19102026_jinyuntae_midterm_report

1. 프로그램 구성 설명

1. Library & Path Setting
2. Data Load
3. Model Architecture
4. Training Step
5. Plot of Training Result
6. Evaluation

2. 하이퍼파라미터(Epochs, Batch_Size)에 따른 모델 성능비교

Library & Path Setting

필요한 라이브러리를 불러옵니다. ImageDataGenerator 결과를 EDA하던 중 scipy가 오류가 나서 추가했습니다.

Image Classification이기에 CNN을 활용하기 위해서 Conv2D, MaxPooling2D를 불러왔으며 이외의 문제조건인 BatchNormalization, Dropout을 제외하곤 multi-layer perceptron 구조를 구현에 필요한 라이브러리입니다.

train_path와 test_path의 경우, 다운로드 받은 cifar10의 폴더에서 train, test로 디렉토리가 구성되어 있기에 각각 경로를 할당했습니다.

```
# library
import keras
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
import matplotlib.pyplot as plt
import os
import scipy



# load Neural Network Model Library => condition 3 of assignment
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten, BatchNormalization
from keras.layers import Conv2D, MaxPooling2D
```

```
# path
train_path = "./cifar10/train/"
test_path = "./cifar10/test/"
```

Data Load

ImageDataGenerator를 통해서 데이터로드 파이프라인을 세팅합니다. (과제조건 1번)

과제 조건에 따라 로드하면서 rescaling을 합니다. (1./255) (과제조건 2번)

validation data를 생성하기 위해 train data를 로드하면서, ImageDataGenerator의 validation_split=0.1 파라미터로 train_path로부터 불러오는 데이터의 10%는 validation data로 할당합니다. (과제조건 6번의 파라미터)

flow_from_directory 메서드를 사용해서 각 path에 연결된 generator pipeline을 만듭니다. 해당 generator 들의 파라미터인 batch_size로, 학습 hyper parameter인 batch_size를 조절할 수 있습니다.
class가 총 10개인 데이터를 분류해야하므로 class_mode는 categorical로 설정합니다.

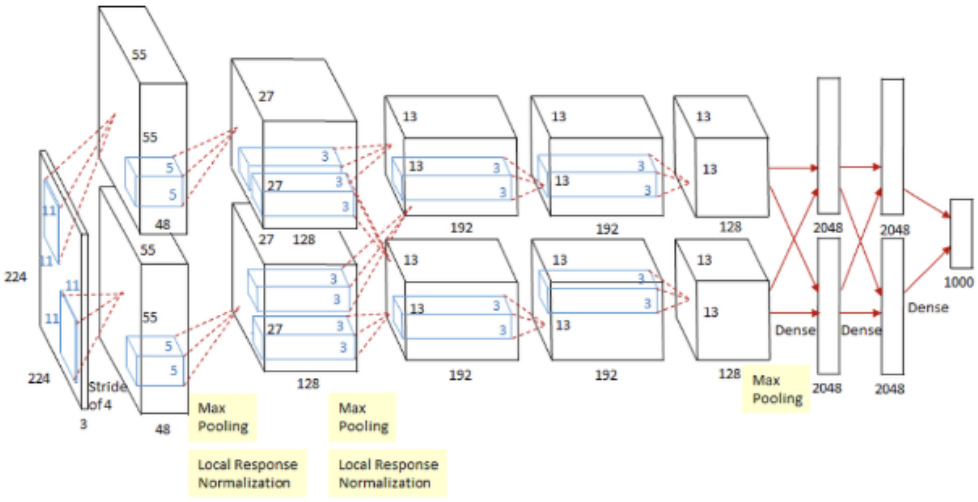
각 generator의 샘플을 뽑아서 살펴본 결과, generator가 batch_size만큼 데이터들을 추출하며 각 데이터는 (256, 256, 3) 형태의 이미지 데이터임을 확인할 수 있습니다. ((batch_size, 256, 256, 3))

```
# Set generator with rescaler(1./255) -> condition 2 of assignment
train_datagen = ImageDataGenerator(rescale=1./255, validation_split=0.1)
test_datagen = ImageDataGenerator(rescale=1./255)

# make a loading data flow from path. It generates data at each batch sizes -> condition 1 of assignment
train_generator = train_datagen.flow_from_directory(train_path,
                                                    classes=sorted(os.listdir(train_path)),
                                                    batch_size = 50,
                                                    target_size = (256, 256),
                                                    subset="training",
                                                    class_mode='categorical')

valid_generator = train_datagen.flow_from_directory(train_path,
                                                    classes=sorted(os.listdir(train_path)),
                                                    batch_size = 50,
                                                    target_size=(256, 256),
                                                    subset="validation",
                                                    class_mode='categorical')


test_generator = test_datagen.flow_from_directory(test_path,
                                                  classes=sorted(os.listdir(test_path)),
                                                  batch_size = 100,
                                                  target_size = (256, 256),
                                                  class_mode='categorical')

print()
# check shape of data shape
print("check shape of data shape")
for x_data, class_data in train_generator:
    print(f"input data shape from train_generator: {x_data.shape}")
    print(f"class data shape from train_generator: {class_data.shape}")
    break

for x_data, class_data in valid_generator:
    print(f"input data shape from valid_generator: {x_data.shape}")
    print(f"class data shape from valid_generator: {class_data.shape}")
    break

for x_data, class_data in test_generator:
    print(f"input data shape from test_generator: {x_data.shape}")
    print(f"class data shape from test_generator: {class_data.shape}")
    break
```

```
Found 45000 images belonging to 10 classes.
Found 5000 images belonging to 10 classes.
Found 10000 images belonging to 10 classes.

check shape of data shape
input data shape from train_generator: (50, 256, 256, 3)
class data shape from train_generator: (50, 10)
input data shape from valid_generator: (50, 256, 256, 3)
class data shape from valid_generator: (50, 10)
input data shape from test_generator: (100, 256, 256, 3)
class data shape from test_generator: (100, 10)
```

Model Architecture

모델의 아키텍쳐는 AlexNet의 구성을 참고했습니다.



Basic AlexNet Architecture

다만 CIFAR10은 최종적으로 10개의 클래스로 이루어져 있으므로 기본 AlexNet에서 output layer만 수정 시에는 성능이 좋지 않았습니다. 학습속도도 gpu를 사용했음에도 매우 느린 것을 확인했습니다.

따라서 Fully Connected Layer 단에서 알맞게 dense layer들의 크기를 수정했으며 Batch Normalization(Batch 형태로 나누어진 데이터를 고려하여 정규화. Local optimum으로 인한 학습속도 저하 및 멈춤을 방지하기 위해 사용), Dropout(뉴런을 부분적으로 생략하여 오버피팅 방지)을 조합하여 사용함으로 학습과정을 보다 효율적으로 만들었습니다.

약 860만 개의 파라미터를 가진 새로운 모델을 생성했습니다.

```
# set neural model architecture (transform AlexNet architecture)

model = Sequential()

# Convolutional layers
model.add(Conv2D(96, (11, 11), strides=(4, 4), padding='valid', activation='relu', input_shape=(256, 256, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))

model.add(Conv2D(256, (5, 5), strides=(1, 1), padding='same', activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))

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

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

model.add(Conv2D(256, (3, 3), strides=(1, 1), padding='same', activation='relu'))

model.add(MaxPooling2D(pool_size=(3, 3), strides=(2, 2)))
model.add(Flatten())

# Fully-connected layers

model.add(Dense(512, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(256, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(128, activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))

model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

model.summary()
```

수정한 모델

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 62, 62, 96)        34944

batch_normalization (BatchN  (None, 62, 62, 96)        384
ormalization)

max_pooling2d (MaxPooling2D  (None, 30, 30, 96)        0
)

conv2d_1 (Conv2D)            (None, 30, 30, 256)       614656

batch_normalization_1 (Batc  (None, 30, 30, 256)       1024
hNormalization)

max_pooling2d_1 (MaxPooling  (None, 14, 14, 256)       0
2D)

conv2d_2 (Conv2D)            (None, 14, 14, 384)       885120

conv2d_3 (Conv2D)            (None, 14, 14, 384)       1327488

conv2d_4 (Conv2D)            (None, 14, 14, 256)       884992

max_pooling2d_2 (MaxPooling  (None, 6, 6, 256)         0
2D)

flatten (Flatten)            (None, 9216)              0

dense (Dense)                (None, 512)               4719104

batch_normalization_2 (Batc  (None, 512)               2048
hNormalization)

dropout (Dropout)            (None, 512)               0

dense_1 (Dense)              (None, 256)               131328

batch_normalization_3 (Batc  (None, 256)               1024
hNormalization)

dropout_1 (Dropout)          (None, 256)               0

dense_2 (Dense)              (None, 128)               32896

batch_normalization_4 (Batc  (None, 128)               512
hNormalization)

dropout_2 (Dropout)          (None, 128)               0

dense_3 (Dense)              (None, 10)                1290

=================================================================
Total params: 8,636,810
Trainable params: 8,634,314
Non-trainable params: 2,496
```

모델 요약

Training Step

fit_generator 메소드를 사용하여 학습했습니다. steps_per_epoch 파라미터의 경우, ImageDatagenerator의 파라미터인 batch_size로 전체 데이터 사이즈를 나누었기에 일반적으로 사용하는 model.fit에서의 iteration과 같습니다.

하이퍼파라미터인 epochs는 여기서 조절하고, batch_size는 Load단에서 ImageDataGenerator의 메서드의 flow_from_directory 파라미터인 batch_size에서 조절합니다.

```
# train with fit_generator method
# steps_per_epoch => same with iterations(train_data_size//batch_size) in model.fit
history = model.fit_generator(
    train_generator,
    steps_per_epoch=train_generator.n//train_generator.batch_size,
    validation_data=valid_generator,
    validation_steps=valid_generator.n//valid_generator.batch_size,
    shuffle=True,
    epochs=20)
```

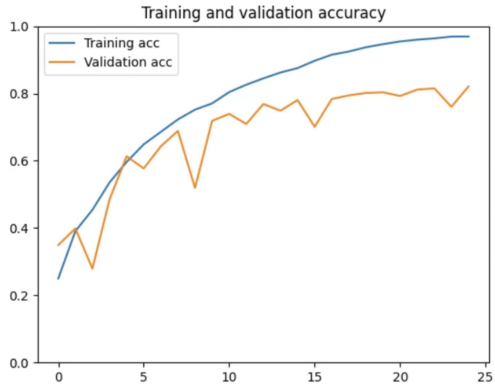
Plot of Training Result

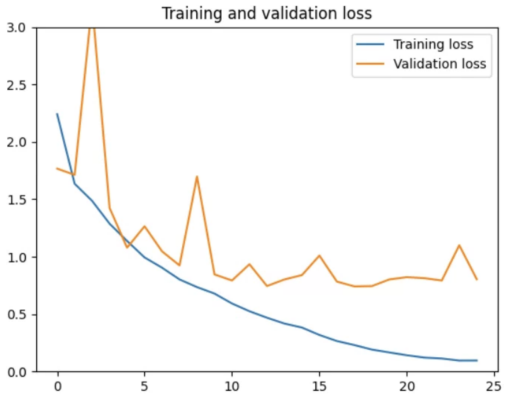
학습과정 기록을 불러와서 확인 후 jpg 형태로 저장합니다.

```
# accuracy plot

acc = history.history['accuracy']
val_acc = history.history['val_accuracy']
epochs = range(len(acc))

plt.plot(epochs, acc, label="Training acc")
plt.plot(epochs, val_acc, label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.ylim(0,1)
plt.show()
plt.savefig('./result_plot/batch_50_epoch_20_acc_plot.jpg', dpi=300)
```

```
# loss plot
acc = history.history['loss']
val_acc = history.history['val_loss']
epochs = range(len(acc))

plt.plot(epochs, acc, label="Training loss")
plt.plot(epochs, val_acc, label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.ylim(0, 3)
plt.show()
plt.savefig('./result_plot/batch_50_epoch_20_loss_plot.jpg', dpi=300)
```

아래는 해당 과제의 accuracy plot과 loss plot의 출력 결과 예시입니다.





Evaluation

학습된 model에 대해서, test 데이터셋을 batch 단위로 불러와서 테스트하고 accuracy 지표를 보입니다. (evaluation)

Evaluation

테스트 데이터에 대해서 학습된 모델의 성능을 확인합니다.

]:
```
# model evaluation
print("-- Evaluate --")
scores = model.evaluate_generator(test_generator, steps=test_generator.n//t
print("%s: %.2f%%" %(model.metrics_names[1], scores[1]*100))
```

```
-- Evaluate --
```

```
<ipython-input-8-8190f70dd792>:3: UserWarning: `Model.evaluate_generator`
is deprecated and will be removed in a future version. Please use `Model.
evaluate`, which supports generators.
  scores = model.evaluate_generator(test_generator, steps=test_generator.
n//test_generator.batch_size)
```

```
accuracy: 81.91%
```

하이퍼파라미터(epochs, Batch_Size)에 따른 모델 성능 비교

최종 모델 비교에서 Hyper Parameter Optimization은 먼저

- batch_size => Train data와 Valid data 크기의 공약수 중 3개
- epoch => 20, 25, 30을 기준으로 진행했습니다.

비교군은 다음과 같습니다. 해당 비교군에 대해서 Grid Search 방식으로 실험을 진행합니다.

(batch_size, epoch) => (100, 20), (100, 25), (100, 30), (200, 20), (200, 25), (200, 30), (250, 20), (250, 25), (250, 30)

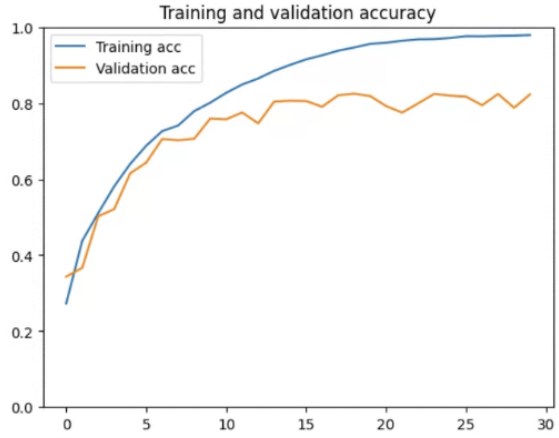
하이퍼파라미터에 따라 학습된 각 모델의 Test 데이터셋에 대한 Accuracy 지표 결과는 다음 표와 같습니다.

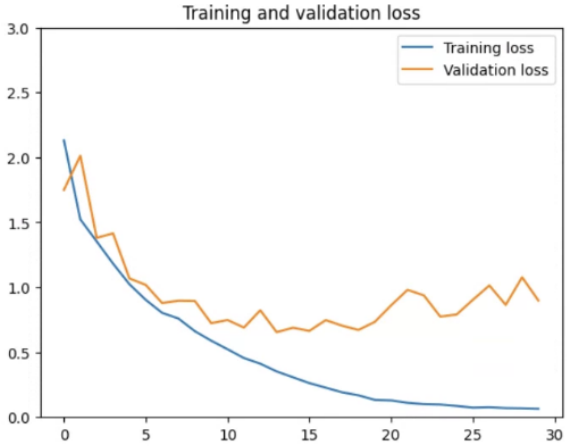
	accuracy
(100, 20)	81.25%
(100, 25)	80.60%
(100, 30)	81.91%
(200, 20)	80.54%
(200, 25)	79.42%
(200, 30)	81.20%
(250, 20)	77.64%
(250, 25)	81.50%
(250, 30)	80.67%

표에 따라서 먼저 가장 좋은 성능을 낸 hyper parameter는 (batch_size, epochs) = (100, 30) 입니다.

학습과정 accuracy, loss 그래프는 아래와 같았습니다.
batch_size 작아서 loss가 작은 폭으로 움직여, 다른 비교 조합에 비해서 validation loss가 비교적 안정적인 그래프를 그리는 것을 볼 수 있었습니다.
validation accuracy 또한 안정적이며 다른 조합에 비해 overfit이 상대적으로 적게 발생했음을 알 수 있었습니다.

학습시간이 조금 더 소요되지만 컴퓨팅 리소스 부담이 적었으며 그래프 변동성이 적어 학습에 적합한 parameter 조합이라고 결론을 지었습니다.


Training and validation accuracy
Training acc
Validation acc
1.0
0.8
0.6
0.4
0.2
0.0
0
5
10
15
20
25
30


Training and validation loss
Training loss
Validation loss
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
5
10
15
20
25
30