과제 11/29

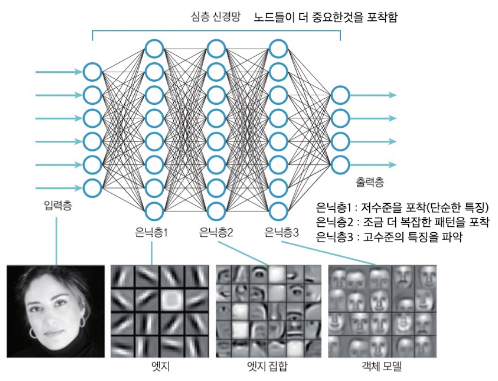
2315028 김성현

이론

▼ 딥러닝

- 심층신경망(Deep Neural Network, DNN)

  : 은닉층의 개수를 증가시켜 깊은 네트워크를 구성한 것



- 심층 신경망의 은닉층은 앞층에서 저수준의 특징들을 잘 포착하고, 층이 깊어질수록 점차 고수준의 특징들을 스스로 발견해나감
- 머신러닝 기법과 달리 모델 자체에서 특징을 자연스레 추출할 수 있음

▼ 문제

- 그래디언트 소멸 : 은닉층이 많아지면 출력층에서 계싼된 그래디언트가 역전파되면서 점점 값이 작아져 소멸되는 문제 발생
- 과적합 : 은닉층이 늘어나면 모델파라미터가 증가하며 훈련 데이터의 노이즈까지 학습할 가능성을 높이기에 과적합 문제 발생 (모델 복잡도와 비례함)

→ 손실함수가 가진 한계, 학습률 설정의 어려움, 연산량 및 메모리 요구량이 증가

▼ 해결

AlexNet : 심층신경망 기반 컴퓨터 비전 모델

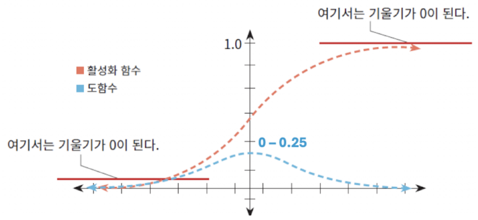
ReLu 사용 → 그래디언트 소멸문제 해결

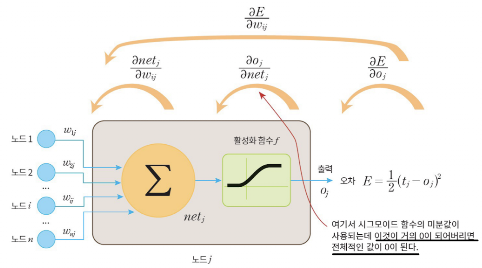
Dropout 사용 → 과적합 방지

GPU 사용 → 계산문제 해결. 대규모 데이터 학습 가능해짐

한계 및 극복

▼ 그래디언트 소멸 (Gradient Vanishing)



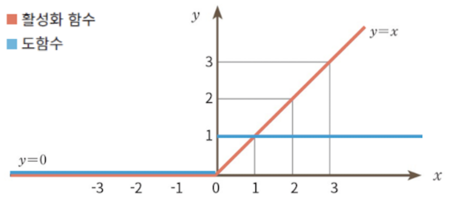


그래디언트 소멸 문제의 원인은 ‘ 시그모이드 함수 ‘

시그모이드 함수의 특성상 아주 큰 양수나 음수가 들어오면 출력이 포화되어 거의 0이 됨

→ 역전파 과정에서 시그모이드함수의 미분값이 거듭해 곱해지면 출력층과 멀어질수록 그래디언트 값이 계속 줄어드는 문제가 발생

- 해결방법 : ReLU 함수를 통해 그래디언트 소멸 극복



$$f(x)=\begin{cases} x & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$

ReLU는 입력값이 양수이면 입력값에 상관없이 항상 동일한 미분값인 1을 내놓기에 기울기소실문제 해결

단순히 임계값 0에 따라 출력값이 결정되기에 연산속도가 빠름

Leak ReLU는 입력값이 음수인 노드를 활용하기 어려운 ReLU의 문제를 해결해주기 위해 나옴

▼ 가중치 초기화 (Weight Initializatioin)

가중치 초기화가 성능에 중요한 영향을 미침

- 초기 가중치를 0으로 하면,
  모든 뉴런이 동일한 입력과 그래디언트를 계산해 학습시 뉴런간 차이가 발생하지 않음
- 가중치를 무작위로 설정하면,
  층이 깊어질수록 입력값의 분산이 급격히 커지거나 작아짐. 그래디언트 소멸 혹은 폭발문제가(→ 모델학습 불안정, 실패) 발생함
- 해결방법

  특정 입력값과 출력값의 분산을 동일하게한 상태로, 정규분포에서 난수를 추출해 가중치를 초기화하는 방법을 제안

  → 역전파과정에서 그래디언트의 크기가 안정적으로 유지됨

▼ 손실함수 MSE의 한계

$$E = \frac{1}{2}\sum_{i=1}^{n}(t-o)^2$$

손실함수로 평균오차제곱(MSE)을 사용

→ 그래디언트 값이 필연적으로 작아져 불필요한 계속되어 학습전체가 느려지는 저속수렴문제가 발생

ex_ 예측값과 실제값 사이의 작은 오차때문에 불필요한 학습이 계속 반복됨

▼ 해결방법 : 교차엔트로피 (Cross Entropy) 함수

$$H(p,q) = -\sum_x p(x)\log_n q(x)$$

$p(x)$:실제값(정답)의 확률분포

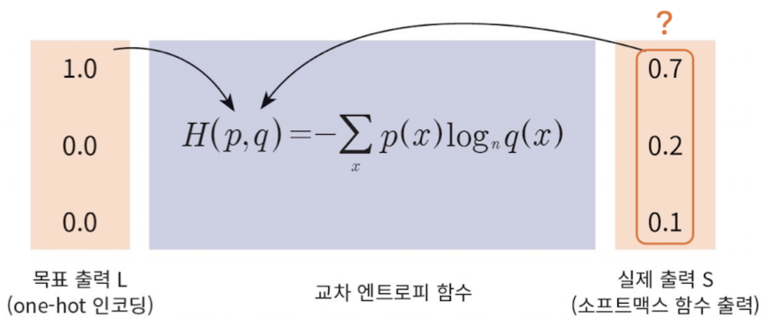
$q(x)$: 모델의 예측 확률 값

보통 n은 e임. 자연상수로그를 사용함

교차 엔트로피 : 두개의 확률분포 간의 차이를 측정하는데 사용함

신경망에서 실제값의 확률분포와 예측값의 확률분포가 얼마나 다른지를 나타냄

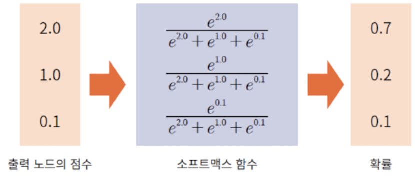
→ 예측이 정답에 가까울수록 낮아지고, 멀어질수록 커짐



$$\begin{aligned} H(p,q) &= -\sum_x p(x)\log_n q(x) \\ &= -\big(1.0 * \ln(0.7) + 0.0 * \ln(0.2) + 0.0 * \ln(0.1)\big) \\ &= 0.35667 \end{aligned}$$

얼마나 비슷한지에대한 값으로, 값이 작을수록 정답이 명확해짐. 오차가 적음

- 소프트맥스 (softmax) 함수



$$\mathrm{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

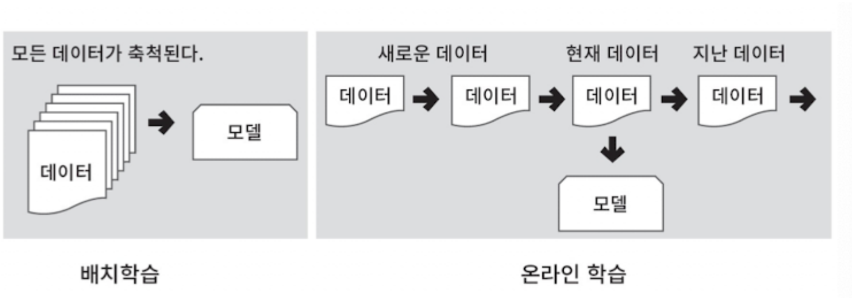
모든 수는 0~1 사이의 값이 됨. 합은 1이 됨

소프트맥스 함수 : 입력값을 확률분포로 변환하는 함수. 입력값들의 합을 **1**로 정규화해 각 값이 확률로 해석될 수 있도록 변환함

출력층 노드에 소프트맥스 함수를 적용함

▼ 미니 배치 (Mini-Batch)

모델 가중치 업데이트 방법



- 배치학습

  : 모든 샘플을 살펴본 후 개별 샘플의 그래디언트를 더해서 이를 바탕으로 가중치를 업데이트하는 방법. 배치 경사하강법을 사용함

  특징) 전체 학습데이터를 전부 처리해 가중치를 업데이트하면 모델 업데이트에 오랜 시간이 걸릴 수 있고, 메모리에 해당 데이터가 올라가지 않을 수 있음

- 온라인 학습

  : 하나의 샘플이 주어질때마다 즉석에서 바로 오차를 계산해 가중치를 업데이트하는 방법

  온라인학습 방법을 적용한 경사하강법이 '확률적 경사하강법'

  특징) 각각의 샘플마다 업데이트를 진행하면 노이즈에 취약해지고, 업데이트를 위한 계산이 많아지게됨

- 미니배치

  : 온라인 학습과 배치학습의 중간. 학습데이터를 작은 배치로 분리시켜 하나의 배치가 끝날때마다 학습을 수행하는 방법

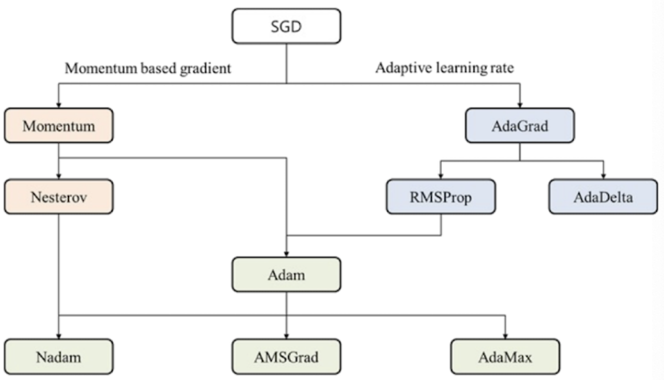
  `1 < 미니배치 < 훈련 데이터`

특징) 빠른 모델 업데이트, 메모리효율성, 정확한 모델 업데이트, 계산 효율성 장점들을 모두 지니게 됨

- ex_ 학습데이터 : 32000개

  배치학습 - 한번에 32000개를 학습 / 온라인학습 - 1개씩 학습 / 미니 배치 - 한번에 32개씩 학습

▼ 향상된 최적화 기법



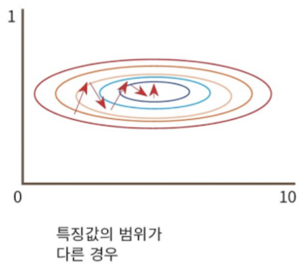
여러 기법을 통해 학습속도를 높이고, 학습률을 쉽게 자동으로 조정할 수 있는 최적화 기법들이 나옴

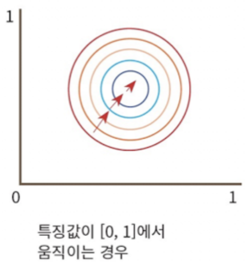
▼ 데이터 정규화 (Data Normalization)

큰 값이 그래디언트 갱신을 주도하게되는 문제를 해결하기위해 등장

모든 입력을 정규화함으로써(같은 범위의 값으로 만듬) 신경망이 각 입력노드에대한 최적의 매개변수를 보다 빨리 습득하게 함

- 최소최대 정규화 : 0~1, -1 ~1 사이의 값으로 범위를 설정
- 가우시안 정규화 : 평균이 0이고 표준편차가 1인 정규분포를 따르는 z값을 사용





가우시안 정규화

$$Z = \frac{X - \mu}{\sigma}$$

μ: 데이터 평균
σ: 데이터 표준편차

▼ 과적합 방지

과적합 : 지나치게 학습 데이터에 특화되어 다른 데이터에 대해 좋지않은 결과가 나옴

- 조기종료
- 가중치 규제

: 가중치에 규제를 가해 영향력을 조절하는 기법 (선을 규제)

손실값 ← 기존 손실값 + 가중치 규제항

- L1 규제 : 중요한 특징을 남기고, 영향력 낮은 특징은 아예 제외시키는 방법
- L2 규제 : 모든 변수가 일정한 영향을 유지하도록하여, 모델을 더 일반화되도록 만듬

→ 중요한 정보에만 집중하도록 만듬

- 데이터 증강

  원본 데이터에 다양한 변형을 가해 새로운 데이터를 생성함으로써, 모델이 더 다양한 상황에 적응하도록 도와줌 (모델의 일반화 성능 향상)

- 드롭아웃

  : 학습시 몇개의 **노드**를 랜덤하게 제외하는 방법 (노드를 규제)

  과적합은 노드가 너무 많아도 발생하기 쉽기에..

실습

▼ 인공신경망

▼ 인공신경망

- 인공 신경망은 퍼셉트론이라는 기본 개념을 바탕으로 설계됨
- 퍼셉트론은 생물학적 뉴런을 논리적으로 표현한 것
- 뉴런은 다른 뉴런으로부터 여러 입력을 받아 이를 처리한 후 "연결된" 다른 뉴런에 결과를 전달함

- 대부분의 프로그래밍 언어에서 부동소수점 연산 시 정밀도 문제가 존재함. 아주 작은 수를 계산하거나 비교할 때 정확한 값이 아니라 근사값을 처리함. 이 때문에 예상치 못한 오류가 발생할 수 있음. 이에 대한 가장 간단한 방법은 0에 가까운 충분히 작은 숫자를 대신 사용

▼ 퍼셉트론 학습

```
def step_func(t): #퍼셉트론의 활성함수인 계단 함수 (0 or 1을 반환)
    if t > 0.0000001:
        return 1
    else:
        return 0
```

활성화함수 : 계단함수

```python
# 퍼셉트론 학습 함수
def perceptron_fit(X, Y, epochs=50, learning_rate=0.2):
    weights = np.zeros(len(X[0]))  # 가중치 초기화
    bias = 0  # 바이어스 초기화

    for t in range(epochs):  # 지정된 횟수만큼 반복
        for i in range(len(X)):  # 각 데이터 포인트에 대해 반복
            predict = step_func(np.dot(X[i], weights) + bias)  # 현재 가중치로 예측값 계산
            error = Y[i] - predict  # 오차 계산
            weights += learning_rate * error * X[i]  # 가중치 업데이트
            bias += learning_rate * error
    return weights, bias  # 학습된 가중치 반환

# 퍼셉트론 예측 함수
def perceptron_predict(X, weights, bias):  # 학습된 가중치를 입력받아 예측
    for i in range(len(X)):
        result = step_func(np.dot(X[i], weights) + bias)
        print(f"입력:{X[i]}, 정답:{y[i]}, 예측:{result}")

# 퍼셉트론 학습 및 예측 실행
weights, bias = perceptron_fit(X, y)  # 학습 실행
print("학습된 가중치:", weights)
print("학습된 편향:", bias)

perceptron_predict(X, weights, bias)  # 예측 실행
```

✓ 0.0s

```
학습된 가중치: [0.4 0.2]
학습된 편향: -0.4
입력:[0 0], 정답:0, 예측:0
입력:[0 1], 정답:0, 예측:0
입력:[1 0], 정답:0, 예측:0
입력:[1 1], 정답:1, 예측:1
```

▼ 다층 퍼셉트론

- 은닉층(hidden layer)를 넣고, 각종 sigmoid 등 활성화 함수를 사용해 복잡한 비선형 문제 해결

- 본 예제에서는 활성화 함수로 sigmoid를 사용

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

```python
# 시그모이드 함수
def sigmoid_func(x):
    return 1 / (1 + np.exp(-x))

# 시그모이드 함수의 미분치, 시그모이드 함수 출력값을 입력으로 받는다.
def sigmoid_deriv(out):
        return out * (1 - out)
```

활성화함수 : 시그모이드 함수

```python
# 입력유닛의 개수, 은닉유닛의 개수, 출력유닛의 개수
inputs, hiddens, outputs = 2, 2, 1
learning_rate=0.2

def MLP_fit(X, T, epochs=10000, learning_rate=0.2):
    # 가중치 및 바이어스 초기화
    W1 = np.random.uniform(-0.1, 0.1, (inputs, hiddens))  # inputs * hiddens 크기 가중치 행렬
    W2 = np.random.uniform(-0.1, 0.1, (hiddens, outputs))  # hiddens * outputs 크기 가중치 행렬
    B1 = np.random.uniform(-0.1, 0.1, hiddens)  # 은닉층(hiddens) 바이어스
    B2 = np.random.uniform(-0.1, 0.1, outputs)  # 출력층(outputs) 바이어스

    for epoch in range(epochs):  # 지정된 에포크 수만큼 반복
        for x, y in zip(X, T):  # 학습 데이터 순차 처리
            x = np.reshape(x, (1, -1))  # 2차원 행렬로 변환
            y = np.reshape(y, (1, -1))  # 2차원 행렬로 변환

            # 순방향 전파
            layer0 = x  # 입력층
            Z1 = np.dot(layer0, W1) + B1  # 은닉층의 선형 결합
            layer1 = sigmoid_func(Z1)  # 은닉층 활성화 함수 적용
            Z2 = np.dot(layer1, W2) + B2  # 출력층의 선형 결합
            layer2 = sigmoid_func(Z2)  # 출력층 활성화 함수 적용

            # 역방향 전파
            layer2_error = layer2 - y  # 출력층 오차 계산
            layer2_delta = layer2_error * sigmoid_deriv(layer2)  # 출력층 델타 계산
            layer1_error = np.dot(layer2_delta, W2.T)  # 은닉층 오차 계산
            layer1_delta = layer1_error * sigmoid_deriv(layer1)  # 은닉층 델타 계산

            # 가중치 및 바이어스 업데이트
            W2 -= learning_rate * np.dot(layer1.T, layer2_delta)
            W1 -= learning_rate * np.dot(layer0.T, layer1_delta)
            B2 -= learning_rate * np.sum(layer2_delta, axis=0)
            B1 -= learning_rate * np.sum(layer1_delta, axis=0)

    return W1, W2, B1, B2  # 학습된 가중치와 바이어스 반환
```

다층 퍼셉트론

```python
def test(X, T, W1, W2, B1, B2):
    for x, y in zip(X, T):
        x = np.reshape(x, (1, -1))  # 2차원 행렬로 변환
        # 순방향 전파만 수행
        layer0 = x  # 입력층
        Z1 = np.dot(layer0, W1) + B1  # 은닉층의 선형 결합
        layer1 = sigmoid_func(Z1)  # 은닉층 활성화 함수 적용
        Z2 = np.dot(layer1, W2) + B2  # 출력층의 선형 결합
        layer2 = sigmoid_func(Z2)  # 출력층 활성화 함수 적용
        print(f"입력: {x}, 정답: {y}, 예측: {layer2}")

W1, W2, B1, B2 = MLP_fit(X, T, epochs=10000, learning_rate=0.2)
test(X, T, W1, W2, B1, B2)
```

✓ 0.8s

```
입력: [[0 0]], 정답: 0, 예측: [[0.04250626]]
입력: [[0 1]], 정답: 1, 예측: [[0.95236738]]
입력: [[1 0]], 정답: 1, 예측: [[0.95224611]]
입력: [[1 1]], 정답: 0, 예측: [[0.06101696]]
```

출력