

# Structuring your first NLP project

Boseop Kim

August 15th, 2019

Tutorial of PyCon Kr 2019

# Intro

Deep Learning에 처음 입문할 때, 다루는 대부분의 예제 데이터는 대부분 수치형 데이터!



# Intro

NLP 공부를 시작하면 바로 눈에 보이는 차이는 데이터! 문자열 데이터를 수치로 바꾸는 적절한 전처리가 필요



★★★★★ 10 보니아... 앤디가 너에게 영원한 파트너를 맡겼는데 그렇게 잊히지게 두면 어떡하니...ㅠㅠ

GILGOO(dlr\*\*\*\*) | 2019.06.20 13:51 | 신고

공감 2193 비공감 75

★★★★★ 10 우디 수고했다.. 새 삶 찾은거 축하하고 꼭 행복해라.

주해진(skg\*\*\*\*) | 2019.06.20 11:18 | 신고

공감 1345 비공감 81

★★★★★ 10 처음 앤디 나올때 왜 눈물이 났는지 모르겠음..더이상 앤디의 장난감들이 아니라 보니아의 장난감이 라는게 아쉽고 나도 앤디가 그리워짐

조요초(dvdy\*\*\*\*) | 2019.06.20 14:15 | 신고

공감 1146 비공감 23

★★★★★ 10 **관람객** 자, 토이스토리 덕후분들은 손수건을 무조건 준비합니다. 안그러면 소중하게 구매한 토이 스토리 팝콘통에 눈물 다 채워갈 지 모르니까요.ㅠㅠ 사랑해 토이스토리♥

영(seyo\*\*\*\*) | 2019.06.20 10:36 | 신고

공감 1156 비공감 91

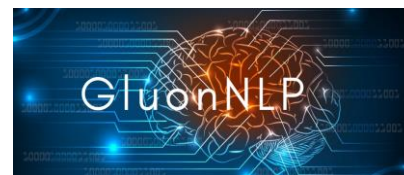
# Intro

전처리를 쉽게 하기위해 NLP 관련 package를 찾아보면...

1. 선택지가 너무 많아, 뭘 써야할 지 모르겠음 선택장애
2. High-level로 지원해서 내부를 알 수가 없음 customizing은 꿈
3. DL framework에 dependent한 경우 난그거안쓰는데
4. 내게 필요한 기능이 구현되어있지 않은 경우도 있음



torchtext

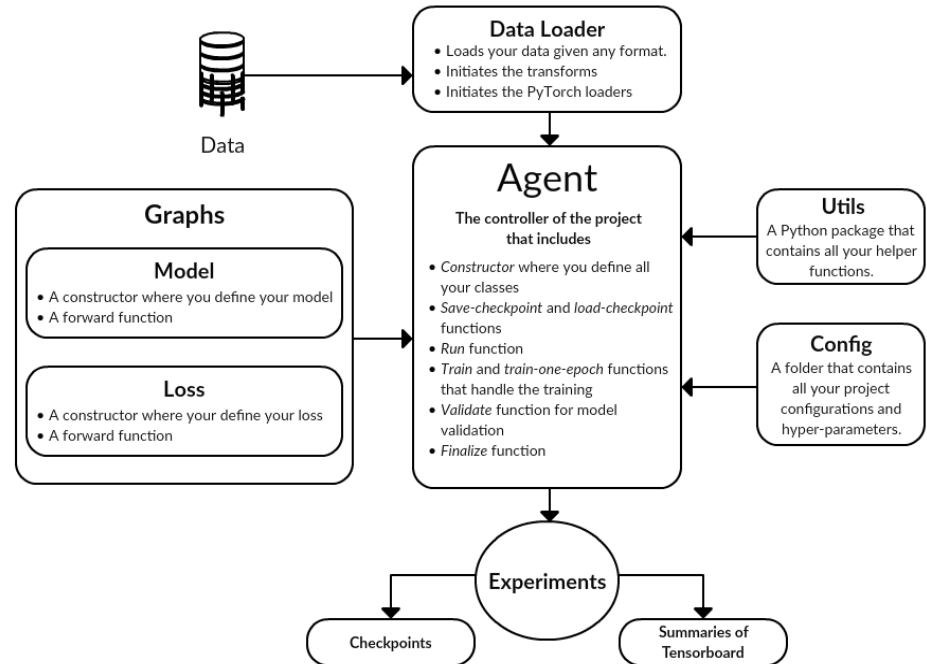


AllenNLP

# Intro

논문구현을 위해 프로젝트 구조화 및 glue code 작성까지 고려하면 시간이 많이듦  
→ 잘 구조화된 걸 참고하려해도... 너무 어렵다!

```
pytorch-template/  
├── train.py - main script to start training  
├── test.py - evaluation of trained model  
  
├── config.json - holds configuration for training  
├── parse_config.py - class to handle config file and cli options  
  
├── new_project.py - initialize new project with template files  
  
├── base/ - abstract base classes  
│   ├── base_data_loader.py  
│   ├── base_model.py  
│   └── base_trainer.py  
  
├── data_loader/ - anything about data loading goes here  
│   └── data_loaders.py  
  
├── data/ - default directory for storing input data  
  
├── model/ - models, losses, and metrics  
│   ├── model.py  
│   ├── metric.py  
│   └── loss.py  
  
├── saved/  
│   ├── models/ - trained models are saved here  
│   └── log/ - default logdir for tensorboardX and logging output  
  
├── trainer/ - trainers  
│   └── trainer.py  
  
├── logger/ - module for tensorboardX visualization and logging  
│   ├── visualization.py  
│   ├── logger.py  
│   └── logger_config.json  
  
└── utils/ - small utility functions  
    ├── util.py  
    └── ...
```



NLP 논문들을 구현하면서, 괜찮은 방법(전처리, 프로젝트  
구조화)이라고 여겼던 것들을 공유하려고 합니다.  
~~정답이 아닙니다~~

# Agenda

---

1. Preprocessing
2. Project structure
3. Summary

# Preprocessing



# Preprocessing

문자열 데이터를 수치화하는 과정은 대부분 아래와 같음, Split과 Transform만 Python으로 구현하면, Look up은 DL framework가 해결



# Preprocessing

문자열 데이터를 수치화하는 과정은 대부분 아래와 같음, Split과 Transform만 Python으로 구현하면, Look up은 DL framework가 해결

히어로 무비 중 가장 어둡지만 가장 참신했다.



## Split

사용하고 싶은 split (eg. 형태소분석기)

['히어로', '무비', '중', '가장', '어둡', '지만', '가장', '참신', '했', '다', '.']



## Transform

split을 토대로 데이터로부터 생성된 Vocabulary 활용

[8275, 3273, 7101, 511, 5305, 7192, 511, 7424, 8439, 1953, 46]



## Look up

random init된 matrix 또는 pretrained token vector에서 index에 해당하는 값을 가져옴

torch.tensor  
tf.tensor

# Preprocessing

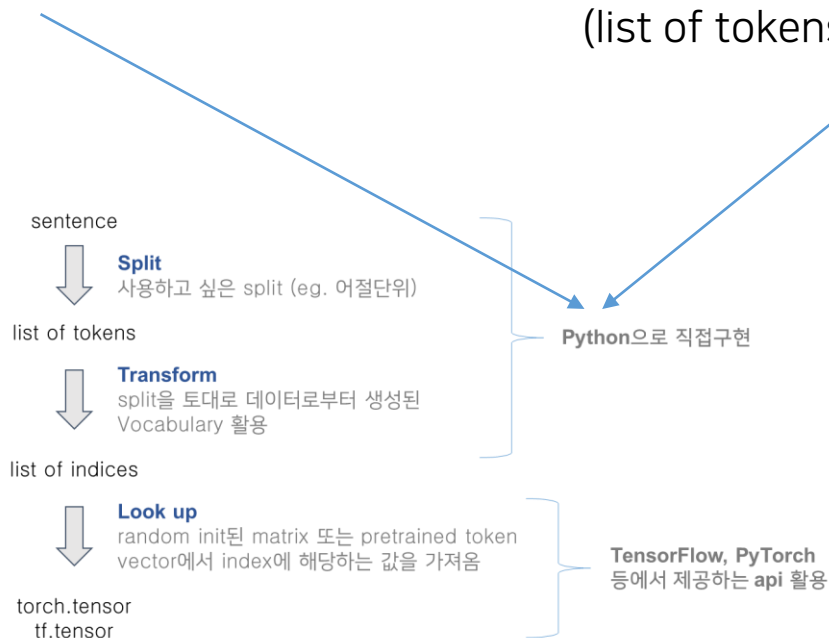
구현하고자 하는 논문, 적용하고자 하는 데이터에 따라서 각기 다른 `split`과 `transform`을 할 수 있어야함 → `Vocab class`, `Tokenizer class`를 구현

## Vocab class

- token을 받아서 index로 바꿔주는 역할
- pretrained vector를 활용하는 논문 구현 시, token의 embedding도 같이 가지고 있어야함

## Tokenizer class

- `split function`을 입력으로 받아 활용 (sentence → list of tokens)
- `Vocab class`의 instance를 입력으로 받아 해당 mapping 정보를 활용 (list of tokens → list of indices)



# Preprocessing

Tokenizer class를 composition으로 구현, split을 해주는 function과 Vocab class의 instance를 parameter로 받음 → wrapping해서 사용하는 형태

```
class Vocab:
    def __init__(self, list_of_tokens=None, padding_token='<pad>', unknown_token='<unk>',
                 bos_token='<bos>', eos_token='<eos>', reserved_tokens=None, unknown_token_idx=0):
        self._unknown_token = unknown_token
        self._padding_token = padding_token
        self._bos_token = bos_token
        self._eos_token = eos_token
        self._reserved_tokens = reserved_tokens
        self._special_tokens = []

        for tkn in [self._padding_token, self._bos_token, self._eos_token]:
            if tkn:
                self._special_tokens.append(tkn)

        if self._reserved_tokens:
            self._special_tokens.extend(self._reserved_tokens)
        if self._unknown_token:
            self._special_tokens.insert(unknown_token_idx, self._unknown_token)

        if list_of_tokens:
            self._special_tokens.extend(list(filter(lambda elm: elm not in self._special_tokens, list_of_tokens)))

        self.token_to_idx, self.idx_to_token = self._build(self._special_tokens)
        self._embedding = None

    def to_indices(self, tokens: Union[str, List[str]]) -> Union[int, List[int]]:
        if isinstance(tokens, list):
            return [self.token_to_idx[tkn] if tkn in self.token_to_idx else self.token_to_idx[self._unknown_token]
                    for tkn in tokens]
        else:
            return self.token_to_idx[tokens] if tokens in self.token_to_idx else \
                self.token_to_idx[self._unknown_token]

    def to_tokens(self, indices: Union[int, List[int]]) -> Union[str, List[str]]:
        if isinstance(indices, list):
            return [self.idx_to_token[idx] for idx in indices]
        else:
            return self.idx_to_token[indices]

    def _build(self, list_of_tokens):
        token_to_idx = {tkn: idx for idx, tkn in enumerate(list_of_tokens)}
        idx_to_token = {idx: tkn for idx, tkn in enumerate(list_of_tokens)}
        return token_to_idx, idx_to_token

    def __len__(self):
        return len(self.token_to_idx)
```

```
class Tokenizer:
    """Tokenizer class"""
    def __init__(self, vocab: Vocab, split_fn: Callable[[str], List[str]],
                 pad_fn: Callable[[List[int]], List[int]] = None) -> None:
        """Instantiating Tokenizer class

        Args:
            vocab (model.utils.Vocab): the instance of model.utils.Vocab created from specific split_fn
            split_fn (Callable): a function that can act as a splitter
            pad_fn (Callable): a function that can act as a padder
        """
        self._vocab = vocab
        self._split = split_fn
        self._pad = pad_fn

    def split(self, string: str) -> List[str]:
        list_of_tokens = self._split(string)
        return list_of_tokens

    def transform(self, list_of_tokens: List[str]) -> List[int]:
        list_of_indices = self._vocab.to_indices(list_of_tokens)
        list_of_indices = self._pad(list_of_indices) if self._pad else list_of_indices
        return list_of_indices

    def split_and_transform(self, string: str) -> List[int]:
        return self.transform(self.split(string))

    @property
    def vocab(self):
        return self._vocab
```

# Preprocessing

위와 같이 구현해둔 Vocabulary class, Tokenizer class는 아래와 같이 간단하게 활용할 수 있음

```
tokenizer = Tokenizer(vocab=vocab, split_fn=MeCab().morphs)
```

히어로 무비 중 가장 어둡지만 가장 참신했다.



## Split

사용하고 싶은 split (eg. 형태소분석기)

['히어로', '무비', '중', '가장', '어둡', '지만', '가장', '참신', '했', '다', '.']



## Transform

split을 토대로 데이터로부터 생성된 Vocabulary 활용

[8275, 3273, 7101, 511, 5305, 7192, 511, 7424, 8439, 1953, 46]



## Look up

random init된 matrix 또는 pretrained token vector에서 index에 해당하는 값을 가져옴

torch.tensor  
tf.tensor

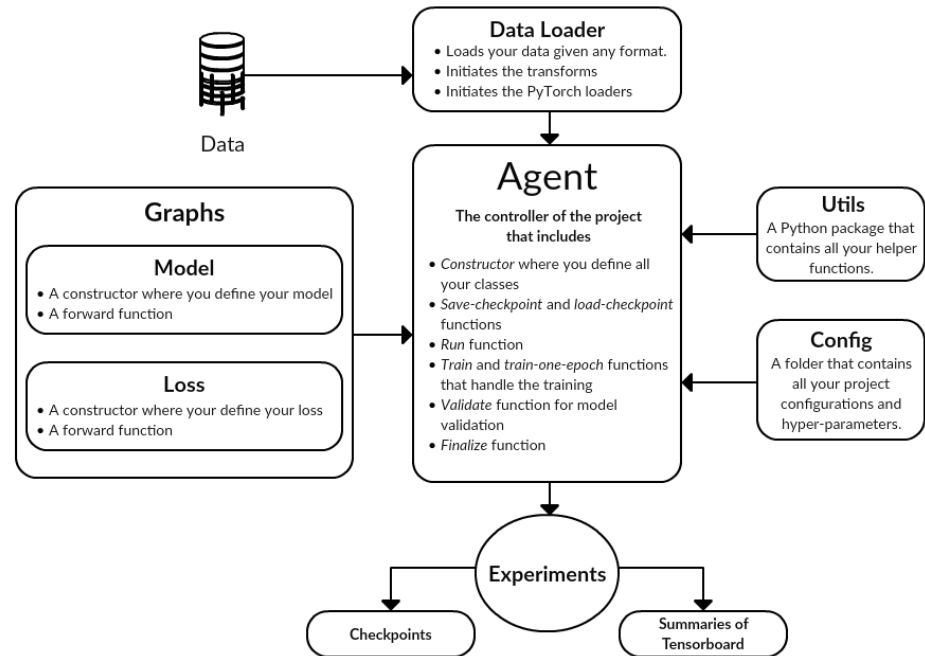
`tokenizer.split_and_transform`  
( '히어로 무비 중 가장 어둡지만 가장 참신했다.' )

# Project structure

# Project structure

잘 구조화된 template을 사용하면 좋지만, 처음 접할 시 매우 복잡하고 Training에 관련 없는 glue code도 존재 → **기본부터 시작하여 차근차근 확장**

```
pytorch-template/
├── train.py - main script to start training
├── test.py - evaluation of trained model
├──
├── config.json - holds configuration for training
├── parse_config.py - class to handle config file and cli options
├──
├── new_project.py - initialize new project with template files
├──
├── base/ - abstract base classes
│   ├── base_data_loader.py
│   ├── base_model.py
│   └── base_trainer.py
├──
├── data_loader/ - anything about data loading goes here
│   └── data_loaders.py
├──
├── data/ - default directory for storing input data
├──
├── model/ - models, losses, and metrics
│   ├── model.py
│   ├── metric.py
│   └── loss.py
├──
├── saved/
│   ├── models/ - trained models are saved here
│   └── log/ - default logdir for tensorboardX and logging output
├──
├── trainer/ - trainers
│   └── trainer.py
├──
├── logger/ - module for tensorboardX visualization and logging
│   ├── visualization.py
│   ├── logger.py
│   └── logger_config.json
├──
├── utils/ - small utility functions
│   ├── util.py
│   └── ...
```



# Project structure

대부분의 복잡한 template이라고 하더라도, 자세히 살펴보면 크게 아래에서 벗어나지는 않음 → [template](#)보다 필요한 부분에 집중

```
data/  
experiments/  
model/  
    data.py  
    net.py  
    ops.py  
    (metric.py) # 기본 api가 지원하면 metric.py는 구현 필요성 ↓  
    utils.py  
train.py  
evaluate.py  
utils.py  
build_vocab.py  
(search_hyperparams.py) # 구현자의 판단  
(synthesize_results.py) # 구현자의 판단
```



# Project structure - model

```
data/  
experiments/  
model/  
    data.py  
    net.py  
    ops.py  
    (metric.py)  
    utils.py  
train.py  
evaluate.py  
utils.py  
build_vocab.py  
(search_hyperparams.py)  
(synthesize_results.py)
```

# Project structure - model

model directory는 에서는 구현하고자 하는 논문에 제시된 architecture와 관련된 코드를 정리, 특히 `net.py`와 `ops.py`에 architecture와 관련된 코드를 정리

```
model/  
  data.py  
  net.py  
  ops.py  
  (metric.py) # 기본 api가 지원하면 metric.py는 구현 필요성 ↓  
  utils.py
```

- `data.py`: model을 training을 하기위한 data pipeline과 관련된 코드
- `net.py`: model architecture를 class로 정의  
(eg. SentenceCNN)
- `ops.py`: model architecture를 구현하기 위한 module을 class로 정의  
(eg. MultiChannelEmbedding, ConvolutionLayer, MaxOverTimePooling)
- `metric.py`: model의 성능을 measure하는 metric과 loss를 function으로 정의  
(eg. cross entropy, accuracy)
- `utils.py`: data pipeline 및 model과 관련 있는 utility들을 function 및 class로 정의  
(eg. Vocab, Tokenizer, PadSequence)

# Project structure – model/[net.py, ops.py]

net.py와 ops.py를 정리할 때 net.py에서 정의되는 architecture가 최대한 linear한 구조를 가질 수 있도록 ops.py에 module을 정리하는 게 중요

ops.py에 정리하는 원칙

- DL framework에서 제공하는 기본 api가 없음 (이미 있는 것을 구현하지 말 것)
- DL framework에서 제공하는 기본 api에 새로운 기능을 추가해서 구현해야 할 경우
- net.py에서 architecture를 linear하게 만들기 위해서만 구현

## Class SenCNN

Class MultiChannelEmbedding

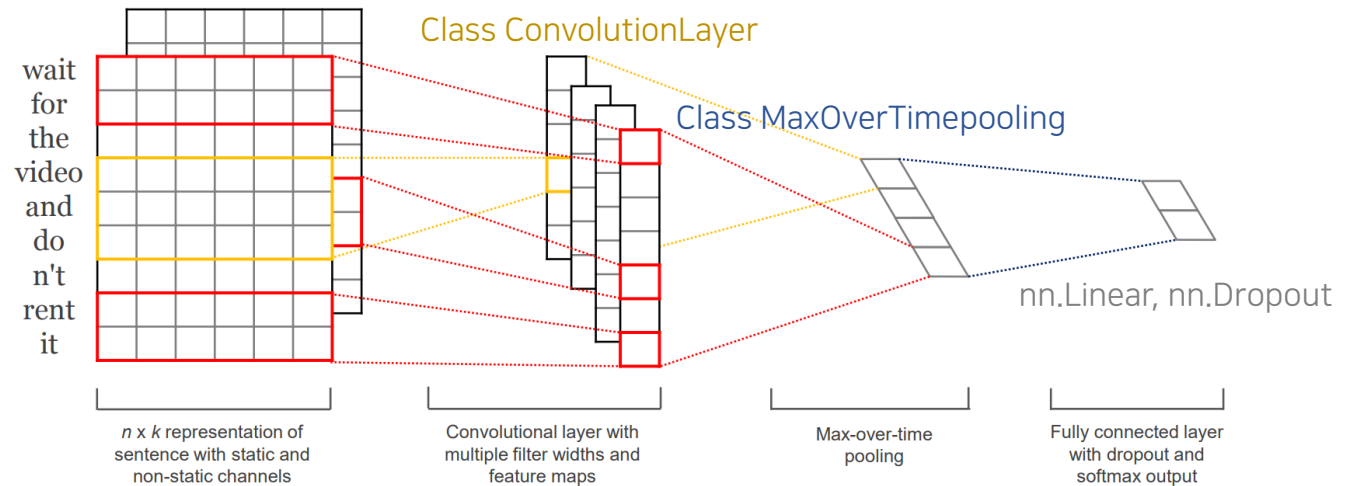


Figure 1: Model architecture with two channels for an example sentence.

# Project structure – model/[net.py, ops.py]

net.py와 ops.py를 정리할 때 net.py에서 정의되는 architecture가 최대한 linear한 구조를 가질 수 있도록 ops.py에 module을 정리하는 게 중요

## net.py

```
class SenCNN(nn.Module):
    """SenCNN class"""

    def __init__(self, num_classes: int, vocab: Vocab) -> None:
        """Instantiating SenCNN class

        Args:
            num_classes (int): the number of classes
            vocab (model.utils.Vocab): the instance of model.utils.Vocab
        """
        super(SenCNN, self).__init__()
        self.embedding = MultiChannelEmbedding(vocab)
        self.convolution = ConvolutionLayer(300, 300)
        self.pooling = MaxOverTimePooling()
        self.dropout = nn.Dropout()
        self.fc = nn.Linear(300, num_classes)

        self.apply(self._init_weights)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        fmap = self.embedding(x)
        fmap = self.convolution(fmap)
        feature = self.pooling(fmap)
        feature = self.dropout(feature)
        score = self.fc(feature)

        return score

    def _init_weights(self, layer) -> None:
        if isinstance(layer, nn.Conv1d):
            nn.init.kaiming_uniform_(layer.weight)
        elif isinstance(layer, nn.Linear):
            nn.init.xavier_normal_(layer.weight)
```

## ops.py

**Class MultiChannelEmbedding**

```
class MultiChannelEmbedding(nn.Module):
    """MultiChannelEmbedding class"""

    def __init__(self, vocab: Vocab) -> None:
        """Instantiating MultiChannelEmbedding class

        Args:
            vocab (model.utils.Vocab): the instance of model.utils.Vocab
        """
        super(MultiChannelEmbedding, self).__init__()
        self._static = nn.Embedding.from_pretrained(torch.from_numpy(vocab.embedding),
                                                    freeze=True, padding_idx=vocab.to_indices(vocab.padding_token))
        self._non_static = nn.Embedding.from_pretrained(torch.from_numpy(vocab.embedding),
                                                       freeze=False, padding_idx=vocab.to_indices(vocab.padding_token))

    def forward(self, x: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor]:
        static = self._static(x).permute(0, 2, 1)
        non_static = self._non_static(x).permute(0, 2, 1)
        return static, non_static
```

**Class ConvolutionLayer**

```
class ConvolutionLayer(nn.Module):
    """ConvolutionLayer class"""

    def __init__(self, in_channels: int, out_channels: int) -> None:
        """Instantiating ConvolutionLayer class

        Args:
            in_channels (int): the number of channels from input feature map
            out_channels (int): the number of channels from output feature map
        """
        super(ConvolutionLayer, self).__init__()
        self._tri_gram = nn.Conv1d(in_channels=in_channels, out_channels=out_channels // 3, kernel_size=3)
        self._tetra_gram = nn.Conv1d(in_channels=in_channels, out_channels=out_channels // 3, kernel_size=4)
        self._penta_gram = nn.Conv1d(in_channels=in_channels, out_channels=out_channels // 3, kernel_size=5)

    def forward(self, x: Tuple[torch.Tensor, torch.Tensor]) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
        static, non_static = x
        tri_fmap = F.relu(self._tri_gram(static)) + F.relu(self._tri_gram(non_static))
        tetra_fmap = F.relu(self._tetra_gram(static)) + F.relu(self._tetra_gram(non_static))
        penta_fmap = F.relu(self._penta_gram(static)) + F.relu(self._penta_gram(non_static))
        return tri_fmap, tetra_fmap, penta_fmap
```

**Class MaxOverTimepooling**

```
class MaxOverTimePooling(nn.Module):
    """MaxOverTimePooling class"""

    def forward(self, x: Tuple[torch.Tensor, torch.Tensor, torch.Tensor]) -> torch.Tensor:
        tri_fmap, tetra_fmap, penta_fmap = x
        fmap = torch.cat([tri_fmap.max(dim=-1)[0], tetra_fmap.max(dim=-1)[0], penta_fmap.max(dim=-1)[0]], dim=-1)
        return fmap
```

# Project structure – model/utis.py

utis.py에서는 model을 training하기 위한 data pipeline을 구현함에 있어서 필요한 전처리 관련 코드들을 작성

```
class Vocab:
    def __init__(self, list_of_tokens=None, padding_token='<pad>', unknown_token='<unk>',
                  bos_token='<bos>', eos_token='<eos>', reserved_tokens=None, unknown_token_idx=0):
        self._unknown_token = unknown_token
        self._padding_token = padding_token
        self._bos_token = bos_token
        self._eos_token = eos_token
        self._reserved_tokens = reserved_tokens
        self._special_tokens = []

        for tkn in [self._padding_token, self._bos_token, self._eos_token]:
            if tkn:
                self._special_tokens.append(tkn)

        if self._reserved_tokens:
            self._special_tokens.extend(self._reserved_tokens)
        if self._unknown_token:
            self._special_tokens.insert(unknown_token_idx, self._unknown_token)

        if list_of_tokens:
            self._special_tokens.extend(list(filter(lambda elm: elm not in self._special_tokens, list_of_tokens)))

        self.token_to_idx, self.idx_to_token = self._build(self._special_tokens)
        self.embedding = None

    def to_indices(self, tokens: Union[str, List[str]]) -> Union[int, List[int]]:
        if isinstance(tokens, list):
            return [self._token_to_idx[tkn] if tkn in self._token_to_idx else self._token_to_idx[self._unknown_token]
                    for tkn in tokens]
        else:
            return self._token_to_idx[tokens] if tokens in self._token_to_idx else \
                self._token_to_idx[self._unknown_token]

    def to_tokens(self, indices: Union[int, List[int]]) -> Union[str, List[str]]:
        if isinstance(indices, list):
            return [self._idx_to_token[idx] for idx in indices]
        else:
            return self._idx_to_token[indices]

    def _build(self, list_of_tokens):
        token_to_idx = {tkn: idx for idx, tkn in enumerate(list_of_tokens)}
        idx_to_token = {idx: tkn for idx, tkn in enumerate(list_of_tokens)}
        return token_to_idx, idx_to_token

    def __len__(self):
        return len(self._token_to_idx)
```

```
class Tokenizer:
    """Tokenizer class"""
    def __init__(self, vocab: Vocab, split_fn: Callable[[str], List[str]],
                  pad_fn: Callable[[List[int]], List[int]] = None) -> None:
        """Instantiating Tokenizer class

        Args:
            vocab (model.utis.Vocab): the instance of model.utis.Vocab created from specific split_fn
            split_fn (Callable): a function that can act as a splitter
            pad_fn (Callable): a function that can act as a padder
        """
        self._vocab = vocab
        self._split = split_fn
        self._pad = pad_fn

    def split(self, string: str) -> List[str]:
        list_of_tokens = self._split(string)
        return list_of_tokens

    def transform(self, list_of_tokens: List[str]) -> List[int]:
        list_of_indices = self._vocab.to_indices(list_of_tokens)
        list_of_indices = self._pad(list_of_indices) if self._pad else list_of_indices
        return list_of_indices

    def split_and_transform(self, string: str) -> List[int]:
        return self.transform(self.split(string))

    @property
    def vocab(self):
        return self._vocab

class PadSequence:
    def __init__(self, length: int, pad_val: int = 0, clip: bool = True) -> None:
        self._length = length
        self._pad_val = pad_val
        self._clip = clip

    def __call__(self, sample):
        sample_length = len(sample)
        if sample_length >= self._length:
            if self._clip and sample_length > self._length:
                return sample[:self._length]
            else:
                return sample
        else:
            return sample + [self._pad_val for _ in range(self._length - sample_length)]
```

# Project structure – model/data.py

data.py에서는 model을 training하기 위한 data pipeline과 관련된 코드를 포함, PyTorch의 경우 Dataset class를 정의하고 DataLoader는 기본 api 활용

## Dataset class

`torch.utils.data.Dataset` is an abstract class representing a dataset. Your custom dataset should inherit `Dataset` and override the following methods:

- `__len__` so that `len(dataset)` returns the size of the dataset.
- `__getitem__` to support the indexing such that `dataset[i]` can be used to get *i*th sample

Let's create a dataset class for our face landmarks dataset. We will read the csv in `__init__` but leave the reading of images to `__getitem__`. This is memory efficient because all the images are not stored in the memory at once but read as required.

Sample of our dataset will be a dict `{'image': image, 'landmarks': landmarks}`. Our dataset will take an optional argument `transform` so that any required processing can be applied on the sample. We will see the usefulness of `transform` in the next section.

```
class Corpus(Dataset):
    """Corpus class"""
    def __init__(self, filepath: str, transform_fn: Callable[[str], List[int]]) -> None:
        """Instantiating Corpus class

        Args:
            filepath (str): filepath
            transform_fn (Callable): a function that can act as a transformer
        """
        self._corpus = pd.read_csv(filepath, sep='\t').loc[:, ['document', 'label']]
        self._transform = transform_fn

    def __len__(self) -> int:
        return len(self._corpus)

    def __getitem__(self, idx: int) -> Tuple[torch.Tensor, torch.Tensor]:
        tokens2indices = torch.tensor(self._transform(self._corpus.iloc[idx]['document']))
        label = torch.tensor(self._corpus.iloc[idx]['label'])
        return tokens2indices, label
```

전처리함수를 parameter로 받아 활용하면 매우 구현이 용이!

from torch.nn.utils.data import DataLoader



```
# training
tr_ds = Corpus(data_config.tr, tokenizer.split_and_transform)
tr_dl = DataLoader(tr_ds, batch_size=model_config.batch_size, shuffle=True, num_workers=4, drop_last=True)
val_ds = Corpus(data_config.val, tokenizer.split_and_transform)
val_dl = DataLoader(val_ds, batch_size=model_config.batch_size)
```

# Project structure – data, experiments

```
data/  
experiments/  
model/  
    data.py  
    net.py  
    ops.py  
    (metric.py)  
    utils.py  
train.py  
evaluate.py  
utils.py  
build_vocab.py  
(search_hyperparams.py)  
(synthesize_results.py)
```

# Project structure – data, experiments

data directory는 dataset을 포함하고, experiments directory는 실험을 directory 단위로 포함, 각각의 directory에는 json 파일로 메타정보를 기록

data/

config.json


train.txt

val.txt

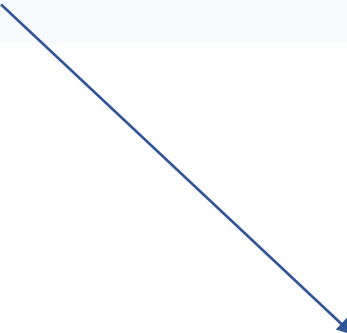
experiments/

base\_model/

config.json



```
{  
  "train": "data/train.txt",  
  "validation": "data/validation.txt",  
  "test": "data/test.txt",  
  "vocab": "data/vocab.pkl"  
}
```



```
{  
  "num_classes": 2,  
  "length": 70,  
  "epochs": 5,  
  "batch_size": 128,  
  "learning_rate": 1e-3,  
  "summary_step": 1000  
}
```



# Project structure – data, experiments

메타정보 (eg. config.json)을 토대로 training한 모델의 성능을 json 형태 (eg. summary.json)로 요약하고, 모델의 weight를 저장



checkpoint와 \*.json들은 어떻게 관리할 것인가?

**Config, CheckpointManager, SummaryManager class**

# Project structure – utils.py

```
data/  
experiments/  
model/  
    data.py  
    net.py  
    ops.py  
    (metric.py)  
    utils.py  
train.py  
evaluate.py  
utils.py  
build_vocab.py  
(search_hyperparams.py)  
(synthesize_results.py)
```

# Project structure – utils.py/Config

실험에 대한 메타정보를 json 파일로 관리하는 이유 → Python에서 parsing하면 dictionary object이므로 관리하기가 매우 쉬움

```
class Config:
    def __init__(self, json_path):
        with open(json_path, mode='r') as io:
            params = json.loads(io.read())
            self.__dict__.update(params)

    def save(self, json_path):
        with open(json_path, mode='w') as io:
            json.dump(self.__dict__, io, indent=4)

    def update(self, json_path):
        with open(json_path, mode='r') as io:
            params = json.loads(io.read())
            self.__dict__.update(params)

    @property
    def dict(self):
        return self.__dict__
```

json을 parsing하면 dictionary → class instance  
의 attribute로 관리할 수 있음

```
In[2]: from utils import Config
...:
...: config = Config('data/config.json')
...: print(config.__dict__)
...: print(config.train)
...: print(config.validation)
...:
```

```
{
    "train": "data/train.txt",
    "validation": "data/validation.txt",
    "test": "data/test.txt",
    "vocab": "data/vocab.pkl"
}
```

```
{'train': 'data/train.txt', 'validation': 'data/validation.txt', 'test': 'data/test.txt', 'vocab': 'data/vocab.pkl'}
data/train.txt
data/validation.txt
```

# Project structure – utils.py/SummaryManager

실험에 대한 성능정보 또한 json으로 관리, 실험을 directory 단위로 관리하기 때  
문에 입력은 directory 경로로 받음

```
class SummaryManager:
```

```
def __init__(self, model_dir):
    if not isinstance(model_dir, Path):
        model_dir = Path(model_dir)
    self._model_dir = model_dir
    self._summary = {}
```

```
def save(self, filename):
    with open(self._model_dir / filename, mode='w') as io:
        json.dump(self._summary, io, indent=4)
```

```
def load(self, filename):
    with open(self._model_dir / filename, mode='r') as io:
        metric = json.loads(io.read())
    self.update(metric)
```

```
def update(self, summary):
    self._summary.update(summary)
```

```
def reset(self):
    self._summary = {}
```

```
@property
```

```
def summary(self):
    return self._summary
```

train.py (일부)

training 시 계속 바뀌는 모델의 최고 성능을 간단하  
게 기록할 수 있음

```
⋮
checkpoint_manager = CheckpointManager(model_dir)
summary_manager = SummaryManager(model_dir)
best_val_loss = 1e+10

⋮

is_best = val_loss < best_val_loss

if is_best:
    state = {'epoch': epoch + 1,
            'model_state_dict': model.state_dict(),
            'opt_state_dict': opt.state_dict()}
    summary = {'tr': tr_summary, 'val': val_summary}

    summary_manager.update(summary)
    summary_manager.save('summary.json')
    checkpoint_manager.save_checkpoint(state, 'best.tar')

    best_val_loss = val_loss
```

# Project structure – utils.py/CheckpointManager

모델의 checkpoint (weight 등)을 관리하는 관리자는 실험을 directory 단위로 관리하기 때문에 입력은 directory 경로로 받음

```
class CheckpointManager:
```

```
def __init__(self, model_dir):  
    if not isinstance(model_dir, Path):  
        model_dir = Path(model_dir)  
    self._model_dir = model_dir
```

```
def save_checkpoint(self, state, filename):  
    torch.save(state, self._model_dir / filename)
```

```
def load_checkpoint(self, filename):  
    state = torch.load(self._model_dir / filename)  
    return state
```

train.py (일부)

training 시 계속 바뀌는 모델의 checkpoint (weight 등)를 저장

```
        :  
        checkpoint_manager = CheckpointManager(model_dir)  
        summary_manager = SummaryManager(model_dir)  
        best_val_loss = 1e+10  
        :  
        is_best = val_loss < best_val_loss  
  
        if is_best:  
            state = {'epoch': epoch + 1,  
                    'model_state_dict': model.state_dict(),  
                    'opt_state_dict': opt.state_dict()}  
            summary = {'tr': tr_summary, 'val': val_summary}  
  
            summary_manager.update(summary)  
            summary_manager.save('summary.json')  
            checkpoint_manager.save_checkpoint(state, 'best.tar')  
  
        best_val_loss = val_loss
```

# Project structure – build\_vocab.py, train.py, evaluate.py

```
data/  
experiments/  
model/  
    data.py  
    net.py  
    ops.py  
    (metric.py)  
    utils.py  
train.py  
evaluate.py  
utils.py  
build_vocab.py  
(search_hyperparams.py)  
(synthesize_results.py)
```

# Project structure – build\_vocab.py

model/utils.py에 정의한 Vocab class, Tokenizer class를 활용하여, 데이터에 맞는 Vocabulary를 생성 → config.json을 이에 맞게 update

data/

config.json

train.txt

val.txt

vocab.pkl

⋮

build\_vocab.py

python build\_vocab.py

training dataset을 토대로 Vocabulary를 생성,  
Vocabulary는 train.py, evaluate.py에서 활용

Update data/config.json

```
{  
  "train": "data/train.txt",  
  "validation": "data/validation.txt",  
  "test": "data/test.txt",  
}
```



```
{  
  "train": "data/train.txt",  
  "validation": "data/validation.txt",  
  "test": "data/test.txt",  
  "vocab": "data/vocab.pkl"  
}
```

# Project structure – train.py

train.py는 directory (eg. data, experiments /base\_model)의 경로를 input으로 받도록 parser를 작성, checkpoint 저장과 모델성능을 기록하도록 구현

```
parser = argparse.ArgumentParser()
parser.add_argument('--data_dir', default='data', help="Directory containing config.json of data")
parser.add_argument('--model_dir', default='experiments/base_model', help="Directory containing config.json of model")

if __name__ == '__main__':
    args = parser.parse_args()
    data_dir = Path(args.data_dir)
    model_dir = Path(args.model_dir)
    data_config = Config(json_path=data_dir / 'config.json')
    model_config = Config(json_path=model_dir / 'config.json')
```

⋮

```
if is_best:
    state = {'epoch': epoch + 1,
            'model_state_dict': model.state_dict(),
            'opt_state_dict': opt.state_dict()}
    summary = {'train': tr_summary, 'validation': val_summary}

    summary_manager.update(summary)
    summary_manager.save('summary.json')
    checkpoint_manager.save_checkpoint(state, 'best.tar')

    best_val_loss = val_loss
```



# Project structure – evaluate.py

evaluate.py는 directory 경로 (eg. data, experiments/base\_model)를 input으로 받도록 parser를 작성, 모델성능기록을 update하도록 구현

```
parser = argparse.ArgumentParser()
parser.add_argument('--data_dir', default='data', help="Directory containing config.json of data")
parser.add_argument('--model_dir', default='experiments/base_model', help="Directory containing config.json of model")
parser.add_argument('--restore_file', default='best', help="name of the file in --model_dir \
                    containing weights to load")
parser.add_argument('--data_name', default='test', help="name of the data in --data_dir to be evaluate")

if __name__ == '__main__':
    args = parser.parse_args()
    data_dir = Path(args.data_dir)
    model_dir = Path(args.model_dir)
    data_config = Config(json_path=data_dir / 'config.json')
    model_config = Config(json_path=model_dir / 'config.json')
```

⋮

```
# evaluation
summary_manager = SummaryManager(model_dir)
filepath = getattr(data_config, args.data_name)
ds = Corpus(filepath, tokenizer.split_and_transform)
dl = DataLoader(ds, batch_size=model_config.batch_size, num_workers=4)

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
model.to(device)

summary = evaluate(model, dl, {'loss': nn.CrossEntropyLoss(), 'acc': acc}, device)

summary_manager.load('summary.json')
summary_manager.update({'{}'.format(args.data_name): summary})
summary_manager.save('summary.json')
```

# Project structure – etc

```
data/  
experiments/  
model/  
    data.py  
    net.py  
    ops.py  
    (metric.py)  
    utils.py  
train.py  
evaluate.py  
utils.py  
build_vocab.py  
(search_hyperparams.py)  
(synthesize_results.py)
```

# Project structure – etc (model/metric.py)

task에 따라 loss와 metric을 정의, 예를 들어 classification task의 경우 accuracy를 함수를 metric.py에 정의하여 train.py, evaluate.py에서 활용

## train.py

```
from model.metric import evaluate, acc
:
for epoch in tqdm(range(model_config.epochs), desc='epochs'):

    tr_loss = 0
    tr_acc = 0

    model.train()
    for step, mb in tqdm(enumerate(tr_dl), desc='steps', total=len(tr_dl)):
        x_mb, y_mb = map(lambda elm: elm.to(device), mb)

        :

    tr_summary = {'loss': tr_loss, 'acc': tr_acc}
    val_summary = evaluate(model, val_dl, {'loss': loss_fn, 'acc': acc}, device)
    scheduler.step(val_summary['loss'])
    tqdm.write('epoch : {}, tr_loss: {:.3f}, val_loss: '
               '{:.3f}, tr_acc: {:.2%}, val_acc: {:.2%}'.format(epoch + 1, tr_summary['loss'],
                                                                val_summary['loss'], tr_summary['acc'],
                                                                val_summary['acc']))
```

## metric.py

task에 맞게 performance를 측정할 수 있도록 함수를 정의

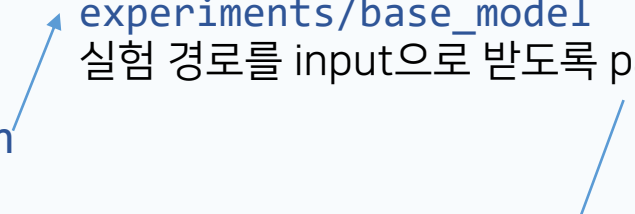
```
def acc(yhat, y):
    with torch.no_grad():
        yhat = yhat.max(dim=1)[1]
        acc = (yhat == y).float().mean()
    return acc
```

# Project structure – etc (synthesize\_results.py)

필요에 따라 experiments에 있는 각각의 실험 director의 summary.json을 parsing하는 코드를 작성 → 사람이 보기 쉽게 정리하는 용도

```
data/  
  config.json  
  train.txt  
  val.txt  
experiments/  
  base_model/  
    config.json  
    summary.json  
    best.tar
```

`python synthesize_results.py --model_dir experiments/base_model`  
실험 경로를 input으로 받도록 parser코드를 포함



원하는 형태로 정리 (eg. Table)

# Project structure – etc (search\_hyperparams.py)

각각의 hyper parameter 설정에 따라 Config instance의 attribute를 update 하고, experiments에 실험 디렉토리를 생성하고 학습의 결과를 저장하도록 작성

```
import argparse
import os
from subprocess import check_call
import sys

import utils

PYTHON = sys.executable
parser = argparse.ArgumentParser()
parser.add_argument('--parent_dir', default='experiments/learning_rate',
                    help='Directory containing params.json')
parser.add_argument('--data_dir', default='data/small', help="Directory containing the dataset")

:

if __name__ == "__main__":
    # Load the "reference" parameters from parent_dir json file
    args = parser.parse_args()
    json_path = os.path.join(args.parent_dir, 'params.json')
    assert os.path.isfile(json_path), "No json configuration file found at {}".format(json_path)
    params = utils.Params(json_path)

    # Perform hypersearch over one parameter
    learning_rates = [1e-4, 1e-3, 1e-2]

    for learning_rate in learning_rates:
        # Modify the relevant parameter in params
        params.learning_rate = learning_rate

        # Launch job (name has to be unique)
        job_name = "learning_rate_{}".format(learning_rate)
        launch_training_job(args.parent_dir, args.data_dir, job_name, params)
```

# Summary

# Summary

```
data/ # dataset과 각 dataset의 경로를 기록한 config.json
experiments/ # 실험 별 director에 config.json, summary.json,
              checkpoint 파일
model/ # model 구현에 필요한 것들 정리
      data.py # data loader (data pipeline) 관련 코드
      net.py # architecture를 정의
      ops.py # architecture 구현에 필요한 operation 정의
             (metric.py) # task에 맞는 loss, metric 정의
      utils.py # data loader와 관련된 utility 코드
train.py # model 학습
evaluate.py # model 평가
utils.py # model의 configuration, performance, checkpoint
          관련 코드
build_vocab.py # NLP에서 필요한 Vocabulary 생성코드
(search_hyperparams.py) # hyper parameter search
(synthesize_results.py) # 실험결과를 사람이 보기 좋게 parsing
```



E-mail: [bsk0130@gmail.com](mailto:bsk0130@gmail.com)  
Github: [github.com/aisolab](https://github.com/aisolab)  
Blog: [aisolab.github.io](https://aisolab.github.io)