9-Recurrent Neural Networks

영상, 이미지 캡션, 음성 합성, 음악 생성과 같은 많은 학습 작업은 순차 데이터 처리를 필요로 한다. 또한 번역, 대화, 로봇 제어와 같은 작업은 모델이 순차적으로 구조화된 데이터를 소화하고 출력해야 한다.

RNN은 순환 연결을 통해 순차적 데이터의 동적을 캡처하는 딥러닝 모델이다. RNN은 각 단계에서 동일한 기본 매개변수가 적용되는 시간 단계별로 펼쳐진다.

뇌 모델에서 비롯된 RNN은 기계 학습에서 실제 모델링 도구로 채택되었고, 2010년대에 필기 인식, 기계 번역, 의료 진단 등의 작업에서 뚜렷한 결과를 내면서 널리 사용되었다. 많은 기계 학습의 기본 작업에 대한 입력과 대상은 고정 길이 벡터로 쉽게 나타낼 수 없지만, 다양한 길이의 일련의 고정 길이 벡터로 나타낼 수 있다.

여기서는 텍스트 데이터를 중심으로 기본적인 연속적 구조 모델의 개념과 예제를 소개한다.

9.1 Sequences

이전까지는 입력이 단일 피처 벡터 $x \in R^d$인 모델에 중점을 두었으나, 시퀀스를 처리할 수 있는 모델을 개발할 때는 입력이 순서대로 정렬된 피처 벡터 $x_1,\ldots,x_T$로 구성된 경우에 중점을 둔다. 일부 데이터셋은 단일 대규모 시퀀스로 구성될 수 있고, 데이터가 시간에 따라 독립적이지 않을 수 있으며, 이전 시간 단계의 데이터에 의존할 수 있다.

- 9.1.1. **Autoregressive Models**

  순차적으로 구조화된 데이터를 다루기 전에 일부 실제 시퀀스 데이터를 살펴보고 기본적인 직관과 통계 도구를 구축한다. 이는 주식 가격 데이터의 autoregressive 모델에 대해 살펴볼 수 있다. 조건부 기대값을 추정하는 데에는 선형 회귀 모델을 적용할 수 있으며, 이러한 모델은 autoregressive 모델로 알려져 있다.

- 9.1.2. **Sequence Models**
  - 언어와 함께 작업할 때 전체 시퀀스의 결합 확률을 추정하는 것을 시퀀스 모델 또는 언어 모델이라고 부른다.
  - 문장의 자연스러움을 비교하거나, 문장의 가능성을 평가하거나, 가장 가능성 있는 시퀀스를 최적화하는 데 사용 가능하다.
  - 이러한 문제를 autoregressive 예측으로 축소하여 조건부 확률의 곱으로 나타낼 수 있다.

$$P(x_1, \ldots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t \mid x_{t-1}, \ldots, x_1).$$

- 9.1.2.1. **Markov Models**
  - Markov 조건이 만족되면 데이터가 Markov 조건을 만족한다고 말하며, 이는 미래가 최근의 히스토리에 대한 조건부 독립성을 가짐을 의미한다.
  - 텍스트 문서와 같은 이산 데이터의 경우, Markov 모델은 각 단어가 이전 단어에 대한 상대적 빈도 추정을 계산한다.
  - Markov 조건이 만족되면 어떤 길이의 이력도 손실 없이 버릴 수 있으며, 이를 통해 일정한 Markov 조건에 따라 모델을 훈련하는 기술이 재발된다.
- 9.1.3. **Training**

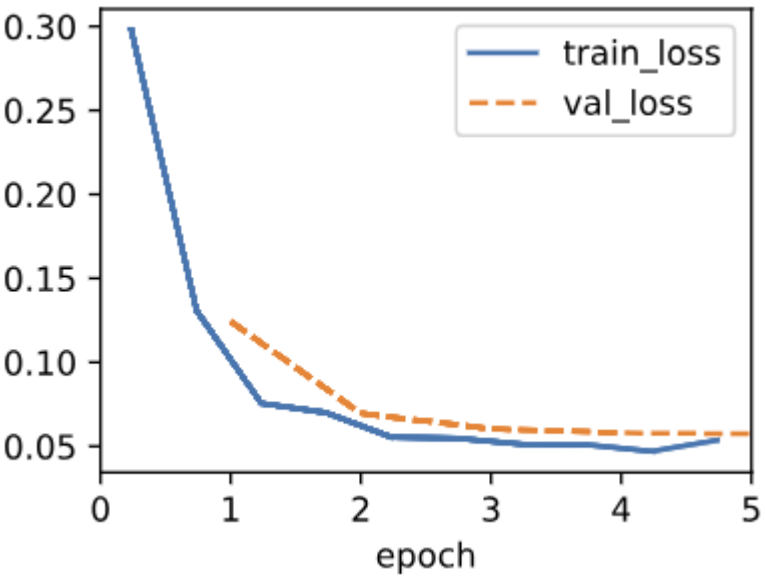
```
class Data(d2l.DataModule):
    def __init__(self, batch_size=16, T=1000, num_train=600, tau=4):
        self.save_hyperparameters()
        self.time = torch.arange(1, T + 1, dtype=torch.float32)
        self.x = torch.sin(0.01 * self.time) + torch.randn(T) * 0.2

data = Data()
d2l.plot(data.time, data.x, 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```

  - 연속 값 합성 데이터로 시작하여 이를 통해 시퀀스 모델을 훈련하는 방법이다.
  - 이항적 시퀀스의 예로 0.01배가 적용된 삼각 함수를 따르는 1000개의 합성 데이터를 사용한다.
  - 각 타임 스텝에 대한 라벨 y=xt와 피처 xt=[xt−τ,…,xt−1]을 사용하여 τ차 마르코프 조건을 충족하는 모델을 시도한다.
  - 데이터 처리:
    - 첫 τ 시퀀스에 대한 히스토리 부족으로 1000-τ 예제를 생성하고, τ 시퀀스를 0으로 채우지 않고 삭제하여 T-τ 예제를 생성한다.
    - 데이터 이터레이터를 통해 첫 600 예제에 대한 sin 함수의 기간을 포함하는 데이터를 생성한다.
  - 선형 회귀 모델:

```
model = d2l.LinearRegression(lr=0.01)
trainer = d2l.Trainer(max_epochs=5)
```

```
trainer.fit(model, data)
```

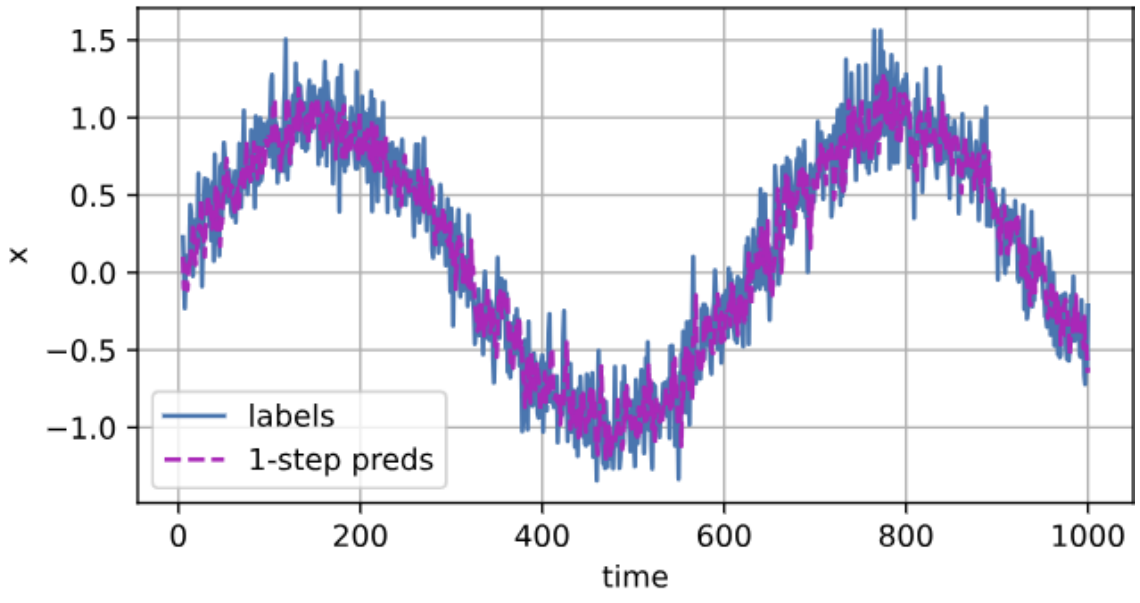
- d2l 라이브러리를 사용하여 선형 회귀 모델 및 트레이너를 초기화한다.



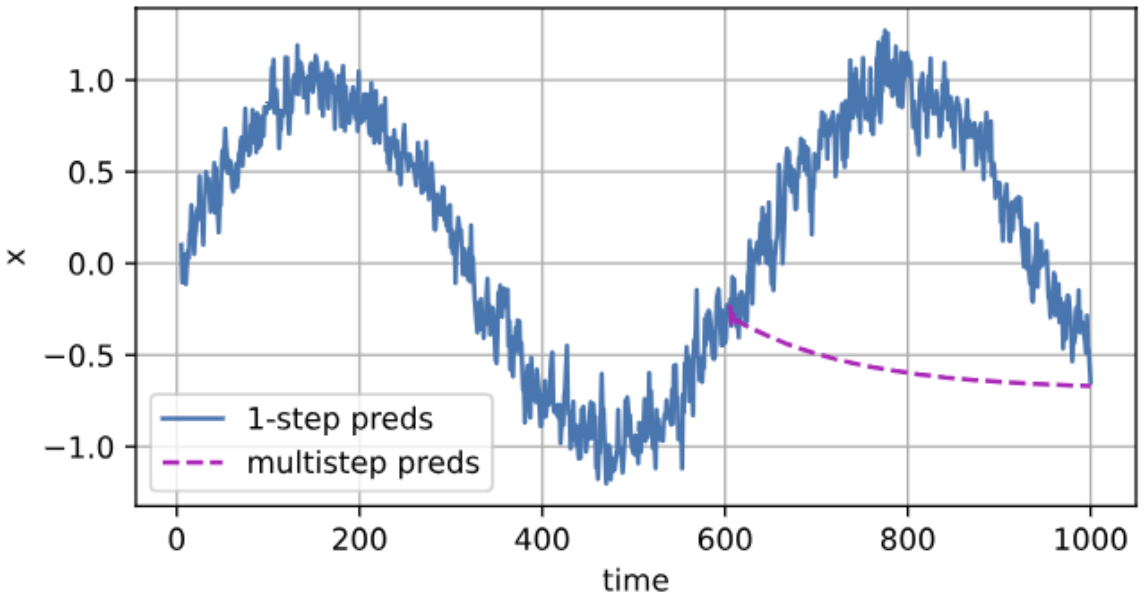
- 5 epoch동안 모델 학습 및 데이터 적합화를 거친다.

- 9.1.4 예측 평가:

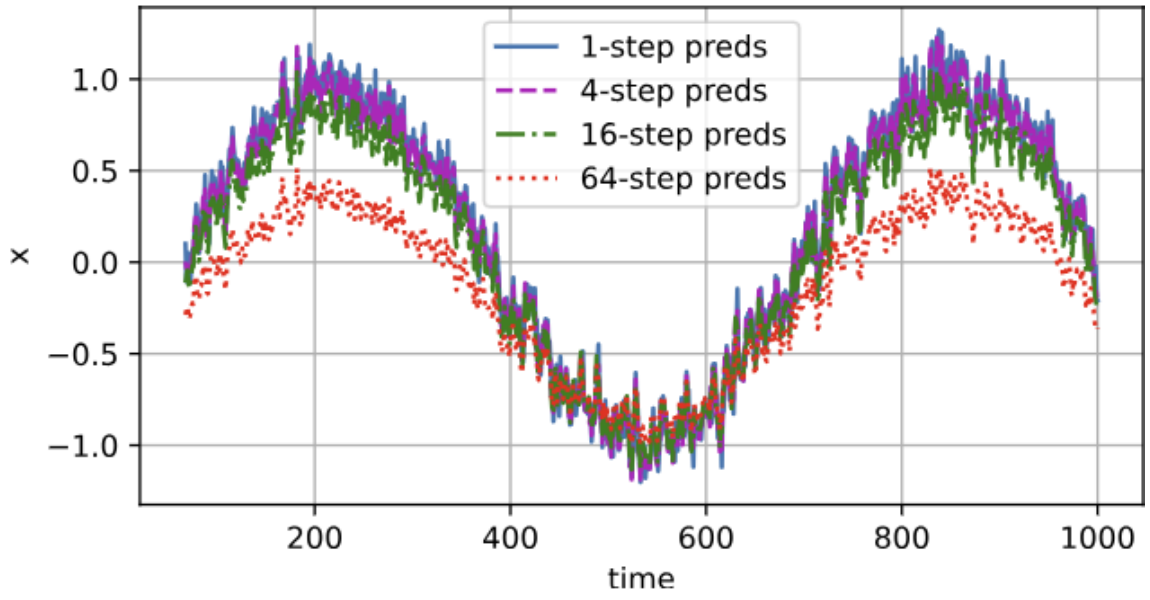
```
onestep_preds = model(data.features).detach().numpy()
d2l.plot(data.time[data.tau:], [data.labels, onestep_preds], 'time', 'x',
         legend=['labels', '1-step preds'], figsize=(6, 3))
```



- 1 스텝 전방 예측을 평가하여 모델 성능을 확인한다.



- 다중 스텝 예측 수행 및 결과를 시각화한 것이다.



- 1 스텝 예측 및 다중 스텝 예측 결과를 시각화한 것이다.
- 예측 한계:
  - 미래로 예측할수록 오차가 누적되는 문제가 있다.
  - 다양한 k에 대한 예측 결과를 시각화하여 미래 예측을 하는 데에 어려움이 있다.

- 9.1.5. 요약:

- 시퀀스 모델은 데이터의 시간적 순서를 중요시하며 특수 통계 도구가 필요하다.
- 관측된 시퀀스에서 t+k에 대한 예측을 k-step-ahead 예측이라고 한다.
- 미래로 예측할수록 오차가 누적되어 예측의 품질이 저하된다.

9.2 원시 텍스트를 시퀀스 데이터로 변환

- **전처리 파이프라인 단계**
  1. 텍스트를 문자열로 메모리에 로드한다.
  2. 문자열을 토큰(단어 또는 문자)으로 분할한다.
  3. 어휘 사전을 구축해 각 어휘 요소를 숫자 인덱스와 연결한다.
  4. 텍스트를 숫자 인덱스의 시퀀스로 변환한다.
- 9.2.1. **데이터셋 읽기**
  - 30,000 단어가 넘는 책인 H.G. Wells의 "The Time Machine"을 사용한다.

```
class TimeMachine(d2l.DataModule): #@save
    """The Time Machine dataset."""
    def _download(self):
        fname = d2l.download(d2l.DATA_URL + 'timemachine.txt', self.root,
                             '090b5e7e70c295757f55df93cb0a180b9691891a')
        with open(fname) as f:
            return f.read()

data = TimeMachine()
raw_text = data._download()
raw_text[:60]
```

  - 기본적인 전처리 파이프라인을 보여주기 위한 코드이다.

```
@d2l.add_to_class(TimeMachine)#@savedef _preprocess(self, text):
return re.sub('[^A-Za-z]+', ' ', text).lower()

text = data._preprocess(raw_text)
text[:60]
```

  - 구두점 및 대소문자를 무시하여 원시 텍스트를 전처리한다.
- 9.2.2. **토큰화**
  - 토큰은 텍스트의 원자(분할할 수 없는 단위) 단위이다.

```
@d2l.add_to_class(TimeMachine)#@savedef _tokenize(self, text):
return list(text)

tokens = data._tokenize(text)
','.join(tokens[:30])
```

- 문자 시퀀스로 텍스트를 토큰화한다.

- 9.2.3. **어휘 구축**

```
classVocab:#@save"""Vocabulary for text."""def __init__(self, tokens=[], min_freq=
0, reserved_tokens=[]):
# Flatten a 2D list if neededif tokensand isinstance(tokens[0], list):
            tokens = [tokenfor linein tokensfor tokenin line]
# Count token frequenciescounter = collections.Counter(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                  reverse=True)
# The list of unique tokensself.idx_to_token = list(sorted(set(['<unk>'] + reserve
d_tokens + [
            tokenfor token, freqin self.token_freqsif freq >= min_freq])))
        self.token_to_idx = {token: idx
for idx, tokenin enumerate(self.idx_to_token)}

def __len__(self):
return len(self.idx_to_token)

def __getitem__(self, tokens):
ifnot isinstance(tokens, (list, tuple)):
return self.token_to_idx.get(tokens, self.unk)
return [self.__getitem__(token)for tokenin tokens]

def to_tokens(self, indices):
if hasattr(indices, '__len__')and len(indices) > 1:
return [self.idx_to_token[int(index)]for indexin indices]
return self.idx_to_token[indices]

@propertydef unk(self):# Index for the unknown tokenreturn self.token_to_idx['<unk
>']
```

- 어휘 클래스 소개하는 코드를 구현한다.
- 'unk' 토큰을 사용하여 모르는 값을 표시한다.

```
vocab = Vocab(tokens)
indices = vocab[tokens[:10]]
print('indices:', indices)
print('words:', vocab.to_tokens(indices))
```

- 토큰을 고유 인덱스와 연결하는 어휘를 구축한다.

- 9.2.4. **전체적인 구성**
  - TimeMachine 클래스의 build 메서드로 모든 단계를 통합한다.

```
@d2l.add_to_class(TimeMachine)#@savedef build(self, raw_text, vocab=None):
    tokens = self._tokenize(self._preprocess(raw_text))
if vocabisNone: vocab = Vocab(tokens)
    corpus = [vocab[token]for tokenin tokens]
return corpus, vocab

corpus, vocab = data.build(raw_text)
len(corpus), len(vocab)
```

  - corpus(토큰 인덱스의 리스트) 및 어휘를 반환한다.

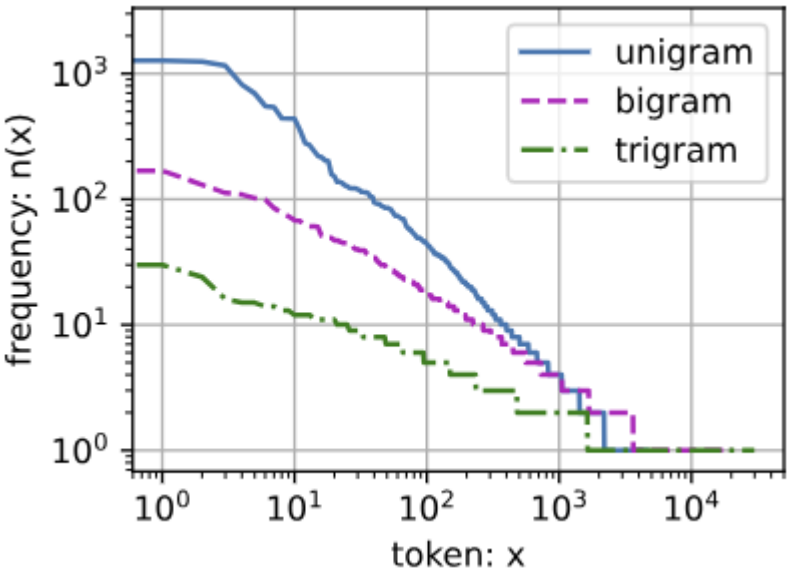
- 9.2.5. **탐험적 언어 통계**
  - 실제 코퍼스 및 단어를 기반으로 한 Vocab 클래스를 사용하여 기본 통계를 검토한다.

```
words = text.split()
vocab = Vocab(words)
vocab.token_freqs[:10]
```

  - 단어 빈도수 상위 10개를 출력하고, 단어 빈도수에 대한 로그-로그 플롯을 표시한다.
  - Zipf's Law for Words
    - i번째로 빈번한 단어의 빈도수 ni는 다음과 같다:

$$\log n_i = -\alpha \log i + c,$$

    - α는 분포를 특징 짓는 지수이고, c는 상수이다.
  - 단어를 통계적으로 모델링하려면 단어의 빈도를 과소 평가할 수 있음에 주의해야 한다.
  - 토큰 빈도 시각화



    - Unigram, Bigram, Trigram의 토큰 빈도를 시각화한다.
    - 시퀀스의 길이에 따라 알파(α) 값이 다르게 나타난다.
    - 다양한 n-gram이 존재하며, 많은 n-gram이 드물게 나타난다.

- 9.2.6. 분석 요약
    - 언어에서 텍스트는 Zipf의 법칙을 따른다.
    - 빈도가 높은 토큰들이 중요한 내용을 나타내지만, 불용어 등의 조합으로 희소한 토큰들이 증가한다.
    - 다양한 n-gram이 존재하지만, 드물게 나타난다.

9.3 언어 모델

통계 도구 및 9.1 섹션에서의 도구를 적용하는 과정을 통해 전체 시퀀스의 결합 확률을 추정한다.

언어 모델은 자연스러운 텍스트를 생성할 때 사용된다. 텍스트 이해가 필요한 완벽한 모델을 구현하기 어려우며, 문서 및 토큰 시퀀스 모델링에 대한 고민이 필요하다. 하지만 텍스트 유사성으로 인한 모호성을 처리할 수 있으며 문서 요약 알고리즘을 활용할 수 있다는 장점이 있다.

- 9.3.1. **언어 모델**
    - 문서 및 토큰 시퀀스 모델링은 단어 수준의 토큰화이며, 아래의 기본 확률 규칙을 적용한다.

$$P(x_1, x_2, \ldots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_1, \ldots, x_{t-1}).$$

- 9.3.1.1 **마르코프 모델 및 n-그램**
  - 마르코프 모델

$$P(x_{t+1} \mid x_t, \ldots, x_1) = P(x_{t+1} \mid x_t)$$

  - n-그램 모델의 근사화

$$\begin{aligned}
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2)P(x_3)P(x_4),\\
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_2)P(x_4 \mid x_3),\\
P(x_1, x_2, x_3, x_4) &= P(x_1)P(x_2 \mid x_1)P(x_3 \mid x_1, x_2)P(x_4 \mid x_2, x_3).
\end{aligned}$$

    - 1-그램, 2-그램, 3-그램 모델이다.
    - 언어 모델 계산에 필요한 확률 및 조건부 확률을 계산한다.
- 9.3.1.2. **단어 빈도**

$$\hat{P}(\text{learning} \mid \text{deep}) = \frac{n(\text{deep, learning})}{n(\text{deep})},$$

  - 대량의 텍스트 corpus를 통한 훈련 데이터로 가정하여 단어 확률을 계산한다.
  - Laplace Smoothing을 적용할 수 있다.
- 9.3.1.3. **Laplace Smoothing**
  - 특정 상수를 모든 카운트에 추가하여 확률 모델 매개변수 조절하고, 심층 학습 기반 언어 모델에 초점을 둔다.

- 9.3.2. **퍼플렉서티 (Perplexity)**
  - 텍스트 예측의 예상도 확인하여 **언어 모델 품질을 측정한다.**

    ex. "It is raining" 다음 예측
    - "It is raining outside"
    - "It is raining banana tree"
    - "It is raining piouw;kcj pwepoiut"

- 성능 측정: 엔트로피, 서프라이즐, 교차 엔트로피의 성능은 역수 형태의 퍼플렉서티 활용를 활용해 단일 토큰 예측 로그 확률의 평균을 구하여 측정한다.
- **Perplexity 계산**

$$\exp\left(-\frac{1}{n}\sum_{t=1}^{n}\log P(x_t \mid x_{t-1}, \ldots, x_1)\right).$$

  - 선택지 수의 기하평균의 역수이며, 성능 비교를 위한 평균화된 교차 엔트로피 손실로 측정할 수 있다.

- 9.3.3. **시퀀스 분할**
  - 신경망을 사용한 언어 모델이며, Perplexity를 사용하여 모델을 평가한다.
  - 미니배치에서 무작위로 입력 및 타깃 시퀀스를 추출한다.
  - 시퀀스 파티셔닝 전략

```
@d2l.add_to_class(d2l.TimeMachine)#@savedef __init__(self, batch_size, num_ste
ps, num_train=10000, num_val=5000):
    super(d2l.TimeMachine, self).__init__()
    self.save_hyperparameters()
    corpus, self.vocab = self.build(self._download())
    array = torch.tensor([corpus[i:i+num_steps+1]
for iin range(len(corpus)-num_steps)])
    self.X, self.Y = array[:,:-1], array[:,1:]
```

1. 입력 시퀀스 및 이전 토큰을 기반으로 `d2l.TimeMachine` 데이터를 생성한다.

```
@d2l.add_to_class(d2l.TimeMachine)#@savedef get_dataloader(self, train):
    idx = slice(0, self.num_train)if trainelse slice(
        self.num_train, self.num_train + self.num_val)
return self.get_tensorloader([self.X, self.Y], train, idx)
```

2. `data.train_dataloader()`를 통해 학습 데이터를 로드한다.

```
data = d2l.TimeMachine(batch_size=2, num_steps=10)
for X, Yin data.train_dataloader():
    print('X:', X, '\nY:', Y)
break
```

3. 두 번의 학습 데이터인 X와 Y를 출력한다.

- 9.3.4. 요약
  - 언어 모델은 텍스트 시퀀스의 결합 확률을 추정한다.
  - N-그램은 잘라냄으로서 의존성을 모델링하며 효과적으로 사용될 수 있다.
  - **Laplace Smoothing**를 통해 빈도가 낮은 단어 조합에 대응할 수 있다.
  - 신경 언어 모델링을 강조하며 무작위 샘플링과 퍼플렉서티를 사용한 훈련을 진행한다.
  - 데이터 크기, 모델 크기, 훈련 계산량 증가로 언어 모델 확장이 가능하다.
  - 대형 언어 모델은 입력 텍스트 지시에 대한 출력 텍스트를 예측하여 원하는 작업을 수행한다.
  - 대화를 모델링하는 방법, 긴 시퀀스 데이터 읽는 다른 방법 등 다양한 질문에 대한 논의가 이루어져야 한다.

9.4 순환 신경망

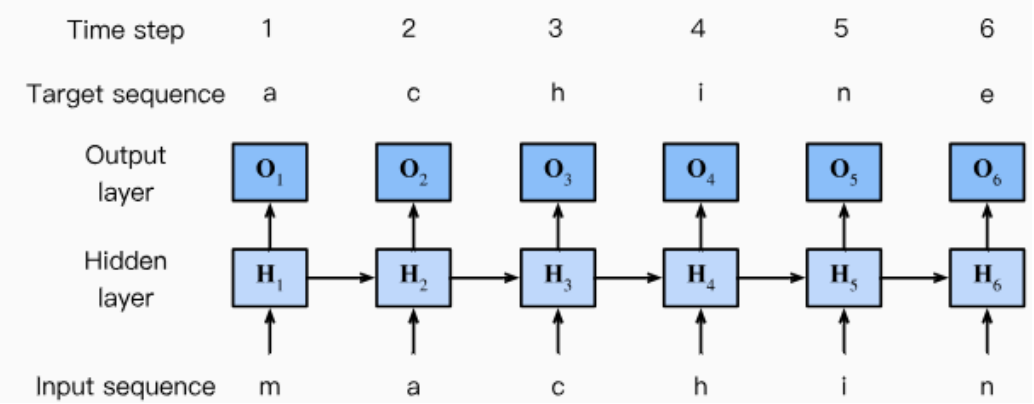
n-그램 모델에서는 조건부 확률이 t시점의 토큰 xt가 n-1 이전 토큰에만 의존했다. n을 증가시키면 모델 파라미터가 지수적으로 증가하므로 $P(x_t \mid x_{t-1},\ldots,x_{t-n+1})$ 모델링이 어렵다. 따라서 잠재 변수 모델 $P(x_t \mid x_{t-1},\ldots,x_1) \approx P(x_t \mid h_{t-1})$을 사용하는 것이 좋다.

- 9.4.1. 숨겨진 상태(ht-1)를 사용하는 잠재 변수 모델
  - h_{t-1}은 t-1 시점까지의 시퀀스 정보를 저장하는 숨겨진 상태이다.
  - 잠재 변수 모델에서, ht는 현재 입력 xt 및 이전 숨겨진 상태 h_{t-1}에 기반하여 계산된다.
- 9.4.2. 숨겨진 상태의 RNN

```
X, W_xh = torch.randn(3, 1), torch.randn(1, 4)
H, W_hh = torch.randn(3, 4), torch.randn(4, 4)
torch.matmul(X, W_xh) + torch.matmul(H, W_hh)
```

  - 현재 입력(xt) 및 이전 숨겨진 상태(ht-1)에 기반한 숨겨진 상태를 행렬을 활용해 계산한다.
  - 순환 신경망(RNN)은 숨겨진 상태를 가지는 신경망이다.
- 9.4.3. RNN 기반 문자 수준 언어 모델

- RNN을 사용하여 문자 수준 언어 모델을 구축할 수 있으며, 학습 과정에서 소프트맥스 및 크로스-엔트로피 손실이 사용된다.
  - 언어 모델에서는 현재 및 과거 토큰에 기반하여 다음 토큰을 예측한다.
  - 아래 그림은 RNN을 사용한 문자 수준 언어 모델이다.



  - 훈련 과정에서 시간 단계별로 출력 레이어의 출력에 대해 소프트맥스 연산을 실행한 다음 교차 엔트로피 손실을 사용하여 모델 출력과 목표 사이의 오차를 계산한다.
  - RNN의 히든 레이어에서 재귀적인 계산으로 인해 모델 파라미터 수가 시간 단계가 증가해도 선형적으로 증가하지 않는다.

- 9.4.4. 요약
  - 순환 신경망(RNN)은 숨겨진 상태에 재귀 계산을 사용하는 신경망이다.
  - RNN의 숨겨진 상태는 현재 시간 단계까지의 시퀀스의 과거 정보를 포착하는 것이고, RNN 모델 파라미터 수는 시간 단계 수가 증가해도 증가하지 않는다.

9.5 RNN 구현

- 9.5.1. RNN 모델 구현

```
classRNNScratch(d2l.Module):#@save"""The RNN model implemented from scratch."""def
__init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W_xh = nn.Parameter(
            torch.randn(num_inputs, num_hiddens) * sigma)
        self.W_hh = nn.Parameter(
            torch.randn(num_hiddens, num_hiddens) * sigma)
        self.b_h = nn.Parameter(torch.zeros(num_hiddens))
```

- **클래스 정의**: RNNScratch 클래스로 RNN 모델을 구현한다.
- **하이퍼파라미터**: `num_inputs`, `num_hiddens` 와 표준편차인 `sigma` 값을 설정한다.

```
@d2l.add_to_class(RNNScratch)#@savedef forward(self, inputs, state=None):
if stateisNone:
# Initial state with shape: (batch_size, num_hiddens)state = torch.zeros((inputs.s
hape[1], self.num_hiddens),
                          device=inputs.device)
else:
        state, = state
    outputs = []
for Xin inputs:# Shape of inputs: (num_steps, batch_size, num_inputs)state = torc
h.tanh(torch.matmul(X, self.W_xh) +
                         torch.matmul(state, self.W_hh) + self.b_h)
        outputs.append(state)
return outputs, state
```

- **Forward 메서드**: Tanh 활성화 함수 사용하여 현재 입력 및 이전 타임 스텝의 모델 상태를 사용하여 출력 및 숨겨진 상태를 계산한다.

```
batch_size, num_inputs, num_hiddens, num_steps = 2, 16, 32, 100
rnn = RNNScratch(num_inputs, num_hiddens)
X = torch.ones((num_steps, batch_size, num_inputs))
outputs, state = rnn(X)
```

- **모델 테스트**: 미니배치 입력 시퀀스를 RNN 모델에 공급하여 결과를 확인한다.

```
def check_len(a, n):#@save"""Check the length of a list."""assert len(a) == n, f'l
ist\'s length{len(a)} != expected length{n}'

def check_shape(a, shape):#@save"""Check the shape of a tensor."""assert a.shape =
= shape, \
            f'tensor\'s shape{a.shape} != expected shape{shape}'

check_len(outputs, num_steps)
check_shape(outputs[0], (batch_size, num_hiddens))
check_shape(state, (batch_size, num_hiddens))
```

- **결과 확인 함수 정의**: RNN 모델이 올바른 형태의 결과를 생성하는지 확인한다.

- 9.5.2. RNN 기반 언어 모델

```
classRNNLMScratch(d2l.Classifier):#@save"""The RNN-based language model implemente
d from scratch."""def __init__(self, rnn, vocab_size, lr=0.01):
        super().__init__()
```

```
        self.save_hyperparameters()
        self.init_params()

def init_params(self):
        self.W_hq = nn.Parameter(
            torch.randn(
                self.rnn.num_hiddens, self.vocab_size) * self.rnn.sigma)
        self.b_q = nn.Parameter(torch.zeros(self.vocab_size))

def training_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('ppl', torch.exp(l), train=True)
return l

def validation_step(self, batch):
        l = self.loss(self(*batch[:-1]), batch[-1])
        self.plot('ppl', torch.exp(l), train=False)
```

  - **클래스 정의**: RNNScratch 클래스로 RNN 모델을 구현한다.
  - **하이퍼파라미터**: `rnn`, `vocab_size`, `lr`
  - **Init 및 파라미터 초기화 메서드 정의**
  - **Training 및 Validation Step 정의**

- 9.5.2.1. One-Hot 인코딩
  - 입력 시퀀스를 one-hot 인코딩으로 변환한다.
- 9.5.2.2. RNN 출력 변환
  - RNN 출력을 토큰 예측으로 변환하는 fully connected output 레이어를 활용해 RNN 출력을 변환한다.

```
@d2l.add_to_class(RNNLMScratch)#@savedef output_layer(self, rnn_outputs):
    outputs = [torch.matmul(H, self.W_hq) + self.b_qfor Hin rnn_outputs]
return torch.stack(outputs, 1)

@d2l.add_to_class(RNNLMScratch)#@savedef forward(self, X, state=None):
    embs = self.one_hot(X)
    rnn_outputs, _ = self.rnn(embs, state)
return self.output_layer(rnn_outputs)
```

  - **Forward 메서드 업데이트**: RNN 출력 변환을 적용한다.
    - 코드는 RNN 기반 언어 모델을 구현하고 훈련하는 것으로 구성되어 있다.
    - Forward 연산, Gradient Clipping, Training, Decoding 등이 구현되어 있다.

        - 언어 모델을 통해 주어진 텍스트 접두사를 기반으로 다음 토큰을 예측할 수 있다.

```
model = RNNLMScratch(rnn, num_inputs)
outputs = model(torch.ones((batch_size, num_steps), dtype=torch.int64))
check_shape(outputs, (batch_size, num_steps, num_inputs))
```

    - RNNLMScratch 클래스:
        - `forward` 메서드: 주어진 입력에 대한 모델의 출력을 생성한다.
        - `output_layer` 메서드: RNN 출력에 대한 최종 출력을 생성한다.

- 9.5.3. Gradient Clipping:
    - 기울기 폭발 및 소실 문제를 해결하기 위해 기울기 클리핑이 구현된다.

```
@d2l.add_to_class(d2l.Trainer)#@savedef clip_gradients(self, grad_clip_val, mode
l):
    params = [pfor pin model.parameters()if p.requires_grad]
    norm = torch.sqrt(sum(torch.sum((p.grad ** 2))for pin params))
if norm > grad_clip_val:
for paramin params:
            param.grad[:] *= grad_clip_val / norm
```

    - `clip_gradients` 메서드는 모델의 기울기를 일정한 값으로 제한한다.

- 9.5.4. Training:
    - Time Machine 데이터셋을 사용하여 RNN 기반 언어 모델을 훈련한다.
    - 훈련 중에는 기울기 클리핑이 사용되며, 최대 에폭 및 기타 하이퍼파라미터가 설정된다.

- 9.5.5. Decoding:

```
@d2l.add_to_class(RNNLMScratch)#@savedef predict(self, prefix, num_preds, vocab, d
evice=None):
    state, outputs =None, [vocab[prefix[0]]]
for iin range(len(prefix) + num_preds - 1):
        X = torch.tensor([[outputs[-1]]], device=device)
        embs = self.one_hot(X)
        rnn_outputs, state = self.rnn(embs, state)
if i < len(prefix) - 1:# Warm-up periodoutputs.append(vocab[prefix[i + 1]])
else:# Predict num_preds stepsY = self.output_layer(rnn_outputs)
            outputs.append(int(Y.argmax(axis=2).reshape(1)))
return ''.join([vocab.idx_to_token[i]for iin outputs])
```

- `predict` 메서드: 주어진 접두사에 대한 언어 모델의 예측을 생성한다.

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

  - 예측된 텍스트는 주어진 접두사 다음에 나오는 가능성 있는 문자이다.

- 9.5.6. 요약
  - 코드는 RNN을 사용하여 언어 모델을 훈련하고 텍스트 생성을 수행하는 방법을 보여준다.
  - Gradient Clipping을 사용하여 기울기 문제를 해결하고, 접두사를 기반으로 다음 토큰을 예측하는 방법을 소개한다.

9.6 RNN의 간결한 구현

고급 API를 사용해 구현 시간과 계산 시간을 줄이는 효율적인 방식을 보여준다.

- 9.6.1 **모델 정의**

```
classRNN(d2l.Module):#@save"""The RNN model implemented with high-level APIs."""de
f __init__(self, num_inputs, num_hiddens):
        super().__init__()
        self.save_hyperparameters()
        self.rnn = nn.RNN(num_inputs, num_hiddens)

def forward(self, inputs, H=None):
return self.rnn(inputs, H)
```

  - 상위 API로 구현된 RNN을 활용한다.

```
classRNNLM(d2l.RNNLMScratch):#@save"""The RNN-based language model implemented wit
h high-level APIs."""def init_params(self):
        self.linear = nn.LazyLinear(self.vocab_size)

def output_layer(self, hiddens):
return self.linear(hiddens).swapaxes(0, 1)
```

  - RNNLM 클래스를 정의한다.

- 9.6.2 **Training and Predicting**

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
rnn = RNN(num_inputs=len(data.vocab), num_hiddens=32)
```

```
model = RNNLM(rnn, vocab_size=len(data.vocab), lr=1)
model.predict('it has', 20, data.vocab)
```

- 모델 예측: 랜덤 가중치로 초기화된 모델을 사용하여 예측한다.

```
trainer = d2l.Trainer(max_epochs=100, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```

- 모델 학습: 고수준 API를 활용하여 모델을 훈련시킨다.

```
model.predict('it has', 20, data.vocab, d2l.try_gpu())
```

- 모델 평가: 지정된 접두어 문자열을 사용하여 예측된 토큰을 생성한다.
- 성능 비교: 9.5와 비교하여 비슷한 Perplexity이며, 최적화된 구현으로 인한 빠른 실행이 가능하다.

- 9.6.3. **요약**
  - 고수준 API는 표준 RNN의 구현을 제공하며, 재구현 시간을 절약하고 최적화된 성능을 제공한다.

9.7 역전파

9.5절 연습을 통해 그라디언트 클리핑이 훈련의 불안정성을 방지하는 데 중요하다는 것을 알았다. 폭발적인 그라디언트는 긴 시퀀스를 통한 역전파에서 비롯되는 것이라는 것을 유추해볼 수 있다.

RNN에서 역전파를 적용하는 것은 시간을 통한 역전파이다. unrolled RNN은 특별한 속성을 가진 피드포워드 신경망이며, 동일한 매개변수가 모든 시간 단계에서 반복되어 나타난다. 따라서 연쇄 법칙을 적용하여 펼친 신경망을 통해 그라디언트를 역전파할 수 있다. 시퀀스가 길면 문제가 발생할 수 있으며, 이는 계산 및 최적화 측면에서 문제를 일으킬 수 있다.

- **9.7.1. RNN에서의 그라디언트 분석**

  숨겨진 상태(ht), 입력(xt), 및 출력(ot)으로 구성된 간단한 RNN 모델을 가정하며, 입력과 숨겨진 상태는 은닉 계층의 하나의 가중 변수와 함께 곱해지기 전에 연결(concatenated)될 수 있다. 역전파를 적용하는 것은 시간별로 RNN의 계산 그래프를 하나씩 펼치는 것이라고 한다. 연쇄 법칙을 적용하여 그라디언트를 펼친 네트워크를 통해 역전파할 수 있으며, 각 매개변수에 대한 그라디언트는 펼친 네트워크 전체에서 발생한 모든 위치에서 합산되어야 한다. 순차적인 계산으로 인해 시퀀스가 길어질수록 계산적(너무 많은 메모리) 및 최적화적(수치적 불안정성)으로 문제가 발생할 수 있다.

- **9.7.1.1. 그라디언트 계산 전략 비교**
    1. **전체 계산**
        - 전체 합을 계산할 수 있지만 매우 느리며 그라디언트가 폭발할 수 있다.
        - 일반적으로 실무에서 거의 사용되지 않는다.
    2. **시간 단계 자르기**
        - 특정 시간 단계 후에 합을 자를 수 있으며, 이것이 일반적으로 사용되는 줄임말 역전파이다.
        - 따라서 모델은 주로 단기적인 영향에 중점을 두며 장기적인 결과에는 중점을 덜 둔다.
    3. **랜덤 시간 단계 자르기**
        - 임의로 그라디언트를 자르는 것이 가능하며, 기대치에서 올바른 확률 변수로 대체된다.
        - 랜덤 트랜케이션을 통해 일부 시퀀스를 제거하여 가중 평균을 얻을 수 있다.
    4. **전략 비교**
        - 그라디언트 계산 전략을 시각화하는 방법이다.
        - 랜덤 트랜케이션이 일반적으로 더 나은 결과를 내지 않기에 시퀀스의 일부를 랜덤하게 자르는 것이 일반적으로 실제 의존성을 잡는 데 충분하다는 것을 제안한다.
- 9.7.2. 세부적인 시간을 통한 역전파
    - Backpropagation through time은 숨겨진 상태를 가진 순차 모델에 역전파를 적용한 것이다.
    - 연산의 편리함과 수치 안정성을 위해 정규 또는 무작위로 자른 Truncation이 필요하다.
    - 행렬의 고차원은 발산 또는 소멸하는 고유값으로 이어질 수 있으며, 이는 폭주 또는 소멸 그라디언트로 나타난다.
- 9.7.3. 요약
    - 효율적인 계산을 위해 역전파동안 중간 값들이 캐싱된다.