tf_crnn Documentation

Sofia ARES OLIVEIRA

Nov 11, 2019

Contents

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
```

You can find more information about the installation procedures of CUDA and CuDNN with Anaconda here.

1.1.2 Using `pip`

Before using `tf_crnn` we recommend creating a virtual environment (python 3.5). Then, install the dependencies using Github repository's `setup.py` file.

```
pip install git+https://github.com/solivr/tf-crnn
```

You will then need to install CUDA and CuDNN libraries manually.

1.1.3 Using Docker

(thanks to PonteIneptique)

The `Dockerfile` in the root directory allows you to run the whole program as a Docker Nvidia Tensorflow GPU container. This is potentially helpful to deal with external dependencies like CUDA and the likes.

You can follow installations processes here :

- docker-ce : Ubuntu
- nvidia-docker : Ubuntu

Once this is installed, we will need to build the image of the container by doing :

```
nvidia-docker build . --tag tf-crnn
```

Our container model is now named `tf-crnn`. We will be able to run it from `nvidia-docker run -it tf-crnn:latest bash` which will open a bash directory exactly where you are. Although, we recommend using

```
nvidia-docker run -it -p 8888:8888 -p 6006:6006 -v /absolute/path/to/here/config:./
↪config -v $INPUT_DATA:/sources  tf-crnn:latest bash
```

where `$INPUT_DATA` should be replaced by the directory where you have your training and testing data. This will get mounted on the `sources` folder. We propose to mount by default `./config` to the current `./config` directory. Path need to be absolute path. We also recommend to change

```
//...
"output_model_dir" : "/.output/"
```

to

```
//...
"output_model_dir" : "/config/output"
```

Do not forget to rename your training and testing file path, as well as renaming the path to their image by `/sources/.../file.{png,jpg}`

Note: if you are uncomfortable with bash, you can always replace bash by `ipython3 notebook --allow-root` and go to your browser on `http://localhost:8888/` . A token will be shown in the terminal

---

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

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_template.json`. The file looks like this :

```
{
  "training_params" : {
    "learning_rate" : 1e-3,
    "learning_decay_rate" : 0.95,
    "learning_decay_steps" : 5000,
    "save_interval" : 1e3,
    "n_epochs" : 50,
    "train_batch_size" : 128,
    "eval_batch_size" : 128
  },
  "input_shape" : [32, 304],
  "string_split_delimiter" : "|",
  "csv_delimiter" : ";",
  "data_augmentation_max_rotation" : 0.1,
  "input_data_n_parallel_calls" : 4,
  "lookup_alphabet_file" : "./data/alphabet/lookup_letters_digits_symbols.json",
  "csv_files_train" : ["./data/csv/train_sample.csv"],
  "csv_files_eval" : ["./data/csv/eval_sample.csv"],
  "output_model_dir" : "./output/"
}
```

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

All the configurable parameters can be found in classes `tf_crnn.config.Params` and `tf_crnn.config.TrainingParams`, 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_template.json`, we will use `sacred` syntax to launch the training :

```
python train.py with config_template.json
```

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

1.3 Using a saved model for prediction

During the training, the model is exported every *n* epochs (you can set *n* in the config file, by default *n=5*). The exported models are `SavedModel` TensorFlow objects, which need to be loaded in order to be used.

Assuming that the output folder is named `output_dir`, the exported models will be saved in `output_dir/export/<timestamp>` with different timestamps for each export. Each `<timestamp>` folder contains a `saved_model.pb` file and a `variables` folder.

The `saved_model.pb` contains the graph definition of your model and the `variables` folder contains the saved variables (where the weights are stored). You can find more information about SavedModel on the TensorFlow dedicated page.

In order to easily handle the loading of the exported models, a `PredictionModel` class is provided and you can use the trained model to transcribe new image segments in the following way :

```
import tensorflow as tf
from tf_crnn.loader import PredictionModel

model_directory = 'output/export/<timestamp>/'
image_filename = 'data/images/b04-034-04-04.png'

with tf.Session() as session:
    model = PredictionModel(model_directory, signature='filename')
    prediction = model.predict(image_filename)
```

CHAPTER 2

Reference guide

The *tf_crnn.data_handler*

2.1 Data handling for input function

`data_loader`(csv_filename, str], params[, labels])	Loads, preprocesses (data augmentation, padding) and feeds the data
`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.
`random_padding`(image, max_pad_w, max_pad_h)	Given an image will pad its border adding a random number of rows and columns
`serving_single_input`(fixed_height, min_width)	Serving input function needed for export (in TensorFlow).

2.2 Config for training

`Alphabet`(lookup_alphabet_file, blank_symbol)	Object for alphabet / symbols units.
`TrainingParams`(**kwargs)	Object for parameters related to the training.
`Params`(**kwargs)	Object for general parameters
`import_params_from_json`(model_directory, ...)	Read the exported json file with parameters of the experiment.

2.3 Model

deep_cnn(input_imgs, input_channels, ...)	CNN part of the CRNN network.
deep_bidirectional_lstm(inputs, params, ...)	Recurrent part of the CRNN network.
crnn_fn(features, labels, mode, params)	CRNN model definition for tf.Estimator.
get_words_from_chars(characters_list, ...[, ...])	Joins separated characters to form words.

2.4 Loading exported model

PredictionModel(model_dir, session, signature)	Helper class to load an exported model and apply it to image segments for transcription.

tf_crnn.data_handler.**augment_data**(*image: <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36cd1550>, max_rotation: float = 0.1*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36cd15f8>

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

Parameters

- **image** – Tensor
- **max_rotation** – float, maximum permitted rotation (in radians)

Returns Tensor

tf_crnn.data_handler.**data_loader**(*csv_filename: Union[List[str], str], params: tf_crnn.config.Params, labels=True, batch_size: int = 64, data_augmentation: bool = False, num_epochs: int = None, image_summaries: bool = False*)

Loads, preprocesses (data augmentation, padding) and feeds the data

Parameters

- **csv_filename** – filename or list of filenames
- **params** – Params object containing all the parameters
- **labels** – transcription labels
- **batch_size** – batch_size
- **data_augmentation** – flag to select or not data augmentation
- **num_epochs** – feeds the data 'num_epochs' times
- **image_summaries** – floag to show image summaries or not

Returns data_loader function

tf_crnn.data_handler.**padding_inputs_width**(*image: <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36cd1518>, target_shape: Tuple[int, int], increment: int*) → Tuple[<sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36c80518>, <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36c802e8>]

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** – Tensor of shape [H,W,C]
- **target_shape** – final shape after padding [H, W]
- **increment** – 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 (image padded, output width)

tf_crnn.data_handler.**random_padding**(*image: <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b42b20400>, max_pad_w: int = 5, max_pad_h: int = 10*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36c729b0>

Given an image will pad its border adding a random number of rows and columns

Parameters

- **image** – image to pad
- **max_pad_w** – maximum padding in width
- **max_pad_h** – maximum padding in height

Returns a padded image

tf_crnn.data_handler.**random_rotation**(*img: <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b42b202e8>, max_rotation: float = 0.1, crop: bool = True*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b42b20320>

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

Parameters

- **img** – Tensor
- **max_rotation** – maximum angle to rotate (radians)
- **crop** – boolean to crop or not the image after rotation

Returns

tf_crnn.data_handler.**serving_single_input**(*fixed_height: int = 32, min_width: int = 8*)

Serving input function needed for export (in TensorFlow). Features to serve :

- *images* : greyscale image
- *input_filename* : filename of image segment
- *input_rgb*: RGB image segment

Parameters

- **fixed_height** – height of the image to format the input data with
- **min_width** – minimum width to resize the image

Returns serving_input_fn

class tf_crnn.config.**Alphabet**(*lookup_alphabet_file: str = None*, *blank_symbol: str = '$'*)
Object 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: List[str], discarded_chars: str = ';|\t\n\r\x0b\x0c', csv_delimiter: str = ';'*) → None
Checks if labels of input files contains only characters that are in the Alphabet.

Parameters

- **csv_filenames** – list of the csv filename
- **discarded_chars** – discarded characters
- **csv_delimiter** – character delimiting field in the csv file

Returns

codes

classmethod create_lookup_from_labels(*csv_files: List[str], export_lookup_filename: str, original_lookup_filename: str = None*)
Create a lookup dictionary for csv files containing labels. Exports a json file with the Alphabet.

Parameters

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

Returns

n_classes

class tf_crnn.config.**Params**(***kwargs*)
Object for general parameters

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
- **csv_delimiter** (*str*) – character to delimit csv input files
- **string_split_delimiter** (*str*) – character that delimits each alphabet unit in the labels

- **num_gpus** (*int*) – number of gpus to use
- **lookup_alphabet_file** (*str*) – json file that contains the mapping alphabet units <-> codes
- **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
- **output_model_dir** (*str*) – output directory where the model will be saved and exported
- ***keep_prob_dropout*** (*float*) – keep probability
- **num_beam_paths** (*int*) – number of paths (transcriptions) to return for ctc beam search (only used when predicting)
- **data_augmentation** (*bool*) – if True augments data on the fly
- **data_augmentation_max_rotation** (*float*) – max permitted roation to apply to image during training (radians)
- **input_data_n_parallel_calls** (*int*) – number of parallel calls to make when using Dataset.map()

keep_prob_dropout

show_experiment_params() → dict
Returns a dictionary with the variables of the class. :return:

class tf_crnn.config.**TrainingParams**(***kwargs*)
Object for parameters related to the training.

Variables

- **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)
- **learning_decay_rate** (*float*) – decay rate for exponential learning rate (default: .96)
- **learning_decay_steps** (*int*) – decay steps for exponential learning rate (default: 1000)
- **evaluate_every_epoch** (*int*) – evaluate every 'evaluate_every_epoch' epoch (default: 5)
- **save_interval** (*int*) – save the model every 'save_interval' step (default: 1e3)
- **optimizer** (*str*) – which optimizer to use ('adam', 'rms', 'ada') (default: 'adam)

to_dict() → dict

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

Parameters

- **model_directory** – Direcoty where the odel was exported

- **json_filename** – filename of the file

Returns a dictionary containing the parameters of the experiment

tf_crnn.model.**crnn_fn**(*features*, *labels*, *mode*, *params*)

CRNN model definition for `tf.Estimator`. Combines `deep_cnn` and `deep_bidirectional_lstm` to define the model and adds loss computation and CTC decoder.

Parameters

- **features** – dictionary with keys : '*images*', '*images_widths*', '*filenames*'
- **labels** – string containing the transcriptions. Flattend (1D) array with encoded label (one code per character)
- **mode** – TRAIN, EVAL, PREDICT
- **params** – dictionary with keys: '*Params*', '*TrainingParams*'

Returns

tf_crnn.model.**deep_bidirectional_lstm**(*inputs: <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36c84e80>*, *params: tf_crnn.config.Params*, *summaries: bool = True*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36c89080>

Recurrent part of the CRNN network. Uses a biderectional LSTM.

Parameters

- **inputs** – output of `deep_cnn`
- **params** – parameters of the model
- **summaries** – flag to enable bias and weight histograms to be visualized in Tensorboard

Returns Tuple : (tensor [width(time), batch, n_classes], raw transcription codes)

tf_crnn.model.**deep_cnn**(*input_imgs: <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36c84e10>*, *input_channels: int*, *is_training: bool*, *summaries: bool = True*) → <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36c84f98>

CNN part of the CRNN network.

Parameters

- **input_imgs** – input images [B, H, W, C]
- **input_channels** – input channels, 1 for greyscale images, 3 for RGB color images
- **is_training** – flag to indicate training or not
- **summaries** – flag to enable bias and weight histograms to be visualized in Tensorboard

Returns tensor of shape [batch, final_width, final_height x final_features]

class tf_crnn.loader.**PredictionModel**(*model_dir: str*, *session: <sphinx.ext.autodoc.importer._MockObject object at 0x7f7b36c89898> = None*, *signature: str = 'predictions'*)

Helper class to load an exported model and apply it to image segments for transcription.

Variables

- **session** (*tf.Session*) – `tf.Session` within which to run the loading process
- ***model*** – loaded exported model

Parameters

- **model_dir** – directory containing the saved model files.
- **session** – `tf.Session` to load the model
- **signature** – which signature to use to select the type of input :
  - predictions (default) : input a grayscale image
  - rgb_images : input a RGB image
  - filename : input the filename of the image segment

predict(*input_to_predict: Union[numpy.ndarray, str]*) → dict

Get transcription for input data.

Parameters **input_to_predict** – input data of the format specified in *signature* when instantiating the object

Returns a dictionary with the predictions

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

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

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index