Sound Event Detection (SED): Preventing Deforestation using Convolutional Neural Networks (CNNs)

Bhavy Mital (Grand), Zwe Wint Naing (Valentine), Ismam Al Hoque
Singapore University of Technology and Design

December 8, 2019

Abstract

Climate changes pose as one of the greatest challenges of humanity and 20% of climate change is attributed to deforestation. 80% of timbers in Amazon are illegally logged and a solution is urgently required.[6] We as students wanted to do a project which will look deeper into this problem and come up with a solution to reduce/prevent further damage to the remaining forests. In this report we describe: how we prepared our dataset to detect logging of trees; pre-processing of our dataset so that it can be used by models as input for classification; look deeper into the theory of Convolutional Neural Networks (CNNs); look into some of the commonly used CNN models for image classification; present results from different models in a controlled environment.

Keywords

Machine Learning, Deforestation, Scaper, Librosa, Synthetic Data, Convolutional Neural Networks, Keras, Tensorflow, Data Augmentation, Spectrogram, VGG, DensNet, ResNet, Inception, Exception

Disclaimer

The idea of this project is inspired by another project run by a not-for-profit called Rainforest Connection. While doing this project and writing this report, we got a lot of help from many online resources which are all cited on the references page.

Source Code

For reference, all of our source codes are uploaded in a GitHub repository and can be found at this link.

Contents

1 Introduction **1**

2 Data Collection and Dataset **1**

3 Data Pre-Processing **2**

4 Convolutional Neural Networks (CNNs) **2**
4.1 Convolutional Layer . . . 3
4.2 Pooling Layer . . . 3
4.3 Flattening Layer . . . 4
4.4 Fully Connected Layer . . . 4
4.5 Dropout Regularization . . . 4
4.6 Batch Normalization . . . 5
4.7 Different Kinds of Classification . . . 5
4.8 Activation Functions (Sigmoid vs Softmax) . . . 6
4.9 Loss Functions (Binary Cross-entropy vs Categorical Cross-entropy) . . . 8
4.10 Optimization Algorithm . . . 9

5 Models **9**
5.1 Basic CNN Model . . . 9
5.2 VGGnet . . . 9
5.3 ResNet . . . 10
5.4 DenseNet . . . 11
5.5 Inception Net . . . 12
5.6 Xception Net . . . 14
5.7 Custom Network . . . 14

6 Results **15**
6.1 Background of the tests . . . 15
6.2 Results for Binary Classification . . . 15
6.2.1 Table of results for Binary CNNs . . . 15
6.2.2 Basic CNN Binary Classification . . . 15
6.2.3 VGGnet Binary Classification . . . 17
6.2.4 ResNet Binary Classification . . . 19
6.2.5 DenseNet Binary Classification . . . 22
6.2.6 Inception Net Binary Classification . . . 25
6.2.7 Xception Net Binary Classification . . . 27
6.2.8 Custom Model for Binary Classification . . . 30
6.3 Results for Multi-Class Classification . . . 32
6.3.1 Basic CNN Multi-Class Classification . . . 32
6.3.2 VGGnet Multi-Class Classification . . . 33
6.3.3 ResNet Multi-Class Classification . . . 35
6.3.4 DenseNet Multi-Class Classification . . . 37
6.3.5 Inception Net Multi-Class Classification: . . . 39
6.3.6 Xception Net Multi-Class Classification . . . 41
6.3.7 Custom Model for Multi-Class Classification . . . 43

7 Conclusions **44**

8 Future Works **45**

9 References **46**

1 Introduction

Only 17% of the world's forests are protected. The rest are subject to deforestation, which contributes to approximately 10% of global Greenhouse Gas emissions as vegetation is cut, burned or decayed. While some deforestation is the result of expanding agriculture or urban developments, most of it comes from the logging industry. Clear-cutting, which has a particularly ruinous effect upon ecosystems and the carbon they sequester, remains a widespread practice across the world.[14]

Forest rangers cannot feasibly protect and look after an exposed stretch of forests. This creates an opportunity for illegal logging activity and a solution for detection and deterrence is urgently required. Providing tools for tracking deforestation can help forest rangers in their job, provide valuable data for informing policy-makers, as well as be a tool for law enforcement to find where deforestation may be conducted illegally.

Satellite images can be used to track how much forest has been cut, but it cannot be used to trace when it happens.[11] A not-for-profit called Rainforest Connection, lead by Topher White, has devised an effective and resourceful solution. Old mobile phones are collected, equipped with solar panels and placed in the branches of trees where they listen for sounds of logging trucks and chainsaws. One mobile phone can detect illegal logging over a kilometer away, protecting over 300 hectares of rainforest and preventing 15,000 tons of CO2 release — more than what 3000 cars release in a year.[6][14]

For this project, we tried to create a classifier to detect sounds of chainsaw and logging, similar to the one present in the phones of Rainforest Connection, to prevent any act of deforestation. This classifier can also be used in other settings for other sound event detection (SED) problems but for this report, we will only be focusing on deforestation.

To prepare our dataset, we will be using Scaper which is a library written in python for audio synthesis and augmentation. We will also be discussing our considerations while choosing audio files for the synthesis and augmentation process. We will also be using Librosa which is a python package for music and audio analysis. We will be using this package to create spectrograms of our audio data-set to feed into our neural networks for classification.

To construct our neural networks, we will be using Keras, which is a python deep-learning library. It'll be used to construct Neural Networks using API calls. We'll be running Keras on top of Tensorflow, which is an open-source library to develop and train ML models.

For this project, since we will be classifying spectrograms of our audio files, we will be using Convolutional Neural Network (CNN) to train our model. We will be exploring multiple variations of CNNs to find the best one for our use case.

2 Data Collection and Dataset

Since we live in Singapore, where we don't see trees getting cut down on a frequent basis, it is difficult for us to collect real data from the source itself. Additionally, we could not find a clean and usable data from the internet. Hence, we decided to create our own synthetic dataset.

To create this synthetic dataset, we will be using the Scaper Python Library. Scaper creates new soundscapes by combining and transforming a set of existing audio files, which is referred to as the source material. By combining/transforming the source material in different ways, Scaper can create an infinite number of different soundscapes from the same source material. The source material is comprised of two groups of files[15]:

- **Background files:** used to create the background of the soundscape, and should contain audio material that is perceived as a single holistic sound which is more distant, ambiguous, and texture-like (e.g. the "hum" or "drone" of an urban environment, or "wind and rain" sounds in a natural environment). Importantly, background files should not contain salient sound events.
- **Foreground files:** used to create sound events. Each foreground audio file should contain a single sound event (short or long) such as a car honk, an animal vocalization, continuous speech, a siren or an idling engine. Foreground files should be as clean as possible with no background noise and no silence before/after the sound event.

For our background audio files, we went to sites like YouTube and SoundCloud to collect a variety of forest recordings. Our current data set includes a few typical day time forest recordings, a few night time forest recordings, a few recordings of the forest during rain and a few recordings of the forest near a river. We could also include many more background-sounds to this but we chose to use only these few for now. We decided only to use 30 minutes of each of these recordings. Total our background audio files dataset is around 5-6 hours long.

For our foreground audio files, we again went to sites like YouTube and SoundCloud to collect a variety of chopping and chainsaw recordings. We could also include a variety of more foreground sounds such as tree falling and logging trucks, but for simplicity's sake, we chose to go with these 2 for now. Each of these recordings is 10 minutes long.

Important Note: All these audio files from YouTube and SoundCloud may or may not be synthetic sounds. So it may yield very different results if we were to records all these sounds on our own from the source itself. But due to our geographic and resource constraints, we were not able to do so. We may consider recording our own sounds in the future.

Once our foreground and background audio files were ready, we ran our Scaper code to generate our dataset. Each of our generated audio files is 10 seconds longs. We generated two types of audio files, one with no foreground sounds added and another one with foreground sounds added. We generated our soundscapes with the foreground in two stages. First, with chainsaw sounds only and the next with chopping sounds only. The Scaper also generates .txt files for each soundscape which details the different events used in the soundscape. Later we will be using these .txt files to label our dataset for our models.

While running our Scaper code, we realized that it was taking a very long time when there were multiple events to be added to a soundscape. So we decided to split our background and foreground audio files into 30 seconds long files using Pydub python library, which is commonly used to manipulate audio with a simple and easy high-level interface. This significantly improved the time Scaper took to generate individual soundscapes with multiple events. We have also set a limit to a maximum of five events in a single soundscape.

3 Data Pre-Processing

Once our dataset was ready, we had two possible approaches we could take to proceed with the project:

1. Use our audio files from the dataset as it is and feed it into a model. The issue with this approach is that it will require much more research on how to pre-process our data and we'll have to find an appropriate model to be used with this pre-processed data for high accuracy and efficiency.
2. Convert audio files to spectrograms which, then, we can feed into some custom CNN models. This approach is much easier as there has already been a lot of research done in the area of image processing and classification.

We decided to go with the second approach and convert all our 10 seconds audio files to a 200x200 pixels size spectrogram. We used the Librosa python package to do the conversion.

Important Note: In the future, before doing this step, we could also use Librosa to do additional data augmentation to increase the size of our dataset but for now, we are satisfied with the size of our current dataset.

Once all our dataset is converted into a spectrogram, we then used Keras to convert each of our images to an array of shape (200, 200, 3). Now it is time to label our dataset. We extracted the labels from .txt files which were generated by scarper earlier during the dataset creation.

We then split our dataset and labels into three parts. Out of 3000 data points, we separated 100 of them for testing. The rest of the dataset was split in the ratio of 80:20 where 80% will be used for training and 20% for validation. Together the training and validation data will be our input for the neural network and the 100 we separated earlier will used for evaluation once our model is trained.

4 Convolutional Neural Networks (CNNs)

We will be using **Convolutional Neural Networks (CNNs)** to go through our dataset and classify them with different labels.

Convolutional Neural Networks (CNNs) are made up of neurons that have learnable weights and biases. Each neuron receives some inputs, performs a dot product and optionally follows it with a non-linearity. The whole network still expresses a single differentiable score function: from the raw image pixels on one end to class scores at the other. Finally, they will have a loss function (e.g. Sigmoid/Softmax) on the last (fully-connected) layer. CNN architectures make the explicit assumption that the inputs are images, which allows us to encode certain properties into the architecture. These then make the forward function more efficient to implement and vastly reduce the number of parameters in the network.[7]

CNNs take advantage of the fact that the input consists of images and they constrain the architecture more sensibly. In particular, unlike a regular Neural Network, the layers of a CNN have neurons arranged in 3 dimensions i.e. **width, height, depth**.[7]

A simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activations to another through a differentiable function. We use three main types of layers to build CNN architectures i.e. **Convolutional Layer, Pooling Layer, Flattening Layer and Fully-Connected Layer**.[7]

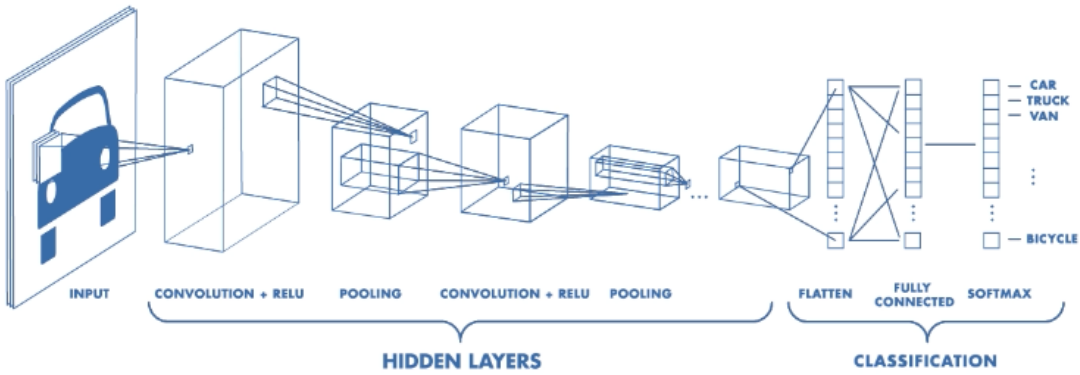


Figure 1: Convolutional Neural Network Architecture[8]

4.1 Convolutional Layer

The Convolutional layer's parameters consist of a set of learnable filters. Every filter is small spatially (along width and height), but extends through the full depth of the input volume. During the forward pass, we slide each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position. Intuitively, the network will learn filters that activate when they see some type of visual features such as an edge of some orientation or a blotch of some color on the first layer, or eventually entire honeycomb or wheel-like patterns on higher layers of the network.[7]

When dealing with high-dimensional inputs such as images, it is impractical to connect neurons to all neurons in the previous volume. Instead, we will connect each neuron to only a local region of the input volume. The spatial extent of this connectivity is a **hyperparameter** called the **receptive field** of the neuron (equivalently this is the **filter size**). The extent of the connectivity along the depth axis is always equal to the depth of the input volume.[7]

Three **hyperparameters** control the size of the output volume: **the depth, stride and zero-padding**. The **depth** of the output volume corresponds to the number of filters we would use for each learning to look for something different in the input.**Stride** is the number of steps we slide the filter by. Sometimes, it will be convenient to pad the input volume with zeros around the border. It will allow us to control the spatial size of the output volumes.[7]

In Convolutional Neural Networks, the output of the convolutional layer will be passed through an activation function. In our case, it is the ReLU activation function.

4.2 Pooling Layer

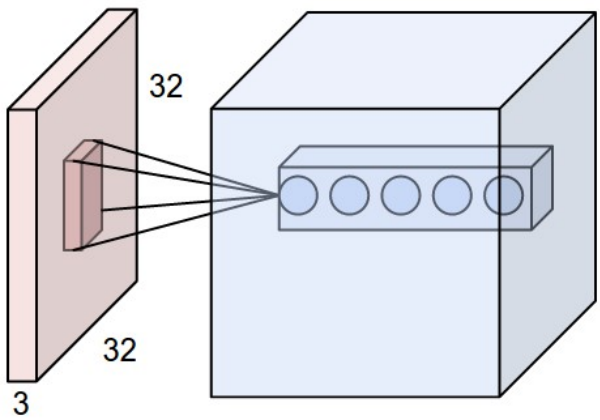


Figure 2: Example volume of neurons in the first Convolutional layer[7]

It is common to periodically insert a Pooling layer in-between successive Convolutional layers in a CNN architecture. Its function is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network, hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. Pooling combines the outputs of clusters of neurons into a single neuron in the next layer. [7]

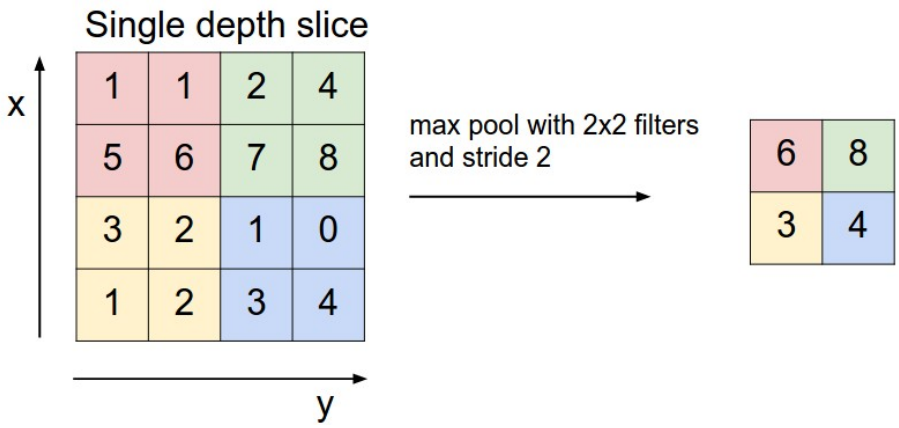


Figure 3: An example of maxpooling of size 2[7]

4.3 Flattening Layer

The output of the pooling layer is passed through the flattening layer before it gets passed to the fully connected layer. It converts the output from the pooling layer to a long one-dimensional feature vector, which is then connected to the final classification model, the fully connected layer.[7]

4.4 Fully Connected Layer

The objective of a fully connected layer is to take the results of the convolution/pooling process and use them to classify the image into a label.[7] The output of convolution/pooling is flattened into a single vector of values, each representing a probability that a certain feature belongs to a label. The fully connected layer is a traditional Multi-Layer Perceptron. It uses a classifier in the output layer. The classifier is usually a softmax activation function (for multi-classification) or sigmoid activation function (for binary-classification). The purpose of this layer is to use the features from the output of the previous layer to classify the input image based on the training data.[4]

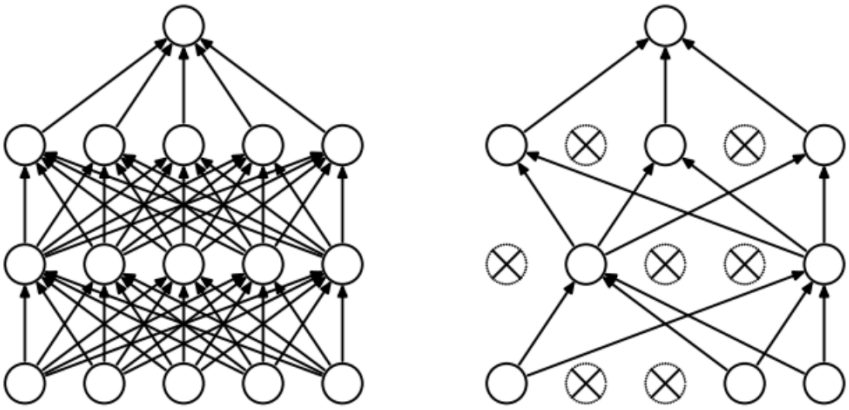
4.5 Dropout Regularization

Dropout is a regularization technique for neural network models where randomly selected neurons are ignored during training. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass.[5]

The effect is that the network becomes less sensitive to the specific weights of neurons. This, in turn, results in a network that is capable of better generalization and is less likely to overfit the training data. As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which, if taken too far, can result in a fragile model too specialized to the training data. This reliant on the context for a neuron during training is referred to as complex co-adaptations.[5]

If neurons are randomly dropped out of the network during training, other neurons will have to step in and handle the representation required to make predictions for the missing neurons. This is believed to result in multiple independent internal representations being learned by the network.[5]

The effect is that the network becomes less sensitive to the specific weights of neurons. This, in turn, results in a network that is capable of better generalization and is less likely to overfit the training data[5].



(a) Standard Neural Net (b) After applying dropout.

Figure 4: Neural Network with and without Dropout[7]

4.6 Batch Normalization

We are using batch normalization in some of our models so it'll be good to go through the basic concept of it.

We normalize the input layer by adjusting and scaling the activations. For example, when we have features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers, that are changing all the time, and get 10 times or more improvement in the training speed. Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift). We can think about batch normalization as doing pre-processing at every layer of the network.[9]

Advantages of batch normalization[9]:

- We can use higher learning rates because batch normalization makes sure that there's no activation that's gone really high or really low. And by that, things that previously couldn't get to train, it will start to train.
- It reduces overfitting because it has a slight regularization effect. Similar to dropout, it adds some noise to each hidden layer's activations. Therefore, if we use batch normalization, we will use less dropout, which is a good thing because we are not going to lose a lot of information. However, we should not depend only on batch normalization for regularization; we should better use it together with dropout.

4.7 Different Kinds of Classification

There are three kinds of classification models based on the given labels for a dataset[13]:

- **Binary classification:** Two exclusive classes. For our case there are two classes:
  1. Illegal logging
  2. No illegal logging

  For this type of classification, we use the sigmoid activation function in the fully connected layer and for loss function, we are using Binary Cross-entropy.
- **Multi-class classification:** More than two exclusive classes. For our case there are three classes:
  1. Illegal logging using chainsaw
  2. Illegal logging using axe
  3. No illegal logging

  For this type of classification, we use the softmax activation function in the fully connected layer and for loss function, we are using Categorical Cross-entropy.
- **Multi-label classification:** Non-exclusive classes. A multi-label classifier can be considered as a multi separate binary classifier. If you have 3 classes here, we will have 3 binary classifiers separately. Each binary classifier is trained independently. Thus, we can produce multi-label for each sample. To make sure at least one label must be acquired, we can select the one with the lowest classification loss function, or by using some other metrics. Multi-class classification is not similar to multi-label classification. Rather, a multi-label classifier borrows the idea from a binary classifier.

4.8 Activation Functions (Sigmoid vs Softmax)

There are three type of activation functions[1]:

1. **Binary Step Function:** A binary step function is a threshold-based activation function. If the input value is above or below a certain threshold, the neuron is activated and sends the same signal to the next layer.

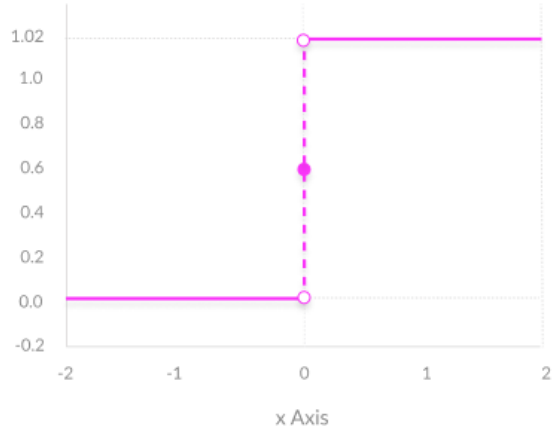


Figure 5: Binary Step Function[1]

The problem with a step function is that it does not allow multi-value outputs. For example, it cannot support classifying the inputs into one of several categories.

2. **Linear Activation Function:** A linear activation function takes the form $\mathbf{A = cx}$:

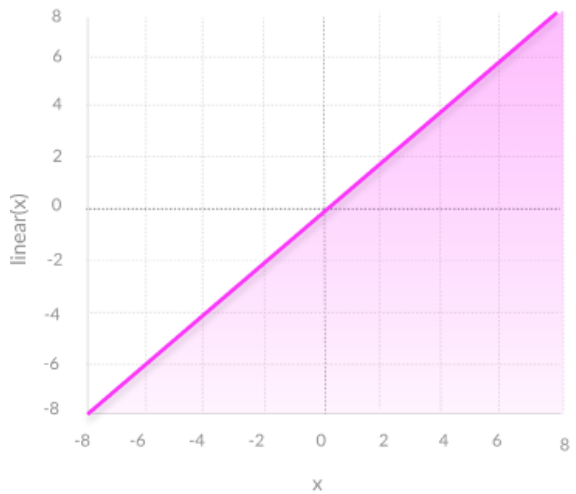


Figure 6: Binary Step Function[1]

It takes the inputs, multiplied by the weights for each neuron, and creates an output signal proportional to the input. In one sense, a linear function is better than a step function because it allows multiple outputs, not just yes and no.

A linear activation function has two major problems:

- Not possible to use back-propagation (gradient descent) to train the model. The derivative of the function is a constant and has no relation to the input, X.
- All layers of the neural network collapse into one. No matter how many layers in the neural network, the last layer will be a linear function of the first layer because a linear combination of linear functions is still a linear function.

3. **Non-Linear Activation Functions:** They allow the models to create complex mappings between the network's inputs and outputs, which are essential for learning and modeling complex data, such as images, video, audio, and data sets which are non-linear or have high dimensionality. They also address all the problems of Linear Activation Functions as well.

   There are seven common Non-linear Activation Functions and each of them have their own advantages and disadvantages. More details can be found in this article.

   For this report we will only go through the differences between Sigmoid and Softmax Activation Functions. In the previous subsection, we mentioned that we use Sigmoid as the activation function for Binary Classifier (or Multi-label Classifier) and Softmax for Multi-class Classifier.

   The table below lists the differences between Sigmoid and Softmax:

Softmax	**Sigmoid**
Probabilities produced by a softmax are interrelated and will always sum to one	Probabilities produced by a sigmoid are independent and the sum is not necessarily one
Multi-Class Classification Problem	Multi-Label or Binary Classification Problem
Only one right answer	More than one right answer
Mutually exclusive outputs	Non-exclusive outputs
Example: If we are classifying images of handwritten digits (the MNIST data set), we want to force the classifier to choose only one identity for the digit	**Example:** A single chest x-ray could show many different medical conditions at the same time
Figure 3 for Softmax equation	Figure 4 for Sigmoid graph and Figure 5 for Sigmoid equation

Table 1: Comparison between Softmax and Sigmoid[10]

$$\text{softmax}(z_j) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \text{ for j = 1,...,K}$$

Figure 7: Equation for Softmax[10]

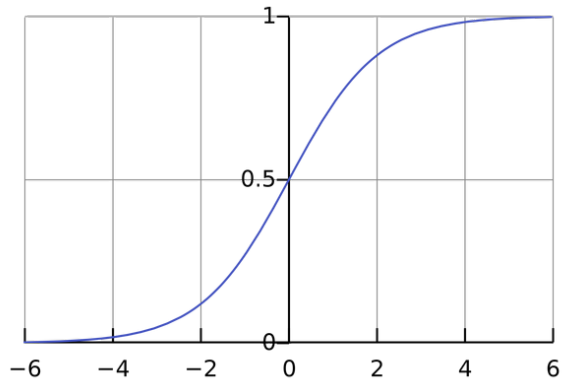


Figure 8: Sigmoid Graph[10]

$$\sigma(z_j) = \frac{e^{z_j}}{1+e^{z_j}}$$

Figure 9: Equation for Sigmoid[10]

4.9 Loss Functions (Binary Cross-entropy vs Categorical Cross-entropy)

The function that is used to compute error is known as Loss Function. Loss functions are helpful to train a neural network. Given an input and a target, they calculate the loss, i.e difference between output and the target variable. Different loss functions will give different errors for the same prediction, and thus have a considerable effect on the performance of the model. Different loss functions are used to deal with different type of tasks, i.e. regression and classification.[2]

Loss functions fall under three major categories[2]:

1. **Embedding loss functions:** It deals with problems where we have to measure, whether two inputs are similar or dissimilar.
2. **Regressive loss functions:** They are used in case of regressive problems, that is when the target variable is continuous.
3. **Classification loss functions:** The output variable in classification problem is usually a probability value f(x), called the score for the input x. Generally, the magnitude of the score represents the confidence of our prediction.

Since we are dealing with a classification problem for this project, we'll look deeper into Cross-entropy classification loss function. The equation below describes Cross-entropy loss, where t_i and s_i are the ground truth and the CNN score for each class i in C[12].

$$CE = -\sum_{i}^{C} t_i log(s_i)$$

Figure 10: Equation for Cross-entropy loss function[12]

Now let's take a deeper look into the two types of Cross-entropy losses that we are using for this project[12]:

- **Binary Cross-Entropy Loss:** Also known as Sigmoid Cross-Entropy loss. It is a Sigmoid activation plus a Cross-Entropy loss. It is independent for each vector component (class), meaning that the loss computed for every CNN output vector component is not affected by other component values. That's why it is also used for multi-label classification. It's called Binary Cross-Entropy Loss because it sets up a binary classification problem between $C^{'} = 2$ classes for every class in C.

$$CE = -\sum_{i=1}^{C'=2} t_i log(f(s_i)) = -t_1 log(f(s_1)) - (1 - t_1) log(1 - f(s_1))$$

Figure 11: The formulation of Cross Entropy Loss for binary problems[12]

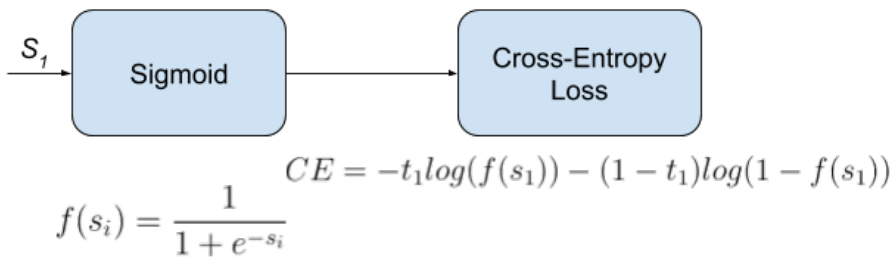


Figure 12: Sigmoid Cross-entropy pipeline[12]

- **Categorical Cross-Entropy Loss:** Also know as Softmax Loss. It is a Softmax activation plus a Cross-Entropy loss. If we use this loss, we will train a CNN to output a probability over the C classes for each image. It is used for multi-class classification.

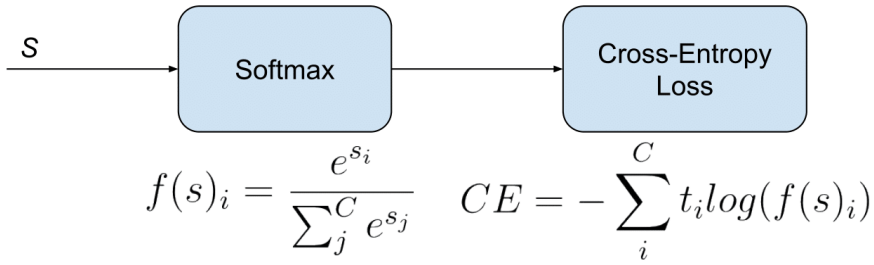


Figure 13: Softmax Cross-entropy pipeline[12]

More details on Cross-entropy Loss function can be found in this article.

4.10 Optimization Algorithm

Optimization algorithms help us to minimize (or maximize) an Objective function (or error) which is simply a mathematical function dependent on the Model's internal learnable parameters which are used in computing the target values(Y) from the set of predictors(X) used in the model[16].

Since we only tested with **Adam** (which stands for Adaptive Moment Estimation) as the optimizer for all our models, we won't be discussing in-depth on this topic. More details can be found in this article

5 Models

We used seven different CNN models to train on our dataset. In this section we will briefly describe each of our models:

5.1 Basic CNN Model

It has an input layer of width, height and depth of 200, 200, and 3. We added a Convolutional layer with 16 filters, a kernel size of 3X3 and stride of 1 and padding of 0. Then, we added a layer of ReLU activation function. For the next layer, we added a pooling layer with a MaxPool function and pool size of 2 and then flattened the output with the flattening layer. Finally, a fully connected layer with the sigmoid/softmax (depending on the type of classification) activation function is added which gives out the result.

5.2 VGGnet

VGGnet is a plain and straight forward CNN architecture. The idea of VGGnet architectures is quite simple. We have to stack the convolutional layers with increasing filter sizes. In every VGGnet architecture, all filters are of size 3*3.

We created a custom VGGnet with 6 convolutional layers in Keras. After every Convolutional layer, we added a Dropout layer to reduce the overfitting and MaxPooling layer after every pair of Convolutional layers for Dimensionality Reduction.

The problem with VGGnet is that this naive architecture is not good for deeper networks, as the deeper the network goes, it is more prone to Vanishing Gradients Problem. More training and also more parameters have to be tuned in deeper VGGnet architectures.[3]

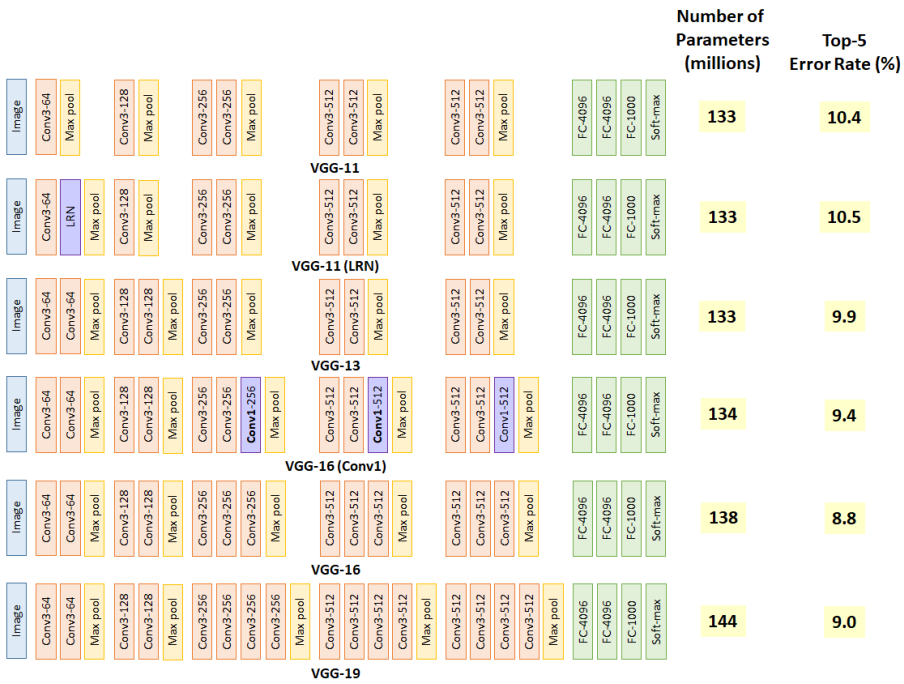


Figure 14: VGGnet Architecture Example

5.3 ResNet

ResNet connects not just the previous layer to the current layer but also a layer behind the previous layer. By incorporating this, now each layer can see more than just its previous layer's observations.

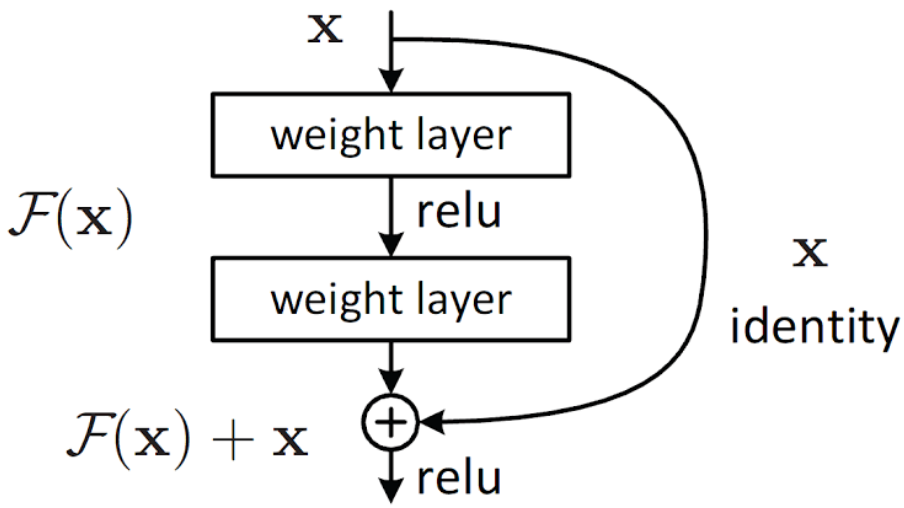


Figure 15: Identity Shortcut Connection.

Training of such a deep Residual Networks is possible by using BatchNormalisation layers after every Convolutional Layer. Batch Normalisation layers will boost the values of weights and hence higher learning rates can be used while training, which will help train faster and also can minimize Vanishing Gradient problem.

We created our custom resnet by first creating a res_layer function. The function takes arguments as follows:

- **x** : input to the res_layer.
- **filters** : number of filters in each convolutional layer.
- **pooling** : whether to add a pooling layer (default is False).
- **dropout** : whether to add a dropout layer (default is No).

The process is as follows: it connects the input layer to a Conv2D layer having filters specified in our function calling and then attaches a BatchNormalization layer, upon which a ReLU Activation layer is added, and then another

Conv2D layer is stacked. Now, this stacked output is added with the initial input, but the initial input is transformed by passing it through a Conv2D layer of given filters. This step is done to match the output sizes of both the layers that are going to be added. Then, if we prefer having a MaxPooling2D layer, we add that and also if any Dropout value is given, then a Dropout layer is also added and then finally one more BatchNormalization layer and Activation layer are added and then this final layer is returned by our res_layer function.

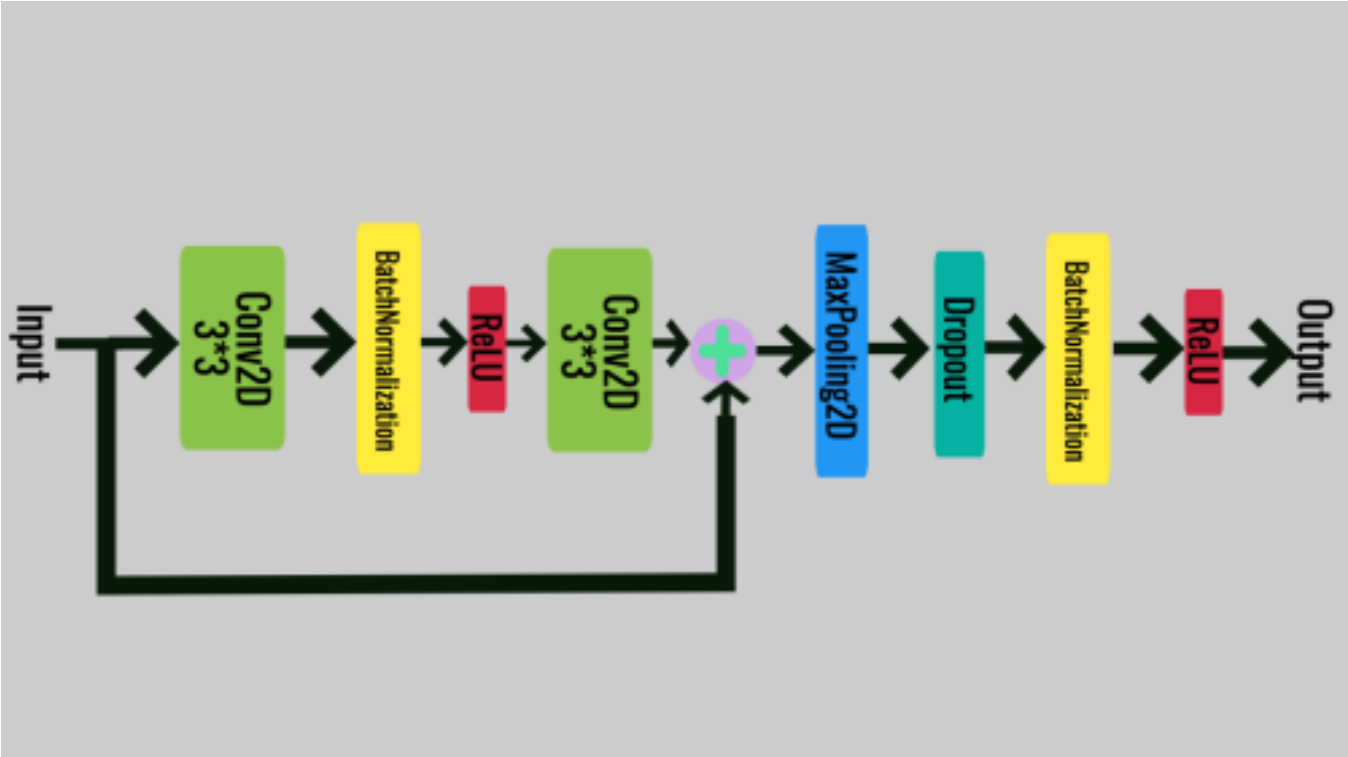


Figure 16: Architecture of res_layer function.

Then we created a simple ResNet model with 6 res_layers.[3]

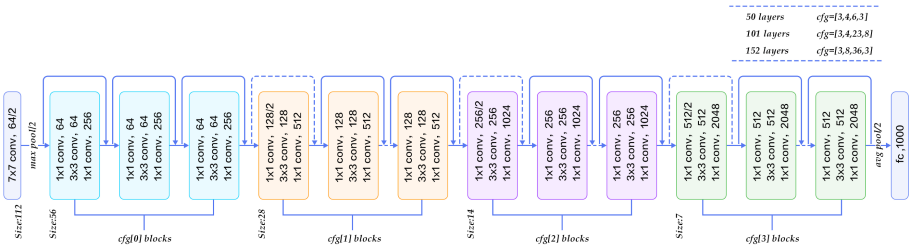


Figure 17: ResNet Architechture Example

5.4 DenseNet

In DenseNet, for a given layer, all other layers preceding it are concatenated and given as input to the current layer. With such an arrangement, we can use smaller filter counts and also, this will minimize the vanishing gradient problem as all layers are directly connected to the out put. Gradients can be calculated directly from the output for each layer.

For our custom DenseNet, we developed a function for the dense_layer. The function takes the following parameters where:

- **x** : input layer
- **layer_configs** : a list of dictionaries, where each dictionary is of the below format. Inside the layer_configs, the dictionaries have the following keys:
  - **"layer_type"** : which type of layer we are going to create. Currently, only the Conv2D layer is supported by the above dense_layer function.
  - **"filters"** : determining number of filters in that layer. An integer is given as value.
  - **"kernel_size"** : size of the kernel. A tuple of kernel size is given as value. like (3, 3).

- **"strides"** : step size of stride. An integer is given as value.
- **"padding"** : type of padding to be applied to the layer.
- **"activation"** : type of activation to be applied to the layer.

We created some custom layer_config using the format below.

```
demo_layer = [
    {
        "layer_type" : "Conv2D", "filters" : 8, "kernel_size" : (3, 3), "strides" : 1, "padding" : "same", "activation" : "relu"
    },{
        "layer_type" : "Conv2D", "filters" : 8, "kernel_size" : (3, 3), "strides" : 1, "padding" : "same", "activation" : "relu"
    },{
        "layer_type" : "Conv2D", "filters" : 8, "kernel_size" : (3, 3), "strides" : 1, "padding" : "same", "activation" : "relu"
    },{
        "layer_type" : "Conv2D", "filters" : 8, "kernel_size" : (3, 3), "strides" : 1, "padding" : "same", "activation" : "relu"
    },{
        "layer_type" : "Conv2D", "filters" : 8, "kernel_size" : (3, 3), "strides" : 1, "padding" : "same", "activation" : "relu"
    }
]
```

Figure 18: Demo Layer_config

Then we created a simple custom DenseNet model with 5 dense_layers.[3]

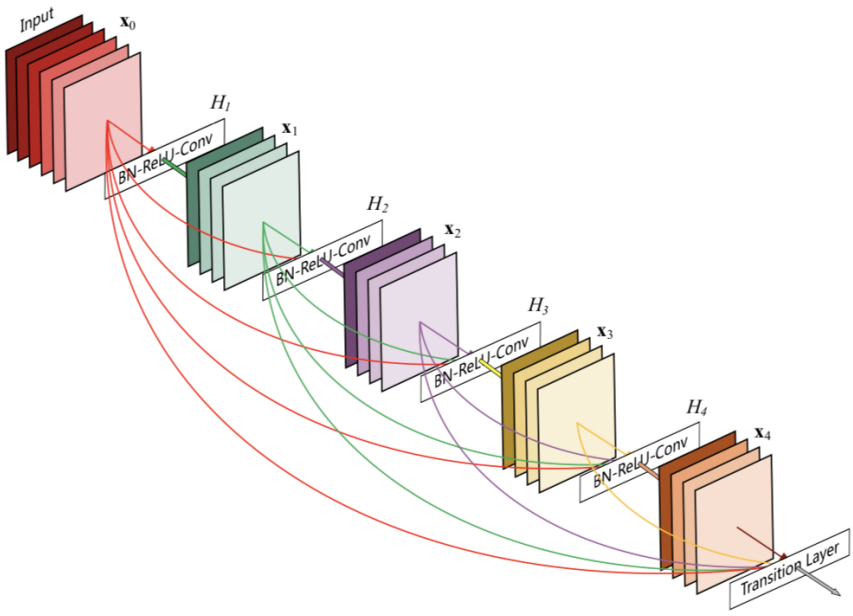
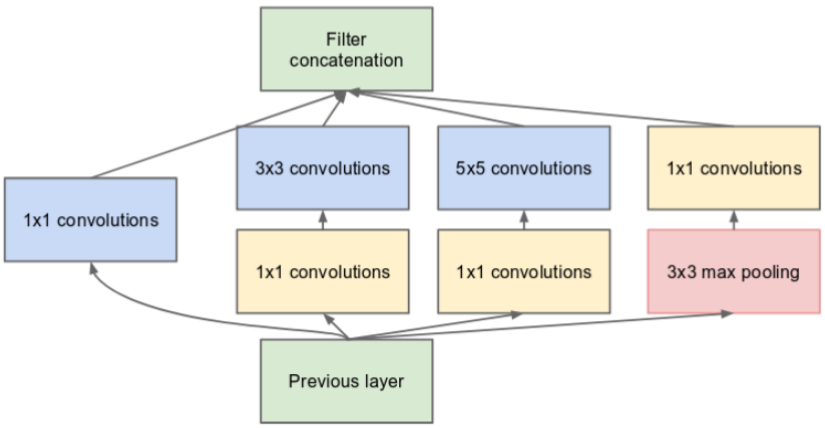


Figure 19: Dense Layer Architecture Example

5.5 Inception Net

The idea of Inception Net is to make the network wider. This can be done by parallel connection of multiple layers having different filters, and then finally concatenating all of those parallel paths to pass to the next layers.



(b) Inception module with dimension reductions

Figure 20: Inception Layer Architecture Example

We wrote a inception_layers function where the parameters are:

- **x** : input layer.
- **layer_configs** : a list of lists, where each list have dictionaries. The keys in the dictionaries are:
  - **"layer_type"** : which type of layer we are going to create. Currently only Conv2D layer is supported by the above dense_layer function.
  - **"filters"** : determining number of filters in that layer. An integer is given as value.
  - **"kernel_size"** : size of the kernel. A tuple of kernel size is given as value. like (3, 3).
  - **"strides"** : step size of stride. An integer is given as value.
  - **"padding"** : type of padding to be applied to the layer.
  - **"activation"** : type of activation to be applied to the layer.

We created some custom layer_config using the format below.

```
custom_layer = [[{
    "layer_type" : "Conv2D", "filters" : 64, "kernel_size" : (1, 1), "strides" : 1, "padding" : "same", "activation" : "relu"
    }
],[
    {
    "layer_type" : "Conv2D", "filters" : 96, "kernel_size" : (1, 1), "strides" : 1, "padding" : "same", "activation" : "relu"
    },{
    "layer_type" : "Conv2D", "filters" : 128, "kernel_size" : (3, 3), "strides" : 1, "padding" : "same", "activation" : "relu"
    }
],[
    {
    "layer_type" : "Conv2D", "filters" : 16, "kernel_size" : (1, 1), "strides" : 1, "padding" : "same", "activation" : "relu"
    },{
    "layer_type" : "Conv2D", "filters" : 32, "kernel_size" : (5, 5), "strides" : 1, "padding" : "same", "activation" : "relu"
    }
]]
```

Figure 21: Demo layer_config

Then we created a simple custom Inception Net model with 3 inception_layers.[3]

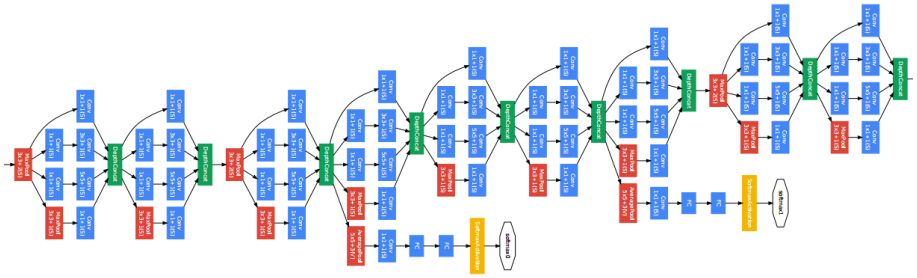


Figure 22: Inception Net Architecture Example

5.6 Xception Net

In Xception Net, Depthwise Separable Convolutional operations are performed. Depthwise Separable Convolutions are different from normal convolutions in a way that, in a normal Conv2D layer for an image, we can use any number of filters in the Conv layer. Each of those filters will be operated over all three channels and the output is the sum of all corresponding values. But in Depthwise separable convolutions, each channel has only one kernel to do convolution. Hence, by performing Depthwise Separable Convolutions, we can reduce the computational complexity as every kernel is of two dimensional only and is convoluting only over one channel. In Keras, we can implement this by using the DepthwiseConv2D layer.

In our custom Xcetion, we made a simple XceptionNet by using a mixture of convolutional layers, BatchNormalization layers, dropout layer, MaxPooling layers, and depthwise convolutional layer. We used activation function ReLU during the convolution process and passed the output layer after flattening it to a fully connected layer with a sigmoid/softmax function.[3]

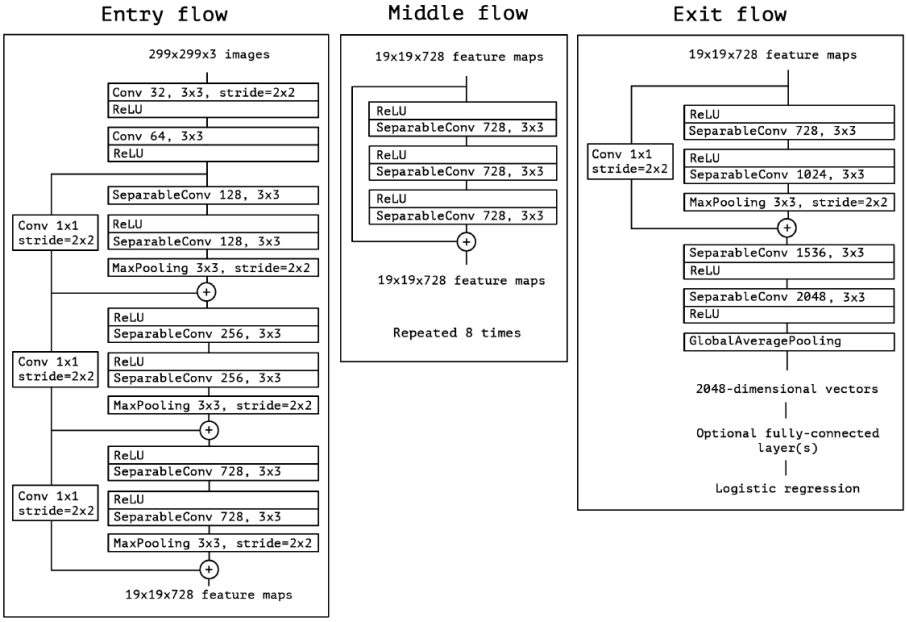


Figure 23: Xception Architecture Example

5.7 Custom Network

It has an input layer of width, height, and depth of 200, 200 and 3. We added a Convolutional layer with 16 filters, a kernel size of 3X3 and stride of 1, padding of 0. Then we added a layer of the ReLU activation function. Next, we added a pooling layer of Maxpool function with dimension 2 and then added a dropout layer with a probability of 0.2.

Then we added another Convolutional layer with 32 filters, a kernel size of 3X3 and stride of 1, padding of 0. Then we added a layer of the ReLU activation function. Next, we added a pooling layer of Maxpool function with dimension 2 and then added a dropout layer with a probability of 0.4.

Finally, we added a flattening layer and a fully connected layer with a sigmoid/softmax activation function, which gives the output.

6 Results

6.1 Background of the tests

The amount of data is 3000 images of 200x200 pixels dimension – 1000 normal spectrograms (rainforest, birds chirping, waterfalls, etc.), while 2000 spectrograms have chainsaw or wood-chopping sounds added into them (1000 for each sound). These images are then converted into their RGB channels and their arrays, with labels indicating the nature of the sounds.

For Binary Classification, if the sounds are "normal", that is, no deforestation happening, then it is labeled as 0. Otherwise, if there are chainsaw or chopping sounds, it will be labeled as 1.

As for Multi-Class Classification, if the sounds are normal forest sounds, the label will be 0. If there is a chainsaw sound, the label will be 1. Otherwise, if there is a chopping sound detected in the sound file / spectrogram, the label will be 2.

The image directories – a mix of normal and deforestation spectrogram images, are shuffled randomly using random.seed(n) (where n is an integer) such that the order of images that were going through are the same across all the different models. 100 random spectrograms are chosen for evaluation, while the rest of the spectrograms are chosen for training.

The machine used for the testing and evaluation is Dell's Inspiron 7559, with Intel(R) Core^{TM} i7-6700HQ CPU @ 2.60 GHz (Intel TurboBoosts to 3.50 GHz). The testing machine has a Physical RAM of 16.0 GB, with the data stored on 256 GB Solid State Drive (SSD) and ran on Jupyter Notebook. The Keras and Python codes make use of Nvidia's CUDA functionality, with the testing machine having Nvidia GeForce GTX 960M, scoring a 5.0 / 10 compatibility with CUDA – a parallel computing platform and application programming interface model created by Nvidia.

All the models run with epochs = 50, the models are restarted and run every time, to prevent OOM (Out of Memory) errors, or the situations when the GPU has too much memory dump, or to prevent contamination of the models from different runs.

6.2 Results for Binary Classification

The results of the Binary Classification are given below.

6.2.1 Table of results for Binary CNNs

Evaluator - Binary	**Custom Model**	**Basic CNN**	**VGG Network**	**Resnet**	**Densenet**	**Inception**	**Xception**
Final Training Loss	0.0660	1.0796e-5	0.0363	0.0445	0.0550	0.6362	0.0304
Final Training Accuracy	0.9772	1.0000	0.9879	0.9845	0.9789	0.6677	0.9927
Final Validation Loss	0.5178	0.6964	0.2262	0.2471	0.6957	1.9194	0.5072
Final Validation Accuracy	0.8586	0.8966	0.9172	0.9190	0.6621	0.6621	0.9121
Final Evaluation Loss	0.3123	0.5305	0.1598	0.1247	0.68447	1.9025	0.3882
Final Evaluation Accuracy	0.8900	0.9000	0.9400	0.9552	0.67000	0.6217	0.9100
Time Taken / Epoch	7 - 9 sec	5 - 8 sec	24 - 27 sec	141 sec	207 - 209 sec	38 - 41 sec	105 - 110 sec
Time Taken / Overall	6 min 50 sec	4 min 42 sec	21 min 12 sec	1 hr 58 min 11 sec	2 hr 53 min 44 sec	34 min 6 sec	1 hr 30 min 30 sec

6.2.2 Basic CNN Binary Classification

For the Basic model, the training accuracy increased rapidly from the beginning, reaching a complete 100% (1.0000) at the end. As for the validation accuracy, it has seen an improvement as well, though it does have some difficulties on some of the epochs as seen by the dips in the graphs. The training loss decreased by several factors. The Basic CNN took an average of 5 to 8 seconds per epoch, and in the end, it took 4 minutes and 42 seconds to be fully finished.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 198, 198, 16)      448
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 99, 99, 16)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 156816)            0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 156817
=================================================================
Total params: 157,265
Trainable params: 157,265
Non-trainable params: 0
_________________________________________________________________
```

Figure 24: Basic CNN's Model

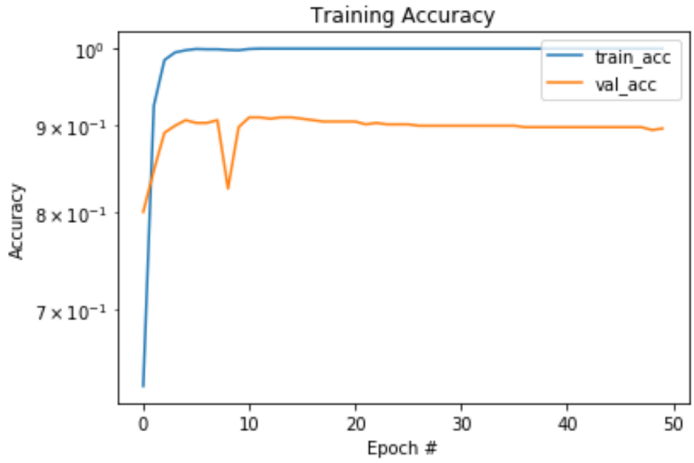


Figure 25: Basic CNN's Training Accuracy

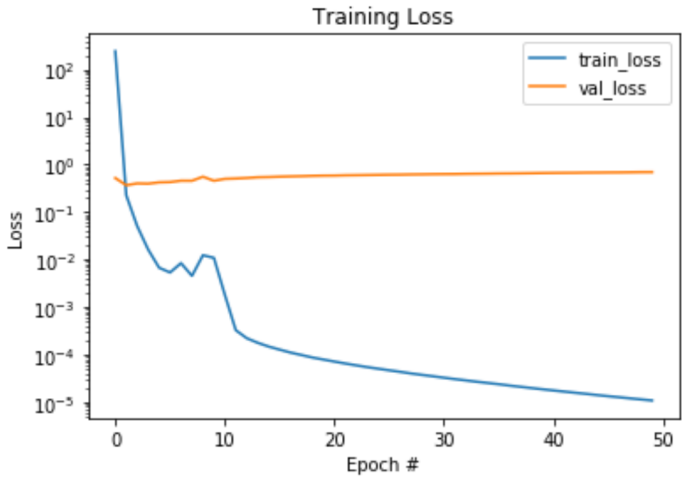


Figure 26: Basic CNN's Training Loss

```
Loss: 0.5305477502942085
Accuracy: 0.8999999761581421
AUC  is:   0.8869289914066034
              precision    recall  f1-score   support

           0       0.85      0.85      0.85        33
           1       0.93      0.93      0.93        67

    accuracy                           0.90       100
   macro avg       0.89      0.89      0.89       100
weighted avg       0.90      0.90      0.90       100

Confusion Matrix:
 [[28  5]
 [ 5 62]]
```

Figure 27: Basic CNN's Report

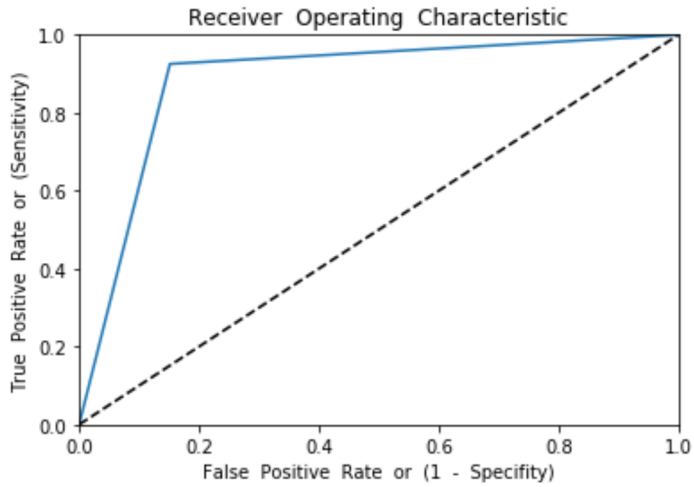


Figure 28: Basic CNN's ROC

6.2.3 VGGnet Binary Classification

For the VGGnet model, the training accuracy increased gradually from the beginning, reaching a 0.9879 at the end. As for validation accuracy, there were dips in it just like the Basic CNN for Binary classification, though there were slightly more frequent drops. The training loss had a sharp decrease at the start, followed by a gradual exponential decrease. The VGGnet CNN took an average of 24 to 27 seconds per epoch, and in the end, it took 21 minutes and 12 seconds to be fully finished.

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 200, 200, 16)      448
_________________________________________________________________
dropout_5 (Dropout)          (None, 200, 200, 16)      0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 200, 200, 16)      2320
_________________________________________________________________
dropout_6 (Dropout)          (None, 200, 200, 16)      0
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 100, 100, 16)      0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 100, 100, 32)      4640
_________________________________________________________________
dropout_7 (Dropout)          (None, 100, 100, 32)      0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 100, 100, 32)      9248
_________________________________________________________________
dropout_8 (Dropout)          (None, 100, 100, 32)      0
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 50, 50, 32)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 50, 50, 64)        18496
_________________________________________________________________
flatten_2 (Flatten)          (None, 160000)            0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 160001
=================================================================
Total params: 195,153
Trainable params: 195,153
Non-trainable params: 0
```

Figure 29: VGGnet CNN's Model

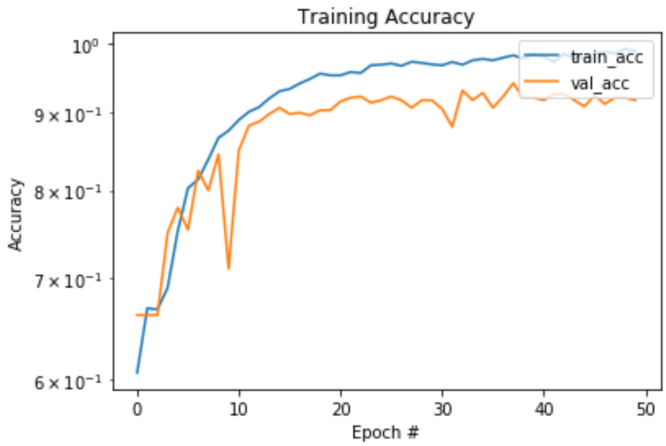


Figure 30: VGGnet CNN's Training Accuracy

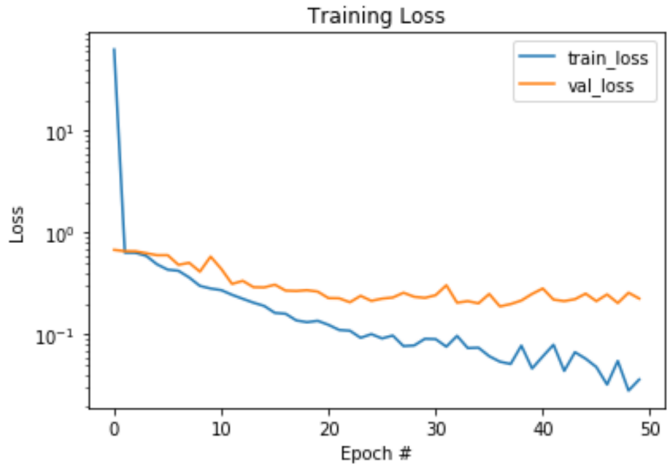


Figure 31: VGGnet CNN's Training Loss

```
Loss: 0.1598198628425598
Accuracy: 0.9399999976158142
AUC  is:   0.9244685662596109
              precision    recall  f1-score   support

           0       0.94      0.88      0.91        33
           1       0.94      0.97      0.96        67

    accuracy                           0.94       100
   macro avg       0.94      0.92      0.93       100
weighted avg       0.94      0.94      0.94       100

Confusion Matrix:
 [[29  4]
 [ 2 65]]
```

Figure 32: VGGnet CNN's Report

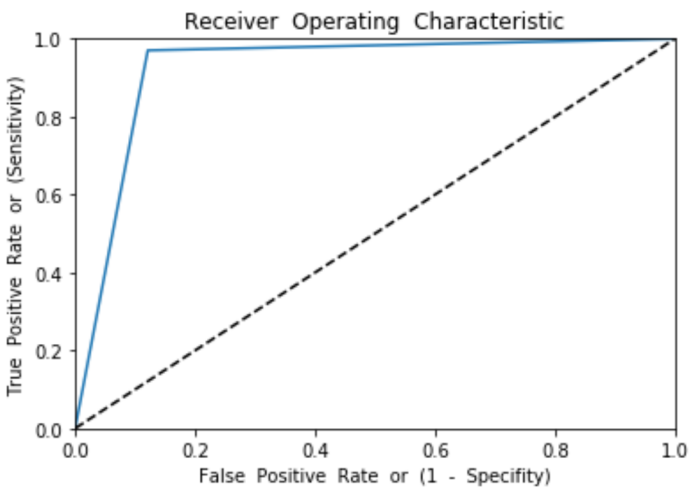


Figure 33: VGGnet CNN's ROC

6.2.4 ResNet Binary Classification

For the Resnet model, the training accuracy increased gradually, though it was much faster than the VGGnet in terms of accuracy improvements. The validation accuracy, however, took some hits, having a frequent amount of difficulty, though it does tend to converge to a reasonable 0.9190. The training loss once again decreased exponentially, but the validation loss had some difficulty on the 12th epoch or so. Resnet took an average of 141 seconds per epoch, while the overall process took 1 hour, 58 minutes, and 11 seconds.

Model: "Resnet"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 200, 200, 3)	0	
conv2d_1 (Conv2D)	(None, 200, 200, 16)	448	input_1[0][0]
batch_normalization_1 (BatchNor	(None, 200, 200, 16)	64	conv2d_1[0][0]
activation_1 (Activation)	(None, 200, 200, 16)	0	batch_normalization_1[0][0]
conv2d_2 (Conv2D)	(None, 200, 200, 16)	2320	activation_1[0][0]
batch_normalization_2 (BatchNor	(None, 200, 200, 16)	64	conv2d_2[0][0]
activation_2 (Activation)	(None, 200, 200, 16)	0	batch_normalization_2[0][0]
conv2d_3 (Conv2D)	(None, 200, 200, 16)	2320	activation_2[0][0]
conv2d_4 (Conv2D)	(None, 200, 200, 16)	2320	activation_1[0][0]
add_1 (Add)	(None, 200, 200, 16)	0	conv2d_3[0][0] conv2d_4[0][0]
dropout_1 (Dropout)	(None, 200, 200, 16)	0	add_1[0][0]
batch_normalization_3 (BatchNor	(None, 200, 200, 16)	64	dropout_1[0][0]
activation_3 (Activation)	(None, 200, 200, 16)	0	batch_normalization_3[0][0]
conv2d_5 (Conv2D)	(None, 200, 200, 16)	2320	activation_3[0][0]
batch_normalization_4 (BatchNor	(None, 200, 200, 16)	64	conv2d_5[0][0]
activation_4 (Activation)	(None, 200, 200, 16)	0	batch_normalization_4[0][0]
conv2d_6 (Conv2D)	(None, 200, 200, 16)	2320	activation_4[0][0]
conv2d_7 (Conv2D)	(None, 200, 200, 16)	2320	activation_3[0][0]
add_2 (Add)	(None, 200, 200, 16)	0	conv2d_6[0][0] conv2d_7[0][0]
dropout_2 (Dropout)	(None, 200, 200, 16)	0	add_2[0][0]
batch_normalization_5 (BatchNor	(None, 200, 200, 16)	64	dropout_2[0][0]
activation_5 (Activation)	(None, 200, 200, 16)	0	batch_normalization_5[0][0]
conv2d_8 (Conv2D)	(None, 200, 200, 16)	2320	activation_5[0][0]
batch_normalization_6 (BatchNor	(None, 200, 200, 16)	64	conv2d_8[0][0]
activation_6 (Activation)	(None, 200, 200, 16)	0	batch_normalization_6[0][0]
conv2d_9 (Conv2D)	(None, 200, 200, 16)	2320	activation_6[0][0]
conv2d_10 (Conv2D)	(None, 200, 200, 16)	2320	activation_5[0][0]
add_3 (Add)	(None, 200, 200, 16)	0	conv2d_9[0][0] conv2d_10[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 100, 100, 16)	0	add_3[0][0]
dropout_3 (Dropout)	(None, 100, 100, 16)	0	max_pooling2d_1[0][0]
batch_normalization_7 (BatchNor	(None, 100, 100, 16)	64	dropout_3[0][0]
activation_7 (Activation)	(None, 100, 100, 16)	0	batch_normalization_7[0][0]
conv2d_11 (Conv2D)	(None, 100, 100, 32)	4640	activation_7[0][0]
batch_normalization_8 (BatchNor	(None, 100, 100, 32)	128	conv2d_11[0][0]
activation_8 (Activation)	(None, 100, 100, 32)	0	batch_normalization_8[0][0]
conv2d_12 (Conv2D)	(None, 100, 100, 32)	9248	activation_8[0][0]
conv2d_13 (Conv2D)	(None, 100, 100, 32)	4640	activation_7[0][0]
add_4 (Add)	(None, 100, 100, 32)	0	conv2d_12[0][0] conv2d_13[0][0]
dropout_4 (Dropout)	(None, 100, 100, 32)	0	add_4[0][0]
batch_normalization_9 (BatchNor	(None, 100, 100, 32)	128	dropout_4[0][0]
activation_9 (Activation)	(None, 100, 100, 32)	0	batch_normalization_9[0][0]
conv2d_14 (Conv2D)	(None, 100, 100, 32)	9248	activation_9[0][0]
batch_normalization_10 (BatchNo	(None, 100, 100, 32)	128	conv2d_14[0][0]
activation_10 (Activation)	(None, 100, 100, 32)	0	batch_normalization_10[0][0]
conv2d_15 (Conv2D)	(None, 100, 100, 32)	9248	activation_10[0][0]
conv2d_16 (Conv2D)	(None, 100, 100, 32)	9248	activation_9[0][0]
add_5 (Add)	(None, 100, 100, 32)	0	conv2d_15[0][0] conv2d_16[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 50, 50, 32)	0	add_5[0][0]
dropout_5 (Dropout)	(None, 50, 50, 32)	0	max_pooling2d_2[0][0]
batch_normalization_11 (BatchNo	(None, 50, 50, 32)	128	dropout_5[0][0]
activation_11 (Activation)	(None, 50, 50, 32)	0	batch_normalization_11[0][0]
conv2d_17 (Conv2D)	(None, 50, 50, 64)	18496	activation_11[0][0]
batch_normalization_12 (BatchNo	(None, 50, 50, 64)	256	conv2d_17[0][0]
activation_12 (Activation)	(None, 50, 50, 64)	0	batch_normalization_12[0][0]
conv2d_18 (Conv2D)	(None, 50, 50, 64)	36928	activation_12[0][0]
conv2d_19 (Conv2D)	(None, 50, 50, 64)	18496	activation_11[0][0]
add_6 (Add)	(None, 50, 50, 64)	0	conv2d_18[0][0] conv2d_19[0][0]
dropout_6 (Dropout)	(None, 50, 50, 64)	0	add_6[0][0]
batch_normalization_13 (BatchNo	(None, 50, 50, 64)	256	dropout_6[0][0]
activation_13 (Activation)	(None, 50, 50, 64)	0	batch_normalization_13[0][0]
flatten_1 (Flatten)	(None, 160000)	0	activation_13[0][0]
dropout_7 (Dropout)	(None, 160000)	0	flatten_1[0][0]
dense_1 (Dense)	(None, 64)	10240064	dropout_7[0][0]
dropout_8 (Dropout)	(None, 64)	0	dense_1[0][0]
dense_2 (Dense)	(None, 1)	65	dropout_8[0][0]

Total params: 10,383,121
Trainable params: 10,382,385
Non-trainable params: 736

Figure 34: Resnet CNN's Model

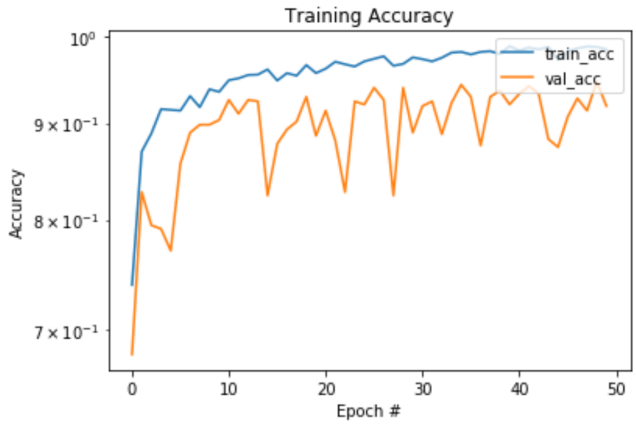


Figure 35: Resnet CNN's Training Accuracy

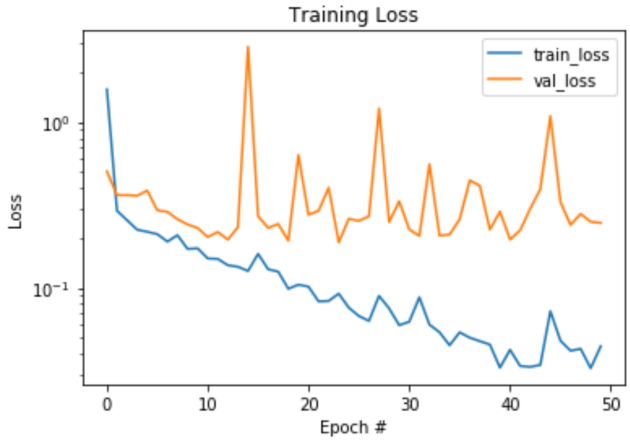


Figure 36: Resnet CNN's Training Loss

```
Loss: 0.12467411011457444
Accuracy: 0.9399999976158142
AUC  is:   0.9552238805970149
              precision    recall  f1-score   support

           0       0.85      1.00      0.92        33
           1       1.00      0.91      0.95        67

    accuracy                           0.94       100
   macro avg       0.92      0.96      0.93       100
weighted avg       0.95      0.94      0.94       100

Confusion Matrix:
 [[33  0]
 [ 6 61]]
```

Figure 37: Resnet CNN's Report

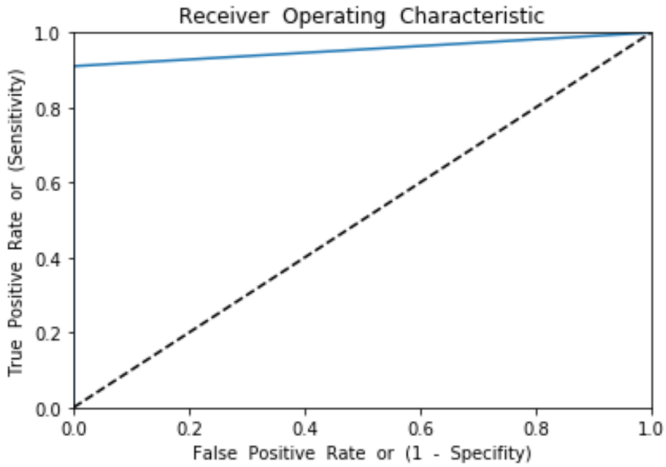


Figure 38: Resnet CNN's ROC

6.2.5 DenseNet Binary Classification

Densenet's training accuracy had a slightly higher start compared to the others, also, in turn, increasing gradually and finishing off at 0.9789. The validation accuracy, on the other hand, lingered at 0.6621, with one sharp decrease considered an outlier. Both the training loss and the validation loss decreases gradually, though with fluctuations along the way. The Densenet model, out of all the models, took the longest so far, with each epoch taking 207 to 209 seconds, with the overall total time being 2 hours and 53 minutes, nearly 3 hours

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 200, 200, 3)	0	
conv2d_1 (Conv2D)	(None, 200, 200, 2)	56	input_1[0][0]
conv2d_2 (Conv2D)	(None, 200, 200, 2)	38	conv2d_1[0][0]
conv2d_3 (Conv2D)	(None, 200, 200, 2)	38	conv2d_1[0][0]
concatenate_1 (Concatenate)	(None, 200, 200, 4)	0	conv2d_2[0][0] conv2d_3[0][0]
conv2d_4 (Conv2D)	(None, 200, 200, 2)	74	concatenate_1[0][0]
concatenate_2 (Concatenate)	(None, 200, 200, 6)	0	conv2d_2[0][0] conv2d_3[0][0] conv2d_4[0][0]
conv2d_5 (Conv2D)	(None, 200, 200, 2)	110	concatenate_2[0][0]
concatenate_3 (Concatenate)	(None, 200, 200, 8)	0	conv2d_2[0][0] conv2d_3[0][0] conv2d_4[0][0] conv2d_5[0][0]
conv2d_6 (Conv2D)	(None, 200, 200, 2)	146	concatenate_3[0][0]
dropout_1 (Dropout)	(None, 200, 200, 2)	0	conv2d_6[0][0]
batch_normalization_1 (BatchNor	(None, 200, 200, 2)	8	dropout_1[0][0]
conv2d_7 (Conv2D)	(None, 200, 200, 4)	76	batch_normalization_1[0][0]
conv2d_8 (Conv2D)	(None, 200, 200, 4)	76	batch_normalization_1[0][0]
concatenate_4 (Concatenate)	(None, 200, 200, 8)	0	conv2d_7[0][0] conv2d_8[0][0]
conv2d_9 (Conv2D)	(None, 200, 200, 4)	292	concatenate_4[0][0]
concatenate_5 (Concatenate)	(None, 200, 200, 12)	0	conv2d_7[0][0] conv2d_8[0][0] conv2d_9[0][0]
conv2d_10 (Conv2D)	(None, 200, 200, 4)	436	concatenate_5[0][0]
concatenate_6 (Concatenate)	(None, 200, 200, 16)	0	conv2d_7[0][0] conv2d_8[0][0] conv2d_9[0][0] conv2d_10[0][0]
conv2d_11 (Conv2D)	(None, 200, 200, 4)	580	concatenate_6[0][0]
dropout_2 (Dropout)	(None, 200, 200, 4)	0	conv2d_11[0][0]
batch_normalization_2 (BatchNor	(None, 200, 200, 4)	16	dropout_2[0][0]
conv2d_12 (Conv2D)	(None, 200, 200, 8)	296	batch_normalization_2[0][0]
conv2d_13 (Conv2D)	(None, 200, 200, 8)	296	batch_normalization_2[0][0]
concatenate_7 (Concatenate)	(None, 200, 200, 16)	0	conv2d_12[0][0] conv2d_13[0][0]
conv2d_14 (Conv2D)	(None, 200, 200, 8)	1160	concatenate_7[0][0]
concatenate_8 (Concatenate)	(None, 200, 200, 24)	0	conv2d_12[0][0] conv2d_13[0][0] conv2d_14[0][0]
conv2d_15 (Conv2D)	(None, 200, 200, 8)	1736	concatenate_8[0][0]
concatenate_9 (Concatenate)	(None, 200, 200, 32)	0	conv2d_12[0][0] conv2d_13[0][0] conv2d_14[0][0] conv2d_15[0][0]
conv2d_16 (Conv2D)	(None, 200, 200, 8)	2312	concatenate_9[0][0]
dropout_3 (Dropout)	(None, 200, 200, 8)	0	conv2d_16[0][0]
batch_normalization_3 (BatchNor	(None, 200, 200, 8)	32	dropout_3[0][0]
conv2d_17 (Conv2D)	(None, 200, 200, 16)	1168	batch_normalization_3[0][0]
conv2d_18 (Conv2D)	(None, 200, 200, 16)	1168	batch_normalization_3[0][0]
concatenate_10 (Concatenate)	(None, 200, 200, 32)	0	conv2d_17[0][0] conv2d_18[0][0]
conv2d_19 (Conv2D)	(None, 200, 200, 16)	4624	concatenate_10[0][0]
concatenate_11 (Concatenate)	(None, 200, 200, 48)	0	conv2d_17[0][0] conv2d_18[0][0] conv2d_19[0][0]
conv2d_20 (Conv2D)	(None, 200, 200, 16)	6928	concatenate_11[0][0]
concatenate_12 (Concatenate)	(None, 200, 200, 64)	0	conv2d_17[0][0] conv2d_18[0][0] conv2d_19[0][0] conv2d_20[0][0]
conv2d_21 (Conv2D)	(None, 200, 200, 16)	9232	concatenate_12[0][0]
dropout_4 (Dropout)	(None, 200, 200, 16)	0	conv2d_21[0][0]
batch_normalization_4 (BatchNor	(None, 200, 200, 16)	64	dropout_4[0][0]
conv2d_22 (Conv2D)	(None, 200, 200, 32)	4640	batch_normalization_4[0][0]
conv2d_23 (Conv2D)	(None, 200, 200, 32)	4640	batch_normalization_4[0][0]
concatenate_13 (Concatenate)	(None, 200, 200, 64)	0	conv2d_22[0][0] conv2d_23[0][0]
conv2d_24 (Conv2D)	(None, 200, 200, 32)	18464	concatenate_13[0][0]
concatenate_14 (Concatenate)	(None, 200, 200, 96)	0	conv2d_22[0][0] conv2d_23[0][0] conv2d_24[0][0]
conv2d_25 (Conv2D)	(None, 200, 200, 32)	27680	concatenate_14[0][0]
concatenate_15 (Concatenate)	(None, 200, 200, 128	0	conv2d_22[0][0] conv2d_23[0][0] conv2d_24[0][0] conv2d_25[0][0]
conv2d_26 (Conv2D)	(None, 200, 200, 32)	36896	concatenate_15[0][0]
dropout_5 (Dropout)	(None, 200, 200, 32)	0	conv2d_26[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 100, 100, 32)	0	dropout_5[0][0]
batch_normalization_5 (BatchNor	(None, 100, 100, 32)	128	max_pooling2d_1[0][0]
conv2d_27 (Conv2D)	(None, 100, 100, 64)	2112	batch_normalization_5[0][0]
batch_normalization_6 (BatchNor	(None, 100, 100, 64)	256	conv2d_27[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 50, 50, 64)	0	batch_normalization_6[0][0]
batch_normalization_7 (BatchNor	(None, 50, 50, 64)	256	max_pooling2d_2[0][0]
flatten_1 (Flatten)	(None, 160000)	0	batch_normalization_7[0][0]
dropout_6 (Dropout)	(None, 160000)	0	flatten_1[0][0]
dense_1 (Dense)	(None, 1)	160001	dropout_6[0][0]

Total params: 286,035
Trainable params: 285,655
Non-trainable params: 380

Figure 39: Densenet CNN's Model

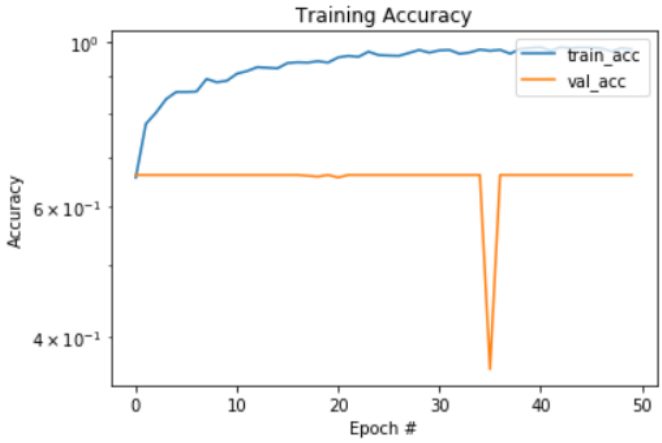


Figure 40: Densenet CNN's Training Accuracy

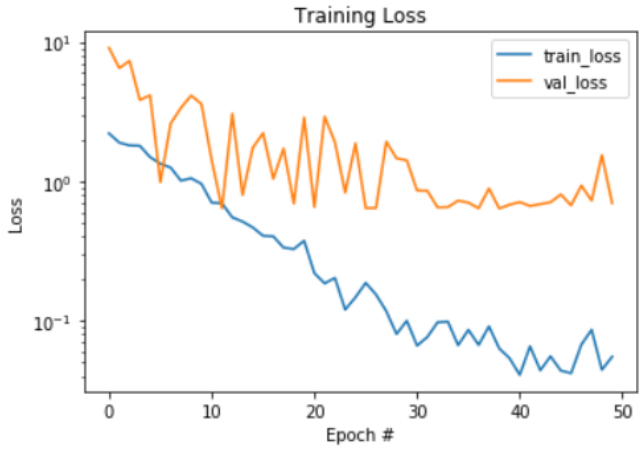


Figure 41: Densenet CNN's Training Loss

```
Loss: 0.6844768905639649
Accuracy: 0.6700000166893005
AUC  is:   0.3152419719583899
              precision    recall  f1-score   support

           0       0.16      0.24      0.20        33
           1       0.51      0.39      0.44        67

    accuracy                           0.34       100
   macro avg       0.34      0.32      0.32       100
weighted avg       0.40      0.34      0.36       100

Confusion Matrix:
 [[ 8 25]
 [41 26]]
```

Figure 42: Densenet CNN's Report

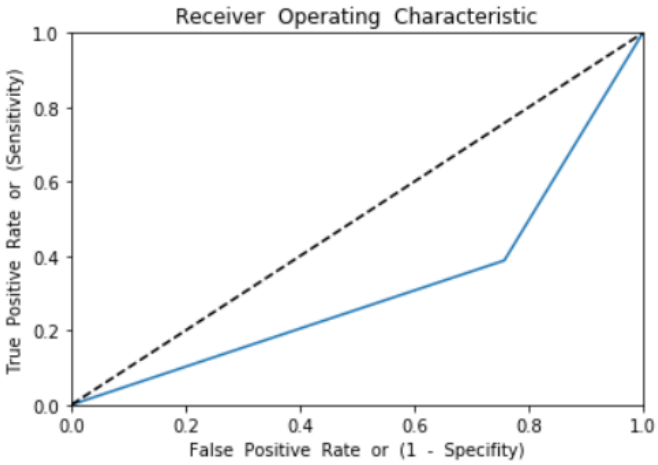


Figure 43: Densenet CNN's ROC

6.2.6 Inception Net Binary Classification

Inception, in its current state, unfortunately, performed the worst, at 0.6677 accuracy and 0.6362 loss. It also has a validation loss of 1.9194, making it the worst in that category, and validation accuracy of 0.6621, on par with DenseNet for the bottom. Inception for binary classification took 34 minutes and 6 seconds, with average times of 38 - 41 seconds. In its current state, we were unable to generate the confusion matrix, precision, F1 Score, and ROC. We'll try to troubleshoot this issue in the future.

Model: "inception_v1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 200, 200, 3)	0	
conv_1_7x7/2 (Conv2D)	(None, 100, 100, 64)	9472	input_1[0][0]
max_pool_1_3x3/2 (MaxPooling2D)	(None, 50, 50, 64)	0	conv_1_7x7/2[0][0]
conv_2a_3x3/1 (Conv2D)	(None, 50, 50, 64)	4160	max_pool_1_3x3/2[0][0]
conv_2b_3x3/1 (Conv2D)	(None, 50, 50, 192)	110784	conv_2a_3x3/1[0][0]
max_pool_2_3x3/2 (MaxPooling2D)	(None, 25, 25, 192)	0	conv_2b_3x3/1[0][0]
conv2d_2 (Conv2D)	(None, 25, 25, 96)	18528	max_pool_2_3x3/2[0][0]
conv2d_4 (Conv2D)	(None, 25, 25, 16)	3088	max_pool_2_3x3/2[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 25, 25, 192)	0	max_pool_2_3x3/2[0][0]
conv2d_1 (Conv2D)	(None, 25, 25, 64)	12352	max_pool_2_3x3/2[0][0]
conv2d_3 (Conv2D)	(None, 25, 25, 128)	110720	conv2d_2[0][0]
conv2d_5 (Conv2D)	(None, 25, 25, 32)	12832	conv2d_4[0][0]
conv2d_6 (Conv2D)	(None, 25, 25, 32)	6176	max_pooling2d_1[0][0]
inception_3a (Concatenate)	(None, 25, 25, 256)	0	conv2d_1[0][0] conv2d_3[0][0] conv2d_5[0][0] conv2d_6[0][0]
conv2d_8 (Conv2D)	(None, 25, 25, 128)	32896	inception_3a[0][0]
conv2d_10 (Conv2D)	(None, 25, 25, 32)	8224	inception_3a[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 25, 25, 256)	0	inception_3a[0][0]
conv2d_7 (Conv2D)	(None, 25, 25, 128)	32896	inception_3a[0][0]
conv2d_9 (Conv2D)	(None, 25, 25, 192)	221376	conv2d_8[0][0]
conv2d_11 (Conv2D)	(None, 25, 25, 96)	76896	conv2d_10[0][0]
conv2d_12 (Conv2D)	(None, 25, 25, 64)	16448	max_pooling2d_2[0][0]
inception_3b (Concatenate)	(None, 25, 25, 480)	0	conv2d_7[0][0] conv2d_9[0][0] conv2d_11[0][0] conv2d_12[0][0]
max_pool_3_3x3/2 (MaxPooling2D)	(None, 13, 13, 480)	0	inception_3b[0][0]
conv2d_14 (Conv2D)	(None, 13, 13, 96)	46176	max_pool_3_3x3/2[0][0]
conv2d_16 (Conv2D)	(None, 13, 13, 16)	7696	max_pool_3_3x3/2[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 13, 13, 480)	0	max_pool_3_3x3/2[0][0]
conv2d_13 (Conv2D)	(None, 13, 13, 192)	92352	max_pool_3_3x3/2[0][0]
conv2d_15 (Conv2D)	(None, 13, 13, 208)	179920	conv2d_14[0][0]
conv2d_17 (Conv2D)	(None, 13, 13, 48)	19248	conv2d_16[0][0]
conv2d_18 (Conv2D)	(None, 13, 13, 64)	30784	max_pooling2d_3[0][0]
inception_4a (Concatenate)	(None, 13, 13, 512)	0	conv2d_13[0][0] conv2d_15[0][0] conv2d_17[0][0] conv2d_18[0][0]
conv2d_21 (Conv2D)	(None, 13, 13, 112)	57456	inception_4a[0][0]
conv2d_23 (Conv2D)	(None, 13, 13, 24)	12312	inception_4a[0][0]
max_pooling2d_4 (MaxPooling2D)	(None, 13, 13, 512)	0	inception_4a[0][0]
conv2d_20 (Conv2D)	(None, 13, 13, 160)	82000	inception_4a[0][0]
conv2d_22 (Conv2D)	(None, 13, 13, 224)	226016	conv2d_21[0][0]
conv2d_24 (Conv2D)	(None, 13, 13, 64)	38464	conv2d_23[0][0]
conv2d_25 (Conv2D)	(None, 13, 13, 64)	32832	max_pooling2d_4[0][0]
inception_4b (Concatenate)	(None, 13, 13, 512)	0	conv2d_20[0][0] conv2d_22[0][0] conv2d_24[0][0] conv2d_25[0][0]
conv2d_27 (Conv2D)	(None, 13, 13, 128)	65664	inception_4b[0][0]
conv2d_29 (Conv2D)	(None, 13, 13, 24)	12312	inception_4b[0][0]
max_pooling2d_5 (MaxPooling2D)	(None, 13, 13, 512)	0	inception_4b[0][0]
conv2d_26 (Conv2D)	(None, 13, 13, 128)	65664	inception_4b[0][0]
conv2d_28 (Conv2D)	(None, 13, 13, 256)	295168	conv2d_27[0][0]
conv2d_30 (Conv2D)	(None, 13, 13, 64)	38464	conv2d_29[0][0]
conv2d_31 (Conv2D)	(None, 13, 13, 64)	32832	max_pooling2d_5[0][0]
inception_4c (Concatenate)	(None, 13, 13, 512)	0	conv2d_26[0][0] conv2d_28[0][0] conv2d_30[0][0] conv2d_31[0][0]
conv2d_33 (Conv2D)	(None, 13, 13, 144)	73872	inception_4c[0][0]
conv2d_35 (Conv2D)	(None, 13, 13, 32)	16416	inception_4c[0][0]
max_pooling2d_6 (MaxPooling2D)	(None, 13, 13, 512)	0	inception_4c[0][0]
conv2d_32 (Conv2D)	(None, 13, 13, 112)	57456	inception_4c[0][0]
conv2d_34 (Conv2D)	(None, 13, 13, 288)	373536	conv2d_33[0][0]
conv2d_36 (Conv2D)	(None, 13, 13, 64)	51264	conv2d_35[0][0]
conv2d_37 (Conv2D)	(None, 13, 13, 64)	32832	max_pooling2d_6[0][0]
inception_4d (Concatenate)	(None, 13, 13, 528)	0	conv2d_32[0][0] conv2d_34[0][0] conv2d_36[0][0] conv2d_37[0][0]
conv2d_40 (Conv2D)	(None, 13, 13, 160)	84640	inception_4d[0][0]
conv2d_42 (Conv2D)	(None, 13, 13, 32)	16928	inception_4d[0][0]
max_pooling2d_7 (MaxPooling2D)	(None, 13, 13, 528)	0	inception_4d[0][0]
conv2d_39 (Conv2D)	(None, 13, 13, 256)	135424	inception_4d[0][0]
conv2d_41 (Conv2D)	(None, 13, 13, 320)	461120	conv2d_40[0][0]
conv2d_43 (Conv2D)	(None, 13, 13, 128)	102528	conv2d_42[0][0]
conv2d_44 (Conv2D)	(None, 13, 13, 128)	67712	max_pooling2d_7[0][0]
inception_4e (Concatenate)	(None, 13, 13, 832)	0	conv2d_39[0][0] conv2d_41[0][0] conv2d_43[0][0] conv2d_44[0][0]
max_pool_4_3x3/2 (MaxPooling2D)	(None, 7, 7, 832)	0	inception_4e[0][0]
conv2d_46 (Conv2D)	(None, 7, 7, 160)	133280	max_pool_4_3x3/2[0][0]
conv2d_48 (Conv2D)	(None, 7, 7, 32)	26656	max_pool_4_3x3/2[0][0]
max_pooling2d_8 (MaxPooling2D)	(None, 7, 7, 832)	0	max_pool_4_3x3/2[0][0]
conv2d_45 (Conv2D)	(None, 7, 7, 256)	213248	max_pool_4_3x3/2[0][0]
conv2d_47 (Conv2D)	(None, 7, 7, 320)	461120	conv2d_46[0][0]
conv2d_49 (Conv2D)	(None, 7, 7, 128)	102528	conv2d_48[0][0]
conv2d_50 (Conv2D)	(None, 7, 7, 128)	106624	max_pooling2d_8[0][0]
inception_5a (Concatenate)	(None, 7, 7, 832)	0	conv2d_45[0][0] conv2d_47[0][0] conv2d_49[0][0] conv2d_50[0][0]
conv2d_52 (Conv2D)	(None, 7, 7, 192)	159936	inception_5a[0][0]
conv2d_54 (Conv2D)	(None, 7, 7, 48)	39984	inception_5a[0][0]
max_pooling2d_9 (MaxPooling2D)	(None, 7, 7, 832)	0	inception_5a[0][0]
average_pooling2d_1 (AveragePoo	(None, 3, 3, 512)	0	inception_4a[0][0]
average_pooling2d_2 (AveragePoo	(None, 3, 3, 528)	0	inception_4d[0][0]
conv2d_51 (Conv2D)	(None, 7, 7, 384)	319872	inception_5a[0][0]
conv2d_53 (Conv2D)	(None, 7, 7, 384)	663936	conv2d_52[0][0]
conv2d_55 (Conv2D)	(None, 7, 7, 128)	153728	conv2d_54[0][0]
conv2d_56 (Conv2D)	(None, 7, 7, 128)	106624	max_pooling2d_9[0][0]
conv2d_19 (Conv2D)	(None, 3, 3, 128)	65664	average_pooling2d_1[0][0]
conv2d_38 (Conv2D)	(None, 3, 3, 128)	67712	average_pooling2d_2[0][0]
inception_5b (Concatenate)	(None, 7, 7, 1024)	0	conv2d_51[0][0] conv2d_53[0][0] conv2d_55[0][0] conv2d_56[0][0]
flatten_1 (Flatten)	(None, 1152)	0	conv2d_19[0][0]
flatten_2 (Flatten)	(None, 1152)	0	conv2d_38[0][0]
avg_pool_5_3x3/1 (GlobalAverage	(None, 1024)	0	inception_5b[0][0]
dense_1 (Dense)	(None, 1024)	1180672	flatten_1[0][0]
dense_2 (Dense)	(None, 1024)	1180672	flatten_2[0][0]
dropout_3 (Dropout)	(None, 1024)	0	avg_pool_5_3x3/1[0][0]
dropout_1 (Dropout)	(None, 1024)	0	dense_1[0][0]
dropout_2 (Dropout)	(None, 1024)	0	dense_2[0][0]
output (Dense)	(None, 1)	1025	dropout_3[0][0]
auxilliary_output_1 (Dense)	(None, 1)	1025	dropout_1[0][0]
auxilliary_output_2 (Dense)	(None, 1)	1025	dropout_2[0][0]

Total params: 8,471,347
Trainable params: 8,471,347
Non-trainable params: 0

Figure 44: Inception CNN's Model

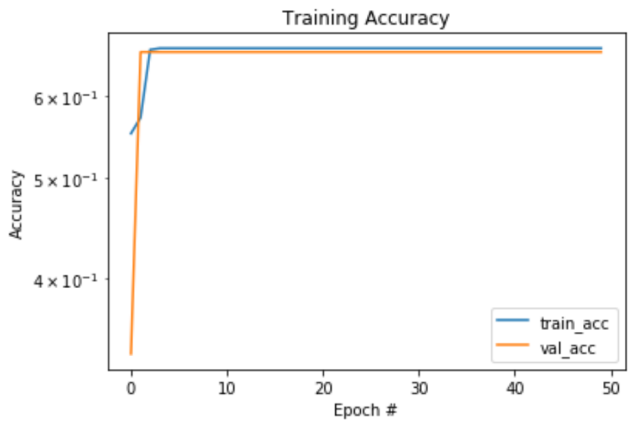


Figure 45: Inception CNN's Training Accuracy

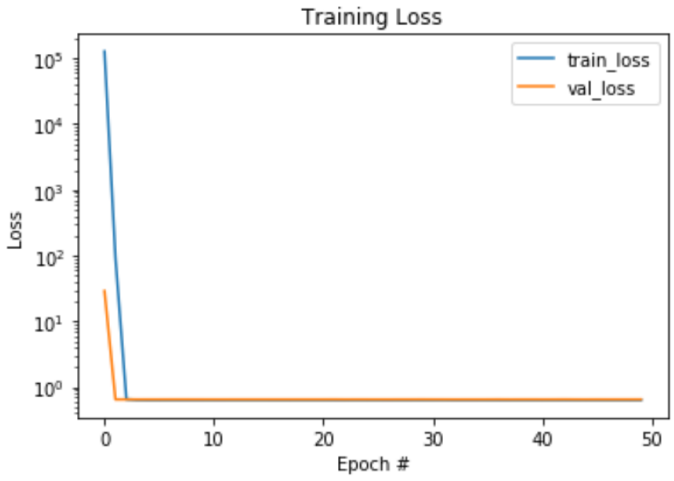


Figure 46: Inception CNN's Training Loss

```
Loss: 1.902536907196045
Accuracy: 0.6217450499534607
```

Figure 47: Inception CNN's Report

6.2.7 Xception Net Binary Classification

Xception Net's training accuracy also increased gradually, finishing off at 0.9927, one of the models with the highest accuracy amongst all the models. The validation accuracy, on the other hand, was rather unpredictable - though it does have a general increase, it was marred by drops. Both the training loss and the validation loss decreases gradually, though the training loss decreases faster. The Xception model took an average of 105 to 110 seconds per epoch, with the overall total time being 1 hour 30 minutes and 30 seconds.

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 200, 200, 3)	0	
conv2d_1 (Conv2D)	(None, 100, 100, 8)	224	input_1[0][0]
batch_normalization_1 (BatchNor	(None, 100, 100, 8)	32	conv2d_1[0][0]
dropout_1 (Dropout)	(None, 100, 100, 8)	0	batch_normalization_1[0][0]
conv2d_2 (Conv2D)	(None, 100, 100, 16)	1168	dropout_1[0][0]
batch_normalization_2 (BatchNor	(None, 100, 100, 16)	64	conv2d_2[0][0]
dropout_2 (Dropout)	(None, 100, 100, 16)	0	batch_normalization_2[0][0]
batch_normalization_3 (BatchNor	(None, 100, 100, 16)	64	dropout_2[0][0]
dropout_3 (Dropout)	(None, 100, 100, 16)	0	batch_normalization_3[0][0]
depthwise_conv2d_1 (DepthwiseCo	(None, 100, 100, 16)	160	dropout_2[0][0]
batch_normalization_4 (BatchNor	(None, 100, 100, 16)	64	dropout_3[0][0]
depthwise_conv2d_2 (DepthwiseCo	(None, 100, 100, 16)	160	depthwise_conv2d_1[0][0]
dropout_4 (Dropout)	(None, 100, 100, 16)	0	batch_normalization_4[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 99, 99, 16)	0	depthwise_conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 99, 99, 16)	1040	dropout_4[0][0]
concatenate_1 (Concatenate)	(None, 99, 99, 32)	0	max_pooling2d_1[0][0] conv2d_3[0][0]
batch_normalization_5 (BatchNor	(None, 99, 99, 32)	128	concatenate_1[0][0]
dropout_5 (Dropout)	(None, 99, 99, 32)	0	batch_normalization_5[0][0]
activation_1 (Activation)	(None, 99, 99, 32)	0	concatenate_1[0][0]
batch_normalization_6 (BatchNor	(None, 99, 99, 32)	128	dropout_5[0][0]
conv2d_4 (Conv2D)	(None, 99, 99, 64)	18496	activation_1[0][0]
dropout_6 (Dropout)	(None, 99, 99, 32)	0	batch_normalization_6[0][0]
depthwise_conv2d_3 (DepthwiseCo	(None, 99, 99, 64)	640	conv2d_4[0][0]
batch_normalization_7 (BatchNor	(None, 99, 99, 32)	128	dropout_6[0][0]
depthwise_conv2d_4 (DepthwiseCo	(None, 99, 99, 64)	640	depthwise_conv2d_3[0][0]
dropout_7 (Dropout)	(None, 99, 99, 32)	0	batch_normalization_7[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 98, 98, 64)	0	depthwise_conv2d_4[0][0]
conv2d_5 (Conv2D)	(None, 98, 98, 64)	8256	dropout_7[0][0]
concatenate_2 (Concatenate)	(None, 98, 98, 128)	0	max_pooling2d_2[0][0] conv2d_5[0][0]
activation_2 (Activation)	(None, 98, 98, 128)	0	concatenate_2[0][0]
conv2d_6 (Conv2D)	(None, 98, 98, 64)	73792	activation_2[0][0]
batch_normalization_8 (BatchNor	(None, 98, 98, 64)	256	conv2d_6[0][0]
dropout_8 (Dropout)	(None, 98, 98, 64)	0	batch_normalization_8[0][0]
conv2d_7 (Conv2D)	(None, 98, 98, 32)	18464	dropout_8[0][0]
batch_normalization_9 (BatchNor	(None, 98, 98, 32)	128	conv2d_7[0][0]
dropout_9 (Dropout)	(None, 98, 98, 32)	0	batch_normalization_9[0][0]
flatten_1 (Flatten)	(None, 307328)	0	dropout_9[0][0]
dense_1 (Dense)	(None, 1)	307329	flatten_1[0][0]

Total params: 431,361
Trainable params: 430,865
Non-trainable params: 496

Figure 48: Xception CNN's Model

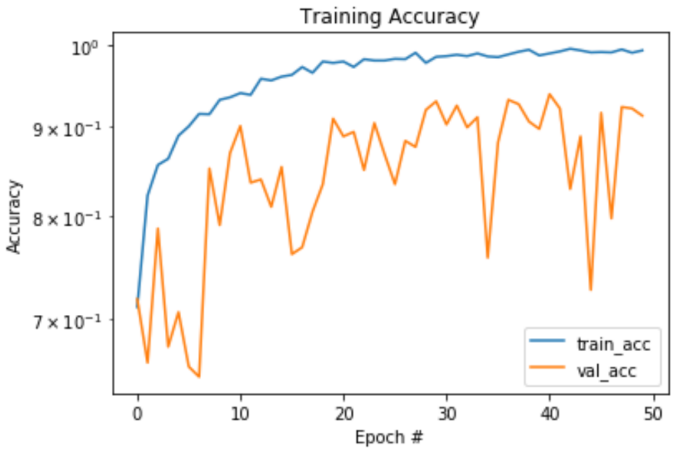


Figure 49: Xception CNN's Training Accuracy

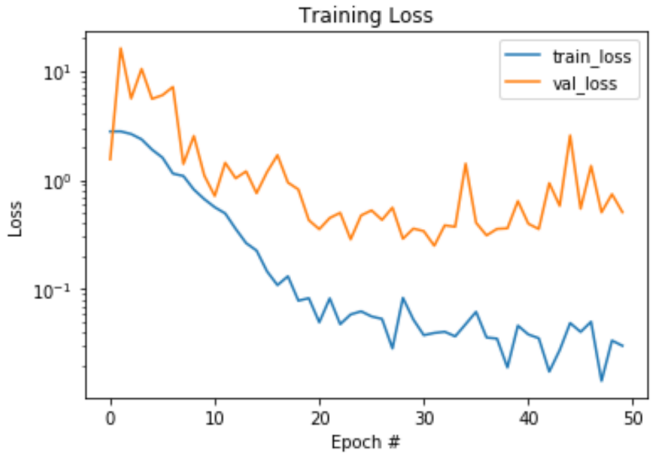


Figure 50: Xception CNN's Training Loss

```
Loss: 0.38816532760858535
Accuracy: 0.9100000262260437
AUC  is:   0.9097693351424694
              precision    recall  f1-score   support

           0       0.83      0.91      0.87        33
           1       0.95      0.91      0.93        67

    accuracy                           0.91       100
   macro avg       0.89      0.91      0.90       100
weighted avg       0.91      0.91      0.91       100

Confusion Matrix:
 [[30  3]
 [ 6 61]]
```

Figure 51: Xception CNN's Report

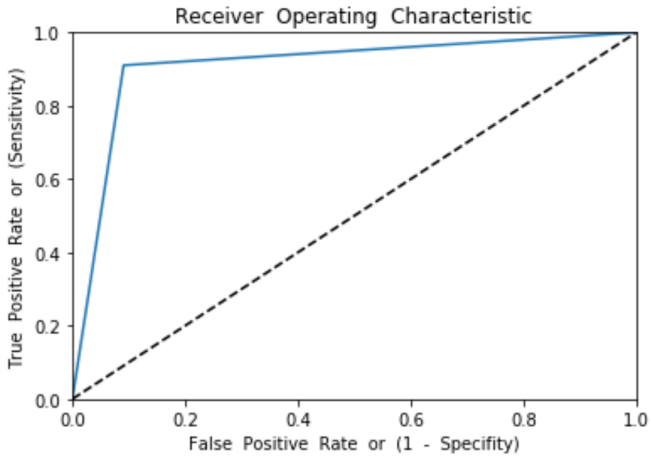


Figure 52: Xception CNN's ROC

6.2.8 Custom Model for Binary Classification

Our Custom Model's training accuracy increased gradually, finishing off at 0.9772. The validation accuracy, while it has a general increase, it has fluctuations along the way. The training loss has a sharp decrease at first, followed by an exponential decrease that is more gradual. In the meantime, the validation loss more of less fluctuated around 0.3000. Our Custom CNN Model took an average of 7 to 9 seconds per epoch, with the overall total time being 6 minutes and 50 seconds.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 198, 198, 16)      448
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 99, 99, 16)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 99, 99, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 97, 97, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 48, 48, 32)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 48, 48, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 73728)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 73729
=================================================================
Total params: 78,817
Trainable params: 78,817
Non-trainable params: 0
```

Figure 53: Custom CNN's Model

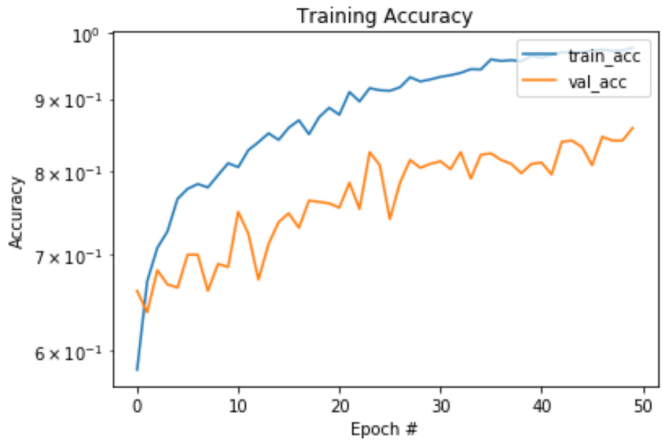


Figure 54: Custom CNN's Training Accuracy

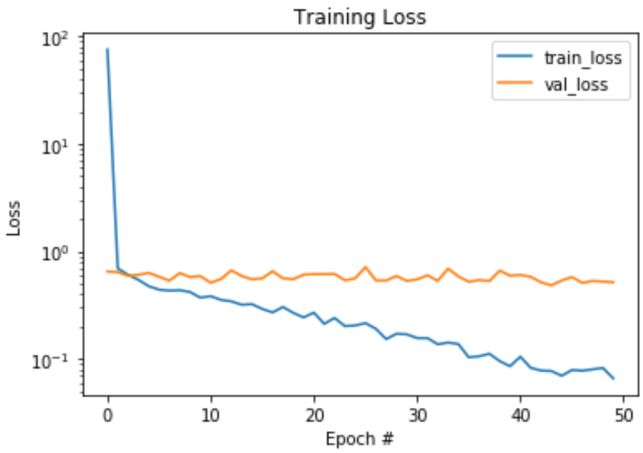


Figure 55: Custom CNN's Training Loss

```
Loss: 0.31230961106717586
Accuracy: 0.8899999856948853
AUC  is:   0.88715513342379
              precision    recall  f1-score   support

           0       0.81      0.88      0.84        33
           1       0.94      0.90      0.92        67

    accuracy                           0.89       100
   macro avg       0.87      0.89      0.88       100
weighted avg       0.89      0.89      0.89       100

Confusion Matrix:
 [[29  4]
 [ 7 60]]
```

Figure 56: Custom CNN's Report

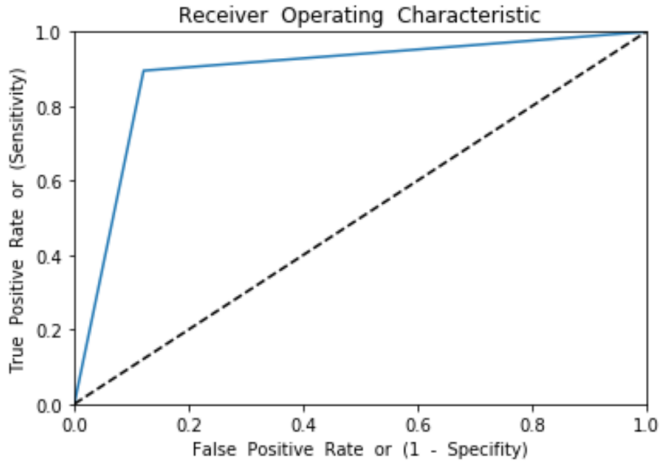


Figure 57: Custom CNN's ROC

6.3 Results for Multi-Class Classification

The results of the Multi-Class Classification are given below:

Evaluator - Multi-Class	**Custom Model**	**Basic CNN**	**VGG Network**	**Resnet**	**Densenet**	**Inception**	**Xception**
Final Training Loss	0.0993	5.1126e-8	0.0881	0.5500	4.1130	1.1001	0.0645
Final Training Accuracy	0.9741	1.0000	0.9737	0.7931	0.5588	0.3276	0.9897
Final Validation Loss	0.6908	1.2789	0.3363	0.4655	6.8057	3.2959	4.8047
Final Validation Accuracy	0.8638	0.8569	0.8810	0.8466	0.5575	0.3379	0.7776
Final Evaluation Loss	0.2956	1.0033	0.2974	0.4288	6.2542	???	4.9358
Final Evaluation Accuracy	0.8900	0.8600	0.8900	0.8600	0.5933	???	0.7400
Time Taken / Epoch	8 - 10 sec	5 - 7 sec	24 - 28 sec	141 - 150 sec	210 - 230 sec	39 - 40 sec	150 - 165 sec
Time Taken / Overall	7 min 41 sec	4 min 42 sec	21 min 49 sec	2 hr 3 min	2 h 59 min	33 min 20 sec	2 hr 12 min

6.3.1 Basic CNN Multi-Class Classification

For the Basic model, the training accuracy has increased tremendously since the beginning, reaching a complete 1.0000 at the end. As for validation accuracy, it fluctuated by a small margin in the beginning, but in the end, it was more or less the same, ending at 0.8569. The training loss decreased by several factors, while the validation loss decreased at 1.2789. The Basic CNN took an average of 7 to 9 seconds per epoch, and in the end, it took 4 minutes and 42 seconds to be fully finished.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 198, 198, 16)      448
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 99, 99, 16)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 156816)            0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 470451
=================================================================
Total params: 470,899
Trainable params: 470,899
Non-trainable params: 0
```

Figure 58: Multiclass Basic CNN's Model

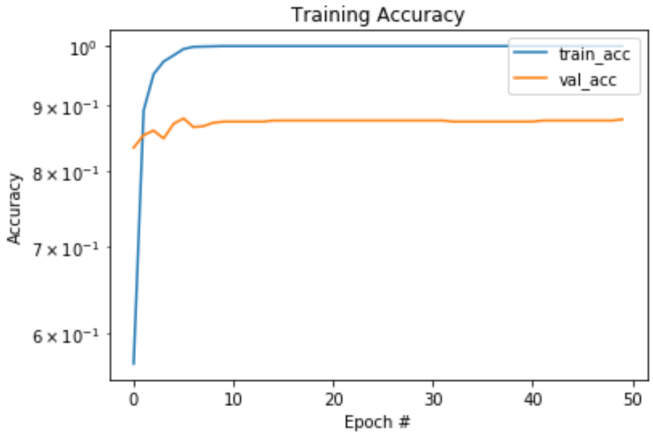


Figure 59: Multiclass Basic CNN's Training Accuracy

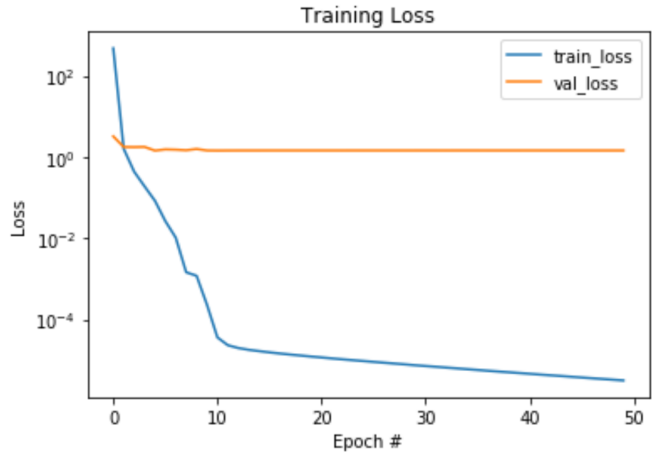


Figure 60: Multiclass Basic CNN's Training Loss

```
Loss: 0.9762647342681885
Accuracy: 0.8899999856948853
Confusion Matrix:
 [[31  2  0]
 [ 5 22  1]
 [ 3  0 36]]
             precision    recall  f1-score  support   pred
0            0.794872  0.939394  0.861111     33.0   39.0
1            0.916667  0.785714  0.846154     28.0   24.0
2            0.972973  0.923077  0.947368     39.0   37.0
avg / total  0.898434  0.890000  0.890563    100.0  100.0
```

Figure 61: Multiclass Basic CNN's Report

6.3.2 VGGnet Multi-Class Classification

For the VGGnet model, the training accuracy was more or less the same for the first 12 epochs, then there was a sharp increase to 0.800, followed by a more gradual increase to 0.9737. As for the validation accuracy, it follows the same trend as of the training accuracy, but it went stagnant and settled around 0.88. The training loss decreased sharply for the first epoch, then it was stagnant at a value of 1.00 for until the 15th epoch. It then decreased exponentially. The validation loss also followed more or less the same trend as the training loss, but it started at around 1.10. Soon after, the validation loss decreased slowly to 0.295. VGGnet Model took an average of 24 to 28 seconds per epoch, while the overall process took 21 minutes and 45 seconds.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 200, 200, 16)      448
_________________________________________________________________
dropout_1 (Dropout)          (None, 200, 200, 16)      0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 200, 200, 16)      2320
_________________________________________________________________
dropout_2 (Dropout)          (None, 200, 200, 16)      0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 100, 100, 16)      0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 100, 100, 32)      4640
_________________________________________________________________
dropout_3 (Dropout)          (None, 100, 100, 32)      0
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 100, 100, 32)      9248
_________________________________________________________________
dropout_4 (Dropout)          (None, 100, 100, 32)      0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 50, 50, 32)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 50, 50, 64)        18496
_________________________________________________________________
flatten_1 (Flatten)          (None, 160000)            0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 480003
=================================================================
Total params: 515,155
Trainable params: 515,155
Non-trainable params: 0
```

Figure 62: Multiclass VGGnet CNN's Model

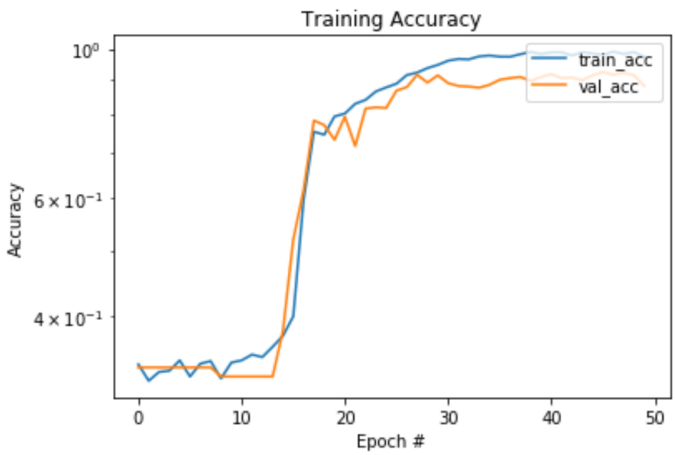


Figure 63: Multiclass VGGnet CNN's Training Accuracy

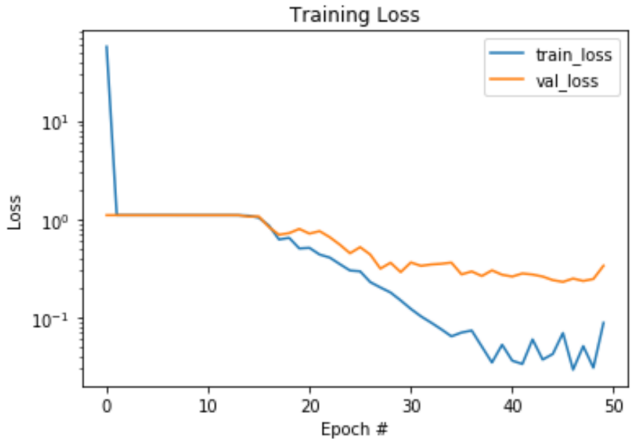


Figure 64: Multiclass VGGnet CNN's Training Loss

```
Loss: 0.2974425405263901
Accuracy: 0.8899999856948853
Confusion Matrix:
 [[25  4  4]
 [ 1 25  2]
 [ 0  0 39]]
             precision    recall  f1-score  support   pred
0            0.961538  0.757576  0.847458     33.0   26.0
1            0.862069  0.892857  0.877193     28.0   29.0
2            0.866667  1.000000  0.928571     39.0   45.0
avg / total  0.896687  0.890000  0.887418    100.0  100.0
```

Figure 65: Multiclass VGGnet CNN's Report

6.3.3 ResNet Multi-Class Classification

For the ResNet model, the training accuracy increased gradually, though it saw a small decrease from the 4th to 5th epoch. the training loss, on the other hand, had a sharp decrease after the 1st epoch, then it decreased exponentially. The validation accuracy and validation loss, however, was more or less unpredictable, fluctuating in large amounts. Resnet took an average of 141 to 150 seconds per epoch, while the overall process took 2 hours, 3 minutes.

Model: "Resnet"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 200, 200, 3)	0	
conv2d_1 (Conv2D)	(None, 200, 200, 16)	448	input_1[0][0]
batch_normalization_1 (BatchNor	(None, 200, 200, 16)	64	conv2d_1[0][0]
activation_2 (Activation)	(None, 200, 200, 16)	0	batch_normalization_2[0][0]
conv2d_3 (Conv2D)	(None, 200, 200, 16)	2320	activation_2[0][0]
conv2d_4 (Conv2D)	(None, 200, 200, 16)	2320	activation_1[0][0]
add_1 (Add)	(None, 200, 200, 16)	0	conv2d_3[0][0] conv2d_4[0][0]
dropout_1 (Dropout)	(None, 200, 200, 16)	0	add_1[0][0]
batch_normalization_3 (BatchNor	(None, 200, 200, 16)	64	dropout_1[0][0]
activation_3 (Activation)	(None, 200, 200, 16)	0	batch_normalization_3[0][0]
conv2d_5 (Conv2D)	(None, 200, 200, 16)	2320	activation_3[0][0]
batch_normalization_4 (BatchNor	(None, 200, 200, 16)	64	conv2d_5[0][0]
activation_4 (Activation)	(None, 200, 200, 16)	0	batch_normalization_4[0][0]
conv2d_6 (Conv2D)	(None, 200, 200, 16)	2320	activation_4[0][0]
conv2d_7 (Conv2D)	(None, 200, 200, 16)	2320	activation_3[0][0]
add_2 (Add)	(None, 200, 200, 16)	0	conv2d_6[0][0] conv2d_7[0][0]
dropout_2 (Dropout)	(None, 200, 200, 16)	0	add_2[0][0]
batch_normalization_5 (BatchNor	(None, 200, 200, 16)	64	dropout_2[0][0]
activation_5 (Activation)	(None, 200, 200, 16)	0	batch_normalization_5[0][0]
conv2d_8 (Conv2D)	(None, 200, 200, 16)	2320	activation_5[0][0]
batch_normalization_6 (BatchNor	(None, 200, 200, 16)	64	conv2d_8[0][0]
activation_6 (Activation)	(None, 200, 200, 16)	0	batch_normalization_6[0][0]
conv2d_9 (Conv2D)	(None, 200, 200, 16)	2320	activation_6[0][0]
conv2d_10 (Conv2D)	(None, 200, 200, 16)	2320	activation_5[0][0]
add_3 (Add)	(None, 200, 200, 16)	0	conv2d_9[0][0] conv2d_10[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 100, 100, 16)	0	add_3[0][0]
dropout_3 (Dropout)	(None, 100, 100, 16)	0	max_pooling2d_1[0][0]
batch_normalization_7 (BatchNor	(None, 100, 100, 16)	64	dropout_3[0][0]
activation_7 (Activation)	(None, 100, 100, 16)	0	batch_normalization_7[0][0]
conv2d_11 (Conv2D)	(None, 100, 100, 32)	4640	activation_7[0][0]
batch_normalization_8 (BatchNor	(None, 100, 100, 32)	128	conv2d_11[0][0]
activation_8 (Activation)	(None, 100, 100, 32)	0	batch_normalization_8[0][0]
conv2d_12 (Conv2D)	(None, 100, 100, 32)	9248	activation_8[0][0]
conv2d_13 (Conv2D)	(None, 100, 100, 32)	4640	activation_7[0][0]
add_4 (Add)	(None, 100, 100, 32)	0	conv2d_12[0][0] conv2d_13[0][0]
dropout_4 (Dropout)	(None, 100, 100, 32)	0	add_4[0][0]
batch_normalization_9 (BatchNor	(None, 100, 100, 32)	128	dropout_4[0][0]
activation_9 (Activation)	(None, 100, 100, 32)	0	batch_normalization_9[0][0]
conv2d_14 (Conv2D)	(None, 100, 100, 32)	9248	activation_9[0][0]
batch_normalization_10 (BatchNo	(None, 100, 100, 32)	128	conv2d_14[0][0]
activation_10 (Activation)	(None, 100, 100, 32)	0	batch_normalization_10[0][0]
conv2d_15 (Conv2D)	(None, 100, 100, 32)	9248	activation_10[0][0]
conv2d_16 (Conv2D)	(None, 100, 100, 32)	9248	activation_9[0][0]
add_5 (Add)	(None, 100, 100, 32)	0	conv2d_15[0][0] conv2d_16[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 50, 50, 32)	0	add_5[0][0]
dropout_5 (Dropout)	(None, 50, 50, 32)	0	max_pooling2d_2[0][0]
batch_normalization_11 (BatchNo	(None, 50, 50, 32)	128	dropout_5[0][0]
activation_11 (Activation)	(None, 50, 50, 32)	0	batch_normalization_11[0][0]
conv2d_17 (Conv2D)	(None, 50, 50, 64)	18496	activation_11[0][0]
batch_normalization_12 (BatchNo	(None, 50, 50, 64)	256	conv2d_17[0][0]
activation_12 (Activation)	(None, 50, 50, 64)	0	batch_normalization_12[0][0]
conv2d_18 (Conv2D)	(None, 50, 50, 64)	36928	activation_12[0][0]
conv2d_19 (Conv2D)	(None, 50, 50, 64)	18496	activation_11[0][0]
add_6 (Add)	(None, 50, 50, 64)	0	conv2d_18[0][0] conv2d_19[0][0]
dropout_6 (Dropout)	(None, 50, 50, 64)	0	add_6[0][0]
batch_normalization_13 (BatchNo	(None, 50, 50, 64)	256	dropout_6[0][0]
activation_13 (Activation)	(None, 50, 50, 64)	0	batch_normalization_13[0][0]
flatten_1 (Flatten)	(None, 160000)	0	activation_13[0][0]
dropout_7 (Dropout)	(None, 160000)	0	flatten_1[0][0]
dense_1 (Dense)	(None, 64)	10240064	dropout_7[0][0]
dropout_8 (Dropout)	(None, 64)	0	dense_1[0][0]
dense_2 (Dense)	(None, 3)	195	dropout_8[0][0]

Total params: 10,383,251
Trainable params: 10,382,515
Non-trainable params: 736

Figure 66: Multiclass Resnet CNN's Model

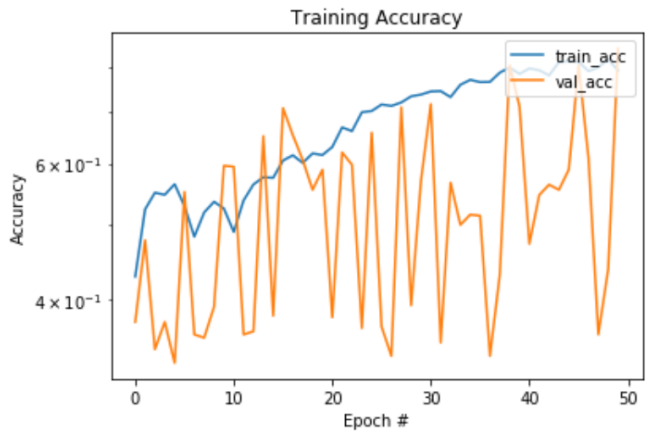


Figure 67: Multiclass Resnet CNN's Training Accuracy

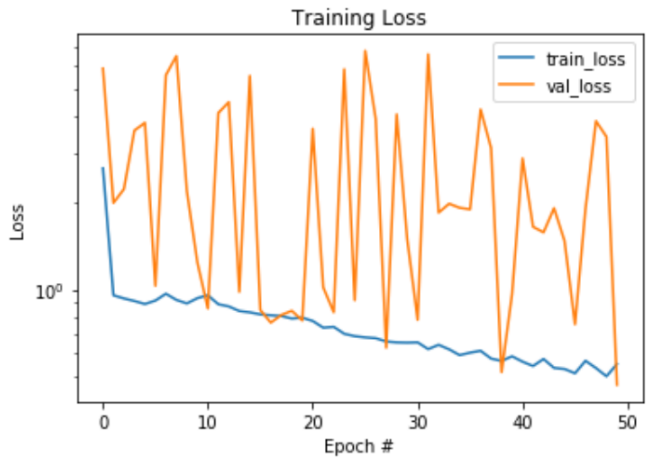


Figure 68: Multiclass Resnet CNN's Training Loss

```
Loss: 0.42883082389831545
Accuracy: 0.8600000143051147
Confusion Matrix:
 [[ 8 15 10]
 [ 7 12  9]
 [11  9 19]]
             precision    recall  f1-score  support   pred
0            0.307692  0.242424  0.271186     33.0   26.0
1            0.333333  0.428571  0.375000     28.0   36.0
2            0.500000  0.487179  0.493506     39.0   38.0
avg / total  0.389872  0.390000  0.386959    100.0  100.0
```

Figure 69: Multiclass Resnet CNN's Report

6.3.4 DenseNet Multi-Class Classification

DenseNet for Multi-class Classification did not work as intended. Initially, for the first 10-12 epochs, it has a promising accuracy, until it dropped to 0.62 and eventually decreasing slowly until 0.5588. The training and validation loss was decreasing until after the 40th epoch, whereby both of them would increase to 6.80 and 6.25. The training took 3 hours, with an average of 210 - 230 seconds per epoch. This would show that at the current state and model, DenseNet is not a model that we can use, lacking in both speed and accuracy.

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_3 (InputLayer)	(None, 200, 200, 3)	0	
conv2d_55 (Conv2D)	(None, 200, 200, 2)	56	input_3[0][0]
conv2d_56 (Conv2D)	(None, 200, 200, 2)	38	conv2d_55[0][0]
conv2d_57 (Conv2D)	(None, 200, 200, 2)	38	conv2d_55[0][0]
concatenate_31 (Concatenate)	(None, 200, 200, 4)	0	conv2d_56[0][0] conv2d_57[0][0]
conv2d_58 (Conv2D)	(None, 200, 200, 2)	74	concatenate_31[0][0]
concatenate_32 (Concatenate)	(None, 200, 200, 6)	0	conv2d_56[0][0] conv2d_57[0][0] conv2d_58[0][0]
conv2d_59 (Conv2D)	(None, 200, 200, 2)	110	concatenate_32[0][0]
concatenate_33 (Concatenate)	(None, 200, 200, 8)	0	conv2d_56[0][0] conv2d_57[0][0] conv2d_58[0][0] conv2d_59[0][0]
conv2d_60 (Conv2D)	(None, 200, 200, 2)	146	concatenate_33[0][0]
dropout_13 (Dropout)	(None, 200, 200, 2)	0	conv2d_60[0][0]
batch_normalization_15 (BatchNo	(None, 200, 200, 2)	8	dropout_13[0][0]
conv2d_61 (Conv2D)	(None, 200, 200, 4)	76	batch_normalization_15[0][0]
conv2d_62 (Conv2D)	(None, 200, 200, 4)	76	batch_normalization_15[0][0]
concatenate_34 (Concatenate)	(None, 200, 200, 8)	0	conv2d_61[0][0] conv2d_62[0][0]
conv2d_63 (Conv2D)	(None, 200, 200, 4)	292	concatenate_34[0][0]
concatenate_35 (Concatenate)	(None, 200, 200, 12)	0	conv2d_61[0][0] conv2d_62[0][0] conv2d_63[0][0]
conv2d_64 (Conv2D)	(None, 200, 200, 4)	436	concatenate_35[0][0]
concatenate_36 (Concatenate)	(None, 200, 200, 16)	0	conv2d_61[0][0] conv2d_62[0][0] conv2d_63[0][0] conv2d_64[0][0]
conv2d_65 (Conv2D)	(None, 200, 200, 4)	580	concatenate_36[0][0]
dropout_14 (Dropout)	(None, 200, 200, 4)	0	conv2d_65[0][0]
batch_normalization_16 (BatchNo	(None, 200, 200, 4)	16	dropout_14[0][0]
conv2d_66 (Conv2D)	(None, 200, 200, 8)	296	batch_normalization_16[0][0]
conv2d_67 (Conv2D)	(None, 200, 200, 8)	296	batch_normalization_16[0][0]
concatenate_37 (Concatenate)	(None, 200, 200, 16)	0	conv2d_66[0][0] conv2d_67[0][0]
conv2d_68 (Conv2D)	(None, 200, 200, 8)	1160	concatenate_37[0][0]
concatenate_38 (Concatenate)	(None, 200, 200, 24)	0	conv2d_66[0][0] conv2d_67[0][0] conv2d_68[0][0]
conv2d_69 (Conv2D)	(None, 200, 200, 8)	1736	concatenate_38[0][0]
concatenate_39 (Concatenate)	(None, 200, 200, 32)	0	conv2d_66[0][0] conv2d_67[0][0] conv2d_68[0][0] conv2d_69[0][0]
conv2d_70 (Conv2D)	(None, 200, 200, 8)	2312	concatenate_39[0][0]
dropout_15 (Dropout)	(None, 200, 200, 8)	0	conv2d_70[0][0]
batch_normalization_17 (BatchNo	(None, 200, 200, 8)	32	dropout_15[0][0]
conv2d_71 (Conv2D)	(None, 200, 200, 16)	1168	batch_normalization_17[0][0]
conv2d_72 (Conv2D)	(None, 200, 200, 16)	1168	batch_normalization_17[0][0]
concatenate_40 (Concatenate)	(None, 200, 200, 32)	0	conv2d_71[0][0] conv2d_72[0][0]
conv2d_73 (Conv2D)	(None, 200, 200, 16)	4624	concatenate_40[0][0]
concatenate_41 (Concatenate)	(None, 200, 200, 48)	0	conv2d_71[0][0] conv2d_72[0][0] conv2d_73[0][0]
conv2d_74 (Conv2D)	(None, 200, 200, 16)	6928	concatenate_41[0][0]
concatenate_42 (Concatenate)	(None, 200, 200, 64)	0	conv2d_71[0][0] conv2d_72[0][0] conv2d_73[0][0] conv2d_74[0][0]
conv2d_75 (Conv2D)	(None, 200, 200, 16)	9232	concatenate_42[0][0]
dropout_16 (Dropout)	(None, 200, 200, 16)	0	conv2d_75[0][0]
batch_normalization_18 (BatchNo	(None, 200, 200, 16)	64	dropout_16[0][0]
conv2d_76 (Conv2D)	(None, 200, 200, 32)	4640	batch_normalization_18[0][0]
conv2d_77 (Conv2D)	(None, 200, 200, 32)	4640	batch_normalization_18[0][0]
concatenate_43 (Concatenate)	(None, 200, 200, 64)	0	conv2d_76[0][0] conv2d_77[0][0]
conv2d_78 (Conv2D)	(None, 200, 200, 32)	18464	concatenate_43[0][0]
concatenate_44 (Concatenate)	(None, 200, 200, 96)	0	conv2d_76[0][0] conv2d_77[0][0] conv2d_78[0][0]
conv2d_79 (Conv2D)	(None, 200, 200, 32)	27680	concatenate_44[0][0]
concatenate_45 (Concatenate)	(None, 200, 200, 128	0	conv2d_76[0][0] conv2d_77[0][0] conv2d_78[0][0] conv2d_79[0][0]
conv2d_80 (Conv2D)	(None, 200, 200, 32)	36896	concatenate_45[0][0]
dropout_17 (Dropout)	(None, 200, 200, 32)	0	conv2d_80[0][0]
max_pooling2d_5 (MaxPooling2D)	(None, 100, 100, 32)	0	dropout_17[0][0]
batch_normalization_19 (BatchNo	(None, 100, 100, 32)	128	max_pooling2d_5[0][0]
conv2d_81 (Conv2D)	(None, 100, 100, 64)	2112	batch_normalization_19[0][0]
batch_normalization_20 (BatchNo	(None, 100, 100, 64)	256	conv2d_81[0][0]
max_pooling2d_6 (MaxPooling2D)	(None, 50, 50, 64)	0	batch_normalization_20[0][0]
batch_normalization_21 (BatchNo	(None, 50, 50, 64)	256	max_pooling2d_6[0][0]
flatten_3 (Flatten)	(None, 160000)	0	batch_normalization_21[0][0]
dropout_18 (Dropout)	(None, 160000)	0	flatten_3[0][0]
dense_3 (Dense)	(None, 3)	480003	dropout_18[0][0]

Total params: 606,037
Trainable params: 605,657
Non-trainable params: 380

Figure 70: Multiclass Densenet CNN's Model

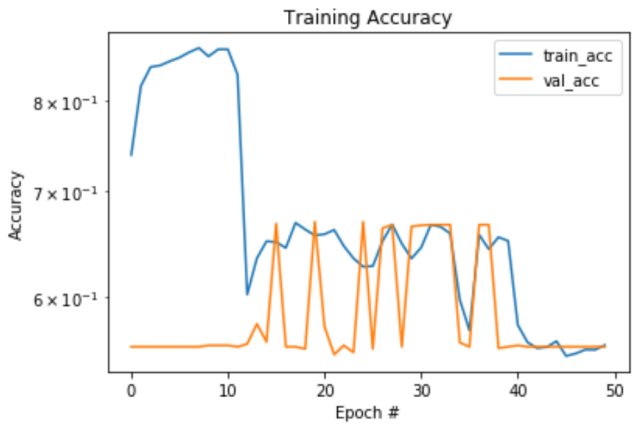


Figure 71: Multiclass Densenet CNN's Training Accuracy

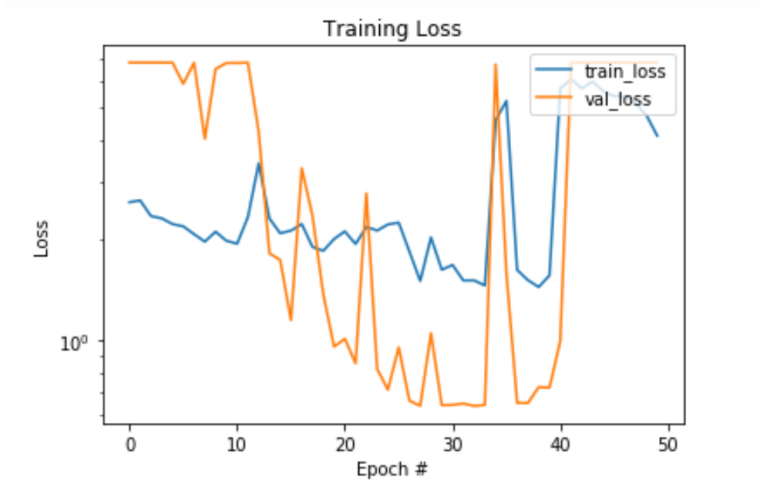


Figure 72: Multiclass Densenet CNN's Training Loss

```
Loss: 6.254164371490479
Accuracy: 0.59333336353302
Confusion Matrix:
 [[ 0  0 33]
 [ 0  0 28]
 [ 0  0 39]]
             precision  recall  f1-score  support   pred
2               0.3900    1.00  0.561151     39.0  100.0
avg / total     0.1521    0.39  0.218849     39.0   39.0
```

Figure 73: Multiclass Densenet CNN's Report

6.3.5 Inception Net Multi-Class Classification:

Inception Net, in its current state, unfortunately, performed the worst, at 0.3276 accuracy and 1.1001 loss. It also has a validation loss of 3.2959, making it the third-worst, and validation accuracy of 0.3379, making it the worst in that category. Inception for multiclass took 33 minutes and 20 seconds, with average times of 39 - 40 seconds. For multi-class, it was unfortunate that the datasets were unable to be evaluated and as a result, the evaluation loss and accuracy were left blank for this model.

Model: "inception_v1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 200, 200, 3)	0	
conv_1_7x7/2 (Conv2D)	(None, 100, 100, 64)	9472	input_1[0][0]
max_pool_1_3x3/2 (MaxPooling2D)	(None, 50, 50, 64)	0	conv_1_7x7/2[0][0]
conv_2a_3x3/1 (Conv2D)	(None, 50, 50, 64)	4160	max_pool_1_3x3/2[0][0]
conv_2b_3x3/1 (Conv2D)	(None, 50, 50, 192)	110784	conv_2a_3x3/1[0][0]
max_pool_2_3x3/2 (MaxPooling2D)	(None, 25, 25, 192)	0	conv_2b_3x3/1[0][0]
conv2d_2 (Conv2D)	(None, 25, 25, 96)	18528	max_pool_2_3x3/2[0][0]
conv2d_4 (Conv2D)	(None, 25, 25, 16)	3088	max_pool_2_3x3/2[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 25, 25, 192)	0	max_pool_2_3x3/2[0][0]
conv2d_1 (Conv2D)	(None, 25, 25, 64)	12352	max_pool_2_3x3/2[0][0]
conv2d_3 (Conv2D)	(None, 25, 25, 128)	110720	conv2d_2[0][0]
conv2d_5 (Conv2D)	(None, 25, 25, 32)	12832	conv2d_4[0][0]
conv2d_6 (Conv2D)	(None, 25, 25, 32)	6176	max_pooling2d_1[0][0]
inception_3a (Concatenate)	(None, 25, 25, 256)	0	conv2d_1[0][0] conv2d_3[0][0] conv2d_5[0][0] conv2d_6[0][0]
conv2d_8 (Conv2D)	(None, 25, 25, 128)	32896	inception_3a[0][0]
conv2d_10 (Conv2D)	(None, 25, 25, 32)	8224	inception_3a[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 25, 25, 256)	0	inception_3a[0][0]
conv2d_7 (Conv2D)	(None, 25, 25, 128)	32896	inception_3a[0][0]
conv2d_9 (Conv2D)	(None, 25, 25, 192)	221376	conv2d_8[0][0]
conv2d_11 (Conv2D)	(None, 25, 25, 96)	76896	conv2d_10[0][0]
conv2d_12 (Conv2D)	(None, 25, 25, 64)	16448	max_pooling2d_2[0][0]
inception_3b (Concatenate)	(None, 25, 25, 480)	0	conv2d_7[0][0] conv2d_9[0][0] conv2d_11[0][0] conv2d_12[0][0]
max_pool_3_3x3/2 (MaxPooling2D)	(None, 13, 13, 480)	0	inception_3b[0][0]
conv2d_14 (Conv2D)	(None, 13, 13, 96)	46176	max_pool_3_3x3/2[0][0]
conv2d_16 (Conv2D)	(None, 13, 13, 16)	7696	max_pool_3_3x3/2[0][0]
max_pooling2d_3 (MaxPooling2D)	(None, 13, 13, 480)	0	max_pool_3_3x3/2[0][0]
conv2d_13 (Conv2D)	(None, 13, 13, 192)	92352	max_pool_3_3x3/2[0][0]
conv2d_15 (Conv2D)	(None, 13, 13, 208)	179920	conv2d_14[0][0]
conv2d_17 (Conv2D)	(None, 13, 13, 48)	19248	conv2d_16[0][0]
conv2d_18 (Conv2D)	(None, 13, 13, 64)	30784	max_pooling2d_3[0][0]
inception_4a (Concatenate)	(None, 13, 13, 512)	0	conv2d_13[0][0] conv2d_15[0][0] conv2d_17[0][0] conv2d_18[0][0]
conv2d_21 (Conv2D)	(None, 13, 13, 112)	57456	inception_4a[0][0]
conv2d_23 (Conv2D)	(None, 13, 13, 24)	12312	inception_4a[0][0]
max_pooling2d_4 (MaxPooling2D)	(None, 13, 13, 512)	0	inception_4a[0][0]
conv2d_20 (Conv2D)	(None, 13, 13, 160)	82080	inception_4a[0][0]
conv2d_22 (Conv2D)	(None, 13, 13, 224)	226016	conv2d_21[0][0]
conv2d_24 (Conv2D)	(None, 13, 13, 64)	38464	conv2d_23[0][0]
conv2d_25 (Conv2D)	(None, 13, 13, 64)	32832	max_pooling2d_4[0][0]
inception_4b (Concatenate)	(None, 13, 13, 512)	0	conv2d_20[0][0] conv2d_22[0][0] conv2d_24[0][0] conv2d_25[0][0]
conv2d_27 (Conv2D)	(None, 13, 13, 128)	65664	inception_4b[0][0]
conv2d_29 (Conv2D)	(None, 13, 13, 24)	12312	inception_4b[0][0]
max_pooling2d_5 (MaxPooling2D)	(None, 13, 13, 512)	0	inception_4b[0][0]
conv2d_26 (Conv2D)	(None, 13, 13, 128)	65664	inception_4b[0][0]
conv2d_28 (Conv2D)	(None, 13, 13, 256)	295168	conv2d_27[0][0]
conv2d_30 (Conv2D)	(None, 13, 13, 64)	38464	conv2d_29[0][0]
conv2d_31 (Conv2D)	(None, 13, 13, 64)	32832	max_pooling2d_5[0][0]
inception_4c (Concatenate)	(None, 13, 13, 512)	0	conv2d_26[0][0] conv2d_28[0][0] conv2d_30[0][0] conv2d_31[0][0]
conv2d_33 (Conv2D)	(None, 13, 13, 144)	73872	inception_4c[0][0]
conv2d_35 (Conv2D)	(None, 13, 13, 32)	16416	inception_4c[0][0]
max_pooling2d_6 (MaxPooling2D)	(None, 13, 13, 512)	0	inception_4c[0][0]
conv2d_32 (Conv2D)	(None, 13, 13, 112)	57456	inception_4c[0][0]
conv2d_34 (Conv2D)	(None, 13, 13, 288)	373536	conv2d_33[0][0]
conv2d_36 (Conv2D)	(None, 13, 13, 64)	51264	conv2d_35[0][0]
conv2d_37 (Conv2D)	(None, 13, 13, 64)	32832	max_pooling2d_6[0][0]
inception_4d (Concatenate)	(None, 13, 13, 528)	0	conv2d_32[0][0] conv2d_34[0][0] conv2d_36[0][0] conv2d_37[0][0]
conv2d_40 (Conv2D)	(None, 13, 13, 160)	84640	inception_4d[0][0]
conv2d_42 (Conv2D)	(None, 13, 13, 32)	16928	inception_4d[0][0]
max_pooling2d_7 (MaxPooling2D)	(None, 13, 13, 528)	0	inception_4d[0][0]
conv2d_39 (Conv2D)	(None, 13, 13, 256)	135424	inception_4d[0][0]
conv2d_41 (Conv2D)	(None, 13, 13, 320)	461120	conv2d_40[0][0]
conv2d_43 (Conv2D)	(None, 13, 13, 128)	102528	conv2d_42[0][0]
conv2d_44 (Conv2D)	(None, 13, 13, 128)	67712	max_pooling2d_7[0][0]
inception_4e (Concatenate)	(None, 13, 13, 832)	0	conv2d_39[0][0] conv2d_41[0][0] conv2d_43[0][0] conv2d_44[0][0]
max_pool_4_3x3/2 (MaxPooling2D)	(None, 7, 7, 832)	0	inception_4e[0][0]
conv2d_46 (Conv2D)	(None, 7, 7, 160)	133280	max_pool_4_3x3/2[0][0]
conv2d_48 (Conv2D)	(None, 7, 7, 32)	26656	max_pool_4_3x3/2[0][0]
max_pooling2d_8 (MaxPooling2D)	(None, 7, 7, 832)	0	max_pool_4_3x3/2[0][0]
conv2d_45 (Conv2D)	(None, 7, 7, 256)	213248	max_pool_4_3x3/2[0][0]
conv2d_47 (Conv2D)	(None, 7, 7, 320)	461120	conv2d_46[0][0]
conv2d_49 (Conv2D)	(None, 7, 7, 128)	102528	conv2d_48[0][0]
conv2d_50 (Conv2D)	(None, 7, 7, 128)	106624	max_pooling2d_8[0][0]
inception_5a (Concatenate)	(None, 7, 7, 832)	0	conv2d_45[0][0] conv2d_47[0][0] conv2d_49[0][0] conv2d_50[0][0]
conv2d_52 (Conv2D)	(None, 7, 7, 192)	159936	inception_5a[0][0]
conv2d_54 (Conv2D)	(None, 7, 7, 48)	39984	inception_5a[0][0]
max_pooling2d_9 (MaxPooling2D)	(None, 7, 7, 832)	0	inception_5a[0][0]
average_pooling2d_1 (AveragePoo	(None, 3, 3, 512)	0	inception_4a[0][0]
average_pooling2d_2 (AveragePoo	(None, 3, 3, 528)	0	inception_4d[0][0]
conv2d_51 (Conv2D)	(None, 7, 7, 384)	319872	inception_5a[0][0]
conv2d_53 (Conv2D)	(None, 7, 7, 384)	663936	conv2d_52[0][0]
conv2d_55 (Conv2D)	(None, 7, 7, 128)	153728	conv2d_54[0][0]
conv2d_56 (Conv2D)	(None, 7, 7, 128)	106624	max_pooling2d_9[0][0]
conv2d_19 (Conv2D)	(None, 3, 3, 128)	65664	average_pooling2d_1[0][0]
conv2d_38 (Conv2D)	(None, 3, 3, 128)	67712	average_pooling2d_2[0][0]
inception_5b (Concatenate)	(None, 7, 7, 1024)	0	conv2d_51[0][0] conv2d_53[0][0] conv2d_55[0][0] conv2d_56[0][0]
flatten_1 (Flatten)	(None, 1152)	0	conv2d_19[0][0]
flatten_2 (Flatten)	(None, 1152)	0	conv2d_38[0][0]
avg_pool_5_3x3/1 (GlobalAverage	(None, 1024)	0	inception_5b[0][0]
dense_1 (Dense)	(None, 1024)	1180672	flatten_1[0][0]
dense_2 (Dense)	(None, 1024)	1180672	flatten_2[0][0]
dropout_3 (Dropout)	(None, 1024)	0	avg_pool_5_3x3/1[0][0]
dropout_1 (Dropout)	(None, 1024)	0	dense_1[0][0]
dropout_2 (Dropout)	(None, 1024)	0	dense_2[0][0]
output (Dense)	(None, 3)	3075	dropout_3[0][0]
auxilliary_output_1 (Dense)	(None, 3)	3075	dropout_1[0][0]
auxilliary_output_2 (Dense)	(None, 3)	3075	dropout_2[0][0]

Total params: 8,477,497
Trainable params: 8,477,497
Non-trainable params: 0

Figure 74: Multiclass Inception CNN's Model

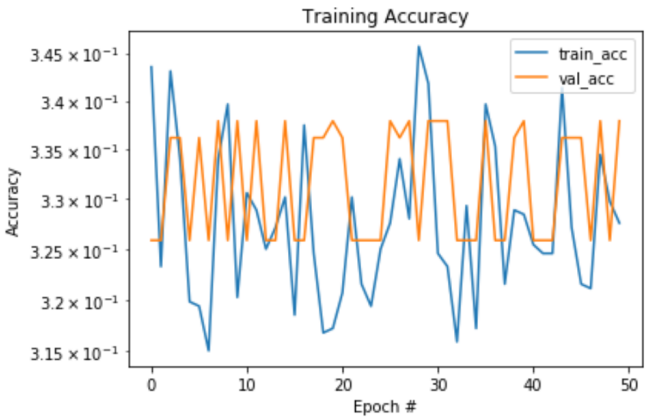


Figure 75: Multiclass Inception CNN's Training Accuracy

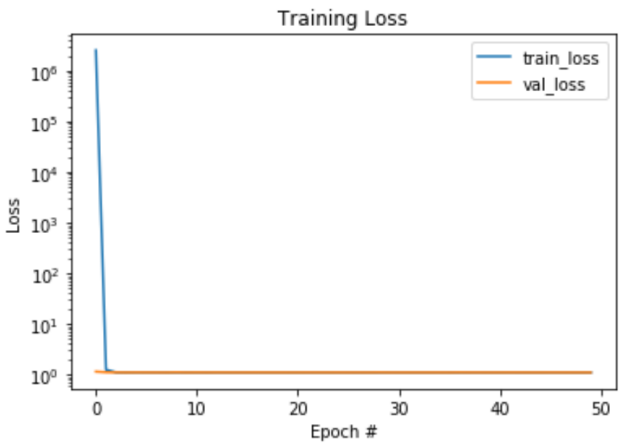


Figure 76: Multiclass Inception CNN's Training Loss

```
Loss: 11.122646484375
Accuracy: 1.0983299016952515
```

Figure 77: Multiclass Inception CNN's Report

6.3.6 Xception Net Multi-Class Classification

Xception Net's training accuracy for multi-class classification increased gradually - sharp increase up until the 3rd epoch, followed by the increase at a decreasing rate. The final accuracy of Xception was 0.9897, one of the models with the highest accuracy amongst all the models. The validation accuracy, on the other hand, was rather unpredictable - though it does have a general increase, it was marred by drops, especially at the end. Both the training loss and the validation loss decreases gradually, though the training loss decreases faster. The training loss decreased exponentially from 1st epoch to the 15th epoch, then the more gradual decrease with some fluctuations. The Xception model took an average of 150 to 165 seconds per epoch, with the overall total time being 2 hours 12 minutes.

Model: "model_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 200, 200, 3)	0	
conv2d_1 (Conv2D)	(None, 100, 100, 8)	224	input_1[0][0]
batch_normalization_1 (BatchNor	(None, 100, 100, 8)	32	conv2d_1[0][0]
dropout_1 (Dropout)	(None, 100, 100, 8)	0	batch_normalization_1[0][0]
conv2d_2 (Conv2D)	(None, 100, 100, 16)	1168	dropout_1[0][0]
batch_normalization_2 (BatchNor	(None, 100, 100, 16)	64	conv2d_2[0][0]
dropout_2 (Dropout)	(None, 100, 100, 16)	0	batch_normalization_2[0][0]
batch_normalization_3 (BatchNor	(None, 100, 100, 16)	64	dropout_2[0][0]
dropout_3 (Dropout)	(None, 100, 100, 16)	0	batch_normalization_3[0][0]
depthwise_conv2d_1 (DepthwiseCo	(None, 100, 100, 16)	160	dropout_2[0][0]
batch_normalization_4 (BatchNor	(None, 100, 100, 16)	64	dropout_3[0][0]
depthwise_conv2d_2 (DepthwiseCo	(None, 100, 100, 16)	160	depthwise_conv2d_1[0][0]
dropout_4 (Dropout)	(None, 100, 100, 16)	0	batch_normalization_4[0][0]
max_pooling2d_1 (MaxPooling2D)	(None, 99, 99, 16)	0	depthwise_conv2d_2[0][0]
conv2d_3 (Conv2D)	(None, 99, 99, 16)	1040	dropout_4[0][0]
concatenate_1 (Concatenate)	(None, 99, 99, 32)	0	max_pooling2d_1[0][0] conv2d_3[0][0]
batch_normalization_5 (BatchNor	(None, 99, 99, 32)	128	concatenate_1[0][0]
dropout_5 (Dropout)	(None, 99, 99, 32)	0	batch_normalization_5[0][0]
activation_1 (Activation)	(None, 99, 99, 32)	0	concatenate_1[0][0]
batch_normalization_6 (BatchNor	(None, 99, 99, 32)	128	dropout_5[0][0]
conv2d_4 (Conv2D)	(None, 99, 99, 64)	18496	activation_1[0][0]
dropout_6 (Dropout)	(None, 99, 99, 32)	0	batch_normalization_6[0][0]
depthwise_conv2d_3 (DepthwiseCo	(None, 99, 99, 64)	640	conv2d_4[0][0]
batch_normalization_7 (BatchNor	(None, 99, 99, 32)	128	dropout_6[0][0]
depthwise_conv2d_4 (DepthwiseCo	(None, 99, 99, 64)	640	depthwise_conv2d_3[0][0]
dropout_7 (Dropout)	(None, 99, 99, 32)	0	batch_normalization_7[0][0]
max_pooling2d_2 (MaxPooling2D)	(None, 98, 98, 64)	0	depthwise_conv2d_4[0][0]
conv2d_5 (Conv2D)	(None, 98, 98, 64)	8256	dropout_7[0][0]
concatenate_2 (Concatenate)	(None, 98, 98, 128)	0	max_pooling2d_2[0][0] conv2d_5[0][0]
activation_2 (Activation)	(None, 98, 98, 128)	0	concatenate_2[0][0]
conv2d_6 (Conv2D)	(None, 98, 98, 64)	73792	activation_2[0][0]
batch_normalization_8 (BatchNor	(None, 98, 98, 64)	256	conv2d_6[0][0]
dropout_8 (Dropout)	(None, 98, 98, 64)	0	batch_normalization_8[0][0]
conv2d_7 (Conv2D)	(None, 98, 98, 32)	18464	dropout_8[0][0]
batch_normalization_9 (BatchNor	(None, 98, 98, 32)	128	conv2d_7[0][0]
dropout_9 (Dropout)	(None, 98, 98, 32)	0	batch_normalization_9[0][0]
flatten_1 (Flatten)	(None, 307328)	0	dropout_9[0][0]
dense_1 (Dense)	(None, 3)	921987	flatten_1[0][0]

Total params: 1,046,019
Trainable params: 1,045,523
Non-trainable params: 496

Figure 78: Multiclass Xception CNN's Model

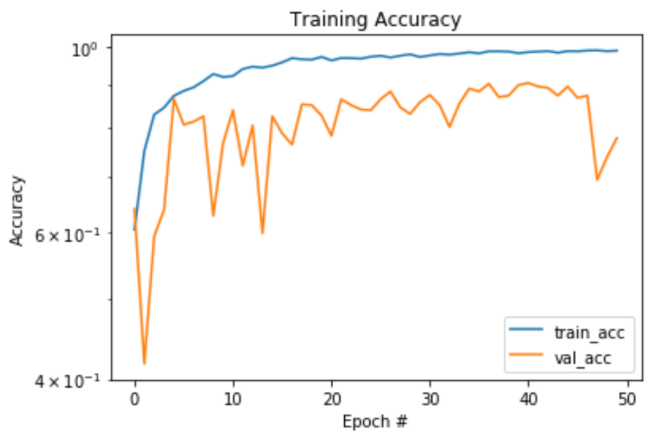


Figure 79: Multiclass Xception CNN's Training Accuracy

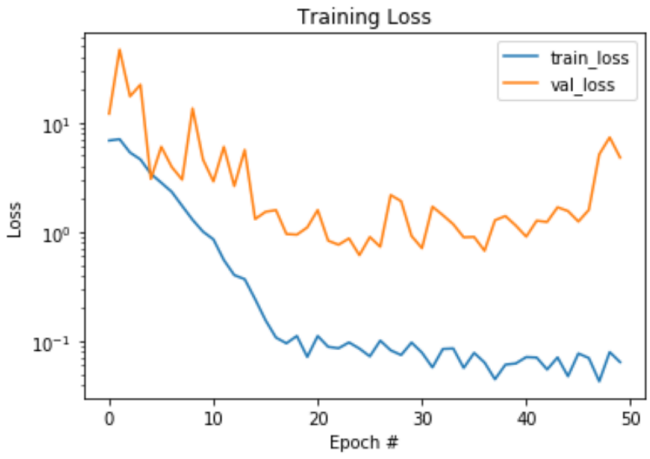


Figure 80: Multiclass Xception CNN's Training Loss

```
Loss: 4.935760517120361
Accuracy: 0.7400000095367432
Confusion Matrix:
 [[19  3 11]
 [ 1 17 10]
 [ 1  0 38]]
             precision    recall  f1-score  support   pred
0             0.904762  0.575758  0.703704     33.0   21.0
1             0.850000  0.607143  0.708333     28.0   20.0
2             0.644068  0.974359  0.775510     39.0   59.0
avg / total   0.787758  0.740000  0.733005    100.0  100.0
```

Figure 81: Multiclass Xception CNN's Report

6.3.7 Custom Model for Multi-Class Classification

Our Custom Model's training accuracy increased gradually, finishing off at 0.9772. The validation accuracy, while it has a general increase, it has fluctuations along the way. The training loss has a sharp decrease at first, followed by an exponential decrease that is more gradual. In the meantime, the validation loss more of less fluctuated around 0.3000. Our Custom CNN Model took an average of 8 to 10 seconds per epoch, with the overall total time being 7 minutes and 41 seconds.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 198, 198, 16)      448
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 99, 99, 16)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 99, 99, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 97, 97, 32)        4640
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 48, 48, 32)        0
_________________________________________________________________
dropout_2 (Dropout)          (None, 48, 48, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 73728)             0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 221187
=================================================================
Total params: 226,275
Trainable params: 226,275
Non-trainable params: 0
```

Figure 82: Multiclass Custom Model CNN's Model

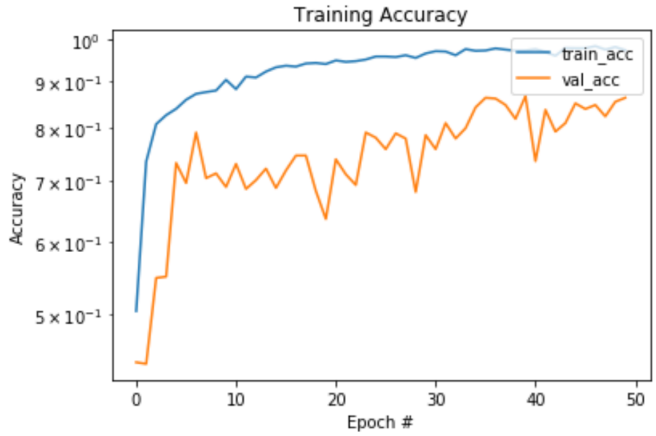


Figure 83: Multiclass Custom Model CNN's Training Accuracy

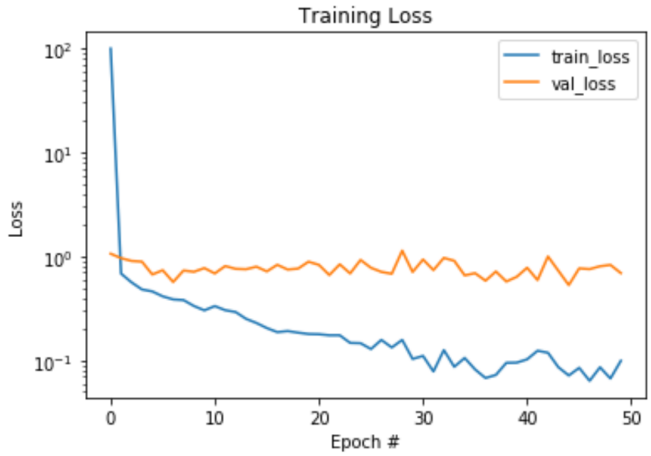


Figure 84: Multiclass Custom Model CNN's Training Loss

```
Loss: 0.295640726685524
Accuracy: 0.8899999856948853
Confusion Matrix:
 [[28  5  0]
 [ 2 25  1]
 [ 2  1 36]]
             precision    recall  f1-score  support   pred
0            0.875000  0.848485  0.861538     33.0   32.0
1            0.806452  0.892857  0.847458     28.0   31.0
2            0.972973  0.923077  0.947368     39.0   37.0
avg / total  0.894016  0.890000  0.891070    100.0  100.0
```

Figure 85: Multiclass Custom Model CNN's Report

7 Conclusions

There are a few takeaways that we got from this project. As we dig deeper into the idea of using the Python libraries to process the sounds, converting into spectrograms, we have also looked through the various models for both binary and multi-class classification. Some of them have managed to work nearly flawlessly, while the other models are not very suitable for this kind of work, at least in their current states.

For Binary Classification, the Basic CNN - Convolution Neural Network, scored the best when it comes to accuracy and loss. For validation loss and accuracy, that would go to the VGG Network. In terms of time taken, Basic CNN would be the fastest. The other models such as Inception and Densenet, would fare the worst when it comes to evaluation accuracy; Densenet worked well when it comes to training, but it does not work as intended when it comes to accuracy. Inception would be entirely undesirable when it comes to accuracy from both training and evaluation.

The same principle applies to Multi-Class classification. ResNet and DenseNet performed the worst when it comes to accuracy for both training and evaluation. As a result, at their current states, ResNet and DenseNet are unsuitable for Multi-Class classification. Basic CNN and VGG Network still performed the best, though the amount of time taken is still a factor.

As a result, we have created our very own Custom Model, which is a mixture of the Basic CNN and VGG Network - to combine the great accuracy of training and evaluation from VGG Network, while also cutting down the time needed to finish the process. For both binary and multi-class, our Custom Model has a performance on par with both Basic CNN and VGG Network, boasting the same amount of evaluation accuracy as the VGG Network, while also taking a shorter amount of time as compared to the VGG Network, though slightly longer than Basic CNN.

8 Future Works

For future, we would like to work on following things:

1. Inception - for both Binary and Multi-Class Classification, could be given more attention and can be further refined. It was ultimately the most difficult model for us to work with.
2. Implement multi-label classification to detect multiple events in a single soundscape.
3. Increase the database of our background and foreground audio files to have a more complete dataset.
4. Replace our current testing data with non-synthetic audio files to test the authenticity of your current training dataset.
5. Record original sounds for our background and foreground audio files for a better real-world use case.
6. Tweak our existing model's hyperparameters to improve prediction results.
7. Test models other than CNN for multi-label classification.
8. Come up with a custom model which best suits our project.
9. Test our custom model on other Sound Event Detection (SED) problems

9 References

[1] *7 Types of Neural Network Activation Functions: How to Choose?* URL: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/. (accessed: 06.12.2019).

[2] Apoorva Agrawal. *Loss Functions and Optimization Algorithms. Demystified.* Sept. 2017. URL: https://medium.com/data-science-group-iitr/loss-functions-and-optimization-algorithms-demystified-bb92daff331c. (accessed: 06.12.2019).

[3] Sai Kumar Basaveswara. *CNN Architectures, a Deep-dive.* Aug. 2019. URL: https://towardsdatascience.com/cnn-architectures-a-deep-dive-a99441d18049. (accessed: 06.12.2019).

[4] Anne Bonner. *The Complete Beginner's Guide to Deep Learning: Convolutional Neural Networks and Image Classification.* Feb. 2019. URL: https://towardsdatascience.com/wtf-is-image-classification-8e78a8235acb/. (accessed: 06.12.2019).

[5] Jason Brownlee. *Dropout Regularization in Deep Learning Models With Keras.* June 2016. URL: https://machinelearningmastery.com/dropout-regularization-deep-learning-models-keras/. (accessed: 06.12.2019).

[6] Caleb Cheng. *Deep learning: Saving rainforests with TensorFlow.* Oct. 2019. URL: https://towardsdatascience.com/deep-learning-saving-rainforests-with-tensorflow-4e906344811a. (accessed: 04.12.2019).

[7] *Convolutional Neural Networks: Architectures, Convolution / Pooling Layers.* URL: http://cs231n.github.io/convolutional-networks/. (accessed: 06.12.2019).

[8] Daphne Cornelisse. *An intuitive guide to Convolutional Neural Networks.* Apr. 2018. URL: https://www.freecodecamp.org/news/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050/. (accessed: 06.12.2019).

[9] F D. *Batch normalization in Neural Networks.* Oct. 2017. URL: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c. (accessed: 06.12.2019).

[10] Rachel Lea Ballantyne Draelos. *Multi-label vs. Multi-class Classification: Sigmoid vs. Softmax.* May 2019. URL: https://glassboxmedicine.com/2019/05/26/classification-sigmoid-vs-softmax/. (accessed: 04.12.2019).

[11] Cat Ferguson. *Old smartphones called in to save Indonesian forests.* June 2013. URL: https://www.newscientist.com/article/mg21829205-600-old-smartphones-called-in-to-save-indonesian-forests/. (accessed: 06.12.2019).

[12] Raúl Gómez. *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names.* May 2018. URL: https://gombru.github.io/2018/05/23/cross_entropy_loss/. (accessed: 06.12.2019).

[13] jeongmin.cha. *Machine Learning: Should I use a categorical cross entropy or binary cross entropy loss for binary predictions?* May 2019. URL: https://stats.stackexchange.com/questions/260505/machine-learning-should-i-use-a-categorical-cross-entropy-or-binary-cross-entro. (accessed: 06.12.2019).

[14] David Rolnick. *Tackling Climate Change with Machine Learning.* Nov. 2019. eprint: https://arxiv.org/pdf/1906.05433.pdf. URL: https://arxiv.org/abs/1906.05433. (accessed: 04.12.2019).

[15] J. Salamon. *Scaper: A library for soundscape synthesis and augmentation.* eprint: http://www.justinsalamon.com/uploads/4/3/9/4/4394963/salamon_scaper_waspaa_2017.pdf. URL: https://scaper.readthedocs.io/en/latest/index.html. (accessed: 06.12.2019).

[16] Anish Singh Walia. *Types of Optimization Algorithms used in Neural Networks and Ways to Optimize Gradient Descent.* June 2017. URL: https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f. (accessed: 06.12.2019).