

知能工学特別講義 第2講

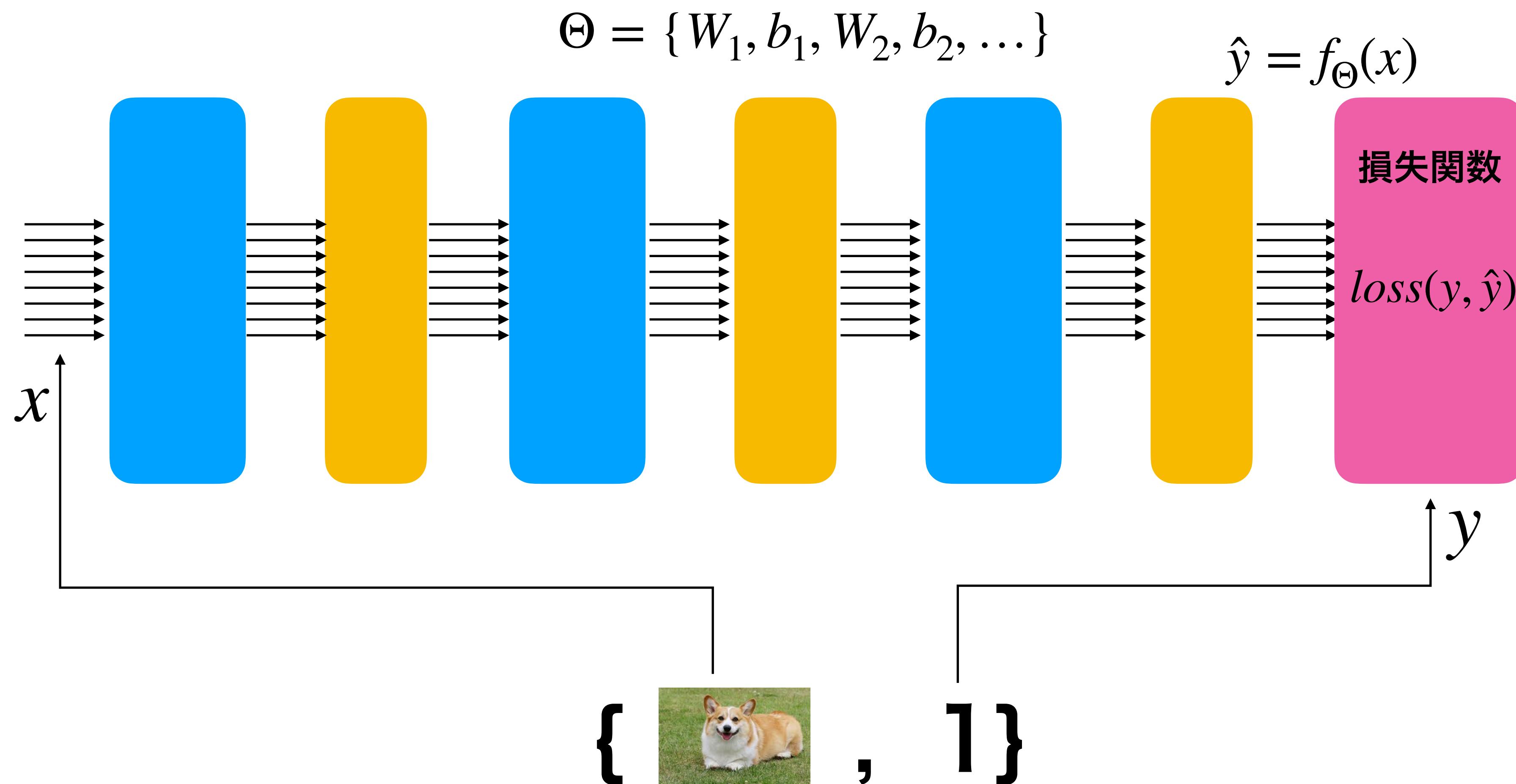
担当：和田山 正

名古屋工業大学

本講義の内容

- 深層学習の概要(復習)
- ニューラルネットワークの歴史をたどる
- 深層ニューラルネットワークの学習プロセス
- PyTorchを使う(AND関数学習)

深層ニューラルネットワークの訓練(学習)



訓練・学習過程では、損失関数值を最小化するように
学習パラメータを変更する

学習プロセス

$$h' = W h + b$$

中身は動かしてよい・うまく
チューンアップしたい！

二乗誤差関数を最小化するようにパラメータを動かせばよい

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

ニューラルネットワークの特徴

- 層構造を持つパラメトリック非線形関数モデル: 学習可能であり、高い表現能力を持つ
- 各層は行列ベクトル積計算（アフィン変換）と非線形関数の要素ごとの適用からなる
- 適切な学習（訓練）プロセスが必要
- 学習プロセスでは、大量の訓練データが必要
- 学習プロセスでは、確率的勾配法を利用してパラメータを調節する

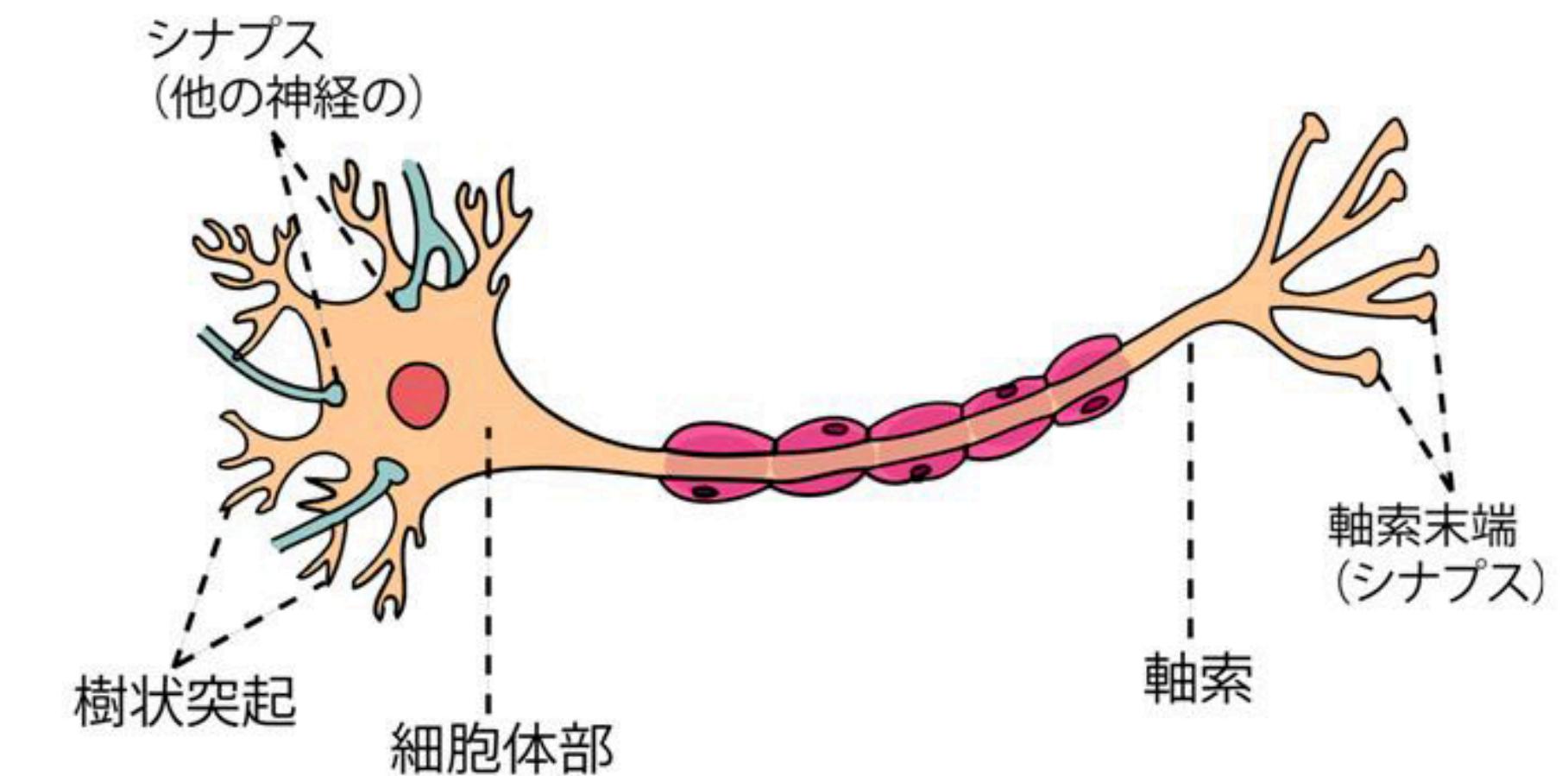
ニューラルネットワークの歴史をたどる



形式ニューロンの登場

- ・脳神経学者のマカロックと數学者のピッツは、神経細胞を模した「形式ニューロン」を導入した。(McCulloch, W. and Pitts, W. (1943).)
- ・現実のニューロンの振る舞いを簡略化して得られた
- ・形式ニューロンは、現実のニューロンが「発火するか(1)」または「発火しないか(0)」という2状態を持つとしてモデル化を行っている。

ニューロン（脳の神経細胞）

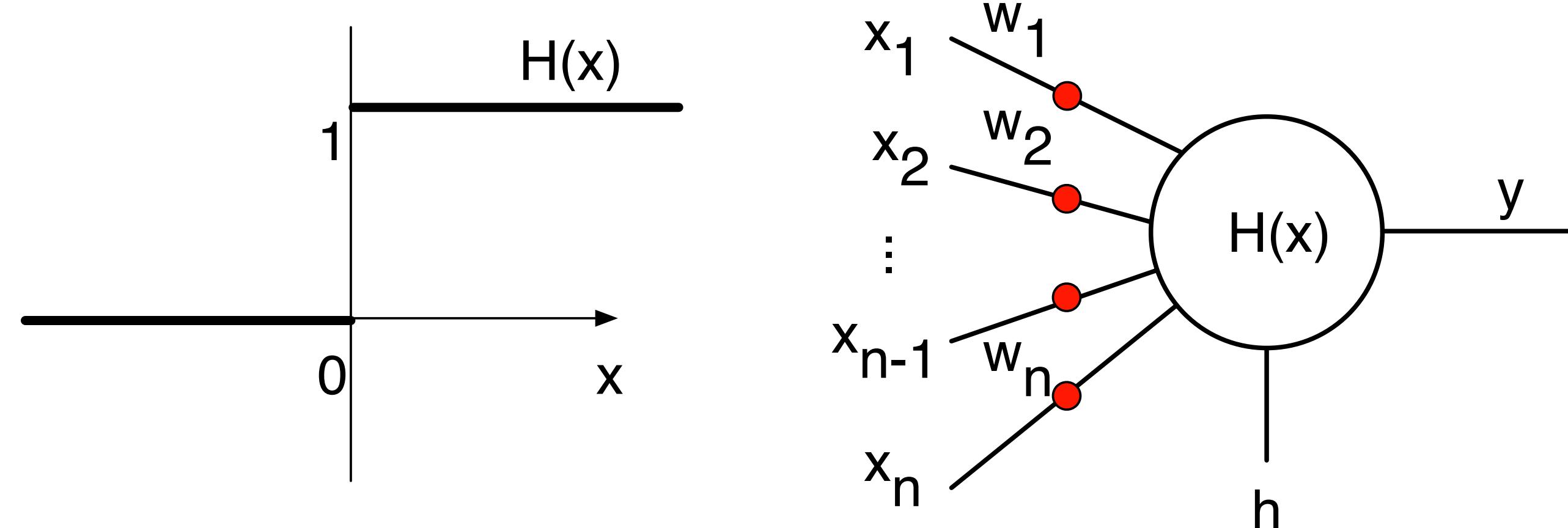


出典

<https://qiita.com/nishiy-k/items/1e795f92a99422d4ba7b>

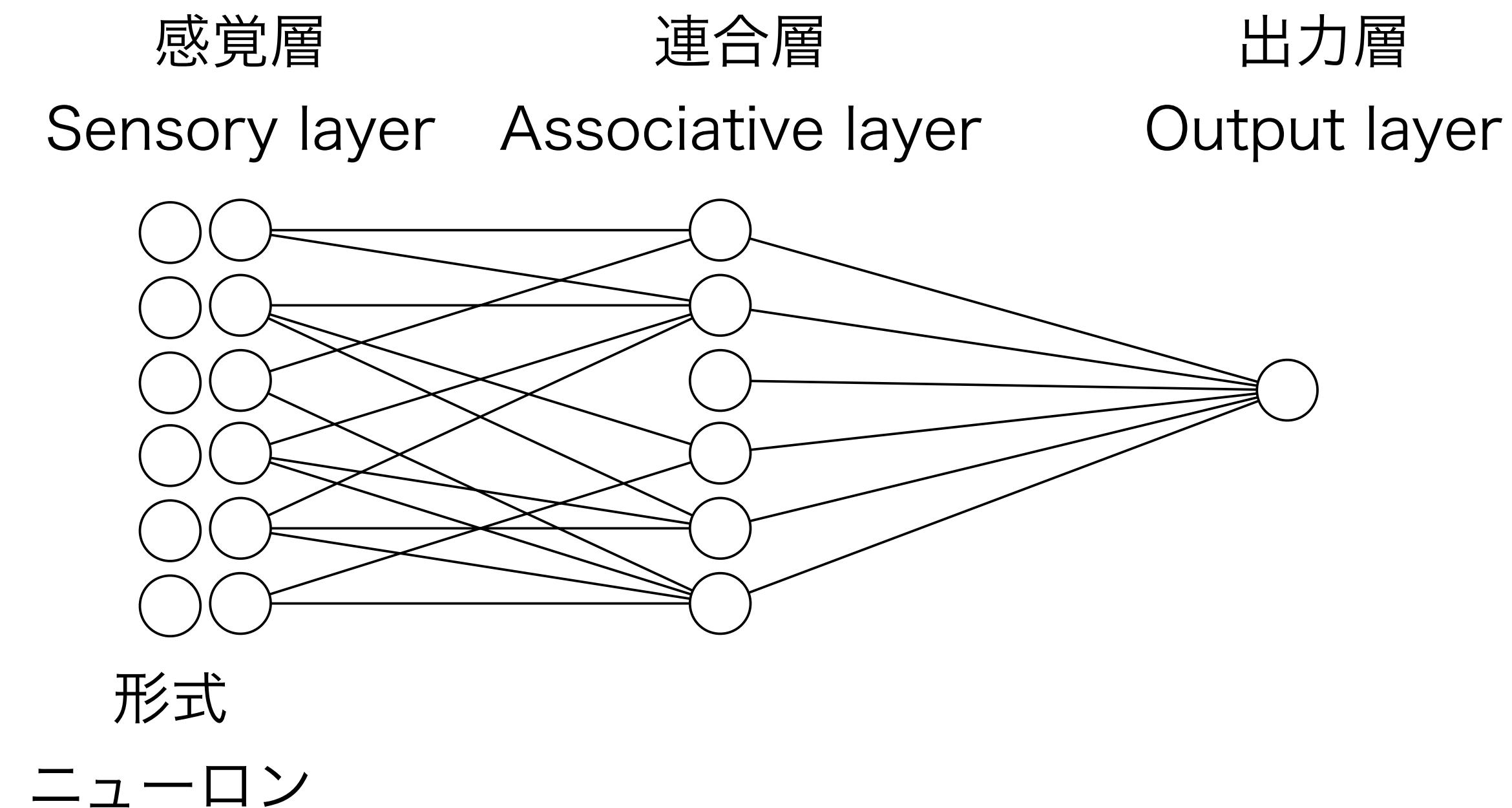
形式ニューロンにおける数理モデル

- ・形式ニューロンは、ステップ関数と線形重み付け計算の組み合わせにより構成される。現実のニューロンの振る舞いを簡略化して得られた
- ・単純なモデルだが、複数の形式ニューロンを組み合わせる強力な能力を持たせることができる
- ・任意の論理回路を構成可能 (AND, OR, NOTのエミュレートが可能)



パーセプトロン

- ・ローゼンブラット(1958)は、形式ニューロンをもとにしたパーセプトロンを発表
- ・パターン認識問題への応用を念頭においている
- ・多層型・順伝播型アーキテクチャの元祖
- ・ミンスキー・パパートにより線形非分離な問題が扱えないことが示される
→第一次ブームの終焉

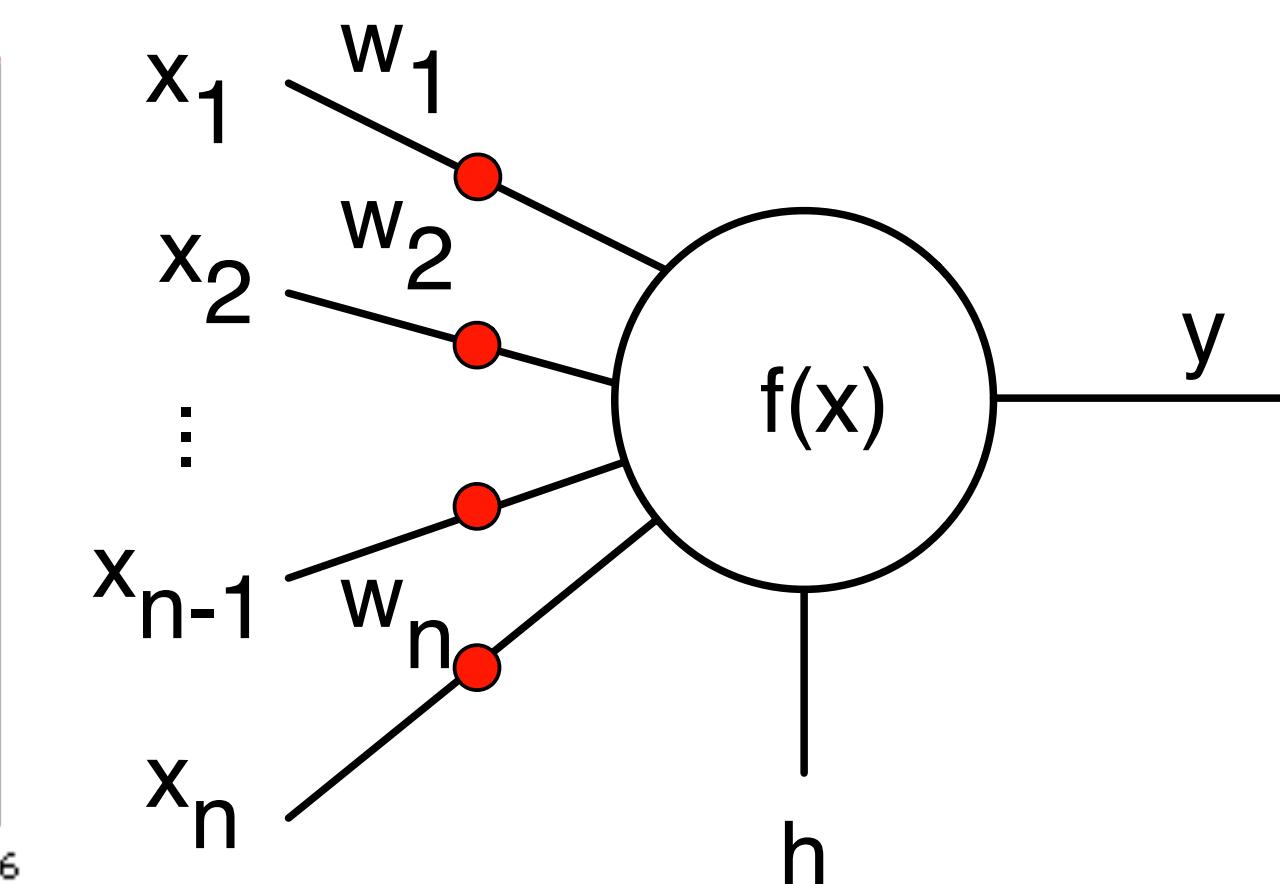
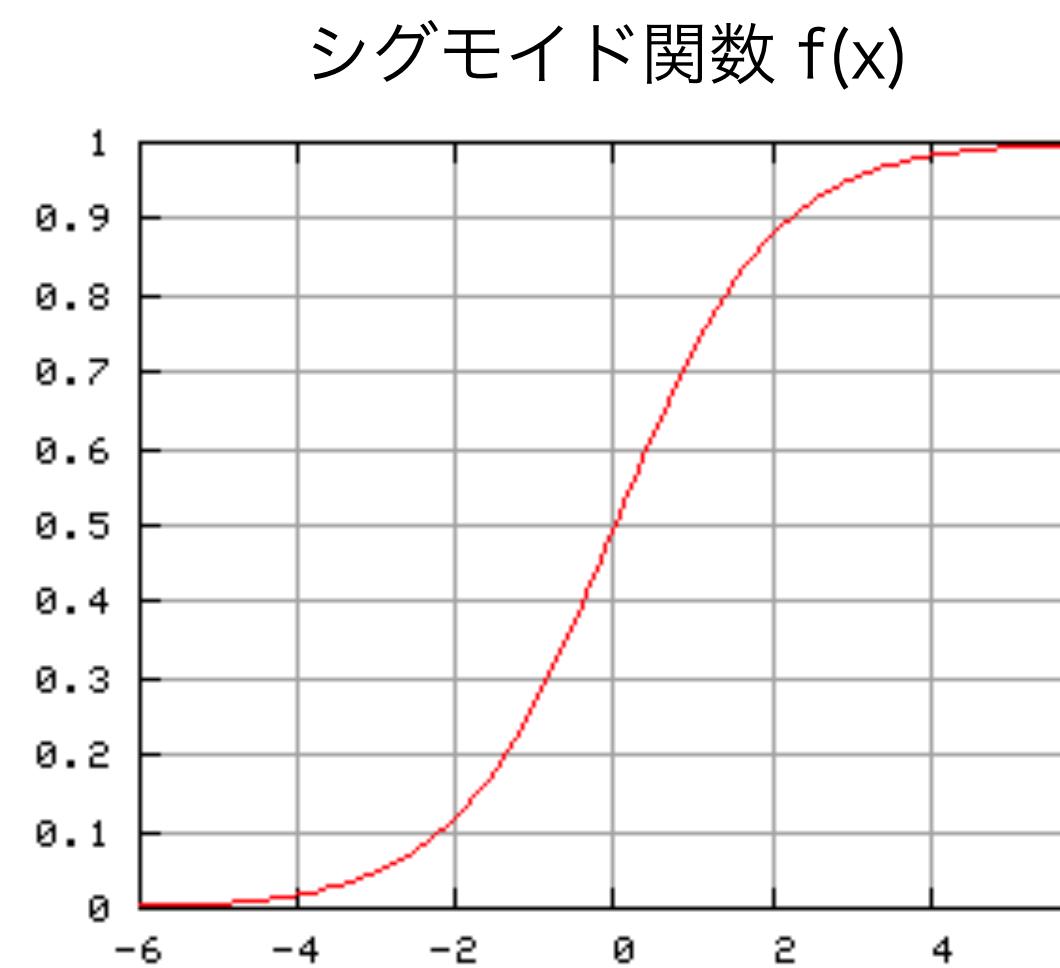


ニューラルネットワークの登場

- 1980年代には、しきい値関数を一般の活性化関数に置き換えたニューラルネットワーク(NN)が登場
- 誤差逆伝播法(back propagation)による効率のよい勾配計算

$$y = f\left(\sum_{i=1}^n w_i x_i + b\right)$$

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



よく利用される活性化関数

シグモイド関数

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

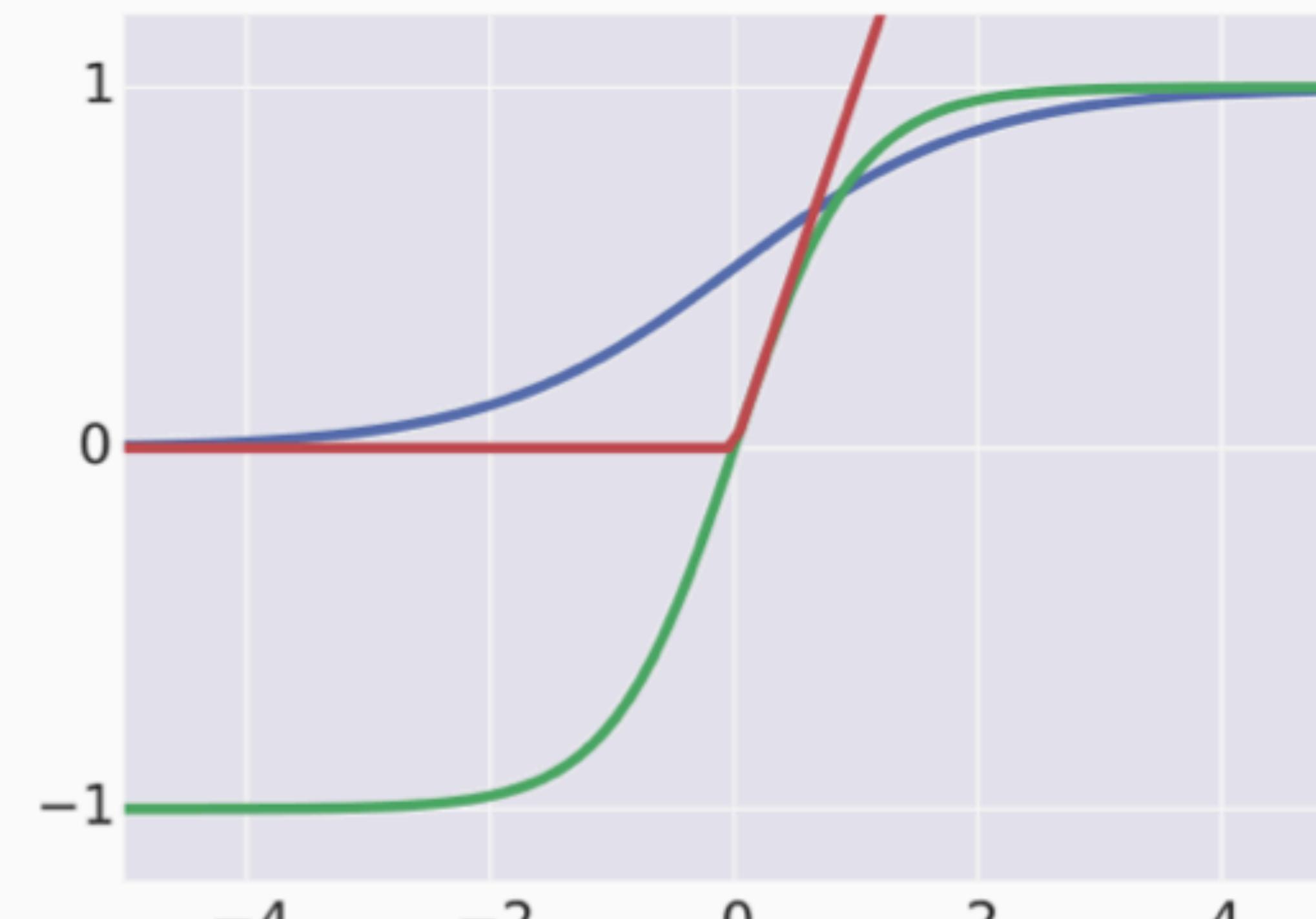
tanh関数

$$\tanh(u) = \frac{e^u - e^{-u}}{e^u + e^{-u}}$$

ReLU関数

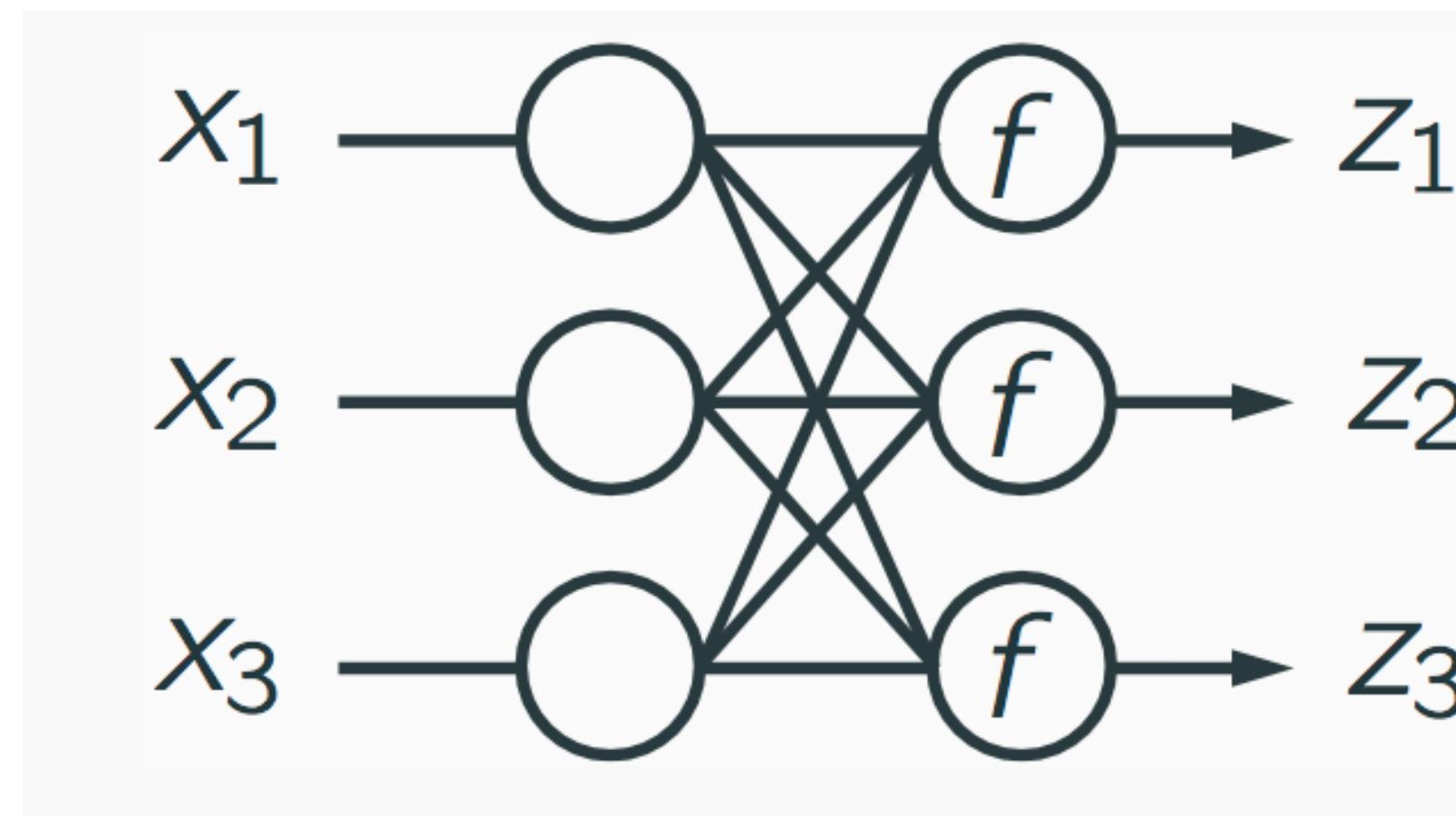
$$\text{ReLU}(u) = \max(0, u)$$

活性化関数



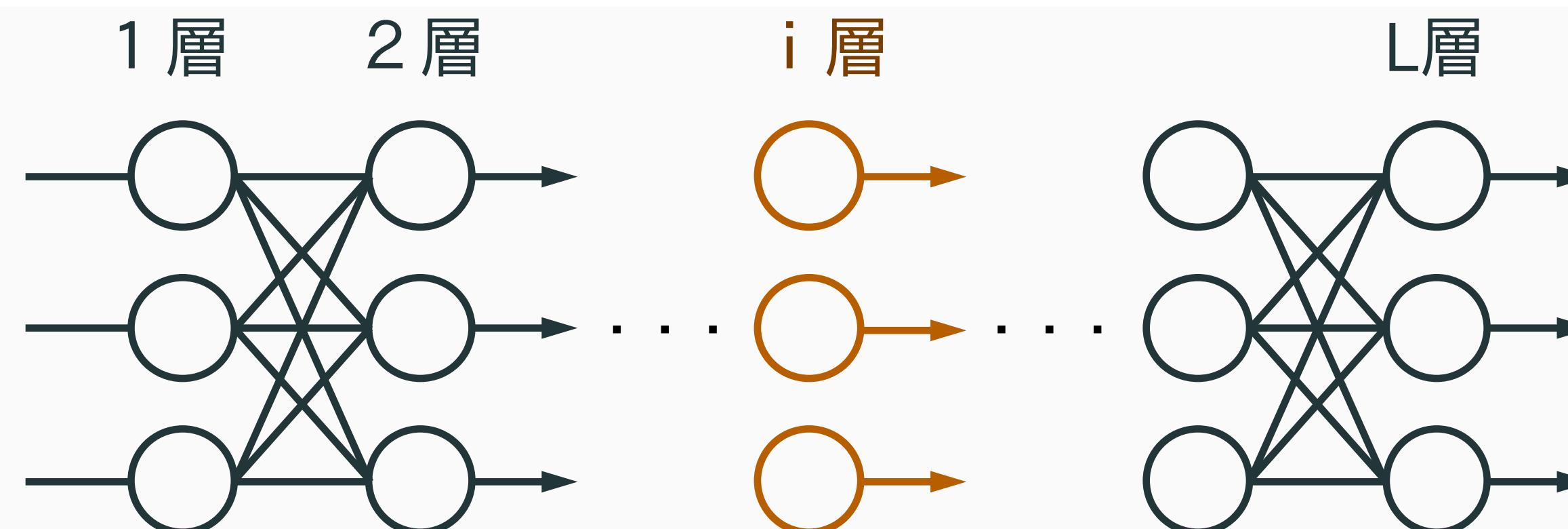
層構造とフィードフォワード型NN

$$z = f(Wx + b)$$



$$W = \begin{pmatrix} W_{1,1} & \dots & W_{1,n} \\ W_{2,1} & \dots & W_{2,n} \\ \vdots & \vdots & \vdots \\ W_{m,1} & \dots & W_{m,n} \end{pmatrix}, \quad b = (b_1, b_2, \dots, b_m)^T$$

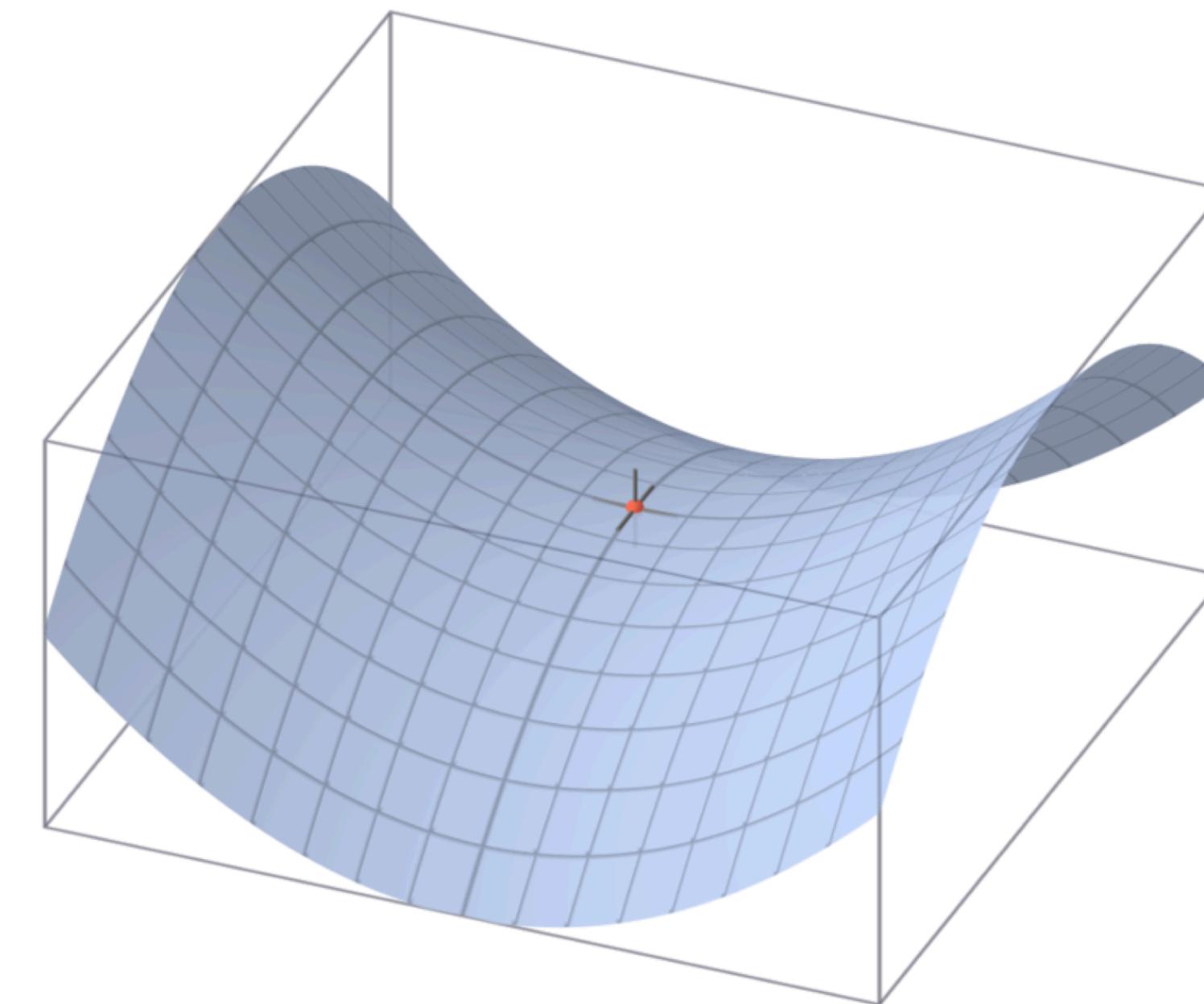
フィードフォワード型NN



重み $W^{(i)}$ バイアス $b^{(i)}$ 活性化関数 $f^{(i)}$

鞍点と学習の停滞

勾配ベクトルが、ほとんどゼロベクトルとなる点（停留点）は、「極大点」「極小点」「鞍点」のいずれか。**探索点が鞍点にとらわれているとき**、ニューラルネットの学習が停滞する

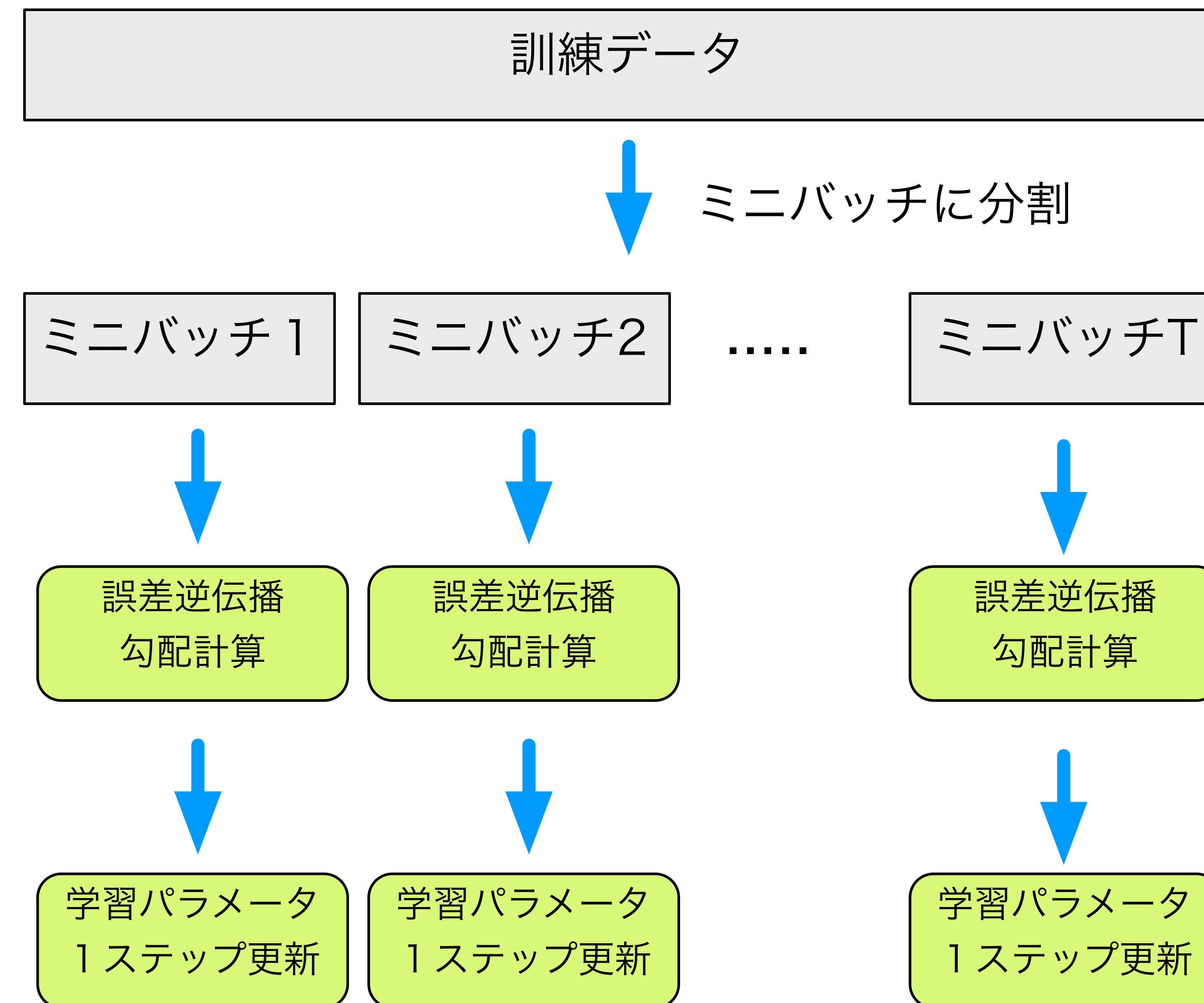


4段以上のネットワークにおいては、停留点による学習の停滞の影響が顕著→学習が進まない！ →NN研究の冬の時代突入

深層ニューラル ネットワークの学習プロセス

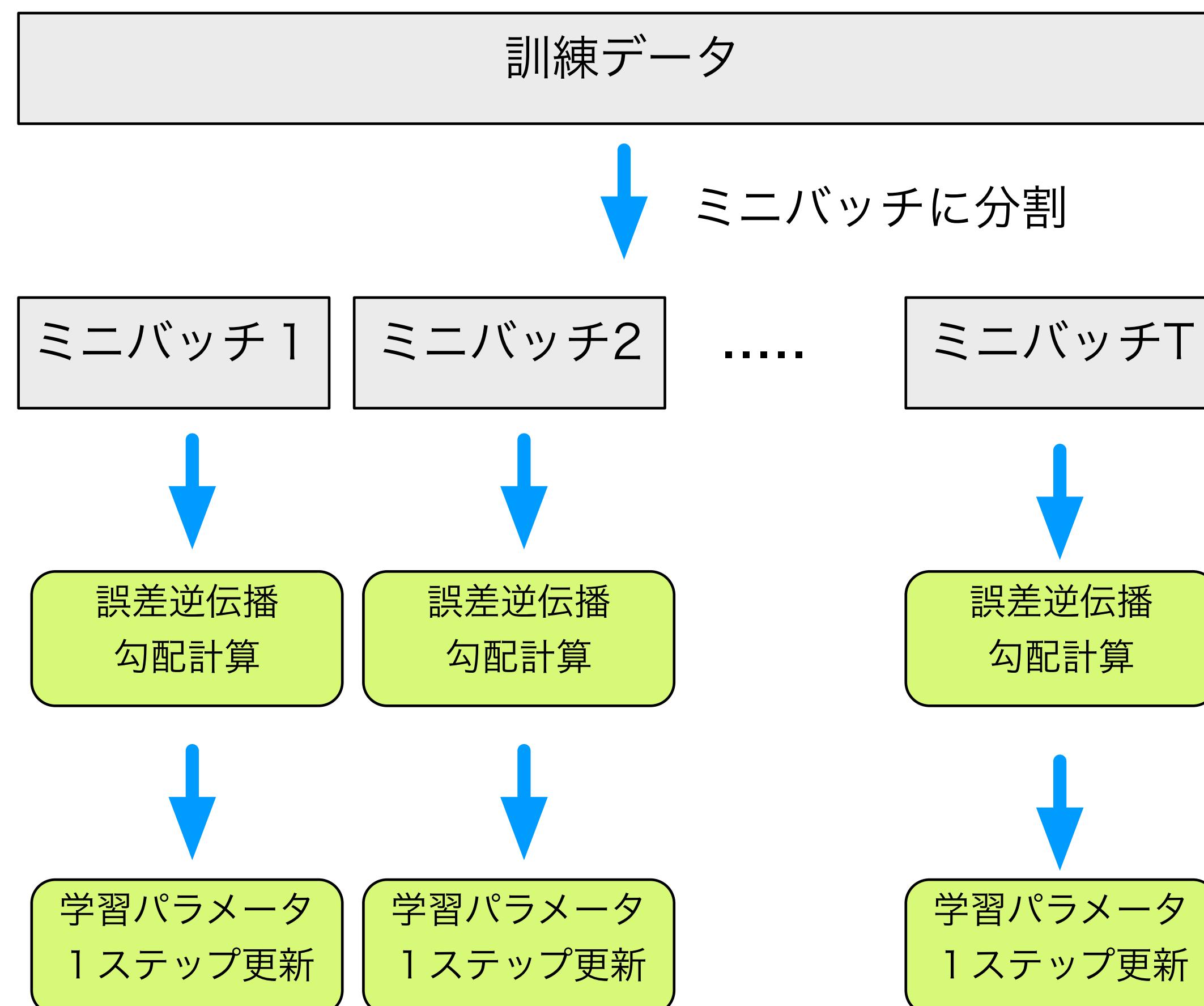


ミニバッチ学習法(1)



ミニバッチ学習法(2)

訓練データ $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_T, y_T)\}$



$$B = \{(x_{b1}, y_{b1}), (x_{b2}, y_{b2}), \dots, (x_{bK}, y_{bK})\}$$

$$G_B(\Theta) = \frac{1}{K} \sum_{k=1}^K loss(y_{bk} - f_\Theta(x_{bk}))$$

目的関数がミニバッチに依存
している点に注意

確率的勾配法(SGD)の手続き

Step 1 (初期点設定) $\Theta := \Theta_0$

Step 2 (ミニバッチ取得) B をランダムに生成

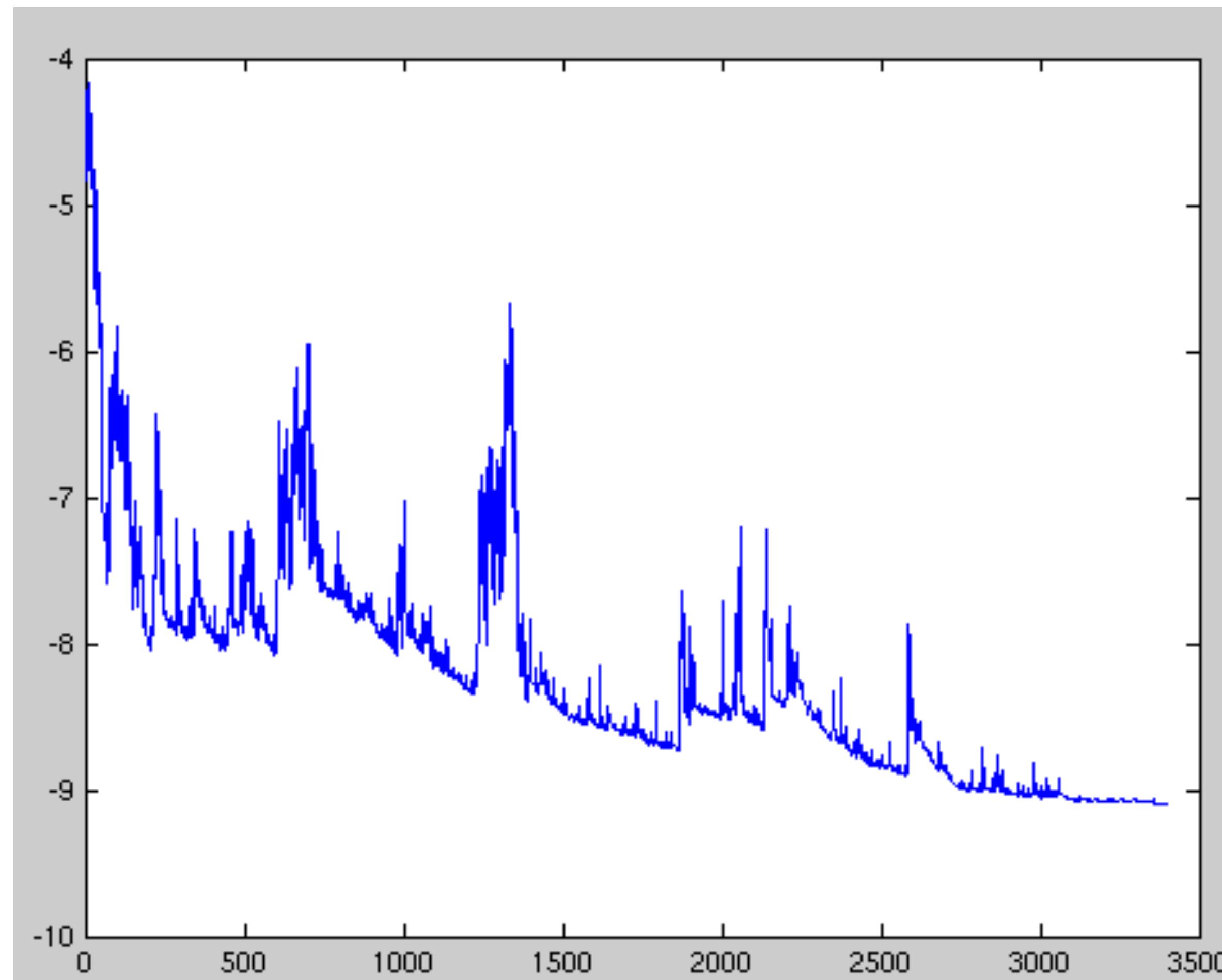
Step 3 (勾配ベクトルの計算) $g := \nabla G_B(\Theta)$

Step 4 (探索点更新) $\Theta := \Theta - \alpha g$

Step 5 (反復) Step 2 に戻る

- ▶ Momentum (慣性法)
- ▶ Adagrad
 - どのがよいかは状況によりけり
- ▶ Adadelta
 - SGD, Momentum, Adam が比較的よく利用されている
- ▶ RMSprop
- ▶ Adam

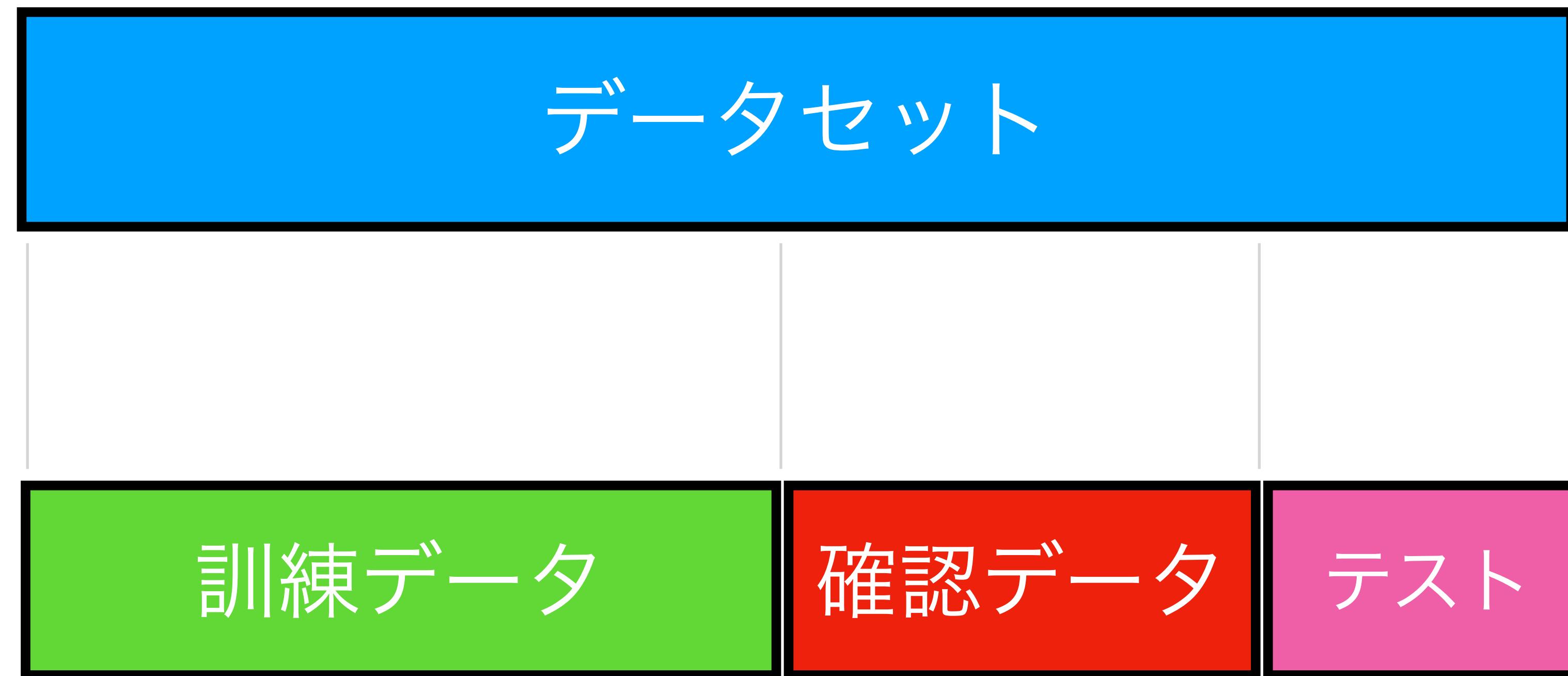
SGDによる損失値の変化



出典 <http://postd.cc/optimizing-gradient-descent/>

確率的なゆらぎにより、鞍点や極値から脱出できる
より詳細な情報→<https://ja.wikipedia.org/wiki/確率的勾配降下法>

データセットの区分



データセットが小規模の場合には、データセットの使い回し技法として、クロスバリデーションなどの技法がしばしば利用される

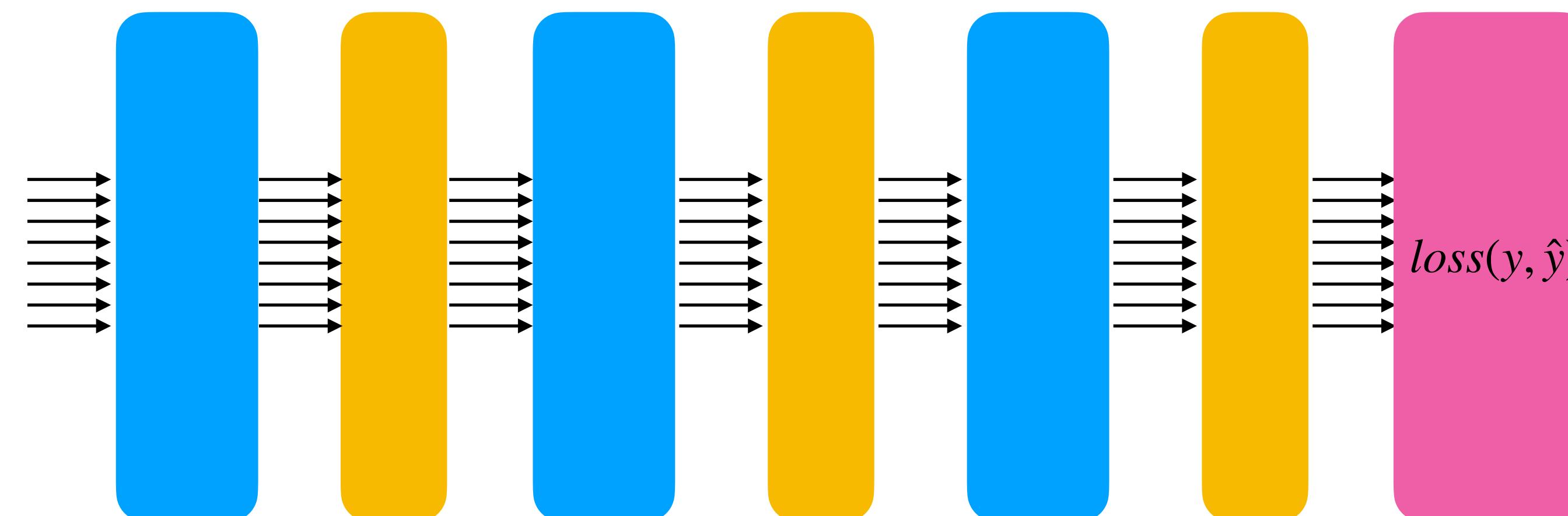
データセットの使い分け

- 訓練データ: 学習プロセスにおいて利用する(なるべく大量のデータがあることが望ましい)
- 確認データ(Validation data): ハイパーパラメータ(ミニバッチサイズ、ネットワーク構造、学習率など) を調整するために利用する
- テストデータ: 汎化誤差(テスト誤差) を評価するために利用する。汎化誤差を小さくすることがわれわれの目標

勾配ベクトルの計算

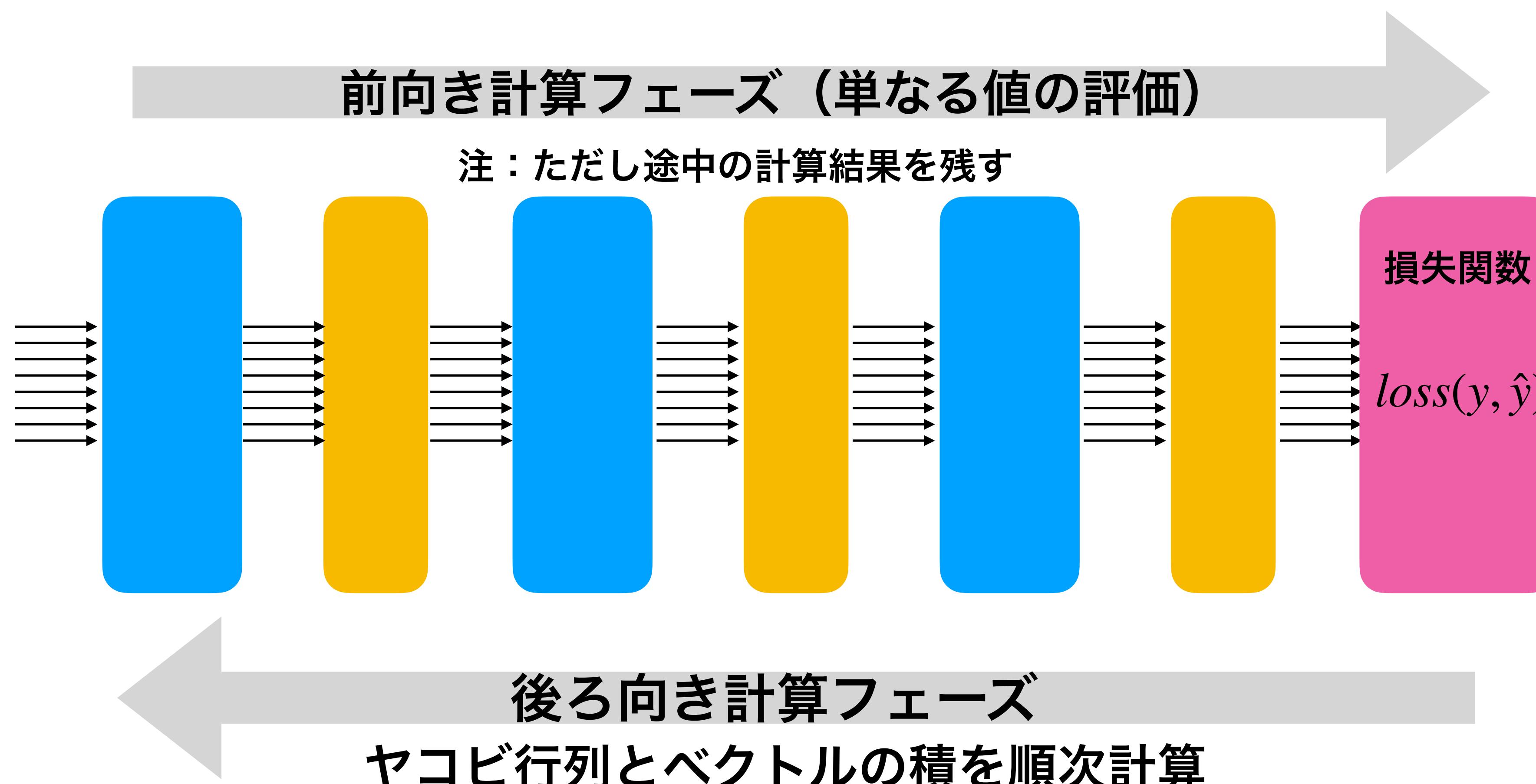
- ✓ 確率的勾配法を利用するためには、学習パラメータ Θ に関する勾配ベクトル(gradient)の計算が必要
- ✓ 学習における計算量は勾配計算が支配する
- ✓ 層構造のネットワークに対して、効率のよい計算方法とは？

$$\Theta = \{W_1, b_1, W_2, b_2, \dots\}$$



誤差逆伝播法(詳細は次回)

- ・パラメータの勾配ベクトルを効率良く求めることが目的



中間まとめ

-  **最適化技法として確率的勾配法を利用**
-  学習パラメータの勾配ベクトルの効率的な計算には誤差逆伝播法(back propagation)を利用
-  よい汎化性能(十分に小さい汎化誤差)を得るためには、大量の訓練データが必要
-  一般には、非凸最適化となる

Q&A

- Q: さきほどの話を聞いていると簡単そうに聞こえるが、なぜ2000年台まで、深層学習が登場しなかったの？

A: 深い層のネットワークの学習は、**勾配消失問題**のため、実際には簡単ではない。確率的勾配法・ReLU関数の登場・強力な正則化技法・大量のラベル付きデータの利用などにより深層ネットワークが扱えるようになってきました。

Q&A

- Q: 学習にはすごい計算パワーがいると聞くけど、どうなんですか？
A: パラメータの勾配ベクトルを求めるために**バックプロパゲーション**というアルゴリズムを利用します。バックプロパゲーションでは大規模なテンソル計算（行列計算）が必要になります。専用ハードウェアやGPUの利用が広がっています。



深層学習向けハードウェア

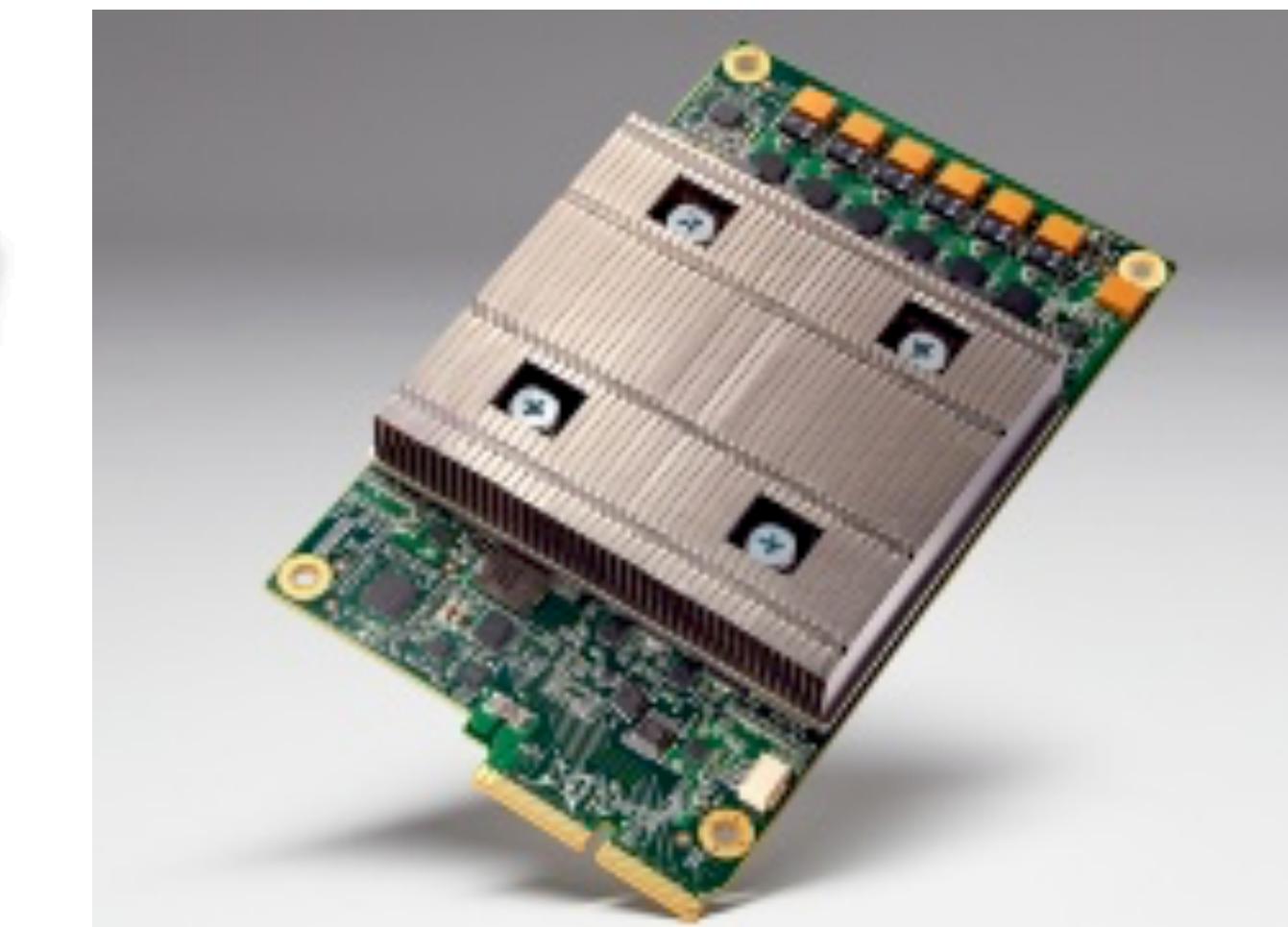
推論フェーズ（前向き計算）、逆伝播フェーズ（後ろ向き計算）のいずれにも大量の行列ベクトル積計算が必要とされる。汎用 CPU 以上に効率の良い計算が可能な GPU、または、専用のハードウェアへの注目が集まっている。

NVIDIA TESLA GPU



出典 <http://www.nvidia.co.jp/object/tesla-servers-jp.html>

Google Tensor Processing Unit (TPU)



出典 <http://itpro.nikkeibp.co.jp/atcl/ncd/14/457163/052001464/>

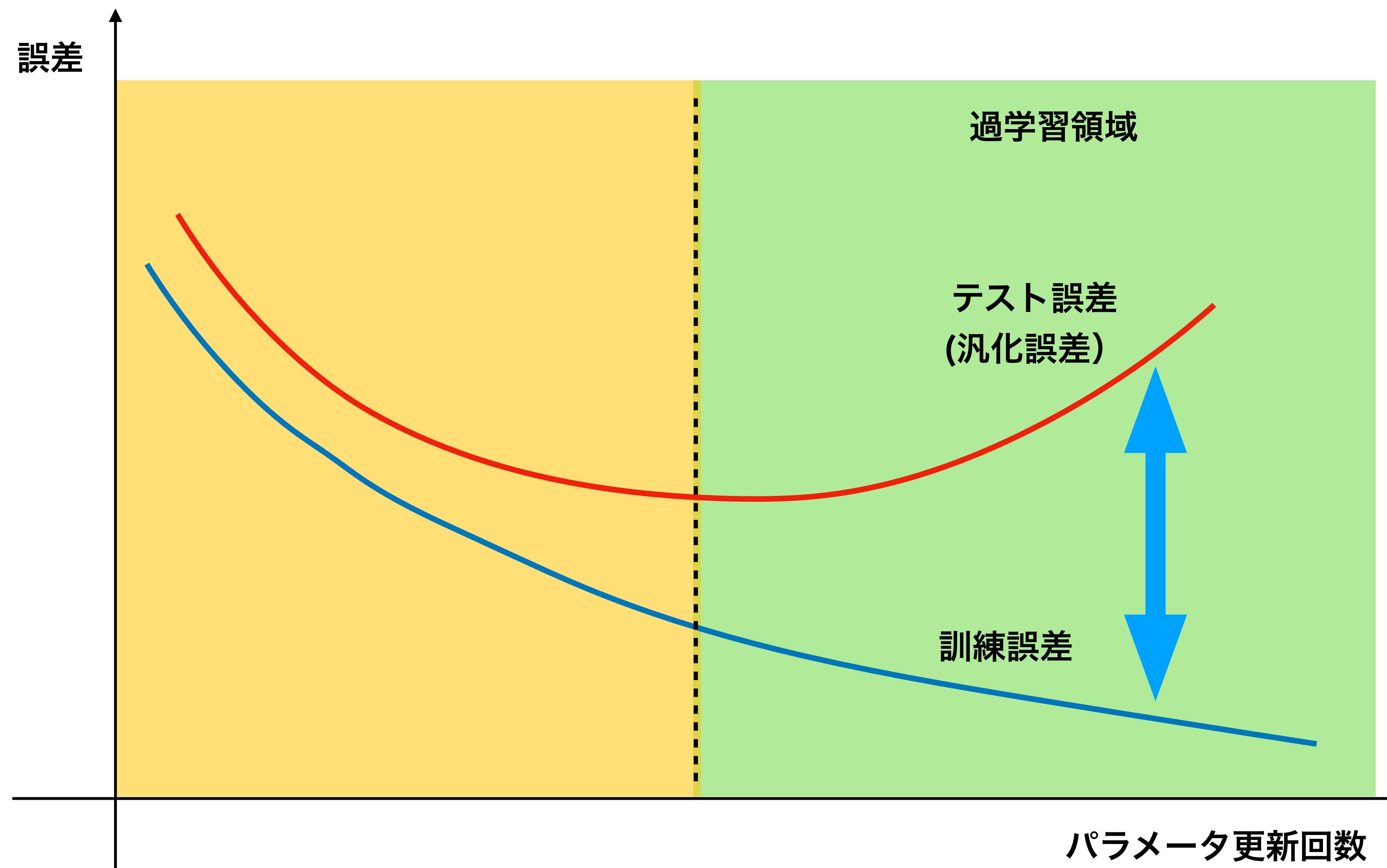
最低でも 1080ti 1枚程度のGPUパワーがないと実験などを行うのはツライ！

Q&A

- Q:ミニバッチ学習では、訓練データからミニバッチをいくつも生成して学習を進めます。繰り返せば繰り返すほど認識器の性能はよくなるんでしょうか？

A: **過学習**の影響で、次第にテスト誤差（こちらが本来重要）が劣化する場合があります。次のページの図も参考にしてください。（この図をしっかり頭にいれておくことは機械学習技術を実践で利用する際にはとても重要です）。

訓練誤差とテスト誤差



PyTorchを使う



Deep-Learning(DL)フレームワーク

- ・深層学習のコードをフルスクラッチで(ゼロから)作るのは辛い！
- ・DLフレームワークにより簡単にプログラミング可能
- ・最も書くのが面倒な誤差逆伝播法の逆方向計算を自動で計算してくれる（自動微分）
- ・SGDなどの最適化関数やミニバッチ学習の仕組みも内蔵されている
- ・現状、TensorFlow、もしくはPyTorchがメジャー

PyTorchのコード例(1)

- PyTorchによるプログラム



```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 2)
        self.fc2 = nn.Linear(2, 2)

    def forward(self, x):
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        return x
```

ネットワークの定義部

順方向計算のみを記述すればよい。

誤差逆伝播法の後ろ向き計算フェーズ

```
model = Net()
```

ネットワークのインスタンス

```
loss_func = nn.MSELoss()
```

損失関数の指

```
optimizer = optim.Adam(model.parameters(), lr=0.1)
```

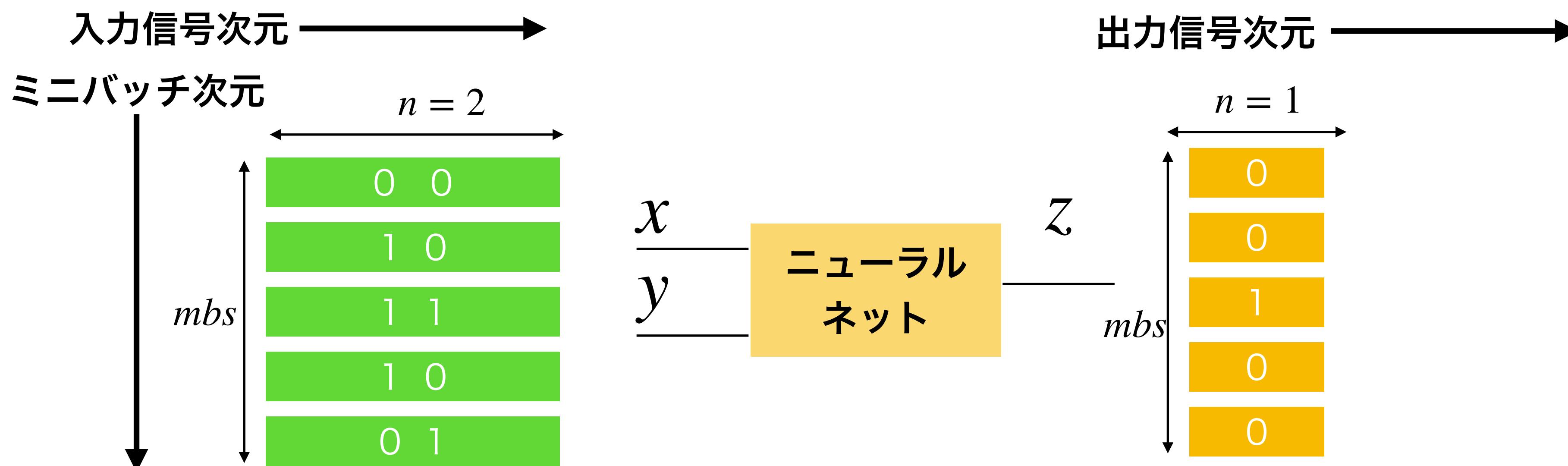
オプティマイザの指

PyTorchのコード例(2)

```
for i in range(1000):
    inputs = torch.bernoulli(0.5 * torch.ones(mb_size, 2))
    labels = torch.Tensor(mb_size, 2)
    for j in range(mb_size):
        if (inputs[j, 0] == 1.0) and (inputs[j, 1] == 1.0):
            labels[j, 0] = 1.0
            labels[j, 1] = 0.0
        else:
            labels[j, 0] = 0.0
            labels[j, 1] = 1.0
    optimizer.zero_grad()
    outputs = model(inputs)          前向き計算
    loss = loss_func(outputs, labels)
    loss.backward()
    optimizer.step()                後ろ向き計算
                                    パラメータ更新
```

PyTorchとテンソル計算(1)

- ・テンソル = 多次元配列 (行列・ベクトルの多次元版)
- ・1次元テンソル = ベクトル, 2次元テンソル = 行列
- ・ミニバッチを一つのテンソルとして、ニューラルネットワークに入力する



PyTorchとテンソル計算(2)

<https://www.atmarkit.co.jp/ait/articles/2002/13/news006.html>

テンソルの作成

```
# テンソルの新規作成
x = torch.empty(2, 3) # 2行×3列のテンソル（未初期化状態）を生成
x = torch.rand(2, 3) # 2行×3列のテンソル（ランダムに初期化）を生成
x = torch.zeros(2, 3, dtype=torch.float) # 2行×3列のテンソル（0で初期化、torch.float型）を生成
x = torch.ones(2, 3, dtype=torch.float) # 2行×3列のテンソル（1で初期化、torch.float型）を生成
x = torch.tensor([[0.0, 0.1, 0.2],
                  [1.0, 1.1, 1.2]]) # 1行×2列のテンソルをPythonリスト値から作成
```

テンソルの計算

```
# テンソルの計算操作
x + y          # 演算子を使う方法
torch.add(x, y) # 関数を使う方法
```

インデキシング

```
# インデクシングやスライシング（NumPyのような添え字を使用可能）
print(x)        # 元は、2行3列のテンソル
x[0, 1]         # 1行2列目（※0スタート）を取得
```

AND関数を学習する

論理関数の学習可能性はパーセプトロンの頃は重要な問題(だった)



AND関数の真理値表

x	y	z
0	0	0
0	1	0
1	0	0
1	1	1

ここで考える学習タスク:
ニューラルネットでAND関数を模擬



データセット: $\{(0,0), 0\}, \{(0,1), 0\}, \{(1,1), 1\}, \dots$

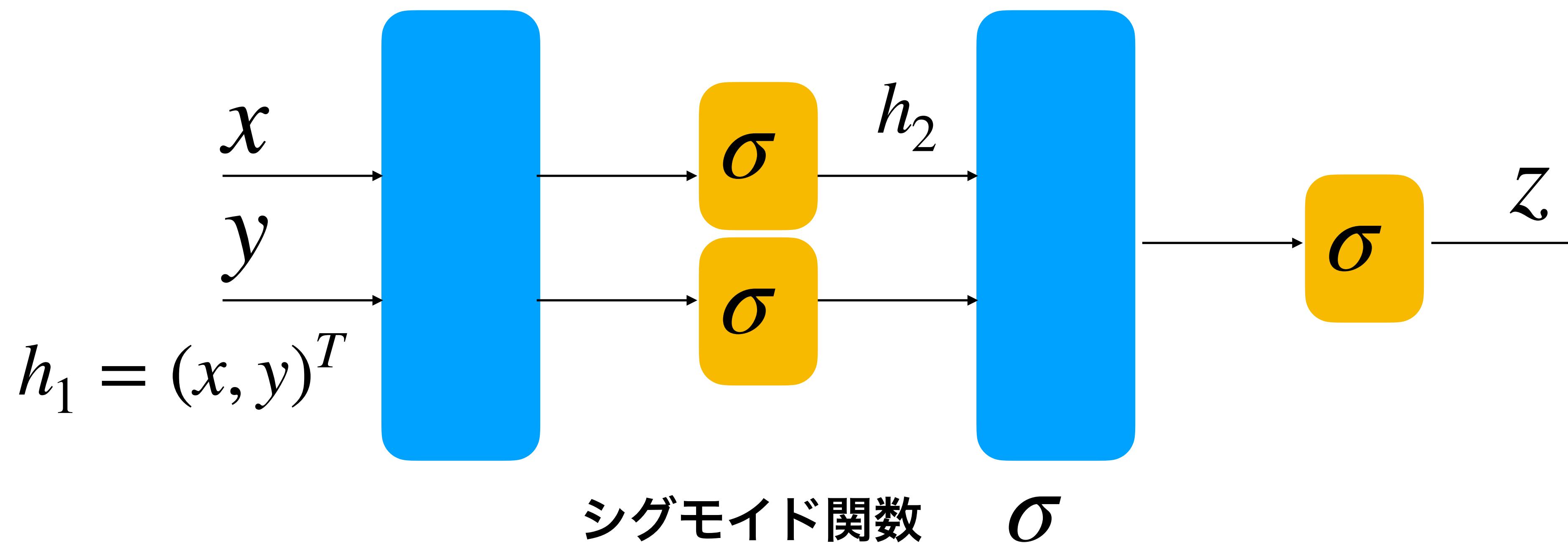
ニューラルネットワーク(NN)モデル

$$W_1 \in \mathbb{R}^{2 \times 2}$$

$$W_1 h_1 + b_1$$

$$W_2 \in \mathbb{R}^{1 \times 2}$$

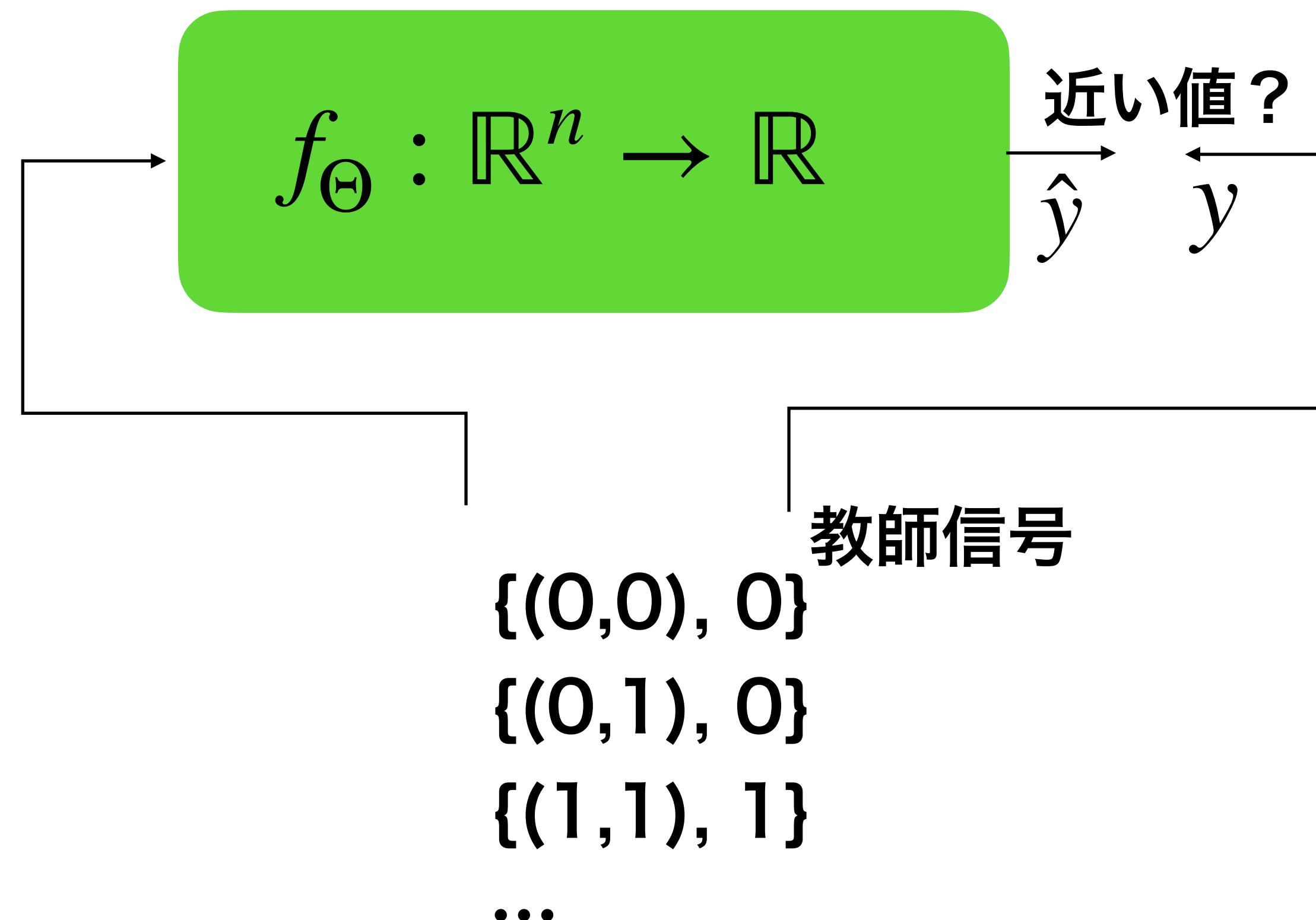
$$W_2 h_2 + b_2$$



$\{W_1, W_2, b_1, b_2\}$ は調整可能 → 「AND関数」に近づくように調整

学習プロセス

出力と教師ラベルとの間の**食い違い(損失関数値)**がなるべく小さくなるように学習可能パラメータを更新する



ここでは二乗誤差関数を利用

AND学習のコード(NN定義部)

In []:

AND学習のコード

 Open in Colab

本ノートブックでは、ニューラルネットワークによりAND関数 $AND(a,b)$ の学習を行う。

必要なパッケージのインポート

```
In [ ]: import torch # テンソル計算など  
import torch.nn as nn # ネットワーク構築用  
import torch.optim as optim # 最適化関数
```

PyTorch利用の場合
にはインポート

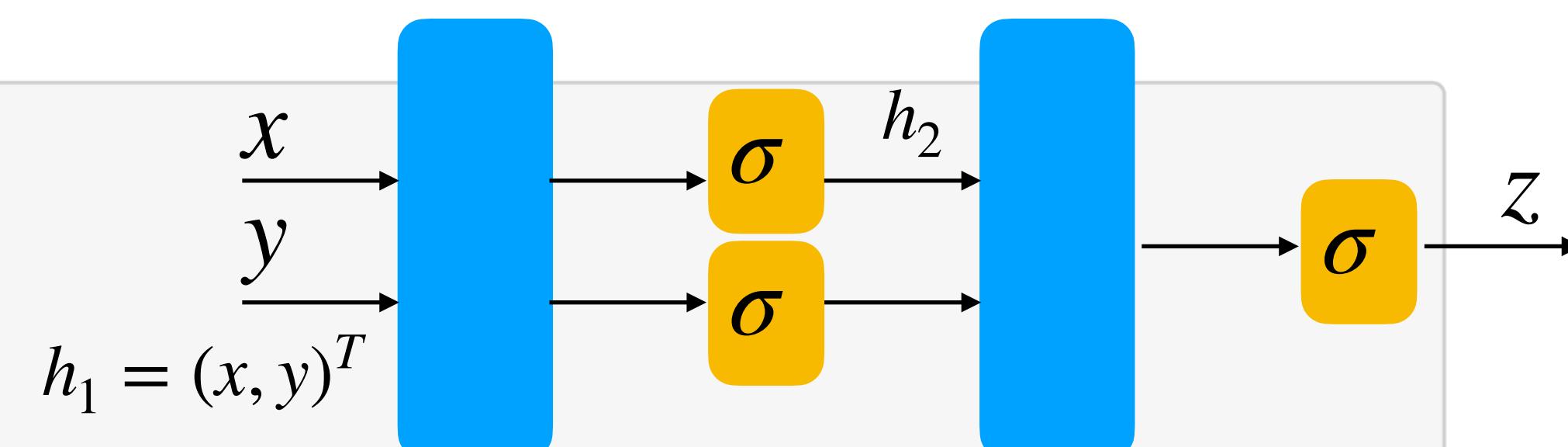
グローバル定数の設定

```
In [ ]: mbs = 5 # ミニバッチサイズ
```

ミニバッチ学習を利用するので
バッチサイズを5と設定

ネットワークの定義

```
In [ ]: class Net(nn.Module): # nn.Module を継承  
    def __init__(self): # コンストラクタ  
        super(Net, self).__init__()  
        self.fc1 = nn.Linear(2, 2) # w_1, b_1  
        self.fc2 = nn.Linear(2, 1) # w_2, b_2  
    def forward(self, x): # 推論計算をforwardに書く  
        x = torch.sigmoid(self.fc1(x)) # 活性化関数としてシグモイド関数を利用  
        x = torch.sigmoid(self.fc2(x))  
        return x
```

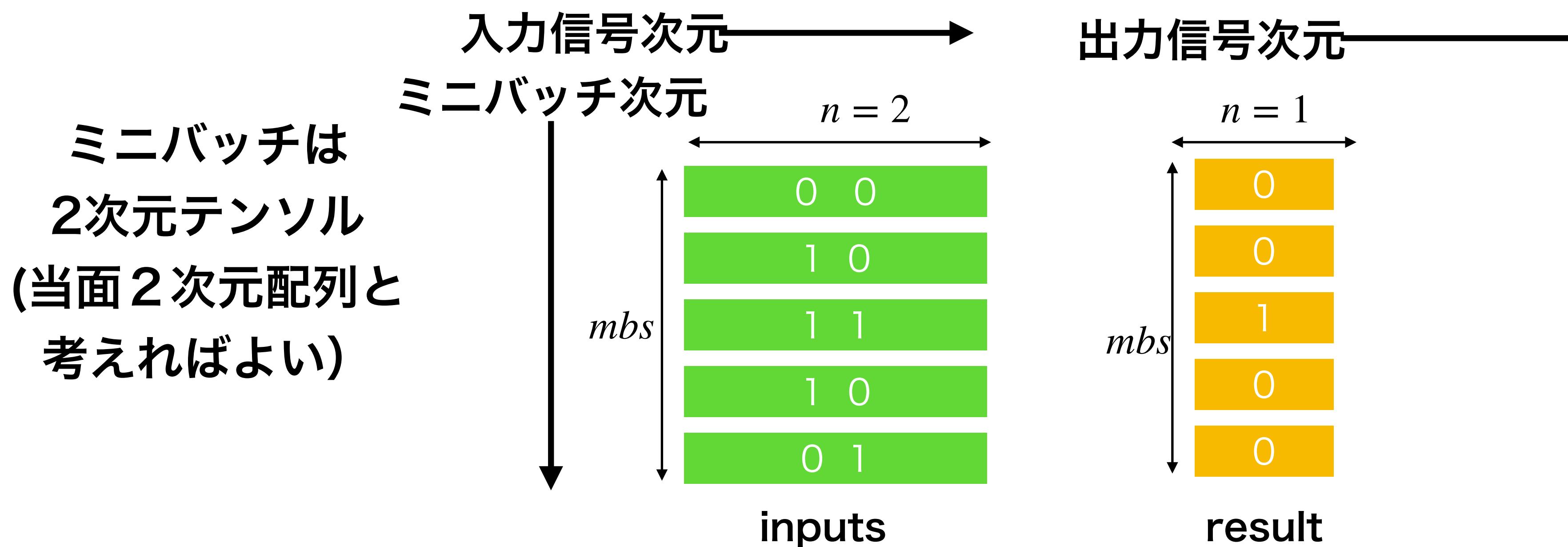


ミニバッチ生成部

ミニバッチ生成関数

```
In [ ]: def gen_minibatch():
    inputs = torch.bernoulli(0.5 * torch.ones(mbs, 2)) # (0,1)ランダム乱数テンソル(サイズ mbs x 2)
    result = torch.Tensor(mbs, 1)
    for j in range(mbs):
        if (inputs[j, 0] == 1.0) and (inputs[j, 1] == 1.0): # AND関数
            result[j] = 1.0
        else:
            result[j] = 0.0
    return inputs, result
```

```
In [ ]: inputs,result = gen_minibatch() # ミニバッチ生成の実行例
print('inputs = ', inputs)
print('result = ', result)
```



訓練ループ

訓練ループ

```
In [ ]: model= Net() # ネットワークインスタンス生成  
loss_func = nn.MSELoss() # 損失関数の生成(二乗損失関数)  
optimizer = optim.Adam(model.parameters(), lr=0.1) # オプティマイザの生成(Adamを利用)  
for i in range(1000):  
    inputs, result = gen_minibatch() # ミニバッチの生成  
    optimizer.zero_grad() # オプティマイザの勾配情報初期化  
    outputs = model(inputs) # 推論計算  
    loss = loss_func(outputs, result) # 損失値の計算  
    loss.backward() # 誤差逆伝播法(後ろ向き計算の実行) ← バックプロップ  
    optimizer.step() # 学習可能パラメータの更新  
    if i % 100 == 0:  
        print('i =', i, 'loss =', loss.item()) ← 損失関数値の表示
```

訓練ループの構造はどのプログラムでもほとんど同じ
学習率の設定は、学習の成否に大きく影響を与える

本日のまとめ

- ニューラルネットワークの歴史を振り返る
- 深層ニューラルネットワークの学習プロセス
- PyTorchを使う
- AND関数学習のコードを学ぶ