



# Homework 7

| hw7-doc.pdf 과제보고서

| ID: 2021220699 NAME: 이은찬

## 간단한 분석

🔑 [simple\\_convnet.py](#)

- Class SimpleConvNet 코드
  - **Conv1→Relu1→Pooling1→Affine1→Relu2→Affine2→Softmax→출력**

그림 7-23 단순한 CNN의 네트워크 구성



- 아래와 같이 **Conv(콘볼루션)-Relu-Pool(풀링)-Affine(Fully-connected)-Relu-Affine-Softmax** 순으로 구현되어있음
  - **가중치(W,b) 뜻: ex) (W1,b1) = 첫번째 Conv의 가중치와 편향**

```
# 계층 생성
self.layers = OrderedDict()
self.layers['Conv1'] = Convolution(self.params['W1'], self.params['b1'],
                                   conv_param['stride'], conv_param['pad'])

self.layers['Relu1'] = Relu()
self.layers['Pool1'] = Pooling(pool_h=2, pool_w=2, stride=2)
self.layers['Affine1'] = Affine(self.params['W2'], self.params['b2'])
self.layers['Relu2'] = Relu()
self.layers['Affine2'] = Affine(self.params['W3'], self.params['b3'])

self.last_layer = SoftmaxWithLoss()
```

- Predict(예측)/loss(오차)/gradient(오차역전법)을 통해 **딥러닝**을 수행

- 파라미터 설명(*in class SimpleConvNet:*)

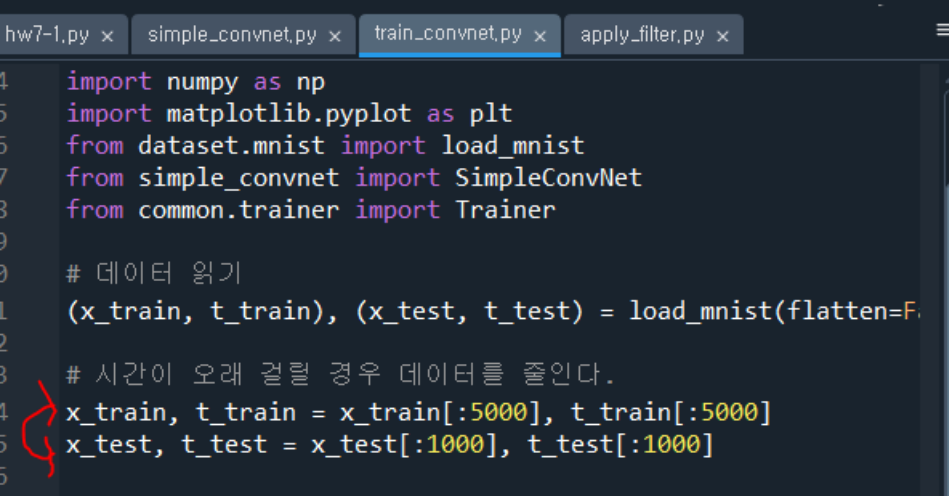
```
def __init__(self, input_dim=(1, 28, 28),
              conv_param={'filter_num':30, 'filter_size':5, 'pad':0, 'stride':1},
              hidden_size=100, output_size=10, weight_init_std=0.01):
```

초기화 때 받는 인수

- `input_dim` - 입력 데이터(채널 수, 높이, 너비)의 차원
- `conv_param` - 합성곱 계층의 하이퍼파라미터(딕셔너리). 딕셔너리의 키는 다음과 같다.
  - `filter_num` - 필터 수
  - `filter_size` - 필터 크기
  - `stride` - 스트라이드
  - `pad` - 패딩
- `hidden_size` - 은닉층(완전연결)의 뉴런 수
- `output_size` - 출력층(완전연결)의 뉴런 수
- `weight_init_std` - 초기화 때의 가중치 표준편차

## 🔑 train\_convnet.py

- SimpleConvNet을 MNIST를 통해 훈련시키는 코드
- **훈련 오래걸림** → 데이터가 많은데 쪼개서 연산하므로 모델 반복횟수가 크고 CNN연산의 복잡도 때문
  - data를 일부만 잘라서 하면 낫다



```
import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from simple_convnet import SimpleConvNet
from common.trainer import Trainer

# 데이터 읽기
(x_train, t_train), (x_test, t_test) = load_mnist(flatten=True,
                                                  return_labels=True,
                                                  shuffle=True,
                                                  verbose=1)

# 시간이 오래 걸릴 경우 데이터를 줄인다.
x_train, t_train = x_train[:5000], t_train[:5000]
x_test, t_test = x_test[:1000], t_test[:1000]
```

- → 과제 1)처럼 단순히 Simple ConvNet - 1 ConvNet+ 1 Fully-Connected Layer의 경우 좀더 빠를 듯

교재에 의하면 5000과 1000으로 데이터를 자르지 않고 모든 데이터를 CNN으로 훈련시키면 정확도는 아래처럼 나올 수 있다.

- SimpleConvNet + MNIST → 성능 훈련데이터:99.82%, 시험 데이터:98.96% (Very High)

그러나 이에는 대략 30분 이상이 지나도 훈련이 완료되지 않았고 데이터를 줄이는 코드를 사용해서 진행하였기에 정확도는 92~93%수준이다.

## 결과보고서

### HW 1) Simple ConvNet - 1 ConvNet+ 1 Fully-Connected Layer

Simple ConvNet - 1 conv + 1 fully connected layer을 구현해 보았습니다.

이는 교재의 CNN SimpleNet 코드를 조금 응용해서 아래 구조로 만들었습니다.

- Conv - Relu -Pool -Affine

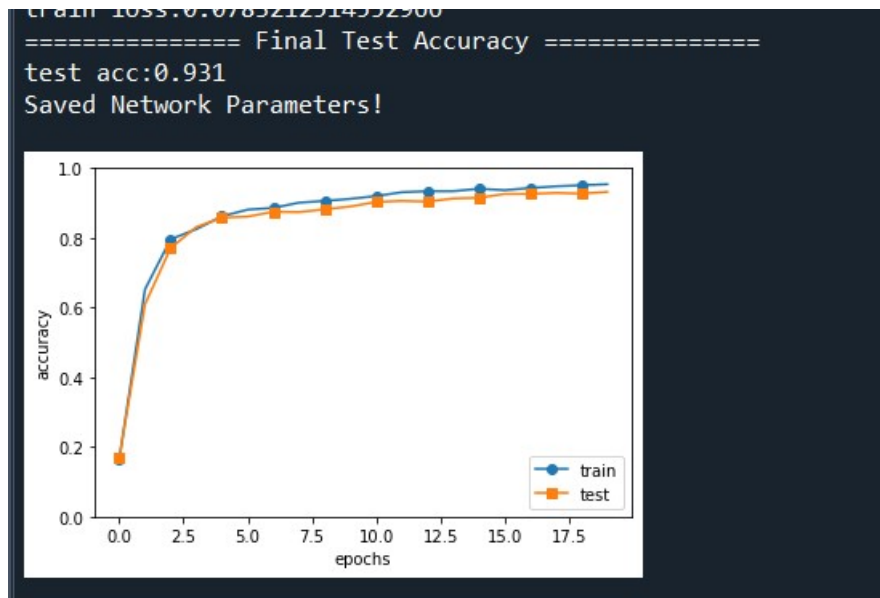
```
# 가중치 초기화
self.params = {}
self.params['W1'] = weight_init_std * \
    np.random.randn(filter_num, input_dim[0], filter_size, filter_size)
self.params['b1'] = np.zeros(filter_num)
self.params['W2'] = weight_init_std * \
    np.random.randn(pool_output_size, hidden_size)
self.params['b2'] = np.zeros(hidden_size)
self.params['W3'] = weight_init_std * \
    np.random.randn(hidden_size, output_size)
self.params['b3'] = np.zeros(output_size)

# 계층 생성
self.layers = OrderedDict()
self.layers['Conv1'] = Convolution(self.params['W1'], self.params['b1'],
                                   conv_param['stride'], conv_param['pad'])
self.layers['Relu1'] = Relu()
self.layers['Pool1'] = Pooling(pool_h=2, pool_w=2, stride=2)
self.layers['Affine1'] = Affine(self.params['W2'], self.params['b2'])
self.layers['Relu2'] = Relu()
```

## SimpleLayer Info

Aa 레 이어	≡ W SHAPE	≡ 특징
<u>Conv1</u>	(30, 1, 5, 5)	Convolution 연산을 im2col을 통해 수행하는 레이어
<u>Relu1</u>	-	Relu 연산을 활성화함수로 이용함
<u>Pool1</u>	-	Max Pooling을 취해서 중간 데이터 크기를 $h \rightarrow h/2$ , $w \rightarrow w/2$ 로 <b>축소</b>
<u>Affine1</u>	(4320, 10)	Full Connection Layer으로 Pool1 output 데이터의 차원을 10차원으로 줄임. 문제 조건이 하나의 Full Connection Layer만을 사용하므로 매우 극단적으로 차원을 축소시키고 있다.

## 결과 정확도



정확도 : **93.1 %**

CNN의 놀라운 성능으로 MNIST에 대해서 단순한 1겹 레이어 네트워크로도 93%의 정확도를 달성하여 굉장히 놀라웠다.

전체 MNIST 데이터를 훈련시켜서 결과를 확인한다면 95% 이상도 가능해보인다.

## HW2)

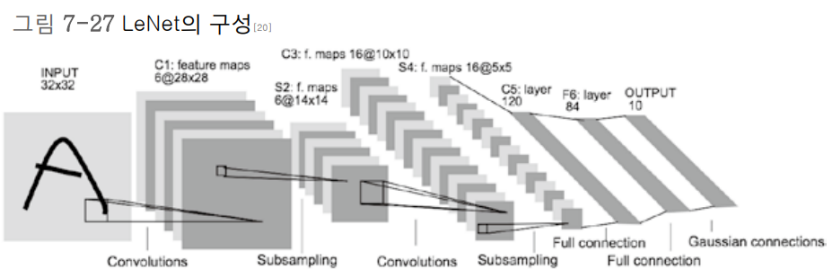
### LeNet 구현 -

모양만 유사하면 됨. 꼭 최적의 성능을 보이진 않아도 됨.

### Original LeNet Model 구조

LeNet Original Model의 구조는 아래와 같습니다. CNN이 2겹입니다.

**Conv1 → Pooling(Subsamp)1 → Conv2 → Pooling(Subsamp)2 → Affine1 → Affine2  
→ Softmax(=Gauss??)**



### LeNet 구조 분석

**합성곱 계층(=Convolution)과 풀링계층(=Subsampling, 원소를 줄이는 계층)을 반복하고 마지막으로 완전연결계층(=Affine, Fully Connection)을 거치면서 결과를 출력하는 것으로 생각했습니다.**

LeNet은 풀링 계층이 나오기 이전의 모델(1998)이므로 단순히 크기를 줄이는 Subsampling가 사용되었다. 따라서 Subsampling 부분을 맥스 풀링을 이용한 Pooling()으로 이용하는 것이 좋다고 생각했다.

그러나 Convolution Layer를 두겹 쌓아서 모델링을 돌려보니 행렬연산을 위한 dimention을 맞추는 것이 너무 어려웠다. 모델의 self 변수를 통해 행렬의 크기 관련 변수가 구성되기 때문에 데이터를 확인해가며 코딩하기도 쉽지가 않았습니다.

그래서 교재 7.5장에 나오는 예제 CNN 모델의 Filter\_num이 30으로 Original LeNet의 모델의 필터 사이즈의 6의 제곱과 비슷한 크기인 점에 영감을 받아서 Conv Layer를 두겹 쌓는 것에 실패한 점을 filter\_num을  $6 \times 6 = 36$ 으로 두어 유사한 성능을 가져가고자 하였습니다.

제가 설계한 모델은 아래와 같았으며 SimpleNet에 비해 성능이 꽤 좋았습니다.

## My LeNet

```

# 가중치 초기화
self.params = {}
self.params['W1'] = weight_init_std * \
    np.random.randn(filter_num, input_dim[0], filter_size, filter_size)
self.params['b1'] = np.zeros(filter_num)
self.params['W2'] = weight_init_std * \
    np.random.randn(pool_output_size, hidden_size)
self.params['b2'] = np.zeros(hidden_size)
self.params['W3'] = weight_init_std * \
    np.random.randn(hidden_size, output_size)
self.params['b3'] = np.zeros(output_size)
#self.params['W4'] = weight_init_std * \
#    np.random.randn(pool_output_size, input_dim[0], filter_size, filter_size)
#self.params['b4'] = np.zeros(filter_num2)

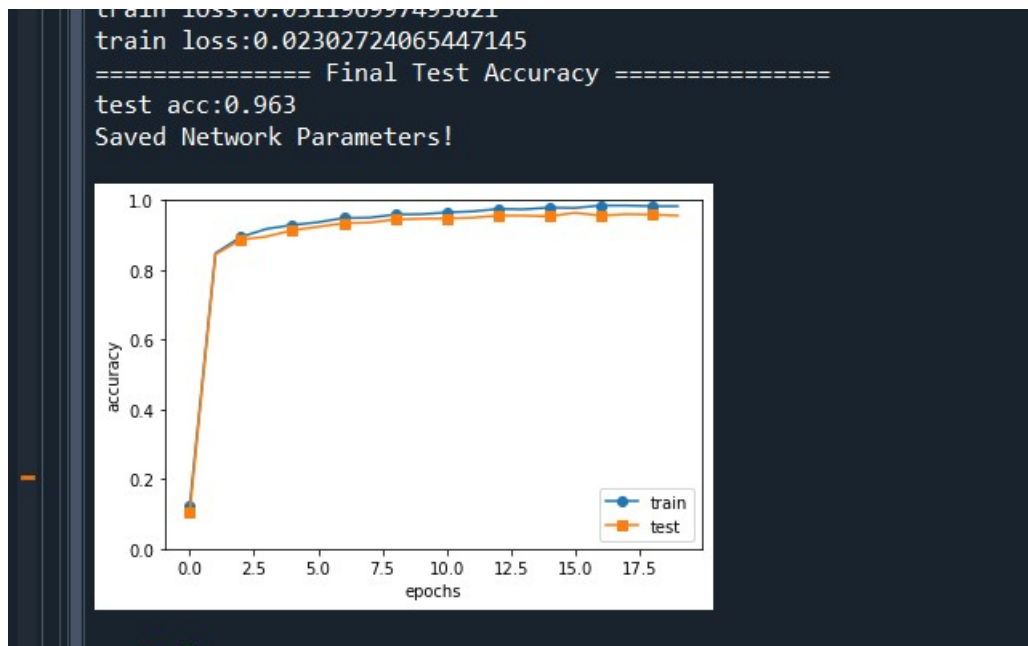
# 계층 생성
self.layers = OrderedDict()
self.layers['Conv1'] = Convolution(self.params['W1'], self.params['b1'],
    conv_param['stride'], conv_param['pad'])
#self.layers['Relu1'] = Relu()
self.layers['Pool1'] = Pooling(pool_h=2, pool_w=2, stride=2)
self.layers['Pool2'] = Pooling(pool_h=2, pool_w=2, stride=2)
#self.layers['Conv2'] = Convolution(self.params['W4'], self.params['b4'],
#    conv_param['stride'], conv_param['pad'])
self.layers['Affine1'] = Affine(self.params['W2'], self.params['b2'])
#self.layers['Relu2'] = Relu()
self.layers['Affine2'] = Affine(self.params['W3'], self.params['b3'])

```

## My LeNet Model Info

Aa 레이어	≡ Layer 형상	≡ 특징
<u>Conv1</u>	(36, 1, 5, 5)	FilterSize 기존 6 → 36으로 제곱증가하여 두개의 Conv 레이어 효과를 줌
<u>Pool1</u>	-	Max Pooling을 취해서 중간 데이터 크기를 $h \rightarrow h/2$ , $w \rightarrow w/2$ 로 <b>축소</b>
<u>Pool2</u>	-	Max Pooling을 취해서 중간 데이터 크기를 $h \rightarrow h/2$ , $w \rightarrow w/2$ 로 <b>한번 더 축소</b>
<u>Affine1</u>	(1188,120)	Full Connection Layer으로 Pool2 output 데이터의 차원을 120차원으로 줄임
<u>Affine2</u>	(120, 10)	Full Connection을 통해 120차원의 Affine1 output 데이터의 차원을 최종 출력의 차원인 10차원으로 축소
<u>Softmax</u>	-	Softmax를 통해서 최종 출력 값을 도출하고 훈련을 진행시킬 수 있음.

## 결과 정확도



정확도 : **96.3 %**

SimpleNet의 0.93보다 조금 더 높은 결과를 얻을 수 있었다.

2중 CNN을 묘사했기에 굉장히 높은 정확도를 보인다.

CNN 2중 구현을 위한 dimension을 맞춰서 더 좋은 성능을 이끌지 못했던 점이 조금 아쉬웠다.

전체 데이터를 훈련시켰다면 98%이상의 정확도가 가능할 것으로 추측되었다.