시스템 소프트웨어
Sequential model 구현 및 비교분석

20191016 최정윤

덕성여자대학교 IT미디어공학과

Contents

MNIST

코드분석

```
[1] from tensorflow.keras import datasets
    import tensorflow as tf
```

```
[2] mnist = datasets.mnist
    (X_train, t_train), (X_test, t_test) = mnist.load_data()

    X_train.shape, t_train.shape

    image = X_train[0]
    image.shape

    Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
    11490434/11490434 [==============================] - 0s 0us/step
    (28, 28)
```

Mnist 파일을 가져와 잘 불러와 졌는지 확인하고 2차원 이미지 데이터 shape을 확인한다.

```
[5] X_train = X_train[..., tf.newaxis]
    X_test = X_test[..., tf.newaxis]

    X_train.shape, X_test.shape

    ((60000, 28, 28, 1), (10000, 28, 28, 1))
```

```
[6] X_train = X_train / 255.0
    X_test = X_test / 255.0

    import numpy as np
    np.min(X_train), np.max(X_train)

    (0.0, 1.0)
```

Mnist 내의 3차원 데이터인 x_train에 channel을 추가하여 4차원으로 만들어준다.

이미지 값은 0에서 255로 되어 있지만 tensorflow에서 작업은 0~1 사이의 float 값일 때 학습을 더 잘하기 때문에 이미지를 255.0으로 나누어 준다.

MNIST

코드분석

Sequential model 구조 만들기

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, ReLU, MaxPooling2D, Flatten, Dense

model = Sequential([
    # 입력층에는 배치를 제외한 나머지 이미지의 형상을 받아야 한다.
    # Feature Extraction
    Conv2D(filters=64, kernel_size=3, padding='SAME', input_shape=(28, 28, 1)),
    ReLU(),
    Conv2D(filters=64, kernel_size=3, padding='SAME'),
    ReLU(),
    MaxPooling2D(pool_size=2),

    Conv2D(filters=32, kernel_size=3, padding='SAME', activation='relu'),
    Conv2D(filters=32, kernel_size=3, padding='SAME', activation='relu'),
    MaxPooling2D(pool_size=2),
    # Fully Connected
    Flatten(),
    Dense(512, activation='relu'),
    Dense(10, activation='softmax')
])

model.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 64)	640
re_lu (ReLU)	(None, 28, 28, 64)	0
conv2d_1 (Conv2D)	(None, 28, 28, 64)	36928
re_lu_1 (ReLU)	(None, 28, 28, 64)	0
max_pooling2d (MaxPooling2D)	(None, 14, 14, 64)	0
conv2d_2 (Conv2D)	(None, 14, 14, 32)	18464
conv2d_3 (Conv2D)	(None, 14, 14, 32)	9248
max_pooling2d_1 (MaxPooling2D)	(None, 7, 7, 32)	0
flatten (Flatten)	(None, 1568)	0
dense (Dense)	(None, 512)	803328
dense_1 (Dense)	(None, 10)	5130

Total params: 873,738
Trainable params: 873,738
Non-trainable params: 0

위 코드는 교재를 참고 하여 작성한 sequential model 코드이다.
Activation함수로 relu를 사용하고 활성화 함수로 softmax를 활용하였다.
입력층에는 배치를 제외한 나머지 이미지의 형상을 받아야 하기 때문에 위와 같이 input_shape을 설정해주었다.

MNIST

비교분석

```
[15] model.compile(
        optimizer = tf.keras.optimizers.Adam(),
        loss = tf.keras.losses.sparse_categorical_crossentropy,
        metrics=['acc']
    )
```

옵티마이저 : adam( )

mnist_sequential.ipynb

파일 수정 보기 삽입 런타임 도구 도움말 모든 변경사항이 저장됨

```
model.fit(
    X_train, t_train,
    validation_split =  0.2,
    epochs=10,
    batch_size=32
    )
```

```
Epoch 1/10
1500/1500 [==============================] - 392s 261ms/step - loss: 0.1227 - acc: 0.9610 - val_loss: 0.0530 - val_acc: 0.9837
Epoch 2/10
1500/1500 [==============================] - 378s 252ms/step - loss: 0.0409 - acc: 0.9870 - val_loss: 0.0417 - val_acc: 0.9887
Epoch 3/10
1500/1500 [==============================] - 379s 253ms/step - loss: 0.0282 - acc: 0.9909 - val_loss: 0.0385 - val_acc: 0.9891
Epoch 4/10
1500/1500 [==============================] - 379s 253ms/step - loss: 0.0215 - acc: 0.9936 - val_loss: 0.0443 - val_acc: 0.9869
Epoch 5/10
1500/1500 [==============================] - 376s 251ms/step - loss: 0.0172 - acc: 0.9944 - val_loss: 0.0494 - val_acc: 0.9865
Epoch 6/10
1500/1500 [==============================] - 377s 251ms/step - loss: 0.0142 - acc: 0.9955 - val_loss: 0.0454 - val_acc: 0.9886
Epoch 7/10
1500/1500 [==============================] - 376s 251ms/step - loss: 0.0117 - acc: 0.9963 - val_loss: 0.0486 - val_acc: 0.9899
Epoch 8/10
1500/1500 [==============================] - 376s 251ms/step - loss: 0.0101 - acc: 0.9971 - val_loss: 0.0438 - val_acc: 0.9922
Epoch 9/10
1500/1500 [==============================] - 379s 253ms/step - loss: 0.0100 - acc: 0.9967 - val_loss: 0.0438 - val_acc: 0.9912
Epoch 10/10
1500/1500 [==============================] - 379s 253ms/step - loss: 0.0075 - acc: 0.9977 - val_loss: 0.0497 - val_acc: 0.9902
<keras.callbacks.History at 0x7fd48643c940>
```

[파라미터 정보]

validation_split = 0.2,
epochs=10,
batch_size=32

```
[ ] image = X_test[0]
    image.shape

    image = image[tf.newaxis, ...]
    image.shape

    model.predict(image)
```

```
1/1 [==============================] - 0s 254ms/step
array([[5.8267450e-23, 1.3324473e-16, 9.5818571e-17, 3.1765009e-16,
        5.9193732e-19, 4.3669660e-21, 1.0791384e-25, 1.0000000e+00,
        4.7481764e-18, 4.4688003e-11]], dtype=float32)
```

✓ 0초 오전 11:29에 완료됨

파라미터 정보는 위와 같고
Loss 값은 0.0075 accuracy값은 0.9977이다.

MNIST

비교분석

```
[15] model.compile(
        optimizer = tf.keras.optimizers.Adam(),
        loss = tf.keras.losses.sparse_categorical_crossentropy,
        metrics=['acc']
    )
```

옵티마이저 : adam( )

mnist_sequential.ipynb

파일 수정 보기 삽입 런타임 도구 도움말 모든 변경사항이 저장됨

```
[10] model.fit(
        X_train, t_train,
        validation_split =  0.2,
        epochs=10,
        batch_size=64
        )
```

```
Epoch 1/10
750/750 [==============================] - 315s 420ms/step - loss: 0.0580 - acc: 0.9818 - val_loss: 0.0417 - val_acc: 0.9868
Epoch 2/10
750/750 [==============================] - 311s 414ms/step - loss: 0.0321 - acc: 0.9895 - val_loss: 0.0330 - val_acc: 0.9899
Epoch 3/10
750/750 [==============================] - 312s 416ms/step - loss: 0.0245 - acc: 0.9923 - val_loss: 0.0332 - val_acc: 0.9902
Epoch 4/10
750/750 [==============================] - 314s 419ms/step - loss: 0.0172 - acc: 0.9944 - val_loss: 0.0301 - val_acc: 0.9914
Epoch 5/10
750/750 [==============================] - 314s 419ms/step - loss: 0.0162 - acc: 0.9947 - val_loss: 0.0287 - val_acc: 0.9909
Epoch 6/10
750/750 [==============================] - 313s 418ms/step - loss: 0.0133 - acc: 0.9955 - val_loss: 0.0404 - val_acc: 0.9898
Epoch 7/10
750/750 [==============================] - 314s 419ms/step - loss: 0.0115 - acc: 0.9959 - val_loss: 0.0379 - val_acc: 0.9899
Epoch 8/10
750/750 [==============================] - 312s 416ms/step - loss: 0.0082 - acc: 0.9974 - val_loss: 0.0393 - val_acc: 0.9912
Epoch 9/10
750/750 [==============================] - 313s 417ms/step - loss: 0.0082 - acc: 0.9976 - val_loss: 0.0330 - val_acc: 0.9918
Epoch 10/10
750/750 [==============================] - 313s 417ms/step - loss: 0.0064 - acc: 0.9981 - val_loss: 0.0307 - val_acc: 0.9923
<keras.callbacks.History at 0x7fdb5271fc10>
```

```
[11] image = X_test[0]
     image.shape

     image = image[tf.newaxis, ...]
     image.shape

     model.predict(image)
```

```
1/1 [==============================] - 0s 121ms/step
array([[7.5083874e-13, 7.6194268e-10, 2.1193353e-11, 1.0227884e-09,
        3.8092827e-14, 7.7747418e-15, 2.3411596e-16, 9.9999988e-01,
        5.2883076e-12, 9.3412439e-08]], dtype=float32)
```

0초 오후 12:34에 완료됨

[파라미터 정보]

validation_split = 0.2,
epochs=10,
batch_size=64

배치사이즈를 32에서 64로 변경하여 진행하여 보았다.
Loss 값은 0.0054 accuracy값은 0.9981이다.
배치사이즈 늘리니 정확도가 증가하고 손실 값이 낮아졌다.

MNIST

비교분석

```
model.compile(
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.01),
    loss = tf.keras.losses.sparse_categorical_crossentropy,
    metrics=['acc']
)
```

옵티마이저 : adam(learning_rate=0.01)

```
model.fit(
    X_train, t_train,
    validation_split =  0.2,
    epochs=10,
    batch_size=64
    )
```

```
Epoch 1/10
750/750 [==============================] - 333s 444ms/step - loss: 0.0851 - acc: 0.9753 - val_loss: 0.1000 - val_acc: 0.9709
Epoch 2/10
750/750 [==============================] - 328s 438ms/step - loss: 0.0827 - acc: 0.9773 - val_loss: 0.0660 - val_acc: 0.9800
Epoch 3/10
750/750 [==============================] - 330s 440ms/step - loss: 0.0778 - acc: 0.9785 - val_loss: 0.0781 - val_acc: 0.9781
Epoch 4/10
750/750 [==============================] - 327s 436ms/step - loss: 0.0763 - acc: 0.9790 - val_loss: 0.0669 - val_acc: 0.9805
Epoch 5/10
750/750 [==============================] - 330s 440ms/step - loss: 0.0604 - acc: 0.9837 - val_loss: 0.0678 - val_acc: 0.9859
Epoch 6/10
750/750 [==============================] - 334s 445ms/step - loss: 0.0659 - acc: 0.9820 - val_loss: 0.0783 - val_acc: 0.9817
Epoch 7/10
750/750 [==============================] - 328s 437ms/step - loss: 0.0754 - acc: 0.9821 - val_loss: 0.0657 - val_acc: 0.9830
Epoch 8/10
750/750 [==============================] - 325s 433ms/step - loss: 0.0690 - acc: 0.9830 - val_loss: 0.0695 - val_acc: 0.9844
Epoch 9/10
750/750 [==============================] - 325s 434ms/step - loss: 0.0789 - acc: 0.9811 - val_loss: 0.0776 - val_acc: 0.9840
Epoch 10/10
750/750 [==============================] - 328s 438ms/step - loss: 0.0616 - acc: 0.9845 - val_loss: 0.0553 - val_acc: 0.9872
<keras.callbacks.History at 0x7fdb4d9bbc40>
```

[파라미터 정보]

validation_split = 0.2,
epochs=10,
batch_size=64

```
[18] image = X_test[0]
     image.shape

     image = image[tf.newaxis, ...]
     image.shape

     model.predict(image)
```

```
1/1 [==============================] - 0s 78ms/step
array([[4.9027245e-21, 3.2331817e-13, 6.2031306e-15, 5.3327182e-14,
        1.2476906e-15, 5.9107283e-20, 2.2254479e-28, 1.0000000e+00,
        1.5146456e-20, 1.8271668e-14]], dtype=float32)
```

이전 조건에서 학습률을 0.01로 변경하여 진행해 보았다.
Loss 값은 0.0616 accuracy값은 0.9845이다.
학습률 높이니 정확도가 눈에 띄게 감소하고 손실 값이 높아졌다.

MNIST

AlexNet

정규화 방식
Augmentation과
Dropout을 사용한다.

해당 모델에서는
Dropout을 사용하여
아웃풋에 0.5를 곱하는
방식으로 정규화를
진행하였다.

```
from keras.models import Sequential
from keras.layers import Conv2D, AveragePooling2D, Flatten, Dense,Activation,MaxPool2D, BatchNormalization, Dropout
from keras.regularizers import l2
```

```
Using TensorFlow backend.
```

```
# Instantiate an empty sequential model
model = Sequential(name="Alexnet")
# 1st layer (conv + pool + batchnorm)
model.add(Conv2D(filters= 96, kernel_size= (11,11), strides=(4,4), padding='valid', kernel_regularizer=l2(0.0005),
input_shape = (227,227,3)))
model.add(Activation('relu'))  #<---- activation function can be added on its own layer or within the Conv2D function
model.add(MaxPool2D(pool_size=(3,3), strides= (2,2), padding='valid'))
model.add(BatchNormalization())

# 2nd layer (conv + pool + batchnorm)
model.add(Conv2D(filters=256, kernel_size=(5,5), strides=(1,1), padding='same', kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(MaxPool2D(pool_size=(3,3), strides=(2,2), padding='valid'))
model.add(BatchNormalization())

# layer 3 (conv + batchnorm)      <--- note that the authors did not add a POOL layer here
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same', kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# layer 4 (conv + batchnorm)      <--- similar to layer 3
model.add(Conv2D(filters=384, kernel_size=(3,3), strides=(1,1), padding='same', kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())

# layer 5 (conv + batchnorm)
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), padding='same', kernel_regularizer=l2(0.0005)))
model.add(Activation('relu'))
model.add(BatchNormalization())
model.add(MaxPool2D(pool_size=(3,3), strides=(2,2), padding='valid'))

# Flatten the CNN output to feed it with fully connected layers
model.add(Flatten())

# layer 6 (Dense layer + dropout)
model.add(Dense(units = 4096, activation = 'relu'))
model.add(Dropout(0.5))

# layer 7 (Dense layers)
model.add(Dense(units = 4096, activation = 'relu'))
model.add(Dropout(0.5))

# layer 8 (softmax output layer)
model.add(Dense(units = 1000, activation = 'softmax'))

# print the model summary
model.summary()
```

Model: "Alexnet"

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 55, 55, 96)	34944
activation_1 (Activation)	(None, 55, 55, 96)	0
max_pooling2d_1 (MaxPooling2	(None, 27, 27, 96)	0
batch_normalization_1 (Batch	(None, 27, 27, 96)	384
conv2d_2 (Conv2D)	(None, 27, 27, 256)	614656
activation_2 (Activation)	(None, 27, 27, 256)	0
max_pooling2d_2 (MaxPooling2	(None, 13, 13, 256)	0
batch_normalization_2 (Batch	(None, 13, 13, 256)	1024
conv2d_3 (Conv2D)	(None, 13, 13, 384)	885120
activation_3 (Activation)	(None, 13, 13, 384)	0
batch_normalization_3 (Batch	(None, 13, 13, 384)	1536
conv2d_4 (Conv2D)	(None, 13, 13, 384)	1327488
activation_4 (Activation)	(None, 13, 13, 384)	0
batch_normalization_4 (Batch	(None, 13, 13, 384)	1536
conv2d_5 (Conv2D)	(None, 13, 13, 256)	884992
activation_5 (Activation)	(None, 13, 13, 256)	0
batch_normalization_5 (Batch	(None, 13, 13, 256)	1024
max_pooling2d_3 (MaxPooling2	(None, 6, 6, 256)	0
flatten_1 (Flatten)	(None, 9216)	0
dense_1 (Dense)	(None, 4096)	37752832
dropout_1 (Dropout)	(None, 4096)	0
dense_2 (Dense)	(None, 4096)	16781312
dropout_2 (Dropout)	(None, 4096)	0
dense_3 (Dense)	(None, 1000)	4097000

Total params: 62,383,848
Trainable params: 62,381,096
Non-trainable params: 2,752

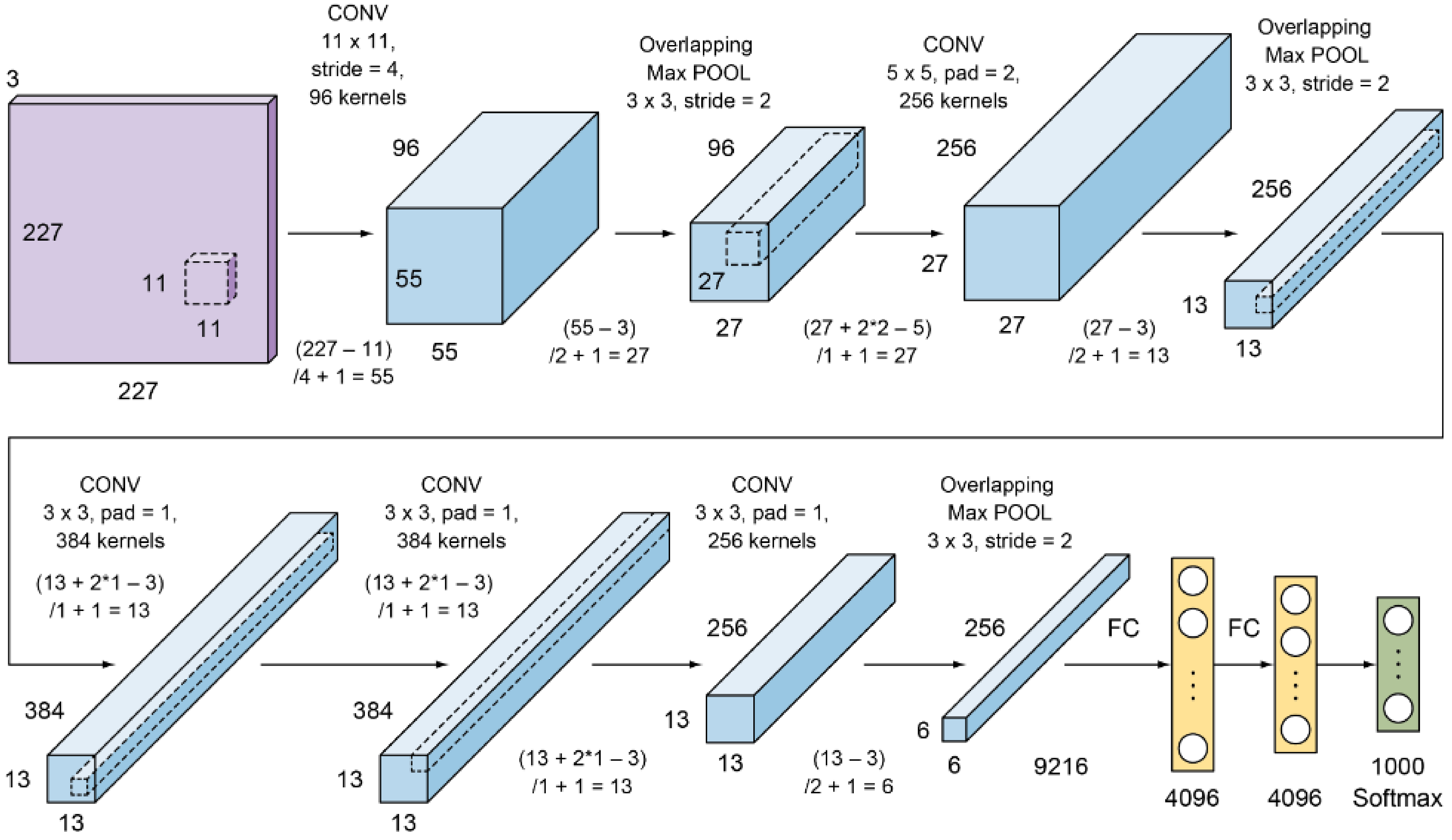
AlexNet은 모델을 두부분으로 나누어 각각 학습시키는 방식으로 진행된다.
원래의 sequential model과 마찬가지로 relu와 softmax를 활용하였다.

MNIST

AlexNet

Deep Learning for Vision Systems Book

AlexNet implementation with Keras



AlexNet model의 구조

MNIST

AlexNet

AlexNet model 만들기

```
[13]
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, ReLU, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential([
    Conv2D(filters=96, input_shape=(227,227,1), kernel_size=(11, 11), strides=(4, 4), activation='relu'),
    MaxPooling2D(pool_size=(2,2), strides=(2,2)),
    # Conv2D(filters=96, input_shape=(28,28,1), kernel_size=3, strides=4, activation='relu'),
    # MaxPooling2D(pool_size=2, strides=2),

    Conv2D(filters=256, kernel_size=(5, 5), strides=(1, 1), activation='relu'),
    MaxPooling2D(pool_size=(2,2), strides=(2,2)),
    # Conv2D(filters=256, kernel_size=5, strides=1, activation='relu'),
    # MaxPooling2D(pool_size=2, strides=2),

    Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1), activation='relu'),
    Conv2D(filters=384, kernel_size=(3, 3), strides=(1, 1), activation='relu'),
    Conv2D(filters=256, kernel_size=(3, 3), strides=(1, 1), activation='relu'),
    MaxPooling2D(pool_size=(2,2), strides=(2,2)),
    # Conv2D(filters=384, kernel_size=3, strides=1, activation='relu'),
    # Conv2D(filters=384, kernel_size=3, strides=1, activation='relu'),
    # Conv2D(filters=384, kernel_size=3, strides=1, activation='relu'),
    # MaxPooling2D(pool_size=2, strides=2),

    Flatten(),
    Dense(4096, activation='relu'),
    Dropout(0.4),

    Dropout(0.4),
    Dropout(0.4),

    Dense(1000, activation='relu'),
    Dropout(0.4),

    Dense(10, activation='softmax')
])

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 55, 55, 96)        11712

max_pooling2d (MaxPooling2D  (None, 27, 27, 96)        0
)

conv2d_1 (Conv2D)            (None, 23, 23, 256)       614656

max_pooling2d_1 (MaxPooling  (None, 11, 11, 256)       0
2D)

conv2d_2 (Conv2D)            (None, 9, 9, 384)         885120

conv2d_3 (Conv2D)            (None, 7, 7, 384)         1327488

conv2d_4 (Conv2D)            (None, 5, 5, 256)         884992

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

flatten (Flatten)            (None, 1024)              0

dense (Dense)                (None, 4096)              4198400

dropout (Dropout)            (None, 4096)              0

dropout_1 (Dropout)          (None, 4096)              0

dropout_2 (Dropout)          (None, 4096)              0

dense_1 (Dense)              (None, 1000)              4097000

dropout_3 (Dropout)          (None, 1000)              0

dense_2 (Dense)              (None, 10)                10010

=================================================================
Total params: 12,029,378
Trainable params: 12,029,378
Non-trainable params: 0
_________________________________________________________________
```

위의 코드를 활용하여 mnist 데이터셋에 모델을 적용해 보았다.
데이터셋과 모델의 배열 차원수가 달라 resize를 활용하여 맞춰주는 작업을 진행하였고
이전 sequential model을 활용할 때보다 시간이 훨씬 오래걸렸다.
이전 모델보다 정확도는 떨어지고, 손실값은 높아지는 결과를 보였다.

MNIST

VGG16

In [1]:

```python
from keras.models import Sequential
from keras.layers import Conv2D, AveragePooling2D, Flatten, Dense, Activation, MaxPool2D, BatchNormalization, Dropout, ZeroPadding2D
```

```
Using TensorFlow backend.
```

VGG-16 (configuration D)

In [2]:

```python
model = Sequential()

# first block
model.add(Conv2D(filters=64, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same',input_shape=(224,224, 3)))
model.add(Conv2D(filters=64, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))

# second block
model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(Conv2D(filters=128, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))

# third block
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(Conv2D(filters=256, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))

# forth block
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))

# fifth block
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(Conv2D(filters=512, kernel_size=(3,3), strides=(1,1), activation='relu', padding='same'))
model.add(MaxPool2D(pool_size=(2,2), strides=(2,2)))

# sixth block (classifier)
model.add(Flatten())
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(4096, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1000, activation='softmax'))

model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 224, 224, 64)      1792
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 224, 224, 64)      36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 112, 112, 64)      0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 112, 112, 128)     73856
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 112, 112, 128)     147584
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 56, 56, 128)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 56, 56, 256)       295168
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 56, 56, 256)       590080
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 56, 56, 256)       590080
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 28, 28, 256)       0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 28, 28, 512)       1180160
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 28, 28, 512)       2359808
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 28, 28, 512)       2359808
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 14, 14, 512)       0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
conv2d_13 (Conv2D)           (None, 14, 14, 512)       2359808
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 7, 7, 512)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 25088)             0
_________________________________________________________________
dense_1 (Dense)              (None, 4096)              102764544
_________________________________________________________________
dropout_1 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_2 (Dense)              (None, 4096)              16781312
_________________________________________________________________
dropout_2 (Dropout)          (None, 4096)              0
_________________________________________________________________
dense_3 (Dense)              (None, 1000)              4097000
=================================================================
Total params: 138,357,544
Trainable params: 138,357,544
Non-trainable params: 0
_________________________________________________________________
```

VGG16 model 코드이다. AlexNet과 마찬가지로 GPU를 활용하여 학습을 진행한다.
Colab으로 진행하던 도중 메모리가 다운되는 문제가 발생하였다.
너무 딥한 모델인 것에 비해 mnist는 굉장히 간단한 데이터셋이라 적합하지 않다는 결과를 도출해냈다.

MNIST

VGG16

```
Epoch 1/10
600/600 [==============================] - 69s 110ms/step - loss: 1.8400 - accuracy: 0.5679 - val_loss: 0.1082 - val_accuracy: 0.9654
Epoch 2/10
600/600 [==============================] - 67s 112ms/step - loss: 0.1255 - accuracy: 0.9633 - val_loss: 0.0888 - val_accuracy: 0.9746
Epoch 3/10
600/600 [==============================] - 65s 109ms/step - loss: 0.0870 - accuracy: 0.9753 - val_loss: 0.1271 - val_accuracy: 0.9636
Epoch 4/10
600/600 [==============================] - 67s 112ms/step - loss: 0.0817 - accuracy: 0.9783 - val_loss: 0.1091 - val_accuracy: 0.9724
Epoch 5/10
600/600 [==============================] - 65s 109ms/step - loss: 0.1070 - accuracy: 0.9735 - val_loss: 0.0840 - val_accuracy: 0.9800
Epoch 6/10
600/600 [==============================] - 67s 111ms/step - loss: 20100694016.0000 - accuracy: 0.4631 - val_loss: 2.3010 - val_accuracy: 0.1135
Epoch 7/10
600/600 [==============================] - 66s 109ms/step - loss: 32190714.0000 - accuracy: 0.1123 - val_loss: 2.3010 - val_accuracy: 0.1135
<keras.callbacks.History object at 0x7f200a38a5d0>
313/313 [==============================] - 7s 20ms/step - loss: 2.3010 - accuracy: 0.1135
[2.301023006439209, 0.11349999904632568]
```

Epoch5 까지는 높은 성능으로 학습되다가 갑자기 loss값이 올라가면서 정확도가 0.11로 떨어지는 문제가 발생하였다.

모델을 여러 번 돌려보았지만 학습이 제대로 이루어지지 않는 모습을 보였다.
-> gradient vanishing등의 문제가 발생한 것으로 예상된다.

층이 많고 깊은 모델이라고 해서 무조건 좋은 것이 아니라는 결과를 도출해낼 수 있었다.

* Gradient vanishing이란?
깊은 인공 신경망을 학습하다보면 역전파 과정에서 입력층으로 갈 수록 기울기가 점차적으로 작아지는 현상이 발생할 수 있다.
입력층에 가까운 층들에서 가중치들이 업데이트가 제대로 되지 않으면 결국 최적의 모델을 찾을 수 없게 된다.

Thank You