IRIS, MNIST의 다차원 분류를 위한 MLP from scratch 구현 및 모델 성능 향상을 위한 노력

휴먼지능정보공학과
201910830 이환수

1. Abstract

IRIS 와 MNIST 데이터의 다차원 분류를 하기 위해 라이브러리 모델이 아니라, 직접 MLP 모델을 구현한다. 해당 구현한 모델의 성능을 라이브러리 모델과 비교하여 비슷하거나 우수한 성능을 확인하였다. 해당 모델을 개선하기 위해 데이터의 표준화나 정규화 같은 데이터의 전처리 뿐 아니라 Mini-batch 학습을 구현하였고, 두 모델의 손실의 변화를 시각화 하였다. 성능 비교나 시각화로 구현한 방법들의 효과를 확인할 수 있었다. 현재 과제에서 구현한 방법들 뿐 아니라 여러 다른 개선 방안들 또한 생각해보았다.

2. Introduction

딥러닝 수업의 중간고사 대체 과제로 다층 퍼셉트론을 구현한다. 다층 퍼셉트론은 라이브러리에 있는 모델을 사용하지 않고 순전파와 역전파를 직접 구현하여 가중치를 학습한다. 구현한 다층 퍼셉트론을 이용하여 IRIS와 MNIST데이터셋을 분류하는 다차원 분류 모델을 생성하고, 해당 모델의 성능을 라이브러리의 모델과 비교한다. 라이브러리 모델과 성능 차이를 확인하며 구현한 모델의 성능을 향상 시키기 위한 여러 방법들을 수행한다.

IRIS데이터는 붓꽃의 3가지 종에 대한 데이터이다. 꽃받침의 길이와 너비, 꽃잎의 길이와 너비 데이터와 함께 부채붓꽃, 버지니카, 버시칼라 종의 구분이 되어있다. 데이터 분석과 모델링을 통해 붓꽃을 분류하기 위해 사용되는 데이터셋으로 이번 과제에서 MLP를 이용해 분류해본다.

MNIST 데이터는 28x28 픽셀 크기에 10진수 0~255인 8bit의 밝기 값 데이터가 담겨 있는 손글씨 이미지 데이터이다. 해당 이미지는 0~9까지의 숫자로 분류할 수 있는 숫자 이미지로 iris 데이터와 마찬가지로 분류에 사용되는 데이터셋이다. MNIST 데이터 또한 구현한 MLP로 분류를 진행한다.

모델의 성능은 Accuracy와 각 라벨의 개수에 비례한 가중치를 이용한 Weighted F1 Score를 통해 측정하였고 K-Fold 기법을 통해 교차검증을 하였다. 또한 모델의 올바른 예측 여부를 판단하기위해 혼동행렬 또한 확인해보았다.

성능을 측정하고 구현한 모델의 성능을 향상 시키기 위해 입력 데이터의 표준화나 정규화와 같은 전처리를 진행하거나, 미니 배치 학습을 구현하고 학습을 수행해보았다.

3. Background

다층 퍼셉트론을 이해하기 전 단층 퍼셉트론을 먼저 이해 해본다. 퍼셉트론은 다수의 입력을 하나의 결과로 나타나게 하는 알고리즘이고, 이는 뇌의 신경 세포 뉴런의 동작과 유사하여 인공 신경망으로 불린다. 각각의 입력 값에는 가중치가 존재하고, 입력 값과 그 입력 값에 해당하는 가중치의 곱들의 합이 입력 값으로써 새로운 뉴런에 보내진다. 뉴런에서는 받은 입력값에 따른 출력 신호로 변환한다. 이렇게 출력 신호로 변환하는 함수를 출력하는 함수를 출력함수라고 한다. 출력 함수의 예시는 임계치에 따라 0과 1로 출력하는 함수는 계단 함수가 있다. 즉 단층 퍼셉트론은 입력층과 출력층으로 구성되어 입력 값에 대한 선형 함수들로 표현할 수 있으며 출력함수를 통해 입력 값을 분류할 수 있는 선형 분류 모델이라고 말할 수 있다. 해당 선형 분류 모델을 통해 AND와 OR, NAND 게이트를 구현할 수 있다. 하지만 XOR 게이트는 직선 하나를 통해 분류 할 수 없기 때문에 구현할 수 없다. XOR 게이트와 같이 선형성 모델이 해결하지 못하는 문제들이 생긴다.

해당 문제들을 해결 하기 위해 다층 퍼셉트론이 등장하였다. XOR 게이트의 경우 AND와 OR, NAND 게이트들을 조합하여 만들 수 있고, 이는 퍼셉트론의 층을 쌓는다면 구현할 수 있다는 것을 의미한다. 입력층과 출력층외의 추가적인 층들을 생성하여 비선형성의 문제들을 해결하는 것이고, 이러한 추가적인 층들은 실제 사용자들에게 보이지 않는 층이므로 은닉층이라고 부른다. 다층 퍼셉트론도 단층 퍼셉트론과 마찬가지로 입력 값들에 해당하는 가중치들이 있고, 입력 값과 그 입력 값에 해당하는 가중치의 곱들의 합을 다음 층인 은닉층으로 보낸다. 단층 퍼셉트론에서는 입력층 이후 출력층이었기 출력 신호로 변환하는 출력함수를 사용하였지만, 다층 퍼셉트론의 은닉층에서는 입력 신호의 총합의 활성화 여부를 결정하는 활성화 함수를 사용한다. 활성화 함수로 비선형 함수를 사용하여 비선형성 문제들을 해결하는 것이다. 활성화 함수의 예시로는 시그모이드, 하이퍼볼릭 탄젠트 함수, 렉티파이어 등이 있다.

퍼셉트론에서 학습은 가중치 조정을 통해 퍼셉트론에서 출력되는 신호가 학습하는 데이터의 신호와 유사할 수 있도록 하는 것을 의미하고, 이를 통해 우리가 보지 못한 데이터의 신호를 예측할 수 있는 것을 의미한다. 학습의 목적은 예측 값의 신호와 학습 값의 신호간의 오류를 줄이는 것이다. 해당 오류는 MSE, Kross Entropy 등의 손실함수를 통해 계산할 수 있고, 손실함수의 값이 작아지는 방향으로 가중치를 조정해 퍼셉트론을 최적화 하는 것을 경사 하강법이라고 한다. 이러한 학습은 순전파와 역전파 과정을 통해 이루어진다.

순전파는 입력층에서 시작하여 입력 값과 입력 값에 해당하는 가중치들의 곱들이 출력층까지 전파되어 출력 신호를 만들어내는 과정으로, 현재 학습한 가중치들을 이용하여 입력 값에 대해 출력 값을 예측하는 과정이다. 해당 예측 값과 입력 데이터에 대해 학습 시 주어진 출력 신호와의 오류를 손실함수를 통해 계산하여 현재 가중치들의 오차를 구한다. 해당 오차를 출력층부터 입력층으로 전파하며 각각의 층에 있는 가중치 조정으로 경사하강법 즉 손실함수의 값이 최소가 되는 지점을 찾아 나가는 과정을 역전파라고 한다.

4. MLP

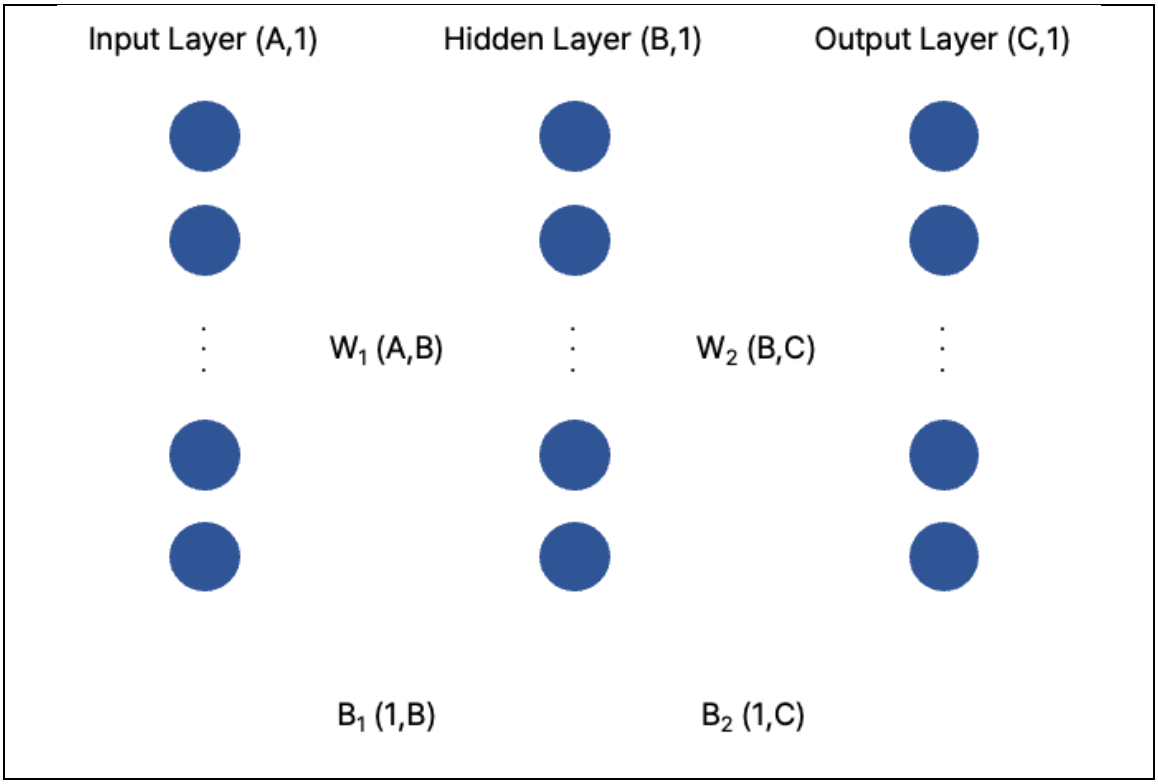


Figure 1. 구현하고자 하는 MLP의 구조

입력층과 출력층 그리고 하나의 은닉층을 가지고 그에 따른 가중치와 편향 값이 존재하는 구조의 기본 MLP 모델을 구현하려 한다. 이때 은닉층에서의 활성화 함수는 시그모이드 함수를 사용하며, 다차원 분류를 위해 출력함수로는 소프트맥스를 사용한다. 해당 출력함수와 주로 함께 사용하는 손실함수인 크로스 엔트로피를 사용하여 학습을 진행하고자 한다.

$$\mathrm{X} = \mathrm{x}_1, \mathrm{x}_2, \cdots, \mathrm{x}_\mathrm{n} \text{ and } \mathrm{Y} = \mathrm{y}_1, \mathrm{y}_2, \cdots, \mathrm{y}_\mathrm{n} \ and \ T = t_1, t_2, \cdots, t_n$$

$$\tau_1 = sigmoid \ and \ \tau_2 = softmax$$

$$L = lossfunction = crossEntropy$$

$$Y = \tau_2(\tau_1(W_1X + B_1)W_2 + B_2)$$

$$\text{if } \tau_1(W_1X + B_1)W_2 + B_2 = A = a_1, a_2, a_3$$
$$\text{and } \tau_2(a_1, a_2, a_3) = Y = y_1, y_2, y_3$$

$$L(a_1, a_2, a_3) = -t_1 \log y_1 - t_2 \log y_2 - t_3 \log y_3$$

$$= -t_1 \log \frac{e^{a_1} + e^{a_2} + e^{a_3}}{e^{a_1}} - t_2 \log \frac{e^{a_1} + e^{a_2} + e^{a_3}}{e^{a_2}} - t_3 \log \frac{e^{a_1} + e^{a_2} + e^{a_3}}{e^{a_3}}$$
$$= -t_1 \log e^{a_1} - t_2 \log e^{a_2} - t_3 \log e^{a_3} + (t_1 + t_2 + t_3) \log(e^{a_1} + e^{a_2} + e^{a_3})$$
$$= -t_1 a_1 - t_2 a_2 - t_3 a_3 + \log e^{a_1} + e^{a_2} + e^{a_3}$$
$$\frac{\partial L}{\partial a} = (y_1 - t_1, y_2 - t_2, y_3 - t_3) = Y - T$$

$$\frac{\partial \mathrm{L}}{\partial W_2} = \frac{\partial \mathrm{L}}{\partial \mathrm{A}} \frac{\partial \mathrm{A}}{\partial W_2} = (Y - T)\tau_1(W_1X + B_1)$$
$$\frac{\partial \mathrm{L}}{\partial W_1} = \frac{\partial \mathrm{L}}{\partial \mathrm{A}} \frac{\partial \mathrm{A}}{\partial \left(\tau_1(W_1X + B_1)\right)} \frac{\partial \left(\tau_1(W_1X + B_1)\right)}{\partial (W_1X + B_1)} \frac{\partial (W_1X + B_1)}{\partial W_1}$$
$$= (\mathrm{Y} - \mathrm{T})\mathrm{W}_2\tau_1'(\mathrm{W}_1\mathrm{X} + \mathrm{B}_1)\mathrm{X}$$

Equation 1. 소프트 맥스와 크로스 엔트로피의 순전파, 역전파 과정

위 식1을 통해 소프트 맥스 함수와 크로스 엔트로피 함수를 합성하여 도함수를 계산한 결과가 간단한 식으로 나옴을 확인할 수 있다. 해당 도함수를 활용하기 위해 출력층에 소프트맥스 함수를, 손실 함수로는 크로스 엔트로피함수를 사용하였다. 두 함수를 통해 역전파의 계산이 줄어들 수 있다. 예측 값과 실제 값의 오차를 소프트맥스 함수 입력 값으로 편미분 한 것이 Y-T로 간단하게 계산되는 것을 이용해 역전파를 진행하여 가중치와 편향 값을 조정한다. 해당 순전파와 역전파 과정을 MLP 클래스로 구현하였다.

```
class MLP:
    def __init__(self, hidden_size, learning_rate, epochs):
        self.hidden_size = hidden_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        # 입력층과 은닉층 사이의 가중치, 은닉층 편향
        self.weights_1 = None
        self.bias_1 = None
        # 은닉층과 출력층 사이의 가중치, 출력층 편향
```

```
        self.weights_2 = None
        self.bias_2 = None
        # loss 값 저장
        self.loss_list = []

    # 활성화함수, 시그모이드 -> 0~1 사이의 확률값
    def sigmoid(self, X):
        X = np.clip(X, -300, 300)
        return 1 / (1 + np.exp(-X))

    # 시그모이드의 미분
    def sigmoid_derivative(self, X):
        X = np.clip(X, -300, 300)
        return (np.exp(-X)) / ((np.exp(-X) + 1) ** 2)

    # 출력함수, 소프트맥스 -> 총합이 1 인 0~1 사이의 확률값
    def softmax(self, X):
        e_x = np.exp(X - np.max(X, axis = 1).reshape(X.shape[0],1))
        return e_x / np.sum(e_x, axis=1).reshape(X.shape[0],1)

    # 손실함수, Cross Entropy
    def cross_entropy(self, y_true, y_pred):
        y_pred = np.clip(y_pred, 0.0001, 0.9999)
        return -np.mean(np.sum(y_true * np.log(y_pred), axis=1))

    def feedforward(self, X):
        # (데이터의 개수,데이터 속성의 개수) * (데이터 속성의 개수,Hidden) + (1,Hidden) =
(데이터의 개수,Hidden)
        self.hidden_layer_input = np.dot(X, self.weights_1) + self.bias_1
        self.hidden_layer_output = self.sigmoid(self.hidden_layer_input)

        # (데이터의 개수,Hidden) * (Hidden,Output) + (1,Output) = (데이터의
개수,Output)
        self.output_layer_input = np.dot(self.hidden_layer_output, self.weights_2) +
self.bias_2
        self.output_layer_output = self.softmax(self.output_layer_input)
```

```
    def backpropagation(self, X, y):
        # cross_entropy 오차의 역전파를 통한 가중치와 편향을 조정
        # 전체 loss
        loss = self.cross_entropy(y, self.output_layer_output)

        # 출력층 오차, 전체 loss 에 대한 출력층 입력값(self.output_layer_input)의 편미분
값 : Equation 1 을 통한 증명
        error_output = (self.output_layer_output - y)
        # 전체 데이터에 대한 누적값이기 때문에 평균 연산 수행
        deltha_weight_2 = np.dot(self.hidden_layer_output.T, error_output) /
X.shape[0]
        deltha_bias_2 = np.mean(error_output, axis=0)

        # 은닉층 오차, 전체 loss 에 대한 은닉층 입력값(self.hidden_layer_input)의 편미분
값 : Equation 1 을 통한 증명
        error_hidden = np.dot(error_output, self.weights_2.T) *
self.sigmoid_derivative(self.hidden_layer_input)
        # 전체 데이터에 대한 누적값이기 때문에 평균 연산 수행
        deltha_weight_1 = np.dot(X.T, error_hidden) / X.shape[0]
        deltha_bias_1 = np.mean(error_hidden, axis=0)

        # 가중치 조정
        self.weights_2 -= deltha_weight_2 * self.learning_rate
        self.bias_2 -= deltha_bias_2 * self.learning_rate
        self.weights_1 -= deltha_weight_1 * self.learning_rate
        self.bias_1 -= deltha_bias_1 * self.learning_rate

    def train(self, X, y):
        for epoch in range(self.epochs):
            self.feedforward(X)
            self.backpropagation(X,y)
            if epoch % 10 == 0:
                loss = self.cross_entropy(y, self.output_layer_output)
                print(f"epoch {epoch}, loss {loss}")

    def retrunLossList(self):
        return self.loss_list
```

```
    def predict(self, X):
        self.feedforward(X)
        return np.argmax(self.output_layer_output, axis=1)

    def fit(self, X, y):
        # 데이터의 속성 개수
        input_size = X.shape[1]
        # 데이터의 분류 개수, 라벨의 개수
        output_size = len(np.unique(y))

        # 입력층 -> 은닉층 가중치 : (데이터 속성의 개수,Hidden) and 은닉층 편향 :
(1,hidden)
        self.weights_1 = np.random.randn(input_size, self.hidden_size)
        self.bias_1 = np.ones((1, self.hidden_size))

        # 은닉층 -> 출력층 가중치 : (Hidden,Output) and 출력층 편향 : (1,Output)
        self.weights_2 = np.random.randn(self.hidden_size, output_size)
        self.bias_2 = np.ones((1, output_size))

        # numpy 배열로의 변환 및 target 값 원핫인코딩
        X = np.array(X)
        y = np.array(pd.get_dummies(y['target']))

        # 입력된 데이터로 학습 진행
        self.train(X, y)
```

Code 1. MLP Vanilla Model

MLP의 기본 클래스를 구현하였다. MLP 클래스는 은닉층의 노드 개수와 학습률, 에포크를 설정하여 생성한다. 이후 fit 함수를 통해 입력 데이터에 대한 학습을 진행한다. X 입력 데이터의 행은 데이터의 개수, 열은 데이터의 속성을 가지고 y 입력 데이터는 원 핫 인코딩하지 않은 1열의 라벨 데이터인 X, y를 모델 내부에서 가공하여 학습과 예측을 진행한다. 활성화 함수는 sigmoid, 출력함수는 softmax, 손실함수는 cross entropy로 해당 함수를 클래스 내부에 생성하였다.

입력 값을 자연상수의 지수 값으로 변환하는 np.exp 연산의 경우 입력 값이 커지거나 작아지면 그 반환 값이 기하급수적으로 커지거나 0으로 수렴하게 된다. 이는 컴퓨터의 계산 범위를 벗어날 수 있기에 sigmoid 함수에서는 입력 값의 범위를 제한하여 클리핑 처리하였다. softmax의

경우 클리핑 처리하지 않고 입력된 값의 가장 큰 값으로 뺄셈 연산을 통해 np.exp 연산을 수행하였다. cross entropy의 경우 입력 값은 0~1 사이의 값이다. 해당 손실함수는 log 연산이 포함되어 있어 np.exp 연산과는 반대로 기하급수적으로 작아질 수 있다. 컴퓨터의 연산을 위해 입력 값의 범위를 제한하여 너무 작은 입력 값이 없도록 구현하였다.

5. Experiments

1. IRIS 분류

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	1.0	0.9666	1.0	1.0	0.9333
F1	1.0	0.9663	1.0	1.0	0.9333

Table 1. 구현한 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	1.0	0.9333	1.0	1.0	0.9333
F1	1.0	0.9333	1.0	1.0	0.9333

Table 2. scikit learn 모델의 교차 검증 성능 지표

구현한 모델과 라이브러리의 모델을 k-fold를 통한 교차 검증을 진행하였고 성능을 측정하였다. 두 모델은 10개의 은닉층 노드를 가지고 1800회의 epoch로 학습을 동일하게 진행하였다. 성능 지표를 통해 두 모델은 비슷한 성능을 내는 것으로 확인된다. IRIS 데이터 자체가 많은 양의 데이터로 이루어져 있지 않기 때문에 충분한 학습이 된 두 모델 모두 좋은 결과가 나오는 것이라 추측할 수 있다.

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.9666	0.9333	1.0	1.0	0.9333
F1	0.9657	0.9339	1.0	1.0	0.9333

Table 3. 표준화 후 구현한 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	1.0	0.9	0.9666	1.0	0.9
F1	1.0	0.9009	0.9665	1.0	0.8992

Table 4. 표준화 후 scikit learn 모델의 교차 검증 성능 지표

데이터의 각 속성 별 단위가 다르거나 범위가 다를 수 있다. 이를 조정하기 위해 데이터 스케일링 기법 중 표준화를 사용하였다. 표준화 된 데이터로 모델 학습을 진행하여 성능을 평가했다. Fold를 나눈 이후에 학습하는 데이터로 표준화 모델을 학습하였고

예측해야 할 데이터에 적용하여 Data leakage를 방지하였다. 입력된 데이터의 속성을 표준화 하였으나 모델의 뚜렷한 성능 향상은 보이지 않았다. 이 또한 데이터의 양이 적지만 학습을 많이 했기에 어느정도 고정된 성능을 보이는 듯 하다.

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.9333	0.9	1.0	0.9333	0.8666
F1	0.9353	0.8989	1.0	0.9343	0.8639

Table 5. 구현한 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.5666	0.5	0.6666	0.6666	0.6666
F1	0.4626	0.3571	0.6632	0.5757	0.6458

Table 6. scikit learn 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.9666	0.8333	0.9333	1.0	0.9
F1	0.9657	0.8349	0.9326	1.0	0.8992

Table 7. 표준화 후 구현한 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.6666	0.7	0.9	0.9	0.8666
F1	0.6382	0.7	0.8976	0.8978	0.8639

Table 8. 표준화 후 scikit learn 모델의 교차 검증 성능 지표

표 1~4의 네 성능 지표는 모두 1800회의 epoch로 학습을 진행하였다. 뚜렷한 성능 차이를 보이지 않아 입력 데이터에 대한 과적합을 의심하였고 epoch만 300으로 줄여 각각의 모델의 성능을 다시 평가해본다. epoch를 줄인 실험에서 직접 구현한 모델은 여전히 좋은 성능을 보이지만 라이브러리의 모델은 학습이 잘 되지 않은 경우를 볼 수 있었고, 표준화에 따른 성능을 분석해보니 라이브러리 모델에서 그 성능 차이가 돋보였다. 해당 성능 결과의 원인은 MLP의 구조적인 문제이지 않을까 생각이 되었다.

현재 구현한 모델의 경우 배치학습으로 한 epoch에서 전체 데이터에 대해 학습을 진행하고 있다. 라이브러리 모델의 문서를 확인해보니 데이터의 크기가 200보다 작다면 전체 데이터를 학습에 사용하며 200보다 크다면 200개의 batch 데이터를 사용한다. 전체 데이터에서 무작위로 batch를 추출하는 stochastic gradient descent를 사용하는 것이다. 현재 IRIS 데이터의 전체 데이터 개수가 적기 때문에

라이브러리 모델의 경우 전체 데이터로 학습을 진행하게 된다. 해당 데이터로 학습 중 기울기의 계수나 학습률의 계수를 변형시키는 Adam Optimizer를 사용한다. Adam Optimizer의 경우 stochastic mini batch 학습을 통해 학습에서의 계산량을 줄이고 가중치의 업데이트 횟수는 늘리는 효과를 얻는데, 현재 full batch로 학습이 진행되어 stochastic mini batch의 장점은 잃어버리고 오히려 기울기의 계수나 학습률의 계수의 학습이 수행되어야 하는 더 복잡한 상황에 처한 것이라 생각하였다. 더 복잡한 상황 속에서 표준화된 입력 데이터 값으로 인해 성능이 어느정도 유지된 모습을 보이지 않나 추측된다. 이러한 구조적인 차이로 인해 구현한 모델에서는 우수한 성능을 여전히 보여주고 있지만 라이브러리 모델에서는 급격한 성능 저하가 이루어진 것으로 판단된다.

```
class MLP_MINIBATCH:
    def __init__(self, hidden_size, learning_rate, epochs, batch_size):
        self.hidden_size = hidden_size
        self.learning_rate = learning_rate
        self.epochs = epochs
        self.batch_size = batch_size
        # 입력층과 은닉층 사이의 가중치, 은닉층 편향
        self.weights_1 = None
        self.bias_1 = None
        # 은닉층과 출력층 사이의 가중치, 출력층 편향
        self.weights_2 = None
        self.bias_2 = None
        # loss 값 저장
        self.loss_list = []

    def train(self, X, y):
        num_samples = X.shape[0]
        for epoch in range(self.epochs):
            for i in range(0, num_samples, self.batch_size):
                X_batch = X[i:i+self.batch_size]
                y_batch = y[i:i+self.batch_size]

                self.feedforward(X_batch)
                self.backpropagation(X_batch, y_batch)

            if epoch % 10 == 0:
                loss = self.cross_entropy(y_batch, self.output_layer_output)
                print(f"epoch {epoch}, loss {loss}")
```

Code 2. mini-batch 모델 구현

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.9333	0.9333	0.8666	1.0	0.9
F1	0.9353	0.9317	0.8611	1.0	0.8992

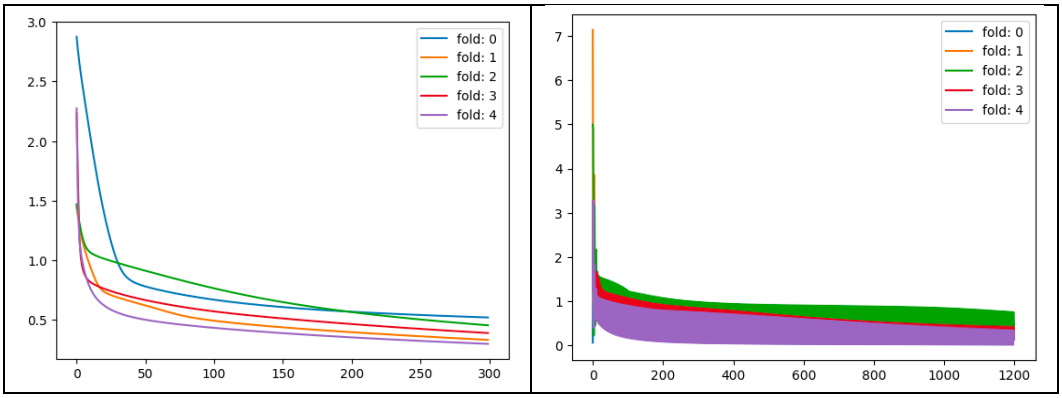
Table 9. 구현한 mini-batch 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	1.0	0.8666	1.0	1.0	0.9
F1	1.0	0.8679	1.0	1.0	0.8992

Table 10. 표준화 후 구현한 mini-batch 모델의 교차 검증 성능 지표

라이브러리 모델 문서를 통해 구조를 살펴보고 구현한 모델과 비교하며 mini batch 학습의 장점을 알 수 있었고 해당 mini batch를 구현해본다. Code1번 과 비교해 클래스 선언과 학습에서의 코드만 달라진 것을 확인할 수 있다. 또한 K-fold로 성능 검증을 진행하였는데 이 때 데이터의 분포를 섞어서 진행하였다. 이로 인해 단순히 데이터를 mini batch로만 나누어 학습하는 것이 아니라 shochastic mini batch로 학습한 모델의 성능을 확인할 수 있다.

표 5~8과 같이 은닉층의 노드는 10개, epoch는 300으로 설정 후 배치 사이즈만 32로 하여 학습을 진행하였고 기본 MLP 모델과 유사한 성능이 측정되었다. IRIS의 데이터 양이 적고 기본 MLP 모델에서도 충분히 우수한 성능을 가지고 있었기 때문에 큰 성능 향상이 이루어 지지 않은 듯 하다.



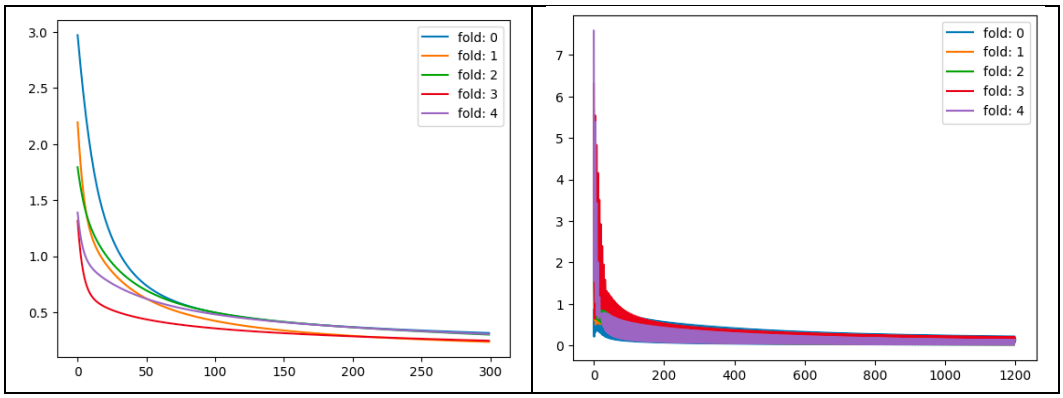


Figure 2. IRIS 학습에 따른 손실 변화

왼쪽 위 : 기본 모델, 오른쪽 위 : mini batch 모델,

왼쪽 아래 : 표준화 후 기본 모델, 오른쪽 아래 : 표준화 후 mini batch 모델

위 그림 2를 통해 mini batch 학습이 손실함수를 더 많이 계산하고 이를 통해 더 많은 파라미터 조정을 했음을 알 수 있다. 또한 전체 데이터에 대한 손실함수 계산이 아니기 때문에 진폭이 생겼음을 확인할 수 있다. 그리고 표준화의 효과도 볼 수 있는데 mini batch의 경우 진폭의 크기가 줄어들었음을 확인할 수 있고, 기본 모델의 경우 학습시 좀 더 완만하게 손실이 줄어드는 것을 알 수 있다.

2. MNIST 분류

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.2322	0.2092	0.2210	0.2161	0.2730
F1	0.2318	0.2077	0.2203	0.2146	0.2721

Table 11. 기본 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.2280	0.2221	0.2287	0.2441	0.2481
F1	0.2277	0.2214	0.2253	0.2425	0.2484

Table 12. 정규화 후 기본 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.8864	0.8892	0.8936	0.8835	0.8900
F1	0.8862	0.8889	0.8936	0.8835	0.8899

Table 13. mini-batch 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.9082	0.9098	0.9112	0.9109	0.9083
F1	0.9082	0.9098	0.9112	0.9109	0.9083

Table 14. 정규화 후 mini-batch 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.9473	0.9542	0.9537	0.9495	0.9547
F1	0.9473	0.542	0.9537	0.9495	0.9547

Table 15. scikit learn 모델의 교차 검증 성능 지표

	iteration 1	iteration 2	iteration 3	iteration 4	iteration 5
Accuracy	0.9750	0.9763	0.9757	0.9789	0.9782
F1	0.9749	0.9763	0.9757	0.9789	0.9782

Table 16. 정규화 후 scikit learn 모델의 교차 검증 성능 지표

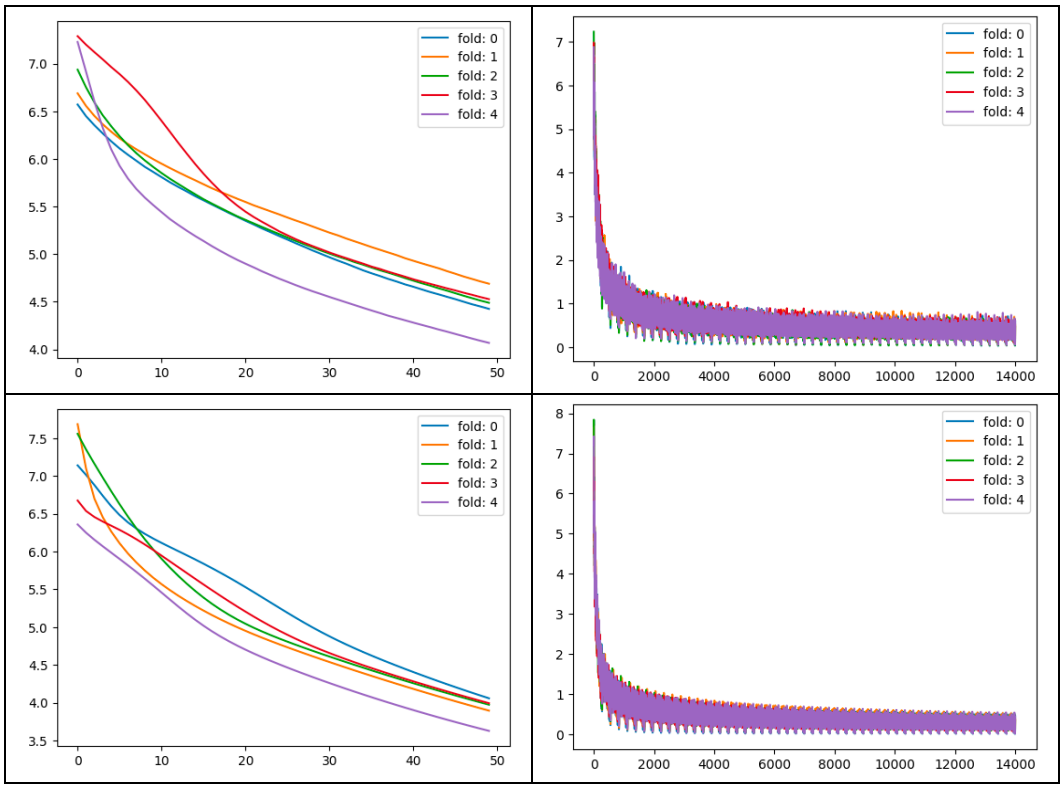


Figure 3. MNIST 학습에 따른 손실 변화
왼쪽 위 : 기본 모델, 오른쪽 위 : mini batch 모델,
왼쪽 아래 : 정규화 후 기본 모델, 오른쪽 아래 : 정규화 후 mini batch 모델

IRIS 데이터 세트를 통해 구현한 두 모델을 파라미터만 조정하여 MNIST 데이터에 적용해 보았다. 은닉층은 128개의 노드로 구성하였고 epoch는 50, batch의 size는 200으로 적용하여 수행해보았다. MINIST 데이터는 0~255의 밝기값으로 이루어져 있어 해당 수치 차이가 크게 작용하기 때문에 정규화 처리를 적용하였다. 정규화를 통해 epoch에 따라 loss값의 변화량이 많이 일어나지 않는 모습을 그림 3을 통해 확인할 수

있다. 또한 표 11~16을 통해 mini-batch 모델 학습이 우수한 성능을 보여줌을 확인할 수 있다. 라이브러리 모델의 성능보단 약간 부족하지만, 기본 모델보다 매우 좋은 성능을 보이고 있다. 앞서 데이터의 개수가 적었던 IRIS 분류 문제에서는 확인할 수 없던 성능의 차이이다. 현재 MNIST 데이터가 7만개이고 속성은 784개로 IRIS와 비교할 수 없이 방대하고 복잡한 데이터이다. 이런 데이터로 50번의 학습을 진행한다면 좋지 못한 결과를 얻는 것을 확인하였다. mini batch의 경우 한 반복에서 7만개의 데이터를 200개 씩 학습하여 약 14,000번의 가중치 조정이 이루어진 것이다. 현재 본인의 M1 맥북 에어 기준으로 학습에 비슷한 시간이 소요되었지만 가중치의 조정의 차이가 발생했고, 이에 따른 성능 지표의 차이가 발생한 것으로 파악했다.

6. Discussion

MLP를 scikit learn의 라이브러리를 사용하지 않고 구현하였다. 학과 수업시간에 배운 내용 뿐 아니라 여러 내용들을 참고하여 순전파 및 역전파 과정을 이해할 수 있었다. 이론을 이해하고 그 내용을 코드로 구현하는 과정에서 많은 오류와 어려움을 겪을 수 있었다. MLP를 구현하면서 사전에 이해했던 내용들이 몸소 느낄 수 있었다. 실제로 생성한 모델이 데이터를 예측하는 것을 보고 뿌듯함을 가질 수 있었고, 라이브러리의 모델과 유사한 성능을 낼 수 있도록 모델을 수정하고 여러 시도를 진행해 보았다. 하지만 역전파 과정을 이해하는데 오랜 시간이 걸려 기발하고 다양한 시도들을 못해 본 것이 아쉽다.

모델의 성능을 평가할 때 교차 검증을 이용했기 때문에 해당 성능 지표가 일반화된 성능을 대표할 것을 기대했다. 구현한 모델과 라이브러리 모델의 성능 비교를 하며 궁금한 점이 생겼고, 궁금한 점을 해결하는 과정에서 라이브러리 모델의 구조를 확인할 수 있었다. 해당 모델 구조에서 mini batch 학습의 장점을 공부할 수 있었고, 이를 바탕으로 기본 모델에서 mini batch 모델을 구현할 수 있었다. 기본 모델의 활성화 함수는 시그모이드를 사용하고, 출력 함수로는 소프트맥스, 크로스 엔트로피 손실함수를 사용하여 구성을 했다. 기본 모델에서 파생한 mini batch 모델 또한 같은 구성으로 이루어져있다. 시그모이드 활성화 함수의 경우 양 극단에서 기울기의 값이 0이 되는 문제가 발생하는데, 이를 방지할 수 있는 Relu 함수와 같은 다른 활성화 함수들을 사용해보지 못한 것이 아쉽다. 뿐만 아니라 현재 학습률을 0.08로 고정하였는데, 이 또한 여러 옵티마이저를 직접 구현 후 적용해보았다면 조금 더 좋은 성능의 모델을 생성할 수 있지 않았을까 싶다.

구현한 모델을 학습하면서 과적합에 대해 신경 쓰며 성능을 평가하려고 노력하였다. 구현했던 stochastic mini batch 이외의 과적합을 방지할 수 있는 규제나 드롭아웃 기법들을 구현해보았다면 더 좋았을 것 같다. 뿐만 아니라 현재 모델은 하나의 은닉층으로 구성되어 있다. 여러

개의 은닉층으로 구성한다면 더욱더 방대하고 복잡한 데이터들을 분류하는 모델을 구성할 수 있을 것 같다.

784개의 속성을 가지는 7만개의 데이터인 MNIST의 데이터를 이번 과제를 통해 처음 다루어 보았는데, 현업에서 사용하는 데이터들은 더욱 복잡하고 크기가 큰 데이터일 것이라 생각한다. 현대 사회에서 현재 구현하는 딥러닝 모델들 뿐 아니라 데이터들의 중요성을 알 수 있던 과제라 생각한다. 데이터가 현 시대 및 미래에 가장 큰 자산으로, 이를 처리하고 모델링을 통한 다양한 서비스들을 개발하여 사회를 변화시킬 수 있는 힘을 가진다고 생각한다.

현재 과제에서 멈추지 않고 딥러닝 모델링과 같은 공부 뿐 아니라 데이터 수집과 처리 기법 등의 공부 및 여러 분야의 지식들을 같이 공부하며 그 이론적인 내용들을 충분히 이해하고 있다면 살아가면서 사회를 더 좋은 방향으로 변화시킬 수 있는 능력이 있는 인재가 될 수 있을 거라 생각한다.

References

1. MLP (https://tensorflow.blog/%ED%95%B4%EC%BB%A4%EC%97%90%EA%B2%8C-%EC%A0%84%ED%95%B4%EB%93%A4%EC%9D%80-%EB%A8%B8%EC%8B%A0%EB%9F%AC%EB%8B%9D-3/)
2. Softmax-with-Loss (https://value-error.tistory.com/55)
3. 오차역전파의 연쇄 법칙 (https://deep-learning-study.tistory.com/16)