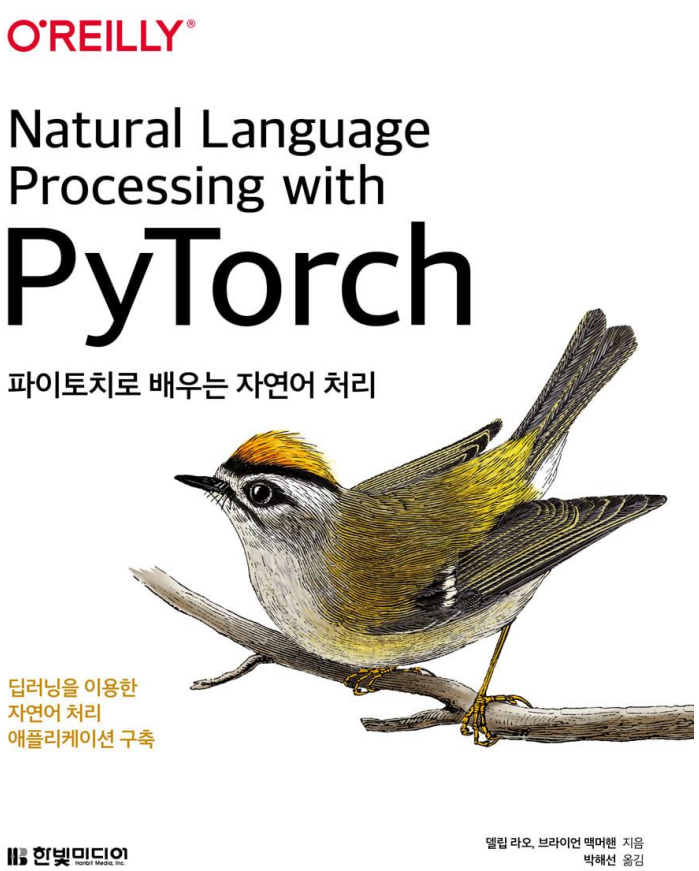
Data Intelligence Lab

2022.7.13 ~ 7.20

Progress

인천대학교 컴퓨터공학부 강병하

학습한 내용



6장 자연어 처리를 위한 시퀀스 모델링 - 초급

- 엘만 RNN
- 문자 RNN으로 성씨 국적 분류하기

7장 자연어 처리를 위한 시퀀스 모델링 - 중급

- 엘만 RNN의 문제점
- 엘만 RNN의 해결책 : 게이팅 (LSTM)
- 문자 RNN으로 성씨 생성하기

순차 데이터(sequence data)

- 언어에는 **순서 정보**가 있다

→ 순서가 변경되면 고유의 특성 잃어버림

Cogito, Ergo sum (나는 생각한다, 고로 존재한다.)

Sum, Ergo Cogito (나는 존재한다, 고로 생각한다.)

- 앞 뒤 단어의 제한을 받는다 (ex. 주어가 단수/복수)

The book is on the table.

The books are on the table.

➔ 앞에 어떤 단어가 나왔는지 기억하고 있어야 함

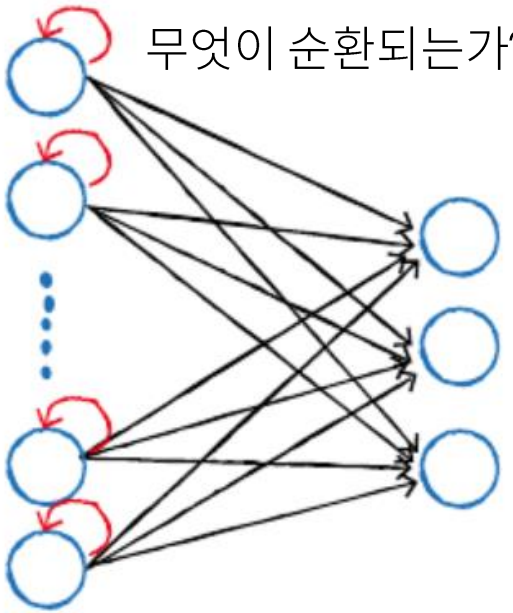
순환 신경망(RNN, Recurrent neural network)

시퀀스 모델링 in 딥러닝

시퀀스에 있는 각 항목을 만나면서 은닉 상태(hidden state) 업데이트

지금까지 시퀀스에서 본 모든 정보를 담은 벡터 ≒ 맥락 ≒ 요약

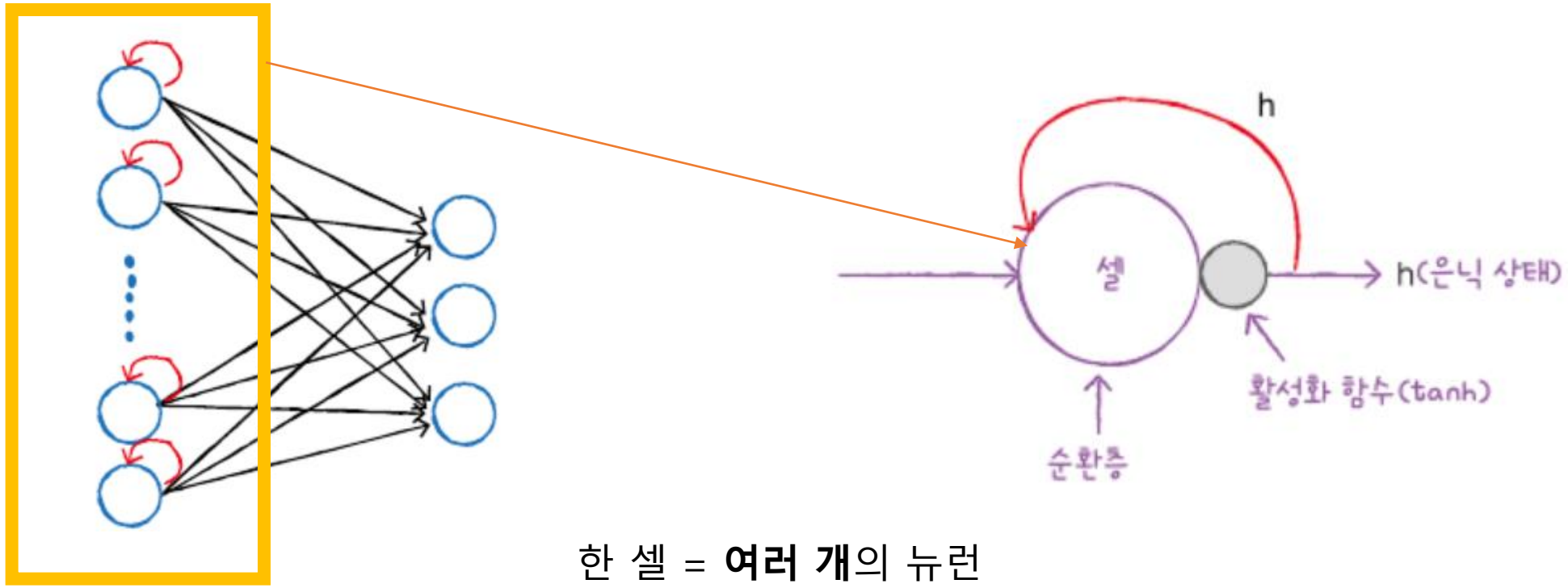
RNN = 신경망에 순환 고리 추가



무엇이 순환되는가? → 이전 은닉 상태(hidden state) 벡터가 입력으로 사용됨

"이전 샘플에 대한 기억을 가지고 있다."

| 순환 신경망(RNN, Recurrent neural network)

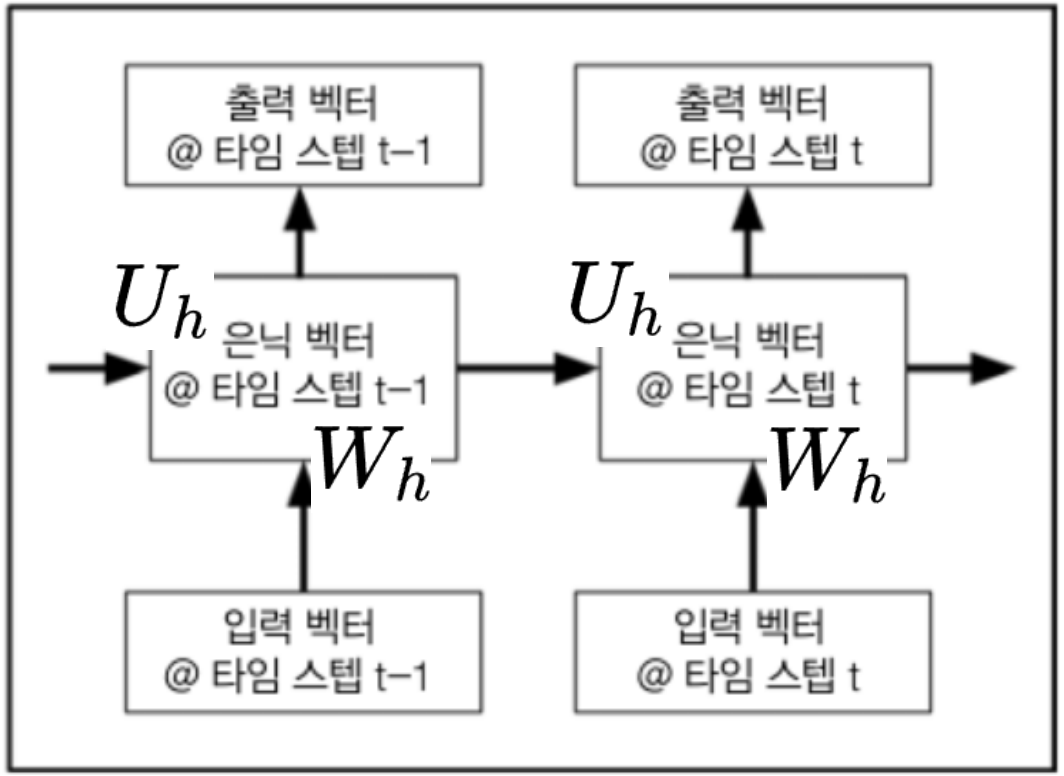


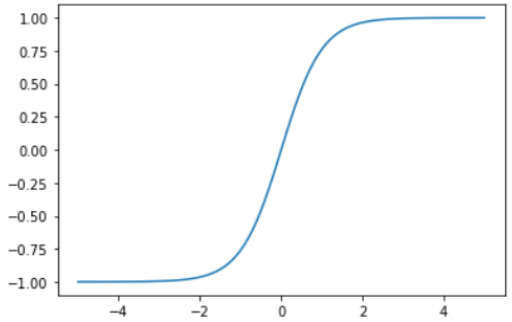
한 셀 = **여러 개**의 뉴런

순환 신경망 : 엘만 RNN

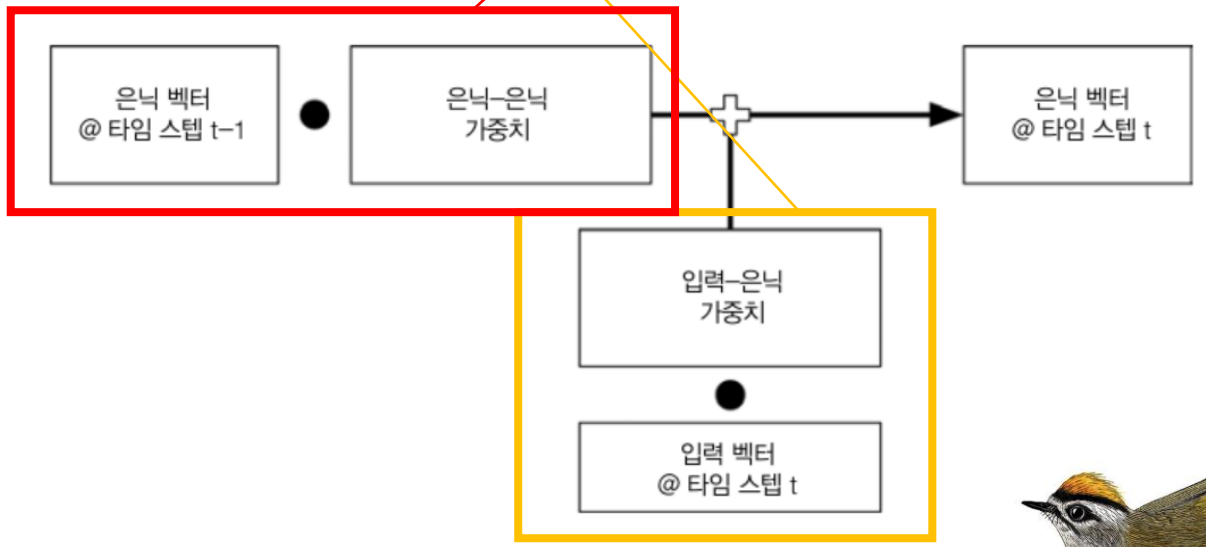
엘만 RNN

가장 기본적 형태의 RNN



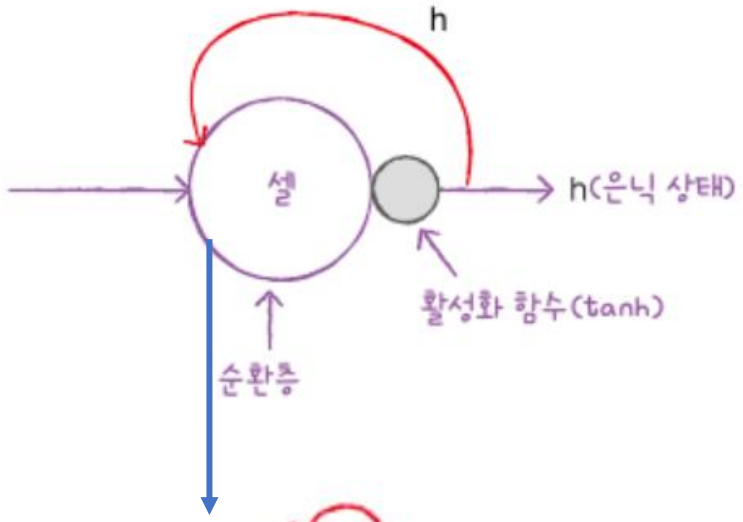


$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

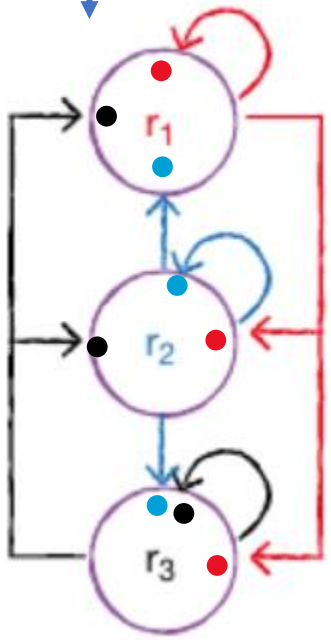


가중치는 매 타임스텝에 걸쳐 **공유됨** = **동일한** 가중치 사용

순환 신경망 : 은닉 상태 가중치



$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$



각 뉴런마다 나온 은닉 상태는
다음 타임스텝에 **모든 뉴런**에 모두 전달됨.

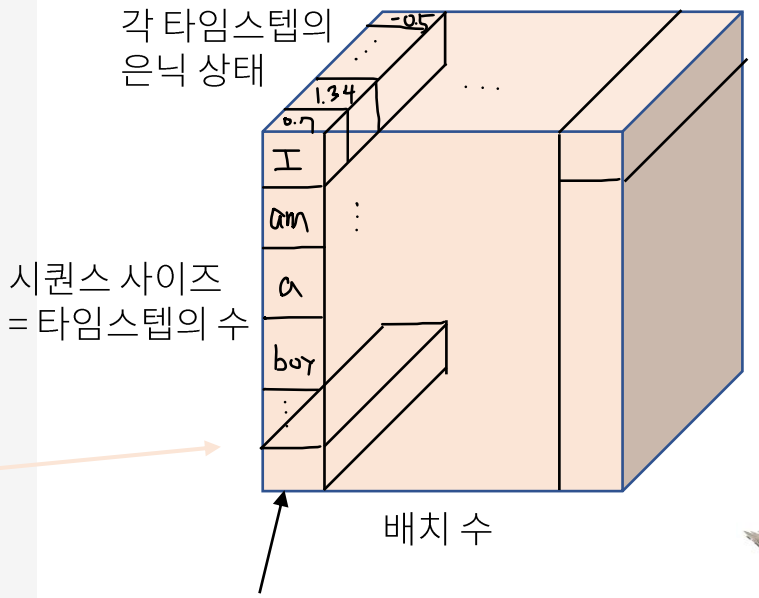
➔ **뉴런이 3개**라면 은닉 상태 가중치는

3(뉴런의 개수) x 3(각 뉴런의 은닉 상태 가중치 수) = **9개** 이다.

은닉 상태 가중치 9개와, 입력에 대한 가중치(Wh), 절편 파라미터를 찾는 것

엘만 RNN in PyTorch

```python
# 파이토치 RNNCell을 사용한 Elman RNN 구현
import torch.nn as nn

class ElmanRNN(nn.Module):
  def __init__(self, input_size, hidden_size, batch_first=False): # batch_first(bool) : 배치 차원이 0번째에 있는지
    # input_size(int) : 입력 벡터크기, hidden_size(int) : 은닉 상태 벡터 크기
    super(ElmanRNN, self).__init__()
    self.rnn_cell = nn.RNNCell(input_size, hidden_size)

    self.batch_first = batch_first
    self.hidden_size = hidden_size

  def _initialize_hidden(self, batch_size):
    return torch.zeros((batch_size, self.hidden_size))

  def forward(self, x_in, initial_hidden=None): # x_in : 입력 데이터 텐서
    if self.batch_first:
      batch_size, seq_size, feat_size = x_in.size() # batch_first가 True인 경우 입력 텐서의 0번째와 1번째 차원을 바꿈.
      x_in = x_in.permute(1,0,2)
    else:
      seq_size, batch_size, feat_size = x_in.size()

    hiddens = []

    if initial_hidden is None:
      initial_hidden = self._initialize_hidden(batch_size)
      initial_hidden = initial_hidden.to(x_in.device)

    hidden_t = initial_hidden

    for t in range(seq_size): # seq_size는 단어의 수 --> 타임스텝의 수
      hidden_t = self.rnn_cell(x_in[t], hidden_t) # 타임 스텝마다 은닉 상태 벡터 계산
      hiddens.append(hidden_t) # 각 타임스텝마다의 은닉 상태 벡터를 수집하여 쌓아 놓는다.

    hiddens = torch.stack(hiddens).stack() → 차원을 확장하여 tensor 쌓기

    if self.batch_first:
      hiddens = hiddens.permute(1, 0, 2)
    return hiddens # 출력도 3차원 텐서이다. (배치에 있는 각 데이터 포인트와 타임 스텝에 대한 은닉 상태 벡터)
```



문자 RNN으로 성씨 국적 분류하기 : 데이터셋

Surname Dataset (18개국 성씨 10000개)

	nationality	nationality_index	split	surname
0	Arabic	15	train	Totah
1	Arabic	15	train	Abboud
2	Arabic	15	train	Fakhoury
3	Arabic	15	train	Srour
4	Arabic	15	train	Sayegh

in MLP

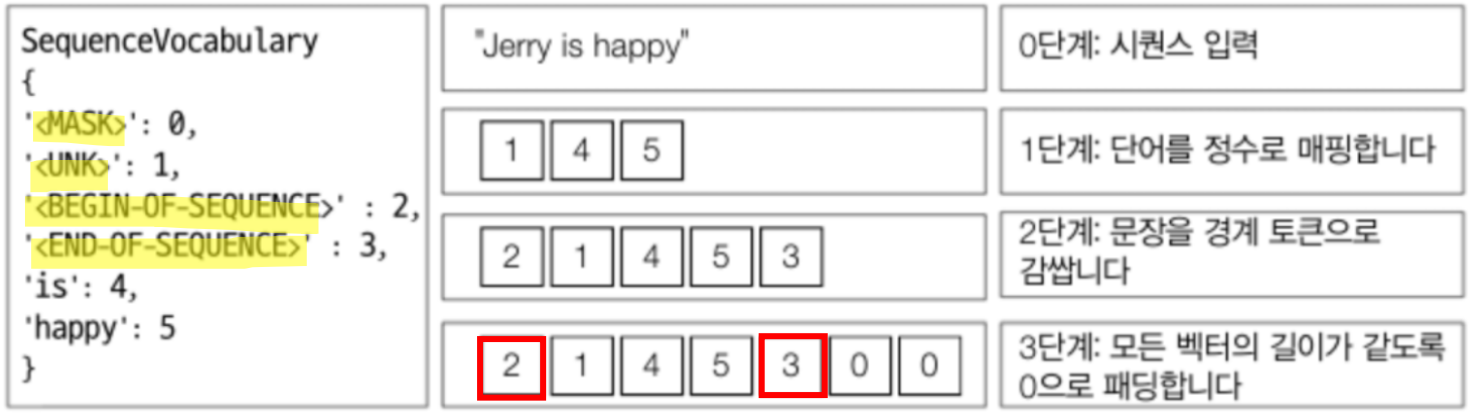
Totah = Haott

in RNN

Totah ≠ Haott

가변 길이 시퀀스 표현

단어와 문장은 길이가 다양 (입출력 크기 제각각) → 모든 시퀀스 길이를 인위적으로 동일하게



시작과 끝 경계를 표시

특수 토큰의 의미

<MASK> - 공백

<UNK> - 어휘사전에 없는 단어

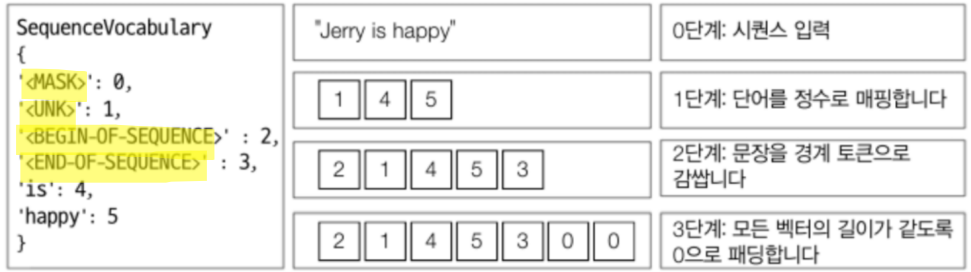
<BOS> - 문장의 시작

<EOS> - 문장 종료

문자 RNN으로 성씨 국적 분류하기 : 어휘사전

특수 토큰 추가

```
class SequenceVocabulary(Vocabulary):
    def __init__(self, token_to_idx=None, unk_token="<UNK>",
                 mask_token="<MASK>", begin_seq_token="<BEGIN>",
                 end_seq_token="<END>"):

        super(SequenceVocabulary, self).__init__(token_to_idx)

        self._mask_token = mask_token
        self._unk_token = unk_token
        self._begin_seq_token = begin_seq_token
        self._end_seq_token = end_seq_token

        self.mask_index = self.add_token(self._mask_token)
        self.unk_index = self.add_token(self._unk_token)
        self.begin_seq_index = self.add_token(self._begin_seq_token)
        self.end_seq_index = self.add_token(self._end_seq_token)

    def to_serializable(self):
        contents = super(SequenceVocabulary, self).to_serializable()
        contents.update({'unk_token': self._unk_token,
                         'mask_token': self._mask_token,
                         'begin_seq_token': self._begin_seq_token,
                         'end_seq_token': self._end_seq_token})
        return contents

    def lookup_token(self, token):
        """ 토큰에 대응하는 인덱스를 추출합니다.
        토큰이 없으면 UNK 인덱스를 반환합니다.

        매개변수:
            token (str): 찾을 토큰
        반환값:
            index (int): 토큰에 해당하는 인덱스
        노트:
            UNK 토큰을 사용하려면 (Vocabulary에 추가하기 위해)
            `unk_index`가 0보다 커야 합니다.
        """
        if self.unk_index >= 0:
            return self._token_to_idx.get(token, self.unk_index)
        else:
            return self._token_to_idx[token]
```




```
def add_token(self, token):
    """ 토큰을 기반으로 매핑 딕셔너리를 업데이트합니다

    매개변수:
        token (str): Vocabulary에 추가할 토큰
    반환값:
        index (int): 토큰에 상응하는 정수
    """
    if token in self._token_to_idx:
        index = self._token_to_idx[token]
    else:
        index = len(self._token_to_idx)
        self._token_to_idx[token] = index
        self._idx_to_token[index] = token
    return index
```

문자 RNN으로 성씨 국적 분류하기 : 모델

```
class SurnameClassifier(nn.Module):
    """ RNN으로 특성을 추출하고 MLP로 분류하는 분류 모델 """
    def __init__(self, embedding_size, num_embeddings, num_classes,
                 rnn_hidden_size, batch_first=True, padding_idx=0):
        """
        매개변수:
            embedding_size (int): 문자 임베딩의 크기
            num_embeddings (int): 임베딩할 문자 개수
            num_classes (int): 예측 벡터의 크기
                노트: 국적 개수
            rnn_hidden_size (int): RNN의 은닉 상태 크기
            batch_first (bool): 입력 텐서의 0번째 차원이 배치인지 시퀀스인지 나타내는 플래그
            padding_idx (int): 텐서 패딩을 위한 인덱스;
                torch.nn.Embedding을 참고하세요
        """
        super(SurnameClassifier, self).__init__()

        self.emb = nn.Embedding(num_embeddings=num_embeddings,
                                embedding_dim=embedding_size,
                                padding_idx=padding_idx)
        self.rnn = ElmanRNN(input_size=embedding_size,
                            hidden_size=rnn_hidden_size,
                            batch_first=batch_first)
        self.fc1 = nn.Linear(in_features=rnn_hidden_size,
                             out_features=rnn_hidden_size)
        self.fc2 = nn.Linear(in_features=rnn_hidden_size,
                             out_features=num_classes)
```

정수로 매핑한 입력 시퀀스를 임베딩 → 각 문자가 벡터로 변환됨

각 타임스텝에서의 은닉벡터 반환

국적 클래스 수(18개)

문자 RNN으로 성씨 국적 분류하기 : 모델

모델의 정방향 연산

```
def forward(self, x_in, x_lengths=None, apply_softmax=False):
    """ 분류기의 정방향 계산

    매개변수:
        x_in (torch.Tensor): 입력 데이터 텐서
            x_in.shape는 (batch, input_dim)입니다
        x_lengths (torch.Tensor): 배치에 있는 각 시퀀스의 길이
            시퀀스의 마지막 벡터를 찾는데 사용합니다
        apply_softmax (bool): 소프트맥스 활성화 함수를 위한 플래그
            크로스-엔트로피 손실을 사용하려면 False로 지정합니다
    반환값:
        결과 텐서. tensor.shape는 (batch, output_dim)입니다.
    """
    x_embedded = self.emb(x_in)
    y_out = self.rnn(x_embedded)

    if x_lengths is not None:
        y_out = column_gather(y_out, x_lengths)
    else:
        y_out = y_out[:, -1, :]

    y_out = F.relu(self.fc1(F.dropout(y_out, 0.5)))
    y_out = self.fc2(F.dropout(y_out, 0.5))

    if apply_softmax:
        y_out = F.softmax(y_out, dim=1)

    return y_out
```

배치의 각 타임스텝별 은닉 상태 벡터 모음(3차원 텐서)

배치의 **마지막** 타임스텝 은닉 상태 벡터 반환

y_out에 렐루 활성화함수 적용 + 드롭아웃 0.5

드롭아웃 0.5

다중 클래스 분류(국적)이므로 소프트맥스 적용

문자 RNN으로 성씨 국적 분류하기 : 평가

```
테스트 손실: 1.7554078006744385;
테스트 정확도: 42.62499999999999
```

```
성씨를 입력하세요:kang
{'nationality': 'Korean', 'probability': 0.5701490640640259, 'surname': 'kang'}
```

```
성씨를 입력하세요:DiCaprio
{'nationality': 'Italian', 'probability': 0.4410761594772339, 'surname': 'DiCaprio'}
```

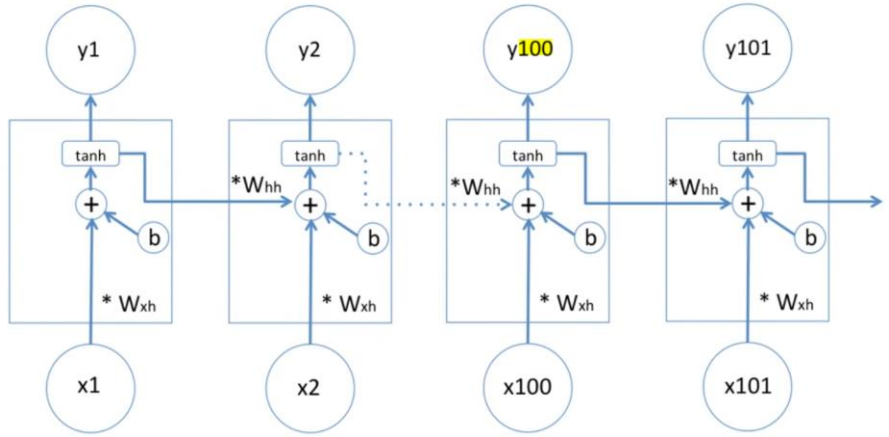
엘만 RNN의 문제점 – 길이가 긴 시퀀스 처리

(나)

새는 새장 밖으로 나가지 못한다.
매번 머리를 부딪치고 날개를 상하고 나야 보이는,
창살 사이의 간격보다 큰, 몸뚱어리.
하늘과 산이 보이고 ㉢울음 실은 공기가 자유로이 드나드는
그러나 살랑거리며 날개를 굳게 다리에 매달아 놓는,
그 적당한 간격은 슬프다.
그 창살의 간격보다 넓은 몸은 슬프다.
넓게, 힘차게 뻗을 날개가 있고
㉣날개를 힘껏 떠받쳐 줄 공기가 있지만
새는 다만 네 발 달린 짐승처럼 걷는다.
부지런히 걸어 다리가 굵어지고 튼튼해져서
닭처럼 날개가 귀찮아질 때까지 걷는다.
새장 문을 활짝 열어 놓아도 날지 않고
닭처럼 모이를 향해 달려갈 수 있을 때까지 걷는다.
㉤걸으면서, 가끔, 창살 사이를 채우고 있는 바람을
부리로 쪼아 본다, 아직도 벽이 아니고
공기라는 걸 증명하려는 듯.
유리보다도 더 환하고 선명하게 전망이 보이고
울음 소리 숨내음 자유롭게 움직이도록 고안된 공기,
그 최첨단 신소재의 부드러운 질감을 음미하려는 듯.

– 김기택, 「새」 –

단어, 문장이 아닌 긴 글도 RNN이 학습할 수 있을까?



100이 넘는 타임 스텝

엘만 RNN의 문제점 – 기울기 소실

가중치 업데이트 ➡ $$W := W - \alpha \frac{\partial}{\partial W} cost(W)$$

학습률

연쇄법칙(chain rule)

합성함수의 미분, 겉미분*속미분

$$z = (x + y)^2$$

$$\frac{\partial z}{\partial x} = \frac{\partial (x^2 + 2xy + y^2)}{\partial x} = 2x + 2y$$

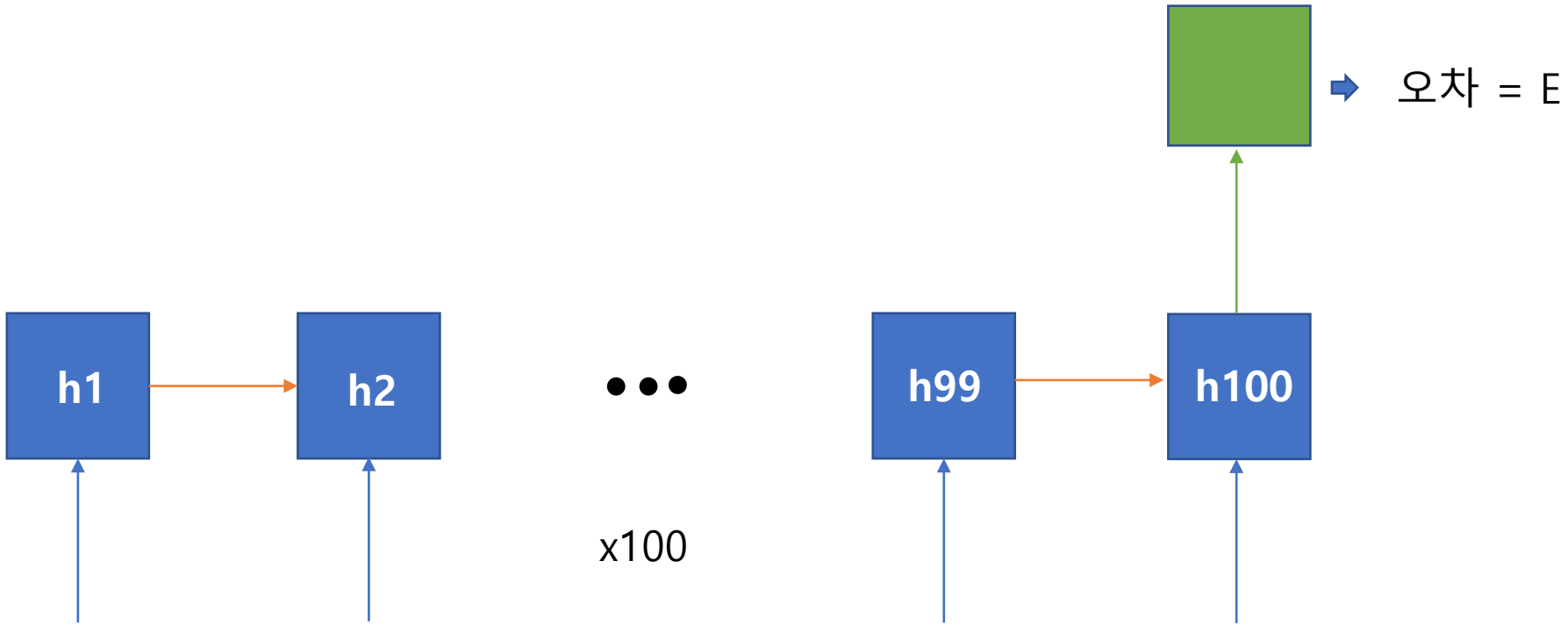
$$z = t^2, \quad t = x + y$$

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} = 2t \cdot 1 = 2t$$

← Z에 대한 함수와 T에 대한 함수를 미분한 값의 **곱**

→ 합성이 100번 정도 되었다면?

엘만 RNN의 문제점 – 기울기 소실



$$\frac{\partial E}{\partial W} = \frac{\partial h100}{\partial h99} \times \frac{\partial h99}{\partial h98} \cdots \frac{\partial h2}{\partial h1} \times \frac{\partial h1}{\partial w}$$

만약 **1보다 작은** 값을

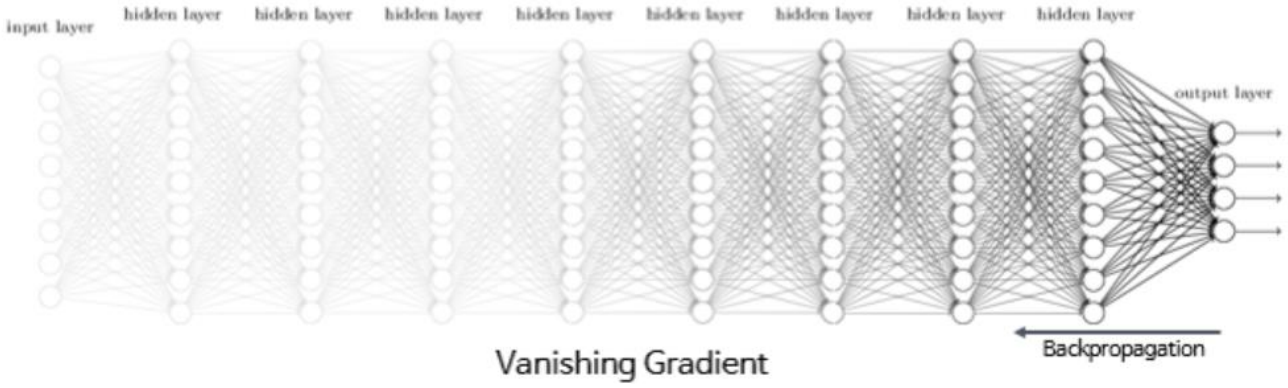
100번 곱한다면?

→ **0으로 수렴**

엘만 RNN의 문제점 – 기울기 소실

기울기 소실(Gradient Vanishing)

역전파 과정에서 입력층으로 갈 수록 기울기(Gradient)가 점차적으로 작아지는 현상

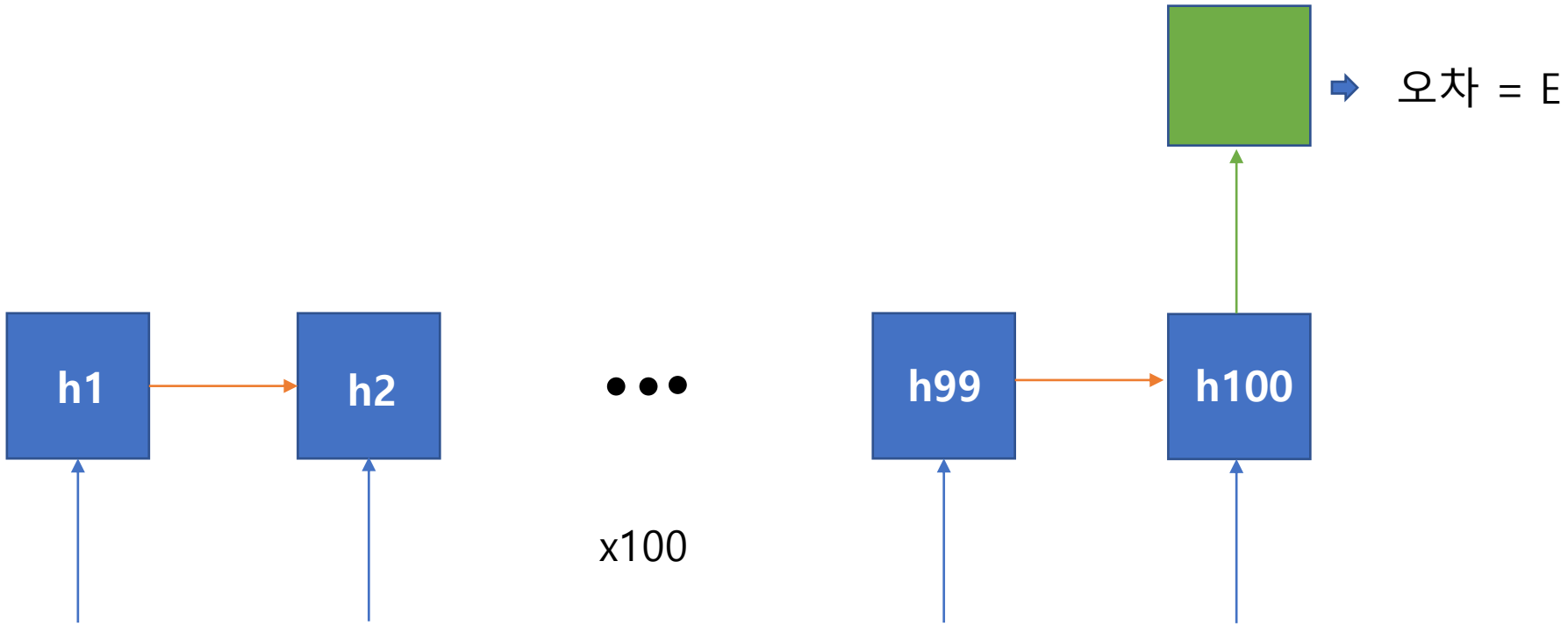


$$W := W - \alpha \frac{\partial}{\partial W} cost(W)$$

학습률

0에 가깝다면 가중치는 거의 갱신되지 않음

엘만 RNN의 문제점 – 기울기 폭주



$$\frac{\partial E}{\partial W} = \frac{\partial h100}{\partial h99} \times \frac{\partial h99}{\partial h98} \cdots \frac{\partial h2}{\partial h1} \times \frac{\partial h1}{\partial w}$$

만약 **1보다 큰** 값을

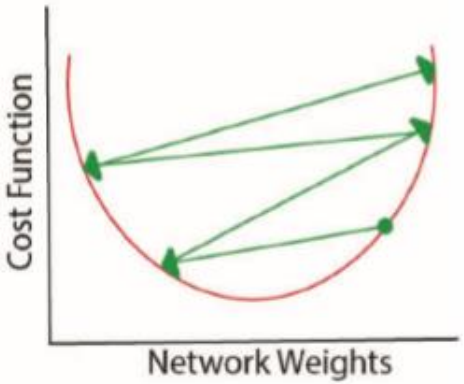
100번 곱한다면?

→ 매우 **큰 값**

엘만 RNN의 문제점 – 기울기 폭주

기울기 폭주(Gradient Exploding)

역전파 과정에서 입력층으로 갈 수록 기울기(Gradient)가 발산하는 현상



$$W := W - \underbrace{\alpha}_{\text{학습률}} \frac{\partial}{\partial W} cost(W)$$

가중치가 매우 급격하게 갱신됨

→ 수렴되지 못하고 **발산**

엘만 RNN의 문제점 – 장기 의존성 문제

"나는 그제 가족과 함께 광화문 광장에서 즐겁게 놀았다"

ㄱ

나는 그제 가족과 함께 **광화문 광장에서 즐겁게 놀았다** (누가 놀았다는 거지...?)

→ 타임스텝이 길어질수록 앞에 있던 단어에 대한 정보 감소

Why?

매 타임스텝마다 정보의 유익성 관계 없이 은닉 벡터 업데이트

엘만 RNN의 해결책 - 게이팅

RNN이 은닉 벡터를 선택적으로 업데이트 해야 함

= 필요 없는 정보는 잊고, 중요한 정보는 기억

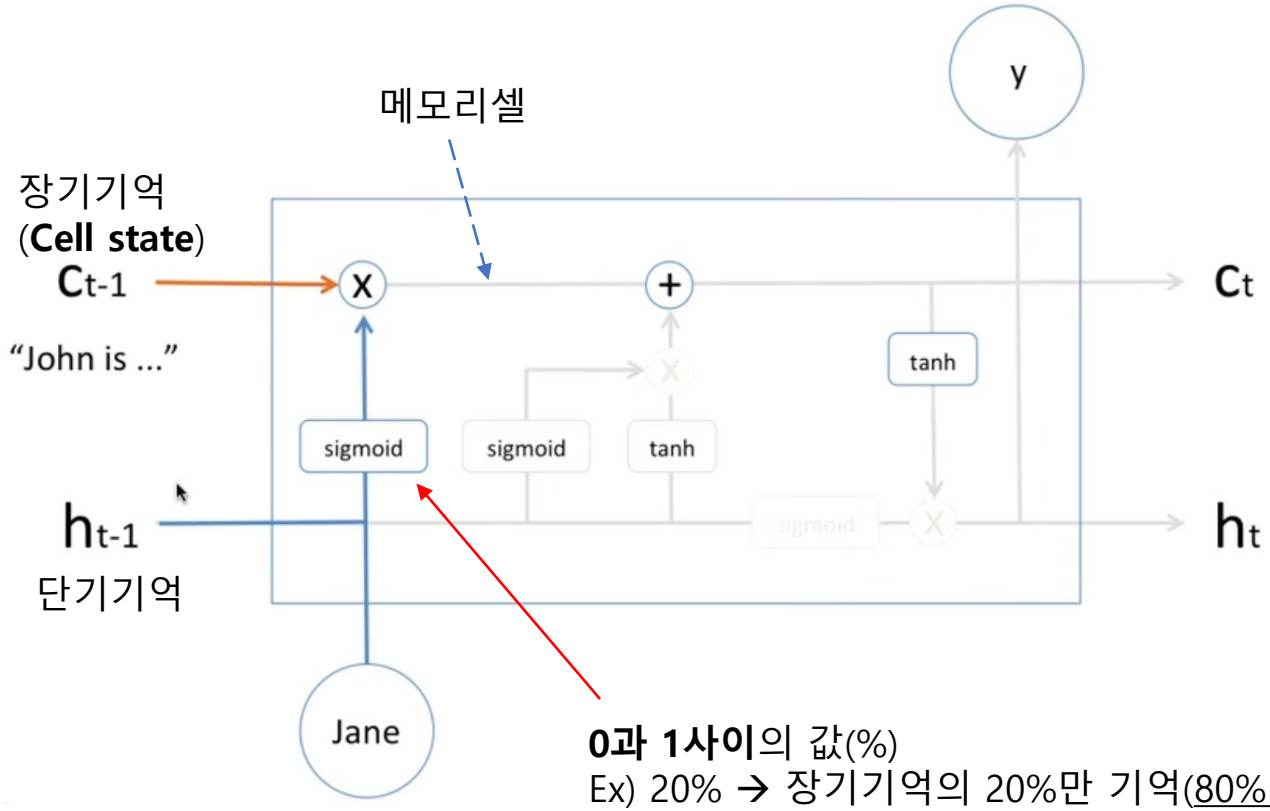
a에 b를 더할 때, b가 더해지는 양을 조절하고 싶다면?

$$a + \lambda b$$

➔ 람다(0~1)를 b에 **곱하여** 기여도 조절

'게이트' 혹은 '스위치'

| LSTM(long short-term memory network)

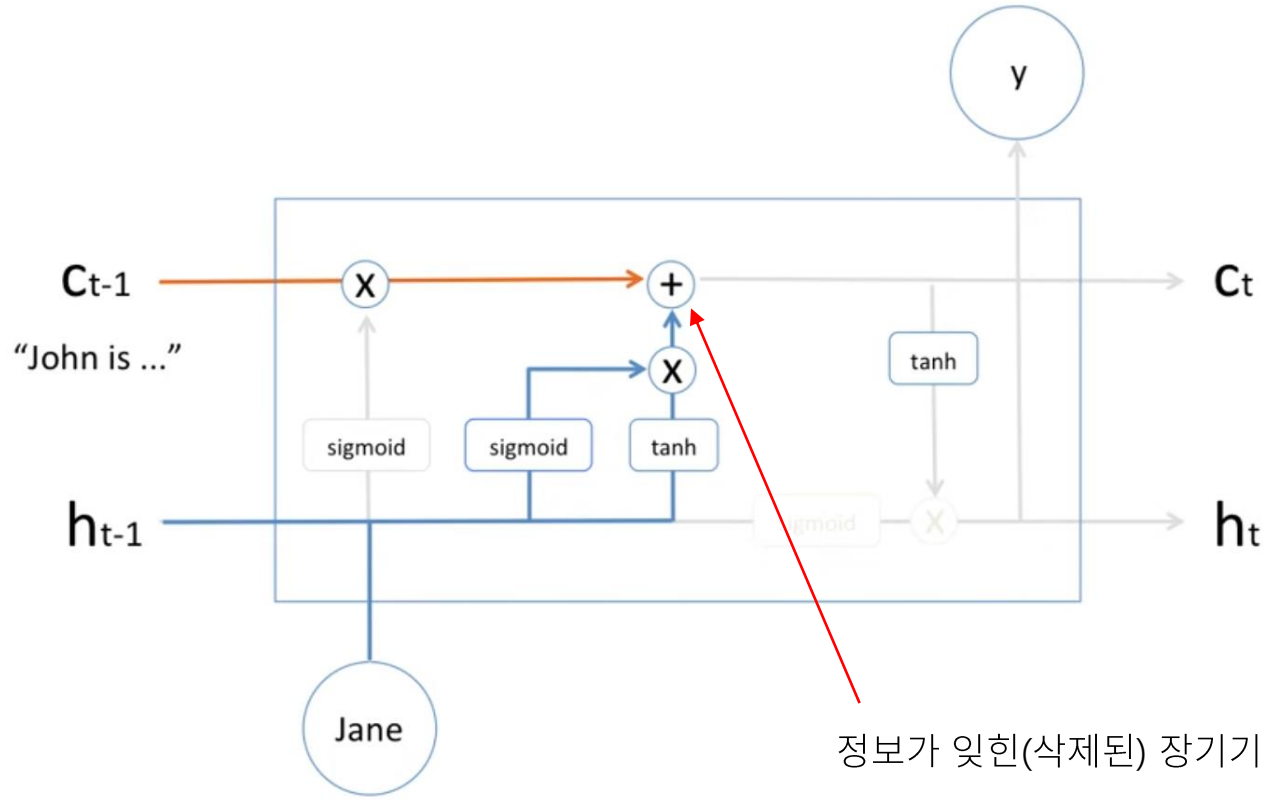


0과 1사이의 값(%)
Ex) 20% → 장기기억의 20%만 기억(80% 정도는 기억에서 지워라!)

"새로 들어온 정보와 관련 없는 예전 기억을 **잊는** 메커니즘"

참고 : [딥러닝] LSTM 쉽게 이해하기 https://youtu.be/bX6GLbpw-A4

| LSTM(long short-term memory network)

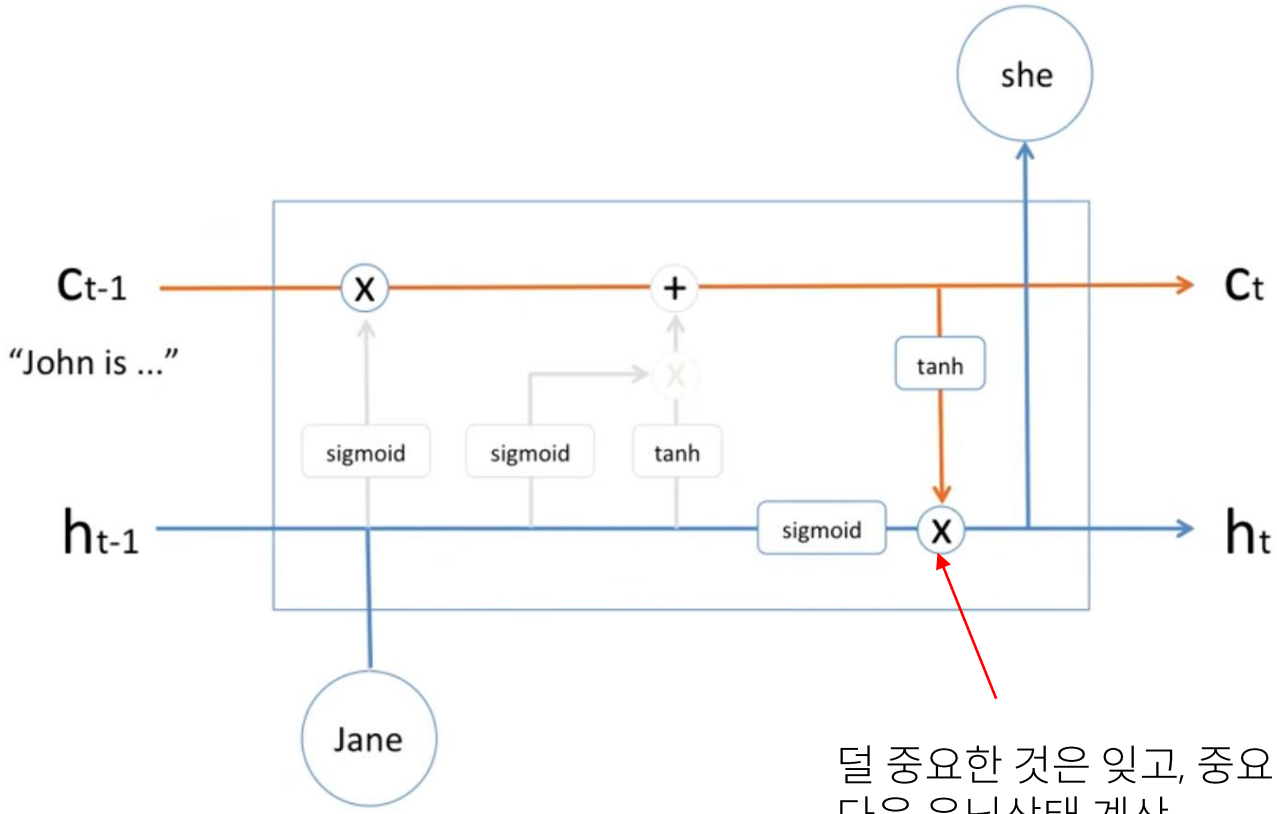


정보가 잊힌(삭제된) 장기기억에, 새로 입력된 정보를 더함

"상대적으로 중요한 새로운 정보를 **기억**하는 메커니즘"

참고 : [딥러닝] LSTM 쉽게 이해하기 https://youtu.be/bX6GLbpw-A4

| LSTM(long short-term memory network)



덜 중요한 것은 잊고, 중요한 것은 기억한 장기기억을 참고하여
다음 은닉상태 계산

참고 : [딥러닝] LSTM 쉽게 이해하기 https://youtu.be/bX6GLbpw-A4

문자 RNN으로 성씨 생성하기

문자 시퀀스에 확률을 할당

→ 새로운 시퀀스 생성

Surname Dataset (18개국 성씨 10000개)

	nationality	nationality_index	split	surname
0	Arabic	15	train	Totah
1	Arabic	15	train	Abboud
2	Arabic	15	train	Fakhoury
3	Arabic	15	train	Srour
4	Arabic	15	train	Sayegh

타깃은 무엇일까?

문자 RNN으로 성씨 생성하기 : Vectorizer

하나의 토큰 시퀀스(문자 한글자)에서 토큰을 하나씩 **엇갈리게** 샘플과 타깃 구성

샘플 : [<BOS> <K> <a> <n> <g > <Mask>…]

타깃 : [<K> <a> <n> <g> <EOS> <Mask>…]

```
indices = [self.char_vocab.begin_seq_index]
indices.extend(self.char_vocab.lookup_token(token) for token in surname)
indices.append(self.char_vocab.end_seq_index)
```

```
from_vector = np.empty(vector_length, dtype=np.int64)
from_indices = indices[:-1]  # 샘플
from_vector[:len(from_indices)] = from_indices
from_vector[len(from_indices):] = self.char_vocab.mask_index

to_vector = np.empty(vector_length, dtype=np.int64)
to_indices = indices[1:]  # 타깃
to_vector[:len(to_indices)] = to_indices
to_vector[len(to_indices):] = self.char_vocab.mask_index

return from_vector, to_vector
```

문자 RNN으로 성씨 생성하기 : 모델

```
class SurnameGenerationModel(nn.Module):
    def __init__(self, char_embedding_size, char_vocab_size, rnn_hidden_size,
                 batch_first=True, padding_idx=0, dropout_p=0.5):
        """
        매개변수:
            char_embedding_size (int): 문자 임베딩 크기
            char_vocab_size (int): 임베딩될 문자 개수
            rnn_hidden_size (int): RNN의 은닉 상태 크기
            batch_first (bool): 0번째 차원이 배치인지 시퀀스인지 나타내는 플래그
            padding_idx (int): 텐서 패딩을 위한 인덱스;
                torch.nn.Embedding를 참고하세요
            dropout_p (float): 드롭아웃으로 활성화 출력을 0으로 만들 확률
        """
        super(SurnameGenerationModel, self).__init__()

        self.char_emb = nn.Embedding(num_embeddings=char_vocab_size,
                                     embedding_dim=char_embedding_size,
                                     padding_idx=padding_idx)

        self.rnn = nn.GRU(input_size=char_embedding_size,
                          hidden_size=rnn_hidden_size,
                          batch_first=batch_first)

        self.fc = nn.Linear(in_features=rnn_hidden_size,
                            out_features=char_vocab_size)

        self._dropout_p = dropout_p
```

문자 RNN으로 성씨 생성하기 : 시퀀스 생성

```python
def sample_from_model(model, vectorizer, num_samples=1, sample_size=20,
                      temperature=1.0):
    """모델이 만든 인덱스 시퀀스를 샘플링합니다.

    매개변수:
        model (SurnameGenerationModel): 훈련 모델
        vectorizer (SurnameVectorizer): SurnameVectorizer 객체
        num_samples (int): 샘플 개수
        sample_size (int): 샘플의 최대 길이
        temperature (float): 무작위성 정도
            0.0 < temperature < 1.0 이면 최대 값을 선택할 가능성이 높습니다
            temperature > 1.0 이면 균등 분포에 가깝습니다
    반환값:
        indices (torch.Tensor): 인덱스 행렬
        shape = (num_samples, sample_size)
    """
    begin_seq_index = [vectorizer.char_vocab.begin_seq_index
                       for _ in range(num_samples)]
    begin_seq_index = torch.tensor(begin_seq_index,
                                   dtype=torch.int64).unsqueeze(dim=1)
    indices = [begin_seq_index]
    h_t = None

    for time_step in range(sample_size):
        x_t = indices[time_step]
        x_emb_t = model.char_emb(x_t)
        rnn_out_t, h_t = model.rnn(x_emb_t, h_t)
        prediction_vector = model.fc(rnn_out_t.squeeze(dim=1))
        probability_vector = F.softmax(prediction_vector / temperature, dim=1)
        indices.append(torch.multinomial(probability_vector, num_samples=1))
    indices = torch.stack(indices).squeeze().permute(1, 0)
    return indices
```

타임스텝마다 예측을 계산한 뒤
다음 타임스텝의 입력으로 사용

인덱스 확률에 비례하여 인덱스 선택

```python
[32] # 생성할 이름 개수
     num_names = 3
     model = model.cpu()
     # 이름 생성
     sampled_surnames = decode_samples(
         sample_from_model(model, vectorizer, num_samples=num_names),
         vectorizer)
     # 결과 출력
     print ("-"*15)
     for i in range(num_names):
         print (sampled_surnames[i])
```

```
---------------
Kond
Srahineem
Skokonki
```

문자 RNN으로 성씨 생성하기 : 시퀀스 생성

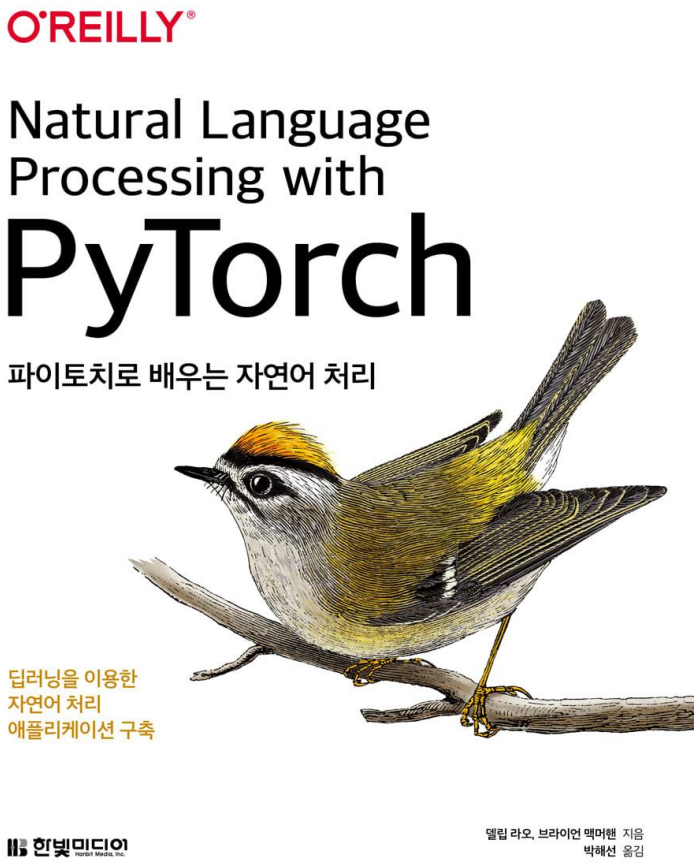
```
1번째 prediction_vector : tensor([[-4.7221, -4.9306, -4.8457,  1.0407, -3.2755,  4.8415,  1.7422,  5.4523,
          3.1282, -3.1834,  1.1327,  4.1971,  0.8482, -3.3748,  1.0463,  2.9055,
          2.3414, -3.0398,  4.6904,  0.6930, -3.3525,  1.2651, -3.2333,  4.5329,
         -3.1578,  1.2091, -3.3832,  1.9169, -0.6184, -3.1074, -3.0818,  2.6289,
         -2.9602,  1.0210, -3.3096, -3.5555,  0.5423, -3.1525, -3.9178,  0.0580,
         -3.4912, -3.4746, -3.3459,  1.8541, -3.3215, -3.3320, -3.3832, -3.6302,
         -3.9080, -3.3109, -4.7577,  0.7897, -4.3997, -0.6802, -3.8146, -4.5305,
         -3.4810, -2.0608, -4.6700, -2.3646, -2.1360, -4.9520, -1.5773, -4.8708,
         -4.0528, -3.4208, -0.8242, -2.2079, -1.9915, -2.9620, -2.9723, -4.4012,
         -4.4024, -2.5454, -4.6746, -5.1380, -4.7717, -4.0531, -4.9431, -2.7125,
         -4.4783, -5.3407, -5.2898, -4.4338, -4.9704, -3.0775, -3.6156, -4.5651],
```

```
1번째 probability_vector : tensor([[1.1954e-05, 9.7051e-06, 1.0565e-05, 3.8043e-03, 5.0792e-05, 1.7020e-
         7.6724e-03, 3.1350e-01, 3.0681e-02, 5.5694e-05, 4.1712e-03, 8.9353e-02,
         3.1382e-03, 4.5988e-05, 3.8257e-03, 2.4557e-02, 1.3969e-02, 6.4292e-05,
         1.4633e-01, 2.6871e-03, 4.7027e-05, 4.7615e-03, 5.2983e-05, 1.2501e-01,
         5.7133e-05, 4.5021e-03, 4.5607e-05, 9.1373e-03, 7.2404e-04, 6.0091e-05,
         6.1645e-05, 1.8623e-02, 6.9620e-05, 3.7303e-03, 4.9088e-05, 3.8389e-05,
         2.3113e-03, 5.7438e-05, 2.6719e-05, 1.4240e-03, 4.0937e-05, 4.1623e-05,
         4.7338e-05, 8.5814e-03, 4.8507e-05, 4.8001e-05, 4.5604e-05, 3.5625e-05,
         2.6982e-05, 4.9023e-05, 1.1537e-05, 2.9600e-03, 1.6503e-05, 6.8061e-04,
         2.9626e-05, 1.4479e-05, 4.1356e-05, 1.7113e-04, 1.2594e-05, 1.2629e-04,
         1.5873e-04, 9.4992e-06, 2.7754e-04, 1.0303e-05, 2.3347e-05, 4.3924e-05,
         5.8937e-04, 1.4772e-04, 1.8341e-04, 6.9497e-05, 6.8778e-05, 1.6477e-05,
         1.6458e-05, 1.0541e-04, 1.2537e-05, 7.8872e-06, 1.1376e-05, 2.3340e-05,
         9.5843e-06, 8.9184e-05, 1.5255e-05, 6.4401e-06, 6.7764e-06, 1.5949e-05,
         9.3263e-06, 6.1912e-05, 3.6150e-05, 1.3988e-05],
```

```
----------------
Kond
Srahineem
Skokonki
```

```
최종 indices : tensor([[ 2, 24,  5, 25, 12,  3, 43,  5, 53,  3, 53, 19,  8, 23, 28,  3, 21,  7,
          6, 12, 18],
        [ 2, 17, 15,  7,  8, 23, 25, 18, 18, 21,  3, 43,  8, 16, 31, 18,  8,  3,
         27,  3, 43],
        [ 2, 17, 14,  5, 14,  5, 25, 14, 23,  3,  3, 25, 18, 18, 53,  3, 11, 31,
          5,  6,  7]])
```

| What's next?



8장 자연어 처리를 위한 시퀀스 모델링 - 고급

- 시퀀스-투-시퀀스 모델링(S2S)
- 예제 : 신경망 기계 번역

Sequence to Sequence Learning with Neural Networks (NIPS 2014)