음성인식 실습 (5/5)

WFST 기반의 디코딩 네트워크를 사용한 end-to-end 음성인식과 언어 모델 실습

김지환

서강대학교 컴퓨터공학과

Table of contents

실습 소개

- End-to-end에 WFST가 필요한 이유
  - end-to-end는 인간이 정한 '가정'들과의 전쟁

    1) HMM에 의해 정해진 가정들
    - 음성은 음소로 이루어진다. (음소의 수 n개)
    - 하나의 음소는 앞, 뒤 context를 통해 n^2종류로 세분화된다. (총 음소의 수 n^3개)
    - 세분화된 음소는 너무 많기 때문에 k-means algorithm을 통해 decision tree를 만들어 임의의 숫자로 clustering한다.
    - 결국 어떤 음소에 대하여, 음성 cluster의 생성 확률을 구한다. (ex> 'a' 음소는 어떤 벡터공간에 분포되어 있는가)

    - 가정 1: 음성인식을 위해서는 사전에 음소가 정의되어야 하며, 음소는 고정된 벡터 차원 위의 정규 분포로 표현된다.

실습 소개

- End-to-end에 WFST가 필요한 이유

  2) DNN에 의해 정해진 가정

  * 하나의 음소는 복수개의 프레임으로 이루어진다. (many-to-one 문제를 연속적으로 반복)
  * DNN은 복수개의 프레임을 받아 하나의 음소를 출력하는 과정을 반복하며, 이를 위해 HMM의 segmentation 결과를 필요로 한다.

  * 가정 2: DNN으로 학습된 음향 모델은 독립된 음소(출력)과 그에 대응하는 frame의 열(입력)의 반복으로 이루어짐

  3) WFST에 의해 정해진 가정

  * 음소의 열은 단어를 구성하고, 단어의 열은 문장을 구성한다.

  * 가정 3: 인식 가능한 음소와 단어는 사전 정의되어야 한다.

실습 소개

- End-to-end에 WFST가 필요한 이유
  - 3개의 가정은 음성인식 성능 향상을 위해 극복해야 할 과제

1) 음성인식을 위해서는 사전에 음소가 정의되어야 하며, 음소는 고정된 벡터 차원 위의 정규 분포로 표현된다.
   - 인식 가능한 음성의 최소단위는 음소가 맞는가?
   - 음성은 정규 분포가 맞는가?

2) 가정 2: DNN으로 학습된 음향 모델은 독립된 음소(출력)과 그에 대응하는 frame의 열(입력)의 반복으로 이루어짐
   - 음소 간의 관계성은 WFST를 통해 표현되며, 모델이 음소 간의 관계성을 표현하지 못함

3) 인식 가능한 음소와 단어는 사전 정의되어야 한다.
   - 최적화된 음소, 단어의 개수는 어떻게 결정되는가?

실습 소개

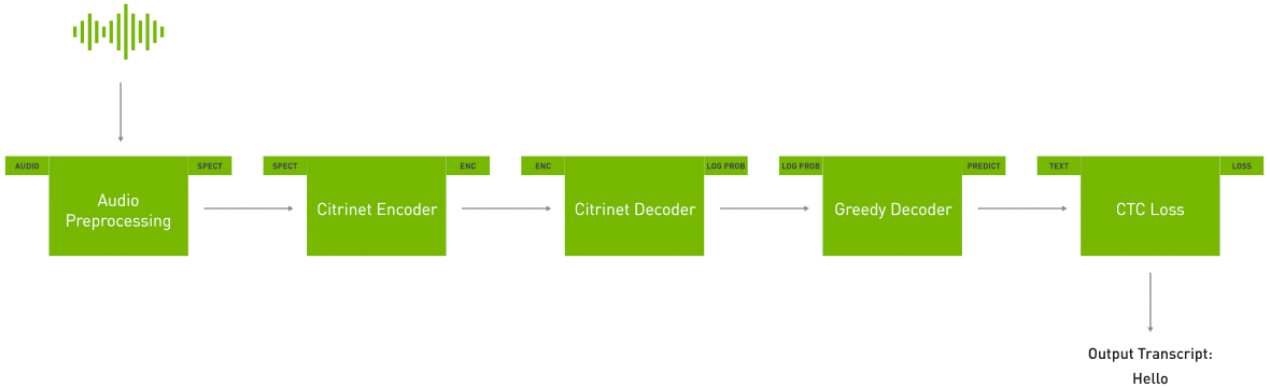
- End-to-end는 위 가정들을 극복하여 학습이 가능하기 때문에 높은 성능을 보임
  - 가정을 제거할 수 있었던 이유: 많은 데이터, 복잡한 모델
  - 가정 1,2를 극복한 모델
    - CTC, LF-MMI, AED, RNN-T, transformer 등 대부분을 이루는 모델
  - 가정 3을 극복한 모델
    - 없음(wav2vec2.0 일부 극복)
      - Wav2vec2.0은 self-supervised learning을 통해 target에 구애받지 않는 speech representation을 학습 가능하지만, fine-tuning을 통해서 단어를 결정해 주어야 하기 때문
  - 결국, end-to-end라 할지라도 output unit은 아직까지 사람이 지정해 주어야 함

실습 소개

- WFST가 잘하는 것
  - 지정된 unit들의 sequence를 가장 최적화해서 모델링할 수 있음
    - (입력, 출력이 정해져 있는 경우에 대해 가장 좋은 모델링 방법)
  - WFST의 효과
    - 빠른 처리 속도 (shortest path가 지정되어 있으므로)
- 높은 성능을 보이는 음향 모델인 end-to-end 모델에 대하여, WFST로 표현된 word set과 언어 모델을 결합해 성능을 높임

Nemo 소개

- NeMo (NVIDIA, 2019)
  - https://developer.nvidia.com/nvidia-nemo (homepage)
  - https://github.com/NVIDIA/NeMo (source)
  - NVIDIA NeMo™ is an open-source framework for developers to build and train state-of-the-art (SOTA) conversational AI models.
  - PyTorch, PyTorch Lightning을 기반으로 작성된 E2E toolkit
  - SOTA model들의 pretrained model을 제공
    - ASR pretrained models : https://catalog.ngc.nvidia.com/



Google colab 소개

- Google Colaboratory (Google)
  - https://colab.research.google.com/
  - 웹 브라우저에서 파이썬을 작성하고 실행할 수 있는 서비스
  - 클라우드 기반의 주피터 노트북 개발환경
  - 기본적으로 파이썬을 사용가능
    - Tensorflow, PyTorch, mataplotlib, scikit-learn, pandas 등의 ML/DL에 사용하는 라이브러리들을 기본적으로 지원
  - K80 GPU를 무료로 사용 가능
    - 사용량의 제한이 있으나 일반적으로 교육용으로 사용하기에는 문제 없음

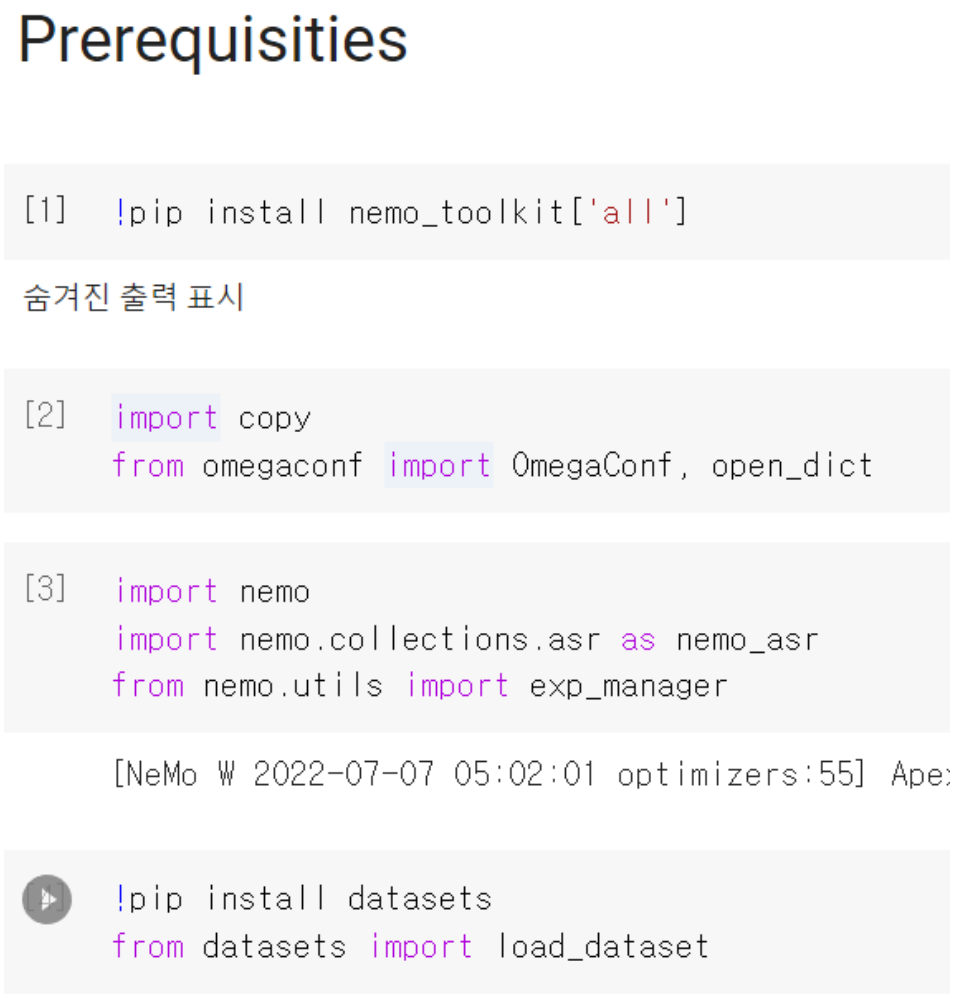
	Colab Free	Colab Pro	Colab Pro +
Guarantee of resources	Low	High	Even Higher
GPU	K80	K80, T4 and P100	K80, T4 and P100
RAM	16 GB	32 GB	52 GB
Runtime	12 hours	24 hours	24 hours
Background execution	No	No	Yes
Costs	Free	9.99$ per month	49.99$ per month
Target group	Casual user	Regular user	Heavy user

Nemo 설치 및 환경 세팅 (Colab 기준)

- Nemo 설치
  - PIP를 이용한 설치 (colab 권장)
    - $pip install nemo_toolkit['all']
  - Source code를 이용한 설치
    - $apt-get update && apt-get install -y libsndfile1 ffmpeg
    - $git clone https://github.com/NVIDIA/NeMo
    - $cd NeMo
    - $./reinstall.sh

Nemo 설치 및 환경 세팅 (Colab 기준)

- Nemo 설치 (실습)
  - Pip 명령어를 통해 nemo 설치
  - 주요 library import
    - omegaconf: yaml, json등의 configuration 파일을 읽고 쓸 수 있는 lib. Nemo에서 기본적으로 사용함
    - nemo.collections.asr: Nemo ASR class
    - nemo.utils.exp_manager: 학습 로그, conf등에 사용되는 lib
    - datasets.load_dataset: 학습 및 테스트 데이터 관리 lib로, huggingface에서 사용됨

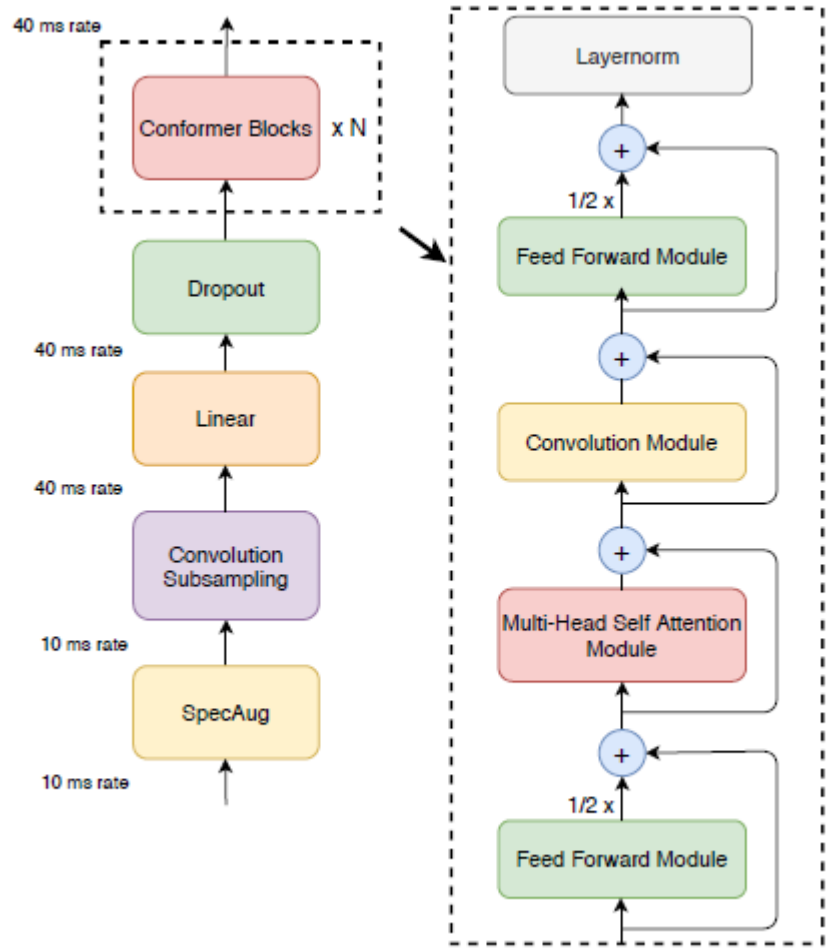


Nemo ASR 실습

- 실습 순서
  - 1) Pre-trained 모델 불러오기 (영어)
  - 2) Small data를 통한 모델 확인 (영어)
  - 3) Train/test 데이터 불러오기 및 데이터 확인 (한국어)
  - 4) 한국어 fine-tuning을 위한 모델 설정
  - 5) 한국어 output unit 설정 및 training 세팅
  - 6) 학습
  - 7) 테스트

Nemo ASR 실습

- 1) Pre-trained 모델 불러오기 (영어)
  - Nemo pretrained catalog에서 모델을 받아 사용 가능
    - https://catalog.ngc.nvidia.com/models
  - 본 실습에서는 Conformer을 사용함 (실습용)
    - Transformer block에 convolution module을 추가한 모델
    - 2020년 이후 계속해서 SOTA 성능을 보이고 있음
    - Librispeech test set 기준 2.7%의 word error rate (WER)을 보임



Gulati, A., Qin, J., Chiu, C.-C., Parmar, N., Zhang, Y., Yu, J., Han, W., Wang, S., Zhang, Z., Wu, Y., Pang, R. (2020) Conformer: Convolution-augmented Transformer for Speech Recognition. Proc. Interspeech 2020, 5036-5040, doi: 10.21437/Interspeech.2020-3015

Nemo ASR 실습

- 1) Pre-trained 모델 불러오기 (영어)
  - 모델 불러오기

```
import nemo.collections.asr as nemo_asr
```

```
asr_model = nemo_asr.models.EncDecCTCModelBPE.from_pretrained(model_name="stt_en_conformer_ctc_large_ls")
```

  - 다른 추천 ASR 모델
    - Citrinet, Contextnet, Quartznet 등

Nemo ASR 실습

- 2) Small data를 통한 모델 확인 (영어)
  - Huggingface의 librispeech test corpus를 불러와 사용
    - 모델 동작 확인용
  - 임의의 sample에 대해 재생 및 인식 과정 수행
  - 모델에 대한 음성인식
    - {model}.transcribe({listOfFilepath})를 통해 결과 확인 가능

Pre-trained model

```
[5]  char_model = nemo_asr.models.ASRModel.from_pretrained("stt_en_quartznet15x5", map_location='cpu')
```

숨겨진 출력 표시

```
[6]  ds = load_dataset("kresnik/librispeech_asr_test", "clean")
```

숨겨진 출력 표시

```
[7]  test_ds = ds['test']
     sample = test_ds[0]
     sample
```

숨겨진 출력 표시

```
[8]  import IPython

     IPython.display.Audio(sample['file'])
```

0:04 / 0:04

```
[9]  result = char_model.transcribe([sample['file']])
     results = char_model.transcribe(test_ds['file'][:10])
```

Transcribing: 100% 1/1 [00:12<00:00, 12.23s/it]

Transcribing: 100% 3/3 [00:11<00:00, 3.22s/it]

```
print("Hypothesis: "+ result[0])
print("Reference:  " +sample['text'].lower())
```

```
Hypothesis:  it is sixteen years since john bergson died
Reference:   it is sixteen years since john bergson died
```

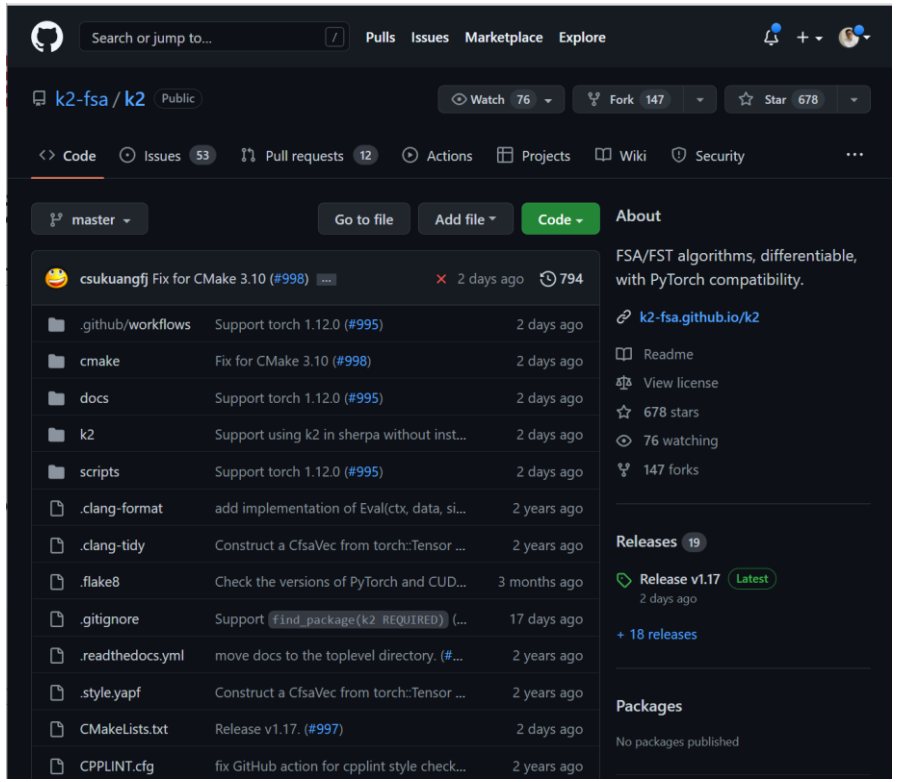
K2 소개

- k2-fsa (k2)
  - OpenFST를 개량한 python 기반의 WFST 패키지
  - End-to-end 모델과 WFST를 결합할 목적으로 제작됨
  - FSA, FST를 tensor 형태로 구현하여 GPU에서 WFST 연산을 가능하게 함
    - Pytorch에서 동작 가능
  - Open-source: https://github.com/k2-fsa/k2

  - Kaldi 이후 세대의 ASR toolkit 중, WFST를 담당하는 toolkit
    - 등산 시리즈 (가칭)
      - Lhotse: data preparation, feature extraction
      - K2-fsa: WFST 구현
      - Icefall: lhotse와 k2, pytorch를 이용한 ASR recipe 제공
      - Sherpa: ASR streaming server
    - Icefall에서 제공하는 recipe는 state-of-the-art 성능을 보이지 않아서, 실습에서는 Nvidia Nemo를 사용

K2 설치 및 기본 환경 세팅

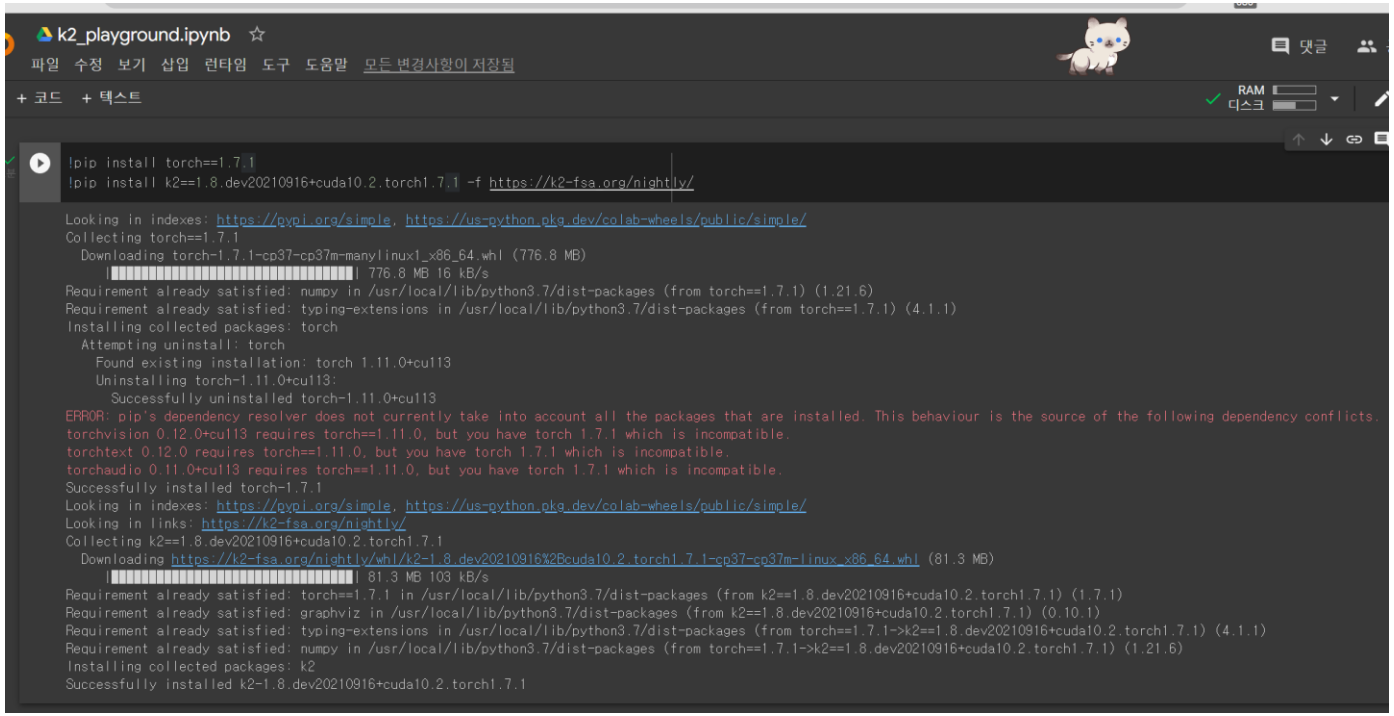
- K2 설치 방법
  - Prerequisities (pip 기준)
    - Python >= 3.6
    - CUDA >= 10.1
    - PyTorch == 1.7.1 (conda 설치의 경우 >=1.7.1)
  - PyPI (PIP)를 이용한 설치 방법 (권장, but colab에선 비권장)
    - $pip install k2
      - Prerequisities 자동으로 설치됨
  - Colab에서 돌아가는 pip 설치 방법
    - $pip install torch==1.7.1
    - ~~$pip install k2==1.8.dev20210916+cuda10.2.torch1.7.1 -f https://k2-fsa.org/nightly/~~
    - $pip install k2==1.17.dev20220710+cuda10.2.torch1.7.1 -f https://k2-fsa.org/nightly/
  - Source code (github)을 이용한 설치 방법
    - $git clone https://github.com/k2-fsa/k2.git
    - $cd k2
    - $python3 setup.py install



K2 설치 및 기본 환경 세팅

- K2 설치 실습 (Google colab)
  - 1) 새 colab notebook 생성

  - 2) 새 colab notebook 생성



K2 설치 및 기본 환경 세팅

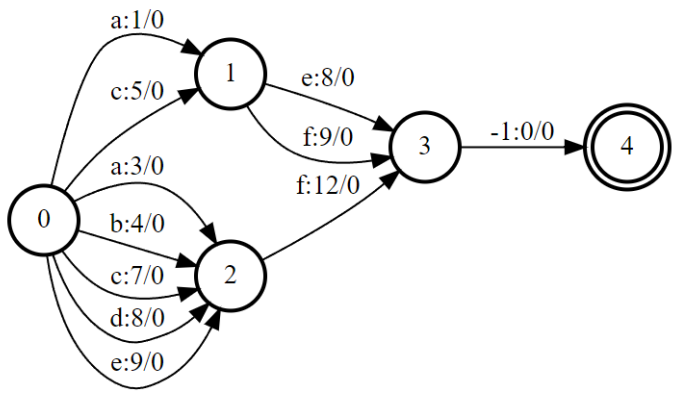
- K2 설치 실습 (Google colab)
  - 3) 설치 확인

```
[2]  import k2, torch

[3]  import k2.version
     k2.version.version.main()

     Collecting environment information...

     k2 version: 1.8
     Build type: Release
     Git SHA1: 646704e142438bcd1aaf4a6e32d95e5ccd93a174
     Git date: Thu Sep 16 13:05:12 2021
     Cuda used to build k2: 10.2
     cuDNN used to build k2: 8.0.2
     Python version used to build k2: 3.7
     OS used to build k2: Ubuntu 18.04.5 LTS
     CMake version: 3.21.2
     GCC version: 7.5.0
     CMAKE_CUDA_FLAGS:  --expt-extended-lambda -gencode arch=compute_35,code=sm_35 -
     CMAKE_CXX_FLAGS:  -D_GLIBCXX_USE_CXX11_ABI=0 -Wno-strict-overflow
     PyTorch version used to build k2: 1.7.1
     PyTorch is using Cuda: 10.2
     NVTX enabled: True
     With CUDA: True
     Disable debug: True
     Sync kernels : False
     Disable checks: False
```

K2를 이용한 WFST 구현

- 기본 사용법
  - 표현법
    - Openfst와 동일함
      - WFSA: StartState \t EndState \t Symbol \t Weight
    - Symbol은 기본적으로 index로 표현/계산됨.
      - 시각화할 일이 있을 때만 symbol을 표현.
      - Symbol table 정의가 필요함.
  - Weighted Finite-state Acceptor (WFSA)
    - End-to-end 음성인식의 출력 및 최종 결과 생성을 위해 사용
    - 일반적으로 k2.Fsa.from_str()를 사용하여 생성함
    - Openfst의 결과를 이용할 경우, k2.Fsa.from_openfst()를 사용
    - draw() or to_dot() methods를 이용하여 시각화

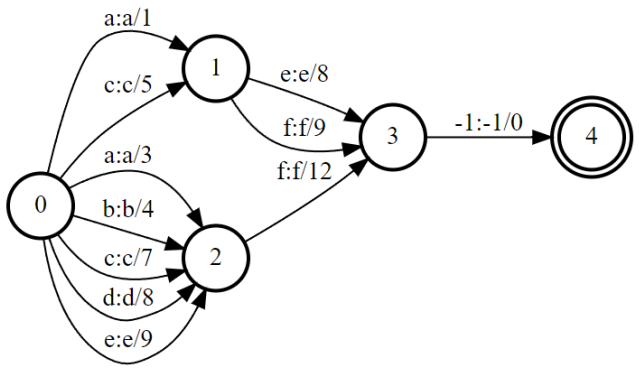
```
s = '''
0 1 1 1
0 1 3 5
0 2 1 3
0 2 2 4
0 2 3 7
0 2 4 8
0 2 5 9
1 3 6 9
1 3 5 8
2 3 6 12
3 4 -1 0
4
'''

a_fsa = k2.Fsa.from_str(s)
sym_str = '''
 <eps> 0
 a 1
 b 2
 c 3
 d 4
 e 5
 f 6
'''
a_fsa.symbols = k2.SymbolTable.from_str(sym_str)
a_fsa.labels_sym = k2.SymbolTable.from_str(sym_str)

a_fsa = k2.arc_sort(a_fsa)
a_fsa.draw('a_fsa.svg')
```



K2를 이용한 WFST 구현

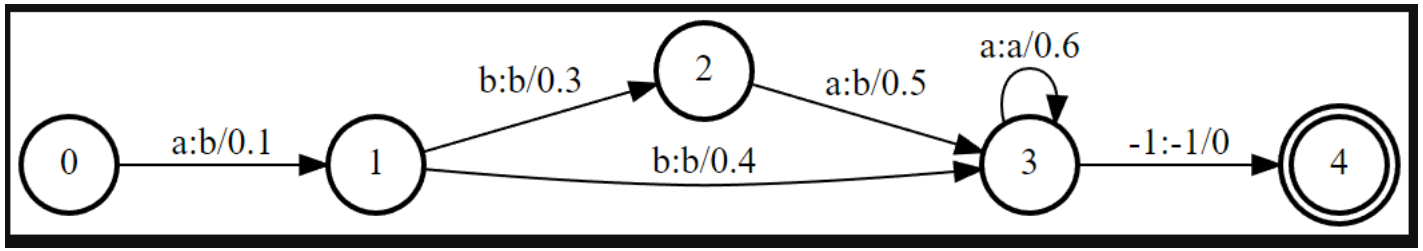
- 기본 사용법
  - 표현법
    - Openfst와 동일함
      - WFST: StartState \t EndState \t InputSymbol \t OutputSymbol\t Weight
    - 시각화된 표현은 arc당 InputSymbol:OutputSymbol/weight
  - Weighted Finite-state Transducer (WFST)
    - CTC, lexicon, language model의 최적화된 표현을 위해 사용함
    - k2.Fsa.from_str의 acceptor=False option을 이용하여 생성함

```
s = '''
0 1 1 1 1
0 1 3 3 5
0 2 1 1 3
0 2 2 2 4
0 2 3 3 7
0 2 4 4 8
0 2 5 5 9
1 3 6 6 9
1 3 5 5 8
2 3 6 6 12
3 4 -1 -1 0
4
'''

#a_fsa = k2.Fsa.from_str(s)
a_fst = k2.Fsa.from_str(s, acceptor=False)
sym_str = '''
 <eps> 0
 a 1
 b 2
 c 3
 d 4
 e 5
 f 6
'''
#a_fsa.symbols = k2.SymbolTable.from_str(sym_str)
#a_fsa.labels_sym = k2.SymbolTable.from_str(sym_str)
a_fst.symbols = k2.SymbolTable.from_str(sym_str)
a_fst.labels_sym = k2.SymbolTable.from_str(sym_str)
a_fst.aux_labels_sym = k2.SymbolTable.from_str(sym_str)
a_fst = k2.arc_sort(a_fst)
a_fst.draw('a_fst.svg')
```



K2를 이용한 WFST 구현

- 기본 사용법 (실습)
  - WFST 생성 실습



  - 1) Symbol table 정의
    - <eps>는 k2 내부로 -1로 정의되어 있지만, 표기 편의를 위해 1부터 symbol을 설정함

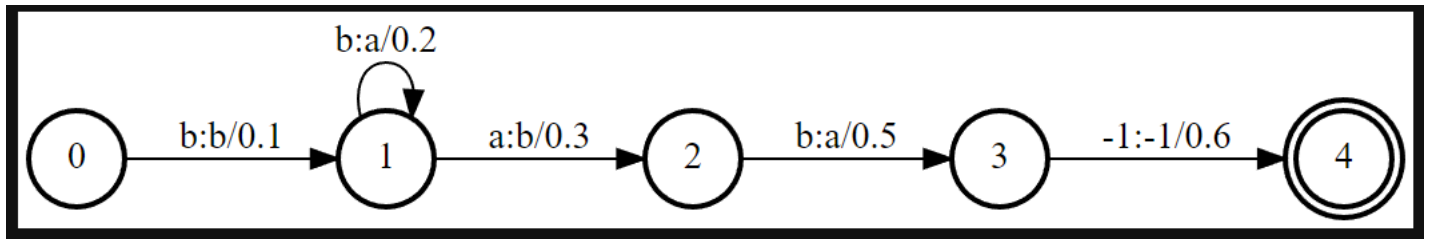
```
sym_str = '''
 <eps> 0
 a 1
 b 2
'''
```

  - 2) Arcs 정의
    - K2는 종료 state를 별도로 두도록 되어 있어, eps symbol을 사용해 종료 state를 정의함

```
s = '''
0 1 1 2 0.1
1 2 2 2 0.3
1 3 2 2 0.4
2 3 1 2 0.5
3 3 1 1 0.6
3 4 -1 -1 0
4
'''
```

K2를 이용한 WFST 구현

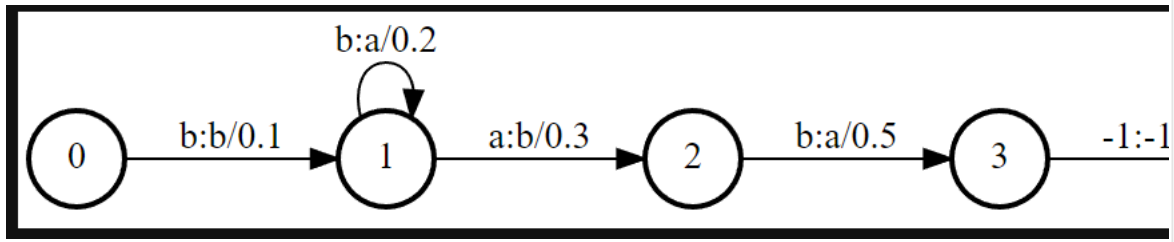
- 기본 사용법 (실습)
  - WFST 생성 실습



  - 1) Symbol table 정의
    - <eps>는 k2 내부로 -1로 정의되어 있지만, 표기 편의를 위해 1부터 symbol을 설정함
  - 2) Arcs 정의
    - K2는 종료 state를 별도로 두도록 되어 있어, eps symbol을 사용해 종료 state를 정의함

K2를 이용한 WFST 구

- 기본 사용법 (실습)
  - WFST 생성 실습




```
import k2
```

```
s = '''
0 1 1 2 0.1
1 2 2 2 0.3
1 3 2 2 0.4
2 3 1 2 0.5
3 3 1 1 0.6
3 4 -1 -1 0
4
'''
a_fsa = k2.Fsa.from_str(s, acceptor=False)
sym_str = '''
 <eps> 0
 a 1
 b 2
'''
a_fsa.symbols = k2.SymbolTable.from_str(sym_str)
a_fsa.labels_sym = k2.SymbolTable.from_str(sym_str)
a_fsa.aux_labels_sym = k2.SymbolTable.from_str(sym_str)
a_fsa = k2.arc_sort(a_fsa)
a_fsa.draw('fsa_symbols.svg')
```

  - 1) Symbol table 정의
    - <eps>는 k2 내부로 -1로 정의되어 있지만, 표기 편의를 위해 1부터 symbol을 설

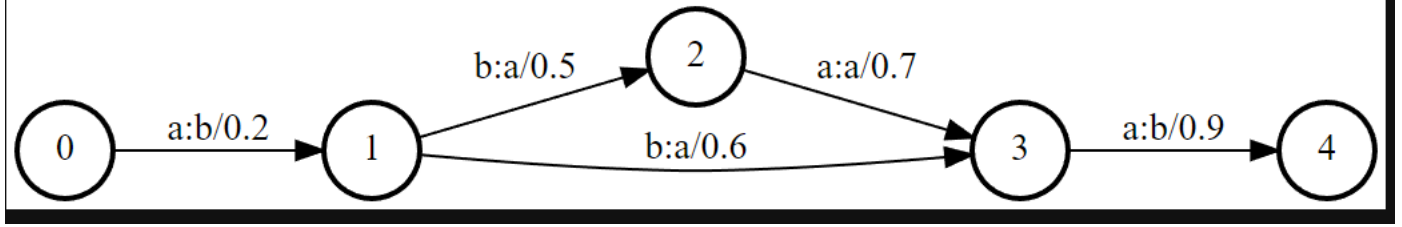
```
s = '''
0 1 2 2 0.1
1 1 2 1 0.2
1 2 1 2 0.3
2 3 2 1 0.5
3 4 -1 -1 0.6
4
'''
b_fsa = k2.Fsa.from_str(s, acceptor=False)
sym_str = '''
 <eps> 0
 a 1
 b 2
'''
b_fsa.symbols = k2.SymbolTable.from_str(sym_str)
b_fsa.labels_sym = k2.SymbolTable.from_str(sym_str)
b_fsa.aux_labels_sym = k2.SymbolTable.from_str(sym_str)
b_fsa = k2.arc_sort(b_fsa)
b_fsa.draw('fsa_symbols.svg')
```

  - 2) Arcs 정의
    - K2는 종료 state를 별도로 두도록 되어 있어, eps symbol을 사용해 종료 state를

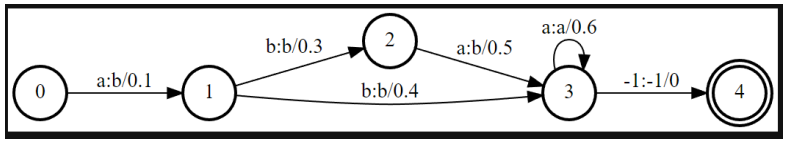
K2를 이용한 WFST 구현

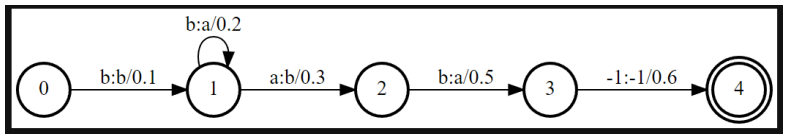
- 기본 사용법 (실습)
  - Composition 연산

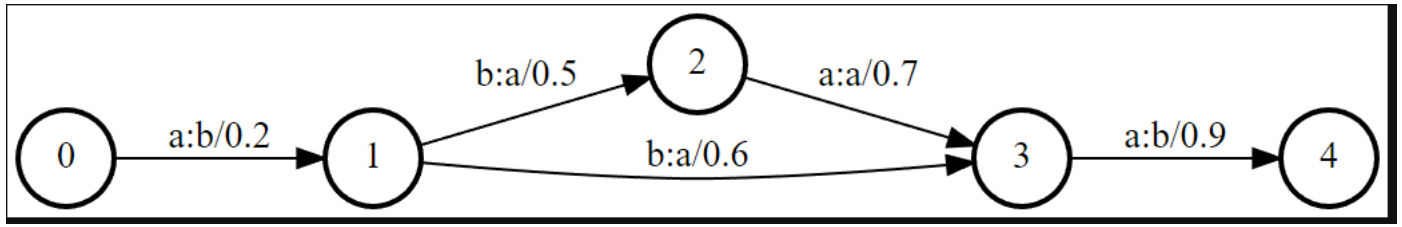
```
ab = k2.compose(a_fsa, b_fsa)
ab.draw('fsa_composed.svg')
```



  - K2 내장 method k2.compose()를 사용하여 연산 가능함

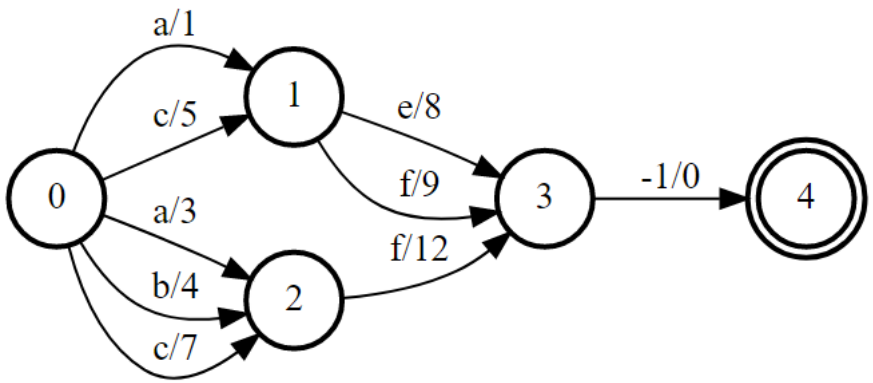


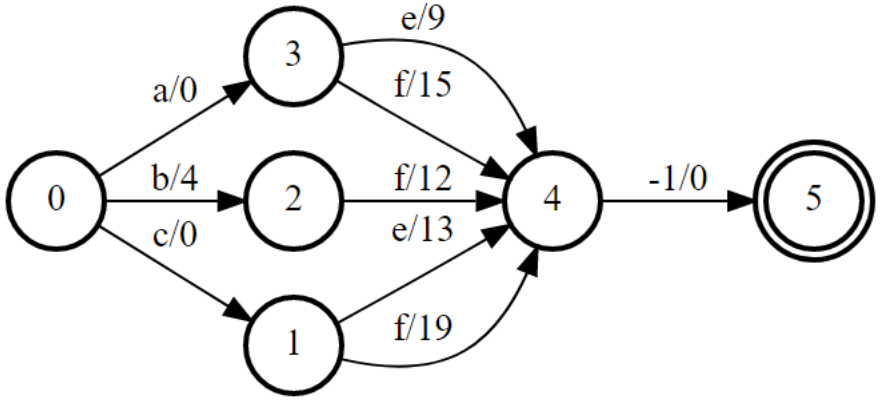




K2를 이용한 WFST 구현

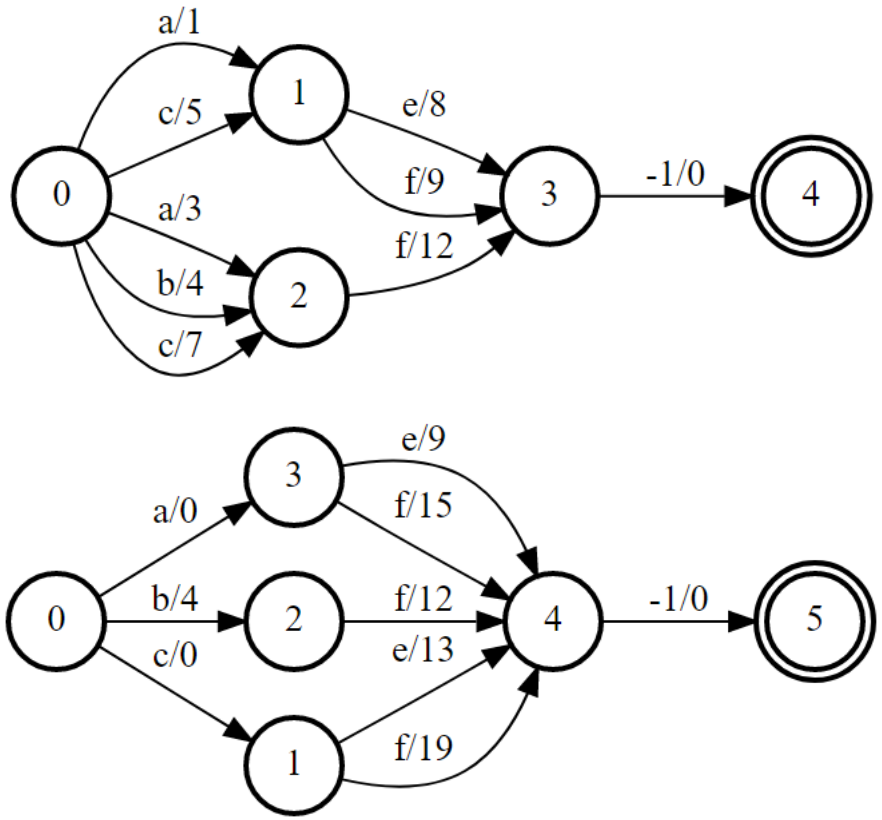
- 기본 사용법
  - Determinization 연산
    - 'af ', 'cf '에 대해 non-deterministic이 발생함
    - 디코딩 최적화를 위해 꼭 필요한 operation





K2를 이용한 WFST 구현

- 기본 사용법 (실습)
  - Determinization 연산




```
s = '''
0 1 1 1
0 1 3 5
0 2 1 3
0 2 2 4
0 2 3 7
1 3 6 9
1 3 5 8
2 3 6 12
3 4 -1 0
4
'''
a_fsa = k2.Fsa.from_str(s)
sym_str = '''
 <eps> 0
 a 1
 b 2
 c 3
 d 4
 e 5
 f 6
'''
a_fsa.symbols = k2.SymbolTable.from_str(sym_str)
a_fsa.labels_sym = k2.SymbolTable.from_str(sym_str)
a_fsa = k2.arc_sort(a_fsa)
a_fsa.draw('fsa_symbols.svg')
```

```
a_fsa_deter = k2.determinize(a_fsa)
```

```
a_fsa_deter = k2.arc_sort(a_fsa_deter)
a_fsa_deter.labels_sym = k2.SymbolTable.from_str(sym_str)
a_fsa_deter.draw('deter.svg')
```

음성인식을 위한 WFST 구현

- 개요
  - 음성인식 decoding을 위해 필요한 것
    - U (utterance) WFSA: End-to-end 모델로부터 생성 (본 실습에서는 CTC 기반의 모델)
    - C (CTC) WFST: CTC output을 subword로 collapsing할 수 있는 transducer
    - L (Lexicon) WFST: subword를 word로 바꿔주는 trasducer
    - G (Grammar) WFST: 언어 모델을 사용하여 scoring하는 transducer
  - 각 transducer의 구현 및 composition & determinization을 통해 디코딩 네트워크 구성

음성인식을 위한 WFST 구현

- 실습 순서
  - 1) U, C, L, G를 만들기 위한 data preparation (Nemo)
  - 2) C WFST 만들기
  - 3) L WFST 만들기
  - 4) G WFST 만들기
  - 5) LG composition&determinization, CLG composition&determinization
  - 6) U WFSA 만들기
  - 7) Lattice 생성 및 음성인식 결과 확인

음성인식을 위한 WFST 구현

- 1) U, C, T, G를 만들기 위한 data preparation (Nemo)
  - End-to-end 모델의 word 정보, token 정보, test 데이터에 대한 음성인식 결과 확률 분포가 필요함
  - 언어 모델, word 정보
    - 별도 제공
    - 출처: http://openslr.org/11/
  - Nemo 모델을 불러와 정보를 추출

```
import nemo.collections.asr as nemo_asr
```

```
asr_model = nemo_asr.models.EncDecCTCModelBPE.from_pretrained(model_name="stt_en_conformer_ctc_large_ls")
```

음성인식을 위한 WFST 구현

- 1) U, C, T, G를 만들기 위한 data preparation (Nemo)
  - Token (subword)와 word 정보를 얻기 위한 tokenizer loading

```
tokenizer = asr_model.tokenizer.tokenizer
print(tokenizer.encode_as_pieces('hello world'))
```

```
['_he', 'll', 'o', '_w', 'or', 'l', 'd']
```

  - Token 추출

```
vocab = asr_model.tokenizer.vocab
print(vocab)
```

```
['<unk>', 'e', 's', '_', 't', 'a', 'o', 'i', '_the'
```

```
with open('tokens.txt', 'w') as f:
  for k, v in enumerate(vocab):
    f.write(str(v) + ' '+str(k)+ '₩n')
```

  - Words 추출

```
with open('word.raw', 'r') as f:
  wlist = f.read().splitlines()
```

```
len(wlist)
```

```
976754
```

```
with open('words.txt', 'w') as f:
  for k, v, in enumerate(wlist):
    f.write(str(v)+' '+str(k)+'₩n')
```

음성인식을 위한 WFST 구현

- 1) U, C, L, G를 만들기 위한 data preparation (Nemo)
  - Word 와 subword의 관계를 정리한 lexicon 만들기

```
pieces = []
for i in wlist:
  pieces.append(tokenizer.encode_as_pieces(i))
```

```
lexicon = list(zip(wlist, pieces))
```

```
with open('lexicon.txt', "w", encoding="utf-8") as f:
  for word, tokens in lexicon[1:-1]: # special symbol removal
    f.write(f"{word} {' '.join(tokens)}₩n")
```

```
COHNAH _co h n a h
COHNAN'S _co h n an ' s
COHNER _co h n er
COHNFELD _co h n f e l d
COHNFELD'S _co h n f e l d ' s
COHNFIEL _co h n f i e l
COHNHEIM _co h n h e i m
COHNINGSBY _co h n ing s b y
COHNS _co h n s
COHO _co h o
```

음성인식을 위한 WFST 구현

- 1) U, C, L, G를 만들기 위한 data preparation (Nemo)
  - U FSA를 만들기 위한 output probabilities 추출
  - LibriSpeech test set에 대해 모델의 output 추출
    - Huggingface에서 test set을 받아와 모델로부터 결과 추출

```
!pip install datasets
from datasets import load_dataset
ds = load_dataset("kresnik/librispeech_asr_test", "clean")
test_ds = ds['test']
fl = test_ds['file']
```

```
r = asr_model.transcribe(fl, logprobs=True)
```

```
Transcribing: 0%                                                    0/655 [00:00<?, ?it/s]
[NeMo W 2022-07-13 08:29:37 nemo_logging:349] /usr/local/lib/python3.7/dis
      warnings.warn('User provided device_type of ₩'cuda₩', but CUDA is no
```

음성인식을 위한 WFST 구현

- 1) U, C, L, G를 만들기 위한 data preparation (Nemo)
  - 'logits.pt' 이름으로 output probabilites를 저장함
  - Output probabilities 설명
    - Shape: [2620, T, 129]
    - 2620개의 test file에 대해 길이가 T이고
    - Output unit이 129(128+blank)인 tensor

```
import torch

out_list = []
for i in r:
    out_list.append(torch.tensor(i))
```

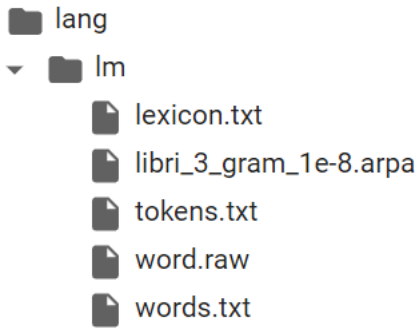
```
out_list
```

```
[tensor([[-68.8626, -23.0890, -24.1295,  ..., -27.1258, -29.3522,   0.0000],
         [-70.4840, -25.3311, -24.9210,  ..., -28.4772, -28.4988,   0.0000],
         [-71.2156, -26.0197, -27.0724,  ..., -31.1981, -29.9467,   0.0000],
         ...,
         [-77.4005, -26.6909, -25.8251,  ..., -34.9537, -33.7952,   0.0000],
         [-77.6541, -28.4947, -27.5985,  ..., -35.2744, -34.1950,   0.0000],
         [-73.8650, -28.6409, -28.6736,  ..., -33.5648, -33.0858,   0.0000]]),
 tensor([[-6.8194e+01, -2.2865e+01, -2.4683e+01,  ..., -2.8076e+01,
          -3.5040e+01,  0.0000e+00],
         [-6.8657e+01, -2.3835e+01, -2.3681e+01,  ..., -2.9930e+01,
          -3.3698e+01,  0.0000e+00],
         [-6.8468e+01, -2.2803e+01, -2.4785e+01,  ..., -3.2392e+01,
          -3.3759e+01,  0.0000e+00],
         ...,
         [-6.6545e+01, -2.6771e+01, -2.7661e+01,  ..., -4.0800e+01,
          -3.5352e+01, -7.1526e-07],
         [-6.7504e+01, -2.6293e+01, -2.9085e+01,  ..., -3.9639e+01,
          -3.6268e+01, -1.1921e-07],
         [-6.4749e+01, -2.3709e+01, -2.7339e+01,  ..., -3.5628e+01,
```

```
torch.save(out_list, 'logits.pt')
```

음성인식을 위한 WFST 구현

- 2) C FST 만들기
  - Nemo에서 만들어진 준비물



  - Word정보와 token 정보를 symbol table 형식으로 불러옴

```
words_values = k2.SymbolTable.from_file('lang/lm/words.txt')
tokens_values = k2.SymbolTable.from_file('lang/lm/tokens.txt')
```

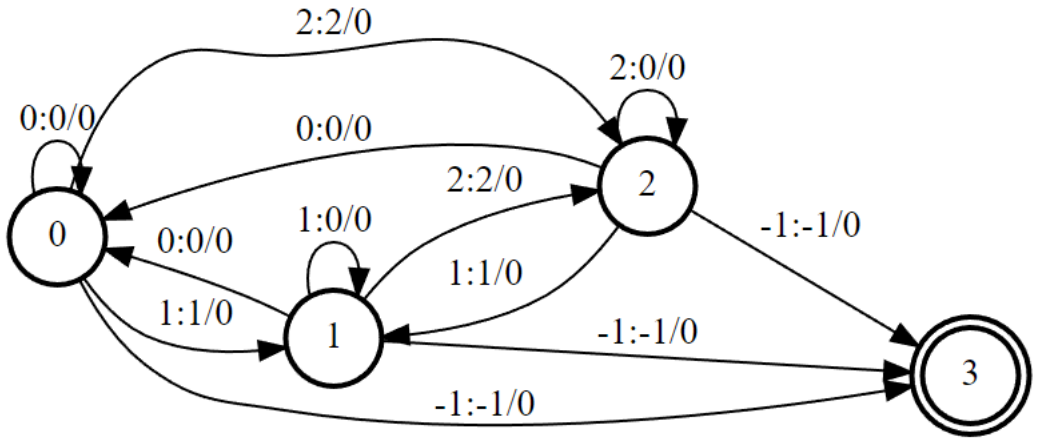
음성인식을 위한 WFST 구현

- 2) C FST 만들기
  - CTC transducer의 경우 k2의 내장 함수를 이용해서 쉽게 구현할 수 있음

```
C = k2.ctc_topo(max_token = 129, modified=False)
```

  - 2개 token으로 단순화한 CTC transducer 예시(0이 blank)

```
import k2
C_draw = k2.ctc_topo(max_token=2)
C_draw.draw('asdf.svg')
```



음성인식을 위한 WFST 구현

- 3) L FST 만들기
  - 제공된 read_lexicon() method를 이용하여 lexicon을 불러옴
  - Composition을 위한 disambig symbols 추가
  - 추가된 disambig. Symbol을 token_values에 추가 후 확인

```
words_values = k2.SymbolTable.from_file('lang/lm/words.txt')
tokens_values = k2.SymbolTable.from_file('lang/lm/tokens.txt')
```

```
C = k2.ctc_topo(max_token=129, modified=False)
```

```
from utils import read_lexicon

lexicon = read_lexicon("lang/lm/lexicon.txt")
```

```
from utils import add_disambig_symbols

lexicon_disambig, max_disambig = add_disambig_symbols(lexicon)
```

```
max_disambig
```

```
1
```

```
tokens_values.add('#0', 128)
tokens_values.add('#1', 129)

print(tokens_values.get('#0'), tokens_values.get('#1'))
```

```
128 129
```

음성인식을 위한 WFST 구현

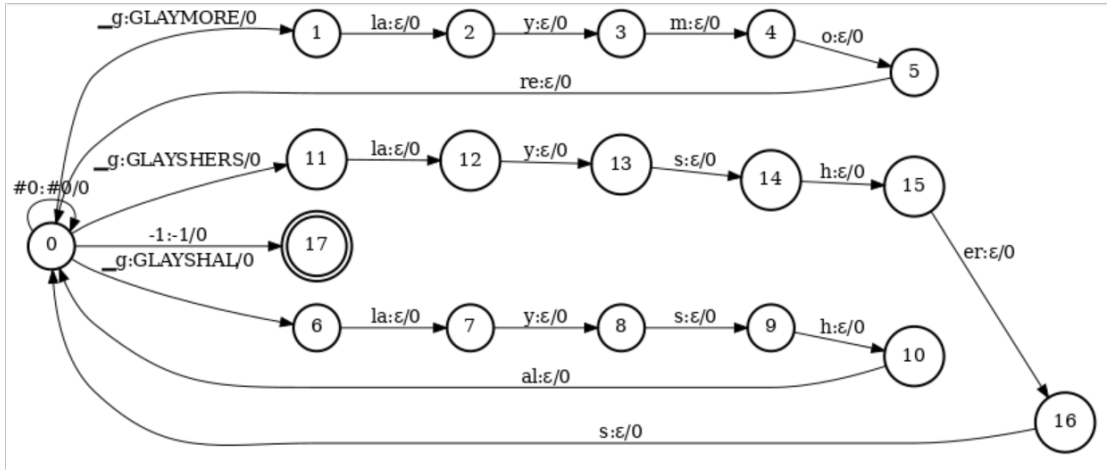
- 3) L FST 만들기
  - lexicon_to_fst() method를 이용하여 lexicon을 fst로 변경

```
from utils import lexicon_to_fst

L = lexicon_to_fst(
    lexicon_disambig,
    token2id=tokens_values._sym2id,
    word2id=words_values._sym2id,
    need_self_loops=True
)
```

음성인식을 위한 WFST 구현

- 3) L FST 만들기
  - 시각화 확인을 위해 일부 lexicon을 이용해 fst를 구성
    - Lexicon.txt의 일부를 잘라 lexicon_mini.txt를 구성함 (아래 예시)

```
GLAYMORE _g la y m o re
GLAYSHAL _g la y s h al
GLAYSHERS _g la y s h er s
```




```
from lexicon import read_lexicon

lexicon = read_lexicon("lang/lm/lexicon_mini.txt")
```

```
from utils import add_disambig_symbols

lexicon_disambig, max_disambig = add_disambig_symbols(lexicon)
```

```
max_disambig
```

```
0
```

```
from utils import lexicon_to_fst

L = lexicon_to_fst(
    lexicon_disambig,
    token2id=tokens_values._sym2id,
    word2id=words_values._sym2id,
    need_self_loops=True
)
```

```
L.labels_sym=tokens_values
L.aux_labels_sym=words_values
```

```
re=L.draw('asdf.svg')
```

```
re.format = 'png'
re.render(filename='/home/hosung/works/nemo_nb/s1/asdf')
```

음성인식을 위한 WFST 구현

- 4) G FST 만들기
  - Arpa format을 openFST 스타일의 fst로 변환할 수 있는 kaldilm library 설치 및 실행
  - OpenFST 스타일의 fst.txt를 불러와서 k2스타일로 저장

```
!pip install kaldilm
!python3 -m kaldilm --read-symbol-table="lang/lm/words.txt" --disambig-symbol='#0' --max-order=3 lang/lm/libri_3_gram_1e-8.arpa > lang/G.fst.txt
```

```
import torch

with open("lang/G.fst.txt") as f:
    G = k2.Fsa.from_openfst(f.read(), acceptor=False)
    torch.save(G.as_dict(), "lang/G.pt")
```

음성인식을 위한 WFST 구현

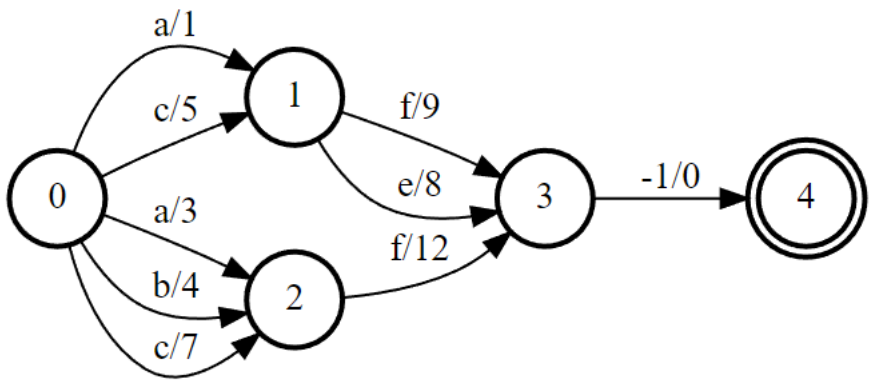
- 5) LG composition&determinization, CLG composition&determinization
  - WFST간 통합 최적화 순서
    - 1) Arc sorting: arc의 배치 순서를 index 순서대로 조정하여 연산 최적화
    - 2) Composition
    - 3) Connect: composition 이후, final state에 도달하지 못하는 arc를 삭제
    - 4) Determinization
    - 5) epsilon removal: composition 과정에 필요했던 epsilon symbol들을 제거

음성인식을 위한 WFST 구현

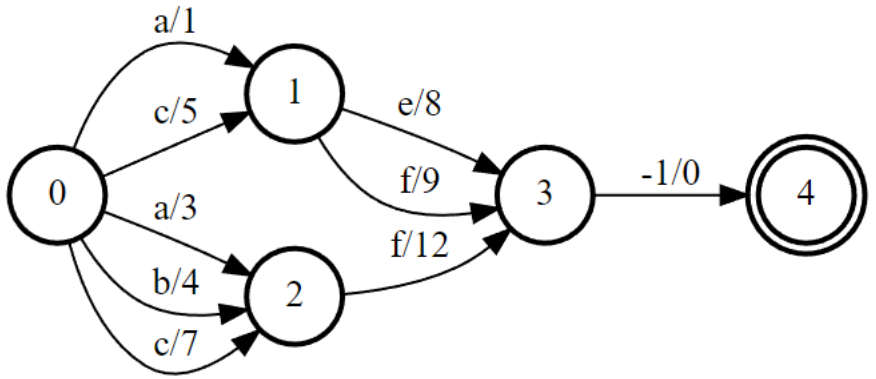
- 5) LG composition&determinization, CLG composition&determinization
  - arc_sort() 를 통해 arc의 배치 순서를 index 순서대로 조정함
    - Composition & determinization의 연산 최적화를 위함

```
L = k2.arc_sort(L)
G = k2.arc_sort(G)

LG = k2.compose(L, G)
#L_inv = L.invert()
#L_inv = k2.arc_sort(L_inv)
#L_inv.rename_tensor_attribute_('aux_labels', 'left_labels')
#LG = k2.intersect(L_inv, G, treat_epsilons_specially=True)
#LG.rename_tensor_attribute_('left_labels', 'labels')
LG = k2.connect(LG)

print(LG.shape)
```

```
LG = k2.determinize(LG)
LG = k2.connect(LG)
print(LG.shape)
```




```
a_fsa = k2.arc_sort(a_fsa)
a_fsa.draw('fsa_symbols.svg')
```



음성인식을 위한 WFST 구현

- 5) LG composition&determinization, CLG composition&determinization
  - Compose() and determinize()

```
L = k2.arc_sort(L)
G = k2.arc_sort(G)

LG = k2.compose(L, G)
#L_inv = L.invert()
#L_inv = k2.arc_sort(L_inv)
#L_inv.rename_tensor_attribute_('aux_labels', 'left_labels')
#LG = k2.intersect(L_inv, G, treat_epsilons_specially=True)
#LG.rename_tensor_attribute_('left_labels', 'labels')
LG = k2.connect(LG)

print(LG.shape)
```

```
LG = k2.determinize(LG)
LG = k2.connect(LG)
print(LG.shape)
```

음성인식을 위한 WFST 구현

- 5) LG composition&determinization, CLG composition&determinization
  - Token 과 Word index 상 '#0'보다 높은 index를 가진 label을 모두 epsilon으로 치환함
    - 실제 음성인식의 결과로 쓰이지 않기 때문
  - remove_epsilon()을 통해 epsilon을 모두 삭제함
  - 완료된 LG FST를 torch.save로 저장

```
LG.labels[LG.labels >= tokens_values["#0"]] = 0
# See https://github.com/k2-fsa/k2/issues/874
# for why we need to set LG.properties to None
LG.__dict__["_properties"] = None

assert isinstance(LG.aux_labels, k2.RaggedTensor)
LG.aux_labels.values[LG.aux_labels.values >= words_values["#0"]] = 0

LG = k2.remove_epsilon(LG)
#logging.info(f"LG shape after k2.remove_epsilon: {LG.shape}")

LG = k2.connect(LG)
LG.aux_labels = LG.aux_labels.remove_values_eq(0)
```

```
print(LG.shape)
```

```
torch.save(LG.as_dict(), "lang/LG.pt")
```

음성인식을 위한 WFST 구현

- 5) LG composition&determinization, CLG composition
  - 같은 방법으로 C와 LG를 composition한 뒤, 저장함

```
C = k2.arc_sort(T)
LG = k2.arc_sort(LG)
```

```
CLG = k2.compose(C, LG)

CLG = k2.connect(CLG)

print(CLG.shape)
```

```
torch.save(CLG.as_dict(), 'lang/CLG.pt')
```

음성인식을 위한 WFST 구현

- 6) U WFSA 만들기
  - 1)에서 만들어졌던 'logits.pt' 파일로부터 end-to-end의 output probabilites를 불러옴
  - End-to-end 모델은 맨 뒤 index가 epsilon인데, WFST는 맨 앞 0번 index가 epsilon이기 때문에 이를 통일시켜 주기 위해 method를 정의해서 변경함

```
import torch
nnet_outputs = torch.load('logits.pt')
```

```
print(len(nnet_outputs))
print(nnet_outputs[0].shape)
```

```
2620
torch.Size([88, 129])
```

```
def rearrange_blksym(nnet_outputs):
    nnet_t = nnet_outputs.T
    tmp = nnet_t[1:-1]
    tmp2 = nnet_t[-1:]
    logits = torch.cat([tmp2, tmp])
    logits = logits.T
    logits = torch.tensor([logits.numpy()])

    return logits
```

```
#logits = torch.tensor([nnet_outputs[0]])
logits = rearrange_blksym(torch.tensor(nnet_outputs[0]))
```

음성인식을 위한 WFST 구현

- 6) U WFSA 만들기
  - Segments정의
    - 파일 별 구분을 두기 위해, 파일의 길이 정보를 입력함
  - DenseFsaVec method를 사용하여 logits정보를 Torch tensor 형태의 U WFSA로 변경

```
supervision_segments = torch.tensor([[0, 0, logits.shape[1]]], dtype=torch.int32)
```

```
dense_fsa_vec = k2.DenseFsaVec(
    logits,
    supervision_segments)
```

음성인식을 위한 WFST 구현

- 7) Lattice 생성 및 음성인식 결과 확인
  - Intersect_dense_pruned() method를 사용해서 U FSA를 CLG에 통과 시켜 accept되는 lattice를 출력함
    - Method parameter
      - WFST : CLG
      - U FSA: dense_fsa_vec
      - Search_beam: 음성인식 결과를 내기 위한 디코딩 과정에서 고려하는 beam의 개수. 낮을수록 인식 속도가 빨라지고, 높을 수록 성능이 좋아짐
      - Output_beam: 음성인식의 최종 결과의 개수 (8의 경우, 최고 확률 순서대로 8개까지의 결과를 출력)
      - Min_active_states: 하나의 음성 프레임 입력에서 고려하는 최소 state의 개수
      - Max_active_states: 하나의 음성 프레임 입력에서 고려하는 최대 state의 개수

```
lattice = k2.intersect_dense_pruned(CLG, dense_fsa_vec, 20.0, 8, 30, 1000000)
```

```
best_path = k2.shortest_path(lattice, use_double_scores=True)
```

```
from utils import get_texts

token_ids = get_texts(best_path)
#token_ids = f
hyp = [[words_values[i] for i in ids] for ids in token_ids]
print(" ".join(hyp[0]))
```

음성인식을 위한 WFST 구현

- 7) Lattice 생성 및 음성인식 결과 확인
  - Shortest_path() method를 통해 가장 확률이 높은 경로를 추출함
    - use_double_score가 false인 경우, float을 사용함 (본 실습에서는 차이 없음)
  - 결과는 lattice의 word index로 나타내어지므로, get_texts 를 통해 자연어로 변경함

```
lattice = k2.intersect_dense_pruned(CLG, dense_fsa_vec, 20.0, 8, 30, 1000000)
```

```
best_path = k2.shortest_path(lattice, use_double_scores=True)
```

```
from utils import get_texts

token_ids = get_texts(best_path)
#token_ids = f
hyp = [[words_values[i] for i in ids] for ids in token_ids]
print(" ".join(hyp[0]))
```

```
from utils import get_texts

token_ids = get_texts(best_path)
hyp = [[words_values[i] for i in ids] for ids in token_ids]
print(" ".join(hyp[0]))
```

```
CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
```

```
token_ids
```

```
[[2468, 420162, 565318, 346919, 859237, 683846, 162514, 313855]]
```

```
#ref
CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
```

음성인식을 위한 WFST 구현

- 7) Lattice 생성 및 음성인식 결과 확인
  - Best path 이외에 N-best 결과를 보고자 lattice로부터 nbest결과를 추출

```
from utils import Nbest
```

```
nb = Nbest.from_lattice(lattice, num_paths=20)
```

```
max_indices = nb.tot_scores().argmax()
```

```
max_indices
```

```
tensor([1], dtype=torch.int32)
```

```
nb = nb.intersect(lattice)
```

```
nb.tot_scores().tolist()
```

```
[[-56.097835540771484, -63.19219207763672]]
```

```
[sorted(nb.tot_scores().tolist()[0]).index(x) for x in nb.tot_scores().tolist()[0]]
```

```
[1, 0]
```

```
best_path = k2.index_fsa(nb.fsa, max_indexes)
```

```
for i in range(0, len(nb.tot_scores().tolist()[0])):
    best_path = k2.index_fsa(nb.fsa, torch.tensor([i], dtype=torch.int32))
    token_ids = get_texts(best_path)
    hyp = [[words_values[j] for j in ids] for ids in token_ids]
    hyp_text = " ".join(hyp[0])
    print(hyp_text)
```

```
CONCORD RETURNED TO ITS PLACE AMIDST THE TENTS
CONCORD RETURN TO ITS PLACE AMIDST THE TENTS
```

음성인식을 위한 WFST 구현

- 7) Lattice 생성 및 음성인식 결과 확인
  - 현재까지 진행한 내용들을 토대로 전체 test set에 대한 음성인식 결과를 출력함
    - Transcribes list에 결과가 저장됨
    - 만일 결과가 나오지 않는 test file이 있으면, isolated에 저장됨.
      - 이 경우, beam size 조정 등으로 결과를 낼 수 있음

```
from tqdm import tqdm

transcribes = []
isolated = []

for i in tqdm(range(0, len(nnet_outputs))):
    logits = rearrange_blksym(nnet_outputs[i])
    supervision_segments = torch.tensor([[0, 0, logits.shape[1]]], dtype=torch.int32)
    dense_fsa_vec = k2.DenseFsaVec(logits,supervision_segments)
    lattice = k2.intersect_dense_pruned(CLG, dense_fsa_vec, 30.0, 10, 30, 1000000)
    best_path = k2.shortest_path(lattice, use_double_scores=True)

    token_ids = get_texts(best_path)
    hyp = [[words_values[j] for j in ids] for ids in token_ids]
    hyp_text = " ".join(hyp[0])
    if hyp_text =='':
        isolated.append(i)

    transcribes.append(hyp_text)
```

음성인식을 위한 WFST 구현

- 7) Lattice 생성 및 음성인식 결과 확인
  - 저장해놓은 reference text를 불러와서 성능 확인
    - 약 2.3%의 WER을 보임 (baseline: 2.7%)

```
ref = []
with open('ref.txt', 'r') as f:
    ref = f.read().splitlines()
```

```
from jiwer import wer

wer(transcribes, ref)
```

```
0.02301537525894673
```

결론

- Summary / things to remember
  - End-to-end는 현재 시점 가장 성능이 좋은 음향 모델
  - WFST는 현재 시점 가장 최적화된 End-to-end 모델+언어 모델의 디코딩 네트워크를 구성하는 방법
    - 준비물: 언어 모델의 arpa format, word set, end-to-end 모델의 token set, test set의 output probabilities
    - Utterance acceptor, CTC transducer, lexicon transducer, grammar transducer의 구현으로 이루어짐
  - K2-fsa와 Nemo를 이용하여 각각 WFST와 end-to-end를 구현
    - Nemo toolkit은 논문 성능 재현이 가장 잘 되어 있음
    - k2는 python 기반으로 WFST를 구현할 수 있는 library
  - 구현 결과, end-to-end 모델에 비해 0.4%의 절대적 성능 개선을 보임