**과목명: 시스템프로그래밍**

**CSE4110-02**

**<<Project #4>>**

**서강대학교 [공학부 컴퓨터공학과]**

**[20171662]**

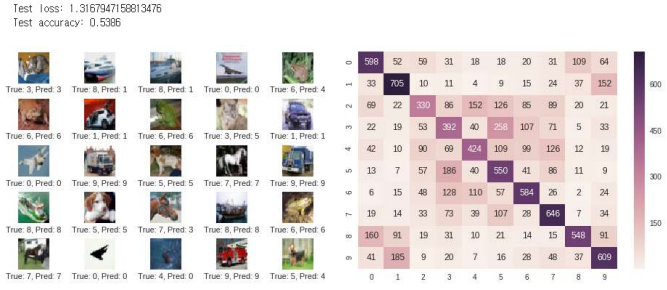
**[이나연]**

목 차

1. **프로그램 개요**
2. **프로그램 실행 방법**
3. **모델의 구성**
4. **Confusion Matrix**
5. **Loss Function**
6. **Data Reshaping과정에서 rank를 4로 맞추는 이유**

# 프로그램 개요

CIFAR-10이라는 data set을 사용하여 이미지 분류를 하는 머신 러닝 모델을 개발하는 프로그램이다. CIFAR-10 데이터 셋엔 총 60,000개의 32\*32픽셀의 컬러 이미지로 총 10개의 클래스, 클래스 당 6,000개의 이미지가 존재한다. 이 때 하나의 이미지에 대하여 두개의 클래스가 동시에 포함하고 있는 경우는 없다. 하나의 이미지는 하나의 클래스로만 mapping된다. 해당 프로그램의 기본 BASE가 되는 모델에 대한 코드는 Google Colab에서 제공하는 코드를 사용하였다. 그러나 해당 코드는 MNIST data set에 최적화되어 있어 정확도가 50프로 정도로 되어 있었다. 따라서 본 프로젝트를 통해 직접 이미지 분류 성능을 측정하고 모델의 파라미터들을 바꾸는 등 주어진 모델을 수정하여 정확도가 80% 이상이 나오게끔 수정하였다.



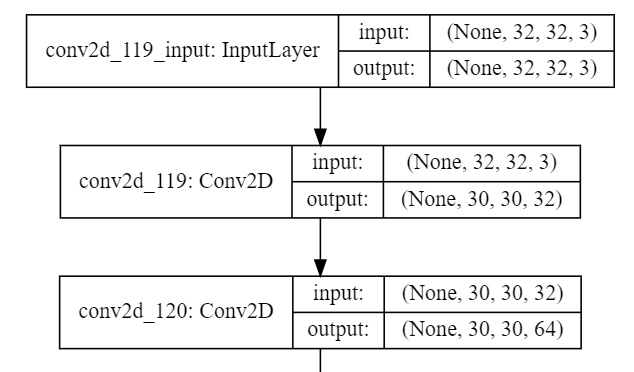
<기존의 코드를 사용하여 머신 러닝을 진행한 결과>

# 프로그램 실행 방법

해당 프로그램은 Google에서 제공하고 있는 Colaboratory기능을 사용하여 구현하였다. 따로 IDE가 필요 없기 때문에 실행 방법으로는 해당 .ipynb파일을 Colab에서 열어 실행하면 된다. 셀의 개수가 많으므로, 런타임>모두 실행 또는 Ctrl+F9를 사용하여 실행한다. Colab의 또 다른 장점으로는 GPU를 지원하고 있고, GPU를 사용하지 않을 때와 사용할 때의 속도 차이가 많이 나므로 수정>노트 설정에서 반드시 GPU로 설정되어 있는지를 확인한다.

# 모델의 구성

해당 모델은 layer들을 위에서 아래로 추가한 순서대로 똑같이 나타나고 있다.



CIFAR-10 데이터셋은 60000개의 32\*32 픽셀의 컬러 이미지로 클래스 당 6000개의 이미지로 총 10개의 클래스가 있다. input channel은 3개로 R, G, B로 나뉘어 input이 들어오게 된다. 따라서 Input Layer의 shape이 (32,32,3)이 된다.

첫번째로

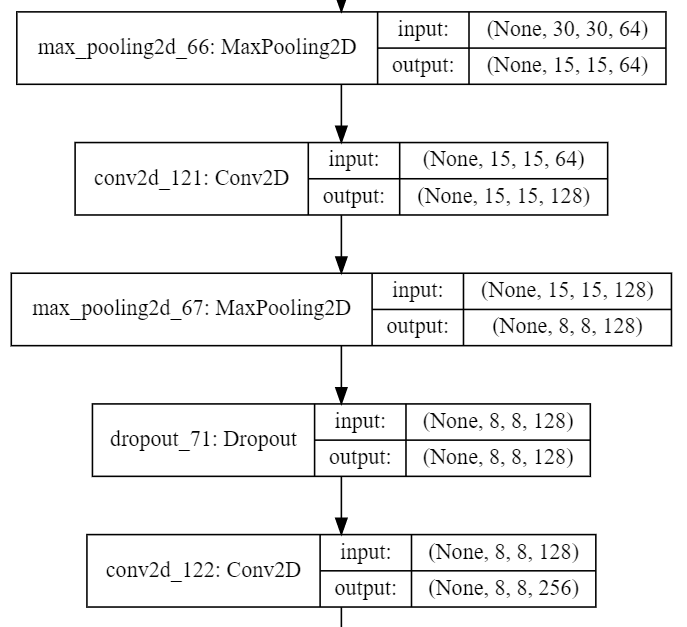
**model.add(Conv2D(32, kernel\_size=(3, 3), activation='relu', input\_shape=x\_train.shape[1:]))**

32개의 3\*3 kernel size의 convolution filters가 있는 layer를 추가하였다. activation function으로 ‘ReLU’를 사용하였다. 3\*3 kernel로 각 layer에 대하여 convolution을 실행하게 되면, 기존의 행과 열의 수에서 2가 감소한 30\*30이 된다. 한 쌍의 r,g,b는 하나의 layer로 묶이게 되고, 이러한 변형된 output layer의 channel의 수가 총 32개가 되는 것이다. 이러한 방법으로 1차적으로 feature를 추출할 수 있게끔 하였다. padding=”same”으로 주지 않은 이유는 처음부터 가장자리의 값까지 다 가지고 오면 data가 너무 많아져 핵심 feature 찾기에 오히려 방해가 된다 판단하여 넣지 않았다. 그러나 이 이후 시행되는 모든 convolution과 max pooling에는 padding=”same”을 넣어 주었다. activation function으로는 ReLU 함수를 사용하여 output channel들에는 음수였던 data들이 0으로 표기될 수 있도록 하였다.

같은 방법으로

**model.add(Conv2D(64, (3, 3), activation='relu',padding='same'))**

로 64개의 3\*3 kernel size의 convolution filter가 있는 layer를 추가하였다. 이를 통해 조금 더 자세하게 이미지상에서의 feature를 추출해낼 수 있게끔 하였다. 해당 줄에는 padding=’same’이라는 옵션을 주어 convolution filter가 씌워질 때 가장자리의 값들은 없어지지 않게끔 zero padding을 사용하게 하였다. padding을 사용하지 않을 경우, data의 spatial 크기는 convolution layer들을 지나며 점점 작아지게 되므로 가장자리의 정보들이 소실되는 문제점이 생길 수 있기 때문이다. 따라서 해당 layer 이전 layer의 dimension에서의 30\*30 값은 그대로 고정되고, channel수는 64로 늘어나 dimension이 (30,30,64)로 변하게 된다.



그런 다음 max pooling 기법을 사용한다.

**model.add(MaxPooling2D(pool\_size=(2, 2),padding='same'))**

이를 통해 data의 크기를 줄일 수 있어 전체 parameter의 개수 역시 줄일 수 있다. 이는 나중에 있을 fully connected layer를 구현할 때에 도움이 된다. 또한 pooling 자체를 시행하면 찾고자 하는 feature가 map상에서 어느 곳에 위치하여도 크게 영향을 받지 않게끔 해준다. 현재 layer에선 2\*2의 max pooling을 적용하였다. 각 channel마다 똑같이 적용하기 때문에 channel의 수는 똑같이 64이고, pooling의 결과로 30\*30의 크기를 가지던 data가 각 변의 1/2, 전체적으로 1/4 줄은 15\*15의 크기를 가지는 data로 변하였다. 또한 padding=’same’옵션까지 주어 가장자리의 정보들이 손실되지 않도록 하여 정확도를 높이게끔 하였다.

그런 다음 다시 convolution을 시행하였다.

**model.add(Conv2D(128, (3, 3), activation='relu',padding='same'))**

위에서 설명한대로 똑같이 3\*3 kernel size를 가지는 convolution filter를 128개 가지고 있는 layer를 추가하였다. 이전 layer에서 (15,15,64)의 dimension이 넘어 왔는데, padding=’same’ 옵션이 주어졌으므로 output의 dimension의 각 channel의 크기 역시 변함이 없다. 따라서 output dimension은 (15,15,128)이 된다.

또 한 번 max pooling을 해준다.

**model.add(MaxPooling2D(pool\_size=(2, 2),padding='same'))**

위와 같은 방법으로 2\*2의 크기를 가지는 max pooling을 진행하게 되면 original channel의 높이와 너비가 각각 1/2로 줄어들게 된다. 따라서 dimension은 (8,8,128)이 된다. 15는 2로 나누어 떨어지지 않지만 ceil이 적용되어 8이 되는 것이다. 이렇게 함으로써 또 한번 data의 size를 감소시켜주었다. 그와 동시에 padding=’same’ 옵션을 주어 가장자리의 data가 소실되지 않게끔 하였다.

그런 다음 Dropout layer를 추가해주었다.

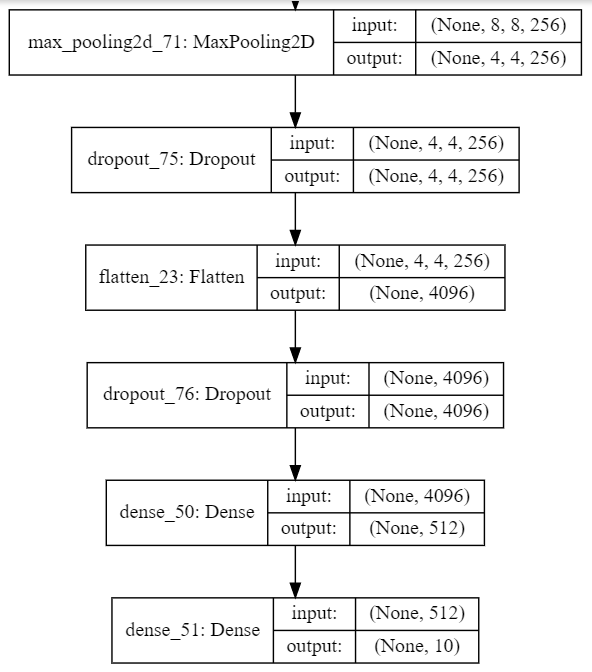
**model.add(Dropout(0.125))**

Dropout이란 네트워크가 전체 뉴런에 걸쳐 학습된 표현을 배포하도록 강제한다. layer안의 임의의 뉴런을 선택하여 값을 0으로 설정하여 ‘꺼버리는’ 방식이다. 전체 비율의 12.5%를 선택하여 없애 주었다. 이러한 방식은 overfitting을 방지해주는 역할을 한다. 너무 비율이 크면 정확도가 떨어질 수 있기 때문에 적당한 값을 집어넣어주었다. 이러한 과정을 거쳐도 dimension엔 아무런 변화가 없으므로 output dimension은 여전히 (8,8,128)이 된다.

다시 convolution layer를 추가해준다.

**model.add(Conv2D(256, (3, 3), activation='relu',padding='same'))**

256개의 3\*3 kernel size의 convolution filter가 있는 layer를 추가하였다. 이를 통해 조금 더 자세하게 이미지상에서의 feature를 추출해낼 수 있게끔 하였다. 해당 줄에는 padding=’same’이라는 옵션을 주어 convolution filter가 씌워질 때 가장자리의 값들은 없어지지 않게끔 zero padding을 사용하게 하였다. 따라서 channel 자체의 크기는 input과 동일하게 8\*8이 되고 channel의 수는 256개가 된다. 따라서 output dimension은 (8,8,256)이 된다.



다시 max pooling을 해준다.

**model.add(MaxPooling2D(pool\_size=(2, 2),padding='same'))**

방식 역시 위와 동일하다. 각 channel의 높이와 너비가 1/2배 되므로 크기는 4\*4로 줄어든다. 그러나 channel의 수는 이전과 동일하게 256개가 된다. 따라서 output dimension은 (4,4,256)이 된다.

그런 다음 Dropout layer를 추가해주었다.

**model.add(Dropout(0.25))**

전체 비율의 25%를 선택하여 없애 주어 또 다시 overfitting을 방지해주는 역할을 한다. 너무 비율이 크면 정확도가 떨어질 수 있기 때문에 적당한 값을 집어넣어주었다. 이러한 과정을 거쳐도 dimension엔 아무런 변화가 없으므로 output dimension은 여전히 (4,4,256)이 된다.

이러한 과정을 거친 후 Flatten layer를 추가해주었다.

**model.add(Flatten())**

Flatten 과정은 2차원의 feature map을 fully connected layer으로 전달하기 위한 중간 과정으로 2차원을 1차우너 형식으로 바꿔준다. 따라서 input dimension이 (4,4,256) 이었으므로 총 4\*4\*256개의 data가 있는 것인데, 이를 1차원으로 바꾸어 주는 것이다. 즉 4\*4\*256 = 2^12 = 4096이 된다. 그 결과 output dimension은 (4096)이 된다.

그런 다음 다시 Dropout layer를 추가해주었다.

**model.add(Dropout(0.25))**

전체 비율의 25%를 선택하여 없애 주어 또 다시 overfitting을 방지해주는 역할을 한다. 너무 비율이 크면 정확도가 떨어질 수 있기 때문에 적당한 값을 집어넣어주었다. 이러한 과정을 거쳐도 dimension엔 아무런 변화가 없으므로 output dimension은 여전히 (4096)이 된다.

그런 다음 Dense layer를 추가해주었다.

**model.add(Dense(512, activation='relu'))**

Dense()를 사용함으로써 fully-connected layer를 추가하였다. 첫번째 인자는 출력 뉴런의 수로, 512의 값을 주었기 때문에 총 512개의 출력 뉴런을 생성한다. activation은 ‘relu’의 옵션을 주어 활성화 함수를 ReLU로 설정해주었다. 이로써 음수값들은 모두 0으로 전환하였다. 그러나 그 다음 layer가 존재하기 때문에 사실상 output layer가 아닌 hidden layer에 512개의 뉴런이 생긴 것이다.

**model.add(Dense(num\_classes, activation='softmax'))**

이 layer는 마지막 layer로 class의 개수가 첫번째 인자로 들어가게 된다. 총 10개의 뉴런이 output layer에 보이게 되고, 이전 layer에서 또 fully-connected하게 생긴다. 또한 여기서는 활성화 함수를 ‘softmax’로 주었다. 해당 활성화 함수는 셋 이상을 분류하는 다중 클래스 분류 문제에서 출력층에 주로 사용되는 활성화 함수이다.

**optimizer = keras.optimizers.SGD(lr=learning\_rate,momentum=0.9,decay=learning\_rate/epochs)**

layer의 추가 외에도 optimizer에 대한 수정도 추가로 진행하였다. 우선 momentum=0.9로 설정해주었다. momentum은 0보다 크거나 같은 float값을 가지는데, SGD를 적절한 방향으로 가속화하여 진동을 줄여주는 매개변수이다. 이를 0.9로 설정하여 흔들림을 조금이나마 방지해주었다. decay 역시 0보다 크거나 같은 float 값을 가지는데, 업데이트마다 적용되는 학습률의 감소율을 설정해준다. 이를 learning\_rate/epochs로 하여 epochs가 늘어날수록 lr의 값이 점점 작아지게끔 설정해주었다.

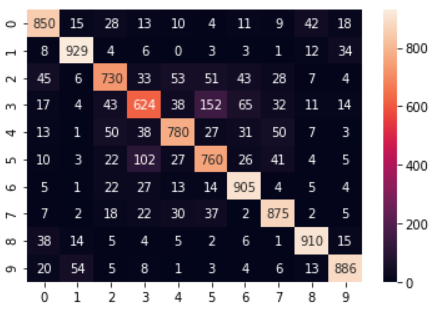
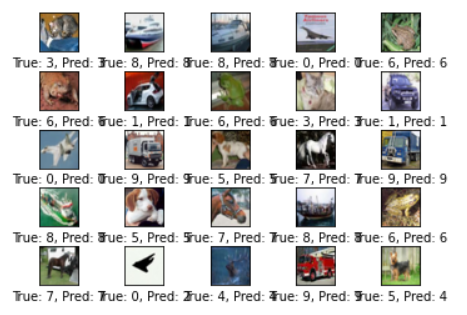
이렇게 convolution과 max pooling, 그리고 dense까지 여러 기능을 써보며 training을 한 결과,

Test loss: 0.6832042122364044

Test accuracy: 0.8248999714851379

의 결과가 도출되었다. convolution layer의 추가 횟수, channel 수, max pooling layer의 추가 횟수 등을 여러 번 조절해가며 얻은 결과이다.

# Confusion matrix

# Loss Function

머신러닝의 기본은 모델의 학습을 진행하며 최대한 오류가 적게 나게끔 정확하게 학습을 진행하는 것이 목표이다. 따라서 학습의 과정을 거치며 오류가 제일 작게 나타나는 부분을 찾아내며 optimization을 진행하는 것이 필요하고, 이러한 과정을 generalization이라 하는 것이다. 이렇게 오류가 제일 작게 나타나는 것을 찾아내는 것을 loss function을 통해서 한다. 쉽게 말해 학습시키려는 데이터가 linear function이라면 그 직선에서부터 현재의 데이터 학습 결과가 얼마나 떨어져 있는지를 계산하는 함수이다. 통계 개념에서의 표준편차의 개념을 생각하면 쉽다. data를 가지고 parameter를 설정한 후, 예상된 값이 얼마나 떨어져 있는지를 함수식으로 표현한다. 함수의 결과값이 크면 클수록 많이 틀렸다는 것이고, 0에 가까울수록 완벽하게 추정하였다는 의미를 가진다. 해당 프로그램에서도 CIFAR-10의 data set을 가지고도 학습을 진행하여 이미지가 클래스에 맞게 맞출 수 있도록 loss function이 필요하다. 아래는 본 프로그램에서 사용한 loss function을 설정하는 부분이다.

model.compile(loss=keras.losses.categorical\_crossentropy,

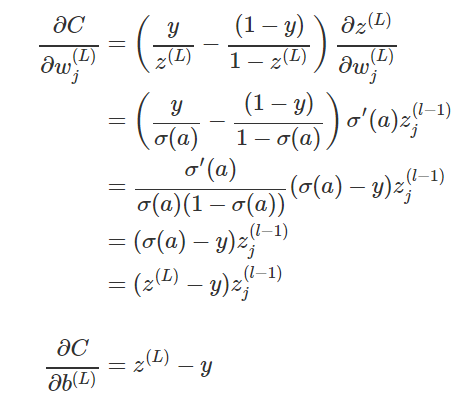
              optimizer=optimizer,

              metrics=['accuracy'])

이렇게 모델을 컴파일하는 과정에서 loss function을 cross entropy로 설정해주었다. cross entropy란 실제 분포q에 대해 알지 못하는 상태에서 모델링을 통하여 구한 분포 p를 통하여 q를 예측하는 것이다. 두개의 인자 p와 q가 식에 쓰이기 때문에 ‘cross’라는 워딩을 사용한다. 이는 실제값과 예측값이 비슷할수록 0에 가까워지고, 반대로 다를수록 값이 증가하기 때문에 그 둘의 차이를 줄이기 위해 쓰이는 entropy이다. cross entropy의 형태를 가지는 loss function을 사용하게 되면 output layer에서 activation function의 도함수에 의한 영향을 제거해줄 수 있다. 아래의 식을 보면 알 수 있다.



이러한 식이 성립할 때에, 다음의 식 역시 성립한다.



도함수의 제거가 필요한 이유는 학습 속도 저하 현상의 원인 중 하나가 activation function의 도함수이기 때문이다. 만약 도함수의 식이 제거된다면 원래 기대했던 것처럼 학습 속도는 오차의 크기에 비례하게 되기 때문이다. 이러한 cross entropy에선 낮은 확률로 예측해서 맞추거나, 높은 확률로 예측하여 틀리는 경우 loss가 더 크다. 이러한 entropy에서 categorical\_crossentropy는 카테고리가 3개 이상일 때에 사용된다. 따라서 현재 training의 목표는 총 10개의 class에 대하여 설정해야 하므로 categorical\_crossentropy를 사용해주었다. 여기서 실제값과 예측값사이의 차이를 각 클래스 별로 계산하여 그 오차값을 모두 더하는 기능을 제공한다. 또한 이러한 loss function을 쓰기 전에 다중 클래스 분류에서 많이 쓰이는 activation function인 softmax함수를 통하여 예측값을 계산한 후 해당 loss function에 대입하게 된다. 실제로 model 구성의 제일 마지막 layer에서 softmax함수를 activation function으로 사용하여 Dense()로 output layer의 노드를 10개로 줄였다.

# Data Reshaping과정에서 rank를 4로 맞추는 이유

어떠한 data set에 대하여 훈련을 시키고 새로운 data에 대한 예측을 수행하려면 해당 모듈이 지원하는 data의 입력 크기/dimension을 맞춰 주어야 한다. 본 프로그램에서 사용하는 keras 모듈에 대하여서는 원하는 data set에 대한 훈련, 예측 및 분류를 하기 위해서는 총 4개의 dimension으로 reshaping을 한 후 훈련을 진행하여야 한다. 아래는 본 프로그램에서 image file을 reshaping하는 과정이다.

if len(x\_train.shape) < 4:

  x\_train = x\_train.reshape(x\_train.shape[0], x\_train.shape[1], x\_train.shape[2], 3)

x\_test = x\_test.reshape(x\_test.shape[0], x\_train.shape[1], x\_train.shape[2], 3)

만약 data file의 rank가 4보다 작게 되면 4를 가지도록 바꿔주는 코드이다.

위의 과정을 거치며 이미지 데이터를 담은 set의 rank를 4로 맞춘다. 이러한 이유는 4개의 차원이 필요하기 때문이다. 우선 첫번째 차원은 image file들의 개수를 위함이고 (CIFAR-10의 경우 train용: 50000개, test용: 10000개), 두번째와 세번째 치원은 image file의 높이와 너비를 위함이고, 마지막 차원은 channel, 즉 각 색깔 RGB를 위함이다. 만약 주어지는 입력이 흑백이라면 오직 하나의 channel만 쓰이겠지만, 색이 있는 image file들을 대상으로 machine learning을 진행하여야 한다면 3개의 channel이 쓰이는 것이다. 따라서 제일 마지막 rank가 1이라면 흑백, 3이라면 컬러 사진에 대한 훈련을 진행하겠다는 의미인 것이다. 만약 rank가 이미 4라면 위 과정을 거치지 않는다. CIFAR-10의 image들은 32\*32픽셀로 중간 2개의 rank의 shape이 32,32이다. rank1,2,3의 값들은 사용자가 직접 입력하여도 되지만, 주어진 data set은 shape에 미리 해당 정보가 순서대로 저장되어 있어 따로 숫자를 지정해주는 것은 channel의 수밖에 없었다.