Image Classification of Fish Species Into Different Categories

ABSTRACT

This project provides an attempt to solve the challenge Kaggle Nature Conservancy Fisheries Monitoring. The models reported in this report provide a solution to categorize the eight-different species of fish and thus it becomes a multi-class classification problem. Several different models, ranging from naïve baseline models to sophisticated models using CNN have been trained for the classification task. To evaluate the success of our models and check their performance we use a metric named multi-class logarithmic loss. For training our baseline models we have used Naïve Bayes, support vector machines and K-Nearest Neighbor models. Whereas for training our Neural Nets models we use Convolutional Neural Networks, VGG16 and InceptionV3 architecture, Data Augmentation, and regularization techniques such as dropout. After fine tuning the models and compensating for overfitting using different regularization techniques we find that a pretrained VGG-16 model performs the best for our scenario on the validation set. fish species and on the test set provided by Kaggle. We see that the results are quite encouraging despite the computational limitations we faced but they could be improved if the computational limitation is removed and a technique named as bounding box regression is employed on annotated training data that provides the location of fish in the image. This opens a whole new avenue for future work and interesting outcomes that will improve the performance of the models.

GOAL

Seafood, it's one of the main sources of protein. Countries like the USA, Japan, China and Australia lying in Western and Pacific region contribute high quality fish to make up an industry of a whopping $7 billion market. Around 50% population of the world is dependent on seafood for their protein needs. But, serving rare and exotic fish as delicacies in restaurants and promotion of deep sea tourism has led to illegal catching of fish. A surprising fact is that the blue whale is only caught for its fins and once the fins are cut off the remaining is thrown back into the water. The fins are used to make a costly delicacy known as Blue fin soup. Illegal catching of fish is not new but has increased lately and poses a serious threat to our marine ecosystem. A report on this situation states that fishing operators located in the pacific region should be ideally sending a

person to fishing trips nearly 200 times per year but, it's happening just 10 times a year. Thus, there is no one to monitor the extent of overfishing and illegal catching of rare species of fish rest of the times. To provide a solution to this problem, a global non-profit organization called The Nature Conservancy has stepped up. It proposes to use technology to mitigate this problem. With the help of monitoring devices like sensors, GPS devices and cameras it plans to monitor all illegal activities happening on board. The solution is effective but not practical enough as hours and hours of raw footage needs to be manually reviewed, also the weather conditions during fishing trips are not good and the footage might get affected due to water droplets on camera lens or due to insufficient amount of light thereby making it difficult for a human to observe the footage. Therefore, to make this a working solution this organization has partnered with Kaggle to ask machine learning professionals and enthusiasts to build a system based on convolution neural network that will automatically scan the raw video footage and detect and classify various species of fishes. Using machine learning techniques will help reduce costs of human labor and increase the accuracy with which fish can be classified.

OBJECTIVE

The objective of the project is to train a CNN effectively enough to classify fishes into eight categories **namely Albacore Tune, Bigeye Tuna, Yellowfin Tune, Mahi Mahi, Opah, Sharks, Other (fish present but not of mentioned category) and No fish (no fish in the image)**. This is a multi-class classification problem and we need to predict the likelihood that a fish in the image is from one of these categories. We would be benchmarking the model initially using Naïve Bayes, Support Vector Machines and K-Nearest Neighbors so that we can compare it to the CNN's performance. We would implement a basic CNN and then compare its performance with respect to CNN using VGG-16 model and InceptionV3 model, CNN using data augmentation and MLP using batch normalization and dropout. To evaluate our model's performance, we would be using the confusion metrics, multi-class logarithmic loss metric, accuracy and F1-score.

DATASET DESCRIPTION

The dataset has 3777 images into 8 labels and was compiled by The Nature Conservancy in partnership with Satlink, Archipelago Marine Research, the Pacific Community, the Solomon Islands Ministry of Fisheries and Marine Resources, the Australia Fisheries Management Authority, and the governments of New Caledonia and Palau. Each image has only one fish category, except that there are sometimes very small fish in the pictures that are used as bait. The data presents some challenges like the small size of the dataset because CNN's are known to run on millions of images. Then the fish only make up a small portion of the image therefore making it difficult to identify and classify. Some of the pictures contain more than one species of fish making it difficult to classify the different species. Lastly, the images are not all the same size, they have been captured from different angles and at various times of the day with no set dimensions and borders.

LITERATURE REVIEW

Classification of fish species is an interesting topic in the field of machine learning. The process of image classification involving identification of fish species from under water videos is a challenging task. Papp et al. carried out experiments on identification of whales from videos and images after they applied various image processing techniques like segmentation, SIFT, etc. In 2010, to understand fish behaviors by studying videos from reefs Spampinato et al. used image processing techniques like edge detection, use of boundary and text features with the use of grey level histograms and grey level co-occurrence matrix. To detect clusters, use of R-CNN known as Regional convolutional neural network was done, it gave a means of detecting fish species in real time. Furthermore, to detect fishes from images a Haar classifier was proposed. A technique named as Artificial Radius Immune algorithm in combination with KNN and PCA gave good results for the fish classification problem as suggested by Rodriguez et al. To counter the practical scenarios that involve parameters such as speed of boat, clarity in background Kalman filters in combination with Gaussian Mixture Models have been used by Nguyen et al. in an attempt to detect and classify fish species. For getting better results in this challenge specifically, participants have used Faster R-CNN models and annotated training data, providing location of fish in images, with bounding box regression technique.

IMPLEMENTATION

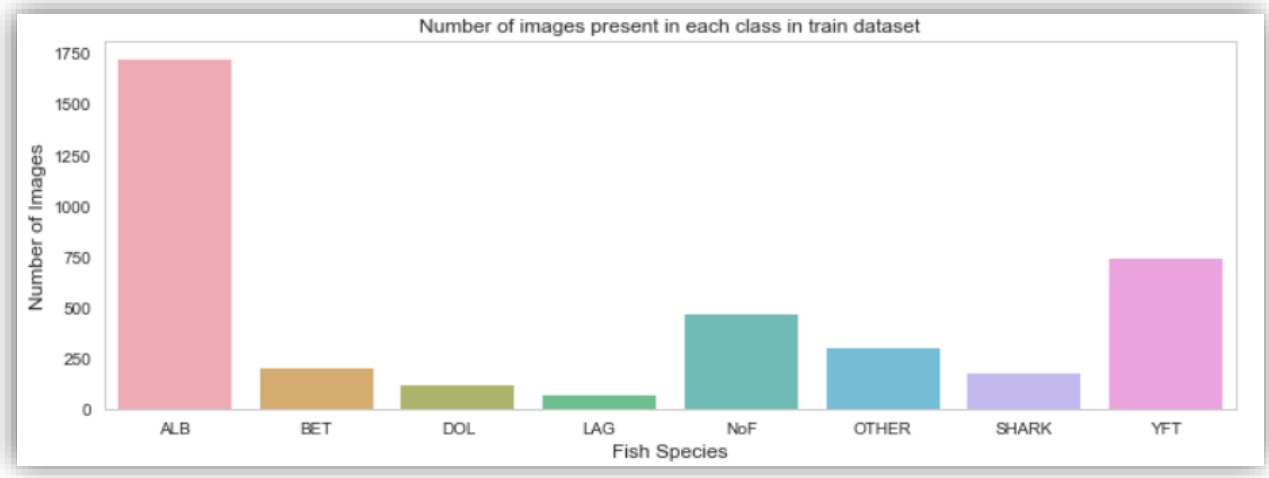
We first perform an exploratory data analysis on the dataset provided by Kaggle to get a feel for the data and know more about it.

Exploratory Data Analysis

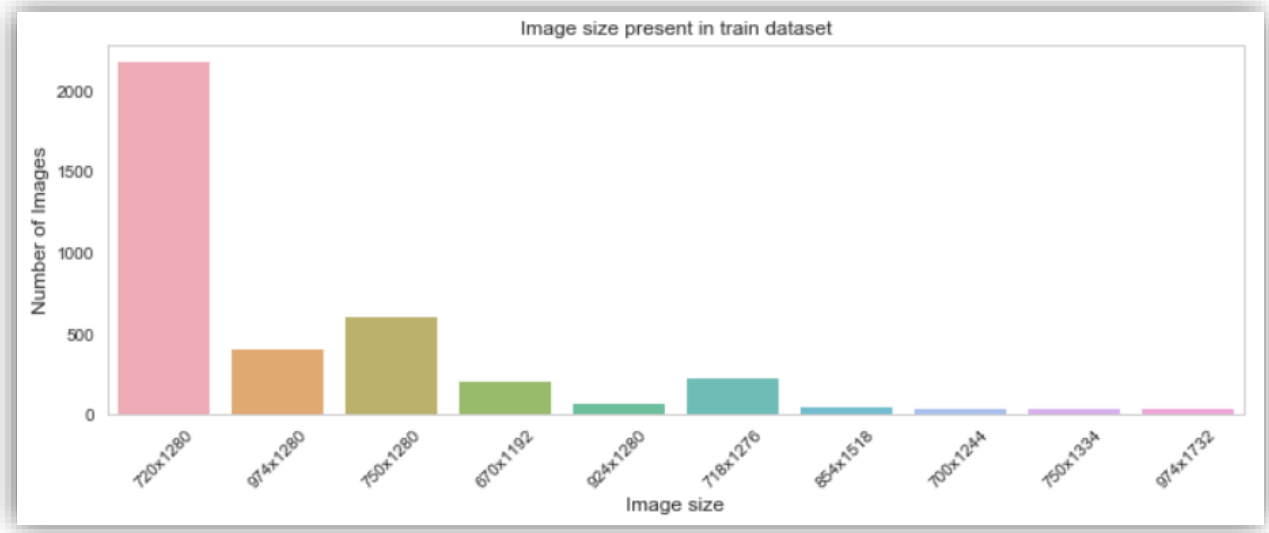
Some of the aspects of the dataset were pretty clear to us when we ran initial script to check the class distribution. As the classification problem is a multi-class classification task it is very important to check for class imbalance and we see that in the plot below fish species ALB has around 1700 images whereas some of the species like DOL and LAG have just about 100 images under their labels. This could be an issue because the classifier could be biased into predicting class having more number of images as it would get less training data to learn for classes with low number of images. We also needed to be careful about splitting our training dataset into train and validation set. But we thought a better approach would be to use KFold cross validation where we chose K as 10. This value has been selected in accordance with the value suggested by a paper written by Ron Kohavi (1995). The reason is that the lower the K is, the model is more biased but as runtime is less it is cheaper. On the contrary, larger the K is the model takes more runtime so it is expensive but less baised but can suffer from large variability. Larger K is more expensive (unless you can cleverly build it into your fitting process), less biased, but can suffer from large variability. Therefore, K=10 is the most standard and recommended value.

As said earlier we had 3777 images for our training set and 13,153 images on the test set provided by Kaggle. The test images didn't contain labels and, so we made submissions into the Kaggle competition so that our models could get evaluated. For evaluating the models Kaggle used multi-

class log loss score as the evaluation metric. That's why for test set we had presented the evaluation of model's performances in terms of log loss.

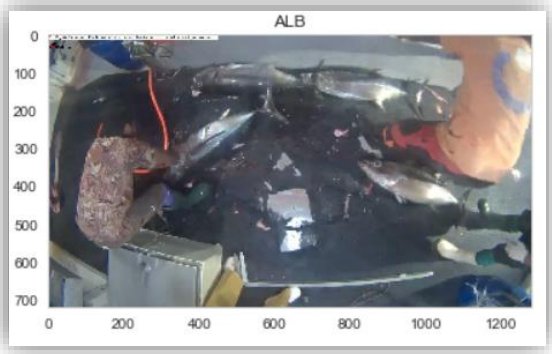


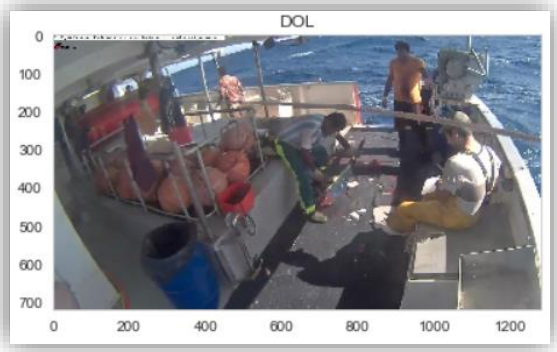
Another issue that we learn about is that the training set had images of different sizes. The most common image size found in the dataset is 720x1280 and we can see that some of the less common sizes like 700x1244 have just about 100 images. Thus, to combat this issue, we resized the images into one common size. The below plot shows us the different sizes of images present and their count in the training dataset.

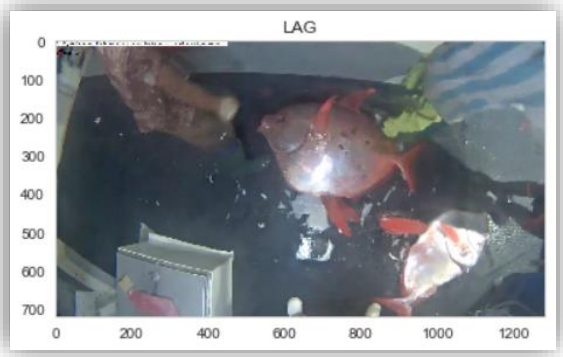


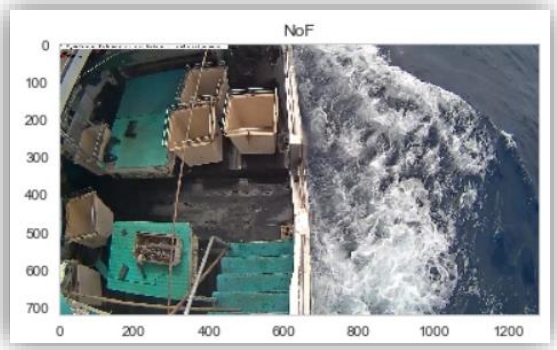
Next, we move onto the exploring how each species of fish looks like.

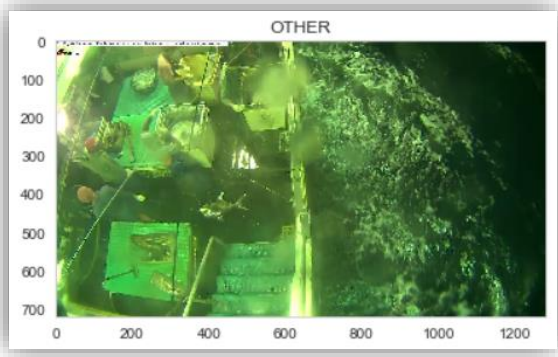
Thus, we wrote a script that displays the first image present in each category and we see the images as below:

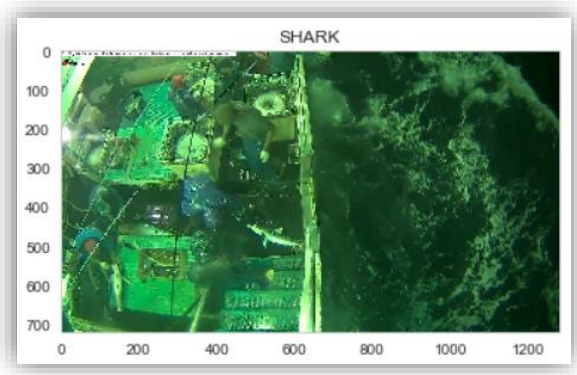












In all we found some interesting things from these images like the images are taken both during day as well as night. The background in the images contains a lot of things except fishes like boat, boatmen, ocean water, appliances present on the boat, etc. It would be great to reduce this noise and detect the location of fish. This would increase our chances for classifying the fish species correctly. Another thing, we observed is that the fish species are not distinguishable to the human eye some of the fish species are quite difficult to recognize in comparison to others.

Moreover, in the discussion forum for Kaggle we were provided with the information that there are twelve boats that have been used to capture these images. Running a clustering script, we come to know that some of the images are highly correlated to one another inside a cluster. This can be attributed to the fact that images have been captured from a continuous video and so when the images are clustered by boats, images for a particular boat may be from the same video. Our aim would be to remove such images and add boat id as an additional feature for our data.

Data Pre-processing

The first thing we did was to split our original training data into train and validation sets. Our aim was to maintain the class distribution as per the original train data and therefore we used K-Fold cross validation to negate the class imbalance problem and less training dataset size. We randomly shuffled the images as a first step for K-Fold cross validation. The next thing we did was to resize all images to one common size of 256x256. Though this degraded some of the image quality, but we were limited by our computational resources and this size was the best with which our models could perform. We also normalized our data by dividing it by 255 as we have seen feature scaling helps in the faster convergence of models and brings data on the same scale. Furthermore, we divided the data into features and labels and further on one hot encoded the categorical variables of the dataset.

Feature Extraction

One of the most interesting parts of this project is feature extraction. When dealing with images there are a few ways we can extract features from images namely: Scale Invariant Feature Transform (SIFT), is a technique that helps extract key points from images in a Nx128 matrix. But the problem is that this method does not work on Kaggle servers, hence I used ORB which is an open source library provided by opencv and is good alternative for SIFT. The features extracted

from images in the form of key points can be of the order NxM depending on the dimension returned by ORB for a feature. Another technique worthwhile exploring is Histogram of Gradients (HOG). It helps in detection of humans as well as animals in image classification. It has worked well on Kaggle image classification challenges in the past as per the posts on the discussion forum. The main aim of any of the technique discussed above is to extract most important key features and then convert them into a higher dimensional space. Once they are converted then we can employ techniques such as Naïve Bayes and SVM on these in a hope that the descriptors help us reach a feature space that is shared by image vectors belonging to the same class. These features can also be fed to complex neural nets or any other classification algorithm, but they will be useful in comparison to raw pics.

Benchmark Models

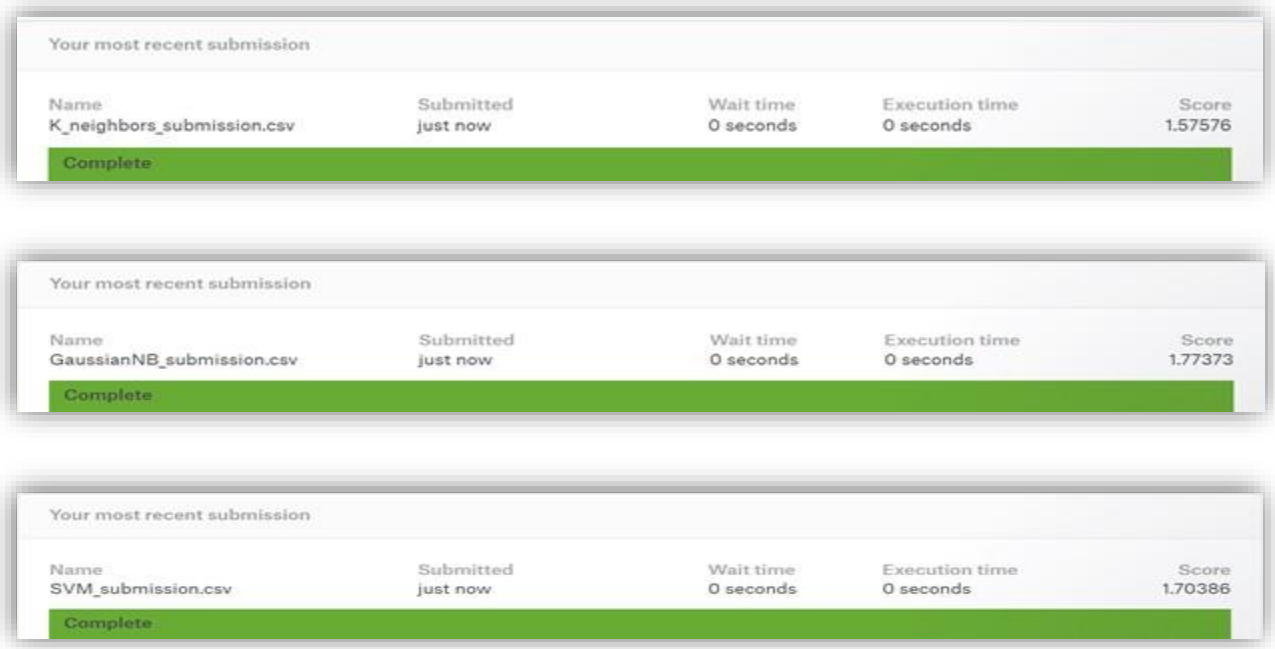
For benchmarking purposes, we used three classification algorithms namely **Naïve Bayes, Support Vector Machines and K-Nearest Neighbor Models**. For the Naïve Bayes and Support vector machine models we used **ORB** technique and detectandcompute() to detect keypoints from the images and tried to remove the unwanted features in background image. We then clustered these features so that we get a bag of visual words model on top of which we could implement Gaussian Naïve Bayes and SVM. For the K-Nearest Neighbor algorithm baseline model our aim was to extract features from the color histogram of the images. This approach brought up a flaw in the dataset, we observed that the images were taken from a finite number of boats and the background images were highly correlated with each other for images taken from a particular boat. This is not the scenario in a real world setting where there are large number of boats on which the camera is placed.

Performance Comparison of Baseline Models

	Accuracy	F1-score	Log-loss score	Log-loss score on Kaggle
Models	On Validation Set			Test set
KNN	**0.51**	**0.53**	**1.49**	**1.57**
NB	0.47	0.34	1.57	1.77
SVM	0.49	0.34	1.51	1.70

On the test set we see that KNN baseline model performs the best with a multiclass log loss score of 1.57. We see the same trend on validation dataset too where we get the least log-loss score and maximum accuracy for the KNN model. In terms of accuracy and F1-score KNN gives the best accuracy score with 0.51 and F1-score of 0.53. Therefore, **overall best baseline model is K-Nearest Neighbors model**.

Below we see the submissions for baseline models on Kaggle test set and their respective multi-class log loss score:



Experiments, Analysis, and Results

Let's first look at the **computational power and libraries** used for the project. For building the classification models we have used Windows OS, Intel Core i7 7th generation 2.7GHz processor with 8GB RAM and 256 GB SSD. The code has been written using jupyter notebook in python 3.6. In our experiments pertaining to Neural Net models we have used Keras to build our models and used tensorflow as backend for it. Sklearn libraries have been used widely for Gaussian Naïve Bayes, SVM, and K-Nearest Neighbor model but also for using classification metrics like log loss score, accuracy and F1-score. KFold cross validation and train test split have been used for splitting the data while LabelEncoder has been used to convert categorical data into numerical.

Use of numpy is done to convert the data into numpy arrays so that images can be processed whereas scipy and cv2 have been used for image processing mainly for resizing and reading the images and converting them to RGB and grayscale at times. Matplotlib and seaborn libraries have been used for plotting graphs. Libraries such as glob and shutil have been used for accessing directories and files.

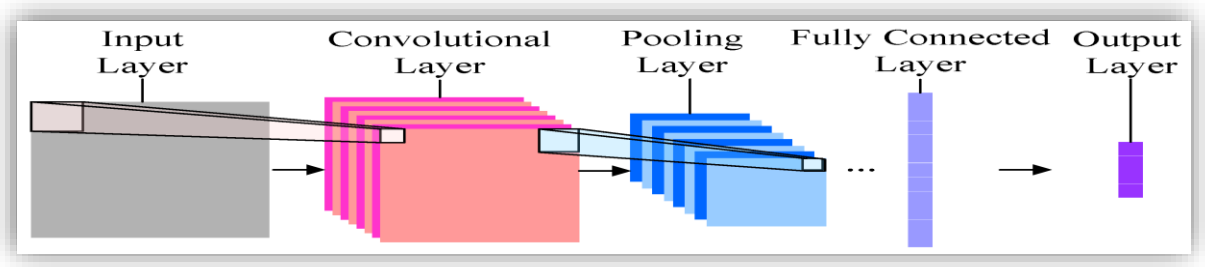
Keras libraries have been used to access sequential models, flatten, dense, dropout, batch normalization and reshape operations. Callbacks have been used in the form of early stopping, history and model checkpoint for saving the best models, getting history of accuracy and validation loss for various epochs whereas early stopping helps in stopping the training when validation accuracy starts to decrease. Maxpooling and convolution2D layers have been used to create CNN

models. VGG 16 architecture has been used using keras to train the model on a well-known architecture and ImageDataGenerator has been used for data augmentation purposes.

The major **difference between a Multi-Layer Perceptron and a Convolutional Neural Net** is that the MLP has fully connected layers and that it accepts input only in vector forms. Now, imagine if we have a 256x256 image we will have lots of features and in a MLP each node in the current layer would be connected to all nodes in the previous layer. The computational complexity would be very large. Another issue is that as the input is a feature vector instead of a matrix the spatial information regarding where each pixel is located wrt the other pixel is lost. Convolutional Neural Nets are built exactly for this purpose, they make use of the 2-D information. The layers of the CNN are sparsely connected, and they accept input as 2D matrices.

Architecture of Neural Networks

Designing the architecture of Neural Networks is very important part of Image Classification. While designing the architecture of our CNN, we need to keep in mind that the input image array should gradually get much deeper than being tall or wide. Therefore, as we progress through the layers depth should increase but the height and width should decrease. The depth increases as we add Convolution layers throughout the network but the effect of adding MaxPooling layers is that it decreases the spatial dimensions height and width.



Let's look at the **layers that help us building our CNN model**. We have an input layer, hidden layers and output layers. The hidden layers consist of a convolution layer. To build our **convolution layer**, we have a convolution window of a fixed height and width that slides vertically and horizontally on our input matrix. At each position within the image the window specifies a small portion that is a collection of pixels and we connect a hidden node to it. Such a layer of hidden nodes is called a convolution layer. For our architecture we have used a **2x2 filter** specified by kernel size value of 2.

We have used **RELU as an activation function** for our convolution layers as it leaves the positive values as it is and convert the negative values to zeros. The problem of **vanishing gradients** is common in neural networks that use sigmoid function. The reason is that in the region close to 0 and 1 in the sigmoid function the derivative tends to zero. This effect is intensified when we do back propagation. To overcome this problem, we use another activation function known as RELU.

A **filter** is a representation of weights in the form of a vector with which the input is convolved and the size of the filter always matches the size of the convolution window. We see that with each convolution layer we double the size of filters. We started from 16 filters to 32 to finally 64 filters as we want to increase the depth of our arrays. These filters help us uncover various patterns in the image. 2D filters are used for grayscale images whereas for RGB images we have a third parameter other than height and width known as depth.

For our architecture, the **stride value is one** for the convolution window meaning that the convolution window slides by 1 pixel each time. We have used padding as "same" results in padding the input such that the original input has length equal to the output.

In this case of multi-class classification, **categorical cross-entropy will be our loss function** as in the backpropagation phase while training our neural net model, the filters are updated at each epoch to take on values that minimize loss function.

Another layer which we use in our CNN model is the **MaxPooling layer**. The input to these layers is the convolution layer. When we have a dataset as complex as ours, we our bound to use many filters as more number of filters will uncover various patterns and help categorize the data, with this increases the dimensionality. As the dimensionality increases so does the number of parameters and the model can tend to overfit. Therefore, to keep the dimensionality in check we use pooling layers. We use a MaxPooling layer that's takes as input stacks of feature maps. For our architecture we have a convolution window of size 2x2 with strides of 2. With this technique the maximum value in the sliding window is chosen, thus reducing the overall dimensionality by half of the previous convolution layer. Typically, we have one Max Pooling layer for every one or two Convolution layers. This has the effect of making the dimension half of what they were in the previous layer. We see that spatial information is lost coming towards the end layer of the CNN, but our CNN model is capable of answering much more complex questions and gives us the probabilities of the object present in the image and then we can pass the output of max pooling layer by flattening it to a Dense fully connected layer to identify the object in the image with higher probabilities. These features are learnt through back propagation while training the network, but we just give our CNN model the structure that allows us to learn better on training set so that we can do a better job at classification.

In summary, we see that the **Convolution layer helps us in recognizing regional patterns** in the image and **Max Pooling helps in reducing the dimensionality** of the arrays. These arrays along with a fully connected layer at the end make up the architecture of our CNN.

Some of the design choices we use for our Neural Networks

Before training our model, we need to specify a loss function. For each model we have used **categorical cross-entropy as our loss function** as we are solving a multi-class classification problem. This loss function compares the predicted labels by the model with the true labels provided and checks if our model has performed well. As the predictions our in form of vectors because each label is one-hot encoded. The categorical cross entropy compares the two vectors (predicted and true) and returns a lower value if the vectors are same. Therefore, higher the probability of predicted and true label being the same lower will be the categorical cross-entropy loss value. We always attempt to build models that have lower categorical cross-entropy loss value.

The standard way for descending a loss function like categorical cross-entropy is gradient descent. In Keras each gradient descent method has a corresponding optimizer. As an optimizer for our Neural Net model we have used **Adam (Adaptive Moment Estimation)**. The Adam optimizer is a combination of momentum and RMSPROP optimizers. It uses a more sophisticated exponential decay that contains average and variance of the previous steps. The Adam optimizer has the following values for the parameters: learning rate = 0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, and decay=0.0. We have used the above values for the Adam optimizer.

While compiling our Neural Net model we have specified **accuracy as a metric**. Thus, we will come to know how the accuracy of the model changes during the K-Fold cross validation or training process.

Another optimization that we use is mini batch gradient descent. If our dataset would have been too small say less than 2000 data points we would have used normal batch gradient descent but since we have around 3000 data points we use mini batch gradient descent and our mini batch size is 32. Typically, we choose 64, 128, 256 as the mini batch sizes but due to less computational power in our hardware we chose 32. Typically, the cost function for mini batch gradient descent is a little noisy and this is because the gradient descent updates weights after processing each small mini batch instead of the full batch. But then too the cost function decreases as number of iterations increase. Using a size for mini batch between stochastic gradient (batch size = 1) and batch gradient descent (batch size=whole dataset) is the best as it incorporates the use of vectorization due to which training is faster and time per iteration is also less as data points are processed in a batch and then weights for gradient descent our updated.

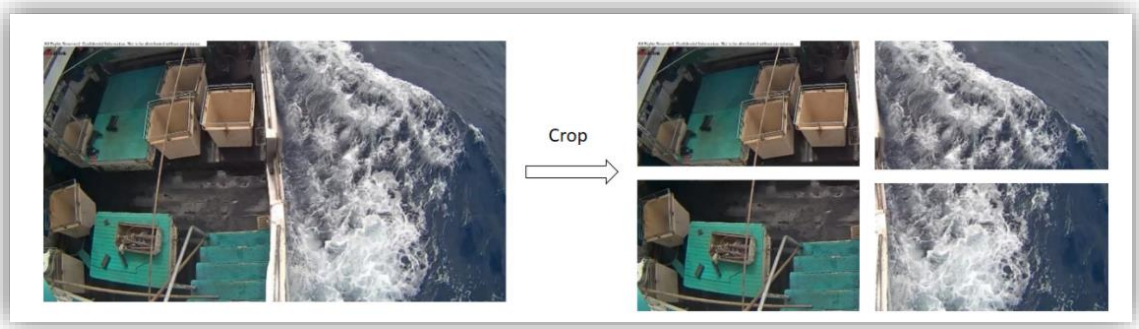
The **number of epochs have kept low around 5 or 10** as training time is in hours we didn't get the time to increase epochs to bigger value. This would have taken a lot of time and computational resources.

Data augmentation

We have also used the **data augmentation technique** in combination with our CNN model. Ideally, we want our model to classify the object present in the image correctly irrespective of the size of the object, angle at which the object is present in the image or the position at which the object is present in the image be it to right, left, top or down portion if the image. Therefore, our model should be scale invariant, rotation invariant and translation invariant. In an ideal scenario,

we can just add images to our training set with different rotations of the object and make our model rotation invariant but sometimes its not feasible to get more training data. In such cases and for the scenario in this project where training data is less we make use of data augmentation technique where we generate images that have objects rotated, horizontally or vertically flipped, objects at every position in the image be it right, top, down, left etc. These new augmented images are generated using Keras's ImageDataGenerator and it also helps in reducing overfitting as the model is trained on additional new images and therefore should generalize well on test set. A reason for using data augmentation technique was to counter the fact that we had less training data.

For e.g. below we can see that an image is cropped into 4 images and added to dataset.

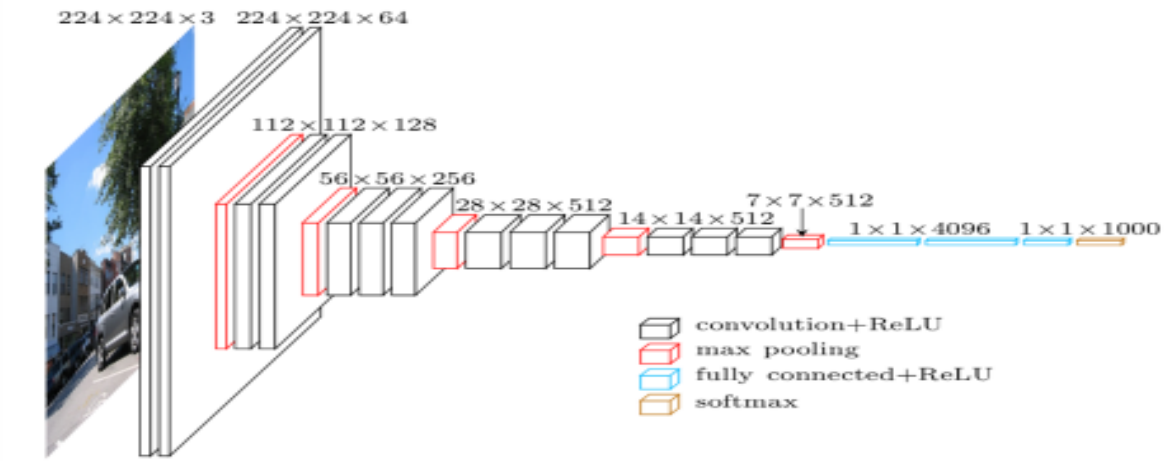


VGG-16 architecture

We have made use of **VGG-16 architecture** in our CNN model. The reason for doing this was designing a CNN architecture for image classification is not a trivial task. Building a good CNN model involves fine tuning of parameters such as weights and bias but also fine tuning hyper-parameters.

Hyper-parameter tuning

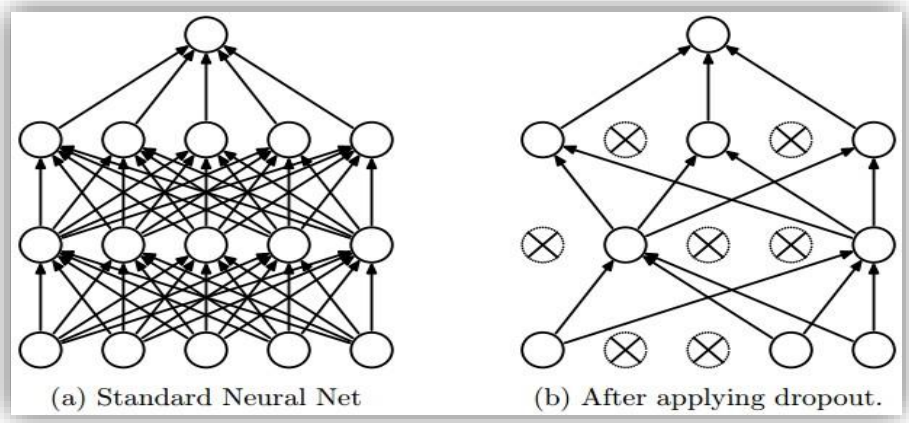
It involves tuning parameters such as size of convolution windows, number of filters, padding, stride, loss function, optimizer, setting alpha that determines how our parameters evolve, the number of iterations on gradient descent, number of hidden layers, number of hidden units, choice of activation function be it RELU/ tanh/ sigmoid, etc. and many more. The VGG-16 architecture available to us through Keras has been developed through careful experimentation, countless architectures and extensive hyper-parameter tuning. This architecture has been trained on the ImageNet database for weeks on GPU's. Therefore, we adopt this architecture and use if for our own classification purposes on our dataset. For doing so, we take the VGG-16 model trained on ImageNet database and transfer its learning to our model instead of training our CNN from scratch. This technique is called **Transfer Learning**. The first few convolution layers are aimed at detecting general features like edges, geometrical shapes, blobs of colors, etc.



These layers can therefore be used on any database. It's the end convolution layers where more specific features are discovered. Therefore, we remove the final layers that are more specific in the VGG-16 architecture and keep the earlier layers. Finally, we can add a dense layer in the model to suit our purposes depending on our dataset. The above image shows pretrained VGG architecture.

Dropout Regularization

It is very easy to overfit on the training set we have for this project. The dataset size is small and there is a high-class imbalance. To consider the effects of overfitting and prevent our model from doing so we made use of the **Dropout Regularization** technique. Sometimes one part of a neural net can have large weights and it ends up dominating all the training while another part doesn't contribute and so does not get much trained. To solve this issue, we give a probability say 0.5, this means that sometimes with probability 0.5 the dominant part will turn off so that the other part can get well trained and vice versa. Therefore, as we go through the epochs we randomly turn off the nodes with some probability and the other nodes pick up the slack for the turned off nodes. Therefore, what we really do is spread the weights over the network and reduce the dimensionality and thus overcome overfitting. Below we see the image of a network with and without dropout.



Batch Normalization

For our Multi-layer perceptron models, we see that we use a technique called as **Batch Normalization**. When we trained the logistic regression model in the second assignment we saw that we normalized input features before passing it to the sigmoid function. Here in MLP we normalize the output of activation functions before passing it further to the next layer. As the learning process became faster and the model converged much faster when normalized the inputs the same intuition is carried to batch normalization. The training process of the MLP gets speeded up because of batch normalization. Also, batch normalization gives a slight effect of regularization as each mini batch is scaled by mean/variance and some noise is added to activation functions of the hidden layer. Therefore, this technique has a slight regularization effect, but it should not be used for regularization purposes.

To make sure we do not overfit over the course of epochs while training our model we use **Early Stopping**. In Early stopping technique we want the just right point till which the test error decreases and then starts to increase. At this point the training as well as test error are low and we neither underfit not overfit. At this moment we want to stop the training and thus this technique is known as Early stopping.

In our classification models we see that we have used a **Softmax function** instead of sigmoid function. This is because in the case of linear regression in a multi-class classification setting when output for the model are negative numbers calculating probability gets tricky. In order to deal with negative numbers, we raise the scores as powers of e. Therefore, formally to find out the probability of a class we raise e to the power score of the class we get from our linear model and divide it by the summation of e to the power scores for all the classes.

Models based on Neural Nets

For this project we have built six neural net models namely: Multi-layer Perceptron having two fully connected layers with batch normalization and another model having batch normalization and dropout. Apart from these two models using convolutional neural network we have four models: a simple basic CNN model, a CNN model that uses data augmentation technique and a CNN model based on VGG-16 architecture using data augmentation technique. Our final model is a CNN model based on InceptionV3 architecture combined with data augmentation technique.

Basic simple CNN Model

Model Summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_31 (Conv2D)           (None, 256, 256, 16)      208
_________________________________________________________________
max_pooling2d_31 (MaxPooling (None, 128, 128, 16)      0
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 128, 128, 32)      2080
_________________________________________________________________
max_pooling2d_32 (MaxPooling (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 64, 64, 64)        8256
_________________________________________________________________
max_pooling2d_33 (MaxPooling (None, 32, 32, 64)        0
_________________________________________________________________
flatten_11 (Flatten)         (None, 65536)             0
_________________________________________________________________
dense_21 (Dense)             (None, 512)               33554944
_________________________________________________________________
dropout_11 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_22 (Dense)             (None, 8)                 4104
=================================================================
Total params: 33,569,592
Trainable params: 33,569,592
Non-trainable params: 0
_________________________________________________________________
13153/13153 [==============================] - 407s
```

Above we basically see the model architecture of basic CNN model, the parameters at each layer, different layers like convolution and max pooling layers. We get to know the total, trainable and non-trainable parameters in our model. Also, it's depicting the output shape after each layer.

Simple CNN model with Data Augmentation technique

Model Summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_31 (Conv2D)           (None, 256, 256, 16)      208
_________________________________________________________________
max_pooling2d_31 (MaxPooling (None, 128, 128, 16)      0
_________________________________________________________________
conv2d_32 (Conv2D)           (None, 128, 128, 32)      2080
_________________________________________________________________
max_pooling2d_32 (MaxPooling (None, 64, 64, 32)        0
_________________________________________________________________
conv2d_33 (Conv2D)           (None, 64, 64, 64)        8256
_________________________________________________________________
max_pooling2d_33 (MaxPooling (None, 32, 32, 64)        0
_________________________________________________________________
flatten_11 (Flatten)         (None, 65536)             0
_________________________________________________________________
dense_21 (Dense)             (None, 512)               33554944
_________________________________________________________________
dropout_11 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_22 (Dense)             (None, 8)                 4104
=================================================================
Total params: 33,569,592
Trainable params: 33,569,592
Non-trainable params: 0
_________________________________________________________________
13153/13153 [==============================] - 407s
```

This is the same architecture as the previous model. We can see that in the output we have a dense layer of 8 nodes for our multi-class classification. In the input we have a shape of 256x256x3 signifying we have color images of size 256x256. We see the parameters explode as depth of the neural net increases, but the spatial dimensions decrease as you can see in the output shape of each layer. Our model increases in depth but loses spatial dimensions.

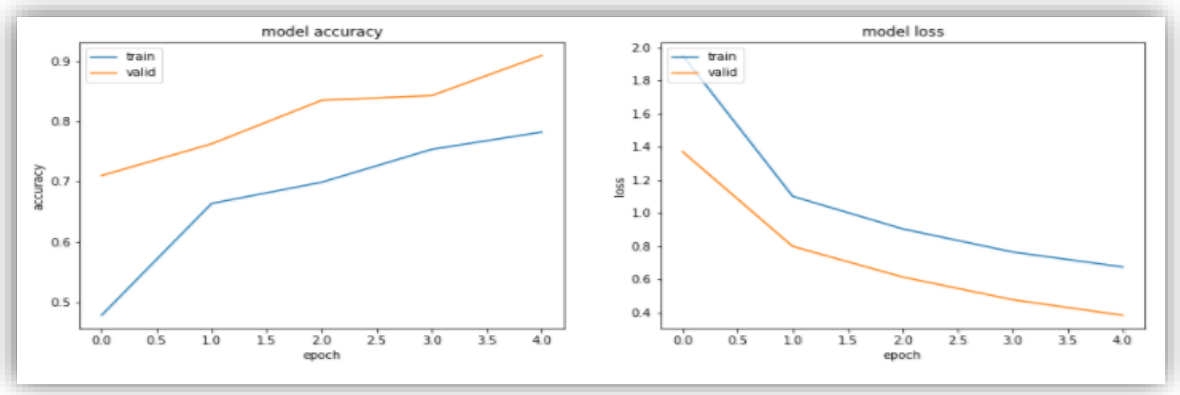
CNN model with VGG-16 architecture and Data Augmentation technique

Model Summary:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 256, 256, 3)	0
block1_conv1 (Conv2D)	(None, 256, 256, 64)	1792
block1_conv2 (Conv2D)	(None, 256, 256, 64)	36928
block1_pool (MaxPooling2D)	(None, 128, 128, 64)	0
block2_conv1 (Conv2D)	(None, 128, 128, 128)	73856
block2_conv2 (Conv2D)	(None, 128, 128, 128)	147584
block2_pool (MaxPooling2D)	(None, 64, 64, 128)	0
block3_conv1 (Conv2D)	(None, 64, 64, 256)	295168
block3_conv2 (Conv2D)	(None, 64, 64, 256)	590080
block3_conv3 (Conv2D)	(None, 64, 64, 256)	590080
block3_pool (MaxPooling2D)	(None, 32, 32, 256)	0
block4_conv1 (Conv2D)	(None, 32, 32, 512)	1180160
block4_conv2 (Conv2D)	(None, 32, 32, 512)	2359808
block4_conv3 (Conv2D)	(None, 32, 32, 512)	2359808
block4_pool (MaxPooling2D)	(None, 16, 16, 512)	0
block5_conv1 (Conv2D)	(None, 16, 16, 512)	2359808
block5_conv2 (Conv2D)	(None, 16, 16, 512)	2359808
block5_conv3 (Conv2D)	(None, 16, 16, 512)	2359808
block5_pool (MaxPooling2D)	(None, 8, 8, 512)	0
flatten_1 (Flatten)	(None, 32768)	0
dense_1 (Dense)	(None, 512)	16777728
batch_normalization_1 (Batch	(None, 512)	2048
dropout_1 (Dropout)	(None, 512)	0
dense_2 (Dense)	(None, 8)	4104

Total params: 31,498,568
Trainable params: 16,782,856
Non-trainable params: 14,715,712

We see that initially for every two convolution layers we have a max pooling layer but after two blocks the convolution layers increase to three per block. We have also introduced dropout to take care of overfitting and dimensionality reduction. Batch normalization has been applied so that the training process can speed up and activation functions are on the same scale for intermediate hidden layers.



From the above plots we see the how accuracy and log-loss score change over the course of the epochs. Looking at the second plot of the two plots shown above for model log loss score we can see that there is no overfitting on the data as train loss was higher than the validation loss. Also, we see in the plot of Accuracy for train and validation set over epochs, as we can see that accuracy on validation set is higher more than 70% over all epochs compared to the accuracy on training set which varied in the range of between 50% to 75% over all epochs.

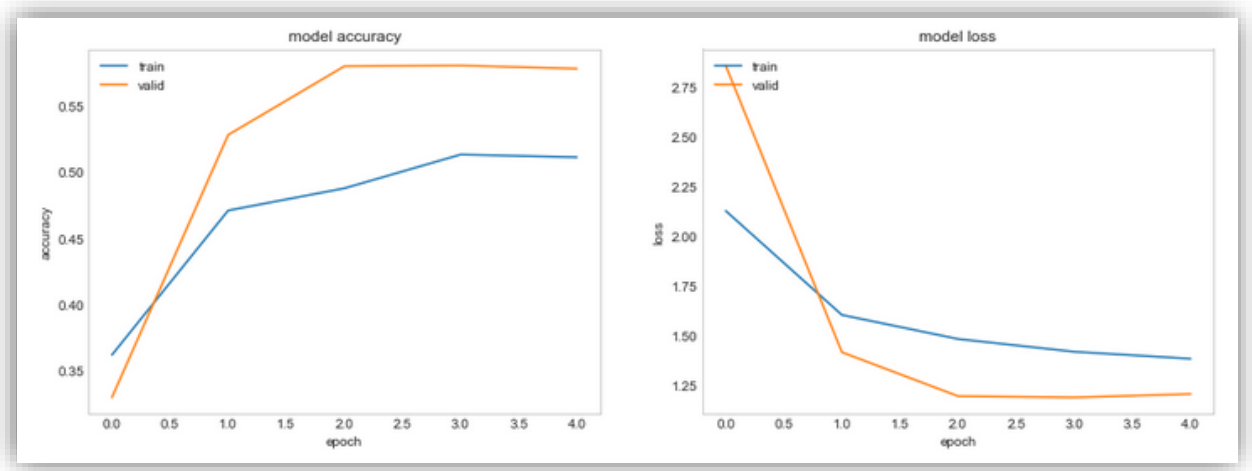
CNN model with Inception V3 architecture and Data Augmentation technique

Model Summary: As Inception V3 had a lot of layers I have pasted just the end part of the model summary

activation_750 (Activation)	(None, 6, 6, 384)	0	batch_normalization_757[0][0]
activation_751 (Activation)	(None, 6, 6, 384)	0	batch_normalization_758[0][0]
batch_normalization_759 (BatchNo	(None, 6, 6, 192)	576	conv2d_752[0][0]
activation_744 (Activation)	(None, 6, 6, 320)	0	batch_normalization_751[0][0]
mixed9_1 (Concatenate)	(None, 6, 6, 768)	0	activation_746[0][0] activation_747[0][0]
concatenate_16 (Concatenate)	(None, 6, 6, 768)	0	activation_750[0][0] activation_751[0][0]
activation_752 (Activation)	(None, 6, 6, 192)	0	batch_normalization_759[0][0]
mixed10 (Concatenate)	(None, 6, 6, 2048)	0	activation_744[0][0] mixed9_1[0][0] concatenate_16[0][0] activation_752[0][0]
flatten_8 (Flatten)	(None, 73728)	0	mixed10[0][0]
dense_15 (Dense)	(None, 512)	37749248	flatten_8[0][0]
batch_normalization_760 (BatchNo	(None, 512)	2048	dense_15[0][0]
dropout_8 (Dropout)	(None, 512)	0	batch_normalization_760[0][0]
dense_16 (Dense)	(None, 8)	4104	dropout_8[0][0]

Total params: 59,558,184
Trainable params: 37,754,376
Non-trainable params: 21,803,808

Inception V3 is a model that was trained by Google on ImageNet database. We used that pre-trained architecture and its weights and transfer that learning on to our classification problem,



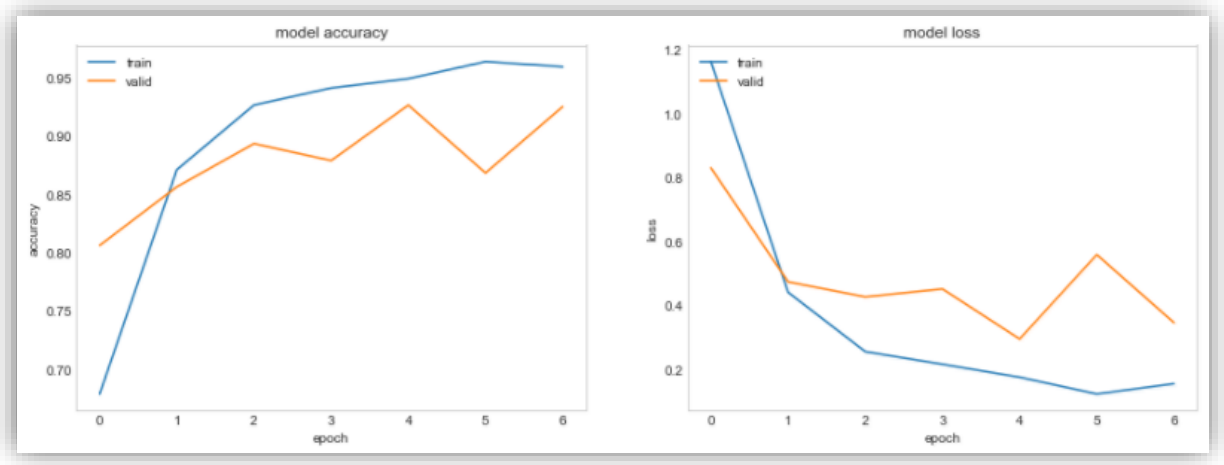
In the above plot for model log loss we can see that there is no overfitting on validation set and the model is able to generalize well. Also, we see that accuracy is higher on validation set that train set over the epochs.

Multi-Layer Perceptron with Batch Normalization

Model Summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 196608)            0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              805310464
_________________________________________________________________
batch_normalization_1 (Batch (None, 4096)              16384
_________________________________________________________________
dense_2 (Dense)              (None, 512)               2097664
_________________________________________________________________
batch_normalization_2 (Batch (None, 512)               2048
_________________________________________________________________
dense_3 (Dense)              (None, 8)                 4104
=================================================================
Total params: 807,430,664
Trainable params: 807,421,448
Non-trainable params: 9,216
```

For this model architecture we have three fully connected dense layers, but we do not have dropout. We implement this model to see the effects of dropout.



In the plot between **epochs Vs Model loss** as shown above, we see that on some epochs there is a lot of difference around 0.5 in log loss score between train and validation set. This indicates overfitting but to be sure we would run the model on more number of epochs. The model stopped after 6 epochs due to Early Stopping and, so it means the difference between loss on validation and train set would have increased. But since the patience parameter was set to 2 it may be the case that log loss score was not improving, and the model stopped. Some more experimentation was further to fully understand what is happening. In the other plot between Accuracy vs epoch, we can see train set attaining the accuracy even more than 95%, around 99%, while validation set was around 90%, this indicates overfitting in the model as confirmed from the other plot between model loss vs epochs.

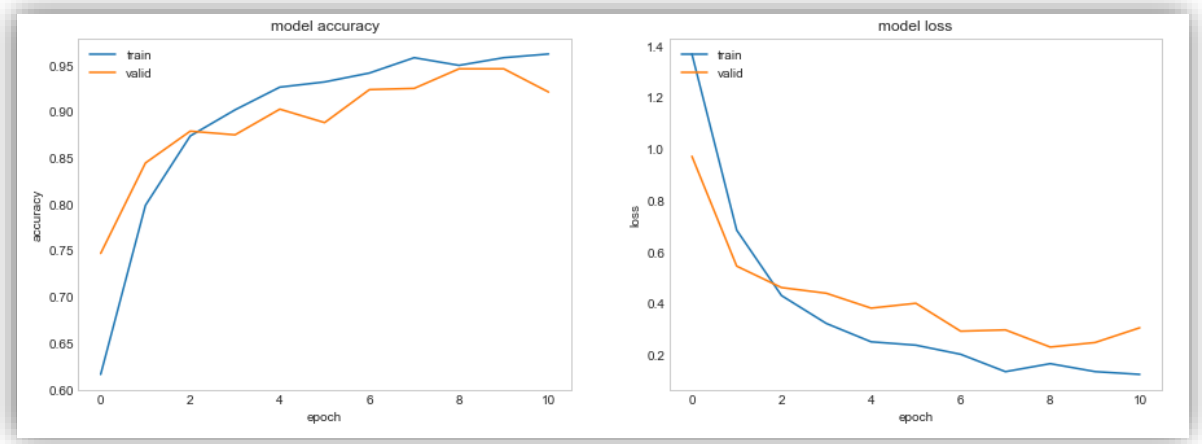
Multi-Layer Perceptron with Batch Normalization & Dropout

Model Summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_2 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_2 (Dense)              (None, 4096)              12587008
_________________________________________________________________
batch_normalization_1 (Batch (None, 4096)              16384
_________________________________________________________________
dropout_1 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_3 (Dense)              (None, 512)               2097664
_________________________________________________________________
batch_normalization_2 (Batch (None, 512)               2048
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_4 (Dense)              (None, 8)                 4104
=================================================================
Total params: 14,707,208
Trainable params: 14,697,992
Non-trainable params: 9,216
_________________________________________________________________
```

We have in all three fully connected dense layers in our MLP model architecture. To understand see the effects of dropout we had a MLP with and without dropout. Below plots we see that to

some extent overfitting has been reduced by using this model. This is because the curves are near to each other with less difference of log loss between them. Still at epoch 10 we see the log loss score on validation set rise and, so it may indicate overfitting. To combat this case, we may use L-2 regularization or try increasing the dropout probability. We can also increase the number of epochs and remove early stopping to see how the curves pan out.



Performance Comparison of Neural Network Models

Models	Validation Set			Test Set
	Log Loss score	Accuracy	F1	Log Loss score
Simple CNN model	0.55	0.69	0.70	1.46
CNN with Data Augmentation	0.35	0.84	0.77	1.23
VGG-16 model	**0.23**	**0.93**	**0.94**	**1.14**
Inception V3 model	0.31	0.87	0.78	1.20
MLP with Batch Normalization	0.48	0.73	0.69	1.30
MLP with Batch Norm & Dropout	0.44	0.78	0.82	1.25

From the above table we can see that on the test set provided by Kaggle, the best performing model was CNN model with VGG-16 architecture with a multi-class log-loss score of 1.14 on the test set. On the validation set also this model was able to obtain the least log loss score and maximum accuracy of 0.23 and 0.93 respectively. Also, we see that when we used data augmentation with the simple CNN model, its performance in terms of log-loss score increased from 0.55 to 0.35(less log loss score is better). For the case of Multi-Layer perceptron model, we can see that the model with dropout performed better than the model without it because it took care of overfitting and generalized well. The log-loss score decreased from 0.48 to 0.44 when we used dropout on MLP.

If we compare the best of baseline models with best of Neural net models, the comparison would be between KNN and CNN model with VGG-16 architecture. On the test set provided by Kaggle

CNN model with VGG-16 architecture performed best with a multi-class log-loss score of 1.14 in comparison to 1.57 from KNN. The CNN model with Inception V3 architecture performed the second best on test data with a multi-class log-loss score of 1.20.

Another important thing we observe is that models on pretrained architecture (VGG-16 CNN) performed better and transfer learning technique was successful. Designing a Neural Net is an art and required lot of hard work and experimentation, computation power, memory and extensive hyper-parameter tuning. Role of K Fold Cross Validation was quite important in our project as there was a high-class imbalance. The Winning Model on the Kaggle competition implemented a Faster R-CNN model with VGG-16 as the feature extractor and it used bounding box regression technique. It required several hours of training on GPU. It had a score of multi-class log loss score of 0.27 on the leaderboard.

Below we see the submissions for Neural Network models on Kaggle test set and their respective multi-class log loss score:

CONCLUSION

Overall, we are quite satisfied with the performance of our convolutional neural network models as they achieved a decent log loss on the validation set. This was a unconventional and non-trivial challenge by Kaggle wherein the images contained a lot of background noise in the form of water, boat, fishermen, fishing tools etc. As the images were captured from a camera placed on the boat, some of the images were blurry and shaken as the boat was in motion. Considering that we had to reduce the image size to 256x256 in respect to the most common size of 720x1280 was a huge setback. This made it more challenging for our CNN models to classify the species of fish correctly. As said before, due to the static position of cameras on the boats much of the background in the images were similar. This affected the baseline K-NN model which showed an accuracy of 53%. As the species of fish occupied a small area in the image the K-NN model classified the images wrt the background in the images. This revealed a flaw of the dataset as in a real world setting there would be numerous boats on which the camera would have been placed and the background would have been different. The only way to overcome this issue was to find a way to detect fish in the image. For this an annotated training dataset could be used which had boxes made around the fishes or we could annotate the fish position by marking the head and tails of the fish. These are various approaches over which bounding box regression technique could be applied.

Due to limited time on our hands we were not able to experiment with RESNET architectures. The results would have been different for sure, but would they have been better is a question unanswered. Another approach that we so wanted to implement, and we will as part of our future work, was the bounding box regression technique. A participant in the competition had annotated the training data with the location of fish. The participants who have implemented the bounding box technique for object detection and localization on the annotated training dataset have a log loss which is much (around 0.4) better than on validation set. Moreover, the different sizes of images were an evident problem from the start. Once we resized our images to 256x256, the models were able to perform whereas if we tried a size greater than this, the computational power of the laptop would be a major hindrance and we would get a resources exhausted error. Due to a larger image size, the images would require more computational and memory requirements at each layer of the convolutional neural network because a more feature vectors would be generated at each layer. Since we knew that the image size was creating a problem for us we transformed the images to grey scale images. Another problem due to image size was that our CNN could not be of a larger depth because that would again require more computational resources. The major limitation we faced was the configuration of our machine. We had an Intel core i7, 7^{th} gen 2.7GHz processor with 8 GB RAM and 256 GB SSD. This computational and memory resources were incapable of handling CNN with higher depths. The fully connected CNN's took about a complete 24 hours to run whereas the VGG-16 model with pretrained weights took longer. We had to lower the number of epochs and our batch size to get the performance for our CNN's.

FUTURE WORK

In future we would like to run this project using Amazon Web Services with high computation and memory resources. We would love to test our models using the bounding box regression techniques on annotated data. We would also love to explore Faster R-CNN models. Overall, we feel there is still a lot to learn from this project. Image classification using Convolutional Neural Nets is a vast topic and learning and experimenting everything in one semester is not possible. There are a few approaches on Kaggle discussion forum related to this topic that are worthwhile looking into like for example as the images have been taken both during day and night, they should have processed separately so that the model performs better. Another approach we would like to explore is extract features from images using SIFT and PCA.