전자공학종합설계 최종보고서

- 팀 구성
  - 12171470 김성우
  - 12171535 이준영
- 목차

1. 연구목적

Speech Command dataset을 이용해 음성인식 task를 수행하고, CER을 높인다.

2. 모델 구조

Conformer, Squeezeformer 두 개의 모델로 실험을 진행하였지만, 최종적으로 결정한 모델은 Squeezeformer이다.

두 개의 모델을 학습하면서 실험한 이유는, baseline에서 LAS 모델을 대체하기 위해 Conformer 모델을 먼저 사용하는 데 성공하였고, 유사한 architecture인 Squeezeformer 로 대체하여 실험하는 것은 상대적으로 쉽게 진행할 수 있었기 때문이다. 게다가 두 모델의 성능 차이가 크게 나지 않아서 두 모델에 대한 최적 hyper-parameter를 동시에 찾고, 조금이라도 더 나은 성능을 보이는 모델을 최종으로 선택하고자 두 모델로 동시에 실험을 진행한 것이다.

Squeezeformer의 architecture는 Conformer를 수정하며 제안된 것이다. 따라서 Conformer의 architecture를 먼저 알아보고, Squeezeformer와 어떤 차이를 가지고 있는지를 중심으로 설명해보자.

2-1. Conformer [2]

Conformer는 Transformer와 CNN의 장단점을 서로 보완하기 위해 제안된 모델이다.

	Convolution	Transformer
장점	Local Feature를 잘 잡아냄	Global Feature를 잘 잡아냄

단점	Global Feature를 잡아내기 어려움	Local Feature를 잡아내기 어려움

이러한 단점을 보완하기 위해 설계된 Conformer 모델의 architecture는 아래와 같다.

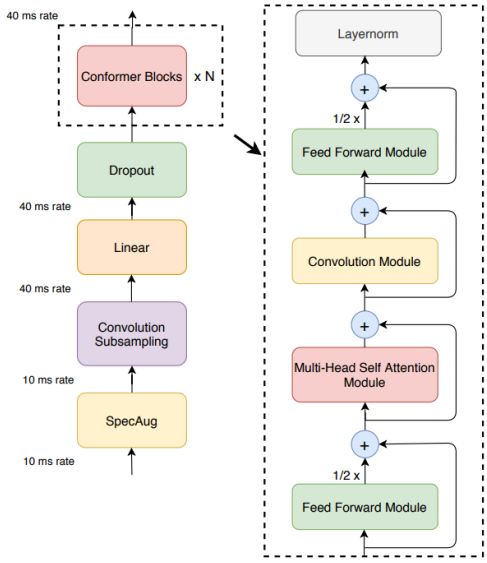


Figure 1. Conformer의 architecture [2]

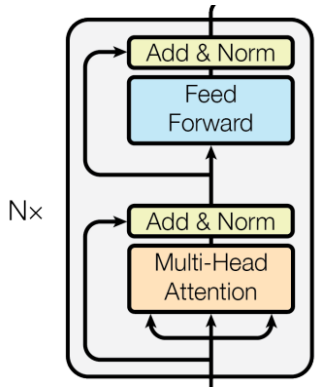


Figure 2. Transformer의 encoder block [5]

처음에는 subsampling을 위한 convolution 계층이 있고, 한 개의 Linear 계층과 Dropout을 거친 뒤 N개의 Conformer Block을 통과한다. Conformer Block은 위의 Figure 1과 같이 4개의 모듈로 이루어져 있고, 각 모듈은 residual connection을 통해 다음 모듈로 data를 전달한다.

Conformer와 Transformer의 encoder block 구조를 비교한 결과는 다음과 같다.

	Conformer	Transformer
Block 구성	Feed Forward → Multi-head Self Attention → Convolution → Feed Forward	Multi-head Attention → Layer Normalization → Feed Forward → Layer Normalization

Conformer 논문에서는 Transformer block과 성능 차이를 만드는 가장 큰 요소는 Convolution 모듈이라고 주장한다.

Model Architecture	dev clean	dev other	test clean	test other
Conformer Model	1.9	4.4	2.1	4.3
– SWISH + ReLU	1.9	4.4	2.0	4.5
– **Convolution Block**	2.1	4.8	2.1	4.9
– Macaron FFN	2.1	5.1	2.1	5.0
– Relative Pos. Emb.	2.3	5.8	2.4	5.6

2-2. Squeezeformer [3]

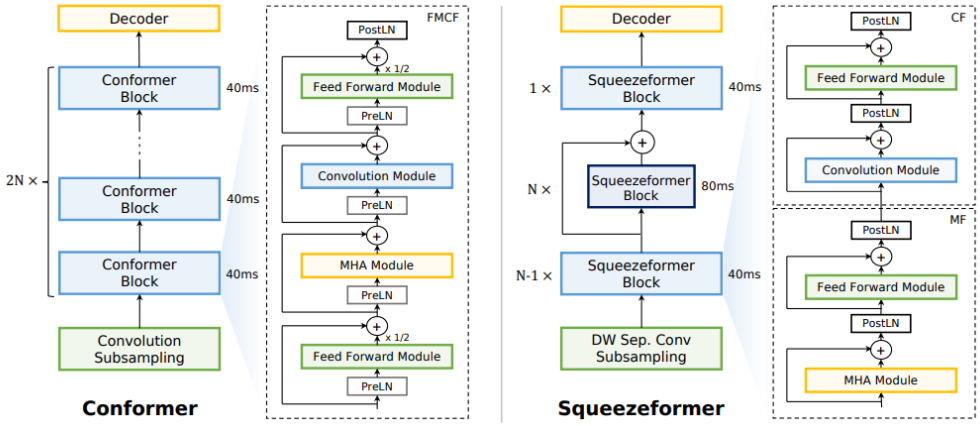


Figure 3. Conformer와 Squeezeformer의 architecture 비교

Figure 3 은 Squeezeformer의 baseline모델인 Conformer와 Squeezeformer를 비교한 그림이다. 주요한 구조의 차이점은 아래와 같다.

	Conformer	Squeezeformer
Block Architecture	[FMCF 구조] Feed Forward → Multi-head Self Attention → Convolution → Feed Forward	[MF/CF 구조] Multi-head Self Attention →Feed Forward →Convolution →Feed Forward
Layer Normalization	pre Layer Normalization	post Layer Normalization

Conformer와 비교할 때, Squeezeformer Block에 추가된 요소는 다음과 같다.

- macro architecture
  - Temporal U-Net Architecture
    - Figure 3을 보면, Conformer Block을 지나면서 시간 축에 대해 subsampling되는 일은 없지만, Squeezeformer block을 지나면서 한번의 down sampling, up sampling이 발생하는 U-Net architecture를 볼 수 있다. 이로 인해 Multi-head attention을 위한 계산량이 quadratic하게 줄어든다.
  - Transformer style block
    - 위에서 Conformer와 Squeezeformer의 block을 비교한 table에서 알 수 있듯, Squeezeformer는 Transformer와 비슷하게 post layer normalization을 사용하고 있다. 또한 MHA 와 Feed-Forward가 이어지는 부분이 transformer와 더 비슷하다고 할 수 있다.
- micro architecture : macro architecture의 squeezeformer를 더 최적화하기 위해 도입된 방법들이다.
  - Simplified Layer Normalizations
    - pre-LN(Layer Normalization)대신 post-LN으로 통일한 후, pre-LN을 learnable scale layer로 대체한다.
  - Depthwise Separable Subsampling
    - Figure 3에서 Conformer와 Squeezeformer의 Convolution Subsampling layer를 관찰하면, Conformer의 경우 그냥 naive Convolution layer이고, Squeezeformer의 경우 Depthwise Separable Convolution을 사용함을 알 수 있다.

Model	Design change	test-clean	test-other	Params (M)	GFLOPs
Conformer-CTC-M	Baseline	3.20	7.90	27.4	71.7
	+ Temporal U-Net (§ 3.1.1)	2.97	7.28	27.5	57.0
	+ Transformer-style Block (§ 3.1.2)	2.93	7.12	27.5	57.0
	+ Unified activations (§ 3.2.1)	2.88	7.09	28.7	58.4
	+ Simplified LayerNorm (§ 3.2.2)	2.85	6.89	28.7	58.4
Squeezeformer-SM	+ DW sep. subsampling (§ 3.2.3)	2.79	6.89	28.2	42.7
Squeezeformer-M	+ Model scale-up (§ 3.2.3)	2.56	6.50	55.6	72.0

Squeezeformer의 baseline 모델인 Conformer에 Squeezeformer의 요소들을 더해 나가면서 실험할 때, Squeezeformer의 각 요소들은 비슷한 정도의 성능 향상을 일으켰다고 논문에서 주장한다.

3. 실험 내용

3-1. Audio Feature 생성 파라미터 선정

먼저, 학습에 필요한 audio feature를 결정할 시점에서는 baseline 이외의 다른 모델을 실험하지 못하는 상황이었으므로, 학습에 적절한 audio feature가 어떤 것 인지에 대한 통찰을 얻기 위해 baseline 모델인 LAS 모델에서 여러가지 feature를 적용해보며 실험을 진행했다.

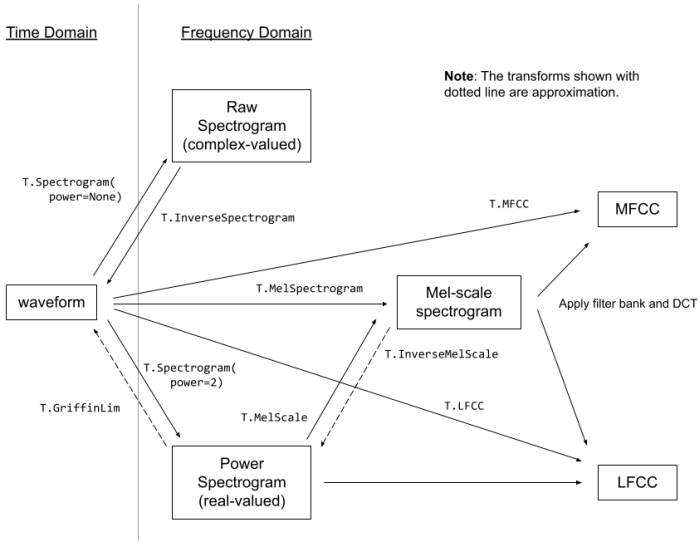


Figure 4. 여러 audio representation들의 관계를 나타낸 그림. [그림 출처]

Audio data를 표현할 수 있는 여러가지 형식의 data 형식이 존재한다. Baseline 코드에서는 STFT(Short-time Fourier Transform)를 적용한 Raw Spectrogram이 사용되고 있다. 하지만 실제로는 사람이 소리를 인식하는 log scale로 Fourier 변환을 실행한 Mel-spectrogram과, 이에

아래 table은 baseline 모델에서 `batch_size` = 32, `lr` =1e-3로 설정한 뒤, feature를 변화 시켜가며 cer성능을 확인한 결과이다.

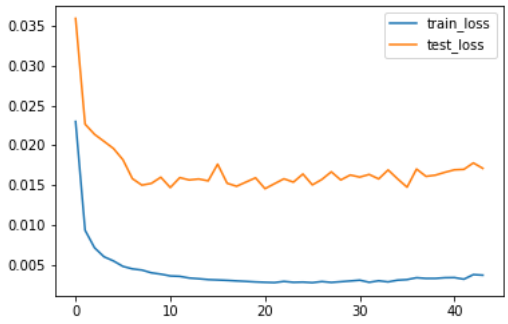


Figure 5. n_mels=26, melspectrogram 일 때의 loss

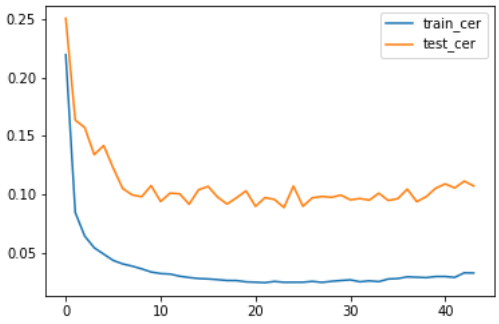


Figure 6. n_mels=26, melspectrogram 일 때의 cer

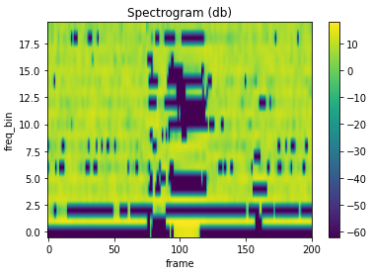
feature \ n_mels	26	40
Mel-scale spectrogram	Loss 0.012496 CER 0.109421	Loss 0.011905 CER 0.116520
MFCC	Loss 0.011989 CER 0.112972	Loss 0.011828 CER 0.115478

```
self.set_spec_params(win_ms=20, overlap_ratio=0.25)
...(중략)...
def set_spec_params(self, win_ms, overlap_ratio):
    """
    win_ms - 윈도우 사이즈를 ms로 나타낸걸 parameter로 받음.
    overlap_ratio - 윈도우가 이동하면서 겹쳐지는 비율
    """
    nfft_candidates = [2**i for i in range(11)]
    self.win_length = int(win_ms*self.sample_rate/1000)
    self.hop_length = int(self.win_length*overlap_ratio)
    for nfft_candidate in nfft_candidates:
        if self.win_length <= nfft_candidate:
            self.n_fft = nfft_candidate
            break
```

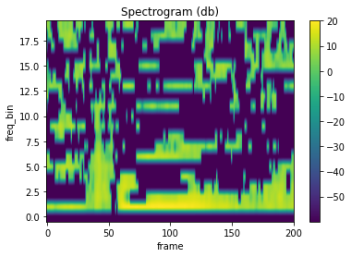
또한 위의 feature를 생성할 때 window size, hop size, n_fft를 결정해주는 `set_spec_params` 메서드를 `Configuration` 클래스에 구현했다.

window length를 time axis에서 몇 ms로 할지, window를 이전 frame과 어느 정도의 비율로 겹치면서 이동할지 입력 받으면 window size, hop size, n_fft를 자동으로 설정해주도록 한 것이다.

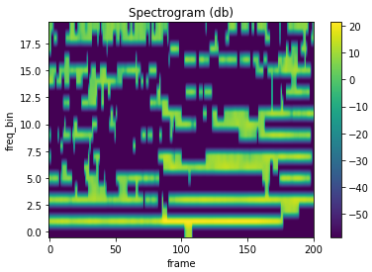
이번 실험에서 window 길이는 20ms, 겹치는 비율은 25%로 하였다. 또한 n_fft는 2의 제곱수로 하는 것이 좋다고 알려져 있으므로, window sample 갯수보다 큰 2의 제곱수 중 가장 작은 수로 결정되도록 하였다.



cat 이 녹음된 음원의 mfcc



no 가 녹음된 음원의 mfcc



forward 가 녹음된 음원의 mfcc

추후 뒤에서 소개할 모델들에도 이 feature를 그대로 적용하였다. 다만 window length를 20ms가 아닌 15ms, 25ms 인 경우도 적용해보았지만, 15ms일 때는 학습 속도의 측면에서 비효율성이 있었고, 25ms일 때는 발산하는 문제가 있었다.

이는 mfcc 자체가 오디오에 대한 정보를 압축한 feature이므로, window length를 넓혀 frame 개수를 줄이는 것은 학습하는데 충분하지 못한 feature를 제공하는 원인이 될 수 있기 때문인 것으로 생각된다.

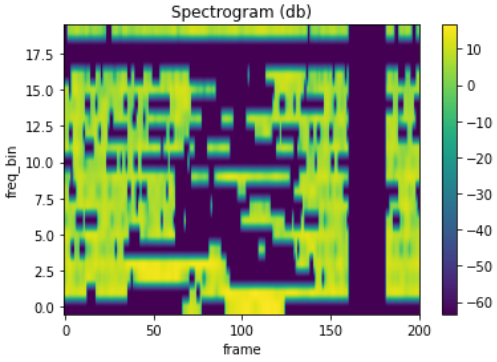
3-2. SpecAugment 적용 [1]

주파수 축에서 audio feature를 augmentation 할 수 있는 대표적인 방법 SpecAugment를 적용하였다. SpecAugment는 2019년 Google Brain에서 발표한 방법으로, 크게 time masking, frequency masking, time warping 세가지 방법을 제시한다.

Data augmentation은 overfitting의 문제를 underfitting으로 바꿔주는 장점이 있다.

3-2-1. Time Masking

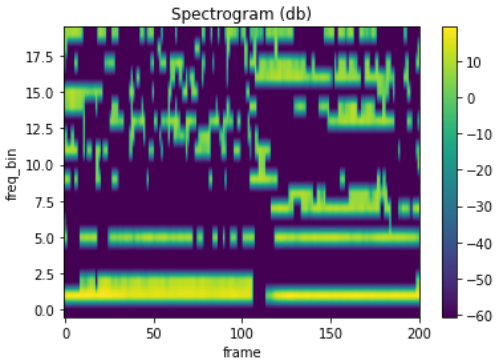
말 그대로 시간 축에서 random한 구간을 frequency 축을 따라 0으로 masking 하는 것이다.



ex) mfcc에 적용한 time masking

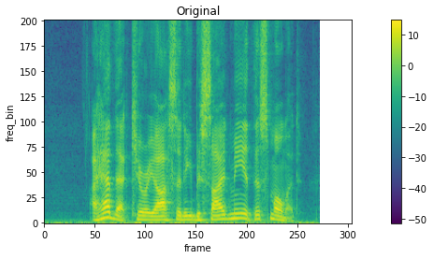
3-2-2. Frequency Masking

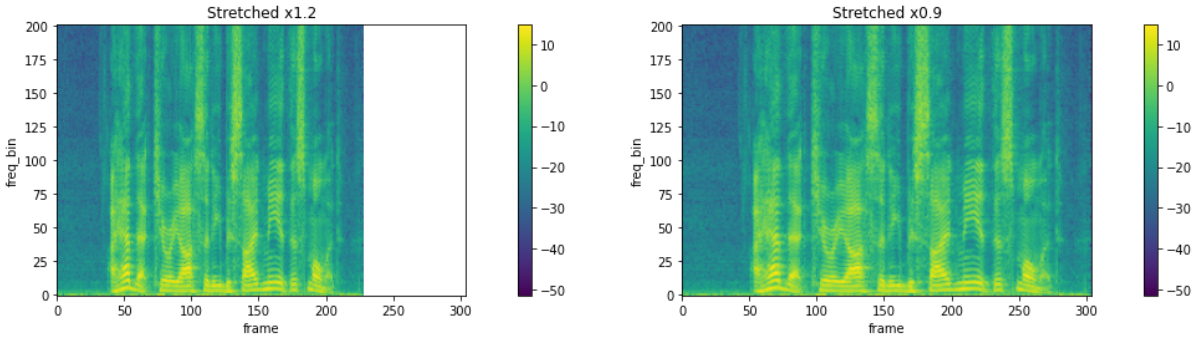
Frequency masking은 주파수 축에서 random 한 구간을 시간축을 따라 0으로 masking 하는 것이다.



ex) mfcc에 적용한 frequency masking

3-2-3. Time Warping





위의 두 이미지는 raw spectrogram에 time warping을 적용한 것이다. 시간 축을 따라 spectrogram이 압축되거나 늘어남을 알 수 있다.

뒤에서도 언급하겠지만, time warping은 SpecAugment 의 세 가지 방법 중에서 가장 효과가 적고, torchaudio에 구현된 `TimeStretch` 는 MFCC에 적용하기에 무리가 있으므로 사용하지 않았다.

3-2-4. 구현

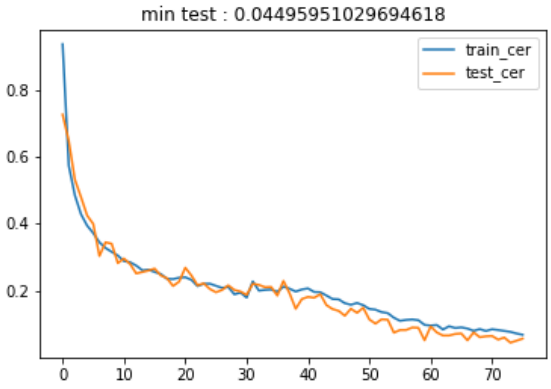
```
import torchaudio.transforms as T

class SpecAugment:
    def __init__(self, low_bound = 1.0, upper_bound = 1.2):
        self.low_bound = low_bound
        self.upper_bound = upper_bound
    def __call__(self, spect):
        """
        spect : (H, W)
        """
        if spect.ndim == 3:
            batch, freq, time = spect.shape
        elif spect.ndim == 2:
            freq, time = spect.shape

        timemask = T.TimeMasking(time//6)
        freqmask = T.FrequencyMasking(freq//6)
        n_freq = config.n_mels//2 if config.feature_type=="mfcc" else config.n_mels

        if random.randint(0,1):
            spect = timemask(spect)
        if random.randint(0,1):
            spect = freqmask(spect)
        return spect
```

SpecAugment를 구현한 코드는 위와 같다. Time masking, frequency masking, time warping 모두 torchaudio에 구현되어 있어 사용할 수 있었다.

하지만, torchaudio에 구현된 time warping의 경우 내부적으로 stft의 결과를 입력 받아 phase vocoder를 이용하여 time warping을 구현되었으므로 실수를 결과로 return 하는 mfcc에 적용할 수 없었다.

또한 time warping은 3가지의 SpecAugment 방법 중 가장 효과가 적다[1]고 논문에 언급되어 있으므로 mfcc를 위한 time warping을 따로 구현하여 사용하지 않았다.

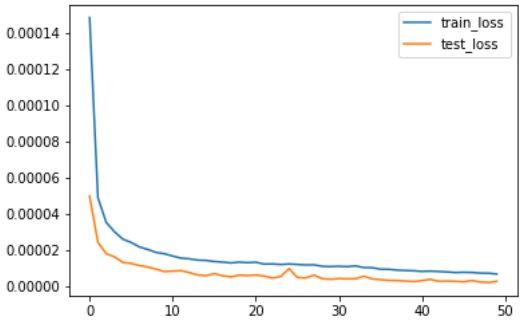
위와 같이 SpecAugment를 사용한 결과, 학습 곡선이 대체로 아래와 같은 경향으로 나타나는 것을 볼 수 있었다.



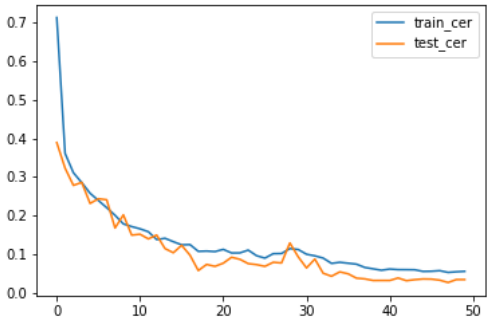
3-3. Early Stopping

Early stopping은 오버피팅을 막기 위해 학습을 지정한 조건에 따라 일찍 끝내도록 하는 기법이다.

학습을 진행하면서 train dataset과 validation dataset의 loss와 CER 을 각각 모니터링 하였다. 그 결과 대체로 validation loss보다 validation의 CER이 더욱 분산이 크게 값이 변하며 줄어든다는 것을 알 수 있었다.



Loss의 변화 예시. CER과 비교하여 상대적으로 안정적으로 변화하고 있다.



CER 변화의 예시. Loss와 비교하여 상대적으로 분산이 크다.

이에 따라 Loss 개선이 `config.LOSS_TOLERANCE` 만큼의 epoch이 지나도록 개선이 없으면 학습을 중단하도록 했고, 또한 CER의 개선이 `config.CER_TOLERANCE` 만큼의 epoch이 지나도록 개선이 없으면 학습을 중단하도록 했다. 이 때 위에서 살펴본 것 처럼 validation CER이 validation loss보다 변동이 크기 때문에 `config.CER_TOLERANCE` 을 `config.LOSS_TOLERANCE` 보다 크게 설정하도록 하였다. 최종 산출물에는 `config.LOSS_TOLERANCE` =10, `config.CER_TOLERANCE` =20 으로 설정하였다.

아래 코드는 training 을 진행하는 코드에서 Early stopping이 적용된 부분만 나타낸 것이다.

```
for epoch in range(config.start_epoch, config.max_epoch+1):

    ...(중략)...

    if Total_eval_loss < best_eval_loss:
        best_eval_loss = Total_eval_loss

        LOSS_NOT_IMPROVED = 0
    else:
        LOSS_NOT_IMPROVED += 1
        if LOSS_NOT_IMPROVED >= config.LOSS_TOLERANCE:
            break

    # CER이 가장 낮게 나온 경우의 모델 저장
    if Total_eval_cer < best_eval_cer:
        best_eval_cer = Total_eval_cer
```

```
        best_cer_model = copy.deepcopy(model)
        best_cer_model_path = get_save_path(f"{config.training_id}_best_cer_model.pt")
        torch.save(best_cer_model, best_cer_model_path)

        CER_NOT_IMPROVED = 0
    else:
        CER_NOT_IMPROVED += 1
        if CER_NOT_IMPROVED >= config.CER_TOLERANCE:
            break
```

또한 validation CER이 가장 낮게 나온 모델을 저장하면서 학습을 진행하였다.

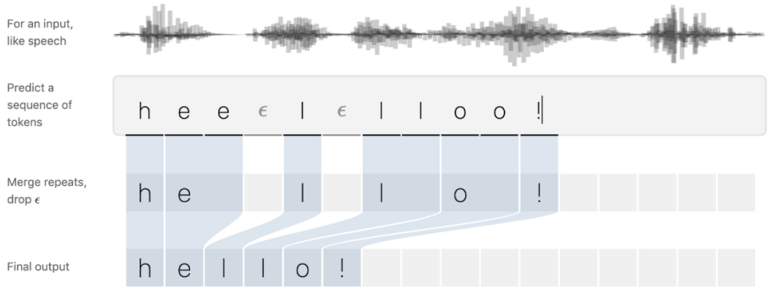
원래 validation Loss가 가장 낮은 모델도 함께 저장하였지만, 최종적으로 test dataset에 대해 검증을 진행할 때마다 CER이 가장 좋은 모델의 성능이 더 좋았기 때문이다.

3-4. CTC decoding / best path decoding [4]

Conformer와 Squeezeformer 모델이 최종적으로 내는 결과물은 3차원 Tensor(batch_size, sequence length, class_num)이다. 이는 각 sequence frame마다 화자의 발음을 알파벳으로 mapping한 결과여야 한다. 따라서 loss function은 자동으로 sequence와 label을 align 시켜주는 CTC loss를 이용하였다.

```
# CTC loss 함수를 criterion으로 선언
criterion = nn.CTCLoss().to(device)

...(중략)...

# train, eval 함수에서 CTC loss를 계산하는 모습
logit, output_lengths = model(feats, feat_lengths)

target = targets[:, 1:]

# Loss 계산
logit = logit.transpose(0,1)
output_lengths = torch.IntTensor(output_lengths)
targets_lengths = torch.IntTensor(targets_lengths)-1

loss = criterion(logit, target, output_lengths, targets_lengths)
```

또한 아래 그림과 같이 CTC decoding 결과를 거쳐야 한다. 그 중에서도 best path decoding은 아래 그림과 같이 각 시점에서 가장 확률이 높은 token을 `hee-l-lloo!`라고 할 때, blank로 구분되지 않고 반복되는 문자들을 제거한 후, blank를 제거하면 `hello!` 가 되는 원리를 이용하여 decoding한다.



CTC decoding을 그림으로 나타낸 것. [그림 출처]

이를 코드로 작성한 것은 아래와 같다. 기존 baseline의 `label_to_string` 함수를 변형하여 `label_to_string_hyp` 를 추가로 작성해서 model의 output에 대해서만 `label_to_string_hyp` 를 적용했다.

```
#label_to_string
def label_to_string_hyp(labels):
    SWITCH = False
    if len(labels.shape) == 1:
```

```
        sent = str()
        for i in labels:
            if i.item() == config.EOS_token:
                break
            if i.item() == config.PAD_token: # pad 토큰이 나오는 경우는 그냥 넘어간다
                SWITCH = False
                continue
            if len(sent)>0 and i.item() == sent[-1] and SWITCH:
                continue
            sent += config.index2char[i.item()]
            SWITCH = True
        return sent

    elif len(labels.shape) == 2:
        sents = list()
        for i in labels:
            sent = str()
            for j in i:
                if j.item() == config.EOS_token:
                    break
                if j.item() == config.PAD_token:
                    SWITCH = False
                    continue
                if len(sent)>0 and j.item() == sent[-1] and SWITCH:
                    continue
                sent += config.index2char[j.item()]
            sents.append(sent)

        return sents
```

이에 따라 `label_to_string` 함수가 적용되었던 기존의 함수 `get_distance` 를 다음과 같이 변형했다.

```
def get_distance(ref_labels, hyp_labels):
    total_dist = 0
    total_length = 0
    for i in range(len(ref_labels)):

        ref = label_to_string(ref_labels[i])
        hyp = label_to_string_hyp(hyp_labels[i])
        # print(ref, hyp)
        dist, length = char_distance(ref, hyp)
        total_dist += dist
        total_length += length

    return total_dist, total_length
```

4. 실험결과

3에서 적용한 실험 내용들을 처음부터 모두 반영한 채로 실험을 진행한 것은 아니다. 기능을 하나씩 추가하면서 실험을 진행한 관계로, 진행 epoch 등 정보가 비어있는 곳이 존재한다.

또한 시간 상의 문제로 인하여 hyper-parameter 설정을 직관에 많이 의존할 수 밖에 없었다. 따라서 계획한 search grid에 대해 모든 실험을 진행할 수 없었다.

4-1. Conformer

Conformer 논문에서 제안한 모델의 hyper-parameter는 아래와 같다. 이 실험에서 사용한 Speech Command dataset은 원래 논문에서 사용한 dataset인 Librispeech보다 매우 작은 dataset이다. 따라서 아래 논문에서 제안된 hyper-parameter 중 Conformer-S, Conformer-M 으로 실험을 진행했다.

Model	Conformer (S)	Conformer (M)	Conformer (L)
Num Params (M)	10.3	30.7	118.8
Encoder Layers	16	16	17
Encoder Dim	144	256	512
Attention Heads	4	4	8
Conv Kernel Size	32	32	32
Decoder Layers	1	1	1
Decoder Dim	320	640	640

Conformer-S

batch_size\lr	5e-5	1e-4	5e-4
16	-	-	발산
32	-	-	-
64	0.048362 / 76	**0.03 / ?**	0.03582
128	-	-	0.04053

Conformer-M

batch_size\lr	5e-5	1e-4	5e
32	-	-	-
64	0.016941 / 100 (valid)	0.015237 / 50 (valid)	발
128	0.052413 / 60	0.0803 / 17	발

Conformer-M의 실험 결과에서 CER이 0.015 ~ 0.016이 나온 부분(노란색)은 런타임이 끊겨 최종적으로 test dataset에 대한 결과를 얻을 수 없었다. Conformer-M의 batch size 64, learning rate 5e-5, 1e-4인 경우를 최종 모델로 선택하지 않은 이유는, validation set에서 월등한 성능을 보이는 모델의 경우, 의외로 test set에서는 다른 hyper-parameter 조합의 모델들과 다르지 않은 성능을 보이는 경우가 많았기 때문이다.

4-2. Squeezeformer

Squeezeformer 논문에서 제안된 hyper-parameter에 따라 실험을 진행하였다. 역시 model의 학습 시간에 대한 문제로 Squeezeformer-SM 모델, Squeezeformer-M 모델에 대해 실험을 진행했다.

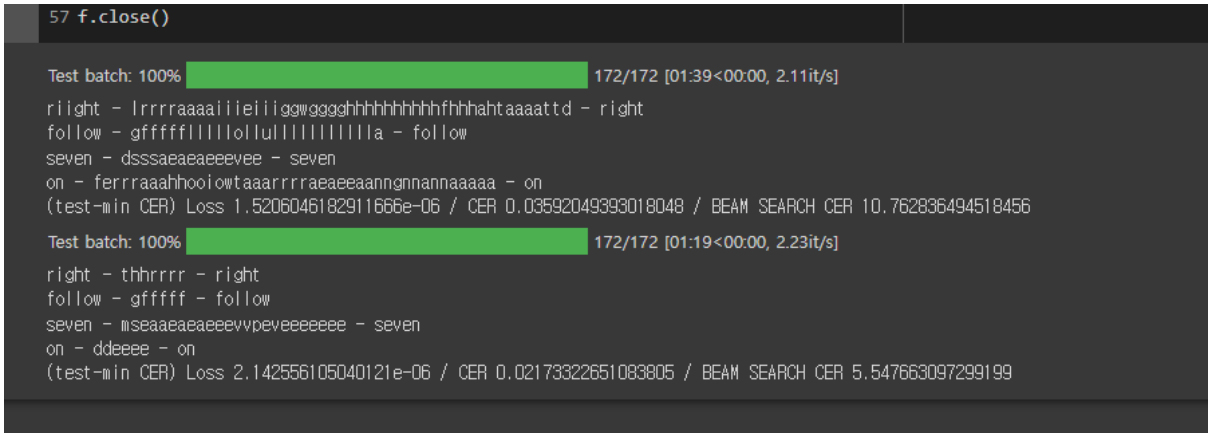
Model	# Layers	Dimension	# Heads	Params (M)	GFLOPs
Conformer-CTC-S	16	144	4	8.7	26.2
Squeezeformer-XS	16	144	4	9.0	15.8
Squeezeformer-S	18	196	4	18.6	26.3
Conformer-CTC-M	16	256	4	27.4	71.7
Squeezeformer-SM	16	256	4	28.2	42.7
Squeezeformer-M	20	324	4	55.6	72.0
Conformer-CTC-L	18	512	8	121.5	280.6
Squeezeformer-ML	18	512	8	125.1	169.2
Squeezeformer-L	22	640	8	236.3	277.9

SM model

batch_size\lr	5e-5	1e-4	5e-4
64		0.096 / 18	
128	0.195 / 33		0.10330

M model

batch_size\lr	5e-5	1e-4	5e
64		0.035 / 12 0.0304 / 24 0.0429 / 22 **0.0217 / ?**	
128	0.08 / 21	0.0301 / 14 0.0638 / 30	



test dataset에 대해 CER이 가장 낮게 나온 순간을 캡처한 것이다.

5. 고찰 및 결론

5-1. 실험하면서 얻은 Insights

- 학습에 좋은 audio feature를 한 번 정해 놓고 나니, 다른 모델에서도 어느 정도는 통용되는 것을 알 수 있었다. 하지만 절대적으로 좋은 것은 없으므로 모델을 바꾸고 나서도 다양한 audio feature에 대해 실험을 해 볼 수 있었다면 좋았을 것이다.
- Conformer와 Squeezeformer 두 모델 모두 비슷하게 좋은 성능을 내는 모델이다. 따라서 Hyper-parameter를 조절하기에 따라서는 Conformer가 더 나은 결과를 냈을 수도 있을 것이다.

5-2. 계획에 있었지만 시도해보지 못한 것들

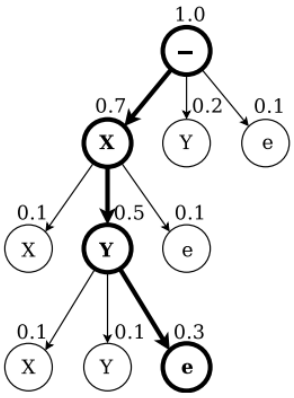
5-2-1. Ablation study of Conformer

Conformer 논문에서 모델의 크기와 구조를 결정하는 hyper-parameter들의 조합에 따라 각 요소들이 모델의 성능에 미치는 영향을 정리한 부분이다. 논문의 ablation study를 적용하였다면 더 좋은 hyper-parameter를 찾을 수 있었을 것이다. Conformer의 ablation study를 정리하면 다음과 같다.

- Conformer는 Attention head의 개수를 16개 까지 늘리는 것은 성능에 좋은 영향을 주지만, 그 이상으로 늘리면 오히려 성능이 떨어진다는 실험 결과가 있다.
- 또한 Convolution kernel size를 32까지 늘리면 성능이 좋아지지만, 그 이상 늘리면 성능이 떨어진다는 실험 결과가 있다.

5-2-2. Beam Search Decoding (Prefix-Search Decoding)

최종 산출물 코드에서는 각 출력 frame마다 가장 확률이 높은 token(알파벳 또는 blank)로 greedy decoding(또는 best path decoding)하였지만, Beam Search Decoding은 설정한 beam size 만큼의 상위 토큰을 고려하는 알고리즘이다. 그렇게 해서 가장 확률이 높은 path들을 beam size 갯수 만큼 남기면서 decoding을 진행하는 알고리즘이다. 아래 그림은 beam size가 3일 때의 예시를 나타낸 것이다.



beam size가 3일 때 [4]

일반적으로 Beam search decoding을 하는 것이 성능이 더 좋다고 알려져 있으므로 이를 적용해보고자 하였다. 하지만 위의 Squeezeformer에서 살펴본 결과와 같이, decoding 결과가 매우 좋지 않게 나와 사용할 수 없을 정도였으므로 사용하지 않았다.

5-2-3. VAD (Voice Activity Detection)

VAD는 음원에서 음성이 발생한 구간을 자동으로 감지하여 음원을 trimming 해주는 기술로, torchaudio에 구현되어 있다.

하지만, VAD를 사용하여 음성 구간만 남기고 나머지 부분을 버리는 것은 연산을 효율적으로 해 주는 것은 분명하지만, 실제로 음성 부분을 온전히 남긴다는 보장이 없기 때문에 사용하지 않았다.

References

[1] Park, Daniel S., William Chan, Yu Zhang, Chung-Cheng Chiu, Barret Zoph, Ekin D. Cubuk, and Quoc V. Le. "Specaugment: A simple data augmentation method for automatic speech recognition." *arXiv preprint arXiv:1904.08779* (2019).

[2] Gulati, Anmol, James Qin, Chung-Cheng Chiu, Niki Parmar, Yu Zhang, Jiahui Yu, Wei Han et al. "Conformer: Convolution-augmented transformer for speech recognition." *arXiv preprint arXiv:2005.08100* (2020).

[3] Kim, Sehoon, Amir Gholami, Albert Shaw, Nicholas Lee, Karttikeya Mangalam, Jitendra Malik, Michael W. Mahoney, and Kurt Keutzer. "Squeezeformer: An Efficient Transformer for Automatic Speech Recognition." *arXiv preprint arXiv:2206.00888* (2022).

[4] Graves, Alex, Santiago Fernández, Faustino Gomez, and Jürgen Schmidhuber. "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks." In *Proceedings of the 23rd international conference on Machine learning*, pp. 369-376. 2006.

[5] Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. "Attention is all you need." *Advances in neural information processing systems* 30 (2017).