langml

Release 0.0.1

Sean Lee

Jun 29, 2022

GETTING STARTED

LangML (Language ModeL) is a Keras-based and TensorFlow-backend language model toolkit, which provides mainstream pre-trained language models, e.g., BERT/RoBERTa/ALBERT, and their downstream application models.

CHAPTER

ONE

INSTALLATION

1.1 From pip

You can install or upgrade langml/langml-cli from pip:

```
pip install -U langml
```

1.2 From Github

You can also install the latest langml/langml-cli from Github:

```
git clone https://github.com/4AI/langml.git
cd langml
python setup.py install
```

CHAPTER

TWO

USE LANGML-CLI TO QUICKLY TRAIN BASELINE MODELS

You can use LangML-CLI to train baseline models quickly. You don't need to write any code and just need to prepare the dataset in a specific format.

You can train various baseline models using *langml-cli*:

```
$ langml-cli --help
Usage: langml [OPTIONS] COMMAND [ARGS]...

LangML client

Options:
--version  Show the version and exit.
--help     Show this message and exit.

Commands:
baseline  LangML Baseline client
```

2.1 Text Classification

Prepare your data into *JSONLines* format, and provide text and label field in each line, for example:

```
{"text": "this is sentence1", "label": "label1"}
{"text": "this is sentence2", "label": "label2"}
```

1. Bert

```
$ langml-cli baseline clf bert --help
Usage: langml baseline clf bert [OPTIONS]

Options:
--backbone TEXT              specify backbone: bert | roberta | albert
--epoch INTEGER              epochs
--batch_size INTEGER         batch size
--learning_rate FLOAT        learning rate
--max_len INTEGER            max len
--lowercase                  do lowercase
--tokenizer_type TEXT        specify tokenizer type from [`wordpiece`,
                            `sentencepiece`]
```

(continues on next page)

(continued from previous page)

```
--monitor TEXT                monitor for keras callback
--early_stop INTEGER          patience to early stop
--use_micro                   whether to use micro metrics
--config_path TEXT            bert config path  [required]
--ckpt_path TEXT              bert checkpoint path  [required]
--vocab_path TEXT             bert vocabulary path  [required]
--train_path TEXT             train path  [required]
--dev_path TEXT               dev path  [required]
--test_path TEXT              test path
--save_dir TEXT               dir to save model  [required]
--verbose INTEGER             0 = silent, 1 = progress bar, 2 = one line per
                             epoch

--distributed_training        distributed training
--distributed_strategy TEXT   distributed training strategy
--help                        Show this message and exit.
```

2. BiLSTM

```
$ langml-cli baseline clf bilstm --help
Usage: langml baseline clf bilstm [OPTIONS]

Options:
--epoch INTEGER               epochs
--batch_size INTEGER          batch size
--learning_rate FLOAT         learning rate
--embedding_size INTEGER      embedding size
--hidden_size INTEGER         hidden size of lstm
--max_len INTEGER             max len
--lowercase                   do lowercase
--tokenizer_type TEXT         specify tokenizer type from [`wordpiece`,
                             `sentencepiece`]

--monitor TEXT                monitor for keras callback
--early_stop INTEGER          patience to early stop
--use_micro                   whether to use micro metrics
--vocab_path TEXT             vocabulary path  [required]
--train_path TEXT             train path  [required]
--dev_path TEXT               dev path  [required]
--test_path TEXT              test path
--save_dir TEXT               dir to save model  [required]
--verbose INTEGER             0 = silent, 1 = progress bar, 2 = one line per
                             epoch

--with_attention              apply attention mechanism
--distributed_training        distributed training
--distributed_strategy TEXT   distributed training strategy
--help                        Show this message and exit.
```

3. TextCNN

```
$ langml-cli baseline clf textcnn --help
```

(continues on next page)

(continued from previous page)

```
Usage: langml baseline clf textcnn [OPTIONS]

Options:
--epoch INTEGER              epochs
--batch_size INTEGER         batch size
--learning_rate FLOAT        learning rate
--embedding_size INTEGER     embedding size
--filter_size INTEGER        filter size of convolution
--max_len INTEGER            max len
--lowercase                  do lowercase
--tokenizer_type TEXT        specify tokenizer type from [`wordpiece`,
                            `sentencepiece`]

--monitor TEXT               monitor for keras callback
--early_stop INTEGER         patience to early stop
--use_micro                  whether to use micro metrics
--vocab_path TEXT            vocabulary path  [required]
--train_path TEXT            train path  [required]
--dev_path TEXT              dev path  [required]
--test_path TEXT             test path
--save_dir TEXT              dir to save model  [required]
--verbose INTEGER            0 = silent, 1 = progress bar, 2 = one line per
                            epoch

--distributed_training       distributed training
--distributed_strategy TEXT  distributed training strategy
--help                       Show this message and exit.
```

2.2 Named Entity Recognition

Prepare your data in the following format:

use “t” to separate entity segment and entity type in a sentence, and use “nn” to separate different sentences.

An English example:

```
I like    O
apples  Fruit

I like    O
pineapples  Fruit
```

A Chinese example:

```
  O
    LOC

  O
    LOC
```

1) BERT-CRF

```
$ langml-cli baseline ner bert-crf --help
Usage: langml baseline ner bert-crf [OPTIONS]

Options:
    --backbone TEXT              specify backbone: bert | roberta | albert
    --epoch INTEGER              epochs
    --batch_size INTEGER         batch size
    --learning_rate FLOAT        learning rate
    --dropout_rate FLOAT         dropout rate
    --max_len INTEGER            max len
    --lowercase                  do lowercase
    --tokenizer_type TEXT        specify tokenizer type from [`wordpiece`,
                                `sentencepiece`]
    --config_path TEXT           bert config path  [required]
    --ckpt_path TEXT             bert checkpoint path  [required]
    --vocab_path TEXT            bert vocabulary path  [required]
    --train_path TEXT            train path  [required]
    --dev_path TEXT              dev path  [required]
    --test_path TEXT             test path
    --save_dir TEXT              dir to save model  [required]
    --monitor TEXT               monitor for keras callback
    --early_stop INTEGER         patience to early stop
    --verbose INTEGER            0 = silent, 1 = progress bar, 2 = one line per
                                epoch
    --distributed_training       distributed training
    --distributed_strategy TEXT  distributed training strategy
    --help                       Show this message and exit.
```

2) LSTM-CRF

```
$  langml-cli baseline ner lstm-crf --help
Usage: langml baseline ner lstm-crf [OPTIONS]

Options:
    --epoch INTEGER              epochs
    --batch_size INTEGER         batch size
    --learning_rate FLOAT        learning rate
    --dropout_rate FLOAT         dropout rate
    --embedding_size INTEGER     embedding size
    --hidden_size INTEGER        hidden size
    --max_len INTEGER            max len
    --lowercase                  do lowercase
    --tokenizer_type TEXT        specify tokenizer type from [`wordpiece`,
                                `sentencepiece`]
    --vocab_path TEXT            vocabulary path  [required]
    --train_path TEXT            train path  [required]
    --dev_path TEXT              dev path  [required]
    --test_path TEXT             test path
    --save_dir TEXT              dir to save model  [required]
    --monitor TEXT               monitor for keras callback
    --early_stop INTEGER         patience to early stop
    --verbose INTEGER            0 = silent, 1 = progress bar, 2 = one line per
                                epoch
```

(continues on next page)

(continued from previous page)

```
    --distributed_training       distributed training
    --distributed_strategy TEXT  distributed training strategy
    --help                       Show this message and exit.
```

2.3 Contrastive Learning

Prepare your data into *JSONLines* format:

a) for evaulation, should include *text_left*, *text_right*, and *label* fields

```
{"text_left": "text left1", "text_right": "text right1", "label": "0/1"}
{"text_left": "text left1", "text_right": "text right2", "label": "0/1"}
```

b) no need to evaluate, just provide *text* field.

```
{"text": "this is a text1"}
{"text": "this is a text2"}
```

1. simcse

```
$ langml-cli baseline contrastive simcse --help
Usage: langml baseline contrastive simcse [OPTIONS]

Options:
    --backbone TEXT              specify backbone: bert | roberta | albert
    --epoch INTEGER              epochs
    --batch_size INTEGER         batch size
    --learning_rate FLOAT        learning rate
    --dropout_rate FLOAT         dropout rate
    --temperature FLOAT          temperature
    --pooling_strategy TEXT      specify pooling_strategy from ["cls", "first-
                                last-avg", "last-avg"]
    --max_len INTEGER            max len
    --early_stop INTEGER         patience of early stop
    --monitor TEXT               metrics monitor
    --lowercase                  do lowercase
    --tokenizer_type TEXT        specify tokenizer type from [`wordpiece`,
                                `sentencepiece`]
    --config_path TEXT           bert config path  [required]
    --ckpt_path TEXT             bert checkpoint path  [required]
    --vocab_path TEXT            bert vocabulary path  [required]
    --train_path TEXT            train path  [required]
    --test_path TEXT             test path
    --save_dir TEXT              dir to save model  [required]
    --verbose INTEGER            0 = silent, 1 = progress bar, 2 = one line per
                                epoch

    --apply_aeda                 apply AEDA to augment data
    --aeda_language TEXT         specify AEDA language, ["EN", "CN"]
    --do_evaluate                do evaluation
    --distributed_training       distributed training
```

(continues on next page)

(continued from previous page)

```
    --distributed_strategy TEXT  distributed training strategy
    --help                       Show this message and exit.
```

2.4 Text Matching

Prepare your data into *JSONLines* format, three fields *text_left*, *text_right*, and *label* are required.

```
{"text_left": "text left1", "text_right": "text right1", "label": "label1"}
{"text_left": "text left1", "text_right": "text right2", "label": "label2"}
```

1. sentence bert

For the regression task, the label should be a float value or an integer. For the classification task, the label should be an integer or a string value.

```
$ langml-cli baseline matching sbert --help

Usage: langml baseline matching sbert [OPTIONS]

Options:
    --backbone TEXT              specify backbone: bert | roberta | albert
    --epoch INTEGER              epochs
    --batch_size INTEGER         batch size
    --learning_rate FLOAT        learning rate
    --dropout_rate FLOAT         dropout rate
    --task TEXT                  specify task from ["regression",
                                "classification"]
    --pooling_strategy TEXT      specify pooling_strategy from ["cls", "mean",
                                "max"]
    --max_len INTEGER            max len
    --early_stop INTEGER         patience of early stop
    --monitor TEXT               metrics monitor
    --lowercase                  do lowercase
    --tokenizer_type TEXT        specify tokenizer type from [`wordpiece`,
                                `sentencepiece`]
    --config_path TEXT           bert config path  [required]
    --ckpt_path TEXT             bert checkpoint path  [required]
    --vocab_path TEXT            bert vocabulary path  [required]
    --train_path TEXT            train path  [required]
    --dev_path TEXT              dev path  [required]
    --test_path TEXT             test path
    --save_dir TEXT              dir to save model  [required]
    --verbose INTEGER            0 = silent, 1 = progress bar, 2 = one line per
                                epoch

    --distributed_training       distributed training
    --distributed_strategy TEXT  distributed training strategy
    --help                       Show this message and exit.
```

CHAPTER

THREE

EXAMPLES OF FINETUNEING

To finetune a model, you need to prepare pretrained language models (PLMs). Currently, LangML supports BERT/RoBERTa/ALBERT PLMs. You can download PLMs from google-research/bert , google-research/albert , Chinese RoBERTa etc.

3.1 1. Prepare datasets

You need to use specific tokenizers in terms of PLMs to initialize a tokenizer and convert texts to vocabulary indices. LangML wraps huggingface/tokenizers and google/sentencepiece to provide a uniform interface. Specifically, you can initialize a WordPiece tokenizer via *langml.tokenizer.WPTokenizer*, and initialize a sentencepiece tokenizer via *langml.tokenizer.SPTokenizer*.

```
from langml import keras, L
from langml.tokenizer import WPTokenizer


vocab_path = '/path/to/vocab.txt'
tokenizer = WPTokenizer(vocab_path)
# specify max token length
tokenizer.enable_trunction(max_length=512)


class DataLoader:
   def __init__(self, tokenizer):
      # define initializer here
      self.tokenizer = tokenizer

   def __iter__(self, data):
      # define your data generator here
      for text, label in data:
         tokenized = self.tokenizer.encode(text)
         token_ids = tokenized.ids
         segment_ids = tokenized.segment_ids
         # ...
```

3.2 2. Build models

You can use *langml.plm.load_bert* to load a BERT/RoBERTa model, and use *langml.plm.load_albert* to load an ALBERT model.

```
from langml import keras, L
from langml.plm import load_bert

config_path = '/path/to/bert_config.json'
ckpt_path = '/path/to/bert_model.ckpt'
vocab_path = '/path/to/vocab.txt'

bert_model, bert_instance = load_bert(config_path, ckpt_path)
# get CLS representation
cls_output = L.Lambda(lambda x: x[:, 0])(bert_model.output)
output = L.Dense(2, activation='softmax',
                 kernel_intializer=bert_instance.initializer)(cls_output)
train_model = keras.Model(bert_model.input, cls_output)
train_model.summary()
train_model.compile(loss='categorical_crossentropy', optimizer=keras.optimizer.Adam(1e-
↪5))
```

3.3 3. Train and Eval

After defining the data loader and model, you can train and evaluate your model as most Keras models do.

CHAPTER

FOUR

EXAMPLES OF PROMPT-BASED TUNING

Prompt-based tuning is the latest paradigm to adapt PLMs to downstream NLP tasks, which embeds a textual template into the input text and directly uses the MLM task of PLMs to train models.

Currently support:

- PTuning: GPT Understands, Too

4.1 Prompt-based Classification

There are three steps to build a prompt-based classifier.

1. Define a template

```
from langml.prompt import Template
from langml.tokenizer import WPTokenizer

vocab_path = '/path/to/vocab.txt'

tokenizer = WPTokenizer(vocab_path, lowercase=True)
template = Template(
    #  must specify tokens that are defined in the vocabulary, and the mask token is␣
↪required
    template=['it', 'was', '[MASK]', '.'],
    # must specify tokens that are defined in the vocabulary.
    label_tokens_map={
        'positive': ['good'],
        'negative': ['bad', 'terrible']
    },
    tokenizer=tokenizer
)
```

2. Defina a prompt-based model

```
from langml.prompt import PTuniningPrompt, PTuningForClassification

bert_config_path = '/path/to/bert_config.json'
bert_ckpt_path = '/path/to/bert_model.ckpt'

prompt_model = PTuniningPrompt('bert', bert_config_path, bert_ckpt_path,
```

(continues on next page)

(continued from previous page)

```
                                template, freeze_plm=False, learning_rate=5e-5, encoder=
↪'lstm')
prompt_classifier = PTuningForClassification(prompt_model, tokenizer)
```

3. Train on dataset

```
data = [('I do not like this food', 'negative'),
        ('I hate you', 'negative'),
        ('I like you', 'positive'),
        ('I like this food', 'positive')]

X = [d for d, _ in data]
y = [l for _, l in data]

prompt_classifier.fit(X, y, X, y, batch_size=2, epoch=50, model_path='best_model.weight')
# load pretrained model
# prompt_classifier.load('best_model.weight')
print("pred", prompt_classifier.predict('I hate you'))
```

For more examples visit langml/examples

CHAPTER

FIVE

HOW TO TRAIN PLMS DISTRIBUTEDLY?

To train distributedly, you need to use *tensorflow.keras*. First, you need to define an environment variable *TF_KERAS* and assign *1* to it, for example, *export TF_KERAS=1* for Linux. Then manually restore PLMs weights after model compiling, as follows:

```
from langml import keras, L
from langml.plm import load_bert

config_path = '/path/to/bert_config.json'
ckpt_path = '/path/to/bert_model.ckpt'
vocab_path = '/path/to/vocab.txt'

# lazy resotre
bert_model, bert_instance, restore_weight_callback = load_bert(config_path, ckpt_path,␣
↪lazy_restore=True)
# get CLS representation
cls_output = L.Lambda(lambda x: x[:, 0])(bert_model.output)
output = L.Dense(2, activation='softmax',
                 kernel_intializer=bert_instance.initializer)(cls_output)
train_model = keras.Model(bert_model.input, cls_output)
train_model.summary()
train_model.compile(loss='categorical_crossentropy', optimizer=keras.optimizer.Adam(1e-
↪5))
# restore weights
restore_weight_callback(bert_model)
```

CHAPTER

SIX

API REFERENCE

This page contains auto-generated API reference documentation[1].

6.1 langml

6.1.1 Subpackages

langml.baselines

Subpackages

langml.baselines.clf

Submodules

langml.baselines.clf.bert

Module Contents

Classes

BertClassifier

class langml.baselines.clf.bert.**BertClassifier**(*config_path: str*, *ckpt_path: str*, *params:* langml.baselines.Parameters, *backbone: str = 'roberta'*)

Bases: *langml.baselines.BaselineModel*

build_model(*self*, *lazy_restore=False*) → langml.tensor_typing.Models

[1] Created with sphinx-autoapi

langml.baselines.clf.bilstm

Module Contents

Classes

BiLSTMClassifier	

class langml.baselines.clf.bilstm.**BiLSTMClassifier**(*params:* langml.baselines.Parameters, *with_attention: bool = False*)

Bases: *langml.baselines.BaselineModel*

build_model(*self*) → langml.tensor_typing.Models

langml.baselines.clf.cli

Module Contents

Functions

train(model_instance: object, params: langml.baselines.Parameters, epoch: int, save_dir: str, train_path: str, dev_path: str, test_path: str, vocab_path: str, tokenizer_type: str, lowercase: bool, max_len: int, batch_size: int, distributed_training: bool, distributed_strategy: str, use_micro: bool, monitor: str, early_stop: int, verbose: int)	
clf()	classification command line tools
bert(backbone: str, epoch: int, batch_size: int, learning_rate: float, max_len: Optional[int], lowercase: bool, tokenizer_type: Optional[str], monitor: str, early_stop: int, use_micro: bool, config_path: str, ckpt_path: str, vocab_path: str, train_path: str, dev_path: str, test_path: str, save_dir: str, verbose: int, distributed_training: bool, distributed_strategy: str)	
textcnn(epoch: int, batch_size: int, learning_rate: float, embedding_size: int, filter_size: int, max_len: Optional[int], lowercase: bool, tokenizer_type: Optional[str], monitor: str, early_stop: int, use_micro: bool, vocab_path: str, train_path: str, dev_path: str, test_path: str, save_dir: str, verbose: int, distributed_training: bool, distributed_strategy: str)	
bilstm(epoch: int, batch_size: int, learning_rate: float, embedding_size: int, hidden_size: int, max_len: Optional[int], lowercase: bool, tokenizer_type: Optional[str], monitor: str, early_stop: int, use_micro: bool, vocab_path: str, train_path: str, dev_path: str, test_path: str, save_dir: str, verbose: int, with_attention: bool, distributed_training: bool, distributed_strategy: str)	

langml.baselines.clf.cli.**train**(*model_instance: object*, *params:* langml.baselines.Parameters, *epoch: int*, *save_dir: str*, *train_path: str*, *dev_path: str*, *test_path: str*, *vocab_path: str*, *tokenizer_type: str*, *lowercase: bool*, *max_len: int*, *batch_size: int*, *distributed_training: bool*, *distributed_strategy: str*, *use_micro: bool*, *monitor: str*, *early_stop: int*, *verbose: int*)

langml.baselines.clf.cli.**clf**()

classification command line tools

langml.baselines.clf.cli.**bert**(*backbone: str*, *epoch: int*, *batch_size: int*, *learning_rate: float*, *max_len: Optional[int]*, *lowercase: bool*, *tokenizer_type: Optional[str]*, *monitor: str*, *early_stop: int*, *use_micro: bool*, *config_path: str*, *ckpt_path: str*, *vocab_path: str*, *train_path: str*, *dev_path: str*, *test_path: str*, *save_dir: str*, *verbose: int*, *distributed_training: bool*, *distributed_strategy: str*)

langml.baselines.clf.cli.**textcnn**(*epoch: int*, *batch_size: int*, *learning_rate: float*, *embedding_size: int*, *filter_size: int*, *max_len: Optional[int]*, *lowercase: bool*, *tokenizer_type: Optional[str]*, *monitor: str*, *early_stop: int*, *use_micro: bool*, *vocab_path: str*, *train_path: str*, *dev_path: str*, *test_path: str*, *save_dir: str*, *verbose: int*, *distributed_training: bool*, *distributed_strategy: str*)

langml.baselines.clf.cli.**bilstm**(*epoch: int*, *batch_size: int*, *learning_rate: float*, *embedding_size: int*, *hidden_size: int*, *max_len: Optional[int]*, *lowercase: bool*, *tokenizer_type: Optional[str]*, *monitor: str*, *early_stop: int*, *use_micro: bool*, *vocab_path: str*, *train_path: str*, *dev_path: str*, *test_path: str*, *save_dir: str*, *verbose: int*, *with_attention: bool*, *distributed_training: bool*, *distributed_strategy: str*)

langml.baselines.clf.dataloader

Module Contents

Classes

DataLoader
TFDataLoader

class langml.baselines.clf.dataloader.**DataLoader**(*data: List*, *tokenizer: object*, *label2id: Dict*, *batch_size: int = 32*, *is_bert: bool = True*)

Bases: *langml.baselines.BaseDataLoader*

__len__(*self*) → int

static load_data(*fpath: str*, *build_vocab: bool = False*) → List

make_iter(*self*, *random: bool = False*)

class langml.baselines.clf.dataloader.**TFDataLoader**(*data: List*, *tokenizer: object*, *label2id: Dict*, *batch_size: int = 32*, *is_bert: bool = True*)

Bases: *DataLoader*

make_iter(*self*, *random: bool = False*)

__call__(*self*, *random: bool = False*)

langml.baselines.clf.textcnn

Module Contents

Classes

TextCNNClassifier

class langml.baselines.clf.textcnn.**TextCNNClassifier**(*params:* langml.baselines.Parameters)

Bases: *langml.baselines.BaselineModel*

build_model(*self*) → langml.tensor_typing.Models

Package Contents

Classes

Infer

Functions

compute_detail_metrics(infer: object, datas: List, use_micro=False) → Tuple[float, float, Union[str, Dict]]

Attributes

TF_VERSION
Models

langml.baselines.clf.**TF_VERSION**

langml.baselines.clf.**Models**

class langml.baselines.clf.**Infer**(*model: langml.tensor_typing.Models*, *tokenizer: object*, *id2label: Dict*, *is_bert: bool = True*)

__call__(*self*, *text: str*)

langml.baselines.clf.**compute_detail_metrics**(*infer: object*, *datas: List*, *use_micro=False*) → Tuple[float, float, Union[str, Dict]]

langml.baselines.contrastive

Subpackages

langml.baselines.contrastive.simcse

Submodules

langml.baselines.contrastive.simcse.dataloder

Module Contents

Classes

DataLoader

TFDataLoader

class langml.baselines.contrastive.simcse.dataloder.**DataLoader**(*data: List*, *tokenizer: object*, *batch_size: int = 32*)

Bases: *langml.baselines.BaseDataLoader*

__len__(*self*) → int

static load_data(*fpath: str*, *apply_aeda: bool = True*, *aeda_tokenize: Callable = whitespace_tokenize*, *aeda_language: str = 'EN'*) → Tuple[List[Tuple[str, str]], List[Tuple[str, str, int]]]

Parameters

- **fpath** – str, path of data
- **apply_aeda** – bool, whether to apply the AEDA technique to augment data, default True
- **aeda_tokenize** – Callable, specify aeda tokenize function, it works when set apply_aeda=True
- **aeda_language** – str, specifying the language, it works when set apply_aeda=True

make_iter(*self*, *random: bool = False*)

class langml.baselines.contrastive.simcse.dataloder.**TFDataLoader**(*data: List*, *tokenizer: object*, *batch_size: int = 32*)

Bases: *DataLoader*

make_iter(*self*, *random: bool = False*)

__call__(*self*, *random: bool = False*)

langml.baselines.contrastive.simcse.model

Module Contents

Classes

SimCSE

Functions

simcse_loss(y_true, y_pred)	

langml.baselines.contrastive.simcse.model.**simcse_loss**(*y_true*, *y_pred*)

class langml.baselines.contrastive.simcse.model.**SimCSE**(*config_path: str*, *ckpt_path: str*, *params:* langml.baselines.Parameters, *backbone: str = 'roberta'*)

Bases: *langml.baselines.BaselineModel*

get_pooling_output(*self*, *model: langml.tensor_typing.Models*, *output_index: int*, *pooling_strategy: str = 'cls'*) → langml.tensor_typing.Tensors

get pooling output :param model: keras.Model, BERT model :param output_index: int, specify output index of feedforward layer. :param pooling_strategy: str, specify pooling strategy from ['cls', 'first-last-avg', 'last-avg'], default *cls*

build_model(*self*, *pooling_strategy: str = 'cls'*, *lazy_restore: bool = False*) → langml.tensor_typing.Models

Package Contents

Classes

DataLoader	
TFDataLoader	
SimCSE	

class langml.baselines.contrastive.simcse.**DataLoader**(*data: List*, *tokenizer: object*, *batch_size: int = 32*)

Bases: *langml.baselines.BaseDataLoader*

__len__(*self*) → int

static load_data(*fpath: str*, *apply_aeda: bool = True*, *aeda_tokenize: Callable = whitespace_tokenize*, *aeda_language: str = 'EN'*) → Tuple[List[Tuple[str, str]], List[Tuple[str, str, int]]]

Parameters

- **fpath** – str, path of data
- **apply_aeda** – bool, whether to apply the AEDA technique to augment data, default True
- **aeda_tokenize** – Callable, specify aeda tokenize function, it works when set apply_aeda=True
- **aeda_language** – str, specifying the language, it works when set apply_aeda=True

make_iter(*self*, *random: bool = False*)

class langml.baselines.contrastive.simcse.**TFDataLoader**(*data: List*, *tokenizer: object*, *batch_size: int = 32*)

Bases: *DataLoader*

make_iter(*self*, *random: bool = False*)

__call__(*self*, *random: bool = False*)

class langml.baselines.contrastive.simcse.**SimCSE**(*config_path: str*, *ckpt_path: str*, *params:* langml.baselines.Parameters, *backbone: str = 'roberta'*)

Bases: *langml.baselines.BaselineModel*

get_pooling_output(*self*, *model: langml.tensor_typing.Models*, *output_index: int*, *pooling_strategy: str = 'cls'*) → langml.tensor_typing.Tensors

get pooling output :param model: keras.Model, BERT model :param output_index: int, specify output index of feedforward layer. :param pooling_strategy: str, specify pooling strategy from ['cls', 'first-last-avg', 'last-avg'], default *cls*

build_model(*self*, *pooling_strategy: str = 'cls'*, *lazy_restore: bool = False*) → langml.tensor_typing.Models

Submodules

langml.baselines.contrastive.cli

Module Contents

Functions

contrastive()	contrastive learning command line tools
simcse(backbone: str, epoch: int, batch_size: int, learning_rate: float, dropout_rate: float, temperature: float, pooling_strategy: str, max_len: Optional[int], early_stop: int, monitor: str, lowercase: bool, tokenizer_type: Optional[str], config_path: str, ckpt_path: str, vocab_path: str, train_path: str, test_path: str, save_dir: str, verbose: int, apply_aeda: bool, aeda_language: str, do_evaluate: bool, distributed_training: bool, distributed_strategy: str)	

langml.baselines.contrastive.cli.**contrastive**()

contrastive learning command line tools

langml.baselines.contrastive.cli.**simcse**(*backbone: str*, *epoch: int*, *batch_size: int*, *learning_rate: float*, *dropout_rate: float*, *temperature: float*, *pooling_strategy: str*, *max_len: Optional[int]*, *early_stop: int*, *monitor: str*, *lowercase: bool*, *tokenizer_type: Optional[str]*, *config_path: str*, *ckpt_path: str*, *vocab_path: str*, *train_path: str*, *test_path: str*, *save_dir: str*, *verbose: int*, *apply_aeda: bool*, *aeda_language: str*, *do_evaluate: bool*, *distributed_training: bool*, *distributed_strategy: str*)

langml.baselines.contrastive.utils

Module Contents

Functions

aeda_augment(words: List[str], ratio: float = 0.3, language: str = 'EN') → str	AEDAAn Easier Data Augmentation Technique for Text Classification
whitespace_tokenize(text: str) → List[str]	

Attributes

CN_PUNCTUATIONS
EN_PUNCTUATIONS

langml.baselines.contrastive.utils.**CN_PUNCTUATIONS = ['', '', '', '', '']**

langml.baselines.contrastive.utils.**EN_PUNCTUATIONS = ['.', ',', '!', '?', ';', ':']**

langml.baselines.contrastive.utils.**aeda_augment**(*words: List[str]*, *ratio: float = 0.3*, *language: str = 'EN'*) → str

AEDAAn Easier Data Augmentation Technique for Text Classification :param text: str, input text :param ratio: float, ratio to add punctuation randomly :param language: str, specify language from [‘EN’, ‘CN’], default EN

langml.baselines.contrastive.utils.**whitespace_tokenize**(*text: str*) → List[str]

langml.baselines.matching

Subpackages

langml.baselines.matching.sbert

Submodules

langml.baselines.matching.sbert.dataloder

Module Contents

Classes

DataLoader
TFDataLoader

class langml.baselines.matching.sbert.dataloder.**DataLoader**(*data: List*, *tokenizer: object*, *batch_size: int = 32*)

Bases: *langml.baselines.BaseDataLoader*

__len__(*self*) → int

static load_data(*fpath: str*, *build_vocab: bool = False*, *label2idx: Optional[Dict] = None*) → Union[List[Tuple[str, str, int]], Tuple[List[Tuple[str, str, int]], Dict]]

Parameters

- **fpath** – str, path of data
- **build_vocab** – bool, whether to build vocabulary
- **label2idx** – Optional[Dict], label to index dict

make_iter(*self*, *random: bool = False*)

class langml.baselines.matching.sbert.dataloder.**TFDataLoader**(*data: List*, *tokenizer: object*, *batch_size: int = 32*)

Bases: *DataLoader*

make_iter(*self*, *random: bool = False*)

__call__(*self*, *random: bool = False*)

langml.baselines.matching.sbert.model

Module Contents

Classes

SentenceBert

class langml.baselines.matching.sbert.model.**SentenceBert**(*config_path: str*, *ckpt_path: str*, *params: langml.baselines.Parameters*, *backbone: str = 'roberta'*)

Bases: *langml.baselines.BaselineModel*

get_pooling_output(*self*, *model: langml.tensor_typing.Models*, *output_index: int*, *pooling_strategy: str = 'cls'*) → langml.tensor_typing.Tensors

get pooling output :param model: keras.Model, BERT model :param output_index: int, specify output index of feedforward layer. :param pooling_strategy: str, specify pooling strategy from ['cls', 'first-last-avg', 'last-avg'], default *cls*

build_model(*self*, *task: str = 'regression'*, *pooling_strategy: str = 'cls'*, *lazy_restore: bool = False*) → langml.tensor_typing.Models

Package Contents

Classes

DataLoader
TFDataLoader
SentenceBert

class langml.baselines.matching.sbert.**DataLoader**(*data: List*, *tokenizer: object*, *batch_size: int = 32*)

Bases: *langml.baselines.BaseDataLoader*

__len__(*self*) → int

static load_data(*fpath: str*, *build_vocab: bool = False*, *label2idx: Optional[Dict] = None*) → Union[List[Tuple[str, str, int]], Tuple[List[Tuple[str, str, int]], Dict]]

Parameters

- **fpath** – str, path of data
- **build_vocab** – bool, whether to build vocabulary
- **label2idx** – Optional[Dict], label to index dict

make_iter(*self*, *random: bool = False*)

class langml.baselines.matching.sbert.**TFDataLoader**(*data: List*, *tokenizer: object*, *batch_size: int = 32*)

Bases: *DataLoader*

make_iter(*self*, *random: bool = False*)

__call__(*self*, *random: bool = False*)

class langml.baselines.matching.sbert.**SentenceBert**(*config_path: str*, *ckpt_path: str*, *params: langml.baselines.Parameters*, *backbone: str = 'roberta'*)

Bases: *langml.baselines.BaselineModel*

get_pooling_output(*self*, *model: langml.tensor_typing.Models*, *output_index: int*, *pooling_strategy: str = 'cls'*) → langml.tensor_typing.Tensors

get pooling output :param model: keras.Model, BERT model :param output_index: int, specify output index of feedforward layer. :param pooling_strategy: str, specify pooling strategy from ['cls', 'first-last-avg', 'last-avg'], default *cls*

build_model(*self*, *task: str = 'regression'*, *pooling_strategy: str = 'cls'*, *lazy_restore: bool = False*) → langml.tensor_typing.Models

Submodules

langml.baselines.matching.cli

Module Contents

Functions

matching()	text matching command line tools
sbert(backbone: str, epoch: int, batch_size: int, learning_rate: float, dropout_rate: float, task: str, pooling_strategy: str, max_len: Optional[int], early_stop: int, monitor: str, lowercase: bool, tokenizer_type: Optional[str], config_path: str, ckpt_path: str, vocab_path: str, train_path: str, dev_path: str, test_path: str, save_dir: str, verbose: int, distributed_training: bool, distributed_strategy: str)	

langml.baselines.matching.cli.**matching**()

text matching command line tools

langml.baselines.matching.cli.**sbert**(*backbone: str*, *epoch: int*, *batch_size: int*, *learning_rate: float*, *dropout_rate: float*, *task: str*, *pooling_strategy: str*, *max_len: Optional[int]*, *early_stop: int*, *monitor: str*, *lowercase: bool*, *tokenizer_type: Optional[str]*, *config_path: str*, *ckpt_path: str*, *vocab_path: str*, *train_path: str*, *dev_path: str*, *test_path: str*, *save_dir: str*, *verbose: int*, *distributed_training: bool*, *distributed_strategy: str*)

langml.baselines.ner

Submodules

langml.baselines.ner.bert_crf

Module Contents

Classes

BertCRF

class langml.baselines.ner.bert_crf.**BertCRF**(*config_path: str*, *ckpt_path: str*, *params:* langml.baselines.Parameters, *backbone: str = 'roberta'*)

Bases: *langml.baselines.BaselineModel*

build_model(*self*, *lazy_restore=False*) → langml.tensor_typing.Models

langml.baselines.ner.cli

Module Contents

Functions

train(model_instance: object, params: langml.baselines.Parameters, epoch: int, save_dir: str, train_path: str, dev_path: str, test_path: str, vocab_path: str, tokenizer_type: str, lowercase: bool, max_len: int, batch_size: int, distributed_training: bool, distributed_strategy: str, monitor: str, early_stop: int, verbose: int)	
ner()	ner command line tools
bert_crf(backbone: str, epoch: int, batch_size: int, learning_rate: float, dropout_rate: float, max_len: Optional[int], lowercase: bool, tokenizer_type: Optional[str], config_path: str, ckpt_path: str, vocab_path: str, train_path: str, dev_path: str, test_path: str, save_dir: str, monitor: str, early_stop: int, verbose: int, distributed_training: bool, distributed_strategy: str)	
lstm_crf(epoch: int, batch_size: int, learning_rate: float, dropout_rate: float, embedding_size: int, hidden_size: int, max_len: Optional[int], lowercase: bool, tokenizer_type: Optional[str], vocab_path: str, train_path: str, dev_path: str, test_path: str, save_dir: str, monitor: str, early_stop: int, verbose: int, distributed_training: bool, distributed_strategy: str)	

langml.baselines.ner.cli.**train**(*model_instance: object*, *params:* langml.baselines.Parameters, *epoch: int*, *save_dir: str*, *train_path: str*, *dev_path: str*, *test_path: str*, *vocab_path: str*, *tokenizer_type: str*, *lowercase: bool*, *max_len: int*, *batch_size: int*, *distributed_training: bool*, *distributed_strategy: str*, *monitor: str*, *early_stop: int*, *verbose: int*)

langml.baselines.ner.cli.**ner**()

ner command line tools

langml.baselines.ner.cli.**bert_crf**(*backbone: str*, *epoch: int*, *batch_size: int*, *learning_rate: float*, *dropout_rate: float*, *max_len: Optional[int]*, *lowercase: bool*, *tokenizer_type: Optional[str]*, *config_path: str*, *ckpt_path: str*, *vocab_path: str*, *train_path: str*, *dev_path: str*, *test_path: str*, *save_dir: str*, *monitor: str*, *early_stop: int*, *verbose: int*, *distributed_training: bool*, *distributed_strategy: str*)

langml.baselines.ner.cli.**lstm_crf**(*epoch: int*, *batch_size: int*, *learning_rate: float*, *dropout_rate: float*, *embedding_size: int*, *hidden_size: int*, *max_len: Optional[int]*, *lowercase: bool*, *tokenizer_type: Optional[str]*, *vocab_path: str*, *train_path: str*, *dev_path: str*, *test_path: str*, *save_dir: str*, *monitor: str*, *early_stop: int*, *verbose: int*, *distributed_training: bool*, *distributed_strategy: str*)

langml.baselines.ner.dataloader

Module Contents

Classes

DataLoader
TFDataLoader

class langml.baselines.ner.dataloader.**DataLoader**(*data: List*, *tokenizer: object*, *label2id: Dict*, *batch_size: int = 32*, *max_len: Optional[int] = None*, *is_bert: bool = True*)

Bases: *langml.baselines.BaseDataLoader*

encode_data(*self*, *data: List[Tuple[str, str]]*) → Tuple[List[int], List[int], List[int]]

static load_data(*fpath: str*, *build_vocab: bool = False*) → List

__len__(*self*) → int

make_iter(*self*, *random: bool = False*)

class langml.baselines.ner.dataloader.**TFDataLoader**(*data: List*, *tokenizer: object*, *label2id: Dict*, *batch_size: int = 32*, *max_len: Optional[int] = None*, *is_bert: bool = True*)

Bases: *DataLoader*

make_iter(*self*, *random: bool = False*)

__call__(*self*, *random: bool = False*)

langml.baselines.ner.lstm_crf

Module Contents

Classes

LSTMCRF

class langml.baselines.ner.lstm_crf.**LSTMCRF**(*params:* langml.baselines.Parameters)

Bases: *langml.baselines.BaselineModel*

build_model(*self*) → langml.tensor_typing.Models

Package Contents

Classes

Infer	

Functions

bio_decode(tags: List[str]) → List[Tuple[int, int, str]]	Decode BIO tags
compute_detail_metrics(model: langml.tensor_typing.Models, dataloader: object, id2label: Dict, is_bert: bool = True)	

Attributes

TF_VERSION	
Models	
re_split	

langml.baselines.ner.**TF_VERSION**

langml.baselines.ner.**bio_decode**(*tags: List[str]*) → List[Tuple[int, int, str]]

Decode BIO tags

Examples: >>> bio_decode(['B-PER', 'I-PER', 'O', 'B-ORG', 'I-ORG', 'I-ORG']) >>> [(0, 1, 'PER'), (3, 5, 'ORG')]

langml.baselines.ner.**Models**

langml.baselines.ner.**re_split**

class langml.baselines.ner.**Infer**(*model: langml.tensor_typing.Models*, *tokenizer: object*, *id2label: Dict*, *max_chunk_len: Optional[int] = None*, *is_bert: bool = True*)

decode_one(*self*, *text: str*, *base_position: int = 0*)

Parameters

- **text** (-) – str
- **base_position** (-) – int

Returns

[(entity, start, end, entity_type)]

Return type

list of tuple

__call__(*self*, *text: str*)

langml.baselines.ner.**compute_detail_metrics**(*model: langml.tensor_typing.Models*, *dataloader: object*, *id2label: Dict*, *is_bert: bool = True*)

Submodules

langml.baselines.cli

Module Contents

Functions

baseline()	LangML Baseline client

langml.baselines.cli.**baseline**()

LangML Baseline client

Package Contents

Classes

BaselineModel	
BaseDataLoader	
Parameters	Hyper-Parameters

class langml.baselines.**BaselineModel**

abstract build_model(*self*, **args*, ***kwargs*)

class langml.baselines.**BaseDataLoader**

abstract static load_data()

abstract make_iter(*self*, *random: bool = False*)

abstract __len__(*self*)

__call__(*self*, *random: bool = False*)

class langml.baselines.**Parameters**(*data: Optional[Dict] = None*)

Hyper-Parameters

_wrap(*self*, *value: Any*)

add(*self*, *name*, *value*)

langml.common

Subpackages

langml.common.evaluator

Submodules

langml.common.evaluator.spearman

Module Contents

Classes

SpearmanEvaluator

class langml.common.evaluator.spearman.**SpearmanEvaluator**(*encoder: langml.tensor_typing.Models*, *tokenizer:* langml.tokenizer.Tokenizer)

compute_corrcoef(*self*, *data: List[Tuple[str, str, int]]*) → float

Package Contents

Classes

SpearmanEvaluator

class langml.common.evaluator.**SpearmanEvaluator**(*encoder: langml.tensor_typing.Models*, *tokenizer:* langml.tokenizer.Tokenizer)

compute_corrcoef(*self*, *data: List[Tuple[str, str, int]]*) → float

langml.layers

Submodules

langml.layers.attention

Module Contents

Classes

SelfAttention	
SelfAdditiveAttention	
ScaledDotProductAttention	ScaledDotProductAttention
MultiHeadAttention	MultiHeadAttention
GatedAttentionUnit	Gated Attention Unit

class langml.layers.attention.**SelfAttention**(*attention_units: Optional[int] = None*, *return_attention: bool = False*, *is_residual: bool = False*, *attention_activation: langml.tensor_typing.Activation = 'relu'*, *attention_epsilon: float = 10000000000.0*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_attention_bias: bool = True*, *attention_penalty_weight: float = 0.0*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

_attention_penalty(*self*, *attention: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

class langml.layers.attention.**SelfAdditiveAttention**(*attention_units: Optional[int] = None*, *return_attention: bool = False*, *is_residual: bool = False*, *attention_activation: langml.tensor_typing.Activation = 'relu'*, *attention_epsilon: float = 10000000000.0*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_attention_bias: bool = True*, *attention_penalty_weight: float = 0.0*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

_attention_penalty(*self*, *attention: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

class langml.layers.attention.**ScaledDotProductAttention**(*return_attention: bool = False*, *history_only: bool = False*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

ScaledDotProductAttention

$Attention(Q, K, V) = softmax(frac{Q K^T}{sqrt{d_k}}) V$

https://arxiv.org/pdf/1706.03762.pdf

get_config(*self*) → dict

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]] = None*, ***kwargs*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

class langml.layers.attention.**MultiHeadAttention**(*head_num: int*, *return_attention: bool = False*, *attention_activation: langml.tensor_typing.Activation = 'relu'*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_attention_bias: bool = True*, *history_only: bool = False*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

MultiHeadAttention https://arxiv.org/pdf/1706.03762.pdf

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

static _reshape_to_batches(*x*, *head_num*)

static _reshape_attention_from_batches(*x*, *head_num*)

static _reshape_from_batches(*x*, *head_num*)

static _reshape_mask(*mask*, *head_num*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

compute_output_shape(*self*, *input_shape: Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

class langml.layers.attention.**GatedAttentionUnit**(*attention_units: int*, *attention_activation: langml.tensor_typing.Activation = 'relu'*, *attention_normalizer: langml.tensor_typing.Activation = relu2*, *attention_epsilon: float = 10000000000.0*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_attention_bias: bool = True*, *use_attention_scale: bool = True*, *use_relative_position: bool = True*, *use_offset: bool = True*, *use_scale: bool = True*, *is_residual: bool = True*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

Gated Attention Unit https://arxiv.org/abs/2202.10447

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

apply_rotary_position_embeddings(*self*, *sinusoidal: langml.tensor_typing.Tensors*, **tensors*)

apply RoPE modified from: https://github.com/bojone/bert4keras/blob/master/bert4keras/backend.py#L310

attn(*self*, *x: langml.tensor_typing.Tensors*, *v: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

langml.layers.crf

Module Contents

Classes

CRF

class langml.layers.crf.**CRF**(*output_dim: int*, *sparse_target: bool = True*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *sequence_lengths: Optional[langml.tensor_typing.Tensors] = None*, *training: Optional[Union[bool, int]] = None*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

property loss(*self*) → Callable

property accuracy(*self*) → Callable

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

property trans(*self*) → langml.tensor_typing.Tensors

transition parameters

get_config(*self*) → dict

static get_custom_objects() → dict

langml.layers.layer_norm

Module Contents

Classes

LayerNorm

class langml.layers.layer_norm.**LayerNorm**(*center: bool = True*, *scale: bool = True*, *epsilon: float = 1e-07*, *gamma_initializer: langml.tensor_typing.Initializer = 'ones'*, *gamma_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *gamma_constraint: Optional[langml.tensor_typing.Constraint] = None*, *beta_initializer: langml.tensor_typing.Initializer = 'zeros'*, *beta_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *beta_constraint: Optional[langml.tensor_typing.Constraint] = None*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, ***kwargs*) → langml.tensor_typing.Tensors

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[langml.tensor_typing.Tensors, None]

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

langml.layers.layers

Module Contents

Classes

AbsolutePositionEmbedding	
SineCosinePositionEmbedding	Sine Cosine Position Embedding.
ScaleOffset	Scale Offset
ConditionalLayerNormalization	Conditional Layer Normalization

class langml.layers.layers.**AbsolutePositionEmbedding**(*input_dim: int*, *output_dim: int*, *mode: str = 'add'*, *embeddings_initializer: langml.tensor_typing.Initializer = 'uniform'*, *embeddings_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *embeddings_constraint: Optional[langml.tensor_typing.Constraint] = None*, *mask_zero: bool = False*, ***kwargs*)

Bases: langml.L.Layer

get_config(*self*) → dict

static get_custom_objects() → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

call(*self*, *inputs: langml.tensor_typing.Tensors*, ***kwargs*) → langml.tensor_typing.Tensors

class langml.layers.layers.**SineCosinePositionEmbedding**(*mode: str = 'add'*, *output_dim: Optional[int] = None*, ***kwargs*)

Bases: langml.L.Layer

Sine Cosine Position Embedding. https://arxiv.org/pdf/1706.03762

get_config(*self*)

static get_custom_objects() → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

class langml.layers.layers.**ScaleOffset**(*scale: bool = True*, *offset: bool = True*, ***kwargs*)

Bases: langml.L.Layer

Scale Offset

get_config(*self*)

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*)

call(*self*, *inputs: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

class langml.layers.layers.**ConditionalLayerNormalization**(*center: bool = True*, *epsilon: Optional[float] = None*, *scale: bool = True*, *offset: bool = True*, ***kwargs*)

Bases: langml.L.Layer

Conditional Layer Normalization https://arxiv.org/abs/2108.00449

get_config(*self*)

build(*self*, *input_shapes: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*)

call(*self*, *inputs: List[langml.tensor_typing.Tensors]*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

Package Contents

Classes

CRF	
LayerNorm	
AbsolutePositionEmbedding	
SineCosinePositionEmbedding	Sine Cosine Position Embedding.
ScaleOffset	Scale Offset
ConditionalLayerNormalization	Conditional Layer Normalization
SelfAttention	
SelfAdditiveAttention	
ScaledDotProductAttention	ScaledDotProductAttention
MultiHeadAttention	MultiHeadAttention
GatedAttentionUnit	Gated Attention Unit

Attributes

TF_KERAS	
custom_objects	

langml.layers.**TF_KERAS**

class langml.layers.**CRF**(*output_dim: int*, *sparse_target: bool = True*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *sequence_lengths: Optional[langml.tensor_typing.Tensors] = None*, *training: Optional[Union[bool, int]] = None*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

property loss(*self*) → Callable

property accuracy(*self*) → Callable

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

property trans(*self*) → langml.tensor_typing.Tensors

transition parameters

get_config(*self*) → dict

static get_custom_objects() → dict

class langml.layers.**LayerNorm**(*center: bool = True*, *scale: bool = True*, *epsilon: float = 1e-07*, *gamma_initializer: langml.tensor_typing.Initializer = 'ones'*, *gamma_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *gamma_constraint: Optional[langml.tensor_typing.Constraint] = None*, *beta_initializer: langml.tensor_typing.Initializer = 'zeros'*, *beta_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *beta_constraint: Optional[langml.tensor_typing.Constraint] = None*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, ***kwargs*) → langml.tensor_typing.Tensors

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[langml.tensor_typing.Tensors, None]

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

class langml.layers.**AbsolutePositionEmbedding**(*input_dim: int*, *output_dim: int*, *mode: str = 'add'*, *embeddings_initializer: langml.tensor_typing.Initializer = 'uniform'*, *embeddings_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *embeddings_constraint: Optional[langml.tensor_typing.Constraint] = None*, *mask_zero: bool = False*, ***kwargs*)

Bases: langml.L.Layer

get_config(*self*) → dict

static get_custom_objects() → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

call(*self*, *inputs: langml.tensor_typing.Tensors*, ***kwargs*) → langml.tensor_typing.Tensors

class langml.layers.**SineCosinePositionEmbedding**(*mode: str = 'add'*, *output_dim: Optional[int] = None*, ***kwargs*)

Bases: langml.L.Layer

Sine Cosine Position Embedding. https://arxiv.org/pdf/1706.03762

get_config(*self*)

static get_custom_objects() → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

class langml.layers.**ScaleOffset**(*scale: bool = True*, *offset: bool = True*, ***kwargs*)

Bases: langml.L.Layer

Scale Offset

get_config(*self*)

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*)

call(*self*, *inputs: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

class langml.layers.**ConditionalLayerNormalization**(*center: bool = True*, *epsilon: Optional[float] = None*, *scale: bool = True*, *offset: bool = True*, ***kwargs*)

Bases: langml.L.Layer

Conditional Layer Normalization https://arxiv.org/abs/2108.00449

get_config(*self*)

build(*self*, *input_shapes: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*)

call(*self*, *inputs: List[langml.tensor_typing.Tensors]*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

class langml.layers.**SelfAttention**(*attention_units: Optional[int] = None*, *return_attention: bool = False*, *is_residual: bool = False*, *attention_activation: langml.tensor_typing.Activation = 'relu'*, *attention_epsilon: float = 10000000000.0*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_attention_bias: bool = True*, *attention_penalty_weight: float = 0.0*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

_attention_penalty(*self*, *attention: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

class langml.layers.**SelfAdditiveAttention**(*attention_units: Optional[int] = None*, *return_attention: bool = False*, *is_residual: bool = False*, *attention_activation: langml.tensor_typing.Activation = 'relu'*, *attention_epsilon: float = 10000000000.0*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_attention_bias: bool = True*, *attention_penalty_weight: float = 0.0*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

_attention_penalty(*self*, *attention: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

class langml.layers.**ScaledDotProductAttention**(*return_attention: bool = False*, *history_only: bool = False*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

ScaledDotProductAttention

$Attention(Q, K, V) = softmax(frac{Q K^T}{sqrt{d_k}}) V$

https://arxiv.org/pdf/1706.03762.pdf

get_config(*self*) → dict

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]] = None*, ***kwargs*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

class langml.layers.**MultiHeadAttention**(*head_num: int*, *return_attention: bool = False*, *attention_activation: langml.tensor_typing.Activation = 'relu'*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_attention_bias: bool = True*, *history_only: bool = False*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

MultiHeadAttention https://arxiv.org/pdf/1706.03762.pdf

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

static _reshape_to_batches(*x*, *head_num*)

static _reshape_attention_from_batches(*x*, *head_num*)

static _reshape_from_batches(*x*, *head_num*)

static _reshape_mask(*mask*, *head_num*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

compute_output_shape(*self*, *input_shape: Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

class langml.layers.**GatedAttentionUnit**(*attention_units: int*, *attention_activation: langml.tensor_typing.Activation = 'relu'*, *attention_normalizer: langml.tensor_typing.Activation = relu2*, *attention_epsilon: float = 10000000000.0*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_attention_bias: bool = True*, *use_attention_scale: bool = True*, *use_relative_position: bool = True*, *use_offset: bool = True*, *use_scale: bool = True*, *is_residual: bool = True*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

Gated Attention Unit https://arxiv.org/abs/2202.10447

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

apply_rotary_position_embeddings(*self*, *sinusoidal: langml.tensor_typing.Tensors*, **tensors*)

apply RoPE modified from: https://github.com/bojone/bert4keras/blob/master/bert4keras/backend.py#L310

attn(*self*, *x: langml.tensor_typing.Tensors*, *v: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

static **get_custom_objects**() → dict

langml.layers.**custom_objects**

langml.plm

Submodules

langml.plm.albert

Module Contents

Functions

load_albert(config_path: str, checkpoint_path: str, seq_len: Optional[int] = None, pretraining: bool = False, with_mlm: bool = True, with_nsp: bool = True, lazy_restore: bool = False, weight_prefix: Optional[str] = None, dropout_rate: float = 0.0, **kwargs) → Union[Tuple[langml.tensor_typing.Models, Callable], Tuple[langml.tensor_typing.Models, Callable, Callable]]	Load pretrained ALBERT

langml.plm.albert.**load_albert**(*config_path: str*, *checkpoint_path: str*, *seq_len: Optional[int] = None*, *pretraining: bool = False*, *with_mlm: bool = True*, *with_nsp: bool = True*, *lazy_restore: bool = False*, *weight_prefix: Optional[str] = None*, *dropout_rate: float = 0.0*, ***kwargs*) → Union[Tuple[langml.tensor_typing.Models, Callable], Tuple[langml.tensor_typing.Models, Callable, Callable]]

Load pretrained ALBERT :param - config_path: str, path of albert config :param - checkpoint_path: str, path of albert checkpoint :param - seq_len: Optional[int], specify fixed input sequence length, default None :param - pretraining: bool, pretraining mode, default False :param - with_mlm: bool, whether to use mlm task in pretraining, default True :param - with_nsp: bool, whether to use nsp/sop task in pretraining, default True :param - lazy_restore: bool, whether to restore pretrained weights lazily, default False.

Set it as True for distributed training.

Parameters

weight_prefix (-) –

Optional[str], prefix name of weights, default None.

You can set a prefix name in unshared siamese networks.

- dropout_rate: float, dropout rate, default 0.

Returns

keras model - bert: bert instance - restore: conditionally, it will return when lazy_restore=True

Return type

- model

langml.plm.bert

Module Contents

Classes

BERT

Functions

load_bert(config_path: str, checkpoint_path: str, seq_len: Optional[int] = None, pretraining: bool = False, with_mlm: bool = True, with_nsp: bool = True, lazy_restore: bool = False, weight_prefix: Optional[str] = None, dropout_rate: float = 0.0, **kwargs) → Union[Tuple[langml.tensor_typing.Models, Callable], Tuple[langml.tensor_typing.Models, Callable, Callable]]	Load pretrained BERT/RoBERTa

class langml.plm.bert.**BERT**(*vocab_size: int*, *position_size: int = 512*, *seq_len: int = 512*, *embedding_dim: int = 768*, *hidden_dim: Optional[int] = None*, *transformer_blocks: int = 12*, *attention_heads: int = 12*, *intermediate_size: int = 3072*, *dropout_rate: float = 0.1*, *attention_activation: langml.tensor_typing.Activation = None*, *feed_forward_activation: langml.tensor_typing.Activation = 'gelu'*, *initializer_range: float = 0.02*, *pretraining: bool = False*, *trainable_prefixs: Optional[List] = None*, *share_weights: bool = False*, *weight_prefix: Optional[str] = None*)

get_weight_name(*self*, *name: str*) → str

build(*self*)

get_inputs(*self*) → List[langml.tensor_typing.Tensors]

get_embedding(*self*, *inputs: List[langml.tensor_typing.Tensors]*) → List[langml.tensor_typing.Tensors]

is_trainable(*self*, *layer: tensorflow.keras.layers.Layer*) → bool

__call__(*self*, *inputs: Optional[Union[Tuple, List]] = None*, *return_model: bool = True*, *with_mlm: bool = True*, *with_nsp: bool = True*, *custom_embedding_callback: Optional[Callable] = None*) → langml.tensor_typing.Models

langml.plm.bert.**load_bert**(*config_path: str*, *checkpoint_path: str*, *seq_len: Optional[int] = None*, *pretraining: bool = False*, *with_mlm: bool = True*, *with_nsp: bool = True*, *lazy_restore: bool = False*, *weight_prefix: Optional[str] = None*, *dropout_rate: float = 0.0*, ***kwargs*) → Union[Tuple[langml.tensor_typing.Models, Callable], Tuple[langml.tensor_typing.Models, Callable, Callable]]

Load pretrained BERT/RoBERTa :param - config_path: str, path of albert config :param - checkpoint_path: str, path of albert checkpoint :param - seq_len: Optional[int], specify fixed input sequence length, default None :param - pretraining: bool, pretraining mode, default False :param - with_mlm: bool, whether to use mlm task in pretraining, default True :param - with_nsp: bool, whether to use nsp task in pretraining, default True :param - lazy_restore: bool, whether to restore pretrained weights lazily, default False.

Set it as True for distributed training.

Parameters

- **weight_prefix** (-) – Optional[str], prefix name of weights, default None. You can set a prefix name in unshared siamese networks.
- **dropout_rate** (-) – float, dropout rate, default 0.

Returns

keras model - bert: bert instance - restore: conditionally, it will return when lazy_restore=True

Return type

- model

langml.plm.layers

Module Contents

Classes

TokenEmbedding	
EmbeddingMatching	
Masked	Generate output mask based on the given mask.

class langml.plm.layers.**TokenEmbedding**

Bases: tensorflow.keras.layers.Embedding

static get_custom_objects() → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → List[Union[langml.tensor_typing.Tensors, None]]

call(*self*, *inputs: langml.tensor_typing.Tensors*) → List[langml.tensor_typing.Tensors]

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → List[langml.tensor_typing.Tensors]

class langml.plm.layers.**EmbeddingMatching**(*initializer: langml.tensor_typing.Initializer = 'zeros'*, *regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_bias: bool = True*, *use_softmax: bool = True*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

class langml.plm.layers.**Masked**(*return_masked: bool = False*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

Generate output mask based on the given mask. https://arxiv.org/pdf/1810.04805.pdf

static get_custom_objects() → dict

get_config(*self*) → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

Package Contents

Classes

TokenEmbedding	
EmbeddingMatching	
Masked	Generate output mask based on the given mask.

Functions

load_bert(config_path: str, checkpoint_path: str, seq_len: Optional[int] = None, pretraining: bool = False, with_mlm: bool = True, with_nsp: bool = True, lazy_restore: bool = False, weight_prefix: Optional[str] = None, dropout_rate: float = 0.0, **kwargs) → Union[Tuple[langml.tensor_typing.Models, Callable], Tuple[langml.tensor_typing.Models, Callable, Callable]]	Load pretrained BERT/RoBERTa
load_albert(config_path: str, checkpoint_path: str, seq_len: Optional[int] = None, pretraining: bool = False, with_mlm: bool = True, with_nsp: bool = True, lazy_restore: bool = False, weight_prefix: Optional[str] = None, dropout_rate: float = 0.0, **kwargs) → Union[Tuple[langml.tensor_typing.Models, Callable], Tuple[langml.tensor_typing.Models, Callable, Callable]]	Load pretrained ALBERT

Attributes

custom_objects

class langml.plm.**TokenEmbedding**

Bases: tensorflow.keras.layers.Embedding

static get_custom_objects() → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → List[Union[langml.tensor_typing.Tensors, None]]

call(*self*, *inputs: langml.tensor_typing.Tensors*) → List[langml.tensor_typing.Tensors]

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → List[langml.tensor_typing.Tensors]

class langml.plm.**EmbeddingMatching**(*initializer: langml.tensor_typing.Initializer = 'zeros'*, *regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_bias: bool = True*, *use_softmax: bool = True*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → langml.tensor_typing.Tensors

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

class langml.plm.**Masked**(*return_masked: bool = False*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

Generate output mask based on the given mask. https://arxiv.org/pdf/1810.04805.pdf

static get_custom_objects() → dict

get_config(*self*) → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*) → langml.tensor_typing.Tensors

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

langml.plm.**load_bert**(*config_path: str*, *checkpoint_path: str*, *seq_len: Optional[int] = None*, *pretraining: bool = False*, *with_mlm: bool = True*, *with_nsp: bool = True*, *lazy_restore: bool = False*, *weight_prefix: Optional[str] = None*, *dropout_rate: float = 0.0*, ***kwargs*) → Union[Tuple[langml.tensor_typing.Models, Callable], Tuple[langml.tensor_typing.Models, Callable, Callable]]

Load pretrained BERT/RoBERTa :param - config_path: str, path of albert config :param - checkpoint_path: str, path of albert checkpoint :param - seq_len: Optional[int], specify fixed input sequence length, default None :param - pretraining: bool, pretraining mode, default False :param - with_mlm: bool, whether to use mlm task in pretraining, default True :param - with_nsp: bool, whether to use nsp task in pretraining, default True :param - lazy_restore: bool, whether to restore pretrained weights lazily, default False.

Set it as True for distributed training.

Parameters

- **weight_prefix** (-) – Optional[str], prefix name of weights, default None. You can set a prefix name in unshared siamese networks.
- **dropout_rate** (-) – float, dropout rate, default 0.

Returns

keras model - bert: bert instance - restore: conditionally, it will return when lazy_restore=True

Return type

- model

langml.plm.**load_albert**(*config_path: str*, *checkpoint_path: str*, *seq_len: Optional[int] = None*, *pretraining: bool = False*, *with_mlm: bool = True*, *with_nsp: bool = True*, *lazy_restore: bool = False*, *weight_prefix: Optional[str] = None*, *dropout_rate: float = 0.0*, ***kwargs*) → Union[Tuple[langml.tensor_typing.Models, Callable], Tuple[langml.tensor_typing.Models, Callable, Callable]]

Load pretrained ALBERT :param - config_path: str, path of albert config :param - checkpoint_path: str, path of albert checkpoint :param - seq_len: Optional[int], specify fixed input sequence length, default None :param - pretraining: bool, pretraining mode, default False :param - with_mlm: bool, whether to use mlm task in pretraining, default True :param - with_nsp: bool, whether to use nsp/sop task in pretraining, default True :param - lazy_restore: bool, whether to restore pretrained weights lazily, default False.

Set it as True for distributed training.

Parameters

weight_prefix (-) –

Optional[str], prefix name of weights, default None.

You can set a prefix name in unshared siamese networks.

- dropout_rate: float, dropout rate, default 0.

Returns

keras model - bert: bert instance - restore: conditionally, it will return when lazy_restore=True

Return type

- model

langml.plm.**custom_objects**

langml.prompt

Subpackages

langml.prompt.clf

Submodules

langml.prompt.clf.ptuning

Module Contents

Classes

DataGenerator
PTuningForClassification

class langml.prompt.clf.ptuning.**DataGenerator**(*data: List[str]*, *labels: List[str]*, *tokenizer:* langml.tokenizer.Tokenizer, *template:* langml.prompt.base.Template, *batch_size: int = 32*)

Bases: *langml.prompt.base.BaseDataGenerator*

__len__(*self*)

make_iter(*self*, *random: bool = False*)

class langml.prompt.clf.ptuning.**PTuningForClassification**(*prompt_model:* BasePromptModel, *tokenizer:* langml.tokenizer.Tokenizer)

Bases: *langml.prompt.base.BasePromptTask*

fit(*self*, *data: List[str]*, *labels: List[str]*, *valid_data: Optional[List[str]] = None*, *valid_labels: Optional[List[str]] = None*, *model_path: Optional[str] = None*, *epoch: int = 20*, *batch_size: int = 16*, *early_stop: int = 10*, *do_shuffle: bool = True*, *f1_average: str = 'macro'*, *verbose: int = 1*)

Fitting ptuning model for classification :param - data: List[str], texts of traning data :param - labels: List[Union[str, List[str]]], traning labels :param - valid_data: List[str], texts of valid data :param - valid_labels: List[Union[str, List[str]]], labels of valid data :param - model_path: Optional[str], path to save model, default *None*, do not to save model :param - epoch: int, epochs to train :param - batch_size: int, batch size, :param - early_stop: int, patience of early stop :param - do_shuffle: whether to shuffle data in training phase :param - f1_average: str, {'micro', 'macro', 'samples','weighted', 'binary'} or None :param - verbose: int, 0 = silent, 1 = progress bar, 2 = one line per epoch

predict(*self*, *text: str*) → str

load(*self*, *model_path: str*)

load model :param - model_path: str, model path

langml.prompt.clf.utils

Module Contents

Classes

MetricsCallback

Functions

merge_template_tokens(template_ids: List[int], token_ids: List[int], max_length: Optional[int] = None) → Tuple[List[int], List[int]]	Merge template and token ids

langml.prompt.clf.utils.**merge_template_tokens**(*template_ids: List[int]*, *token_ids: List[int]*, *max_length: Optional[int] = None*) → Tuple[List[int], List[int]]

Merge template and token ids :param - template_ids: List[int], template ids :param - token_ids: List[int], token ids :param - max_length: int, max length

Returns

List[int], merged token ids - template_mask: List[int], template mask

Return type

- token_ids

class langml.prompt.clf.utils.**MetricsCallback**(*data: List[str]*, *labels: List[str]*, *mask_id: int*, *template:* langml.prompt.base.Template, *patience: int = 10*, *batch_size: int = 32*, *model_path: Optional[str] = None*, *f1_average: str = 'macro'*)

Bases: langml.keras.callbacks.Callback

on_train_begin(*self*, *logs=None*)

on_epoch_end(*self*, *epoch*, *logs=None*)

on_train_end(*self*, *logs=None*)

Package Contents

Classes

PTuningForClassification

class langml.prompt.clf.**PTuningForClassification**(*prompt_model:* BasePromptModel, *tokenizer:* langml.tokenizer.Tokenizer)

Bases: *langml.prompt.base.BasePromptTask*

fit(*self*, *data: List[str]*, *labels: List[str]*, *valid_data: Optional[List[str]] = None*, *valid_labels: Optional[List[str]] = None*, *model_path: Optional[str] = None*, *epoch: int = 20*, *batch_size: int = 16*, *early_stop: int = 10*, *do_shuffle: bool = True*, *f1_average: str = 'macro'*, *verbose: int = 1*)

Fitting ptuning model for classification :param - data: List[str], texts of traning data :param - labels: List[Union[str, List[str]]], traning labels :param - valid_data: List[str], texts of valid data :param - valid_labels: List[Union[str, List[str]]], labels of valid data :param - model_path: Optional[str], path to save model, default *None*, do not to save model :param - epoch: int, epochs to train :param - batch_size: int, batch size, :param - early_stop: int, patience of early stop :param - do_shuffle: whether to shuffle data in training phase :param - f1_average: str, {'micro', 'macro', 'samples','weighted', 'binary'} or None :param - verbose: int, 0 = silent, 1 = progress bar, 2 = one line per epoch

predict(*self*, *text: str*) → str

load(*self*, *model_path: str*)

load model :param - model_path: str, model path

langml.prompt.models

Submodules

langml.prompt.models.ptuning

Implementation P-Tuning

Paper: GPT Understands, Too URL: https://arxiv.org/pdf/2103.10385.pdf

Module Contents

Classes

PartialEmbedding

PTuniningPrompt

class langml.prompt.models.ptuning.**PartialEmbedding**(*input_dim: int*, *output_dim: int*, *active_start: int*, *active_end: int*, *embeddings_initializer: Optional[langml.tensor_typing.Initializer] = 'uniform'*, *embeddings_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *activity_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *embeddings_constraint: Optional[langml.tensor_typing.Constraint] = None*, *mask_zero: bool = False*, *input_length: Optional[int] = None*, ***kwargs*)

Bases: langml.L.Embedding

static get_custom_objects() → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → List[Union[langml.tensor_typing.Tensors, None]]

call(*self*, *inputs: langml.tensor_typing.Tensors*) → List[langml.tensor_typing.Tensors]

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → List[langml.tensor_typing.Tensors]

class langml.prompt.models.ptuning.**PTuningPrompt**(*plm_backbone: str*, *plm_config_path: str*, *plm_ckpt_path: str*, *template: langml.prompt.base.Template*, *learning_rate: float = 1e-05*, *freeze_plm: bool = True*, *encoder: str = 'mlp'*)

Bases: *langml.prompt.base.BasePromptModel*

build_model(*self*) → langml.tensor_typing.Models

Package Contents

Classes

PartialEmbedding

PTuningPrompt

Attributes

custom_objects

class langml.prompt.models.**PartialEmbedding**(*input_dim: int*, *output_dim: int*, *active_start: int*, *active_end: int*, *embeddings_initializer: Optional[langml.tensor_typing.Initializer] = 'uniform'*, *embeddings_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *activity_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *embeddings_constraint: Optional[langml.tensor_typing.Constraint] = None*, *mask_zero: bool = False*, *input_length: Optional[int] = None*, ***kwargs*)

Bases: langml.L.Embedding

static get_custom_objects() → dict

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → List[Union[langml.tensor_typing.Tensors, None]]

call(*self*, *inputs: langml.tensor_typing.Tensors*) → List[langml.tensor_typing.Tensors]

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → List[langml.tensor_typing.Tensors]

class langml.prompt.models.**PTuniningPrompt**(*plm_backbone: str*, *plm_config_path: str*, *plm_ckpt_path: str*, *template:* langml.prompt.base.Template, *learning_rate: float = 1e-05*, *freeze_plm: bool = True*, *encoder: str = 'mlp'*)

Bases: *langml.prompt.base.BasePromptModel*

build_model(*self*) → langml.tensor_typing.Models

langml.prompt.models.**custom_objects**

Submodules

langml.prompt.base

Module Contents

Classes

Template
BasePromptModel
BasePromptTask
BaseDataGenerator

class langml.prompt.base.**Template**(*template: List[str]*, *label_tokens_map: Dict[str, List[str]]*, *tokenizer:* langml.tokenizer.Tokenizer)

__len__(*self*) → int

encode_template(*self*, *template: str*) → List[int]

encode_label_tokens_map(*self*, *label_tokens_map: Dict[str, List[str]]*) → Dict[str, List[int]]

decode_label(*self*, *idx: int*, *default='<UNK>'*) → str

class langml.prompt.base.**BasePromptModel**(*plm_backbone: str*, *plm_config_path: str*, *plm_ckpt_path: str*, *template:* Template, *learning_rate: float = 1e-05*, *freeze_plm: bool = True*)

abstract build_model(*self*) → langml.tensor_typing.Models

class langml.prompt.base.**BasePromptTask**(*prompt_model:* BasePromptModel, *tokenizer:* langml.tokenizer.Tokenizer)

abstract fit(*self*)

abstract predict(*self*)

class langml.prompt.base.**BaseDataGenerator**

abstract make_iter(*self*, *random: bool = False*)

abstract __len__(*self*)

__call__(*self*, *random: bool = False*)

Package Contents

Classes

Template
PTunininingPrompt
PTuningForClassification

class langml.prompt.**Template**(*template: List[str]*, *label_tokens_map: Dict[str, List[str]]*, *tokenizer:* langml.tokenizer.Tokenizer)

__len__(*self*) → int

encode_template(*self*, *template: str*) → List[int]

encode_label_tokens_map(*self*, *label_tokens_map: Dict[str, List[str]]*) → Dict[str, List[int]]

decode_label(*self*, *idx: int*, *default='<UNK>'*) → str

class langml.prompt.**PTunininingPrompt**(*plm_backbone: str*, *plm_config_path: str*, *plm_ckpt_path: str*, *template:* langml.prompt.base.Template, *learning_rate: float = 1e-05*, *freeze_plm: bool = True*, *encoder: str = 'mlp'*)

Bases: *langml.prompt.base.BasePromptModel*

build_model(*self*) → langml.tensor_typing.Models

class langml.prompt.**PTuningForClassification**(*prompt_model:* BasePromptModel, *tokenizer:* langml.tokenizer.Tokenizer)

Bases: *langml.prompt.base.BasePromptTask*

fit(*self*, *data: List[str]*, *labels: List[str]*, *valid_data: Optional[List[str]] = None*, *valid_labels: Optional[List[str]] = None*, *model_path: Optional[str] = None*, *epoch: int = 20*, *batch_size: int = 16*, *early_stop: int = 10*, *do_shuffle: bool = True*, *f1_average: str = 'macro'*, *verbose: int = 1*)

Fitting ptuning model for classification :param - data: List[str], texts of traning data :param - labels: List[Union[str, List[str]]], traning labels :param - valid_data: List[str], texts of valid data :param - valid_labels: List[Union[str, List[str]]], labels of valid data :param - model_path: Optional[str], path to save model, default *None*, do not to save model :param - epoch: int, epochs to train :param - batch_size: int, batch size, :param - early_stop: int, patience of early stop :param - do_shuffle: whether to shuffle data in training phase :param - f1_average: str, {'micro', 'macro', 'samples','weighted', 'binary'} or None :param - verbose: int, 0 = silent, 1 = progress bar, 2 = one line per epoch

predict(*self*, *text: str*) → str

load(*self*, *model_path: str*)

load model :param - model_path: str, model path

langml.third_party

Submodules

langml.third_party.conlleval

Module Contents

Classes

EvalCounts

Functions

parse_args(argv)
parse_tag(t)
evaluate(iterable, options=None, delimiter=None)
uniq(iterable)
calculate_metrics(correct, guessed, total)
metrics(counts)
report(counts, out=None)
report_notprint(counts, out=None)
end_of_chunk(prev_tag, tag, prev_type, type_)
start_of_chunk(prev_tag, tag, prev_type, type_)
return_report(input_file)
main(argv)

Attributes

ANY_SPACE
Metrics

langml.third_party.conlleval.**ANY_SPACE = <SPACE>**

exception langml.third_party.conlleval.**FormatError**

Bases: Exception

Common base class for all non-exit exceptions.

langml.third_party.conlleval.**Metrics**

class langml.third_party.conlleval.**EvalCounts**

Bases: object

langml.third_party.conlleval.**parse_args**(*argv*)

langml.third_party.conlleval.**parse_tag**(*t*)

langml.third_party.conlleval.**evaluate**(*iterable*, *options=None*, *delimiter=None*)

langml.third_party.conlleval.**uniq**(*iterable*)

langml.third_party.conlleval.**calculate_metrics**(*correct*, *guessed*, *total*)

langml.third_party.conlleval.**metrics**(*counts*)

langml.third_party.conlleval.**report**(*counts*, *out=None*)

langml.third_party.conlleval.**report_notprint**(*counts*, *out=None*)

langml.third_party.conlleval.**end_of_chunk**(*prev_tag*, *tag*, *prev_type*, *type_*)

langml.third_party.conlleval.**start_of_chunk**(*prev_tag*, *tag*, *prev_type*, *type_*)

langml.third_party.conlleval.**return_report**(*input_file*)

langml.third_party.conlleval.**main**(*argv*)

langml.third_party.crf

Module Contents

Classes

AbstractRNNCell	Abstract object representing an RNN cell.
CrfDecodeForwardRnnCell	Computes the forward decoding in a linear-chain CRF.

Functions

viterbi_decode(score: langml.tensor_typing.Tensors, trans: langml.tensor_typing.Tensors) → Tuple[langml.tensor_typing.Tensors, langml.tensor_typing.Tensors]	**param score** A [seq_len, num_tags] matrix of unary potentials.
_generate_zero_filled_state_for_cell(cell, inputs, batch_size, dtype)	Generate a zero filled tensor with shape [batch_size, state_size].
crf_filtered_inputs(inputs: langml.tensor_typing.Tensors, tag_bitmap: langml.tensor_typing.Tensors) → tensorflow.Tensor	Constrains the inputs to filter out certain tags at each time step.
crf_sequence_score(inputs: langml.tensor_typing.Tensors, tag_indices: langml.tensor_typing.Tensors, sequence_lengths: langml.tensor_typing.Tensors, transition_params: langml.tensor_typing.Tensors) → tensorflow.Tensor	Computes the unnormalized score for a tag sequence.
crf_multitag_sequence_score(inputs: langml.tensor_typing.Tensors, tag_bitmap: langml.tensor_typing.Tensors, sequence_lengths: langml.tensor_typing.Tensors, transition_params: langml.tensor_typing.Tensors) → tensorflow.Tensor	Computes the unnormalized score of all tag sequences matching
crf_log_norm(inputs: langml.tensor_typing.Tensors, sequence_lengths: langml.tensor_typing.Tensors, transition_params: langml.tensor_typing.Tensors) → tensorflow.Tensor	Computes the normalization for a CRF.
crf_log_likelihood(inputs: langml.tensor_typing.Tensors, tag_indices: langml.tensor_typing.Tensors, sequence_lengths: langml.tensor_typing.Tensors, transition_params: Optional[langml.tensor_typing.Tensors] = None) → tensorflow.Tensor	Computes the log-likelihood of tag sequences in a CRF.
crf_unary_score(tag_indices: langml.tensor_typing.Tensors, sequence_lengths: langml.tensor_typing.Tensors, inputs: langml.tensor_typing.Tensors) → tensorflow.Tensor	Computes the unary scores of tag sequences.
crf_binary_score(tag_indices: langml.tensor_typing.Tensors, sequence_lengths: langml.tensor_typing.Tensors, transition_params: langml.tensor_typing.Tensors) → tensorflow.Tensor	Computes the binary scores of tag sequences.
crf_forward(inputs: langml.tensor_typing.Tensors, state: langml.tensor_typing.Tensors, transition_params: langml.tensor_typing.Tensors, sequence_lengths: langml.tensor_typing.Tensors) → tensorflow.Tensor	Computes the alpha values in a linear-chain CRF.
crf_decode_forward(inputs: langml.tensor_typing.Tensors, state: langml.tensor_typing.Tensors, transition_params: langml.tensor_typing.Tensors, sequence_lengths: langml.tensor_typing.Tensors) → tensorflow.Tensor	Computes forward decoding in a linear-chain CRF.
crf_decode_backward(inputs: langml.tensor_typing.Tensors, state: langml.tensor_typing.Tensors) → tensorflow.Tensor	Computes backward decoding in a linear-chain CRF.
crf_decode(potentials: langml.tensor_typing.Tensors, transition_params: langml.tensor_typing.Tensors, sequence_length: langml.tensor_typing.Tensors) → tensorflow.Tensor	Decode the highest scoring sequence of tags.
crf_constrained_decode(potentials: langml.tensor_typing.Tensors, tag_bitmap:	Decode the highest scoring sequence of tags under constraints.

langml.third_party.crf.**viterbi_decode**(*score: langml.tensor_typing.Tensors*, *trans: langml.tensor_typing.Tensors*) → Tuple[langml.tensor_typing.Tensors, langml.tensor_typing.Tensors]

Parameters

- **score** – A [seq_len, num_tags] matrix of unary potentials.
- **trans** – A [num_tags, num_tags] matrix of binary potentials.

Returns

A [seq_len] list of integers containing the highest scoring tag indices.

viterbi_score: A float containing the score for the Viterbi sequence.

Return type
viterbi

langml.third_party.crf.**_generate_zero_filled_state_for_cell**(*cell*, *inputs*, *batch_size*, *dtype*)

Generate a zero filled tensor with shape [batch_size, state_size].

langml.third_party.crf.**crf_filtered_inputs**(*inputs: langml.tensor_typing.Tensors*, *tag_bitmap: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Constrains the inputs to filter out certain tags at each time step. tag_bitmap limits the allowed tags at each input time step. This is useful when an observed output at a given time step needs to be constrained to a selected set of tags. Args: inputs: A [batch_size, max_seq_len, num_tags] tensor of unary potentials

to use as input to the CRF layer.

tag_bitmap: A [batch_size, max_seq_len, num_tags] boolean tensor representing all active tags at each index for which to calculate the unnormalized score.

Returns: filtered_inputs: A [batch_size] vector of unnormalized sequence scores.

langml.third_party.crf.**crf_sequence_score**(*inputs: langml.tensor_typing.Tensors*, *tag_indices: langml.tensor_typing.Tensors*, *sequence_lengths: langml.tensor_typing.Tensors*, *transition_params: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Computes the unnormalized score for a tag sequence. :param inputs: A [batch_size, max_seq_len, num_tags] tensor of unary potentials

to use as input to the CRF layer.

Parameters

- **tag_indices** – A [batch_size, max_seq_len] matrix of tag indices for which we compute the unnormalized score.
- **sequence_lengths** – A [batch_size] vector of true sequence lengths.
- **transition_params** – A [num_tags, num_tags] transition matrix.

Returns
A [batch_size] vector of unnormalized sequence scores.

Return type
sequence_scores

langml.third_party.crf.**crf_multitag_sequence_score**(*inputs: langml.tensor_typing.Tensors*, *tag_bitmap: langml.tensor_typing.Tensors*, *sequence_lengths: langml.tensor_typing.Tensors*, *transition_params: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Computes the unnormalized score of all tag sequences matching tag_bitmap. tag_bitmap enables more than one tag to be considered correct at each time step. This is useful when an observed output at a given time step is consistent with more than one tag, and thus the log likelihood of that observation must take into account all possible consistent tags. Using one-hot vectors in tag_bitmap gives results identical to crf_sequence_score. :param inputs: A [batch_size, max_seq_len, num_tags] tensor of unary potentials

to use as input to the CRF layer.

Parameters

- **tag_bitmap** – A [batch_size, max_seq_len, num_tags] boolean tensor representing all active tags at each index for which to calculate the unnormalized score.
- **sequence_lengths** – A [batch_size] vector of true sequence lengths.
- **transition_params** – A [num_tags, num_tags] transition matrix.

Returns
A [batch_size] vector of unnormalized sequence scores.

Return type
sequence_scores

langml.third_party.crf.**crf_log_norm**(*inputs: langml.tensor_typing.Tensors*, *sequence_lengths: langml.tensor_typing.Tensors*, *transition_params: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Computes the normalization for a CRF. :param inputs: A [batch_size, max_seq_len, num_tags] tensor of unary potentials

to use as input to the CRF layer.

Parameters

- **sequence_lengths** – A [batch_size] vector of true sequence lengths.
- **transition_params** – A [num_tags, num_tags] transition matrix.

Returns
A [batch_size] vector of normalizers for a CRF.

Return type
log_norm

langml.third_party.crf.**crf_log_likelihood**(*inputs: langml.tensor_typing.Tensors*, *tag_indices: langml.tensor_typing.Tensors*, *sequence_lengths: langml.tensor_typing.Tensors*, *transition_params: Optional[langml.tensor_typing.Tensors] = None*) → tensorflow.Tensor

Computes the log-likelihood of tag sequences in a CRF. :param inputs: A [batch_size, max_seq_len, num_tags] tensor of unary potentials

to use as input to the CRF layer.

Parameters

- **tag_indices** – A [batch_size, max_seq_len] matrix of tag indices for which we compute the log-likelihood.
- **sequence_lengths** – A [batch_size] vector of true sequence lengths.
- **transition_params** – A [num_tags, num_tags] transition matrix, if available.

Returns

A [batch_size] *Tensor* containing the log-likelihood of
each example, given the sequence of tag indices.

transition_params: A [num_tags, num_tags] transition matrix. This is
either provided by the caller or created in this function.

Return type
log_likelihood

langml.third_party.crf.**crf_unary_score**(*tag_indices: langml.tensor_typing.Tensors*, *sequence_lengths: langml.tensor_typing.Tensors*, *inputs: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Computes the unary scores of tag sequences. :param tag_indices: A [batch_size, max_seq_len] matrix of tag indices. :param sequence_lengths: A [batch_size] vector of true sequence lengths. :param inputs: A [batch_size, max_seq_len, num_tags] tensor of unary potentials.

Returns
A [batch_size] vector of unary scores.

Return type
unary_scores

langml.third_party.crf.**crf_binary_score**(*tag_indices: langml.tensor_typing.Tensors*, *sequence_lengths: langml.tensor_typing.Tensors*, *transition_params: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Computes the binary scores of tag sequences. :param tag_indices: A [batch_size, max_seq_len] matrix of tag indices. :param sequence_lengths: A [batch_size] vector of true sequence lengths. :param transition_params: A [num_tags, num_tags] matrix of binary potentials.

Returns
A [batch_size] vector of binary scores.

Return type
binary_scores

langml.third_party.crf.**crf_forward**(*inputs: langml.tensor_typing.Tensors*, *state: langml.tensor_typing.Tensors*, *transition_params: langml.tensor_typing.Tensors*, *sequence_lengths: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Computes the alpha values in a linear-chain CRF. See http://www.cs.columbia.edu/~mcollins/fb.pdf for reference. :param inputs: A [batch_size, num_tags] matrix of unary potentials. :param state: A [batch_size, num_tags] matrix containing the previous alpha

values.

Parameters

- **transition_params** – A [num_tags, num_tags] matrix of binary potentials. This matrix is expanded into a [1, num_tags, num_tags] in preparation for the broadcast summation occurring within the cell.

- **sequence_lengths** – A [batch_size] vector of true sequence lengths.

Returns

A [batch_size, num_tags] matrix containing the
new alpha values.

Return type
new_alphas

class langml.third_party.crf.**AbstractRNNCell**

Bases: tensorflow.keras.layers.Layer

Abstract object representing an RNN cell. This is the base class for implementing RNN cells with custom behavior. Every *RNNCell* must have the properties below and implement *call* with the signature *(output, next_state) = call(input, state)*. Examples: ```python

class MinimalRNNCell(AbstractRNNCell): def __init__(self, units, **kwargs):

self.units = units super(MinimalRNNCell, self).__init__(**kwargs)

@property def state_size(self):

return self.units

def build(self, input_shape):

self.kernel = self.add_weight(shape=(input_shape[-1], self.units),
initializer='uniform', name='kernel')

self.recurrent_kernel = self.add_weight(
shape=(self.units, self.units), initializer='uniform', name='recurrent_kernel')

self.built = True

def call(self, inputs, states):
prev_output = states[0] h = K.dot(inputs, self.kernel) output = h + K.dot(prev_output, self.recurrent_kernel) return output, output

``` This definition of cell differs from the definition used in the literature. In the literature, 'cell' refers to an object with a single scalar output. This definition refers to a horizontal array of such units. An RNN cell, in the most abstract setting, is anything that has a state and performs some operation that takes a matrix of inputs. This operation results in an output matrix with *self.output_size* columns. If *self.state_size* is an integer, this operation also results in a new state matrix with *self.state_size* columns. If *self.state_size* is a (possibly nested tuple of) TensorShape object(s), then it should return a matching structure of Tensors having shape *[batch_size].concatenate(s)* for each *s* in *self.batch_size*.

**abstract call**(*self*, *inputs*, *states*)

The function that contains the logic for one RNN step calculation. Args: inputs: the input tensor, which is a slide from the overall RNN input by

the time dimension (usually the second dimension).

**states: the state tensor from previous step, which has the same shape**
as *(batch, state_size)*. In the case of timestep 0, it will be the initial state user specified, or zero filled tensor otherwise.

Returns: A tuple of two tensors:

1. output tensor for the current timestep, with size *output_size*.
```

2. state tensor for next step, which has the shape of *state_size*.

property state_size(*self*)

size(s) of state(s) used by this cell. It can be represented by an Integer, a TensorShape or a tuple of Integers or TensorShapes.

property output_size(*self*)

Integer or TensorShape: size of outputs produced by this cell.

get_initial_state(*self*, *inputs=None*, *batch_size=None*, *dtype=None*)

class langml.third_party.crf.**CrfDecodeForwardRnnCell**(*transition_params: langml.tensor_typing.Tensors*, ***kwargs*)

Bases: *AbstractRNNCell*

Computes the forward decoding in a linear-chain CRF.

property state_size(*self*)

size(s) of state(s) used by this cell. It can be represented by an Integer, a TensorShape or a tuple of Integers or TensorShapes.

property output_size(*self*)

Integer or TensorShape: size of outputs produced by this cell.

build(*self*, *input_shape*)

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

call(*self*, *inputs: langml.tensor_typing.Tensors*, *state: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, ***kwargs*)

Build the CrfDecodeForwardRnnCell. :param inputs: A [batch_size, num_tags] matrix of unary potentials. :param state: A [batch_size, num_tags] matrix containing the previous step's

score values.

Returns
A [batch_size, num_tags] matrix of backpointers. new_state: A [batch_size, num_tags] matrix of new score values.

Return type
backpointers

get_config(*self*) → dict

classmethod from_config(*cls*, *config: dict*) → *CrfDecodeForwardRnnCell*

langml.third_party.crf.**crf_decode_forward**(*inputs: langml.tensor_typing.Tensors*, *state: langml.tensor_typing.Tensors*, *transition_params: langml.tensor_typing.Tensors*, *sequence_lengths: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Computes forward decoding in a linear-chain CRF. :param inputs: A [batch_size, num_tags] matrix of unary potentials. :param state: A [batch_size, num_tags] matrix containing the previous step's

score values.

Parameters

- **transition_params** – A [num_tags, num_tags] matrix of binary potentials.
- **sequence_lengths** – A [batch_size] vector of true sequence lengths.

Returns
A [batch_size, num_tags] matrix of backpointers. new_state: A [batch_size, num_tags] matrix of new score values.

Return type
backpointers

langml.third_party.crf.**crf_decode_backward**(*inputs: langml.tensor_typing.Tensors*, *state: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Computes backward decoding in a linear-chain CRF. :param inputs: A [batch_size, num_tags] matrix of

backpointer of next step (in time order).

Parameters
state – A [batch_size, 1] matrix of tag index of next step.

Returns

A [batch_size, num_tags]
tensor containing the new tag indices.

Return type
new_tags

langml.third_party.crf.**crf_decode**(*potentials: langml.tensor_typing.Tensors*, *transition_params: langml.tensor_typing.Tensors*, *sequence_length: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Decode the highest scoring sequence of tags. :param potentials: A [batch_size, max_seq_len, num_tags] tensor of

unary potentials.

Parameters

- **transition_params** – A [num_tags, num_tags] matrix of binary potentials.
- **sequence_length** – A [batch_size] vector of true sequence lengths.

Returns

A [batch_size, max_seq_len] matrix, with dtype *tf.int32*.
Contains the highest scoring tag indices.

best_score: A [batch_size] vector, containing the score of *decode_tags*.

Return type
decode_tags

langml.third_party.crf.**crf_constrained_decode**(*potentials: langml.tensor_typing.Tensors*, *tag_bitmap: langml.tensor_typing.Tensors*, *transition_params: langml.tensor_typing.Tensors*, *sequence_length: langml.tensor_typing.Tensors*) → tensorflow.Tensor

Decode the highest scoring sequence of tags under constraints. This is a function for tensor. :param potentials: A [batch_size, max_seq_len, num_tags] tensor of

unary potentials.

Parameters

- **tag_bitmap** – A [batch_size, max_seq_len, num_tags] boolean tensor representing all active tags at each index for which to calculate the unnormalized score.
- **transition_params** – A [num_tags, num_tags] matrix of binary potentials.
- **sequence_length** – A [batch_size] vector of true sequence lengths.

Returns

A [batch_size, max_seq_len] matrix, with dtype *tf.int32*.
Contains the highest scoring tag indices.

best_score: A [batch_size] vector, containing the score of *decode_tags*.

Return type
decode_tags

langml.transformer

Submodules

langml.transformer.encoder

Yet another transformer implementation.

Module Contents

Classes

TransformerEncoder
TransformerEncoderBlock

class langml.transformer.encoder.**TransformerEncoder**(*attention_heads: int*, *hidden_dim: int*, *attention_activation: langml.tensor_typing.Activation = None*, *feed_forward_activation: langml.tensor_typing.Activation = gelu*, *dropout_rate: float = 0.0*, *trainable: bool = True*, *name: str = 'Transformer-Encoder'*)

__call__(*self*, *inputs: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

class langml.transformer.encoder.**TransformerEncoderBlock**(*blocks: int*, *attention_heads: int*, *hidden_dim: int*, *attention_activation: langml.tensor_typing.Activation = None*, *feed_forward_activation: langml.tensor_typing.Activation = gelu*, *dropout_rate: float = 0.0*, *trainable: bool = False*, *name: str = 'TransformerEncoderBlock'*, *share_weights: bool = False*)

__call__(*self*, *inputs: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

langml.transformer.layers

Yet another transformer implementation.

Module Contents

Classes

FeedForward	Feed Forward Layer

class langml.transformer.layers.**FeedForward**(*units*, *activation: langml.tensor_typing.Activation = 'relu'*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_bias: bool = True*, *dropout_rate: float = 0.0*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

Feed Forward Layer https://arxiv.org/pdf/1706.03762.pdf

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, *training: Optional[Any] = None*, ***kwargs*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

Package Contents

Classes

FeedForward	Feed Forward Layer

Attributes

TF_KERAS
custom_objects

langml.transformer.**TF_KERAS**

class langml.transformer.**FeedForward**(*units*, *activation: langml.tensor_typing.Activation = 'relu'*, *kernel_initializer: langml.tensor_typing.Initializer = 'glorot_normal'*, *kernel_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *kernel_constraint: Optional[langml.tensor_typing.Constraint] = None*, *bias_initializer: langml.tensor_typing.Initializer = 'zeros'*, *bias_regularizer: Optional[langml.tensor_typing.Regularizer] = None*, *bias_constraint: Optional[langml.tensor_typing.Constraint] = None*, *use_bias: bool = True*, *dropout_rate: float = 0.0*, ***kwargs*)

Bases: tensorflow.keras.layers.Layer

Feed Forward Layer https://arxiv.org/pdf/1706.03762.pdf

get_config(*self*) → dict

build(*self*, *input_shape: langml.tensor_typing.Tensors*)

call(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[langml.tensor_typing.Tensors] = None*, *training: Optional[Any] = None*, ***kwargs*) → Union[List[langml.tensor_typing.Tensors], langml.tensor_typing.Tensors]

compute_mask(*self*, *inputs: langml.tensor_typing.Tensors*, *mask: Optional[Union[langml.tensor_typing.Tensors, List[langml.tensor_typing.Tensors]]] = None*) → Union[List[Union[langml.tensor_typing.Tensors, None]], langml.tensor_typing.Tensors]

static get_custom_objects() → dict

compute_output_shape(*self*, *input_shape: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

langml.transformer.**custom_objects**

6.1.2 Submodules

langml.activations

Activations

Module Contents

Functions

gelu(x: langml.tensor_typing.Tensors) → langml.tensor_typing.Tensors		Gaussian Error Linear Units (GELUs)
relu2(x: langml.tensor_typing.Tensors) → langml.tensor_typing.Tensors		

Attributes

custom_objects

langml.activations.**gelu**(*x: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

Gaussian Error Linear Units (GELUs) https://arxiv.org/abs/1606.08415

$GELU(x) = 0.5x(1 + tanh[sqrt(2 / Pi) (x + 0.044715x^3)])$

langml.activations.**relu2**(*x: langml.tensor_typing.Tensors*) → langml.tensor_typing.Tensors

langml.activations.**custom_objects**

langml.cli

Module Contents

Functions

cli()	LangML client
main()	

langml.cli.**cli**()

LangML client

langml.cli.**main**()

langml.log

Module Contents

Functions

print_log(level: int, msg: str, *args)

Attributes

debug
info
warn
error

langml.log.**print_log**(*level: int*, *msg: str*, **args*)

langml.log.**debug**

langml.log.**info**

langml.log.**warn**

langml.log.**error**

langml.model

Module Contents

Functions

get_random_string(length)
export_model_v1(model, export_model_dir) **param export_model_dir** type string, save dir for exported model url
save_frozen(model: langml.tensor_typing.Models, fpath: str)
load_frozen(model_dir: str, session: Any = None) → Any

Attributes

SAVED_MODEL_TAG

langml.model.**SAVED_MODEL_TAG = serve**

langml.model.**get_random_string**(*length*)

langml.model.**export_model_v1**(*model*, *export_model_dir*)

Parameters

- **export_model_dir** – type string, save dir for exported model url
- **model_version** – type int best

:return:no return

langml.model.**save_frozen**(*model: langml.tensor_typing.Models*, *fpath: str*)

langml.model.**load_frozen**(*model_dir: str*, *session: Any = None*) → Any

langml.tensor_typing

Module Contents

langml.tensor_typing.**Number**

langml.tensor_typing.**Initializer**

langml.tensor_typing.**Regularizer**

langml.tensor_typing.**Constraint**

langml.tensor_typing.**Activation**

langml.tensor_typing.**Optimizer**

langml.tensor_typing.**Tensors**

langml.tensor_typing.**Models**

langml.tokenizer

LangML Tokenizer

- WPTokenizer: WordPiece Tokenizer
- SPTokenizer: SentencePiece Tokenizer

Wrap for:

- tokenizers.BertWordPieceTokenizer
- sentencepiece.SentencePieceProcessor

We don't provide all functions of raw tokenizer, please use raw tokenizer for full usage.

Module Contents

Classes

Encoding	Product of tokenizer encoding
SpecialTokens	

Tokenizer	Base Tokenizer
SPTokenizer	SentencePiece Tokenizer
WPTokenizer	WordPieceTokenizer

class langml.tokenizer.**Encoding**(*ids: Union[numpy.ndarray, List[int]]*, *segment_ids: Union[numpy.ndarray, List[int]]*, *tokens: List[str]*)

Product of tokenizer encoding

ids

segment_ids

tokens

class langml.tokenizer.**SpecialTokens**

PAD = [PAD]

UNK = [UNK]

MASK = [MASK]

CLS = [CLS]

SEP = [SEP]

__contains__(*self*, *token: str*) → bool

Check if the input token exists in special tokens. :param - token: str

Returns
bool

tokens(*self*) → List[str]

class langml.tokenizer.**Tokenizer**(*vocab_path: str*, *lowercase: bool = False*)

Base Tokenizer

enable_truncation(*self*, *max_length: int*, *strategy: str = 'post'*)

Parameters

- **max_length** (-) – int,
- **strategy** (-) – str, optional, truncation strategy, options: *post* or *pre*, default *post*

tokens_mapping(*self*, *sequence: str*, *tokens: List[str]*) → List[Tuple[int, int]]

Get tokens to their corresponding sequence position mapping. Tokens may contain special marks, e.g., *##*, , and *[UNK]*. Use this function can obtain the corresponding raw token in the sequence.

Parameters

- **sequence** (-) – str, the input sequence

- **tokens** (-) – List[str], tokens of the input sequence

Returns
List[Tuple[int, int]]

Examples: >>> sequence = 'I like watermelons' >>> tokens = ['[CLS]', 'i', 'like', 'water', 'mel', 'ons', '[SEP]'] >>> mapping = tokenizer.tokens_mapping(tokens) >>> start_index, end_index = 3, 5 >>> print("current token", tokens[start_index: end_index + 1]) ['water', 'mel', 'ons'] >>> print("raw token", sequence[mapping[start_index][0]: mapping[end_index][1]]) watermelons

Reference:
https://github.com/bojone/bert4keras

encode(*self*, *sequence: str*, *pair: Optional[str] = None*, *return_array: bool = False*) → *Encoding*

Parameters

- **sequence** (-) – str, input sequence
- **pair** (-) – str, optional, pair sequence, default *None*
- **return_array** (-) – bool, optional, whether to return numpy array, default *True*

Returns
Encoding object

encode_batch(*self*, *inputs: Union[List[str], List[Tuple[str, str]], List[List[str]]]*, *padding: bool = True*, *padding_strategy: str = 'post'*, *return_array: bool = False*) → *Encoding*

Parameters

- **inputs** (-) – Union[List[str], List[Tuple[str, str]], List[List[str]]], list of texts or list of text pairs.
- **padding** (-) – bool, optional, whether to padding sequences, default *True*
- **padding_strategy** (-) – str, optional, options: *post* or *pre*, default *post*
- **return_array** (-) – bool, optional, whether to return numpy array, default *True*

Returns
Encoding object

stem(*self*, *token*)

sequence_lower(*self*, *sequence: str*) → str

Do lower to sequence, except for special tokens. :param - sequence: str

Returns
str

sequence_truncating(*self*, *max_token_length: int*, *tokens: List[str]*, *pair_tokens: Optional[List[str]] = None*) → Tuple[List[str], Optional[List[str]]]

Truncating sequence :param - max_token_length: int, maximum token length :param - tokens: List[str], input tokens :param - pair_tokens: Optional[List[str]], optional, input pair tokens, default None

Returns
Tuple[List[str], Optional[List[str]]]

raw_tokenizer(*self*) → object

Return raw tokenizer, i.e. object of *tokenizers.BertWordPieceTokenizer* or *sentencepiece.SentencePieceProcessor*

abstract tokenize(*self*, *sequence: str*) → List[str]

abstract decode(*self*, *ids: List[int]*, *skip_special_tokens: bool = True*) → List[str]

abstract get_vocab_size(*self*) → int

abstract id_to_token(*self*, *idx: int*) → str

abstract token_to_id(*self*, *token: str*) → int

abstract get_vocab(*self*) → Dict

class langml.tokenizer.**SPTokenizer**(*vocab_path: str*, *lowercase: bool = False*)

Bases: *Tokenizer*

SentencePiece Tokenizer Wrap for *sentencepiece*.

get_vocab_size(*self*) → int

Return vocab size

token_to_id(*self*, *token: str*) → int

Convert the input token to corresponding index :param - token: str

Returns
int

id_to_token(*self*, *idx: int*) → str

Convert index to corresponding token :param - idx: int

Returns
str

tokenize(*self*, *sequence: str*) → List[str]

Tokenize sequence to token peices. :param - sequence: str

Returns
List[str]

decode(*self*, *ids: List[int]*, *skip_special_tokens: bool = True*) → List[str]

Decode indexs to tokens :param - ids: List[int] :param - skip_special_tokens: bool, optioanl, whether to skip special tokens, default *True*

Returns
List[str]

get_vocab(*self*) → Dict

Return vocabulary

class langml.tokenizer.**WPTokenizer**(*vocab_path: str*, *lowercase: bool = False*)

Bases: *Tokenizer*

WordPieceTokenizer Wrap for *BertWordPieceTokenizer*.

get_vocab_size(*self*) → int

Return vocab size

token_to_id(*self*, *token: str*) → int

Convert the input token to corresponding index :param - token: str

Returns
int

id_to_token(*self*, *idx: int*) → str

Convert index to corresponding token :param - idx: int

Returns
str

tokenize(*self*, *sequence: str*) → List[str]

Tokenize sequence to token peices. :param - sequence: str

Returns
List[str]

decode(*self*, *ids: List[int]*, *skip_special_tokens: bool = True*) → List[str]

Decode indexs to tokens :param - ids: List[int] :param - skip_special_tokens: bool, optioanl, whether to skip special tokens, default *True*

Returns
List[str]

get_vocab(*self*) → Dict

Return vocabulary

add_special_tokens(*self*, *tokens: List[str]*)

Specify special tokens, the tokenizer will reserve special tokens as a whole (i.e. don't split them) in tokenizing. Currently, only the WPTokenizer supports specifying special tokens. :param - tokens: List[str], special tokens

langml.utils

Module Contents

Functions

deprecated_warning(msg='this function is deprecated! it might be removed in a future version.')	
bio_decode(tags: List[str]) → List[Tuple[int, int, str]]	Decode BIO tags
load_variables(checkpoint_path: str) → Callable	load variables from chechkpoint
auto_tokenizer(vocab_path: str, lowercase: bool = False) → langml.tokenizer.Tokenizer	

langml.utils.**deprecated_warning**(*msg='this function is deprecated! it might be removed in a future version.'*)

langml.utils.**bio_decode**(*tags: List[str]*) → List[Tuple[int, int, str]]

Decode BIO tags

Examples: >>> bio_decode(['B-PER', 'I-PER', 'O', 'B-ORG', 'I-ORG', 'I-ORG']) >>> [(0, 1, 'PER'), (3, 5, 'ORG')]

langml.utils.**load_variables**(*checkpoint_path: str*) → Callable

load variables from chechkpoint

langml.utils.**auto_tokenizer**(*vocab_path: str*, *lowercase: bool = False*) → *langml.tokenizer.Tokenizer*

6.1.3 Package Contents

langml.**__version__ = 0.4.2**

langml.**TF_VERSION**

langml.**TF_KERAS**

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

I

L

M

N

O

P

R

S

T

U

V

W