Multi-Layer Perceptron from Scratch

목차

1. 서론

이 연구는 Multi-Layer Perceptron 의 성능을 개선하여 데이터들을 높은 정확도로 예측하는 데에 목적이 있다. 인공신경망을 이용한 Multi-Layer Perceptron 을 이용하여 iris dataset 과 mnist dataset 을 높은 성능으로 분류하는 MLP 모델을 직접 구현해 설명한다.

직접 구현한 Multi-Layer Perceptron 을 이용해 분류해 볼 dataset 은 iris 와 mnist 이다. Iris dataset 은 붓꽃에 대한 데이터이다. X 가 될 data 들은 총 4 가지로 'sepal length', 'sepal width', 'petal length', 'petal width'이다. y 가 될 target 은 'setosa', 'versicolor', 'virginica' 3 가지이다. Mnist dataset 은 0 부터 9 까지의 이미지로 구성된 손글씨 dataset 이다. 총 60000 개의 훈련 데이터와 10000 개의 테스트 데이터와 레이블(10 개)로 구성되어 있다.[1] 이 데이터들에 맞는 encoding 방법과 learning rate, training size 를 각각 설정해주고 학습을 진행하였다.

성능 개선을 위해 직접 수정할 수 있는 파라미터들은 학습률, epoch, 은닉층 개수, 노드 수, 배치 등이 있다. 먼저, 학습률은 손실함수의 미분 값이 0 이 되는 지점을 찾을 때, 경사를 따라 이동하는 양이다.[2] Epoch 는 학습을 종료할 때까지의 실행 횟수이다. 모델은 은닉층의 개수나 노드 수를 조절하며 복잡도를 조절할 수 있으며, 배치는 모든 데이터를 한 번에 처리하지 않고 epoch 를 작은 데이터 셋으로 나누어 수행하는 것이다.[3]

---

[1] 유원준, 안상준, PyTorch 로 시작하는 딥 러닝입문, https://wikidocs.net/60324

[2] Douglas' Space, 하이퍼 파라미터(Hyper Parameters)란?, 2022.04.28, https://doug.tistory.com/m/44

[3] 위의 글.

모델의 성능평가는 sklearn 의 model 과 정확도(Accuracy)를 비교하여 평가하였다. 정확도를 기준으로 비교한 이유는 모델이 올바르게 예측한 샘플의 비율로 계산되는 정확도는 예측 능력을 직관적으로 평가할 수 있도록 해주기 때문이다.[4] Sklearn 의 모델을 default 설정으로 맞춰 놓고, 똑같은 model 을 직접 구현해 성능을 비교 후, 구현한 모델의 정확도를 높이기 위해 많은 하이퍼 파라미터, 활성함수 등을 변경하면서 연구해보았다. iris 데이터는 정확도 면에서 sklearn model 과 크게 차이는 없지만 어떤 특정 파라미터에서 정확도가 유독 낮음을 알게 되었다. Mnist 데이터는 하이퍼 파라미터를 변경할 때마다 정확도가 계속 다르게 출력되었고, 모델이 복잡하다고 해서 더 좋은 성능을 보여주지는 않았다.

2. 본론

2.1. MLP 개요

기계학습은 실제 값과 예측 값의 차이가 최소가 되는 정확한 예측 모델을 만드는 것이 최종 목표라고 할 수 있다. Multi-Layer Perceptron 은 입력층과 출력층 사이에 하나 이상의 중간층이 존재하는 신경망[5]으로, 모델의 활성 함수 출력 값과 실제 값의 활성 함수 출력 값이 같아질 때까지 값을 계속 수정한다. MLP 는 다양한 형태의 퍼셉트론 구조가 가능할 뿐만 아니라, 각 계층의 가중치를 모두 학습하여 결정할 수 있다는 장점이 있다.[6]

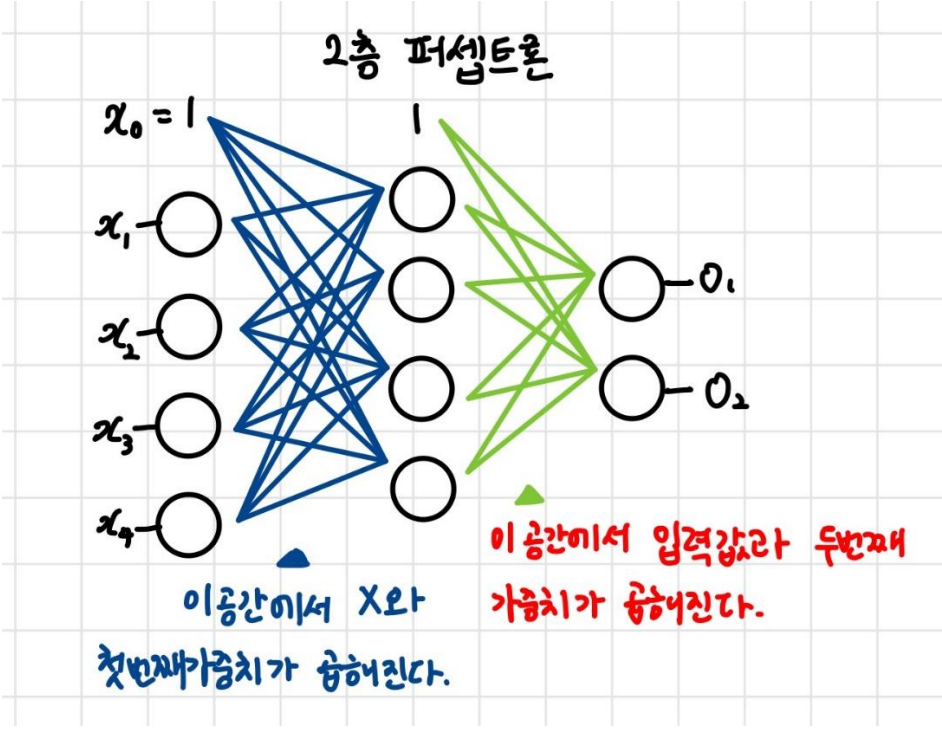
---

[4] Chat gpt

[5] 박종덕, "표본 데이터 선정 방법에 따른 MLP 분류기의 성능 고찰." 국내석사학위논문 명지대학교 산업기술대학원, 1997, 3 쪽.

[6] 위의 글, 4 쪽.

중간층인 은닉층은 원래 특징 공간을 분류하는 데 훨씬 유리한 새로운 특징 공간으로 변환한다. 또한, 층과 층을 연결하는 edge 는 가중치(W)를 가지고 있다.



Feedforward 와 back propagation 을 사용해 optimization 을 진행하여 최종 분류를 진행하게 된다.

2.2. 설명

Feedforward

입력층으로 데이터가 입력되고, 한 개 이상으로 구성되는 은닉층을 거쳐 마지막에 있는 출력층으로 값을 내보낸다.

Multi-Layer Perceptron 은 특징 값과 가중치를 곱한 결과를 모두 더하여 s 를 구하고, 활성함수를 적용한다.

$$s = w_0 + \sum_{i=1}^{d} w_i x_i \qquad y = \sigma(s)$$

퍼셉트론은 계단함수를 활성함수로 사용하고 이는 경성(hard) 의사결정에 해당한다. 다층 퍼셉트론은 연성(soft) 의사결정이 가능한 시그모이드 활성함수를 도입한다.[7]

---

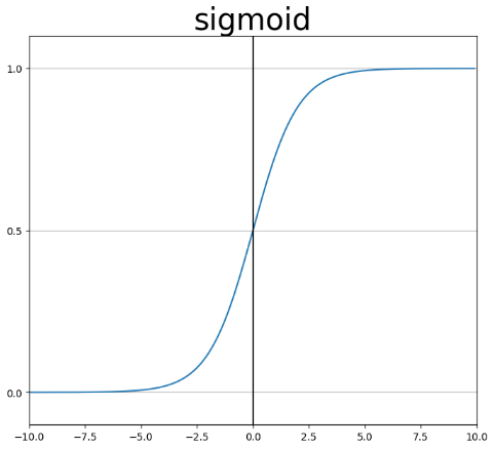
[7] 강의자료 WO3, 19 쪽.

```
def feedforward(self, X, y):

      input_x = self.feed(X, self.w1, self.b1)
      input_x = self.sigmoid(input_x)

      osum = self.feed(input_x, self.w2, self.b2)
      o_1 = self.softmax(osum)

      return o_1, input_x
```

```
def feed(self, x, w, b):
      f = np.dot(x, w) + b
      return f
```

위 함수는 은닉층 1 개로 이루어진 다층 퍼셉트론의 feedforward 를 진행하는 함수이다. Inpux_x 는 W x X + B 를 계산하고 그 값을 sigmoid 함수로 보낸다.

```
def sigmoid(self, z):
    return 1 / (1 + np.exp(-z))
```



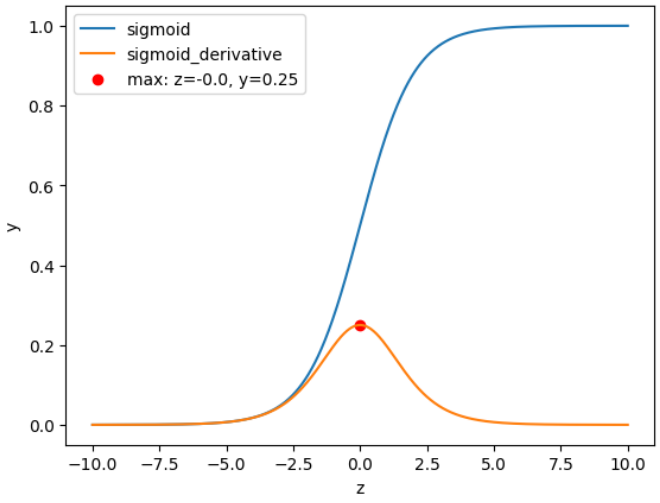
(sigmoid 함수)

sigmoid 함수는 0 ~ 1 사이의 실수를 반환한다. 무조건 0과 1로 반환하는 계단 함수의 단점을 보완한다. 출력값이 어느 값에 가까운지를 통해 어느 분류에 속하는지 쉽게 알 수 있다. 그러나, 입력값이 아무리 크더라도 출력되는 값의 범위가 매우 좁기 때문에 경사하강법 수행 시에 범위가 너무 좁아서 0에 수렴하는 기울기 소실(gradient vanishing)이 발생할 수 있다.[8] 기울기 소실이란 역전파 과정 중, 전달되는 기울기의 크기가 매우 작아져 실질적으로 학습이 이루어지지 않는 문제이다.[9]

---

[8] 만년필잉크, 딥 러닝-3.1. 활성화함수(2)-시그모이드 함수, 2021.1.25, https://gooopy.tistory.com/52

[9] 박종덕, 앞의 글, 10 쪽.



(z는 입력값)

Sigmoid의 미분 값은 z=0일 때, 0.25로 가장 크다. z값이 크거나 작아질수록 기울기는 거의 0에 가까워짐을 알 수 있다. 뒤에서 다룰 역전파 과정에서 chain rule이 적용되어 sigmoid의 미분값이 계속 곱해지면 출력층과 멀어질수록 gradient값이 매우 작아진다.[10] Gradient가 작아질수록 학습 속도 저하에 영향을 끼칠 가능성이 커진다.

다층 퍼셉트론의 기울기 소실 문제를 해결할 수 있는 방법은 sigmoid 대신 ReLU를 사용하는 것이다. 양수인 값을 가진 구간에 대해 기울기 값이 1을 갖는 ReLU 함수를 사용하여 기울기 소실을 해결할 수 있다.[11]

```
def ReLU(self, z):
        return np.maximum(0, z)
```

```
def feedforward(self, X, y):

        input_x = self.feed(X, self.w1, self.b1)
        input_x = self.ReLU(input_x)

        osum = self.feed(input_x, self.w2, self.b2)
        o_1 = self.softmax(osum)

        return o_1, input_x
```

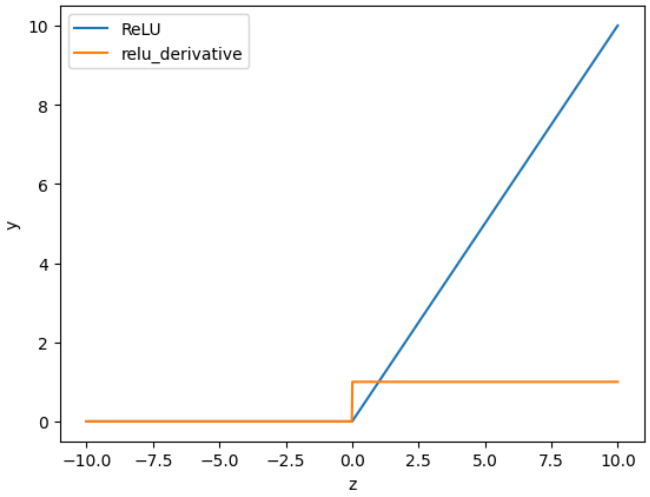
ReLU함수는 max함수를 통해 입력값과 0을 비교해 더 큰 값을 찾는다. 즉, 입력값이 0보다 작으면 0을 반환한다. 입력값이 양수면 값이 무엇이던 항상 1을 반환하기 때문에 기울기 소실 문제

---

[10] Tony Park, [딥러닝] 기울기 소실(Vanishing Gradient)의 의미와 해결방법, 2022. 5. 22, https://heytech.tistory.com/388

[11] 고영민. "인공지능 심층신경망의 기울기 소실 문제 3 가지 유형 분류와 파라메트릭 활성함수를 사용한 완화 방법." 국내석사학위논문 전주대학교 일반대학원, 2022. 전라북도, 10 쪽.

를 해결할 수 있다.



(ReLU함수와 도함수 그래프)

활성함수를 통해서 나온 값은 첫번째 은닉층의 입력 값이 된다. 똑같이 가중치를 곱하고 바이어스를 더해준다. 그렇게 나온 값을 osum이라는 변수에 저장하였다. osum은 출력층에서 활성함수를 사용하기 전 값이다. 출력층의 활성함수는 softmax 함수를 사용하였다.

```
def softmax(self, z):
    exp_z = np.exp(z)
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)
```
(그림 softmax-1)

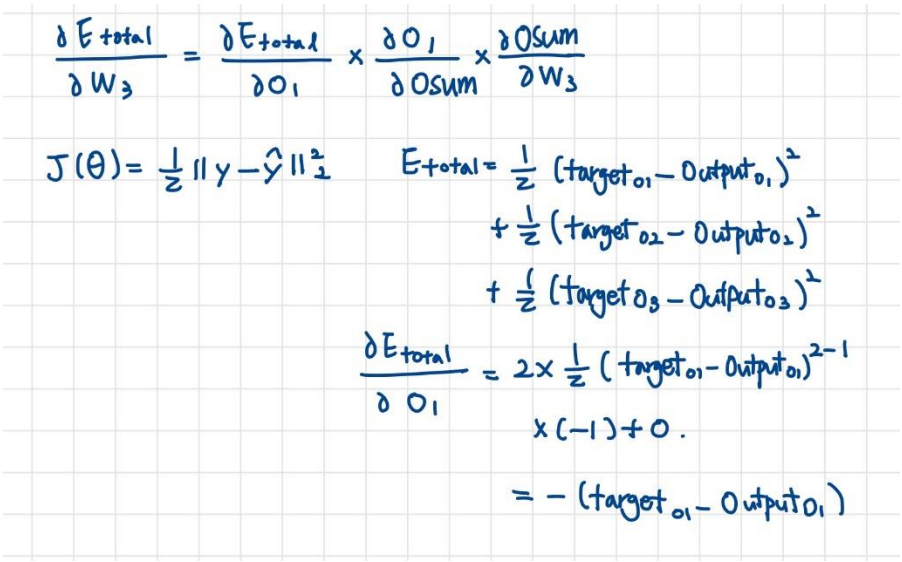
```
def softmax(self, z):
     z_max = np.max(z)
     exp_z = np.exp(z-z_max)
     y = exp_z / np.sum(exp_z, axis=1, keepdims=True)
     #print(y)
     return y
```
(그림 softmax-2)

softmax는 세개 이상으로 분류하는 다중 클래스 분류에서 주로 사용한다. 출력은 0 ~ 1사이의 실수이고, 출력의 총합은 1이다. softmax는 확률값을 출력하므로 최종 출력층에서 주로 사용하는 활성함수이다. 확률값이기 때문에 다른 출력값들과의 상대적인 크기 비교를 할 수 있다. 또한, cross entropy를 손실함수로 사용할 때, 확률값을 사용하기 때문에 계산이 쉬워진다. 그러나, softmax는 함수에서 지수를 사용해 너무 큰 값은 overflow가 생길 수도 있다. 그 부분을 보완하기 위해 softmax 함수의 입력값 중 최댓값을 빼고 구현하였다. (그림 softmax-2)

back propagation

역전파는 학습하기 전 초기화했던 가중치를 feedforward 를 마친 후, 업데이트하여 최적의 값을 찾는 방법이다. Feedforward 는 순방향이고 역전파는 역방향이다. 출력층 바로 이전의 은닉층을 N 층이라고 하였을 때, 출력층과 N 층 사이의 가중치를 업데이트하는 단계를 역전파 1 단계, 그리고 N 층과 N 층의 이전층 사이의 가중치를 업데이트 하는 단계를 역전파 2 단계라고 생각하면 된다.[12]

2 개의 은닉층을 가진 mlp 의 출력층에서 마지막 가중치를 업데이트하는 식이다. chain rule 을 사용한다.



이를 반영하여 알고리즘을 구현하였다.

[12] 안상준, 유원준, 딥 러닝을 이용한 자연어 처리 입문, https://wikidocs.net/37406

```
def backpropagation(self, X, y, epochs, lr):

    batches =X_train.shape[0] // self.size

    for epoch in range(epochs):

        #섞어서 하나 뽑기
        shuffle = np.random.permutation(len(X))
        X = X[shuffle]
        y = y[shuffle]

        for batch in range(batches) :
          start = batch * self.size
          end = (batch+1) * self.size

          X_batch = X[start:end]
          y_batch = y[start:end]

          o_1, input_x= self.feedforward(X_batch,y_batch) #feedfoward한 값을 받아온다.
          loss = self.cross_entropy(y_batch, o_1)
          sgd_loss = (o_1 - y_batch)

          update_w2, update_b2, pass_loss =self.batch(X_batch, input_x, sgd_loss, self.w2)
          update_w1 = np.dot(X_batch.T, pass_loss) / self.size
          update_b1 = np.sum(pass_loss, axis=0, keepdims=True) / self.size

          # 가중치와 편향 업데이트

          self.w2 -= lr * update_w2
          self.b2 -= lr * update_b2
          self.w1 -= lr * update_w1
          self.b1 -= lr * update_b1
```

```
def batch (self, X, o, loss, w):

      update_w = np.dot(o.T,loss) / self.size #이전 레이어에 전달하여 기울기를 계산
      update_b = np.sum(loss, axis=0, keepdims=True) / self.size #(o_1 - y)에서 샘플에 대한 오차를 모두 더한다.
      pass_loss = np.dot(loss, w.T) #출력층에서 역전파되는 오차를 이전 은닉층에 전달하기 위한 값
      pass_loss[o <= 0] = 0                #이전 은닉층에서의 가중치 업데이트에 사용
      pass_loss = pass_loss * (o > 0) #활성화(relu) 함수 미분과 곱하여 두번째 은닉층에서의 오차값을 구함

      return update_w, update_b, pass_loss
```

Update_w2, Update_b2 를 계산하기 위해서 X, 첫번째 은닉층 출력값, 오차, 가중치를 보낸다. 이는 출력층 오차에 대한 가중치 미분을 계산하는 것이다. 이 가중치는 출력층과 첫번째 은닉층 사이의 연결을 나타낸다. 따라서, 이 결과값은 파라미터로 보낸 가중치를 업데이트할 수 있게 한다.

Update_b2 는 오차를 모두 더하여 업데이트 한다.

Pass_loss 는 출력층과 첫번째 은닉층 사이의 출력값에 대한 미분이다. 이전 은닉층을 구하기 위해 loss 와 가중치를 곱한다. 출력층에서 역전파되는 오차를 이전 은닉층에 전달하기 위하여 계산한다.

그 다음, 0 보다 작은 값을 0 으로 두고, 계산한 값을 relu 함수를 미분한 것과 곱해서 반환한다. 그렇게 동일한 방법으로 첫번째 가중치와 바이어스까지 업데이트를 해준다.

손실함수는 교차 엔트로피를 사용하였다. 엔트로피는 정보량을 의미한다. 교차 엔트로피는 P 에 대한 확률분포 Q 가 있을 때, Q 가 P 와 얼마나 비슷한지를 나타내는 의미가 있다.[13]교차 엔트로피에서 실제값과 예측값이 맞는 경우에는 0 으로 수렴하고 값이 틀릴 경우에는 값이 커지기 때문에 두 확률분포가 서로 얼마나 다른 지를 나타낸다.[14]

이것은 sigmoid 를 사용했을 때의 역전파코드이다.

```
def batch (self, X, o, loss, w):

      update_w = np.dot(o.T,loss) / self.size #이전 레이어에 전달하며 기울기를 계산
      update_b = np.sum(loss, axis=0, keepdims=True) / self.size #(o_1 - y)에서 샘플에 대한 오차를 모두 더한다.
      d_a = np.dot(loss, w.T)
      pass_loss = d_a * o * (1 - o)


      return update_w, update_b, pass_loss
```

활성함수의 미분 부분이 다른 것을 알 수 있다. Relu 와 sigmoid 의 차이는 실험 파트에서 알아보도록 하겠다.

2.3. 실험

Iris 데이터를 사용한 실험

```
iris = load_iris()
X = iris.data
y = iris.target
```

sklearn에서 iris dataset을 불러온다. X에는 iris의 data들을 넣고 y는 label 값을 넣는다.

```
# X 데이터 정규화
minmaxscalar = MinMaxScaler()
X = minmaxscalar.fit_transform(X)
```

가중치와 바이어스가 제대로 업데이트가 되게 하려면 데이터의 정규화가 필요하다. Iris의 X 최솟값과 최댓값을 이용해 0과 1사이로 반환해주는 minmaxscalar를 사용해 정규화하였다.

---

[13] hojp7874, 교차 엔트로피, 2020.12.12, https://velog.io/@hojp7874/%EA%B5%90%EC%B0%A8-%EC%97%94%ED%8A%B8%EB%A1%9C%ED%94%BC

[14] 호락호락한순무, [DL] 손실함수 (2) - 교차 엔트로피 (cross entropy, 크로스 엔트로피), Kullback-Leibler 발산 손실, 2022. 12. 1, https://gr-st-dev.tistory.com/39

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, random_state=42)
```

sklearn의 'train_test_split' 함수를 이용해 0.4를 test size로 정해 training data와 test data를 나누었다.

```
# 가중치와 편향 초기화
self.w1 = np.random.normal(size=(self.n_input, self.n_hidden1))
self.b1 = np.zeros((1, self.n_hidden1))
self.w2 = np.random.normal(size=(self.n_hidden1, self.n_output))
self.b2 = np.zeros((1, self.n_output))
```

그 다음 numpy 라이브러리의 random.normal을 사용해서 가중치를 초기화해주었다. random.normal은 정규분포로부터 무작위 샘플을 만든다. 그리고 추출을 반복할 때마다 값이 달라진다. 바이어스는 0으로 초기화하였다. 그러나 모델을 실행시켰을 때, 아래 그림처럼 epoch=100일 때부터 loss가 nan이 나오는 것을 알 수 있다.

```
Epoch 0, Loss 17.4689
<ipython-input-12-9729cdef5816>:37: RuntimeWarning: overflow encountered in exp
  exp_z = np.exp(z)
<ipython-input-12-9729cdef5816>:38: RuntimeWarning: invalid value encountered in true_divide
  return exp_z / np.sum(exp_z, axis=1, keepdims=True) #표현 범위를 초과해버려 nan값이 출력
<ipython-input-12-9729cdef5816>:54: RuntimeWarning: divide by zero encountered in log
  loss = -np.mean(y * np.log(o_1))
<ipython-input-12-9729cdef5816>:54: RuntimeWarning: invalid value encountered in multiply
  loss = -np.mean(y * np.log(o_1))
Epoch 100, Loss nan
Epoch 200, Loss nan
Epoch 300, Loss nan
Epoch 400, Loss nan
Epoch 500, Loss nan
Epoch 600, Loss nan
Epoch 700, Loss nan
Epoch 800, Loss nan
Epoch 900, Loss nan
Test Accuracy: 0.4222
```

Runtime warning을 확인해보면 softmax를 계산할 때, exp에서 overflow가 발생했고, loss를 계산할 때, 0이 입력이 들어가 log 계산이 안되는 것을 알 수 있다. 그래서 초기화이후, feedforward 함수 안에서 nan이 발생하는 것으로 인지하고 isnan()함수를 이용해 True를 반환하는 변수를 검사했다.

```
def feedforward(self, X, y):
      print(np.isnan(self.w1))
      input_x = np.dot(X, self.w1) +self.b1
```

```
[[False False  True ... False  True  True]
 [False False  True ... False  True  True]
 [False False  True ... False  True  True]
 ...
 [False False  True ... False  True  True]
 [False False  True ... False  True  True]
 [False False  True ... False  True  True]]
[[False False  True ... False  True  True]
 [False False  True ... False  True  True]
 [False False  True ... False  True  True]
```

결과적으로, w1에서 True가 나왔고 가중치 초기화 방법을 바꿔보았다. 총 5가지 방법을 적용해 보았다.

```
# 가중치와 편향 초기화
#self.w1 = 0.01 * np.random.randn(self.n_input, self.n_hidden1 )
#self.w1 = tf.keras.initializers.HeNormal()
self.w1 = np.random.normal(0,1,size=(self.n_input, self.n_hidden1))
#self.w1 = np.random.uniform(low =0.0, high = 1.0, size=(self.n_input, self.n_hidden1))
```

(Nan이 나온 가중치 초기화 방법들)

```
self.w1 = np.random.randn(self.n_input, self.n_hidden1)/np.sqrt(self.n_input/2)
self.b1 = np.zeros((1, self.n_hidden1))

self.w2 = np.random.randn(self.n_hidden1, self.n_output )/np.sqrt(self.n_hidden1/2)
self.b2 = np.zeros((1, self.n_output))
```

(He 초기화)

최종적으로 He 초기화를 사용하여 해결하였다. He 초기화 방법은 Relu함수와 주로 쓰인다. 입력의 개수가 적을수록 더 작은 값으로 가중치를 나누어준다. 그렇기 때문에 입력이 적으면 가중치는 반대로 큰 값을 가지게 해준다. 이는 입력 개수가 적으면 가중치 값을 크게 하여 한 가중치 값의 영향력을 늘려주고, 반대로 입력의 개수가 많으면 그 영향력을 분산한다.[15] ReLu함수는 0으로 출력되는 값이 많기 때문에 이를 보완하기 위해 '/2'를 해준다. Gradient vanishing 문제를 해결하기 위해 sigmoid에서 relu로 활성함수를 바꿨기 때문에 초기화 방법도 He 초기화 방법으로 바꿔주었다.

[15] Mountain96, [CS231n - Lecture 6] Training Neural Networks - Part I-2, 2021.08.02, https://mountain96.tistory.com/32

```
# 레이블을 one-hot encoding으로 변환
lb = LabelBinarizer()
y_train = lb.fit_transform(y_train)
y_test = lb.transform(y_test)
```

본격적으로 학습을 시작하기 전에 y 들을 one-hot encoding 을 사용해 표현할 레이블에 1 의 값을 부여하고 다른 레이블은 0 을 부여한다.[16]

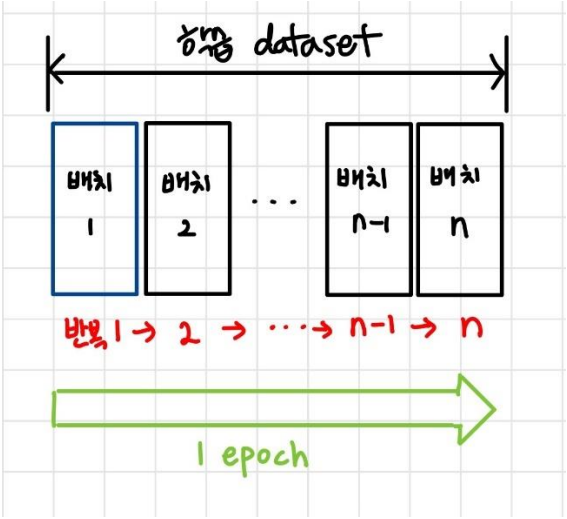
sklearn 모델과 성능 비교를 해보고 다양한 파라미터를 바꿔보면서 성능개선에 도움을 준 파라미터가 있는지 알아보겠다. 정확한 비교를 위해 Epoch 는 모두 1000 으로 진행하였고 학습률은 0.01 로 고정시켜 두었다. 결론/고찰 파트에서 epoch 와 학습률을 바꿔보았다.

```
Iris = load_iris()

X = Iris['data']
y = Iris['target']

X_train, X_test, y_train, y_test =  train_test_split(X,y, test_size=0.4, random_state=42)

scaler = StandardScaler()
scaler.fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

mlp = MLPClassifier(hidden_layer_sizes=(128), learning_rate_init=0.01, max_iter=1000)
mlp.fit(X_train, y_train)

y_pred = mlp.predict(X_test)
accuracy_sklearn = accuracy_score(y_test, y_pred)

print("Accuracy:", accuracy_sklearn)
```

```
Accuracy: 0.9833333333333333
```

sklearn 의 모델은 0.9833 의 정확도가 나왔다.

스토캐스틱 경사하강법은 샘플을 섞어 뽑아서 학습을 진행한다. 스토캐스틱은 size 가 1 이다. 미니 배치는 size 가 0, 1 이 아니며, 아래 그림과 같은 흐름으로 진행된다.



[16] 안상준, 유원준, 딥 러닝을 이용한 자연어 처리 입문, https://wikidocs.net/22647

```
def backpropagation(self, X, y, epochs, lr):

    batches =X_train.shape[0] // self.size

    for epoch in range(epochs):

        #섞어서 하나 뽑기
        shuffle = np.random.permutation(len(X))
        X = X[shuffle]
        y = y[shuffle]

        for batch in range(batches) :
          start = batch * self.size
          end = (batch+1) * self.size

          X_batch = X[start:end]
          y_batch = y[start:end]

          o_1, input_x= self.feedforward(X_batch,y_batch) #feedfoward한 값을 받아온다.
```

X.shpae[0]를 배치 사이즈 크기로 나눈 몫을 batches 로 저장하고, np.random.permutation 을 사용해 무작위로 섞이게 한다. 이렇게 하면 사이즈 크기만큼 데이터를 뽑아 feedforward 할 수 있다.

1. **히든 레이어 1 층**(노드 개수:128), 활성함수: relu 와 softmax, 스토캐스틱(size = 1)

```
# MLP 모델 생성
mlp = MLP(n_input=4, n_hidden1=128, n_output=3, size=1)
```

128 개의 노드 수로 단 하나의 은닉층만을 생성해 학습을 진행하였다.

```
Epoch 0, Loss 0.2471
Epoch 100, Loss 0.0010
Epoch 200, Loss 0.0003
Epoch 300, Loss 0.0039
Epoch 400, Loss 0.0000
Epoch 500, Loss 0.0007
Epoch 600, Loss 0.3532
Epoch 700, Loss 0.0001
Epoch 800, Loss 0.1078
Epoch 900, Loss 0.0000
Test Accuracy: 0.9833
```

Sklearn 과 같은 설정으로 해주었다. 정확도는 0.9833 이 나왔으며 sklearn 모델과 같은 정확도임을 알 수 있다. 그런데 중간에 loss 가 갑자기 증가하는 경우가 있다. Epoch 400 에서 매우 낮은 loss 값이 나오다가 다시 증가한다. 이것은 overfitting 의 가능성이 있다. 그렇기 때문에 노드의 개수를 늘릴 필요가 없다.

Epoch 0, Loss 0.0008
Epoch 10, Loss 0.1100
Epoch 20, Loss 0.0747
Epoch 30, Loss 0.0000
Epoch 40, Loss 0.0345

Epoch 30 에서 loss 가 0 이 나오는 것을 보면 학습을 잘 하고 있다는 것을 알 수 있지만, 그 다음 epoch 에서 loss 가 다시 증가하였다. 모델이 과적합 되었다는 의미로 받아들였고, epoch 의 수를 줄여보고 학습률을 조절해보았다.

```
# MLP 모델 생성
mlp = MLP(n_input=4, n_hidden1=128, n_output=3, size=1)

# 모델 학습
mlp.backpropagation(X_train, y_train, epochs=50, lr=0.01)

# 모델 테스트
y_pred = mlp.predict(X_test, y_test)

# 정확도 계산
y_test_true = np.argmax(y_test, axis=1)
accuracy = np.mean(y_pred == y_test_true)
print(f'Test Accuracy: {accuracy:.4f}')
```

```
Epoch 0, Loss 0.0008
Epoch 10, Loss 0.1100
Epoch 20, Loss 0.0747
Epoch 30, Loss 0.0000
Epoch 40, Loss 0.0345
Test Accuracy: 0.6833
```

위 사진은 epoch 를 50 으로 줄인 시도이다. 그러나, 똑같이 loss 가 epoch 40 에서 증가함을 알 수 있었다.

```
# MLP 모델 생성
mlp = MLP(n_input=4, n_hidden1=128, n_output=3, size=1)

# 모델 학습
mlp.backpropagation(X_train, y_train, epochs=1000, lr=1)

# 모델 테스트
y_pred = mlp.predict(X_test, y_test)

# 정확도 계산
y_test_true = np.argmax(y_test, axis=1)
accuracy = np.mean(y_pred == y_test_true)
print(f'Test Accuracy: {accuracy:.4f}')
```

```
Epoch 0, Loss 0.2409
Epoch 100, Loss 0.7798
Epoch 200, Loss 0.0000
Epoch 300, Loss 0.4774
Epoch 400, Loss 0.7003
Epoch 500, Loss 0.0000
Epoch 600, Loss 0.0588
Epoch 700, Loss 0.1125
Epoch 800, Loss 0.4116
Epoch 900, Loss 0.5529
Test Accuracy: 0.6833
```

학습률을 1 로 바꿨을 때, loss 가 튀고 정확도가 오히려 줄었다.

```
if loss < 0.000001:
                break
```

이번에는 loss 가 너무 낮아지면 overfitting 의 가능성이 높아지기 때문에 break 문을 써주었다. 하지만, 학습이 충분이 되지 않아서 0.68 의 낮은 정확도가 나옴을 알 수 있었다.

2. **히든 레이어 1 층**(노드 개수:128), 활성함수: relu 와 softmax, 미니배치(size = 5)

```
Epoch 0, Loss 0.3361
Epoch 100, Loss 0.0953
Epoch 200, Loss 0.0379
Epoch 300, Loss 0.0628
Epoch 400, Loss 0.0245
Epoch 500, Loss 0.0463
Epoch 600, Loss 0.0062
Epoch 700, Loss 0.0241
Epoch 800, Loss 0.0446
Epoch 900, Loss 0.0020
Test Accuracy: 0.9667
```

사이즈 크기를 1 에서 5 로 늘렸을 때, 정확도가 0.9667 로 줄어들었다. 따라서, 첫번째 은닉층의 노드 개수가 128 일 때, size 가 1 인 경우가 정확도가 더 높았다.

3. **히든 레이어 2 층**(노드 개수:256, 128), 활성함수: relu 와 softmax, 스토캐스틱 (size=1)

```
def feedforward(self, X, y):

      input_x = self.feed(X, self.w1, self.b1)
      input_x = self.sigmoid(input_x)

      zsum = self.feed(input_x, self.w2, self.b2)
      z_1 = self.sigmoid(zsum)

      osum = self.feed(z_1, self.w3, self.b3)
      o_1 = self.softmax(osum)

      return o_1, input_x, z_1
```

```
Test Accuracy: 0.9833
```

은닉층을 2 개로 늘리면 0.9833 의 정확도가 나옴을 알 수 있다.

4. **히든 레이어 2 층**(노드 개수:256, 128), 활성함수: relu 와 softmax, 미니배치(size=5)

```
 # MLP 모델 생성
mlp = MLP(n_input=4, n_hidden1=256, n_hidden2=128, n_output=3, size=5)
```

```
Test Accuracy: 0.9833
```

Size = 1 일 때와 같은 0.9833 의 정확도가 나옴을 알 수 있다.

결과적으로, sklearn 모델보다 더 높은 정확도가 나온 설정은 없었다. 그렇지만 2 번과 같이 sklearn 모델보다 낮은 정확도가 나온 조건이 있었다.

추가적으로, sigmoid 함수에서 relu 함수로 바꾸게 된 이유는 gradient vanishing 문제였다. 그렇다면 각각 relu 함수와 sigmoid 함수를 사용했을 때, 정확도 면에서 어떤 차이가 있는지 궁금해졌다.

1. **히든 레이어 1 층**(노드 개수:128), 스토캐스틱(size = 1)

Relu	sigmoid
0.9833	**0.9833**

2. **히든 레이어 1 층**(노드 개수:128), 미니배치(size = 5)

Relu	sigmoid
0.9667	**0.9833**

3. **히든 레이어 2 층**(노드 개수:256, 128), 스토캐스틱 (size=1)

Relu	sigmoid
0.9833	0.9667

4. **히든 레이어 2 층**(노드 개수:256, 128), 미니배치(size=5)

Relu	sigmoid
0.9833	**0.9833**

6 번과 7 번에서 정확도가 다른 것을 알 수 있다. relu 함수는 size 가 1 일 때, 공통적으로 0.9833 이 나왔다. sigmoid 함수는 size 가 5 일 때, 공통적으로 0.9833 이 나온 것을 알 수 있었다.

mnist 데이터를 사용한 실험

```
mnist = fetch_openml('mnist_784', cache=False)
mnist.data.shape

X, y = mnist["data"], mnist["target"]
```

mnist 는 sklearn 에서 불러오고 X, y 에 각각의 data 를 넣어준다.

```
# 모델 학습
mlp.backpropagation(X_train, y_train, epochs=50, lr=0.01)
```

mnist data 를 학습시킬 때, epoch 를 50 으로 제한하고 학습률을 0.01 로 설정하고 다른 파라미터를 수정하였다.

```
# X 데이터 정규화
minmaxscalar = MinMaxScaler()
X = minmaxscalar.fit_transform(X)
```

X 는 minmaxscalar 를 사용해 정규화한다.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

test data 의 size 를 0.2 로 해준다.

```
# 레이블을 one-hot encoding으로 변환
lb = LabelBinarizer()
y_train = lb.fit_transform(y_train)
y_test = lb.transform(y_test)
X_train = X_train.reshape(-1, 784) / 255.0
X_test = X_test.reshape(-1, 784) / 255.0

# MLP 모델 생성
mlp = MLP(n_input=X_train.shape[1], n_hidden1=128, n_output=10, size=1)
```

Mnist 의 data 는 28x28 크기의 2 차원 배열이기 때문에 784 크기의 1 차원 배열로 reshape 해주었다. 또한, 이미지를 255 로 나눠 0 과 1 사이의 값으로 정규화하였다.

sklearn 모델과 성능 비교를 해보고 다양한 파라미터를 바꿔보면서 성능개선에 도움을 준 파라미터가 있는지 알아보겠다. 정확한 비교를 위해 Epoch 는 모두 50 으로 진행하였고 학습률은 0.01 로 고정시켜 두었다.

```
mnist = fetch_openml('mnist_784', cache=False)
mnist.data.shape

X, y = mnist["data"], mnist["target"]


X_train, X_test, y_train, y_test =  train_test_split(X,y, test_size=0.2, random_state=42)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

scaler.fit(X_train)

X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

mlp = MLPClassifier(hidden_layer_sizes=(128), learning_rate_init=0.01, max_iter=50)
mlp.fit(X_train, y_train)

# 테스트 데이터로 예측을 수행하고 정확도를 계산합니다.
y_pred = mlp.predict(X_test)
accuracy_sklearn = accuracy_score(y_test, y_pred)

# 결과를 출력합니다.
print("Accuracy:", accuracy_sklearn)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/datasets/_openml.py:968: FutureWarning: The default value of `parser` wil
  warn(
Accuracy: 0.9705714285714285
```

sklearn 의 모델은 0.9705 의 정확도가 나왔다.

1. **히든 레이어 1 층**(노드 개수:128), 활성함수: relu 와 softmax, 스토캐스틱(size = 1)

```
Epoch 0, Loss 0.2280
Epoch 10, Loss 0.0733
Epoch 20, Loss 0.0499
Epoch 30, Loss 0.0008
Epoch 40, Loss 0.0014
Test Accuracy: 0.8511
```

정확도는 0.8511 이 나왔다. sklearn 모델에 비해 현저히 낮은 정확도가 나온 것을 알 수 있다.

2. **히든 레이어 1 층**(노드 개수:128), 활성함수: relu 와 softmax, 미니배치(size = 5)

```
Epoch 0, Loss 0.2231
Epoch 10, Loss 0.2269
Epoch 20, Loss 0.1039
Epoch 30, Loss 0.1376
Epoch 40, Loss 0.0631
Test Accuracy: 0.8471
```

1 번에서 상대적으로 낮은 정확도가 나왔기 때문에 배치 사이즈를 5 로 늘려보았다. 정확도는 0.8471 이 나온 것을 알 수 있다. 1 번보다 조금 더 낮은 정확도가 나왔다.

3. **히든 레이어 1 층**(노드 개수:256), 활성함수: relu 와 softmax, 스토캐스틱(size = 1)

```
Epoch 0, Loss 0.2146
Epoch 10, Loss 0.0370
Epoch 20, Loss 0.0014
Epoch 30, Loss 0.0003
Epoch 40, Loss 0.0012
Test Accuracy: 0.8870
```

1 번과 은닉층의 수는 같지만 노드 수를 늘렸더니 조금 더 높은 정확도가 나온 것을 알 수 있다.

4. 히든 레이어 1 층(노드 개수: 256), 활성함수: relu 와 softmax, 미니배치(size = 5)

```
Epoch 0, Loss 0.2296
Epoch 10, Loss 0.1904
Epoch 20, Loss 0.1026
Epoch 30, Loss 0.1179
Epoch 40, Loss 0.0492
Test Accuracy: 0.8469
```

2 번과 거의 비슷한 정확도가 나왔음을 알 수 있다.

5. 히든 레이어 2 층(노드 개수:256, 128), 활성함수: relu 와 softmax, 스토캐스틱 (size=1)

```
Epoch 0, Loss 0.2218
Epoch 10, Loss 0.0004
Epoch 20, Loss 0.0045
Epoch 30, Loss 0.0021
Epoch 40, Loss 0.0050
Test Accuracy: 0.9076
```

이전 조건들에 비해 정확도가 많이 향상된 것을 알 수 있다. 조금 더 복잡한 모델이라 학습이 잘 된 것 같다.

6. 히든 레이어 2 층(노드 개수:256, 128), 활성함수: relu 와 softmax, 미니배치(size=5)

```
Epoch 0, Loss 0.2286
Epoch 10, Loss 0.0878
Epoch 20, Loss 0.1480
Epoch 30, Loss 0.0104
Epoch 40, Loss 0.0101
Test Accuracy: 0.8774
```

Size 를 5 로 늘렸을 때, 더 낮은 정확도가 나왔다.

결과적으로, 5 번에서 정확도가 0.9076 으로 가장 높게 나온 것을 알 수 있다.

3. 결론/고찰

Iris data 는
[히든 레이어 1 층(노드 개수:128), 활성함수: relu 와 softmax, 스토캐스틱(size = 1)],
[히든 레이어 2 층(노드 개수:256, 128), 활성함수: relu 와 softmax, 스토캐스틱 (size=1)],
[히든 레이어 2 층(노드 개수:256, 128), 활성함수: relu 와 softmax, 미니배치(size=5)]
위 세 조건에서 0.9833 으로 정확도가 가장 높았다.

Mnist data 는 [**히든 레이어 2 층**(노드 개수:256, 128), 활성함수: relu 와 softmax, 스토캐스틱 (size=1)] 조건에서 정확도 0.9076 으로 제일 높은 정확도가 나왔다.

모델을 구조에 대해서 은닉층의 개수와 각 층의 뉴런 수를 조정함으로써 모델의 복잡도가 어떤 영향을 끼치는 지 알 수 있었다.
iris 에서는 size 가 **5** 일 때, 은닉층이 1 개일 때보다 2 개일 때가 정확도가 더 높게 나왔다.

은닉층 1 개 (128 개 노드)	은닉층 2 개 (256, 128)
0.9667	0.9833

Sklearn 과 직접 구현을 한 model 중 제일 높은 조건과 비교를 해보면 아래와 같이 같은 정확도를 가진 것을 알 수 있다.

sklearn	구현 model
0.9833	0.9833

하이퍼 파라미터 등을 바꿔보면서 sklearn model 보다 더 좋은 성능을 구현하진 못했지만, 더 낮은 경우를 알게 되면서 더 복잡한 구조가 더 효과가 있었음을 알게 되었다.

mnist 에서는 size 가 **1** 일 때, 은닉층이 더 복잡할수록 정확도가 더 높게 나왔다.

은닉층 1 개 (128 개 노드)	은닉층 1 개 (256 개 노드)	은닉층 2 개 (256, 128)
0.8511	0.8870	0.9076

Sklearn 과 직접 구현을 한 model 중 제일 높은 조건과 비교를 해보면 아래와 같이 같은 정확도를 가진 것을 알 수 있다.

sklearn	구현 model
0.9705	0.9076

Mnist 는 처음 조건에서 굉장히 낮은 정확도를 보여줬기 때문에 다른 파라미터를 수정하면서 0.9076 까지 정확도를 높일 수 있었다.

다양한 활성함수 중, sigmoid 함수대신 Relu 를 선택한 이유는 sigmoid 함수는 gradient vanishing 문제를 야기하기 때문이다.
직접 relu 함수와 sigmoid 함수를 비교해보면서 두 함수의 미분값이 gradient 계산에 어떤 영향을 끼치는 지 알게 되었다. Sigmoid 는 미분했을 때, 0~0.25 사이의 값이기 때문에 출력층과 멀어질수록 역전파가 잘 진행되지 않는다.

iris data 를 사용해 각각 두 활성함수를 적용해 정확도를 비교해보았을 때, 큰 차이는 존재하지 않았다. relu 함수는 size 가 1 일 때, 공통적으로 0.9833 이 나왔다. sigmoid 함수는 size 가 5 일 때, 공통적으로 0.9833 이 나온 것을 알 수 있었다.

그러나 mnist data 를 사용해 비교해 보았을 때,
[히든 레이어 2 층(노드 개수:256, 128), 스토캐스틱 (size=1)] 조건에서 큰 차이가 있음을 알게 되었다. 아래는 relu 함수에서 sigmoid 함수로 바꿔서 실행한 결과이다.

```
/usr/local/lib/python3.10/dist-packages/sklearn/datasets/_openml.py:968: FutureWarning: The default value of `parser`
  warn(
Epoch 0, Loss 0.2231
Test Accuracy: 0.1143
```

sigmoid	relu
0.1143	0.9076

relu 함수가 훨씬 더 높은 정확도를 보여주었다.

또한, Loss 가 nan 값이 나오는 이유에 대해서 알게 되었다.

1. Softmax 에 너무 큰 입력값이 들어갔을 경우

```
def softmax(self, z):
        z_max = np.max(z)
        exp_z = np.exp(z-z_max)
        y = exp_z / np.sum(exp_z, axis=1, keepdims=True)
        return y
```

Max 함수를 사용해서 최댓값을 뺐다.

2. 가중치 초기화 방법이 잘못되었을 경우
Isnan() 함수를 사용해서 가중치 초기화에서 nan 이 있음을 알게 되었다. 따라서, 여러 가중치 초기화 방법을 적용해보았다.

만약 가중치 초기화를 모두 0 으로 하게 되면 모든 뉴런이 같은 일만 하게 된다. 그렇기 때문에 0 으로 초기화할 수는 없다. 총 5 가지의 초기화 방법으로 수정해보았다.

```
# 가중치와 편향 초기화
#self.w1 = 0.01 * np.random.randn(self.n_input, self.n_hidden1 )
#self.w1 = tf.keras.initializers.HeNormal()
self.w1 = np.random.normal(0,1,size=(self.n_input, self.n_hidden1))
#self.w1 = np.random.uniform(low =0.0, high = 1.0, size=(self.n_input, self.n_hidden1))
```

가중치 초기화 범위를 0~1 로 설정해보거나 random.randn 을 사용한 값 x 0.01 을 해주는 방법도 적용해보았다. 그러나 아래 사진처럼 nan 이 계속 출력되었다.

```
Epoch 0, Loss 17.4689
<ipython-input-12-9729cdef5816>:37: RuntimeWarning: overflow encountered in exp
  exp_z = np.exp(z)
<ipython-input-12-9729cdef5816>:38: RuntimeWarning: invalid value encountered in true_divide
  return exp_z / np.sum(exp_z, axis=1, keepdims=True) #표현 범위를 초과해버려 nan값이 출력
<ipython-input-12-9729cdef5816>:54: RuntimeWarning: divide by zero encountered in log
  loss = -np.mean(y * np.log(o_1))
<ipython-input-12-9729cdef5816>:54: RuntimeWarning: invalid value encountered in multiply
  loss = -np.mean(y * np.log(o_1))
Epoch 100, Loss nan
Epoch 200, Loss nan
Epoch 300, Loss nan
Epoch 400, Loss nan
Epoch 500, Loss nan
Epoch 600, Loss nan
Epoch 700, Loss nan
Epoch 800, Loss nan
Epoch 900, Loss nan
Test Accuracy: 0.4222
```

최종적으로 Relu 함수와 주로 같이 쓰이는 He 초기화 방법을 사용하였다.

```
# 가중치와 편향 초기화
self.w1 = np.random.randn(self.n_input, self.n_hidden1)/np.sqrt(self.n_input/2)
self.b1 = np.zeros((1, self.n_hidden1))

self.w2 = np.random.randn(self.n_hidden1, self.n_hidden2 )/np.sqrt(self.n_hidden1/2)
self.b2 = np.zeros((1, self.n_hidden2))

self.w3 = np.random.randn(self.n_hidden2, self.n_output )/np.sqrt(self.n_hidden2/2)
self.b3 = np.zeros((1, self.n_output))
```

3. 학습률이 너무 높은 경우
   학습률을 바꿔보면서 진행했을 경우, 실제로 nan 값이 나오지 않았지만 정확도가 매우 낮게 나왔다.

Loss nan 문제를 해결하면서 다시 마주하게 된 문제는 loss 가 증가하거나 진동하는 문제였다. loss 가 줄지 않는 문제를 다양한 방법(1. Epoch 줄이기, 2. 학습률 조절하기, 3. loss 제한하기)으로 고치기 위해 시도했지만 해결하지 못해 아쉬움이 남는다.

다양한 파라미터, 활성함수, 가중치 초기화 등을 바꿔보면서 sklearn model 보다 더 좋은 성능을 구현하는 model 을 만들고 싶었지만, iris 는 같은 정확도, mnist 는 더 낮은 정확도를 보여주었다. Model 을 구현하기 위해 구글링을 하면서 어떤 활성함수가 많이 쓰이고, 그 이유는 무엇인지를 알게 되었고 과적합을 만드는 요인이 무엇인지 알게 되었다. 어떤 model 이 더 성능이 좋을지 고민해보면서 어떤 파라미터가 성능을 높이고 줄이는 지 직접 경험할 수 있었다.

참고문헌

Chat gpt

Douglas' Space, 하이퍼 파라미터(Hyper Parameters)란?, 2022.04.28, https://doug.tistory.com/m/44

hojp7874, 교차 엔트로피, 2020.12.12,

https://velog.io/@hojp7874/%EA%B5%90%EC%B0%A8-%EC%97%94%ED%8A%B8%EB%A1%9C%ED%94%BC

Mountain96, [CS231n - Lecture 6] Training Neural Networks - Part I-2, 2021.08.02, https://mountain96.tistory.com/32

Tony Park, [딥러닝] 기울기 소실(Vanishing Gradient)의 의미와 해결방법, 2022. 5. 22, https://heytech.tistory.com/388

고영민. "인공지능 심층신경망의 기울기 소실 문제 3가지 유형 분류와 파라메트릭 활성함수를 사용한 완화 방법." 국내석사학위논문 전주대학교 일반대학원, 2022.

만년필잉크, 딥 러닝-3.1. 활성화함수(2)-시그모이드 함수, 2021.1.25, https://gooopy.tistory.com/52

안상준, 유원준, 딥 러닝을 이용한 자연어 처리 입문, https://wikidocs.net/22647

유원준, 안상준, PyTorch로 시작하는 딥 러닝입문, https://wikidocs.net/60324.

이지항, 강의자료 WO3.

호락호락한순무,
[DL] 손실함수 (2) - 교차 엔트로피 (cross entropy, 크로스 엔트로피), Kullback-Leibler 발산 손실, 2022. 12. 1, https://gr-st-dev.tistory.com/39.