks (test sets C, D, T). If there are additional transformations applied (test set T), the performance drops even more. The model's performance is very similar to the error rate reported in the AFFACT paper [18].

# Chapter 4

# Milestone 2

## 4.1 Motivation

The second milestone's goal was to implement the data augmentation technique from the AFFACT paper [18]. The method includes the following operations: shifting, scaling, gamma manipulation, blurring, mirroring, and rotating. These operations were performed to increase the robustness of the network regarding misalignment, focus, and quality. For example, we randomly shifted the image, so the network does not blindly memorize specific attributes' locations but actually learn to identify these attributes correctly. Furthermore, we used gamma transformations, so the network gets more robust to different exposures. Blurring, mirroring, and rotating the image is self-explanatory and contributes to higher training data variance. Additionally, we transformed each image differently in every epoch during training, making it harder for the model to overfit the training data.

## 4.2 Methodology

To generate randomly perturbed images of faces according to the AFFACT paper [18], we implemented a custom transformer that can be used in the data loader. As discussed in the previous section, the data loader is responsible for providing input data batches based on the training dataset. The transformer is called every time the data loader has to create a new batch of input data. For each input image, the transformations defined in the custom transformer are applied. As the data loader generates the batches again for each epoch of training, the images are also perturbed differently each epoch.

The transformer needs to receive the input image and either landmarks or bounding boxes provided by the dataset or by a face detector to apply transformations to an image. If landmarks are provided, the bounding box is calculated according to the technique and equations of Subsection 3.2.1. The technique described previously yields a newly calculated bounding box as well as a rotation angle. The final crop size is loaded from the training configuration file. Based on the crop size and the size of the bounding box, a scale variable is calculated. A randomly drawn number based on a normal distribution is added to the rotation angle. The mean and standard deviation of the normal distribution are loaded from the training configuration file. The original scaling factor is multiplied by 2 to the power of a randomly drawn factor based on a normal distribution. Additionally, a horizontal and vertical shift is calculated on a normal distribution as well. Based on the crop size and the horizontal and vertical random shift, a crop center is defined. Using the `bob.ip.base` library [5], we can define a geometric normalization using the newly

defined crop center, the angle, the scale, and the crop size. In the process of geometrically normalizing the image, missing pixels are extrapolated. Extrapolating images is necessary when the bounding box exceeds the border of the image. If that is the case, missing pixels are estimated using the nearest neighbor technique [6]. After these processing steps, we have an image with the input shape required by the network that is centered (when neglecting the random shifts, rotations, and scaling) on a person's face. We then can randomly flip the image, apply a Gaussian blur, or change the gamma of the image. The algorithm described is shown in Algorithm 2.

**Algorithm 2** AFFACT Transformations for a single image

**if** landmarks for an image are given **then**
    Calculate bounding box according to equations (3.1), (3.2), (3.3), (3.4)
**else**
    Use the bounding box given
**end if**
Read the final image crop size from the configuration given
Calculate a $scale\_factor$ based on: $min(\frac{crop\_size_x}{bounding\_box\_width}, \frac{crop\_size_y}{bounding\_box\_height})$
**if** scaling enabled **then**
    $scale\_factor = scale\_factor \cdot 2^x$, $x \sim \mathcal{N}(0,\, 0.1)$
**end if**
**if** rotation enabled **then**
    $rotation\_angle = rotation\_angle + x$, $x \sim \mathcal{N}(0,\, 20)$
**end if**
**if** shift enabled **then**
    $shift_x = crop\_size_x \cdot z$, $z \sim \mathcal{N}(0,\, 0.05)$
    $shift_y = crop\_size_y \cdot z$, $z \sim \mathcal{N}(0,\, 0.05)$
**end if**
Calculate crop center according to: $crop\_center = [\frac{crop\_size_x}{2} + shift_x, \frac{crop\_size_y}{2} + shift_y]$
$geomnorm$ = *bob.ip.base*.$GeomNorm(rotation\_angle, scale\_factor, crop\_size, crop\_center)$
**for** $color\_channel$ of input image **do**
    Apply geometric normalization on $color\_channel$ using $geomnorm.process$
    Extrapolate missing pixels using $bob.ip.base.extrapolate\_mask$
**end for**
**if** mirror enabled **then**
    Flip image horizontally
**end if**
**if** gaussian blur enabled **then**
    Calculate $\sigma$ based on: $\sigma \sim \mathcal{N}(0,\, 3)$
    gaussian = *bob.ip.base*.$Gaussian((\sigma,\, \sigma),\, (int(3 \cdot \sigma),\, int(3 \cdot \sigma)))$
    Apply gaussian filter to image
**end if**
**if** gamma enabled **then**
    Calculate $\gamma$ based on: $\gamma \sim 2^{\mathcal{N}(0,\, 1)}$
    Apply transformation to the image with $image = \min(\max(((\frac{image}{255})^\gamma) \cdot 255,\, 0),\, 255)$
**end if**
normalize image based on: $image = \frac{image}{255}$

## 4.3 Results

Examples of transformation are shown in figure 4.1. The transformations are applied to three sample images separately and combined.

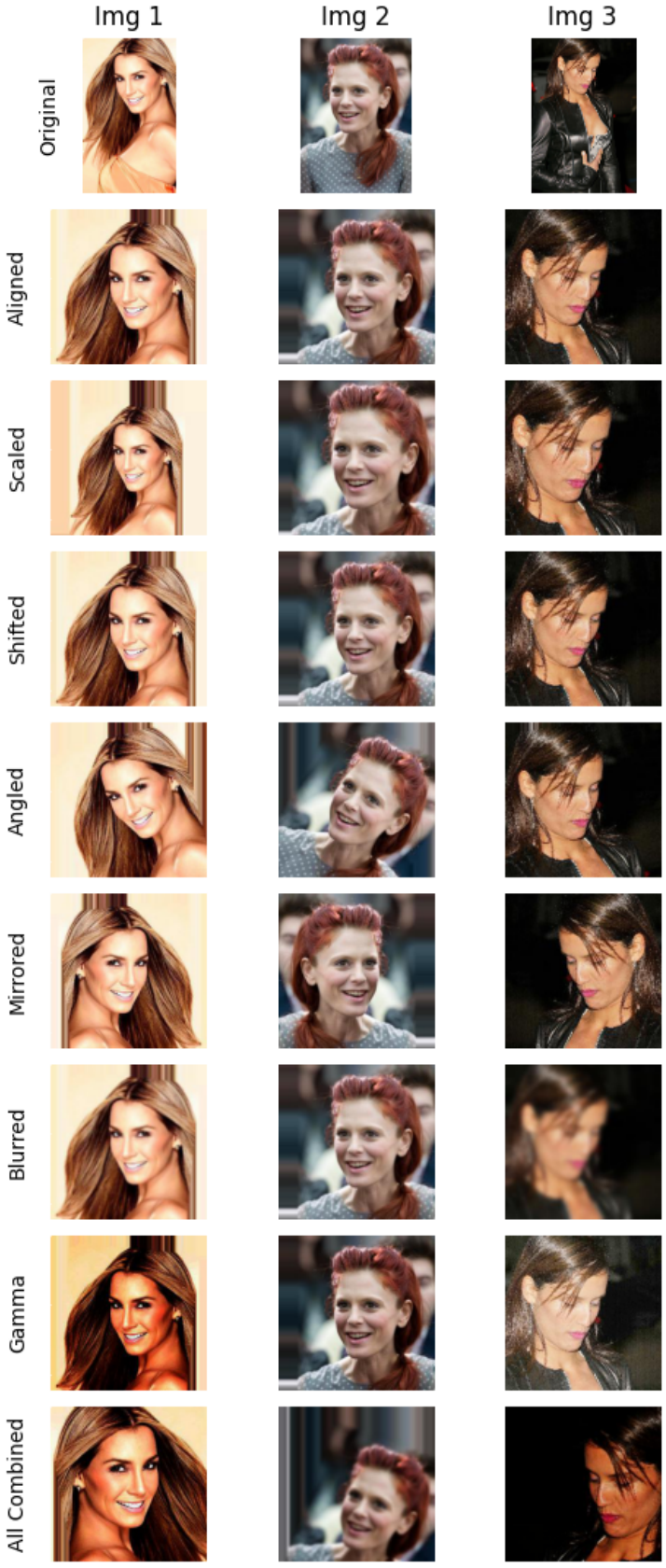



**Figure 4.1**: All transformations shown separately and combined on three sample images.

# Chapter 5

# Milestone 3

## 5.1 Motivation

Training a neural network is usually very computationally expensive and takes a lot of time, depending on the network's size and complexity. Typically, the computations needed to train a network are performed on the GPU. During training, a forward pass and backward propagation have to be calculated for each sample. This can be speeded up by calculating all samples of a batch at once using matrix calculations. GPUs are very efficient in matrix calculations and, furthermore, calculate in parallel instead of a sequential manner. In order to make use of the full speed of modern days GPUs, they need to be fed with the data batches in time. These batches are prepared on the CPU, which is very powerful in regards to sequential computing. But as for each epoch in the training loop, multiple batches of multiple samples need to be prepared; the CPU with the sequential approach might be overwhelmed and bottleneck the GPU. Especially in image augmentation, the preprocessing/preparation of the images is more computationally expensive than in other cases. To mitigate this, we can make use of the multiple cores that are found in modern processors. Each processor core can prepare/augment an image in a sequential manner. Since there are multiple cores on a CPU, we can also prepare the images in parallel and mitigate the bottleneck.

## 5.2 Methodology

Preprocessing images in a parallel manner can be easily implemented in Python by making use of the `torch.utils.data` package from PyTorch [8]. As described in the methodology section of milestone 1, we implemented a custom CelebA dataset class which holds all information about a sample that is needed to preprocess an image. Such a dataset class can be passed to a data loader. The data loader needs to know whether the batches should be shuffled, how many workers (CPU cores) can be used for preprocessing and how many samples should be included in one batch. Furthermore, a sampler can be passed to the data loader. The sampler determines in what order the samples are drawn from the dataset. We use a random sampler, which draws the samples in a random fashion which ensures the batches shown in training are shuffled. The data loader makes sure that there are always enough preprocessed images during training but also allows the images to be preprocessed on demand and whilst the GPU is already training on a previous batch. This speeds up the process by distributing the workload and by neglecting the need of preliminary preprocessing of the whole dataset before training.

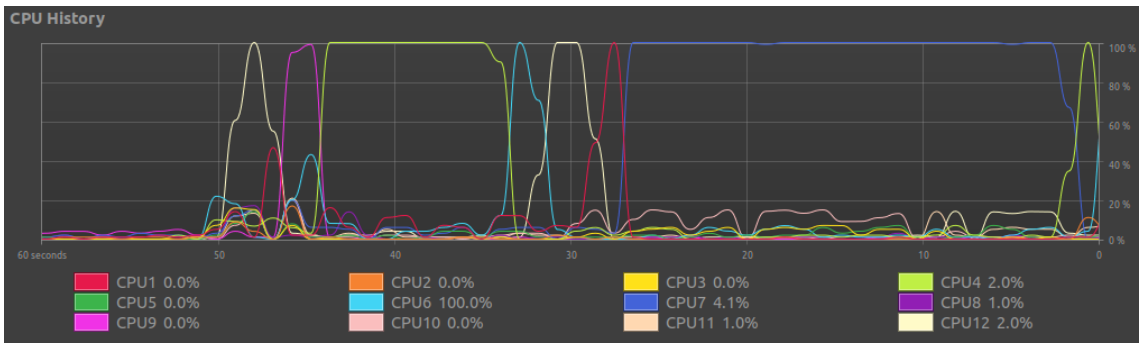



**Figure 5.1**: CPU thread utilisation when training and using a data loader with one worker.

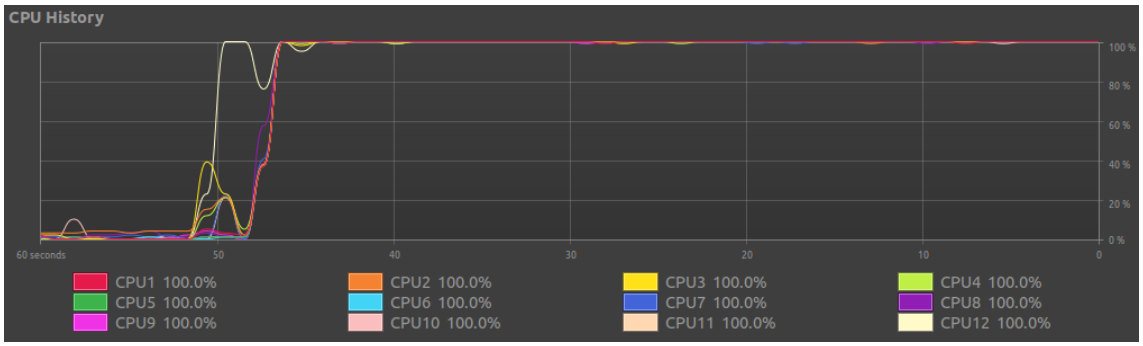



**Figure 5.2**: CPU thread utilisation when training and using a data loader with multiple workers.

## 5.3 Results

To test whether the data is preprocessed on multiple threads in parallel, we ran two training cycles and recorded the CPU thread utilization. We set up the data loader to only have one worker (=one thread) in the first run. The resulting utilization is shown in Figure 5.1. The utilization charts show that even though the active thread changes sometimes, there is never more than one thread active at a certain point in time. In contrast, when running a training cycle with a data loader configured to utilize more than one thread, Figure 5.2 shows that the load from preprocessing the images is distributed over all threads. In both experiments, background tasks were reduced to a minimum. Furthermore, we also measured the GPU utilization during these experiments. When using one worker, the GPU utilization did not exceed a value of 9%, whereas when using 8 workers, the GPU was also fully utilized. These numbers may vary on different systems.

## 5.4 Conclusion

Running these two experiments and recording CPU and GPU utilization, we can show that the preprocessing runs in parallel and helps to fully utilize the computing power of the GPU, which speeds up the training process. The right amount of workers depends on the processing power of each CPU thread and the GPU's total performance.

# Chapter 6

# Milestone 4

## 6.1 Motivation

The goal of milestone 4 was to create a more stable model than the ResNet-51-S under real-world conditions. We combined the previous milestones (1-3) and called the resulting network the AFFACT model.

## 6.2 Methodology

We adjusted the data loader of the ResNet-51-S model such that it combined the data augmentation and generation of randomly perturbed images in parallel from milestone 2 and 3. In each training epoch, the pictures were augmented differently to improve the robustness of the network.

## 6.3 Results

This section discusses the quantitative and qualitative analysis of the AFFACT model.

### 6.3.1 Quantitative Analysis

Analogous to the quantitative analysis in milestone 1, we analyze the AFFACT model's accuracy on the four test sets (A, C, D, T) and compare it against a baseline and the ResNet-51-S. The baseline model is always predicting the majority class of each attribute, as it was seen in training. For example, if most people in the dataset are male, the baseline model predicts each person shown to be male, regardless of the actual input.

Figure 6.1 shows the overall error rate of the AFFACT model, the error rate per attribute on all test sets, and the error rate of the baseline model. The AFFACT model improved its performance on all attributes and on all test sets compared to the baseline model, which was to be expected. Furthermore, the error rates per attribute show less variance across the different test sets, as is the case in the ResNet-51-S model. The AFFACT model shows consistent performance on all attributes, regardless of the test set, which means it less susceptible to disturbances and, therefore, more stable under real-world conditions.

The overall error rate of the AFFACT model on the aligned test set (A) is 8.21%, only slightly higher than the error rate of the ResNet-51-S model (8.15%), which was trained on aligned images.

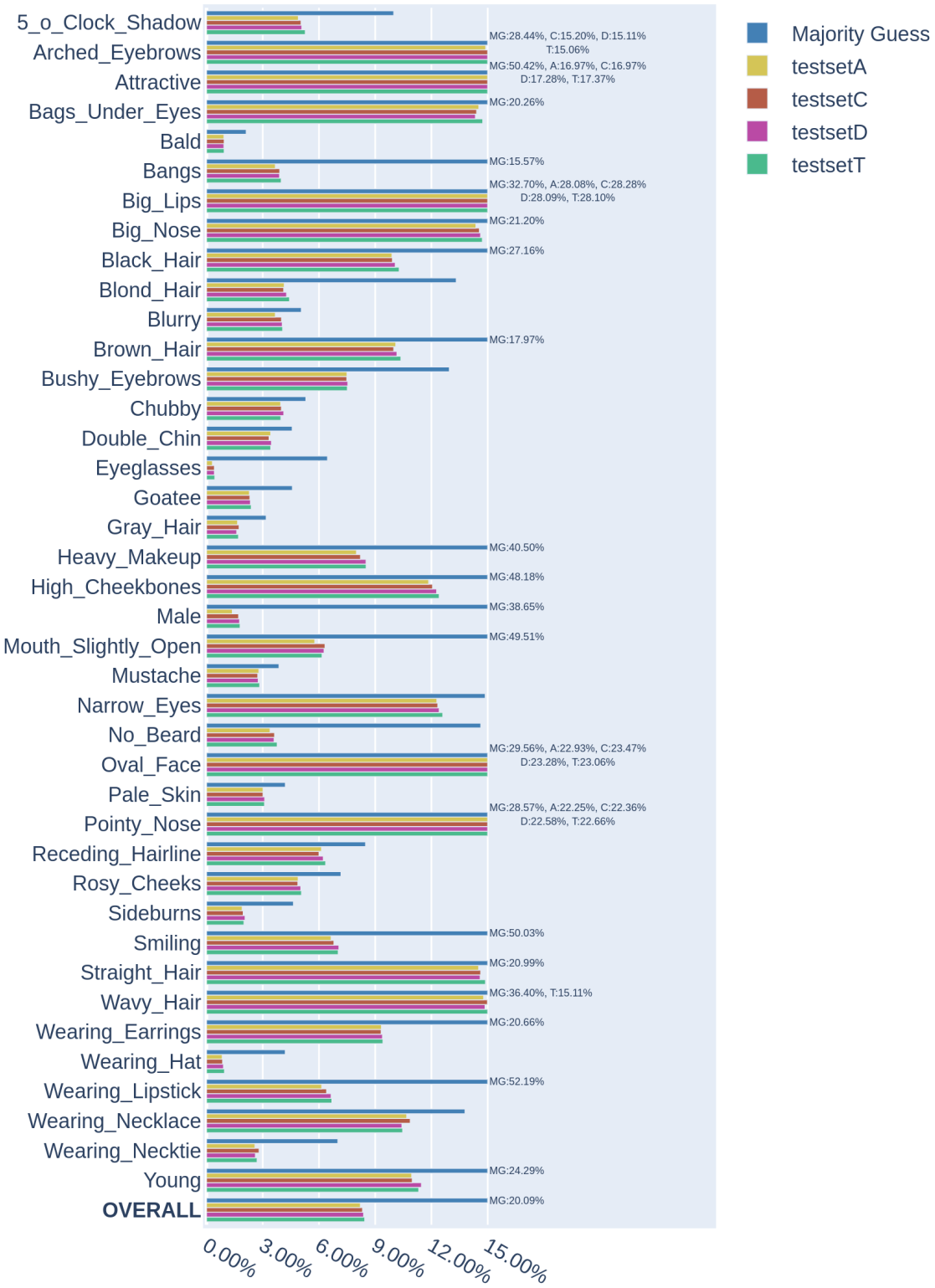



**Figure 6.1**: Attribute-wise accuracy of the AFFACT model on the different test sets. The error rates are cut off if the value exceeds 15%. In that case, the true error rate is annotated next to it.

On the other test sets, the AFFACT model achieved error rates of 8.32% (test set C), 8.38% (test set D), and 8.44% (test set T). This is a notable improvement compared to the ResNet-51-S model, especially on test set C with a relative performance improvement of 26.32% and on test set T with a relative performance improvement of 55.69%.

In general, the AFFACT model seems to perform better on attributes that are less subject to interpretation (e.g. wearing a hat or wearing eyeglasses) than on attributes that leave some room for interpretation (e.g. curly hair, attractive, pointy nose).

### 6.3.2 Qualitative Analysis

Similarly to the qualitative analysis for the ResNet-51-S model, we want to look at the same six images and manually assess the quality of the AFFACT model's prediction. This time, the images are taken from the transformed test set (T). The six images are shown in Figure 6.2. The related table with the attribute-wise predictions can be found in the appendix (Figure A.2). In general, predicting attributes on the images might be harder, since due to the transformations, the attribute's location is not standardized anymore. This effect is welcome since it improves the model's stability under real-world conditions. Transformations can also have a negative impact, e.g. in image 3 the contrast of the final image is so low that specific attributes are hard to identify even for us humans correctly. On picture 5, the prediction quality decreased significantly, which might be explained by the red bar produced by the extrapolation process of the `bob.ip.base` library [6].

## 6.4 Conclusion

The reported relative decrement of the error rate on test sets C, D and T indicates that the network is more robust under real-world conditions. The data augmentation from the AFFACT paper seems to fulfill its purpose and makes the network more robust toward misalignment. Also, the real accuracy might be higher than the numbers shown in the qualitative analysis due to wrongly labeled attributes. The number of wrongly labeled attributes is expected to contribute at least 25% to the reported error rate [18].

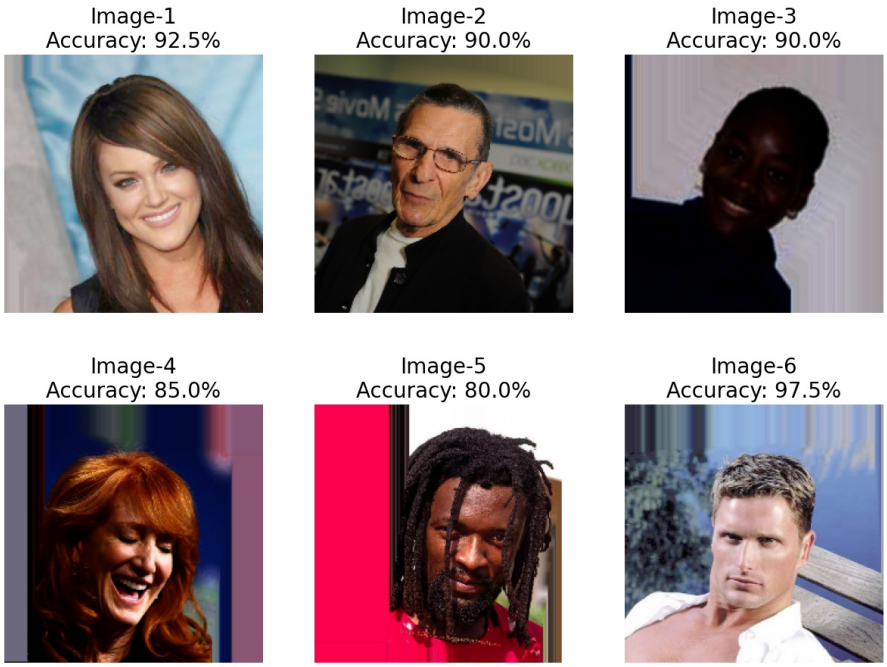



**Figure 6.2**: Sample images with prediction accuracy over all attributes. Images are taken from the test set T.

# Chapter 7

# Additional Work

The following sections describe the additional work we did when converting the AFFACT model from Caffe to PyTorch. We mainly focused on improving the model's performance and automating different processes.

## 7.1 Bounding Box Detector

To generate the test sets C, D, and T, we used the MTCNN [4] face detector to predict bounding boxes that were used to crop the images. As the bounding boxes detected are too narrow, we scaled them by a factor of 2. Based on these newly calculated bounding boxes, we cropped the images used in the test sets. If the face detector could not find a face on an image, we used the bounding box from the CelebA dataset instead. This was the case for 45 out of 19'962 images.

## 7.2 Weights and Biases

WandB (Weights and Biases) is a popular tool to keep track of machine learning experiments [12]. In contrast to ordinary software projects, machine learning projects require more experimentation [2]. Furthermore, deep learning code usually throws fewer errors but underfits or overfits. Hence, debugging is more difficult, and time-consuming [2]. Additionally, training a neural network requires a lot of hyperparameters. Tuning these hyperparameters is essential in deep learning and reproducing earlier work often requires an exact match [2]. WandB organizes the experiments in a more meaningful way than just using a plain document. Since machine learning is computationally expensive, the use of WandB is a more cost-efficient way to run experiments. Big companies such as OpenAI use WandB to track their current experiments and make discussions about future changes.

In our project, we allow the use of WandB in the training configuration file. When enabled, each experiment creates a report containing the training and validation loss/accuracy. WandB then monitors the training progress, and after every epoch, the report is automatically updated. Moreover, system metrics such as memory, CPU, GPU usage are recorded and updated on the fly. Additionally, WandB integrates very well with our source code structure. We can easily keep track of the current neural network architecture as well as the values for the hyperparameters. WandB has excellent tools that facilitate the comparison of results across experiments. Using WandB, we were able to conduct hyperparameter optimization efficiently.

## 7.3 Automated Re-Initialization During Training

The models of the AFFACT paper were trained in multiple training runs. After each run, the model was re-initialized with the weights of the previous run and then trained again with a smaller learning rate. We automate this process by using the `ReduceLROnPlateau` learning rate scheduler of PyTorch [9]. This scheduler is configured to track the validation loss during the training process. Furthermore, a patience factor is defined. Suppose the model's performance (measured by the validation loss) during training does not improve for the number of epochs defined by the patience factor. In that case, the learning rate is multiplied by another factor, called the gamma factor. Additionally, we had to implement a hook. Whenever the scheduler reduces the learning rate, re-initializes the model with the weights of the previously best performing model of the same training run. We used a patience factor of 2 epochs and a gamma of 0.1 in our training process since this resembled the training process of the AFFACT paper the closest. For further experimentation, the patience factor could be increased.

## 7.4 Extending The AFFACT Network

We tried to improve the performance of the AFFACT network by extending or modifying the networks' architecture. Since the publication of the AFFACT paper, the ResNet model family has grown. There is now a ResNet-152 available, which outperforms the ResNet-50 model by 2.16% on the top-1 error and by 1.19% on the top-5 error [14]. Therefore, we replaced the pre-trained ResNet-50 part in the ResNet-51-S with the newer pre-trained ResNet-152. The additional fully connected layer with 40 nodes and the sigmoid activation function were retained. We call this model ResNet-153-S. We hoped by changing the underlying architecture to a model that performs better in image recognition tasks; we can also improve the performance in alignment-free facial attribute classification. Unfortunately, this modification did not yield the performance increase we expected.

As a next step, we tried to improve the AFFACT model by modifying its underlying ResNet-51-S model. We ran many experiments with dropout layers, different dropout rations, different learning rates, learning rate decays, additional fully connected layers, and another activation function for the fully connected layers. We found that with the added two layers (with 40 and 100 nodes), dropout does not improve the model's performance as the data augmentation provides enough variety to prevent overfitting. Instead of using a rectified linear unit activation function (ReLU) we used a leaky ReLU for the last two fully connected layers. A leaky ReLU allows a small positive gradient if a node is not active and therefore prevents a loss of signal after training for a few epochs. We used the leaky ReLU since we observed the loss not improving significantly after training a while. We still were not able to find a model that performed better than the original AFFACT model. The final model that we decided to go forward with was the ResNet-51-S model with one additional fully connected layer with 40 nodes. We call this model AFFACT-Ext.

## 7.5 Hyperparameter Optimization

In deep learning, a model's performance heavily depends on the hyperparameters that were chosen during training. During our project, we observed that even the choice of the batch size (e.g. 16 vs. 32) yields a performance difference of up to 2%. Hyperparameters, in general, accept categorical values or continuous numbers. In the case of numerical values, they can sometimes differ by a factor of 10 000 (e.g. learning rate of $10^{-1}$ vs. $10^{-5}$) and still yield acceptable results, given the choice of other hyperparameters is fitting. Optimizing these hyperparameters in an isolated

fashion is impossible since they often impact each other. Furthermore, trying to test all (or a lot) of these combinations is a time-consuming task since there are endless combinations when considering the ranges of valid values.

In the process of improving the AFFACT model, we also considered optimizing the hyperparameters since they heavily impact the models' performance. To do so, we used WandB, which allows running automated hyperparameter optimization. For each hyperparameter we wanted to optimize, we either had to specify distinct values (categorical) or the range (minimum and maximum) we wanted to test. We could further specify whether we want the values for each hyperparameter chosen each run randomly, or if we want the hyperparameters chosen by applying statistics of past runs and calculating the probability of improvement using a Gaussian process and Bayesian optimization [15]. We executed multiple optimization runs using both strategies. Besides optimizing standard hyperparameters, we also provided the option to change the underlying model architecture since the goal was to find the best model overall, regardless of the model's architecture. The hyperparameters we wanted to optimize were the following:

- The underlying model architecture: ResNet-51-S, AFFACT-Ext, ResNet-153-S
- Batch size: 8, 16, 32, 64 (we chose these sizes since they are a power of two's and still fit on the GPU using the largest model (ResNet-153-S))
- Initial learning rate: $10^{-5}$ - $10^{0}$
- The factor by which the learning rate is multiplied after a certain amount of steps: 0.01 - 0.5
- Momentum: 0 - 1
- Optimizer Type: SGD, Adam, RMSProp
- Number of epochs to train: 5 - 15

Figure 7.1 shows an overview chart of a hyperparameter optimization run. In total, we executed 4 runs with a total of 335 different models trained. Unfortunately, we were not able to achieve significantly better results than the original AFFACT model.

## 7.6 Color-Temperature Transformation

Since images in real-world conditions do not always have the same white balance, we want to account for this by adding a color-temperature transformation to the data augmentation pipeline. To shift the color-temperature of an image, we multiplied the pixels of each color channel with a conversion matrix that contained values according to predefined Kelvin temperatures. An example of such transformations is shown in Figure 7.2.

We could not improve the model's performance by enabling this augmentation during the training process, which is to be expected since the test sets do not contain images with the same transformation. Further, the performance did not decrease even though the training conditions are more difficult due to the additional transformation. We created another test set based on test set T, which also contains the color-shift transformation. The original AFFACT, which is not trained on this transformation, performed as well as our AFFACT-Ext model trained on the color-shift transformation. We, therefore, conclude that the images with different white balances do not negatively impact facial attribute classification performance.

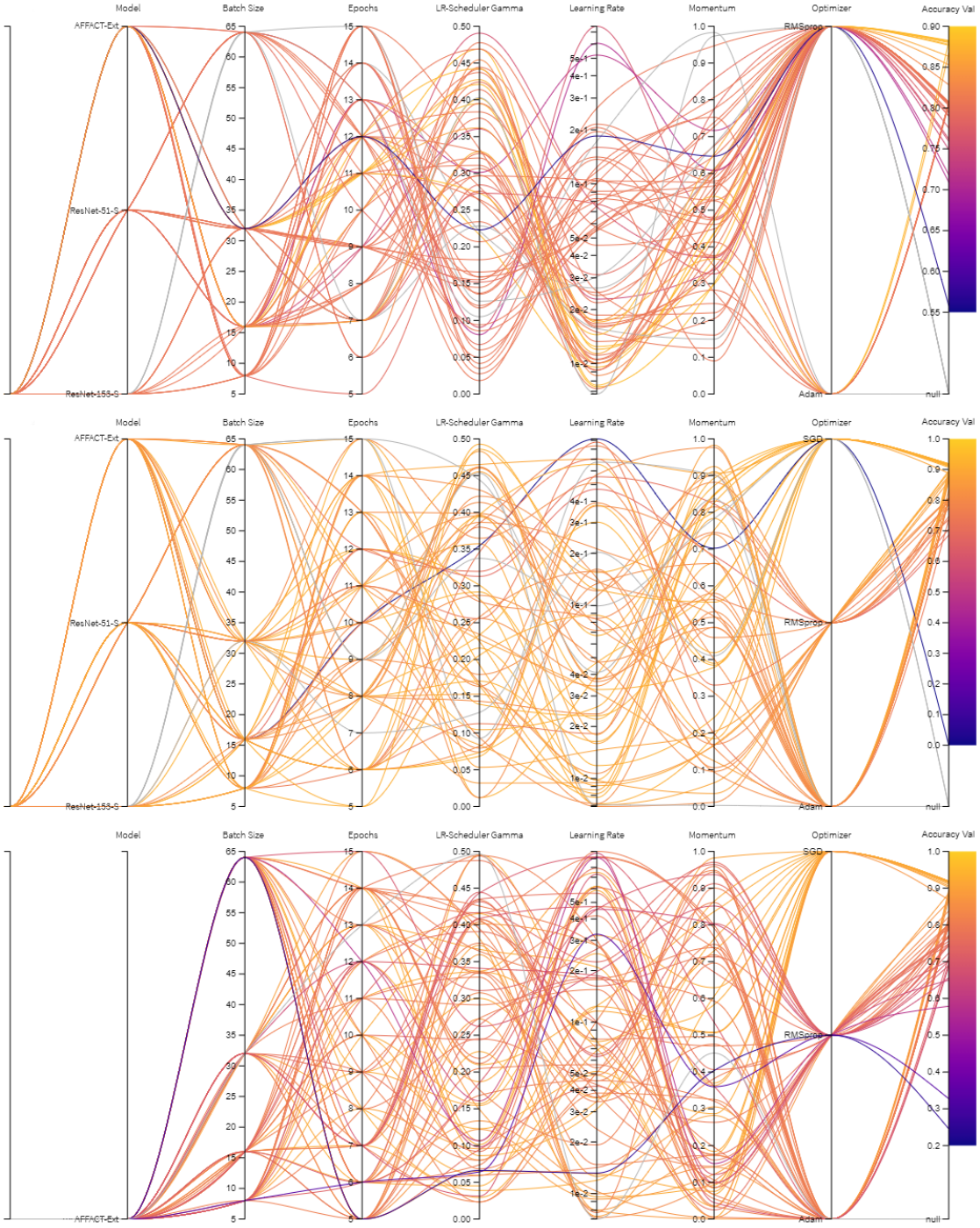



**Figure 7.1**: Hyperparameter optimization performance chart of three different runs

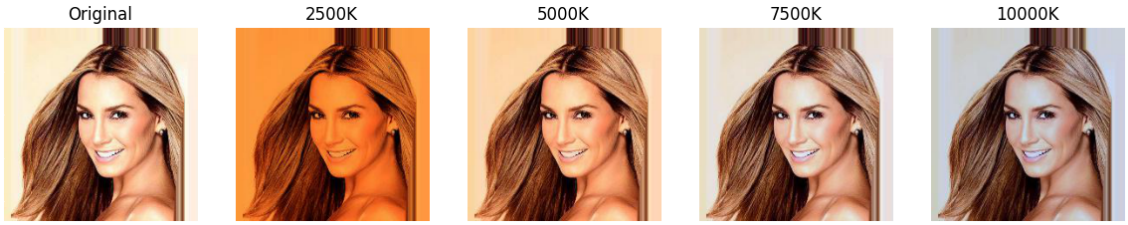



**Figure 7.2**: Original image compared to transformed images with different color-temperatures

## 7.7 Comparison of All Models

Table 7.1 shows the error rate of all models (ResNet-51-S, AFFACT, AFFACT-Ext) on all test sets (A, C, D, T). The ResNet-51-S model performed worse on test sets C, D and T. Only on test set A, the ResNet-51-S achieves slightly better results than the AFFACT model and the AFFACT-Ext model. The AFFACT and AFFACT-Ext model perform similarly across all test sets. Interestingly, the AFFACT model archieves a lower error rate for test set D, although the AFFACT-Ext model has a lower error rate for 22 out of 40 attributes. On test set T, the AFFACT-Ext model performs better by a very low margin of 0.01 and has a lower error rate on 60% of the attributes.

Overall we conclude that the authors of the AFFACT paper did an excellent job at choosing an appropriate architecture for the deep learning model [18]. The hyperparameter optimization could not significantly boost the performance. To further decrease the error rate, the three models could be combined into a model ensemble to make a prediction.

| | ResNet-51-S | AFFACT | AFFACT-Ext | ResNet-51-S | AFFACT | AFFACT-Ext | ResNet-51-S | AFFACT | AFFACT-Ext | ResNet-51-S | AFFACT | AFFACT-Ext |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | A | | | C | | | D | | | T | | |
| 5_o_Clock_Shadow | 4.95 | 4.90 | **4.87** | 8.05 | 5.04 | **4.97** | 5.34 | 5.08 | **4.99** | 7.45 | 5.27 | **5.13** |
| Arched_Eyebrows | 14.94 | **14.89** | 15.10 | 17.82 | **15.20** | 15.32 | 18.02 | **15.11** | 15.37 | 21.52 | **15.06** | 15.29 |
| Attractive | 17.16 | 16.97 | **16.76** | 22.26 | 16.97 | **16.80** | 19.41 | 17.28 | **17.20** | 32.59 | 17.37 | **17.19** |
| Bags_Under_Eyes | **13.97** | 14.53 | 14.56 | 17.35 | **14.41** | 14.62 | 16.30 | 14.35 | **14.38** | 18.09 | 14.74 | **14.73** |
| Bald | 0.91 | 0.93 | **0.88** | 1.19 | 0.95 | **0.90** | 0.95 | 0.93 | **0.92** | 1.20 | 0.95 | **0.93** |
| Bangs | 3.89 | 3.67 | **3.66** | 5.98 | 3.91 | **3.89** | 4.60 | **3.90** | 3.99 | 7.51 | 3.99 | **3.96** |
| Big_Lips | **27.92** | 28.08 | 28.14 | 29.97 | **28.28** | 28.46 | 29.17 | **28.09** | 28.18 | 30.26 | **28.10** | 28.18 |
| Big_Nose | **14.10** | 14.37 | 14.70 | 16.49 | **14.56** | 14.65 | 15.42 | 14.62 | **14.60** | 18.17 | **14.72** | 14.74 |
| Black_Hair | 10.04 | **9.89** | 10.02 | 10.20 | **9.92** | 10.12 | **10.03** | 10.07 | 10.19 | 19.47 | **10.28** | 10.30 |
| Blond_Hair | **4.07** | 4.15 | 4.10 | 4.30 | 4.12 | **4.08** | **4.27** | **4.27** | 4.30 | 9.21 | 4.43 | **4.41** |
| Blurry | **3.56** | 3.67 | 3.63 | 5.01 | **4.00** | 4.04 | 4.26 | **4.04** | 4.07 | 4.79 | **4.06** | 4.08 |
| Brown_Hair | **9.97** | 10.10 | 10.15 | 10.65 | 9.99 | **9.94** | 10.50 | **10.16** | 10.22 | 15.47 | **10.37** | **10.37** |
| Bushy_Eyebrows | **7.19** | 7.49 | 7.43 | 11.06 | **7.48** | 7.52 | 8.31 | 7.54 | **7.42** | 11.69 | 7.51 | **7.49** |
| Chubby | 4.05 | **3.96** | **3.96** | 4.36 | **4.00** | 4.01 | 4.17 | 4.12 | **4.10** | 4.61 | 3.97 | **3.96** |
| Double_Chin | 3.38 | 3.43 | 3.49 | 3.91 | **3.34** | 3.45 | 3.68 | **3.47** | 3.49 | 4.00 | **3.43** | 3.45 |
| Eyeglasses | 0.28 | 0.32 | 0.31 | 1.85 | **0.42** | 0.43 | 1.01 | 0.42 | **0.39** | 2.68 | 0.45 | **0.43** |
| Goatee | 2.26 | 2.29 | **2.24** | 3.28 | **2.31** | 2.38 | 2.49 | 2.34 | **2.33** | 3.48 | 2.39 | **2.34** |
| Gray_Hair | 1.67 | **1.66** | 1.67 | 1.69 | 1.74 | **1.67** | 1.66 | **1.61** | 1.64 | 2.53 | 1.71 | **1.70** |
| Heavy_Makeup | **7.77** | 8.00 | 7.85 | 10.82 | 8.22 | **8.08** | 9.99 | 8.51 | **8.30** | 21.4 | 8.52 | **8.27** |
| High_Cheekbones | 11.86 | **11.85** | 11.87 | 25.41 | **12.06** | 12.15 | 15.95 | 12.28 | **12.26** | 29.16 | 12.41 | **12.26** |
| Male | 1.51 | **1.38** | 1.46 | 3.14 | **1.71** | 1.79 | 2.32 | **1.76** | 1.86 | 5.14 | **1.79** | 1.91 |
| Mouth_Slightly_Open | **5.68** | 5.78 | 5.93 | 12.2 | 6.32 | **6.27** | 7.69 | **6.27** | 6.30 | 15.33 | **6.17** | 6.33 |
| Mustache | 2.88 | 2.80 | **2.70** | 3.83 | 2.74 | **2.71** | 3.30 | 2.76 | **2.74** | 3.64 | 2.84 | **2.80** |
| Narrow_Eyes | **12.17** | 12.29 | 12.37 | 13.52 | 12.34 | **12.33** | 12.87 | **12.41** | 12.55 | 14.11 | **12.60** | 12.63 |
| No_Beard | 3.50 | **3.39** | 3.44 | 4.97 | 3.63 | **3.51** | 3.91 | 3.61 | **3.60** | 6.43 | 3.77 | **3.65** |
| Oval_Face | **21.97** | 22.93 | 22.45 | 27.43 | 23.47 | **23.09** | 25.05 | 23.28 | **23.00** | 28.51 | 23.06 | **22.83** |
| Pale_Skin | **2.71** | 3.02 | 3.00 | **2.99** | 3.02 | 3.06 | **2.92** | 3.11 | 3.17 | 15.00 | **3.10** | 3.15 |
| Pointy_Nose | 22.33 | **22.25** | 22.47 | 25.15 | **22.36** | 22.51 | 24.07 | **22.58** | **22.58** | 27.08 | **22.66** | 22.70 |
| Receding_Hairline | 6.05 | 6.14 | **5.99** | 6.40 | 6.00 | **5.89** | 6.44 | 6.23 | **6.07** | 7.26 | 6.36 | **6.20** |
| Rosy_Cheeks | **4.55** | 4.89 | 4.93 | 7.12 | **4.87** | 4.88 | 5.64 | 5.03 | **4.94** | 6.81 | 5.07 | **4.94** |
| Sideburns | 1.97 | **1.90** | 1.92 | 2.09 | 1.96 | **1.93** | 2.08 | 2.06 | **1.92** | 2.87 | 2.00 | **1.94** |
| Smiling | 6.75 | **6.65** | **6.65** | 20.19 | **6.80** | 6.88 | 11.98 | 7.06 | **6.99** | 24.14 | 7.03 | **7.02** |
| Straight_Hair | **14.26** | 14.52 | 14.46 | 15.38 | 14.63 | **14.55** | 14.71 | 14.60 | **14.39** | 16.85 | 14.88 | **14.69** |
| Wavy_Hair | 15.30 | **14.78** | 15.04 | 15.96 | **14.99** | 15.33 | 15.12 | **14.86** | 15.33 | 20.02 | **15.11** | 15.35 |
| Wearing_Earrings | **9.19** | 9.33 | 9.42 | 10.67 | **9.32** | 9.34 | 10.39 | **9.39** | 9.45 | 13.43 | 9.41 | **9.40** |
| Wearing_Hat | 0.87 | **0.84** | 0.88 | 0.99 | **0.86** | 0.89 | 0.92 | **0.91** | 0.92 | 1.60 | **0.96** | 0.98 |
| Wearing_Lipstick | 6.20 | **6.15** | **6.15** | 9.31 | 6.41 | **6.38** | 7.94 | 6.65 | **6.61** | 19.14 | 6.69 | **6.64** |
| Wearing_Necklace | **10.61** | 10.68 | 10.69 | 11.73 | **10.87** | 11.06 | 10.75 | **10.42** | 10.60 | 11.92 | **10.46** | 10.68 |
| Wearing_Necktie | 2.60 | **2.58** | 2.60 | 3.27 | **2.81** | 2.82 | 2.71 | **2.61** | 2.64 | 4.14 | **2.70** | 2.71 |
| Young | 11.05 | **10.95** | 11.10 | 12.43 | **10.98** | 11.03 | 12.34 | 11.47 | **11.45** | 16.82 | **11.32** | 11.45 |
| TOTAL | **8.15** | 8.21 | 8.23 | 10.51 | **8.32** | 8.34 | 9.22 | **8.38** | 8.39 | 13.14 | 8.44 | **8.43** |

**Table 7.1**: Table comparing the error rates of all our models on all the test sets.

# Chapter 8

# Conclusion

In this master project we reimplemented the model produced by the AFFACT paper and achieved very similar results [18]. To do so, we had to reimplement the AFFACT network architecture, reimplement the original AFFACT transformations, and ensure parallel preprocessing of the images. While reusing the already pretrained ResNet-50 and parallel preprocessing of the images was straightforward to implement, understanding and implementing the data augmentation was a more sophisticated task. Reproducing similar results turned out the be more difficult than expected. Many small details had quite a big effect on the final result. E.g. we saved the images of the test sets in a JPG format instead of using the PNG format, which contributed to a 1% higher overall error rate. Also, small adjustments of hyperparameters (e.g. batch size 64 vs 512) have a big impact on the model's final prediction accuracy. We tried to improve the AFFACT model by extending and modifying it, adding a face detector for dataset generation a color shift transformation, and implementing automated hyperparameter optimization. We further propose a model that achieves similar scores across all test sets and is also trained on the color-temperature transformation.

We have shown that the AFFACT model is more robust to misalignment and hence is more suitable for real-world scenarios. The AFFACT transformations improved the error rate on the test sets C, D, and T while maintaining the high accuracy on test set A. We think it is almost impossible to achieve much better results since there are many wrongly classified attributes in the CelebA dataset [18]. As the error rate with approx. 8% is already very low, wrongly labeled data might prevent further improvement.

It would be interesting to visualize the convolutions for different images during evaluation to see how good the convolutional layers are at removing unnecessary noise from the input image. This way, the hyperparameters for the data augmentation could be optimized. Furthermore, the data augmentation proposed in the AFFACT paper [18] could be applied in other image processing related tasks.

Appendix A

# Attachements

| Attribute | Image-1 | Image-2 | Image-3 | Image-4 | Image-5 | Image-6 |
|---|---|---|---|---|---|---|
| 5_o_Clock_Shadow | no | no | no | no | no | yes |
| Arched_Eyebrows | no | no | yes | no | no | no |
| Attractive | yes | no | no | yes | no | no |
| Bags_Under_Eyes | no | no | no | no | no | no |
| Bald | no | no | no | no | no | no |
| Bangs | yes | no | no | yes | no | no |
| Big_Lips | no | no | yes | no | no | no |
| Big_Nose | no | yes | no | no | yes | no |
| Black_Hair | no | no | yes | no | no | no |
| Blond_Hair | no | no | no | no | no | no |
| Blurry | no | no | yes | no | no | no |
| Brown_Hair | yes | no | no | no | no | no |
| Bushy_Eyebrows | no | no | no | no | no | no |
| Chubby | no | no | no | no | yes | no |
| Double_Chin | no | no | no | no | no | no |
| Eyeglasses | no | yes | no | no | no | no |
| Goatee | no | no | no | no | yes | no |
| Gray_Hair | no | no | no | no | no | no |
| Heavy_Makeup | yes | no | no | yes | no | no |
| High_Cheekbones | yes | no | yes | yes | yes | no |
| Male | no | yes | no | no | yes | yes |
| Mouth_Slightly_Open | yes | no | yes | yes | no | no |
| Mustache | no | no | no | no | no | no |
| Narrow_Eyes | no | no | no | yes | no | no |
| No_Beard | yes | yes | yes | yes | no | yes |
| Oval_Face | yes | no | yes | no | no | no |
| Pale_Skin | no | no | no | no | no | no |
| Pointy_Nose | no | no | no | no | no | no |
| Receding_Hairline | no | no | yes | no | no | no |
| Rosy_Cheeks | no | no | no | no | no | no |
| Sideburns | no | no | no | no | yes | no |
| Smiling | yes | no | yes | yes | no | no |
| Straight_Hair | no | no | no | no | no | no |
| Wavy_Hair | no | no | no | yes | yes | no |
| Wearing_Earrings | no | no | no | no | no | no |
| Wearing_Hat | no | no | no | no | no | no |
| Wearing_Lipstick | yes | no | no | yes | no | no |
| Wearing_Necklace | no | no | no | no | no | no |
| Wearing_Necktie | no | yes | no | no | no | no |
| Young | yes | no | yes | yes | no | yes |

**Figure A.1**: Accuracy Table related to the sample images of the baseline experiment from Milestone 1.

| Attribute | Image-1 | Image-2 | Image-3 | Image-4 | Image-5 | Image-6 |
|---|---|---|---|---|---|---|
| 5_o_Clock_Shadow | no | no | no | no | no | yes |
| Arched_Eyebrows | yes | no | no | no | no | no |
| Attractive | yes | no | no | no | no | no |
| Bags_Under_Eyes | no | yes | no | yes | no | no |
| Bald | no | no | no | no | no | no |
| Bangs | yes | no | no | yes | no | no |
| Big_Lips | no | no | yes | no | no | no |
| Big_Nose | no | yes | yes | no | yes | no |
| Black_Hair | no | no | yes | no | no | no |
| Blond_Hair | no | no | no | no | no | no |
| Blurry | no | no | no | no | no | no |
| Brown_Hair | yes | no | no | no | no | no |
| Bushy_Eyebrows | no | no | no | no | no | no |
| Chubby | no | no | no | no | yes | no |
| Double_Chin | no | no | no | no | no | no |
| Eyeglasses | no | yes | no | no | no | no |
| Goatee | no | no | no | no | no | no |
| Gray_Hair | no | no | no | no | no | no |
| Heavy_Makeup | yes | no | no | no | no | no |
| High_Cheekbones | yes | no | yes | yes | no | no |
| Male | no | yes | no | no | yes | yes |
| Mouth_Slightly_Open | yes | no | yes | yes | yes | no |
| Mustache | no | no | no | no | no | no |
| Narrow_Eyes | no | no | no | yes | no | no |
| No_Beard | yes | yes | yes | yes | no | yes |
| Oval_Face | yes | no | yes | no | no | no |
| Pale_Skin | no | no | no | no | no | no |
| Pointy_Nose | no | no | no | no | no | no |
| Receding_Hairline | no | no | yes | no | no | no |
| Rosy_Cheeks | no | no | no | no | no | no |
| Sideburns | no | no | no | no | yes | no |
| Smiling | yes | no | yes | yes | no | no |
| Straight_Hair | no | no | no | no | no | no |
| Wavy_Hair | no | no | no | yes | yes | no |
| Wearing_Earrings | no | no | no | no | no | no |
| Wearing_Hat | no | no | no | no | no | no |
| Wearing_Lipstick | yes | no | no | no | no | no |
| Wearing_Necklace | no | no | no | no | no | no |
| Wearing_Necktie | no | yes | no | no | no | no |
| Young | yes | no | yes | no | no | yes |

**Figure A.2**: Accuracy Table related to the sample images of the AFFACT experiment from Milestone 4.

# Bibliography


[1] Bcewithlogitsloss - pytorch 1.7.0 documentation. https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html, Nov. 2020.

[2] Machine learning experiment tracking. https://towardsdatascience.com/machine-learning-experiment-tracking-93b796e501b0, Nov. 2020.

[3] Multi-label classification with deep learning. https://machinelearningmastery.com/multi-label-classification-with-deep-learning/, Nov. 2020.

[4] Pretrained pytorch face detection (mtcnn) and recognition (inceptionresnet) models. https://github.com/timesler/facenet-pytorch/, Nov. 2020.

[5] Python api - bob.ip.base 2.2.6 documentation. https://www.idiap.ch/software/bob/docs/bob/bob.ip.base/stable/py_api.html#bob.ip.base.GeomNorm, Nov. 2020.

[6] Python api - bob.ip.base 2.2.6 documentation. https://www.idiap.ch/software/bob/docs/bob/bob.ip.base/stable/py_api.html#bob.ip.base.extrapolate_mask, Nov. 2020.

[7] torch.optim - pytorch 1.7.0 documentation. https://pytorch.org/docs/stable/optim.html?highlight=rmsprop#torch.optim.RMSprop, Dec. 2020.

[8] torch.utils.data - pytorch 1.7.0 documentation. https://pytorch.org/docs/stable/data.html, Nov. 2020.

[9] torchvision.optim.lr_scheduler.reducelronplateau - pytorch 1.7.0 documentation. https://pytorch.org/docs/stable/optim.html#torch.optim.lr_scheduler.ReduceLROnPlateau, Nov. 2020.

[10] torchvision.transforms - pytorch 1.7.0 documentation. https://pytorch.org/docs/stable/torchvision/transforms.html, Nov. 2020.

[11] torchvision.transforms.tencrop - pytorch 1.7.0 documentation. https://pytorch.org/docs/stable/torchvision/transforms.html#torchvision.transforms.TenCrop, Nov. 2020.

[12] Weights and biases - developer tools for machine learning. https://www.wandb.com/, Nov. 2020.

[13] Linear - pytorch 1.7.0 documentation. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear, Jan. 2021.

[14] Resnet | pytorch. https://pytorch.org/hub/pytorch_vision_resnet/, Jan. 2021.

[15] Weights and biases - configuration - documentation. https://docs.wandb.ai/sweeps/configuration, Jan. 2021.

[16] N. Chavannes and Y. Rutishauser. Pytorch implementation of AFFACT. https://github.com/noahch/PyAffact, Jan. 2021.

[17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[18] M. Günther, A. Rozsa, and T. Boult. AFFACT: Alignment-free facial attribute classification technique. In *2017 IEEE International Joint Conference on Biometrics (IJCB)*, pages 90–99, 10 2017.

[19] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[20] Z. Liu, P. Luo, X. Wang, and X. Tang. Deep learning face attributes in the wild. In *Proceedings of International Conference on Computer Vision (ICCV)*, December 2015.

[21] A. Nakamura and T. Harada. Revisiting fine-tuning for few-shot learning. *CoRR*, abs/1910.00216, 2019.

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library, 12 2019.