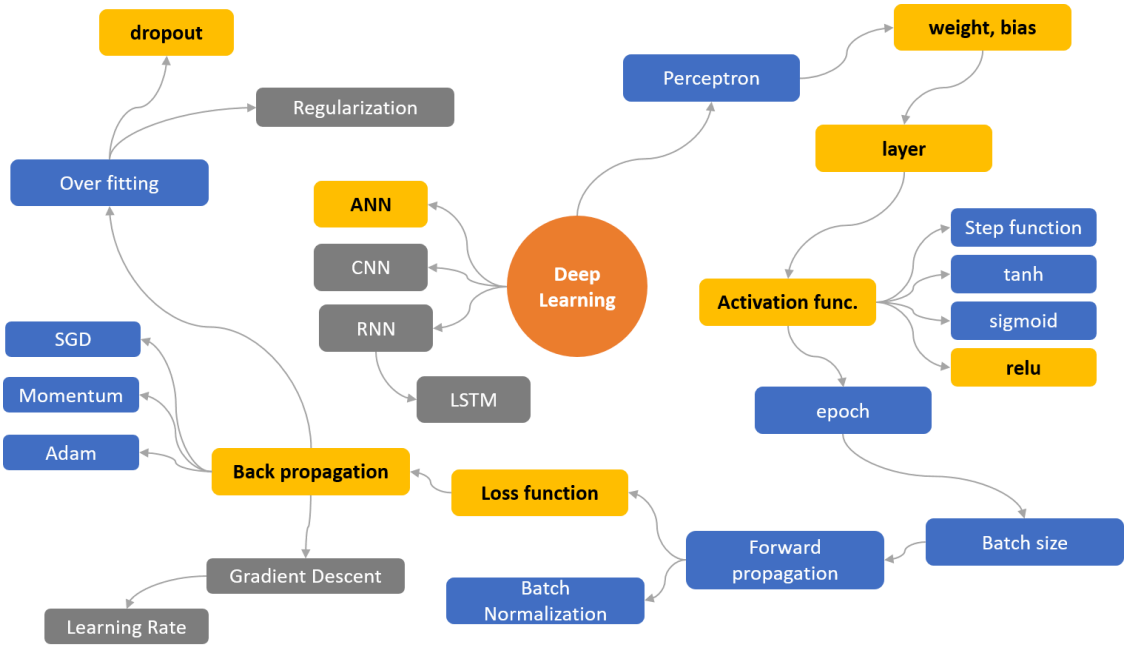
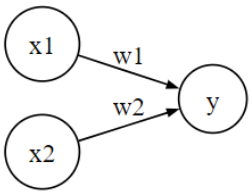


Perceptron

단층 퍼셉트론

- w1, w2: weight
- theta: bias(편향)
- weight, bias 가 parameter
- 아래 단층 퍼셉트론의 모수는 모두 3개(w1, w2, theta)

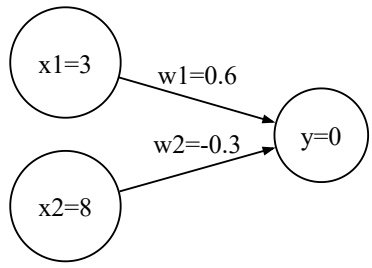


$$y = \begin{cases} 0 & if \quad w_1x_1 + w_2x_2 <= \theta \\ 1 & if \quad w_1x_1 + w_2x_2 > \theta \end{cases}$$

In [1]:
```
from graphviz import Digraph

# 그래프 정의
dot = Digraph()

# 노드 추가
dot.node('Input1', 'x1=3', shape='circle')
dot.node('Input2', 'x2=8', shape='circle')
dot.node('Output', 'y=0', shape='circle')

# 연결 추가 (화살표 크기 작게 조절, 레이블 추가)
dot.edge('Input1', 'Output', label='w1=0.6', arrowsize='0.5')
dot.edge('Input2', 'Output', label='w2=-0.3', arrowsize='0.5')

# 그래프 속성 설정 (오른쪽에서 왼쪽으로 표시)
dot.attr(rankdir='LR')


print("theta = -1 인 경우, 3 * 0.6 + 8 * -0.3 = -0.6 <= -1")
# 그래프 표시
dot


```

```
theta = -1 인 경우, 3 * 0.6 + 8 * -0.3 = -0.6 <= -1
```

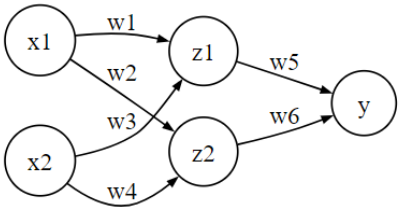
Out[1]:



Multi-Layer Perceptron

Perceptron 을 여러개의 층으로 쌓아올린 것으로 보다 복잡한 분류 작업을 할 수 있습니다. 입력층부터 0층, 1층, 2층 순서로 부르는 것이 관례이며, 마지막층은 출력층이라고 부릅니다.

- 아래 MLP 의 모수는 모두 9개(w1 ~ 6, theta1 ~ 3)

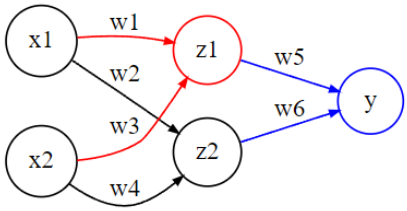


$$z_1 = \begin{cases} 0 & if \quad w_1x_1 + w_2x_2 <= \theta_1 \\ 1 & if \quad w_1x_1 + w_2x_2 > \theta_1 \end{cases}$$

$$z_2 = \begin{cases} 0 & if \quad w_1x_1 + w_2x_2 <= \theta_2 \\ 1 & if \quad w_1x_1 + w_2x_2 > \theta_2 \end{cases}$$

$$y = \begin{cases} 0 & if \quad w_1z_1 + w_2z_2 <= \theta_3 \\ 1 & if \quad w_1z_1 + w_2z_2 > \theta_3 \end{cases}$$

계산 과정



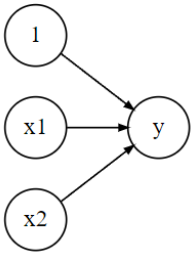
ANN(Artificial Nueral Network)

신경망의 기본 구조는 MLP 와 동일합니다. 실제로 MLP와 인공신경망을 동일한 의미로 사용하기도 합니다. 엄밀히 구분을 하면 "활성화 함수"라는 개념이 확장된 것이 인공신경망입니다. 퍼셉트론은 "Step Function(계단함수)"를 이용해서 퍼셉트론의 활성화 여부를 결정합니다. 반면, 신경망은 활성화 함수를 보다 확장해서 sigmoid, tanh, relu 등을 적용합니다. 계단함수를 제외한 신경망은 "미분가능"하며, 이로 인하여 역전파를 이용한 weight update 가 가능해졌습니다. 입력층과 출력층을 제외한 중간층을 "은닉층"이라고 부르며, 은닉층이 1개인 경우 Simple Nueral Network 라고 부르고, 은닉층이 2개 이상이면 Deep Nueral Network 라고 부릅니다.

퍼셉트론을 기준으로 활성화 함수를 표현한다면,

$$y = \begin{cases} 0 & if \quad w_1x_1 + w_2x_2 <= \theta \\ 1 & if \quad w_1x_1 + w_2x_2 > \theta \end{cases}$$

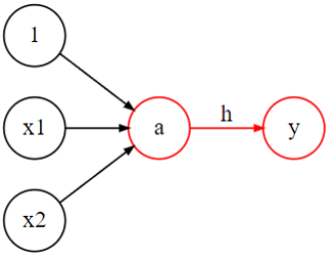
는 아래와 같이 다시 표현할 수 있습니다. 아래의 h(x) 가 계단함수라고 부르는 활성화 함수이며, 입력값의 선형 변환 결과를 활성화 함수를 이용해서 퍼셉트론을 활성화시킵니다. 파라메터 θ 를 -b 로 치환하며, bias 라고 부릅니다. Bias 를 하나의 뉴런으로 해석하여 퍼셉트론의 활성화 조건에 포함되어 있던 파라메터를 명확하게 표현할 수 있습니다. DNN을 도식화할 때에는 Bias는 대체로 생략해서 표현합니다.



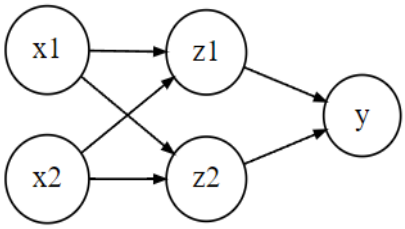
$$h(x) = \begin{cases} 0 & if \quad x <= 0 \\ 1 & if \quad x > 0 \end{cases} \qquad \dots \quad StepFunction$$

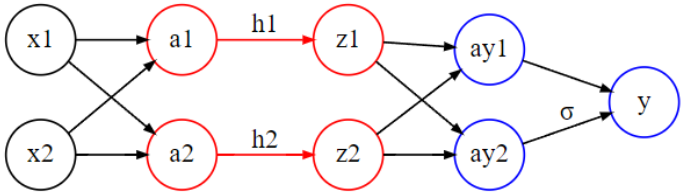
$$y = h(b + w_1x_1 + w_2x_2)$$

출력층에서 활성화 함수를 처리하는 순서입니다. 입력층에서 전달받은 신호를 선형 결합하고, 그 결과를 활성화함수 h를 이용해서 최종 결과 y를 산출합니다.



층이 늘어나는 경우에도 동일한 형태로 전달 받은 신호에 대해서 선형 결합을 하고(Affine 변환), 활성화 함수를 이용해서 다음 층으로 출력 신호를 전달합니다. 활성화 함수는 뉴런 단위로 지정할 수 있지만, 은닉층에 전체에 하나의 활성화 함수를 사용하는 것이 일반적입니다. 출력층의 활성화 함수는 목적에 따라 정의하는데, 회귀 분석의 목적으로는 Identity Function, 이진 분류를 위해서는 Sigmoid, 다중클래스 분류를 위해서는 Softmax Function을 사용합니다.

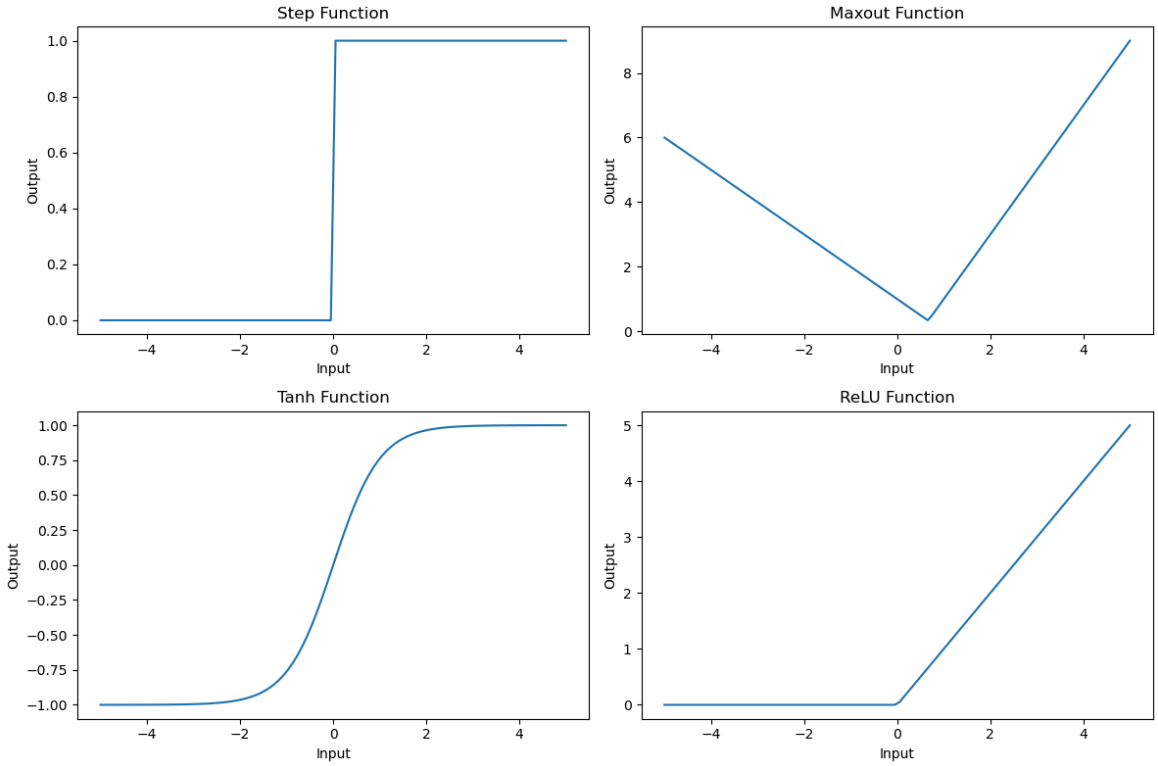




활성화 함수

In [2]:

```python
import numpy as np
import matplotlib.pyplot as plt

# 계단 함수(Step function)
def step_function(x):
    return np.where(x <= 0, 0, 1)

# 시그모이드 함수(Sigmoid function)
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# 하이퍼볼릭 탄젠트 함수(Tanh function)
def tanh(x):
    return np.tanh(x)

# ReLU 함수(Rectified Linear Unit function)
def relu(x):
    return np.maximum(0, x)

# Maxout activation function
def maxout(x, w1, b1, w2, b2):
    return np.maximum(np.dot(x, w1) + b1, np.dot(x, w2) + b2)


# 입력 데이터 생성
x = np.linspace(-5, 5, 100)

# 각 활성화 함수의 출력 계산
y_step = step_function(x)
y_sigmoid = sigmoid(x)
y_tanh = tanh(x)
y_relu = relu(x)
y_maxout = maxout(x, 2, -1, -1, 1)

# 활성화 함수 그래프 그리기
plt.figure(figsize=(12, 8))

# subplot 1: 계단 함수
plt.subplot(2, 2, 1)
plt.plot(x, y_step)
plt.title('Step Function')
plt.xlabel('Input')
plt.ylabel('Output')

# subplot 2: 시그모이드 함수
# plt.subplot(2, 2, 2)
# plt.plot(x, y_sigmoid)
# plt.title('Sigmoid Function')
# plt.xlabel('Input')
# plt.ylabel('Output')
plt.subplot(2, 2, 2)
plt.plot(x, y_maxout)
plt.title('Maxout Function')
plt.xlabel('Input')
plt.ylabel('Output')


# subplot 3: 하이퍼볼릭 탄젠트 함수
plt.subplot(2, 2, 3)
plt.plot(x, y_tanh)
plt.title('Tanh Function')
plt.xlabel('Input')
plt.ylabel('Output')

# subplot 4: ReLU 함수
plt.subplot(2, 2, 4)
plt.plot(x, y_relu)
plt.title('ReLU Function')
plt.xlabel('Input')
plt.ylabel('Output')

plt.tight_layout()
plt.show()

```



출력층의 활성화 함수

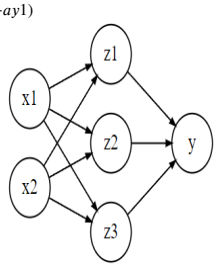
신용평가모형은 우량과 불량을 구분하는 Binary Classification 입니다. 이를 위해서는 로지스틱 회귀분석과 동일하게 Cross-Entropy-Loss를 사용합니다. Nueral Network의 출력층은 예측하고자 하는 Class와 동일하게 구성하며, 출력층의 활성화 함수는 sigmoid 또는 softmax를 이용합니다.

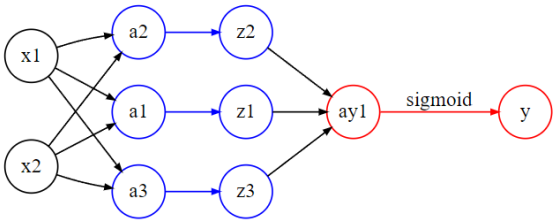
시각적인 효과를 위하여 간단한 구조의 NN을 구성합니다. 본 강의에서는 일반적인 형태인 sigmoid를 사용하여 Neural Network를 구성합니다.

sigmoid

Binary Classification 에서는 하나의 Class의 확률만 예측하여 분류를 수행할 수 있습니다. 출력층의 구조가 간단하기 때문에 sigmoid를 사용하는 것이 일반적입니다. 출력층에서 affine 변환을 통해서 산출된 ay1를 sigmoid 함수에 통과 시켜서 최종 확률값을 산출합니다.

$$y = \frac{1}{1+exp(-ay1)}$$



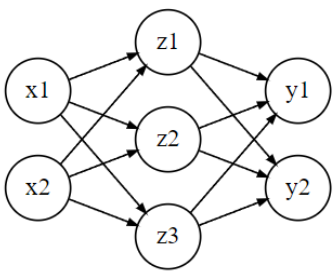


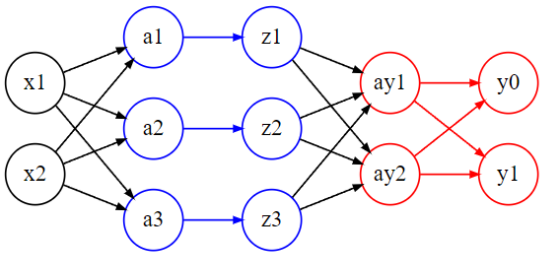
softmax

softmax 함수는 다중 클래스 분류에서 활용합니다. 각 클래스의 결과 모두 더하면 1이 되도록 구성하기 때문에 클래스의 확률로 해석할 수 있습니다. 활성화 함수는 뉴런 내에서만 적용하는 것이 일반적이지만, softmax 활성화 함수는 출력층 내의 뉴런간 교차 연산이 필요합니다.

$$y0 = \frac{exp(ay0}{exp(ay0)+exp(ay1)}$$

$$y1 = \frac{exp(ay1}{exp(ay0)+exp(ay1)}$$





손실함수

Classification 에서는 cross-entropy-loss 를 손실함수로 사용합니다.(회귀분석에서는 SSE)

$$CrossEntropyLoss = -(ylog(\hat{y})) + (1-y)log(1-\hat{y}))$$

손실함수(L) 를 모든 데이터에 대해서 표현하면,

$$L = -\sum_i^N -(y_i log(\hat{y}_i)) + (1-y_i log(1-\hat{y}_i))$$

와 같이 표현할 수 있고 모든 데이터에 대한 평균적인 지표를 구할 수 있습니다. 대용량 데이터의 경우에는 매번 모든 데이터에 대한 손실을 계산하고 매개변수를 찾아가는 것이 쉽지 않습니다. 효율적인 방법을 위해서 **미니배치 학습** 을 수행합니다. 전체 데이터에서 무작위로 일부를 추출하고, 이를 이용해서 손실함수를 구하고 손실함수를 줄이기 위한 방향으로 매개변수를 업데이트 하는 방법입니다. 이때, 무작위로 추출하는 크기를 "batch size" 라고 합니다. 또한, 전체 데이터를 1번 사용하는 것을 1 epoch라고 하며, 100 epoch 만큼 학습한다는 것은 전체 데이터를 100번 사용한다는 뜻입니다.

회귀분석에서 학습했던 Gradient Descent에서는 데이터를 1개씩 처리하면서 매개변수를 업데이트 했지만, 여러개의 데이터를 한번에 사용하는 것이 효율적이면서도 Local Minimum을 회피할 수 있는 방법입니다. 미니배치 학습을 통한 Gradient Descent를 Stochastic Gradient Decent(확률적 경사하강법)라고 부릅니다.

Forward Propagation

손실함수까지 산출하는 것이 순전파라고 부르는 Forward Propagation 단계 입니다. 이후 매개변수 업데이트를 위하여 오류역전파(Backward Propagation of erros)를 활용합니다.

Nueral Network 구성

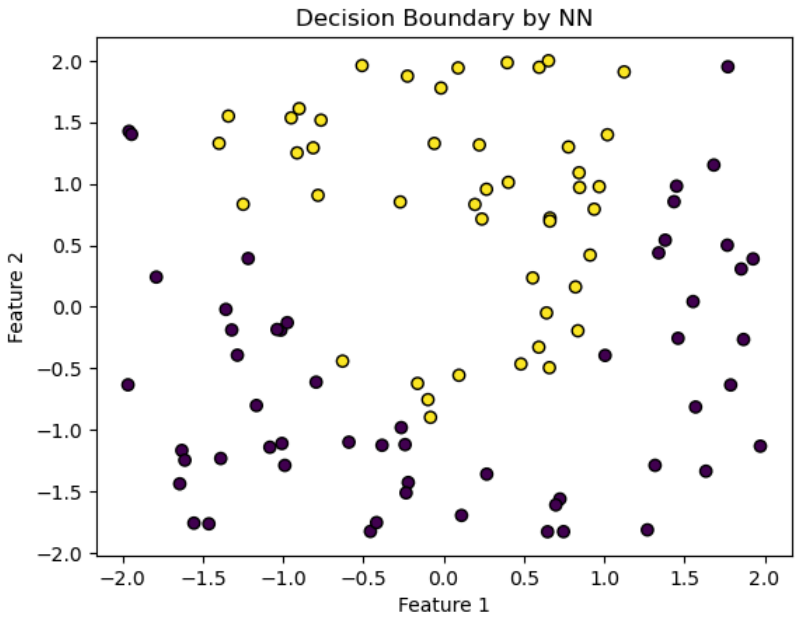
- Input Layer: 2개(x1, x2)
- Hidden Layer: 1층, 3개
  - activation function: relu
- Output Layer: 1개(y)
  - activation function: sigmoid
- Loss: Cross Entropy

- Parameters: 13개
  - weight: 9개
    - x1 에서 a1, a2, a3: w11, w12, w13
    - x2 에서 a1, a2, a3: w21, w22, w23
    - z1, z2, z3 에서 ay: w31, w32, w33
  - bias: 4개
    - 입력층에서 은닉층: b11, b12, b13
    - 은닉층에서 출력층: b21

In [3]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# 데이터셋 생성
np.random.seed(42)
X = np.random.rand(100, 2) * 4 - 2  # -2에서 2 사이의 난수 생성
y = (X[:, 1] -1 > X[:, 0]**2).astype(int)  # 2차 함수 모양을 따라 y값 생성


```

In [4]:
```
# 데이터
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_classification

# 데이터셋 생성
# X, y = make_classification(n_samples=100, n_features=2, n_informative=2, n_redundant=0, n_classes=2, n_clusters_p

np.random.seed(317)
X = np.random.rand(100, 2) * 4 - 2  # -2에서 2 사이의 난수 생성
y = (X[:, 1] + 1 > X[:, 0]**2).astype(int)  # 2차 함수 모양을 따라 y값 생성

plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Decision Boundary by NN')
plt.show()
```



In [5]:
```
data = pd.DataFrame(X, columns=['x1', 'x2'])
data['y'] = y
data.head()
```

Out[5]:

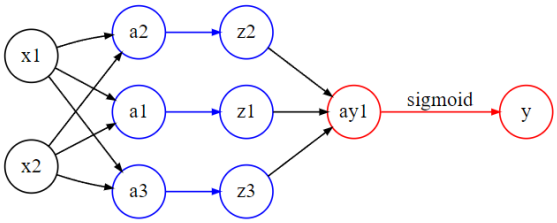
	x1	x2	y
0	-0.782599	0.905209	1
1	1.854088	0.307004	0
2	-1.248618	0.832538	1
3	0.238576	0.712806	1
4	-1.166176	-0.802204	0

In [6]:
```
x1 = data['x1'].values
x2 = data['x2'].values
```

Hyper-Parameter 정의

In [7]:
```
activation = 'relu'
learning_rate = 0.1
```

Forward Propagation



In [8]:

```
# 파라미터 초기화
b11, b12, b13 = 0, 0, 0
b21 = 0
w11, w12, w13 = 0.1, 0.2, 0.3
w21, w22, w23 = 0.1, 0.2, 0.3
w31, w32, w33 = 0.1, 0.3, 0.3

# Foward Propagation
a1 = b11 + w11 * x1 + w21 * x2
a2 = b12 + w12 * x1 + w22 * x2
a3 = b13 + w13 * x1 + w23 * x2

if activation == 'relu':
    z1, z2, z3 = relu(a1), relu(a2), relu(a3)
elif activation == 'step':
    z1, z2, z3 = step_function(a1), step_function(a2), step_function(a3)

ay1 = b21 + w31 * z1 + w32 * z2 + w33 * z3
yhat = sigmoid(ay1)

loss = - ( y * np.log(yhat) + (1-y) * np.log(1-yhat) ).mean()

loss
```

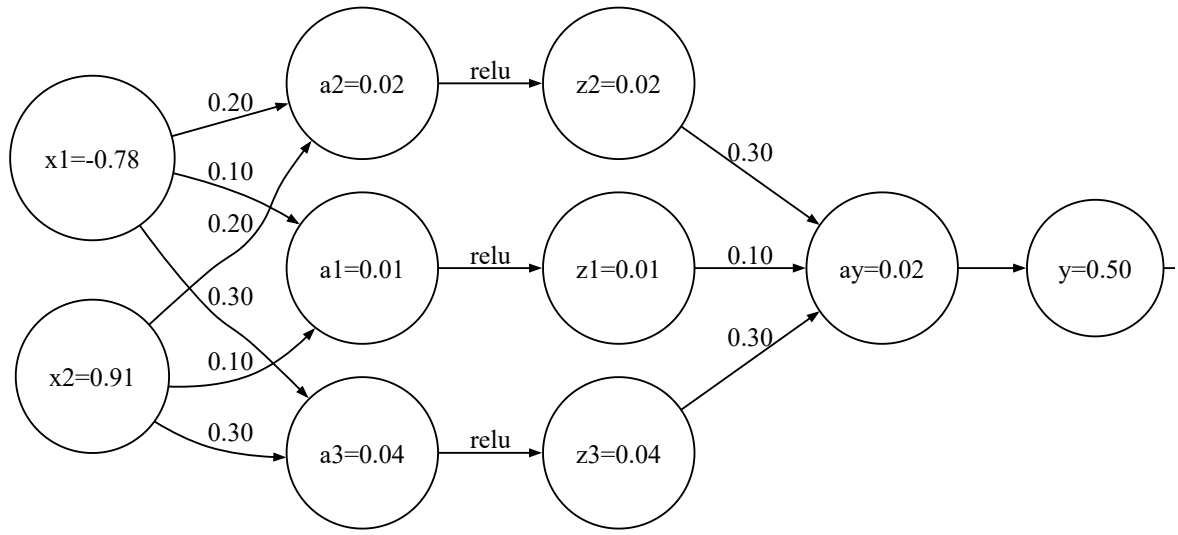
Out[8]: 0.6832187873470724

In [9]:

```
# Layer 도식화 - 첫번째 데이터 기준으로 산출되는 값 처리

def graph():
    dot = Digraph()

    dot.node('x1', 'x1={:.2f}'.format(x1[0]), shape='circle')
    dot.node('x2', 'x2={:.2f}'.format(x2[0]), shape='circle')

    dot.node('a1', 'a1={:.2f}'.format(a1[0]), shape='circle')
    dot.node('a2', 'a2={:.2f}'.format(a2[0]), shape='circle')
    dot.node('a3', 'a3={:.2f}'.format(a3[0]), shape='circle')

    dot.node('z1', 'z1={:.2f}'.format(z1[0]), shape='circle')
    dot.node('z2', 'z2={:.2f}'.format(z2[0]), shape='circle')
    dot.node('z3', 'z3={:.2f}'.format(z3[0]), shape='circle')

    dot.node('ay', 'ay={:.2f}'.format(ay1[0]), shape='circle')
    dot.node('y', 'y={:.2f}'.format(yhat[0]), shape='circle')

    dot.node('loss', 'loss={:.2f}'.format(loss), shape='circle')

    dot.edge('x1', 'a1', label='{:.2f}'.format(w11), arrowsize='0.5')
    dot.edge('x1', 'a2', label='{:.2f}'.format(w12), arrowsize='0.5')
    dot.edge('x1', 'a3', label='{:.2f}'.format(w13), arrowsize='0.5')
    dot.edge('x2', 'a1', label='{:.2f}'.format(w21), arrowsize='0.5')
    dot.edge('x2', 'a2', label='{:.2f}'.format(w22), arrowsize='0.5')
    dot.edge('x2', 'a3', label='{:.2f}'.format(w23), arrowsize='0.5')

    dot.edge('a1', 'z1', label=f'{activation}', arrowsize='0.5')
    dot.edge('a2', 'z2', label=f'{activation}', arrowsize='0.5')
    dot.edge('a3', 'z3', label=f'{activation}', arrowsize='0.5')

    dot.edge('z1', 'ay', label='{:.2f}'.format(w31), arrowsize='0.5')
    dot.edge('z2', 'ay', label='{:.2f}'.format(w32), arrowsize='0.5')
    dot.edge('z3', 'ay', label='{:.2f}'.format(w33), arrowsize='0.5')

    dot.edge('ay', 'y', arrowsize='0.5')

    dot.edge('y', 'loss', arrowsize='0.5')

    dot.attr(rankdir='LR')

    dot

    print('b11:', round(b11, 3), '  / b12:', round(b12, 3), '  / b13:', round(b13, 3), '  / b21:', round(b21, 3))
    return dot


graph()
```

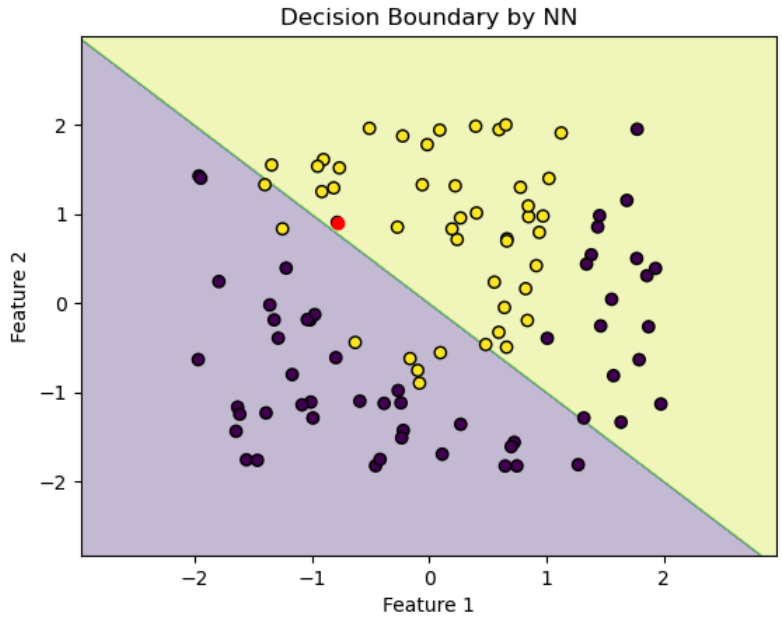
```
b11: 0   / b12: 0   / b13: 0   / b21: 0
```

Out[9]:



In [10]:
```
# Forward Propagation 을 이용해서 최종 판단 결과를 예측
def predict_nn(xx, yy):
    a1 = b11 + w11 * xx + w21 * yy
    a2 = b12 + w12 * xx + w22 * yy
    a3 = b13 + w13 * xx + w23 * yy

    if activation == 'relu':
        z1, z2, z3 = relu(a1), relu(a2), relu(a3)
    elif activation == 'step':
        z1, z2, z3 = step_function(a1), step_function(a2), step_function(a3)

    ay1 = b21 + w31 * z1 + w32 * z2 + w33 * z3
    yhat = sigmoid(ay1)
    y_pred = yhat > 0.5

    return y_pred * 1

```

In [11]:
```
# 결정경계 시각화
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))

Z = predict_nn(xx, yy)
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')
plt.scatter([X[0,0]], [X[0,1]], color='red', zorder=5)

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Decision Boundary by NN')
plt.show()
```



수치 미분을 이용한 Gradient Descent

수치 미분은 손실 함수의 Gradient를 계산하기 위해 사용되는 기법 중 하나입니다. 이는 작은 변화에 대한 함수의 변화를 근사적으로 계산하여 Gradient를 추정하는 방법입니다. Gradient Descent에서는 이러한 추정된 Gradient를 사용하여 매개변수를 업데이트합니다.

손실 함수 정의: 먼저 최적화할 손실 함수를 정의합니다. 일반적으로 평균 제곱 오차(Mean Squared Error)나 크로스 엔트로피(Cross Entropy) 등의 손실 함수를 사용합니다.

초기화: 매개변수(가중치)를 임의의 값으로 초기화합니다.

Gradient 계산: 현재의 매개변수에 대해 손실 함수의 Gradient를 계산합니다. Gradient는 손실 함수의 기울기를 나타내며, 각 매개변수에 대한 편미분으로 구할 수 있습니다.

경사 하강(Gradient Descent): Gradient를 사용하여 매개변수를 업데이트합니다. 경사 하강 알고리즘은 매개변수를 손실 함수의 Gradient의 반대 방향으로 조금씩 이동시킵니다. 이는 손실 함수의 값을 줄이는 방향으로 매개변수를 업데이트하는 것을 의미합니다.

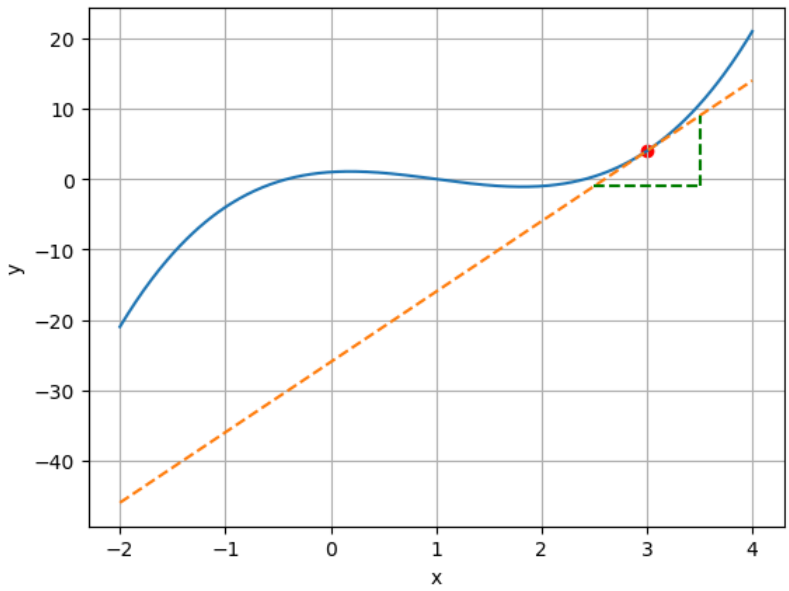
반복: 일정한 반복 횟수나 수렴 기준까지 3단계와 4단계를 반복합니다. 매 반복마다 Gradient를 다시 계산하고 매개변수를 업데이트하여 손실 함수의 값을 최소화합니다.

수렴 확인: 손실 함수의 값이 충분히 작아지거나 일정한 기준에 도달하면 알고리즘이 수렴했다고 판단하고 종료합니다.

수치 미분은 손실 함수의 Gradient를 계산하기 위해 사용되는 기법 중 하나입니다. 이는 작은 변화에 대한 함수의 변화를 근사적으로 계산하여 Gradient를 추정하는 방법입니다. Gradient Descent에서는 이러한 추정된 Gradient를 사용하여 매개변수를 업데이트합니다.

In [12]:

```
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore")

# 3차 함수 정의
def cubic_function(x):
    return x**3 - 3*x**2 + x + 1

# 접선의 기울기와 절편을 계산하는 함수
def tangent_slope_intercept(x, f):
    # 기울기는 도함수(미분)의 값
    slope = 3*x**2 - 6*x + 1
    # 절편은 함수값에서 기울기를 곱한 값 빼기
    intercept = f - slope*x
    return slope, intercept

# 그래프를 그리기 위한 x값 범위 설정
x_values = np.linspace(-2, 4, 400)
y_values = cubic_function(x_values)

# 3차 함수 그래프 그리기
plt.plot(x_values, y_values)

# 접선의 x 좌표 설정
x_tangent = 3
h = 0.5
# 접선의 기울기와 절편 계산
slope, intercept = tangent_slope_intercept(x_tangent, cubic_function(x_tangent))

# 접선의 방정식: y = slope * x + intercept
# 접선 그리기
plt.plot(x_values, slope*x_values + intercept, linestyle='--')

# 기울기 표시
plt.plot([x_tangent + h, x_tangent + h], [slope*(x_tangent-h) + intercept, slope*(x_tangent+h) + intercept], color
plt.plot([x_tangent - h, x_tangent + h], [slope*(x_tangent-h) + intercept, slope*(x_tangent-h) + intercept], color


# 접점 표시
plt.scatter(x_tangent, cubic_function(x_tangent), color='red')

# 그래프 축과 범례 설정
plt.xlabel('x')
plt.ylabel('y')


# 그래프 출력
plt.grid(True)
plt.show()

```



목적함수 산출

In [13]:
```
# 전체 데이터에 대해서 loss를 한번에 산출 - 전체 데이터 단위로 학습
# 실제로는 batch_size를 부여하고 batch 단위로 학습
def get_loss( params ):
    _a1 = params['b11'] + params['w11'] * x1 + params['w21'] * x2
    _a2 = params['b12'] + params['w12'] * x1 + params['w22'] * x2
    _a3 = params['b13'] + params['w13'] * x1 + params['w23'] * x2

    if activation == 'relu':
        _z1, _z2, _z3 = relu(_a1), relu(_a2), relu(_a3)
    elif activation == 'step':
        _z1, _z2, _z3 = step_function(_a1), step_function(_a2), step_function(_a3)
    elif activation == 'tanh':
        _z1, _z2, _z3 = tanh(_a1), tanh(_a2), tanh(_a3)

    _ay1 = params['b21'] + params['w31'] * _z1 + params['w32'] * _z2 + params['w33'] * _z3
    _yhat = sigmoid(_ay1)

#     print(params)
#     print(_a1, _a2, _a3, _z1, _z2, _z3, _ay1, _yhat, - ( y * np.log(_yhat) + (1-y) * np.log(1-_yhat) ).mean())

    return - ( y * np.log(_yhat) + (1-y) * np.log(1-_yhat) ).mean()
```

기울기 산출

- 개별 파라미터 각각에 대한 수치미분으로 기울기를 산출
- 정해진 learning_rate 에 따라 각 파라미터를 조정

In [14]:
```
# 개별 파라미터에 대한 기울기 산출
def get_gradient():
    h = 1e-7

    params = {'b11': b11, 'b12': b12, 'b13': b13, 'b21': b21,
              'w11': w11, 'w12': w12, 'w13': w13,
              'w21': w21, 'w22': w22, 'w23': w23,
              'w31': w31, 'w32': w32, 'w33': w33}

    grad = {}

    for key, value in params.items():
        _params = params.copy()
        _params[key] = params[key] - h
        loss1 = get_loss(_params)

        _params[key] = params[key] + h
        loss2 = get_loss(_params)

        grad[key] = (loss2 - loss1) / (2 * h)

    return grad

grad = get_gradient()
grad
```

Out[14]:
```
{'b11': -0.006909978145230866,
 'b12': -0.0207299344356926,
 'b13': -0.0207299344356926,
 'b21': 0.08090021430540162,
 'w11': 0.012743330835007782,
 'w12': 0.03822999361524637,
 'w13': 0.03822999361524637,
 'w21': -0.016027190130607494,
 'w22': -0.048081571502045506,
 'w23': -0.048081571502045506,
 'w31': -0.003283858185376687,
 'w32': -0.006567718036087911,
 'w33': -0.009851577331687622}
```

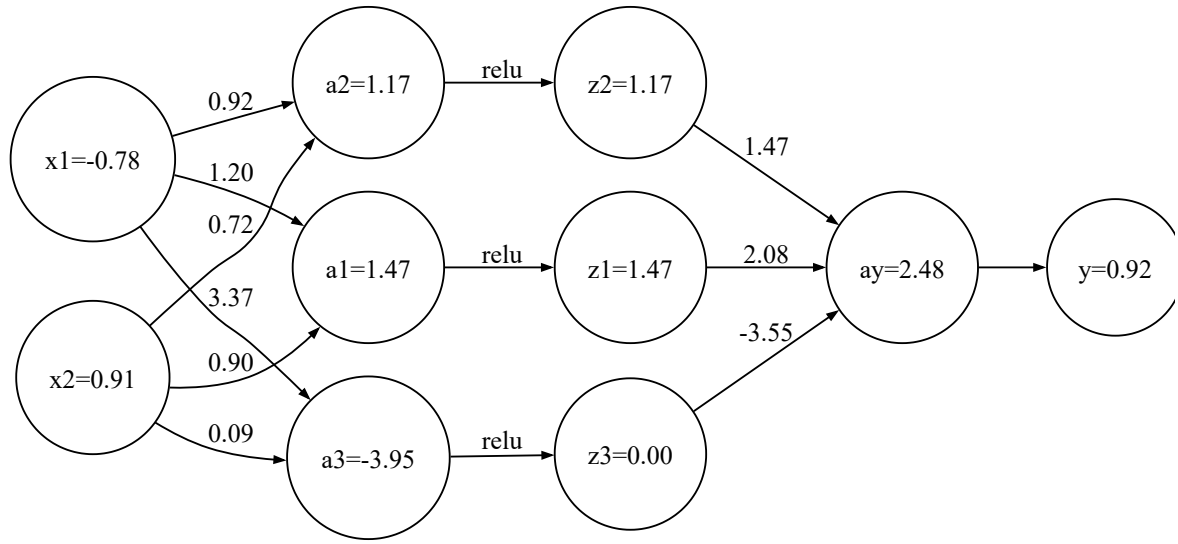
전체 학습

In [15]:

```
# 초기화
b11, b12, b13 = 0, 0, 0
b21 = 0

w11, w12, w13 = 0.2, 0.3, 0.8
w21, w22, w23 = 0.7, 0.8, 0.3
w31, w32, w33 = 0.3, 0.1, 0.2

iteration_num = 0

learning_rate = 0.1
epoch = 1000
activation = 'relu'
loss_bucket = []

for i in range(epoch):

    iteration_num += 1

    # gradient descent
    grad = get_gradient()

    # update params
    b11 -= grad['b11'] * learning_rate
    b12 -= grad['b12'] * learning_rate
    b13 -= grad['b13'] * learning_rate
    b21 -= grad['b21'] * learning_rate

    w11 -= grad['w11'] * learning_rate
    w12 -= grad['w12'] * learning_rate
    w13 -= grad['w13'] * learning_rate

    w21 -= grad['w21'] * learning_rate
    w22 -= grad['w22'] * learning_rate
    w23 -= grad['w23'] * learning_rate

    w31 -= grad['w31'] * learning_rate
    w32 -= grad['w32'] * learning_rate
    w33 -= grad['w33'] * learning_rate

    # Foward Propagation
    a1 = b11 + w11 * x1 + w21 * x2
    a2 = b12 + w12 * x1 + w22 * x2
    a3 = b13 + w13 * x1 + w23 * x2

    if activation == 'relu':
        z1, z2, z3 = relu(a1), relu(a2), relu(a3)
    elif activation == 'step':
        z1, z2, z3 = step_function(a1), step_function(a2), step_function(a3)

    ay1 = b21 + w31 * z1 + w32 * z2 + w33 * z3
    yhat = sigmoid(ay1)

    loss = - ( y * np.log(yhat) + (1-y) * np.log(1-yhat) ).mean()

    loss_bucket.append(loss)


graph()
```

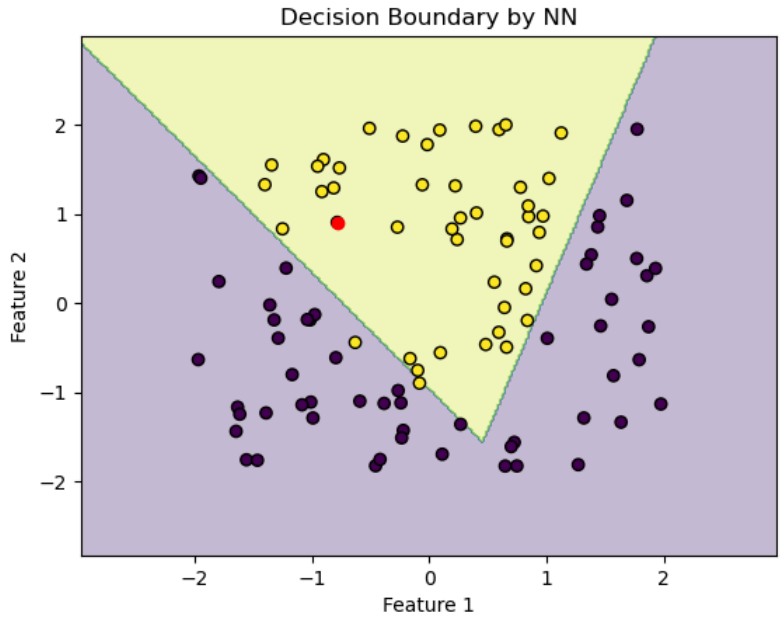
```
b11: 1.587   / b12: 1.236   / b13: -1.398   / b21: -2.293
```

Out[15]:



In [16]:

```
# 시각화
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02), np.arange(y_min, y_max, 0.02))

Z = predict_nn(xx, yy)
Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3)
plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k')
plt.scatter(X[0, 0], X[0, 1], c='red')

plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Decision Boundary by NN')
plt.show()
```



In [17]:

```
b11, w11, w21
```

Out[17]: (1.587260123697698, 1.1973577285438128, 0.901568947325037)

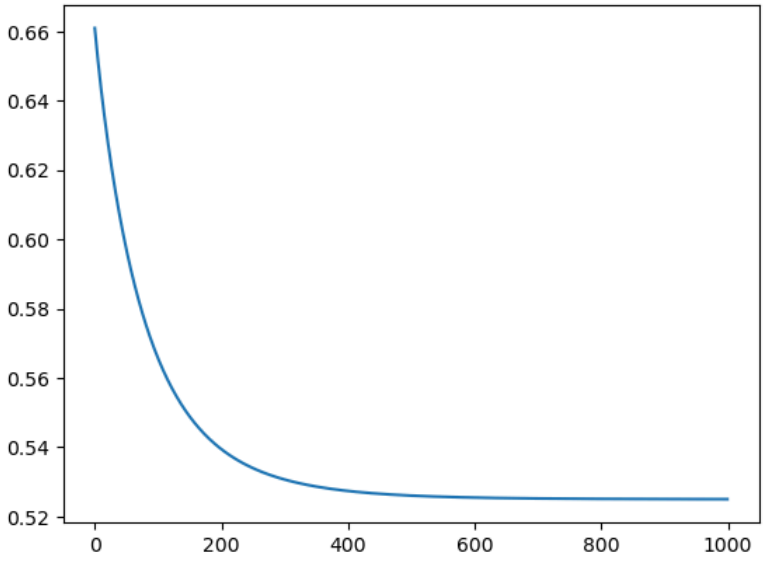
In [18]:
```
def predict_nn(xx, yy):
    a1 = b11 + w11 * xx + w21 * yy
    a2 = b12 + w12 * xx + w22 * yy
    a3 = b13 + w13 * xx + w23 * yy

    if activation == 'relu':
        z1, z2, z3 = relu(a1), relu(a2), relu(a3)
    elif activation == 'step':
        z1, z2, z3 = step_function(a1), step_function(a2), step_function(a3)

    ay1 = b21 + w31 * z1 + w32 * z2 + w33 * z3
    yhat = sigmoid(ay1)
    y_pred = yhat > 0.5

    return y_pred * 1

predict_nn(xx, yy)
```

Out[18]:
```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [1, 1, 1, ..., 0, 0, 0],
       [1, 1, 1, ..., 0, 0, 0],
       [1, 1, 1, ..., 0, 0, 0]])
```

In [19]:

```
# 초기화
b11, b12, b13 = 0, 0, 0
b21 = 0

w11, w12, w13 = 0.2, 0.3, 0.8
w21, w22, w23 = 0.7, 0.8, 0.3
w31, w32, w33 = 0.3, 0.1, 0.2

iteration_num = 0

learning_rate = 0.1
epoch = 1000
activation = 'step'
# activation = 'relu'
loss_bucket = []

for i in range(epoch):

    iteration_num += 1

    # Foward Propagation
    a1 = b11 + w11 * x1 + w21 * x2
    a2 = b12 + w12 * x1 + w22 * x2
    a3 = b13 + w13 * x1 + w23 * x2

    if activation == 'relu':
        z1, z2, z3 = relu(a1), relu(a2), relu(a3)
    elif activation == 'step':
        z1, z2, z3 = step_function(a1), step_function(a2), step_function(a3)

    ay1 = b21 + w31 * z1 + w32 * z2 + w33 * z3
    yhat = sigmoid(ay1)

    loss = - ( y * np.log(yhat) + (1-y) * np.log(1-yhat) ).mean()

    loss_bucket.append(loss)

    # gradient descent - 여기서는 수치미분. 실제로는 back propagation
    grad = get_gradient()

    # update params
    b11 -= grad['b11'] * learning_rate
    b12 -= grad['b12'] * learning_rate
    b13 -= grad['b13'] * learning_rate
    b21 -= grad['b21'] * learning_rate

    w11 -= grad['w11'] * learning_rate
    w12 -= grad['w12'] * learning_rate
    w13 -= grad['w13'] * learning_rate

    w21 -= grad['w21'] * learning_rate
    w22 -= grad['w22'] * learning_rate
    w23 -= grad['w23'] * learning_rate

    w31 -= grad['w31'] * learning_rate
    w32 -= grad['w32'] * learning_rate
    w33 -= grad['w33'] * learning_rate

print('전체 데이터 반복횟수: ', iteration_num)
graph()

_ = plt.plot(loss_bucket)
```

```
전체 데이터 반복횟수:  1000
b11: 0.0   / b12: 0.0   / b13: 0.0   / b21: -1.631
```



역전파

역전파(Backpropagation)는 인공 신경망에서 학습 알고리즘인 경사 하강법을 이용하여 모델의 가중치를 조정하기 위한 효율적인 방법입니다.

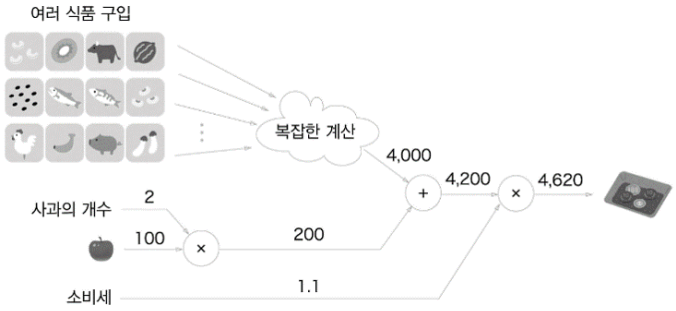
역전파는 입력값을 전파하여 신경망의 출력을 계산한 후, 출력과 실제 값의 차이를 계산합니다. 이 오차를 역방향으로 전파하여 각 가중치에 대한 오차의 기여도를 계산합니다. 역전파 알고리즘은 출력층에서부터 입력층으로 가중치의 그래디언트(기울기)를 계산하고, 이를 이용하여 가중치를 업데이트합니다. 이 과정에서, 미분의 연쇄 법칙을 사용하여 각 층의 오차를 이전 층으로 전파합니다.

본 강의에서는 역전파에 대해서는 구체적으로 다루지 않습니다. 역전파를 가장 쉽게 이해할 수 있는 계산그래프를 이용해서 역전파의 기본 컨셉에 대한 이해를 목적으로 합니다.

아래 계산그래프는 "밑바닥부터 시작하는 딥러닝(한빛미디어, 사이토 고키 지음)"에서 발췌한 내용입니다.

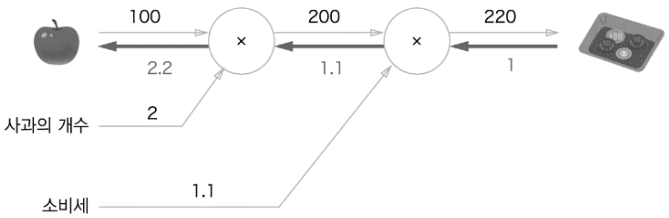
계산 그래프

계산 그래프의 특징은 "국소적 계산"을 전파함으로써 최종 결과를 얻는다는 것입니다. 전체 계산이 복잡하더라도 각 단계에서는 가장 단순한 형태의 계산을 하고, 이를 계속해서 전달하는 것이 계산 그래프입니다. Neural Network에서는 순전파에 해당하는 과정입니다.



계산 그래프를 이용한 미분

계산그래프를 이용하는 이유는 역전파를 통해 "미분"을 효율적으로 계산할 수 있기 때문입니다. 순전파에서 복잡한 계산을 국소적으로 해결한것 처럼, 역전파에서는 복잡한 미분을 국소 미분을 이용하여 결과를 만들어냅니다. 사과 가격이 오르면 최종적으로 지불 금액이 얼마나 오르는지를 알고 싶다고 가정합니다. 사과의 값을 x, 지불 금액을 L 이라고 표현한다면 $\frac{\partial L}{\partial x}$ 를 구하는 것과 동일합니다. 사과값이 아주 조금 올랐을 때 지불 금액이 얼마나 증가하는지를 의미합니다.



미분의 연쇄법칙과 계산 그래프

아래 2개 식이 있에 대해서 x의 변화에 대한 z의 변화를 산출하는 문제는 가정합니다.

$$z = t^2$$

$$t = x + y$$

미분의 연쇄법칙을 이용하여 아래와 같이 도출할 수 있습니다.

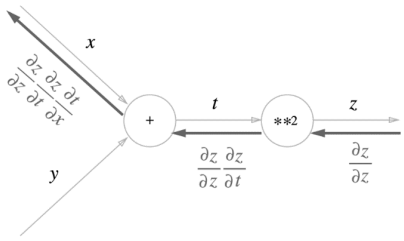
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x}$$

$$\frac{\partial z}{\partial t} = 2t$$

$$\frac{\partial t}{\partial x} = 1$$

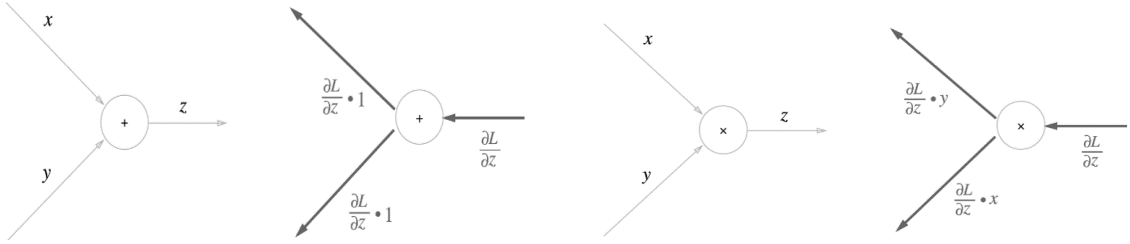
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial t}\frac{\partial t}{\partial x} = 2t \cdot 1 = 2(x + y)$$

위 과정을 계산 그래프로 나타내면 아래와 같습니다.

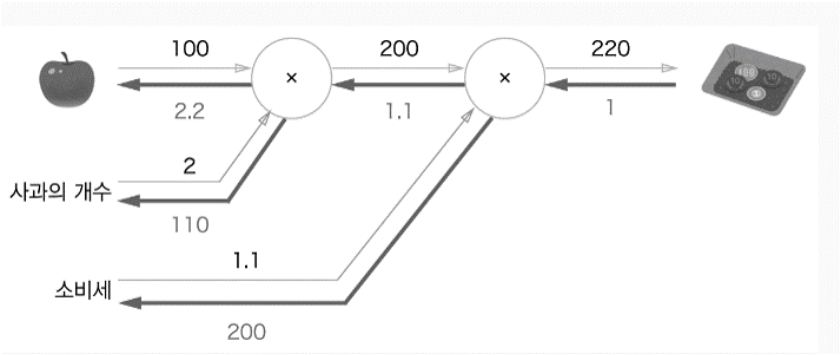


덧셈, 곱셈에 대한 계산 그래프 예시

덧셈은 결과를 상위로 그대로 흘려보내고, 곰셈은 계수만큼을 곱해서 상위로 전달합니다.



사과 문제에서의 역전파 그래프는 아래와 같습니다.



In [20]:

```
# 사과가격이 1원 상승했다면?

price = 100
count = 2
tax_rate = 1.1

graph1 = price * count
graph2 = round(graph1 * tax_rate, 1)

graph1, graph2
```

Out[20]: (200, 220.0)

In [21]:
```
# 사과가격이 1원 상승했다면?
price = 100 + 1
count = 2
tax_rate = 1.1

graph1 = price * count
graph2 = round(graph1 * tax_rate, 1)

graph1, graph2
```

Out[21]: (202, 222.2)

In [ ]:
```

```

In [ ]:
```

```

In [ ]:
```

```