

# Machine Learning

## Classification of CIFAR-10

과 목	머신러닝
분 반	월4, 수3
담당교수	이혁준 교수님
학 과	컴퓨터공학과
학 번	2018202007
성 명	최예진
날 짜	2021. 06. 14 (월)



광운대학교  
KwangWoon University

## Introduction

- CIFAR-10 dataset 을 input 으로 한 classification model 의 성능을 올리는 프로젝트이다.  
model2 의 경우, 주어진 structure 에 맞게 코드를 구성하고, model3 의 경우 convolution layer 와 pooling layer 의 개수 등등을 자유롭게 구성하여 최소 73%의 정확도를 갖는 모델을 만든다.

## Implementation

### A. model 2

#### 1. Convolution layer1 + Max Pooling

```

14 # Conv1 + Pooling1
15 with tf.variable_scope('conv1') as scope:
16     conv = tf.layers.conv2d(
17         inputs=x_image,
18         filters=64,
19         kernel_size=[5, 5],
20         padding='SAME',
21         activation=tf.nn.relu
22     )
23     pool = tf.layers.max_pooling2d(conv, pool_size=[2, 2], strides=2, padding='SAME')
24 
```

#### 2. Convolution layer2 + Max Pooling

```

25 # Conv2 + Pooling2
26 with tf.variable_scope('conv2') as scope:
27     conv = tf.layers.conv2d(
28         inputs=pool,
29         filters=64,
30         kernel_size=[5, 5],
31         padding='SAME',
32         activation=tf.nn.relu
33     )
34     pool = tf.layers.max_pooling2d(conv, pool_size=[2, 2], strides=2, padding='SAME')
35 
```

#### 3. Convolution layer3

```

36 # Conv3
37 with tf.variable_scope('conv3') as scope:
38     conv = tf.layers.conv2d(
39         inputs=pool,
40         filters=128,
41         kernel_size=[3, 3],
42         padding='SAME',
43         activation=tf.nn.relu
44     )
45 
```

## 4. Convolution layer4

```

46 # Conv4
47 with tf.variable_scope('conv4') as scope:
48     conv = tf.layers.conv2d(
49         inputs=conv,
50         filters=128,
51         kernel_size=[3, 3],
52         padding='SAME',
53         activation=tf.nn.relu
54     )
55 
```

## 5. Convolution layer5

```

56 # Conv5
57 with tf.variable_scope('conv5') as scope:
58     conv = tf.layers.conv2d(
59         inputs=conv,
60         filters=128,
61         kernel_size=[3, 3],
62         padding='SAME',
63         activation=tf.nn.relu
64     )
65 
```

## 6. Fully connected layer

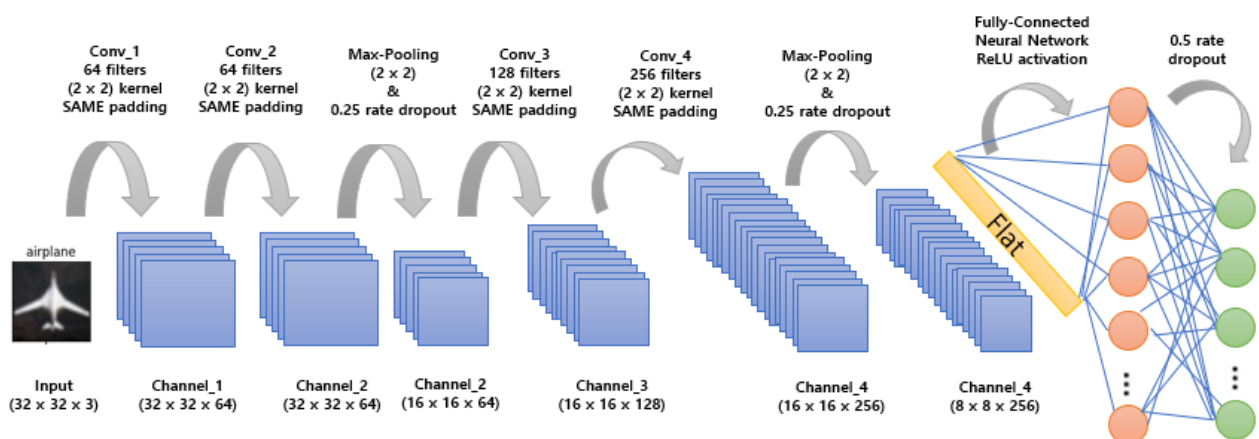
```

66 # Fully connected
67 with tf.variable_scope('fully_connected') as scope:
68     flat = tf.reshape(conv, [-1, 8 * 8 * 128])
69
70     fc = tf.layers.dense(inputs=flat, units=1024, activation=tf.nn.relu)
71     drop = tf.layers.dropout(fc, rate=0.5)
72     softmax = tf.layers.dense(inputs=drop, units=NUM_CLASSES, activation=tf.nn.softmax, name=scope.name)
73
74     y_pred_cls = tf.argmax(softmax, axis=1)
75
76     return x, y, softmax, y_pred_cls, global_step, learning_rate
77 
```

→ model2 는 model 을 base code 로 하여 구현했으며, pooling 개수를 고려해 flat 함수를 호출하였다.

## B. model 3

## 1. model3 그림



## 2. 성능 향상 시도 결과

가장 처음으로는, model2 의 코드에서 convolution layer 의 마지막 줄 마다 batch normalization 을 적용한 뒤, epoch 를 20 으로 하여 시뮬레이션 해보았다. 그랬더니 아래와 같이 약 50%의 정확도를 확인할 수 있었다.

```
Global step: 3905 - [=====>---] 92% - acc: 0.4766 - loss: 1.9847 - 8994.2 samp
Global step: 3915 - [=====>---] 97% - acc: 0.5938 - loss: 1.8745 - 9301.3 samp
#####
Test ACC Rate --> 56.45 %
main
airplane automobile cat deer dog frog horse ship truck
0 0 0 0 0 0 0 0 0 0
```

epoch 를 50 으로 하면 정확도가 얼마나 높아질 지 궁금하여 epoch 를 50 으로 하여 simulation 해보았더니 10%로, 성능이 오히려 매우 낮아졌다.

```
Global step: 9775 - [=====>---] 87% - acc: 0.0898 - loss: 2.3713 - 9824.2 sample/sec
Global step: 9785 - [=====>---] 92% - acc: 0.1055 - loss: 2.3557 - 9470.8 sample/sec
Global step: 9795 - [=====>---] 97% - acc: 0.0820 - loss: 2.3791 - 9576.8 sample/sec
#####
Test ACC Rate --> 10.0 %
main
airplane automobile cat deer dog frog horse ship truck
0 0 0 0 0 0 0 0 0 0
```

model2 의 코드를 base 로 사용하지 않는 것이 좋겠다고 생각하여 model 의 코드를 사용하기로 했다. layer 가 깊을수록 정확도가 높아질 것이라고 생각하여 convolution layer 를 하나 더 추가하고 epoch 를 20 으로 하여 simulation 해봤더니, 이전의 결과와 다르게 70%정도의 정확도를 확인할 수 있었다.

```
Global step: 3905 - [=====>---] 92% - acc: 0.7852 - loss: 1.6759 - 8929.6 sample/sec
Global step: 3915 - [=====>---] 97% - acc: 0.8164 - loss: 1.6430 - 9020.8 sample/sec
#####
Test ACC Rate --> 70.8 %
main
airplane automobile cat deer dog frog horse ship truck
0 0 0 0 0 0 0 0 0 0
```

위의 상태에서 pooling layer 를 한 개 제거한 뒤, epoch 를 20 으로 하여 simulation 하였더니 2%가 올라서 과제의 조건인 72%를 조금 넘는 정확도를 얻을 수 있었다.

```
Global step: 3885 - [=====>---] 82% - acc: 0.7878 - loss: 1.6981 - 9734.7 sample/sec
Global step: 3895 - [=====>---] 87% - acc: 0.7695 - loss: 1.6887 - 9544.0 sample/sec
Global step: 3905 - [=====>---] 92% - acc: 0.8086 - loss: 1.6539 - 9591.7 sample/sec
Global step: 3915 - [=====>---] 97% - acc: 0.7969 - loss: 1.6645 - 9655.3 sample/sec
#####
Test ACC Rate --> 72.31 %
main
airplane automobile cat deer dog frog horse ship truck
0 0 0 0 0 0 0 0 0 0
```

위의 상태에서 Batch size 를 늘리면 조금 더 높은 정확도를 얻게 될 거라고 생각해서 256 -> 512 로 size 를 키워서 simulation 해보았더니, 아주 약간 오른 정확도인 73%의 정확도를 확인할 수 있었다.

```
Global step: 1943 - [=====>---] 82% - acc: 0.8086 - loss: 1.6501 - 9762.8 sample/sec
Global step: 1953 - [=====>---] 92% - acc: 0.7812 - loss: 1.6768 - 9680.1 sample/sec
#####
Test ACC Rate --> 73.11999999999999 %
main
airplane automobile cat deer dog frog horse ship truck
0 0 0 0 0 0 0 0 0 0
```

이 상태에서 정확도를 더 높이고 싶어서, filter 개수를 128->256 개로 늘린 뒤 epoch 값을 20 으로 하여 simulation 하였더니 아래와 같이 76.42 프로의 정확도를 확인할 수 있었다. 다만, filter 의 개수를 2 배로 키운 결과로서 느린 시뮬레이션 시간이 단점으로 작용했다는 것을 알 수 있었다.

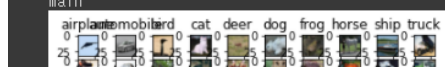
```
Global step: 3895 - [=====>-----] 87% - acc: 0.9258 - loss: 1.5323 - 6528.3 sample/sec
Global step: 3905 - [=====>-----] 92% - acc: 0.8906 - loss: 1.5737 - 6511.6 sample/sec
Global step: 3915 - [=====>-----] 97% - acc: 0.9102 - loss: 1.5435 - 6197.1 sample/sec
#####
Test ACC Rate --> 76.42 %
main
```

위의 상태에서 epoch 를 50 으로 하였더니 약 80%의 정확도를 얻을 수 있었다.

```
Global step: 3745 - [=====>-----] 77% - acc: 0.9414 - loss: 1.5132 - 6333.4 sample/sec
Global step: 3755 - [=====>-----] 77% - acc: 0.9648 - loss: 1.4958 - 6283.7 sample/sec
Global step: 3765 - [=====>-----] 82% - acc: 0.9453 - loss: 1.5150 - 6391.1 sample/sec
Global step: 3775 - [=====>-----] 87% - acc: 0.9414 - loss: 1.5191 - 6166.5 sample/sec
Global step: 3785 - [=====>-----] 92% - acc: 0.9375 - loss: 1.5233 - 6130.8 sample/sec
Global step: 3795 - [=====>-----] 97% - acc: 0.9648 - loss: 1.4958 - 6331.2 sample/sec
#####
Test ACC Rate --> 79.49000000000001 %
main
```

convolution layer 를 1 개 더 늘리고, filter 의 size 를 키운 뒤, epoch 값을 20 으로 하여 시뮬레이션 했더니 오히려 더 낮은 정확도를 확인할 수 있었다. convolution layer 를 많이 쌓는다고 정확도가 무조건 오르는게 아니라는 것을 알 수 있었다.

```
Global step: 3895 - [=====>-----] 87% - acc: 0.8594 - loss: 1.6041 - 7247.5 sample/sec
Global step: 3905 - [=====>-----] 92% - acc: 0.8008 - loss: 1.6636 - 7260.1 sample/sec
Global step: 3915 - [=====>-----] 97% - acc: 0.8672 - loss: 1.5928 - 6884.1 sample/sec
#####
Test ACC Rate --> 75.14999999999999 %
main
```



model 코드에서 마지막 layer 의 filter 개수를 늘리고, 직전 layer kernel size 를 낮춘 뒤 epoch 를 20 으로 하여 simulation 해보았더니 76%의 정확도를 확인할 수 있었다.

```
Global step: 3875 - [=====>-----] 77% - acc: 0.9219 - loss: 1.5409 - 65
Global step: 3885 - [=====>-----] 82% - acc: 0.9375 - loss: 1.5261 - 641
Global step: 3895 - [=====>-----] 87% - acc: 0.8867 - loss: 1.5750 - 633
Global step: 3905 - [=====>-----] 92% - acc: 0.9141 - loss: 1.5479 - 642
Global step: 3915 - [=====>-----] 97% - acc: 0.9414 - loss: 1.5211 - 637
#####
Test ACC Rate --> 76.87 %
main
```

위의 조건에서 직전 layer filter 개수를 올렸더니 오히려 정확도가 줄어들었다.

```
Global step: 3865 - [=====>-----] 71% - acc: 0.8984 - loss: 1.5643 - 4822.3 sample/sec
Global step: 3875 - [=====>-----] 77% - acc: 0.9336 - loss: 1.5343 - 5128.0 sample/sec
Global step: 3885 - [=====>-----] 82% - acc: 0.9219 - loss: 1.5410 - 5019.8 sample/sec
Global step: 3895 - [=====>-----] 87% - acc: 0.9297 - loss: 1.5326 - 5062.4 sample/sec
Global step: 3905 - [=====>-----] 92% - acc: 0.9102 - loss: 1.5503 - 4992.7 sample/sec
Global step: 3915 - [=====>-----] 97% - acc: 0.9375 - loss: 1.5240 - 4923.5 sample/sec
#####
Test ACC Rate --> 76.53 %
main
```

여러 simulation 을 통해 적절한 개수의 convolution layer 와 pooling layer, kernel 크기와 filter 개수를 추정해보았다. 그 결과, 4 개의 convolution layer, 2 x 2 의 kernel size, 2 배씩 증가하며 최대 128 개의 개수를 갖는 filter 로 최종 model3 를 만들었다. kernel size 가 매우 작고, filter 의 개수는 많은 편이라 오랜 시뮬레이션 시간이 걸렸다. epoch 를 20 으로 했을 경우 아래와 같이 76.24%의 정확도를 얻을 수 있었다.

```
Global step: 3795 - [=====>-----] 36% - acc: 0.9062 - loss: 1.5525 - 5247.6 sample/sec
Global step: 3805 - [=====>-----] 41% - acc: 0.8945 - loss: 1.5700 - 5223.2 sample/sec
Global step: 3815 - [=====>-----] 46% - acc: 0.8945 - loss: 1.5710 - 5223.4 sample/sec
Global step: 3825 - [=====>-----] 51% - acc: 0.9023 - loss: 1.5597 - 5047.9 sample/sec
Global step: 3835 - [=====>-----] 56% - acc: 0.9219 - loss: 1.5413 - 5205.1 sample/sec
Global step: 3845 - [=====>-----] 61% - acc: 0.8750 - loss: 1.5884 - 5197.6 sample/sec
Global step: 3855 - [=====>-----] 66% - acc: 0.8789 - loss: 1.5868 - 5338.7 sample/sec
Global step: 3865 - [=====>-----] 71% - acc: 0.9219 - loss: 1.5387 - 4993.8 sample/sec
Global step: 3875 - [=====>-----] 77% - acc: 0.9375 - loss: 1.5282 - 5195.9 sample/sec
Global step: 3885 - [=====>-----] 82% - acc: 0.9297 - loss: 1.5378 - 5114.5 sample/sec
Global step: 3895 - [=====>-----] 87% - acc: 0.9141 - loss: 1.5484 - 5247.2 sample/sec
Global step: 3905 - [=====>-----] 92% - acc: 0.8672 - loss: 1.5918 - 5227.7 sample/sec
Global step: 3915 - [=====>-----] 97% - acc: 0.9102 - loss: 1.5521 - 5324.2 sample/sec
#####
Test ACC Rate --> 76.24 %
main
```

epoch 값을 50 으로 한 뒤 simulation 해보았더니, 약 77 프로의 정확도를 얻을 수 있었다.

The screenshot shows a Jupyter Notebook titled 'test.ipynb' with a code cell containing training progress logs. The logs show global steps from 9655 to 9795, with accuracy increasing from 26% to 97% and loss decreasing from 1.4957 to 1.5230. The test accuracy rate is 77.3%.

Below the logs, a confusion matrix is displayed for the following classes: airplane, automobile, cat, deer, dog, frog, horse, ship, truck. The matrix shows the relationship between predicted and actual classes, with a scale of 0 to 25 for each cell.

simulation number	epoch	batch size	accuracy
#1	20	256	56.45%
#2	50	256	10.00%
#3	20	256	70.08%
#4	20	256	72.31%
#5	20	512	73.12%
#6	20	512	76.42%
#7	50	512	79.49%
#8	20	256	75.14%
#9	20	256	76.87%
#10	20	256	76.53%
#11	20	256	76.24%
#12	50	256	77.03%

## Result &amp; Analysis

## A. model 2 와 3 의 비교 결과표

- model2

epoch = 20, batch size = 256

```
Global step: 3665 - [=====>-----] 71% - acc: 0.7812 - loss: 1.6741 - 8861.3 sample/sec
Global step: 3675 - [=====>-----] 77% - acc: 0.8281 - loss: 1.6276 - 8021.3 sample/sec
Global step: 3685 - [=====>-----] 82% - acc: 0.7969 - loss: 1.6611 - 8850.8 sample/sec
Global step: 3695 - [=====>-----] 87% - acc: 0.7773 - loss: 1.6796 - 8986.2 sample/sec
Global step: 3905 - [=====>-----] 92% - acc: 0.8008 - loss: 1.6627 - 8290.5 sample/sec
Global step: 3915 - [=====>-----] 97% - acc: 0.8320 - loss: 1.6326 - 9145.3 sample/sec
#####
Test ACC Rate --> 66.89 %
main
```

epoch = 20, batch size = 512

```
Global step: 1913 - [=====>-----] 51% - acc: 0.8262 - loss: 1.6385 - 11158.8 sample/sec
Global step: 1923 - [=====>-----] 61% - acc: 0.7832 - loss: 1.6773 - 11329.0 sample/sec
Global step: 1933 - [=====>-----] 71% - acc: 0.8184 - loss: 1.6435 - 11316.1 sample/sec
Global step: 1943 - [=====>-----] 82% - acc: 0.8145 - loss: 1.6542 - 11313.1 sample/sec
Global step: 1953 - [=====>-----] 92% - acc: 0.7852 - loss: 1.6744 - 11121.2 sample/sec
#####
Test ACC Rate --> 69.13 %
main
```

epoch = 50, batch size = 256

```
Global step: 9745 - [=====>-----] 71% - acc: 0.8594 - loss: 1.6007 - 8880.2 sample/sec
Global step: 9755 - [=====>-----] 77% - acc: 0.8867 - loss: 1.5736 - 8898.1 sample/sec
Global step: 9765 - [=====>-----] 82% - acc: 0.8594 - loss: 1.6011 - 9024.2 sample/sec
Global step: 9775 - [=====>-----] 87% - acc: 0.8867 - loss: 1.5734 - 9060.1 sample/sec
Global step: 9785 - [=====>-----] 92% - acc: 0.8672 - loss: 1.5934 - 8944.2 sample/sec
Global step: 9795 - [=====>-----] 97% - acc: 0.8906 - loss: 1.5713 - 8915.2 sample/sec
#####
Test ACC Rate --> 68.73 %
main
```

➔ model2 의 경우, 최대 69 의 정확도를 갖는 것을 확인하였다.

- model3

epoch = 20, batch size = 256

```
Global step: 3665 - [=====>-----] 71% - acc: 0.9213 - loss: 1.3307 - 4335.6 sample/sec
Global step: 3675 - [=====>-----] 77% - acc: 0.9375 - loss: 1.5282 - 5195.9 sample/sec
Global step: 3685 - [=====>-----] 82% - acc: 0.9297 - loss: 1.5378 - 5114.5 sample/sec
Global step: 3695 - [=====>-----] 87% - acc: 0.9141 - loss: 1.5484 - 5247.2 sample/sec
Global step: 3905 - [=====>-----] 92% - acc: 0.8672 - loss: 1.5918 - 5227.7 sample/sec
Global step: 3915 - [=====>-----] 97% - acc: 0.9102 - loss: 1.5521 - 5324.2 sample/sec
#####
Test ACC Rate --> 76.24 %
main
```

epoch = 50, batch size = 256

```
Global step: 9755 - [=====>-----] 77% - acc: 0.9531 - loss: 1.5078 - 2033.2 sample/sec
Global step: 9765 - [=====>-----] 82% - acc: 0.9492 - loss: 1.5107 - 2051.1 sample/sec
Global step: 9775 - [=====>-----] 87% - acc: 0.9648 - loss: 1.4958 - 2025.7 sample/sec
Global step: 9785 - [=====>-----] 92% - acc: 0.9219 - loss: 1.5386 - 2073.9 sample/sec
Global step: 9795 - [=====>-----] 97% - acc: 0.9375 - loss: 1.5230 - 2037.6 sample/sec
#####
Test ACC Rate --> 77.3 %
main
```

➔ model3 의 경우, 최대 77 의 정확도를 갖는 것을 확인하였다



- 성능 비교 결과 표

	epoch	batch size	accuracy
model2	20	256	66.89 %
model2	50	256	68.73 %

→ model2 의 경우, epoch 가 20 일 때 66.89%의 정확도를 갖다가, epoch 를 50 으로 올려주면 약 2%의 정확도가 향상되는 것을 확인할 수 있었다.

	epoch	batch size	accuracy
model3	20	256	76.24 %
model3	50	256	77.03 %

→ model3 의 경우, epoch 가 20 일 때 76.24%의 정확도를 갖다가, epoch 를 50 으로 올려주면 약 1%의 정확도가 올랐다. epoch 를 50 으로 올리면서 훨씬 많은 시간이 시뮬레이션에 소요됐음에도, 1%남짓 한 만큼의 성능만이 향상되었다.

### Consideration

- 머신러닝 강의에서 들은 내용들과, 구글링으로 검색하여 얻은 모델 성능 올리는 방법을 하나씩 적용하면서 시뮬레이션을 해보았는데, 이 내용들을 모두 적용시킨다고 좋은 성능을 얻지는 못한다는 것을 알게되었다.

convolution layer 를 추가하고 code 를 많이 수정할수록 오히려 더 낮아지는 성능을 보며, 생각대로 성능이 올라주지 않아 해결법을 찾아 헤메었다. 그런데 기존의 base code 에서 kernel 크기나 feature 개수들 중 한 가지만 수정하였는데 갑자기 큰 폭으로 성능이 올라 당황스러웠다. 검색해보니 이와 같은 경우가 꽤 빈번히 있는 듯 해서 이유가 궁금했다.

기회가 된다면 다음번에는 딥러닝 프로젝트를 keras 와 연동이 되는 tensorflow v2 를 사용해보고싶다는 생각을 했다. 이번 과제에서는 simulation 에도 많은 시간과 생각이 필요했지만, 그보다도 tensorflow v1 를 사용하기 위한 컴퓨터 내 환경 문제가 많이 생겨 시작에 어려움을 겪었다. 이를 구글 Colab 사용으로 해결하였지만, 다음 프로젝트를 위해 환경 설정에 관한 공부 역시 해봐야겠다고 생각했다.