University of Novi Sad

Faculty of Sciences

Department of Mathematics and Informatics

Student: Vasilije Simonović

Advisor: dr Miloš Savić

Committee members: dr Davor Kumozec

dr Vladimir Kurbalija

Classification of Racing Pigeons Based on Eye Images Using Deep Learning Methods

Master Thesis

Novi Sad, 2026

Acknowledgements

I would like to express my sincere gratitude to my advisor, dr Miloš Savić, for his guidance, patience, and valuable suggestions throughout the process of researching and writing this thesis. I would also like to thank the members of the examination committee for their time and constructive feedback. I would also like to thank John Smits and the other members of the EPW auction house for their generously provided photographs and their continued support of this work. Finally, I would like to thank my family and friends for their continued support during my studies.

Abstract

Assessment of a racing pigeon's potential is traditionally based on visual inspection by breeders, with particular attention often given to the eye and the so-called "eye sign" theory. Although this theory is widely used among breeders, it has not been scientifically validated. Photographs of pigeon eyes are commonly shown together with pedigree information and auction listings, especially on platforms such as PIPA (Pigeon Paradise). The value of top racing pigeons can be very high. For example, Armando was sold for approximately EUR 1.25 million, while New Kim was sold for approximately EUR 1.6 million through PIPA. Because eye photographs are already widely available in this context, they provide a suitable source of visual data for automated analysis. This thesis investigates whether deep learning models can classify racing pigeons into "good" and "bad" quality categories using only an image of the eye.

A dataset of 826 pigeon eye images was used in this thesis, including 348 images labeled as "good" and 478 images labeled as "bad". Each image in the dataset was evaluated individually, taking into account the racing results achieved by the pigeon as well as the information available in its pedigree. Based on this evaluation, each pigeon was placed into one of the two quality categories. Nine different model configurations were then trained and evaluated. These models belonged to three main groups: convolutional neural networks trained from scratch, transfer learning models based on ResNet50 and EfficientNetB0, and transformer-based models including Vision Transformer and a DINOv2-style model. The models were evaluated using accuracy, precision, recall, and F1 score. Confusion matrices were also analyzed in order to better understand how each model classified the two categories and whether it showed a tendency to favor one class over the other.

The best results were achieved by the Deep CNN and Simple CNN models. Deep CNN reached an accuracy of 0.747 and an F1 score of 0.744, while Simple CNN achieved an accuracy of 0.735 and an F1 score of 0.733. Both models were trained directly on the pigeon eye dataset and showed relatively balanced performance for both classes. The transformer-based models did not perform well. Both the Vision Transformer and the DINOv2-style model predicted the majority class for all validation images. Because of this, their accuracy appeared acceptable at first, but the models did not actually learn to distinguish between "good" and "bad" pigeons. The results show that, for a relatively small and specific dataset such as the one used in this thesis, simpler convolutional neural networks can perform better than more complex models. They also show that accuracy should not be considered alone, especially when the number of samples in the two classes is not equal.

Contents

List of Abbreviations

ANN	Artificial Neural Network
CNN	Convolutional Neural Network
ViT	Vision Transformer
ResNet	Residual Network
DINOv2	self-DIstillation with NO labels, version 2
ReLU	Rectified Linear Unit
MLP	Multi-Layer Perceptron
BN	Batch Normalization
Adam	Adaptive Moment Estimation (optimizer)
TP / FP / TN / FN	True Positive / False Positive / True Negative / False Negative
F1	F1 score, the harmonic mean of precision and recall
PIPA	Pigeon Paradise, an online racing pigeon auction platform
EPW	European Pigeon Website, an online racing pigeon auction platform
RGB	Red-Green-Blue (color image format)

1 Introduction

1.1 Motivation and Background

Racing pigeon sport has a long tradition in many parts of the world, especially in Europe. Belgium and the Netherlands are among the countries where the sport is particularly popular and where many well-known racing pigeon breeders are located. Racing pigeons are selectively bred and trained to compete over distances ranging from approximately one hundred to more than one thousand kilometers. Their performance is usually determined by the time needed to complete the race and the average flight speed achieved.

In addition to racing performance, successful pigeons can have significant breeding value. Pigeons with strong racing results or pedigrees containing successful ancestors are often sold through specialized auctions. In some cases, their prices can reach very high values. For young pigeons that have not yet competed, their future racing and breeding potential cannot be directly measured. Because of this, breeders often use pedigree information, physical characteristics, family results, and their own experience when evaluating a pigeon.

The financial importance of racing pigeons can be seen through several record-breaking sales organized through PIPA (Pigeon Paradise), one of the best-known online platforms for racing pigeons. In 2019, the Belgian racing pigeon Armando was sold for approximately EUR 1.25 million to a buyer from China. At that time, this was the highest price ever paid for a racing pigeon. Only one year later, in 2020, the hen New Kim was sold through the same platform for approximately EUR 1.6 million, also to a Chinese buyer. These examples show that the value of top racing pigeons can be very high and that estimating the quality of a pigeon can have an important financial role for breeders and buyers.

One of the characteristics traditionally observed by some breeders is the pigeon's eye. This practice is commonly known as the "eye sign" theory. According to this theory, certain characteristics of the iris, such as its color, structure, circles, and pigmentation, may be related to the racing or breeding quality of a pigeon. However, this theory has not been scientifically confirmed and opinions about its importance differ among breeders.

Close-up photographs of pigeon eyes are commonly used in racing pigeon pedigrees, advertisements, and auctions. Together with photographs of the pigeon itself, they represent a standard part of the presentation of many racing pigeons. Because a large number of these images are available, pigeon eye photographs can also be used as a source of data for computer vision and deep learning methods. The purpose of this thesis is not to prove or disprove the eye sign theory. Instead, the goal is to investigate

whether deep learning models can learn visual differences between the eyes of pigeons classified as "good" and "bad" based on their racing results and pedigree information.

Evaluation of pigeon eyes is usually subjective and strongly depends on the experience of the person performing the evaluation. Different breeders may have different opinions about the same pigeon or the same eye. For this reason, such evaluation is difficult to standardize. An automated system based on image analysis could provide an additional and more consistent source of information. Such a system would not replace the experience of breeders, but could be used as an additional tool when evaluating pigeons.

An additional motivation for this thesis is to show that high-quality racing pigeons are not found only in countries such as Belgium and the Netherlands. Competitive pigeons are also bred in Serbia, often with significantly lower market values than pigeons from the most internationally recognized lofts. One example is "Olympic Petronella" (SRB 20-108458), bred by the author. She won a gold medal in category H at the Pigeon Olympiad held in Oradea, Romania, in 2022. She also won a gold medal in category B at the East and Central Europe exhibition held in Braşov in 2023, together with numerous national and regional results.

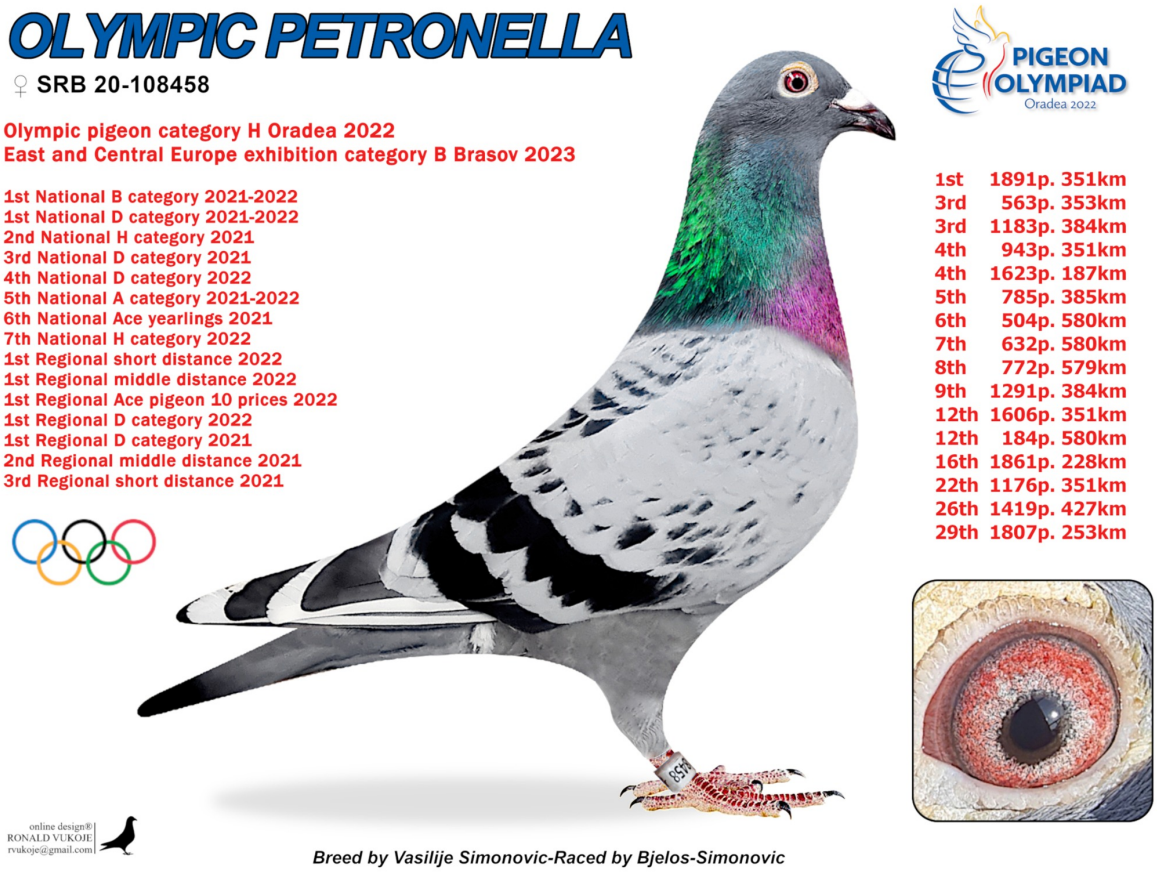

Figure 1: Official photo and racing results for "Olympic Petronella" (SRB 20-108458), bred by the author and raced under the Bjelos-Simonović partnership, including the eye close-up photograph (bottom right) of the type analyzed in this thesis.

Figure 1 presents Olympic Petronella together with her competition results certificate and a close-up photograph of her eye. This type of eye photograph is similar to the images used in the dataset analyzed in this thesis. Practical experience in breeding and racing pigeons, together with experience in evaluating pedigrees and racing results, was one of the main reasons for writing this work.

1.2 Deep Learning for Automated Visual Assessment

Recent developments in deep learning have made it possible to automate many image-based tasks that previously depended mainly on human experts. An important step in this development was AlexNet [3], developed in 2012 by Alex Krizhevsky in collaboration with Ilya Sutskever and his Ph.D. advisor Geoffrey Hinton at the University of Toronto which achieved strong results at the ImageNet Large Scale Visual Recognition Challenge in 2012. AlexNet showed that deep convolutional neural networks could learn useful image features directly from data and outperform many traditional approaches based on manually designed features.

This was followed by the development of other convolutional architectures. VGGNet [4] showed that deeper networks built from multiple small 3×3 convolutional filters could achieve very good results in image classification. ResNet [5] introduced skip connections, which made it possible to train much deeper neural networks more effectively. These architectures became an important basis for many later computer vision applications.

A relevant example for this thesis can be found in ophthalmology. Convolutional neural networks have been successfully applied to retinal images for detecting diabetic retinopathy [14] and glaucomatous optic neuropathy [15]. These studies show that neural networks can learn visual patterns from eye images that may be difficult to describe manually, even for experienced specialists.

This is also relevant for the classification of racing pigeon eyes. Experienced breeders often claim that they can recognize certain patterns in the eye that may be related to the quality of a pigeon. These patterns are not easy to define precisely or measure manually. For this reason, deep learning provides an interesting approach for testing whether such visual differences can be learned directly from pigeon eye images.

In addition to convolutional neural networks, transformer-based architectures have also become important in computer vision. The original Transformer architecture was first introduced for sequence modeling [11]. This later led to the development of the Vision Transformer (ViT) [12], which applies the self-attention mechanism to images by dividing an image into smaller patches and processing them as a sequence.

More recently, self-supervised models such as DINOv2 [13] have shown that useful visual representations can be learned from large collections of unlabeled images and then

adapted to specific tasks with a smaller amount of labeled data. In this thesis, three main approaches are compared: convolutional neural networks trained from scratch, transfer learning models using ImageNet-pretrained backbones, and transformer-based models. These approaches are described in more detail in Chapter 2.

Computer vision is also increasingly used in animal-related applications. Examples include body condition assessment in livestock, automatic breed recognition, and health monitoring in poultry using camera images. These applications show that image analysis can support or partially automate tasks that were traditionally based on manual observation.

The classification of racing pigeons based on eye images follows the same general idea. However, compared with areas such as cattle or poultry farming, racing pigeon sport has received much less attention from the computer vision research community. This makes the problem investigated in this thesis a relatively unexplored application of deep learning.

1.3 Objectives and Contributions of this Thesis

This thesis investigates whether different deep learning approaches can distinguish between high-quality ("good") and lower-quality ("bad") racing pigeons using only photographs of their eyes. Three main groups of models are considered: custom convolutional neural networks trained from scratch, transfer learning models using ImageNet-pretrained networks, and transformer-based models.

The study is based on a dataset of pigeon eye images divided into two classes. The labels were assigned by the author based on practical experience in racing pigeon sport. For every pigeon in the dataset, the available racing results and pedigree information were considered before assigning the image to the "good" or "bad" category.

Nine different model configurations were implemented and compared: a baseline CNN, a CNN with data augmentation, a CNN with batch normalization, a deeper CNN, a lightweight Simple CNN, ResNet50, EfficientNetB0, a Vision Transformer, and a DINOv2-style model. The purpose of this comparison was to determine which type of architecture is most suitable for this specific classification problem and for a relatively small dataset.

Another objective was to compare not only the final accuracy of the models, but also their behavior during classification. For this reason, precision, recall, F1 score, and confusion matrices were also analyzed. This made it possible to determine whether a model was able to classify both categories successfully or whether it mainly predicted one class.

The main contribution of this thesis is the application of several deep learning approaches to racing pigeon classification based on eye images. To the best of the author's

knowledge, this type of systematic comparison has not previously been performed for this specific problem. The results also provide a comparison between different CNN architectures, transfer learning, data augmentation, batch normalization, and transformer-based approaches under the same dataset conditions.

In addition, the confusion matrices were reconstructed from the available evaluation metrics and class counts, as described in Section 3.3. This provided a more detailed view of model performance than accuracy alone. The experiments also showed an important limitation of the transformer-based models trained from scratch, since both models converged to predicting only the majority class. This result demonstrates the importance of selecting an architecture that is appropriate for the size and characteristics of the available dataset.

1.4 Thesis Structure

This thesis is organized into five chapters. Chapter 2 describes the pigeon eye dataset used for training and evaluation, the preprocessing steps, and the theoretical background relevant to the implemented models. It includes artificial neural networks, convolutional neural networks, transfer learning, Vision Transformer, and DINOv2-style models. The nine implemented architectures and the evaluation metrics are also presented in this chapter.

Chapter 3 describes the experimental setup and training configuration of the models. It presents the quantitative results, confusion matrix analysis, the effect of data augmentation, and the performance of the transformer-based models.

Chapter 4 discusses the obtained results in more detail. The relationship between model complexity, dataset size, and classification performance is analyzed, together with the main limitations and threats to validity.

Chapter 5 concludes the thesis by summarizing the main findings and contributions and presenting possible directions for future work.

2 Materials and Methods

2.1 Data

2.1.1 Data Source and Description

The dataset used in this thesis consists of close-up photographs of racing pigeon eyes divided into two classes: "good" and "bad". The photographs were obtained from Yella, a professional photographer associated with the European Pigeon Website (EPW), who photographed the pigeons included in the dataset.

The classification labels were assigned by the author based on practical experience in racing pigeon sport. Each pigeon was evaluated individually using its available racing results and pedigree information. Based on this information, the corresponding eye image was classified as either "good" or "bad".

The complete dataset contains 826 images. It was divided into a training set of 660 images and a validation set of 166 images. The training set contains 275 "good" and 385 "bad" images, while the validation set contains 73 "good" and 93 "bad" images.

The validation set was used for model evaluation and comparison. The limitations of this approach are discussed in Section 4.4.

2.1.2 Class Distribution and Implications for Evaluation

The dataset is not completely balanced between the two classes. The "bad" class contains 478 images, which represents approximately 57.9% of the complete dataset, while the "good" class contains 348 images, or approximately 42.1%. A similar distribution is present in the validation set, where 93 of 166 images, or 56.0%, belong to the "bad" class.

This class imbalance is important when interpreting accuracy. If a model simply predicted every validation image as "bad", it would achieve an accuracy of $93/166 = 0.5602$ without actually learning how to distinguish between the two classes. As shown in Chapter 3, two of the evaluated models produced exactly this type of result.

For this reason, accuracy was not used as the only measure of model performance. Precision, recall, F1 score, and the confusion matrices described in Section 3.3 were also analyzed. These metrics provide a better understanding of whether a model is able to classify both "good" and "bad" pigeons correctly.

2.2 Data Preprocessing

The pigeon eye photographs used in this thesis are standard RGB images, so no specialized preprocessing steps were required. The preprocessing pipeline was implemented

using the TensorFlow/Keras `image_dataset_from_directory` utility, which automatically assigns class labels based on the folder structure in which the images are stored.

The images were loaded from the corresponding class folders and resized before being used as input for the models. Two image resolutions were used. All custom CNN models and both transformer-based models used images resized to 128 × 128 pixels. ResNet50 and EfficientNetB0 used 224 × 224 pixel images, which corresponds to the input size expected by their ImageNet-pretrained backbones.

For the custom architectures, pixel values were rescaled from the original range of [0, 255] to [0, 1]. This was done using a `Rescaling` layer included directly in the model. For ResNet50 and EfficientNetB0, the corresponding Keras `preprocess_input` function was used instead. This preprocessing is consistent with the normalization applied during the original ImageNet training of these models.

Data augmentation was applied only to the "CNN + Augmentation" model. This model used the same basic architecture as the Simple CNN, but additional transformations were applied to the training images. These transformations included random horizontal flipping, random rotation with a factor of 0.12, random zoom of up to 12%, and random contrast adjustment of up to 10%. The other models were trained without augmentation, which made it possible to directly compare the Simple CNN and CNN + Augmentation configurations.

The complete eye-region image was provided as input to the models, allowing each network to learn which parts of the image were most relevant for classification.

2.3 Theoretical Background

2.3.1 Artificial Neural Networks

Artificial neural networks are computational models composed of connected layers of artificial neurons. Each neuron receives one or more input values, applies a set of weights and a bias, and then passes the result through an activation function.

For an input vector x, weight vector w, and bias b, the value before applying the activation function can be written as:

$$z = w^T x + b$$

The output of the neuron is then:

$$a = f(z)$$

where f represents the activation function. Common activation functions include sigmoid, hyperbolic tangent, and Rectified Linear Unit (ReLU). The sigmoid function is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

while ReLU is defined as:

$$\text{ReLU}(z) = \max(0, z)$$

ReLU was used in the hidden layers of the convolutional neural networks implemented in this thesis because it is simple to compute and works well during neural network training.

For the final classification layer, the softmax function was used. Softmax converts the output values of the network into probabilities for each class:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

The resulting probabilities indicate how confident the model is that an input image belongs to each of the two classes.

During training, the parameters of the neural network, including its weights and biases, are adjusted by minimizing a loss function. In this thesis, the models use a two-unit softmax output for the two-class classification problem. Sparse categorical cross-entropy was therefore used as the loss function:

$$L(\theta) = -\frac{1}{N} \sum_{n=1}^{N} \log \left(p_{n,y^{(n)}} \right)$$

where N represents the number of training samples, $y^{(n)}$ is the true class of sample n, and $p_{n,y^{(n)}}$ is the probability predicted by the model for the correct class.

The network parameters are updated during training using gradient-based optimization. A general update step can be written as:

$$\theta \leftarrow \theta - \eta \nabla L(\theta)$$

where θ represents the model parameters, η is the learning rate, and $\nabla L(\theta)$ is the gradient of the loss function. The gradients are calculated using backpropagation, which propagates the classification error from the output layer back through the network.

All models in this thesis were trained using the Adam optimizer [8]. Adam adjusts the learning rate for individual parameters by keeping running estimates of the first and second moments of the gradients. These estimates are calculated as:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t,$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2.$$

The model parameters are then updated using the bias-corrected estimates:

$$\theta_t = \theta_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where g_t is the gradient at training step t, β_1 and β_2 control the moving averages, and ϵ is a small constant used to avoid division by zero.

For the models implemented in this thesis, Adam was generally used with the standard Keras settings: learning rate $\eta = 0.001$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$, except where a different learning rate is explicitly specified for a particular experiment.

2.3.2 Regularization Techniques

Because the training set used in this thesis contains only 660 images, there is a risk that the models may overfit the training data. Regularization techniques were therefore used to reduce this risk and improve the ability of the models to generalize to new images.

One of the regularization techniques used in this thesis is Dropout [7]. During training, dropout randomly sets a certain percentage of neuron activations to zero. This prevents the network from depending too strongly on individual neurons and encourages it to learn more general features.

The dropout rate is represented by pp. In the models implemented in this thesis, dropout rates between 0.3 and 0.4 were used in the fully connected layers of the custom CNN models and in the classification heads of the transfer learning models. Dropout is active only during training and is disabled when the model is used for prediction.

Another technique used in this thesis is Batch Normalization [6]. Batch normalization standardizes the activations within each mini-batch by using the mean and variance calculated from that batch. The operation can be written as:

$$\mathrm{BN}(x) = \gamma \frac{x - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} + \beta$$

where μ_B and σ_B^2 represent the mean and variance of the current mini-batch, while γ and β are parameters learned during training. The value ϵ is a small constant used to prevent division by zero.

Batch normalization can make the training process more stable and can also help the network converge faster. In this thesis, it was used only in the Batch Norm CNN model described in Section 2.4. A Batch Normalization layer was added after each convolutional layer in this architecture. The other custom CNN architectures were trained without batch normalization.

2.3.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) are neural networks designed specifically for processing images and other grid-structured data. Instead of connecting every input value directly to every neuron, CNNs use small filters that move across the image and learn local visual patterns [2].

For an input feature map XX and a convolutional filter KK, the convolution operation at position (i,j)(i,j) can be written as:

$$Y_{i,j} = \sum_{m,n,c} X_{i+m,j+n,c}\, K_{m,n,c} + b$$

where b represents the bias term. After the convolution operation, an activation function such as ReLU is usually applied.

Pooling layers are commonly used after convolutional layers to reduce the spatial size of the feature maps. In this thesis, max pooling is used. Max pooling keeps the largest value within a small local region and can be written as:

$$Y_{i,j} = \max_{(m,n)\in R_{i,j}} X_{m,n}$$

Reducing the size of the feature maps lowers the computational cost of the network and allows deeper layers to focus on the most important detected features.

By combining several convolutional and pooling layers, a CNN can learn increasingly complex image features. Early layers usually detect simple patterns such as edges, colors, and gradients, while deeper layers can learn more specific shapes and textures. Architectures such as AlexNet [3] and VGGNet [4] demonstrated the effectiveness of this approach for image classification.

For the pigeon eye classification task, this is particularly useful because differences between the two classes may be related to visual characteristics such as iris color, pigmentation, texture, or circular patterns. Instead of defining these characteristics manually, the CNN learns relevant features directly from the training images.

Five of the nine models evaluated in this thesis are custom CNN architectures trained from scratch on the pigeon eye dataset. They differ in the number of convolutional layers, number of filters, use of batch normalization, and use of data augmentation. The individual architectures are described in more detail in Section 2.4.

2.3.4 Transfer Learning and Residual Networks

Training a deep convolutional neural network from scratch usually requires a large amount of labeled data. In this thesis, only 660 images were available for training. For this reason, transfer learning was also evaluated as an alternative approach.

Transfer learning uses a model that has already been trained on a large dataset and

applies the learned features to a new task. In this thesis, models pretrained on ImageNet [9] were used. ImageNet contains more than one million labeled images from approximately one thousand object categories. Features learned from such a large dataset, including edges, colors, shapes, and textures, can also be useful for other image classification problems.

Two pretrained architectures were evaluated: ResNet50 and EfficientNetB0.

ResNet50 [5] is based on residual connections, also called skip connections. Instead of learning only a direct transformation of the input, a residual block combines the learned transformation with the original input:

$$y = F(x, W_i) + x$$

where x is the input and $F(x, W_i)$ represents the transformation learned by the residual block. These skip connections make it easier to train deeper neural networks because they improve the flow of gradients through the network. ResNet50 contains 50 layers organized into residual blocks.

EfficientNetB0 [10] uses a different approach. It was designed to achieve good classification performance while keeping the number of model parameters relatively low. EfficientNet scales the depth, width, and input resolution of the network together instead of increasing only one of these components.

For both ResNet50 and EfficientNetB0, the ImageNet-pretrained backbone was kept frozen during training. This means that the pretrained convolutional layers were not updated using the pigeon eye dataset. Only the classification part added on top of the pretrained network was trained.

The classification head consisted of global average pooling, a fully connected layer, dropout, and a final softmax layer with two outputs corresponding to the "good" and "bad" classes.

This approach reduces the number of trainable parameters and therefore reduces the risk of overfitting on a small dataset. However, the pretrained features were originally learned from general natural images rather than pigeon eyes. Because of this, they may not be fully adapted to fine details such as iris texture and pigmentation, which may be important for the classification task investigated in this thesis.

2.3.5 Vision Transformer

The Transformer architecture [11] was originally developed for sequence processing in natural language processing. Unlike convolutional neural networks, Transformers use a self-attention mechanism to model relationships between different parts of the input.

The Vision Transformer (ViT) [12] adapts this idea for image classification. Instead of processing the complete image as a single input, the image is divided into smaller

non-overlapping patches. Each patch is converted into a fixed-size vector called an embedding. Because the Transformer does not automatically know the position of each patch in the image, positional embeddings are added to preserve spatial information.

The sequence of patch embeddings is then processed through several Transformer encoder blocks. Each block contains a multi-head self-attention mechanism and a feed-forward neural network. Residual connections and layer normalization are also used to improve the training process.

The self-attention operation can be written as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where Q, K, and V represent the query, key, and value matrices, while d_k represents the dimensionality of the key vectors.

The attention mechanism allows each image patch to consider information from all other patches. Multiple attention heads are used in parallel, which allows the model to learn different relationships between parts of the image.

A major difference between CNNs and Vision Transformers is the way spatial information is learned. CNNs naturally focus on local image patterns through convolutional filters. Vision Transformers do not have the same built-in local structure and therefore depend more strongly on the available training data.

This is particularly important for this thesis because the available training set contains only 660 images. Vision Transformers are usually more effective when trained on much larger datasets or when pretrained models are used [12]. The effect of training a Vision Transformer from scratch on this relatively small dataset is presented in Chapter 3.

2.3.6 Self-Supervised Learning and DINOv2

Self-supervised learning is an approach in which a model learns useful visual features from images without requiring manually assigned class labels. One example is DINOv2 [13], which was developed to learn general-purpose image representations from very large collections of unlabeled images.

DINOv2 uses a teacher-student training approach. The student network is trained to produce representations similar to those generated by the teacher network for different versions of the same image. The teacher network is updated gradually from the student network during training. In this way, the model can learn useful visual features without using manually defined labels.

Different crops and transformations of the same image can also be used during training. The objective is to make the model recognize that different views of the same image still represent the same underlying visual content. This approach builds on the original

DINO method [20] and was later developed further in DINOv2.

Pretrained DINOv2 representations can then be used for different computer vision tasks, including image classification, segmentation, and image retrieval. This can be especially useful when the target dataset contains only a limited number of labeled examples.

However, an important distinction must be made for the model used in this thesis. The "DINOv2-style" model implemented here is not the original pretrained DINOv2 model. It is a custom self-attention model with a structure similar to the Vision Transformer described in Section 2.3.5, but with a different projection head.

The model was trained completely from scratch using the labeled pigeon eye dataset. It did not use pretrained DINOv2 weights, self-supervised teacher-student training, multi-crop training, or the large dataset used to train the original DINOv2 model. This distinction is important when analyzing the results presented in Chapter 3 and is discussed further in Section 4.2.

2.3.7 Comparison of Architectural Paradigms

The models evaluated in this thesis belong to three main architectural groups: convolutional neural networks trained from scratch, transfer learning models, and transformer-based models trained from scratch. Each approach has different advantages and limitations, especially when working with a relatively small dataset.

CNN models learn directly from the target dataset and have a strong ability to detect local image features such as edges, textures, and patterns. Transfer learning models use features learned previously from a much larger dataset, which can be useful when only a limited amount of training data is available. Transformer-based models are more flexible in learning relationships between different parts of an image, but usually require much more training data or pretraining.

Table A summarizes the main differences between these three approaches and also shows the general behavior observed in the experiments presented in Chapter 3.

Table A: Conceptual comparison of the three architectural approaches evaluated in this thesis.

Property	**CNN (from scratch)**	**Transfer Learning**	**Transformer (from scratch)**
Inductive bias	Strong, based on local image features and translation equivariance	Strong, inherited from the pretrained CNN backbone	Limited, spatial relationships must be learned from data
Data efficiency	Moderate	High because of pretraining	Low without pretraining
Trainable parameters	All layers	Classification head only	All layers
Adaptation to pigeon eye images	High, since the complete model is trained on the target dataset	Limited by the frozen pretrained backbone	Potentially high if sufficient training data is available
Main risk with a small dataset	Overfitting	Features may not fully match the target domain	Failure to converge or majority-class collapse
Result observed in Chapter 3	Best overall performance	Intermediate performance	Complete majority-class collapse

2.4 Implemented Architectures

The following nine model configurations were implemented and trained, each summarized with its input resolution, layer composition, and approximate output head:

CNN (baseline). 128x128 input; three convolutional blocks (16, 32, 64 filters, 3x3 kernels, ReLU) each followed by max pooling; global average pooling; Dense(64, ReLU); Dropout(0.3); Dense(2, softmax).

Simple CNN. 128x128 input; three convolutional blocks (32, 64, 128 filters) each followed by max pooling; Flatten; Dense(128, ReLU); Dropout(0.4); Dense(2, softmax).

CNN + Augmentation. Identical architecture to Simple CNN, trained with the augmentation pipeline described in Section 2.2.

Batch Norm CNN. 128x128 input; four convolutional blocks (32, 64, 64, 96 filters), each followed by batch normalization (Section 2.3.2) and max pooling; Flatten; Dense(256, ReLU); Dropout(0.4); Dense(2, softmax).

Deep CNN. 128x128 input; four convolutional blocks (16, 32, 64, 64 filters) each followed by max pooling; Flatten; Dense(256, ReLU); Dropout(0.4); Dense(2, softmax).

ResNet50. 224x224 input; frozen ImageNet-pretrained ResNet50 backbone; GlobalAveragePooling2D; Dropout(0.3); Dense(128, ReLU); Dropout(0.3); Dense(2, softmax).

EfficientNetB0. 224x224 input; frozen ImageNet-pretrained EfficientNetB0 backbone; GlobalAveragePooling2D; Dropout(0.3); Dense(128, swish); Dropout(0.3); Dense(2, softmax).

Vision Transformer (ViT). 128x128 input; 16x16 patches (64 patches total, arranged as an 8x8 grid); patch embedding dimension 64; learnable position embedding; four transformer encoder blocks (4 attention heads each); global average pooling; MLP head [128]; Dense(2, softmax); trained fully from scratch.

DINOv2-style. Same patch embedding and transformer encoder backbone as ViT; MLP projection head [256, 128] with an additional layer normalization; Dense(2, softmax); trained fully from scratch. Among the custom CNNs, the baseline CNN and Deep CNN configurations are the most directly comparable in filter progression (16-32-64 versus 16-32-64-64 filters), differing primarily in network depth and the use of a global-average-pooling head versus a Flatten-based head, a distinction that proves consequential in Chapter 3.

2.5 Evaluation Metrics

The performance of all models was evaluated on the validation set using four metrics: accuracy, precision, recall, and F1 score. These metrics were calculated from the confusion matrix, which shows how many samples from each true class were classified correctly or incorrectly.

For a two-class classification problem, the confusion matrix contains True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN). These values can be used to calculate the evaluation metrics.

Accuracy represents the proportion of correctly classified samples:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Accuracy provides a general measure of model performance, but it can be misleading when the classes are not equally distributed. As explained in Section 2.1.2, a model that predicts only the majority "bad" class would still achieve an accuracy of 0.5602 on the validation set, even though it would not be able to distinguish between the two classes.

Precision measures how many samples predicted as a particular class actually belong to that class:

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall measures how many samples belonging to a particular class were correctly identified:

$$\text{Recall} = \frac{TP}{TP + FN}$$

The F1 score combines precision and recall into a single value:

$$F1 = 2\,\frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

Precision, recall, and F1 score were calculated separately for both classes and then macro-averaged. Macro-averaging gives the same importance to the "good" and "bad" classes, regardless of how many images each class contains. This is useful for the dataset used in this thesis because the two classes are not completely balanced.

If a model does not predict any samples for a particular class, the corresponding metric is set to zero in order to avoid division-by-zero errors. The macro-averaging approach used in this thesis follows standard evaluation practices for classification problems with class imbalance [18].

All reported metrics have values between 0 and 1, where higher values indicate better performance. However, the metrics should be considered together rather than individually. In particular, accuracy alone does not provide enough information about whether the model performs well for both classes.

For this reason, confusion matrices were also analyzed in Section 3.3. The exact confusion matrix values were reconstructed using the reported accuracy, macro recall, and the known number of "good" and "bad" images in the validation set. This made it possible to analyze in more detail how each model classified the two classes.

2.6 Implementation and Software Environment

All nine models were implemented in Python using TensorFlow [19] and the Keras API. Model performance was evaluated using scikit-learn, which was used to calculate accuracy, precision, recall, and F1 score.

A shared `model_utils` module was used for data loading, image resizing, batching, prediction extraction, metric calculation, and result logging. Using the same utility functions for all models ensured that every architecture was trained and evaluated using the same validation dataset and the same evaluation procedure.

Each model had its own training script, while the final evaluation results were automatically saved to a common results file. This made it easier to compare all nine model configurations under the same conditions.

All experiments were performed locally on the author's own computer.

3 Experimental Results

3.1 Experimental Setup and Training Configuration

All nine models were trained using the same data-loading procedure described in Section 2.2. The main training settings for each model are presented in Table 1. These include the input image size, batch size, number of training epochs, and the main characteristics of each architecture.

All models were trained using the Adam optimizer [8] and sparse categorical cross-entropy loss. Adam was selected because it adapts the learning rate for individual model parameters and generally provides stable training without requiring extensive manual tuning. The standard Adam learning rate of 0.001 was used in the main experiments, unless otherwise stated. Each model used a final softmax layer with two outputs corresponding to the "good" and "bad" classes.

All main experiments were performed during a single experimental session on June 3, 2026. Two ResNet50 training runs were performed during this session. The second run is the one reported in the main comparison presented in Table 2.

Based on the timestamps automatically recorded by the training scripts, the complete set of experiments required approximately 2 hours and 21 minutes. The relatively short training time was mainly possible because of the small dataset of 826 images, the relatively low input image resolutions, and the fact that the task is image classification rather than a more computationally demanding pixel-level task such as image segmentation.

Table 1: Hyperparameter and configuration summary for all trained models.

Model	Input Size	Batch Size	Epochs	Notes
CNN	128 × 128	8	15	Global average pooling head
Simple CNN	128 × 128	32	15	No augmentation
CNN + Augmentation	128 × 128	32	15	Same architecture as Simple CNN
Batch Norm CNN	128 × 128	8	18	4 convolutional blocks + BatchNorm
Deep CNN	128 × 128	32	18	4 convolutional blocks, no BatchNorm
ResNet50	224 × 224	16	12	Frozen ImageNet backbone
EfficientNetB0	224 × 224	16	12	Frozen ImageNet backbone
Vision Transformer	128 × 128	16	12	Trained from scratch
DINOv2-style	128 × 128	16	12	Trained from scratch

3.2 Quantitative Results

Table 2 presents the final validation results for all nine model configurations. The models are ordered according to validation accuracy.

Table 2: Final validation performance of all nine model configurations, ordered by accuracy.

Model	Accuracy	Precision	Recall	F1
Deep CNN	0.7470	0.7434	0.7447	0.7440
Simple CNN	0.7349	0.7318	0.7340	0.7325
ResNet50	0.6988	0.6979	0.6840	0.6856
Batch Norm CNN	0.6747	0.7201	0.6979	0.6708
EfficientNetB0	0.6205	0.6657	0.5773	0.5355
CNN + Augmentation	0.5904	0.6083	0.5446	0.4888
Vision Transformer	0.5602	0.2801	0.5000	0.3591
DINOv2-style	0.5602	0.2801	0.5000	0.3591
CNN (baseline)	0.5301	0.3450	0.4746	0.3577

The best-performing model was Deep CNN, with an accuracy of 0.7470 and an F1 score of 0.7440. The second-best result was achieved by Simple CNN, with an accuracy of 0.7349 and an F1 score of 0.7325.

Both models are custom convolutional neural networks trained directly on the pigeon eye dataset without using a pretrained backbone. Their accuracy values are clearly above the majority-class baseline of 0.5602. In addition, their precision, recall, and F1 scores are relatively close to each other, indicating that both models performed well across the two classes rather than simply favoring the majority "bad" class.

The classification behavior of all models is analyzed in more detail using confusion matrices in Section 3.3.

3.3 Confusion Matrix Analysis

Accuracy, precision, recall, and F1 score provide useful information about model performance, but they do not show exactly how the model classifies each individual class. For this reason, confusion matrices were also analyzed.

The validation set contains 166 images: 73 from the "good" class and 93 from the "bad" class. Because these class counts are known, the confusion matrix for each model can be reconstructed from the reported accuracy and macro recall.

Let a represent the number of "good" images correctly classified as "good", and d the number of "bad" images correctly classified as "bad". These values can be calculated using the following equations:

$$a + d = \text{Accuracy} \times 166,$$
$$\frac{a}{73} + \frac{d}{93} = 2 \times \text{Recall}_{\text{macro}}.$$

After calculating a and d, the remaining values can be obtained as:

$$b = 73 - a,$$
$$c = 93 - d.$$

where b represents "good" images incorrectly classified as "bad", and c represents "bad" images incorrectly classified as "good".

The reconstructed confusion matrices were also checked using the reported macro precision values. The calculated values reproduced the reported metrics, confirming that the reconstructed confusion matrices are consistent with the evaluation results.

Table 3: Reconstructed confusion matrices and recall for each class.

Model	a	b	c	d	Recall Good	Recall Bad
Deep CNN	53	20	22	71	72.6%	76.3%
Simple CNN	53	20	24	69	72.6%	74.2%
ResNet50	41	32	18	75	56.2%	80.6%
Batch Norm CNN	65	8	46	47	89.0%	50.5%
EfficientNetB0	16	57	6	87	21.9%	93.5%
CNN + Augmentation	12	61	7	86	16.4%	92.5%
Vision Transformer	0	73	0	93	0.0%	100.0%
DINOv2-style	0	73	0	93	0.0%	100.0%
CNN (baseline)	1	72	6	87	1.4%	93.5%

The results in Table 3 show that several models have a strong tendency to predict the majority "bad" class. The most extreme examples are the Vision Transformer and DINOv2-style models. Both models classified all 166 validation images as "bad". As a result, they correctly classified all 93 "bad" images but failed to correctly classify any of the 73 "good" images.

The baseline CNN showed a similar behavior, although it correctly classified one "good" image. EfficientNetB0 and CNN + Augmentation also showed a strong preference for the "bad" class. EfficientNetB0 correctly classified only 16 of the 73 "good" images, while CNN + Augmentation correctly classified only 12.

Deep CNN and Simple CNN showed the most balanced classification performance. Deep CNN correctly classified 53 of the 73 "good" images and 71 of the 93 "bad" images. Simple CNN also correctly classified 53 "good" images and 69 "bad" images. Their recall values for both classes are therefore relatively similar, which indicates that these models learned to distinguish between the two classes rather than mainly predicting the majority class.

Batch Norm CNN showed a different behavior. It correctly classified 65 of the 73 "good" images, giving a "good" class recall of 89.0%, but correctly classified only 47 of the 93 "bad" images. In total, it predicted 111 validation images as "good". This means that, unlike most of the lower-performing models, Batch Norm CNN was biased toward the minority "good" class rather than the majority "bad" class.

One possible explanation for this behavior is the use of batch normalization together with a relatively small batch size of 8. The statistics calculated for each small mini-batch may influence the training process and the final classification behavior.

3.4 Effect of Data Augmentation

The effect of data augmentation was evaluated by comparing the Simple CNN with the CNN + Augmentation model. Both models use the same architecture. The only difference is that random horizontal flipping, rotation, zoom, and contrast adjustment were applied during the training of the CNN + Augmentation model, as described in Section 2.2.

The Simple CNN achieved an accuracy of 0.7349, while the CNN + Augmentation model achieved 0.5904. The confusion matrices in Table 3 also show a clear difference in classification behavior. Simple CNN predicted the "good" class 77 times, while CNN + Augmentation predicted the "good" class only 19 times out of 166 validation images. This shows that the augmented model developed a much stronger bias toward the majority "bad" class.

One possible explanation is that some of the applied transformations changed the visual characteristics of the eye too much. For this task, important information may be contained in fine details such as iris texture, pigmentation, and circular patterns. Rotation, zoom, and contrast changes may therefore make these features more difficult to learn.

For future research, a larger dataset with more pigeon eye images would be useful to evaluate the effect of data augmentation more reliably.

3.5 Failure Analysis of Transformer-Based Models

The Vision Transformer and DINOv2-style models achieved identical results: accuracy of 0.5602, precision of 0.2801, recall of 0.5000, and F1 score of 0.3591. As shown in Section 3.3, both models classified every validation image as belonging to the majority "bad" class.

This means that all 93 "bad" images were classified correctly, while none of the 73 "good" images were correctly identified. Although the accuracy of 0.5602 may appear reasonable at first, it is exactly the same as the majority-class baseline. Therefore, these models did not learn to distinguish between the two classes.

Both transformer-based models were trained completely from scratch using only 660 training images. Unlike CNNs, transformer architectures do not have the same built-in ability to focus on local image patterns and must learn these relationships from the training data. With the relatively small dataset used in this thesis and without pretraining, the models were not able to learn useful visual features for the classification task.

The results show that training these transformer-based architectures from scratch was not suitable for the current dataset size. Future experiments should include a larger number of pigeon eye images and could also evaluate pretrained Vision Transformer and

DINOv2 models. This would make it possible to determine whether transformer-based approaches can achieve better results when more training information is available.

3.6 Comparative Discussion

The results presented in Sections 3.2–3.5 show several clear differences between the evaluated model groups.

The custom CNN models achieved the best overall results. Deep CNN and Simple CNN showed the highest accuracy and F1 scores and also produced the most balanced confusion matrices. These results indicate that the CNN architectures were able to learn useful visual features directly from the pigeon eye images.

The transfer learning models, ResNet50 and EfficientNetB0, achieved intermediate results. ResNet50 performed better than EfficientNetB0, but neither model reached the performance of Deep CNN or Simple CNN. One possible reason is that the pretrained features were originally learned from general ImageNet images and may not be fully suitable for detecting small differences in pigeon iris texture and pigmentation. In addition, the pretrained backbones were kept frozen, so the feature extraction layers were not adapted directly to the pigeon eye dataset.

The transformer-based models achieved the weakest results. Both the Vision Transformer and DINOv2-style model were trained from scratch and classified all validation images as the majority "bad" class. This shows that the available 660 training images were not sufficient for these models to learn useful features in the current experimental setup.

Overall, the results show that the custom convolutional neural networks were the most suitable models for the dataset used in this thesis. Transfer learning provided reasonable results, while the transformer-based models would require further experiments with a larger dataset or pretrained models.

3.7 Ablation Study

An ablation study was performed to examine how different training settings affect the performance of the Simple CNN architecture described in Section 2.4. Simple CNN was selected because it achieved one of the best results in the main comparison while remaining relatively simple and fast to train.

The experiments examined the effect of learning rate, batch size, number of training epochs, and early stopping. The results are presented in Table 7.

Table 7: Ablation study results using the Simple CNN architecture.

Configuration	LR	Batch	Max Epochs	Early Stop	Epochs Run	Accuracy	Recall	F1
LR = 0.0001	0.0001	32	15	No	15	0.6446	0.6077	0.5848
LR = 0.001	0.001	32	15	No	15	0.6627	0.6768	0.6622
LR = 0.01	0.01	32	15	No	15	0.5602	0.5000	0.3591
Batch = 8	0.001	8	15	No	15	0.7289	0.7360	0.7287
Batch = 32	0.001	32	15	No	15	0.7048	0.6953	0.6968
Batch = 64	0.001	64	15	No	15	0.6867	0.6527	0.6397
30 epochs, no ES	0.001	32	30	No	30	0.6747	0.6935	0.6730
30 epochs, with ES	0.001	32	30	Yes, patience = 3	4	0.5602	0.5000	0.3591

Effect of data augmentation. The effect of data augmentation was already analyzed in Section 3.4. The Simple CNN achieved an accuracy of 0.7349, while the CNN + Augmentation model achieved an accuracy of 0.5904. The augmented model also showed a stronger tendency to classify images as belonging to the majority "bad" class. For this reason, no additional augmentation experiment was included in this section.

For future research, the effect of augmentation should be tested on a larger dataset containing more pigeon eye images. It would also be useful to train the augmented model for more epochs and test different combinations and strengths of image transformations.

Effect of learning rate. The learning rate had a clear influence on model performance. With a learning rate of 0.0001, the model achieved an accuracy of 0.6446. Increasing the learning rate to 0.001 improved the accuracy to 0.6627.

A learning rate of 0.01 produced a considerably worse result. The model achieved an accuracy of 0.5602, recall of 0.5000, and F1 score of 0.3591. These values are the same as the majority-class result observed for the Vision Transformer and DINOv2-style models. In this case, the Simple CNN also classified all validation images as belonging to the "bad" class.

The results show that the learning rate has an important effect on the ability of the model to learn from the available data. A learning rate that is too high can prevent the model from reaching a useful solution, while a smaller learning rate may require a longer training period.

Effect of batch size. Batch size 8 achieved the best result in the ablation study, with an accuracy of 0.7289, recall of 0.7360, and F1 score of 0.7287. Batch size 32 achieved an accuracy of 0.7048, while batch size 64 achieved 0.6867.

With a smaller batch size, the model performs more parameter updates during each epoch. For the training dataset used in this thesis, batch size 8 results in approximately

83 updates per epoch, compared with approximately 21 updates when batch size 32 is used. These more frequent updates may have helped the model learn useful features more effectively.

The Simple CNN result reported in Table 2 achieved an accuracy of 0.7349, while the batch-size-32 experiment in the ablation study achieved 0.7048. This difference also shows that neural network training can produce somewhat different results because of factors such as random weight initialization and data shuffling.

Effect of training duration. The effect of a longer training period was evaluated by increasing the maximum number of epochs from 15 to 30. Without early stopping, validation accuracy continued to improve during later stages of training and reached approximately 0.7289 around epoch 28. At the end of epoch 30, the validation accuracy was 0.6747.

At the same time, training accuracy increased to approximately 0.997. The difference between the training and validation results indicates that the model began to overfit the training data during the later epochs.

These results show that the number of training epochs is an important parameter for this classification task. A shorter training period may stop before the model reaches its best validation result, while training for too many epochs can increase overfitting. Future experiments should therefore investigate the training duration in more detail.

Effect of early stopping. Early stopping was tested using validation accuracy with a patience value of three epochs. Training stopped after four epochs, and the resulting model achieved an accuracy of 0.5602, recall of 0.5000, and F1 score of 0.3591.

The training log shows that validation accuracy remained at 0.5602 during the first three epochs. In the experiment without early stopping, the accuracy started to increase after this initial period. The selected patience value therefore stopped the training too early for this dataset.

This result shows that early stopping must be configured carefully. For this classification problem, a larger patience value could allow the model more time to move beyond the initial majority-class prediction and begin learning useful differences between the two classes.

Combining Deep CNN with batch size 8. Deep CNN achieved the highest accuracy in the main model comparison, while batch size 8 produced the best result in the Simple CNN ablation study. For this reason, an additional experiment was performed using the Deep CNN architecture with batch size 8.

Two runs of this configuration produced considerably different results. The first reached a validation accuracy of 0.7289 at epoch 18. However, the process ended before

precision, recall, and F1 score could be calculated because the results file was locked by another program.

The second run completed the full evaluation and achieved an accuracy of 0.6205, precision of 0.6124, recall of 0.6053, and F1 score of 0.6049.

The difference between the two results shows that combining the best-performing architecture with the best batch size from another experiment does not automatically produce a better model. Neither result exceeded the original Deep CNN accuracy of 0.7470 reported in Table 2.

Effect of synthetic images. The use of synthetic pigeon eye images generated with a diffusion model was considered but was not implemented in this thesis. Producing useful synthetic images would require adapting a generative model to the specific appearance of pigeon eyes, which would require additional computational resources and a separate experimental procedure.

This approach could be investigated in future research, especially together with a larger real dataset. Synthetic images could potentially increase the diversity of the training data and provide additional examples for model training.

4 Discussion

4.1 Architectural Complexity versus Performance

The results of this thesis show that a more complex architecture does not necessarily provide better performance for a relatively small and specific image classification problem. In this study, the best results were achieved by custom convolutional neural networks trained directly on the pigeon eye dataset. These models performed better than both the transfer learning models with frozen pretrained backbones and the transformer-based models trained from scratch.

This is an important observation for future work with similar datasets. When only a limited number of labeled images is available, increasing model complexity does not automatically improve classification performance. The results suggest that simpler CNN architectures can be a suitable starting point before moving to larger or more complex models.

The difference between the baseline CNN and Deep CNN is particularly interesting. The two models use a similar sequence of convolutional filters, with 1616-3232-6464 filters in the baseline CNN and 1616-3232-6464-6464 filters in Deep CNN. However, the baseline CNN uses global average pooling, while Deep CNN uses a `Flatten` layer followed by a larger fully connected layer.

Deep CNN achieved an accuracy of 0.7470, while the baseline CNN achieved only 0.5301. One possible reason for this difference is that global average pooling reduces the spatial information before classification. For the pigeon eye images, small differences in iris texture, pigmentation, and structure may be important for distinguishing between the two classes. The `Flatten`-based architecture may preserve more of this information before the final classification layers.

The Batch Norm CNN showed a different classification pattern compared with most of the other lower-performing models. Instead of mainly predicting the majority "bad" class, it showed a strong tendency toward the "good" class. It achieved a recall of 89.0% for "good" pigeons but only 50.5% for the "bad" class.

One possible reason for this behavior is the combination of batch normalization and the relatively small batch size of 8. Batch normalization calculates statistics from each mini-batch, and with a small batch size these statistics can vary more during training. This may influence the learned decision boundary and classification behavior of the model.

Further experiments with different batch sizes and repeated training would be useful to determine whether this behavior remains consistent. A larger dataset would also make it possible to evaluate the effect of batch normalization more reliably.

4.2 Implications of the Transformer Collapse

Both transformer-based models classified all validation images as belonging to the majority "bad" class. This was the clearest failure observed among the evaluated architectures and shows that the transformer models used in this thesis were not suitable for the available dataset in their current form.

An important point is that the DINOv2-style model implemented in this thesis is not the original pretrained DINOv2 model. As explained in Section 2.3.6, it uses a similar self-attention-based structure but was trained from scratch on the pigeon eye dataset. Therefore, it does not benefit from the large-scale self-supervised pretraining used by the original DINOv2 model.

The same limitation applies to the Vision Transformer, which was also trained from scratch. With only 660 training images, neither transformer-based model was able to learn useful features for distinguishing between the two classes.

Future research should therefore test pretrained DINOv2 and Vision Transformer models. A similar approach to the one used with ResNet50 and EfficientNetB0 could be applied, where a pretrained backbone is used and a new classification head is trained on the pigeon eye dataset. A larger dataset would also provide a better basis for evaluating transformer-based architectures on this classification problem.

4.3 Limitations

The main limitation of this study is the size of the dataset. The complete dataset contains 826 pigeon eye images, of which 660 were used for training. This is a relatively small number for deep learning, particularly for more complex architectures such as Vision Transformer and DINOv2-style models.

The dataset is also moderately imbalanced. Approximately 57.9% of the images belong to the "bad" class, while 42.1% belong to the "good" class. As shown in the experimental results, this imbalance influenced some models, which developed a strong tendency to predict the majority class.

Another limitation is that no separate independent test set was used. The available data were divided into training and validation sets, and the validation set was also used for the final comparison of the models. Future work should use a larger dataset that can be divided into separate training, validation, and test sets. Cross-validation could also be used to obtain a more reliable estimate of model performance.

The labels represent another important limitation. The "good" and "bad" categories were assigned based on racing results, pedigree information, and the author's practical experience in racing pigeon sport. Although this provides a practical method for creating the dataset, pigeon quality cannot be represented perfectly by only two categories. Future datasets could include more detailed information about racing performance, pedi-

gree quality, age, distance category, and other relevant characteristics.

Finally, the current models provide only a classification result and do not show which parts of the eye influenced the prediction. Explainability methods such as Grad-CAM [16] could be used in future work to visualize the image regions that contribute most strongly to the model decision. This would make it possible to examine whether the best-performing CNN models are mainly using iris characteristics or other parts of the image.

4.4 Threats to Validity

Several factors may affect how the results of this study should be interpreted.

One limitation is related to the definition of the "good" and "bad" classes. The labels were assigned by the author based on racing results, pedigree information, and practical experience in racing pigeon sport. Although this provides a meaningful basis for classification, pigeon quality is complex and cannot be described perfectly by only two categories. Racing performance can also depend on many factors that are not visible in an eye image, including training, health condition, race distance, weather conditions, and management.

Another limitation is that all eye photographs used in the dataset were obtained from the same data source. This means that the images may have similar lighting conditions, camera settings, image quality, and photographic style. It is therefore not yet known how well the trained models would perform on eye photographs taken by different photographers, with different cameras, or under different lighting conditions.

The dataset also represents a limited number of pigeons compared with the overall diversity present in racing pigeon populations. Pigeons from different breeders, countries, bloodlines, and racing categories may have different visual characteristics. A larger and more diverse dataset would therefore be necessary before the results could be generalized more broadly.

For future research, the dataset should include images from multiple sources and different racing pigeon populations. It would also be useful to include more detailed racing and pedigree information in order to investigate whether the models perform consistently across different groups of pigeons.

4.5 Practical Deployment Considerations

This thesis focuses mainly on comparing different deep learning models rather than developing a complete application for practical use. However, the results show that the best-performing models could potentially be used as a basis for a future tool for racing pigeon breeders, buyers, or auction platforms such as PIPA.

Deep CNN and Simple CNN are relatively small models compared with ResNet50

and the transformer-based architectures. Both use 128 × 128 pixel input images and contain only several convolutional layers. Because of this, they could potentially be used on standard consumer hardware or integrated into a web or mobile application without requiring expensive computing resources.

A possible practical system could allow a user to upload a close-up photograph of a pigeon eye and receive a classification result from the trained model. This result could then be used as one additional source of information together with racing results, pedigree, physical examination, and the breeder's own experience.

Such a system should not be considered a replacement for experienced pigeon breeders. The quality of a racing pigeon depends on many characteristics, and the current models use only the eye image as input. The model also learns from the "good" and "bad" categories created using racing results and pedigree information, meaning that its prediction should be interpreted only within the context of the dataset used for training.

Before practical deployment, the dataset should be significantly expanded and should include photographs from different breeders, countries, cameras, and lighting conditions. The models should also be evaluated on a separate test dataset containing pigeons that were not involved in model development.

Explainability methods such as Grad-CAM could also be added to a future system. This would allow users to see which parts of the pigeon eye had the strongest influence on the model prediction. Together with a larger and more diverse dataset, this could provide a better understanding of what visual features the model is actually using and make the system more useful as an additional tool for racing pigeon assessment.

5 Conclusion

5.1 Summary of Contributions

This thesis investigated the use of deep learning for classifying racing pigeons based on photographs of their eyes. The idea was motivated by the traditional practice among pigeon breeders of observing eye characteristics when evaluating the potential quality of a pigeon. Although the so-called "eye sign" theory has not been scientifically confirmed, eye photographs are commonly used in pedigrees, advertisements, and auction listings. The high financial value of top racing pigeons, illustrated by sales such as Armando for approximately EUR 1.25 million and New Kim for approximately EUR 1.6 million, also shows the practical importance of pigeon quality assessment.

A dataset of 826 pigeon eye images was used in this study. Each image was assigned to either the "good" or "bad" class by the author based on the racing results of the pigeon, pedigree information, and practical experience in racing pigeon sport.

Nine different model configurations were implemented and compared. These included custom convolutional neural networks trained from scratch, transfer learning models based on ResNet50 and EfficientNetB0, and transformer-based models including Vision Transformer and a DINOv2-style architecture.

The best result was achieved by Deep CNN, with an accuracy of 0.7470 and an F1 score of 0.7440. Simple CNN achieved the second-best result, with an accuracy of 0.7349 and an F1 score of 0.7325. Both models were trained directly on the pigeon eye dataset and achieved more balanced classification between the "good" and "bad" classes than the other evaluated models.

The transfer learning models achieved intermediate results. ResNet50 performed better than EfficientNetB0, but neither model achieved the performance of the two best custom CNN architectures. Since their pretrained ImageNet backbones were kept frozen, the models were limited to features originally learned from general natural images rather than features adapted specifically to pigeon eye images.

The Vision Transformer and DINOv2-style models achieved an accuracy of 0.5602. Analysis of their confusion matrices showed that both models classified all validation images as belonging to the majority "bad" class. Therefore, the transformer-based models trained from scratch were not able to learn useful differences between the two classes with the available training dataset.

The results of this thesis show that relatively simple convolutional neural networks can perform well on a small and specific image classification problem. They also show the importance of evaluating models using several metrics rather than accuracy alone. Confusion matrix analysis was particularly useful because it revealed classification behavior that could not be clearly seen from accuracy values alone.

The experiments with data augmentation, learning rate, batch size, training duration, and early stopping also showed that training configuration can have a considerable effect on the final model performance. These results provide a basis for further development of deep learning methods for racing pigeon eye classification.

5.2 Future Work

The most important direction for future research is the expansion of the dataset. A larger number of pigeon eye images would allow the models to learn from a wider variety of visual characteristics and would provide a stronger basis for evaluating more complex architectures.

Future datasets should also include images from different breeders, countries, photographers, cameras, and lighting conditions. This would make it possible to evaluate whether the models can generalize to images that differ from those used in the current study. A larger dataset could also be divided into separate training, validation, and independent test sets.

More detailed information about each pigeon could also be included. In addition to the "good" and "bad" labels, future datasets could contain racing results, race distances, rankings, pedigree information, bloodlines, age, and other relevant characteristics. Labels could also be evaluated by several experienced breeders in order to compare different opinions about the same pigeons.

The transformer-based models should be investigated further using pretrained Vision Transformer and DINOv2 models instead of training them completely from scratch. A pretrained backbone could be combined with a classification head trained specifically on the pigeon eye dataset. This would provide a better evaluation of whether transformer-based representations are useful for this task.

Further experiments with ResNet50 and EfficientNetB0 could also include fine-tuning some of the pretrained backbone layers. In the current experiments, the backbones were completely frozen. Allowing some layers to adapt to pigeon eye images could improve the ability of these models to recognize iris texture, pigmentation, and other specific visual characteristics.

Data augmentation should also be studied further. A larger dataset and a greater number of training epochs would allow different augmentation settings to be tested more reliably. Less aggressive transformations could be compared with the augmentation strategy used in this thesis to determine which transformations are most suitable for pigeon eye images.

The results of the ablation study also indicate that learning rate, batch size, number of epochs, and early stopping settings should be investigated in more detail. A more systematic search of these parameters could potentially improve the performance of the

best CNN architectures.

Synthetic pigeon eye images generated using diffusion models could also be investigated as a possible way to increase dataset diversity. Such images should be carefully evaluated to ensure that they contain realistic visual characteristics before being included in model training.

Finally, explainability methods such as Grad-CAM [16] could be integrated into future models. This would make it possible to visualize which areas of the pigeon eye have the strongest influence on a classification decision. Such analysis could help determine whether the models are mainly using iris texture, pigmentation, circular structures, or other parts of the image when distinguishing between the two classes.

With a larger and more diverse dataset, improved model training, and better model interpretation, the approach presented in this thesis could provide a foundation for a future practical tool that supports breeders in the evaluation of racing pigeons.

References

[1] PIPA (Pigeon Paradise), "PIPA Milestones," pipa.be. Accessed: 2026-08-16.

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," Nature, vol. 521, no. 7553, pp. 436–444, 2015.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in Advances in Neural Information Processing Systems (NeurIPS), 2012, pp. 1097–1105.

[4] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv preprint arXiv:1409.1556, 2014.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 770–778.

[6] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," arXiv preprint arXiv:1502.03167, 2015.

[7] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," Journal of Machine Learning Research, vol. 15, no. 1, pp. 1929–1958, 2014.

[8] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," arXiv preprint arXiv:1412.6980, 2014.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2009, pp. 248–255.

[10] M. Tan and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in Proceedings of the International Conference on Machine Learning (ICML), 2019, pp. 6105–6114.

[11] A. Vaswani et al., "Attention Is All You Need," in Advances in Neural Information Processing Systems (NeurIPS), 2017, pp. 5998–6008.

[12] A. Dosovitskiy et al., "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale," in International Conference on Learning Representations (ICLR), 2021.

[13] M. Oquab et al., "DINOv2: Learning Robust Visual Features without Supervision," arXiv preprint arXiv:2304.07193, 2023.

[14] D. S. W. Ting, C. Y. Cheung, G. Lim, et al., "Development and Validation of a Deep Learning System for Diabetic Retinopathy and Related Eye Diseases Using Retinal Images from Multiethnic Populations with Diabetes," JAMA, vol. 318, no. 22, pp. 2211–2223, 2017.

[15] Z. Li, Y. He, S. Keel, W. Meng, R. T. Chang, and M. He, "Efficacy of a Deep Learning System for Detecting Glaucomatous Optic Neuropathy Based on Color Fundus Photographs," Ophthalmology, vol. 125, no. 8, pp. 1199–1206, 2018.

[16] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization," in Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2017, pp. 618–626.

[17] J. Cohen, "A Coefficient of Agreement for Nominal Scales," Educational and Psychological Measurement, vol. 20, no. 1, pp. 37–46, 1960.

[18] D. M. W. Powers, "Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness and Correlation," Journal of Machine Learning Technologies, vol. 2, no. 1, pp. 37–63, 2011.

[19] M. Abadi et al., "TensorFlow: A System for Large-Scale Machine Learning," in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI), 2016, pp. 265–283.

[20] M. Caron, H. Touvron, I. Misra, H. Jégou, J. Mairal, P. Bojanowski, and A. Joulin, "Emerging Properties in Self-Supervised Vision Transformers," in Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2021, pp. 9650–9660.

A Source Code

This appendix contains the source code used for dataset loading, model training, evaluation, result recording, and the ablation experiments described in this thesis. The implementation is organized into separate scripts for the individual model configurations together with a shared 'model_utils.py' module that provides common functions for dataset loading, prediction extraction, metric calculation, and result storage. The individual training scripts correspond to the architectures described in Section 2.4 and use the image sizes, batch sizes, and training durations summarized in Table 1. The ablation study script implements the additional experiments described in Section 3.7. It can also accept an optional command-line index, allowing a selected ablation configuration to be executed individually when needed. The source code is included to document the implementation used in the experimental part of the thesis and to show how the model configurations and evaluation procedure were organized.

`model_utils.py`

```
import numpy as np
from pathlib import Path
import tensorflow as tf
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score
from tensorflow.keras import layers
from tensorflow.keras.utils import get_custom_objects
import csv
from datetime import datetime

BASE_DIR = Path(__file__).resolve().parent
DATA_DIR = BASE_DIR / "data" / "images"
MODEL_DIR = BASE_DIR / "models"
MODEL_DIR.mkdir(exist_ok=True)


class PositionEmbedding(layers.Layer):
    def __init__(self, num_patches, projection_dim, **kwargs):
        super().__init__(**kwargs)
        self.num_patches = num_patches
        self.projection_dim = projection_dim

    def build(self, input_shape):
        self.position_embeddings = self.add_weight(
            shape=(1, self.num_patches, self.projection_dim),
            initializer="random_normal",
            trainable=True,
            name="position_embeddings",
        )
        super().build(input_shape)

    def call(self, inputs):
        return inputs + self.position_embeddings
```

```
    def get_config(self):
        config = super().get_config()
        config.update({
            "num_patches": self.num_patches,
            "projection_dim": self.projection_dim,
        })
        return config


get_custom_objects().update({"PositionEmbedding": PositionEmbedding})


def get_dataset_dirs():
    train_dir = DATA_DIR / "train"
    val_dir = DATA_DIR / "val"
    if not train_dir.exists() or not val_dir.exists():
        raise FileNotFoundError(
            "Place images in data/images/train/<class>/ and data/images/val/<class>/"
        )
    return train_dir, val_dir


def load_datasets(image_size=(128, 128), batch_size=32, augment=False,
                  preprocess_fn=None):
    train_dir, val_dir = get_dataset_dirs()
    train_ds = tf.keras.preprocessing.image_dataset_from_directory(
        train_dir,
        image_size=image_size,
        batch_size=batch_size,
        label_mode="int",
    )
    class_names = train_ds.class_names
    val_ds = tf.keras.preprocessing.image_dataset_from_directory(
        val_dir,
        image_size=image_size,
        batch_size=batch_size,
        label_mode="int",
    )

    if augment:
        augmentation = tf.keras.Sequential([
            tf.keras.layers.RandomFlip("horizontal"),
            tf.keras.layers.RandomRotation(0.12),
            tf.keras.layers.RandomZoom(0.12),
            tf.keras.layers.RandomContrast(0.1),
        ])
        train_ds = train_ds.map(
            lambda x, y: (augmentation(x, training=True), y),
            num_parallel_calls=tf.data.AUTOTUNE,
        )

    if preprocess_fn is not None:
        train_ds = train_ds.map(
            lambda x, y: (preprocess_fn(x), y),
            num_parallel_calls=tf.data.AUTOTUNE,
        )
        val_ds = val_ds.map(
            lambda x, y: (preprocess_fn(x), y),
            num_parallel_calls=tf.data.AUTOTUNE,
```

```
        )

    train_ds = train_ds.prefetch(tf.data.AUTOTUNE)
    val_ds = val_ds.prefetch(tf.data.AUTOTUNE)
    return train_ds, val_ds, class_names


def extract_labels_and_predictions(model, dataset, num_classes):
    y_true = []
    y_pred = []
    for images, labels in dataset:
        logits = model.predict(images, verbose=0)
        if num_classes == 1:
            preds = np.zeros(shape=(labels.shape[0],), dtype=np.int32)
        elif logits.ndim == 1 or logits.shape[-1] == 1:
            preds = (logits.ravel() > 0.5).astype(np.int32)
        else:
            preds = np.argmax(logits, axis=-1)
        y_true.append(labels.numpy().ravel())
        y_pred.append(preds)

    y_true = np.concatenate(y_true, axis=0)
    y_pred = np.concatenate(y_pred, axis=0)
    return y_true, y_pred


def evaluate_model(model, dataset, num_classes):
    y_true, y_pred = extract_labels_and_predictions(model, dataset, num_classes)
    metrics = {
        "accuracy": float(accuracy_score(y_true, y_pred)),
        "precision": None,
        "recall": None,
        "f1": None,
    }
    if num_classes > 1:
        metrics["precision"] = float(
            precision_score(y_true, y_pred, average="macro", zero_division=0)
        )
        metrics["recall"] = float(
            recall_score(y_true, y_pred, average="macro", zero_division=0)
        )
        metrics["f1"] = float(
            f1_score(y_true, y_pred, average="macro", zero_division=0)
        )
    else:
        metrics["precision"] = float(
            precision_score(y_true, y_pred, average="binary", zero_division=0)
        )
        metrics["recall"] = float(
            recall_score(y_true, y_pred, average="binary", zero_division=0)
        )
        metrics["f1"] = float(
            f1_score(y_true, y_pred, average="binary", zero_division=0)
        )
    return metrics


def print_metrics(name, metrics):
    print(f"\n{name} evaluation:")
```

```
    print(f" accuracy : {metrics['accuracy']:.4f}")
    print(f" precision: {metrics['precision']:.4f}")
    print(f" recall : {metrics['recall']:.4f}")
    print(f" f1 : {metrics['f1']:.4f}")


def save_results_to_csv(model_name, metrics):
    """Writes model results to results.csv"""
    try:
        csv_file = BASE_DIR / "results.csv"
        file_exists = csv_file.exists()
        with open(csv_file, 'a', newline='', encoding='utf-8') as f:
            writer = csv.writer(f)
            if not file_exists:
                writer.writerow(['Model', 'Timestamp', 'Accuracy', 'Precision',
                                 'Recall', 'F1'])
            timestamp = datetime.now().strftime("%Y-%m-%d %H:%M:%S")
            writer.writerow([
                model_name,
                timestamp,
                f"{metrics['accuracy']:.4f}",
                f"{metrics['precision']:.4f}" if metrics['precision'] is not None else "N/A",
                f"{metrics['recall']:.4f}" if metrics['recall'] is not None else "N/A",
                f"{metrics['f1']:.4f}" if metrics['f1'] is not None else "N/A"
            ])
        print(f"Results for {model_name} saved to results.csv")
    except Exception as e:
        print(f"Error saving to CSV: {e}")
```

train_cnn.py

```
import tensorflow as tf
from tensorflow.keras import layers, models
from model_utils import (
    load_datasets,
    evaluate_model,
    print_metrics,
    save_results_to_csv,
    MODEL_DIR,
)

IMAGE_SIZE = (128, 128)
BATCH_SIZE = 8
EPOCHS = 15


def build_model(num_classes):
    model = models.Sequential([
        layers.Input(shape=(*IMAGE_SIZE, 3)),
        layers.Rescaling(1.0 / 255),
        layers.Conv2D(16, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(32, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D(),
```

```
        layers.GlobalAveragePooling2D(),
        layers.Dense(64, activation="relu"),
        layers.Dropout(0.3),
        layers.Dense(num_classes, activation="softmax"),
    ])
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
        loss="sparse_categorical_crossentropy",
        metrics=["accuracy"],
    )
    return model


def main():
    train_ds, val_ds, class_names = load_datasets(
        image_size=IMAGE_SIZE,
        batch_size=BATCH_SIZE,
    )
    train_ds = train_ds.cache().prefetch(tf.data.AUTOTUNE)
    val_ds = val_ds.cache().prefetch(tf.data.AUTOTUNE)
    num_classes = len(class_names)
    print("Classes:", class_names)
    model = build_model(num_classes)
    model.summary()
    model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    metrics = evaluate_model(model, val_ds, num_classes)
    print_metrics("CNN", metrics)
    save_results_to_csv("CNN", metrics)
    model_path = MODEL_DIR / "cnn.keras"
    model.save(model_path)
    print(f"Model saved to: {model_path}")


if __name__ == "__main__":
    main()
```

train_simple.py

```
import tensorflow as tf
from tensorflow.keras import layers, models
from model_utils import load_datasets, evaluate_model, print_metrics, save_results_to_csv, MODEL_DIR

IMAGE_SIZE = (128, 128)
BATCH_SIZE = 32
EPOCHS = 15


def build_model(num_classes):
    model = models.Sequential([
        layers.Rescaling(1.0 / 255, input_shape=(*IMAGE_SIZE, 3)),
        layers.Conv2D(32, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(128, 3, activation="relu"),
        layers.MaxPooling2D(),
```

```
        layers.Flatten(),
        layers.Dense(128, activation="relu"),
        layers.Dropout(0.4),
        layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid"),
    ])
    loss = "sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy"
    model.compile(optimizer="adam", loss=loss, metrics=["accuracy"])
    return model


def main():
    train_ds, val_ds, class_names = load_datasets(image_size=IMAGE_SIZE, batch_size=BATCH_SIZE)
    num_classes = len(class_names)
    print("Classes:", class_names)
    model = build_model(num_classes)
    model.summary()
    model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    metrics = evaluate_model(model, val_ds, num_classes)
    print_metrics("Simple CNN", metrics)
    save_results_to_csv("Simple CNN", metrics)
    model_path = MODEL_DIR / "simple_cnn.h5"
    model.save(model_path)
    print(f"Model saved to: {model_path}")


if __name__ == "__main__":
    main()
```

train_cnn_aug.py

```
import tensorflow as tf
from tensorflow.keras import layers, models
from model_utils import load_datasets, evaluate_model, print_metrics, save_results_to_csv, MODEL_DIR

IMAGE_SIZE = (128, 128)
BATCH_SIZE = 32
EPOCHS = 15


def build_model(num_classes):
    model = models.Sequential([
        layers.Rescaling(1.0 / 255, input_shape=(*IMAGE_SIZE, 3)),
        layers.Conv2D(32, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(128, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(128, activation="relu"),
        layers.Dropout(0.4),
        layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid"),
    ])
    loss = "sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy"
    model.compile(optimizer="adam", loss=loss, metrics=["accuracy"])
    return model
```

```
def main():
    train_ds, val_ds, class_names = load_datasets(
        image_size=IMAGE_SIZE,
        batch_size=BATCH_SIZE,
        augment=True,
    )
    num_classes = len(class_names)
    print("Classes:", class_names)
    model = build_model(num_classes)
    model.summary()
    model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    metrics = evaluate_model(model, val_ds, num_classes)
    print_metrics("CNN + Augmentation", metrics)
    save_results_to_csv("CNN + Augmentation", metrics)
    model_path = MODEL_DIR / "cnn_augmentation.h5"
    model.save(model_path)
    print(f"Model saved to: {model_path}")


if __name__ == "__main__":
    main()
```

train_bn.py

```
import tensorflow as tf
from tensorflow.keras import layers, models
from model_utils import load_datasets, evaluate_model, print_metrics, save_results_to_csv, MODEL_DIR

IMAGE_SIZE = (128, 128)
BATCH_SIZE = 8
EPOCHS = 18


def build_model(num_classes):
    model = models.Sequential([
        layers.Rescaling(1.0 / 255, input_shape=(*IMAGE_SIZE, 3)),
        layers.Conv2D(32, 3, activation="relu"),
        layers.BatchNormalization(),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.BatchNormalization(),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.BatchNormalization(),
        layers.MaxPooling2D(),
        layers.Conv2D(96, 3, activation="relu"),
        layers.BatchNormalization(),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(256, activation="relu"),
        layers.Dropout(0.4),
        layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid"),
    ])
    loss = "sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy"
```

```
    model.compile(optimizer="adam", loss=loss, metrics=["accuracy"])
    return model


def main():
    train_ds, val_ds, class_names = load_datasets(image_size=IMAGE_SIZE, batch_size=BATCH_SIZE)
    num_classes = len(class_names)
    print("Classes:", class_names)
    model = build_model(num_classes)
    model.summary()
    model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    metrics = evaluate_model(model, val_ds, num_classes)
    print_metrics("Batch Norm CNN", metrics)
    save_results_to_csv("Batch Norm CNN", metrics)
    model_path = MODEL_DIR / "batchnorm_cnn.h5"
    model.save(model_path)
    print(f"Model saved to: {model_path}")


if __name__ == "__main__":
    main()
```

train_deep.py

```
import tensorflow as tf
from tensorflow.keras import layers, models
from model_utils import load_datasets, evaluate_model, print_metrics, save_results_to_csv, MODEL_DIR

IMAGE_SIZE = (128, 128)
BATCH_SIZE = 32
EPOCHS = 18


def build_model(num_classes):
    model = models.Sequential([
        layers.Rescaling(1.0 / 255, input_shape=(*IMAGE_SIZE, 3)),
        layers.Conv2D(16, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(32, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(256, activation="relu"),
        layers.Dropout(0.4),
        layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid"),
    ])
    loss = "sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy"
    model.compile(optimizer="adam", loss=loss, metrics=["accuracy"])
    return model


def main():
    train_ds, val_ds, class_names = load_datasets(image_size=IMAGE_SIZE, batch_size=BATCH_SIZE)
```

```
    num_classes = len(class_names)
    print("Classes:", class_names)
    model = build_model(num_classes)
    model.summary()
    model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    metrics = evaluate_model(model, val_ds, num_classes)
    print_metrics("Deep CNN", metrics)
    save_results_to_csv("Deep CNN", metrics)
    model_path = MODEL_DIR / "deep_cnn.h5"
    model.save(model_path)
    print(f"Model saved to: {model_path}")


if __name__ == "__main__":
    main()
```

train_resnet50.py

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input
from model_utils import load_datasets, evaluate_model, print_metrics, save_results_to_csv, MODEL_DIR

IMAGE_SIZE = (224, 224)
BATCH_SIZE = 16
EPOCHS = 12


def build_model(num_classes):
    base_model = ResNet50(
        include_top=False,
        weights="imagenet",
        input_shape=(*IMAGE_SIZE, 3),
    )
    base_model.trainable = False

    inputs = layers.Input(shape=(*IMAGE_SIZE, 3))
    x = preprocess_input(inputs)
    x = base_model(x, training=False)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dropout(0.3)(x)
    x = layers.Dense(128, activation="relu")(x)
    x = layers.Dropout(0.3)(x)
    outputs = layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid")(x)
    model = models.Model(inputs, outputs)

    loss = "sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy"
    model.compile(optimizer="adam", loss=loss, metrics=["accuracy"])
    return model


def main():
    train_ds, val_ds, class_names = load_datasets(
        image_size=IMAGE_SIZE,
        batch_size=BATCH_SIZE,
        preprocess_fn=preprocess_input,
```

```
    )
    num_classes = len(class_names)
    print("Classes:", class_names)
    model = build_model(num_classes)
    model.summary()
    model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    metrics = evaluate_model(model, val_ds, num_classes)
    print_metrics("ResNet50", metrics)
    save_results_to_csv("ResNet50", metrics)
    model_path = MODEL_DIR / "resnet50.h5"
    model.save(model_path)
    print(f"Model saved to: {model_path}")


if __name__ == "__main__":
    main()
```

train_efficientnet.py

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications.efficientnet import EfficientNetB0, preprocess_input
from model_utils import load_datasets, evaluate_model, print_metrics, save_results_to_csv, MODEL_DIR

IMAGE_SIZE = (224, 224)
BATCH_SIZE = 16
EPOCHS = 12


def build_model(num_classes):
    base_model = EfficientNetB0(
        include_top=False,
        weights="imagenet",
        input_shape=(*IMAGE_SIZE, 3),
    )
    base_model.trainable = False

    inputs = layers.Input(shape=(*IMAGE_SIZE, 3))
    x = preprocess_input(inputs)
    x = base_model(x, training=False)
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dropout(0.3)(x)
    x = layers.Dense(128, activation="swish")(x)
    x = layers.Dropout(0.3)(x)
    outputs = layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid")(x)
    model = models.Model(inputs, outputs)

    loss = "sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy"
    model.compile(optimizer="adam", loss=loss, metrics=["accuracy"])
    return model


def main():
    train_ds, val_ds, class_names = load_datasets(
        image_size=IMAGE_SIZE,
        batch_size=BATCH_SIZE,
```

```
        preprocess_fn=preprocess_input,
    )
    num_classes = len(class_names)
    print("Classes:", class_names)
    model = build_model(num_classes)
    model.summary()
    model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    metrics = evaluate_model(model, val_ds, num_classes)
    print_metrics("EfficientNetB0", metrics)
    save_results_to_csv("EfficientNetB0", metrics)
    model_path = MODEL_DIR / "efficientnetb0.h5"
    model.save(model_path)
    print(f"Model saved to: {model_path}")


if __name__ == "__main__":
    main()
```

train_vit.py

```
import tensorflow as tf
from tensorflow.keras import layers, models
from model_utils import PositionEmbedding, load_datasets, evaluate_model, print_metrics,
     save_results_to_csv, MODEL_DIR

IMAGE_SIZE = (128, 128)
BATCH_SIZE = 16
EPOCHS = 12
PATCH_SIZE = 16
PROJECTION_DIM = 64
NUM_HEADS = 4
TRANSFORMER_LAYERS = 4
MLP_UNITS = [128]


def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation="gelu")(x)
        x = layers.Dropout(dropout_rate)(x)
    return x


def build_vit_classifier(num_classes):
    num_patches = (IMAGE_SIZE[0] // PATCH_SIZE) * (IMAGE_SIZE[1] // PATCH_SIZE)
    inputs = layers.Input(shape=(*IMAGE_SIZE, 3))
    x = layers.Rescaling(1.0 / 255)(inputs)
    x = layers.Conv2D(
        PROJECTION_DIM,
        kernel_size=PATCH_SIZE,
        strides=PATCH_SIZE,
        padding="valid",
    )(x)
    x = layers.Reshape((num_patches, PROJECTION_DIM))(x)
    x = PositionEmbedding(num_patches, PROJECTION_DIM)(x)

    for _ in range(TRANSFORMER_LAYERS):
```

```
        x1 = layers.LayerNormalization(epsilon=1e-6)(x)
        attention_output = layers.MultiHeadAttention(
            num_heads=NUM_HEADS,
            key_dim=PROJECTION_DIM,
        )(x1, x1)
        x = layers.Add()([x, attention_output])
        x1 = layers.LayerNormalization(epsilon=1e-6)(x)
        x = layers.Add()([x, mlp(x1, [PROJECTION_DIM * 2, PROJECTION_DIM], 0.1)])

    x = layers.LayerNormalization(epsilon=1e-6)(x)
    x = layers.GlobalAveragePooling1D()(x)
    x = mlp(x, MLP_UNITS, 0.4)
    outputs = layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid")(x)
    return models.Model(inputs=inputs, outputs=outputs)


def main():
    train_ds, val_ds, class_names = load_datasets(image_size=IMAGE_SIZE, batch_size=BATCH_SIZE)
    num_classes = len(class_names)
    print("Classes:", class_names)
    model = build_vit_classifier(num_classes)
    model.compile(
        optimizer="adam",
        loss="sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy",
        metrics=["accuracy"],
    )
    model.summary()
    model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    metrics = evaluate_model(model, val_ds, num_classes)
    print_metrics("ViT", metrics)
    save_results_to_csv("ViT", metrics)
    model_path = MODEL_DIR / "vit.keras"
    model.save(model_path)
    print(f"Model saved to: {model_path}")


if __name__ == "__main__":
    main()
```

train_dinov2.py

```
import tensorflow as tf
from tensorflow.keras import layers, models
from model_utils import PositionEmbedding, load_datasets, evaluate_model, print_metrics,
     save_results_to_csv, MODEL_DIR

IMAGE_SIZE = (128, 128)
BATCH_SIZE = 16
EPOCHS = 12
PATCH_SIZE = 16
PROJECTION_DIM = 64
NUM_HEADS = 4
TRANSFORMER_LAYERS = 4
MLP_HEAD_UNITS = [256, 128]
```

```
def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation="gelu")(x)
        x = layers.Dropout(dropout_rate)(x)
    return x


def build_dinov2_style_model(num_classes):
    num_patches = (IMAGE_SIZE[0] // PATCH_SIZE) * (IMAGE_SIZE[1] // PATCH_SIZE)
    inputs = layers.Input(shape=(*IMAGE_SIZE, 3))
    x = layers.Rescaling(1.0 / 255)(inputs)
    x = layers.Conv2D(
        PROJECTION_DIM,
        kernel_size=PATCH_SIZE,
        strides=PATCH_SIZE,
        padding="valid",
    )(x)
    x = layers.Reshape((num_patches, PROJECTION_DIM))(x)
    x = PositionEmbedding(num_patches, PROJECTION_DIM)(x)

    for _ in range(TRANSFORMER_LAYERS):
        x1 = layers.LayerNormalization(epsilon=1e-6)(x)
        attention_output = layers.MultiHeadAttention(
            num_heads=NUM_HEADS,
            key_dim=PROJECTION_DIM,
        )(x1, x1)
        x = layers.Add()([x, attention_output])
        x1 = layers.LayerNormalization(epsilon=1e-6)(x)
        x = layers.Add()([x, mlp(x1, [PROJECTION_DIM * 2, PROJECTION_DIM], 0.1)])

    x = layers.LayerNormalization(epsilon=1e-6)(x)
    x = layers.GlobalAveragePooling1D()(x)
    projection = mlp(x, MLP_HEAD_UNITS, 0.2)
    projection = layers.LayerNormalization(epsilon=1e-6)(projection)
    outputs = layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid")(projection
    )
    return models.Model(inputs=inputs, outputs=outputs)


def main():
    train_ds, val_ds, class_names = load_datasets(image_size=IMAGE_SIZE, batch_size=BATCH_SIZE)
    num_classes = len(class_names)
    print("Classes:", class_names)
    model = build_dinov2_style_model(num_classes)
    model.compile(
        optimizer="adam",
        loss="sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy",
        metrics=["accuracy"],
    )
    model.summary()
    model.fit(train_ds, validation_data=val_ds, epochs=EPOCHS)
    metrics = evaluate_model(model, val_ds, num_classes)
    print_metrics("DINOv2-style", metrics)
    save_results_to_csv("DINOv2-style", metrics)
    model_path = MODEL_DIR / "dinov2.keras"
    model.save(model_path)
    print(f"Model saved to: {model_path}")
```

```
if __name__ == "__main__":
    main()
```

evaluate_models.py

```
from pathlib import Path
import tensorflow as tf
from tensorflow.keras.models import load_model
from model_utils import load_datasets, evaluate_model, MODEL_DIR

MODEL_FILES = [
    ("CNN", "cnn"),
    ("CNN + Augmentation", "cnn_augmentation"),
    ("ResNet50", "resnet50"),
    ("EfficientNet", "efficientnetb0"),
    ("ViT", "vit"),
    ("DINOv2", "dinov2"),
]
MODEL_EXTENSIONS = [".keras", ".h5"]


def find_model_path(base_name):
    for ext in MODEL_EXTENSIONS:
        candidate = MODEL_DIR / f"{base_name}{ext}"
        if candidate.exists():
            return candidate
    return None


def format_value(value):
    if value is None:
        return "N/A"
    return f"{value:.4f}"


def main():
    train_ds, val_ds, class_names = load_datasets()
    num_classes = len(class_names)
    print("Classes:", class_names)
    print("\nEvaluation of saved models on the validation set:\n")

    table = [
        "| Model | Accuracy | Precision | Recall | F1 |",
        "| ------------------ | -------- | --------- | ------ | -- |",
    ]

    for label, base_name in MODEL_FILES:
        model_path = find_model_path(base_name)
        if model_path is None:
            table.append(f"| {label} | missing | missing | missing | missing |")
            continue
        try:
            model = load_model(model_path)
        except Exception as exc:
            print(f"Could not load model {model_path}: {exc}")
            table.append(f"| {label} | error | error | error | error |")
```

```
            continue

        metrics = evaluate_model(model, val_ds, num_classes)
        table.append(
            f"| {label} | {format_value(metrics['accuracy'])} | "
            f"{format_value(metrics['precision'])} | {format_value(metrics['recall'])} | "
            f"{format_value(metrics['f1'])} |"
        )

    print("\n".join(table))
    print("\nMissing models must first be trained and saved in models/.")


if __name__ == "__main__":
    main()
```

ablation_study.py

```
import csv
import os
import sys
import time
from pathlib import Path

os.environ.setdefault("TF_ENABLE_ONEDNN_OPTS", "0")

import tensorflow as tf
from tensorflow.keras import layers, models
from model_utils import load_datasets, evaluate_model, MODEL_DIR

BASE_DIR = Path(__file__).resolve().parent
RESULTS_FILE = BASE_DIR / "ablation_results.csv"
IMAGE_SIZE = (128, 128)


def build_simple_cnn(num_classes, learning_rate):
    model = models.Sequential([
        layers.Rescaling(1.0 / 255, input_shape=(*IMAGE_SIZE, 3)),
        layers.Conv2D(32, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(128, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(128, activation="relu"),
        layers.Dropout(0.4),
        layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid"),
    ])
    loss = "sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy"
    optimizer = tf.keras.optimizers.Adam(learning_rate=learning_rate)
    model.compile(optimizer=optimizer, loss=loss, metrics=["accuracy"])
    return model


CONFIGS = [
```

```
    {"name": "LR=0.0001", "lr": 0.0001, "batch": 32, "epochs": 15, "early_stopping": False},
    {"name": "LR=0.001 (default)", "lr": 0.001, "batch": 32, "epochs": 15, "early_stopping": False},
    {"name": "LR=0.01", "lr": 0.01, "batch": 32, "epochs": 15, "early_stopping": False},
    {"name": "Batch=8", "lr": 0.001, "batch": 8, "epochs": 15, "early_stopping": False},
    {"name": "Batch=32 (default)", "lr": 0.001, "batch": 32, "epochs": 15, "early_stopping": False},
    {"name": "Batch=64", "lr": 0.001, "batch": 64, "epochs": 15, "early_stopping": False},
    {"name": "Epochs=30, no early stopping", "lr": 0.001, "batch": 32, "epochs": 30, "early_stopping":
     False},
    {"name": "Epochs=30, with early stopping", "lr": 0.001, "batch": 32, "epochs": 30, "early_stopping":
     True},
]


def run_config(cfg):
    train_ds, val_ds, class_names = load_datasets(
        image_size=IMAGE_SIZE,
        batch_size=cfg["batch"],
    )
    num_classes = len(class_names)
    model = build_simple_cnn(num_classes, cfg["lr"])
    callbacks = []
    if cfg["early_stopping"]:
        callbacks.append(
            tf.keras.callbacks.EarlyStopping(
                monitor="val_accuracy",
                patience=3,
                restore_best_weights=True,
            )
        )

    start = time.time()
    history = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=cfg["epochs"],
        callbacks=callbacks,
        verbose=2,
    )
    elapsed = time.time() - start
    epochs_run = len(history.history["loss"])
    metrics = evaluate_model(model, val_ds, num_classes)

    return {
        "Config": cfg["name"],
        "LearningRate": cfg["lr"],
        "BatchSize": cfg["batch"],
        "MaxEpochs": cfg["epochs"],
        "EarlyStopping": cfg["early_stopping"],
        "EpochsRun": epochs_run,
        "TrainingSeconds": round(elapsed, 1),
        "Accuracy": round(metrics["accuracy"], 4),
        "Precision": round(metrics["precision"], 4),
        "Recall": round(metrics["recall"], 4),
        "F1": round(metrics["f1"], 4),
    }


def main():
    fieldnames = [
```

```
        "Config", "LearningRate", "BatchSize", "MaxEpochs", "EarlyStopping",
        "EpochsRun", "TrainingSeconds", "Accuracy", "Precision", "Recall", "F1",
    ]

    if len(sys.argv) > 1:
        indices = [int(sys.argv[1])]
    else:
        indices = list(range(len(CONFIGS)))

    file_exists = RESULTS_FILE.exists()
    with open(RESULTS_FILE, "a", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        if not file_exists:
            writer.writeheader()

        for i in indices:
            cfg = CONFIGS[i]
            print(f"\n{'='*60}\nRunning ablation config [{i}]: {cfg['name']}\n{'='*60}")
            row = run_config(cfg)
            writer.writerow(row)
            f.flush()
            print(f"Result: {row}")

    print(f"\nAblation run complete for indices {indices}. Results saved to: {RESULTS_FILE}")


if __name__ == "__main__":
    main()
```

deepcnn_batch8_test.py

```
import csv
import os
import time
from pathlib import Path

os.environ.setdefault("TF_ENABLE_ONEDNN_OPTS", "0")

import tensorflow as tf
from tensorflow.keras import layers, models
from model_utils import load_datasets, evaluate_model

BASE_DIR = Path(__file__).resolve().parent
RESULTS_FILE = BASE_DIR / "ablation_results.csv"
IMAGE_SIZE = (128, 128)
BATCH_SIZE = 8
EPOCHS = 18
CONFIG_NAME = "Deep CNN + Batch=8"


def build_deep_cnn(num_classes):
    model = models.Sequential([
        layers.Rescaling(1.0 / 255, input_shape=(*IMAGE_SIZE, 3)),
        layers.Conv2D(16, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(32, 3, activation="relu"),
```

```
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Conv2D(64, 3, activation="relu"),
        layers.MaxPooling2D(),
        layers.Flatten(),
        layers.Dense(256, activation="relu"),
        layers.Dropout(0.4),
        layers.Dense(num_classes, activation="softmax" if num_classes > 1 else "sigmoid"),
    ])
    loss = "sparse_categorical_crossentropy" if num_classes > 1 else "binary_crossentropy"
    model.compile(optimizer="adam", loss=loss, metrics=["accuracy"])
    return model


def main():
    train_ds, val_ds, class_names = load_datasets(
        image_size=IMAGE_SIZE,
        batch_size=BATCH_SIZE,
    )
    num_classes = len(class_names)
    model = build_deep_cnn(num_classes)

    start = time.time()
    history = model.fit(
        train_ds,
        validation_data=val_ds,
        epochs=EPOCHS,
        verbose=2,
    )
    elapsed = time.time() - start
    metrics = evaluate_model(model, val_ds, num_classes)

    row = {
        "Config": CONFIG_NAME,
        "LearningRate": 0.001,
        "BatchSize": BATCH_SIZE,
        "MaxEpochs": EPOCHS,
        "EarlyStopping": False,
        "EpochsRun": len(history.history["loss"]),
        "TrainingSeconds": round(elapsed, 1),
        "Accuracy": round(metrics["accuracy"], 4),
        "Precision": round(metrics["precision"], 4),
        "Recall": round(metrics["recall"], 4),
        "F1": round(metrics["f1"], 4),
    }
    print(f"Result: {row}")

    fieldnames = list(row.keys())
    try:
        file_exists = RESULTS_FILE.exists()
        with open(RESULTS_FILE, "a", newline="", encoding="utf-8") as f:
            writer = csv.DictWriter(f, fieldnames=fieldnames)
            if not file_exists:
                writer.writeheader()
            writer.writerow(row)
        print(f"Saved to {RESULTS_FILE}")
    except PermissionError as e:
        print(f"Could not write results to {RESULTS_FILE}: {e}")
```

```
if __name__ == "__main__":
    main()
```

B Main Experimental Results

This appendix presents the final validation results used in the main comparison of the nine model configurations in Chapter 3. The values correspond to the results reported in Table 2 and are shown again here in a compact form together with all four evaluation metrics. Timestamps and intermediate experimental records are not included because they are not required for the comparison of model performance.

Table B.1: Final validation results used in the main model comparison.

Model	Accuracy	Precision	Recall	F1
CNN	0.5301	0.3450	0.4746	0.3577
ResNet50	0.6988	0.6979	0.6840	0.6856
EfficientNetB0	0.6205	0.6657	0.5773	0.5355
Batch Norm CNN	0.6747	0.7201	0.6979	0.6708
ViT	0.5602	0.2801	0.5000	0.3591
CNN + Augmentation	0.5904	0.6083	0.5446	0.4888
Deep CNN	0.7470	0.7434	0.7447	0.7440
DINOv2-style	0.5602	0.2801	0.5000	0.3591
Simple CNN	0.7349	0.7318	0.7340	0.7325

The best result in the main comparison was achieved by Deep CNN, with an accuracy of 0.7470 and an F1 score of 0.7440. Simple CNN achieved the second-best accuracy of 0.7349 and an F1 score of 0.7325. Both models also showed relatively balanced precision and recall values, which is consistent with the confusion matrix analysis presented in Section 3.3.

ResNet50 achieved an accuracy of 0.6988, while EfficientNetB0 achieved 0.6205. These transfer learning models therefore performed below the two best custom CNN architectures in the main comparison. Batch Norm CNN achieved an accuracy of 0.6747 but showed a different class prediction pattern, as discussed in Section 3.3.

The Vision Transformer and DINOv2-style models both achieved an accuracy of 0.5602, precision of 0.2801, recall of 0.5000, and F1 score of 0.3591. As shown in Section 3.3, these values correspond to prediction of the majority "bad" class for all validation images. The baseline CNN achieved an accuracy of 0.5301, which was below the majority-class baseline.

The results in this appendix are therefore consistent with the comparison and discussion presented throughout Sections 3.2–3.6.

C Ablation Study Results

This appendix presents the detailed results of the ablation experiments described in Section 3.7. The Simple CNN architecture was used as the main base model for examining the effects of learning rate, batch size, training duration, and early stopping. An additional Deep CNN experiment with batch size 8 was also included to test whether the strongest batch-size result from the Simple CNN experiments would improve the best-performing architecture from the main comparison.

Table C.1: Detailed ablation study results.

Config	LearningRate	BatchSize	MaxEpochs	EarlyStopping	EpochsRun	TrainingSeconds	Accuracy	Precision	Recall	F1
LR=0.0001	0.0001	32	15	False	15	351.3	0.6446	0.6792	0.6077	0.5848
LR=0.001	0.001	32	15	False	15	346.4	0.6627	0.6810	0.6768	0.6622
LR=0.01	0.01	32	15	False	15	349.5	0.5602	0.2801	0.5000	0.3591
Batch=8	0.001	8	15	False	15	376.3	0.7289	0.7334	0.7360	0.7287
Batch=32	0.001	32	15	False	15	354.9	0.7048	0.7009	0.6953	0.6968
Batch=64	0.001	64	15	False	15	345.0	0.6867	0.7362	0.6527	0.6397
Epochs=30 no early stopping	0.001	32	30	False	30	708.6	0.6747	0.7051	0.6935	0.6730
Epochs=30 with early stopping	0.001	32	30	True	4	95.9	0.5602	0.2801	0.5000	0.3591
Deep CNN + Batch=8	0.001	8	18	False	18	368.9	0.6205	0.6124	0.6053	0.6049

The learning-rate experiments show that the selected learning rate had a clear influence on the final classification performance. A learning rate of 0.0001 produced an accuracy of 0.6446, while 0.001 achieved 0.6627. Increasing the learning rate to 0.01 resulted in an accuracy of 0.5602 and an F1 score of 0.3591. These values correspond to majority-class prediction and show that the larger learning rate was not suitable for this training configuration.

Batch size also affected the obtained results. Batch size 8 achieved the strongest result in this part of the ablation study, with an accuracy of 0.7289, recall of 0.7360, and F1 score of 0.7287. Batch size 32 achieved an accuracy of 0.7048, while batch size 64 achieved 0.6867. The result therefore suggests that the smaller batch size was more suitable for the Simple CNN in these experiments.

The effect of a longer training period was evaluated by increasing the maximum number of epochs from 15 to 30. The 30-epoch configuration without early stopping finished with an accuracy of 0.6747, recall of 0.6935, and F1 score of 0.6730. As discussed in Section 3.7, validation accuracy reached a higher value during the later stages of training before decreasing again, while training accuracy continued to increase. This pattern indicates overfitting during the later epochs and shows that simply increasing the maximum training duration does not necessarily improve the final validation result.

The early-stopping configuration used a patience value of three epochs and stopped after four epochs. It achieved an accuracy of 0.5602, recall of 0.5000, and F1 score of 0.3591. The validation accuracy remained at the majority-class level during the initial training period, so the selected patience value stopped training before the model moved beyond this initial classification behavior. A larger patience value could therefore be

considered in future experiments.

The additional Deep CNN experiment used batch size 8 with a learning rate of 0.001 and a maximum of 18 epochs. The completed evaluation achieved an accuracy of 0.6205, precision of 0.6124, recall of 0.6053, and F1 score of 0.6049. This result did not improve on the original Deep CNN accuracy of 0.7470 reported in Table 2. The experiment therefore shows that a batch size that performs well with one architecture does not necessarily produce the same improvement when applied to another architecture.

Together, these ablation results support the discussion in Section 3.7 that model performance is affected not only by architecture but also by the selected training settings. Learning rate, batch size, training duration, and stopping criteria all influenced the obtained validation results and should therefore be considered carefully in future experiments.

Biography

Vasilije Simonović was born in 1991 in Sombor, Serbia. He completed high school at the Economic High School in Sombor. He then continued his education at the University of Novi Sad, Faculty of Sciences, where he completed a Bachelor's degree in Applied Mathematics – Financial Mathematics. He is currently completing his master's studies in Data Science at the same faculty, within the Department of Mathematics and Informatics.

Professionally, he works as a Senior AI & Automation Engineer at Simplify. His work is focused on the design and implementation of automation and artificial intelligence solutions, including robotic process automation, machine learning, intelligent document processing, and the practical application of AI technologies in business processes.

He is also the creator of the PremiumPedigree platform (`www.premiumpedigree.com`), developed for the professional creation and management of racing pigeon pedigrees. The platform applies artificial intelligence for extracting and processing pedigree data from different types of images and PDF documents, reducing the amount of manual data entry required when creating pedigrees.

His professional experience in artificial intelligence and automation, together with his long-standing interest in racing pigeon sport, provided the motivation for this master's thesis and for investigating the application of deep learning methods to pigeon eye image classification.

UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCES
KEY WORDS DOCUMENTATION

Accession number: **ANO**
Identification number: **INO**
Document type: monograph type **DT**
Type of record: printed text **TR**
Contents code: master thesis **CC**
Author: Vasilije Simonović **AU**
Mentor: PhD Miloš Savić **MN**
Title: Classification of Racing Pigeons Based on Eye Images Using Deep Learning Methods **TI**
Language of text: English **LT**
Language of abstract: English **LA**
Country of publication: Republic of Serbia **CP**
Locality of publication: Vojvodina **LP**
Publication year: 2026 **PY**
Publisher: author's reprint **PU**
Publication place: Novi Sad, Trg D. Obradovića 3 **PP**
Physical description (chapters/pages/references/tables/ pictures/charts/supplements): (5/58/20/7/2/0/3) **PD**
Scientific field: computer science **SF**
Scientific discipline: artificial intelligence and machine learning **SD**
Subject, key words: racing pigeons, pigeon eye images, deep learning, convolutional neural networks, image classification, transfer learning, Vision Transformer, DINOv2 **SKW**
UC: **UC**
Holding data: Library of Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad **HD**
Note: N **N**

Abstract: This thesis investigates whether deep learning models can classify racing pigeons into "good" and "bad" quality categories using only photographs of their eyes. The dataset contains 826 pigeon eye images, including 348 images labeled as "good" and 478 labeled as "bad". Nine model configurations were evaluated, including custom convolutional neural networks, ImageNet-pretrained transfer learning models, a Vision Transformer, and a DINOv2-style model. The best results were achieved by Deep CNN, with an accuracy of 0.7470 and an F1 score of 0.7440, followed by Simple CNN with an accuracy of 0.7349 and an F1 score of 0.7325. The results show that relatively simple convolutional architectures can perform well on a small, domain-specific dataset and that accuracy should be interpreted together with precision, recall, F1 score, and confusion matrices. **AB**

Accepted on Scientific Board on: 23.6.2026 **AS**
Defended: **DE**
Thesis Defense Board:

President: PhD Davor Kumozec, assistant professor, Faculty of Sciences, University of Novi Sad

Member: PhD Vladimir Kurbalija, full professor, Faculty of Sciences, University of Novi Sad

Mentor: PhD Miloš Savić, full professor, Faculty of Sciences, University of Novi Sad

DB