A Neural Network Approach to Bird Classification

—

Jvosten

Fachbereich Vogelwissenschaften
Science Str. 1
99001 Berlin

Jvosten
M.Nr.:654321
5. Fachsemester

Contents

1 Introduction

As an avid bird watcher I face a classification problem every single time I find my way into the field: to determine bird species. Some birds are easy to label due to their size or unique color pattern. But other bird species are more difficult to identify, for instance the family of woodpeckers inhabiting the south east of Canada. Most of these woodpeckers share phaenotypical features: they have a black body with white sprinkles toward the chest, a white chest and a read head. But the *Yellow Bellied Sapsucker* for instance has a yellow chest. As we can see in Figure 1, there are some differences between those types, but overall there is a great similarity. Spotting *a* woodpecker from a distance of 10 to 25 meter through binoculars is relatively easy, due to its particular appearance and movements (the pecking!). But to determine the exact species, it takes some experience or patience to identify the birds species by comparing its observed feature with some look-up table. A bird watcher usually employs a combination of methods for classifying an observed bird:

- Observe the birds movement; a hawk circles in a different manner then a vulture
- Listen to its singing; a black bird is easy to identify due to its distincitve singing
- Look at the birds external anatomy:

Figure 1: *Sapsucker*

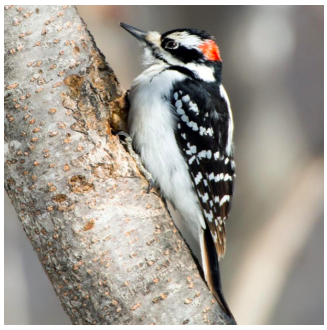

Figure 2: *Hairy Woodpecker*

Figure 3: *Downy Woodpecker*

In this paper we will be interested in the last classification method. As any bird species can be exactly identified by looking at its body parts and their shape, size and color, this makes it a perfect classification task for the bird watcher to delegate it to a machine learning algorithm. A made observation in the woods only would have to be transformed into a photographic image and an (appropiate) learning algorithm could unfold its magic and tell the bird watcher, what he was just seeing. This would be a great tool for anyone watching birds. Furthermore there would also be a great scientific use for this kind of application.

The goal of this paper is to give a sketch for a deep learning architecture which could be the core of a potential bird identification application.[1] Therefore we compare four different *Convolutional Neural Network* approaches for building a classifier from scratch: a baseline model; an augmented baseline model; and two types of pretrained networks. For training the models we will use a kaggle data set, which contains images of 210 different bird species.

[1]Of course there already exist a comparable project: Svarovski company is developing a binocular which has a build-in camera for bird identification. We will discuss their approach in the final section of this paper.

Due to the computational constraint of not employing a GPU for the training process, we downsample the data set for the training process for most of the models; two approaches will be evaluted on two different sample sizes, the other two only on one, resulting in six models total for the final comparison.

The paper is organised as follows: Section 2 describes the basic method used in this paper, the concept of Convolutional Neural Networks. In Section 3, we shortly discuss data preparation. In the following Sections 4 and 5 we explain the specifics of the applied models and analyze the results of their application to the data. In the last section we draw a conclusion.

2 Methodology

Convolutional Neual Networks (CNNs) are the predominant type of deep-learning models when it comes to image-classification problems and more general for problems involving grid-like data. While Artificial Neural Networks (ANNs), the base for any deep-learning model, are inspired by biological neural networks, the concept of CNNs is inspired by the human visual cortex. Therefore CNNs are considered a special case of ANNs: "Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers." (Goodfellow et al., 2016, p.326). That means, instead of learning global patterns from the entire input space, convolutional layers learn local patterns (see Figure 4).

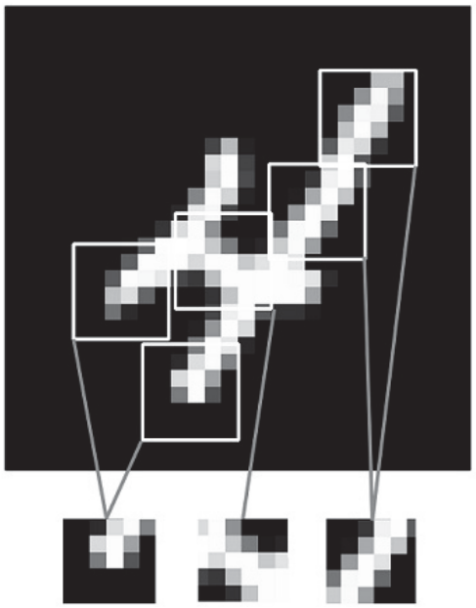

Figure 4: Learning local patterns of a digit

When it comes to image classification, the CNN approach has two great advantages over a normal neural network:

- It can learn *spatial hierachies*. That means, the first layer of the network learns small local patterns, for instance parts of the contour of a bird. The next layer will then

learn larger patterns made of small extracted patterns of the first layer, for instance a certain arrangement of small patterns forms a birds beak. With every layer the CNN learns more and more abstract patterns.

- Learnt patterns are *translation-invariant*. If the CNN once learns a certain pattern in the middle of the picture, for instance a birds eye, it can recognize it anywhere else. That makes the CNN very data efficient, as it does not require large amounts of training data to achieve high generalization power.

For a further understanding of the functioning of a CNN we will look closely at the structure of an entire convolutional layer. Furthermore we will give a brief sketch of the overall network architecture.

A convolutional layer for image classification takes a so called 3D tensor as its input. A tensor can be understood as a generalization of the concepts of scalar, vector and matrix. For images, a 3D tensor contains two spatial axis, height and width, and a depth axis. For a black-and-white image, the dimension of the depth axis is one, for a coloured, RGB image, the depth dimension is three, as the image contains three colour channels. In the convolutional operation, the *kernel* slides over the input 3D tensor, which is also called *feature map*, and in each step along the way it computes the dot product between the kernel and the extracted patch of the image. The output of this operation is called *output feature map*. In Figure 5 we can see a $5 \times 5 \times 3$ kernel sliding over a $32 \times 32 \times 3$ image, creating a $28 \times 28 \times 1$ output feature map, because there are 28×28 unique positions the kernel can be located at.

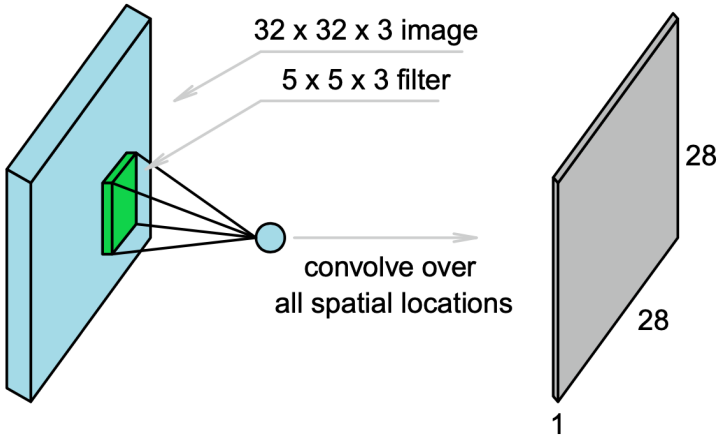

Figure 5: A kernel slides over the entire image to detect local patterns

This output feature map is still a 3D tensor, since it contains a height, weight and depth dimension. But in the example in Figure 5 the depth of the output is not the same as in the original input. this is the case, because the *output depth* is a parameter of the convolutional layer, which means it can be arbitrary and does not depend anymore on the input data. Therefore the output depth does not represent the specific colour of the RGB channels anymore, it is rather considered as a *filter*. These filters encode certain aspects of the input tensor, on a low level it could encode edges or certain colour patterns, on a high level a

filter could encode a birds head or tail. In Figure 5 only one filter gets computed, but usually the amount of filters is much higher. If the example had 32 filters instead of one, the convolution operation would result in a tensor with 32 depth channels, each containing a 28×28 grid. Each single grids is representing the response of that single filter to the input. The convolution operation is therefore defined by two hyperparameters[2]:

Nr of Filters Determines the depth of the feature output map. A filter itself can be thought of as a concatenation of multiple convolutional kernels, whereas each kernel is assigned to one channel of the input. The kernels are simply matrices of weights, which are multiplied with the input for feature extraction (see Figure 6 and 7). Varied values in the kernel lead to different filters, which extract different features; e.g. one filter detects edges, another sharpens the image.

Kernel Size Describes the size of the window, which gets extracted from the input in each step; it therefore determines the size of the filter. Common choices for size are 3×3 or 5×5.

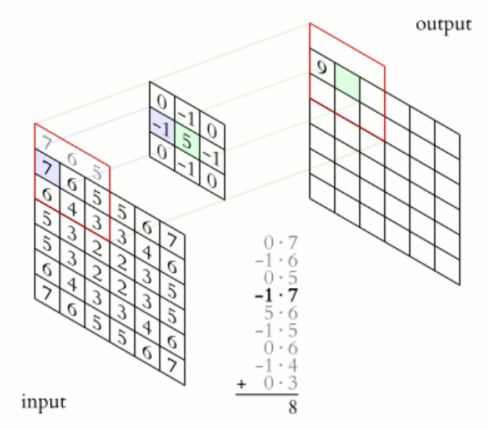

Figure 6: A convoultional kernel extracting feature values

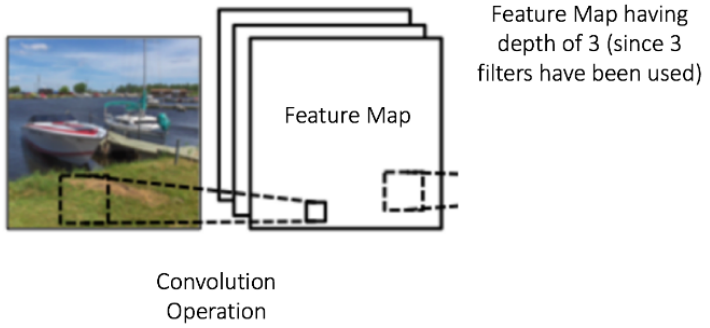

Figure 7: Input image and output feature map

Beside the those two key hyperparameters, there a two further parameters. We just briefly introduce them, as neither of them will be playing an active role in our models for this paper:

Padding If we want to obtain a feature output map with the *same* spatial dimensions as the input, padding can be activated. This is especially useful, if the information from the edges of the image are important, because padding simply creates an additional border around all edges.

Stride is a way of downsampling the input by leaving out rows and columns in the sliding process.

A more common way for downsampling the input of each convolutional layer is the *max-pooling* operation. Conceptually it is similar to convolution, as it also slides an extracting kernel over

[2]In contrast to *parameters*, hyperparameters can not be derived via training, but have to be set and optimized by the designer of the model.

the input, but it does not transform the extracted values, as the convolution does, it just stores the maximum of the values of the input. It is usually applied with a kernel size of 2×2 and stride of 2, so the input data gets downsampled by half. By reducing the size of the network the numbers of parameters and computations are also reduced, which helps to control overfitting.

Convolutional layers are the building block of any CNN, as they function as a *feature extractor*. To wrap-up the concept of convolutional layers we put it in place within an overall CNN architecture by briefly going through every step of the entire algorithm:

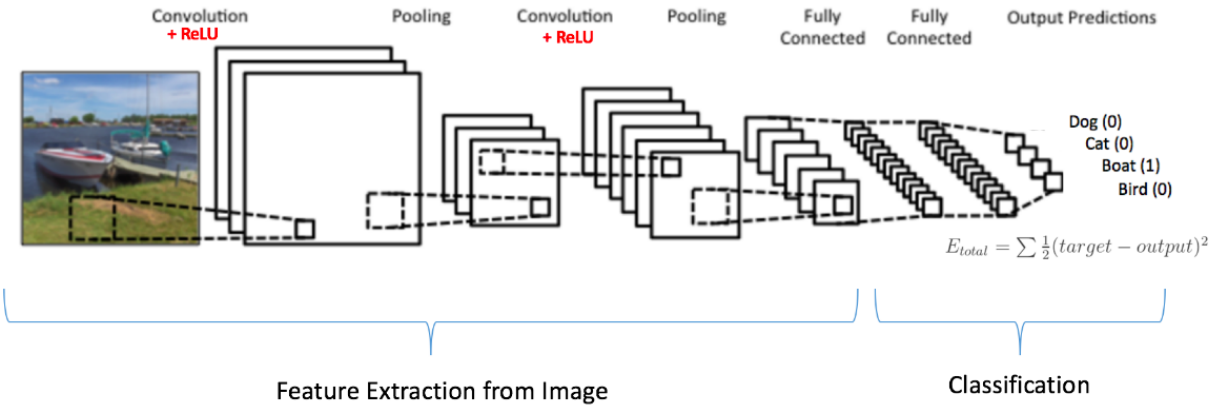

Figure 8: Training process of a CNN

1. **Initializing** all filters and parameters with random values.
2. **Forward propagating:** a training image as input is going through all blocks of the network (*forward propagation step*) to find the output probabilities for each class:
   - Convolution Block: a kernel window slides over the 3D input tensor extracting patches. All extracted patches get transformed by the convolutional kernel into a vector. All these vectors are put together in the 3D output feature map and then transformed by an activation function (usually ReLU) to ensure the non-linearity of the output.
   - Pooling Block: for downsampling the size of the output feature map.
   - Dense Layer Block: takes the output of the last pooling block as input; a dense layer is just a fully connected layer as we know it from a regular ANN, it serves the purpose of doing the actual classification. The output of the convolution block of the network represents high-level features of the training images; the dense layer then gives via Softmax activation a prediction for the class of the image.
3. **Calculating** the total error at the output layer by using a loss function; as we are facing a multiclass problem, we are using the categorical cross entropy loss function:

$$Loss = -\sum_{i=1}^{C} y_i \cdot log\ \hat{y}_i$$

4. **Backpropagating** Hyperparameters, e.g. number of filters, filter sizes, optimizers, etc., have all been set before Step 1 and do not change during training process – only the values of the convolutional kernel and connection weights get updated. Therefore *backpropagation* is used to calculate the gradients of the error and then update all filter and parameter values to minimize the output error by using *gradient descent*. The CNN then learns to classify a particular image correctly by adjusting its filters, so the output error gets reduced.

5. **Repeating** $1 - 4$ for all images in the train set.

3 Data Preparation

To train our CNN models we will use a data set from kaggle, which contains images of 210 different bird species.[3] The data set consists of four directories: a training directory with 28792 images, a validation directory with 1050 images, a test directory with 1050 images and a directory called "consolidated", which contains all images bundled for users who want to creat their own split. Each of those directories contains 210 sub directories, one for each bird species. All images are $224 \times 224 \times 3$ color images in `.jpg` format. Before resizing, all images were cropped so that the bird occupies at minimum 50% of the area in the image. The cropping helps to build a classifier focused on identifying a birds species.

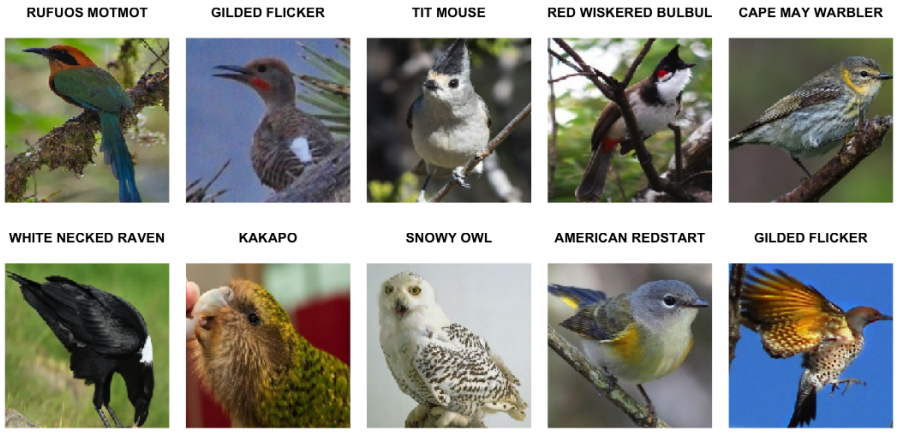

Figure 9: A few birds of the data set

In Figure 10 we can see, that the training data set is slightly imbalanced and sample size per class is relatively low (~100 to ~300 images). This should not be a problem as CNNs can be optimized to learn from small sample sizes and the ratio imbalance is not too high either (Buda et al., 2018). More problematic ist the size of the test and validation set: both only contain 5 images per class. We will reflect upon this fact when it comes to the evaluation of our models in the results Section; we will see how each type of model handles it. To carefully

[3]As the data set is frequently updated by the author, it might contain more species by now. It can be obtained from https://www.kaggle.com/gpiosenka/100-bird-species.

consider the size of those sets is especially necessary, as we are not using all classes for the training process of most of our models.

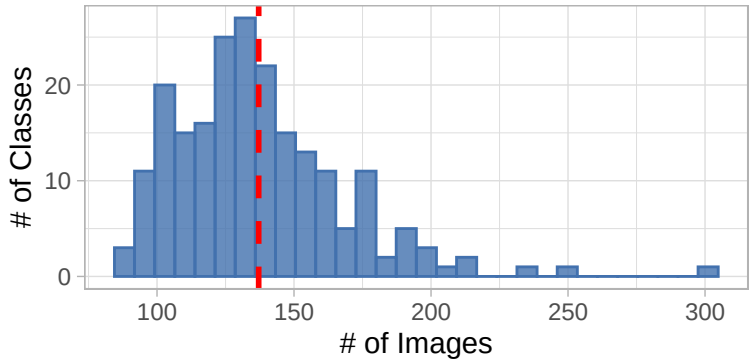

Figure 10: Distribution of Nr of Images per Class; dashed line shows the mean

The reason for the training process of each model being run only on a subset of the available classes is that of computational constraint, as already announced in the introduction. Even for a small date set as ours, CNN models become computational expensive very quickly, especially when it comes to models which include data augmentation. Therefore our modeling process is oriented on a data set size, which does not require a GPU. On a "normal" computer[4] with a normal sized CPU the chosen sample size permits a calculation of each model in a moderate time. Therefore we create two samples: "*tiny*" contains 5 randomly sampled classes and "*small*" contains 25 classes. Both samples result in new subdirectories.

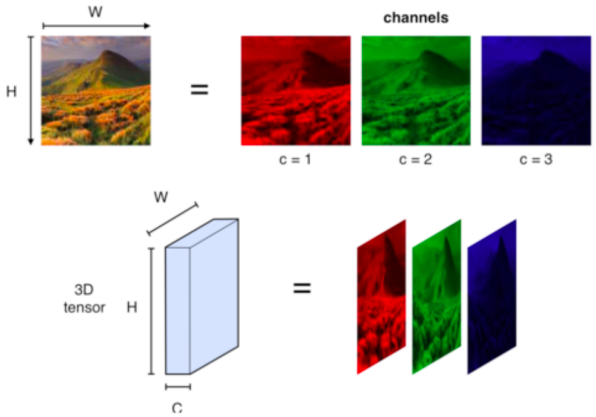

Figure 11: Transformation of an image into a Tensor

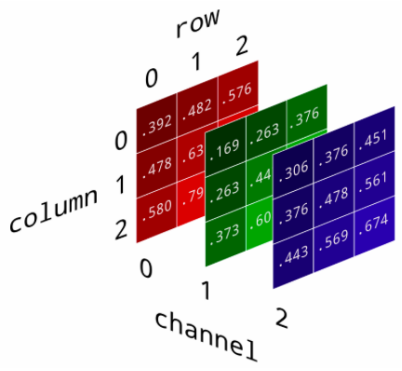

Figure 12: RGB Image Tensor

The last step of data preparation is the transformation of the images from `.jpg` format to a format which is readible for a neural network. As we have seen earlier, the input of a CNN has to be a tensor. Furthermore input data for neural networks in general should be scaled, as large input values can trigger large gradient updates that will prevent the network from converging. The process for creating a data set which can be processed by a CNN starts at reading the pictures files; the `.jpg` content then has to be decoded in a RGB grid of pixels,

[4]For instance a MacBook Air, which certainly is not a powerhouse.

which then have to be transformed in floating-point tensors. In a last step those pixel values between $(1, 255)$ are rescaled to the $[0, 1]$ interval. For each model we create a train, test and validate set through a generator function which automatically turns images from a directory into batches of pre-processed tensors. The generator then yields batches of RGB images and binary labels. Batch and image size can be determined as needed; it yields these batches indefinitely by looping endlessly over the images in the target directory.

We can then specifiy the generator parameters for every data set to obtain a customed data set:

- **Baseline Model** (5 classes): We downsize the image to a size of $150 \times 150 \times 3$ to further ease the computation. Batch size is set to 32, a standard value.
- **Baseline Model** (25 classes): Takes the same configuration as the 5 class baseline model.
- **Augmented Model**: Same configuration as the baseline model, except that the training data is not only rescaled, but it also is augmented. That means the train images are randomly transformed into similar looking pictures to generate more training data from the available training data (see Figure 13). The goal is to expose the model to more aspects of the data, prevent overfitting and achieve a better generalization.

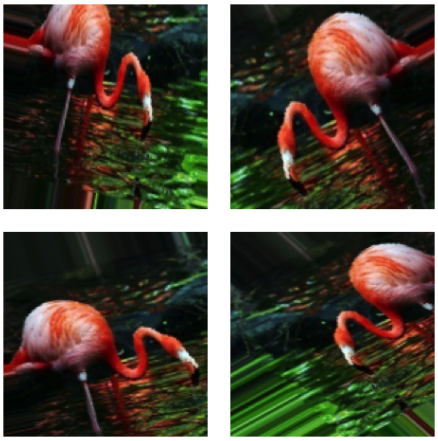

Figure 13: A kernel slides over the entire image to detect local patterns.

- **VGG16 Model**: The image is downsized to $150 \times 150 \times 3$ format. The batch size is only 25 here, because of the architecture of the feature extraction algorithm.[5] The technique of feature extraction uses the weights learned by a pretrained network to extract features (like textures, visual edges and colors) from new data. A new classifier, trained from scratch, is then applied on this features. This means we take the convolutional base (the sum of all convolutional layers of a network) of the VGG16 network, run new

[5]The algorithm used here was originally designed by (Chollet & Allaire, 2018, p.138) for a binary classification problem; it was adapted for our multiclass problem. Somehow the amount of samples from the train set in the model has to be divisible by the batch size without remainder. Therefore it is easier to operate with a batch size of 25 instead of 32.

data on it and train a new classifier on top of the output. We will supply a further description of the technique in the next section, where the entire modeling apporach will be described.

- **MobileNetV2** (25 classes): Since this feature extracting architecture is optimized for an image shape of $224 \times 224 \times 3$ and the computational cost of this model is quite low in general, we stick to this input size. Batch size for the generator is 32 again. We again create the three data set generators, as the entire feature extracting process is controlled by the `tensorflow` API, which we use for this model.
- **MobileNetV2** (210 classes): Takes the same configuration as the 25 class MV2 model.

4 Model Creation

To create a deep learning image classification architecture which could be used on regular (smartphone) images, it probably would be necessary to apply an ensemble learning approach, using one network for object detection (is there a bird in the image?) and another for classification (which species is the bird from?). We only focus on the second task in this paper. In a potential application a classifier would be need, which takes a single image as an input and predicts for the user, which species the bird is from. In order to obtain such a classifier, the underlying algorithm has to be trained on all possible bird species. In our model we are restricted to 210 different Northern American birds, but every modeling approch shown here should be generalizable. The four modeling approaches discussed here can be divided in two kinds: baseline models and pretrained models. In this Section we describe the specifications of each model. All models have been implemented using the `Keras` module in `R`.

Table 1: Hyperparameter Specifications

model name	classes	# epochs	batch size	image shape	optimizer	learn. rate
Base CNN	5	50	32	150, 150, 3	RMS prop	0.0001
Base CNN	25	50	32	150, 150, 3	RMS prop	0.0001
Augmented CNN	5	50	32	150, 150, 3	RMS prop	0.0001
VGG CNN	25	50	25	150, 150, 3	RMS prop	0.0001
MV2 CNN	25	25	32	224, 224, 3	Adam	0.0001
MV2 CNN	210	10	32	224, 224, 3	Adam	0.0001

Table 1 shows our selection of hyperparameters for all models. We chose the same parameters for all models. Any exception from the general choice is due to the specific architecture of the pretrained models (e.g. batch size for VGG16 or image shape for MobileNetV2). Our metric of choice for comparing the performance of each model is Accuracy, which is simply defined as the percentage of the correctly classified positive and negative examples:

$$ACC = \frac{TP + TN}{TP + FP + TN + FN}$$

It assesses the overall effectiveness of each model. But due to the structure of our data set we should be careful with the interpretation of Accuracy values.

4.1 Baseline Models

Our baseline models follow the architecture sketched in Section II of convolution and dense layers. We will evaluate two different kind of baseline model, a plain model and one with data augmentation and regularization:

- **Baseline Model** This model only employs the most basic features of a CNN: 3 convolutional layers connected to a dense layer for classification. Since the training process is computationally expensive, even with a base model, we applied downsampling to the data set. A problem of the data set is the low amount of validation and test images. Therefore we test the base model on two different downsample sizes, to check, whether enlarging the amount of classes has an impact on the base model or not.
- **Augmented Model** The augmented model has the same layout as the baseline model, but it uses augmented training data and drop-out regularization is applied to it. The goal of this extension is to reduce overfitting.

4.2 Pretrained Models

As our data set is relatively small, it is an effective approach to use a pretrained network. That means we are re-using the model weights from previously trained models that were developed for computer vision benchmark datasets, like the ImageNet data set. The great advantage of this approach is that

> [i]f this original dataset is large enough and general enough, then the spatial-feature hierarchy learned by the pretrained network can effectively act as a generic model of the visual world, and hence its features can prove useful for many different computer-vision problems, even though these new problems may involve completely different classes than those of the original task. (Chollet & Allaire, 2018, p.133)

Using a pretrained model is possible in two ways: feature extraction and fine-tuning. For our comparison we will only focus on the first way. We use two different pretrained models, VGG16 and MobileNetV2:

- **VGG16** Since this network is trained on the ImageNet set, which contains several bird classes, it is a good choice for our purpose. For our model we adapt an algorithm from (Chollet & Allaire, 2018, p.138). We borrow the convolutional base of the VGG16 network, to run our data through it, and then train our bird classifier on top of the output. To do so we instantiate the VGG16 model. We again generate images with the above mentioned image generator. Then we use the convolutional base of VGG16 to predict features of every image of the batch. Our wrapper funtion then creates a list of features and their corresponding labels. Technically we are transforming our input data set. This transformed data is then fed into a two dense layer block to train

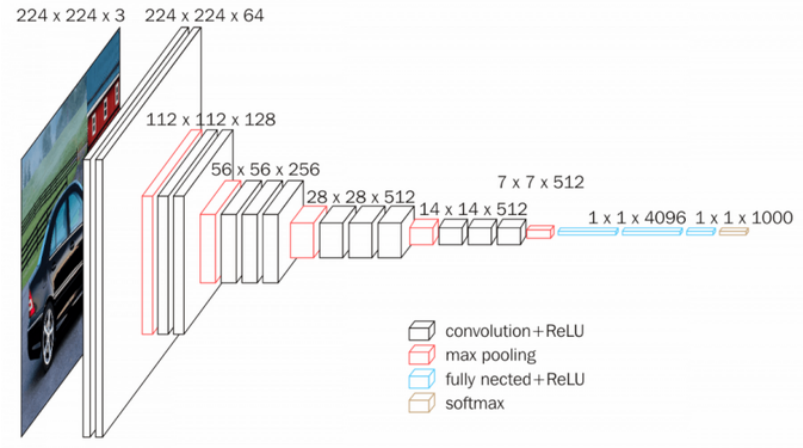

Figure 14: Architecture of VGG16.

the classifier, which should run fast, since only two layers are involved. The feature extraction process is computation intense, therefore we use a sample size of two thirds of the available image data of the small sample.

- **MobileNetV2** We obtain this model via TensorFlow Hub, a library for reusable machine learning models. The procedure for the model is the same as for the VGG16 model, but this time we do not extract the features manually, but let the tensorflow API do the job, simply by adding a feature extraction layer to our model. As this operation is optimized, it is much faster then the VGG16 model; we therefore only train it for 25 and 10 epochs. Since MobileNetV2 requires $224 \times 224 \times 3$ image format, we forgo downsizing the images; its standard version also uses the `Adam` optimizer, which we will use, too.

5 Empirical Results

Table 2: Results of all 6 Models

model name	classes	training acc	validation acc	test accuracy	run time
Base CNN	5	0.981	0.880	0.840	16.254
Base CNN	25	0.999	0.768	0.727	88.209
Augmented CNN	5	0.949	1.000	0.920	89.685
VGG CNN	25	0.998	0.936	0.920	2.146
MV2 CNN	25	0.994	0.990	0.992	47.485
MV2 CNN	210	0.956	0.948	0.958	133.280

Table 2 displays the final results of our study, a comparison of four different CNN approaches resulting in 6 models with alternating input sizes due to the number of classes. All models were only evaluated once on the set of hyperparameters described in Table 1. Tuning of the

hyperparameters did not happen due to computational constraints; as we can see in Table 1 for "run time", half of the models had a training time over one hour.

If we now look closer at the results for each model, we can see the effect of the structure of the data set, it containing only very small test and validation sets. Especially the baseline model illustrates the problem of overfitting very well. All models achieve a good training accuracy of 0.95 or higher, but only the score of the baseline models drops by 10 points or more when it comes to validation and test accuracy. Figure 15 shows the effect of overfitting in the base model: for the 25 classes base model overvitting already starts after a few epochs of training and the validation accuracy stagnates around $\sim 75\%$; for the smaller model we can see the same behaviour with a lower discrepancy in accuracy and a later start of overfitting. We interpret this difference in behaviour as a poor ability to generalize, the model is just memorizing the training data. If we would run the model on the full data, the resulting validation accuracy would probably be rather poor. This is a strong indication of the model not learning general features too well.

If we then look at the augmented model with drop-out regularization, validation and traing curve do not cross. The model does not seem to overfit and it results in a perfect validation accuracy of 1. Problem is, that the test accuracy is 8 points lower then validation and the model is only trained on a 5 class sample. The perfect validation accuracy indicates, that the network memorizes the data set too well and is focussing strongly on the seen birds.

The VGG16 approach with manually created feature extraction still shows slight overfitting, but gap between validation and test accuracy is much smaller then at any of the other models we have looked at so far. As the run time seems incredibly short compared to the other models, it does not take into account the run time for feature extraction; which took about the same time as the run time of the MV2 model with 25 classes. Run on the entire data set the VGG16 approach would probably generalize much better then any of the baseline models.

The MobileNetV2 model with `tensorflow hub` configuration delievers a very satisfying result: a test accuracy of almost 96% on the full data set. As we can see in Figure 15, neither of the MV2 models overfits and Accuracy overall only drops by ~ 4 points from the 25 class model to the 210 model. This shows us, that the MV2 model generalizes well. It would be interesting to investigate individual layers, to see, which kind of features are extracted and to see, if the algorithm learns the abstractions of birds in general.

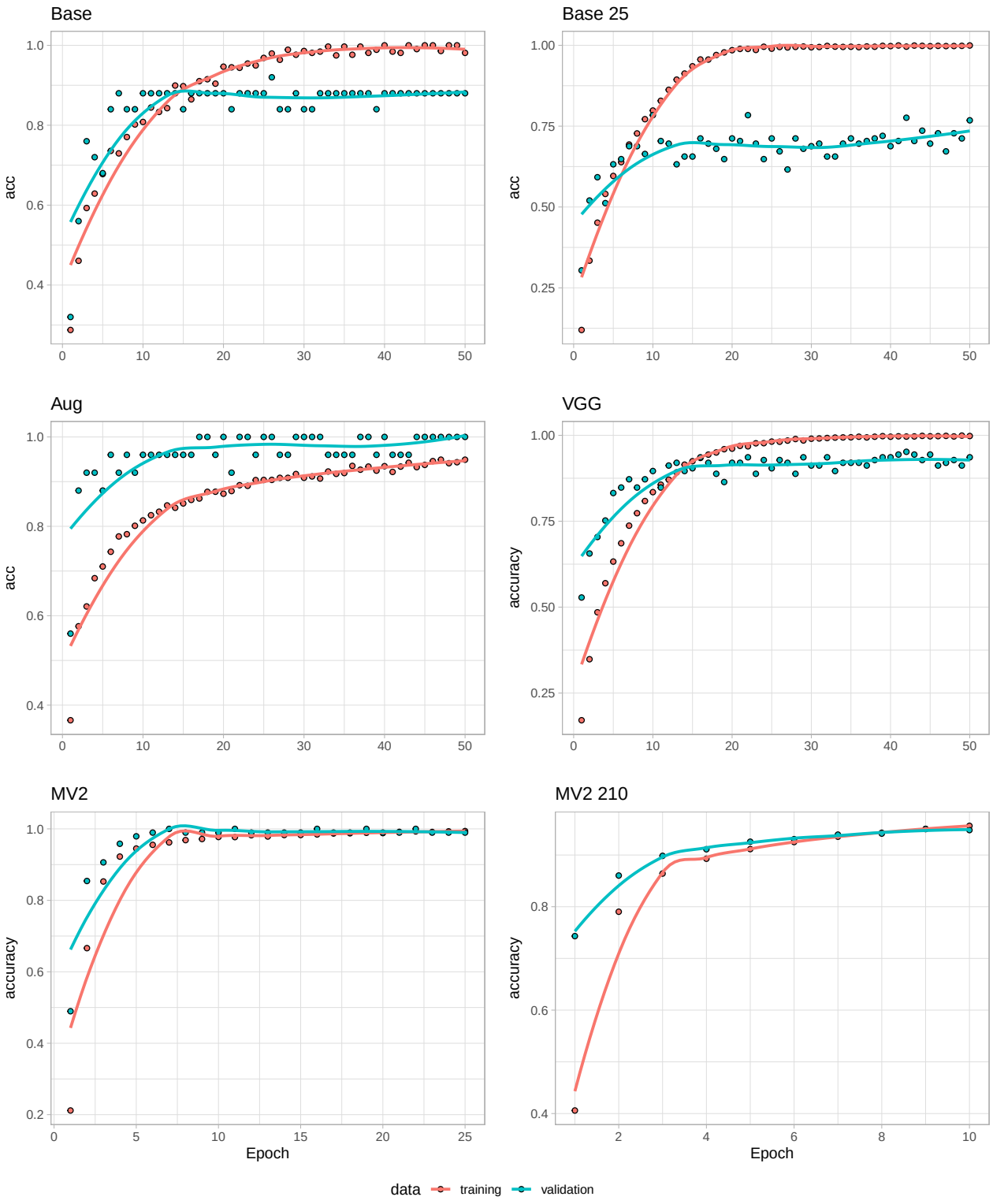

Figure 15: Train and Validation Accuracy

6 Conclusion

The objective of this paper was to compare four different CNN approaches to build a bird classifier from scratch, which could possibly be the core of a potential bird identification application. The result we achieved so far shows us that using a pretrained network and the optimized `tf hub` API would be the the approach in favour, since it reached a high accuracy of almost 96% on the full data set. And these results were obtained without using a GPU device for computation. There is a lot of potential to improve the pretrained model sketched in this paper. If data augmentation would be added to the process, an Accuracy of 98% could probably be reached.[6]

For a potential application a better data base for training the classifier would be needed. A company offering a complete solution is Swarovski, with their binoculars being connected to Cornell Lab of Ornithology and their bird recognition application Merlin Bird ID. Since it has been trained on millions of pictures, it results are very accurate, as long as the input image has a certain quality. The next step of our CNN to improve would be to get access to more data, creat another network for object detection and integrate them in one application.

[6]See the results of other users of the data set at kaggle: https://www.kaggle.com/gpiosenka/100-bird-species/kernels

References

[1] Buda, M., Maki, A., & Mazurowski, M. A. (2018). A systematic study of the class imbalance problem in convolutional neural networks. *Neural Networks*, 106, 249-259.

[2] Chollet, F., & Allaire, J. J. (2018). *Deep Learning mit R und Keras*. MITP-Verlags GmbH & Co. KG.

[3] GERRY (2020). *210 Bird Species*. https://www.kaggle.com/gpiosenka/100-bird-species, Version 26, retrieved 16. July 2020.

[4] Ghatak, A. (2019). *Deep Learning with R* (pp. 1-245). Springer.

[5] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Images

Figure	Source
1	https://www.allaboutbirds.org/guide/Yellow-bellied_Sapsucker/media-browser-overview/65051951
2	https://www.allaboutbirds.org/guide/Hairy_Woodpecker/media-browser-overview/68929201
3	https://www.allaboutbirds.org/guide/Downy_Woodpecker/media-browser-overview/60397941
4	(Chollet & Allaire, 2018, p.115)
5	(Ghatak, 2019, p.172)
6	https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37?gi=35964649b9bd
7	https://www.kdnuggets.com/2016/11/intuitive-explanation-convolutional-neural-networks.html/2
8	https://www.kdnuggets.com/2016/11/intuitive-explanation-convolutional-neural-networks.html/2
9	(GERRY, 2020)
11	https://www.esantus.com/blog/2019/1/31/convolutional-neural-networks-a-quick-guide-for-newbies
12	https://www.kdnuggets.com/2019/12/convert-rgb-image-grayscale.html
13	(GERRY, 2020)
14	https://neurohive.io/en/popular-networks/vgg16/