tf_crnn Documentation

Sofia ARES OLIVEIRA

Nov 20, 2019

Contents

1 Quickstart **1**

1.1 Installation 1

1.2 How to train a model 1

1.3 Example of training 3

2 Reference guide **5**

2.1 Data handling for input function 5

2.2 Model definitions 5

2.3 Config for training 5

2.4 Custom Callbacks 6

2.5 Preprocessing data 6

3 References **13**

4 Indices and tables **15**

Bibliography **17**

Python Module Index **19**

Index **21**

CHAPTER 1

Quickstart

1.1 Installation

`tf_crnn` uses `tensorflow-gpu` package, which needs CUDA and CuDNN libraries for GPU support. Tensorflow GPU support page lists the requirements.

1.1.1 Using Anaconda

When using Anaconda (or Miniconda), conda will install automatically the compatible versions of CUDA and CuDNN

```
conda env create -f environment.yml
```

From this page:

> When the GPU accelerated version of TensorFlow is installed using conda, by the command "conda install tensorflow-gpu", these libraries are installed automatically, with versions known to be compatible with the tensorflow-gpu package. Furthermore, conda installs these libraries into a location where they will not interfere with other instances of these libraries that may have been installed via another method. Regardless of using pip or conda-installed tensorflow-gpu, the NVIDIA driver must be installed separately.

1.2 How to train a model

`sacred` package is used to deal with experiments. If you are not yet familiar with it, have a quick look at the documentation.

1.2.1 Input data

In order to train a model, you should input a csv file with each row containing the filename of the image (full path) and its label (plain text) separated by a delimiting character (let's say `;`). Also, each character should be separated by a splitting character (let's say `|`), this in order to deal with arbitrary alphabets (especially characters that cannot be encoded with `utf-8` format).

An example of such csv file would look like :

```
/full/path/to/image1.{jpg,png};|s|t|r|i|n|g|_|l|a|b|e|l|1|
/full/path/to/image2.{jpg,png};|s|t|r|i|n|g|_|l|a|b|e|l|2| |w|i|t|h| |special_char|
...
```

1.2.2 Input lookup alphabet file

You also need to provide a lookup table for the *alphabet* that will be used. The term *alphabet* refers to all the symbols you want the network to learn, whether they are characters, digits, symbols, abbreviations, or any other graphical element.

The lookup table is a dictionary mapping alphabet units to integer codes (i.e {'char' : <int_code>}). Some lookup tables are already provided as examples in `data/alphabet/`.

For example to transcribe words that contain only the characters *'abcdefg'*, one possible lookup table would be :

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4. 'e': 5, 'f': 6, 'g': 7}
```

The lookup table / dictionary needs to be saved in a json file.

1.2.3 Config file (with `sacred`)

Set the parameters of the experiment in `config.json`. The file looks like this :

```
{
  "lookup_alphabet_file" : "./data/alphabet/lookup.json",
  "csv_files_train" : "./data/csv_experiments/train_data.csv",
  "csv_files_eval" : "./data/csv_experiments/validation_data.csv",
  "output_model_dir" : "./output_model",
  "num_beam_paths" : 1,
  "max_chars_per_string" : 80,
  "n_epochs" : 50,
  "train_batch_size" : 64,
  "eval_batch_size" : 64,
  "learning_rate": 1e-4,
  "input_shape" : [128, 1400],
  "restore_model" : false
}
```

In order to use your data, you should change the parameters `csv_files_train`, `csv_files_eval` and `lookup_alphabet_file`.

All the configurable parameters can be found in class `tf_crnn.config.Params`, which can be added to the config file if needed.

1.2.4 Training

Once you have your input csv and alphabet file completed, and the parameters set in `config.json`, we will use `sacred` syntax to launch the training :

```
python training.py with config.json
```

The saved model and logs will then be exported to the folder specified in the config file (`output_model_dir`).

1.3 Example of training

We will use the IAM Database [MB02] as an example to generate the data in the correct input data and train a model.

Go to the official page to download the dataset and create an account in order to access the data. You don't need to download the data yourself, the `prepare_iam.py` script will take care of that for you.

1.3.1 Generating data

First create the `IAM_USER` and `IAM_PWD` environment variable to store your credentials, they will be used by the download script

```
export IAM_USER=<your-username>
export IAM_PWD=<your-password>
```

Run the script `hlp/prepare_iam.py` in order to download the data, extract it and format it correctly to train a model.

```
cd hlp
python prepare_iam.py --download_dir ../data/iam --generated_data_dir ../data/iam/
↪generated
cd ..
```

The images of the lines are extracted in `data/iam/lines/` and the folder `data/generated/` contains all the additional files necessary to run the experiment. The csv files are saved in `data/generated/generated_csv` and the alphabet is placed in `data/generated/generated_alphabet`.

1.3.2 Training the model

Make sure the `config.json` file has the correct paths for training and validation data, as well as for the alphabet lookup file and run:

```
python training.py with config.json
```

CHAPTER 2

Reference guide

2.1 Data handling for input function

`dataset_generator`(csv_filename, params[, . . .])	Generates the dataset for the experiment.
`padding_inputs_width`(image, target_shape, . . .)	Given an input image, will pad it to return a target_shape size padded image.
`augment_data`(image[, max_rotation, . . .])	Data augmentation on an image (padding, brightness, contrast, rotation)
`random_rotation`(img[, max_rotation, crop, . . .])	Rotates an image with a random angle.

2.2 Model definitions

`ConvBlock`(features, kernel_size, stride, . . .)	Convolutional block class.
`get_model_train`(parameters)	Constructs the full model for training.
`get_model_inference`(parameters[, weights_path])	Constructs the full model for prediction.
`get_crnn_output`(input_images[, parameters])	Creates the CRNN network and returns it's output.

2.3 Config for training

`Alphabet`([lookup_alphabet_file, blank_symbol])	Class for alphabet / symbols units.
`Params`(**kwargs)	Class for parameters of the model and the experiment
`import_params_from_json`([model_directory, . . .])	Read the exported json file with parameters of the experiment.

2.4 Custom Callbacks

CustomSavingCallback(output_dir, saving_freq)	Callback to save weights, architecture, and optimizer at the end of training.
LRTensorBoard(log_dir, **kwargs)	Adds learning rate to TensorBoard scalars.
CustomLoaderCallback(loading_dir)	Callback to load necessary weight and parameters for training, evaluation and prediction.
CustomPredictionSaverCallback(output_dir, . . .)	Callback to save prediction decoded outputs.

2.5 Preprocessing data

data_preprocessing(params)	Preporcesses the data for the experiment (training and evaluation data).
preprocess_csv(csv_filename, parameters, . . .)	Converts the original csv data to the format required by the experiment.

`tf_crnn.data_handler.`**`augment_data`**(*image*, *max_rotation=0.1*, *minimum_width=0*)

Data augmentation on an image (padding, brightness, contrast, rotation)

Parameters

- **`image`** (`MockObject`) – Tensor
- **`max_rotation`** (`float`) – float, maximum permitted rotation (in radians)
- **`minimum_width`** (`int`) – minimum width of image after data augmentation

Return type `MockObject`

Returns Tensor

`tf_crnn.data_handler.`**`dataset_generator`**(*csv_filename*, *params*, *use_labels=True*, *batch_size=64*, *data_augmentation=False*, *num_epochs=None*, *shuffle=True*)

Generates the dataset for the experiment.

Parameters

- **`csv_filename`** (`Union[List[str], str]`) – Path to csv file containing the data
- **`params`** (*`Params`*) – parameters df the experiment (`Params`)
- **`use_labels`** (`bool`) – boolean to indicate dataset generation during training / evaluation (true) or prediction (false)
- **`batch_size`** (`int`) – size of the generated batches
- **`data_augmentation`** (`bool`) – whether to use data augmentation strategies or not
- **`num_epochs`** (`Optional[int]`) – number of epochs to repeat the dataset generation
- **`shuffle`** (`bool`) – whether to suffle the data

Returns `tf.data.Dataset`

`tf_crnn.data_handler.`**`get_resized_width`**(*image*, *target_height*, *increment*)

Resizes the image according to *target_height*.

Parameters

- **image** (`MockObject`) – image to resize
- **target_height** (`int`) – height of the resized image
- **increment** (`int`) – reduction factor due to pooling between input width and output width, this makes sure that the final width will be a multiple of increment

Returns resized image

`tf_crnn.data_handler.`**`padding_inputs_width`**(*image*, *target_shape*, *increment*)

Given an input image, will pad it to return a target_shape size padded image. There are 3 cases:

- image width > target width : simple resizing to shrink the image
- image width >= 0.5*target width : pad the image
- image width < 0.5*target width : replicates the image segment and appends it

Parameters

- **image** (`MockObject`) – Tensor of shape [H,W,C]
- **target_shape** (`Tuple[int, int]`) – final shape after padding [H, W]
- **increment** (`int`) – reduction factor due to pooling between input width and output width, this makes sure that the final width will be a multiple of increment

Return type `Tuple[MockObject, MockObject]`

Returns (image padded, output width)

`tf_crnn.data_handler.`**`random_rotation`**(*img*, *max_rotation=0.1*, *crop=True*, *minimum_width=0*)

Rotates an image with a random angle. See https://stackoverflow.com/questions/16702966/rotate-image-and-crop-out-black-borders for formulae

Parameters

- **img** (`MockObject`) – Tensor
- **max_rotation** (`float`) – maximum angle to rotate (radians)
- **crop** (`bool`) – boolean to crop or not the image after rotation
- **minimum_width** (`int`) – minimum width of image after data augmentation

Return type `MockObject`

Returns

class `tf_crnn.config.`**`Alphabet`**(*lookup_alphabet_file=None*, *blank_symbol='$'*)

Class for alphabet / symbols units.

Variables

- **_blank_symbol** (*str*) – Blank symbol used for CTC
- **_alphabet_units** (*List[str]*) – list of elements composing the alphabet. The units may be a single character or multiple characters.
- **_codes** (*List[int]*) – Each alphabet unit has a unique corresponding code.
- **_nclasses** (*int*) – number of alphabet units.

`alphabet_units`

`blank_symbol`

`check_input_file_alphabet`(*csv_filenames*, *discarded_chars=';|\t\n\r\x0b\x0c'*, *csv_delimiter=';'*)

Checks if labels of input files contains only characters that are in the Alphabet.

Parameters

- **`csv_filenames`** (`List[str]`) – list of the csv filename
- **`discarded_chars`** (`str`) – discarded characters
- **`csv_delimiter`** (`str`) – character delimiting field in the csv file

Return type `None`

Returns

`codes`

`classmethod create_lookup_from_labels`(*csv_files*, *export_lookup_filename*, *original_lookup_filename=None*)

Create a lookup dictionary for csv files containing labels. Exports a json file with the Alphabet.

Parameters

- **`csv_files`** (`List[str]`) – list of files to get the labels from (should be of format path;label)
- **`export_lookup_filename`** (`str`) – filename to export alphabet lookup dictionary
- **`original_lookup_filename`** (`Optional[str]`) – original lookup filename to update (optional)

Returns

`classmethod load_lookup_from_json`(*json_filenames*)

Load a lookup table from a json file to a dictionnary :type json_filenames: `Union[List[str], str]` :param json_filenames: either a filename or a list of filenames :rtype: `dict` :return:

`classmethod make_json_lookup_alphabet`(*string_chars=None*)

Parameters **`string_chars`** (`Optional[str]`) – for example string.ascii_letters, string.digits

Return type `dict`

Returns

`classmethod map_lookup`(*lookup_table*, *unique_entry=True*)

Converts an existing lookup table with minimal range code ([1, len(lookup_table)-1]) and avoids multiple instances of the same code label (bijectivity)

Parameters

- **`lookup_table`** (`dict`) – dictionary to be mapped {alphabet_unit : code label}
- **`unique_entry`** (`bool`) – If each alphabet unit has a unique code and each code a unique alphabet unique ('bijective'), only True is implemented for now

Return type `dict`

Returns a mapped dictionary

`n_classes`

`class` `tf_crnn.config.`**`Params`**(***kwargs*)

Class for parameters of the model and the experiment

Variables

- **input_shape** (*Tuple[int, int]*) – input shape of the image to batch (this is the shape after data augmentation). The original will either be resized or pad depending on its original size
- **input_channels** (*int*) – number of color channels for input image (default: 1)
- **cnn_features_list** (*List(int)*) – a list of length *n_layers* containing the number of features for each convolutionl layer (default: [16, 32, 64, 96, 128])
- **cnn_kernel_size** (*List(int)*) – a list of length *n_layers* containing the size of the kernel for each convolutionl layer (default: [3, 3, 3, 3, 3])
- **cnn_stride_size** (*List((int, int))*) – a list of length *n_layers* containing the stride size each convolutionl layer (default: [(1, 1), (1, 1), (1, 1), (1, 1), (1, 1)])
- **cnn_pool_size** (*List((int, int))*) – a list of length *n_layers* containing the pool size each MaxPool layer default: ([(2, 2), (2, 2), (2, 2), (2, 2), (1, 1)])
- **cnn_batch_norm** (*List(bool)*) – a list of length *n_layers* containing a bool that indicated wether or not to use batch normalization (default: [False, False, False, False, False])
- **rnn_units** (*List(int)*) – a list containing the number of units per rnn layer (default: 256)
- **num_beam_paths** (*int*) – number of paths (transcriptions) to return for ctc beam search (only used when predicting)
- **csv_delimiter** (*str*) – character to delimit csv input files (default: ';')
- **string_split_delimiter** (*str*) – character that delimits each alphabet unit in the labels (default: '|')
- **csv_files_train** (*str*) – csv filename which contains the (path;label) of each training sample
- **csv_files_eval** (*str*) – csv filename which contains the (path;label) of each eval sample
- **lookup_alphabet_file** (*str*) – json file that contains the mapping alphabet units <-> codes
- ***blank_symbol*** (*str*) – symbol for to be considered as blank by the CTC decoder (default: '$')
- **max_chars_per_string** (*int*) – maximum characters per sample (to avoid CTC decoder errors) (default: 75)
- **data_augmentation** (*bool*) – if True augments data on the fly (default: true)
- **data_augmentation_max_rotation** (*float*) – max permitted roation to apply to image during training in radians (default: 0.005)
- **data_augmentation_max_slant** (*float*) – maximum angle for slant augmentation (default: 0.7)
- **n_epochs** (*int*) – numbers of epochs to run the training (default: 50)
- **train_batch_size** (*int*) – batch size during training (default: 64)
- **eval_batch_size** (*int*) – batch size during evaluation (default: 128)
- **learning_rate** (*float*) – initial learning rate (default: 1e-4)

- **evaluate_every_epoch** (*int*) – evaluate every 'evaluate_every_epoch' epoch (default: 5)
- **save_interval** (*int*) – save the model every 'save_interval' epoch (default: 20)
- **optimizer** (*str*) – which optimizer to use ('adam', 'rms', 'ada') (default: 'adam')
- **output_model_dir** (*str*) – output directory where the model will be saved and exported
- **restore_model** (*bool*) – boolean to continue training with saved weights (default: False)

classmethod from_json_file(*json_file*)
Given a json file, creates a `Params` object.

Parameters **json_file** (`str`) – path to the json file

Returns `Params` object

show_experiment_params()
Returns a dictionary with the variables of the class.

Return type `dict`

Returns

to_dict()
Returns the parameters as a dictionary

Return type `dict`

Returns

`tf_crnn.config.`**import_params_from_json**(*model_directory=None*, *json_filename=None*)
Read the exported json file with parameters of the experiment.

Parameters

- **model_directory** (`Optional[str]`) – Direcoty where the odel was exported
- **json_filename** (`Optional[str]`) – filename of the file

Return type `dict`

Returns a dictionary containing the parameters of the experiment

class `tf_crnn.model.`**ConvBlock**(*features*, *kernel_size*, *stride*, *cnn_padding*, *pool_size*, *batchnorm*, ***kwargs*)
Convolutional block class. It is composed of a *Conv2D* layer, a *BatchNormaization* layer (optional), a *MaxPool2D* layer (optional) and a *ReLu* activation.

Variables

- **features** (*int*) – number of features of the convolutional layer
- **kernel_size** (*int*) – size of the convolutional kernel
- **stride** (*int, int*) – stride of the convolutional layer
- **cnn_padding** – padding of the convolution ('same' or 'valid')
- **pool_size** (*int, int*) – size of the maxpooling
- **batchnorm** (*bool*) – use batch norm or not

call(*inputs*, *training=False*)

get_config()

Get a dictionary with all the necessary properties to recreate the same layer.

Return type `dict`

Returns dictionary containing the properties of the layer

`tf_crnn.model.`**get_crnn_output**(*input_images*, *parameters=None*)

Creates the CRNN network and returns it's output. Passes the *input_images* through the network and returns its output

Parameters

- **input_images** – images to process (B, H, W, C)
- **parameters** (`Optional[`*Params*`]`) – parameters of the model (`Params`)

Return type `MockObject`

Returns the output of the CRNN model

`tf_crnn.model.`**get_model_inference**(*parameters*, *weights_path=None*)

Constructs the full model for prediction. Defines inputs and outputs, and loads the weights.

Parameters

- **parameters** (*Params*) – parameters of the model (`Params`)
- **weights_path** (`Optional[str]`) – path to the weights (.h5 file)

Returns the model (`tf.Keras.Model`)

`tf_crnn.model.`**get_model_train**(*parameters*)

Constructs the full model for training. Defines inputs and outputs, loss function and metric (CER).

Parameters **parameters** (*Params*) – parameters of the model (`Params`)

Returns the model (`tf.Keras.Model`)

class `tf_crnn.callbacks.`**CustomLoaderCallback**(*loading_dir*)

Callback to load necessary weight and parameters for training, evaluation and prediction.

Variables **loading_dir** (*str*) – path to directory to save logs

set_model(*model*)

class `tf_crnn.callbacks.`**CustomPredictionSaverCallback**(*output_dir*, *parameters*)

Callback to save prediction decoded outputs. This will save the decoded outputs into a file.

Variables

- **output_dir** (*str*) – path to directory to save logs
- **parameters** (`Params`) – parameters of the experiment (`Params`)

on_predict_batch_end(*batch*, *logs*)

on_predict_begin(*logs=None*)

class `tf_crnn.callbacks.`**CustomSavingCallback**(*output_dir*, *saving_freq*, *save_best_only=False*, *keep_max_models=5*)

Callback to save weights, architecture, and optimizer at the end of training. Inspired by *ModelCheckpoint*.

Variables

- **output_dir** (*str*) – path to the folder where files will be saved
- **saving_freq** (*int*) – save every *n* epochs

- **save_best_only** (*bool*) – wether to save a model if it is best thant the last saving
- **keep_max_models** (*int*) – number of models to keep, the older ones will be deleted

on_epoch_begin(*epoch*, *logs=None*)

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

on_train_end(*logs=None*)

class tf_crnn.callbacks.**LRTensorBoard**(*log_dir*, ***kwargs*)

Adds learning rate to TensorBoard scalars.

Variables **logdir** (*str*) – path to directory to save logs

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

tf_crnn.preprocessing.**data_preprocessing**(*params*)

Preporcesses the data for the experiment (training and evaluation data). Exports the updated csv files into *<output_model_dir>/preprocessed/updated_{eval,train}.csv*

Parameters **params** (*Params*) – parameters of the experiment (Params)

Return type (<class 'str'>, <class 'str'>, <class 'int'>, <class 'int'>)

Returns output path files, number of samples (for train and evaluation data)

tf_crnn.preprocessing.**preprocess_csv**(*csv_filename*, *parameters*, *output_csv_filename*)

Converts the original csv data to the format required by the experiment. Removes the samples which labels have too many characters. Computes the widths of input images and removes the samples which have more characters per label than image width. Converts the string labels to dense codes. The output csv file contains the path to the image, the dense list of codes corresponding to the alphabets units (which are padded with 0 if *len(label)* < *max_len*) and the length of the label sequence.

Parameters

- **csv_filename** (str) – path to csv file
- **parameters** (*Params*) – parameters of the experiment (Params)
- **output_csv_filename** (str) – path to the output csv file

Return type int

Returns number of samples in the output csv file

CHAPTER 3

References

A TensorFlow implementation of the Convolutional Recurrent Neural Network (CRNN) for image-based sequence recognition tasks, such as scene text recognition and OCR.

This implementation uses `tf.keras` to build the model and `tf.data` modules to handle input data.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Bibliography

[MB02] U-V Marti and Horst Bunke. The iam-database: an english sentence database for offline handwriting recognition. *International Journal on Document Analysis and Recognition*, 5(1):39–46, 2002.

Python Module Index

t

tf_crnn, 5
tf_crnn.callbacks, 11
tf_crnn.config, 7
tf_crnn.data_handler, 6
tf_crnn.model, 10
tf_crnn.preprocessing, 12

Index

A

`Alphabet` (*class in tf_crnn.config*), 7
`alphabet_units` (*tf_crnn.config.Alphabet attribute*), 7
`augment_data()` (*in module tf_crnn.data_handler*), 6

B

`blank_symbol` (*tf_crnn.config.Alphabet attribute*), 7

C

`call()` (*tf_crnn.model.ConvBlock method*), 10
`check_input_file_alphabet()` (*tf_crnn.config.Alphabet method*), 8
`codes` (*tf_crnn.config.Alphabet attribute*), 8
`ConvBlock` (*class in tf_crnn.model*), 10
`create_lookup_from_labels()` (*tf_crnn.config.Alphabet class method*), 8
`CustomLoaderCallback` (*class in tf_crnn.callbacks*), 11
`CustomPredictionSaverCallback` (*class in tf_crnn.callbacks*), 11
`CustomSavingCallback` (*class in tf_crnn.callbacks*), 11

D

`data_preprocessing()` (*in module tf_crnn.preprocessing*), 12
`dataset_generator()` (*in module tf_crnn.data_handler*), 6

F

`from_json_file()` (*tf_crnn.config.Params class method*), 10

G

`get_config()` (*tf_crnn.model.ConvBlock method*), 10
`get_crnn_output()` (*in module tf_crnn.model*), 11
`get_model_inference()` (*in module tf_crnn.model*), 11
`get_model_train()` (*in module tf_crnn.model*), 11
`get_resized_width()` (*in module tf_crnn.data_handler*), 6

I

`import_params_from_json()` (*in module tf_crnn.config*), 10

L

`load_lookup_from_json()` (*tf_crnn.config.Alphabet class method*), 8
`LRTensorBoard` (*class in tf_crnn.callbacks*), 12

M

`make_json_lookup_alphabet()` (*tf_crnn.config.Alphabet class method*), 8
`map_lookup()` (*tf_crnn.config.Alphabet class method*), 8

N

`n_classes` (*tf_crnn.config.Alphabet attribute*), 8

O

`on_epoch_begin()` (*tf_crnn.callbacks.CustomSavingCallback method*), 12
`on_epoch_end()` (*tf_crnn.callbacks.CustomSavingCallback method*), 12
`on_epoch_end()` (*tf_crnn.callbacks.LRTensorBoard method*), 12
`on_predict_batch_end()` (*tf_crnn.callbacks.CustomPredictionSaverCallback method*), 11
`on_predict_begin()` (*tf_crnn.callbacks.CustomPredictionSaverCallback method*), 11

on_train_end() (*tf_crnn.callbacks.CustomSavingCallback method*), 12

P

padding_inputs_width() (*in module tf_crnn.data_handler*), 7
Params (*class in tf_crnn.config*), 8
preprocess_csv() (*in module tf_crnn.preprocessing*), 12

R

random_rotation() (*in module tf_crnn.data_handler*), 7

S

set_model() (*tf_crnn.callbacks.CustomLoaderCallback method*), 11
show_experiment_params() (*tf_crnn.config.Params method*), 10

T

tf_crnn (*module*), 5
tf_crnn.callbacks (*module*), 11
tf_crnn.config (*module*), 7
tf_crnn.data_handler (*module*), 6
tf_crnn.model (*module*), 10
tf_crnn.preprocessing (*module*), 12
to_dict() (*tf_crnn.config.Params method*), 10