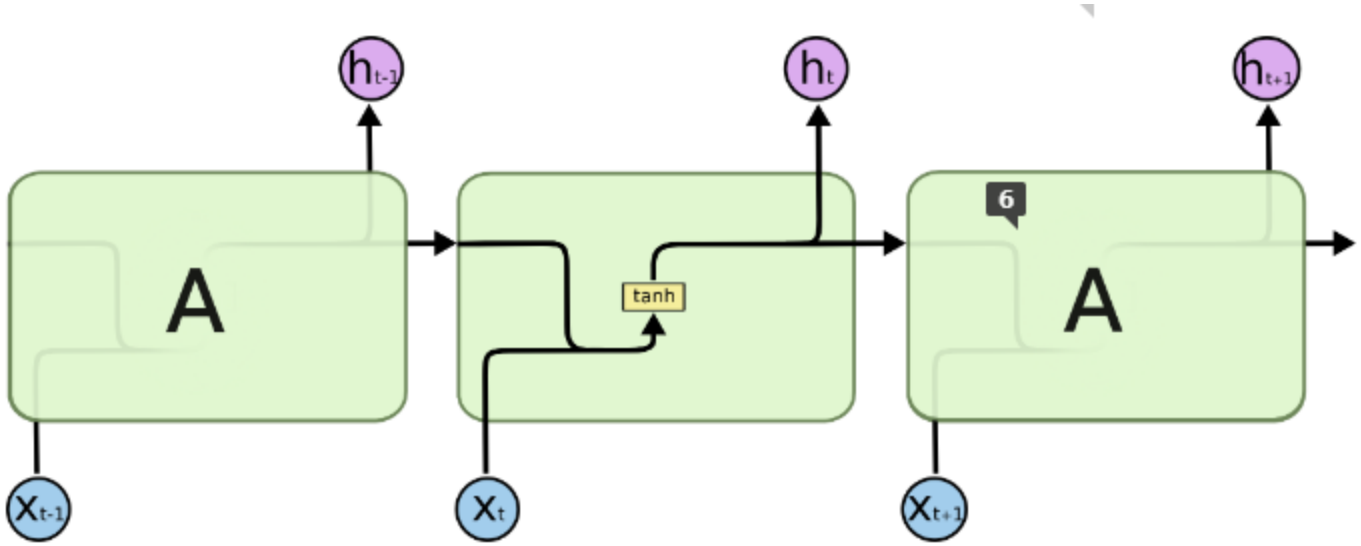
# 7장
# 자연어 처리를 위한 시퀀스 모델
# -중급-

발표자 : 최환규

# 목차

- 엘만 RNN의 문제점
- 게이팅 : 엘만 RNN의 문제 해결책
- 실습
  - RNN으로 성씨 생성하기

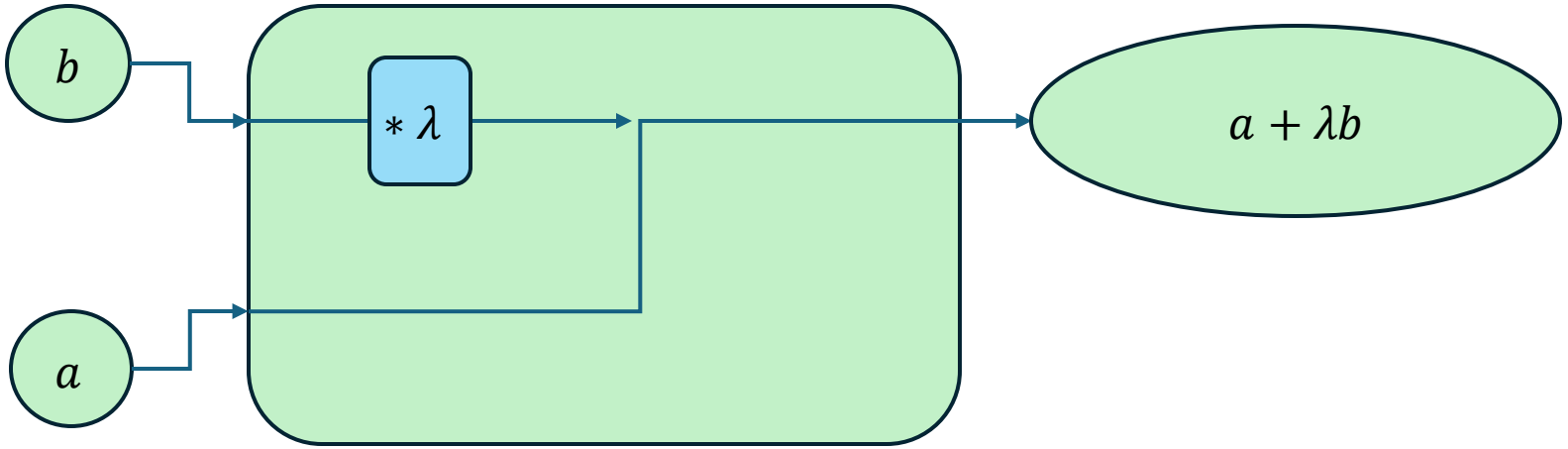
# 엘만 RNN의 문제점




The repeating module in a standard RNN contains a single layer.

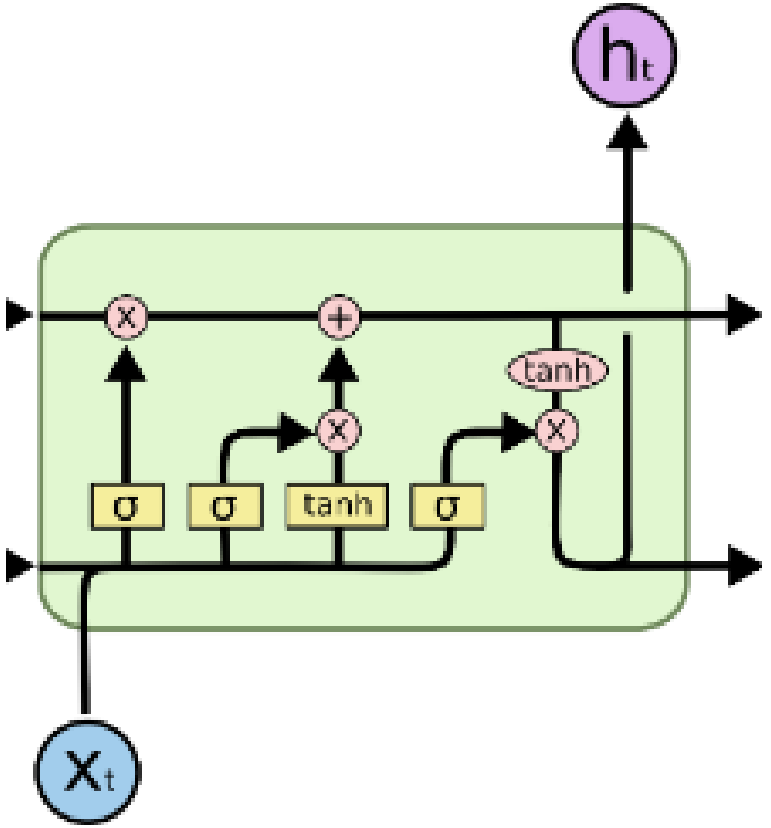
- 멀리 떨어진 정보를 예측에 사용하지 못함.
  - 멀리 떨어진 정보의 유지가 어려움.
- 불안정한 그레디언트
  - 극단적인 수치문제 (그레디언트 소실 또는 그레디언트 폭주)

# 게이팅 : 엘만 RNN의 문제 해결책



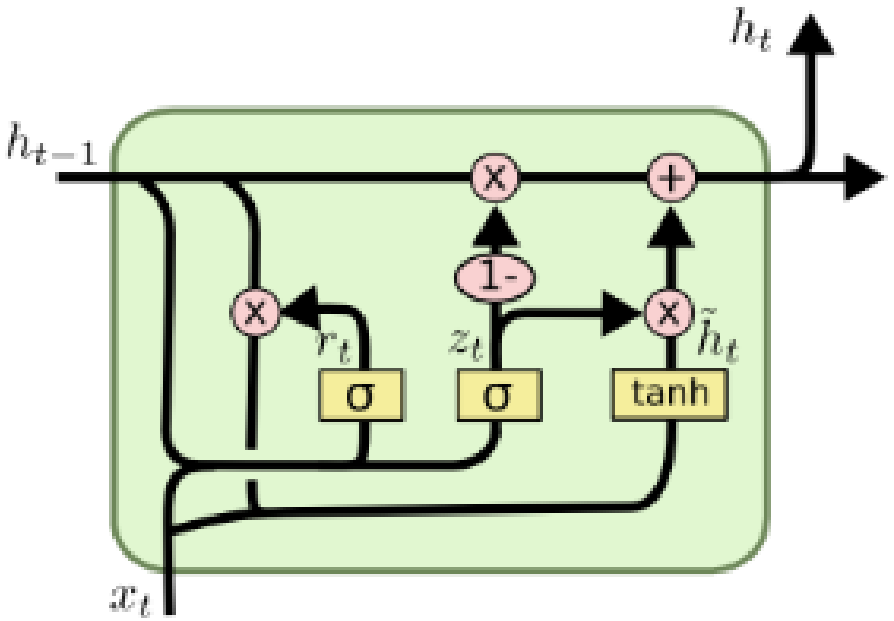

- $a + \lambda b$
- b가 덧셈에 얼마나 영향을 끼칠 수 있는지를 조절해주는 것

# LSTM




- $h_t = \mu(h_{t-1}, x_t)h_{t-1} + \lambda(h_{t-1}, x_t), F(h_{t-1}, x_t)$
- LSTM은 조건에 따라
  이전은닉 상태의 값을 지우거나, 갱신합니다.

# GRU




$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

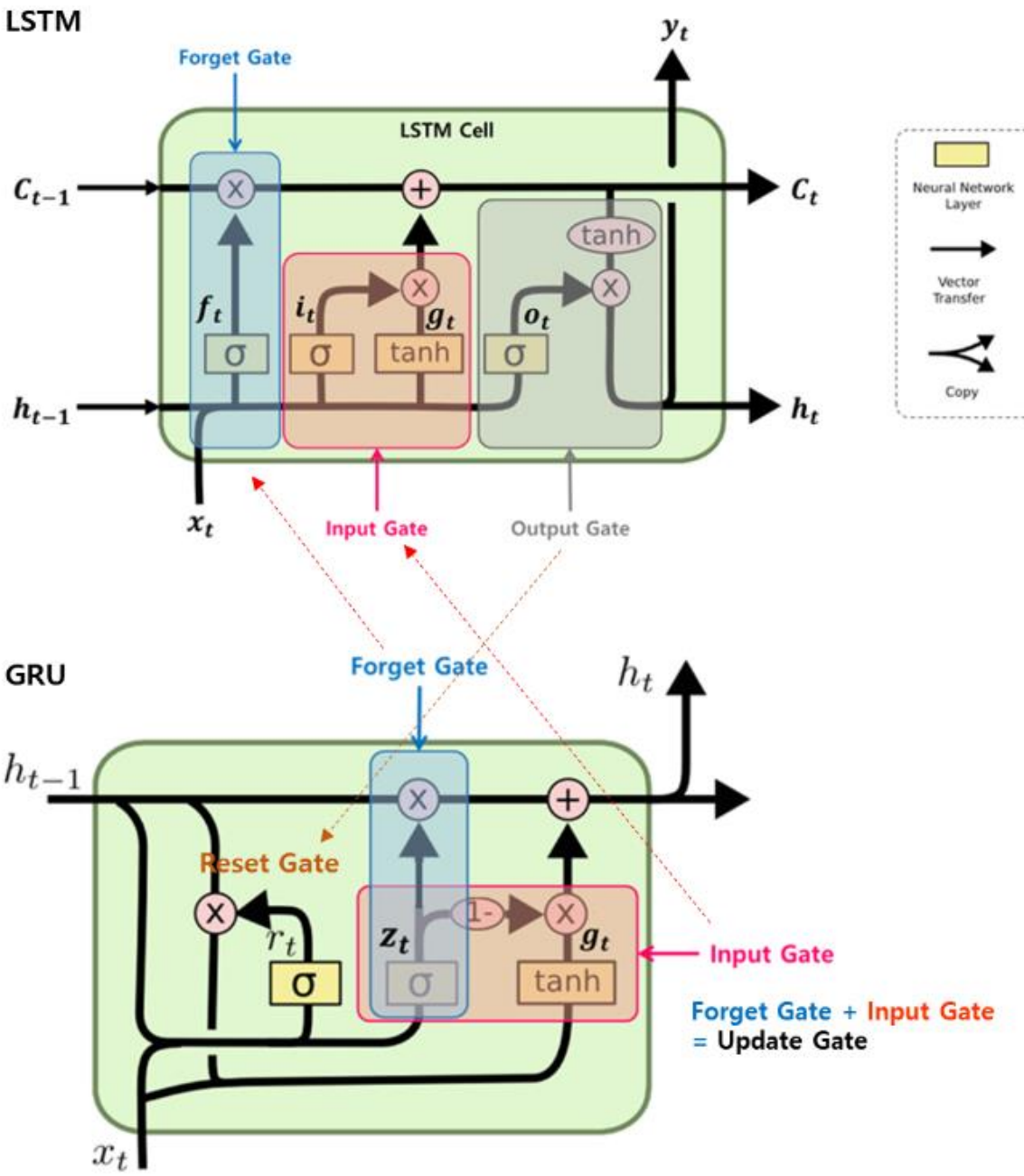
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

- LSTM을 간소화 한 버전.

# LSTM vs GRU


LSTM
Forget Gate
LSTM Cell
$y_t$
$C_{t-1}$
$C_t$
$h_{t-1}$
$h_t$
$x_t$
$f_t$
$i_t$
$g_t$
$o_t$
σ
tanh
Input Gate
Output Gate
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy
σ : Logistic
GRU
Forget Gate
$h_t$
$h_{t-1}$
Reset Gate
$r_t$
$z_t$
1-
$g_t$
Input Gate
Forget Gate + Input Gate
= Update Gate
$x_t$

# LSTM VS GRU

**1.게이트의 개수:**

1. LSTM은 입력, 망각, 출력 세 가지 게이트를 사용합니다.
2. GRU는 재설정과 업데이트 두 가지 게이트를 사용합니다.

**2.셀 상태와 은닉 상태의 분리:**

1. LSTM은 셀 상태와 은닉 상태를 명시적으로 분리하여 계산합니다.
2. GRU는 하나의 은닉 상태만을 사용합니다.

**3.계산 복잡성:**

1. LSTM은 더 많은 파라미터를 가지고 있어 계산이 더 복잡합니다.
2. GRU는 더 간단한 구조를 가지고 있어 계산이 더 경제적입니다.

# 실습 : RNN으로 성씨 생성하기

- 학습에 사용된 데이터셋 소개
- 벡터변환 클래스
- ElmanRNN GRU로 바꾸기
- 분류모델 소개
  - 모델 1: 조건이 없는 생성모델
  - 모델 2: 조건이 있는 생성모델
  - 모델훈련 결과.

# 학습에 사용된 데이터셋

- 6장에서 사용한 성씨데이터를 사용

# Dataset class

- 예측 타깃에 대한 정수 시퀀스를 반환해줌.

```
def __getitem__(self, index):
    """파이토치 데이터셋의 주요 진입 메서드

    매개변수:
        index (int): 데이터 포인트에 대한 인덱스
    반환값:
        데이터 포인트(x_data, y_target, class_index)를 담고 있는 딕셔너리
    """
    row = self._target_df.iloc[index]

    from_vector, to_vector = ₩
        self._vectorizer.vectorize(row.surname, self._max_seq_length)

    nationality_index = ₩
        self._vectorizer.nationality_vocab.lookup_token(row.nationality)

    return {'x_data': from_vector,
            'y_target': to_vector,
            'class_index': nationality_index}
```

# 벡터 변환 클래스

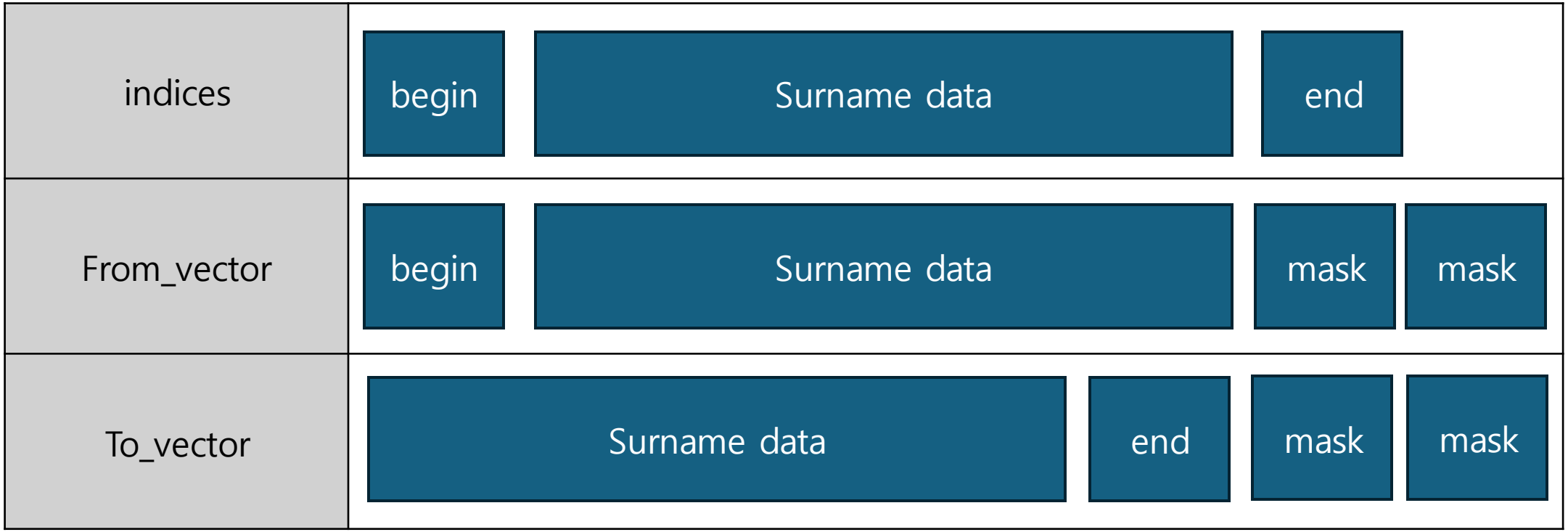
- Sequence Vocabulary로 개별토큰을 정수로 매핑
- Surname Vectorizer로 정수매핑을 관리
- DataLoader로 Vectorizer의 출력, 결과를 미니배치로 생성

# SurnameVectorizer 클래스

```
if vector_length < 0:
    vector_length = len(indices) - 1

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

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

return from_vector, to_vector
```

- From_vector : 입력으로 사용되는 정수 시퀀스
- To_vector : 출력으로 사용되는 정수 시퀀스

# from_vector, to_vector



# Elman RNN -> GRU로 바꾸기

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

self.rnn_cell = nn.RNNCell(input_size, hidden_size)
```



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

- RNNCell 부분을 GRU로 변경하면 끝.

# 조건이 없는 모델

- 성씨를 생성하기 전 국적 정보를 사용하지 않음

```
"""
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.nation_emb = nn.Embedding(num_embeddings=num_nationalities,
                               embedding_dim=rnn_hidden_size)

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
```

# 조건이 없는 모델

- Linear층 계산을 위해 벡터를 3->2 차원으로 변환해줌.
- 계산을 마친 후 2->3 차원으로 다시 변환.

```
"""모델의 정방향 계산

매개변수:
    x_in (torch.Tensor): 입력 데이터 텐서
        x_in.shape는 (batch, input_dim)입니다.
    apply_softmax (bool): 소프트맥스 활성화를 위한 플래그로 훈련시에는
반환값:
    결과 텐서. tensor.shape는 (batch, char_vocab_size)입니다.
"""
x_embedded = self.char_emb(x_in)

y_out, _ = self.rnn(x_embedded)

batch_size, seq_size, feat_size = y_out.shape
y_out = y_out.contiguous().view(batch_size * seq_size, feat_size)

y_out = self.fc(F.dropout(y_out, p=self._dropout_p))

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

new_feat_size = y_out.shape[-1]
y_out = y_out.view(batch_size, seq_size, new_feat_size)

return y_out
```

# 조건이 없는 모델 샘플링

- 어떤 성씨를 생성 했는지 조사하여 질적으로 평가.

```
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
```

# 조건이 없는 모델 샘플링

```python
def decode_samples(sampled_indices, vectorizer):
    """인덱스를 성씨 문자열로 변환합니다

    매개변수:
        sampled_indices (torch.Tensor): `sample_from_model` 함수에서 얻은 인덱스
        vectorizer (SurnameVectorizer): SurnameVectorizer 객체
    """
    decoded_surnames = []
    vocab = vectorizer.char_vocab

    for sample_index in range(sampled_indices.shape[0]):
        surname = ""
        for time_step in range(sampled_indices.shape[1]):
            sample_item = sampled_indices[sample_index, time_step].item()
            if sample_item == vocab.begin_seq_index:
                continue
            elif sample_item == vocab.end_seq_index:
                break
            else:
                surname += vocab.lookup_index(sample_item)
        decoded_surnames.append(surname)
    return decoded_surnames
```

- 샘플링 인덱스를 읽을 수 있도록 문자열로 변환하여 반환

```python
# 생성할 이름 개수
num_names = 10
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])
```

```
---------------
Anbanboy
Gatn
Kekaiva
Vaskin
Tongif
Pralarovei
Vichacovs
Zalbata
Titpan
Tepkanla
```

# 모델 손실 측정

```python
def normalize_sizes(y_pred, y_true):
    """텐서 크기 정규화

    매개변수:
        y_pred (torch.Tensor): 모델의 출력
            3차원 텐서이면 행렬로 변환합니다.
        y_true (torch.Tensor): 타깃 예측
            행렬이면 벡터로 변환합니다.
    """
    if len(y_pred.size()) == 3:
        y_pred = y_pred.contiguous().view(-1, y_pred.size(2))
    if len(y_true.size()) == 2:
        y_true = y_true.contiguous().view(-1)
    return y_pred, y_true
```

- 예측과 타깃을 손실 함수가 기대하는 크기로 정규화 (예측은 2차원, 타깃은 1차원)

# 조건이 없는 모델 결과

```
1 print("테스트 손실: {};".format(train_state['test_loss']))
2 print("테스트 정확도: {}".format(train_state['test_acc']))
```

```
테스트 손실: 2.5687183340390525;
테스트 정확도: 24.895795429495628
```

# 조건이 있는 모델

- nation_emb층이 추가됨

```
"""
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.nation_emb = nn.Embedding(num_embeddings=num_nationalities,
                               embedding_dim=rnn_hidden_size)

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
```

# 조건이 있는 모델

- 국적 인덱스
  ->RNN의 은닉 상태를 초기화 하는데 사용

```
"""모델의 정방향 계산

매개변수:
    x_in (torch.Tensor): 입력 데이터 텐서
        x_in.shape는 (batch, max_seq_size)입니다.
    nationality_index (torch.Tensor): 각 데이터 포인트를 위한 국적 인덱
        RNN의 은닉 상태를 초기화하는데 사용합니다.
    apply_softmax (bool): 소프트맥스 활성화를 위한 플래그로 훈련시에는
반환값:
    결과 텐서. tensor.shape는 (batch, char_vocab_size)입니다.
"""
x_embedded = self.char_emb(x_in)

# hidden_size: (num_layers * num_directions, batch_size, rnn_hidden_siz
nationality_embedded = self.nation_emb(nationality_index).unsqueeze(0)

y_out, _ = self.rnn(x_embedded, nationality_embedded)

batch_size, seq_size, feat_size = y_out.shape
y_out = y_out.contiguous().view(batch_size * seq_size, feat_size)

y_out = self.fc(F.dropout(y_out, p=self._dropout_p))

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

new_feat_size = y_out.shape[-1]
y_out = y_out.view(batch_size, seq_size, new_feat_size)

return y_out
```

# 조건이 있는 모델 샘플링

- 국적 별로 샘플링

```
    """모델이 만든 인덱스 시퀀스를 샘플링합니다.

    매개변수:
        model (SurnameGenerationModel): 훈련 모델
        vectorizer (SurnameVectorizer): SurnameVectorizer 객체
        nationalities (list): 국적을 나타내는 정수 리스트
        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)
    """
    num_samples = len(nationalities)
    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]
    nationality_indices = torch.tensor(nationalities, dtype=torch.int64).unsquee
    h_t = model.nation_emb(nationality_indices)

    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
```

```python
model = model.cpu()
for index in range(len(vectorizer.nationality_vocab)):
    nationality = vectorizer.nationality_vocab.lookup_index(index)
    print("{} 샘플: ".format(nationality))
    sampled_indices = sample_from_model(model, vectorizer,
                                        nationalities=[index] * 3,
                                        temperature=0.7)
    for sampled_surname in decode_samples(sampled_indices, vectorizer):
        print("-  " + sampled_surname)
```

```
Arabic 샘플:
-  Bakin
-  Heran
-  Soib
Chinese 샘플:
-  Luag
-  Rur
-  Dao
Czech 샘플:
-  Ponnoir
-  Stonaj
-  Teutche
```

# 조건이 있는 모델 결과

```
1 print("테스트 손실: {};".format(train_state['test_loss']))
2 print("테스트 정확도: {}".format(train_state['test_acc']))
```

```
테스트 손실: 2.458139201000529 96;
테스트 정확도: 28.882996326062 82
```