**과목명: 시스템프로그래밍**

**분반: 1반(홀수)**

**<<Project #4>>**

**서강대학교 컴퓨터공학과**

**학번 : 20161641**

**이름 : 정성엽**

**목 차**

1. 프로그램 개요

2. 프로그램 실행 방법

3. 모델 구성 방법

4. Confusion Matrix

5. Loss function

6. 데이터 전처리

1. 프로그램 개요

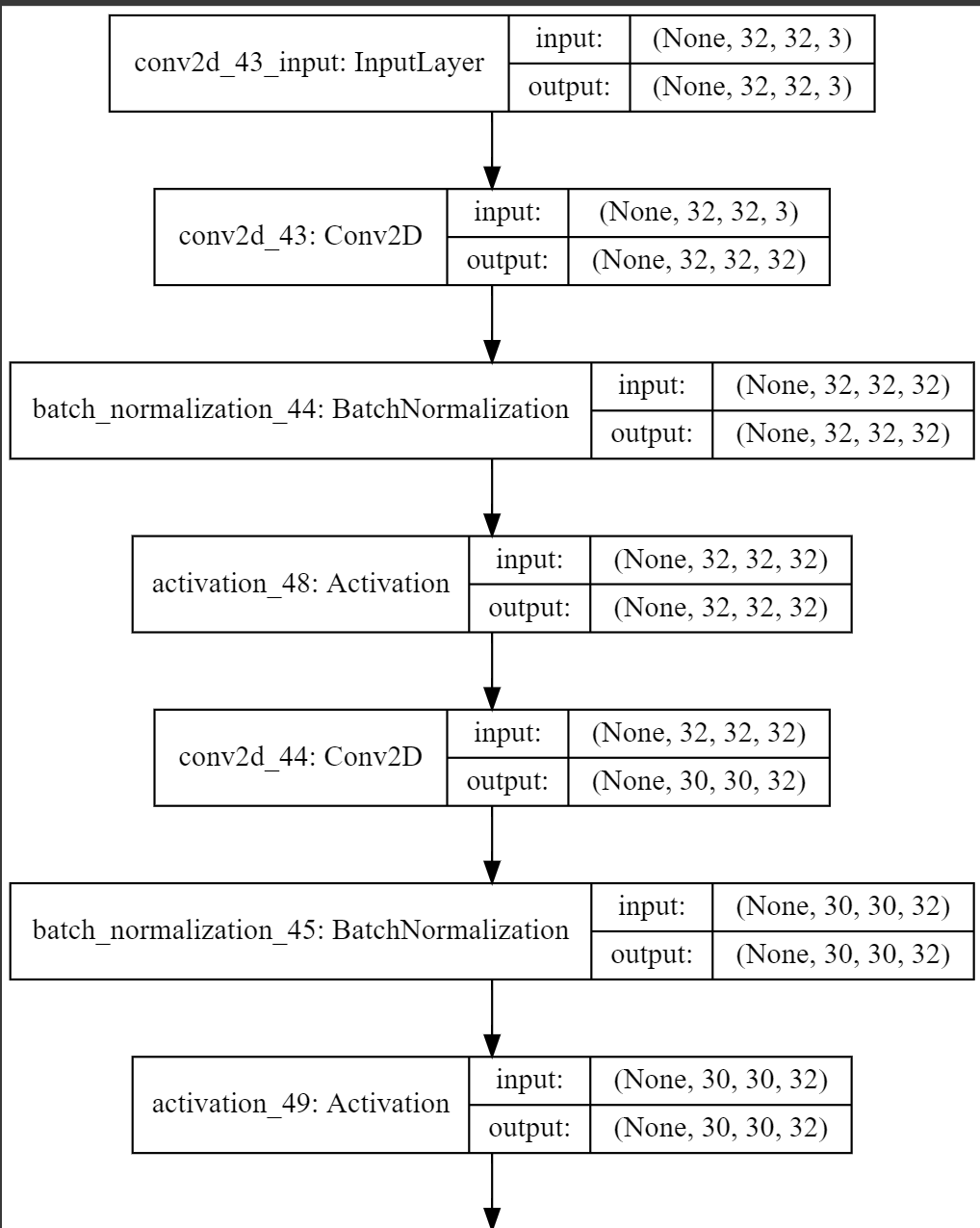
: 이 프로그램은 CIFAR-10 데이터 셋에 대하여 80%이상의 정확도를 얻을 수 있는 모델을 구현한 프로그램이다. 기본적인 Keras 코드에 성능 향상을 위해 여러가지 변화와 추가를 하였다. 이는 모델 구성 방법에서 확인 가능하다.

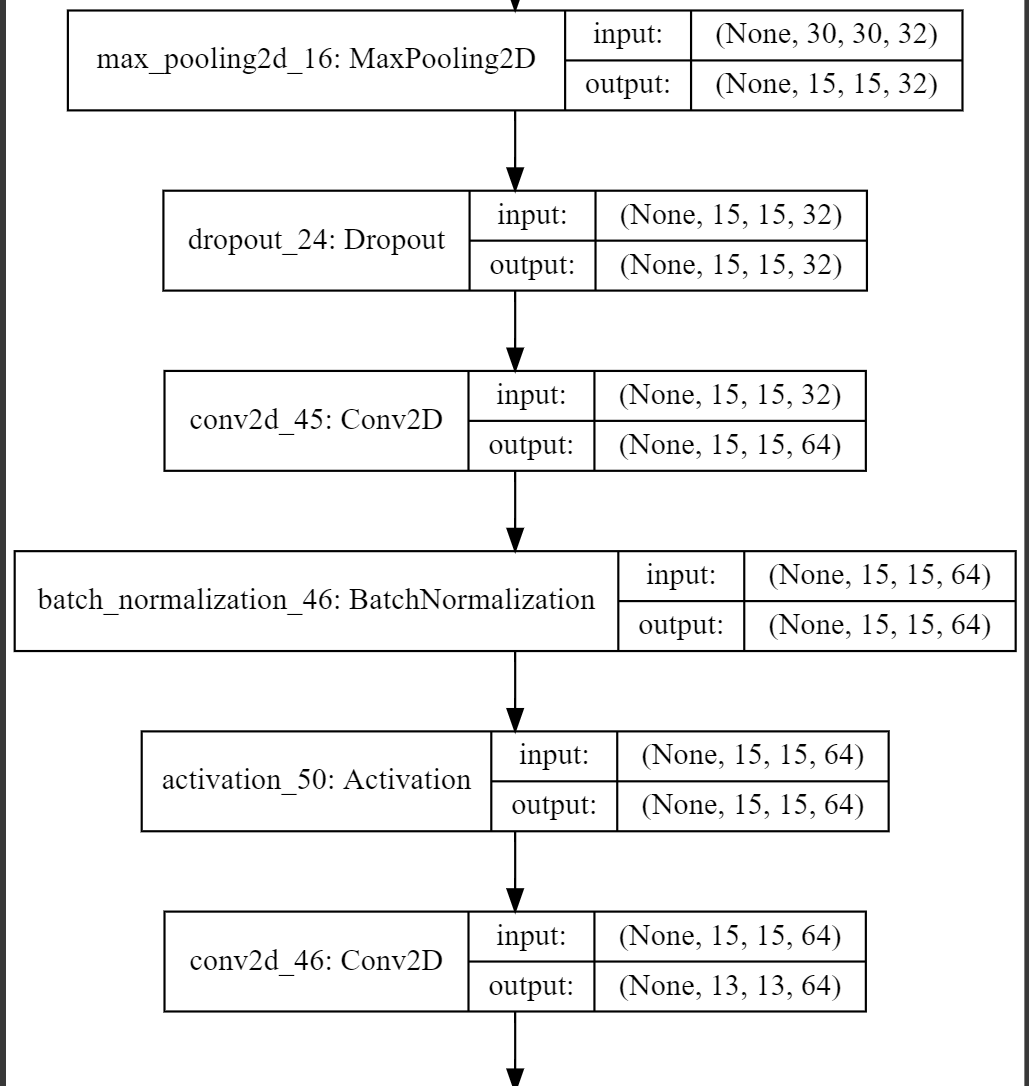
2. 프로그램 실행 방법

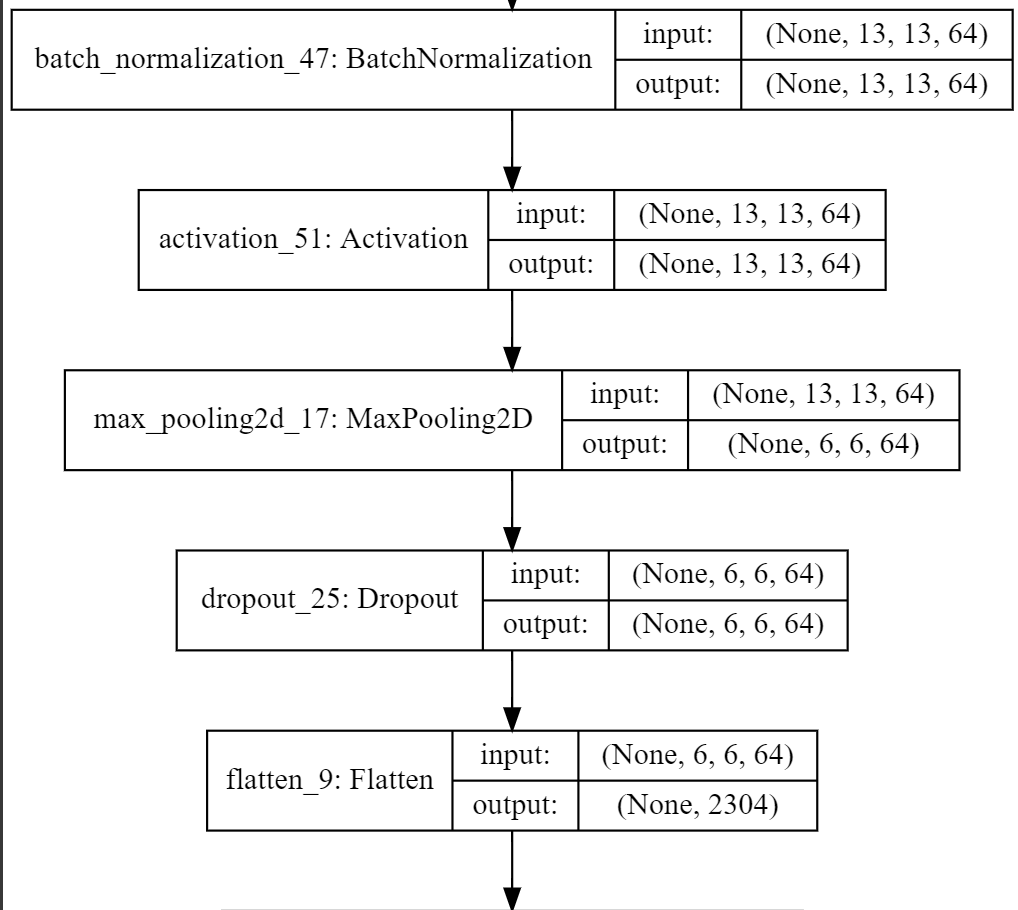
이 프로그램은 Colab환경에서 실행 가능하다. Colab에서 실행하기 위해서는 google drive에 ipynb파일을 옮기고 google colab에서 실행을 눌러 colab환경으로 불러올 수 있다.

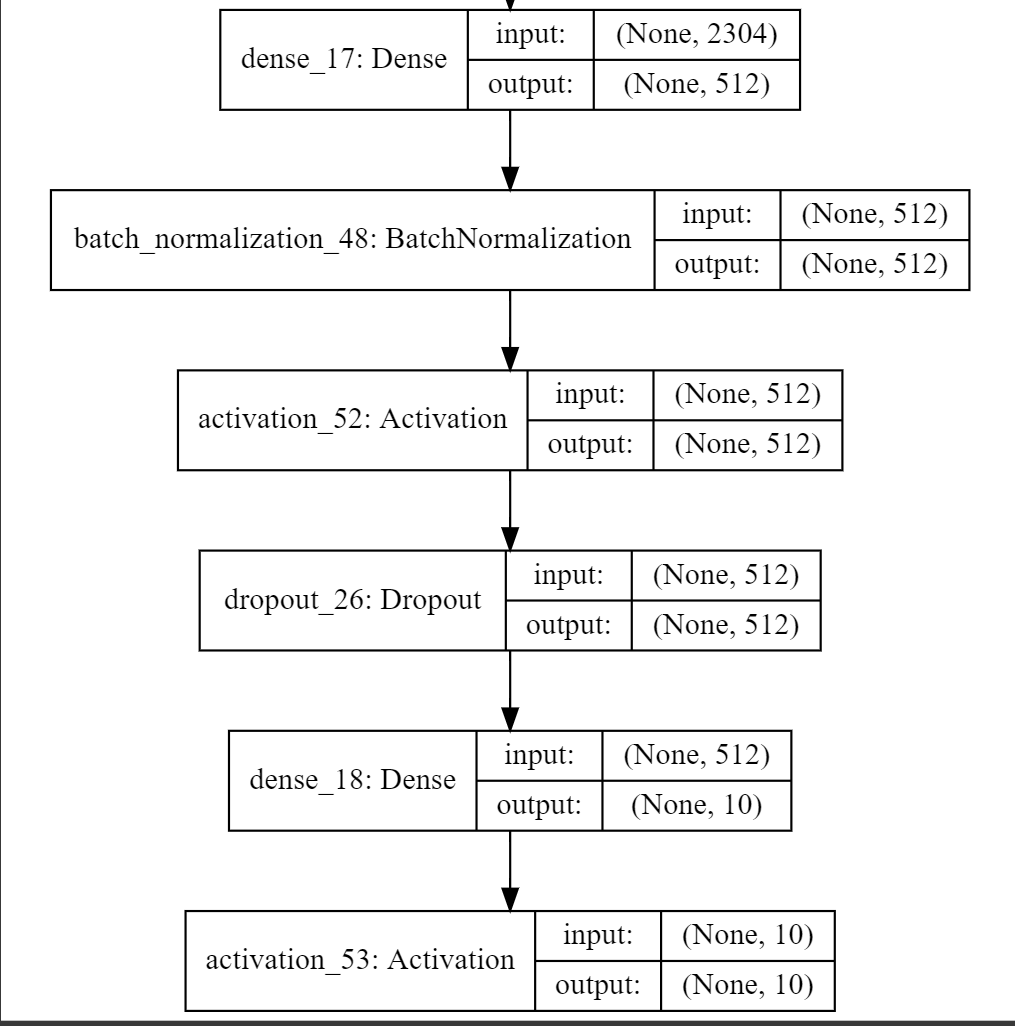
이 프로그램은 epoch를 50으로 설정하여 총 50번의 학습을 실시하는데, 만약 더 많은 학습을 원한다면 epoch를 증가시킨 후 런타임 란의 모두실행을 눌러 모든 코드를 실행하면 된다. (ctrl+F9) 이 외에도 모델을 변경시키는 등의 추가를 하고자 한다면, Adding layers to the model란에 layer추가 코드를 작성하면 된다. Training the model란을 통해 학습의 진행도를 확인할 수 있다. 학습이 완료된 후 모델을 사용하고자 한다면 아래의 코드블록에 사용할 용도에 맞게 코드를 작성하면 된다.

3. 모델 구성 방법









모델에 추가된 명령어는 아래와 같은데 명령어와 주석을 통해 각 코드의 의미와 모델의 구성을 설명하였습니다.

model = Sequential()

# model의 레이어를 선형으로 연결하여 구성

model.add(Conv2D(32, (3, 3), padding='same', input\_shape=x\_train.shape[1:]))

# 이미지를 학습할 것임으로 2D 컨볼루션 레이어를 사용하였습니다.

# Filter size = 32, kernel size = (3,3)으로 구성

# padding을 same으로 하여 실행 후 data의 크기가 줄어들지 않게 하였습니다

# 결과적으로 (32,32,3) 크기의 input은, 32개의 filter로 나뉘어 (32,32,32)크기의 output이 됩니다.

model.add(BatchNormalization())

# data를 평균0, 분산1로 정규화 하여 학습이 잘 일어나도록 돕는다.

# input = (32,32,32), output = (32,32,32)로 동일

model.add(Activation('relu'))

# Activation function은 gradient venishing을 보완하기 위해 ReLu를 사용한다.

# input = (32,32,32), output = (32,32,32)로 동일

model.add(Conv2D(32, (3, 3)))

# 이미지를 학습할 것임으로 2D 컨볼루션 레이어를 사용하였습니다.

# Filter size = 32, kernel size = (3,3)으로 구성

# padding을 default로 설정하여, 실행 후 data의 사이즈가 줄어듭니다.

# 결과적으로 (32,32,32)크기의 데이터는 (30,30,32)크기로 줄어듭니다.

model.add(BatchNormalization())

# data를 평균0, 분산1로 정규화 하여 학습이 잘 일어나도록 돕는다.

# input = (30,30,32), output = (30,30,32)로 동일

model.add(Activation('relu'))

# Activation function은 gradient venishing을 보완하기 위해 ReLu를 사용한다.

# input = (30,30,32), output = (30,30,32)로 동일

model.add(MaxPooling2D(pool\_size=(2, 2)))

# 이미지의 최대값만을 pooling하여 사용합니다.

# pooling size = (2,2)으로 설정

# 2\*2 data를 1\*1 data로 축소가 일어납니다.

# 결과적으로 (30,30,32)크기의 데이터는 (15,15,32)크기로 줄어듭니다.

model.add(Dropout(0.25))

# model의 overfitting을 방지하기 위해 Dropout을 설정하였습니다.

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

# 이미지를 학습할 것임으로 2D 컨볼루션 레이어를 사용하였습니다.

# Filter size = 64, kernel size = (3,3)으로 구성

# padding을 same으로 하여 실행 후 data의 크기가 줄어들지 않게 하였습니다

# 결과적으로 (15,15,32) 크기의 input은, 32개의 filter로 나뉘어 (15,15,64)크기의 output이 됩니다.

model.add(BatchNormalization())

# data를 평균0, 분산1로 정규화 하여 학습이 잘 일어나도록 돕는다.

# input = (15,15,64), output = (15,15,64)로 동일

model.add(Activation('relu'))

# Activation function은 gradient venishing을 보완하기 위해 ReLu를 사용한다.

# input = (15,15,64), output = (15,15,64)로 동일

model.add(Conv2D(64, (3, 3)))

# 이미지를 학습할 것임으로 2D 컨볼루션 레이어를 사용하였습니다.

# Filter size = 64, kernel size = (3,3)으로 구성

# padding을 default로 설정하여, 실행 후 data의 사이즈가 줄어듭니다.

# 결과적으로 (15,15,64)크기의 데이터는 (13,13,64)크기로 줄어듭니다.

model.add(BatchNormalization())

# data를 평균0, 분산1로 정규화 하여 학습이 잘 일어나도록 돕는다.

# input = (13,13,64), output = (13,13,64)로 동일

model.add(Activation('relu'))

# Activation function은 gradient venishing을 보완하기 위해 ReLu를 사용한다.

# input = (13,13,64), output = (13,13,64)로 동일

model.add(MaxPooling2D(pool\_size=(2, 2)))

# 이미지의 최대값만을 pooling하여 사용합니다.

# pooling size = (2,2)으로 설정

# 2\*2 data를 1\*1 data로 축소가 일어납니다.

# 결과적으로 (13,13,64)크기의 데이터는 (6,6,64)크기로 줄어듭니다.

model.add(Dropout(0.25))

# model의 overfitting을 방지하기 위해 Dropout을 설정하였습니다.

model.add(Flatten())

# 다차원의 형태인 자료는 NN에 사용하기 어려움으로 이를 1차원으로 Flatten해줍니다.

# input = (6,6,64), output = (2304)

model.add(Dense(512))

# 노드의 수를 512로 다음 Layer를 추가합니다

# input = (2304), output = (512)

model.add(BatchNormalization())

# data를 평균0, 분산1로 정규화 하여 학습이 잘 일어나도록 돕는다.

# input = (512), output = (512) 로 동일

model.add(Activation('relu'))

# Activation function은 gradient venishing을 보완하기 위해 ReLu를 사용한다.

# input = (512), output = (512) 로 동일

model.add(Dropout(0.5))

# model의 overfitting을 방지하기 위해 Dropout을 설정하였습니다.

model.add(Dense(num\_classes))

# output layer는 Classification 할 model의 개수만큼으로 설정하였습니다.

# input = (512), output = (10)

model.add(Activation('softmax'))

# 마지막으로 SoftMax function을 Activation Function으로 사용하여 Classification을 합니다.

# input = (10), output = (10)

model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy'])

# 선형으로 연결하여 구성된 모델을 학습방식에 맞도록 환경설정함

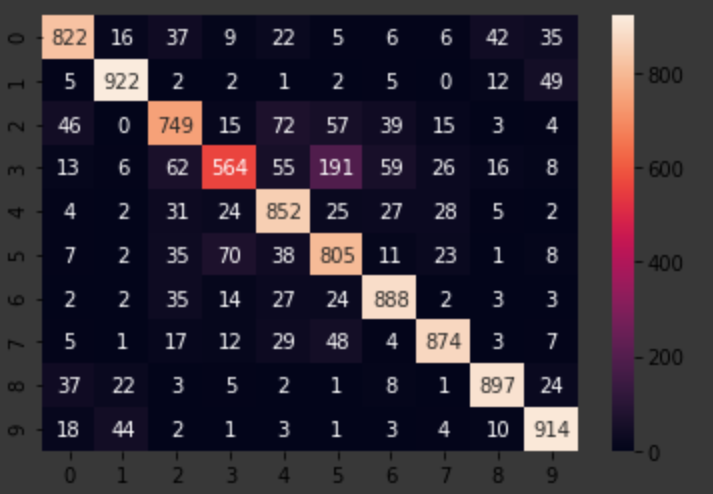
# Loss Function = Categorical Cross Entropy function

# optimizer = Adam

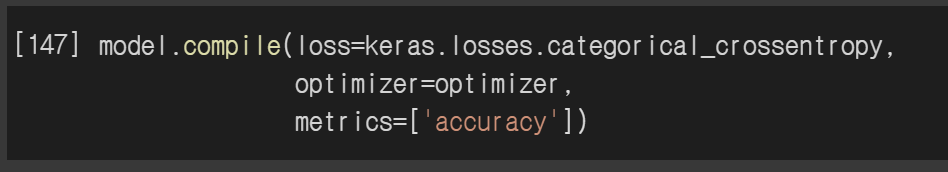
# Metric : 기준 리스트를 설정하는데, 분류를 할 것임으로 [‘accuracy’]로 설정

모델을 구성함에 있어서 가장 중점을 둔 것은 여러 층의 filter를 사용하는 것 입니다. CNN의 특성상 여러 특징들을 뽑아낼 수 있어야 좋은 성능을 보입니다. 때문에 여러 층의 Convolution layer를 구성하였습니다. 그리고 다음으로 신경쓴 것은 정규화 입니다. 일반적으로 대부분의 결과에 있어서 정규화를 추가할수록 좋은 성능을 보였고 때문에 정규화를 많이 사용하였습니다. Activation Function의 경우, Sigmoid보다 ReLu가 더 좋은 성능을 보였습니다. 이는 Gradient venishing을 보완할 수 있어서 인 것으로 보입니다. Layer가 진행될수록 filter의 수는 증가하고 image의 size는 작아지는데, 이를 통해 데이터의 추상화를 이루었습니다. 덕분에 좋은 성능을 달성할 수 있는 것으로 보입니다.

4. Confusion Matrix

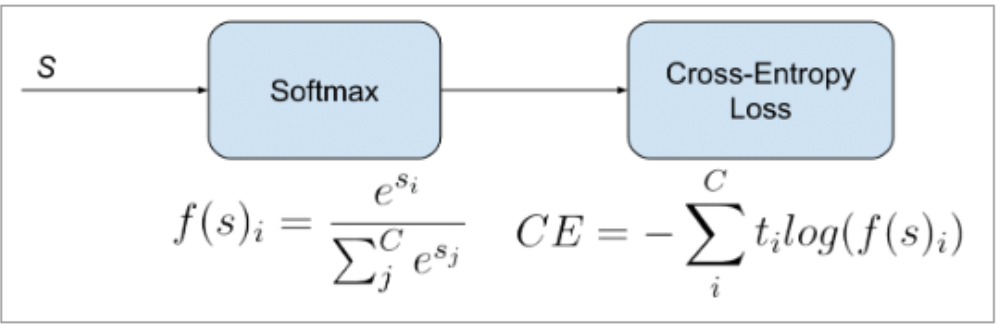


5. Loss function

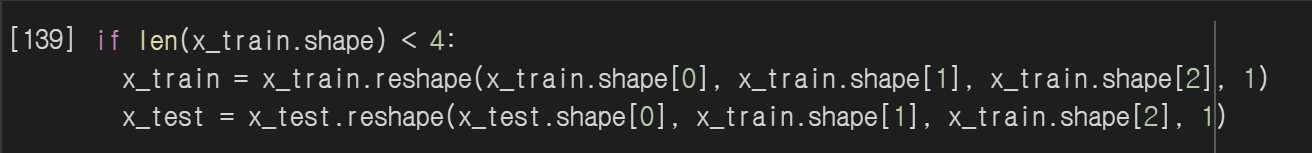


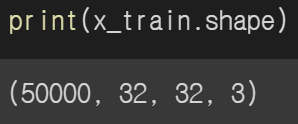
loss function은 keras에 내장된 Categorical Cross Entropy Loss을 사용하였다. 이러한 Cross Entropy Loss는 주로 Multi class의 classification에 사용된다. 우리는 CIFAR-10을 분류하고자 하는데 이는 multiclass임으로 해당 loss function을 사용하였다.

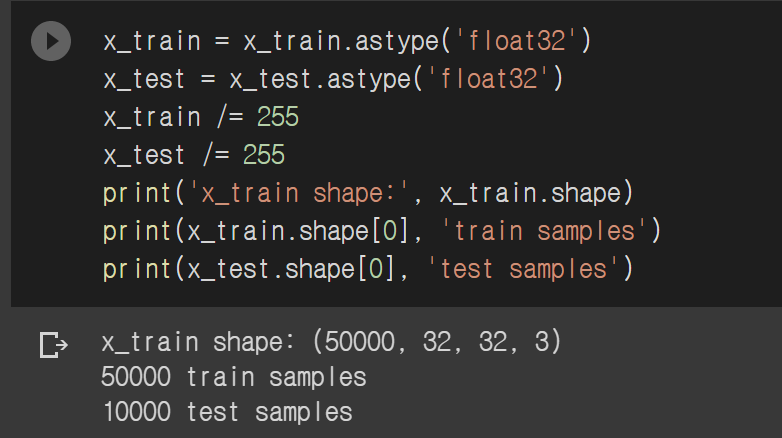
Categorical Cross Entropy Loss Function의 구성은 아래와 같다.



6. 데이터 전처리



기존의 전처리 이전의 데이터의 rank가 만약 4보다 작다면, (1개의 layer만 가지는 경우 등) 학습을 위해 rank를 4로 맞추어 준다. 



그리고 data의 type을 32bit기준 float으로 변경한다. 그리고 각 data를 255로 나누어 주는데 이는 범위를 0~254까지의 값을 갖도록 normalization을 해준 것이다.