Faculty of Mathematics and Information Science
WARSAW UNIVERSITY OF TECHNOLOGY

ML Face-Age Prediction

Project documentation and report

Filip Rak, 313128
filip.rak2.stud@pw.edu.pl

Jakub Świstak, 313135
jakub.swistak.stud@pw.edu.pl

Łukasz Zalewski, 329532
lukasz.zalewski8.stud@pw.edu.pl

Mikołaj Zalewski, 306025
mikolaj.zalewski2.stud@pw.edu.pl

Władysław Olejnik,290593
wladyslaw.olejnik.stud@pw.edu.pl

Warsaw, 27th January 2022

Contents

1 Introduction

In the field of computer vision, accurately determining a person's age based on their facial features in a photo is a challenging task. Variations in facial appearance, such as lighting conditions and expressions, make it difficult for computer vision algorithms to consistently and reliably predict age. Additionally, age is not always reflected in a person's facial features in a consistent and reliable manner, making this problem even more complex.

In this project, we aim to tackle the task of estimating ages from facial images using deep learning models implemented in PyTorch. The potential applications of this technology are vast, ranging from security and healthcare to marketing. We also aim to develop web application with Streamlit, which allows users to test our model using either their computer camera or by uploading photos.

2 Technologies

List of libraries used during development:

1. torch <- deep learning framework
2. torchvision <- computer vision extension of pytorch
3. pytorch-lightning <- pytorch wrapper for speeding up the development
4. torchmetrics <- pytorch metrics
5. efficientnet-pytorch <- implementation of EfficientNet
6. opencv-python <- library for processing images
7. mediapipe <- face detection models
8. Pillow <- library for processing images
9. pandas <- used for processing the dataset
10. seaborn <- used for visualizing labels and predictions
11. wandb <- ML metric logging tool
12. streamlit <- library for building data apps
13. pyrootutils <- project root setup
14. tqdm <- displaying progress bar

3 Background knowledge

3.1 Section introduction

The main purpose of this section is to present a theoretical background of the components that were used to build the application.

3.2 Convolutional Neural Networks

3.2.1 Problem description

During development, we tested different flavours of CNN's in order to determine which architecture has the best balance of efficiency and accuracy for our application. These were:

1. Simple CNN - a custom model which we used as a baseline.
2. EfficientNet - a popular state of the art architecture, which has occupied top spots of ImageNet benchmark for the last three years.

The main goal of the project was to use above architectures for predicting age based on the face photo. We decided to express this as a regression problem instead of classification, as we believed we had enough data to train accurate enough regression model, and we thought such approach would simplify the implementation.

3.2.2 Simple CNN architecture

The idea of Convolution is to reduce number of parameters in neural network, thanks to using a sliding window of learnable weights.

Convolutions are translation equivariant - if an object in an image is at area A and through the convolution a feature is detected, then the same feature would be detected when the object in the image is translated to area B.

Stacking many convolutions can result in network learning more complex features with each layer - the first layer might learn the to detect edges and textures, the second layer might encode them into shapes, and so on.

By combining such operation with pooling and fully-connected layers, we can obtain a very efficient architecture.

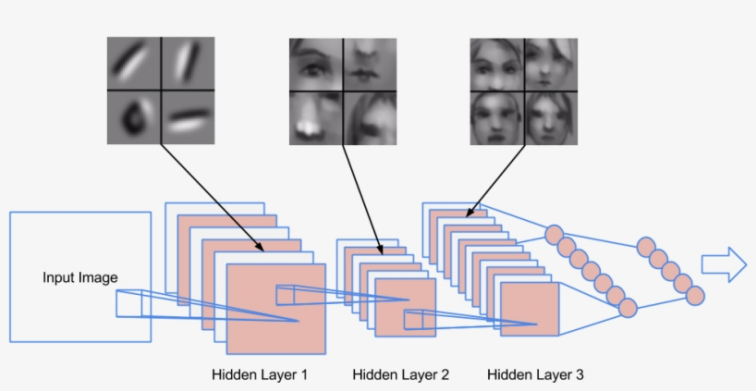


Figure 1: Example CNN.

3.2.3 EfficientNet architecture

EfficientNet is a convolutional neural network architecture and scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coefficient. Unlike conventional practice that arbitrary scales these factors, the EfficientNet scaling method uniformly scales network width, depth, and resolution with a set of fixed scaling coefficients. For example, if we want to use 2^N times more computational resources, then we can simply increase the network depth by α^N, width by β^N, and image size by γ^N, where α, β, γ are constant coefficients determined by a small grid search on the original small model. EfficientNet uses a compound coefficient to uniformly scales network width, depth, and resolution in a principled way.

The compound scaling method is justified by the intuition that if the input image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image.

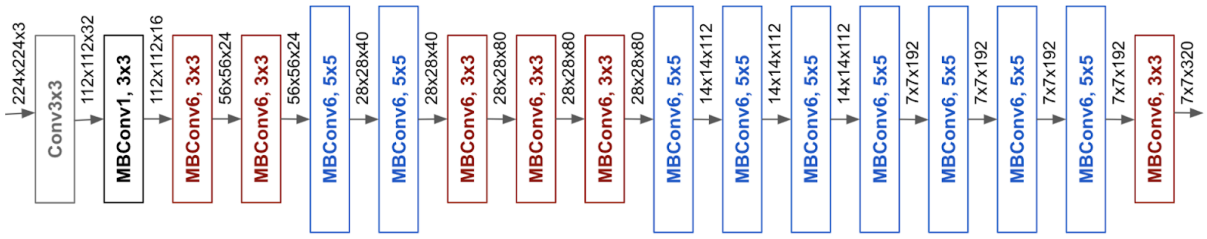


Figure 2: The architecture for baseline network EfficientNet-B0.

4 Data exploration and processing

4.1 Introduction

This section describes the process and presents the results of the data exploration and processing performed on the dataset of images with labels describing the age of the person. The primary purpose of this section is to help understand what the structure of the labels looks like, and what steps had to be taken to prepare data for model training.

4.2 Data exploration

The data set consists of images with integer labels. There are 23708 images in total. The original sizes of images are diverse. The age label values are from range $[1 - 118]$.

The mean value of age in dataset is 33.30.
The standard deviation of age in dataset is 19.89.

Below there is a histogram that visualizes the age distribution.

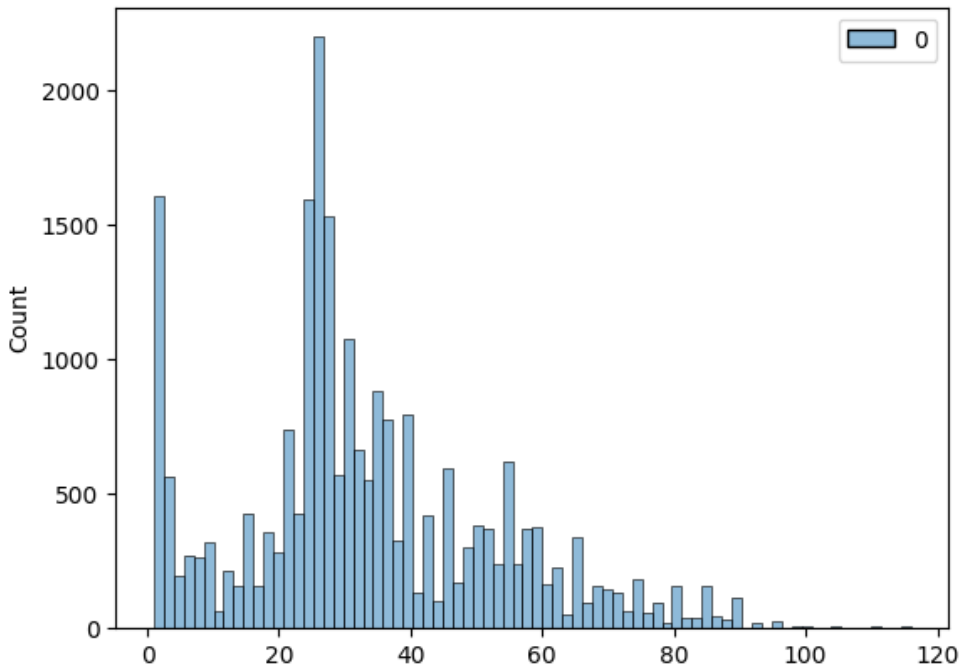


Figure 3: The distribution of the data element's age in the data set

4.3 Data processing

4.3.1 Label clipping and undersampling

The first step was to clip labels to maximum age of 80. We decided to go with that as labels above age 80 are very sparse in the original dataset, and we believe faces of very old people become too similar to evaluate their age.

Next we undersample the dataset to max 500 datapoints per class to make the distribution a little more uniform.

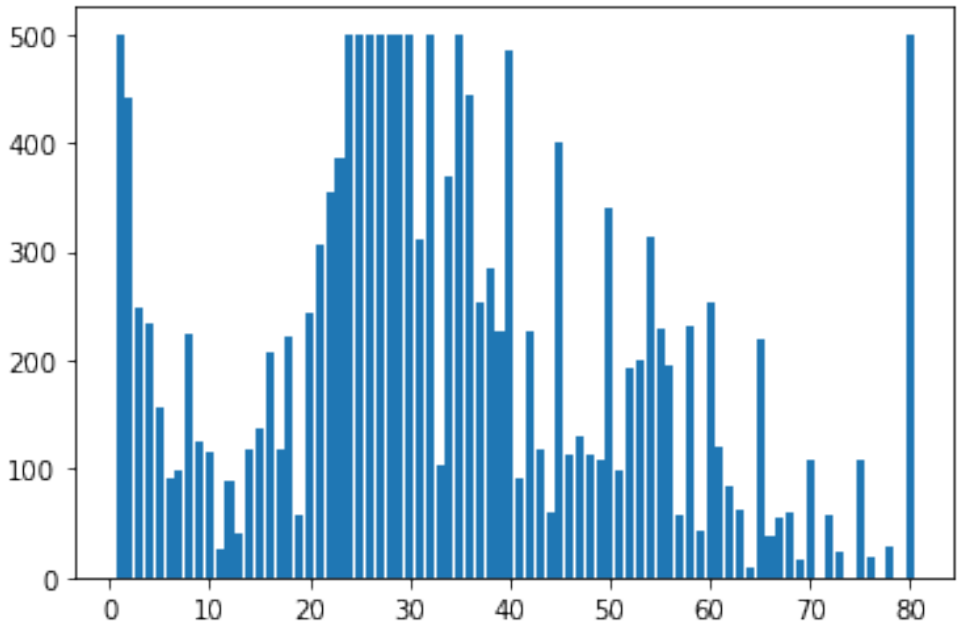


Figure 4: The distribution of the age labels after clipping and undersampling.

4.3.2 Data split

When preparing data to be processed by a model, the first thing is to split it into 3 sets.

The first subset is the training set used to fit the model.
The second subset is the validation set used to evaluate model during training.
The last subset is the test set used for final evaluation (it allows us to verify that model is not overfitting to validation set).

Final sizes:

- Training set - 16535
- Validation set - 1600
- Test set - 1556

This means the validation and test set contain 20 datapoints per each of 80 label classes.

Due to the fact that in some age groups, there were fewer than forty images per class, the test set has a little less images than validation set (1556 vs 1600). This also impacted the number of train set classes which decreased from 80 to 76.

4.3.3 Data augmentation

It should be notices that differences between sizes of each class of the train set vary by a lot. This could result in model overfitting to certain classes.

To prevent that, the augmentation process has been conducted to have exactly 500 images in each age class. The generation process is a process where existing photos are taken and predefined transformations are applied which results in a new image from the perspective of the image's visual properties. It was decided to use the following transformations:

1. rotation - rotates the image by the predefined angle
2. horizontal flip - flips the image horizontally
3. color jittering - changes image's brightness to 0.2, contrast to 0.1, saturation to 0.1, and the hue to 0.2
4. affining - applying translation, scaling, and shearing
5. perspective change - distorting and changing fill property
6. erasing - erasing a randomly selected rectangular region

After defining the set of transformations, the iterative augmentation has been applied. As a result, the train set consists of 76 classes and in each class, there are exactly 500 images (38000 images in total). Below there is a histogram that visualizes the structure of the train set.

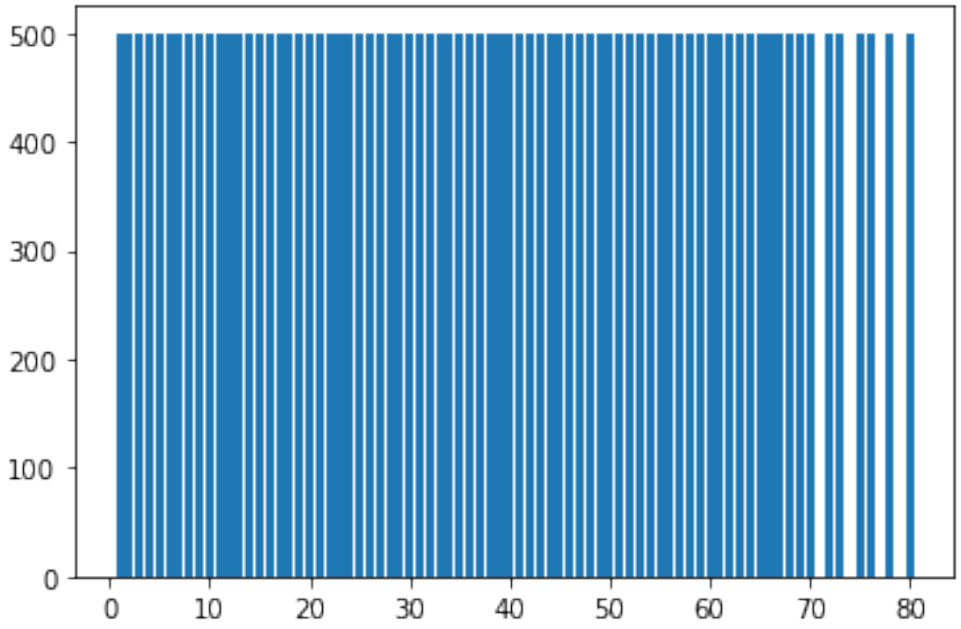


Figure 5: The distribution of labels in the train set after augmentation.

5 Model training and results

5.1 Training setup

We tried 3 different architectures:

- Custom CNN trained from scratch on image sizes of 100x100.
- Custom CNN trained from scratch on image sizes of 224x224.
- EfficientNet (version B0) pretrained on ImageNet and fine-tuned on our dataset, with image sizes of 224x224.

We expressed the learning of age prediction as a regression problem. We applied the MSE loss function, since it's the usually the default cost function used for regression in machine learning field.

As a main metric we use mean average error (MAE), as it seemed to us the most intuitive way for comparing results.

After generating the prediction by the network, we clip it to [0, 1] range and rescale to [1, 80] range for the final calculation of MAE metric.

The best checkpoint is chosen based on the best validation loss achieved during the run. This checkpoint is then used for testing.

We used the following architecture for CNN:

```
class SimpleConvNet_100x100(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(32 * 25 * 25, 256)
        self.fc2 = nn.Linear(256, 1)
```

To maintain comparability, we used the same hyperparameters for all our runs:

- seeds: 42, 43, 44
- num_epochs: 10
- batch_size: 32
- learning_rate: 1e−3
- optimizer: Adam
- checkpointing metric: validation loss

The following chart visualizes progress of training for different architectures.

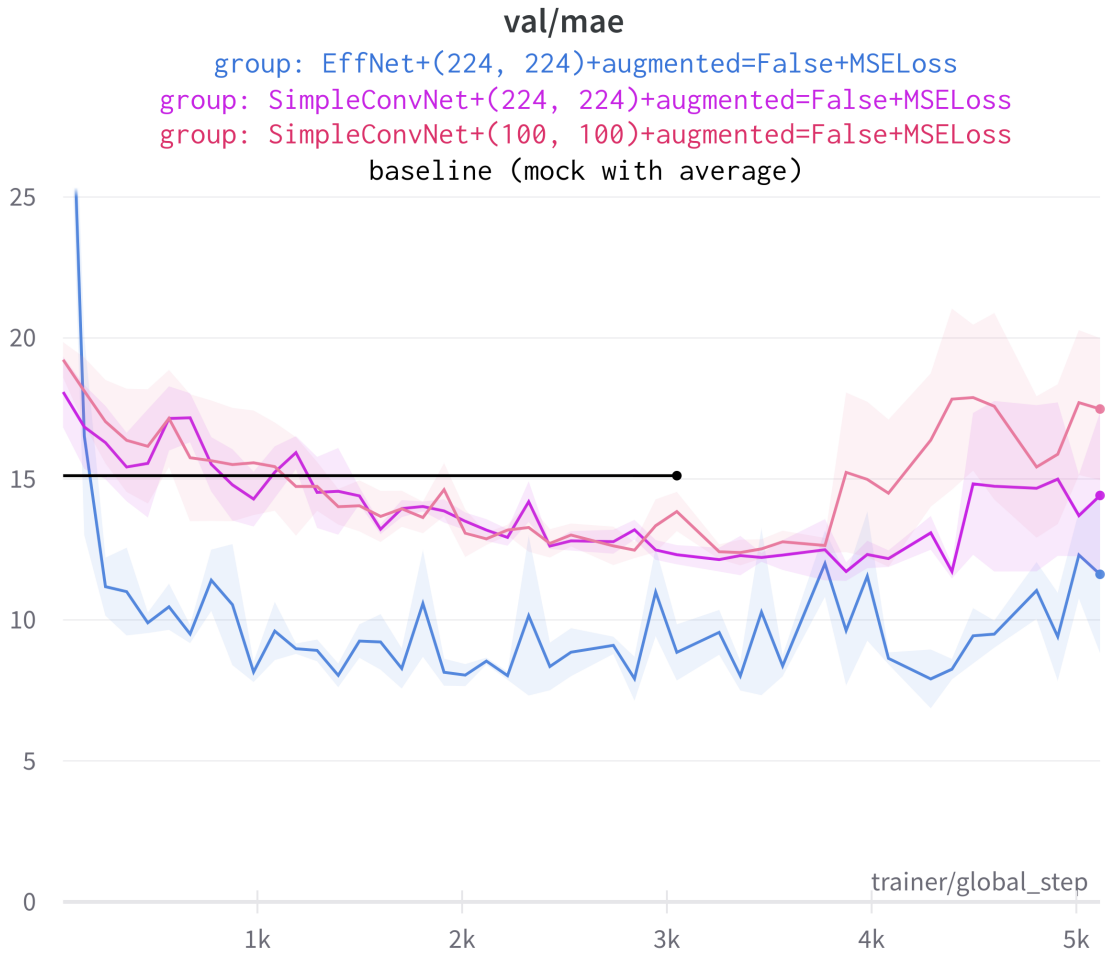


Figure 6: Mean absolute error (MAE) on validation set during training. The chart shows results for CNNs trained from scratch, and EfficientNet pretrained on ImageNet. The black line denotes the MAE achieved when simply mocking model predictions with average of all ages in the dataset. The results have been averaged over 3 seeds. The bright area around each run denotes 1 standard deviation from the mean (calculated from those 3 seeds).

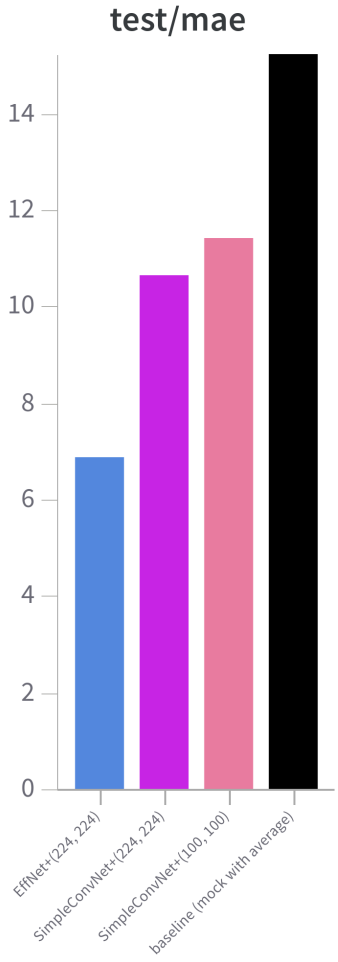


Figure 7: Mean absolute error (MAE) on test set generated by weights which obtained best validation loss on each run. EfficientNet scored the best with 6.896 MAE.

5.2 Distribution of predictions

We took the model weights of EfficientNet, which scored the best on validation set during training and used it to generate and analyse predictions.

Test dataset consists of 1556 photos. The histograms below visualize the structure of predictions and original labels.

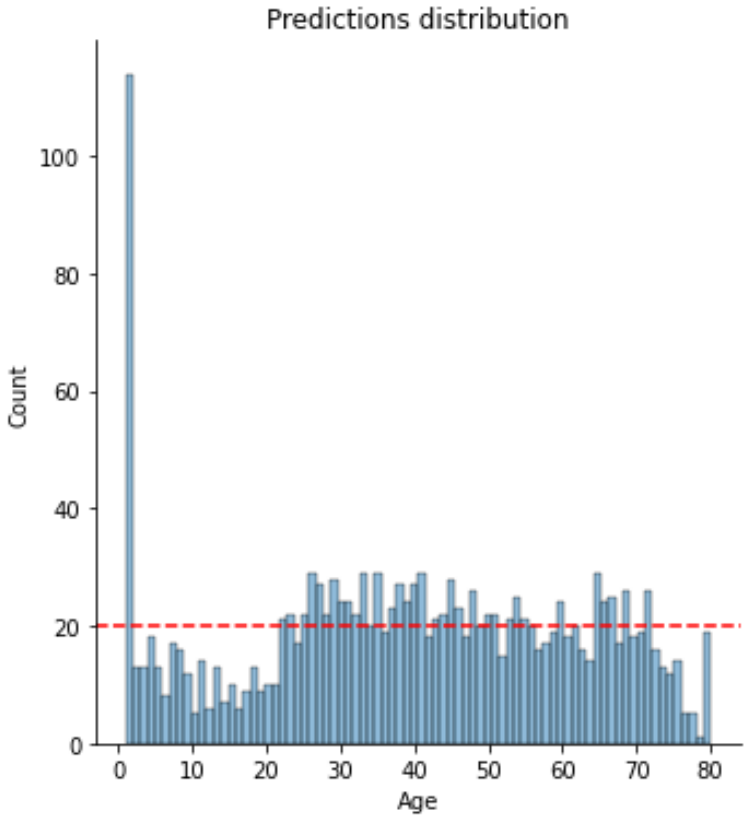


Figure 8: The distribution of age labels predicted by the best model on test set. The red line denotes the original number of labels per each class in test set. Notice the spike on the left side of the histogram - the model learned to be biased towards predicting label "1". We hypothesise distinguishing faces in the age range of 1-20 is hard, which made the model go for a shortcut by learning to predict "1" when it's not confident.

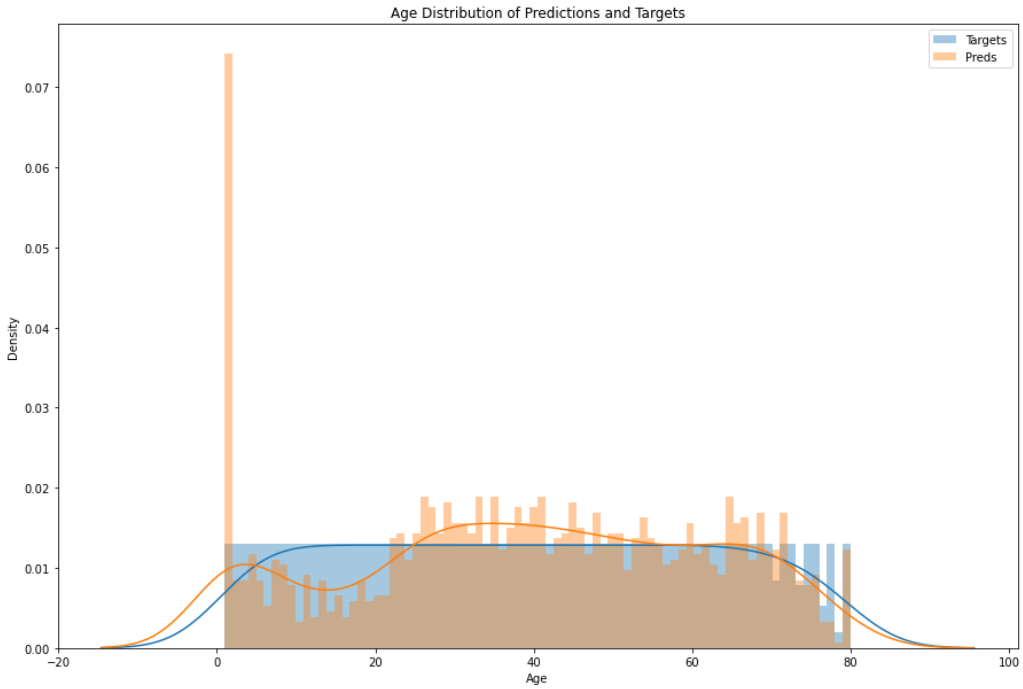


Figure 9: The overlap comparison between distribution of original test set labels and labels predicted by final model.

5.3 Error analysis

The mean average error (MAE) determines, what is the average difference between the real age of the person on the picture and the age obtained from our model. For the test set, the best achieved MAE equals 6.896.

The graph below shows how the error was distributed across age and the table below shows 20 highest errors of the best model, which were found in the test set.

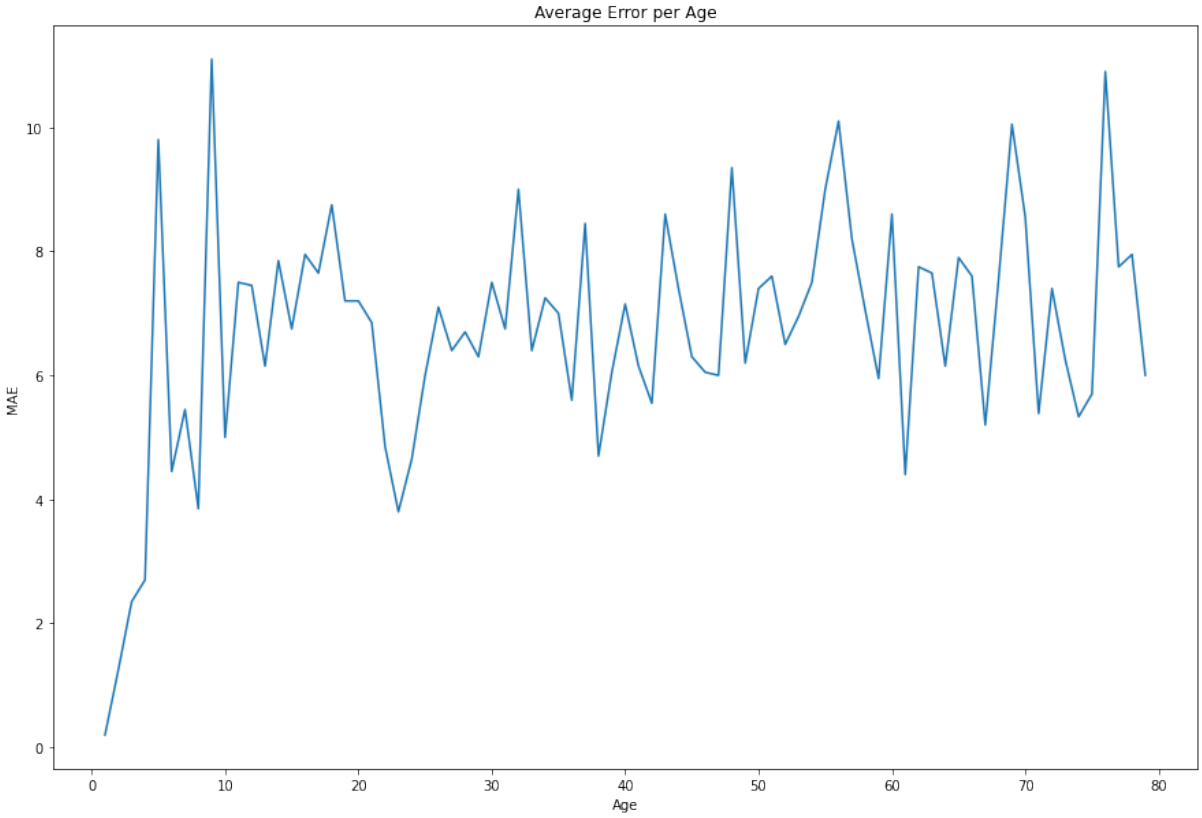
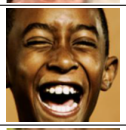
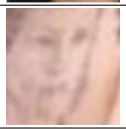
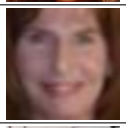
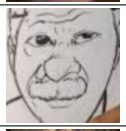
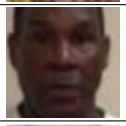


Figure 10: Mean error distribution across age. Notice the model obtained near zero error for age 1 and very low errors for ages 1-5.

Nr	Image	Predicted Value	Target value	Absolute Error
1		75	5	70
2		67	9	58
3		44	5	39
4		39	76	37
5		21	55	34
6		29	63	34
7		43	12	31
8		41	72	31
9		27	56	29
10		40	69	29
11		48	76	28
12		65	37	28
13		41	69	28
14		40	68	28

Nr	Image	Predicted Value	Target value	Absolute Error
15		51	78	27
16		53	26	27
17		43	70	27
18		49	76	27
19		31	58	27
20		69	43	26

5.4 Data binning analysis

Here we can see the exact error means for each age group. Data bucketing is the method that allows us to smooth the error by aggregating labels and predictions over intervals. To check how the bucketing improves the predictions it was decided to look into three possibilities of bucketing. In the first scenario the intervals of length 5 were chosen, in the second scenarios the intervals of the length 10 were chosen, and in the last scenario by analogy the intervals of 15 were used. In this case, accuracy is the factor that describes what percent of predictions and corresponding targets belong to the same interval. Below there is a table showing the results. We can easily notice that choosing wider intervals results in better accuracy which is not surprising.

interval length	accuracy
5	27.06 %
10	49.16%
15	62.28%

5.5 Experiment: Training with and without augmentation

We trained EfficientNet with and without oversampling through the data augmentation step described in section 4.3.3. The results are depicted on figure 11.

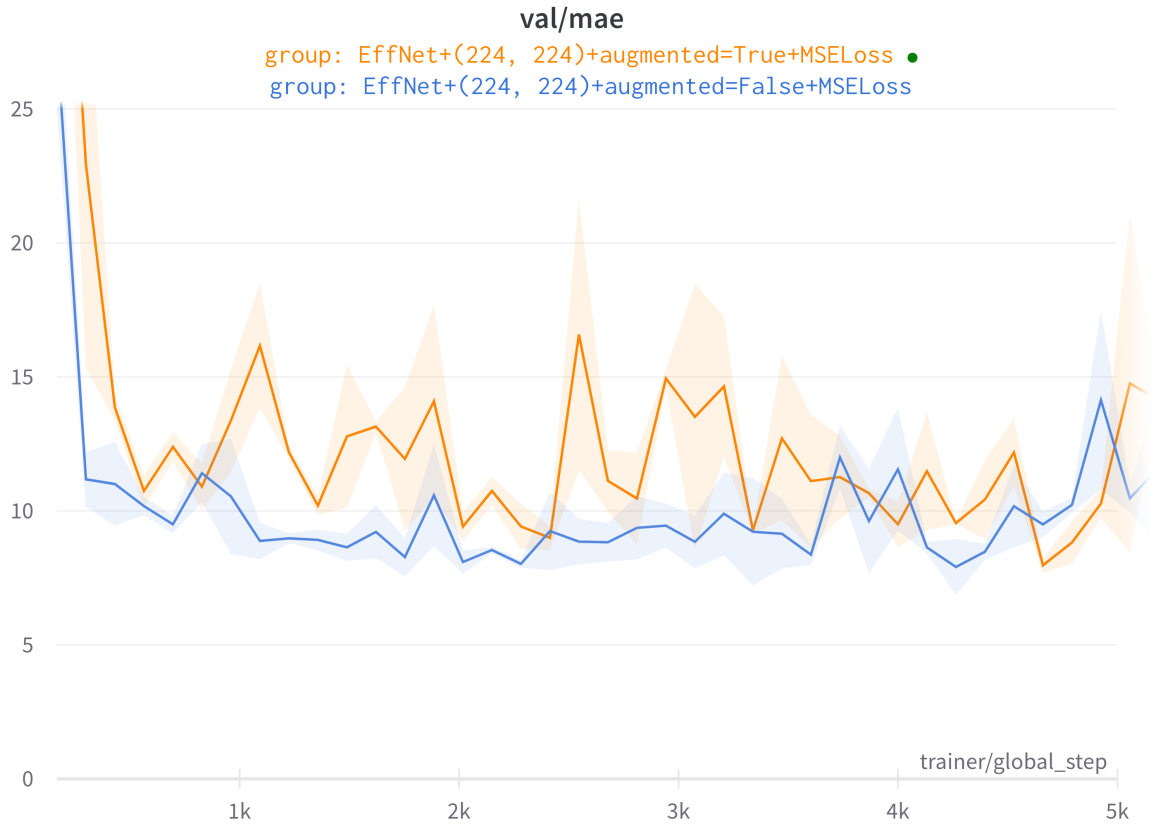


Figure 11: Comparison: Training EfficientNet with and without oversampling through data augmentation. We conclude the technique doesn't improve the accuracy of the model, and it seems to make the convergence to under 7.0 MAE a little slower.

6 Manual

6.1 Introduction

To demonstrate the capabilities of our model, we built an application to use the trained network to capture images directly from a computer camera, or from user-supplied images. The app is based on Streamlit which is an open-source Python library that makes it easy to build interactive web-based applications for machine learning and data science. It allows you to create simple and beautiful user interfaces for your data science projects with little to no JavaScript or HTML knowledge. It also allows you to quickly prototype and test out new ideas and visualizations, making it a great tool for data exploration and experimentation. Streamlit also supports a wide range of widgets, such as sliders, checkboxes, and dropdown menus, which allow users to interact with your app and change the input parameters. Therefore, it was an ideal choice for a project where the main motive was to demonstrate skills related to neural networks.

6.2 Application setup

You don't need to train anything to run the app because it uses saved model.

To launch the application, do the following steps:

1. Create conda enviornment or python virtual enviornment and activate it.
2. Navigate to project root directory using

```
    cd path
```

   where path is path to root directory.

3. Install required packages using

```
    pip install -r requirements.txt
```

   command.

This is everything you need to do before launching application.

6.3 Model training setup

First download dataset from:
`https://www.kaggle.com/datasets/jangedoo/utkface-new`

Unpack the data to `data/` folder.

Next you should process dataset by running `notebooks/data_generation.ipynb`. This can take about 5–20 minutes depending on your hardware. The output weights around 7GB and will be saved to `data/face_age_dataset/` folder.

Now you can run training:

```
python src/train.py
```

The default architecture is custom CNN with img (input) size 100x100. 10 epochs should train on CPU for about 10–30 minutes depending on your hardware.

You can change the architecture in `src/train.py` file.

6.4 How to use application?

When you want to use or application, simply launch it from project root directory by using

```
streamlit run src/app.py
```

command. After few seconds application should appear in your web browser:

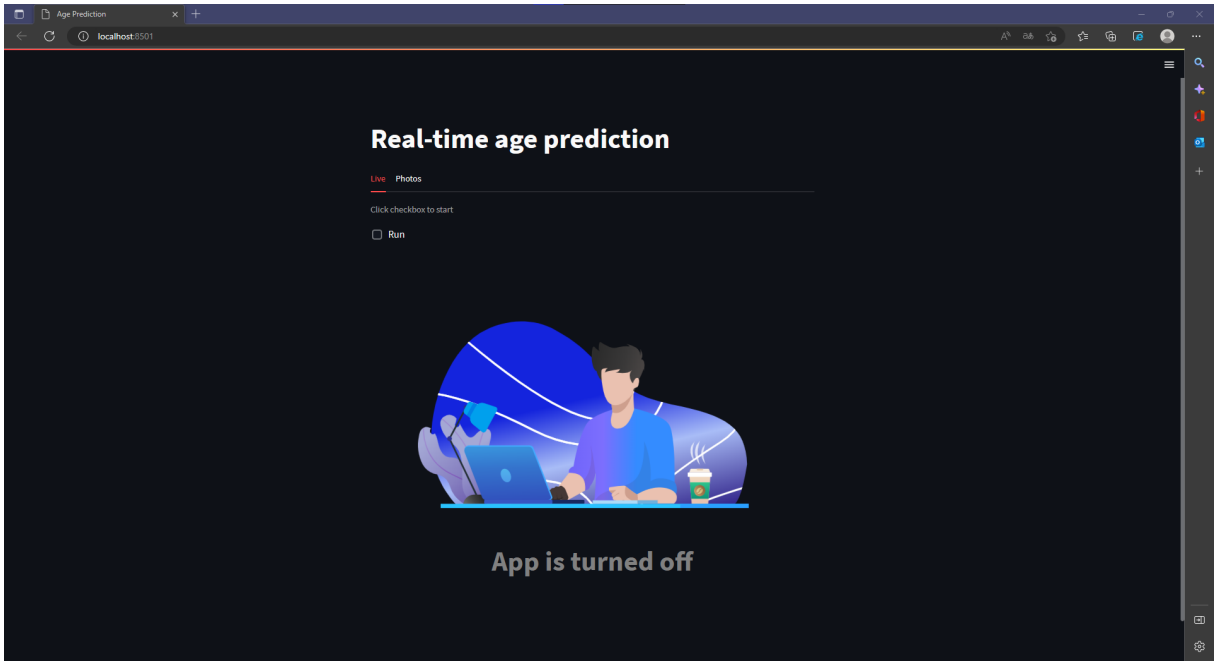


Figure 12: Application view after successful launching

After launching the application you can use it in live camera mode and in photo mode.

6.4.1 Live camera

To use application in live camera mode, you need to check "Run" box to start capturing image from your computer camera.

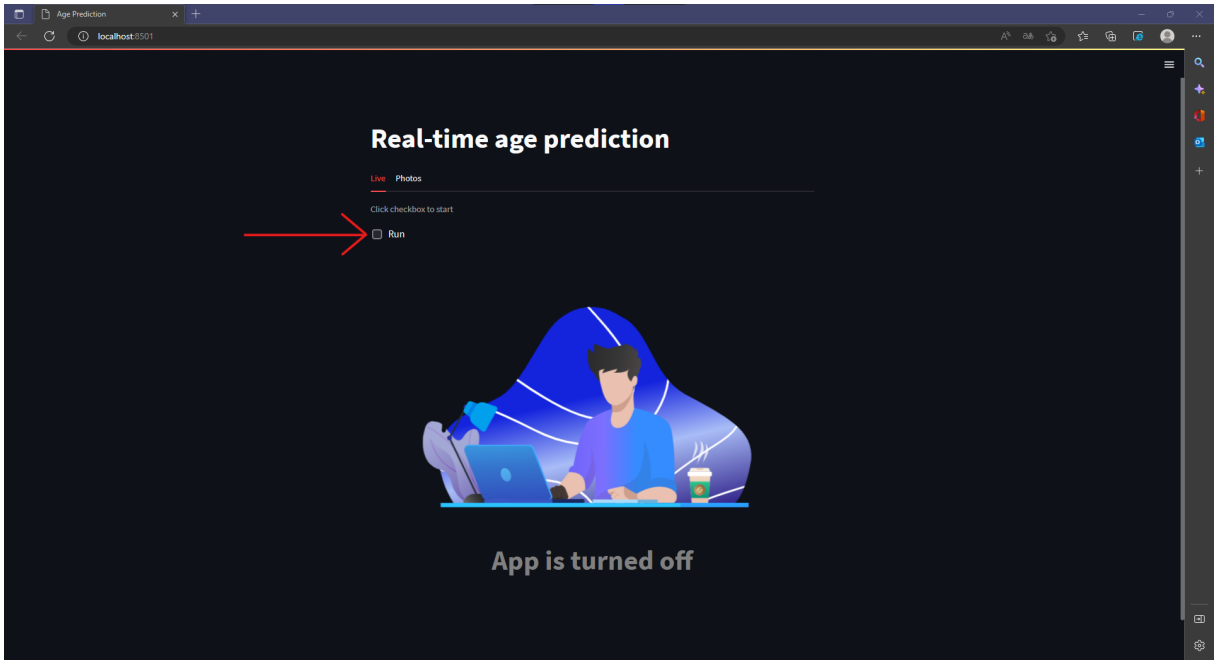


Figure 13: Turning on live camera mode

After few seconds your should see captured images with box and predicted age.

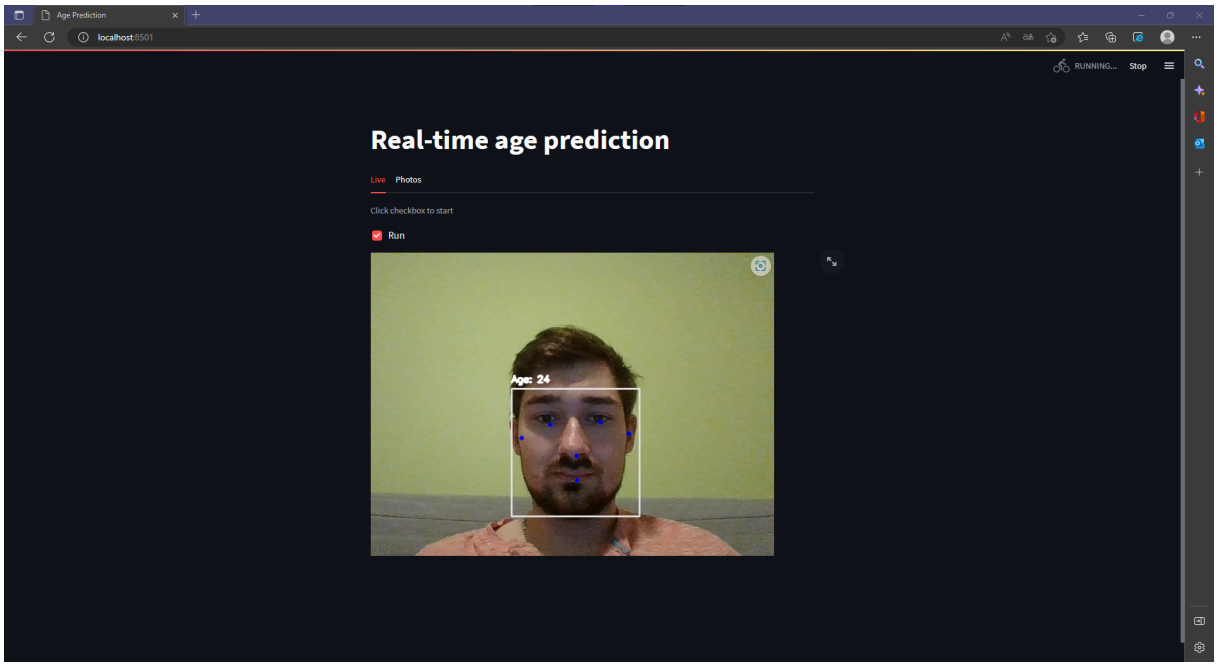


Figure 14: Live camera mode

To turn of the application, check out the box placed above view from camera.

6.4.2 Photos

To predict age in user provided photos you need to click on "Photos" tab.

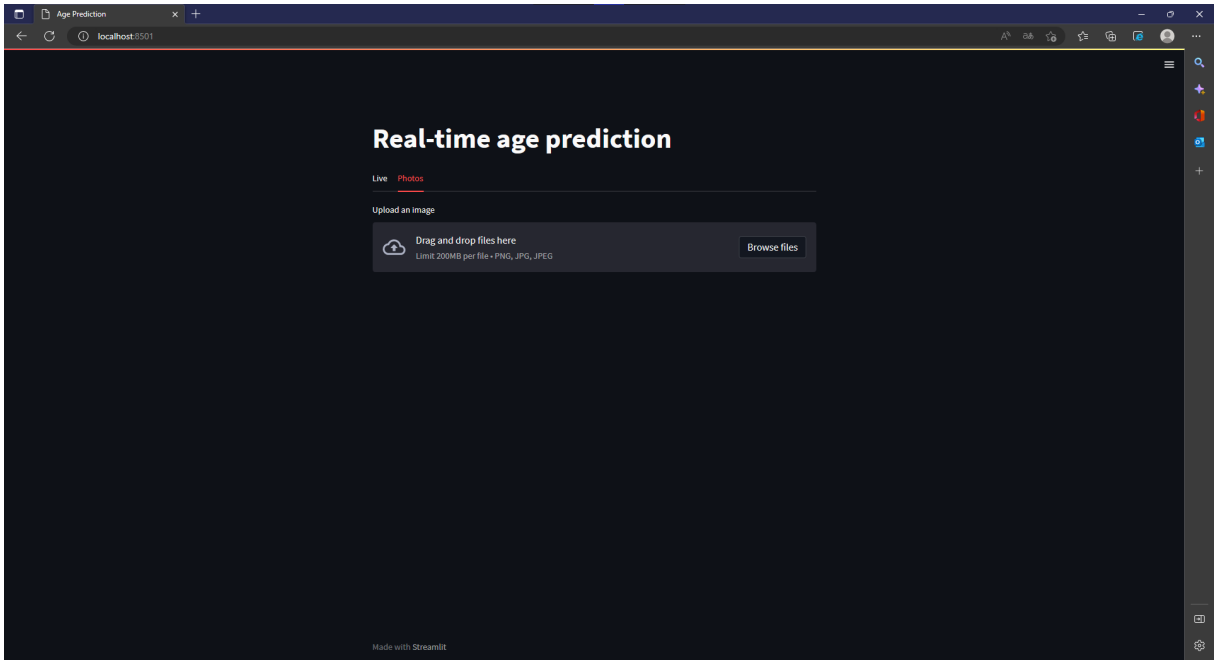


Figure 15: Photo mode

In "Photos" tab you can upload photos from your computer by browsing them, or provide them by drag and drop feature. After uploading image, application will process it using our model. Then (if everything went correctly) success message will appear on screen.

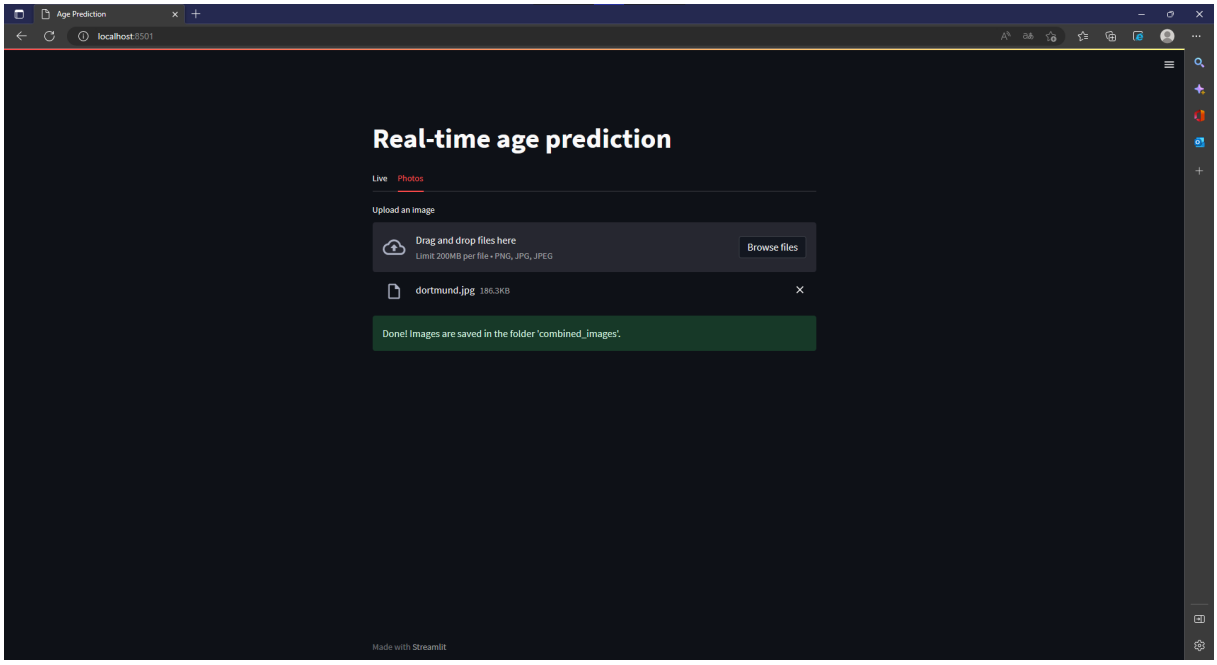


Figure 16: Success

Images with boundary boxes on human faces and predicted age are saved in combined_images directory.

Figure 17: Example

7 Project structure

```
root_folder
├── .gitignore <- list of files to ignore by git
├── .pre-commit-config.yaml <- hooks for autoformatting code
├── .project-root <- file used to detect project root folder
├── requirements.txt <- python dependencies
├── data
│   ├── archive <- raw dataset downloaded from kaggle
│   └── face_age_dataset <- processed dataset
├── models
│   └── best-checkpoint.ckpt <- default checkpoint used by the app
├── notebooks
│   ├── 01_data_exploration.ipynb
│   ├── 02_data_generation.ipynb
│   ├── 03_predictions_analysis.ipynb
│   ├── 04_running_many_seeds.ipynb
│   └── 05_case_study_augmentation.ipynb
└── src
    ├── data
    │   ├── face_age_datamodule.py <- datamodule encapsulating pytorch dataset
    │   └── face_age_dataset.py <- pytorch dataset abstraction
    ├── models
    │   ├── architectures.py <- model architectures
    │   └── face_age_module.py <- module encapsulating train/val/test loop
    ├── utils
    │   ├── face_recognition.py <- app face recognition module
    │   ├── functions.py <- app utilites
    │   └── predict.py <- app prediction module
    ├── app.py <- file for running streamlit app
    └── train.py <- file for running training
```

8 Main code abstractions

The application comprises of two core components: app.py and train.py.

The app.py is the main script that runs the application. It uses the streamlit library to create a user-friendly interface and allows users to test the model by either using their computer camera or by uploading photos.

The train.py is used to train different models, depending on hyperparameters chosen in the code file.

The main abstraction used for encapsulating dataset is called LightningDataModule and the main abstraction used for training, validating and testing is called LightningModule. Both abstractions are described with docstrings and their implementations can be seen below.

8.1 face_age_datamodule.py

```
class FaceAgeDataModule(LightningDataModule):
    """
    LightningDataModule for our FaceAge dataset.

    A DataModule implements 4 key methods:
        def setup(self, stage):
            # things to do on every process in DDP
            # load data, set variables, etc...
        def train_dataloader(self):
            # return train dataloader
        def val_dataloader(self):
            # return validation dataloader
        def test_dataloader(self):
            # return test dataloader

    This allows to share a full dataset without explaining how to download,
    split, transform and process the data.
    """

    def __init__(
        self,
        data_dir: str = "data/",
        img_size: tuple = (224, 224),
        normalize_age_by: int = 80,
        batch_size: int = 32,
        num_workers: int = 0,
        pin_memory: bool = False,
    ):
        super().__init__()

        # this line allows to access init params with 'self.hparams'
    attribute
        # also ensures init params will be stored in ckpt
        self.save_hyperparameters()

        self.data_train: Optional[Dataset] = None
        self.data_val: Optional[Dataset] = None
```

```
        self.data_test: Optional[Dataset] = None

    def setup(self, stage: Optional[str] = None):
        """
        Load data. Set variables: `self.data_train`, `self.data_val`, `self.data_test`.
        """
        if not self.data_train and not self.data_val and not self.data_test:

            # image transformations
            transform_list = []
            transform_list.append(transforms.Resize(self.hparams.img_size))
            transform_list.append(transforms.RandomHorizontalFlip(p=0.5))
            transform = transforms.Compose(transform_list)

            self.data_train = FaceAgeDatasetFromPath(
                img_dir="data/face_age_dataset/train",
                normalize_age_by=self.hparams.normalize_age_by,
                transform=transform,
            )
            self.data_val = FaceAgeDatasetFromPath(
                img_dir="data/face_age_dataset/val",
                normalize_age_by=self.hparams.normalize_age_by,
                transform=transform,
            )
            self.data_test = FaceAgeDatasetFromPath(
                img_dir="data/face_age_dataset/test",
                normalize_age_by=self.hparams.normalize_age_by,
                transform=transform,
            )

    def train_dataloader(self):
        return DataLoader(
            dataset=self.data_train,
            batch_size=self.hparams.batch_size,
            num_workers=self.hparams.num_workers,
            pin_memory=self.hparams.pin_memory,
            shuffle=True,
        )

    def val_dataloader(self):
        return DataLoader(
            dataset=self.data_val,
            batch_size=self.hparams.batch_size,
            num_workers=self.hparams.num_workers,
            pin_memory=self.hparams.pin_memory,
            shuffle=False,
        )

    def test_dataloader(self):
        return DataLoader(
            dataset=self.data_test,
            batch_size=self.hparams.batch_size,
            num_workers=self.hparams.num_workers,
            pin_memory=self.hparams.pin_memory,
            shuffle=False,
        )
```

8.2 face_age_module.py

```python
from typing import Any, List

import pytorch_lightning as pl
import torch
from torchmetrics import MeanAbsoluteError as MAE
from torchmetrics import MeanMetric, MinMetric
from src.models import models


class FaceAgeModule(pl.LightningModule):
    """
    FaceAgeModule is a PyTorch Lightning module for training a model to predict the age of a face in an image.
    It uses a pre-trained model (either SimpleConvNet_100x100, SimpleConvNet_224x224, or PretrainedEfficientNet)
    and fine-tunes it on the input dataset. The module has several methods for training, validation, and testing,
    as well as for logging metrics such as mean absolute error (MAE) and loss.
    """

    def __init__(self, net: str = "EffNet_224x224", rescale_age_by: int = 80.0):
        """
        Initializes the FaceAgeModule with the specified rescale value for the labels.
        The rescale value is used to convert the predicted age value from a range of [0,1] to [0, rescale_age_by].
        """
        super().__init__()

        # this line allows to access init params with 'self.hparams' attribute
        self.save_hyperparameters()

        # architecture
        if net == "SimpleConvNet_100x100":
            self.net = models.SimpleConvNet_100x100()
        elif net == "SimpleConvNet_224x224":
            self.net = models.SimpleConvNet_224x224()
        elif net == "EffNet_224x224":
            self.net = models.PretrainedEfficientNet()
        else:
            raise ValueError(f"Unknown net: {net}")

        # loss function
        self.criterion = torch.nn.MSELoss()
        # self.criterion = torch.nn.SmoothL1Loss()

        # metric objects for calculating and averaging MAE across batches
        self.train_mae = MAE()
        self.val_mae = MAE()
        self.test_mae = MAE()

        # for averaging loss across batches
        self.train_loss = MeanMetric()
```

```
        self.val_loss = MeanMetric()
        self.test_loss = MeanMetric()

        # for tracking best so far validation maeuracy
        self.val_mae_best = MinMetric()

    def forward(self, x: torch.Tensor):
        """
        The forward method is called during training, validation, and testing to make predictions on the input data.
        It takes in a tensor 'x' and returns the model's predictions.
        """
        return self.net(x)

    def predict(self, batch):
        """
        The predict method is called to make predictions on a single batch of data.
        It takes in a batch of data as input and returns the model's predictions.
        """
        x, y = batch
        preds = self.forward(x)
        preds = preds.clip(0, 1)
        return preds

    def on_train_start(self):
        """
        The on_train_start method is called before the training process begins.
        It resets the val_mae_best metric to ensure that it doesn't store any values from the validation step sanity checks.
        """
        self.val_mae_best.reset()

    def model_step(self, batch: Any):
        """
        The model_step method is called during training, validation, and testing to make predictions and calculate loss.
        It takes in a batch of input data and returns the calculated loss and predictions.
        """
        x, y = batch
        preds = self.forward(x)
        loss = self.criterion(preds, y)

        # clip prediction to [0-1]
        preds = preds.clip(0, 1)

        # rescale prediction from [0-1] to [0-80]
        if self.hparams.rescale_age_by:
            preds = preds * self.hparams.rescale_age_by
            y = y * self.hparams.rescale_age_by
            preds = preds.clip(1, self.hparams.rescale_age_by)

        return loss, preds, y

    def training_step(self, batch: Any, batch_idx: int):
```

```
        """
        The training_step method is called during training to calculate the loss and update the model's parameters.
        It also logs the loss and mean absolute error metric to track training progress.
        """
        loss, preds, targets = self.model_step(batch)

        self.train_loss(loss)
        self.train_mae(preds, targets)
        self.log("train/loss", self.train_loss, on_step=False, on_epoch=True, prog_bar=True)
        self.log("train/mae", self.train_mae, on_step=False, on_epoch=True, prog_bar=True)

        return {"loss": loss}

    def validation_step(self, batch: Any, batch_idx: int):
        """
        The validation_step method is called during validation to calculate the loss and update metrics.
        It also logs the loss and mean absolute error metric to track validation progress.
        """
        loss, preds, targets = self.model_step(batch)

        self.val_loss(loss)
        self.val_mae(preds, targets)
        self.log("val/loss", self.val_loss, on_step=False, on_epoch=True, prog_bar=True)
        self.log("val/mae", self.val_mae, on_step=False, on_epoch=True, prog_bar=True)

    def validation_epoch_end(self, outputs: List[Any]):
        """
        The validation_epoch_end method is called at the end of each validation epoch.
        It updates the best mean absolute error metric and logs it.
        """
        self.val_mae_best(self.val_mae.compute())
        self.log("val/mae_best", self.val_mae_best.compute(), prog_bar=True)

    def test_step(self, batch: Any, batch_idx: int):
        """
        The test_step method is called during testing to calculate the loss and update metrics.
        It also logs the loss and mean absolute error metric to track data, and returns the calculated loss, predictions, and targets.
        """
        loss, preds, targets = self.model_step(batch)

        self.test_loss(loss)
        self.test_mae(preds, targets)
        self.log("test/loss", self.test_loss, on_step=False, on_epoch=True, prog_bar=True)
        self.log("test/mae", self.test_mae, on_step=False, on_epoch=True, prog_bar=True)
```

```
144
145     def configure_optimizers(self):
146         """
147         The configure_optimizers method is used to configure the optimizers
     used for training.
148         This method should return a single optimizer or a list of
    optimizers.
149         In this implementation, it returns an instance of the Adam
    optimizer with a learning rate of 0.01.
150         """
151         return torch.optim.Adam(self.parameters(), lr=0.01)
```

9 References

Below there is a list of referenced materials used in the process of project creation:

1. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks Mingxing Tan, Quoc V. Le
2. EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling quality

10 Division of work

Division of work in a group:

- Mikołaj Zalewski - application development
- Łukasz Zalewski - model development, data and results analysis
- Jakub Świstak - model development, data augmentation
- Władysław Olejnik - documentation
- Filip Rak - documentation