Task 2. Deep Learning for Image Classification

I. Introduction

Deep learning has been implemented in many aspect of our life. In the near future we will have driverless car on the street thanks to this technology. By 2011, the speed of GPUs had increased significantly, making it possible to train convolutional neural networks without layer-by-layer. With the increased computing speed, it became obvious Deep Learning had advantages in efficiency and speed. One example is AlexNet, a convolutional neural network whose architecture won several international competitions during 2011 and 2012[8].

We have been provided a image dataset contains 10 different classes. We will utilise the MobileNet V2 model to to classify our dataset and fine tune the model to improve performance.

II. Data and Preliminary Analysis

An image dataset is provided, in the dataset contains two folders, "train" and "validation", with 9,469 and 3,925 images inside respectively. In each folder images were separated into 10 classes i.e. "fish", "dog", "truck", "church", "golf_ball", "french_horn", "chainsaw", "truck", "petrol_station", "radio", Fig 8. gives an example of the images.

The dataset was first given with numeric classes which made it hard to interpret, so we manually change the classes to the represented object in there.

Figure 8. Example of the images from dataset

III. Methods

Different from the traditional image classification machine learning methods that use feature engineering to manually label features. Convolutional Neural Network (CNN) uses kernels(filters) to learn different features through automatic learning. The independence from requiring prior knowledge and human intervention is an major advantage.

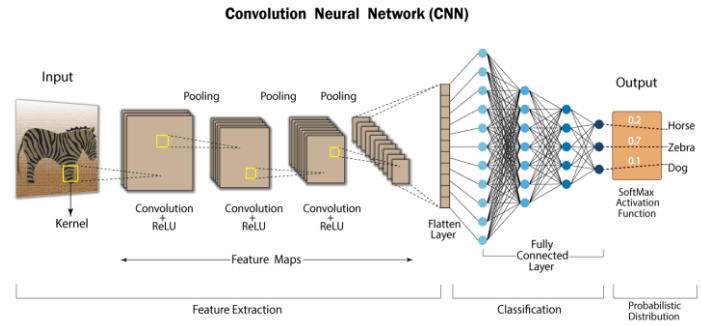

Figure 9. CNN stucture

The convolutional neural network consists of an input layer, hidden layers, and an out put layer. As the kernel slides along the input layers, the convolution operation generates feature maps, which will be the input of the next layer. This is followed by other layers e.g. pooling layers, normalisation layers, fully connected layers, a demonstration of structure can be found in Fig. 9[5]. Within each convolutional layer we need to set the activation function which is a node that is put at the end of or in between Neural Networks. They help to decide if the neuron would fire or not.

Pooling layers reduce the dimensions of data by summarising the outputs of neighboring groups of neurons in the same kernel map[6]. Typical pooling functions are *max pooling* which outputs the maximum of each sub-region from kernel, and *average pooling* which outputs the average of each sub-region from kernel. ReLU function is the most widely used activation function in neural networks today. One of the greatest advantage ReLU has over other activation functions is that it does not activate all neurons at the same time[7].

Fully connected layer is where the final classification is done after a several convolutional and pooling layers. The input to the fully connected layer is the output from the final convolutional layer or pooling layer, which will be flattened and feed into fully connected layer.

For our experiment we will observe the performance of MobileNetV2, a model developed at google trained on the ImageNet dataset, a dataset consisting of 1.4 million images with 1000 classes. ImageNet contains a wide variety of categories. As our dataset includes random categories of images, we assume the MobileNet V2 will be a applicable model.

IV. Experiments

A. Preprocess Data

With Tensorflow Keras package, we can import the image dataset straight from directory. When importing the images, we will need to define the size of our image when imported and the batch size. The reason we train model in batches instead of the whole training set is that the training procedure will require less memory and the network will train faster with small batches, which saved us more time and resource. Keras Application expects a specific kind of input preprocessing. For MobileNet, call the "mobilenet.preprocess_input" function to convert our inputs before passing them to the model. The function will scale input pixels between -1 and 1.

B. *Create Base Model*

To create the base model we need to decide which layer of the MobileNet V2 we will use for feature extraction. The classification layer can be discard as our prediction classes is different from the ImageNet data. We will define our fully connected layer to train our data in next step. This leave us the convolutional layers from MobileNet V2 which was pretrained with the ImageNet data. We assume the features it learnt will be cooperative with our task.

C. *Feature Extraction*

Feature Extraction use the representations learnt by a previous network to extract meaningful features from new samples. By adding a new classifier, which will be trained from scratch, on top of the pretrained model so that we can repurpose the feature maps learnt previously for the dataset.

We do not need to retrain the entire model. The base convolutional network already contains features that are generically useful for classifying pictures. However, the final, classification part of the pretrained model is specific to the original classification task in which we will add new classification layers and train with our dataset.

D. *Add dense layer*

Before adding dense layers, we need to convert features learnt from the convolutional layers to a vector per image in order to pass it through the dense layer. Here we added one dense layer with 64 kernels and a dropout layer to reduce parameters and avoid overfitting, finally we add the out put layer and set the activation function to "sigmoid" to get the probability of which class the image might be. See Fig. 10 for the model summary.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 160, 160, 3)]     0

 tf.math.truediv (TFOpLambda  (None, 160, 160, 3)      0
 )

 tf.math.subtract (TFOpLambd  (None, 160, 160, 3)      0
 a)

 mobilenetv2_1.00_160 (Funct  (None, 5, 5, 1280)       2257984
 ional)

 global_average_pooling2d (G  (None, 1280)             0
 lobalAveragePooling2D)

 dense (Dense)               (None, 64)                81984

 dropout (Dropout)           (None, 64)                0

 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 2,340,618
Trainable params: 82,634
Non-trainable params: 2,257,984
_________________________________________________________________
```

Figure 10. Final structure of the model.

E. *Compile Model*

After completing the model structure, we need to compile the model by defining the loss function, optimizer, and the metrics. For optimizer, we use Adam optimisation which combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimisation algorithm that can handle sparse gradients on noisy problems[9]. The learning rate which is the step size at each iteration to move toward a minimum of loss function will be 0.0001. Our dataset has more than two classes so we will use SparseCategoricalCrossentropy as loss function which expect labels to be integers. For metrics we use accuracy.

F. *Model training*

With the problem of computation speed, we will train our model, with only 3 epochs. The result of each training is in table III, the accuracy of the training set had increase 21% within 3 epochs and loss function dropped 76%.

TABLE I. PERFORMANCE OF MODEL TRAINING I

	Loss	Accuracy	Val_loss	Val_accuracy
Epoch 1	0.85	0.77	0.15	0.97
Epoch 2	0.15	0.97	0.08	0.98
Epoch 3	0.09	0.98	0.06	0.98

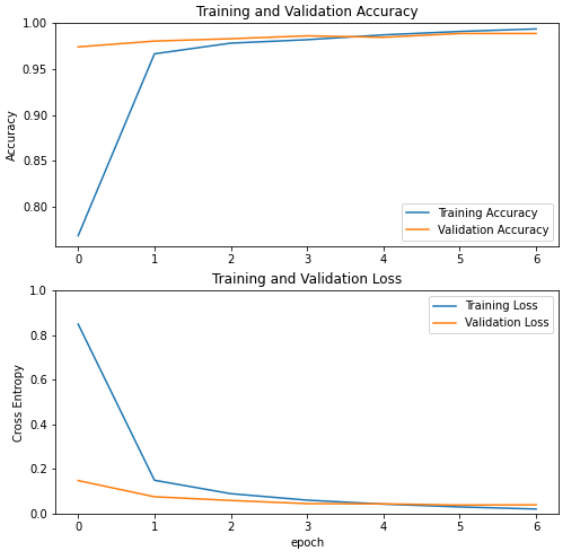

Figure 11. Accuracy and Loss performance in each epochs.

G. *Fine Tuning*

Although we have achieved high performance, applying finetuning might optimised our model. Unfreeze the base model, that consisting of 154 layers by setting the base model trainable, then define the fine tune number at 130. This will set all layers from 1 to 130 as not trainable(freeze), the rest 24 layers trainable(unfreeze). The reason why we unfreeze the upper layers is that at the bottom of the model layers will usually learn features that are more general, as the upper layers tend to be more specific to the dataset. By retraining the upper layers of the base model will increase the model to learn more features specific to the data. Continue training the previous model after fine tuning.

TABLE II. PERFORMANCE OF MODEL TRAINING II

	Loss	Accuracy	Val_loss	Val_accuracy
Epoch 4	0.04	0.98	0.05	0.98
Epoch 5	0.03	0.99	0.04	0.98
Epoch 6	0.02	0.99	0.04	0.98

We can see from table IV that the loss of training decrease to 0.02 at the 6th epoch and achieved 0.99 accuracy. With the validation dataset it also achieved very high performance.

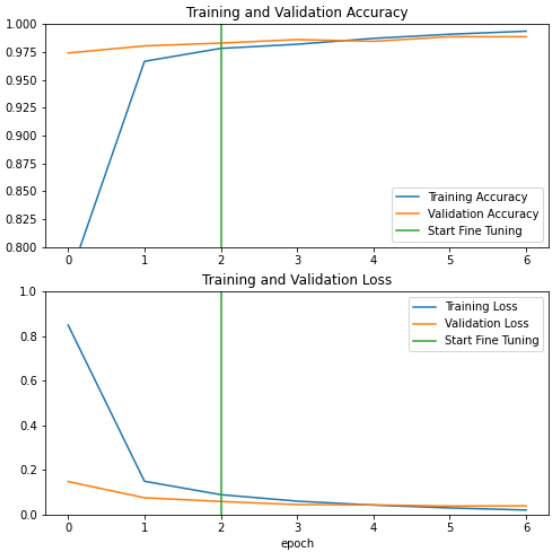

Figure 12. Accuracy and Loss performance after fine tuning.

H. Evaluation

At last we tested the fine tuned model on testing dataset and achieved 0.98 accuracy and 0.06 loss. The result is not so different from out training result which is a good sign representing the model is not over fitting.

V. REFLECTION

Transfer learning allow us to use robust models to apply on our data, which dramatically reduce the amount of time compare to train a model from scratch. But there are also disadvantages we should be aware of. Transfer learning only works if the initial and target problems are similar. For example, we probably will not use a model that was train on classifying animals and transfer the structure to work on medical data. Having high performance on one area does not mean we can apply it everywhere.

We have also tried to build a basic CNN model from scratch, but have notice the limitation of hardware. As the networks gets deeper the longer it takes to compute results, which makes it hard to optimised our model. In future we should also consider the resource we have access to when designing the model.