인공지능프로그래밍

Lab4

AlexNet for ImageNet Challenge

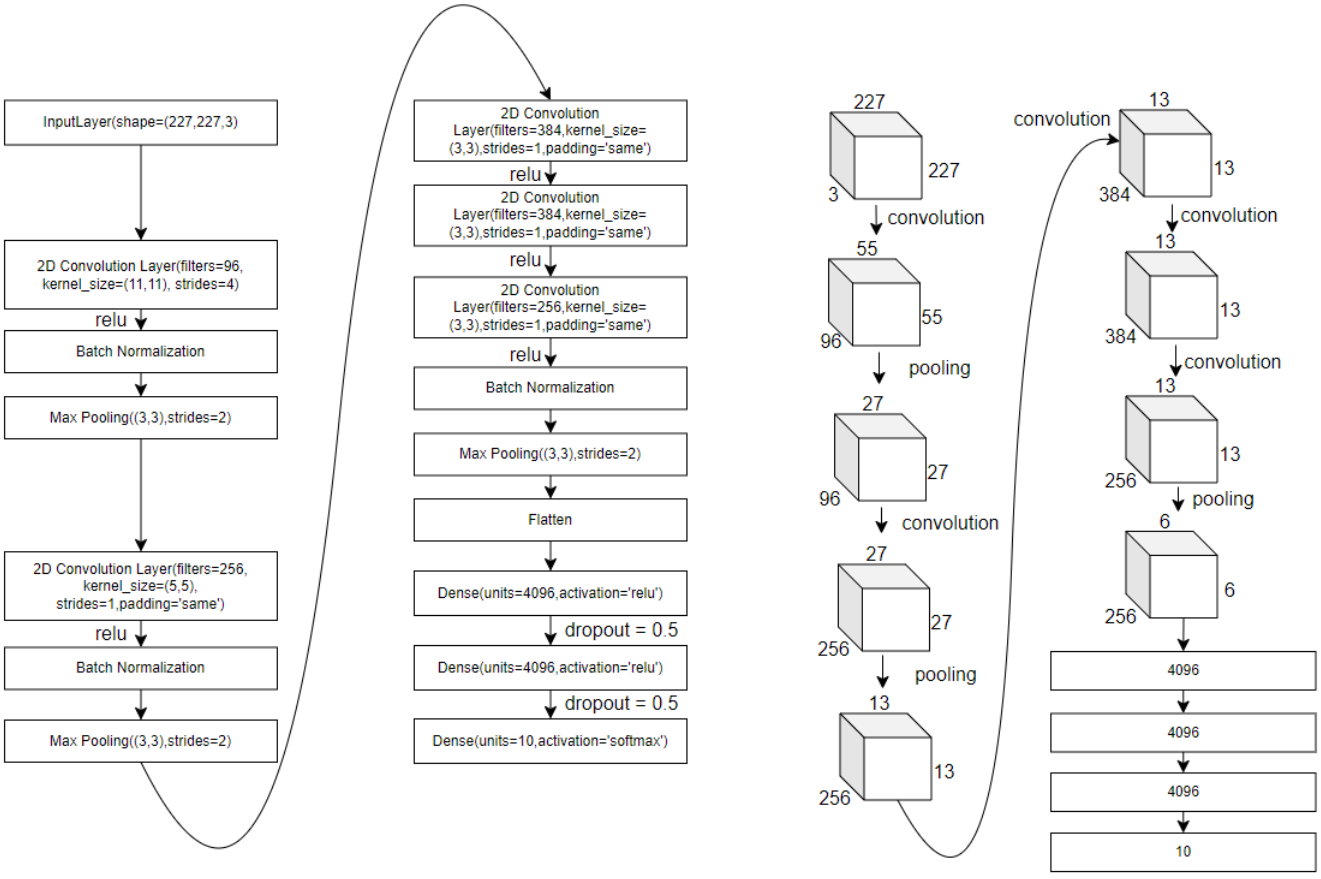
2024/10/12

2019202050 이강현

Lab Objective

CNN 구조에 대해 이해하고 이를 기반으로 AlexNet을 구현해본다. tensorflow로 네트워크를 구성하는 방법을 이해하고 tensorflow dataset을 이용하여 데이터를 불러오고 사용하는 법을 익힌다. 구현된 모델의 parameter를 조정해가며 결과를 추출한다. 생성된 결과를 분석하며 모델의 성능을 추정한다.

Program Flow



모델의 전체적인 흐름을 나타낸다. 왼쪽은 사용된 함수들을 기반으로 모델 속 레이어들을 나타내었고 오른쪽은 데이터가 입력되었을 때 어떤 shape으로 변화해 가는지 그 양상을 확인할 수 있게 표현하였다.

Result

```
import os
os.environ["KERAS_BACKEND"] = "tensorflow"

import numpy as np
import tensorflow as tf
import tensorflow_datasets as tfds
import keras
import matplotlib.pyplot as plt

gpus = tf.config.list_physical_devices('GPU')
print("Num GPUs Available: ", len(gpus))
```

```
Num GPUs Available:  1
```

먼저 필요한 라이브러리를 import하고 사용가능한 gpu를 확인하는 코드이다. gpu가 필요하여 구글 코랩을 사용하여 진행하였다.

```
dataset = 'cifar10'

if dataset == 'cifar10':
    # Load the original CIFAR10 dataset
    # CIFAR10 dataset contains 50000 training images and 10000 test images of 32x32x3 pixels
    # Each image contains a small object such as bird, truck, etc...
    (ds_train, ds_test, ds_val), ds_info = tfds.load('cifar10', split=['train[:80%]', 'test', 'train[80%:]'],
                                                     batch_size=None, shuffle_files=True, as_supervised=True,
                                                     with_info=True)
elif dataset == 'imagenette':
    # Imagenette is a subset of 10 easily classified classes from Imagenet
    # (tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute).
    (ds_train, ds_test, ds_val), ds_info = tfds.load('imagenette/320px-v2', split=['train', 'validation[:50%]', 'validation[50%:]'],
                                                     batch_size=None, shuffle_files=True, as_supervised=True,
                                                     with_info=True)
else:
    print('Dataset Error')

print(ds_info.features)
print(ds_info.splits)
print(ds_info.splits['train'].num_examples)
```

dataset은 우선 cifar10을 사용하여 진행하였다. 원하는 데이터셋을 선택하고 데이터셋을 불러와 train,valid,test로 split할 수 있고 이 과정에서 분할할 양이나 데이터를 섞을지 여부등을 선택할 수 있다.

```
Dataset cifar10 downloaded and prepared to /root/tensorflow_datasets/cifar10/3.0.2. Subsequent calls will reuse this data.
FeaturesDict({
    'id': Text(shape=(), dtype=string),
    'image': Image(shape=(32, 32, 3), dtype=uint8),
    'label': ClassLabel(shape=(), dtype=int64, num_classes=10),
})
{Split('train'): <SplitInfo num_examples=50000, num_shards=1>, Split('test'): <SplitInfo num_examples=10000, num_shards=1>}
50000
```

데이터의 정보를 출력한 결과 cifar10과 같은 경우 32,32,3의 형태로 저장되어있고 10개의 라벨을 가지고 있음을 대략적으로 확인할 수 있다.

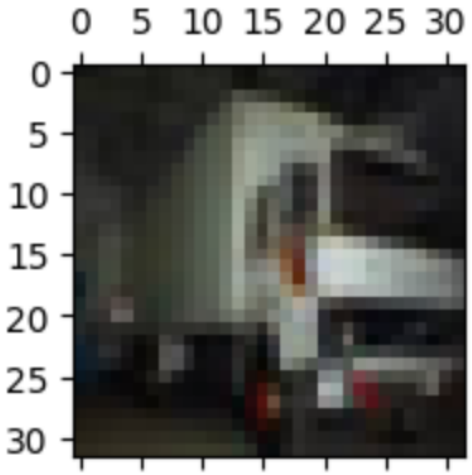
```
n_channels = ds_info.features['image'].shape[-1]

if dataset == 'imagenette':
    classes = ['tench', 'English springer', 'cassette player', 'chain saw',
               'church', 'French horn', 'garbage truck', 'gas pump',
               'golf ball', 'parachute']
else:
    classes = ds_info.features['label'].names
n_classes = ds_info.features['label'].num_classes

n_train = len(ds_train)
n_test = len(ds_test)
n_val = len(ds_val)

print(n_train,n_test,n_val)
```

```
40000 10000 10000
```

위 코드를 이용하여 채널 수나 클래스 수를 저장하고 데이터들이 몇개씩 train,test,validation으로 분할되었는지 확인할 수 있다.

```
idx = np.random.randint(n_train-1)

for element in ds_train.skip(idx).take(1):
    image, label = element

print('Image demension:', image.shape, ', label:',label.numpy())

dimage = tf.reshape(image, image.shape)
plt.figure(figsize=(2,2))
plt.matshow(dimage, 1)
plt.show()
print('The picture is', classes[label])
```

```
Image demension: (32, 32, 3) , label: 9
```




```
The picture is truck
```

데이터중 하나를 무작위로 선택하여 확인해본 결과 이미지 하나의 shape은 32x32x3으로 3채널 32x32이미지임을 확인할 수 있다. 사진이 어떻게 생겼는지 시각화하여 확인할 수 있다.

```
def tfds_4_NET(image, label):

    image = tf.image.resize((image / 255), [227,227], method='bilinear')
    label = tf.one_hot(label, n_classes)

    return image, label
```

```
n_batch = 64

dataset = ds_train.map(tfds_4_NET, num_parallel_calls=tf.data.AUTOTUNE)
dataset = dataset.shuffle(buffer_size = 256).batch(batch_size=n_batch)

valiset = ds_val.map(tfds_4_NET, num_parallel_calls=tf.data.AUTOTUNE)
valiset = valiset.shuffle(buffer_size = 256).batch(batch_size=n_batch)

testset = ds_test.map(tfds_4_NET, num_parallel_calls=tf.data.AUTOTUNE)
testset = testset.shuffle(buffer_size = 256).batch(batch_size=n_batch)
```

tfds_4_NET이라는 함수를 정의한다. 해당 함수는 0~255까지의 값을 가지는 픽셀로 이루어진 이미지에 255로 나누어 0부터 1사이의 값을 가지도록 정규화를 진행하고 227x227로 resize함으로써 AlexNet 모델이 원하는 입력으로 데이터를 바꾸는 함수이다. label은 one-hot encoding을 함으로써 추후 classification를 진행할 때 softmax에 적합하게 바꾸는 것을 확인한다.

batch 크기는 64로 설정하였고 map함수를 통해 각 데이터셋에 함수를 적용하고 무작위로 섞는다.

```
# model
AlexNet = keras.Sequential([
    ### START CODE HERE ###
    # Input Layer
    keras.layers.InputLayer(shape=(227,227,3)),

    # Layer 1
    keras.layers.Conv2D(filters=96,kernel_size=(11,11),strides=4,activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPooling2D((3,3),strides=2),

    # Layer 2
    keras.layers.Conv2D(filters=256,kernel_size=(5,5),strides=1,padding='same',activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPooling2D((3,3),strides=2),

    # Layer 3 to 5
    keras.layers.Conv2D(filters=384,kernel_size=(3,3),strides=1,padding='same',activation='relu'),
    keras.layers.Conv2D(filters=384,kernel_size=(3,3),strides=1,padding='same',activation='relu'),
    keras.layers.Conv2D(filters=256,kernel_size=(3,3),strides=1,padding='same',activation='relu'),
    keras.layers.BatchNormalization(),
    keras.layers.MaxPooling2D((3,3),strides=2),
    keras.layers.Flatten(),
    # Layer 6 to 8
    keras.layers.Dense(units=4096,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=4096,activation='relu'),
    keras.layers.Dropout(0.5),
    keras.layers.Dense(units=10,activation='softmax')

    ### END CODE HERE ###
])

AlexNet.summary()
```

위 코드는 과제의 메인이 되는 model을 구성하는 부분이다. AlexNet의 구조에 맞게 그대로 구현하였다. 아래 사진을 통해 구현된 모델을 한눈에 확인할 수 있다.

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 55, 55, 96)	34,944
batch_normalization (BatchNormalization)	(None, 55, 55, 96)	384
max_pooling2d (MaxPooling2D)	(None, 27, 27, 96)	0
conv2d_1 (Conv2D)	(None, 27, 27, 256)	614,656
batch_normalization_1 (BatchNormalization)	(None, 27, 27, 256)	1,024
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 256)	0
conv2d_2 (Conv2D)	(None, 13, 13, 384)	885,120
conv2d_3 (Conv2D)	(None, 13, 13, 384)	1,327,488
conv2d_4 (Conv2D)	(None, 13, 13, 256)	884,992
batch_normalization_2 (BatchNormalization)	(None, 13, 13, 256)	1,024
max_pooling2d_2 (MaxPooling2D)	(None, 6, 6, 256)	0
flatten (Flatten)	(None, 9216)	0
dense (Dense)	(None, 4096)	37,752,832
dropout (Dropout)	(None, 4096)	0
dense_1 (Dense)	(None, 4096)	16,781,312
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 10)	40,970

Total params: 58,324,746 (222.49 MB)
Trainable params: 58,323,530 (222.49 MB)
Non-trainable params: 1,216 (4.75 KB)

convolution layer, batch normalization, max pooling, flatten, dense, dropout이 사용되었고 이들이 적절히 조합된 모습을 한 CNN임을 확인할 수 있다. 초기 convolution을 비교적 큰 커널 사이즈인 11x11과 4만큼 stride를 주어 이미지 크기가 크게 줄어드는 것을 확인한다. 이후 커널 크기와 stride를 줄이고 이미지 크기가 convolution layer에서는 줄어들지 않도록 입력의 이미지 크기와 출력의 이미지 크기가 유지되도록 padding처리를 하는 것을 확인한다. 5번의 convolution 연산 후 flatten하여 fully connected layer를 통과한 후 마지막 레이어에서는 classification을 위해 relu가 아닌 softmax activation function을 사용한 것을 확인할 수 있다. 추가적으로 dense layer 사이에 dropout regularization을 적용한 것은

overfitting을 방지하기 위함으로 보여진다.

```
opt = keras.optimizers.Adam(learning_rate=0.001)
AlexNet.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['acc'], jit_compile=True)
```

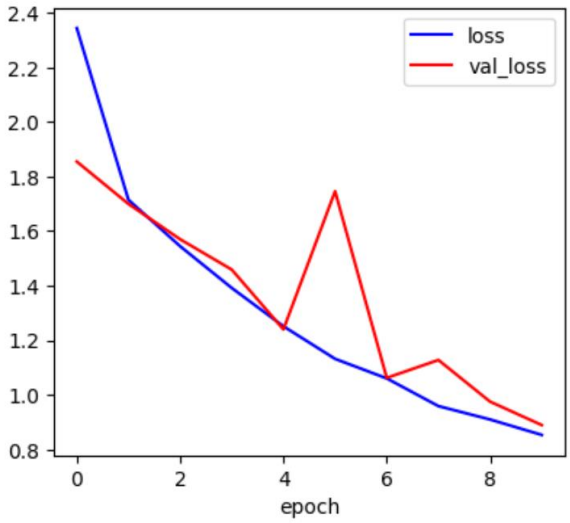
```
n_epochs = 40

results = AlexNet.fit(dataset, epochs=n_epochs, batch_size=n_batch,
                      validation_data=valiset, validation_batch_size=n_batch,
                      verbose=1)
```

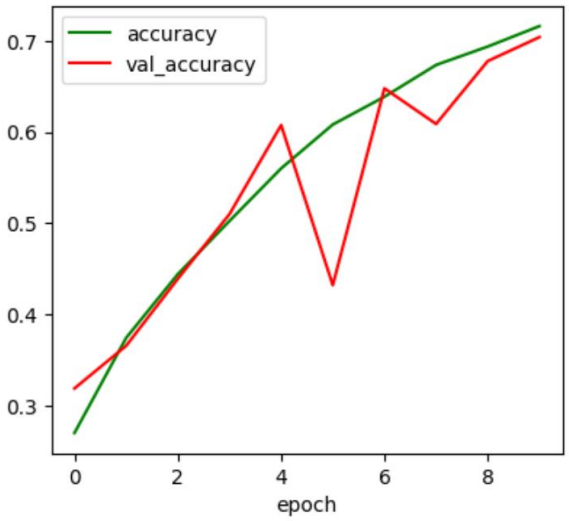
optimizer는 Adam을 사용하였고 모델을 미리 기계어로 컴파일하여 실행속도를 빠르게 하려고 하는 것을 확인할 수 있다. epoch은 원래 10으로 주어졌으나 추가로 loss가 더 떨어질 수 있을 것으로 예상하여 40으로 넣어 model을 학습시켜보았다.

```
# plot loss and accuracy
plt.figure(figsize=(10,4))
plt.subplot(1,2,1)
plt.plot(results.history['loss'], 'b-', label='loss')
plt.plot(results.history['val_loss'], 'r-', label='val_loss')
plt.xlabel('epoch')
plt.legend()

plt.subplot(1,2,2)
plt.plot(results.history['acc'], 'g-', label='accuracy')
plt.plot(results.history['val_acc'], 'r-', label='val_accuracy')
plt.xlabel('epoch')
plt.legend()

plt.show()
```

위 코드를 통해 결과 그래프를 출력하였다.

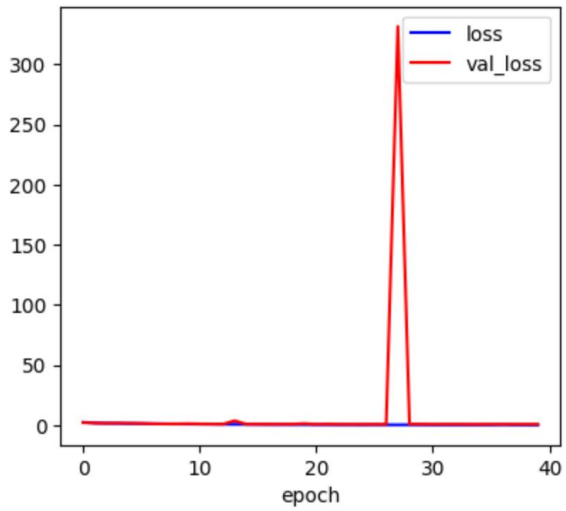
```
Epoch 1/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 88s 117ms/step - acc: 0.2280 - loss: 3.8989 - val_acc: 0.3191 - val_loss: 1.8548
Epoch 2/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 70s 111ms/step - acc: 0.3543 - loss: 1.7697 - val_acc: 0.3658 - val_loss: 1.6998
Epoch 3/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 78s 106ms/step - acc: 0.4296 - loss: 1.5867 - val_acc: 0.4391 - val_loss: 1.5704
Epoch 4/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 83s 107ms/step - acc: 0.4876 - loss: 1.4333 - val_acc: 0.5099 - val_loss: 1.4591
Epoch 5/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 83s 109ms/step - acc: 0.5475 - loss: 1.2895 - val_acc: 0.6074 - val_loss: 1.2400
Epoch 6/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 80s 107ms/step - acc: 0.6004 - loss: 1.1553 - val_acc: 0.4322 - val_loss: 1.7463
Epoch 7/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 66s 106ms/step - acc: 0.6334 - loss: 1.0937 - val_acc: 0.6477 - val_loss: 1.0625
Epoch 8/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 85s 111ms/step - acc: 0.6667 - loss: 0.9822 - val_acc: 0.6087 - val_loss: 1.1280
Epoch 9/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 66s 106ms/step - acc: 0.6888 - loss: 0.9215 - val_acc: 0.6776 - val_loss: 0.9754
Epoch 10/10
625/625 ━━━━━━━━━━━━━━━━━━━━ 87s 114ms/step - acc: 0.7086 - loss: 0.8718 - val_acc: 0.7038 - val_loss: 0.8897
```

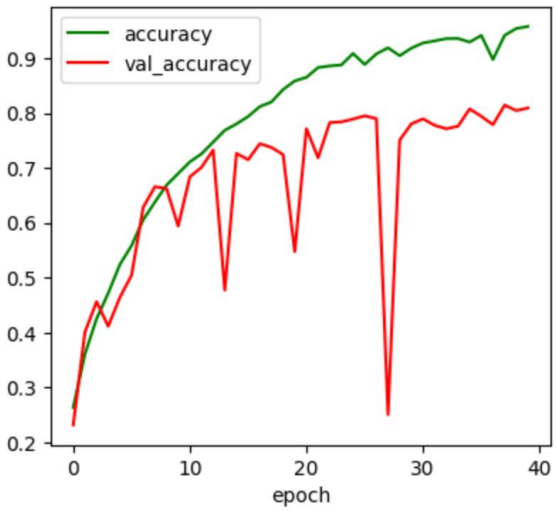




먼저 10만큼 학습시켜본 결과이다. 한번 validation loss와 accuracy가 튀기는 했지만 epoch이 진행될 수록 성능이 올라가는 추세를 보인다. 따라서 40만큼 학습하도록 해보았다.

```
Epoch 1/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 101s 129ms/step - acc: 0.2188 - loss: 4.0173 - val_acc: 0.2314 - val_loss: 2.1290
Epoch 2/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 115s 113ms/step - acc: 0.3335 - loss: 1.8160 - val_acc: 0.4005 - val_loss: 1.7582
Epoch 3/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 71s 113ms/step - acc: 0.4075 - loss: 1.6411 - val_acc: 0.4562 - val_loss: 1.5472
Epoch 4/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 68s 108ms/step - acc: 0.4571 - loss: 1.5061 - val_acc: 0.4114 - val_loss: 1.6832
Epoch 5/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 85s 113ms/step - acc: 0.5114 - loss: 1.3771 - val_acc: 0.4640 - val_loss: 1.5438
Epoch 6/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 70s 112ms/step - acc: 0.5562 - loss: 1.2774 - val_acc: 0.5044 - val_loss: 1.4238
Epoch 7/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 72s 115ms/step - acc: 0.5955 - loss: 1.1685 - val_acc: 0.6281 - val_loss: 1.1318
Epoch 8/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 83s 117ms/step - acc: 0.6308 - loss: 1.0799 - val_acc: 0.6658 - val_loss: 1.0125
Epoch 9/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 68s 108ms/step - acc: 0.6644 - loss: 0.9807 - val_acc: 0.6625 - val_loss: 1.0263
Epoch 10/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 73s 116ms/step - acc: 0.6795 - loss: 0.9536 - val_acc: 0.5941 - val_loss: 1.2014
Epoch 11/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 116ms/step - acc: 0.7054 - loss: 0.8766 - val_acc: 0.6838 - val_loss: 1.0173
Epoch 12/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 73s 116ms/step - acc: 0.7231 - loss: 0.8308 - val_acc: 0.7011 - val_loss: 0.9212
Epoch 13/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 83s 117ms/step - acc: 0.7469 - loss: 0.7649 - val_acc: 0.7326 - val_loss: 0.8081
Epoch 14/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 69s 110ms/step - acc: 0.7621 - loss: 0.7138 - val_acc: 0.4773 - val_loss: 3.6443
Epoch 15/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 73s 116ms/step - acc: 0.7751 - loss: 0.6847 - val_acc: 0.7265 - val_loss: 0.8190
Epoch 16/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 116ms/step - acc: 0.7885 - loss: 0.6379 - val_acc: 0.7150 - val_loss: 0.8135
Epoch 17/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 67s 108ms/step - acc: 0.7998 - loss: 0.6089 - val_acc: 0.7442 - val_loss: 0.7723
Epoch 18/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 85s 112ms/step - acc: 0.8191 - loss: 0.5587 - val_acc: 0.7374 - val_loss: 0.7866

Epoch 20/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 68s 109ms/step - acc: 0.8557 - loss: 0.4521 - val_acc: 0.5474 - val_loss: 1.2724
Epoch 21/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 108ms/step - acc: 0.8479 - loss: 0.4758 - val_acc: 0.7716 - val_loss: 0.6919
Epoch 22/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 72s 115ms/step - acc: 0.8792 - loss: 0.3793 - val_acc: 0.7184 - val_loss: 0.9203
Epoch 23/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 72s 115ms/step - acc: 0.8834 - loss: 0.3566 - val_acc: 0.7827 - val_loss: 0.7034
Epoch 24/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 72s 116ms/step - acc: 0.8754 - loss: 0.3943 - val_acc: 0.7839 - val_loss: 0.6672
Epoch 25/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 67s 107ms/step - acc: 0.9038 - loss: 0.2999 - val_acc: 0.7891 - val_loss: 0.7212
Epoch 26/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 87s 116ms/step - acc: 0.8860 - loss: 0.3744 - val_acc: 0.7950 - val_loss: 0.7049
Epoch 27/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 72s 116ms/step - acc: 0.9126 - loss: 0.2860 - val_acc: 0.7899 - val_loss: 0.6763
Epoch 28/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 72s 115ms/step - acc: 0.9160 - loss: 0.2657 - val_acc: 0.2502 - val_loss: 331.662
Epoch 29/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 69s 109ms/step - acc: 0.8848 - loss: 0.3964 - val_acc: 0.7508 - val_loss: 0.8198
Epoch 30/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 67s 107ms/step - acc: 0.9200 - loss: 0.2601 - val_acc: 0.7804 - val_loss: 0.8957
Epoch 31/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 84s 110ms/step - acc: 0.9217 - loss: 0.2629 - val_acc: 0.7894 - val_loss: 0.7048
Epoch 32/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 85s 115ms/step - acc: 0.9279 - loss: 0.2390 - val_acc: 0.7777 - val_loss: 0.7730
Epoch 33/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 82s 116ms/step - acc: 0.9327 - loss: 0.2200 - val_acc: 0.7714 - val_loss: 0.8173
Epoch 34/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 72s 115ms/step - acc: 0.9406 - loss: 0.1995 - val_acc: 0.7760 - val_loss: 0.7269

Epoch 35/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 80s 113ms/step - acc: 0.9207 - loss: 0.2701 - val_acc: 0.8076 - val_loss: 0.6919
Epoch 36/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 84s 116ms/step - acc: 0.9422 - loss: 0.1993 - val_acc: 0.7941 - val_loss: 0.7795
Epoch 37/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 78s 109ms/step - acc: 0.9238 - loss: 0.2565 - val_acc: 0.7788 - val_loss: 0.7309
Epoch 38/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 68s 108ms/step - acc: 0.9348 - loss: 0.2146 - val_acc: 0.8145 - val_loss: 0.7300
Epoch 39/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 86s 115ms/step - acc: 0.9531 - loss: 0.1557 - val_acc: 0.8046 - val_loss: 0.7655
Epoch 40/40
625/625 ━━━━━━━━━━━━━━━━━━━━ 78s 109ms/step - acc: 0.9564 - loss: 0.1531 - val_acc: 0.8093 - val_loss: 0.7942
```





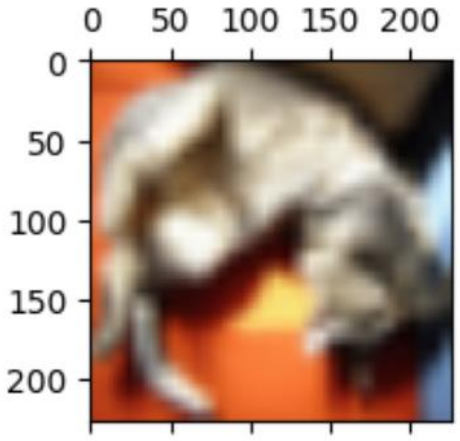
40번 학습시킨 결과 28번째 학습에서 한번 크게 요동쳐 loss 그래프가 미세한 차이를 확인할 수 없게 되었지만 accuracy를 확인해보면 미세하게 올라가는 추세임을 확인한다. epoch 35에서 validation loss가 가장 낮았고 80.76%의 정확도를 보이면서 더 높은 성능을 얻었다.

```
AlexNet.evaluate(testset)
```

```
157/157 ━━━━━━━━━━━━━━━━━━━━━━━━━ 10s 63ms/step - acc: 0.8062 - loss: 0.8301
[0.8443588614463806, 0.8040000200271606]
```

test를 통해 성능을 확인해보니 80.62%의 정확도를 얻었다.

```
idx = np.random.randint(n_test-1)

for element in ds_test.skip(idx).take(1):
    img, lbl = element
    X_test, y_test = tfds_4_NET(img, lbl)

X_test = tf.expand_dims(X_test, axis=0)

dimage = np.array(X_test[0])
plt.figure(figsize=(2,2))
plt.matshow(dimage, fignum=1)
plt.show()

outt_4 = AlexNet.predict(X_test)
p_pred = np.argmax(outt_4, axis=-1)

print('My prediction is ' + classes[p_pred[0]])
print('Actual image is  ' + classes[tf.argmax(y_test, -1)])
```




```
1/1 ——————————————————————————— 1s 1s/step
My prediction is frog
Actual image is  cat
```

마지막으로 하나의 샘플을 뽑아내어 예측해본 결과 예측에 실패한 것을 확인했다. 무작위로 뽑은 샘플이 오답인 경우였을 것으로 예상한다.

Discussion

tensorflow를 이용하여 CNN을 구현해보았다. tensorflow라는 새로운 프레임워크를 사용하여 구현하였어서 익숙하지 않아 어려움이 있었다. AlexNet을 그대로 구현하는 것이라 모델 구현에 있어서는 상대적으로 DNN을 구현할때보다 쉬웠으나 데이터가 어떻게 변하는지와 같은 것을 summary 함수가 없었다면 이해하기 어려웠을 것 같다. 특히 그대로 convolution만 하였을 때 이미지 크기가 expected output과 맞지 않아서 어려움이 있었는데 이는 padding부분에 same을 주어 입력크기와 출력 크기가 맞도록 하여 해결했다. 마지막으로 local에서 코드를 구현하고 싶었으나 tensorflow 2.10까지만 gpu를 window환경에서 제공하고 이후 버전은 linux나 WSL에서 특정 라이브러리를 깔아야 했는데 tensorflow 이전 버전과 dataset과 tensorboard 버전에서 충돌이 나서 구글 코랩을 사용해야만 했다. 이때문에 gpu를 계속 사용할 수 없어서 중간중간 cpu를 사용했는데 실행 속도를 차이를 보고 CNN 구조를 실행하는데 많은 리소스가 필요하다는 것을 알았다.