Neural Machine Translation

Kyunghyun Cho

Machine Translation (MT)

번역은 인류가 언어를 사용하기 시작한 이래로 큰 관심사 중에 하나였습니다. 그러한 의미에서 기계 번역(Machine Translation)은 단순히 언어를 번역하는 것이 아닌, 자연언어처리 영역에서의 종합예술이라 할 수 있습니다. 사실 이번 챕터를 위해서

이전 챕터들을 다루었다고 해도 과언이 아닙니다. Neural Machine Translation (NMT)는 end-to-end learning 으로써, Rule Based Machine Translation (RBMT), Statistical Machine Translation (SMT)로 이어져온 기계 번역의 30년 역사 중에서 가장 큰 성취를 이루어냈습니다. 이번 챕터는 NMT에 대한 인사이트를 얻을 수 있도록, seq2seq with attention의 동작 방식/원리를 이해하고, 더 나아가 응용 방법에 대해서도 소개 합니다. 또한, 기계번역 시스템을 만들기 위한 프로세스와 최신 연구 동향을 아울러 소개 합니다.

Objective

$$\hat{e} = argmax P_{f \to e}(e|f)$$

번역의 궁극적인 목표는 어떤 언어(f, e.g. french)의 문장이 주어졌을 때, 우리가 원하는 언어(e, e.g. english)로 확률을 최대로 하는 문장을 찾아내는 것 입니다.

Why it is so hard?

인간의 언어는 컴퓨터의 언어(e.g. programming language)와 같이 명확하지 않습니다. 우리는 언어의 모호성(ambiguity)을 늘림으로써, 의사소통의 효율을 극대화 시켰습니다. 예를 들어 우리는 정보를 생략하고 말을 짧게 해 버리고, 같은 단어 같은 구절이라고 하더라도 때에 따라 다른 의미로 해석될 수 있습니다. 더욱이 한국어의 경우에는 첫 챕터에서 다루었듯이 어순이 불규칙하고 주어가 생략 되는 등, 더욱 더 난이도가 높아졌습니다. 또한, 언어라는 것은 그 민족의 문화를 담고 있기 때문에, 수천년의 세월동안 쌓여온 사람들의 의식, 철학이 담겨져있고, 그러한 차이들로 하여금 또 다시 번역을 어렵게 만듭니다. 결국, 이러한 점들은 컴퓨터로 하여금 우리의 말을 번역하고자 할 때 큰 장벽으로 다가옵니다.

대표적인 번역 실패 사례

In brightest day, in blackest night,
(일기가 좋은 날, 진흙같은 어두운 밤,)
No evil shall escape my sight.
(아니다 이 악마야, 내 앞에서 사라지지.)
Let those who worship evil's might,
(누가 사악한 수도악마를 숭배하는지 볼까,)
Beware my power, Green Lantern's light!!!
(나의 능력을 조심해라, 그린 랜턴 빛!)

Why it is so important?

하지만 이러한 어려움에도 불구하고 기계 번역은 우리에게 꼭 필요한 과제입니다. 이 순간에도 전세계에서는 기계번역을 통해서 많은 일들이 일어납니다. Facebook과 같은 세계인이 소통하는 SNS, Amazon과 같은 전 세계를 대상으로 하는 인터넷 쇼핑몰에서도 번역 서비스를 제공하며 사용자들은 이를 통해 편리함을 얻을 수 있습니다.

History

Rule based Machine Translation

전통적인 방식의 번역이라고 할 수 있습니다. 우리가 흔히 어릴때 부터 배워 온 방식입니다. 문장의 구조를 분석하고, 그 분석에 따라 규칙을 세우고 분류를 나누어서 정해진 규칙에 따라서 번역을 합니다. 밑에서 다룰 SMT에 비해서 자연스러운 표현이

가능하지만, 그 규칙을 일일히 사람이 만들어내야 하므로 번역기를 만드는데 많은 자원과 시간이 소모됩니다. 따라서 번역 언어쌍을 확장할 때에도, 매번 새로운 규칙을 찾아내고 적용해야 하기 때문에 굉장히 불리합니다.

Statistical Machine Translation

NMT이전에 세상을 지배하던 번역 방식입니다. 대량의 양방향 corpus로부터 통계를 얻어내어 번역 시스템을 구성합니다. Google이 자신의 번역 시스템에 도입하면서 더욱 유명해졌습니다. 이 시스템 또한 여러가지 모듈로 이루어져 굉장히 복잡합니다. 통계기반 방식을 사용하므로 언어쌍을 확장할 때, 대부분의 알고리즘이나 시스템은 유지되므로 기존의 RBMT에 비하여 매우 유리하였습니다.

Neural Machine Translation

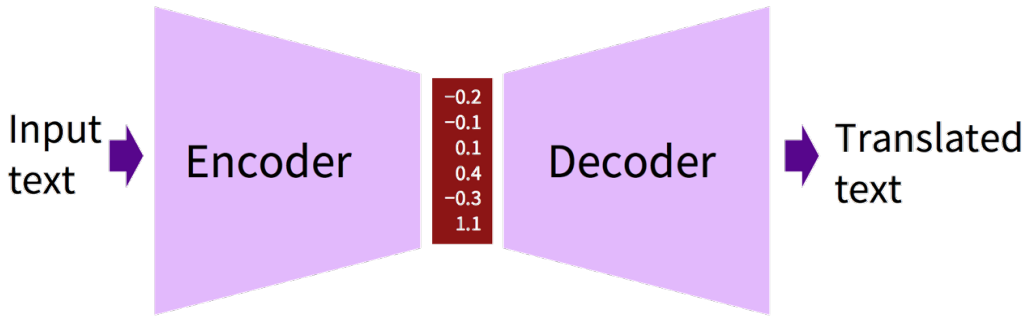

Image from CS224n

사실, 딥러닝 이전의 AI의 전성기(1980년대)에도 Neural Network을 사용하여 Machine Translation 문제를 해결하려는 시도는 여럿 있었습니다. 하지만 당시에도 $Encoder \longrightarrow Decoder$ 형태의 구조를 가지고 있었지만, 당연히 지금과 같은 성능을 내기는 어려웠습니다.

Invasion of NMT

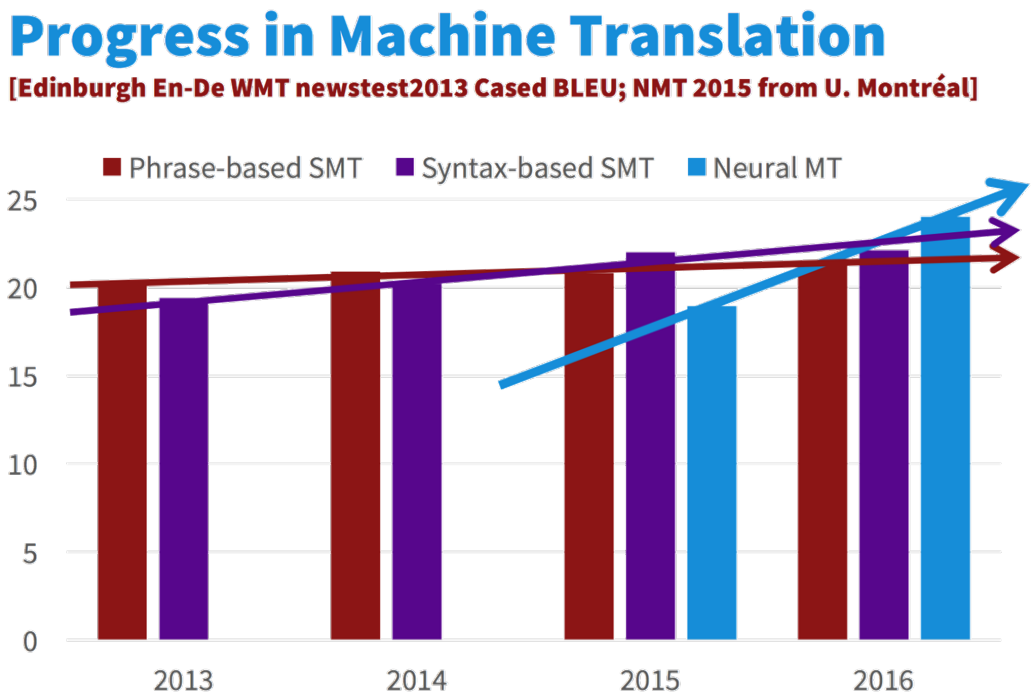

Image from CS224n

현재의 NMT 방식이 제안되고, 곧 기존의 SMT 방식을 크게 앞질러 버렸습니다. 이제는 구글의 번역기에도 NMT 방식이 사용됩니다.

Why it works well?

왜 Neural Machine Translation이 잘 동작하는 것일까요?

1. End-to-end Model

- NMT 이전의 SMT의 경우에는 번역시스템이 여러가지 모듈로 구성이 되어 있었고, 이로 하여금 시스템의 복잡도를 증가시켜 훈련에 있어서 훨씬 지금보다 어려운 경향이 있었습니다. 하지만 NMT는 단 하나의 모델로 번역을 해결함으로써, 성능을 극대화 하였습니다.

2. Better language model

- 신경망 언어모델(Neural Network Language Model)을 기반으로 하는 구조이므로 기존의 SMT방식의 언어모델보다 더 강합니다. 따라서 희소성(sparseness)문제가 해결 되었으며, 자연스러운 번역 결과 문장을 생성함에 있어서 더 강점을 나타냅니다.

3. Great context embedding

- Neural Network의 특기를 십분 발휘하여 문장의 의미를 벡터화(vectorize)하는 능력이 뛰어납니다. 따라서, 노이즈나 희소성(sparseness)에도 훨씬 더 잘 대처할 수 있게 되었습니다.

Sequence to Sequence

Architecture Overview

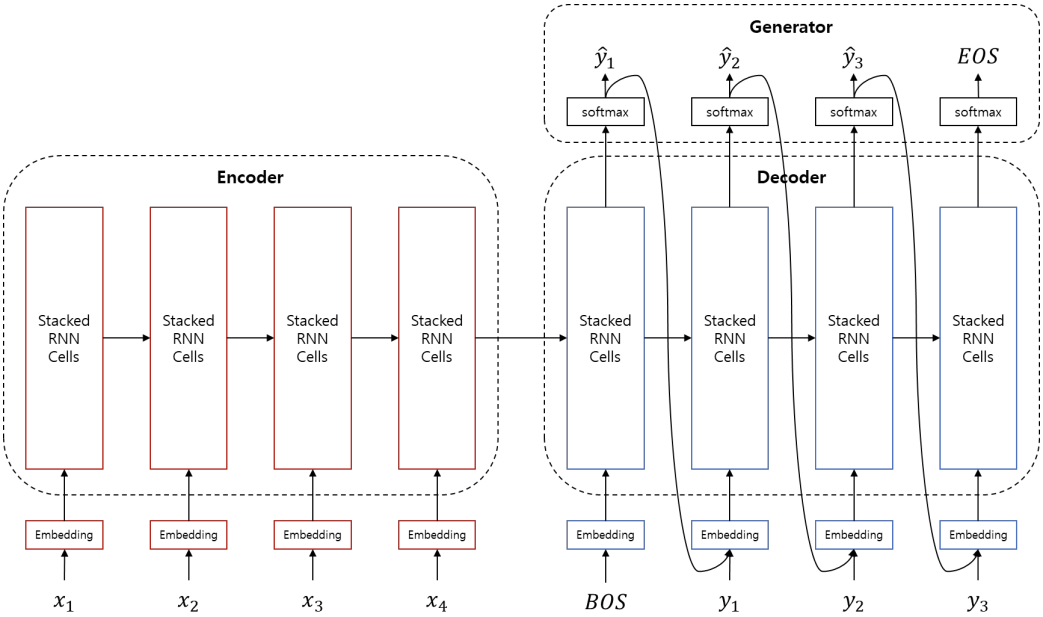

Figure 1:

먼저 번역 또는 seq2seq 모델을 이용한 작업을 간단하게 수식화 해보겠습니다.

$$\theta^* \approx argmax P(Y|X;\theta) \; where \; X = \{x_1, x_2, \cdots, x_n\}, \; Y = \{y_1, y_2, \cdots, y_m\}$$

$P(Y|X;\theta)$를 최대로 하는 모델 파라미터(θ)를 Maximum Likelihood Estimation(MLE)를 통해 찾아야 합니다. 즉, 모델 파라미터가 주어졌을 때, source 문장 X를 받아서 target 문장 Y를 반환할 확률을 최대로 하는 모델 파라미터를 학습하는 것 입니다. 이를 위해서 seq2seq는 크게 3개 서브 모듈(encoder, decoder, generator)로 구성되어 있습니다.

Encoder

인코더는 source 문장을 입력으로 받아 문장을 함축하는 의미의 vector로 만들어 냅니다. $P(X)$를 모델링 하는 것이라고 볼 수 있습니다. 사실 새로운 형태라기 보단, 이전 챕터에서 다루었던 텍스트 분류(Text Classificaion)에서 사용되었던 RNN 모델과 거의 같다고 볼 수 있습니다. $P(X)$를 모델링하여, 주어진 문장을 벡터화(vectorize)하여 해당 도메인의 매니폴드(manifold or hyper-plane)의 어떤 한 점에 투영 시키는 작업이라고 할 수 있습니다.

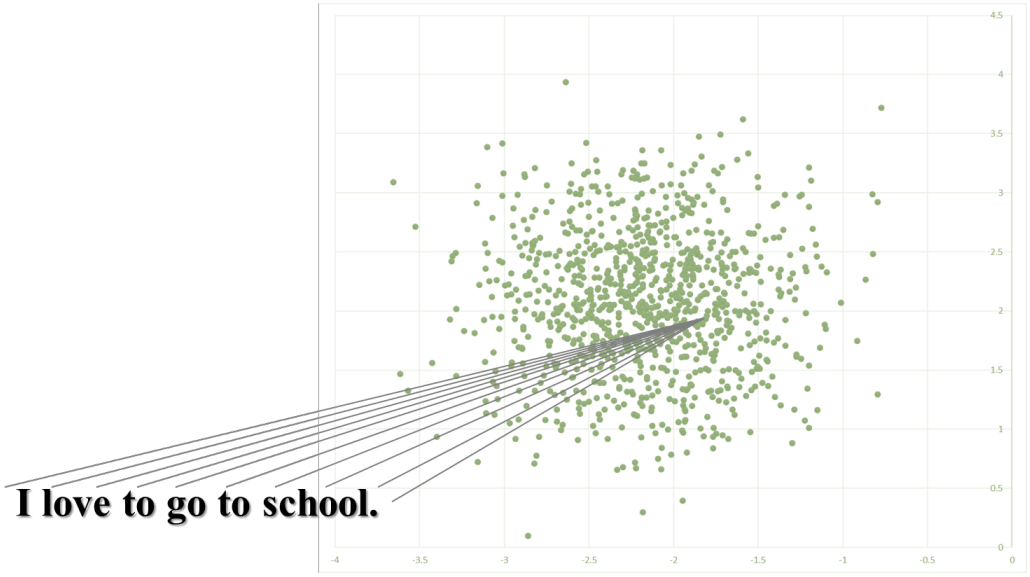

Figure 2:

다만, 기존의 text classification에서는 모든 정보가 필요하지 않기 때문에 (예를들어 감성분석(Sentiment Analysis)에서는 "나는"과 같이 중립적인 단어는 감성을

분류하는데 필요하지 않기 때문에 해당 정보를 굳이 간직해야 하지 않을 수도 있습니다.) vector로 만들어내는 과정인 정보를 압축함에 있어서 손실 압축을 해도 되는 작업이지만, 기계번역에 있어서는 이상적으로는 거의 무손실 압축을 해내야 하는 차이는 있습니다.

$$h_t^{src} = RNN_{enc}(emb_{src}(x_t), h_{t-1}^{src})$$
$$H^{src} = [h_1^{src}; h_2^{src}; \cdots ; h_n^{src}]$$

Encoder를 수식으로 나타내면 위와 같습니다. [;]는 concatenate를 의미합니다. 위의 수식은 time-step 별로 GRU를 통과시킨 것을 나타낸 것이고, 사실상 실제 코딩을 하게 되면 아래와 같이 됩니다.

$$H^{src} = RNN_{enc}(emb_{src}(X), h_0^{src})$$

Decoder

마찬가지로 디코더도 사실 새로운 개념이 아닙니다. 이전 챕터에서 다루었던 신경망언어모델(Nerual Network Langauge Model, NNLM)의 연장선으로써, 조건부 신경망언어모델(Conditional Neural Network Language Model)이라고 할 수 있습니다. 위에서 다루었던 seq2seq모델의 수식을 좀 더 time-step에 대해서 풀어서 써보면 아래와 같습니다.

$$P_\theta(Y|X) = \prod_{t=1}^{m} P_\theta(y_t|X, y_{<t})$$
$$\log P_\theta(Y|X) = \sum_{t=1}^{m} \log P_\theta(y_t|X, y_{<t})$$

보면 RNNLM의 수식에서 조건부에 X가 추가 된 것을 확인 할 수 있습니다. 즉, 이제까지 번역 한 (이전 time-step의) 단어들과 encoder의 결과에 기반해서 현재 time-step의 단어를 유추해 내는 작업을 수행합니다.

$$h_t^{tgt} = RNN_{dec}(emb_{tgt}(y_{t-1}), h_{t-1}^{tgt}) \;\; where\; h_0^{tgt} = h_n^{src} and\; y_0 = BOS$$

위의 수식은 decoder를 나타낸 것입니다. 특기할 점은 decoder 입력의 초기값으로써, y_0에 BOS를 넣어준다는 것 입니다.

Generator

이 모듈은 아래와 같이 Decoder에서 vector를 받아 softmax를 계산하여 최고 확률을 가진 단어를 선택하는 단순한 작업을 하는 모듈 입니다. $|Y| = m$일때, y_m은 EOS 토큰이 됩니다. 주의할 점은 이 마지막 y_m은 decoder 계산의 종료를 나타내기 때문에, 이론상으로는 decoder의 입력으로 들어가는 일이 없습니다.

$$\hat{y}_t = softmax(linear_{hs \to |V_{tgt}|}(h_t^{tgt})) \;\; and \; \hat{y}_m = EOS$$
$$where \; hs \; is \; hidden \; size \; of \; RNN, \; and \; |V_{tgt}| \; is \; size \; of \; output \; vocabulary.$$

Applications of seq2seq

이와 같이 구성된 Seq2seq 모델은 꼭 기계번역의 task에서만 사용해야 하는 것이 아니라 정말 많은 분야에 적용할 수 있습니다. 특정 도메인의 sequential한 입력을 다른 도메인의 sequential한 데이터로 출력하는데 탁월한 능력을 발휘합니다.

Seq2seq Applications	Task (From-To)
Neural Machine Translation (NMT)	특정 언어 문장을 입력으로 받아 다른 언어의 문장으로 출력
Chatbot	사용자의 문장 입력을 받아 대답을 출력
Summarization	긴 문장을 입력으로 받아 같은 언어의 요약된 문장으로 출력
Other NLP Task	사용자의 문장 입력을 받아 프로그래밍 코드로 출력 등
Automatic Speech Recognition (ASR)	사용자의 음성을 입력으로 받아 해당 언어의 문자열(문장)으로 출력
Lip Reading	입술 움직임의 동영상을 입력으로 받아 해당 언어의 문장으로 출력
Image Captioning	변형된 seq2seq를 사용하여 이미지를 입력으로 받아 그림을 설명하

Limitation

사실 seq2seq는 AutoEncoder와 굉장히 역할이 비슷하다고 볼 수 있습니다. 그 중에서도 특히 Sequential한 데이터에 대한 task에 강점이 있는 모델이라고 볼 수 있습니다. 하지만 아래와 같은 한계점들이 있습니다.

Memorization

Neural Network 모델은 데이터를 압축하는데에 탁월한 성능(Manifold Assumption 참고)을 지녔습니다. 하지만, seq2seq를 통하여도 도라에몽의 주머니처럼 무한하게 정보를 압축 할 수 없습니다. 따라서 압축 할 수 있는 정보는 한계가 있기 때문에, 문장(또는 sequence)이 길어질수록 기억력이 떨어지게 됩니다. 비록 LSTM이나 GRU를 사용함으로써 RNN에 비하여 성능을 끌어올릴 수 있었지만, 한계가 있기 마련입니다.

Lack of Structural Information

현재 주류의 Deeplearning NLP는 문장을 이해함에 있어서 구조 정보를 사용하기보단, 단순히 시계열(time-series) 데이터로 다루는 경향이 있습니다. 비록 이러한 접근방법은 현재까지 대성공을 거두고 있지만, 다음 단계로 나아가기 위해서는 구조 정보도 필요할 것이라 생각하는 사람들이 많습니다.

Chatbot?

사실 이 항목은 단점이라기보다는 그냥 당연한 이야기일 수 있습니다. seq2seq는 sequential한 데이터를 입력으로 받아서 다른 도메인의 sequential한 데이터로 출력하는 능력이 뛰어납니다. 따라서, 처음에는 많은 사람들이 seq2seq를 잘 훈련시키면 Chatbot의 기능도 어느정도 할 수 있지 않을까 하는 기대를 했습니다. 하지만 자세히 생각해보면, 대화의 흐름에서 대답은 질문에 비해서 새로운 정보(지식-knowledge, 문맥-context)가 추가 된 경우가 많습니다. 따라서 기존의 typical한 seq2seq의 task(번역, 요약)등은 새로운 정보의 추가가 없기 때문에 잘 해결할 수 있었지만, 대화의 경우에는 좀 더 발전된 구조가 필요할 것 입니다.

Code

NMT를 목표로하는 seq2seq를 PyTorch로 구현하는 방법을 소개합니다. 이번 챕터에서 사용될 전체 코드는 저자의 깃허브에서 다운로드 할 수 있습니다. (업데이트 여부에 따라 코드가 약간 다를 수 있습니다.)

- github repo url: https://github.com/kh-kim/simple-nmt

Encoder Class

```
class Encoder(nn.Module):

    def __init__(self, word_vec_dim, hidden_size, n_layers = 4, dropout_p = .2):
        super(Encoder, self).__init__()

        # Be aware of value of 'batch_first' parameter.
        # Also, its hidden_size is half of original hidden_size, because it is b
        self.rnn = nn.LSTM(word_vec_dim, int(hidden_size / 2), num_layers = n_laye

    def forward(self, emb):
        # |emb| = (batch_size, length, word_vec_dim)

        if isinstance(emb, tuple):
            x, lengths = emb
            x = pack(x, lengths.tolist(), batch_first = True)
        else:
            x = emb

        y, h = self.rnn(x)
        # |y| = (batch_size, length, hidden_size)
        # |h[0]| = (num_layers * 2, batch_size, hidden_size / 2)

        if isinstance(emb, tuple):
            y, _ = unpack(y, batch_first = True)

        return y, h
```

Pack Padded Sequence

아래는 pack_padded_sequence 함수가 동작하는 모습 입니다. 이 함수는 기존의 sample 별 mini-batch를 time-step 별로 표현 해 줍니다. PackedSequence로 표현된 time-step 별 mini-batch는 각 time-step 별 sample의 숫자를 추가적인 정보로 갖고 있습니다. 따라서, 이를 위해서는 mini-batch 내에는 가장 긴 길이의 문장부터 차례대로 정렬되어 있어야 합니다.

```
a = [torch.tensor([1, 2, 3]), torch.tensor([3, 4])]
```

```
b = torch.nn.utils.rnn.pad_sequence(a, batch_first=True)
>>>>
tensor([[ 1,  2,  3],
        [ 3,  4,  0]])
torch.nn.utils.rnn.pack_padded_sequence(b, batch_first=True, lengths=
>>>>PackedSequence(data=tensor([ 1,  3,  2,  4,  3]), batch_sizes=tens
```

Decoder Class

디코더 클래스의 코드는 이후 섹션에서 다루기로 합니다.

Generator Class

```
class Generator(nn.Module):

    def __init__(self, hidden_size, output_size):
        super(Generator, self).__init__()

        self.output = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim = -1)

    def forward(self, x):
        # |x| = (batch_size, length, hidden_size)

        y = self.softmax(self.output(x))
        # |y| = (batch_size, length, output_size)

        # Return log-probability instead of just probability.
        return y
```

Loss function

seq2seq는 기본적으로 각 time-step 별로 가장 확률이 높은 단어를 선택하는 분류 작업(classification task)이므로, cross entropy를 손실함수(loss function)으로 사용합니다. 또한 기본적으로 조건부 언어모델(conditional language model)이라고

볼 수 있기 때문에, 이전 언어모델 챕터에서 다루었듯이 perplexity를 통해 번역 모델의 성능을 나타낼 수 있고, 이 또한 (cross) entropy와 매우 깊은 연관을 가집니다.

아래는 손실(loss)값을 계산하기 위해 PyTorch로부터 손실함수를 준비하는 모습입니다. 사실, 실제 구현할 때에는 softmax layer + cross entropy를 사용하기보단, log softmax layer + negative log likelihood를 사용합니다.

```
loss_weight = torch.ones(output_size)
loss_weight[data_loader.PAD] = 0
criterion = nn.NLLLoss(weight = loss_weight, size_average = False)
```

아래와 같이 cross entropy 수식은 실제 정답의 확률과 feed-forward를 통해 얻은 신경망(θ)의 해당 로그(log)확률 값을 곱하여 평균을 구합니다. 사실 $P(y_i)$의 값은 1이므로 수식에서 생략할 수 있습니다.

$$\begin{aligned}\mathcal{L}(P, P_\theta) &= -\frac{1}{N}\sum_{i=1}^{N} P(y_i)\log P_\theta(y_i) \\ &= -\frac{1}{N}\sum_{i=1}^{N} \log P_\theta(y_i)\end{aligned}$$

따라서 softmax layer 대신, log softmax layer를 사용하여 로그 확률을 구하고, 수식의 나머지 작업을 수행하면 됩니다.

```
def get_loss(y, y_hat, criterion, do_backward = True):
    # |y| = (batch_size, length)
    # |y_hat| = (batch_size, length, output_size)
    batch_size = y.size(0)

    loss = criterion(y_hat.contiguous().view(-1, y_hat.size(-1)), y.contiguous().v
    if do_backward:
        loss.div(batch_size).backward()

    return loss
```

Attention

Motivation

한 문장으로 Attention을 정의하면 **쿼리(Query)와 비슷한 값을 가진 키(Key)를 찾아서 그 값(Value)을 얻는 과정** 입니다. 따라서, 우리가 흔히 json이나 프로그래밍에서 널리 사용하는 Key-Value 방식과 비교하며 attention에 대해서 설명 하겠습니다.

Key-Value function

Attention을 본격 소개하기 전에 먼저 우리가 알고 있는 자료형을 짚고 넘어갈까 합니다. Key-Value 또는 Python에서 Dictionary라고 부르는 자료형 입니다.

```
>>> dic = {'computer': 9, 'dog': 2, 'cat': 3}
```

위와 같이 Key와 Value에 해당하는 값들을 넣고 Key를 통해 Value 값에 접근 할 수 있습니다. 좀 더 바꿔 말하면, Query가 주어졌을 때, Key 값에 따라 Value 값에 접근 할 수 있습니다. 위의 작업을 함수로 나타낸다면, 아래와 같이 표현할 수 있을겁니다. (물론 실제 Python Dictionary 동작은 매우 다릅니다.)

```
def key_value_func(query):
    weights = []

    for key in dic.keys():
        weights += [is_same(key, query)]

    weight_sum = sum(weights)
    for i, w in enumerate(weights):
        weights[i] = weights[i] / weight_sum

    answer = 0

    for weight, value in zip(weights, dic.values()):
        answer += weight * value

    return answer

def is_same(key, query):
```

```
    if key == query:
        return 1.
    else:
        return .0
```

코드를 살펴보면, 순차적으로 dic 내부의 key값들과 query 값을 비교하여, key가 같을 경우 weights 에 1.0을 추가하고, 다를 경우에는 0.0을 추가합니다. 그리고 weights를 weights의 총 합으로 나누어 weights의 합이 1이 되도록 (마치 확률과 같이) normalize하여 줍니다. 다시 dic 내부의 value값들과 weights의 값을 inner product (스칼라곱, dot product) 합니다. 즉, $weight = 1.0$ 인 경우에만 value 값을 answer에 더합니다.

Differentiable Key-Value function

좀 더 발전시켜서, 만약 is same 함수 대신에 다른 함수를 써 보면 어떻게 될까요? how similar 라는 key와 query 사이의 유사도를 리턴 해 주는 가상의 함수가 있다고 가정해 봅시다.

```
>>> query = 'puppy'
>>> how_similar('computer', query)
0.1
>>> how_similar('dog', query)
0.9
>>> how_similar('cat', query)
0.7
```

그리고 해당 함수에 'puppy'라는 단어를 테스트 해 보았더니 위와 같은 값들을 리턴해 주었다고 해 보겠습니다. 그럼 아래와 같이 실행 될 겁니다.

```
>>> query = 'puppy'
>>> key_value_func(query)
2.823 # = .1 / (.9 + .7 + .1) * 9 + .9 / (.9 + .7 + .1) * 2 + .7 / (.9 + .7 + .1
```

2.823 라는 값이 나왔습니다. 강아지와 고양이, 그리고 컴퓨터의 유사도의 비율에 따른 dic의 값의 비율을 지녔다라고 볼 수 있습니다. is same 함수를 쓸 때에는 두 값이 같은지 if문을 통해 검사하고 값을 할당했기 때문에, 미분을 할 수 없거나 할 수 있더라도 gradient가 0이 됩니다. 하지만, 이제 우리는 key_value_func을 미분 할 수 있습니다.

Differentiable Key-Value Vector function

- 만약, dic 의 value에는 100차원의 vector로 들어있었다면 어떻게 될까요?
- 거기에, query와 key 값 모두 vector라면 어떻게 될까요? 즉, Word Embedding Vector라면?
- how similar 함수는 이 vector 들 간의 cosine similarity를 반환 해 주는 함수라면?
- 그리고, dic 의 key 값과 value 값이 서로 같다면 어떻게 될까요?

그럼 다시 가상의 함수를 만들어보겠습니다. word2vec 함수는 단어를 입력으로 받아서 그 단어에 해당하는 미리 정해진 word embedding vector를 반환 해 준다고 가정하겠습니다. 그럼 좀 전의 how similar 함수는 두 vector 간의 cosine similarity 값을 반환 할 겁니다.

```
def key_value_func(query):
    weights = []

    for key in dic.keys():
        weights += [how_similar(key, query)]    # cosine similarity        .

    weights = softmax(weights)    #   weight       softmax    .
    answer = 0

    for weight, value in zip(weights, dic.values()):
        answer += weight * value

    return answer
```

이번에 key_value_func는 그럼 그 값을 받아서 weights에 저장 한 후, 모든 weights의 값이 채워지면 softmax를 취할 겁니다. 여기서 softmax는 weights의 합의 크기를 1로 고정시키는 normalization의 역할을 합니다. 따라서 similarity의 총 합에서 차지하는 비율 만큼 weight의 값이 채워질 겁니다.

```
>>> len(word2vec('computer'))
100
>>> word2vec('dog')
[0.1, 0.3, -0.7, 0.0, ...
>>> word2vec('cat')
[0.15, 0.2, -0.3, 0.8, ...
```

```
>>> dic = {word2vec('computer'): word2vec('computer'), word2vec('dog'): word2vec(
>>>
>>> query = 'puppy'
>>> answer = key_value_func(word2vec(query))
```

그럼 이제 answer의 값에는 어떤 vector 값이 들어 있을 겁니다. 그 vector는 'puppy' vector와 'dog', 'computer', 'cat' vector들의 코사인 유사도에 따라서 값이 정해집니다.

즉, 이 함수는 query와 비슷한 key 값을 찾아서 비슷한 정도에 따라서 weight를 나누고, 각 key의 value값을 weight 값 만큼 가져와서 모두 더하는 것 입니다. 이것이 Attention이 하는 역할 입니다.

Attention for Machine Translation task

Overview

그럼 번역에서 attention은 어떻게 작용할까요? 번역 과정에서는 encoder의 각 time-step 별 출력을 Key와 Value로 삼고, 현재 time-step의 decoder 출력을 Query로 삼아 attention을 취합니다.

- Query: 현재 time-step의 decoder output
- Keys: 각 time-step 별 encoder output
- Values: 각 time-step 별 encoder output

```
>>> context_vector = attention(query = decoder_output, keys = encoder_outputs, val
```

Attention을 추가한 seq2seq의 수식은 아래와 같은부분이 추가/수정 됩니다.

$$w = softmax(h_t^{tgt^T} W \cdot H^{src})$$
$$c = H^{src} \cdot w \quad \textit{and } c \textit{ is a context vector}$$
$$\tilde{h}_t^{tgt} = \tanh(linear_{2hs \to hs}([h_t^{tgt}; c]))$$
$$\hat{y}_t = softmax(linear_{hs \to |V_{tgt}|}(\tilde{h}_t^{tgt}))$$
$$\textit{where } hs \textit{ is hidden size of } RNN, \textit{ and } |V_{tgt}| \textit{ is size of output vocabulary.}$$

원하는 정보를 attention을 통해 encoder에서 획득한 후, 해당 정보를 decoder output과 concatenate하여 $tanh$를 취한 후, softmax 계산을 통해 다음 time-step의 입력이 되는 $\hat{y}_t$을 구합니다.

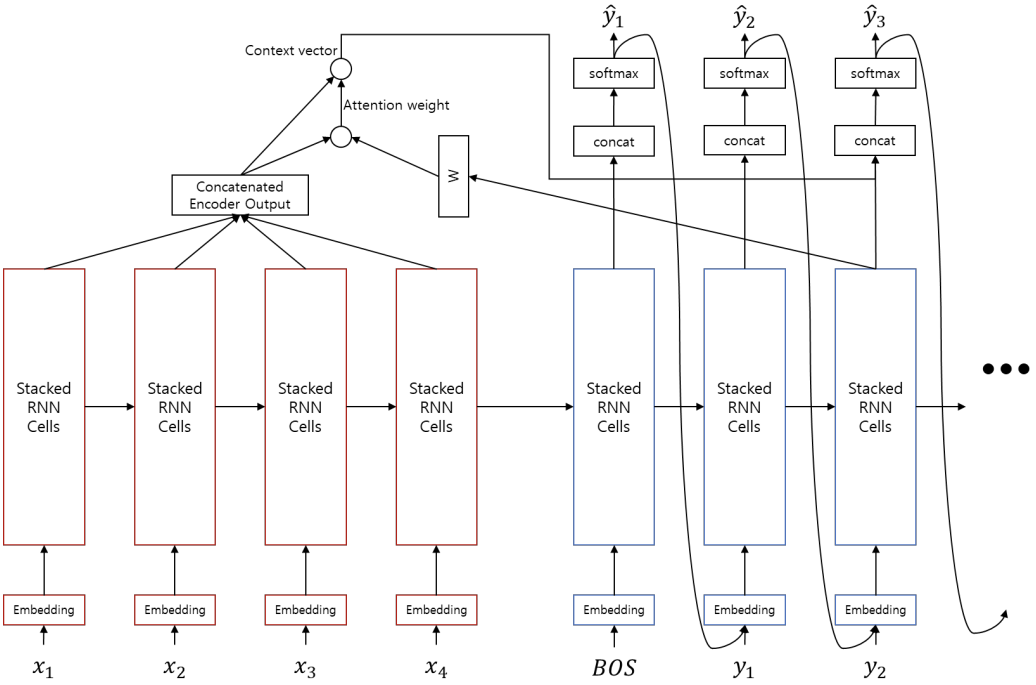
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
Context vector
Attention weight
softmax
softmax
softmax
concat
concat
concat
W
Concatenated Encoder Output
Stacked RNN Cells
Embedding
x_1
x_2
x_3
x_4
BOS
y_1
y_2

Figure 3:

Linear Transformation

이때, 각 input parameter들은 다음을 의미한다고 볼 수 있습니다.

1. decoder_output: 현재 time-step 까지 번역 된 target language 단어들 또는 문장, 의미
2. encoder_outputs: 각 time-step 에서의 source language 단어 또는 문장, 의미

사실 신경망 내부의 각 차원들은 숨겨진 특징값(latent feature)이므로 딱 잘라 정의할 수 없습니다. 하지만 분명한건, source 언어와 target 언어가 다르다는 것입니다. 따라서 단순히 dot product를 해 주기보단 source 언어와 target 언어 간에 연결고리를 하나 놓아주어야 합니다. 그래서 우리는 두 언어의 embedding hyper plane이 선형(linear) 관계에 있다고 가정하고, dot product 하기 전에 선형 변환(linear transformation)을 해 줍니다.

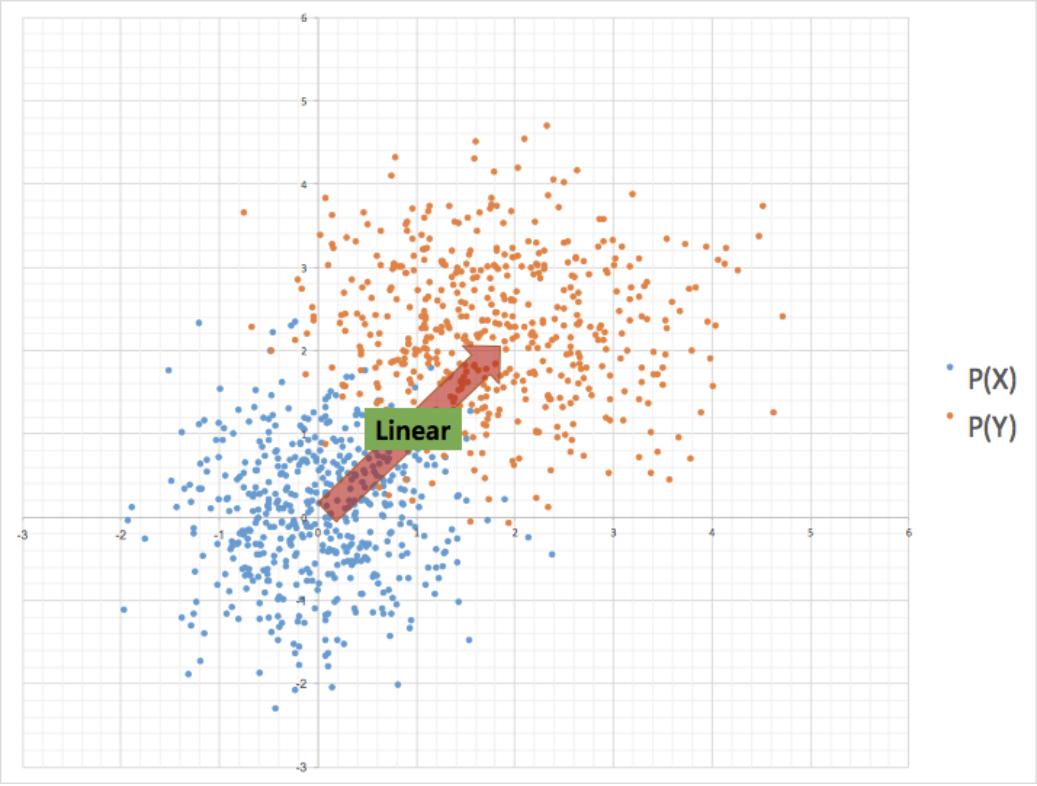

Figure 4:

위와 같이 꼭 번역이 아니더라도, 두 다른 도메인(domain) 사이의 비교를 위해서

사용합니다.

Why

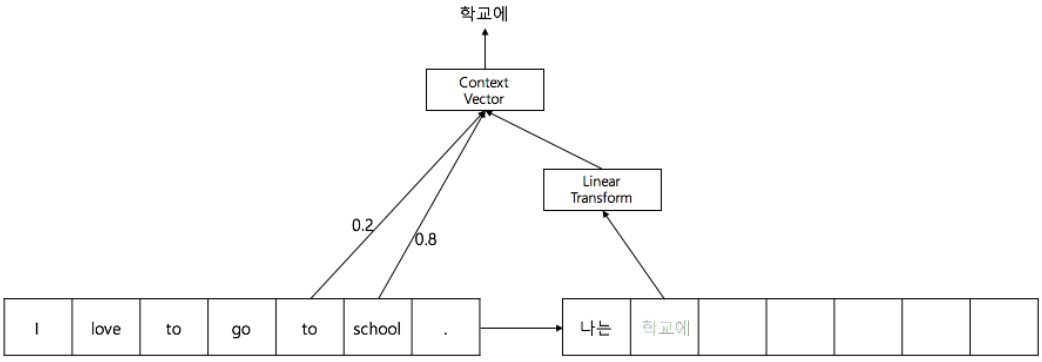

Figure 5:

왜 Attention이 필요한 것일까요? 기존의 seq2seq는 두 개의 RNN(encoder와 decoder)로 이루어져 있습니다. 여기서 압축된 문장의 의미에 해당하는 encoder의 정보를 hidden state (LSTM의 경우에는 + cell state)의 vector로 전달해야 합니다. 그리고 decoder는 그 정보를 이용해 다시 새로운 문장을 만들어냅니다. 이 때, hidden state만으로는 문장의 정보를 완벽하게 전달하기 힘들기 때문입니다. 따라서 decoder의 각 time-step 마다, 시간을 뛰어넘어, hidden state의 정보에 따라 필요한 encoder의 정보에 접근하여 끌어다 쓰겠다는 것 입니다.

Evaluation

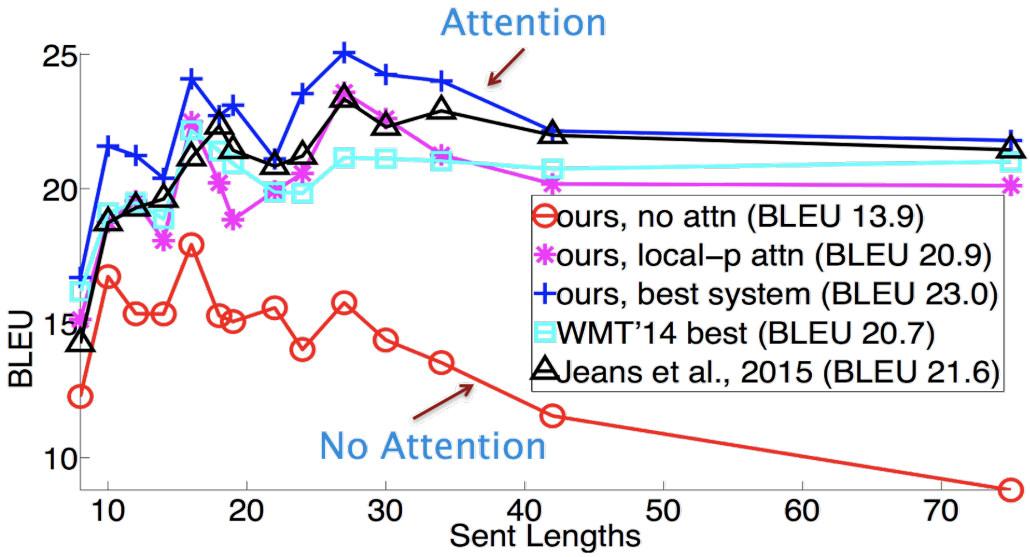

Image from CS224n

Attention을 사용하지 않은 seq2seq는 전반적으로 성능이 떨어짐을 알수 있을 뿐만 아니라, 특히 문장이 길어질수록 성능이 더욱 하락함을 알 수 있습니다. 하지만 이에 비해서 attention을 사용하면 문장이 길어지더라도 성능이 크게 하락하지 않음을 알 수 있습니다.

Code

```
class Attention(nn.Module):

    def __init__(self, hidden_size):
        super(Attention, self).__init__()

        self.linear = nn.Linear(hidden_size, hidden_size, bias = False)
        self.softmax = nn.Softmax(dim = -1)

    def forward(self, h_src, h_t_tgt, mask = None):
        # |h_src| = (batch_size, length, hidden_size)
        # |h_t_tgt| = (batch_size, 1, hidden_size)
```

```
        # |mask| = (batch_size, length)

        query = self.linear(h_t_tgt.squeeze(1)).unsqueeze(-1)
        # |query| = (batch_size, hidden_size, 1)

        weight = torch.bmm(h_src, query).squeeze(-1)
        # |weight| = (batch_size, length)
        if mask is not None:
            # Set each weight as -inf, if the mask value equals to 1.
            # Since the softmax operation makes -inf to 0, masked weights would
            # Thus, if the sample is shorter than other samples in mini-batch, t
            weight.masked_fill_(mask, -float('inf'))
        weight = self.softmax(weight)

        context_vector = torch.bmm(weight.unsqueeze(1), h_src)
        # |context_vector| = (batch_size, 1, hidden_size)

        return context_vector
```

Input Feeding

Overview

Decoder output과 Attention 결과값을 concatenate한 이후에 Generator 모듈에서 softmax를 취하여 $\hat{y}_t$을 구합니다. 하지만 이러한 softmax 과정에서 많은 정보(예를 들어 attention 정보 등)가 손실됩니다. 따라서 단순히 다음 time-step에 $\hat{y}_t$을 feeding 하는 것보다, concatenation layer의 출력도 같이 feeding 해주면 정보의 손실 없이 더 좋은 효과를 얻을 수 있습니다.

y와 달리 concatenation layer의 출력은 y가 embedding layer에서 dense vector(=embedding vector)로 변환되고 난 이후에 embedding vector와 concatenate되어 decoder RNN에 입력으로 주어지게 됩니다. 이러한 과정을 _input feeding_이라고 합니다.

$$h_t^{src} = RNN_{enc}(emb_{src}(x_t), h_{t-1}^{src})$$

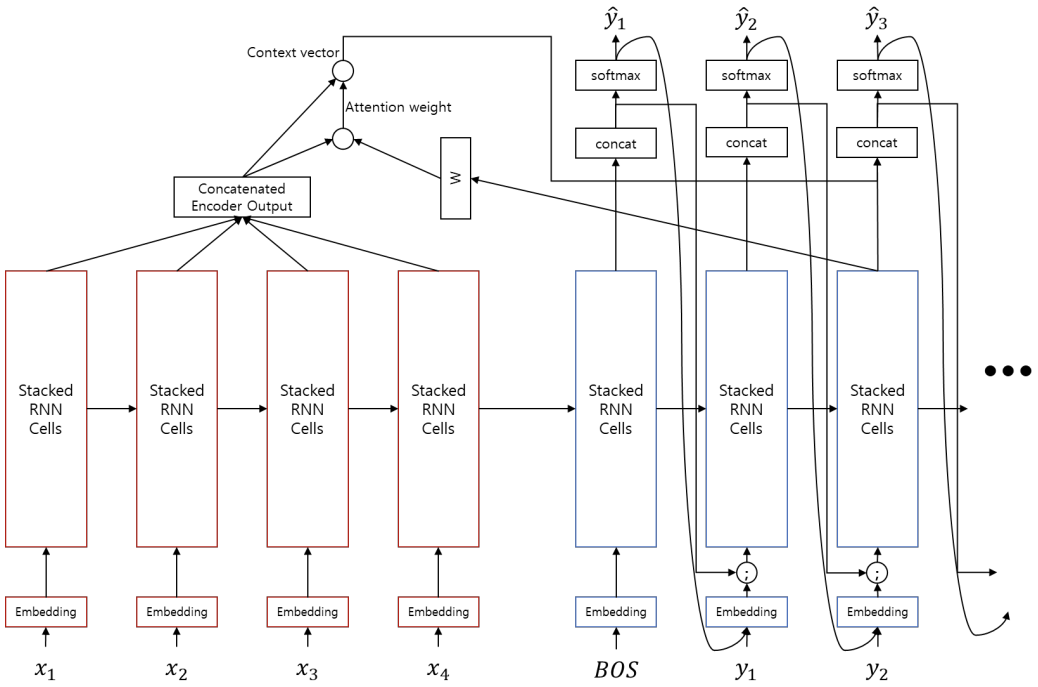
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
Context vector
Attention weight
softmax
softmax
softmax
concat
concat
concat
W
Concatenated Encoder Output
Stacked RNN Cells
Stacked RNN Cells
Stacked RNN Cells
Stacked RNN Cells
Stacked RNN Cells
Stacked RNN Cells
Stacked RNN Cells
Embedding
Embedding
Embedding
Embedding
Embedding
Embedding
Embedding
x_1
x_2
x_3
x_4
BOS
y_1
y_2

Figure 6:

$$H^{src} = [h_1^{src}; h_2^{src}; \cdots ; h_n^{src}]$$

$$h_t^{tgt} = RNN_{dec}([emb_{tgt}(y_{t-1}); \tilde{h}_{t-1}^{tgt}], h_{t-1}^{tgt}) \;\; where\ h_0^{tgt} = h_n^{src} and\ y_0 = BOS$$

$$w = softmax(h_t^{tgtT} W \cdot H^{src})$$

$$c = H^{src} \cdot w \quad \quad and\ c\ is\ a\ context\ vector$$

$$\tilde{h}_t^{tgt} = \tanh(linear_{2hs \to hs}([h_t^{tgt}; c]))$$

$$\hat{y}_t = softmax(linear_{hs \to |V_{tgt}|}(\tilde{h}_t^{tgt}))$$

$$where\ hs\ is\ hidden\ size\ of\ RNN,\ and\ |V_{tgt}|\ is\ size\ of\ output\ vocabulary.$$

위의 수식은 attention과 input feeding이 추가된 seq2seq의 처음부터 끝까지 입니다. RNN_{dec}는 이제 $\tilde{h}_{t-1}^{tgt}$를 입력으로 받기 때문에, 모든 time-step을 한번에 처리하도록 구현할 수 없다는 점이 구현상의 차이점입니다.

Disadvantage

이 방식은 훈련 속도 저하라는 단점을 가집니다. input feeding이전 방식에서는 훈련 할 때에는 teacher forcing 방식이기 때문에(모든 입력을 알고 있기 때문에), encoder와 마찬가지로 decoder도 모든 time-step에 대해서 한번에 feed-forward 작업이 가능했습니다. 하지만 input feeding으로 인해, decoder RNN의 input으로 이전 time-step의 결과가 필요하게 되어, 다시 추론(inference)할 때 처럼 auto-regressive 속성으로 인해 각 time-step 별로 순차적으로 계산을 해야 합니다.

하지만 이 단점이 크게 부각되지 않는 이유는 어차피 추론(inference)단계에서는 decoder는 input feeding이 아니더라도 time-step 별로 순차적으로 계산되어야 하기 때문입니다. 추론 단계에서는 이전 time-step의 output인 $\hat{y}_t$를 decoder(정확하게는 decoder 이전의 embedding layer)의 입력으로 사용해야 하기 때문에, 어쩔 수 없이 병렬처리가 아닌 순차적으로 계산해야 합니다. 따라서 추론 할 때, input feeding으로 인한 속도 저하는 거의 없습니다.

Evaluation

NMT system	Perplexity	BLEU
Base	10.6	11.3
Base + reverse	9.9	12.6(+1.3)

NMT system	Perplexity	BLEU
Base + reverse + dropout	8.1	14.0(+1.4)
Base + reverse + dropout + attention	7.3	16.8(+2.8)
Base + reverse + dropout + attention + feed input	6.4	18.1(+1.3)

WMT'14 English-German results Perplexity(PPL) and BLEU [Loung, arXiv 2015]

현재 방식을 처음 제안한 [Loung et al.2015] Effective Approaches to Attention-based Neural Machine Translation에서는 실험 결과를 위와 같이 주장하였습니다. 실험 대상은 아래와 같습니다.

- Baseline: 기본적인 seq2seq 모델
- Reverse: Bi-directional LSTM을 encoder에 적용
- Dropout: probability 0.2
- Global Attention
- Input Feeding

우리는 이 실험에서 attention과 input feeding을 사용함으로써, 훨씬 더 나은 성능을 얻을 수 있음을 알 수 있습니다.

Code

NMT를 목표로하는 seq2seq를 PyTorch로 구현하는 방법을 소개합니다. 이번 챕터에서 사용될 전체 코드는 저자의 깃허브에서 다운로드 할 수 있습니다. (업데이트 여부에 따라 코드가 약간 다를 수 있습니다.)

- github repo url: https://github.com/kh-kim/simple-nmt

Decoder Class

```
class Decoder(nn.Module):

    def __init__(self, word_vec_dim, hidden_size, n_layers = 4, dropout_p = .2):
        super(Decoder, self).__init__()

        # Be aware of value of 'batch_first' parameter and 'bidirectional' param
        self.rnn = nn.LSTM(word_vec_dim + hidden_size, hidden_size, num_layers = n
```

```
    def forward(self, emb_t, h_t_1_tilde, h_t_1):
        # |emb_t| = (batch_size, 1, word_vec_dim)
        # |h_t_1_tilde| = (batch_size, 1, hidden_size)
        # |h_t_1[0]| = (n_layers, batch_size, hidden_size)
        batch_size = emb_t.size(0)
        hidden_size = h_t_1[0].size(-1)

        if h_t_1_tilde is None:
            # If this is the first time-step,
            h_t_1_tilde = emb_t.new(batch_size, 1, hidden_size).zero_()

        # Input feeding trick.
        x = torch.cat([emb_t, h_t_1_tilde], dim = -1)

        # Unlike encoder, decoder must take an input for sequentially.
        y, h = self.rnn(x, h_t_1)

        return y, h
```

Sequence-to-Sequence

Initialization

```
    def __init__(self, input_size, word_vec_dim, hidden_size, output_size, n_layer
        self.input_size = input_size
        self.word_vec_dim = word_vec_dim
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p

        super(Seq2Seq, self).__init__()

        self.emb_src = nn.Embedding(input_size, word_vec_dim)
        self.emb_dec = nn.Embedding(output_size, word_vec_dim)
```

```
        self.encoder = Encoder(word_vec_dim, hidden_size, n_layers = n_layers, dro
        self.decoder = Decoder(word_vec_dim, hidden_size, n_layers = n_layers, dro
        self.attn = Attention(hidden_size)

        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.tanh = nn.Tanh()
        self.generator = Generator(hidden_size, output_size)
```

Mask Generation

```
    def generate_mask(self, x, length):
        mask = []

        max_length = max(length)
        for l in length:
            if max_length - l > 0:
                # If the length is shorter than maximum length among samples,
                # set last few values to be 1s to remove attention weight.
                mask += [torch.cat([x.new_ones(1, l).zero_(), x.new_ones(1, (max_l
            else:
                # If the length of the sample equals to maximum length among sam
                # set every value in mask to be 0.
                mask += [x.new_ones(1, l).zero_()]

        mask = torch.cat(mask, dim = 0).byte()

        return mask
```

Convert Hidden State from Encoder to Decoder

```
    def merge_encoder_hiddens(self, encoder_hiddens):
        new_hiddens = []
        new_cells = []

        hiddens, cells = encoder_hiddens

        # i-th and (i+1)-th layer is opposite direction.
```

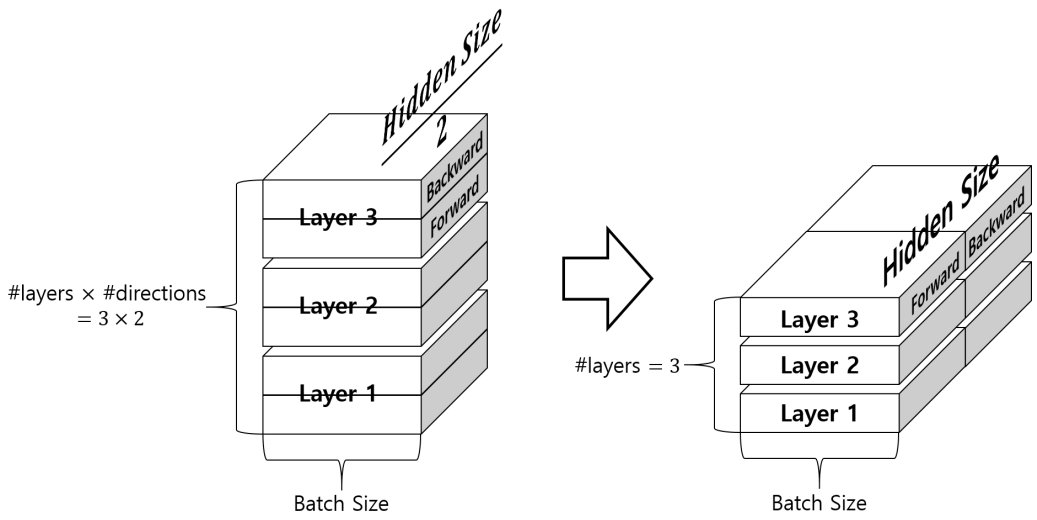

Figure 7:

```
        # Also, each direction of layer is half hidden size.
        # Therefore, we concatenate both directions to 1 hidden size layer.
        for i in range(0, hiddens.size(0), 2):
            new_hiddens += [torch.cat([hiddens[i], hiddens[i + 1]], dim = -1)]
            new_cells += [torch.cat([cells[i], cells[i + 1]], dim = -1)]

        new_hiddens, new_cells = torch.stack(new_hiddens), torch.stack(new_cells)

        return (new_hiddens, new_cells)
```

Forward

```
    def forward(self, src, tgt):
        batch_size = tgt.size(0)

        mask = None
        x_length = None
        if isinstance(src, tuple):
            x, x_length = src
            # Based on the length information, gererate mask to prevent that sho
```

```
        mask = self.generate_mask(x, x_length)
        # |mask| = (batch_size, length)
    else:
        x = src

    if isinstance(tgt, tuple):
        tgt = tgt[0]

    # Get word embedding vectors for every time-step of input sentence.
    emb_src = self.emb_src(x)
    # |emb_src| = (batch_size, length, word_vec_dim)

    # The last hidden state of the encoder would be a initial hidden state o
    h_src, h_0_tgt = self.encoder((emb_src, x_length))
    # |h_src| = (batch_size, length, hidden_size)
    # |h_0_tgt| = (n_layers * 2, batch_size, hidden_size / 2)

    # Merge bidirectional to uni-directional
    # We need to convert size from (n_layers * 2, batch_size, hidden_size /
    # Thus, the converting operation will not working with just 'view' metho
    h_0_tgt, c_0_tgt = h_0_tgt
    h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
    c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
    # You can use 'merge_encoder_hiddens' method, instead of using above 3 l
    # 'merge_encoder_hiddens' method works with non-parallel way.
    # h_0_tgt = self.merge_encoder_hiddens(h_0_tgt)

    # |h_src| = (batch_size, length, hidden_size)
    # |h_0_tgt| = (n_layers, batch_size, hidden_size)
    h_0_tgt = (h_0_tgt, c_0_tgt)

    emb_tgt = self.emb_dec(tgt)
    # |emb_tgt| = (batch_size, length, word_vec_dim)
    h_tilde = []

    h_t_tilde = None
    decoder_hidden = h_0_tgt
    # Run decoder until the end of the time-step.
```

```
        for t in range(tgt.size(1)):
            # Teacher Forcing: take each input from training set, not from the l
            # Because of Teacher Forcing, training procedure and inference proce
            # Of course, because of sequential running in decoder, this causes s
            emb_t = emb_tgt[:, t, :].unsqueeze(1)
            # |emb_t| = (batch_size, 1, word_vec_dim)
            # |h_t_tilde| = (batch_size, 1, hidden_size)

            decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decode
            # |decoder_output| = (batch_size, 1, hidden_size)
            # |decoder_hidden| = (n_layers, batch_size, hidden_size)

            context_vector = self.attn(h_src, decoder_output, mask)
            # |context_vector| = (batch_size, 1, hidden_size)

            h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_v
            # |h_t_tilde| = (batch_size, 1, hidden_size)

            h_tilde += [h_t_tilde]

        h_tilde = torch.cat(h_tilde, dim = 1)
        # |h_tilde| = (batch_size, length, hidden_size)

        y_hat = self.generator(h_tilde)
        # |y_hat| = (batch_size, length, output_size)

        return y_hat
```

Seq2Seq Class

```
class Seq2Seq(nn.Module):

    def __init__(self, input_size, word_vec_dim, hidden_size, output_size, n_layer
        self.input_size = input_size
        self.word_vec_dim = word_vec_dim
        self.hidden_size = hidden_size
        self.output_size = output_size
```

```
        self.n_layers = n_layers
        self.dropout_p = dropout_p

        super(Seq2Seq, self).__init__()

        self.emb_src = nn.Embedding(input_size, word_vec_dim)
        self.emb_dec = nn.Embedding(output_size, word_vec_dim)

        self.encoder = Encoder(word_vec_dim, hidden_size, n_layers = n_layers, dro
        self.decoder = Decoder(word_vec_dim, hidden_size, n_layers = n_layers, dro
        self.attn = Attention(hidden_size)

        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.tanh = nn.Tanh()
        self.generator = Generator(hidden_size, output_size)

    def generate_mask(self, x, length):
        mask = []

        max_length = max(length)
        for l in length:
            if max_length - l > 0:
                # If the length is shorter than maximum length among samples,
                # set last few values to be 1s to remove attention weight.
                mask += [torch.cat([x.new_ones(1, l).zero_(), x.new_ones(1, (max_l
            else:
                # If the length of the sample equals to maximum length among sam
                # set every value in mask to be 0.
                mask += [x.new_ones(1, l).zero_()]

        mask = torch.cat(mask, dim = 0).byte()

        return mask

    def merge_encoder_hiddens(self, encoder_hiddens):
        new_hiddens = []
        new_cells = []
```

```
        hiddens, cells = encoder_hiddens

        # i-th and (i+1)-th layer is opposite direction.
        # Also, each direction of layer is half hidden size.
        # Therefore, we concatenate both directions to 1 hidden size layer.
        for i in range(0, hiddens.size(0), 2):
            new_hiddens += [torch.cat([hiddens[i], hiddens[i + 1]], dim = -1)]
            new_cells += [torch.cat([cells[i], cells[i + 1]], dim = -1)]

        new_hiddens, new_cells = torch.stack(new_hiddens), torch.stack(new_cells)

        return (new_hiddens, new_cells)

    def forward(self, src, tgt):
        batch_size = tgt.size(0)

        mask = None
        x_length = None
        if isinstance(src, tuple):
            x, x_length = src
            # Based on the length information, gererate mask to prevent that sho
            mask = self.generate_mask(x, x_length)
            # |mask| = (batch_size, length)
        else:
            x = src

        if isinstance(tgt, tuple):
            tgt = tgt[0]

        # Get word embedding vectors for every time-step of input sentence.
        emb_src = self.emb_src(x)
        # |emb_src| = (batch_size, length, word_vec_dim)

        # The last hidden state of the encoder would be a initial hidden state o
        h_src, h_0_tgt = self.encoder((emb_src, x_length))
        # |h_src| = (batch_size, length, hidden_size)
        # |h_0_tgt| = (n_layers * 2, batch_size, hidden_size / 2)
```

```
        # Merge bidirectional to uni-directional
        # We need to convert size from (n_layers * 2, batch_size, hidden_size /
        # Thus, the converting operation will not working with just 'view' metho
        h_0_tgt, c_0_tgt = h_0_tgt
        h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
        c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
        # You can use 'merge_encoder_hiddens' method, instead of using above 3 l
        # 'merge_encoder_hiddens' method works with non-parallel way.
        # h_0_tgt = self.merge_encoder_hiddens(h_0_tgt)

        # |h_src| = (batch_size, length, hidden_size)
        # |h_0_tgt| = (n_layers, batch_size, hidden_size)
        h_0_tgt = (h_0_tgt, c_0_tgt)

        emb_tgt = self.emb_dec(tgt)
        # |emb_tgt| = (batch_size, length, word_vec_dim)
        h_tilde = []

        h_t_tilde = None
        decoder_hidden = h_0_tgt
        # Run decoder until the end of the time-step.
        for t in range(tgt.size(1)):
            # Teacher Forcing: take each input from training set, not from the l
            # Because of Teacher Forcing, training procedure and inference proce
            # Of course, because of sequential running in decoder, this causes s
            emb_t = emb_tgt[:, t, :].unsqueeze(1)
            # |emb_t| = (batch_size, 1, word_vec_dim)
            # |h_t_tilde| = (batch_size, 1, hidden_size)

            decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decode
            # |decoder_output| = (batch_size, 1, hidden_size)
            # |decoder_hidden| = (n_layers, batch_size, hidden_size)

            context_vector = self.attn(h_src, decoder_output, mask)
            # |context_vector| = (batch_size, 1, hidden_size)

            h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_v
            # |h_t_tilde| = (batch_size, 1, hidden_size)
```

```
            h_tilde += [h_t_tilde]

        h_tilde = torch.cat(h_tilde, dim = 1)
        # |h_tilde| = (batch_size, length, hidden_size)

        y_hat = self.generator(h_tilde)
        # |y_hat| = (batch_size, length, output_size)

        return y_hat
```

Auto-regressive and Teacher Focing

많은 분들이 여기까지 잘 따라왔다면 궁금증을 하나 가질 수 있습니다. Decoder의 입력으로 이전 time-step의 출력이 들어가는것이 훈련 때도 같은 것인가? 사실, 안타깝게도 seq2seq의 기본적인 훈련(training) 방식은 추론(inference)할 때의 방식과 상이합니다.

Auto-regressive

Sequence-to-sequence의 훈련 방식과 추론 방식의 차이는 근본적으로 auto-regressive라는 속성 때문에 생겨납니다. Auto-regressive는 과거의 자신의 값을 참조하여 현재의 값을 추론(또는 예측) 해 내기 때문에 붙은 이름입니다. 이는 수식에서도 확인 할 수 있습니다. 아래는 전체적인 신경망 기계번역의 수식 입니다.

$$Y = argmax_Y P(Y|X) = argmax_Y \prod_{i=1}^{n} P(y_i|X, y_{<i})$$

$$or$$

$$y_i = argmax_y P(y|X, y_{<i})$$
$$where\ y_0 = BOS.$$

위와 같이 현재 time-step의 출력값 y_t는 encoder의 입력 문장(또는 시퀀스) X와 이전 time-step까지의 $y_{<t}$를 조건부로 받아 결정 되기 때문에, 과거 자신의 값을 참조하게 되는 것 입니다.

이것은 과거에 잘못된 예측을 하게 되면 점점 시간이 지날수록 더 큰 잘못된 예측을 할 가능성을 야기하기도 합니다. 또한, 과거의 결과값에 따라 문장(또는 시퀀스)의 구성이 바뀔 뿐만 아니라, 그 길이 마저도 바뀌게 됩니다. 따라서 우리는 이런 auto-regressive 속성을 유지한 채 훈련을 할 수 없습니다.

$$\hat{y}_t = argmax_y P(y|X, y_{<t}; \theta) \, where \, X = \{x_1, x_2, \cdots, x_n\} \, and \, Y = \{y_0, y_1, \cdots, y_{m+1}\}$$

훈련 할 때에 각 time-step 별 수식은 다음과 같습니다. 위와 같이 조건부에 $\hat{y}_{<t}$가 들어가는 것이 아닌, $y_{<t}$가 들어가는 것이기 때문에, 훈련시에는 이전 time-step의 출력 $\hat{y}_{<t}$을 현재 time-step의 입력으로 넣어줄 수 없습니다. 만약 넣어주게 된다면 현재 time-step의 decoder에겐 잘못된 것을 가르쳐 주는 꼴이 될 것입니다.

$$\mathcal{L}(Y) = -\sum_{i=1}^{m+1} \log P(y_i|X, y_{<i}; \theta)$$

$$\theta \leftarrow \theta - \lambda \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(Y_i)$$

또한, 실제 손실함수(loss function)을 계산하여 gradient descent를 수행할 때도, $\hat{y}_i$의 확률을 사용하지 않고, softmax layer에서 정답에 해당하는 y_i의 인덱스(index)에 있는 로그(log)확률값을 사용 합니다.

Teacher Forcing

따라서 우리는 Teacher Forcing이라고 불리는 방법을 사용하여 훈련 합니다.

중요한 점은 **훈련(training)시에는 decoder의 입력으로 이전 time-step의 decoder의 출력값이 아닌, 실제 Y가 들어간다**는 것입니다. 하지만, 추론(inference) 할 때에는 실제 Y를 모르기 때문에, 이전 time-step에서 계산되어 나온 $\hat{y_{t-1}}$를 decoder의 입력으로 사용합니다. 이 훈련 방법을 **Teacher Forcing**이라고 합니다.

추론(inference) 할 때에는 auto-regressive 속성 때문에 과거 자신을 참조해야 합니다. 따라서 각 time-step 별로 순차적(sequential)으로 진행해야 합니다. 하지만 훈련(training) 할 때에는 입력값이 정해져 있으므로, 모든 time-step을 한번에 계산할 수 있습니다. 그러므로 decoder도 모든 time-step을 합쳐 수식을 정리할 수 있습니다.

$$H^{tgt} = RNN_{dec}(emb_{tgt}([BOS; Y[:-1]]), h_n^{src})$$

이런 auto-regressive 속성 및 teacher forcing 방법은 신경망 언어모델(NNLM)에도 똑같이 적용되는 문제 입니다. 하지만 언어모델의 경우에는 perplexity는 문장의 확률과 직접적으로 연관이 있기 때문에, 큰 문제가 되지 않는 반면에 기계번역에서는 좀 더 큰 문제로 다가옵니다. 이에 대해서는 추후 다루도록 하겠습니다. # Inference

Overview

이제까지 X와 Y가 모두 주어진 훈련상황을 가정하였습니다만, 이제부터는 X만 주어진 상태에서 $\hat{Y}$을 예측하는 방법에 대해서 서술하겠습니다. 이러한 과정을 우리는 Inference 또는 Search 라고 부릅니다. 우리가 기본적으로 이 방식을 search라고 부르는 이유는 search 알고리즘에 기반하기 때문입니다. 결국 우리가 원하는 것은 state로 이루어진 단어(word) 사이에서 최고의 확률을 갖는 path를 찾는 것이기 때문입니다.

Sampling

사실 먼저 우리가 생각할 수 있는 가장 정확한 방법은 각 time-step별 $\hat{y}_t$를 고를 때, 마지막 **softmax** layer에서의 확률 분포(probability distribution)대로 sampling을 하는 것 입니다. 그리고 다음 time-step에서 그 선택($\hat{y}_t$)을 기반으로 다음 $\hat{y}_{t+1}$을 또 다시 sampling하여 최종적으로 EOS가 나올 때 까지 sampling을 반복하는 것 입니다. 이렇게 하면 우리가 원하는 $P(Y|X)$ 에 가장 가까운 형태의 번역이 완성될 겁니다. 하지만, 이러한 방식은 같은 입력에 대해서 매번 다른 출력 결과물을 만들어낼 수 있습니다. 따라서 우리가 원하는 형태의 결과물이 아닙니다.

Gready Search

우리는 자료구조, 알고리즘 수업에서 수 많은 search 방법에 대해 배웠습니다. DFS, BFS, Dynamic Programming 등. 우리는 이 중에서 Greedy algorithm을 기반으로 search를 구현합니다. 즉, softmax layer에서 가장 값이 큰 index를 뽑아 해당 time-step의 $\hat{y}_t$로 사용하게 되는 것 입니다.

Code

아래의 코드는 sampling과 greedy search를 위한 코드 입니다. Encoder가 동작하는 부분까지는 완전히 똑같습니다. 다만, 이후 inference(추론)를 위한 부분은 기존 훈련 코드와 상이합니다. Teacher Forcing을 사용하였던 훈련 방식(실제 정답 $ y_{t - 1} $을 $ t $ time-step의 입력으로 사용함)과 달리, 실제 이전 time-step의 출력을 현재 time-step의 입력으로 사용 합니다.

```
    def search(self, src, is_greedy = True, max_length = 255):
        mask = None
        x_length = None
        if isinstance(src, tuple):
            x, x_length = src
            mask = self.generate_mask(x, x_length)
        else:
            x = src
        batch_size = x.size(0)

        emb_src = self.emb_src(x)
        h_src, h_0_tgt = self.encoder((emb_src, x_length))
        h_0_tgt, c_0_tgt = h_0_tgt
        h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
        c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
        h_0_tgt = (h_0_tgt, c_0_tgt)

        # Fill a vector, which has 'batch_size' dimension, with BOS value.
        y = x.new(batch_size, 1).zero_() + data_loader.BOS
        is_undone = x.new_ones(batch_size, 1).float()
        decoder_hidden = h_0_tgt
        h_t_tilde, y_hats, indice = None, [], []

        # Repeat a loop while sum of 'is_undone' flag is bigger than 0, or curre
        while is_undone.sum() > 0 and len(indice) < max_length:
            # Unlike training procedure, take the last time-step's output during
            emb_t = self.emb_dec(y)
            # |emb_t| = (batch_size, 1, word_vec_dim)

            decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decode
```

```
            context_vector = self.attn(h_src, decoder_output, mask)
            h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_v
            y_hat = self.generator(h_t_tilde)
            # |y_hat| = (batch_size, 1, output_size)
            y_hats += [y_hat]

            if is_greedy:
                y = torch.topk(y_hat, 1, dim = -1)[1].squeeze(-1)
            else:
                # Take a random sampling based on the multinoulli distribution.
                y = torch.multinomial(y_hat.exp().view(batch_size, -1), 1)
            # Put PAD if the sample is done.
            y = y.masked_fill_((1. - is_undone).byte(), data_loader.PAD)
            is_undone = is_undone * torch.ne(y, data_loader.EOS).float()
            # |y| = (batch_size, 1)
            # |is_undone| = (batch_size, 1)
            indice += [y]

        y_hats = torch.cat(y_hats, dim = 1)
        indice = torch.cat(indice, dim = -1)
        # |y_hat| = (batch_size, length, output_size)
        # |indice| = (batch_size, length)

        return y_hats, indice
```

Beam Search

하지만 우리는 자료구조, 알고리즘 수업에서 배웠다시피, greedy algorithm은 굉장히 쉽고 간편하지만, 최적의(optimal) 해를 보장하지 않습니다. 따라서 최적의 해에 가까워지기 위해서 우리는 약간의 trick을 첨가합니다. Beam Size 만큼의 후보를 더 tracking 하는 것 입니다.

현재 time-step에서 Top-k개를 뽑아서 (여기서 k는 beam size와 같습니다) 다음 time-step에 대해서 k번 inference를 수행합니다. 그리고 총 $k*|V|$ 개의 softmax 결과값 중에서 다시 top-k개를 뽑아 다음 time-step으로 넘깁니다. ($|V|$는 Vocabulary size) 여기서 중요한 점은 두가지 입니다.

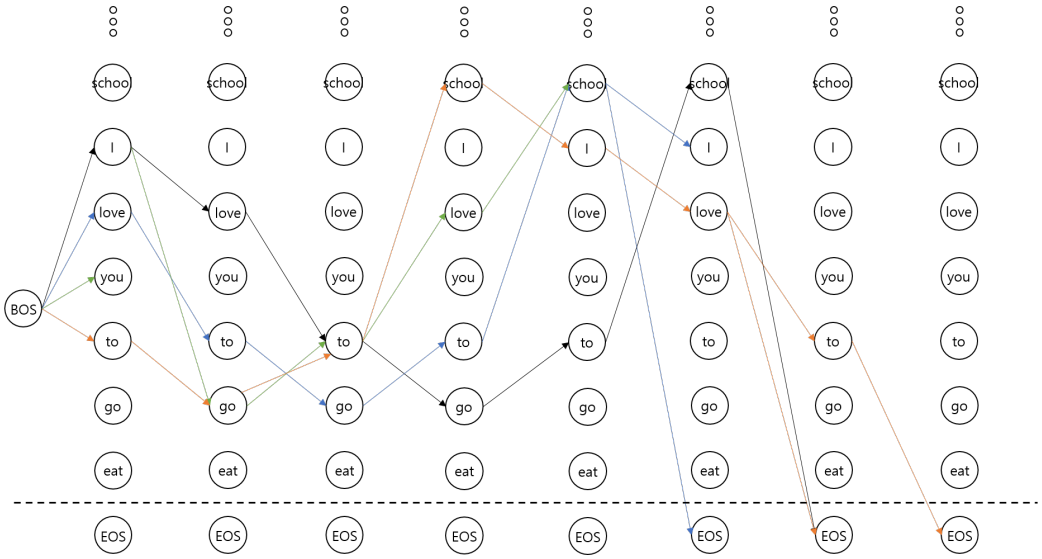

Figure 8:

$$\hat{y}_t^k = argmax_{k-th}\hat{Y}_t$$
$$\hat{Y}_t = f_\theta(X, y_{<t}^1) \cup f_\theta(X, y_{<t}^2) \cup \cdots \cup f_\theta(X, y_{<t}^k)$$

$$X = \{x_1, x_2, \cdots, x_n\}$$

1. 누적 확률을 사용하여 top-k를 뽑습니다. 이때, 보통 로그 확률을 사용하므로 현재 time-step 까지의 로그확률에 대한 합을 tracking 하고 있어야 합니다.
2. top-k를 뽑을 때, 현재 time-step에 대해 k번 계산한 모든 결과물 중에서 뽑습니다.

Beam Search를 사용하면 좀 더 넓은 path에 대해서 search를 수행하므로 당연히 좀 더 나은 성능을 보장합니다. 하지만, beam size만큼 번역을 더 수행해야 하기 때문에 속도에 저하가 있습니다. 다행히도 우리는 이 작업을 mini-batch로 만들어 수행하기 때문에, 병렬처리로 인해서 약간의 속도저하만 생기게 됩니다.

아래는 [Cho et al.2016]에서 주장한 **Beam Search**의 성능향상에 대한 실험 결과입니다. Sampling 방법은 단순한 Greedy Search 보다 더 좋은 성능을 제공하지만, Beam search가 가장 좋은 성능을 보여줍니다. 특기할 점은 Machine Translation task에서는 보통 beam size를 10이하로 사용한다는 것 입니다.

Strategy	# Chains	Valid Set		Test Set	
		NLL	BLEU	NLL	BLEU
Ancestral Sampling	50	22.98	15.64	26.25	16.76
Greedy Decoding	-	27.88	15.50	26.49	16.66
Beamsearch	5	20.18	17.03	22.81	18.56
Beamsearch	10	19.92	17.13	22.44	18.59

En-Cz: 12m training sentence pairs [Cho, arXiv 2016]

How to implement

하나의

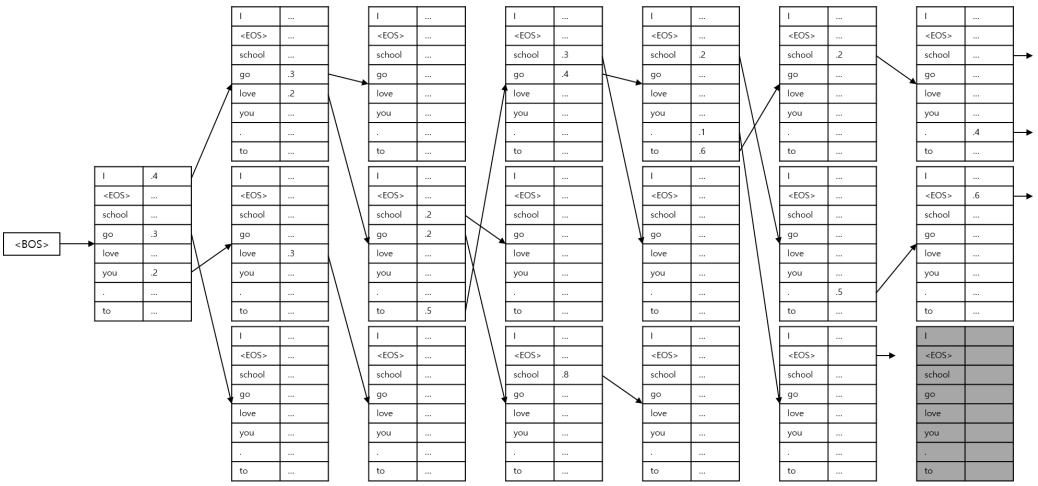

Figure 9:

Length Penalty

위의 search 알고리즘을 직접 짜서 수행시켜 보면 한가지 문제점이 발견됩니다. 현재 time-step 까지의 확률을 모두 곱(로그확률의 경우에는 합)하기 때문에 문장이 길어질수록 확률이 낮아진다는 점 입니다. 따라서 짧은 문장일수록 더 높은 점수를 획득하는

경향이 있습니다. 우리는 이러한 현상을 방지하기 위해서 length penalty를 주어 search가 조기 종료되는 것을 막습니다.

수식은 아래와 같습니다. 불행히도 우리는 2개의 hyper-parameter를 추가해야 합니다. (주의: log확률에 곱하는 것이 맞습니다.)

$$\log \tilde{P}(\hat{Y}|X) = \log P(\hat{Y}|X) * penalty$$

$$penalty = \frac{(1+length)^{\alpha}}{(1+\beta)^{\alpha}}$$

$$where\ \beta\ is\ hyper\ parameter\ of\ minimum\ length.$$

Code

SingleBeamSearchSpace Class

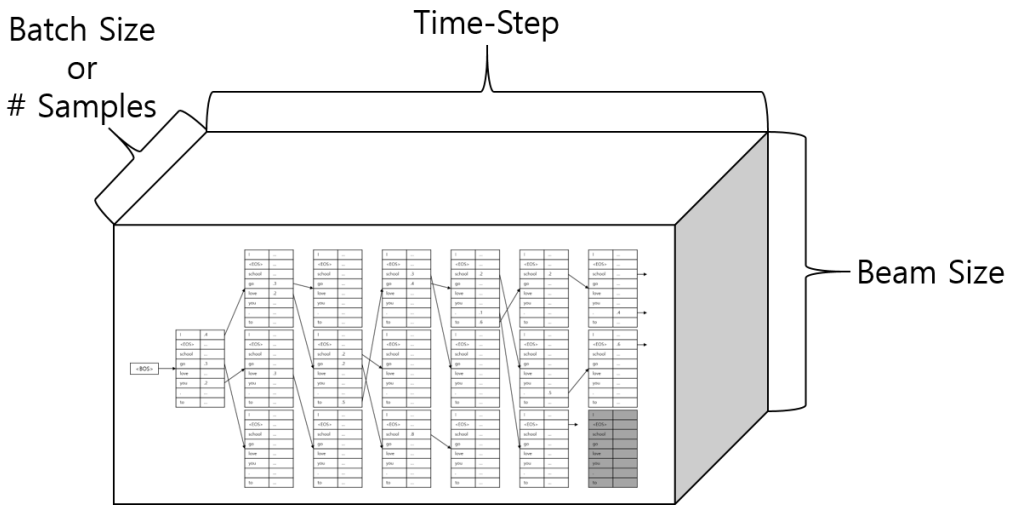

Figure 10:

Initialization

```
from operator import itemgetter
```

```
import torch
import torch.nn as nn

import data_loader

LENGTH_PENALTY = 1.2
MIN_LENGTH = 5

class SingleBeamSearchSpace():

    def __init__(self, hidden, h_t_tilde = None, beam_size = 5, max_length = 255)
        self.beam_size = beam_size
        self.max_length = max_length

        super(SingleBeamSearchSpace, self).__init__()

        # To put data to same device.
        self.device = hidden[0].device
        # Inferred word index for each time-step. For now, initialized with init
        self.word_indice = [torch.LongTensor(beam_size).zero_().to(self.device) +
        # Index origin of current beam.
        self.prev_beam_indice = [torch.LongTensor(beam_size).zero_().to(self.devic
        # Cumulative log-probability for each beam.
        self.cumulative_probs = [torch.FloatTensor([.0] + [-float('inf')] * (beam_
        # 1 if it is done else 0
        self.masks = [torch.ByteTensor(beam_size).zero_().to(self.device)]

        # We don't need to remember every time-step of hidden states: prev_hidde
        # What we need is remember just last one.
        # Future work: make this class to deal with any necessary information fo

        # |hidden[0]| = (n_layers, 1, hidden_size)
        self.prev_hidden = torch.cat([hidden[0]] * beam_size, dim = 1)
        self.prev_cell = torch.cat([hidden[1]] * beam_size, dim = 1)
        # |prev_hidden| = (n_layers, beam_size, hidden_size)
        # |prev_cell| = (n_layers, beam_size, hidden_size)

        # |h_t_tilde| = (batch_size = 1, 1, hidden_size)
```

```
        self.prev_h_t_tilde = torch.cat([h_t_tilde] * beam_size, dim = 0) if h_t_t
        # |prev_h_t_tilde| = (beam_size, 1, hidden_size)

        self.current_time_step = 0
        self.done_cnt = 0

    def get_length_penalty(self, length, alpha = LENGTH_PENALTY, min_length = MIN_
        # Calculate length-penalty, because shorter sentence usually have bigger
        # Thus, we need to put penalty for shorter one.
        p = (1 + length) ** alpha / (1 + min_length) ** alpha

        return p

    def is_done(self):
        # Return 1, if we had EOS more than 'beam_size'-times.
        if self.done_cnt >= self.beam_size:
            return 1
        return 0

    def get_batch(self):
        y_hat = self.word_indice[-1].unsqueeze(-1)
        hidden = (self.prev_hidden, self.prev_cell)
        h_t_tilde = self.prev_h_t_tilde

        # |y_hat| = (beam_size, 1)
        # |hidden| = (n_layers, beam_size, hidden_size)
        # |h_t_tilde| = (beam_size, 1, hidden_size) or None
        return y_hat, hidden, h_t_tilde

    def collect_result(self, y_hat, hidden, h_t_tilde):
        # |y_hat| = (beam_size, 1, output_size)
        # |hidden| = (n_layers, beam_size, hidden_size)
        # |h_t_tilde| = (beam_size, 1, hidden_size)
        output_size = y_hat.size(-1)

        self.current_time_step += 1

        # Calculate cumulative log-probability.
```

```
        # First, fill -inf value to last cumulative probability, if the beam is
        # Second, expand -inf filled cumulative probability to fit to 'y_hat'. (
        # Third, add expanded cumulative probability to 'y_hat'
        cumulative_prob = y_hat + self.cumulative_probs[-1].masked_fill_(self.mask
        # Now, we have new top log-probability and its index. We picked top inde
        # Be aware that we picked top-k from whole batch through 'view(-1)'.
        top_log_prob, top_indice = torch.topk(cumulative_prob.view(-1), self.beam_
        # |top_log_prob| = (beam_size)
        # |top_indice| = (beam_size)

        self.word_indice += [top_indice.fmod(output_size)] # Because we picked fr
        self.prev_beam_indice += [top_indice.div(output_size).long()] # Also, we

        # Add results to history boards.
        self.cumulative_probs += [top_log_prob]
        self.masks += [torch.eq(self.word_indice[-1], data_loader.EOS)] # Set fin
        self.done_cnt += self.masks[-1].float().sum() # Calculate a number of fin

        # Set hidden states for next time-step, using 'index_select' method.
        self.prev_hidden = torch.index_select(hidden[0], dim = 1, index = self.pre
        self.prev_cell = torch.index_select(hidden[1], dim = 1, index = self.prev_
        self.prev_h_t_tilde = torch.index_select(h_t_tilde, dim = 0, index = self.

    def get_n_best(self, n = 1):
        sentences = []
        probs = []
        founds = []

        for t in range(len(self.word_indice)): # for each time-step,
            for b in range(self.beam_size): # for each beam,
                if self.masks[t][b] == 1: # if we had EOS on this time-step and b
                    # Take a record of penaltified log-proability.
                    probs += [self.cumulative_probs[t][b] / self.get_length_penalt
                    founds += [(t, b)]

        # Also, collect log-probability from last time-step, for the case of EOS
        for b in range(self.beam_size):
            if self.cumulative_probs[-1][b] != -float('inf'):
```

```
                if not (len(self.cumulative_probs) - 1, b) in founds:
                    probs += [self.cumulative_probs[-1][b]]
                    founds += [(t, b)]

        # Sort and take n-best.
        sorted_founds_with_probs = sorted(zip(founds, probs),
                                          key = itemgetter(1),
                                          reverse = True
                                          )[:n]
        probs = []

        for (end_index, b), prob in sorted_founds_with_probs:
            sentence = []

            # Trace from the end.
            for t in range(end_index, 0, -1):
                sentence = [self.word_indice[t][b]] + sentence
                b = self.prev_beam_indice[t][b]

            sentences += [sentence]
            probs += [prob]

        return sentences, probs
```

Initialization

```
    def __init__(self, hidden, h_t_tilde = None, beam_size = 5, max_length = 255)
        self.beam_size = beam_size
        self.max_length = max_length

        super(SingleBeamSearchSpace, self).__init__()

        # To put data to same device.
        self.device = hidden[0].device
        # Inferred word index for each time-step. For now, initialized with init
        self.word_indice = [torch.LongTensor(beam_size).zero_().to(self.device) +
        # Index origin of current beam.
```

```
        self.prev_beam_indice = [torch.LongTensor(beam_size).zero_().to(self.devic
        # Cumulative log-probability for each beam.
        self.cumulative_probs = [torch.FloatTensor([.0] + [-float('inf')] * (beam_
        # 1 if it is done else 0
        self.masks = [torch.ByteTensor(beam_size).zero_().to(self.device)]

        # We don't need to remember every time-step of hidden states: prev_hidde
        # What we need is remember just last one.
        # Future work: make this class to deal with any necessary information fo

        # |hidden[0]| = (n_layers, 1, hidden_size)
        self.prev_hidden = torch.cat([hidden[0]] * beam_size, dim = 1)
        self.prev_cell = torch.cat([hidden[1]] * beam_size, dim = 1)
        # |prev_hidden| = (n_layers, beam_size, hidden_size)
        # |prev_cell| = (n_layers, beam_size, hidden_size)

        # |h_t_tilde| = (batch_size = 1, 1, hidden_size)
        self.prev_h_t_tilde = torch.cat([h_t_tilde] * beam_size, dim = 0) if h_t_t
        # |prev_h_t_tilde| = (beam_size, 1, hidden_size)

        self.current_time_step = 0
        self.done_cnt = 0
```

Implement Length Penalty

```
    def get_length_penalty(self, length, alpha = LENGTH_PENALTY, min_length = MIN_
        # Calculate length-penalty, because shorter sentence usually have bigger
        # Thus, we need to put penalty for shorter one.
        p = (1 + length) ** alpha / (1 + min_length) ** alpha

        return p
```

Mark if is done

```
    def is_done(self):
        # Return 1, if we had EOS more than 'beam_size'-times.
        if self.done_cnt >= self.beam_size:
```

```
            return 1
        return 0
```

Generate Fabricated Mini-batch

```
    def get_batch(self):
        y_hat = self.word_indice[-1].unsqueeze(-1)
        hidden = (self.prev_hidden, self.prev_cell)
        h_t_tilde = self.prev_h_t_tilde

        # |y_hat| = (beam_size, 1)
        # |hidden| = (n_layers, beam_size, hidden_size)
        # |h_t_tilde| = (beam_size, 1, hidden_size) or None
        return y_hat, hidden, h_t_tilde
```

Collect the Result and Pick Top-K

```
    def collect_result(self, y_hat, hidden, h_t_tilde):
        # |y_hat| = (beam_size, 1, output_size)
        # |hidden| = (n_layers, beam_size, hidden_size)
        # |h_t_tilde| = (beam_size, 1, hidden_size)
        output_size = y_hat.size(-1)

        self.current_time_step += 1

        # Calculate cumulative log-probability.
        # First, fill -inf value to last cumulative probability, if the beam is
        # Second, expand -inf filled cumulative probability to fit to 'y_hat'. (
        # Third, add expanded cumulative probability to 'y_hat'
        cumulative_prob = y_hat + self.cumulative_probs[-1].masked_fill_(self.mask
        # Now, we have new top log-probability and its index. We picked top inde
        # Be aware that we picked top-k from whole batch through 'view(-1)'.
        top_log_prob, top_indice = torch.topk(cumulative_prob.view(-1), self.beam_
        # |top_log_prob| = (beam_size)
        # |top_indice| = (beam_size)

        self.word_indice += [top_indice.fmod(output_size)] # Because we picked fr
```

```
        self.prev_beam_indice += [top_indice.div(output_size).long()] # Also, we

        # Add results to history boards.
        self.cumulative_probs += [top_log_prob]
        self.masks += [torch.eq(self.word_indice[-1], data_loader.EOS)] # Set fin
        self.done_cnt += self.masks[-1].float().sum() # Calculate a number of fin

        # Set hidden states for next time-step, using 'index_select' method.
        self.prev_hidden = torch.index_select(hidden[0], dim = 1, index = self.pre
        self.prev_cell = torch.index_select(hidden[1], dim = 1, index = self.prev_
        self.prev_h_t_tilde = torch.index_select(h_t_tilde, dim = 0, index = self.
```

Back-trace the History

```
    def get_n_best(self, n = 1):
        sentences = []
        probs = []
        founds = []

        for t in range(len(self.word_indice)): # for each time-step,
            for b in range(self.beam_size): # for each beam,
                if self.masks[t][b] == 1: # if we had EOS on this time-step and b
                    # Take a record of penaltified log-proability.
                    probs += [self.cumulative_probs[t][b] / self.get_length_penalt
                    founds += [(t, b)]

        # Also, collect log-probability from last time-step, for the case of EOS
        for b in range(self.beam_size):
            if self.cumulative_probs[-1][b] != -float('inf'):
                if not (len(self.cumulative_probs) - 1, b) in founds:
                    probs += [self.cumulative_probs[-1][b]]
                    founds += [(t, b)]

        # Sort and take n-best.
        sorted_founds_with_probs = sorted(zip(founds, probs),
                                          key = itemgetter(1),
                                          reverse = True
```

```
                                        )[:n]
    probs = []

    for (end_index, b), prob in sorted_founds_with_probs:
        sentence = []

        # Trace from the end.
        for t in range(end_index, 0, -1):
            sentence = [self.word_indice[t][b]] + sentence
            b = self.prev_beam_indice[t][b]

        sentences += [sentence]
        probs += [prob]

    return sentences, probs
```

Mini-batch Parallelized Beam-Search

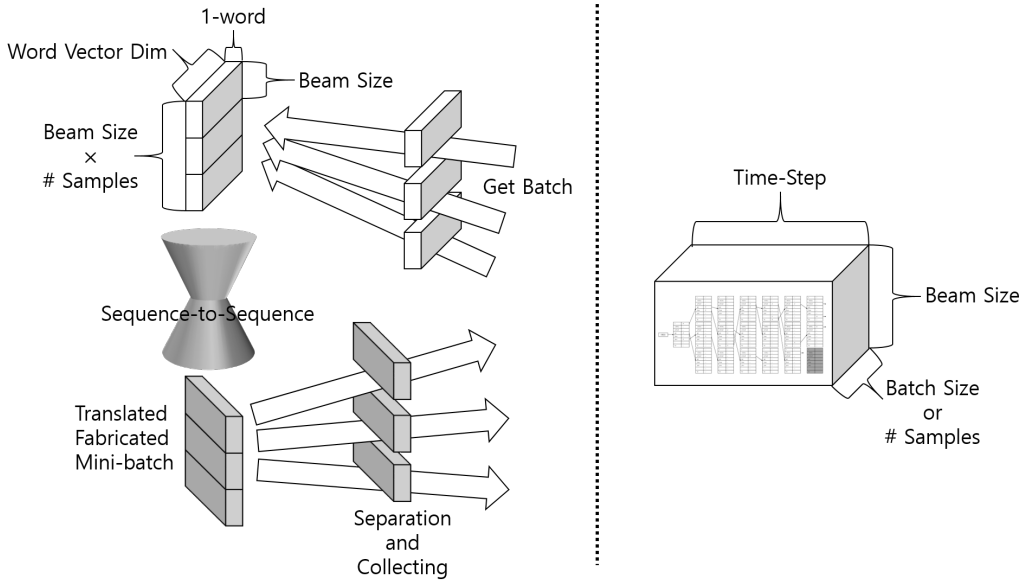

Figure 11:

```
def batch_beam_search(self, src, beam_size = 5, max_length = 255, n_best = 1)
    mask = None
    x_length = None
    if isinstance(src, tuple):
        x, x_length = src
        mask = self.generate_mask(x, x_length)
        # |mask| = (batch_size, length)
    else:
        x = src
    batch_size = x.size(0)

    emb_src = self.emb_src(x)
    h_src, h_0_tgt = self.encoder((emb_src, x_length))
    # |h_src| = (batch_size, length, hidden_size)
    h_0_tgt, c_0_tgt = h_0_tgt
    h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
    c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
    # |h_0_tgt| = (n_layers, batch_size, hidden_size)
    h_0_tgt = (h_0_tgt, c_0_tgt)

    # initialize 'SingleBeamSearchSpace' as many as batch_size
    spaces = [SingleBeamSearchSpace((h_0_tgt[0][:, i, :].unsqueeze(1),
                                     h_0_tgt[1][:, i, :].unsqueeze(1)),
                                     None,
                                     beam_size,
                                     max_length = max_length
                                     ) for i in range(batch_size)]
    done_cnt = [space.is_done() for space in spaces]

    length = 0
    # Run loop while sum of 'done_cnt' is smaller than batch_size, or length
    while sum(done_cnt) < batch_size and length <= max_length:
        # current_batch_size = sum(done_cnt) * beam_size

        # Initialize fabricated variables.
        # As far as batch-beam-search is running,
        # temporary batch-size for fabricated mini-batch is 'beam_size'-time
        fab_input, fab_hidden, fab_cell, fab_h_t_tilde = [], [], [], []
```

```
fab_h_src, fab_mask = [], []

# Build fabricated mini-batch in non-parallel way.
# This may cause a bottle-neck.
for i, space in enumerate(spaces):
    if space.is_done() == 0: # Batchfy only if the inference for the
        y_hat_, (hidden_, cell_), h_t_tilde_ = space.get_batch()

        fab_input += [y_hat_]
        fab_hidden += [hidden_]
        fab_cell += [cell_]
        if h_t_tilde_ is not None:
            fab_h_t_tilde += [h_t_tilde_]
        else:
            fab_h_t_tilde = None

        fab_h_src += [h_src[i, :, :]] * beam_size
        fab_mask += [mask[i, :]] * beam_size

# Now, concatenate list of tensors.
fab_input = torch.cat(fab_input, dim = 0)
fab_hidden = torch.cat(fab_hidden, dim = 1)
fab_cell = torch.cat(fab_cell, dim = 1)
if fab_h_t_tilde is not None:
    fab_h_t_tilde = torch.cat(fab_h_t_tilde, dim = 0)
fab_h_src = torch.stack(fab_h_src)
fab_mask = torch.stack(fab_mask)
# |fab_input| = (current_batch_size, 1)
# |fab_hidden| = (n_layers, current_batch_size, hidden_size)
# |fab_cell| = (n_layers, current_batch_size, hidden_size)
# |fab_h_t_tilde| = (current_batch_size, 1, hidden_size)
# |fab_h_src| = (current_batch_size, length, hidden_size)
# |fab_mask| = (current_batch_size, length)

emb_t = self.emb_dec(fab_input)
# |emb_t| = (current_batch_size, 1, word_vec_dim)

fab_decoder_output, (fab_hidden, fab_cell) = self.decoder(emb_t, fab_h
```

```
        # |fab_decoder_output| = (current_batch_size, 1, hidden_size)
        context_vector = self.attn(fab_h_src, fab_decoder_output, fab_mask)
        # |context_vector| = (current_batch_size, 1, hidden_size)
        fab_h_t_tilde = self.tanh(self.concat(torch.cat([fab_decoder_output, c
        # |fab_h_t_tilde| = (current_batch_size, 1, hidden_size)
        y_hat = self.generator(fab_h_t_tilde)
        # |y_hat| = (current_batch_size, 1, output_size)

        # separate the result for each sample.
        cnt = 0
        for space in spaces:
            if space.is_done() == 0:
                # Decide a range of each sample.
                from_index = cnt * beam_size
                to_index = from_index + beam_size

                # pick k-best results for each sample.
                space.collect_result(y_hat[from_index:to_index],
                                     (fab_hidden[:, from_index:to_index
                                         fab_cell[:, from_index:to_inde
                                     fab_h_t_tilde[from_index:to_index]
                                     )
                cnt += 1

        done_cnt = [space.is_done() for space in spaces]
        length += 1

    # pick n-best hypothesis.
    batch_sentences = []
    batch_probs = []

    # Collect the results.
    for i, space in enumerate(spaces):
        sentences, probs = space.get_n_best(n_best)

        batch_sentences += [sentences]
        batch_probs += [probs]
```

```
        return batch_sentences, batch_probs
```

Evaluation

번역기의 성능을 평가하는 방법은 크게 두 가지로 나눌 수 있습니다. 정성적(implicit) 평가와 정량적(explicit) 평가 방식입니다.

Implicit Evaluation

정성평가 방식은 보통 사람이 번역된 문장을 채점하는 형태로 이루어집니다. 사람은 선입견 등이 채점하는데 있어서 방해요소로 작용될 수 있기 때문에, 보통은 blind test를 통해서 채점합니다. 이를 위해서 여러개의 다른 알고리즘을 통해 (또는 경쟁사의) 여러 번역결과를 누구의 것인지 밝히지 않은 채, 채점하여 우열을 가립니다. 이 방식은 가장 정확하다고 할 수 있지만, 자원과 시간이 많이 드는 단점이 있습니다.

Explicit Evalution

위의 단점 때문에, 보통은 자동화 된 정량평가를 주로 수행합니다. 두 평가를 모두 주기적으로 수행하되, 정성평가의 평가 주기를 좀 더 길게 가져가거나, 무언가 확실한 성능의 jump가 이루어졌을 때 수행하는 편 입니다.

Cross Entropy and Perplexity

신경망 기계번역도 기본적으로 매 time-step마다 최고 확률을 갖는 단어를 선택(분류) 하는 작업이기 때문에 기본적으로 분류(classification)작업에 속합니다. 따라서 Cross Entropy를 손실함수(loss function)로 사용합니다. 이전 섹션에서 언급하였듯이, 신경망 기계번역도 조건부 언어모델이기 때문에 perplexity를 통해 성능을 측정할 수 있습니다. 그러므로 이전 언어모델 챕터에서 다루었듯이, cross entropy loss 값에 exponantial을 취하여 perplexity(PPL) 값을 얻을 수 있습니다.

BLEU

위의 PPL은 우리가 사용하는 Loss function과 직결되어 바로 알 수 있는 간편함이 있지만, 사실 안타깝게도 실제 번역기의 성능과 완벽한 비례관계에 있다고 할 수는 없습니다. Cross Entropy의 수식을 해석 해 보면, 각 time-step 별 실제 정답에 해당하는 단어의 확률만 채점하기 때문입니다.

원문	I	love	to	go	to	school	.
index	0	1	2	3	4	5	6
정답	나는	학교에	가는	것을	좋아한다	.	
번역1	학교에	가는	것을	좋아한다	나는	.	
번역2	나는	오락실에	가는	것을	싫어한다	.	

예를 들어, 번역1은 cross entropy loss에 의하면 매우 높은 loss값을 가집니다. 하지만 번역2는 번역1에 비해 완전 틀린 번역이지만 loss가 훨씬 낮을 겁니다. 따라서 실제 번역문의 품질과 cross entropy 사이에는 (특히 teacher forcing 방식이 더해져) 괴리가 있습니다. 이러한 간극을 줄이기 위해 여러가지 방법들이 제시되었습니다 - METEOR, BLEU. 이번 섹션은 그 중 가장 널리 쓰이는 BLEU에 대해 짚고 넘어가겠습니다.

$$BLEU = brevity\text{-}penalty * \prod_{n=1}^{N} p_n^{w_n}$$

$$where\ brevity\text{-}penalty = \min(1, \frac{|prediction|}{|reference|})$$

$$and\ p_n\ is\ precision\ of\ n\text{-}gram\ and\ w_n\ is\ weight\ that\ w_n = \frac{1}{2^n}$$

BLEU는 정답 문장과 예측 문장 사이에 일치하는 n-gram의 갯수의 비율의 기하평균에 따라 점수가 매겨집니다. brevity penalty는 예측 된 번역문이 정답 문장보다 짧을 경우 점수가 좋아지는 것을 방지하기 위함입니다. 보통 위 수식의 결과 값에 100을 곱하여 0-100의 scale로 점수를 표현합니다. 실제 위의 예제 '번역1'에서 나타난 2-gram을 count하여 간단하게 BLEU를 측정 하여 보겠습니다.

2-gram	count	hit count
BOS 학교에	1	0

2-gram	count	hit count
학교에 가는	1	1
가는 것을	1	1
것을 좋아한다	1	1
좋아한다 나는	1	0
나는 .	1	0
. EOS	1	1
합계	7	4

따라서 2-gram의 BLEU score는 4/7이 됩니다. 이번에는 '번역2'에 대한 2-gram BLEU를 측정 해 보겠습니다.

2-gram	count	hit count
BOS 나는	1	1
나는 오락실에	1	0
오락실에 가는	1	0
가는 것을	1	1
것을 싫어한다	1	0
싫어한다 .	1	0
. EOS	1	1
합계	7	3

'번역2'의 2-gram BLEU는 3/7가 나왔습니다. 그러므로 (brevity penalty나 1-gram, 2-gram, 3-gram, 4-gram의 점수를 평균내지는 않았지만) 2-gram에 한해서 $4/7 > 3/7$이므로 '번역1'이 더 잘 번역되었다고 볼 수 있습니다. 즉, 위의 예제에서 '번역1'이 '번역2'보다 일치하는 n-gram이 더 많으므로 더 높은 BLEU 점수를 획득할 수 있습니다. 이와 같이 BLEU는 대체로 실제 정성평가의 결과와 일치하는 경향이 있다고 여겨집니다.

결론적으로 우리는 성능 평가 결과를 해석할 때 Perplexity(=Loss)는 낮을수록 좋고, BLEU는 높을수록 좋다고 합니다. 앞으로 설명할 알고리즘들의 성능을 평가할 때 참고 바랍니다. 실제 성능을 측정하기 위해서는 보통 SMT 프레임웍인 MOSES의 multi-bleu.perl을 주로 사용합니다.

Future Work

[Koehn at el.2017]에 따르면 신경망기계번역(NMT)에는 아직 다음과 같은 도전 과제들이 남아있습니다.

Full Source Code for Neural Machine Translation via RNN Sequence-to-Sequence

github repo url: https://github.com/kh-kim/simple-nmt

train.py

```
import argparse, sys

import torch
import torch.nn as nn

from data_loader import DataLoader
import data_loader
from simple_nmt.seq2seq import Seq2Seq
import simple_nmt.trainer as trainer


def define_argparser():
    p = argparse.ArgumentParser()

    p.add_argument('-model', required = True, help = 'Model file name to save. Ado
    p.add_argument('-train', required = True, help = 'Training set file name excep
    p.add_argument('-valid', required = True, help = 'Validation set file name exc
    p.add_argument('-lang', required = True, help = 'Set of extention represents l
    p.add_argument('-gpu_id', type = int, default = -1, help = 'GPU ID to train. C

    p.add_argument('-batch_size', type = int, default = 32, help = 'Mini batch siz
    p.add_argument('-n_epochs', type = int, default = 18, help = 'Number of epochs
```

```
    p.add_argument('-print_every', type = int, default = 1000, help = 'Number of g
    p.add_argument('-early_stop', type = int, default = -1, help = 'The training w

    p.add_argument('-max_length', type = int, default = 80, help = 'Maximum length
    p.add_argument('-dropout', type = float, default = .2, help = 'Dropout rate. D
    p.add_argument('-word_vec_dim', type = int, default = 512, help = 'Word embedd
    p.add_argument('-hidden_size', type = int, default = 768, help = 'Hidden size
    p.add_argument('-n_layers', type = int, default = 4, help = 'Number of layers

    p.add_argument('-max_grad_norm', type = float, default = 5., help = 'Threshold
    p.add_argument('-adam', action = 'store_true', help = 'Use Adam instead of usi
    p.add_argument('-lr', type = float, default = 1., help = 'Initial learning rat
    p.add_argument('-min_lr', type = float, default = .000001, help = 'Minimum lea
    p.add_argument('-lr_decay_start_at', type = int, default = 10, help = 'Start l
    p.add_argument('-lr_slow_decay', action = 'store_true', help = 'Decay learning
    p.add_argument('-lr_decay_rate', type = float, default = .5, help = 'Learning

    config = p.parse_args()

    return config

def overwrite_config(config, prev_config):
    # This method provides a compatibility for new or missing arguments.
    for key in vars(prev_config).keys():
            if '-%s' % key not in sys.argv or key == 'model':
                if vars(config).get(key) is not None:
                    vars(config)[key] = vars(prev_config)[key]
                else:
                    # Missing argument
                    print('WARNING!!! Argument "-%s" is not found in current argum
            else:
                # Argument value is change from saved model.
                print('WARNING!!! Argument "-%s" is not loaded from saved model.\t

    return config


if __name__ == "__main__":
```

```
config = define_argparser()

import os.path
# If the model exists, load model and configuration to continue the training
if os.path.isfile(config.model):
    saved_data = torch.load(config.model)

    prev_config = saved_data['config']
    config = overwrite_config(config, prev_config)
    config.lr = saved_data['current_lr']
else:
    saved_data = None

# Load training and validation data set.
loader = DataLoader(config.train,
                    config.valid,
                    (config.lang[:2], config.lang[-2:]),
                    batch_size = config.batch_size,
                    device = config.gpu_id,
                    max_length = config.max_length
                    )

input_size = len(loader.src.vocab) # Encoder's embedding layer input size
output_size = len(loader.tgt.vocab) # Decoder's embedding layer input size ar
# Declare the model
model = Seq2Seq(input_size,
                config.word_vec_dim, # Word embedding vector size
                config.hidden_size, # LSTM's hidden vector size
                output_size,
                n_layers = config.n_layers, # number of layers in LSTM
                dropout_p = config.dropout # dropout-rate in LSTM
                )

# Default weight for loss equals to 1, but we don't need to get loss for PAD
# Thus, set a weight for PAD to zero.
loss_weight = torch.ones(output_size)
loss_weight[data_loader.PAD] = 0.
# Instead of using Cross-Entropy loss, we can use Negative Log-Likelihood(NL
```

```
    criterion = nn.NLLLoss(weight = loss_weight, size_average = False)

    print(model)
    print(criterion)

    # Pass models to GPU device if it is necessary.
    if config.gpu_id >= 0:
        model.cuda(config.gpu_id)
        criterion.cuda(config.gpu_id)

    # If we have loaded model weight parameters, use that weights for declared m
    if saved_data is not None:
        model.load_state_dict(saved_data['model'])

    # Start training. This function maybe equivalant to 'fit' function in Keras.
    trainer.train_epoch(model,
                        criterion,
                        loader.train_iter,
                        loader.valid_iter,
                        config,
                        start_epoch = saved_data['epoch'] if saved_data is not No
                        others_to_save = {'src_vocab': loader.src.vocab, 'tgt_voca
                        )
```

translate.py

```
import argparse, sys
from operator import itemgetter

import torch
import torch.nn as nn

from data_loader import DataLoader
import data_loader
from simple_nmt.seq2seq import Seq2Seq
import simple_nmt.trainer as trainer
```

```
def define_argparser():
    p = argparse.ArgumentParser()

    p.add_argument('-model', required = True, help = 'Model file name to use')
    p.add_argument('-gpu_id', type = int, default = -1, help = 'GPU ID to use. -1

    p.add_argument('-batch_size', type = int, default = 128, help = 'Mini batch si
    p.add_argument('-max_length', type = int, default = 255, help = 'Maximum seque
    p.add_argument('-n_best', type = int, default = 1, help = 'Number of best infe
    p.add_argument('-beam_size', type = int, default = 5, help = 'Beam size for be

    config = p.parse_args()

    return config

def read_text():
    # This method gets sentences from standard input and tokenize those.
    lines = []

    for line in sys.stdin:
        if line.strip() != '':
            lines += [line.strip().split(' ')]

    return lines

def to_text(indice, vocab):
    # This method converts index to word to show the translation result.
    lines = []

    for i in range(len(indice)):
        line = []
        for j in range(len(indice[i])):
            index = indice[i][j]

            if index == data_loader.EOS:
                #line += ['<EOS>']
                break
            else:
```

```
                line += [vocab.itos[index]]

            line = ' '.join(line)
            lines += [line]

        return lines

if __name__ == '__main__':
    config = define_argparser()

    # Load saved model.
    saved_data = torch.load(config.model)

    # Load configuration setting in training.
    train_config = saved_data['config']
    # Load vocabularies from the model.
    src_vocab = saved_data['src_vocab']
    tgt_vocab = saved_data['tgt_vocab']

    # Initialize dataloader, but we don't need to read training & test corpus.
    # What we need is just load vocabularies from the previously trained model.
    loader = DataLoader()
    loader.load_vocab(src_vocab, tgt_vocab)
    input_size = len(loader.src.vocab)
    output_size = len(loader.tgt.vocab)

    # Declare sequence-to-sequence model.
    model = Seq2Seq(input_size,
                    train_config.word_vec_dim,
                    train_config.hidden_size,
                    output_size,
                    n_layers = train_config.n_layers,
                    dropout_p = train_config.dropout
                    )
    model.load_state_dict(saved_data['model']) # Load weight parameters from the
    model.eval() # We need to turn-on the evaluation mode, which turns off all d

    # We don't need to draw a computation graph, because we will have only infer
```

```
torch.set_grad_enabled(False)

# Put models to device if it is necessary.
if config.gpu_id >= 0:
    model.cuda(config.gpu_id)

# Get sentences from standard input.
lines = read_text()

with torch.no_grad(): # Also, declare again to prevent to get gradients.
    while len(lines) > 0:
        # Since packed_sequence must be sorted by decreasing order of length
        # sorting by length in mini-batch should be restored by original ord
        # Therefore, we need to memorize the original index of the sentence.
        sorted_lines = lines[:config.batch_size]
        lines = lines[config.batch_size:]
        lengths = [len(_) for _ in sorted_lines]
        orders = [i for i in range(len(sorted_lines))]

        sorted_tuples = sorted(zip(sorted_lines, lengths, orders), key = itemg
        sorted_lines = [sorted_tuples[i][0] for i in range(len(sorted_tuples))
        lengths = [sorted_tuples[i][1] for i in range(len(sorted_tuples))]
        orders = [sorted_tuples[i][2] for i in range(len(sorted_tuples))]

        # Converts string to list of index.
        x = loader.src.numericalize(loader.src.pad(sorted_lines), device = 'cu

        if config.beam_size == 1:
            # Take inference for non-parallel beam-search.
            y_hat, indice = model.search(x)
            output = to_text(indice, loader.tgt.vocab)

            sorted_tuples = sorted(zip(output, orders), key = itemgetter(1))
            output = [sorted_tuples[i][0] for i in range(len(sorted_tuples))]

            sys.stdout.write('\n'.join(output) + '\n')
        else:
            # Take mini-batch parallelized beam search.
```

```
                batch_indice, _ = model.batch_beam_search(x,
                                                          beam_size = config.bea
                                                          max_length = config.ma
                                                          n_best = config.n_best
                                                          )

                # Restore the original orders.
                output = []
                for i in range(len(batch_indice)):
                    output += [to_text(batch_indice[i], loader.tgt.vocab)]
                sorted_tuples = sorted(zip(output, orders), key = itemgetter(1))
                output = [sorted_tuples[i][0] for i in range(len(sorted_tuples))]

                for i in range(len(output)):
                    sys.stdout.write('\n'.join(output[i]) + '\n')
```

data_loader.py

```
import os
from torchtext import data, datasets

PAD = 1
BOS = 2
EOS = 3

class DataLoader():

    def __init__(self, train_fn = None,
                       valid_fn = None,
                       exts = None,
                       batch_size = 64,
                       device = 'cpu',
                       max_vocab = 99999999,
                       max_length = 255,
                       fix_length = None,
                       use_bos = True,
                       use_eos = True,
```

```
            shuffle = True
            ):

        super(DataLoader, self).__init__()

        self.src = data.Field(sequential = True,
                              use_vocab = True,
                              batch_first = True,
                              include_lengths = True,
                              fix_length = fix_length,
                              init_token = None,
                              eos_token = None
                              )

        self.tgt = data.Field(sequential = True,
                              use_vocab = True,
                              batch_first = True,
                              include_lengths = True,
                              fix_length = fix_length,
                              init_token = '<BOS>' if use_bos else None,
                              eos_token = '<EOS>' if use_eos else None
                              )

        if train_fn is not None and valid_fn is not None and exts is not None:
            train = TranslationDataset(path = train_fn, exts = exts,
                                       fields = [('src', self.src), ('tgt', s
                                       max_length = max_length
                                       )
            valid = TranslationDataset(path = valid_fn, exts = exts,
                                       fields = [('src', self.src), ('tgt', s
                                       max_length = max_length
                                       )

            self.train_iter = data.BucketIterator(train,
                                                  batch_size = batch_size,
                                                  device = 'cuda:%d' % device if
                                                  shuffle = shuffle,
                                                  sort_key=lambda x: len(x.tgt)
```

```
                                                    sort_within_batch = True
                                                    )
            self.valid_iter = data.BucketIterator(valid,
                                                    batch_size = batch_size,
                                                    device = 'cuda:%d' % device if
                                                    shuffle = False,
                                                    sort_key=lambda x: len(x.tgt)
                                                    sort_within_batch = True
                                                    )

            self.src.build_vocab(train, max_size = max_vocab)
            self.tgt.build_vocab(train, max_size = max_vocab)

    def load_vocab(self, src_vocab, tgt_vocab):
        self.src.vocab = src_vocab
        self.tgt.vocab = tgt_vocab

class TranslationDataset(data.Dataset):
    """Defines a dataset for machine translation."""

    @staticmethod
    def sort_key(ex):
        return data.interleave_keys(len(ex.src), len(ex.trg))

    def __init__(self, path, exts, fields, max_length=None, **kwargs):
        """Create a TranslationDataset given paths and fields.

        Arguments:
            path: Common prefix of paths to the data files for both languages.
            exts: A tuple containing the extension to path for each language.
            fields: A tuple containing the fields that will be used for data
                in each language.
            Remaining keyword arguments: Passed to the constructor of
                data.Dataset.
        """
        if not isinstance(fields[0], (tuple, list)):
            fields = [('src', fields[0]), ('trg', fields[1])]
```

```
        if not path.endswith('.'):
            path += '.'

        src_path, trg_path = tuple(os.path.expanduser(path + x) for x in exts)

        examples = []
        with open(src_path) as src_file, open(trg_path) as trg_file:
            for src_line, trg_line in zip(src_file, trg_file):
                src_line, trg_line = src_line.strip(), trg_line.strip()
                if max_length and max_length < max(len(src_line.split()), len(trg
                    continue
                if src_line != '' and trg_line != '':
                    examples.append(data.Example.fromlist(
                        [src_line, trg_line], fields))

        super(TranslationDataset, self).__init__(examples, fields, **kwargs)

if __name__ == '__main__':
    import sys
    loader = DataLoader(sys.argv[1], sys.argv[2], (sys.argv[3], sys.argv[4]), batc

    print(len(loader.src.vocab))
    print(len(loader.tgt.vocab))

    for batch_index, batch in enumerate(loader.train_iter):
        print(batch.src)
        print(batch.tgt)

        if batch_index > 1:
            break
```

simple_nmt/seq2seq.py

```
import numpy as np
import torch
import torch.nn as nn
from torch.nn.utils.rnn import pack_padded_sequence as pack
```

```
from torch.nn.utils.rnn import pad_packed_sequence as unpack

import data_loader
from simple_nmt.search import SingleBeamSearchSpace

class Attention(nn.Module):

    def __init__(self, hidden_size):
        super(Attention, self).__init__()

        self.linear = nn.Linear(hidden_size, hidden_size, bias = False)
        self.softmax = nn.Softmax(dim = -1)

    def forward(self, h_src, h_t_tgt, mask = None):
        # |h_src| = (batch_size, length, hidden_size)
        # |h_t_tgt| = (batch_size, 1, hidden_size)
        # |mask| = (batch_size, length)

        query = self.linear(h_t_tgt.squeeze(1)).unsqueeze(-1)
        # |query| = (batch_size, hidden_size, 1)

        weight = torch.bmm(h_src, query).squeeze(-1)
        # |weight| = (batch_size, length)
        if mask is not None:
            # Set each weight as -inf, if the mask value equals to 1.
            # Since the softmax operation makes -inf to 0, masked weights would
            # Thus, if the sample is shorter than other samples in mini-batch, t
            weight.masked_fill_(mask, -float('inf'))
        weight = self.softmax(weight)

        context_vector = torch.bmm(weight.unsqueeze(1), h_src)
        # |context_vector| = (batch_size, 1, hidden_size)

        return context_vector

class Encoder(nn.Module):

    def __init__(self, word_vec_dim, hidden_size, n_layers = 4, dropout_p = .2):
```

```
        super(Encoder, self).__init__()

        # Be aware of value of 'batch_first' parameter.
        # Also, its hidden_size is half of original hidden_size, because it is b
        self.rnn = nn.LSTM(word_vec_dim, int(hidden_size / 2), num_layers = n_laye

    def forward(self, emb):
        # |emb| = (batch_size, length, word_vec_dim)

        if isinstance(emb, tuple):
            x, lengths = emb
            x = pack(x, lengths.tolist(), batch_first = True)

            # Below is how pack_padded_sequence works.
            # As you can see, PackedSequence object has information about mini-b
            #
            # a = [torch.tensor([1,2,3]), torch.tensor([3,4])]
            # b = torch.nn.utils.rnn.pad_sequence(a, batch_first=True)
            # >>>>
            # tensor([[ 1,  2,  3],
            #     [ 3,  4,  0]])
            # torch.nn.utils.rnn.pack_padded_sequence(b, batch_first=True, lengt
            # >>>>PackedSequence(data=tensor([ 1,  3,  2,  4,  3]), batch_sizes=
        else:
            x = emb

        y, h = self.rnn(x)
        # |y| = (batch_size, length, hidden_size)
        # |h[0]| = (num_layers * 2, batch_size, hidden_size / 2)

        if isinstance(emb, tuple):
            y, _ = unpack(y, batch_first = True)

        return y, h

class Decoder(nn.Module):

    def __init__(self, word_vec_dim, hidden_size, n_layers = 4, dropout_p = .2):
```

```
        super(Decoder, self).__init__()

        # Be aware of value of 'batch_first' parameter and 'bidirectional' param
        self.rnn = nn.LSTM(word_vec_dim + hidden_size, hidden_size, num_layers = n

    def forward(self, emb_t, h_t_1_tilde, h_t_1):
        # |emb_t| = (batch_size, 1, word_vec_dim)
        # |h_t_1_tilde| = (batch_size, 1, hidden_size)
        # |h_t_1[0]| = (n_layers, batch_size, hidden_size)
        batch_size = emb_t.size(0)
        hidden_size = h_t_1[0].size(-1)

        if h_t_1_tilde is None:
            # If this is the first time-step,
            h_t_1_tilde = emb_t.new(batch_size, 1, hidden_size).zero_()

        # Input feeding trick.
        x = torch.cat([emb_t, h_t_1_tilde], dim = -1)

        # Unlike encoder, decoder must take an input for sequentially.
        y, h = self.rnn(x, h_t_1)

        return y, h

class Generator(nn.Module):

    def __init__(self, hidden_size, output_size):
        super(Generator, self).__init__()

        self.output = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim = -1)

    def forward(self, x):
        # |x| = (batch_size, length, hidden_size)

        y = self.softmax(self.output(x))
        # |y| = (batch_size, length, output_size)
```

```
        # Return log-probability instead of just probability.
        return y

class Seq2Seq(nn.Module):

    def __init__(self, input_size, word_vec_dim, hidden_size, output_size, n_layer
        self.input_size = input_size
        self.word_vec_dim = word_vec_dim
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.n_layers = n_layers
        self.dropout_p = dropout_p

        super(Seq2Seq, self).__init__()

        self.emb_src = nn.Embedding(input_size, word_vec_dim)
        self.emb_dec = nn.Embedding(output_size, word_vec_dim)

        self.encoder = Encoder(word_vec_dim, hidden_size, n_layers = n_layers, dro
        self.decoder = Decoder(word_vec_dim, hidden_size, n_layers = n_layers, dro
        self.attn = Attention(hidden_size)

        self.concat = nn.Linear(hidden_size * 2, hidden_size)
        self.tanh = nn.Tanh()
        self.generator = Generator(hidden_size, output_size)

    def generate_mask(self, x, length):
        mask = []

        max_length = max(length)
        for l in length:
            if max_length - l > 0:
                # If the length is shorter than maximum length among samples,
                # set last few values to be 1s to remove attention weight.
                mask += [torch.cat([x.new_ones(1, l).zero_(), x.new_ones(1, (max_l
            else:
                # If the length of the sample equals to maximum length among sam
                # set every value in mask to be 0.
```

```
                mask += [x.new_ones(1, l).zero_()]

        mask = torch.cat(mask, dim = 0).byte()

        return mask

    def merge_encoder_hiddens(self, encoder_hiddens):
        new_hiddens = []
        new_cells = []

        hiddens, cells = encoder_hiddens

        # i-th and (i+1)-th layer is opposite direction.
        # Also, each direction of layer is half hidden size.
        # Therefore, we concatenate both directions to 1 hidden size layer.
        for i in range(0, hiddens.size(0), 2):
            new_hiddens += [torch.cat([hiddens[i], hiddens[i + 1]], dim = -1)]
            new_cells += [torch.cat([cells[i], cells[i + 1]], dim = -1)]

        new_hiddens, new_cells = torch.stack(new_hiddens), torch.stack(new_cells)

        return (new_hiddens, new_cells)

    def forward(self, src, tgt):
        batch_size = tgt.size(0)

        mask = None
        x_length = None
        if isinstance(src, tuple):
            x, x_length = src
            # Based on the length information, gererate mask to prevent that sho
            mask = self.generate_mask(x, x_length)
            # |mask| = (batch_size, length)
        else:
            x = src

        if isinstance(tgt, tuple):
            tgt = tgt[0]
```

```
        # Get word embedding vectors for every time-step of input sentence.
        emb_src = self.emb_src(x)
        # |emb_src| = (batch_size, length, word_vec_dim)

        # The last hidden state of the encoder would be a initial hidden state o
        h_src, h_0_tgt = self.encoder((emb_src, x_length))
        # |h_src| = (batch_size, length, hidden_size)
        # |h_0_tgt| = (n_layers * 2, batch_size, hidden_size / 2)

        # Merge bidirectional to uni-directional
        # We need to convert size from (n_layers * 2, batch_size, hidden_size /
        # Thus, the converting operation will not working with just 'view' metho
        h_0_tgt, c_0_tgt = h_0_tgt
        h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
        c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
        # You can use 'merge_encoder_hiddens' method, instead of using above 3 l
        # 'merge_encoder_hiddens' method works with non-parallel way.
        # h_0_tgt = self.merge_encoder_hiddens(h_0_tgt)

        # |h_src| = (batch_size, length, hidden_size)
        # |h_0_tgt| = (n_layers, batch_size, hidden_size)
        h_0_tgt = (h_0_tgt, c_0_tgt)

        emb_tgt = self.emb_dec(tgt)
        # |emb_tgt| = (batch_size, length, word_vec_dim)
        h_tilde = []

        h_t_tilde = None
        decoder_hidden = h_0_tgt
        # Run decoder until the end of the time-step.
        for t in range(tgt.size(1)):
            # Teacher Forcing: take each input from training set, not from the l
            # Because of Teacher Forcing, training procedure and inference proce
            # Of course, because of sequential running in decoder, this causes s
            emb_t = emb_tgt[:, t, :].unsqueeze(1)
            # |emb_t| = (batch_size, 1, word_vec_dim)
            # |h_t_tilde| = (batch_size, 1, hidden_size)
```

```
            decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decode
            # |decoder_output| = (batch_size, 1, hidden_size)
            # |decoder_hidden| = (n_layers, batch_size, hidden_size)

            context_vector = self.attn(h_src, decoder_output, mask)
            # |context_vector| = (batch_size, 1, hidden_size)

            h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_v
            # |h_t_tilde| = (batch_size, 1, hidden_size)

            h_tilde += [h_t_tilde]

        h_tilde = torch.cat(h_tilde, dim = 1)
        # |h_tilde| = (batch_size, length, hidden_size)

        y_hat = self.generator(h_tilde)
        # |y_hat| = (batch_size, length, output_size)

        return y_hat

    def search(self, src, is_greedy = True, max_length = 255):
        mask = None
        x_length = None
        if isinstance(src, tuple):
            x, x_length = src
            mask = self.generate_mask(x, x_length)
        else:
            x = src
        batch_size = x.size(0)

        emb_src = self.emb_src(x)
        h_src, h_0_tgt = self.encoder((emb_src, x_length))
        h_0_tgt, c_0_tgt = h_0_tgt
        h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
        c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
        h_0_tgt = (h_0_tgt, c_0_tgt)
```

```
        # Fill a vector, which has 'batch_size' dimension, with BOS value.
        y = x.new(batch_size, 1).zero_() + data_loader.BOS
        is_undone = x.new_ones(batch_size, 1).float()
        decoder_hidden = h_0_tgt
        h_t_tilde, y_hats, indice = None, [], []

        # Repeat a loop while sum of 'is_undone' flag is bigger than 0, or curre
        while is_undone.sum() > 0 and len(indice) < max_length:
            # Unlike training procedure, take the last time-step's output during
            emb_t = self.emb_dec(y)
            # |emb_t| = (batch_size, 1, word_vec_dim)

            decoder_output, decoder_hidden = self.decoder(emb_t, h_t_tilde, decode
            context_vector = self.attn(h_src, decoder_output, mask)
            h_t_tilde = self.tanh(self.concat(torch.cat([decoder_output, context_v
            y_hat = self.generator(h_t_tilde)
            # |y_hat| = (batch_size, 1, output_size)
            y_hats += [y_hat]

            if is_greedy:
                y = torch.topk(y_hat, 1, dim = -1)[1].squeeze(-1)
            else:
                # Take a random sampling based on the multinoulli distribution.
                y = torch.multinomial(y_hat.exp().view(batch_size, -1), 1)
            # Put PAD if the sample is done.
            y = y.masked_fill_((1. - is_undone).byte(), data_loader.PAD)
            is_undone = is_undone * torch.ne(y, data_loader.EOS).float()
            # |y| = (batch_size, 1)
            # |is_undone| = (batch_size, 1)
            indice += [y]

        y_hats = torch.cat(y_hats, dim = 1)
        indice = torch.cat(indice, dim = -1)
        # |y_hat| = (batch_size, length, output_size)
        # |indice| = (batch_size, length)

        return y_hats, indice
```

```
def batch_beam_search(self, src, beam_size = 5, max_length = 255, n_best = 1)
    mask = None
    x_length = None
    if isinstance(src, tuple):
        x, x_length = src
        mask = self.generate_mask(x, x_length)
        # |mask| = (batch_size, length)
    else:
        x = src
    batch_size = x.size(0)

    emb_src = self.emb_src(x)
    h_src, h_0_tgt = self.encoder((emb_src, x_length))
    # |h_src| = (batch_size, length, hidden_size)
    h_0_tgt, c_0_tgt = h_0_tgt
    h_0_tgt = h_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
    c_0_tgt = c_0_tgt.transpose(0, 1).contiguous().view(batch_size, -1, self.h
    # |h_0_tgt| = (n_layers, batch_size, hidden_size)
    h_0_tgt = (h_0_tgt, c_0_tgt)

    # initialize 'SingleBeamSearchSpace' as many as batch_size
    spaces = [SingleBeamSearchSpace((h_0_tgt[0][:, i, :].unsqueeze(1),
                                     h_0_tgt[1][:, i, :].unsqueeze(1)),
                                     None,
                                     beam_size,
                                     max_length = max_length
                                     ) for i in range(batch_size)]
    done_cnt = [space.is_done() for space in spaces]

    length = 0
    # Run loop while sum of 'done_cnt' is smaller than batch_size, or length
    while sum(done_cnt) < batch_size and length <= max_length:
        # current_batch_size = sum(done_cnt) * beam_size

        # Initialize fabricated variables.
        # As far as batch-beam-search is running,
        # temporary batch-size for fabricated mini-batch is 'beam_size'-time
        fab_input, fab_hidden, fab_cell, fab_h_t_tilde = [], [], [], []
```

```
fab_h_src, fab_mask = [], []

# Build fabricated mini-batch in non-parallel way.
# This may cause a bottle-neck.
for i, space in enumerate(spaces):
    if space.is_done() == 0: # Batchfy only if the inference for the
        y_hat_, (hidden_, cell_), h_t_tilde_ = space.get_batch()

        fab_input += [y_hat_]
        fab_hidden += [hidden_]
        fab_cell += [cell_]
        if h_t_tilde_ is not None:
            fab_h_t_tilde += [h_t_tilde_]
        else:
            fab_h_t_tilde = None

        fab_h_src += [h_src[i, :, :]] * beam_size
        fab_mask += [mask[i, :]] * beam_size

# Now, concatenate list of tensors.
fab_input = torch.cat(fab_input, dim = 0)
fab_hidden = torch.cat(fab_hidden, dim = 1)
fab_cell = torch.cat(fab_cell, dim = 1)
if fab_h_t_tilde is not None:
    fab_h_t_tilde = torch.cat(fab_h_t_tilde, dim = 0)
fab_h_src = torch.stack(fab_h_src)
fab_mask = torch.stack(fab_mask)
# |fab_input| = (current_batch_size, 1)
# |fab_hidden| = (n_layers, current_batch_size, hidden_size)
# |fab_cell| = (n_layers, current_batch_size, hidden_size)
# |fab_h_t_tilde| = (current_batch_size, 1, hidden_size)
# |fab_h_src| = (current_batch_size, length, hidden_size)
# |fab_mask| = (current_batch_size, length)

emb_t = self.emb_dec(fab_input)
# |emb_t| = (current_batch_size, 1, word_vec_dim)

fab_decoder_output, (fab_hidden, fab_cell) = self.decoder(emb_t, fab_h
```

```
        # |fab_decoder_output| = (current_batch_size, 1, hidden_size)
        context_vector = self.attn(fab_h_src, fab_decoder_output, fab_mask)
        # |context_vector| = (current_batch_size, 1, hidden_size)
        fab_h_t_tilde = self.tanh(self.concat(torch.cat([fab_decoder_output, c
        # |fab_h_t_tilde| = (current_batch_size, 1, hidden_size)
        y_hat = self.generator(fab_h_t_tilde)
        # |y_hat| = (current_batch_size, 1, output_size)

        # separate the result for each sample.
        cnt = 0
        for space in spaces:
            if space.is_done() == 0:
                # Decide a range of each sample.
                from_index = cnt * beam_size
                to_index = from_index + beam_size

                # pick k-best results for each sample.
                space.collect_result(y_hat[from_index:to_index],
                                     (fab_hidden[:, from_index:to_index
                                         fab_cell[:, from_index:to_inde
                                     fab_h_t_tilde[from_index:to_index]
                                     )
                cnt += 1

        done_cnt = [space.is_done() for space in spaces]
        length += 1

    # pick n-best hypothesis.
    batch_sentences = []
    batch_probs = []

    # Collect the results.
    for i, space in enumerate(spaces):
        sentences, probs = space.get_n_best(n_best)

        batch_sentences += [sentences]
        batch_probs += [probs]
```

```
        return batch_sentences, batch_probs
```

simple_nmt/trainer.py

```
import time
import numpy as np

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.utils as torch_utils

import utils

def get_loss(y, y_hat, criterion, do_backward = True):
    # |y| = (batch_size, length)
    # |y_hat| = (batch_size, length, output_size)
    batch_size = y.size(0)

    loss = criterion(y_hat.contiguous().view(-1, y_hat.size(-1)), y.contiguous().v
    if do_backward:
        loss.div(batch_size).backward()

    return loss

def train_epoch(model, criterion, train_iter, valid_iter, config, start_epoch = 1
    current_lr = config.lr

    lowest_valid_loss = np.inf
    no_improve_cnt = 0

    for epoch in range(start_epoch, config.n_epochs + 1):
        if config.adam:
            optimizer = optim.Adam(model.parameters(), lr = current_lr)
        else:
            optimizer = optim.SGD(model.parameters(), lr = current_lr)
        print("current learning rate: %f" % current_lr)
```

```
print(optimizer)

sample_cnt = 0
total_loss, total_word_count, total_parameter_norm, total_grad_norm = 0, 0
start_time = time.time()
train_loss = np.inf

for batch_index, batch in enumerate(train_iter):
    # You have to reset the gradients of all model parameters before to
    optimizer.zero_grad()

    current_batch_word_cnt = torch.sum(batch.tgt[1])
    x = batch.src
    # Raw target variable has both BOS and EOS token.
    # The output of sequence-to-sequence does not have BOS token.
    # Thus, remove BOS token for reference.
    y = batch.tgt[0][:, 1:]
    # |x| = (batch_size, length)
    # |y| = (batch_size, length)

    # Take feed-forward
    # Similar as before, the input of decoder does not have EOS token.
    # Thus, remove EOS token for decoder input.
    y_hat = model(x, batch.tgt[0][:, :-1])
    # |y_hat| = (batch_size, length, output_size)

    # Calcuate loss and gradients with back-propagation.
    loss = get_loss(y, y_hat, criterion)

    # Simple math to show stats.
    total_loss += float(loss)
    total_word_count += int(current_batch_word_cnt)
    total_parameter_norm += float(utils.get_parameter_norm(model.parameter
    total_grad_norm += float(utils.get_grad_norm(model.parameters()))

    # Print current training status in every this number of mini-batch i
    if (batch_index + 1) % config.print_every == 0:
        avg_loss = total_loss / total_word_count
```

```
            avg_parameter_norm = total_parameter_norm / config.print_every
            avg_grad_norm = total_grad_norm / config.print_every
            elapsed_time = time.time() - start_time

            # You can check the current status using parameter norm and grad
            # Also, you can check the speed of the training.
            print("epoch: %d batch: %d/%d\t|param|: %.2f\t|g_param|: %.2f\tlos

            total_loss, total_word_count, total_parameter_norm, total_grad_nor
            start_time = time.time()

            train_loss = avg_loss

        # In orther to avoid gradient exploding, we apply gradient clipping.
        torch_utils.clip_grad_norm_(model.parameters(), config.max_grad_norm)
        # Take a step of gradient descent.
        optimizer.step()

        sample_cnt += batch.tgt[0].size(0)
        if sample_cnt >= len(train_iter.dataset.examples):
            break

    sample_cnt = 0
    total_loss, total_word_count = 0, 0

    with torch.no_grad(): # In validation, we don't need to get gradients.
        model.eval() # Turn-on the evaluation mode.

        for batch_index, batch in enumerate(valid_iter):
```

```
            current_batch_word_cnt = torch.sum(batch.tgt[1])
            x = batch.src
            y = batch.tgt[0][:, 1:]
            # |x| = (batch_size, length)
            # |y| = (batch_size, length)

            # Take feed-forward
            y_hat = model(x, batch.tgt[0][:, :-1])
            # |y_hat| = (batch_size, length, output_size)

            loss = get_loss(y, y_hat, criterion, do_backward = False)

            total_loss += float(loss)
            total_word_count += int(current_batch_word_cnt)

            sample_cnt += batch.tgt[0].size(0)
            if sample_cnt >= len(valid_iter.dataset.examples):
                break

        # Print result of validation.
        avg_loss = total_loss / total_word_count
        print("valid loss: %.4f\tPPL: %.2f" % (avg_loss, np.exp(avg_loss)))

        if lowest_valid_loss > avg_loss:
            lowest_valid_loss = avg_loss
            no_improve_cnt = 0

            # Altough there is an improvement in last epoch, we need to deca
            if epoch >= config.lr_decay_start_at:
                current_lr = max(config.min_lr, current_lr * config.lr_decay_r
        else:
            # Decrease learing rate if there is no improvement.
            current_lr = max(config.min_lr, current_lr * config.lr_decay_rate)
            no_improve_cnt += 1

        # Again, turn-on the training mode.
        model.train()
```

```
        # Set a filename for model of last epoch.
        # We need to put every information to filename, as much as possible.
        model_fn = config.model.split(".")
        model_fn = model_fn[:-1] + ["%02d" % epoch, "%.2f-%.2f" % (train_loss, np

        # PyTorch provides efficient method for save and load model, which uses
        to_save = {"model": model.state_dict(),
                   "config": config,
                   "epoch": epoch + 1,
                   "current_lr": current_lr
                   }
        if others_to_save is not None: # Add others if it is necessary.
            for k, v in others_to_save.items():
                to_save[k] = v
        torch.save(to_save, '.'.join(model_fn))

        # Take early stopping if it meets the requirement.
        if config.early_stop > 0 and no_improve_cnt > config.early_stop:
            break
```

simple_nmt/search.py

```
from operator import itemgetter

import torch
import torch.nn as nn

import data_loader

LENGTH_PENALTY = 1.2
MIN_LENGTH = 5

class SingleBeamSearchSpace():

    def __init__(self, hidden, h_t_tilde = None, beam_size = 5, max_length = 255)
        self.beam_size = beam_size
        self.max_length = max_length
```

```
        super(SingleBeamSearchSpace, self).__init__()

        # To put data to same device.
        self.device = hidden[0].device
        # Inferred word index for each time-step. For now, initialized with init
        self.word_indice = [torch.LongTensor(beam_size).zero_().to(self.device) +
        # Index origin of current beam.
        self.prev_beam_indice = [torch.LongTensor(beam_size).zero_().to(self.devic
        # Cumulative log-probability for each beam.
        self.cumulative_probs = [torch.FloatTensor([.0] + [-float('inf')] * (beam_
        # 1 if it is done else 0
        self.masks = [torch.ByteTensor(beam_size).zero_().to(self.device)]

        # We don't need to remember every time-step of hidden states: prev_hidde
        # What we need is remember just last one.
        # Future work: make this class to deal with any necessary information fo

        # |hidden[0]| = (n_layers, 1, hidden_size)
        self.prev_hidden = torch.cat([hidden[0]] * beam_size, dim = 1)
        self.prev_cell = torch.cat([hidden[1]] * beam_size, dim = 1)
        # |prev_hidden| = (n_layers, beam_size, hidden_size)
        # |prev_cell| = (n_layers, beam_size, hidden_size)

        # |h_t_tilde| = (batch_size = 1, 1, hidden_size)
        self.prev_h_t_tilde = torch.cat([h_t_tilde] * beam_size, dim = 0) if h_t_t
        # |prev_h_t_tilde| = (beam_size, 1, hidden_size)

        self.current_time_step = 0
        self.done_cnt = 0

    def get_length_penalty(self, length, alpha = LENGTH_PENALTY, min_length = MIN_
        # Calculate length-penalty, because shorter sentence usually have bigger
        # Thus, we need to put penalty for shorter one.
        p = (1 + length) ** alpha / (1 + min_length) ** alpha

        return p
```

```
def is_done(self):
    # Return 1, if we had EOS more than 'beam_size'-times.
    if self.done_cnt >= self.beam_size:
        return 1
    return 0

def get_batch(self):
    y_hat = self.word_indice[-1].unsqueeze(-1)
    hidden = (self.prev_hidden, self.prev_cell)
    h_t_tilde = self.prev_h_t_tilde

    # |y_hat| = (beam_size, 1)
    # |hidden| = (n_layers, beam_size, hidden_size)
    # |h_t_tilde| = (beam_size, 1, hidden_size) or None
    return y_hat, hidden, h_t_tilde

def collect_result(self, y_hat, hidden, h_t_tilde):
    # |y_hat| = (beam_size, 1, output_size)
    # |hidden| = (n_layers, beam_size, hidden_size)
    # |h_t_tilde| = (beam_size, 1, hidden_size)
    output_size = y_hat.size(-1)

    self.current_time_step += 1

    # Calculate cumulative log-probability.
    # First, fill -inf value to last cumulative probability, if the beam is
    # Second, expand -inf filled cumulative probability to fit to 'y_hat'. (
    # Third, add expanded cumulative probability to 'y_hat'
    cumulative_prob = y_hat + self.cumulative_probs[-1].masked_fill_(self.mask
    # Now, we have new top log-probability and its index. We picked top inde
    # Be aware that we picked top-k from whole batch through 'view(-1)'.
    top_log_prob, top_indice = torch.topk(cumulative_prob.view(-1), self.beam_
    # |top_log_prob| = (beam_size)
    # |top_indice| = (beam_size)

    self.word_indice += [top_indice.fmod(output_size)] # Because we picked fr
    self.prev_beam_indice += [top_indice.div(output_size).long()] # Also, we
```

```
        # Add results to history boards.
        self.cumulative_probs += [top_log_prob]
        self.masks += [torch.eq(self.word_indice[-1], data_loader.EOS)] # Set fin
        self.done_cnt += self.masks[-1].float().sum() # Calculate a number of fin

        # Set hidden states for next time-step, using 'index_select' method.
        self.prev_hidden = torch.index_select(hidden[0], dim = 1, index = self.pre
        self.prev_cell = torch.index_select(hidden[1], dim = 1, index = self.prev_
        self.prev_h_t_tilde = torch.index_select(h_t_tilde, dim = 0, index = self.

    def get_n_best(self, n = 1):
        sentences = []
        probs = []
        founds = []

        for t in range(len(self.word_indice)): # for each time-step,
            for b in range(self.beam_size): # for each beam,
                if self.masks[t][b] == 1: # if we had EOS on this time-step and b
                    # Take a record of penaltified log-proability.
                    probs += [self.cumulative_probs[t][b] / self.get_length_penalt
                    founds += [(t, b)]

        # Also, collect log-probability from last time-step, for the case of EOS
        for b in range(self.beam_size):
            if self.cumulative_probs[-1][b] != -float('inf'):
                if not (len(self.cumulative_probs) - 1, b) in founds:
                    probs += [self.cumulative_probs[-1][b]]
                    founds += [(t, b)]

        # Sort and take n-best.
        sorted_founds_with_probs = sorted(zip(founds, probs),
                                          key = itemgetter(1),
                                          reverse = True
                                          )[:n]
        probs = []

        for (end_index, b), prob in sorted_founds_with_probs:
            sentence = []
```

```
            # Trace from the end.
            for t in range(end_index, 0, -1):
                sentence = [self.word_indice[t][b]] + sentence
                b = self.prev_beam_indice[t][b]

            sentences += [sentence]
            probs += [prob]

        return sentences, probs
```

utils.py

```
import torch

def get_grad_norm(parameters, norm_type = 2):
    parameters = list(filter(lambda p: p.grad is not None, parameters))

    total_norm = 0

    try:
        for p in parameters:
            param_norm = p.grad.data.norm(norm_type)
            total_norm += param_norm ** norm_type
        total_norm = total_norm ** (1. / norm_type)
    except Exception as e:
        print(e)

    return total_norm

def get_parameter_norm(parameters, norm_type = 2):
    total_norm = 0

    try:
        for p in parameters:
            param_norm = p.data.norm(norm_type)
```

```
            total_norm += param_norm ** norm_type
        total_norm = total_norm ** (1. / norm_type)
    except Exception as e:
        print(e)

    return total_norm
```