A Convolutional Neural Network to Classify Satellite Imagery

Nikolas Anic
University of Zurich
nikolas.anic@uzh.ch

Jorge de la Cal Medina
University of Zurich
jorge.delacalmedina@uzh.ch

Jannick Sicher
University of Zurich
jannick.sicher@uzh.ch

1. Introduction

In this project, we construct a custom Convolutional Neural Network (CNN) for satellite image classification. We build upon a model proposed by [Simonyan and Zisserman, 2014] and used by the Stanford Visual Recognition team, which stands out for its accuracy in image recognition settings and came to be known as VGGNet.

A slightly adjusted version of the VGGNet architecture will serve us as a baseline model. We optimize it with respect to the task at stake, changing some features of its architecture and tuning specific hyperparameters. This process results in a customized architecture.

We further experiment with different data inputs: the raw data provided by the EuroSAT dataset, the same data but with balanced labels, and the balanced data with a false-color NGR coding instead of standard RGB.

For proper comparison, we take two deep models as benchmarks. We implement transfer learning on the DenseNet201 and the VGG-16 architectures.

Subsequently, we present our results and discuss them shortly. Finally, we give some concluding remarks on how to adapt CNN for satellite image classification.

2. Dataset

For our analysis, we will use the EuroSAT dataset (see GitHub Repository) presented by [Helber et al., 2019]. The dataset consists of Sentinel-2 satellite images covering 13 spectral bands and incorporates 27,000 geo-referenced images, each labelled as one of ten different classes. The satellite images depict metropolitan areas of cities in 34 different European countries covered in the European Urban Atlas, which was used to conduct the labelling.

The classes describe the type or usage of the depicted terrain. They distinguish between industrial area, residential area, annual crop fields, permanent crop fields, river, sea or lake, herbaceous vegetation, highway, pasture and forest. An example of an image of the class *PermanentCrop* is shown in Figure 1.

The images have been checked and corrected, and images with bad image patches due to snow or ice have been sorted out. Hence, we can assume that the dataset is of high quality. In fact, it is considered to be the standard benchmark for the land-cover classification task.

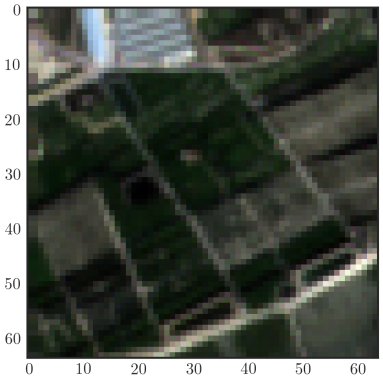

Figure 1: Example image of EuroSat dataset

The authors provide two versions of the dataset, one readily available version using RGB color-coding, and one multi-spectral version. We consider this a great advantage since it allows to experiment with different color-codings. Specifically, we make use of this and experiment with false-color coding (see Section 3.3.12).

One issue to notice with the dataset is that it is unbalanced. There is not an equal number of observations in each class. As can be seen in Figure 2 the classes *Highway*, *Industrial*, *Pasture*, *Permanent Crop* and *River* are minority classes. We address this issue by *upsampling* the minority classes, described in Section 3.3.11.

3. Constructing the Models

To create our Custom model, we first determine a Baseline model. Starting from it, we temper with the network's depth, the number of hidden neurons per layer, and hyperparameters; such as learning rate, filter and kernel size. Moreover, we try different *activation functions* and *initializers*. We take the *overall validation accuracy* to guide our decision upon the parameter choice.

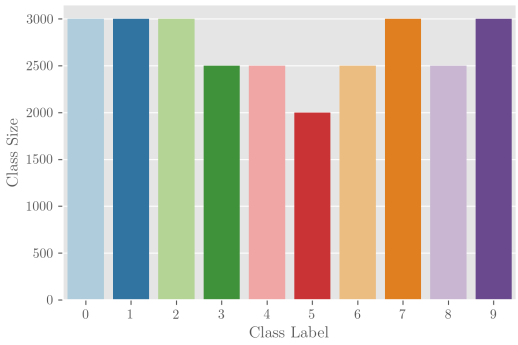

Figure 2: Distribution of Labels

3.1. Data Split

We conduct a train/test split of 80/20. Before the split, we shuffle the data since it comes in a particular order which could cause bias in the results. The split gives us a test set size of 5,400 images. The remaining data is then assigned a training status, on which the weights of our model are trained, and a validation status, which we use for model selection. This partition gives us a training set of 16,600 images and a validation set size of 5,000 images.

3.2. Baseline model

The Baseline model architecture is a slightly modified version of the VGGNet [Simonyan and Zisserman, 2014]. We consider this architecture to be particularly suited for our purpose since it has a simple and understandable structure that allows for extensions.

The Baseline model consists of a repeating pattern of two convolutional layers, followed by a pooling layer, resulting in 16 convolutional layers in total. It uses 64 filters per layer with a small receptive field with a filter size of 3×3. The convolution stride is fixed to one pixel; the spatial padding of the convolutional layer input is such that the spatial resolution is preserved after the convolution, i.e. "same" padding. There are five max-pooling layers for pooling, which follow some convolutional layers (however, not all the convolutional layers are followed by max-pooling). Max-pooling is performed over a 2×2 pooling kernel, with stride 2. Three fully-connected layers follow the stack of convolutional layers and pooling layers. The final layer uses a *softmax* activation, apt for multi-label classification. We use a Rectified Linear Unit (ReLU) activation function and use it in combination with He Normal initialization, as is recommended by [Kumar, 2017].

For the Baseline model, we define the batch size to 50, the number of samples that the CNN must process before updating the internal model parameters. The larger the batch size, the more robust our model will be. However, a larger size comes with a drawback: the larger its size, the higher the computational cost. We put the number of epochs equal to 50 and implement a simple early-stopping feature consisting of 10 epochs. This way, if the accuracy does not improve after ten epochs, the training process will end. This procedure allows us to prevent possible overfitting at an early stage. Since the architecture exhibits many trainable parameters (approximately 15M), its evaluation is rather extensive.

3.3. Custom model

This section describes how we conduct fine-tuning and hyperparameter tuning on the Baseline model experimenting with different parameter values to construct our Custom model. We aim to achieve a higher overall validation accuracy and also better performance in the single class predictions. Furthermore, we are aware of overfitting issues that could compromise the generalizability of the Custom model.

To tune the hyperparameters, we need to define a range of potential parameter values. In theory, this range could cover all possible values that allow successful compiling of the model, which is computationally unfeasible. Instead, it is more reasonable to narrow the range following widely accepted knowledge and established conventions from existing literature.

We will follow the suggestions of the VGGNet authors for the initial ranges of kernel size, filter size, pooling layer and padding. Moreover, we will follow some suggestions by [Géron, 2019] for the number of neurons per hidden layer, batch sizes, number of epochs, activation functions, learning rates. We also implement two suggested approaches aimed against overfitting, namely batch normalization and dropout rates.

Note that we refrain from trying out multiple optimizers, such as Nesterov, SGD or RMSProp, and apply the Adam optimizer directly. It is commonly accepted that Adam outperforms the alternatives on convergence speed and quality [Géron, 2019]. Even so, to ensure some form of variation in the learning rate, we use it with cyclical learning rate [Smith, 2017].

In the following, we will provide an overview of our tuning process. Table 1 summarizes the resulting parameter values. The final product of the tuning process is our Custom model, whose architecture is depicted in Figure 3. We will discuss its results in Section 5.

3.3.1 Data Augmentation

From Section 2, it is evident that our dataset is limited in size. To enhance the dataset, we employ *data augmentation*, which creates slightly modified copies of existing images in the dataset. Another advantage of this method is that it serves as a regularizer [Shorten and Khoshgoftaar, 2019].

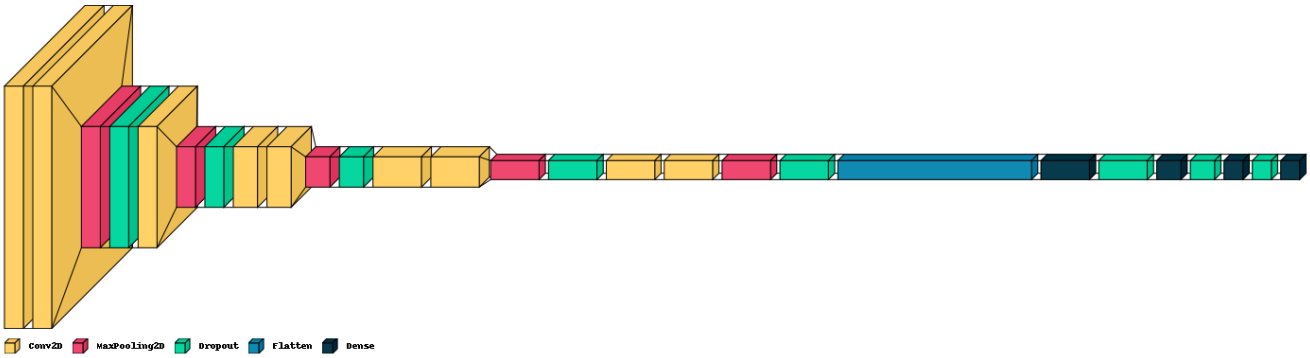

Figure 3: Architecture of the Custom model

Table 1: Results of experimentation

Features	Value
Batch Size	40
Epochs	80
Depth of the Model	Shallow
Neurons per Hidden Layer	16
Learning Rate	0.00007
Activation Function	ReLU
Number of Filters	64
Filter Size	3
Batch Normalization	None
Dropout Regularization	0.1

Specifically, we employ so-called *on the fly* data augmentation, where the model is trained on copies of the original data that are slightly modified in each training cycle (epoch). This way, the model receives "new" data at each epoch but without adding them to the overall image corpus, which limits memory costs [Shorten, 2019].

3.3.2 Batch Size

We define first the batch size and the number of epochs since they are fundamental parameters for the architecture and largely independent of the other hyperparameters.

To define the optimal batch size and the number of epochs, we conduct a grid search for values between 40 and 100 in steps of 10, which results in 7×7 possible combinations. The best combination in terms of validation accuracy is a batch size of 40 and 80 epochs.

3.3.3 Depth of the Model

Networks with deep structure have a higher parameter efficiency than ones with shallow structures. Complex functions can be modelled using exponentially fewer neurons to reach much better performance (on the training set) with the same amount of training data [Géron, 2019]. At the same time, this can lead to overfitting, which results in the model's deferred ability to generalise its performance on outside data, a situation we naturally need to prevent.

To assess whether a deeper network structure is suited, we implement three instead of two successive convolutional layers resulting in 19 convolutional layers in total. We find, however, that a deeper model does not increase the overall accuracy, so we the "shallow" model.

3.3.4 Neurons per Layer

A similar argument about the depth of the model also applies to the number of neurons per layer: More neurons per layer may fit the data more efficiently but may also lead to overfitting [Géron, 2019].

To define the number of neurons per hidden layer, we compare initial neuron sizes of 16 and 32, respectively. We then multiply the potential values by a factor of 2, 4, 8 and 16 to create a triangular shape of the model, as it is common in practice. In our experimentation, we find that with 16, we achieve better overall accuracy.

3.3.5 Learning Rate

The learning rate is arguably the most important hyperparameter for training. It controls the step size of the weights for each iteration to approach a minimum of the loss function [Murphy, 2012]. There is a trade-off in its size: If we set a learning rate too high, it might overshoot a potential minimum and "jump around" without converging. Conversely, a learning rate that is too low can take a long time to converge and may end up in a local minimum [Buduma and Locascio, 2017].

We use an Adam optimizer, which combines two important features: Adaptive learning rates and momentum.

Adam uses estimates of the first and second moments of the gradients to adjust the learning rate accordingly.

Additionally, we use a cyclical learning rate with Adam. Instead of having a learning rate that monotonically decreases during training, it varies the learning rate cyclically within a band of values.

Its benefit lies in increasing the learning rate, which although possibly leading to greater loss in the short run, leads to a better optimum in the longer run. It might, for instance, prevent staying stuck at a saddle point and can also lead to faster convergence [Smith, 2017].

To implement it, we define two parameters; the bounds within which the learning rate can vary and the step size that defines the length and magnitude of the cycles.

As is recommended by [Smith, 2017], to determine the bounds of the cyclic learning rate, we run a grid search for different values and assess the model loss. We note that the performance of the models does not vary too much for values within the range of 0.0001 and 0.001 but declines substantially when for higher or lower values. Hence, we choose bounds of the range as the lower and upper bound of the cyclic learning rate.

[Smith, 2017] suggests using a step size of between 2-10 times the number of iterations divided by the batch size. We put the step size to $\frac{5 \cdot N}{40}$.

Using cyclical learning rate with Adam, we run a grid search for the learning rate parameter of the Adam optimiser for values $\eta \in$ [0.00005, 0.00007, 0.0001, 0.0003, 0.0005, 0.0007, 0.0009, 0.001, 0.01]. A learning rate of 0.0007 leads to the best overall accuracy.

We find that varying this parameter leads to differences in the overall accuracy of around three percentage points. This finding stands somewhat against the claim by [Smith, 2017] that testing different rates is not necessary since it has no considerable influence when using cyclical learning rates. Our results suggest that it is beneficial to experiment.

3.3.6 Activation Function

There are several options to choose from for the activation function. We are testing two activation functions against ReLU, namely Exponential Linear Unit (ELU) and Scaled Exponential Linear Unit (SELU). ELU may counteract the well-known *dead ReLU* problem.[1] Using SELU leads to faster convergence. It is, however, not yet widely used in CNN [Madasu and Rao, 2019].

Using ELU and SELU did not improve our results. We thus keep using ReLU as activation function.

[1] A *dead* ReLU always outputs the same value for any input.

3.3.7 Number of Filters

Each filter is responsible for capturing a distinct pattern. For instance, the first layer of filters is responsible for detecting edges, corners or dots. Subsequent layers combine these patterns to create larger patterns. Intuitively, more filters can recognize more detailed patterns but also leads to higher computation costs. We thus test whether 32 filters per convolutional is sufficient.

Our results, however, suggest 64 leads to higher accuracy, so we decide to keep this number for our Custom model.

3.3.8 Filter size

The filter size determines the receptive field of a neuron. A smaller filter size is more capable of recognizing detailed patterns and local features of the image. However, a small size could be too narrow because information of neighboring pixels is lost that way.

We test how a filter size of 5×5 influences the overall accuracy and observe that it leads to worse results. So, we keep a filter size of 3×3.

3.3.9 Batch Normalization

We experiment with batch normalization to address the problems of exploding/vanishing gradients. This method consists of normalizing the data before the activation function and scaling and shifting the result optimally. Additionally, it acts as a regularizer that counteracts overfitting [Géron, 2019].

Surprisingly, however, using batch normalization gives us a much worse accuracy, so we do not include it in our Custom model. We try other regularization techniques instead.

3.3.10 Dropout Regularization

To address overfitting, we employ the dropout regularization method, proposed by [Srivastava et al., 2014]. At every training step, every neuron has a certain probability of being ignored during the training step. In other words, their contribution to the activation of the downstream neurons during the forward pass is temporally switched off. Weight updates during the backward pass are not applied to these neurons.

We test the following dropout rates for our fine-tuning [0.05, 0.1, 0.2, 0.4, 0.6, 0.8]. From our experimentation, we get the best result with a dropout rate of 0.1.

3.3.11 Balancing Data

As mentioned in Section 2 our data labels are unbalanced. This might compromise the generalizability of the model

since it may be biased away from minority classes. Additionally, evaluation metrics such as overall accuracy cannot account for class imbalances and are thus to be interpreted with care. To balance the dataset, we artificially upscale the minority classes using the *ImageDataGenerator* method.

We note, however, that we already use the same technique to augment data, described in 3.3.1, which could pose an issue. We are unfortunately not aware of any related literature on this matter.

To be precautious, we decide to keep the image augmentation only for some factors when balancing the labels. We use different augmentation characteristics during the training process, such that the same image augmentation is not applied to one image twice.

3.3.12 Using NGR Color-coding

An additional experimental approach we follow is to change the nature of the input in a way that its features are better recognisable. The EuroSat dataset incorporates images in multi-spectral form, ranging over 13 color-bands. This enables us to use different color-coding for the satellite image than the standard RGB.

Remote sensing theory tells us that false-color can effectively detect features that are not readily discernible otherwise. Color-codes with near-infrared are used to detect vegetation in satellite images [Rees, 2012]. Since several classes contain different vegetation types, e.g. forest and pasture, it intuitively seems promising for better classification accuracy.

The multi-spectral image data contains near-infrared (N). We combine it with green (G) and red (R) to get the false-color model NGR. That is, our images are composed of the layers near-infrared, green and red. For illustration, Figure 4 depicts the NGR-coded example image in Figure 1. We will henceforth refer to the Custom model using NGR-images as input simply as NGR Custom model. Its results are discussed in Section 5.3.

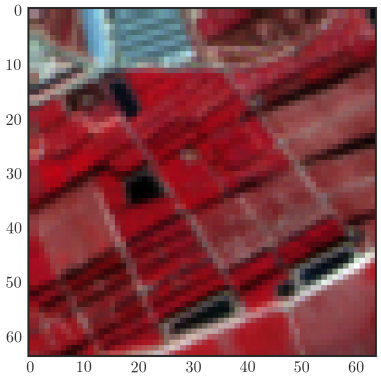

Figure 4: Example image in NGR

4. Transfer Learning Benchmarks

We implement a transfer learning approach with two different, deep architectures to use as benchmarks for comparison with our Custom model.

First, we use the DenseNet201 proposed by [Huang et al., 2017]. This architecture is considerably deeper than our Custom model. It takes into account that architectures reach higher accuracy and are more efficient when having shorter connections between layers located close to the input and output. The DenseNet201 connects every layer to every other layer in a feed-forward manner.

Second, we use the VGG-16 presented by [Simonyan and Zisserman, 2014]. This model also uses depth to achieve better accuracy in large-scale image recognition settings. It has a similar structure to our Custom model. In the ImageNet Challenge in 2014, this model achieved first and second place.

5. Results & Discussion

In this section, we discuss the results of the models described in Sections 3 and 4. We base our discussion primarily on the overall validation accuracy and class-specific precision and recall. We briefly define these three evaluation metrics in the following.

C_{ij} denotes the number of instances where an image of class i has been labelled as class j, where $i, j \in \{1, 2, \ldots, 10\}$.

The overall accuracy gives us the probability that an image will be classified correctly. It is computed by the fraction of overall correctly classified instances, where N is the number of all observations:

$$Accuracy = \frac{\sum_i C_{ii}}{N} \tag{1}$$

Precision is the proportion of instances that were correctly classified as i, out of all instances classified as i:

$$Precision_i = \frac{C_{ii}}{\sum_j C_{ji}} \tag{2}$$

Recall is the proportion of instances that were correctly classified as i, out of all instances which are actual i:

$$Recall_i = \frac{C_{ii}}{\sum_j C_{ij}} \tag{3}$$

Table 2: Performance Metrics on Validation Set of all CNN Architectures (highest values per class in **bold**)

	Precision / Recall					
		Custom			*Transfer Learning*	
Class	Baseline	Unbalanced	Balanced	NGR	DenseNet201	VGG-16
Annual Crop	0.90 / 0.89	0.95 / 0.95	0.95 / 0.97	**0.97** / 0.97	0.96 / **0.98**	0.91 / 0.93
Forest	0.95 / 0.98	0.93 / **1.00**	0.95 / **1.00**	0.96 / **1.00**	**0.98** / 0.99	0.93 / 0.97
Herbaceous Vegetation	0.86 / 0.79	0.92 / 0.94	0.94 / **0.97**	0.95 / 0.93	**0.98** / **0.97**	0.91 / 0.91
Highway	0.85 / 0.81	0.97 / **0.97**	0.98 / **0.97**	**0.99** / 0.96	0.97 / 0.96	0.84 / 0.81
Industrial	0.97 / 0.89	0.97 / 0.98	0.98 / **0.99**	**0.99** / 0.97	**0.99** / 0.97	0.95 / 0.92
Pasture	0.91 / 0.85	0.94 / 0.95	**0.98** / 0.95	0.97 / 0.96	**0.98** / **0.98**	0.90 / 0.87
Permanent Crop	0.76 / 0.85	**0.96** / 0.87	0.95 / 0.92	**0.96** / 0.94	**0.96** / **0.95**	0.90 / 0.85
Residential	0.95 / 0.97	0.98 / 0.99	**0.99** / **1.00**	0.95 / **1.00**	0.97 / **1.00**	0.90 / 0.99
River	0.81 / 0.89	**0.98** / 0.94	**0.98** / 0.97	**0.98** / **0.99**	**0.98** / 0.96	0.86 / 0.85
Sea Lake	0.97 / 0.99	0.99 / 0.97	0.98 / 0.98	0.99 / **1.00**	**1.00** / 0.99	0.97 / 0.97
Overall Accuracy	0.89	0.96	0.97	0.97	**0.98**	0.91

5.1. Baseline model

With the Baseline model, we achieve an overall validation accuracy of 0.89 (see Table 2). Figure 5 depicts the validation accuracy and loss during training. We can diagnose that we have overfitting problems from these learning curves: While the training loss decreases with experience (number of epochs), the validation loss first decreases up to a point and then begins to increase again. At this point, the model is overfitting the train data, and we would do better with a simpler model that is more generalizable.

Figure 6 shows the confusion matrix for the Baseline model.[2] It can be seen that for certain classes it already achieves a high accuracy on the validation set, namely *Forest* (0.98), *Residential* (0.97) and *SeaLake* (0.99).

On the other hand, *Herbaceous Vegetation* has the lowest classification accuracy on the validation set (0.79). This is attributable to the fact that in 13% of the cases, it was falsely classified as *Permanent Crop*. Similarly, *Highway* has an accuracy of 0.81 and was mislabelled as *River* in 8% of the cases. We take this as an indication that the Baseline model cannot sufficiently distinguish between similar land-cover classes.

This was to be expected since the model is not tuned to this particular land-cover classification task. In conclusion, the Baseline model does a fine job distinguishing high-level features between classes but is less capable at the low level, compromising its classification ability for similar classes.

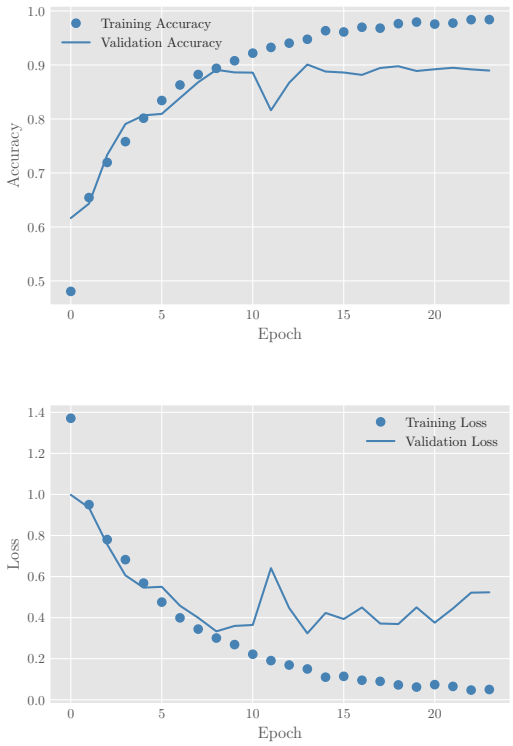

Figure 5: Learning curves of Baseline model

[2]Note that in all the confusion matrices shown, the Y-axis shows the actual and the X-axis the predicted labels.

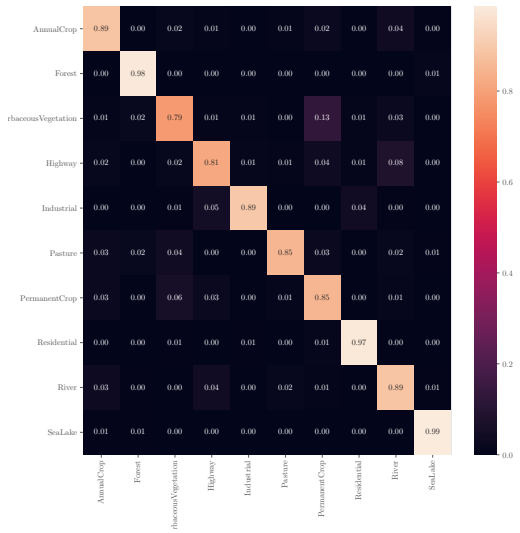

Figure 6: Confusion matrix of Baseline model

5.2. Custom model

The evaluation of the Custom model on the unbalanced data shows a great improvement in the overall validation accuracy from the Baseline model. It reaches a value of 0.96 (see Table 2). However, the precision of some classes worsens relative to the Baseline model. We notice that using unbalanced data generally leads to a lower precision in the minority classes.

The Custom model applied to the balanced data achieves even better results with an overall accuracy of 0.96. It further has higher precision and recall than the Baseline model in all classes. Interestingly, the model achieves higher precision on those classes which had relatively low precision applying it to the unbalanced data. In comparison, it achieves lower precision in *SeaLake* which had the highest precision of 0.99 in the unbalanced data. From the Table 2 and Figure 8 we can see that we can marginally improve the performance in most classes again. The most considerable improvement in precision can be achieved with the class *Pasture*. However, it is crucial to notice that the classes *Sea Lake*, *Permanent Crop* achieve a slightly lower precision than with unbalanced data. Further, the Custom model using balanced data can distinguish much better between *HerbaceousVegetation* and *PermanentCrop*, as well as between *River* and *Highway* compared to the Baseline model.

Comparing the learning curves for the Custom model with the ones of the Baseline model shows much better progress in terms of validation accuracy. We have less of an upward trend which indicates that we managed to contain overfitting (see Figures 5 and 7).

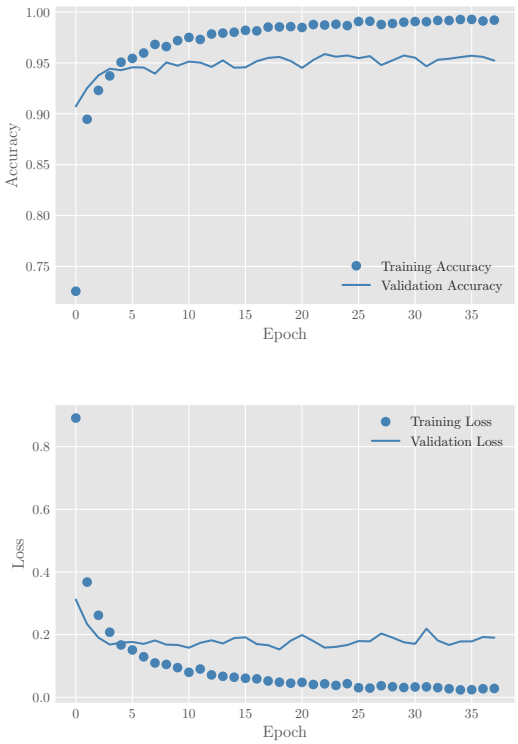

Figure 7: Learning curves of Custom model

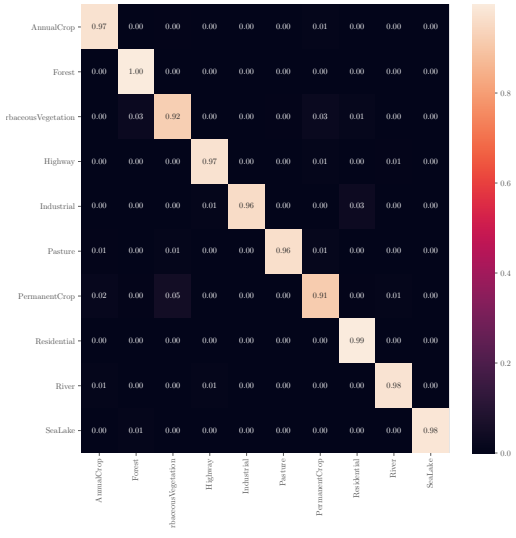

Figure 8: Confusion matrix of Custom model

5.3. NGR Custom model

The results of the NGR Custom model as input show the same overall accuracy of 0.97 as using the standard Custom model (see Table 2).

A peculiarity that we observe in the training of the NGR Custom model is the fact that the learning curves, as shown in Figure 9 differ substantially from those of the Custom model. The learning curves show a somewhat erratic validation accuracy and loss, but the discrepancy between training and validation accuracy seems to be considerably narrower than previously. Consequently, we infer that overfitting in the NGR model is reduced compared to the baseline and Custom model.

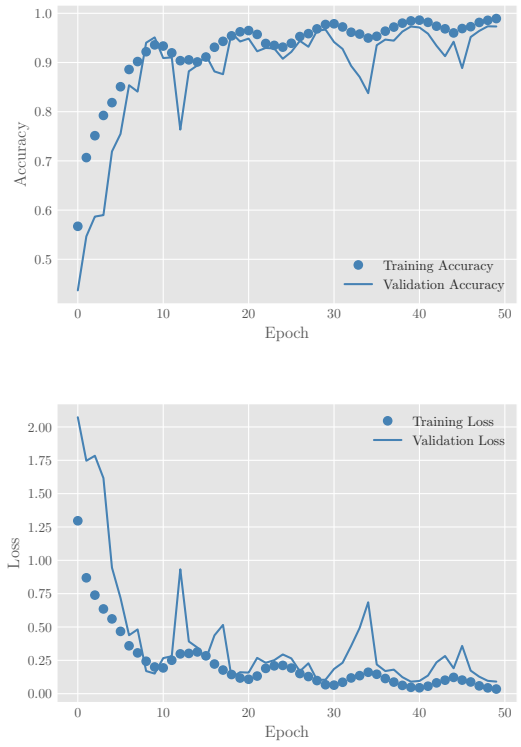

Figure 9: Learning curves of Custom model with NGR

As discussed in Section 3.3.12, the purpose of using NGR-images is to discern certain features better. Effectively, the NGR Custom model achieves the highest precision across all models in *AnnualCrop, Highway, Industrial, PermanentCrop* and *River* (see Table 2).

The confusion matrix in Figure 10 shows that in most classes, the NGR Custom model slightly improved class-specific accuracy and misclassification.

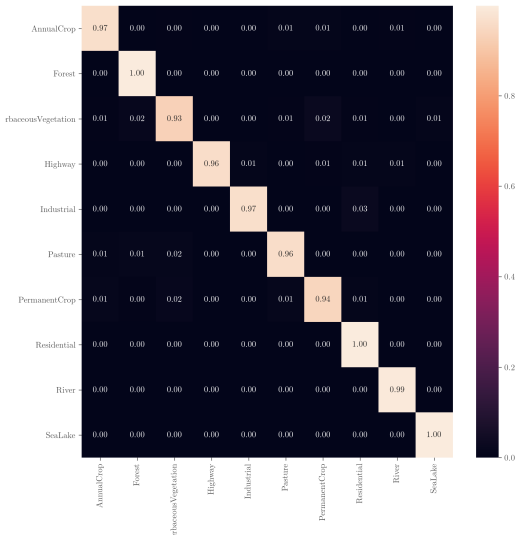

Figure 10: Confusion matrix of custom NGR model

5.4. DenseNet201 and VGG-16

Considering the results of our benchmark models from the transfer learning in Table 2, we find that the DenseNet201 is the best model with an overall accuracy of 0.98. This is, however, hardly surprising since it has a deeper network structure than our Custom model. Nonetheless, the increase in accuracy seems to be only marginally improved.

Concerning the precision and recall values of DenseNet201, it has the highest performance across all models for most classes. Although our Custom model does not perform much worse in this respect either.

More surprisingly, the results of the VGG-16 show that it performs significantly worse than the Custom model. Despite the similarities, it has a lower overall accuracy of 0.91 and achieves worse precision and recall in all classes.

We take these results to demonstrate that it is not only a deep network architecture or special features of the architecture that determine the final performance of a model. Fine-tuning and experimentation by changing inputs and adjusting hyperparameters are essential for good model performance.

6. Conclusion

We investigate the classification of satellite images using the EuroSAT dataset and develop a custom model, experimenting with hyperparameters and architecture structure. Subsequently, we compare it to benchmark models generated using a transfer learning approach.

During experimentation, we focus especially on the learning rate, a key component for a well-performing model. For this, we implement an Adam optimizer with a

cyclical learning rate proposed by [Smith, 2017]. Additionally, we apply a label balancing approach using data augmentation.

Although we could manage to contain overfitting to some extent by changing the architecture, we find that our model still suffers from overfitting. This could compromise its generalizability. Further fine-tuning could be effective to counteract this drawback.

Finally, we investigate how NGR color-coding affects precision and recall of the individual classes compared to conventional RGB color-coding. We conclude that this helps to discern more detailed patterns in some cases. We also observe that the learning curves in this setting show more erratic behavior than using RGB coding.

Comparing our results with the benchmark models with deep architectures, we find that we can produce good model accuracy with our Custom model. This highlights that not only an increased depth of the CNN is relevant for optimal performance, but that experimentation with structural features, hyperparameters and different input formats is the more promising approach to achieve better performing models.

References

[Buduma and Locascio, 2017] Buduma, N. and Locascio, N. (2017). *Fundamentals of deep learning: Designing next-generation machine intelligence algorithms*. " O'Reilly Media, Inc.".

[Géron, 2019] Géron, A. (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc., 2nd edition.

[Helber et al., 2019] Helber, P., Bischke, B., Dengel, A., and Borth, D. (2019). Eurosat: A novel dataset and deep learning benchmark for land use and land cover classification. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*.

[Huang et al., 2017] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708.

[Kumar, 2017] Kumar, S. K. (2017). On weight initialization in deep neural networks.

[Madasu and Rao, 2019] Madasu, A. and Rao, V. A. (2019). Effectiveness of self normalizing neural networks for text classification. *arXiv preprint arXiv:1905.01338*.

[Murphy, 2012] Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

[Rees, 2012] Rees, W. (2012). Physical principles of remote sensing.

[Shorten and Khoshgoftaar, 2019] Shorten, C. and Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(1):1–48.

[Shorten, 2019] Shorten, Connor; Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. *Journal of Big Data*, 6(60).

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[Smith, 2017] Smith, L. N. (2017). Cyclical learning rates for training neural networks.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.