Machine Learning II
Final Project Report

Topic: Computer Vision Real World Utilization
—— Humpback Whale Identification

Yuan Meng

Introduction

Computer vision is rapidly evolving every day. One reason for this is deep learning. When we talk about computer vision, we think of the term convolutional neural network (abbreviated as CNN). Examples of CNNs in computer vision are face recognition, image classification, and so on. This project is to build a CNN model to identify different whale species by recognizing the shape of whales' tails and unique markings found in footage using over 25,000 images stored in a database called Happy Whale. The human activity of killing whales for oil and food has decreased the population of whales in the world, especially humpback whales. For a long time, oceanographers and marine scientists have photographed ocean activity to keep a log on the population of whales, but this is a slow and work-intensive way of manually keeping a record. Therefore, a faster and more effective way of identifying whale species could help a lot.

This is an interesting problem because it's regarding computer vision. Nowadays there are lots of companies developing tools and software that can identify objects. For instance, take the concept of self-driving cars being developed by Uber, Tesla, etc. These cars need to identify objects around them in order to drive carefully in the surrounding environment. If a person comes in front of the car all of a sudden, the car applies brakes and stops. But this only happens when the car successfully recognizes there's an object in front and stops right away. Objection detection systems and algorithms that can work flawlessly are the utmost expectation of this project. And therefore I am interested in the computer vision topic and want to experiment with this real-world image recognition and classification problem. Moreover, it's about keeping successful surveillance on the ocean activity to make sure the whale's marine life is going normal. If fewer whales are seen, this could indicate a problem. So for the sake of marine life and the natural ecosystem, it's important.

The report firstly displays the description of the data set, and secondly discusses the research studies about computer vision and works relating to this project, which explains why I chose ResNet as the network to train the data. It thirdly describes the network and training algorithm. Next, the experimental setup is discussed, and then the report shows the experiments with their results. At last, is the conclusion of this project.

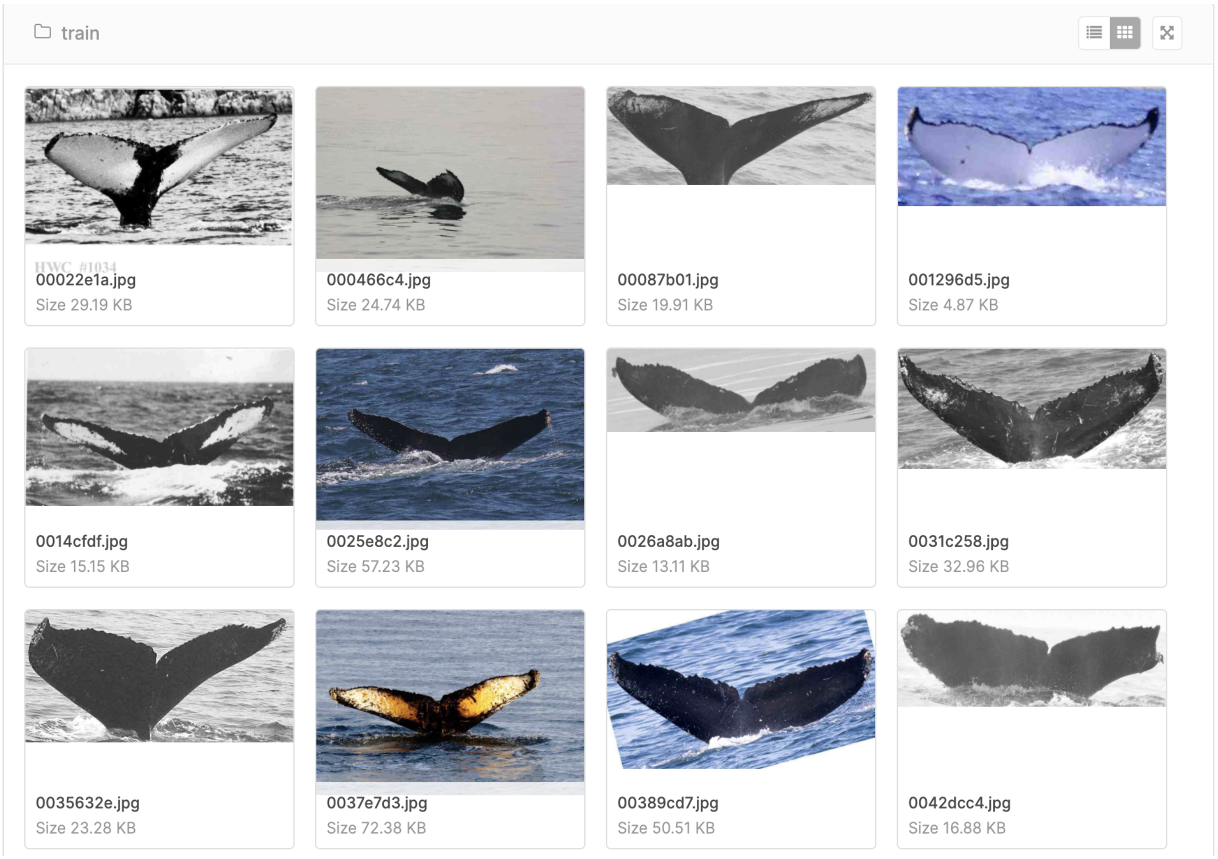
Data Description

Collected from the Happywhale Organization [1], the Kaggle [2] dataset involves both training and testing datasets. The training dataset for whale identification includes JPG files and CSV files. The 9850 images are humpback whale flukes' photographs, and the CSV file is to match each image with the related whale ID. If the individual whale has been identified by researchers, it will have an ID, such as "w_1287fbc". Otherwise, the individual whale will be labeled as a new whale with the ID "new_whale". Here is the CSV file of the training data.

```
             Image          Id
0     00022e1a.jpg  w_e15442c
1     000466c4.jpg  w_1287fbc
2     00087b01.jpg  w_da2efe0
3     001296d5.jpg  w_19e5482
4     0014cfdf.jpg  w_f22f3e3
...            ...        ...
9845  ffe5c306.jpg  w_2ceab05
9846  ffeaa7a4.jpg  w_b067417
9847  ffecec63.jpg  w_8b56cb1
9848  fff04277.jpg  w_2dcbf82
9849  fffd4260.jpg  w_b9bfd4e

[9850 rows x 2 columns]
```

And here are some pictures of the whale tail that I need to train and identify.



From these views, I can learn that the shapes and the edges of the tail are important characteristics for whale identification. Some interesting things I noted were that among the 9580 images, 8% are new whales and the rest belong to 4250 identified humpback whales, which means that there are only a few examples for each of the whale IDs. The testing data, then, includes 15610 humpback whale fluke images. My goal is to predict such 15610 images' whale ID based on the 9580 labeled images. I expected more examples of each of whale ID's though.

According to the Kaggle [2] evaluation rules, the performance of a method was to be evaluated by the Mean Average Precision $MAP = \frac{\sum_{q=1}^{Q} AveP(q)}{Q}$, where Q is the number of queries. And precision is the positive predictive value (PPV), where $Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$. For my prediction result, I will predict 5 labels for the whale ID for each image, so Q is 5 here. The higher the MAP is, the better my method is.

Research Studies

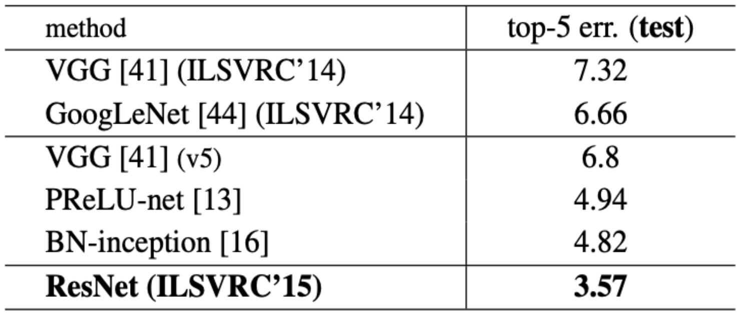
Related work has been adapted and compiled from three sources: two research studies done at Microsoft and New York University respectively and one done on humpback whale identification methods at Universidad Politécnica de Madrid in Spain. have evaluated neural networks with various image recognition applications to deeper extents and thus closely related to our target problem. Given below is the summary of each of the related work.

1. *Humpback whale identification with convolutional neural networks by* Universidad Politécnica de Madrid*. (2018). [3]*

This research paper was the backbone of our project. It laid the fundamentals of identifying humpback whale specimens from the patterns of different tails of whales. Starting with using Python along with main packages like Keras/Pytorch, Scikit learn, Python Imaging Library (PIL), etc. this paper is a whole guide to the problem. First image processing gets done and data is homogenized. Then various data augmentation techniques are used like rotation, translation, etc. to increase the number of data for training purposes. Convolutional layers are established by normalization of batch and pool reduction after which training starts. The loss function is vital because it tells us how far are the predicted labels from real labels. In this case, the loss function used is Categorical Cross-Entropy (CCE). Adam (also known as Adaptive Moment Estimation) is the optimization algorithm used to modify the learning rate. The learning process is executed over iterations over different batches. Thus this is a brief description of how fundamentals were laid for identifying humpback whale specimens from the patterns of different tails of whales.

2. *Deep Residual Learning for Image Recognition by Microsoft Research Group (Kaiming, Xiangyu, Shaoqing, Jian). (2015). [4]*

This paper was focused on making the most out of a neural network for image recognition tasks. Sometimes it can be hard to train neural networks and this paper suggested a residual learning framework to facilitate the learning of networks that are deeper than their counterparts. The basic comparison was between different layered ResNets and VGG neural networks. They both were applied to the ImageNet 2012 classification dataset that consisted of a thousand classes and CIFAR-10 dataset. VGG and ResNet are both very solid neural networks when it comes to image recognition and after intense competition, ResNet emerged as the winner. The main reason for ResNet winning was that it showed the least amount of error rate in predictions as shown in figure 0.



method	top-5 err. (**test**)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

Figure 0: Error rates (%) reported by test servers. [4]

While VGG reduced the number of parameters in the convolutional layers and enhanced the training time, ResNet architecture made use of shortcut connections to deal with vanishing gradient problems.

3. *Learning a Similarity Metric Discriminatively with Application to Face Verification by Courant Institute of Mathematical Sciences at New York University. (2017). [5]*

This study was very similar to this project. It dealt with recognizing images where categories are very large and unknown at training time. However, here a unique tool was used for image recognition called Face Verification with Learned Similarity Metrics. This approach works on two principles: evaluating the percentage of false accepts and false rejects on features of an image. At its core, it uses Siamese neural architecture which works in the following way: a system for training is formulated that non-linearly maps features of images to certain points in a space dimension. Distance between the points and features is small if the images belong to the same person and larger if the images don't belong to the same person. The similarity metric plays a huge role in calculating these distances and the Siamese neural network is what powers this metric. This is a unique application of neural networks being used differently for image recognition.

Deep Learning Network: CNN

It is similar to a basic neural network, CNN also has learnable parameters, such as neural networks, weights, biases, and so on. Below are some mathematical notations for the data. Although it's not possible to go into detail about each and every math notation, there are a few we'd like to mention.

1: Kernel Convolution which gives indexes of rows and columns of data matrices. M and N are indexes, f is the filter size. [6]

$$G[m,n] = (f * h)[m,n] = \sum_{j}\sum_{k} h[j,k]f[m-j,n-k]$$

2: Padding formula where f is the dimension of the filter is applied. [6]

$$p = \frac{f-1}{2}$$

3: Strided Convolution giving output data matrix. P is padding, f is filter size and s is stride. [6]

$$n_{out} = \left\lfloor \frac{n_{in} + 2p - f}{s} + 1 \right\rfloor$$

4: Dimension Adjusting formula which applies multiple filters to adjust dimensions. N is the image size, f is filter size, p is padding, s is stride. [6]

$$[n, n, n_c] * [f, f, n_c] = \left[\left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor, \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor, n_f \right]$$

Here are some different network architectures I used for the project.

Architecture 1: Customized ResNet

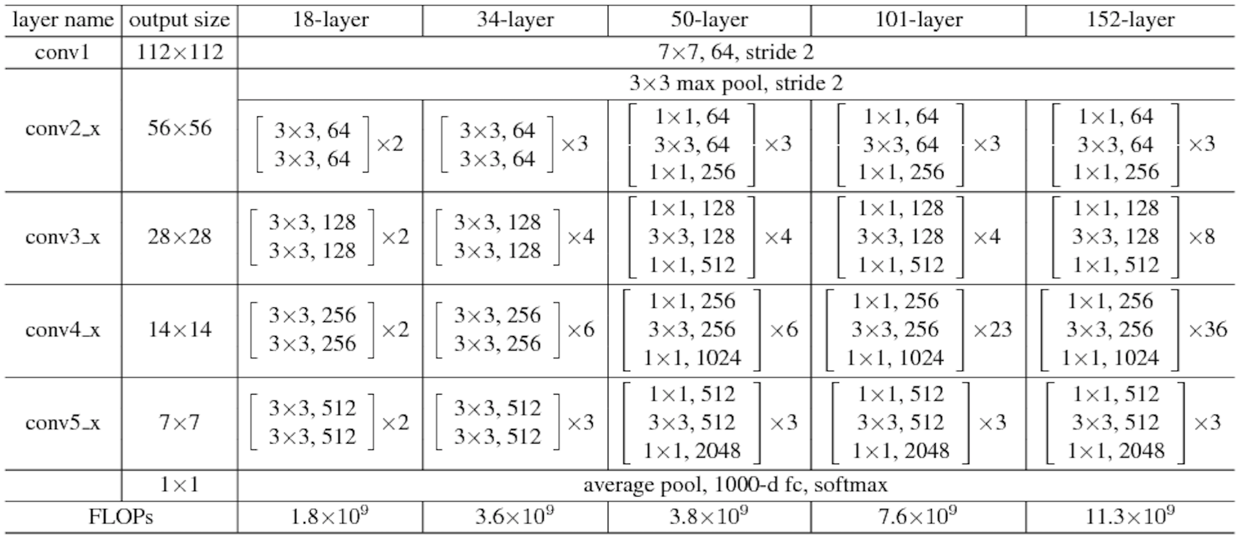
After reading Kaiming's paper, I decided to construct the ResNet network as the model for this project to train the dataset. Its layers were fitted in a residual mapping, instead of directly trying to fit a desired underlying mapping. It had one 2D convolution layer and one Batch Normalization at the beginning, and followed with three sequential layers that had three basic blocks connecting between each layer. Each basic block contained two 2D convolution layers and two Batch Normalizations and followed with one shortcut. It had 3x3 kernel size, 1x1 stride, and 1x1 zero-padding on each convolution layer, and it had 1e-05 eps, 0.1 momenta, learnable affine parameters and it tracked the running mean and variance. Lastly, there was a linear layer as the output layer. The figure is shown below.

```
ResNet(
  (conv1): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
    (1): BasicBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
    (2): BasicBlock(
      (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (shortcut): Sequential()
    )
  )
```

```
(layer2): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): LambdaLayer()
  )
  (1): BasicBlock(
    (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential()
  )
  (2): BasicBlock(
    (conv1): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential()
  )
)
(layer3): Sequential(
  (0): BasicBlock(
    (conv1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): LambdaLayer()
  )
  (1): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential()
  )
  (2): BasicBlock(
    (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (shortcut): Sequential()
```

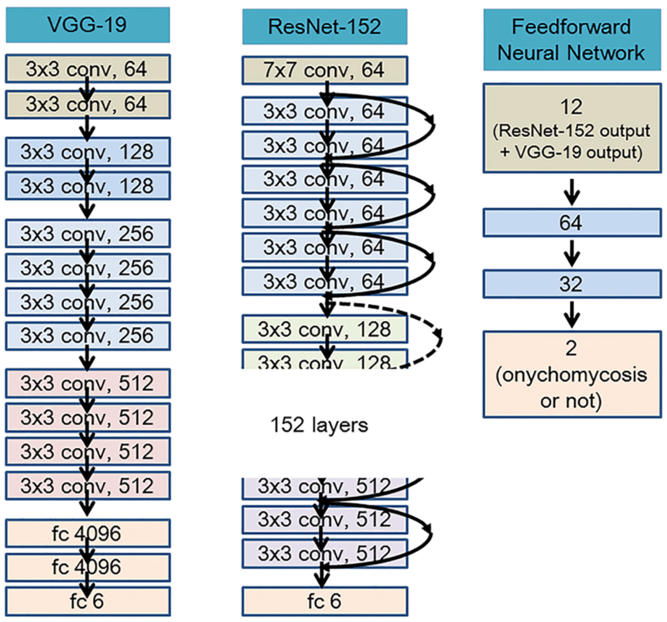
Architecture 2 & 3: ResNet-152 and ResNet-50

Inspired by research studies, I found that it is more efficient and convenient to train a convolutional neural network for image classification using transfer learning because the pre-trained networks have sufficient training datasets and reasonable architectures. For instance, according to the document of RESNET posting by Pytorch Team, Resnet models were proposed in "Deep Residual Learning for Image Recognition" and different versions of models are listed below. Thus, the second network I tried was the ResNet-152, referring to the codes from the PyTorch library[7].

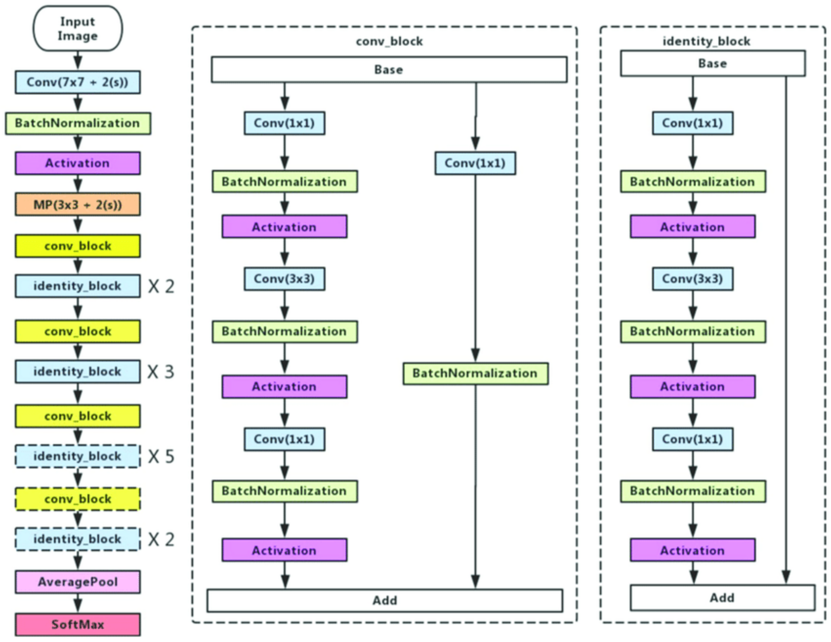


layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 64\\3\times3, 64\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 64\\3\times3, 64\\1\times1, 256\end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 128\\3\times3, 128\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times4$	$\begin{bmatrix}1\times1, 128\\3\times3, 128\\1\times1, 512\end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 256\\3\times3, 256\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times6$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times23$	$\begin{bmatrix}1\times1, 256\\3\times3, 256\\1\times1, 1024\end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times2$	$\begin{bmatrix}3\times3, 512\\3\times3, 512\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$	$\begin{bmatrix}1\times1, 512\\3\times3, 512\\1\times1, 2048\end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

The middle part of the graph below shows the visual architecture of ResNet-152[8].



From the graph above we can see that ResNet-152 has too many layers and blocks, so I spent too much time running on it and got the model overfitting. Therefore, I tried the next method, ResNet-50. Here is the visual architecture of ResNet-50[9].



From the table above we can see that ResNet-50 has half the number of conv3 layers less than that of ResNet-152, and it only has one-sixth of conv4 layers of ResNet-152. So it won't be too complex to train the data.

Training Algorithm

I used the CrossEntropyLoss function in the PyTorch library[10] as the loss function, and the sum of the output will be divided by the number of elements in the output. The loss function is:

$$\text{loss}(x, class) = -\log\left(\frac{\exp(x[class])}{\sum_j \exp(x[j])}\right) = -x[class] + \log\left(\sum_j \exp(x[j])\right).$$

Also, I used the stochastic gradient descent (SGD) optimizer with 0.9 momenta and 1e-4 weight decay, and the network parameters and the learning rate changed while training. According to the PyTorch document[11], the implementation of SGD with Momentum/Nesterov subtly differs from Sutskever et. al. and implementations in some other frameworks.

Considering the specific case of Momentum, the update can be written as

$$\upsilon_{t+1} = \mu * \upsilon_t + g_{t+1},$$
$$p_{t+1} = p_t - lr * \upsilon_{t+1},$$

where p, g, v, and μ denote the parameters, gradient, velocity, and momentum respectively.

This is in contrast to Sutskever et. al. and other frameworks which employ an update of the form

$$\upsilon_{t+1} = \mu * \upsilon_t + lr * g_{t+1},$$
$$p_{t+1} = p_t - \upsilon_{t+1}.$$

The Nesterov version is analogously modified.

Experimental Setup

To train the network, I used an annealed learning rate scheme as follows: start with an initial learning rate of 0.01, and multiplied it by a decay factor of 0.1 after 16 and 19 epochs, respectively, and stopped the training after 20 epochs. I determined such learning rate because in the scheme I found the loss had the largest decrease after each epoch with lr=0.01. Also, I used the mini-batch size of 128 for the training parameter. I started from 512 and continually decreased the batch size to find one that won't run out of the GPU memory, because the larger batch size is the better efficiency.

For every epoch, every training image would be put into the network along with loss function and optimizer in forward and backward, and then for validation without computing gradient and with no backward. The training loss and the validation loss would then be aggregated to total train loss and total validate loss separately, and if the total validate loss was less than the previous best error record, then the whole state of the module would be saved to the given path, and the best error record would be updated as the total validate loss. When a total of twenty epochs was completed, the best model would be loaded from the path, and the total train loss and total validate loss would be returned.

To judge the performance, I computed the validation mean average precision by taking the top 5 possible torch indexes as the predicted labels and comparing them with the real encoded label. If the first predicted label equals the real class, then the score adds 1, for the second one adds ½, and so on, until if the fifth predicted label equals the real one, then adds ⅕. Finally, with the total score divided by the total length of the data, I could get the mean average precision.

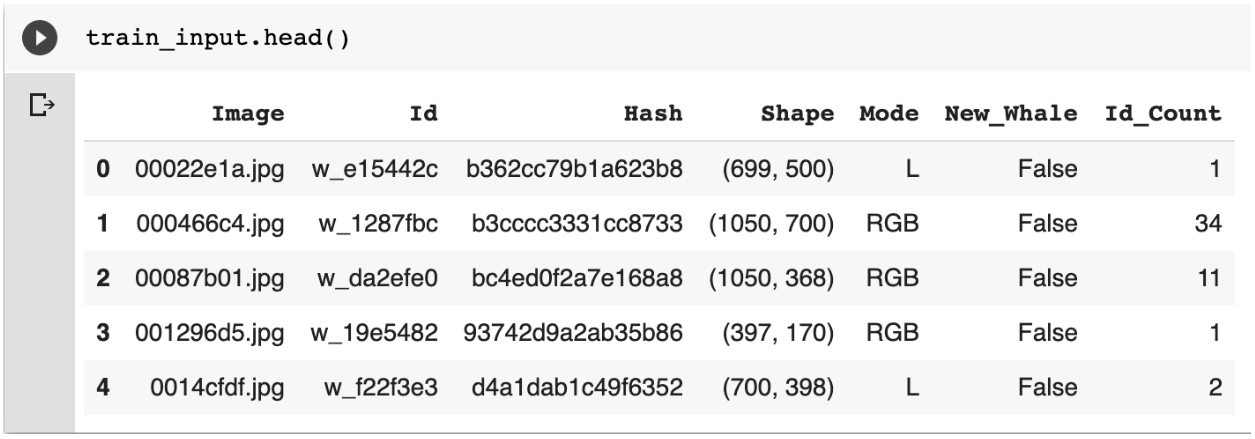
I also plot the training and validation loss of each epoch to detect overfitting: if the training loss decreases but the validation loss increases, it is overfitted. And this occurred in the following part. In addition, I used data augmentation combining data loaders to prevent overfitting. For each iteration of the data loading, the data would be randomly transformed depending on the transformation function I defined, so for each epoch, the transformed data would be slightly different to avoid overfitting. The detailed data augmentation method will be explained in the data processing part.

Experiments with their results

This part shows how I complete the project step by step, and the results of the experiments.

Data Processing

Exploring and understanding data files is the fundamental but important step of the project. Though from the visuals above I can learn the basic information about images, I want to know the specific data of these images. Using the PIL and the imagehash library, I got their hash, shape, mode, number of each id, and whether the image belonged to the "new whale".



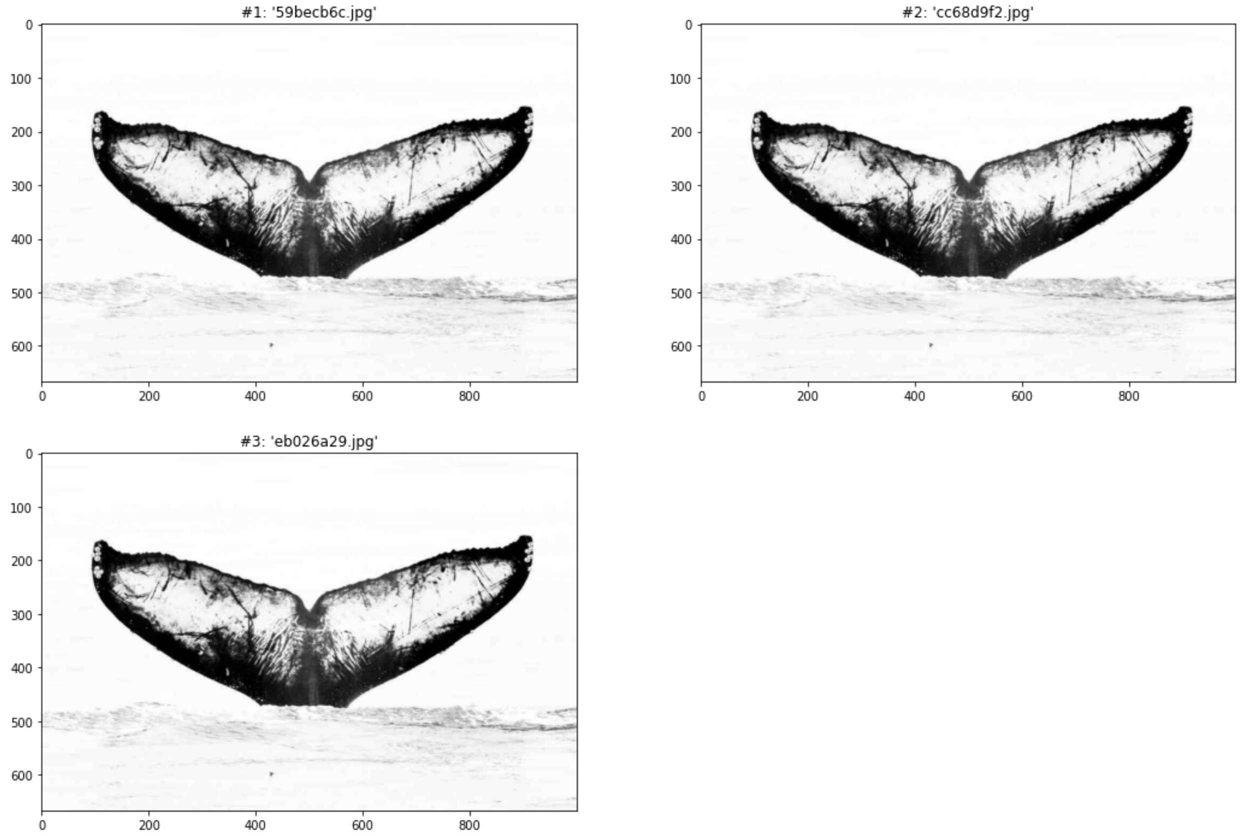
```
train_input.head()
```

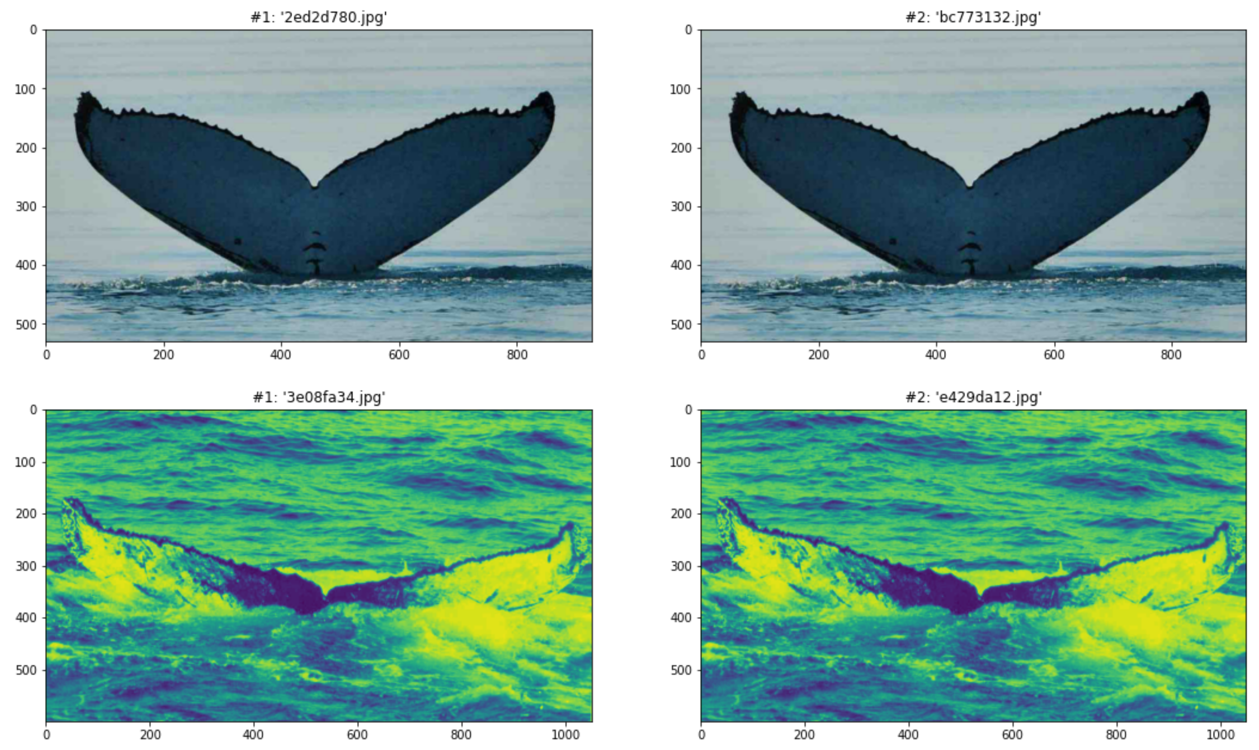
	Image	Id	Hash	Shape	Mode	New_Whale	Id_Count
0	00022e1a.jpg	w_e15442c	b362cc79b1a623b8	(699, 500)	L	False	1
1	000466c4.jpg	w_1287fbc	b3cccc3331cc8733	(1050, 700)	RGB	False	34
2	00087b01.jpg	w_da2efe0	bc4ed0f2a7e168a8	(1050, 368)	RGB	False	11
3	001296d5.jpg	w_19e5482	93742d9a2ab35b86	(397, 170)	RGB	False	1
4	0014cfdf.jpg	w_f22f3e3	d4a1dab1c49f6352	(700, 398)	L	False	2

First, I found that there were some duplicated images in the training data sets due to the same hash.

```
There are 780 duplicate images.
bb8ec43039cb663c    3
bcccd3346b342d0b    2
b619898ea6a6e1e9    2
e89a85e3b661d871    2
8f90e168da67b4c9    2
Name: Hash, dtype: int64
```

Here are some sample visuals of the duplicated images.





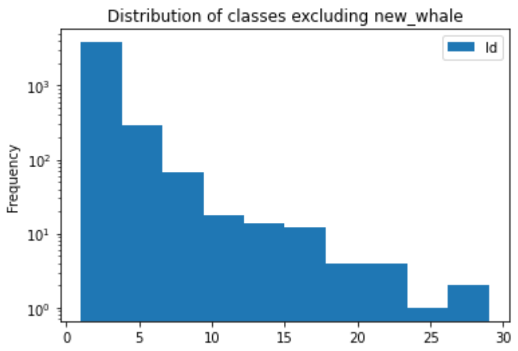
Most of these images had the same labeled ids, but some of them had different ids.

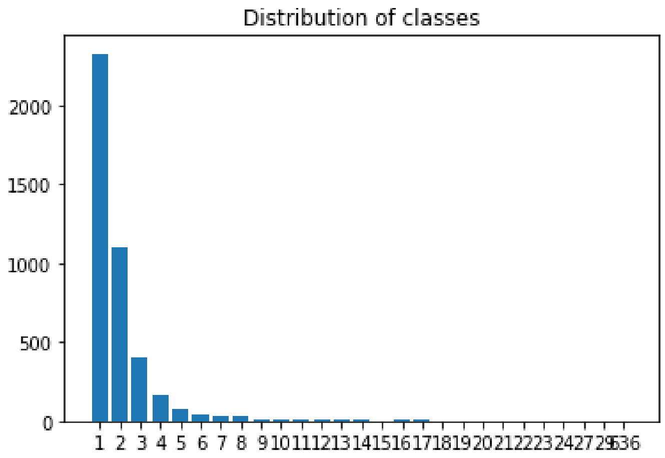
	Hash	Ids	Ids_count	Ids_contain_new_whale
bb8ec43039cb663c	3	{w_cae7677, new_whale}	2	True
bcccd3346b342d0b	2	{w_4848a3c}	1	False
b619898ea6a6e1e9	2	{w_9548fcf, new_whale}	2	True
e89a85e3b661d871	2	{new_whale, w_7c18f3c}	2	True
8f90e168da67b4c9	2	{w_4fd48e7}	1	False

For those whose ids count was equal to one, I dropped the duplicated image directly. For those ids counts were more than one and contained the “new whale” label, I dropped the “new whale” images. The rest of them were the images that contained different labels but not new whales, so I dropped all of them to avoid confusing the training model. The number of images was decreased from 9850 to 9060, and the number of classes was decreased from 4251 to 4246.

Second, I found that the distribution of the number of classes was extremely unbalanced.

```
new_whale     636
w_1287fbc      29
w_98baff9      27
w_7554f44      24
w_1eafe46      23
                ...
w_ab39440       1
w_7e8305f       1
w_f801078       1
w_c493795       1
w_3e9d82e       1
Name: Id, Length: 4246, dtype: int64
```





From the table and the graphs above, I learned that except for the new whale category which had 636 images, the large size classes only had about twenty files, and most of the class only had less than five files. Especially, there were 2329 classes that only had one image, which would cause extremely negative effects on the prediction process. Therefore, I used the oversampling method to reduce the influence. For those classes which had a number of images less than ten, I increased these samples based on data transformation methods.

Third, I decided to process the data augmentation and normalization. As from the previous image information table, I found that the images had different sizes and models (L/RGB), so we used the torchvision.transforms function to deal with them. For original 9060 training data, I converted them to grayscale with output channels equaled 3, random horizontal and vertical flip, resized them to 224x224 image size, and normalized them using the Normalize function with the per-channel mean [0.485, 0.456, 0.406] and per-channel standard deviation [0.229, 0.224, 0.225]. To compose these transforms together, I used the Compose function, and I used the ToTensor to convert a PIL Image to tensor. For the images generated for oversampling, I also implemented center crop and random rotation with $\pm$ 15.

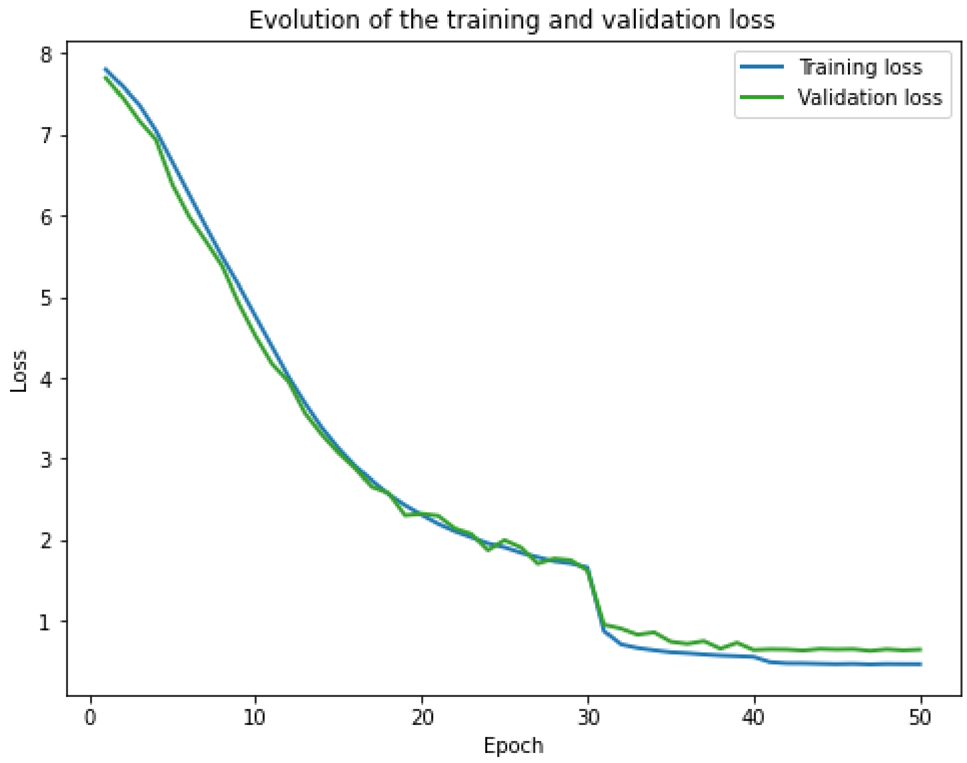
Finally, I got the training set with length 64286, 4246 classes, and at least 10 files per class. And 9060 validation images which only got converted to the grayscale, resized, to tensor, and normalized.

Result 1: Customized ResNet with 50 epoch

Then I implemented model training following steps described in the experiment setup part, and here are some layer details.

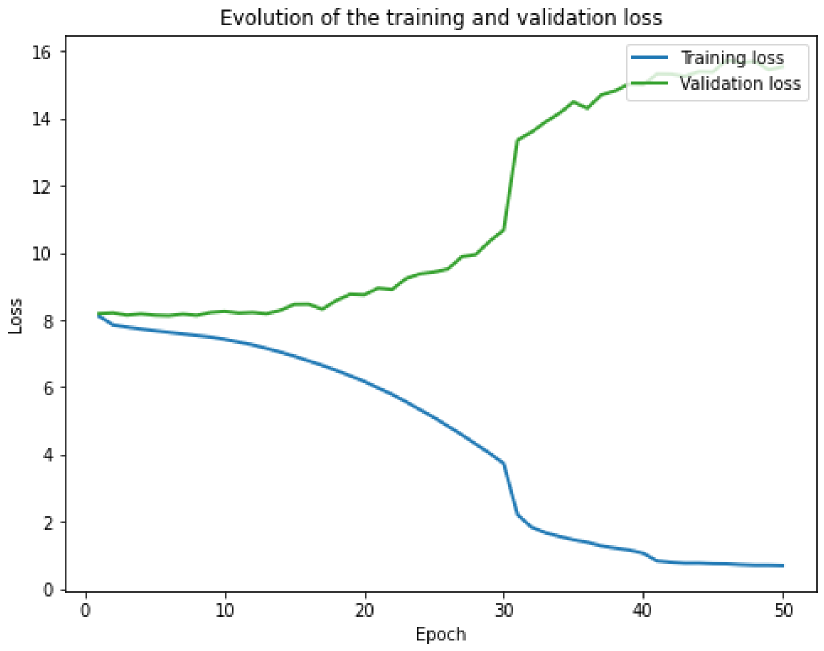
In 2D convolutional layers, the filter ran through the training data in two dimensions. After convolution was concluded, the features of images were downsampled. Before downsampling, the layers thoroughly inspected the images like first identifying the original image then going to detail to extract smaller features of the images. Essentially this helped the learning process. 2D max-pooling layers helped it in reducing the size of the data and control overfitting. So this way I could ensure the accuracy of our model. I set the learning rate starting with 0.01, and multiplied it by a decay factor of 0.1 after 30 and 40 epochs, respectively, and stopped the training after 50 epochs, with other model settings introduced in previous sections.

After training and testing, I got a 0.80 validation MAP score and a 0.65 validation loss, which seems to have worked well; however, I got a 0.32203 Kaggle score. I thought the reason it was not good enough as it did not have enough layers.



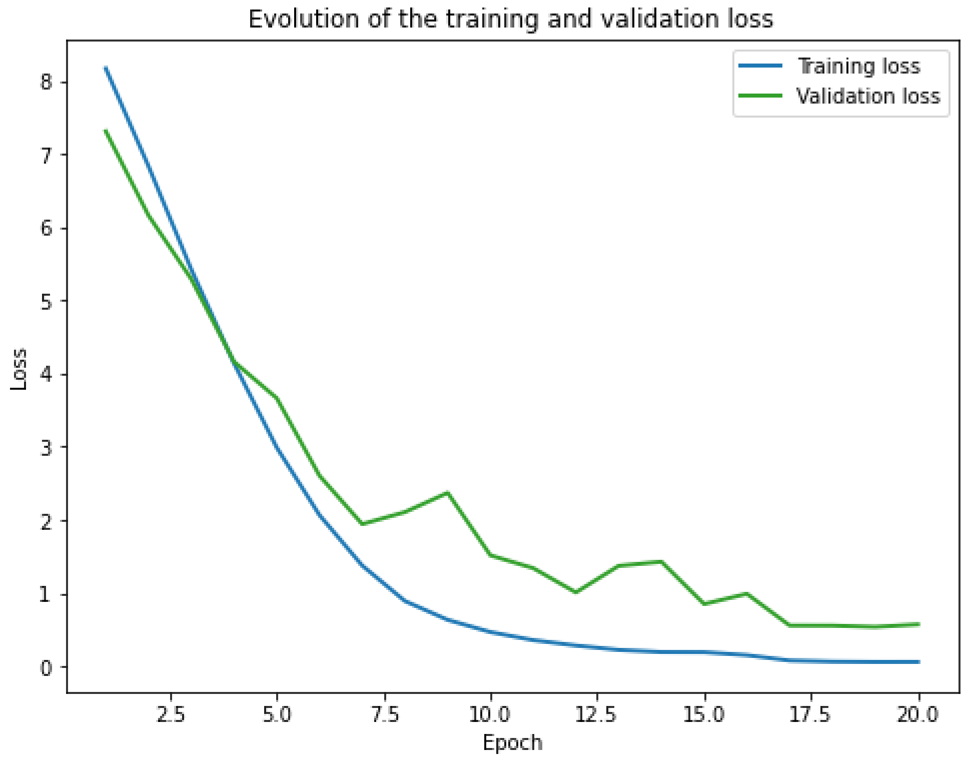
Result 2: ResNet-152 with 50 epoch

I secondly tried the most layered ResNet, ResNet-152, for training, but it took too much time. I got validation loss increasing while training loss decreasing, so the model was overfitting. I thought this occurred due to too many unnecessary layers, so I then simplified the architecture.



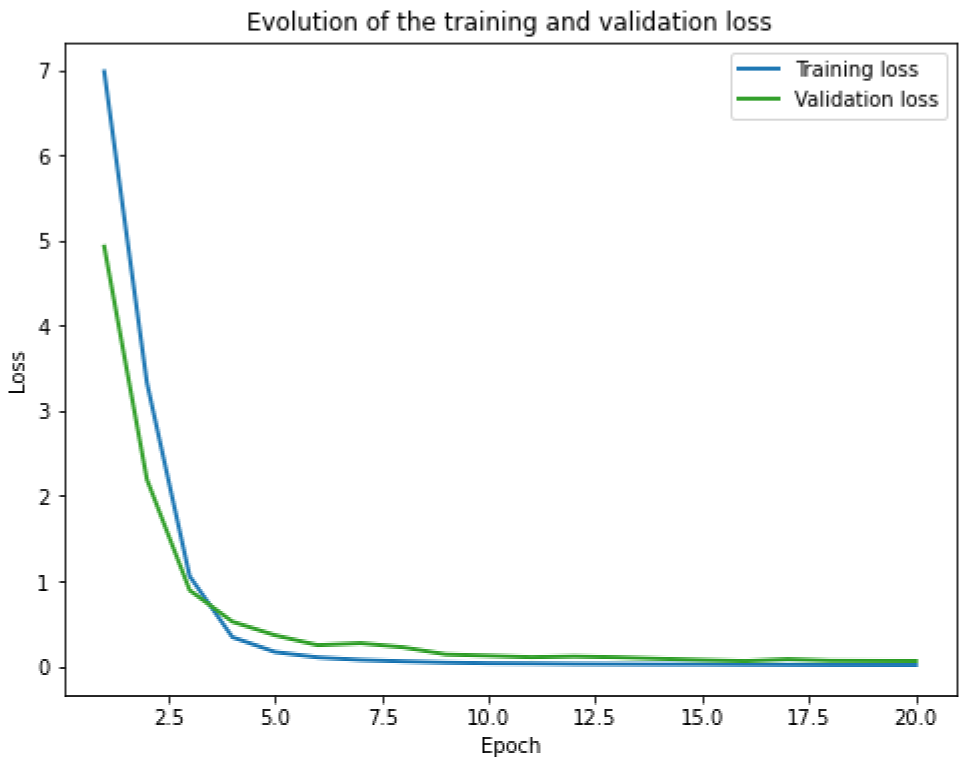
Result 3: ResNet-50 with 20 epoch

Then I decided to try less layered architecture, ResNet-50, which still cost 12641.62s for a total of 20 epoched, but it didn't overfit. I got a 0.93 validation MAP score and a 0.54 validation loss.



Result 4: Pretrained ResNet-50 with 20 epoch

For previous methods, I only used the architecture of ResNets but didn't try the pre-trained network. Because I thought my dataset was very different from the original one, it might not work well. However, when I set the pre-trained function as True, I surprisingly found it got efficiently better than before. The results are 0.99874 validation MAP score, 0.01 training loss, and 0.06 validation loss.



This is the best performance, and I got a Kaggle score of 0.44343. Kaggle does not show position on the leaderboard after the competition deadline passes and this competition ended 3 years ago. Nonetheless, our Kaggle score puts me in the 55th position out of a total of 527 entries for the competition. This qualifies me to the top 10% highest achievers of the competition which is pretty well done and way above average.

Conclusion

This project was an interesting and in-depth experience of using different tools and approaches for image recognition. I got hands-on experience with data processing and pre-trained networks. Throughout this project, I learned that the reason ResNet is better than its counterparts is that it introduced a genuine solution to training deep networks called "Residual Block". Residual Block is basically a neural network layer with an important phenomenon called Skip Connections or Identity Mapping. This connection does not comprise parameters of its own but rather adds the outputs of previous layers of networks to the layer of networks ahead. Then a linear projection is multiplied with the mapping to expand channels to match the residual. Hence this is how ResNet is different from its competitors and is really successful for image recognition. There are also various data augmentation tricks that I learned which are useful when data is not in an appropriate format and you need to get it into a standardized format or increase training data by using functions like rotation, horizontal flips, etc. This project has imparted to me one of the most important skills and that is how to build a solid neural network and enhance it using optimizers, loss functions, etc. according to the need of the task. As for future possible improvements, I will try different pre-trained networks and try to find better-oversampling methods.

References

1. Happywhale.
https://happywhale.com/home

2. Kaggle Humpback Whale Identification Challenge.
https://www.kaggle.com/c/whale-categorization-playground/overview

3. Can, D. (2018). Humpback whale identification with convolutional neural networks.
https://www.preprints.org/manuscript/201902.0257/v2

4. ResNet paper: Deep Residual Learning for Image Recognition.
https://arxiv.org/pdf/1512.03385.pdf

5. Learning a Similarity Metric Discriminatively with Application to Face Verification.
http://yann.lecun.com/exdb/publis/pdf/chopra-05.pdf

6. Skalski, Piotr. Gentle Dive into Math Behind Convolutional Neural Networks. Towards data science, 2019.
https://towardsdatascience.com/gentle-dive-into-math-behind-convolutional-neural-networks-79a07dd44cf9

7. Pytorch Team. "RESNET". PyTorch. https://pytorch.org/hub/pytorch_vision_resnet/

8. ResNet-152
https://www.researchgate.net/figure/The-representation-of-model-architecture-image-for-ResNet-152-VGG-19-and-two-layered_fig2_322621180

9. ResNet-50
https://www.researchgate.net/figure/Left-ResNet50-architecture-Blocks-with-dotted-line-represents-modules-that-might-be_fig3_331364877

10. CROSSENTROPYLOSS
https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

11. SGD https://pytorch.org/docs/master/generated/torch.optim.SGD.html

Codes borrowed and adjusted from:

12. TheGoose. https://www.kaggle.com/stehai/duplicate-images

13. TheGoose. https://www.kaggle.com/stehai/duplicate-images-data-cleaning

15. https://www.kaggle.com/artgor/pytorch-whale-identifier/notebook

14. Chilamkurthy, Sasank. “WRITING CUSTOM DATASETS, DATALOADERS AND TRANSFORMS.” PyTorch. https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

15. Chilamkurthy, Sasank. “TRANSFER LEARNING FOR COMPUTER VISION TUTORIAL.” PyTorch. https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html

17. Github 30 Contributors. https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py

Documented computer listings (code) can be found in the code folder.