파이토치로 배우는 자연어 처리

-7장 : 자연어 처리를 위한 시퀀스 모델링 - 중급

이상윤

시퀀스 예측

- 목표
  - 시퀀스 예측 : 시퀀스의 각 항목에 레이블을 할당하여 주어진 시퀀스를 기반으로 다음 단어 등을 예측하는 자연어 모델링

    ex) 다음 단어를 예측하는 자연어 모델링, 단어의 품사를 예측하는 품사 태깅, 개체명 인식 등

- 6장의 엘만 RNN과 다른점?
  - 엘만 RNN은 예측 작업에 사용할 순 있으나 멀리 떨어진 의존성을 잘 감지하지 못함.
  - 게이트 네트워크라는 새로운 RNN구조와 자연어 생성 작업을 소개한다.

feed-forward network <> 순환 신경망 (RNN)

엘만 RNN의 문제점

1. 멀리 떨어진 정보를 예측에 사용하지 못함

- 6장의 RNN에서 타임 스텝마다 **정보의 유익성**에 상관없이 은닉 상태 벡터를 업데이트 했다. 그 결과 RNN은 은닉 상태에 어떤 값을 유지하고 버릴지 제어하지 못하게 되었다.

1. 그레디언트가 불안정함.

- RNN은 현재 타임 스텝의 입력 벡터와 이전 타임 스텝의 은닉 상태 벡터를 사용하여 타임 스텝마다 은닉 상태 벡터를 계산하는데, 이 계산이 RNN을 강력하게 만들기도 하지만 극단적인 수치 문제도 발생시킬 수 있다.
- 그레디언트가 통제되지 않고 0이나 무한대를 만드는 경향이 있다. (그레디언트 소실/폭주) ReLU함수, 그레디언트 클리핑, 가중치 초기화 등으로 문제를 다루지만 불충분하다.

엘만 RNN의 문제 해결책 : 게이팅

a+b를 수행할 때 b가 더해지는 양을 제어하고 싶다면?

- a+xb (0<=x<=1) : x를 b가 포함되는 양을 조절하는 스위치 혹은 게이트라고 생각할 수 있다.

게이트를 적용하여 RNN에 조건부 업데이트를 수행하는 방법

$$h_t = h_{t-1} + F(h_{t-1}, x_t)$$

x : 현재 입력
h : 은닉 상태 벡터
F : 순환 계산

$$h_t = h_{t-1} + \lambda(h_{t-1}, x_t)F(h_{t-1}, x_t)$$

이전 은닉 상태 벡터 (h t-1)를 업데이트하는 데 현재 입력이 얼마나 관여하는가? 를 감마가 제어하게 된다.

$$h_t = \mu(h_{t-1}, x_t)h_{t-1} + \lambda(h_{t-1}, x_t)F(h_{t-1}, x_t)$$

LSTM 신경망은 이전 은닉 상태의 값을 의도적으로 지우기도 한다.

예제 : 문자 RNN으로 성씨 생성하기

RNN으로 성씨를 생성하는 간단한 시퀀스 예측 작업

- 각 타임 스텝에서 RNN이 성씨에 포함될 수 있는 문자 집합에 대한 확률 분포를 계산한다.
- 이 확률 분포를 사용해 신경망을 최적화하여 예측을 향상시키거나 새로운 성씨를 생성할 수 있다

데이터셋

- 이전의 성씨 데이터를 사용하나, 시퀀스 예측을 위해 데이터 샘플을 구성하는 점에 차이가 있음.

모델

- 국적 정보를 사용하지 않고 성씨 문자의 시퀀스를 예측하는 조건이 없는 모델
- 초기 은닉 상태에 임베딩된 특정 국적을 활용해서 시퀀스 예측에 편향을 준 모델

SurnameDataset 클래스

- 성씨와 해당 국적으로 구성된 데이터
- 이전처럼 분류가 아닌 문자 시퀀스에 확률을 할당하여 새로운 시퀀스를 생성하도록 훈련

1. SurnameVectorizer 객체를 사용하여 현재 작업과 모델에 필요한 토큰-정수 매핑을 캡슐화
2. Vectorizer를 사용하여 입력으로 사용되는 정수 시퀀스 from_vector와 출력으로 사용되는 정수 시퀀스 to_vector를 계산

```
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

이전 예제에서처럼 성씨의 문자 시퀀스를 벡터화된 형태로 변환

- SequenceVocabulary로 개별 토큰을 정수로 매핑
- SurnameVectorizer는 정수 매핑을 관리
- DataLoader는 SurnameVectorizer의 결과를 미니배치로 만듬

SurnameVectorizer와 END-OF-SEQUENCE

시퀀스 예측 훈련 : 타임 스텝마다 토큰 샘플과 토큰 타깃을 표현하는 2개의 정수 시퀀스 존재

- 일반적으로 훈련 시퀀스가 예측 대상
- 즉 하나의 토큰 시퀀스에서 토큰을 하나씩 엇갈리게 하여 샘플과 타깃을 구성함

```python
def vectorize(self, surname, vector_length=-1):
    """ 성씨를 샘플과 타깃 벡터로 변환합니다
    성씨 벡터를 두 개의 벡터 surname[:-1]와 surname[1:]로 나누어 출력합니다.
    각 타임스텝에서 첫 번째 벡터가 샘플이고 두 번째 벡터가 타깃입니다.

    매개변수:
        surname (str): 벡터로 변경할 성씨
        vector_length (int): 인덱스 벡터의 길이를 맞추기 위한 매개변수
    반환값:
        튜플: (from_vector, to_vector)
            from_vector (numpy.ndarray): 샘플 벡터
            to_vector (numpy.ndarray): 타깃 벡터 vector
    """
    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)

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

    from_vector = np.empty(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.empty(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
```

- SurnameVectorizer.vectorize() 메소드

1. 문자열 surname을 문자를 나타내는 정수 리스트인 indices로 매핑
2. 시작 인덱스(begin_seq_index)를 indices 앞에 추가하고 종료 인덱스(end_seq_index)를 indices 뒤에 추가
3. from_vector와 to_vector를 생성

이후 ElmanRNN을 GRU 또는 LSTM으로 바꿈,
torch.nn.GRU/torch.nn.LSTM 등 손쉽게 변경 가능

모델 1 : 조건이 없는 SurnameGenerationModel

성씨를 생성하기 전에 국적 정보를 사용하지 않는 모델

- GRU가 어떤 국적에도 편향된 계산을 수행하지 않는다.

```python
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
```

```python
def forward(self, x_in, apply_softmax=False):
    """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 False가 되어야 합니다.
    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
```

- 초기 은닉 상태 벡터가 계산에 영향을 미치지 않도록 모두 0으로 초기화

일반적으로 SurnameGenerationModel은 문자 인덱스를 임베딩하여 GRU로 상태를 순서대로 계산하고 Linear층을 사용해 토큰의 예측 확률을 계산한다.

1. Embedding 층
2. GRU
3. Linear 층

모델 2 : 조건이 있는 SurnameGenerationModel

성씨를 생성할 때 국적을 고려

- 모델이 특정 성씨에 상대적으로 편향 될 수 있다.

```python
class SurnameGenerationModel(nn.Module):
    def __init__(self, char_embedding_size, char_vocab_size, num_nationalities,
                 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.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
```

```python
def forward(self, x_in, nationality_index, apply_softmax=False):
    """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 인덱스
            RNN의 은닉 상태를 초기화하는데 사용합니다.
        apply_softmax (bool): 소프트맥스 활성화를 위한 플래그로 훈련시에는 False가 되어야 합니다.
    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_size)
    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
```

- 은닉 상태 크기의 벡터로 국적을 임베딩하여 RNN의 초기 은닉 상태를 만든다.
- 모델이 수정될 때 임베딩 행렬의 값도 조정되어 성씨의 국적과 규칙에 더 민감하게 예측 가능

국적 인덱스를 RNN 은닉 층과 같은 크기의 벡터로 매핑하는 Embedding 층이 추가됨.
그다음 forward 계산에서 국적 인덱스를 매핑하고 RNN의 초기 은닉 상태로 전달됨.

모델 훈련과 결과

시퀀스의 타임 스텝마다 예측을 만들기 때문에 손실 계산을 위해 두 가지를 변경해야함

1. 계산을 위해 3차원 텐서를 2차원 텐서로 변환
- 3차원 텐서는 (배치 차원, 시퀀스, 예측 벡터) 로 이루어져 있다.
1. 가변 길이 시퀀스를 위해 마스킹 인덱스를 준비
- 마스킹된 위치는 손실을 계산하지 않음.

```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 크기 (예측 2차원, 타깃 1차원)로 정규화
각 행은 하나의 샘플, 즉 시퀀스에 있는 하나의 타임 스텝을 나타냄

모델 훈련과 결과

- 하이퍼파라미터

문자 어휘 사전의 크기에 따라 결정됨. 이 크기는 모델 입력에 나타나는 이산적인 토큰의 개수이고 타임 스텝마다 출력에 나타나는 클래스 개수
그 외 모델 하이퍼파라미터는 문자 임베딩 크기와 RNN은닉 상태 크기

```
args = Namespace(
    # 날짜와 경로 정보
    surname_csv="data/surnames/surnames_with_splits.csv",
    vectorizer_file="vectorizer.json",
    model_state_file="model.pth",
    save_dir="model_storage/ch7/model1_unconditioned_surname_generation",
    # 모델 하이퍼파라미터
    char_embedding_size=32,
    rnn_hidden_size=32,
    # 훈련 하이퍼파라미터
    seed=1337,
    learning_rate=0.001,
    batch_size=128,
    num_epochs=100,
    early_stopping_criteria=5,
    # 실행 옵션
    catch_keyboard_interrupt=True,
    cuda=True,
    expand_filepaths_to_save_dir=True,
    reload_from_files=False,
)
```

시퀀스 모델 훈련 노하우

- **가능하면 게이트가 있는 셀을 사용합니다**

  게이트 구조는 그렇지 않은 구조에서 발생하는 수치 안정성과 관련된 여러 문제를 해결하여 훈련을 쉽게 만든다.
- **가능하면 LSTM보다 GRU를 사용합니다**

  GRU는 LSTM과 거의 비슷한 성능을 제공하면서 파라미터가 훨씬 적고 계산 자원도 덜 사용한다.
- **Adam 옵티마이저를 사용합니다**

  안정적이고 다른 옵티마이저보다 빠르게 수렴하기 때문이다.
- **그레디언트 클리핑을 사용합니다.**

  수치상의 문제 발견시 훈련 과정의 그레디언트 값을 그래프로 출력해보고 범위를 가늠한 후 이상치를 클리핑하면 훈련 과정을 안정시킬 수 있다.
- **조기 종료를 사용합니다.**

  시퀀스 모델은 과적합이 되기 쉬우므로 오차가 상승하기 시작하면 훈련을 일찍 멈추는게 좋다.