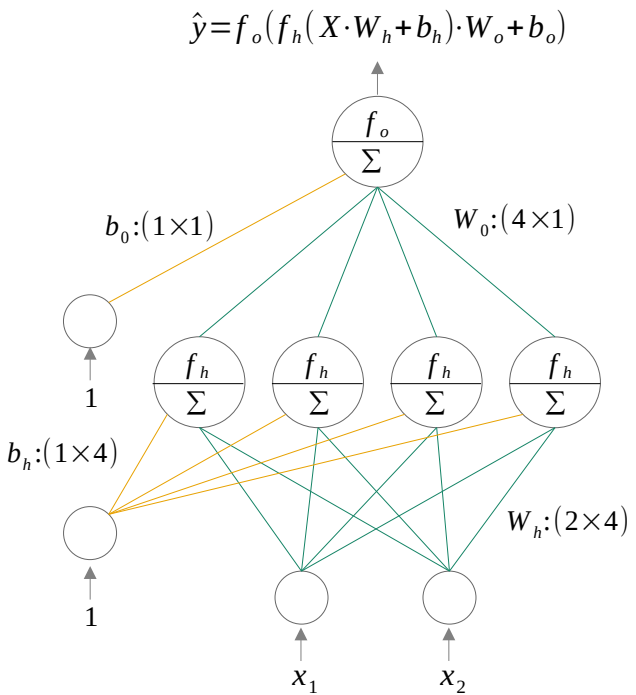


1. Artificial Neural Network

Part 1: Introduction to ANN

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

1. Artificial Neural Network (ANN)

1-1. ANN architecture
1-2. Training process of ANN
1-3. Implementing ANN
1-4. Classification and Regression

2. Techniques for improving ANN performance

2-1. Optimizers
2-2. Error Backpropagation
2-3. Tensorflow and Keras
2-4. Regularization
2-5. Dropout
2-6. Batch Normalization
2-7. Weights initialization
2-8. Highway Network

3. Recurrent Neural Network (RNN)

3-1. Simple RNN
3-2. LSTM
3-3. Peephole LSTM, GRU

4. Attention Network

- For time series prediction, not for NLP

4-1. Seq-to-Seq network
4-2. Attention network
4-3. Self attention (Transformer)

5. Convolutional Neural Network (CNN)

5-1. 1D, 2D, 3D convolution
5-2. Convolutional LSTM

6. Autoencoder

6-1. Dimension reduction
6-2. Noise reduction
6-3. Anomaly detection
6-4. Variational Autoencoder (VAE)

7. Generative Adversarial Networks (GAN)

7-1. Standard GAN
7-2. DCGAN, Unrolled GAN, LSGAN WGAN, WGAN-GP, D2GAN

8. Unsupervised Learning

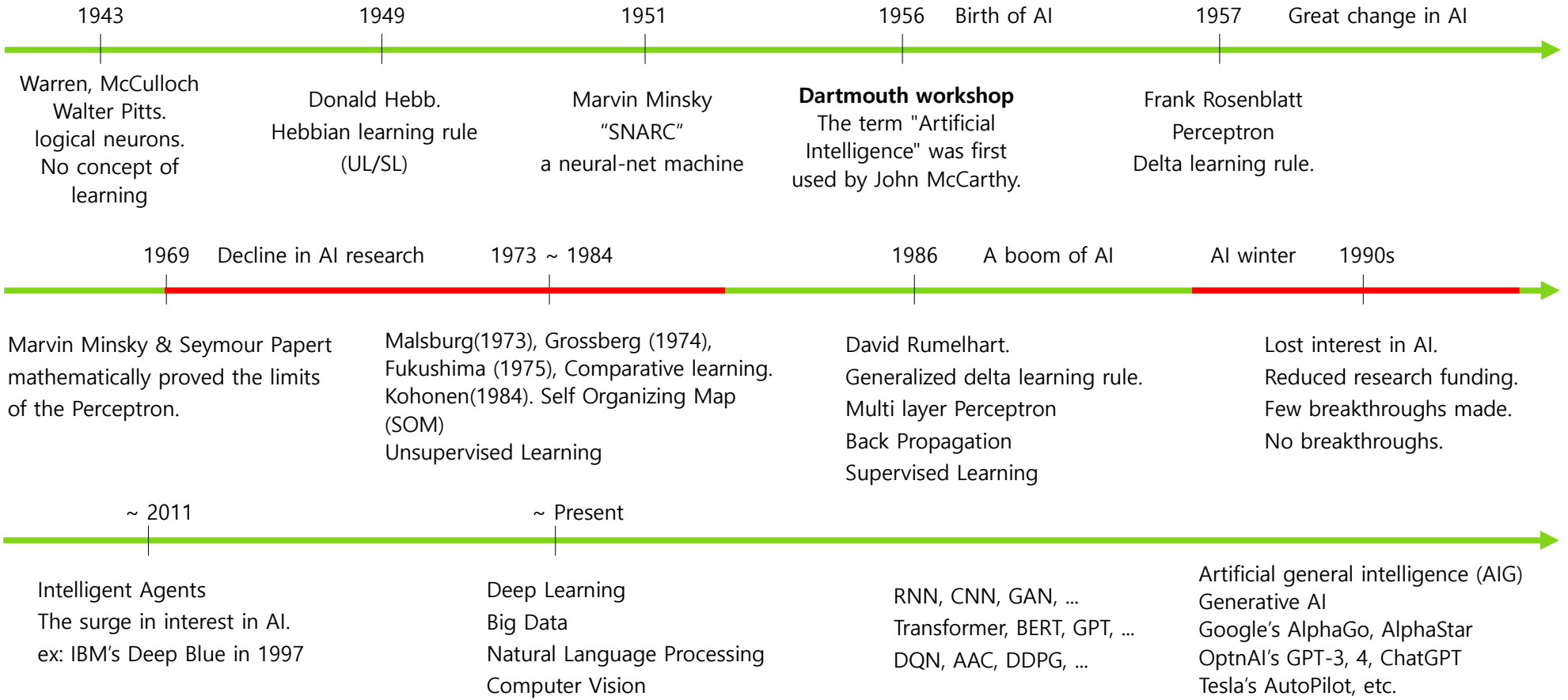
8-1. Hebbian learning rule
8-2. Instar rule
8-3. Comparative learning
8-4. Self Organizing Map (Kohonen network)

MX-AI

MX-AI

Brief history of Artificial Intelligence (AI)

- AI has a long history and has gone through several booms and winters to reach what it is today.

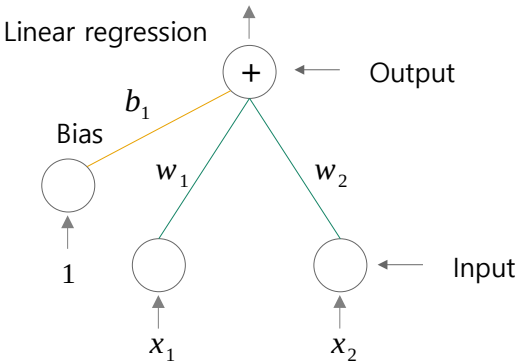


MX-AI

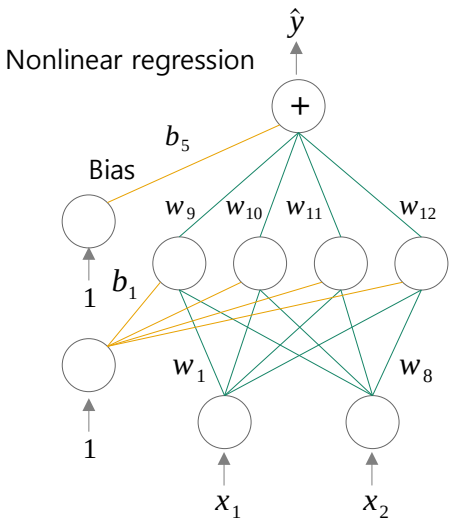
■ Feed Forward Network

- Regression and classification problems can be expressed in the form of a feed forward network.
- We can fit the network model to the training data to get w and b, and predict the target value of the test data.

$$\hat{y} = w_1 x_1 + w_2 x_2 + b$$



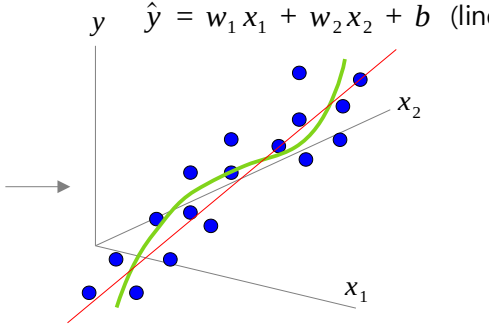
[Feed Forward Network]



Regression problem

[training data]

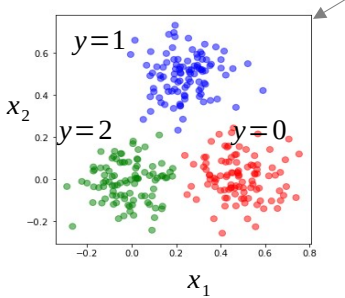
feature		target
x_1	x_2	y
0.16	0.01	0.54
-0.09	-0.03	-0.23
0.06	-0.08	1.82
0.58	-0.1	-0.92
0.46	-0.07	2.01
0.23	0.54	1.30
0.55	0.07	0.37
⋮	⋮	⋮



[test data]

feature		target
x_1	x_2	y
0.5	0.4	?
0.19	-0.21	?

Classification problem



[training data]

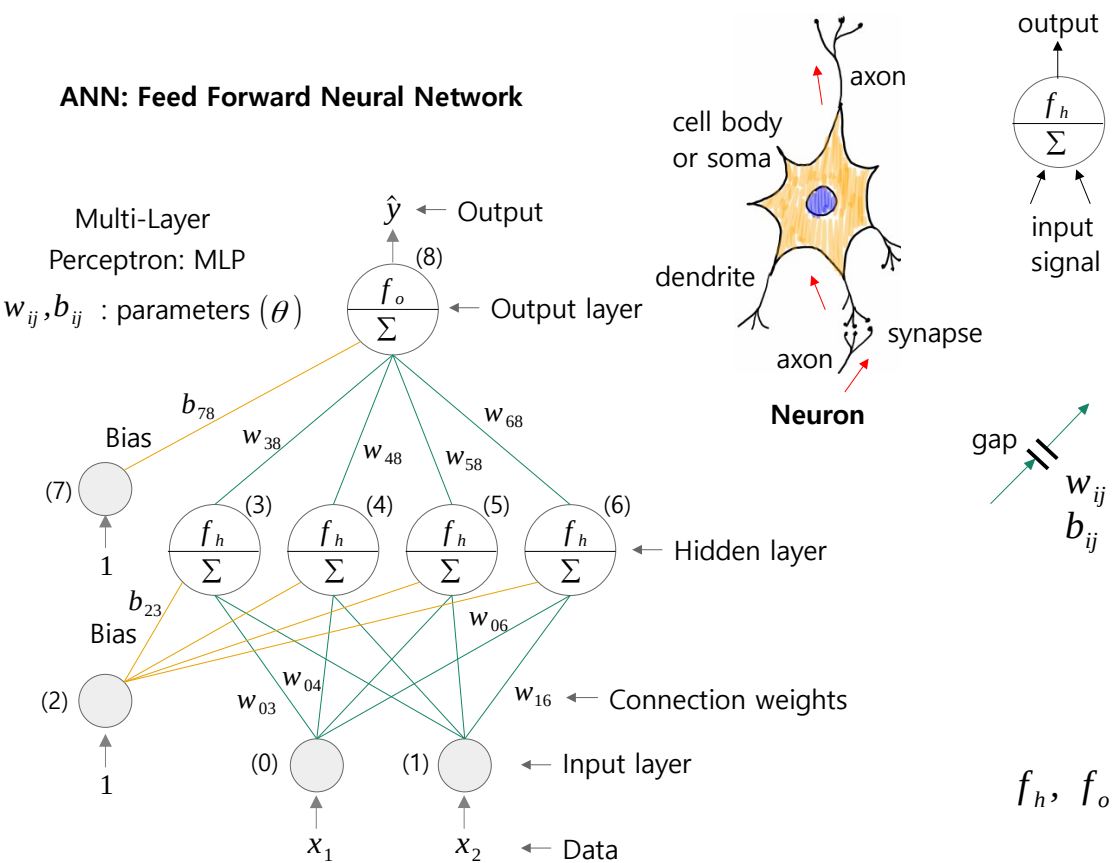
feature		target
x_1	x_2	y
0.16	0.01	0
-0.09	-0.03	0
0.06	-0.08	0
0.58	-0.1	2
0.46	-0.07	2
0.23	0.54	1
0.55	0.07	2
⋮	⋮	⋮

[test data]

feature		target
x_1	x_2	y
0.5	0.4	?
0.19	-0.21	?

MX-AI

■ Basic Structure of ANN



$f_h,\ f_o$

Artificial Neuron – The cells in the input layer pass the input signal as is to the upper node. The cells in the hidden layer and output layer add all input signals received from lower cells (∑), convert the result using function f, and output it. A bias is connected to each cell in the hidden layer and output layer. This always outputs 1.

Connection weights – Connect two neurons (i, j) with a weight (w or b). The output of the lower neuron is multiplied by the weight and passed to the upper neuron. The weights w and b can be considered the synaptic gap, i.e. the small gap between the output of the lower neuron and the input of the higher neuron. When w or b is small, i.e. when the gap is wide, only a small portion of the information from the lower neuron is passed on to the upper neuron. And when w is large, i.e., when the gap is narrow, much of the information from lower neuron is passed on to the upper neuron.

* Input signal of neuron 6 = x_1 * w_{06}, and x_2 * w_{16}, + 1 * b_{23}

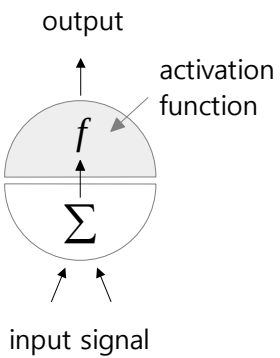
Activation function – Controls the output size of the neuron. Neurons in the hidden layer use non-linear activation function, and neurons in the output layer use linear or non-linear activation function.

* Output signal of neuron 6 = $f_h(x_1 * w_{06}, + x_2 * w_{16}, + 1 * b_{23})$

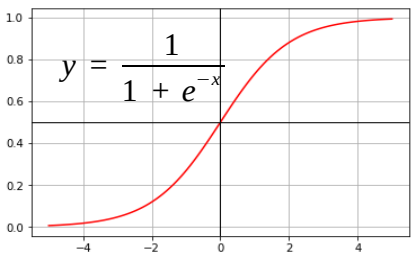
MX-AI

Activation function

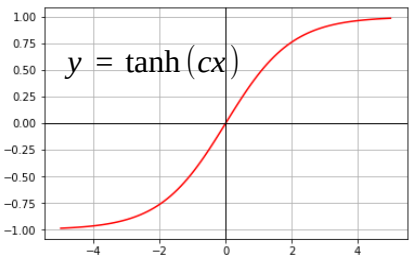
- Activation function is used to convert the output size of a neuron into a desired range of values. For example, in the case of binary classification, a sigmoid-like function is used in the output layer because the output value must be between 0 and 1, or between -1 and +1. And in the case of regression, a linear function is used because the output value should not change.
- Another purpose of the activation function is to introduce non-linearity into the hidden layer. If you do not use an activation function or use a linear function in the hidden layer, the network behaves as a single-layer neural network for linear learning. Therefore, for nonlinear learning, a nonlinear function must be used in the hidden layer.
- ReLU, tanh, Leaky ReLU, softplus, etc. are used in the hidden layer for nonlinear learning. In particular, ReLU is most widely used in the hidden layer.



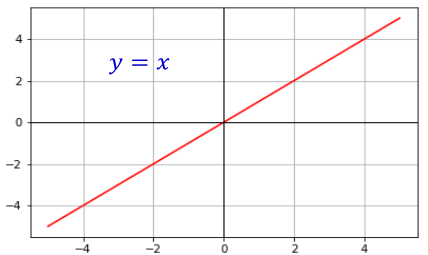
sigmoid



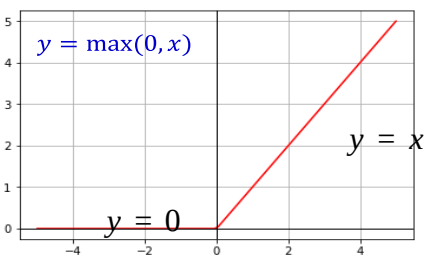
hyperbolic tangent



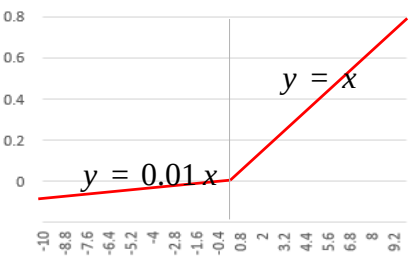
linear



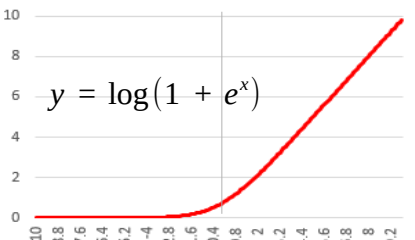
Rectified Linear Unit (ReLU)

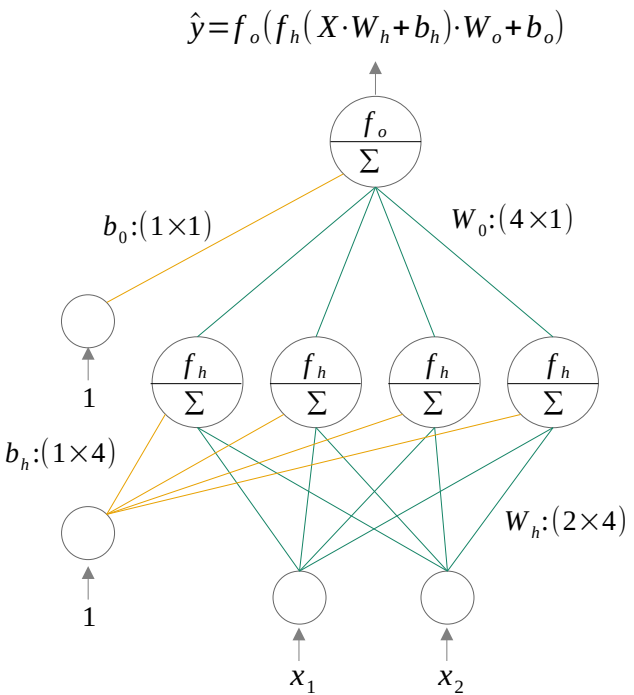


Leaky ReLU



softplus





1. Artificial Neural Network

Part 2: Operations of ANN and Objective function

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

Operations of ANN: Prediction process

- Once ANN training is complete, W_h, b_h, W_o, and b_o are determined and the network is ready to make predictions.
- The computational process of ANN is mainly matrix multiplication.

$$\hat{y} = f_o(H \cdot W_o + b_o) = f_o(f_h(X \cdot W_h + b_h) \cdot W_o + b_o)$$

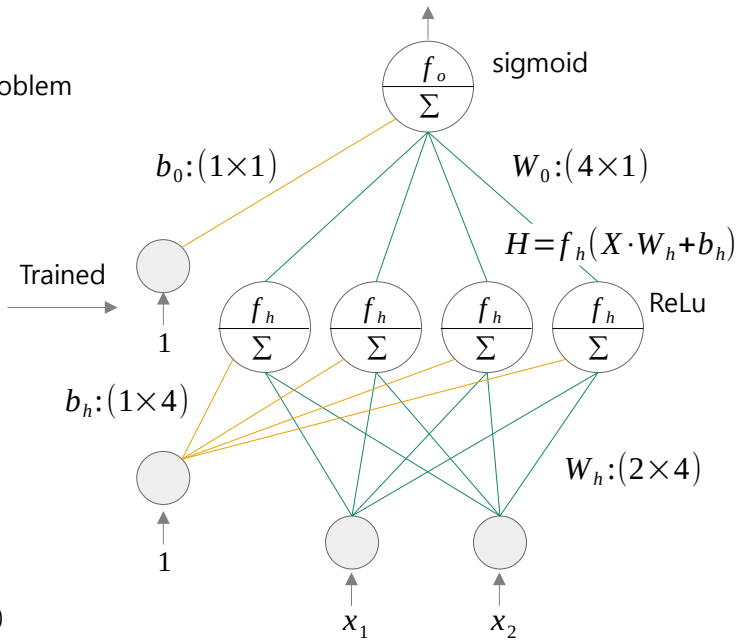
Binary classification problem

feature (X) | target

x_1	x_2	y
-0.05	0.14	0
0.04	0.17	0
0.67	-0.09	1
0.51	0.08	1
0.00	0.09	0
0.11	0.09	0
0.62	-0.06	1
0.32	-0.03	1
0.06	-0.10	0
0.58	0.12	1
0.46	-0.02	1
0.14	0.03	0

(n x 2) (n x 1)

Trained



Trainable parameters = [W_h, b_h, W_o, b_o]

```
def ReLU(x): return np.maximum(0, x)
def sigmoid(x): return 1. / (1. + np.exp(-x))

X = np.array([[-0.05,  0.14],
              [0.04,  0.17],
              [0.67, -0.09],
              [0.51,  0.08],
              [0.00,  0.09],
              [0.11,  0.09],
              [0.62, -0.06],
              [0.32, -0.03],
              [0.06, -0.10],
              [0.58,  0.12],
              [0.46, -0.02],
              [0.14,  0.03]])
Wh = np.array([[ 1.17, -1.20, -1.07,  0.58],
               [-1.31, -0.12,  1.11, -0.68]])
bh = np.array([0.18, 0.66, 0.70, 0.12])
Wo = np.array([[1.17], [-0.81], [-0.67], [1.48]])
bo = np.array([-0.38])

y_pred = sigmoid(np.dot(ReLU(np.dot(X, Wh) + bh), Wo) + bo)
```

Assume we know these parameters.

y_pred:	$\hat{y}$: P(y=1)	$\hat{y}$>0.5	y : actual target value
[0.17],		0	0
[0.2],		0	0
[0.85],		1	1
[0.68],		1	1
[0.22],		0	0
[0.3],		0	0
[0.82],		1	1
[0.59],		1	1
[0.39],		0	0
[0.7],		1	1
[0.7],		1	1
[0.37]])		0	0

Accuracy = 100%

MX-AI

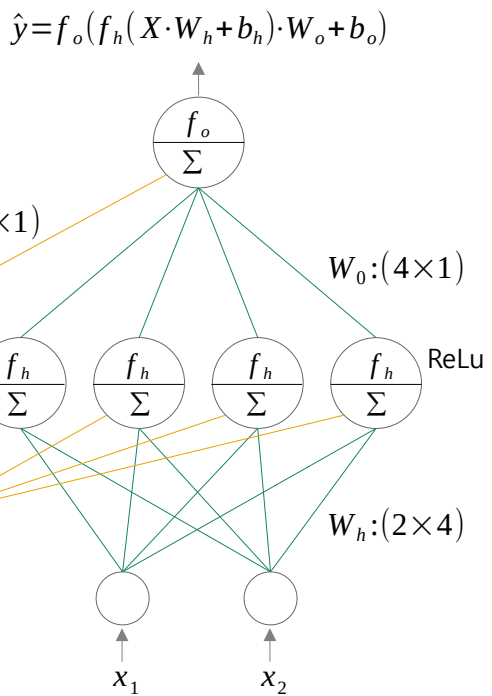
Objective function

- In order to train an ANN, a loss function, which is the objective function, is needed. In regression analysis, the mean square error is used as the loss function, in binary classification, binary cross entropy is used, and in multiclass classification, cross entropy is used as the loss function.
- Regression uses a linear activation function for the neurons in the output layer, and binary classification uses a sigmoid activation function. And multiclass classification uses a softmax activation function.

feature (X) | target

x_1	x_2	y
-0.05,	0.14	0
0.04,	0.17	0
0.67,	-0.09	1
0.51,	0.08	1
0.00,	0.09	0
0.11,	0.09	0
0.62,	-0.06	1
0.32,	-0.03	1
0.06,	-0.10	0
0.58,	0.12	1
0.46,	-0.02	1
0.14,	0.03	0

train



- Loss function

1) Mean squared error (MSE) for Regression

$$L(w,b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}(y_i-\hat{y}_i)^2$$

(N is the number of data points)

2) Binary Cross Entropy (BCE) for binary classification

$$L(w,b) = -\frac{1}{N}\sum_{i=1}^{N}[y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y})]$$

3) Cross Entropy (CE) for multiclass classification

$$L(w,b) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{C} y_{i,k}\log(\hat{y}_{i,k})$$

C is the number of classes in y, and it is the number of neurons in output layer.

- Objective function

$$\min_{w,b} L(w,b)$$

Our goal is to find parameters w and b that make the loss function as small as possible.

MX-AI

■ Objective function

- As shown below, the y classes predicted by ANN models (A) and (B) are both identical. However, the MSE and BCE of model-(A) are smaller than those of model-(B). This means that the model-(A) has better performance.
- The accuracy of both models is 100%, but the predictions of model-(B) are somewhat unstable. This is because Model-(B) predicts y-hat 0.49 as class 0 and y-hat 0.51 as class 1.
- Although MSE can be used for classification, it is better to use BCE or CE.

ANN model-(A)

actual y / predicted y

y	$\hat{y}$	$\hat{y}>0.5$
0	0.17	0
0	0.2	0
1	0.85	1
1	0.68	1
0	0.22	0
0	0.3	0
1	0.82	1
1	0.59	1
0	0.39	0
1	0.7	1
1	0.7	1
0	0.37	0

```
import numpy as np

y = np.array([0,0,1,1,0,0,1,1,0,1,1,0])
y_hat = np.array([0.17, 0.20, 0.85, 0.68,
                  0.22, 0.30, 0.82, 0.59,
                  0.39, 0.70, 0.70, 0.37])

mse = np.mean(np.square(y - y_hat))
bce = -np.mean(y * np.log(y_hat) +
               (1-y) * np.log(1-y_hat))
```

MSE: 0.083
BCE: 0.330

ANN model-(B)

actual y / predicted y

y	$\hat{y}$	$\hat{y}>0.5$
0	0.47	0
0	0.35	0
1	0.65	1
1	0.58	1
0	0.32	0
0	0.39	0
1	0.58	1
1	0.59	1
0	0.49	0
1	0.60	1
1	0.51	1
0	0.37	0

```
import numpy as np

y = np.array([0,0,1,1,0,0,1,1,0,1,1,0])
y_hat = np.array([0.47, 0.35, 0.65, 0.58,
                  0.32, 0.39, 0.58, 0.59,
                  0.49, 0.60, 0.60, 0.37])

mse = np.mean(np.square(y - y_hat))
bce = -np.mean(y * np.log(y_hat) +
               (1-y) * np.log(1-y_hat))
```

MSE: 0.168
BCE: 0.526

MX-AI

Understanding cross entropy from the perspective of information theory.

- **Shannon entropy:** The concept of information entropy was introduced by Claude Shannon in his 1948 paper "A Mathematical Theory of Communication" and is also known as Shannon entropy. He defined the amount of information as being inversely proportional to probability, with the idea that "the amount of information for high-probability events is small, and the amount of information for low-probability events is large." The probability of two independent events occurring simultaneously is the product of their respective probabilities. However, to calculate the total information amount of two events, it is reasonable to add the information amount of each event. Therefore, it makes sense to define the amount of information as the logarithm of the inverse of probability. And the amount of information is finally defined as the average of possible events.

  amount of information → $\frac{1}{p}$ → $\log(\frac{1}{p})$ → $\sum_i p_i \cdot \log(\frac{1}{p_i}) = -\sum_i p_i \cdot \log(p_i) = H(p)$ (Shannon used log2)

- **KL divergence**: It is the difference in the amount of information between the two probability distributions p and q. And it is defined to be a positive number.

  y: actual class probability distribution $\hat{y}$: predicted class probability distribution

  Difference in the amount of information between y and y hat: $\Delta I = \log(\frac{1}{\hat{y}}) - \log(\frac{1}{y}) = -\log(\hat{y}) + \log(y)$ ← The smaller the difference, the more similar the two distributions are.

  Expectation: $E[\Delta I] = -\sum_i y_i \cdot \log(\hat{y}_i) + \sum_i y_i \cdot \log(y_i) \equiv D_{KL}(\hat{y} \| y) \geq 0$ * This inequality is established by Gibb's inequality theorem.

- **Cross Entropy (CE)**

  $\min_{\hat{y}} D_{KL}(\hat{y} \| y)$ → $\min_{\hat{y}}[-\sum_i y_i \cdot \log(\hat{y}_i) + \sum_i y_i \cdot \log(y_i)]$ → $\min_{\hat{y}}[-\sum_i y_i \cdot \log(\hat{y}_i)]$: The second term is independent of y-hat.

  $CE = -\sum_i y_i \cdot \log(\hat{y}_i)$: Cross Entropy → Minimizing CE makes the distributions of $\hat{y}$ and y similar, so it can be used as the objective function of ANN.

MX-AI

Understanding cross entropy from the perspective of information theory.

- According to the results below, we can see that as CE gets smaller, the predicted probability y-hat gets exponentially closer to the actual probability y.
- Therefore, it is very reasonable to use cross-entropy as the objective function in classification models.

i = 0 1 2

$y = [1,0,0]$: actual probability (one-hot encoded)

$\hat{y} = [0.7,0.1,0.2]$: predicted probability

$$CE = -\sum_i y_i \log(\hat{y}_i)$$

- Jensen's inequality for log(x)

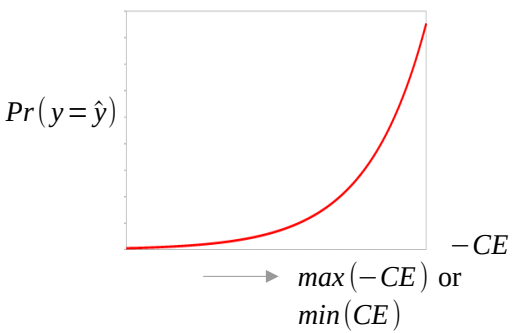
$$\log(E[\hat{y}_i]) \geq E[\log(\hat{y}_i)]$$

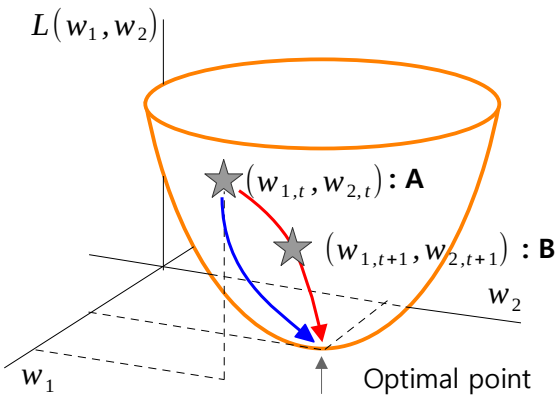
$$\log(\sum_i y_i \hat{y}_i) \geq \sum_i y_i \log(\hat{y}_i) = -CE$$

$$\sum_i y_i \hat{y}_i = 1\times0.7+0\times0.1+0\times0.2 = 0.7 \rightarrow Pr(y=\hat{y})$$

$\log(Pr(y=\hat{y})) \geq -CE$ Making the negative CE larger, i.e. making the CE smaller, increases the lower bound and therefore increases the probability that y-hat will be y.

$Pr(y=\hat{y}) \geq e^{-CE}$ $max(-CE) \rightarrow min(CE) \rightarrow Pr(y=\hat{y})$ increases exponentially.





$$w_{i,t+1} = w_{i,t} - \alpha \nabla_{w_i} L(w_1, w_2)$$

$$\nabla L(w_1, w_2) = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}]$$

1. Artificial Neural Network

Part 3: Gradient Descent Method

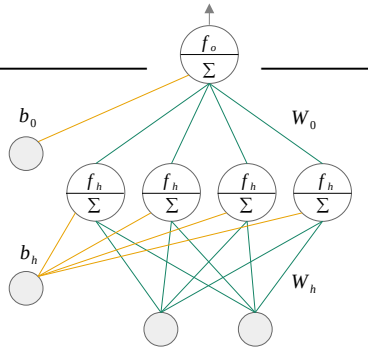
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

www.youtube.com/@meanxai

MX-AI

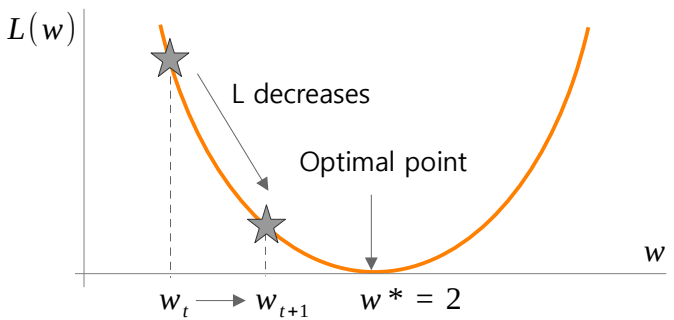
Gradient Descent

- The parameters of this ANN, Wh, bh, Wo, and bo, are determined through the training process. Initially, parameters are initialized randomly. The parameters are then iteratively updated in the direction that the loss function becomes smaller. Gradient descent is a widely used method for updating parameters.
- As the parameters are updated, the loss function value becomes smaller, so the predicted value gets closer to the actual target value.



ex: $\min_w L(w) = \min_w (w-2)^2$ $\quad \frac{\partial L(w)}{\partial w} = 2w-4$

$$w_{t+1} = w_t - \alpha \frac{\partial L(w)}{\partial w} \quad \leftarrow \text{Gradient Descent}$$



$$\left.\frac{\partial L(w)}{\partial w}\right|_{w_t} = -3$$

$w_t = 0.5$ $\quad w_{t+1} = 0.5+0.25\times 3 = 1.25$

$L_t(w) = 2.25$ $\quad L_{t+1}(w) = 0.56$ $\leftarrow$ L decreased.

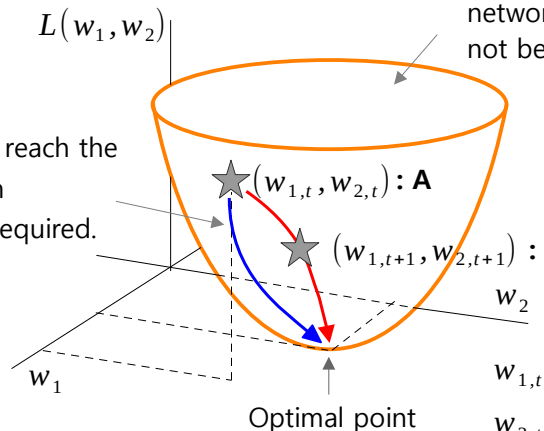
w gets closer to the optimal point.

$$w_{i,t+1} = w_{i,t} - \alpha \nabla_{w_i} L(w_1, w_2)$$

$$\nabla L(w_1, w_2) = [\frac{\partial L}{\partial w_1}, \frac{\partial L}{\partial w_2}]$$

the actual shape of L is much more complex than this, due to the many variables and nonlinear activation functions in the neural network. The actual shape of L will not be convex.

there are many paths to reach the optimal point. So, a path optimization process is required.



$w_{1,t+1} = 3-0.25\times 2 = 2.5$

$w_{2,t+1} = 2+0.25\times 2 = 2.5$

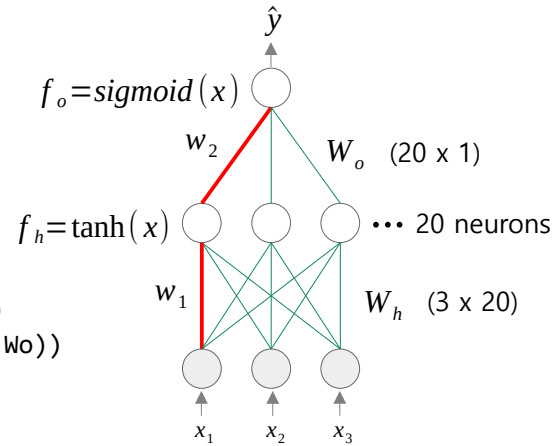
$L(w_1, w_2) = (w_1-2)^2 + (w_2-3)^2$ $\quad \nabla L(w_1, w_2)|_A = (2,-2)$

$\nabla L(w_1, w_2) = [2w_1-4,\ 2w_2-6]$ $\quad A:(3,2),\ B:(2.5,2.5)$

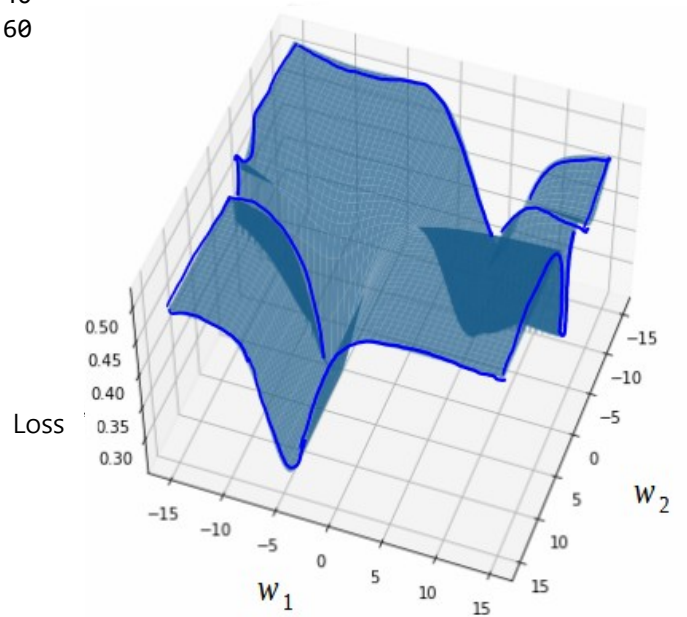
MX-AI

■ Loss surface and Local minimum problem

```
# [MXDL-1-03] 1.loss_surface.py
import matplotlib.pyplot as plt
import numpy as np

x = np.random.rand(50, 3)
y = (np.random.rand(50, 1) > 0.5) * 1.0

def sigmoid(x): return 1. / (1. + np.exp(-x))
def mse(y, y_hat): return np.mean((y - y_hat) ** 2)

# weights, no biases
Wh = np.random.rand(3, 20)
Wo = np.random.rand(20, 1)

# MSE loss function
def loss(w1, w2):
    Wh[0,0] = w1
    Wo[0,0] = w2
    h = np.tanh(np.dot(x, Wh))
    y_hat = sigmoid(np.dot(h, Wo))

    return mse(y, y_hat)

w1, w2 = np.meshgrid(np.arange(-15, 15, .1),
                     np.arange(-15, 15, .1))
zs = np.array([loss(a, b) for [a, b] in zip(np.ravel(w1),
                np.ravel(w2))])
z = zs.reshape(w1.shape)
```




```
fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(w1, w2, z, alpha=0.7)

ax.set_xlabel('w1')
ax.set_ylabel('w2')
ax.set_zlabel('loss')
ax.azim = 40
ax.elev = 60
plt.show()
```



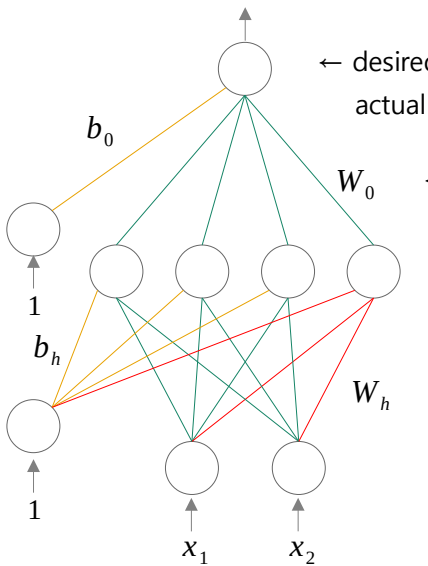
MX-AI

Finding Gradients

- Automatic differentiation allows us to obtain the accurate gradients of the loss with respect to each parameter.
- We can use these gradients to adjust the weights and biases in the output layer using the Gradient Descent method.
- The error in the output layer can be propagated to the hidden layers through the error backpropagation algorithm. Using the propagated error, we can obtain the gradients with respect to the parameters of the hidden layer. We can then use these gradients to adjust the weights and biases of the hidden layer using gradient descent method.

Error Backpropagation and Automatic differentiation

$$\hat{y} = f_o(f_h(X \cdot W_h + b_h) \cdot W_o + b_o) \qquad error = y - \hat{y} \qquad L(w,b) = \frac{1}{2}(y - \hat{y})^2$$



← desired output = y
actual output = $\hat{y}$
} We can define the error and L.

← Update these parameters by Gradient Descent

← We cannot directly define the error or loss because we do not know what the desired output of this neuron is. So how do we adjust Wh?

↓

The error from the output layer can be propagated backward to the hidden layers. This is the **Error Backpropagation** algorithm. We'll look at this in more detail later.

Numerical differentiation

Neural network can be trained roughly using numerical differentiation, without using error backpropagation or automatic differentiation. Instead of finding the exact gradient, we find an approximate gradient and use gradient descent to adjust the parameters.

- Forward difference approximation

$$\frac{\partial L}{\partial w_1} \approx \frac{L(w_1+h, w_2, \ldots, b) - L(w_1, w_2, \ldots, b)}{h}$$

- Center difference approximation

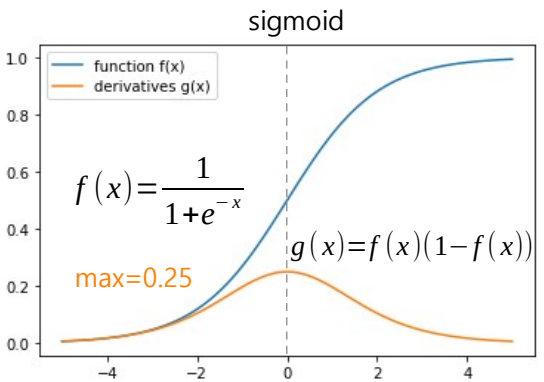
$$\frac{\partial L}{\partial w_1} \approx \frac{L(w_1+h, w_2, \ldots, b) - L(w_1-h, w_2, \ldots, b)}{2h}$$

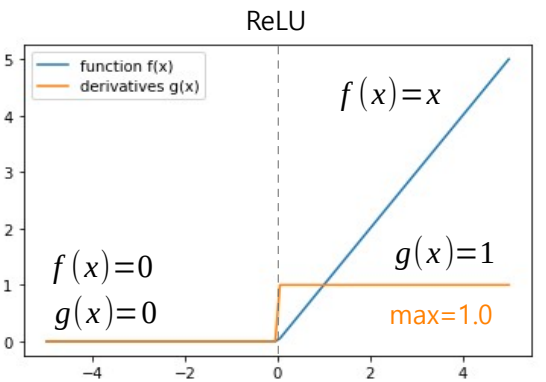
* In this series we will use this method, and in later series we will use automatic differentiation and error backpropagation algorithm.

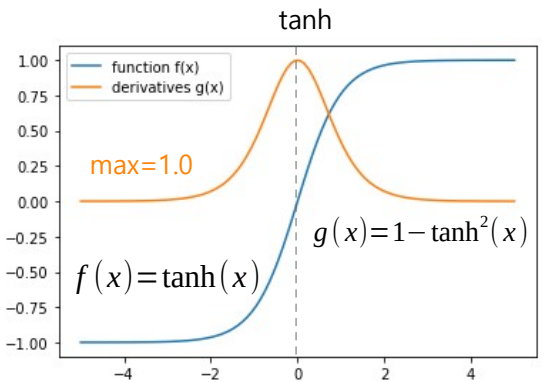
MX-AI

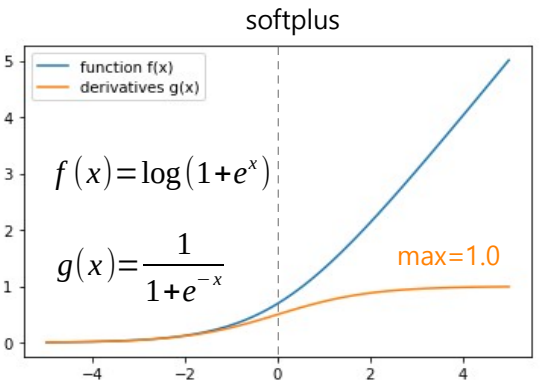
- Finding approximate gradients of activation functions using numerical differentiation.

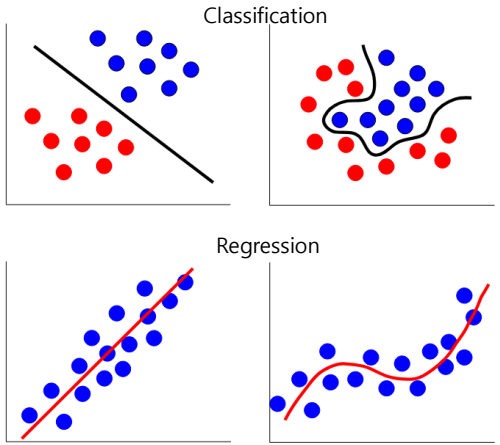
```
# [MXDL-1-03] 2.activation_diff.py
import numpy as np
import matplotlib.pyplot as plt

# activation functions
def sigmoid(x): return 1. / (1. + np.exp(-x))
def relu(x): return np.maximum(0, x)
def tanh(x): return np.tanh(x)
def softplus(x): return np.log(1 + np.exp(x))

# Numerical Differentiation
def num_differentiation(f, x, h):
    return (f(x+h) - f(x-h)) / (2. * h)

x = np.linspace(-5, 5, 100)
h = 1e-08
f = sigmoid
# f = relu
# f = tanh
# f = softplus
fx = f(x)
gx = num_differentiation(f, x, h)

# Visualization
plt.plot(x, fx, label='function f(x)')
plt.plot(x, gx, label='derivatives g(x)')
plt.legend()
plt.show()
```

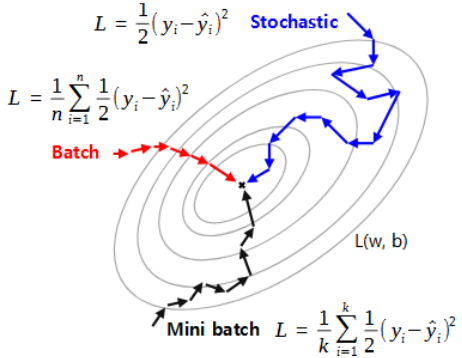












1. Artificial Neural Network

Part 4: Linearity and Non-Linearity

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

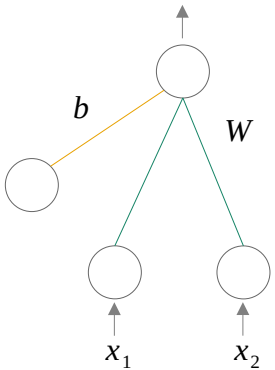
www.youtube.com/@meanxai

MX-AI

▪ Linearity vs Non-linearity

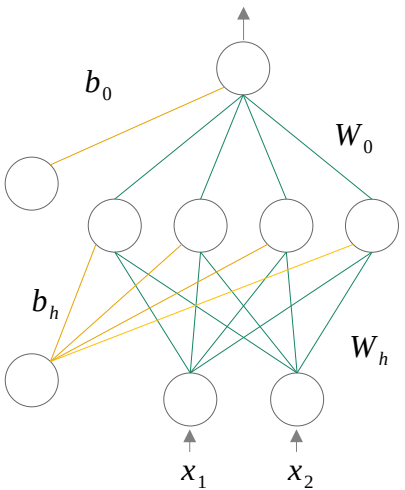
Single layer Perceptron (SLP)

$$\hat{y}=f_o(X\cdot W+b)$$



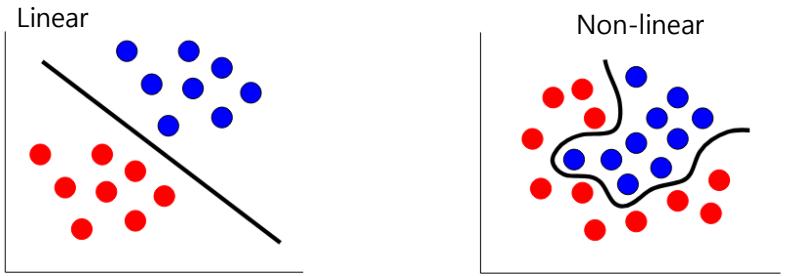
Multi-layer Perceptron (MLP)

$$\hat{y}=f_o(f_h(X\cdot W_h+b_h)\cdot W_o+b_o)$$

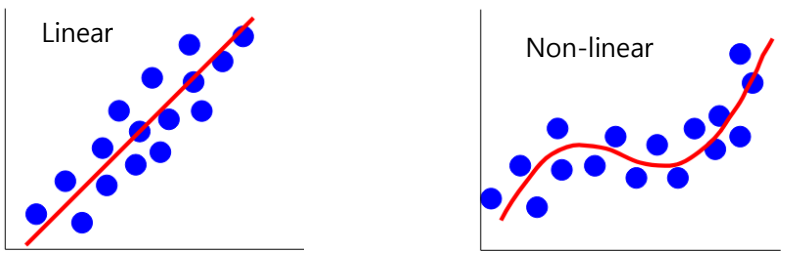


▪ SLP can only learn linear functions.

Classification



Regression



$\hat{y}=f_o(f_h(X\cdot W_h+b_h)\cdot W_o+b_o)$ ← if $f_h(x)=x$, linear function

$\hat{y}=f_o((X\cdot W_h+b_h)\cdot W_o+b_o)$

$\hat{y}=f_o(X\cdot W_h\cdot W_o+b_h\cdot W_o+b_o)$ ← $W=W_h\cdot W_o,\quad b=b_h\cdot W_o+b_o$

$\hat{y}=f_o(X\cdot W+b)$ □ ← It becomes single layer perceptron.

proof ←

▪ MLP can learn not only linear functions but also non-linear functions.

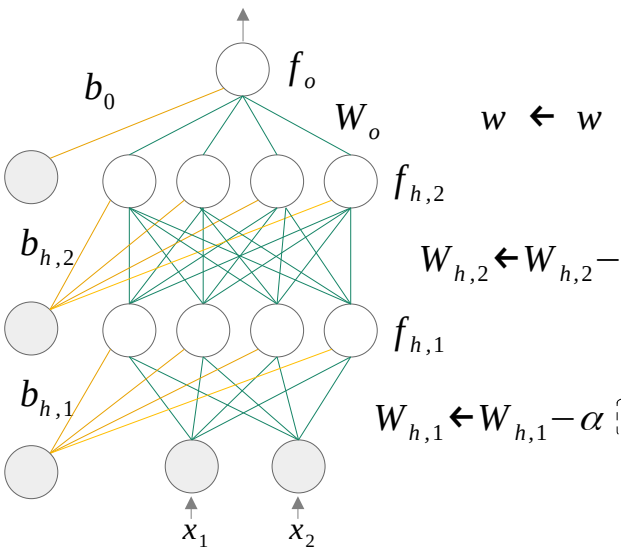
▪ However, even with a multi-layer perceptron, nonlinear problems cannot be solved if a linear activation function is used in the hidden layer. This is because multilayer neural networks with linear activation functions behave like single-layer neural networks.

MX-AI

▪ Activation function and Vanishing Gradient

- If hidden layers use nonlinear activation functions, such as sigmoid or hyperbolic tangent, the gradients of loss function may not propagate properly to lower layers. This is called the **Vanishing Gradient** problem. The maximum value of the derivative of the sigmoid function is 0.25. During backpropagation, the gradient is repeatedly multiplied by values less than 0.25, which can cause it to decrease exponentially or vanish.
- To alleviate this problem, ReLU is widely used in hidden layers. ReLU is a non-linear function that combines two linear functions. The gradient of ReLU is either 0 or 1. If it is 1, then the gradient propagates well to the lower layer. If it is 0, the gradient will not propagate. However, in this case, the output value of some neurons in the hidden layer is 0, causing some of the dropout and normalization effects that we will look at later.
- In the worst case, if the output of all neurons in the hidden layer becomes 0, the gradient cannot propagate to lower layers. This is called the **dying ReLU** problem. This problem may appear if the learning rate, alpha is large or the bias is large and negative. This is because they make the weights negative and the output of ReLU 0. To alleviate this problem, you can try lowering the learning rate and preventing the bias from becoming large. You can also try using the Leaky ReLU or softplus activation function.

$$\hat{y}=f_o(f_{h,2}(f_{h,1}(X\cdot W_{h,1}+b_{h,1})\cdot W_{h,2}+b_{h,2})\cdot W_o+b_o)$$

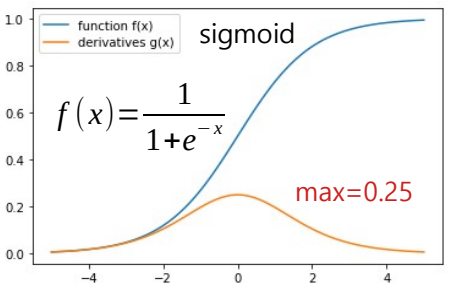


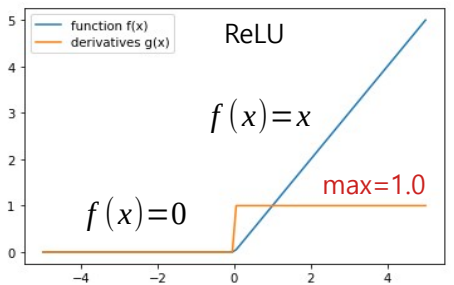
gradient of L

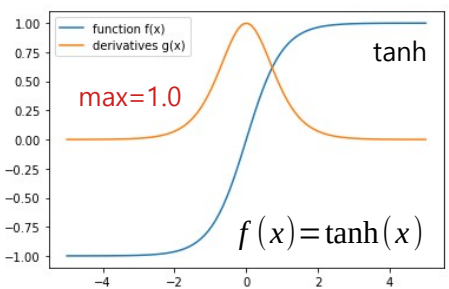
$$w \leftarrow w - \alpha \frac{\partial L}{\partial w}$$

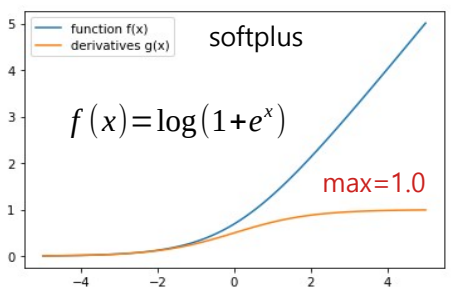
$$W_{h,2} \leftarrow W_{h,2} - \alpha \square\, f_o' f_{h,2}'$$

$$W_{h,1} \leftarrow W_{h,1} - \alpha \square\, f_o' f_{h,2}' f_{h,1}'$$









MX-AI

■ 3 Types of Gradient Descent: Stochastic, Batch, mini-Batch

[training data] output from ANN

	feature		target	predict
i	x_1	x_2	y	$\hat{y}$
1	0.16	0.01	0.54	0.49
2	-0.09	-0.03	-0.23	-0.18
3	0.06	-0.08	1.82	1.78
4	0.58	-0.1	-0.92	-0.89
5	0.46	-0.07	2.01	2.37
6	0.23	0.54	1.30	1.20
6	0.55	0.07	0.37	0.35
8	0.1	0.42	1.34	1.03
9	0.18	0.63	-1.29	-1.81
10	-0.03	-0.09	0.52	1.05
11	0.59	-0.14	1.20	1.30
12	0.32	0.06	-2.29	-1.91
⋮			⋮	⋮
n	0.75	-0.64	0.34	0.37

n is the number of data points.

1) Stochastic gradient descent (SGD)

Randomly select data points one by one, compute the loss for each data point, and update the parameters via gradient descent. If we iterate over the entire dataset once, that is, select n data points, the parameters are updated n times per iteration. Convergence may be fast because the parameters are updated frequently. However, because the loss can fluctuate greatly depending on the data selected, convergence may be unstable.

2) Batch gradient descent (GD or BGD)

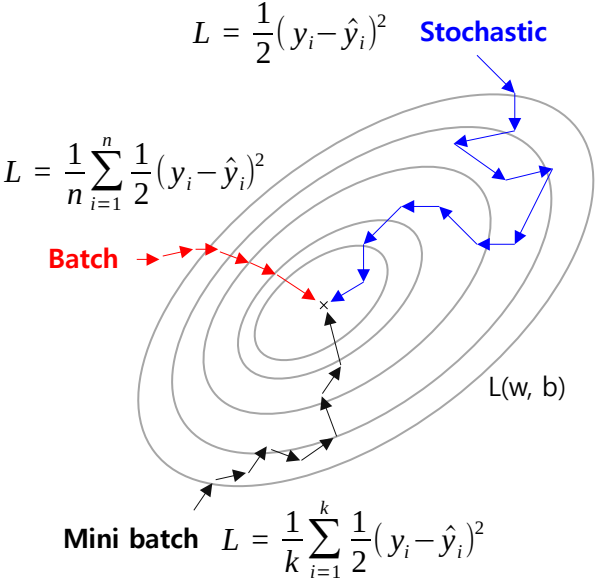
Compute the average loss for all data points at once and update the parameters once. The parameters are updated once per iteration. Convergence may be slow because parameters are updated infrequently. However, since the average loss is stable, convergence will also be stable. In addition, if the data is very large, it may not be possible to store it all in memory at once.

3) Mini-batch update

Split the shuffled dataset into multiple subsets to compute the loss for each subset and update the parameters. If the number of subsets is m, the parameters are updated m times in one iteration. Mini-batch gradient descent aims to find a balance between the speed of stochastic gradient descent and the stability of batch gradient descent. It is most widely used in deep learning.

$$L_{CE} = -\frac{1}{n}\sum_{i}^{n}\sum_{k}^{c} y_{i,k}\cdot\log(\hat{y}_{i,k})$$

$$L_{BCE} = -\frac{1}{n}\sum_{i}^{n}[\,y_i\cdot\log(\hat{y}_i)+(1-y_i)\cdot\log(1-\hat{y}_i)\,]$$



(k: batch size)

▪ Gradient Descent

$$w_{i,t+1} = w_{i,t} - \alpha\nabla_{w_i}L$$

Data normalization and shuffling

Data normalization or Standardization

- Data normalization or standardization is essential for training not only machine learning models but also artificial neural network models.
- For example, let's say feature x1 ranges from -1 to +1 and x2 ranges from -30 to +30, as shown in the data below. Then the linear output for the first data point is 0.16 w1 + 10.1 w2. If w is (1,1), it generally means that the weights of x1 and x2 are equal to 1. However, because x2 is much larger, the output value is more affected by the second term. The impact of the first term is very small. Therefore, the neural network will have difficulty learning w1.
- Methods for normalizing data include Min-max normalization and Z-score normalization or standardization, as shown below.

[Data]

	feature		class
i	x_1	x_2	y
0	0.16	10.01	0
1	-0.09	-20.03	0
2	0.06	-10.08	0
3	0.58	-13.1	2
4	0.46	-11.07	2
5	0.23	12.54	1
6	0.55	15.07	2
7	0.1	21.42	1
8	0.18	17.63	1
9	-0.03	-19.09	0
10	0.59	-23.14	2
11	0.32	29.06	2
12	-0.08	-18.03	0
13	0.27	16.52	1

- Min-Max normalization

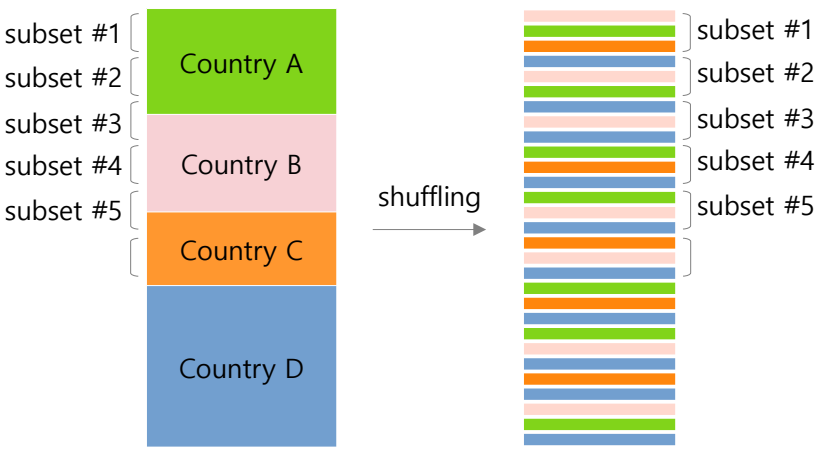
$$x'^{(i)}_k = \frac{x^{(i)}_k - min(x_k)}{max(x_k) - min(x_k)}$$

- Z-score normalization or standardization

$$x'^{(i)}_k = \frac{x^{(i)}_k - mean(x_i)}{std(x_i)}$$

Data shuffling

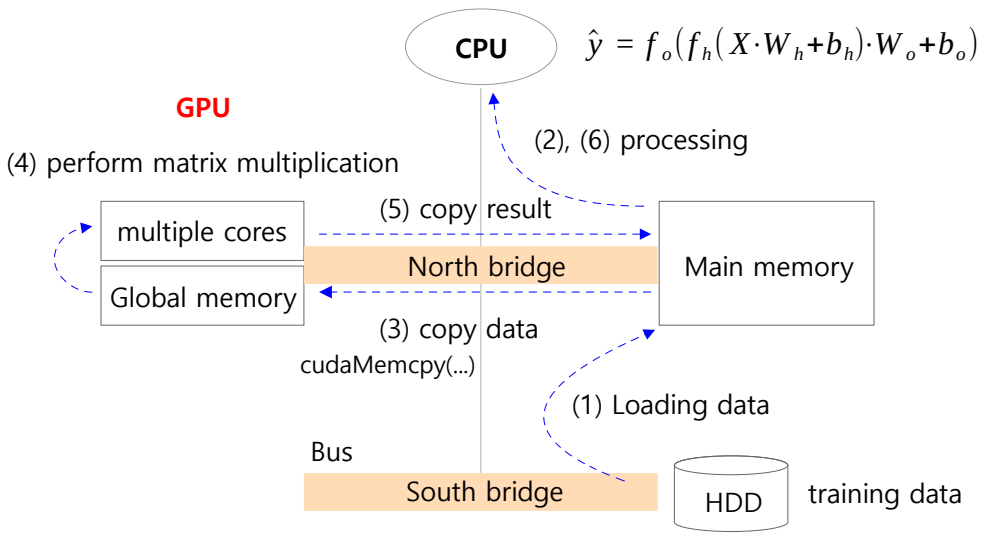
- Data shuffling is not required when using batch gradient descent, but is required when using stochastic or mini-batch method.
- Let's assume we have a dataset collected by country as shown below. We want to train on this dataset using mini-batch method. Subsets 1 and 2 both contain the data points from country A. In other words, these data points are highly correlated. If the neural network trains on this data sequentially, it will first train on the data from country A, then the data from countries B, C, and D. When training on data from country D, the network may forget the knowledge it previously trained on country A.
- To prevent this, data shuffling is required. Random shuffling of this data would look like this. Because each subset contains data points from different countries, the correlation becomes smaller. In other words, data shuffling is necessary to reduce the correlation of sequentially input data.



MX-AI

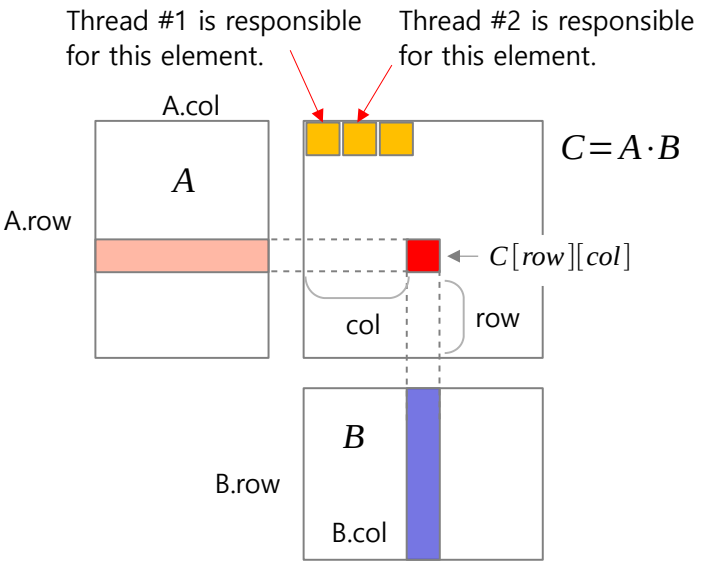
▪ Matrix multiplication and GPU

- Neural networks primarily perform large-scale matrix multiplication iteratively. The more features in the training data and the more neurons in the hidden layer, the larger the matrix w becomes, and the computational cost of multiplication increases exponentially.
- GPUs have numerous cores and can perform matrix multiplication in parallel, allowing matrix multiplication to be performed very quickly.
- GPUs are essential for training on large amounts of data for deep learning.
- Deep learning tools like TensorFlow or PyTorch natively use GPUs to perform the matrix multiplication.

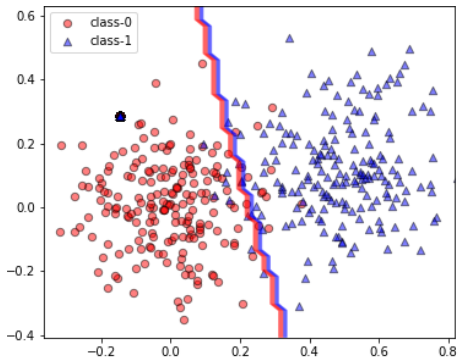


▪ Example of matrix multiplication on GPU

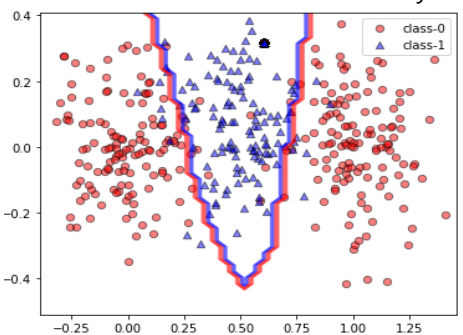
- Multiply two 1024 by 1024 matrices. Each matrix has 1 million elements.
- Sequentially multiplying these matrices on a CPU using three for-loops would require thousands of millions of multiplications.
- A GPU can use thousands of cores to create a million threads. Each thread performs a multiplication of each matrix element simultaneously. This way, the GPU can perform the multiplication of these matrices in just one step.



Linear decision boundary



Non-linear decision boundary



1. Artificial Neural Network

Part 5: Implement an ANN from scratch using numerical differentiation

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

MX-AI

■ Implement an ANN using numerical differentiation and gradient descent

- We compute the gradient of the loss function with respect to each parameter via numerical differentiation. And then we use gradient descent to update all parameters [wh, bh, wo, bo].

$$\frac{\partial L}{\partial w_1} \approx \frac{L(w_1+h, w_2, \ldots, b) - L(w_1-h, w_2, \ldots, b)}{2h}$$

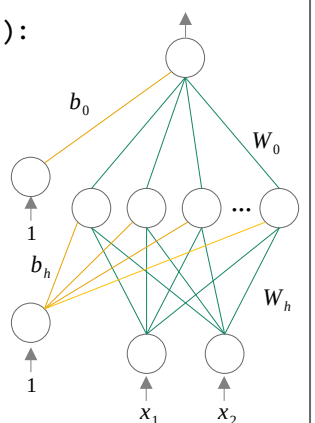
```
# [MXDL-1-05] gradient_descent.py
import numpy as np

h = 1e-4    # small value for numerical differentiation
def numerical_differentiation(x, y, f_loss, f_predict, parameters):
    p = parameters
    p_gradients = []
    for i in range(len(p)):
        # ex) p[0]= wh, p[1]= bh,
        #     p[2]= wo, p[3]= bo
        rows, cols = p[i].shape
        grad = np.zeros_like(p[i])

        # Apply numerical differentiation to all elements in p[i].
        for row in range(rows):
            for col in range(cols):
                # Measures the change in loss when the p[i][row, col]
                # element changes by h. The remaining elements are
                # fixed. This is an approximate gradient.
                p_org = p[i][row, col]      # original value of p
                p[i][row, col] = p_org + h # The element at position
                                           # (row, col) increases by h.
                y_hat = f_predict(x)       # calculate y_hat
                f1=f_loss(y, y_hat)        # The amount of change in loss
```

$$p[i] = \begin{pmatrix} w_{0,0} & w_{0,1} & \cdots \\ w_{1,0} & w_{1,1} & \cdots \\ \vdots & \vdots & \end{pmatrix}$$

$$grad = \begin{pmatrix} \frac{\partial L}{\partial w_{0,0}} & \frac{\partial L}{\partial w_{0,1}} & \cdots \\ \frac{\partial L}{\partial w_{1,0}} & \frac{\partial L}{\partial w_{1,1}} & \cdots \\ \vdots & \vdots & \end{pmatrix}$$

```
                p[i][row, col] = p_org - h # decreases by h.
                y_hat = f_predict(x)       # calculate y_hat
                f2 = f_loss(y, y_hat)      # The amount of
                                           # change in loss.
                p[i][row, col] = p_org     # Restore p back to
                                           # its original value.
                # the gradient at position (row, col)
                grad[row, col] = (f1 - f2) / (2. * h)
        p_gradients.append(grad)# gradients for all parameters
    return p_gradients

# Perform Gradient Descent
def gradient_descent(x, y, alpha, f_loss,
                     f_predict, parameters):
    p = parameters
    grad = numerical_differentiation(x, y,
                     f_loss,
                     f_predict,
                     parameters)

    for i in range(len(p)):
        p[i] = p[i] - alpha * grad[i]
```

$$w_{t+1} = w_t - \alpha \frac{\partial L(w)}{\partial w}$$

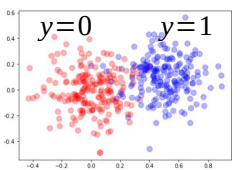


MX-AI

■ Implement an ANN using numerical differentiation: single-layered Perceptron

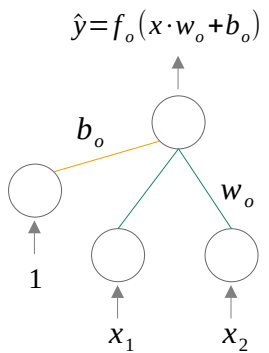
- Create an single-layered ANN model and perform binary classification using numerical differentiation and gradient descent.
- To calculate the gradient accurately, you must use automatic differentiation. However, here we use numerical differentiation to approximate the gradient.
- Gradient descent with automatic differentiation will be discussed in detail in the Backpropagation part.

```python
# [MXDL-1-05] 3.binary_single_layer.py
import numpy as np
from sklearn.datasets import make_blobs
from gradient_descent import gradient_descent
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# Generate a dataset for binary classification
x, y = make_blobs(n_samples=400, n_features=2,
                  centers=[[0., 0.], [0.5, 0.1]],
                  cluster_std=0.15, center_box=(-1., 1.))
```




```python
y = y.reshape(-1, 1)

# See the data visually.
plt.figure(figsize=(7,5))
color = [['red', 'blue'][a] for a in y.reshape(-1,)]
plt.scatter(x[:, 0], x[:, 1], s=100, c=color, alpha=0.3)
plt.show()

# Generate the training and test data
x_train, x_test, y_train, y_test = train_test_split(x, y)
```

```python
# Create an single-layered ANN model
n_input = x.shape[1]  # the number of input neurons
n_output = 1          # the number of output neurons
alpha = 0.1           # learning rate

# initialize the parameters randomly.
wo = np.random.normal(size=(n_input, n_output)) # weights of output layer
bo = np.zeros(shape=(1, n_output))              # bias of output layer
parameters = [wo, bo]                           # parameter list

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

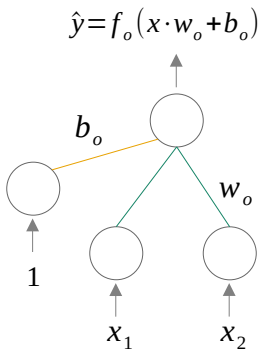



```python
# loss function: binary cross entropy
def loss(y, y_hat):
    return -np.mean(y * np.log(y_hat) + (1. - y) * np.log(1. - y_hat))
```

MX-AI

- Implement an ANN using numerical differentiation: single-layered Perceptron

```
# Output from the ANN model: prediction process
def predict(x, proba=True):
    p = parameters
    o_output = sigmoid(np.dot(x, p[0]) + p[1])  # output from
                                                # output layer
    if proba:
        return o_output              # return probability
    else:
        return (o_output > 0.5) * 1  # return class

# Perform training and track the loss history.
def train(x, y, x_val, y_val, epochs, batch_size):
    ht_loss = []  # loss history of training data
    hv_loss = []  # loss history of test data
    for epoch in range(epochs):
        # measure the losses during training
        # loss for training data
        ht_loss.append(loss(y, predict(x)))

        # loss for validation data
        hv_loss.append(loss(y_val, predict(x_val)))

        # Perform training using mini-batch gradient descent
        for batch in range(int(x.shape[0] / batch_size)):
            idx = np.random.choice(x.shape[0], batch_size)
            gradient_descent(x[idx], y[idx], alpha, loss, predict,
                             parameters)
```




```
        if epoch % 10 == 0:
            print("{}: train_loss={:.4f}, val_loss={:.4f}".\
                  format(epoch, ht_loss[-1], hv_loss[-1]))

    return ht_loss, hv_loss

# Perform training
train_loss, val_loss = train(x_train, y_train, x_test, y_test,
                             epochs=200, batch_size=50)

# Visually check the loss history.
plt.plot(train_loss, c='blue', label='train loss')
plt.plot(val_loss, c='red', label='test loss')
plt.legend()
plt.show()

# Check the accuracy.
y_pred = predict(x_train, proba=False)
acc = (y_pred == y_train).mean()
print("Accuracy of training data = {:4f}".format(acc))

y_pred = predict(x_test, proba=False)
acc = (y_pred == y_test).mean()
print("Accuracy of test data = {:4f}".format(acc))
```

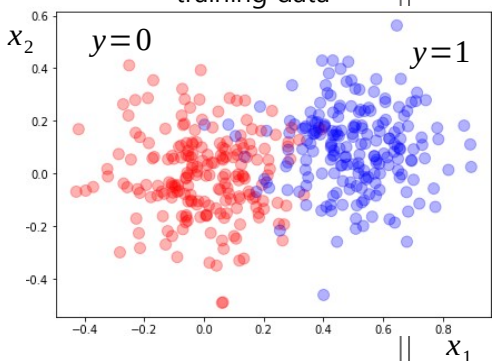
MX-AI

- Implement an ANN using numerical differentiation: single-layered Perceptron

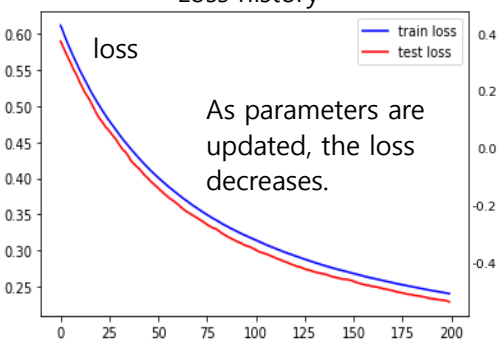
```
# Visualize the linear decision boundary
# reference : https://psrivasin.medium.com/
# plotting-decision-boundaries-using-numpy-and-matplotlib-
# f5613d8acd19
x_min, x_max = x_test[:, 0].min() - 0.1, x_test[:,0].max() + 0.1
y_min, y_max = x_test[:, 1].min() - 0.1, x_test[:, 1].max() + 0.1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 50),
                     np.linspace(y_min, y_max, 50))
x_in = np.c_[xx.ravel(), yy.ravel()]

# Predict the classes of the data points in the x_in variable.
y_pred = predict(x_in, proba=False).astype('int8')
y_pred = y_pred.reshape(xx.shape)

plt.figure(figsize=(5, 5))
m = ['o', '^']
color = ['red', 'blue']
for i in [0, 1]:
    idx = np.where(y == i)
    plt.scatter(x[idx, 0], x[idx, 1],
                c = color[i],
                marker = m[i],
                s = 40,
                edgecolor = 'black',
                alpha = 0.5,
                label='class-' + str(i))

plt.contour(xx, yy, y_pred, cmap=ListedColormap(color), alpha=0.5)
plt.axis('tight')
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()
```

```
  0: train_loss=0.6115, val_loss=0.5897
 10: train_loss=0.5487, val_loss=0.5318
...
170: train_loss=0.2553, val_loss=0.2446
180: train_loss=0.2498, val_loss=0.2386
190: train_loss=0.2446, val_loss=0.2335

Accuracy of training data = 0.946667
Accuracy of test data = 0.950000
```

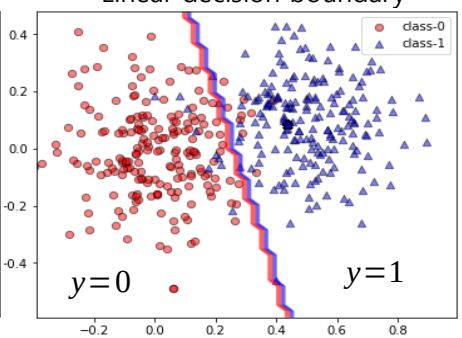
training data



Loss history



Linear decision boundary

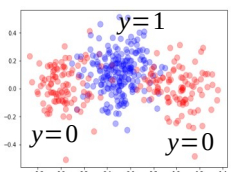


MX-AI

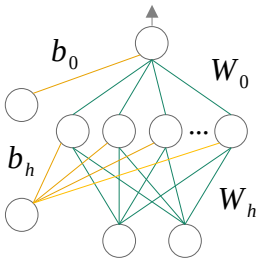
■ Implement an ANN using numerical differentiation: 2-layered Perceptron

- Create an two-layered ANN and perform binary classification using numerical differentiation and gradient descent.
- To calculate the gradient accurately, you must use automatic differentiation. However, here we use numerical differentiation to approximate the gradient.
- Gradient descent with automatic differentiation will be discussed in detail in the Backpropagation part.

```
# [MXDL-1-05] 4.binary_two_layers.py
import numpy as np
from sklearn.datasets import make_blobs
from gradient_descent import gradient_descent
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# Generate a dataset
x, y = make_blobs(n_samples=400, n_features=2,
                  centers=[[0., 0.], [0.5, 0.1], [1., 0.]],
                  cluster_std=0.15, center_box=(-1., 1.))

y[y == 2] = 0 # y = [0, 1, 2] → [0, 1]
y = y.reshape(-1, 1)

# See the data visually.
plt.figure(figsize=(7,5))
color = [['red', 'blue'][a] for a in y.reshape(-1,)]
plt.scatter(x[:, 0], x[:, 1], s=100, c=color, alpha=0.3)
plt.show()

# Generate the training and test data
x_train, x_test, y_train, y_test = train_test_split(x, y)
```




```
# Create an two-layered ANN model.
n_input = x.shape[1]  # the number of input neurons
n_output = 1          # the number of output neurons
n_hidden = 16         # the number of hidden neurons
alpha = 0.1           # learning rate

# initialize the parameters randomly.
wh = np.random.normal(size=(n_input, n_hidden))  # weights of hidden layer
bh = np.zeros(shape=(1, n_hidden))               # biases of hidden layer
wo = np.random.normal(size=(n_hidden, n_output)) # weights of output layer
bo = np.zeros(shape=(1, n_output))               # bias of output layer
parameters = [wh, bh, wo, bo]                    # parameter list

# activation functions
def sigmoid(x): return 1. / (1. + np.exp(-x))
def relu(x):    return np.maximum(0, x)

# loss function: binary cross entropy
def loss(y, y_hat):
    return -np.mean(y * np.log(y_hat) + (1. - y) * np.log(1. - y_hat))
```

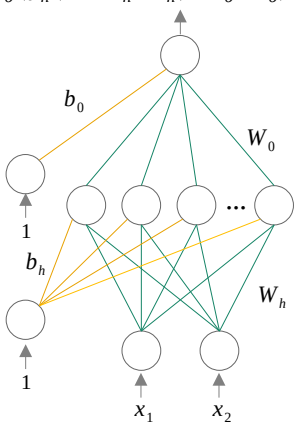


MX-AI

■ Implement an ANN using numerical differentiation: 2-layered Perceptron

```
# Output from the ANN model: prediction process
def predict(x, proba=True):
    p = parameters
    h_out = relu(np.dot(x, p[0]) + p[1])        # output from hidden layer
    o_out = sigmoid(np.dot(h_out, p[2]) + p[3]) # output from output layer

    if proba:
        return o_out              # return probability
    else:
        return (o_outt > 0.5) * 1 # return class
```

$$\hat{y}=f_o(f_h(X\cdot W_h+b_h)\cdot W_o+b_o)$$




```
# Perform training and track the loss history.
def train(x, y, x_val, y_val, epochs, batch_size):
    ht_loss = []  # loss history of training data
    hv_loss = []  # loss history of test data
    for epoch in range(epochs):
        # measure the losses during training
        # loss for training data
        ht_loss.append(loss(y, predict(x)))

        # loss for validation data
        hv_loss.append(loss(y_val, predict(x_val)))

        # Perform training using mini-batch gradient descent
        for batch in range(int(x.shape[0] / batch_size)):
            idx = np.random.choice(x.shape[0], batch_size)
            gradient_descent(x[idx], y[idx], alpha, loss, predict,
                             parameters)
        if epoch % 10 == 0:
            print("{}: train_loss={:.4f}, val_loss={:.4f}".\
                  format(epoch, ht_loss[-1], hv_loss[-1]))

    return ht_loss, hv_loss

# Perform training
train_loss, val_loss = train(x_train, y_train, x_test, y_test,
                             epochs=200, batch_size=50)

# Visually check the loss history.
plt.plot(train_loss, c='blue', label='train loss')
plt.plot(val_loss, c='red', label='test loss')
plt.legend()
plt.show()

# Check the accuracy.
y_pred = predict(x_train, proba=False)
acc = (y_pred == y_train).mean()
print("Accuracy of training data = {:4f}".format(acc))

y_pred = predict(x_test, proba=False)
acc = (y_pred == y_test).mean()
print("Accuracy of test data = {:4f}".format(acc))
```

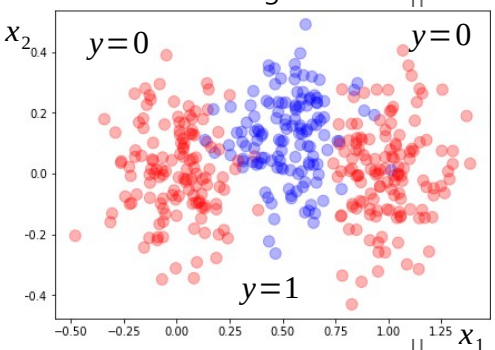
MX-AI

■ Implement an ANN using numerical differentiation: 2-layered Perceptron

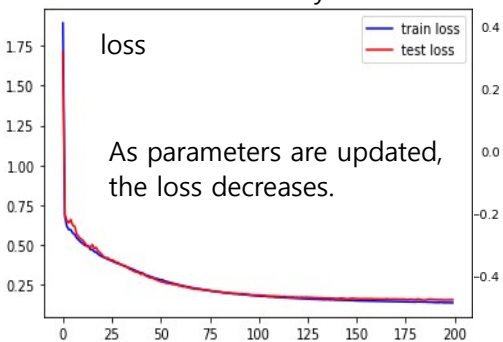
```
# Visualize the non-linear decision boundary
# reference : https://psrivasin.medium.com/
# plotting-decision-boundaries-using-numpy-and-matplotlib-
# f5613d8acd19
x_min, x_max = x_test[:, 0].min() - 0.1, x_test[:,0].max() + 0.1
y_min, y_max = x_test[:, 1].min() - 0.1, x_test[:, 1].max() + 0.1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 50),
                     np.linspace(y_min, y_max, 50))
x_in = np.c_[xx.ravel(), yy.ravel()]

# Predict the classes of the data points in the x_in variable.
y_pred = predict(x_in, proba=False).astype('int8')
y_pred = y_pred.reshape(xx.shape)

plt.figure(figsize=(5, 5))
m = ['o', '^']
color = ['red', 'blue']
for i in [0, 1]:
    idx = np.where(y == i)
    plt.scatter(x[idx, 0], x[idx, 1],
                c = color[i],
                marker = m[i],
                s = 40,
                edgecolor = 'black',
                alpha = 0.5,
                label='class-' + str(i))

plt.contour(xx, yy, y_pred, cmap=ListedColormap(color), alpha=0.5)
plt.axis('tight')
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()
```

```
  0: train_loss=1.8925, val_loss=1.7120
 10: train_loss=0.5088, val_loss=0.5304
...
170: train_loss=0.1428, val_loss=0.1578
180: train_loss=0.1402, val_loss=0.1537
190: train_loss=0.1379, val_loss=0.1557

Accuracy of training data = 0.966667
Accuracy of test data = 0.910000
```

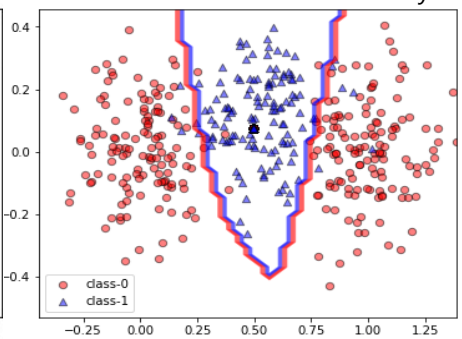
training data



Loss history



Non-linear decision boundary

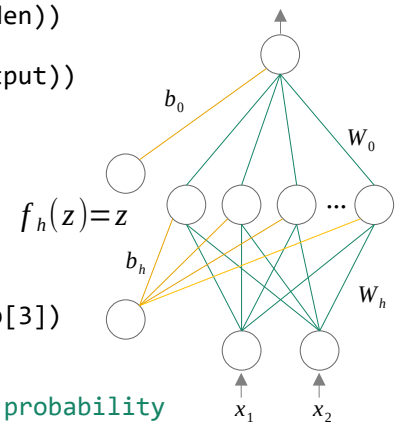


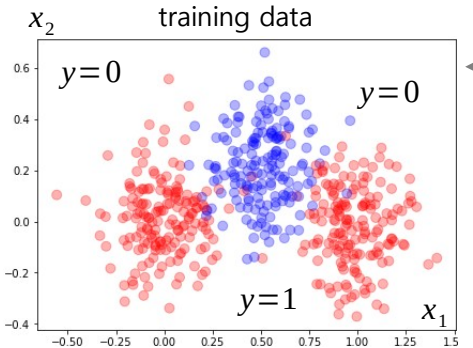
MX-AI

■ 2-layered Perceptron with linear activation function: Nonlinear problems cannot be solved.

- Even with a multi-layer perceptron, nonlinear problems cannot be solved if a linear activation function is used in the hidden layers.
- Let's check this with an experiment.

```python
# [MXDL-1-05] 5.linear_activation.py

# Create an two-layered ANN model.
n_input = x.shape[1]  # the number of input neurons
n_output = 1          # the number of output neurons
n_hidden = 16         # the number of hidden neurons
alpha = 0.05          # learning rate
h = 1e-4              # constant for numerical differentiation

# initialize the parameters randomly.
wh = np.random.normal(size=(n_input, n_hidden))
bh = np.zeros(shape=(1, n_hidden))
wo = np.random.normal(size=(n_hidden, n_output))
bo = np.zeros(shape=(1, n_output))
parameters = [wh, bh, wo, bo]

# Output from the ANN model
def predict(x, proba=True):
    p = parameters
    h_out = np.dot(x, p[0]) + p[1]
    o_out = sigmoid(np.dot(h_out, p[2]) + p[3])

    if proba:
        return o_out              # return probability
    else:
        return (o_out > 0.5) * 1  # return class

# The rest is the same as the previous code.
```





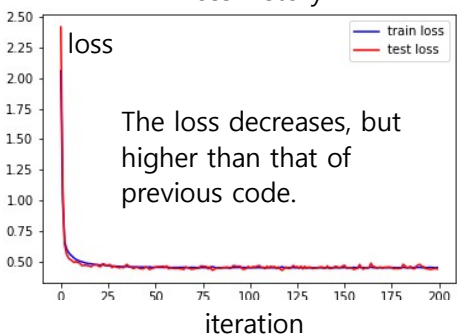
← Linear separation is not possible.

```
Accuracy of training data = 0.79
Accuracy of test data = 0.79
```

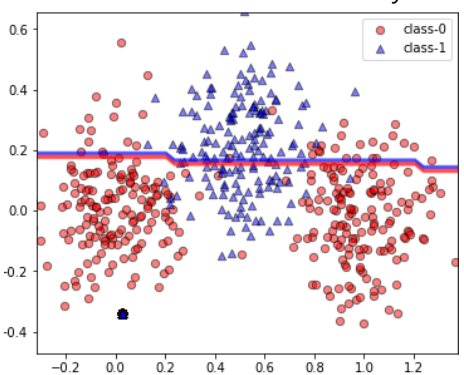
The accuracy is low.

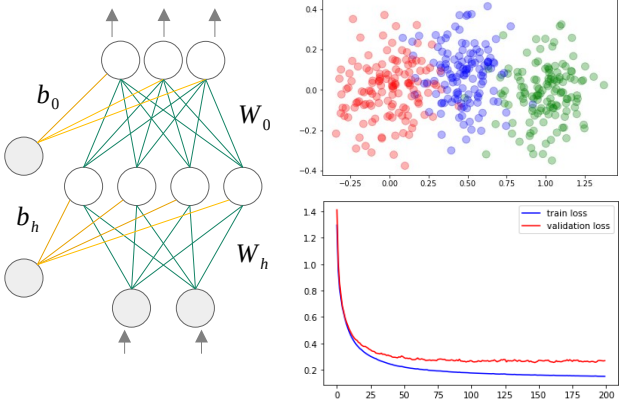
```
170: train_loss=0.4502, val_loss=0.4455
180: train_loss=0.4493, val_loss=0.4448
190: train_loss=0.4523, val_loss=0.4753
```

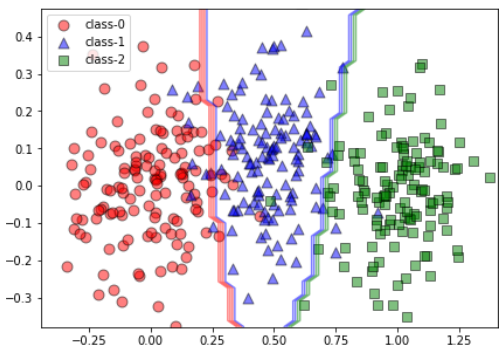
Loss history



Linear decision boundary







1. Artificial Neural Network

Part 6: Multiclass Classification

This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

MX-AI

Multiclass Classification

- When the number of classes is more than 2, such as y = 0, 1, 2, it is called multiclass classification.
- Convert the target class y to one-hot encoding and set the number of neurons in the output layer of a neural network to the number of classes y.
- Convert multiple outputs from a neural network into a probability distribution using the softmax function.
- The softmax function converts a vector of outputs from the ANN, which are C real values, into a probability distribution of C classes.
- The class of the input data point is predicted as the index number of the neuron with the largest output value. You can easily find this by taking argmax over the output vector of the neurons.

[training data]

output from ANN

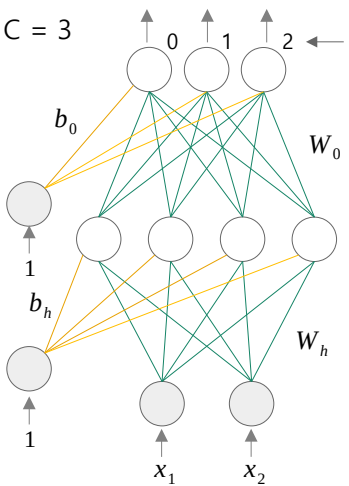
	feature		target	one hot encode	predict			argmax
i	x_1	x_2	y	y_{ohe}	$\hat{y}$			$\hat{y}$
1	0.16	0.01	0	1 0 0	0.7	0.1	0.2	0
2	-0.09	-0.03	1	0 1 0	0.2	0.6	0.2	1
3	0.06	-0.08	2	0 0 1	0.1	0.1	0.8	2
4	0.58	-0.1	1	0 1 0	0.1	0.7	0.2	1
5	0.46	-0.07	0	1 0 0	0.9	0.0	0.1	0
6	0.23	0.54	1	0 1 0	0.1	0.6	0.3	1
6	0.55	0.07	2	0 0 1	0.2	0.1	0.7	2
8	0.1	0.42	0	1 0 0	0.4	0.3	0.3	0
9	0.18	0.63	1	0 1 0	0.1	0.8	0.1	1
10	-0.03	-0.09	2	0 0 1	0.1	0.5	0.4	1
11	0.59	-0.14	0	1 0 0	0.9	0.1	0.0	0
12	0.32	0.06	1	0 1 0	0.1	0.7	0.2	1
	⋮		⋮	⋮	⋮			⋮
N	0.75	-0.64						

* N is the number of data points.
C is the number of classes.

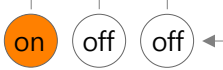
$$\hat{y} = softmax\left(f_h\left(X \cdot W_h + b_h\right) \cdot W_o + b_o\right)$$: output from ANN

$y = 0 \rightarrow 1\ 0\ 0$
$y = 1 \rightarrow 0\ 1\ 0$
$y = 2 \rightarrow 0\ 0\ 1$
} desired output (one-hot encoded)

C = 3



class → [0 1 2]
$\hat{y}$ → [0.7 0.1 0.2] ← If the first neuron has the largest activation level, the class of the input data point is predicted to be 0.
$y = 0$ → [1 0 0]
target



output neurons ← The first neuron is activated and the remaining neurons are inhibited.

MX-AI

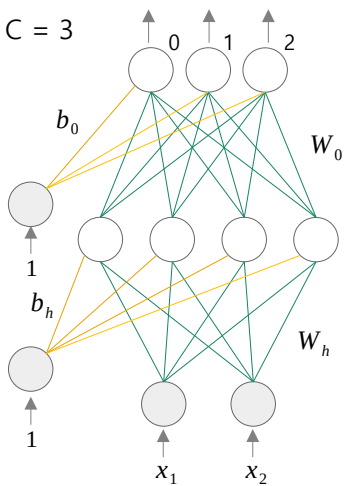
■ Multiclass Classification: Loss and Activation function

$$\hat{y} = softmax\left(f_o\left(f_h\left(X \cdot W_h + b_h\right) \cdot W_o + b_o\right)\right)$$

$\hat{y}$ → [0.7 0.1 0.2]

y = 0 → 1 0 0

C = 3



▪ Loss function

Cross Entropy (CE):

$$L(w,b) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{C} y_{i,k}\log\left(\hat{y}_{i,k}\right)$$

(C is the number of classes.)

* example: y = [1 0 0], y_hat = [0.7, 0.1, 0.2]

$$L(w,b) = -[1\times\log(0.7) + \quad \#\ C = 0$$
$$0\times\log(0.1) + \quad \#\ C = 1$$
$$0\times\log(0.2)] \quad \#\ C = 2$$

* if C = 2, it becomes binary cross entropy loss.

C = 1 C = 0

$$L(w,b) = -\frac{1}{N}\sum_{i=1}^{N}[\,y_i\log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

▪ Softmax

$$\hat{y}_i = \frac{\exp(z_i)}{\sum_{k=1}^{n}\exp(z_k)} \leftarrow$$

- z is the output vector from an ANN.
  ex: z = [0.3, 0.9. 07],
  y hat = [0.23, 0.42, 0.36]
- y hat is a probability distribution.

- This equation has the potential to overflow if z is large.
- You can prevent overflow by modifying it as follows:

$$\hat{y}_i = \frac{A\cdot\exp(z_i)}{A\cdot\sum_{k=1}^{n}\exp(z_k)}$$ (A: an arbitrary constant)

$$\hat{y}_i = \frac{\exp(z_i + \log A)}{\sum_{k=1}^{n}\exp(z_k + \log A)}$$

$$\hat{y}_i = \frac{\exp(z_i - A')}{\sum_{k=1}^{n}\exp(z_k - A')} \qquad \left(\sum_{i=1}^{n}\hat{y}_i = 1.0\right)$$

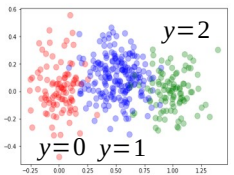
$A' = max(z_i) \leftarrow$ You can safely prevent overflow by defining A as max(x).

MX-AI

■ Implement multiclass classification using numerical differentiation: 2-layered Perceptron

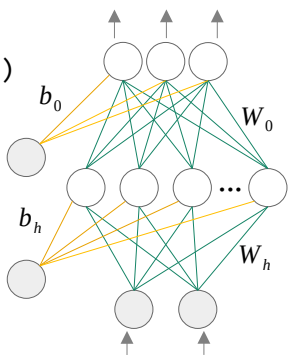
- Create a two-layered ANN model and perform multiclass classification using numerical differentiation and gradient descent.
- To calculate the gradient accurately, you must use automatic differentiation. However, here we use numerical differentiation to approximate the gradient.
- Gradient descent with automatic differentiation will be discussed in detail in the Backpropagation part.

```
# [MXDL-1-06] 6.multiclass_classification.py
import numpy as np
from sklearn.datasets import make_blobs
from gradient_descent import gradient_descent
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap

# Generate a dataset for multiclass classification
x, y = make_blobs(n_samples=400, n_features=2,
                  centers=[[0., 0.], [0.5, 0.1], [1., 0.]],
                  cluster_std=0.15, center_box=(-1., 1.))

n_class = np.unique(y).shape[0]  # the number of classes

# See the data visually.
plt.figure(figsize=(7,5))
color = [['red', 'blue', 'green'][a] for a in y.reshape(-1,)]
plt.scatter(x[:, 0], x[:, 1], s=100, c=color, alpha=0.3)
plt.show()

# one-hot encode class y, y = [0,1,2]
y_ohe = np.eye(n_class)[y]
```



$$\hat{y}_i = \frac{\exp(z_i - A')}{\sum_{k=1}^{n} \exp(z_k - A')}$$

```
# Generate the training and test data
x_train, x_test, y_train, y_test = train_test_split(x, y_ohe)

# Create an two-layered ANN model.
n_input = x.shape[1]     # number of input neurons
n_output = n_class       # number of output neurons
n_hidden = 16            # number of hidden neurons
alpha = 0.05             # learning rate
h = 1e-4                 # constant for numerical differentiation

# initialize the parameters randomly.
wh = np.random.normal(size=(n_input, n_hidden))
bh = np.zeros(shape=(1, n_hidden))
wo = np.random.normal(size=(n_hidden, n_output))
bo = np.zeros(shape=(1, n_output))
parameters = [wh, bh, wo, bo]

# activation functions
def softmax(x):
    A = np.max(x, axis=1, keepdims=True)
    e = np.exp(x - A)
    return e / np.sum(e, axis=1, keepdims=True)

def relu(x):
    return np.maximum(0, x)
```



MX-AI

- Implement multiclass classification using numerical differentiation: 2-layered Perceptron

```
# loss function: categorical cross entropy
def loss(y, y_hat):
    ce = -np.sum(y * np.log(y_hat), axis=1)
    return np.mean(ce)
```

$$-\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{C} y_{i,k}\log(\hat{y}_{i,k})$$

```
# Output from the ANN model: prediction process
def predict(x, proba=True):
    p = parameters
    o_hidden = relu(np.dot(x, p[0]) + p[1])
    o_output = softmax(np.dot(o_hidden, p[2]) + p[3])
```

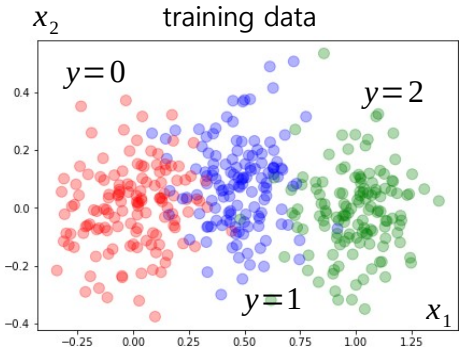
$$\hat{y} = softmax(f_h(X \cdot W_h + b_h) \cdot W_o + b_o)$$

```
    if proba:
        return o_output          # return probability distribution
    else:
        return np.argmax(o_output, axis=1)  # return class

# Perform training and track the loss history.
def train(x, y, x_val, y_val, epochs, batch_size):
    ht_loss = []     # loss history of training data
    hv_loss = []     # loss history of test data
    for epoch in range(epochs):
        # measure the losses during training
        ht_loss.append(loss(y, predict(x)))  # train loss
        hv_loss.append(loss(y_val, predict(x_val))) # validation
                                                    # loss
        # Perform training using mini-batch gradient descent
        for batch in range(int(x.shape[0] / batch_size)):
            idx = np.random.choice(x.shape[0], batch_size)
            gradient_descent(x[idx], y[idx], alpha, loss,
                             predict, parameters)
        if epoch % 10 == 0:
            print("{}: train_loss={:.4f}, val_loss={:.4f}".\
                  format(epoch, ht_loss[-1], hv_loss[-1]))
    return ht_loss, hv_loss

# Perform training
train_loss, val_loss = train(x_train, y_train, x_test, y_test,
                             epochs=200, batch_size=50)

# Visually check the loss history.
plt.plot(train_loss, c='blue', label='train loss')
plt.plot(val_loss, c='red', label='validation loss')
plt.legend()
plt.show()

# Check the accuracy.
y_pred = predict(x_train, proba=False)
acc = (y_pred == np.argmax(y_train, axis=1)).mean()
print("Accuracy of training data = {:4f}".format(acc))

y_pred = predict(x_test, proba=False)
acc = (y_pred == np.argmax(y_test, axis=1)).mean()
print("Accuracy of test data = {:4f}".format(acc))
```

MX-AI

- Implement multiclass classification using numerical differentiation: 2-layered Perceptron

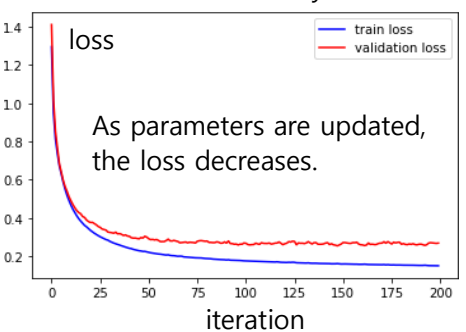
```
# Visualize the decision boundaries.
# reference :
# https://psrivasin.medium.com/
#   plotting-decision-boundaries-using-numpy-and-matplotlib-
#   f5613d8acd19
x_min, x_max = x_test[:, 0].min() - 0.1, x_test[:,0].max() + 0.1
y_min, y_max = x_test[:, 1].min() - 0.1, x_test[:, 1].max() + 0.1
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 50),
                     np.linspace(y_min, y_max, 50))
x_in = np.c_[xx.ravel(), yy.ravel()]

# Predict the classes of the data points in the x_in variable.
y_pred = predict(x_in, proba=False).astype('int8')
y_pred = y_pred.reshape(xx.shape)

plt.figure(figsize=(7, 5))
m = ['o', '^', 's']
color = ['red', 'blue', 'green']
for i in [0, 1, 2]:
    idx = np.where(y == i)
    plt.scatter(x[idx, 0], x[idx, 1],
                c = color[i],
                marker = m[i],
                s = 80,
                edgecolor = 'black',
                alpha = 0.5,
                label='class-' + str(i))
plt.contour(xx, yy, y_pred, cmap=ListedColormap(color), alpha=0.5)
plt.axis('tight')
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xlabel('x1')
plt.ylabel('x2')
plt.legend()
plt.show()
```



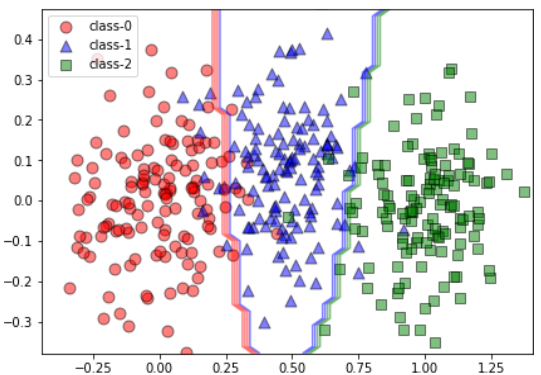

```
Accuracy of training data = 0.95
Accuracy of test data = 0.90
```

```
150: train_loss=0.1591, val_loss=0.2711
160: train_loss=0.1573, val_loss=0.2587
170: train_loss=0.1551, val_loss=0.2680
180: train_loss=0.1530, val_loss=0.2623
190: train_loss=0.1514, val_loss=0.2629
```

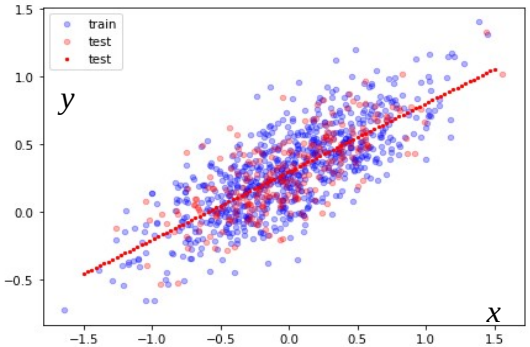
Loss history



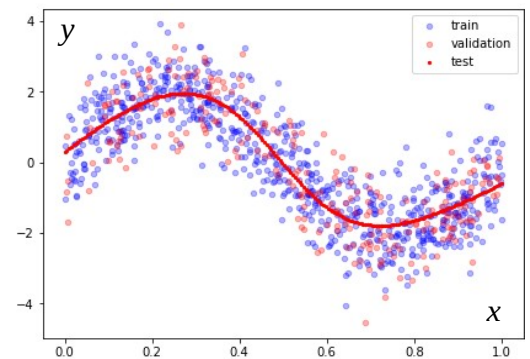
two decision boundaries



Linear Regression



Nonlinear Regression



1. Artificial Neural Network

Part 7: Linear and Non-linear Regression

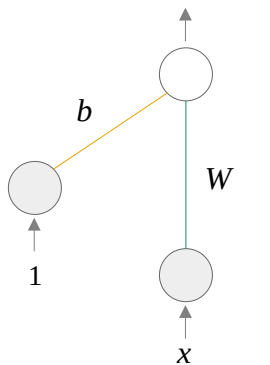
This video was produced in Korean and translated into English, and the audio was generated by AI (TTS).

Linear and Non-linear Regression

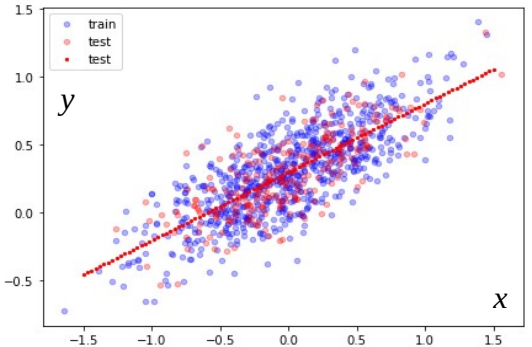
- linear regression can be implemented using a single-layer Perceptron (SLP), and non-linear regression can be implemented using a multi-layer Perceptron (MLP)
- The output layer uses a linear activation function (f(z) = z), and the hidden layer uses nonlinear activation function, such as sigmoid, tanh, ReLU.
- Mean squared error (MSE) is used as the loss function.

Linear Regression

$$\hat{y} = X \cdot W + b$$

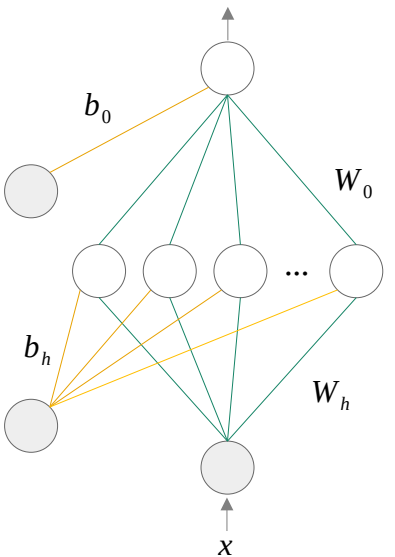


[Single-layer Perceptron]



Nonlinear Regression

$$\hat{y} = f_h(X \cdot W_h + b_h) \cdot W_o + b_o$$



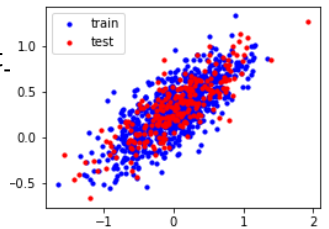
[Multi-layer Perceptron (MLP)]

MX-AI

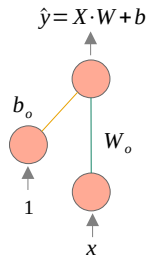
■ Implement linear regression using numerical differentiation: single-layered Perceptron

- Create a single-layered ANN model and perform the linear regression using numerical differentiation and gradient descent.
- To calculate the gradient accurately, you must use automatic differentiation. However, here we use numerical differentiation to approximate the gradient.
- Gradient descent with automatic differentiation will be discussed in detail in the backpropagation part later.

```
# [MXDL-1-07] 7.linear_regression.py
import numpy as np
from gradient_descent import gradient_descent
from sklearn.model_selection import train_test_
import matplotlib.pyplot as plt

# Generate training data set
# y = 0.5x + 0.3 + noise
x = np.random.normal(0.0, 0.5, (1000, 1))
y = 0.5 * x + 0.3 + np.random.normal(0.0, 0.2, (1000, 1))

# Generate training, validation, and test data set
x_train, x_valid, y_train, y_valid = train_test_split(x, y)
x_test = np.linspace(-1.5, 1.5, 100).reshape(-1, 1)

# See the data visually.
plt.figure(figsize=(7,5))
plt.scatter(x_train, y_train, s=20, c='blue', alpha=0.3,
            label='train')
plt.scatter(x_valid, y_valid, s=20, c='red', alpha=0.3,
            label='test')
plt.legend()
plt.show()

# Create a single-layered ANN model.
n_input = x.shape[1]       # number of input neurons
n_output = 1               # number of output neurons
```




```
alpha = 0.01  # learning rate
h = 1e-4      # constant for numerical differentiation

# initialize the parameters randomly.
wo = np.random.normal(size=(n_input, n_output))
bo = np.zeros(shape=(1, n_output))
parameters = [wo, bo]

# loss function: mean squared error
def loss(y, y_hat):
    return np.mean(np.square(y - y_hat))

# Output from the ANN model: prediction process
def predict(x):
    p = parameters
    o_output = np.dot(x, p[0]) + p[1]  # output from output layer
    return o_output

# Perform training and track the loss history.
def train(x, y, x_val, y_val, epochs, batch_size):
    ht_loss = []  # loss history of training data
    hv_loss = []  # loss history of validation data
    for epoch in range(epochs):
        # measure the losses during training
        ht_loss.append(loss(y, predict(x)))
        hv_loss.append(loss(y_val, predict(x_val)))
```



$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

$$\hat{y}=X\cdot W+b$$

MX-AI

- Implement linear regression using numerical differentiation: single-layered Perceptron

```
        # Perform training using mini-batch gradient descent
        for batch in range(int(x.shape[0] / batch_size)):
            idx = np.random.choice(x.shape[0], batch_size)
            gradient_descent(x[idx], y[idx], alpha, loss,
                             predict, parameters)

        if epoch % 10 == 0:
            print("{}: train_loss={:.4f}, val_loss={:.4f}".\
                  format(epoch, ht_loss[-1], hv_loss[-1]))
    return ht_loss, hv_loss

# Perform training
train_loss, val_loss = train(x_train, y_train, x_valid, y_valid,
                             epochs=100, batch_size=50)

# Visually check the loss history.
plt.plot(train_loss, c='blue', label='train loss')
plt.plot(val_loss, c='red', label='validation loss')
plt.legend()
plt.show()

# Visually check the prediction result.
y_pred = predict(x_test)
plt.figure(figsize=(7,5))
plt.scatter(x_train, y_train, s=20, c='blue', alpha=0.3,
            label='train')
plt.scatter(x_valid, y_valid, s=20, c='red', alpha=0.3,
            label='validation')
plt.scatter(x_test, y_pred, s=5, c='red', label='test')
plt.legend()
plt.show()
print(parameters)
```

```
 0: train_loss=0.1543, val_loss=0.1619
10: train_loss=0.0488, val_loss=0.0456
20: train_loss=0.0415, val_loss=0.0407
...
80: train_loss=0.0429, val_loss=0.0420
90: train_loss=0.0429, val_loss=0.0421
```

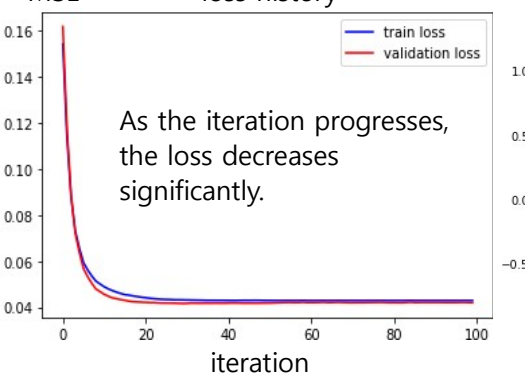
parameter: wo, bo

```
[array([[0.50171724]]), array([[0.29801076]])]
```

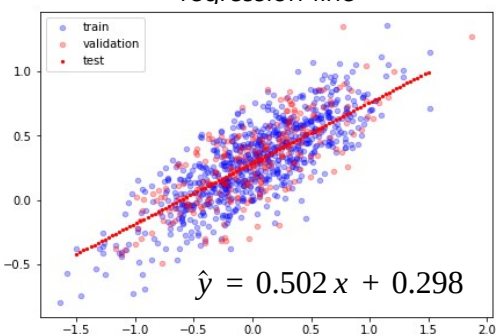
The equation of the original line: $y = 0.5x + 0.3$

Predicted line: $\hat{y} = 0.502x + 0.298$

MSE loss history



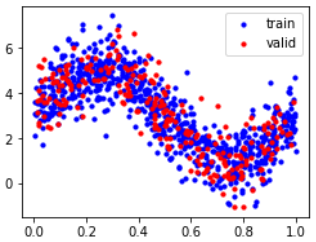
regression line



MX-AI

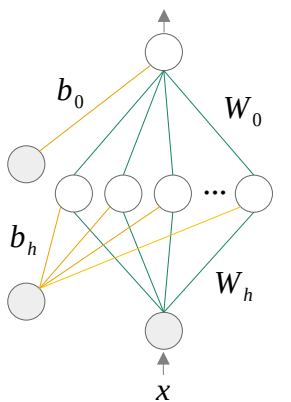
■ Implement non-linear regression using numerical differentiation: two-layered Perceptron

```
# [MXDL-1-07] 7.nonlinear_regression.py
# Create a two-layered ANN model and perform non-linear regression
# using numerical differentiation and gradient descent.
import numpy as np
from gradient_descent import gradient_descent
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

# Generate training data set
x = np.random.random((1000, 1))
y = 2.0 * np.sin(2.0 * np.pi * x) + \
    np.random.normal(0.0, 0.8, (1000, 1))
```




```
# Generate training, validation, and test data sets
x_train, x_valid, y_train, y_valid = train_test_split(x, y)
x_test = np.linspace(0, 1, 200).reshape(-1, 1)

# See the data visually.
plt.figure(figsize=(7,5))
plt.scatter(x_train, y_train, s=20, c='blue', alpha=0.3,
            label='train')
plt.scatter(x_valid, y_valid, s=20, c='red', alpha=0.3,
            label='valid')
plt.legend()
plt.show()

# Create a two-layered ANN model.
n_input = x.shape[1]       # number of input neurons
n_output = 1               # number of output neurons
n_hidden = 16              # number of hidden1 neurons
alpha = 0.01   # learning rate
h = 1e-4       # constant for numerical differentiation

# initialize the parameters randomly.
wh = np.random.normal(size=(n_input, n_hidden))
bh = np.zeros(shape=(1, n_hidden))
wo = np.random.normal(size=(n_hidden, n_output))
bo = np.zeros(shape=(1, n_output))
parameters = [wh, bh, wo, bo]

# loss function: mean squared error
def loss(y, y_hat):
    return np.mean(np.square(y - y_hat))

# Output from the ANN model
def predict(x):
    p = parameters
    h_out = np.tanh(np.dot(x, p[0]) + p[1])
    o_out = np.dot(h_out, p[2]) + p[3]
    return o_out

# Perform training and track the loss history.
def train(x, y, x_val, y_val, epochs, batch_size):
    ht_loss = []  # loss history of training data
    hv_loss = []  # loss history of validation data
    for epoch in range(epochs):
        # measure the losses during training
        ht_loss.append(loss(y, predict(x)))
        hv_loss.append(loss(y_val, predict(x_val)))
```

$$\hat{y} = f_h(X \cdot W_h + b_h) \cdot W_o + b_o$$



MX-AI

■ Implement non-linear regression using numerical differentiation: two-layered Perceptron

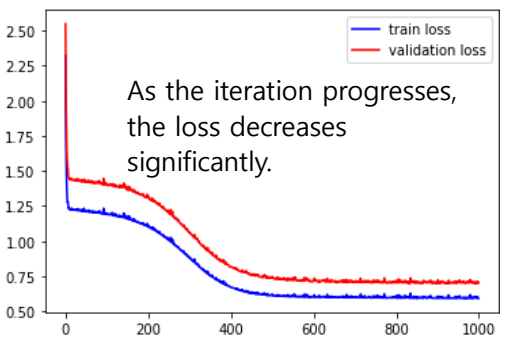
```
        # Perform training using mini-batch gradient descent
        for batch in range(int(x.shape[0] / batch_size)):
            idx = np.random.choice(x.shape[0], batch_size)
            gradient_descent(x[idx], y[idx], alpha, loss,
                             predict, parameters)

        if epoch % 10 == 0:
            print("{}: train_loss={:.4f}, val_loss={:.4f}".\
                  format(epoch, ht_loss[-1], hv_loss[-1]))

    return ht_loss, hv_loss

# Perform training
train_loss, val_loss = train(x_train, y_train,
                             x_valid, y_valid,
                             epochs=1000,
                             batch_size=50)

# Visually check the loss history.
plt.plot(train_loss, c='blue', label='train loss')
plt.plot(val_loss, c='red', label='validation loss')
plt.legend()
plt.show()

# Visually check the prediction result.
y_pred = predict(x_test)
plt.figure(figsize=(7,5))
```

```
plt.scatter(x_train, y_train, s=20, c='blue', alpha=0.3, label='train')
plt.scatter(x_valid, y_valid, s=20, c='red', alpha=0.3,
            label='validation')
plt.scatter(x_test, y_pred, s=5, c='red', label='test')
plt.legend()
plt.show()
```

```
0: train_loss=2.3254, val_loss=2.5496
10: train_loss=1.2254, val_loss=1.4387
20: train_loss=1.2228, val_loss=1.4369
30: train_loss=1.2312, val_loss=1.4450
...
980: train_loss=0.5931, val_loss=0.7081
990: train_loss=0.5984, val_loss=0.7101
```

MSE

loss history



regression line