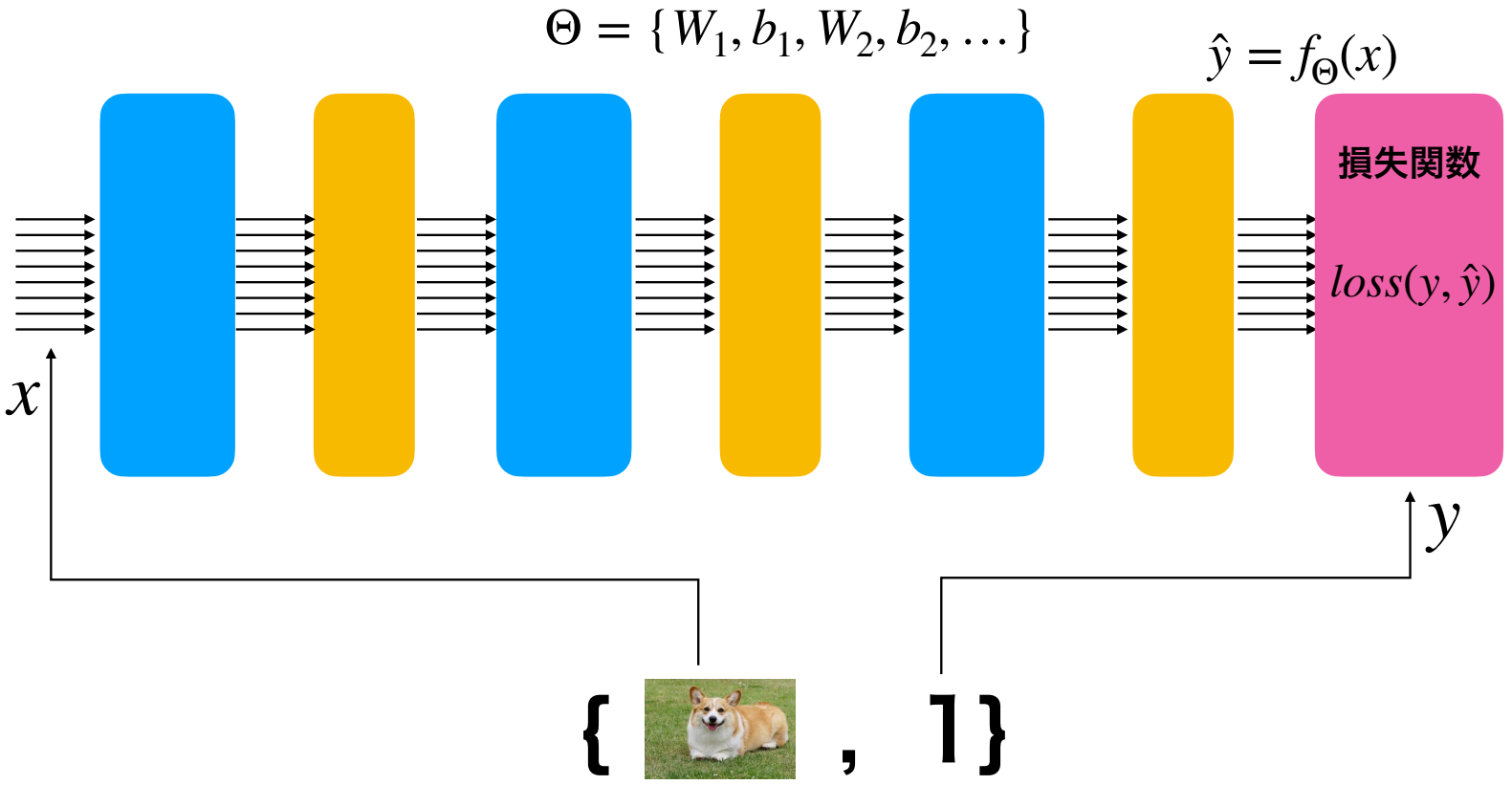
研究室ローテーション 第４回

担当：和田山 正・中井彩乃

本講義の内容

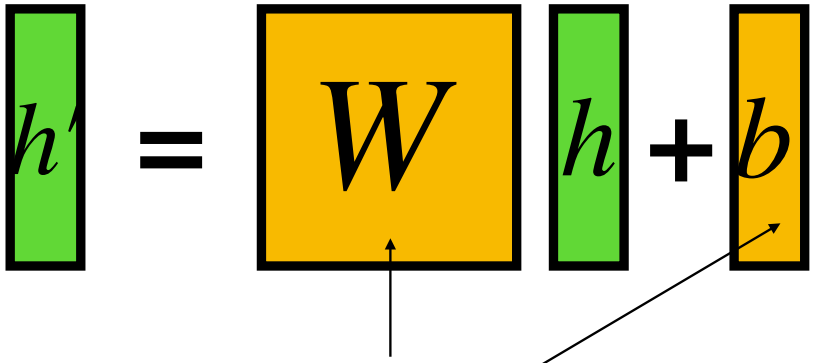
- PyTorchを使う(AND関数学習)

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，損失関数値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



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

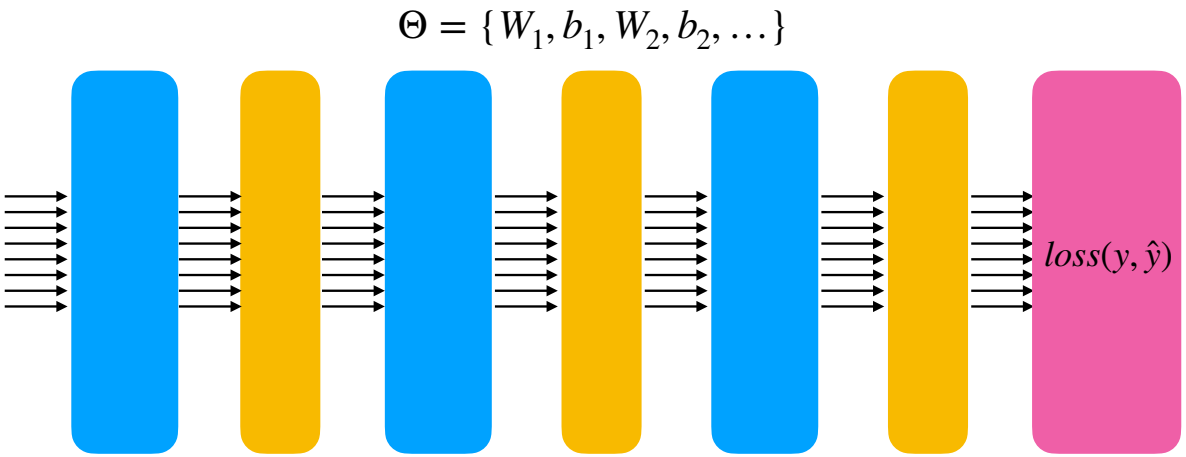
$$loss(y, \hat{y}) = \frac{1}{2} \sum_{i=1}^{n} |y_i - \hat{y}_i|^2$$

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

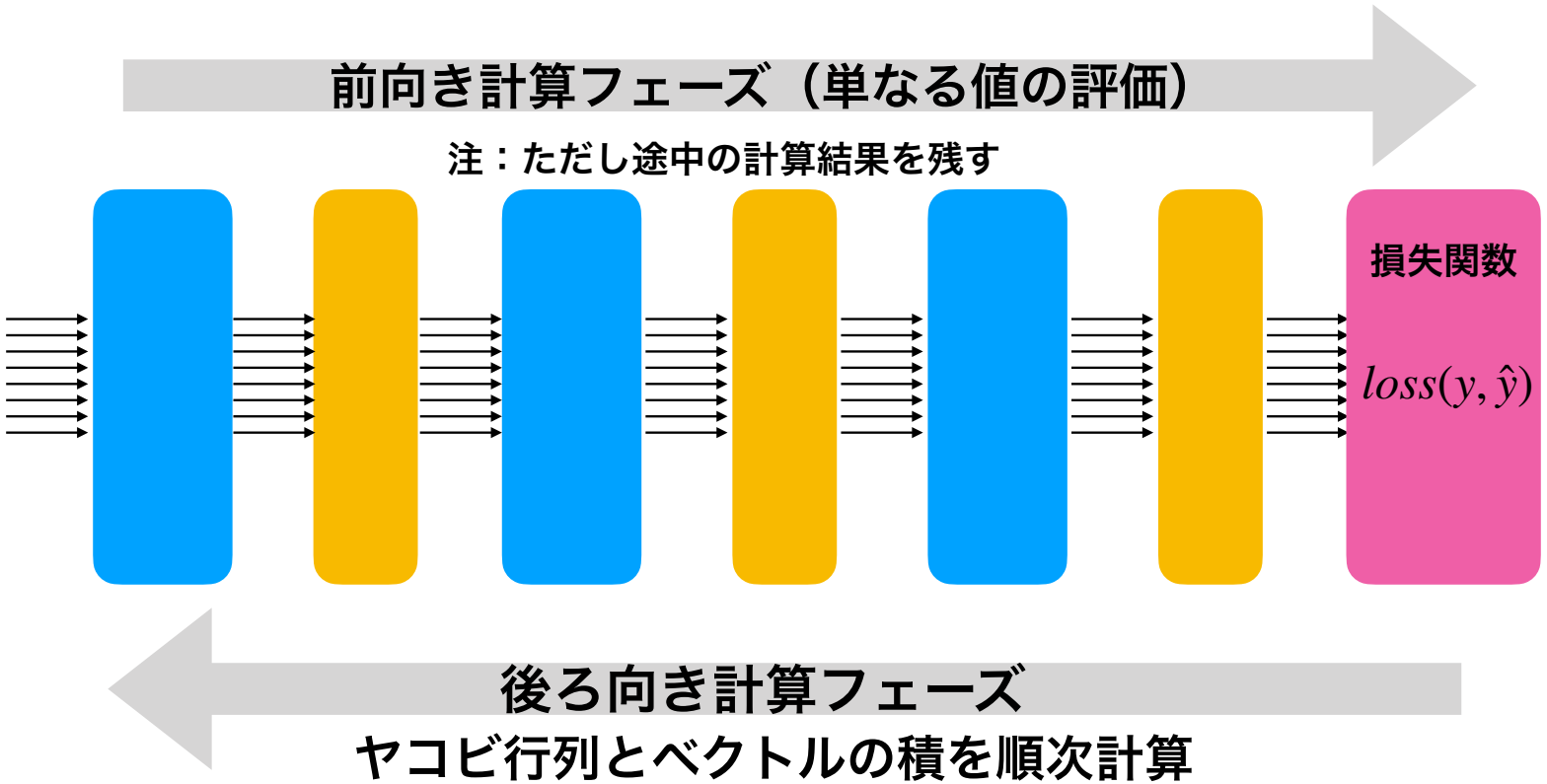
勾配ベクトルの計算

- 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?



誤差逆伝播法

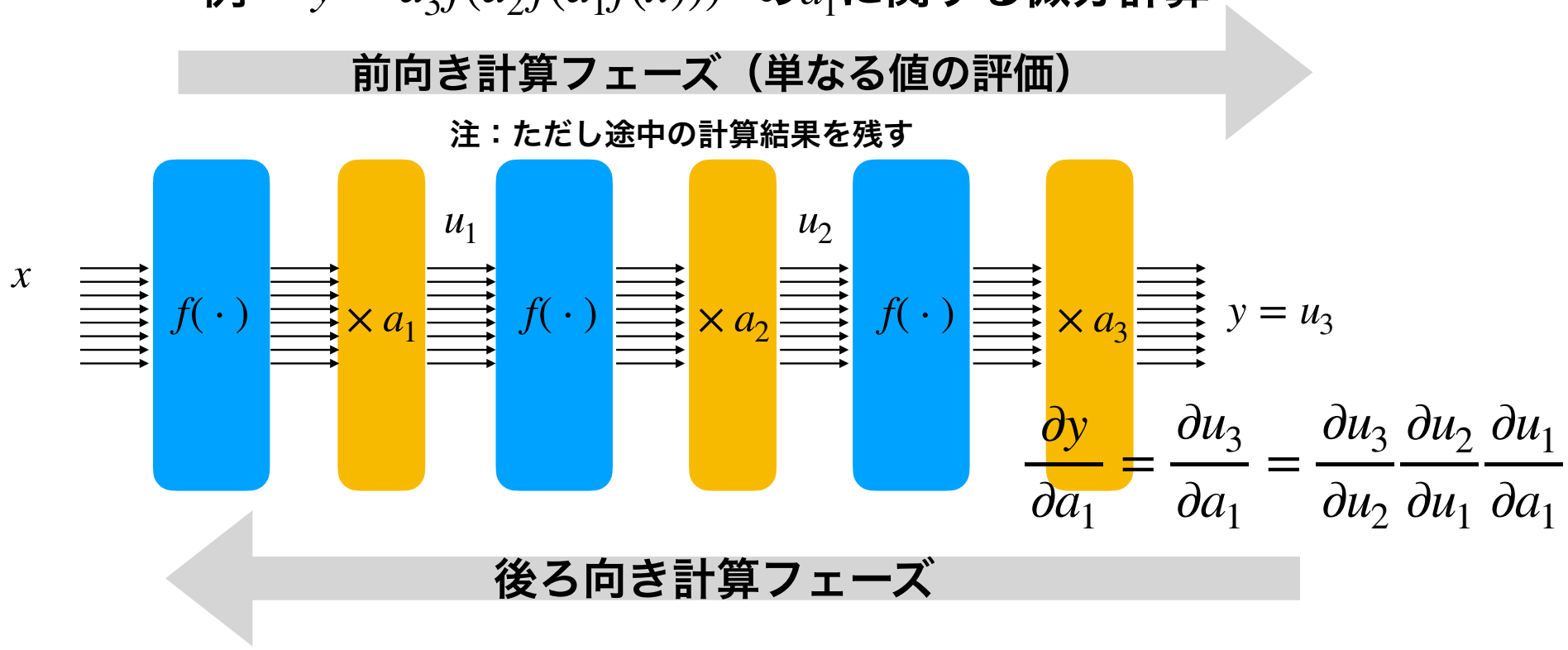
・パラメータの勾配ベクトルを効率良く求めることが目的



誤差逆伝播法の例

微分の連鎖律$\frac{df}{dx} = \frac{df}{dg}\frac{dg}{dx}$を利用する計算方法.

例： $y = a_3 f(a_2 f(a_1 f(x)))$ のa_1に関する微分計算



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によるプログラム例


```
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は明示的にユーザが書く必要ない

ネットワークのインスタンス化

```
model = Net()
loss_func = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.1)
```

損失関数の指定

オプティマイザの指定

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)
    loss = loss_func(outputs, labels)
    loss.backward()
    optimizer.step()
```

訓練データのフィード

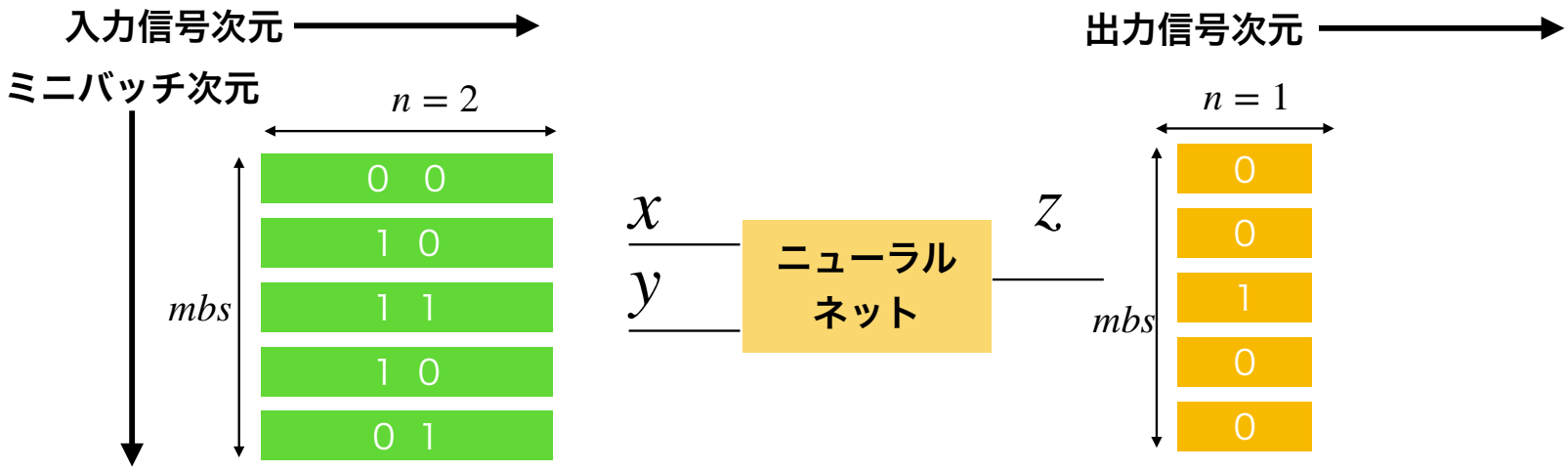
前向き計算

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

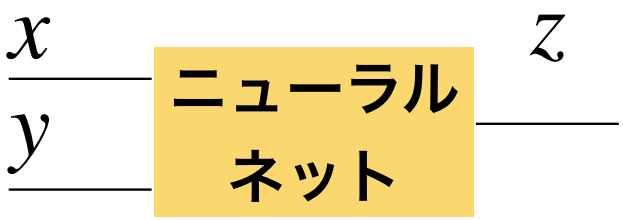
論理関数の学習可能性はパーセプトロンの頃は重要な問題(だった)

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



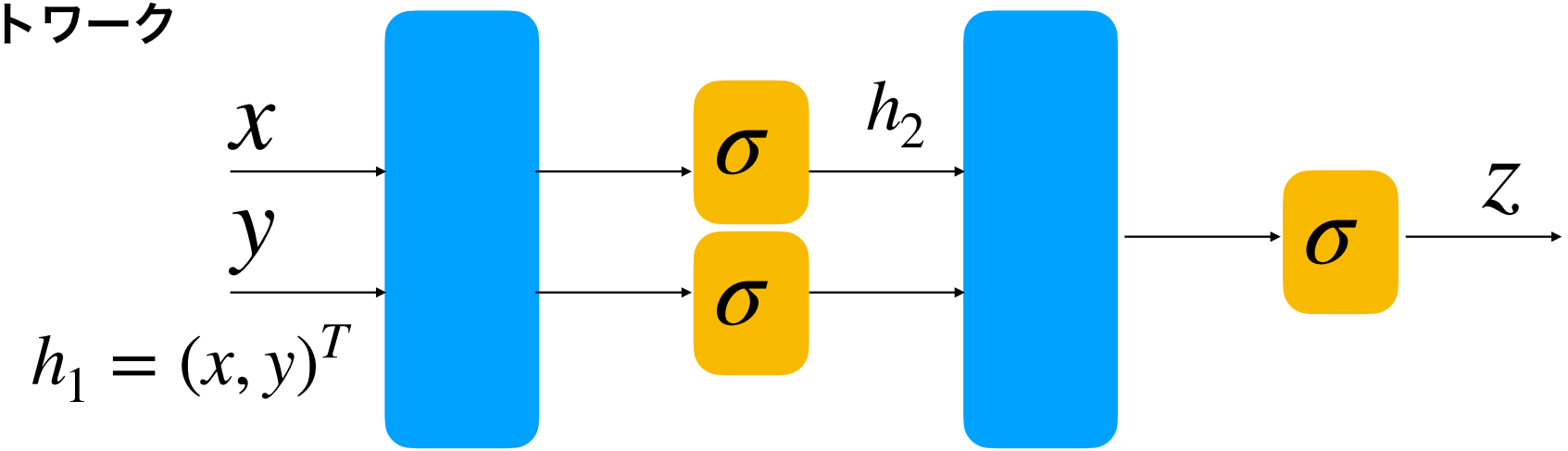
データセット: {(0,0), 0}, {(0,1),0}, {(1,1),1},…

今回使うニューラルネットワーク(NN)モデル

$$W_1 \in \mathbb{R}^{2\times 2} \qquad W_2 \in \mathbb{R}^{1\times 2}$$

$$W_1 h_1 + b_1 \qquad W_2 h_2 + b_2$$

2層のネットワーク

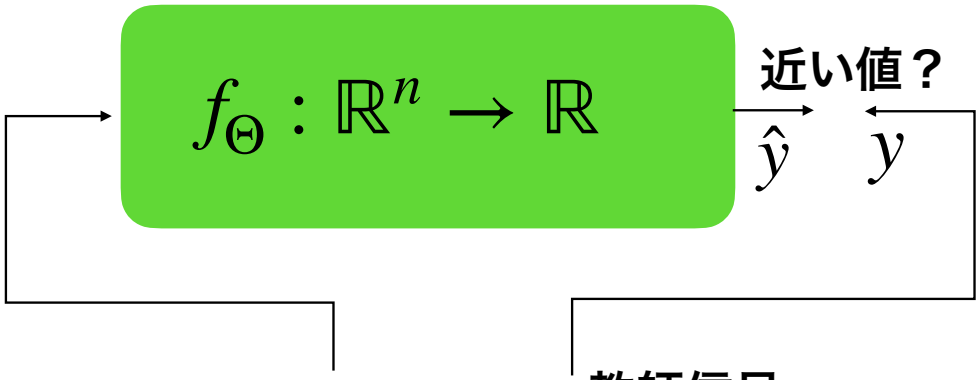


シグモイド関数　σ

$\{W_1, W_2, b_1, b_2\}$ 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



{(0,0), 0}
{(0,1), 0}
{(1,1), 1}
…

ここでは二乗誤差関数を利用

AND学習のコード(NN定義部)

le display

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

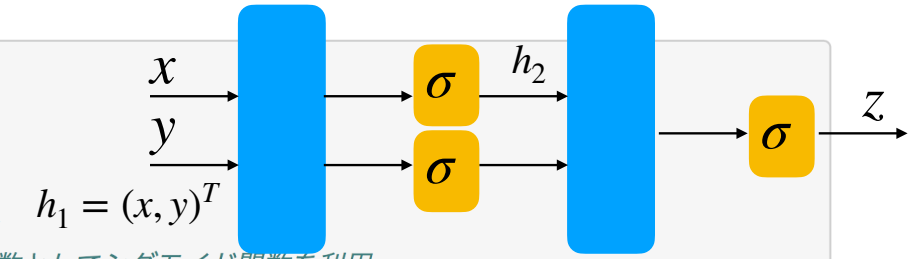
In [ ]:
```
mbs = 5 # ミニバッチサイズ
```

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
```

2層



前向き計算をforwardに書く

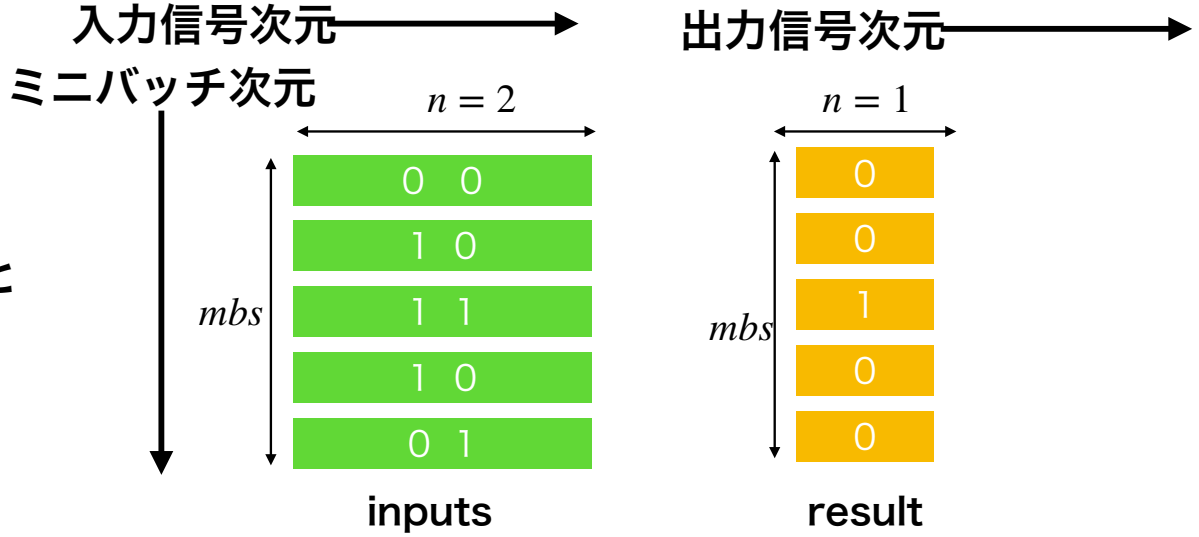
ミニバッチ生成部

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)
```

ミニバッチは
2次元テンソル
(当面２次元配列と
考えればよい)



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)
    optimizer.step()  # 学習可能パラメータの更新
    if i % 100 == 0:
        print('i =', i, 'loss =', loss.item())
```

バックプロップ

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

- 学習プロセス(復習)
- PyTorchを使う
- AND関数学習のコードを学ぶ