Deep Learning Assignment #5 답안

High Performance Integrated Circuit Design Lab.

➢ 실행해보는 문제는 직접 실행하고, 그 캡쳐본 또는 코드 및 결과를 답변으로 제출하시면 됩니다.

Chapter 7 합성곱 신경망 (CNN)

7.1 전체 구조

7.1.1 CNN으로 이뤄진 네트워크(신경망)와 완전연결 계층으로 이루어진 네트워크의 차이점을 서술하시오.

1. 완전연결 신경망은 Affine 계층 뒤에 활성화 함수를 갖는 ReLU 계층이 이어지고 마지막 층은 Affine 계층 뒤에 Softmax 계층에서 최종 결과를 출력한다.
2. CNN에서는 합성곱 계층(Conv) 와 풀링 계층(Pooling)이 추가된다.
3. CNN에서는 Affine-ReLU 연결이 Conv-ReLU-Pooling으로 바뀐다.

7.2 합성곱 계층

7.2.1 완전연결 계층의 문제점

7.2.1.1 완전연결 계층의 문제점과 그 해결책을 서술하시오.

1. 문제점 - 세로 가로 색상의 3차원 이미지 데이터를 1차원으로 평탄화하는 과정에서 데이터가 소실된다.
2. 해결책 - 이미지 데이터를 3차원 데이터로 입력받고 출력하는 합성곱 계층을 사용한다. 이를 입력 특징 맵, 출력 특징 맵이라고도 한다.

7.2.2 합성곱 연산

7.2.2.1 아래의 그림을 보고 입력 데이터에 필터를 적용하여 연산을 수행하고 계산과정을 * 와 + 으로 나타내시오.

1. 3*1+3*2+0*3+5*2 = 19
2. (중략)

7.2.3 패딩

7.2.3.1 합성곱 연산에서 패딩을 적용하는 목적이 무엇인지 설명하시오.

패딩은 출력 크기를 조정할 목적으로 사용된다. 합성곱 연산을 거칠 때마다 크기가 작아지기 때문이다. 패딩을 적용하면 입력 데이터의 공간적 크기를 고정한 채로 다음 계층에 전달할 수 있다.

7.2.4 스트라이드

7.2.4 패딩, 스트라이드와 출력 크기의 관계를 설명하고, 입력: (5, 6), 패딩: 2, 스트라이드:1, 필터: (4,4) 에 해당하는 출력 크기 OH, OW를 계산하시오.

1. 패딩을 키우면 출력 크기는 커진다.
   스트라이드를 키우면 출력 크기는 작아진다.
2. OH= (5 + 2*2 - 4) / 1 + 1 = 6
   OW= (6 + 2*2 - 4) / 1 + 1 = 7

7.2.5 3차원 데이터의 합성곱 연산

2차원 데이터의 합성곱 연산과 3차원 데이터의 합성곱 연산의 차이를 서술하고, 3차원의 합성곱 연산에서 주의할 점을 쓰시오.

1. 2차원과 비교하면 길이방향 (채널방향)으로 특징 맵이 늘어나서 합성곱 연산을 채널마다 수행하고 그 결과를 모두 더해서 하나의 출력을 얻는다.
2. 주의할 점은 입력 데이터의 채널 수와 필터의 채널 수가 같아야 한다.

7.2.6 블록으로 생각하기

7.2.6.1 입력 데이터 * 필터 + 편향 = 출력 데이터의 (합성곱 연산의 처리) 흐름을 하나의 그림으로 나타내시오.

7.2.7 배치 처리

7.2.7.1 배치 처리의 장점을 간단히 설명하시오.

입력 데이터를 한 덩어리로 묶어 한번에 처리함으로써 연산의 속도와 효율성을 높인다.

7.3 풀링 계층

7.3.1 풀링 계층의 특징

7.3.1.1 풀링의 개념을 설명하고, 풀링 계층의 특징 3가지를 서술하시오.

풀링이란 가로세로방향의 공간을 줄이는 연산이다. 최대 풀링은 대상 영역의 최댓값을 구하는 연산이다. 이미지 인식 분야에서는 주로 최대 풀링을 사용한다.

1. 학습해야 할 매개변수가 없다.
2. 채널 수가 변하지 않는다.

3. 입력의 변화에 영향을 적게 받는다.

7.3.1.2 이미지 인식 분야에서는 주로 최대 풀링을 사용한다. 그 이유가 무엇일지 추측해 보고 서술하시오.

최대 풀링은 주어진 풀링 영역에서 가장 큰 값을 선택한다. 이는 이미지의 중요한 특징을 강조하는 효과가 있다. 이미지에서 가장 큰 값은 종종 가장 두드러진 특징을 나타내므로, 최대 풀링을 통해 중요한 정보가 유지되고 강조된다. 또한 노이즈를 무시하는 효과도 있다.

7.4 합성곱/풀링 계층 구현하기

7.4.1 4차원 배열

7.4.1.1 데이터의 형상이 (105, 2, 50,50)이라면 이는 무슷 뜻인가?

높이 50, 너비 50, 채널 2개인 데이터가 105개라는 뜻이다.

7.4.2 im2col로 데이터 전개하기

7.4.2.1 im2col이 무엇인지 설명하시오.

im2col은 입력 데이터를 필터링 (가중치 계산)하기 좋게 펼치는 함수이다. 즉 필터 적용 영역을 앞에서부터 순서대로 1줄로 펼친다.

7.4.2.2 합성곱 연산의 필터 처리 상세 과정을 간단히 그리시오.

(데이터 → im2col) X (필터 → im2col) = 2차원 출력 데이터 → reshape → 4차원 출력 데이터

7.4.3 합성곱 계층 구현하기

7.4.3.1 다음은 im2col 함수의 인터페이스이다. 함수의 인수들을 설명하시오.

```
im2col(input_data, filter_h, filter_w, stride,pad)
```

input_data - (데이터 수, 채널 수, 높이, 너비)의 4차원 배열로 이뤄진 입력 데이터

filter_h - 필터의 높이

filter_w - 필터의 너비

stride - 스트라이드

pad - 패딩

7.4.3.2 im2col을 사용해 합성곱 계층을 구현하라. 합성곱 계층은 Convolution이라는 클래스로 구현한다.

```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        #구현


    def forward(self, x):
        #구현

        return out
```

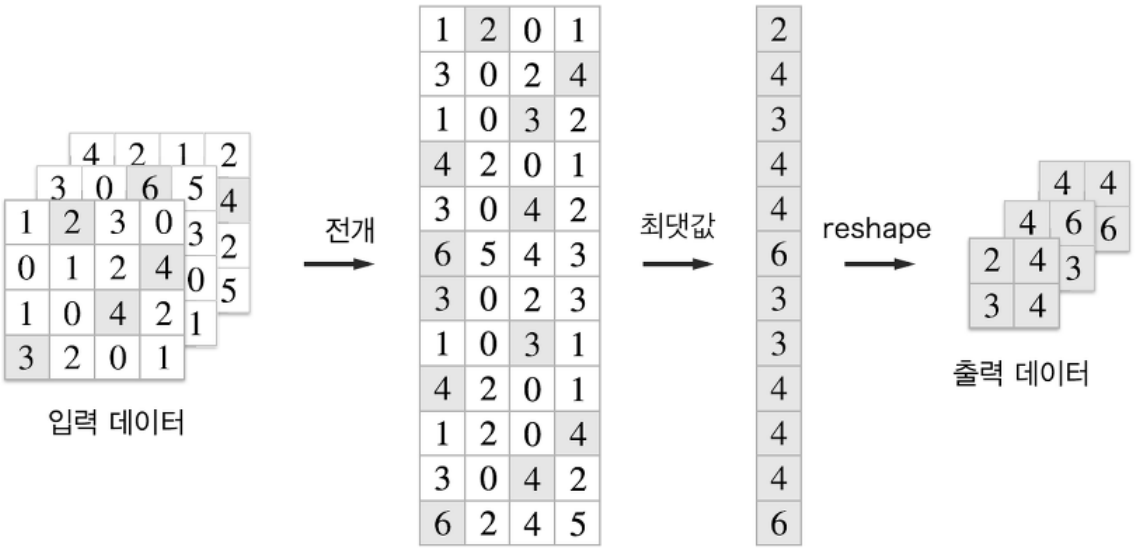
```
class Convolution:
    def __init__(self, W, b, stride=1, pad=0):
        self.W = W  # 필터(weight) 매개변수 초기화
        self.b = b  # 편향(bias) 매개변수 초기화
        self.stride = stride  # 스트라이드 값 초기화
        self.pad = pad  # 패딩 값 초기화

    def forward(self, x):
```

```
        FN, C, FH, FW = self.W.shape  # 필터의 형상(개수, 채널 수, 높이, 너비) 저장
        N, C, H, W = x.shape  # 입력 데이터의 형상(배치 크기, 채널 수, 높이, 너비) 저장
        out_h = (H + 2*self.pad - FH)  # 출력 데이터의 높이 계산 (패딩과 필터 크기 반영)
        out_w = (W + 2*self.pad - FW)  # 출력 데이터의 너비 계산 (패딩과 필터 크기 반영)

        col = im2col(x, FH, FW, self.stride, self.pad)  # 입력 데이터를 컬럼 형태로 전개
        col_W = self.W.reshape(FN, -1).T  # 필터를 컬럼 형태로 전개

        out = np.dot(col, col_W) + self.b  # 행렬 곱 수행 후 편향 더하기
        out = out.reshape(N, out_h, out_w, -1).transpose(0, 3, 1, 2)
        # 결과를 적절한 형상으로 성형 후 축 변경

        return out  # 최종 출력 반환
```

7.4.4 풀링 계층 구현하기

7.4.4.1 다음은 풀링 계층 구현의 forward 처리 흐름이다. 이를 파이썬으로 구현하시오.

1. 입력 데이터를 전개한다
2. 행별 최댓값을 구한다
3. 적절한 모습으로 성형한다




```
class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        #구현

    def forward(self, x):
        #구현

        # 전개

        # 최댓값

        # 성형

        return out
```

```
class Pooling:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h = pool_h  # 풀링 영역의 높이 초기화
        self.pool_w = pool_w  # 풀링 영역의 너비 초기화
        self.stride = stride  # 스트라이드 값 초기화
        self.pad = pad  # 패딩 값 초기화

    def forward(self, x):
        N, C, H, W = x.shape  # 입력 데이터의 형상(배치 크기, 채널 수, 높이, 너비) 저장
        out_h = (H - self.pool_h) // self.stride + 1  # 출력 데이터의 높이 계산
        out_w = (W - self.pool_w) // self.stride + 1  # 출력 데이터의 너비 계산

        # 전개
        col = im2col(x, self.pool_h, self.pool_w, self.stride, self.pad)
        # 입력 데이터를 컬럼 형태로 전개
        col = col.reshape(-1, self.pool_h * self.pool_w)
        # 전개된 데이터를 풀링 영역 크기에 맞게 재배열

        # 최댓값
        out = np.max(col, axis=1)  # 각 풀링 영역에서 최댓값 계산

        # 성형
        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)
        # 결과를 적절한 형상으로 재배열하고 축 변경

        return out  # 최종 출력 반환
```

7.5 CNN 구현하기

합성곱 계층과 풀링 계층을 조합하여 손글씨 숫자를 인식하는 CNN 코드의 구조는 다음과 같다.

[Convolution - ReLU - Pooling - Affine - ReLU - Affine - Softmax]

이를 SimpleConvNet이라는 클래스로 구현한다.

아래 코드는 SimpleConvNet의 (__init__), 초기화 부분이다.

```
class SimpleConvNet:
    def __init__(self, input_dim = (1, 28, 28)
                 conv_param={'filter_num':30, 'filter_size':5,
                             'pad':0, 'stride':1},
                 hidden_size=100, output_size=10, weight_init_std=0.01):
```

7.5.1.1 초기화 때는 위와 같은 인수들을 받는다. 각각의 의미를 서술하시오.

input_dim :

conv_param :

filter_num :

filter_size :

pad :

stride :

hidden_size :

output_size :

weight_init_std :

input_dim : 입력 데이터

conv_param : 합성곱 계층의 하이퍼파라미터

filter_num : 필터 수

filter_size : 필터 크기

pad : 패딩

stride : 스트라이드

hidden_size : 은닉층의 뉴런 수

output_size : 출력층의 뉴런 수

weight_init_std : 초기화 때의 가중치 표준편차

7.5.1.2 초기화 인수로 주어진 합성곱 계층의 하이퍼파라미터를 딕셔너리에서 꺼내고 합성곱 계층의 출력 크기를 계산하는 코드이다. 코드를 완성하시오.

```
class SimpleConvNet:
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param={'filter_num': 30, 'filter_size': 5,
                             'pad': 0, 'stride': 1},
                 hidden_size=100, output_size=10, weight_init_std=0.01):

        # 필터 개수 저장
        # 필터 크기 저장
        # 패딩 크기 저장
        # 스트라이드 크기 저장
        # 입력 이미지의 높이(또는 너비) 저장 (여기서는 28)

        # 합성곱 계층의 출력 크기 계산

        # 풀링 계층의 출력 크기 계산 (여기서는 2x2 풀링을 가정)
```

```
class SimpleConvNet:
    def __init__(self, input_dim=(1, 28, 28),
                 conv_param={'filter_num': 30, 'filter_size': 5,
                             'pad': 0, 'stride': 1},
                 hidden_size=100, output_size=10, weight_init_std=0.01):
        filter_num = conv_param['filter_num']  # 필터 개수 저장
        filter_size = conv_param['filter_size']  # 필터 크기 저장
        filter_pad = conv_param['pad']  # 패딩 크기 저장
        filter_stride = conv_param['stride']  # 스트라이드 크기 저장
        input_size = input_dim[1]  # 입력 이미지의 높이(또는 너비) 저장 (여기서는 28)

        # 합성곱 계층의 출력 크기 계산
        conv_output_size = (input_size - filter_size + 2 * filter_pad) // filter_strid
e + 1

        # 풀링 계층의 출력 크기 계산 (여기서는 2x2 풀링을 가정)
        pool_output_size = int(filter_num * (conv_output_size / 2) * (conv_output_size
/ 2))
```

7.5.1.3 다음 코드는 가중치 매개변수를 초기화하는 부분이다. 학습에 필요한 매개변수는 1번째 층의 합성곱 계층과 나머지 두 완전연결 계층의 가중치와 편향이다. 이 매개변수들은 인스턴스 변수 params 딕셔너리에 저장한다.

코드를 완성하시오.

```
self.params = {}  # 네트워크 매개변수를 저장할 딕셔너리 초기화
self.params['W1'] = # 첫 번째 합성곱 계층의 가중치 초기화
self.params['b1'] = # 첫 번째 합성곱 계층의 편향 초기화
self.params['W2'] = # 두 번째 합성곱 계층의 가중치 초기화
self.params['b2'] = # 두 번째 합성곱 계층의 편향 초기화
self.params['W3'] = # 세 번째 합성곱 계층의 가중치 초기화
self.params['b3'] = # 세 번째 합성곱 계층의 편향 초기화
```

```
self.params = {}  # 네트워크 매개변수를 저장할 딕셔너리 초기화
self.params['W1'] = weight_init_std * \
                    np.random.randn(filter_num, input_dim[0], filter_size, filter_siz
e)  # 첫 번째 합성곱 계층의 가중치 초기화
self.params['b1'] = np.zeros(filter_num)  # 첫 번째 합성곱 계층의 편향 초기화
self.params['W2'] = weight_init_std * \
                    np.random.randn(filter_num, input_dim[0], filter_size, filter_siz
e)  # 두 번째 합성곱 계층의 가중치 초기화
self.params['b2'] = np.zeros(filter_num)  # 두 번째 합성곱 계층의 편향 초기화
self.params['W3'] = weight_init_std * \
                    np.random.randn(filter_num, input_dim[0], filter_size, filter_siz
e)  # 세 번째 합성곱 계층의 가중치 초기화
self.params['b3'] = np.zeros(filter_num)  # 세 번째 합성곱 계층의 편향 초기화
```

7.5.1.4

마지막으로 CNN을 구성하는 계층들을 생성한다. 순서가 있는 딕셔너리인 layers에 계층들을 차례로 추가한다. 마지막 SoftmaxWIthLoss 계층만큼은 last_layer라는 별도 변수에 저장한다.

코드를 완성하시오.

```
self.layers = OrderedDict()  # 레이어를 순서대로 저장할 OrderedDict 초기화
self.layers['Conv1'] = # 첫 번째 합성곱 계층, 가중치 매개변수
                       # 첫 번째 합성곱 계층, 편향 매개변수
                       # 합성곱 계층의 스트라이드
                       # 합성곱 계층의 패딩
self.layers['Relu1'] = # 첫 번째 활성화 함수 (ReLU)
self.layers['pool1'] = # 첫 번째 풀링 계층 (2x2 풀링, 스트라이드 2)
self.layers['Affine1'] = # 첫 번째 완전연결 계층
self.layers['Relu2'] = # 두 번째 활성화 함수 (ReLU)
self.layers['Affine2'] = # 두 번째 완전연결 계층

# 소프트맥스 계층과 손실 함수를 포함하는 마지막 레이어
```

```
self.layers = OrderedDict()  # 레이어를 순서대로 저장할 OrderedDict 초기화
self.layers['Conv1'] = Convolution(self.params['W1'],  # 첫 번째 합성곱 계층, 가중치 매개변
수
                                   self.params['b1'],  # 첫 번째 합성곱 계층, 편향 매개변수
                                   conv_params['stride'],  # 합성곱 계층의 스트라이드
                                   conv_params['pad'])  # 합성곱 계층의 패딩
self.layers['Relu1'] = Relu()  # 첫 번째 활성화 함수 (ReLU)
self.layers['pool1'] = Pooling(pool_h=2, pool_w=2, stride=2)  # 첫 번째 풀링 계층 (2x2 풀
링, 스트라이드 2)
self.layers['Affine1'] = Affine(self.params['W2'], self.params['b2'])  # 첫 번째 완전연결
계층
```

```
self.layers['Relu2'] = Relu()  # 두 번째 활성화 함수 (ReLU)
self.layers['Affine2'] = Affine(self.params['W3'], self.params['b3'])  # 두 번째 완전연결
계층

self.last_layer = SoftmaxWithLoss()  # 소프트맥스 계층과 손실 함수를 포함하는 마지막 레이어
```

7.5.1.5

초기화를 마친 다음에는 추론을 수행하는 predict 메서드와 손실 함수의 값을 구하는 loss 메서드를 다음과 같이 구현한다.

코드를 완성하시오.

```
def predict(self, x):  # 예측을 위한 메서드
    # 모든 레이어를 순차적으로 적용
    # 각 레이어의 순전파(forward) 연산 수행
    # 최종 출력 반환

def loss(self, x, t):  # 손실 계산을 위한 메서드
    # 예측 결과를 얻기 위해 predict 메서드 호출
    # 예측 결과와 정답 레이블 t를 사용하여 손실 계산
```

```
def predict(self, x):  # 예측을 위한 메서드
    for layer in self.layers.values():  # 모든 레이어를 순차적으로 적용
        x = layer.forward(x)  # 각 레이어의 순전파(forward) 연산 수행
    return x  # 최종 출력 반환

def loss(self, x, t):  # 손실 계산을 위한 메서드
    y = self.predict(x)  # 예측 결과를 얻기 위해 predict 메서드 호출
    return self.last_layer.forward(y, t)  # 예측 결과와 정답 레이블 t를 사용하여 손실 계산
```

7.5.1.6

이어서 오차역전파법으로 기울기를 구한다.

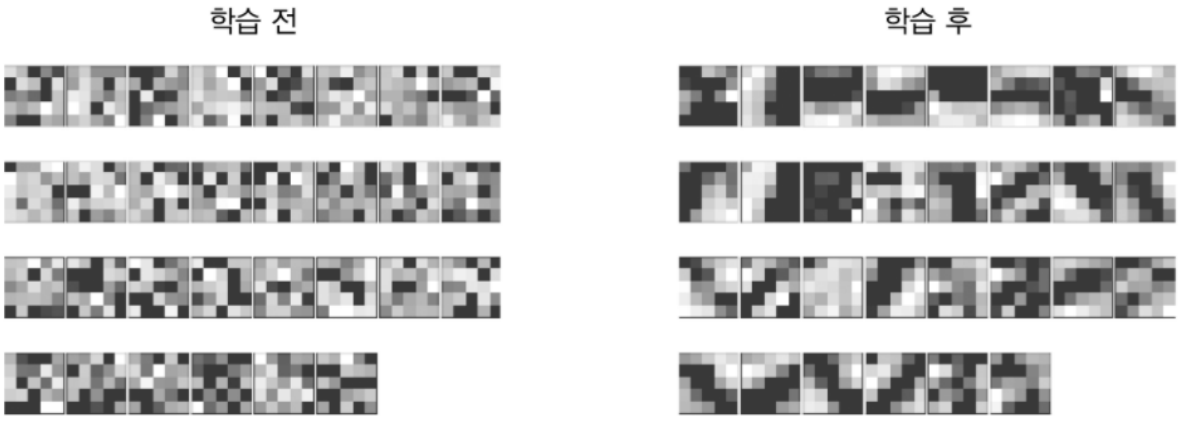
```
def gradient(self, x, t):  # 기울기(gradient)를 계산하는 메서드
    # 순전파
    self.loss(x, t)  # 손실을 계산하면서 순전파 수행

    # 역전파
    dout = # 역전파의 시작은 손실 함수의 미분 값인 1로 설정
    dout = # 마지막 레이어부터 역전파 수행

    layers = # 모든 레이어를 리스트로 변환
    layers.reverse()  # 역전파를 위해 레이어 순서를 뒤집음

    for layer in layers:  # 모든 레이어에 대해 역전파 수행
    # 각 레이어의 backward 메서드 호출

    # 결과 저장
    grads = {}  # 기울기를 저장할 딕셔너리
    grads['W1'] = # Conv1 레이어의 가중치 기울기 저장
    grads['b1'] = # Conv1 레이어의 편향 기울기 저장
    grads['W2'] = # Affine1 레이어의 가중치 기울기 저장
    grads['b2'] = # Affine1 레이어의 편향 기울기 저장
    grads['W3'] = # Affine2 레이어의 가중치 기울기 저장
    grads['b3'] = # Affine2 레이어의 편향 기울기 저장
```

```
    return grads  # 기울기 딕셔너리 반환
```

```
def gradient(self, x, t):  # 기울기(gradient)를 계산하는 메서드
    # 순전파
    self.loss(x, t)  # 손실을 계산하면서 순전파 수행

    # 역전파
    dout = 1  # 역전파의 시작은 손실 함수의 미분 값인 1로 설정
    dout = self.last_layer.backward(dout)  # 마지막 레이어부터 역전파 수행

    layers = list(self.layers.values())  # 모든 레이어를 리스트로 변환
    layers.reverse()  # 역전파를 위해 레이어 순서를 뒤집음

    for layer in layers:  # 모든 레이어에 대해 역전파 수행
        dout = layer.backward(dout)  # 각 레이어의 backward 메서드 호출

    # 결과 저장
    grads = {}  # 기울기를 저장할 딕셔너리
    grads['W1'] = self.layers['Conv1'].dW  # Conv1 레이어의 가중치 기울기 저장
    grads['b1'] = self.layers['Conv1'].db  # Conv1 레이어의 편향 기울기 저장
    grads['W2'] = self.layers['Affine1'].dW  # Affine1 레이어의 가중치 기울기 저장
    grads['b2'] = self.layers['Affine1'].db  # Affine1 레이어의 편향 기울기 저장
    grads['W3'] = self.layers['Affine2'].dW  # Affine2 레이어의 가중치 기울기 저장
    grads['b3'] = self.layers['Affine2'].db  # Affine2 레이어의 편향 기울기 저장

    return grads  # 기울기 딕셔너리 반환
```

7.6 CNN 시각화하기

7.6.1 1번째 층의 가중치 시각화하기



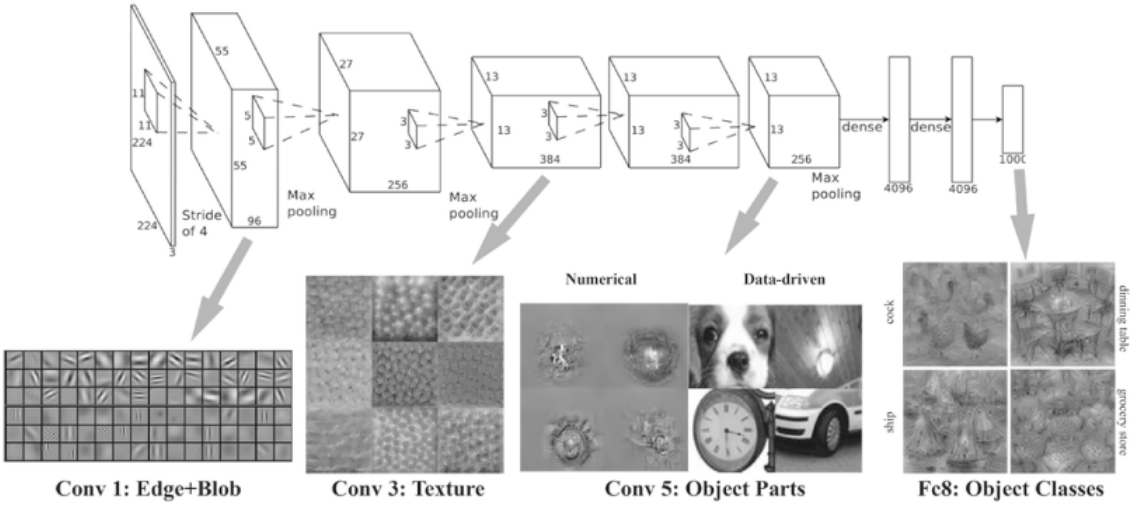
7.6.1.1 위 그림은 학습 전과 후의 1번째 층의 합성곱 계층의 가중치를 시각화한 것이다. 이미지가 어떻게 변했는지 필터와 연관지어 서술하시오.

학습 전 필터는 무작위로 초기화되고 있어 흑백의 정도에 규칙성이 없다.

한편, 학습을 마친 필터는 규칙성이 있는 이미지화 되었다.

흰색에서 검은색으로 점차 변화하는 필터와 덩어리를 가진 필터 등, 규칙을 띄는 필터로 바뀌었다.

7.6.2 층 깊이에 따른 추출 정보 변화



7.6.2.1 위 그림은 층 깊이에 따른 추출 정보 변화에 관한 그림이다. 층이 깊어질수록 추출 정보가 어떻게 변화하는지 서술하고, 그 이유는 무엇일지 추측하시오.

위 그림은 합성곱 신경망(CNN)의 각 층이 입력 데이터로부터 추출하는 특징의 변화를 보여주는 그림이다. 층이 깊어질수록 추출되는 특징이 점점 더 추상화되고 고수준의 정보로 변하는 과정을 시각적으로 설명하고 있다.

1. **Conv 1: Edge + Blob (가장 얕은 층)**:
   - **특징**: 첫 번째 합성곱 층에서는 주로 에지(경계)와 블롭(작은 패턴)과 같은 저수준 특징을 추출한다.
   - **이유**: 초기 층에서는 필터가 작은 영역을 처리하며, 이는 주로 이미지의 기본적인 형태나 패턴을 감지하기 때문이다.
2. **Conv 3: Texture (중간 층)**:
   - **특징**: 중간 층에서는 질감과 같은 더 복잡한 패턴을 추출한다.
   - **이유**: 몇 개의 합성곱과 풀링 층을 거치면서, 네트워크는 작은 형태들을 결합하여 더 복잡한 패턴을 감지할 수 있게 된다.
3. **Conv 5: Object Parts (깊은 층)**:
   - **특징**: 더 깊은 층에서는 객체의 일부분과 같은 고수준 특징을 추출한다.
   - **이유**: 깊은 층에서는 더 큰 수용 영역을 가지며, 이는 여러 저수준 특징을 결합하여 객체의 일부분을 감지할 수 있게 한다.
4. **Fc8: Object Classes (가장 깊은 층)**:
   - **특징**: 마지막 완전연결층에서는 전체 객체 클래스와 같은 가장 고수준 특징을 추출한다.
   - **이유**: 가장 깊은 층에서는 전체 이미지의 맥락을 이해하고, 추출된 고수준 특징들을 결합하여 최종적으로 객체를 분류할 수 있게 된다.

이유 추측

- **계층적 특징 학습**: CNN은 계층적으로 특징을 학습하는 구조를 가지고 있다. 얕은 층에서는 저수준 특징(에지, 코너 등)을 학습하고, 중간 층에서는 이러한 저수준 특징을 결합하여 더 복잡한 패턴(질감 등)을 학습한다. 깊은 층으로 갈수록 객체의 일부분과 같은 고수준 특징을 학습하게 된다.
- **수용 영역의 증가**: 층이 깊어질수록 뉴런의 수용 영역이 커지게 된다. 이는 뉴런이 더 큰 영역의 정보를 종합하여 더 복잡하고 추상화된 특징을 추출할 수 있게 한다.
- **특징 조합의 복잡성 증가**: 깊은 층에서는 많은 저수준 및 중간 수준의 특징을 결합하여 고수준 특징을 형성한다. 이는 신경망이 이미지의 더 높은 수준의 정보를 이해하고 처리할 수 있게 한다.

7.7 대표적인 CNN

7.7.1 CNN의 원조 LeNet, 딥러닝을 주목 받도록 이끈 AlexNet에 대해 서술하고, LeNet과 '현재의 CNN'을 비교한 차이점 2가지, LeNet와 비교한 AlexNet의 차이점 3가지를 설명하시오.

LeNet과 '현재의 CNN'을 비교한 차이점 2가지

첫 번째 차이는 활성화 함수이다.

LeNet은 시그모이드 함수를 사용하는 데 반해, 현재는 주로 ReLU를 사용하다.

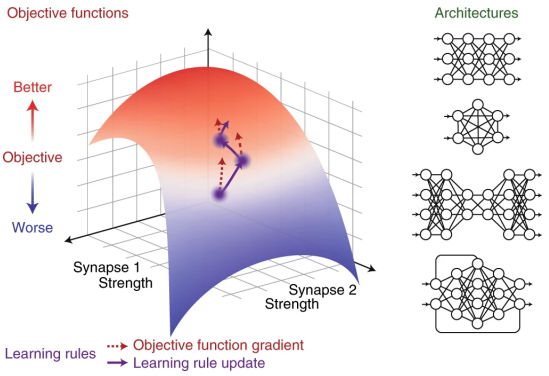
두 번째 차이는 서브 샘플링과 최대 풀링이다.

LeNet은 서브 샘플링을 하여 중간 데이터의 크기를 줄이지만 현재는 최대 풀링이 주류이다.

LeNet와 비교한 AlexNet의 차이점 3가지

1. 활성화 함수로 ReLU를 이용한다.
2. LRN(Local Response Normalization)이라는 국소적 정규화를 실시하는 계층을 이용한다.
3. 드롭아웃을 사용한다.

Chapter 8 딥러닝



8.1 더 깊게

8.1.2 정확도를 더 높이려면

8.1.2.1 정확도 향상을 위해 데이터 확장을 이용하는 이유를 설명하시오.

데이터 확장은 입력이미지(훈련 이미지)를 알고리즘을 동원해 인위적으로 확장한다. 입력 이미지를 회전하거나 세로로 이동하는 등 미세한 변화를 주어 이미지의 개수를 늘릴 수 있는데 이는 데이터가 몇 개 없을 때 특히 효과적이다. 이 외에도 다양한 방법으로 이미지를 확장할 수 있다. 예를 들어 이미지 일부를 잘라내는 crop이나 좌우를 뒤집는 flip 등이 있다. 데이터 확장을 동원해 훈련 이미지의 개수를 늘릴 수 있다면 딥러닝의 인식 수준을 개선할 수 있다.

8.1.3 깊게 하는 이유

8.1.3.1 신경망의 층을 깊게 할 때의 이점을 설명하시오.

층을 깊게할 때의 이점중 하나는 신경망의 매개변수 수가 줄어든다. 층을 깊게 한 신경망은 깊지 않은경우보다 적은 매개변수로 같은 수준 혹은 그 이상의 표현력을 달성할 수 있다.

8.2 딥러닝의 초기 역사

8.2.4 ResNet

8.2.4 ResNet의 구성요소의 도식을 그리고, ResNet에서 도입한 스킵 연결에 대해 서술하라. 이때 층이 깊어져도 학습을 효율적으로 할 수 있는 이유를 설명하시오.

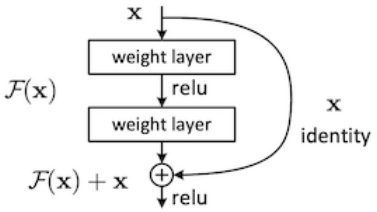


Figure 2. Residual learning: a building block.

스킵 연결은 한 층의 출력을 다음 층의 입력으로 직접 연결하는 대신, 몇 개의 층을 건너뛰어 더 후속 층의 입력에 더하는 방식이다.

8.3 더 빠르게(딥러닝 고속화)

8.3.1 풀어야 할 숙제

8.3.1 딥러닝 모델의 학습과 추론 과정에서 가장 큰 연산 시간을 차지하는 과정은 무엇인가?

딥러닝 연산 과정 중 가장 오래 걸리는 과정은 합성곱 연산이다. 이는 높은 계산 복잡도, 다중 필터 사용, 고해상도 입력 데이터 등의 이유로 인해 많은 연산 자원을 요구하기 때문이다. 이러한 이유로 합성곱 연산은 딥러닝 모델의 학습과 추론 과정에서 가장 큰 연산 시간을 차지한다.

8.3.2 GPU를 활용한 고속화

8.3.2 GPU가 CPU보다 딥러닝 연산에 적합한 이유를 서술하시오.

GPU는 수천 개의 코어를 통한 병렬 처리 능력, 높은 연산 속도와 메모리 대역폭, 특화된 연산 유닛, 딥러닝 프레임워크의 최적화된 지원, 에너지 효율성 등 여러 가지 이유로 CPU보다 딥러닝 연산에 훨씬 더 적합하다. 이러한 특성들 덕분에 GPU는 딥러닝 모델의 학습과 추론을 가속화하고, 대규모 데이터 처리와 복잡한 연산을 효율적으로 수행할 수 있다.

8.4 딥러닝의 활용

8.4.1 사물 검출

8.4.1 사물 검출에 쓰이는 대표적인 기법(R-CNN, YOLO 등)을 소개하고, 사물 검출이 쓰일 수 있는 적용 분야 3가지를 서술하시오.

사물 검출(Object Detection)은 이미지나 비디오에서 객체의 위치와 종류를 식별하는 기술로, 컴퓨터 비전 분야에서 중요한 역할을 한다. 대표적인 사물 검출 기법들은 다음과 같다:

1. **R-CNN (Region-Based Convolutional Neural Networks)**:
   - **R-CNN**: 이미지를 여러 영역으로 분할하고, 각 영역에 대해 CNN을 사용하여 객체를 분류한다. R-CNN은 정확하지만 속도가 느리다.
   - **Fast R-CNN**: R-CNN의 속도를 개선한 버전으로, 영역 제안과 객체 분류를 동시에 수행하여 연산 효율을 높였다.
   - **Faster R-CNN**: Fast R-CNN을 더욱 개선한 버전으로, 영역 제안을 위한 별도의 네트워크(Region Proposal Network, RPN)를 사용하여 성능과 속도를 크게 향상시켰다.
2. **YOLO (You Only Look Once)**:
   - YOLO는 이미지를 그리드로 나누고, 각 그리드 셀에서 객체의 존재 여부와 경계 상자를 예측하는 단일 CNN을 사용한다. YOLO는 실시간 객체 검출이 가능할 정도로 매우 빠르지만, 작은 객체에 대한 검출 성능이 다소 떨어질 수 있다.
3. **SSD (Single Shot MultiBox Detector)**:
   - SSD는 다양한 크기의 경계 상자와 특징 맵을 사용하여 객체를 검출하는 기법으로, YOLO와 유사하게 빠르면서도 작은 객체에 대한 성능을 개선하였다.

사물 검출의 적용 분야

사물 검출 기술은 다양한 산업과 응용 분야에서 중요한 역할을 한다. 다음은 사물 검출이 사용될 수 있는 세 가지 주요 적용 분야이다:

1. **자율 주행 차량**:
   - **교통 신호 및 표지판 인식**: 자율 주행 차량은 도로의 교통 신호와 표지판을 실시간으로 인식하여 적절한 주행 결정을 내릴 수 있다.

- **보행자 및 장애물 검출**: 차량 주변의 보행자와 장애물을 정확하게 검출하여 사고를 예방하고 안전한 주행을 보장한다.

2. **보안 및 감시 시스템**:
   - **침입자 검출**: CCTV 카메라와 연계하여 특정 영역에 침입한 사람이나 물체를 실시간으로 감지하고 경고를 발령할 수 있다.
   - **행동 분석**: 공공 장소에서 사람들의 행동을 분석하여 비정상적인 행동을 감지하고, 범죄 예방 및 사고 대응에 활용할 수 있다.
3. **의료 영상 분석**:
   - **질병 진단**: 의료 영상(예: X-ray, MRI)에서 종양이나 병변과 같은 이상 징후를 자동으로 검출하여 조기 진단과 치료에 도움을 준다.
   - **수술 보조**: 수술 중에 실시간으로 특정 장기나 구조물을 식별하여 수술의 정확성과 안전성을 높인다.

이처럼 사물 검출 기술은 다양한 분야에서 중요한 역할을 하며, 실시간성, 정확성, 효율성을 요구하는 응용에서 특히 유용하게 활용되고 있다.

8.4.2 분할

8.4.2 이미지 분할에 쓰이는 대표적인 기법을 소개하고, 분할이 쓰일 수 있는 적용 분야 3가지를 서술하시오.

이미지 분할에 쓰이는 대표적인 기법

이미지 분할(Image Segmentation)은 이미지에서 픽셀 단위로 객체 또는 영역을 식별하는 기술이다. 이미지 분할은 이미지의 각 픽셀을 특정 클래스나 객체에 할당하는 것을 목표로 한다. 대표적인 이미지 분할 기법들은 다음과 같다:

1. **FCN (Fully Convolutional Networks)**:
   - FCN은 기존의 CNN을 변형하여 완전 연결층을 제거하고, 모든 계층을 합성곱 계층으로 구성한다. 이를 통해 입력 이미지와 동일한 크기의 출력 분할 맵을 생성할 수 있다.
2. **U-Net**:
   - U-Net은 FCN의 확장으로, 주로 의료 영상 분할에 사용된다. U-Net은 대칭적인 U자형 구조를 가지며, 인코더와 디코더 경로를 통해 특징을 추출하고 세밀한 분할 결과를 얻는다.
3. **SegNet**:
   - SegNet은 FCN과 유사한 구조를 가지지만, 디코더 부분에서 맥스풀링 인덱스를 사용하여 원본 해상도를 복원한다. 이는 분할 결과의 공간적 정확성을 높이는 데 도움을 준다.
4. **Mask R-CNN**:
   - Mask R-CNN은 객체 검출과 이미지 분할을 동시에 수행하는 모델이다. Faster R-CNN을 기반으로 하며, 객체의 경계 상자를 예측한 후, 각 객체에 대한 세밀한 마스크를 예측한다.

이미지 분할의 적용 분야

1. **의료 영상 분석**:
   - **종양 및 병변 검출**: MRI, CT, X-ray 등의 의료 영상에서 종양, 병변, 장기 등을 정확히 분할하여 의사의 진단을 돕는다.
   - **조직 분할**: 조직 샘플의 디지털 이미지를 분석하여 세포, 조직 구조 등을 분할함으로써 연구 및 진단에 활용된다.
2. **자율 주행 차량**:
   - **도로 및 차선 인식**: 도로와 차선을 분할하여 차량이 주행할 경로를 인식하고 유지할 수 있게 한다.
   - **주변 객체 분할**: 보행자, 차량, 자전거 등 도로 위의 다양한 객체를 정확하게 분할하여 자율 주행 시스템이 안전하게 주행할 수 있도록 한다.
3. **위성 및 항공 이미지 분석**:
   - **토지 이용 및 피복 분류**: 위성 이미지를 분할하여 농지, 도시, 산림, 수역 등의 토지 이용 유형을 분류하고 분석한다.
   - **재난 모니터링**: 산불, 홍수, 지진 등 자연재해 발생 시 위성 이미지를 분할하여 피해 지역을 신속히 파악하고 대응할 수 있다.

8.4.3 사진 캡션 생성

8.4.3 멀티모달 처리와 RNN이 무엇인지 설명하시오.

멀티모달 처리

멀티모달 처리(Multimodal Processing)는 여러 유형의 데이터 소스(모달리티)를 동시에 처리하여 정보를 통합하고 분석하는 기술이다. 여기서 모달리티는 다양한 형태의 데이터를 의미하며, 예를 들어 텍스트, 이미지, 음성, 영상 등이 포함된다. 멀티모달 처리의 목표는 각 모달리티의 정보를 결합하여 더 풍부하고 정확한 결과를 도출하는 것이다.

멀티모달 처리의 주요 특징

1. **다양한 데이터 소스 통합**:
   - 서로 다른 데이터 소스의 정보를 결합하여 더 포괄적이고 의미 있는 분석을 가능하게 한다.
2. **상호 보완적 정보 제공**:
   - 한 모달리티에서 얻기 어려운 정보를 다른 모달리티가 제공할 수 있어, 더 높은 정확도와 신뢰성을 확보할 수 있다.
3. **다양한 응용 분야**:
   - 자연어 처리, 컴퓨터 비전, 음성 인식 등 여러 분야에서 활용되며, 특히 인간-컴퓨터 상호작용(HCI), 감정 인식, 멀티모달 번역 등에 사용된다.

예시: 이미지 캡션 생성

이미지 캡션 생성은 멀티모달 처리의 대표적인 응용 중 하나이다. 이는 이미지를 입력으로 받아, 해당 이미지에 대한 설명을 텍스트로 생성하는 작업이다. 이 과정에서는 이미지 정보와 텍스트 정보를 결합하여 의미 있는 설명을 생성한다.

RNN (Recurrent Neural Network)

RNN(Recurrent Neural Network)은 순환 신경망으로, 시퀀스 데이터를 처리하는 데 특화된 신경망 구조이다. RNN은 이전 단계의 출력을 다음 단계의 입력으로 사용하여, 시간적 순서가 있는 데이터를 효과적으로 모델링할 수 있다.

RNN의 주요 특징

1. **순환 구조**:
   - RNN은 순환 구조를 통해 이전 상태의 정보를 현재 상태로 전달함으로써, 시간적 연속성을 유지한다.
2. **시퀀스 데이터 처리**:
   - RNN은 시퀀스 데이터(예: 시계열 데이터, 자연어 텍스트 등)를 처리하는 데 적합하며, 입력 시퀀스의 각 요소에 대해 순차적으로 연산을 수행한다.
3. **메모리 기능**:
   - RNN은 이전 단계의 출력을 메모리에 저장하고, 이를 다음 단계의 연산에 활용함으로써, 과거 정보를 기억하고 활용할 수 있다.

RNN의 한계와 개선

1. **기울기 소실 문제**:
   - RNN은 긴 시퀀스를 처리할 때, 역전파 과정에서 기울기가 소실되거나 폭발하는 문제를 겪을 수 있다.
2. **LSTM(Long Short-Term Memory)와 GRU(Gated Recurrent Unit)**:
   - 기울기 소실 문제를 해결하기 위해 LSTM과 GRU와 같은 변형된 RNN 구조가 제안되었다. 이들은 게이트 메커니즘을 사용하여 중요한 정보를 더 오래 유지하고, 불필요한 정보를 제거한다.

예시: 언어 모델링

언어 모델링은 RNN의 대표적인 응용 중 하나이다. 이는 주어진 단어 시퀀스에서 다음 단어를 예측하는 작업으로, 텍스트 생성, 번역, 음성 인식 등 다양한 자연어 처리 작업에 사용된다.

결론

멀티모달 처리는 여러 모달리티의 데이터를 결합하여 더 풍부한 정보를 얻고 분석하는 기술이며, RNN은 시퀀스 데이터를 처리하는 데 특화된 신경망 구조이다. 이 두 가지 기술은 각각 다양한 응용 분야에서 중요한 역할을 하며, 복잡한 데이터 처리 및 분석을 가능

하게 한다.