Problem Statement:

Intensive whaling(hunting of whales) has unfortunately severely damaged whale populations. In addition to this, recovering whale populations now have difficulty competing for food with the industrial fishing industry and adapting to the warming oceans. Scientists hope to combat this problem and help in whale conservation efforts using photo surveillance to monitor ocean activity. They use whale tails to identify the whale species and log essentially data about their movements and whale pod dynamics. The objective of this project is to be able to classify a whale by an image of its tail. This classification is usually done manually but an algorithm that automatically classifies whales with high accuracy would aid drastically. A good algorithm would hopefully recognize patterns not even observed by humans and be able to classify even better than the scientists.

Data Description/Cleaning/Pre-Processing

I received my data from Kaggle. The data consists of a folder of thousands of jpeg images of whale tails. These images have been identified by researchers and have been labeled with a whale Id. Images that do not have a clear whale ID are simply labeled as new whale. There was a cvs file that contains the file name for each image with the corresponding whale label. The first order of business was to convert the cvs file into a pandas data frame. To feed the data into a model, it needs to be pre-processed into the proper format. I prepared the data by creating two methods, one that would create an array for the images and one that created an array for the corresponding labels. I converted the all the images into an 4D array of pixel values (each image is a 3D array of pixel values) and scaled the images so that pixel values ranged from -1 to 1. I then converted the corresponding labels to a one-hot encoded 2D array. This means for a particular sample/image, the correct label will be “1” and the rest of of the labels in the array for that sample/image will be “0”.

Model:

Pre-Planning Strategy:

I created a Convolutional Neural Network(CNN) to most accurately predict the label of each whale tail image. The goal was to design the best network possible for this problem for recognizing features and classifying the whale images as accurately as possible. The general strategy/approach I used to design the best network was to keep adding convolutional network layers until the model overfit. To check for overfitting, I checked whether the training data accuracy greatly exceeded the validation data accuracy at each epoch or until the validation data accuracy no longer increased with each epoch. Another way

was to check whether the validation loss was increasing while the training loss was decreasing which would again indicate the model is learning too strongly on the training data. The goal was to increase validation accuracy and decrease validation loss with each epoch until they until the model could no longer be improved. There needed to be a delicate balance of model accuracy and computational efficiency. In the early stages of the network, I kept the feature space wide and in the later stages of the network towards the end I made the feature space deeper and narrower. It was important to to me that the number of channels/filters was relatively low in the beginning before increasing in later layer of the model. This would allow for the model to better identify low level features. Later I would increase the number of channels/filters so those low level features could form into shapes that were more complex which would aid in differentiating between classes. I also wanted to keep the filter sizes low so the model could collect as much local information on the images as possible, especially in the early stages of the model. In later stages, I could potentially increase filter size to more high level information.

The Big Problem: Computational Issues and Class Imbalance

It became clear to me early on that this project required a computer with extremely high computational power that my computer did not possess. This project required running a highly complex model through hundreds of epochs with all if not at least the vast majority of the 25361 samples. It also required a large batch size to minimize training time. This was impossible to do on my computer. An epoch would take far too long to train(many hours to days) or the debugger would crash if the sample size, batch size, or if complexity of the model was too high for my computer which it was being run on. Thus, I knew I needed to decrease sample size, batch size, and/or model complexity to the point where I can run multiple epochs at a decent amount of time on my computer.

Overfitting due to the sample size was first major issue. Unfortunately I was extremely restricted due to the computational limits of my computer. Since using the full 25361 images was not possible on my computer, I used 6300 images total. I decided to use 20% (1260 samples) of the data as the validation set and 80% (5040 samples) as the the training set. This was about the maximum my computer could train on without taking enormous amounts of time, although it still required a bit of time. In general, two major problems that lead to a high chance of overfitting are not enough samples overall and too few samples per class. Through heavy experimentation, I discovered that I needed to work with the majority, if not all, of the 25361 samples in the dataset to successfully prevent overfitting. I needed to have more training data from which the model could learn. However, this was not possible due to the computational limitations of my computer. Ultimately, I could not give the model enough samples to not overfit on

the training data. In addition to not training on enough samples overall, there were not enough samples in each class for it to not overfit. I discovered the lack of samples caused overfitting no matter what model I was training on. Overfitting was also occurring very early on in the training process.

This project was particularly challenging because there were too few samples for majority of the classes due to the fact that there were over 5005 classes to classify the images on. There was also a tremendous class imbalance with one class in particular. For example**,** while there were 2073 classes that only had a single image from which it could learn, the class 'new whale' had over 9664 images. Because of this, the model was already extremely prone to overfitting. Oversampling was not an option due to increasing the sample size of 5005 classes would lead to far more samples which were not trainable due to my computers limitations. Under sampling was also not an option due to most classes having a single image.

A problem like this in particular in which there are so many classes and so few samples for each class, would require a particularly large number of epochs before it could start truly learning. When working with a project this prone to overfitting, you need to work with as many images as possible. 25361 images may have been an adequate number but 6300 was certainly not. There are already so many classes and so few samples of whales in each class even when working with the full sample size of 25361 images; as such, working with only 6300 images made it more difficult to learn. The 6300 sample had about the same class percentages as the original dataset so stratified sampling was not necessary. I needed the computational power to train on far more samples, which would allow the model to learn for a much higher number of epochs (learn longer). This project ideally required training of over a hundred epochs.

After thorough experimentation with the most basic model (one layer) and onwards, I discovered that every model was going to overfit immediately after the first epoch or second epoch. As I added layers, it consistently stopped learning after the first or second epoch. I knew it was overfitting/not learning because of how the validation loss was moving. The consistent increase in the validation loss immediately after the first or second epoch indicated it was overfitting right away with any model that I tried. The validation loss was increasing while the training loss was decreasing which means it was overfitting to the training data. It was learning too strongly on the training data because there are too few samples per class and too many classes. This was causing it to overfit after that second epoch and not learn properly. Thus, it fit worse on the unseen (validation) data because it is learning too strongly the data it sees (the training data). The validation accuracy was also not increasing after each epoch, but rather staying the same due to every model over-learning on the training data. Thus, the

problem of not learning past the first couple of epochs was unavoidable without the need for much stronger computational power that would allow me to the train on majority or if not then the entire image dataset.

This posed a huge problem for me and I decided what I was going to do was find the model that gave me the lowest initial validation loss values. Since the knew the validation loss was going to increase after the first or second epoch regardless, I decided the I would try my best to reduce the validation loss of the first few epochs as much as I possibly could and that would tell me the quality of my model. Thus, my criteria and goal for determining the quality of my model was to reduce the initial validation loss as much as possible due to the fact that not learning/overfitting was unavoidable without stronger computation power. This would give me a model that I was confident would scale well if given the correct computational power and thus the correct number of epochs, sample size, and batch size to work with.

Process:

I experimented with different structures inspired by various pertained models such as densenet121, resnet50, and VGG16/VGG19. The structure that ultimately gave me the lowest validation loss and maximized the validation accuracy was inspired by the VGG16 structure, specifically the convolution-convolution-pooling pattern. I added a 2 by 2 max pooling layer after every 2 convolutional layers, with a 1 by 1 stride, as is done by the VGG16 model. After some experimentation, I settled on using a 32-32 approach for the number of channels/filters to use as I progressed through the layers, with 64-64 for the last four layers. I used a filter size of 3 by 3 throughout the network. I experimented with gradually increasing the filter size, but that proved to only increase the validation loss and thus worsen the model so I decided to keep the filter size 3 by 3 throughout all the layers. I experimented with starting with 64 channels initially and going up to 128 as the layers progressed but that a did not prove to be as effective increasing validation accuracy and starting with 32 channels and gradually moving up to 64.

After each convolutional layer, I added an relu activation layer to handle non-linearity. In total, I added 11 convolutional layers and 6 max pooling layers to get to model with the lowest validation loss.

After each convolutional Layer, I set padding to ‘same’ so that the dimensions of the picture would not change after every convolutional layer. This would pad the image evenly left and right and make the output size the same as the input size.This allowed me to add more convolutions without shrinking the size of the image. After all the CNN layers, I flattened the output so that it could

be input into a fully connected layer. I experimented with the number of fully connected layers and the number of nodes in each. I ultimately settled on one fully connected layer with 500 nodes as it performed the best on the validation loss. I added a dropout of about 80% after the fully connected layer to decrease how much the model overfit. The final layer was the output layer with a softmax activation function that will assign a probability of that image being in each class. I then compiled the model using categorical cross entropy loss function and the adam optimizer. The adam optimizer has proven to be reliable because its is an adaptive learning rate algorithm.

For each model I experimented with, I trained using batch sizes of 256 and ran as many epochs as I could before the model overfit. Although most models overfit after the first or second epoch, I ran three to ten epochs, or full training runs, to be safe. Essentially the training was done through 6300 samples of images in batch sizes of 256 for three to ten epochs.

In general, a higher batch size would mean fewer weigh updates which would affect the quality of the model negatively. A higher batch size also requires higher computational power. However, even though a higher batch size can lower the quality of the model, it tremendously increases training speed and time. Inversely, decreasing the batch size could increase model quality however takes much longer to train.

Due to the fact using a batch size that was too low would greatly increase training time, I decided to experiment with a batch size of either 128 or 256. In addition, I did not want to make the batch size too high as it would require more computational power, which my computer did not possess, and I did not want to affect my model too negatively. Using a batch size of 128 decreased the validation loss by about .10 compared to using a batch size of 256, so there wasn't an enormous difference in results. However as expected, a batch size of 128 performed better as the validation loss of the model was lower. The increase in training time was also not too drastic. Thus, I determined a batch size of 128 was the best option.

I attempted to use the VGG16 pre-trained model itself to see if I can could get better results with transfer learning. I froze all bottom layers of VGG16, and created my own fully connected layers. I created two fully connected layers, one with 512 neurons/nodes and the final output layer with soft-max activation. I used the same sample size of 6300 (5040 training samples and 1260 validation samples) since I knew my computer could not train on more without extensive time. I ran five epochs with a batch size of 256. The results however did not prove to be better than my model. The initial validation loss was higher than the one my model achieved. The vgg16 model validation loss did decrease in the second epoch, however then started overfit and the validation loss consistently

increase from there. The lowest validation loss occurred in the second epoch of 5.49, which was higher than the 5.48 validation loss achieved on my model. Thus, my model ultimately proved to be slightly better.

Results from pretrained VGG16:

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1180160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2359808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2359808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2359808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2359808

```
block5_pool (MaxPooling2D)   (None, 7, 7, 512)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               12845568
_________________________________________________________________
dense_2 (Dense)              (None, 2282)              1170666
=================================================================
Total params: 28,730,922
Trainable params: 14,016,234
Non-trainable params: 14,714,688
_________________________________________________________________
Train on 5040 samples, validate on 1260 samples
Epoch 1/5
5040/5040 [==============================] - 8362s 2s/step - loss: 5.8428 - acc: 0.3706 -
val_loss: 5.6079 - val_acc: 0.3889
Epoch 2/5
5040/5040 [==============================] - 8292s 2s/step - loss: 5.3108 - acc: 0.3885 -
val_loss: 5.4997 - val_acc: 0.3889
Epoch 3/5
5040/5040 [==============================] - 8022s 2s/step - loss: 5.1524 - acc: 0.3885 -
val_loss: 5.5504 - val_acc: 0.3889
Epoch 4/5
5040/5040 [==============================] - 7994s 2s/step - loss: 5.0337 - acc: 0.3885 -
val_loss: 5.6056 - val_acc: 0.3889
Epoch 5/5
5040/5040 [==============================] - 7963s 2s/step - loss: 4.9292 - acc: 0.3885 -
val_loss: 5.6571 - val_acc: 0.3889
```

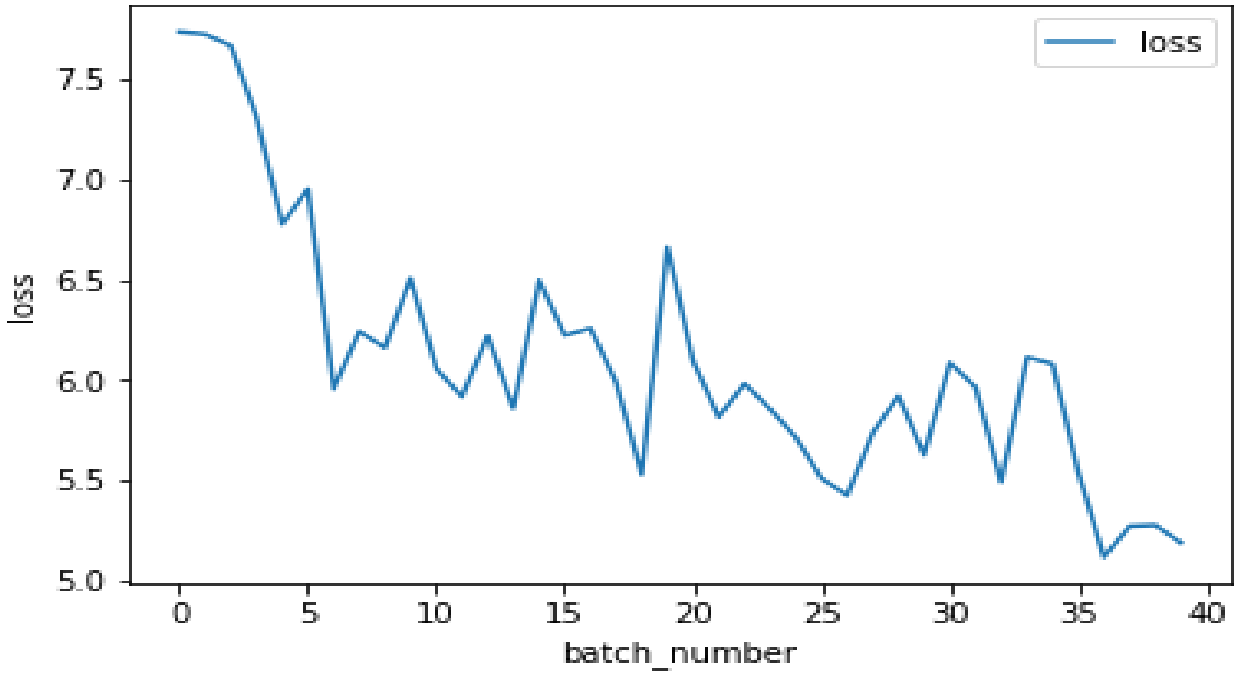
Results from Scratch CNN:

```
Layer (type)                 Output Shape              Param #
=================================================================
conv0 (Conv2D)               (None, 224, 224, 32)      896
_________________________________________________________________
activation_13 (Activation)   (None, 224, 224, 32)      0
_________________________________________________________________
conv1 (Conv2D)               (None, 224, 224, 32)      9248
_________________________________________________________________
activation_14 (Activation)   (None, 224, 224, 32)      0
_________________________________________________________________
max_pool (MaxPooling2D)      (None, 112, 112, 32)      0
_________________________________________________________________
conv2 (Conv2D)               (None, 112, 112, 32)      9248
```

activation_15 (Activation)	(None, 112, 112, 32)	0
conv3 (Conv2D)	(None, 112, 112, 32)	9248
activation_16 (Activation)	(None, 112, 112, 32)	0
max_pool2 (MaxPooling2D)	(None, 56, 56, 32)	0
conv4 (Conv2D)	(None, 56, 56, 32)	9248
activation_17 (Activation)	(None, 56, 56, 32)	0
conv5 (Conv2D)	(None, 56, 56, 32)	9248
activation_18 (Activation)	(None, 56, 56, 32)	0
max_pool3 (MaxPooling2D)	(None, 28, 28, 32)	0
conv6 (Conv2D)	(None, 28, 28, 32)	9248
activation_19 (Activation)	(None, 28, 28, 32)	0
conv7 (Conv2D)	(None, 28, 28, 32)	9248
activation_20 (Activation)	(None, 28, 28, 32)	0
max_pool4 (MaxPooling2D)	(None, 14, 14, 32)	0
conv8 (Conv2D)	(None, 14, 14, 64)	18496
activation_21 (Activation)	(None, 14, 14, 64)	0
conv9 (Conv2D)	(None, 14, 14, 64)	36928
activation_22 (Activation)	(None, 14, 14, 64)	0
max_pool5 (MaxPooling2D)	(None, 7, 7, 64)	0
conv10 (Conv2D)	(None, 7, 7, 64)	36928
activation_23 (Activation)	(None, 7, 7, 64)	0
conv11 (Conv2D)	(None, 7, 7, 64)	36928

```
_________________________________________________________________
activation_24 (Activation)   (None, 7, 7, 64)        0
_________________________________________________________________
max_pool6 (MaxPooling2D)     (None, 3, 3, 64)        0
_________________________________________________________________
flatten_2 (Flatten)          (None, 576)             0
_________________________________________________________________
dense_2 (Dense)              (None, 500)             288500
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)             0
_________________________________________________________________
sm (Dense)                   (None, 2282)            1143282
=================================================================
Total params: 1,626,694
Trainable params: 1,626,694
Non-trainable params: 0
_________________________________________________________________
Train on 5040 samples, validate on 1260 samples
Epoch 1
5040/5040 [==============================] - 896s 178ms/step - loss: 6.1124 - acc:
0.3732 - val_loss: 5.4830 - val_acc: 0.3889
Epoch 2
5040/5040 [==============================] - 906s 180ms/step - loss: 5.5190 - acc:
0.3885 - val_loss: 5.5284 - val_acc: 0.3889
Epoch 3
5040/5040 [==============================] - 867s 172ms/step - loss: 5.4625 - acc:
0.3885 - val_loss: 5.5577 - val_acc: 0.3889
Epoch 4
5040/5040 [==============================] - 942s 187ms/step - loss: 5.4398 - acc:
0.3885 - val_loss: 5.6139 - val_acc: 0.3889
Epoch 5
5040/5040 [==============================] - 945s 188ms/step - loss: 5.3850 - acc:
0.3885 - val_loss: 5.6355 - val_acc: 0.3889
Epoch 6
5040/5040 [==============================] - 894s 177ms/step - loss: 5.3885 - acc:
0.3885 - val_loss: 5.6897 - val_acc: 0.3889
Epoch 7
5040/5040 [==============================] - 875s 174ms/step - loss: 5.3400 - acc:
0.3885 - val_loss: 5.6925 - val_acc: 0.3889
```

Graph indicating the change in training loss per batch for one Epoch (Scratch CNN):



Corresponding Pandas Dataframe:

batch	loss
0	7.732180
1	7.720391
2	7.664803
3	7.306093
4	6.776271
5	6.950381
6	5.954907
7	6.241403
8	6.159783
9	6.504392
10	6.055659
11	5.916712
12	6.219953
13	5.859550
14	6.497148
15	6.221243
16	6.257275
17	5.990325
18	5.527941
19	6.661013
20	6.094501

21	5.814867
22	5.982471
23	5.854703
24	5.713407
25	5.509876
26	5.423401
27	5.736097
28	5.920380
29	5.626149
30	6.084803
31	5.964196
32	5.491403
33	6.113189
34	6.079313
35	5.540252
36	5.118431
37	5.271231
38	5.276492
39	5.189406

As we can see here, batch numbers 14, 19, and 33 proved to be the most difficult for the model to learn. Each one of those batches saw a large spike in the loss function.