UiA Fakultet for teknologi og realfag

IKT450 - Deep Neural Networks

Assignments

Autumn 2024

Andrea Zatti, Yannik Weber, Sikko Tammena

December 6, 2024

Mandatory Group Declaration

Each student is solely responsible for familiarizing themselves with the legal aids, guidelines for their use, and rules regarding source usage. The declaration aims to raise awareness among students of their responsibilities and the consequences of cheating. Lack of declaration does not exempt students from their responsibilities.

1.	We hereby declare that our submission is our own work and that we have not used other sources or received any help other than what is mentioned in the submission.	Yes / No
2.	**We further declare that this submission:** • Has not been used for any other examination at another department/university/-college domestically or abroad. • Does not reference others' work without it being indicated. • Does not reference our own previous work without it being indicated. • Has all references included in the bibliography. • Is not a copy, duplicate, or transcription of others' work or submission.	Yes / No
3.	We are aware that violations of the above are considered to be cheating and can result in cancellation of the examination and exclusion from universities and colleges in Norway, according to the Universities and Colleges Act, sections 4-7 and 4-8 and the Examination Regulation, sections 31.	Yes / No
4.	We are aware that all submitted assignments may be subjected to plagiarism checks.	Yes / No
5.	We are aware that the University of Agder will handle all cases where there is suspicion of cheating according to the university's guidelines for handling cheating cases.	Yes / No
6.	We have familiarized ourselves with the rules and guidelines for using sources and references on the library's website.	Yes / No
7.	We have in the majority agreed that the effort within the group is notably different and therefore wish to be evaluated individually. Ordinarily, all participants in the project are evaluated collectively.	Yes / No

Publishing Agreement

Authorization for Electronic Publication of Work The author(s) hold the copyright to the work. This means, among other things, the exclusive right to make the work available to the public (Copyright Act. §2).

Theses that are exempt from public access or confidential will not be published.

We hereby grant the University of Agder a royalty-free right to make the work available for electronic publication:	Yes / No
Is the work confidential?	Yes / No
Is the work exempt from public access?	Yes / No

Contents

1 K-Nearest Neighbours (KNN)

1.1 Introduction

In this task, we were required to implement the k-nearest neighbors (KNN) algorithm from scratch, using only Python, without relying on pre-existing models from popular machine learning libraries such as Scikit-learn. Specifically, the goal was to implement the KNN algorithm to perform binary classification on the 'Pima Indians Diabetes' dataset. The KNN algorithm is a simple and intuitive method for classification in which the k nearest neighbours of a data point are taken into account in order to determine its class by majority voting. The goal of this implementation was to understand how the KNN algorithm works and apply it to a given dataset to make accurate predictions. This dataset originates from the National Institute of Diabetes and Digestive and Kidney Diseases. Its primary objective is to predict diagnostically whether a patient has diabetes, based on specific diagnostic measurements provided within the dataset. The dataset is curated with certain constraints, specifically including only female patients who are at least 21 years old and of Pima Indian heritage. In total it contains medical data from 768 female Pima Indian patients. The dataset includes eight different features like the number of pregnancies, the blood glucose concentration, the blood pressure and some more.

1.2 Data Analysis

Before implementing the KNN algorithm, we inspected the data using various functions from the Pandas library. Specifically, we checked for the presence of null values, and when found, we decided to replace them with the median of the corresponding feature. We then explored other techniques to refine the data and prepare it in a more suitable format, such as removing outliers using the Interquartile Range (IQR) method, and adding new features to see if combining existing ones in different ways could be effective. Additionally, we attempted to balance the diabetic and non-diabetic classes to make the classifier more general and independent of this factor. Despite these efforts, the techniques did not prove sufficiently useful, likely due to the dataset being too small, making it more challenging to apply modifications effectively. Alternatively, it's possible that the dataset was already sufficiently clean and well-organized from the start. Ultimately, the only data cleaning techniques we retained to achieve better results were replacing null values with the median and applying standard scaling to the feature values.

1.3 Implementation and Evaluation

The K-Nearest Neighbors (KNN) algorithm is a simple yet powerful supervised learning method used for classification and regression tasks. KNN operates by identifying the 'k' closest data points (neighbors) to a given input within the feature space and then predicts the output based on the majority class among the neighbors, for classification, or by averaging the neighbors' values, for regression. The distance between data points is typically measured using metrics such as Euclidean distance, like in our case. KNN is non-parametric, meaning it makes no assumptions about the underlying data distribution, and is highly interpretable. However, its performance is sensitive to the choice of

'k' and the scale of the data, making preprocessing steps like scaling essential for optimal results. The final Python code with which we implemented KNN can be viewed in the attached file below.

```
class KNN:
    def __init__(self, k=3):
        self.k = k

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train

    def predict(self, X_test, y_test=None):
        predictions = [self._predict(x) for x in X_test]
        predictions = np.array(predictions)
        return predictions

    def _predict(self, x):
        distances = [euclidean_distance(x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

    def mse(self, y_true, y_pred):
        return np.mean((y_true - y_pred) ** 2)
```

The performance of the K-Nearest Neighbors (KNN) algorithm can be adversely affected if the data is not scaled prior to its application. The K-nearest neighbors (KNN) algorithm calculates the distance between data points in order to determine which data points are the most similar. Consequently, features with larger numerical ranges (e.g., glucose or age) will have a greater influence on the distance calculation. This can result in the generation of biased results, whereby the algorithm assigns greater significance to features with larger scales, which may ultimately compromise the accuracy of the predictions and lead to suboptimal outcomes. The StandardScaler() function in scikit-learn is a preprocessing tool that standardizes the features of a dataset by scaling them to have a mean of 0 and a standard deviation of 1. Specifically, StandardScaler() subtracts the mean and divides by the standard deviation for each feature. This process is essential when using algorithms that are sensitive to the scale of features, such as K-Nearest Neighbors (KNN) or linear regression, as it ensures that all features contribute equally to the model.

```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

The K-Nearest Neighbors (KNN) algorithm may yield disparate results on each iteration if the dataset contains ties or if the algorithm randomly selects neighbors when distances are equal. This instability arises because KNN is highly sensitive to the local structure of the data, meaning that minor alterations to the dataset or the random selection of neighbors can result in disparate predictions. This variability indicates that KNN may not generalize effectively, as it is contingent upon the specific distribution of the training data, rendering it less robust in comparison to alternative models.

1.4 Hyperparameter Tuning

Hyperparameter tuning is the process of optimizing a model's performance by adjusting its hyperparameters—settings defined before training begins, in our case k which represents the number of neighbors considered when evaluating each point. Unlike model parameters, which are learned during training, hyperparameters need to be chosen manually. Tuning involves experimenting with different values to find the best combination, improving the model's accuracy and generalization, and that is what we have tried to do. In the 'K Value Selection' section of the code, we present the six metrics we chose to evaluate the model. Specifically, we display their values as the parameter k varies. As observed in the graphs [1], all the metrics indicate that the optimal value for k is 90. Subsequently, the confusion matrix [2] provides a clearer view of the model's performance when tested on the dataset with the k parameter setted at the best value.

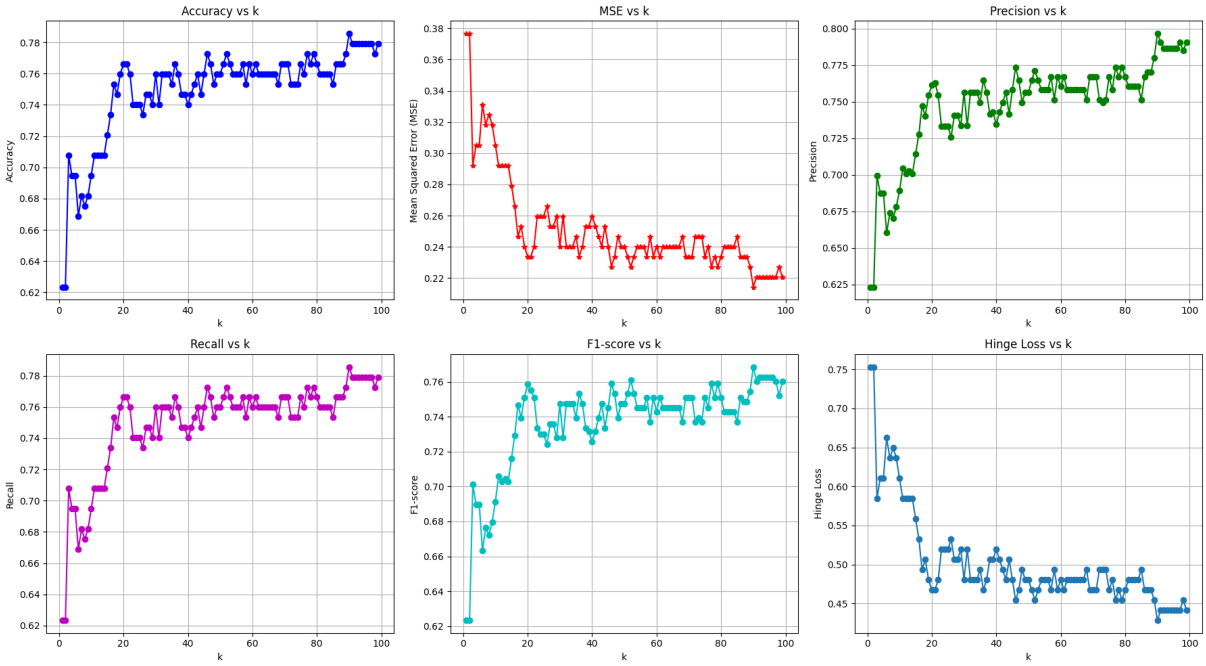


Figure 1: Performance Metrics

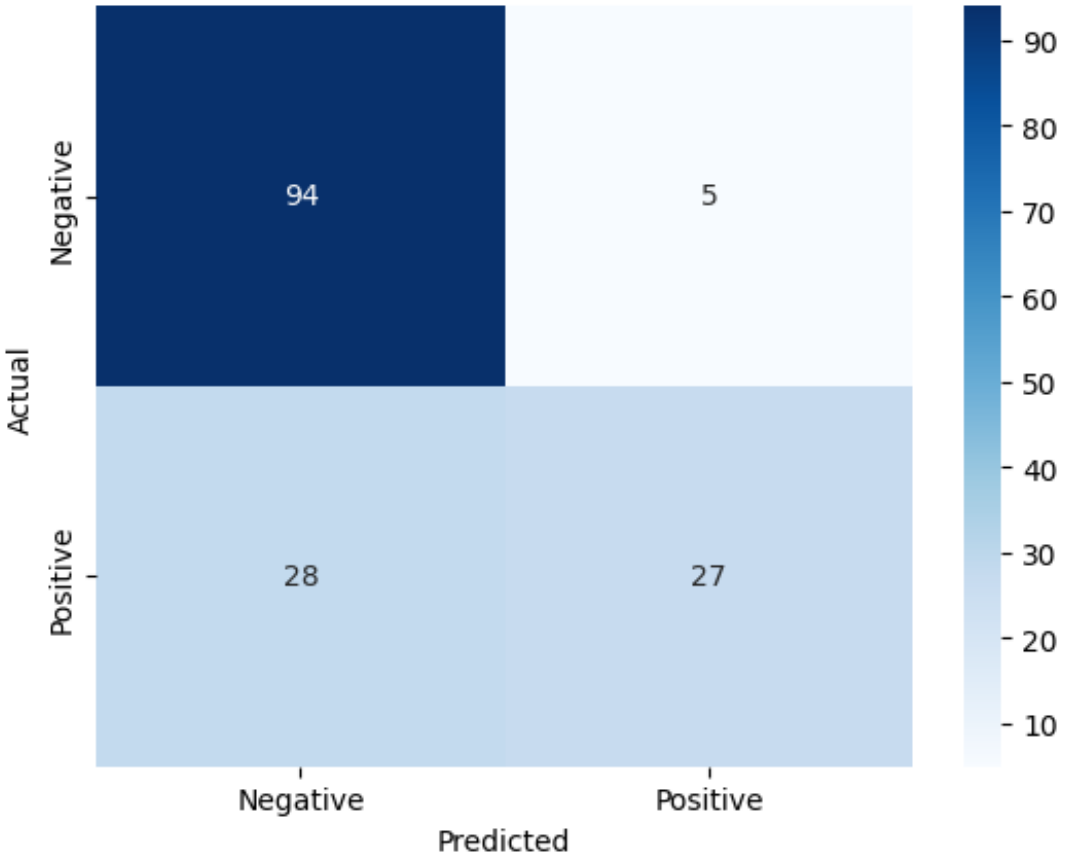


Figure 2: Confusion Matrix

2 Neural Networks

In this task, a basic neural network was constructed in Python, based on the architectural principles of a multilayer perceptron (MLP). The network comprises DenseLayer and activation functions, including Sigmoid and Leaky ReLU. The forward and backward passes are implemented to calculate the predictions and optimise the weights based on the mean squared error. The use of momentum improves the efficiency of the learning process. The network is trained using training data and can make predictions by rounding outputs to a threshold of 0.5. In this implementation, a custom neural network was designed to classify classes *cp* and *im* from the E. coli dataset. The architecture consists of two dense layers, with the first layer containing 7 input neurons and 3 output neurons, followed by a Sigmoid activation function. The second dense layer then maps the 3 outputs from the first layer to a single output neuron, also utilizing a Sigmoid activation function. With these parameters and this architecture, the model was trained over 1000 epochs with a learning rate of 0.1 and an impulse of 0.9 and achieves an accuracy of almost 100%.

A similar outcome was achieved using PyTorch as a high-level library. Significant improvements were observed in convergence time and accuracy when utilizing optimizers such as Adam or Stochastic Gradient Descent (SGD) with momentum. Various activation functions were also explored during the experimentation; however, the best results were consistently obtained with the Sigmoid activation function.

The subsequent Figure 3 illustrates the loss and accuracy during training with the high-level library and the aforementioned architectural design.

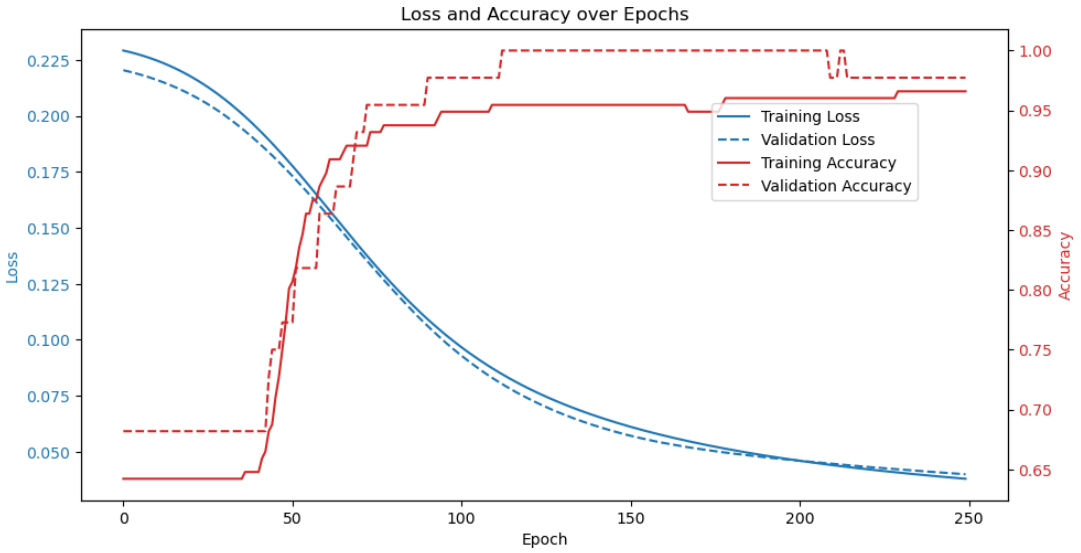


Figure 3: Loss and Accuracy over Epochs

3 Convolutional Neural Networks

The objective was to implement a Convolutional Neural Network (CNN) for the purpose of classifying food images. For illustrative purposes, an example of the images can be seen in the provided figure (Figure 4). The dataset comprises 16,643 food images, which have been grouped into 11 main categories: bread, dairy products, desserts, eggs, fried food, meat, noodles/pasta, rice, seafood, soup, and vegetables/fruit. As illustrated in Figure 4b the classification of this image is not always straightforward. The image in question could conceivably be assigned to the rice class as well. These discrepancies are also reflected in the accuracy metrics.

(a) A picture of the class *bread* (b) A picture of the class *meat* (c) A picture of the class *egg*

Figure 4: Food dataset example images

Additionally, we tested pretrained models, including ResNet18 and VGG16, and compared them to the simpler FoodNet model. In the case of **VGG16**, despite applying various techniques such as regularization and data augmentation, test accuracies remained at only 15%. While further hyperparameter tuning might improve the accuracy, due to the model's size and the substantial time required for training, this architecture was not pursued further.

In contrast, **ResNet18** achieved test accuracies of around 50% without any regularization or data augmentation techniques. When these techniques were applied, the test accuracy improved significantly to 83%. Achieving 83% accuracy on a moderately complex dataset with 11 classes is satisfactory, particularly considering that some images are difficult to classify even for the human eye due to additional elements present in the images. A major advantage of ResNet18 is its compact size coupled with high performance. This is largely due to the use of residual blocks, which include skip connections that allow the output of one layer to bypass intermediate layers and be fed directly to later layers. This design prevents gradient vanishing and simplifies training. ResNet18 consists of 17 weighted layers: 16 convolutional layers grouped into 4 blocks, each with 2 convolutional layers, followed by a fully connected layer. Each convolutional layer is paired with batch normalization and ReLU activation.

As shown in Figure 5, both the training and test accuracies converge relatively quickly. The training could have been halted after just 10 epochs. To facilitate automatic training termination, an early stopping mechanism was implemented.

The *EarlyStopping* class monitors the validation accuracy during training. If the validation accuracy does not improve for a specified number of epochs (patience), the training is halted to prevent overfitting. The class also saves the model with the best accuracy, ensuring that the most effective version is retained. Key parameters include:

- **patience**: The number of epochs to wait after the last improvement before stopping.
- **delta**: The minimum change in validation accuracy required to consider it an improvement.

In comparison, the **FoodNet** model is a much smaller, simpler CNN. FoodNet consists of three convolutional blocks, each containing a convolutional layer, ReLU activation, and max-pooling, which progressively extract key image features. The number of feature maps increases from 64 to 256 through the network. Classification is then performed by two fully connected layers, with a dropout layer included to mitigate overfitting. The final layer outputs predictions for the predefined number of classes. Similarly, the accuracy achieved in this instance was within the range of 15%. It is therefore evident that improvements must be made to the architectural design in order to achieve enhanced performance.

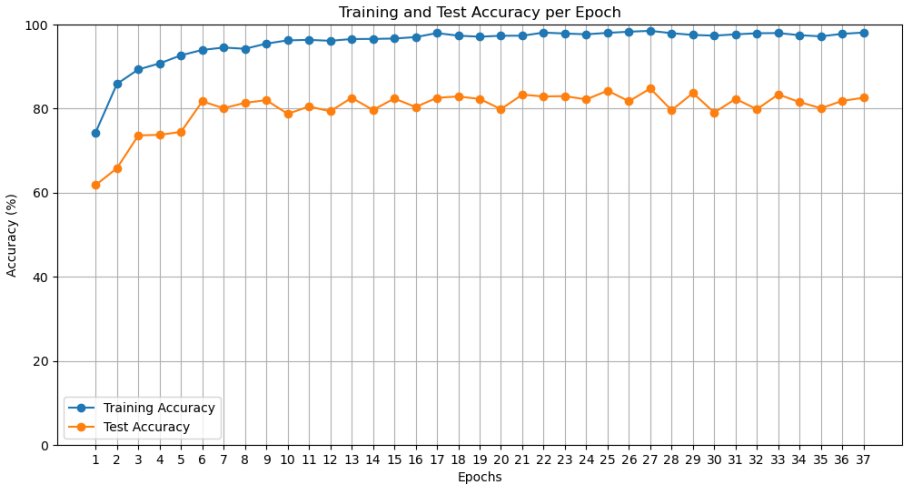


Figure 5: Training and Test Accuracy per Epoch

4 Object Detection

This task demanded the implementation of a convolutional network for object detection and instance segmentation. This example was done by using the Balloon dataset. A pretrained model (FasterRCNN ResNet50) was used to get better results and require less training time. The used dataset has already annotations of bounding boxes and masks and 74 images in total. The dataset was split into training, testing and validation (60%, 22%, 18%). Some images contain only one balloon, but there are also several images with multiple overlapping balloons.

The box predictor is set to a `FastRCNNPredictor` and the mask predictor is set to a `MaskRCNNPredictor`. Since there are only the balloons to detect, the number of classes to detect is two. The objects are either background (class 0) or a balloon (class 1). As data augmentation, a random horizontal flip was introduced, but no other transformation of the training data was applied.
The training was done for 20 epochs and with a batch size of two for better results. The results of the object detection and instance segmentation are presented in 6. In the results it should be noted that the score for object detection for false positives varies between 0.5 and 0.94 with the majority of false positives having a low score. The overall precision is 0.82, the recall is 0.86 and the f1-score is 0.81.
Further resarch could investigate the usage of different feature extraction layer, better data augmentation and more epochs.

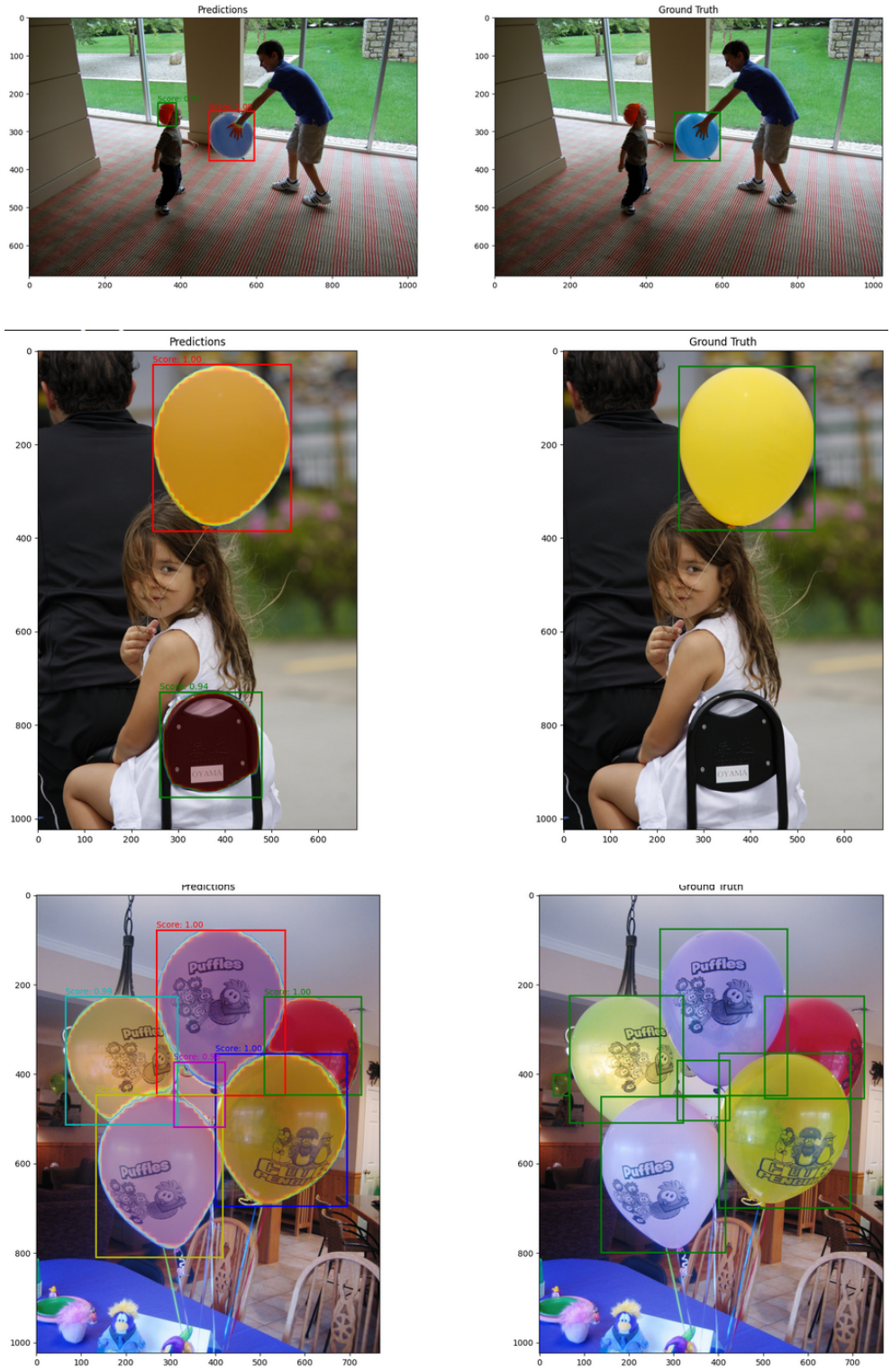


Figure 6: Results of object detection and instance segmentation

5 Recurrent Neural Networks

In this task a recurrent neural network was constructed with PyTorch. Based on a dataset about stackoverflow questions and answers, the model was trained and evaluated. The dataset had some limitations, that occurred during the training and evaluation. We preprocessed the data in several ways. At first, a stemmer was applied to the dataset. Afterward, the number of tags to classify was reduced. The total amount of tags was too high to keep all of them. We decide to only classify the forty most common tags. Nevertheless, the distribution of the tags is really uneven. More than 99% of the questions are tagged with the python tag. We decided to train the network with the question body. We also tried the question title and the results were roughly similar. The input text for the neural network was converted to numbers, but only the first thirty words of the question body were taken.
The chosen network has a simple architecture. The starting point can be seen in the code below. We then enhanced the architecture further to a more complex architecture to sort out some issues.

```
class Classifier(nn.Module):
    def __init__(self):
        super(Net2, self).__init__()
        self.embedding = nn.Embedding(len(uniquewords), 20)

        self.lstm = nn.LSTM(input_size=20,
                            hidden_size=16,
                            num_layers=1,
                            batch_first=True,
                            bidirectional=False)

        self.fc1 = nn.Linear(16,256)
        self.fc2 = nn.Linear(256, num_classes)

    def forward(self,inp):
        e = self.embedding(inp)
        output, hidden = self.lstm(e)

        x = self.fc1(output[:, -1, :])
        x = F.relu(x)

        x = self.fc2(x)
        x = torch.sigmoid(x)

        return x
```

The training was done with the whole dataset for five epochs. We finally reached an accuracy of 0.57, a precision of 0.82, a recall of 0.72 and a f1-score of 0.74.f

6 Autoencoders, Generative Adversarial Networks and Diffusion Models

The aim of this task was to develop three different models for removing systematic noise in the form of a "date stamp" from images: an autoencoder, a generative adversarial network (GAN), and a diffusion model. Specifically, for this task, we used the Food-11 dataset, which includes a wide variety of images depicting different types of food. Each image was modified with the addition of a "date stamp" using a script that randomly places a date in a random location. This added element serves as artificial systematic noise, allowing us to effectively test the models' capability in removing visual noise. In the following four sections, we will describe the data preparation process and provide a detailed explanation of the three models, presenting their architecture and analyzing their results.

6.1 Data Preparation

Before building the various models, we focused on data preparation. Specifically, we used the Food-11 dataset, which contains 16643 food images grouped into 11 major categories: Bread, Dairy Product, Dessert, Egg, Fried Food, Meat, Noodles/Pasta, Rice, Seafood, Soup, and Vegetable/Fruit. The images in this dataset vary in size, so the first preprocessing step is resizing each image to 256x256 pixels. Next, systematic noise is added in the form of a date stamp. A random date between January 1, 2000, and today is added to each image. To improve the models' performance and generalizability, the date appears each time in a random position with a random color. There are several ways to further enhance the data. Incorporating more advanced data augmentation techniques, such as rotation, scaling, and random cropping, would strengthen the model's robustness and enrich the diversity of the training data.

6.2 Autoencoder

The autoencoder is an unsupervised neural network used to learn an efficient data representation (encoding) that enables input reconstruction while removing unwanted noise. In this case, the model takes a noisy image as input and produces a clean output image, reducing noise through the encoding-decoding process. The architecture of the autoencoder consists of two main components: an encoder and a decoder. Specifically, the architecture we implemented in the code defines the model using the Autoencoder class, which leverages convolutional layers to compress and decompress image information. More specifically, the encoder is composed of three convolutional layers (nn.Conv2d), each followed by a ReLU activation function. Each convolutional layer progressively reduces the spatial dimensions of the image while increasing the channel depth. The layer specifications for the encoder are as follows:

1. **First layer:** from 3 channels (RGB input) to 64 channels, with a 3x3 kernel, stride of 2, and padding of 1.

2. **Second layer:** from 64 to 128 channels, with a 3x3 kernel, stride of 2, and padding of 1.
3. **Third layer:** from 128 to 256 channels, with a 3x3 kernel, stride of 2, and padding of 1.

This process allows the encoder to compress the image into a reduced latent representation, preserving essential information. The decoder uses three transposed convolutional layers (nn.ConvTranspose2d) to convert the latent representation back into a three-channel image with the original dimensions. Each transposed convolutional layer is followed by a ReLU activation function, except for the last layer, which uses a Sigmoid function to scale the output values between 0 and 1.

1. **First layer:** from 256 channels to 128 channels, with a 3x3 kernel, a stride of 2, padding of 1, and output padding of 1.
2. **Second layer:** from 128 to 64 channels, with a 3x3 kernel, a stride of 2, padding of 1, and output padding of 1.
3. **Third layer:** from 64 to 3 channels, with a 3x3 kernel, a stride of 2, padding of 1, and output padding of 1.

The *output_padding* function in the transposed convolutional layers allows the decoder to progressively restore the spatial dimensions of the image. This process enables the decoder to generate a reconstructed image, effectively reducing the original noise. The model architecture can be more clearly seen in the code snippet below:

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()

        self.encoder = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=2, padding=1), nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1), nn.ReLU(),
            nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1), nn.ReLU()
        )

        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1,
            output_padding=1), nn.ReLU(),
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1,
            output_padding=1), nn.ReLU(),
            nn.ConvTranspose2d(64, 3, kernel_size=3, stride=2, padding=1,
            output_padding=1), nn.Sigmoid()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

For the training process, we opted to use the Mean Squared Error (MSE) function as the loss function, which is well-suited to measure the average squared difference between the model's output and the clean target image. The Adam optimizer is employed with an initial learning rate of 10^{-4}, paired with a scheduler (specifically, StepLR) that decreases

the learning rate by a factor of 0.1 every 100 epochs. During each epoch, the model performs a forward pass on image batches, calculates the loss, backpropagates the gradient to update the weights, and adjusts the learning rate using the scheduler. At the end of each epoch, the loss and the current learning rate are printed to monitor training progress. The test function allows for evaluating the model on a single noisy image. The image is preprocessed, passed through the model, and the output is presented as a reconstructed, cleaned image. Below are three results obtained from the test dataset Figure 7.

The proposed autoencoder architecture is straightforward and well-suited for the task of image denoising. Nonetheless, as we observed in the test images, the final result is nicely satisfactory, even though the architecture used is relatively simple. Potential improvements could include:

- Adding more layers to increase the model's capacity.
- Enhancing the input and output image resolution by doubling the current size to 512 pixels, for more detailed results. However, to avoid significantly lengthening training times, we opted to use only 256 pixels.
- Introducing regularization techniques to prevent overfitting.
- Exploring alternative loss functions, such as the Structural Similarity Index (SSIM), to achieve greater sensitivity to visual quality.

6.3 Generative Adversarial Network

The second model we implemented is a Generative Adversarial Network (GAN). GANs are neural networks composed of two competing models, a generator and a discriminator, which learn to generate realistic images from noisy inputs. The generator creates cleaner images from noisy versions, while the discriminator evaluates the authenticity of the generated images in comparison to the original ones. During training, the generator and discriminator are updated in alternating steps: the generator attempts to deceive the discriminator by producing increasingly realistic images, while the discriminator learns to more accurately identify subtle differences between real and synthetic images. This competitive dynamic drives both models to gradually improve in a process known as adversarial learning. As training progresses, the system becomes increasingly sophisticated, enabling the generator to produce noise-free images that even the discriminator struggles to distinguish from the originals. As mentioned, the GAN architecture consists of two distinct neural networks: the generator and the discriminator. Their specific architectures are detailed below.

Generator

1. The first three convolutional layers, with a 3x3 kernel and stride of 2, progressively reduce the spatial dimensions of the input.
2. Two transposed convolutional layers with output padding of 1 restore the image to its original spatial dimensions.

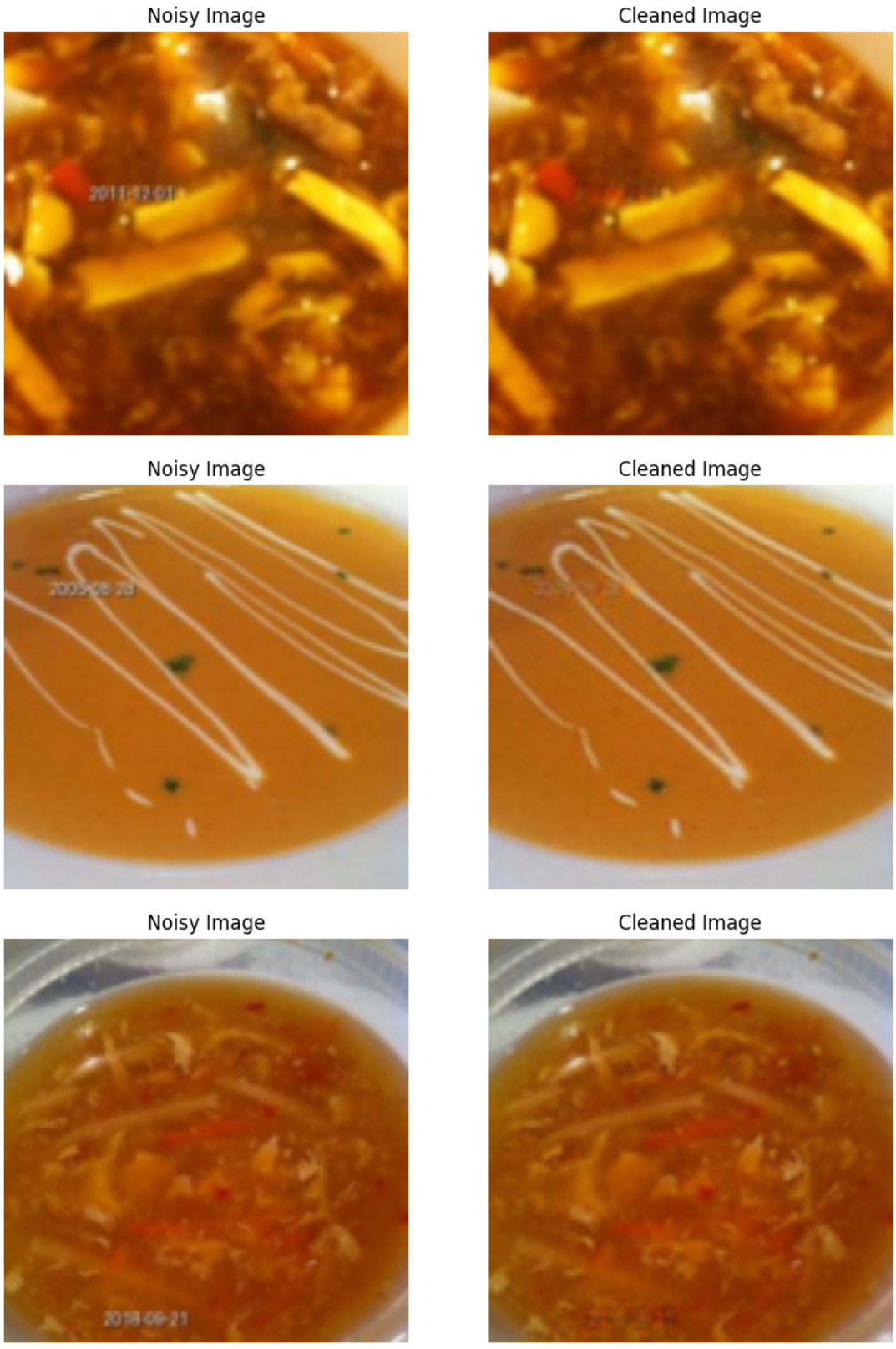


Figure 7: Comparison of Noisy and Denoised Images using Autoencoder

3. A final convolutional layer, followed by a ‘Tanh‘ activation function, scales the output image values between -1 and 1, adapting to a normalized pixel representation.

The idea behind this architecture is that the initial layers of the generator, composed of convolutional layers, are designed to extract and compress the visual features of the input image. This sequence of convolutions progressively increases the number of channels from 3 (RGB colors) to 256 while reducing spatial dimensions through a stride of 2 in the second and third convolutional layers. These initial layers capture the key features of the image (such as textures, edges, and details) into a more compact yet information rich latent representation. Following this, the transposed convolutional layers reconstruct the spatial dimensions of the image, gradually increasing the resolution to produce an output with the same dimensions as the input. Each transposed convolution layer has a stride of 2 and uses output padding of 1 to ensure the image is reconstructed to its original size. This operation, the inverse of standard convolution, effectively "expands" the latent representation, restoring detail to the output image. These layers not only return the image to its original resolution but also learn to generate the visual details that make the image appear realistic and credible. Finally, the last convolutional layer reduces the channels back to 3 (RGB), producing a color image with the same dimensions as the input. This final step generates a denoised and normalized image that the discriminator will evaluate against the original image.

```
    class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1), nn.ReLU(),
            nn.Conv2d(64, 128, kernel_size=3, stride=2, padding=1), nn.ReLU(),
            nn.Conv2d(128, 256, kernel_size=3, stride=2, padding=1), nn.ReLU(),
            nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1,
            output_padding=1), nn.ReLU(),
            nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1,
            output_padding=1), nn.ReLU(),
            nn.Conv2d(64, 3, kernel_size=3, stride=1, padding=1), nn.Tanh()
        )

    def forward(self, x):
        return self.model(x)
```

Discriminator

The discriminator's goal is to learn key visual features that help it recognize subtle differences between authentic and synthetic images. Specifically, the initial convolutional layer, with a 4x4 kernel and a stride of 2, progressively reduces the spatial dimensions of the image while increasing the number of channels to capture more complex and detailed features. This layer has 64 filters, which progressively increase up to 256 in the following layers, enabling the model to extract high-level representations. Each convolutional layer, except the last one, is followed by a LeakyReLU activation function, allowing gradient backpropagation even for negative inputs and enhancing the model's ability to learn fine details within images. Additionally, the intermediate convolutional layers are paired with a BatchNorm2d function, which normalizes the output of the previous layer. This normalization improves training stability by counteracting vanishing or exploding gradient issues, allowing the discriminator to learn more effectively. Batch normalization also accelerates the training process, making the model less sensitive to batch variations and improving its ability to generalize. Finally, the last convolutional layer reduces the channel count to 1, outputting a single value per image that represents the probability of the image being real or generated. The final activation function is a Sigmoid, which scales

the output between 0 and 1, interpreted as a probability. An output close to 1 indicates a high likelihood that the image is real, while an output close to 0 suggests a generated image.

```
    class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1), nn.LeakyReLU
            (0.2),
            nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(128), nn.LeakyReLU(0.2),
            nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(256), nn.LeakyReLU(0.2),
            nn.Conv2d(256, 1, kernel_size=4, stride=1, padding=0), nn.Sigmoid()
        )

    def forward(self, x):
        return self.model(x).view(-1)
```

The "train_gan" function implements the training loop for the GAN, alternating optimization steps for the generator and discriminator. Binary Cross Entropy (BCE) is used as the loss function for both networks, measuring the difference between predicted and actual labels. Real labels are scaled to 0.9 (instead of 1) to reduce overconfidence in the discriminator's accuracy, while labels for generated images are set to 0.1 (instead of 0) to prevent the discriminator from becoming overly dominant. The Adam optimizer is used for both models with an initial learning rate adjusted through "StepLR", which decreases the learning rate every 100 epochs to encourage more stable convergence. The discriminator is updated twice for each generator update to improve GAN stability. At the end of each epoch, the generator and discriminator losses and current learning rates are printed to monitor training progress. Below are the images generated by the model discussed above Figure 8.

To conclude, the implemented GAN architecture provides an effective solution for image denoising, where the generator and discriminator learn in a competitive, adversarial setup. Potential future improvements include:

- Adding Residual Structures in the generator to enhance the quality of generated images.
- Implementing Stabilization Techniques, such as gradient penalty, to improve training stability.

6.4 Diffusion Model

Diffusion models are a class of deep generative models that have shown remarkable results in data generation, particularly for images. Well-known models based on this approach include DALL-E, Midjourney, and Stable Diffusion. These models operate through a "diffusion" process, in which noise is gradually added to a given input until the information distribution becomes nearly random. The model then learns to reverse this process, recovering the original data from the noisy version or generating new, realistic samples

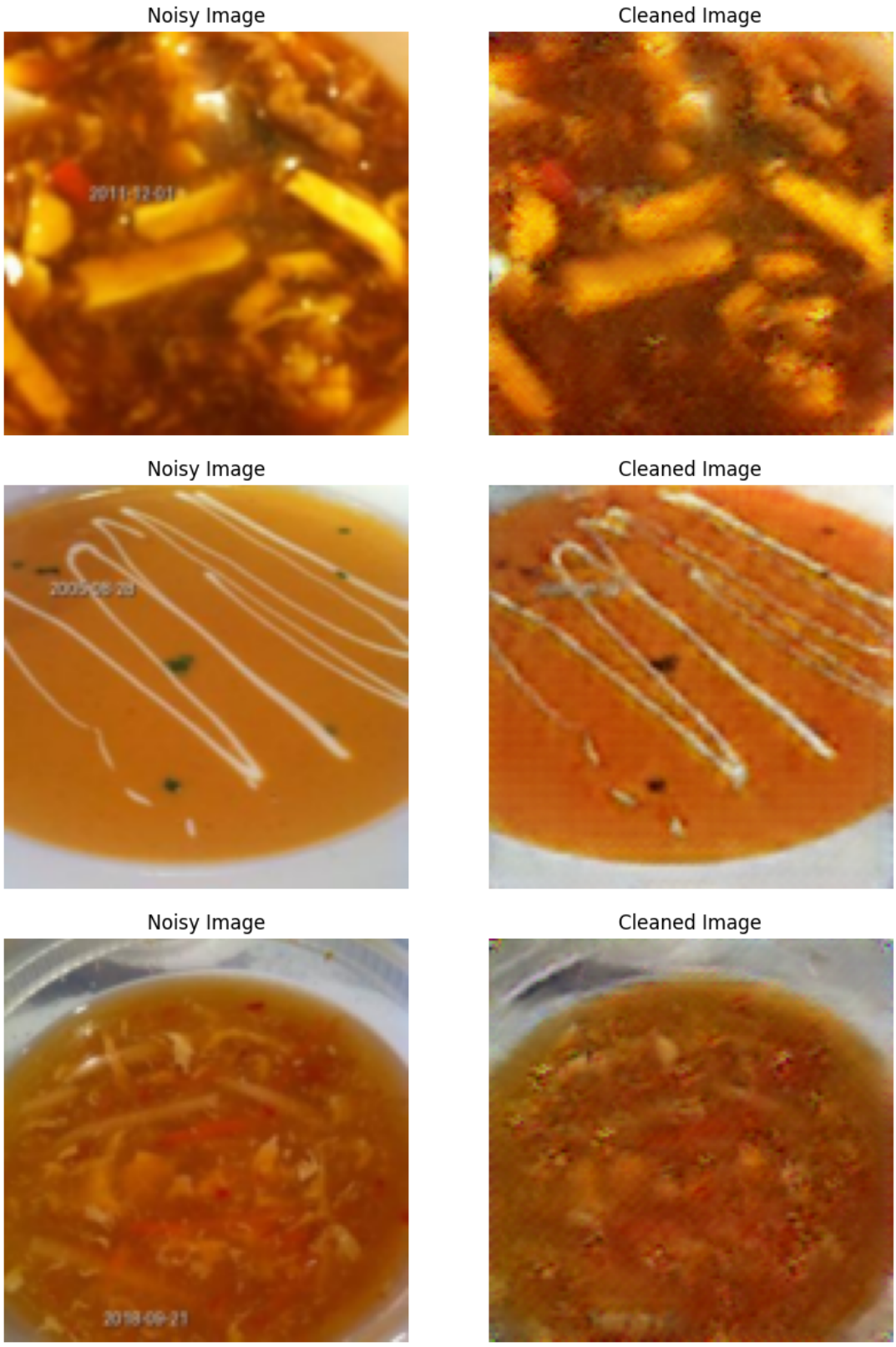


Figure 8: Comparison of Noisy and Denoised Images using GAN

starting from pure noise. The core principle of diffusion models involves simulating a stochastic process where noise is incrementally added to an image (or data) across multiple steps, progressively degrading its content. During training, the model learns to reverse

each of these noisy steps, ultimately reconstructing a noise-free image. An innovative aspect of diffusion models is that, unlike other generative architectures like GANs, these models are more stable during training and less prone to divergence issues. Thanks to the gradual nature of this process, diffusion models are well-suited to learn robust data representations, making them ideal for a variety of applications—from synthetic image generation to noise or artifact removal, such as removing unwanted overlays or date stamps from images, as in our case, to controlled manipulation of generated content. Ultimately, diffusion models provide a powerful and versatile framework for generative modeling, balancing high-quality sample generation with training stability, and are proving to be impactful tools in practical and research applications across fields like computer vision, audio synthesis, and synthetic data generation. The architecture we chose to implement is a simplified U-Net, a deep convolutional network commonly used for segmentation and denoising tasks. The U-Net architecture follows an encoder-decoder structure, with convolutional layers to extract features and upsampling layers to reconstruct the image at its original resolution.

Encoder

The encoder extracts features from the input image, reducing its resolution through convolution and pooling operations. In this model, the encoder is structured as follows:

1. First convolution block: two convolutions with 64 filters each, followed by a ‘ReLU‘ activation function.
2. Pooling: a ‘MaxPool2d‘ operation reduces the spatial dimension, halving the resolution while preserving key features.
3. Second convolution block: two convolutions with 128 filters, each followed by ‘ReLU‘.

Decoder

The decoder uses transposed convolutions (also known as deconvolutions) to restore the image to its original resolution and remove noise. This includes:

1. Upconvolution: a transposed convolution that reduces the number of filters (from 128 to 64) and restores the resolution to the same level as the output of the encoder’s first block.
2. Concatenation and decoding: concatenation of the decoder’s features with those of the encoder’s first block to preserve low-level details, followed by two convolutions with 64 filters and ‘ReLU‘ activations.
3. Final layer: a 1x1 convolution that reduces the number of channels to 3 to match the RGB output.

The code for the network is attached below.

```
    class UNet(nn.Module):
    def __init__(self):
        super(UNet, self).__init__()
        # Encoder
```

```
        self.enc1 = nn.Sequential(nn.Conv2d(3, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.pool = nn.MaxPool2d(2, 2)
        self.enc2 = nn.Sequential(nn.Conv2d(64, 128, 3, padding=1), nn.ReLU(),
        nn.Conv2d(128, 128, 3, padding=1), nn.ReLU())

        # Decoder
        self.upconv1 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.dec1 = nn.Sequential(nn.Conv2d(128, 64, 3, padding=1), nn.ReLU(),
        nn.Conv2d(64, 64, 3, padding=1), nn.ReLU())
        self.final = nn.Conv2d(64, 3, kernel_size=1)

    def forward(self, x, t=0):
        # Encoder
        e1 = self.enc1(x)
        e2 = self.enc2(self.pool(e1))

        # Decoder
        d1 = self.upconv1(e2)
        d1 = torch.cat([d1, e1], dim=1)
        d1 = self.dec1(d1)
        return self.final(d1)
```

During the training phase, the model was trained over 400 epochs using a Mean Squared Error (MSE) loss function, which measures pixel-by-pixel similarity between the model's output and the clean reference images. The Adam optimizer was employed to adjust the model's weights. The learning rate was initially set to 10^{-4} and then adjusted every 100 epochs with a StepLR scheduler, reducing it by a factor of 0.1 to encourage convergence in the later stages of training. The training process showed a consistent reduction in loss, indicating that the model was effectively learning to remove noise from the images and improve output quality with each epoch. Finally, results on the same test images confirm that this model outperformed the autoencoder model and the GAN model, achieving remarkably realistic noise free outputs closely resembling the original images without the date stamp. Importantly, this level of performance was achieved with less training time than the two previous models, highlighting that diffusion models represent the state of the art in generating high fidelity images. This is clearly demonstrated in the test images produced by the model, shown below Figure 9.

To conclude, the implemented diffusion model has proven to be highly effective, as it is the best performing model among the three tested, producing the highest quality images while achieving this level with the shortest training time. However, some possible future improvements include:

- Use of a more complex network: The implemented UNet is a simplified version without specialized features. The model could be made deeper, with additional BatchNorm2d layers after each convolutional layer to help stabilize and accelerate the training process. Adding dropout layers in the intermediate levels of the encoder and decoder could further improve the model's generalization ability, reducing the risk of overfitting.
- Customized loss function: Consider a loss function focused on structural similarity, such as SSIM, to better emphasize noise removal while preserving image quality.

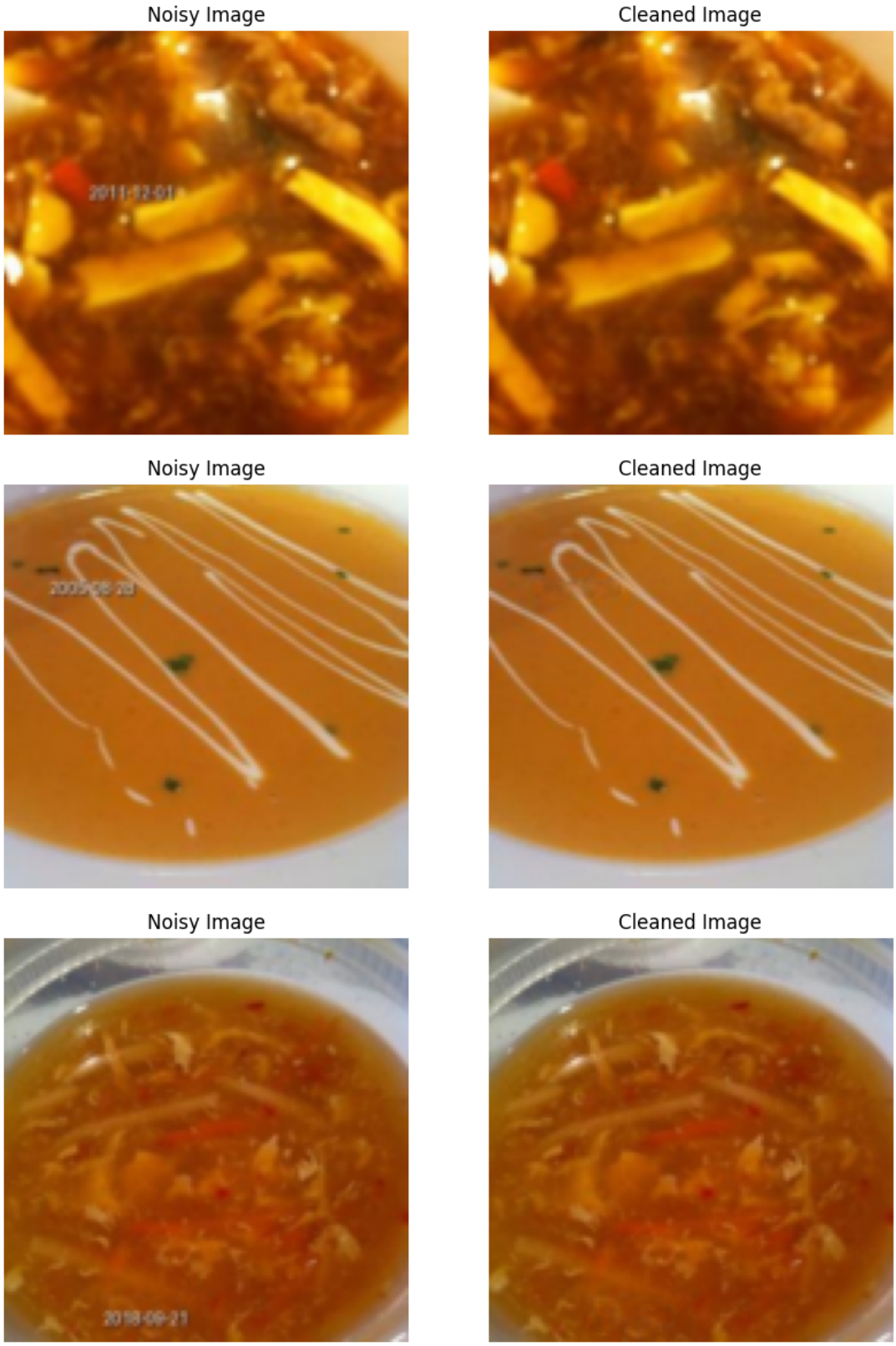


Figure 9: Comparison of Noisy and Denoised Images using Diffusion Model

7 Project

7.1 Introduction

As part of this project, various deep learning models were developed with the objective of enabling the automatic recognition and differentiation of bird calls. The analysis of bird calls represents a fundamental approach within the field of ecological research and nature conservation [1]. Birds perform vital functions in ecosystems, including the dispersal of seeds, the control of pests and the pollination of plants. Moreover, fluctuations in avian populations serve as crucial indicators of biodiversity and environmental change [2].

In light of the challenges posed by climate change and the promotion of renewable energies, the monitoring of bird populations with precision is becoming an increasingly crucial endeavour. In particular, wind turbines present a significant risk to bird species, which underscores the necessity for strategic site planning. However, traditional monitoring methods are becoming increasingly inadequate due to their high time and labour costs [3].

The utilisation of contemporary deep learning methodologies facilitates the provision of an efficacious and scalable alternative, which has the potential to transform the field of bird monitoring and significantly advance research into biodiversity and ecological processes. The objective of the project is to assess the efficacy of machine learning techniques and to provide a contribution to the development of sustainable conservation solutions.

The development of a model for determining bird calls was initiated with the data set of the LifeCLEF 2020 Bird Monophone Challenge[1], which was launched in 2020. The objective was to develop a model for the recognition of bird calls. This comprehensive database comprises bird calls from disparate regions, as contributed by the community-driven platform Xeno-Canto[2]. The data is only weakly annotated, yet encompasses a considerable variety of species and acoustic environments, rendering it an optimal resource for the development and testing of deep learning models.

[1] `https://www.aicrowd.com/challenges/lifeclef-2020-bird-monophone`
[2] `https://xeno-canto.org`

7.2 Methodology

The following section will give a brief introduction into the used dataset. The selected architecture and models will be described in the following sections.

7.2.1 Dataset

The dataset comprises 72,307 bird voice recordings, pertaining to 960 species. The original audio files have a total size of 64GB. Prior to training any architecture, the audio data must be split into segments of equal length. Two distinct approaches were employed.

The first approach to preprocess the data was to precompute the audio data. The audio data was read and split into segments of equal size. Additionally, an overlap of 50% was applied to the segments. This approach resulted in various problems. The amount of the resulting data was too huge for the provided server from the University of Agder (UiA). The JupyterServer restarted randomly. Batch processing, parallel processing and the usage of special libraries (dask[3]) for huge datasets were tried to overcome the issues. But with an overlap of 50% the resulting data was still too big for loading it into one data loader. Because of these issues, another approach was investigated.

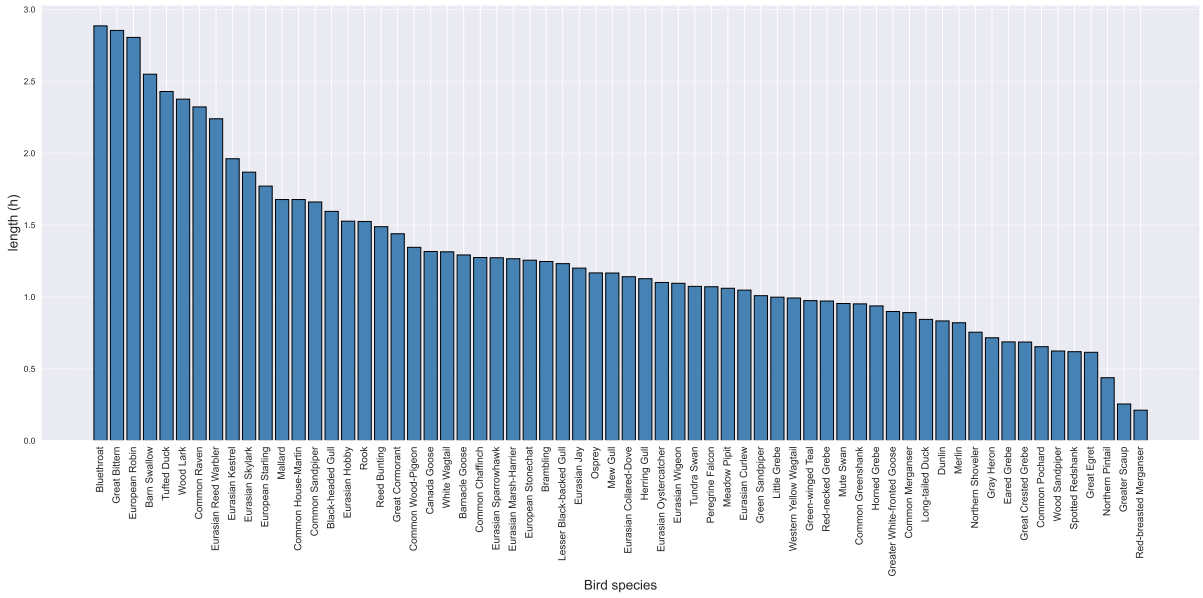
In order to overcome the aforementioned computational issues, only a subset of the original dataset was utilized. The subset comprises the most commonly occurring species of birds in Germany. The subset comprises 64 species with a total of 6,040 recordings (the distribution of the recordings by bird can be seen in Figure 10b). The total duration of the recordings included in the subset is 82 hours (the distribution of the total duration by bird can be seen in Figure 10a).

The second approach results from the issues and problems of loading all data once into the memory of the provided server. Therefore a streaming dataloader was used. It is balanced over all classes, so that the different length of audio data per class are weighted. As shown in Figure 10a the length of the available recordings differ a lot per class. To have no difference in the training, the streaming dataloader ensures the equal usage of every class. It selects random items from classes. Within a class, an overlap of 50% is also applied to all segments.

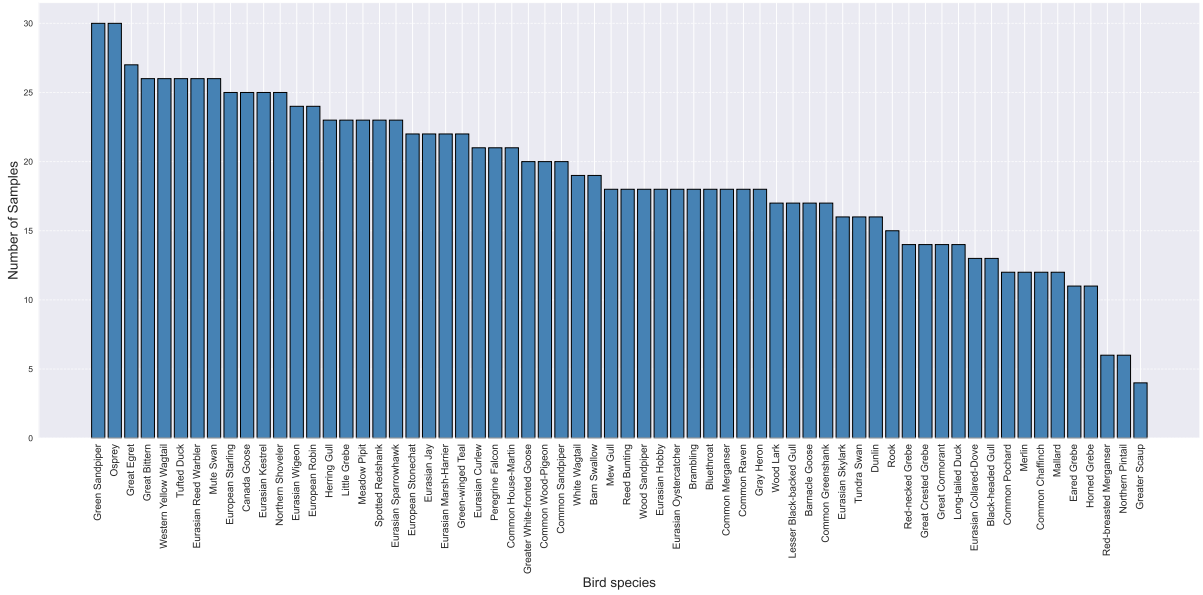
A key challenge in the analysis and classification of bird calls lies in the availability of suitable training data. Many bird species avoid the proximity of humans, which makes it difficult to record their calls in a targeted manner. As a result, most audio recordings contain not only the target calls, but also background noise or calls of other bird species, which can affect the quality of the data and the accuracy of the classification. Also, as already mentioned, the available recording length differ per class (see Figure 10a). This needs to be taken into consideration, that every bird specie is trained to the same extent.

Another problem arises with the choice of segment length. If longer segments are used, the probability that several bird species can be heard in one section increases. Bird calls from the background can be recognized by the model as dominant, even if the label of the segment actually describes a different species. This makes clear assignment to a class more difficult and can lead to misclassification. On the other hand, if the segment length

[3] `https://docs.dask.org/en/stable/`



(a) Bird Recording Lengths



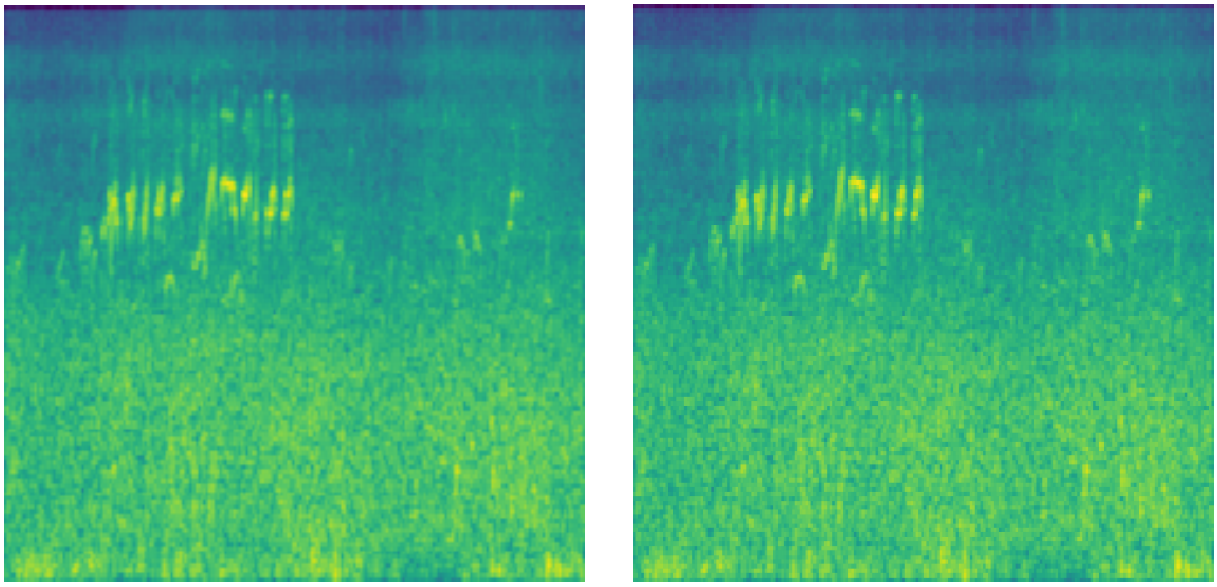
(b) Trainingsdataset number of samples

Figure 10: Overview of bird voice data: (a) Bird Recording Lengths and (b) Number of Samples in the training dataset.

is too short, species with long pauses between their calls - such as birds of prey - may have segments extracted that do not contain an actual bird call. Instead, these segments may only contain silence or noise, but are still incorrectly assigned to the corresponding species. Furthermore, the classification of birds with calls longer than the segment length is challenging due to the limited length of the segments, although this issue is typically confined to the shortest segments.

For CNNs, data transformation follows a different process, converting audio files into mel spectrograms. This involves a series of steps that transform acoustic signals into visual representations better suited for analysis by CNN models. Initially, the audio files are

loaded and normalized in terms of sampling rate (22,050 Hz) and maximum duration (5 seconds). This step ensures the standardization of audio data, facilitating comparisons across samples of different origins and qualities. Subsequently, a mel spectrogram is computed, which converts the audio signal into a time-frequency representation compressed on a logarithmic scale. This transformation highlights features critical for sound or voice recognition. During this process, the mel spectrogram values are further converted to a decibel scale to enhance the visibility of intensity variations. The results are then saved as images, removing axes and other graphical information to focus solely on spectral features. These images are saved in a standard format (248x246 pixels), making them suitable for input into convolutional neural networks. For the training of a technically superior architecture, a new dataset of images with higher resolution and quality was created to assess whether including more information in the spectrograms would improve model performance. This new dataset consists of spectrograms with dimensions of (697x693 pixels). Below, you can observe examples of both versions of the mel spectrogram: the first with lower resolution (Figure 11a) and the second with higher resolution (Figure 11b).



(a) The mel spectrogram 248x246

(b) The mel spectrogram 697x693

Figure 11: Comparison of mel spectrograms at different resolutions.

7.2.2 Models

In this project, three distinct methodologies were developed and compared to evaluate their performance in bird call classification. The first approach involved the use of a pretrained model specifically designed for bird voice recognition. This model, fine-tuned for bioacoustic applications by the HuggingFace community, was assessed for its ability to classify bird calls directly from raw audio data. The second approach builds upon the Transformer architectural framework and utilizes the *wav2vec2.0* model, originally developed by Facebook for speech processing. Transformer-based models, while still relatively experimental in bioacoustics, have shown promise in recent studies. For example, Rauch, Schwinger, Wirth, *et al.* explored a similar methodology, leveraging *wav2vec2.0* for bioacoustic tasks [3].

The third approach employs a Convolutional neural network (CNN) to process spectro-

grams generated from audio data. This method, similar to the architecture used in the popular BirdNET application [1], remains a dominant choice in animal voice recognition tasks [4]. CNNs have demonstrated their effectiveness in various bioacoustic studies, including those by Jeppe Have Rasmussen. His work has extensively focused on the automatic detection and classification of animal vocalizations, such as baleen whale calls [5]. During the development of our CNN-based approach, we were fortunate to draw upon Rasmussen's expertise through a video call, where he provided valuable insights and practical tips for optimizing CNN architectures. His guidance, along with his extensive research, underscores the potential of CNNs for bird song classification.

Each approach offers unique strengths. While the CNN-based method benefits from established practices in animal voice recognition, the transformer-based wav2vec2.0 model introduces the potential for innovative processing of raw audio data. These methodologies are further detailed and evaluated in the subsequent sections of this report.

7.2.3 Wav2Vec2.0 model

As a foundation for our work, two models were utilized: the `facebook/wav2vec-base-960h` model and a fine-tuned version of the same model, `dima806/bird_sounds_classification`. The latter is specifically adapted for bird sound classification, having been fine-tuned on a dataset of 50 bird species over 10 epochs[4]. Both models were further fine-tuned on our custom dataset, which consists of recordings from 64 bird species native to Germany.

The principal benefit of the `facebook/wav2vec-base-960h` is its capacity to attain robust outcomes in speech processing tasks even with minimal training data. The question is whether this strength can also be applied to the classification of bird sounds, especially for more than 50 bird species native to Germany. This question will be examined in the following sections, as explored by the Huggingface user *dima806*.

In order to classify bird calls, Wav2Vec 2.0 employs a feature encoder that converts raw audio into latent representations utilising convolutional neural networks. The convolutional neural networks analyse short audio segments with a receptive field of 25 ms and shift by 20 ms, thereby creating overlapping frames that are capable of capturing fine temporal details [6]. The raw audio is initially normalised to a mean of zero and a variance of one, thereby ensuring consistent input quality. Subsequently, the latent representations generated by the convolutional neural network are conveyed to a transformer, which processes contextual information by modelling dependencies across the entire sequence. The transformer is capable of integrating time segments of varying lengths through self-attention mechanisms, which is crucial for capturing the diverse durations and structures of bird calls. In contrast to applications in speech recognition, no quantisation is applied here, as feature discretisation is not necessary for classification. Instead, the focus is on leveraging the continuous latent representations to effectively classify bird vocalisations based on their unique temporal and spectral patterns.

Accordingly, the initial duration of the audio segments was set to 5-10 seconds, depending on the training sessions. The model was trained with a batch size of 32.

Furthermore, approaches were also pursued to use other models, such as OpenAi's *Whis-*

[4] `https://www.kaggle.com/datasets/soumendraprasad/sound-of-114-species-of-birds-till-2022`

per, but here it turns out that the preset of the audio files must always be 30 seconds. In contrast to the Facebook *Wave2vec2.0* model, *Whisper* also works internally with specograms [7].

7.2.4 Pre-trained model with silence

The foundation for the model is the already mentioned `dima806/bird_sounds_classification`. This model is a fine-tuned version of the `facebook/wav2vec-base-960h` model. One additional class with silence was added to the model. To generate silence, two different approaches were tried. The first one was the generation of silent section on the base of all bird voice recordings. The silent segments should be cut out of the recordings. The advantage of this method would be the existence of the same soundscape as in the recordings. The background noises could remain. Unfortunately, a satisfying result was not achievable. With a silence detection algorithm, there were still too many bird voice artefacts in the silence segments. Finally, the silence was just randomly created white noise.

7.2.5 Convolutional Neural Network

Building upon the insights from the paper "BirdNET: A Deep Learning Solution for Avian Diversity Monitoring" [1], we decided to implement different Convolutional Neural Networks (CNNs) to classify the mel spectrograms derived from audio recordings of bird calls. The paper presents the first version of BirdNET, based on `ResNet`, followed by a subsequent implementation using `EfficientNetB0`, whose architecture is shown in the following page. With this context, our approach was to initially implement two models: a ResNet inspired model built from scratch and an EfficientNetB0 model, to compare their performance. The ResNet like model from scratch yielded poor results, with an average classification accuracy below 50%. Conversely, the EfficientNetB0 model delivered significantly better results, even with less training time. This initial phase of constructing and experimenting with different models was crucial to identify the most suitable architecture for the task. We decided to focus exclusively on prebuilt models, with particular emphasis on the EfficientNet family of CNNs. Additionally, in 2021, the paper "EfficientNetV2: Smaller Models and Faster Training" [8], introduced new additions to the EfficientNet family, featuring fewer parameters and enhanced training efficiency. These advances provided valuable insights that helped us refine our approach and achieve better performance. For the implementation of the two EfficientNet models, we used the open-source library compatible with TensorFlow: `Keras`. This library allows the import of pre-developed architectures that have been pre-trained on ImageNet. To address our classification problem, we focused on two specific models, `EfficientNetV2M` and `EfficientNetV2L`, as indicated in the Keras documentation, which highlighted their excellent performance in classifying images across more than 1000 different classes.

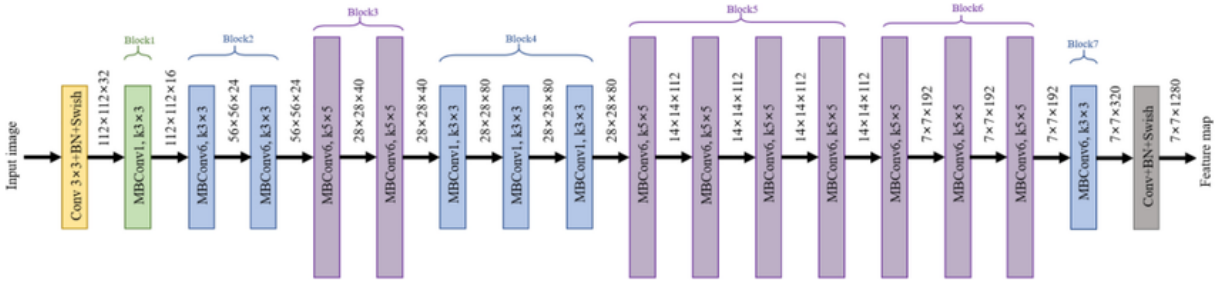


Figure 12: The architecture of the EfficientNetB0 CNN

7.3 Training

7.3.1 Transformers

For training, we fine-tuned the `facebook/wav2vec2-base-960h` model over 100 epochs with a learning rate of 1×10^{-5}, optimized using Adam. The loss function was categorical cross-entropy, suitable for multi-class classification tasks. A linear learning rate scheduler with warm-up was employed to stabilize convergence during training. Evaluation metrics such as accuracy, precision, recall, and F1-score were used to assess the model's performance.

7.3.2 Convolutional Neural Networks

Regarding the training methodologies employed, we initially experimented with training each model on a subset of data confined to Central Europe to determine the most effective training approach. Our first attempt involved fine-tuning the EfficientNetV2M network, pre-trained on ImageNet. However, this approach did not yield the expected results. As a result, we decided to unlock all layers of the network and train it from scratch using our new dataset. This second approach delivered excellent results on the European data subset, achieving an accuracy of 94%.

Building on this success, we applied the same method to subsequent models. Among these, the best-performing model was trained for three days on the complete dataset. We used Adam as the optimizer and selected categorical crossentropy as the loss function. This loss function is particularly suitable for multi-class classification problems, as it effectively measures the distance between the predicted probabilities and the true labels. It penalizes the model for assigning incorrect probabilities to the correct class and encourages maximizing the likelihood of the correct class.

To further enhance training performance, we implemented learning rate reduction techniques. When the model showed no improvement over a certain number of epochs, the learning rate was reduced to prevent stagnation in learning. Additionally, if the learning rate reached a predefined minimum threshold, training was stopped, as this indicated that a satisfactory local minimum had been found.

All model training and testing processes were carried out on the JupyterLab server of the University of Agder, leveraging the Nvidia A100 GPU to optimize computational performance.

7.4 Results

Three main different models based on wav2vec2.0 are compared in the following section and finally, the results of the EfficientNetV2M CNN are presented. The first is the plain pre-trained bird model. The second is the extension of the pre-trained model with one additional class 'silence'. Directly based on facebooks wav2vec2.0 is our third wav2vec2.0 model.

7.4.1 Evaluation of the pre-trained model

The pre-trained model reaches an accuracy of 85%, a precision of 88%, a recall of 85% and a f1-score of 86%. As shown in Figure 13 the results are quite well. The confidence of the correct predicted species is quite high. The confidence value of the wrong predicted birds are quite low. With an additional algorithm cutting the prediction at a certain threshold, superior results are to expect.
The dependence of the precision to the bird species is shown in Figure 14. Some species are predicted with a low prediction. But no correlation between the precision and the length of the recordings could be taken. The length of the recordings per bird species was shown in Figure 10a. For some species, possible reasons for the worse performance could be found. For example, the Common Raven is known to produce a very high variability of calls in dependence if they know other individuals, their situation and other environmental conditions [9]. They are known for imitating sounds of their environment [9].

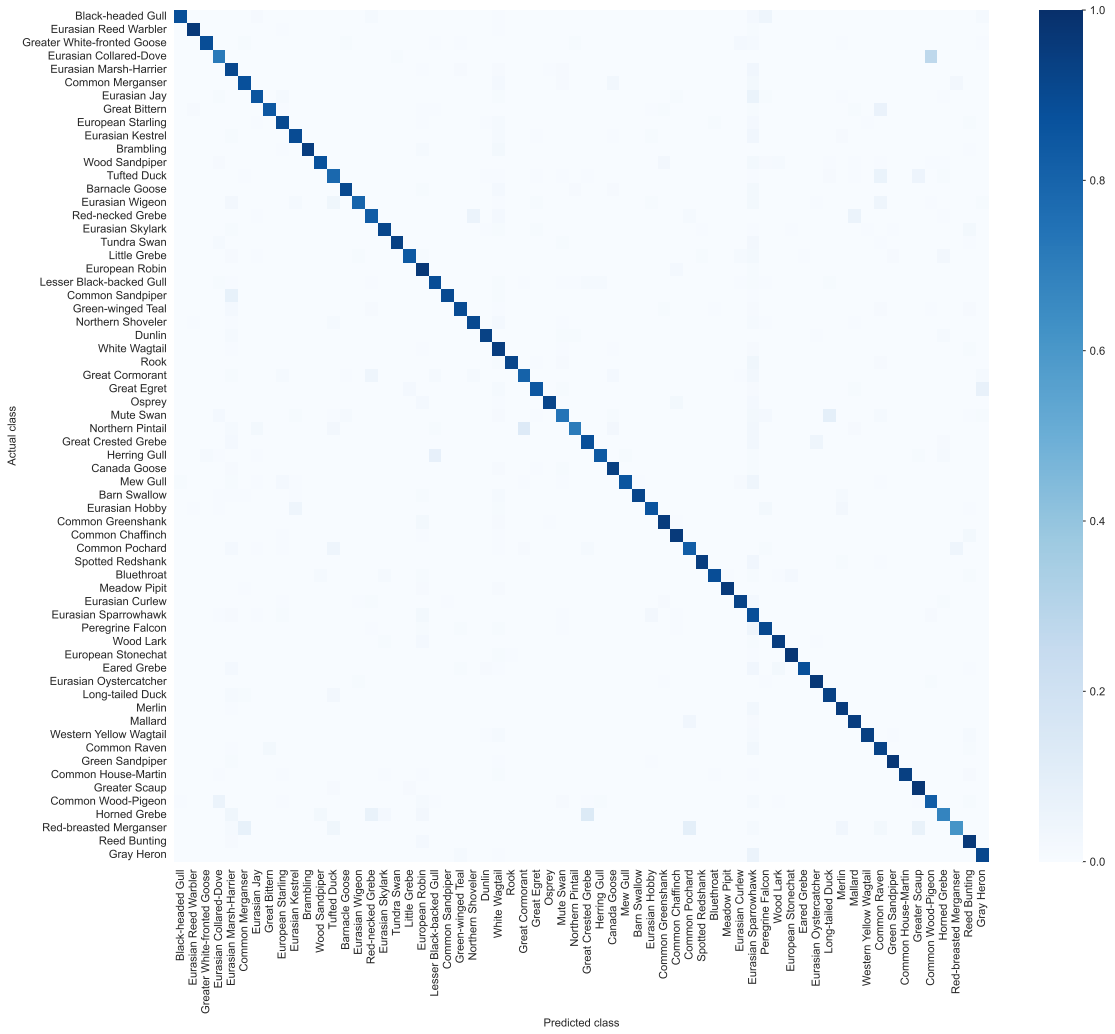


Figure 13: *Pre-trained bird model* - weighted Confusion Matrix (with confidence)

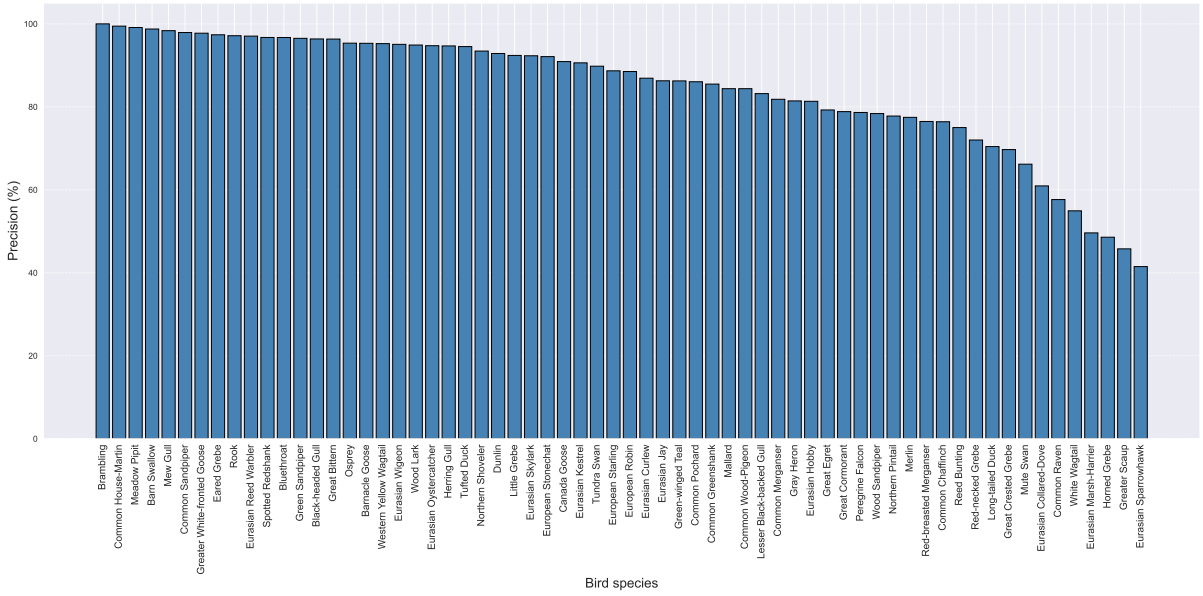


Figure 14: *Pre-trained bird model* - precision as a function of the bird species

7.4.2 Evaluation of the wav2vec2.0 model

The wav2vec2.0 reaches an accuracy of 78% and a precision of 79% as shown in Table 1 Additionally, the length of the test segments affect the performance metrics. As illustrated in Table 1 the length of the testing segments influences the accuracy, precision, recall and the f1-score. The optimal length is laying between five and fifteen seconds. The length of segments needs to match the bird call length. Also, the silence between the separate bird calls needs to be taken into consideration. If a call happens only every three to five seconds, a classification with a segment length of only three seconds is less accurate.

Furthermore, the precision of the prediction differs from bird specie to bird specie. The different bird specie result in different precision values in dependence of the segment length. These results are show in Figure 15

Table 1: Results for different testing segment lengths - Model *wav2vec2.0*

Segment length	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1 Score (%)**
3 Seconds	68.89	70.55	68.89	68.67
5 Seconds	75.64	76.30	75.65	75.11
10 Seconds	78.55	79.87	78.55	78.11
15 Seconds	77.84	79.47	77.84	77.49
30 Seconds	72.48	75.81	72.48	72.44

7.4.3 Model with silence

To address the issue of misclassified silent segments, the model with silence was introduced. An additional class for randomly generated white noise was added to explicitly account for silence and background noise.

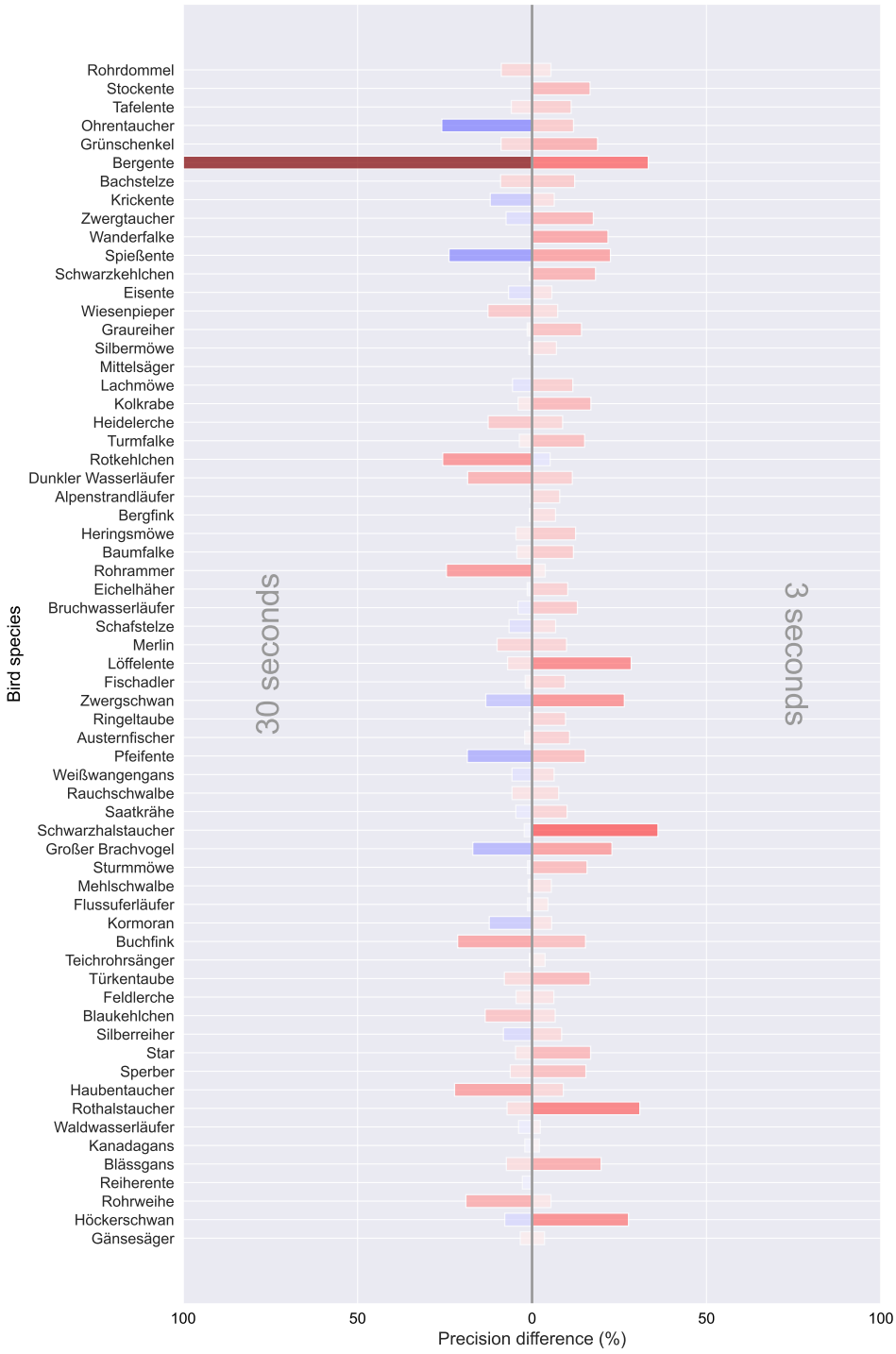


Figure 15: The following chart presents a comparison of the precision values of the *wav2vec2.0* model for varying segment lengths. The differences between the 30-second and 10-second segments are displayed on the left side of the Y-axis, while the differences between the 10-second and 3-second segments are displayed on the right side. A red bar indicates that the precision for the 10-second segment was higher than that for the other segment length, whereas a blue bar signifies that the other segment length demonstrated superior precision to that of the 10-second segment.

However, as shown in Figure 17 for quite some species, performs the model with silence better than the pre-trained model. The White Wagtail, the Common Raven, the Horned

Grebe and the red-breasted Merganser are better predicted from the model with silence. Some of the recordings of these species have a high amount of silent or noisy segments. On the other hand, the characteristics of the bird calls of the Eurasian Sparrowhawk are quite similar. We could not find a relation between the characteristics of different birds and the precision of the different models. However, as seen in Table 2, the overall metrics of the model without the silence class remained superior. As shown in Figure 13 the pre-trained bird model has an acceptable accuracy for all different species. The confidence differs from specie to specie.

7.4.4 EfficientNetV2

The classification model developed using EfficientNetV2M to process Mel spectrogram images has achieved outstanding results, with an accuracy of 90.04%, precision of 88.50%, recall of 91.20%, and an F1 score of 89.86%. These values indicate that the model correctly classifies the vast majority of images, demonstrating excellent generalization capability. The balanced precision and recall suggest that the model is highly effective in distinguishing classes without significant errors, while the high F1 score confirms the overall robustness of the system. This success is attributable to the combination of EfficientNetV2M, a deep and efficient network, with the powerful acoustic representation offered by Mel spectrograms. Despite the remarkable performance, an analysis of residual errors could help identify potential issues, such as classes that are harder to distinguish or acoustic interferences. Furthermore, evaluating performance on data collected in different environmental conditions would further validate the model's generalization ability. Finally, these results pave the way for numerous practical applications, such as automated environmental monitoring, offering a significant improvement in reliability compared to previous models. The following image displays the confusion matrix for the first 50 classes analyzed by the model during the testing phase (Figure 16).

7.4.5 Comparsion of the models

Table 2: Comparison of evaluation metrics - **10 seconds length of test segments**

Model	**Accuracy (%)**	**Precision (%)**	**Recall (%)**	**F1-Score (%)**
EfficientNetV2M	90.04	88.50	91.20	89.86
Pre-trained bird model	85.40	87.95	85.40	86.00
Model with Silence	81.13	85.11	81.13	81.75
Wav2vec2.0	78.55	79.87	78.55	78.11

In Table 2 are the different evaluation metrics shown for the four main different models. The test dataset was always the same. The models were tested with segments of the length of ten seconds and no overlap was applied, except in the case of EfficientNetV2, which was trained on Mel spectrograms obtained from 5 second audio files. All three transformer models were trained with a segment length of five seconds.
The pre-trained bird model outperforms both other models. Additionally, the pre-trained model with silence performs better than wav2vec2.0. However, with an additional silence class, the results of the pre-trained model are worse than without the silence class.

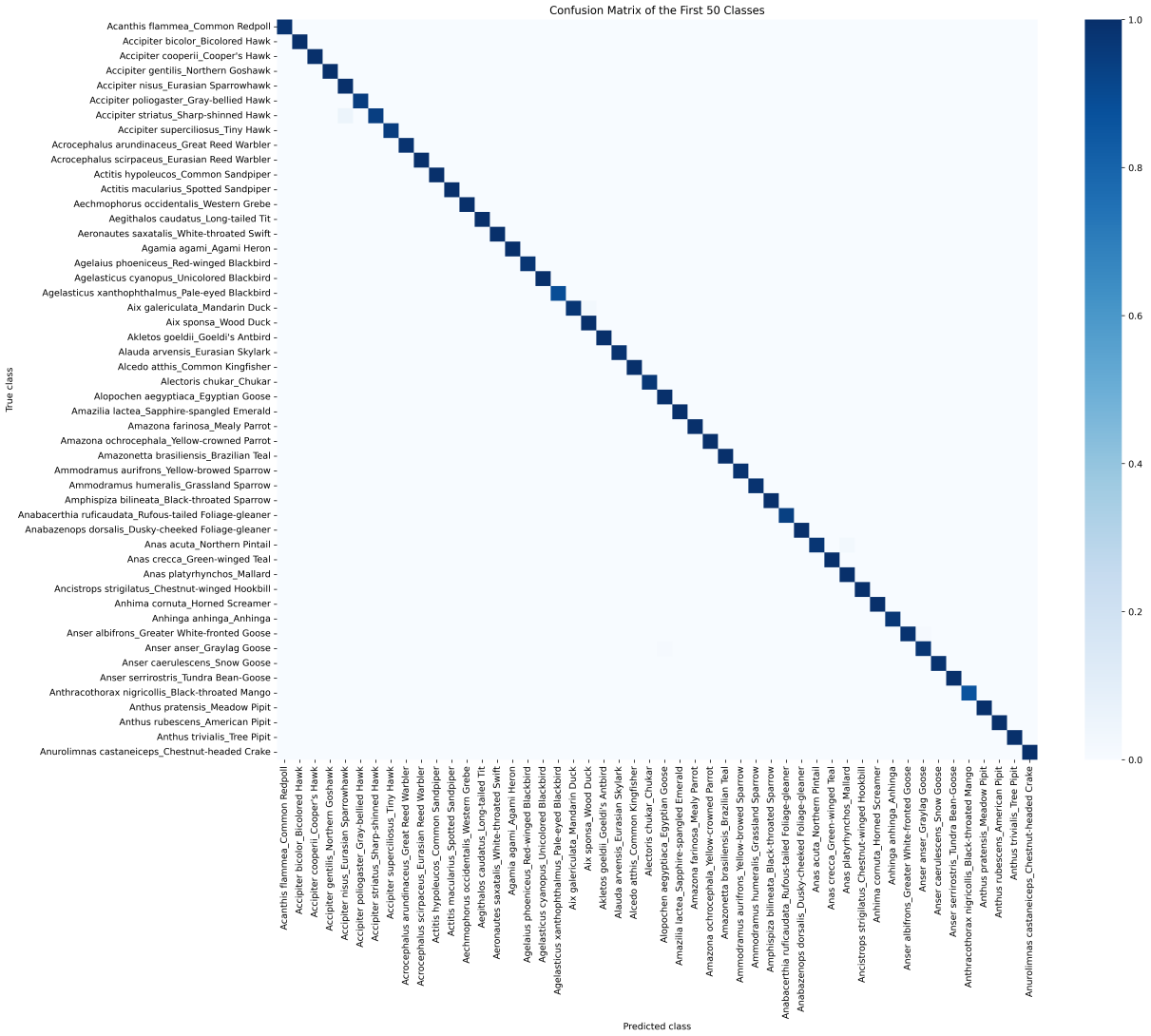


Figure 16: The confusion matrix of the EfficientNetV2M for the first 50 classes

As shown in Figure 17 the different models also perform different per bird specie. As mentioned in subsubsection 7.4.3 some birds are better classified from the model with silence. The wav2vec predicts the Greater Scaup, the Common Raven, the Eurasian Colared-Dove and the White Wagtail better. The White Wagtail, the Eurasian Colared-Dove and the Common Raven are better predicted with the wav2vec and the model with silence. Both models do not perform that better in comparison to the pre-trained model. The overlap of the classes, that are better predicted, is surprising. The precision of the pre-trained model for this species is around or below 60%. As shown in Figure 17 these species are the species with the worst precision in the pre-trained model. No explanation could be found for this behaviour.

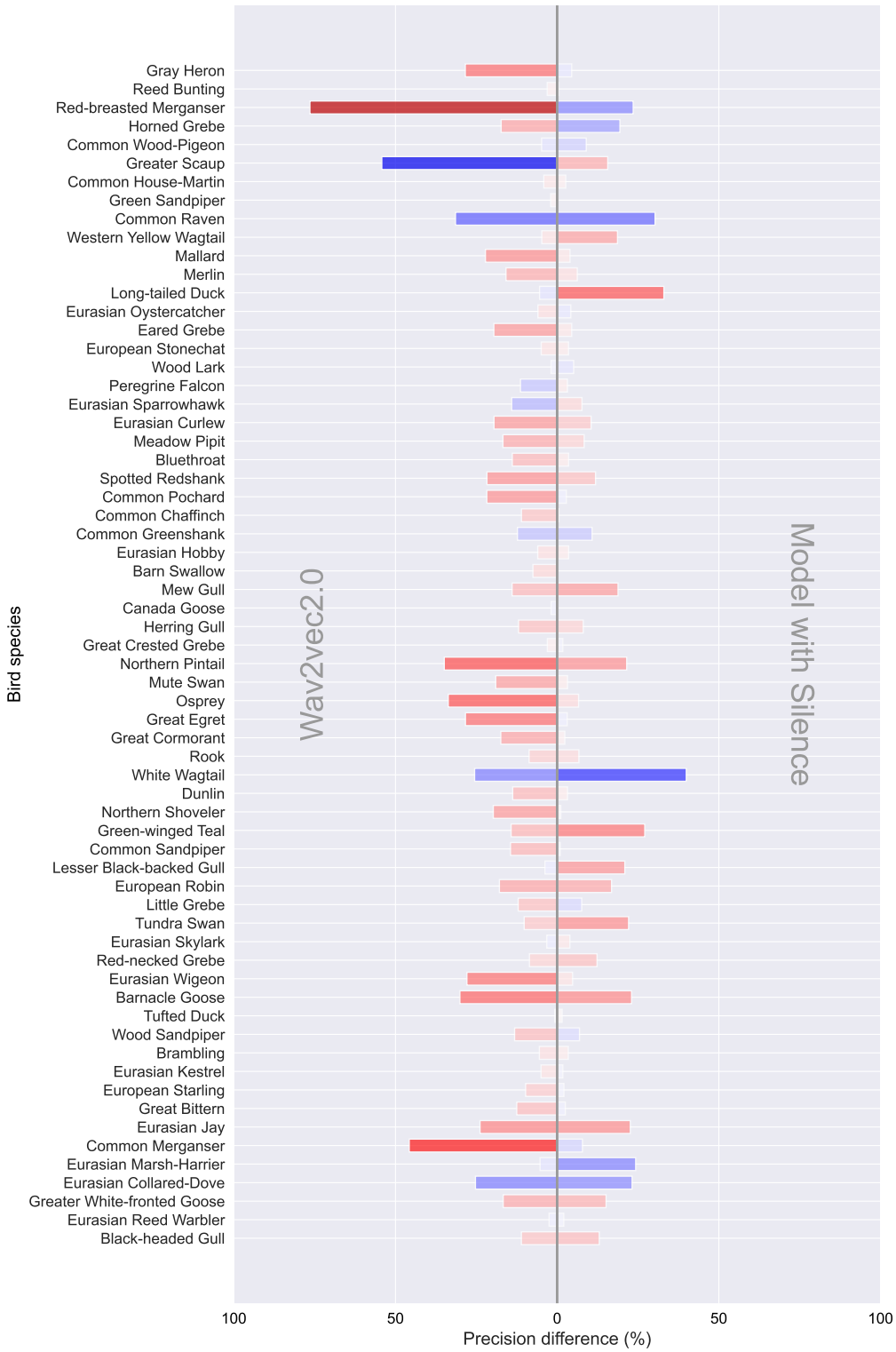


Figure 17: The following chart presents a comparison of the precision values of various models. The differences between the *pre-trained bird model* and *Wav2vec2.0* are displayed on the left side of the Y-axis, while the differences between the *pre-trained bird model* and the *model with silence* are displayed on the right side. A red bar indicates that the pre-trained bird model exhibited superior precision (in absolute terms) compared to the model being evaluated. Conversely, a blue bar signifies that the evaluated model demonstrated superior precision compared to the *pre-trained bird model.* The segment length was 10 seconds for all models.

7.5 Conclusion

In this project it could be shown, that transformer-based models are worth to know as a solution for voice recognition tasks for birds. The achieved results were accomplished without a lot of fine-tuning and improvements. The pre-trained model performs well for bird voice recognition, especially if the confidence values are taken into consideration. To achieve the same results with wav2vec2.0 based on the **facebook/wav2vec-base-960h** model, further fine-tuning and research needs to be done. Despite this, for this task CNNs and in particular, the EfficientNetV2M model demonstrates the best overall performance. Its architecture and training on Mel spectrograms allow it to achieve superior accuracy and precision, particularly in distinguishing bird species with subtle acoustic differences. This makes EfficientNetV2M a robust and reliable choice for this classification task.

7.5.1 Questions for further research

In this project, some open questions remain unanswered. Since the precision of the models differs from the segment length, a combined evaluation of the models could also be interesting. Besides that, a combination of the pre-trained model and the model with silence would also be interesting. More advanced pre-processing could also be taken into consideration. Filtering of noise, background sounds like church bells, barking dogs or rain could improve the quality of the training data. On the other hand, should the model also be robust enough to handle this type of disturbance for practical relevance. A more complex architecture with noise and silence removal in advance would be still worth to take into consideration. This could also be realised with a neural network.

Meanwhile, the training with different segment length was not fully researched in this project. The optimal segment length was not found yet. It is supposed to be between five and fifteen seconds. The testing with different segment lengths result in different evaluation metrics. Because of that, a closer evaluation and comparison of the training with different segment length would also be enlightening. Additionally, the evaluation metrics differ in dependence of the testing segment length per bird species. Therefore, the research of an intelligent combination of different segment length would be interesting.

Furthermore, the limitations under which this project was conducted could be eliminated in further research. The chosen data subset should be enlarged with the whole dataset. With the addition of other bird species, other side effects could also appear. As stated in “Overview of BirdCLEF 2020: Bird Sound Recognition in Complex Acoustic Environments” the soundscape of the recordings differ per country and recording location [10]. The density of the bird voices can be challenging.

References

[1] S. Kahl, C. M. Wood, M. Eibl, and H. Klinck, "Birdnet: A deep learning solution for avian diversity monitoring," *Ecological Informatics*, vol. 61, p. 101 236, 2021, ISSN: 1574-9541. DOI: https://doi.org/10.1016/j.ecoinf.2021.101236. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1574954121000273.

[2] Ç. H. Sekercioglu, D. G. Wenny, and C. J. Whelan, *Why Birds Matter: Avian Ecological Function and Ecosystem Services*. University of Chicago Press, Aug. 2016, ISBN: 9780226382463. DOI: 10.7208/chicago/9780226382777.001.0001. [Online]. Available: https://doi.org/10.7208/chicago/9780226382777.001.0001.

[3] L. Rauch, R. Schwinger, M. Wirth, B. Sick, S. Tomforde, and C. Scholz, *Active bird2vec: Towards end-to-end bird sound monitoring with transformers*, 2023. arXiv: 2308.07121 [cs.SD]. [Online]. Available: https://arxiv.org/abs/2308.07121.

[4] D. Stowell, "Computational bioacoustics with deep learning: A review and roadmap," *PeerJ*, vol. 10, e13152, Mar. 2022, ISSN: 2167-8359. DOI: 10.7717/peerj.13152. [Online]. Available: http://dx.doi.org/10.7717/peerj.13152.

[5] J. H. Rasmussen and A. Širović, "Automatic detection and classification of baleen whale social calls using convolutional neural networksa)," *The Journal of the Acoustical Society of America*, vol. 149, no. 5, pp. 3635–3644, May 2021, ISSN: 0001-4966. DOI: 10.1121/10.0005047. eprint: https://pubs.aip.org/asa/jasa/article-pdf/149/5/3635/16700120/3635_1_online.pdf. [Online]. Available: https://doi.org/10.1121/10.0005047.

[6] A. Baevski, H. Zhou, A. Mohamed, and M. Auli, *Wav2vec 2.0: A framework for self-supervised learning of speech representations*, 2020. arXiv: 2006.11477 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2006.11477.

[7] A. Radford, J. W. Kim, T. Xu, G. Brockman, C. McLeavey, and I. Sutskever, *Robust speech recognition via large-scale weak supervision*, 2022. arXiv: 2212.04356 [eess.AS]. [Online]. Available: https://arxiv.org/abs/2212.04356.

[8] M. Tan and Q. Le, "Efficientnetv2: Smaller models and faster training," in *Proceedings of the 38th International Conference on Machine Learning*, M. Meila and T. Zhang, Eds., ser. Proceedings of Machine Learning Research, vol. 139, PMLR, Jul. 2021, pp. 10 096–10 106. [Online]. Available: https://proceedings.mlr.press/v139/tan21a.html.

[9] M. Boeckle and T. Bugnyar, "Long-term memory for affiliates in ravens," *Current Biology*, vol. 22, no. 9, pp. 801–806, 2012, ISSN: 0960-9822. DOI: https://doi.org/10.1016/j.cub.2012.03.023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0960982212003107.

[10] S. Kahl, M. Clapp, W. Hopping, *et al.*, "Overview of birdclef 2020: Bird sound recognition in complex acoustic environments," in *Overview of BirdCLEF 2020: Bird Sound Recognition in Complex Acoustic Environments*, Sep. 2020.

List of abbreviations

CNN Convolutional neural network. 24, 25

UiA University of Agder. 22