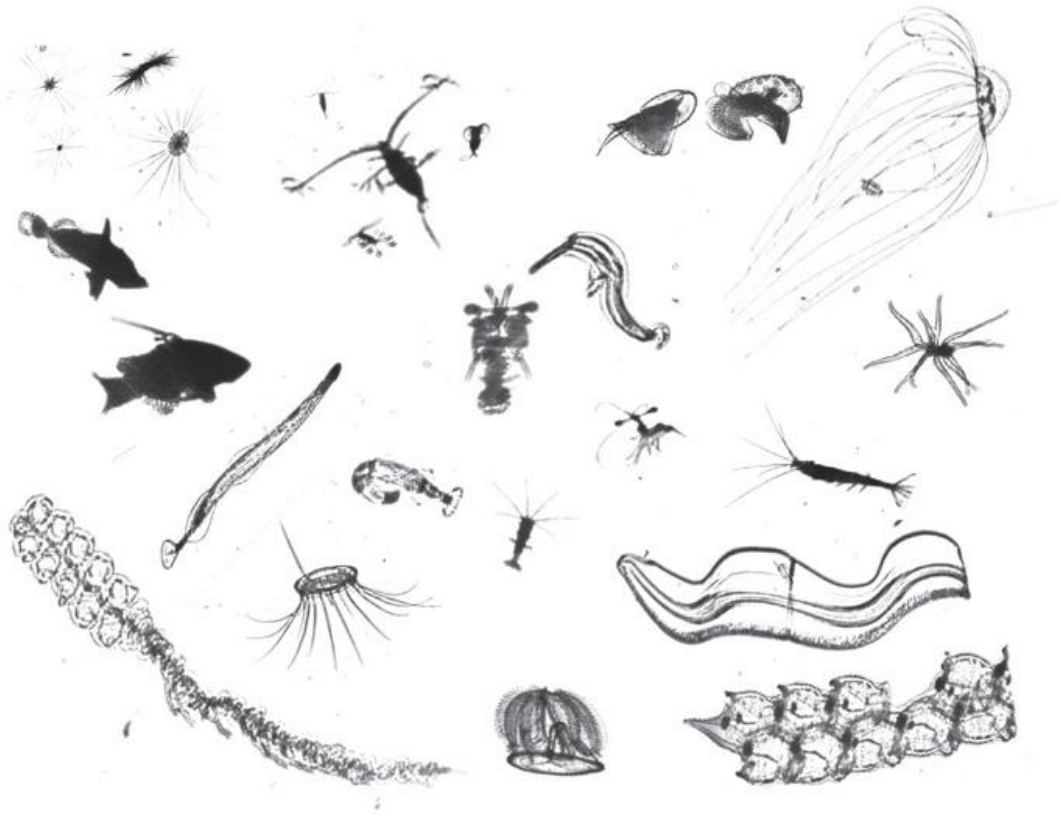
Machine Learning Project Report 2022



Team DreamTeam

Bram Fresen

Bram Huis

Max Burger

Moos Middelkoop

Chapter 1.1: Introduction

In this given model and its subsequent versions, we are addressing and optimizing a classification problem of plankton, as was described in the National Data Science Bowl 2015. In the presented problem, a model is required to classify a new picture of plankton based on a prepared training set to the best of its ability. As plankton is considered to be an indicator of a given oceanic ecosystem, a model that accurately predicts the plankton quantity and class(es) in a sample can help researchers in a faster manner to determine certain diagnostics about a given water sample and its ecosystem.

This specific classification task consists out of 121 possible classes of plankton, of which 3 classes are dedicated to unknown or undiscovered plankton classes. For the classification task, we utilized a Convolution Neural Network (CNN) as is highlighted in our model section. The model was trained with roughly 30.000 JPEG-images of plankton, which vary in class and image-size. In the base version of our model, we applied some basic alterations to our input, as is highlighted in our data-analysis chapter, and utilized a very general model to tackle the classification problem. The base model classifies the plankton to the correct class with an accuracy of 47,85%, compared to 0,83% if the model chose at random. Further alterations to the analysis are discussed and presented in order of model-version.

Chapter 1.2: Data Analysis and Pre-processing

1.2.1: Image resizing

During the pre-processing phase we had to tackle a series of problems to make our data useable. Because the Plankton dataset contains images of different sizes, resizing the images was imperative. We had a couple of choices: crop larger images, stretch the smaller images, or compress the images. Because the images are not all centred, cropping the images would involve a great deal of manual labour as all images would have to be processed individually. Therefore, we chose to compress or stretch the images. We considered a couple of variations of this method: stretching the images close to the largest values for each dimension, compressing the images to close to the smallest values for each dimension, or taking the average of all dimensions and using these values. In the end, we chose to compress the images close to the smallest values for each dimension. This decision was based on the fact that, in the first version of our model, increasing the image size dramatically increased training time and seemed to add a lot of noise to the data as the accuracies oscillated a great deal. Thus, it seemed evident that compressing the images was the best approach for our purposes.

1.2.2: Extra channels

Each of our input images consists of three channels. However, when trying to display the data, assuming the channels represented an RBG colour map, all the images were displayed in black and white. After writing some code to analyse all three channels, we found out that all channels had the exact same values, turning all images grayscale. This essentially means that two of the three channels for all the data are redundant, as they all carry exactly the same data. In response to this discovery, we chose to drop two of the channels for each image, using only one channel to train the model.

Chapter 1.3: Base model

For the analysis of the given problem, we utilize a convolutional neural network (CNN) for the classification task at hand. The model itself should be able to handle inputs which are all of the same size, an array of size (28, 28, 3), as is discussed in the pre-processing chapter. Following, the inputs are being processed through a CNN which has a kernel size of (3, 3), 128 hidden nodes, ReLU activation functions, and 121 output notes, as we aim to classify between the 121 different classes. Moreover, the model utilizes a SoftMax function to determine the prediction in the final layer and builds up to 20 epochs. In order to train the given model, we utilize 70% of the given training set for the training data, whereas the remaining 30% is utilized for the validation of the predictions and the improvement of the given model. No augmentations were applied on the training data prior to training, and no other final alterations were made for the output in the base-version of our model. The resulting accuracies are discussed in the conclusion section, and the model summary is highlighted in figure 1.2 below.

Layers	**Activation function**	**Output shape**	Parameters	**Further data**	
Convolutional 2D	reLU	(28, 28, 32)	320	Total parameters	852985
Max pooling 2D	-	(14, 14, 32)	0	Total learnable parameters	852985
Convolutional 2D	reLU	(14, 14, 64)	18496	Non-trainable parameters	0
Max pooling 2D	-	(7, 7, 64)	0	Image input shape:	28, 28, 3
Flatten	-	(3136, 1)	0	Convolutional layer kernel size:	3, 3
Dense	reLU	(256, 1)	803072	Max pooling kernel size	2, 2
Dense	softmax	(121, 1)	31097		

Figure 1.2: Model summary of the first model

Chapter 1.4: Conclusion

The first model had a training accuracy of 74,72% and a validation accuracy of 47,85% after 20 epochs (figure 1). In the left image, the validation loss is the lowest at around 5 epochs, after that, the validation costs start rising. In the right image, the validation training accuracy and validation accuracy are getting farther away from each other. This shows that the model is overfitting.

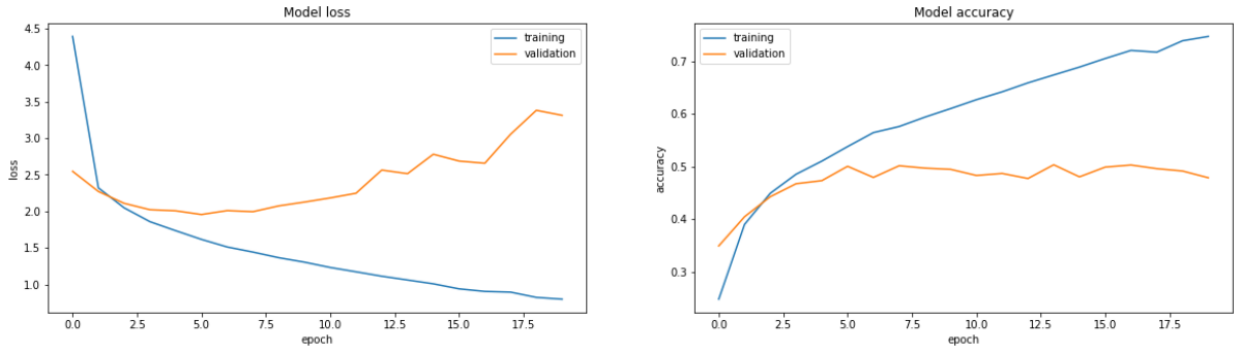


Figure 1.2: Training vs validation loss (left side) and training vs validation accuracy (right side).

In the next version we could try to fix the overfitting. For instance, by using dropout layers or batch normalization. For milestone 2, we could also look into other image shapes. The data is not equally distributed, some classes have more images than others. As a result, the model trains more on certain classes than others, meaning the model can overfit more easily. A final thing we could do, next milestone, is generating a confusion matrix to see how some classes might influence the predictions. This is particularly useful to solve the class imbalance. The given model (version 1.0) seems to be overfitting quite easily as the given model lacks the precaution measures and additional alterations which should prevent overfitting in the dense neural network.

Chapter 2.0: Cleaning-up code, Confusion Matrix, and Summary

2.0.1: Deleting extra channels

In the first version of the model, we figured out that all images had three channels that contained exactly the same data. In the previous version, we deleted these channels for all images after loading the data. In this version, we chose to do so directly when loading the data and also removing the code we used to make the discovery.

2.0.2: Confusion matrix

To give more insight into the distribution between the predictions and actual classes labels, we computed several confusion matrices to highlight any potential faultiness in our model and see how the class labels are distributed. We first utilized a confusion matrix that was based on the absolute number of classifications, as can be seen on the left in figure 2.1, whereas the right computes the confusion matrix as a percentage of the total classification of the given predicted class.

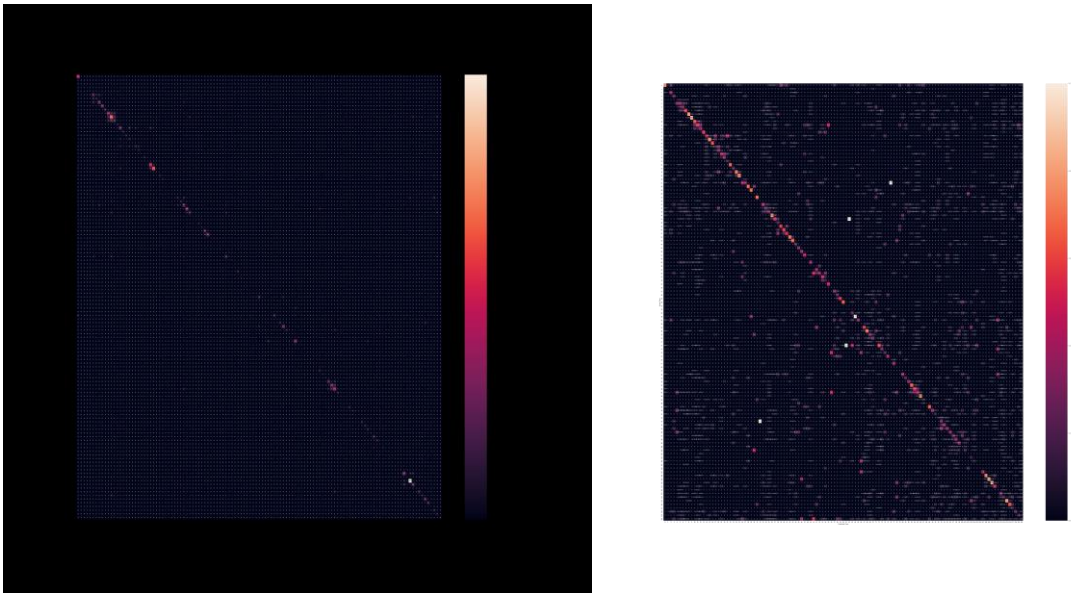


Figure 2.1: Confusion matrix based on the absolute (left) and relative (right) number of classifications.

We see in the relative confusion matrix, which visually presents the data clearer compared to the absolute one, that the overall trend is diagonal across the plot, indicating correct classifications. We do see some areas in which this trend either fades away, indicated by black areas, or some high degree of misclassification, indicated by light spots which are not aligned with the diagonal trend. Further investigation into the 4 spots with the highest degree of misclassification far off the diagonal reveal that these are classes that have a relatively low number of training- and validation data.

As we are only interested in the misclassifications in absolute numbers, as these indicate where we can improve the most, we again compute the confusion matrix without the inclusion of the diagonal, as the exclusion amplifies the colour-signalling of the misclassification in the matrix. Moreover, we

include the different classes of plankton on both axes, to allow for a more insightful overview of the misclassifications. The extended matrix can be seen in figure 2.2.

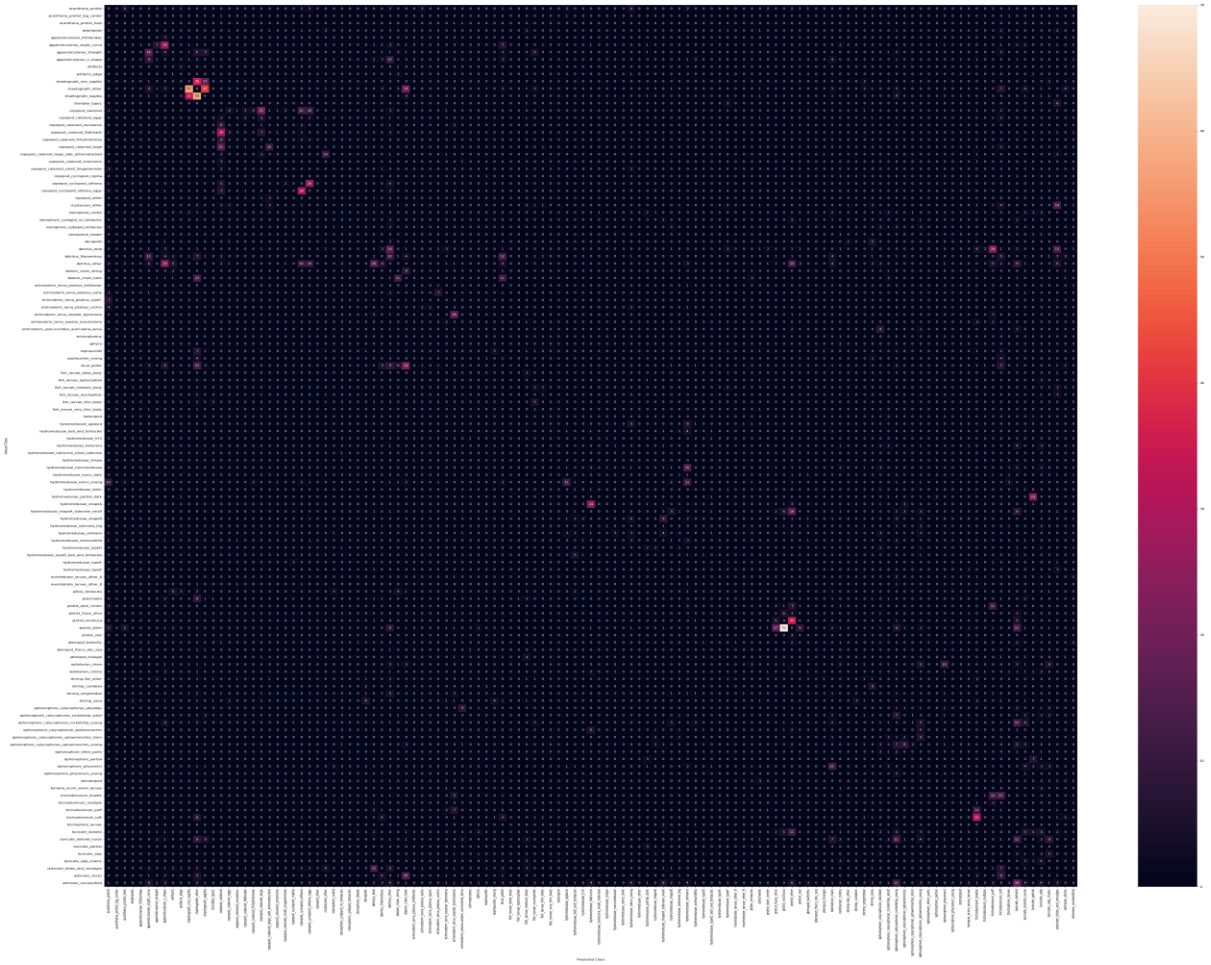


Figure 2.2: Confusion matrix with an excluded diagonal based on the absolute number of classifications.

While the given figure might be overextended due to the number of possible classes, and therefore hard to read without zooming, one can observe that there are several large misclassification clusters, namely that of the Chaetognath, Protist, and Trichodesmium classes. The highest rate of misclassification occurs within these classes, entailing that our model recognizes that a given picture belongs to a given class, yet fails to classify it to the correct subclass, for example to either the Chaetognath Sagitta or Chaetognath Non-Sagitta class. This tells us that we can achieve higher accuracy and improve our model significantly by including features into our model that allow for better differentiation between such subclasses of plankton species. This inclusion will be explored and implemented in the upcoming versions of our model.

Chapter 2.1: Dealing with class size difference

Since we are dealing with an imbalanced dataset, some classes are so small that we run the risk of only encountering examples of that class in either the train or test set. This would mean, amongst other things, our model is fitted to examples it will not be tested on, or being tested on examples whose class it has never seen before. We can combat this problem in several manners. One of such is the making of multiple splits in the training and testing of our model on each of these splits, also known as K-folds. Another approach is splitting the data based on the distribution of classes, also known as stratifying, to ensure that we are neither training nor testing our model based on a dataset whose class-distribution differs significantly from the actual class-distribution of the model.

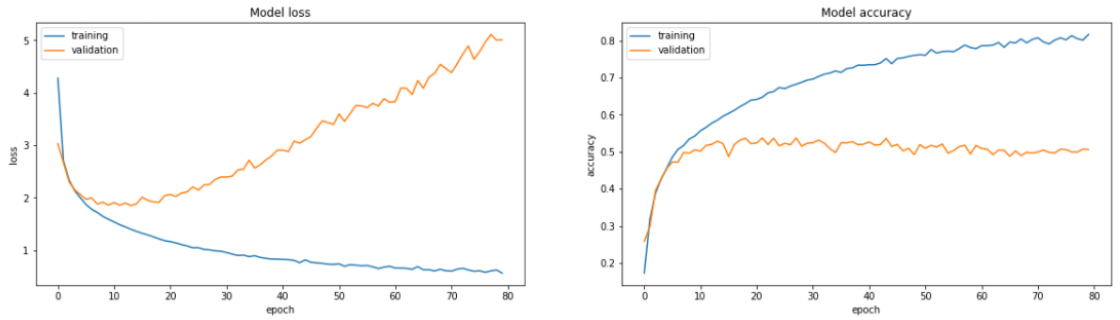
We have applied both these solutions in one go, using the Stratfied-KFold function of SK-learn. Applying these methods does not seem to directly impact accuracy scores on the simplest model. However, in theory, it should make our results more stable and reliable. This is because now, with the correct number of folds, each point of our data gets used as both training and validation set in a fold, thus assuring results are not influenced as much by the data imbalance. However, we will only use this approach for every model at the end of a chapter since the K-folds method makes training K times slower.

Moreover, another manner of dealing with the imbalance in the class size is adding weights to the model in order to balance the classes within the model and to ensure that classes with a low number of observations are preserved within the trained model. A quick test with class weights was executed, yet, when looking at the confusion matrix before the class weights and after, there was not a noticeable difference to be observed. Henceforth, for the rest of the model, we continued without the class weights.

Chapter 2.2: Overfitting, deeper network, and max pooling size

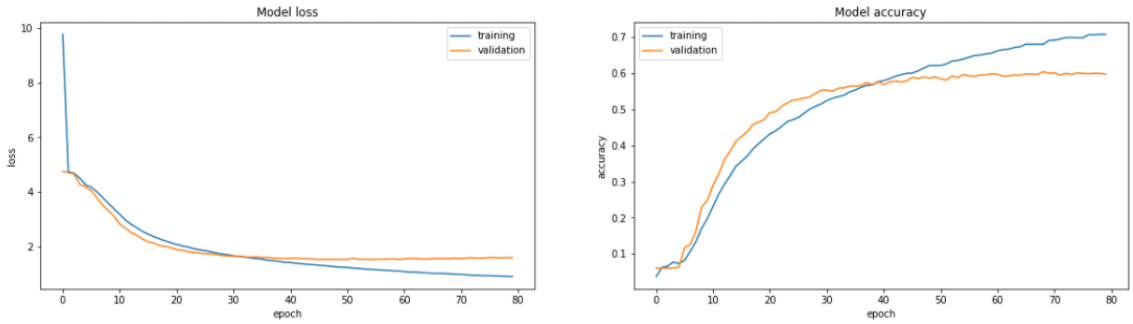
2.2.1: model A

Since images with shape (28, 28, 1) can't be divided by 3 to get an integer, we set the images to shape (27, 27, 1) for this given version. With a max pooling size of (3, 3), the shape of the images goes to (9, 9, 1) and then to (3, 3, 1) (with two pooling layers). The model seems to get to a validation accuracy of above 0.5 a bit easier, but it is still overfitting a lot.



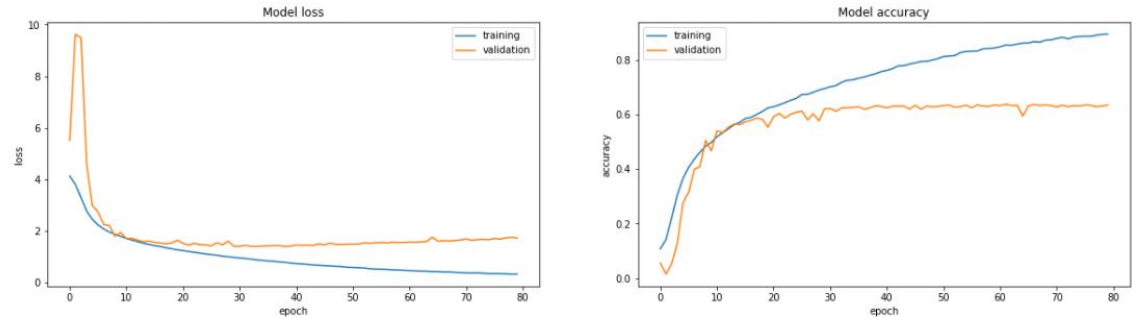
2.2.2: model B

As a next step, dropout layers were added, 3 dropout layers with a value of 0.15 after the first max pooling layer, after the flatten layer and after the hidden layer. Also, the number of nodes in the first hidden layer was set to 256, so it could learn better with these added dropout layers. This reduced overfitting. The learning parameter was manually set to 0.0001 and with these changes the model got a validation accuracy of around 61%.



2.2.3: model C

A second hidden layer was added with 512 hidden nodes, with a dropout layer after this layer with a value of 0.15. The number of nodes in the first hidden layer was set to 1024, so that the model can learn more. A batch normalization was added after the second max pooling layer. The accuracy went up to around 63-64%, but the model overfits a lot and shows drop spikes.



2.2.1: model D

To fix the drop spikes, we tried a dropout layer value after the first max pooling layer of 0.2, the dropout layer values of the dropout layers after the flatten layer and the two hidden layers were set to 0.35 (figure 2.4). The accuracy stayed the same at around 63% - 64%, but the overfitting was not that bad anymore.

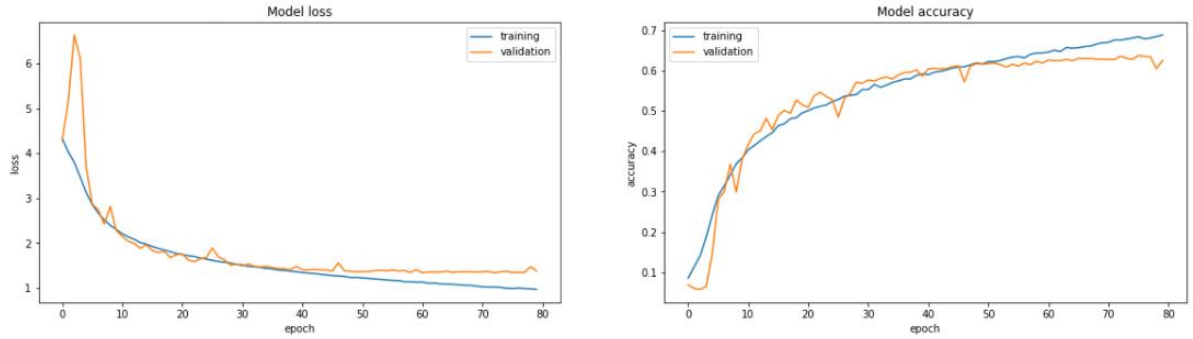


Figure 2.4: The model loss and model accuracy of the newest version of the model.

As a last step we used the StratifiedKfold. We ran it 5 times and got an average accuracy of 64% with 80 epochs each run.

Layers	Activation function	Output shape	parameters	Further data	
Convolutional 2D	reLU	(27, 27, 32)	320	Total parameters	1,196,793
Max Pooling 2D	-	(9, 9, 32)	0	Total learnable parameters	1,196,665
Dropout (0.2)	-	(9, 9, 32)	0	Non-trainable parameters	128
Convolutional 2D	reLU	(9, 9, 64)	18,496	Image input shape	(27, 27, 1)
Max Pooling 2D	-	(3, 3, 64)	0	Convolutional layer kernel size	(3, 3)
Batch Normalization	-	(3, 3, 64)	256	Max pooling kernel size	(3, 3)
Flatten	-	(576, 1)	0		
Dropout (0.35)	-	(576, 1)	0		
Dense	reLU	(1024, 1)	590,848		
Dropout (0.35)	-	(1024, 1)	0		
Dense	reLU	(512, 1)	524,800		
Dropout (0.35)	-	(512, 1)	0		
Dense	SoftMax	(121, 1)	62,073		

Table 2.1: The layers we used in this model with the information about the layers.

Max pooling with (3, 3) is a little fast, after pooling 2 times, the images are already 3 by 3 instead of the original 27 by 27. For the next milestone we want to increase the image size (chapter 2.3) so that is divisible by 2, multiple times. When the image is bigger, we can extract more data from it, so the model can learn better. The image size must be divisible by 2, because we want to use a maxpooling size of 2 by 2 so that the image size doesn't get downscaled too quickly.

Chapter 2.3: Different resizing methods

As was noted previously in the data pre-processing chapter, we chose in this version to downscale all the images to a size that was either equal or close to the dimensions of the smallest image. As we strive for the least lossy compression method, henceforth maintain the highest degree of information within the small picture. For the resizing operation, which was implemented through the OpenCV library, we had the possibility to opt for various interpolation methods that allowed for the resizing of a given image. The five possible methods are highlighted below graphically in figure 5, where they are utilized to downsize an image to a 32 by 32-dimension picture. Not only illustrates this the difference in results that the interpolation methods can have on the resizing process graphically, but also in the way it is inputted into the trainable model, as the picture generated by pixel area relation interpolation and bicubic interpolation differ significantly.

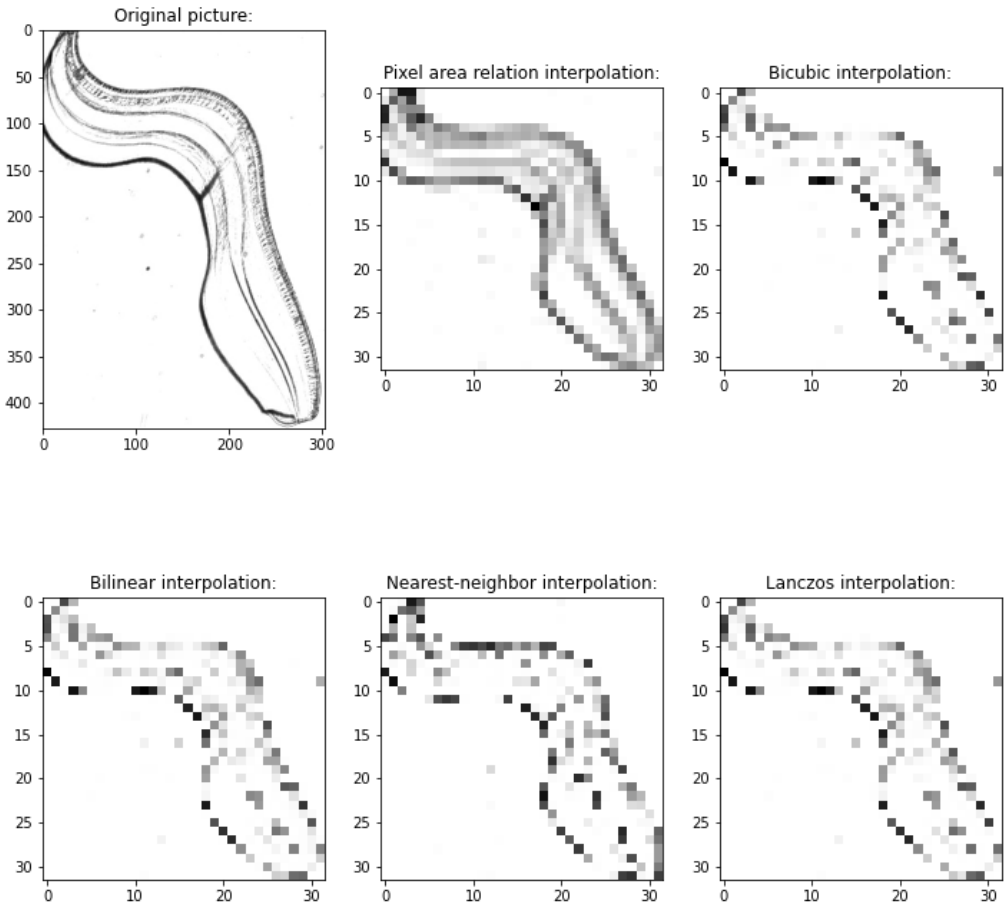


Figure 2.5: The different manners of downsizing an image with the OpenCV library.

While in the example above the most optimal compression method seems obvious, namely the pixel area relation interpolation, in practice the choice is not so clear-cut. If we look at the literature, most note that for downscaling an image, pixel area relation interpolation is the least lossy compression method, given certain factors such as the dimension of the original image as well as target dimension and their divisibility. For enlarging an image, it is noted that either bilinear or bicubic interpolation methods are considered better, yet this also depends on the previously mentioned methods. Below in figure 2.6, it is graphically highlighted how these methods result in different enlarged output. Whereas the difference between the methods is not as clear as with the downsizing, one does observe that the methods result into different images, hence data input.

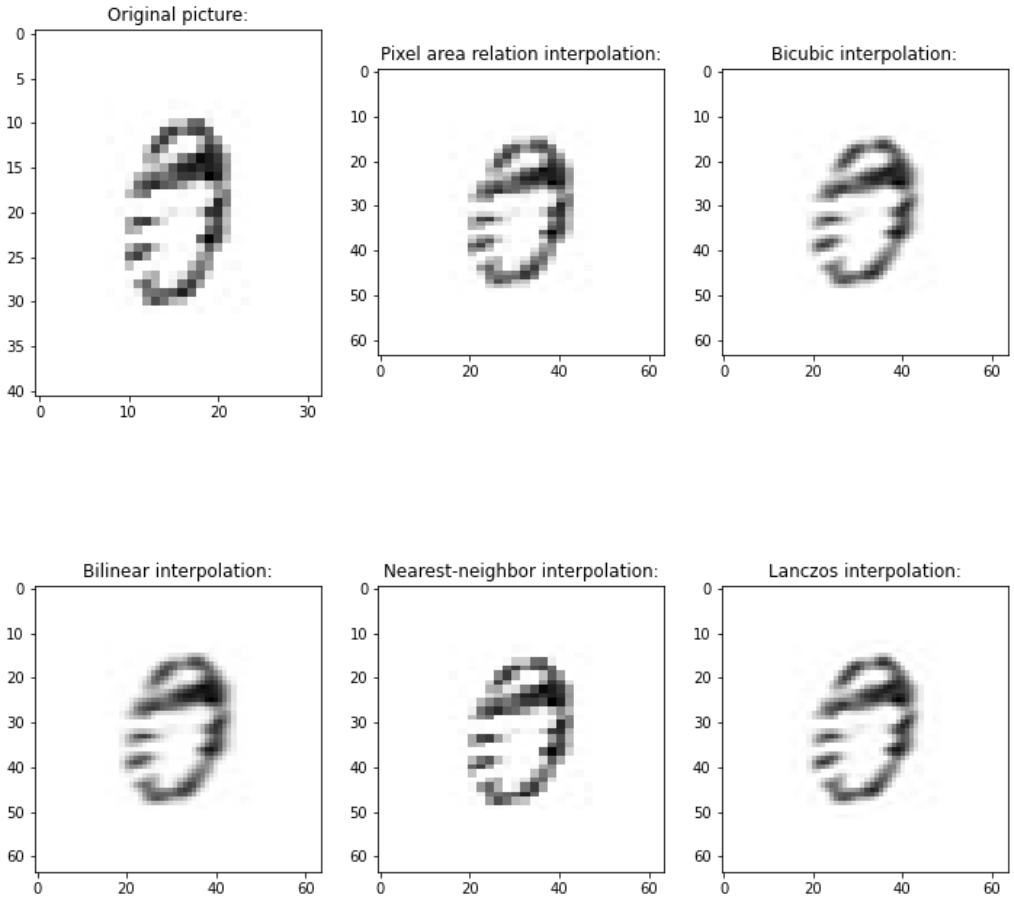


Figure 2.6: The different manners of upsizing an image with the OpenCV library.

In version 2.2 of our model, we investigated the resizing methods and its effect on the accuracy of our model, as is displayed below in table 1. From this, we see that pixel area relation results in the highest achieved accuracy, yet, in this model we implement one method for all images, without taking into account whether we are up- or downscaling an image, nor what the area or shape of the given image is. In the future versions of our model, we strive to expand the interpolation in such manner that we not only utilize the optimal interpolation method per image rather than for the whole dataset, but also that informative attributes such as area and shape are persevered after the rescaling and within the model.

Method	***Epochs***	***Validation Accuracy***
Bilinear (default)	80	0.6270%
Pixel area relation	80	0.6321%*
Bicubic	80	0.6319%
Nearest-neighbor	80	0.6106%
Lanczos	80	0.6237%

Table 2.2: The difference in accuracy achieved with the various interpolation methods.

Chapter 2.4: Conclusion

When we analyse the adjustments to our model with regards to the base (1.0) model as previously described, we see that we made an improvement of roughly 16%, from 47,85% to 64% validation accuracy. This improvement of our model can be attributed to several aspects that we have investigated and implemented. Firstly, we cleaned up our code more and investigate that distribution of misclassifications to allow for more insight. Following this, we applied a form of stratified k-fold in order to make our model more stable and potentially address the class imbalance (2.1), however, we did not observe any differences in validation accuracy, henceforth, we left this feature unimplemented for now.

Following, we continued and investigated a more extended version of our model, which eventually contained more hidden notes, drop-out layers, epochs, and utilized different input shapes (2.2). Moreover, we investigated this in combination with the different resizing methods (2.3) and their corresponding accuracy, which eventually led to the achieved accuracy of 64% with the optimal global interpolation method; pixel area relation. Future improvements are further discussed.

Chapter 3.1: Leaky ReLu, VGG-net, different input, and max pooling

Since the neural networks we've trained on the plankton dataset seem to be overfitting a lot, leaky ReLU seemed like a good idea. With leaky ReLU, there are no dead nodes, because the derivative of a leaky ReLU is never 0. When using this activation function, the dropout layers become much more powerfull; the nodes that get 'dropped out' are never dead nodes, so the effect of these layers is bigger.

We used some information from the VGG-net (Simonyan & Zisserman, 2014), where there were 2 or more convolutional layers before a pooling layer. The image size was set to 64 x 64 instead of the previous 27 x 27 chapter 3.2). With this new preprocessing step, the leaky ReLU activation function and the VGGnet set up, the following model was built:

Layer	Activation function	Output shape	Parameters	Further data	
Conv2D	Leaky ReLU	(64, 64, 32)	320	Total parameters	3,266,649
Conv2D	Leaky ReLU	(64, 64, 32)	9248	Trainable parameters	3,266,649
Max pooling	-	(32, 32, 32)	0	Non-trainable parameters	0
Dropout(0.45)	-	(32, 32, 32)	0	Image input shape	(64, 64, 1)
Conv2D	Leaky ReLU	(32, 32, 64)	18496	Convolutional layers input shape	(3, 3)
Conv2D	Leaky ReLU	(32, 32, 64)	36928	Maxpooling size	(2, 2)
Max pooling	-	(16, 16, 64)	0	Epochs	110
Dropout(0.45)	-	(16, 16, 64)	0	Learning rate	0.00009
Conv2D	Leaky ReLU	(16, 16, 128)	73856		
Conv2D	Leaky ReLU	(16, 16, 128)	147584		
Max pooling	-	(8, 8, 128)	0		
Dropout(0.45)	-	(8, 8, 128)	0		
Conv2D	Leaky ReLU	(8, 8, 128)	147584		
Conv2D	Leaky ReLU	(8, 8, 128)	147584		
Max pooling	-	(4, 4, 128)	0		
Dropout(0.45)	-	(4, 4, 128)	0		
Flatten	-	(2048, 1)	0		
Dense	Leaky ReLU	(1024, 1)	2098176		
Dropout(0.5)	-	(1024, 1)	0		
Dense	Leaky ReLU	(512, 1)	524800		
Dropout(0.5)		(512, 1)	0		
Dense	Softmax	(121, 1)	62073		

To get to this new and improved model, the following changes were made. Firstly, the learning parameter was set to 0.00009. After experimenting with a learning rate from 0.00001 to 0.0001, 0.00009 seemed like the best one for the current model. There are four max pooling layers with size 2 x 2, with 2 convolutional layers before each max pooling layer. The first two convolutional layers have 32 filters, the second two 64, the third and fourth two have 128 filters. All the convolutional layers use leaky ReLU with a value of 0.15. After each of the max pooling layers there is a dropout layer with the value 0.45.

After these layers there is a flatten layer, two dense layers with 1024 nodes and 512 nodes respectively a dropout layer after each of the dense layers, with a value of 0.5 and a dense softmax layer. The validation accuracy oscillated around the 70,7% and ended with 71.05%. The model is still overfitting, but not a lot.

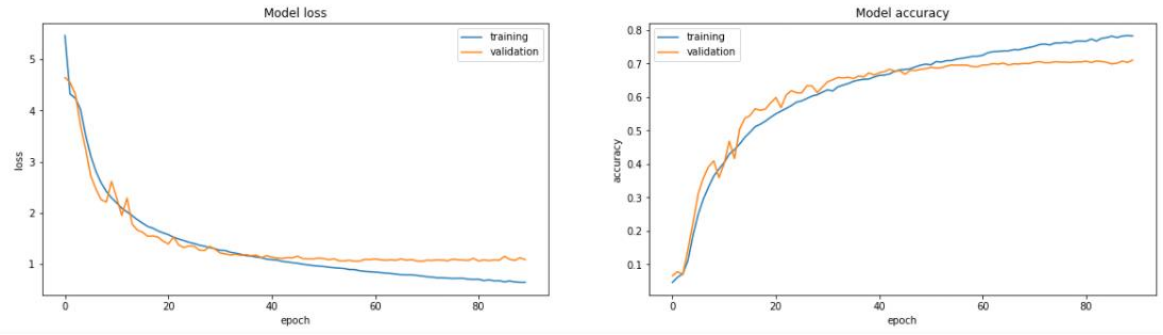


Figure 3.1: The model loss and model accuracy of the newest version of the model.

Chapter 3.2: Different resizing methods and techniques

As was noted in chapter 2.3, one could utilize several interpolation methods of resizing the images in order to maintain the highest degree of information. The problem with this approach is that the optimal method is dependent on whether we are up- or downscaling a given image, which on its turn is dependent on the original image dimensions, which vary in our data set, and on the desired input size of our image. To solve this problem of optimal interpolation method, we adjusted our code in such a manner that it would apply the optimal interpolation method per image, rather than applying the interpolation method for the data set as a whole.

The implementation of this entails that for every image, we look what interpolation method is the least lossy alteration method. We determine this by comparing the original image area (*pixel-height* x *pixel-width*) with the target area size, which was 4.096 (*64* x *64*) in the case of this version. Whenever the image area needed to be increased, we utilized the bicubic interpolation method, as we determined in the previous chapter that this was the optimal manner of interpolation for upscaling, whereas we utilized the inter area linear interpolation method for the scenario in which we needed to downscale an image.

We computed and utilized the image area rather than either height or width of an image, as the aspect ratio was not equal across the images, henceforth, it could lead to both the up- and downscaling of the axis, hence requiring more than one interpolation method. While this method of image-specific interpolation seems to have a positive effect on the found accuracy of several percentage-points, we do have to acknowledge that the aspect ratio, and therefore shape of image, is getting lost in the process. In the future versions we will look more into preserving the aspect ratio of an image, through both the addition of manual features or more advanced resizing and pre-processing methods.

Chapter 3.3: Inverse class weights

As was mentioned in chapter 2.1, the given data set has quite a significant imbalance with regard to the class sizes. We tried to counter this imbalance within the model in the previous chapter by assigning weights to classes that had a low frequency, to avoid the scenario where the model would not train these classes, hence misclassifying them structurally. We came to the conclusion that such a manner of class-weighting, where more weight was assigned to smaller classes, did not seem to improve our model, henceforth, it was left out of the following utilized models.

For this given version of the model, we again tested for the utilization of class weights, yet we took a different approach. As was observed in chapter 2.0, large clusters of misclassifications occurred within the predominant classes and their respective subclasses. To combat these large clusters of misclassifications, we experimented with inverse class-weights, where we assigned more weight to classes that were overrepresented rather than underrepresented, with the expectation that our model would be better in dealing with these clusters that seemed to be present. After testing with our most recent model, we concluded that the given inverted weighted classes did not alter the achieved accuracy significantly, as was the case with the ‘regular’ weighted classes. Henceforth, we excluded the class weights from the model for now, again.

Chapter 3.4: Conclusion

The newest model had a maximum accuracy of 71.05%, this is around 7% better than the previous model. With this new and improved model, we think it is time to add augmentation for the next milestone. For the next milestone we could look into more filters for the two convolutional layers at the end of the model. A second thing we might look into is two more convolutional layers and one max pooling layer, because the filters we put into the flatten layer, are 4x4 and 2x2 with more filters might be better. The validation loss stagnates, but it doesn't drop. This means that the dropout layers are doing their job. The training accuracy keeps rising, this means that we can still get some gain in validation accuracy. The amount of overfitting could be reduced by increasing the leaky ReLU value, but that is not sure yet.

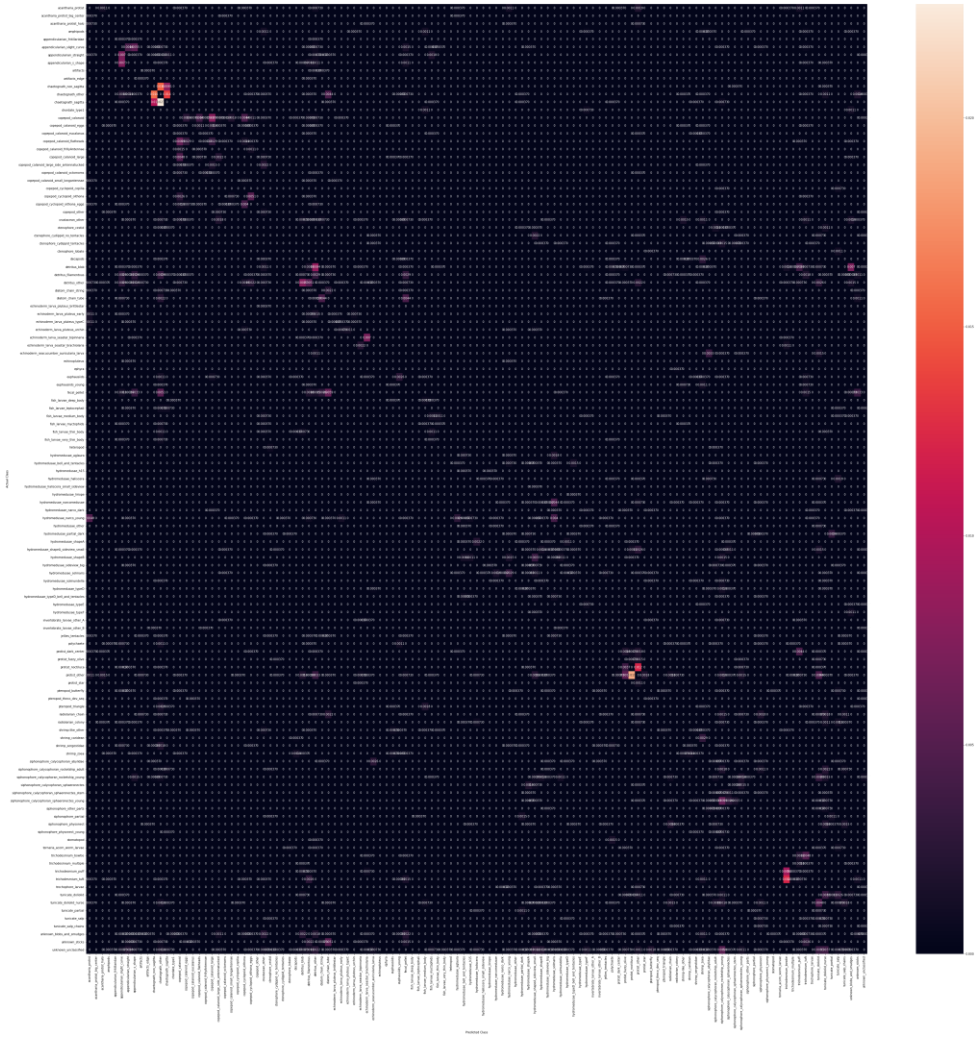


Figure 3.2: Confusion matrix with an excluded diagonal based on the relative number of classifications.

Moreover, if we look at the distribution of the misclassifications in our model, we see that the pattern of clustering around large classes with more than one subclass continues to be present in our newer model. Moreover, this trend even seems to intensify to some degree, as the clusters tend to account for a higher percentage in the misclassification relative to previous versions, as is highlighted in figure 3.2 above.

This trend indicates that the increase in accuracy that is achieved by our model is most likely due to the better classification of classes in general, rather than its ability to distinguish subclasses from one another. It will be interesting to see whether in the upcoming versions, the model will break with this trend when the model is extended through the addition of (manual-)features and augmentations.

Chapter 4.0: Model adjustments

4.0.1: Padding preprocessing

During this version of the model, we also investigate the possibility to expand rescaling process to preserve more features within the picture itself. As will be noted in chapter 4.4, we also expand our model to include features that are not being part of the convolution, such as possibly aspect ratio or other information. However, we also investigated the possibility to include such features, like aspect ratio, without excluding it from the convolutions. As was previously highlighted, we use a method that chooses the most optimal resizing method to resize all images to the desired dimension. A problem that arises from this method is that images that are not (perfectly) cubic will be distorted more in the rescaling process, as the aspect ratio (width : height) of the original image is lost. A way of solving this loss of information if padding the image on the shortest dimension before rescaling, as is highlighted below in figure 4.0.

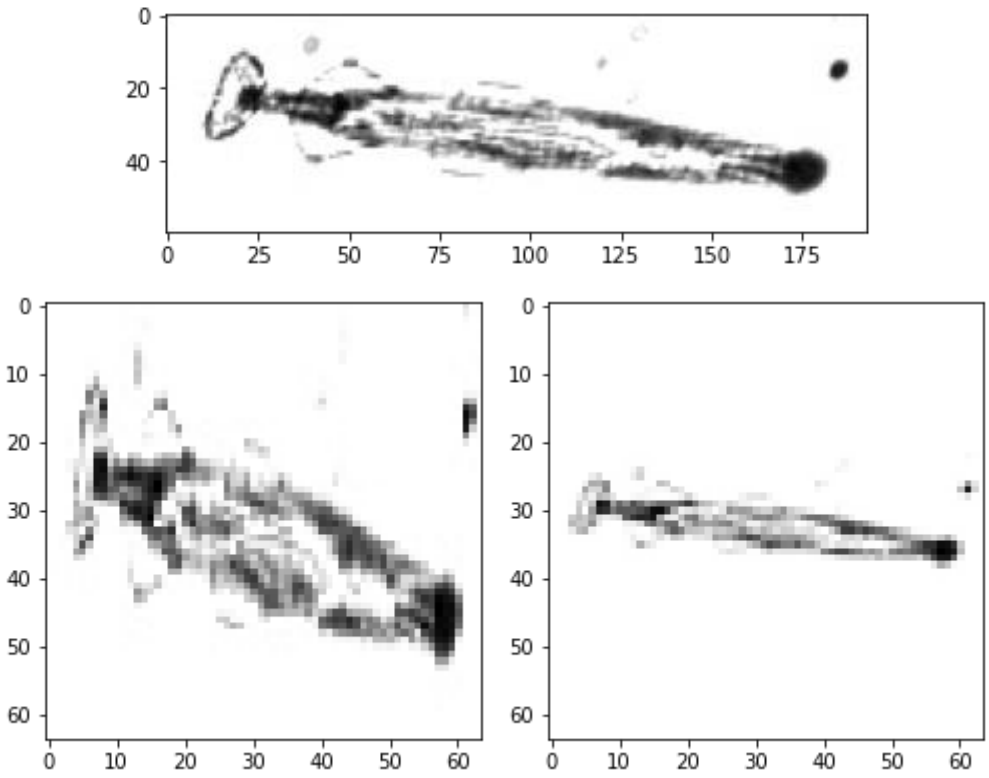


Figure 4.: The original image (top), and the rescaling without (left) and with padding (right).

We can observe from the figure that the padding has multiple effects which we should take into consideration. When utilizing padding, the general shape of the picture, and therefore aspect ratio, is being preserved and has the dimensions suitable for our model. One can observe, however, that certain information is also lost with this method, as the interior of the given plankton is relatively more faded compared to the non-padded version. It is therefore unclear a priori whether the padding method will influence our findings positively or negatively. Due to the time and computing-power constraints, we did not yet have the opportunity to investigate its effect to the fullest, hence, this will be presented at a later stage.

Chapter 4.1: Augmentations

4.1.0: General

As we try to optimize our model, we will investigate the usage of augmentations to our dataset in order to train our model more accurately. The augmentations that we are testing are used in the preprocessing part of the model, and are part of the ImageGenerator module within *TensorFlow/Keras*. Even though the list of possible augmentations consists only out of 22 possible augmentations, the possible combinations are endless, henceforth, as we are constraint with regards to time and computing power, we will use logical reasoning to try and optimize the set of utilized augmentations. For each of the listed augmentations below, we will highlight the reason for their usage.

4.1.0: Horizontal and Vertical Flip

The first pair of augmentations that were considered were the horizontal- and vertical-flip, which is mirroring a picture on either axis. This was considered following the observation that some classes contained data which was mirrored in either way, which followed most likely from the observation process and the fact that they were biological beings, hence differing in direction when observed. This 'mirroring' is best illustrated with figure 4.1, which shows this occurrence for a class of plankton.

Figure 4.1: Three different images of the chaetognath-class.

We applied all combinations of the horizontal- and vertical-flip augmentation, yet we failed to find any improvements in our utilized model, and it even obtained a lower accuracy of roughly 1.5-2%. The most logical reason for this model deterioration is that not all classes were subject to this 'mirroring' in the observations, henceforth, where some classes might have benefitted from the in data augmentation and trained better, most classes did not due to the absence of this mirroring.

4.1.1: Shear Range and Rotation Range

As is noted in chapter 4.1.0, the images of plankton are not all identical due to the manner of observation, henceforth, augmentations to the perspective of the image can be useful. Another augmentation that proves to be useful is the shearing and rotation of an image. These methods have a similar effect as both types of flipping, yet are more possibly conservative and apply to most classes of plankton, as was not the case with the flipping. Below we tested multiple tested multiple compositions of augmentations.

4.1.3: Zoom Range

Lastly, we investigate the usage of a zoom range augmentation in our model, with ranges such as [0.75 - 1.25] and [0.5 - 1.5]. This follows from the hypothesis that the observations differ in dimensions, henceforth, a zoom would allow our model to train more on this possible difference. When testing, we see that it negatively impacts our accuracy by roughly 4% in both given zoom-ranges. This most likely follows from the fact that most of the images are already of the same size and cropped evenly, as can be observed in the data set. A zoom would therefore only introduce more noise into our model, resulting in the observed decrease in accuracy.

Chapter 4.2: Model alterations

Following our findings regarding the model augmentations in chapter 4.1, we will continue with the deepening of our model and investigate different alterations compared to the model as was presented within chapter 3. Moreover, we'll utilize and implement the augmentations from chapter 4.1 that were found to have a positive effect on our validation accuracy.

4.2.1: model A

The models in this section start with a slight change, which is the number of filters in the fourth set of convolutional layers, this went from 128 to 256. This was done to give the model a bit more complexity. We noticed that the training accuracy and validation accuracy became closer and closer (which is good), so we wanted the model to be able to learn more.

As a first step, a shear range of around 20 would seem like a logical value. Using the latest model from the previous milestone, the shear range augmentation was set to 20. The model ran for 110 epochs, the model learns very slowly, but at epoch 100, the validation accuracy is above 71% for almost all the remaining epochs. The average accuracy around which the model oscillates at the went from 70.7% to 71.2%, so there is a slight increase.

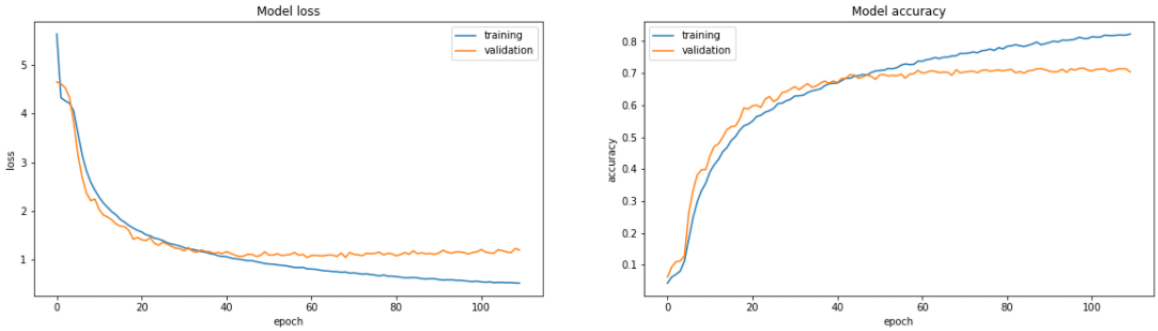


Figure 4.2: Loss- and accuracy function of the model A.

4.2.2: model B

Since the model is slightly overfitting, the dropout values will all be set to 0.5. Some epochs get a bit higher validation accuracy, but it is not that much. The highest value was 71.55% whereas previously this was 71.4%.

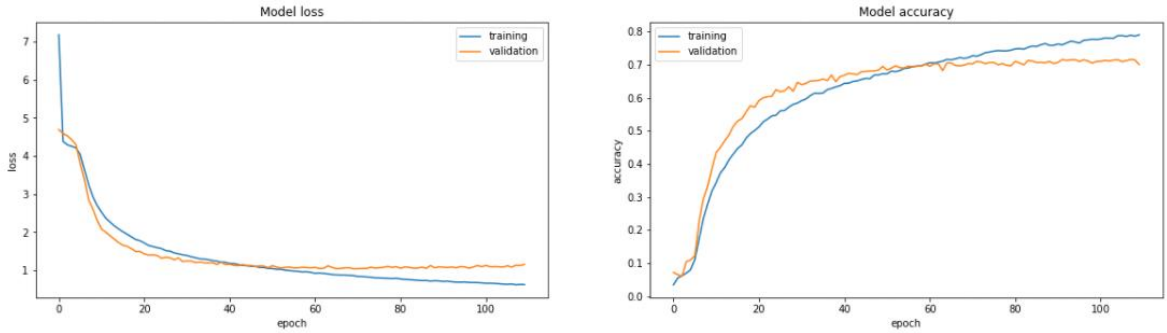


Figure 4.3: Loss- and accuracy function of the model B.

4.2.3: model C

In this model, the learning parameter was set to 0.00012 instead of 0.00009. The highest validation accuracy in this model was 71.95% which is higher than the last model. This was the accuracy at the last epoch, so maybe the number of epochs must be higher in the future models.

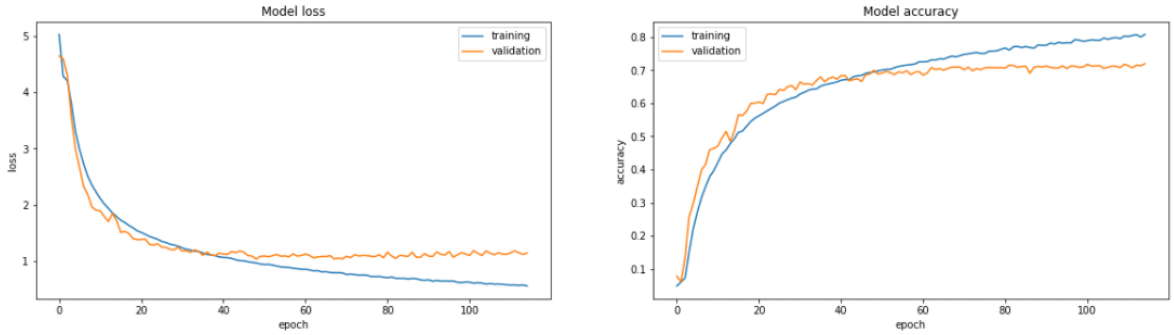


Figure 4.4: Loss- and accuracy function of the model C.

4.2.4: model D

In this version of the model, rotation range with a value of 45 degrees was added. This seemed like a logical augmentation, because some images looked rotated compared to others from the same class. The epochs were set to 200 and the learning rate was set to 0.00012. The highest validation accuracy we got was 74.00%, the model oscillated around 73.3% and the last epoch had a validation accuracy of 73.68%.

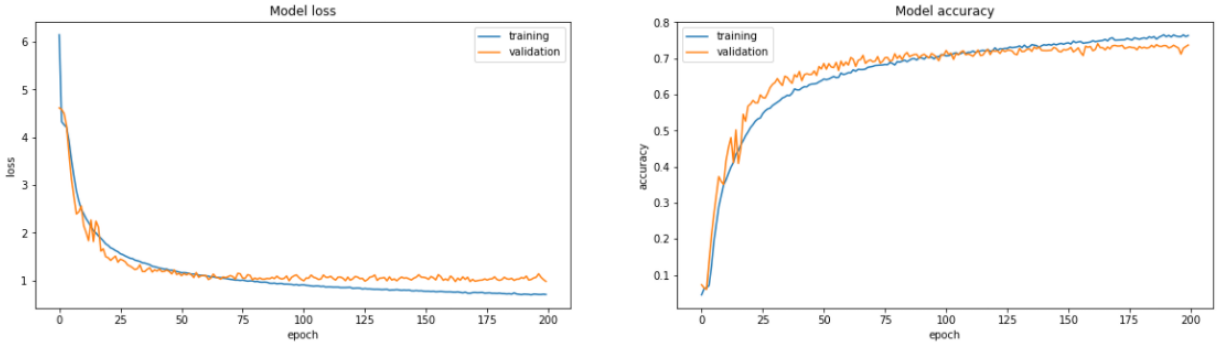


Figure 4.5: Loss- and accuracy function of the model D

Layer	Activation function	Output shape	Parameters	Further data	
Conv2D	Leaky ReLU	(64, 64, 32)	320	Total parameters	5,953,811
Conv2D	Leaky ReLU	(64, 64, 32)	9248	Trainable parameters	5,953,881
Max pooling	-	(32, 32, 32)	0	Non-trainable parameters	0
Dropout(0.5)	-	(32, 32, 32)	0	Image input shape	(64, 64, 1)
Conv2D	Leaky ReLU	(32, 32, 64)	18496	Convolutional layers input shape	(3, 3)
Conv2D	Leaky ReLU	(32, 32, 64)	36928	Maxpooling size	(2, 2)
Max pooling	-	(16, 16, 64)	0	Epochs	200
Dropout(0.5)	-	(16, 16, 64)	0	Learning rate	0.00012
Conv2D	Leaky ReLU	(16, 16, 128)	73856		
Conv2D	Leaky ReLU	(16, 16, 128)	147584		
Max pooling	-	(8, 8, 128)	0		
Dropout(0.5)	-	(8, 8, 128)	0		
Conv2D	Leaky ReLU	(8, 8, 256)	295168		
Conv2D	Leaky ReLU	(8, 8, 256)	590080		
Max pooling	-	(4, 4, 256)	0		
Dropout(0.5)	-	(4, 4, 256)	0		
Flatten	-	(4096, 1)	0		
Dense	Leaky ReLU	(1024, 1)	4195328		
Dropout(0.5)	-	(1024, 1)	0		
Dense	Leaky ReLU	(512, 1)	524800		
Dropout(0.5)		(512, 1)	0		
Dense	Softmax	(121, 1)	62073		

Tabel 4.1: The layers we used in this model (D) with the information about the layers.

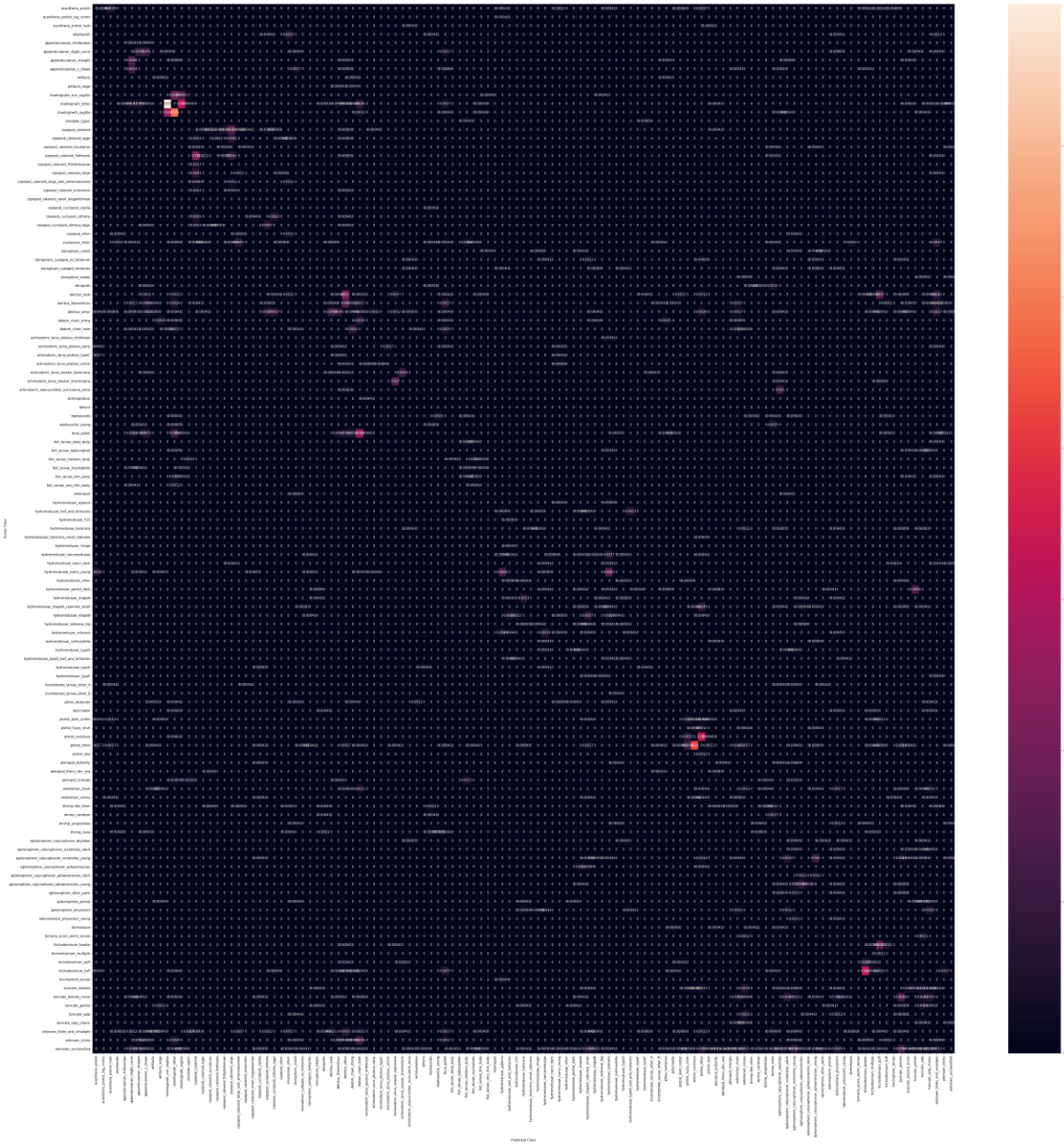


Figure 4.6: *Confusion matrix with an excluded diagonal based on the relative number of classifications.*

Chapter 4.3: Handling mixed input data

Our current model is built to handle image data, and only that. The image data is put through a convolutional neural network and result is flattened and put through a number of fully connected layers. But what if we had more data? Maybe there are additional features we can extract from the original data that could be used by the model to learn to classify the images. In order for the model

to be able to learn something from additional features, we'd have to make it able to handle mixed input data. To do this, we made use of the Keras functional API.

The network was split into two parts. The first part consisted of the sequential model we had before, a convolutional neural network which handles the input images, ending with a flattening layer. For the second part, we concatenated the output of the first part of the network with an array of extra features and used that as the input for a number of fully connected dense layers, which makes up the second part of the network. This part of the network is programmed using the Keras functional API.

We tested this model using three extra features: the height, the width, and the aspect ratio of the training images. However, this didn't lead to an increase in accuracy (see image). In later versions, more and different features may be tried.

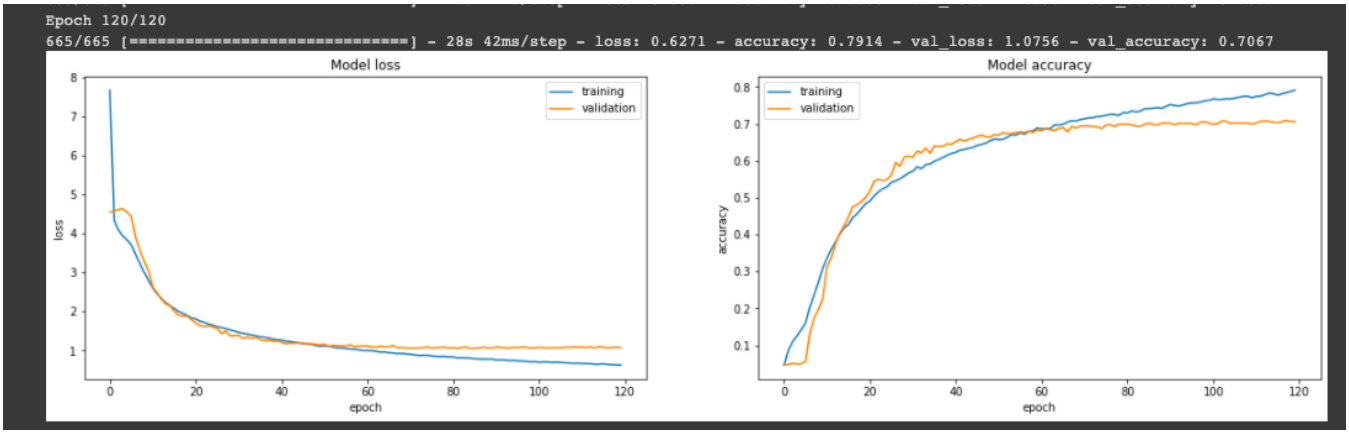


Figure 4.7: Loss and accuracy of the model with the mixed input data.

Chapter 4.4: SMOTE

As a last experiment to try and combat misclassification due to class imbalances, we implemented the Synthetic Minority Oversampling Technique (SMOTE). This technique creates synthetic training examples by taking a training example from a minority class (or a specific given class) and finding the K nearest neighbours for this training example. One of these neighbours is chosen at random, and a new data point is created which lies between the original training example and the randomly chosen neighbour. In this manner, we can generate a large number of new training examples which could assist in correctly classifying classes with very few training examples. It is important to note, however, that this technique involves creating and training on the new training examples, thus significantly increasing training time. Therefore, we implemented this technique by selectively oversampling the classes on which our model made a noticeable number of errors. For these classes, synthetic data points were created to increase the number of data points to roughly the number of training examples in the largest class, 1300. Sadly, this technique did not yield better results, as the model's validation accuracy halted around 50%. Therefore, we cut the training short as we were already on a tight schedule and there had been virtually no increase in accuracy for quite some time. When we inspected the images generated by SMOTE and compared them with real training examples from that class, they did not seem to truly capture the features of the class, see figure 4.8. We speculate that this is due to the fact that some classes contain plankton that are visually very distinct with a large range of features. Simply taking the mean of said features does not accurately reflect the characteristics of the class. Perhaps if we would have the time to tweak the parameters of

SMOTE further, or even make them learnable parameters we could achieve more success with this technique. However, we believe that more advanced and fruitful methods exist to increase training data for minority classes such as deepSMOTE (Dablain et al., 2022), and with more time on our hands we would certainly have tried to implement them.

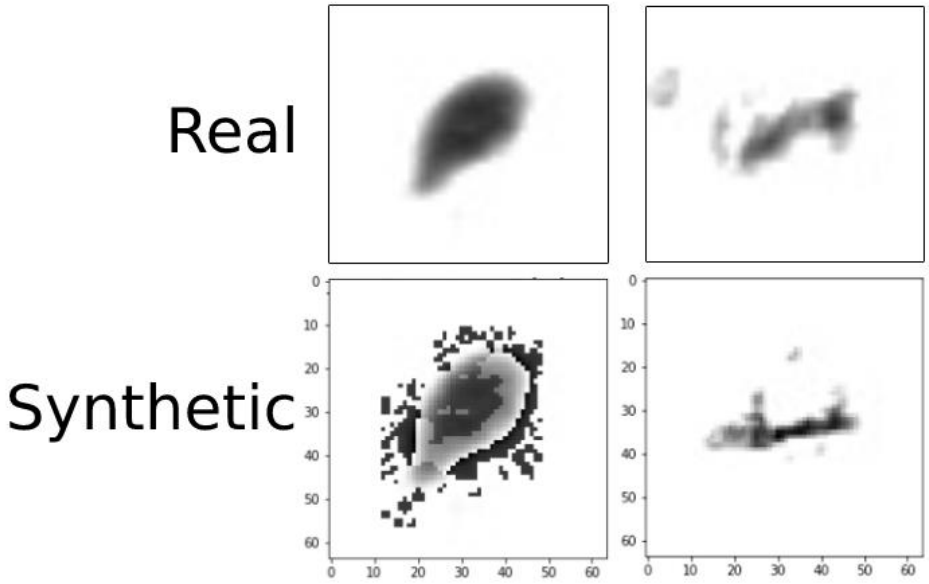


Figure 4.8: Synthetic and real training examples of the Detritus Blob compared

Chapter 4.5: Conclusion

During this chapter, we primarily focussed on decreasing the distance between our validation accuracy and our training accuracy, I.e. reducing overfitting. By performing data augmentation, we achieved just that. During testing, the last version of our model, model D, achieved a validation accuracy of almost 74%, thus improving on our previous model by 3%. In the current version, validation and training accuracy are almost indistinguishable, thus suggesting that adding the data augmentation, especially the shearing range, did indeed solve the slight overfitting of our model (figure 4.5) . Additionally, all models in this chapter saw an increase in the number of filters in the fourth convolutional layer which was changed from 128 to 256. This change was made to make the model more robust against the noisy data that comes with performing data augmentation. Moreover, we tried to add extra features to the dense layer part of our network. Specifically, we added the original image sizes as features to try and salvage any useable data before our images are resized. This method did yield a better validation accuracy, but since the possibilities of adding other streams of data are quite extensive, with more time on our hands this may prove to be a fruitful endeavour. Lastly, we implemented SMOTE to solve the inaccuracy that comes with the class imbalances. Sadly, this technique did not yield any improvements, which seems to be due to the large variety within certain classes and the small variety between other classes, see figure 4.9. All in all, we are quite pleased with the validation accuracy of our final model. There are still some approaches that we would like to have tried, but we must accept that in projects such as these there is always something else you can try. Nevertheless, we think have tried a wide variety of experiments and truly deepened our knowledge on how to deal with challenging image datasets.

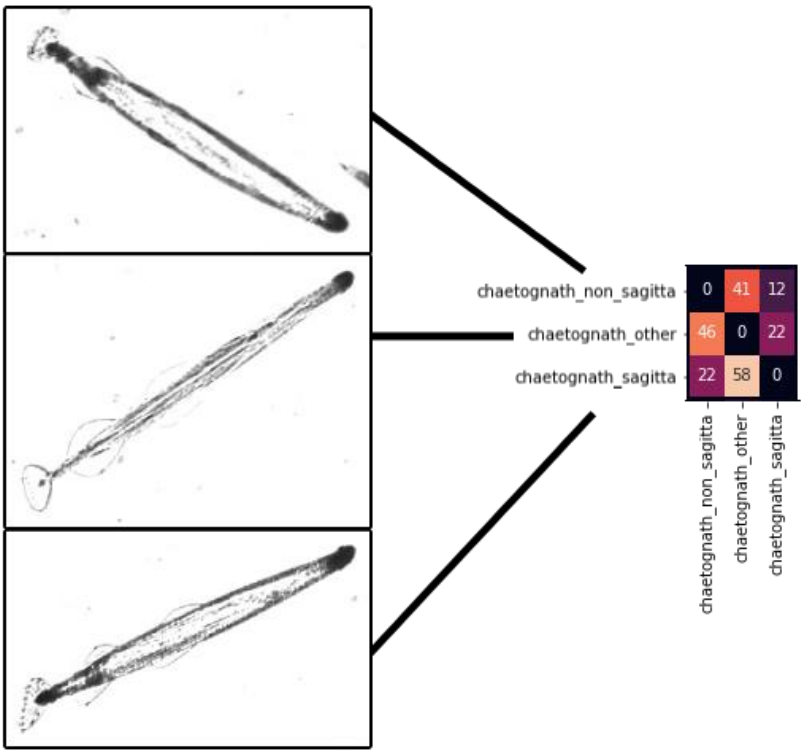


Figure 4.8: comparison between training examples from classes that led to some of the most erroneous predictions to show their visual similarity

Chapter 5.0: Last model adjustment

The only thing that was done in this chapter is running the model with 400 epochs instead of 200. The validation accuracy oscillated at around 74-75% (figure 5.1).

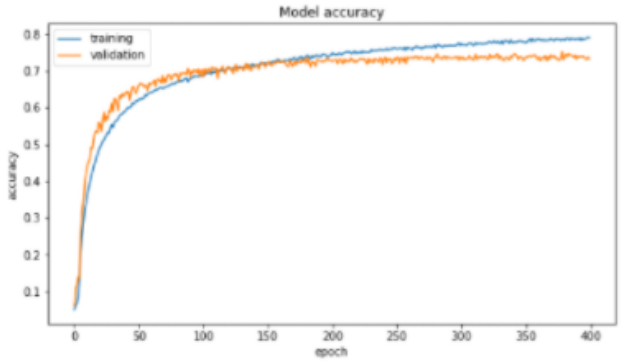


Figure 5.1: The final model with an accuracy of 74-75%.

This model seems to be slowly overfitting, but that can't be said with complete certainty. Overall, this model performs pretty good.

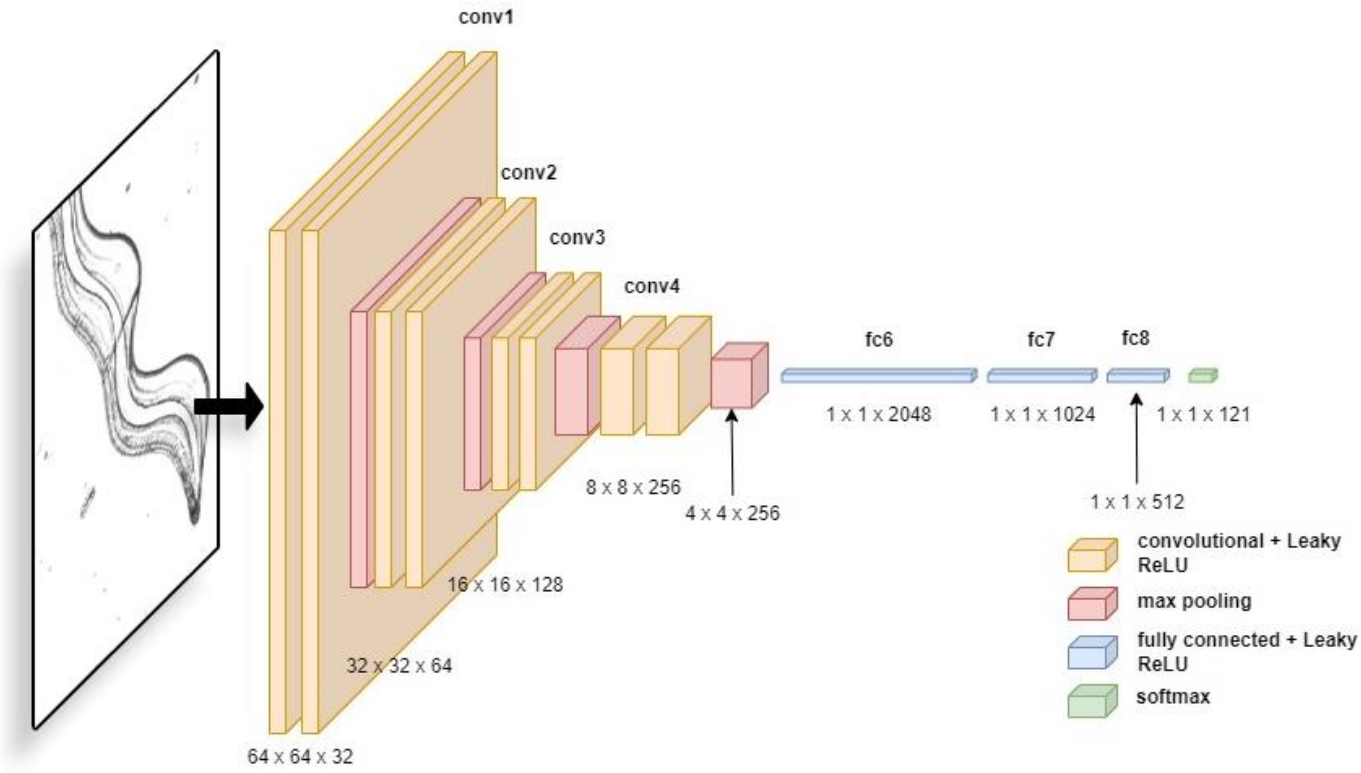


Figure 5.2: Schematic overview of the final version of our CNN

References

Dablain, Krawczyk, B., & Chawla, N. V. (2022). DeepSMOTE: Fusing Deep Learning and SMOTE for Imbalanced Data. *IEEE Transaction on Neural Networks and Learning Systems*, *PP*, 1–15. https://doi.org/10.1109/TNNLS.2021.3136503

Simonyan, & Zisserman, A. (2014). *Very Deep Convolutional Networks for Large-Scale Image Recognition*.