

Homework 2 - Perceptron

32184801 하승원

0 Introduction

이번 과제에서는 주어진 데이터를 활용하여서 기본적인 Neural Network를 직접 구현을 해보는 과제입니다. 이번 과제를 통해서 Neural Network 에 대한 기본적인 프로세스를 익히고 더 나아가 MLP(Multi Layer Perceptron) 에 대한 실험까지도 진행을 하였습니다.

1 Related Work

1-1 Neural

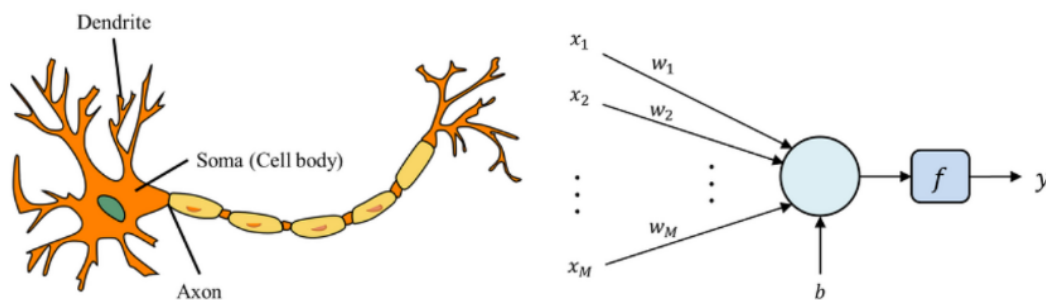


그림 1 Neural & Perceptron

퍼셉트론은 인공신경망의 초기 형태로, 인간의 뉴런 구조를 모방하여 설계된 방식입니다. 기본적으로는 생물학적 뉴런이 전기적 신호를 받아들이고 일정 임계값을 넘어가면 다음 뉴런으로 신호를 전달하는 원리를 따르고 있습니다. 퍼셉트론에서는 각 노드가 이전 노드에서 전달받은 입력값에 가중치를 곱하여 계산하고, 이를 활성화 함수를 통해 처리한 후 다음 노드로 전달합니다.

이와 같은 구조는 뇌의 뉴런 간의 상호 작용을 모방함으로써 기계 학습 및 패턴 인식 등의 과제에 응용됩니다. 퍼셉트론은 입력값을 적절한 가중치와 활성화 함수를 활용하여 출력값을 생성하는데, 이를 통해 모델이 학습하고 판단할 수 있는 능력을 갖추게 됩니다. 이러한 퍼셉트론의 개념은 후에 다양한 인공신경망 모델의 기초가 되었습니다.

1-2 Single Perceptron

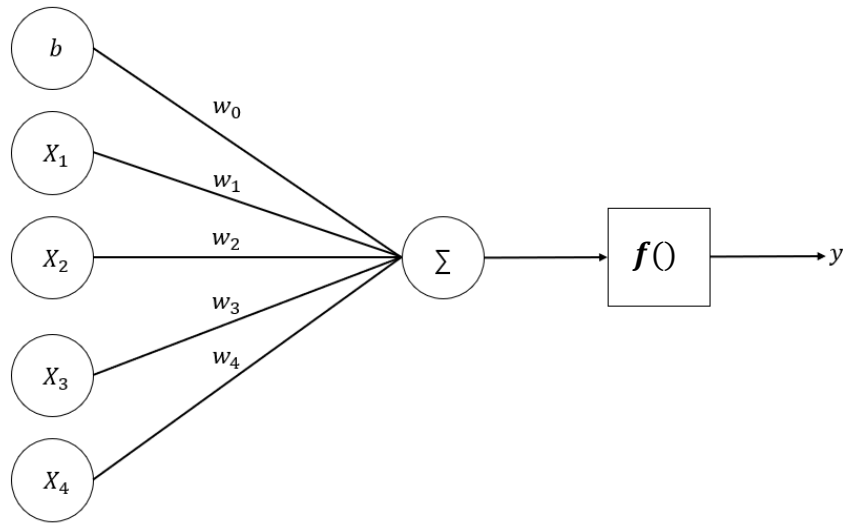


그림 2 Perceptron

그림 2 와 같이 퍼셉트론은 여러 개의 입력값을 이전 노드에서 전달받고 이값을 가중치(w) 값들과 각각 곱해져 다음 노드로 전달합니다. 그 이후에는 활성화 함수(f) 를 통해서 다음 노드로 값을 전달하게 됩니다.

Mathematical Expression Perceptron

여기서 입력값이 들어가는 부분을 입력층 그리고 결과가 나오는 부분을 출력층 이라고 합니다.

$$\begin{aligned} f(w_0b + w_1X_1 + w_2X_2 + w_3X_3 + w_4X_4) &\geq \theta \quad (\text{if } y = 1) \\ f(w_0b + w_1X_1 + w_2X_2 + w_3X_3 + w_4X_4) &< \theta \quad (\text{if } y = 0) \end{aligned}$$

주어진 수식은 퍼셉트론의 기본 형태를 나타내고 있습니다. 여기서 f 는 활성화 함수를 나타내며, w_0 은 bias 에 해당하는 가중치, w_1, w_2, w_3, w_4 는 각각의 입력 변수 x_1, x_2, x_3, x_4 에 대한 가중치를 나타냅니다. θ 는 임계값(threshold)을 나타내며, 이 임계값을 넘으면 출력 y 는 1 이 됩니다.

Multi Layer Perceptron 의 발전

OR 연산을 통해 퍼셉트론이 두 개의 입력 값에 대한 분류를 수행한다고 가정해봅시다. 입력 변수 x_1 과 x_2 에 따라 출력 y 가 다음과 같이 결정됩니다

X1	X2	y
0	0	0
0	1	1
1	0	1
1	1	1

두 개의 입력값에 대하여 기존 perceptron 연산을 통해서 결괏값이 0 혹은 1로 결과가 나오는 것을 확인할 수 있습니다. 아래의 그림 3의 OR 연산의 그래프처럼 직접 시각화를 하여 확인하면 classifier를 통해서 y 값의 결과를 도출하는 것이 가능하다는 것을 확인할 수 있습니다.

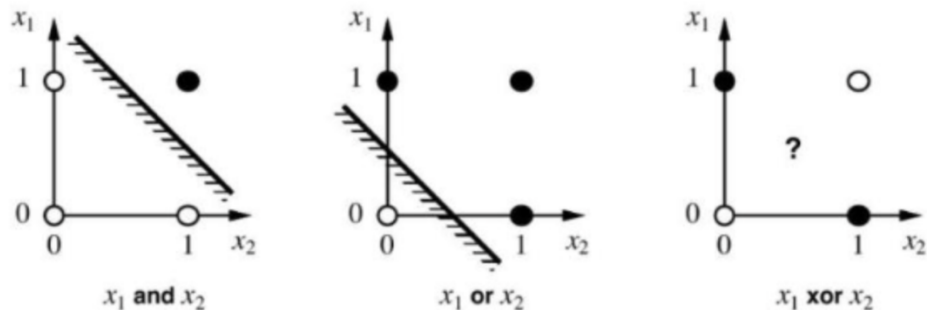


그림 3 OR & XOR 그래프 시각화

그러나 XOR 연산과 같은 비선형 문제를 해결하기 위해서는 단일 층의 퍼셉트론만으로는 한계가 있습니다. XOR 연산의 경우, 하나의 직선으로 올바른 결과를 도출하는 것이 불가능하며, 이를 해결하기 위해 다층 퍼셉트론(multi-layer perceptron)이 도입되었습니다.

1-3 Multi-Layer Perceptron

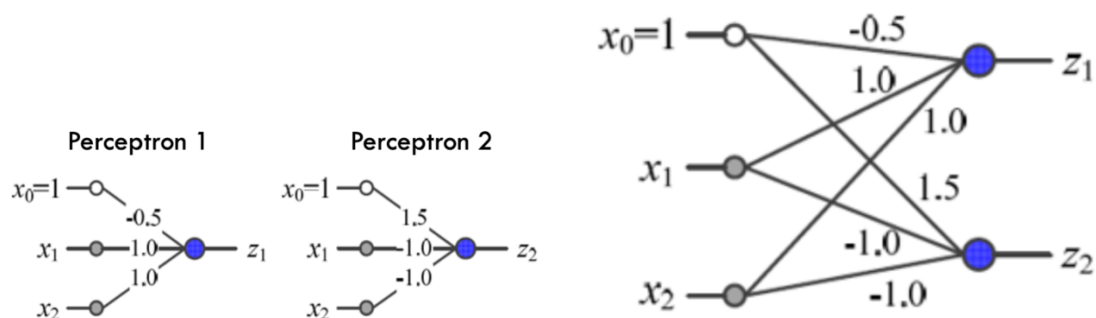


그림 4 Multi-Layer Perceptron 시각화

Multi-Layer Perceptron(MLP)은 기존의 Single Perceptron을 여러 개 연결하여 구성하는 인공 신경망의 한 유형입니다. Single Perceptron은 입력을 받아 가중치와 곱하고 활성화 함수를 통과한 결과를 출력하는 간단한 모델이었습니다. 그러나 Single Perceptron은 선형적인 판별 경계만을 학습할 수 있어서 비교적 간단한 문제에만 적용할 수 있었습니다.

MLP는 여러 개의 Single Perceptron을 쌓아 올려 비선형적인 판별 경계를 학습할 수 있도록 한 것입니다. 각각의 Perceptron은 하나의 뉴런으로 생각할 수 있으며, 이들 뉴런은 여러 층(layer)에 배치됩니다. 각 층은 여러 개의 뉴런으로 구성되어 있고, 한 층의 뉴런들은 이전 층의 뉴런들과 연결되어 있습니다.

그림 4 에서 보여지는 것처럼, 2 개의 Perceptron 이 쌓여 있는 경우를 예로 들면, 입력이 첫 번째 Perceptron 에 들어가고 그 결과가 두 번째 Perceptron 으로 전달됩니다. 각 Perceptron 은 가중치와 활성화 함수를 가지며, 이를 통해 입력에 대한 복잡한 비선형 관계를 모델링할 수 있습니다.

여러 개의 Perceptron 을 여러 층에 걸쳐 쌓으면서, 모델은 더 복잡하고 추상화된 특징을 학습할 수 있게 됩니다. 이러한 층을 추가하면서 모델은 입력과 출력 간의 복잡한 관계를 모델링하고 데이터의 다양한 특징을 효과적으로 학습할 수 있습니다.

MLP 는 각 층에서의 가중치를 학습하여 입력과 출력 간의 관계를 최적화합니다. 학습은 일반적으로 역전파(backpropagation) 알고리즘을 사용하여 이루어지며, 손실 함수를 최소화하는 방향으로 가중치를 조정합니다. 이러한 방식으로 MLP 는 다양한 복잡한 문제를 해결할 수 있는 강력한 모델이 됩니다.

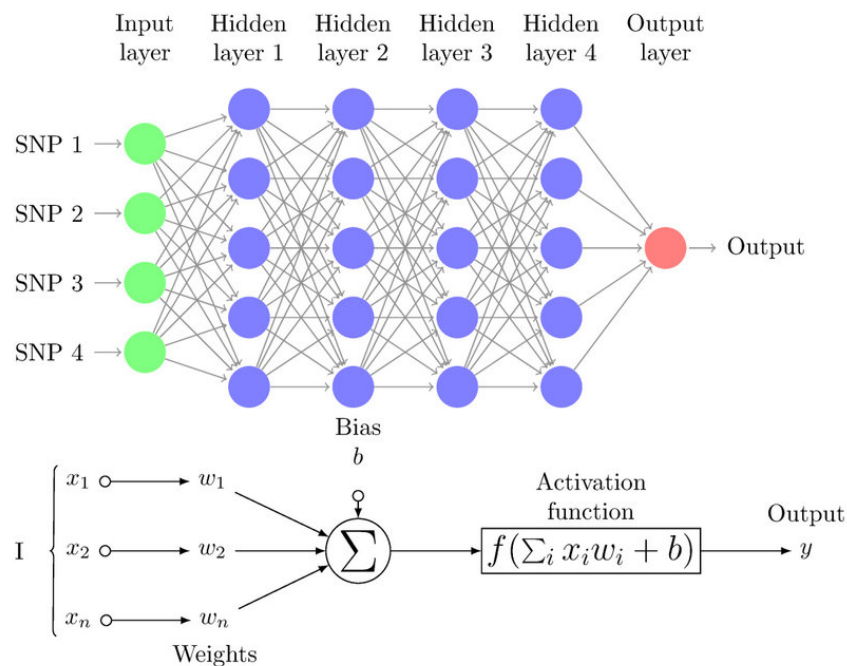


그림 5 MLP 전체 시각화

1-4 Activation Function

Perceptron 에서 input 값과 weight 를 곱한 값의 총 합을 구한 이후에 그 값은 Activation Function(활성화 함수)을 지나게 됩니다. 활성화 함수의 목적은 3 가지 정도가 있습니다.

Large derivative value

MLP(Multi-Layer Perceptron) 구조는 여러 층으로 이루어진 구조입니다. 각 층은 여러 weight 값을 포함하며, 이러한 weight 값을 업데이트하는 과정에서는 cost 함수를 weight 에 대해 미분한 값을 계산하고, 이 값을 learning rate 와 곱하여 기존 weight 에서 빼는 연산을 수행합니다.

그러나 이러한 과정에서 발생하는 문제 중 하나가 gradient vanishing problem 입니다. 이는 기존에 존재하는 weight 값이 계속해서 작아져서 거의 0 에 가까워지는 현상을 나타냅니다.

이러한 문제를 해결하기 위해 Activation Function 이 사용됩니다. Activation Function 은 각 뉴런의 출력을 결정하는 함수로, 비선형성을 도입하여 모델이 다양하고 복잡한 패턴을 학습할 수 있도록 합니다. 특히, Activation Function 의 비선형성은 gradient vanishing problem 을 완화시켜줍니다. 선형 Activation Function 을 사용하면 여러 층의 조합도 결국 선형 변환으로 표현되어 문제를 해결하는 데 한계가 있습니다. 따라서 비선형 Activation Function 을 사용하여 뉴런의 출력에 비선형성을 부여하여, 역전파 과정에서 그래디언트 소실 문제를 완화하고 학습의 안정성을 향상시킵니다.

Easy to compute the derivate

Activation Function 을 사용하면 weight 를 업데이트하는 과정에서 미분값을 계산하는 과정이 간편해지기 때문에 파라미터를 업데이트하는 과정이 효율적으로 이루어집니다. 이는 모델이 학습 중에 빠르게 수렴하고 높은 효율성을 보이게 하는 중요한 특성입니다.

MLP 나 다른 딥러닝 모델을 학습하는 과정에서는 주로 오차(또는 손실) 값을 활용하여 그래디언트(gradient)를 구하고, 이를 사용하여 모델의 파라미터를 업데이트합니다. Activation Function 이 효율적으로 미분 가능하다면, 오차를 역전파하여 그래디언트를 계산하는 과정이 효율적으로 이루어집니다.

Activation Function 의 효율성은 특히 딥러닝 모델에서 계산 비용을 줄이고 학습 속도를 높이는 데에 중요한 역할을 합니다. 따라서 적절한 Activation Function 을 선택하면 모델의 학습이 빠르고 효율적으로 수행될 수 있습니다.

Zero-centered

만약 활성화 함수의 출력이 항상 양수이거나 항상 음수일 경우, 그래디언트가 가중치를 항상 같은 방향으로 밀 수 있으며, 이는 훈련 중 수렴이 느리거나 비효율적일 수 있습니다.

"Zero-centered" 활성화 함수를 사용하면 출력이 0 주변에 집중되어 있기 때문에, 훈련 중에 매개변수(가중치)가 항상 같은 방향으로 지속적으로 업데이트되는 것을 방지할 수 있습니다. 이는 최적화 알고리즘이 효과적으로 매개변수를 업데이트할 수 있도록 도와줍니다. 이는 훈련 중에 빠른 수렴을 도와줍니다.

그림 6 의 시그모이드 함수를 예로 들면, 이 함수는 zero-center 를 하지 않은 활성화 함수입니다. 이러한 함수를 사용하면 그래디언트 값을 계산하면 항상 양수가 되어, 가중치를 업데이트하는 과정에서 활성화 함수의 미분값이 항상 양수가 됩니다. 이는 가중치를 계속해서 한 방향으로 업데이트하게 만들어, 학습 과정에서 가중치가 왔다 갔다(zig-zag)하는 문제를 발생시킬 수 있습니다.

요약하면, zero-centered 활성화 함수를 사용하지 않으면 가중치 업데이트가 항상 양수 방향으로 치우치는 문제가 발생할 수 있고, 이는 학습을 불안정하게 만들어 가중치가 업데이트될 때마다 왔다 갔다하는 현상을 유발할 수 있습니다.


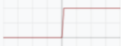

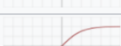

Name	Plot	Equation	Derivative (with respect to x)
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a. Sigmoid or Soft step)		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$ [1]	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})}$	$f'(x) = 1 - f(x)^2$
Rectified linear unit (ReLU) [12]		$f(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ x & \text{for } x > 0 \end{cases} = \max\{0, x\} = x \mathbf{1}_{x>0}$	$f'(x) = \begin{cases} 0 & \text{for } x \leq 0 \\ 1 & \text{for } x > 0 \end{cases}$

그림 6 Activation Function 예시

1-5 Backpropagation

Backpropagation 은 딥러닝 모델에서 모델을 구성하는 파라미터를 학습하는 과정에서 가장 중요한 이론 중 하나입니다. 데이터를 딥러닝 모델에 Feed Forward 을 통해서 데이터를 먼저 흘려보내는 과정 이후에 Backpropagation 과정이 이루어 집니다. 이 과정에서 최종적으로 모델이 예측하는 예측값과 정답을 비교하여 error 를 구하고 뒤로 전달하면서 그 값에 대한 gradient 값들을 각각의 layer 에 저장하는 과정으로 진행됩니다. 그 이후에 그 gradient 값을 기준으로 연결되어 있는 weight 값을 업데이트를 시켜줍니다. 이런 과정을 통해서 딥러닝의 파라미터들이 error 를 통해서 파라미터를 업데이트합니다.

Chain Rule

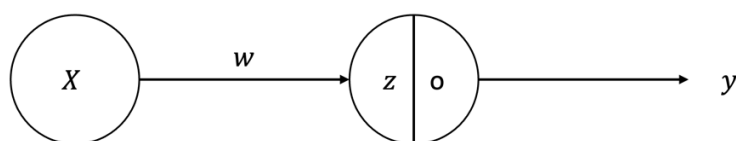


그림 7 Single Perceptron

Backpropagation 에서 가장 중요한 이론은 chain rule 을 사용한다는것 입니다. 여기서 chain rule 은 output 의 값에 따라 error 값을 구하고 출력층과 멀리 떨어져 있는 weight 의 gradient 값을 계산하기 위해서 반드시 필요합니다. 간단하게 위의 그림 7 을 예시로 설명을 해보겠습니다.

주어진 값

입력 : x , 가중치 : w , 가중합 : z , Activation 함수 이후의 값 : o , 출력 : y , 손실함수 : L

수식

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial o} * \frac{\partial o}{\partial z} * \frac{\partial z}{\partial w}$$

- $\frac{\partial L}{\partial o}$: 손실 함수에 대한 출력 o 의 미분. 이는 예측 값과 실제 값의 차이를 나타냅니다.
- $\frac{\partial o}{\partial z}$: 활성화 함수에 대한 가중합 z 의 미분. 활성화 함수의 도함수입니다.
- $\frac{\partial z}{\partial w}$: 가중합 z 에 대한 가중치 w 의 미분. 입력 값에 대한 도함수입니다.

역전파 진행

- 계산한 미분 값을 사용하여 출력층에서 입력층으로 거꾸로 이동하면서 각 가중치에 대한 미분값을 구합니다.
- 이러한 과정을 통해 네트워크의 모든 가중치에 대한 미분값이 구해집니다.

이렇게 weight 데 대하여 모든 layer 에 대하여 weight 에 대한 gradient 값을 전체적으로 구하는게 가능합니다.

Backpropagation

아래의 그림 8 에서처럼 MLP 의 대략적인 구조를 도식화를 하면 각각의 Layer 마다 그래프의 weight 의 연결 방향과 각각의 가중합, 노드 output 들이 모두 저장되어 있는 것을 확인할 수 있습니다. 이제 여기서 예측한 값 y 값을 기준으로 Backpropagation 을 진행이 되는 것을 Layer 별로 구분해서 설명을 해보겠습니다.

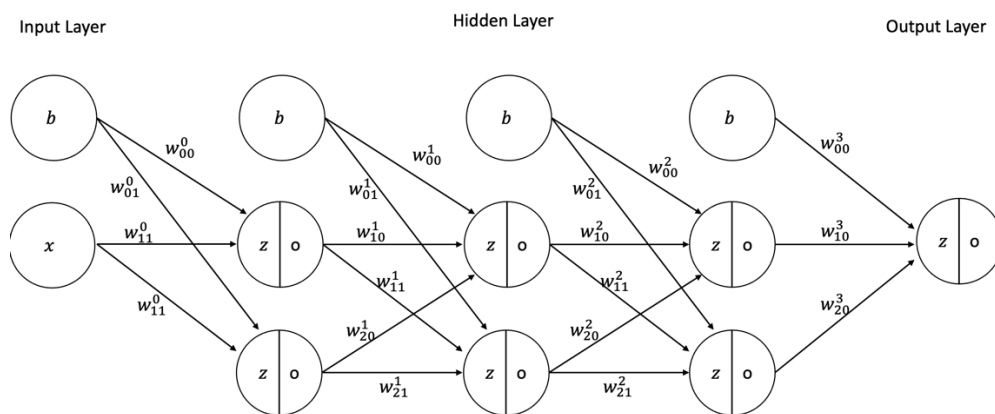


그림 8 MLP 구조 시각화

Output Layer

Output Layer

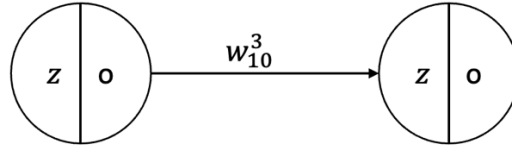


그림 9 단층 구조 Layer 시각화

$$\text{Cost Function} = L = \frac{1}{2} (o(z(o_{(L-1)} * w_{10}^3)) - y)^2$$

수식을 전개하면 cost function L 은 위와 같이 전개 하는게 가능합니다. 이제 이 식은 w_{10}^3 에 대하여 gradient 값을 구해야 하는데 이때 이전에 설명했던 chain rule 을 사용합니다.

$$\frac{\partial L}{\partial w_{10}^3} = \frac{\partial L}{\partial o} * \frac{\partial o}{\partial z} * \frac{\partial z}{\partial w_{10}^3}$$

$$\frac{\partial L}{\partial o} \rightarrow o(z(o_{(L-1)} * w_{10}^3)) - y \rightarrow \hat{y} - y$$

$$\frac{\partial o}{\partial z} \rightarrow o(z(o_{(L-1)} * w_{10}^3)) (1 - o(z(o_{(L-1)} * w_{10}^3))) \rightarrow \sigma'(z) \text{ (where } o = \text{sigmoid activation function)}$$

$$\frac{\partial z}{\partial w_{10}^3} \rightarrow o_{(L-1)}$$

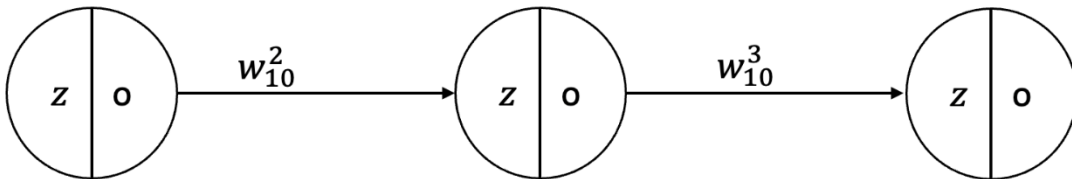
$$\frac{\partial L}{\partial w_{10}^3} = \frac{\partial L}{\partial o} * \frac{\partial o}{\partial z} * \frac{\partial z}{\partial w_{10}^3} = (\hat{y} - y) * \sigma'(z) * o_{(L-1)}$$

이렇게 weight 값을 단순하게 구하는게 가능해진다. 다음으로는 이제 조금 더 복잡한 Hidden Layer 의 gradient 를 구해보겠습니다.

Hidden Layer

Hidden Layer

Output Layer



$$\text{Cost Function} = L = \frac{1}{2} (o_L(z_L(o_{L-1}(z_{L-1}(o_{L-2} * w_{10}^2)) * w_{10}^3)) - y)^2$$

$$\frac{\partial L}{\partial w_{10}^2} = \frac{\partial L}{\partial o_L} * \frac{\partial o_L}{\partial z_L} * \frac{\partial z_L}{\partial o_{L-1}} * \frac{\partial o_{L-1}}{\partial z_{L-1}} * \frac{\partial z_{L-1}}{\partial w_{10}^2}$$

$$\frac{\partial L}{\partial o_L} \rightarrow o_L(z_L(o_{L-1}(z_{L-1}(o_{L-2} * w_{10}^2)) * w_{10}^3)) - y \rightarrow \hat{y} - y$$

$$\frac{\partial o_L}{\partial z_L} \rightarrow o_L(z_L(o_{L-1}(z_{L-1}(o_{L-2} * w_{10}^2)) * w_{10}^3)) (1 - o_L(z_L(o_{L-1}(z_{L-1}(o_{L-2} * w_{10}^2)) * w_{10}^3))) \rightarrow \sigma'(z) \text{ (where } o = \text{sigmoid activation function)}$$

$$\frac{\partial z_L}{\partial o_{L-1}} \rightarrow o_{L-1}(z_{L-1}(o_{L-2} * w_{10}^2)) * w_{10}^3 \rightarrow w_{10}^3$$

$$\frac{\partial o_{L-1}}{\partial z_{L-1}} \rightarrow o_{L-1}(z_{L-1}(o_{L-2} * w_{10}^2)) (1 - o_{L-1}(z_{L-1}(o_{L-2} * w_{10}^2))) \rightarrow \sigma'(z_{L-1})$$

$$\frac{\partial z_{L-1}}{\partial w_{10}^2} \rightarrow o_{L-2}$$

$$\frac{\partial L}{\partial w_{10}^2} = \frac{\partial L}{\partial o_L} * \frac{\partial o_L}{\partial z_L} * \frac{\partial z_L}{\partial o_{L-1}} * \frac{\partial o_{L-1}}{\partial z_{L-1}} * \frac{\partial z_{L-1}}{\partial w_{10}^2} = (\hat{y} - y) * \sigma'(z_L) * w_{10}^3 * \sigma'(z_{L-1}) * o_{(L-2)}$$

이렇게 각 weight 마다 gradient 값이 backpropagation 을 통해 구해진다면 이후에는 이 값을 통해서 weight 값을 업데이트를 해주어야 합니다. Weight 를 업데이트 해주는 방식은 이후에 나오는 Optimization 설명에서 진행을 하도록 하겠습니다.

1.6 Optimization

Stochastic Gradient Descent(SGD)

$$w := w - \alpha * gradient$$

SGD (Stochastic Gradient Descent)는 훈련 과정에서 각 반복(iteration)에서 개별 데이터 포인트에 대해 가중치를 업데이트합니다. 즉, 매 반복에서 하나의 데이터 포인트를 랜덤하게 선택하여 그 데이터 포인트에 대한 gradient 를 계산하고 가중치를 업데이트합니다. 이러한 방식으로 진행되기 때문에 "Stochastic"이라는 용어가 사용되었습니다.

여기서 핵심은 전체 데이터셋이 아니라 개별 데이터 포인트를 사용하여 gradient 및 업데이트를 계산한다는 것입니다. 이는 전체 데이터를 고려하는 것보다 계산 비용이 낮아지고, 특히 대규모 데이터셋에서 효과적일 수 있습니다. 그러나 이로 인해 업데이트가 불안정하고 가중치의 변동이 크다는 단점이 있습니다.

Mini-Batch

Mini-Batch 는 SGD(Stochastic Gradient Descent)에서 발생하는 문제를 해결하기 위한 방법으로, 훈련 데이터 세트를 일정한 크기로 나누어 모델을 업데이트합니다. 예를 들어, Mini-Batch 의 Batch_size 가 16 이라면 16 번의 반복을 수행한 후에 가중치를 업데이트합니다. 이를 통해 전체 데이터에 대한 가중치 업데이트를 진행함으로써 발생하는 문제를 완화할 수 있습니다.

Adaptive Moment Estimation(ADAM)

$$\begin{aligned} m_t &= \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t & (\text{모멘텀}) \\ v_t &= \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2 & (\text{RMSprop}) \\ \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} & (\text{모멘텀 보정}) \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} & (\text{RMSprop 보정}) \\ \theta_{t+1} &= \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \cdot \hat{m}_t & (\text{가중치 업데이트}) \end{aligned}$$

ADAM 방식은 최근에 가장 많이 사용되고 있는 방식으로 Momentum 와 RMSprop 방식을 결합한 방식으로 이루어 집니다.

- 모멘텀

Momentum 은 간단하게 설명하면 gradient 를 업데이트 할 때 이전 gradient 의 이동 방향과 업데이트 하려는 gradient 의 방향의 내적인 방향으로 gradient 를 업데이트 합니다. 이런 방식은 기존에 가지고있던 값의 방향성을 고려하기 때문에 지그제그로 gradient 값들이 변화되는 문제를 해결합니다.

- RMSprop

RMSprop 은 이동 평균을 사용하여 매개변수의 갱신 속도를 조절함으로써, 각 매개변수에 대한 학습률을 자동으로 조절하는 효과를 얻습니다. 이는 각 매개변수의 스케일에 따라 학습률을 다르게 설정하므로, 전역적인 학습률을 설정하는 것보다 효과적인 학습이 가능하게 합니다.

2. Code

- Main Idea & Environment

기본적인 전체 틀은 파이토치 프레임워크에서 모델을 선언하고 구성하는 방식을 참고하였습니다. 하지만 외부에서 제공하는 라이브러리를 사용하지 않고 직접 함수를 구성하고 진행하였습니다.

작업한 환경을 주피터 노트북 파일인 ipynb 파일을 통해서 진행하였습니다. 코드를 작업하기 위해서는 기본적인 가상환경 커널 환경을 설정을 해야 하며 데이터 파일은 코드 파일과 같은 폴더안에 존재하는 상태로 코드를 실행할 수 있습니다.

- Data preprocess

```
- df=pd.read_csv('hw2_data.csv')
-
- X=df[['x']]
- y=df[['y']]
-
- import numpy as np
-
- def train_test_split(X, y, train_ratio=0.8, random_seed=None):
-     X_values = X.values
-     y_values = y.values
-
-     if random_seed is not None:
-         np.random.seed(random_seed)
-
-     indices = np.random.permutation(len(X_values))
-     X_values = X_values[indices]
-     y_values = y_values[indices]
- 
```

```

- train_size = int(len(X_values) * train_ratio)
-
- train_x = np.array(X_values[:train_size])
- train_y = np.array(y_values[:train_size])
-
- test_x = np.array(X_values[train_size:])
- test_y = np.array(y_values[train_size:])
-
- return train_x, test_x, train_y, test_y
-
- train_x, test_x, train_y, test_y=train_test_split(X, y, train_ratio=0.8, random_seed=10)

```

데이터를 모델 학습을 진행하는 데이터와 성능을 측정하기 위한 데이터를 분리함으로써 모델이 학습을 진행하면서 성능이 높아지고 있는지를 확인하기 위해서 데이터를 분리합니다.

Model

Model define

```

in_dim = 1
hidden_dim = 150
out_dim = 1
layers=3
optimization='sgd'
activation='leakyrelu'

model = MyModel(in_dim, hidden_dim, out_dim, layers, activation, optimization)

```

모델을 선언하는 부분에서는 입력층의 차원, 은닉층의 차원, 출력층의 결과 차원을 초기값으로 설정하며, 네트워크를 구성하는 총 레이어 수를 지정하여 모델을 초기화합니다. 또한, 학습 중에 필요한 방식들을 설정하는데, 최적화 방법(optimization)과 활성화 함수(activation)를 지정하여 초기화합니다.

DenseLayer

weight initialize

```

class DenseLayer:
    def __init__(self, in_features, out_features, activation, bias=True ):
        self.in_features = in_features+1
        self.out_features = out_features
        self.activation_function=activation
        self.weight = np.random.randn(self.in_features, self.out_features) * np.sqrt(2 / (self.in_features +
self.out_features)) ①

```

①는 He 초기화의 주요 아이디어는 가중치를 무작위로 초기화할 때 적절한 분포를 사용하여 그레이디언트의 소실 또는 폭발 문제를 완화하는 것입니다. 특히 ReLU와 같이 양수 부분에서 비선형성을 갖는 활성화 함수를 사용할 때 중요합니다.

forward

```
def forward(self, x):
    if x.ndim == 1: # If a single sample
        x_with_bias = np.insert(x, 0, 1)
        self.input_data = x_with_bias
    else: # If a batch of samples
        x_with_bias = np.insert(x, 0, 1, axis=1)
        self.input_data = np.mean(x_with_bias, axis=0)

    result = np.dot(x_with_bias, self.weight). ①
    self.weight_sum = result

    if self.activation_function is None: ②
        result_after_activation = result
    else:
        result_after_activation = self.activation_function(result)

    self.node_output = result_after_activation

    return result_after_activation
```

모델 학습을 진행하는 과정에서 데이터가 모델에 들어오면 ①에서는 weight 값을 곱한 값을 layer에 들어온 값과 곱하여 weighted sum을 구합니다. ②에서는 weighted sum된 노드의 Input을 activation 함수를 통해 결과를 도출합니다. 여기서 activation이 없는 부분을 조건문에 걸어준 이유는 출력층에는 activation 함수가 없기 때문입니다.

calculate_gradient

```
def calculate_gradient(self, gradient):

    if gradient.ndim > 1: ①
        gradient=np.mean(gradient, axis=0)

    tmp = np.zeros(self.in_features)
    output = np.zeros(self.in_features - 1)
```

```

if self.activation_function is None: ②
    for i in range(self.out_features):
        tmp[self.in_features - 1] += gradient[i] * self.input_data[self.in_features - 1]
        output += gradient[i] * self.weight[self.in_features - 1, i]
    self.gradient = tmp

else:
    activation_derivative = self.activation_function.derivative(self.weight_sum)
    activation_derivative = np.mean(activation_derivative, axis=0)

    for i in range(self.out_features):
        tmp[self.in_features - 1] += gradient[i] * activation_derivative[i] *
self.input_data[self.in_features - 1]
        output += gradient[i] * activation_derivative[i] * self.weight[self.in_features - 1, i]

    self.gradient = tmp

return output

```

이 함수에서는 먼저 ① 에서 gradient 의 차원을 기준으로 조건문을 수행합니다. 여기서 조건문을 수행하는 이유는 batch_size 에 따라서 각 batch 에서 생성되는 gradient 의 값을 평균을 내기 위해서 입니다. ② 에서는 이전 방식과 같이 출력층과 출력층이 아닌 부분을 구분하기 위해서 조건문을 추가를 하였고, 1.5 Backpropagation 에서 설명을 했듯이 각각 layer 와 출력층에서 수행하는 방식을 다르게 표현을 하였습니다.

update_weights

```

def update_weights(self, learning_rate, optimization):

    if optimization == 'sgd':
        if len(self.gradient.shape) == 1:
            self.gradient = self.gradient.reshape((-1, 1))

        self.weight -= learning_rate * self.gradient
        return

    elif optimization == 'adam':
        if len(self.gradient.shape) == 1:
            self.gradient = self.gradient.reshape((-1, 1))

        self.t += 1

        self.m = self.beta1 * self.m + (1 - self.beta1) * self.gradient
        self.v = self.beta2 * self.v + (1 - self.beta2) * (self.gradient ** 2)

```

```

        m_hat = self.m / (1 - self.beta1 ** self.t)
        v_hat = self.v / (1 - self.beta2 ** self.t)
        self.weight -= learning_rate * m_hat / (np.sqrt(v_hat) + self.epsilon)

        self.m = self.m.reshape(self.weight.shape)
        self.v = self.v.reshape(self.weight.shape)

    return

def __call__(self, x):
    return self.forward(x)

```

Update_weights 의 함수에서는 1.6 Optimization 이론에서 이야기를 했듯이 SGD 방식과 ADAM 을 구현하였습니다.

Train

```

batch_size=4
early_stop_loss=0
for epoch in range(num_epochs):

    loss_list=[]
    # Mini-batch training
    for i in range(0, num_samples, batch_size):
        x_batch = np.array(train_x[i:i + batch_size])
        y_batch = np.array(train_y[i:i + batch_size])
        # Forward pass
        y_pred_batch = model.forward(x_batch)

        loss = loss_function(y_pred_batch, y_batch)
        model.backward(loss_function.derivative(y_pred_batch, y_batch))
        model.update_weights(lr)

        loss_list.append(loss)
        total_train_loss.append(loss)

    # Validation
    total_val_loss = 0

    for i in range(test_x.shape[0]):
        x_val = test_x[i]
        y_val = test_y[i]
        y_pred_val = model.forward(x_val)
        val_loss = loss_function(y_pred_val, y_val)
        total_val_loss += val_loss

```

```

avg_val_loss = total_val_loss / len(test_x)

if abs(np.mean(loss_list) - early_stop_loss) <= 0.00001: ①
    print("early stop!!")
    break
early_stop_loss=np.mean(loss_list)

print(f"Epoch [{epoch + 1}/{num_epochs}], Training Loss: {np.mean(loss_list)},Validation Loss: {avg_val_loss[0]}")

avg_train_loss = np.mean(total_train_loss)
print(f"Final Training Loss: {avg_train_loss}")
print(f"Final Validation Loss: {avg_val_loss}")

```

훈련을 담당하는 코드에서 가장 크게 신경쓴 부분은 batch_size 를 지정하면 Mini-batch 방식으로 훈련이 진행되는 부분과 이전에 코드에서 분리했던 test 데이터 세트를 모델의 성능을 평가하는 데이터로 사용하여 모델을 학습하고 이후에 모델 성능을 epoch 마다 판단하였습니다. 또한 overfitting 을 방지하기 위해서 이전의 학습한 loss 값과 새롭게 얻은 loss 값을 비교하여서 학습을 빠르게 종료하는 early stop 을 ①에서 구현을 하였습니다.

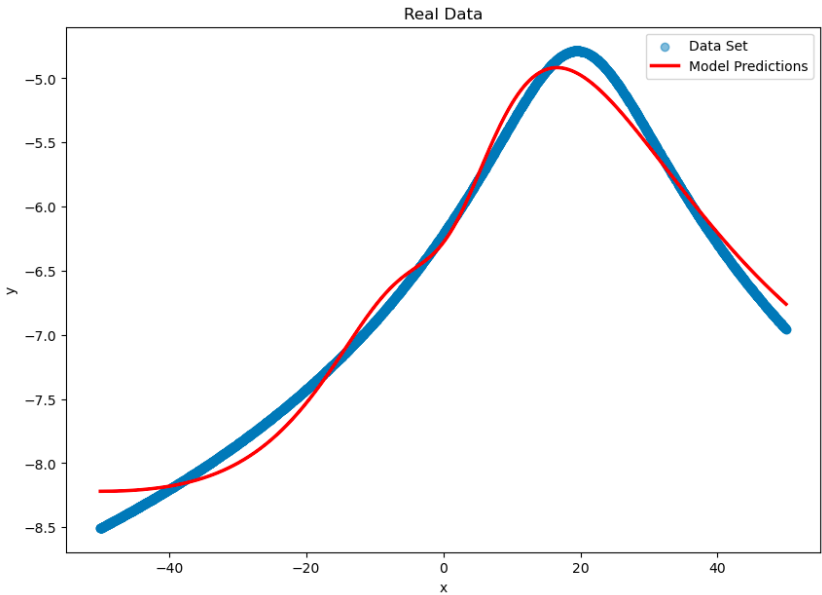
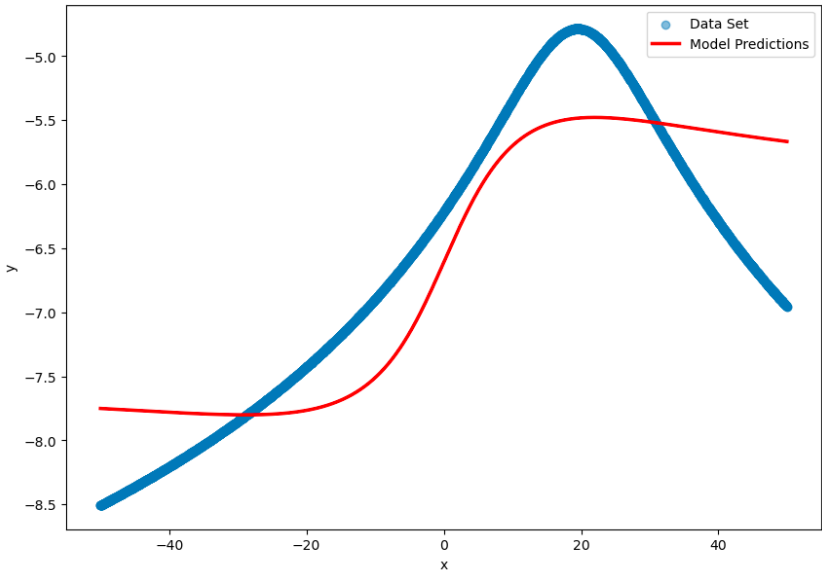
3. Result

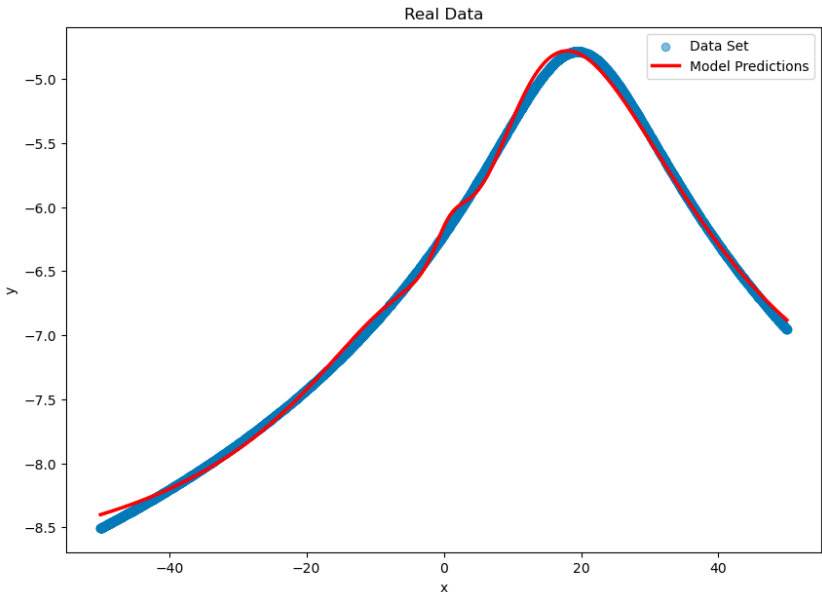
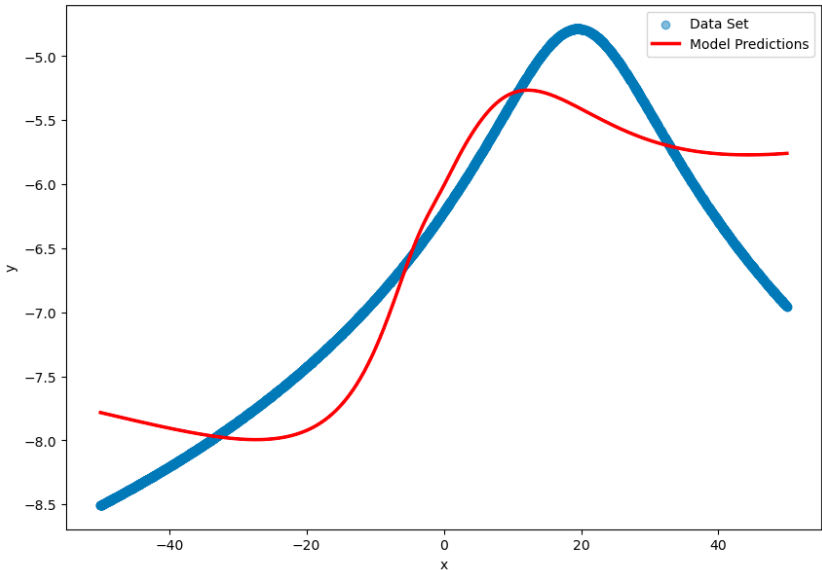
실험은 각각 조건에 따라 동일하게 진행을 하기 위해서 모델을 구성하는 in_dim, hidden_dim, out_dim, layers, 에 대한 조건과 epochs, lr 조건을 모두 같게 두고 진행을 하였습니다. 여러 번의 실험을 통해서 모델의 layer 가 복잡해지면 underfitting 문제가 많이 발생하여서 3 layer 만을 사용하였습니다.

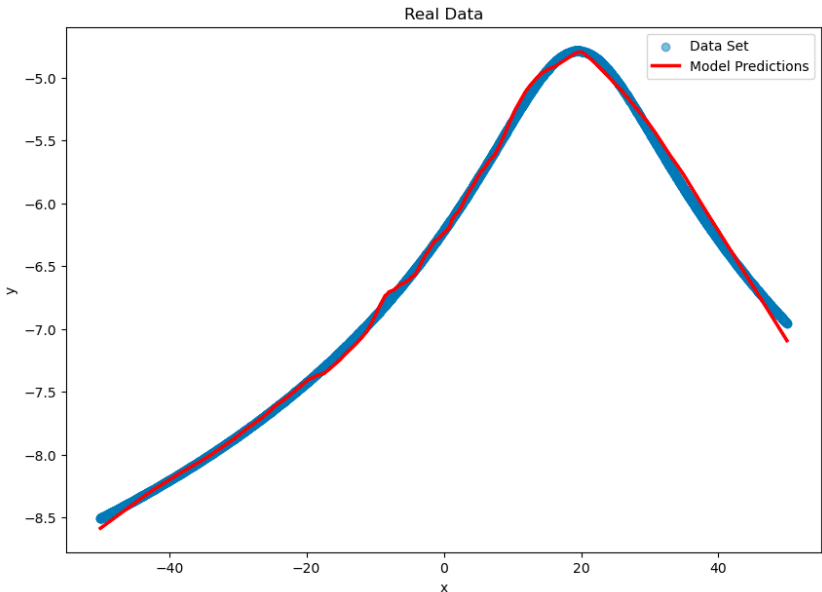
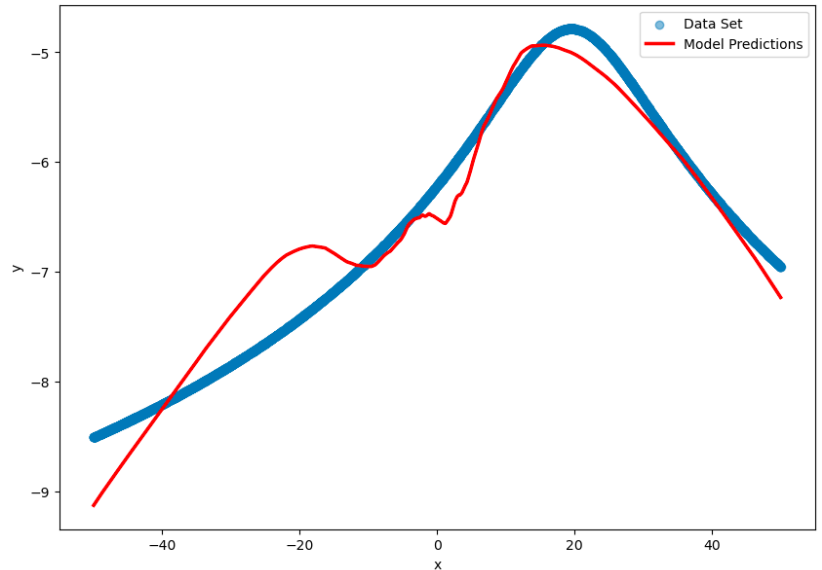
- In_dim : 1
- Hidden_dim : 150
- Out_dim : 1
- Layers : 3
- Epochs : 100
- Lr : 0.0001

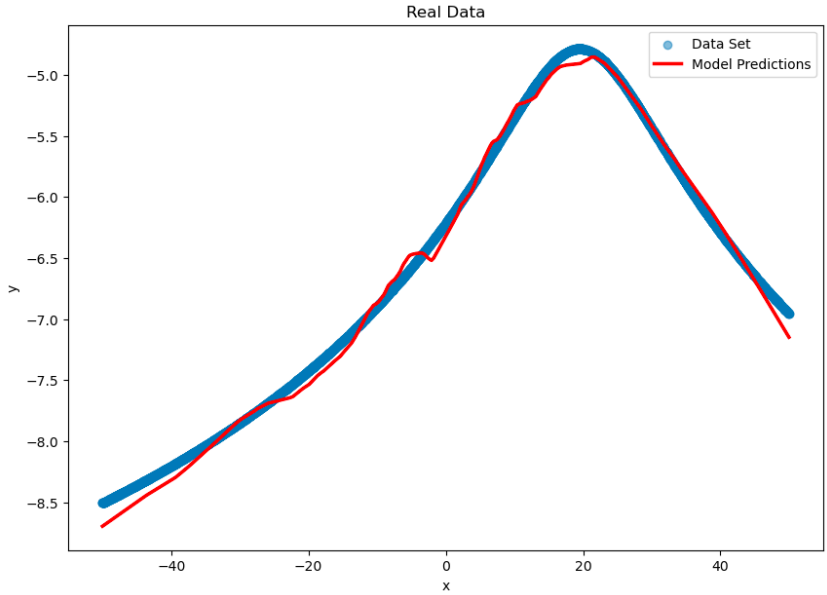
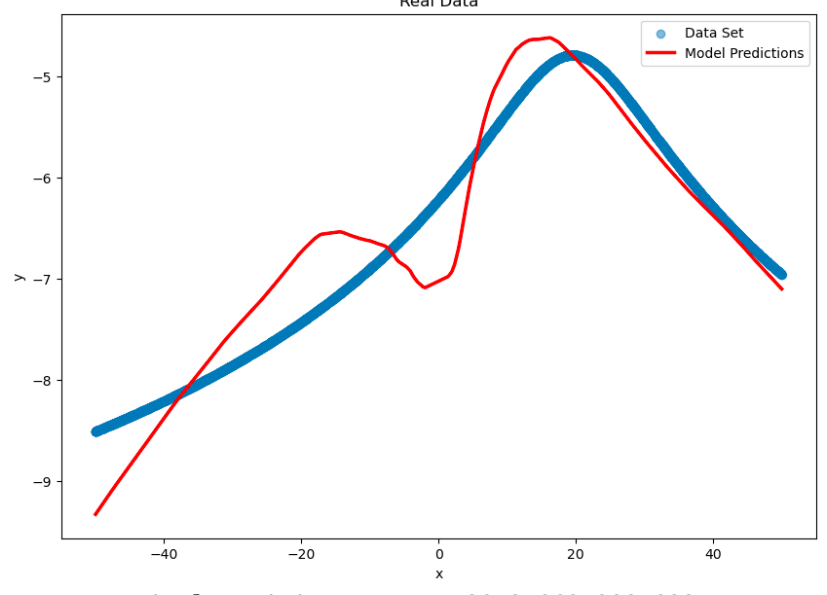
SGD

Activation	Batch	Result
------------	-------	--------

sigmoid	1 epoch (100/100)	 <p>Final Training Loss: 0.06898230667482408 Final Validation Loss: 0.00460076</p>
	8 epoch (22/100)	 <p>Final Training Loss: 0.40747993311133746 Final Validation Loss: 0.23277374883112395</p>

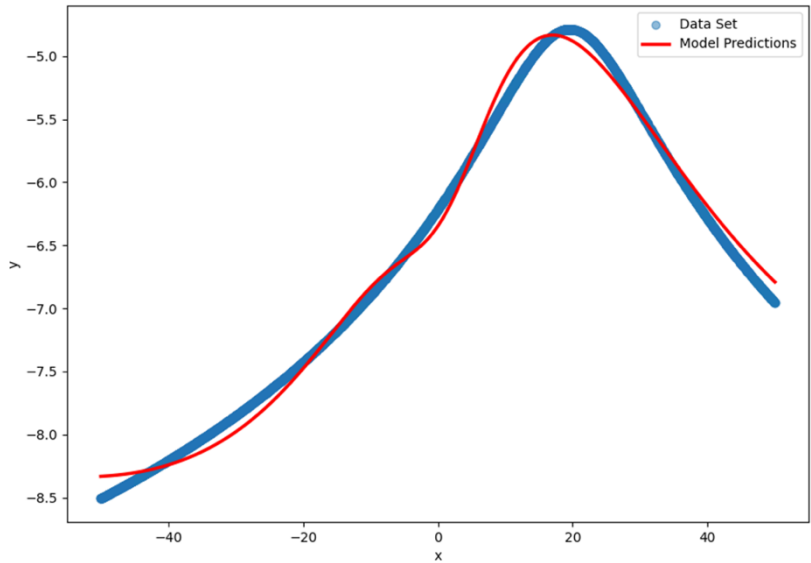
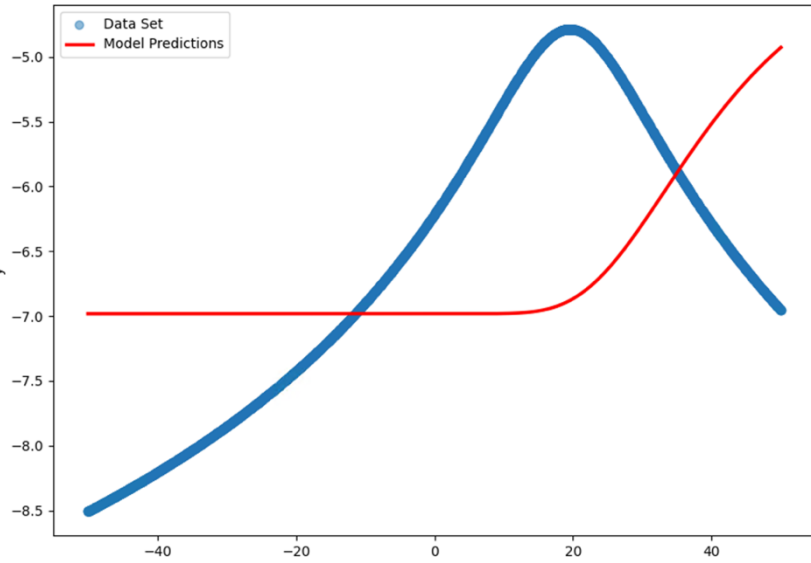
tanh	1 epoch (100/100)	 <p>Final Training Loss: 0.03483162672906155 Final Validation Loss: 0.0019200851974026161</p>
	8 epoch (100/100)	 <p>Final Training Loss: 0.40330699609552406 Final Validation Loss: 0.10265002969882384</p>

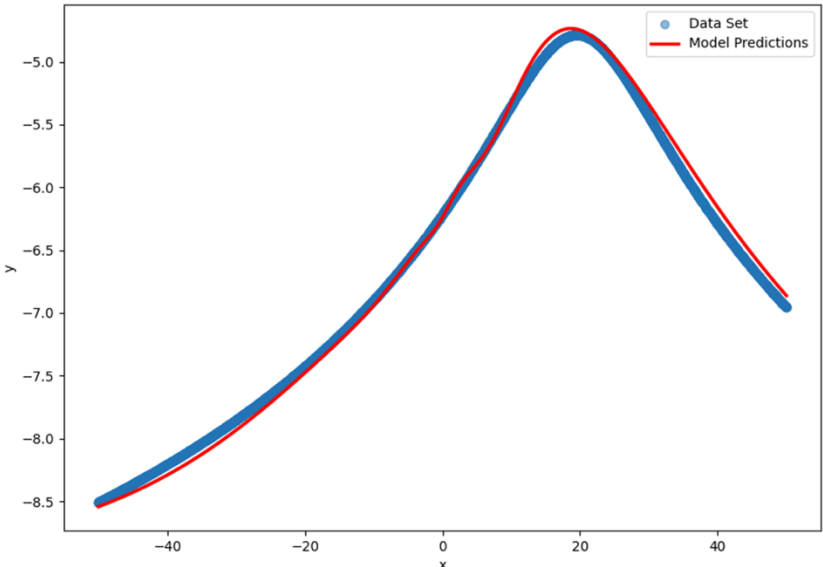
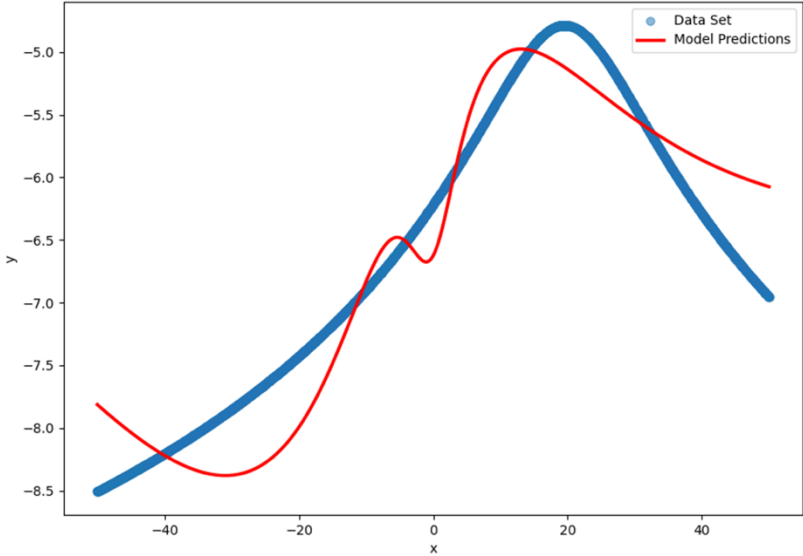
relu	1 epoch (100/100)	 <p>Final Training Loss: 0.023085266624503876 Final Validation Loss: 0.0010136134984646944</p>
	8 epoch (100/100)	 <p>Final Training Loss: 0.6735503612983282 Final Validation Loss: 0.04528592210671582</p>

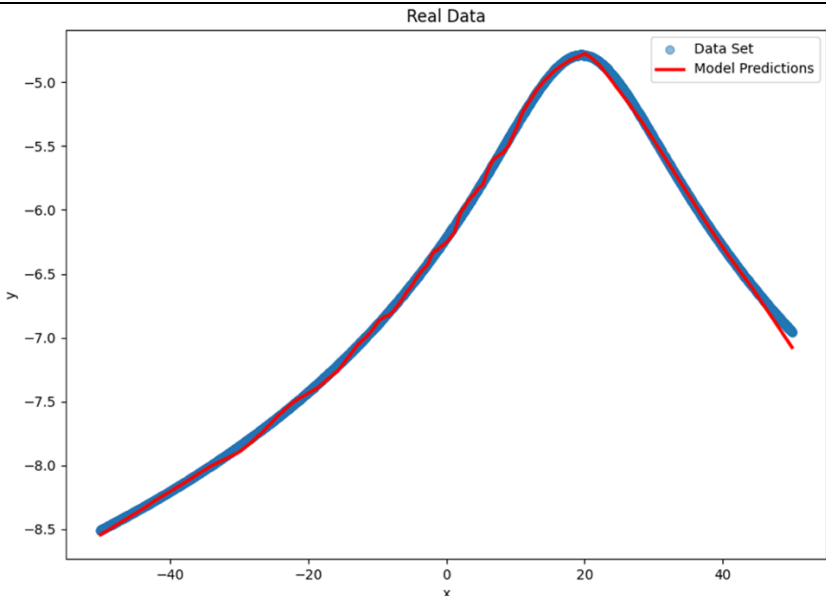
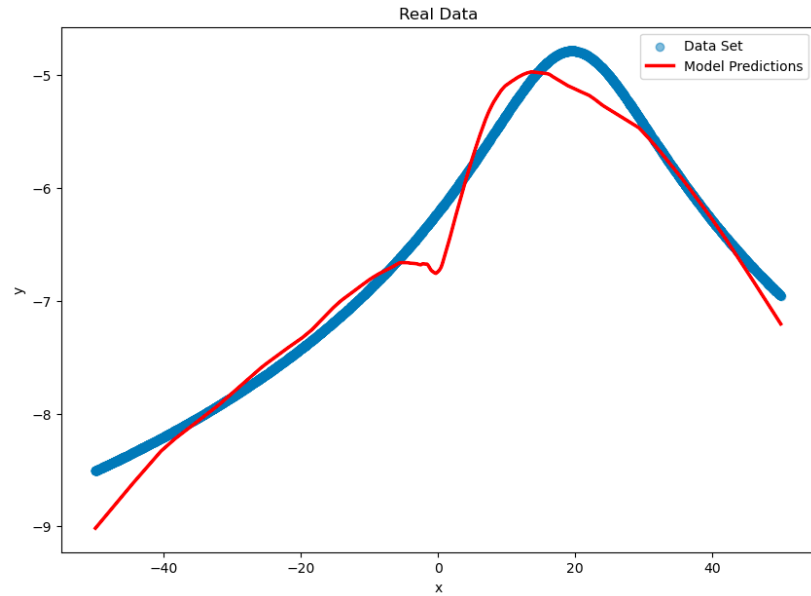
leakyrelu	1 epoch (19/100)	 <p>Final Training Loss: 0.11190984935116764 Final Validation Loss: 0.0034959228947957334</p>
	8 epoch (100/100)	 <p>Final Training Loss: 1.2853186252230833 Final Validation Loss: 0.0431024834902267</p>

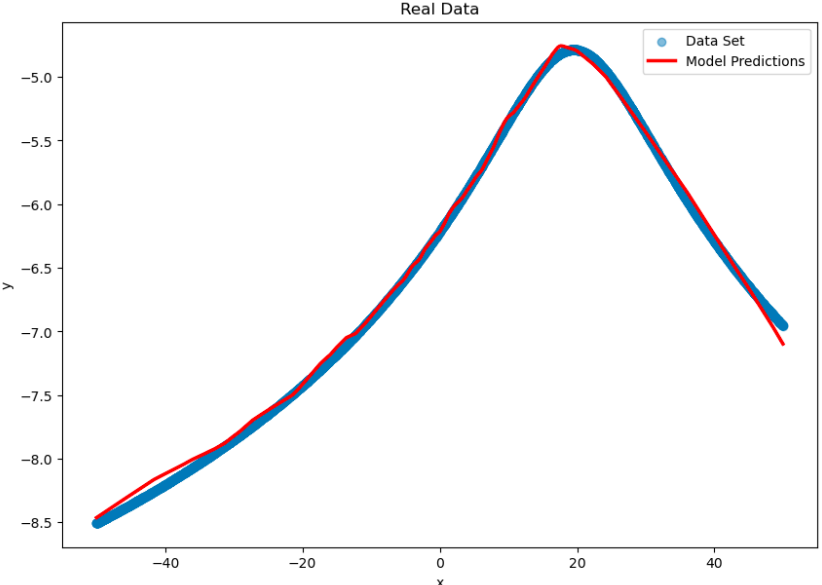
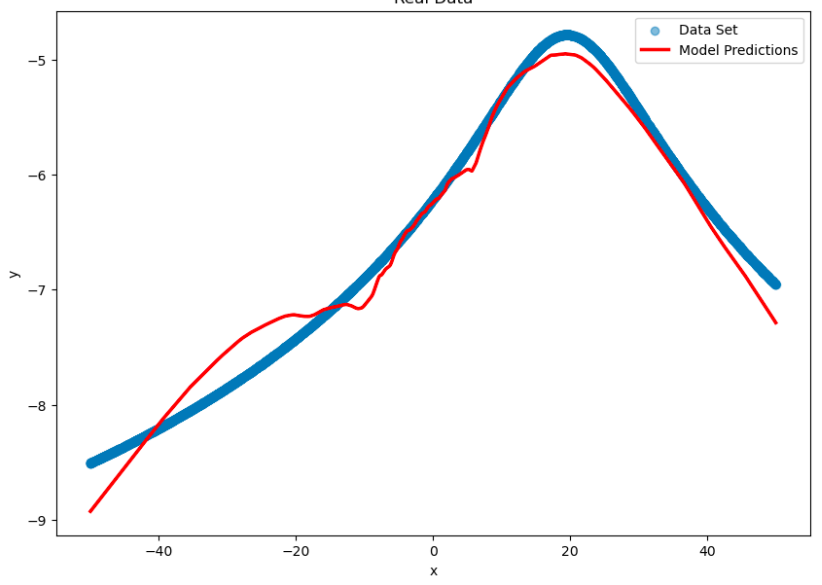
SGD 에서의 실험 결과는 각각의 activation function 의 종류별로는 성능이 많이 차이가 나지는 않았습니다. 하지만 학습하는 시간을 계산해 보면 relu, leakyrelu 의 학습시간이 다른 것에 비하여 빠르게 진행이 되었습니다. Batch 의 관점으로 본다면 batch size 를 1로 두고 학습을 진행한 것 이 8로 한 거보다 더 좋은 성능을 보였습니다. Batch 를 나눠서 학습을 진행한다면 모델의 overfitting 문제를 해결하는 데 도움을 줄 수 있습니다. 하지만 이번 실험에서는 충분한 양의 데이터가 존재하지 않아서 Batch 를 나눠서 학습을 진행을 한 거보다 SGD 방식이 더 성능이 좋은 모습을 보였습니다.

Adam

Activation	Batch	Result
sigmoid	1 epoch (30/100)	<p>Real Data</p>  <p>Final Training Loss: 0.08337918584826294 Final Validation Loss: 0.004499946834310984</p>
	8 epoch (100/100)	<p>Real Data</p>  <p>Final Training Loss: 0.495231064679146 Final Validation Loss: 0.6505775102870373</p>

tanh	1 epoch (87/100)	<p>Real Data</p>  <p>Final Training Loss: 0.10539833901173236 Final Validation Loss: 0.002957721559675478</p>
	8 epoch (100/100)	<p>Real Data</p>  <p>Final Training Loss: 0.6027795360881153 Final Validation Loss: 0.031911459126184955</p>

relu	1 epoch (82/100)	 <p>Final Training Loss: 0.05220791761070657 Final Validation Loss: 0.000709317745901239</p>
	8 epoch (100/100)	 <p>Final Training Loss: 0.809613691194922 Final Validation Loss: 0.019461317006439355</p>

leakyrelu	1 epoch (29/100)	 <p>Final Training Loss: 0.1776761955503764 Final Validation Loss: 0.0010569483111174882</p>
	8 epoch (100/100)	 <p>Final Training Loss: 0.792039308356282 Final Validation Loss: 0.016649402546852682</p>

SGD 와는 다르게 다른 Optimization 을 ADAM 으로 적용해서 실험을 진행하였습니다. 여기서의 각각 다른 함수들에서 성능이 잘 나왔지만 가장 주목할 만한 점은 학습 속도입니다. Relu 를 기준으로 adam 을 적용한 이후에 더욱 빠르게 최적의 파라미터를 찾았스비다. 그리고 tanh 함수 같은 경우에는 같은 epoch 로 학습을 진행하였지만 결과적으로 loss 값이 줄어드는 모습을 보였으며 모델의 정확성을 높였습니다.

Best Model

결론적으로 훈련의 성능으로 보나 결과로 보나 relu 함수에서 batch size 를 1 로 SGD 방식으로 실험한 결과가 가장 좋았습니다.

Weights

Output layer : (2 150)

Hidden layer : (151 150)

Input layer : (151 1)

Total number of parameters in the model: 23402

```
0.07239135 -0.12030545 0.1370441 0.06664621 -0.04635901 -0.03539416
-0.01614381 0.04472527 0.05537885 0.07747641 -0.05701403 0.06358739
-0.02441473 -0.08212166 0.07581772 -0.23045361 -0.09700245 -0.0812667
0.06787895 0.07763091 -0.08540933 -0.02665614 0.12537278 -0.05152724
0.00117784 -0.25034053 0.1364745 -0.07333614 0.08609364 -0.04691761
0.04103995 0.08727845 -0.04135786 -0.03055816 -0.1409265 -0.01986565
0.06850167 0.02109147 -0.26753896 0.06288356 -0.00674405 -0.25743007
-0.10063024 0.04760479 -0.04254363 -0.32546525 0.10153038 0.16412092
-0.21856255 0.15926691 0.05482637 0.00627518 0.28818303 -0.12451147
0.01074939 -0.02819475 -0.13261133 -0.16847236 -0.09091906 -0.19278327
-0.01290196 -0.15938265 0.11926076 0.06042918 0.20144381 0.1106425
0.40299241 0.03737493 0.0947226 -0.01372075 -0.16744299 -0.10369826
0.09176079 -0.04413693 0.10835018 -0.14643893 0.03119788 0.08641163
0.03635451 0.13442924 0.2594686 0.04922741 0.11362267 -0.17118119]]
layer 1
weights [[ 0.78477382 0.80393407 0.68034974 ... 0.60199317 0.61287841
0.75355394]
[ 0.04703905 -0.03683477 0.02131628 ... -0.06223254 -0.04606612
...
[-2.75741725e-01]
[-1.79199434e-02]
[-2.50346291e-01]
[-2.51484748e-01]]
```

4. Conclusion

시행착오

1. Model architecture

모델 구조는 현재 실험을 할 때 layer 의 수를 3 개로 하였습니다. 하지만 MLP, DNN 구조는 1 개 이상의 hidden layer 를 구성해야 합니다. 그런데 이번 과제에서 하지 못했던 이유는 데이터의 구조가 단순하고 데이터의 양이 많지 않아서 모델의 구조를 깊게 만들면 과적 합의 문제가 쉽게 발생하여서 모델의 성능이 잘 나오지 않았습니다. 그래서 이번 과제에서는 단순한 구조를 사용하여 실험을 진행하였습니다. 이후에는 추가로 데이터가 더 많이 주어진다면 더 깊은 구조의 모델도 실험을 추가로 진행을 해야 합니다.

2. Activation function

Activation function 은 현재 sigmoid, tanh, relu, leakyrelu 를 구현하였습니다. 하지만 이런 activation 함수들은 각각 함수들에 대한 특징들이 존재합니다. 그래서 어떤 데이터를 사용하냐, 어떤 구조를 가지냐에 따라서 다양하게 사용이 가능하였고 성능도 달라짐을 확인했습니다.

3. Optimization

Optimization 의 방식을 여러 개 존재하는 이유는 기존의 방식의 단점들을 보완하기 위해서 이론이 등장하였습니다. 하지만 실험에서는 모델의 구조가 단순한 데이터의 양이 많지 않아서 기존의 SGD 방식으로도 충분히 학습이 잘 이루어졌으며 오히려 ADAM 을 적용하면 학습이 잘 이루어지지 않는 문제가 있었습니다.

4. Data

딥러닝의 기반한 모델을 구성하기 위해서는 여러 layer, weight 에 대하여 모델의 예측값이 반환됩니다. 따라서 이런 딥러닝의 특성상 많은 weight 들이 존재하기 때문에 이런 모델을 학습하기 위해서는 충분히 많은 양의 데이터가 필요합니다. 하지만 이번 과제에서는 충분한 양의 데이터가 존재하지는 않아서 이론과는 조금 다른 결과가 나오기도 했습니다.

느낀점

이번 과제를 통해서 딥러닝 구조들의 가장 기본적인 구조인 MLP 에 대한 프로세스를 직접 구현을 해 보았습니다. 이번 과제를 진행하면서 MLP 에 대한 구조를 조금 더 명확하게 이해를 하였습니다. 하지만 이번 과제를 통해서 이전에는 알지 못했던 부분들도 정확히 알 수 있는 기회가 되어서 좋았습니다. 딥러닝 구조를 쌓으면서 구조적으로 적용되는 activation 들의 활용 이유와 그리고 구조들에 따른 성능 변화들을 더욱 확고하게 알 수 있었습니다. 시간 관계상 더 많은 추가적인 구현을 해보지는 못했지만 그래도 노력을 한 만큼 얻는 것도 많은 과제였던 것 같습니다.