Spinning Up Documentation

Release 1.1.1

Joshua Achiam

Mar 13, 2020

Overview

1 Our Goal 3

2 Performance 5

3 Tutorials 7

4 Quick start 9

5 Sponsors 11

6 Contributors 13

7 Reference 15

8 Contributors 17

We are proud to announce that we entirely rewrote Kashgari with tf.keras, now Kashgari comes with easier to understand API and is faster!

Kashgari is a simple and powerful NLP Transfer learning framework, build a state-of-art model in 5 minutes for named entity recognition (NER), part-of-speech tagging (PoS), and text classification tasks.

- **Human-friendly**. Kashgari's code is straightforward, well documented and tested, which makes it very easy to understand and modify.
- **Powerful and simple**. Kashgari allows you to apply state-of-the-art natural language processing (NLP) models to your text, such as named entity recognition (NER), part-of-speech tagging (PoS) and classification.
- **Built-in transfer learning**. Kashgari built-in pre-trained BERT and Word2vec embedding models, which makes it very simple to transfer learning to train your model.
- **Fully scalable**. Kashgari provides a simple, fast, and scalable environment for fast experimentation, train your models and experiment with new approaches using different embeddings and model structure.
- **Production Ready**. Kashgari could export model with `SavedModel` format for tensorflow serving, you could directly deploy it on the cloud.

CHAPTER 1

Our Goal

- **Academic users** Easier experimentation to prove their hypothesis without coding from scratch.
- **NLP beginners** Learn how to build an NLP project with production level code quality.
- **NLP developers** Build a production level classification/labeling model within minutes.

CHAPTER 2

Performance

Task	Language	Dataset	Score	Detail
Named Entity Recognition	Chinese	People's Daily Ner Corpus	**94.46** (F1)	Text Labeling Performance Report

CHAPTER 3

Tutorials

Here is a set of quick tutorials to get you started with the library:

- Tutorial 1: Text Classification
- Tutorial 2: Text Labeling
- Tutorial 3: Text Scoring
- Tutorial 4: Language Embedding

There are also articles and posts that illustrate how to use Kashgari:

- 15
- BERT NER)
- BERT/ERNIE
- BERTNER
- Multi-Class Text Classification with Kashgari in 15 minutes

CHAPTER 4

Quick start

4.1 Requirements and Installation

We renamed again for consistency and clarity. From now on, it is all `kashgari`.

The project is based on Python 3.6+, because it is 2019 and type hinting is cool.

Backend	pypi version	desc
TensorFlow 2.x	`pip install 'kashgari>=2.0.0'`	coming soon
TensorFlow 1.14+	`pip install 'kashgari>=1.0.0,<2.0.0'`	current version
Keras	`pip install 'kashgari<1.0.0'`	legacy version

Find more info about the name changing.

4.2 Example Usage

Let's run an NER labeling model with Bi_LSTM Model.

```
from kashgari.corpus import ChineseDailyNerCorpus
from kashgari.tasks.labeling import BiLSTM_Model

train_x, train_y = ChineseDailyNerCorpus.load_data('train')
test_x, test_y = ChineseDailyNerCorpus.load_data('test')
valid_x, valid_y = ChineseDailyNerCorpus.load_data('valid')

model = BiLSTM_Model()
model.fit(train_x, train_y, valid_x, valid_y, epochs=50)

"""
_________________________________________________________________
Layer (type)                 Output Shape              Param #
```

(continues on next page)

(continued from previous page)

```
=================================================================
input (InputLayer)           (None, 97)                0
_________________________________________________________________
layer_embedding (Embedding)  (None, 97, 100)           320600
_________________________________________________________________
layer_blstm (Bidirectional)  (None, 97, 256)           235520
_________________________________________________________________
layer_dropout (Dropout)      (None, 97, 256)           0
_________________________________________________________________
layer_time_distributed (Time (None, 97, 8)             2056
_________________________________________________________________
activation_7 (Activation)    (None, 97, 8)             0
=================================================================
Total params: 558,176
Trainable params: 558,176
Non-trainable params: 0
_________________________________________________________________
Train on 20864 samples, validate on 2318 samples
Epoch 1/50
20864/20864 [==============================] - 9s 417us/sample - loss: 0.2508 - acc:␣
↪0.9333 - val_loss: 0.1240 - val_acc: 0.9607

"""
```

4.3 Run with GPT-2 Embedding

```
from kashgari.embeddings import GPT2Embedding
from kashgari.corpus import ChineseDailyNerCorpus
from kashgari.tasks.labeling import BiGRU_Model

train_x, train_y = ChineseDailyNerCorpus.load_data('train')
valid_x, valid_y = ChineseDailyNerCorpus.load_data('valid')

gpt2_embedding = GPT2Embedding('<path-to-gpt-model-folder>', sequence_length=30)
model = BiGRU_Model(gpt2_embedding)
model.fit(train_x, train_y, valid_x, valid_y, epochs=50)
```

4.4 Run with Bert Embedding

```
from kashgari.embeddings import BERTEmbedding
from kashgari.tasks.labeling import BiGRU_Model
from kashgari.corpus import ChineseDailyNerCorpus

bert_embedding = BERTEmbedding('<bert-model-folder>', sequence_length=30)
model = BiGRU_Model(bert_embedding)

train_x, train_y = ChineseDailyNerCorpus.load_data()
model.fit(train_x, train_y)
```

CHAPTER 5

Sponsors

Support this project by becoming a sponsor. Your issues and feature request will be prioritized.[Become a sponsor]

CHAPTER 6

Contributors

Thanks goes to these wonderful people. And there are many ways to get involved. Start with the contributor guidelines and then check these open issues for specific tasks.

Feel free to join the Slack group if you want to more involved in Kashgari's development.

Slack Group Link

CHAPTER 7

Reference

This library is inspired by and references following frameworks and papers.

- flair - A very simple framework for state-of-the-art Natural Language Processing (NLP)
- anago - Bidirectional LSTM-CRF and ELMo for Named-Entity Recognition, Part-of-Speech Tagging
- Chinese-Word-Vectors

This project follows the all-contributors specification. Contributions of any kind welcome!

CHAPTER 8

Contributors

8.1 Code Contributors

This project exists thanks to all the people who contribute. [Contribute].

8.2 Financial Contributors

Become a financial contributor and help us sustain our community. [Contribute]

8.2.1 Individuals

8.2.2 Organizations

Support this project with your organization. Your logo will show up here with a link to your website. [Contribute]

We are proud to announce that we entirely rewrote Kashgari with tf.keras, now Kashgari comes with easier to understand API and is faster!

Overview

Kashgari is a simple and powerful NLP Transfer learning framework, build a state-of-art model in 5 minutes for named entity recognition (NER), part-of-speech tagging (PoS), and text classification tasks.

- **Human-friendly**. Kashgari's code is straightforward, well documented and tested, which makes it very easy to understand and modify.
- **Powerful and simple**. Kashgari allows you to apply state-of-the-art natural language processing (NLP) models to your text, such as named entity recognition (NER), part-of-speech tagging (PoS) and classification.
- **Built-in transfer learning**. Kashgari built-in pre-trained BERT and Word2vec embedding models, which makes it very simple to transfer learning to train your model.

- **Fully scalable**. Kashgari provides a simple, fast, and scalable environment for fast experimentation, train your models and experiment with new approaches using different embeddings and model structure.
- **Production Ready**. Kashgari could export model with `SavedModel` format for tensorflow serving, you could directly deploy it on the cloud.

Our Goal

- **Academic users** Easier experimentation to prove their hypothesis without coding from scratch.
- **NLP beginners** Learn how to build an NLP project with production level code quality.
- **NLP developers** Build a production level classification/labeling model within minutes.

Performance

Task	Language	Dataset	Score	Detail
Named Entity Recognition	Chinese	People's Daily Ner Corpus	**94.46** (F1)	Text Labeling Performance Report

Tutorials

Here is a set of quick tutorials to get you started with the library:

- Tutorial 1: Text Classification
- Tutorial 2: Text Labeling
- Tutorial 3: Text Scoring
- Tutorial 4: Language Embedding

There are also articles and posts that illustrate how to use Kashgari:

- 15
- BERT NER)
- BERT/ERNIE
- BERTNER
- Multi-Class Text Classification with Kashgari in 15 minutes

Quick start

Requirements and Installation

We renamed again for consistency and clarity. From now on, it is all `kashgari`.

The project is based on Python 3.6+, because it is 2019 and type hinting is cool.

Backend	pypi version	desc
TensorFlow 2.x	`pip install 'kashgari>=2.0.0'`	coming soon
TensorFlow 1.14+	`pip install 'kashgari>=1.0.0,<2.0.0'`	current version
Keras	`pip install 'kashgari<1.0.0'`	legacy version

Find more info about the name changing.

Example Usage

Let's run an NER labeling model with Bi_LSTM Model.

```
from kashgari.corpus import ChineseDailyNerCorpus
from kashgari.tasks.labeling import BiLSTM_Model

train_x, train_y = ChineseDailyNerCorpus.load_data('train')
test_x, test_y = ChineseDailyNerCorpus.load_data('test')
valid_x, valid_y = ChineseDailyNerCorpus.load_data('valid')

model = BiLSTM_Model()
model.fit(train_x, train_y, valid_x, valid_y, epochs=50)

"""
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           (None, 97)                0
_________________________________________________________________
layer_embedding (Embedding)  (None, 97, 100)           320600
_________________________________________________________________
layer_blstm (Bidirectional)  (None, 97, 256)           235520
_________________________________________________________________
layer_dropout (Dropout)      (None, 97, 256)           0
_________________________________________________________________
layer_time_distributed (Time (None, 97, 8)             2056
_________________________________________________________________
activation_7 (Activation)    (None, 97, 8)             0
=================================================================
Total params: 558,176
Trainable params: 558,176
Non-trainable params: 0
_________________________________________________________________
Train on 20864 samples, validate on 2318 samples
Epoch 1/50
20864/20864 [==============================] - 9s 417us/sample - loss: 0.2508 - acc:␣
↪0.9333 - val_loss: 0.1240 - val_acc: 0.9607

"""
```

Run with GPT-2 Embedding

```
from kashgari.embeddings import GPT2Embedding
from kashgari.corpus import ChineseDailyNerCorpus
from kashgari.tasks.labeling import BiGRU_Model

train_x, train_y = ChineseDailyNerCorpus.load_data('train')
valid_x, valid_y = ChineseDailyNerCorpus.load_data('valid')

gpt2_embedding = GPT2Embedding('<path-to-gpt-model-folder>', sequence_length=30)
model = BiGRU_Model(gpt2_embedding)
model.fit(train_x, train_y, valid_x, valid_y, epochs=50)
```

Run with Bert Embedding

```
from kashgari.embeddings import BERTEmbedding
from kashgari.tasks.labeling import BiGRU_Model
from kashgari.corpus import ChineseDailyNerCorpus

bert_embedding = BERTEmbedding('<bert-model-folder>', sequence_length=30)
model = BiGRU_Model(bert_embedding)

train_x, train_y = ChineseDailyNerCorpus.load_data()
model.fit(train_x, train_y)
```

Sponsors

Support this project by becoming a sponsor. Your issues and feature request will be prioritized.[Become a sponsor]

Contributors

Thanks goes to these wonderful people. And there are many ways to get involved. Start with the contributor guidelines and then check these open issues for specific tasks.

Feel free to join the Slack group if you want to more involved in Kashgari's development.

Slack Group Link

Reference

This library is inspired by and references following frameworks and papers.

- flair - A very simple framework for state-of-the-art Natural Language Processing (NLP)
- anago - Bidirectional LSTM-CRF and ELMo for Named-Entity Recognition, Part-of-Speech Tagging
- Chinese-Word-Vectors

This project follows the all-contributors specification. Contributions of any kind welcome!

Contributors

Code Contributors

This project exists thanks to all the people who contribute. [Contribute].

Financial Contributors

Become a financial contributor and help us sustain our community. [Contribute]

Individuals

Organizations

Support this project with your organization. Your logo will show up here with a link to your website. [Contribute]

Frequently Asked Questions

How can I run Keras on GPU

Kashgari will use GPU by default if available, but you need to setup the Tensorflow GPU environment first. You can check gpu status using the code below:

```
import tensorflow as tf
print(tf.test.is_gpu_available())
```

Here is the official document of TensorFlow-GPU

How to specify witch CPU or GPU for training and prediction

TensorFlow allows to use specific device for train and predict.

```
from tensorflow.python.client import device_lib
print(device_lib.list_local_devices())

# This will return the device list
"""[name: "/device:CPU:0"
 device_type: "CPU"
 memory_limit: 268435456
 locality {
 }
 incarnation: 3933047686559574430
 physical_device_desc: "device: XLA_GPU device", name: "/device:GPU:0"
 device_type: "GPU"
 memory_limit: 11330115994
 locality {
   bus_id: 1
   links {
   }
 }
 incarnation: 16701328925727941592
 physical_device_desc: "device: 0, name: Tesla K80, pci bus id: 0000:00:04.0, compute␣
↪capability: 3.7"]"""

# Then specific which device to use, for example to use CPU for prediction

with tf.device('/device:CPU:0'):
    model.predict(x)

# Or use second GPU for training
with tf.device('/device:GPU:1'):
    model.fit(test_x, test_y, valid_x, valid_y, epochs=1)
```

How to save and resume training with ModelCheckpoint callback

You can use tf.keras.callbacks.ModelCheckpoint for saving model during training.

```
from tensorflow.python.keras.callbacks import ModelCheckpoint

filepath = "saved-model-{epoch:02d}-{acc:.2f}.hdf5"
```

(continues on next page)

(continued from previous page)

```
checkpoint_callback = ModelCheckpoint(filepath,
                                      monitor = 'acc',
                                      verbose = 1)

model = CNN_GRU_Model()
model.fit(train_x,
          train_y,
          valid_x,
          valid_y,
          callbacks=[checkpoint_callback])
```

ModelCheckpoint will save models struct and weights to target file, but we need token dict and label dict to fully restore the model, so we have to save model using `model.save()` function.

So, the full solution will be like this.

```
from tensorflow.python.keras.callbacks import ModelCheckpoint

filepath = "saved-model-{epoch:02d}-{acc:.2f}.hdf5"
checkpoint_callback = ModelCheckpoint(filepath,
                                      monitor = 'acc',
                                      verbose = 1)

model = CNN_GRU_Model()

# This function will build token dict, label dict and model struct.
model.build_model(train_x, train_y, valid_x, valid_y)
# Save full model info and initial weights to the full_model folder.
model.save('full_model')

# Start Training
model.fit(train_x,
          train_y,
          valid_x,
          valid_y,
          callbacks=[checkpoint_callback])

# Load Model
from kashgari.utils import load_model

# We only need model struct and dicts
new_model = load_model('full_model', load_weights=False)
# Load weights from ModelCheckpoint
new_model.tf_model.load_weights('saved-model-05-0.96.hdf5')

# Resume Training
# Only need to set {'initial_epoch': 5} when you wish to start new epoch from 6
# Otherwise epoch will start from 1
model.fit(train_x,
          train_y,
          valid_x,
          valid_y,
          callbacks=[checkpoint_callback],
          epochs=10,
          fit_kwargs={'initial_epoch': 5})
```

Text Classification Model

Kashgari provides several models for text classification, All labeling models inherit from the `BaseClassificationModel`. You could easily switch from one model to another just by changing one line of code.

Available Models

Train basic classification model

Kashgari provices basic intent-classification corpus for expirement. You could also use your corpus in any language for training.

```
# Load build-in corpus.
from kashgari.corpus import SMP2018ECDTCorpus

train_x, train_y = SMP2018ECDTCorpus.load_data('train')
valid_x, valid_y = SMP2018ECDTCorpus.load_data('valid')
test_x, test_y = SMP2018ECDTCorpus.load_data('test')

# Or use your own corpus
train_x = [['Hello', 'world'], ['Hello', 'Kashgari']]
train_y = ['a', 'b']

valid_x, valid_y = train_x, train_y
test_x, test_x = train_x, train_y
```

Then train our first model. All models provided some APIs, so you could use any labeling model here.

```
import kashgari
from kashgari.tasks.classification import BiLSTM_Model

import logging
logging.basicConfig(level='DEBUG')

model = BiLSTM_Model()
model.fit(train_x, train_y, valid_x, valid_y)

# Evaluate the model
model.evaluate(test_x, test_y)

# Model data will save to `saved_ner_model` folder
model.save('saved_classification_model')

# Load saved model
loaded_model = kashgari.utils.load_model('saved_classification_model')
loaded_model.predict(test_x[:10])

# To continue training, compile the newly loaded model first
loaded_model.compile_model()
model.fit(train_x, train_y, valid_x, valid_y)
```

That's all your need to do. Easy right.

Text classification with transfer learning

Kashgari provides varies Language model Embeddings for transfer learning. Here is the example for BERT Embedding.

```
import kashgari
from kashgari.tasks.classification import BiGRU_Model
from kashgari.embeddings import BERTEmbedding

import logging
logging.basicConfig(level='DEBUG')

bert_embed = BERTEmbedding('<PRE_TRAINED_BERT_MODEL_FOLDER>',
                           task=kashgari.CLASSIFICATION,
                           sequence_length=100)
model = BiGRU_Model(bert_embed)
model.fit(train_x, train_y, valid_x, valid_y)
```

You could replace bert_embedding with any Embedding class in `kashgari.embeddings`. More info about Embedding: LINK THIS.

Adjust model's hyper-parameters

You could easily change model's hyper-parameters. For example, we change the lstm unit in `BiLSTM_Model` from 128 to 32.

```
from kashgari.tasks.classification import BiLSTM_Model

hyper = BiLSTM_Model.get_default_hyper_parameters()
print(hyper)
# {'layer_bi_lstm': {'units': 128, 'return_sequences': False}, 'layer_dense': {
↪'activation': 'softmax'}}

hyper['layer_bi_lstm']['units'] = 32

model = BiLSTM_Model(hyper_parameters=hyper)
```

Use custom optimizer

Kashgari already supports using customized optimizer, like RAdam.

```
from kashgari.corpus import SMP2018ECDTCorpus
from kashgari.tasks.classification import BiLSTM_Model
# Remember to import kashgari before than RAdam
from keras_radam import RAdam

train_x, train_y = SMP2018ECDTCorpus.load_data('train')
valid_x, valid_y = SMP2018ECDTCorpus.load_data('valid')
test_x, test_y = SMP2018ECDTCorpus.load_data('test')

model = BiLSTM_Model()
# This step will build token dict, label dict and model structure
model.build_model(train_x, train_y, valid_x, valid_y)
# Compile model with custom optimizer, you can also customize loss and metrics.
```

(continues on next page)

(continued from previous page)

```
optimizer = RAdam()
model.compile_model(optimizer=optimizer)

# Train model
model.fit(train_x, train_y, valid_x, valid_y)
```

Use callbacks

Kashgari is based on keras so that you could use all of the tf.keras callbacks directly with Kashgari model. For example, here is how to visualize training with tensorboard.

```
from tensorflow.python import keras
from kashgari.tasks.classification import BiGRU_Model
from kashgari.callbacks import EvalCallBack

import logging
logging.basicConfig(level='DEBUG')

model = BiGRU_Model()

tf_board_callback = keras.callbacks.TensorBoard(log_dir='./logs', update_freq=1000)

# Build-in callback for print precision, recall and f1 at every epoch step
eval_callback = EvalCallBack(kash_model=model,
                             valid_x=valid_x,
                             valid_y=valid_y,
                             step=5)

model.fit(train_x,
          train_y,
          valid_x,
          valid_y,
          batch_size=100,
          callbacks=[eval_callback, tf_board_callback])
```

Multi-Label Classification

Kashgari support multi-label classification, Here is how we build one.

Let's assume we have a dataset like this.

```
x = [
   ['This','news','are' , 'very','well','organized'],
   ['What','extremely','usefull','tv','show'],
   ['The','tv','presenter','were','very','well','dress'],
   ['Multi-class', 'classification', 'means', 'a', 'classification', 'task', 'with',
↪'more', 'than', 'two', 'classes']
]

y = [
   ['A', 'B'],
   ['A',],
   ['B', 'C'],
```

(continues on next page)

(continued from previous page)

```
    []
]
```

Now we need to init a `Processor` and `Embedding` for our model, then prepare model and fit.

```
from kashgari.tasks.classification import BiLSTM_Model
from kashgari.processors import ClassificationProcessor
from kashgari.embeddings import BareEmbedding

import logging
logging.basicConfig(level='DEBUG')

processor = ClassificationProcessor(multi_label=True)
embed = BareEmbedding(processor=processor)

model = BiLSTM_Model(embed)
model.fit(x, y)
```

Customize your own model

It is very easy and straightforward to build your own customized model, just inherit the `BaseClassificationModel` and implement the `get_default_hyper_parameters()` function and `build_model_arc()` function.

```
from typing import Dict, Any

from tensorflow import keras

from kashgari.tasks.classification.base_model import BaseClassificationModel
from kashgari.layers import L

import logging
logging.basicConfig(level='DEBUG')

class DoubleBLSTMModel(BaseClassificationModel):
    """Bidirectional LSTM Sequence Labeling Model"""

    @classmethod
    def get_default_hyper_parameters(cls) -> Dict[str, Dict[str, Any]]:
        """
        Get hyper parameters of model
        Returns:
            hyper parameters dict
        """
        return {
            'layer_blstm1': {
                'units': 128,
                'return_sequences': True
            },
            'layer_blstm2': {
                'units': 128,
                'return_sequences': False
            },
            'layer_dropout': {
                'rate': 0.4
```

(continues on next page)

(continued from previous page)

```
            },
            'layer_time_distributed': {},
            'layer_activation': {
                'activation': 'softmax'
            }
        }

    def build_model_arc(self):
        """
        build model architectural
        """
        output_dim = len(self.processor.label2idx)
        config = self.hyper_parameters
        embed_model = self.embedding.embed_model

        # Define your layers
        layer_blstm1 = L.Bidirectional(L.LSTM(**config['layer_blstm1']),
                                       name='layer_blstm1')
        layer_blstm2 = L.Bidirectional(L.LSTM(**config['layer_blstm2']),
                                       name='layer_blstm2')

        layer_dropout = L.Dropout(**config['layer_dropout'],
                                  name='layer_dropout')

        layer_time_distributed = L.TimeDistributed(L.Dense(output_dim,
                                                           **config['layer_time_
↪distributed']),
                                                   name='layer_time_distributed')
        layer_activation = L.Activation(**config['layer_activation'])

        # Define tensor flow
        tensor = layer_blstm1(embed_model.output)
        tensor = layer_blstm2(tensor)
        tensor = layer_dropout(tensor)
        tensor = layer_time_distributed(tensor)
        output_tensor = layer_activation(tensor)

        # Init model
        self.tf_model = keras.Model(embed_model.inputs, output_tensor)

model = DoubleBLSTMModel()
model.fit(train_x, train_y, valid_x, valid_y)
```

Speed up with CuDNN cell

You can speed up training and inferencing process using CuDNN cell. CuDNNLSTM and CuDNNGRU layers are much faster than LSTM and GRU layer, but they must be used on GPU. If you want to train on GPU and inferencing on CPU, you cannot use CuDNN cells.

```
# Enable use cudnn cell
kashgari.config.use_cudnn_cell = True
```

Text Labeling Model

Kashgari provides several models for text labeling, All labeling models inherit from the `BaseLabelingModel`. You could easily switch from one model to another just by changing one line of code.

Available Models

Train basic NER model

Kashgari provices basic NER corpus for expirement. You could also use your corpus in any language for training.

```
# Load build-in corpus.
## For Chinese
from kashgari.corpus import ChineseDailyNerCorpus

train_x, train_y = ChineseDailyNerCorpus.load_data('train')
valid_x, valid_y = ChineseDailyNerCorpus.load_data('valid')
test_x, test_y = ChineseDailyNerCorpus.load_data('test')

## For English
from kashgari.corpus import CONLL2003ENCorpus

train_x, train_y = CONLL2003ENCorpus.load_data('train')
valid_x, valid_y = CONLL2003ENCorpus.load_data('valid')
test_x, test_y = CONLL2003ENCorpus.load_data('test')

# Or use your own corpus
train_x = [['Hello', 'world'], ['Hello', 'Kashgari'], ['I', 'love', 'Beijing']]
train_y = [['O', 'O'], ['O', 'B-PER'], ['O', 'B-LOC']]

valid_x, valid_y = train_x, train_y
test_x, test_x = train_x, train_y
```

Or use your own corpus, it needs to be tokenized like this.

```
>>> print(train_x[0])
['', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '']

>>> print(train_y[0])
['O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-LOC', 'I-LOC', 'O', 'B-LOC', 'I-LOC', 'O', 'O',
↪ 'O', 'O', 'O', 'O']
```

Then train our first model. All models provided some APIs, so you could use any labeling model here.

```
import kashgari
from kashgari.tasks.labeling import BLSTMModel

model = BLSTMModel()
model.fit(train_x, train_y, valid_x, valid_y)

# Evaluate the model

model.evaluate(test_x, test_y)

# Model data will save to `saved_ner_model` folder
model.save('saved_ner_model')
```

(continues on next page)

(continued from previous page)

```
# Load saved model
loaded_model = kashgari.utils.load_model('saved_ner_model')
loaded_model.predict(test_x[:10])

# To continue training, compile the newly loaded model first
loaded_model.compile_model()
model.fit(train_x, train_y, valid_x, valid_y)
```

That's all your need to do. Easy right.

Sequence labeling with transfer learning

Kashgari provides varies Language model Embeddings for transfer learning. Here is the example for BERT Embedding.

```
import kashgari
from kashgari.tasks.labeling import BLSTMModel
from kashgari.embeddings import BERTEmbedding

bert_embed = BERTEmbedding('<PRE_TRAINED_BERT_MODEL_FOLDER>',
                           task=kashgari.LABELING,
                           sequence_length=100)
model = BLSTMModel(bert_embed)
model.fit(train_x, train_y, valid_x, valid_y)
```

You could replace bert_embedding with any Embedding class in `kashgari.embeddings`. More info about Embedding: LINK THIS.

Adjust model's hyper-parameters

You could easily change model's hyper-parameters. For example, we change the lstm unit in `BLSTMModel` from 128 to 32.

```
from kashgari.tasks.labeling import BLSTMModel

hyper = BLSTMModel.get_default_hyper_parameters()
print(hyper)
# {'layer_blstm': {'units': 128, 'return_sequences': True}, 'layer_dropout': {'rate':␣
↪0.4}, 'layer_time_distributed': {}, 'layer_activation': {'activation': 'softmax'}}

hyper['layer_blstm']['units'] = 32

model = BLSTMModel(hyper_parameters=hyper)
```

Use custom optimizer

Kashgari already supports using customized optimizer, like RAdam.

```
from kashgari.corpus import SMP2018ECDTCorpus
from kashgari.tasks.classification import BiLSTM_Model
```

(continues on next page)

(continued from previous page)

```
# Remember to import kashgari before than RAdam
from keras_radam import RAdam

train_x, train_y = SMP2018ECDTCorpus.load_data('train')
valid_x, valid_y = SMP2018ECDTCorpus.load_data('valid')
test_x, test_y = SMP2018ECDTCorpus.load_data('test')

model = BiLSTM_Model()
# This step will build token dict, label dict and model structure
model.build_model(train_x, train_y, valid_x, valid_y)
# Compile model with custom optimizer, you can also customize loss and metrics.
optimizer = RAdam()
model.compile_model(optimizer=optimizer)

# Train model
model.fit(train_x, train_y, valid_x, valid_y)
```

Use callbacks

Kashgari is based on keras so that you could use all of the tf.keras callbacks directly with Kashgari model. For example, here is how to visualize training with tensorboard.

```
from tensorflow.python import keras
from kashgari.tasks.labeling import BLSTMModel
from kashgari.callbacks import EvalCallBack

model = BLSTMModel()

tf_board_callback = keras.callbacks.TensorBoard(log_dir='./logs', update_freq=1000)

# Build-in callback for print precision, recall and f1 at every epoch step
eval_callback = EvalCallBack(kash_model=model,
                             valid_x=valid_x,
                             valid_y=valid_y,
                             step=5)

model.fit(train_x,
          train_y,
          valid_x,
          valid_y,
          batch_size=100,
          callbacks=[eval_callback, tf_board_callback])
```

Customize your own model

It is very easy and straightforward to build your own customized model, just inherit the `BaseLabelingModel` and implement the `get_default_hyper_parameters()` function and `build_model_arc()` function.

```
from typing import Dict, Any

from tensorflow import keras

from kashgari.tasks.labeling.base_model import BaseLabelingModel
```

(continues on next page)

(continued from previous page)

```
from kashgari.layers import L

import logging
logging.basicConfig(level='DEBUG')

class DoubleBLSTMModel(BaseLabelingModel):
    """Bidirectional LSTM Sequence Labeling Model"""

    @classmethod
    def get_default_hyper_parameters(cls) -> Dict[str, Dict[str, Any]]:
        """
        Get hyper parameters of model
        Returns:
            hyper parameters dict
        """
        return {
            'layer_blstm1': {
                'units': 128,
                'return_sequences': True
            },
            'layer_blstm2': {
                'units': 128,
                'return_sequences': True
            },
            'layer_dropout': {
                'rate': 0.4
            },
            'layer_time_distributed': {},
            'layer_activation': {
                'activation': 'softmax'
            }
        }

    def build_model_arc(self):
        """
        build model architectural
        """
        output_dim = len(self.processor.label2idx)
        config = self.hyper_parameters
        embed_model = self.embedding.embed_model

        # Define your layers
        layer_blstm1 = L.Bidirectional(L.LSTM(**config['layer_blstm1']),
                                       name='layer_blstm1')
        layer_blstm2 = L.Bidirectional(L.LSTM(**config['layer_blstm2']),
                                       name='layer_blstm2')

        layer_dropout = L.Dropout(**config['layer_dropout'],
                                  name='layer_dropout')

        layer_time_distributed = L.TimeDistributed(L.Dense(output_dim,
                                                           **config['layer_time_
↪distributed']),
                                                   name='layer_time_distributed')
        layer_activation = L.Activation(**config['layer_activation'])

        # Define tensor flow
```

(continues on next page)

(continued from previous page)

```
        tensor = layer_blstm1(embed_model.output)
        tensor = layer_blstm2(tensor)
        tensor = layer_dropout(tensor)
        tensor = layer_time_distributed(tensor)
        output_tensor = layer_activation(tensor)

        # Init model
        self.tf_model = keras.Model(embed_model.inputs, output_tensor)

model = DoubleBLSTMModel()
model.fit(train_x, train_y, valid_x, valid_y)
```

Speed up using CuDNN cell

You can speed up training and inferencing process using CuDNN cell. CuDNNLSTM and CuDNNGRU layers are much faster than LSTM and GRU layer, but they must be used on GPU. If you want to train on GPU and inferencing on CPU, you cannot use CuDNN cells.

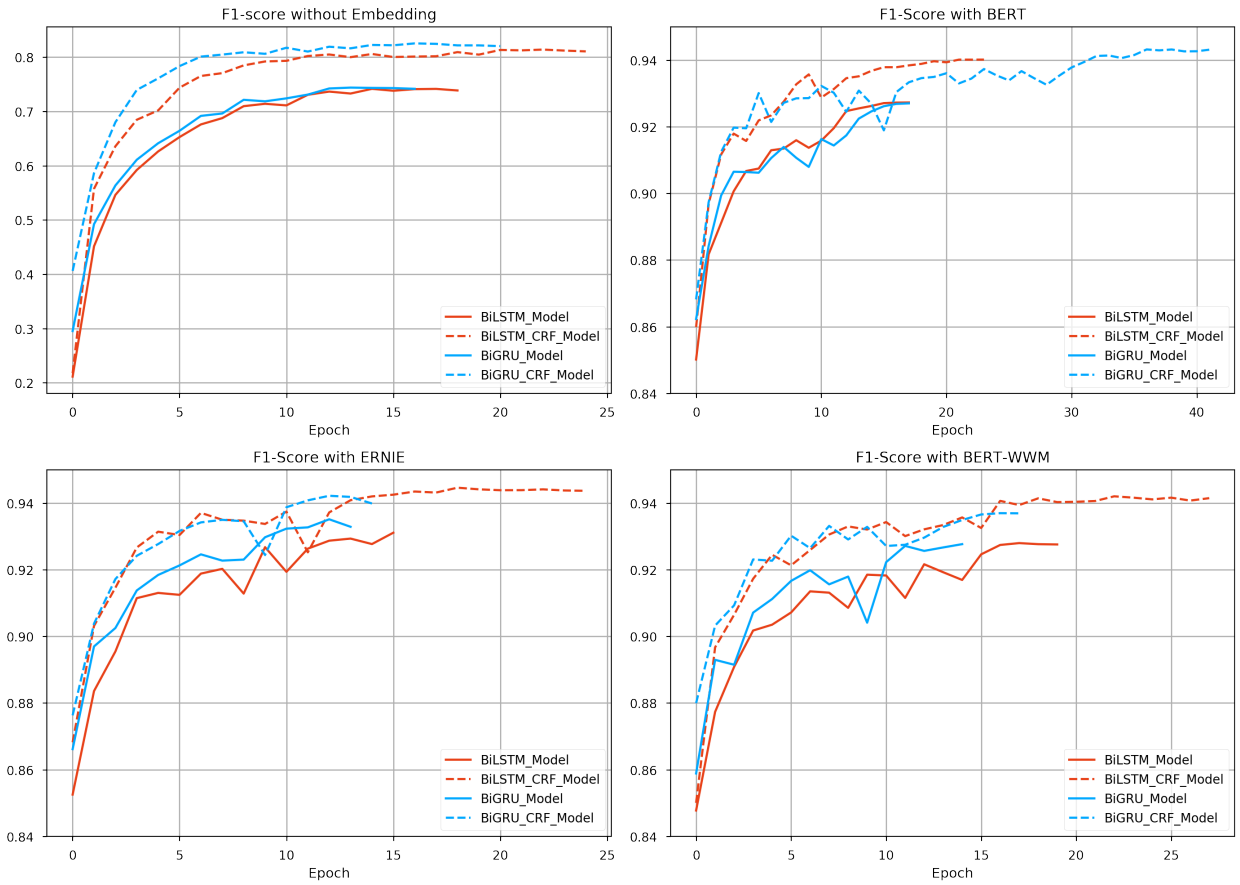
```
# Enable use cudnn cell
kashgari.config.use_cudnn_cell = True
```

Performance report

Available model list, matrics based on this training:

- corpus: ChineseDailyNerCorpus
- epochs: 50 epochs with callbacks
- batch_size: 64
- T4 GPU / 2 CPU / 30 GB on openbayes
- colab link

```
early_stop = keras.callbacks.EarlyStopping(patience=10)
reduse_lr_callback = keras.callbacks.ReduceLROnPlateau(factor=0.1, patience=5)
```



Text Scoring Model

Kashgari provides several models for text scoring, which could be use for Sentiment analysis tasks. Model input is text and output is continuous float value. All labeling models inherit from the `BaseScoringModel`. You could easily switch from one model to another just by changing one line of code.

Available Models

Train basic scoring model

```
# Load build-in corpus.
from kashgari.corpus import SMP2018ECDTCorpus

# Sample x is tokenized text, y is float value
train_x = [['Hello', 'world'], ['Hello', 'Kashgari'], ['I', 'hate', 'you']]
train_y = [5.0, 5.0, 1.2]

valid_x, valid_y = train_x, train_y
test_x, test_x = train_x, train_y
```

Then train our first model. All models provided some APIs, so you could use any scoring model here.

```
import kashgari
from kashgari.tasks.scoring import BiLSTM_Model

import logging
logging.basicConfig(level='DEBUG')

model = BiLSTM_Model()
model.fit(train_x, train_y, valid_x, valid_y)

# Evaluate the model
model.evaluate(test_x, test_y)

# Evaluate the model with round funcion
model.evaluate(test_x, test_y, should_round=True)

# Model data will save to `saved_scoring_model` folder
model.save('saved_scoring_model')

# Load saved model
loaded_model = kashgari.utils.load_model('saved_scoring_model')
loaded_model.predict(test_x[:10])

# To continue training, compile the newly loaded model first
loaded_model.compile_model()
model.fit(train_x, train_y, valid_x, valid_y)
```

That's all your need to do. Easy right.

Text scoring with transfer learning

Kashgari provides varies Language model Embeddings for transfer learning. Here is the example for BERT Embedding.

```
import kashgari
from kashgari.tasks.scoring import BiGRU_Model
from kashgari.embeddings import BERTEmbedding

import logging
logging.basicConfig(level='DEBUG')

bert_embed = BERTEmbedding('<PRE_TRAINED_BERT_MODEL_FOLDER>',
                           task=kashgari.SCORING,
                           sequence_length=100)
model = BiGRU_Model(bert_embed)
model.fit(train_x, train_y, valid_x, valid_y)
```

You could replace bert_embedding with any Embedding class in `kashgari.embeddings`. More info about Embedding: LINK THIS.

Adjust model's hyper-parameters

You could easily change model's hyper-parameters. For example, we change the lstm unit in `BiLSTM_Model` from 128 to 32.

```
from kashgari.tasks.scoring import BiLSTM_Model

hyper = BiLSTM_Model.get_default_hyper_parameters()
print(hyper)
# {'layer_bi_lstm': {'units': 128, 'return_sequences': False}, 'layer_dense': {
↪'activation': 'softmax'}}

hyper['layer_bi_lstm']['units'] = 32

model = BiLSTM_Model(hyper_parameters=hyper)
```

Use custom optimizer

Kashgari already supports using customized optimizer, like RAdam.

```
from kashgari.corpus import SMP2018ECDTCorpus
from kashgari.tasks.scoring import BiLSTM_Model
# Remember to import kashgari before than RAdam
from keras_radam import RAdam

model = BiLSTM_Model()
# This step will build token dict, label dict and model structure
model.build_model(train_x, train_y, valid_x, valid_y)
# Compile model with custom optimizer, you can also customize loss and metrics.
optimizer = RAdam()
model.compile_model(optimizer=optimizer)

# Train model
model.fit(train_x, train_y, valid_x, valid_y)
```

Use callbacks

Kashgari is based on keras so that you could use all of the tf.keras callbacks directly with Kashgari model. For example, here is how to visualize training with tensorboard.

```
from tensorflow.python import keras
from kashgari.tasks.scoring import BiGRU_Model
from kashgari.callbacks import EvalCallBack

import logging
logging.basicConfig(level='DEBUG')

model = BiGRU_Model()

tf_board_callback = keras.callbacks.TensorBoard(log_dir='./logs', update_freq=1000)

model.fit(train_x,
          train_y,
          valid_x,
          valid_y,
          batch_size=100,
          callbacks=[tf_board_callback])
```

Customize your own model

It is very easy and straightforward to build your own customized model, just inherit the `BaseScoringModel` and implement the `get_default_hyper_parameters()` function and `build_model_arc()` function.

```
from typing import Dict, Any

from tensorflow import keras

from kashgari.tasks.scoring.base_model import BaseScoringModel
from kashgari.layers import L

import logging
logging.basicConfig(level='DEBUG')

class DoubleBiLSTMModel(BaseScoringModel):
    """Bidirectional LSTM Sequence Labeling Model"""

    @classmethod
    def get_default_hyper_parameters(cls) -> Dict[str, Dict[str, Any]]:
        """
        Get hyper parameters of model
        Returns:
            hyper parameters dict
        """
        return {
            'layer_blstm1': {
                'units': 128,
                'return_sequences': True
            },
            'layer_blstm2': {
                'units': 128,
                'return_sequences': False
            },
            'layer_dropout': {
                'rate': 0.4
            },
            'layer_time_distributed': {},
            'layer_activation': {
                'activation': 'linear'
            }
        }

    def build_model_arc(self):
        """
        build model architectural
        """
        output_dim = self.processor.output_dim
        config = self.hyper_parameters
        embed_model = self.embedding.embed_model

        # Define your layers
        layer_blstm1 = L.Bidirectional(L.LSTM(**config['layer_blstm1']),
                                       name='layer_blstm1')
        layer_blstm2 = L.Bidirectional(L.LSTM(**config['layer_blstm2']),
                                       name='layer_blstm2')
```

(continues on next page)

(continued from previous page)

```
        layer_dropout = L.Dropout(**config['layer_dropout'],
                                  name='layer_dropout')

        layer_time_distributed = L.TimeDistributed(L.Dense(output_dim,
                                                           **config['layer_time_
↪distributed']),
                                                   name='layer_time_distributed')
        layer_activation = L.Activation(**config['layer_activation'])

        # Define tensor flow
        tensor = layer_blstm1(embed_model.output)
        tensor = layer_blstm2(tensor)
        tensor = layer_dropout(tensor)
        tensor = layer_time_distributed(tensor)
        output_tensor = layer_activation(tensor)

        # Init model
        self.tf_model = keras.Model(embed_model.inputs, output_tensor)

model = DoubleBLSTMModel()
model.fit(train_x, train_y, valid_x, valid_y)
```

Speed up with CuDNN cell

You can speed up training and inferencing process using CuDNN cell. CuDNNLSTM and CuDNNGRU layers are much faster than LSTM and GRU layer, but they must be used on GPU. If you want to train on GPU and inferencing on CPU, you cannot use CuDNN cells.

```
# Enable use cudnn cell
kashgari.config.use_cudnn_cell = True
```

Language Embeddings

Kashgari provides several embeddings for language representation. Embedding layers will convert input sequence to tensor for downstream task. Availabel embeddings list:

All embedding classes inherit from the `Embedding` class and implement the `embed()` to embed your input sequence and `embed_model` property which you need to build you own Model. By providing the `embed()` function and `embed_model` property, Kashgari hides the the complexity of different language embedding from users, all you need to care is which language embedding you need.

You could check out the Embedding API here: *link*

Quick start

Feature Extract From Pre-trained Embedding

Feature Extraction is one of the major way to use pre-trained language embedding. Kashgari provides simple API for this task. All you need to is init a embedding object then call `embed` function. Here is the example. All embedding shares same embed API.

```
import kashgari
from kashgari.embeddings import BERTEmbedding

# need to spesify task for the downstream task,
# if use embedding for feature extraction, just set `task=kashgari.CLASSIFICATION`
bert = BERTEmbedding('<BERT_MODEL_FOLDER>',
                     task=kashgari.CLASSIFICATION,
                     sequence_length=100)
# call for bulk embed
embed_tensor = bert.embed([['', '', '', '']])

# call for single embed
embed_tensor = bert.embed_one(['', '', '', ''])

print(embed_tensor)
# array([[-0.5001117 ,  0.9344998 , -0.55165815, ...,  0.49122602,
#         -0.2049343 ,  0.25752577],
#        [-1.05762   , -0.43353617,  0.54398274, ..., -0.61096823,
#          0.04312163,  0.03881482],
#        [ 0.14332692, -0.42566583,  0.68867105, ...,  0.42449307,
#          0.41105768,  0.08222893],
#        ...,
#        [-0.86124015,  0.08591427, -0.34404194, ...,  0.19915134,
#         -0.34176797,  0.06111742],
#        [-0.73940575, -0.02692179, -0.5826528 , ...,  0.26934686,
#         -0.29708537,  0.01855129],
#        [-0.85489404,  0.007399  , -0.26482674, ...,  0.16851354,
#         -0.36805922, -0.0052386 ]], dtype=float32)
```

Classification and Labeling

See details at classification and labeling tutorial.

Customized model

You can access the tf.keras model of embedding and add your own layers or any kind customizion. Just need to access the `embed_model` property of the embedding object.

Bare Embedding

```
kashgari.embeddings.BareEmbedding(task: str = None,
                                  sequence_length: Union[int, str] = 'auto',
                                  embedding_size: int = 100,
                                  processor: Optional[BaseProcessor] = None)
```

BareEmbedding is a random init `tf.keras.layers.Embedding` layer for text sequence embedding, which is the defualt embedding class for kashgari models.

Arguments

- **task**: `kashgari.CLASSIFICATION` `kashgari.LABELING`. Downstream task type, If you only need to feature extraction, just set it as `kashgari.CLASSIFICATION`.

- **sequence_length**: `'auto'`, `'variable'` or integer. When using `'auto'`, use the 95% of corpus length as sequence length. When using `'variable'`, model input shape will set to None, which can handle various length of input, it will use the length of max sequence in every batch for sequence length. If using an integer, let's say `50`, the input output sequence length will set to 50.
- **embedding_size**: Dimension of the dense embedding.

Here is the sample how to use embedding class. The key difference here is that must call `analyze_corpus` function before using the embed function. This is because the embedding layer is not pre-trained and do not contain any word-list. We need to build word-list from the corpus.

```
import kashgari
from kashgari.embeddings import BareEmbedding

embedding = BareEmbedding(task=kashgari.CLASSIFICATION,
                          sequence_length=100,
                          embedding_size=100)

embedding.analyze_corpus(x_data, y_data)

embed_tensor = embedding.embed_one(['', '', '', ''])
```

Word Embedding

```
kashgari.embeddings.WordEmbedding(w2v_path: str,
                                  task: str = None,
                                  w2v_kwargs: Dict[str, Any] = None,
                                  sequence_length: Union[Tuple[int, ...], str, int] =
↪'auto',
                                  processor: Optional[BaseProcessor] = None)
```

WordEmbedding is a `tf.keras.layers.Embedding` layer with pre-trained Word2Vec/GloVe Emedding weights.

When using pre-trained embedding, remember to use same tokenize tool with the embedding model, this will allow to access the full power of the embedding

Arguments

- **w2v_path**: Word2Vec file path.
- **task**: `kashgari.CLASSIFICATION` `kashgari.LABELING`. Downstream task type, If you only need to feature extraction, just set it as `kashgari.CLASSIFICATION`.
- **w2v_kwargs**: params pass to the `load_word2vec_format()` function of `gensim.models.KeyedVectors` - https://radimrehurek.com/gensim/models/keyedvectors.html#module-gensim.models.keyedvectors
- **sequence_length**: `'auto'`, `'variable'` or integer. When using `'auto'`, use the 95% of corpus length as sequence length. When using `'variable'`, model input shape will set to None, which can handle various length of input, it will use the length of max sequence in every batch for sequence length. If using an integer, let's say `50`, the input output sequence length will set to 50.

BERT Embedding

BERTEmbedding is based on keras-bert. The embeddings itself are wrapped into our simple embedding interface so that they can be used like any other embedding.

BERTEmbedding support BERT variants like **ERNIE**, but need to load the **tensorflow checkpoint**. If you intrested to use ERNIE, just download tensorflow_ernie and load like BERT Embedding.

!!! tip When using pre-trained embedding, remember to use same tokenize tool with the embedding model, this will allow to access the full power of the embedding

```
kashgari.embeddings.BERTEmbedding(model_folder: str,
                                  layer_nums: int = 4,
                                  trainable: bool = False,
                                  task: str = None,
                                  sequence_length: Union[str, int] = 'auto',
                                  processor: Optional[BaseProcessor] = None)
```

Arguments

- **model_folder**: path of checkpoint folder.
- **layer_nums**: number of layers whose outputs will be concatenated into a single tensor, default `4`, output the last 4 hidden layers as the thesis suggested.
- **trainable**: whether if the model is trainable, default `False` and set it to `True` for fine-tune this embedding layer during your training.
- **task**: `kashgari.CLASSIFICATION` `kashgari.LABELING`. Downstream task type, If you only need to feature extraction, just set it as `kashgari.CLASSIFICATION`.
- **sequence_length**: `'auto'`, `'variable'` or integer. When using `'auto'`, use the 95% of corpus length as sequence length. When using `'variable'`, model input shape will set to None, which can handle various length of input, it will use the length of max sequence in every batch for sequence length. If using an integer, let's say `50`, the input output sequence length will set to 50.

Example Usage - Text Classification

Let's run a text classification model with BERT.

```
sentences = [
    "Jim Henson was a puppeteer.",
    "This here's an example of using the BERT tokenizer.",
    "Why did the chicken cross the road?"
           ]
labels = [
    "class1",
    "class2",
    "class1"
]
########## Load Bert Embedding #########
import kashgari
from kashgari.embeddings import BERTEmbedding

bert_embedding = BERTEmbedding(bert_model_path,
                               task=kashgari.CLASSIFICATION,
                               sequence_length=128)

tokenizer = bert_embedding.tokenizer
sentences_tokenized = []
for sentence in sentences:
    sentence_tokenized = tokenizer.tokenize(sentence)
    sentences_tokenized.append(sentence_tokenized)
```

(continues on next page)

(continued from previous page)

```
"""
The sentences will become tokenized into:
[
    ['[CLS]', 'jim', 'henson', 'was', 'a', 'puppet', '##eer', '.', '[SEP]'],
    ['[CLS]', 'this', 'here', "'", 's', 'an', 'example', 'of', 'using', 'the', 'bert',
↪ 'token', '##izer', '.', '[SEP]'],
    ['[CLS]', 'why', 'did', 'the', 'chicken', 'cross', 'the', 'road', '?', '[SEP]']
]
"""

# Our tokenizer already added the BOS([CLS]) and EOS([SEP]) token
# so we need to disable the default add_bos_eos setting.
bert_embedding.processor.add_bos_eos = False

train_x, train_y = sentences_tokenized[:2], labels[:2]
validate_x, validate_y = sentences_tokenized[2:], labels[2:]

########## build model ##########
from kashgari.tasks.classification import CNNLSTMModel
model = CNNLSTMModel(bert_embedding)

########## /build model ##########
model.fit(
    train_x, train_y,
    validate_x, validate_y,
    epochs=3,
    batch_size=32
)
# save model
model.save('path/to/save/model/to')
```

Use sentence pairs for input

let's assume input pair sample is `"First do it" "then do it right"`, Then first tokenize the sentences using bert tokenizer. Then

```
sentence1 = ['First', 'do', 'it']
sentence2 = ['then', 'do', 'it', 'right']

sample = sentence1 + ["[SEP]"] + sentence2
# Add a special separation token `[SEP]` between two sentences tokens
# Generate a new token list
# ['First', 'do', 'it', '[SEP]', 'then', 'do', 'it', 'right']

train_x = [sample]
```

Pre-trained models

BERT Embedding V2

BERTEmbeddingV2 is based on bert4keras. The embeddings itself are wrapped into our simple embedding interface so that they can be used like any other embedding.

BERTEmbeddingV2 support models:

!!! tip When using pre-trained embedding, remember to use same tokenize tool with the embedding model, this will allow to access the full power of the embedding

```
kashgari.embeddings.BERTEmbedding(vacab_path: str,
                                  config_path: str,
                                  checkpoint_path: str,
                                  bert_type: str = 'bert',
                                  task: str = None,
                                  sequence_length: Union[str, int] = 'auto',
                                  processor: Optional[BaseProcessor] = None,
                                  from_saved_model: bool = False):
```

Arguments

- **vacab_path**: path of model's `vacab.txt` file
- **config_path**: path of model's `model.json` file
- **checkpoint_path**: path of model's checkpoint file
- **bert_type**: `bert`, `albert`, `nezha`. Type of BERT model.
- **task**: `kashgari.CLASSIFICATION` `kashgari.LABELING`. Downstream task type, If you only need to feature extraction, just set it as `kashgari.CLASSIFICATION`.
- **sequence_length**: `'auto'` or integer. When using `'auto'`, use the 95% of corpus length as sequence length. If using an integer, let's say `50`, the input output sequence length will set to 50.

Example Usage - Text Classification

Let's run a text classification model with BERT.

```
sentences = [
    "Jim Henson was a puppeteer.",
    "This here's an example of using the BERT tokenizer.",
    "Why did the chicken cross the road?"
            ]
labels = [
    "class1",
    "class2",
    "class1"
]
# ------------ Load Bert Embedding ------------
import os
import kashgari
from kashgari.embeddings.bert_embedding_v2 import BERTEmbeddingV2
from kashgari.tokenizer import BertTokenizer

# Setup paths
model_folder = '/Users/brikerman/Desktop/nlp/language_models/albert_base'
checkpoint_path = os.path.join(model_folder, 'model.ckpt-best')
config_path = os.path.join(model_folder, 'albert_config.json')
vacab_path = os.path.join(model_folder, 'vocab_chinese.txt')

tokenizer = BertTokenizer.load_from_vacab_file(vacab_path)
embed = BERTEmbeddingV2(vacab_path, config_path, checkpoint_path,
                        bert_type='albert',
                        task=kashgari.CLASSIFICATION,
```

(continues on next page)

(continued from previous page)

```
                        sequence_length=100)

sentences_tokenized = [tokenizer.tokenize(s) for s in sentences]
"""
The sentences will become tokenized into:
[
    ['jim', 'henson', 'was', 'a', 'puppet', '##eer', '.'],
    ['this', 'here', "'", 's', 'an', 'example', 'of', 'using', 'the', 'bert', 'token',
↪ '##izer', '.'],
    ['why', 'did', 'the', 'chicken', 'cross', 'the', 'road', '?']
]
"""

train_x, train_y = sentences_tokenized[:2], labels[:2]
validate_x, validate_y = sentences_tokenized[2:], labels[2:]

# ------------ Build Model Start ------------
from kashgari.tasks.classification import CNNLSTMModel
model = CNNLSTMModel(bert_embedding)

# ------------ Build Model End ------------

model.fit(
    train_x, train_y,
    validate_x, validate_y,
    epochs=3,
    batch_size=32
)
# save model
model.save('path/to/save/model/to')
```

GPT2 Embedding

GPT2Embedding is based on keras-gpt-2. The embeddings itself are wrapped into our simple embedding interface so that they can be used like any other embedding.

!!! tip When using pre-trained embedding, remember to use same tokenize tool with the embedding model, this will allow to access the full power of the embedding

```
kashgari.embeddings.GPT2Embedding(model_folder: str,
                                  task: str = None,
                                  sequence_length: Union[str, int] = 'auto',
                                  processor: Optional[BaseProcessor] = None)
```

Arguments

- **model_folder**: path of checkpoint folder.
- **task**: `kashgari.CLASSIFICATION` `kashgari.LABELING`. Downstream task type, If you only need to feature extraction, just set it as `kashgari.CLASSIFICATION`.
- **sequence_length**: `'auto'`, `'variable'` or integer. When using `'auto'`, use the 95% of corpus length as sequence length. When using `'variable'`, model input shape will set to None, which can handle various length of input, it will use the length of max sequence in every batch for sequence length. If using an integer, let's say `50`, the input output sequence length will set to 50.

Numeric Features Embedding

NumericFeaturesEmbedding is a random init `tf.keras.layers.Embedding` layer for numeric feature embedding. Which usally comes togather with StackedEmbedding for represent non text features.

More details checkout the example: *Handle Numeric features*

```
kashgari.embeddings.NumericFeaturesEmbedding(feature_count: int,
                                             feature_name: str,
                                             sequence_length: Union[str, int] = 'auto
↪',
                                             embedding_size: int = None,
                                             processor: Optional[BaseProcessor] =␣
↪None)
```

Arguments

- **feature_count**: count of the features of this embedding.
- **feature_name**: name of the feature.
- **sequence_length**: `'auto'`, `'variable'` or integer. When using `'auto'`, use the 95% of corpus length as sequence length. When using `'variable'`, model input shape will set to None, which can handle various length of input, it will use the length of max sequence in every batch for sequence length. If using an integer, let's say `50`, the input output sequence length will set to 50.
- **embedding_size**: Dimension of the dense embedding.

Stacked Embedding

StackedEmbedding is a special kind of embedding class, which will able to stack other embedding layers togather for multi-input models.

More details checkout the example: *Handle Numeric features*

```
kashgari.embeddings.StackedEmbedding(embeddings: List[Embedding],
                                     processor: Optional[BaseProcessor] = None)
```

Arguments

- **embeddings**: list of embedding object.

Customize Multi Output Model

It is very easy to customize your own multi output model. Lets assume you have dataset like this, One input and two output.

Example code at file `tests/test_custom_multi_output_classification.py`.

```
x = [
    ['', '', '', '', '', ''],
    ['', '', '', 'q', 'q', ''],
    ['', '', '', '', '', ''],
    ['', '', '', '', '', '', '', '', '', '', '', '', '', '', '', '', ''],
    ['', '', '', '', '']
]

output_1 = [
```

(continues on next page)

(continued from previous page)

```
    [0. 0. 1.]
    [1. 0. 0.]
    [1. 0. 0.]
    [0. 0. 1.]
    [1. 0. 0.]]

output_2 = [
    [0. 1. 0.]
    [0. 0. 1.]
    [0. 0. 1.]
    [1. 0. 0.]
    [0. 0. 1.]]
```

Then you need to create a customized processor inherited from the `ClassificationProcessor`.

```
import kashgari
import numpy as np
from typing import Tuple, List, Optional, Dict, Any
from kashgari.processors.classification_processor import ClassificationProcessor

class MultiOutputProcessor(ClassificationProcessor):
    def process_y_dataset(self,
                          data: Tuple[List[List[str]], ...],
                          maxlens: Optional[Tuple[int, ...]] = None,
                          subset: Optional[List[int]] = None) -> Tuple[np.ndarray, ...
↪]:
        # Data already converted to one-hot
        # Only need to get the subset
        result = []
        for index, dataset in enumerate(data):
            if subset is not None:
                target = kashgari.utils.get_list_subset(dataset, subset)
            else:
                target = dataset
            result.append(np.array(target))

        if len(result) == 1:
            return result[0]
        else:
            return tuple(result)
```

Then build your own model inherited from the `BaseClassificationModel`

```
import kashgari
import tensorflow as tf
from typing import Tuple, List, Optional, Dict, Any
from kashgari.layers import L
from kashgari.tasks.classification.base_model import BaseClassificationModel

class MultiOutputModel(BaseClassificationModel):
    @classmethod
    def get_default_hyper_parameters(cls) -> Dict[str, Dict[str, Any]]:
        return {
            'layer_bi_lstm': {
                'units': 256,
                'return_sequences': False
            }
```

(continues on next page)

(continued from previous page)

```
        }

    # Build your own model
    def build_model_arc(self):
        config = self.hyper_parameters
        embed_model = self.embedding.embed_model

        layer_bi_lstm = L.Bidirectional(L.LSTM(**config['layer_bi_lstm']), name=
↪'layer_bi_lstm')
        layer_output_1 = L.Dense(3, activation='sigmoid', name='layer_output_1')
        layer_output_2 = L.Dense(3, activation='sigmoid', name='layer_output_2')

        tensor = layer_bi_lstm(embed_model.output)
        output_tensor_1 = layer_output_1(tensor)
        output_tensor_2 = layer_output_2(tensor)

        self.tf_model = tf.keras.Model(embed_model.inputs, [output_tensor_1, output_
↪tensor_2])

    # Rewrite your predict function
    def predict(self,
                x_data,
                batch_size=None,
                debug_info=False,
                threshold=0.5):
        tensor = self.embedding.process_x_dataset(x_data)
        pred = self.tf_model.predict(tensor, batch_size=batch_size)

        output_1 = pred[0]
        output_2 = pred[1]

        output_1[output_1 >= threshold] = 1
        output_1[output_1 < threshold] = 0
        output_2[output_2 >= threshold] = 1
        output_2[output_2 < threshold] = 0

        return output_1, output_2
```

Tada, all done, Now build your own model with customized processor

```
from kashgari.embeddings import BareEmbedding

# Use your processor to init embedding, You can use any embedding layer provided by␣
↪kashgari here

processor = MultiOutputProcessor()
embedding = BareEmbedding(processor=processor)

m = MultiOutputModel(embedding=embedding)
m.build_model(train_x, (output_1, output_2))
m.fit(train_x, (output_1, output_2))
```

Handle Numeric features

This feature is a experimental feature

https://github.com/BrikerMan/Kashgari/issues/90

At times there have some additional features like text formatting (italic, bold, centered), position in text and more. Kashgari provides `NumericFeaturesEmbedding` and `StackedEmbedding` for this kind data. Here is the details:

If you have a dataset like this.

```
token=NLP       start_of_p=True    bold=True     center=True     B-Category
token=Projects  start_of_p=False   bold=True     center=True     I-Category
token=Project   start_of_p=True    bold=True     center=False    B-Project-name
token=Name      start_of_p=False   bold=True     center=False    I-Project-name
token=:         start_of_p=False   bold=False    center=False    I-Project-name
```

First, numerize your additional features. Convert your data to this. Remember to leave `0` for padding.

```
text = ['NLP', 'Projects', 'Project', 'Name', ':']
start_of_p = [1, 2, 1, 2, 2]
bold = [1, 1, 1, 1, 2]
center = [1, 1, 2, 2, 2]
label = ['B-Category', 'I-Category', 'B-Project-name', 'I-Project-name', 'I-Project-
↪name']
```

Then you have four input sequence and one output sequence. Prepare your embedding layers.

```
import kashgari
from kashgari.embeddings import NumericFeaturesEmbedding, BareEmbedding,␣
↪StackedEmbedding

import logging
logging.basicConfig(level='DEBUG')

text = ['NLP', 'Projects', 'Project', 'Name', ':']
start_of_p = [1, 2, 1, 2, 2]
bold = [1, 1, 1, 1, 2]
center = [1, 1, 2, 2, 2]
label = ['B-Category', 'I-Category', 'B-ProjectName', 'I-ProjectName', 'I-ProjectName
↪']

text_list = [text] * 100
start_of_p_list = [start_of_p] * 100
bold_list = [bold] * 100
center_list = [center] * 100
label_list = [label] * 100

SEQUENCE_LEN = 100

# You can use Word Embedding or BERT Embedding for your text embedding
text_embedding = BareEmbedding(task=kashgari.LABELING, sequence_length=SEQUENCE_LEN)
start_of_p_embedding = NumericFeaturesEmbedding(feature_count=2,
                                                feature_name='start_of_p',
                                                sequence_length=SEQUENCE_LEN)

bold_embedding = NumericFeaturesEmbedding(feature_count=2,
                                          feature_name='bold',
                                          sequence_length=SEQUENCE_LEN)

center_embedding = NumericFeaturesEmbedding(feature_count=2,
                                            feature_name='center',
```

(continues on next page)

(continued from previous page)

```
                                                    sequence_length=SEQUENCE_LEN)

# first attribute, must be the text embedding
stack_embedding = StackedEmbedding([
    text_embedding,
    start_of_p_embedding,
    bold_embedding,
    center_embedding
])

x = (text_list, start_of_p_list, bold_list, center_list)
y = label_list
stack_embedding.analyze_corpus(x, y)

# Now we can embed using this stacked embedding layer
print(stack_embedding.embed(x))
```

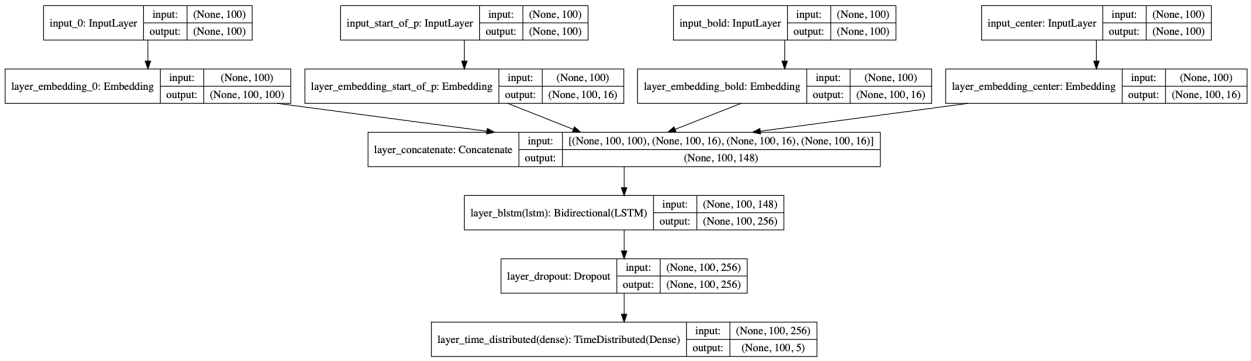
Once the embedding layer prepared, you can use all of the classification and labeling models.

```
# We can build any labeling model with this embedding

from kashgari.tasks.labeling import BLSTMModel
model = BLSTMModel(embedding=stack_embedding)
model.fit(x, y)

print(model.predict(x))
print(model.predict_entities(x))
```

This is the structurer of this model.



Tensorflow Serving

```
from kashgari.tasks.classification import BiGRU_Model
from kashgari.corpus import SMP2018ECDTCorpus
from kashgari import utils

train_x, train_y = SMP2018ECDTCorpus.load_data()

model = BiGRU_Model()
model.fit(train_x, train_y)

# Save model
```

(continues on next page)

(continued from previous page)

```
utils.convert_to_saved_model(model,
                             model_path='saved_model/bgru',
                             version=1)
```

Then run tensorflow-serving.

```
docker run -t --rm -p 8501:8501 -v "path_to/saved_model:/models/" -e MODEL_NAME=bgru␣
↪tensorflow/serving
```

Load processor from model, then predict with serving.

```
import requests
from kashgari import utils
import numpy as np

x = ['Hello', 'World']
# Pre-processor data
processor = utils.load_processor(model_path='saved_model/bgru/1')
tensor = processor.process_x_dataset([x])

# array([[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], dtype=int32)

# if you using BERT, you need to reformat tensor first
# ------ Only for BERT Embedding Start --------
tensor = [{
   "Input-Token:0": i.tolist(),
   "Input-Segment:0": np.zeros(i.shape).tolist()
} for i in tensor]
# ------ Only for BERT Embedding End ----------

# predict
r = requests.post("http://localhost:8501/v1/models/bgru:predict", json={"instances":␣
↪tensor.tolist()})
preds = r.json()['predictions']

# Convert result back to labels
labels = processor.reverse_numerize_label_sequences(np.array(preds).argmax(-1))

# labels = ['video']
```

corpus

Kashgari provides several build-in corpus for testing.

Chinese Daily Ner Corpus

Chinese Ner corpus cotains 20864 train samples, 4636 test samples and 2318 valid samples.

Usage:

```
from kashgari.corpus import ChineseDailyNerCorpus

train_x, train_y = ChineseDailyNerCorpus.load_data('train')
```

(continues on next page)

(continued from previous page)

```
test_x, test_y = ChineseDailyNerCorpus.load_data('test')
valid_x, valid_y = ChineseDailyNerCorpus.load_data('valid')
```

Data Sample:

```
>>> x[0]
['', '', '', '', '', '', '', '', '']
>>> y[0]
['O', 'O', 'O', 'O', 'O', 'O', 'O', 'B-LOC', 'I-LOC']
```

SMP2018 ECDT Human-Computer Dialogue Classification Corpus

https://worksheets.codalab.org/worksheets/0x27203f932f8341b79841d50ce0fd684f/

This dataset is released by the Evaluation of Chinese Human-Computer Dialogue Technology (SMP2018-ECDT) task 1 and is provided by the iFLYTEK Corporation, which is a Chinese human-computer dialogue dataset.

```
      label           query
0   weather
1       map
2  cookbook
3    health
4      chat
```

Usage:

```
from kashgari.corpus import SMP2018ECDTCorpus

train_x, train_y = SMP2018ECDTCorpus.load_data('train')
test_x, test_y = SMP2018ECDTCorpus.load_data('test')
valid_x, valid_y = SMP2018ECDTCorpus.load_data('valid')

# Change cutter to jieba, need to install jieba first
train_x, train_y = SMP2018ECDTCorpus.load_data('train', cutter='jieba')
test_x, test_y = SMP2018ECDTCorpus.load_data('test', cutter='jieba')
valid_x, valid_y = SMP2018ECDTCorpus.load_data('valid', cutter='jieba')
```

Data Sample:

```
# char cutted
>>> x[0]
[['', '', '', '', '', '']]
>>> y[0]
['message']

# jieba cutted
>>> x[0]
[['', '', '']]
>>> y[0]
['message']
```

embeddings

__init__

Embedding layers have its own __init__ function, check it out from their document page.

All embedding layer shares same API except the `__init__` function.

Properties

token_count

int, corpus token count

sequence_length

int, model sequence length

label2idx

dict, label to index dict

token_count

int, corpus token count

tokenizer

Built-in Tokenizer of Embedding layer, available in `BERTEmbedding`.

Methods

analyze_corpus

Analyze data, build the token dict and label dict

```
def analyze_corpus(self,
                   x: List[List[str]],
                   y: Union[List[List[str]], List[str]]):
```

Args:

- **x**: Array of input data
- **y_train**: Array of label data

process_x_dataset

Batch process feature data to tensor, mostly call processor's `process_x_dataset` function to handle the data.

```
def process_x_dataset(self,
                      data: List[List[str]],
                      subset: Optional[List[int]] = None) -> np.ndarray:
```

Args:

- **data**: target dataset
- **subset**: subset index list

Returns:

- vectorized feature tensor

process_y_dataset

Batch process labels data to tensor, mostly call processor's `process_y_dataset` function to handle the data.

```
def process_y_dataset(self,
                      data: List[List[str]],
                      subset: Optional[List[int]] = None) -> np.ndarray:
```

Args:

- **data**: target dataset
- **subset**: subset index list

Returns:

- vectorized label tensor

reverse_numerize_label_sequences

```
def reverse_numerize_label_sequences(self,
                                     sequences,
                                     lengths=None):
```

embed

Batch embed sentences, use this function for feature extraction. Input text then get the tensor representation.

```
def embed(self,
          sentence_list: Union[List[List[str]], List[List[int]]],
          debug: bool = False) -> np.ndarray:
```

Args:

- **sentence_list**: Sentence list to embed
- **debug**: Show debug info, defualt False

Returns:

- A list of numpy arrays representing the embeddings

embed_one

Dummy function for embed single sentence.

Args:

- **sentence**: Target sentence, list of tokens

Returns:

- Numpy arrays representing the embeddings

info

Returns a dictionary containing the configuration of the model.

```
def info(self) -> Dict:
```

tasks.classification

All Text classification models share the same API.

__init__

```
def __init__(self,
             embedding: Optional[Embedding] = None,
             hyper_parameters: Optional[Dict[str, Dict[str, Any]]] = None)
```

Args:

- **embedding**: model embedding
- **hyper_parameters**: a dict of hyper_parameters.

You could change customize hyper_parameters like this:

```
# get default hyper_parameters
hyper_parameters = BiLSTM_Model.get_default_hyper_parameters()
# change lstm hidden unit to 12
hyper_parameters['layer_blstm']['units'] = 12
# init new model with customized hyper_parameters
labeling_model = BiLSTM_Model(hyper_parameters=hyper_parameters)
labeling_model.fit(x, y)
```

Properties

token2idx

Returns model's token index map, type: `Dict[str, int]`

label2idx

Returns model's label index map, type: `Dict[str, int]`

Methods

get_default_hyper_parameters

Return the defualt hyper parameters

!!! attention "You must implement this function when customizing a model" When you are customizing your own model, you must implement this function.

```
Customization example: [customize-your-own-mode](../tutorial/text-classification.md
↪#customize-your-own-model)
```

```
@classmethod
def get_default_hyper_parameters(cls) -> Dict[str, Dict[str, Any]]:
```

Returns:

- dict of the defualt hyper parameters

build_model_arc

build model architectural, define models structure in this function.

!!! attention "You must implement this function when customizing a model" When you are customizing your own model, you must implement this function.

```
Customization example: [customize-your-own-mode](../tutorial/text-classification.md
↪#customize-your-own-model)
```

```
def build_model_arc(self):
```

build_model

build model with corpus

```
def build_model(self,
                x_train: Union[Tuple[List[List[str]], ...], List[List[str]]],
                y_train: Union[List[List[str]], List[str]],
                x_validate: Union[Tuple[List[List[str]], ...], List[List[str]]] =␣
↪None,
                y_validate: Union[List[List[str]], List[str]] = None)
```

Args:

- **x_train**: Array of train feature data (if the model has a single input), or tuple of train feature data array (if the model has multiple inputs)
- **y_train**: Array of train label data

- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data

build_multi_gpu_model

Build multi-GPU model with corpus

```
def build_multi_gpu_model(self,
                          gpus: int,
                          x_train: Union[Tuple[List[List[str]], ...],␣
↪List[List[str]]],
                          y_train: Union[List[List[str]], List[str]],
                          cpu_merge: bool = True,
                          cpu_relocation: bool = False,
                          x_validate: Union[Tuple[List[List[str]], ...],␣
↪List[List[str]]] = None,
                          y_validate: Union[List[List[str]], List[str]] = None):
```

Args:

- **gpus**: Integer >= 2, number of on GPUs on which to create model replicas.
- **cpu_merge**: A boolean value to identify whether to force merging model weights under the scope of the CPU or not.
- **cpu_relocation**: A boolean value to identify whether to create the model's weights under the scope of the CPU. If the model is not defined under any preceding device scope, you can still rescue it by activating this option.
- **x_train**: Array of train feature data (if the model has a single input), or tuple of train feature data array (if the model has multiple inputs)
- **y_train**: Array of train label data
- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data

build_tpu_model

Build TPU model with corpus

```
def build_tpu_model(self, strategy: tf.contrib.distribute.TPUStrategy,
                    x_train: Union[Tuple[List[List[str]], ...], List[List[str]]],
                    y_train: Union[List[List[str]], List[str]],
                    x_validate: Union[Tuple[List[List[str]], ...], List[List[str]]] =␣
↪None,
                    y_validate: Union[List[List[str]], List[str]] = None):
```

Args:

- **strategy**: `TPUDistributionStrategy`. The strategy to use for replicating model across multiple TPU cores.
- **x_train**: Array of train feature data (if the model has a single input), or tuple of train feature data array (if the model has multiple inputs)

- **y_train**: Array of train label data
- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data

compile_model

Configures the model for training.

Using `compile()` function of tf.keras.Model

```
def compile_model(self, **kwargs):
```

Args:

- ****kwargs**: arguments passed to `compile()` function of `tf.keras.Model`

Defaults:

- **loss**: `categorical_crossentropy`
- **optimizer**: `adam`
- **metrics**: `['accuracy']`

get_data_generator

data generator for fit_generator

```
def get_data_generator(self,
                       x_data,
                       y_data,
                       batch_size: int = 64,
                       shuffle: bool = True)
```

Args:

- **x_data**: Array of feature data (if the model has a single input), or tuple of feature data array (if the model has multiple inputs)
- **y_data**: Array of label data
- **batch_size**: Number of samples per gradient update, default to 64.
- **shuffle**:

Returns:

- data generator

fit

Trains the model for a given number of epochs with fit_generator (iterations on a dataset).

```
def fit(self,
        x_train: Union[Tuple[List[List[str]], ...], List[List[str]]],
        y_train: Union[List[List[str]], List[str]],
        x_validate: Union[Tuple[List[List[str]], ...], List[List[str]]] = None,
        y_validate: Union[List[List[str]], List[str]] = None,
        batch_size: int = 64,
        epochs: int = 5,
        callbacks: List[keras.callbacks.Callback] = None,
        fit_kwargs: Dict = None):
```

Args:

- **x_train**: Array of train feature data (if the model has a single input), or tuple of train feature data array (if the model has multiple inputs)
- **y_train**: Array of train label data
- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data
- **batch_size**: Number of samples per gradient update, default to 64.
- **epochs**: Integer. Number of epochs to train the model. default 5.
- **callbacks**:
- **fit_kwargs**: additional arguments passed to `fit_generator()` function from tensorflow.keras.Model

Returns:

- A `tf.keras.callbacks.History` object.

fit_without_generator

Trains the model for a given number of epochs (iterations on a dataset). Large memory Cost.

```
def fit_without_generator(self,
                          x_train: Union[Tuple[List[List[str]], ...],␣
↪List[List[str]]],
                          y_train: Union[List[List[str]], List[str]],
                          x_validate: Union[Tuple[List[List[str]], ...],␣
↪List[List[str]]] = None,
                          y_validate: Union[List[List[str]], List[str]] = None,
                          batch_size: int = 64,
                          epochs: int = 5,
                          callbacks: List[keras.callbacks.Callback] = None,
                          fit_kwargs: Dict = None):
```

Args:

- **x_train**: Array of train feature data (if the model has a single input), or tuple of train feature data array (if the model has multiple inputs)
- **y_train**: Array of train label data
- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data

- **batch_size**: Number of samples per gradient update, default to 64.
- **epochs**: Integer. Number of epochs to train the model. default 5.
- **callbacks**:
- **fit_kwargs**: additional arguments passed to `fit_generator()` function from tensorflow.keras.Model

Returns:

- A `tf.keras.callbacks.History` object.

predict

Generates output predictions for the input samples. Computation is done in batches.

```
def predict(self,
            x_data,
            batch_size=None,
            multi_label_threshold: float = 0.5,
            debug_info=False,
            predict_kwargs: Dict = None):
```

Args:

- **x_data**: The input data, as a Numpy array (or list of Numpy arrays if the model has multiple inputs).
- **batch_size**: Integer. If unspecified, it will default to 32.
- **multi_label_threshold**:
- **debug_info**: Bool, Should print out the logging info.
- **predict_kwargs**: Dict, arguments passed to `predict()` function of tensorflow.keras.Model

Returns:

- array of predictions.

predict_top_k_class

Generates output predictions with confidence for the input samples.

Computation is done in batches.

```
def predict_top_k_class(self,
                        x_data,
                        top_k=5,
                        batch_size=32,
                        debug_info=False,
                        predict_kwargs: Dict = None) -> List[Dict]:
```

Args:

- **x_data**: The input data, as a Numpy array (or list of Numpy arrays if the model has multiple inputs).
- **top_k**: int
- **batch_size**: Integer. If unspecified, it will default to 32.
- **debug_info**: Bool, Should print out the logging info.

- **predict_kwargs**: Dict, arguments passed to `predict()` function of tensorflow.keras.Model

Returns:

array(s) of prediction result dict.

- sample result of single-label classification:

```
[
  {
    "label": "chat",
    "confidence": 0.5801531,
    "candidates": [
      { "label": "cookbook", "confidence": 0.1886314 },
      { "label": "video", "confidence": 0.13805099 },
      { "label": "health", "confidence": 0.013852648 },
      { "label": "translation", "confidence": 0.012913573 }
    ]
  }
]
```

- sample result of multi-label classification:

```
[
  {
    "candidates": [
      { "confidence": 0.9959336, "label": "toxic" },
      { "confidence": 0.9358089, "label": "obscene" },
      { "confidence": 0.6882098, "label": "insult" },
      { "confidence": 0.13540423, "label": "severe_toxic" },
      { "confidence": 0.017219543, "label": "identity_hate" }
    ]
  }
]
```

evaluate

Evaluate model

```
def evaluate(self,
             x_data,
             y_data,
             batch_size=None,
             digits=4,
             debug_info=False) -> Tuple[float, float, Dict]:
```

Args:

- **x_data**:
- **y_data**:
- **batch_size**:
- **digits**:
- **debug_info**:

save

Save model info json and model weights to given folder path

```
def save(self, model_path: str):
```

Args:

- **model_path**: target model folder path

info

Returns a dictionary containing the configuration of the model.

```
def info(self)
```

tasks.labeling

All Text labeling models share the same API.

__init__

```
def __init__(self,
             embedding: Optional[Embedding] = None,
             hyper_parameters: Optional[Dict[str, Dict[str, Any]]] = None)
```

Args:

- **embedding**: model embedding
- **hyper_parameters**: a dict of hyper_parameters.

You could change customize hyper_parameters like this::

```
# get default hyper_parameters
hyper_parameters = BiLSTM_Model.get_default_hyper_parameters()
# change lstm hidden unit to 12
hyper_parameters['layer_blstm']['units'] = 12
# init new model with customized hyper_parameters
labeling_model = BiLSTM_Model(hyper_parameters=hyper_parameters)
labeling_model.fit(x, y)
```

Properties

token2idx

Returns model's token index map, type: `Dict[str, int]`

label2idx

Returns model's label index map, type: `Dict[str, int]`

Methods

get_default_hyper_parameters

Return the defualt hyper parameters

!!! attention "You must implement this function when customizing a model" When you are customizing your own model, you must implement this function.

```
Customization example: [customize-your-own-mode](../tutorial/text-classification.md
↪#customize-your-own-model)
```

```
@classmethod
def get_default_hyper_parameters(cls) -> Dict[str, Dict[str, Any]]:
```

Returns:

- dict of the defualt hyper parameters

build_model_arc

build model architectural, define models structure in this function.

!!! attention "You must implement this function when customizing a model" When you are customizing your own model, you must implement this function.

```
Customization example: [customize-your-own-mode](../tutorial/text-classification.md
↪#customize-your-own-model)
```

```
def build_model_arc(self):
```

build_model

build model with corpus

```
def build_model(self,
                x_train: Union[Tuple[List[List[str]], ...], List[List[str]]],
                y_train: Union[List[List[str]], List[str]],
                x_validate: Union[Tuple[List[List[str]], ...], List[List[str]]] =␣
↪None,
                y_validate: Union[List[List[str]], List[str]] = None)
```

Args:

- **x_train**: Array of training feature data (if the model has a single input), or tuple of training feature data array (if the model has multiple inputs)
- **y_train**: Array of training label data
- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data

build_multi_gpu_model

Build multi-GPU model with corpus

```
def build_multi_gpu_model(self,
                          gpus: int,
                          x_train: Union[Tuple[List[List[str]], ...],␣
↪List[List[str]]],
                          y_train: Union[List[List[str]], List[str]],
                          cpu_merge: bool = True,
                          cpu_relocation: bool = False,
                          x_validate: Union[Tuple[List[List[str]], ...],␣
↪List[List[str]]] = None,
                          y_validate: Union[List[List[str]], List[str]] = None):
```

Args:

- **gpus**: Integer >= 2, number of on GPUs on which to create model replicas.
- **cpu_merge**: A boolean value to identify whether to force merging model weights under the scope of the CPU or not.
- **cpu_relocation**: A boolean value to identify whether to create the model's weights under the scope of the CPU. If the model is not defined under any preceding device scope, you can still rescue it by activating this option.
- **x_train**: Array of training feature data (if the model has a single input), or tuple of training feature data array (if the model has multiple inputs)
- **y_train**: Array of training label data
- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data

build_tpu_model

Build TPU model with corpus

```
def build_tpu_model(self, strategy: tf.contrib.distribute.TPUStrategy,
                    x_train: Union[Tuple[List[List[str]], ...], List[List[str]]],
                    y_train: Union[List[List[str]], List[str]],
                    x_validate: Union[Tuple[List[List[str]], ...], List[List[str]]] =␣
↪None,
                    y_validate: Union[List[List[str]], List[str]] = None):
```

Args:

- **strategy**: `TPUDistributionStrategy`. The strategy to use for replicating model across multiple TPU cores.
- **x_train**: Array of training feature data (if the model has a single input), or tuple of training feature data array (if the model has multiple inputs)
- **y_train**: Array of training label data
- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data

compile_model

Configures the model for training.

Using `compile()` function of tf.keras.Model

```
def compile_model(self, **kwargs):
```

Args:

- ****kwargs**: arguments passed to `compile()` function of `tf.keras.Model`

Defaults:

- **loss**: `categorical_crossentropy`
- **optimizer**: `adam`
- **metrics**: `['accuracy']`

get_data_generator

data generator for fit_generator

```
def get_data_generator(self,
                       x_data,
                       y_data,
                       batch_size: int = 64,
                       shuffle: bool = True)
```

Args:

- **x_data**: Array of feature data (if the model has a single input), or tuple of feature data array (if the model has multiple inputs)
- **y_data**: Array of label data
- **batch_size**: Number of samples per gradient update, default to 64.
- **shuffle**:

Returns:

- data generator

fit

Trains the model for a given number of epochs with fit_generator (iterations on a dataset).

```
def fit(self,
        x_train: Union[Tuple[List[List[str]], ...], List[List[str]]],
        y_train: Union[List[List[str]], List[str]],
        x_validate: Union[Tuple[List[List[str]], ...], List[List[str]]] = None,
        y_validate: Union[List[List[str]], List[str]] = None,
        batch_size: int = 64,
        epochs: int = 5,
        callbacks: List[keras.callbacks.Callback] = None,
        fit_kwargs: Dict = None):
```

Args:

- **x_train**: Array of training feature data (if the model has a single input), or tuple of training feature data array (if the model has multiple inputs)
- **y_train**: Array of training label data
- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data
- **batch_size**: Number of samples per gradient update, default to 64.
- **epochs**: Integer. Number of epochs to train the model. default 5.
- **callbacks**:
- **fit_kwargs**: additional arguments passed to `fit_generator()` function from tensorflow.keras.Model

Returns:

- A `tf.keras.callbacks.History` object.

fit_without_generator

Trains the model for a given number of epochs (iterations on a dataset). Large memory Cost.

```
def fit_without_generator(self,
                          x_train: Union[Tuple[List[List[str]], ...],␣
↪List[List[str]]],
                          y_train: Union[List[List[str]], List[str]],
                          x_validate: Union[Tuple[List[List[str]], ...],␣
↪List[List[str]]] = None,
                          y_validate: Union[List[List[str]], List[str]] = None,
                          batch_size: int = 64,
                          epochs: int = 5,
                          callbacks: List[keras.callbacks.Callback] = None,
                          fit_kwargs: Dict = None):
```

Args:

- **x_train**: Array of training feature data (if the model has a single input), or tuple of training feature data array (if the model has multiple inputs)
- **y_train**: Array of training label data
- **x_validate**: Array of validation feature data (if the model has a single input), or tuple of validation feature data array (if the model has multiple inputs)
- **y_validate**: Array of validation label data
- **batch_size**: Number of samples per gradient update, default to 64.
- **epochs**: Integer. Number of epochs to train the model. default 5.
- **callbacks**:
- **fit_kwargs**: additional arguments passed to `fit_generator()` function from tensorflow.keras.Model

Returns:

- A `tf.keras.callbacks.History` object.

predict

Generates output predictions for the input samples. Computation is done in batches.

```
def predict(self,
            x_data,
            batch_size=32,
            debug_info=False,
            predict_kwargs: Dict = None):
```

Args:

- **x_data**: The input data, as a Numpy array (or list of Numpy arrays if the model has multiple inputs).
- **batch_size**: Integer. If unspecified, it will default to 32.
- **debug_info**: Bool, Should print out the logging info.
- **predict_kwargs**: Dict, arguments passed to `predict()` function of tensorflow.keras.Model

Returns:

- array of predictions.

predict_entities

Gets entities from sequence.

```
def predict_entities(self,
                     x_data,
                     batch_size=None,
                     join_chunk=' ',
                     debug_info=False,
                     predict_kwargs: Dict = None):
```

Args:

- x_data: The input data, as a Numpy array (or list of Numpy arrays if the model has multiple inputs).
- batch_size: Integer. If unspecified, it will default to 32.
- join_chunk: str or False,
- debug_info: Bool, Should print out the logging info.
- **predict_kwargs**: Dict, arguments passed to `predict()` function of tensorflow.keras.Model

Returns:

- list: list of entity.

evaluate

Evaluate model

```
def evaluate(self,
             x_data,
             y_data,
             batch_size=None,
```

(continues on next page)

(continued from previous page)

```
            digits=4,
            debug_info=False) -> Tuple[float, float, Dict]:
```

Args:

- **x_data**:
- **y_data**:
- **batch_size**:
- **digits**:
- **debug_info**:

save

Save model info json and model weights to given folder path

```
def save(self, model_path: str):
```

Args:

- **model_path**: target model folder path

info

Returns a dictionary containing the configuration of the model.

```
def info(self)
```

utils

Methods

unison_shuffled_copies

```
def unison_shuffled_copies(a, b)
```

get_list_subset

```
def get_list_subset(target: List, index_list: List[int]) -> List
```

custom_object_scope

```
def custom_object_scope()
```

load_model

Load saved model from saved model from `model.save` function

```
def load_model(model_path: str, load_weights: bool = True) -> BaseModel
```

Args:

- model_path: model folder path
- load_weights: only load model structure and vocabulary when set to False, default True.

Returns:

load_processor

```
def load_processor(model_path: str) -> BaseProcessor
```

Load processor from model, When we using tf-serving, we need to use model's processor to pre-process data

Args: model_path:

Returns:

convert_to_saved_model

Export model for tensorflow serving

```
def convert_to_saved_model(model: BaseModel,
                           model_path: str,
                           version: str = None,
                           inputs: Optional[Dict] = None,
                           outputs: Optional[Dict] = None):
```

Args:

- model: Target model
- model_path: The path to which the SavedModel will be stored.
- version: The model version code, default timestamp
- inputs: dict mapping string input names to tensors. These are added to the SignatureDef as the inputs.
- outputs: dict mapping string output names to tensors. These are added to the SignatureDef as the outputs.

callbacks

class EvalCallBack

__init__

Evaluate callback, calculate precision, recall and f1 at the end of each epoch step.

```
def __init__(self,
             kash_model: BaseModel,
             valid_x,
             valid_y,
             step=5,
             batch_size=256):
```

Args:

- **kash_model**: the kashgari model to evaluate
- **valid_x**: feature data for evaluation
- **valid_y**: label data for evaluation
- **step**: evaluate step, default 5
- **batch_size**: batch size, default 256

Methods

on_epoch_end

```
def on_epoch_end(self, epoch, logs=None):
```

Contributing & Support

We are happy to accept contributions that make `Kashgari` better and more awesome! You could contribute in various ways:

Bug Reports

1. Please **read the documentation** and **search the issue tracker** to try and find the answer to your question **before** posting an issue.
2. When creating an issue on the repository, please provide as much info as possible:
   - Version being used.
   - Operating system.
   - Version of Python.
   - Errors in console.
   - Detailed description of the problem.
   - Examples for reproducing the error. You can post pictures, but if specific text or code is required to reproduce the issue, please provide the text in a plain text format for easy copy/paste.

   The more info provided the greater the chance someone will take the time to answer, implement, or fix the issue.
3. Be prepared to answer questions and provide additional information if required. Issues in which the creator refuses to respond to follow up questions will be marked as stale and closed.

Reviewing Code

Take part in reviewing pull requests and/or reviewing direct commits. Make suggestions to improve the code and discuss solutions to overcome weakness in the algorithm.

Answer Questions in Issues

Take time and answer questions and offer suggestions to people who've created issues in the issue tracker. Often people will have questions that you might have an answer for. Or maybe you know how to help them accomplish a specific task they are asking about. Feel free to share your experience with others to help them out.

Pull Requests

Pull requests are welcome, and a great way to help fix bugs and add new features.

Accuracy Benchmarks

Use Kashgari your own data, and report the F-1 score.

Adding New Models

New models can be of two basic types:

Adding New Tasks

Currently, Kashgari can handle text-classification and sequence-labeling tasks. If you want to apply Kashgari for a new task, please submit a request issue and explain why we would consider adding the new task to Kashgari

Documentation Improvements

A ton of time has been spent not only creating and supporting this tool, but also spent making this documentation. If you feel it is still lacking, show your appreciation for the tool by helping to improve/translate the documentation.

Release notes

Upgrading

To upgrade Material to the latest version, use `pip`:

```
pip install --upgrade kashgari-tf
```

To inspect the currently installed version, use the following command:

```
pip show kashgari-tf
```

Current Release

[1.1.1] - 2020.03.13

- Add BERTEmbeddingV2.
- Migrate documents to https://readthedoc.org for the version control.

1.1.0 - 2019.12.27

- Add Scoring task. (#303)
- Add tokenizers.
- Fixing multi-label classification model loading. #304

1.0.0 - 2019.10.18

Unfortunately, we have to change the package name for clarity and consistency. Here is the new naming sytle.

Here is how the existing versions changes

0.5.4 - 2019.09.30

- Add shuffle parameter to fit function (#249)
- Improved type hinting for loaded model (#248)
- Fix loading models with CRF layers (#244, #228)
- Fix the configuration changes during embedding save/load (#224)
- Fix stacked embedding save/load (#224)
- Fix evaluate function where the list has int instead of str ([#222])
- Renaming model.pre_processor to model.processor
- Removing TensorFlow and numpy warnings
- Add docs how to specify which CPU or GPU
- Add docs how to compile model with custom optimizer

0.5.3 - 2019.08.11

- Fixing CuDNN Error (#198)

0.5.2 - 2019.08.10

- Add CuDNN Cell config, disable auto CuDNN cell. (#182, #198)

0.5.1 - 2019.07.15

- Rewrite documents with mkdocs
- Add Chinese documents
- Add `predict_top_k_class` for classification model to get predict probabilities (#146)
- Add `label2idx`, `token2idx` properties to Embeddings and Models
- Add `tokenizer` property for BERT Embedding. (#136)
- Add `predict_kwargs` for models `predict()` function
- Change multi-label classification's default loss function to binary_crossentropy (#151)

0.5.0 - 2019.07.11

tf.keras version

- Rewrite Kashgari using `tf.keras` (#77)
- Rewrite Documents
- Add TPU support
- Add TF-Serving support.
- Add advance customization support, like multi-input model
- Performance optimization

Legacy Version Changelog

0.2.6 - 2019.07.12

- Add tf.keras version info
- Fixing lstm issue in labeling model (#125)

0.2.4 - 2019.06.06

- Add BERT output feature layer fine-tune support. Discussion: (#103)
- Add BERT output feature layer number selection, default 4 according to BERT paper
- Fix BERT embedding token index offset issue (#104

0.2.1 - 2019.03.05

- fix missing `sequence_labeling_tokenize_add_bos_eos` config

0.2.0

- multi-label classification for all classification models
- support cuDNN cell for sequence labeling
- add option for output `BOS` and `EOS` in sequence labeling result, fix #31

0.1.9

- add `AVCNNModel`, `KMaxCNNModel`, `RCNNModel`, `AVRNNModel`, `DropoutBGRUModel`, `DropoutAVRNNModel` model to classification task.
- fix several small bugs

0.1.8

- fix BERT Embedding model's `to_json` function, issue #19

0.1.7

- remove class candidates filter to fix #16
- overwrite init function in CustomEmbedding
- add parameter check to custom_embedding layer
- add `keras-bert` version to setup.py file

0.1.6

- add `output_dict`, `debug_info` params to text_classification model
- add `output_dict`, `debug_info` and `chunk_joiner`params to text_classification model
- fix possible crash at data_generator

0.1.5

- fix sequence labeling evaluate result output
- refactor model save and load function

0.1.4

- fix classification model evaluate result output
- change test settings