파이토치로 배우는 자연어 처리

-4장 : 자연어 처리를 위한 피드 포워드 신경망

이상윤

feed-forward network

- 다층 퍼셉트론 (MLP)
  - 퍼셉트론을 구조적으로 확장한 신경망

- 합성곱 신경망 (CNN)
  - 디지털 신호 처리에 사용하는 윈도우 필터에 영향을 받아 만든 신경망
  - 입력의 국부 패턴을 학습할 수 있어 컴퓨터 비전에 적합
  - 단어나 문장 같은 순차 데이터에서 부분 구조 감지에도 용이

feed-forward network <> 순환 신경망 (RNN)

MLP

- 다층 퍼셉트론 (MLP)
  - 퍼셉트론 : 데이터 벡터를 입력으로 받고 출력값 하나를 계산
  - 다층 퍼셉트론 : 많은 퍼셉트론이 모여있으므로 층의 출력은 벡터

[그림4-2] 3단계의 표현과 Linear 층 2개로 구성한 간단한 MLP

첫 번째 선형층이 입력벡터를 받아 은닉벡터를 계산하고,
두 번째 선형층이 은닉벡터를 받아 출력벡터를 계산한다.

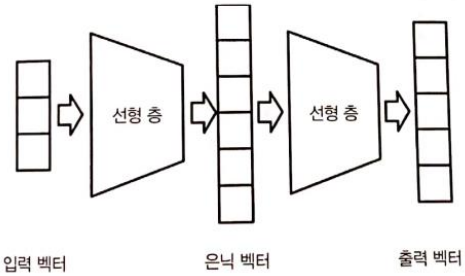


그림 4-2 선형 층 2개와 3단계 표현(입력 벡터, 은닉 벡터, 출력 벡터)으로 구성한 MLP의 시각적 표현

간단한 예 XOR

그림과 같이 XOR은 선형적으로 분류가 불가능하다.

퍼셉트론 훈련

하나의 퍼셉트론은 하나의 다층 퍼셉트론입니다. 이를 잘 드러내기 위해 변수 이름을 mlp1이라고 쓰겠습니다. 위에서 정의한 클래스에 num_hidden_layers=0를 지정해 퍼셉트론을 만들겠습니다.

```
input_size = 2
output_size = len(set(LABELS))
num_hidden_layers = 0
hidden_size = 2 # 실제로 사용하지 않지만 지정합니다

seed = 24

torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
np.random.seed(seed)

mlp1 = MultilayerPerceptron(input_size=input_size,
                            hidden_size=hidden_size,
                            num_hidden_layers=num_hidden_layers,
                            output_size=output_size)
print(mlp1)
batch_size = 1000

x_data_static, y_truth_static = get_toy_data(batch_size)
fig, ax = plt.subplots(1, 1, figsize=(10,5))
visualize_results(mlp1, x_data_static, y_truth_static,
                  ax=ax, title='Initial Perceptron State', levels=[0.5])

plt.axis('off')
plt.savefig('perceptron_initial.png', dpi=300)
```

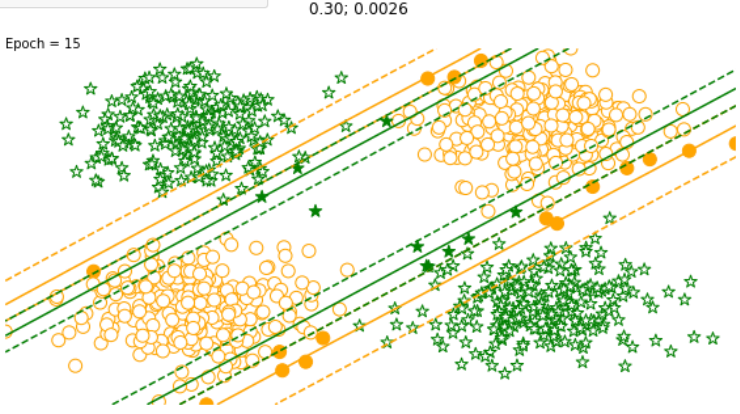
```
MultilayerPerceptron(
  (module_list): ModuleList()
  (fc_final): Linear(in_features=2, out_features=2, bias=True)
)
```

간단한 예 XOR

```
losses = []
batch_size = 10000
n_batches = 10
max_epochs = 15

loss_change = 1.0
last_loss = 10.0
change_threshold = 1e-5
epoch = 0
all_imagefiles = []

lr = 0.01
optimizer = optim.Adam(params=mlp2.parameters(), lr=lr)
cross_ent_loss = nn.CrossEntropyLoss()

def early_termination(loss_change, change_threshold, epoch, max_epochs):
    terminate_for_loss_change = loss_change < change_threshold
    terminate_for_epochs = epoch > max_epochs

    return terminate_for_epochs

while not early_termination(loss_change, change_threshold, epoch, max_epochs):
    for _ in range(n_batches):
        # 단계 0: 데이터 추출
        x_data, y_target = get_toy_data(batch_size)

        # 단계 1: 그레이디언트 초기화
        mlp2.zero_grad()

        # 단계 2: 정방향 계산
        y_pred = mlp2(x_data).squeeze()

        # 단계 3: 손실 계산
        loss = cross_ent_loss(y_pred, y_target.long())

        # 단계 4: 역방향 계산
        loss.backward()

        # 단계 5: 옵티마이저 단계 수행
        optimizer.step()

        # 부가정보
        loss_value = loss.item()
        losses.append(loss_value)
        loss_change = abs(last_loss - loss_value)
        last_loss = loss_value

    fig, ax = plt.subplots(1, 1, figsize=(10,5))
    visualize_results(mlp2, x_data_static, y_truth_static, ax=ax, epoch=epoch,
                      title=f"{loss_value:0.2f}; {loss_change:0.4f}")
    plt.axis('off')
    epoch += 1
    all_imagefiles.append(f'mlp2_epoch{epoch}_toylearning.png')
    plt.savefig(all_imagefiles[-1], dpi=300)
```

2개 층을 가진 다층 퍼셉트론 훈련하기

```
input_size = 2
output_size = len(set(LABELS))
num_hidden_layers = 1
hidden_size = 2

seed = 2

torch.manual_seed(seed)
torch.cuda.manual_seed_all(seed)
np.random.seed(seed)

mlp2 = MultilayerPerceptron(input_size=input_size,
                            hidden_size=hidden_size,
                            num_hidden_layers=num_hidden_layers,
                            output_size=output_size)
print(mlp2)
batch_size = 1000

x_data_static, y_truth_static = get_toy_data(batch_size)
fig, ax = plt.subplots(1, 1, figsize=(10,5))
visualize_results(mlp2, x_data_static, y_truth_static,
                  ax=ax, title='Initial 2-Layer MLP State', levels=[0.5])

plt.axis('off')
plt.savefig('mlp2_initial.png', dpi=300);
```



파이토치로 MLP 구현하기

Linear 클래스 2개로 구성된 MLP 구현

- Linear객체 fc1, fc2 (완전 연결 층 fully connected layer)
- ReLU 활성화 함수가 fc1의 출력에 적용, fc2 입력됨
  (순서대로 놓인 층의 출력-입력 개수가 같아야 하므로)
- 역전파의 정방향 계산만 구현함
  파이토치가 자동으로 역방향, 기울기 업데이트 수행

```python
class MultilayerPerceptron(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        매개변수:
            input_dim (int): 입력 벡터 크기
            hidden_dim (int): 첫 번째 Linear 층의 출력 크기
            output_dim (int): 두 번째 Linear 층의 출력 크기
        """
        super(MultilayerPerceptron, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """MLP의 정방향 계산

        매개변수:
            x_in (torch.Tensor): 입력 데이터 텐서
                x_in.shape는 (batch, input_dim)입니다.
            apply_softmax (bool): 소프트맥스 활성화 함수를 위한 플래그
                크로스-엔트로피 손실을 사용하려면 False로 지정해야 합니다.
        반환값:
            결과 텐서. tensor.shape은 (batch, output_dim)입니다.
        """
        intermediate = F.relu(self.fc1(x_in))
        output = self.fc2(intermediate)

        if apply_softmax:
            output = F.softmax(output, dim=1)
        return output
```

예제: MLP로 성씨 분류하기

- 성씨 데이터셋과 전처리 과정
  - **성씨 데이터셋**
    18개 국적의 성씨 10,000개를 모은 데이터셋.
    특징1. 불균형이 심함. 최상위 클래스 3개가 데이터의 60% 차지 (영어 27%, 러시아어 21%, 아랍어 14%)
    특징2. 출신 국가와 성씨 맞춤법 사이에 의미있고 직관적인 관계가 있다.
  - **전처리 과정**
    Vocabulary를 적용해 성씨 문자열을 벡터로 바꾸는 역할을 함.
    성씨는 문자의 시퀀스 이므로 공백으로 문자열을 나누지 않음. 원-핫 벡터 표현으로 나타냄.
  - **성씨 문자열을 벡터의 미니배치로 변환하는 파이프라인**
    Vocabulary, Vectorizer, DataLoader 클래스를 사용해 성씨 문자열을 벡터의 미니배치로 변환.
    각 문자의 등장 위치에 상관없이 동일한 토큰으로 처리, 단어 토큰을 정수로 매핑하여 벡터로 변환하지 않고 문자를 정수에 매핑하는 식으로 데이터를 벡터로 바꿈.

예제: MLP로 성씨 분류하기

- Vocabulary, Vectorizer, DataLoader
  - **Vocabulary**
    Vocabulary 클래스는 토큰(문자)과 정수 간의 상호 변환에 사용하는 파이썬 딕셔너리, 두 개를 관리함.
    첫 번째 딕셔너리는 문자를 정수 인덱스에 매핑, 두 번째는 정수 인덱스를 문자에 매핑.

  - **SurnameVectorizer**
    원본 데이터셋의 70% 이상이 러시아 이름. (샘플링이 편향되었거나 러시아에 고유한 성씨가 많을 가능성)
    불균형을 줄이기 위해 러시아 성씨의 부분 집합을 랜덤하게 선택해 편중된 클래스를 서브 샘플링함.

  - **SurnameClassifier 모델**
    MLP로 모델을 구현:
    첫 번째 Linear 층이 입력 벡터를 중간 벡터로 매핑하고 이 벡터에 비선형 활성화 함수를 적용.
    두 번째 Linear 층이 중간 벡터를 예측 벡터로 매핑.

예제: MLP로 성씨 분류하기

모델: SurnameClassifier

```
class SurnameClassifier(nn.Module):
    """ 성씨 분류를 위한 다층 퍼셉트론 """
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        매개변수:
            input_dim (int): 입력 벡터 크기
            hidden_dim (int): 첫 번째 Linear 층의 출력 크기
            output_dim (int): 두 번째 Linear 층의 출력 크기
        """
        super(SurnameClassifier, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x_in, apply_softmax=False):
        """MLP의 정방향 계산

        매개변수:
            x_in (torch.Tensor): 입력 데이터 텐서
                x_in.shape는 (batch, input_dim)입니다.
            apply_softmax (bool): 소프트맥스 활성화 함수를 위한 플래그
                크로스-엔트로피 손실을 사용하려면 False로 지정해야 합니다.
        반환값:
            결과 텐서. tensor.shape은 (batch, output_dim)입니다.
        """
        intermediate_vector = F.relu(self.fc1(x_in))
        prediction_vector = self.fc2(intermediate_vector)

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

        return prediction_vector
```

```
def make_train_state(args):
    return {'stop_early': False,
            'early_stopping_step': 0,
            'early_stopping_best_val': 1e8,
            'learning_rate': args.learning_rate,
            'epoch_index': 0,
            'train_loss': [],
            'train_acc': [],
            'val_loss': [],
            'val_acc': [],
            'test_loss': -1,
            'test_acc': -1,
            'model_filename': args.model_state_file}

def update_train_state(args, model, train_state):
    """ 훈련 상태를 업데이트합니다.

    Components:
     - 조기 종료: 과대 적합 방지
     - 모델 체크포인트: 더 나은 모델을 저장합니다

    :param args: 메인 매개변수
    :param model: 훈련할 모델
    :param train_state: 훈련 상태를 담은 딕셔너리
    :returns:
        새로운 훈련 상태
    """

    # 적어도 한 번 모델을 저장합니다
    if train_state['epoch_index'] == 0:
        torch.save(model.state_dict(), train_state['model_filename'])
        train_state['stop_early'] = False

    # 성능이 향상되면 모델을 저장합니다
    elif train_state['epoch_index'] >= 1:
        loss_tm1, loss_t = train_state['val_loss'][-2:]

        # 손실이 나빠지면
        if loss_t >= train_state['early_stopping_best_val']:
            # 조기 종료 단계 업데이트
            train_state['early_stopping_step'] += 1
        # 손실이 감소하면
        else:
            # 최상의 모델 저장
            if loss_t < train_state['early_stopping_best_val']:
                torch.save(model.state_dict(), train_state['model_filename'])

            # 조기 종료 단계 재설정
            train_state['early_stopping_step'] = 0

        # 조기 종료 여부 확인
        train_state['stop_early'] = \
            train_state['early_stopping_step'] >= args.early_stopping_criteria

    return train_state

def compute_accuracy(y_pred, y_target):
    _, y_pred_indices = y_pred.max(dim=1)
    n_correct = torch.eq(y_pred_indices, y_target).sum().item()
    return n_correct / len(y_pred_indices) * 100
```

예제: MLP로 성씨 분류하기

- 모델 평가와 예측
  - **테스트 세트에서 평가하기**
    이 모델은 테스트 세트에서 약 50%의 정확도를 보임. 근본적인 원인은 원-핫 표현이 적합하지 않기 때문, 성씨를 벡터 하나로 간결하게 표현하면 국가 판별에 중요한 문자 사이의 순서 정보를 잃어버리기 때문

  - **새로운 성씨 분류하기**
    문자열로 성씨를 전달하면 함수는 벡터화 과정 적용 후
    모델 예측을 만듬.

  - **새로운 성씨에 대해 최상위 k개 예측 만들기**
    위와 비슷함.

```python
def predict_nationality(surname, classifier, vectorizer):
    """새로운 성씨로 국적 예측하기

    매개변수:
        surname (str): 분류할 성씨
        classifier (SurnameClassifer): 분류기 객체
        vectorizer (SurnameVectorizer): SurnameVectorizer 객체
    반환값:
        가장 가능성이 높은 국적과 확률로 구성된 딕셔너리
    """
    vectorized_surname = vectorizer.vectorize(surname)
    vectorized_surname = torch.tensor(vectorized_surname).view(1, -1)
    result = classifier(vectorized_surname, apply_softmax=True)

    probability_values, indices = result.max(dim=1)
    index = indices.item()

    predicted_nationality = vectorizer.nationality_vocab.lookup_index(index)
    probability_value = probability_values.item()

    return {'nationality': predicted_nationality, 'probability': probability_value}
```

```python
new_surname = input("분류하려는 성씨를 입력하세요: ")
classifier = classifier.to("cpu")
prediction = predict_nationality(new_surname, classifier, vectorizer)
print("{} -> {} (p={:0.2f})".format(new_surname,
                                    prediction['nationality'],
                                    prediction['probability']))
```

```
McMahan -> Irish (p=0.43)
```

CNN

- 합성곱 신경망 (CNN)
  - MLP는 순차 패턴을 감지하는데 유용하지 못하다.
    - 성씨 데이터셋에서 O'Neill의 O, Antonopoulos에서 opoulos, Nagasawa에서 sawa 등 성씨는 출신 국가 정보를 담은 요소를 포함한다.
  - 공간상의 부분 구조를 감지하는데 적합한 CNN, 소수의 가중치를 사용해 입력 데이터 텐서를 스캔하는 식으로 이를 수행

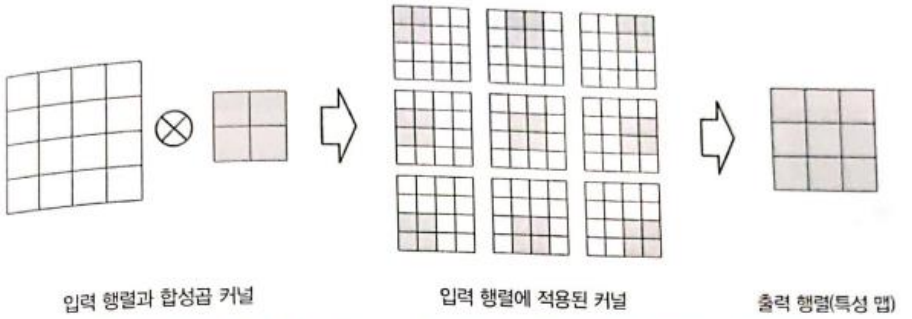


그림 4-6 2차원 합성곱 연산. 입력 행렬이 커널 하나와 합성곱 연산을 수행하여 출력 행렬(또는 특성 맵)을 만듭니다. 합성곱은 입력 행렬의 위치마다 커널을 적용합니다. 합성곱 연산을 수행할 때마다 커널과 입력 행렬의 각 원소를 곱한 후 모두 더합니다. 이 예에서 커널에는 다음과 같은 하이퍼파라미터가 있습니다. kernel_size=2, stride=1, padding=0, dilation=1. 이 하이퍼파라미터들은 다음 문단에서 설명합니다.

CNN 하이퍼파라미터

- 합성곱 연산의 차원
  - 파이토치에서는 1,2,3차원 합성곱이 가능하며 각 Conv1d, 2d, 3d 클래스로 구현
    - NLP에서 합성곱 연산은 대부분 1차원이며 2차원 합성곱은 데이터의 두 방향을 따라 시공간 패턴을 감지 (ex. 이미지의 높이와 너비 차원)
- 채널
  - 입력의 각 포인트에 있는 특성 차원을 의미함. (이미지는 픽셀마다 RGB에 해당하는 차원 3개)
    - 텍스트 문서의 픽셀이 단어라면 채널 개수는 어휘 사전의 크기
    - 문자에 대한 합성곱을 수행한다면 채널 개수는 문자 집합의 크기

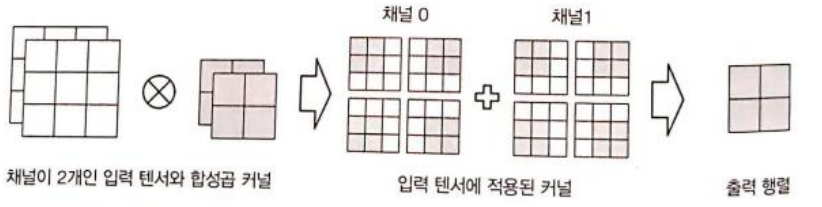


그림 4-7 입력 행렬이 2개(입력 채널이 2개)인 합성곱 연산. 해당 커널에도 채널이 2개 있습니다. 각 채널에 독립적으로 곱한 다음 결과를 모두 더합니다. 하이퍼파라미터 설정은 다음과 같습니다. input_channels=2, output_channels=1, kernel_size=2, stride=1, padding=0, dilation=1

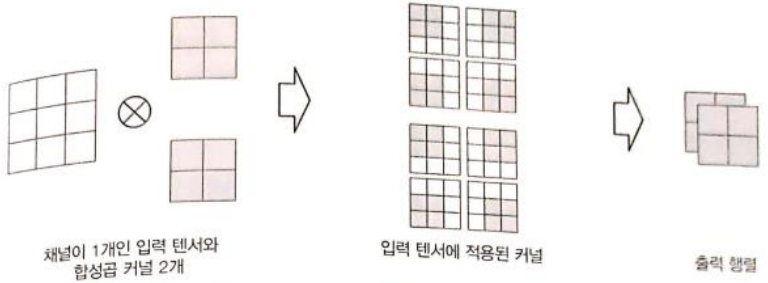


그림 4-8 입력 행렬 1개(입력 채널 1개)와 합성곱 커널 2개(출력 채널 2개)가 있는 합성곱 연산. 커널이 독립적으로 입력 행렬에 적용되어 출력 텐서에 차례대로 쌓입니다. 하이퍼파라미터 설정은 다음과 같습니다. input_channels=1, output_channels=2, kernel_size=2, stride=1, padding=0, dilation=1

CNN 하이퍼파라미터

- 커널 크기
  - 커널 행렬의 너비 (kernel_size)
    - 합성곱마다 얻어지는 정보의 양은 커널 크기로 조절됨 => 커널크기를 늘리면 출력 크기가 줄어듬
- 스트라이드
  - 합성곱 간의 스텝 크기를 제어함
    - 스트라이드가 커널 크기와 같으면 커널 연산이 겹치지 않음. 1일 경우 커널이 가장 많이 겹침.

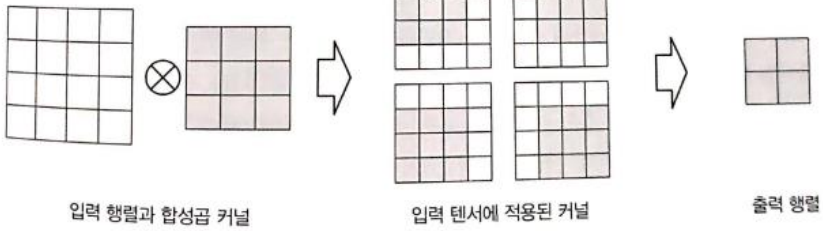


그림 4-9 입력 행렬에 적용한 kernel_size=3인 합성곱. 결과에는 장단점이 있습니다. 커널이 행렬에 적용될 때 더 많은 국부적인 정보가 사용됩니다. 하지만 출력 크기는 작아집니다.

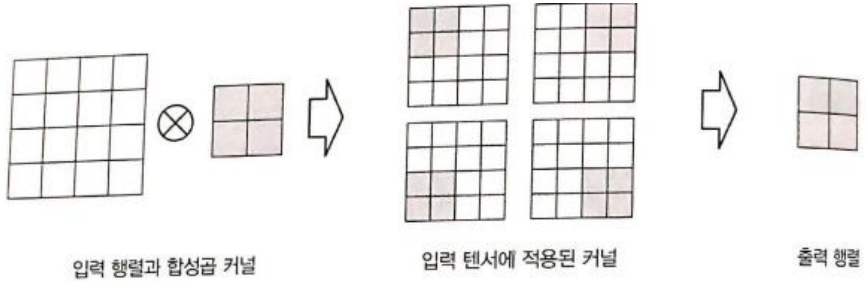


그림 4-10 스트라이드 2로 입력에 적용된 kernel_size=2인 합성곱 커널. 커널이 더 큰 스트라이드를 사용해서 출력 행렬을 작게 만드는 효과를 냅니다. 이는 입력 행렬을 듬성듬성 샘플링하는 데 유용합니다.

CNN 하이퍼파라미터

- 패딩
  - 특성 맵 (합성곱의 출력)의 전체 크기를 유지하기 위해 사용하는 방법
    - 가장 자리에 0을 추가
- 다일레이션
  - 합성곱 커널이 입력 행렬에 적용되는 방식을 제어
    - 커널의 원소 사이에 공간이 생김. 구멍 뚫린 커널을 적용한다고 말함.
    - 파라미터의 개수를 늘리지 않고 넓은 입력 공간을 요약하는 데 유용

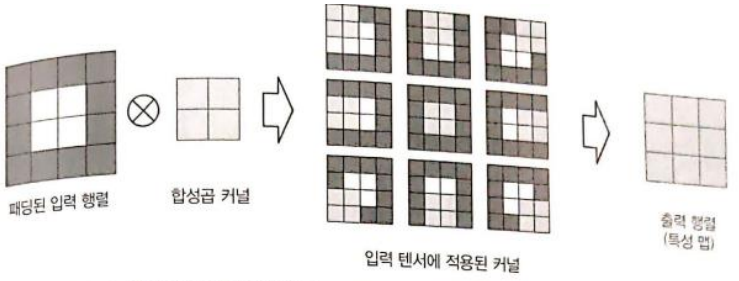


그림 4-11 높이와 너비가 2인 입력 행렬에 적용된 kernel_size=2인 합성곱. 하지만 패딩(어두운 회색 사각형 부분) 때문에 입력 행렬의 높이와 너비는 더 커집니다. 가장 널리 사용하는 커널 크기는 3이므로 출력 행렬이 입력 행렬의 크기와 정확히 같아집니다.

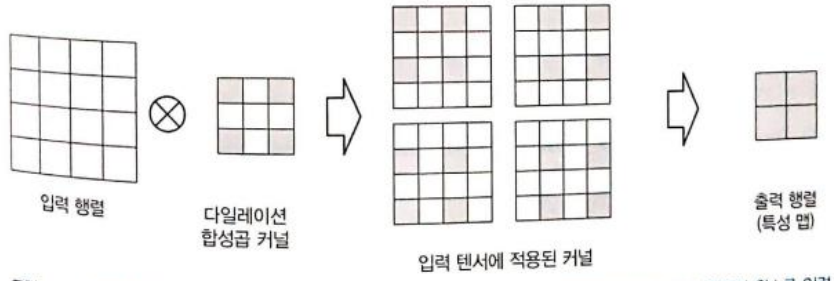


그림 4-12 입력 행렬에 적용된 kernel_size=2인 합성곱. 다일레이션을 기본값보다 늘리면 커널 행렬의 원소를 입력 행렬에 곱할 때 넓게 퍼집니다. 다일레이션을 증가시킬수록 더 많이 퍼집니다.

예제: CNN으로 성씨 분류하기

대부분 MLP로 만든 예제와 같지만 모델 구성과 벡터 변환 과정이 다르다.
모델 입력은 원-핫 표현이 아닌 원-핫 벡터의 행렬이고 CNN이 나열된 문자를 더 잘 볼 수 있게 된다.

- SurnameDataset
  데이터셋 클래스가 가장 긴 선씨를 찾아 이를 SurnameVectorizer 클래스의 객체에 행렬의 행 크기로 전달
  열의 크기는 원-핫 벡터의 크기(Vocabulary의 크기)

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

    매개변수:
        index (int): 데이터 포인트의 인덱스
    반환값:
        데이터 포인트의 특성(x_surname)과 레이블(y_nationality)로 이루어진 딕셔너리
    """
    row = self._target_df.iloc[index]

    surname_matrix = ▼
        self._vectorizer.vectorize(row.surname)

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

    return {'x_surname': surname_matrix,
            'y_nationality': nationality_index}
```

```
class SurnameVectorizer(object):
    """ 어휘 사전을 생성하고 관리합니다 """
    def __init__(self, surname_vocab, nationality_vocab, max_surname_length):
        """
        매개변수:
            surname_vocab (Vocabulary): 문자를 정수에 매핑하는 Vocabulary 객체
            nationality_vocab (Vocabulary): 국적을 정수에 매핑하는 Vocabulary 객체
            max_surname_length (int): 가장 긴 성씨 길이
        """
        self.surname_vocab = surname_vocab
        self.nationality_vocab = nationality_vocab
        self._max_surname_length = max_surname_length

    def vectorize(self, surname):
        """ 성씨에 대한 원-핫 벡터를 만듭니다

        매개변수:
            surname (str): 성씨
        반환값:
            one_hot (np.ndarray): 원-핫 벡터의 행렬
        """
        one_hot_matrix_size = (len(self.surname_vocab), self._max_surname_length)
        one_hot_matrix = np.zeros(one_hot_matrix_size, dtype=np.float32)

        for position_index, character in enumerate(surname):
            character_index = self.surname_vocab.lookup_token(character)
            one_hot_matrix[character_index][position_index] = 1

        return one_hot_matrix
```

예제: CNN로 성씨 분류하기

- SurnameClassifier
  - **Sequential**
    연속적인 선형 연산을 캡슐화해주는 래퍼 클래스, 연속된 Conv1d 층을 캡슐화
  - **ELU**
    ReLU와 비슷한 비선형 함수.

```
class SurnameClassifier(nn.Module):
    def __init__(self, initial_num_channels, num_classes, num_channels):
        """
        매개변수:
            initial_num_channels (int): 입력 특성 벡터의 크기
            num_classes (int): 출력 예측 벡터의 크기
            num_channels (int): 신경망 전체에 사용될 채널 크기
        """
        super(SurnameClassifier, self).__init__()

        self.convnet = nn.Sequential(
            nn.Conv1d(in_channels=initial_num_channels,
                      out_channels=num_channels, kernel_size=3),
            nn.ELU(),
            nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
                      kernel_size=3, stride=2),
            nn.ELU(),
            nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
                      kernel_size=3, stride=2),
            nn.ELU(),
            nn.Conv1d(in_channels=num_channels, out_channels=num_channels,
                      kernel_size=3),
            nn.ELU()
        )
        self.fc = nn.Linear(num_channels, num_classes)

    def forward(self, x_surname, apply_softmax=False):
        """모델의 정방향 계산

        매개변수:
            x_surname (torch.Tensor): 입력 데이터 텐서.
                x_surname.shape은 (batch, initial_num_channels, max_surname_length)입니다.
            apply_softmax (bool): 소프트맥스 활성화 함수를 위한 플래그
                크로스-엔트로피 손실을 사용하려면 False로 지정해야 합니다.
        반환값:
            결과 텐서. tensor.shape은 (batch, num_classes)입니다.
        """
        features = self.convnet(x_surname).squeeze(dim=2)

        prediction_vector = self.fc(features)

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

        return prediction_vector
```

예제: CNN로 성씨 분류하기

- 모델 훈련
  입력 매개변수만 조금 다르다.

```
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/ch4/cnn",
    # 모델 하이퍼파라미터
    hidden_dim=100,
    num_channels=256,
    # 훈련 하이퍼파라미터
    seed=1337,
    learning_rate=0.001,
    batch_size=128,
    num_epochs=100,
    early_stopping_criteria=5,
    dropout_p=0.1,
    # 실행 옵션
    cuda=True,
    reload_from_files=False,
    expand_filepaths_to_save_dir=True,
    catch_keyboard_interrupt=True
)
```

- 평가와 예측
  약 56% 정확성

```
def predict_nationality(surname, classifier, vectorizer):
    """새로운 성씨로 국적 예측하기

    매개변수:
        surname (str): 분류할 성씨
        classifier (SurnameClassifer): 분류기 객체
        vectorizer (SurnameVectorizer): SurnameVectorizer 객체
    반환값:
        가장 가능성이 높은 국적과 확률로 구성된 딕셔너리
    """
    vectorized_surname = vectorizer.vectorize(surname)
    vectorized_surname = torch.tensor(vectorized_surname).unsqueeze(0)
    result = classifier(vectorized_surname, apply_softmax=True)

    probability_values, indices = result.max(dim=1)
    index = indices.item()

    predicted_nationality = vectorizer.nationality_vocab.lookup_index(index)
    probability_value = probability_values.item()

    return {'nationality': predicted_nationality, 'probability': probability_value}
```