Machine Learning Engineer Nanodegree

Image Classification with Transfer Learning by implementing InceptionV3 and MobileNetV2

Jeffen Chen
September 8th, 2018

I. Definition

Project Overview

Neural networks have revolutionized many areas, such as natural language processing, image classification, and autonomous driving. However, as neural networks become more powerful, their complexity and training efficiency also pose challenges for humans. How to balance between accuracy and training efficiency of neural networks is a hot topic in recent years. Many excellent models and algorithms have also been developed, such as AlexNet[1], VGGNet[2], GoogLeNet[3], Inception[4]. These models and algorithms can achieve good results but take a lot of time and effort to train.

In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet[1], which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest.

Problem Statement

In the project, we will use the transfer learning method to train an existing large neural network for image classification. We expect the new model to have higher accuracy and training efficiency than a simple neural network built manually. Keras[5] already comes with some deep neural network models and trained weights. We will choose one model for pre-training and fine tuning. We will redefine the input layer and the fully connected layer associated with the output to better fit the image size and output we enter. The goal of this project is to use the transfer learning to classify the image dataset CIFAR-100 [6]. The project will select a model trained with imageNet and a manually built model to compare the epoch time and accuracy with benchmark. Our goal is to achieve higher training accuracy while exceeding the benchmark to assess the pros and cons of transfer learning in the CIFAR-100 application.

Metrics

The evaluation metric for this competition is multi-class classification accuracy i.e. the proportion of true class labels correctly predicted. Because the distribution of each type of image in the project is uniform, so accuracy can be calculated under following equation:

$$accuracy = \frac{True\ Predictions}{All\ Predictions} \times 100\,\%$$

II. Analysis

Data Exploration

I decided to choose CIFAR-100 dataset after research. This dataset is publicly available at University of Toronto website[7] and it has images in 100 classes containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the

CIFAR-100 are grouped into 20 superclasses. The 100 object class labels are shown in fig.1. 50000 labelled examples are provided for training, with a further 10000 unlabelled examples used for testing. Each images has 3 RGB colour channels and pixel dimensions 32×32 for an overall size per input of 3×32×32=3072.

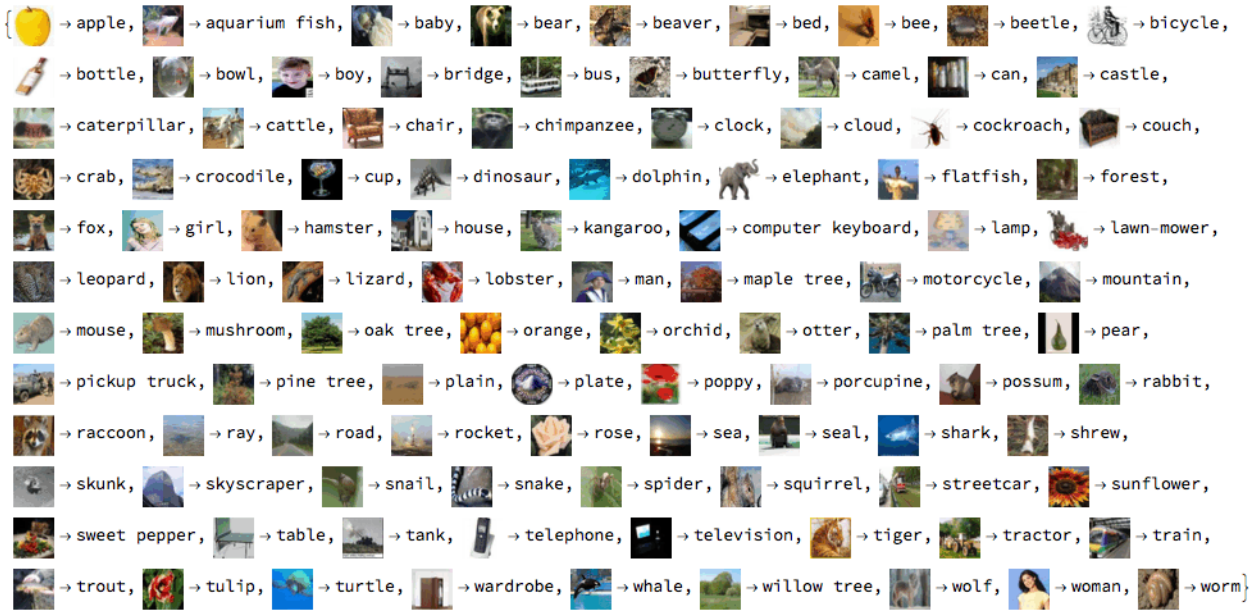

Figure 1: Image labels in CIFAR-100 dataset.

Exploratory Visualization

Since most existing deep neural networks do not support small images, we need to first convert the 32×32 image to 224×224 size using the cv2.resize method. However, in the actual application process, we found that the transformed picture has similar resolution but some pictures are distorted in colour, for example, the human face in the lower right corner becomes purple. Here are 24 images example after conversion, which will be the source of input data.stat_train.

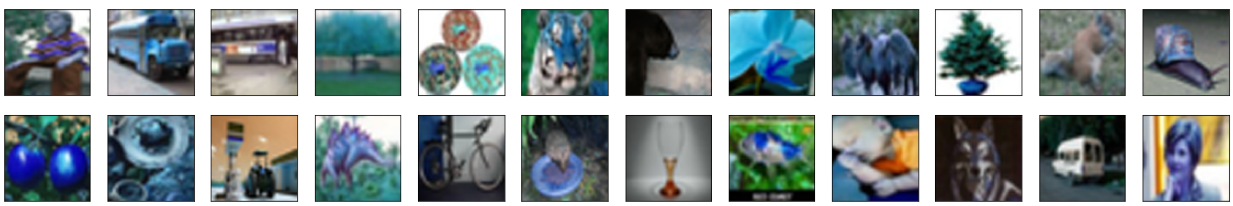

Figure 2: Sample images in CIFAR-100

I separately counted the number of images in different categories for train, validation, and test (50% of all data), as shown in Figure 2-1. It is worth noting that the distribution of the validation data and the training is not uniform, so this may cause the trained model to have a preference for some categories. For example, #60 and #61. And could also cause less accurate result in some class such as #56 and #9.

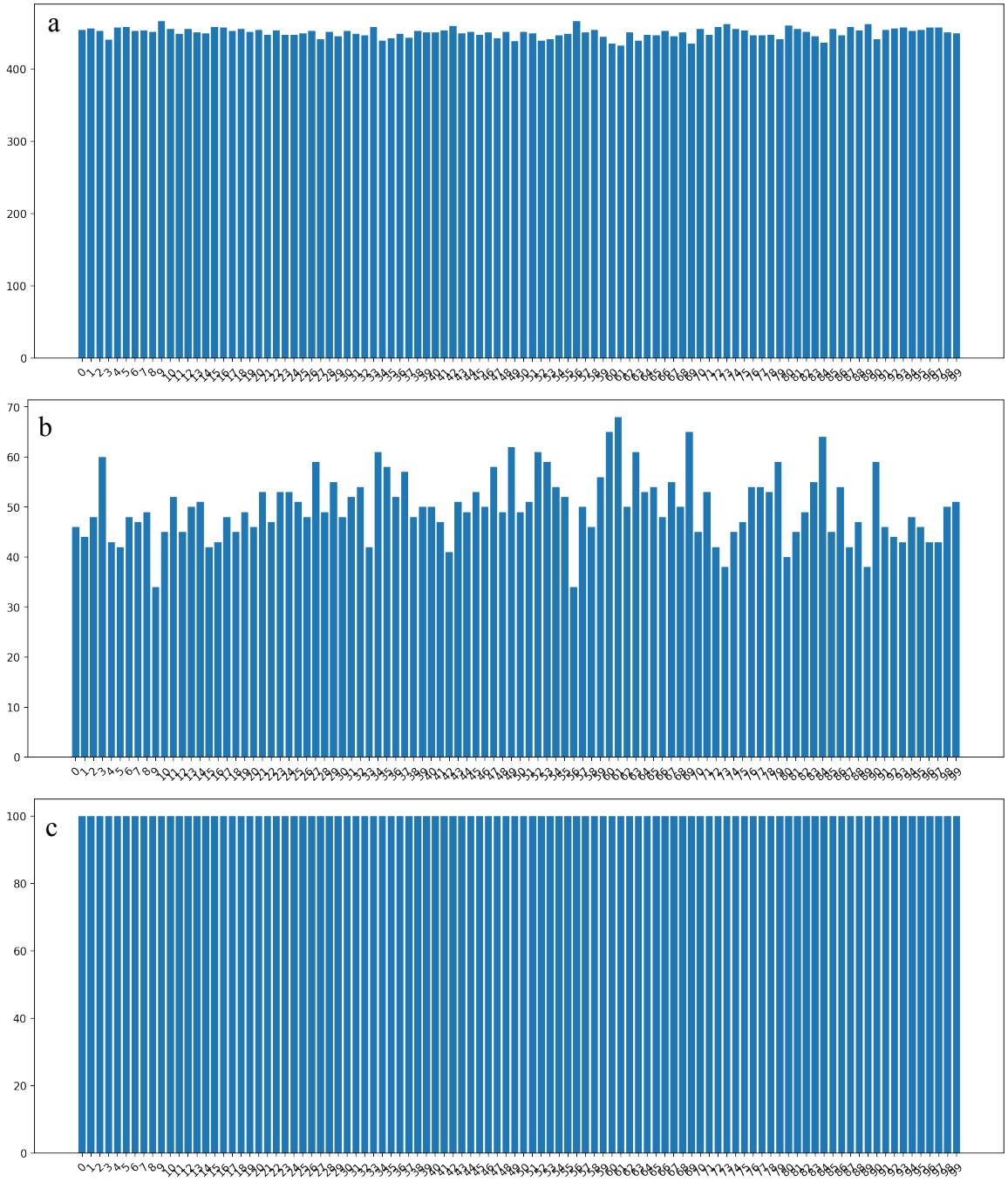

Figure 2-1: Image count for different data subset. a. Train dataset; b. Validation dataset; c. Test dataset

Algorithms and Techniques

Transfer learning is introduced in this project. I used InceptionV3 with weights pre-trained on ImageNet, and MobileNetV2 initialized randomly which built manually according to the paper by Mark Sandler. InceptionV3 is released by google in 2016. The namesake of Inception v3 is the Inception modules it uses, which are basically mini models inside the bigger model. The advantages of this model is that you can use 1×1 convolutions to reduce the dimensionality of your input to large convolutions, thus keeping your computations reasonable. According to the paper, InceptionV3 has better performance and accuracy than other frameworks like VGG and GoogLeNet. I was wondering how it will perform in this case. Because we are using transfer learning so only the top layers is replaced with the following structure:

global_average_pooling2d_1 (Glo	(None, 2048)	0	mixed10[0][0]
dense_1 (Dense)	(None, 1024)	2098176	global_average_pooling2d_1[0][0]
dropout_1 (Dropout)	(None, 1024)	0	dense_1[0][0]
dense_2 (Dense)	(None, 100)	102500	dropout_1[0][0]

Figure 3: InceptionV3 top layer structure

In the pre-training of InceptionV3, we freeze all layers except the top layer. In fine tuning, I want to keep the low-level features and only train the advanced features, so I only freeze the first two blocks of InceptionV3 and reduce the learning rate[8].

MobileNetV2 is a general purpose computer vision neural networks designed with mobile devices in mind to support classification, detection and more. It is released by google in 2018 with improvements over MobileNetV1 and optimized for mobile visual recognition including classification, object detection and semantic segmentation[9]. I will be interesting to see how it will work in CIFAR-100. In MobileNetV2, I just added one fully-connected layer as the top layer

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$28^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times k$	conv2d 1x1	-	k	-	

Figure 4: MobileNetV2 structure

to output 100 categories. The complete MobileNetV2 structure is shown in figure 4. Each line describes a sequence of 1 or more identical (modulo stride) layers, repeated n times. All layers in the same sequence have the same number c of output channels. The first layer of each sequence has a stride s and all others use stride 1. All spatial convolutions use 3×3 kernels. The expansion factor t is always applied to the input size.

Benchmark

The benchmark model will be a simple CNN consist 3 fully-connected layers and a dense layer that shown in the figure.2. This will be relatively simple model to train and probably will get very low score. But it will still be more accurate than most supervised machine learning

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_98 (Conv2D)           (None, 224, 224, 16)      208
_________________________________________________________________
max_pooling2d_8 (MaxPooling2 (None, 112, 112, 16)      0
_________________________________________________________________
conv2d_99 (Conv2D)           (None, 112, 112, 32)      2080
_________________________________________________________________
max_pooling2d_9 (MaxPooling2 (None, 56, 56, 32)        0
_________________________________________________________________
conv2d_100 (Conv2D)          (None, 56, 56, 64)        8256
_________________________________________________________________
max_pooling2d_10 (MaxPooling (None, 28, 28, 64)        0
_________________________________________________________________
global_average_pooling2d_3 ( (None, 64)                0
_________________________________________________________________
dense_4 (Dense)              (None, 100)               6500
=================================================================
Total params: 17,044
Trainable params: 17,044
Non-trainable params: 0
_________________________________________________________________
```

Figure 5: Simple CNN model structure

algorithms. Using this benchmark, you can more clearly compare the advantages of transfer learning in terms of operational efficiency and results relative to CNN.

III. Methodology

Data Preprocessing

When using TensorFlow as backend, Keras CNNs require a 4D array (which we'll also refer to as a 4D tensor) as input, with shape $(nb\_samples, rows, columns, channels)$ where nb_samples corresponds to the total number of images (or samples), and rows, columns, and channels correspond to the number of rows, columns, and channels for each image, respectively. The path_to_tensor function above takes a string-valued file path to a colour image as input and returns a 4D tensor suitable for supplying to a Keras CNN. The paths_to_tensor function takes a numpy array of string-valued image paths as input and returns a 4D tensor with shape $(nb\_sample$,224,224,3). Because most of the neural network cannot process small picture, we have scaled all of our pictures from 32×32 to 224×224 to facilitate data processing.

```
from keras.preprocessing import image
from tqdm import tqdm

def path_to_tensor(img_path):
    # loads RGB image as PIL.Image.Image type
    img = image.load_img(img_path, target_size=(224, 224))
    # convert PIL.Image.Image type to 3D tensor with shape (224, 224, 3)
    x = image.img_to_array(img)
    # convert 3D tensor to 4D tensor with shape (1, 224, 224, 3) and return 4D tensor
    return np.expand_dims(x, axis=0)

def paths_to_tensor(img_paths):
    list_of_tensors = [path_to_tensor(img_path) for img_path in tqdm(img_paths)]
    return np.vstack(list_of_tensors)
```

Figure 6: Code that convert image to tensor

```python
def generate(batch, train_tensors, train_targets, valid_tensors, valid_targets):
    #  Using the data Augmentation in traning data
    datagen1 = ImageDataGenerator(
        shear_range=0.2,
        rotation_range=90,
        width_shift_range=0.2,
        height_shift_range=0.2,
        horizontal_flip=True)

    datagen2 = ImageDataGenerator(rotation_range=90)

    train_generator = datagen1.flow(
        train_tensors,
        train_targets,
        batch_size=batch)

    validation_generator = datagen2.flow(
        valid_tensors,
        valid_targets,
        batch_size=batch)

    count1 = len(train_tensors)
    count2 = len(valid_tensors)
    return train_generator, validation_generator, count1, count2
```

Figure 7: Image augmentation configuration

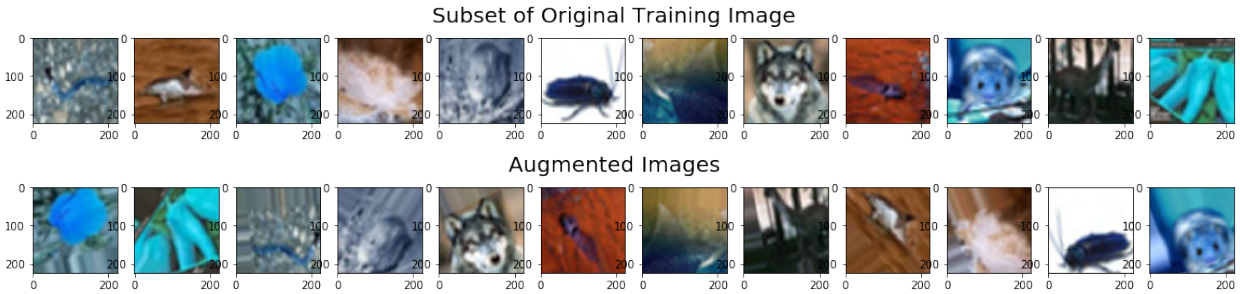

Figure 8: Original training images and augmented images

After the image is converted to tensor, we can use ImageDataGenerator for image augmentation. In this project, we rotate, scale, cut and flip the image, and use image flipping for the validation data. The image before and after conversion is shown in the figure below. However, in practice, since image augmentation will greatly reduce the training efficiency, this method is not used in real training process.

Implementation

InceptionV3

Because the Keras already includes InceptionV3 and the trained ImageNet weights, all we need to do is introduce the model and replace fully-connected layer. In pre-training, all layers except the fully-connected layer need to be frozen. In fine tuning, we only need to freeze the first 249 layers (2 blocks). Also, the optimizer and learning rate used in both trainings are different. In fine tuning we want to avoid overfitting with a lower learning rate. The code looks like figure 9.

```
def Inception(input_shape, k):
    # create the base pre-trained model
    base_model = InceptionV3(input_shape=input_shape,
                             weights='imagenet', include_top=False)

    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    x = Dense(1024, activation='relu')(x)
    x = Dropout(0.3)(x)
    # and a logistic layer -- let's say we have 100 classes(k)
    predictions = Dense(k, activation='softmax')(x)

    model = Model(inputs=base_model.input, outputs=predictions)

    for layer in base_model.layers:
        layer.trainable = False

    return model


def fine_tune(model):
    model.load_weights('saved_models/weights.best.inception.hdf5')

    for layer in model.layers[:249]:
        layer.trainable = False
    for layer in model.layers[249:]:
        layer.trainable = True

    return model


def train(batch, epochs, num_classes, size, weights, train_tensors, train_targets, valid_tensors, valid_targets):

    if weights:
        model = Inception((size, size, 3), num_classes)
        model = fine_tune(model)
        model.compile(optimizer=SGD(lr=0.0001, momentum=0.9), loss='categorical_crossentropy')
    else:
        model = Inception((size, size, 3), num_classes)
        model.compile(optimizer='rmsprop', loss='categorical_crossentropy')


    earlystop = EarlyStopping(monitor='val_loss', patience=15, verbose=1, mode='auto')
    checkpointer = ModelCheckpoint(filepath='saved_models/weights.best.inception.hdf5',
                                   verbose=1, save_best_only=True)

    model.fit(train_tensors, train_targets,
            validation_data=(valid_tensors, valid_targets),
            epochs=epochs, verbose=2, batch_size=batch, callbacks=[earlystop, checkpointer])
```

Figure 9: InceptionV3 implementation

```python
def _conv_block(inputs, filters, kernel, strides):
    """Convolution Block
    This function defines a 2D convolution operation with BN and relu6.
    # Arguments
        inputs: Tensor, input tensor of conv layer.
        filters: Integer, the dimensionality of the output space.
        kernel: An integer or tuple/list of 2 integers, specifying the
            width and height of the 2D convolution window.
        strides: An integer or tuple/list of 2 integers,
            specifying the strides of the convolution along the width and height.
            Can be a single integer to specify the same value for
            all spatial dimensions.
    """

    channel_axis = 1 if K.image_data_format() == 'channels_first' else -1

    x = Conv2D(filters, kernel, padding='same', strides=strides)(inputs)
    x = BatchNormalization(axis=channel_axis)(x)
    return Activation(relu6)(x)


def _bottleneck(inputs, filters, kernel, t, s, r=False):
    """Bottleneck
    # Arguments
        inputs: Tensor, input tensor of conv layer.
        filters: Integer, the dimensionality of the output space.
        kernel: An integer or tuple/list of 2 integers, specifying the
            width and height of the 2D convolution window.
        t: Integer, expansion factor.
            t is always applied to the input size.
        s: An integer or tuple/list of 2 integers,specifying the strides
            of the convolution along the width and height.Can be a single
            integer to specify the same value for all spatial dimensions.
        r: Boolean, Whether to use the residuals.
    """

    channel_axis = 1 if K.image_data_format() == 'channels_first' else -1
    tchannel = K.int_shape(inputs)[channel_axis] * t

    x = _conv_block(inputs, tchannel, (1, 1), (1, 1))

    x = DepthwiseConv2D(kernel, strides=(s, s), depth_multiplier=1, padding='same')(x)
    x = BatchNormalization(axis=channel_axis)(x)
    x = Activation(relu6)(x)

    x = Conv2D(filters, (1, 1), strides=(1, 1), padding='same')(x)
    x = BatchNormalization(axis=channel_axis)(x)

    if r:
        x = add([x, inputs])
    return x


def _inverted_residual_block(inputs, filters, kernel, t, strides, n):

    x = _bottleneck(inputs, filters, kernel, t, strides)

    for i in range(1, n):
        x = _bottleneck(x, filters, kernel, t, 1, True)

    return x


def MobileNetv2(input_shape, k):

    inputs = Input(shape=input_shape)
    x = _conv_block(inputs, 32, (3, 3), strides=(2, 2))

    x = _inverted_residual_block(x, 16, (3, 3), t=1, strides=1, n=1)
    x = _inverted_residual_block(x, 24, (3, 3), t=6, strides=2, n=2)
    x = _inverted_residual_block(x, 32, (3, 3), t=6, strides=2, n=3)
    x = _inverted_residual_block(x, 64, (3, 3), t=6, strides=2, n=4)
    x = _inverted_residual_block(x, 96, (3, 3), t=6, strides=1, n=3)
    x = _inverted_residual_block(x, 160, (3, 3), t=6, strides=2, n=3)
    x = _inverted_residual_block(x, 320, (3, 3), t=6, strides=1, n=1)

    x = _conv_block(x, 1280, (1, 1), strides=(1, 1))
    x = GlobalAveragePooling2D()(x)
    x = Reshape((1, 1, 1280))(x)
    x = Dropout(0.3, name='Dropout')(x)
    x = Conv2D(k, (1, 1), padding='same')(x)

    x = Activation('softmax', name='softmax')(x)
    output = Reshape((k,))(x)

    model = Model(inputs, output)
    plot_model(model, to_file='images/MobileNetv2.png', show_shapes=True)

    return model
```

Figure 10: MobileNetV2 implementation

MobileNetV2

The implementation of the model is mainly based on given structure in the paper which also been shown in figure 4. For coding detail see figure 10. Same as InceptionV3, earlyStopping and checkpoint is used to save time and prevent model from over-fitting.

During implementing the model from scratch by reading the paper, I felt difficult to find a way to properly achieve the architecture though I have a understanding of its structure. Error mostly occurs in wrong input parameters and output of functions. I think that exposed my weakness in lack of strong basic knowledges in coding in python and related packages.

Refinement

Tuning the learning rate turned out to have the most significant impact on the result of the project. If the learning rate is low, then training is more reliable, but optimization will take a lot of time because steps towards the minimum of the loss function are tiny. If the learning rate is high, then training may not converge or even diverge. Weight changes can be so big that the optimizer overshoots the minimum and makes the loss worse. By adjusting learning rate from 0.01 to 0.0001 in SGD optimizer in InceptionV3 in fine-tuning stage, the accuracy was raised from 46.01% to 49.54%.

InceptionV3 Optimizer Configuration and Accuracy

Stage	Optimizer	Learning Rate	Other params	Accuracy
Pre-train	rmsprop	0.001(Default)	N/A	N/A
Fine-tuning	SGD	0.01(Default)	N/A	46.01%
Fine-tuning	SGD	0.001	Momentum=0.9	47.90%
Fine-tuning	SGD	0.0001	Momentum=0.9	49.54%

However, in MobileNetV2, we observed that using the default learning rate not only achieves the best training results but also saves time. Decreasing the learning rate will lead to a decline in the final accuracy rate. One of the possible reason is because we start to train the model with randomly initialized parameters with a relatively small size of image data. That may cause each time we train, the result is different. When we can train with more consistent initialization weights, it would be easier to analysis the reason behind.

MobileNetV2 Optimizer Configuration and Accuracy

Stage	Optimizer	Learning Rate	Other params	Accuracy
Pre-train	Adam	0.001(Default)	N/A	N/A
Fine-tuning	Adam	0.001(Default)	N/A	50.49%
Fine-tuning	Adam	1E-04	N/A	48.29%
Fine-tuning	Adam	1E-05	N/A	46.60%

IV. Results

Model Evaluation and Validation(N)

Training InceptionV3 and MobileNetV2 took me about 8 hours each on AWS p2.xlarge instance. For efficiency, I have to limit training dataset to half of the whole dataset and train model with a relatively medium rate, otherwise it will cost too much time to train. I trained all these models several times with different optimizer and learning rate.

	Benchmark CNN	InceptionV3	MobileNetV2
Accuracy	15.3%	49.54%	50.49%
Time per epoch	40s	217s	360s

Considering about the training efficiency and final result, 1e-4 is chosen to be the learning rate of a SGD optimizer for InceptionV3 and default configuration is used for Adam optimizer in MobileNetV2. The final accuracy on CIFAR-100 test is shown below, the accuracy is based on the best result I got in the training.

Justification

From the training results, we can see that the two models finally achieved an accuracy of about 50%, which is 3-4 times better than of benchmark CNN, and the difference is obvious. It is worth noting that all three models use the normal size of the original data set for training, so if you use more training data and use image preprocessing, we should be able to achieve better results.

After MobileNetV2 reaches the training bottleneck, the loss value will soon start to be discrete, so it is necessary to further reduce the learning rate. But because of time, I did not make further optimizations to the model.

Since the image was exported and resized, some images are distorted (not sure whether it is the cause of cv2), so there should be errors when processing the actual real world image.

V. Conclusion

Free-Form Visualization

Figure 11 shows the confusion matrix of InceptionV3 and MobileNetV2. It would be too small to see clear in this report, you can check out the file in `saved_models/` directory. From the matrix we can see that certain class has better performance compare to others. For example, class 36(mice) and 58(mountain). Also, some class tends to be more easy be recognized as another class. For example, class 42(leopard) and 48, class 55(shrew) and 48. I guess the reason behind

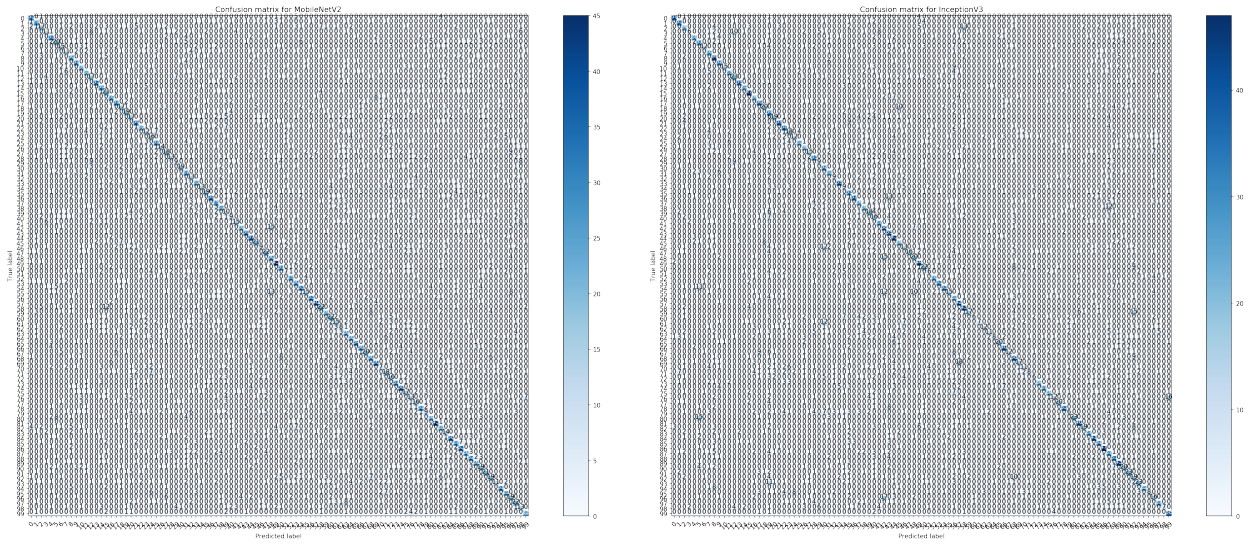

Figure 11: Confusion matrix for two models.

might because the two classes share the same pixel feature. After we now which classes are easy to cause confusion, we can train our model purposely based on the data we have.

Reflection

I think this is a very challenging project. From the initial selection of research directions, the selection of data sets, the cleaning of data, the development of models and training and debugging all require a lot of research. Because I am not only a beginner in machine learning, but also a beginner in Python, there are many basic knowledge in this process that I need to learn. I think there are two most difficult points in the whole project. One is read the research papers and build MobileNetV2 by hand, and the other is to choose the right optimizer and parameters to train. These two processes cost me a lot of time to read and debug the model. Due to the computational power limitation, I can't use whole datasets for training, but still achieved a 50% accuracy based on 0.5 datasets. The training efficiency of these two models surprised me

because I was able to reach the training bottleneck in a short time. I am looking forward to using this model in other image recognition applications.

During this project, I learnt how to properly build a neural network from scratch, learnt some basic data visualization skills using sklearn also data manipulation using numpy and python packages. Most importantly, I had my first complete experience in machine learning area to raised a problem, research the paper and online resource to come up with a plan. During the process, I managed to investigate different ways to achieve my goal and improve it. Surly there are lot to learn and I will keep evolve myself.

Improvement

Due to computational constraints, I only used 50% of the dataset for training and did not apply image augmentation. So using more training data and using image augmentation is an improvement. In addition, whether using InceptionV3 or MobileNetV2, the training quickly reached the bottleneck, val_loss and val_acc no longer improved, and the model gradually over-fitting. Choosing and using more appropriate optimizers and parameters, reducing the learning rate should be able to alleviate this situation to some extent. According to the actual training experience, adjusting the top layer is not effective in improving the accuracy, so this method still needs to continue to explore.

References

[1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.

[2] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. arXiv preprint arXiv:1503.03832, 2015.

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 1–9, 2015.

[4] Szegedy, Christian, et al. "Going deeper with convolutions." Proceedings of the IEEE conference on computer vision and pattern recognition. 2015.

[5] https://keras.io/applications/

[6] https://keras.io/datasets/#cifar100-small-image-classification

[7] https://www.cs.toronto.edu/~kriz/cifar.html

[8] Inception modules: explained and implemented. Retrieved from https://hacktilldawn.com/2016/09/25/inception-modules-explained-and-implemented/

[9] MobileNetV2: The Next Generation of On-Device Computer Vision Networks. Retrieved from https://ai.googleblog.com/2018/04/mobilenetv2-next-generation-of-on.html

[10] Szegedy, Christian, et al. "Rethinking the inception architecture for computer vision. " *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.

[11] Sandler, Mark, et al. "Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation." arXiv preprint arXiv:1801.04381 (2018).