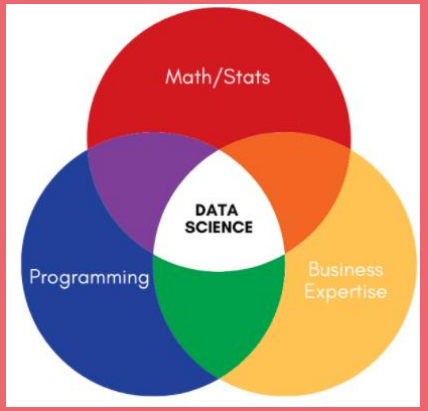
# Deep Learning

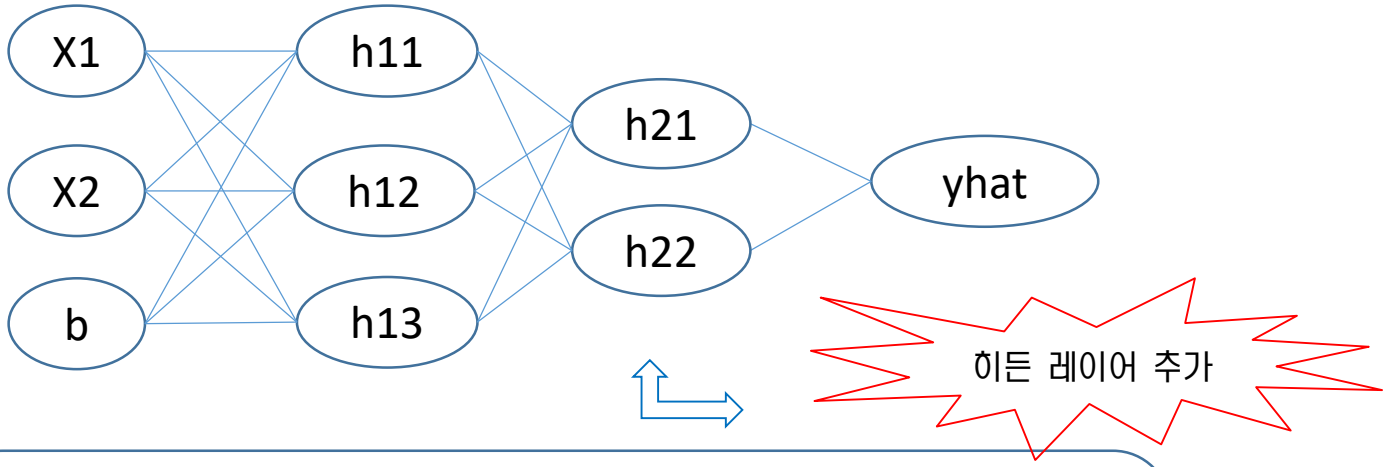
인하대학교
데이터사이언스 학과
김 승 환

swkim4610@inha.ac.kr



# 1 Deep Learning

Deep Learning 모형은 Neural Network 모형에서 Hidden Layer를 2개 이상으로 확장한 모형이므로 간단하게 신경망 모형에서 딥러닝 모형으로 확장 가능하다.




$$X \Rightarrow n \times p, W_1 \Rightarrow p \times n_{h_1}, h_1 = S(X \cdot W_1) \Rightarrow n \times n_{h_1}$$

$$h_1 = n \times n_{h_1}, W_2 \Rightarrow n_{h_1} \times n_{h_2}, h_2 = S(h_1 \cdot W_2) \Rightarrow n \times n_{h_2}$$

$$h_2 = n \times n_{h_2}, W_3 \Rightarrow n_{h_2} \times 1, \hat{y} = S(h_2 \cdot W_3) \Rightarrow n \times 1$$

# 1 Deep Learning

텐서플로우에서는 아래와 같이 model.add()에 의해 Deep Learning 모형으로 확장한다.

아래는 노드가 10개인 히든 레이어를 많이 추가한 결과다.

이와 같이 텐서 플로우에서 Hidden Layer 를 추가하는 것은 매우 간단한 작업이다.

```
actFunc = 'sigmoid'
model = tf.keras.Sequential()
model.add(layers.Dense(10, activation=actFunc, input_dim=2))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(1, activation='sigmoid'))
```

XOR_DNN_TF2.ipynb

이처럼 매우 간단하게 Hidden Layer를 추가하여 딥러닝 구조로 학습할 수 있다.

하지만, XOR 문제를 위와 같이 딥러닝 모형으로 확장하면 원하는 결과를 얻을 수 없다.

왜 일까?

# 1 Deep Learning

텐서플로우에서는 아래와 같이 model.add()에 의해 Deep Learning 모형으로 확장한다.

아래는 노드가 10개인 히든 레이어를 많이 추가한 결과다.

이와 같이 텐서 플로우에서 Hidden Layer 를 추가하는 것은 매우 간단한 작업이다.

```
actFunc = 'sigmoid'
model = tf.keras.Sequential()
model.add(layers.Dense(10, activation=actFunc, input_dim=2))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(1, activation='sigmoid'))
```

XOR_DNN_TF2.ipynb

이처럼 매우 간단하게 Hidden Layer를 추가하여 딥러닝 구조로 학습할 수 있다.

하지만, XOR 문제를 위와 같이 딥러닝 모형으로 확장하면 원하는 결과를 얻을 수 없다.

왜 일까?

# 2 Gradient Vanishing problem

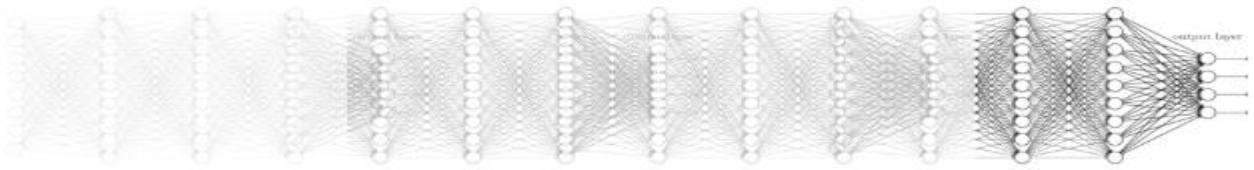
XOR 문제를 딥러닝으로 확장하니 전혀 학습이 되지 않았다. 이유는 무엇일까?
이 문제가 유명한 gradient Vanishing problem 이다.
우리는 활성함수로 Sigmoid 함수를 사용하는데 이는 출력을 0과 1사이로 만들고자 하는 의도에서 사용했다.

Backpropagation 계산과정에서 가중 값을 계산할 때, Sigmoid 함수의 미분 값을 계산하는데 이 때, 시그모이드 함수의 입력값(WX+b)의 절대값이 크면, 기울기가 거의 "0"이 되어 가중 값이 더 이상 업데이트 되지 않는 문제가 발생한다.

$$W_i = W_i + t(h_{i-1})\lambda\delta_i, \delta_i = \delta_{i+1}t(W_{i+1})h_i(1-h_i)$$

위 식에서 h(1-h)의 최대값은 h=0.5 일 때, 0.25가 되고 많은 레이어를 통과하면서 계속 0.25보다 작은 수를 $\delta_i$에 곱해 결국 0으로 만든다.

Vanishing gradient (NN winter2: 1986-2006)

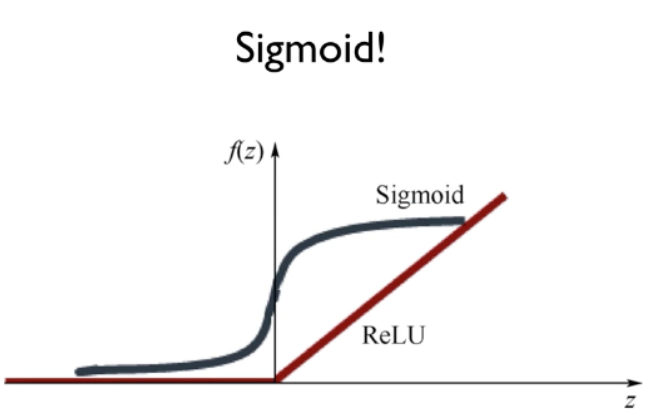


# 2 Gradient Vanishing problem

이 경우, 가장 흔한 솔루션은 시그모이드 함수 대신 ReLU 함수 ReLU(x)=max(0,x)를 사용하는 것이다.
이 함수는 기울기가 0 혹은 1이다.
딥러닝의 최종 목표치는 0과 1사이에 있으므로 활성함수의 입력을 0~1 사이로 만드는 것이 필요하다. 하지만, 시그모이드 함수를 사용할 경우, 이 작업을 너무 많이 하여 입력이 거의 "0"이 되어 버리는 문제가 발생한다.
ReLU 함수는 Sigmoid 함수처럼 음수는 "0"으로 처리하고 양수는 Bypass 하는 것이다.
ReLU가 음수들을 모두 0으로 처리하기 때문에 그 노드는 학습되지 않는다는 단점이 있어 요즘은 Leakly ReLU, maxout 등을 더 많이 사용한다.

구현은 간단하다. 아래와 같이 sigmoid 대신 relu 함수로 바꿔주면 된다.
최근에는 Leakly RELU, ELU, Maxout 과 같은 함수를 많이 사용한다.

이제, 직접 relu 함수로 바꿔 문제를 해결해보세요~




```
actFunc = 'relu'
model = tf.keras.Sequential()
model.add(layers.Dense(10, activation=actFunc,
input_dim=2))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(10, activation=actFunc))
model.add(layers.Dense(1, activation='sigmoid'))
```

Q: 마지막은 왜 Sigmoid 일까요?

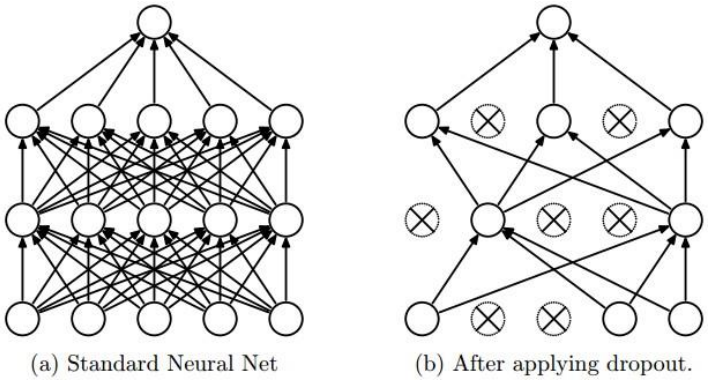
XOR_DNN_TF2.ipynb

# 2 Gradient Vanishing problem

신경망 모형에서 초기값에 따라 결과가 달라지기도 하고 해에 수렴하는 속도도 달라진다.
그런 이유로 초기값을 어떻게 줄 것이냐 역시 연구의 대상이다.
가중치의 분산을 일정 수준 이하로 작게 만들면 더 빠른 수렴을 보인다는 연구결과에 의해 나타난 방법이
Xavier 초기값이다.

```
W=np.random.randn(fan_in, fan_out)/np.sqrt(fan_in) # Xavier initialization(2010)
W=np.random.randn(fan_in, fan_out)/np.sqrt(fan_in/2) #He initialization(2015)
```

여기서, np.random.randn()는 표준정규분포의 난수이고, fan_in은 입력노드의 수, fan_out은 출력 노드의 수이다.
TF 2.X 버전에서는 kernel_initializer='he_normal' 형식으로 사용할 수 있다.



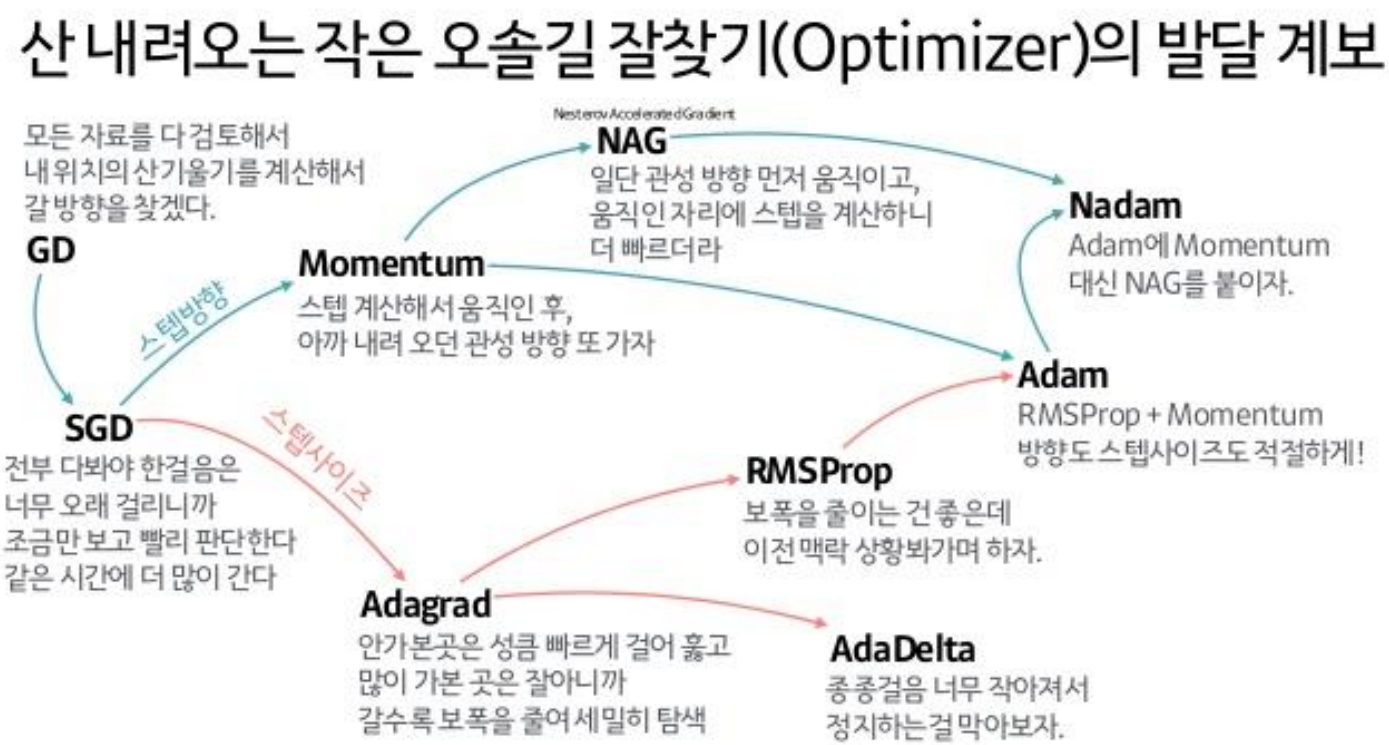
(a) Standard Neural Net (b) After applying dropout.

DropOut: 레이어 마다 어떤 노드를 죽일 것인지를 랜덤하게 결정한다.

TF 2.X에서는 layers.Dropout(0.3) 과 같은 형식으로 사용한다. 0.3은 30%를 죽인다는 의미다.

# 3 Optimizer

- Gradient Descent 방법은 SGD → Momentum → Adam 등으로 발달한다.
- 궁극적인 목적은 빠르게 해에 도달하기 위해 여러 아이디어를 사용하는 것이다.



# 4 Mnist – Acc. 92%

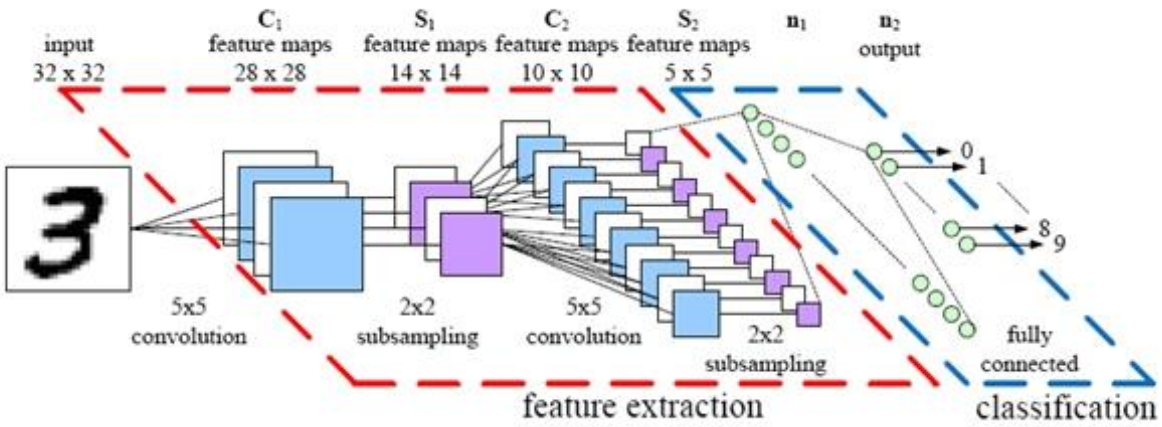
Mnist Data를 Softmax Regression으로 예측한 결과 약 89% 정도의 정확도를 얻을 수 있었다. 이제, 노드 256개 가진 히든 레이어가 2개를 연결하여 Deep Neural Network 모형으로 예측해보자. Softmax Regression에서 아래의 부분만 달라지면 된다. 적용 결과, 약 92%의 정확도를 가진다.

```
# Mnist_DNN_Model_Save_TF2.ipynb

#Hidden Layer 추가
from tensorflow.keras import layers
model = tf.keras.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=784))
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.compile(optimizer='sgd', loss='categorical_crossentropy',metrics=['accuracy'])
model.fit(train_x, train_y_onehot, batch_size = 100, epochs=5)
```

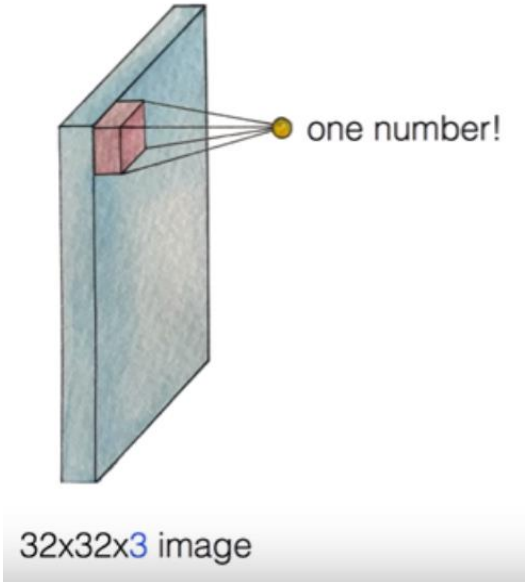
# 5 CNN

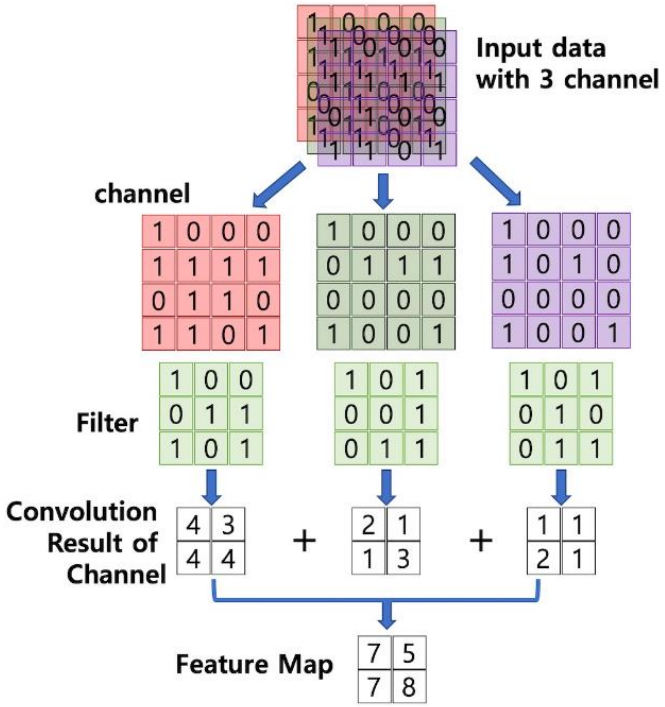
- CNN 처리 과정은 크게 두 과정을 구분된다.
  첫번째 과정은 이미지로부터 Feature를 Extraction 하는 과정이다. 다음은 Feature를 이용하여 딥러닝을 수행하는 것이다.
- Feature Extraction은 Convolution layer를 이용하여 이미지의 특징을 추출하고 Pooling Layer에서 대표값을 만드는 과정을 반복하여 특징을 단순화해간다.
- 단순화된 특징은 이미지를 구별할 수 있는 간결한 특징만 남고 판별에 도움이 되지 않는 자세한 이미지 정보는 삭제된다.
- CNN은 사람이 이미지를 인식하는 과정과 유사하다.
  사람이 이미지를 볼 때, 이미지 전체를 학습하는 것이 아니라 몇 개의 특징을 학습한다고 한다.



# 5 CNN

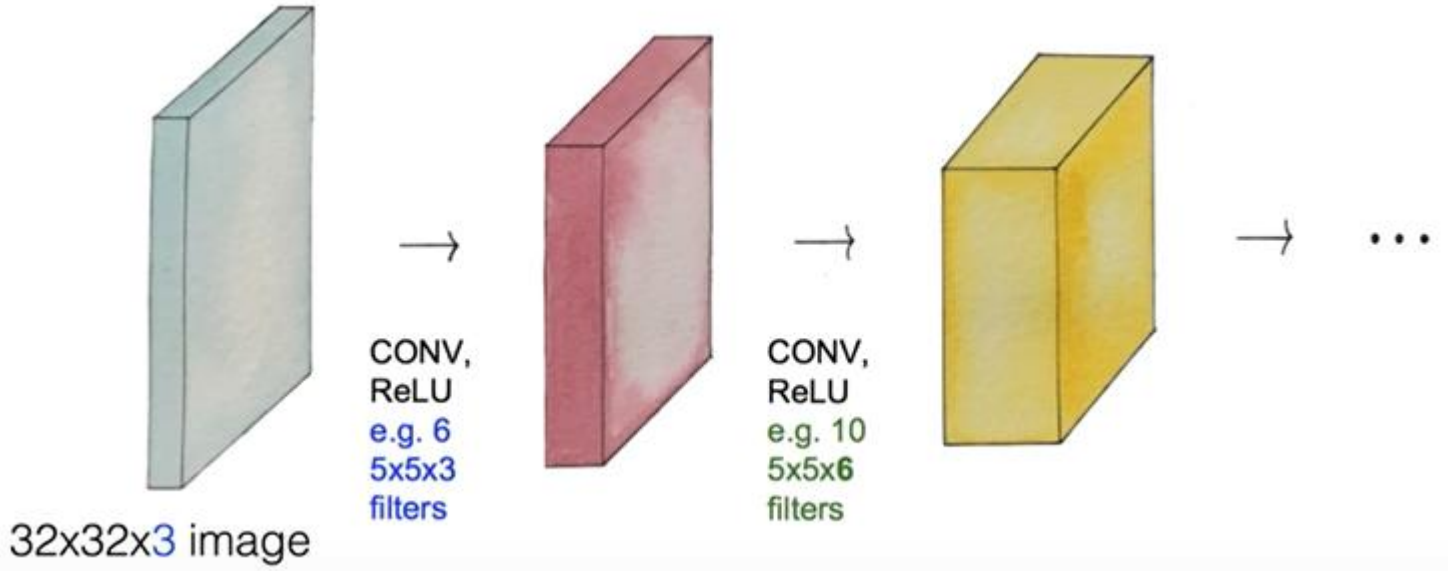
- 좌측은 32*32 크기의 컬러 이미지에 Convolution Filter를 끼워 하나의 값을 만드는 설명이다.
- 이미지는 4*4 크기 R, G, B 채널이 있고, 필터는 3*3 이라고 하자.
- 필터를 왼쪽 상단에 끼고 필터와 채널 값을 elementwise 곱을 구한 후 더하면 하나의 값이 만들어 진다. 이 과정을 세 채널에 대해 반복하여 모두 더하면 하나의 값이 생성된다.
- 이후, 필터를 우로 한 칸 이동하여 같은 계산을 반복한다.
- 우로 끝까지 가면 다시 왼쪽에서 한 칸 내려 위의 연산과정을 반복한다.



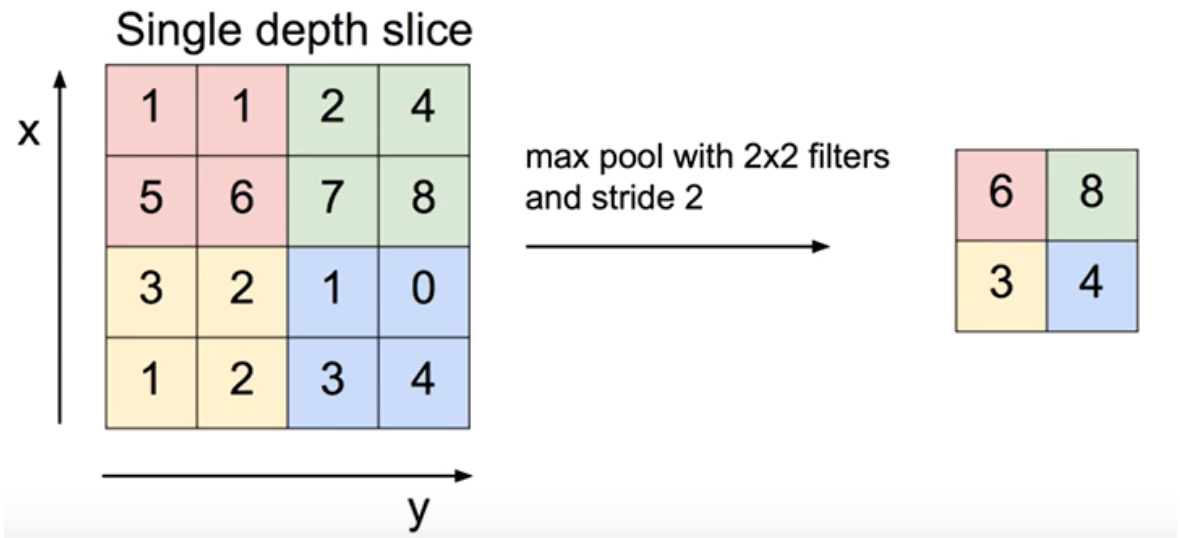
# 5 CNN

- 아래 그림은 원본 이미지에 5*5*3 필터를 끼워 6장의 Convolution Feature를 만든 후, 다시 5*5*6 필터를 끼워 10장의 Convolution Feature를 만드는 예다.
- 이 과정을 반복하면서 원본 이미지는 점차 정보가 줄어들면서 Feature Map으로 바뀌는 것이다.



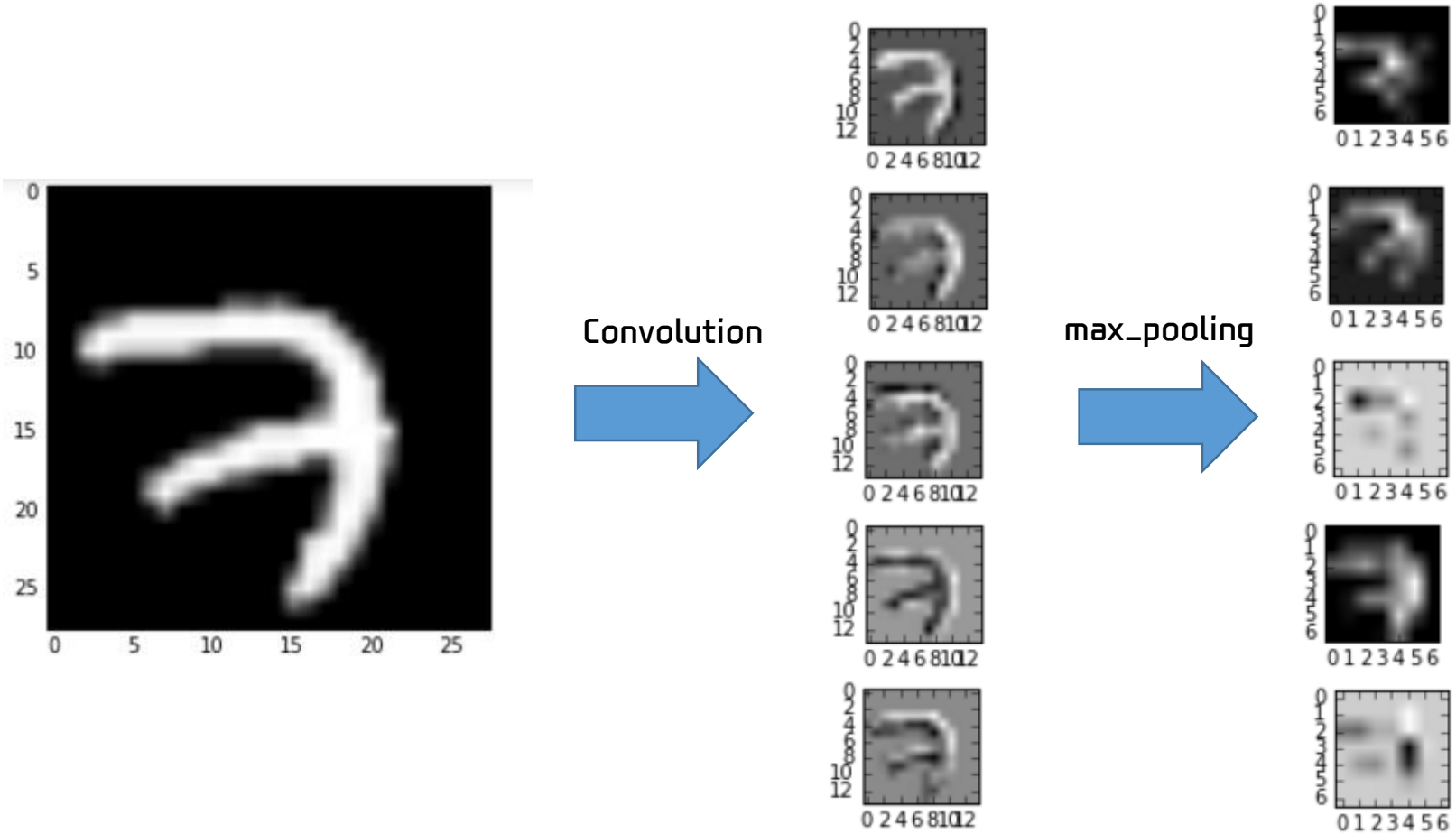
# 5 CNN

- 아래와 같이 가장 큰 값을 Sampling 하는 방법을 Max Pooling이라고 한다.
- 평균을 추출할 수도 있고, 최소값을 추출할 수도 있다. 일반적으로 Max Pooling을 가장 많이 사용한다.
- 아래는 2*2 커널을 2*2 Stride를 사용해 얻은 결과이다.



# 5 CNN

- 이와 같이 Convolution Layer, Pooling Layer를 통과하면서 이미지는 축약된다.
- 축약된 이미지는 원본 이미지보다 인식률이 더 좋은 결과를 줄 수 있다는 것이 CNN의 핵심이다.



# 5 CNN

- train_images는 60000장이고 28*28*1 형식으로 행렬을 바꾼다.
- 여기서, 1은 단색 컬러이기 때문이다. 만약, RGB 컬러의 경우는 3이 된다.
- 이후, 픽셀 값을 255 로 나누어 픽셀 값을 0~1 사이로 만들고, 레이블을 one-hot encoding 한다.

```
# Mnist_CNN_TF2.ipynb
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras import datasets
from tensorflow.keras.utils import to_categorical

(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1))
test_images = test_images.reshape((10000, 28, 28, 1))

# 픽셀 값을 0~1 사이로 정규화합니다.
train_images, test_images = train_images / 255.0, test_images / 255.0
train_y_onehot = to_categorical(train_labels)
test_y_onehot = to_categorical(test_labels)
```

# CNN

학습해야 하는 파라미터 수가 34,826개 임을 알 수 있다.

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 26, 26, 32)        320
max_pooling2d (MaxPooling2D) (None, 13, 13, 32)        0
conv2d_1 (Conv2D)            (None, 11, 11, 64)        18496
max_pooling2d_1 (MaxPooling2 (None, 5, 5, 64)          0
flatten (Flatten)            (None, 1600)
dense (Dense)                (None, 10)                16010
=================================================================
Total params: 34,826
Trainable params: 34,826
Non-trainable params: 0
```

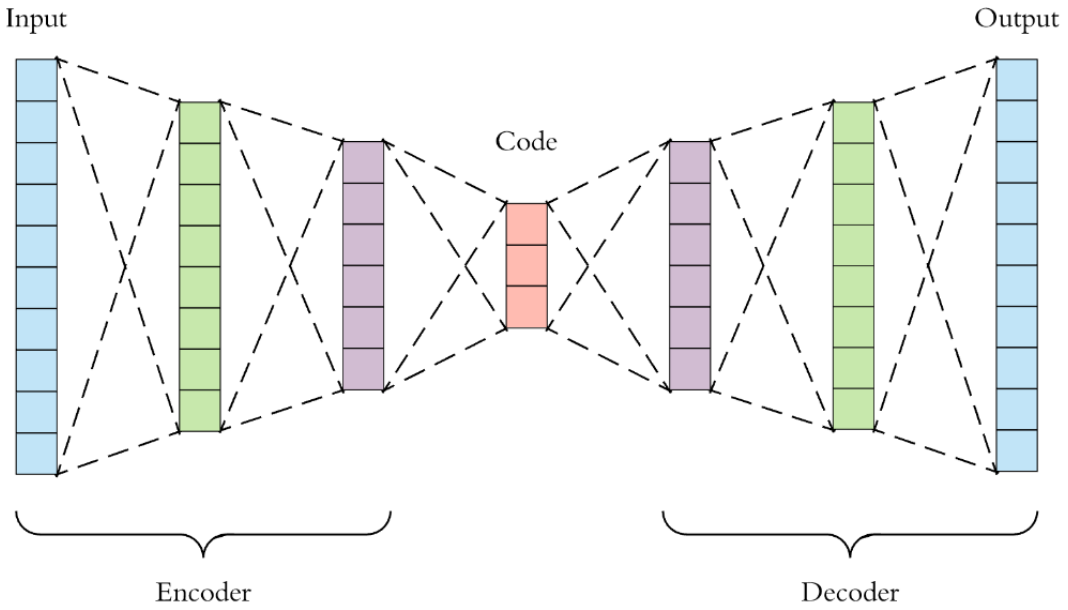
# 5 CNN

히든 레이어가 없는 간단한 신경망 모형을 통과해 얻은 최종 결과는 정확도 99%이다.
상당히 높은 정확도이다. Convolution Layer, Max Pooling을 통해 픽셀 이미지를 축약하는 것이 더 좋은 결과를 가져다 준 것이다. 또한, 이 계산에 GPU는 CPU 대비 10~20배 정도 빠른 성능을 보여준다.

```
model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(train_images, train_y_onehot, epochs=5)
test_loss, test_acc = model.evaluate(test_images,  test_y_onehot, verbose=2)
print(test_acc)
```

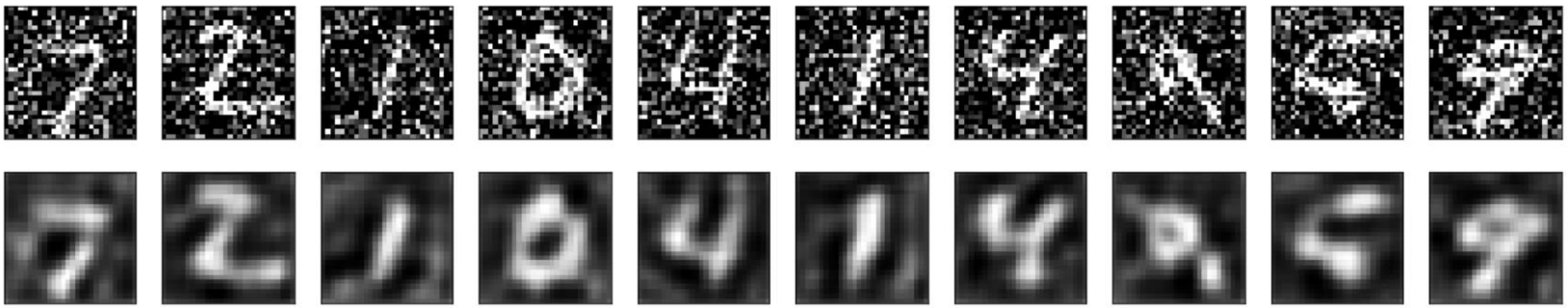
# 6 Deep Learning 응용

- AutoEncoder는 Input과 Output이 같은 딥러닝 모형임
- Target이 없으므로 Unsupervised Learning 형태임
- Input이 히든 레이어를 통과했다가 다시 재현되므로 암호화, 복호화가 일어나는 것임
- 일종의 통계학의 PCA(Principal Component Analysis)에 대한 비선형 버전으로 볼 수 있음
- 아래 그림에서 Code는 무엇일까? Code는 Input의 압축정보로 볼 수 있음
- AutoEncoder는 Noise Reduction, Anomaly Detection 등의 용도로 사용됨



# 6 Deep Learning 응용

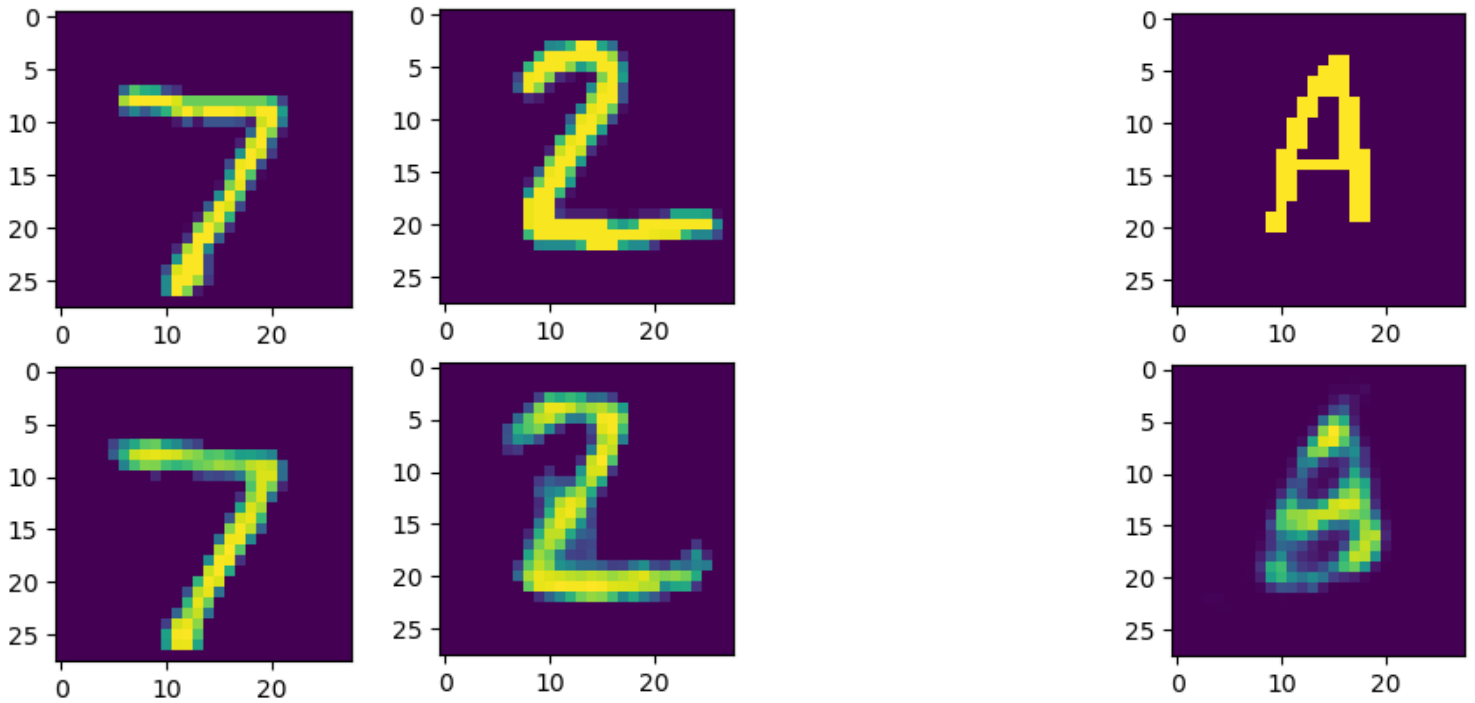
- Noise Reduction, Anomaly Detection 의 원리는 차원축소 과정에서 뚜렷하지 않은 특징은 사라지는 원리를 이용함
- 아래 그림은 상단이 노이즈가 추가된 원본 이미지이고, 하단이 노이즈가 제거된 이미지임
- 이 원리를 이용하면 노이즈가 섞인 이미지에서 노이즈를 제거할 수 있음
- 같은 원리로 비정상적인 신호가 포함된 신호를 오토인코더에 넣어 학습시킨 후, 새로운 신호를 넣었을 때, 다른 신호에 비해 입력과 출력이 많이 다르면 비정상 신호로 판정하는 원리로 Anomaly Detection을 할 수 있음
- 아래의 코드는 flatten 형태와 convolution 층을 사용한 방법의 오토인코더임



Mnist_AE_TF2.ipynb, Mnist_AE_CNN_TF2.ipynb

# 6 Deep Learning 응용

- AE로 anomaly detection을 수행해 보자.
- AE로 정상 신호를 복원하는 훈련을 한다. 이후, 들어오는 신호는 정상도 있고, 비정상도 있을 경우, 정상신호는 비교적 잘 복원할 것이지만, 비정상 신호는 학습한 적이 없기 때문에 복원률이 떨어질 것이다.
- AE의 복원률을 이용해 비정상 신호를 탐지하는 지능형 시스템을 만들 수 있다.



AnomalyDetection_Mnist_TF2.ipynb

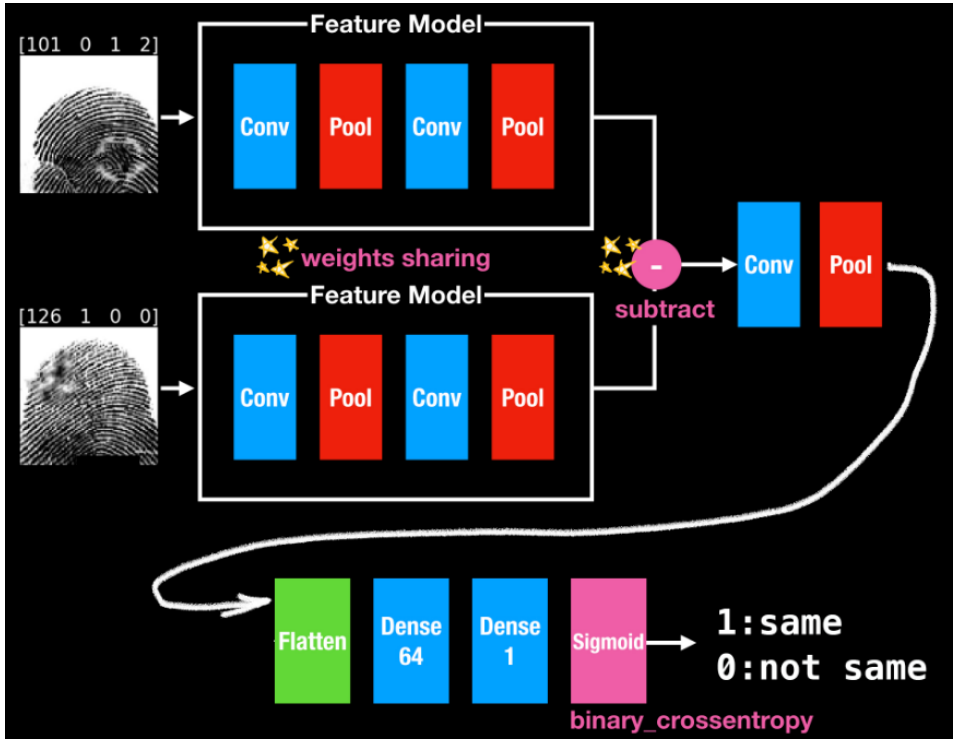
# 6 Deep Learning 응용

- 아래는 오토인코더를 이용해 그레이 이미지를 입력해서 Color 이미지를 출력한 예임
- 이 모형은 다량의 컬러 이미지를 그레이화 하여 그레이 이미지는 입력으로, 컬러 이미지는 출력으로 학습시킨 후, 새로운 그레이 이미지를 주면 컬러로 만들어 주는 기술임
- 컬러 복원이 아니고 색칠하기임
- 이를 응용하면 흑백TV 시절의 이미지나 동영상을 컬러로 만들 수 있음

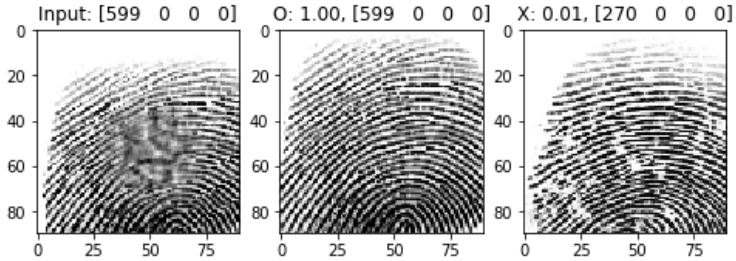
Colorization_TF2.ipynb

# 6 Deep Learning 응용

- 지문인식은 가장 널리 사용되는 생체인식 방법임
- 등록된 지문이 존재할 때, 새로운 지문이 등록된 지문과 일치 여부를 판별하는 문제임
- 상단의 지문이 입력되면 상단에서 추출된 지문의 Feature와 하단의 Feature의 차이를 CNN으로 Same or not same을 판별하는 문제임
- 수행결과, 99.7% 정도의 정확도를 보임

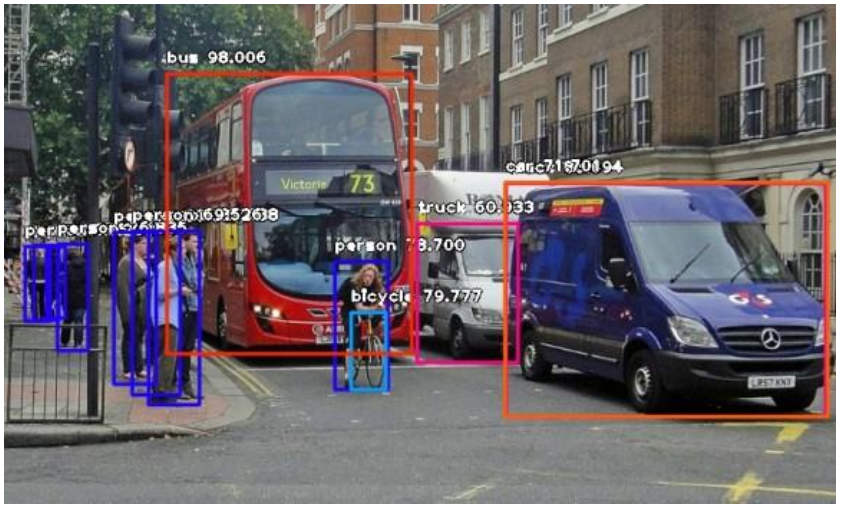



finger.ipynb



# 6 Deep Learning 응용

person : 56.95696473121643
person : 52.80924439430237
person : 70.20382285118103
person : 76.83471441268921
person : 78.70017290115356
bicycle : 79.77737784385681
person : 83.55740308761597
person : 89.43805694580078




truck : 60.93311905860901
person : 69.52624917030334
bus : 98.00647497177124
truck : 83.6944580078125
car : 71.70088291168213

# 감사합니다

인하대학교
데이터사이언스 학과
김 승 환

swkim4610@inha.ac.kr