KAIST Include 동아리 스터디

AlphaGo와 AlphaGo Zero를 만들며 익히는 딥러닝 및 강화학습

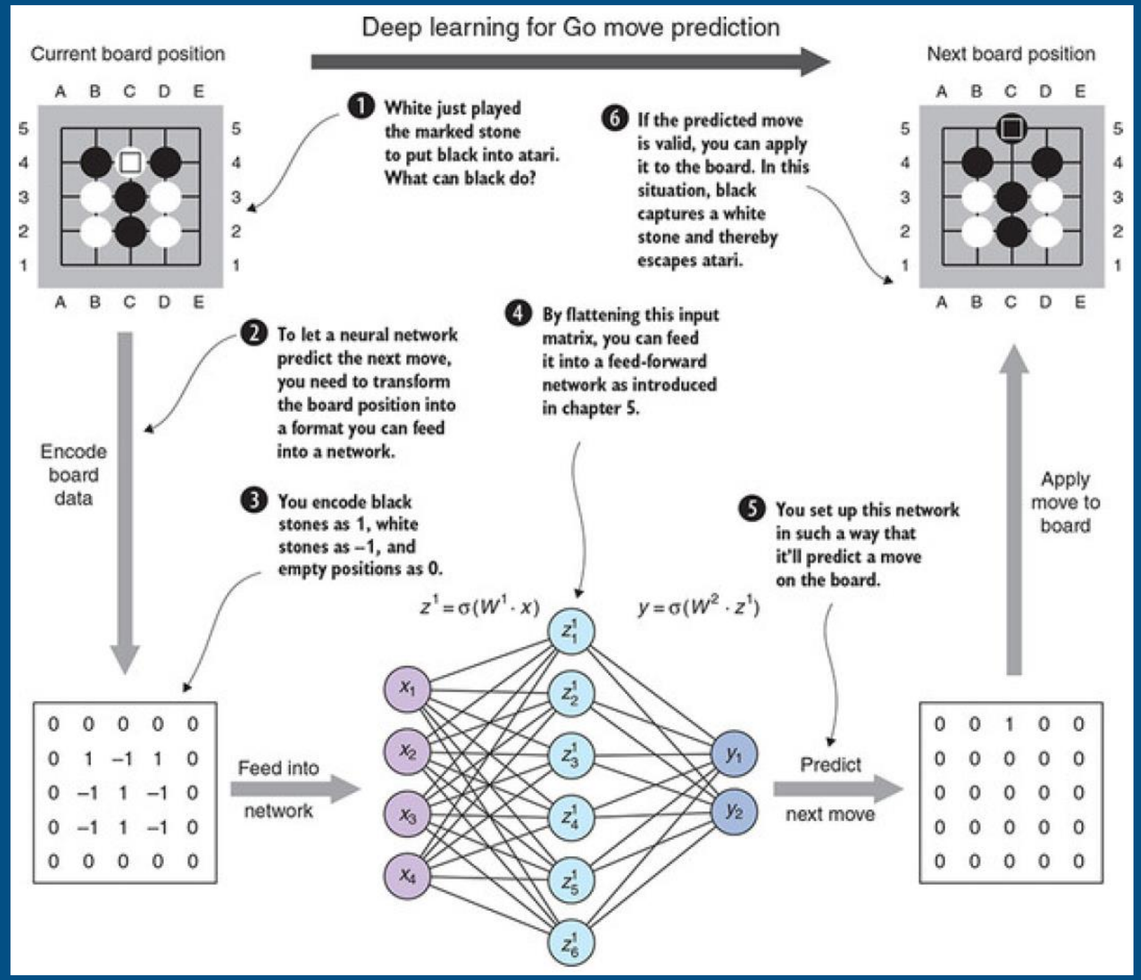
Chris Ohk
utilForever@gmail.com

공지사항

- 5/1, 5/15 : 개인 사정으로 인한 스터디 휴강
- 보충 스터디를 언제 하면 좋을까?
  - 1안 : 5/19, 6/6
  - 2안 : 7/3, 7/10

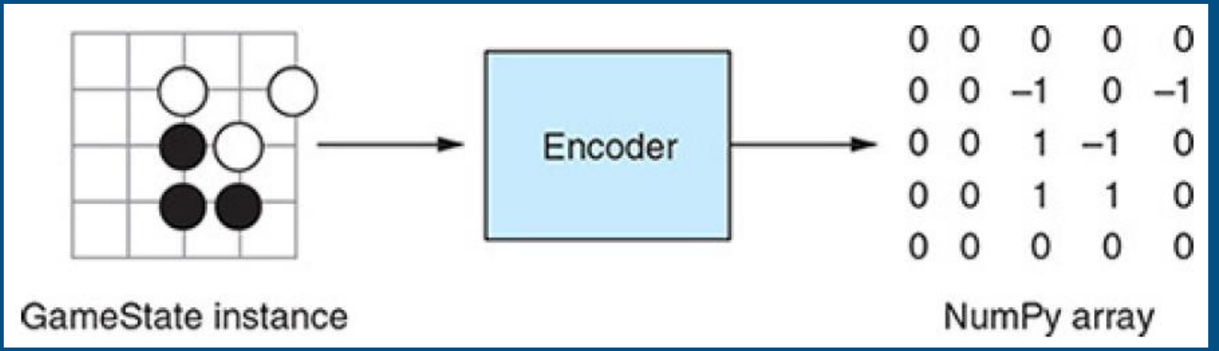
들어가며

- 딥러닝을 사용해서 바둑에서 다음 수를 예측하는 방법



신경망용 바둑경기 변환

- 앞에서 `Player`, `Board`, `GameState` 등을 Python 클래스로 만들어 바둑의 모든 요소를 나타냈었다. 하지만 신경망 같은 수학 모델은 벡터나 행렬 같은 수학 객체만을 다루므로 고차원 객체에는 적용하기 어렵다.
  → `Encoder` 클래스를 만들어서 게임 객체를 수학적 형태로 변환한다.



신경망용 바둑경기 변환

- 바둑 대국 상태를 변환하는 추상 Encoder 클래스

```python
class Encoder:
  def name(self):
    raise NotImplementedError()

  def encode(self, game_state):
    raise NotImplementedError()

  def encode_point(self, point):
    raise NotImplementedError()

  def decode_point_index(self, index):
    raise NotImplementedError()

  def num_points(self):
    raise NotImplementedError()

  def shape(self):
    raise NotImplementedError()
```

신경망용 바둑경기 변환

- 이름으로 바둑판 변환기 참조

```python
import importlib

def get_encoder_by_name(name, board_size):
  if isinstance(board_size, int):
    board_size = (board_size, board_size)
  module = importlib.import_module('dlgo.encoders.' + name)
  constructor = getattr(module, 'create')
  return constructor(board_size)
```

신경망용 바둑경기 변환

- 1차 평면 바둑판 변환기로 게임 상태 변환하기
  - 이번 차례에 돌을 놓을 선수 = 1
    상대 = -1
    빈 점 = 0

```python
import numpy as np

from dlgo.encoders.base import Enocder
from dlgo.goboard import Point

class OnePlaneEncoder(Encoder):
  def __init__(self, board_size):
    self.board_width, self.board_height = board_size
    self.num_planes = 1

  def name(self):
    return 'oneplane'

  def encode(self, game_state):
    board_matrix = np.zeros(self.shape())
    next_player = game_state.next_player
    for r in range(self.board_height):
      for c in range(self.board_width):
        p = Point(row=r+1, col=c+1)
        go_string = game_state.board.get_go_string(p)
        if go_string is None:
          continue
        if go_string.color == next_player:
          board_matrix[0, r, c] = 1
        else:
          board_matrix[0, r, c] = -1
    return board_matrix
```

신경망용 바둑경기 변환

- 1차 평면 바둑판 변환기에서 점을 변환 및 역변환하는 방법

```
def encode_point(self, point):
  return self.board_width * (point.row - 1) + (point.col - 1)

def decode_point_index(self, index):
  row = index // self.board_width
  col = index % self.board_width
  return Point(row=row+1, col=col+1)

def num_points(self):
  return self.board_width * self.board_height

def shape(self):
  return self.num_planes, self.board_height, self.board_width
```

MCTS 게임을 훈련 데이터로 만들기

- 머신러닝을 바둑 대국에 적용하려면 훈련 데이터셋이 필요하다.
  - 고수들의 대국 데이터를 가져와 사용하는 방법
  - 대국 데이터를 직접 생성해서 사용하는 방법
- 여기서는 MCTS 봇을 사용해서 대국 기록을 생성한다.

MCTS 게임을 훈련 데이터로 만들기

- MCTS용 대국 데이터 생성 기능 불러오기

```
import argparse
import numpy as np

from dlgo.encoders import get_encoder_by_name
from dlgo import goboard_fast as goboard
from dlgo import mcts
from dlgo.utils import print_board, print_move
```

MCTS 게임을 훈련 데이터로 만들기

- MCTS 경기 생성

```python
def generate_game(board_size, rounds, max_moves, temperature):
  boards, moves = [], []
  encoder = get_encoder_by_name('oneplane', board_size)
  game = goboard.GameState.new_game(board_size)
  bot = mcts.MCTSAgent(rounds, temperature)

  num_moves = 0
  while not game.is_over():
    print_board(game.board)
    move = bot.select_move(game)
    if move.is_play:
      boards.append(encoder.encode(game))

      move_one_hot = np.zeros(encoder.num_points())
      move_one_hot[encoder.encode_point(move_point)] = 1
      moves.append(move_one_hot)
      print_move(game.next_player, move)
      game = game.apply_move(move)
      num_moves += 1
      if num_moves > max_moves:
        break

  return np.array(boards), np.array(moves)
```

MCTS 게임을 훈련 데이터로 만들기

- MCTS 게임 데이터 생성용 핵심 코드

```python
def main():
  parser = argparse.ArgumentParser()
  parser.add_argument('--board-size', '-b', type=int, default=9)
  parser.add_argument('--rounds', '-r', type=int, default=1000)
  parser.add_argument('--temperature', '-t', type=float, default=0.8)
  parser.add_argument('--max-moves', '-m', type=int, default=60, help='Max moves per game.')
  parser.add_argument('--num-games', '-n', type=int, default=10)
  parser.add_argument('--board-out')
  parser.add_argument('--move-out')

  args = parser.parse_args()
  xs = []
  ys = []

  for i in range(args.num_games):
    print('Generating game %d/%d...' % (i + 1, args.num_games))
    x, y = generate_game(args.board_size, args.rounds, args.max_moves, args.temperature)
    xs.append(x)
    ys.append(y)

  x = np.concatenate(xs)
  y = np.concatenate(ys)

  np.save(args.board_out, x)
  np.save(args.move_out, y)
```

MCTS 게임을 훈련 데이터로 만들기

- MCTS 게임 데이터 생성 방법

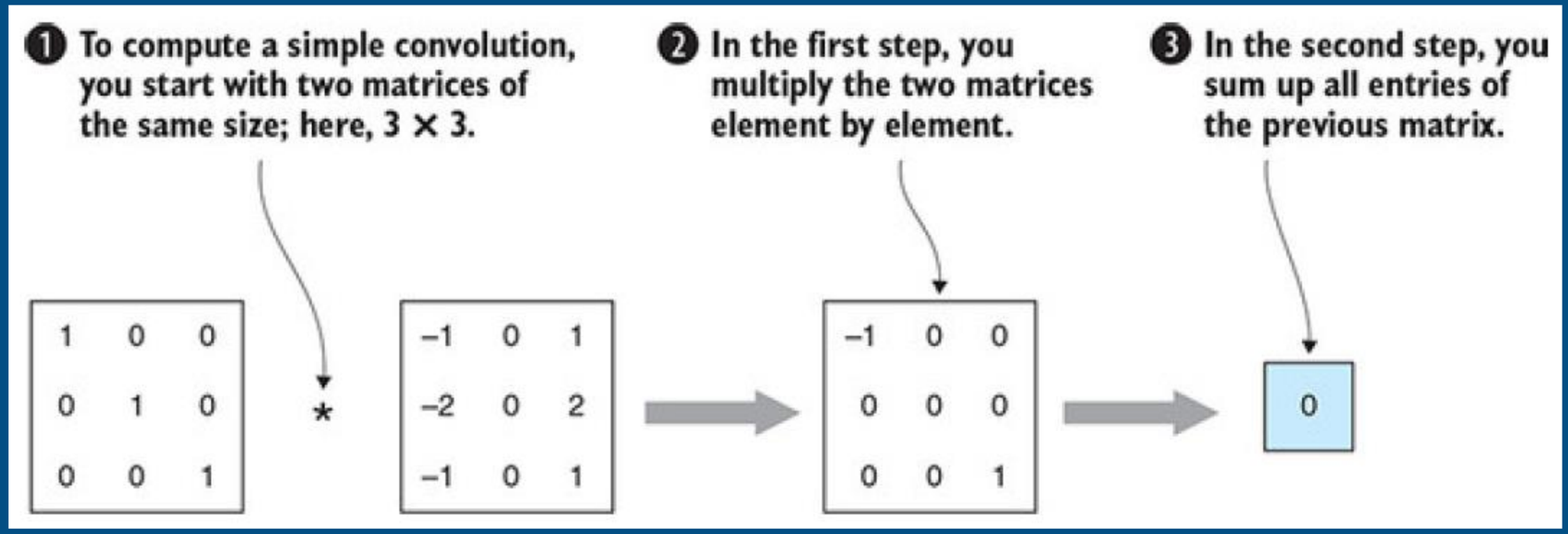
```
python generate_mcts_games.py -n 20
--board-out features.npy --move-out labels.npy
```

합성곱 신경망으로 공간 분석하기

- 바둑에서는 돌의 특정 지역적 패턴이 반복되는 걸 종종 볼 수 있다. 사람처럼 의사 결정을 하려면 바둑 AI 역시 많은 공간적 배열을 파악해야 한다.
- **합성곱 신경망 (Convolutional Neural Network; CNN)**이라는 특정 유형의 신경망은 이런 공간적 관계를 찾아내는 데 특화되어 있다.

합성곱 신경망으로 공간 분석하기

- 합성곱 역할에 대한 직관적 이해
  - 동일한 두 행렬의 단순 합성곱 계산 과정
    1. 두 행렬을 원소별로 곱한다.
    2. 결과 행렬의 모든 값을 더한다.

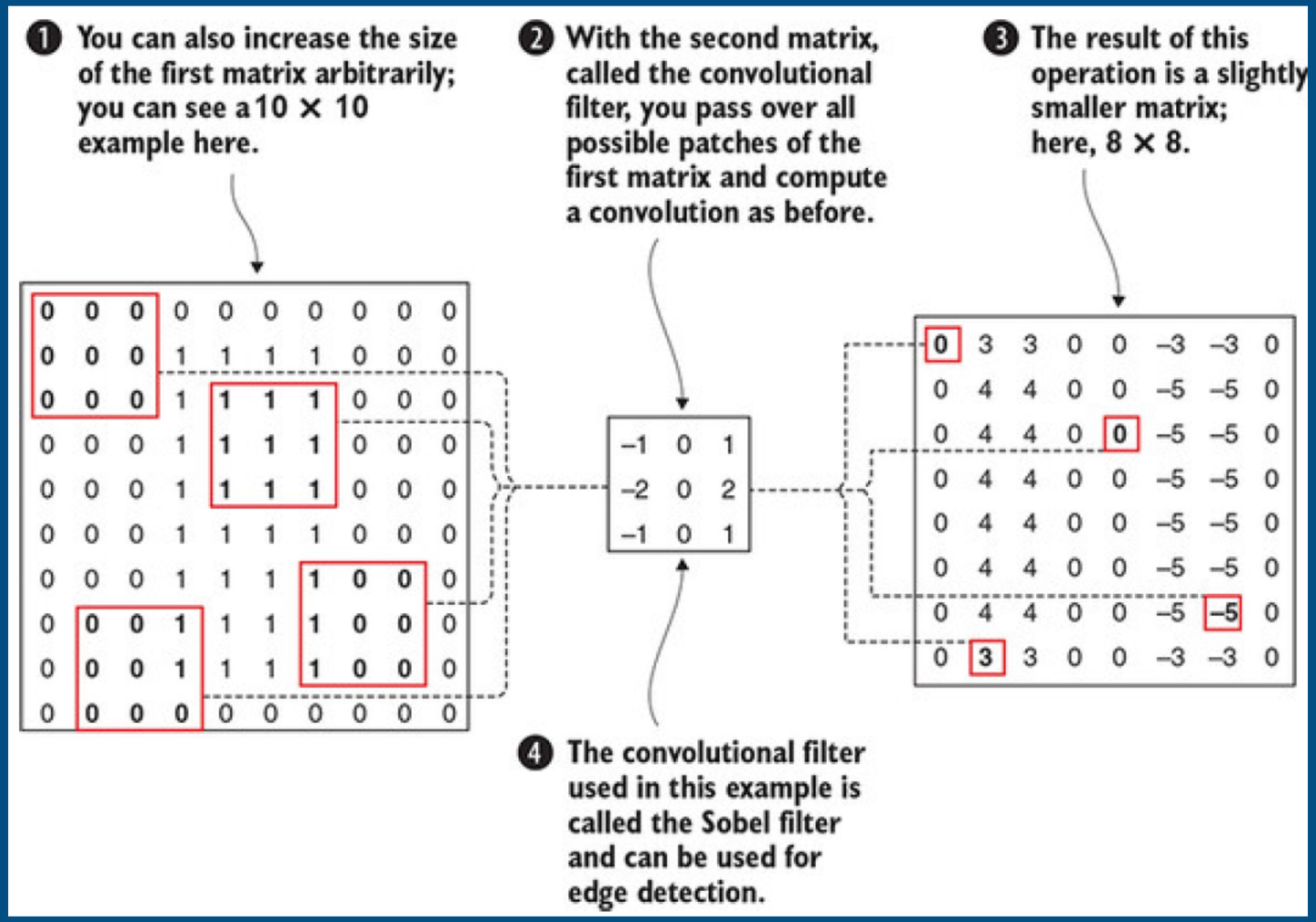


합성곱 신경망으로 공간 분석하기

- 합성곱 역할에 대한 직관적 이해
  - 똑같은 두 행렬 대신 두번째 행렬을 고정한 후 첫번째 행렬의 크기를 임의로 확대하자.
    이때 첫번째 행렬을 **입력 이미지**라 하고 두번째 행렬을 **합성곱 커널** 또는 **커널**이라 하자.
  - 커널은 입력 이미지보다 작으므로 입력 이미지의 많은 조각에 대한 단순 합성곱을 구한다.

합성곱 신경망으로 공간 분석하기

- 합성곱 역할에 대한 직관적 이해



합성곱 신경망으로 공간 분석하기

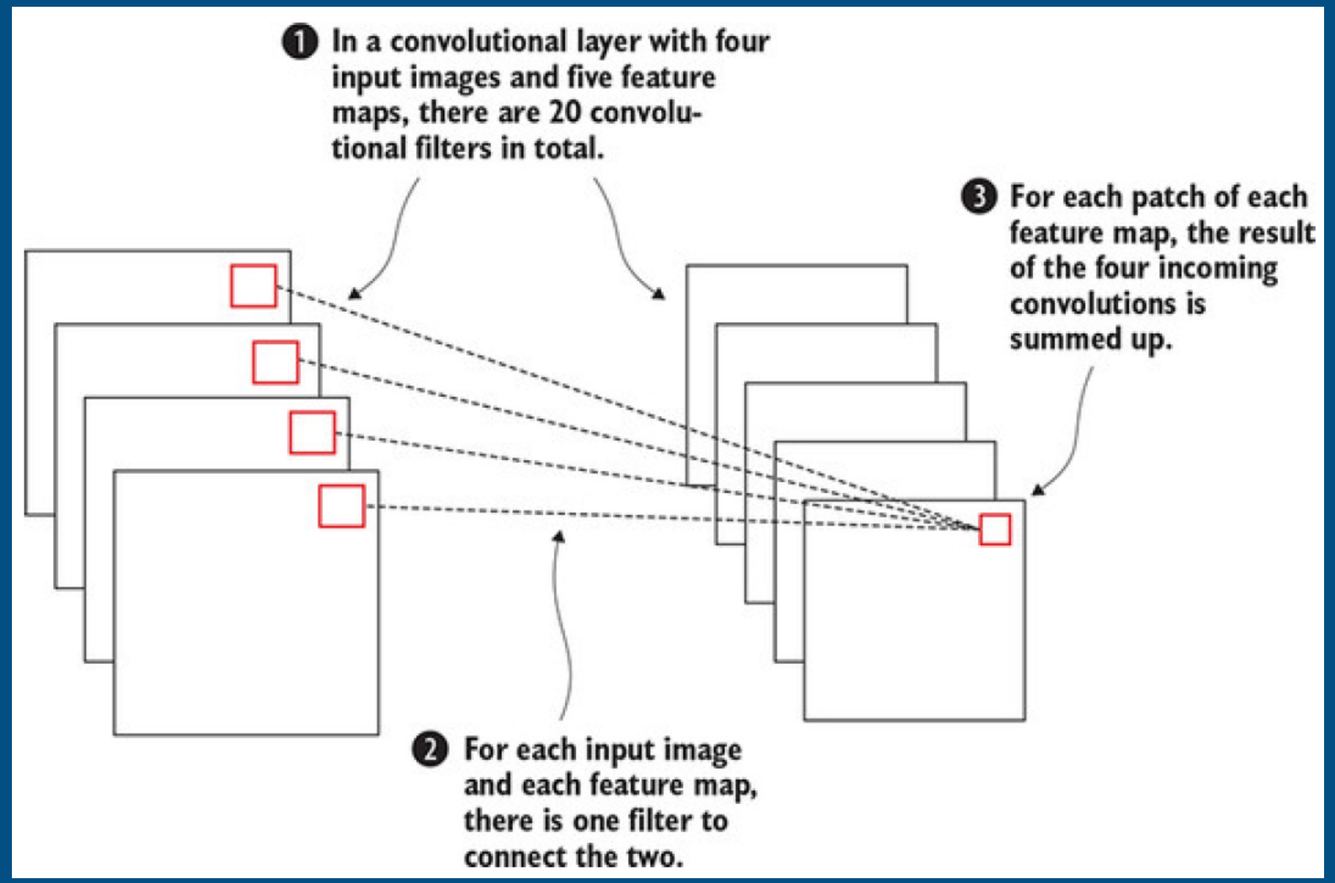
- 합성곱 역할에 대한 직관적 이해
  - 입력 이미지는 가운데 4×8 사각형이 1이고 주변이 0으로 채워진 10×10 행렬이다.
    커널 행렬의 첫번째 열(-1, -2, -1)은 세번째 열(1, 2, 1)을 음의 값으로 바꾼 것이고
    가운데 열은 모두 0이다. 따라서 다음의 내용은 모두 참이다.
    - 이 커널을 모든 픽셀값이 동일한 입력 이미지의 3×3 패치에 적용할 때마다 합성곱의 출력은 0이 된다.
    - 이 합성곱 커널을 좌측 열의 값이 우측 열의 값보다 큰 이미지 패치에 적용하면 합성곱 값은 음의 값이 된다.
    - 이 합성곡 커널을 우측 열의 값이 좌측 열의 값보다 큰 이미지 패치에 적용하면 합성곱 값은 양의 값이 된다.
  - 합성곱 커널은 입력 이미지의 세로 모서리 감지에 쓰인다.
    물체의 왼쪽 모서리는 양의 값을 가질 것이고, 오른쪽 모서리는 음의 값을 가질 것이다.
  - 흥미로운 사실은 합성곱을 사용해서 이미지 데이터에서 정확히 바둑 수를 예측하는 데
    필요한 정보를 뽑아낼 수 있다는 것이다.

합성곱 신경망으로 공간 분석하기

- 합성곱 역할에 대한 직관적 이해
  - 지금까지는 이미지 하나에 합성곱 커널 하나를 적용하는 예를 설명했다.
    보통 여러 커널을 여러 이미지에 적용해서 여러 출력 이미지를 만드는 게 유용하다.
  - 입력 이미지가 4개고 커널도 4개라고 해보자. 그러면 각 이미지에 대한 합성곱을 더해 하나의 출력 이미지에 도달할 수 있다. → **특징 지도 (Feature Map)**
  - 예를 들어, 특징 지도가 하나가 아니라 5개 필요하다면 입력 이미지 당 하나의 커널 대신 5개의 커널을 정의해야 한다. 입력 이미지 n개에 합성곱 커널 $n \times m$개를 특징 지도 m개로 연결하는 걸 **합성곱층 (Convolutional Layer)**이라고 한다.
    - 합성곱층과 밀집층만으로 이루어진 신경망을 **합성곱 신경망**이라고 한다.

합성곱 신경망으로 공간 분석하기

- 합성곱 역할에 대한 직관적 이해
  - 합성곱층(Convolutional Layer)



합성곱 신경망으로 공간 분석하기

- TensorFlow 2로 합성곱 신경망 만들기
  - 합성곱 신경망에 사용할 바둑 데이터를 가져와서 전처리하기

```python
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Conv2D, Flatten

np.random.seed(123)
X = np.load('../generated_games/features-200.npy')
Y = np.load('../generated_games/labels-200.npy')

samples = X.shape[0]
size = 9
input_shape = (size, size, 1)

X = X.reshape(samples, size, size, 1)

train_samples = int(0.9 * samples)
X_train, X_test = X[:train_samples], X[train_samples:]
Y_train, Y_test = Y[:train_samples], Y[train_samples:]
```

합성곱 신경망으로 공간 분석하기

- TensorFlow 2로 합성곱 신경망 만들기
  - TensorFlow 2로 바둑 데이터에 대한 간단한 합성곱 신경망 만들기

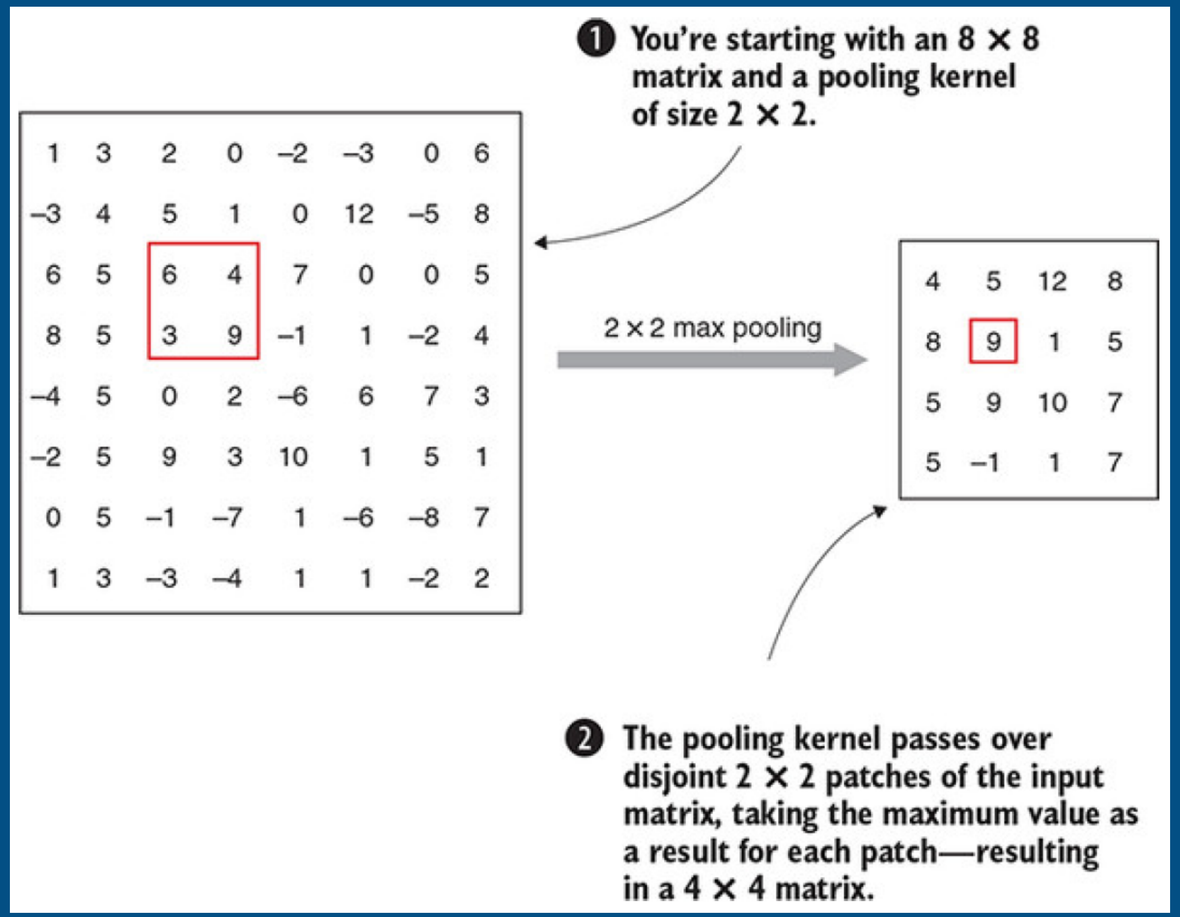
```
model = Sequential()
model.add(Conv2D(filters=48,
                 kernel_size=(3, 3),
                 activation='sigmoid',
                 padding='same',
                 input_shape=input_shape))

model.add(Conv2D(48, (3, 3), padding='same', activation='sigmoid'))

model.add(Flatten())

model.add(Dense(128, activation='sigmoid'))
model.add(Dense(size * size, activation='sigmoid'))
model.summary()
```

합성곱 신경망으로 공간 분석하기

- 풀링층을 사용한 공간 감소
  - 합성곱층을 사용하는 딥러닝 애플리케이션에서 대부분 **풀링(Pooling)** 기법을 사용한다. 풀링을 사용해서 이미지를 줄이고 앞 층에서 사용하는 뉴런 수를 감소시킬 수 있다.
  - 풀링 개념은 설명하기 쉽다. 이미지의 패치를 단일 값으로 묶거나 풀링해서 이미지를 다운샘플링하면 된다. 예를 들어, 최댓값만 남기거나 평균값을 구해서 사용할 수 있다.

합성곱 신경망으로 공간 분석하기

- 풀링층을 사용한 공간 감소
  - 최댓값 풀링(Max Pooling)



합성곱 신경망으로 공간 분석하기

- 풀링층을 사용한 공간 감소
  - 풀 크기 (2, 2)의 최댓값 풀링층을 TensorFlow 2 모델에 추가

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

바둑 수 확률 예측하기

- 지금까지 활성화 함수로 로지스틱 시그모이드 함수만 사용했다. 또한 손실 함수로 평균제곱오차만 사용했다. 이 두 함수는 처음 사용하기에 매우 좋지만 바둑 수 예측에서는 다소 적합하지 않다.
- 바둑 수 예측에 있어서 궁극의 질문은 '각 가능한 수가 다음 수가 될 가능성은 어느 정도일까?'다. 즉, 모든 가능한 수의 **확률분포**를 예측하길 원한다.
  - 시그모이드 활성화 함수로는 어려울 수 있다.
  - 대신 마지막 층에서 확률 예측에 사용할 수 있는 **소프트맥스 활성화 함수**를 사용한다.

바둑 수 확률 예측하기

- 마지막 층에서 소프트맥스(Softmax) 활성화 함수 사용
  - 소프트맥스 활성화 함수는 로지스틱 시그모이드 σ를 간단히 일반화한 것이다.
  - 벡터 $x = (x_1, \dots, x_l)$의 소프트맥스를 구하려면 우선 각 부분에 지수 함수를 취해서 e^{x_i}를 구한다. 그런 다음 모든 값을 합해 각 값을 **정규화**한다.

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{l} e^{x_j}}$$

  - 정의에 따르면 소프트맥스 함수의 각 성분은 음이 아니며 더해서 1이 되는 수이므로 소프트맥스 함수가 확률을 구해준다고 볼 수 있다.

바둑 수 확률 예측하기

- 마지막 층에서 소프트맥스(Softmax) 활성화 함수 사용
  - Python으로 소프트맥스 활성화 함수 정의하기

```python
import numpy as np

def softmax(x):
  e_x = np.exp(x)
  e_x_sum = np.sum(e_x)
  return e_x / e_x_sum

x = np.array([100, 100])
print(softmax(x))
```

바둑 수 확률 예측하기

- 마지막 층에서 소프트맥스(Softmax) 활성화 함수 사용
  - 풀 크기 (2, 2)의 최댓값 풀링층을 TensorFlow 2 모델에 추가

```
model.add(Dense(9*9, activation='softmax'))
```

바둑 수 확률 예측하기

- 분류 문제에서의 교차 엔트로피 손실
  - MSE의 문제
    - 수 예측 사례를 분류 문제로 만들어서 가능한 9×9 경우 수 중 하나만 맞는 식으로 사용했던 것을 떠올려보자.
      맞으면 1로, 아니면 0으로 라벨을 붙였다. 각 경우에 대한 예측값은 항상 0과 1 사이의 값으로 나온다.
      MSE가 예측과 라벨의 차이를 제곱하는 것을 보면 범위를 0과 1 사이로 제약하는 건 아무런 소용이 없다.
      사실 MSE는 출력의 범위가 연속적인 회귀 문제에서 가장 유용하다. (예를 들어, 사람의 키를 예측하는 경우)
      지금 문제에서 예측값과 실제 출력값의 절대적 차이는 1이다.

바둑 수 확률 예측하기

- 분류 문제에서의 교차 엔트로피 손실
  - MSE의 문제
    - 또 다른 문제는 모든 예측값에 동일한 벌점을 부과한다는 것이다. 이 문제는 값이 1인 참 하나만 예측하면 된다.
      예를 들어, 옳은 수 하나에 0.6점을 주고 나머지 중 하나에는 0.4점, 그 외에는 모두 0점을 주는 모델이 있다고 하자.
      이 경우 MSE는 $(1-0.6)^2+(0-0.4)^2=2\times0.4^2=0.32$가 될 것이다.
      예측은 맞겠지만 두 0이 아닌 예측값에 모두 0.16 정도의 동일한 손실값이 나온다.
      더 작은 값을 동일하게 보여주는 것이 정말로 필요할까?
      이 상황을 옳은 수에 0.6이 나오고 다른 두 수에 0.2가 나오는 경우와 비교해 보자.
      이 경우 MSE는 $(1-0.6)^2+2\times(0.2)^2=0.24$다. 앞의 경우에 비해 확연히 낮은 값이 나온다.
      만약 0.4가 더 정확한 값이라면 **이 값도 다음 수의 후보가 됨직한** 강한 수인 것일까?
      이 값에 손실 함수로 불이익을 주어야 할까?

바둑 수 확률 예측하기

- 분류 문제에서의 교차 엔트로피 손실
  - **교차 엔트로피 (Cross Entropy)** 손실 함수
    - 라벨 $\hat{y}$와 모델 예측값 y에 대한 교차 엔트로피 손실 함수는 다음과 같다.

$$-\sum_{i} \hat{y}_i \log(y_i)$$

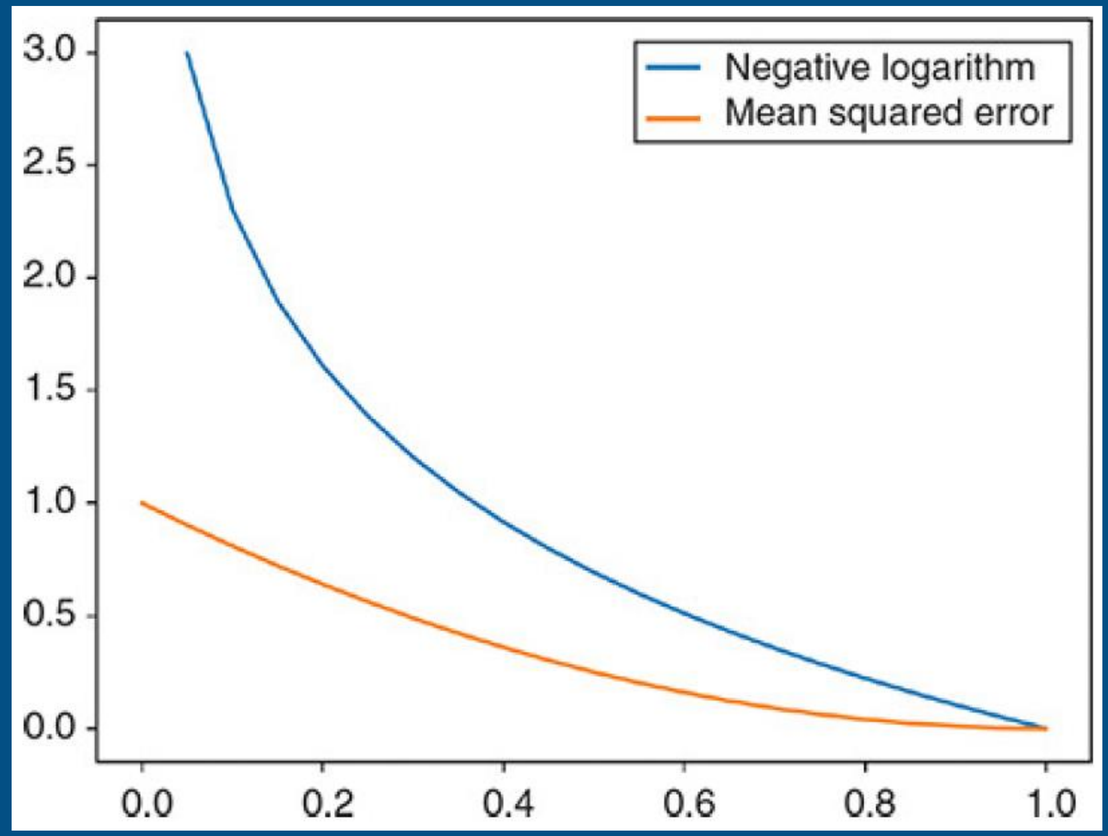
    - 많은 항목을 더해서 계산량이 많을 거 같지만 우리 예제의 경우 이 식은 $\hat{y}_i = 1$인 단일항으로 확 줄어든다.
    - 교차 엔트로피 오차는 $\hat{y}_i = 1$인 i에 대해 $-\log(y_i)$다.

바둑 수 확률 예측하기

- 분류 문제에서의 교차 엔트로피 손실
  - **교차 엔트로피 (Cross Entropy)** 손실 함수
    - 교차 엔트로피 손실은 라벨이 1인 항에만 벌점을 부과하므로 다른 값의 분포에는 직접 영향을 끼치지 않는다. 특히 0.6 확률로 정확한 다음 수를 예측하는 시나리오에서는 다른 한 수의 가능도를 얼마로 하든 차이가 없다. 이때 교차 엔트로피 손실값은 $-\log(0.6) = 0.51$이 된다.
    - 교차 엔트로피 손실값은 $[0, 1]$ 범위에 존재한다. 만약 모델이 특정 수의 확률이 0이라고 예측한다면 뭔가 잘못된 것이다. 우리는 $\log(1) = 0$이고 0과 1 사이의 x에 대해 x가 0에 가까워질수록 $-\log(x)$는 무한대에 가까워지면서 빠른 속도로 커진다는 걸 알고 있다.
    - 게다가 x가 1에 가까워질수록 MSE는 더 빨리 떨어지게 되는데 신뢰도가 낮은 예측일수록 훨씬 적은 손실을 입게 된다는 것을 의미한다.

바둑 수 확률 예측하기

- 분류 문제에서의 교차 엔트로피 손실
  - 교차 엔트로피(Cross Entropy) 손실 함수



바둑 수 확률 예측하기

- 분류 문제에서의 교차 엔트로피 손실
  - 교차 엔트로피와 MSE를 구분하는 또 다른 중요한 점은
    확률적 경사하강법(SGD) 학습 중의 행동이다.
    - MSE 경사가 갱신되는 정도는 예측값이 점점 커질수록 작아지며 학습 속도가 줄어든다.
    - 이에 비해 교차 엔트로피 손실에서는 SGD 상의 이런 감속이 일어나지 않으며,
      파라미터 갱신은 예측값과 실젯값 간의 차이에 비례한다.

바둑 수 확률 예측하기

- 분류 문제에서의 교차 엔트로피 손실
  - 교차 엔트로피를 사용하는 TensorFlow 2 모델 컴파일

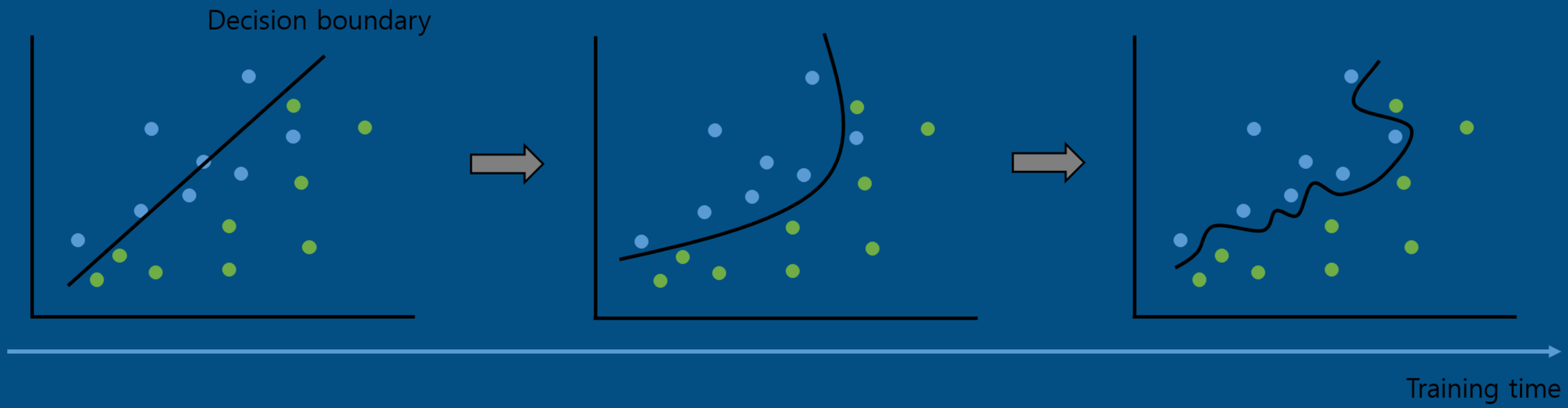
```
model.compile(loss='categorical_crossentropy'...)
```

드롭아웃과 정류 선형 유닛

- 지금까지 2~4개 층을 갖는 신경망을 만들었다. 여기서 생각해 보자.
  층을 계속 추가한다고 해서 무조건 성능이 향상될까?
  - 신경망을 더 깊게 만들수록 모델의 파라미터 수가 증가한다.
  - 여기에 넣어줘야 할 데이터양도 증가한다.
  - 가장 문제가 되는 것은 **과적합(Overfitting)**!

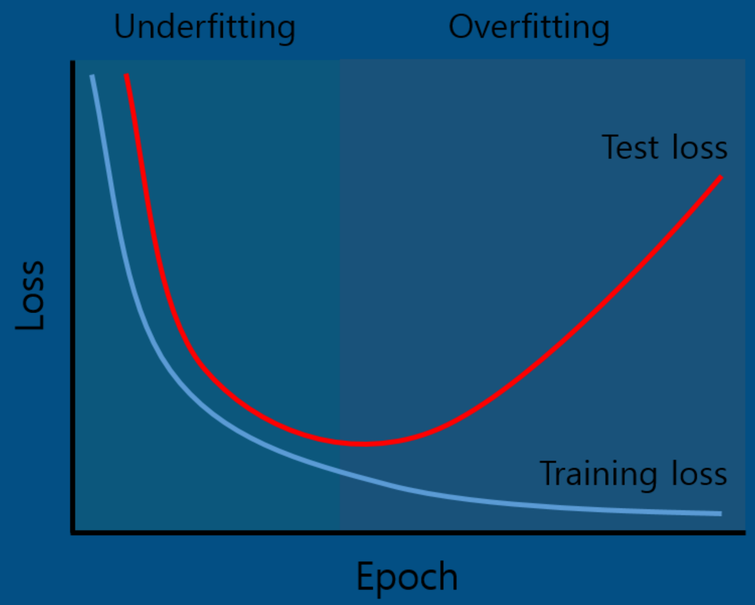
드롭아웃과 정류 선형 유닛

- 과적합(Underfitting) vs 과대적합(Overfitting)



드롭아웃과 정류 선형 유닛

- 과적합(Underfitting) vs 과대적합(Overfitting)

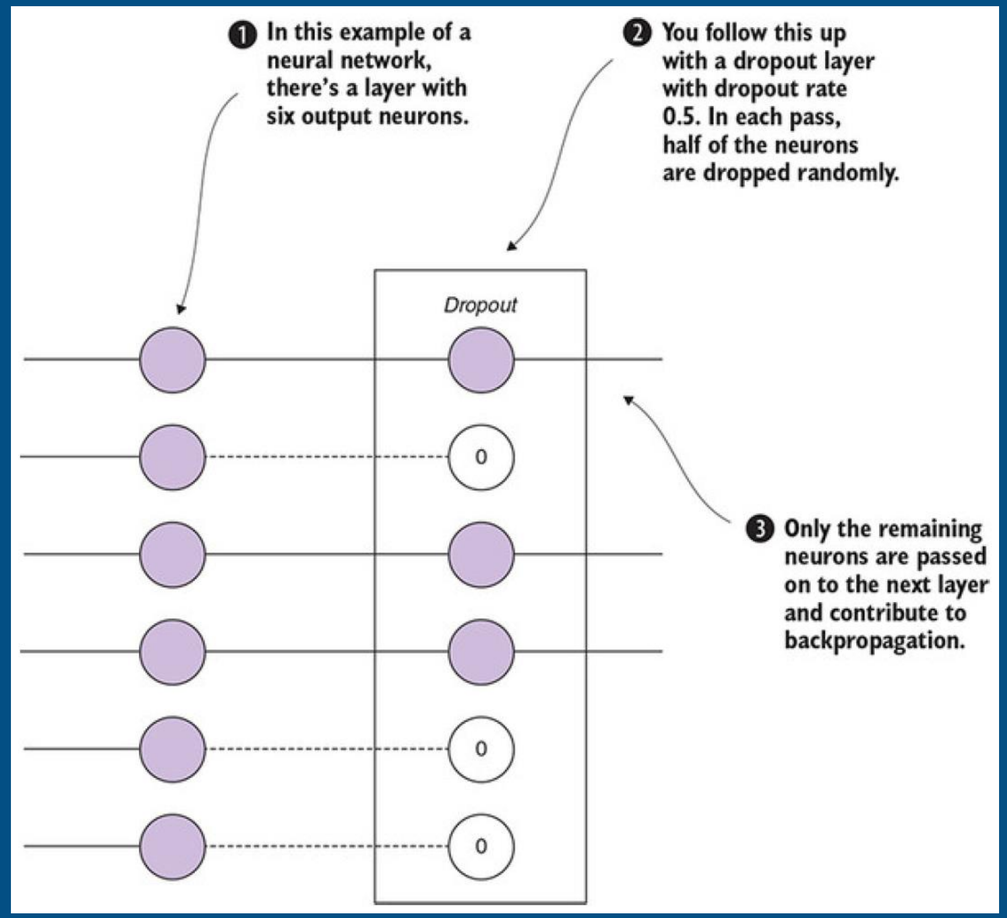


드롭아웃과 정류 선형 유닛

- 표준화를 위해 일부 뉴런 제거하기
  - 과적합 문제를 해결하는 방법은 여러 가지가 있다.
  - 그 중 하나로 간단한데다 효과적이기까지 한 **드롭아웃(Dropout)** 기법이 있다.
  - 신경망의 층에 드롭아웃을 적용하면 각 훈련 단계에서 임의로 뉴런 수를 정하고 그 값을 0으로 만들어 이 뉴런을 훈련 과정에서 완전히 제거하게 된다.
  - 보통 해당 층에서 제거할 뉴런의 비율인 **드롭아웃 비율**을 정하는 식으로 정의한다.

드롭아웃과 정류 선형 유닛

- 표준화를 위해 일부 뉴런 제거하기
  - **드롭아웃(Dropout)** 기법



드롭아웃과 정류 선형 유닛

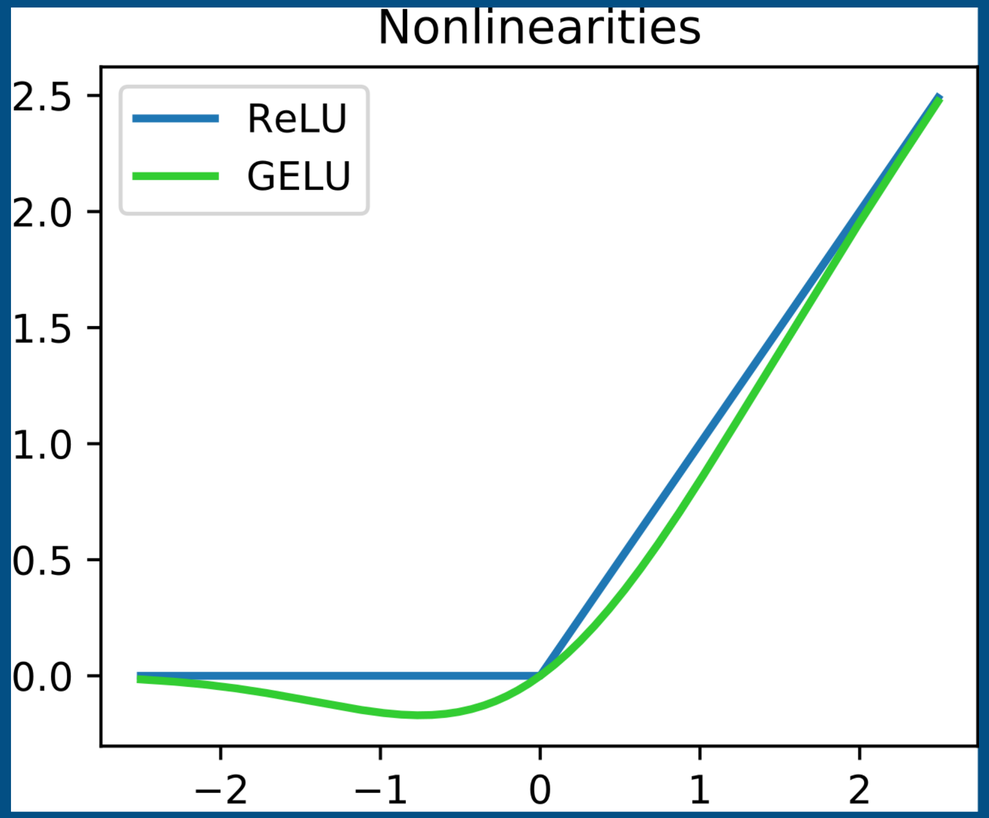
- 표준화를 위해 일부 뉴런 제거하기
  - TensorFlow 2 모델에 Dropout 층 불러와서 추가하기

```
from tensorflow.keras.layers import Dropout
...
model.add(Dropout(rate=0.25))
```

드롭아웃과 정류 선형 유닛

- ReLU 활성화 함수

$$f(x) = x^+ = \max(0, x)$$



https://en.wikipedia.org/wiki/Rectifier_(neural_networks)

드롭아웃과 정류 선형 유닛

- ReLU 활성화 함수
  - ReLU 활성화 함수를 Dense층에 추가하기

```
from tensorflow.keras.layers import Dense
...
model.add(Dense(activation='relu'))
```

기능을 결합해 강력한 신경망 만들기

- 지금까지 살펴본 기능들
  - 최댓값 풀링층을 사용하는 합성곱 신경망
  - 교차 엔트로피 손실 함수
  - 최종 층에서의 소프트맥스 활성화 함수
  - 표준화하는 드롭아웃
  - 신경망 성능을 향상시키는 ReLU 활성화 함수

기능을 결합해 강력한 신경망 만들기

- 합성곱 신경망에 사용할 바둑 데이터 불러와서 전처리하기

```python
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, Flatten
from tensorflow.keras.layers import Conv2D, MaxPooling2D

np.random.seed(123)
X = np.load('../generated_games/features-200.npy')
Y = np.load('../generated_games/labels-200.npy')

samples = X.shape[0]
size = 9
input_shape = (size, size, 1)

X = X.reshape(samples, size, size, 1)

train_samples = int(0.9 * samples)
X_train, X_test = X[:train_samples], X[train_samples:]
Y_train, Y_test = Y[:train_samples], Y[train_samples:]
```

기능을 결합해 강력한 신경망 만들기

- 앞에서 만들었던 합성공 신경망을 다음과 같이 바꿔보자.
  - 기본 구조는 그대로 두 합성곱층으로 두되 최댓값 풀링과 두 개의 밀집층을 추가한다.
  - 표준화를 수행할 드롭아웃층 3개도 추가한다. 각 합성곱층 다음에 하나씩 두고, 하나는 첫 밀집층 뒤에 둔다.
  - 출력층을 소프트맥스 활성화 함수로 바꾸고, 내부 층의 활성화 함수는 ReLU로 한다.
  - 손실 함수를 평균제곱오차 대신 교차 엔트로피 손실 함수로 변경한다.

기능을 결합해 강력한 신경망 만들기

- 드롭아웃과 ReLU를 사용한 바둑 데이터의 합성곱 신경망

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Dropout(rate=0.6))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(rate=0.6))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(rate=0.6))
model.add(Dense(size * size, activation='softmax'))
model.summary()

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

기능을 결합해 강력한 신경망 만들기

- 수정된 합성곱 신경망 평가하기

```
model.fit(X_train, Y_train,
          batch_size=64,
          epochs=100,
          verbose=1,
          validation_data=(X_test, Y_test))
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test loss:', score[0])
print('Test accuracy:', score[1])
```

기능을 결합해 강력한 신경망 만들기

- 수정된 합성곱 신경망 평가하기

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 7, 7, 32)          320
_________________________________________________________________
dropout (Dropout)            (None, 7, 7, 32)          0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 5, 5, 64)          18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 2, 2, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 2, 2, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 256)               0
_________________________________________________________________
dense (Dense)                (None, 128)               32896
_________________________________________________________________
dropout_2 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 81)                10449
=================================================================
Total params: 62,161
Trainable params: 62,161
Non-trainable params: 0
```

기능을 결합해 강력한 신경망 만들기

- 수정된 합성곱 신경망 평가하기

```
2021-05-08 13:44:17.001261: I tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116]
None of the MLIR optimization passes are enabled (registered 2)
Epoch 1/5
157/157 [==============================] - 2s 9ms/step - loss: 4.4638 - accuracy: 0.0129 - val_loss: 4.3925 - val_accuracy: 0.0217
Epoch 2/5
157/157 [==============================] - 1s 6ms/step - loss: 4.4131 - accuracy: 0.0132 - val_loss: 4.3906 - val_accuracy: 0.0158
Epoch 3/5
157/157 [==============================] - 1s 6ms/step - loss: 4.4066 - accuracy: 0.0099 - val_loss: 4.3894 - val_accuracy: 0.0129
Epoch 4/5
157/157 [==============================] - 1s 6ms/step - loss: 4.3987 - accuracy: 0.0152 - val_loss: 4.3883 - val_accuracy: 0.0135
Epoch 5/5
157/157 [==============================] - 1s 6ms/step - loss: 4.3913 - accuracy: 0.0170 - val_loss: 4.3873 - val_accuracy: 0.0147
Test loss: 4.38734245300293
Test accuracy: 0.014662756584584713
```

기능을 결합해 강력한 신경망 만들기

- 고민거리들
  - 이 문제에서 가장 효율적인 풀링은 최댓값 풀링일까, 평균 풀링일까, 혹은 풀링을 안쓰는 것일까? 풀링층을 제거하면 모델의 훈련 파라미터가 늘어난다는 것을 상기하자. 만약 정확도가 늘어난다면 이는 추가 계산 덕에 늘어나는 것임을 명심하자.
  - 세번째 합성곱층을 추가하는 게 좋을까, 아니면 기존 두 층에 필터 수를 늘리는 게 좋을까?
  - 성능이 그대로라는 전제하에 두번째에서 마지막으로 이어지는 Dense층을 얼마나 작게 만들 수 있을까?
  - 드롭아웃 비율을 조정해서 성능을 개선할 수 있을까?
  - 합성곱층을 사용하지 않고 모델을 얼마나 정확하게 만들 수 있을까? CNN을 사용해서 나왔던 가장 좋은 결과와 비교했을 때 모델 크기와 훈련 시간은 어떤가?

과제 #2

- 딥러닝 라이브러리 사용하기
  - TensorFlow 2
  - PyTorch
- 과제 내용 업로드 : 5월 9일 (일) 14:00
- 과제 마감 : 5월 21일 (금) 24:00

감사합니다!

스터디 듣느라 고생 많았습니다.