AllenNLP

**All you need for a simple classifier: writing a DatasetReader and Model.** Then AllenNLP takes care of the rest: connecting your input files to the dataset reader, intelligently batching together your instances and feeding them to the model, and optimizing the model's parameters by using backprop on the loss.

# Data

## 字段Field

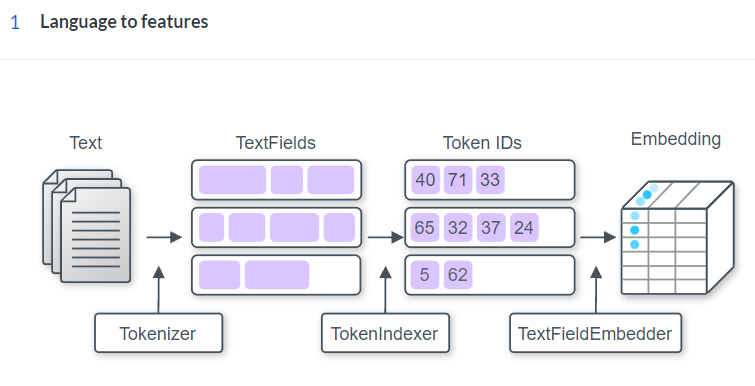
常用：

TextField

LabelField

SequenceLabelField

如何得到TextFiel并数值化



## 标记Tokenizer

Tokenizer (Text → Tokens)【标记器】

将字符串拆分为标记

常用：

WhitespaceTokenizer(): 把文本转换为单词的组合

CharacterTokenizer(): 把文本转换为字符的组合

SpacyTokenizer: 转换为单词同时获得额外属性

tokenize()方法返回a list of Tokens

### Token Class

tag\_是其一个属性

### SpacyTokenizer

@Tokenizer.register("spacy")

class SpacyTokenizer(Tokenizer):

**Registered as a Tokenizer with name "spacy", which is currently the default.**

**Parameters:**

pos\_tags : bool, optional (default = False)

If True, performs POS tagging with spacy model on the tokens. Generally used in conjunction with PosTagIndexer.

split\_on\_spaces : bool, optional (default = False)

If True, will split by spaces without performing tokenization. Used when your data is already tokenized, but you want to perform pos, ner or parsing on the tokens.

## 标记索引

TextField, TokenIndexer, and Vocabulary (Tokens → Ids)

### TextField

#### 调用方法

text\_field = TextField(tokens, token\_indexers)

TextField: 把 a list of Tokens 表示为an array

利用Vocabulary索引: text\_field.index(vocab)

然后提取padding信息: padding\_lengths = text\_field.get\_padding\_lengths()

\_num表示有填充

最后得到填充后的TextFieldTensors:

tensor\_dict = text\_field.as\_tensor(padding\_lengths)

另外: tensor\_dict = text\_field.batch\_tensors([token\_tensor])

batch\_tensors(List[TextFieldTensors])方法按相同属性打包TextFieldTensors中的Tensors → TextFieldTensors

token\_indexers: Dict[str, Token\_Indexer]

#### 操作输出(TextFieldTensors)

1. get\_text\_field\_mask()

Takes the dictionary of tensors produced by a TextField and returns a mask with 0 where the tokens are padding, and 1 otherwise. **(0: padding)**

→If num\_wrapping\_dims == 0, the returned mask has shape (batch\_size, num\_tokens)

In order to get a token mask, we use the tensor in the dictionary with the lowest number of dimensions.

**参数: num\_wrapping\_dims可以解决多个TexField的情形**

2. get\_token\_ids\_from\_text\_field\_tensors()

Get the token ids out of the data structure

### ListField

A ListField is a list of other fields. You would use this to represent, e.g., a list of answer options that are themselves TextFields.

Input: a list of TextFields that have shape (num\_words, num\_characters)

Output: a tensor of shape (num\_sentences, num\_words, num\_characters).

### TokenIndexer

决定如何表示each Token【标记索引器】

0: padding

1: Vocabulary未知的标记

常用索引

SingleIdTokenIndexer(): 把每个标记表示为一个ID（根据命名空间）

TokenCharactersIndexer(): 把每个标记表示成一串字符的ID

#### PretrainedTransformerMismatchedIndexer(model\_name=transformer\_model)

Use a tokenizer that splits strings into words, while the transformer expects wordpieces as input. This indexer splits the words into wordpieces and flattens them out.

#### SingleIdTokenIndexer

把每个标记表示为一个ID

@TokenIndexer.register("single\_id")

class SingleIdTokenIndexer(TokenIndexer):

Registered as a TokenIndexer with name "single\_id".

**Parameters**

namespace : Optional[str], optional (**default = "tokens"**)

We will use this namespace in the Vocabulary to map strings to indices.

feature\_name : str, optional (**default = "text"**)

We will use the Token attribute with this name as input. If you use a non-default value here, you almost certainly want to also change the namespace parameter, and you might want to give a default\_value.

default\_value : str, optional

When you want to use a non-default feature\_name, you sometimes want to have a default value to go with it.

## 嵌入(Embedder)

TextFieldEmbedder (Ids → Vectors)

分别编码token\_indexers的输出，然后组合所有输出，使得每个标记对应一个向量

TextFieldTensors:

filed\_name[token\_indexer\_name][tokens]←→Dict[str, Dict[str, torch.Tensor]]

外部字典中的每个条目对应一个标记索引器

内部字典为索引tensor

常用: embedder = BasicTextFieldEmbedder(token\_embedders)

token\_embedders = Dict[str, Embddding]

embedded\_tokens = embedder(token\_tensor)

### TokenEmbedder

Takes as input a tensor with integer ids that have been output from a TokenIndexer and outputs a vector per token in the input.

Input: (batch\_size, num\_tokens) or (batch\_size, num\_tokens, num\_characters)

Output: (batch\_size, num\_tokens, output\_dim)

重要方法: get\_output\_dim

Returns the final output dimension that this TokenEmbedder uses to represent each token. This is not the shape of the returned tensor, but the last element of that shape.

#### Embedding(嵌入)

**Registered as a TokenEmbedder with name "embedding".**

@TokenEmbedder.register("embedding")

class Embedding(TokenEmbedder):

Parameters

num\_embeddings: int

Size of the dictionary of embeddings (vocabulary size).

embedding\_dim: int

The size of each embedding vector. (get\_output\_dim()方法的返回值)

Return

An Embedding module.

#### TokenCharactersEncoder

Takes the output of a TokenCharactersIndexer, which is a tensor of shape **(batch\_size, num\_tokens, num\_characters)**, embeds the characters, runs a token-level encoder, and returns the result, which is a tensor of shape **(batch\_size, num\_tokens, encoding\_dim)**.

@TokenEmbedder.register("character\_encoding")

class TokenCharactersEncoder(TokenEmbedder):

**Registered as a TokenEmbedder with name "character\_encoding"(注册为名为“character\_encoding”的标记嵌入程序)**

Parameters:

embedding: Embedding

encoder: Seq2vecEncoder

#### PretrainedTransformerMismatchedEmbedder

@TokenEmbedder.register("pretrained\_transformer\_mismatched")

class PretrainedTransformerMismatchedEmbedder(TokenEmbedder):

Registered as a TokenEmbedder with name "pretrained\_transformer\_mismatched"

#### PretrainedTransformerEmbedder

@TokenEmbedder.register("pretrained\_transformer")

class PretrainedTransformerEmbedder(TokenEmbedder):

Registered as a TokenEmbedder with name "pretrained\_transformer"

### TextFieldEmbedder

A Module that takes as input the DataArray produced by a TextField and returns as output an embedded representation of the tokens in that field.

### BasicTextFieldEmbedder

@TextFieldEmbedder.register("basic")

class BasicTextFieldEmbedder(TextFieldEmbedder):

def \_\_init\_\_(self, token\_embedders: Dict[str, TokenEmbedder]) -> None

This is a TextFieldEmbedder that wraps a collection of TokenEmbedder objects. Each TokenEmbedders embeds its input, and the result is concatenated in an arbitrary (but consistent) order.

**token\_embedders : Dict[str, TokenEmbedder]**

A dictionary mapping token embedder names to implementations. These names should match the corresponding indexer used to generate the tensor passed to the TokenEmbedder.

**Registered as a TextFieldEmbedder with name "basic", which is also the default.**

**Input:**

**text\_field\_input: TextFieldTensors**

**Output: torch.Tensor**

## 三部分组合

**1. Using a word-level tokenizer** (such as SpacyTokenizer or WhitespaceTokenizer):

I. SingleIdTokenIndexer → Embedding (for things like GloVe or other simple embeddings, including learned POS tag embeddings)

II. TokenCharactersIndexer → TokenCharactersEncoder (for things like a character CNN)

III. ElmoTokenIndexer → ElmoTokenEmbedder (for ELMo)

IV. PretrainedTransformerMismatchedIndexer

→ PretrainedTransformerMismatchedEmbedder(for using a transformer like BERT when you really want to do modeling at the word level, e.g., for a tagging task; more on what this does below)

**2. Using a character-level tokenizer** (such as CharacterTokenizer):

I. SingleIdTokenIndexer → Embedding

**3. Using a wordpiece tokenizer** (such as PretrainedTransformerTokenizer):

I. PretrainedTransformerIndexer → PretrainedTransformerEmbedder

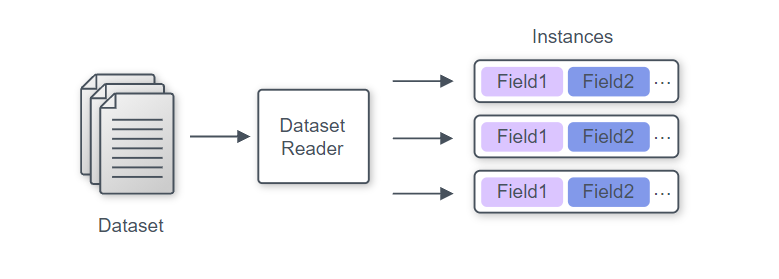
II. SingleIdTokenIndexer → Embedding (if you don't want contextualized wordpieces for some reason)

Contextualized representations in TextFields, like ELMo and BERT

Pretrained embeddings, like GloVe

## 实例Instance

字段→实例→数据集



实例为字典，其中key-value对应字段名-字段值

创建一个实例

instance = Instance(fields)

fields: Dict[str, Field] = {“f\_n-1”, field1, “f\_n-2”, field2}

Dataset readers 数据读取器

读取数据，得到实例迭代器（an iterable of Instances）

三步自定义

1. 继承DatasetReader并取名

2. 初始化\_\_init\_\_

3. 重写

text\_to\_instance()方法: fields→instance

\_read()方法: instance→instances

## Vocabulary

把为标记转化为整数ID提供映射，用**命名空间**（标记集合名）来区分，分为两类：Padded和Non-padded

按标记出现的频率索引

namespace一般取名: tokens和labels

创建方法: Vocabulary.from\_instances(instances)

from\_files ← “from\_files”

from\_files\_and\_instances ← “extend”

常用方法：

看标记的索引: get\_token\_index()

看索引所对应的标记: get\_token\_from\_index()

看索引列表get\_index\_to\_token\_vocabulary()

看vocab的大小: get\_vocab\_size

看vocab的命名空间: get\_namesapces()

## Dataset Loader

→ An iterable of batched tensor dictionaries

把Instance打包后转换成数值（本质是把instances的Tokens换成IDs）

常用数据加载器:

1. MutiProcesssDataloader

建立方法: data\_loader = MultiProcessDataLoader(reader: DatasetReader, path\_to\_data: str, batch\_size: int)

封装: BatchSampler

data\_loader = MultiProcessDataLoader(reader: DatasetReader, path\_to\_data: str, batch\_sampler=BucketBatchSampler())

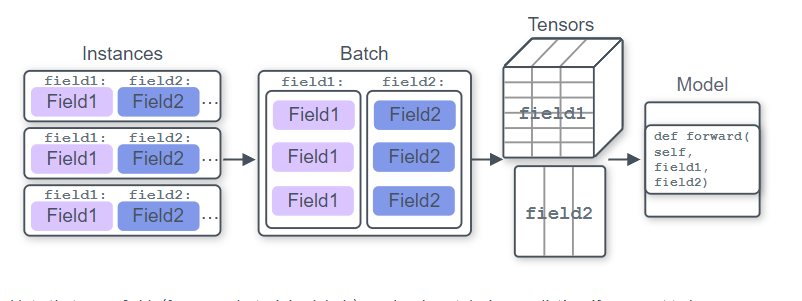
常用采样器: BucketBatchSample

2. SimpleDataLoader

利用Vocabulary数值化: data\_loader.index\_with(vocab: Vocabulary)

# Model

## 模型定义



1. 重写forward()方法

In forward, we use the parameters that we created in our constructor(def \_\_init\_\_(): ) to transform the inputs into outputs. After we've predicted the outputs, we compute some loss function based on how close we got to the true outputs, and then return that loss (along with whatever else we want) so that we can use it to train the parameters.

2. 定义get\_metrics()方法

常用方法：

make\_output\_human\_readable(): 结果可读↓

forward\_on\_instance()←Predictors

Takes a list of Instances, converts that text into arrays using this model's Vocabulary, passes those arrays through self.forward() and self.make\_output\_human\_readable() (which by default does nothing) and returns the result.

Metrics的get\_metric()方法: 自定义输出

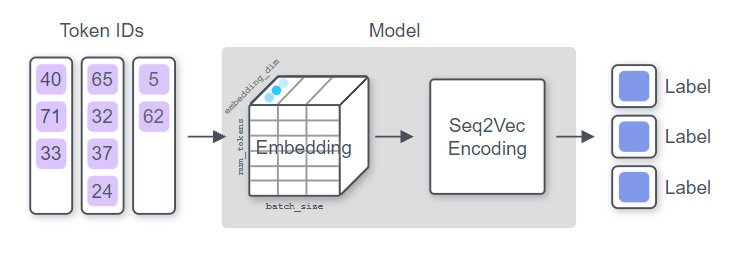
常用: CategoricalAccuracy()

TextFieldTensors:

filed\_name[tokennizer\_name][tokens]←→Dict[str, Dict[str, torch.Tensor]]

## 模型处理3大流程

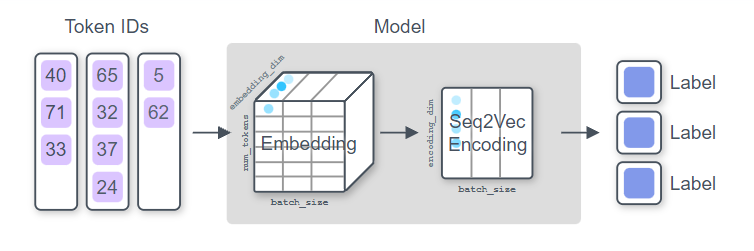
### Embedding tokens



Apply an Embedding function that converts each token ID that we got as input into a vector. This gives us a vector for each input token, so we have a large tensor here.

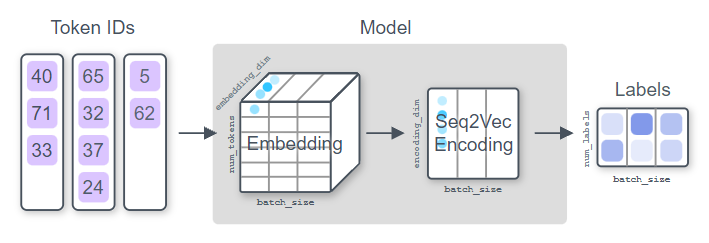
**Output: (batch\_size, num\_tokens, embedding\_dim).**

### Apply Seq2Vec encoder



Apply some function that takes the sequence of vectors for each input token and squashes it into a single vector. Before the days of pretrained language models like BERT, this was typically an LSTM or convolutional encoder.

### Computing distribution over labels



Take that single feature vector (for each Instance in the batch), and classify it as a label, which will give us a categorical probability distribution over our label space.

**Input: (batch\_size, num\_tokens, embedding\_dim)**

**Output: (batch\_size, encoding\_dim).**

## Saving Model

模型三要素：

1. Model config (specifications used to train the model)

Params class to\_file()方法

2. Model weights (trained parameters of the model)

model.state\_dict()方法（←torch.save()）

3. Vocabulary

Vocabulary.save\_to\_files()

进行封装: archive\_model()方法→tar.gz（自动调用）

## Loading Model

Model.load()方法

load\_archive()方法

# Common Architectures

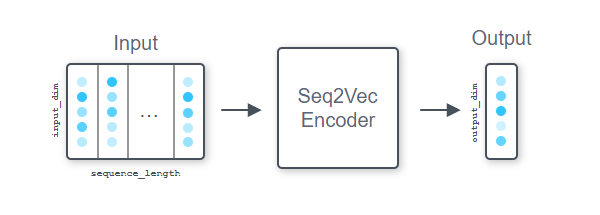
常用的neural architectures and AllenNLP abstractions

## Summarizing sequences

Take a sequence of vectors and summarize it to a single vector of fixed size.

Seq2VecEncoder:

(batch\_size, sequence\_length, input\_size) → (batch\_size, output\_size)



常用:

1. RNN: LstmSeq2VecEncoder and GruSeq2VecEncoder

2. CnnEncoder and BertPooler

### Seq2VecEncoder

A Seq2VecEncoder is a Module that takes as input a sequence of vectors and returns a single vector.

Input shape : (batch\_size, sequence\_length, input\_dim)

Output shape: (batch\_size, output\_dim).

#### 重要方法

1. get\_input\_dim()

2. get\_output\_dim()

#### CnnEncoder

A CnnEncoder is a combination of multiple convolution layers and max pooling layers.

Input shape (batch\_size, num\_tokens, input\_dim)

Output shape (batch\_size, output\_dim)

@Seq2VecEncoder.register("cnn")

class CnnEncoder(Seq2VecEncoder):

Registered as a Seq2VecEncoder with name "cnn"

**Parameters:**

embedding\_dim : int = Input dimension to the encoder

num\_filters : int

the output dim for each convolutional layer = the number of "filters"

ngram\_filter\_sizes : Tuple[int], optional (default = (2, 3, 4, 5))

the number of convolutional layers and sizes.

(2, 3, 4, 5) corresponding to encoding ngrams of size 2 to 5 with some number of filters.

conv\_layer\_activation : Activation, optional (default = torch.nn.ReLU)

Activation to use after the convolution layers

output\_dim : Optional[int], optional (default = None)

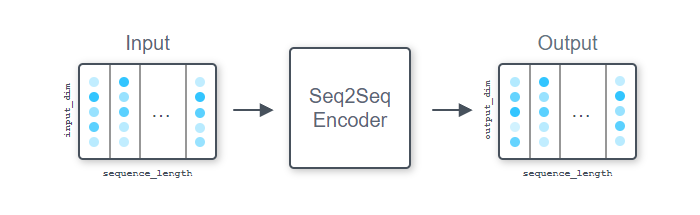
After doing convolutions and pooling, we'll project the collected features into a vector of this size. If this value is None, we will just return the result of the max pooling, giving an output of shape len(ngram\_filter\_sizes) \* num\_filters.

## Contextualizing sequences

Process a sequence of tokens and obtain another sequence of some embeddings.

Seq2SeqEncoder:

(batch\_size, sequence\_length, input\_dim) → (batch\_size, sequence\_length, output\_dim)



常用:

1. RNN: LstmSeq2SeqEncoder and GruSeq2SeqEncoder

2. FeedForwardEncoder

3. ComposeEncoder

4. PassThroughEncoder

## Similarities between sequences

Make use of attention to compute similarities (some sort of relatedness score) between sequences

AllenNLP provides two abstractions for attention—Attention and MatrixAttention.

### Attention

Attention modules compute similarities

**(each row of)** a matrix of size (batch\_size, sequence\_length, embedding\_dim)

a vector of size (batch\_size, embedding\_dim)

→a (typically normalized) similarity vector of size (batch\_size, sequence\_length)

常用:

1. DotProductAttention

Computes the dot product between the vector and each row of the matrix

2. BilinearAttention

Computes x^T W y + b for a given vector x and a matrix y using a matrix of weights W and a bias b ( The sizes of the embedding dimensions do not need to match)

3. LinearAttention

### MatrixAttention

MatrixAttention takes two matrices of size

(batch\_size, sequence\_length1, embedding\_dim)

(batch\_size, sequence\_length2, embedding\_dim)

→a (not normalized)matrix of size (batch\_size, sequence\_length1, sequence\_length2), which contains the similarity between each row of two input matrices

## Other common neural network building blocks

### FeedForward

### GatedSum and Highway layers

### TimeDistributed

### Conditional random field

### Activations

# Training and prediction

## Overall

Assuming that the model, dataset reader, and predictor we built so far are defined in some module named my\_text\_classifier, you would use the following AllenNLP commands to train the model, evaluate it, and make predictions for unseen data. Remember that you need to supply --include-package option so that AllenNLP can find your module. All the example code is set up in this repo. You just need to clone it, cd to the quick\_start directory, and run these commands just like this was your own project directory.

**train**

allennlp train my\_text\_classifier.jsonnet --serialization-dir model --include-package my\_text\_classifier

**evaluate**

allennlp evaluate model/model.tar.gz data/movie\_review/test.tsv --include-package my\_text\_classifier

**predict**

allennlp predict model/model.tar.gz data/movie\_review/test.jsonl --include-package my\_text\_classifier --predictor sentence\_classifier

**file structure**

quick\_start(package)

model(created\_folder)

vocabulary(folder)

model.tar.gz

config.json

metrcis.json

out.log

best.th

model\_state\*.th

train\_state\*.th

…

data(folder)

movie\_review(folder)

dev.tsv

test.tsv

train.tsv

test.jsonl

my\_text\_classfifier(package)

dataset\_readers(package)

classification\_tsv.py

\_\_init\_\_.py

models(package)

simple\_classifier.py

\_\_init\_\_.py

predictors(package)

sentence\_classifier\_predictor.py

\_\_init\_\_.py

\_\_init\_\_.py

my\_text\_classifier.jsonnet

my\_text\_classifier\_bert.jsonnet

evaluate.py

predict.py

train.py

\_\_init\_\_.py

## Training the model with own script

1. Testing your dataset reader

2. Feeding instances to the model

3. Training the model

### Trainer(GradientDescentTrainer)

@Trainer.register("gradient\_descent", constructor="from\_partial\_objects")

class GradientDescentTrainer(Trainer):

Registered as a Trainer with the name "gradient\_descent" (and is also the default Trainer).

A trainer for doing supervised learning with gradient descent. It just takes a labeled dataset and a DataLoader, and uses the supplied Optimizer to learn the weights for your model over some fixed number of epochs.

**Parameters**

* model : Model

An AllenNLP model to be optimized. Pytorch Modules can also be optimized if their forward method returns a dictionary with a "loss" key, containing a scalar tensor representing the loss function to be optimized.

* optimizer : torch.nn.Optimizer

An instance of a Pytorch Optimizer, instantiated with the parameters of the model to be optimized.

* data\_loader : DataLoader

A DataLoader containing your Dataset, yielding padded indexed batches.

* validation\_data\_loader : DataLoader, optional (default = None)

A DataLoader to use for the validation set. If None, then use the training DataLoader with the validation data.

* num\_epochs : int, optional (default = 20)

Number of training epochs.

* serialization\_dir : str, optional (default = None)

Path to directory for saving and loading model files. Models will not be saved if this parameter is not passed.

**重要方法**

* train(): Trains the supplied model with the supplied parameters.

## Training the model with allennlp train

We have a built-in training script that handles all of these things for you and makes it so the **only code that you have to write are your DatasetReader and Model classes.** Instead of writing all of the build\_\* methods that we had above, we **write a JSON configuration file specifying all necessary parameters.** Our training script takes those parameters, creates all of the objects in the right order, and runs the training loop.

### Configuration files

Configuration files in allennlp just take constructor parameters for various objects and put them into a JSON dictionary.

#### 两个重要的点

1. Select a particular subclass of a base type (e.g., SimpleClassifier as a subclass of Model, or BagOfEmbeddingsEncoder as a subclass of Seq2VecEncoder) we need an additional **"type"**: "simple\_classifier" key. The string "simple\_classifier" comes from the call to Model.register that we named before.

2. model 的参数中没有vocab参数. That's for the same reason that vocab was an argument to the build\_model method, not constructed inside it—the vocabulary gets constructed separately, based on data, then passed in to the model. Generally, the sequential dependencies between objects that show up as arguments to your build\_\* methods are left out of the configuration file, as they are handled in a different way.

#### Build configuration files —.jsonnet格式文件

常见参数：

dataset\_reader, train\_data\_path, validation\_data\_path, model, data\_loader, trainer

For the training loop, the object we're constructing is called **TrainModel**, and you can see its constructor here. The keys here must exactly match those parameters, otherwise you get a **ConfigurationError**.

#### Train the model

终端运行命令：

allennlp train [config.json] -s [serialization\_directory] --include-package [my\_python\_module]

config.json: 表示配置文件位置(后缀名.jsonnet也可以)

serialization\_directory: 训练结果保存位置

my\_python\_module: 为了使数据集读取器、模型和其他自定义组件能够被allennlp命令识别(找到注册的dataset\_reader, model)

实例（常用结构和方法）：

Project\_master目录结构

configs

config1.jsonnet

project\_name(python package)

\_\_init\_\_.py

readers(python package)

\_\_init\_\_.py

dataset\_reader1\_register\_name.py

models(python package)

\_\_init\_\_.py

model1\_register\_name.py

predictors(python package)

\_\_init\_\_.py

predictor1\_register\_name.py

data

download.sh

train.tsv

validation.tsv

test.tsv

results

train

0

1

终端运行命令：

allennlp train "configs/config1.jsonnet" -s "results/train/0" --include-package project\_name

### Evaluating the model

Evaluate the text classification model we just trained above, by computing the evaluation metrics against the test set.

#### Defining the metrics

#### Evaluating the model

evaluate() function

#### From command line

In order to evaluate your model from command line, you can use the **allennlp evaluate** command. This command takes the path to the model archive file created by the allennlp train command, along with the path to a file containing test instances, and returns the computed metrics.

**allennlp evaluate model/model.tar.gz data/movie\_review/test.tsv --include-package my\_text\_classifier**

model/model.tar.gz: 训练保存的模型

data/movie\_review/test.tsv: 待评估的instances的CSV格式文件.tsv，like

positive\_text \t pos

negative\_text \t neg

my\_text\_classifier: 编写dataset\_reader, model的package目录

### Making predictions for unlabeled inputs

Make predictions for new, unlabeled inputs using the trained text classification model.

#### Modifying the dataset reader

**DatasetReader中（必须）重写text\_to\_instanc()方法，供预测时调用**

#### Modifying the model

During prediction, the instances the model gets are not labeled. The forward() method doesn't need to (in fact, it can't) compute the loss—it just needs to **return the prediction.**

**修改model使其可以用来计算预测数据**

1. Make the label parameter optional by specifying a default value of None.

2. Compute the loss and accuracy only when the label is supplied.

#### Writing predictor

For making predictions in a demo setting, AllenNLP uses **Predictor**s, which are a thin wrapper around your trained model. A Predictor's main job is to take a JSON representation of an instance, convert it to an Instance using the dataset reader (the **text\_to\_instance mentioned** above), pass it through the model, and return the prediction in a JSON serializable format.

In order to build a Predictor for your task, you only need to inherit from Predictor and implement a few methods (see **predict() and \_json\_to\_instances()** below)—the rest will be taken care of by the base class.

**Note:** AllenNLP provides implementations of Predictors for common tasks. In fact, it includes TextClassifierPredictor, a generic Predictor for text classification tasks, so you don't even need to write your own! Here, we are writing one from scratch solely for demonstration, but you should **always check whether the predictor for your task is already there.**

1. 实现predic()方法

把text转化为J sonDict，调用predict\_json()方法

2. 重写\_json\_to\_instances方法

#### Making predictions

1. Training model as above.

2. Wrap the model with a SentenceClassifierPredictor to make predictions for new instances. (Because the returned result (output['probs']) is just an array of probabilities for class labels, we use **vocab.get\_token\_from\_index()** to convert a label ID back to its label string.)

→Get results similar to:

[('neg', 0.48853254318237305), ('pos', 0.511467456817627)]

#### From command line

Use **allennlp predict** command to make predictions. This command is very similar to allennlp evaluate (see above)—it takes the path to the model archive file created by the allennlp train command, along with the path to a JSON file containing serialized test instances, and runs the model against these instances.

**allennlp predict model/model.tar.gz data/movie\_review/test.jsonl --include-package my\_text\_classifier --predictor sentence\_classifier**

model/model.tar.gz: 训练保存的模型

data/movie\_review/test.jsonl: 待预测的instances的JSON-lines格式文件.jsonl，like

{“text”, text1}

{“text”, text2}

my\_text\_classifier: 编写dataset\_reader, model, predictor的package目录

sentence\_classifier: 预测器的注册名

# Optimizer

class Optimizer(torch.optim.Optimizer, Registrable)

## AdamOptimizer

@Optimizer.register("adam")

class AdamOptimizer(Optimizer, torch.optim.Adam):

Registered as an Optimizer with name "adam".

**Parameters**

model\_parameters: List[Tuple[str, torch.nn.Parameter]] 模型参数

model\_parameters = [(n, p) for n, p in model.named\_parameters() if p.requires\_grad]

# PyTorch常用模块

## torch.nn.Linear

Give a score (commonly called a logit) for each possible label.

## torch.nn.functional.softmax

Normalize those scores using a softmax operation to get a probability distribution over labels that we can return to a consumer of this model.

## torch.nn.functional.cross\_entropy

Computes the cross entropy between the logits and the true label distribution

# 需要了解的方法

model.forward\_on\_instances: 出现在Part1→Training and Prediction→1.Feeding instances to the model

encoded\_text = self.encoder(embedded\_text, mask): 出现在Part1→Training and Prediction→1.Training the model

BagOfEmbeddingsEncoder(embedding\_dim=10): 出现在Part1→Training and Prediction→1.Training the model

两个Instance实例对象的加法vocab = build\_vocab(training\_data + validation\_data):

出现在Part1→Training and Prediction→1.Training the model

DataLoader的学习→training\_loader.index\_with(vocab): 出现在Part1→Training and Prediction→1.Training the model（参见1.8 ）

**Optimizer: 出现在Tr ainer的参数中**

# API

## commands

### subcommand

Base class for subcommands under allennlp.run.

### train

The train subcommand can be used to train a model. It requires a configuration file and a directory in which to write the results.

### train\_model\_from\_file(函数—在python内部调用执行)

A wrapper around train\_model which loads the params from a file.

#### **Parameters**

* parameter\_filename : str 指定AllenNLP实验的json参数文件
* serialization\_dir : str 保存结果和日志的目录
* overrides : Union[str, Dict[str, Any]], optional (default = "")

将使用JSON字符串或dict覆盖输入参数文件中的值

* recover : bool, optional (default = False)

如果为True，我们将尝试从现有序列化目录恢复训练运行。这仅适用于在运行过程中实际发生故障的情况。

* force : bool, optional (default = False)

如果为True，我们将覆盖序列化目录（如果它已经存在）

* node\_rank : int, optional 分布式训练中当前节点的排名

分布式训练中当前节点的排名

* include\_package : str, optional 导入提到的额外packages

In distributed mode, extra packages mentioned will be imported in trainer workers.

* dry\_run : bool, optional (default = False)

不要训练模型，而是创建词汇表、显示数据集统计信息和其他训练信息。

* file\_friendly\_logging : bool, optional (default = False)

If True, we add newlines to tqdm output, even on an interactive terminal, and we slow down tqdm's output to only once every 10 seconds.

* return\_model : Optional[bool], optional (default = None)

Whether or not to return the final model. If not specified, this defaults to False for distributed training and True otherwise.

#### **Returns**

* best\_model : Optional[str]

The path to the archived model with the best weights or None if in dry run.

best\_model : Optional[Model] 具有最佳历元权重的模型

The model with the best epoch weights or None, depending on the value of return\_model and dry\_run.

### train\_model(函数—在python内部调用执行)

**Trains the model specified in the given Params object**, using the data and training parameters also specified in that object, and saves the results in serialization\_dir.

### TrainModel

class TrainModel(Registrable):

This class exists so that we can easily read a configuration file with the allennlp train command.

The basic logic:

train\_loop = TrainModel.from\_params(params\_from\_config\_file)

train\_loop.run()

This class performs very little logic, pushing most of it to the Trainer that has a train() method. Literally all we do after \_\_init\_\_ is call trainer.train(). You can do that yourself, if you've constructed a Trainer already. What this class gives you is a way to construct the Trainer by means of a config file. The actual constructor that we use with from\_params in this class is from\_partial\_objects. See that method for a description of all of the allowed top-level keys in a configuration file used with allennlp train.

**重要方法**

**from\_partial\_objects()**

@classmethod

def from\_partial\_objects(cls,…) -> "TrainModel":

This method is intended for use with our FromParams logic, to construct a TrainModel object from a config file passed to the allennlp train command. The arguments to this method are the allowed top-level keys in a configuration file (except for the first three, which are obtained separately).

The Lazy type annotations here are a mechanism for building dependencies to an object sequentially - the TrainModel object needs data, a model, and a trainer, but the model needs to see the data before it's constructed (to create a vocabulary) and the trainer needs the data and the model before it's constructed. Objects that have sequential dependencies like this are labeled as Lazy in their type annotations, and we pass the missing dependencies when we call their construct() method, which you can see in the code below. （惰性类型注释是一种按顺序构建对象依赖关系的机制）

**Parameters**

* *serialization\_dir* : str 将保存日志和模型存档的目录

The directory where logs and model archives will be saved.

In a typical AllenNLP configuration file, this parameter does not get an entry as a top-level key, it gets passed in separately.

* *local\_rank* : int 使用GPU设备id初始化的进程索引。

The process index that is initialized using the GPU device id.

In a typical AllenNLP configuration file, this parameter does not get an entry as a top-level key, it gets passed in separately.

* dataset\_reader : DatasetReader

The DatasetReader that will be used for training and (by default) for validation.

* train\_data\_path : str 传递给dataset\_reader.read()构造训练数据的文件目录

The file (or directory) that will be passed to dataset\_reader.read() to construct the training data.

* model : Lazy[Model]

The model that we will train. This is lazy because it depends on the Vocabulary; after constructing the vocabulary we call model.construct(vocab=vocabulary).

* data\_loader : Lazy[DataLoader]

The data\_loader we use to batch instances from the dataset reader at training and (by default) validation time. This is lazy because it takes a dataset in it's constructor.

* trainer : Lazy[Trainer]

The Trainer that actually implements the training loop. This is a lazy object because it depends on the model that's going to be trained.

* vocabulary : Lazy[Vocabulary], optional (default = Lazy(Vocabulary))

The Vocabulary that we will use to convert strings in the data to integer ids (and possibly set sizes of embedding matrices in the Model). By default we construct the vocabulary from the instances that we read.

datasets\_for\_vocab\_creation : List[str], optional (default = None)

If you pass in more than one dataset but don't want to use all of them to construct a vocabulary, you can pass in this key to limit it. Valid entries in the list are "train", "validation" and "test".

validation\_dataset\_reader : DatasetReader, optional (default = None)

If given, we will use this dataset reader for the validation data instead of dataset\_reader.

validation\_data\_path : str, optional (default = None)

If given, we will use this data for computing validation metrics and early stopping.

validation\_data\_loader : Lazy[DataLoader], optional (default = None)

If given, the data\_loader we use to batch instances from the dataset reader at validation and test time. This is lazy because it takes a dataset in it's constructor.

test\_data\_path : str, optional (default = None)

If given, we will use this as test data. This makes it available for vocab creation by default, but nothing else.

evaluate\_on\_test : bool, optional (default = False)

If given, we will evaluate the final model on this data at the end of training. Note that we do not recommend using this for actual test data in every-day experimentation; you should only very rarely evaluate your model on actual test data.

batch\_weight\_key : str, optional (default = "")

The name of metric used to weight the loss on a per-batch basis. This is only used during evaluation on final test data, if you've specified evaluate\_on\_test=True.

ddp\_accelerator : Optional[DdpAccelerator], optional (default = None)

A DdpAccelerator to use in distributed trainer. Passed to the model and the trainer.

**run()等价于call trainer.train()**

## common

### util

#### JsonDict

JsonDict = Dict[str, Any]

### Parmameter & Registrable

#### from\_params

#### params

#### registrable

## data

### data\_loaders

#### DataLoader

class DataLoader(Registrable)

A DataLoader is responsible for generating batches of instances from a **DatasetReader**, or another source of data.

This is purely an abstract base class. All concrete subclasses must provide implementations of the following methods:

* ***\_\_iter\_\_() that creates an iterable of TensorDicts***,

**TensorDict = Dict[str, Union[torch.Tensor, Dict[str, torch.Tensor]]]**

* iter\_instances() that creates an iterable of Instances,
* ***index\_with() that should index the data with a vocabulary***, and
* set\_target\_device(), which updates the device that batch tensors should be put it when they are generated in \_\_iter\_\_().

Additionally, this class should also implement \_\_len\_\_() when possible.

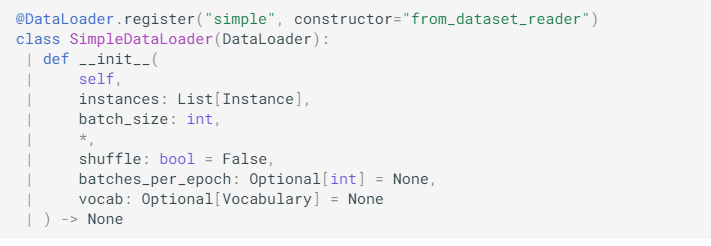
**The default implementation is MultiProcessDataLoader.**

#### SimpleDataLoader

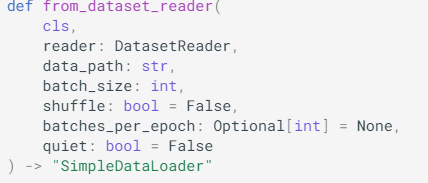
@DataLoader.register("simple", constructor="from\_dataset\_reader")

class SimpleDataLoader(DataLoader):

A very simple DataLoader that is mostly used for **testing.**



**重要方法：from\_dataset\_reader()**



## nn

### module

#### Module

class Module(torch.nn.Module)

This is just torch.nn.Module with some extra functionality.

## models

### model

#### Model

class Model(Module, Registrable):

Model is an abstract class representing an AllenNLP model.

## predictors

### Predictor

class Predictor(Registrable):



A Predictor is a thin wrapper around an AllenNLP model that **handles JSON -> JSON** predictions that can be used for serving models through the web API or making predictions in bulk.

**私有属性和方法：**

self.\_model = model

self.\_dataset\_reader = dataset\_reader

***\_json\_to\_instance(json\_dict: JsonDict) -> Instance***

把Json字典转换为Instance

***需要自己实现***

text = json\_dict[“text”]

return self.\_dataset\_reader.text\_to\_instance(text)

**重要方法：**

* load\_line(line: str) -> JsonDict:

return json.loads(line) 解码Json文件

* dump\_line(outputs: JsonDict) -> str:

return json.dumps(outputs) + “\n” 编码Json文件

* ***predict\_instance(instance: Instance) -> Jsondict:***

根据给定的model，dataset\_reader输出Instance的预测结果

* **predict\_json(inputs: Jsondict) -> JsonDict:**

根据给定的model，dataset\_reader输出inputs(JsonDict)的预测结果

***先调用\_json\_to\_instance再调用predict\_instance***

**具体操作：**

1. 定义 一个predict方法

def predict(self, text: str) -> JsonDict:

inputs = {“text”: text}

return predict\_json(inputs)

2. 重写方法json\_to\_instances()

def \_json\_to\_instance(self, json\_dict: JsonDict) -> Instance:

text = json\_dict[“text”]

return self.\_dataset\_reader.text\_to\_instance(text)

### TextClassifierPredictor

@Predictor.register("text\_classifier")

class TextClassifierPredictor(Predictor)

Predictor for any model that takes in a sentence and returns a single class for it. In particular, it can be used with the BasicClassifier model.

Registered as a Predictor with name "text\_classifier".

重要方法：

predict(text: str) -> : JsonDict: 返回预测结果

## training

### metrics

#### Metric

class Metric(Registrable)

A very general abstract class representing a metric which can be accumulated.

重要方法：

* get\_metric(): Compute and return the metric.
* 具有call方法

**Parameters**

* predictions : torch.Tensor

A tensor of predictions of shape (batch\_size, ..., num\_classes).

* gold\_labels : torch.Tensor

A tensor of integer class label of shape (batch\_size, ...). It must be the same shape as the predictions tensor without the num\_classes dimension.

* mask : torch.BoolTensor, optional (default = None)

A masking tensor the same size as gold\_labels.

#### CategoricalAccuracy

Metric.register("categorical\_accuracy")

class CategoricalAccuracy(Metric):

**Categorical Top-K accuracy**. Assumes integer labels, with each item to be classified having a single correct class. Tie break enables equal distribution of scores among the classes with same maximum predicted scores. (Computes the fraction of instances for which model predicted the label correctly)

应用步骤：

1. Create an instance of CategoricalAccuracy in model constructor:

self.accuracy = CategoricalAccuracy()

2. For each forward pass, update the metric by feeding the prediction and the gold labels:

self.accuracy(logits, label)

You can pull out the computed metric by calling get\_metrics(reset) with a flag specifying whether to reset the counts.

3. Implement the get\_metrics() method in model, which returns a dictionary from metric names to their values, as shown below:

def get\_metrics(self, reset: bool = False) -> Dict[str, float]:

return {"accuracy": self.accuracy.get\_metric(reset)}

AllenNLP's default training loop will call this method at the **appropriate times** and provide logging information with current metric values.

### util

#### evaluate

**Parameters**

* model : Model

The model to evaluate

* data\_loader : DataLoader

The DataLoader that will iterate over the evaluation data (data loaders already contain their data).

* cuda\_device : Union[int, torch.device], optional (default = -1)

The cuda device to use for this evaluation. The model is assumed to already be using this device; this parameter is only used for moving the input data to the correct device.

* batch\_weight\_key : str, optional (default = None)

If given, this is a key in the output dictionary for each batch that specifies how to weight the loss for that batch. If this is not given, we use a weight of 1 for every batch.

* metrics\_output\_file : str, optional (default = None)

Optional path to write the final metrics to.

* predictions\_output\_file : str, optional (default = None)

Optional path to write the predictions to.

**Returns**

Dict[str, Any]

The final metrics.

# Python 函数和模块

tempfile.TemporaryDirectory(): 出现在Part1→Training and Prediction→1.Training the model

json.dump(config, config\_file): 出现在Part1→Training and Prediction→1.Training the model with allennlp train

# 数据集

## 训练数据集（Training Set)

是一些我们已经知道输入和输出的数据集训练机器去学习，通过拟合去寻找模型的初始参数。例如在神经网络（Neural Networks)中， 我们用训练数据集和反向传播算法（Backpropagation）去每个神经元找到最优的比重（Weights)

## 验证数据集（Validation Set）

也是一些我们已经知道输入和输出的数据集，通过让机器学习去优化调整模型的参数，在神经网络中， 我们用验证数据集去寻找最优的网络深度（number of hidden layers)，或者决定反向传播算法的停止点；在普通的机器学习中常用的交叉验证（Cross Validation) 就是把训练数据集本身再细分成不同的验证数据集去训练模型

## 测试数据集（Test Set）

用户测试模型表现的数据集，根据误差（一般为预测输出与实际输出的不同）来判断一个模型的好坏

## 为什么验证数据集和测试数据集两者都需要？

因为验证数据集（Validation Set)用来调整模型参数从而选择最优模型，模型本身已经同时知道了输入和输出，所以从验证数据集上得出的误差（Error)会有偏差（Bias)

但是我们只用测试数据集(Test Set) 去评估模型的表现，并不会去调整优化模型

在传统的机器学习中，这三者一般的比例为training/validation/test = 50/25/25, 但是有些时候如果模型不需要很多调整只要拟合就可时，或者training本身就是training+validation (比如cross validation)时，也可以training/test =7/3

但是在深度学习中，由于数据量本身很大，而且训练神经网络需要的数据很多，可以把更多的数据分给training，而相应减少validation和test