Machine Learning II- Final Project: Individual Report

Iliana Maifeld-Carucci

December 10, 2018

---

1 Introduction

Plankton are a necessary component of our ecosystem. They are small and microscopic organisms that float or drift in the ocean, providing a critical food source for larger organisms as they are the first link in the aquatic food chain. Since they are the primary food producers for ocean life, a decrease or loss of these fundamental species could detrimentally impact our ecosystem. Thus, it is not only crucial to measure and monitor plankton population with underwater cameras, but to be able to classify the high volume of images rapidly, as manual analysis would take about a year for images captured in one day. Therefore, to address this efficiency problem regarding image recognition, we will be applying deep learning techniques to build a convolutional neural network to classify images of various plankton in a timely and accurate manner.

We approached our project from a mostly collaborative perspective. Each of us researched the theory, methodology, and main components of the code, as well as worked together to edit and add details to our report, presentation, and code. I mainly focused on the theory and the code, while Alexa was the driver for the plot code and report, and Bryan took control of the presentation and proposal.

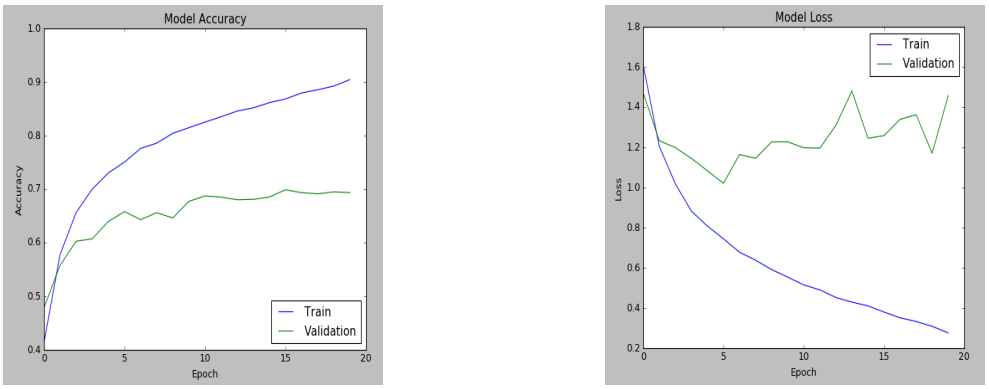
2 Individual Work

I did most of the coding and testing. To start with this involved creating code to separate our data into train, validation, and test sets. From there, I created a baseline three layer convolutional neural network using 20 epochs, a batch size of 175, a categorical cross entropy loss function, and an RMSProp optimizer. For this baseline I only implemented the most basic preprocessing steps of rescaling the channel values and resizing the images to 64 by 64 pixels. I then tested three additional preprocessing steps oftentimes used for images which were mean normalization, standardization, and zca whitening. Moving on to the algorithm development, we first approached our issue of unbalanced data in our classes. I did substantial research on

methods to tackle this issue and while there are a couple different approaches such as data augmentation and error weighting or cost-sensitive learning, we decided to do error weighting. Therefore, next I implemented this weighting to see if it improved our model. Following this, I took charge of testing various parameter inputs for our algorithm like adjusting the resizing dimensions, playing with different activation and loss functions, as well as seeing if early stopping would help prevent our model from overfitting. After Alexa and I finished all the algorithm development testing, I then experimented with the most promising parameters, and used the highest performing combination to test various experimental set ups.

3 Individual Output and Results

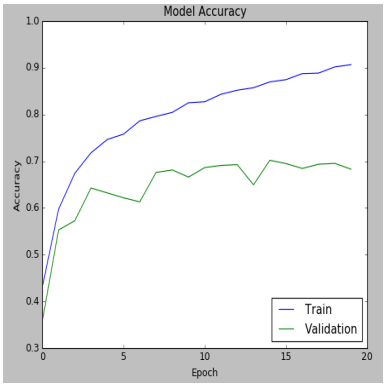
The first step in developing our algorithm was splitting up our data into training, validation, and testing sets. We decided to use 70 percent of our data for training, 20 percent for validation, and 10 percent for testing. We chose this split ratio because our model will have a good amount of hyperparameters to tune, so we need a larger validation set to employ practice tests on. To accomplish this, we wrote a function that moves a specified percentage of random samples of data from one folder to another. Once we had our data properly proportioned, I coded the baseline model that I spoke of above using standard Keras syntax. Aside from rescaling and resizing for basic preprocessing steps and using max pooling and dense layers, this was a completely bare bones model. For this model we got a training accuracy of 90.48% and a validation accuracy of 69.35% (Figure).



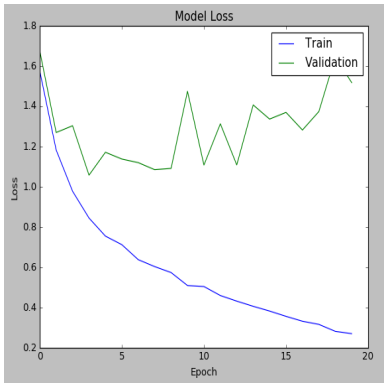
(a) (b)

Figure 1: Accuracy and Loss of the Base Model

I proceeded to test three different preprocessing methods. First, I used the keras parameter featurewise_center to apply mean normalization across the dataset. Doing this sets the mean to zero and helps minimize the variance which makes it easier to solve for our coefficients. This increased our training accuracy to 90.68% and our validation accuracy to 69.35% (Figure).



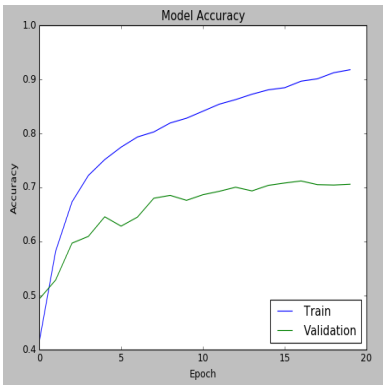
(a)



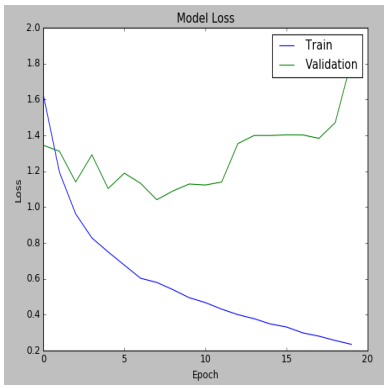
(b)

Figure 2: Accuracy and Loss with Mean Normalization

Next, I added the parameter featurewise_std_normalization to use standardization. Standardization makes the standard deviation one and helps our algorithm converge which again increased our training accuracy to 91.77% and validation accuracy to 70.59% (Figure).



(a)

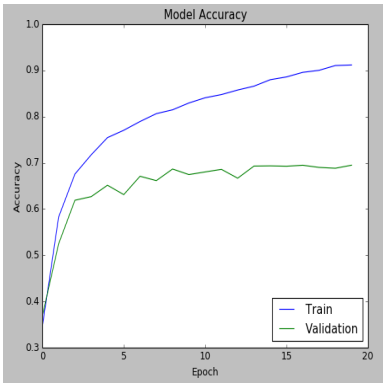


(b)

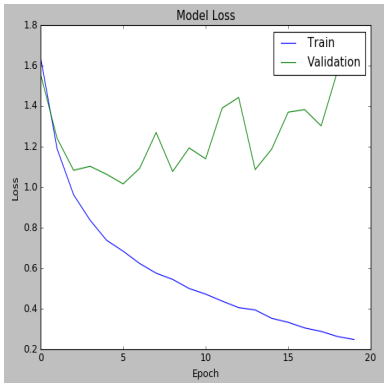
Figure 3: Accuracy and Loss with Standardization

Finally, I tried implementing ZCA whitening with the keras parameter zca_whitening using the default epsilon. This preprossing step whitens the image while sharpening the edges. However, when we added this step, our training accuracy went down to 91.19% and it dropped out validation accuracy to 69.49% (Figure) so we decided not to use this preprocessing for our model.

Once we had our preprocessing done, I moved on to dealing with our class imbalance by adding the keras parameter class_weight to our .fit_generator() function. Error weighting focuses more on under-represented classes and adjusts the weights in the loss function accordingly by increasing the weights for under-represented classes, and decreasing the weights for over-represented classes, giving each class a balanced platform in the model and removing the bias influence to-



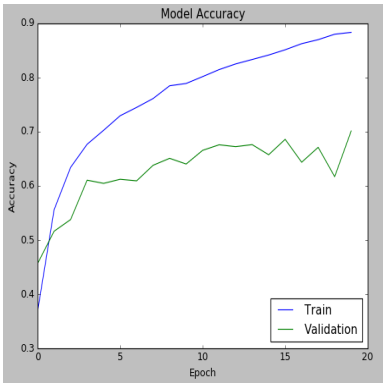
(a)



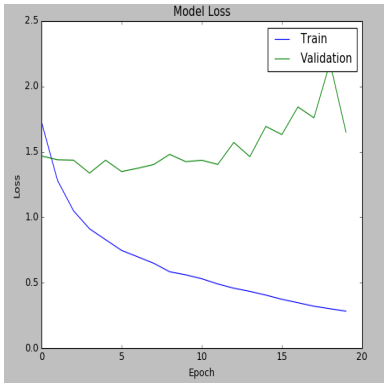
(b)

Figure 4: Accuracy and Loss with ZCA Whitening

wards larger class sizes that would have occurred otherwise. To calculate the balance of weights for each class I used sklearn's class weight package to automatically calculate a balanced set of weights by taking into account the number of samples in each class in comparison to other classes. This dictionary of adjusted weights was then put in to the class_weight parameter which resulted in a training accuracy of 88.35% and a validation accuracy of 70.14% (Figure). While our training accuracy decreased this is actually a sign that our model is not overfitting as much and the weights are working as they should, so we kept the weighting moving forward.



(a)



(b)

Figure 5: Accuracy and Loss with Error Weighting

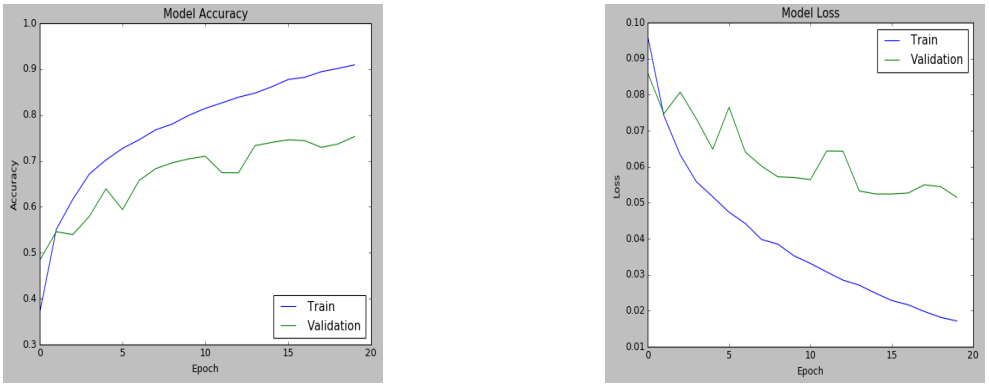
From here I transitioned into helping Alexa do hyperparameter tuning of our algorithm using the preprocessing and weighting described above. I experimented with resizing dimensions, playing with different activation and loss functions, as well as seeing if early stopping would help prevent our model from overfitting. For resizing I tried different values as described in the table below and found that a sizing dimension of 64 by 64 gave us the best performance where our training accuracy was 88.35% and our validation accuracy was 70.14%.

Image Width	*Image Height*	*Train accuracy*	*Train loss*	*Validation accuracy*	*Validation loss*
32	32	75.86	.6229	62.38	1.4873
64	64	88.35	.2821	70.14	1.6505
96	96	93.05	.1751	67.08	2.0698

Next, I examined different activation functions that are shown in the table below and found that while sigmoid provided the best accuracy, we decided to use softmax because the function sums the output probabilities from the fully connected layer to one which is commonly used for CNNs.

Activation Function	*Train accuracy*	*Train loss*	*Validation accuracy*	*Validation loss*
Softmax	90.68	.2345	68.03	1.8731
Softplus	89.52	.3610	68.77	1.7412
Sigmoid	89.59	.2622	68.83	1.8125

After this I explored using different loss functions and found that using the mean squared error gave us the best performance where training accuracy was 90.94% and validation accuracy was 75.3% (Figure).

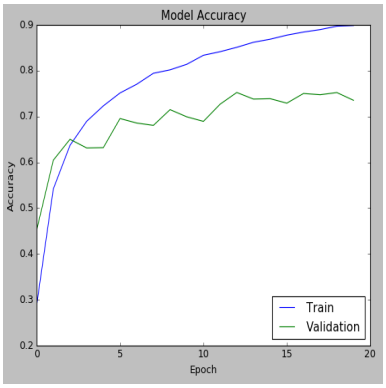


(a) (b)

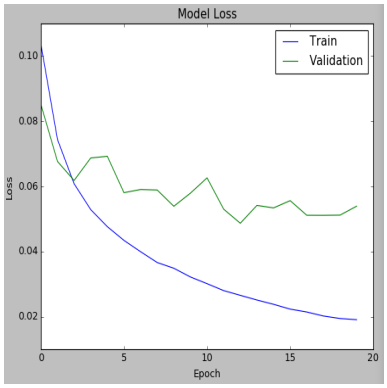
Figure 6: Accuracy and Loss using MSE as Loss Function

Finally, I implemented early stopping which helps prevent from overfitting by stopping the training as soon as the loss starts increasing again. However, this didn't help our accuracy any so we decided not to include it.

After we experiemented with the best performing parameters from our algorithm development, I moved forward to experiment with different enviromental set ups. Here I experimented with different mini-batch sizes and different number of epochs. I found that using a batch size of 100 gave us the highest accuracy with a training accuracy of 89% and a validation accuracy of 74.05% (Figure), while using 50 epochs provided a train accuracy of 94.71% and validation accuracy of 75.1% (Figure). We used both of these values for our final model.

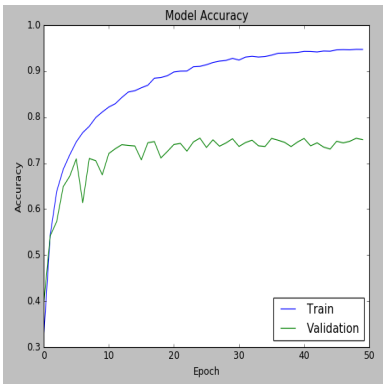


(a)

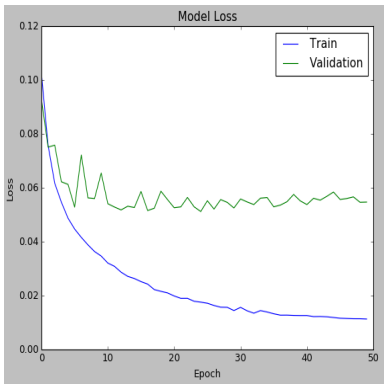


(b)

Figure 7: Accuracy and Loss with Batch Size 100



(a)



(b)

Figure 8: Accuracy and Loss with 50 Epochs

4 Summary and Conclusions

We found that when running convolutional neural networks on image data there are many things to consider and weigh against each other, but even when you have a small amount of unbalanced data, CNNs still are able to perform quite well. It was enjoyable to learn about how to deal with unbalanced data and the sort of techniques that can be used to counter act this since unbalanced data is pretty common in the real world. If we were to continue this work we could do more to explore different ways to handle unbalanced data and boost our performance.

5 Percent of Original Code

Approximately 15 percent of my code is copied.

6 References

Image Generators Moving Data Function Data Preprocessing and Augmentation Sectioning out Data Normalization and Augmentation Explanation PCA Whitening vs. ZCA Whitening Unbalanced Data Early Stopping Preprocessing Steps Keras Image Generator