Convolutional Neural Network

https://youtu.be/HFAHK5orJjg

Contents

CryptoCraft LAB

Deep learning

machine learning

❖ machine learning

- 선형회귀를 사용하여 데이터가 생성된 일정한 패턴을 추론
- 지도학습 / 비지도학습으로 나뉨
  - 지도학습 : input vector를 통해 output label을 예측
    1. 옳은 label을 가진 training set을 통해 모델 생성
    2. 오차를 줄이기 위해 parameter 값을 조절
    3. 학습에 사용하지 않은 데이터로 검증
    4. linear regression, decision tree, naïveBayes, neural network 등의 방법 사용
- training data & test data 사용
  - training data : generate models
  - test data : test

deep learning

❖ machine learning은 데이터로부터 특징들을 학습

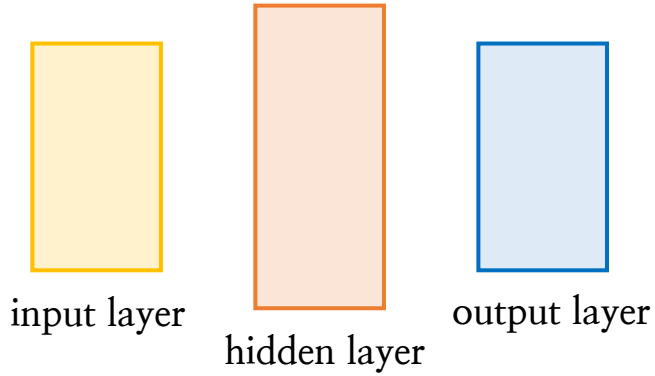
- hidden layer가 2개 이상이면 deep

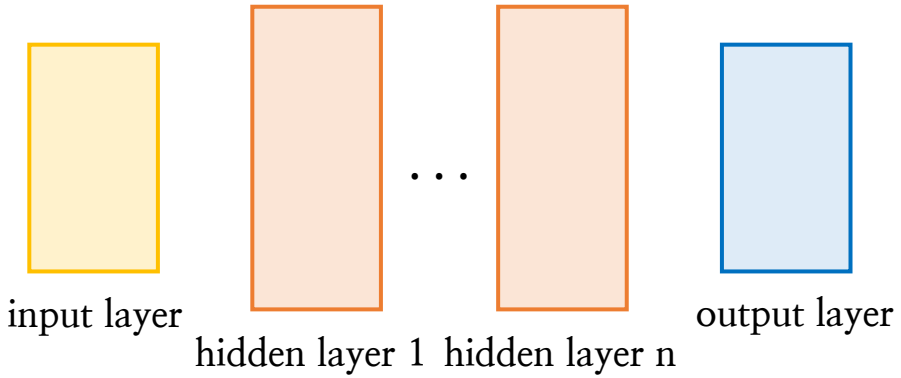
❖ 목표

- 최적의 parameter 설정 통한 최적화 → 높은 정확도
  → 일반화

❖ deep learning 관점

1. 문제 분석 : 가지고 있는 자원, 해결하려는 문제
2. 구조 선택 : 어떤 구조의 Neural Network 사용할지, layer 개수 등
3. function, 학습 전략 선택





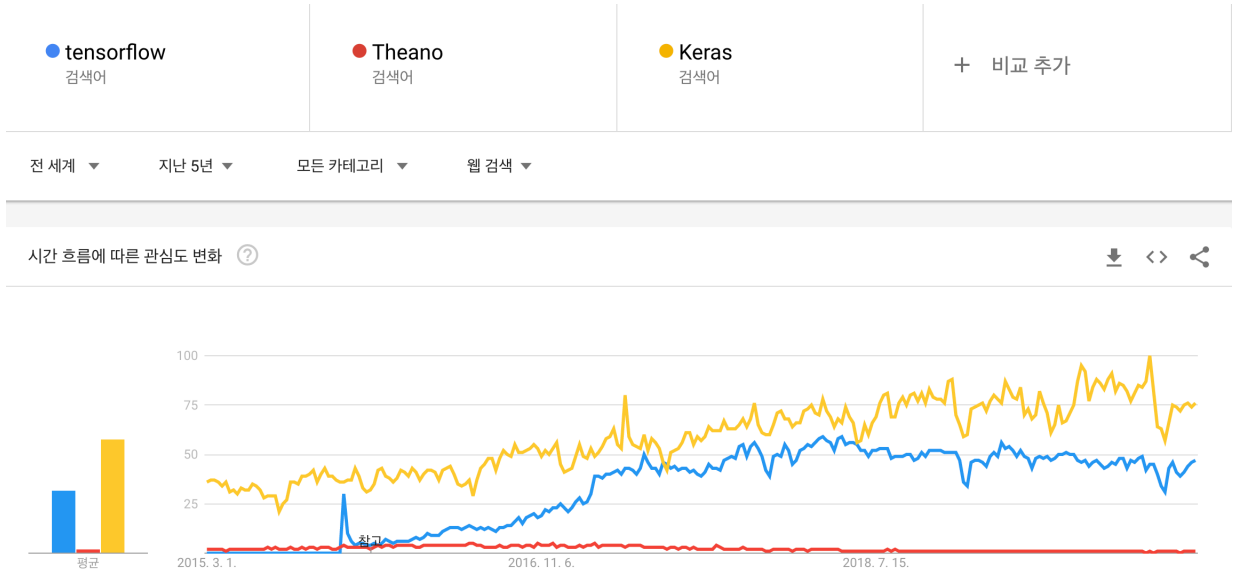
❖ neural network 종류

- Convolutional NN(CNN), Recurrent NN(RNN), Deep NN(DNN), Generative Adversarial Networks(GANs) 등

deep learning framework

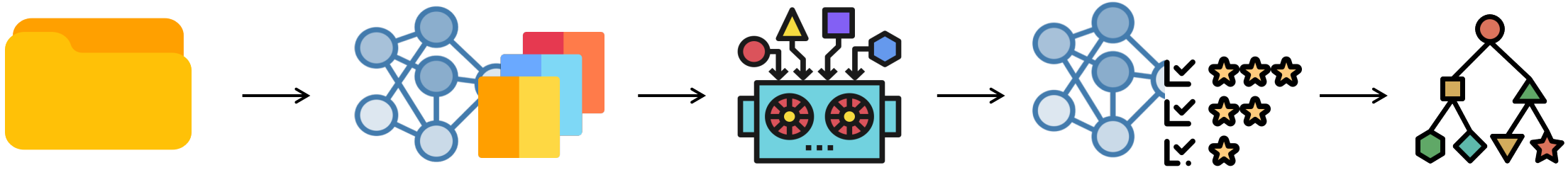
❖ Tensorflow vs Theano vs Keras

- TF
  - Python, C/C++ 모두 사용 가능
  - Tensorboard 통해 기록된 log를 실시간으로 시각화 → 학습 과정 확인 가능
  - 병렬처리에 적합
  - data augmentation 쉽게 가능 (rotation, flipping 등)
- Theano
  - 확장이나 개선이 어려운 복잡한 구조
  - 컴파일이 오래 걸림
- Keras
  - Tensorflow와 Theano 기반
  - 가장 빠르게 발전 중인 프레임워크이며 사용하기 쉬움
  - Tensorflow에서 Keras 모듈 통해 neural network를 쉽게 생성 가능

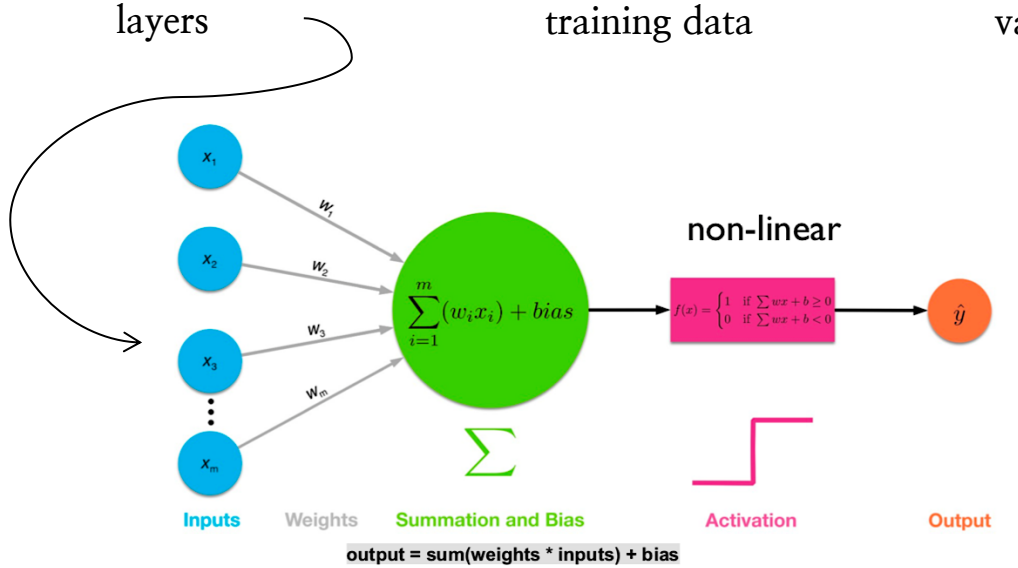


deep learning

데이터 생성 → 모델 구성 → 학습 → 모델 검증 & 갱신 → 테스트



training / validation / test | layers | training data | validation data | test data



이전 layer의 output = 현재 layer의 input

활성화 함수 거쳐 output으로

deep learning process

data generation → modeling → training → validation → test

- data set
- data augmentation

data

❖ training set

- 학습 데이터
- 보통 accuracy 100%에 도달

❖ validation set

- 학습에 사용, 학습 정도를 검증하기 위한 데이터
- 보통 accuracy 60~80%
- 오류 추정 통해 모델 선택에 사용
  - ➢ hidden layer수 등의 hyperparameter 선택 & 변경하며 모델 설정

❖ test set

- 최종 선택된 모델을 통해 분류할 데이터

```
Epoch 99/100
Epoch 1/100
100/100 - 8s - loss: 0.0064 - acc: 0.9985 - val_loss: 2.3711 - val_acc: 0.7500
Epoch 100/100
Epoch 1/100
100/100 - 8s - loss: 5.1598e-05 - acc: 1.0000 - val_loss: 2.3921 - val_acc: 0.7580
```

data

❖ data 구성

- 원본 = training / test, training → training / validation 으로 나눔
- 보통 60 : 20 : 20 or 50 : 25 : 25 비율

➢ sklearn의 train_test_split() method 사용

➢ Training set에 대해 한 번 더 수행하면 validation set으로도 나눌 수 있음

```
from sklearn.model_selection import train_test_split
train_test_split(arrays, test_size, train_size, random_state, shuffle, stratify)
```

전체 data set을 받아 랜덤하게 training/ test set 으로 분리해주는 함수

arrays : 분할 시킬 데이터
test/train_size : 비율(float) or 개수(int)
random_state : 데이터 셔플 위한 시드값
shuffle : 셔플 여부 (default = True)
stratify : 데이터의 비율
전체 데이터의 label의 비율이 30 : 70이면 분할한 데이터도 해당 비율 유지

data generation

❖ Keras에서 사용 가능한 ImageDataGenerator class를 import

```
from keras.preprocessing.image import ImageDataGenerator
```

이미지 전처리 위한 클래스

- 이미지를 batch단위로 불러오는 generator 생성
- 이미지 변형 & 정규화
  - ➢ training data 뿐만 아니라 새로운 데이터도 잘 분류하도록
    - ➢ augmentation → overfitting 방지

```
train_datagen = ImageDataGenerator(rescale=1./255)
test_datagen = ImageDataGenerator(rescale=1./255)
```

- 효과적 학습 위해 원본 이미지 RGB(0~255)를 1./255로 스케일링 → 0~1범위로 변환

```
train_generator = train_datagen.flow_from_directory(
        'dataset/train',                #디렉토리명
        target_size=(24, 24),           #이미지 사이즈
        batch_size=3,
        class_mode='categorical')       #다중클래스 분류
```

generator 생성

- flow(data, labels)
- flow_from_directory(directory)
  - ➢ directory 형태로 데이터 가져와서 사용 가능

batch size

data set

batch size

- 전체 dataset 중, 한번에 넘겨주는 데이터의 수

1 epoch = batch size x iteration

- 전체 데이터를 batch size로 나누어 iteration → 전체를 한 번 학습

data augmentation

❖ augmentation

- 다른 데이터로 보이도록 변형하여 다양한 데이터로 학습하는 효과
- 이미지 특징에 대해 더 많은 시나리오를 생성
- training set에 없는 데이터도 제대로 분류되도록 함
  - ex) 서있는 사람 이미지를 45도, 90도 등으로 회전시켜 학습 → 누워있는 사람 이미지도 사람으로 분류 가능

➢ overfitting 방지

- 데이터를 받아와서 전처리 하는 과정에서 변환
  - tensorflow가 데이터를 직접 변경 X
  - 원본은 그대로

```
train_datagen = ImageDataGenerator(rescale=1./255)
```

```
train_datagen = ImageDataGenerator(
    rescale=1./255,                 #0~1사이로 정규화
    rotation_range = 40,            #회전 범위
    width_shift_range = 0.2,        #가로로 랜덤 shift 하는 범위
    height_shift_range = 0.2,       #세로로 랜덤 shift 하는 범위
    shear_range = 0.2,              #잘라서 왜곡시킴
    zoom_range = 0.2,               #20%까지 랜덤 줌
    horizontal_flip = True,         #랜덤 뒤집기
    fill_mode = 'nearest'           #이미지 변형으로 손실되는 픽셀 채움
)
```

val_acc: 0.7580	val_acc: 0.8320
before	after

❖ overfitting

- Training data에 대해 학습이 너무 잘 되어서 해당 데이터들에 대해 높은 정확도 가짐
  but test data 등 실제 모델 사용 시 일반화 성능이 떨어지는 현상

❖ Overfitting 방지 기법

- L2 regularization
  가중치가 클 수록 큰 패널티 부과
  → 영향을 크게 미치는 입력데이터에 대해 더 큰 패널티
- dropout
  각 layer마다 뉴런을 일정 비율로 drop하여 사용하지 않음
- hyperparameter optimization
  손실 함수가 극소값을 갖는 hyperparameter 설정
- data augmentation
  전처리 과정에서 이미지 회전, 뒤집기 등의 변형을 통해 데이터 수 증가
- transfer learning
  미리 학습된 가중치를 초기값으로 설정하여 재학습

deep learning process

data generation → modeling → training → validation → test

- parameter
- layer
- activation

layer

❖ neural network는 layer를 쌓아 구성

- dense layer / convolution layer / pooling layer / dropout layer / LSTM 등의 layer 사용
- MLP, CNN, DCNN, RNN …

```
model = Sequential()
model.add(Dense(1, input_dim=1))
```

```
model = Sequential()
model.add(LSTM(32, input_shape=(None, 1)))
model.add(Dropout(0.3))
model.add(Dense(1))
```

```
model = Sequential()
model.add(Dense(64, input_dim=12, activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

layer 추가

- keras의 모델 설계 인터페이스
- layer 구성 위해 사용

parameter

❖ parameter와 hyperparameter로 나뉨

- parameter
  1. 모델 내부적으로 생성하는 값 (사람이 설정하지 않음, 초기값 신경 안 써도 됨)
  2. weight, bias
  3. 각 layer를 거치면서 계산되는 값 = 각 layer의 output의 수
  4. fitting 과정에서 모델이 알아서 수정해나감
- hyperparameter
  1. data & model에 맞게 사람이 직접 설정
  2. neuron 개수, activation function, metric 종류 등
  3. model.summary() : 각 layer의 output의 변화(hyperparameter) 볼 수 있음
  4. 튜닝 통해 모델의 일반화 성능 개선 가능

* weight
  각 뉴런에 대한 가중치
  가중치 높음 : 입력 뉴런이 출력에 미치는 영향이 큼
  가중치 낮음 : 입력 뉴런이 출력에 미치는 영향이 적음
* bias
  activation function을 좌우로 이동 가능(절편)
  → 결과값 조절 역할
  보통 1로 고정

parameter

❖ parameter 계산 방법

- convolution layer
  - 처음 filter 개수 x (input channel 개수 x filter matrix의 성분 개수 +1)
    - ➢ Channel개수 : 흑백(1), RGB(3)
    - ➢ Bias : +1
- dense layer
  - neuron 개수 x (input 개수 +1)
- dense에 convolution layer를 함께 사용할 때 계산할 parameter가 줄어듦
  - ➢ CNN은 parameter수가 매우 적음

hyperparameter optimization

❖ loss function

- 예측값(학습 결과)과 실제 label의 오차를 수치화 해주는 함수
  - ➢ 모델 성능의 나쁨의 지표 (낮을수록 좋음)
- loss가 최소화되는 weight, bias를 찾는 것이 중요 → loss function 선택 중요
- 낮은 확률로 예측 → 정답 & 높은 확률로 예측 → 오답인 경우 loss가 더 큼
- 일반적으로 평균 제곱 오차(Mean Squared Error), 교차 엔트로피 오차(Cross Entropy Error) 사용

  MSE : 연속형 데이터, 회귀 문제

  CEE : 범주형 데이터 분류, one-hot encoding에서 유효한 계산법
- 이진 분류 (binary_crossentropy), 다중 클래스 분류(categorical_crossentropy) 사용

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

hyperparameter optimization

❖ optimizer： loss function의 극소값을 찾는 알고리즘

gradient descent
함수의 기울기가 낮은 쪽으로 계속 이동, 극값에 이를 때까지 반복
→ 기울기를 통해 loss 최솟값을 찾는 방법

- gradient descent (경사 하강법)

  batch / stochastic / mini-batch gradient descent

- momentum

  경사하강법 + 관성

  → 기울기 = 0이 되더라도 관성이 더해져서 해당 local minimum을 global min으로 인식 x

- adagrad

  Parameter별로 다른 학습률 적용 but 극소값에 도달하기 전에 학습률이 0이 되는 문제 발생 가능

  → 변화가 큰 param은 적은 학습률, 그 반대는 높게 설정

- RMSprop

  adagrad를 보완하기 위해 기울기의 단순 누적이 아닌 최신 기울기를 더 크게 반영

- adam

  momentum + RMSprop

  가장 많이 사용

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

hyperparameter optimization

❖ search algorithm

- manual search / grid search / random search / bayesian optimization 등
- 최적의 hyperparameter 값을 찾아내는 알고리즘
- grid search

  일종의 brute-force 방식

  특정 구간 내에서 가능한 모든 조합의 hyperparameter들을 탐색하여 에러 계산 → 정확함

  cross-validation과 함께 사용

  hyperparam 수가 늘어남에 따라 연산량이 과도하게 증가할 수 있음

- random search

  grid search와 비슷

  탐색 대상을 랜덤 샘플링 통해 선정 → 정확도는 약간 떨어짐

  불필요한 반복 수행 줄임 → grid search에 비해 빠름

activation function

❖ $f(x) = Wx + b$ *(W = weight, b = bias, x = input)*

- 뉴런에 입력된 input data에 weight를 곱한 후 bias를 더함
- 비선형 함수에 입력 값을 넣어 연산한 후 다음 layer로 전달

  → 선형 함수는 여러 layer를 거쳐도 큰 의미가 없어 주로 비선형 함수 사용

  $f(x) = Cx$

  $y(x) = f(f(x))$

  $y(x) = f(f(f(x))) = C \cdot C \cdot Cx = Ax$

  학습 가능한 가중치는 생기지만 층을 쌓는 의미가 없음

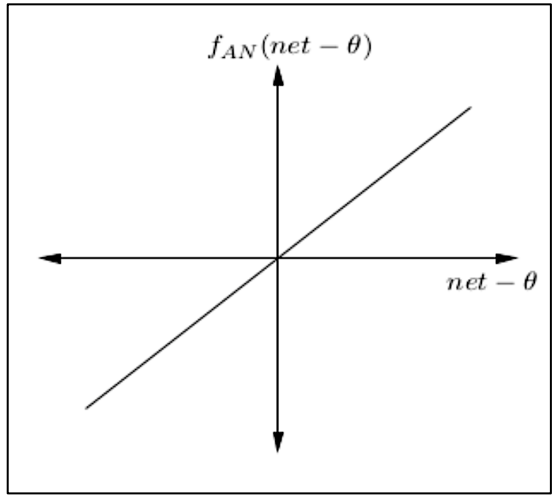
  → 이 때 사용하는 함수 = 활성화 함수

- 종류
  - Linear / ReLu / Leak ReLu / sigmoid / softmax 등 다양

default : linear
hidden layer : 주로 ReLu
output layer : sigmoid & softmax
이진 분류 : sigmoid
다중 클래스 분류 : softmax

activation function

- linear
  - 입력 뉴런 & 가중치 계산 결과를 그대로 출력 (default)



*Leak ReLu

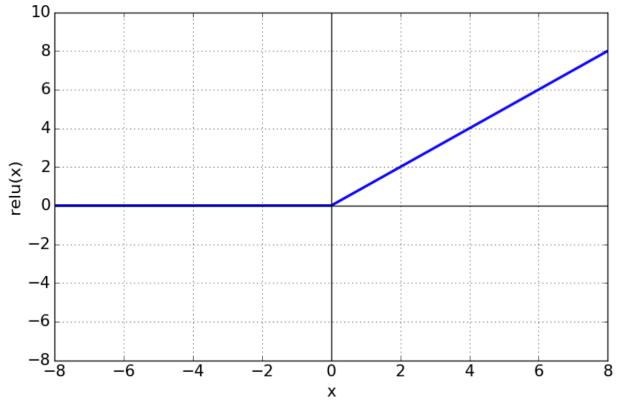
ReLu에서 $x < 0$인 경우, 뉴런이 죽는 현상을 해결하기 위함

$f(x) = max(0.01\,x, x)$: 0.01 등 매우 작은 값 사용

$x < 0$ 에서 0이 되지 않음

- ReLu (Rectified Linear Unit)
  - 주로 hidden layer에 적용
  - $f(x) = max\,(0, x)$
    - $if\ x < 0, f(x) = 0$ → 뉴런 죽음

      $x > 0, f(x)\ (linear)$ → 입력값 출력
  - 학습 속도가 빠름 (linear → 미분 연산 간단)
  - 부분적 활성화(Sparse activation) : $x > 0$
  - 에러가 전파되지 않아 더 정교하게 학습 가능




```
model.add(Dense(128, activation='relu'))
```

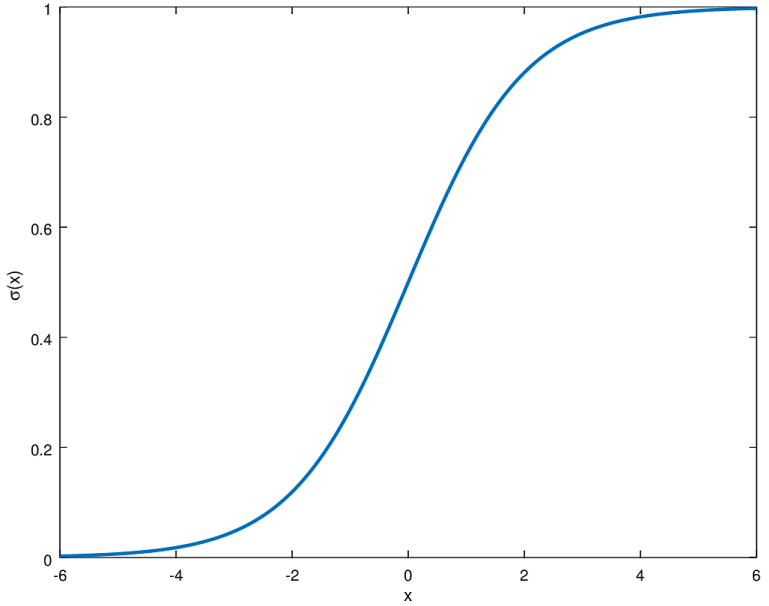
activation function

- sigmoid
  - 양극화 된 형태 → binary에 주로 쓰임 (0 or 1)
  - layer가 많을 경우, vanishing gradient 발생

    각 layer의 값을 미분하여 input layer까지 값을 전달 (역전파)

    → 0~1사이의 값을 계속 곱하므로 최종 기울기는 0에 수렴

    → 기울기 사라짐(vanishing gradient)
    - 최초 입력 값이 결과에 별 영향을 끼치지 않음
    - 최근에는 해당 취약점으로 인해 사용 비추천
  - 이러한 문제 해결 위해 hidden layer에는 ReLu & output layer에는 sigmoid 적용
    - 정확도 향상




```
model.add(Dense(1, activation='sigmoid'))
```

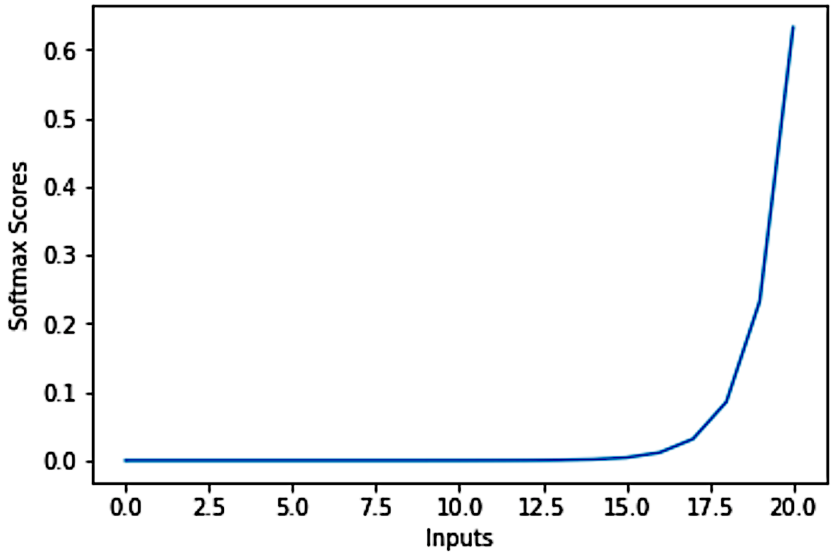
activation function

- softmax
  - output layer에서 다중 클래스 분류 위해 사용
  - 0~1 사이 값 & 각 label의 값의 합 = 1

    → input data의 class별 확률 값 통해 분류
    - Normalization 효과
  - 1 neuron per class
    - neuron의 수 = 분류할 class의 수
  - one-hot encoding 으로 표현 가능
    - categorical → binary 로 표현

      ex) 원 = 0, 세로 = 1, 가로 = 2

      0 = [1, 0, 0], 1 = [0, 1, 0], 2 = [0, 0, 1]




```
y_train2 = to_categorical(y_train)
y_val2 = to_categorical(y_val)
y_test2 = to_categorical(y_test)
```

```
model.add(Dense(3, activation='softmax'))
```

neuron 수

3개의 class로 분류하는 경우

compile

❖ 손실 함수(loss function), 최적화 함수(optimizer), 평가 기준(metrics)을 설정

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

다중 클래스 분류의 경우 사용

가장 일반적으로 사용

정확도 기준
accuracy = 1 − error rate

deep learning process

data generation → modeling → training → validation → test

training

❖ fit_generator()통해 학습

```
hist = model.fit_generator(
        train_generator,
        steps_per_epoch=50,  #150개의 data를 3(batch_size)개씩 50번
        epochs=50,
        validation_data = validation_generator,
        validation_steps = 10, #30개의 data를 3개씩 10번
        verbose = 2
        )
```

❖ train_generator → 디렉토리에서 데이터 로드

❖ epochs → 전체 데이터에 대한 학습 횟수

❖ epochs를 얼마나 해야 적절한 지 모를 경우, callback 사용

- loss가 일정 수준 이하가 되면 training 중단
- callback 함수를 통해 fit_generator()의 옵션으로 설정 가능

deep learning process

data generation → modeling → training → validation → test

validation

❖ validation data을 사용하여 모델 성능 검증

- training 과정과 동일
- 목적

  새로운 데이터에 대한 성능 예측

  최적 모델 설계 (hyperparameter tuning 통해)

❖ Cross validation(교차 검증)

- 보통 training data set이 작은 경우 사용
- k-fold cross validation 주로 사용
  - 모든 data가 validation data로 한번씩 사용
    → 특정 dataset에 overfitting 방지
  - 모든 data가 training data로 한번씩 사용
    → 정확도 향상 & underfitting 방지
  - training, validation에 많은 시간 소요

test	training	training
training	test	training
training	training	test

각 경우의 정확도의 평균으로 최종 평가

deep learning process

data generation → modeling → training → validation → test

predict

❖ training에 사용하지 않은 test data set 사용

❖ predict_generator() 사용

```
output = model.predict_generator(test_generator, steps=5)
```

Convolutional Neural Network

이미지 인식

❖ 컴퓨터의 이미지 인식

- 0 ~ 255의 RGB 값을 갖는 data matrix로 인식
- 해당 matrix의 조합을 학습시키는 것은 매우 어려움

➢ Neural Network 사용!

input data에 옳은 label을 붙여 학습하면 쉽게 분류

❖ 그냥 print하면 pixel 정보이므로 이미지를 보려면

➢ import matplotlib.pyplot as plt
plt.imshow(image)

Convolutional Neural Network (CNN)

❖ 다양한 종류의 Neural Network 중, 이미지 분류 성능이 뛰어난 모델

❖ Deep Neural Network에 비해 학습이 느림

- DNN에 filter 개수만큼의 이미지를 압축하고 convolution 수행
- loss, accuracy 향상

❖ 다층 퍼셉트론(multi-layer perceptron)의 확장 형태

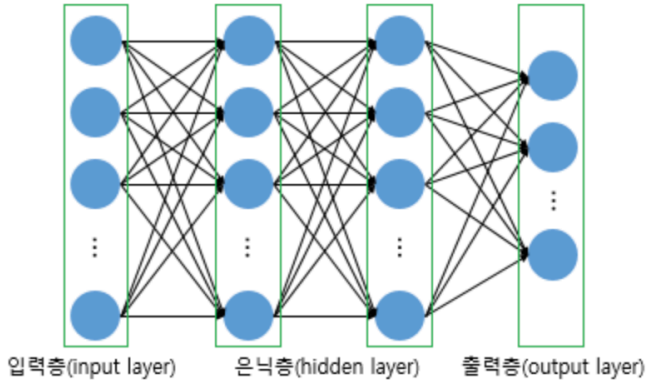
- 핵심

  1. 다중 필터 2. 가중치 공유 3. pooling

- 기존의 MLP(=MLNN)의 문제점
  - 데이터 : 이미지의 형상을 고려하지 않음 → 회전, 변형에 대해 새로운 학습데이터 필요
  - 학습시간 : parameter 개수 (= 계산 수) 증가 등

➢ 이러한 문제를 해결하기 위해 convolution layer 사용

다층 퍼셉트론 구조



Convolutional Neural Network (CNN)

❖ 다음과 같은 layer들을 쌓아 구성

convolution layer
dropout layer
pooling layer
flatten layer
dense layer

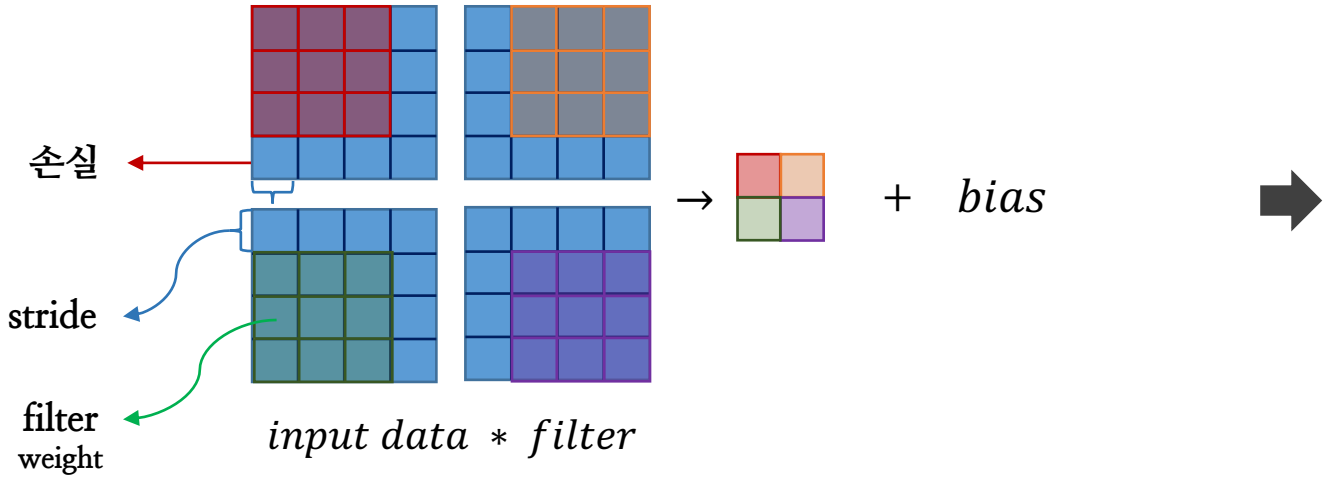
```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(24,24,3)))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Dropout(0.2))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dense(3, activation='softmax'))
```

convolution layer

❖ Convolution(∗)

- 이미지를 filter에 통과(filter matrix와 image matrix의 행렬 곱)시켜 이미지 변형
- 한 이미지에 대해 filter 개수만큼의 데이터가 생성, 이를 기반으로 학습 → 성능 good
- 주변 이미지가 손실될 수 있음 → padding 적용

ex) input : (4,4)픽셀, filter(3,3) 〉〉 (1,1) 4개로 변환



특징 맵(Feautre Map)

conv layer의 입출력은 3차원 이미지 (matrix 형태)

→ 형상 유지

Stride

input data에 filter 적용 시 이동할 간격 조절

output의 크기를 조절하기 위함

작은 값이 더 좋음 (보통 1)

➢ 하나의 filter가 이미지를 순회하며 동일한 가중치 적용(parameter 공유) → 학습할 parameter 매우 적음

➢ Convolution layer → 해당 영역의 특징 추출

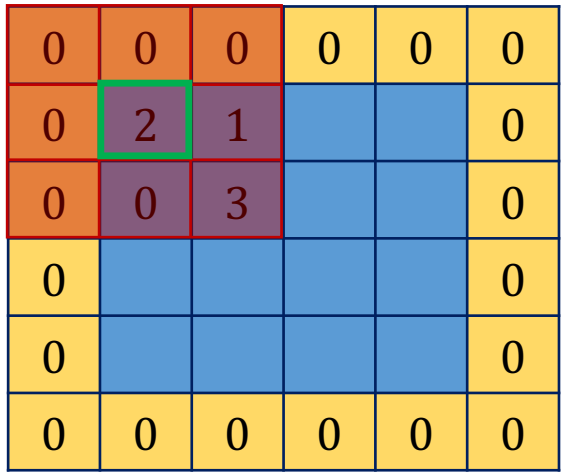
convolution layer

❖ zero-padding

- 이미지의 손실을 막기 위함 → 입력데이터의 크기 유지
- 각 픽셀별로 filter와 image의 (3,3) 정사각행렬 곱셈

→ 이미지의 상하좌우 극단에 있는 픽셀들은 해당 픽셀을 중심으로 하는 (3,3)행렬이 없음

→ convolution 수행 전에 해당 픽셀의 주변을 0으로 채움 (hyper parameter 설정)



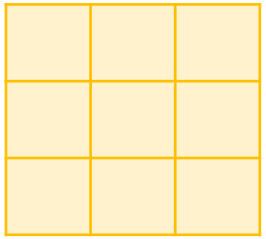
convolution layer

```
(Conv2D(32, kernel_size=(3, 3), activation='relu', input_shape=(24,24,3)))
```

convolution filter 개수

주로 32의 배수

filter size (행, 열)



주로 (3,3) 사용

활성화 함수

주로 hidden layer에 사용

빠른 fitting 속도

첫 layer에서만 정의
input data의 (행, 열, 채널 수)

흑백 = 1
컬러 = 3

*padding= 'valid' or 'same'

- valid : 유효 영역만 출력(input size 〉 output size)
- same : input size = output size

pooling layer

❖ filter 수만큼의 압축된 이미지 생성

- 데이터를 압축하여 추출한 특징 강화
- 데이터 크기 감소

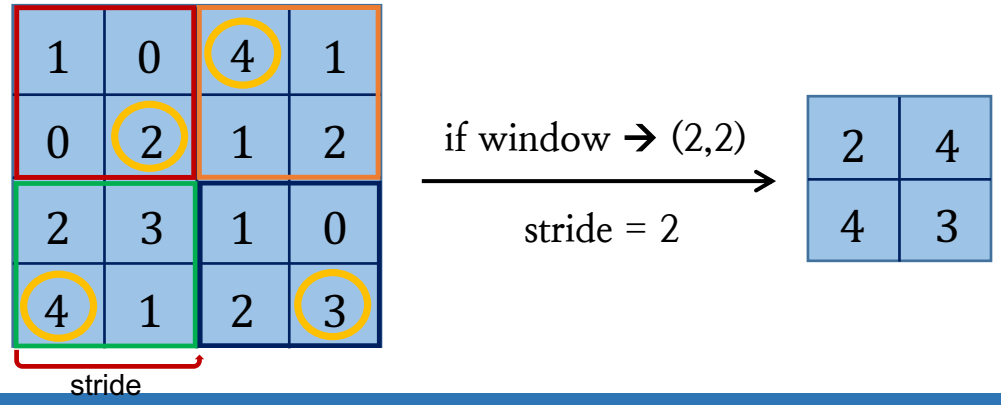
❖ convolution은 pooling과 함께 사용될 때 더 강력

- 하나의 이미지에 convolution & pooling을 반복 = 유의미한 특징 정보 추출 및 강화

❖ 주로 max-pooling 사용 (속도가 빠름)

- 픽셀을 적당한 크기의 matrix로 잘라서 해당 영역에서의 최대값 추출

❖ window size와 stride는 같은 값으로 설정

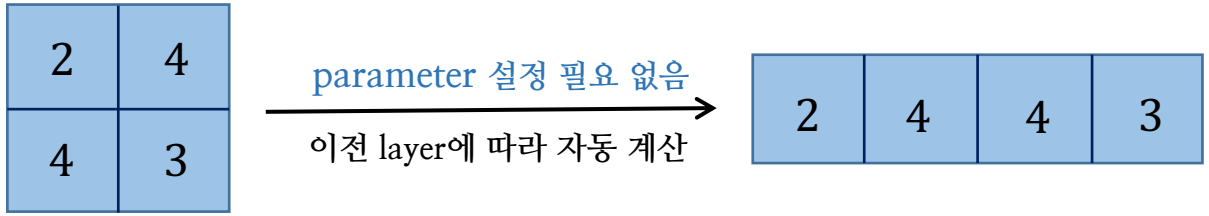



```
model.add(MaxPooling2D(pool_size=(2, 2)))
```

flatten layer

❖ convolution & pooling 통해 추출된 주요 특징들을 1차원 벡터로 변환

- 이전 layer의 output들을 1차원으로 이어서 해당 벡터를 학습




```
model.add(Flatten())
```

dense layer

❖ 모든 입력과 출력을 연결하는 전결합층

❖ 각 뉴런은 가중치 포함

- 뉴런의 개수 = 가중치의 수

  ex) 입력 뉴런 4개, 출력 뉴런 8개 → 총 32개의 뉴런, 32개의 가중치

```
model.add(Dense(128, activation='relu'))
model.add(Dense(3, activation='softmax'))
```

출력 뉴런의 수

output layer의 출력 뉴런 수 = class 수

다중 클래스 분류

...

❖ training → validation (hyperparameter tuning, 최적화) → test 과정을 거쳐 설계한 모델 통해 분류

Q & A