Predicting Plankton Classification with Convolutional Neural Networks

Alexa Giftopoulos | Iliana Maifeld-Carucci | Bryan Egan

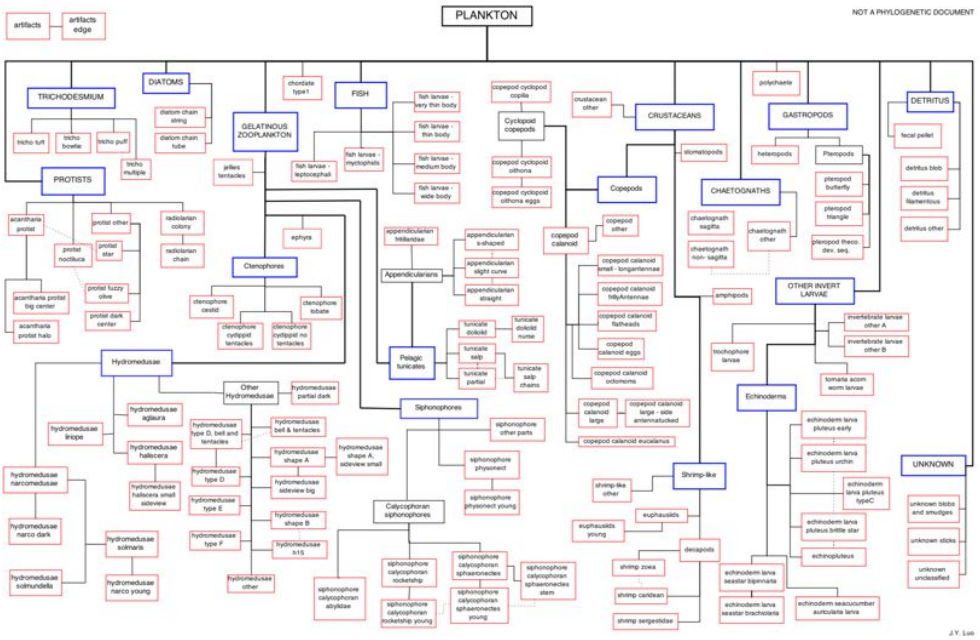
Introduction

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.

Data Background

Our dataset was obtained from Kaggle and includes colored images of plankton captured from Oregon State University's Hatfield Marine Science Center. We were provided with a file containing folders of images for each class. Looking at the image below, the original plankton data contained a hierarchy of classes with 11 overarching species (large blue), 7 subspecies (small blue), and various remaining subspecies (red).

To lessen the size and complexity of our classes, we simply rolled up each subclass into their main, overarching class, resulting in a total of 11 classes. Next, we eliminated any classes that did not provide a sufficient number of images to train and test on, specifically, classes with less than a few hundred images. As a result, for our final data, we will be using a total of 8 classes with the corresponding number of images: *Chaetognaths* (3,443)*, Crustaceans*(4,792)*, Detritus* (2,182)*, Diatoms* (1,019), *Gelatinous Zooplankton* (8,024), *Other Invert Larvae* (1,445), *Protists* (3,808), and *Trichodesmium* (3,419).



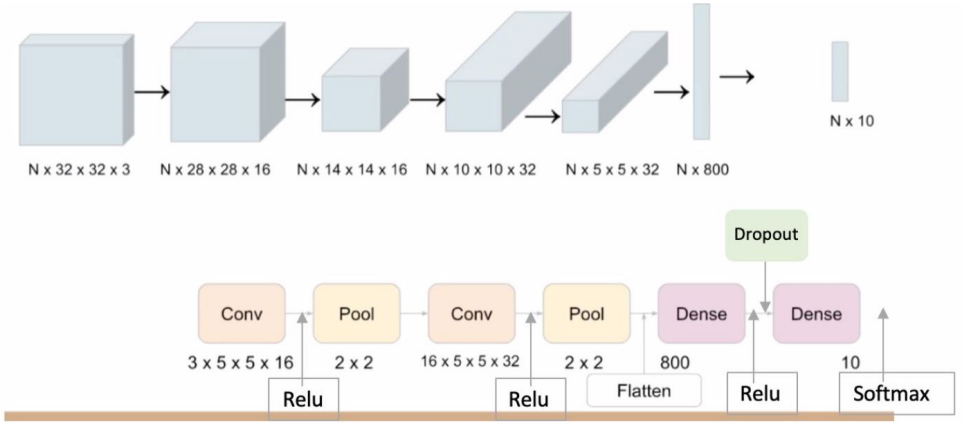
Algorithm Development

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. Additionally, as you can see from the class sizes above, we have an imbalanced dataset with a wide range of images in each class. This poses an issue because the classes are not represented equally. The training model will learn on a skewed distribution of data, causing a misleading classification accuracy biased towards classes with a higher number of images. To tackle this issue, we will implement the *class_weight* parameter during our model training so it focuses more on under-represented classes and adjusts the weights in the loss function accordingly. This function increases the weights for under-represented classes, and decreases the weights for over-represented classes, giving each class a balanced platform in the model and removing the bias influence towards larger class sizes that would have occurred otherwise. The weights are determined by calculating the ratio of each smaller class size to the largest class size; so if one of our classes is three-times smaller than our largest class size, it will get three-times the weight increase in our training model.

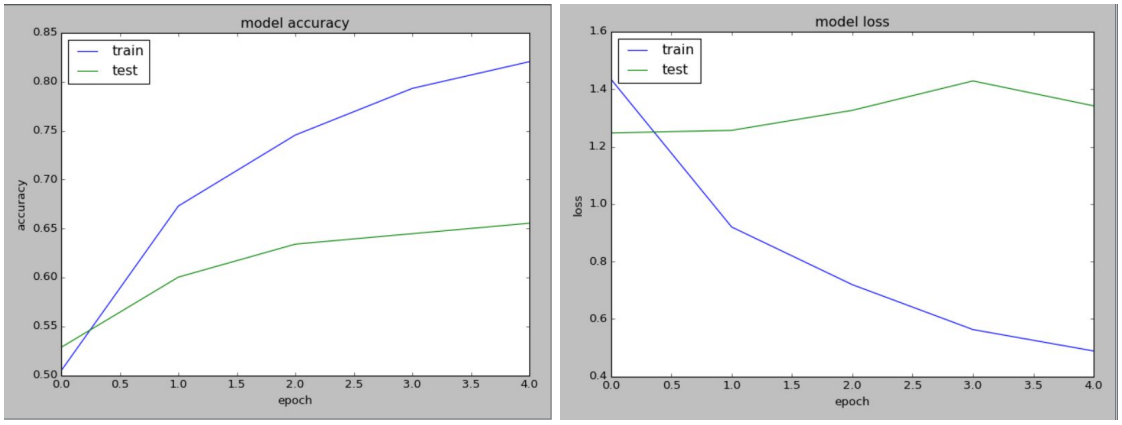
Next, we used a method from the *ImageDataGenerator* class in Keras to assign our 8 class labels to their corresponding images in the training, testing, and validation sets. Additionally, we resized our images so they were all consistent at 64 x 64 pixels, and rescaled our images by simply dividing each pixel by 255 (maximum value in pixel color range) so they were normally distributed on a [0,1] interval. Neural networks work most efficiently when the input data contains point values between 0 and 1, aiding in a more rapid convergence towards optimization. After completing the steps above, our final image input shape is 64 x 64 x 3. Finally, we standardized our images using feature wise centering, which essentially

centers the mean at zero and subtracts the mean to normalize the image so the algorithm isn't biased by extreme or outlier images.

Moving into the network development section of our project, we will be using Keras to implement our CNN in Python. Before we refined our model, we began with a simple network structure as a benchmark to build off off. We had a network following the structure below, with two convolutional layers, each with a kernel size of dimension 5 x 5. Our total output filters for the first layer was 16, and the second layer contained 32. We used the *adam* optimizer, the *relu* activation function for each layer, and had max pooling layers of dimension 2 x 2. We flattened the 2-Dimensional array into a 1-Dimensional array to be processed by the dense layer with a length of 800 (32 x 5 x 5) and a *relu* activation function. Our final dense output layer was a 1-Dimensional array of length 8 because we have 8 output classes, and contained a *softmax* activation function. This simple, base network resulted in a testing accuracy of about 65% on the validation set, and you can see the loss and accuracy results in the graphs below.



https://www.udemy.com/deep-learning-convolutional-neural-networks-theano-tensorflow/learn/v4/overview



To begin refining our model, we tested it on our validation set using 5 epochs and a batch size of 200. Since plankton are small and microscopic organisms, our images are very detailed and the key differences can sometimes be minute. Including additional convolutional layers adds more depth to a neural network and aids in deciphering more complex patterns. Therefore, we began by experimenting with various layers, and found that our original two layer network actually performed the best with

about 65% accuracy. Each additional layer we added decreased our accuracy by about 2%. Next, in determining the appropriate kernel size, we wanted to choose dimensions that were small enough to pick up on detailed patterns in our images, but large enough to increase efficiency and not leave out any key information. We, again, found that our original 5 x 5 kernel size resulted in the highest accuracy of around 65%; with a 3 x 3 kernel being too small and inefficient and a 8 x 8 kernel being too large to pick up on certain patterns with about 5% less accuracy. Next, we decided to add a dropout layer and determine the best dropout rate that would force our model to learn more efficiently, while not removing significant information. We found that a dropout rate of 10% and 20% resulted in the same accuracy of about 65%, and each additional 10% dropout after that point would begin decreasing the accuracy score. Lastly, I tried various optimizers in my model and found that the *Adam, Adadelta,* and *Nadam* optimizers all resulted in about the same high accuracy of 65%.

Next, we 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. After this we 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%. Finally, we 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 experimented with the best performing parameters from our algorithm development, we moved forward to experiment with different environmental setups. 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%, while using 50 epochs provided a train accuracy of 94.71% and validation accuracy of 75.1%. We used both of these values for our final model.

Activation functions:

1. Softmax: Train- Acc: 90.68%, Loss: .2345, Val- Acc: 68.03%, Loss: 1.8731
2. Elu: Train- Acc: 7.9%, Loss: 8.1212, Val- Acc: 6.57%, Loss: 8.7970
3. Sigmoid: Train- Acc: 89.59%, Loss: .2622, Val- Acc: 68.83%, Loss: 1.8125
4. Relu: Train- Acc: 74.21%, Loss: 1.0940, Val- Acc: 63.36%, Loss: 2.7959
5. Softplus: Train- Acc: 89.52%, Loss: .3610, Val- Acc: 68.77%, Loss: 1.7412
6. Tanh: Train- Acc: 17.17%, Loss: 6.1768, Val- Acc: 14.43%, Loss: 9.0834
7. Hard Sigmoid: Train- Acc: 70.47%, Loss: 1.2093, Val- Acc: 62.26%, Loss: 2.0036
8. Linear: Train- Acc: 12.21%, Loss: 4.0717, Val- Acc: 10.36%, Loss: 2.7163

Loss Functions:

1. Mean Squared Error: loss: 0.0171 - acc: 0.9094 - val_loss: 0.0515 - val_acc: 0.7530
2. Categorical Hinge: loss: 0.3448 - acc: 0.7944 - val_loss: 0.5640 - val_acc: 0.7062
3. Logcosh: loss: 0.0099 - acc: 0.8827 - val_loss: 0.0266 - val_acc: 0.7152
4. Categorical Crossentropy: acc: 0.9327 - val_loss: 1.3486 - val_acc: 0.7506

5. Kullback Leibler Divergence: loss: -1.1329e-05 - acc: 0.1218 - val_loss: -1.1282e-05 - val_acc: 0.1124
6. Poisson: loss: 0.1574 - acc: 0.9037 - val_loss: 0.2766 - val_acc: 0.7366
7. Cosine Proximity: loss: -0.9064 - acc: 0.8920 - val_loss: -0.7575 - val_acc: 0.7338

Resizing Images:

1. 32x32: Train- Acc: 75.86%, Loss: .6229, Val- Acc: 62.38%, Loss:1.4873
2. 64x64: Train- Acc: 88.35%, Loss: .2821, Val- Acc: 70.14%, Loss: 1.6505
3. 96x96: Train- Acc: 93.05%, Loss:.1751, Val- Acc: 67.08%, Loss: 2.0698
4. 128x128: Train- Acc: 95.78%, Loss: .1076, Val- Acc: 67.67%, Loss: 2.799
5. 192x192: Train- Acc: 97.74%, Loss: .0809, Val- Acc: 67.67%, Loss 2.9293
6. 258x258: Train- Acc: 97.75%, Loss: .0576, Val- Acc: 64.71%, Loss: 3.4141
7.

Experimental Setup

Mini-Batch (using rmsprop and 3 layers):
20- loss: 0.0294 - acc: 0.8556 - val_loss: 0.0597 - val_acc: 0.7207
60- loss: 0.0209 - acc: 0.8916 - val_loss: 0.0548 - val_acc: 0.7368
100- loss: 0.0206 - acc: 0.8900 - val_loss: 0.0536 - val_acc: 0.7405
150- loss: 0.0228 - acc: 0.8770 - val_loss: 0.0581 - val_acc: 0.7091
200- loss: 0.0248 - acc: 0.8641 - val_loss: 0.0515 - val_acc: 0.7368
500- loss: 0.0412 - acc: 0.7611 - val_loss: 0.0586 - val_acc: 0.6877

Epochs:
5- loss: 0.0493 - acc: 0.7158 - val_loss: 0.0554 - val_acc: 0.6993
20- loss: 0.0218 - acc: 0.8841 - val_loss: 0.0541 - val_acc: 0.7412
50- loss: 0.0113 - acc: 0.9471 - val_loss: 0.0547 - val_acc: 0.7510
100- loss: 0.0102 - acc: 0.9543 - val_loss: 0.0573 - val_acc: 0.7352
500 (with 200 batch and early stopping)- loss: 0.0170 - acc: 0.9123 - val_loss: 0.0566 - val_acc: 0.7310
Learning Rate:

Results

	Loss	**Accuracy**
Train	0.0042	98%
Validation	0.0579	74%

Conclusion

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 counteract 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.