PART 3

합성곱 신경망(CNN)

1장. 이론편

2장. 코딩편

딥러닝 & 강화학습 담당
이재화 강사

이 장에서 다룰 내용

1. Min-Max Normalization 데이터 정규화 작업
2. 다층 퍼셉트론에서의 이미지 처리
3. 첫 번째 실험 – 합성곱 계층만 활용
4. 두 번째 실험 – 합성곱, 풀링, 드랍아웃 활용
5. 세 번째 실험 - VGGNet
6. 네 번째 실험 – VGGNet + 이미지 보강
7. 최종 결론

Part 3. 합성곱 모델

1.1 Min-Max Normalization 데이터 정규화 작업

- 텐서플로우 2.0 불러오기

```
import tensorflow as tf
print(tf.__version__)
```

- 오늘 사용할 데이터 불러오기

fashion_mnist - 의류 이미지 / 10개의 범주 / 단일 채널 / 28 * 28

```
fashion_mnist = tf.keras.datasets.fashion_mnist
(train_X, train_Y), (test_X,test_Y) = fashion_mnist.load_data()
```

- 데이터의 수 확인하기 (학습 데이터 수 6만개 / 테스트 데이터 수 1만개)

```
print(len(train_X), len(test_X))
```

```
60000 10000
```

1.1 Min-Max Normalization 데이터 정규화 작업

- 이미지 확인

cmap = ' ' 을 통해 이미지의 출력 색상 선택

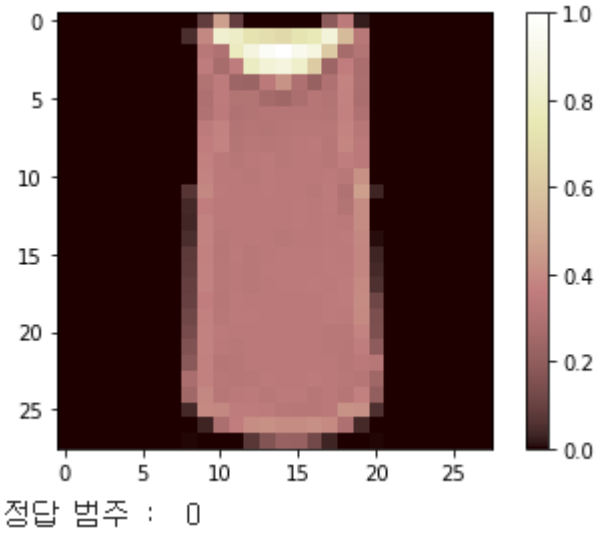
plt.colorbar() : 우측에 색상값의 정보를 바(bar)형태로 출력

색상값은 0 ~ 255의 값을 가지고 있음을 확인

print(train_Y[2]) 로 확인결과 이 이미지는 **0번 티셔츠/상의**


```
import matplotlib.pyplot as plt
plt.imshow(train_X[2], cmap = 'pink')
plt.colorbar()
plt.show()

# 정답 범주 확인
print("정답 범주 : ",train_Y[2])
```



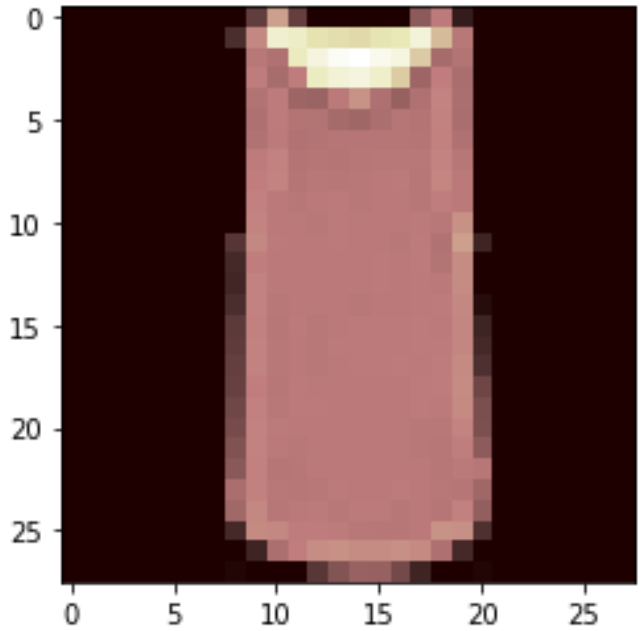
Part 3. 합성곱 모델

1.1 Min-Max Normalization 데이터 정규화 작업

- 데이터 정규화 이전의 [이미지 픽셀 행렬]

```
print(train_X[2])
```

```
[[  0   0   0   0   0   0   0   0   0  22 118  24   0   0   0   0   0  48  88   5   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  12 100 212 205 185 179 173 186 193 221 142  85   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  85  76 199 225 248 255 238 226 157  68  80   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  91  69  91 201 218 225 209 158  61  93  72   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  79  89  61  59  87 108  75  56  76  97  73   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  75  89  80  80  67  63  73  83  80  96  72   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  77  88  77  80  83  83  83  83  81  95  76   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  89  96  80  83  81  84  85  85  85  97  84   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  93  97  81  85  84  85  87  88  84  99  87   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  95  87  84  87  88  85  87  87  84  92  87   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  97  87  87  85  88  87  87  87  88  85 107   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  17 100  88  87  87  88  87  87  85  89  77 118   8   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  10  93  87  87  87  87  87  88  87  89  80 103   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   9  96  87  87  87  87  87  88  87  88  87 103   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  12  96  85  87  87  87  85  87  87  88  89 100   2   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  20  95  84  88  85  87  88  88  88  89  88  99   8   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  21  96  85  87  85  88  88  88  88  89  89  99  10   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  24  96  85  87  85  87  88  88  89  88  91 102  14   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  25  93  84  88  87  87  87  87  87  89  91 103  29   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  30  95  85  88  88  87  87  87  87  89  88 102  37   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  34  96  88  87  87  87  87  87  87  85  85  97  38   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  40  96  87  85  87  87  87  87  87  85  84  92  49   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  46  95  83  84  87  87  87  87  87  87  84  87  84   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  72  95  85  84  85  88  87  87  89  87  85  83  63   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  64 100  84  87  88  85  88  88  84  87  83  95  53   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  10 102 100  91  91  89  85  84  84  87 108 106  14   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   8  73  93 104 107 103 103 106 102  75  10   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   1   0   0   0  18  42  57  56  32   8   0   0   1   0   0   0   0   0   0   0]]
```



3장. 합성곱 신경망

1.1 Min-Max Normalization 데이터 정규화 작업

최소-최대 정규화는 데이터를 정규화하는 가장 일반적인 방법.

모든 feature에 대해 각각의 최소값 0, 최대값 1

그리고 0 과 1 사이의 값으로 변환하는 거다.

- $\frac{(X - MIN)}{(MAX - MIN)} == \frac{X - 0}{(255 - 0)}$

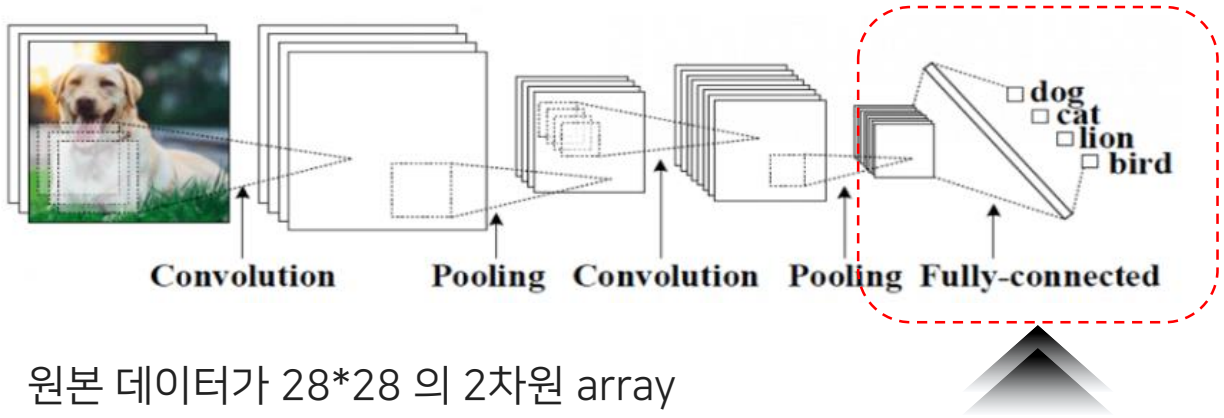
(※ 바로 위의 작업을 통해 이미지의 최소값은 0, 최대값은 255 임을 확인)

```
train_X = train_X / 255.0
test_X = test_X / 255.0

print(train_X[2])
```

```
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.00392157 0.         0.         0.05098039 0.28627451 0.
 0.         0.00392157 0.01568627 0.         0.         0.
 0.         0.00392157 0.00392157 0.        ]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.01176471 0.         0.14117647 0.53333333 0.49803922 0.24313725
 0.21176471 0.         0.         0.         0.00392157 0.01176471
 0.01568627 0.         0.         0.01176471]
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.02352941 0.         0.4        0.8        0.69019608 0.5254902
 0.56470588 0.48235294 0.09019608 0.         0.         0.
 0.         0.04705882 0.03921569 0.        ]
[0.         0.         0.         0.         0.         0.
```

1.2 다층 퍼셉트론에서의 이미지 처리



원본 데이터가 28*28 의 2차원 array

Flatten()을 활용하면 다차원 이미지를 1차원으로 평평하게 바꿔주는 단순 레이어

Flatten()은 다차원 이미지를 1차원으로 평평하게 바꿔주는 단순 레이어

- input_shape : 원본 데이터의 크기를 입력

Dense 레이어는 완전연결 레이어

마지막 레이어의 units값 10. 즉 정답 범주의 수 와 동일.

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape = (28,28)),
    tf.keras.layers.Dense(units=128,activation='relu'),
    tf.keras.layers.Dense(units=10, activation='softmax')
])
```

1.2 다층 퍼셉트론에서의 이미지 처리

컴파일 단계

- Adam의 기본값 : lr = 0.0001
- spares_categorical_crossentropy : 희소행렬을 나타내는 데이터를 전처리 없이 정답행렬로 사용가능.
    - 희소행렬 : 대부분이 값이 0인 행렬
    - ex) [0,0,0,0,0,0,1], [0,0,0,1,0,0,0,0]

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

```
model.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_3 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               100480
_________________________________________________________________
dense_7 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

총 파라미터 수

1.2 다층 퍼셉트론에서의 이미지 처리

- 신경망 학습 단계

```
history = model.fit(train_X,train_Y,
                    epochs = 500,
                    validation_split = 0.25,
                    callbacks = [tf.keras.callbacks.EarlyStopping
(patience=5,monitor='val_loss')])
```

```
Epoch 1/500
1407/1407 [==============================] - 3s 2ms/step - loss: 0.5195 - accuracy: 0.8185 - val_loss: 0.4345 - val_accuracy: 0.8454
Epoch 2/500
1407/1407 [==============================] - 3s 2ms/step - loss: 0.3838 - accuracy: 0.8611 - val_loss: 0.3912 - val_accuracy: 0.8569
Epoch 3/500
1407/1407 [==============================] - 3s 2ms/step - loss: 0.3493 - accuracy: 0.8722 - val_loss: 0.3579 - val_accuracy: 0.8721
Epoch 4/500
1407/1407 [==============================] - 3s 2ms/step - loss: 0.3223 - accuracy: 0.8828 - val_loss: 0.3379 - val_accuracy: 0.8811
Epoch 5/500
1407/1407 [==============================] - 3s 2ms/step - loss: 0.3025 - accuracy: 0.8876 - val_loss: 0.3436 - val_accuracy: 0.8780
```

⋮

1.2 다층 퍼셉트론에서의 이미지 처리

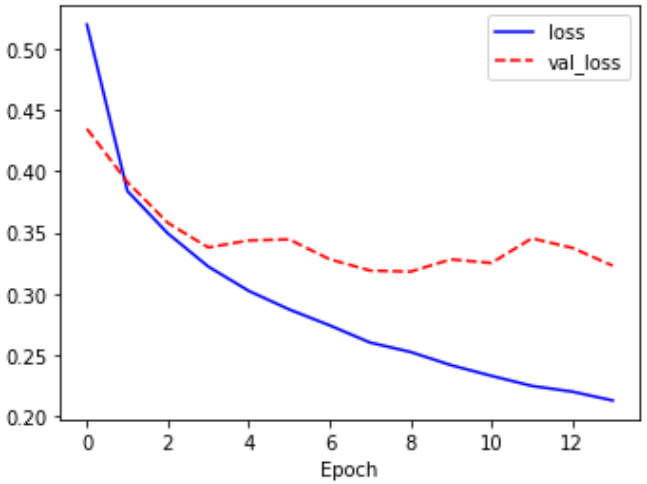
- 정확도 및 손실값에 대한 결과 시각화
  - 검증데이터와 학습데이터 비교
  - 과적합 혹은 과소적합 확인

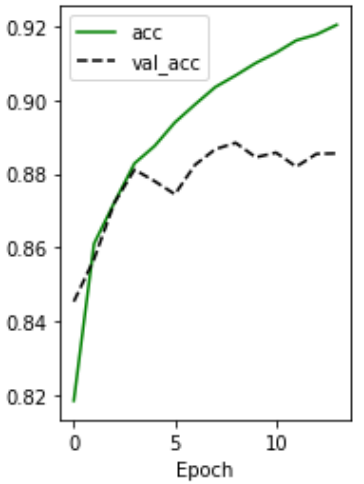
```
import matplotlib.pyplot as plt
plt.figure(figsize = (12,4))

plt.subplot(1,2,1)
plt.plot(history.history['loss'],'b-',label = 'loss')
plt.plot(history.history['val_loss'],'r--',label = 'val_loss')
plt.xlabel('Epoch')
plt.legend()
plt.show()

plt.subplot(1,2,2)
plt.plot(history.history['accuracy'],'g-',label = 'acc')
plt.plot(history.history['val_accuracy'],'k--',label = 'val_acc')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```

과적합 포착





1.2 다층 퍼셉트론에서의 이미지 처리

최종 평가

- evaluate() : 최종 정확도 및 손실값 확인

```
model.evaluate(test_X,test_Y)
```

313/313 [==============================] - 1s 3ms/step - loss: 0.5899 - accuracy: 0.8197

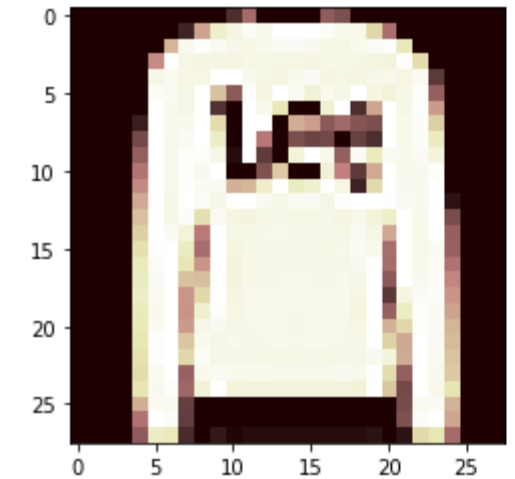
1.2 다층 퍼셉트론에서의 이미지 처리

예측

- predict() : 학습된 신경망을 활용하여 예측을 수행

```
#예측 수행
pred_X = model.predict(test_X[[1]])
#예측 이미지 확인
plt.imshow(test_X[1], cmap = 'pink')
plt.show()

#예측 결과 확인
print(pred_X+1)
print("예측 수행 범주 :",test_Y[1])
```



1.3 첫 번째 실험 – 합성곱 계층만 활용

- 실험 1. 합성곱 계층만 사용한 경우
- 실험 2. 풀링 계층과 정규화 기법인 드랍아웃을 함께 사용한 경우
- 실험 3. VGGNet 스타일 구축
- 실험 4. 이미지 보강
- 합성곱 연산을 진행하기 위한 4차원 (미니배치 데이터, 입력 이미지 행, 입력 이미지 열, 입력 채널 수)
- reshape() : 기존 3차원 에서 채널이 추가된 4차원 형태로 차원 변경

```
print(train_X.shape, test_X.shape)
rain_X = train_X.reshape(-1, 28, 28, 1)
test_X = test_X.reshape(-1, 28, 28, 1)
print(train_X.shape,test_X.shape)
```

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

#4차원 정보

4차원 입력 - [미니배치 크기, 입력 이미지 행 수, 입력 이미지 열 수, 입력 채널 수]
4차원 커널 - [커널 행 수, 커널 열 수, 입력 채널 수, 출력 채널 수]
4차원 출력 - [미니배치 크기, 출력 이미지 행 수, 출력 이미지 열 수, 출력 채널 수]

✓ 입력 형태 $[mb, xh, xw, xchn]$

✓ 커널 형태 $[kh, kw, xchn, ychn]$

✓ 출력 형태 $[mb, yh, yw, ychn]$

1.3 첫 번째 실험 – 합성곱 계층만 활용

- 분류 대상 이미지 및 범주 확인

plt.subplot(4,4,i+1) : (행,열, 순서)

plt.imshow(train_X[i].reshape(28,28), cmap='pink')

: reshape(28,28)을 통해 기존의 4차원에서 다시 2차원으로 변경하여 이미지 출력

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10,10))

for i in range(16):
  plt.subplot(4,4,i+1)
  plt.imshow(train_X[i].reshape(28,28), cmap='pink')

plt.show()
print(train_Y[:16])
```

1.3 첫 번째 실험 – 합성곱 계층만 활용

```
model= tf.keras.Sequential([
    tf.keras.layers.Conv2D(input_shape=(28,28,1),kernel_size=(3,3),
                           filters=16, strides = (1,1),padding = 'valid'),
    tf.keras.layers.Conv2D(kernel_size=(3,3),filters=32),
    tf.keras.layers.Conv2D(kernel_size=(3,3),filters=64),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation = 'relu'),
    tf.keras.layers.Dense(units=10, activation = 'softmax')
])
```

 # 단순 합성곱 계층만 활용!

실험 1. 합성곱 계층만 사용

- 풀링 레이어 없이, 단순 합성곱 계층만 활용
- tf.keras.layers.Conv2D()
    - input_shape : 입력 이미지의 크기 및 채널
    - kernel_size : 커널 사이즈
    - filters : 몇 개의 필터를 생성할 것 인지 (※ 점차 증가하면서 쌓아주셔야 합니다.)
    - strides : 건너뛰기 (기본값 (1,1))
    - padding : 패딩 방식(기본값 'valid')
- tf.keras.layers.Flatten() : 다차원 레이어를 1차원으로 펼쳐준다.

1.3 첫 번째 실험 – 합성곱 계층만 활용

- 컴파일 단계

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

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 16)        160
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 24, 24, 32)        4640
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 22, 22, 64)        18496
_________________________________________________________________
flatten (Flatten)            (None, 30976)             0
_________________________________________________________________
dense (Dense)                (None, 128)               3965056
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 3,989,642
Trainable params: 3,989,642
Non-trainable params: 0
_________________________________________________________________
```

'풀링계층'이 없기에

약 400만개의 파라미터 생성

(파라미터가 너무 많다.)

1.3 첫 번째 실험 – 합성곱 계층만 활용

- 신경망 학습 단계

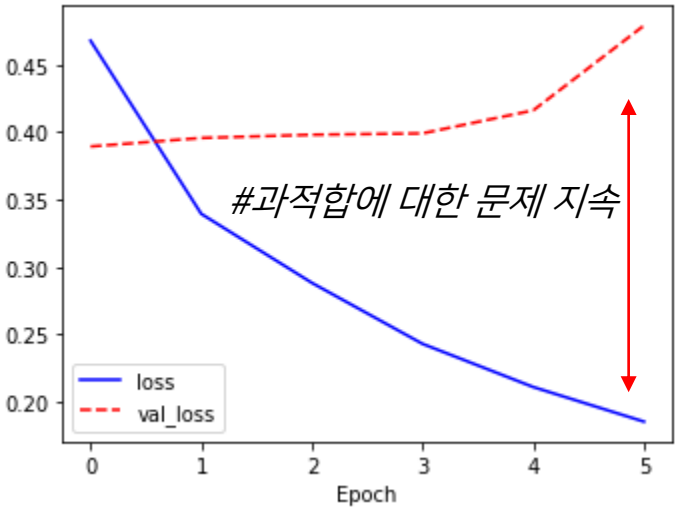
```
history = model.fit(train_X,train_Y,
                    epochs=500,
                    validation_split=0.25,
                    callbacks = [tf.keras.callbacks.EarlyStopping
                                 (patience = 5, monitor = 'val_loss')])
```

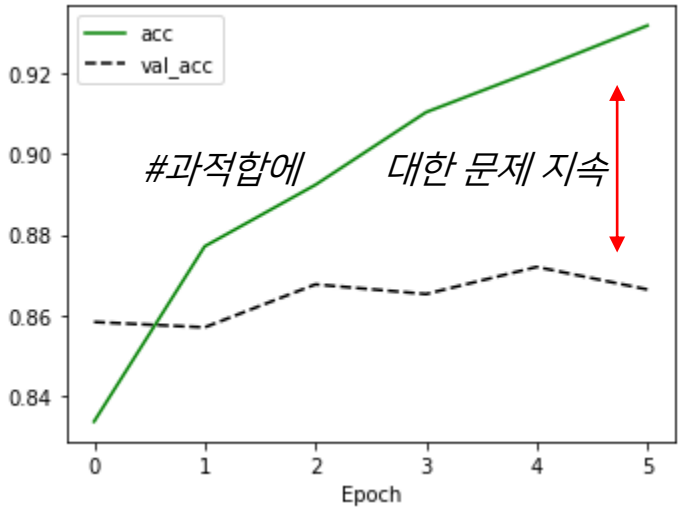
```
Epoch 1/500
1407/1407 [==============================] - 7s 5ms/step - loss: 0.4677 - accuracy: 0.8336 - val_loss: 0.3893 - val_accuracy: 0.8583
Epoch 2/500
1407/1407 [==============================] - 7s 5ms/step - loss: 0.3394 - accuracy: 0.8771 - val_loss: 0.3956 - val_accuracy: 0.8570
Epoch 3/500
1407/1407 [==============================] - 7s 5ms/step - loss: 0.2882 - accuracy: 0.8924 - val_loss: 0.3980 - val_accuracy: 0.8677
Epoch 4/500
1407/1407 [==============================] - 7s 5ms/step - loss: 0.2429 - accuracy: 0.9104 - val_loss: 0.3990 - val_accuracy: 0.8653
Epoch 5/500
1407/1407 [==============================] - 7s 5ms/step - loss: 0.2109 - accuracy: 0.9210 - val_loss: 0.4161 - val_accuracy: 0.8720
Epoch 6/500
1407/1407 [==============================] - 7s 5ms/step - loss: 0.1853 - accuracy: 0.9319 - val_loss: 0.4790 - val_accuracy: 0.8664
```

⋮

1.3 첫 번째 실험 – 합성곱 계층만 활용

- 정확도 및 손실값에 대한 결과 시각화






```
plt.figure(figsize = (12,4))

plt.subplot(1,2,1)
plt.plot(history.history['loss'],'b-',label = 'loss')
plt.plot(history.history['val_loss'],'r--',label = 'val_loss')
plt.xlabel('Epoch')
plt.legend()
plt.subplot(1,2,2)
plt.plot(history.history['accuracy'],'g-',label = 'acc')
plt.plot(history.history['val_accuracy'],'k--',label = 'val_acc')
plt.xlabel('Epoch')
plt.legend()

plt.show()
```

1.3 첫 번째 실험 – 합성곱 계층만 활용

```
model.evaluate(test_X,test_Y)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.5002 - accuracy: 0.8683
```

- 최종 평가

• 다층 퍼셉트론의 결과 loss: 0.5899 - accuracy: 0.8197

• 합성곱 계층만 사용한 결과 loss: 0.5002 - accuracy: 0.8683

 # 하이퍼 파라미터는 동일

 # 정확도 및 손실값이 어느정도 향상

 # 퍼셉트론 수가 너무 많다보니 학습에 많은 시간 소요

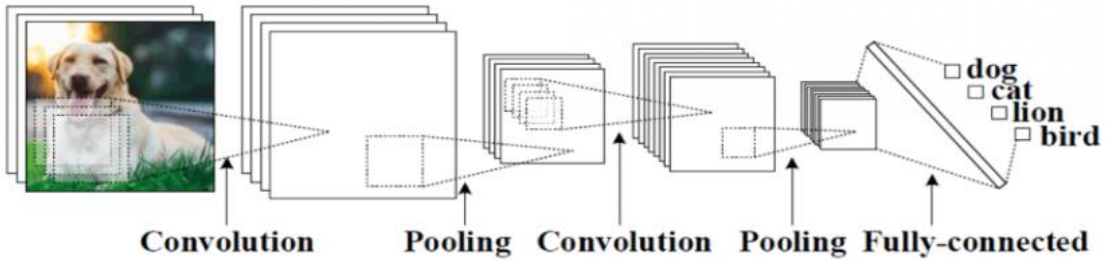
 # 과적합 현상 발생

1.4 두 번째 실험 – 합성곱, 풀링, 드랍아웃 활용

```
model= tf.keras.Sequential([
    tf.keras.layers.Conv2D(input_shape=(28,28,1),kernel_size=(3,3),filters=32),
    tf.keras.layers.MaxPool2D(strides = (2,2),pool_size=(2,2)),
    tf.keras.layers.Conv2D(kernel_size=(3,3),filters = 64),
    tf.keras.layers.AvgPool2D(strides = (2,2)),
    tf.keras.layers.Conv2D(kernel_size=(3,3),filters=128),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=128, activation = 'relu'),
    tf.keras.layers.Dropout(rate = 0.3),
    tf.keras.layers.Dense(units=10, activation = 'softmax')
])
```

 # 합성곱 계층과 풀링 계층이 번갈아 가며 등장

- 최대치 풀링 : tf.keras.layers.MaxPool2D()
  - strides : 건너뛰기
  - pool size : 풀링 사이즈 설정 - 기본값 : (2,2)
- 평균치 풀링 : tf.keras.layers.AveragePool2D()
- 드랍아웃 계층 생성 : tf.keras.layers.Dropout()
  - rate : 이전 계층에서 제외할 뉴런의 비율 설정



1.4 두 번째 실험 – 합성곱, 풀링, 드랍아웃 활용

- 컴파일 단계

```
model.compile(optimizer = tf.keras.optimizers.Adam(),
              loss = 'sparse_categorical_crossentropy',
              metrics = ['accuracy'])
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_6 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 11, 11, 64)        18496
_________________________________________________________________
average_pooling2d_1 (Average (None, 5, 5, 64)          0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 3, 3, 128)         73856
_________________________________________________________________
flatten_2 (Flatten)          (None, 1152)              0
_________________________________________________________________
dense_4 (Dense)              (None, 128)               147584
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_5 (Dense)              (None, 10)                1290
=================================================================
Total params: 241,546
Trainable params: 241,546
Non-trainable params: 0
_________________________________________________________________
```

 # 이전 실험보다 파라미터가 확연하게 감소.

 # 풀링계층의 효과

1.4 두 번째 실험 – 합성곱, 풀링, 드랍아웃 활용

- 신경망 학습 단계

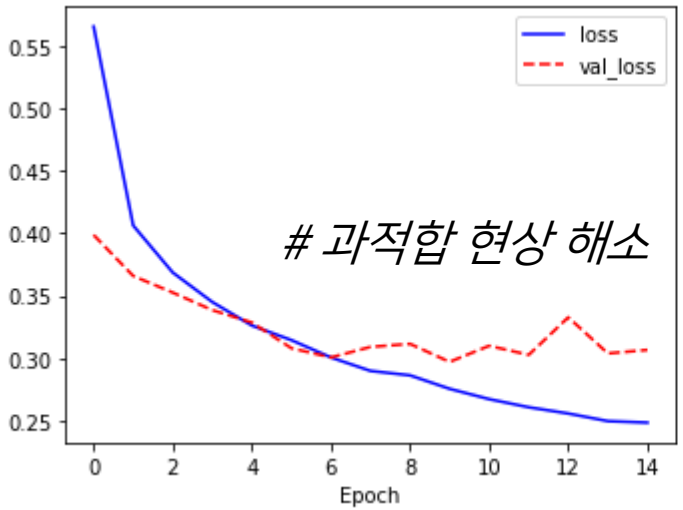
```
history = model.fit(train_X,train_Y,
                    epochs = 500,
                    validation_split=0.25,
                    callbacks = [tf.keras.callbacks.EarlyStopping
                                 (patience = 5, monitor = 'val_loss')])
```

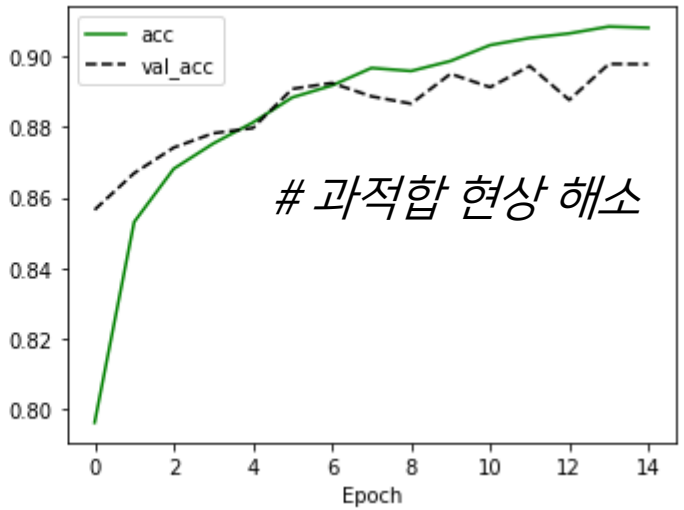
```
Epoch 1/500
1407/1407 [==============================] - 6s 4ms/step - loss: 0.5654 - accuracy: 0.7961 - val_loss: 0.3987 - val_accuracy: 0.8565
Epoch 2/500
1407/1407 [==============================] - 5s 4ms/step - loss: 0.4063 - accuracy: 0.8531 - val_loss: 0.3660 - val_accuracy: 0.8669
Epoch 3/500
1407/1407 [==============================] - 5s 4ms/step - loss: 0.3687 - accuracy: 0.8682 - val_loss: 0.3526 - val_accuracy: 0.8742
Epoch 4/500
1407/1407 [==============================] - 5s 4ms/step - loss: 0.3451 - accuracy: 0.8754 - val_loss: 0.3386 - val_accuracy: 0.8783
Epoch 5/500
1407/1407 [==============================] - 5s 4ms/step - loss: 0.3263 - accuracy: 0.8813 - val_loss: 0.3287 - val_accuracy: 0.8797
```

⋮

1.4 두 번째 실험 – 합성곱, 풀링, 드랍아웃 활용

- 정확도 및 손실값에 대한 결과 시각화






```
plt.figure(figsize = (12,4))

plt.subplot(1,2,1)
plt.plot(history.history['loss'],'b-',label = 'loss')
plt.plot(history.history['val_loss'],'r--',label = 'val_loss')
plt.xlabel('Epoch')
plt.legend()
plt.subplot(1,2,2)
plt.plot(history.history['accuracy'],'g-',label = 'acc')
plt.plot(history.history['val_accuracy'],'k--',label = 'val_acc')
plt.xlabel('Epoch')
plt.legend()

plt.show()
```

1.4 두 번째 실험 – 합성곱, 풀링, 드랍아웃 활용

```
model.evaluate(test_X,test_Y)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.3248 - accuracy: 0.8921
```

최종 평가

- 다층 퍼셉트론 → loss: 0.5899 - accuracy: 0.8197
- 합성곱 계층만 사용 → loss: 0.5002 - accuracy: 0.868
- **합성곱, 풀링, 드롭아웃 사용 → loss: 0.3248 - accuracy: 0.8921**

 # 하이퍼 파라미터는 동일

 # 정확도 및 손실값 어느정도 향상

 # 풀링계층을 활용하여 퍼셉트론 수 감축

 # 학습속도 개선

 # 과적합 현상 발생을 억제하기 위해
정규화 기법인 드랍아웃 적용

 # 일정부분 과적합 현상 해소

 # 최대치 풀링과 평균치 풀링 활용

1.5 세 번째 실험 – VGGNet 스타일 적용

Style Transfer 논문에서도 VGGNet활용

- VGGNet 스타일 **일부** 적용
  - 합성곱 계층 2개 적용 / 풀링 레이어 삽입
  - 합성곱 계층 2개 적용 / 풀링 레이어 삽입
  - 완전연결계층 적용
  - 완전연결계층 적용
  - 완전연결계층 적용

VGGNet 참고 문헌 :

https://arxiv.org/pdf/1709.01921.pdf

Style Transfer 참고 문헌 :

https://arxiv.org/pdf/1508.06576.pdf

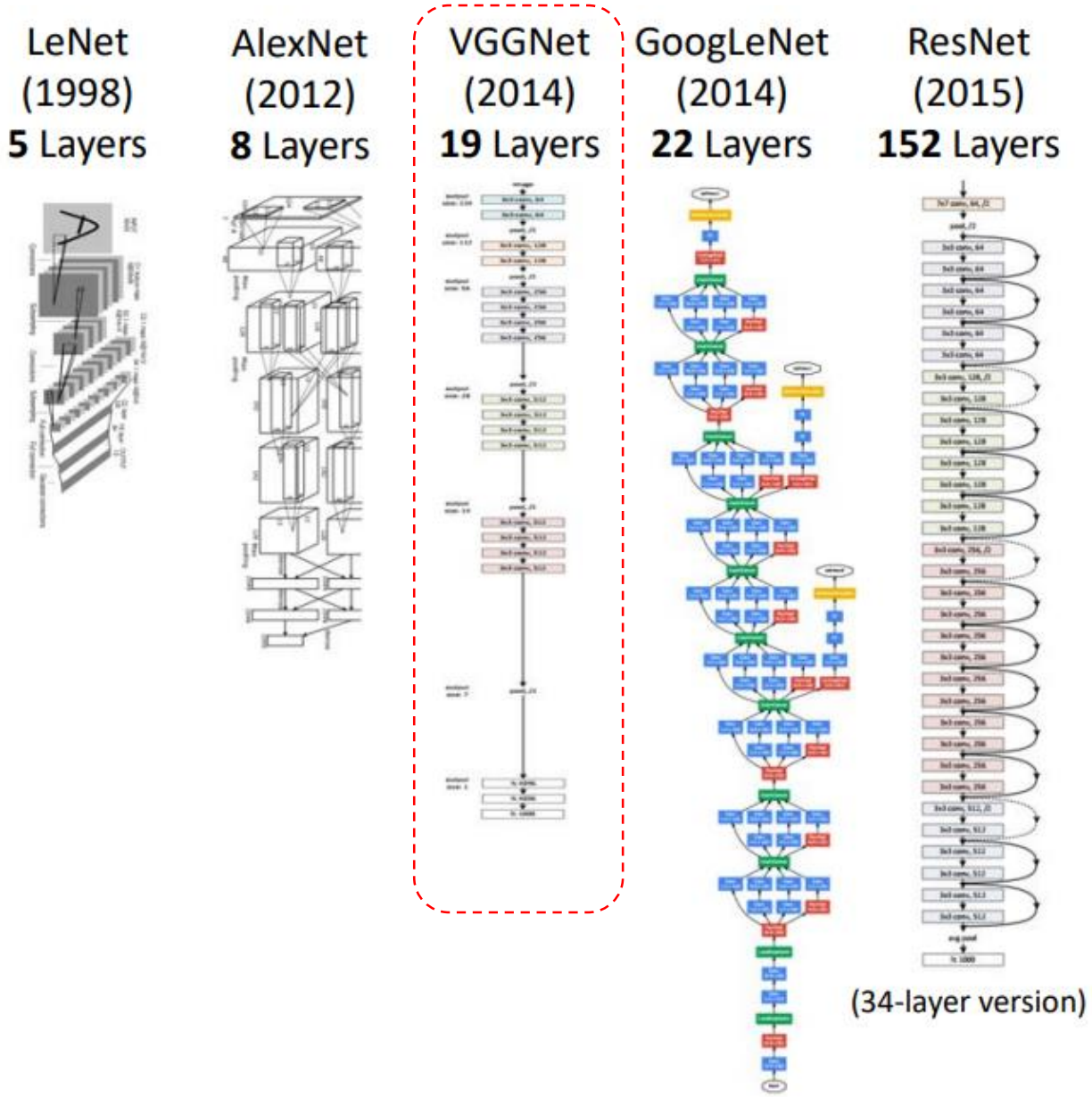


Figure 1. Progression towards deeper neural network structures in recent years (see, *e.g.*, [6], [7], [8], [9], [10]).

1.5 세 번째 실험 – VGGNet 스타일 적용

```
model= tf.keras.Sequential([
    tf.keras.layers.Conv2D(input_shape=(28,28,1),kernel_size=(3,3),filters=32,
                           padding='same',activation='relu'),
    tf.keras.layers.Conv2D(input_shape=(28,28,1),kernel_size=(3,3),filters=64,
                           padding='same',activation='relu'),
    tf.keras.layers.MaxPool2D(strides = (2,2)),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Conv2D(kernel_size=(3,3),filters = 128, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(kernel_size=(3,3),filters = 256, padding = 'valid', activation='relu'),
    tf.keras.layers.MaxPool2D(strides = (2,2)),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=512, activation = 'relu'),
    tf.keras.layers.Dropout(rate = 0.5),
    tf.keras.layers.Dense(units=256, activation = 'relu'),
    tf.keras.layers.Dropout(rate = 0.5),
    tf.keras.layers.Dense(units=10, activation = 'softmax')
])
```

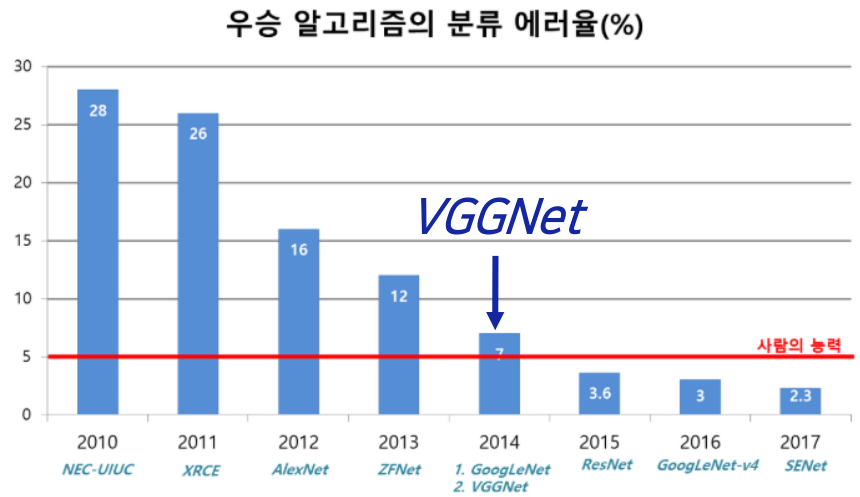


그림1. ILSVRC 대회 역대 우승 알고리즘들과 인식 에러율.

1.5 세 번째 실험 – VGGNet 스타일 적용

- 컴파일 단계

```
model.compile(optimizer = tf.keras.optimizers.Adam(),
              loss = 'sparse_categorical_crossentropy',
              metrics = ['accuracy'])
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 28, 28, 64)        18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 64)        0
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 128)       73856
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 12, 12, 256)       295168
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 256)         0
_________________________________________________________________
dropout_1 (Dropout)          (None, 6, 6, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 9216)              0
_________________________________________________________________
dense (Dense)                (None, 512)               4719104
_________________________________________________________________
dropout_2 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               131328
_________________________________________________________________
dropout_3 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 5,240,842
Trainable params: 5,240,842
Non-trainable params: 0
_________________________________________________________________
```

 # 풀링계층을 거쳤음에도 불구하고,
레이어가 깊게 쌓인결과 파라미터 다수 생성.

 # 레이어가 깊게 쌓인만큼 높은 성능 기대

 # 컴퓨팅 파워 중요

1.5 세 번째 실험 – VGGNet 스타일 적용

- 신경망 학습 단계

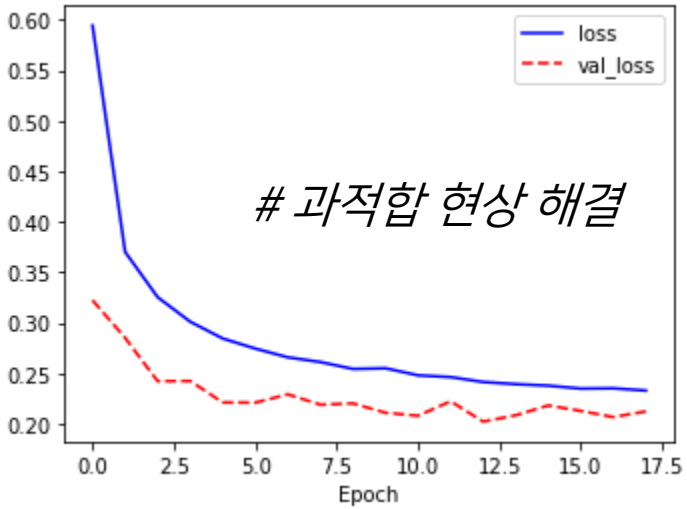
```
history = model.fit(train_X,train_Y,
                    epochs=500,
                    validation_split=0.25,
                    callbacks = [tf.keras.callbacks.EarlyStopping
                                 (patience = 5, monitor = 'val_loss')])
```

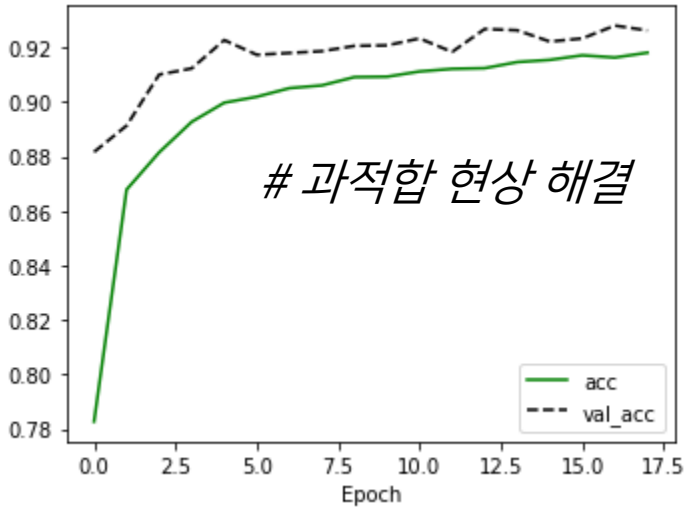
```
Epoch 1/500
1407/1407 [==============================] - 11s 8ms/step - loss: 0.5945 - accuracy: 0.7826 - val_loss: 0.3222 - val_accuracy: 0.8816
Epoch 2/500
1407/1407 [==============================] - 11s 8ms/step - loss: 0.3701 - accuracy: 0.8678 - val_loss: 0.2849 - val_accuracy: 0.8913
Epoch 3/500
1407/1407 [==============================] - 11s 8ms/step - loss: 0.3252 - accuracy: 0.8814 - val_loss: 0.2419 - val_accuracy: 0.9100
Epoch 4/500
1407/1407 [==============================] - 11s 8ms/step - loss: 0.3009 - accuracy: 0.8927 - val_loss: 0.2421 - val_accuracy: 0.9123
Epoch 5/500
1407/1407 [==============================] - 11s 8ms/step - loss: 0.2843 - accuracy: 0.8997 - val_loss: 0.2209 - val_accuracy: 0.9227
```

⋮

1.5 세 번째 실험 – VGGNet 스타일 적용

- 정확도 및 손실값에 대한 결과 시각화






```
plt.figure(figsize = (12,4))

plt.subplot(1,2,1)
plt.plot(history.history['loss'],'b-',label = 'loss')
plt.plot(history.history['val_loss'],'r--',label = 'val_loss')
plt.xlabel('Epoch')
plt.legend()
plt.subplot(1,2,2)
plt.plot(history.history['accuracy'],'g-',label = 'acc')
plt.plot(history.history['val_accuracy'],'k--',label = 'val_acc')
plt.xlabel('Epoch')
plt.legend()

plt.show()
```

1.5 세 번째 실험 – VGGNet 스타일 적용

```
model.evaluate(test_X,test_Y)
```

```
313/313 [==============================] - 1s 4ms/step - loss: 0.2306 - accuracy: 0.9175
```

최종 평가

- 다층 퍼셉트론 → loss: 0.5899 - accuracy: 0.8197
- 합성곱 계층만 사용 → loss: 0.5002 - accuracy: 0.868
- 합성곱, 풀링, 드롭아웃 사용 → loss: 0.3248 - accuracy: 0.8921
- VGGNet스타일 기법 → loss: 0.2306 - accuracy: 0.9175

 # 하이퍼 파라미터는 동일

 # 정확도 및 손실값 어느정도 향상

 # VGGNet 스타일 적용

 # 깊은 레이어만큼 다수의 파라미터 생성

 # 과적합 현상 발생을 억제하기 위해

정규화 기법인 드랍아웃 적용

 # 과적합 현상 해소

 # 90% 이상의 정확도 달성

1.6 네 번째 실험 – VGGNet + 이미지 보강

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator
import numpy as np

image_generator = ImageDataGenerator(
    rotation_range = 10,
    zoom_range = 0.10,
    shear_range = 0.5,
    width_shift_range = 0.10,
    height_shift_range = 0.10,
    horizontal_flip = True,
    vertical_flip = False)
```

이미지 추가 생성 연습

- rotation_range : 이미지 회전값
- zoom_range : 이미지 일부 확대
- shear_range : 이미지 기울기
- width_shift_range : 좌우 이동
- height_shift_range : 상하 이동
- horizontal_flip : 이미지 가로 뒤집기
- vertical_filp : 이미지 세로 뒤집기

1.6 네 번째 실험 – VGGNet + 이미지 보강

```
augment_size = 100 #한번에 생성할 이미지의 수
                                                    #np.tile() : A를 reps에 정해진 형식만큼 반복
x_augment = image_generator.flow(x = np.tile(A = train_X[0].reshape(28*28),
                                             reps = 100).reshape(-1,28,28,1),
                                 y = np.zeros(augment_size), #라벨값은 딱히 줄 필요 없기에 np.zeros() 할당
                                 batch_size = augment_size,  #배치 사이즈
                                 shuffle = False).next()[0]  #next()로서 실제 값을 꺼냄.
```

```
print(x_augmented.shape)
print(x_augmented_1.shape)
```

```
(30000, 28, 28, 1)
(30000,)
```

```
x_augmented_1 = image_generator.flow(x = x_augmented,
                                     y = np.zeros(augment_size),
                                     batch_size = augment_size,
                                     shuffle = False).next()[1]
```

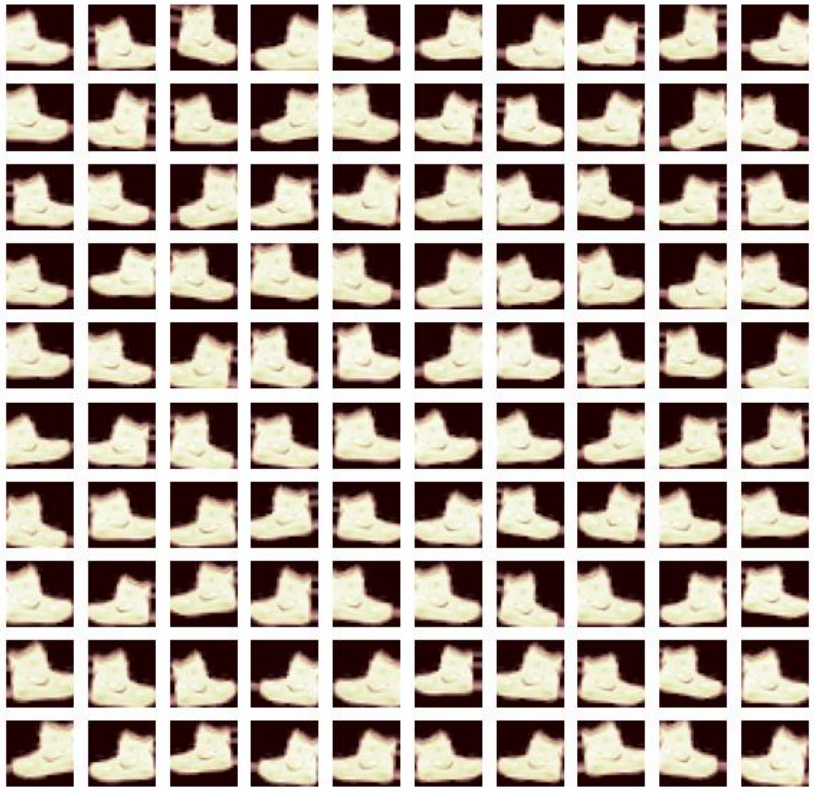
#flow()는 실제 보강된 이미지를 생성하는 함수는 Iterator 라는 객체를 생성하는데,

#이 객체에서는 값을 순서대로 꺼낼 수 있다. 그 방법은 next()를 사용해서 꺼낼 수 있고,

#보강된 이미지들이 첫 번째에 위치해 있기 때문에 [0]을 할당하여 꺼내주고, 한번에 꺼내는 이미지는 100장이 된다.

1.6 네 번째 실험 – VGGNet + 이미지 보강

```
#이미지 보강 확인
plt.figure(figsize=(10,10))
for i in range(100):          # 각각 반복하여 이미지를 출력
  plt.subplot(10,10,i+1)
  plt.axis('off')             # 축에 대한 정보 끄기
  plt.imshow(x_augment[i].reshape(28,28), cmap='pink')  # 차원수를 reshape()으로 재조정하여 출력
plt.show()
```



약간씩 다른 이미지가 생성

1.6 네 번째 실험 – VGGNet + 이미지 보강

- 이미지 추가 생성

• 훈련 데이터의 50% 추가생성 (60000 + 30000)

```
image_generator = ImageDataGenerator(
    rotation_range = 10,
    zoom_range = 0.10,
    shear_range = 0.5,
    width_shift_range = 0.10,
    height_shift_range = 0.10,
    horizontal_flip = True,
    vertical_flip = False)

augment_size = 30000

# 원본 이미지 무작위 선택 및 데이터 복사
x_choice = np.random.choice(train_X.shape[0], size = augment_size, replace=False)
x_augmented = train_X[x_choice].copy()
y_augmented = train_Y[x_choice].copy()

# 이미지를 변형할 원본 이미지를 찾기 위한 함수 예제 (중복허용o / 중복허용x)
print(np.random.randint(train_X.shape[0], size = augment_size))
print(np.random.choice(train_X.shape[0], size = augment_size, replace=False))
```

```
[56175 40732 8901 ... 29614 27565 50727]
[38706 40710 21005 ... 37882 58361 53357]
```

1.6 네 번째 실험 – VGGNet + 이미지 보강

- 이미지 추가 생성

• 훈련 데이터의 50% 추가생성 (60000 + 30000)

```
# 보강 이미지 데이터 생성
x_augmented = image_generator.flow(x = x_augmented,
                                   y = np.zeros(augment_size),
                                   batch_size = augment_size,
                                   shuffle = False).next()[0]
```

```
#보강 이미지 확인
plt.figure(figsize=(10,10))
for i in range(100):
  plt.subplot(10,10,i+1)
  plt.axis('off')
  plt.imshow(x_augmented[i].reshape(28,28), cmap='pink')
plt.show()
```

1.6 네 번째 실험 – VGGNet + 이미지 보강

```
#데이터 합쳐주기
train_X = np.concatenate((train_X,x_augmented))
train_Y = np.concatenate((train_Y,y_augmented))

#보강 데이터 결합 확인
print(train_X.shape)
```

```
(90000, 28, 28, 1)
```

1.6 네 번째 실험 – VGGNet + 이미지 보강

- 신경망 모델 생성

•가장 성능이 좋았던 VGGNet 스타일 적용

```
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(input_shape=(28,28,1),kernel_size=(3,3),
                           filters=32,padding='same',activation='relu'),
    tf.keras.layers.Conv2D(input_shape=(28,28,1),kernel_size=(3,3),
                           filters=64,padding='same',activation='relu'),
    tf.keras.layers.MaxPool2D(strides = (2,2)),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Conv2D(kernel_size=(3,3),filters = 128, padding='same', activation='relu'),
    tf.keras.layers.Conv2D(kernel_size=(3,3),filters = 256, padding = 'valid', activation='relu'),
    tf.keras.layers.MaxPool2D(strides = (2,2)),
    tf.keras.layers.Dropout(rate=0.5),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(units=512, activation = 'relu'),
    tf.keras.layers.Dropout(rate = 0.5),
    tf.keras.layers.Dense(units=256, activation = 'relu'),
    tf.keras.layers.Dropout(rate = 0.5),
    tf.keras.layers.Dense(units=10, activation = 'softmax')
])
```

1.6 네 번째 실험 – VGGNet + 이미지 보강

- 컴파일 단계

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

- 신경망 학습 단계

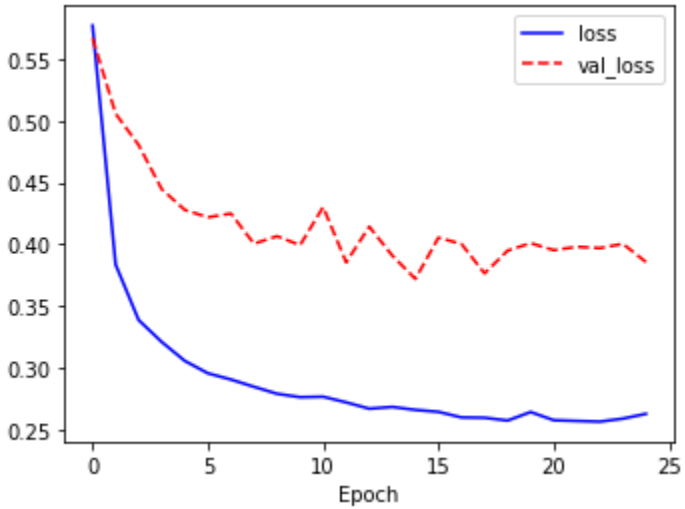
```
history = model.fit(train_X,train_Y,
                    epochs=500,
                    validation_split=0.25,
                    callbacks = [tf.keras.callbacks.EarlyStopping
                                 (patience = 10 , monitor = 'val_loss')])
```

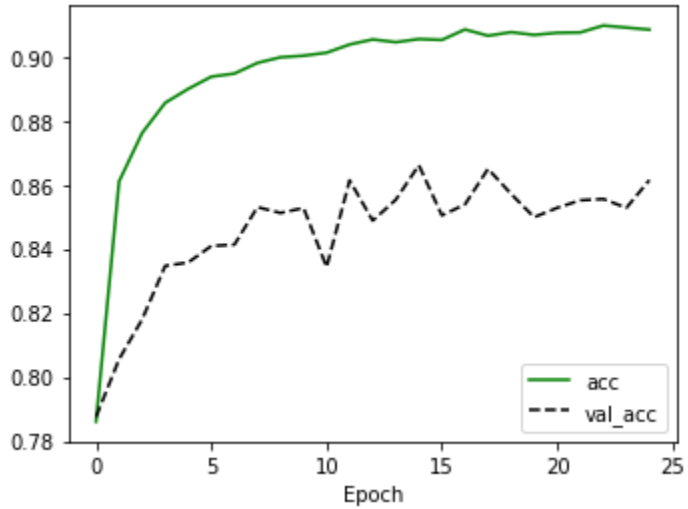
```
Epoch 1/500
2110/2110 [==============================] - 17s 8ms/step - loss: 0.5772 - accuracy: 0.7860 - val_loss: 0.5670 - val_accuracy: 0.7875
Epoch 2/500
2110/2110 [==============================] - 17s 8ms/step - loss: 0.3836 - accuracy: 0.8611 - val_loss: 0.5058 - val_accuracy: 0.8055
Epoch 3/500
2110/2110 [==============================] - 17s 8ms/step - loss: 0.3387 - accuracy: 0.8762 - val_loss: 0.4804 - val_accuracy: 0.8180
Epoch 4/500
2110/2110 [==============================] - 17s 8ms/step - loss: 0.3210 - accuracy: 0.8856 - val_loss: 0.4447 - val_accuracy: 0.8347
Epoch 5/500
2110/2110 [==============================] - 17s 8ms/step - loss: 0.3057 - accuracy: 0.8900 - val_loss: 0.4279 - val_accuracy: 0.8357
```

⋮

1.6 네 번째 실험 – VGGNet + 이미지 보강

- 정확도 및 손실값에 대한 결과 시각화






```
plt.figure(figsize = (12,4))
plt.subplot(1,2,1)
plt.plot(history.history['loss'],'b-',label = 'loss')
plt.plot(history.history['val_loss'],'r--',label = 'val_loss')
plt.xlabel('Epoch')
plt.legend()
plt.subplot(1,2,2)
plt.plot(history.history['accuracy'],'g-',label = 'acc')
plt.plot(history.history['val_accuracy'],'k--',label = 'val_acc')
plt.xlabel('Epoch')
plt.legend()
plt.show()
```

1.6 네 번째 실험 – VGGNet + 이미지 보강

```
model.evaluate(test_X,test_Y)
```

```
313/313 [==============================] - 1s 3ms/step - loss: 0.2097 - accuracy: 0.9262
```

최종 평가

- 다층 퍼셉트론 → loss: 0.5899 - accuracy: 0.8197
- 합성곱 계층만 사용 → loss: 0.5002 - accuracy: 0.868
- 합성곱, 풀링, 드롭아웃 사용 → loss: 0.3248 - accuracy: 0.8921
- VGGNet스타일 기법 → loss: 0.2306 - accuracy: 0.9175
- VGGNet + 이미지 보강 → loss: 0.2097 - accuracy: 0.9262

 # 하이퍼 파라미터는 동일

 # 정확도 및 손실값 어느정도 향상

 # VGGNet 스타일 적용

 # 이미지 보강기법 활용

 # 과적합 현상 발생

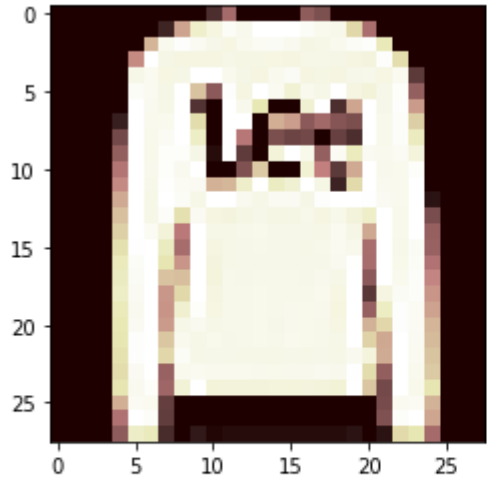
 # 92% 이상의 정확도 달성

1.6 네 번째 실험 – VGGNet + 이미지 보강

- 예측 수행

```
pred_X = model.predict(test_X[[1]])
#예측 이미지 확인
plt.imshow(test_X[1].reshape(28,28), cmap = 'pink')
plt.show()

#예측 결과 확인
print(pred_X+1)
print("예측 수행 범주 :",test_Y[1])
```




```
[[1.        1.        1.9994457 1.        1.0001094 1.        1.000445
  1.        1.        1.        ]]
예측 수행 범주 : 2
```