

エッジで深層学習やれちゃう未来？  
Forward ForwardでNN学習してみる

東京大学 D2

宮城竜大 (みやぎりょうた)

# About Me

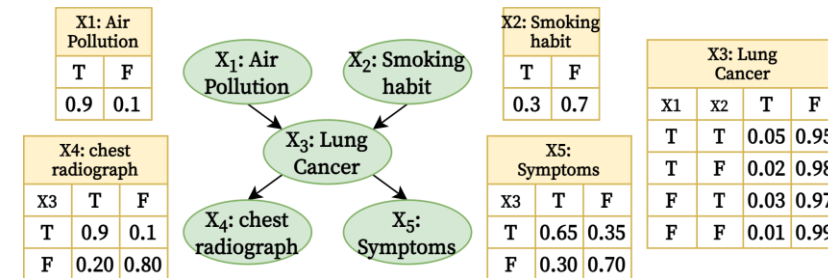
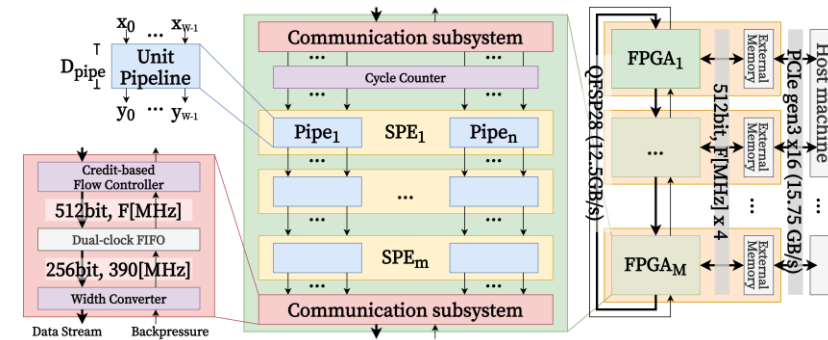
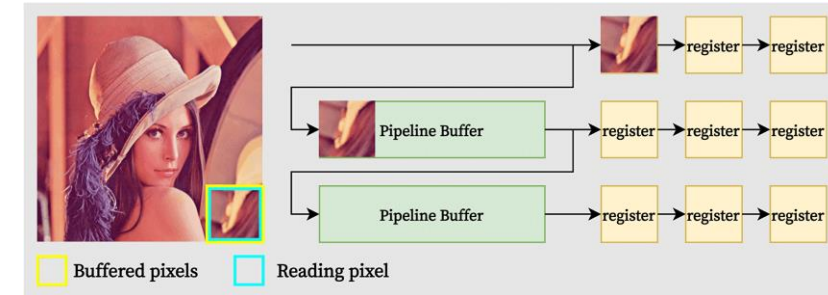
## • 来歴

- 2021年3月 京都大学 工学部情報学科卒業
- 2023年3月 // 情報学研究科修士課程修了
- 2023年4月 東京大学 情報理工学系研究科
  - 学部の指導教員が高瀬先生，修士は高木直史先生
- 2024年4月 日本学術振興会特別研究員(DC2)

## • 研究：「領域特化アーキテクチャで高速化！」

1. FPGAによる低遅延画像処理 (AMR, SmartNIC)
2. 複数FPGAデータフロー計算の性能モデリング
3. FPGAによるベイジアンネットワークの構造学習
4. 【WIP】ニューラルネットの学習...

• <https://urashima0429.github.io/>



# Motivation

- ~~TOPPERSカンファの後，飲みの席で適当に頷いていたら生えた~~
- 現在のNNの学習は誤差逆伝播法という手法が広く使われている。
  - 勾配の連鎖律を利用して各層のパラメータの勾配を効率的に計算
  - しかし，代償として，大域的なデータ依存性と非常に高いメモリ要求を伴う。
  - 結果として，エッジデバイスでのNN学習は難しい。
  - また，生物学的な妥当性が低い（脳内で実現が難しい）
- Forward Forward (FF)アルゴリズム
  - NNの権威であるHinton教授が提案する順伝播のみのNNの学習法
  - データフロー型の計算特性 => 省メモリ/局所参照 => エッジでもできるかも？
- FFの詳細とその可能性を小規模タスクで評価してみる
  - 結果は...？

# Neural Network (NN)

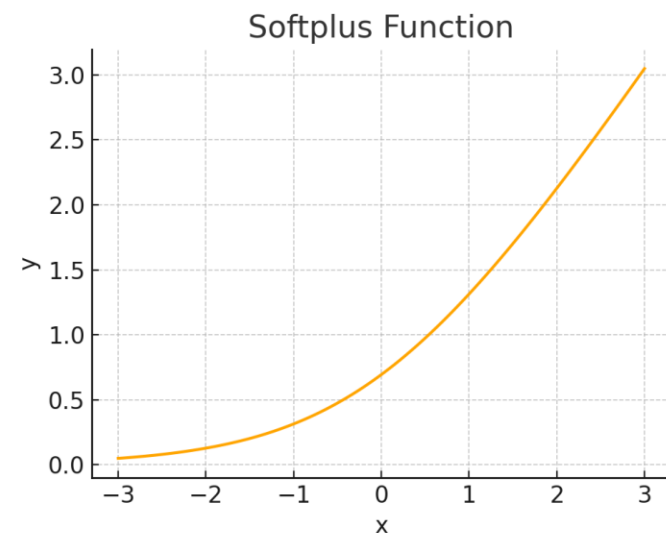
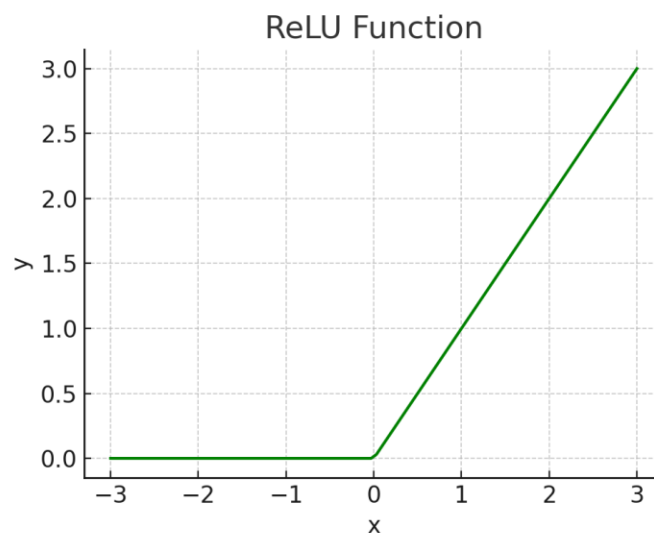
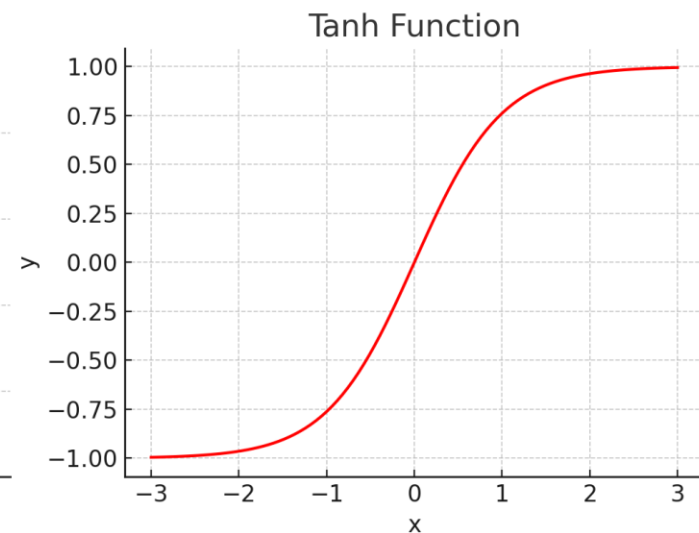
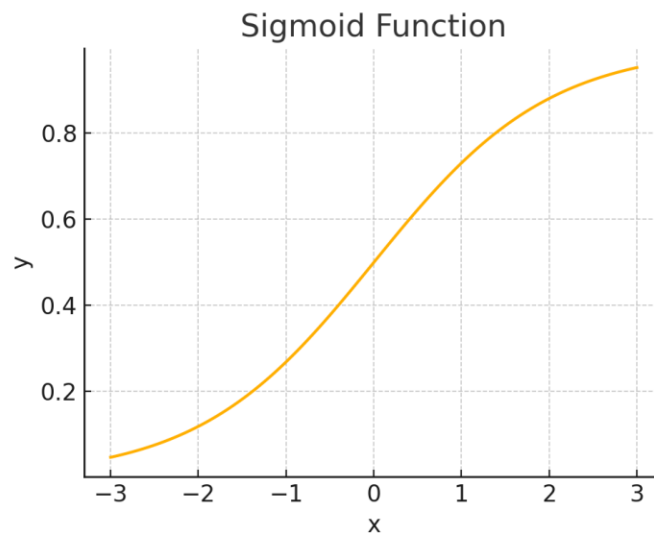
- 線形変換と非線形変換を交互に何層も重ねて関数を表現 (= 近似)

$$\mathbf{x}^{(l+1)} = f(\mathbf{W}^{(l)} \mathbf{x}^{(l)})$$

- $\mathbf{W}^{(l)} \in \mathbb{R}^{N^{(l+1)} \times N^{(l)}}$  は線形変換,  $f$  は要素毎の非線形活性化関数
- なぜこの形がいいの...? (私なりの理解だと)
  - 複雑な関数を簡単な関数の組合せ, 重ね合わせで表現したい
  - 一番簡単な関数は線形変換, ただし線形変換は何度重ねても一つの線形変換
  - そこで非常にシンプルな非線形変換を間に挟む (次頁)
- NNのすごさ
  - NNは万能で効率的でデカくても学習がうまくいく (次々頁)
  - なんで?

# よく使われる非線形変換

- Sigmoid( $x$ ) =  $\frac{1}{1+e^{-x}}$
- tanh( $x$ ) =  $\frac{e^x - e^{-x}}{e^x + e^{-x}}$
- ReLU( $x$ ) =  $\max(x, 0)$ 
  - 計算効率的
- Softplus( $x$ ) =  $\log_e(1 + e^x)$ 
  - ReLUの平滑化



# NNのすごさ(1): 万能近似定理

- NNに十分な幅があれば重要な関数クラスを任意の精度で近似できる
- 直感的な証明
  1. Sigmoid関数は1つのtanh関数または、2つのSoftplus関数で表現できる
    - $$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}} = \frac{1}{2} \cdot \frac{2}{1+e^{-x}} = \frac{1}{2} \cdot \frac{2e^{\frac{x}{2}}}{e^{\frac{x}{2}}+e^{-\frac{x}{2}}} = \frac{1}{2} \left( \frac{e^{\frac{x}{2}}-e^{-\frac{x}{2}}}{e^{\frac{x}{2}}+e^{-\frac{x}{2}}} + \frac{e^{\frac{x}{2}}+e^{-\frac{x}{2}}}{e^{\frac{x}{2}}+e^{-\frac{x}{2}}} \right) = \frac{1}{2} \left( \tanh\left(\frac{x}{2}\right) + 1 \right)$$
    - $$\text{Sigmoid}(x) = \frac{1}{1+e^{-x}} = \frac{1}{2} \log_e \left( \frac{1+e^{x+1}}{1+e^{x-1}} \right) = \frac{1}{2} (\text{Softplus}(x+1) - \text{Softplus}(x-1))$$
  2. Sigmoid関数で階段関数をいくらでもよく近似できる
  3. 階段関数を近似できれば、Sin, Cosをいくらでもよく近似できる (?)
  4. Sin, Cosが近似できれば、Fourier(逆)変換ができる
  5. 任意の連続関数が近似できる
- ただし学習できるとは言っていない (十分な幅が指数的になる等)

# NNのすごさ(2): 深さと表現力

- 万能近似能力という意味では浅層で十分だが実際は多層, なぜ?
- NNの表現力
- = 何個の領域に分割できるか
  - 幅に対して多項式的
  - 深さに対して指数式的
- 浅くても万能だが,
- 深いほうが効率的

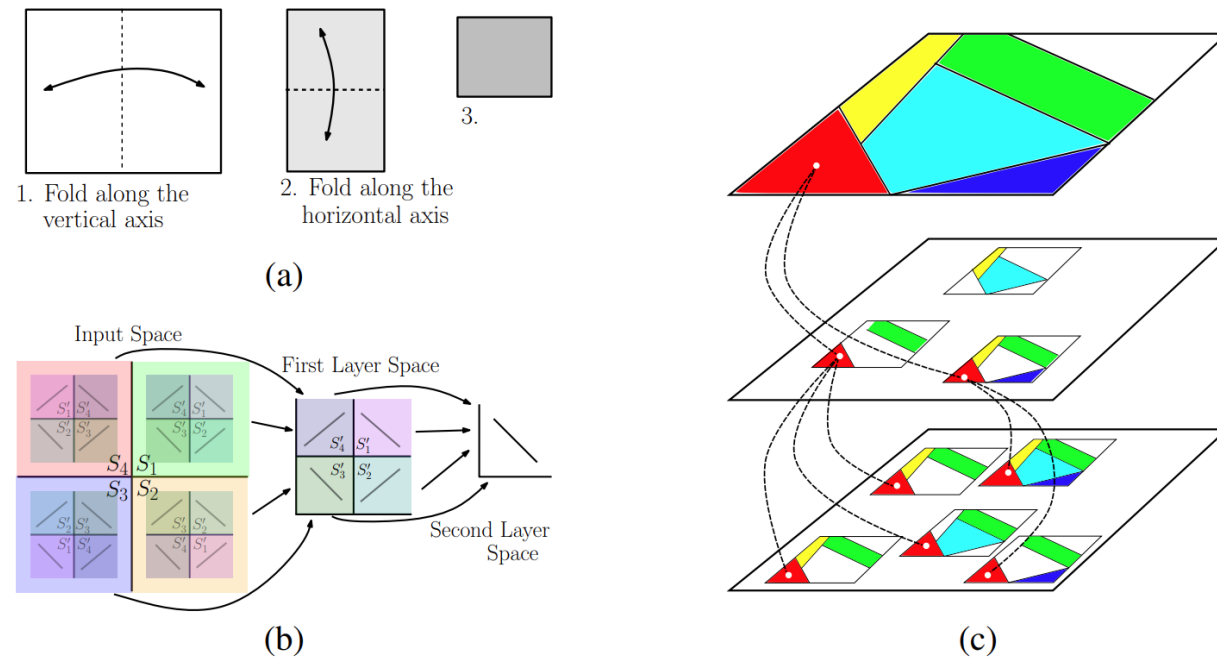


Figure 2: (a) Space folding of 2-D Euclidean space along the two axes. (b) An illustration of how the top-level partitioning (on the right) is replicated to the original input space (left). (c) Identification of regions across the layers of a deep model.



Montufar, Guido F., et al. "On the number of linear regions of deep neural networks." Advances in neural information processing systems 27 (2014).

# NNのすごさ(3): 宝くじ仮説 (Lottery Ticket Hypothesis)

- NNの学習に関して，以下が経験則的に知られていた
  1. 古典的な統計学習理論に反して，パラメータ数が過剰でも過学習しない
  2. 枝刈りによって精度を落とさずにパラメータ数をある程度まで削減できる
    - 枝刈り：絶対値が小さい（つまり重要度が低いと思われる）パラメータを削減
  3. ただし，最初から削減したパラメータ数では同程度の性能を得られない
- 宝くじ仮説「NN内にそれ単独で同程度学習させても同等の性能を発揮する部分NN（＝当たりくじ）が存在する」
- 直感的な説明
  - パラメータ数の増加に対し，部分NNの選び方は指数的に増える
  - => 運良く高い収束性能と汎化性能を持つ部分NNを含む確率が高くなる
  - => よい部分NNが選ばれて学習が進む，関係ない部分は学習が進まない
  - => 過剰なパラメータ数を持っていても学習・汎化がうまくいく
- NNは学習も大きければ大きいほどよい



# NNの学習

- 基本的には、各層の線形変換 $W^{(l)}$ を勾配法で調節して学習する
  - Note: 一部非線形変換も学習する場合もある (Parametric ReLUなど)
- 線形変換にランダム摂動を与えて損失関数の増減をみる (強化学習)
  - もちろん、スケールしない、大規模なものは作れない
- 誤差関数 $E$ に対する各層の線形変換の勾配 $\frac{\partial E}{\partial W^{(l)}}$ をどのように求める？
- => 誤差逆伝播法 (Backpropagation, BP)
  - 連鎖律を利用して誤差に対する各層の線形変換 $W^{(l)}$ の勾配を効率的に計算 
  - 現在のDNNは誤差逆伝播法の汎用性と学習効率の高さに依存している
  - 具体的にはどんな感じ？ (次頁) 

# 順伝播（以降の話ですべて共通）

- $\mathbf{W}^{(l)}$ :  $l$ 層目の線形変換
- $f^{(l)}$ :  $l$ 層目の非線形変換（活性化関数）
- $\mathbf{x}^{(l)}$ :  $l$ 層目の出力ベクトル（ $l+1$ 層目の入力ベクトル）
- $\mathbf{y}^{(l)}$ :  $l$ 層目の線形変換適用後，活性化関数適用前のベクトル
- $l$ 層目の順伝播

$$\mathbf{y}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l-1)}, \quad \mathbf{x}^{(l)} = f^{(l)}(\mathbf{y}^{(l)})$$

- 誤差関数  $E$

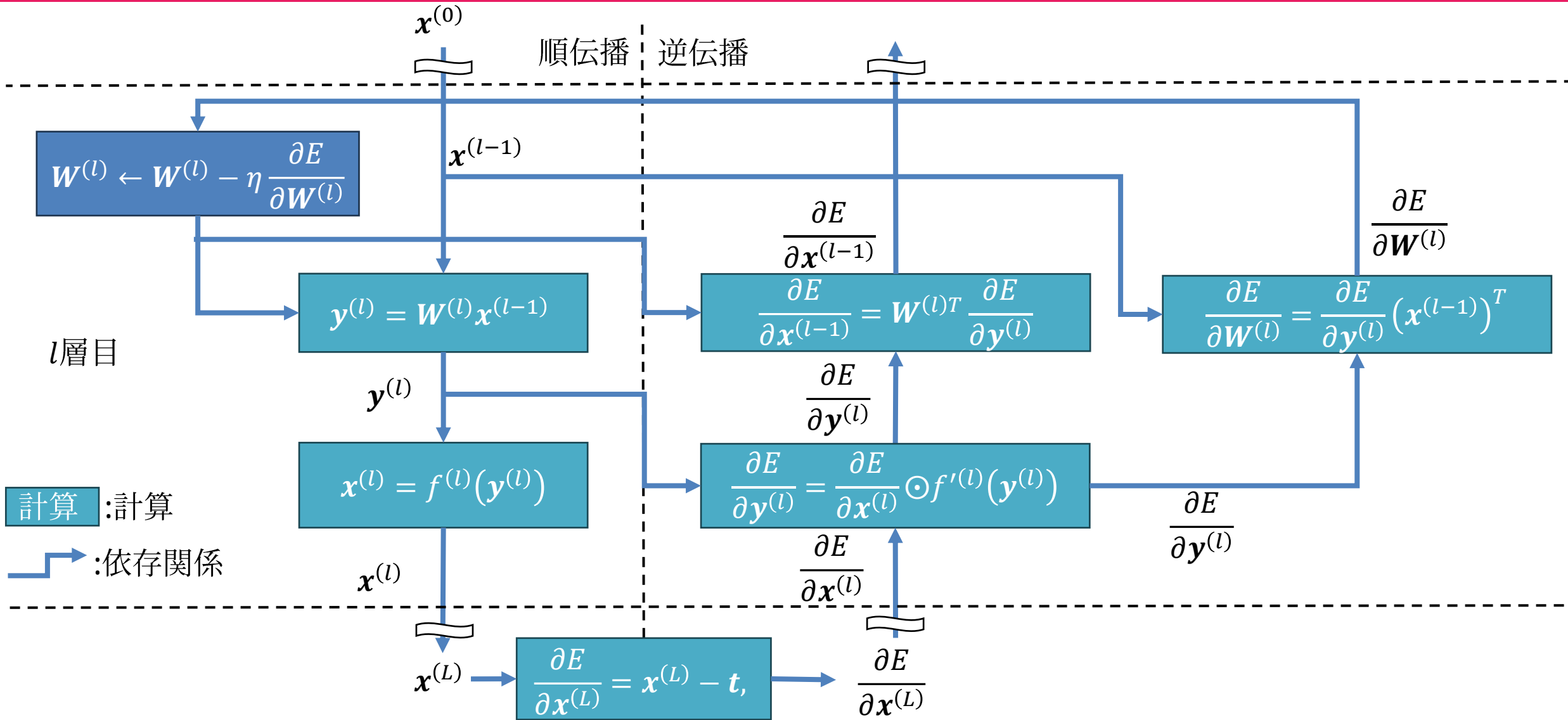
$$E = \frac{1}{2} \|\mathbf{x}^{(L)} - \mathbf{t}\|^2 = \frac{1}{2} \sum_n \left( x_n^{(L)} - t_n \right)^2$$

# 逆伝播

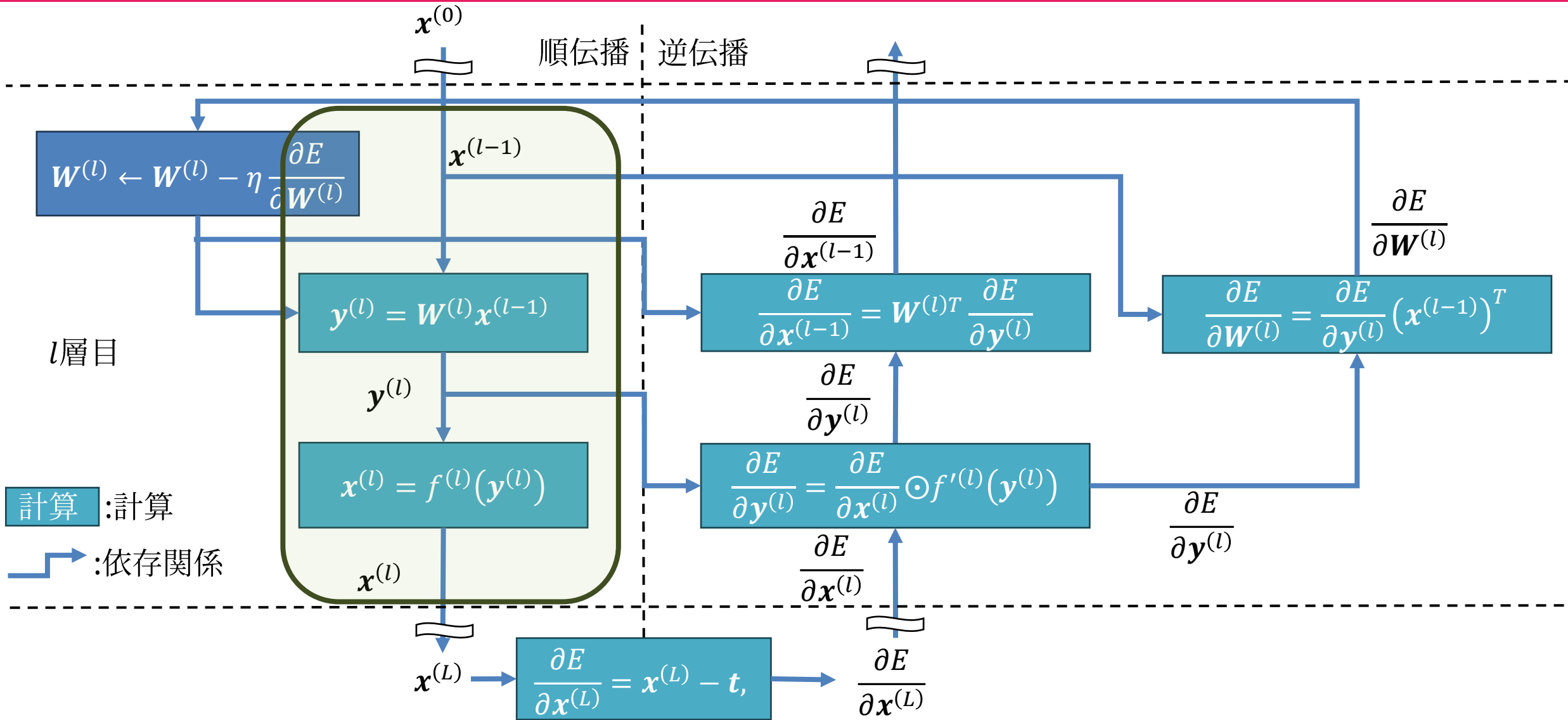
	順伝播	逆伝播
$L$ 層目	$E = \frac{1}{2}  \mathbf{x}^{(L)} - \mathbf{t} ^2$	$\frac{\partial E}{\partial \mathbf{x}^{(L)}} = \mathbf{x}^{(L)} - \mathbf{t},$
$l \in [0, \dots, L]$ 層目	$\mathbf{x}^{(l)} = f^{(l)}(\mathbf{y}^{(l)})$	$\frac{\partial E}{\partial \mathbf{y}^{(l)}} = \frac{\partial E}{\partial \mathbf{x}^{(l)}} \odot f'^{(l)}(\mathbf{y}^{(l)})$
	$\mathbf{y}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l-1)}$	$\frac{\partial E}{\partial \mathbf{W}^{(l)}} = \frac{\partial E}{\partial \mathbf{y}^{(l)}} (\mathbf{x}^{(l-1)})^T$
	$\mathbf{y}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l-1)}$	$\frac{\partial E}{\partial \mathbf{x}^{(l-1)}} = \mathbf{W}^{(l)T} \frac{\partial E}{\partial \mathbf{y}^{(l)}}$

$$\mathbf{W}^{(l)} \leftarrow \mathbf{W}^{(l)} - \eta \frac{\partial E}{\partial \mathbf{W}^{(l)}}$$

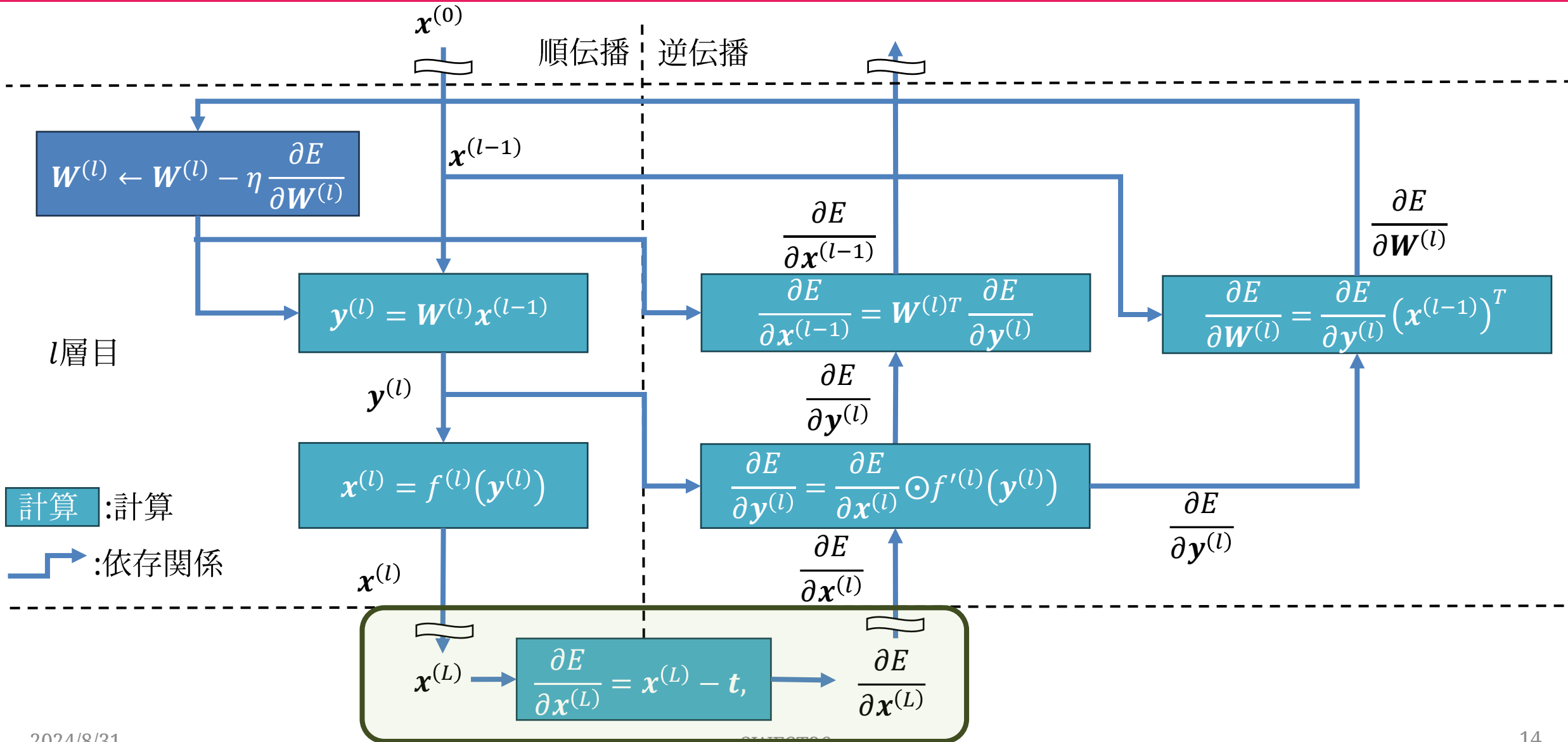
# 誤差逆伝播法 (Backpropagation, BP)



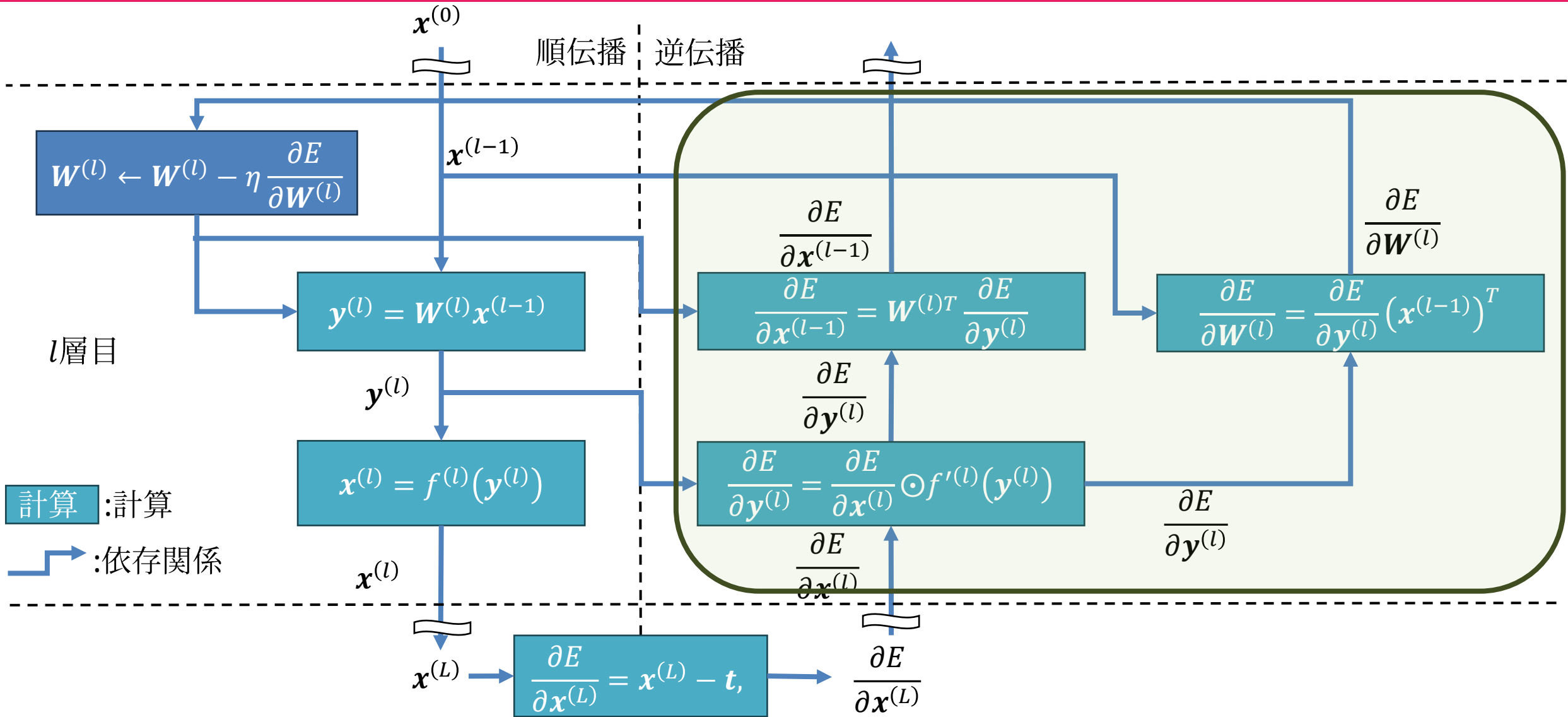
# 誤差逆伝播法 (Backpropagation, BP)



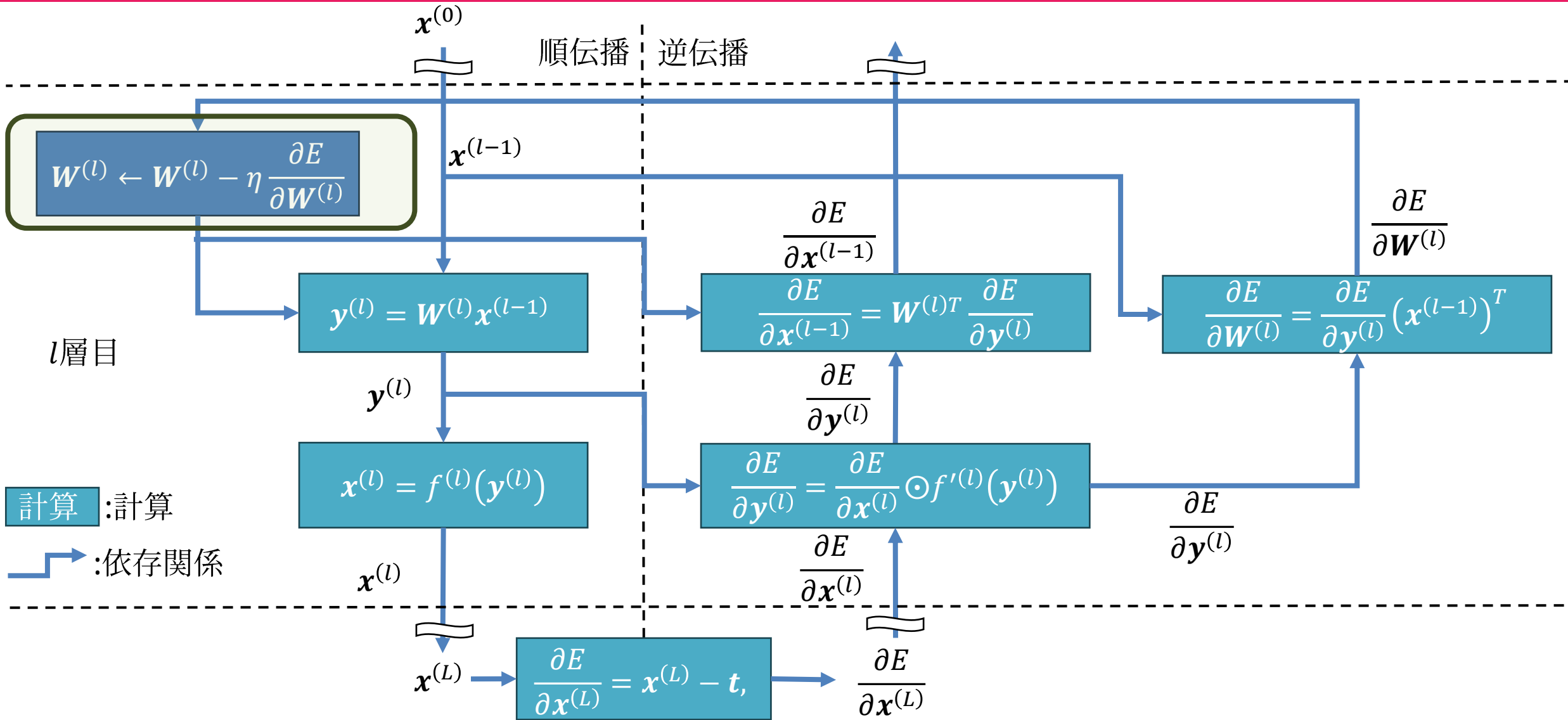
# 誤差逆伝播法 (Backpropagation, BP)



# 誤差逆伝播法 (Backpropagation, BP)



# 誤差逆伝播法 (Backpropagation, BP)





# 誤差逆伝播法 (Backpropagation, BP)

- 😊 勾配の連鎖律を利用して各層の線形変換 $W^{(l)}$ の勾配を効率的に計算
  - => 実質的にはBP一択
- BPにもマイナスはある
  - 😞 順伝播時の $x^{(l)}, y^{(l)}, W^{(l)}$ に依存, 必要なメモリとその帯域幅が大
  - 😞 各層の計算が逐次的, パイプライン並列性の制限
  - 😞 生物学的な妥当性がない (別になくてもいいけど, 次頁)

# 生物学的な妥当性がない

- BPは脳内では難しいと考えられている
  - 順伝播時に使用した重みの逆行列が必要
  - 出力層からの誤差が逆伝播されるまで、神経活動を保持する必要
  - 順方向と逆方向の計算が交じらないようにモデル全体の大域的な同期が必要
  - シナプスの接続は一方向的で正確に対応する逆方向パスが考えにくい
  - アナログ回路ではノイズによって誤差が消失する可能性が高い
  - Etc...
- 注：そもそもANNが正確に脳の仕組みを模倣しているわけではない

AI技術の最前線 これからのAIを読み解く先端技術73, 2-1 脳内で誤差逆伝播法が起きているか, 岡之原大輔

# Backpropagation and the brain (Nature, 2020)

- 生物学的妥当性の観点からのBPの代替学習手法が提案されている
- 学習効率のスペクトラム (弱 => 強)
  - Hebbian learning: 発火に寄与すれば結合を強化
  - Perturbation learning: ランダム摂動に対する変化を見る, スケールしない
  - (???)
  - Backpropagation: 正確な誤差を伝播させる, ただし脳では難しそう
- 実際はもっとうまくやるアルゴリズムが存在する可能性を示唆
- ちなみにHinton教授も共著者

Lillicrap, Timothy P., et al. "Backpropagation and the brain." Nature Reviews Neuroscience 21.6 (2020): 335-346.

## ここまでのまとめ

- 線形変換と非線形変換を何層も重ねて関数を表現（=近似）する
  - 万能近似定理，理論的にはなんでも近似できるよ！
  - 深さに対して指数的な表現力を持つよ！
  - 宝くじ仮説，でかくすればするほど学習もうまくいくよ！
- NNの学習は誤差逆伝播法（Backpropagation, BP）一択
  - 🥰連鎖律を利用して誤差 $E$ に対する各層の線形変換の勾配 $\frac{\partial E}{\partial W^{(l)}}$ を効率的に計算
  - 😞順伝播時の $\mathbf{x}^{(l)}, \mathbf{y}^{(l)}, \mathbf{W}^{(l)}$ に依存，必要なメモリと帯域幅が大
  - 😞各層の計算が逐次的，パイプライン並列性の制限
  - 😞生物学的な妥当性がない（別になくてもいいけど）
    - 実際はもっとうまくやるアルゴリズムが存在する可能性を示唆
- BP以外でNNの学習ができないか...??

# the Forward-Forward Algorithm (FF)

- 多くのデータで確率的勾配降下することが近年のDNNの成功をもたらした
  - 勾配は通常、BPを用いて計算される
- しかし、脳は本当に誤差逆伝播を行っている...??
  - △大脳新皮質が誤差的なものを伝播させたり、神経活動を保存するという証拠はない
  - △タイムアウトなく感覚入力に対処するにはパイプライン的な推論と学習機構が必要
  - △正しい導関数を計算するために順方向パスの計算結果を知っておく必要がある
- じゃあ、誤差に頼らずに強化学習してみる...?
  - 重みや神経活動にランダムな摂動を与え、その摂動と目的関数の変化を関連付ける
  - もちろん、スケールしない、大規模なものには作れない
- 誤差逆伝播せずに勾配法的に学習するには...?
  - 各層で誤差を決めてしまえばいいのでは...? => **Forward Forward**

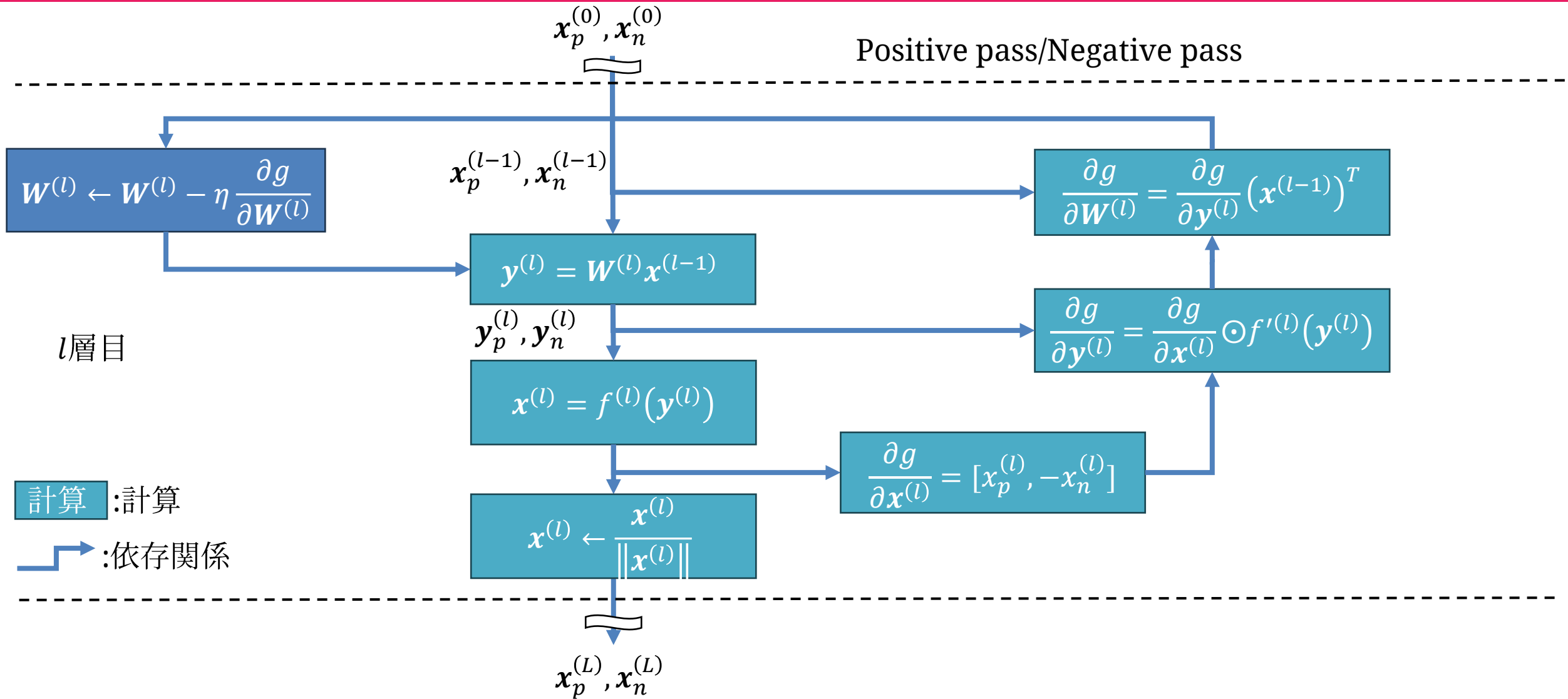
The Forward-Forward Algorithm: Some Preliminary, Geoffrey Hinton (Google Brain), Neural IPS 2022

# the Forward-Forward Algorithm (FF)

- 入力为正例かダミーかの2値を分類するタスクに変換
- ラベルも一緒に入力し，順方向のみで学習
- 層の良さ  $g(x^{(l)})$  が正例/ダミーではある閾値以上/以下に
  - Positive path: 正データの特徴ベクトル  $x_p^{(l)}$  を基に層の良さ  $g(x_p^{(l)})$  を増加
  - Negative path: **ダミーデータ**の特徴ベクトル  $x_n^{(l)}$  を基に層の良さ  $g(x_n^{(l)})$  を減少
  - 例えば，神経活動の二乗和，  $g(x^{(l)}) = \|x^{(l)}\|^2 = \sum_i (x_i^{(l)})^2$
- 特徴ベクトル  $x^{(l)}$  は正規化され，次の層は向きだけを使用する

The Forward-Forward Algorithm: Some Preliminary, Geoffrey Hinton (Google Brain), Neural IPS 2022

# Forward Forward (FF)



# the Forward-Forward Algorithm (FF)

1. BPと違い順伝播時の $\mathbf{x}^{(l)}, \mathbf{y}^{(l)}, \mathbf{W}^{(l)}$ の履歴を保持する必要がない
    - パラメータ自体を除いて、流れたあとは捨てていい、非常に省メモリ
    - 電力消費はメモリが大きいので、省電力にも資する
  2. パイプライン並列性が無制限
    - 各層を完全に別のマシンで分担、通信遅延も隠蔽できる
  3. 推論しながらタイムアウトなしで学習できる
    - 多様な動作環境で適応的に学習できる
- 組み込み/IoTでめっちゃうれしい

The Forward-Forward Algorithm: Some Preliminary, Geoffrey Hinton (Google Brain), Neural IPS 2022



# the Forward-Forward Algorithm (FF)

## Forward-Forward (FF)

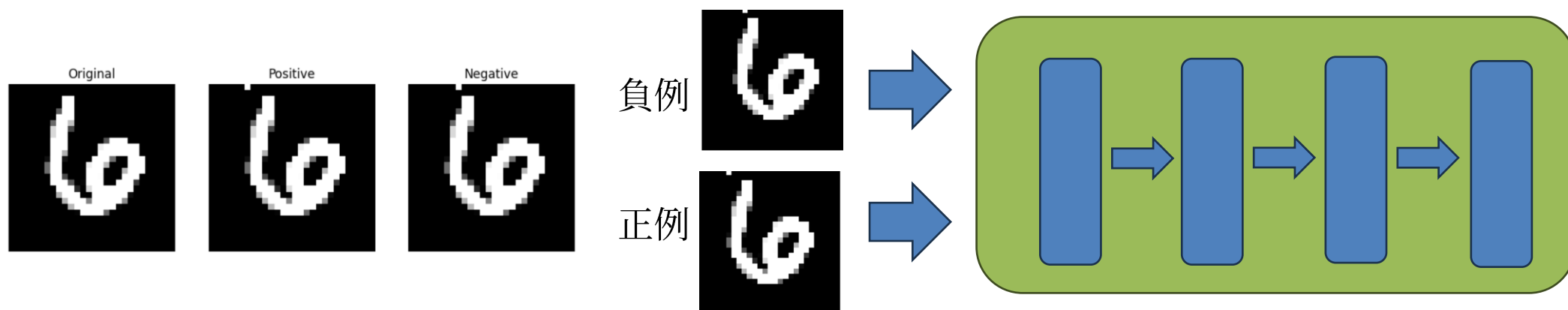
- 各層が独立して学習, 推論
- ラベル/誤差は各層で決まる
- 計算依存が少なく省メモリ
- パイプライン並列性が無制限
- タイムアウトなしで学習可

## Backpropagation (BP)

- 全体で関数を学習, 推論
- ラベル/誤差は出力層で決まる
- 必要なメモリと帯域幅が大
- パイプライン並列性が制限される
- 逆伝播時に推論はタイムアウト

# 教師アリFF

- ラベル情報を画像に埋め込む
  - 正データ：6の画像には6番目のピクセルをassertする
  - ダミーデータ：画像とは異なる番目のピクセルをassertする



The Forward-Forward Algorithm: Some Preliminary, Geoffrey Hinton (Google Brain), Neural IPS 2022

# 教師ありFFの元論文の結果

- MNIST, 全結合層 (空間レイアウトに関する情報なし)
- 2000ReLU x 4層, 60エポック, test error は1.36%
  - BPは約20エポックで同様のtest errorを達成する
- FFの学習率を2倍にし、40エポックで, test errorは1.46%
- Data augmentationで畳み込み的なことを考える
  - 各方向に最大2ピクセルずつ画像をずらす, 各画像に対して25種類のversion
  - 全結合でもピクセルの空間レイアウトに関する知識を埋め込める
  - 500エポック, test error が0.64%, CNNをBPで学習した場合と同程度

The Forward-Forward Algorithm: Some Preliminary, Geoffrey Hinton (Google Brain), Neural IPS 2022

# 教師なしFF

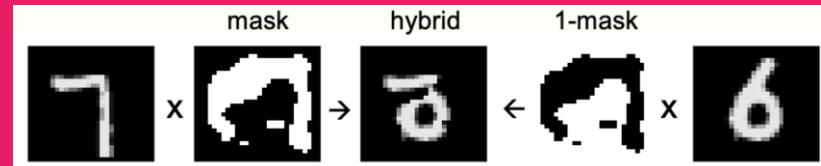
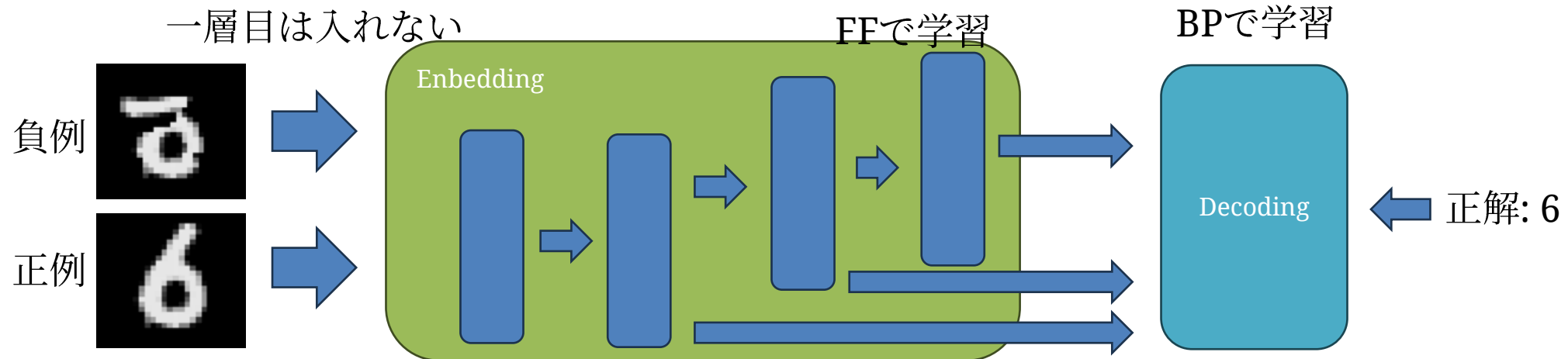


Figure 1: A hybrid image used as negative data

- 教師なしFFでEncoderを学習，教師ありBPでDecoderを学習
- 局所はそれっぽいが全体見ると違うダミーデータ
  - ランダムなビット画像から始めて，水平方向と垂直方向に[1/4,1/2,1/4]のフィルタを使ってぼかしを繰り返した後，0.5で閾値処理してマスクを作成



The Forward-Forward Algorithm: Some Preliminary, Geoffrey Hinton (Google Brain), Neural IPS 2022

## 教師なしFFの元論文の結果

- MNIST, 全結合層 (空間レイアウトに関する情報なし)
- 2000ReLU x 4層, 100エポック, でtest error率は1.37%
- 最初の隠れ層の出力を使用するとテスト性能が悪化 (なんで?)
- Local receptive fields (without weight-sharing)で性能向上
  - 畳み込み層の各位置で異なるフィルタを使用する

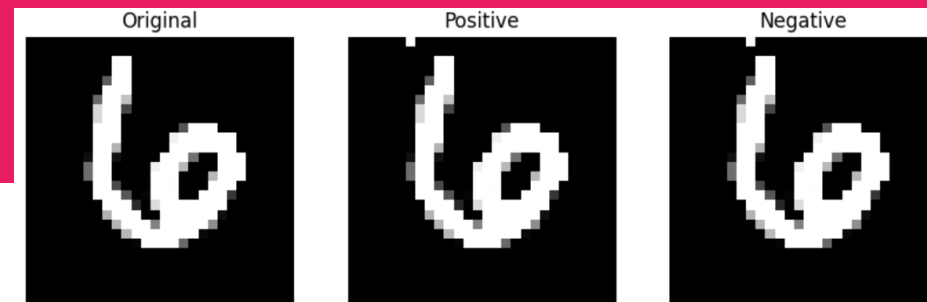
The Forward-Forward Algorithm: Some Preliminary, Geoffrey Hinton (Google Brain), Neural IPS 2022

# 教師ありFFの検証

- FFは本当に学習できるのか？
- Loss関数の定義の検討
- スレッシュホールドも学習できる…？
- 各層を並列に学習できる？
- 枝刈りできる？
- 層を深くしてメリットがある…？

# 教師ありFFの検証, 実験設定

- NVIDIA GeForce RTX 4070 Laptop GPU
- 対象データ
  - MNIST (0~9までの手書き文字, 28x28x1)
  - Fashion-MNIST (10種の衣服の画像, 28x28x1)
  - CIFAR-10 (10種の乗り物とか生き物の画像, 32x32x3)
- ラベルの埋め込みは右上図, 全結合層のみ, 活性化関数はReLU
- 推論: 各層の良さの平均を最も高くするラベル入力
  - ラベルごとに推論が必要 (10クラス分類なら10回推論する)
- コード ([Github](#))
  - 多分Dockerが動けばOK, GPUがあるとなおよし



# 教師ありFFの検証

- 基底クラス (本質)
- 閾値もパラメータ
  - 予備実験では性能Good
  - 元論文では不明
- 誤差の定義 (次項)
- 層のメンバ関数として
  - 誤差の定義
  - 誤差の逆伝播
  - パラメータの更新
- detach()
  - 自動微分の計算グラフ切断

```
class FFBase(nn.Module):
    def __init__(self):
        super(FFBase, self).__init__()
        self.initial_threshold = torch.tensor([5.0])
        self.threshold = nn.Parameter(self.initial_threshold)

    def forward(self, x):
        raise NotImplementedError("Each subclass must implement the forward method.")

    def goodness(self, activations):
        raise NotImplementedError("Each subclass must implement the goodness method.")

    def loss(self, g_pos, g_neg):
        inputs = torch.cat([
            # (g_pos - self.threshold), -(g_neg - self.threshold) # min-ssq
            -(g_pos - self.threshold), (g_neg - self.threshold) # max-ssq
        ])
        # losses = F.relu(inputs) # relu
        losses = torch.log(1 + torch.exp(inputs)) # softplus
        return losses.mean()

    def train(self, pos_x, neg_x):
        a_pos, a_neg = self.forward(pos_x), self.forward(neg_x)
        g_pos, g_neg = self.goodness(a_pos), self.goodness(a_neg)
        self.opt.zero_grad()
        loss = self.loss(g_pos, g_neg)
        loss.backward()
        self.opt.step()
        return loss, a_pos.detach(), a_neg.detach()
```

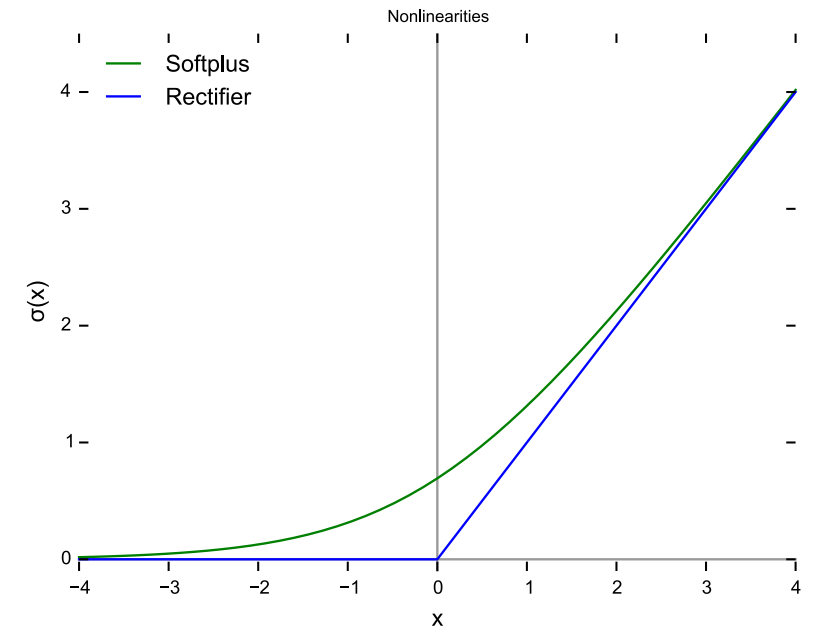


# 教師ありFFの検証

- 誤差の定義

- 正例に対する活性化が閾値を超えるようにするか下回るようにするか
  - 正直どっちでもOK, 性能差は見られず
- 正しい側に分けられた場合, lossはナシ,
- 誤った側に分けられた場合, lossは差分
- つまりReLU
- これを平滑化すると...?
  - 閾値に余裕をもって学習するようにできるはず

```
def loss(self, g_pos, g_neg):
    inputs = torch.cat([
        # (g_pos - self.threshold), -(g_neg - self.threshold) # min-ssq
        -(g_pos - self.threshold), (g_neg - self.threshold) # max-ssq
    ])
    # losses = F.relu(inputs) # relu
    losses = torch.log(1 + torch.exp(inputs)) # softplus
    return losses.mean()
```



# 教師ありFFの検証

- 活性化関数はReLU
  - 元論文通り
- 学習率は高め
  - 元論文ではBPの2倍
  - 今回はBP: 0.01, FF: 0.1
- Forward
  - 入力ベクトルを正規化
  - それ以外は普通
- 良さの定義
  - 論文に従って, 2乗平均

```
class FFLinear(nn.Linear, FFBase):
    def __init__(self, in_features, out_features, bias=True, device=None, dtype=None):
        super().__init__(in_features, out_features, bias, device, dtype)

        self.act_fun = torch.nn.ReLU()
        self.opt = Adam(self.parameters(), lr=0.1)

    # [batch_size][in_features] => [batch_size][out_features]
    def forward(self, x):
        x_direction = x / (x.norm(2, 1, keepdim=True) + 1e-4)
        return self.act_fun(
            torch.mm(x_direction, self.weight.T) +
            self.bias.unsqueeze(0))

    # [batch_size][out_features] => [batch_size]
    def goodness(self, activations):
        return activations.pow(2).mean(1)
```

# 教師ありFFの検証

- FF (左) , FF with 平滑化 (右)
  - 平滑化したほうが性能が良い

```
dataset=MNIST, learning_rate=0.1, batch_size=64, epochs=3, update_count=2811
```

```
SupervisedFF(  
  (layers): ModuleList(  
    (0): ForwardForward(  
      in_features=784, out_features=512, bias=True  
      (relu): ReLU()  
    )  
    (1): ForwardForward(  
      in_features=512, out_features=256, bias=True  
      (relu): ReLU()  
    )  
    (2): ForwardForward(  
      in_features=256, out_features=10, bias=True  
      (relu): ReLU()  
    )  
  )  
)
```

```
FF Sequential training starts ...
```

```
Epochs for Layer 1: 100%|██████████| 3/3 [00:25<00:00, 8.54s/it]  
Epochs for Layer 2: 100%|██████████| 3/3 [00:32<00:00, 10.77s/it]  
Epochs for Layer 3: 100%|██████████| 3/3 [00:37<00:00, 12.65s/it]
```

```
FF Sequential training completed in 95.88 seconds.
```

```
threshold: 5.0 => 0.08575159311294556
```

```
threshold: 5.0 => 0.015413613989949226
```

```
threshold: 5.0 => 0.08027050644159317
```

Train accuracy: 0.94, Test accuracy: 0.94

```
dataset=MNIST, learning_rate=0.1, batch_size=64, epochs=3, update_count=2811
```

```
SupervisedFF(  
  (layers): ModuleList(  
    (0): ForwardForward(  
      in_features=784, out_features=512, bias=True  
      (relu): ReLU()  
    )  
    (1): ForwardForward(  
      in_features=512, out_features=256, bias=True  
      (relu): ReLU()  
    )  
    (2): ForwardForward(  
      in_features=256, out_features=10, bias=True  
      (relu): ReLU()  
    )  
  )  
)
```

```
FF Sequential training starts ...
```

```
Epochs for Layer 1: 100%|██████████| 3/3 [00:28<00:00, 9.35s/it]  
Epochs for Layer 2: 100%|██████████| 3/3 [00:32<00:00, 10.83s/it]  
Epochs for Layer 3: 100%|██████████| 3/3 [00:38<00:00, 12.79s/it]
```

```
FF Sequential training completed in 98.90 seconds.
```

```
threshold: 5.0 => 10.811964988708496
```

```
threshold: 5.0 => 11.334756851196289
```

```
threshold: 5.0 => 12.138172149658203
```

Train accuracy: 0.97, Test accuracy: 0.96

# 教師ありFFの検証

- 各層を並列に学習することができるか？
  - 元論文では層ごとに学習してる（1層目学習，固定 => 2層目）
  - 層ごとFF（左），層並列FF（右），性能は下がるが，十分学習できる

```
dataset=MNIST, learning_rate=0.1, batch_size=64, epochs=3, update_count=2811
```

```
SupervisedFF(  
  (layers): ModuleList(  
    (0): ForwardForward(  
      in_features=784, out_features=512, bias=True  
      (relu): ReLU()  
    )  
    (1): ForwardForward(  
      in_features=512, out_features=256, bias=True  
      (relu): ReLU()  
    )  
    (2): ForwardForward(  
      in_features=256, out_features=10, bias=True  
      (relu): ReLU()  
    )  
  )  
)  
FF Sequential training starts ...  
Epochs for Layer 1: 100% ██████████ 3/3 [00:27<00:00, 9.16s/it]  
Epochs for Layer 2: 100% ██████████ 3/3 [00:33<00:00, 11.22s/it]  
Epochs for Layer 3: 100% ██████████ 3/3 [00:39<00:00, 13.17s/it]  
FF Sequential training completed in 100.69 seconds.  
threshold: 5.0 => 10.811964988708496  
threshold: 5.0 => 11.334756851196289  
threshold: 5.0 => 12.138172149658203
```

**Train accuracy: 0.97, Test accuracy: 0.96**

```
dataset=MNIST, learning_rate=0.1, batch_size=64, epochs=3, update_count=2811
```

```
SupervisedFF(  
  (layers): ModuleList(  
    (0): ForwardForward(  
      in_features=784, out_features=512, bias=True  
      (relu): ReLU()  
    )  
    (1): ForwardForward(  
      in_features=512, out_features=256, bias=True  
      (relu): ReLU()  
    )  
    (2): ForwardForward(  
      in_features=256, out_features=10, bias=True  
      (relu): ReLU()  
    )  
  )  
)  
FF Parallel training starts ...  
Epochs for all Layer: 100% ██████████ 3/3 [01:19<00:00, 26.48s/it]  
FF Parallel training completed in 79.44 seconds.  
threshold: 5.0 => 10.418304443359375  
threshold: 5.0 => 10.055129051208496  
threshold: 5.0 => 6.534235000610352
```

**Train accuracy: 0.94, Test accuracy: 0.94**

# 教師ありFFの検証

- 重みの絶対値の下位90%を枝刈りして推論してみる
  - 層ごとFF(lr=0.1) (左), 層並列FF(lr=0.1) (中), BP(lr=0.01) (下)
  - FFの方はほとんど精度が落ちない

dataset=MNIST, learning\_rate=0.1, batch\_size=64, epochs=3, update\_count=2811

```
SupervisedFF(
  (layers): ModuleList(
    (0): ForwardForward(
      in_features=784, out_features=512, bias=True
      (relu): ReLU()
    )
    (1): ForwardForward(
      in_features=512, out_features=256, bias=True
      (relu): ReLU()
    )
    (2): ForwardForward(
      in_features=256, out_features=10, bias=True
      (relu): ReLU()
    )
  )
)
FF Sequential training starts ...
Epochs for Layer 1: 100% ██████████ 3/3 [00:30<00:00, 10.20s/it]
Epochs for Layer 2: 100% ██████████ 3/3 [00:36<00:00, 12.25s/it]
Epochs for Layer 3: 100% ██████████ 3/3 [00:41<00:00, 13.99s/it]
FF Sequential training completed in 109.33 seconds.
threshold: 5.0 => 10.811964988708496
threshold: 5.0 => 11.334756851196289
threshold: 5.0 => 12.138172149658203
Using first 1 layers: Train accuracy: 0.95, Test accuracy: 0.94
Using first 2 layers: Train accuracy: 0.97, Test accuracy: 0.96
Using first 3 layers: Train accuracy: 0.97, Test accuracy: 0.96
prune_weight 0.9
Using first 1 layers: Train accuracy: 0.91, Test accuracy: 0.91
Using first 2 layers: Train accuracy: 0.92, Test accuracy: 0.92
Using first 3 layers: Train accuracy: 0.91, Test accuracy: 0.92
```

dataset=MNIST, learning\_rate=0.1, batch\_size=64, epochs=3, update\_count=2811

```
SupervisedFF(
  (layers): ModuleList(
    (0): ForwardForward(
      in_features=784, out_features=512, bias=True
      (relu): ReLU()
    )
    (1): ForwardForward(
      in_features=512, out_features=256, bias=True
      (relu): ReLU()
    )
    (2): ForwardForward(
      in_features=256, out_features=10, bias=True
      (relu): ReLU()
    )
  )
)
FF Parallel training starts ...
Epochs for all Layer: 100% ██████████ 3/3 [01:31<00:00, 30.39s/it]
FF Parallel training completed in 91.17 seconds.
threshold: 5.0 => 10.418304443359375
threshold: 5.0 => 10.055129051208496
threshold: 5.0 => 6.534235000610352
Using first 1 layers: Train accuracy: 0.94, Test accuracy: 0.94
Using first 2 layers: Train accuracy: 0.95, Test accuracy: 0.95
Using first 3 layers: Train accuracy: 0.94, Test accuracy: 0.94
prune_weight 0.9
Using first 1 layers: Train accuracy: 0.89, Test accuracy: 0.89
Using first 2 layers: Train accuracy: 0.86, Test accuracy: 0.86
Using first 3 layers: Train accuracy: 0.77, Test accuracy: 0.78
```

dataset=MNIST, learning\_rate=0.01, batch\_size=64, epochs=3, update\_count=2811

```
SupervisedBP(
  (layers): ModuleList(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): Linear(in_features=512, out_features=256, bias=True)
    (2): Linear(in_features=256, out_features=10, bias=True)
  )
  (activation): ReLU()
)
BP training starts ...
100% ██████████ 3/3 [00:14<00:00, 4.87s/it]
BP training completed in 19.06 seconds.
BP Model: Train accuracy: 0.9566, Test accuracy: 0.952
prune_weight 0.9
BP Model: Train accuracy: 0.6596166666666666, Test accuracy: 0.6629
```

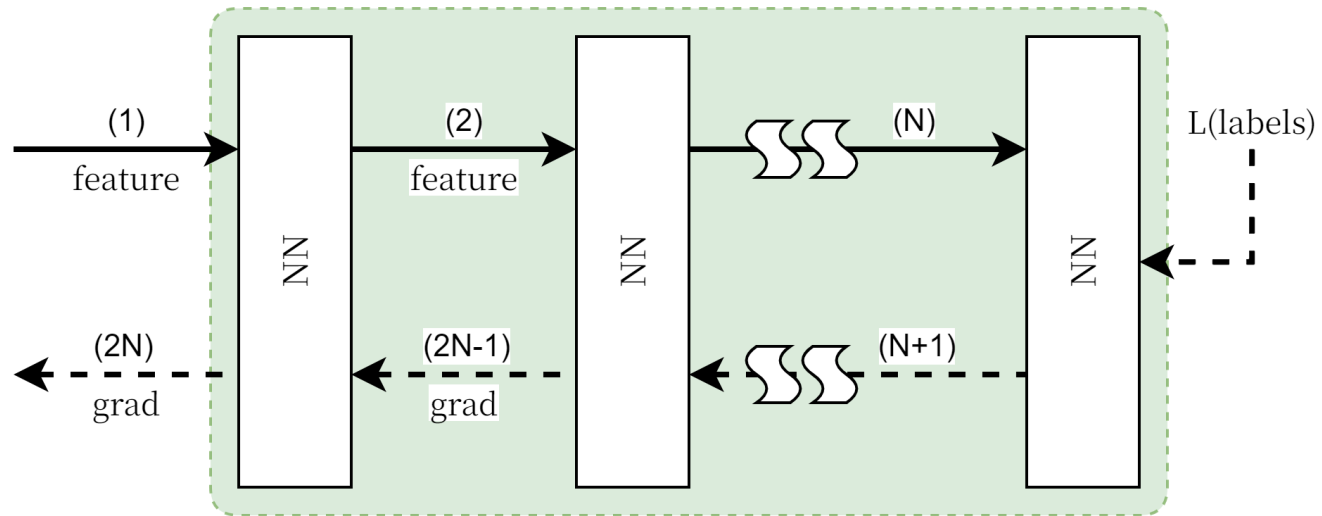
# 教師ありFFの検証

- FFで層を深くしてメリットがあるのか？
  - 特定の層の「良さ」のみを用いて推論
- ノート参照
  - 重ねることでちゃんと精度が上っているものの…？

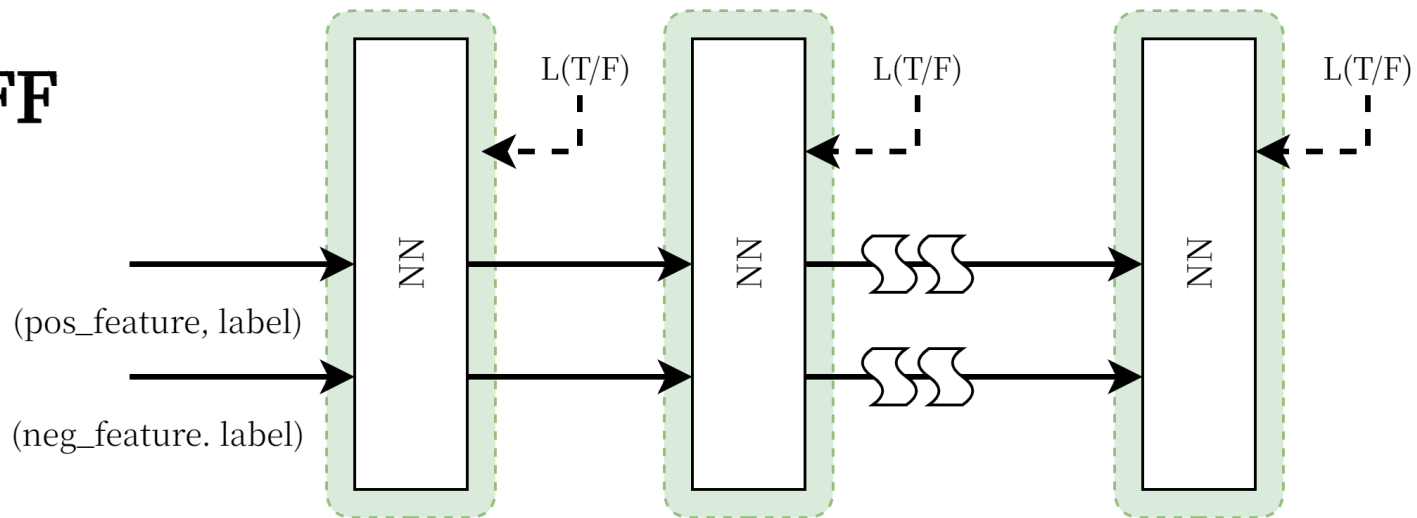
# FFを試した所感

- BP
  - ロスが全体を最適化
  - 🥰 階層的な表現
  - 🧐 計算の相互依存
- FF
  - 各層ごとのロスを最適化
  - 🥰 各層が独立
  - 🥰 枝刈り向いてる
  - 🧐 階層的な表現
- 結論, う〜ん
  - 畳み込み層

**BP**



**FF**



# 誤差逆伝播法以外で学習しようという流れ

- 各層ごとブロックごとに分割して学習できないか…？

- **Forward-Forward**

- **Block/Layer-wise Local Learning (右)**

- 誤差を注入するための補助NNを介して
- Block毎に誤差を注入し，局所的に学習

- **Synthesis Gradient**

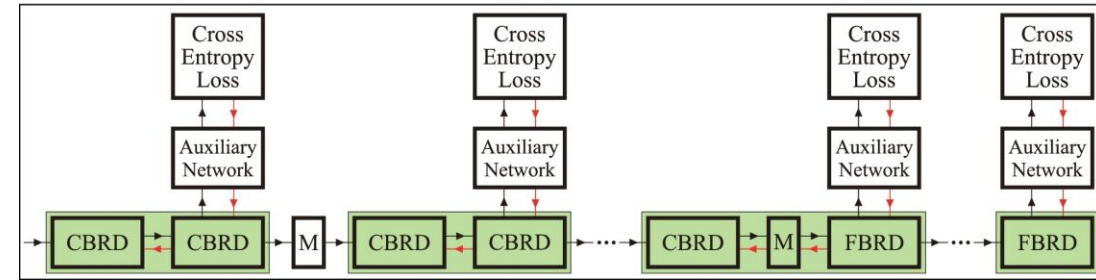
- 逆伝播される真の誤差勾配の代わりに合成勾配 (synthetic gradient)

- **Target Propagation**

- BPを緩和して近似的に再現できないか…？

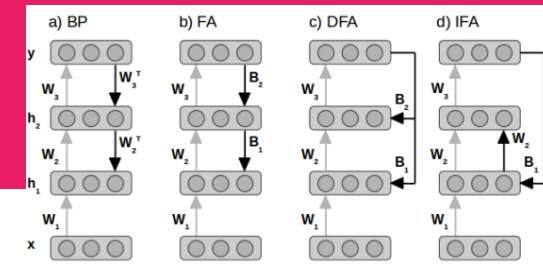
- **Feedback Alignment**

- **Uniform Sign-concordant Feedbacks**





# Feedback Alignment (FA)



- Feedback Alignment (FA)

- $W^{(l),T}$ を誤差の固定ランダム行列 $B_{FA}^{(l),T}$ にしても学習できる
- $B^{(l)}\delta^{(l)}$ と $W^{(l)T}\delta^{(l)}$ が90度以下 (i.e.,  $\delta^{(l)T}W^{(l)}B^{(l)}\delta^{(l)} > 0$ ) なら学習が進む
- 徐々に $W^{(l)}$ が $B^{(l)}$ にalignされ,  $\frac{\delta^{(l)T}W^{(l)}B^{(l)}\delta^{(l)}}{|W^{(l)T}\delta^{(l)}||B^{(l)}\delta^{(l)}|}$ が1に近づく

- Direct Feedback Alignment (DFA)

- 誤差を固定ランダム行列 $B_{DFA}^{(l)}$ で直接返しても学習できる

- Indirect Feedback Alignment (IFA)

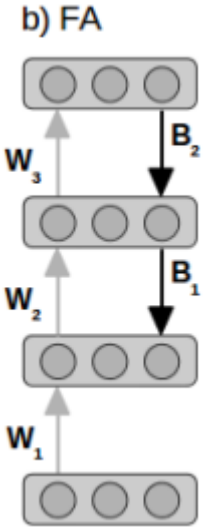
- 誤差を固定ランダム行列 $B_{IFA}$ で入力層に返し, 誤差も順伝播させる
- ただし, 動作例なし, 多分学習できない

Lillicrap, Timothy P., et al. "Random synaptic feedback weights support error backpropagation for deep learning." Nature communications 7.1 (2016): 13276.

Nøkland, Arild. "Direct feedback alignment provides learning in deep neural networks." Advances in neural information processing systems 29 (2016).

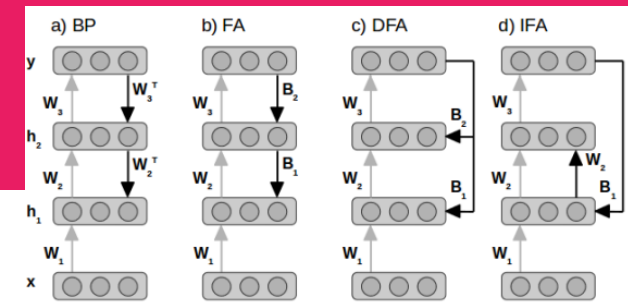
# How Important Is Weight Symmetry in Backpropagation?

- 重み輸送問題, BPの生物学的妥当性について
- 逆伝播の $\mathbf{W}^{(l)T}$ の正確性がどの程度性能に影響を与えるのか
- 15種類の分類データセット, ReLUで実験的に以下を示した
  1. 逆伝播時の重みの大きさは性能に関係ない
  2. 符号は重要で, 一致すればするほど良い
  3. 重みがランダムでも, 符号が同じ場合には同等の性能が得られる
  4. このような非対称的なBPがには, normalizations/stabilizationsが不可欠
    - Batch Normalization and/or Batch Manhattan
- つまり,  $\mathbf{W}^{(l),T}$ を $\text{sign}(\mathbf{W}^{(l),T})$ にしても同等の性能が得られる
  - Uniform Sign-concordant Feedbacks (uSF)



Liao, Qianli, Joel Leibo, and Tomaso Poggio. "How important is weight symmetry in backpropagation?." Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 30. No. 1. 2016.

# BPの緩和する方式について



• ただし、性能は相当劣るように見える

- <https://github.com/jsalbert/biotorch/blob/main/Benchmarks.md>
- FA, DFAは表現の自由度が下がるので、当然といえば当然

	順伝播	逆伝播	備考
BP	$\mathbf{y}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l-1)}$	$\frac{\partial E}{\partial \mathbf{x}^{(l-1)}} = \mathbf{W}^{(l)T} \frac{\partial E}{\partial \mathbf{y}^{(l)}}$	
FA	$\mathbf{y}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l-1)}$	$\frac{\partial E}{\partial \mathbf{x}^{(l-1)}} = \mathbf{B}_{FA}^{(l),T} \frac{\partial E}{\partial \mathbf{y}^{(l)}}$	学習はできるが精度が大幅減
DFA	$\mathbf{y}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l-1)}$	$\frac{\partial E}{\partial \mathbf{x}^{(l-1)}} = \mathbf{B}_{DFA}^{(l),T} \frac{\partial E}{\partial \mathbf{y}^{(l)}}$	学習はできるが精度が大幅減
IFA	$\mathbf{y}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l-1)}$	$\frac{\partial E}{\partial \mathbf{x}^{(l-1)}} = ???$	詳細なし、実際学習できるか不明
uSF	$\mathbf{y}^{(l)} = \mathbf{W}^{(l)} \mathbf{x}^{(l-1)}$	$\frac{\partial E}{\partial \mathbf{x}^{(l-1)}} = \text{sign}(\mathbf{W}^{(l),T}) \frac{\partial E}{\partial \mathbf{y}^{(l)}}$	精度も悪くない なんでうまくいくの...

# Key Takeaways

- NNは万能で効率的でデカくても学習できる
- BPはアーキテクチャフレンドリーではない点がある
  - 深さ方向に順/逆方向に相互に依存した逐次的な計算フロー
- BPはNNを学習するただ一つの方法ではない
  - BPの緩和/近似
  - 各層, ブロックごとに独立して誤差, 最適化
- Forward Forwardはそのうちのひとつ
  - データフロー型 (パイプライン並列性), 省メモリ, 高い参照局所性,
  - だが, あんまり重ねるメリットない気がする...

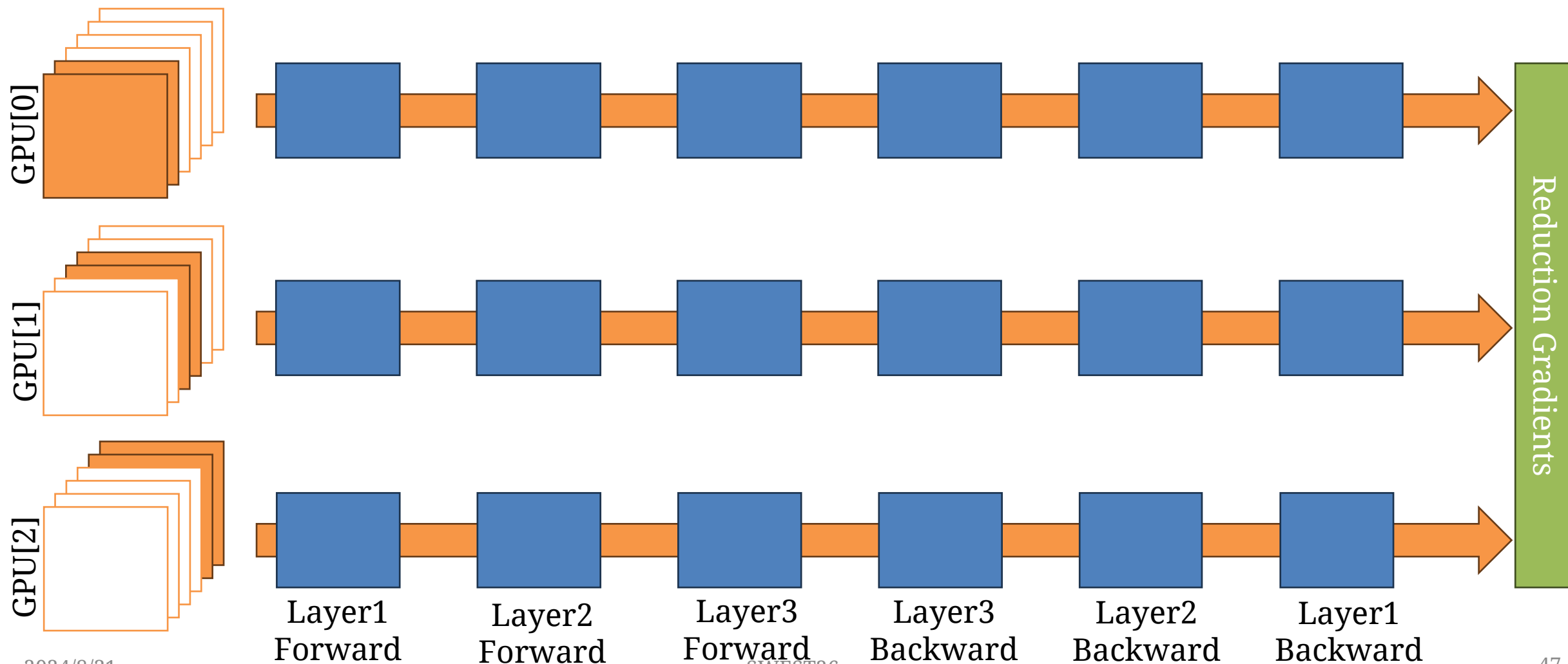
# Appendix

# NNの並列性

- 現在のNNの並列性は以下の3つに大別できる
  - • Data Parallel (訓練データを分割)
  - • Tensor Parallel (モデルを幅方向に分割)
  - • Pipeline Parallel (モデルを深さ方向に分割)
- BPは各層の計算が逐次
- 😞 順伝播時の情報がメモリを圧迫し続ける
- 😞 深さに応じてパイプライン並列性の利用は困難に

# NNの並列性(1): Data Parallel

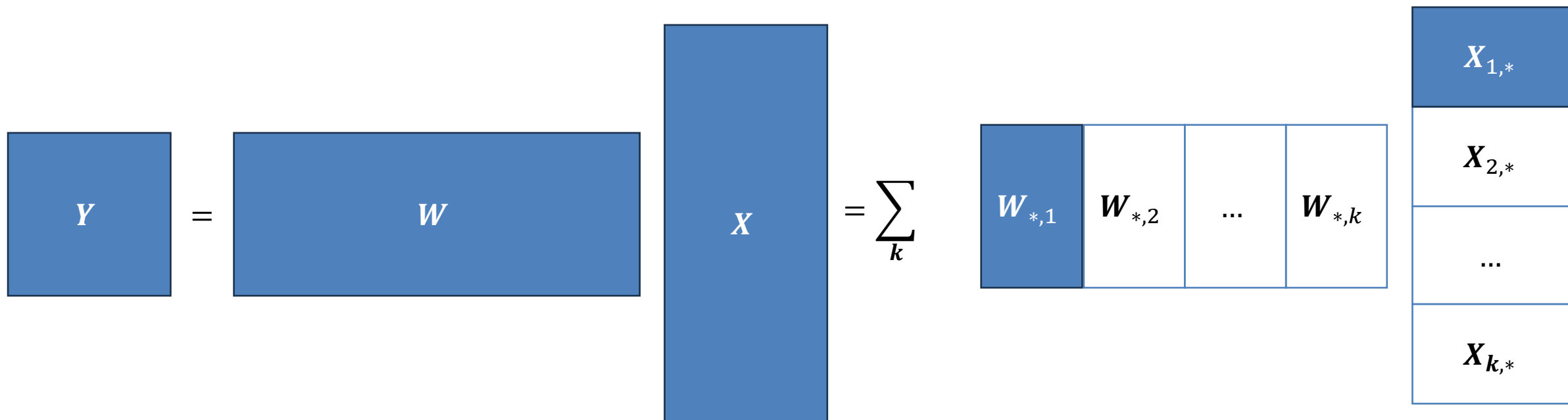
- 😊 学習データを分割して並列処理/ 😞 勾配を集約する通信コスト



# NNの並列性(2): Tensor Parallel

- 🥰 テンソルを分割して並列処理

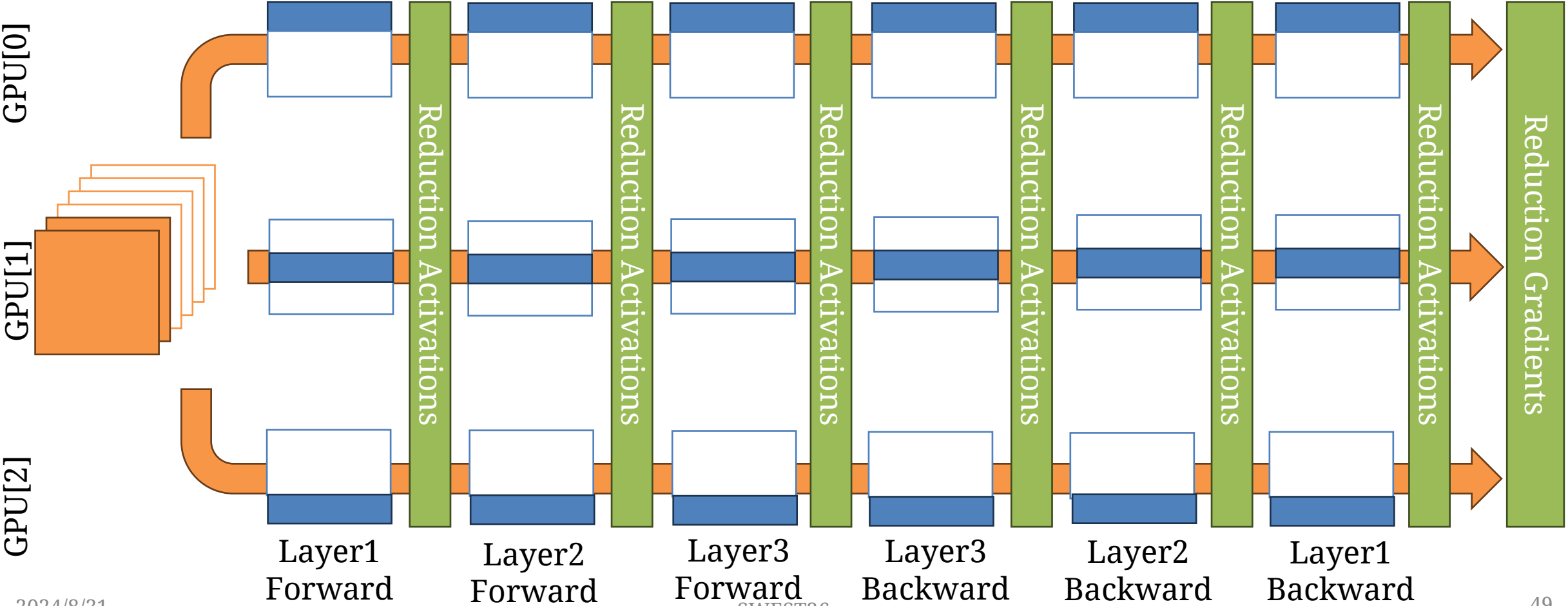
- $Y = WX = X[W_{*,1}, W_{*,2}, \dots, W_{*,k}] \begin{bmatrix} X_{1,*} \\ X_{2,*} \\ \dots \\ X_{k,*} \end{bmatrix} = \sum_k W_{*,k} X_{k,*}$





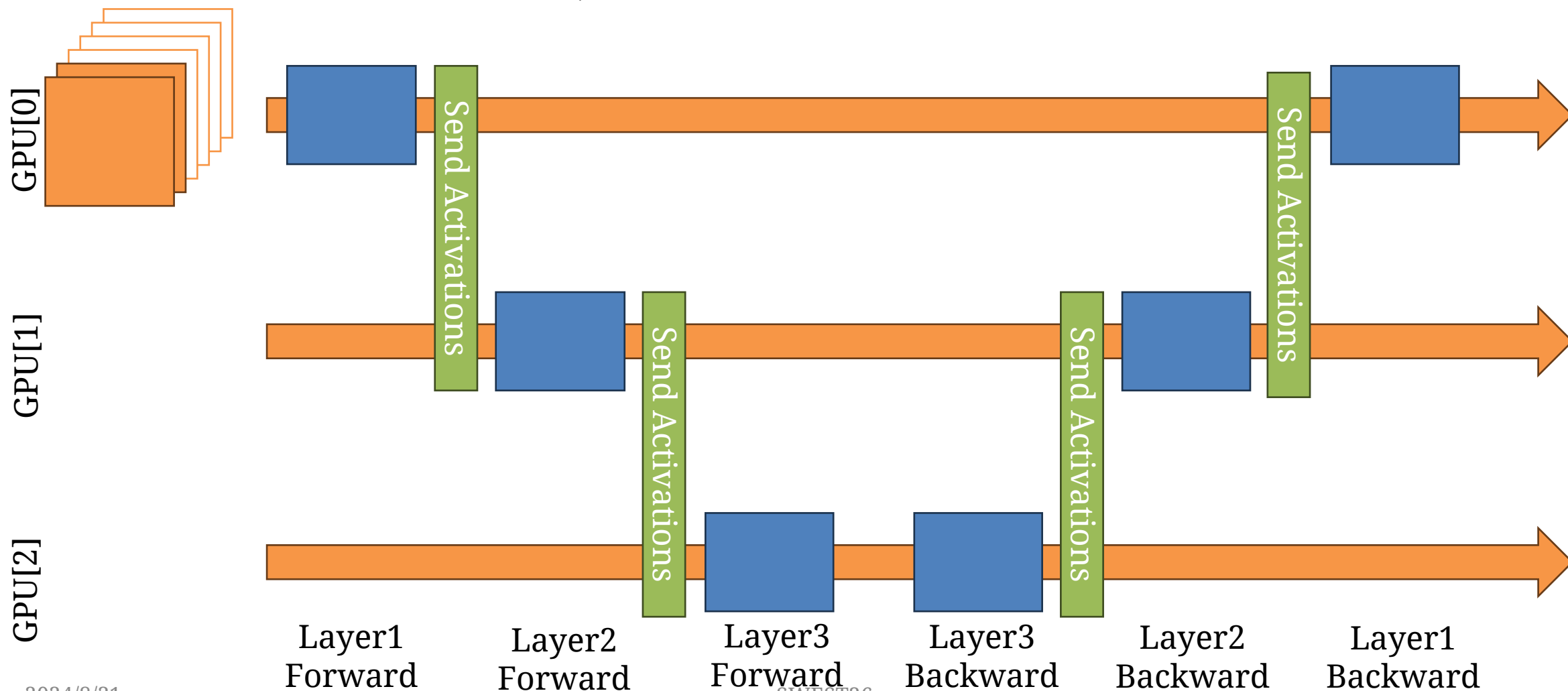
# NNの並列性(2): Tensor Parallel

- 😓 アクティベーションと勾配を集約する通信コスト



# NNの並列性(3): Pipeline Parallel

- 😊 パイプライン並列性, データ移動が最小/😭 バブルが不可避



# NNの並列性(3): Pipeline Parallel

- 十分なメモリ容量を仮定するとバブルなし
  - F: 順伝播の計算
  - B: 入力勾配計算
  - W: 重みの勾配計算
- いずれにしても膨大なメモリは必要

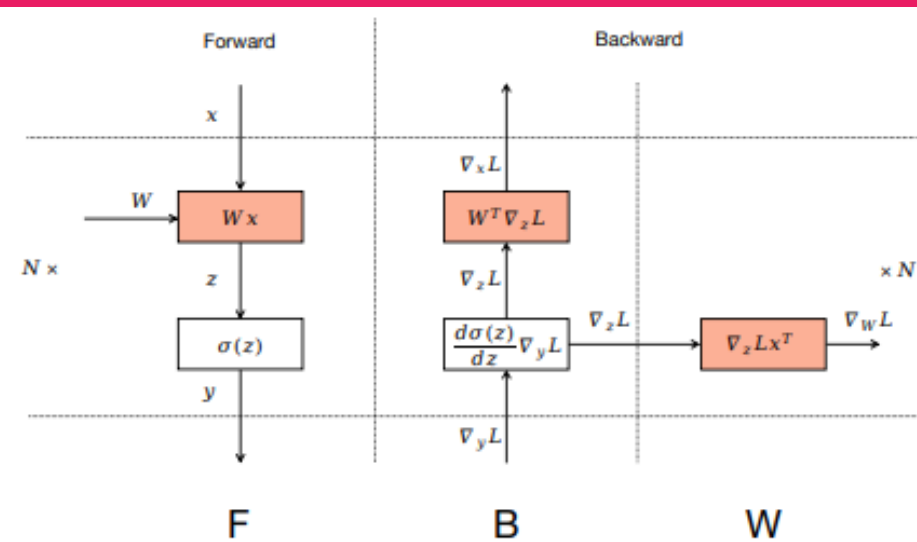


Figure 1: Computation Graph for MLP.

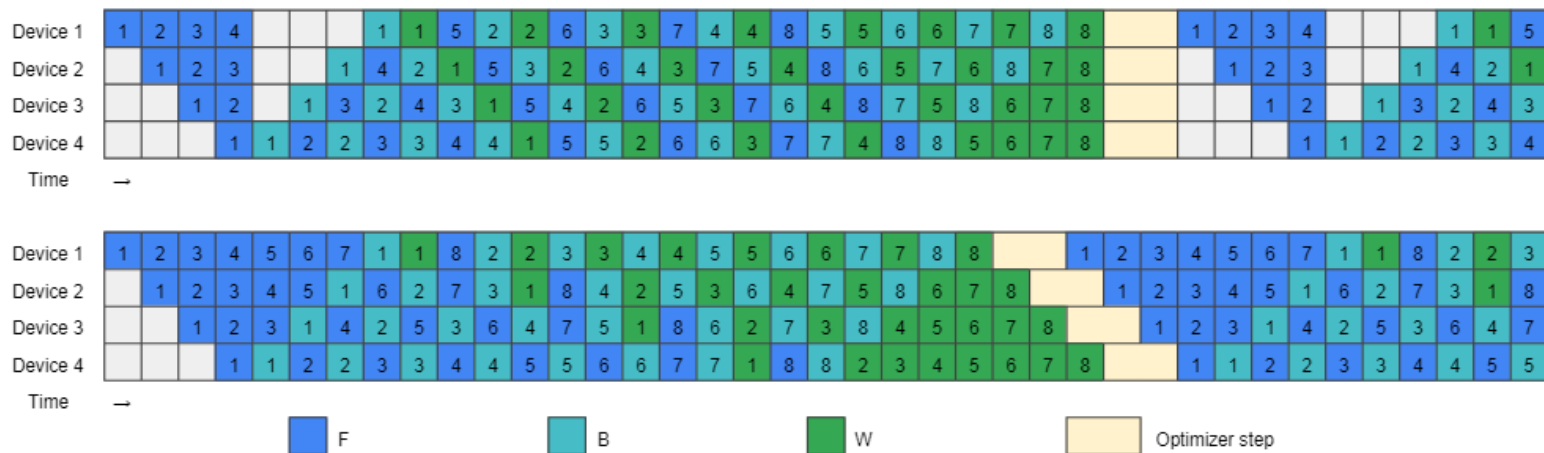


Figure 3: Handcrafted pipeline schedules, top: ZB-H1; bottom: ZB-H2

<https://arxiv.org/abs/2401.10241>

# NNの並列性まとめ

- 通信コストはTensor Parallel >> Data Parallel >> Pipeline Parallel
- Tensor Parallel (2~8)
  - 単一ノード内で
- Data Parallel
  - 可能な限り使う
- Pipeline Parallel
  - 頑張れば利用できなくはないが、基本的には一番非効率的
  - 上2つをどうしても使えない場合に (Necessary Evil)