THE UNIVERSITY OF SYDNEY

*The University of Sydney*

*School of Computer Science*

# Assignment 2 Report


*Author:*
Enze Ren 520638064
Taoxu Zhao 520549357
Yunhui Song 510611602

*Supervisor:*
Chang Xu


An Assignment submitted for the UoS:

*COMP5329 Deep Learning*

May 18, 2023

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Aim and Objective

In recent years, the application of deep neural networks in large models has witnessed significant advancements, leading to their widespread deployment in various fields to form application products [LeCun et al., 2015]. However, applications in critical domains, such as autonomous driving [Bojarski et al., 2016] and medical health monitoring [Esteva et al., 2019], necessitate high temporal and spatial efficiency for effective deployment. Many large model products that are distributed or even integrated into cloud computing face challenges in meeting the requirements of these target fields. In this context, lightweight models often prove to be more capable of addressing the needs of target users. Despite the prominence of various machine learning algorithms, deep learning remains at the forefront of delivering high accuracy [Schmidhuber, 2015]. Consequently, exploring and developing lightweight deep neural network models for adaptive scenarios and investigating further optimization is a necessary research endeavor in many specific contexts.

In addition to the implementation of lightweight models, this project also emphasises the integrated deployment of convolutional neural networks and nature language processing models.

## 1.2 Motivation

Our motivation stems from the need to create efficient and effective lightweight deep neural network models that can cater to the requirements of various adaptive scenarios, particularly those with stringent temporal and spatial constraints. By focusing on the development of such models, we aim to contribute to fields that demand high-performance solutions without compromising on efficiency. This includes applications in real-time object detection for autonomous vehicles, as well as instant analysis of medical data for health monitoring systems [Howard et al., 2017].

## 1.3 Importance

In this study, we focus on a dataset comprising 30,000 data entities with the objective of investigating and developing lightweight deep neural network models that satisfy specific constraints (i.e., size < 100MB). Our research builds upon well-established lightweight deep neural network models, including EfficientNet [Tan and Le, 2019] and BERT [Devlin et al., 2018], by examining their adaptive counterparts, which are characterized by parameter adjustments tailored to this particular context. Simultaneously, we strive to achieve the highest possible performance F1 scores and, subsequently, propose potential optimization strategies.

To this end, we systematically analyze the trade-offs between model size, computational complexity, and performance, providing valuable insights into the design of efficient lightweight deep neural network models. Furthermore, our research addresses the challenge of finding the optimal balance between accuracy and efficiency, which is crucial for practical applications in resource-constrained environments. By offering optimized models for various adaptive scenarios, we aspire to facilitate advancements in fields that rely on real-time processing and swift decision-making.

# Chapter 2

# Related works

## 2.1 CV Models

Since the introduction of AlexNet [Krizhevsky et al., 2017], which sparked the surge of deep neural network computer vision models, numerous attempts have been made to optimize the scale of model parameters. SqueezeNet [Iandola et al., 2016] was the first to achieve this by using 1x1 small convolution kernels and adopting the fire module for modular convolution. Consequently, the model size was reduced to 1/50 of AlexNet while maintaining a similar level of accuracy. CondenseNet [Huang et al., 2018] enhanced feature reuse and reduced the number of parameters in the neural network by employing dense connectivity between layers and a "learned group convolution." ShiftNet [Yan et al., 2018] utilized shift convolution for video input data to enhance the position invariance of the network, while further reducing computational costs through "second convolution."MobileNet [Howard et al., 2017] adopted depthwise separable convolution to divide traditional convolution into two separate modules: depthwise convolution and pointwise convolution, thereby decreasing computational costs. MobileNet V2 [Sandler et al., 2018] built on this foundation by incorporating a "linear bottleneck" module, which further improved the model performance through the addition of linear units and inverted residual structures. V3 [Howard et al., 2019] introduced "mobile and expand" modules, enabling the model to dynamically adjust the size and shape of the convolution kernels based on the input data. ShuffleNet [Zhang et al., 2018] utilized channel shuffling to divide the input feature channels into ordered groups, thereby reducing the computational cost of convolution. GhostNet [Han et al., 2020] proposed and introduced a "Ghost Module" to reduce feature redundancy, consequently decreasing the number of parameters and enhancing model performance.

In this work, we employ EfficientNet [Tan and Le, 2019] for handling the image processing part of our task.

EfficientNet excels in achieving outstanding performance under limited computational resources as a highly efficient CNN architecture. The key strategy of EfficientNet lies in introducing Compound Scaling, a method that adjusts the depth, width, and input resolution of the network simultaneously to maintain relatively balanced parameter numbers and lower computational complexity, thus achieving the highest possible model performance. Compound Scaling consists of two hyperparameters: depth multiplier

and width multiplier. The depth multiplier adjusts the network depth, thereby affecting the model's representational capacity. Increasing the depth multiplier makes the model deeper and learns more abstract features. The width multiplier adjusts the number of channels per layer, thereby affecting the model's capacity. Increasing the width multiplier increases the number of channels per layer, helping to capture more information. In addition, Compound Scaling adjusts the input resolution to enable the model to handle larger input images, thus capturing more spatial information. Apart from that, to improve computational efficiency and parameter utilization, EfficientNet also adopts optimization techniques such as Inverted Residual Blocks and Squeeze-and-Excitation (SE) Blocks [Howard et al., 2017, Sandler et al., 2018, Hu et al., 2018]. Specifically, Inverted Residual Blocks use a special residual block structure that reduces parameter numbers and computational complexity by employing a bottleneck layer. The bottleneck layer reduces input and output channel numbers by adding 1x1 convolution layers between convolution operations. An Inverted Residual Block's basic structure can be roughly divided into four modules: a 1x1 convolution layer used to reduce the input channel number, a depthwise separable convolution layer employing special convolution operations to separate spatial convolution and channel convolution and reduce computational complexity, another 1x1 convolution layer to increase the output channel number, and the residual connection which achieves residual learning by adding input and output, reducing the likelihood of gradient vanishing problems to a certain extent. Squeeze-and-Excitation Blocks are modules used for adaptively adjusting channel-wise features, enabling the model to focus more on channels with a greater impact on performance. SE blocks first compress the input image features by global average pooling, mapping each channel's feature to a scalar, and then calculate the channel-wise feature dependencies. Subsequently, fully connected layers generate channel weights, and the channel description vector obtained from the squeeze stage is outputted through a hidden layer and an output layer to output the weights. This output serves as an evaluation indicator of the importance of each channel, activating and learning the channel dependencies, thus achieving adaptive channel adjustment. These strategies and techniques allow EfficientNet to have a smaller model size, enabling fast and efficient computation and resource utilization.

## 2.2 NLP Models

In the field of deep learning natural language processing models, the emergence of the word2vec model, applied to word embedding techniques, laid a solid foundation for natural language processing models. The word2vec model maps traditional text sequences to higher-dimensional Euclidean spaces, enabling computability of language text and, to a large extent, providing mathematical interpretability for the text [Mikolov et al., 2013]. Soon after, GloVe further optimized word2vec's word embedding method by taking advantage of global word to word co-occurrence matrix factorization, which further introduced global information into consideration, while also adding more linear information capture.

Recurrent Neural Network models initiated by RNN then have been continuously applied as a classical architecture in subsequent models. RNNs take text as sequential input and continuously update parameter weights with data input, thus achieving se-

mantic understanding by combining the context in the text [Elman, 1990]. Based on the RNN model, the Long Short-Term Memory (LSTM) model specifically addresses the gradient explosion and gradient vanishing problems faced by RNNs and proposes an architecture for long and short-term memory. The bidirectional LSTM architecture further increases the ability to capture text information [Hochreiter and Schmidhuber, 1997]. In many ultra-lightweight model architectures, it is still popular for developers.

The Transformer model, through the development and application of the Attention mechanism, is based on the self-attention architecture and achieves the interpretation of multiple relationships between tokens in the text sequence, further addressing the model's memory problem [Vaswani et al., 2017]. Afterward, large-scale unsupervised pre-trained models based on the Transformer architecture, such as BERT and GPT, became mainstream neural network model construction methods due to their stability and cost-saving training features [Devlin et al., 2018, Radford et al., 2018]. GPT employs a unidirectional Transformer decoder architecture and learns language knowledge through large-scale unsupervised pre-trained. It then fine-tunes on specific tasks. The pre-trained process is mainly based on auto-regressive language models, which predict the next word given the preceding text [Radford et al., 2018]. BERT, on the other hand, starts with an encoder and uses a bidirectional Transformer encoder to capture the context information in the text simultaneously. BERT learns rich language knowledge through large-scale unsupervised pre-trained and fine-tunes on specific tasks [Devlin et al., 2018]. The pre-trained process includes two tasks: Masked Language Model (MLM) and Next Sentence Prediction (NSP). It has achieved significant results in various NLP tasks.

These models based on transformer architecture not only improve task performance but also broaden application domains. Through continuous optimization and improvement, these models have great potential for future development. However, these models are difficult to be carried out in some specific scenarios due to their large scale. Follows are some specific introduction of some lightweight variants of BERT that still perform well on some tasks. If the size requirement of our task could be a little bit larger, they are also considerable to be loaded.

ALBERT, which a lightweight variant of BERT, reduces the model's parameter size through hierarchical parameter sharing. Additionally, it factorizes the embedding matrix into two smaller matrices, reducing computational complexity [Lan et al., 2019]. Furthermore, it adopts a new Sentence Order Prediction (SOP) task to adjust the NSP task, focusing more on the coherence between sentences. As a result, ALBERT maintains similar performance to BERT while reducing model size and computational cost. DistilBERT is a distilled version of BERT that learns from BERT through knowledge distillation techniques [Sanh et al., 2019]. Knowledge distillation is a method of training small models (student models) to mimic the performance of large models (teacher models). The structure of the DistilBERT model is similar to BERT, but the model parameters and computational complexity are reduced by approximately half. By retaining BERT's key features and structure, DistilBERT achieves similar performance but is more efficient in terms of runtime speed and memory usage.

# Chapter 3

# Techniques

## 3.1 Principle

In our research, we focus on a specific scenario, the construction of lightweight neural network models, and a particular training dataset comprising 30,000 entries of captions and images as inputs, with multiple labels serving as classification outputs. To effectively address these challenges, we introduce a multi-modal pre-trained model incorporating both Computer Vision (CV) and Natural Language Processing (NLP) components. Moreover, we employ an array of methodologies to optimize both the data and the training process. The core methods are as follows:

### 3.1.1 Sigmoid

The Sigmoid function is a classic activation function that has seen extensive use in the field of neural networks [Hinton et al., 2012]. It transforms linear data flow into a non-linear format, thereby significantly enhancing the versatility of the models it is utilized in. The Sigmoid function is monotonically increasing, and its mathematical representation is as follows:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{3.1}$$

The defining feature of the Sigmoid function is its ability to map any real-valued number into a distinct value in the interval between 0 and 1, with these two endpoints serving as its asymptotic limits. Notably, as a smooth and differentiable non-linear function, the Sigmoid function's widespread application in various domains is well justified.

### 3.1.2 Batch Normalization

Batch Normalization is a sampling-based method that has been successful in accelerating the training of neural networks and stabilizing gradient computations [Ioffe and Szegedy, 2015]. This technique alleviates the influence of different dimensional data on the parameters and mitigates the vanishing gradient problem to a certain extent by normalizing the input data of the corresponding layer. The normalization of input data

is performed as follows:

$$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \varepsilon}} \tag{3.2}$$

In this formula, $\mu$ and $\sigma$ represent the estimated mean and standard deviation of the batch samples, respectively. Meanwhile, $\varepsilon$ is a small value introduced to prevent the occurrence of $\sigma$ equalling zero (which would be the case if all values in the batch are identical).

### 3.1.3 Dropout

Dropout is a regularization technique commonly employed in the context of deep learning to prevent overfitting. The Dropout method operates by randomly setting a subset of neuron outputs in a layer to zero during training. This stochastic nature promotes the dispersion of learned features across the network, thereby reducing reliance on specific neurons and enhancing the model's generalization capabilities. Notably, the proportion of neurons to drop, often referred to as the dropout rate, is a tunable parameter and a subject of empirical optimization.

### 3.1.4 Adam Optimizer

Adam, an acronym for Adaptive Moment Estimation, is a widely-used optimization algorithm in the field of deep learning. Unlike traditional stochastic gradient descent (SGD), which maintains a single learning rate for all weight updates, Adam utilizes adaptive learning rates for different parameters. This behavior is achieved by maintaining an exponentially decaying average of past gradients and squared gradients, which serve as estimates of the first moment (the mean) and the second raw moment (the uncentered variance) of the gradients, respectively. Mathematically, Adam's weight update rules can be represented as:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.3}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{3.4}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{3.5}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.6}$$

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \tag{3.7}$$

Where $\beta_1$ and $\beta_2$ control the decay rates of the first and second moment estimates, $g_t$ is the gradient at timestep $t$, $\alpha$ is the learning rate, and $\varepsilon$ is a small constant to avoid division by zero.

### 3.1.5 ReduceLROnPlateau

The *ReduceLROnPlateau* is a strategy for adjusting the learning rate based on the model's performance, typically used in conjunction with PyTorch. This strategy reduces the learning rate when a metric has stopped improving, which can be particularly beneficial for finding the optimal solution or escaping from saddle points during the model's training process. The basic idea is to monitor a specified metric, and when that metric has ceased to improve for a 'patience' number of epochs, the learning rate is reduced by a certain factor.

### 3.1.6 Multi-Modal

Multi-modal learning is a subfield of machine learning that seeks to build models capable of processing and correlating information from multiple input modalities. A modality refers to a particular way of capturing or representing data, such as text, images, audio, etc. The ultimate goal of multi-modal learning is to create models that can understand and leverage the intrinsic correlations among multiple modalities, thereby generating more holistic and nuanced representations of data. This approach is particularly applicable to tasks where different types of data are naturally intertwined, like video processing (combining visual and auditory data) or social media analysis (combining text, image, and possibly other types of data).

### 3.1.7 Data Preprocessing

The data preprocessing step is a crucial part of any machine learning and deep learning project. The given code implements a custom dataset class for loading and preprocessing the data, which includes images and their corresponding captions. This class, named *DataLoad*, is a subclass of the $torch.utils.data.Dataset$ class in PyTorch, which is an abstract class representing a dataset and allows for easy iteration over the data during training.

The *DataLoad* class includes several methods. The '$\_\_init\_\_$' method initializes the dataset by reading the data file and performing some preprocessing on the captions. The '$\_\_len\_\_$' method returns the total number of samples in the dataset. The '$\_\_getitem\_\_$' method returns a specific sample from the dataset based on an index.

If $text\_csv$ is $True$, it implies the data file includes caption text in a separate CSV file. In this case, each sample returned by '$\_\_getitem\_\_$' is a dictionary that contains the image, the image ID, and the caption. If $text\_csv$ is $False$, each sample also includes the label of the image. The label is one-hot encoded and summed across all classes.

If a transform is provided when initializing the dataset, it is applied to the image before returning the sample. This could include transformations such as resizing, normalization, data augmentation, etc. Here are some of the key equations involved in the data preprocessing process:

$$\text{image labels} = \text{one-hot encode}(\text{image labels}) \tag{3.8}$$

$$\text{final labels} = \text{sum}(\text{image labels across all classes}) \tag{3.9}$$

$$\text{final image} = \begin{cases} \text{apply transformation to the image, if transform is provided} \\ \text{original image, otherwise} \end{cases} \tag{3.10}$$

**Image Data Processing**

Image data requires preprocessing and augmentation to create a robust model. In the provided code, a dictionary named $transforms$ is defined, which contains different image preprocessing pipelines for the training and validation datasets.

For the training dataset, the preprocessing pipeline includes converting the image to a tensor, resizing it to 256x256, applying random horizontal flip for data augmentation, adding random color jitter, applying random affine transformations (like translations, rotations, and scaling), and normalizing the pixel values using the mean and standard deviation of the ImageNet dataset.

For the validation dataset, the preprocessing pipeline is simpler, involving only converting the image to a tensor, resizing, and normalizing.

Following the definition of the $transforms$, the training and validation datasets are created using the custom *DataLoad* class, with the corresponding transforms applied. The training dataset is then split into a training set and a validation set using the $torch.utils.data.random\_split()$ function. The proportion of the dataset used for training is defined by $TRAIN\_VAL\_PROP$.

Finally, *DataLoader* objects are created for the training, validation, and test datasets. These provide an iterable over the datasets, returning batches of data that can be used for training or evaluating the model. The batch size is defined by $BATCH\_SIZE$, and $shuffle = True$ is set for the training and validation loaders to ensure that the data is shuffled before being divided into batches. Here is the key equation involved in the data preprocessing process:

$$\text{final images} = \text{apply corresponding transformations}(\text{original images}) \tag{3.11}$$

**NLTK & Other Caption Data Processing**

Natural Language Toolkit (NLTK) is a leading platform for building Python programs to work with human language data. In the given Python code, the NLTK library is used for text preprocessing of the captions. The process involves several steps including converting the text to lowercase, removing non-alphabetic characters, splitting the text into words, and performing lemmatization using NLTK's $WordNetLemmatizer()$. Lemmatization reduces words to their base or root form. For instance, "running" is transformed to "run".

In our code of preprocessing, we also remove stopwords, which are commonly used words (e.g., "the", "a", "an", "in") that do not contribute much meaning to the sentences. A set of specific 'error words' are defined and filtered out. The processed captions are then tokenized, which involves converting the words into corresponding indices based on a vocabulary dictionary. This dictionary is created using a pre-trained word embedding model. The tokenized captions are then padded with a $[PAD]$ token or truncated to a maximum length, making them uniform in size. Here are some of the key equations

involved in the word embedding process:

$$\text{words in caption} = \text{lemmatize}(\text{words in caption without stopwords}) \tag{3.12}$$

$$\text{tokenized captions} = \text{convert words to corresponding indices in vocabulary dictionary} \tag{3.13}$$

$$\text{final captions} = \begin{cases} \text{pad captions to maximum length, if caption length} < \text{max length} \\ \text{truncate captions to maximum length, if caption length} > \text{max length} \end{cases} \tag{3.14}$$

### 3.1.8 CV Model Introduce

In the scope of our research, we have opted for EfficientNet B1 as the primary model for handling computer vision (CV) tasks. The decision to employ EfficientNet B1, a member of the EfficientNet family, is both deliberate and crucial to our study. This family of models, designed to scale in a structured manner surpassing traditional methods, offers an outstanding equilibrium between accuracy and computational efficiency. The selection of EfficientNet B1, in particular, is influenced by its balanced trade-off between model size, computational demands, and performance. This balance renders it apt for our study, given the significant consideration of computational resources and efficiency. Moreover, EfficientNet B1 exhibits competence in managing diverse and intricate image features with its depthwise separable convolution and compound scaling method. This capability not only enhances the model's learning proficiency but also bolsters its generalizability.

### 3.1.9 NLP Model Introduce

Our methodology for processing captions necessitates the use of Long Short-Term Memory (LSTM) as the core model for natural language processing (NLP) tasks. LSTM, a variant of recurrent neural networks, is well-equipped to handle long-term dependencies in sequential data, such as textual content, courtesy of its unique memory cell structure. This makes LSTM an exemplary choice for processing caption text, which frequently contains contextual information spanning multiple words or even sentences. Further, the ability of LSTM to selectively remember and forget information via its gating mechanisms considerably mitigates the common issue of vanishing and exploding gradients encountered during the training of deep recurrent networks. This feature ensures both stability and efficacy in our training regimen.

## 3.2 Reasonability

The selection of various techniques and models used in our study is based on their suitability and effectiveness in addressing the problem at hand. In this section, we provide the rationale behind the choices made for some key components.

The choice of the sigmoid activation function in our approach is crucial, and it cannot be replaced with other activation functions such as ReLU, Tanh, or softmax. This choice

is dictated by the specifics of our study case, particularly the multi-label output for each entity. The sigmoid function affords each output label an independent probability, which is a stark contrast to the softmax function that confers conditional probabilities relative to other labels.

Batch normalization is used to mitigate the problems of internal covariate shift by normalizing layer inputs. This is particularly beneficial in our case as it stabilizes the learning process and drastically reduces the number of training epochs required.

Dropout is an essential tool to combat overfitting in our model. By randomly nullifying outputs from neurons during training, it introduces randomness and helps the model generalize better to unseen data.

Data preprocessing is necessary to ensure that our model receives quality data in an acceptable format. By applying image transformations and text tokenization, we are able to standardize the input and improve the overall performance of our model.

The multi-modal approach, combining Computer Vision (CV) and Natural Language Processing (NLP) components, allows our model to process and learn from both image and text data. This is particularly effective for our data set, which comprises both images and their corresponding captions.

The Adam optimizer was chosen for our model due to its efficient memory usage and capability for large-scale problems. It combines the benefits of two other extensions of stochastic gradient descent, AdaGrad and RMSProp, making it particularly suitable for our dataset.

ReduceLROnPlateau is used as a learning rate scheduler that reduces the learning rate when a metric has stopped improving. This helps us fine-tune our model in the later stages of training when the model is close to the optimal solution.

The choice of EfficientNet B1 as the primary model for computer vision tasks in our study is deliberate and significant. EfficientNet, a family of models that scale in a more structured manner compared to conventional methods, provides an excellent balance between accuracy and computational efficiency. Specifically, EfficientNet B1 is chosen due to its balanced trade-off between model size, computational demands, and performance, making it particularly suited for our study where computational resources and efficiency are critical considerations. Furthermore, EfficientNet B1 is capable of handling diverse and complex image features with its depthwise separable convolution and compound scaling method, thereby enhancing the model's ability to learn and generalize.

Finally, Long Short-Term Memory (LSTM) has been selected as the core model for natural language processing tasks in our study. LSTM, a type of recurrent neural network, is particularly adept at handling long-term dependencies in sequence data such as textual content, owing to its unique memory cell structure. This makes LSTM an ideal choice for processing caption text, which often contains contextual information that spans across multiple words or even sentences. Additionally, LSTM's ability to remember and forget information selectively through its gating mechanisms significantly mitigates the problem of vanishing and exploding gradients, a common issue in training deep recurrent networks, thereby ensuring stability and effectiveness in our training process.

## 3.3 Advantage & Novelty

Beyond these fundamental methods, we also explored several novel strategies, retaining those that yielded beneficial results. Detailed descriptions of these innovations are as follows:

### 3.3.1 Focal Loss

Focal Loss is an improved version of the cross-entropy loss function, designed specifically to tackle the problem of class imbalance in classification tasks. It operates by down-weighting the contribution of easy-to-classify examples and focusing more on the hard-to-classify ones, which are typically under-represented. The mathematical formulation of Focal Loss is as follows:

$$FL(p_t) = -(1-p_t)^{\gamma}\log(p_t) \tag{3.15}$$

In this formula, $p_t$ is the model's predicted probability for the true class, and $\gamma$ is a tunable focusing parameter that adjusts the rate at which easy-to-classify examples are down-weighted.

### 3.3.2 BCEWithLogitsLoss

*BCEWithLogitsLoss*() is a loss function provided in the PyTorch library. It combines a Sigmoid activation function and Binary Cross Entropy (BCE) Loss into a single class. This confluence is beneficial because it addresses the numerical stability issues that can arise when applying a standalone Sigmoid function to the output layer of a network. The formula for *BCEWithLogitsLoss*() is:

$$L = -w\left[y \cdot \log(\sigma(x)) + (1-y) \cdot \log(1-\sigma(x))\right] \tag{3.16}$$

Where $w$ is a weight, $y$ is the target value, $x$ is the predicted value, and $\sigma(x)$ is the result of applying the Sigmoid function to the predicted value.

# Chapter 4

# Experiments and Results

## 4.1 Hardware and Software Specification

Above all, all the performance of models are evaluated on the Colab GPU hardware accelerator with the type of T4 in a High-RAM runtime shape.

- *Platform*: Google Colab Pro
- *GPU*: NVIDIA Tesla T4 High-RAM

The further details of the NVIDIA Tesla T4 High-RAM are shown below,

| Parameter | Value |
|---|---|
| CUDA Cores | 320 |
| Tensor Cores | 320 |
| Memory Type | GDDR6 |
| Memory Capacity | 16 GB |
| Memory Bandwidth | 320 GB/s |
| Max Power Consumption | 70 W |
| Interface Type | PCIe 3.0 x16 |
| Max Resolution | 7680 x 4320 |
| Dimensions | 111.15 mm x 267 mm |

Table 4.1: NVIDIA Tesla T4 Server Specifications

In addition, the python environment and used python packages are shown below,

- *Python* 3.10.11
- *pytorch* 2.0.0
- *CUDA* 11.8
- *gensim* 4.3.1
- *numpy* 1.22.4

- *pandas* 1.5.3
- *scikit − image* 0.19.3
- *matplotlib* 3.7.1
- *tqdm* 4.65.0
- *torchvision* 0.15.1
- *scikit − learn* 1.2.2
- *nltk* 3.8.1

## 4.2 Performance

Only the performance of the model with the best hyperparameters will be analyzed in this part because there are too many experiments. Therefore, it is complicated to list all performance of models. This part will illustrate one of the best model's performance in terms of different evaluation metrics and its efficiency.

The embedding model $glove - wiki - gigaword - 50$ and pre-trained CNN model $EfficientNet\_b1$ are used for this classification task. The learning rate is 0.0005 which is obtained after several experiments. At the same time, the scheduler $ReduceLROnPlateau()$ will adjust the learning rate after each epoch. In addition, the optimizer *Adam* and the loss function $BCEWithLogitsLoss()$ are used for the model training. The detail performance are shown below,

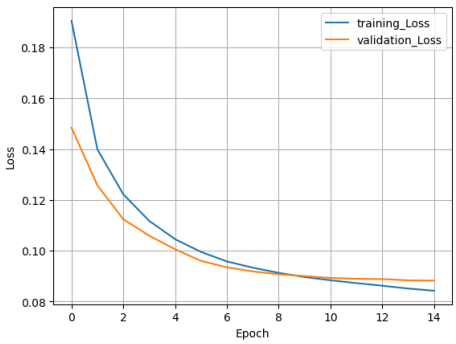



Figure 4.1: Loss Performance

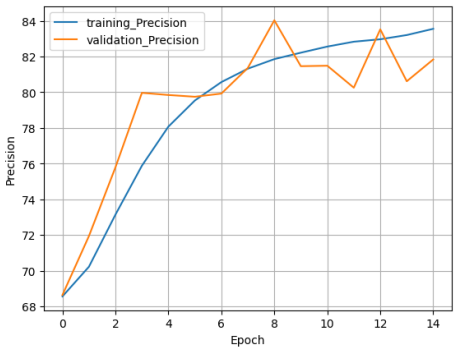



Figure 4.2: Precision Performance

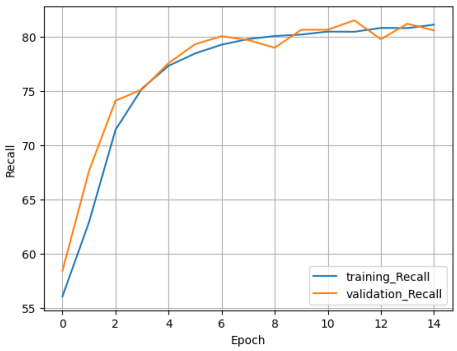



Figure 4.3: Recall Performance

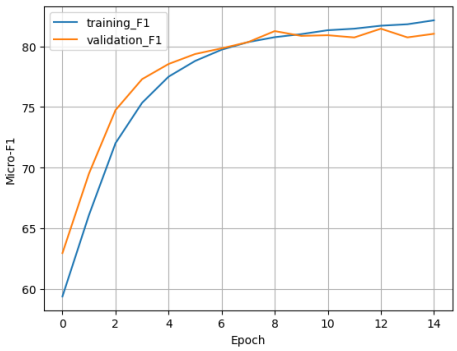



Figure 4.4: Micro-F1 Performance

The metrics in this project are the Micro-F1, which reflects the overall performance of the model in multi-class classification tasks. It considers both the accuracy of the model's predictions for each class and its ability to recognize true positive cases. In addtion, the precision and recall are also considered as the evaluation metrics. The precision value quantifies how many of the predicted positive samples are truly positive. In simple terms, precision reflects the accuracy of the model's positive predictions. The recall value quantifies the proportion of true positive samples that are correctly identified as positive by the model. In other words, recall reflects the model's ability to identify true positive cases.

According to the Figure 4.1, both the training loss and the validation loss of the this model become very stable and approaches to zero after 15 epochs. But there is a slight overfitting about 0.01 difference after 10 epochs, which are be ignored and regarded as a normal condition. The training precision of this model increases stably, while the validation precision shocks after 6 epochs. In addition, there is also a slight overfitting in the last few epochs according to the Figure 4.2. From the Figure 4.3, the recall value of the both training process and validation process are same and reach about 80 percents. At the same time, there is no overfitting phenomenon for this metric. The Micro-F1 is the most important evaluation metric for this experiment. The Figure 4.4 shows that the F1 score reaches more than 80 percents on the validation dataset. The overfitting phenomenon is same as the recall, which only has a slight overfitting. More details of the performance are shown below,

| | **Loss** | **Precision** | **Recall** | **Micro-F1** | **Time** |
|---|---|---|---|---|---|
| **Train** | 0.0843 | 83.5519% | 81.1203% | 82.1465% | 272min 38s |
| **Validation** | 0.0882 | 81.8350% | 80.5979% | 81.0348% | 68min 6s |

Table 4.2: Performance Results

As shown in the Table 4.2, the performance of the training dataset and validation dataset are all well, which are more than 80 percents. The precision, recall and F1 scores can all show the efficiency of the model. The higher of them, the better performance model is. It is expected that the time duration is much long because the large dataset and the deep neural network model. The training process takes it about 340 minutes where the training dataset takes 4 times duration longer than the validation dataset. In addition, the duration of the first epoch is about 3 hours. But each iteration only takes about 10 minutes after the first iteration.

The score of the test dataset on the Kaggle Competition is about 86.01%, which can be regarded as a good performance for the dataset. In conclusion, the performance of the model has reached the initial expectation.

## 4.3 Extensive Analysis

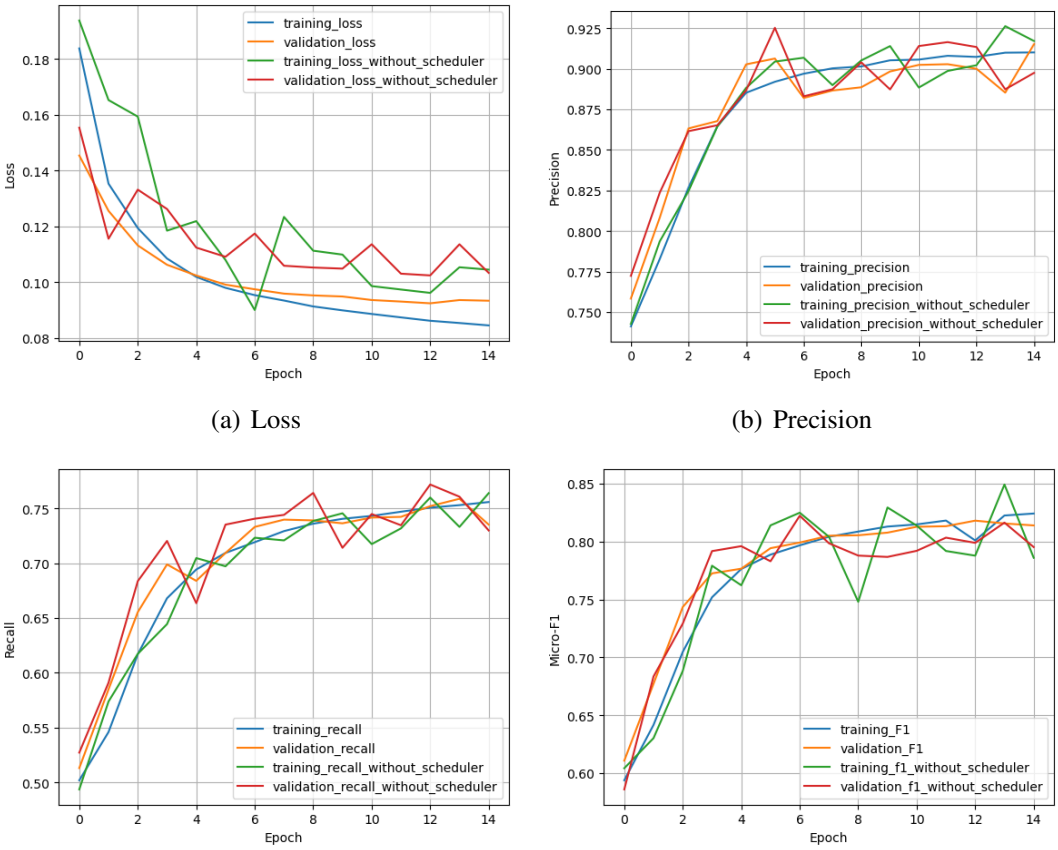
For extensive analysis, the project compares different methods and optimizer to find the best performance of the model. In addition, the ablation studies are applied to investigate the influence of the different modules and functions. Therefore, this part will select several methods, modules and different hyperparameters to apply for the model

experiments. The results and performance of these different parts will be elaborated in the following.

### 4.3.1 Ablation Studies

In this project, the *torch.optim.lr_scheduler* module is used to adjust the learning rate during training. It is technique used to dynamically modify the learning rate of an optimizer over time, typically based on the epoch or iteration number. The reason for why use it is that it can improve the stability of training and help control the learning rate to reduce the risk of overfitting.

Therefore, the project completes two experiments which are one with the learning rate scheduler and one without it. The results are shown below,




(a) Loss (b) Precision

(c) Recall (d) Micro-F1

Figure 4.5: Scheduler Ablation Study

According to the Figure 4.5, the learning rate scheduler enables the model to be trained in a smoother manner. With the use of a learning rate scheduler, all indicators can steadily rise and fall. At the same time, the increasing trend of f1 value is more stable. Without the use of learning rate, the loss of the model decreases more unstable. In addition, the f1 value did not quickly reach the convergence value. That is to say, the convergence speed of the model has slowed down due to the lack of use of a learning rate scheduler. If a relatively high learning rate is used, it may result in the model not being able to find local or global optima. More details are shown below,

| | Loss | Precision | Recall | Micro-F1 | Time |
|---|---|---|---|---|---|
| **With Scheduler** | 0.0843 | 91.0119% | 75.5747% | 82.4106% | 340m 44s |
| **Without Scheduler** | 0.1045 | 91.7160% | 76.3839% | 78.5927% | 320m 31s |

Table 4.3: Scheduler Ablation Study Results

The Table 4.3 shows that if the learning rate scheduler is not used, the final loss of the model will become higher under the same epoch. In addition, the obtained f1 value is approximately 4% lower than the model using a learning rate scheduler. This is all because not using a learning rate scheduler will reduce the convergence speed of the model and make the training process more bumpy. Although the Table 4.3 shows a 20 minute reduction in training time, this is an acceptable uncertainty factor. Because the state of the hardware may vary during each training session.

### 4.3.2 Comparison Methods

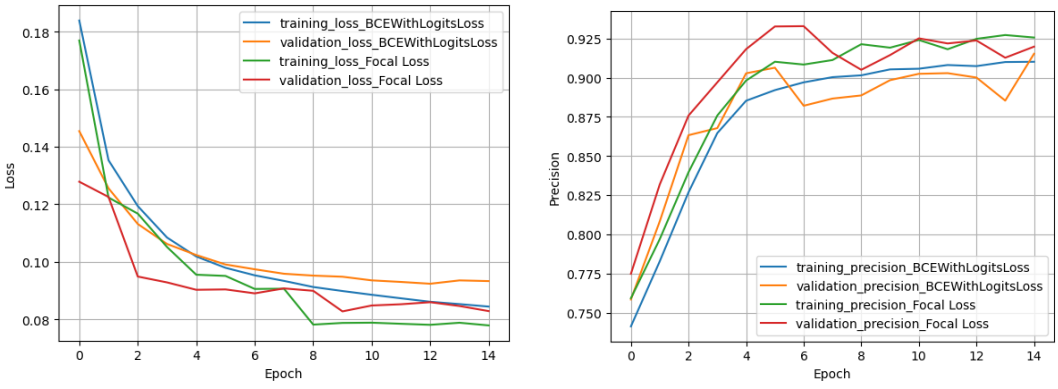
The main part of this project is the loss function. There are many different loss function in the *torch.nn* module. For multi-label classification tasks, the most popular function is the *BCELoss*() or *BCEWithLogitsLoss*(), which are used in this project before. However, there is a serious imbalance in the dataset labels used in this project. The number of a certain label accounts for half of the total number of labels, while all other labels combined are not as many as this one label. Therefore, on this basis, focal loss was used, which can effectively improve the problem of data imbalance to improve model performance.

The project compares the different performance between the *BCEWithLogitsLoss*() loss function and the Focal Loss. The results are shown in Figure 4.6. According to the Figure 4.6, focal loss can effectively slightly improve the performance of the model. It not only reduces the loss of the model, but also improves by about 2% compared to *BCEWithLogitsLoss*() on other standards such as precision, recall, and f1. Focal loss has the same trend as using *BCEWithLogitsLoss*(), because the Focal loss in this project is implemented based on *BCEWithLogitsLoss*(). The focal loss of this project adds two parameters on the basis of *BCEWithLogitsLoss*() to control the weight of positive and negative examples. In a word, using focal loss can improve the performance on micro-f1 value. More details are shown below,

| | Loss | Precision | Recall | Micro-F1 | Time |
|---|---|---|---|---|---|
| **BCEWithLogitsLoss** | 0.0843 | 91.0119% | 75.5747% | 82.4106% | 340m 44s |
| **Focal Loss** | 0.0725 | 92.5239% | 78.5234% | 84.8712% | 371m 29s |

Table 4.4: Compare Loss Function Results

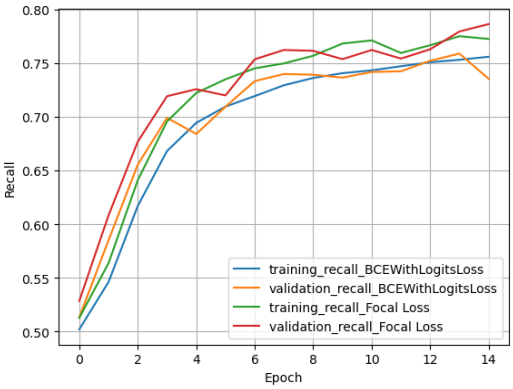
It is obviously that using the focal loss is better than *BCEWithLogitsLoss*(). Although the time duration is 30 minutes longer than *BCEWithLogitsLoss*(), the reason is that the computation of the focal loss is a bit complex than the *BCEWithLogitsLoss*(). In conclusion, focal loss can effectively reduce the influence of the imbalanced dataset to improve the performance of the final score. Therefore, this project chooses the focal loss as the loss function, which is a advantage part mentioned before.



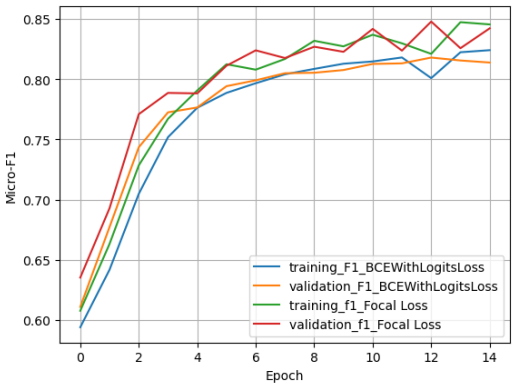

(a) Loss




(b) Precision




(c) Recall



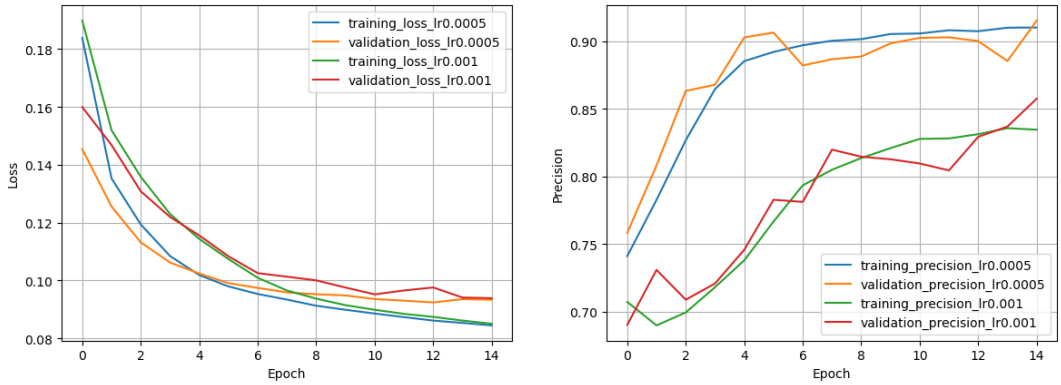

(d) Micro-F1

Figure 4.6: Compare Loss Function Method

### 4.3.3 Hyper Parameter Analysis

There are not many hyperparameter available for analysis in this project. Because the project uses pre-trained models where most of the parameters are set well. Therefore, the only hyperparameter that can be modified are the proportion of training set test set, learning rate and batch size. Obviously, only the learning rate has analytical significance. On the other hand, the project uses the learning rate scheduler mentioned before. So, the learning rate can not be very small because the scheduler will reduce the learning rate gradually. A larger learning rate can accelerate convergence but may lead to instability and failure to converge. A smaller learning rate can improve accuracy and generalization but requires more iterations.

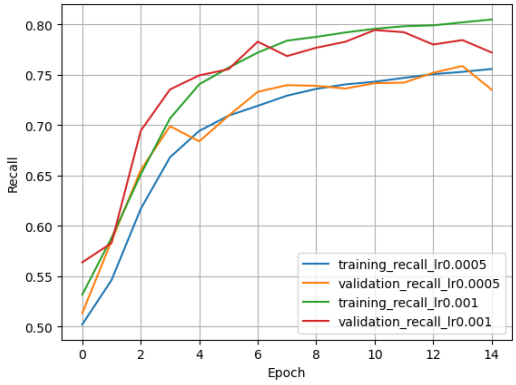
Therefore, The 0.001 and 0.0005 learning rate are compared in this project. The results are shown in Figure 4.7. According to the Figure 4.7, the model achieved a lower loss under training with a learning rate of 0.0005, although only 0.01 lower. The reason may be that the smaller learning rate improves the accuracy and generalization ability of the model. According to the Figure 4.7(b) and Figure 4.7(c), there is a phenomenon where the precision obtained with a learning rate of 0.0005 is higher than that obtained with a learning rate of 0.001, but the recall is lower. This is because a small learning rate makes the model more stable, reduces the possibility of mistake on classification, and thus improves accuracy. However, due to the small parameter updates, the model may require more iterations to achieve a higher recall rate, as some positive samples may



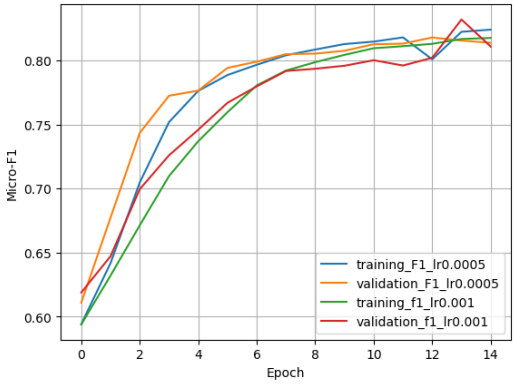

(a) Loss




(b) Precision




(c) Recall




(d) Micro-F1

Figure 4.7: Compare learning Rate

be misclassified as negative samples, resulting in a decrease in recall rate. Finally, the performance of two different learning rates on f1 values is almost identical. The detail results are shown below,

| | **Loss** | **Precision** | **Recall** | **Micro-F1** | **Time** |
|---|---|---|---|---|---|
| **lr=0.0005** | 0.0843 | 91.0119% | 75.5747% | 82.4106% | 340m 44s |
| **lr=0.001** | 0.0851 | 83.4609% | 80.4874% | 81.7625% | 346m 40s |

Table 4.5: Compare Learning Rate Results

The Table 4.5 shows that, although the performance of the two learning rates is basically the same, there is still a slight difference. Firstly, they consumed the same amount of time and the final loss was similar. However, overall, the learning rate of 0.0005 is slightly better than the learning rate of 0.001. They obtained similar f1 values under the same epoch. Therefore, more epochs may be needed in the later stages to achieve better performance with lower learning rates.

# Chapter 5

# Discussion and Conclusion

To draw a conclusion, this study achieves the multi-label classification task with efficient pre-processing methods, pre-trained models and relevant improvement methods based on given image and caption data to catch a higher F1 Score.

After trying various CNN and NLP models, this study ultimately uses the $EfficientNet\_b1$ and (Bi-)LSTM as the basic CNN model and NLP model respectively, integrated with regularisation and optimisation methods, such as Dropout and Adam, to mitigate the problem of overfitting and relevant hyperparameters tuning. In particular, $ReduceLROnPlateau()$ method is applied for adaptive learning rate tuning. Besides, in order to gain better performance of the overall model, some pre-processing procedures towards raw data are also implemented and applied in this study. In the aspect of NLP model, caption embedding (glove) is used for lexical and semantic feature extraction and word embedding based on foundational natural language processing steps like tokenisation to obtain a consolidated input of (Bi-)LSTM. For the raw data of EfficientNet, the raw label data is then transferred into one-hot vectors, and input image data is then transferred into tensors in a uniformed size of 256 x 256 with random horizontal flip, affine transformation and normalisation. To be specific, this study separates train dataset into train dataset and validation dataset so as to improve the holistic capability. Instead of widely used Cross-Entropy Loss Function, after many times of testing training, this study adhibits $BCEWithLogitsLoss()$ with an initial integrated Sigmoid function for stable probability.

On the basis of pre-defined training model, this study took many times of experiments with different values of different hyperparameters, obtaining the result that with learning rate and epochs of 0.0005 and 15 respectively, the overall model can reach the highest Micro-F1 score of 86.01% from several experiments.

Still, there can be other combinations of different pre-trained models, improvement methods and values of hyperparameters that can result in higher score. For instance, the Adam used in this study is not that much ideal to capture a stable performance. It may be better to use another optimiser for training. Meanwhile, though $BCEWithLogitsLoss()$ helps this study with higher score among many experiments, it is designed in the purpose of achieving better performance in binary classification tasks rather than multi-classification tasks. This study has tried Focal Loss which is more capable for multi-classification tasks, however, it turns out that Focal Loss is not that suitable for this model on the contrary. Thus, an improvement can be implemented starting in the direction of loss function and optimiser. In addition to improvement methods, since imple-

mented experiments could not cover all possible efficient models for image and caption learning, there can be other useful models that can be applied in this study to achieve better performance with higher score. Thus, there is still a lot of room for enhancement.

Overall, though there are still other possible solutions to reach a higher score, this study achieves its best performance via many times of testing and training.

# Bibliography


[Bojarski et al., 2016] Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., et al. (2016). End to end learning for self-driving cars. In *2016 European conference on computer vision*, pages 1–1. ECCV.

[Devlin et al., 2018] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

[Elman, 1990] Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2):179–211.

[Esteva et al., 2019] Esteva, A., Robicquet, A., Ramsundar, B., Kuleshov, V., DePristo, M., Chou, K., Cui, C., Corrado, G., Thrun, S., and Dean, J. (2019). A guide to deep learning in healthcare. *Nature medicine*, 25(1):24–29.

[Han et al., 2020] Han, K., Wang, Y., Tian, Q., Guo, J., Xu, C., and Xu, C. (2020). Ghostnet: More features from cheap operations. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1580–1589.

[Hinton et al., 2012] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[Howard et al., 2019] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al. (2019). Searching for mobilenetv3. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 1314–1324.

[Howard et al., 2017] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

[Hu et al., 2018] Hu, J., Shen, L., and Sun, G. (2018). Squeeze-and-excitation networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 7132–7141.

[Huang et al., 2018] Huang, G., Liu, S., Van der Maaten, L., and Weinberger, K. Q. (2018). Condensenet: An efficient densenet using learned group convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2752–2761.

[Iandola et al., 2016] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size. *arXiv preprint arXiv:1602.07360*.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

[Krizhevsky et al., 2017] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90.

[Lan et al., 2019] Lan, Z., Chen, M., Goodman, S., Gimpel, K., Sharma, P., and Soricut, R. (2019). Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*.

[LeCun et al., 2015] LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

[Mikolov et al., 2013] Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

[Radford et al., 2018] Radford, A., Narasimhan, K., Salimans, T., and Sutskever, I. (2018). Improving language understanding by generative pre-training. *OpenAI*.

[Sandler et al., 2018] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520.

[Sanh et al., 2019] Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2019). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*.

[Schmidhuber, 2015] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

[Tan and Le, 2019] Tan, M. and Le, Q. (2019). Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[Yan et al., 2018] Yan, Z., Li, X., Li, M., Zuo, W., and Shan, S. (2018). Shift-net: Image inpainting via deep feature rearrangement. In *Proceedings of the European conference on computer vision (ECCV)*, pages 1–17.

[Zhang et al., 2018] Zhang, X., Zhou, X., Lin, M., and Sun, J. (2018). Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856.

# Appendix A

# Code Link

https://drive.google.com/drive/folders/10N-tz-fD3NBOW1U6THLsOzT2Tu86NiJ5?usp=share_link

# Appendix B

# Instruction for Code

The code is build on the Google Colab. Therefore, you need to open the web link in Appendix A. The link is a shared folder in Google Drive. This folder includes 5 files, which are train data file, test data file, a zip file of image data, a predict file of the test data named *Predicted_labels.csv*, a model trained before named *best_steps.pth* and a code file named *Assignment2.ipynb*.

To run the code, you only need to open the *Assignment2.ipynb* file and **Run block by block**. The process of loading the dataset and downloading the pre-trained mdoels are included in the block. Before running the code, you should change the Colab runtime to the GPU runtime type.

To successful run the code. The following packages are required.

- *Python* 3.10.11
- *pytorch* 2.0.0
- *CUDA* 11.8
- *gensim* 4.3.1
- *numpy* 1.22.4
- *pandas* 1.5.3
- $scikit-image$ 0.19.3
- *matplotlib* 3.7.1
- *tqdm* 4.65.0
- *torchvision* 0.15.1
- $scikit-learn$ 1.2.2
- *nltk* 3.8.1