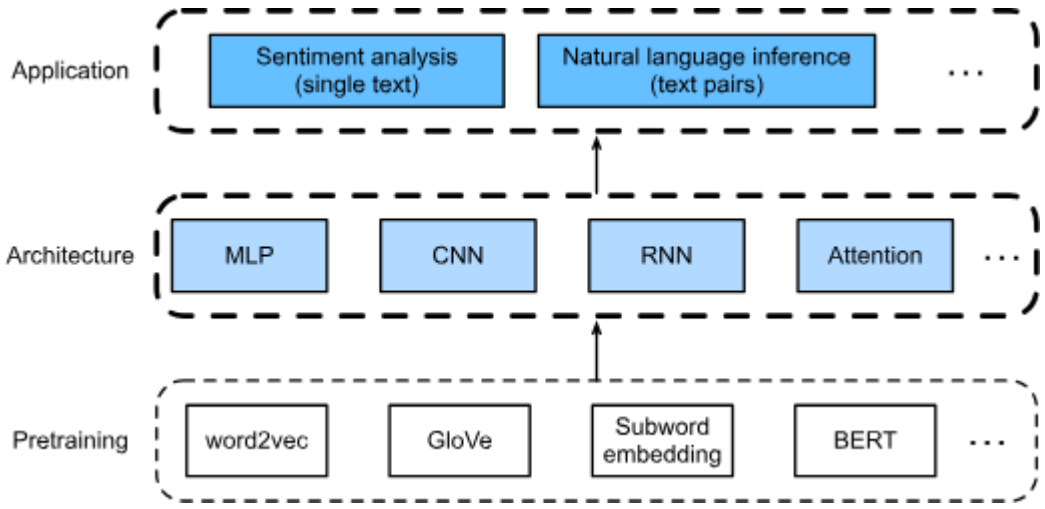
Dive into DeepLearning Ch.16

상태	Done
작성자	Ⓚ KHM
최종 편집 일시	@2023년 12월 9일 오후 9:31
태그	

16. Natural Language Processing: Applications

자연어 처리 문제를 해결하기 위해 언어의 심층 표현 학습을 적용하는 방법

*모든 응용 프로그램을 포괄적으로 다루는 것은 아니지만, 이 장에서는 자언어 처리 응용을 위한 모델을 설계하는 방법에 중점을 둠.



위의 사진처럼 MLP, CNN, RNN, Attention 등과 같은 다른 유형의 딥러닝 아키텍처를 사용하여 자연어 처리 모델을 디자인하는 기본 아이디어에 중점을 둠.

특히, 감정 분석(sentiment analysis)을 위해 인기 있는 아키텍처인 RNN, CNN을 살펴볼 것임.

자연어 추론의 경우 어텐션과 MLP로 텍스트 쌍을 분석하는 방법을 살펴봄.

마지막으로, sequence level(단일 텍스트 분류 및 텍스트 쌍 분류) 및 token level(텍스트 태깅 및 질문 답변)과 같은 광범위한 자연어 처리 응용에 대해 사전 훈련된 BERT 모델을 미세 조정하는 방법을 소개.

16.1. Sentiment Analysis and the Dataset

감정 분석 및 데이터셋

온라인 소셜 미디어와 리뷰 플랫폼이 확산되면서, 수많은 의견이 있는 데이터가 기록되어 의사 결정 프로세스를 지원하는 큰 잠재력을 갖게 되었다.

> 💡 Sentiment Analysis는 제품 리뷰, 블로그 댓글, 포럼 토론 등에서 생성된 텍스트에서 사람들의 감정을 연구한다.

→ 이는 정치, 금융, 마케팅 등 다양한 분야에 폭넓게 적용될 수 있다.

감정은 추상적인 양극 또는 척도로(긍정, 부정) 분류될 수 있기 때문에, 감정 분석을 **다양한 길이의 텍스트 시퀀스를 고정 길이 텍스트 범주로 변환하는 텍스트 분류 작업**으로 간주할 수 있다.

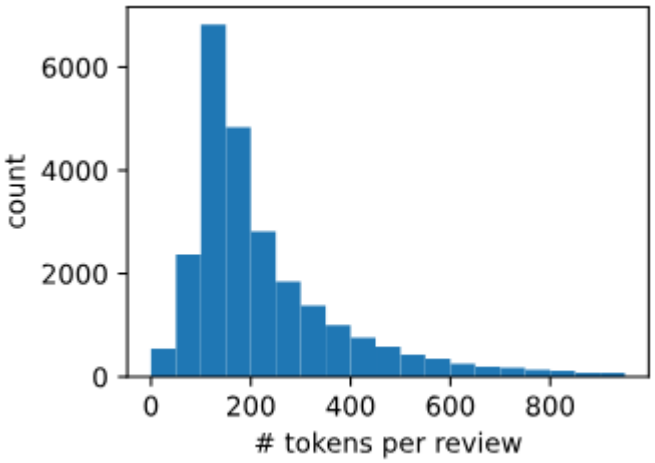
이 장의 실습은 Stanford의 대규모 영화 리뷰 데이터 셋을 사용하였다.

여기에는 25000개의 영화 리뷰를 포함한 training set와 test set로 구성되어 있다.

두 데이터 셋 모두 동일한 수의 긍정 레이블과 부정 레이블이 있다.

```
import os
import torch
from torch import nn
from d2l import torch as d2l

#IMDb 리뷰 데이터셋 읽기
d2l.DATA_HUB['aclImdb'] = (d2l.DATA_URL + 'aclImdb_v1.tar.gz'
```

```
                              '01ada507287d82875905620988597833ad

data_dir = d2l.download_extract('aclImdb', 'aclImdb')

#훈련, 테스트 데이터셋 읽기
#각 예는 리뷰이며 긍정인 경우 1, 부정의 경우 0의 라벨을 갖는다.
def read_imdb(data_dir, is_train):
    """Read the IMDb review dataset text sequences and labels
    data, labels = [], []
    for label in ('pos', 'neg'):
        folder_name = os.path.join(data_dir, 'train' if is_tr
                                   label)
        for file in os.listdir(folder_name):
            with open(os.path.join(folder_name, file), 'rb') 
                review = f.read().decode('utf-8').replace('\n
                data.append(review)
                labels.append(1 if label == 'pos' else 0)
    return data, labels

train_data = read_imdb(data_dir, is_train=True)
print('# trainings:', len(train_data[0]))
for x, y in zip(train_data[0][:3], train_data[1][:3]):
    print('label:', y, 'review:', x[:60])


#데이터셋 전처리
#각 단어를 토큰으로 처리
#5번 미만으로 나타나는 단어를 필터링하여 훈련 데이터셋에서 어휘를 생성
train_tokens = d2l.tokenize(train_data[0], token='word')
vocab = d2l.Vocab(train_tokens, min_freq=5, reserved_tokens=[

#리뷰 길이의 히스토그램을 토큰 단위로
d2l.set_figsize()
d2l.plt.xlabel('# tokens per review')
d2l.plt.ylabel('count')
d2l.plt.hist([len(line) for line in train_tokens], bins=range
```



위처럼 리뷰의 길이가 매우 다양함.

```
#minibatch를 처리하기 위하여 각 리뷰의 길이를 500으로 설정
num_steps = 500  # sequence length
train_features = torch.tensor([d2l.truncate_pad(
    vocab[line], num_steps, vocab['<pad>']) for line in train_
print(train_features.shape)

#데이터 반복자 생성(각 반복마다 예제의 미니배치가 반환됨)
train_iter = d2l.load_array((train_features, torch.tensor(tra

for X, y in train_iter:
    print('X:', X.shape, ', y:', y.shape)
    break
print('# batches:', len(train_iter))
```

💡 X: torch.Size([64, 500]) , y: torch.Size([64])
batches: 391

```
#Last, we wrap up the above steps into the load_data_imdb fun
#훈련 및 테스트 데이터 반복자와 IMDb 검토 데이터 세트의 어휘를 반환
def load_data_imdb(batch_size, num_steps=500):
    """Return data iterators and the vocabulary of the IMDb r
    data_dir = d2l.download_extract('aclImdb', 'aclImdb')
    train_data = read_imdb(data_dir, True)
```

```
    test_data = read_imdb(data_dir, False)
    train_tokens = d2l.tokenize(train_data[0], token='word')
    test_tokens = d2l.tokenize(test_data[0], token='word')
    vocab = d2l.Vocab(train_tokens, min_freq=5)
    train_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in t
    test_features = torch.tensor([d2l.truncate_pad(
        vocab[line], num_steps, vocab['<pad>']) for line in t
    train_iter = d2l.load_array((train_features, torch.tensor
                                 batch_size)
    test_iter = d2l.load_array((test_features, torch.tensor(t
                               batch_size,
                               is_train=False)
    return train_iter, test_iter, vocab
```

요약

- 감정 분석은 생산된 텍스트에서 사람들의 감정을 연구하는데, 이는 다양한 길이의 텍스트 시퀀스를 고정 길이의 텍스트 범주로 변환하는 텍스트 분류 문제로 간주됨.
- 전처리 후에 Standford 대규모 영화 리뷰 데이터셋을 어휘가 있는 데이터 반복기에 로드할 수 있음.

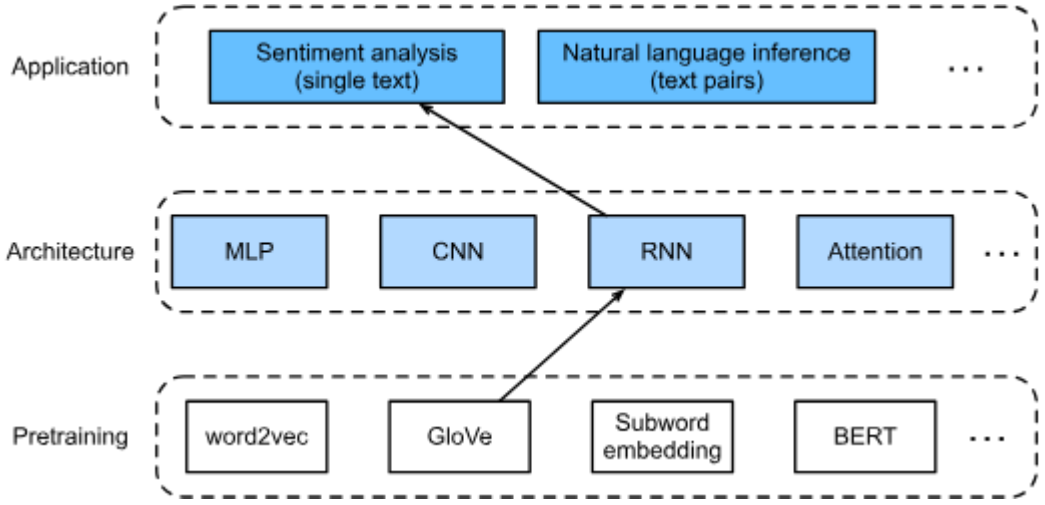
16.2. Sentiment Analysis: Using Recurrent Neural Networks

감정 분석: RNN 사용

RNN이란?

RNN은 시계열 또는 순차 데이터를 예측하는 딥러닝을 위한 신경망 아키텍처.
내부의 순환 구조가 포함되어 있기 때문에 시간에 의존적이거나 순차적인 데이터 학습에 활용됨.
내부에 있는 순환 구조에 의해 현재 정보에 이전 정보가 쌓이면서 정보 표현이 가능한 알고리즘으로, 데이터가 순환되기 때문에 정보가 끊임없이 갱신될 수 있는 구조

단어 유사성 및 유추 작업과 마찬가지로, 사전 훈련된 단어 벡터를 감정 분석에 적용할 수 있음.



여기서는 감정 분석을 위해 사전 훈련된 GloVe를 RNN 기반 아키텍처에 공급

→ GloVe 모델을 사용하여 각 토큰을 표현하고 이러한 토큰 표현을 다층 양방향 RNN에 공급하여 감정 분석 출력으로 변환될 텍스트 시퀀스 표현을 얻는다.

```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

감정 분석과 같은 텍스트 분류 작업에서는 다양한 길이의 텍스트 시퀀스가 고정 길이 범주로 변환됨.

여기서는 텍스트 시퀀스의 각 토큰이 임베딩 레이어(`self.embedding`)를 통해 사전 훈련된 개별 토큰을 가져오고, 전체 시퀀스는 양방향 RNN(`self.encoder`)로 인코딩됨.

- 양방향 LSTM은 입력 시퀀스를 앞뒤 양쪽에서 동시에 처리하여 각 시점에서의 문맥 정보를 효과적으로 활용할 수 있음.
- 초기와 최종 시간 단계에서는 양방향 LSTM의 마지막 레이어의 숨겨진 상태가 텍스트 시퀀스의 표현으로 연결됨.
- 초기 및 최종 시간 단계 모두에서 양방향 LSTM의 숨겨진 상태(마지막 레이어)는 텍스트 시퀀스의 표현으로 연결
- `self.decoder`이 단일 텍스트 표현은 두 개의 출력("양수" 및 "음수")이 있는 완전히 연결된 레이어에 의해 출력 범주로 변환

```
class BiRNN(nn.Module):
    def __init__(self, vocab_size, embed_size, num_hiddens,
                 num_layers, **kwargs):
        super(BiRNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # Set `bidirectional` to True to get a bidirectional
        self.encoder = nn.LSTM(embed_size, num_hiddens, num_l
                               bidirectional=True)
        self.decoder = nn.Linear(4 * num_hiddens, 2)

    def forward(self, inputs):
        # The shape of `inputs` is (batch size, no. of time s
        # LSTM requires its input's first dimension to be the
        # dimension, the input is transposed before obtaining
        # representations. The output shape is (no. of time s
        # word vector dimension)
        embeddings = self.embedding(inputs.T)
        self.encoder.flatten_parameters()
        # Returns hidden states of the last hidden layer at d
        # steps. The shape of `outputs` is (no. of time steps
        # 2 * no. of hidden units)
        outputs, _ = self.encoder(embeddings)
        # Concatenate the hidden states at the initial and fi
```

```
        # the input of the fully connected layer. Its shape i
        # 4 * no. of hidden units)
        encoding = torch.cat((outputs[0], outputs[-1]), dim=1
        outs = self.decoder(encoding)
        return outs
```

감정 분석을 위한 단일 텍스트를 나타내기 위해 두 개의 숨겨진 레이어가 있는 양방향 RNN을 구성

```
embed_size, num_hiddens, num_layers, devices = 100, 100, 2, d
net = BiRNN(len(vocab), embed_size, num_hiddens, num_layers)

def init_weights(module):
    if type(module) == nn.Linear:
        nn.init.xavier_uniform_(module.weight)
    if type(module) == nn.LSTM:
        for param in module._flat_weights_names:
            if "weight" in param:
                nn.init.xavier_uniform_(module._parameters[pa
net.apply(init_weights);
```

사전 훈련된 단어 벡터 로드

- 사전 훈련된 100차원 GloVe 임베딩을 로드
- 단어의 모든 토큰에 대한 vector 형태 출력

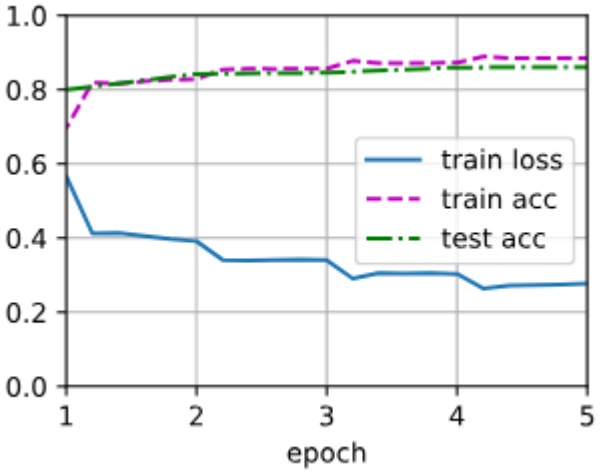
```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')

embeds = glove_embedding[vocab.idx_to_token]
embeds.shape
```

torch.Size([49346, 100])

모델 훈련

```
lr, num_epochs = 0.01, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num
```

💡 loss 0.277, train acc 0.884, test acc 0.861
2608.4 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]



손실 (0.277): 훈련 중에 최소화하려는 값

훈련 정확도 (0.884): 이는 모델이 훈련 데이터에서 얼마나 정확한 예측을 하는지

테스트 정확도 (0.861): 모델이 훈련 중에 본 적이 없는 별도의 테스트 데이터에서의 정확도. 즉, 모델이 새로운, 보지 못한 데이터에 얼마나 잘 일반화되는지

초당 처리 속도 (2608.4): 이는 훈련 속도를 나타내는 지표

epoch: 전체 훈련 데이터셋이 모델에 한 번 전달되는 주기

+

훈련된 모델을 사용하여 텍스트 시퀀스의 감정 예측을 위한 함수 정의

```
#@save
def predict_sentiment(net, vocab, sequence):
    """Predict the sentiment of a text sequence."""
```

```
    sequence = torch.tensor(vocab[sequence.split()], device=d
    label = torch.argmax(net(sequence.reshape(1, -1)), dim=1)
    return 'positive' if label == 1 else 'negative'
```

감정 예측 시도 1

```
predict_sentiment(net, vocab, 'this movie is so great')
```

💡 'positive'

```
predict_sentiment(net, vocab, 'this movie is so bad')
```

💡 'negative'

요약

- 사전 훈련된 단어 벡터는 텍스트 시퀀스의 개별 토큰을 나타낼 수 있음
- 양방향 RNN은 초기 및 최종 시간 단계에서 hidden states를 연결하는 등의 방법으로 텍스트 시퀀스를 나타낼 수 있음
    - 이 단일 텍스트 표현은 완전히 연결된 레이어를 사용하여 카테고리로 변환될 수 있음

16.3. Sentiment Analysis: Using Convolutional Neural Networks

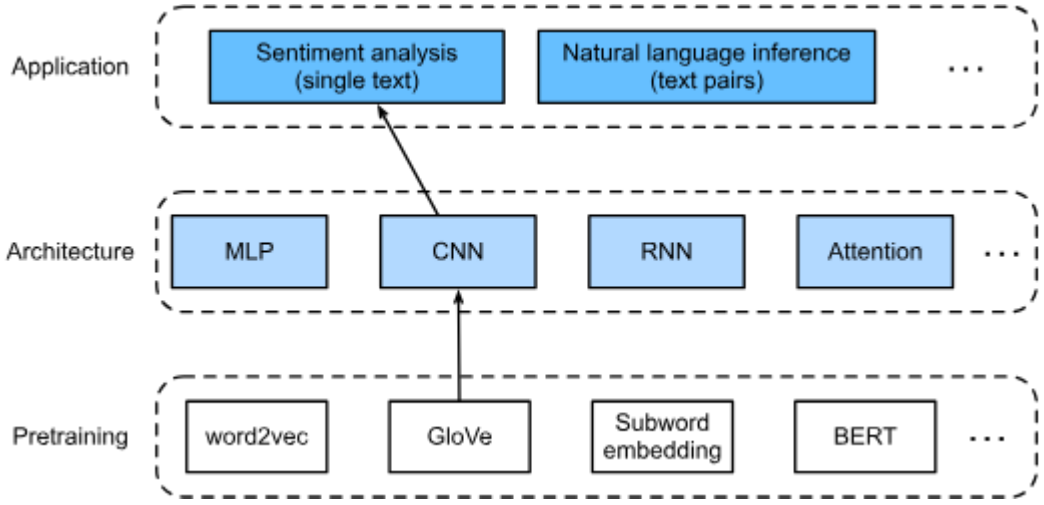
감정 분석: CNN 사용

💡 **CNN이란?**

데이터로부터 직접 학습하는 딥러닝의 신경망 아키텍처
주로 이미지나 영상 데이터를 처리할 때 쓰임
Convolution이라는 전처리 작업이 들어가는 Neural Network 모델

CNN은 원래 컴퓨터 비전용으로 설계되었지만 자연어 처리에도 널리 사용됨.

→ 텍스트 시퀀스를 1차원 이미지로 생각

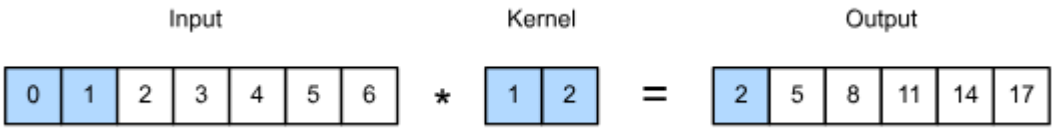



```
import torch
from torch import nn
from d2l import torch as d2l

batch_size = 64
train_iter, test_iter, vocab = d2l.load_data_imdb(batch_size)
```

1차원 Convolution

원본 데이터를 2차원으로 놓고 필터라는 사각형 윈도우를 씌운 후에 움직이며 새로운 값을 만들어 냄.

이 때 움직이는 과정을 convolution이라고 하고, 필터가 움직이는 방향이 한 방향이면 1D-CNN, 두 방향이면 2D-CNN이라고 함.



16.3.3

→ 음영 처리된 부분: 첫 번째 출력 요소이자 출력 계산에 사용되는 입력 및 커널 텐서 요소.

$0 \times 1 + 1 \times 2 = 2$

1차원 컨볼루션의 경우 컨볼루션 창이 입력 텐서를 가로질러 왼쪽에서 오른쪽으로 미끄러진다.

슬라이딩하는 동안, 특정 위치에서 합성곱 창에 포함된 입력 서브 텐서(위 예시의 0, 1)와 커널 텐서(위 예시의 1, 2)는 요소별로 곱해진다.

이러한 곱셈의 합은 출력 텐서의 해당 위치에 단일 스칼라 값을 제공한다.

아래의 corr1d 함수에서 일차원 교차 상관을 구현하자. 입력 텐서 X와 커널 텐서 K가 주어지면 출력 텐서 Y를 반환한다.

```
def corr1d(X, K):
    w = K.shape[0]
    Y = torch.zeros((X.shape[0] - w + 1))
    for i in range(Y.shape[0]):
        Y[i] = (X[i: i + w] * K).sum()
    return Y
```

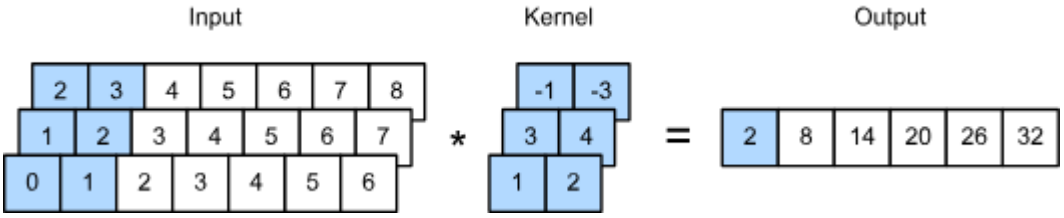
입력 텐서 X와 커널 텐서 K를 만들어 위의 함수의 출력을 검증해보자.

```
X, K = torch.tensor([0, 1, 2, 3, 4, 5, 6]), torch.tensor([1,
corr1d(X, K)
```

💡 tensor([2., 5., 8., 11., 14., 17.])

다중 채널을 가진 일차원 입력에 대해 컨볼루션 커널은 동일한 입력 채널 수를 가져야 한다.

그 후 각 채널에 대해 입력의 일차원 텐서와 컨볼루션 커널의 일차원 텐서 간의 교차 상관 연산을 수행하고 결과를 모든 채널에 대해 합하여 일차원 출력 텐서를 생성한다.



16.3.4

위의 그림은 3개의 입력 채널을 가진 일차원 교차 상관 작업을 보여준다.

다중 입력 채널에 대한 일차원 교차 상관 작업을 구현해보자.

```
def corr1d_multi_in(X, K):
    # First, iterate through the 0th dimension (channel dimen
    # `K`. Then, add them together
    return sum(corr1d(x, k) for x, k in zip(X, K))

X = torch.tensor([[0, 1, 2, 3, 4, 5, 6],
                  [1, 2, 3, 4, 5, 6, 7],
                  [2, 3, 4, 5, 6, 7, 8]])
K = torch.tensor([[1, 2], [3, 4], [-1, -3]])
corr1d_multi_in(X, K)
```

💡 tensor([2., 8., 14., 20., 26., 32.])

다중 입력 채널을 가진 일차원 교차 상관은 단일 입력 채널을 가진 이차원 교차 상관과 동일하다.

예를 들어, Fig. 16.3.3의 다중 입력 채널을 가진 일차원 교차 상관의 등가 형태는 Fig. 16.3.4의 단일 입력 채널을 가진 이차원 교차 상관이다. 이 경우 컨볼루션 커널의 높이는 입력 텐서와 동일해야 한다.

The textCNN Model

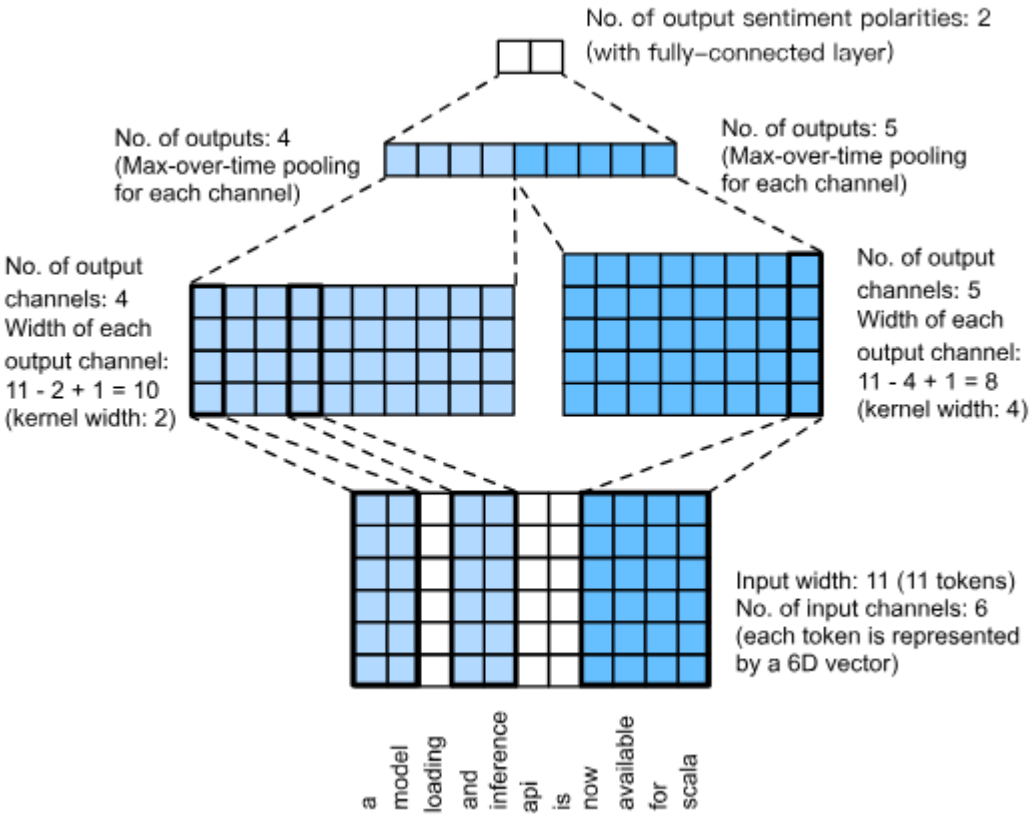
일차원 합성곱과 최대-시간 풀링을 사용하여, textCNN 모델은 개별 사전 훈련된 토큰 표현을 입력으로 사용하고, 그 후에는 시퀀스 표현을 다운스트림 응용 프로그램에 맞게 얻고 변환한다.

d차원 벡터로 표현된 n개의 토큰으로 이루어진 단일 텍스트의 경우:

입력 텐서의 너비, 높이, 그리고 채널 수는 각각 n, 1, d이다.

textCNN 모델은 다음과 같이 입력을 출력으로 변환한다.

1. 여러 개의 일차원 컨볼루션 커널을 정의하고 각각의 입력에 대해 별도로 컨볼루션 연산을 수행한다. 다른 폭을 가진 컨볼루션 커널은 서로 다른 수의 인접한 토큰 간의 지역적 특징을 포착할 수 있다.
2. 모든 출력 채널에 대해 최대-시간 풀링을 수행하고, 그 다음에 모든 스칼라 풀링 출력을 벡터로 연결한다.
3. 연결된 벡터를 완전히 연결된 레이어를 사용하여 출력 카테고리로 변환한다. 과적합을 줄이기 위해 Dropout을 사용할 수 있다.



16.3.5

위 그림은 textCNN의 모델 아키텍처를 예제와 함께 설명하고 있다.

입력은 각 토큰이 6차원 벡터로 표현되는 11개의 토큰으로 이루어진 문장이다. 따라서 폭이 11이고 6개의 채널을 가진 입력이 있다.

각각 폭이 2와 4인 두 개의 일차원 컨볼루션 커널을 정의하고, 각각 4개와 5개의 출력 채널을 가지도록 한다. 이것은 각각 '11-2+1=10'의 폭을 가진 4개의 출력 채널과 '11-4+1=8'의 폭을 가진 5개의 출력 채널을 생성한다.

이 9개 채널의 다른 폭에도 불구하고 최대-시간 풀링은 연결된 9차원 벡터를 생성하며, 이는 최종적으로 이진 감정 예측을 위한 2차원 출력 벡터로 변환된다.

Defining the Model

아래에서 textCNN 모델을 구현하였다.

```
class TextCNN(nn.Module):
    def __init__(self, vocab_size, embed_size, kernel_sizes,
                 **kwargs):
        super(TextCNN, self).__init__(**kwargs)
        self.embedding = nn.Embedding(vocab_size, embed_size)
        # The embedding layer not to be trained
        self.constant_embedding = nn.Embedding(vocab_size, em
        self.dropout = nn.Dropout(0.5)
        self.decoder = nn.Linear(sum(num_channels), 2)
        # The max-over-time pooling layer has no parameters,
        # can be shared
        self.pool = nn.AdaptiveAvgPool1d(1)
        self.relu = nn.ReLU()
        # Create multiple one-dimensional convolutional layer
        self.convs = nn.ModuleList()
        for c, k in zip(num_channels, kernel_sizes):
            self.convs.append(nn.Conv1d(2 * embed_size, c, k)

    def forward(self, inputs):
        # Concatenate two embedding layer outputs with shape
        # of tokens, token vector dimension) along vectors
        embeddings = torch.cat((
            self.embedding(inputs), self.constant_embedding(i
        # Per the input format of one-dimensional convolution
```

```
        # rearrange the tensor so that the second dimension s
        embeddings = embeddings.permute(0, 2, 1)
        # For each one-dimensional convolutional layer, after
        # pooling, a tensor of shape (batch size, no. of chan
        # obtained. Remove the last dimension and concatenate
        encoding = torch.cat([
            torch.squeeze(self.relu(self.pool(conv(embeddings
            for conv in self.convs], dim=1)
        outputs = self.decoder(self.dropout(encoding))
        return outputs
```

textCNN 인스턴스를 만들자.

커널 너비는 3, 4, 5이고 모두 100개의 출력 채널을 갖는 3개의 컨볼루션 레이어가 있다.

```
embed_size, kernel_sizes, nums_channels = 100, [3, 4, 5], [10
devices = d2l.try_all_gpus()
net = TextCNN(len(vocab), embed_size, kernel_sizes, nums_chan

def init_weights(module):
    if type(module) in (nn.Linear, nn.Conv1d):
        nn.init.xavier_uniform_(module.weight)

net.apply(init_weights);
```

사전 훈련된 단어 벡터 로드

16.2와 동일하게 사전 학습된 100차원 GloVe 임베딩을 초기화된 토큰 표현으로 로드한다.

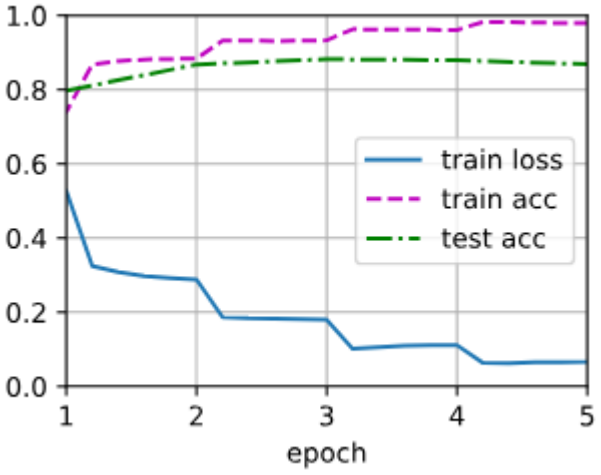
```
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.data.copy_(embeds)
net.constant_embedding.weight.requires_grad = False
```

Training and Evaluationg the Model

이를 훈련하고 감정 분석을 진행해보자.

```
lr, num_epochs = 0.001, 5
trainer = torch.optim.Adam(net.parameters(), lr=lr)
loss = nn.CrossEntropyLoss(reduction="none")
d2l.train_ch13(net, train_iter, test_iter, loss, trainer, num
```

💡 loss 0.066, train acc 0.979, test acc 0.868
4354.2 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]




```
d2l.predict_sentiment(net, vocab, 'this movie is so great')
```

💡 'positive'

```
d2l.predict_sentiment(net, vocab, 'this movie is so bad')
```

💡 'negative'

16.4. Natural Language Inference and the Dataset

자연어 추론은 전제와 가설 두 텍스트 시퀀스 간의 논리적 관계를 결정하는 작업으로, 가설이 전제에서 추론될 수 있는지 여부를 확인한다.

이러한 관계는 일반적으로 세 가지 유형으로 나뉜다.

1. 함의(Entailment): 가설이 전제에서 추론될 수 있다.
2. 모순(Contradiction): 가설의 부정이 전제에서 추론될 수 있다.
3. 중립(Neutral): 그 외의 경우

자연어 추론은 또한 텍스트의 두 시퀀스 간의 관계를 판별하는 과제로 '텍스트 함의 인식 작업'으로도 알려져 있다.

자연어 추론은 자연어를 이해하는 데 중요한 주제로 여겨져 왔다. 이는 정보 검색부터 오픈 도메인 질문 응답까지 다양한 응용 분야에서 활용된다.

이 장은 Standford Natural Language Inference(SNLI)을 사용하여 실습하였다.

SNLI는 500,000개가 넘는 레이블된 영어 문장 쌍의 모음이다.

다운로드

```
import os
import re
import torch
from torch import nn
from d2l import torch as d2l

#@save
d2l.DATA_HUB['SNLI'] = (
    'https://nlp.stanford.edu/projects/snli/snli_1.0.zip',
    '9fcde07509c7e87ec61c640c1b2753d9041758e4')
```

```
data_dir = d2l.download_extract('SNLI')
```

데이터셋 읽기

실습에 필요한 정보만 읽어들이기 위해 데이터셋의 일부만 추출하고 전제, 가설 및 해당 레이블 목록을 반환하는 함수를 정의하자.

```
#@save
def read_snli(data_dir, is_train):
    """Read the SNLI dataset into premises, hypotheses, and l
    def extract_text(s):
        # Remove information that will not be used by us
        s = re.sub('\\(', '', s)
        s = re.sub('\\)', '', s)
        # Substitute two or more consecutive whitespace with
        s = re.sub('\\s{2,}', ' ', s)
        return s.strip()
    label_set = {'entailment': 0, 'contradiction': 1, 'neutra
    file_name = os.path.join(data_dir, 'snli_1.0_train.txt'
                             if is_train else 'snli_1.0_test.
    with open(file_name, 'r') as f:
        rows = [row.split('\t') for row in f.readlines()[1:]]
    premises = [extract_text(row[1]) for row in rows if row[0
    hypotheses = [extract_text(row[2]) for row in rows if row
    labels = [label_set[row[0]] for row in rows if row[0] in
    return premises, hypotheses, labels
```

전제와 가설의 첫 3쌍과 레이블을 출력해보자.

0: 함의 (Entailment)

1: 모순 (Contradiction)

2: 중립 (Neutral)

```
train_data = read_snli(data_dir, is_train=True)
for x0, x1, y in zip(train_data[0][:3], train_data[1][:3], tr
    print('premise:', x0)
```

```
    print('hypothesis:', x1)
    print('label:', y)
```

premise: A person on a horse jumps over a broken down airplane .
hypothesis: A personis training his horsefor a competition .
label: 2
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A personis at a diner , ordering an omelette .
label: 1
premise: A person on a horse jumps over a broken down airplane .
hypothesis: A personis outdoors , on a horse .
label: 0

Defining a Class for Loading the Dataset

아래에서는 Gluon의 Dataset 클래스를 상속하여 SNLI 데이터셋을 로드하기 위한 클래스를 정의한다.

num_steps: 각 미니배치의 시퀀스 길이를 지정하며, 이를 통해 각 시퀀스의 모양이 동일하게 유지된다.

→ 더긴 시퀀스의 첫 num_steps 토큰 이후의 토큰은 잘리며, 더 짧은 시퀀스에는 그 길이가 num_steps가 될 때까지 특수 토큰 <pad>가 추가된다.

getitem 함수를 구현함으로써 인덱스(idx)를 사용하여 임의로 전제, 가설 및 레이블에 접근할 수 있다.

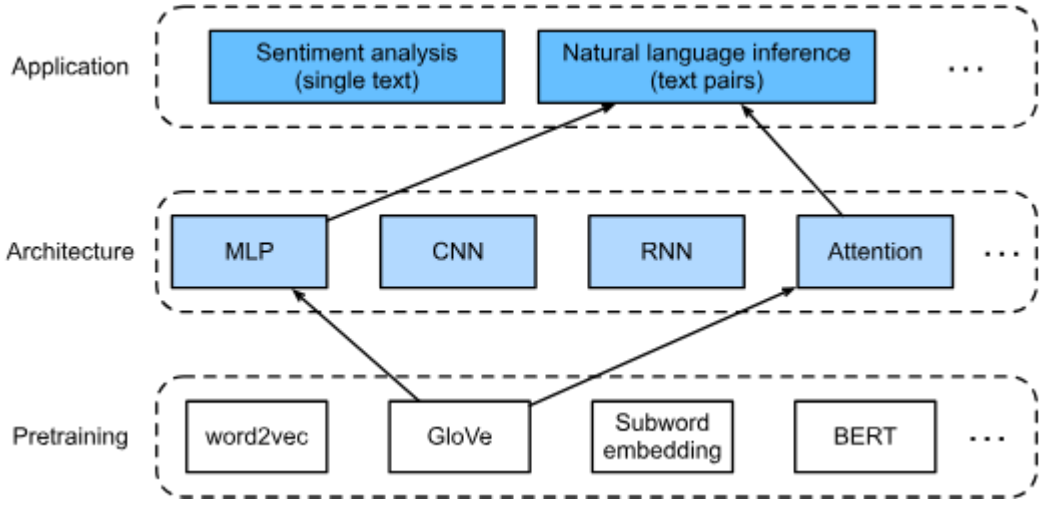
```
#@save
class SNLIDataset(torch.utils.data.Dataset):
    """A customized dataset to load the SNLI dataset."""
    def __init__(self, dataset, num_steps, vocab=None):
        self.num_steps = num_steps
        all_premise_tokens = d2l.tokenize(dataset[0])
        all_hypothesis_tokens = d2l.tokenize(dataset[1])
        if vocab is None:
            self.vocab = d2l.Vocab(all_premise_tokens + all_h
                                   min_freq=5, reserved_token
        else:
```

```
            self.vocab = vocab
        self.premises = self._pad(all_premise_tokens)
        self.hypotheses = self._pad(all_hypothesis_tokens)
        self.labels = torch.tensor(dataset[2])
        print('read ' + str(len(self.premises)) + ' examples'

    def _pad(self, lines):
        return torch.tensor([d2l.truncate_pad(
            self.vocab[line], self.num_steps, self.vocab['<pa
                        for line in lines])

    def __getitem__(self, idx):
        return (self.premises[idx], self.hypotheses[idx]), se

    def __len__(self):
        return len(self.premises)
```

이제 read_snli 함수와 SNLIDataset 클래스를 호출하여 SNLI 데이터셋을 다운로드하고 교육 및 테스트 세트에 대한 DataLoader 인스턴스와 함께 교육 세트의 어휘를 반환할 수 있다.

```
#@save
def load_data_snli(batch_size, num_steps=50):
    """Download the SNLI dataset and return data iterators an
    num_workers = d2l.get_dataloader_workers()
    data_dir = d2l.download_extract('SNLI')
    train_data = read_snli(data_dir, True)
    test_data = read_snli(data_dir, False)
    train_set = SNLIDataset(train_data, num_steps)
    test_set = SNLIDataset(test_data, num_steps, train_set.vo
    train_iter = torch.utils.data.DataLoader(train_set, batch
                                             shuffle=True,
                                             num_workers=num_
    test_iter = torch.utils.data.DataLoader(test_set, batch_s
                                            shuffle=False,
                                            num_workers=num_w
    return train_iter, test_iter, train_set.vocab
```

16.5. Natural Language Inference: Using Attention



Attention 및 MLP를 기반으로 하는 아키텍처에 사전 훈련된 GloVe를 제공하여 자연어 추론 실습을 해보자.

The Model

전제와 가설에서 토큰의 순서를 유지하는 대신, 한 텍스트 시퀀스의 각 토큰을 다른 텍스트 시퀀스의 모든 토큰과 일치시키고 그 반대도 수행한 다음, 이러한 정보를 비교하고 종합하여 전제와 가설 간의 논리적 관계를 예측할 수 있다.

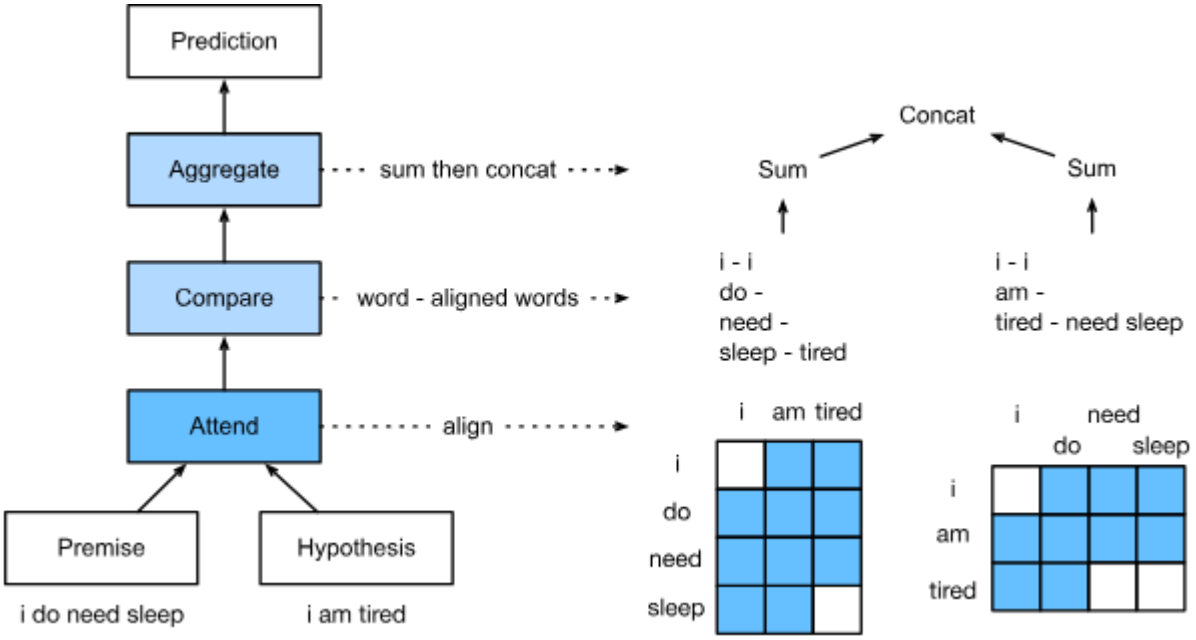


Fig. 16.5.2 Natural Language inference using attention mechanisms

위의 그림은 Attention 매커니즘을 사용한 자연어 추론 방법을 나타낸 것이다.

이는 세 단계(Attend, Compare, Aggregate)로 구성되어 있다.

Attending

Attending은 한 텍스트 시퀀스의 각 토큰을 다른 시퀀스의 각 토큰에 정렬하는 것이다.

예를 들어,

전제: i do need sleep

가설: i am tired

라고 할 때,

> 💡 가설의 'i'→ 전제의 'i'에 정렬하고, 가설의 'tired' → 전제의 'sleep'에 정렬하는 것,
> 또는 가설의 'i' → 전제의 'i', 가설의 'need', 'sleep' → 전제의 'tired'에 정렬하는 것.

이러한 정렬은 가중 평균을 사용하여 부드럽게 이루어지고, 정렬할 토큰에 대해 큰 가중치가 할당된다.

제를 $\mathbf{A} = (\mathbf{a}_1, \ldots, \mathbf{a}_m)$이라고 표시하고, 가설을 $\mathbf{B} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)$이라고 표시할 때, 각각의 토큰 수는 m, n이며, $\mathbf{a}_i, \mathbf{b}_j \in \mathbb{R}^d$은 d 차원의 단어 벡터이다.

attention weights(주의 가중치) eij ∈ R를 계산하는 식은 다음과 같다.

$$e_{ij} = f(\mathbf{a}_i)^\top f(\mathbf{b}_j),$$

여기서 함수 f는 다음과 같이 정의된 MLP이다.

f=mlp(num_hiddens)

```
def mlp(num_inputs, num_hiddens, flatten):
    net = []
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_inputs, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    net.append(nn.Dropout(0.2))
    net.append(nn.Linear(num_hiddens, num_hiddens))
    net.append(nn.ReLU())
    if flatten:
        net.append(nn.Flatten(start_dim=1))
    return nn.Sequential(*net)
```

함수 f는 입력으로 ai, bi를 별도로 취하는 대신에 이들의 쌍을 함께 입력으로 사용하지 않는다.

이 decomposition(분해) 기법은 f의 응용 횟수를 mn 대신에 m+n(선형 복잡도)로 줄인다.

주의 가중치를 정규화한 후, 명제에 인덱싱된 토큰과 부드럽게 정렬된 명제의 모든 토큰 벡터의 가중 평균을 계산하여 전제와 부드럽게 정렬된 가설의 표현을 얻는다.

$$\boldsymbol{\beta}_i = \sum_{j=1}^{n} \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} \mathbf{b}_j.$$

또한, 각각의 토큰에 대해 정렬을 계산하여 전제 토큰을 얻는다.

$$\boldsymbol{\alpha}_j = \sum_{i=1}^{m} \frac{\exp(e_{ij})}{\sum_{k=1}^{m} \exp(e_{kj})} \mathbf{a}_i.$$

아래에서는 'Attend' 클래스를 정의하여 입력 전제 A와 가설 B에 대한 soft alignment을 계산한다.

```
class Attend(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Attend, self).__init__(**kwargs)
        self.f = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B):
        # Shape of `A`/`B`: (`batch_size`, no. of tokens in s
        # `embed_size`)
        # Shape of `f_A`/`f_B`: (`batch_size`, no. of tokens
        # `num_hiddens`)
        f_A = self.f(A)
        f_B = self.f(B)
        # Shape of `e`: (`batch_size`, no. of tokens in seque
        # no. of tokens in sequence B)
        e = torch.bmm(f_A, f_B.permute(0, 2, 1))
        # Shape of `beta`: (`batch_size`, no. of tokens in se
        # `embed_size`), where sequence B is softly aligned w
        # (axis 1 of `beta`) in sequence A
        beta = torch.bmm(F.softmax(e, dim=-1), B)
        # Shape of `alpha`: (`batch_size`, no. of tokens in s
        # `embed_size`), where sequence A is softly aligned w
        # (axis 1 of `alpha`) in sequence B
        alpha = torch.bmm(F.softmax(e.permute(0, 2, 1), dim=-
        return beta, alpha
```

Comparing

이 단계에서는 한 시퀀스의 토큰을 해당 토큰과 soft alignment된 다른 시퀀스와 비교한다.

soft alignment에서는 attention weights가 다를지라도 한 시퀀스의 모든 토큰이 다른 시퀀스의 토큰과 비교된다.

비교 단계에서는 한 시퀀스의 토큰과 다른 시퀀스에서 정렬된 토큰을 연결한 후, 이러한 연결된 표현을 함수 g(MLP)에 입력한다.

$$\mathbf{v}_{A,i} = g([\mathbf{a}_i, \boldsymbol{\beta}_i]), i = 1, \ldots, m$$
$$\mathbf{v}_{B,j} = g([\mathbf{b}_j, \boldsymbol{\alpha}_j]), j = 1, \ldots, n.$$

$\mathbf{v}_{A,i}$는 전제의 토큰 i와 부드럽게 정렬된 모든 가설 토큰 간의 비교이다. 마찬가지로 $\mathbf{v}_{B,j}$는 가설의 토큰 j와 토큰 j와 부드럽게 정렬된 모든 전제 토큰 간의 비교이다.

```
class Compare(nn.Module):
    def __init__(self, num_inputs, num_hiddens, **kwargs):
        super(Compare, self).__init__(**kwargs)
        self.g = mlp(num_inputs, num_hiddens, flatten=False)

    def forward(self, A, B, beta, alpha):
        V_A = self.g(torch.cat([A, beta], dim=2))
        V_B = self.g(torch.cat([B, alpha], dim=2))
        return V_A, V_B
```

Aggregating

비교 벡터 셋 두 개인 $\mathbf{v}_{A,i}(i = 1, \ldots, m)$ 와 $\mathbf{v}_{B,j}(j = 1, \ldots, n)$를 보유한 상태에서, 마지막 단계에서는 이러한 정보를 종합하여 논리적 관계를 추론한다.

먼저 두 세트를 모두 합산한다.

$$\mathbf{v}_A = \sum_{i=1}^{m} \mathbf{v}_{A,i}, \quad \mathbf{v}_B = \sum_{j=1}^{n} \mathbf{v}_{B,j}.$$

다음으로 두 요약 결과를 연결하여 함수 h (MLP)에 입력하여 논리적 관계의 분류 결과를 얻는다.

$$\hat{\mathbf{y}} = h([\mathbf{v}_A, \mathbf{v}_B]).$$

```
class Aggregate(nn.Module):
    def __init__(self, num_inputs, num_hiddens, num_outputs,
        super(Aggregate, self).__init__(**kwargs)
        self.h = mlp(num_inputs, num_hiddens, flatten=True)
        self.linear = nn.Linear(num_hiddens, num_outputs)

    def forward(self, V_A, V_B):
        # Sum up both sets of comparison vectors
        V_A = V_A.sum(dim=1)
        V_B = V_B.sum(dim=1)
        # Feed the concatenation of both summarization result
        Y_hat = self.linear(self.h(torch.cat([V_A, V_B], dim=
        return Y_hat
```

Putting It All Together

위의 과정을 모두 합쳐 모델을 구현해보자.

```
class DecomposableAttention(nn.Module):
    def __init__(self, vocab, embed_size, num_hiddens, num_in
                 num_inputs_compare=200, num_inputs_agg=400,
        super(DecomposableAttention, self).__init__(**kwargs)
        self.embedding = nn.Embedding(len(vocab), embed_size)
        self.attend = Attend(num_inputs_attend, num_hiddens)
        self.compare = Compare(num_inputs_compare, num_hidden
        # There are 3 possible outputs: entailment, contradic
        self.aggregate = Aggregate(num_inputs_agg, num_hidden

    def forward(self, X):
        premises, hypotheses = X
        A = self.embedding(premises)
        B = self.embedding(hypotheses)
        beta, alpha = self.attend(A, B)
        V_A, V_B = self.compare(A, B, beta, alpha)
        Y_hat = self.aggregate(V_A, V_B)
        return Y_hat
```

Training and Evaluating the Model

데이터셋을 읽고, 모델을 생성한다.

```
batch_size, num_steps = 256, 50
train_iter, test_iter, vocab = d2l.load_data_snli(batch_size,

embed_size, num_hiddens, devices = 100, 200, d2l.try_all_gpus
net = DecomposableAttention(vocab, embed_size, num_hiddens)
glove_embedding = d2l.TokenEmbedding('glove.6b.100d')
embeds = glove_embedding[vocab.idx_to_token]
net.embedding.weight.data.copy_(embeds);
```

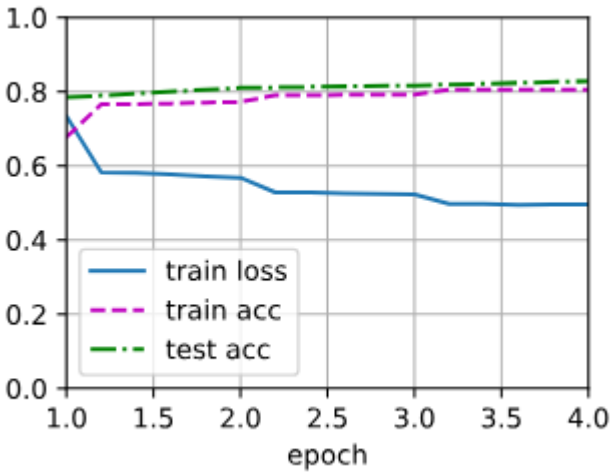
모델을 훈련시킨다.

```
#@save
def split_batch_multi_inputs(X, y, devices):
    """Split multi-input `X` and `y` into multiple devices.""
    X = list(zip(*[gluon.utils.split_and_load(
        feature, devices, even_split=False) for feature in X]
    return (X, gluon.utils.split_and_load(y, devices, even_sp
```

한 번 검증해보자!

```
#@save
def split_batch_multi_inputs(X, y, devices):
    """Split multi-input `X` and `y` into multiple devices.""
    X = list(zip(*[gluon.utils.split_and_load(
        feature, devices, even_split=False) for feature in X]
    return (X, gluon.utils.split_and_load(y, devices, even_sp
```

loss 0.496, train acc 0.805, test acc 0.828
20383.2 examples/sec on [device(type='cuda', index=0), device(type='cuda', index=1)]



Using the Model

```
#@save
def predict_snli(net, vocab, premise, hypothesis):
    """Predict the logical relationship between the premise a
    net.eval()
    premise = torch.tensor(vocab[premise], device=d2l.try_gpu
    hypothesis = torch.tensor(vocab[hypothesis], device=d2l.t
    label = torch.argmax(net([premise.reshape((1, -1)),
                          hypothesis.reshape((1, -1))]), dim
    return 'entailment' if label == 0 else 'contradiction' if
           else 'neutral'
```

```
predict_snli(net, vocab, ['he', 'is', 'good', '.'], ['he', 'i
```

💡 'contradiction'