
deepspeech.pytorch



Implementation of DeepSpeech2 for PyTorch using PyTorch Lightning. The repo supports training/testing and inference using the DeepSpeech2 model. Optionally a kenlm language model can be used at inference time.

Install

Several libraries are needed to be installed for training to work. I will assume that everything is being installed in an Anaconda installation on Ubuntu, with PyTorch installed.

Install PyTorch if you haven't already.

If you want decoding to support beam search with an optional language model, install ctcdecode:

```
1 git clone --recursive https://github.com/parlance/ctcdecode.git
2 cd ctcdecode && pip install .
```

Finally clone this repo and run this within the repo:

```
1 pip install -r requirements.txt
2 pip install -e . # Dev install
```

If you plan to use Multi-node training, you'll need etcd. Below is the command to install on Ubuntu.

```
1 sudo apt-get install etcd
```

Docker

To use the image with a GPU you'll need to have nvidia-docker installed.

```
1 sudo docker run -ti --gpus all -v `pwd`/data:/workspace/data --tmpfs /
  tmp -p 8888:8888 --net=host --ipc=host seannaren/deepspeech.pytorch:
  latest # Opens a Jupyter notebook, mounting the /data drive in the
  container
```

Optionally you can use the command line by changing the entrypoint:

```
1 sudo docker run -ti --gpus all -v `pwd`/data:/workspace/data --tmpfs /
  tmp --entrypoint=/bin/bash --net=host --ipc=host seannaren/
  deepspeech.pytorch:latest
```

Training

Datasets

Currently supports AN4, TEDLIUM, Voxforge, Common Voice and LibriSpeech. Scripts will setup the dataset and create manifest files used in data-loading. The scripts can be found in the data/ folder. Many of the scripts allow you to download the raw datasets separately if you choose so.

Training Commands

AN4

```
1 cd data/ && python an4.py && cd ..
2
3 python train.py +configs=an4
```

LibriSpeech

```
1 cd data/ && python librispeech.py && cd ..
2
3 python train.py +configs=librispeech
```

Common Voice

```
1 cd data/ && python common_voice.py && cd ..
2
3 python train.py +configs=commonvoice
```

TEDLIUM

```
1 cd data/ && python ted.py && cd ..
2
3 python train.py +configs=tedium
```

Custom Dataset To create a custom dataset you must create a JSON file containing the locations of the training/testing data. This has to be in the format of:

```
1 {
2   "root_path":"path/to",
3   "samples":[
4     {"wav_path":"audio.wav","transcript_path":"text.txt"},
5     {"wav_path":"audio2.wav","transcript_path":"text2.txt"},
6     ...
7   ]
8 }
```

Where the `root_path` is the root directory, `wav_path` is to the audio file, and the `transcript_path` is to a text file containing the transcript on one line. This can then be used as stated below.

Note on CSV files ... Up until release V2.1, deepspeech.pytorch used CSV manifest files instead of JSON. These manifest files are formatted similarly as a 2 column table:

```
1 /path/to/audio.wav,/path/to/text.txt
2 /path/to/audio2.wav,/path/to/text2.txt
3 ...
```

Note that this format is incompatible V3.0 onwards.

Merging multiple manifest files To create bigger manifest files (to train/test on multiple datasets at once) we can merge manifest files together like below.

```
1 cd data/
2 python merge_manifests.py manifest_1.json manifest_2.json --out
   new_manifest_dir
```

Modifying Training Configs

Configuration is done via Hydra.

Defaults can be seen in config.py. Below is how you can override values set already:

```
1 python train.py data.train_path=data/train_manifest.json data.val_path=
   data/val_manifest.json
```

Use `python train.py --help` for all parameters and options.

You can also specify a config file to keep parameters stored in a yaml file like so:

Create folder `experiment/` and file `experiment/an4.yaml`:

```
1 data:
2   train_path: data/an4_train_manifest.json
3   val_path: data/an4_val_manifest.json
```

```
1 python train.py +experiment=an4
```

To see options available, check [here](#).

Multi-GPU Training

We support single-machine multi-GPU training via PyTorch Lightning.

Below is an example command when training on a machine with 4 local GPUs:

```
1 python train.py +configs=an4 trainer.gpus=4
```

Multi-Node Training

Also supported is multi-machine capabilities using TorchElastic. This requires a node to exist as an explicit etcd host (which could be one of the GPU nodes but isn't recommended), a shared mount across your cluster to load/save checkpoints and communication between the nodes.

Below is an example where we've set one of our GPU nodes as our etcd host however if you're scaling up, it would be suggested to have a separate instance as your etcd instance to your GPU nodes as this will be a single point of failure.

Assumed below is a shared drive called /share where we save our checkpoints and data to access.

Run on the etcd host:

```
1 PUBLIC_HOST_NAME=127.0.0.1 # Change to public host name for all nodes  
  to connect  
2 etcd --enable-v2 \  
3    --listen-client-urls http://$PUBLIC_HOST_NAME:4377 \  
4    --advertise-client-urls http://$PUBLIC_HOST_NAME:4377 \  
5    --listen-peer-urls http://$PUBLIC_HOST_NAME:4379
```

Run on each GPU node:

```
1 python -m torchelastic.distributed.launch \  
2     --nnodes=2 \  
3     --nproc_per_node=4 \  
4     --rdzv_id=123 \  
5     --rdzv_backend=etcd \  
6     --rdzv_endpoint=$PUBLIC_HOST_NAME:4377 \  
7     train.py data.train_path=/share/data/an4_train_manifest.json \  
8             data.val_path=/share/data/an4_val_manifest.json model.  
9             precision=half \  
10            data.num_workers=8 checkpoint.save_folder=/share/  
11            checkpoints/ \  
12            checkpoint.checkpoint=true checkpoint.  
            load_auto_checkpoint=true checkpointing.  
            save_n_recent_models=3 \  
            data.batch_size=8 trainer.max_epochs=70 \  
            trainer.accelerator=ddp trainer.gpus=4 trainer.  
            num_nodes=2
```

Using the `load_auto_checkpoint=true` flag we can re-continue training from the latest saved checkpoint.

Currently it is expected that there is an NFS drive/shared mount across all nodes within the cluster to load the latest checkpoint from.

Augmentation

There is support for three different types of augmentations: SpecAugment, noise injection and random tempo/gain perturbations.

SpecAugment Applies simple Spectral Augmentation techniques directly on Mel spectrogram features to make the model more robust to variations in input data. To enable SpecAugment, use the `--spec-augment` flag when training.

SpecAugment implementation was adapted from this project.

Noise Injection Dynamically adds noise into the training data to increase robustness. To use, first fill a directory up with all the noise files you want to sample from. The dataloader will randomly pick samples from this directory.

To enable noise injection, use the `--noise-dir /path/to/noise/dir/` to specify where your noise files are. There are a few noise parameters to tweak, such as `--noise_prob` to determine the probability that noise is added, and the `--noise-min`, `--noise-max` parameters to determine the minimum and maximum noise to add in training.

Included is a script to inject noise into an audio file to hear what different noise levels/files would sound like. Useful for curating the noise dataset.

```
1 python noise_inject.py --input-path /path/to/input.wav --noise-path /  
  path/to/noise.wav --output-path /path/to/input_injected.wav --noise-  
  level 0.5 # higher levels means more noise
```

Tempo/Gain Perturbation Applies small changes to the tempo and gain when loading audio to increase robustness. To use, use the `--speed-volume-perturb` flag when training.

Checkpoints

Typically checkpoints are stored in `lightning_logs/` in the current working directory of the script.

This can be adjusted:

```
1 python train.py checkpoint.file_path=save_dir/
```

To load a previously saved checkpoint:

```
1 python train.py trainer.resume_from_checkpoint=lightning_logs/
  deepspeech_checkpoint_epoch_N_iter_N.ckpt
```

This continues from the same training state.

Testing/Inference

To evaluate a trained model on a test set (has to be in the same format as the training set):

```
1 python test.py model.model_path=models/deepspeech.pth test_path=/path/
  to/test_manifest.json
```

An example script to output a transcription has been provided:

```
1 python transcribe.py \
2     model.model_path=models/deepspeech.pth \
3     model.cuda=True \
4     chunk_size_seconds=-1 \
5     audio_path=audio_path=/path/to/audio.wav
```

If you used mixed-precision or half precision when training the model, you can use the `model.precision=half` for a speed/memory benefit. If you want to transcribe a long audio file that does not fit in the GPU, change the value of `chunk_size_seconds` to a positive number which represents the chunk size in seconds that will be used to segment the long audio file based on it.

Inference Server

Included is a basic server script that will allow post request to be sent to the server to transcribe files.

```
1 python server.py --host 0.0.0.0 --port 8000 # Run on one window
2
3 curl -X POST http://0.0.0.0:8000/transcribe -H "Content-type: multipart
  /form-data" -F "file=@/path/to/input.wav"
```

Using an ARPA LM

We support using kenlm based LMs. Below are instructions on how to take the LibriSpeech LMs found [here](#) and tune the model to give you the best parameters when decoding, based on LibriSpeech.

Tuning the LibriSpeech LMs

First ensure you've set up the librispeech datasets from the data/ folder. In addition download the latest pre-trained librispeech model from the releases page, as well as the ARPA model you want to tune from here. For the below we use the 3-gram ARPA model (3e-7 prune).

First we need to generate the acoustic output to be used to evaluate the model on LibriSpeech val.

```
1 python test.py data.test_path=data/librispeech_val_manifest.json model.  
  model_path=librispeech_pretrained_v2.pth save_output=  
  librispeech_val_output.npy
```

We use a beam width of 128 which gives reasonable results. We suggest using a CPU intensive node to carry out the grid search.

```
1 python search_lm_params.py --num-workers 16 --saved-output  
  librispeech_val_output.npy --output-path libri_tune_output.json --lm  
  -alpha-from 0 --lm-alpha-to 5 --lm-beta-from 0 --lm-beta-to 3 --lm-  
  path 3-gram.pruned.3e-7.arpa --model-path librispeech_pretrained_v2  
  .pth --beam-width 128 --lm-workers 16
```

This will run a grid search across the alpha/beta parameters using a beam width of 128. Use the below script to find the best alpha/beta params:

```
1 python select_lm_params.py --input-path libri_tune_output.json
```

Use the alpha/beta parameters when using the beam decoder.

Building your own LM

To build your own LM you need to use the KenLM repo found here. Have a read of the documentation to get a sense of how to train your own LM. The above steps once trained can be used to find the appropriate parameters.

Alternate Decoders

By default, `test.py` and `transcribe.py` use a `GreedyDecoder` which picks the highest-likelihood output label at each timestep. Repeated and blank symbols are then filtered to give the final output.

A beam search decoder can optionally be used with the installation of the `ctcdecode` library as described in the Installation section. The `test` and `transcribe` scripts have a `lm` config. To use the beam decoder, add `lm.decoder_type=beam`. The beam decoder enables additional decoding parameters: - **lm.beam_width** how many beams to consider at each timestep - **lm.lm_path** optional

binary KenLM language model to use for decoding - **lm.alpha** weight for language model - **lm.beta** bonus weight for words

Time offsets

Use the `offsets=true` flag to get positional information of each character in the transcription when using `transcribe.py` script. The offsets are based on the size of the output tensor, which you need to convert into a format required. For example, based on default parameters you could multiply the offsets by a scalar (duration of file in seconds / size of output) to get the offsets in seconds.

Pre-trained models

Pre-trained models can be found under releases [here](#).

Acknowledgements

Thanks to Egor and Ryan for their contributions!