퍼셉트론 보고서

제출일: 25.06.25
제출인: 서윤철

<순서>

- 코드
- 특징
- 비교
- 분석
- 요약
- 느낀점

코드

-MLP_code_서윤철(S_code)-

```
def sigmoid(x):
return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
return x * (1 - x)

# XOR 데이터
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

# 시드 고정
np.random.seed(42)

# 신경망 구조
input_size = 2
hidden_size = 4
output_size = 1
lr = 0.1
epochs = 10000

# 가중치 및 바이어스 초기화
w1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
w2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

loss_history = []

# 학습
for epoch in range(epochs):
# 순전파
z1 = np.dot(X, w1) + b1
a1 = sigmoid(z1)
z2 = np.dot(a1, w2) + b2
y_pred = sigmoid(z2)

# 손실 계산
loss = np.mean((y - y_pred)**2)
loss_history.append(loss)

# 역전파
d_loss_y = 2 * (y_pred - y)
d_output = d_loss_y * sigmoid_derivative(y_pred)
d_hidden = np.dot(d_output, w2.T) * sigmoid_derivative(a1)

# 가중치 업데이트
w2 -= lr * np.dot(a1.T, d_output)
b2 -= lr * np.sum(d_output, axis=0, keepdims=True)
w1 -= lr * np.dot(X.T, d_hidden)
b1 -= lr * np.sum(d_hidden, axis=0, keepdims=True)
```

-MLP_code_김병현(K_code)-

```
class MLP: # 다층 퍼셉트론
def __init__(self, input_size, hidden_size, output_size, lr=0.05, epochs=50000):
self.lr = lr
self.epochs = epochs
self.errors = []

# 가중치 초기화: 작은 난수
self.weights_input_hidden = np.random.randn(input_size, hidden_size)
self.bias_hidden = np.zeros((1, hidden_size))

self.weights_hidden_output = np.random.randn(hidden_size, output_size)
self.bias_output = np.zeros((1, output_size))

def sigmoid(self, x): # sigmoid 함수 (비선형 이진 분류 활성화 함수) : 값이 0.5 이상일 경우, 1로 판단.
x = np.clip(x, -500, 500) # overflow 방지
return 1 / (1 + np.exp(-x)) # hidden layer를 여러개 쌓아도 활성화 함수가 선형이면 비선형 연산은 불가능 (XOR 등)

def sigmoid_derivative(self, x): # sigmoid 기울기 계산 (미분 값)
return x * (1 - x)

def predict(self, x):
# 1. 입력층 -> 은닉층 계산
x = x.reshape(1, -1) # 입력 x를 (1, input_size) 형태의 2차원 배열로 변환 (행렬 연산 준비)
hidden_input = np.dot(x, self.weights_input_hidden) + self.bias_hidden # 입력 x와 input, hidden 가중치를 행렬 곱 한 후, hidden 편향을 더함.
hidden_output = self.sigmoid(hidden_input)

# 2. 은닉층 -> 출력층 계산
output_input = np.dot(hidden_output, self.weights_hidden_output) + self.bias_output
output = self.sigmoid(output_input)
return output[0, 0] # 스칼라 반환

def train(self, X, y):
y = y.reshape(-1, 1) # (4, 1) 형태로

for epoch in range(self.epochs):
total_error = 0
for xi, target in zip(X, y):
xi = xi.reshape(1, -1) # (1, 2)
target = target.reshape(1, -1) # (1, 1)

# 순방향 전파 : 각 층의 입력 및 출력 값이 저장.
# hidden layer
hidden_input = np.dot(xi, self.weights_input_hidden) + self.bias_hidden # input 가중 합 연산
hidden_output = self.sigmoid(hidden_input) # 활성화 함수인 sigmoid 함수를 통과하여 hidden layer의 출력 생성.

output_input = np.dot(hidden_output, self.weights_hidden_output) + self.bias_output # hidden 가중 합 연산
output = self.sigmoid(output_input)

# 오류 계산 : MSE를 사용하여 예측과 정답 사이의 오차를 계산.
error = target - output
total_error += np.sum(error ** 2) # Loss funcion으로 Mean Square Error 사용. 학습이 진행될수록 이 값을 줄이는 것이 목표.

# 역방향 전파
delta_output = error * self.sigmoid_derivative(output) # 출력층의 오류 기울기 계산
delta_hidden = np.dot(delta_output, self.weights_hidden_output.T) * self.sigmoid_derivative(hidden_output) # 은닉층의 오류 기울기 계산

# 가중치 & 편향 업데이트 : 개선된 기울기를 사용하여 업데이트
self.weights_hidden_output += self.lr * np.dot(hidden_output.T, delta_output)
self.bias_output += self.lr * delta_output

self.weights_input_hidden += self.lr * np.dot(xi.T, delta_hidden)
self.bias_hidden += self.lr * delta_hidden

self.errors.append(total_error)
if (epoch + 1) % 1000 == 0:
print(f"Epoch {epoch+1}/{self.epochs} - Total Error: {total_error:.6f}")

# XOR 데이터
x_xor = np.array([[0,0], [0,1], [1,0], [1,1]])
y_xor = np.array([0,1,1,0])

# MLP 모델 생성 및 학습
mlp = MLP(input_size=2, hidden_size=6, output_size=1, lr=0.1, epochs=10000)
mlp.train(x_xor, y_xor)
```

특징

두 코드 모두 MLP의 기본적인 구성 요소(입력층, 은닉층, 출력층)와 학습 과정(순전파, 손실 계산, 역전파, 가중치 업데이트)을 구현하고 있습니다.

- 활성화 함수: 두 구현 모두 은닉층과 출력층에 sigmoid 활성화 함수를 사용합니다. K_code에서는 sigmoid 함수에 대해 "비선형 이진 분류 활성화 함수"이며, "값이 0.5 이상일 경우, 1로 판단"한다고 설명하고 있습니다. 또한 "hidden layer를 여러개 쌓아도 활성화 함수가 선형이면 비선형 연산은 불가능 (XOR 등)"이라는 중요한 점을 언급하여, XOR 문제 해결에 비선형 활성화 함수가 필수적임을 강조합니다.
- 손실 함수: 두 코드 모두 손실 함수로 평균 제곱 오차(MSE)를 사용합니다. K_code에서는 이를 "Loss function으로 Mean Square Error 사용. 학습이 진행될수록 이 값을 줄이는 것이 목표."라고 명시하고 있습니다.
- 학습: 경사 하강법(Gradient Descent)을 기반으로 가중치와 편향을 업데이트합니다. 학습률(lr)을 통해 업데이트 보폭을 조절합니다.
- XOR 문제: 두 코드 모두 비선형 분류 문제의 대표적인 예시인 XOR 데이터를 사용하여 모델을 학습시킵니다.

- S_code

- 함수 기반 구현: class 구조 없이 함수들을 사용하여 MLP의 순전파 및 역전파 로직을 구현합니다.
- 배치 학습: 전체 학습 데이터(X)를 한 번에 입력하여 가중치를 업데이트하는 배치 학습 방식을 사용합니다. 손실 계산과 역전파 과정에서 np.mean이나 np.sum을 사용하여 전체 배치에 대한 연산을 수행합니다.
- 변수명: w1, b1, w2, b2와 같이 간결한 변수명을 사용합니다.
- 손실 기록: loss_history 리스트에 epoch별 손실 값을 기록하여 학습 진행 상황을 추적합니다.

- K_code

- 객체 지향 구현: MLP 클래스를 정의하여 MLP 모델을 캡슐화합니다. 생성자(__init__), 예측(predict), 학습(train) method를 포함합니다.
- 샘플별 학습: train 내부에서 for xi, target in zip(X, y) 루프를 통해 각 학습 샘플(xi)에 대해 순전파, 역전파, 가중치 업데이트를 수행합니다. 이는 배치 학습과 구분되는 중요한 차이점입니다.
- 데이터 형태 변환: predict 및 train 메서드 내에서 입력 데이터 x나 xi, target의 형태를 다시 만들어 행렬 연산에 적합하게 만듭니다. K_code는 x = x.reshape(1, -1)에 대해 "입력 x를 (1, input_size) 형태의 2차원 배열로 변환 (행렬 연산 준비)"라고 주석을 달아 놓았습니다.
- 오류 기록: self.errors 리스트에 epoch별 total_error를 기록합니다.
- 오버플로우 방지: sigmoid 함수 내부에 x = np.clip(x, -500, 500) 코드를 추가하여 np.exp 계산 시 발생할 수 있는 오버플로우를 방지합니다.

비교

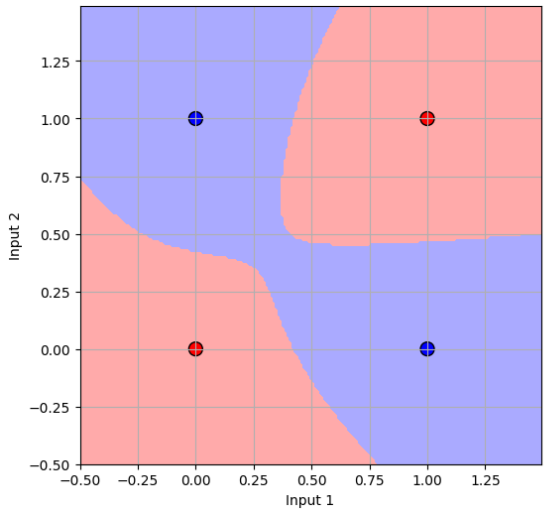
- 공통점

두 코드는 MLP의 기본 아키텍처, 활성화 함수(sigmoid), 손실 함수(MSE), 가중치 업데이트 방식(경사 하강법)을 공유합니다. Epoch 값은 S_code = 10000, K_code = 10000으로 동일합니다. 학습률 면에서도 S_code은 0.1, K_code은 lr = 0.1로 동일합니다.

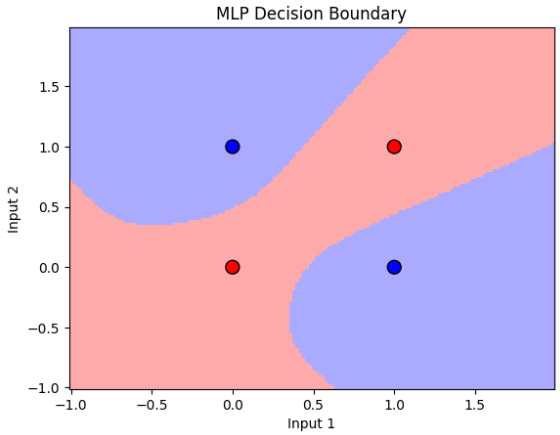
- 차이점

- 학습 방식: S_code는 배치 학습을, K_code는 샘플별 학습을 수행합니다. 이는 가중치 업데이트가 이루어지는 빈도와 방식에 큰 차이를 만듭니다.
- 코드 구조: S_code은 함수 기반, K_code은 Class 기반으로 구현되었습니다.
- Seed: 난수의 Seed를 S_code은 42로 고정했지만, K_code는 고정하지 않았습니다.
- Hidden layer: S_code은 hidden_size = 4, K_code은 hidden_size = 6으로 설정했습니다.
- 오버플로우 처리: K_code에만 sigmoid 함수 내 오버플로우 방지로직이 있습니다.

- 시뮬레이션 차이



<S_code>



<K_code>

cf) 결과 차이 원인

원인	설명
1. 학습 방식 차이 (배치 vs 샘플별)	`서윤철` 은 **배치 학습**으로 안정된 경계 형성, `김병현` 은 **샘플별 업데이트**로 경계가 더 **불안정하거나 비대칭**일 수 있음
2. 순전파/역전파 처리 범위	`서윤철` 은 전체 데이터로 평균적인 경계를 학습, `김병현` 은 샘플마다 바뀌는 가중치로 인해 **결정 경계의 왜곡**이 생길 수 있음
3. 가중치 초기화와 시드 사용	`서윤철` 은 `np.random.seed(42)` 로 시드 고정 → 결과 재현 가능. `김병현` 은 고정 시드 없음 → 실행마다 결정 경계가 바뀜
4. 학습 완료 후 사용된 모델의 시점	`서윤철` 은 **마지막 epoch 결과**로 시각화, `김병현` 도 마찬가지지만 오차 편차가 클 경우 **수렴 상태가 불완전할 수 있음**
5. 예측 함수 차이	`서윤철` 은 `mlp_predict(grid)` 로 **0.5 기준 이진화**. `김병현` 은 `predict()` 가 스칼라 값 출력만 하므로, 시각화를 하려면 별도 처리 필요
6. 히든 사이즈 차이	`서윤철` 은 `hidden_size=4` , `김병현` 은 `hidden_size=6` → 더 복잡한 곡선 가능하지만 과적합 또는 불안정성 초래 가능

분석

- 성능 및 사용성 비교

항목	`MLP_code_서윤철`	`MLP_code_김병현`
학습 안정성	평균 오차 기준으로 안정적인 수렴	샘플 단위 처리로 오차 편차 가능성 있음
재사용성	낮음 (확장 시 불편)	높음 (다른 문제에도 쉽게 적용 가능)
실험 편의성	시각화 포함으로 실험 용이	구조화된 코드, 실험 자동화에 적합
초심자 학습용	직관적이고 간단	객체지향 학습 연습에 좋음
대규모 데이터 대응력	부족	개선 여지 있음 (배치 처리 확장 가능)

- 요약

상황	추천 코드
XOR 같은 단순한 문제 빠른 실험	✅ `MLP_code_서윤철`
MLP 구조 학습, 객체지향 설계 연습	✅ `MLP_code_김병현`
다양한 문제에 재사용, 확장 계획	✅ `MLP_code_김병현`
즉시 결과 시각화 및 확인	✅ `MLP_code_서윤철`

느낀점

기존에 갖고 있던 MLP 결과와 비교하여, 같은 활성 함수(Sigmoid)를 사용하였는데 시뮬레이션 결과에 있어 차이가 발생한다는 점에서 조금 놀랐습니다. 같은 활성 함수를 사용하더라도 어떤 코드 구조를 택했느냐에 따라 다른 결과가 도출될 수 있다는 사실을 깨달았습니다.