Alexa Giftopoulos
December 10, 2018
Machine Learning II

Predicting Plankton Classification with Convolutional Neural Networks

Individual Report

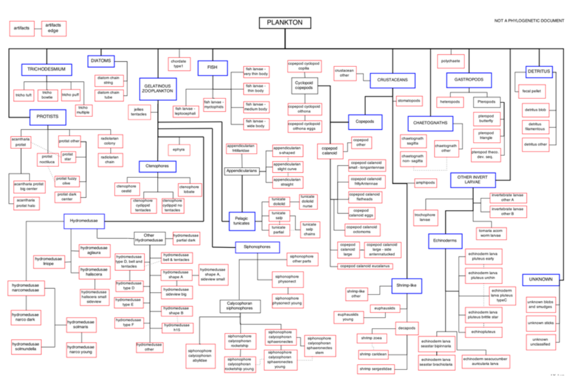
Introduction

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).

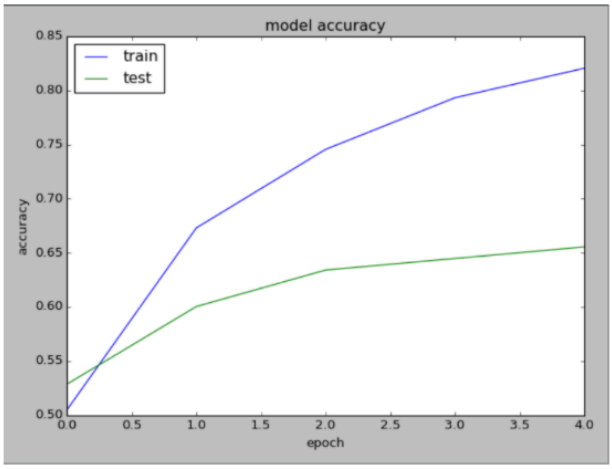


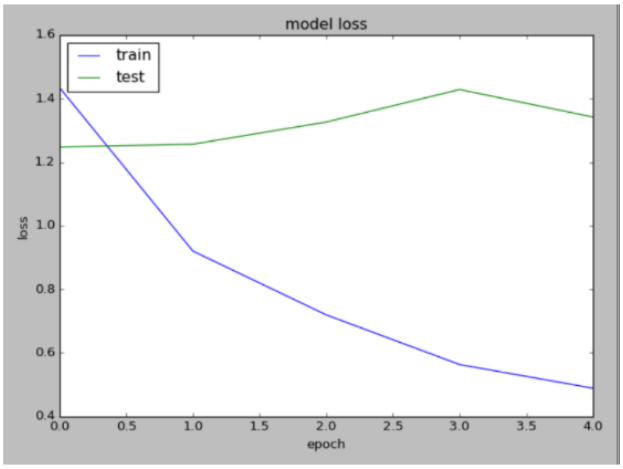
Individual Work and Results

Each team member contributed equally in the research regarding neural network structure and methodology and was engaged in all aspects of this final project via the code, report, presentation, etc. However, we decided to split up majority of the work when it came time to implement our findings. Iliana focused on implementing the theory via code, I focused on implementing the theory via the report, and Brian took the lead on consolidating the presentation. Although we focused mainly on these separate areas, we still engaged in each aspect of the final to make sure we all understood and optimized our learning from this final project. You will see below that although I took the lead on the report, I spent time working through the code and contributed in building out our *algorithm development* section through experimental fine tuning of the model. The portion of my work on the code is explained in the following paragraphs.

After we split the data into training, testing, and validation sets, resampled our imbalanced dataset, resized and rescaled our images, we decided to split up the work in refining our base neural network model and fine tune the parameters. I decided to focus on manipulating the network layers, kernel sizes, dropout layers/rate, optimizers, and batch sizing. Below is a description of my findings.

I started with a base model containing two convolutional layers, each with a kernel size of dimension 5 x 5. The total output filters for the first layer was 16, and the second layer contained 32. I used the *adam* optimizer, the *relu* activation function for each layer, and had max pooling layers of dimension 2 x 2. I 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. The 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 I made below.





I originally was working on *Tensorboard*, however since we used an *ImageDataGenerator* class from Keras, it was causing some problems for me as the graphs would display the accuracy and loss values that were being generated along with additional pseudo lines in the graph that contained no values. Therefore, I decided to implement basic plots in the code to display an accurate depiction of the values I was actually generating. A snippet of the code is displayed below along with the source I referenced it from.

```
#list all data in history
# https://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/
print(history.history.keys())

# summarize history for accuracy
plt.figure(1)
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()

# summarize history for loss
plt.figure(2)
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

https://machinelearningmastery.com/display-deep-learning-model-training-history-in-keras/

To begin refining the model, I 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, I began by experimenting with various layers, and found that the 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, I 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. Again, I 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 an 8 x 8 kernel being too large to pick up on certain patterns with about 5% less accuracy. Next, I 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. I 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%.

Summary and Conclusion

Overall, from my experimentation above, it seems like my original basic model sufficed in providing me about the highest accuracy I could obtain thus far. My portion of the model aspects I was fine tuning did not seem to impact the model accuracy much, and I conveniently chose the most ideal parameters in the original base model I constructed. Iliana was working on fine tuning the other remaining parameters, and her model also included standardized images. It seems she was able to reach higher accuracies than me with her parameters, so we will combine our top findings to create our final, highest accuracy model. In my complete code that I used to run the experiments above, the percentage of code I sourced from the internet was 30% (50 – 35 / 50 * 100).