Introduction to Machine Learning Final Project

State Farm Distracted Driver Detection

Chao-Lin Chen

※The training process is in VGG16.ipynb, VGG16.html, ResNet34.ipynb, ResNet34.html, InceptionV1.ipynb, InceptionV1.html, the pdf report discusses the results and evaluates the performance.

Dataset Description

Link: https://www.kaggle.com/c/state-farm-distracted-driver-detection

We've all been there: a light turns green and the car in front of you doesn't budge. Or, a previously unremarkable vehicle suddenly slows and starts swerving from side-to-side.

When you pass the offending driver, what do you expect to see? You certainly aren't surprised when you spot a driver who is texting, seemingly enraptured by social media, or in a lively hand-held conversation on their phone.

According to the CDC motor vehicle safety division, one in five car accidents is caused by a distracted driver. Sadly, this translates to 425,000 people injured and 3,000 people killed by distracted driving every year.

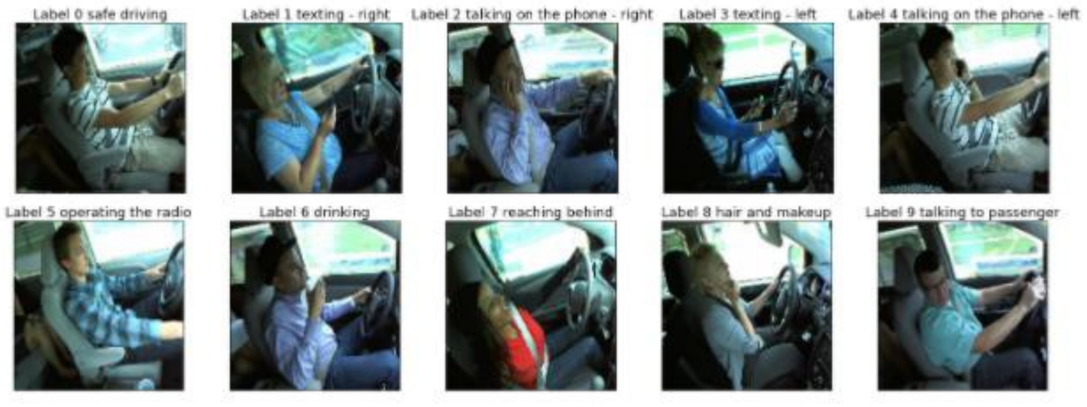
State Farm hopes to improve these alarming statistics, and better insure their customers, by testing whether dashboard cameras can automatically detect drivers engaging in distracted behaviors. Given a dataset of 2D dashboard camera images, State Farm is challenging Kagglers to classify each driver's behavior. Are they driving attentively, wearing their seatbelt, or taking a selfie with their friends in the backseat?

Overview

Our project will analyze this dataset through three different classical CNN architectures. The relevant structure will contain the following information:

Dataset process and Hyperparameter Selection

The dataset is divided into 80% of the training set and 20% of the testing set through data preprocessing, and the label use one-hot encoding. Then, show the number of labels to determine whether the dataset has an even distribution.



```
X_train shape: (17939, 150, 150, 3)
y_train shape: (17939, 10)
X_test shape: (4485, 150, 150, 3)
y_test shape: (4485, 10)

There are  [0 1 2 3 4 5 6 7 8 9] different labels in this dataset.
Labels count in y_train: [1979 1815 1860 1880 1853 1870 1858 1588 1539 1697]
Labels count in y_test: [510 452 457 466 473 442 467 414 372 432]
```

When training different CNN models, use model.fit to divide the training set into 80% and 20% of training set and validation set, and use the validation set to test whether the model is underfitting or overfitting. Then, set parameter patience in Earlystopping to 3 in order to stop training to prevent validation loss after 3 consecutive epochs. Finally, plot the corresponding accuracy and loss diagram for analysis.

```
early_stop  = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=3)
checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=1, save_best_only=True, mode='min')
```

Utilize VGG16_best_weight, ResNet34_best_weight, and InceptionV1_best_weight to get the model with the lowest validation loss under different parameters through ModelCheckpoint. Then, use the testing set to evaluate its performance, and implement the confusion matrix to analyze the result.

```
cm=confusion_matrix(y_test,y_pred)
ac=accuracy_score(y_test,y_pred)
print("Accuracy", ac)
print("Confusion matrix")
plot_confusion_matrix(cm,figsize=(9,9),colorbar=True)
```

For training the CNN models, we set the hyperparameter learning rate(lr) and batch size (batch_size) to 0.0001, 0.0005, 0.001 and 16, 32, 48, respectively, for a total of 9 combinations to evaluate the performance of the model.

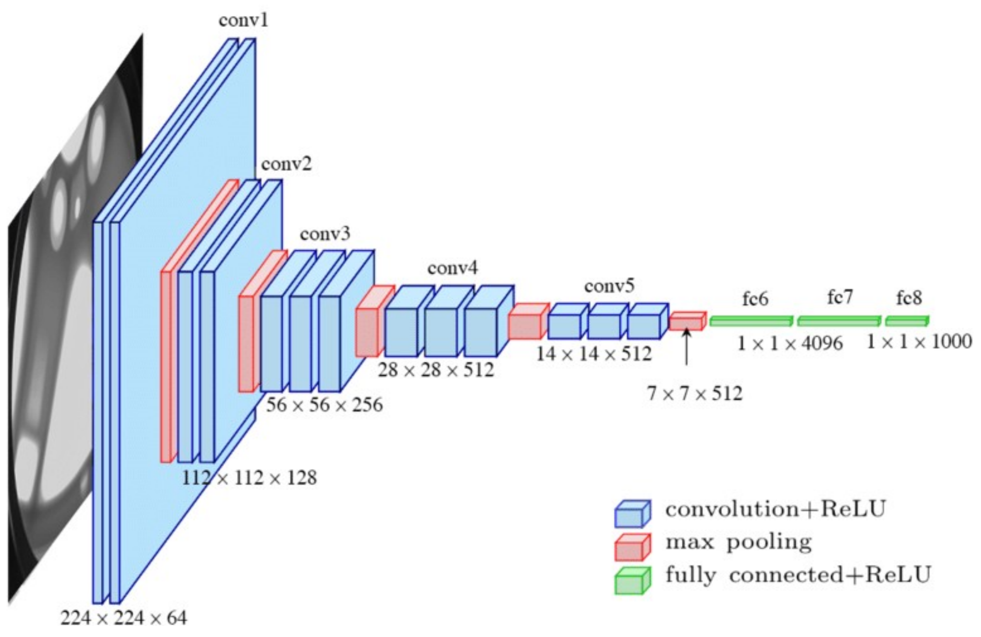
Model

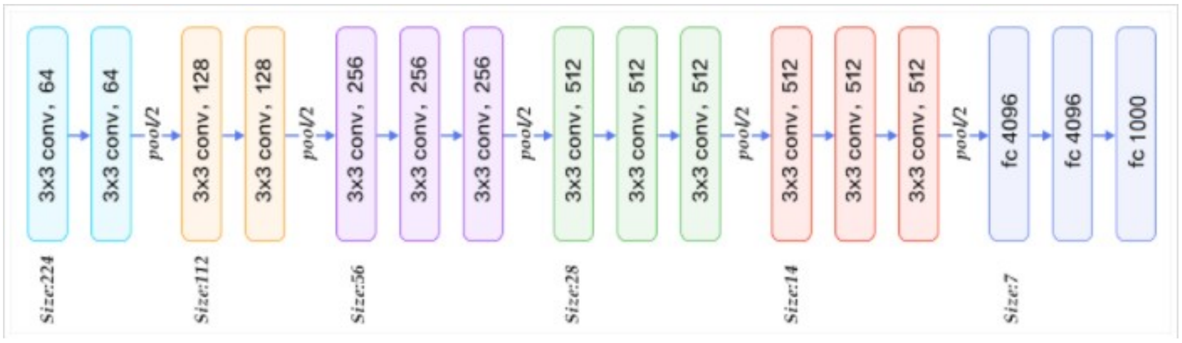
VGG16

1. Introduction

Compared with the traditional CNN architectures LeNet and AlexNet, VGGNet uses a deeper network, which is characterized by repeatedly using the same set of modules, consisting of small kernels instead of medium or large-size kernel in AlexNet. Its architecture consists of n VGG Blocks and 3 fully connected layers.

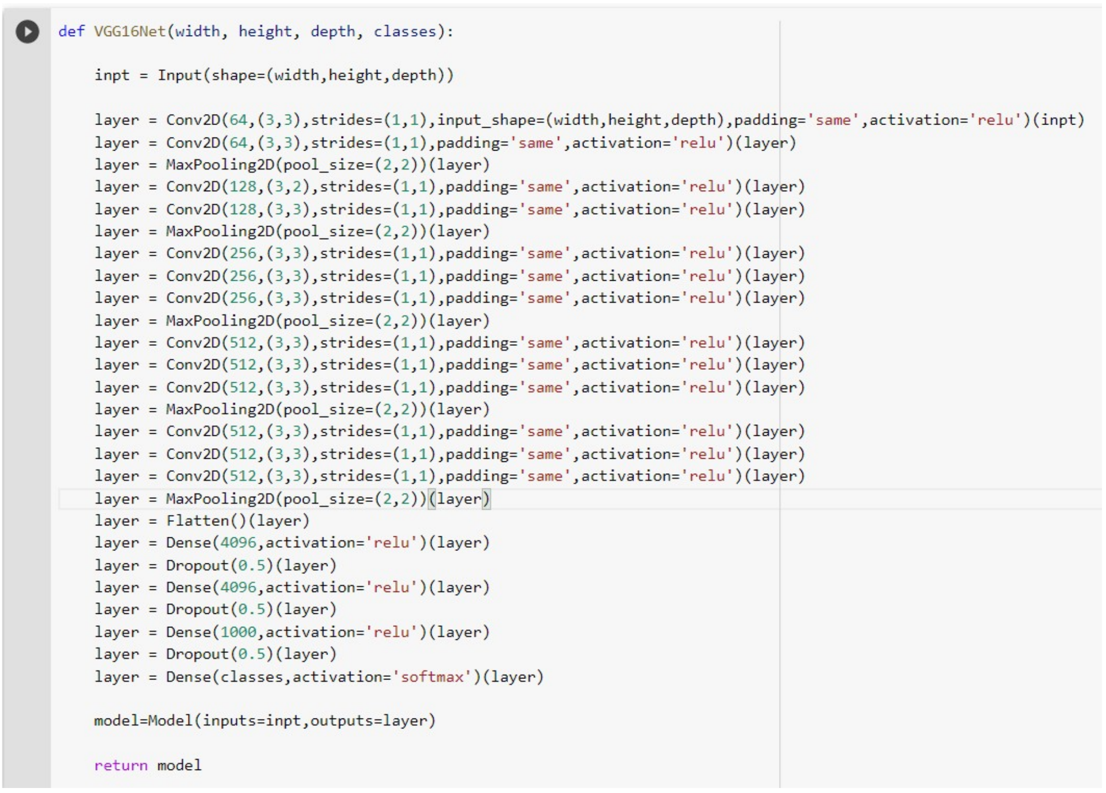
The construction of VGG Block is composed of different numbers of 3x3 convolution layers (kernel size=3x3, stride=1, padding=”same”), and 2x2 Maxpooling (pool size=2, stride=2). VGG16 replaces AlexNet's medium-size 5x5 kernels with multiple small 3x3 kernels not only achieving the same receptive field, avoiding losing edge information but also reducing the number of parameters with the same nonlinear dimension.





VGG16 module:

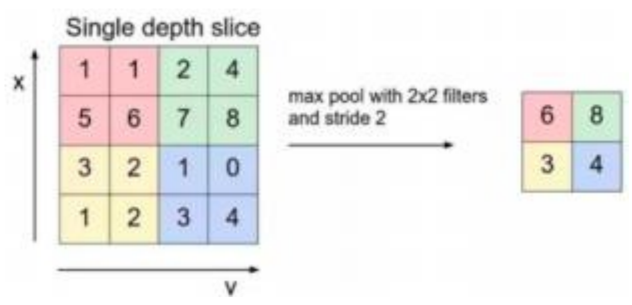
The implementation of VGG16 utilizes Convolution2D, MaxPooling2D, Flatten, and Dense in keras.layers. The model of VGG16 is constructed by the above structure, from input data declaration to multiple layers, and various kernel sizes and channels to structure the multi-layer structure.

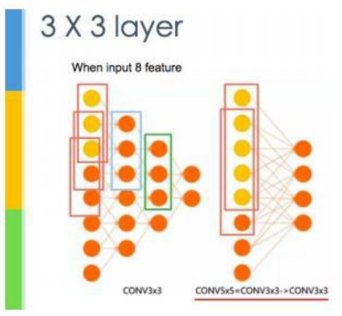


```
def VGG16Net(width, height, depth, classes):

    inpt = Input(shape=(width,height,depth))

    layer = Conv2D(64,(3,3),strides=(1,1),input_shape=(width,height,depth),padding='same',activation='relu')(inpt)
    layer = Conv2D(64,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = MaxPooling2D(pool_size=(2,2))(layer)
    layer = Conv2D(128,(3,2),strides=(1,1),padding='same',activation='relu')(layer)
    layer = Conv2D(128,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = MaxPooling2D(pool_size=(2,2))(layer)
    layer = Conv2D(256,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = Conv2D(256,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = Conv2D(256,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = MaxPooling2D(pool_size=(2,2))(layer)
    layer = Conv2D(512,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = Conv2D(512,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = Conv2D(512,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = MaxPooling2D(pool_size=(2,2))(layer)
    layer = Conv2D(512,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = Conv2D(512,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = Conv2D(512,(3,3),strides=(1,1),padding='same',activation='relu')(layer)
    layer = MaxPooling2D(pool_size=(2,2))(layer)
    layer = Flatten()(layer)
    layer = Dense(4096,activation='relu')(layer)
    layer = Dropout(0.5)(layer)
    layer = Dense(4096,activation='relu')(layer)
    layer = Dropout(0.5)(layer)
    layer = Dense(1000,activation='relu')(layer)
    layer = Dropout(0.5)(layer)
    layer = Dense(classes,activation='softmax')(layer)

    model=Model(inputs=inpt,outputs=layer)

    return model
```

During training, VGG16 will first train an initial model and keep updating the weights to lead to convergence. As the layer goes deeper, the number of filters will gradually increase to 512.

The figure below shows that two 3x3 filters can be equivalent to a 5x5 filter, and the final output of both filters is 4. In addition, VGG16 also changed the pooling layer from 3x3 to 2x2 to get more feature.



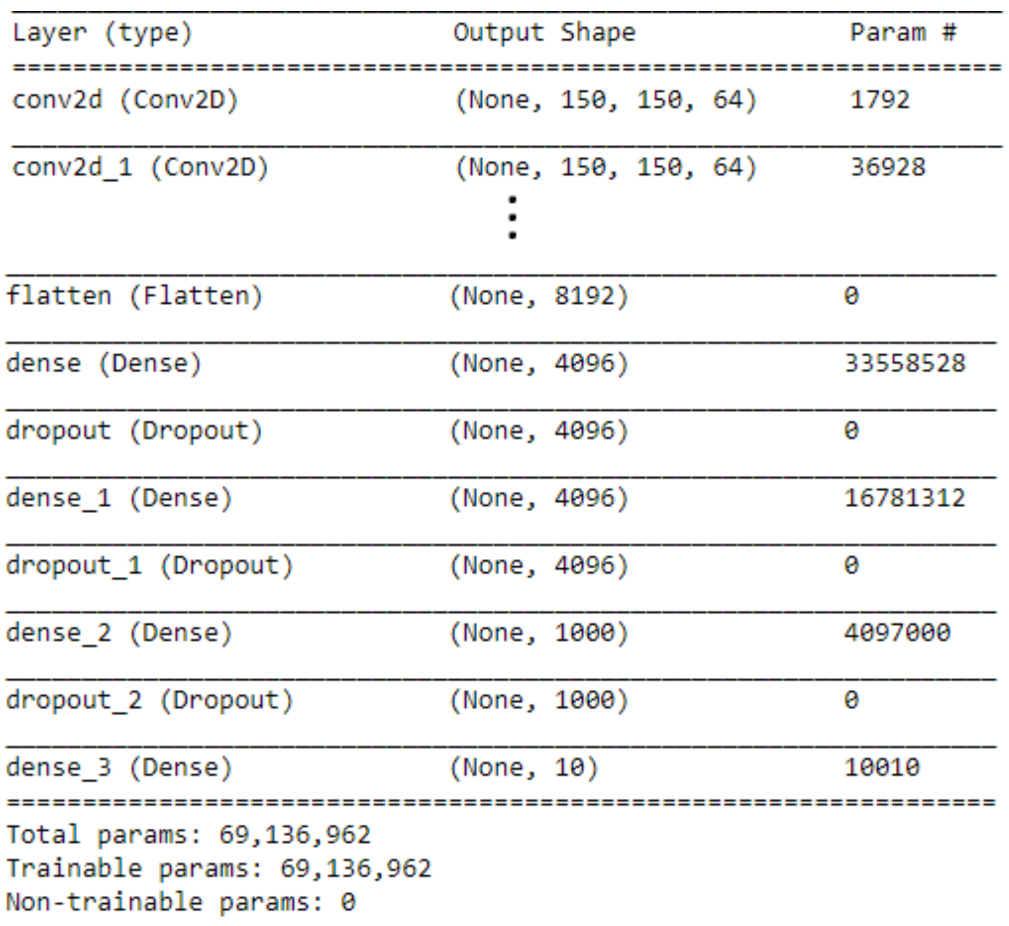


Advantages and disadvantages of VGG16

Advantages:
1. The structure is simpler
2. The small size filter is ideal for locally concentrated images

Disadvantages:
1. A large number of weights
2. The training time and resources are expensive and the parameters are not easy to adjust
3. Required larger memory resource

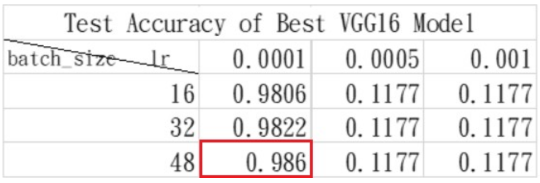
The following is the VGG16 architecture and the number of hyperparameter:



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 150, 150, 64)      1792
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 150, 150, 64)      36928
                                  ⋮
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 4096)              33558528
_________________________________________________________________
dropout (Dropout)            (None, 4096)              0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_1 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 1000)              4097000
_________________________________________________________________
dropout_2 (Dropout)          (None, 1000)              0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                10010
=================================================================
Total params: 69,136,962
Trainable params: 69,136,962
Non-trainable params: 0
```

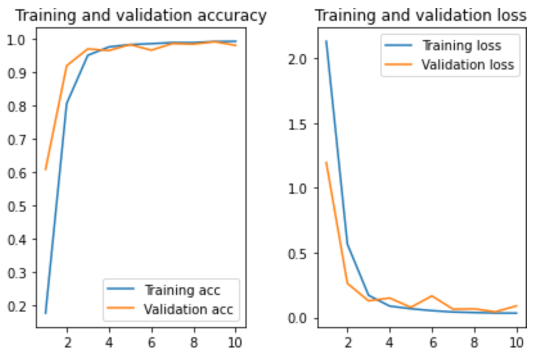
2. Evaluation

The following figure shows the accuracy of the testing set obtained by VGG16 under different parameters. It can be clearly seen from the figure that when the learning rate=0.0001 and the batch size=48, the accuracy is the highest at 0.986, and when the learning rate=0.005 or 0.001, no matter what the batch size is, the testing accuracy is poor.



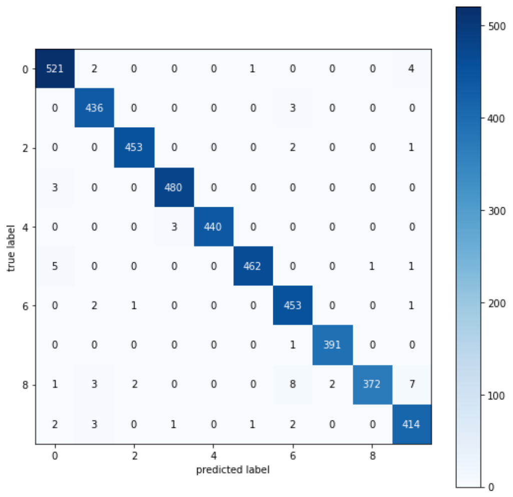
Test Accuracy of Best VGG16 Model			
batch_size \ lr	0.0001	0.0005	0.001
16	0.9806	0.1177	0.1177
32	0.9822	0.1177	0.1177
48	0.986	0.1177	0.1177

In addition, from the accuracy and loss plots of the training and validation set, when epoch=3, its performance reaches a certain level, and after epoch=3, the model is relatively stable without underfitting or overfitting.



3. Confusion Matrix

The Confusion Matrix shows that most of the testing datasets have been correctly classified. It can be roughly inferred that the parameters used by the model are reasonable and contain a certain degree of stability with the dataset.

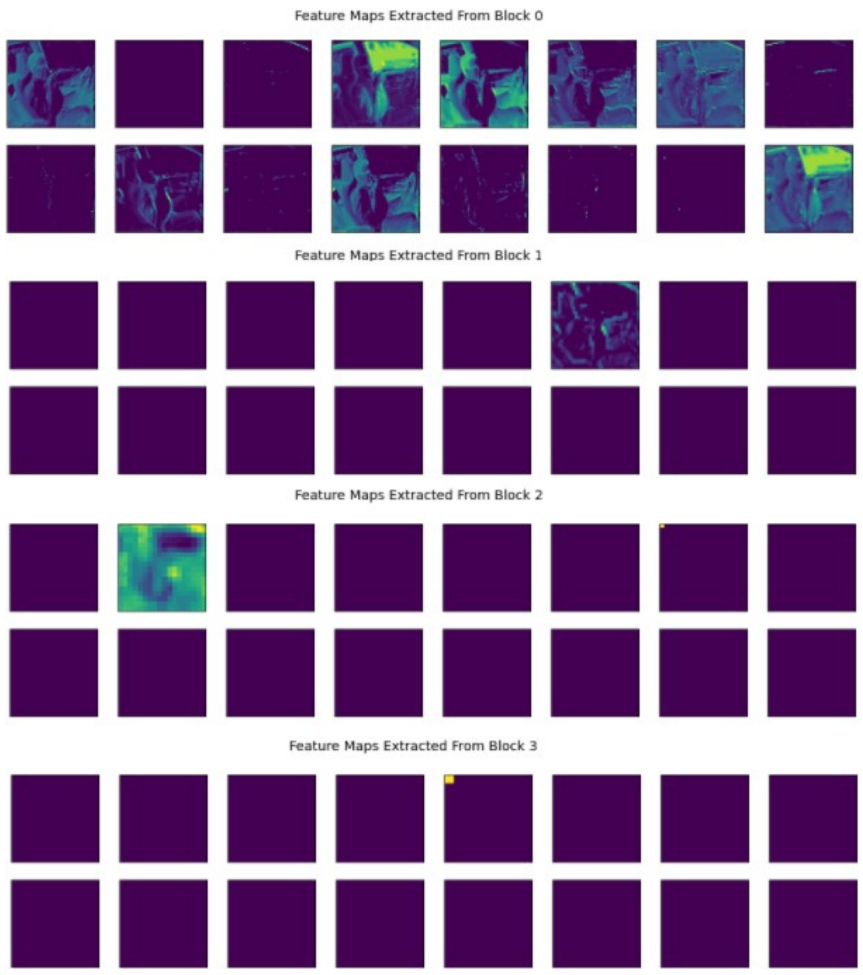


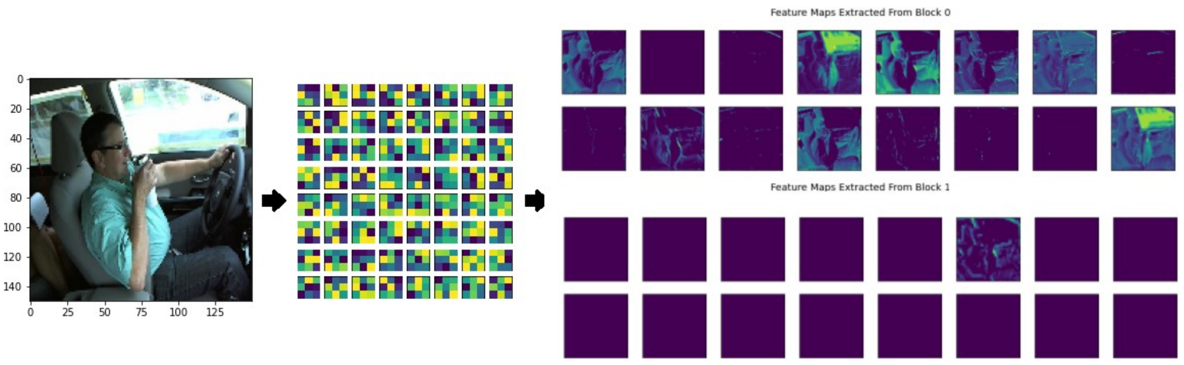
0	521	2	0	0	0	1	0	0	0	4
	0	436	0	0	0	0	3	0	0	0
2	0	0	453	0	0	0	2	0	0	1
	3	0	0	480	0	0	0	0	0	0
4	0	0	0	3	440	0	0	0	0	0
	5	0	0	0	0	462	0	0	1	1
6	0	2	1	0	0	0	453	0	0	1
	0	0	0	0	0	0	1	391	0	0
8	1	3	2	0	0	0	8	2	372	7
	2	3	0	1	0	1	2	0	0	414
	0		2		4		6		8	

4. Filters & Feature Maps Visualization

The following photos are the filter of the VGG16 testing set (partially shown):

The picture will convolute with the filter to get the corresponding feature map, and convert from RGB to BGR to present a blue image. In addition, for the deeper filter, the obtained feature map will be more blurred. It can be seen that the shallower filter will initially identify sharper features such as edges, then determine the smoother part.

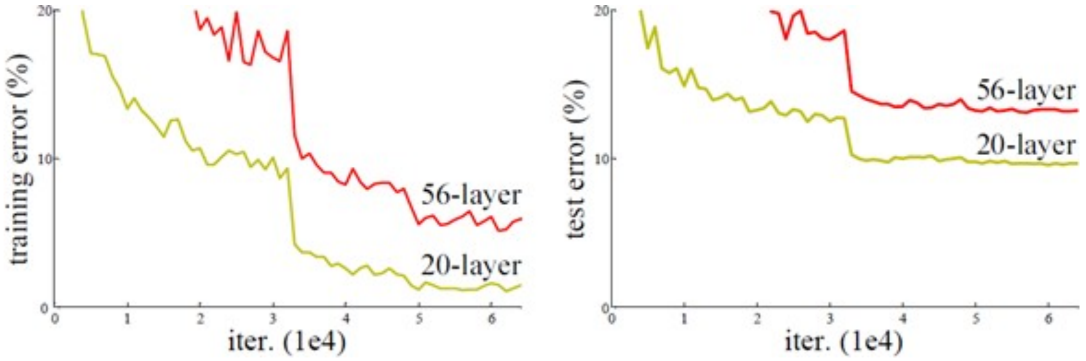

Feature Maps Extracted From Block 0
Feature Maps Extracted From Block 1
Feature Maps Extracted From Block 2
Feature Maps Extracted From Block 3


0
20
40
60
80
100
120
140
0
25
50
75
100
125
Feature Maps Extracted From Block 0
Feature Maps Extracted From Block 1

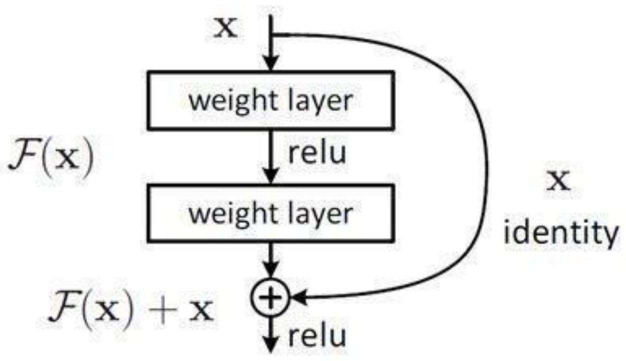
ResNet34

1. Introduction

From experience, the depth of the model is critical to its performance. When the layers of the model increase, the model can extract more complex feature patterns, so theoretically better results can be achieved when the model has a deeper structure. However, it is found that when the model depth increases, the model accuracy will saturate or even decrease. This trend can be seen in the following figure: the model with 56 layers contains worse performance than 20 layers model. This would not be an overfitting issue, since the training error of the 56-layer model is also high. We know that deep models have the problem of vanishing or exploding gradients, which makes deep learning models difficult to train. But now there are other methods such as Batch normalization to alleviate this problem. Therefore, it is very surprising that there is a degradation problem of deep models.

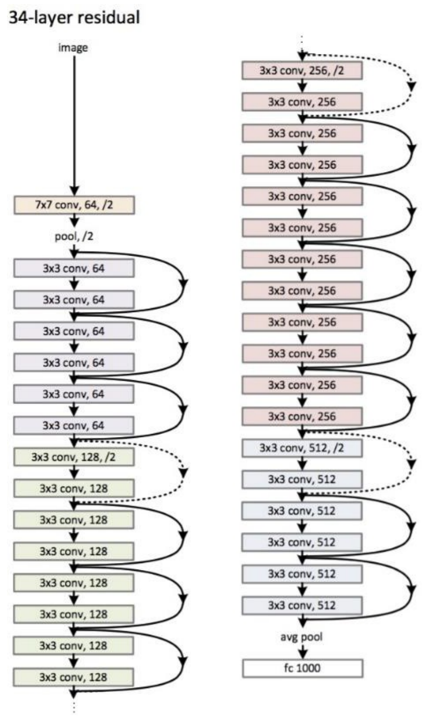


The original network takes x as input and expects an output H(x). Now we change it, we make H(x)=F(x)+x, then our network only needs to learn to output a residual F(x)=H(x)-x. The authors propose that learning the residuals F(x)=H(x)-x is much simpler than directly learning the original features H(x). Intuitively, residual learning requires less learning, because the residual is generally small, and when F(x)=0, H(x)= x, this is identity mapping. Through this principle, we can know that the performance of the deep layer should be at least the same as the previous layer. There should be no degradation, so the residual network becomes training F(x)=H(x) - x, and the goal is to make F(x) close to 0. The network can be deepened and the accuracy rate will not decrease.

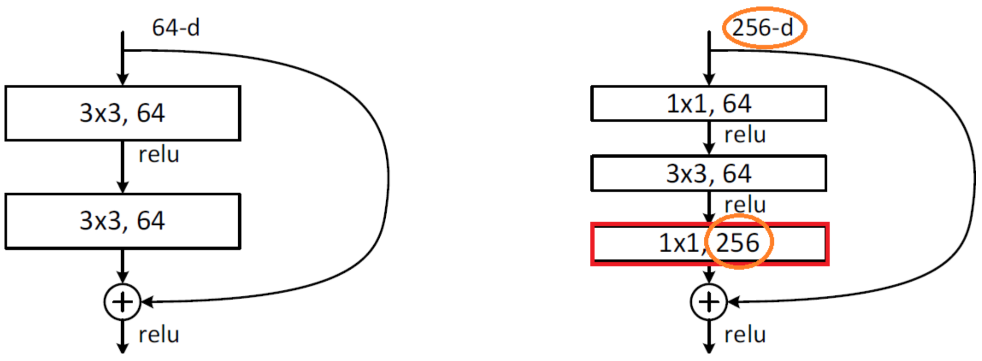

x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

The following figure shows the network architecture of ResNet34 with different layers. We can see that the first layer is a 7x7 convolution layer, connected to 3x3 maxpooling, then a large number of Residual Blocks are used; finally, Global Average Pooling is used and passed to the fully connected layer for classification. For the below diagram, the dotted line part means that in implementing a shortcut connection, the input x and the weight output F(x) channel (channel) have a different number; therefore, you need to use 1x1 convolution layer to adjust the channel dimension so that it can be added.

$$\text{Solid Line: } F\{x, \{W_i\} + x$$
$$\text{Dotted Line: } F\{x, \{W_i\} + W_s x$$



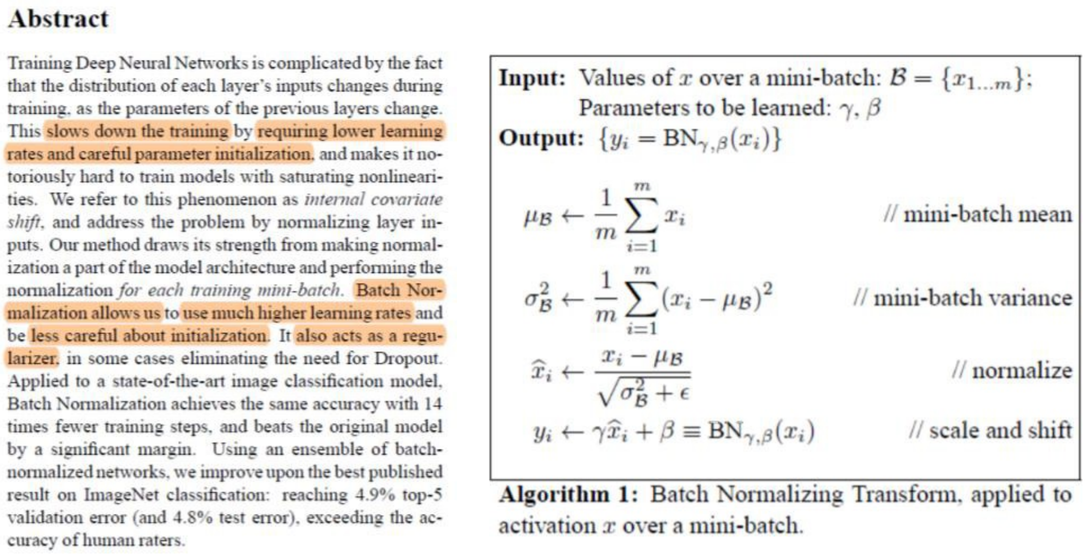
Through the observation of the above figure, it can be seen that when the dimensions are the same, they can be added directly. On the other hand, if they are different, the dimensions need to be increased by using zero padding. A new projection shortcut should be used to make the number of channels consistent. Generally, using a 1×1 convolution, although this will increase the training parameters and consume more computation resources, the impact is relatively low.



Batch Normalization:

In a deep neural model, the input of a specific layer is the output of its previous layer. Therefore, if the parameters of the previous layer change, the distribution of its input will greatly differ.

When training a model using stochastic gradient descent, each update of a parameter will lead to a change in the distribution of inputs for each layer. Basically, as the layer goes deeper, the distribution of its input changes more significantly, just like a small offset on the base floor of a building will cause a relatively large offset on the upper floors. This phenomenon is also known as Internal Covariate Shift.



Abstract

Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate shift*, and address the problem by normalizing layer inputs. Our method draws its strength from making normalization a part of the model architecture and performing the normalization *for each training mini-batch*. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout. Applied to a state-of-the-art image classification model, Batch Normalization achieves the same accuracy with 14 times fewer training steps, and beats the original model by a significant margin. Using an ensemble of batch-normalized networks, we improve upon the best published result on ImageNet classification: reaching 4.9% top-5 validation error (and 4.8% test error), exceeding the accuracy of human raters.

```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma, \beta$
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$     // mini-batch variance
$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                 // normalize
$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i)$       // scale and shift
```

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

The idea of batch normalization is shown as follows:

Assuming that the batch is K, for a small training set B, each time a piece of data is sent into the model. For example, the number of k, in the ℓth layer, $Z_k^{(\ell)}$ can be calculated to obtain the average and standard deviation

$$\boldsymbol{\mu}_B = \frac{1}{K}\sum_{k=1}^{K}\mathbf{z}_k^{(\ell)}$$

$$\boldsymbol{\sigma}_B^2 = \frac{1}{K}\sum_{k=1}^{K}\left(\mathbf{z}_k^{(\ell)} - \boldsymbol{\mu}_B\right)\odot\left(\mathbf{z}_k^{(\ell)} - \boldsymbol{\mu}_B\right)$$

Then normalize $Z_k^{(\ell)}$ to normal distribution, where the constant Ɛ is used to avoid variance equal to zero.

$$\mathbf{z}_k^{(\ell)} \leftarrow \frac{\mathbf{z}_k^{(\ell)} - \boldsymbol{\mu}_B}{\sqrt{\boldsymbol{\sigma}_B^2 + \varepsilon}}$$

Finally, scaling and shift $Z_k^{(\ell)}$

$$\mathbf{z}_k^{(\ell)} \leftarrow \mathbf{z}_k^{(\ell)} \odot \gamma + \beta$$

Advantages of batch normalization:

1. Under certain circumstances where the convergence speed is slow, or the situation when the gradient cannot be trained, such as exploding gradients, batch normalization may be beneficial to improve the situation.
2. Speed up the training process and improve model performance.
3. A higher learning rate can be used to improve the speed of model training, and effectively avoid the disappearance and explosion of gradients.
4. It has the function of regularization, similar to the dropout function.
5. The effect of weight initialization is lower.

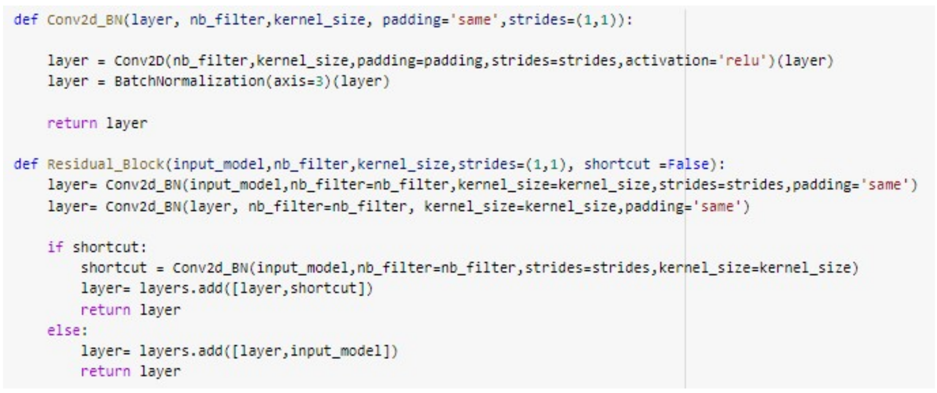
Resblock:

Conv2d_BN

Through the above introduction of Batch Normalization, a function is defined to do batch normalization after each convolution is done and sent to the next layer.

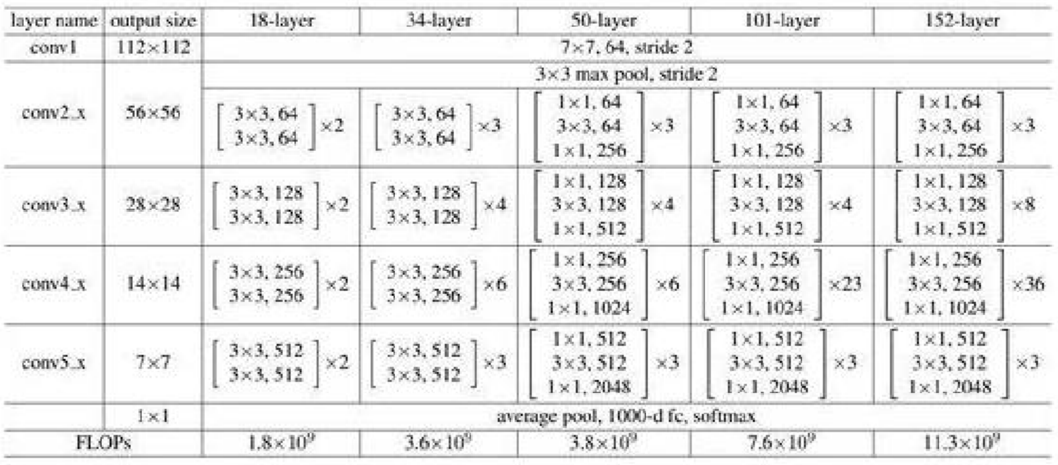
Residual_Block:

Do Conv2d_BN twice each time. If the dimensions of the next layer are different, shortcut connection is required, and make the original input model as a convolution to match the dimensions of the next layer.



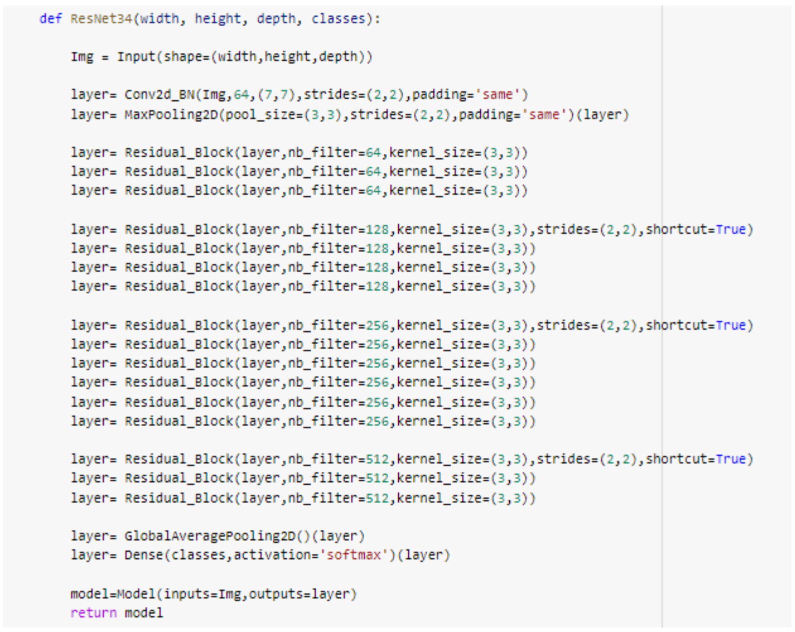
```
def Conv2d_BN(layer, nb_filter,kernel_size, padding='same',strides=(1,1)):

    layer = Conv2D(nb_filter,kernel_size,padding=padding,strides=strides,activation='relu')(layer)
    layer = BatchNormalization(axis=3)(layer)

    return layer

def Residual_Block(input_model,nb_filter,kernel_size,strides=(1,1), shortcut =False):
    layer= Conv2d_BN(input_model,nb_filter=nb_filter,kernel_size=kernel_size,strides=strides,padding='same')
    layer= Conv2d_BN(layer, nb_filter=nb_filter, kernel_size=kernel_size,padding='same')

    if shortcut:
        shortcut = Conv2d_BN(input_model,nb_filter=nb_filter,strides=strides,kernel_size=kernel_size)
        layer= layers.add([layer,shortcut])
        return layer
    else:
        layer= layers.add([layer,input_model])
        return layer
```

Resblock:



layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

The model is created through the above table, and the shortcut is set to True where the dimension is changed to ensure that the dimensions are the same between layers, and Global Average Pooling is used in the final output to replace the original VGG19 multi-layer fully connected layer.



```
def ResNet34(width, height, depth, classes):

    Img = Input(shape=(width,height,depth))

    layer= Conv2d_BN(Img,64,(7,7),strides=(2,2),padding='same')
    layer= MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(layer)

    layer= Residual_Block(layer,nb_filter=64,kernel_size=(3,3))
    layer= Residual_Block(layer,nb_filter=64,kernel_size=(3,3))
    layer= Residual_Block(layer,nb_filter=64,kernel_size=(3,3))

    layer= Residual_Block(layer,nb_filter=128,kernel_size=(3,3),strides=(2,2),shortcut=True)
    layer= Residual_Block(layer,nb_filter=128,kernel_size=(3,3))
    layer= Residual_Block(layer,nb_filter=128,kernel_size=(3,3))
    layer= Residual_Block(layer,nb_filter=128,kernel_size=(3,3))

    layer= Residual_Block(layer,nb_filter=256,kernel_size=(3,3),strides=(2,2),shortcut=True)
    layer= Residual_Block(layer,nb_filter=256,kernel_size=(3,3))
    layer= Residual_Block(layer,nb_filter=256,kernel_size=(3,3))
    layer= Residual_Block(layer,nb_filter=256,kernel_size=(3,3))
    layer= Residual_Block(layer,nb_filter=256,kernel_size=(3,3))
    layer= Residual_Block(layer,nb_filter=256,kernel_size=(3,3))

    layer= Residual_Block(layer,nb_filter=512,kernel_size=(3,3),strides=(2,2),shortcut=True)
    layer= Residual_Block(layer,nb_filter=512,kernel_size=(3,3))
    layer= Residual_Block(layer,nb_filter=512,kernel_size=(3,3))

    layer= GlobalAveragePooling2D()(layer)
    layer= Dense(classes,activation='softmax')(layer)

    model=Model(inputs=Img,outputs=layer)
    return model
```

The following is the ResNet34 architecture and the number of hyperparameters:

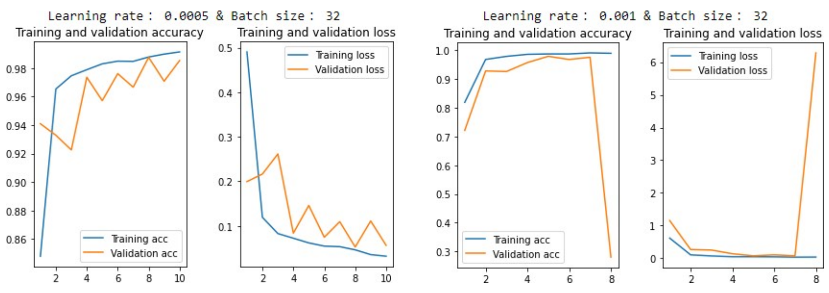
```
______________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==============================================================================
==================
input_11 (InputLayer)           [(None, 150, 150, 3) 0
______________________________________________________________________________
__________________
conv2d_360 (Conv2D)             (None, 75, 75, 64)   9472        input_11[0][0]
______________________________________________________________________________
__________________
batch_normalization_360 (BatchN (None, 75, 75, 64)   256         conv2d_360[0]
[0]
______________________________________________________________________________
__________________
max_pooling2d_10 (MaxPooling2D) (None, 38, 38, 64)   0           batch_normaliz
ation_360[0][0]
______________________________________________________________________________
__________________
conv2d_361 (Conv2D)             (None, 38, 38, 192)  110784      max_pooling2d_
10[0][0]
______________________________________________________________________________
__________________
batch_normalization_361 (BatchN (None, 38, 38, 192)  768         conv2d_361[0]
[0]
______________________________________________________________________________
                                        ⋮
average_pooling2d (AveragePooli (None, 1, 1, 1024)   0           concatenate_8
[0][0]
______________________________________________________________________________
__________________
flatten (Flatten)               (None, 1024)         0           average_poolin
g2d[0][0]
______________________________________________________________________________
__________________
dropout (Dropout)               (None, 1024)         0           flatten[0][0]
______________________________________________________________________________
__________________
dense_10 (Dense)                (None, 1000)         1025000     dropout[0][0]
______________________________________________________________________________
__________________
dense_11 (Dense)                (None, 10)           10010       dense_10[0][0]
==============================================================================
==================
Total params: 6,434,034
Trainable params: 6,433,522
Non-trainable params: 512
```

2. Evaluation

The following figure shows the accuracy of the testing set obtained by ResNet34 under different parameters. It can be clearly seen from the figure that when the learning rate=0.0005 and the batch size=16, the accuracy is the highest at 0.9866, and when the learning rate=0.001 and batch size=32 the testing accuracy is the worst.

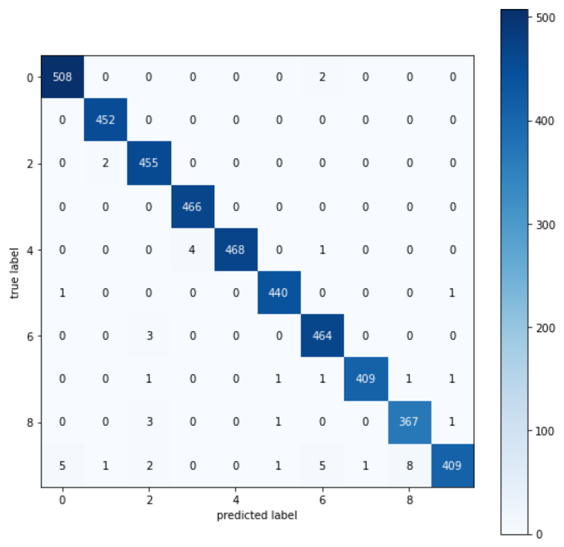
Test Accuracy of Best ResNet34 Model			
batch_size \ lr	0.0001	0.0005	0.001
16	0.9844	0.9866	0.9654
32	0.9777	0.9592	0.2807
48	0.977	0.94	0.8129

From the accuracy and loss curves of the training and validation set. Although the test accuracy in the left figure is higher, it can be clearly seen that there is an obvious overfit between the training set and validation set. The entire model is in a stage of oscillation and instability. This situation may be improved by increasing the epoch to minimize the oscillation. The test accuracy in the right picture obviously shows more overfitting after epoch=7 and its validation loss value is relatively large, illustrating gradient explosion. In contrast, using the data from the figure on the right as ResNet34's parameter, it should set the epoch to 7.



3. Confusion Matrix

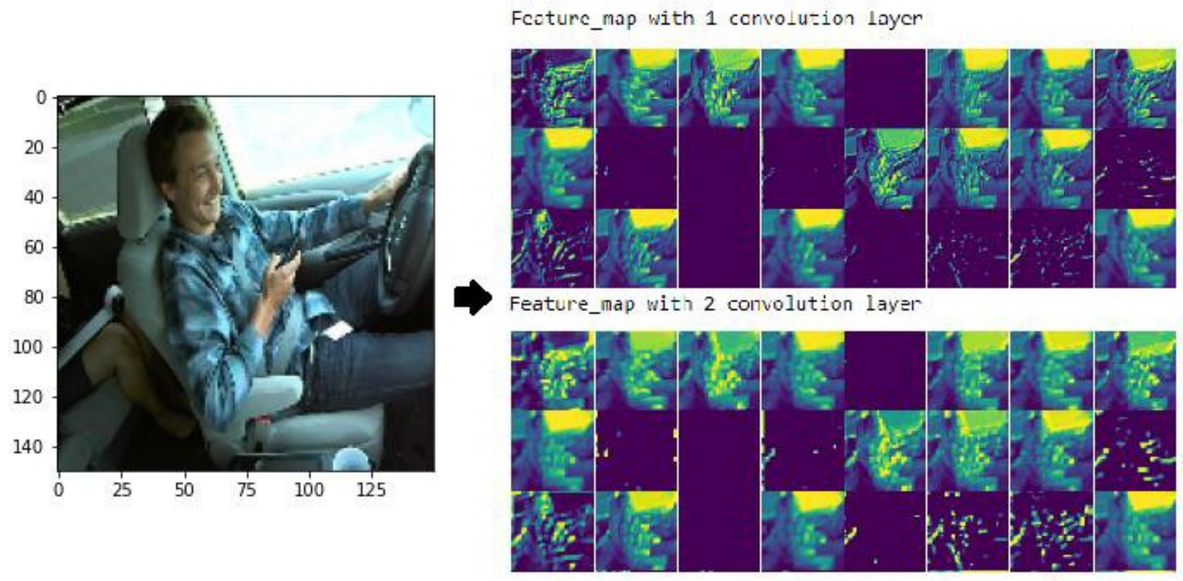
Although the stability of the model at this time is low, it can be seen from the Confusion Matrix that except for a small number of testing datasets that are misclassified, the rest are correct.

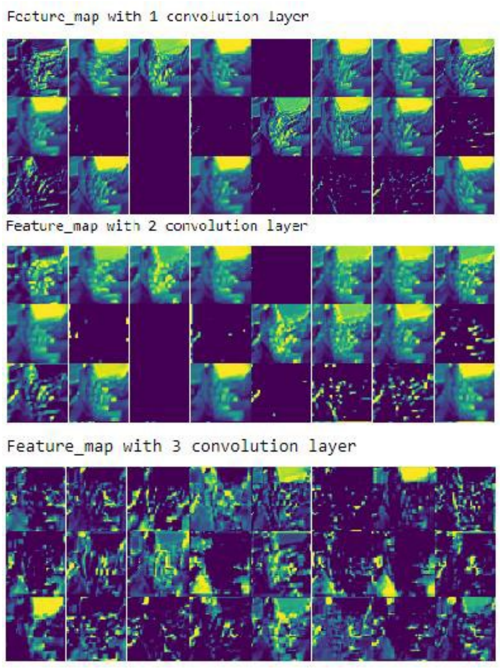


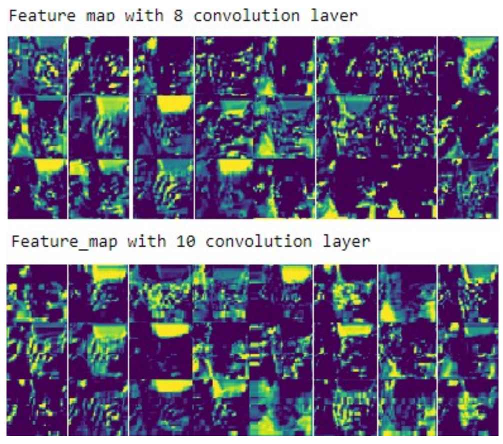
4. Feature Maps Visualization

The following photos are the filter of the ResNet34 testing set (partially shown):

The picture will convolute with the filter to get the corresponding feature map, and convert from RGB to BGR to present a blue image. In addition, for the deeper filter, the obtained feature map will be more blurred. It can be seen that the shallower filter will initially identify sharper features such as edges, then determine the smoother part.


0
20
40
60
80
100
120
140
0
25
50
75
100
125
Feature_map with 1 convolution layer
Feature_map with 2 convolution layer

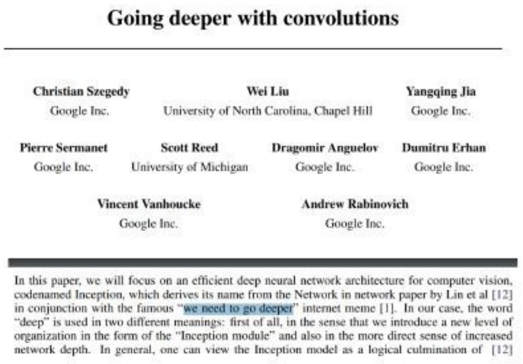

Feature_map with 1 convolution layer
Feature_map with 2 convolution layer
Feature_map with 3 convolution layer


Feature_map with 8 convolution layer
Feature_map with 10 convolution layer

Inception V1 (GoogLeNet)

1.Introduction

GoogLeNet is a deep model structure researched by Google. Why is it called "GoogLeNet" instead of "GoogleNet". It is said to pay tribute to "LeNet", so it is named "GoogLeNet". As for why it is called Inception, it is because It can effectively reduce the amount of parameters in a deeper network to achieve better results, just as mentioned in the paper, it comes from the famous saying "We need to go deeper!" in the movie "Inception".

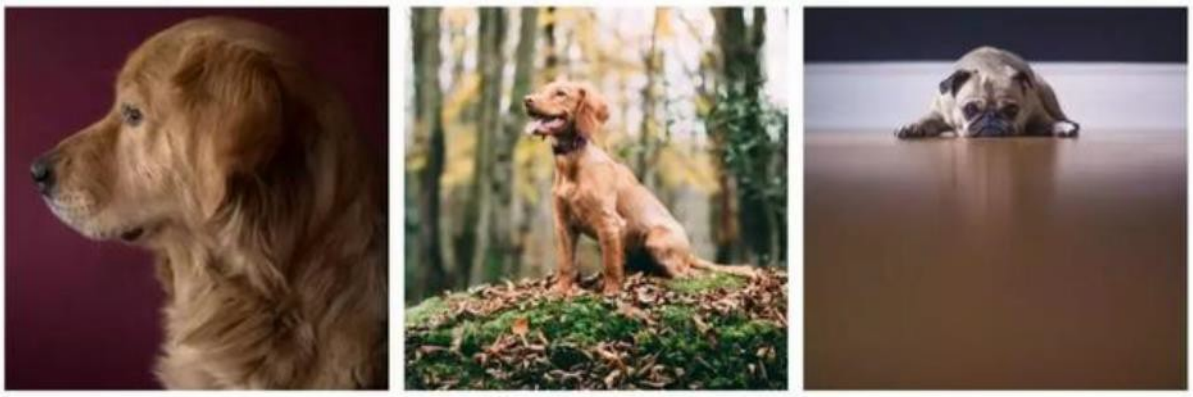


Going deeper with convolutions

Christian Szegedy Google Inc.	**Wei Liu** University of North Carolina, Chapel Hill		**Yangqing Jia** Google Inc.
Pierre Sermanet Google Inc.	**Scott Reed** University of Michigan	**Dragomir Anguelov** Google Inc.	**Dumitru Erhan** Google Inc.
Vincent Vanhoucke Google Inc.	**Andrew Rabinovich** Google Inc.		

In this paper, we will focus on an efficient deep neural network architecture for computer vision, codenamed Inception, which derives its name from the Network in network paper by Lin et al [12] in conjunction with the famous "we need to go deeper" internet meme [1]. In our case, the word "deep" is used in two different meanings: first of all, in the sense that we introduce a new level of organization in the form of the "Inception module" and also in the more direct sense of increased network depth. In general, one can view the Inception model as a logical culmination of [12]

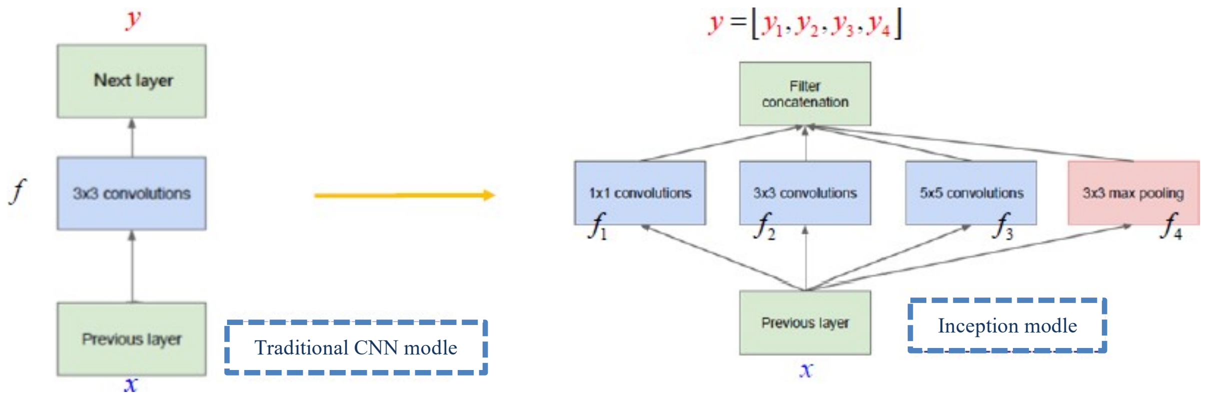
The main concept of Inception V1 stems from the large difference in the size of important parts in the image. For example, an image of a dog can be any of the following. The dog is shown in different area in each image:



Due to differences in object positions, it is more difficult to choose an appropriate kernel size for the convolution. Images with global distribution of information prefer larger kernels, while images with local distribution of objects prefer smaller kernels.

Not only is it easier to overfit for very deep networks, but it is extremely difficult to transmit gradient updates to the entire model. Furthermore, simply stacking larger convolution layers is computationally expensive.

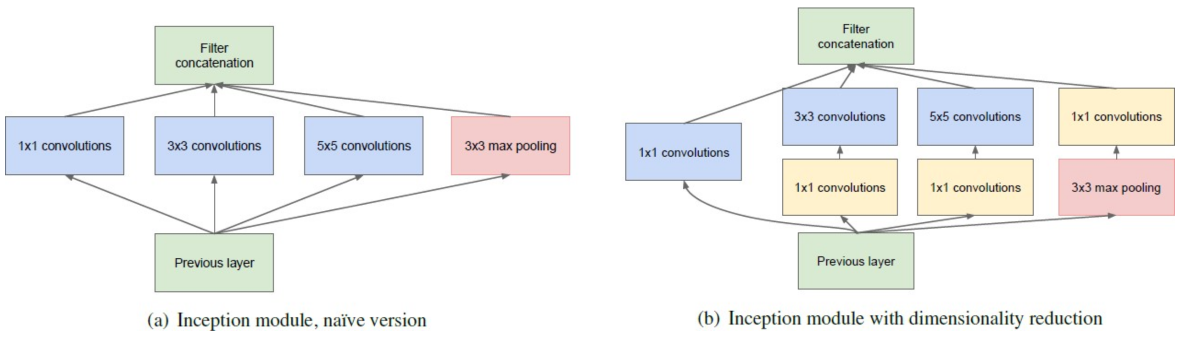
The concept of Inception is to convolute the same layer with different kernels, so that the problem of object position can be solved through different kernels. Different features are extracted through three convolutions of different kernel sizes and 3x3 Maxpooling, then these results are concatenated together with the channel axis. By increasing the network width in this way, more features and details of the image can be captured.



In addition, Inception does an optimization through the above architecture. The Inception module will first reduce the output channel through 1x1 convolution, and connect to the original convolution layer. Then, since max pooling cannot reduce the channel dimension after the input feature map passes through 3x3 Max pooling, 1x1 convolution will be used to reduce the channel dimension.

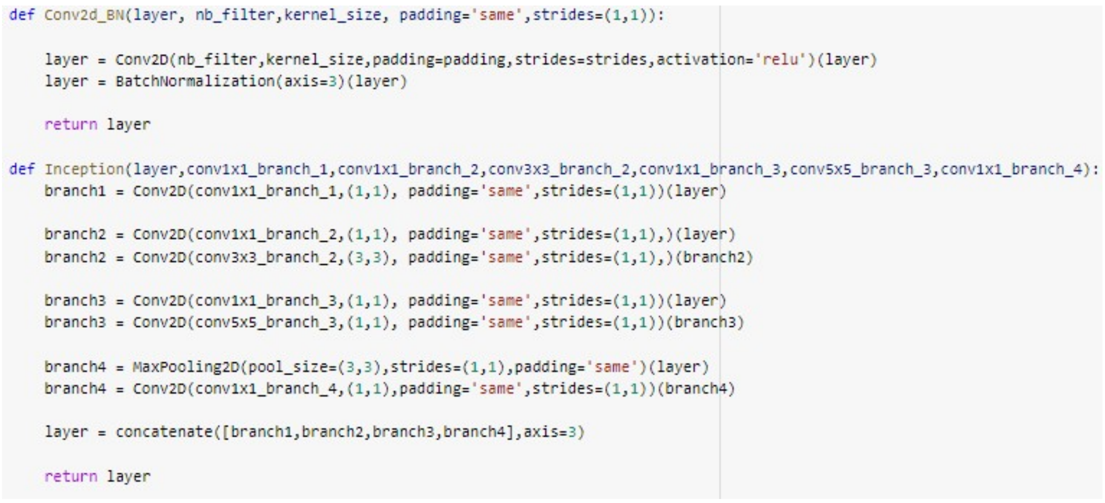
Assuming that the size of our feature map input is 150x150x64, if we do a 5x5 convolution with 128 filters (stride = 1, pad = 2), the output will be 150x150x128, then the parameters of this convolution layer are 64*5*5 *128 = 204800.

If we go through the 1x1 convolution of 32 filters, then go through the 5x5 convolution layer with 128 filters, the output is the same as 150x150x128, but the parameters are reduced to 64x1x1x32+32x5x5x125=104448, which is relatively less than half of the parameters.



(a) Inception module, naïve version (b) Inception module with dimensionality reduction

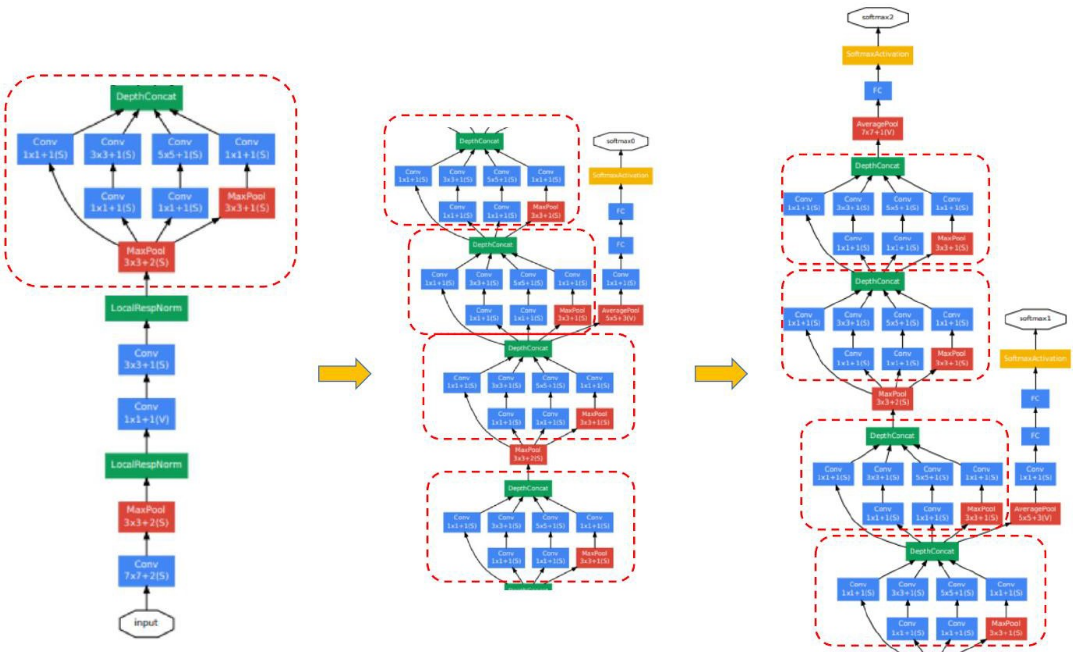
The following code is represented by the above concept:
Inception block:
branch1: The leftmost branch, input from the previous layer to do 1x1 Convolution, and the number of channels is conv1X1_branch_1.
branch2: In the second branch, after getting input from the previous layer, implements 1x1 convolution and then 3x3 convolution, the number of channels is conv1X1_branch_2, conv3x3_branch_2 respectively.
branch3: The third branch, after getting input from the previous layer, implements 1x1 convolution and then 5x5 convolution, the number of channels is conv1X1_branch_3, conv5x5_branch_3 respectively.
branch4: The fourth branch, after getting input from the previous layer, implements 3x3 Max Pooling then 1x1 convolution. The number of channels of Convolution is conv1x1_branch_4. Finally, use Constance to combine the tensors of the 4 branches and send it to the next layer.

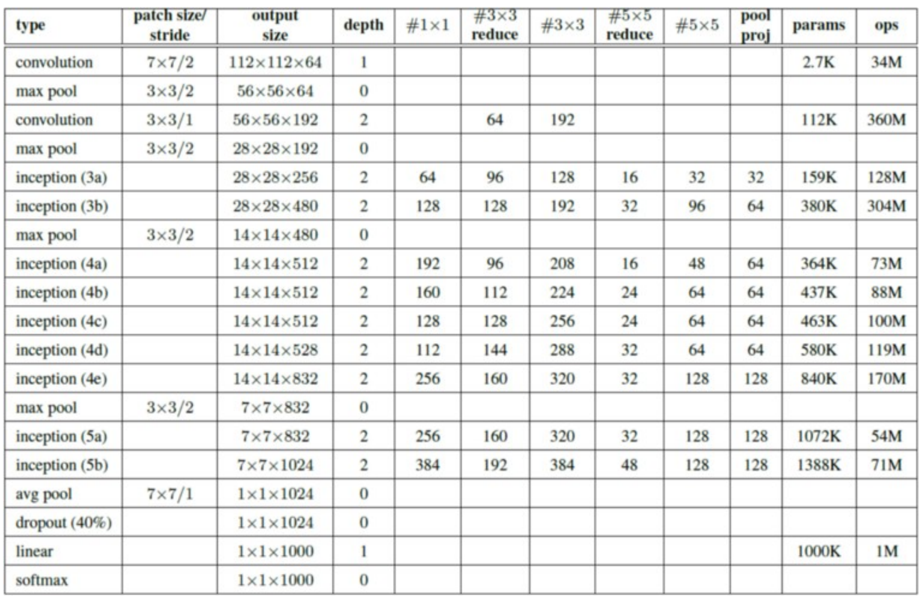


```
def Conv2d_BN(layer, nb_filter,kernel_size, padding='same',strides=(1,1)):

    layer = Conv2D(nb_filter,kernel_size,padding=padding,strides=strides,activation='relu')(layer)
    layer = BatchNormalization(axis=3)(layer)

    return layer

def Inception(layer,conv1x1_branch_1,conv1x1_branch_2,conv3x3_branch_2,conv1x1_branch_3,conv5x5_branch_3,conv1x1_branch_4):
    branch1 = Conv2D(conv1x1_branch_1,(1,1), padding='same',strides=(1,1))(layer)

    branch2 = Conv2D(conv1x1_branch_2,(1,1), padding='same',strides=(1,1),)(layer)
    branch2 = Conv2D(conv3x3_branch_2,(3,3), padding='same',strides=(1,1),)(branch2)

    branch3 = Conv2D(conv1x1_branch_3,(1,1), padding='same',strides=(1,1))(layer)
    branch3 = Conv2D(conv5x5_branch_3,(1,1), padding='same',strides=(1,1))(branch3)

    branch4 = MaxPooling2D(pool_size=(3,3),strides=(1,1),padding='same')(layer)
    branch4 = Conv2D(conv1x1_branch_4,(1,1),padding='same',strides=(1,1))(branch4)

    layer = concatenate([branch1,branch2,branch3,branch4],axis=3)

    return layer
```

The architecture of Inception below shows that there is a total of 22 layers, which is deeper than the 16 layers of VGG16, but the required parameters are less than half of that of VGG16.

When the Inception Block is created, the entire model is assembled through the Inception architecture. The red frame is the above Inception block, and the green part of the Inception is LocalRespNorm. Through the dimension along the channel, the values of different channels adjacent to the same position are used for Normalization. But this technique has been rarely used, and most of them use Batch Normalization as mentioned earlier, using the branches on both sides as 2 auxiliary classifiers to send the results of the intermediate nodes.





type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

In summary, InceptionV1 has the following advantages:

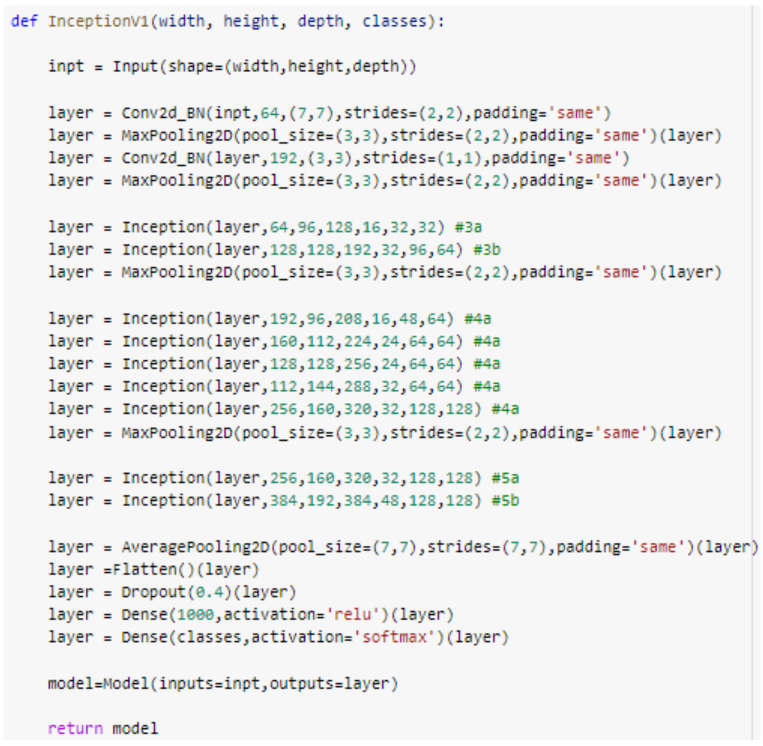
1. Change convolution and Max pooling to Inception architecture

2. Use average pooling instead of fully Connection in the final classification

3. The network adds 2 auxiliary classifiers, in order to avoid the situation of gradient descent

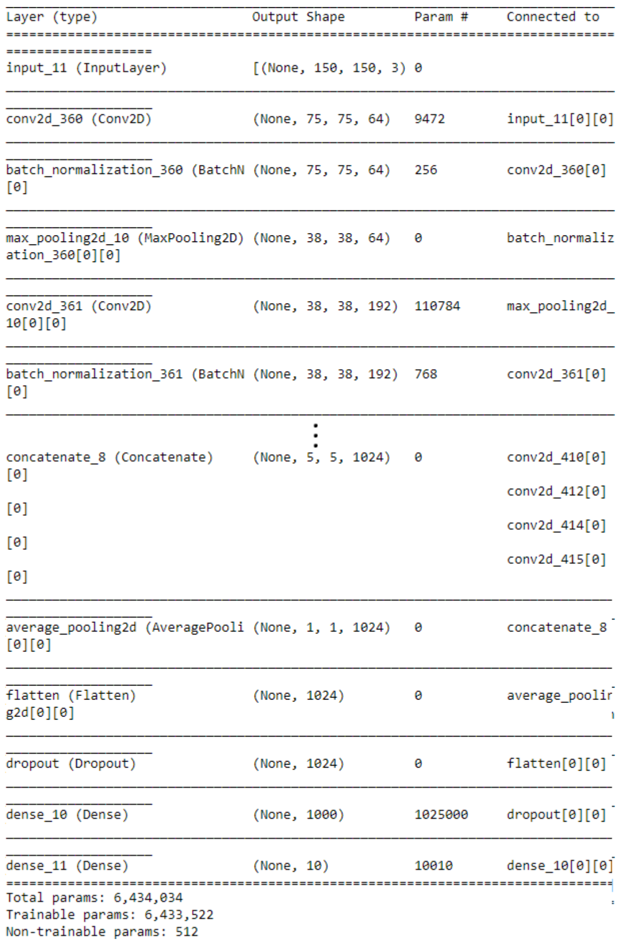
Inception module:

After omitting the two auxiliary classifiers of the branch, sending the output of the complete 22-layer architecture and the Inception block required by different layers as components according to the above table, then group the layers of the entire model as before.



```python
def InceptionV1(width, height, depth, classes):

    inpt = Input(shape=(width,height,depth))

    layer = Conv2d_BN(inpt,64,(7,7),strides=(2,2),padding='same')
    layer = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(layer)
    layer = Conv2d_BN(layer,192,(3,3),strides=(1,1),padding='same')
    layer = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(layer)

    layer = Inception(layer,64,96,128,16,32,32) #3a
    layer = Inception(layer,128,128,192,32,96,64) #3b
    layer = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(layer)

    layer = Inception(layer,192,96,208,16,48,64) #4a
    layer = Inception(layer,160,112,224,24,64,64) #4a
    layer = Inception(layer,128,128,256,24,64,64) #4a
    layer = Inception(layer,112,144,288,32,64,64) #4a
    layer = Inception(layer,256,160,320,32,128,128) #4a
    layer = MaxPooling2D(pool_size=(3,3),strides=(2,2),padding='same')(layer)

    layer = Inception(layer,256,160,320,32,128,128) #5a
    layer = Inception(layer,384,192,384,48,128,128) #5b

    layer = AveragePooling2D(pool_size=(7,7),strides=(7,7),padding='same')(layer)
    layer =Flatten()(layer)
    layer = Dropout(0.4)(layer)
    layer = Dense(1000,activation='relu')(layer)
    layer = Dense(classes,activation='softmax')(layer)

    model=Model(inputs=inpt,outputs=layer)

    return model
```

The following is the InceptionV1 architecture and the number of hyperparameters:



```
________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
================================================================================
==================
input_11 (InputLayer)           [(None, 150, 150, 3) 0

________________________________________________________________________________
__________________
conv2d_360 (Conv2D)             (None, 75, 75, 64)   9472        input_11[0][0]
________________________________________________________________________________
__________________
batch_normalization_360 (BatchN (None, 75, 75, 64)   256         conv2d_360[0]
[0]
________________________________________________________________________________
__________________
max_pooling2d_10 (MaxPooling2D) (None, 38, 38, 64)   0           batch_normaliz
ation_360[0][0]
________________________________________________________________________________
__________________
conv2d_361 (Conv2D)             (None, 38, 38, 192)  110784      max_pooling2d_
10[0][0]
________________________________________________________________________________
__________________
batch_normalization_361 (BatchN (None, 38, 38, 192)  768         conv2d_361[0]
[0]
________________________________________________________________________________
                                       ⋮
concatenate_8 (Concatenate)     (None, 5, 5, 1024)   0           conv2d_410[0]
[0]
                                                                 conv2d_412[0]
[0]
                                                                 conv2d_414[0]
[0]
                                                                 conv2d_415[0]
[0]
________________________________________________________________________________
__________________
average_pooling2d (AveragePooli (None, 1, 1, 1024)   0           concatenate_8
[0][0]
________________________________________________________________________________
__________________
flatten (Flatten)               (None, 1024)         0           average_poolin
g2d[0][0]
________________________________________________________________________________
__________________
dropout (Dropout)               (None, 1024)         0           flatten[0][0]
________________________________________________________________________________
__________________
dense_10 (Dense)                (None, 1000)         1025000     dropout[0][0]
________________________________________________________________________________
__________________
dense_11 (Dense)                (None, 10)           10010       dense_10[0][0]
================================================================================
Total params: 6,434,034
Trainable params: 6,433,522
Non-trainable params: 512
```

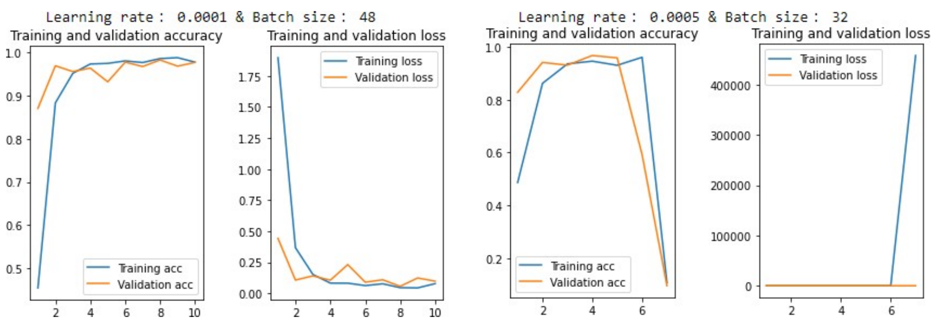
2. Evaluation

The following figure shows the accuracy of the testing set obtained by Inception under different parameters. It can be clearly seen from the figure that when the learning rate=0.0001 and batch size=48, the accuracy is the highest at 0.9777, and when the learning rate=0.005 and batch size=32 the testing accuracy is the worst.

Test Accuracy of Best InceptionV1 Model

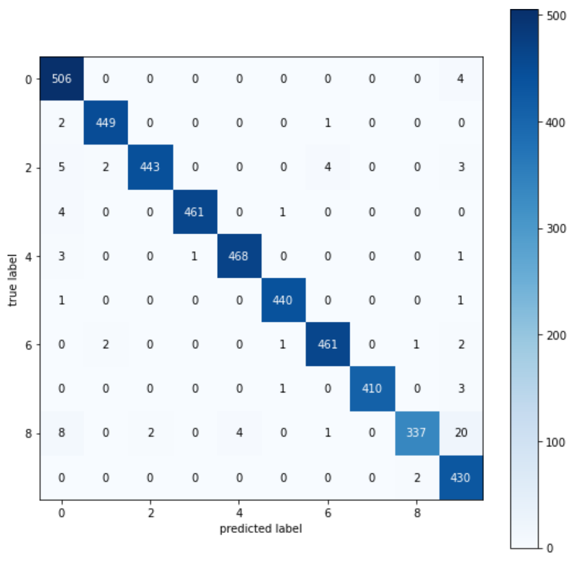
batch_size \ lr	0.0001	0.0005	0.001
16	0.9423	0.1137	0.1137
32	0.9249	0.1055	0.9382
48	0.9777	0.9632	0.835

From the accuracy and loss curves of the training and validation set, test accuracy is higher in the left figure and clearly shows that there is a certain degree of stability between the training set and the validation set; while overfit occurs at epoch=6 in the right image. When overfit occurs, there is a gradient explosion at epoch=7. Not only does its accuracy drop significantly, but its loss also increases significantly. Therefore, if you want to use the data in the right image as the parameter of InceptionV1, it is more reasonable to set the epoch to 6.



3.Confusion Matrix

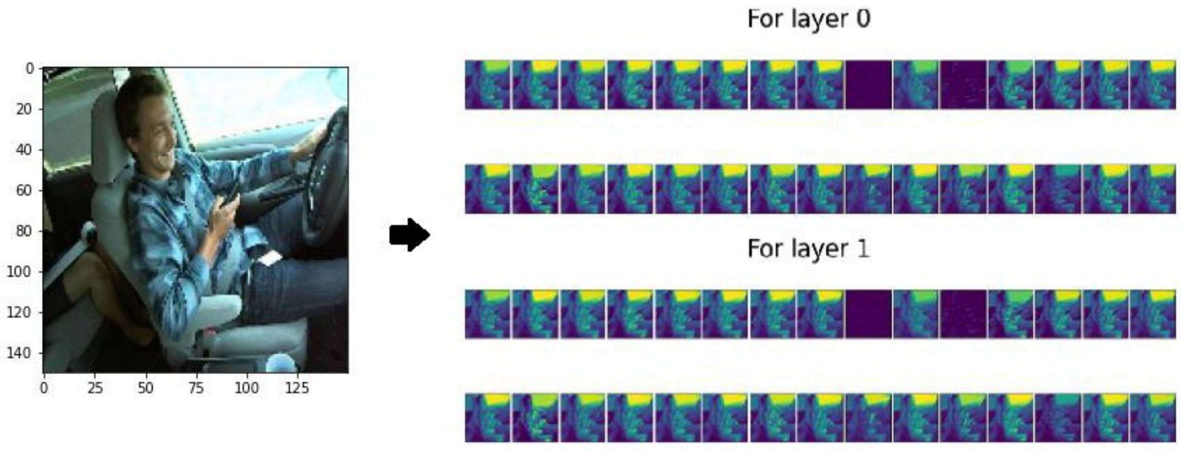
InceptionV1 has a reasonable stability, and it can be seen from the onfusion Matrix that the testing dataset is generally classified correctly.

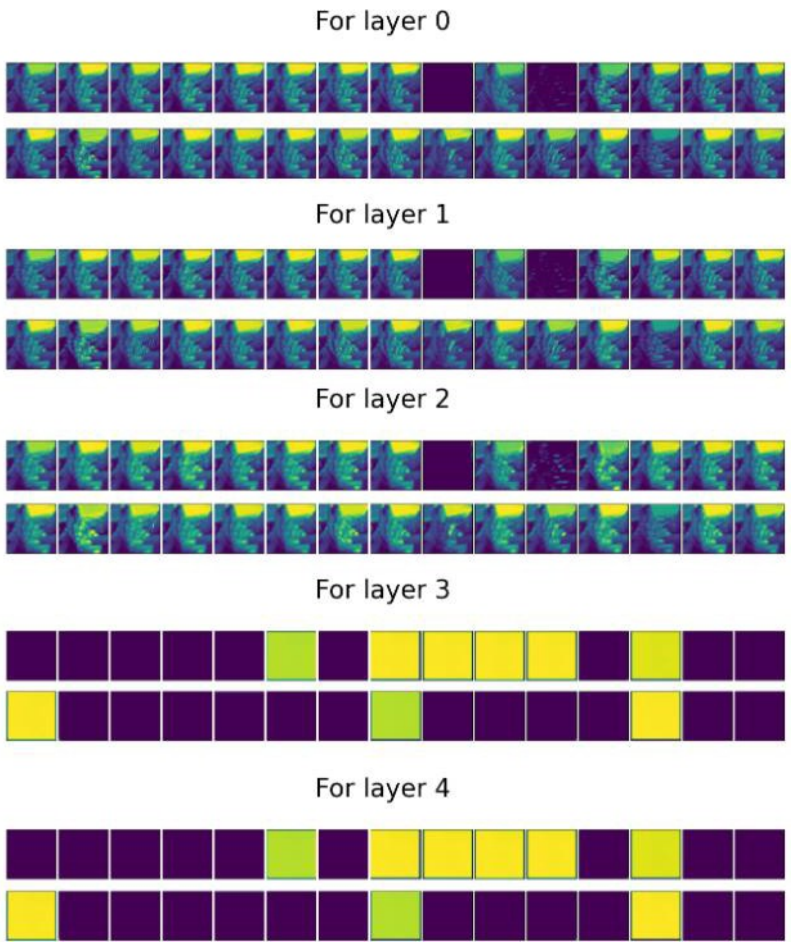


4. Feature Maps Visualization

The following photos are the filter of the InceptionV1 testing set (partially shown):

The picture will convolute with the filter to get the corresponding feature map which similar to VGG16, also convert from RGB to BGR to present a blue image. In addition, for the deeper filter, the obtained feature map will be more blurred. It can be seen that the shallower filter will initially identify sharper features such as edges, then determine the smoother part.

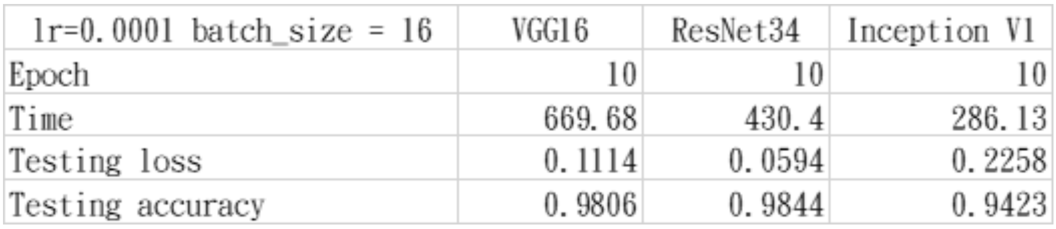




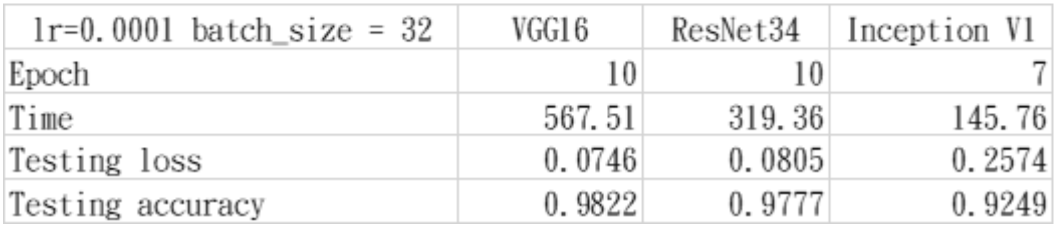
Conclusion

Via VGG16_best_weight, ResNet34_best_weight and InceptionV1_best_weight. Discuss the best performing models.

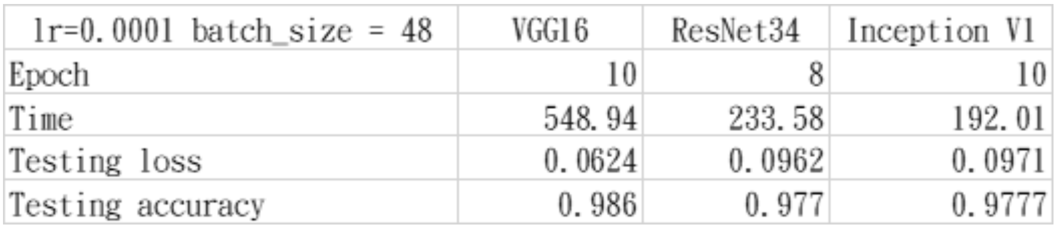
Leaning rate=0.0001



lr=0.0001 batch_size = 16	VGG16	ResNet34	Inception V1
Epoch	10	10	10
Time	669.68	430.4	286.13
Testing loss	0.1114	0.0594	0.2258
Testing accuracy	0.9806	0.9844	0.9423

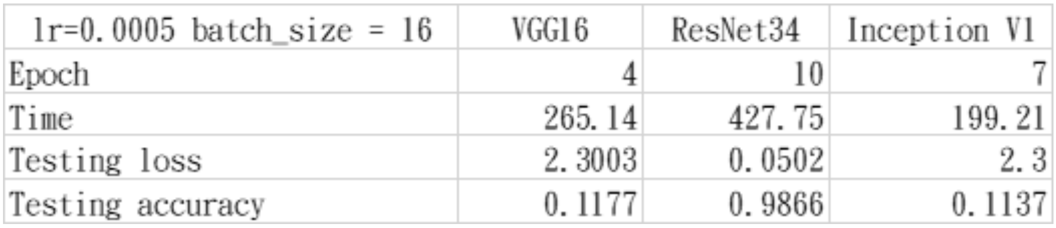


lr=0.0001 batch_size = 32	VGG16	ResNet34	Inception V1
Epoch	10	10	7
Time	567.51	319.36	145.76
Testing loss	0.0746	0.0805	0.2574
Testing accuracy	0.9822	0.9777	0.9249

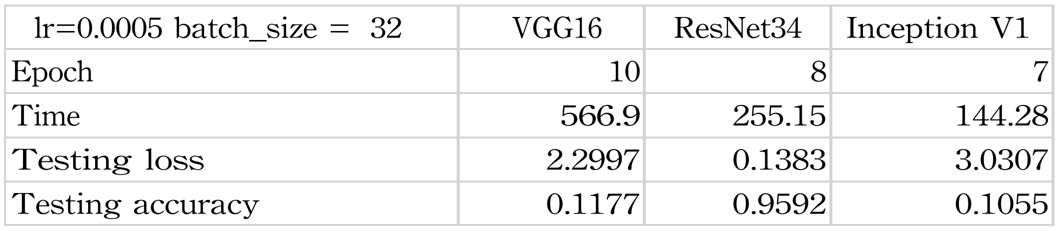


lr=0.0001 batch_size = 48	VGG16	ResNet34	Inception V1
Epoch	10	8	10
Time	548.94	233.58	192.01
Testing loss	0.0624	0.0962	0.0971
Testing accuracy	0.986	0.977	0.9777

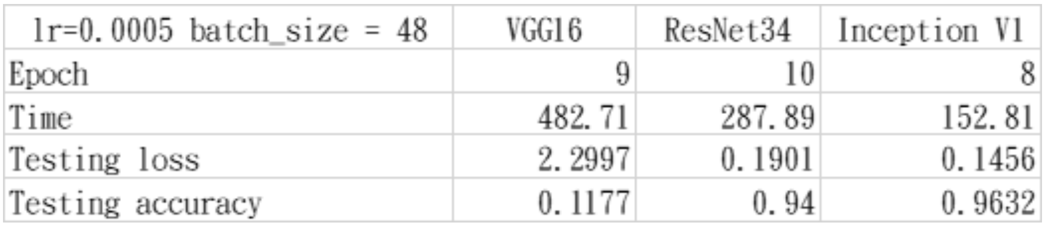
Leaning rate=0.0005



lr=0.0005 batch_size = 16	VGG16	ResNet34	Inception V1
Epoch	4	10	7
Time	265.14	427.75	199.21
Testing loss	2.3003	0.0502	2.3
Testing accuracy	0.1177	0.9866	0.1137



lr=0.0005 batch_size = 32	VGG16	ResNet34	Inception V1
Epoch	10	8	7
Time	566.9	255.15	144.28
Testing loss	2.2997	0.1383	3.0307
Testing accuracy	0.1177	0.9592	0.1055



lr=0.0005 batch_size = 48	VGG16	ResNet34	Inception V1
Epoch	9	10	8
Time	482.71	287.89	152.81
Testing loss	2.2997	0.1901	0.1456
Testing accuracy	0.1177	0.94	0.9632

Leaning rate=0.001

lr=0.001 batch_size = 16	VGG16	ResNet34	Inception V1
Epoch	4	9	8
Time	266.12	382.14	225.43
Testing loss	2.2998	0.1076	2.2998
Testing accuracy	0.1177	0.9654	0.1137

lr=0.001 batch_size = 32	VGG16	ResNet34	Inception V1
Epoch	8	8	8
Time	451.04	255.5	164.22
Testing loss	2.2998	6.1295	0.2035
Testing accuracy	0.1177	0.2807	0.9382

lr=0.001 batch_size = 48	VGG16	ResNet34	Inception V1
Epoch	4	4	10
Time	213.24	118.27	189.09
Testing loss	2.3	0.7794	0.5434
Testing accuracy	0.1177	0.8129	0.835

Number of parameters:

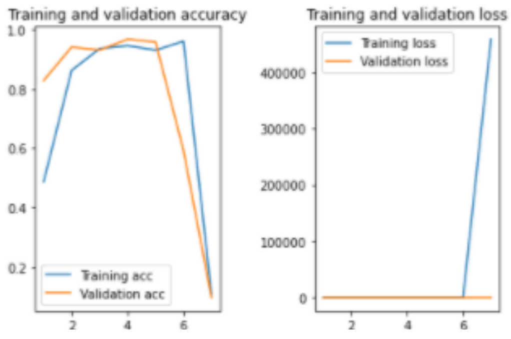
Number of parameter	VGG16	ResNet34	Inception V1
Trainable params	69,136,962	22,674,570	6,433,522
Non-trainable params	0	17,024	512
Total params	69,136,962	22,691,594	6,434,034

By arranging the results of all the above models, it can be clearly seen that the impact of the learning rate for the model is relatively important.

When the learning rate=0.0001, all three models can operate stably, and there is no gradient explosion or gradient disappearance. In addition, because the number of required parameters is different, the training time obviously different. From the above table, we can observe that the parameters trained by VGG16 are even 10 times more than Inception. It can be found that Inception is not only the best in terms of training time but also maintains a good performance.

When the learning rate=0.005, the VGG16 shows underfitting. It can be inferred that the VGG16 oscillates at the local minimum because the learning rate is too high and batch normalization is not used, so it cannot converge to the minimum loss. ResNet34 is based on ResBlock, which can use Identity mapping to prevent gradients from disappearing or exploding, and maintains good stability in a deeper architecture. Inception shows gradient explosion in the model when batch = 16 and 32, which can be clearly seen from the behavior of the curve in the figure below after the 6th epoch.

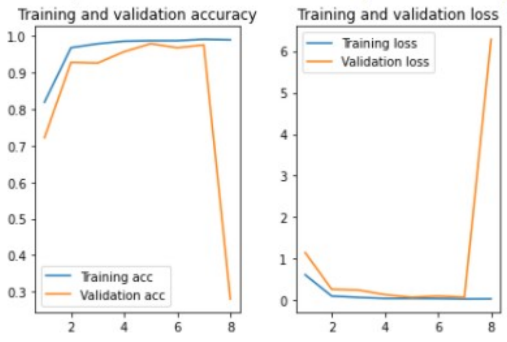
```
449/449 [==============================] - 20s 45ms/step - loss: 0.1835 - accuracy: 0.9463 - val_loss: 0.1366 - val_accuracy: 0.9677
Epoch 5/10
449/449 [==============================] - ETA: 0s - loss: 0.2521 - accuracy: 0.9307
Epoch 00005: val_loss did not improve from 0.13662
449/449 [==============================] - 20s 45ms/step - loss: 0.2521 - accuracy: 0.9307 - val_loss: 0.1473 - val_accuracy: 0.9588
Epoch 6/10
448/449 [============================>.] - ETA: 0s - loss: 0.1639 - accuracy: 0.9610
Epoch 00006: val_loss did not improve from 0.13662
449/449 [==============================] - 20s 44ms/step - loss: 0.1644 - accuracy: 0.9609 - val_loss: 2.2586 - val_accuracy: 0.5925
Epoch 7/10
448/449 [============================>.] - ETA: 0s - loss: 459286.1875 - accuracy: 0.1069
Epoch 00007: val_loss did not improve from 0.13662
449/449 [==============================] - 19s 43ms/step - loss: 458806.1250 - accuracy: 0.1068 - val_loss: 2.3020 - val_accuracy: 0.0953
Training Time : 144.2842893600464

141/141 [==============================] - 2s 15ms/step - loss: 3.0307 - accuracy: 0.1055
```



Finally, increase the learning rate to 0.001, but VGG16 remains underfitting. When ResNet34 has batch size=32, the validation accuracy is high but the test accuracy is only 0.28 (as shown in the figure below). It can be known that the model has an overfitting problem at this time. The accuracy of unseen training data is only 0.28.

The Inception compares learning rates 0.0005 and 0.001. When the smaller batch size=16, the gradient of the model will explode around the 4th to 5th epoch, so it can be known that where the learning rate is larger with the smaller batch size, it is easier for gradient oscillation and gradient explosion to happen. When using a larger batch size, the direction of the extreme value is more accurate, and the gradient oscillation is smaller, but it consumes more memory resources.

```
Epoch 00006: val_loss did not improve from 0.07356
449/449 [==============================] - 31s 68ms/step - loss: 0.0427 - accuracy: 0.9875 - val_loss: 0.1045 - val_accuracy: 0.9682
Epoch 7/10
448/449 [============================>.] - ETA: 0s - loss: 0.0324 - accuracy: 0.9911
Epoch 00007: val_loss did not improve from 0.07356
449/449 [==============================] - 31s 69ms/step - loss: 0.0323 - accuracy: 0.9912 - val_loss: 0.0738 - val_accuracy: 0.9760
Epoch 8/10
448/449 [============================>.] - ETA: 0s - loss: 0.0348 - accuracy: 0.9899
Epoch 00008: val_loss did not improve from 0.07356
449/449 [==============================] - 31s 68ms/step - loss: 0.0351 - accuracy: 0.9898 - val_loss: 6.2853 - val_accuracy: 0.2804
Training Time : 255.49612045288086

141/141 [==============================] - 3s 20ms/step - loss: 6.1295 - accuracy: 0.2807
```



Reference

[1] Karen Simonyan, Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. Visual Geometry Group, Department of Engineering Science, University of Oxford

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun. Deep Residual Learning for Image Recognition. Microsoft Research

[3] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. Going Deeper with Convolutions. Google Inc.

1. http://deanhan.com/2018/07/26/vgg16/?fbclid=IwAR03g4s_Jn9JaJeXKlde2Ggg1F_0Rq8xOSz65f3f0xU9egEKLwWqarSY2Jc
2. https://arxiv.org/abs/1409.1556
3. https://codertw.com/%E7%A8%8B%E5%BC%8F%E8%AA%9E%E8%A8%80/442795/
4. https://ithelp.ithome.com.tw/articles/10219945