(i) (a)This is a function:input a H x W array , and the kernel is kh x kw, stride is 1 and no padding, convolves the kernel to the input array and returns the result, inside the 4 iterations get the output array:

```
def conv2d(input_array, kernels, stride=1, padding=0):
    H, W = input_array.shape
    kh, kw = kernels.shape
    p = padding
    out_h = (H + 2 * padding - kh) // stride + 1  # //floor division
    out_w = (W + 2 * padding - kw) // stride + 1  # //floor division
    outputs = np.zeros([  out_h, out_w])
    for h in range(out_h):
        for w in range(out_w):
            for x in range(kh):
                for y in range(kw):
                    outputs[h][w] += input_array[h * stride + x][w * stride + y] *
kernels[x][y]
    return outputs
```

And I set the input is 5 x 5 :

```
[1, 2, 3, 4, 5],
[2, 3, 4, 5, 6],
[3, 4, 5, 6, 7],
[4, 5, 6, 7, 8],
[5, 6, 7, 8, 9]
```

,the kernel is 3x3:

```
[1, 1, 1],
[1, 1, 1],
[1, 1, 1]
```

,the output is 3x3:

```
[[27. 36. 45.]
 [36. 45. 54.]
 [45. 54. 63.]]
```

(b)I choose a 200x200 picture:

And the output of *array of R pixels*

For the first kernel: The result matrix and image:

```
[[   2.   -1.    1. ...   -3.   14.   15.]
 [   5.    3.    3. ...    0.   10.   22.]
 [   0.   -3.   -5. ...   -3.   -9.   22.]
 ...
 [ 111.  -59. -151. ...   23.  -48.   21.]
 [  98.  -66.  143. ...   43.  -36.   28.]
 [ 192.  -34.  361. ...   49.  -49.   26.]]
```

For the second kernel:

```
[[ 920.  920.  920. ...  751.  765.  757.]
 [ 922.  921.  921. ...  756.  764.  763.]
 [ 916.  915.  914. ...  755.  749.  763.]
 ...
 [ 794.  622.  405. ...  780.  731.  787.]
 [ 829.  673.  744. ...  794.  741.  789.]
 [ 957.  708. 1001. ...  798.  727.  784.]]
```

(ii)(b)

(i) `(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()`

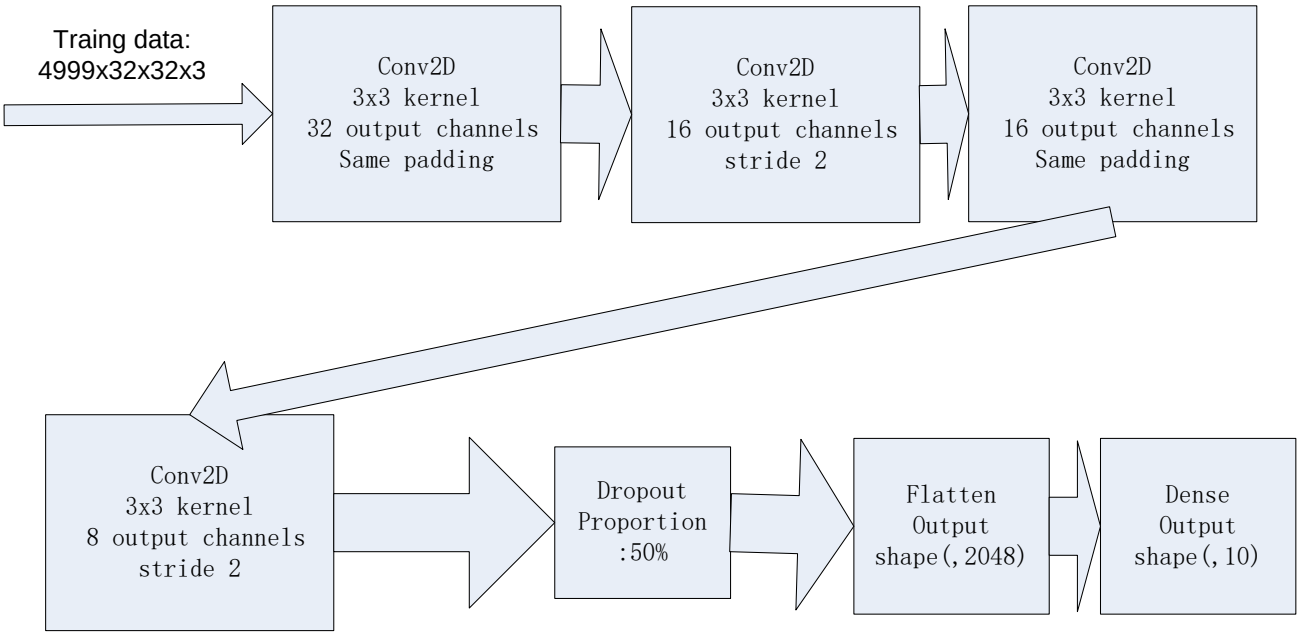
x_train, x_test: uint8 arrays of RGB image data with shape (num_samples, 3, 32, 32) if tf.keras.backend.image_data_format() is 'channels_first', or (num_samples, 32, 32, 3) if the data format is 'channels_last', as in the code shows the num_samples=5000.

`padding='same'`:pad original input then apply kernel → output is same size as input

model.add(Dropout(0.5)) : neurons are randomly deleted in the proportion of 50% to prevent the occurrence of over-fitting.

If input is output from a convolution layer, i.e. a tensor, need to flatten it before it can be used as input to FC layer.That is: flattening → take all elements of tensor and write them as a list/array.

I use microsoft visio to draw the architecture of the ConvNet:



(ii)

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 16)        448
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 16)        2320
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 32)        4640
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 32)          9248
_________________________________________________________________
dropout (Dropout)            (None, 8, 8, 32)          0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 10)                20490
=================================================================
Total params: 37,146
Trainable params: 37,146
Non-trainable params: 0
```

This model has 37146 params, dense layer has the most parameters the number is 20490, dense layer in keras is one layer of MLP, each output is a function of a weighted sum of all of the inputs so a FC-layer can have multiple outputs. If input vector x has n elements and have m outputs then FC-layer has n × m parameters. And this is multiplication, so it has the mose parameters.

For training data:

	precision	recall	f1-score	support
0	0.63	0.49	0.55	505
1	0.73	0.71	0.72	460
2	0.51	0.55	0.53	519
3	0.58	0.44	0.50	486
4	0.51	0.58	0.54	519

```
           5       0.61      0.55      0.58       488
           6       0.72      0.63      0.67       518
           7       0.73      0.61      0.66       486
           8       0.51      0.88      0.64       520
           9       0.73      0.64      0.68       498

    accuracy                           0.61      4999
   macro avg       0.63      0.61      0.61      4999
weighted avg       0.62      0.61      0.61      4999
```

For test data:

```
              precision    recall  f1-score   support

           0       0.48      0.38      0.42      1000
           1       0.65      0.62      0.63      1000
           2       0.40      0.44      0.42      1000
           3       0.38      0.28      0.32      1000
           4       0.39      0.44      0.42      1000
           5       0.46      0.38      0.41      1000
           6       0.58      0.52      0.55      1000
           7       0.62      0.52      0.56      1000
           8       0.42      0.80      0.55      1000
           9       0.60      0.51      0.55      1000

    accuracy                           0.49     10000
   macro avg       0.50      0.49      0.48     10000
weighted avg       0.50      0.49      0.48     10000
```

test data accuracy rate is about 0.50, the training data accuracy rate is about 0.61, and the model of training data performs much better than test data.

Choose DummyClassifier as a baseline model:

```
from sklearn.dummy import  DummyClassifier
dum=DummyClassifier(strategy="most_frequent").fit(x_train,y_train)
ydum=dum.predict(x_test)
ydum_pre=np.argmax(ydum,axis=1)
print(confusion_matrix(y_test1,ydum_pre))
print(classification_report(y_test1,ydum_pre))
```

And the baseline model accuracy rate is 0.1 very low , it’s the model that always predicts at random, the result confusion matrix is :

```
              precision    recall  f1-score   support

           0       0.10      1.00      0.18      1000
           1       0.00      0.00      0.00      1000
           2       0.00      0.00      0.00      1000
           3       0.00      0.00      0.00      1000
           4       0.00      0.00      0.00      1000
           5       0.00      0.00      0.00      1000
           6       0.00      0.00      0.00      1000
           7       0.00      0.00      0.00      1000
           8       0.00      0.00      0.00      1000
           9       0.00      0.00      0.00      1000
```

accuracy			0.10	10000
macro avg	0.01	0.10	0.02	10000
weighted avg	0.01	0.10	0.02	10000

(ii)

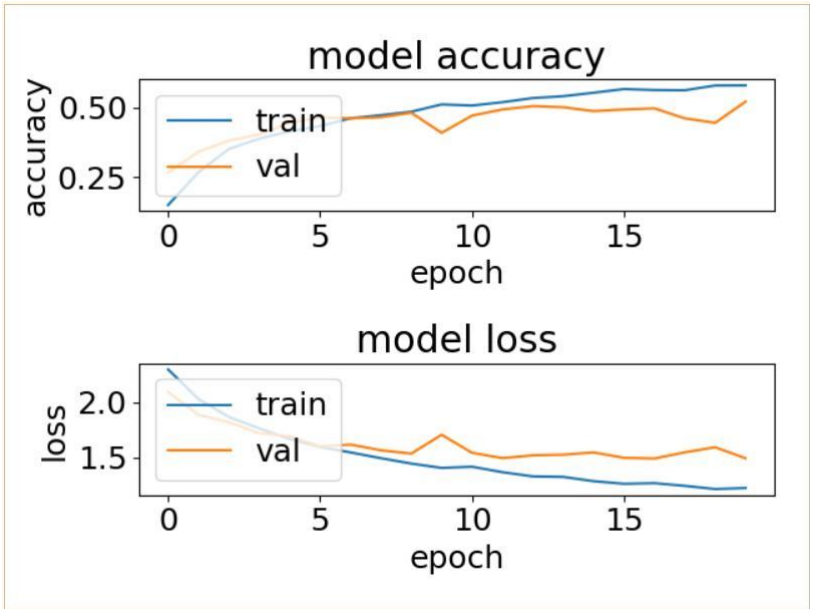


Figure the loss and accuracy of training data and validation data vary as epoch goes

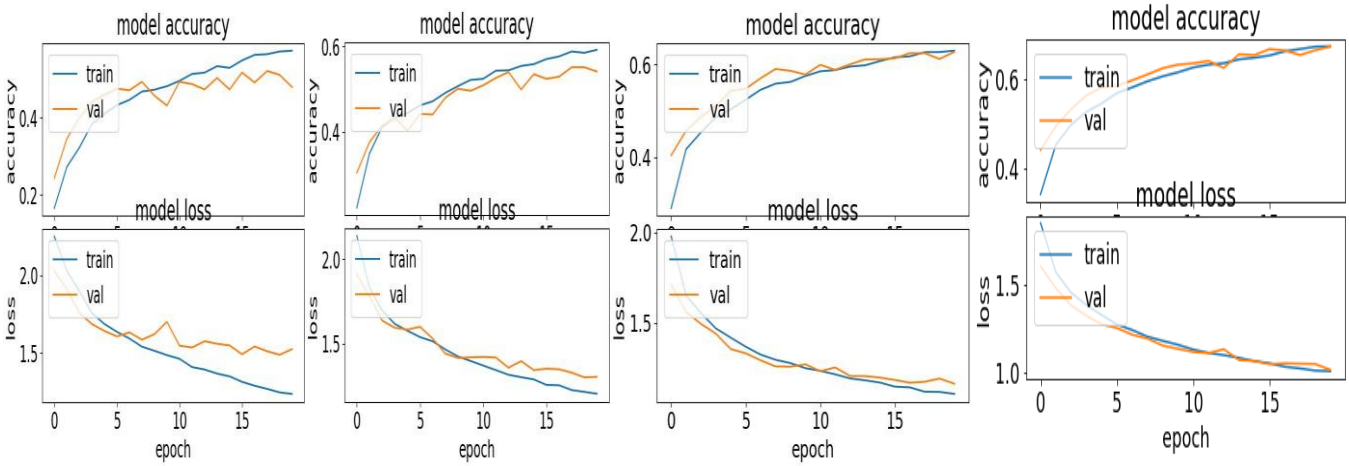
As epoch goes, the accuracy of both data rises, the loss of both models decreases, however, when epoch=8, the accuracy of validation drops and loss of it increases, as epoch goes the model even become worse, this means the model is over-fitting, because it fits the training data too well that it's unable to generalize for validation data.

(iii)

I use google colab using GPU to train the data, and it records the time automatically, the result is as follows:

training data size(K)	time(s)
5	8.436
10	12.102
20	19.553
40	36.318

And the plot of loss and accuracy of training data and validation data vary as epoch goes for training data size 5K, 10K, 20K, 40K is as follows:



We can see from above as the number of training data increases the time of training the model increases, also the accuracy and loss of training data and validation data respectively are becoming closer and closer, they all have the same trend that as the epoch goes up the accuracy of each model rises and the loss

drops, but take it into a closer look, when epoch goes to a certain value the accuracy of validation data drops and the loss of that increases, the complexity of the model increases such that the training data loss reduces but the validation data loss doesn't, which means the model becomes over-fitting, and as the number training data increases the value of the epoch that makes the model over-fitting becomes bigger too.
All in all , to prevent overfitting, the best solution is to use more training data. A model trained on more data will naturally generalize better.

(iv)I choose the range of weight is 0, 0.0001, 0.001,0.1, 1, 10

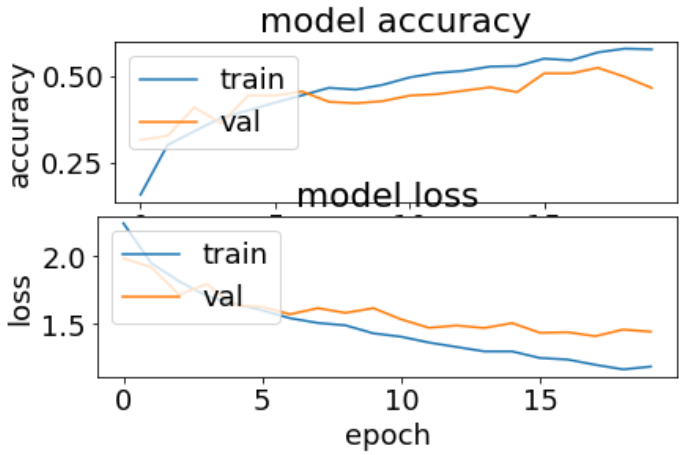


Figure 1 weight=0, accuracy for training data:0.62 and validation data:0.50

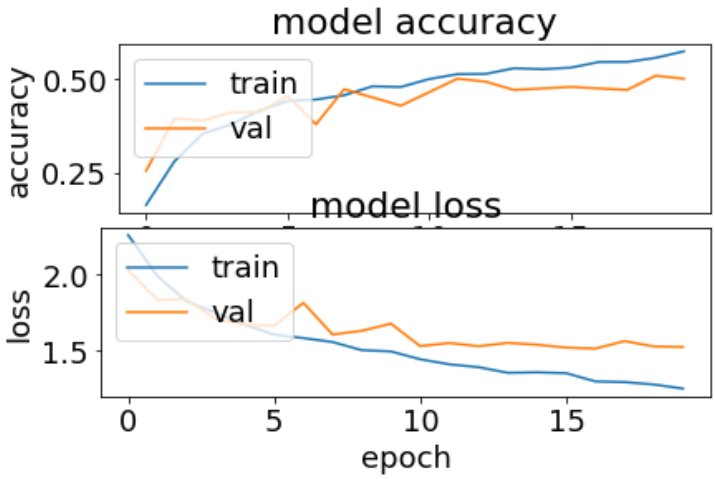


Figure 2 weight=0.0001, accuracy for training data:0.61 and validation data:0.50

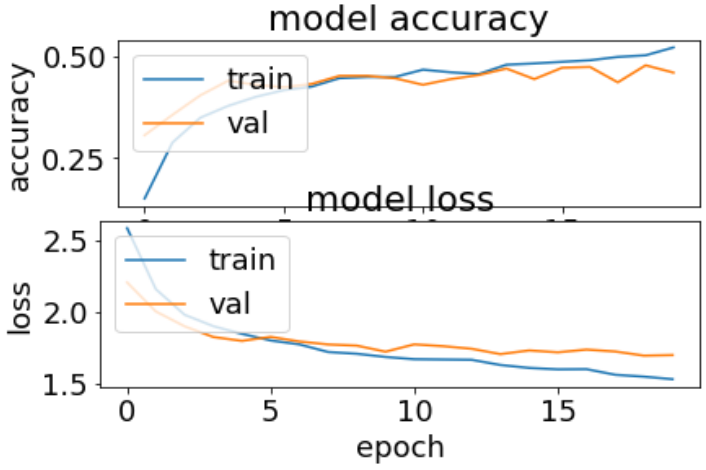


Figure 3 weight=0.001, accuracy for training data:0.54 and validation data:0.47

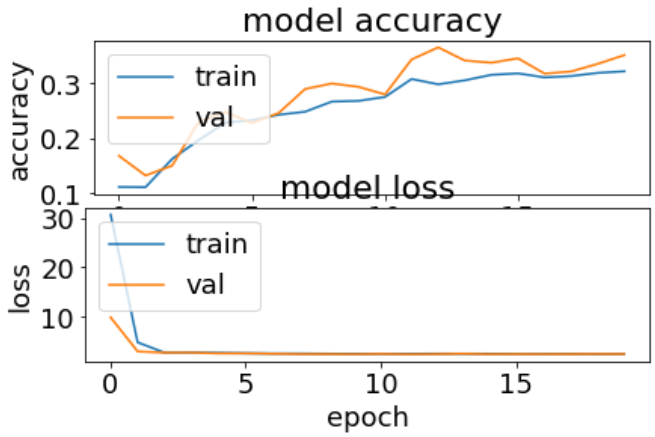


Figure 4 weight=0.1, accuracy for training data:0.34 and validation data:0.32

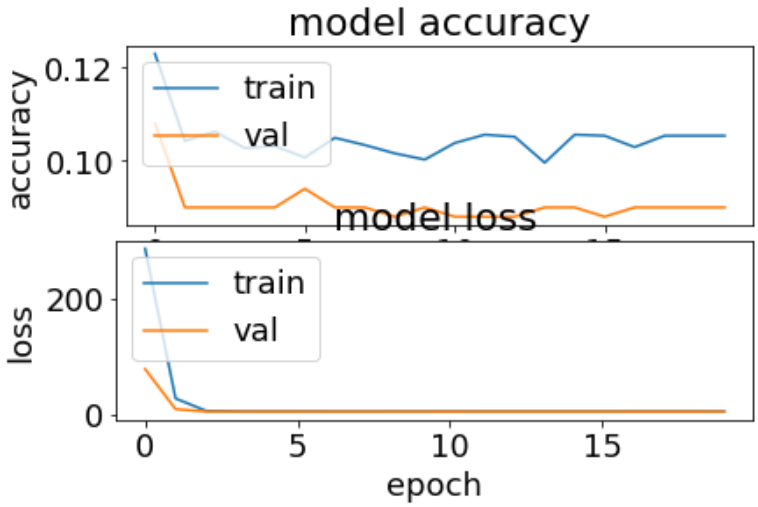


Figure 5 weight=1, accuracy for training data:0.1 and validation data:0.1

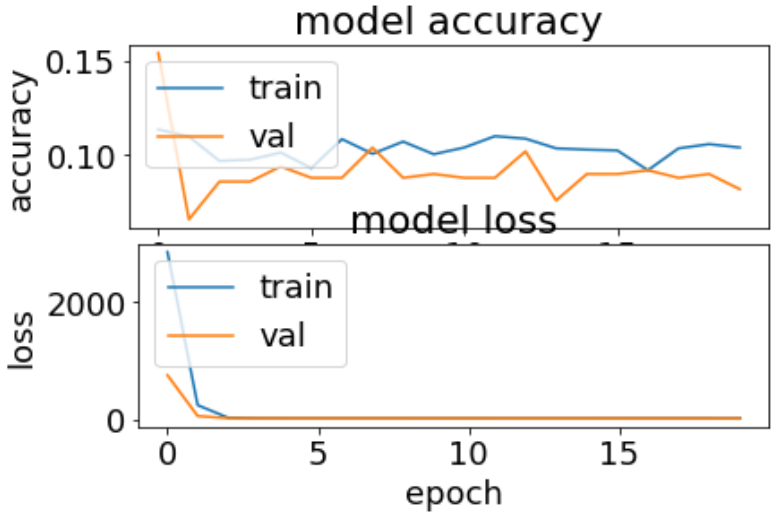


Figure 6 weight=10, accuracy for training data:0.09 and validation data:0.09

As we can see from above, as the L1 weight gets bigger the accuracy of the model drops , the reason for the worse performance is that the model gets over-fitting because as the complexity of the model increases such that the training loss reduces but the validation data loss doesn't.

When weight=0 the accuracy of the training data drops when epoch=17 while the loss of that drops, which means the model is under-fitting, but as weight gets bigger such as weight=0.1 it's over-fitting, so when wd add a reasonable small L2 regulation the models gets better.

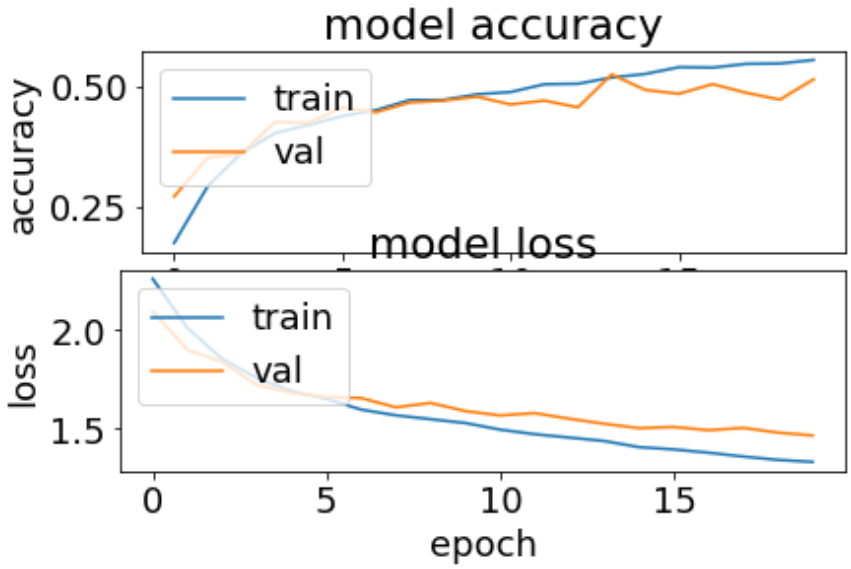
(c)(i)Change the code:

```
model = keras.Sequential()
model.add(Conv2D(16, (3,3), padding='same', input_shape=x_train.shape[1:],activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(32, (3,3), padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.5))
```

The new keras parameters structure:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_52 (Conv2D)           (None, 32, 32, 16)        448
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 16)        0
_________________________________________________________________
conv2d_53 (Conv2D)           (None, 16, 16, 32)        4640
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 32)          0
_________________________________________________________________
dropout_13 (Dropout)         (None, 8, 8, 32)          0
_________________________________________________________________
flatten_13 (Flatten)         (None, 2048)              0
_________________________________________________________________
dense_13 (Dense)             (None, 10)                20490
=================================================================
Total params: 25,578
Trainable params: 25,578
Non-trainable params: 0
```

(ii)




```
Total params: 25,578
Trainable params: 25,578
Non-trainable params: 0
```

The accuracy of training data:0.6 validation data:0.51

The time for training this model is 7.668s, comparing to previous ConvNet, this model has fewer parameters, and thus less training time. The accuracy of the two is close, but the new one is better, and costs less time , and not easily get over-fitting as the old one.

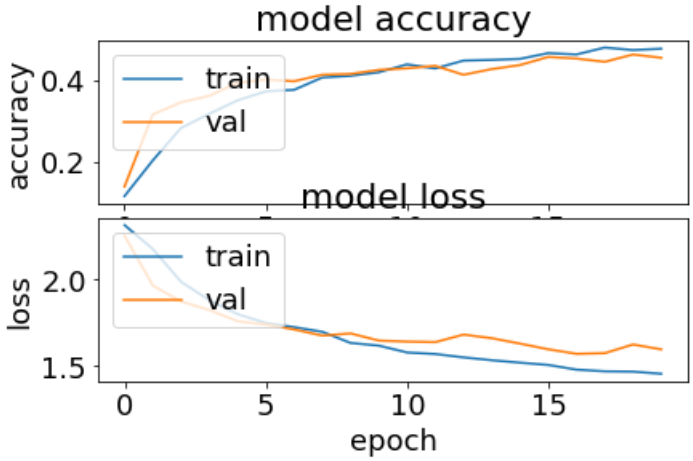
----------(optional)----------

The running time of the model is 7.983s, which is faster than the old one

The new optional keras parameters structure:

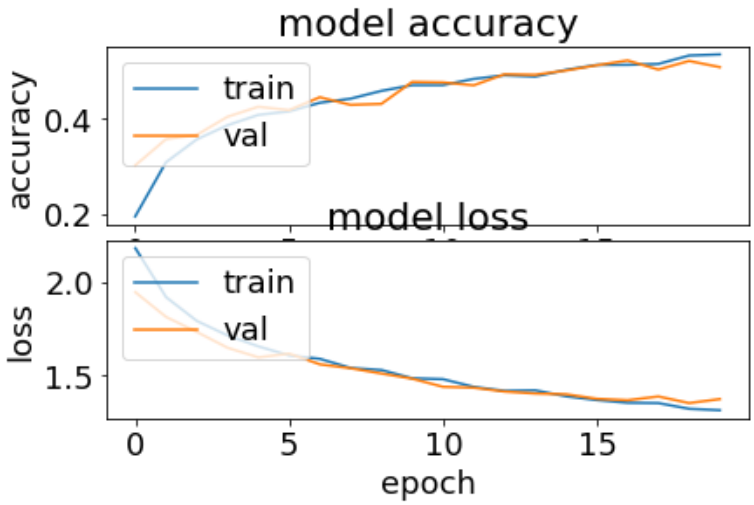
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_54 (Conv2D)           (None, 32, 32, 8)         224
_________________________________________________________________
conv2d_55 (Conv2D)           (None, 16, 16, 8)         584
_________________________________________________________________
conv2d_56 (Conv2D)           (None, 16, 16, 16)        1168
_________________________________________________________________
conv2d_57 (Conv2D)           (None, 8, 8, 16)          2320
_________________________________________________________________
conv2d_58 (Conv2D)           (None, 8, 8, 32)          4640
_________________________________________________________________
conv2d_59 (Conv2D)           (None, 4, 4, 32)          9248
_________________________________________________________________
dropout_14 (Dropout)         (None, 4, 4, 32)          0
_________________________________________________________________
flatten_14 (Flatten)         (None, 512)               0
_________________________________________________________________
dense_14 (Dense)             (None, 10)                5130
=================================================================
Total params: 23,314
Trainable params: 23,314
Non-trainable params: 0
```

The number of parameters of this new model is 23314, which is also less than the original one.

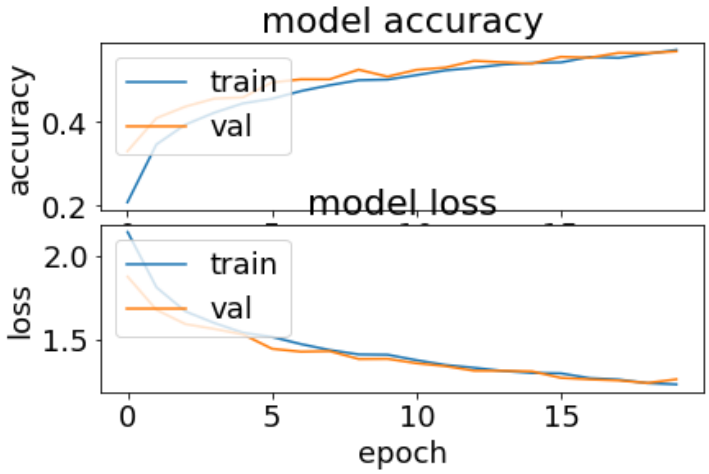


The plot of adding more layers while data size:5K

The accuracy of training data:0.51 validation data:0.45



The plot of adding more layers while training data size:10K
The accuracy of training data:0.57 validation data:0.50



The plot of adding more layers while training data size:20K
The accuracy of training data:0.61 validation data:0.57

Comparing the plot to the original one, the plot get flatter than the old one, but the accuracy is lower, and time costs less too, which means this model has the possibility of underfitting. And as the size of training data gets bigger, the accuracy gets higher while the trend of the plots still very similar, which means the model is under-fitting.
So adding more layers make the model underfitting and less training time and poorer model.

Appendix:

```
import numpy as np
def conv2d(input_array, kernels, stride=1, padding=0):
    H, W = input_array.shape
    kh, kw = kernels.shape
    p = padding
    out_h = (H + 2 * padding - kh) // stride + 1  # //floor division
    out_w = (W + 2 * padding - kw) // stride + 1  # //floor division
    outputs = np.zeros([  out_h, out_w])
    for h in range(out_h):
        for w in range(out_w):
            for x in range(kh):
                for y in range(kw):
                    outputs[h][w] += input_array[h * stride + x][w * stride + y] *
kernels[x][y]
    return outputs
from PIL import Image
im=Image.open('a.jpg')
rgb=np.array(im.convert('RGB'))
r=rgb [ : , : , 0 ] # array of R pixels
Image.fromarray(np. uint8 ( r ) ).show ()
kernel1 = np.array(
        [
            [-1, -1, -1],
            [-1, 8, -1],
            [-1, -1, -1]
        ]
    ).reshape(3,3)
kernel2 = np.asarray(
        [
            [0, -1, 0],
            [-1, 8, -1],
            [0, -1, 0]
        ]
    )
output1=conv2d(r,kernel1)
output2=conv2d(r,kernel2)
print(output1)
Image.fromarray(np. uint8 ( output1 ) ).show ()
print(output2)
Image.fromarray(np. uint8 ( output2 ) ).show ()
```