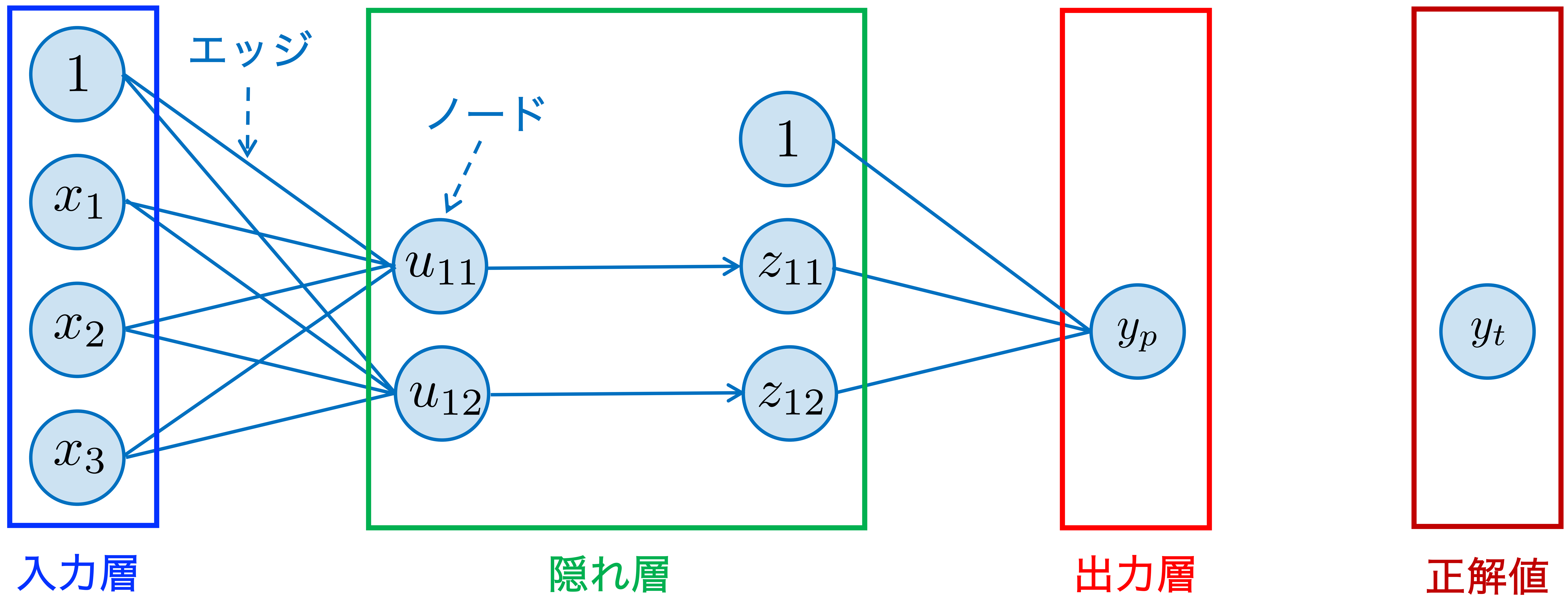
機械学習ゼミ
1. 機械学習の基礎 (ニューラルネットワーク)

別所秀将

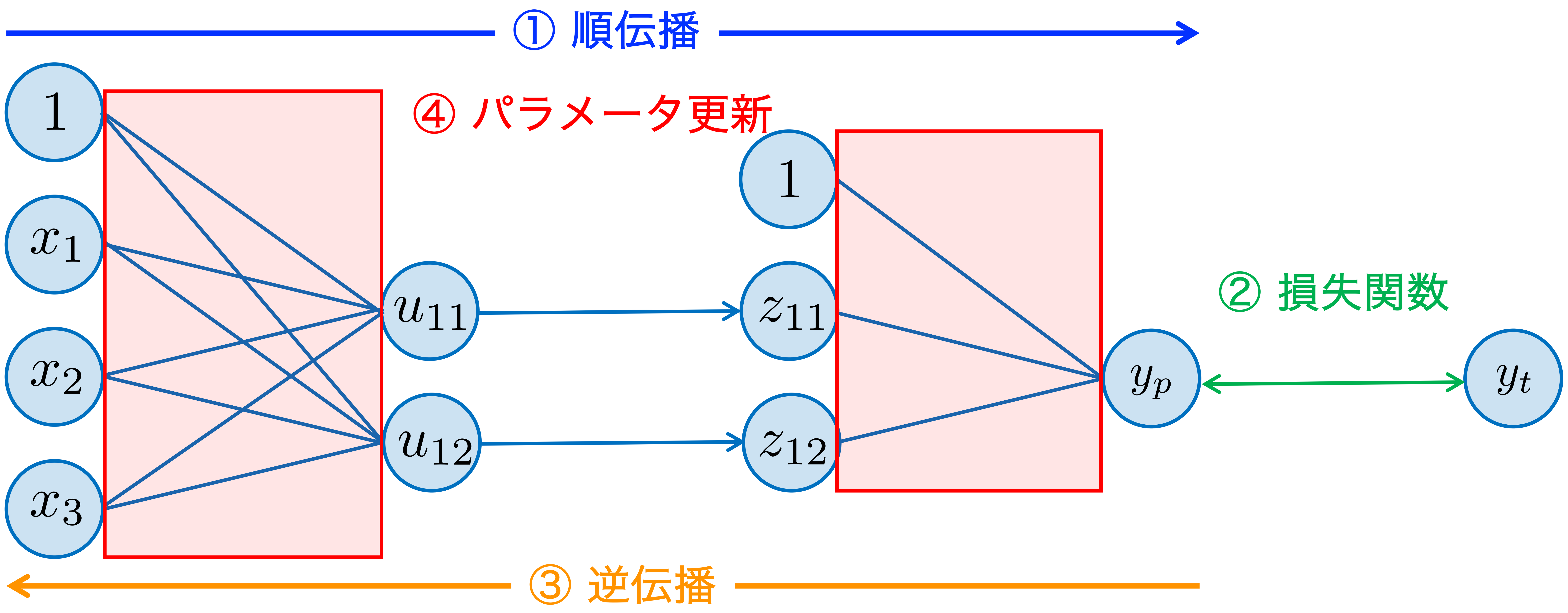
1. ニューラルネットワークとは

■ニューラルネットワークの構造

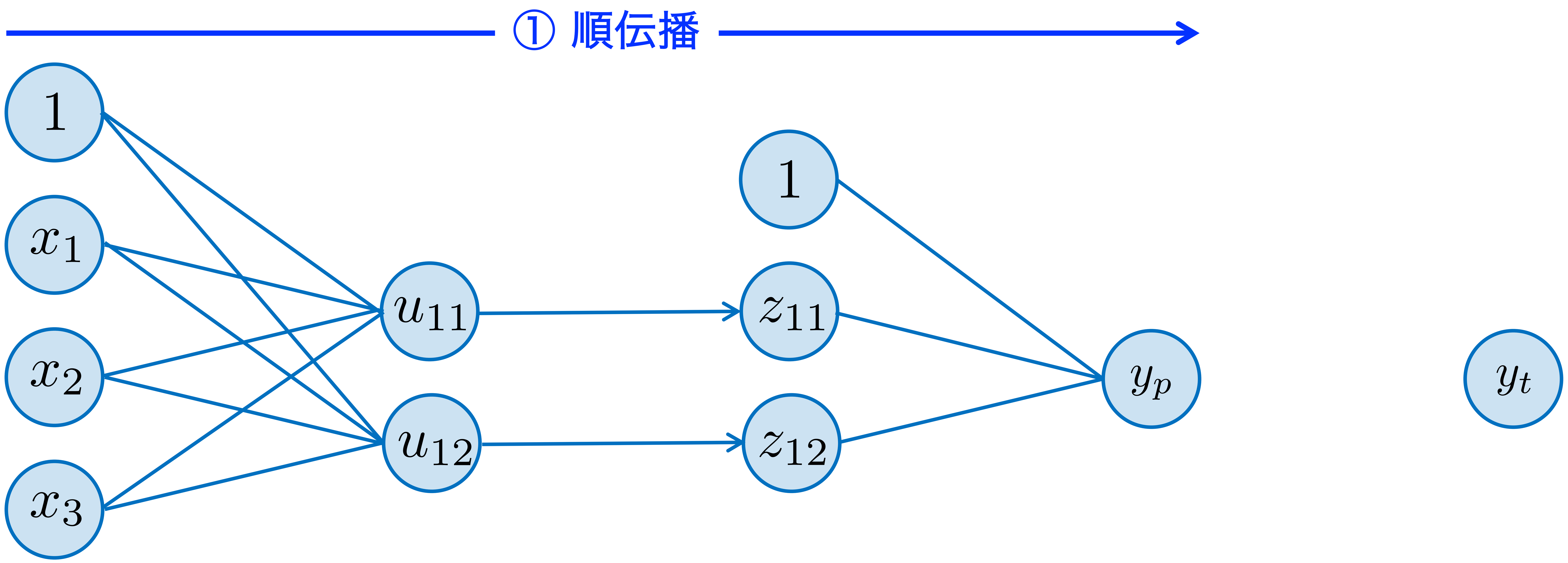


1. ニューラルネットワークとは

■ニューラルネットワークの構造

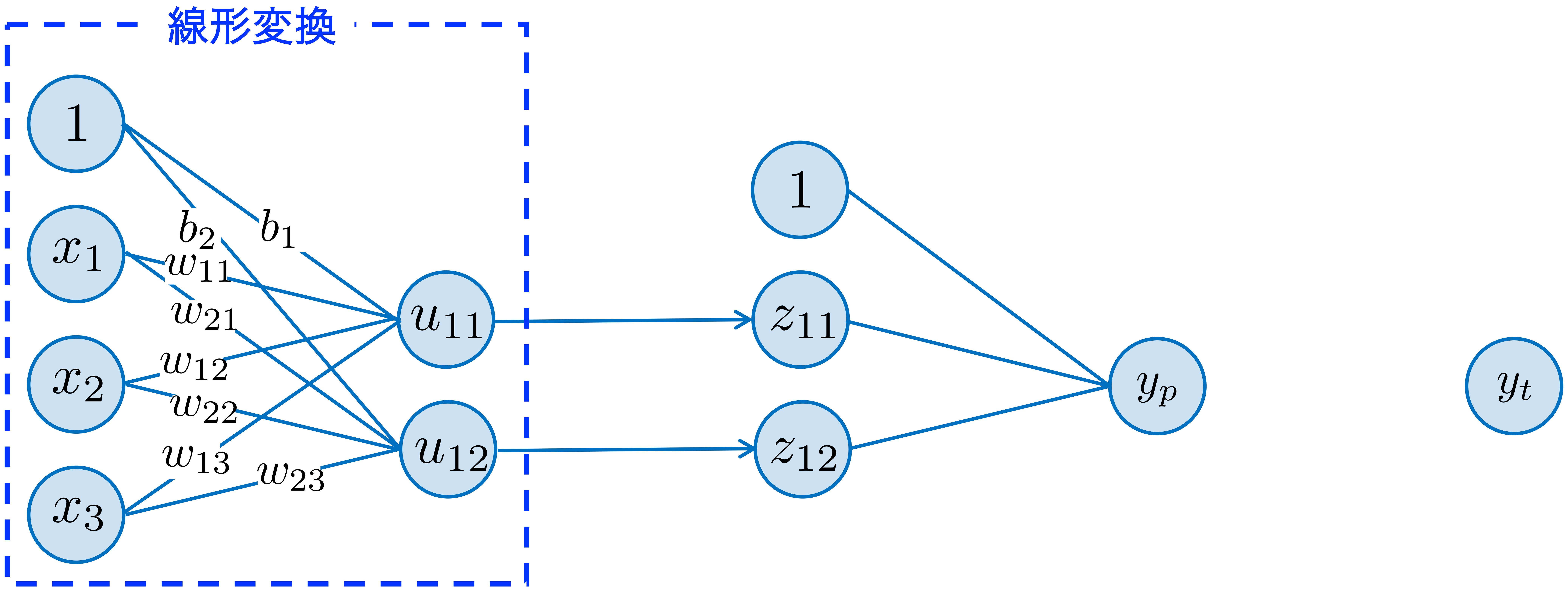


2. 順伝播



2. 順伝播

■線形変換

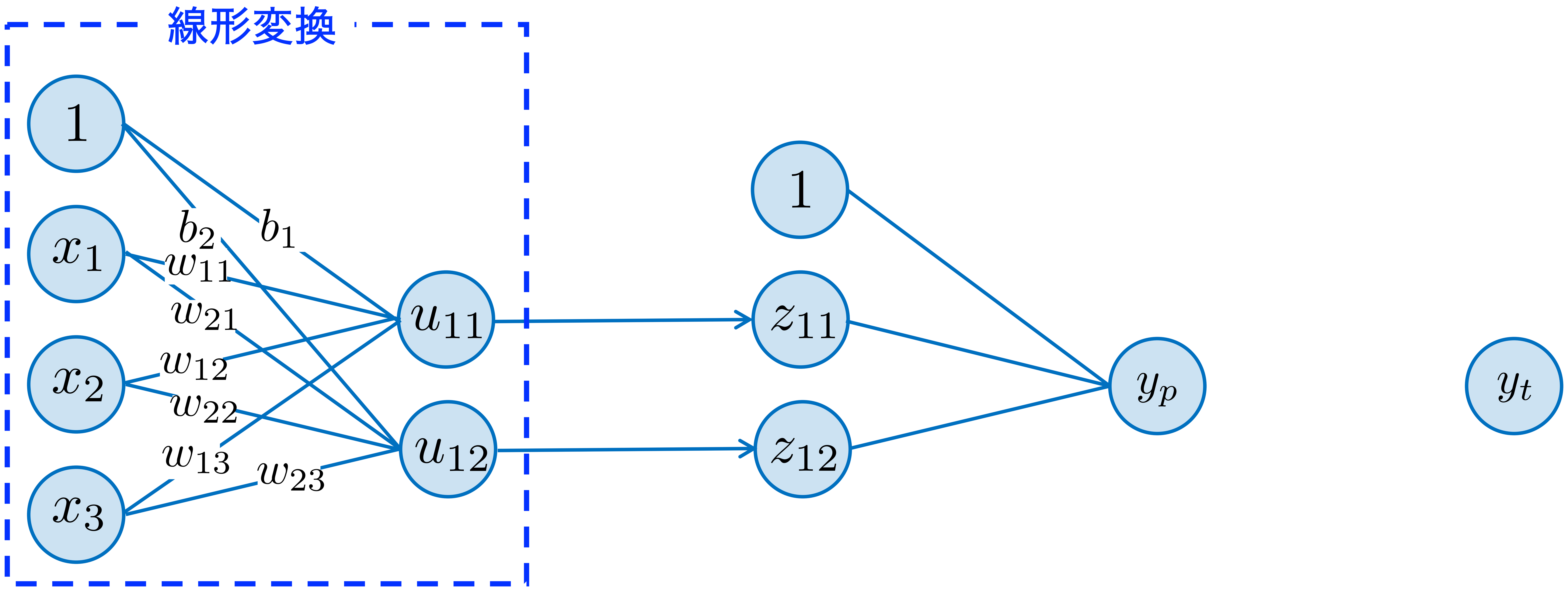


$$u_{11} = w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1$$

$$u_{12} = w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2$$

2. 順伝播

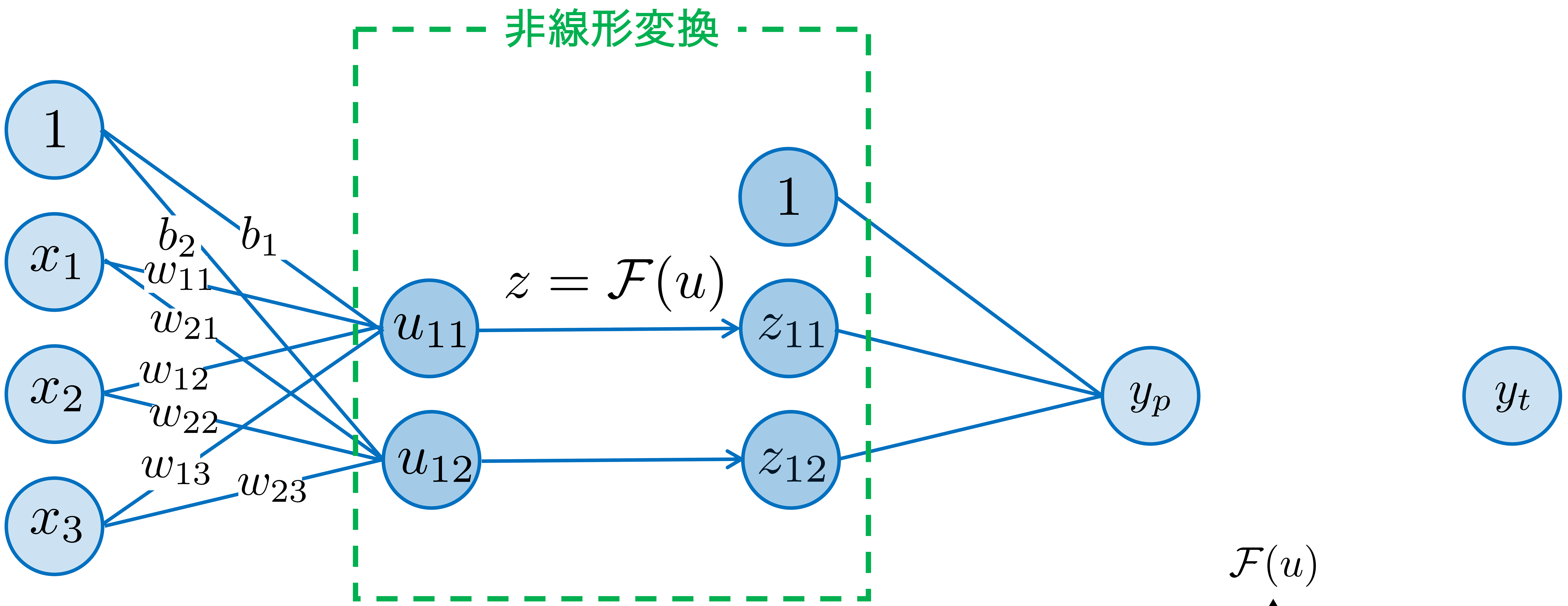
■線形変換



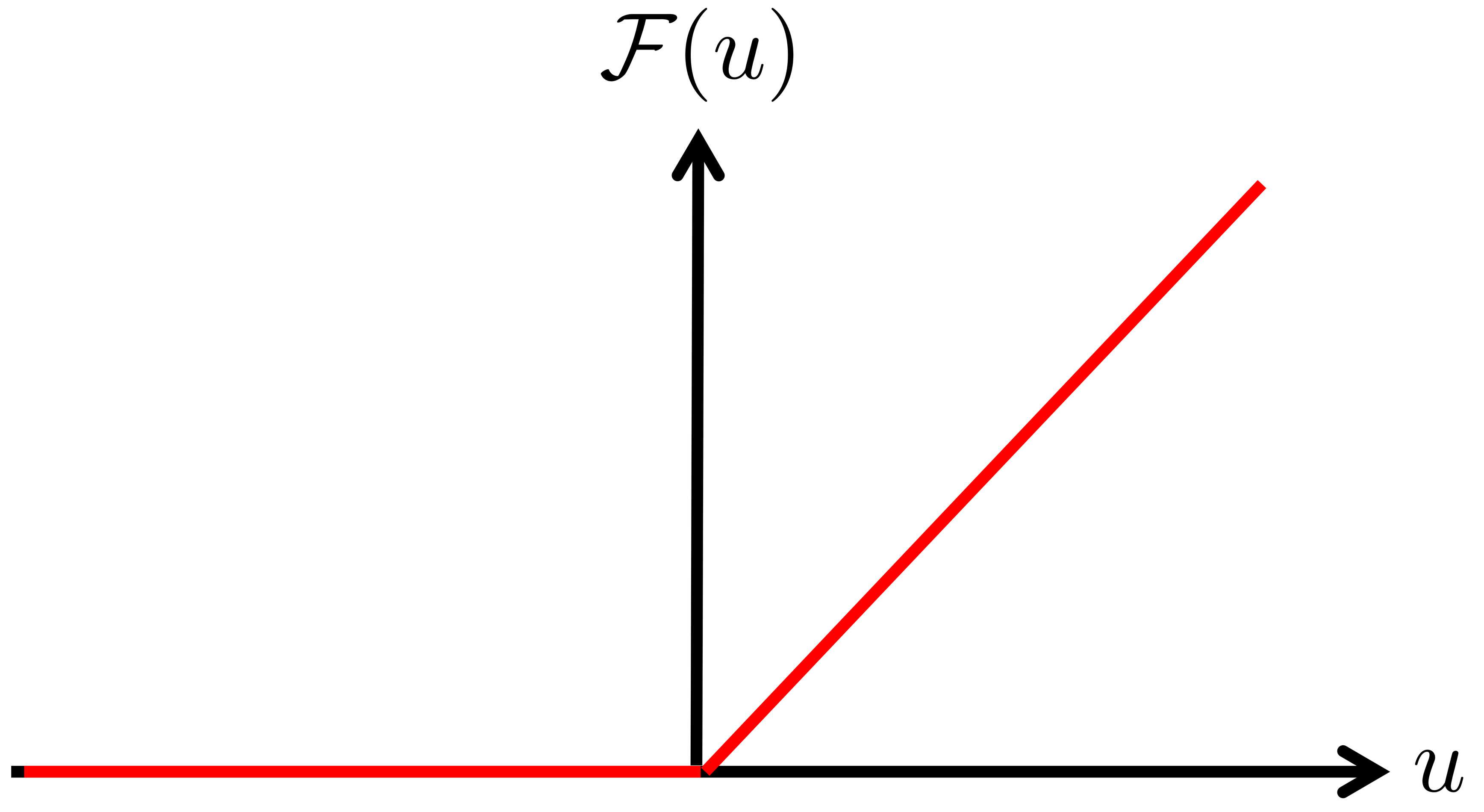
$$\begin{bmatrix} u_{11} \\ u_{12} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad \text{or} \quad \mathbf{u} = \mathbf{w}\mathbf{x} + \mathbf{b}$$

2. 順伝播

■非線形変換



● 非線形変換の例：ReLU関数 $\mathcal{F}(u) = \max(0, u)$

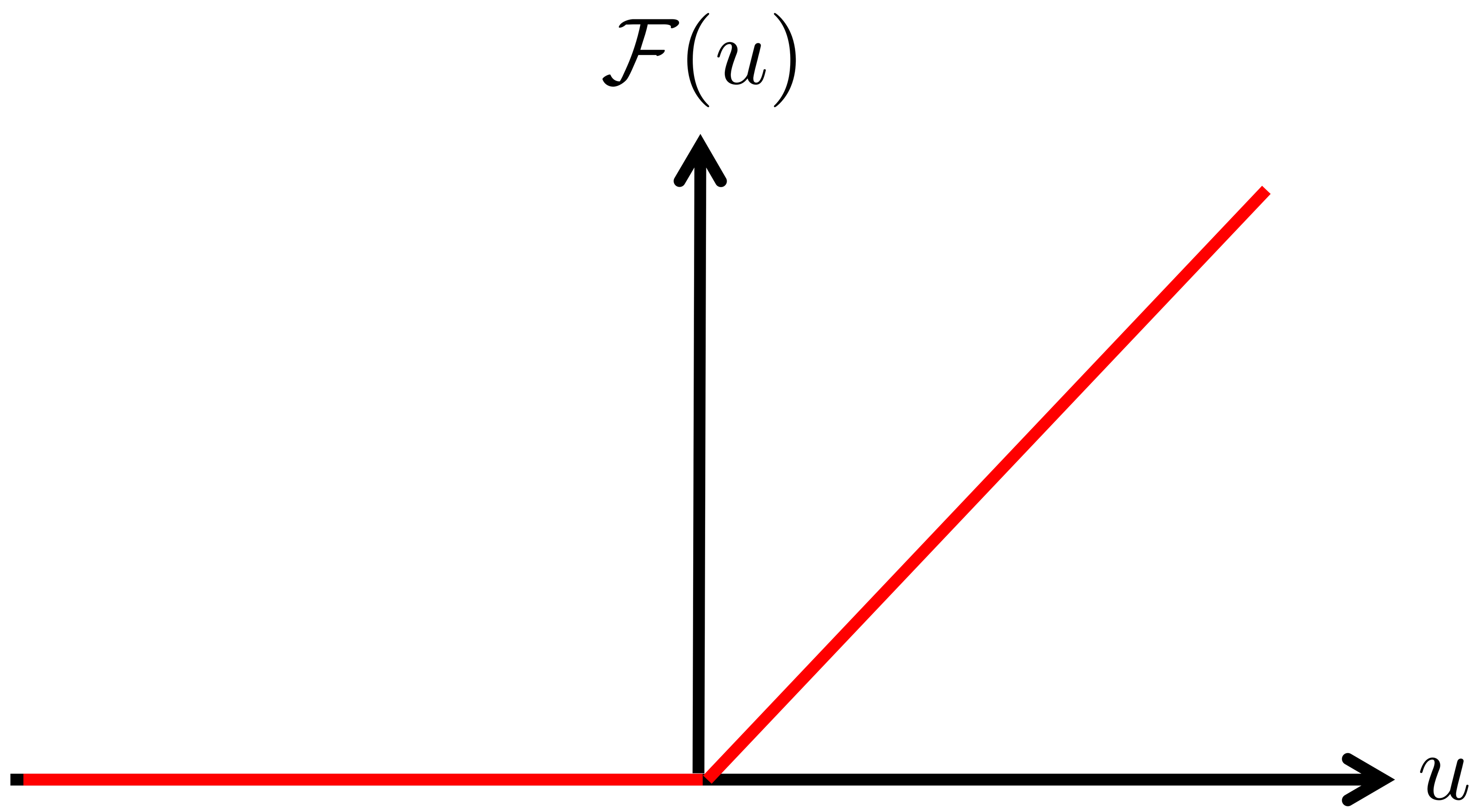


2. 順伝播

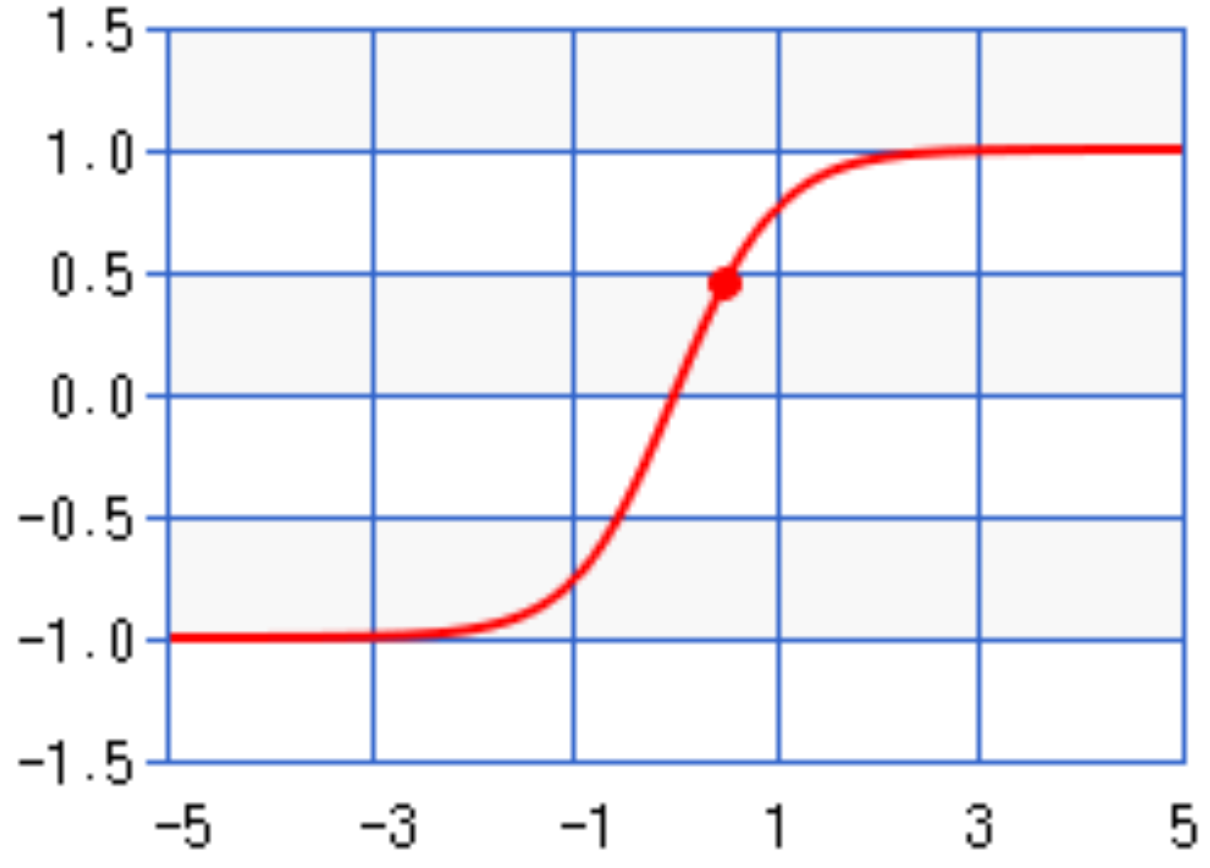
■非線形変換

●非線形変換の例

➢ ReLU関数　$\mathcal{F}(u) = \max(0, u)$

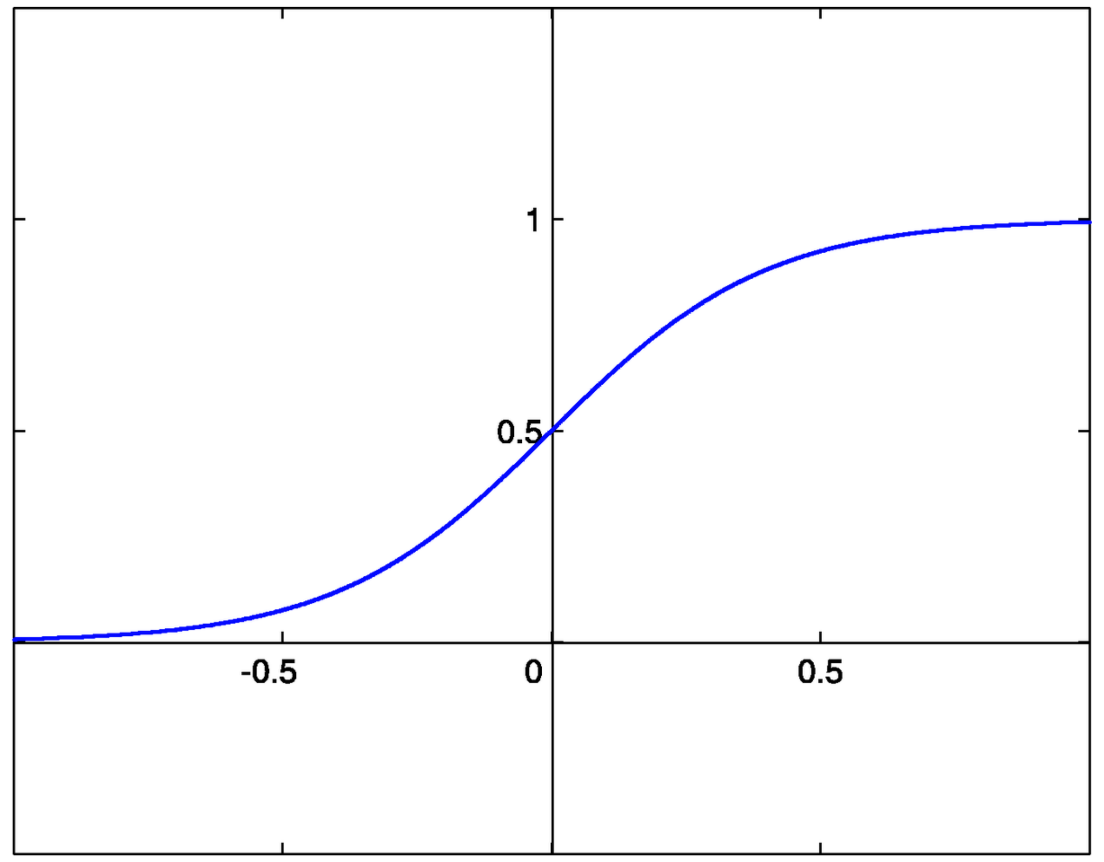


➢ tanh関数　$\mathcal{F}(u) = (e^{u} - e^{-u})/(e^{u} + e^{-u})$



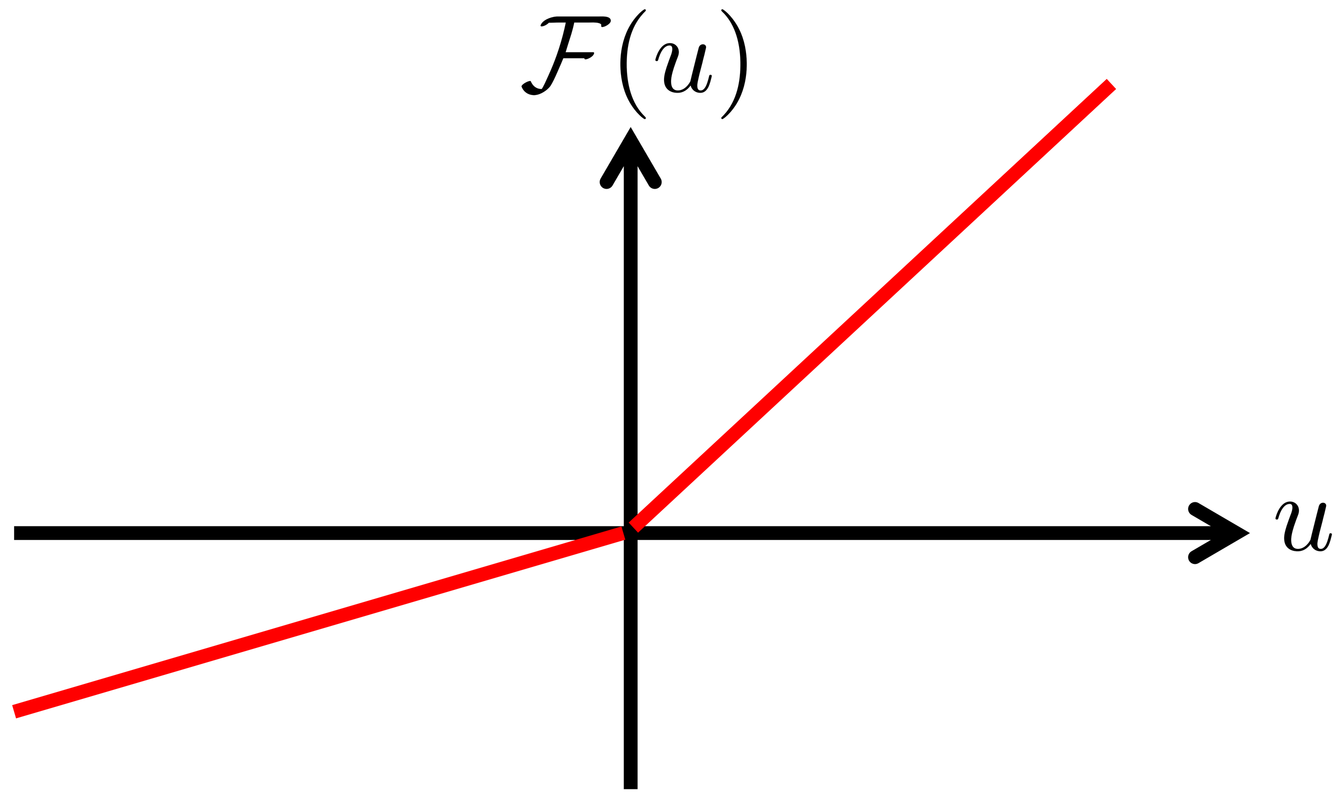
[https://keisan.casio.jp/exec/system/1541125775]

➢ シグモイド関数　$\mathcal{F}(u) = 1/(1 + e^{-u})$



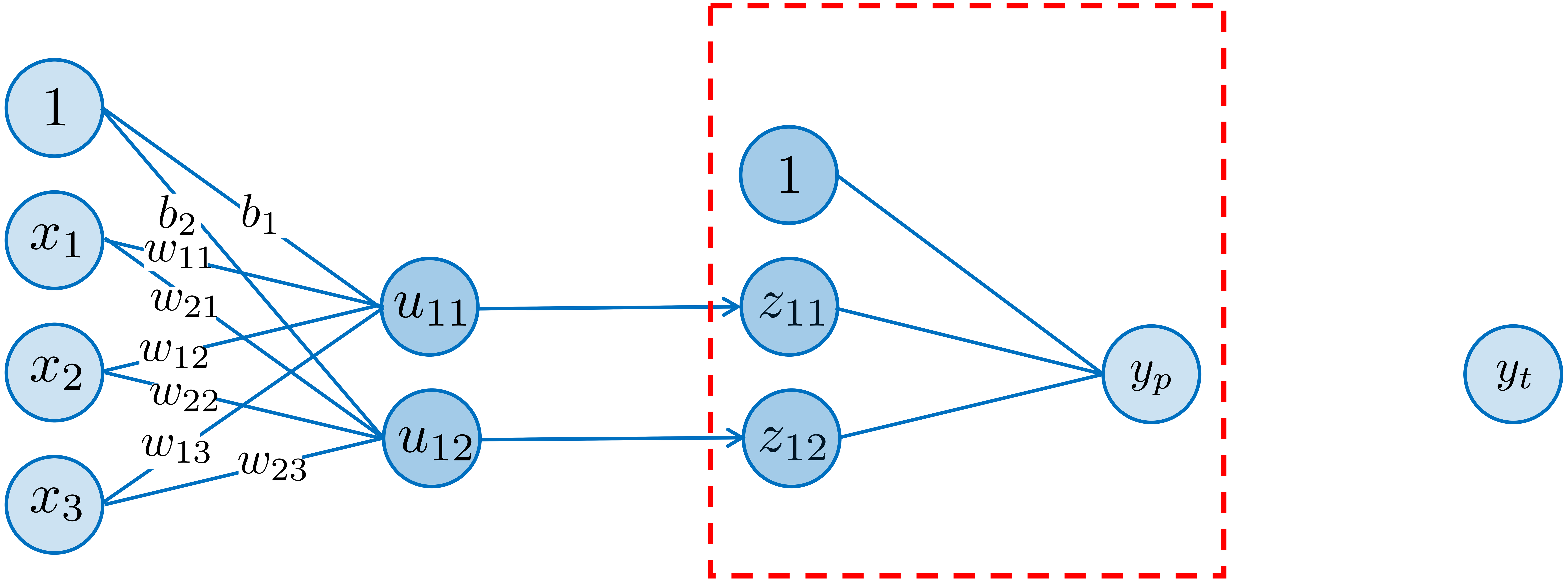
[ウィキペディア]

➢ Leaky ReLU関数



2. 順伝播

■出力層

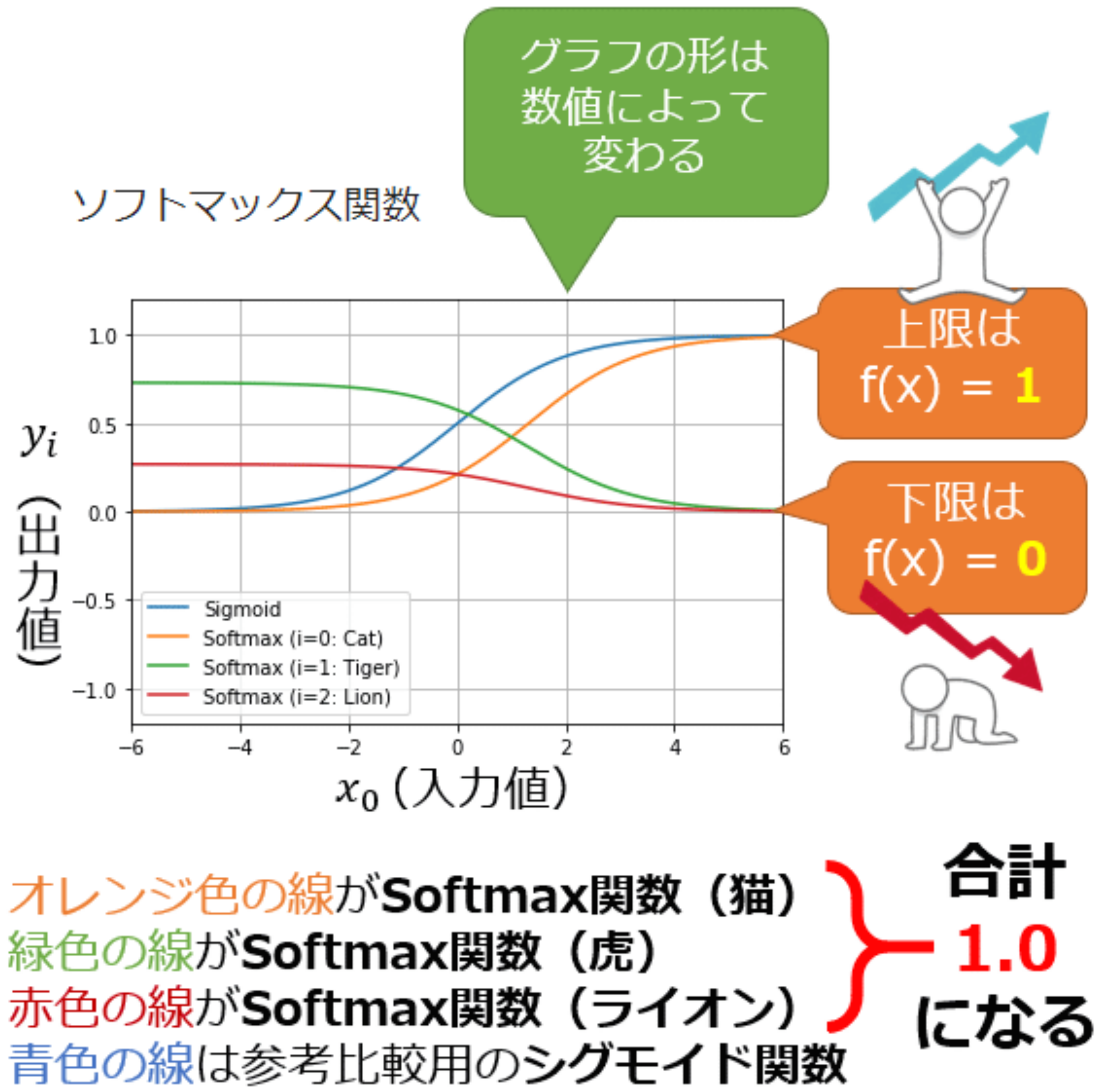


● 線形変換 + 非線形変換(出力層用)

2. 順伝播

■出力層

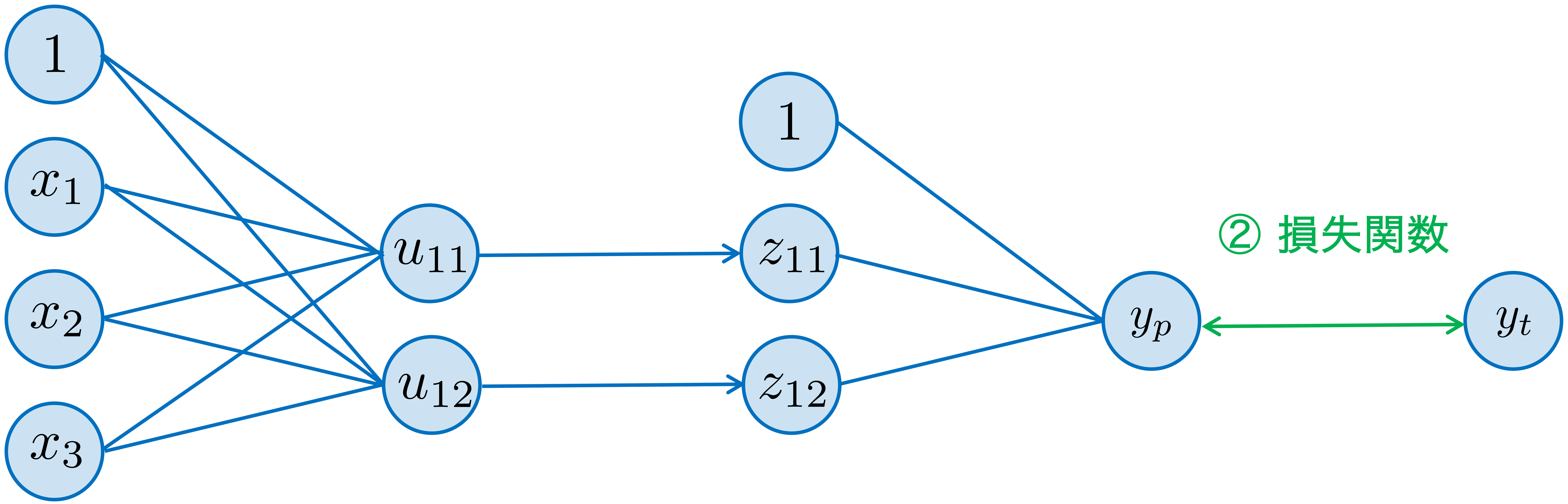
- 非線形変換(出力層用)
  - ➢ 回帰問題：恒等関数 $\mathcal{F}(u) = u$
  - ➢ 分類問題：ソフトマックス関数 $\mathcal{F}(u_j) = \frac{e^{u_j}}{\sum_j e^{u_j}}$



[https://atmarkit.itmedia.co.jp/ait/articles/2004/08/news016.html]

3. 損失関数

■損失関数：予測値と正解値の誤差を表す関数



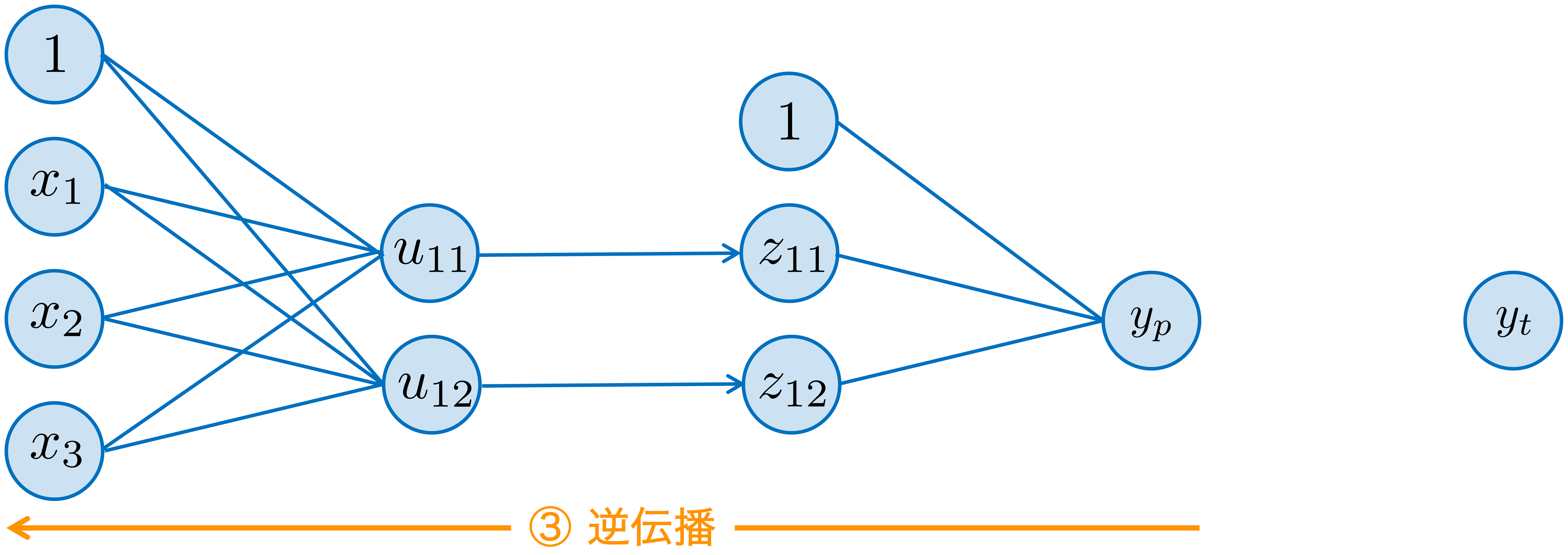
➢ 回帰問題：平均二乗誤差 (MSE) $\mathcal{L} = \frac{1}{N}\sum_{j=1}^{N}(y_p - y_t)^2$

➢ 分類問題：交差エントロピー誤差 $\mathcal{L} = -\sum_{j=1}^{N} y_t^j \log y_p^j$

4. 逆伝播

■なぜ逆伝播する？

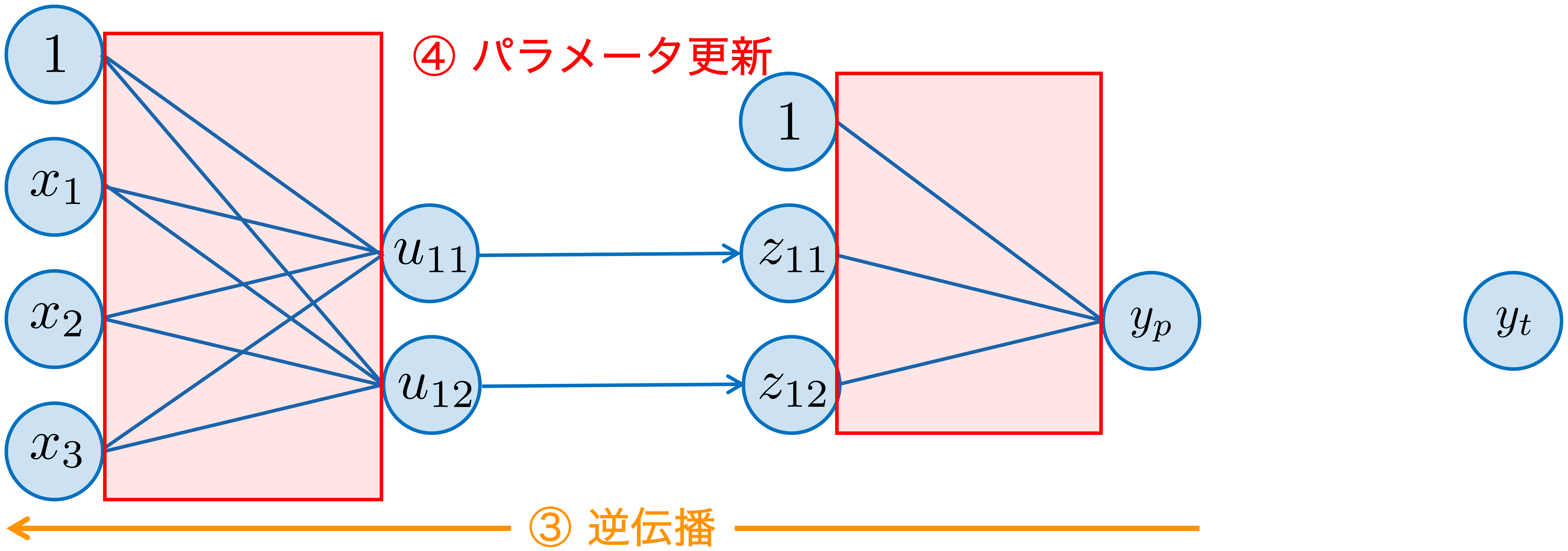
- (誤差)逆伝播：重みパラメータに対する損失関数の勾配を計算する



4. 逆伝播

■なぜ逆伝播する？

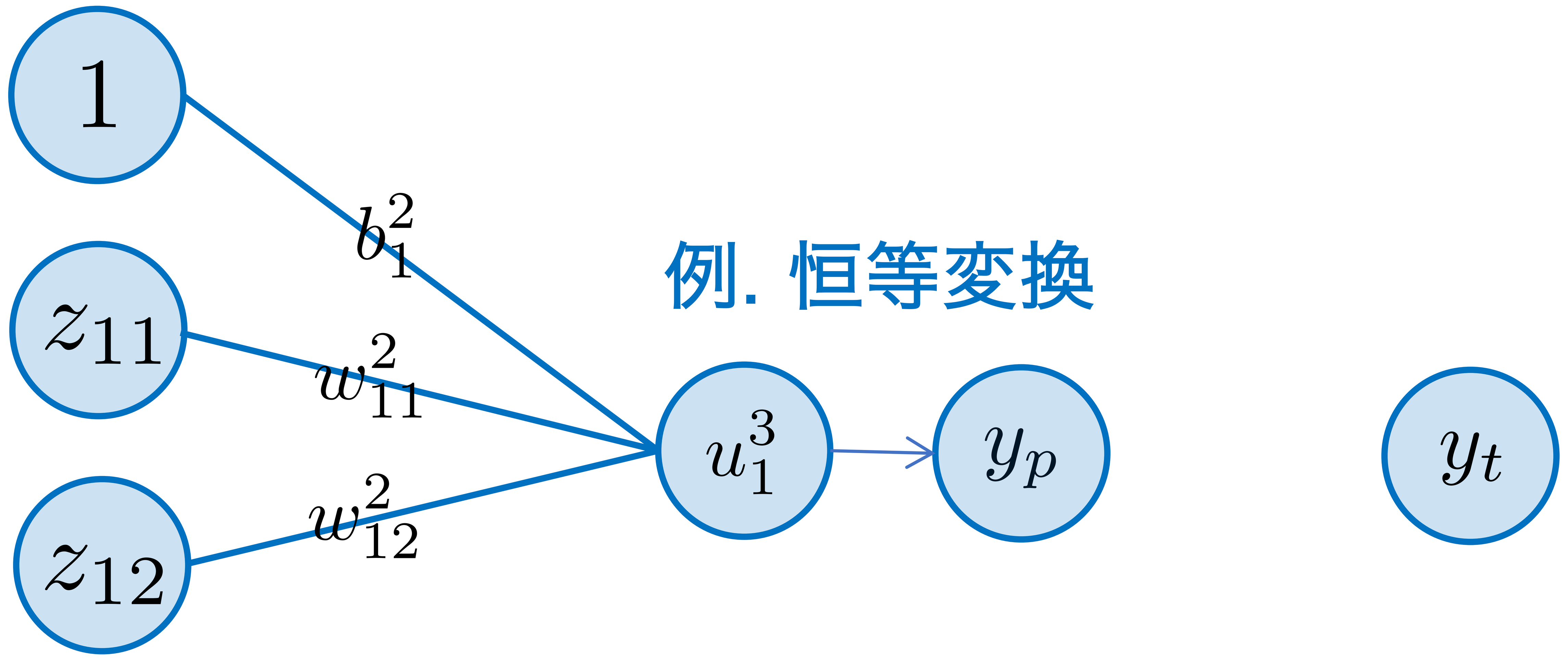
- (誤差)逆伝播：重みパラメータに対する損失関数の勾配を計算する



➢ パラメータ更新に損失関数の勾配が必要！

4. 逆伝播

■出力層



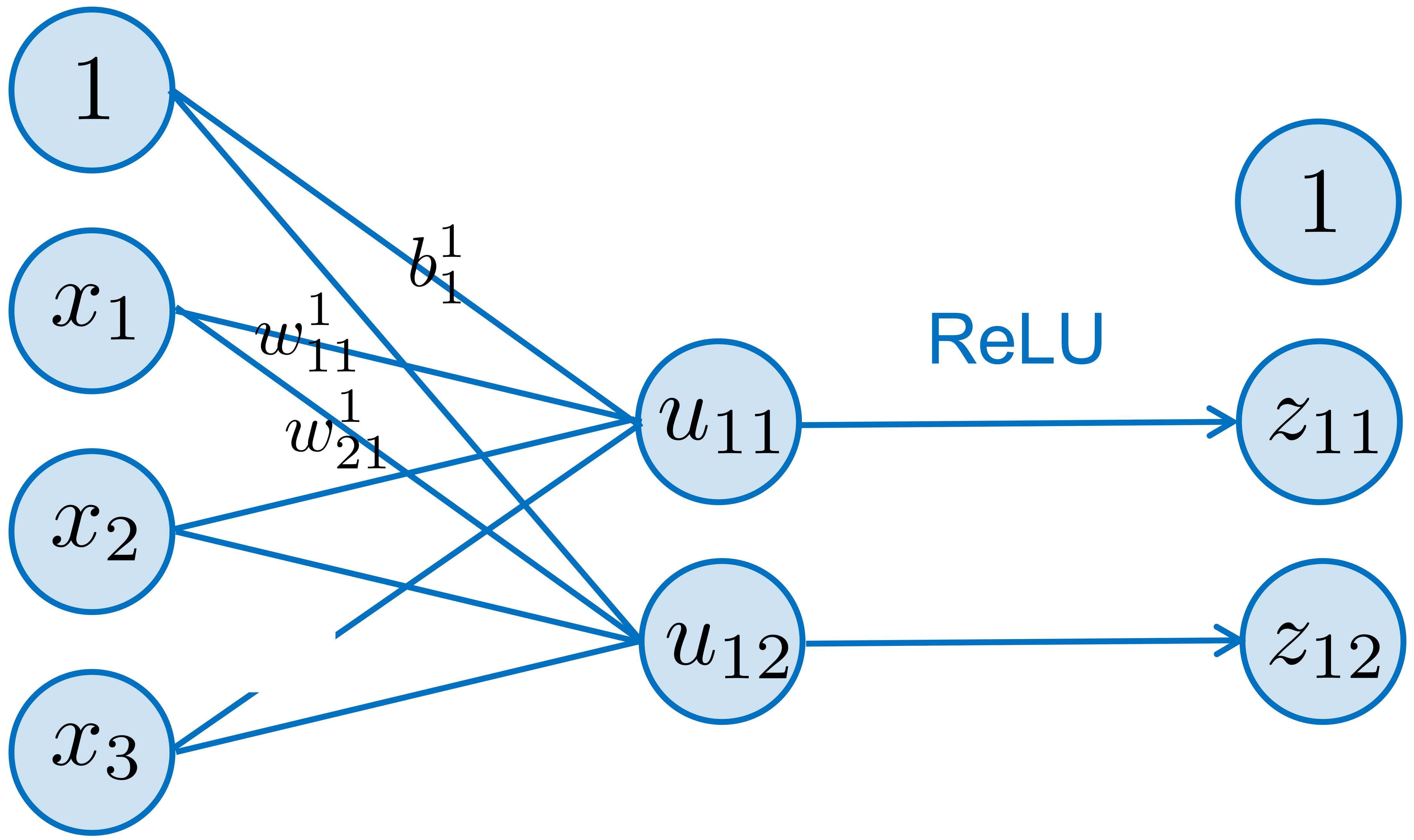
$$\frac{\partial \mathcal{L}}{\partial b_1^2} = \underbrace{\underbrace{\frac{\partial \mathcal{L}}{\partial y_p}}_{\text{back error}} \underbrace{\frac{\partial y_p}{\partial u_1^3}}_{\text{active der}}}_{=\text{delta}} \underbrace{\frac{\partial u_1^3}{\partial b_1^2}}_{=1}$$

$$\frac{\partial \mathcal{L}}{\partial w_{11}^2} = \underbrace{\underbrace{\frac{\partial \mathcal{L}}{\partial y_p}}_{\text{back error}} \underbrace{\frac{\partial y_p}{\partial u_1^3}}_{\text{active der}}}_{=\text{delta}} \underbrace{\frac{\partial u_1^3}{\partial w_{11}^2}}_{=z_{11}}$$

$$\frac{\partial \mathcal{L}}{\partial z_{11}} = \frac{\partial \mathcal{L}}{\partial y_p} \frac{\partial y_p}{\partial u_1^3} \frac{\partial u_1^3}{\partial z_{11}}$$

4. 逆伝播

■隠れ層

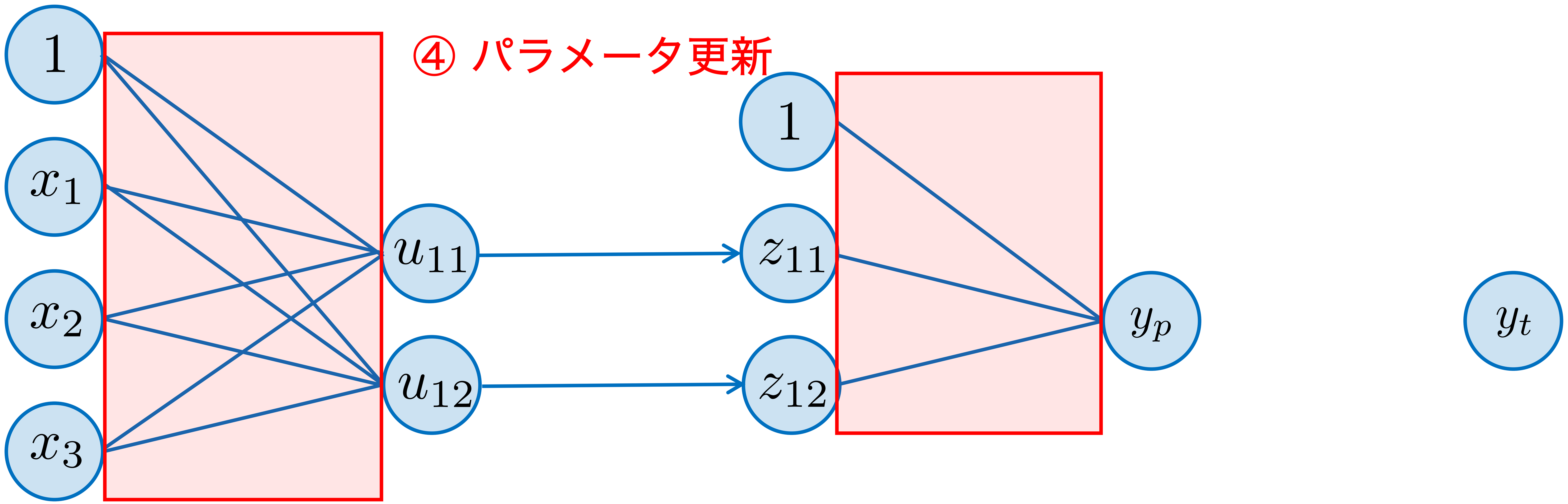


$$\frac{\partial \mathcal{L}}{\partial b_1^1} = \underbrace{\underbrace{\frac{\partial \mathcal{L}}{\partial y_p}\frac{\partial y_p}{\partial u_1^3}\frac{\partial u_1^3}{\partial z_{11}}}_{\text{back error} \; = \frac{\partial \mathcal{L}}{\partial z_{11}}}\underbrace{\frac{\partial z_{11}}{\partial u_{11}}}_{\text{active der}}}_{=\text{delta}}\underbrace{\frac{\partial u_{11}}{\partial b_1^1}}_{=1}$$

$$\frac{\partial \mathcal{L}}{\partial w_{11}^1} = \underbrace{\underbrace{\frac{\partial \mathcal{L}}{\partial y_p}\frac{\partial y_p}{\partial u_1^3}\frac{\partial u_1^3}{\partial z_{11}}}_{\text{back error} \; = \frac{\partial \mathcal{L}}{\partial z_{11}}}\underbrace{\frac{\partial z_{11}}{\partial u_{11}}}_{\text{active der}}}_{=\text{delta}}\underbrace{\frac{\partial u_{11}}{\partial w_{11}^1}}_{=x_1}$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial x_1} &= \frac{\partial \mathcal{L}}{\partial y_p}\frac{\partial y_p}{\partial u_1^3}\left(\frac{\partial u_1^3}{\partial z_{11}}\frac{\partial z_{11}}{\partial u_{11}}\frac{\partial u_{11}}{\partial x_1^1} + \frac{\partial u_1^3}{\partial z_{12}}\frac{\partial z_{12}}{\partial u_{12}}\frac{\partial u_{12}}{\partial x_1^1}\right)\\ &= \frac{\partial \mathcal{L}}{\partial y_p}\frac{\partial y_p}{\partial u_1^3}\sum_j \frac{\partial u_1^3}{\partial z_{1j}}\frac{\partial z_{1j}}{\partial u_{1j}}\frac{\partial u_{1j}}{\partial x_1^1}\end{aligned}$$

5. パラメータ更新

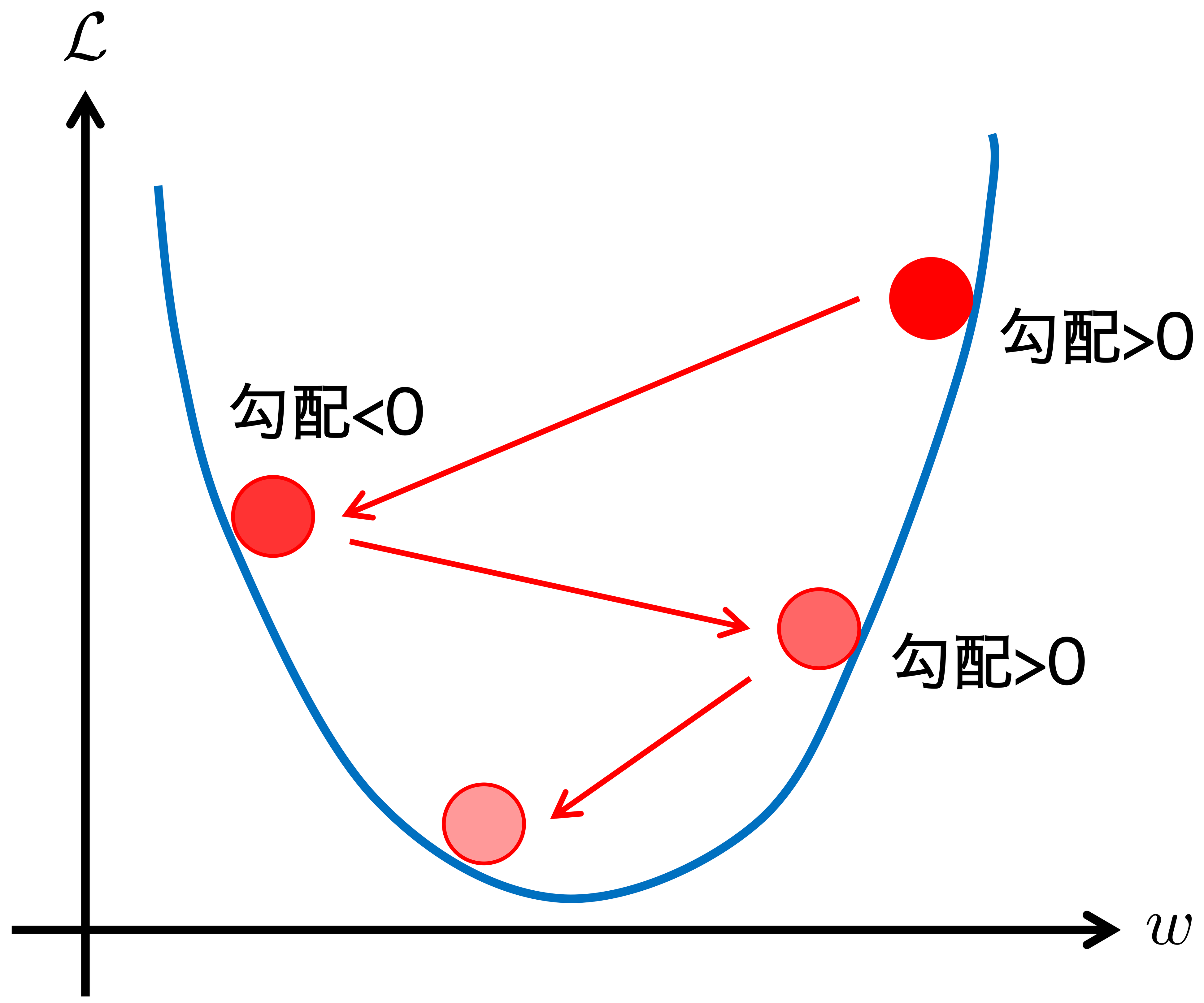


5. パラメータ更新

■勾配降下法

$$w = w - \eta \frac{\partial \mathcal{L}}{\partial w}$$

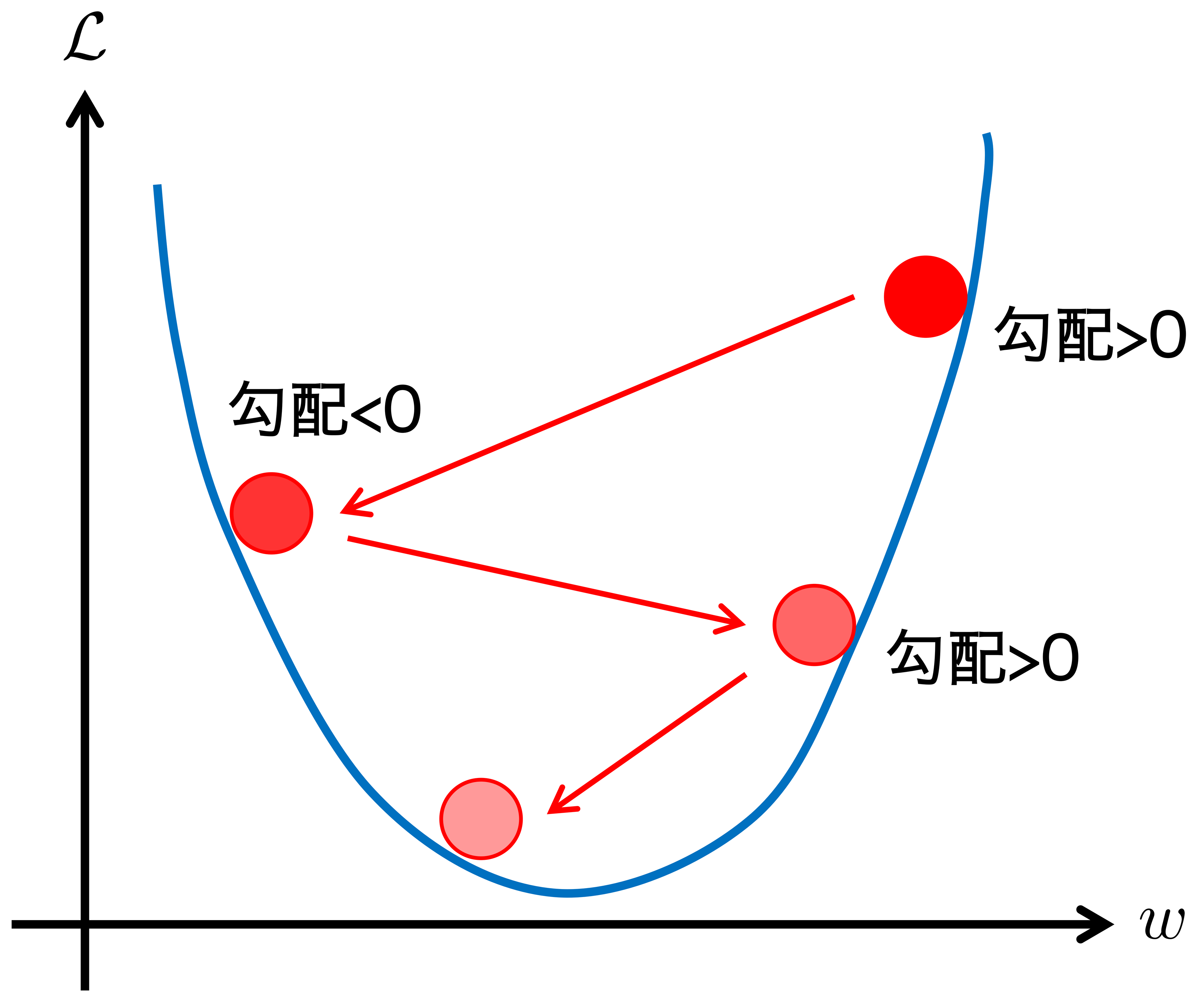
η：学習率



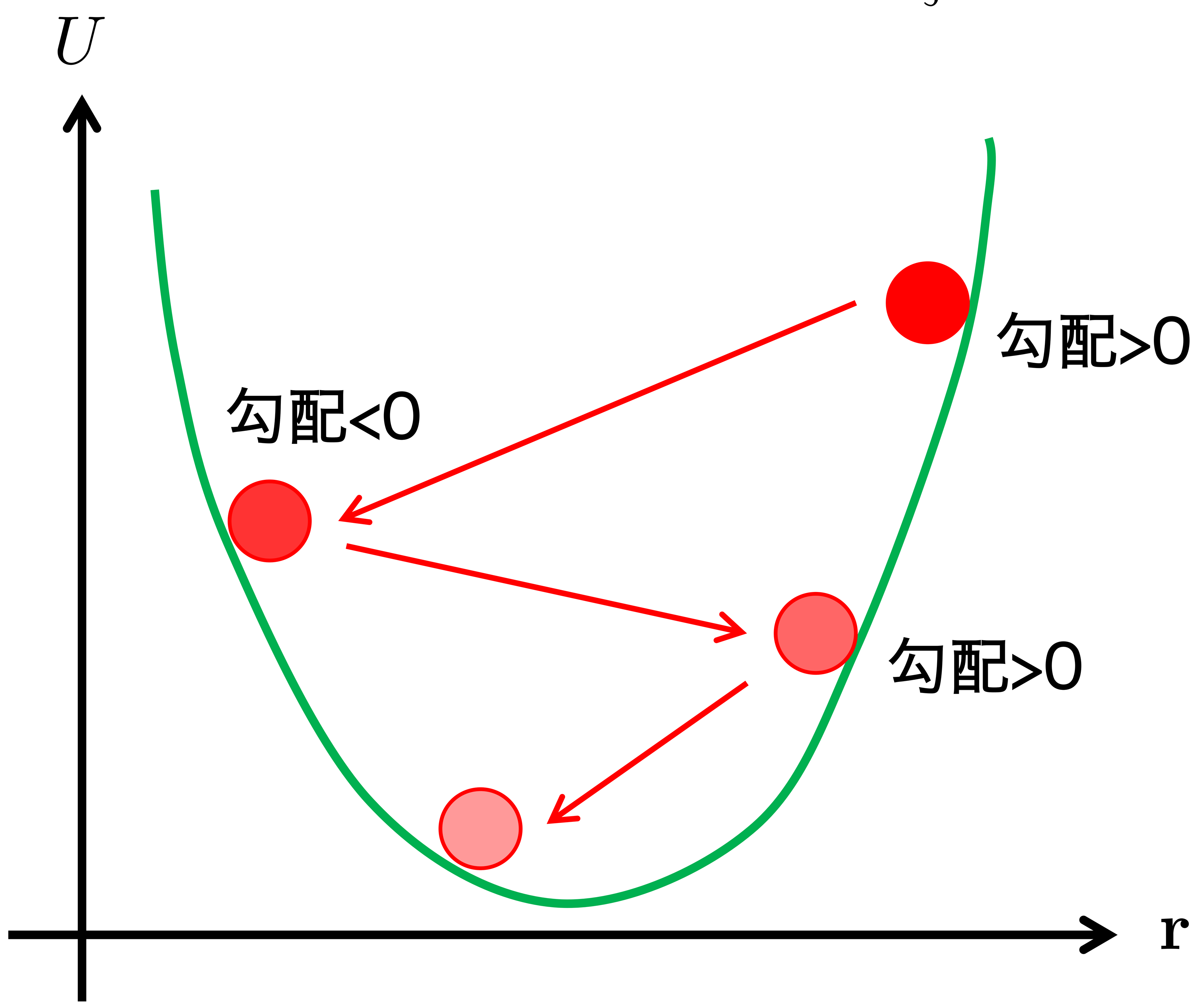
5. パラメータ更新

■勾配降下法

$$w = w - \eta \frac{\partial \mathcal{L}}{\partial w} \qquad \eta：学習率$$



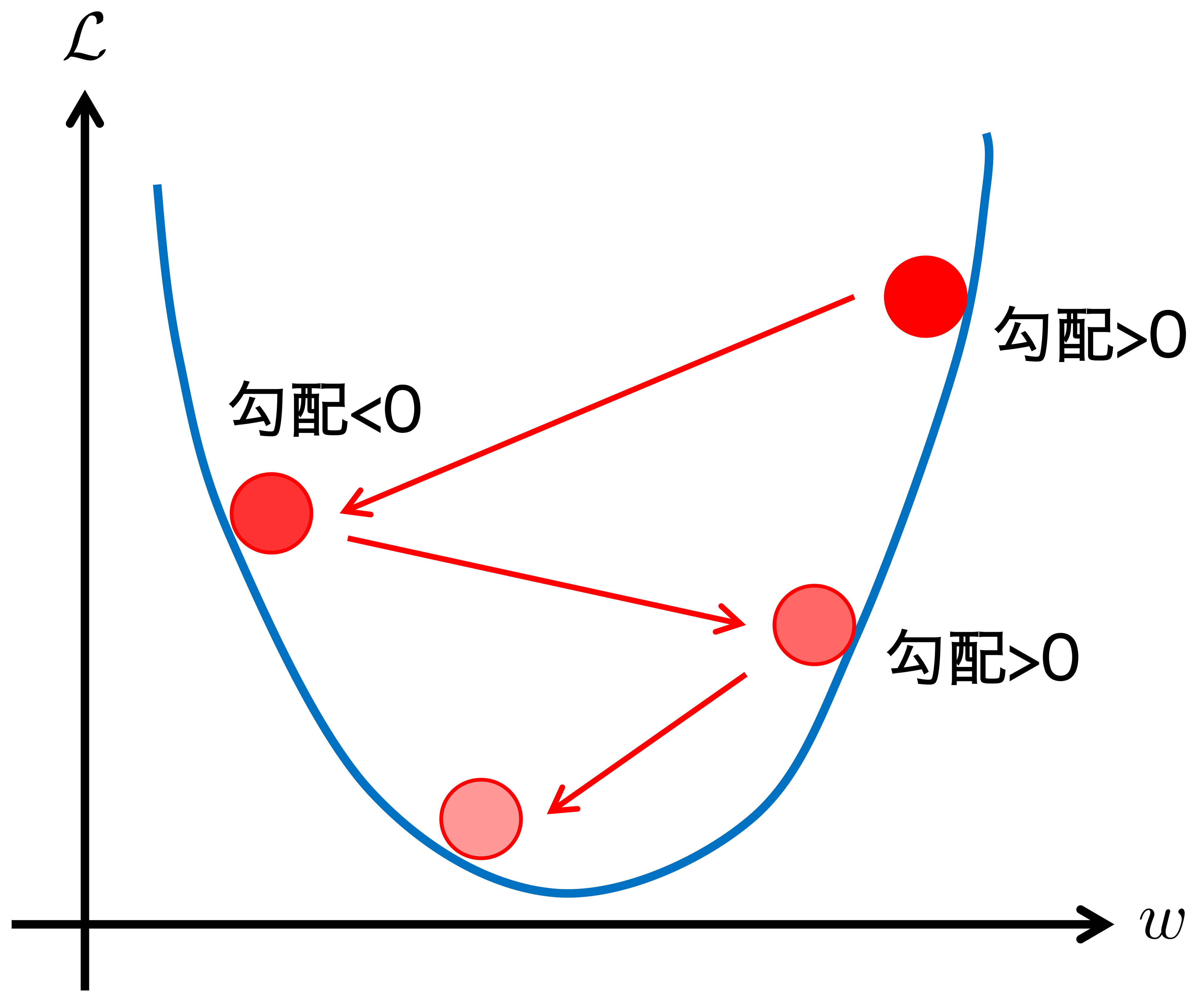
$$\zeta \mathbf{v} = -\nabla U(\mathbf{r}) \quad \rightarrow \quad \mathbf{r} = \mathbf{r} - \frac{\Delta t}{\zeta} \nabla U(\mathbf{r})$$



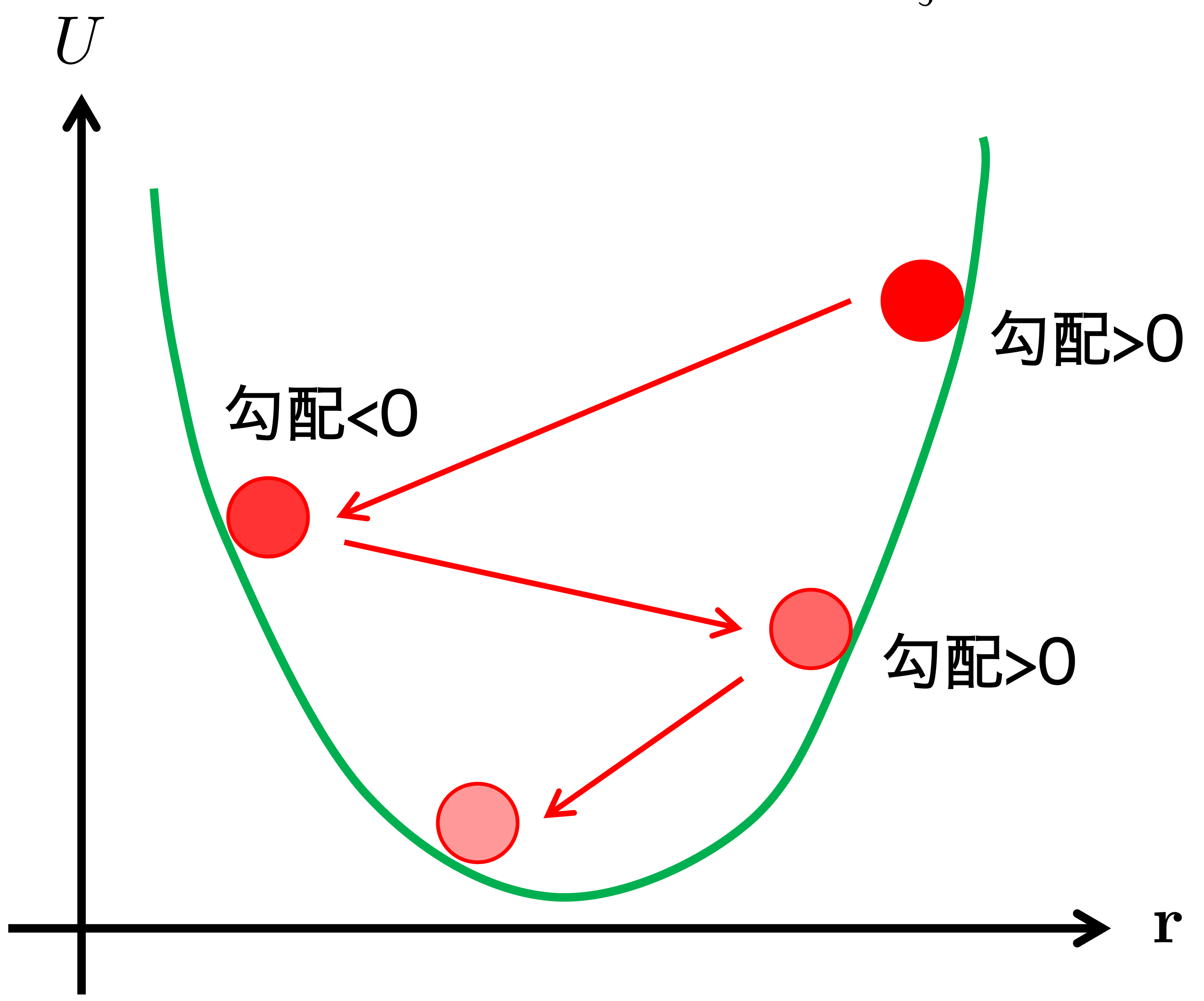
5. パラメータ更新

■勾配降下法

$$w = w - \eta \frac{\partial \mathcal{L}}{\partial w} \qquad \eta\text{：学習率}$$



$$\zeta \mathbf{v} = -\nabla U(\mathbf{r}) \quad \rightarrow \quad \mathbf{r} = \mathbf{r} - \frac{\Delta t}{\zeta} \nabla U(\mathbf{r})$$



● 短所

- ➢ 局所極小値から抜け出すことができない
- ➢ 学習率の調整が難しい

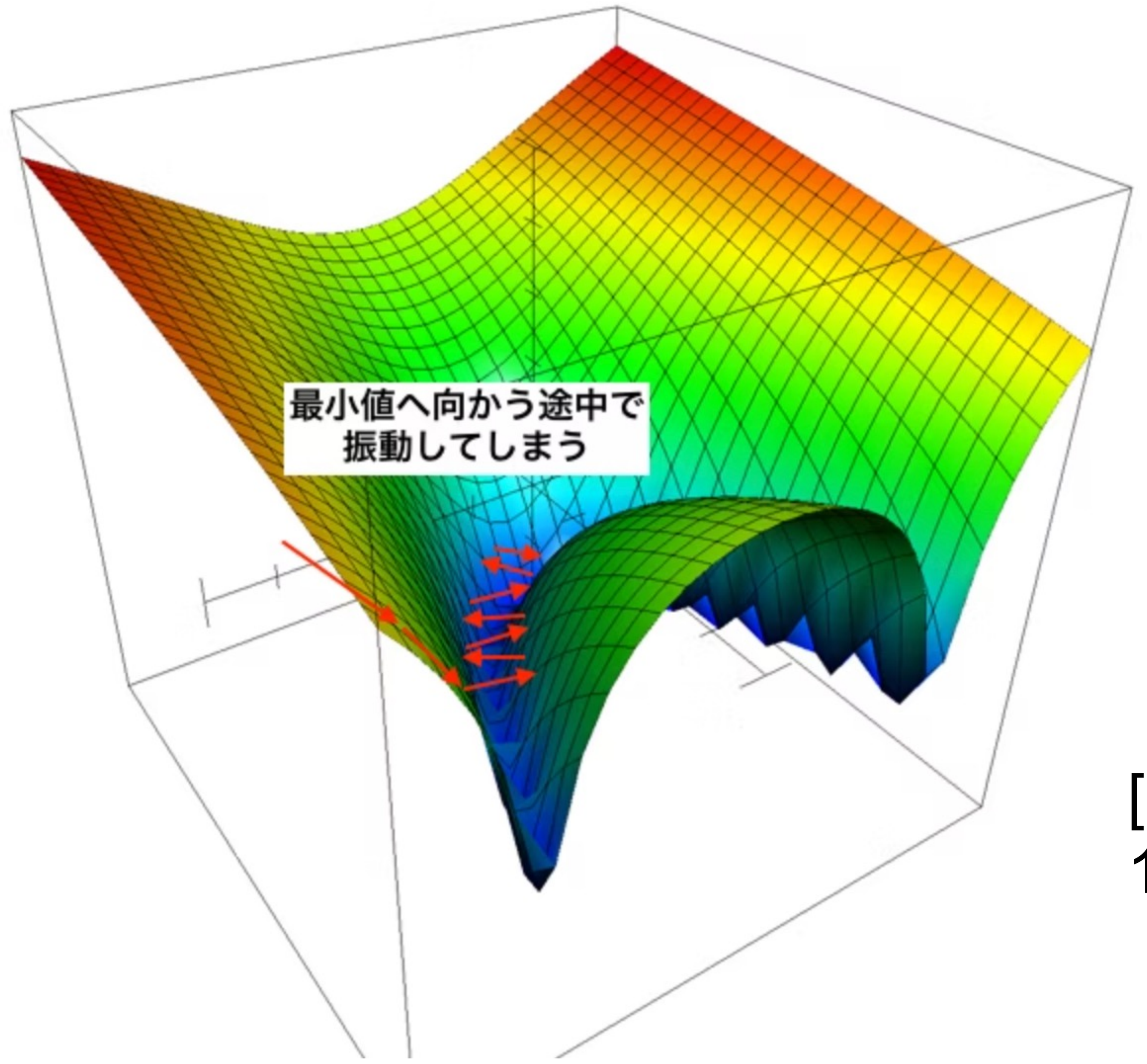
5. パラメータ更新

■確率的勾配降下法 (SGD)

- $\mathbf{w} = \mathbf{w} - \eta \nabla \mathcal{L}_d$
  - ➢ $\mathcal{L}_d$: ランダムに選んだデータのみの損失関数

- SGD : 一部のデータを用いて，パラメータを更新
  - ➢ ランダムにデータを選ぶ (← stochastic)
  - ➢ 局所極小値に陥っても，次のデータの損失関数が大きいので，局所極小値から抜け出せる.
  - ➢ パラメータは再び大きな値に更新され，極小値から抜け出す.

- 短所
  - ➢ 更新量が大きなため，学習が不安定.



Pathological Curvature

[https://qiita.com/omiita/items/1735c1d048fe5f611f80]

5. パラメータ更新

■Adam

- $$\begin{cases} \mathbf{v}_{t+1} = \beta_1 \mathbf{v}_t + (1-\beta_1)\nabla_{\mathbf{w}}\mathcal{L}_d(\mathbf{w}_t) \\ \mathbf{h}_{t+1} = \beta_2 \mathbf{h}_t + (1-\beta_2)\nabla_{\mathbf{w}}\mathcal{L}_d(\mathbf{w}_t) \odot \nabla_{\mathbf{w}}\mathcal{L}_d(\mathbf{w}_t) \\ \hat{\mathbf{v}}_{t+1} = \dfrac{\mathbf{v}_{t+1}}{1-\beta_1^{t+1}} \\ \hat{\mathbf{h}}_{t+1} = \dfrac{\mathbf{h}_{t+1}}{1-\beta_2^{t+1}} \\ \mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \dfrac{\hat{\mathbf{v}}_{t+1}}{\sqrt{\hat{\mathbf{h}}_{t+1}} + \epsilon} \end{cases}$$
- Momentum (慣性を導入) + RMSProp (学習率を調整)
- 現在最も機械学習において用いられている最適化手法

6. まとめ

■ニューラルネットワークによる学習の流れ

① 順伝播

線形変換($\mathbf{u} = \mathbf{w}\mathbf{x} + \mathbf{b}$)+ 非線形変換(ReLU関数など)

↓

② 損失関数

(回帰)平均二乗誤差　(分類)交差エントロピー誤差

↓

③ 逆伝播

損失関数の勾配を計算

↓

④ パラメータ更新

勾配降下法，SGD，Adam

イテレーション

× バッチサイズ = エポック

- サンプルコード：[リンク]
- 問題：サンプルコードは回帰問題を扱っている．では，分類問題のコードを書いてみよう．

7．付録：PyTorch

■PyTorchとは

- Python の機械学習用フレームワーク.
- NumPyのndarrayのような形式のTensorなど便利なライブラリが実装されている.
- 機械学習，特にニューラルネットワークでの学習を行うのに便利！

7．付録：PyTorch

■PyTorchの構成

- `torch`：メインのネームスペースでTensorや様々な数学関数がこのパッケージに含まれる．NumPyの構造を模している．
- `torch.autograd`：自動微分のための関数が含まれる．自動微分のon/offを制御するコンテキストマネージャのenable_grad / no_gradや独自の微分可能関数を定義する際に使用する基底クラスであるFunctionなどが含まれる．
- `torch.nn`：ニューラルネットワークを構築するための様々なデータ構造やレイヤーが定義されている．例えばConvolutionやLSTM，ReLUなどの活性化関数やMSELossなどの損失関数も含まれる．
- `torch.optim`：確率的勾配降下(SGD)を中心としたパラメータ最適化アルゴリズムが実装されている．
- `torch.utils.data`：SGDの繰り返し計算を回す際のミニバッチを作るためのユーティリティ関数が含まれている．

7．付録：PyTorch

■Tensorの生成と変換

- Tensorの生成方法をいくつか紹介する．

```
import torch

# 入れ子のlistを渡して作成
t = torch.tensor([[1, 2], [3, 4]])

# deviceを指定してGPUにTensorを作成
t = torch.tensor([[1, 2], [3, 4]], device = "cuda:0")

# dtypeを指定して倍精度のTensorを作る
t = torch.tensor([[1, 2], [3, 4]], dtype = "torch.float64")

# 0~9の数値で初期化された1次元のTensor
t = torch.arange(0, 10)

# 全値が0の100x100のTensorを作成し、toメソッドでGPUに転送
t = torch.zeros(100,100).to("cuda:0")

# 100x100のランダムなTensor
t = torch.random(100, 100)
```

7. 付録：PyTorch

■Tensorの生成と変換

- TensorはNumPyのndarrayに簡単に変換できる．ただし，GPU上のTensorはそのままでは変換できず，一度CPU上に移す必要がある．

```
import numpy as np
import torch

# numpyメソッドを使用してndarrayに変換
t = torch.tensor([[1, 2], [3, 4]])
nd_arr = t.numpy()

# GPU上のTensorをCPU上に移す
t = torch.tensor([[1, 2], [3, 4]], device = "cuda: 0")
t_cpu = t.to("cpu")
nd_arr = t_cpu.numpy()

# 上記を続けて記入
t = torch.tensor([[1, 2], [3, 4]], device = "cuda: 0")
nd_arr = t.to("cpu").numpy()
```

7．付録：PyTorch

■Tensorのインデクシング操作

- Tensorでもndarrayと同様ににインデクシング操作をサポートしている．スカラー，添字リスト，ByteTensorによるマスク配列での指定が可能である．

```
import torch

t = torch.tensor([[1, 2, 3], [4, 5, 6]])

# スカラーの添字で指定
t[0, 2] # tensor(3)

# スライスで指定
t[:, :2] # tensor([[1, 2], [4, 5]])

# 添字のリストで指定
t[:, [1, 2]] # tensor([[2, 3], [5, 6]])
```

7．付録：PyTorch

■Tensorのインデクシング操作

```
# マスク配列を使用して3より大きい部分のみ選択
t[t > 3] # tensor([4, 5, 6])

# [0, 1]要素を100に置換
t[0, 1] = 100 # tensor([[1, 100, 3], [4, 5, 6]])

# スライスを使用した一括代入
t[:, 1] = 200 # tensor([[1, 200, 3], [4, 200, 6]])

# マスク配列を使用して特定条件の要素のみ置換
t[t > 10] = 20 # tensor([[1, 2, 3], [4, 5, 6]])
```

7. 付録：PyTorch

■Tensorの四則演算

- Tensorは四則演算や数学関数，線形代数計算などを行うことができる．特に行列積や特異値分解などの線形代数計算はGPUが使用可能ということもあって大規模データの場合にはNumPy/SciPyを使用するよりも良いパフォーマンスが多々ある．
- Tensorの四則演算は同じ型(shape) 同士の四則演算が可能である．

```
import torch

# ベクトル
v = torch.tensor([1, 2, 3])
w = torch.tensor([4, 5, 6])

# 行列
m = torch.tensor([[0, 1, 2], [10, 20, 30]])
n = torch.tensor([[3, 4, 5], [40, 50, 60]])

# ベクトル - スカラー
v_pl = v + 10 # tensor([11, 12, 13])
v_mi = v - 10 # tensor([-9, -8, -7])
v_mu = v * 10 # tensor([10, 20, 30])
v_di = v / 10 # tensor([0, 0, 0])
```

7. 付録：PyTorch

■Tensorの演算

```
# ベクトル - ベクトル
v_pl = v + w # tensor([5, 7, 9])
v_mi = v - w # tensor([-3, -3, -3])
v_mu = v * w # tensor([4, 10, 18])
v_di = v / w # tensor([0, 0, 0])

# 行列とベクトル
v_pl = m + v # tensor([[ 1, 3, 5], [11, 22, 33]])
v_mi = m - v # tensor([[-1, -1, -1], [ 9, 18, 27]])
v_mu = m * v # tensor([[ 0, 2, 6], [10, 40, 90]])
v_di = m / v # tensor([[ 0, 0, 0], [10, 10, 10]])

# 行列と行列
v_pl = m + n # tensor([[3, 5, 7], [50, 70, 90]])
v_mi = m - n # tensor([[-3, -3, -3], [-30, -30, -30]])
v_mu = m * n # tensor([[0, 4, 10], [400, 1000, 1800]])
v_di = m / n # tensor([[0, 0, 0], [0, 0, 0]])
```

7. 付録：PyTorch

■Tensorの演算

- PyTorchではTensorに対して様々な数学関数を用意している．
  - `abs`：絶対値
  - `sin`：正弦
  - `cos`：余弦
  - `exp`：指数関数
  - `log`：対数関数
  - `sqrt`：平方根
  - `sum`：tensor内の値の合計
  - `max`：tensor内の値の最大値
  - `min`：tensor内の値の最小値
  - `mean`：tensor内の値の平均値
  - `std`：tensor内の値の標準偏差

7. 付録：PyTorch

■Tensorの演算

```
import torch

t = torch.tensor([[0.1, -0.2, 0.3], [0.4, -0.5, -0.6]])

#絶対値
t_abs = torch.abs(t)
print("絶対値 : ", t_abs)

#合計
t_sum = torch.sum(t)
print("合計 : ", t_sum)

>>>絶対値 : tensor([[0.1000, 0.2000, 0.3000],
[0.4000, 0.5000, 0.6000]])
合計 : tensor(-0.5000)
```

7. 付録：PyTorch

■Tensorの演算

- 線形代数の演算子を用いることもできる.
  - ➢ dot：ベクトルの内積
  - ➢ mv：行列とベクトルの積
  - ➢ mm：行列と行列の積
  - ➢ matmul：引数の種類によって自動的にdot，mv，mmを選択して実行
  - ➢ gesv：LU分解による連立方程式の解
  - ➢ eig，symeig：固有値分解．symeigは対称行列用
  - ➢ svd：特異値分解

7．付録：PyTorch

■Tensorの演算

```
import torch

m = torch.randn(100, 10) # 100x10の行列テストデータを作成
v = torch.randn(10)

# 内積
torch.dot(v, v)

# 行列とベクトルの積
torch.mv(m, v)

# 行列積
torch.mm(m.t(), m)

# 特異値分解
u, s, v = torch.svd(m)
```

7. 付録：PyTorch

■その他よく使う関数

- その他PyTorchでよく用いる関数をまとめておく.
  - ➢ view : tensorの次元を変更する
  - ➢ car, stack : tensor同士を結合する
  - ➢ transpose : 次元を入れ替える

```
import torch

t = torch.tensor([[0.1, -0.2, 0.3], [0.4, -0.5, -0.6]])
print("original tensor shape : ", t.shape)
# テンソルの形状を変更
new_t = t.view(3, 2)
print("new tensor : ", new_t)
print("new tensor shape : ", new_t.shape)

>>>original tensor shape : torch.Size([2, 3])
new tensor : tensor([[ 0.1000, -0.2000],
                     [ 0.3000, 0.4000],
                     [-0.5000, -0.6000]])
new tensor shape : torch.Size([3, 2])
```