1 Introduction

1.1 Background

The question about biodiversity and its importance has been a topic of interest for many years. The term biodiversity is a contraction of biological diversity, which refers to the variety and variability of life forms on Earth. In recent years, a massive decline in biodiversity has been observed, which is mainly due to human activities. The loss of biodiversity is a major concern because it can have a significant impact on the ecosystem and the services it provides (Brondízio et al., 2019). In order to quantify biodiversity and monitor its changes, it is essential to have a reliable and efficient method for measuring biodiversity. Traditional methods for measuring biodiversity are time-consuming and expensive, and they are not suitable for large-scale monitoring. But what if there was a non invasive method that could be used to monitor biodiversity in a fast and efficient way? Ecoacoustics might deliver a promising solution to this problem even tough there remain some issues to be fixed (Scarpelli et al., 2021). Passive acoustic monitoring (PAM) like the using of sound recordings to monitor biodiversity are currently widely researched and developed and even combined with modern artificial intelligence (AI) methods (Deng, 2023).

The focus of this study is to reproduce the results of the paper (Faiss, 2022) and to create a model that can classify the insect sounds with a high accuracy. Furthermore the model will be tested and evaluated for its performance and accuracy. The results will be discussed and compared to the results of the original paper. The goal is to prof that this technology could be accessible for everyone with the knowledge and a regular gaming computer with a graphic processing unit (GPU).

1.2 Insects of Interest

There are countless species of insects in the world, and many of them produce sounds for various reasons. In this study, we are interested in the sounds produced by two groups of insects: Orthoptera and Cicadidae. Orthoptera is an order of insects that includes grasshoppers, crickets, and katydids. ("Orthoptera", 2008) Cicadidae, a members of the superfamily Cicadoidea Westwood are four-winged insects with sucking mouthparts that possess three ocelli and a rostrum that arises from the base of the head. (Sanborn, 2008)

2 Methods

2.1 Dataset

In this study a preexisting dataset is used - the InsectSet32 dataset (Faiss, 2022). The dataset includes 335 recordings of 32 insect species, totaling 57 minutes. About half of the recordings (147) feature nine Orthoptera species from a dataset originally compiled by Baudewijn Odé (unpublished). The remaining 188 recordings, from 23 Cicadidae species, were selected from the Global Cicada Sound Collection on Bioacoustica (Baker et al., 2015a), including recordings published in (Baker et al., 2015b; Popple, 2017). Speech annotations at the beginning of many recordings led to using the last ten seconds of audio. Files with strong noise or multiple species were removed. The number of files per species ranges from four to 22, with durations from 40 seconds to almost nine minutes. All files, originally at least 44.1 kHz, were resampled to 44.1 kHz mono WAV for consistency. The dataset was already split into training, validation and test sets. There are two .csv files containing the labels and the filenames of the recordings.

2.2 Programming Language and Frameworks

To build and train the deep learning model, the programming language Python was used. The Frameworks PyTorch, Lightning are very popular and powerful tools for building deep learning models.

2.3 Deep Learning Model

The deep learning model used in this study is a convolutional neural network (CNN) with residual blocks illustrated in Figure 1. It consists of three main parts: An input layer, a number of residual blocks and an fully connected output layer. The input layer is a convolutional layer with a kernel size of 1 and and output channels set to the base channels (bc) - for this experiment it was set to 8. After the convolutional layer, the input is normed with a batch normalization layer and then passed through a ReLU activation function. This output is than passed trough a number of residual blocks. The model is implemented dynamically, so the number of residual blocks can be set as a hyperparameter. Each residual block consists of two convolutional layers, each followed by a batch normalization layer and a ReLU activation function. The output channels are doubled with every residual block. The residual connection is implemented by passing the input trough a separated convolutional layer with a kernel size of 1 and the same number of output channels as the output of the convolutional layers in the residual block to match dimensions. This output is than added to the output of the transformation in the residual block. At the end of the residual block, the output is passed trough a max pooling layer to reduce the dimensions. The max pooling layer is implemented to alter the kernel size with every residual block. For every odd residual block the kernel size is set to n_max_pool - in this experiment it was set to 3 - reducing the dimensions by a factor of 3. For every even residual block the kernel size is set to 1, so the dimensions stay the same. The output of the residual blocks is than passed trough a fully connected layer, consisting of a global average pooling layer to reduce the dimensions to the number of classes. An additional convolutional layer with kernel size 1 to match the number of classes. A flattening layer to get of the dimensions H and W already reduced to 1 and a softmax activation function to get the probabilities for each class as a vector of length n_classes - in this experiment 32 - elements between 0 and 1 that sum up to 1.

2.4 Data Processing

A custom Dataloader was implemented, to handle the data processing on the fly and provide the trainer with then data samples matching the chosen indices. To implement the dataloader, the PyTorch Dataset and DataLoader classes where used. For the data processing, the torchaudio and numpy libraries where used. There is two steps to the data processing: Sampling and Transformation.

2.4.1 Sampeling

The audio files are of different lengths and the model can only handle inputs of a fixed size. Since the smallest files are of a length of around 1 second and the longest file is around 160 seconds, a compromise had to be made. To only sample the files to a length of 1 second would mean very little information being available for the model to learn from. On the other hand if the files are sampled for a length of more than a second, the short files would need to be padded with zeros meaning the file starts with a basically empty part. This could lead to the model learning from the length of the empty part and not the actual audio signal. To avoid this, the audio files where sampled to a random length between 1 and 10 seconds and then padded with zeros to the fixed length of 10 seconds. To implement this, a custom method was implemented as shown in Listing 1.

Listing 1: Python code for the sampeling of the filies

```python
import numpy as np
import torch

def get_random_part_padded(self, waveform: Tensor, samplerate: int) -> Tensor:

    min_len_in_samples = int(self.min_len_in_seconds * samplerate)
    max_len_in_samples = int(self.max_len_in_seconds * samplerate)

    if self.min_len_in_seconds == -1:
        sample_start_index = -max_len_in_samples
        sample_end_index = None

    else:
        part_length = np.random.randint(min_len_in_samples, max_len_in_samples +
    1)
        sample_length = waveform.shape[1]
        part_length = min(part_length, sample_length)
        sample_start_index = np.random.randint(0, sample_length - part_length + 1)
        sample_end_index = sample_start_index + part_length

    waveform_part = waveform[:, sample_start_index:sample_end_index]
    actual_part_length = waveform_part.shape[1]
    pad_length = max_len_in_samples - actual_part_length
    waveform_pad = torch.nn.functional.pad(waveform_part, pad=(pad_length, 0, 0, 0))

    return waveform_pad
```

2.4.2 Transformation

The model is actually one, usually used for image classification and therefore expects two dimensional input while an audio file has only one dimension. To transform the samples into into a format, that the model can handle, a short time Fourier transformation (STFT) was applied. The Fourier transformation is a mathematical operation that transforms a function of time into a function of frequency. An other way to describe this is, that a series of spectrograms for short time slices of the audio signal are created and aligned in a 2D array - basically a visualization of the audio signal and therefore something a image classification model can handle. In addition a second version was implemented, where the STFT where transformed additionally. The two versions are visualized for a random sample with no padding in Figure 2. The frequency bins were transformed into mel bins, which are a more human like representation of the frequency content of the audio signal. This transformation is called a mel-spectrogram and is commonly used in the field of ecoacoustics (Stowell, 2022, p. 7). Both versions of the transformation where transformed into decibels and normalized before being passed into the model.

To implement the two varieties of the transformation, the library torch and torchaudio where used. A custom transformation method was implemented, that can be used as a layer in the model as shown in Listing 2. Some of the parameters of the transformation where made configurable and others dependant on them where calculated. In the hyperparameter tuning phase, the only parameter that was tuned was the number of mel bins, which was set to either 64 to try the mel-spectrogram or -1 to use the standard spectrogram.

Listing 2: Python code for the transformation of the audio signal

```
import torch
import torchaudio

class NormalizeSpectrogram(torch.nn.Module):
    def forward(self, tensor):
    return (tensor - tensor.min()) / (tensor.max() - tensor.min())

normalize_transform = NormalizeSpectrogram()

if n_mels == -1:
    spectrogram = torchaudio.transforms.Spectrogram(
        n_fft=n_fft,
        hop_length=int(n_fft/2),
        win_length=n_fft)
else:
    spectrogram = torchaudio.transforms.MelSpectrogram(
        n_fft=n_fft,
        hop_length=int(n_fft/2),
        win_length=n_fft,
        n_mels=n_mels,
        f_max=self.sample_rate / 2)

db_transform = torchaudio.transforms.AmplitudeToDB(top_db=top_db)

self.transform = torch.nn.Sequential(
    spectrogram,
    db_transform,
    normalize_transform)

```

2.5 Fitting the Model

The training was completely handled by the PyTorch Lightning framework. The logging was done with the TensorBoard logger and with an additional custom logger to get easier access to the data afterwards. The hyperparameter grid search was implemented with a custom Python script.

2.5.1 Training

The training was done on a single GPU. The model was trained for a maximum of 2000 epochs with a early stopping callback, that stopped the training if the validation loss did not improve for patience = 100 epochs. The model was trined with a batch size of 10. The optimizer used was the AdamW optimizer of the PyTorch library with a weight decay of 0. The loss function used was the CrossEntropyLoss function of the PyTorch library with default parameters and class weights according to available data per class. Different learning rates where tried during the hyperparameter tuning referred to in Section 2.5.2. Only the best model was saved and used for the evaluation of the model. In order to simplify the evaluation, on completion of the training the whole dataset was predicted and saved to a csv file in the log folder.

2.5.2 Hyperparameter Tuning

For the hyperparameter tuning, a select number of hyperparameters where chosen to be tuned. Considerations like the experience of some early tests, the computational resources available and the time frame of the project where taken into account. The hyperparameters that where chosen for the grid search are shown in Table 1. For the grid search, models for all possible combinations of the hyperparameters - in this case $2 \times 3 \times 2 \times 3 = 36$ where trained. To implement the grid search a short Python script was written to create the system commands to start the training with the different hyperparameter combinations.

Table 1: Hyperparameters and values used for the grid search.

Hyperparameter	Description	Variations	Values
n_mels	transformation (-1 for regular STFT)	2	64, -1
n_res_blocks	number of res blocks	3	2, 3, 4
learning_rate	step size during optimization	2	0.001, 0.0001
kernel_size	dimension of the filter	3	3, 5, 7

2.6 Evaluation

To evaluate the models, the predictions already implemented at the end of the training. The output of the model was a vector of length 32 with probabilities for each class where transformed into a class ID by taking the index of the highest probability using the argmax function of the numpy library. For all the models they where stored in a csv file in their log folder. A script was implemented to summarize the results of the models and to calculate the metrics for their performance.

2.6.1 Metrics

Accuracy: The accuracy is the ratio of correctly predicted observations to the total observations. It was calculated using the mean function of the numpy library:

```
np.mean(y_true == y_pred)
```

F1 Score: The F1 score is a metric that combines the precision and recall of a model. In this case the macro average was used, which calculates the F1 score for each class and then takes the mean. It was calculated using the f1_score function of the sklearn library:

```
f1_score(y_true, y_pred, average='macro', sample_weight=None, zero_division='warn')
```

F1 Score per Class: The F1 score per class is the F1 score for each class. It was calculated using the f1_score function of the sklearn library with the average parameter set to None:

```
f1_score(y_true, y_pred, average=None, sample_weight=None, zero_division='warn')
```

It was calculated for each class and for each model from the hyperparameter tuning. The results where than aggregated to get the mean F1 score per class over all models. The results where than visualized in a bar chart combined with the available data per class - refer to figure 5.

Confusion Matrix: The result of the Predictions of the best model where used to create a confusion matrix. The confusion matrix is a table that is often used to analyze the performance of a classification model. Predictions and true labels are compared and visualized in a matrix shown in figure 7. The confusion matrix in Figure was calculated using the confusion_matrix function of the sklearn library.

Pearson Test: The Pearson test was used to test the correlation between the model size and the accuracy and F1 score of the models. The Pearson test was calculated using the pearsonr function of the scipy library:

```
for i, metric in enumerate(['accuracy', 'f1']):
  pearson_corr, p_value = stats.pearsonr(ex.summary[metric], ex.summary['num_trainable_params'])
```

The null hypothesis (H0) posited that there is no correlation between the model size and the models accuracy or F1 score, with the significance level (alpha) set at 0.05.

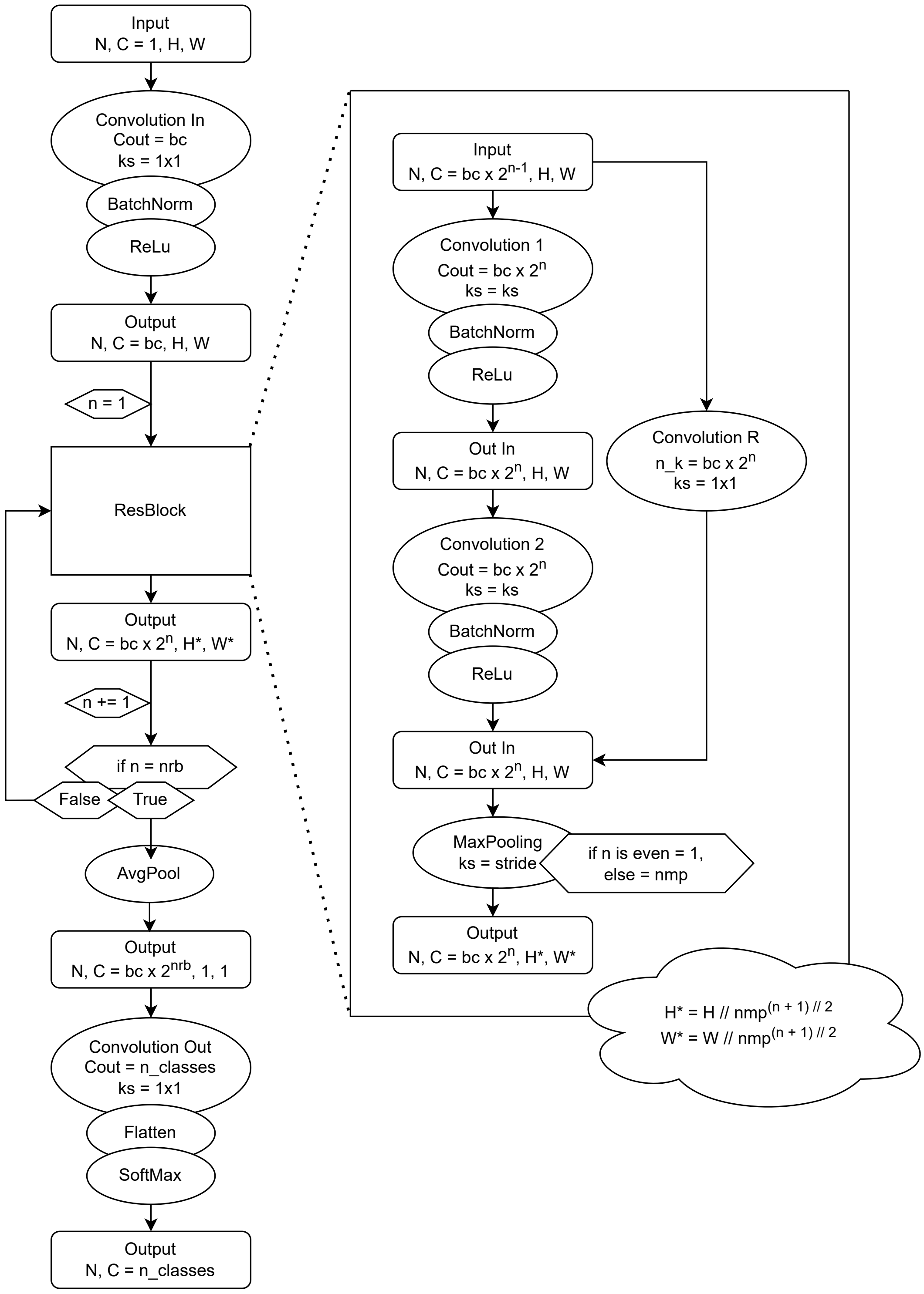

Figure 1: Flow chart illustrating the models architecture. N: elements in batch, C: channels, H: height, W: width, Cout: output channels, ks: kernel size, bc: base channels, nrb: number of residual blocks, n_classes: number of classes, nmp: parameter for max pooling

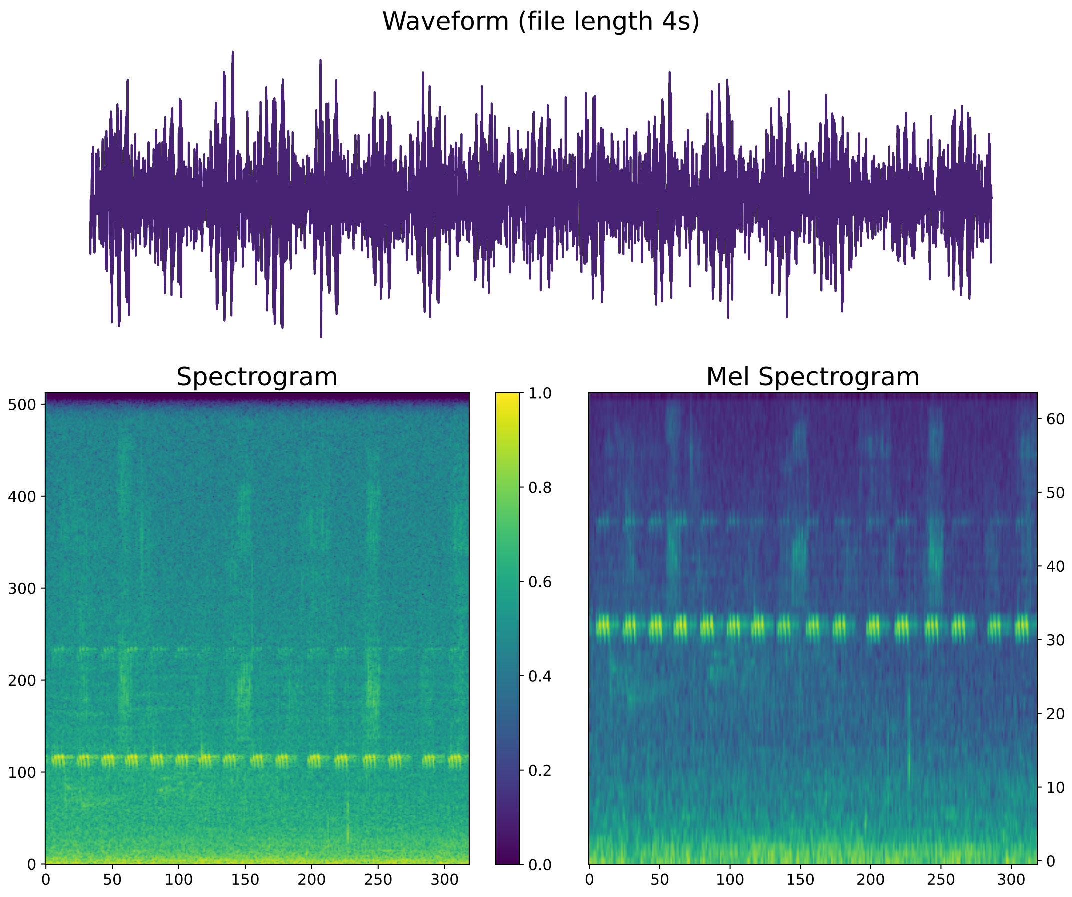

Figure 2: Visualization of the two transformations of the audio signal.

3 Results

3.1 Hyperparameter Tuning

Table **??**

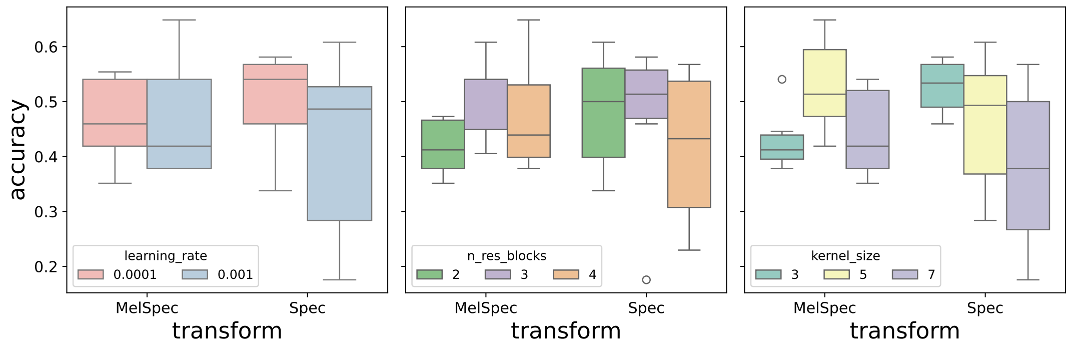

Figure 3: Accuracy of the models for different hyperparameter grouped by the transformation type.

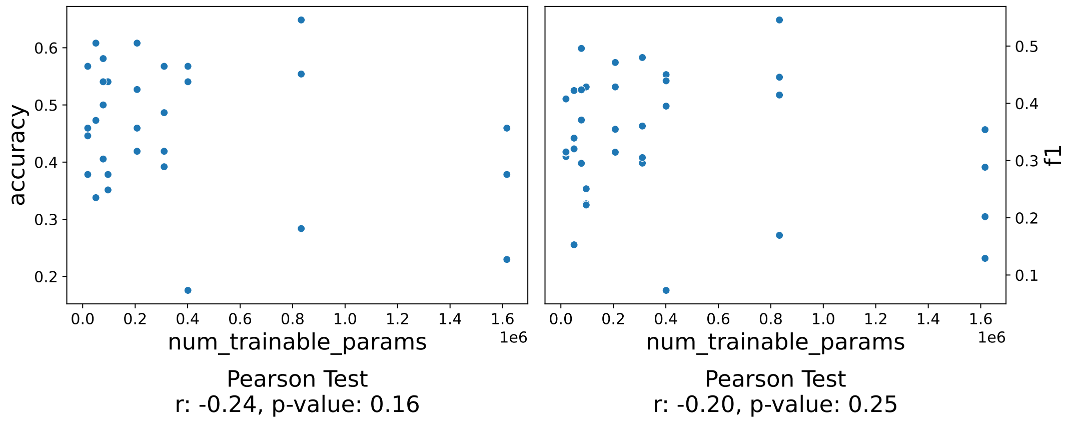

Figure 4: Model size compared to the accuracy and F1 Score of the models.

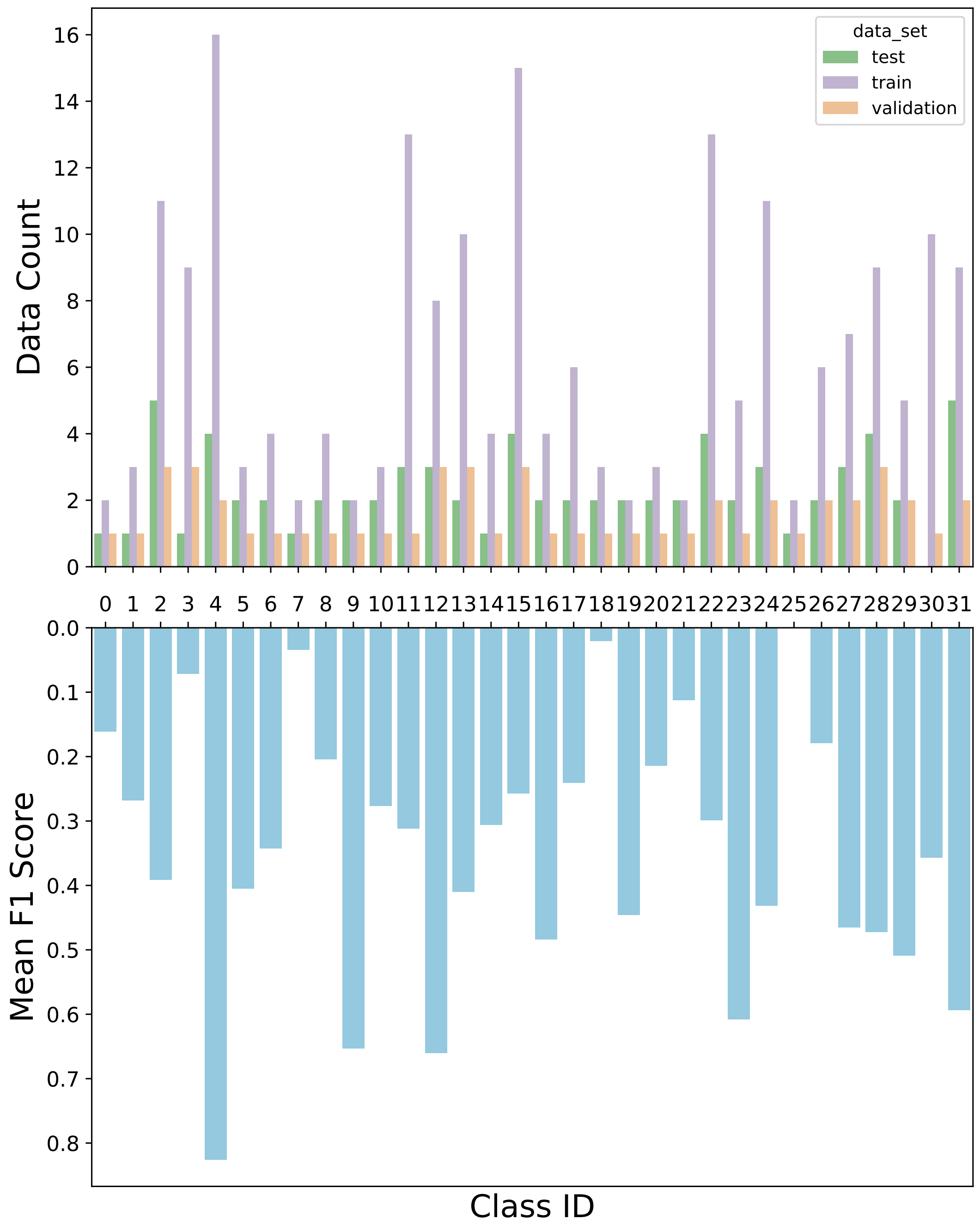

Figure 5: F1 Score per class as mean trough all models compared to data distribution.

3.2 Performance of the best Model

The best performing configuration for the model was found to be the one with the following hyperparameters:

- **n_mels:** 64
- **n_res_blocks:** 4
- **learning_rate:** 0.001
- **kernel_size:** 5

On the test set, the model achieved an accuracy of 0.649. The confusion matrix for the test set is shown in Figure 7.

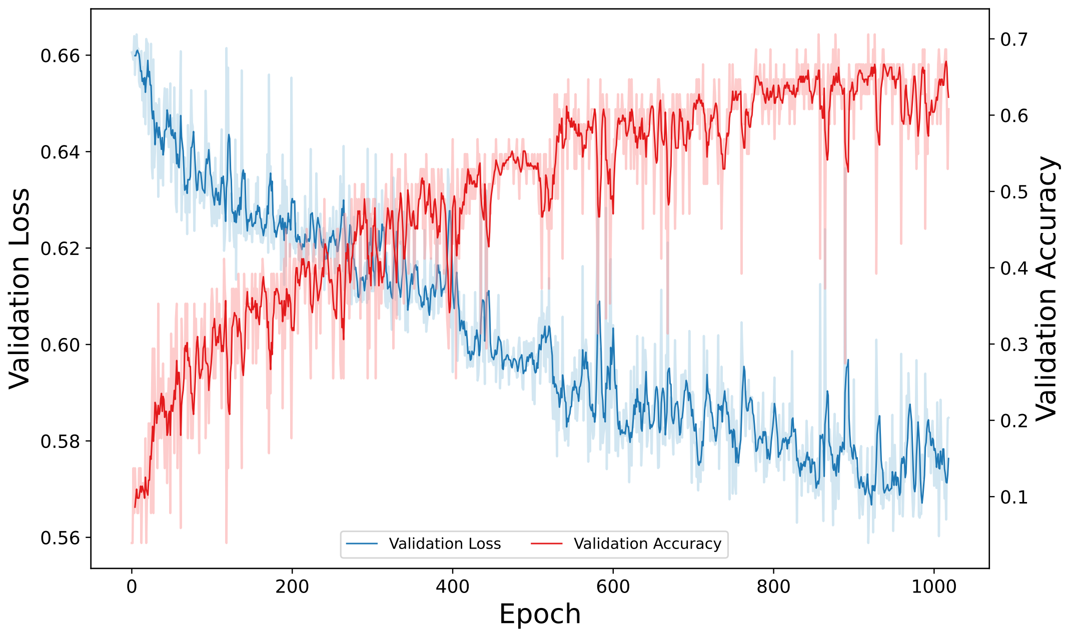

Figure 6: The validation loss and accuracy of the best model. Light version is not smoothed and dark version is smoothed with a window size of 5.

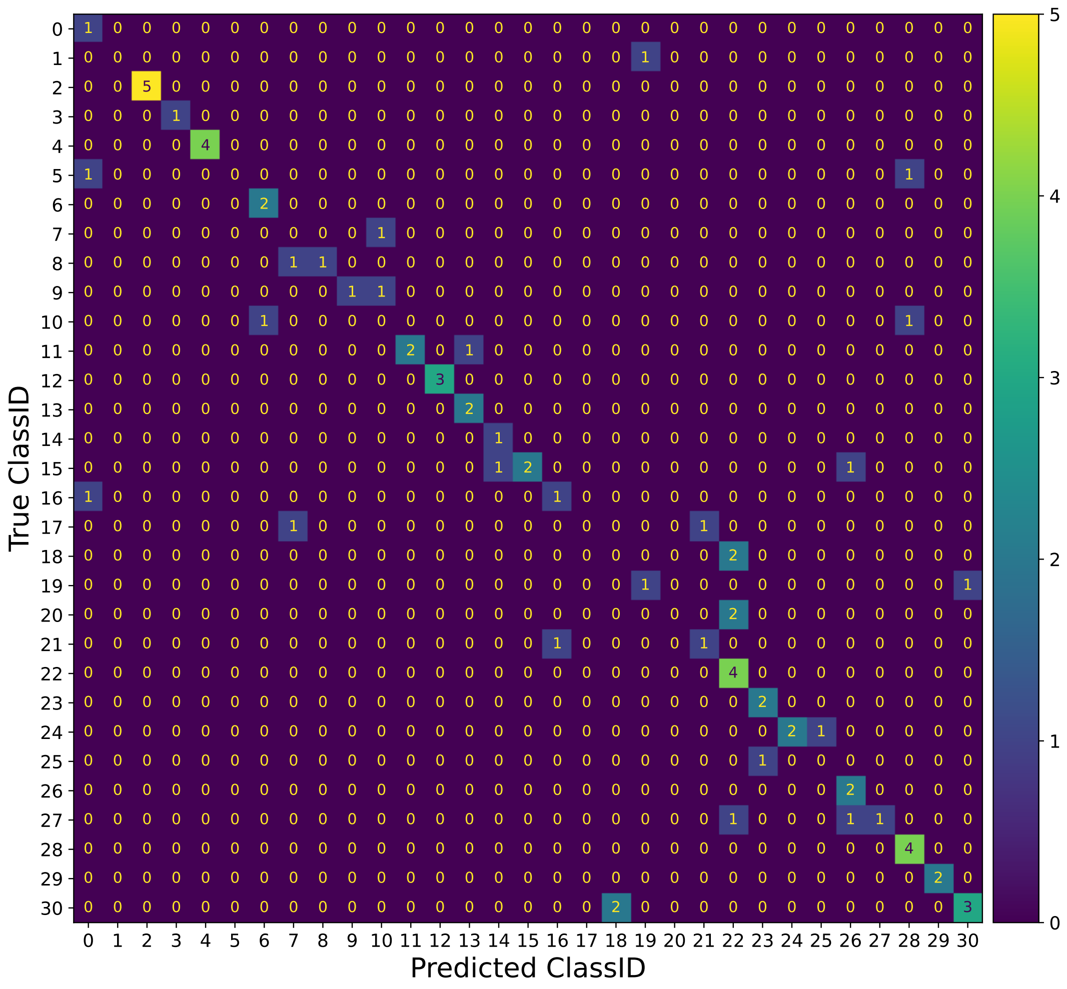

Figure 7: Confusion matrix for the predictions of the test set using the best model.

4 Discussion

5 Conclusion