Machine Learning II- Final Project Report

Iliana Maifeld-Carucci, Alexa Giftopoulos, Bryan Egan

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.

2 Data Background

Our dataset was obtained from Kaggle and includes colored images of plankton captured from Oregon State Universitys Hatfield Marine Science Center. We were provided with a file containing folders of images for each class. Looking at Figure 1 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).

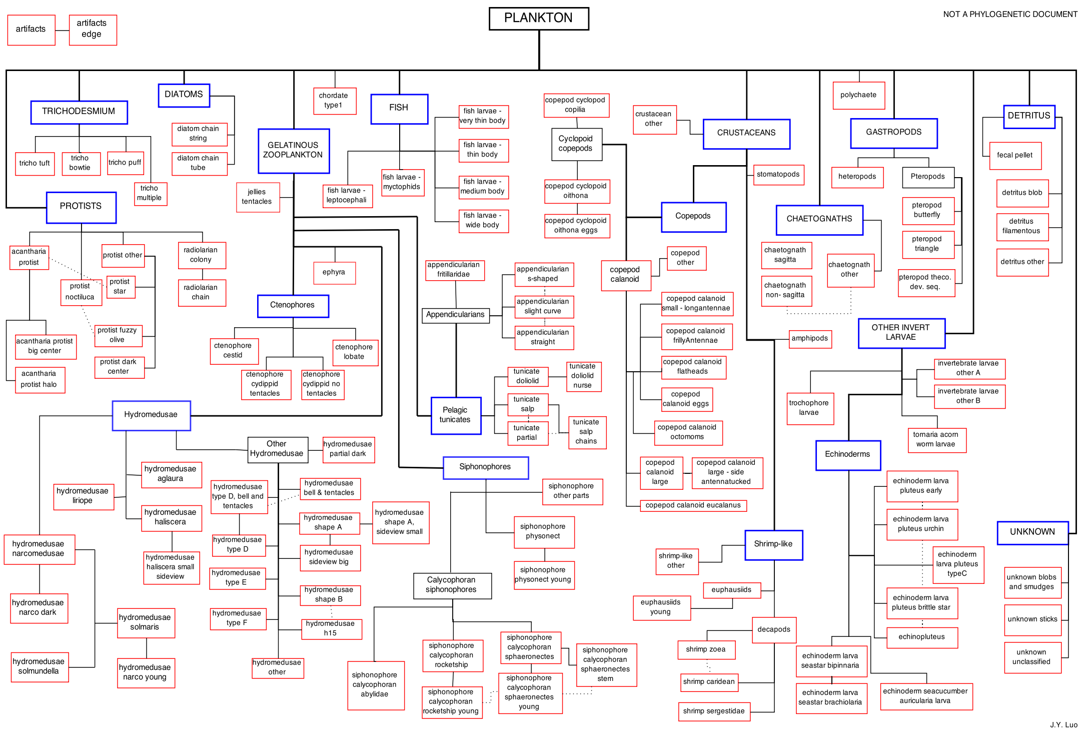


Figure 1

Plankton Classification

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).

3 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 allows you to specify the weights for each class so you can increase 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 the *class weight* function from *sklearn* which adjusts the weight in proportion to the number of samples in the class.

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 while preventing overflow. After completing the steps above, our final image input shape is 64 x 64 x 3. This was all the preprocessing we did to run a bare bones model to get a baseline accuracy. To finish preprocessing, we applied mean normalization to our images using feature-wise centering, which essentially centers the mean at zero and subtracts the mean across features to normalize the image so the algorithm isnt biased by extreme or outlier images. Doing this helps to minimize the variance which makes it easier for the algorithm to solve for our coefficients. We proceeded to standardize our data using feature-wise standardization which makes the standard deviation one and helps our algorithm converge. Additionally, we investigated the implementation of ZCA whitening which helps sharpen the edges of images and can assist an algorithm in learning the important features.

Moving into the network development section of our project, we used Keras to implement our convolutional neural network (CNN) in Python. As mentioned abouve, before we refined our model we began with a simple network structure as a benchmark to build on. We had a network following the structure below (Figure 2), 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.

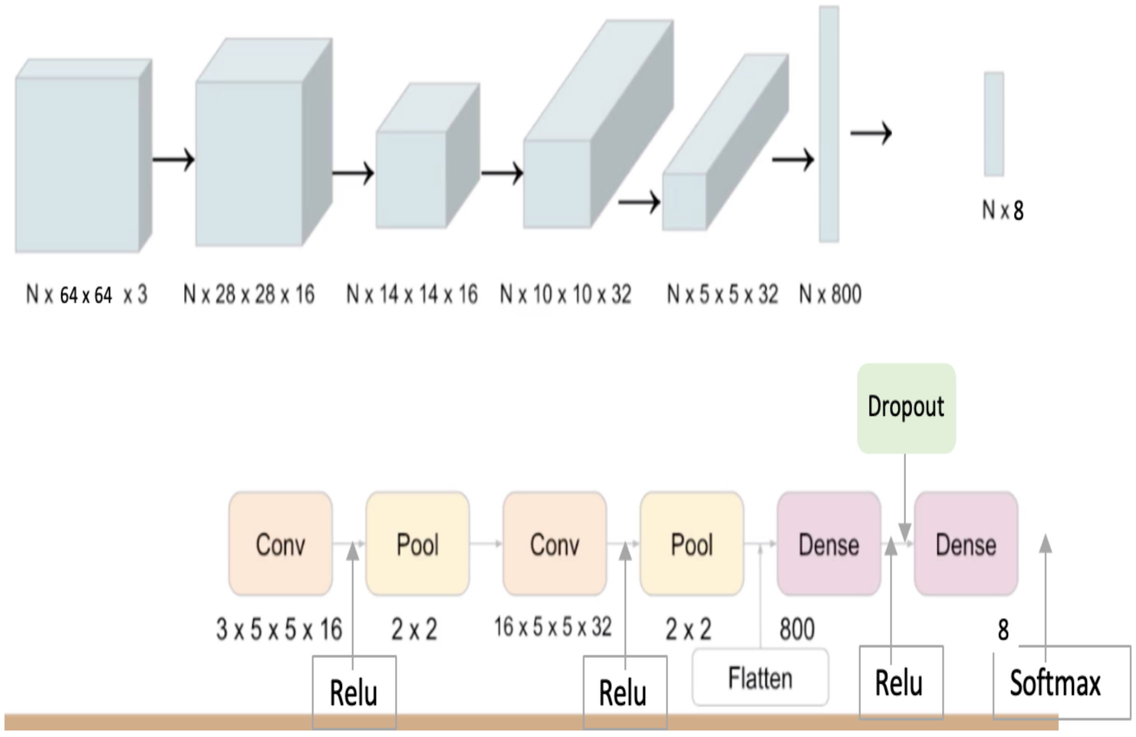


Figure 2

Network Architecture

To begin refining our model, we tested it on our validation set using 5 epochs and a batch size of 200 and then we investigated different image resizing dimensions to see if that would impact our model. 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. 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. Thus, it was important that we investigated various kernel sizes. 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. Additionally, we tried various optimizers, as well as loss and activation functions in our model. Finally, we tested early stopping which helps prevent the model from overfitting by stopping the training as soon as the loss starts increasing again.

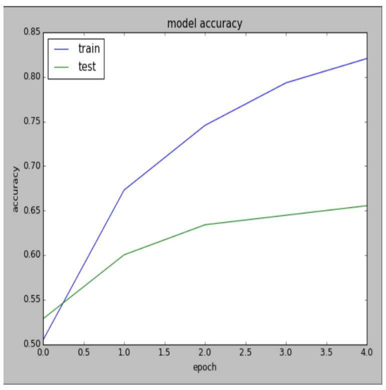
4 Experimental Setup

Once we determined which hyperparameters yielded the highest training and validation accuracies and which combination gave us the best performance, we used those selected hyperparameters to explore different inputs for our experimental set up. Specifically, we investigated how changing the size of the mini-batches affected our accuracies, as well as the effect differing numbers of epochs had on our performance. The best performing number of epochs and mini-batches was used in conjunction with the best combination of hyperparameters for our final model which we applied to our test set.

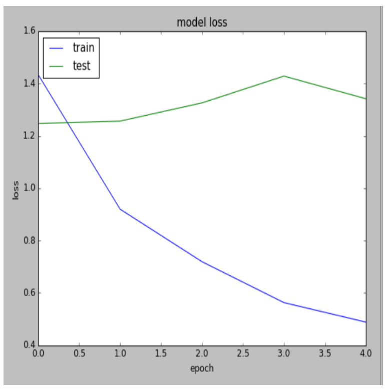
5 Results

5.1 Baseline Model

As we stated above, the first thing we did was create a baseline model with which to compare our other results. In this 2 layer model, we found a training accuracy of 83% and a testing accuracy of about 65% on the validation set, and you can see the loss and accuracy results in the graphs below (Figure 3).



(a)

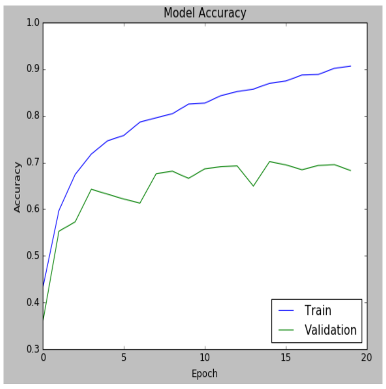


(b)

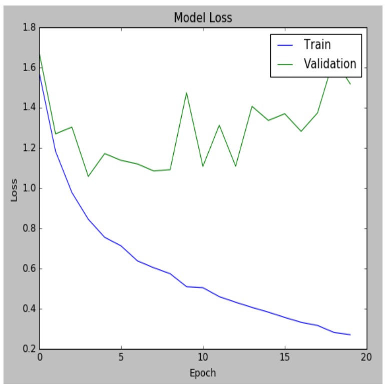
Figure 3: Accuracy and Loss with Base Model

5.2 Preprocessing

From here we implemented our three different preprocessing steps: mean normalization (Figure 4), standardization (Figure 5), and ZCA whitening (Figure 6). We found that adding mean normalization and subsequently standardization both increased our validation accuracy (68.3% and 70.6% respectively), while ZCA whitening decreased our accuracy (69.5%) so we decided drop ZCA whitening and only move forward with our normalization and standardization.

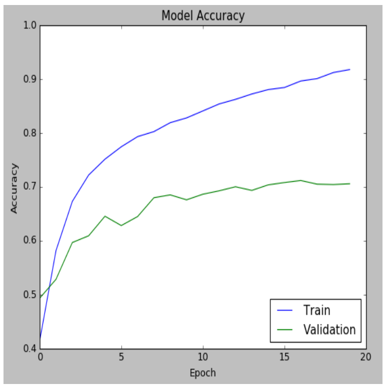


(a)

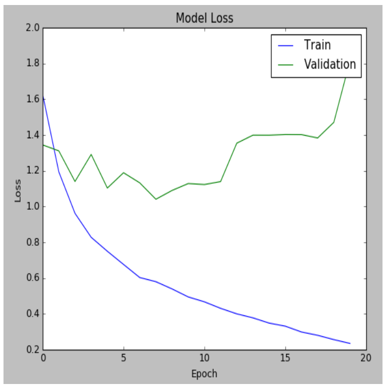


(b)

Figure 4: Accuracy and Loss with Mean Normalization

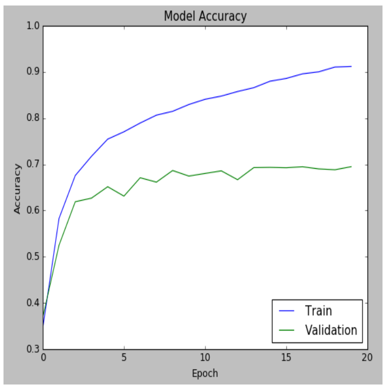


(a)

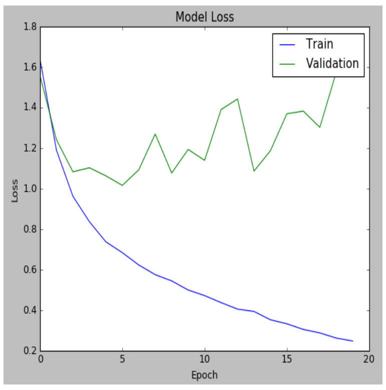


(b)

Figure 5: Accuracy and Loss with Standardization



(a)

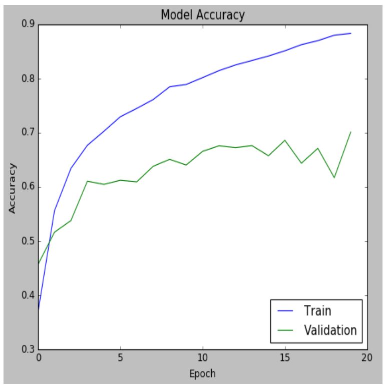


(b)

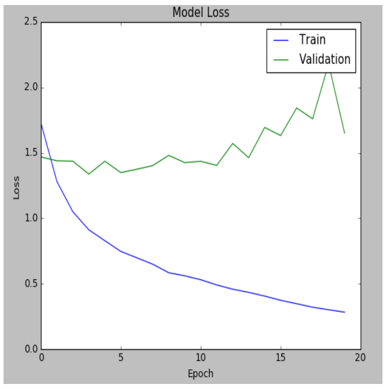
Figure 6: Accuracy and Loss with ZCA Whitening

5.3 Error Weighting

Following our preprocessing, we added error weighting to deal with our class unbalance in our data. Putting this in resulted in a training accuracy of 88.35% and a validation accuracy of 70.14% (Figure 7). 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 7: Accuracy and Loss with Error Weighting

5.4 Algorithm Development

Now that we dealt with both preprocessing and our class unbalance, we explored various functions and values for all of the hyperparameters for our model including image size, number of layers, kernel size, dropout, opimizer, activation, and loss functions, and early stopping. We will go over our results from each of them independently.

5.4.1 Image Size

When we used various sizes for our images, we found that using an image height of 64 pixels and an image width of 64 pixels yielded the highest validation accuracy of 70.1%. However we tried a variety of dimensions as can be seen in the table below.

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
128	128	95.78	.1076	67.67	2.799
192	192	97.74	.0809	67.67	2.9293
258	258	97.75	.0576	64.71	3.4141

5.4.2 Number of Layers

Next we experimented with various layers and found that our original two layer network actually performed the best with about 65% validation accuracy. Each additional layer we added decreased our accuracy by about 2% as can be seen in the table below, so we went forward with two layers.

Number of Layers	*Validation accuracy (%)*
2	65
3	62
4	61

5.4.3 Kernel Size

We tackled kernel size next, finding 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.

Kernel Size	*Validation accuracy (%)*
$3x3$	62
$5x5$	65
$8x8$	61

5.4.4 Dropout

Because we were seeing significant overfitting, we decided to experiment with using a dropout layer. 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.

Dropout Proportion	*Validation accuracy (%)*
.1	65
.2	65
.3	64
.4	64

5.4.5 Activation Function

While we were pretty sure we would use the softmax activation function due to it summing the output probabilities from the fully connected layer to one, which is commonly used for CNNs, we still tried several other functions who's performance is outlined in the table below. It's also good to note that while softmax didn't yield our highest accuracy, it did come very close with a validation accuracy of 68%, so it's still a great choice from a performance perspective.

Activation Function	*Train accuracy*	*Train loss*	*Validation accuracy*	*Validation loss*
Softmax	90.68	.2345	68.03	1.8731
Elu	7.9	8.1212	6.57	8.797
Softplus	89.52	.3610	68.77	1.7412
ReLU	74.21	1.094	63.36	2.7959
Tanh	17.17	6.1768	14.43	9.0834
Sigmoid	89.59	.2622	68.83	1.8125
Hard Sigmoid	70.47	1.2093	62.26	2.0036
Linear	12.21	4.0717	10.36	2.7163

5.4.6 Optimizer Function

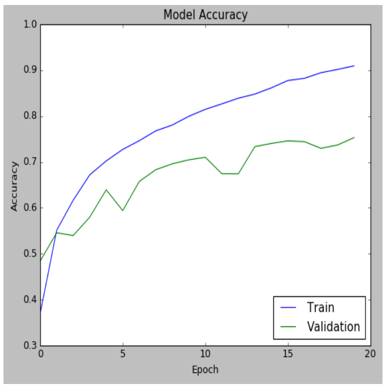
We also investigated the performance of several optimizer functions and found that the Adam and Adadelta optimizers both resulted in about the same high accuracy of 65%. Despite this, when we were experimenting with combining this result with our other top performers, we found that RMSProp performed really well in conjunction with the other hyperparameters, so we decided to stick with it.

Optimizer Function	*Validation accuracy* (%)
SGD	65
Adam	65
Adagrad	64
RMSProp	64

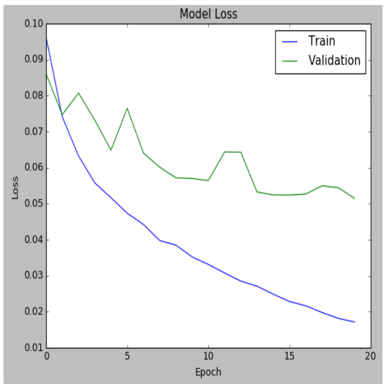
5.4.7 Loss Function

The last function we looked at was using different loss functions. This ended up being one of the parameters that had the biggest impact on the performance of our model and we found that using the mean squared error provided the highest validation accuracy of 75.3% while significantly lowering our loss (Figure 8). The impact of each of the functions can be seen in the table below.

$Loss\,Function$	$Train\,accuracy$	$Train\,loss$	$Validation\,accuracy$	$Validation\,loss$
$Mean\,Squared\,Error$	90.94	.0171	75.3	.0515
$Categorical\,Hinge$	79.44	.3448	70.62	.564
$Logcosh$	88.27	.0099	71.52	.0266
$Categorical\,Cross-entropy$	93.27	.5837	75.06	1.3486
$Kullback\,Leibler\,Divergence$	12.18	.00001	11.24	.00001
$Poisson$	90.37	.1574	73.66	.2766
$Cosine\,Proximity$	89.2	.9064	73.38	.7575



(a)



(b)

Figure 8: Accuracy and Loss using MSE as Loss Function

5.5 Early Stopping

The last thing we tried before moving on to playing with our experimental setup was early stopping in order to minimize our overfitting. We first ran this using only 20 epochs and got a validation accuracy of 60.18%. However, early stopping is most useful when a large number of epochs is used, so we tried it again with 500 epochs, and we still only got 73.1% and we used a high patience level. We concluded that early stopping didn't help much from an accuracy or overfitting side, so we decided to omit it from our final model.

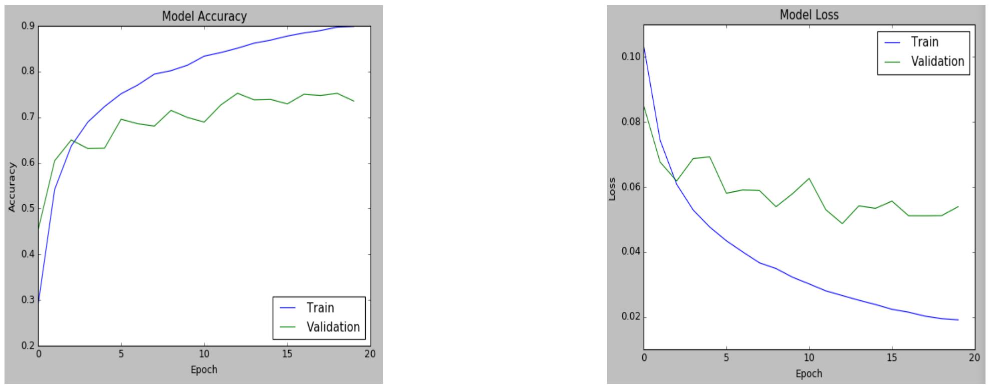
5.6 Experimental Setup

Once we did all of the testing of the algorithm hyperparameters we tried a few different combinations of the best performing hyperparameters and took those forward into testing our experimental set up with batch size and number of epochs. The hyperparameters we chose were an image size of 64x64 pixels, 2 convolutional layers with 5x5 kernel sizes and a 20% dropout layer, a softmax activation function, RMSProp optimizer, and using MSE for our loss function.

5.6.1 Mini-Batches

We tried a variety of different mini-batch sizes and found that using a batch size of 100 worked best, giving us a validation accuracy of 74.05% (Figure 9). You can see how other mini-batch sizes performed below.

Batch Size	*Train accuracy*	*Train loss*	*Validation accuracy*	*Validation loss*
20	85.56	.0294	72.07	.0597
60	89.16	.0209	73.68	.0548
100	89	.0206	74.05	.0536
150	87.70	.0228	70.91	.0515
200	86.41	.0248	73.68	.0515
500	76.11	.0412	68.77	.0586



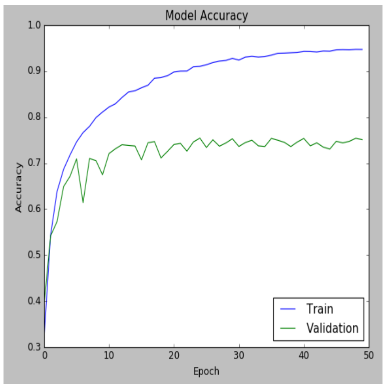
(a) (b)

Figure 9: Accuracy and Loss using Mini-Batch Size of 100

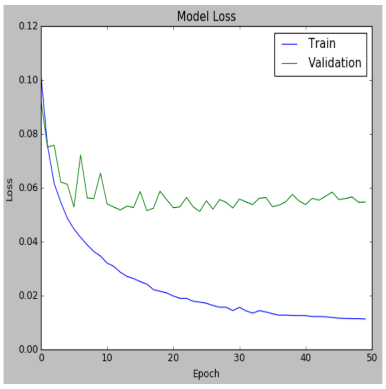
5.6.2 Epochs

To the best of our ability taking into consideration runtime, we also experimented with different epoch lengths. Ultimately, using 50 epochs worked the best, providing a 75.1% validation accuracy (Figure 10).

Number of Epochs	*Train accuracy*	*Train loss*	*Validation accuracy*	*Validation loss*
5	71.58	.0493	69.93	.0554
20	88.41	.0218	74.12	.0541
50	94.71	.0113	75.1	.0547
100	95.43	.0102	73.52	.0573
500 (*w/ early stopping*)	91.23	.0170	73.1	.0566



(a)

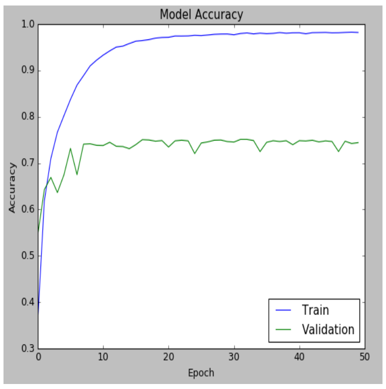


(b)

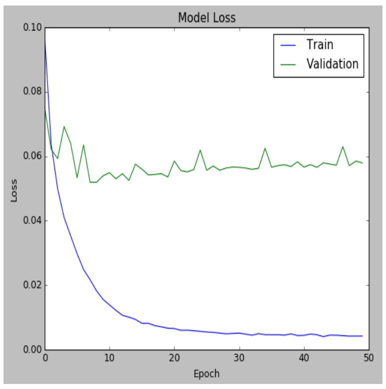
Figure 10: Accuracy and Loss using 50 Epochs

6 Summary and Conclusion

Ultimately we ended up using an image size of 64x64 pixels, 2 convolutional layers with 5x5 kernel sizes and a 20% dropout layer, a softmax activation function, RMSProp optimizer, and using MSE for our loss function with a mini-batch size of 100 and using 50 epochs. You can see from Figure 11 that through hyperparameter tuning and other methods, we were able to increase the accuracy of our base model by about 10%. 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 like data augmentation or boosting and other ways to minimize the overfitting we continually experienced.



(a)



(b)

Figure 11: Accuracy and Loss of our Final Model

7 References

Image Generators

Moving Data Function

Data Preprocessing and Augmentation

Sectioning out Data

Normalization and Augmentation Explanation

PCA Whitening vs. ZCA Whitening

Unbalanced Data

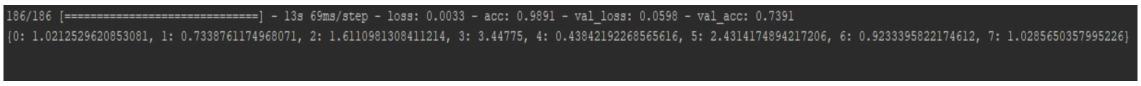
Early Stopping

Preprocessing Steps

Keras Image Generator

Network Architecture Diagram

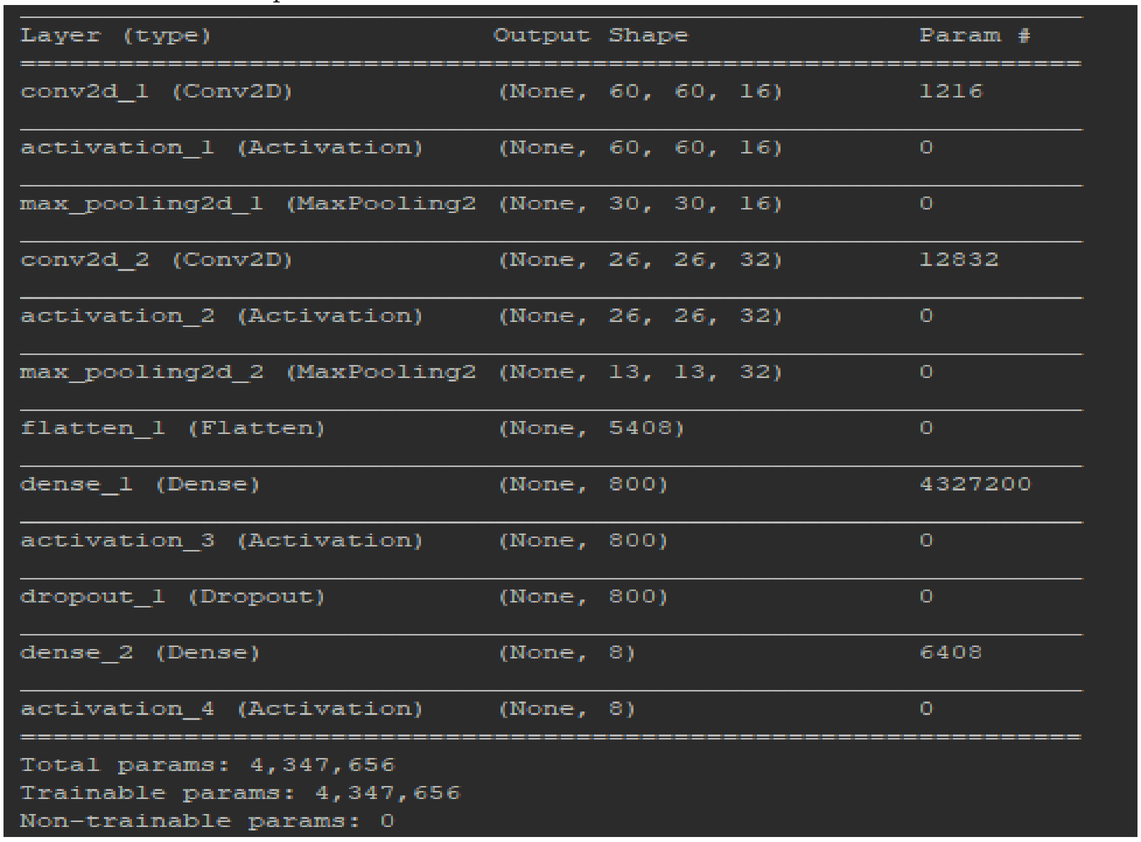
8 Code Output Appendix



```
186/186 [==============================] - 13s 69ms/step - loss: 0.0033 - acc: 0.9891 - val_loss: 0.0598 - val_acc: 0.7391
{0: 1.0212529620853081, 1: 0.7338761174968071, 2: 1.6110981308411214, 3: 3.44775, 4: 0.43842192268565616, 5: 2.4314174894217206, 6: 0.9233395822174612, 7: 1.0285650357995226}
```

Figure 12

Final output accuracies



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 60, 60, 16)        1216
_________________________________________________________________
activation_1 (Activation)    (None, 60, 60, 16)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 30, 30, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 26, 26, 32)        12832
_________________________________________________________________
activation_2 (Activation)    (None, 26, 26, 32)        0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 5408)              0
_________________________________________________________________
dense_1 (Dense)              (None, 800)               4327200
_________________________________________________________________
activation_3 (Activation)    (None, 800)               0
_________________________________________________________________
dropout_1 (Dropout)          (None, 800)               0
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 6408
_________________________________________________________________
activation_4 (Activation)    (None, 8)                 0
=================================================================
Total params: 4,347,656
Trainable params: 4,347,656
Non-trainable params: 0
```

Figure 13

Parmeters Per Layer