1. Introduction

Image classification is an integral part of the digital image analysis process which functions to categorize all the pixels of an image into the respective classes. It is a subset of image recognition and has various industry uses such as facial recognition, object identification, emotion detection, gaming and augmented reality (1Regina 2019).

This study is done to perform object identification in a **multi-label image classification** problem. The study is performed over a dataset containing 30,000 training examples and 10,000 test examples (Kaggle 2020, *'Multi-label classification'*). The models developed for this project make use of text descriptions along with the image data to assign labels to the images. We used transfer learning for implementing the models with which we were able to make use of the learned parameters from other pre-trained models in our modelling process.

The proposed architecture of combining the features from the pre-trained BERT model (sentence embeddings) and the features from pre-trained DenseNet161 with data augmentation gave a personal best kaggle score of 0.79820 and outperformed the baseline ResNet50 model. This shows how features from two state-of-the-art models can be used to accomplish a difficult multi-label image classification task.

1.1. Aim and Importance of the study

The aim of this study is to build a model such that it provides a high performance score (f1-score) over the given multi-label image classification problem. We implement different models for this aim with the objective of achieving as **high prediction performance** score as possible while keeping the **model size** to be less than 100 MB and **feasible training time** (such as under 24 hours). This study is important as it plays a significant role in highlighting the differences between the performance of the various models and thus encourages us to achieve the minimum loss conditions.

1.2. Motivation for the used method

Since their inception, Convolution Neural Networks (CNNs) have demonstrated a promising performance in their processing. These models compute significantly faster as compared to the SVM models which were the previous leaders of image processing. The CNN models also display a property of 'location invariance' and 'compositionality', therefore considering such advantages of the model, this study uses a CNN model (DenseNet-161) as its baseline model.

For the selection of the method to be used, we followed the models from the Imagenet Large Scale Visual Recognition Challenge (ILSVRC) challenges. ResNet, in 2015, was a revolutionary

winner, but still had some flaws which were further mitigated by DenseNet. This study therefore presents the **DenseNet-161** model (an advanced CNN model) **combined with BERT** (state-of-the-art for text embedding) to predict the labels for the given examples. We combined the two models to leverage the text description of the images to classify them into the given labels.

2. Related Works

Advent of CNNs: In the last decade while convolution neural networks (CNNs) were still being studied and improved, SVM used to be a leader in the prediction of the image classification problems such as the Modified National Institute of Standards and Technology (MNIST) database and the ILSVRC (Xu 2020). However, with the advent of the improved CNNs in ILSVRC 2012, a huge leap in classification performance was observed. The CNN model, AlexNet, was developed in 2012 and achieved the top 5 error of 15.4% over ILSVRC which was remarkably lower than the previous winner SVM in 2011 which had an error of 26.1%. In the paper of AlexNet, the authors highlight the importance of depth for the model to be performing well and also advocate the use of better computation powers to gain even better results (Krizhevsky, Sutskever & Hinton 2012). The model used data augmentation for pre-processing the data, ReLU activation function, dropout with a drop rate of 0.5, Local Response Normalization and Overlapping Pooling where pooling stride is less than the pooling kernel size. The input was divided into two sets to compute on two different GPUs.

Revolutionary CNN architectures: Later, in 2013, an improved model ZFNet was proposed which displayed a top-5 error of 11.2%. It reduced the filter size and stride and also increased the number of filters in the model's inner layers along with the use of data augmentation and dropout (Xu 2020). The ablation studies performed over the model also demonstrated the effectiveness of the depth over the performance of the model. The model made architectural changes to the AlexNet model by using 7x7 filters in layer 1 and stride 2 convolutions in layers 1 and 2 (Zeiler & Fergus 2013). However, GoogLeNet stepped further in 2014 with an even better performance of 6.7% on the ILSVRC. It used 1x1 convolution and replaced the fully connected layers at the end of the model with the global average pooling to yield a better performance. The model also employed the use of Inception modules to have stacked output through a number of convolutions of different shapes for a particular input. With the use of numerous Inception modules, the model employed a total of 22 layers in its architecture (Szegedy et al. 2015). VGGNet, which was also incepted in the same year was the runner-up in classification for ILSVRC-2014, but was a winner for the Localization task (Tsang 2018). VGGNet employed 16-19 layers, 3x3 convolution filters with 1 stride, 1 pad, and 2x2 max-pooling with stride 2 (Simonyan & Zisserman 2014). The simple and homogenous as presented by VGGNet is a basis for most of the networks built nowadays (Khan et al. 2019). ResNet further halved the error on the ILSVRC challenge in the

year 2015 while achieving an error of 3.57% which for the first-time exceeded the human performance which had an error of 5.1%. ResNet observed the under-fitting problem in deeper networks, and thus proposed the idea of skip-connections which provided some regularization to the model's architecture, and thereafter deployed 152 layers to perform the calculation (He et al. 2016).

The table below demonstrates the ILSVRC winners from 2012 to 2015 through the models which revolutionized the field.

Table 1: Winning CNN architectures (2012-2015)

Model	AlexNet	ZF Net	GoogLeNet	Resnet
Year	2012	2013	2014	2015
#Layer	8	8	22	152
Top 5 Acc	15.4%	11.2%	6.7%	3.57%
Data augmentation	√	√	√	√
Dropout	√	√		
Batch normalization				√

Source: (Xu 2020)

Advanced variations of CNNs: ResNeXt was a version of ResNet proposed by Xie et al. (2017) which modified the architecture of ResNet with the split-transform-merge paradigm, while keeping the topology of each path the same. The number of paths was a hyper-parameter, called cardinality which further regulated the performance of the model. The further development of Inception layers which was proposed in GoogleNet also led to further increase in the performance of the models (Xu 2020). ResNet posed a limitation of preserving information due to which subsequent layers would contribute to little or no information. **DenseNet** (Huang et al. 2017) resolved this problem by using the previous layers' feature maps as inputs to the subsequent layers. Due to the concatenation of the features instead of addition, DenseNet thus becomes able to better differentiate between the information fed to it (Khan et al. 2019). Therefore, it helps in reducing the number of parameters required for training and also alleviates the vanishing gradient problem since each layer now has access to the gradients of the cost function (Xu 2020).

BERT (Bidirectional Encoder Representations from Transformers): It is a revolutionary model that caused a stir in the Machine Learning community in recent years by giving state-of-the-art results in a wide variety of NLP problems. It pretrains the bidirectional representations from unlabeled text using the bidirectional training from Transformer, an attention model. It jointly conditions on the text from both left and right directions. This allows

to fine-tune the pre-trained BERT model significantly with the addition of just one output layer (Devlin et al. 2019).

A Transformer model uses the complete sequence of words at once as input, therefore it is considered bidirectional and is able to learn the contextual information of the surroundings. BERT uses Masked tokens for its predictions which function as follows:

- The model aims to predict 15% of the randomly picked tokens.
- From these randomly picked tokens, 80% are replaced with a [MASK] token, 10% with a random word and 10% with the original word.

This allows the model to produce good representations for the non-masked words as well, and would also avoid the cases of the model just trivially copying the non-contextual embedding (Horev 2018).

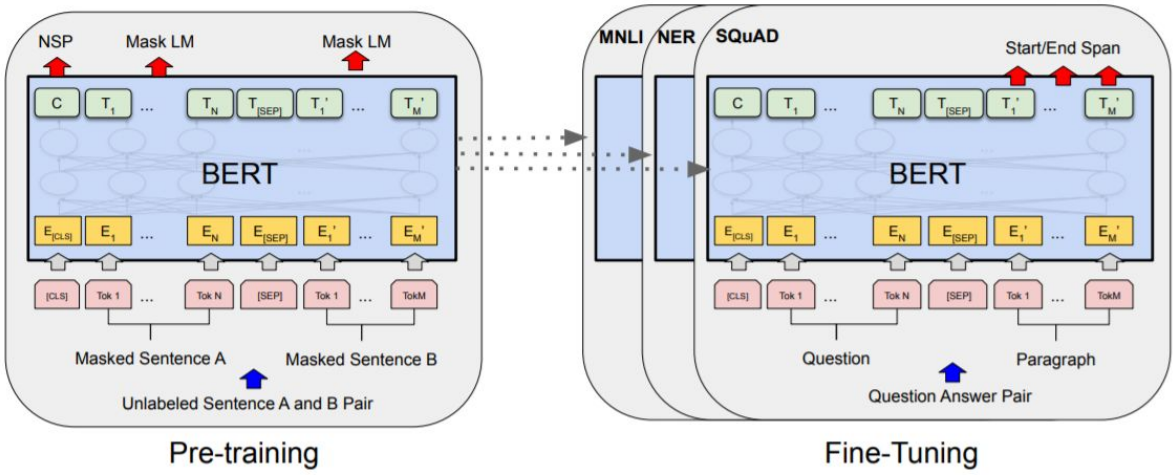


Figure 1: Overall pre-training and fine-tuning procedures for BERT
(Source: Devlin et al. 2019)

3. Techniques

3.1. MLC Pipeline

The multi-label classification pipeline implemented looks as in the diagram below. Further explanation of each of the different modules, their underlying principles, reasons of implementing them for our problem and hyperparameter settings used will be discussed in the later sections.

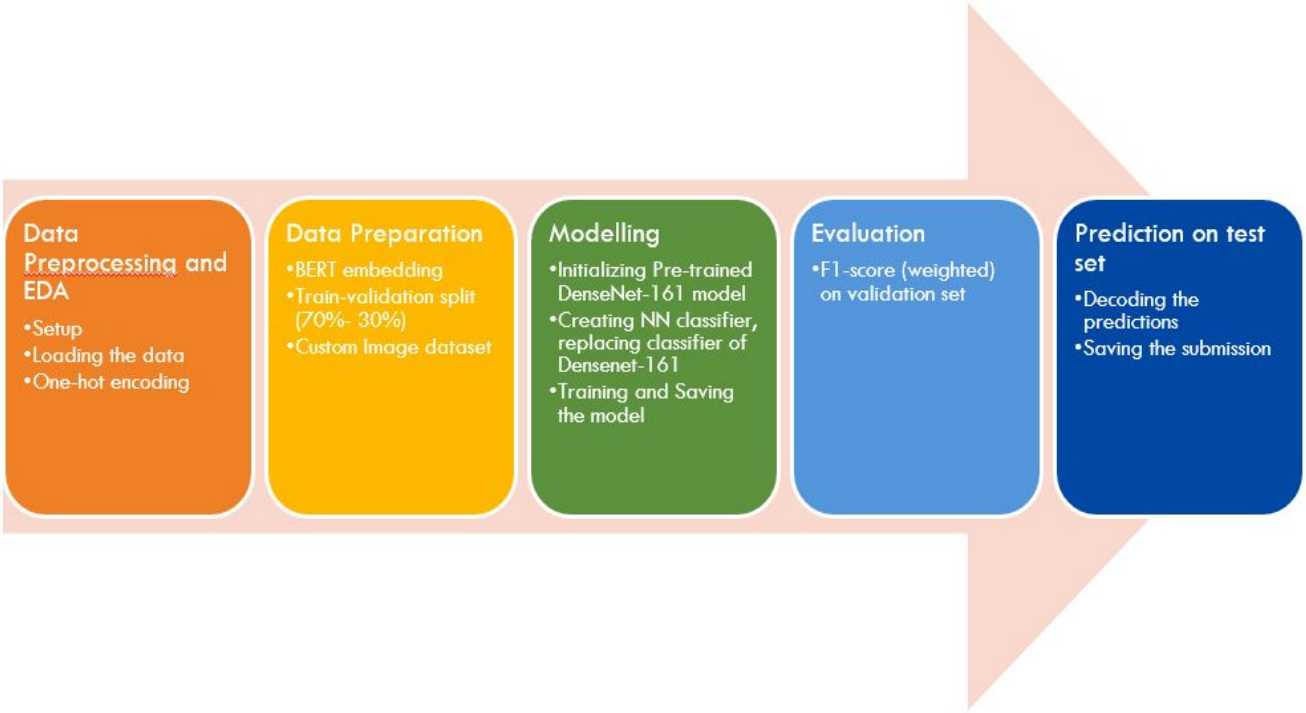


Figure 2: MLC Pipeline

3.2. Data pre-processing

3.2.1. Loading the data

As preliminary data analysis, we analyzed the training dataset. Upon analysis, it was discovered that after removing the unwanted characters from the captions part of data, we are left with 29996 examples out of 30,000. Hence, the shape of the training data is (29996,3). The 'train.csv' file contains the 'ImageId' column having the filenames of the images and the 'labels' column present as a single string containing labels in a space-separated fashion.

3.2.2. One Hot Encoding and Decoding

The labels present in given train_data are in the form of space separated strings. These are split into lists and then later one-hot encoded to convert them to format acceptable by the model. The process involves generating encoder and decoder dictionaries which map the labels to their index positions and vice-versa. The encoder dictionary is used to one-hot encode the labels such that each string is a numpy array/list before the training of the models. The decoder dictionary is leveraged in order to map the predicted outputs to the required original format done after performing the predictions.

3.2.3. Exploratory Data Analysis

During EDA, the distribution of data for each label was checked. The graph below indicates that the data is highly skewed, hence is an imbalance dataset. To overcome this issue experiments with techniques like over-sampling using data augmentation, weighted loss computation etc. were performed. Other important observations during the analysis showed that all the images are not of the same size. Hence, data augmentation needs to be done in order to handle such issues as well.

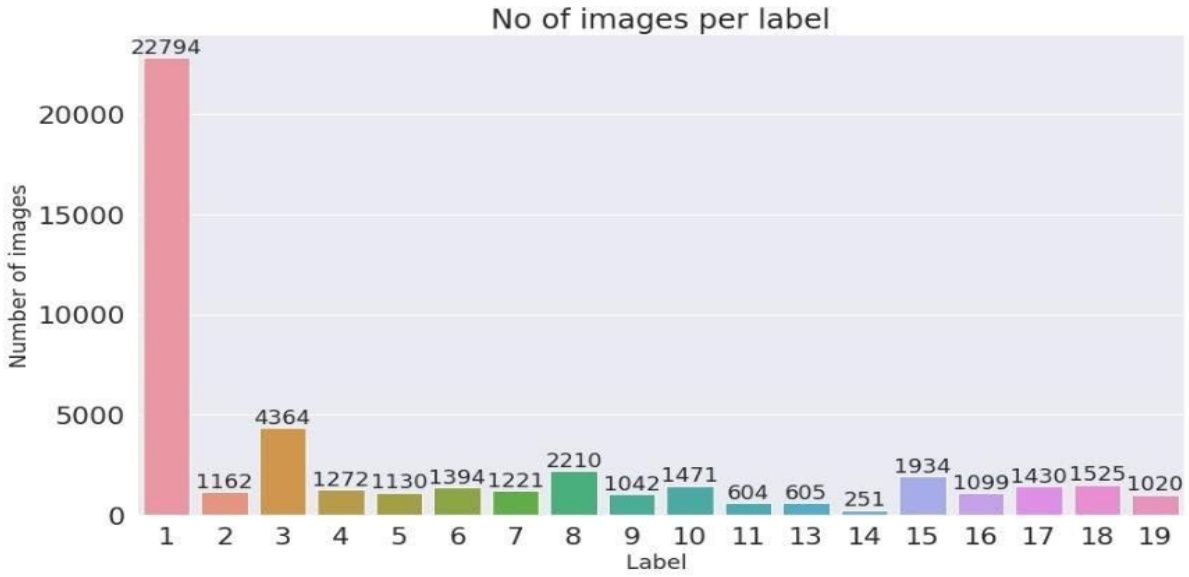


Figure 3: Distribution of data among different classes

3.2.4. Building Custom Dataset

Memory efficiency is a priority when dealing with very large datasets like the one we have. It consists of 40,000 images as training data and testing data, which usually consumes a lot of memory. So, it would not be a wise decision to load them all without requirement. This issue is resolved by building a custom dataset. This technique is memory efficient because all the images are not stored in the memory at once but read as required. For our dataset, we created a custom dataset 'ImageData' that uses the ImageID from the 'train.csv' to get the actual image from the designated folder, transforms it, converts it to tensor and binds it with its corresponding label. This custom dataset inherits its features from the torch.utils.data.Dataset class, overloads its function to get outputs. Our custom dataset also has a 'test' mode which is set to 'False' by default. If 'true' it means a dataset passed as an argument is a test dataset otherwise, train or validation.

3.2.5. Data Augmentation

For some experiments we have used very minimal to some sophisticated Data Augmentation techniques due to imbalance issues and the different image sizes. This is important as the images should be passed as tensors of the same dimensions and be normalized before being passed as an input to the pre-trained models.For our experiments we use the 'transforms' library of PyTorch. For our implementation, same data transformations have been applied to the train and validation datasets whereas a bit different in the test datasets. For example, *RandomResizedCrop(), RandomRotation() etc.* For normalization we use the standard mean and deviation values:

$$\text{mean} = [0.485, 0.456, 0.406], \text{std} = [0.229, 0.224, 0.225]$$

The experiment results related to Data Augmentation will be discussed in the next section.

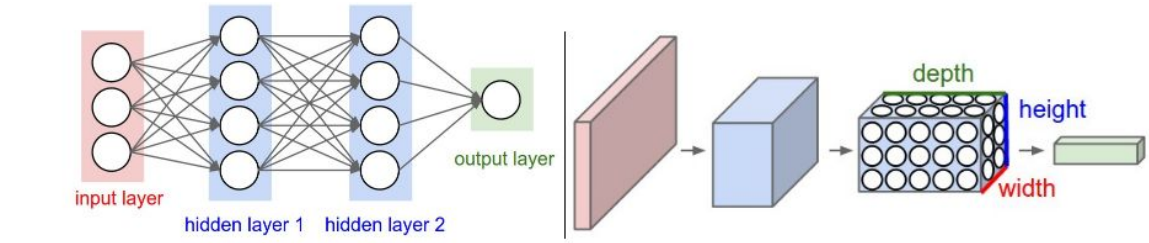
3.3. Modelling

Based on the given dataset, we decided to conduct multiple experiments to get the best classifier with the given training time and memory consumption constraints. The baseline model was decided to be ResNet50 with a neural net classifier with minimal data augmentation. We used the pre-trained ResNet-50 model (Resnet50 n.d.) which is 50 layers deep and is pre-trained over more than a million images. To outperform the baseline, multiple experiments were conducted by changing the decided model to DenseNet-161 and training a Neural Network classifier by using the weights of its last layer as features to train the classifier created. This experiment was too done with/without sophisticated augmentation. However, the advanced architecture proposed in this study involves the use of images and the caption data together to predict the outputs. For this, we concatenated BERT embedding generated from pre-trained BERT Model and the features obtained from the last layer of the DenseNet-161 model and then used as input to the Neural Network Classifier.

The architectures proposed above are CNN architectures. To understand their working well, the underlying principles of CNN are discussed below.

3.3.1. Principle of CNN

CNN model is essentially a neural network consisting of learnable weights and biases and the model still presents a differentiable score function while computing class scores from the raw image pixels. It also calculates the loss function in the last layer which is fully-connected (*CS231n Convolution Neural Networks for Visual Recognition* n.d.).



Left: A regular 3-layer Neural Network. Right: A ConvNet arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a ConvNet transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels).

Figure 4: Neural network (left) vs CNN (right)

(Source: *CS231n Convolution Neural Networks for Visual Recognition* n.d.)

A CNN is primarily built on **three main layers**: Convolution Layer, Pooling Layer and Fully-Connected Layer.

1. **Convolution Layer** - It contains a set of learnable filters which extend across the depth of the input. The filter slides through the input while computing the dot products between the input position and weights of the filter, thus creating a 2-D map. It is therefore able to learn the variations in color such as outlines, edges, color differences in its first layer, and eventually learning the pattern structures in its higher layers. These activation maps are stacked along the depth dimension to produce the output of the Convolution layer. The following parameters are used to control the output from the convolution layer:
   a. **Filter:** A filter is a small spatial learnable parameter that extends through the complete depth of the input volume. The filter slides through the complete input volume to produce a two-dimensional activation map by taking the dot product of weights with the input.
   b. **Stride:** It is used to decide the number of pixels with which the filter slides over the input volume to generate the outputs to the convolution layer.
   c. **Zero-padding:** It is used to pad the input image with zeroes around the border so as to be able to control the size of the output volume.
2. **Pooling layer** - Pooling layer is to reduce the spatial size of the layers and thus to control the number of parameters involved in the network. This therefore helps to reduce the overfitting. Pooling also makes the output representation invariant to small translations of the input image. There are a lot of variations in the pooling operations used. Following are three general types of pooling used in a CNN:
   a. **Max Pooling:** It creates a subsample map by taking the maximum value from the subsample of feature map.
   b. **Average Pooling:** It creates a subsample map by taking the average of all values of the subsample of feature map.

c. L_2 **norm pooling:** It uses a Gaussian window containing weights w_j to generate outputs y_i. The pooling operation applied here is: $y_i = \sqrt{\sum_j w_j x_{i,j}^2}$

3. **Fully connected** - A fully connected layer at the end of CNN is often used to optimize the objective function. It works similar to the neural networks where each neuron is connected to all the previous activations.

These layers are stacked over each other to form the resultant CNN architecture. The input layer to a CNN architecture consists of the raw pixel values of the image. The convolution layer thereafter contains neurons connected to the local regions of the input, where each neuron computes the dot product between the connected region and the weights and an activation layer is thereafter applied to it. The pooling layer then down-samples the data along its spatial dimensions. It thereafter passes through a fully-connected layer where each neuron is connected throughout the previous layer and thereafter computes a single vector of the class scores.

Thus, using DenseNet for the study leverages the **advantage** of CNN models. CNNs avoid training the large number of neurons as in case of fully connected neural network models and thus, it greatly reduces the computation time for the model. Hence, we used **DenseNet-161 (with BERT)** for our study. The further advantages presented by the models used in this study are in the following sections.

3.3.2. DenseNet-161

Densenet-161 is a variation of ResNet which itself is an advanced CNN architecture model. The detailed architecture diagram of DenseNet-161 is summarized in the image below as per the official PyTorch documentation. As each layer of the model has direct access to the gradients of loss function and to the original input signal, this models has the following additional advantages over its counterparts such as:

1. Alleviates the vanishing gradient problem.
2. Strengthens the feature propagation too.
3. It encourages feature reuse thereby reducing the number of parameters used as justified by its performance on standard image datasets in comparison to ResNet models (shown in the figure below).

Layers	Output Size	DenseNet-121($k = 32$)	DenseNet-169($k = 32$)	DenseNet-201($k = 32$)	DenseNet-161($k = 48$)
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6	[1 × 1 conv; 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12	[1 × 1 conv; 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv; 3 × 3 conv] × 24	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 48	[1 × 1 conv; 3 × 3 conv] × 36
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv; 3 × 3 conv] × 16	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 32	[1 × 1 conv; 3 × 3 conv] × 24
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Figure 5: DenseNet architecture

(Source: Huang et al. 2016)

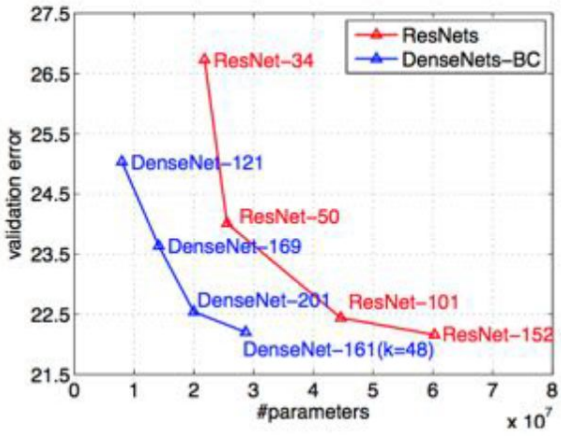


Figure 6: Comparison of DenseNet and ResNet on model size

(Source: Huang et al. 2016)

The ImageData after being loaded into the respective dataloaders, are used to get the features from the pre-trained DenseNet-161 model. This model in the forward function is passed to the densenet model in eval() mode to get the weights from its 3 last layers. All the layers of the model are frozen, except for the last 3 layers . The reason for this could be attributed to the pooling layers because they capture rick features from the convolutional layers and we need to provide them a good classifier to classify easily and effectively reduce the number of linear layers required. The output features are tensors of size 2208. These can be simply used to train the Classifier, or adding a twist to it, can be combined with sentence embeddings to train the classifier.

3.3.3. BERT

It is used for pre-training the text inputs. We used pre-training with BERT to obtain the embedding to be fed to the DenseNet model. This allows us to leverage the text descriptions of the images to better classify the images into their labels.

Being a state-of-the-art model, BERT was helpful for us to leverage efficient embedding from the text descriptions of the images. We used its API which also helped us to keep the final model size low. It provided us a faster computation too by reducing the number of epochs required for parameter-tuning.

The idea was to get the sentence embedding for each image caption for both the train and test datasets. For this, we simply used the pre-trained BERT model 'bert-base-uncased'. The model is then run in the *eval()* mode to get the embedding. The *get_bert_embeddings()* pre-processes the data, gets the word embedding for each caption from the second to the last hidden layers and averages them to get one sentence embedding. As the dataset is quite large, to avoid memory issues, we are splitting the dataset to get embedding batch-wise, then combining them to make a single tensor.

To preprocess the data we used the *BertTokenizer()* to tokenize the sentences after marking the beginning and end of each sentence using [CLS] and [SOS] tokens. After that, words are mapped to their word IDs such that each list was equal to the size of the sentence with maximum length. For our dataset, it was 59. Then an array of segment ids is generated for the sentence IDs. Both these arrays are passed as tensors to the device and sentence embedding are generated as explained above. Hence, for the training dataset, the output is a tensor 'train_emb' of the shape (29996,768) where the shape of 'test_emb' is (10000,768). Hence, we have a tensor of size 768 corresponding to each example in our dataset.

3.3.4. Multi-label-classifier

To replace the default classifier layer of the pre-trained DenseNet161 model, we implement our own classifier. Here we try to leverage the concept of Transfer Learning. The configurations decided for the Neural Network Classifier are as follows:

No. of input features : This parameter is decided based on the type of data to be used for training. For example:

- *Case-1:* n_features = 2208 ; when only image data is used for training
- *Case-2:* n_features = 768; when only caption (text) data is used for training
- *Case-3* n_features = 2976; when both image and caption (text) data is used for training

No. of hidden layers : 3 hidden layers used with 'ReLU' activation function.

Output Layer: No. of neurons equal to the no. of labels with Sigmoid Activation function. The reason to use sigmoid activation as this is a multi-label image classification problem. As we can have more than one class for a particular example, hence, probabilities cannot be considered mutually exclusive.

Loss Function: To compute the function, we use the popular choices, the BCELoss function (does not implicitly include sigmoid) and the BCEWithLogitsLoss function(implicitly includes sigmoid). The reason for not using Cross Entropy Loss Function is the same as these events are not mutually exclusive and hence fails the first condition of using the Naive Bayes based Cross Entropy Loss Function.

Optimizer: Adam Optimizer was chosen as the Optimizer for our experiments due its ability to adapt the learning rates and learn faster.

The no. of epochs, batch sizes and learning rates were few hyperparameters we experimented with. Another setting we changed was the data augmentation techniques used and the different combinations of models and the input features.

3.4. Evaluation metrics used

For the given multi-label problem, the goal was to assign high scores to the ground truth labels of the corresponding inputs. We reviewed *average_precision_score*, *f1_samples* and *f1_weighted* for the evaluation purposes. **f1_weighted** was thereafter used for our study.

Average precision score (AP) computes a weighted mean of precisions at each threshold while using the increase in recall from the previous threshold as weight (Scikit Learn n.d.).

$$AP = \sum_{n}(R_n - R_{n-1})P_n$$

where R_n and P_n are Recall and Precision at n^{th} threshold.

F1-score is another formula that is used to evaluate the problems with imbalanced classes. It could be considered as being equal to the weighted average of precision and recall (Scikit Learn n.d.).

$$f1 = 2 * \frac{(P*R)}{(P+R)}$$

where P and R stand for Precision and Recall respectively.

For a multi-label problem, f1 score uses 'weighted' and 'samples' as its *average* parameters. *f1_sample* calculates the metrics for each instance and finds the average, while the *f1_weighted* calculates the metrics for each label and finds their weighted average by support (Scikit Learn n.d.).

For this study, the competition track uses f1 scores as evaluation metrics (Kaggle 2020, 'Multi-label Classification'). Hence, we used the **f1_weighted**, since it is used to calculate the metrics for each of the labels, thus making it to be more intuitive.

4. Experiments and results

4.1. Baseline Model

The baseline model for our experiment was the pre-trained DenseNet-161.

4.2. Hyper-parameter tuning

The following results were obtained from the different models with the different variations of hyper-parameters during the hyper-parameter tuning process:

Table 2: Hyper-parameter tuning

Experiments			Hyper-parameters				
Models	Dataset	F1 Score	Epochs	Learning Rate	Batch size	Loss Function	Optimizer
ResNet-50	Valid.	0.65271	10	0.001	50	BCELoss	Adam
	Test	0.77723	10	0.001	50	BCELoss	Adam
ResNet-50 +Data Augmentation	Valid.	0.74131	10	0.05	50	BCELoss	Adam
	Test	0.75832	10	0.05	50	BCELoss	Adam
DenseNet - 161	Valid.	0.70617	20	0.001	30	BCEWithLogitsLoss	Adam
	Test	0.67315	20	0.001	30	BCEWithLogitsLoss	Adam
DenseNet - 161+BERT embedding	Valid.	0.68829	20	0.001	50	BCEWithLogitsLoss	Adam
	Test	0.79820	20	0.001	50	BCEWithLogitsLoss	Adam

Transfer learning from ResNet-50 and DenseNet-161 are implemented with a trainable fully connected neural network attached to its end. From the experiment, it is evident that the DenseNet-161 with BERT embedding gives the highest F1-score on the test set, therefore is

regarded to be the top performing model. Data augmentation is also experimented. Since it is found that the classes of the dataset is unbalanced. However, it does not give the highest score. It is suspected that the data augmentation overfits the model.

5. Conclusion and Discussion

The major challenge faced while working on this dataset was the nature of the dataset itself. The dataset is very large and imbalanced. Hence, apart from dealing with its imbalanced nature, training the dataset in itself was a big challenge due to limitation of the systems available with the team. Very major modifications requiring a lot of time had to be done to the code implementations in order to strike a balance between the memory efficiency, time of training and model's performance. Though the model could only achieve a decent personal best score of **0.79,** but can be run well within feasible time, on a system having GPU with less memory or even on Google Colab without any 'Out of Memory' issues. Though the novel architecture proposed hereby performed decently, they could be further explored by experimenting with other advanced CNN architectures like ResNeXt, GoogLeNet etc. which are known to perform really well on such datasets but were rejected due to their size and time constraint for this project.