인공지능개론 6장(하)-7장(상) 정리노트#4

(꼭 PDF로 변환후 제출하기)

ICT 융합공학부 20학번 박준형 / ICT 융합공학부 20학번 김정호

6장

준형: 이번 6장에서는 우리가 계속해서 하고 있는 손실함수의 값을 최대로 작게하는 최적의 매개변수의 값을 찾기위해 여러 학습관련기술을 배웠어. 여기서 배운 기술들을 간단하게 설명해줘.

정호: 우선 우리가 배운 기술들은 확률적 경사 하강법(SGD), 모멘텀, AdaGrad, Adam 이 있어. 간단하게 설명해볼게.

SGD는 기울어진 방향으로 일정 거리만큼 이동하는 단순한 방법으로 구현이 쉽다는 장점이 있지만 비효율 적인 경우가 있어. 이를 개선해서 만들어진 방법이 모멘텀, AdaGrad, Adam 이야.

모멘텀은 물리학적 모델로써 물체로 비유해서 설명하자면 물체가 어떤 방향으로 움직이는 동안 그 방향으로 쌓은 운동량(momentum))을 가짐으로써 다음 단계에서도 그 방향으로 이동하는 방식이야. 이를 통해 조정된 방향으로 계속 이동시키는 거지. 이러한 방식으로 최적의 매개변수를 찾는 방법이야.

AdaGrad는 각각의 파라미터에 대해서 학습률을 동적으로 조절해주면서 보다 효율적인 학습을 진행할 수 있게 도와줘. 파라미터 별로 학습률을 개별적으로 설정한다는 것이 특징이야. 다만 계속 진행하다보면 학습률이 계속해서 작아지는 단점도 있어.

마지막으로 Adam은 모멘텀의 장점과 AdaGrad의 장점을 섞은 방식이야.

준형: 아하! 좀 이해가 되는 것 같아. 그러면 SGD말고 다른 방식으로만 학습을 하면 되는거 아니야?

정호: 그렇게 접근하면 안돼. 내가 설명해줄게. 예를 들어 모델이 단순한 구조와 낮은 계산 비용으로 구성이 되어 있다면 빠르게 초기 모델을 테스트하거나 간단한 문제를 해결할 때에느 SGD가 유용할 수 있어. 그 후 더 복잡하거나 보다 빠르고 안정적인 학습이 필요한 경우에는 3가지 방법 중 하나로 사용하는 것이 좋지. 따라서, 한가지 방법만을 고집하기보다는 다양한 방법을 시도하는 것이 가장 적합한 방법이야.

정호: 그리고 이런 신경망 학습에서 가장 중요한 것은 가중치의 초깃값이라고 할 수 있어. 우리가 이번 시간에 배운 것으 이야기해보자면 가중치를 감소시키면서 0이 되지않는 값으로 설정해야 보다 어느 한 쪽으로 치우치지 않는 결과를 얻을 수 있어, 그 예시로 Xavier, ReLU의 실제 사용되는 예시를 통해 확인할 수 있었지. 그래서 최종적으로 배치 정규화를 통해 학습 속도와 초깃값 설정, 오버피팅 등의 문제를 개선할 수 있었어. 여기서 나는 궁금한게 가중치가 제일 중요한 것은 알겠는데 그러면 또 가중치만큼 중요한 설정은 없는지 궁금해.

준형: 한번 내가 찾아봤는데 다른 하이퍼마라미터나 여러 기법도 중요하다고 하네. 근데 내가 생각했을 때는 하이퍼마라티 중 학습률이 제일 중요할 거라고 생각해. 그 이유는 학습률은 얼마나 빠르게 또는 천천히 학습하는지를 결정하는 파라미터로 너무 크거나 작으면 학습이 불안정하고 느려져서 최적의 가중치에 결코 도달할 수 없다고 생각해. 그리고 모든 과정에 영향을 미치기 때문에 학습률도 가중치 초깃값만큼 중요하다고 생각해.

7장(상)

정호: 7장은 CNN의 전체 구조를 배웠는데 가장 중요한 구성 요소는 합성곱 층으로 그림 설명이 나와 있는데 ppt에 나와 있는 그림과 함수에 대해 잘 이해가 안가는데 설명해줄 수 있어?

준형: 우선 CNN(Convolutional Neural Network)이란 이미지 데이터를 학습하고 인식하는데 특화된 알고리즘을 말해.

준형: Convolutional 의 의미는 신호처리 분야에서 사용되는 용어로 이미지 프로세싱에서 일정한 패턴으로 변환하기 위해 수행하는 행렬연산을 의미한대.

합성곱층에 대해 이해하기 위해서 완전연결층을 이해하는 것이 중요하다고 생각해. 간단히 설명하면 한 층의 모든 뉴런이 그 다음 층의 모든 뉴런과 연결된 상태를 말해. 1차원 배열의 형태로 행렬을 통해 이미지를 분류하는데 사용되는 계층이야. 하지만 이미지는 3차원 형태의 데이터로 완전연결층을 이용하려면 3차원의 데이터를 1차원으로 변환해야하는데 이 과정에서 데이터의 형상이 무시가 되어 정보의 손상을 가질 수 있어.

이러한 문제를 해결한 층이 바로 합성곱층이야. 합성곱층은 입력 데이터의 형상을 유지할 수 있어. 합성곱은 '하나의 함수와 또 다른 함수를 반전 이동한 값을 곱한 다음, 구간에 대해 적분하여 새로운 함수를 구하는 연산자이다' 라고 정의되어 있어. 그 힘수는 합성곱을 시각화하기 위한 식이라고 생각하면 돼.

정호: 음 아주 이해가 잘 됐어. 그러면 이러한 CNN의 한계점도 궁금해. 그리고 그 한계점에 대한 개선책도 있을까?

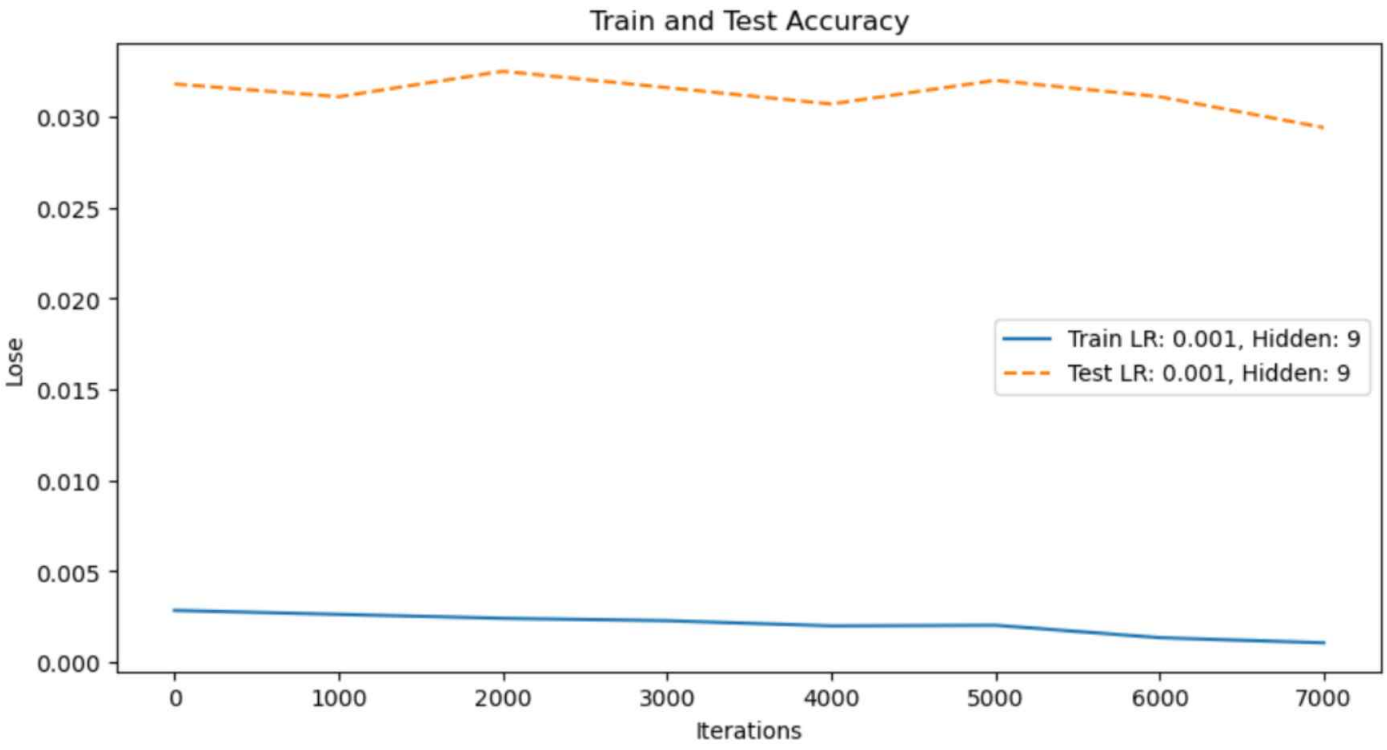
준형: 음 내가 한번 알아볼게. 우선 CNN은 많은 양의 데이터를 통해 학습해야 높은 성능을 가지는데 이러한 점 때문에 데이터가 부족하면 모델이 overfitting이 되기 쉬워 이러한 점은 데이터 증강 기법을 사용해 데이터를 인위적으로 늘리거나, 전이 학습을 활용하여 사전 학습된 모델을 사용하는 방법이 있다고 해. 그리고 많은 연산 자원을 필요로 하며 이미지의 회전, 확대, 축소 등 공간 변형에 민감할 수 있다고해 이러한 점을 개선하기 위해 모델 압축, 프루닝, 양자화 등을 통해 계산 비용을 줄이고 CapsNet 이라는 공간 변형에 강건한 구조도 있다고 해!!

```python
import sys
import os
sys.path.append(os.pardir)

import numpy as np
import matplotlib.pyplot as plt
from dataset.mnist import load_mnist
from multi_layer_net import MultiLayerNet
from common.optimizer import Momentum

# 필요한 레이어 및 유틸리티 함수 임포트
from common.layers import BatchNormalization

# 데이터 로드
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)

# 네트워크 생성
network = MultiLayerNet(input_size=784, hidden_size_list=[50, 30, 20], output_size=10,
                        activation='relu', weight_init_std='relu')

# 배치 정규화를 위한 레이어 추가
network.layers['BatchNorm1'] = BatchNormalization(50, beta=0.0)
network.layers['BatchNorm2'] = BatchNormalization(30, beta=0.0)
network.layers['BatchNorm3'] = BatchNormalization(20, beta=0.0)

# 하이퍼파라미터 설정
iters_num = 10000
train_size = x_train.shape[0]
batch_size = 100
```

```python
# 학습률 범위 설정
learning_rates = np.arange(0.001, 0.02, 0.001)

# 은닉층 크기 범위 설정
hidden_sizes = np.arange(5, 16)

# 모멘텀 파라미터 설정
momentum = 0.9

train_acc_list = []
test_acc_list = []

# 하이퍼파라미터 튜닝
for lr in learning_rates:
    for hidden_size in hidden_sizes:
        network.hidden_size_list = [hidden_size, hidden_size, hidden_size]
        optimizer = Momentum(lr, momentum)  # Momentum으로 변경

        train_acc = []
        test_acc = []

        for i in range(iters_num):
            batch_mask = np.random.choice(train_size, batch_size)
            x_batch = x_train[batch_mask]
            t_batch = t_train[batch_mask]

            # 순전파
            grads = network.gradient(x_batch, t_batch)
            optimizer.update(network.params, grads)

            # 정확도 계산
            if i % 1000 == 0:
                train_acc_val = network.accuracy(x_train, t_train)
                test_acc_val = network.accuracy(x_test, t_test)
                train_acc.append(1 - train_acc_val)
                test_acc.append(1 - test_acc_val)
```

```python
                print(f"Iteration: {i}, Train Acc: {train_acc_val}, Test Acc: {test_acc_val}")

                # 99% 정확도 달성 시 종료
                if train_acc_val >= 0.99 and test_acc_val >= 0.97:
                    train_acc_list.append((lr, hidden_size, train_acc))
                    test_acc_list.append((lr, hidden_size, test_acc))
                    print(f"Learning Rate: {lr}, Hidden Size: {hidden_size}")
                    break
        else:
            continue
        break
    else:
        continue
    break

# 결과 그래프로 출력
plt.figure(figsize=(10, 5))

# 하나의 그래프에 Train Accuracy와 Test Accuracy를 출력
for i, ((lr, hidden_size, train_acc), (_, _, test_acc)) in enumerate(zip(train_acc_list, test_acc_list)):
    plt.plot(range(0, len(train_acc) * 1000, 1000), train_acc, label=f"Train LR: {lr}, Hidden: {hidden_size}")
    plt.plot(range(0, len(test_acc) * 1000, 1000), test_acc, label=f"Test LR: {lr}, Hidden: {hidden_size}", linestyle='--')

plt.title('Train and Test Accuracy')
plt.xlabel('Iterations')
plt.ylabel('Lose')
plt.legend()

plt.show()
```

```
Iteration: 0, Train Acc: 0.15761666666666665, Test Acc: 0.1568
Iteration: 1000, Train Acc: 0.9301666666666667, Test Acc: 0.9245
Iteration: 2000, Train Acc: 0.9479166666666666, Test Acc: 0.9412
Iteration: 3000, Train Acc: 0.9580666666666666, Test Acc: 0.9482
Iteration: 4000, Train Acc: 0.9592166666666667, Test Acc: 0.9498
Iteration: 5000, Train Acc: 0.9664666666666667, Test Acc: 0.9554
Iteration: 6000, Train Acc: 0.9696333333333333, Test Acc: 0.9542
Iteration: 7000, Train Acc: 0.9716666666666667, Test Acc: 0.9557
Iteration: 8000, Train Acc: 0.9746833333333333, Test Acc: 0.9585
Iteration: 9000, Train Acc: 0.9772166666666666, Test Acc: 0.9628
Iteration: 0, Train Acc: 0.98035, Test Acc: 0.9626
Iteration: 1000, Train Acc: 0.9816666666666667, Test Acc: 0.9603
Iteration: 2000, Train Acc: 0.9832, Test Acc: 0.9634
Iteration: 3000, Train Acc: 0.9826833333333334, Test Acc: 0.9621
Iteration: 4000, Train Acc: 0.9844333333333334, Test Acc: 0.964
Iteration: 5000, Train Acc: 0.9872666666666666, Test Acc: 0.9642
Iteration: 6000, Train Acc: 0.9876166666666667, Test Acc: 0.9645
Iteration: 7000, Train Acc: 0.9888166666666667, Test Acc: 0.9646
Iteration: 8000, Train Acc: 0.98825, Test Acc: 0.9644
Iteration: 9000, Train Acc: 0.9887666666666667, Test Acc: 0.9661
Iteration: 0, Train Acc: 0.991, Test Acc: 0.9656
Iteration: 1000, Train Acc: 0.9915166666666667, Test Acc: 0.9674
Iteration: 2000, Train Acc: 0.9917, Test Acc: 0.9676
Iteration: 3000, Train Acc: 0.9931166666666666, Test Acc: 0.9666
Iteration: 4000, Train Acc: 0.9926, Test Acc: 0.9686
Iteration: 5000, Train Acc: 0.9921166666666666, Test Acc: 0.9684
Iteration: 6000, Train Acc: 0.9935666666666667, Test Acc: 0.9674
Iteration: 7000, Train Acc: 0.9940833333333333, Test Acc: 0.967
Iteration: 8000, Train Acc: 0.9939, Test Acc: 0.9679
Iteration: 9000, Train Acc: 0.99435, Test Acc: 0.9674

Iteration: 0, Train Acc: 0.9957166666666667, Test Acc: 0.9681
Iteration: 1000, Train Acc: 0.9962333333333333, Test Acc: 0.9689
Iteration: 2000, Train Acc: 0.9946166666666667, Test Acc: 0.9654
Iteration: 3000, Train Acc: 0.9963333333333333, Test Acc: 0.9674
Iteration: 4000, Train Acc: 0.9951666666666666, Test Acc: 0.9689
Iteration: 5000, Train Acc: 0.9962833333333333, Test Acc: 0.9687
Iteration: 6000, Train Acc: 0.9964666666666666, Test Acc: 0.9693
Iteration: 7000, Train Acc: 0.9965833333333334, Test Acc: 0.9693
Iteration: 8000, Train Acc: 0.9968333333333333, Test Acc: 0.9682
Iteration: 9000, Train Acc: 0.9964333333333333, Test Acc: 0.9671
Iteration: 0, Train Acc: 0.99715, Test Acc: 0.9682
Iteration: 1000, Train Acc: 0.9973666666666666, Test Acc: 0.9689
Iteration: 2000, Train Acc: 0.9975833333333334, Test Acc: 0.9675
Iteration: 3000, Train Acc: 0.9977166666666667, Test Acc: 0.9684
Iteration: 4000, Train Acc: 0.998, Test Acc: 0.9693
Iteration: 5000, Train Acc: 0.9979666666666667, Test Acc: 0.968
Iteration: 6000, Train Acc: 0.99865, Test Acc: 0.9689
Iteration: 7000, Train Acc: 0.9989333333333333, Test Acc: 0.9706
Learning Rate: 0.001, Hidden Size: 9
```



7.2.2 합성곱 연산

- **합성곱 연산**
  - 이미지 처리에서 말하는 필터 연산에 해당함

  - 입력 데이터에 필터를 적용 (필터를 커널이라 칭함)
  - 입력은 (4, 4), 필터는 (3, 3), 출력은 (2, 2) 가 됨
  - CNN 에서는 필터의 매개변수가 가중치임

정리노트

① 2+3+1+3+4+2=15

② 4+2+6+4=16

③ 2+2+2=6

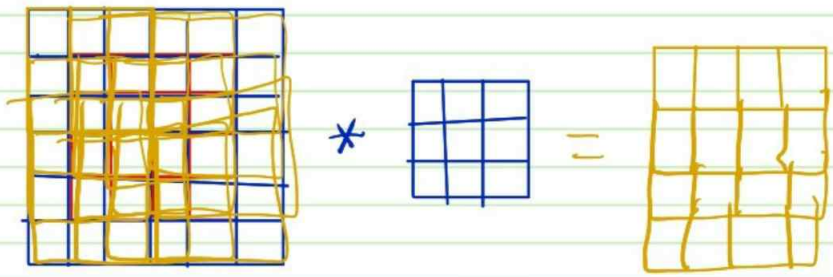
④ 2+3+1+3+4+2=15

위 4가지는 □ 와 필터를 곱하면

① ② / ③ ④ ⇒ 15 16 / 6 15 이다.

정리노트

예제 1)



∴ (4,4)