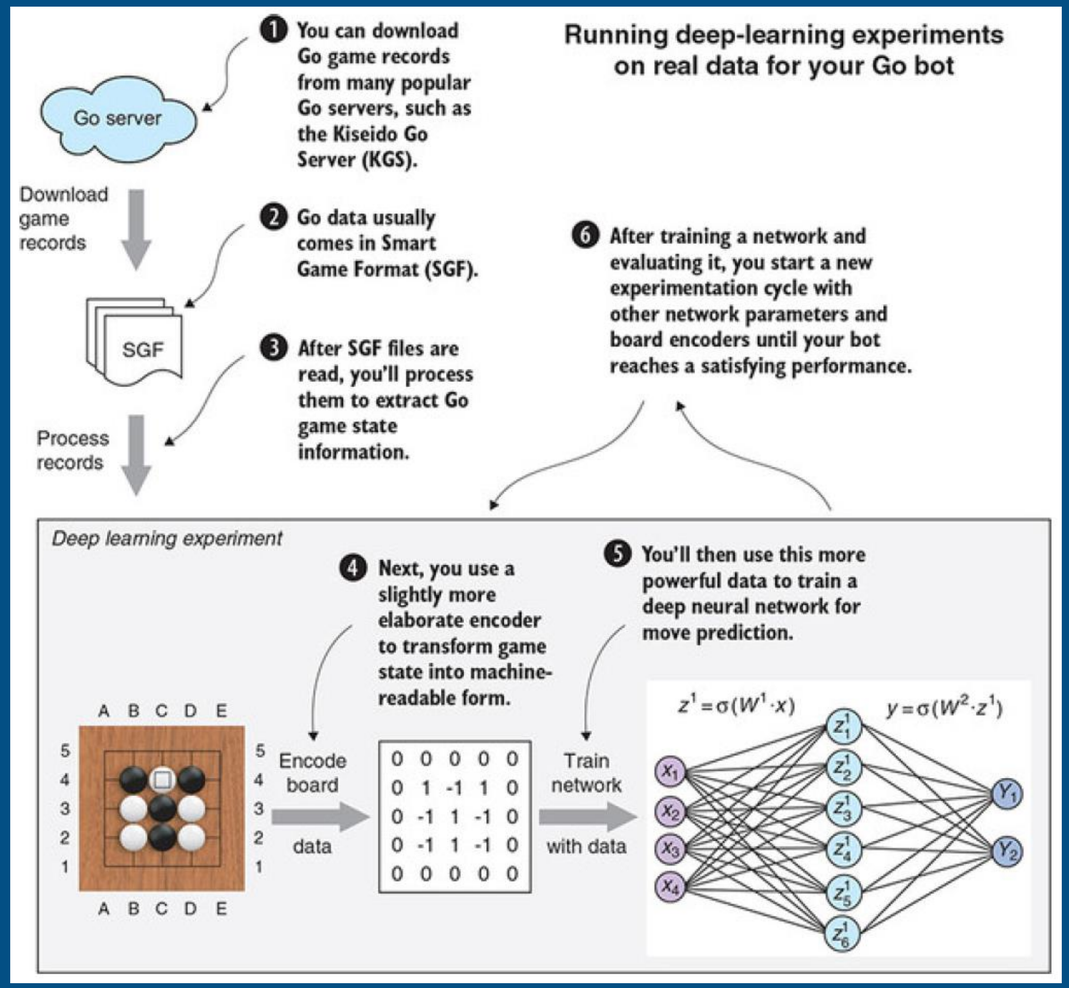
KAIST Include 동아리 스터디

AlphaGo와 AlphaGo Zero를 만들며 익히는 딥러닝 및 강화학습

Chris Ohk
utilForever@gmail.com

들어가며

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기



바둑 대국 기록 가져오기

- 지금까지는 바둑 데이터를 직접 만들어서 사용했었다.
  앞으로는 이미 만들어진 데이터를 사용해서 수를 예측한다.
- 어떤 데이터를 사용할 것인가?
  세계적으로 유명한 바둑 플랫폼인 KGS(Kiseido Go Server)의
  대국 데이터를 사용해서 심층 신경망을 훈련시켜 보자.

바둑 대국 기록 가져오기

- SGF 파일 형식
  - 스마트 게임 포맷(Smart Game Format)의 약자로 80년대 후반에 만들어졌다.
  - 텍스트 기반 형식이어서 바둑 및 다른 보드 게임에서 사용할 수 있다.
  - 참고 : https://senseis.xmp.net/?SmartGameFormat
  - 파일 안에는 대국 및 수에 대한 메타데이터로 구성되어 있다.
    - 메타데이터는 특성을 두 개의 대문자로 나타내고, 대괄호 안에 특성에 대한 값을 넣는다.
    - 예를 들어, 크기(SZ) 9 × 9에서 이루어지는 바둑 대국은 SGF에 `SZ[9]`로 기록된다.
    - 백이 바둑판의 3번째 행과 3번째 열의 좌표에 두었다면 SGF에 `W[cc]`로 기록되고,
      흑이 바둑판의 7번째 행과 3번째 열의 좌표에 두었다면 `B[gc]`로 기록된다.
    - 차례를 넘겼을 경우에는 빈 수인 `B[]`과 `W[]`를 사용한다.

바둑 대국 기록 가져오기

- SGF 파일 형식
  - FF[4] : 현재 SGF 버전, GM[1] : 바둑의 게임 번호
  - HA[0] : 접바둑이 아님, KM[6.5] : 백에게 6.5집의 덤을 줌
  - RU[Japanese] : 대국 규칙은 일본식을 따름, RE[W+9.5] : 백이 9.5집 차이로 이김

```
(;FF[4] GM[1] SZ[9] HA[0] KM[6.5] RU[Japanese] RE[W+9.5]
;B[gc];W[cc];B[cg];W[gg];B[hf];W[gf];B[hg];W[hh];B[ge];W[df];B[dg]
;W[eh];B[cf];W[be];B[eg];W[fh];B[de];W[ec];B[fb];W[eb];B[ea];W[da]
;B[fa];W[cb];B[bf];W[fc];B[gb];W[fe];B[gd];W[ig];B[bd];W[he];B[ff]
;W[fg];B[ef];W[hd];B[fd];W[bi];B[bh];W[bc];B[cd];W[dc];B[ac];W[ab]
;B[ad];W[hc];B[ci];W[ed];B[ee];W[dh];B[ch];W[di];B[hb];W[ib];B[ha]
;W[ic];B[dd];W[ia];B[];
TW[aa][ba][bb][ca][db][ei][fi][gh][gi][hf][hg][hi][id][ie][if][ih][ii]
TB[ae][af][ag][ah][ai][be][bg][bi][ce][df][fe][ga]
W[])
```

바둑 대국 기록 가져오기

- KGS에서 바둑 대국 기록을 다운로드해서 재현하기
  - u-go.net/gamerecords에서 대국 기록 테이블을
    여러 형식(zip, tar.gz)으로 다운로드 할 수 있다.
  - 대국 데이터는 2001년부터 KGS 바둑 서버에서 수집해온 것으로
    기사 1명이 7단 이상이거나 두 기사 모두 6단 이상인 대국 기록만 있다.
  - 숙련된 기사들의 대국 기록이므로 바둑 수 예측에 있어서 엄청나게 좋은 데이터셋이다.

바둑 대국 기록 가져오기

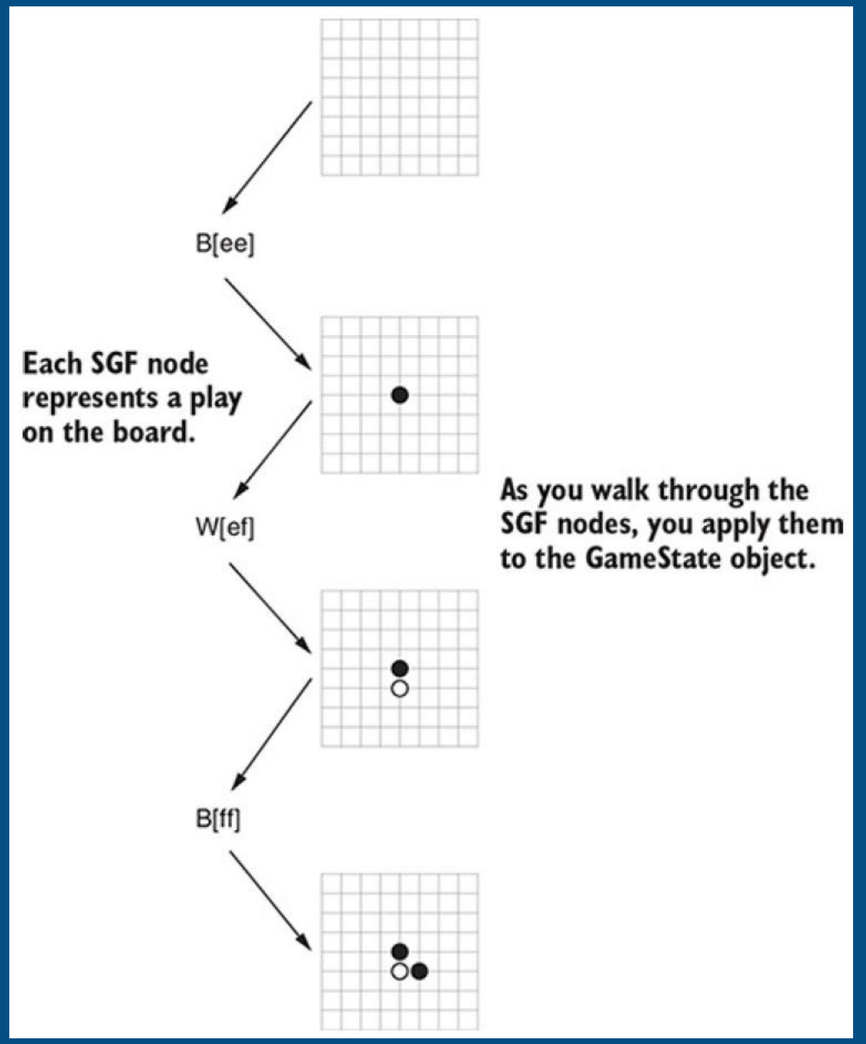
- KGS에서 바둑 대국 기록을 다운로드해서 재현하기
  - KGS에서 바둑 데이터를 포함하는 zip 파일의 인덱스 생성

```
from dlgo.data.index_processor import KGSIndex

index = KGSIndex(data_directory=self.data_dir)
index.download_files()
```

딥러닝용 바둑 데이터 준비

- SGF 기록을 사용해서 바둑 대국 재현하기
  - 바둑 프레임워크에서 SGF 파일로 수 놓기

```
from dlgo.gosgf import Sgf_game
from dlgo.goboard_fast import GameState, Move
from dlgo.gotypes import Point
from dlgo.utils import print_board

sgf_content = "(;GM[1]FF[4]SZ[9];B[ee];W[ef];B[ff]" + \
              ";W[df];B[fe];W[fc];B[ec];W[gd];B[fb])"

sgf_game = Sgf_game.from_string(sgf_content)

game_state = GameState.new_game(19)

for item in sgf.main_sequence_iter():
  color, move_tuple = item.get_move()
  if color is not None and move_tuple is not None:
    row, col = move_tuple
    point = Point(row + 1, col + 1)
    move = Move.play(point)
    game_state = game_state.apply_move(move)
    print_board(game_state.board)
```



딥러닝용 바둑 데이터 준비

- SGF 기록을 사용해서 바둑 대국 재현하기
  - 딥러닝에 사용할 바둑 데이터를 처리하는 과정
    1. 압축된 바둑 대국 파일을 다운로드해서 압축을 푼다.
    2. 이 파일 안의 각 SGF 파일을 Python 문자열로 읽은 후 `Sgf_game` 객체를 만든다.
    3. 각 SGF 문자열에 대한 바둑 대국의 주요 순서를 읽고, 치석(바둑을 두기 전에 미리 깔아두는 돌)과 같은 중요한 세부 사항을 반드시 처리하고, 그 결과로 나온 수 데이터를 `GameState` 객체에 넣어준다.
    4. 각 수에 대해 그 수를 두기 전에 현재 바둑판 정보를 `Encoder`의 특징 정보로 변환하고 각 수를 라벨로 저장한다. 이런 식으로 딥러닝에 사용할 수 예측용 데이터를 만들 수 있다.
    5. 이후에 바로 가져와서 심층 신경망에 넣어줄 수 있도록 결과 특징과 라벨을 적당한 형태로 저장한다.

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - 데이터 및 파일 전처리에 필요한 Python 라이브러리

```
import os.path
import tarfile
import gzip
import glob
import shutil

import numpy as np
from tensorflow.keras.utils import to_categorical
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - dlgo 모듈에서 데이터 전처리에 사용할 내용 불러오기

```
from dlgo.gosgf import Sgf_game
from dlgo.goboard_fast import Board, GameState, Move
from dlgo.gotypes import Player, Point
from dlgo.encoders.base import get_encoder_by_name

from dlgo.data.index_processor import KGSIndex
from dlgo.data.sampling import Sampler
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - 변환기와 로컬 데이터 디렉터리를 지정해서 바둑 데이터 전처리기 초기화하기

```
class GoDataProcessor:
  def __init__(self, encoder='oneplane', data_directory='data'):
    self.encoder = get_encoder_by_name(encoder, 19)
    self.data_dir = data_directory
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - load_go_data에서 데이터를 불러오고 전처리한 후 저장하기

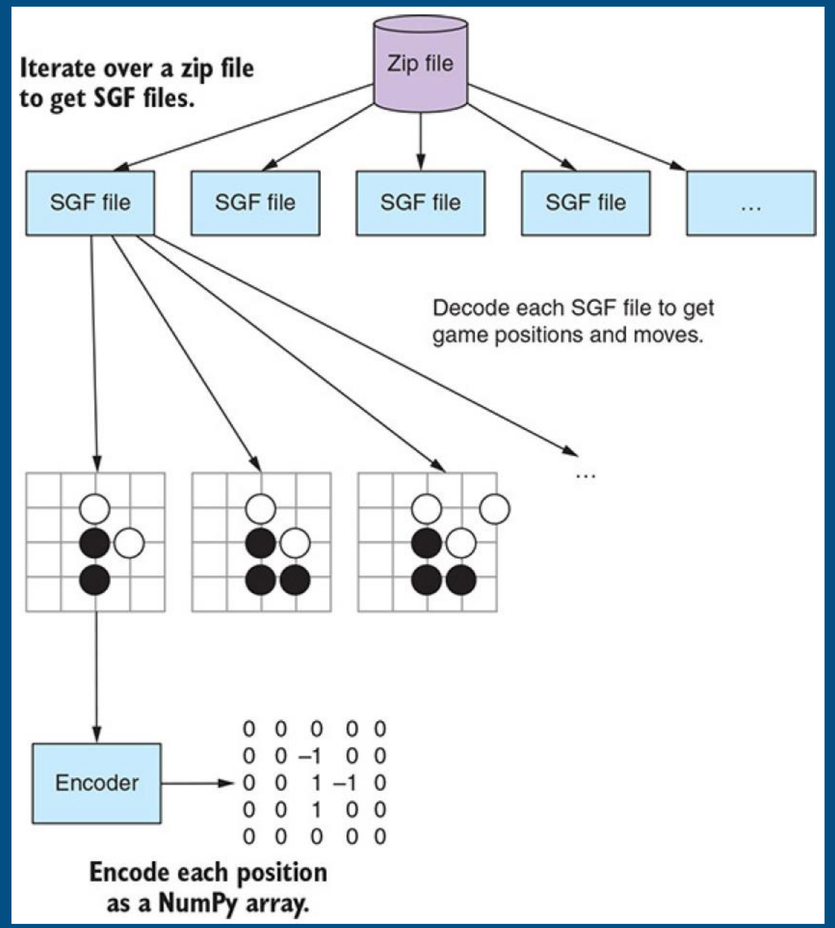
```
def load_go_data(self, data_type='train', num_samples=1000):
  index = KGSIndex(data_directory=self.data_dir)
  index.download_files()

  sampler = Sampler(data_dir=self.data_dir)
  data = sampler.draw_data(data_type, num_samples)

  zip_names = set()
  indices_by_zip_name = {}
  for filename, index in data:
    zip_names.add(filename)
    if filename not in indices_by_zip_name:
      indices_by_zip_name[filename] = []
    indices_by_zip_name[filename].append(index)
  for zip_name in zip_names:
    base_name = zip_name.replace('.tar.gz', '')
    data_file_name = base_name + data_type
    if not os.path.isfile(self.data_dir + '/' + data_file_name):
      self.process_zip(zip_name, data_file_name, indices_by_zip_name[zip_name])

  features_and_labels = self.consolidate_games(data_type, data)
  return features_and_labels
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - 함수 `process_zip()` 구현 과정
    1. 함수 `unzip_data()`를 사용해서 현재 파일의 압축을 푼다.
    2. SGF 기록을 변환하기 위해 `Encoder` 인스턴스를 초기화한다.
    3. 특징과 라벨에 사용할 numpy 배열을 올바른 형태로 초기화한다.
    4. 대국 리스트에 대해 반복하고 대국별로 하나하나 처리한다.
    5. 각 대국에 대해 모든 치석을 우선 적용한다.
    6. 그 후 SGF 기록에서 발견되는 수를 하나하나 읽어들인다.
    7. 각 다음 수에 대해 수를 라벨로 변환한다.
    8. 현재 바둑판의 상태를 특징으로 변환한다.
    9. 바둑판에 다음 수를 두고 다시 처리를 진행한다.
    10. 특징과 라벨을 작은 단위로 묶어서 로컬 파일시스템에 저장한다.

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - 함수 `process_zip()` 구현 과정



딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - zip 파일에 저장된 바둑 기록 처리를 통해 특징과 라벨로 변환하기

```
def process_zip(self, zip_file_name, data_file_name, game_list):
  tar_file = self.unzip_data(zip_file_name)
  zip_file = tarfile.open(self.data_dir + '/' + tar_file)
  name_list = zip_file.getnames()
  total_examples = self.num_total_examples(zip_file, game_list, name_list)

  shape = self.encoder.shape()
  feature_shape = np.insert(shape, 0, np.asarray([total_examples]))
  features = np.zeros(feature_shape)
  labels = np.zeros((total_examples,))

  counter = 0
  for index in game_list:
    name = name_list[index + 1]
    if not name.endswith('.sgf'):
      raise ValueError(name + ' is not a valid sgf')
    sgf_content = zip_file.extractfile(name).read()
    sgf = Sgf_game.from_string(sgf_content)
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - zip 파일에 저장된 바둑 기록 처리를 통해 특징과 라벨로 변환하기

```
      game_state, first_move_done = self.get_handicap(sgf)

      for item in sgf.main_sequence_iter():
        color, move_tuple = item.get_move()
        point = None
        if color is not None:
          if move_tuple is not None:
            row, col = move_tuple
            point = Point(row + 1, col + 1)
            move = Move.play(point)
          else:
            move = Move.pass_turn()
          if first_move_done and point is not None:
            features[counter] = self.encoder.encode(game_state)
            labels[counter] = self.encoder.encode_point(point)
            counter += 1
          game_state = game_state.apply_move(move)
          first_move_done = True
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - 현재 zip 파일에서 가능한 수의 전체 개수 구하기

```
def num_total_examples(self, zip_file, game_list, name_list):
  total_examples = 0
  for index in game_list:
    name = name_list[index + 1]
    if name.endswith('.sgf'):
      sgf_content = zip_file.extractfile(name).read()
      sgf = Sgf_game.from_string(sgf_content)
      game_state, first_move_done = self.get_handicap(sgf)

      num_moves = 0
      for item in sgf.main_sequence_iter():
        color, move = item.get_move()
        if color is not None:
          if first_move_done:
            num_moves += 1
          first_move_done = True
      total_examples = total_examples + num_moves
    else:
      raise ValueError(name + ' is not a valid sgf')
  return total_examples
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - 치석을 찾아서 빈 바둑판에 놓기

```
@staticmethod
def get_handicap(sgf):
  go_board = Board(19, 19)
  first_move_done = False
  move = None
  game_state = GameState.new_game(19)
  if sgf.get_handicap() is not None and sgf.get_handicap() != 0:
    for setup in sgf.get_root().get_setup_stones():
      for move in setup:
        row, col = move
        go_board.place_stone(Player.black, Point(row + 1, col + 1))
    first_move_done = True
    game_state = GameState(go_board, Player.white, None, move)
  return game_state, first_move_done
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - 특징과 라벨을 작은 묶음으로 나눠서 로컬에 저장하기

```
    feature_file_base = self.data_dir + '/' + data_file_name + '_features_%d'
    label_file_base = self.data_dir + '/' + data_file_name + '_labels_%d'

    chunk = 0
    chunksize = 1024
    while features.shape[0] >= chunksize:
      feature_file = feature_file_base % chunk
      label_file = label_file_base % chunk
      chunk += 1
      current_features, features = features[:chunksize], features[chunksize:]
      current_labels, labels = labels[:chunksize], labels[chunksize:]
      np.save(feature_file, current_features)
      np.save(label_file, current_labels)
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - 개별 numpy 배열의 특징과 라벨을 하나로 합침

```python
def consolidate_games(self, data_type, samples):
  files_needed = set(file_name for file_name, index in samples)
  file_names = []
  for zip_file_name in files_needed:
    file_name = zip_file_name.replace('.tar.gz', '') + data_type
    file_names.append(file_name)

  feature_list = []
  label_list = []
  for file_name in file_names:
    file_prefix = file_name.replace('.tar.gz', '')
    base = self.data_dir + '/' + file_prefix + '_features_*.npy'
    for feature_file in glob.glob(base):
      label_file = feature_file.replace('features', 'labels')
      x = np.load(feature_file)
      y = np.load(label_file)
      x = x.astype('float32')
      y = to_categorical(y.astype(int), 19 * 19)
      feature_list.append(x)
      label_list.append(y)
  features = np.concatenate(feature_list, axis=0)
  labels = np.concatenate(label_list, axis=0)
  np.save('{}/features_{}.npy'.format(self.data_dir, data_type), features)
  np.save('{}/labels_{}.npy'.format(self.data_dir, data_type), labels)

  return features, labels
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 전처리기 만들기
  - 100개 대국 기록에서 훈련 데이터 불러오기

```
from dlgo.data.processor import GoDataProcessor

processor = GoDataProcessor
features, labels = processor.load_go_data('train', 100)
```

딥러닝용 바둑 데이터 준비

- 데이터를 효율적으로 불러오는 바둑 데이터 생성기 만들기
  - 다운로드한 KGS 인덱스에는 대국 기록은 약 170,000개다.
    이를 변환하면 예측에 사용할 수 있는 수가 수백만 개가 된다.
  - 이를 numpy 배열 하나로 만드는 건 대국 기록이 늘어날수록 더 어려운 작업이 된다.
  - 결국 모든 신경망은 훈련을 할 때 특징과 라벨의 미니배치를 **하나하나** 넣어줘야 한다.
    즉, 모든 데이터를 항상 메모리에 가지고 있을 필요가 없다.
  - 따라서 바둑 **데이터 생성기(Data Generator)**를 통해
    필요한 데이터의 다음 미니배치만 제공함으로써 효율적으로 만든다.

딥러닝용 바둑 데이터 준비

- 데이터를 효율적으로 불러오는 바둑 데이터 생성기 만들기
  - 바둑 데이터 생성기의 특징

```python
import glob
import numpy as np
from tensorflow.keras.utils import to_categorical


class DataGenerator:
  def __init__(self, data_directory, samples):
    self.data_directory = data_directory
    self.samples = samples
    self.files = set(file_name for file_name, index in samples)
    self.num_samples = None

  def get_num_samples(self, batch_size=128, num_classes=19 * 19):
    if self.num_samples is not None:
      return self.num_samples
    else:
      self.num_samples = 0
      for X, y in self._generate(batch_size=batch_size, num_classes=num_classes):
        self.num_samples += X.shape[0]
      return self.num_samples
```

딥러닝용 바둑 데이터 준비

- 데이터를 효율적으로 불러오는 바둑 데이터 생성기 만들기
  - 바둑 데이터의 다음 배치를 생성해서 불러오는 private 메소드

```
def _generate(self, batch_size, num_classes):
  for zip_file_name in self.files:
    file_name = zip_file_name.replace('.tar.gz', '') + 'train'
    base = self.data_directory + '/' + file_name + '_features_*.npy'
    for feature_file in glob.glob(base):
      label_file = feature_file.replace('features', 'labels')
      x = np.load(feature_file)
      y = np.load(label_file)
      x = x.astype('float32')
      y = to_categorical(y.astype(int), num_classes)
      while x.shape[0] >= batch_size:
        x_batch, x = x[:batch_size], x[batch_size:]
        y_batch, y = y[:batch_size], y[batch_size:]
        yield x_batch, y_batch
```

딥러닝용 바둑 데이터 준비

- 데이터를 효율적으로 불러오는 바둑 데이터 생성기 만들기
  - 모델 훈련에 생성기를 사용하기 위한 generate() 메소드 호출

```
def generate(self, batch_size=128, num_classes=19 * 19):
  while True:
    for item in self._generate(batch_size, num_classes):
      yield item
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 처리 및 생성기의 병렬 실행
  - 100개의 대국 기록을 불러오는데 생각보다 시간이 오래 걸린다. 왜 그럴까?
  - zip 파일을 어떻게 처리했는지 생각해보자.
    한 파일을 처리한 후 다음 파일을 **순차** 처리하도록 구현했다.
  - 자세히 살펴보면 바둑 데이터는 **처치 곤란 병렬 (Embarrassingly Parallel)** 문제다.
    - 병렬 컴퓨팅에서 문제를 몇 개의 병렬 임무로 나누는 데 노력이 거의 들지 않거나 하나도 들지 않는 문제를 말한다.
  - 이때 Python의 멀티프로세싱 라이브러리 등을 사용하면
    컴퓨터의 모든 CPU에 작업을 분배해 zip 파일을 손쉽게 병렬 처리할 수 있다.

딥러닝용 바둑 데이터 준비

- 바둑 데이터 처리 및 생성기의 병렬 실행
  - 생성기를 반환하는 옵션이 있는 `load_go_data()`의 병렬 버전

```
def load_go_data(self, data_type='train', num_samples=1000,
                 use_generator=False):
  index = KGSIndex(data_directory=self.data_dir)
  index.download_files()

  sampler = Sampler(data_dir=self.data_dir)
  data = sampler.draw_data(data_type, num_samples)

  self.map_to_workers(data_type, data)
  if use_generator:
    generator = DataGenerator(self.data_dir, data)
    return generator
  else:
    features_and_labels = self.consolidate_games(data_type, data)
    return features_and_labels
```

딥러닝용 바둑 데이터 준비

- 바둑 데이터 처리 및 생성기의 병렬 실행
  - 100개의 대국 기록에서 훈련 데이터 가져오기

```
from dlgo.data.parallel_processor import GoDataProcessor

processor = GoDataProcessor
generator = processor.load_go_data('train', 100, use_generator=True)
print(generator.get_num_samples())
generator = generator.generate(batch_size=10)
x, y = generator.next()
```

대국 기록으로 딥러닝 모델 훈련하기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 만들려는 신경망은 합성곱층 4개에 최종 밀집층이 있고, ReLU 활성화 함수를 사용한다.
  - 합성곱층 위에는 `ZeroPadding2D`라는 새로운 유틸리티 층을 사용한다.
    - 제로 패딩(Zero Padding)은 입력 특징에 0을 덧대는 연산이다.
    - 예를 들어, 이전에 사용했던 1차 평면 변환기를 사용해서 바둑판을 19×19 행렬로 바꿨다고 해보자.
      크기 2만큼을 덧대어 채운다고 하면 왼쪽과 오른쪽에 0으로 이루어진 두 열을 더하고,
      위와 아래에도 0으로 이루어진 두 행을 더해서 23×23 행렬로 크기를 키우게 된다.
    - 합성곱 입력값에 제로 패딩을 적용해서 크기를 수동으로 늘렸으므로
      합성곱 연산을 통과해도 이미지가 과하게 줄어들지 않는다.

대국 기록으로 딥러닝 모델 훈련하기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 바둑 수 예측용 작은 합성곱 신경망에 사용할 층 정의

```
from tensorflow.keras.layers import Dense, Activation, Flatten
from tensorflow.keras.layers import Conv2D, ZeroPadding2D


def layers(input_shape):
  return [
    ZeroPadding2D(padding=3, input_shape=input_shape, data_format='channels_first'),
    Conv2D(48, (7, 7), data_format='channels_first'),
    Activation('relu'),

    ZeroPadding2D(padding=2, data_format='channels_first'),
    Conv2D(32, (5, 5), data_format='channels_first'),
    Activation('relu'),

    ZeroPadding2D(padding=2, data_format='channels_first'),
    Conv2D(32, (5, 5), data_format='channels_first'),
    Activation('relu'),

    ZeroPadding2D(padding=2, data_format='channels_first'),
    Conv2D(32, (5, 5), data_format='channels_first'),
    Activation('relu'),

    Flatten(),
    Dense(512),
    Activation('relu'),
  ]
```

대국 기록으로 딥러닝 모델 훈련하기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 바둑 데이터 신경망 구축에 필요한 주요 컴포넌트 불러오기

```
from dlgo.data.parallel_processor import GoDataProcessor
from dlgo.encoders.oneplane import OnePlaneEncoder

from dlgo.networks import small
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import ModelCheckpoint
```

대국 기록으로 딥러닝 모델 훈련하기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 훈련 및 검정 데이터 생성기 만들기

```
go_board_rows, go_board_cols = 19, 19
num_classes = go_board_rows * go_board_cols
num_games = 100

encoder = OnePlaneEncoder((go_board_rows, go_board_cols))

processor = GoDataProcessor(encoder=encoder.name())

generator = processor.load_go_data('train', num_games, use_generator=True)
test_generator = processor.load_go_data('test', num_games, use_generator=True)
```

대국 기록으로 딥러닝 모델 훈련하기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 적은 층 구조의 TensorFlow 모델 정의하기

```
input_shape = (encoder.num_planes, go_board_rows, go_board_cols)
network_layers = small.layers(input_shape)
model = Sequential()
for layer in network_layers:
  model.add(layer)
model.add(Dense(num_classes, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
```

대국 기록으로 딥러닝 모델 훈련하기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 생성기를 적용한 TensorFlow 모델을 최적화하고 평가하기

```
epochs = 5
batch_size = 128
model.fit_generator(generator=generator.generate(batch_size, num_classes),
                    epochs=epochs,
                    steps_per_epoch=generator.get_num_samples() / batch_size,
                    validation_data=test_generator.generate(batch_size, num_classes),
                    validation_steps=test_generator.get_num_samples() / batch_size,
                    callbacks=[ModelCheckpoint('../checkpoints/small_model_epoch_{epoch}.h5')])

model.evaluate_generator(generator=test_generator.generate(batch_size, num_classes),
                         steps=test_generator.get_num_samples() / batch_size)
```

실질적인 바둑 데이터 변환기 만들기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 바둑에는 무한 반복을 피하는 패 규칙이 있다.
    즉, 판 위에서 바로 직전에 있었던 모양으로 둘 수 없다.
  - 임의의 바둑판 상태를 주고 여기에서 패가 발생했는지 판단해야 한다면
    임의로 추측할 수 밖에 없다. 왜냐하면 이전 수 위치를 알 수 없기 때문이다.
  - 특히 흑을 -1로, 백을 1로, 빈자리를 0으로 변환하는 1차 평면 변환기에서는
    패에 대해 어떤 정보도 얻기 어렵다.

실질적인 바둑 데이터 변환기 만들기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 여기서는 더 강한 수를 예측하는 더 정교한 변환기 2개를 소개한다.
  - SevenPlaneEncoder : 7가지 특징 평면으로 구성되어 있다.
    각 평면은 19×19 행렬이며 각각 다음과 같은 특징을 나타낸다.
    - 첫 특징 평면에서는 활로가 1인 모든 백돌을 1로 나타냈고, 나머지는 0으로 나타냈다.
    - 두번째와 세번째 특징 평면은 최소 2~3개의 활로가 있는 백돌을 1로 나타냈다.
    - 네번째에서 여섯번째 특징 평면은 흑돌에 대해 동일한 값을 갖는다. 활로가 1, 2, 3인 흑돌을 1로 표기한다.
    - 마지막 특징 평면은 패가 발생해서 움직이지 않는 돌을 1로 표기한다.
  - 이런 특징 집합을 사용하면 패 개념을 명시적으로 나타낼 수 있고,
    자유도 표기 및 흑백 구분도 가능하다.

실질적인 바둑 데이터 변환기 만들기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 간단한 7차 평면 변환기 초기화

```python
import numpy as np

from dlgo.encoders.base import Encoder
from dlgo.goboard import Move, Point


class SevenPlaneEncoder(Encoder):
  def __init__(self, board_size):
    self.board_width, self.board_height = board_size
    self.num_planes = 7

  def name(self):
    return 'sevenplane'
```

실질적인 바둑 데이터 변환기 만들기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - SevenPlaneEncoder로 대국 상태 변환

```
def encode(self, game_state):
  board_tensor = np.zeros(self.shape())
  base_plane = {game_state.next_player: 0,
                game_state.next_player.other: 3}
  for row in range(self.board_height):
    for col in range(self.board_width):
      p = Point(row=row + 1, col=col + 1)
      go_string = game_state.board.get_go_string(p)
      if go_string is None:
        if game_state.does_move_violate_ko(game_state.next_player,
                                           Move.play(p)):
          board_tensor[6][row][col] = 1
      else:
        liberty_plane = min(3, go_string.num_liberties) - 1
        liberty_plane += base_plane[go_string.color]
        board_tensor[liberty_plane][row][col] = 1
  return board_tensor
```

실질적인 바둑 데이터 변환기 만들기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 7차 평면 변환기용 Encoder() 메소드 구현

```
  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


def create(board_size):
  return SevenPlaneEncoder(board_size)
```

실질적인 바둑 데이터 변환기 만들기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 여기서는 `SevenPlaneEnocder` 변환기와 유사한 특징 평면 변환기 11개를 사용한다.
  - `SimpleEncoder`라고 하는 변환기는 다음 특징 평면을 사용한다.
    - 처음 4개의 특징 평면은 1, 2, 3, 4개의 활로를 갖는 흑돌을 나타낸다.
    - 다음 4개의 특징 평면은 1, 2, 3, 4개의 활로를 갖는 백돌을 나타낸다.
    - 9번째 특징 평면은 흑의 차례를 1로 나타내고, 10번째 특징 평면은 백의 차례를 1로 나타낸다.
    - 마지막 특징 평면은 패를 표시한다.

적응 경사법을 사용해 훈련하기

- 바둑 수 예측 모델의 성능을 개선하는 새로운 도구를 알아보자.
  - 이전에 살펴봤던 SGD의 갱신 규칙은 꽤 단순했었다.
  - SGD는 파라미터 W에 대해 ΔW 만큼의 역전파 오차를 얻었고
    학습률이 α로 정의되어 있다면 단순히 $W - \alpha\Delta W$를 계산해서 파라미터를 갱신했다.
  - 많은 경우 이 갱신 규칙을 사용해도 좋은 결과가 나오지만 몇 가지 약점이 있다.
    이때 기본 SGD의 확장형들을 사용해 이 약점을 해결할 수 있다.

적응 경사법을 사용해 훈련하기

- SGD에서의 붕괴와 모멘텀
  - 학습률 **붕괴(Decay)**는 매회 모델 갱신 단계에서 학습률을 점차 낮추는 기법으로 널리 사용되는 아이디어다.
  - 보통 신경망 초기에는 학습한 것이 아무것도 없으므로 손실 함수의 값이 최대로 작아질 때까지는 갱신이 많이 되는 것은 당연하다. 하지만 훈련 과정이 어느 수준에 다다른 후에는 지금까지 나아진 것을 무너뜨리지 않을 정도로 적당한 내용만 다듬는 정도로 갱신해야 한다.
  - 보통은 다음 단계에서 학습률을 몇 퍼센트나 낮출지 나타내는 **붕괴율(Decay Rate)**을 사용해서 학습률 붕괴 정도를 정의한다.

적응 경사법을 사용해 훈련하기

- SGD에서의 붕괴와 모멘텀
  - 학습률의 **모멘텀(Momentum)**이란 마지막 갱신 단계의 얼마가 현재 갱신 단계에 반영되는지를 나타낸다. 예를 들어, 갱신하려는 파라미터 벡터를 W라고 하면 ∂W는 W로부터 구한 현재의 경삿값이고, 최종 갱신에 U를 사용했다면 다음 갱신 단계는 아래와 같이 이루어진다.

$$W \leftarrow W - \alpha(\gamma U + (1-\gamma)\partial W)$$

  - 마지막 갱신 때 유지되는 정도를 **모멘텀 항(Momentum Term)**이라고 한다. 만약 경사 항이 유사한 방향으로 진행된다면 다음 갱신 단계는 더 강화될 것이다. 경사가 반대로 진행된다면 서로 상쇄되면서 경사가 줄어들 것이다.

적응 경사법을 사용해 훈련하기

- SGD에서의 붕괴와 모멘텀
  - 모멘텀과 학습률 붕괴를 사용해서 TensorFlow에서 SGD 초기화하기

```
from tensorflow.keras.optimizers import SGD
sgd = SGD(lr=0.1, momentum=0.9, decay=0.01)
```

적응 경사법을 사용해 훈련하기

- Adagrad로 신경망 최적화하기
  - 학습률 낮추기와 모멘텀은 SGD를 잘 조정해주지만 여전히 몇 가지 약점이 남아 있다.
  - 예를 들어, 바둑 고수라면 보통 처음 몇 번은 바둑판의 세번째에서 다섯번째 줄에 수를 두지, 특별한 경우가 아니고서는 절대 첫번째나 두번째 줄에 수를 두지 않을 것이다. 하지만 종국에는 이와 반대로 많은 수를 바둑판의 모서리 쪽에 둘 것이다.
  - 여기서 사용하는 모든 딥러닝 모델에서 마지막 층은 바둑판 크기의 밀집층을 사용한다. 이 층의 각 뉴런은 바둑판의 위치에 대응하는데, SGD를 사용하는 경우 모멘텀이나 붕괴 정도에 상관없이 **뉴런 모두가 동일한 학습률을 사용한다**.
  - 이 방식은 위험할 수 있다. 훈련 데이터가 제대로 섞이지 않으면 학습률이 무너질 것이고, 결국 첫번째/두번째 줄에 두는 수의 근소하게 차이나는 정보를 찾을 수 없게 될 것이다.

적응 경사법을 사용해 훈련하기

- Adagrad로 신경망 최적화하기
  - 전역 학습률을 사용할 때 나타나는 문제를 해결하는 데 **적용 경사법** 방식을 활용한다.
    여기서는 AdaGrad와 AdaDelta라는 두 가지 기법을 소개한다.
  - AdaGrad에는 전역 학습률이 없다. 대신 **파라미터별로 학습률을 적용한다**.
    - 데이터가 많고 데이터에서 패턴이 아주 가끔 발견되는 경우 AdaGrad를 사용하면 성능이 좋아진다.
  - l 길이의 가중치 벡터 W를 사용해서 각 항목을 W_i로 사용하는 경우를 가정해보자.
    파라미터별 기울기 ∂W가 주어졌다면 학습률 α를 사용하는 SGD에서
    각 W_i의 갱신 규칙은 다음과 같다.

$$W_i \leftarrow W_i - \alpha \partial W_i$$

적응 경사법을 사용해 훈련하기

- Adagrad로 신경망 최적화하기
  - AdaGrad에서는 각 인덱스 i별로 과거에 W_i를 얼마나 갱신했는지 살펴보고 이에 따라 동적으로 적용 가능한 항목을 사용해서 α를 대체한다.
  - AdaGrad에서 사용하는 학습률은 이전 갱신률의 역수를 사용하며, 다음과 같이 파라미터를 갱신한다.

$$W_i \leftarrow W_i - \frac{\alpha}{\sqrt{G_{i,i} + \varepsilon}} \partial W_i$$

적응 경사법을 사용해 훈련하기

- Adagrad로 신경망 최적화하기

$$W_i \leftarrow W_i - \frac{\alpha}{\sqrt{G_{i,i} + \varepsilon}} \partial W_i$$

  - 이 식에서 ε는 0으로 나눠지지 않게 하기 위한 작은 양수고,
    $G_{i,i}$는 이 점에서 받게 되는 W_i의 기울기 제곱의 합이다.
  - 이 값은 길이가 l인 정사각 행렬 G의 일부로 볼 수 있으므로 $G_{i,i}$로 쓴다.
    모든 대각선 항 $G_{i,i}$는 모든 대각항을 제외한 항이 0인 행렬, 즉 대각 행렬이다.
  - 각 파라미터를 갱신한 후 기울기가 직전에 갱신된 정도를 대각 원소에 더해서
    G를 갱신한다.

적응 경사법을 사용해 훈련하기

- Adagrad로 신경망 최적화하기
  - TensorFlow 모델에 AdaGrad 최적화기 사용하기

```
from tensorflow.keras.optimizers import Adagrad
adagrad = Adagrad()
```

적응 경사법을 사용해 훈련하기

- AdaDelta로 적응 경사법 조정하기
  - AdaGrad와 비슷한 최적화기로 확장판인 **AdaDelta**가 있다.
  - G의 모든 과거 기울기값의 제곱을 취하는 대신 모멘텀 기법에서 살펴본 것과 유사한 개념을 사용해서 **직전에 갱신된 비율**만을 취해서 현재 기울기에 더한다.

$$G \leftarrow \gamma G + (1-\gamma)\partial W$$

적응 경사법을 사용해 훈련하기

- AdaDelta로 적응 경사법 조정하기
  - TensorFlow 모델에서 AdaDelta 최적화기 사용하기

```
from tensorflow.keras.optimizers import Adadelta
adadelta = Adadelta()
```

직접 실험하고 성능 평가하기

- 실제 바둑 데이터를 훈련 데이터로 사용하는 딥러닝 바둑봇 만들기
  - 이제 직접 모델을 훈련시킬 시간이다.
  - 머신러닝 실험에서는 층수, 선택할 층, 훈련에 사용할 세대 수 등의 **하이퍼파라미터**를 다양하게 조합해보는 것이 중요하다.
  - 우리가 만드는 실험 설정에서 훌륭한 결과를 나오게 하는 중요한 요인은 **빠른 실험 주기**다. 모델 구조를 만들고, 모델 훈련을 시작하고, 성능 지표를 관측 및 평가하고, 돌아가서 다시 모델을 수정하고 새로운 프로세스를 시작하는데 걸리는 시간이 짧아야 한다.
  - 실제로 관련 대회를 보면 **가장 많은 시도를 해본** 팀이 종종 1등을 차지한다.

직접 실험하고 성능 평가하기

- 모델 구조 및 하이퍼파라미터 검정 지침
  - 합성곱 신경망은 바둑 수 예측에 사용하기 좋다. 하지만 밀집층만 사용하면 예측 품질이 매우 낮아질 것이라는 것을 반드시 인지해야 한다. 여러 합성곱층에 1~2개의 밀집층을 끝에 붙여서 신경망을 만드는 것이 필수다. 이후에는 더 복잡한 신경망 구조를 살펴보게 되겠지만 일단 지금은 합성곱 신경망을 사용하자.
  - 합성곱 신경망에서 커널 크기를 다양하게 사용해서 이를 변경하는 경우 모델 성능이 얼마나 달라지는지 살펴보자. 경험적으로 커널 크기는 2~7이 적합하고, 더 크면 안된다.
  - 풀링층을 사용하려면 최댓값 풀링과 평균 풀링 모두 실험을 해봐야 하고, 무엇보다도 풀링 크기를 너무 크게 잡으면 안된다. 실질적으로 이런 경우는 3을 넘기지 않는다. 풀링층을 사용하지 않고 신경망을 만든다면 계산량이 훨씬 많아지겠지만 성능은 괜찮을 것이다.

직접 실험하고 성능 평가하기

- 모델 구조 및 하이퍼파라미터 검정 지침
  - 표준화를 위해 드롭아웃층을 사용하자. 이전에 드롭아웃을 사용해서 모델 과적합을 막는 것을 확인했다. 신경망에 드롭아웃층을 붙이면 보통 도움은 되지만 드롭아웃을 너무 사용하지 말고 드롭아웃 비율을 너무 높게 잡지도 말자.
  - 마지막 층에 소프트맥스 활성화 함수를 넣어서 확률분포를 생성하고 이를 범주형 크로스 엔트로피 손실 함수와 조합해서 사용하면 수 예측에 잘 들어맞을 것이다.
  - 여러 활성화 함수를 실험해보자. 앞서 현재 기본으로 사용하고 있는 ReLU와 시그모이드 활성화 함수를 살펴봤다. TensorFlow에는 elu, selu, PReLU, LeakyReLU 등 여러 다른 활성화 함수가 많이 있다. 여기서 ReLU의 변형 형태는 다루지 않았지만 TensorFlow 문서에 용례가 잘 나와 있다.

직접 실험하고 성능 평가하기

- 모델 구조 및 하이퍼파라미터 검정 지침
  - 미니배치 크기에 따라 모델 성능이 달라질 것이다. MNIST 같은 예측 문제의 경우 보통 미니배치 수를 클래스 수의 배수로 잡을 것을 추천한다. (예 : 10 또는 50) 만약 데이터가 완벽히 임의로 분포한다면 각 경삿값은 각 클래스로부터 정보를 받아오게 되므로 SGD가 일반적으로 더 성능이 좋다. 여기서 만드는 바둑봇의 경우 몇 가지 수는 다른 수보다 더 자주 두게 된다. 예를 들어, 바둑판의 네 귀퉁이는 화점에 비하면 훨씬 덜 사용한다. 우리는 이를 데이터의 **클래스가 불균형**하다고 한다. 이 경우 미니배치 조절만으로는 모든 클래스를 처리할 수 없으므로 미니배치 크기를 16에서 256 사이에서 다양하게 조율해봐야 한다.

직접 실험하고 성능 평가하기

- 모델 구조 및 하이퍼파라미터 검정 지침
  - 최적화기를 선택하는 것 역시 신경망의 핮브 정도에 의미있는 영향력을 행사한다. 학습률 감소를 사용하거나 SGD, AdaGrad, AdaDelta 등의 최적화기를 사용할 수 있다. TensorFlow 문서에서 모델 훈련 과정에서 사용 가능한 다른 최적화기도 살펴볼 수 있다.
  - 모델 훈련에 사용된 세대 수도 적절하게 정해야 한다. 모델 체크포인트를 사용해서 세대별로 여러 성능 지표를 확인하고 있다면 언제 훈련을 멈추는 것이 좋을지 효과적으로 측정할 수 있을 것이다. 일반적으로 컴퓨터의 여력이 충분하다면 세대 수는 과하게 높은 것이 과하게 낮은 것보다 낫다. 만약 모델 훈련 상태가 더 나아지지 않거나 과적합으로 인해 더 나빠진다고 해도 이전 체크포인트를 사용할 수 있다.

직접 실험하고 성능 평가하기

- 훈련 및 검정 데이터로 성능 지표 평가하기
  - 훈련 정확도와 손실값은 일반적으로 세대가 지날수록 향상되어야 한다. 뒤로 갈수록 이러한 지표가 점점 줄어들고 때로는 약간 흔들릴 것이다. 만약 몇 세대가 지나도록 더 이상 증가하지 않는다면 훈련을 멈추는 것이 좋을 지도 모른다.
  - 또한 검증 손실값과 정확도를 살펴봐야 한다. 초기 세대에는 검증 손실값이 꾸준히 낮아지지만, 후기 세대에는 값이 일정해지다가 간혹 다시 증가하는 것도 보게 될 것이다. 이는 신경망이 훈련 데이터에 과적합되었다는 명백한 신호다.
  - 모델 체크포인트를 사용한다면 훈련 정확도가 가장 높으면서 검증 오차가 낮은 모델을 고르자.

직접 실험하고 성능 평가하기

- 훈련 및 검정 데이터로 성능 지표 평가하기
  - 훈련 오차와 검증 오차가 모두 높다면 층수가 많은 신경망 구조나 다른 하이퍼파라미터를 선택한다.
  - 훈련 오차가 낮지만 검증 오차가 높은 경우 이 모델은 과적합 상태다. 정말 많은 훈련 데이터셋을 사용했다면 이런 경우는 보통 발생하지 않는다. 170,000회 이상의 바둑 대국과 수백만 개의 수를 사용하면 괜찮을 것이다.
  - 훈련 데이터 크기를 하드웨어 성능에 적합하도록 맞춘다. 한 세대를 훈련하는데 몇 시간이 걸린다면 그다지 즐겁지 않을 것이다. 대신 중간 크기의 데이터셋으로 여러번 훈련할 수 있는 모델을 찾은 후 이 모델을 가능한 한 큰 크기의 데이터셋에서 다시 한 번 훈련시켜보자.

직접 실험하고 성능 평가하기

- 훈련 및 검정 데이터로 성능 지표 평가하기
  - 좋은 GPU를 갖추고 있지 않다면 클라우드에서 모델을 훈련시킬 수 있다.
  - 실행 과정을 비교할 때 이전 실행 과정보다 나빠 보인다고 섣불리 실행을 멈추지 말자. 간혹 학습 과정이 다른 때보다 느려질 수도 있다. 하지만 결국은 다른 모델의 실행을 따라잡고 어떤 경우에는 더 좋은 성능을 보이기도 한다.

감사합니다!

스터디 듣느라 고생 많았습니다.