객체지향 프로그래밍 [CSED232-01]

Assignment #1

Due date: ~ 10.09 (Wed.) 23:59

Objective

본 과제에서는 Multi-Layer Perceptron을 구현하여 Two Moon Dataset 과 Iris Dataset 들을 학습하고 결과를 산출한다. 이를 통해 C++의 다양한 Class 기법을 연습한다.

Background – Perceptron, Gradient Descent, Multi-Layer Perceptron Error, backpropagation

Neural Network의 가장 기본 단위를 Perceptron이라고 한다. 이를 살펴보기 이전에 인간의 뉴런과 어떠한 유사점을 가지고 있는지 살펴보자. 그림 1의 뉴런을 살펴보면, 수상돌기를 통해 들어온 다수의 신호를 입력으로 받아 이를 변조하고, 일정 threshold를 넘기는 경우 신호를 출력하는 구조를 가지고 있다.

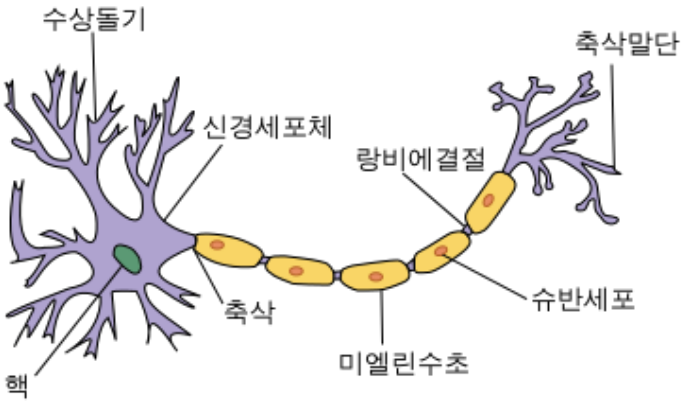


그림 1. 뉴런의 구조

이와 유사한 구조를 가진 인공적인 뉴런을 Perceptron이라고 하며, 실제 뉴런과 유사한 동작 메커니즘을 가지고 있다.

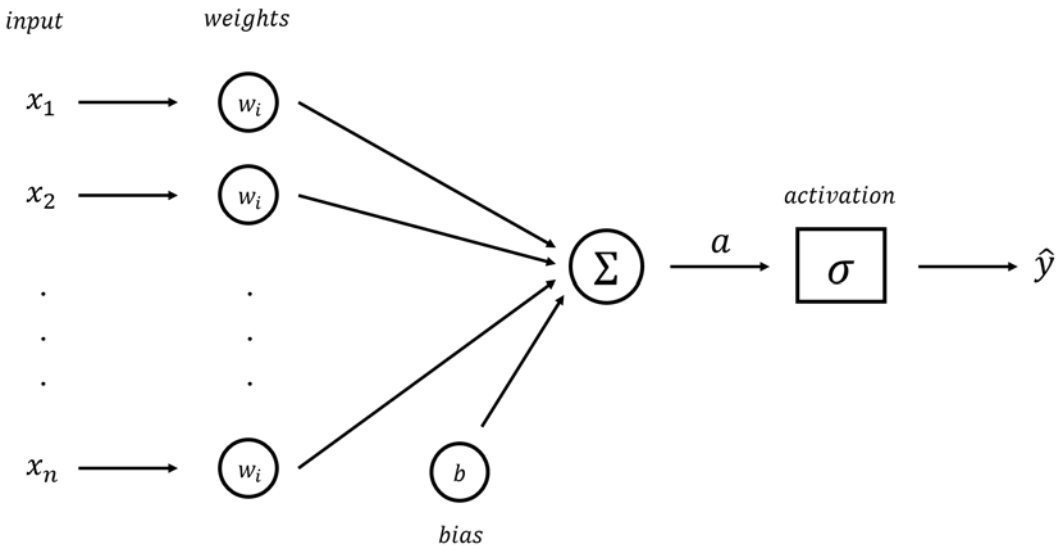


그림 2. Perceptron의 구조

각각의 입력 값은 weighted summation을 통해 하나의 값으로 합쳐져 activation function을 통해 최종 출력이 결정된다. 수학적인 정의는 다음과 같다. (bold는 Vector)

- 입력: $\boldsymbol{X} = (x_1, x_2, \dots, x_n)$
  - 입력 데이터의 차원은 곧 표현 가능한 데이터의 차원으로, feature라고 한다.
- 출력: $\hat{y}$
- 모델 파라미터
  - 가중치: $\boldsymbol{W} = (w_1, w_2, \dots, w_n)^T$
  - 바이어스: b

$$Perceptron\text{:}\ \hat{y} = \sigma\left(\sum_{\boldsymbol{i}} w_i x_i + b\right) = \sigma(\boldsymbol{XW} + b)$$

여기서 σ는 sigmoid function으로 다음과 같이 정의된다.

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

그림 2와 같이 sigmoid function의 값은 0에서 0.5, 그리고 음수와 양수일 때 각각 0.5보다 작고 0.5보다 큰 값을 가지므로 0.5를 기준으로 True와 False로 나누는 activation에 적합한 함수이다.

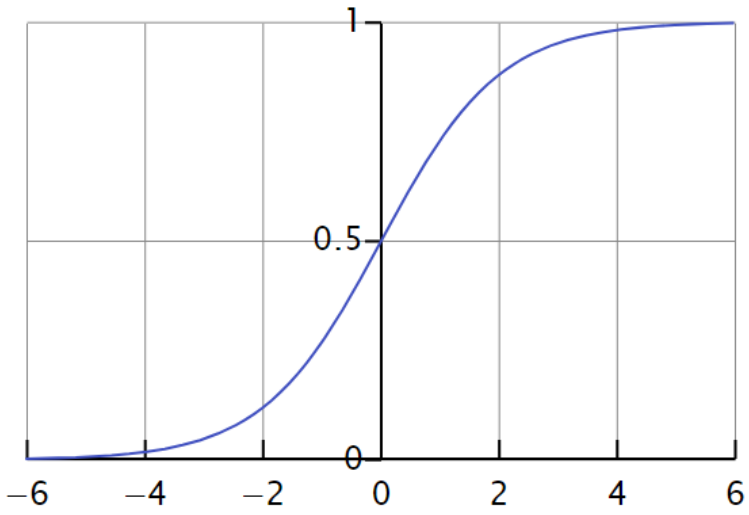


그림 3. Sigmoid function

Perceptron을 학습하는 방법은 Gradient Descent 방법이 주로 사용된다. Gradient Descent를 설명하기 전 목적 함수의 개념을 알아야 한다. Loss Function이라고도 하는 이 개념은 Perceptron의 결과값과 우리가 원하는 정답과의 차이를 설명해주는 함수로, 정답과의 거리가 멀면 큰 값을, 가까우면 작은 값을 출력하여주는 함수이다. 가장 간단한 함수로는 Mean Square Error 가 있다.

$$loss\ function: L(y, \hat{y}) = (\hat{y} - y)^2/2$$

여기서 y는 데이터의 실제 레이블(정답)이다. 위 값이 목적 함수의 값이 0에 가까워질수록 정답에 가까워진다. 손실 함수의 값을 최소로 하는 최적의 해를 구하기 위해서는 손실 함수의 Gradient를 살펴볼 필요가 있다.

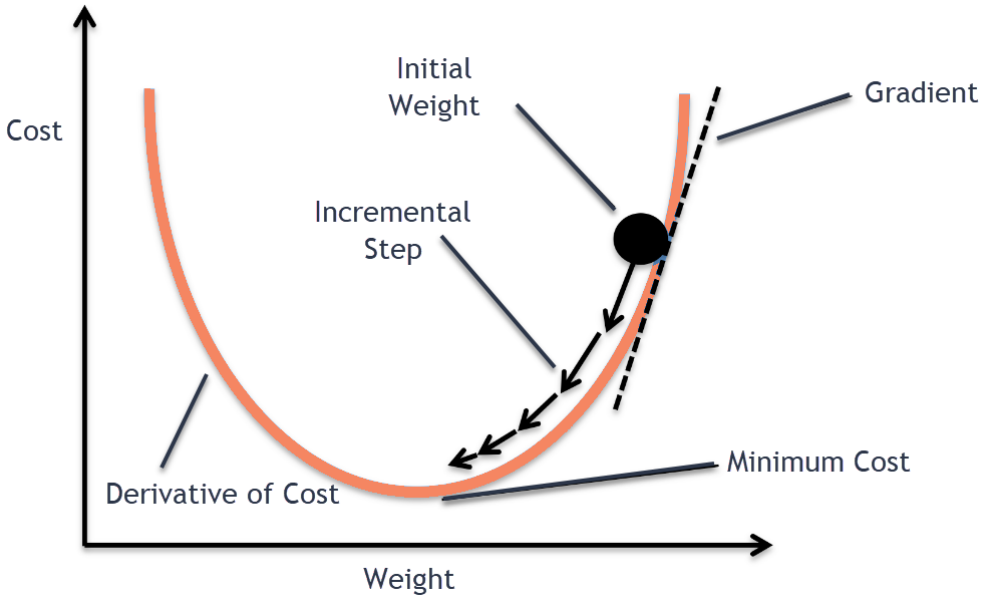


그림 4. Gradient Descent Algorithm

그림 4과 같이, Gradient는 특정 지점에서의 tangential slope를 나타낸다. 만약 Gradient의 negative 방향으로 모델 파라미터를 업데이트 한다면, 손실 함수 값이 줄어드는 방향으로 최적화 될 것이며, 이를 Gradient Descent 방법이라고 하고, 이 과정을 우리는 모델을 "학습"한다고 한다.

위에서 언급한 Perceptron을 Gradient Descent 방법으로 학습하기 위해서는 각 파라미터에 대해 Gradient를 구해야 한다. 여기서 보통 우리는 Chain rule을 이용하여 각 파라미터에 대한 손실함수의 미분 값을 구한다. $a = \sum_i w_i x_i + b$ 라고 정의할 때, 각각의 Gradient를 구하는 방법은 다음과 같다.

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y$$

$$\frac{\partial \hat{y}}{\partial a} = \sigma(a)\big(1 - \sigma(a)\big)$$

$$\frac{\partial a}{\partial w_i} = x_i, \qquad \frac{\partial a}{\partial b} = 1$$

$$\therefore \frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a} \times \frac{\partial a}{\partial w_i} = (\hat{y} - y)\sigma(a)\big(1 - \sigma(a)\big)x_i$$

$$\therefore \frac{\partial L}{\partial b} = \frac{\partial L}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial a} \times \frac{\partial a}{\partial b} = (\hat{y} - y)\sigma(a)\big(1 - \sigma(a)\big)$$

마지막으로 위에서 구한 Gradient를 통해 파라미터들을 업데이트 해주어야 하는데, 이를 그대로 사용할 경우 수렴하지 않고, 진동할 수 있다. 이는 Gradient 값이 어느 정도로 클지 모르기 때문이며, 이 값을 조절하여 적당하게 학습될 수 있도록 learning rate η 라는 hyper-parameter를 설정해주어야 한다. 보통 이 값은 작은 값으로 설정하는데, 너무 작으면 수렴하는데 시간이 오래 걸리며, 너무 크면 오히려 발산하기 때문에 주의해서 설정해주어야 한다.

$$w_i^{new} = w_i^{old} - \eta \frac{\partial L}{\partial w_i^{old}}$$

$$b^{new} = b^{old} - \eta \frac{\partial L}{\partial b^{old}}$$

사실 파라미터에 대한 손실함수의 gradient는 손쉽게 바로 구할 수 있다. 하지만, 굳이 chain rule을 이용하여 구한 이유는 뒤에 나올 error backpropagation 방법에서 사용되는 방법이기 때문에 이해를 돕기 위한 것이다.

더 나아가 위 계산을 편의상 Vector 연산으로 축약할 수 있다. 이 때는 더 이상 편미분 기호가 아닌 Gradient 기호인 nabla를 사용한다.

$$\nabla_{\hat{y}} L = \hat{y} - y$$

$$\nabla_a \hat{y} = \sigma(a)\big(1 - \sigma(a)\big)$$

$$\nabla_{\boldsymbol{W}} a = \boldsymbol{X}^T, \qquad \nabla_b a = 1$$

$$\therefore \nabla_{\boldsymbol{W}} L = \nabla_{\hat{y}} L \times \nabla_a \hat{y} \times \nabla_{\boldsymbol{W}} a = (\hat{y} - y)\sigma(a)\big(1 - \sigma(a)\big)\boldsymbol{X}^T$$

$$\therefore \nabla_{\boldsymbol{b}} L = \nabla_{\hat{y}} L \times \nabla_a \hat{y} \times \nabla_{\boldsymbol{b}} a = (\hat{y} - y)\sigma(a)\big(1 - \sigma(a)\big)$$

이를 사용한 Gradient descent는 다음과 같다

$$\boldsymbol{W}^{new} = \boldsymbol{W}^{old} - \eta \nabla_{\boldsymbol{W}^{old}} L$$

$$b^{new} = b^{old} - \eta \nabla_{b^{old}} L$$

Perceptron의 개념을 확장하면 Multi-Layer Perceptron의 개념까지 확장할 수 있다.

Multi-Layer Perceptron 은 위에서 본 Perceptron들을 stacking 한 구조이다. 따라서 출력 값이 하나의 scalar 값이 아닌 vector 형태인 것이다. Perceptron은 구조적으로 Linear function이다. 따라서 Nonlinear한 정보를 classification하는데 Multi-Layer Perceptron이 유리하다.

그림 5에서 각각의 원은 Perceptron을 의미하고 각 원안의 $\boldsymbol{W}$ 와 b 는 파라미터를 의미한다. Hidden layer는 계층적으로 봤을 때 같은 계층에 있는 Perceptron들의 집합이다.

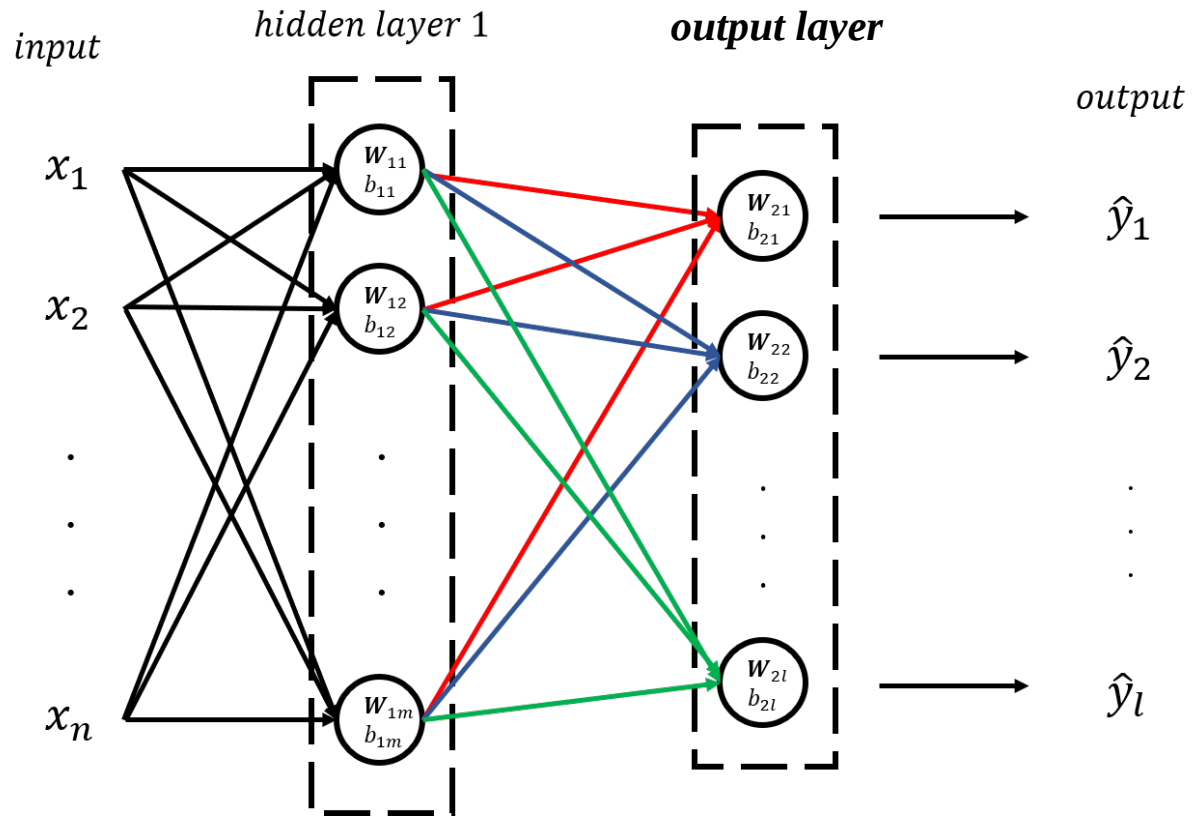


그림 5. Multi-Layer Perceptron

먼저 hidden layer 1을 수식적으로 표현하면 다음과 같다. 여기서 m은 hidden layer 1의

perceptron 개수이다.

$$h_{1i} = \sigma(\boldsymbol{XW_{1i}} + b_{1i}) \, , i = \{1, \dots m\}$$

$$\boldsymbol{h_1} = (h_{11}, h_{12}, \dots, h_{1m})$$

그림 5와 같이 Hidden layer 1의 출력은 곧 output layer의 입력으로 사용된다. 이를 활용하여 Output layer의 수식도 표현하면 다음과 같다. 여기서 m은 output layer의 perceptron 개수이다.

$$h_{2j} = \sigma\left(\boldsymbol{h_1 W_{2j}} + b_{2j}\right) , j = \{1, \dots l\}$$

처음 Perceptron을 다룰 때처럼 위 수식들을 하나의 Matrix multiplication으로 표현하면 더 간결한 표현이 되며, 실제로 프로그램 구현 시 이 방법을 사용한다.

$$\boldsymbol{W_1} = \begin{bmatrix} w_{111} & \cdots & w_{1n1} \\ \vdots & \ddots & \vdots \\ w_{11m} & \cdots & w_{1nm} \end{bmatrix} = [\boldsymbol{W_{11}} \quad \cdots \quad \boldsymbol{W_{1m}}]$$

$$\boldsymbol{b_1} = [b_{11} \quad \cdots \quad b_{1m}]$$

$$\boldsymbol{h_1} = [\sigma(\boldsymbol{XW_{11}} + \boldsymbol{b_{11}}) \quad \cdots \quad \sigma(\boldsymbol{XW_{1m}} + \boldsymbol{b_{1m}})] = \sigma(\boldsymbol{XW_1} + \boldsymbol{b_1})$$

Output layer도 같은 방법으로 구하면 다음과 같다.

$$\hat{\boldsymbol{y}} = \boldsymbol{h_2} = \sigma(\boldsymbol{h_1 W_2} + \boldsymbol{b_2})$$

$$\hat{\boldsymbol{y}} = (\hat{y}_1, \hat{y}_2, \dots, \hat{y}_l)$$

여기서 레이블 $\boldsymbol{y}$는 one-hot encoding을 사용하여 표현한다. 만약 $\boldsymbol{y}$가 총 10의 길이를 가지고 (10개의 클래스를 분류하는 분류기의 경우이다) 그 중 3번 클래스의 레이블을 가진다고 하면 $\boldsymbol{y} = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0)$ 의 형태를 가진다. 분류기의 최종 결과값인 $\hat{\boldsymbol{y}}$에서 가장 높은 값을 추출, 즉 argmax를 취해주어 레이블과의 비교를 할 수 있다.

Multi-Layer Perceptron의 정의는 이렇게 명확하게 내릴 수 있지만, 여러 계층을 쌓게 되면 손실함수에 대한 마땅한 학습 방법이 없다. 이를 해결한 방법이 바로 Error Backpropagation 알고리즘이다. 간단히 설명하자면 Chain rule을 이용하여 각 계층에 있는 파라미터들의 Gradient를 구하여 뒤에서부터 순차적으로 업데이트 해주는 방법이다. $\boldsymbol{a_2} = \boldsymbol{h_1 W_2} + \boldsymbol{b_2}$ 라고 한다면,

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = (\hat{\boldsymbol{y}} - \boldsymbol{y})^{\boldsymbol{2}}$$

$$\nabla_{\hat{\boldsymbol{y}}} L = \hat{\boldsymbol{y}} - \boldsymbol{y}$$

$$\nabla_{\boldsymbol{a_2}} \hat{\boldsymbol{y}} = \sigma(\boldsymbol{a_2})\left(1 - \sigma(\boldsymbol{a_2})\right)$$

$$\nabla_{\boldsymbol{W_2}} \boldsymbol{a_2} = \boldsymbol{h_1^T}, \qquad \nabla_{\boldsymbol{b_2}} \boldsymbol{a_2} = \boldsymbol{1} = (1, \dots, 1)^T$$

여기서 $\boldsymbol{1}$은 벡터임을 주의하자.

$$\therefore \nabla_{\boldsymbol{W_2}} L = \nabla_{\hat{\boldsymbol{y}}} L \odot \nabla_{\boldsymbol{a_2}} \hat{\boldsymbol{y}} \times \nabla_{\boldsymbol{W_1}} \boldsymbol{a_2} = (\hat{\boldsymbol{y}} - \boldsymbol{y}) \odot \{\sigma(\boldsymbol{a_2})\left(1 - \sigma(\boldsymbol{a_2})\right)\} \times \boldsymbol{h_1^T}$$

$$\therefore \nabla_{\boldsymbol{b_2}} L = \nabla_{\hat{\boldsymbol{y}}} L \odot \nabla_{\boldsymbol{a_2}} \hat{\boldsymbol{y}} \times \nabla_{\boldsymbol{b}} \boldsymbol{a_2} = (\hat{\boldsymbol{y}} - \boldsymbol{y}) \odot \{\sigma(\boldsymbol{a_2})\{\left(1 - \sigma(\boldsymbol{a_2})\right)\} \times \boldsymbol{1}$$

⊙ Hadamard product라고 하며 행렬 곱이 아닌, 차원이 같은 행렬에서 각 원소끼리의 곱이다.

이를 활용하여 파라미터를 업데이트 하면 다음과 같다

$$\boldsymbol{W_2^{new}} = \boldsymbol{W_2^{old}} - \eta \nabla_{\boldsymbol{W_2^{old}}} L$$

$$\boldsymbol{b_2^{new}} = \boldsymbol{b_2^{old}} - \eta \nabla_{\boldsymbol{b_2^{old}}} L$$

이제 우리가 알아야 할 Gradient는 $\nabla_{\boldsymbol{W_1}} L$, $\nabla_{\boldsymbol{b_1}} L$ 이다. 이 둘은 비슷한 원리로 hidden layer 1의 output이자 output layer의 input인 $\boldsymbol{h_1}$에 대한 Gradient를 알아야 chain rule을 사용하여 구할 수 있다. 이를 위해서는 $\nabla_{\boldsymbol{h_1}} \boldsymbol{a_2}$ 값을 알아야 $\boldsymbol{a_2}$에 대한 Gradient까지 구한 부분에 적용이 가능하다.

$$\nabla_{\boldsymbol{h_1}} \boldsymbol{a_2} = \boldsymbol{W_2^T}$$

$$\nabla_{\boldsymbol{h_1}} L = \nabla_{\hat{\boldsymbol{y}}} L \odot \nabla_{\boldsymbol{a_2}} \hat{\boldsymbol{y}} \times \nabla_{\boldsymbol{h_1}} \boldsymbol{a_2} = (\hat{\boldsymbol{y}} - \boldsymbol{y}) \odot \{\sigma(\boldsymbol{a_2})\{(1 - \sigma(\boldsymbol{a_2}))\} \times \boldsymbol{W_2^T}$$

위 사실을 활용하면 hidden layer 1의 파라미터들도 아래와 같은 연산을 통해 업데이트가 가능하다.

$$\nabla_{\boldsymbol{a_1}} \boldsymbol{h_1} = \sigma(\boldsymbol{a_1})(1 - \sigma(\boldsymbol{a_1}))$$

$$\nabla_{\boldsymbol{W_1}} \boldsymbol{a_1} = \boldsymbol{X^T}, \qquad \nabla_{\boldsymbol{b_1}} \boldsymbol{a_1} = \boldsymbol{1} = (1, \dots, 1)^T$$

$$\nabla_{\boldsymbol{W_1}} L = \nabla_{\boldsymbol{h_1}} L \odot \nabla_{\boldsymbol{a_1}} \boldsymbol{h_1} \times \nabla_{\boldsymbol{W_1}} \boldsymbol{a_2}$$

$$\nabla_{\boldsymbol{b_1}} L = \nabla_{\boldsymbol{h_1}} L \odot \nabla_{\boldsymbol{a_1}} \boldsymbol{h_1} \times \nabla_{\boldsymbol{b_1}} \boldsymbol{a_2}$$

$$\boldsymbol{W_1^{new}} = \boldsymbol{W_1^{old}} - \eta \nabla_{\boldsymbol{W_1^{old}}} L$$

$$\boldsymbol{b_1^{new}} = \boldsymbol{b_1^{old}} - \eta \nabla_{\boldsymbol{b_1^{old}}} L$$

이제까지 설명한 방법은 하나의 데이터에 대한 설명이었다. 만약, 다수의 데이터에 대해서 한번에 학습하기 위해서는 Gradient의 합 또는 평균을 사용하여 업데이트 하면 된다. 만약 합을 사용한다면 평균을 사용할 때에 비해서 learning rate를 더 작게 사용해야 된다. 따라서 평균이나 합 둘중 어느 것을 사용해도 무리가 없지만, learning rate 조정에 신경 써야 한다. 이 때의 입력 데이터는,

$$\boldsymbol{X^i}, \qquad i = \{1, 2, \dots, N\}$$

같이 나타낼 수 있고, 기존 $\boldsymbol{X}$에 super script i 는 i 번째 데이터라는 의미를 부여한 것이다. 마찬가지로 출력과 레이블(정답) 데이터에도 같은 notation을 적용하여 나타내어 손실 함수를 다시 정의하면,

$$L(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \sum_{\boldsymbol{i=1}}^{\boldsymbol{N}} (\hat{\boldsymbol{y}^i} - \boldsymbol{y^i})^2$$

또한, 데이터 크기가 큰 이미지 데이터 등은 모든 데이터를 한번에 업데이트 하기가 힘들기 때문에 mini batch라는 개념을 적용해 데이터의 일부를 iterative 하게 학습한다.

한 데이터셋을 전부 사용하여 학습하는 iteration을 epoch라고 하며, 여러 epoch을 돌면서 모델을 학습하여야 한다.

이외의 연산은 동일하다.

학습 시 데이터는 train, validation, test 로 보통 나누어 사용한다. Train은 학습에, validation은 학습에 사용하지 않지만, 학습 과정에서 검증하는데 사용한다. 학습에 사용하지 않았기 때문에 학습이 제대로 이루어지는지 확인할 수 있다. 마지막으로 test 데이터는 모델을 평가하기 위한 데이터로 보통 레이블을 공개하지 않고 공정한 모델 평가에 사용된다.

Program Explanation

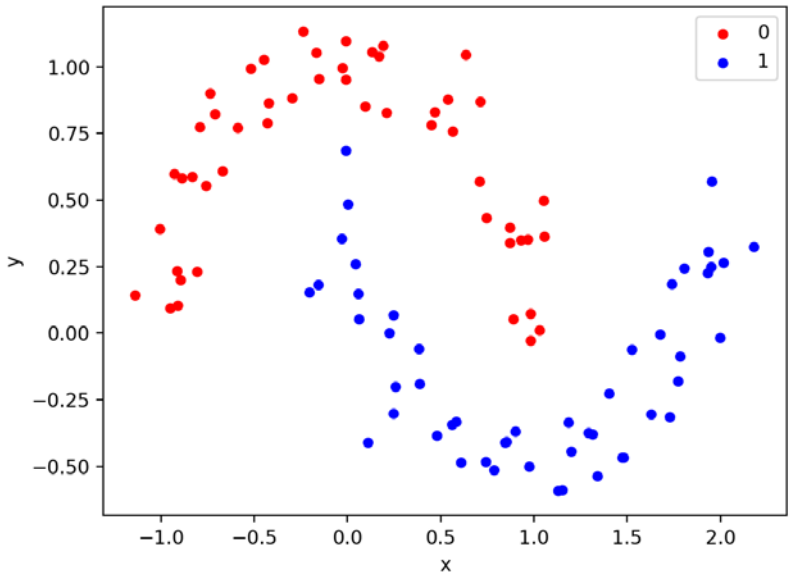


그림 6. Two moon dataset

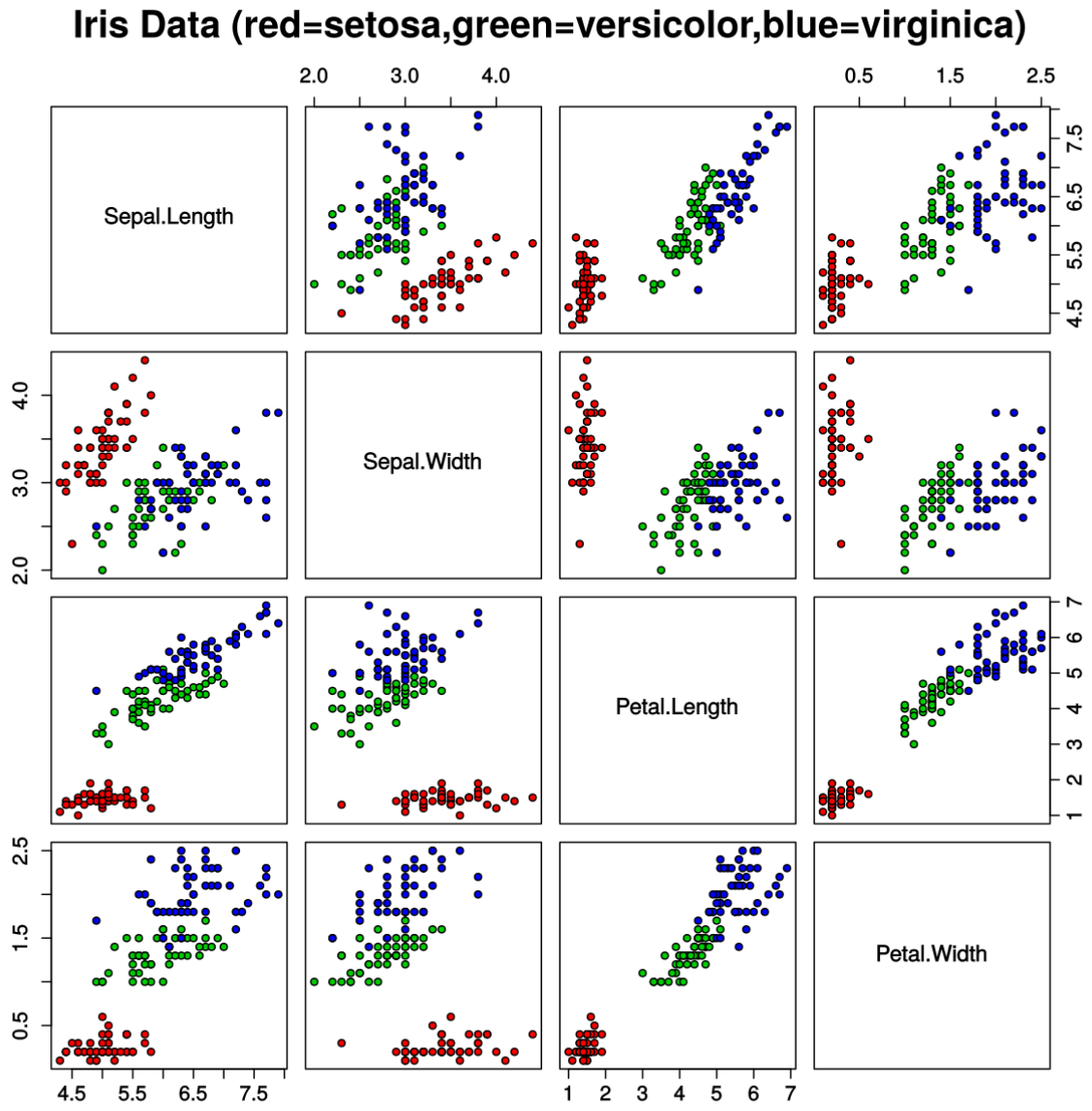


그림 7. iris dataset

Two moon dataset 과 iris dataset은 그림 6과 그림 7에서 알 수 있듯이 non-linear한. 위 데이터를 학습시킬 수 있는 Classifier를 학습시키고 verify하는 코드를 작성한다. 프로그램은 cmd에서 다음과 같은 인자를 받아 실행되어야 한다.

- dataset_type[str] → two moon 또는 iris 데이터를 결정하는 인자
- feature_num[int] → 입력 데이터의 feature 개수
- class_num[int] → 분류하고자 하는 클래스 개수
- hidden_layer_num[int] → hidden layer의 개수
- hidden_layer_neurons[str] → 각 hidden layer의 뉴런의 개수. string으로 받아 int 형식의

array로 변환하여 사용하여야 함 ex) "10 10 10" → [10, 10, 10]

- epochs[int] → 학습 횟수 (epoch)
- learning_rate[float] → learning rate 수치
- train_data_dir[str] → 학습 데이터 파일 위치
- train_data_num[int] → 학습 데이터 개수
- validation_data_dir[str] → validation 데이터 파일 위치
- validation_data_num[int] → validation 데이터 개수
- test_data_dir[str] → test 데이터 파일 위치
- test_data_num[int] → test 데이터 개수
- test_output_dir[str] → test 데이터를 통해 얻은 결과 출력 경로

필수 클래스는 Dataloader, Layer, Loss, Mlp 4개이다. **클래스 추가나 제거가 있으면 보고서에 기록.**

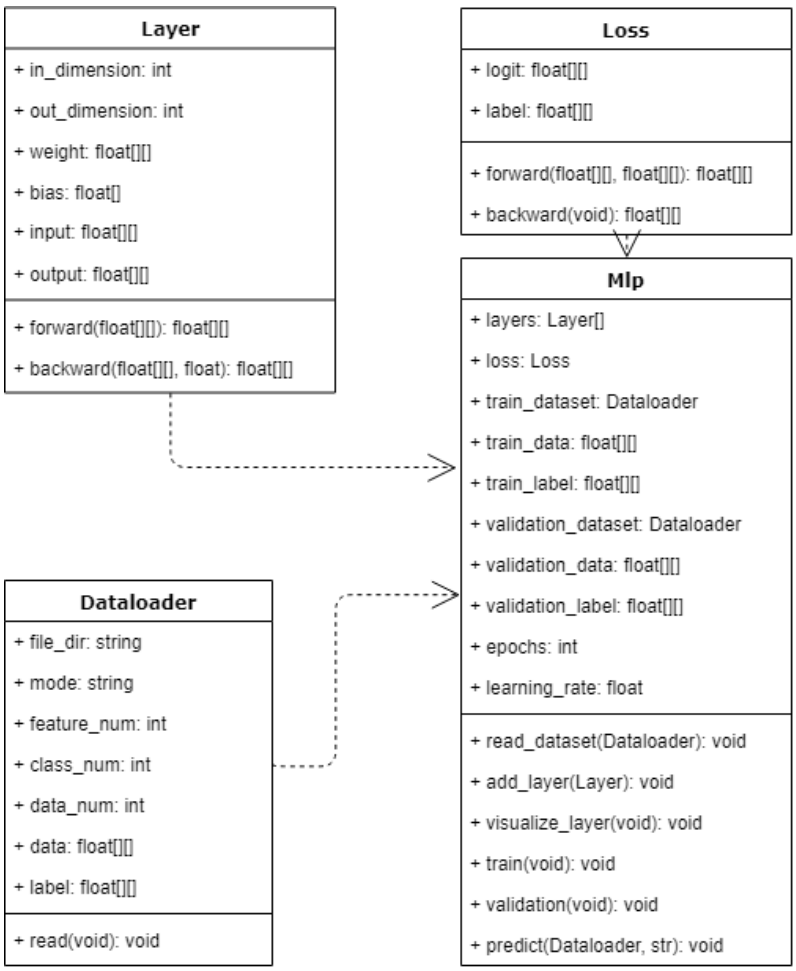


그림 8. Class diagram 예시

그 외 필요한 멤버 변수나 멤버 함수 추가도 허용하지만, 이 경우 반드시 보고서에 그 이유를 명시해야 한다. (입력과 출력 데이터 형식은 위를 따르지 않아도 무방함.) 멤버 변수에 접근하기 위한 생성자(오버로딩), 소멸자(오버로딩), get 멤버 함수, set 멤버 함 수의 경우 자유롭게 구성(추가/삭제)해도 무관하다.

- **Dataloader**

맴버 변수	맴버 함수
● file_dir: 데이터 파일 경로 ● mode: 'train' 'val' 'test' 모드를 지정 ● feature_num: 데이터의 feature 개수 (차원) ● class_num: 분류할 클래스 개수 ● data_num: 데이터 개수 ● data: 학습 데이터 ● label: 레이블 – **test 모드에서는 사용하지 않음**	● void read(void): 데이터를 읽어와서 data와 label에 각각 저장함 – test mode에서는 label에 데이터를 저장하지 않음

- **Layer**

맴버 변수	맴버 함수
● in_dimension: 입력 차원 ● out_dimension: 출력 차원 ● weight: weight 파라미터 ● bias: bias 파라미터 ● input: forward 함수에서 받은 입력을 저장 - backward에서 사용 ● output: forward 함수를 통해 얻은 결과를 저장 - backward에서 사용	● float[][] forward(float[][]): **직전 layer의 output(첫 layer인 경우 입력 데이터)을 input으로 받아** $\sigma(\boldsymbol{hW}+\boldsymbol{b})$ 를 계산하여 출력 ● float[][] backward(float[][], float): forward를 통해 얻은 값과 이전 Gradient 값을 통해 weight와 bias를 업데이트 하고 다음 Layer에 전달할 Gradient를 출력

- **Loss**

맴버 변수	맴버 함수
Logit: 신경망의 최종 결과값 - forward에서 입력 받아 저장 Label: 레이블 - forward에서 입력 받아 저장	● f float forward(float[][], float[][]): logit과 label을 입력 받아 손실 함수 값을 출력 ● float[][] backward(void): 다음 Layer에 전달할 Gradient를 logit과 label을

	사용하여 출력

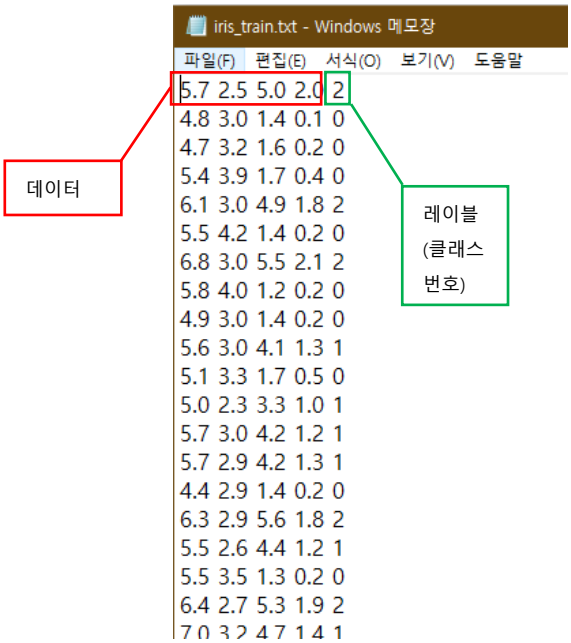
- **Mlp**

맴버 변수	**맴버 함수**
layers: Layer 오브젝트 array – hidden layer개수에 따라 길이 변동 loss: Loss 오브젝트 train_dataset: train에 사용할 Dataloader 오브젝트 train_data: train_dataset에서 얻은 데이터를 저장 train_label: train_dataset에서 얻은 레이블 저장(one_hot) validation_dataset: validation에 사용할 Dataloader 오브젝트 validation_data: validation_dataset에서 얻은 데이터를 저장 validation_label: validation_dataset에서 얻은 레이블 저장(one_hot) epochs: epoch learning_rate: learning rate	void read_dataset(Dataloader): 데이터를 읽어와서 data와 label에 각각 저장함 void add_layer(Layer): layers에 Layer 오브젝트를 추가함 void visualize_layer(void): layer 계층 구조를 출력 void train(void): 학습 – 각 epoch마다 loss와 정확도를 출력 void validation(void): 검증 – validation set에 대해서 loss와 정확도를 출력 void predict(Dataloader, str): 입력한 Dataloader(test임) 의 출력 결과를 텍스트 파일에 저장

학습 데이터 형식

각 라인에 샘플 하나이며, 스페이스로 나뉘어짐. 마지막 줄은 클래스 번호임.

Two moon의 경우 0과 1로 되어있으며, iris는 0, 1, 2번으로 되어 있음.



Test용 데이터셋은 마지막 클래스 번호가 없으므로, 이에 대한 처리가 필요함.

각 데이터셋은 train, val, test 세 분류로 나뉘어 있으며, 각각 150, 20, 30 개의 데이터로 구성되어 있다.

프로그램 예시

1) 학습

```
==================================================
dense layer  0 in_dimension:  2 out_dimension : 10
dense layer  1 in_dimension:  10 out_dimension : 10
dense layer  2 in_dimension:  10 out_dimension : 2
==================================================
epoch:  1  train_loss:  0.7043501504482713 train accuracy:  0.4666666666666667
epoch:  2  train_loss:  0.6994540370968447 train accuracy:  0.5333333333333333
epoch:  3  train_loss:  0.6971733961238578 train accuracy:  0.5333333333333333
epoch:  4  train_loss:  0.6952596833630033 train accuracy:  0.5333333333333333
epoch:  5  train_loss:  0.6934695919622949 train accuracy:  0.5333333333333333
epoch:  6  train_loss:  0.6917623738949743 train accuracy:  0.5333333333333333
epoch:  7  train_loss:  0.6900977602917308 train accuracy:  0.5333333333333333
epoch:  8  train_loss:  0.6884388697962551 train accuracy:  0.5333333333333333
epoch:  9  train_loss:  0.6867510702854607 train accuracy:  0.5333333333333333
epoch:  10  train_loss:  0.6850011179126602 train accuracy:  0.5333333333333333
epoch:  11  train_loss:  0.6831564797443355 train accuracy:  0.5333333333333333
epoch:  12  train_loss:  0.6811848271295383 train accuracy:  0.5333333333333333
epoch:  13  train_loss:  0.6790537027765567 train accuracy:  0.5333333333333333
epoch:  14  train_loss:  0.6767303812166154 train accuracy:  0.5466666666666666
```

2) Validation

```
epoch:  980  train_loss:  0.3272361790942123 train accuracy:  1.0
epoch:  981  train_loss:  0.3272181599298895 train accuracy:  1.0
epoch:  982  train_loss:  0.32720019195050337 train accuracy:  1.0
epoch:  983  train_loss:  0.3271822749355978 train accuracy:  1.0
epoch:  984  train_loss:  0.3271644086659796 train accuracy:  1.0
epoch:  985  train_loss:  0.3271465929237018 5 train accuracy:  1.0
epoch:  986  train_loss:  0.3271288274920709 5 train accuracy:  1.0
epoch:  987  train_loss:  0.3271111121556213 3 train accuracy:  1.0
epoch:  988  train_loss:  0.3270934467001160 3 train accuracy:  1.0
epoch:  989  train_loss:  0.3270758309125348 train accuracy:  1.0
epoch:  990  train_loss:  0.3270582645810671 train accuracy:  1.0
epoch:  991  train_loss:  0.3270407474951021 5 train accuracy:  1.0
epoch:  992  train_loss:  0.3270232794452214 7 train accuracy:  1.0
epoch:  993  train_loss:  0.3270058602231903 train accuracy:  1.0
epoch:  994  train_loss:  0.3269884896219490 4 train accuracy:  1.0
epoch:  995  train_loss:  0.3269711674356056 train accuracy:  1.0
epoch:  996  train_loss:  0.3269538934594264 train accuracy:  1.0
epoch:  997  train_loss:  0.326936667489829 train accuracy:  1.0
epoch:  998  train_loss:  0.3269194893243742 train accuracy:  1.0
epoch:  999  train_loss:  0.3269023587617569 train accuracy:  1.0
epoch:  1000  train_loss:  0.3268852756017997 train accuracy:  1.0
validation_loss:  0.3281289893955214 validation accuracy:  1.0
```

3) **Test 결과 예시**

실험조건

각 데이터셋에 대해서 학습이 잘되는 hyper-parameter(epochs, learning_rate, hidden layer 층수, neuron의 개수 등)를 찾는 실험에 대한 과정 및 결과를 보고서에 제출해야 함. **필수 실험은 다음**

과 같음

Epochs	Learning rate	Hidden layer 개수	Neuron 개수
100	0.01	1	10
500	0.05	2	10, 10
1000	0.01	3	30, 100, 30
1000	0.03	2	100, 100

위 실험 내용을 각 데이터셋에 대해 학습 후, **과정과 결과를 보고서에 기입하고**, 자신이 찾은 최적의 값을 찾은 과정과 결과를 보고서에 작성해야 함.

학습한 네트워크로 각 데이터셋에 존재하는 test 데이터셋(정답 레이블이 없는)에 대해 [Test 결과 예시]와 같은 형식으로 출력한 텍스트 파일을 과제 제출 시 소스코드와 함께 제출해야 함

Requirements

- 본 과제의 프로그램은 리눅스 환경(프로그래밍 실습 서버 또는 MinGW 또는 맥에서의 터미널 등 그와 유사한 환경) 또는 **윈도우 환경(visual studio 2019)에서 컴파일 및 구동 가능하도록 작성한다.**
- 본 과제의 소스코드는 다음과 같이 9개의 파일로 구성하기를 권장한다. 다른 형태의 파일 구 성도 허용하지만, 그에 대한 이유를 보고서에 명시하여야 한다.
  - 메인 함수를 포함하는 .cpp 파일
  - 각 객체 클래스(Dataloader, Layer, Loss, Mlp)의 선언 및 정의에 대한 .cpp 및 .h 파 일
- 리눅스 환경의 경우 Makefile을 작성하여 컴파일이 가능하도록 해야 하며, 작성된 Makefile 역시 제출해야 한다. (**make를 통한 컴파일 실패 시, 컴파일이 안되는 것으로 간주, 윈도우 환경의 경우 해당사항 없음**)
- 실습 서버의 접속 방법은 첨부된 [프로그래밍 실습환경 사용가이드]의 '실습 시스템 접속 방법'을 참조한다.
- 상기한 환경에서의 프로그램 컴파일 및 실행 과정에 대한 설명과 이미지 모두 보고서에 포함해 야 한다
- 상속(Inheritance)은 허용하지만, 이에 대한 설명을 보고서에 기입하여야 함.
- STL이나 Boost 등의 라이브러리는 사용하지 않는다. (사용할 경우 0점)
- 구성원들의 모든 멤버 변수는 'private'이어야 하며, 이들 멤버 변수에 대한 접근은 모두 'public' 멤버 함수들을 통해서만 이루어져야 한다
- 위에 설명된 프로그램의 형식을 모두 만족하여야 한다

- 보고서는 사용된 클래스의 멤버 변수/함수에 대한 설명을 포함하여야 한다.
- 보고서에 위 요구사항 중 만족한 것과 만족하지 못한 것을 정확하게 명시하여야 하며, 정확하게 명시하지 않을 경우 '프로그램 설계 및 구현', '보고서 구성 및 내용, 양식' 점수가 감점될 수 있다.
- 보고서 및 소스코드는 LMS의 과제 디렉토리에 업로드한다.
- 채점 기준은 AssnReadMe_Fall_2019.pdf 파일을 참고한다.
- **구현에 어려움이 있을 시 Assign1_Pseudocode를 참고하여 구현한다.**