Team 7

Lei Liu, Yuchen He, Xiangyu Chen

1. CNN: Convolutional neural network

We did six model tests to select the best model with a high performance. Firstly, we extract data form 5 classes of ‘tiny-imagenet’ as the dataset. Next, the training parameters that are not related to the architectures are guaranteed to be consistent, so as to eliminate the influence of these parameters on the accuracy of the model.

The model architectures we used:

Convolutional Layer: Convolution is actually the inner product, which is the inner product of pixels of a block according to a number of certain weights (i.e., convolution kernel), and its output is one of the extracted features. The size of the convolution kernel is generally smaller than the size of the input image (or full connection if equal to), so the extracted features of the convolution will pay more attention to local. The higher the number of layers, the more global the extracted features will be.

Pooling Layer: To sample or aggregate a piece of data. Pooling layer can effectively reduce the size of the parameter matrix, thus reducing the number of parameters in the final fully connected layer. The use of pooling layer can not only accelerate the calculation speed but also prevent overfitting.

Fully Connected Layer: primarily to implement classification by reassembling the previous local features into a complete graph by weight matrix.

BatchNormalization: The activation input value before the nonlinear transformation is gradually shifted or changed with the deepening of the network depth or during the training process. Through Batch Normalization, the distribution of this input value of any neuron in each layer of the neural network is forcibly pulled back to the standard normal distribution with a mean of 0 and a variance of 1.

Models	Convolutional Layer	Filters on Convolutional Layers	Pooling Layer	Fully Connected Layer	Batch Normalization	Test Loss	Test Acc
model1	4	32 ~ 64	2	2	0	1.247	0.516
model2	5	128	3	3	0	1.530	0.312
model3	9	128 ~ 512	0	2	10	1.048	0.615
***model4**	8	128 ~ 512	4	3	10	0.724	0.772
model5	8	128 ~ 1024	4	4	11	0.898	0.752

* Best performance model

Model1


Training and validation accuracy
Training acc
Validation acc
0.50
0.45
0.40
0.35
0.30
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Training and validation loss
Training loss
Validation loss
1.60
1.55
1.50
1.45
1.40
1.35
1.30
1.25
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Model2


Training and validation accuracy
Training acc
Validation acc
0.30
0.28
0.26
0.24
0.22
0.20
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

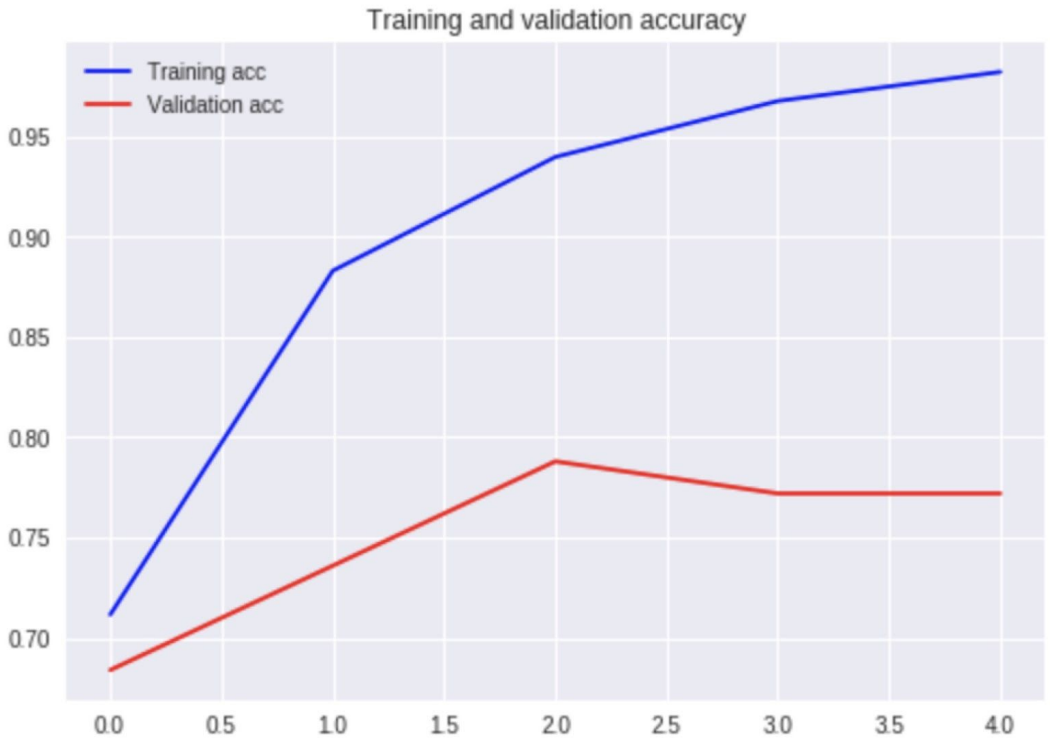

Training and validation loss
Training loss
Validation loss
161
160
159
158
157
156
155
154
153
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Model3


Training and validation accuracy
Training acc
Validation acc
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Training and validation loss
Training loss
Validation loss
1.6
1.5
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

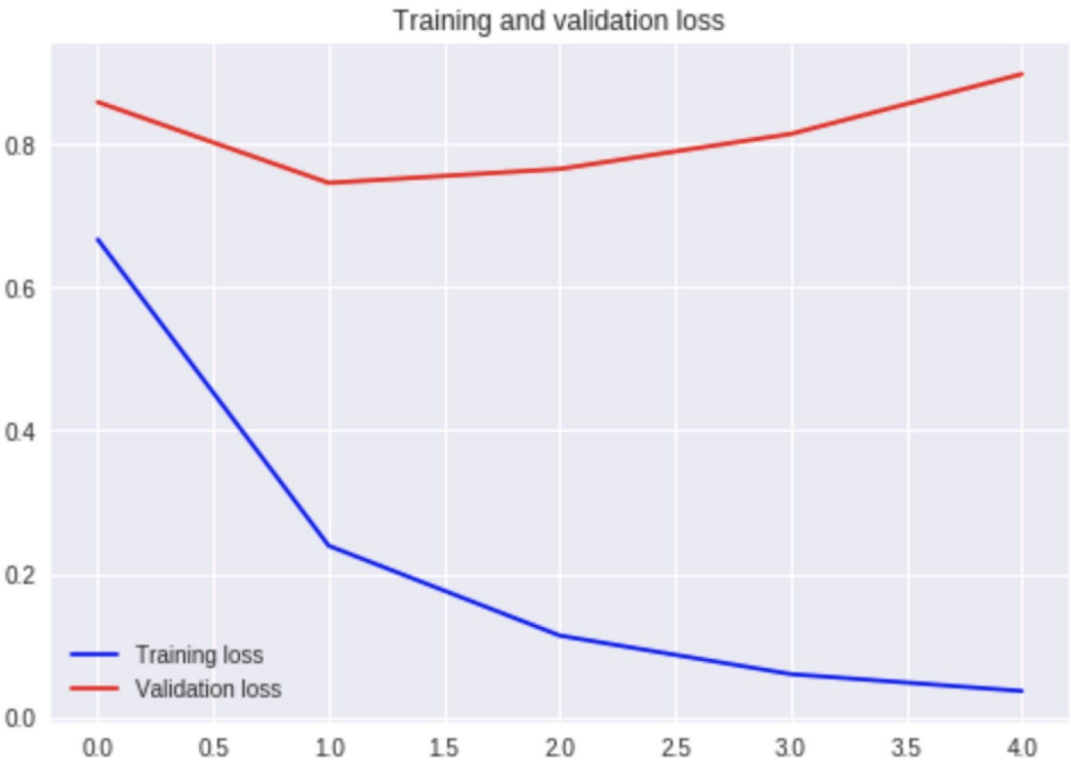
Model4


Training and validation accuracy
Training acc
Validation acc
0.95
0.90
0.85
0.80
0.75
0.70
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Training and validation loss
Training loss
Validation loss
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Model5


Training and validation accuracy
Training acc
Validation acc
1.00
0.95
0.90
0.85
0.80
0.75
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0


Training and validation loss
Training loss
Validation loss
0.8
0.6
0.4
0.2
0.0
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0

Best Model Architecture

After evaluating these models, we select the 4th model as our best model. Our best model has 8 convolutional layers, each two of which are followed by a max-pooling layer with 2x2 kernels, 3 fully connected layers, and 10 batch normalizations in total following every layer above except the last one. These convolutional layers work in groups of two, and these layers have 128 to 512 filters in ascending order. The first layers of each group have a stride of 1 and the second ones have a stride of 2.

Result

Finally, we get test loss as 3.194, and test accuracy as 0.2795.

```
Epoch 1/5
1200/1200 [==============================] - 3763s 3s/step - loss: 4.2464 - acc: 0.1208 - categorical_accuracy: 0.120
8 - val_loss: 4.1045 - val_acc: 0.1354 - val_categorical_accuracy: 0.1354
Epoch 2/5
1200/1200 [==============================] - 4069s 3s/step - loss: 3.6482 - acc: 0.2005 - categorical_accuracy: 0.200
5 - val_loss: 3.7823 - val_acc: 0.1801 - val_categorical_accuracy: 0.1801
Epoch 3/5
1200/1200 [==============================] - 4071s 3s/step - loss: 3.3515 - acc: 0.2491 - categorical_accuracy: 0.249
1 - val_loss: 3.6308 - val_acc: 0.2111 - val_categorical_accuracy: 0.2111
Epoch 4/5
1200/1200 [==============================] - 3711s 3s/step - loss: 3.1380 - acc: 0.2860 - categorical_accuracy: 0.286
0 - val_loss: 3.3196 - val_acc: 0.2617 - val_categorical_accuracy: 0.2617
Epoch 5/5
1200/1200 [==============================] - 3714s 3s/step - loss: 2.9762 - acc: 0.3160 - categorical_accuracy: 0.316
0 - val_loss: 3.1939 - val_acc: 0.2795 - val_categorical_accuracy: 0.2795

Out[38]: <keras.callbacks.History at 0x12e539c88>
```

```
In [42]: score = model.evaluate(x_test, y_test, verbose=1)
         print('Test loss:', score[0])
         print('Test accuracy:', score[1])

         10000/10000 [==============================] - 72s 7ms/step
         Test loss: 3.193865679168701
         Test accuracy: 0.2795
```

Further Work

Due to some issues on loading data and running time, it's obvious that our model is underfitting data. To improve the accuracy and performance of our model, we could add more epochs and steps per epoch. Also we will do more experiments and optimize our model's architecture.

2. Using Autokeras to tune hyperparameters

Introduction of autokeras:

Auto-Keras is an open source software library for automated machine learning. Auto-Keras provides functions to automatically search for architecture and hyperparameters of deep learning models.

Result

```
+---------------------------------------------+
|               Training model 0              |
+---------------------------------------------+

Saving model.
+-------------------------------------------------------------------------+
|        Model ID        |          Loss          |      Metric Value     |
+-------------------------------------------------------------------------+
|           0            |    16.1410089969635    |  0.11520000000000001  |
+-------------------------------------------------------------------------+


+---------------------------------------------+
|               Training model 1              |
+---------------------------------------------+

Saving model.
+-------------------------------------------------------------------------+
|        Model ID        |          Loss          |      Metric Value     |
+-------------------------------------------------------------------------+
|           1            |   12.222918558120728   |         0.296         |
+-------------------------------------------------------------------------+


+---------------------------------------------+
|               Training model 2              |
+---------------------------------------------+

Saving model.
+-------------------------------------------------------------------------+
|        Model ID        |          Loss          |      Metric Value     |
+-------------------------------------------------------------------------+
|           2            |   12.269683456420898   |  0.29240000000000005  |
+-------------------------------------------------------------------------+
```

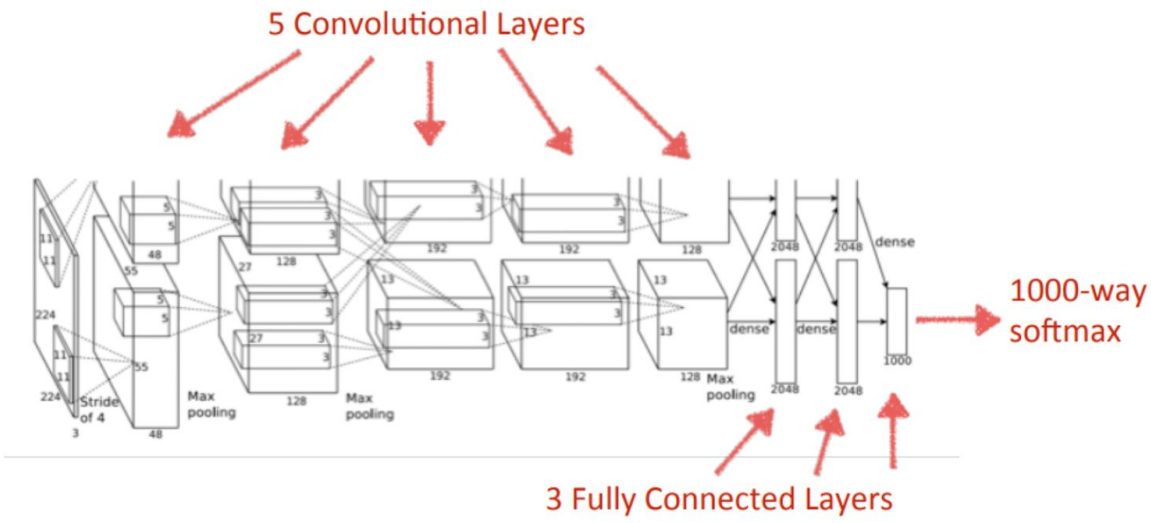
Finally, we get prediction accuracy as 0.0054753608760577405.

Conclusion

Due to early stop, our model is definitely a under fitting model. To improve the accuracy and performance of our model, we could add more epochs and steps per epoch. Also we will continue fit auto-keras model to get a more accurate experiment.

3. Alexnet models in Keras

The model architecture



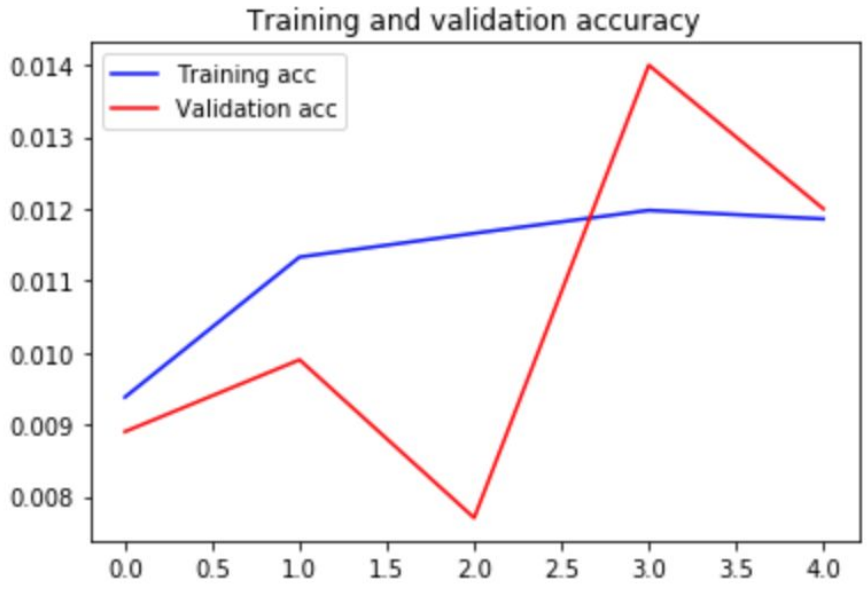
It contains eight learned layers — five convolutional and three fully-connected.
The first convolutional layer filters the 224 × 224 × 3 input image with 96 kernels of size 11 × 11 × 3with a stride of 4 pixels. The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size 5 × 5 × 48. The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size 3 × 3 ×256 connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size 3 × 3 × 192 , and the fifth convolutional layer has 256 kernels of size 3 × 3 × 192. The fully-connected layers have 4096 neurons each.

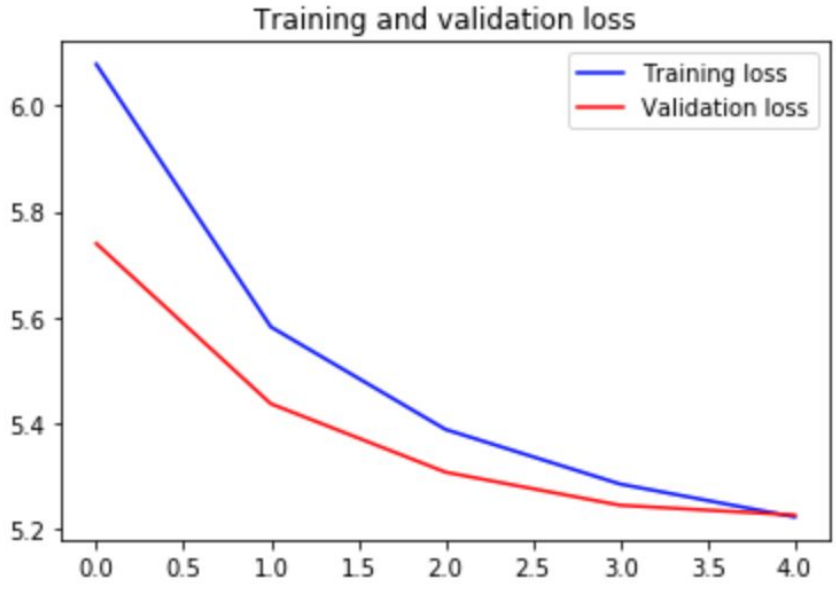
```
model.add(Conv2D(filters=96, input_shape=(32,32,3), kernel_size=(11,11), strides=(4,4), padding='same'))
model.add(Activation('relu'))
# Max pooling
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='same'))


# The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutiona

model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same'))
model.add(Activation('relu'))
# Max Pooling
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='same'))


# The third convolutional layer has 384 kernels of size 3 × 3 × 256 connected to the (normalized, pooled) outputs of

model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))


# The fourth convolutional layer has 384 kernels of size 3 × 3 × 192

model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))


# The fifth convolutional layer has 256 kernels of size 3 × 3 × 192

model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same'))
model.add(Activation('relu'))
# Max Pooling
model.add(MaxPooling2D(pool_size=(3,3), strides=(2,2), padding='same'))


# The fully-connected layers have 4096 neurons each
model.add(Flatten())
model.add(Dense(4096))
model.add(Activation('relu'))
model.add(Dropout(0.4))


# 2nd Dense Layer
model.add(Dense(4096))
model.add(Activation('relu'))
model.add(Dropout(0.4))

model.add(Dense(num_classes))
model.add(BatchNormalization())
model.add(Activation('softmax'))
```

Result

Finally, we get result: test loss: 5.2257, test accuracy 0.0120

```
Train on 100000 samples, validate on 10000 samples
Epoch 1/5
100000/100000 [==============================] - 5519s 55ms/step - loss: 6.0783 - acc: 0.0094 - val_loss: 5.7391 - va
l_acc: 0.0089
Epoch 2/5
100000/100000 [==============================] - 4627s 46ms/step - loss: 5.5815 - acc: 0.0113 - val_loss: 5.4361 - va
l_acc: 0.0099
Epoch 3/5
100000/100000 [==============================] - 2591s 26ms/step - loss: 5.3876 - acc: 0.0117 - val_loss: 5.3068 - va
l_acc: 0.0077
Epoch 4/5
100000/100000 [==============================] - 2005s 20ms/step - loss: 5.2843 - acc: 0.0120 - val_loss: 5.2442 - va
l_acc: 0.0140
Epoch 5/5
100000/100000 [==============================] - 2052s 21ms/step - loss: 5.2222 - acc: 0.0119 - val_loss: 5.2257 - va
l_acc: 0.0120
```





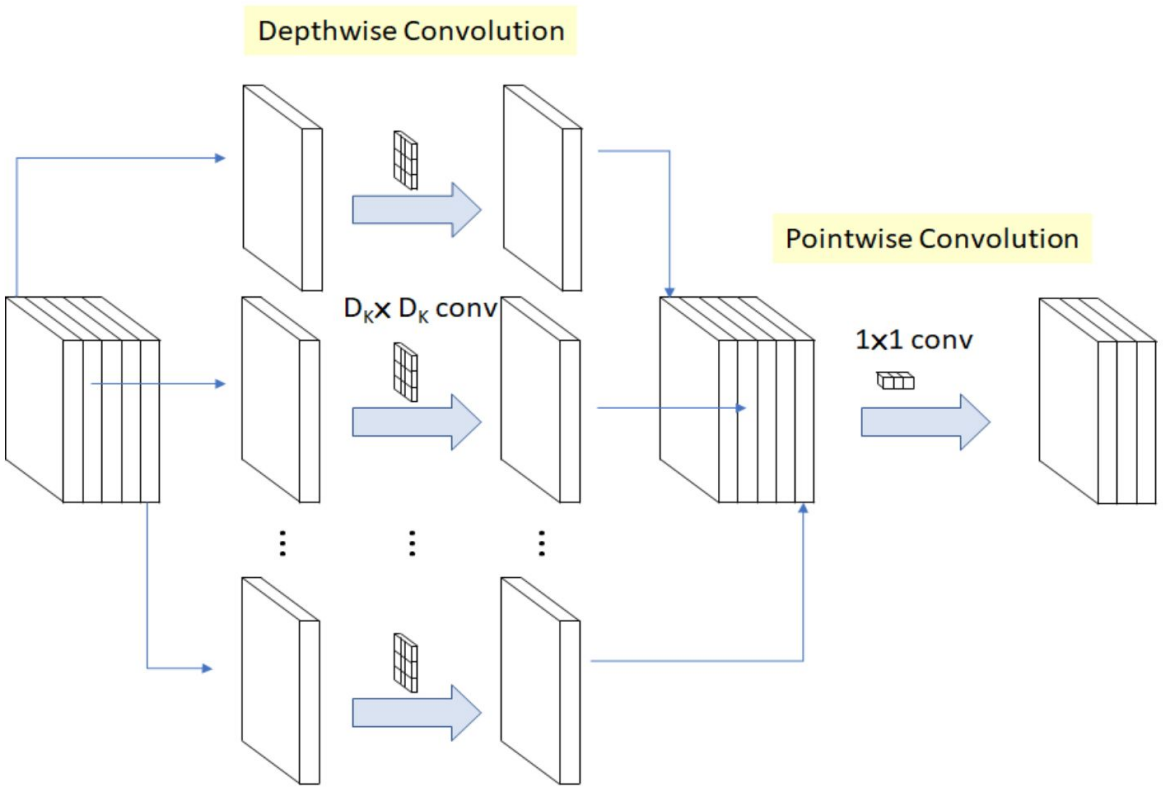
Conclusion

Our model is definitely a under fitting model. One reason might be the data loss when we change image data size from 32,32,3 to 224,224,3. The second reason is lack of epoch fit on our model. To improve the accuracy and performance of our model, we could add more epochs and steps per epoch. Also we will do more experiments and optimize our model's architecture.

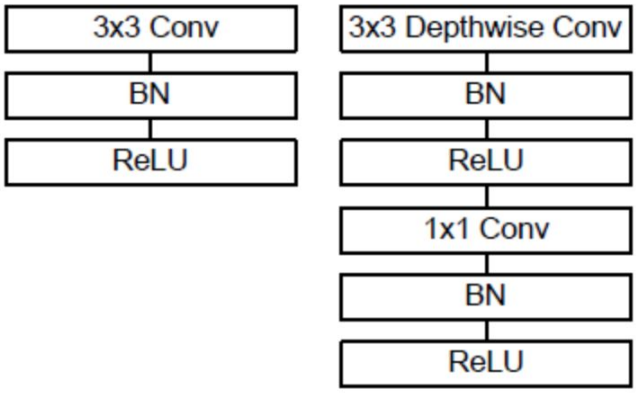
4. MobileNets: Open-Source Models for Efficient On-Device Vision

It has two advantages:
Smaller model size, which leads to fewer number of parameters.
Smaller complexity, which means fewer multiplications and additions.

It is a kind of depth separable convolution



The basic architecture:

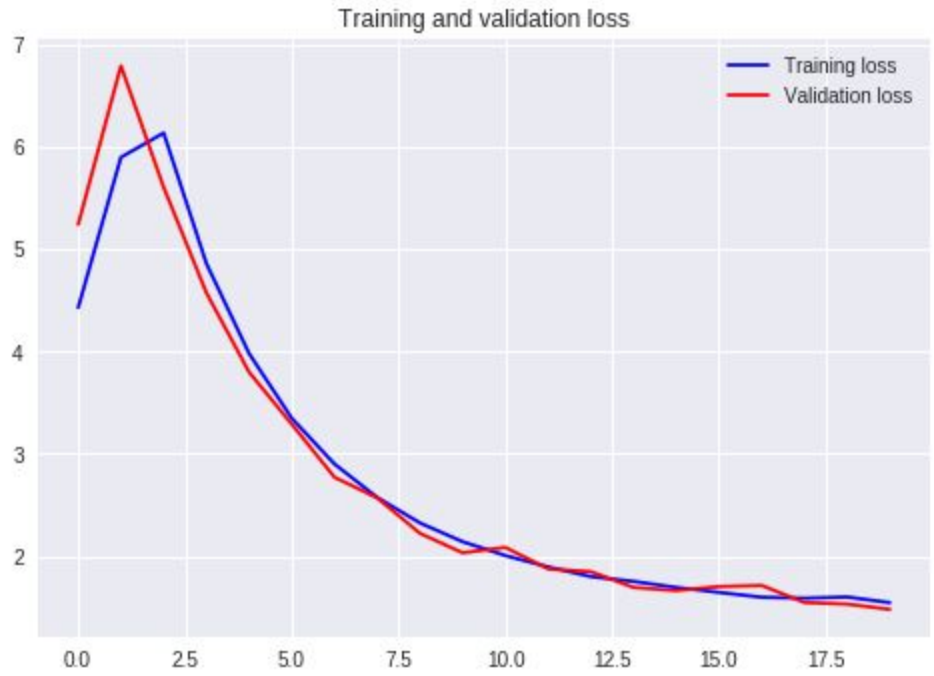


Result

Base MobileNet Model
batch_size = 64
epochs = 20

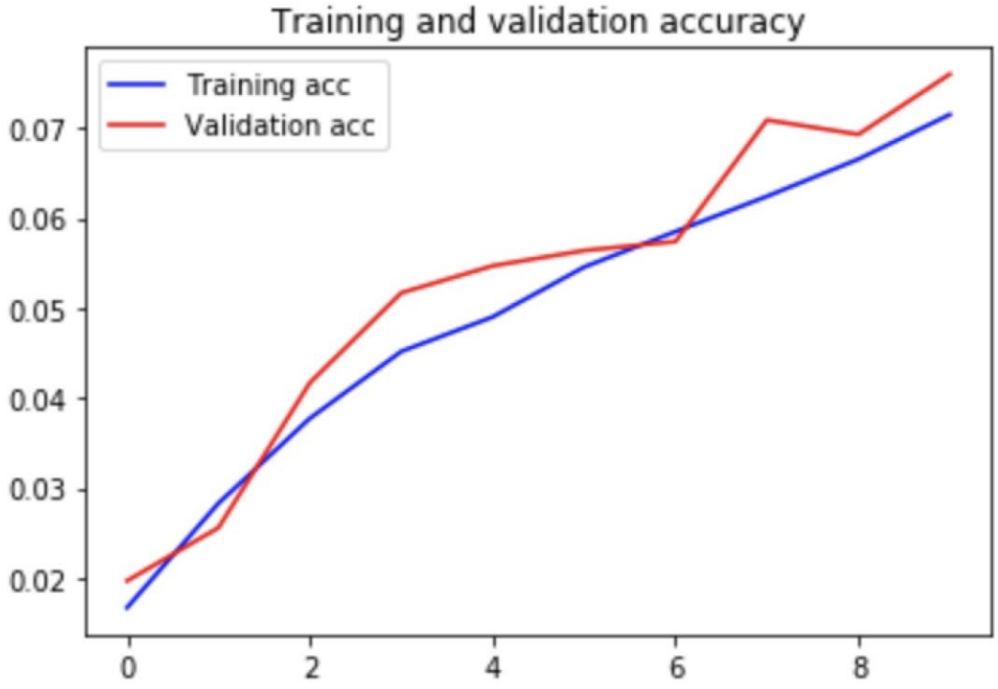
Total params: 3,250,058
Trainable params: 3,228,170
Non-trainable params: 21,888

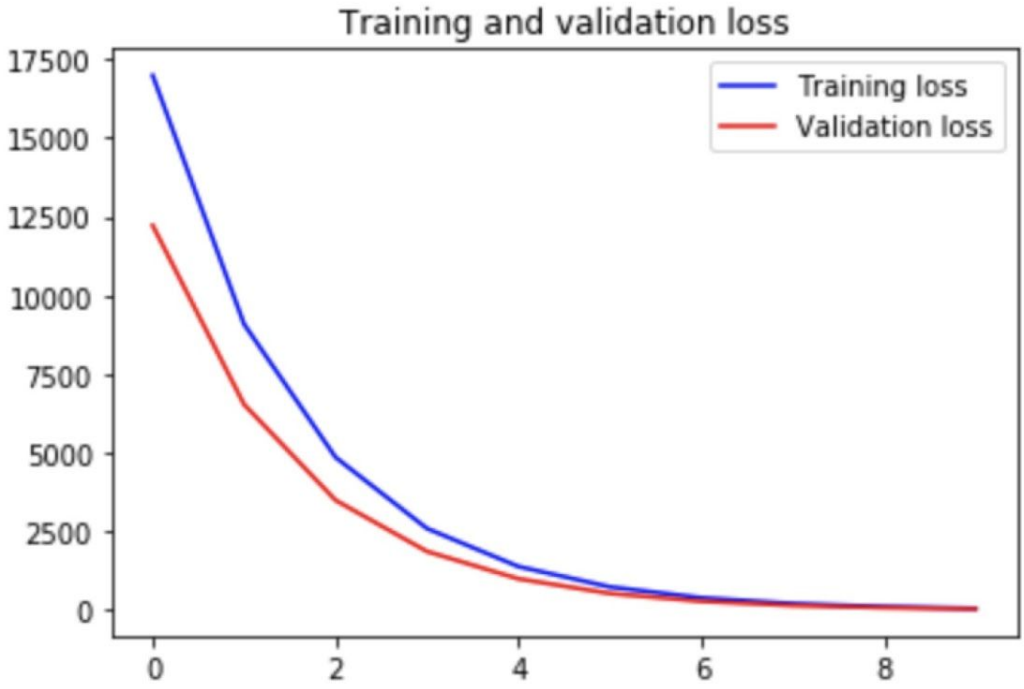
num_classes = 10



Test loss: 1.4796258668899536
Test accuracy: 0.5409

num_classes = 200





Test loss: 48.55855615234375
Test accuracy: 0.076

5. Transfer Learning

The purpose is not to discard the useful information from the previous data, but also to deal with the lack of tags or tag variation caused by data update in the new large amount of data.

Experiment 1:
x = Dense(1024, activation='relu')(x)
x = Dense(1024, activation='relu')(x)
x = Dense(512, activation='relu')(x)
predictions = Dense(num_classes, activation='softmax')(x)

Total params: 4,840,852
Trainable params: 4,818,964
Non-trainable params: 21,888

Test loss: 1.3521895277023315
Test accuracy: 0.6025

Experiment 2:
x = Dense(1024, activation='relu')(x)
x = Dense(1024, activation='relu')(x)
x = Dense(512, activation='relu')(x)
x = Dense(512, activation='relu')(x)
x = Dense(256, activation='relu')(x)

Total params: 5,232,276
Trainable params: 5,210,388
Non-trainable params: 21,888

Test loss: 1.4370202228546143
Test accuracy: 0.5955

Experimrnt 3:
x = Dense(1024, activation='relu')(x)
x = Dense(1024, activation='relu')(x)
x = Dense(512, activation='relu')(x)
x = Dense(512, activation='relu')(x)
x = Dense(256, activation='relu')(x)
x = Dense(256, activation='relu')(x)
x = Dense(128, activation='relu')(x)

Total params: 5,329,684
Trainable params: 5,307,796
Non-trainable params: 21,888

Test loss: 1.582895671272278
Test accuracy: 0.5491

6. Fine Tuning

Use the network model that others have trained to carry out the training. The premise must use the same network with others, because the parameters are based on the network.
The reason why Fine Tuning is effective is that the same network is used, and the parameters used are the data already trained by others, so there is a guarantee of accuracy. In this case, slightly adjusting the parameters trained by others can often achieve the desired effect.

Experiment 1:

```
# Freeze the layers except the last 4 layers
for layer in model.layers[:-4]:
    layer.trainable = False

model_finetune.add(layers.Dense(1024, activation='relu'))
model_finetune.add(layers.Dropout(0.5))
model_finetune.add(layers.Dense(num_classes, activation='softmax'))
```

Total params: 6,467,978
Trainable params: 3,228,170
Non-trainable params: 3,239,808

Test loss: 1.3391967622756957
Test accuracy: 0.604

Experiment 2:

```
# Freeze the layers except the last 3 layers
for layer in model.layers[:-3]:
    layer.trainable = False

model_finetune.add(layers.Dense(1024, activation='relu'))
model_finetune.add(layers.Dense(1024, activation='relu'))
model_finetune.add(layers.Dense(512, activation='relu'))
model_finetune.add(layers.Dropout(0.5))
```

```
model_finetune.add(layers.Dense(num_classes, activation='softmax'))
```

Total params: 6,467,978
Trainable params: 3,228,170
Non-trainable params: 3,239,808

Test loss: 1.3122230989456176
Test accuracy: 0.626

Experiment 3:

```
# Freeze the layers except the last 2 layers
for layer in model.layers[:-2]:
    layer.trainable = False

model_finetune.add(layers.Dense(1024, activation='relu'))
model_finetune.add(layers.Dense(1024, activation='relu'))
model_finetune.add(layers.Dense(512, activation='relu'))
model_finetune.add(layers.Dropout(0.5))
model_finetune.add(layers.Dense(num_classes, activation='softmax'))
```

Total params: 6,467,978
Trainable params: 3,228,170
Non-trainable params: 3,239,808

Test loss: 1.4716102521896361
Test accuracy: 0.573