Catskills Research Company application of NVidia NeMo Quarts 15x5 model trained from scratch for 16000 Hz sample rate for Somali

Lars Ericson

Quantitative Analytics Specialist

Catskills Research Company

1334 Hudson Place Davidson, NC 28036

lars.ericson@wellsfargo.com (mailto:lars.ericson@wellsfargo.com)

12 November 2020

Abstract

We describe the Catskills Research Company system for NIST OpenASR20. In EVAL Constrained condition, this system scored a WER of 1.13849 and 4th place out of 5 on the Leaderboard.

Core algorithmic approach

We used the NVidia NeMo ASR package [1] and followed their instructions [2] for training a new language from scratch (Constrained condition) using the QuartzNet 15x5 model [3]. This involved creating a YAML file for Somali by modifying the example YAML file [4].

The main modifications were to

- Input the **grapheme set** for Somali
- Decide on **maximum duration** in seconds of input sample. We chose 10 seconds and limited our training samples to transcriptions that were 10 seconds or less in the BUILD set.
- Decide on the **sample rate**. Because we initially worked with the pretrained model (Unconstrained condition), which uses 16000Hz sample rate, we stayed with 16000Hz rate for the Constrained condition (probably a mistake, as it increases parameter size for no added value).
- Decide on the initial **learning rate**. We chose a relatively high rate of 0.02 which is double the normal Novograd recommended starting rate of 0.01, because of use of highly augmented samples for training.
- Decide on the **batch size**. We chose a batch size of 180 to fit our GPU, because we felt that larger batch size would minimize overfitting.

So, the following entries were changed in the base YAML file to make the Somali YAML file:

```
1 sample_rate: &sample_rate 16000
```

```
labels: &labels  [' ', "'", 'a', 'b', 'c', 'd', 'e',
 'f', 'g', 'h', 'i', 'j',
 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r',
 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

model:
  train_ds:
    sample_rate: 16000
    batch_size: 180
    max_duration: 10.0

  optim:
    lr: .002
```

A new model from scratch is created by instantiating the `nemo_asr.models.EncDecCTCModel` class in NeMo.

Additional features and tools used, including software packages and publicly available external resources

We used:

- Python 3.7.9
- `sph2pip_v2.5` for SPH to WAV conversion [5]
- Python modules `IPython`, `Levenshtein`, `OpenASR_convert_reference_transcript`, `argparse`, `audioread`, `csv`, `datetime`, `glob`, `itertools`, `json`, `json`, `librosa`, `logging`, `matplotlib`, `multiprocessing`, `nemo`, `numpy`, `omegaconf`, `operator`, `os`, `pandas`, `pathlib`, `pickle`, `pprint`, `pytorch_lightning`, `random`, `re`, `ruamel`, `scipy`, `shutil`, `soundfile`, `sys`, `tarfile`, `torch`, `torchtext`, `tqdm`, `unidecode`, `warnings`

Other data used (outside provided data)

Only NIST BABEL Somali BUILD samples were used for training.

Significant data pre-/post-processing

Data augmentation

Training audio was split according to the transcript into smaller samples per the timecodes on the scripts.

Each sample was then augmented with random variations using NeMo provided perturbations [6][7].

In particular we applied the following 3 perturbations in sequence 10 times to get 10 new samples:

- **Time stretch** from 0.8 to 1.2. (Not pitch preserving.)
- **Speed change** from 0.8 to 1.2. (Pitch preserving.)
- **White noise** from -70db to -35 db.

This is implemented in the following class:

```
from nemo.collections.asr.parts import perturb

class Disturb:

    def __init__(self, _sample_rate):
        self.sample_rate = _sample_rate
        self.white_noise = \
           perturb.WhiteNoisePerturbation(min_level=-70,
max_level=-35)
        self.speed = perturb.SpeedPerturbation(self.sample_rate,
            'kaiser_best', min_speed_rate=0.8,
            max_speed_rate=1.2, num_rates=-1)
        self.time_stretch = \
            perturb.TimeStretchPerturbation(min_speed_rate=0.8,
                    max_speed_rate=1.2, num_rates=3)

    def __call__(self, _sample):
        sample=deepcopy(_sample)
        self.time_stretch.perturb(sample)
        self.speed.perturb(sample)
        self.white_noise.perturb(sample)
        return sample
```

Speaker activity detection and translation

To reduce the unlabelled DEV and EVAL data to clips of at most 10 seconds in length, it is necessary to implement a Speaker Activity Detection function. We explored NeMo templates for training a neural network for this purpose. This approach resulted in a very slow function. We chose instead to implement an ad hoc manually tuned method which relies on the absolute value of the dB level of the mel spectogram to find suitably long periods of silence to cut the clips at. The method is implemented as follows:

```
def smooth(y, w):
    box = np.ones(w)/w
    y_smooth = np.convolve(y, box, mode='same')
    return y_smooth

def smoothhtooms(A,w):
    A1=smooth(A,w)
    A2=smooth(A1[::-1],w)
    return A2[::-1]

def listen_and_transcribe(C, model, max_duration, gold, audio):
    audio /= max(abs(audio.min()), abs(audio.max()))
    size=audio.shape[0]
    T=size/C.sample_rate
    X=np.arange(size)/C.sample_rate
```

```
    Z=np.zeros(size)
    S = librosa.feature.melspectrogram(y=audio,
        sr=C.sample_rate, n_mels=64, fmax=8000)
    dt_S=T/S.shape[1]
    samples_per_spect=int(dt_S*C.sample_rate)
    S_dB = librosa.power_to_db(S, ref=np.max)
    s_dB_mean=np.mean(S_dB,axis=0)
    max_samples=int(max_duration/dt_S)
    min_samples=1
    pred=[]
    cutoffs = np.linspace(-80,-18,200)
    max_read_head=s_dB_mean.shape[0]
    max_read_head, max_samples, min_samples
    read_head=0
    transcriptions=[]
    read_heads=[]
    read_heads=[read_head]
    while read_head < max_read_head:
        finished = False
        while not finished and read_head < max_read_head:
            for cutoff in cutoffs:
                speech_q=(s_dB_mean[read_head:]>cutoff)
                silences=collect_false(speech_q)
                silences=[(x,y) for x,y in silences
                          if x != y and y-x > min_samples]
                n_silences = len(silences)
                if n_silences==0:
                    continue
                elif silences[0][0] == 0 and silences[0][1] != 0:
                    read_head +=silences[0][1]
                    break
                elif silences[0][0] > max_samples:
                    continue
                else:
                    silences=[(x,y) for x,y in silences
                              if x <= max_samples]
                    if not len(silences):
                        continue
                    start_at = read_head
                    stop_at= read_head + silences[0][0]
                    read_head = stop_at
                    finished = True
                    break
            if not finished:
                display_start=read_head*samples_per_spect
                display_end=display_start+max_samples
                start_at = read_head
                stop_at = min(max_read_head,
                              read_head + max_samples)
                read_head = stop_at
                finished = True
        read_heads.append(read_head)
        start=start_at*samples_per_spect
        end=start_at+stop_at*samples_per_spect
        display_start=max(0, start-5*C.sample_rate)
        display_end=end+5*C.sample_rate
        smooth_abs=smoothhtooms(np.abs(audio[start:end]), 100)
```

```
        smooth_abs_max=smooth_abs.max()
        if smooth_abs_max >= 0.05:
            try:
                segment_transcript, timeline, \
                normalized_power, speech_mask, clip_audio=\
                    predicted_segment_transcript(C, \
                          model, audio, \
                          start, end, s_dB_mean, \
                          samples_per_spect, dt_S)
                transcriptions.extend(segment_transcript)
            except:
            print("empty translation")
    transcriptions = [(time, time+duration, pred)
                      for time, duration, pred in transcriptions]
    return transcriptions
```

The translation of a clip of 10 seconds or less in duration is performed by function `predicted_segment_transcript` . This relies on similar thinking to break the transcribed clip into silent and speech components, and then allocate the words of the result proportionally in word size to speech component size:

```
def normalize(A):
    A=np.copy(A)
    A=A-A.min()
    A=A/A.max()
    return A

def predicted_segment_transcript(C, model, audio,
            start, end, s_dB_mean, samples_per_spect, dt_S):
    clip_audio=audio[start:end]
    prediction=transcribe(C, model, clip_audio)
    print(f"PRED {start/C.sample_rate:2f} {prediction}")
    spec_start=int(start/samples_per_spect)
    spec_end=int(end/samples_per_spect)
    clip_power=s_dB_mean[spec_start:spec_end]
    normalized_power=normalize(np.copy(clip_power))
    timeline=np.arange(spec_start,spec_end)*dt_S
    w=min(30, normalized_power.shape[0])

smoothed_normalized_power=normalize(smooth(normalized_power,w))
    speech_mask=extremize(smoothed_normalized_power, 0.2)
    speech_segments=mask_boundaries(speech_mask)+spec_start
    spec_to_words=allocate_pred_to_speech_segments(prediction,
speech_segments)
    if len(spec_to_words)==0:
        return None
    segment_transcript = \
        [(spec1*dt_S, (spec2-spec1)*dt_S, word)
        for spec1, spec2, word in spec_to_words]
    return segment_transcript, timeline, \
           normalized_power, speech_mask, clip_audio
```

This in turn relies on a function to call the model to transcribe the audio into graphemes:

```
def transcribe(C, model, audio):
    fn='tmp.wav'
    sf.write(fn, audio, C.sample_rate)
    translations=model.transcribe(paths2audio_files=[fn],
batch_size=1)
    translation=translations[0]
    translation=translation.split(' ')
    translation=' '.join([x.strip() for x in translation if
len(x)])
    return translation.replace("\u200c",'')  # Just Pashto but
required
```

and a function to do the allocation of predicted text to speech segments:

```
def align_seg_words(seg_words):
    ([seg_start, seg_end], seg_wrds) = seg_words
    seg_duration=seg_end-seg_start
    n_seg_wrds=len(seg_wrds)
    word_duration=seg_duration//n_seg_wrds
    seg_duration, word_duration
    seg_word_boundaries=np.hstack([np.linspace(seg_start, \
        seg_end-word_duration, n_seg_wrds).astype(int),
[seg_end]])
    seg_aligned_wrds=[(seg_word_boundaries[i],
                       seg_word_boundaries[i+1], seg_wrds[i])
        for i in range(n_seg_wrds)]
    return seg_aligned_wrds

def align_segment_words(segment_words):
    return [z for y in [align_seg_words(x) for x in segment_words]
for z in y]

def allocate_pred_to_speech_segments(prediction, speech_segments):
    pred_words=prediction.split(' ')
    n_words=len(pred_words)
    if n_words==0:
        return []
    segment_durations=np.diff(speech_segments)
    speech_duration=segment_durations.sum()
    segment_allocation=n_words*segment_durations/speech_duration
    words_per_segment=np.round(segment_allocation).T.astype(int)
[0]
    # If count is under then add missing word to longest segment
    words_per_segment[np.where(words_per_segment==\
                 words_per_segment.max())[0][0]] \
                 += n_words-words_per_segment.sum()
    word_segment_boundaries=np.cumsum(np.hstack([[0],\
                   words_per_segment]))
    segment_words=list(zip(speech_segments.tolist(),
          [pred_words[word_segment_boundaries[i]:\
                  word_segment_boundaries[i+1]]
           for i in range(len(words_per_segment))]))
    return align_segment_words(segment_words)
```

Features that were the most novel or unusual and/or led to the biggest improvements in system performance

The open source NVidia NeMo project claims that the Quartz 15x5 model is novel in the sense of using 10X fewer parameters than other models in the Encoder/Decoder class [8].

It did not train well in the time we had available and performed poorly. This may be the fault of over-extreme or poorly thought out augmentation choices that we made. We did not understand the model well enough to attempt any configuration changes to increase the learning capacity.

System configuration

Our system configuration was

- Intel core i9 processor
- 3TB SSD
- 64GB RAM
- NVidia RTX 2080TI GPU with 11GB of VRAM
- Ubuntu 20.04LTS operating system
- Python 3.7.9

Minimal required hardware specs to run your system

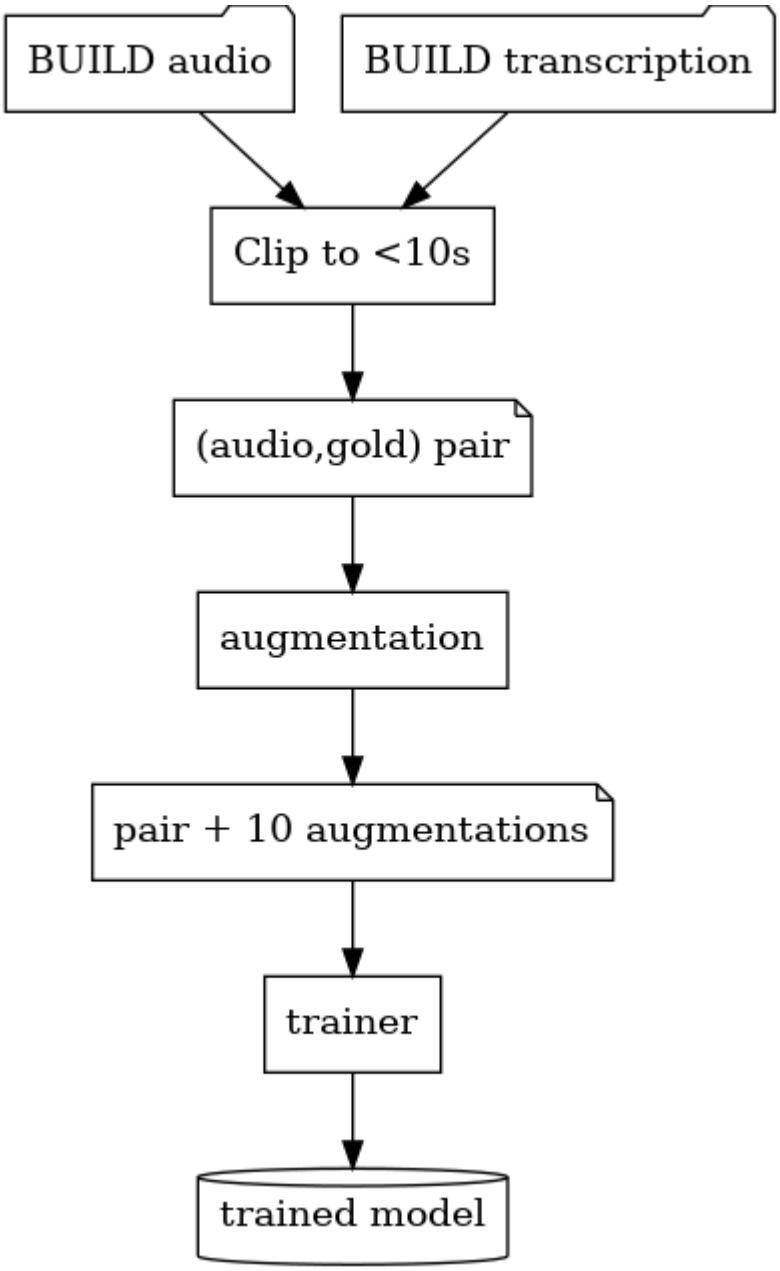
Evaluation required less than 1GB of GPU VRAM and less than 5GB of CPU RAM. Evaluation was fast, less than 5 minutes for a DEV or EVAL run.

Minimal required time and amount of data to train/tune your system

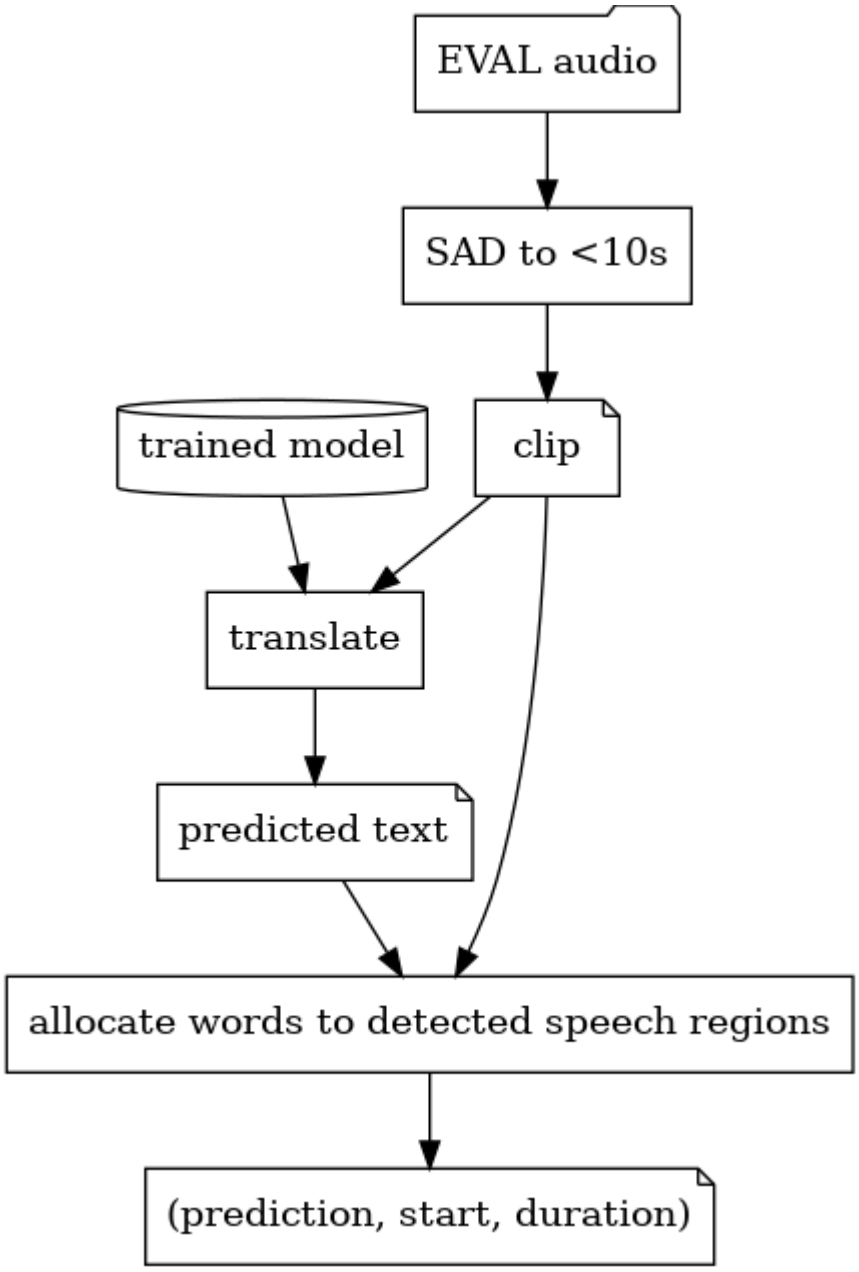
In training we were able to keep the GPU about 88% loaded and using 11.2 out of 11.6GB of available RAM. With the augmented data we were not converging to a training error loss of less than 50 with over 48 hours of training. This may be due to bad choice of augmentations, and also to upsampling to 16KHz when 8KHz was the source level, which may result in unnecessary artifacts. About 24GB of RAM was used during training. We allocated 8 cores for data loading. The 8 cores were periodically busy to 100% but in general stayed in a lower range.

Diagram giving a visual representation of our system's workflow

Training


BUILD audio
BUILD transcription
Clip to <10s
(audio,gold) pair
augmentation
pair + 10 augmentations
trainer
trained model

Evaluation



References

[1] https://github.com/NVIDIA/NeMo (https://github.com/NVIDIA/NeMo)

[2] https://github.com/NVIDIA/NeMo/blob/main/tutorials/asr/01_ASR_with_NeMo.ipynb (https://github.com/NVIDIA/NeMo/blob/main/tutorials/asr/01_ASR_with_NeMo.ipynb)

[3] https://arxiv.org/abs/1910.10261 (https://arxiv.org/abs/1910.10261)

[4] https://github.com/NVIDIA/NeMo/blob/main/examples/asr/conf/config (https://github.com/NVIDIA/NeMo/blob/main/examples/asr/conf/config)

[5] https://www.openslr.org/3/ (https://www.openslr.org/3/)

[6]
https://docs.nvidia.com/deeplearning/nemo/neural_mod_bp_guide/index.html#data_augmentation
(https://docs.nvidia.com/deeplearning/nemo/neural_mod_bp_guide/index.html#data_augmentatio

[7]
https://github.com/NVIDIA/NeMo/blob/main/tutorials/asr/05_Online_Noise_Augmentation.ipynb
(https://github.com/NVIDIA/NeMo/blob/main/tutorials/asr/05_Online_Noise_Augmentation.ipynb)

[8] https://arxiv.org/pdf/1910.10261.pdf (https://arxiv.org/pdf/1910.10261.pdf)