Catskills Research Company OpenASR21 Constrained condition application of NVidia NeMo ContextNet5x1 with subword tokenization model to Farsi and Somali

Lars Ericson

Quantitative Analytics Specialist

Catskills Research Company

1334 Hudson Place Davidson, NC 28036

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

11 November 2021

Abstract

We describe the Catskills Research Company system for NIST OpenASR21. In EVAL Constrained condition, this system scored a WER of 0.986509 for Farsi and 0.993126 for Somali. These are last-place scores for both conditions.

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 ContextNet 5x1 model [3] and BPE a/k/a SentencePiece tokenizer [5]. Vocabulary input to subword tokenizer was that observed in the BUILD transcriptions. Our model configuration was derived from a stock YAML file modified for the subword tokenization of the language [4]. For Farsi the size of the decoder layers was also doubled over stock. Final model for EVAL was selected from trained model checkpoints based on WER on the alphabetically first BUILD audio input file, using the NeMo built-in WER calculation [6]. For training we used the Novograd optimizer [8],[9]. We did many short separate batches of training with manual adjustment of initial learning rate from .01 to 1e-10. We found that initial rates closer to 0.001 were more effective but had to search manually for the right initial training rate to see improvement in error during the batch. We stopped training when we could find no learning rate that would result in continued improvement. Inference with time stamps was done on entire recordings using the frame-by-frame attribution approach [7].

Key modelling choices were:

- Size of layers in Jasper segment of encoder of ContextNext 5x1 pipeline
- Number of subword tokens
- Training sequence
- Model selection.

For Farsi we chose

- Layer sizes 256, 5x512, 2048
- 1024 subword tokens
- Model number 20079 out of our sequence of checkpoints

For Somali we chose

- Layer sizes 128, 5x2565, 1024
- 1024 subword tokens
- Model number 932 out of our sequence of checkpoints

For the training sequence, we

- Split 10 minute audio recordings into smaller segments corresponding to lines in the NIST annotation
- Initially trained the net progressively for segments of length 0.2 to 1 seconds, 0.2 to 1.5 seconds and so on up to 0.2 to 14 seconds, with decreasing batch sizes from 1024 down to whatever would fit in memory (lowest around 32)
- Finally trained on entire recordings with batch size 1

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`, `functools`, `glob`, `librosa`, `matplotlib`, `nemo`, `numpy`, `omegaconf`, `pandas`, `pickle`, `pyaudio`, `pydub`, `pytorch_lightning`, `soundfile`, `sqlite3`, `tarfile`, `torch`, `tqdm`, `unidecode`

Other data used (outside provided data)

Only NIST OpenASR21 BUILD samples were used for training.

Significant data pre-/post-processing

Data augmentation

We used spectral augmentation [9] in the training pipeline with configuration:

In [ ]:
```
  spec_augment:
    _target_: nemo.collections.asr.modules.SpectrogramAugmentation
    rect_freq: 50
    rect_masks: 5
    rect_time: 120
```

Speaker activity detection and translation

We operated on entire 10-minute recordings using the frame-by-frame approach [7] to time stamp subwords followed by knitting of subwords and coalescing of subword timestamps to get the timestamp for the whole word. This means we don't have to do Voice Activity Detection outside of the model or any kind of splitting of the input text into non-silent segments prior to transcription. Not having to do VAD or splitting is a major simplification. The function to obtain inference at frame-by-frame level is:

In [1]:
```
def get_prediction(asr_model, AUDIO_FILENAME):
    idx_to_label=np.array(list(asr_model.decoder.vocabulary) + [' '])
    files = [AUDIO_FILENAME]
    logits = asr_model.transcribe(files, logprobs=True)[0]
    clr()
    probs=softmax(logits)
    preds = np.argmax(probs, axis=1)
    decoded_prediction = []
    decoded_index = []
    blank_id=1024
    previous = blank_id
    for i, p in enumerate(preds):
        if (p != previous or previous == blank_id) \
           and p != blank_id:
            decoded_prediction.append(p)
            decoded_index.append(i)
        previous = p
    decoded_prediction=np.array(decoded_prediction)
    decoded_index=np.array(decoded_index)
    hypothesis_text='&'.join(idx_to_label[decoded_prediction]\
            ).replace('&##','').replace('&',' ')
    return preds, idx_to_label, decoded_prediction, \
            decoded_index, hypothesis_text
```

where

In [2]:
```
def softmax(logits):
    e = np.exp(logits - np.max(logits))
    return e / e.sum(axis=-1).reshape([logits.shape[0], 1])
```

The main function which also does the time stamping of words is:

In [5]:
```
def asr_with_timecode(asr_model, AUDIO_FILENAME, stm_dir,
                      shipping_dir, phase, model_fn,
                      language, debug = False):

    recording = AUDIO_FILENAME.split('/')[-1][0:-4]
    rec=recording.replace('_inLine', '').replace('_outLine', '')
    channel = '1' if '_inLine' in recording else '2'
    rec_chan=f'{rec}_{channel}'
    signal, sample_rate = librosa.load(AUDIO_FILENAME, sr=None)
    preds, idx_to_label, decoded_prediction, \
        decoded_index, hypothesis_text = \
        get_prediction(asr_model, AUDIO_FILENAME)
    starts=np.hstack([decoded_index*.01, [signal.shape[0]/16000]])
    start_end=np.vstack([starts[0:-1], starts[1:]]).T
    tokens=idx_to_label[decoded_prediction]
    n = len(tokens)
    final=[]
    i = 0
    while i < n:
        text1 = tokens[i]
        start1, end1 = start_end[i]
        if i < n-1:
            j=i+1
            while j < n:
                start2, end2 = start_end[j]
                text2 = tokens[j]
                if text2[0:2]=='##':
                    text1 = f'{text1}{text2[2:]}'
                    end1 = end2
                    j=j+1
                else:
                    break
            i = j
        else:
            i=i+1
        if start1 > end1:
            end1 = start1 + 0.02
        final.append([rec, channel, rec_chan, start1, end1, text1])

    stm_columns = ['recording', 'channel', 'rec_chan',
                   'start', 'end', 'text']
    df=pd.DataFrame(final, columns=stm_columns)
    stm_fn = f'{stm_dir}/{recording}.stm'
    df.to_csv(stm_fn, index=False, sep='\t', header=False)
    # print('saved', stm_fn)
    hyp =' '.join(df['text'].values).replace('(<hes>)','')

    if phase != 'eval':
        stms_fn=f'stms/{language}/{phase}/{recording}.stm'
        # print("reading", stms_fn)
        stm=pd.read_csv(stms_fn, delimiter='\t',
                        header=None, names=stm_columns).dropna()
        stm.columns=stm_columns
        gold=' '.join(stm['text'].values).replace('(<hes>)','')
        wer = word_error_rate(hypotheses=[hypothesis_text],
                              references=[gold])
```

```
        print(f"WER,{language},{phase},"
              "{round(wer*100,2):05.2f},{AUDIO_FILENAME},{model_fn}")

    if not debug:
        cmd = f"OpenASR_generate_ctm_file.py " \
               "-f {stm_fn} -o {shipping_dir}"
        os.system(cmd)
```

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

Inference worked well in BUILD for Farsi. We didn't have enough time to make it work for Somali. It didn't transfer as well to DEV. We have two interpretations of this:

- The BUILD and DEV vocabularies intersected only 50%
- There were multipled dialects and different dialects may pronounce the same word using different phonemes in kind and number
- For training purposes there were unequal sized samples of each dialect and in addition the DEV and BUILD samples may have different proportions of each dialect.

Here is a sample of first 5 WER scores alphabetically on file name for BUILD for Farsi with our model 20079:

phase	score
BUILD	30.07
BUILD	07.46
BUILD	24.64
BUILD	14.90
BUILD	39.28
BUILD	15.31

Here are the DEV scores on the first 5 DEV files using the same model:

phase	score
DEV	84.98
DEV	92.37
DEV	91.03
DEV	88.13
DEV	87.97
DEV	85.66

System configuration

Our system configuration was

- Intel core i9 processor
- 3TB SSD
- 64GB RAM
- NVidia RTX 2080TI GPU with 11GB of VRAM
- Ubuntu 21.10 operating system
- Python 3.9.6

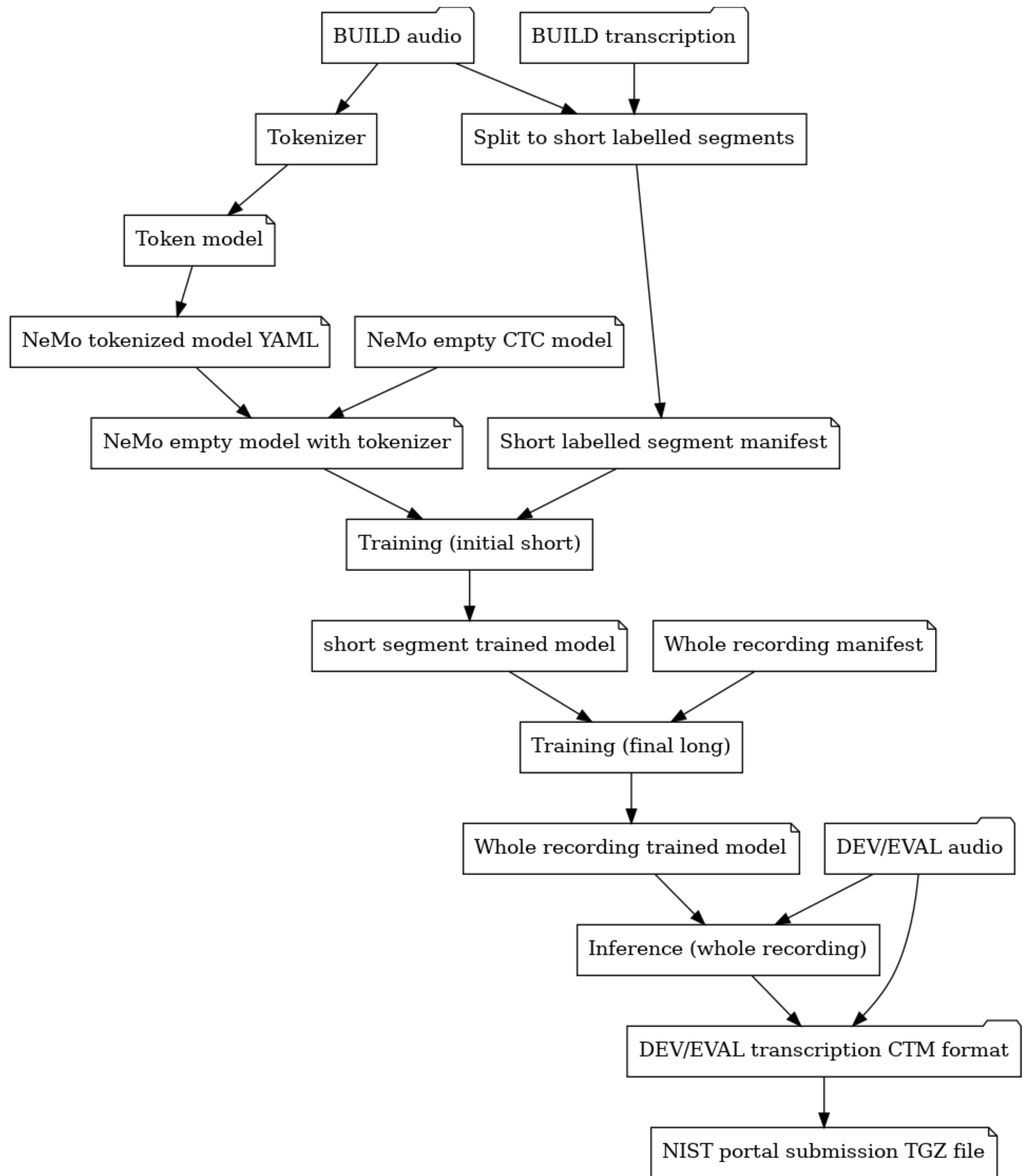
Minimal required hardware specs to run your system

Evaluation required less than 2GB 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

We trained for thousands of sessions under varying conditions of initial learning rate, network size, and size of training samples. We used a single GPU with 11GB of VRAM. We were constrained in our choices by the VRAM size. Approximately 200 hours of training was done for each language under varying conditions using the single GPU.

Diagram giving a visual representation of our system’s workflow



References

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

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

[3] https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/stable/asr/models.html#contextnet (https://docs.nvidia.com/deeplearning/nemo/user-guide/docs/en/stable/asr/models.html#contextnet)

[4]
https://github.com/NVIDIA/NeMo/blob/54e6f6ee688f09810d3e54661275fd5c8718db00/examples/
(https://github.com/NVIDIA/NeMo/blob/54e6f6ee688f09810d3e54661275fd5c8718db00/examples

[5] https://github.com/google/sentencepiece (https://github.com/google/sentencepiece)

[6]
https://github.com/NVIDIA/NeMo/blob/25c61f2e6f68af7fe90853b11bd073e6b2625c72/nemo/colle
(https://github.com/NVIDIA/NeMo/blob/25c61f2e6f68af7fe90853b11bd073e6b2625c72/nemo/colle

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

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

[9] https://github.com/NVIDIA/NeMo/blob/main/tutorials/asr/ASR_with_NeMo.ipynb
(https://github.com/NVIDIA/NeMo/blob/main/tutorials/asr/ASR_with_NeMo.ipynb)