JDLA E資格認定プログラム

全人類がわかるE資格コース

MLP基礎（バッチ処理編）

AVILEN

目次

1）バッチ処理（順伝播）

2）バッチ処理（逆伝播）

3）最適化手法

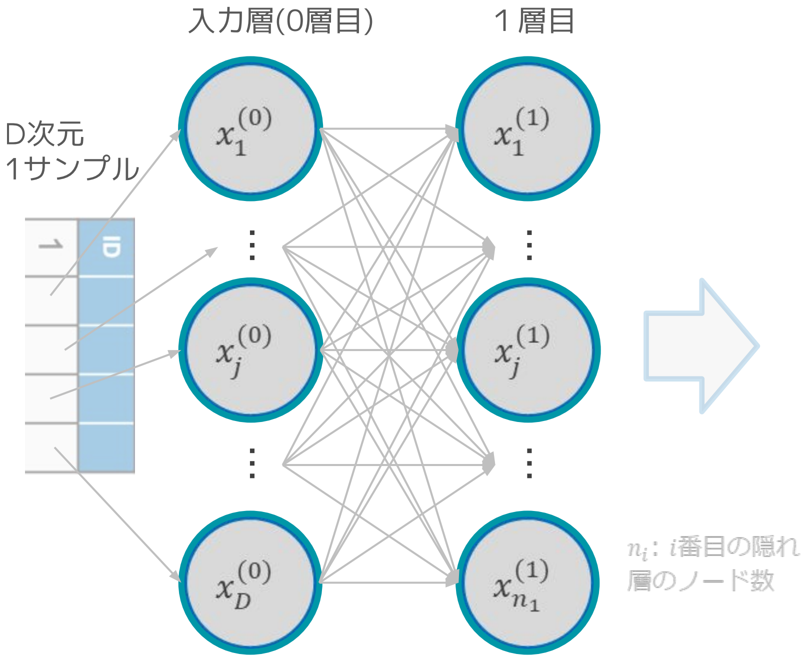
【Chapter03】MLP基礎（バッチ処理編）

バッチ処理（順伝播）

バッチ処理とは

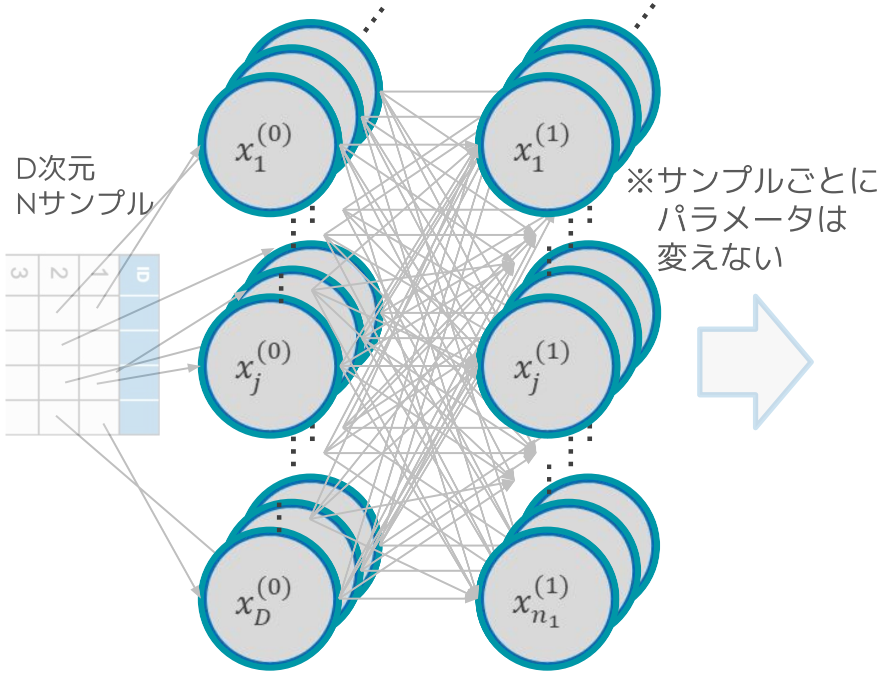
オンライン学習

1サンプルずつ学習する



バッチ処理（バッチ学習）

Nサンプルまとめて学習する

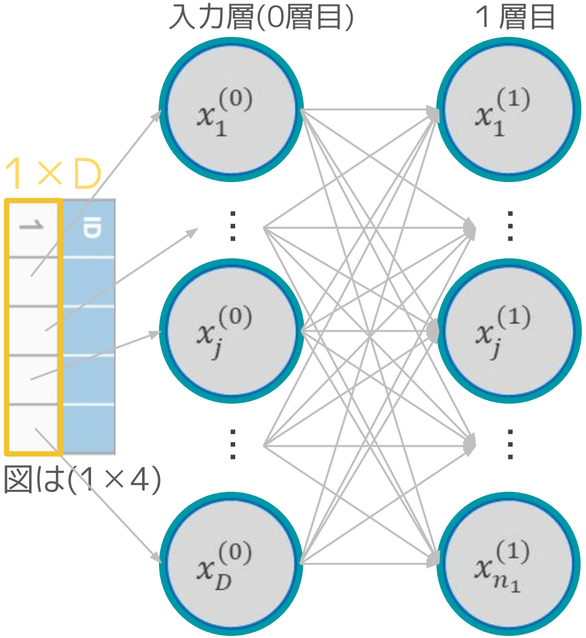


ここでの「学習する」とは、損失関数を計算しパラメータを最適化していく過程のことである

Nサンプルの入力を「行列」で表す

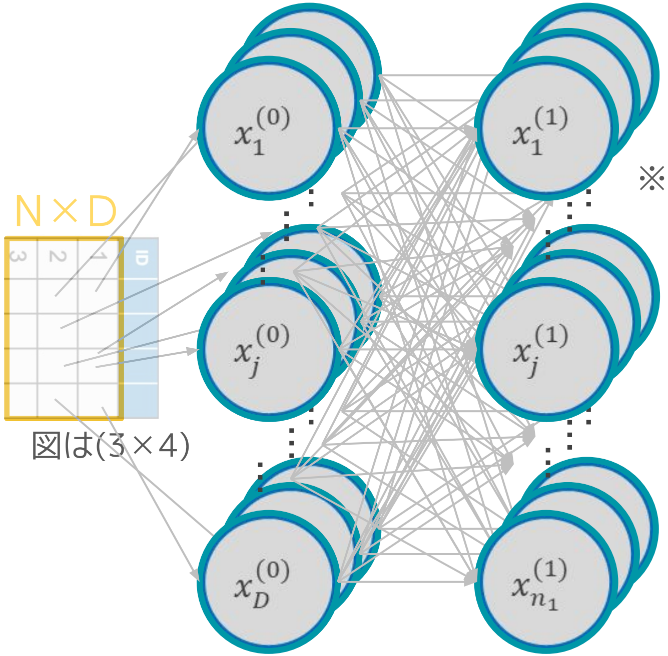
オンライン学習

1サンプルなので入力はベクトル



バッチ処理（バッチ学習）

Nサンプルであり入力は行列



※サンプルごとに
パラメータは
変えない

バッチサイズ

バッチサイズとは、パラメータを一度更新するための損失計算に必要なデータ数

【言い換え】ひとまとまりのグループのことをバッチサイズという

これまでの資料でのNはバッチサイズと呼ばれる。
すなわち・・・

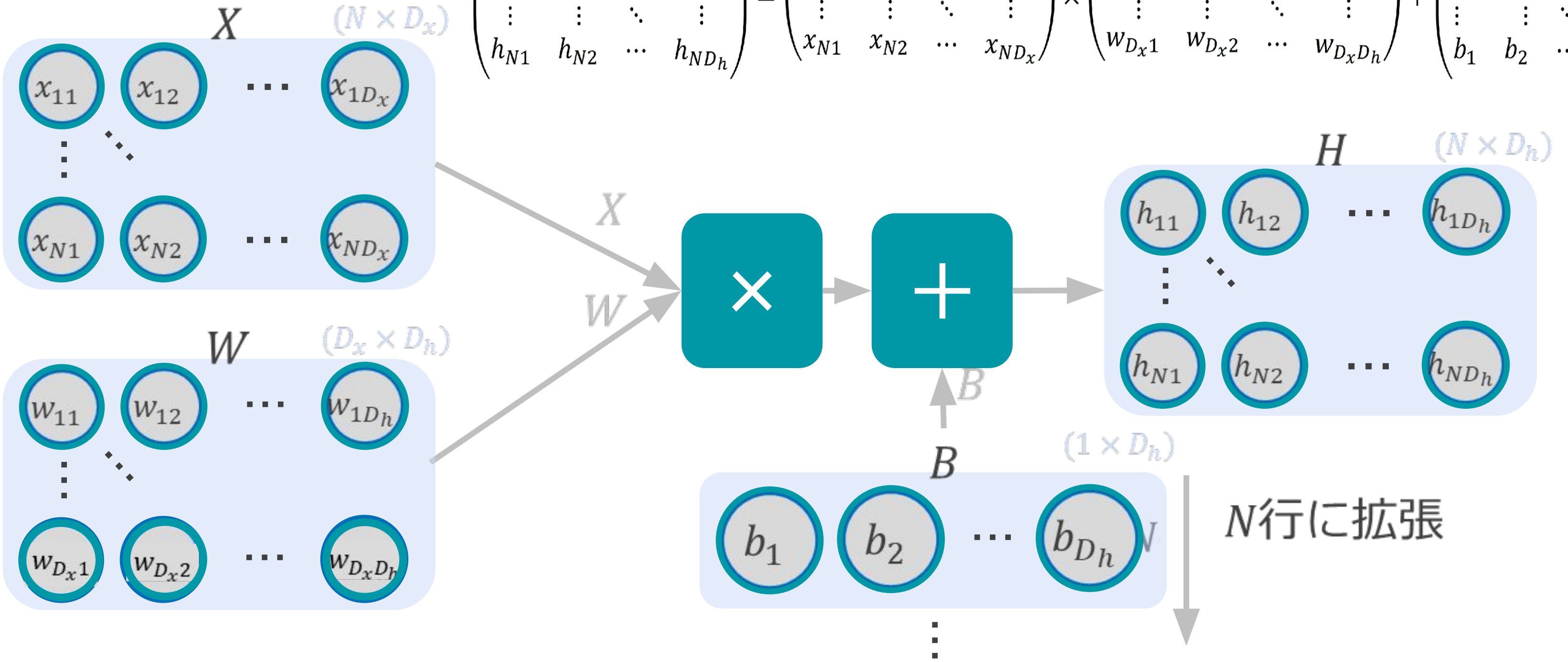
	オンライン学習	バッチ処理（バッチ学習）
バッチサイズN	1	全データ数
損失関数（二乗和誤差関数）	$\sum_{n}^{1}(t_n - y_n)^2$	$\sum_{n}^{N}(t_n - y_n)^2$

計算グラフ：バッチ版Affine変換

バッチ版Affine変換の順伝播を記す

$$H = X \times W + B$$

$$\begin{pmatrix} h_{11} & h_{12} & \cdots & h_{1D_h} \\ h_{21} & h_{22} & \cdots & h_{2D_h} \\ \vdots & \vdots & \ddots & \vdots \\ h_{N1} & h_{N2} & \cdots & h_{ND_h} \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1D_x} \\ x_{21} & x_{22} & \cdots & x_{2D_x} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{ND_x} \end{pmatrix} \times \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1D_h} \\ w_{21} & w_{22} & \cdots & w_{2D_h} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D_x1} & w_{D_x2} & \cdots & w_{D_xD_h} \end{pmatrix} + \begin{pmatrix} b_1 & b_2 & \cdots & b_{D_h} \\ b_1 & b_2 & \cdots & b_{D_h} \\ \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & \cdots & b_{D_h} \end{pmatrix}$$



バッチ版Affine変換の実装例

AVILEN

$$
\begin{pmatrix} h_{11} & h_{12} & \cdots & h_{1D_h} \\ h_{21} & h_{22} & \cdots & h_{2D_h} \\ \vdots & \vdots & \ddots & \vdots \\ h_{N1} & h_{N2} & \cdots & h_{ND_h} \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1D_x} \\ x_{21} & x_{22} & \cdots & x_{2D_x} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{ND_x} \end{pmatrix} \times \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1D_h} \\ w_{21} & w_{22} & \cdots & w_{2D_h} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D_x1} & w_{D_x2} & \cdots & w_{D_xD_h} \end{pmatrix} + \begin{pmatrix} b_1 & b_2 & \cdots & b_{D_h} \\ b_1 & b_2 & \cdots & b_{D_h} \\ \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & \cdots & b_{D_h} \end{pmatrix}
$$

計算フローはオンライン版(バッチサイズ=1の時)とほとんど変わらない

```python
N = 10 # バッチサイズ
Dx = 3 # 入力のカラムは3次元
Dh = 2 # outputのカラムは2次元
linear = nn.Linear(in_features=Dx, out_features=Dh)
input = torch.randn(N, Dx)
output = linear(input)
print(output)
```

```
tensor([[ 0.1459,  0.7930],
        [ 0.7778,  0.4139],
        [ 0.4035,  0.6504],
        [ 0.0781, -0.0228],
        [ 0.0316,  0.0502],
        [-0.0242,  0.2527],
        [-0.3394,  0.5411],
        [-0.1472,  0.3472],
        [-0.6998,  0.1616],
        [ 0.1409,  0.6075]], grad_fn=<AddmmBackward>)
```

inputの次元(N, D_x)=(10, 3)
outputの次元(N, D_h)=(10, 2)

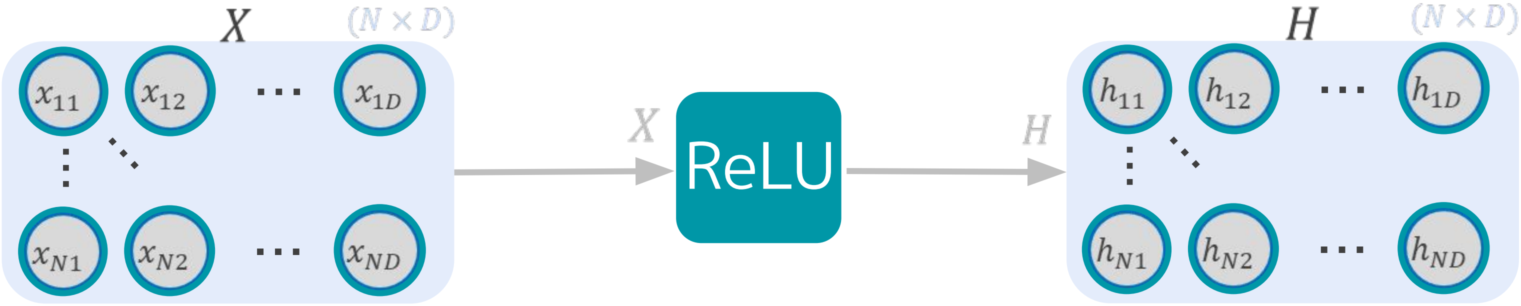
計算グラフ: バッチ版ReLU

ReLU関数のバッチ版順伝播を記す

$$h_{ij} = \mathrm{ReLU}(x_{ij}) = \begin{cases} x_{ij} \ (x_{ij} > 0) \\ 0 \ (x_{ij} \leq 0) \end{cases}$$

Point

1要素ずつReLUに通す



バッチ版ReLUの実装例

$$h_{ij} = \mathrm{ReLU}(x_{ij}) = \begin{cases} x_{ij} \ (x_{ij} > 0) \\ 0 \ (x_{ij} \leq 0) \end{cases}$$

```
import torch
import torch.nn as nn

N = 10 # バッチサイズ
D = 3 # カラムが3次元のデータ
relu = nn.ReLU()
input = torch.randn(N, D)
output = relu(input)
print(output)
```

```
tensor([[1.6895, 0.7285, 0.0000],
        [0.9630, 0.0000, 0.9106],
        [0.1444, 0.0000, 0.1002],
        [0.2353, 0.0000, 0.0000],
        [0.0000, 0.7483, 0.8810],
        [0.0000, 0.0000, 0.8269],
        [0.0000, 0.0000, 0.4797],
        [0.2976, 0.0607, 0.2066],
        [0.9198, 0.0000, 0.5978],
        [0.6547, 0.0000, 0.3788]])
```

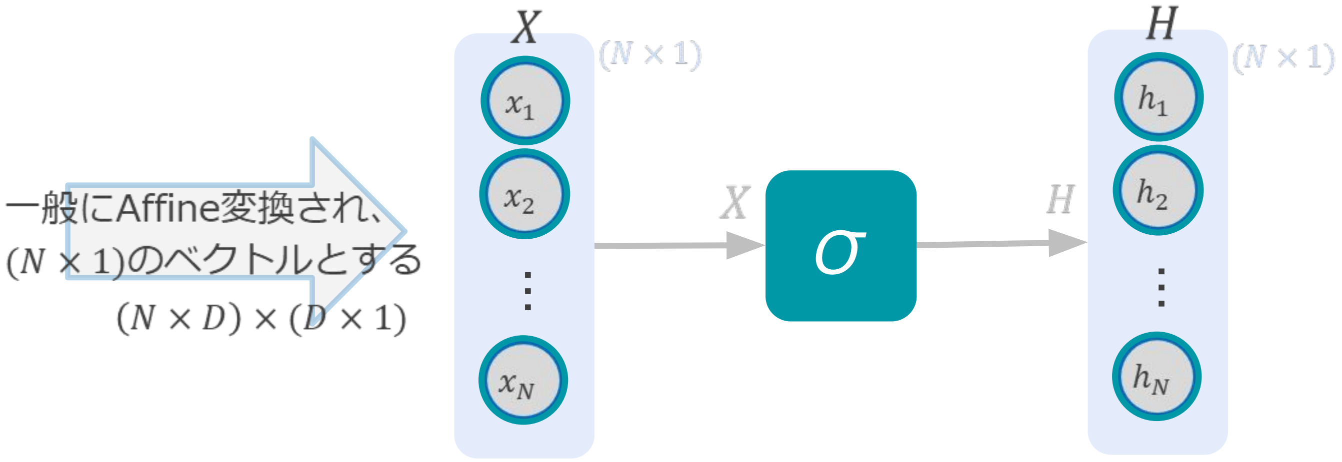
inputの次元(N, D_x)=(10, 3)
outputの次元(N, D_h)=(10, 3)

計算グラフ: バッチ版シグモイド

シグモイド関数のバッチ版順伝播を記す

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

Point
1要素ずつシグモイドに通す



バッチ版シグモイドの実装例

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

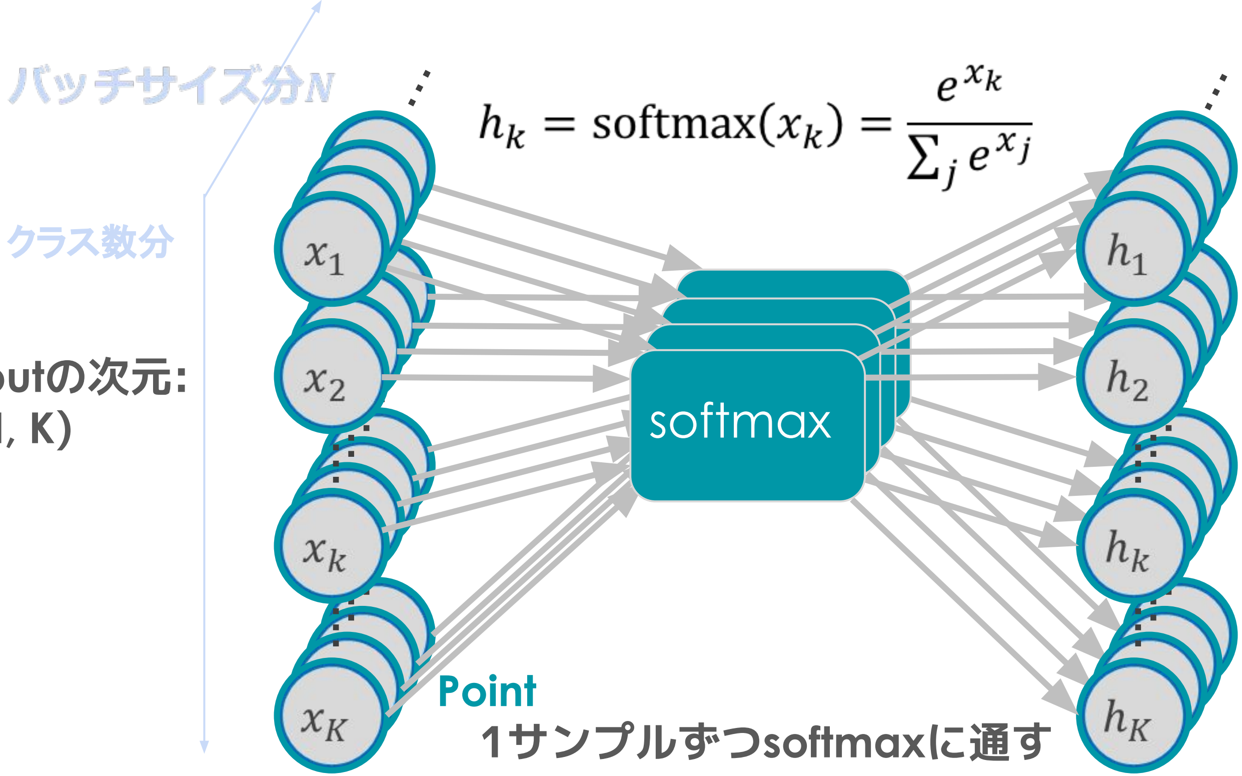
```
N = 10 # バッチサイズ
D = 1 # カラムが1次元のデータ
sigmoid = nn.Sigmoid()
input = torch.randn(N, D)
output = sigmoid(input)
print(output)
```

```
tensor([[0.2597],
        [0.0730],
        [0.2210],
        [0.1687],
        [0.5634],
        [0.6582],
        [0.8052],
        [0.1817],
        [0.4726],
        [0.6259]])
```

inputの次元(N, D_x)=(10, 1)

outputの次元(N, D_h)=(10, 1)

計算グラフ: バッチ版Softmax



$$h_k = \mathrm{softmax}(x_k) = \frac{e^{x_k}}{\sum_j e^{x_j}}$$

inputの次元:
(N, K)

outputの次元:
(N, K)

Point
1サンプルずつsoftmaxに通す

バッチ版Softmax実装例

$$h_k = \mathrm{softmax}(x_k) = \frac{e^{x_k}}{\sum_j e^{x_j}}$$

dim=1

dim=0

1	2	3
1	2	3

```
sm0 = nn.Softmax(dim=0)  # 列方向に足し算
sm1 = nn.Softmax(dim=1)  # 行方向に足し算
input = torch.tensor([[1, 2, 3], [1, 2, 3]], dtype=torch.float64)
output0 = sm0(input)
output1 = sm1(input)

print(input)
print(output0)
print(output1)
```

```
tensor([[1., 2., 3.],
        [1., 2., 3.]], dtype=torch.float64)
tensor([[0.5000, 0.5000, 0.5000],
        [0.5000, 0.5000, 0.5000]], dtype=torch.float64)
tensor([[0.0900, 0.2447, 0.6652],
        [0.0900, 0.2447, 0.6652]], dtype=torch.float64)
```

inputの次元:(2, 3)

outputの次元: (2, 3)

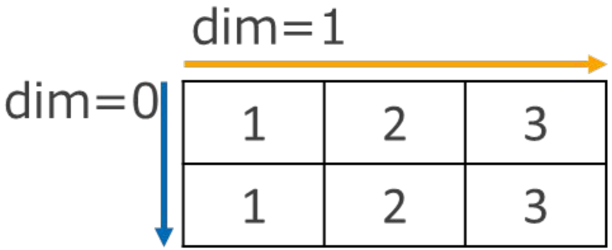
dim=0 OR dim=1 ?

通常、データは以下のような形のため、dim=1に設定する

データ

	説明変数			
ID				
1				
2				
3				
4				
5				
…				

softmax関数の設定



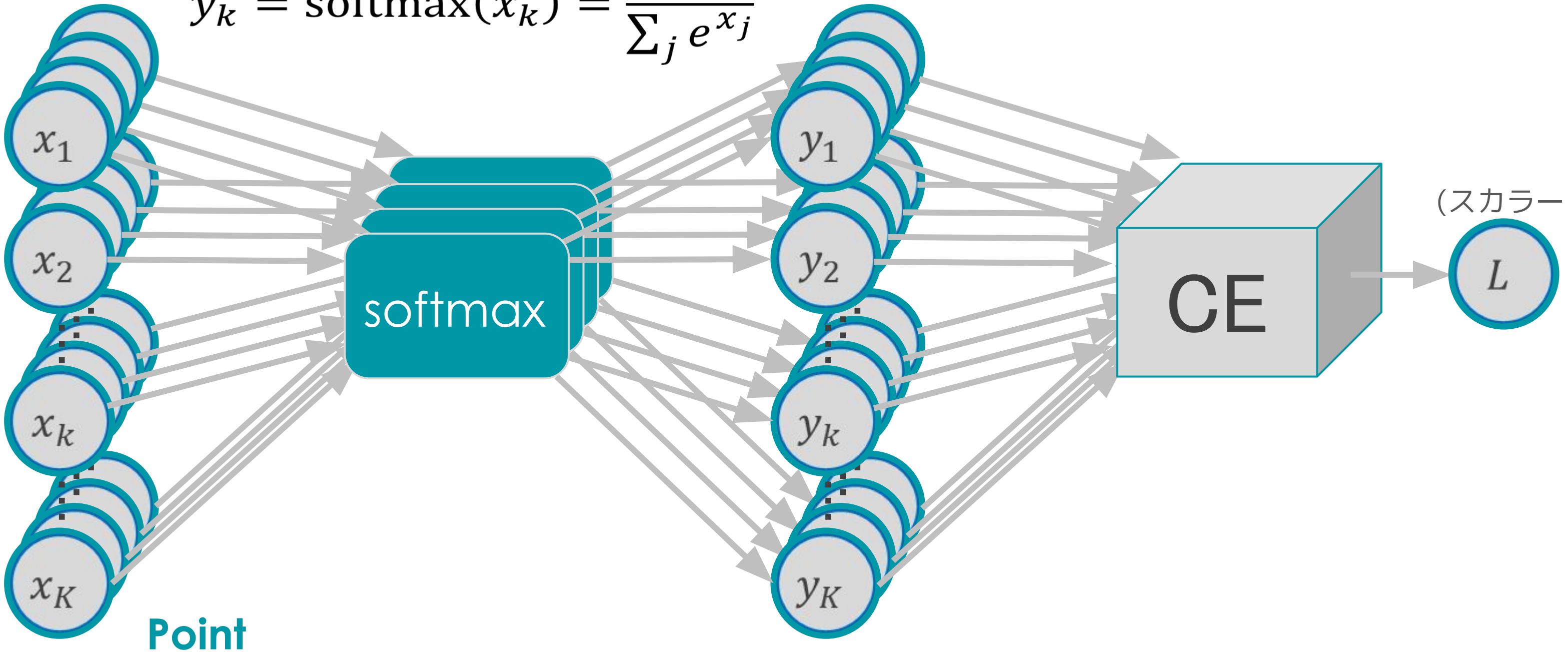
データサンプルの方向(dim=0)ではなく、
変数が並ぶ方向(dim=1)に設定する。

その結果、1サンプルずつsoftmaxに通すことができる

計算グラフ: バッチ版Softmax+クロスエントロピー(CE)

AVILEN

$$y_k = \mathrm{softmax}(x_k) = \frac{e^{x_k}}{\sum_j e^{x_j}}$$



Point

1サンプルずつSoftmax+CEに通し、
CEでバッチサイズ分の損失平均を算出する

【Chapter03】MLP基礎（バッチ処理編）

バッチ処理（逆伝播）

行列での微分記法

スカラー量Lの行列$\boldsymbol{W}(2 \times 3)$における偏微分行列$\frac{\partial L}{\partial \boldsymbol{W}}$を考える

$\boldsymbol{W} = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix}$としたとき　$\frac{\partial L}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \frac{\partial L}{\partial w_{12}} & \frac{\partial L}{\partial w_{13}} \\ \frac{\partial L}{\partial w_{21}} & \frac{\partial L}{\partial w_{22}} & \frac{\partial L}{\partial w_{23}} \end{pmatrix}$

(2×3)の同じ形式

行列での微分記法

一般に、スカラー量Lの行列$\boldsymbol{W}(D \times H)$における偏微分行列$\frac{\partial L}{\partial \boldsymbol{W}}$は

$$\boldsymbol{W} = \begin{pmatrix} w_{11} & \cdots & w_{1H} \\ \vdots & \ddots & \vdots \\ w_{D1} & \cdots & w_{DH} \end{pmatrix}$$

としたとき

$$\frac{\partial L}{\partial \boldsymbol{W}} = \begin{pmatrix} \frac{\partial L}{\partial w_{11}} & \cdots & \frac{\partial L}{\partial w_{1H}} \\ \vdots & \ddots & \vdots \\ \frac{\partial L}{\partial w_{D1}} & \cdots & \frac{\partial L}{\partial w_{DH}} \end{pmatrix}$$

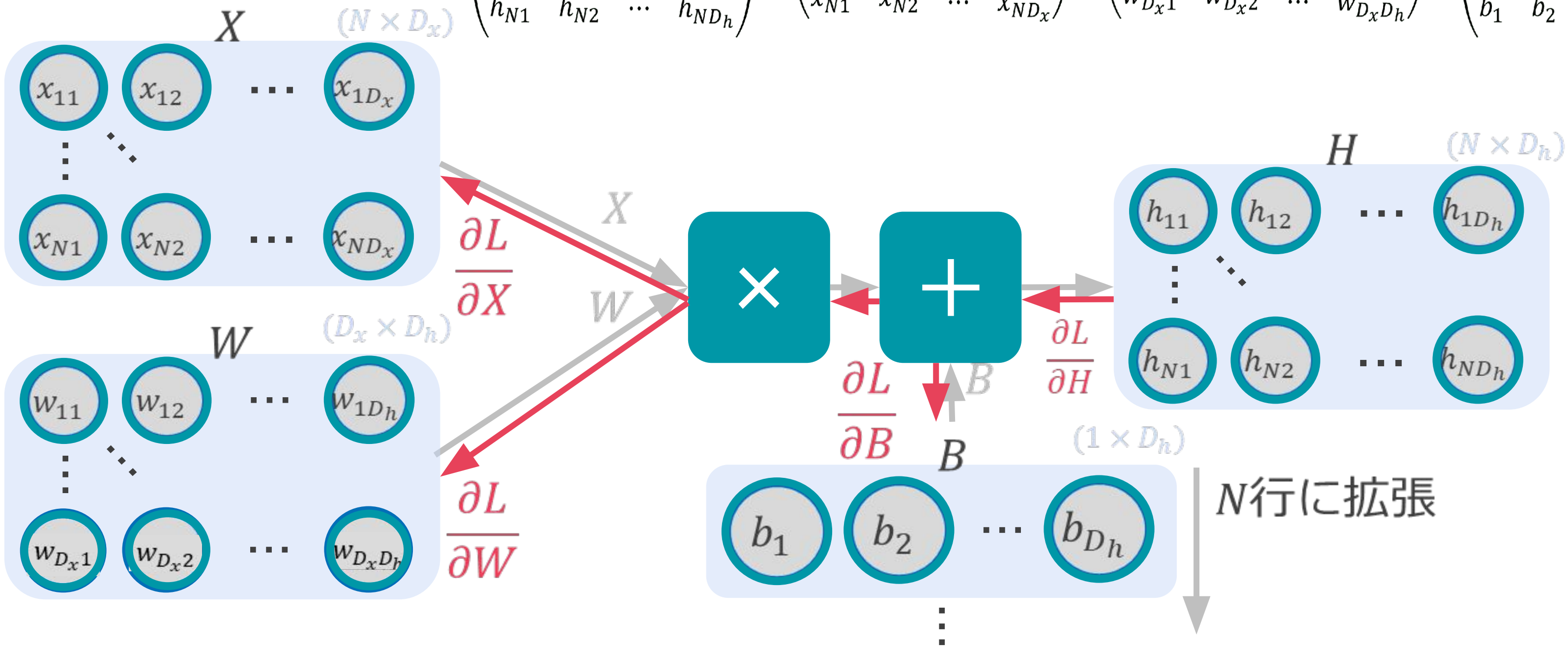
$(D \times H)$の同じ形式

計算グラフ: バッチ版Affine変換

バッチ版Affine変換の逆伝播を記す

$$H = X \times W + B$$

$$\begin{pmatrix} h_{11} & h_{12} & \cdots & h_{1D_h} \\ h_{21} & h_{22} & \cdots & h_{2D_h} \\ \vdots & \vdots & \ddots & \vdots \\ h_{N1} & h_{N2} & \cdots & h_{ND_h} \end{pmatrix} = \begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1D_x} \\ x_{21} & x_{22} & \cdots & x_{2D_x} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \cdots & x_{ND_x} \end{pmatrix} \times \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1D_h} \\ w_{21} & w_{22} & \cdots & w_{2D_h} \\ \vdots & \vdots & \ddots & \vdots \\ w_{D_x1} & w_{D_x2} & \cdots & w_{D_xD_h} \end{pmatrix} + \begin{pmatrix} b_1 & b_2 & \cdots & b_{D_h} \\ b_1 & b_2 & \cdots & b_{D_h} \\ \vdots & \vdots & \ddots & \vdots \\ b_1 & b_2 & \cdots & b_{D_h} \end{pmatrix}$$

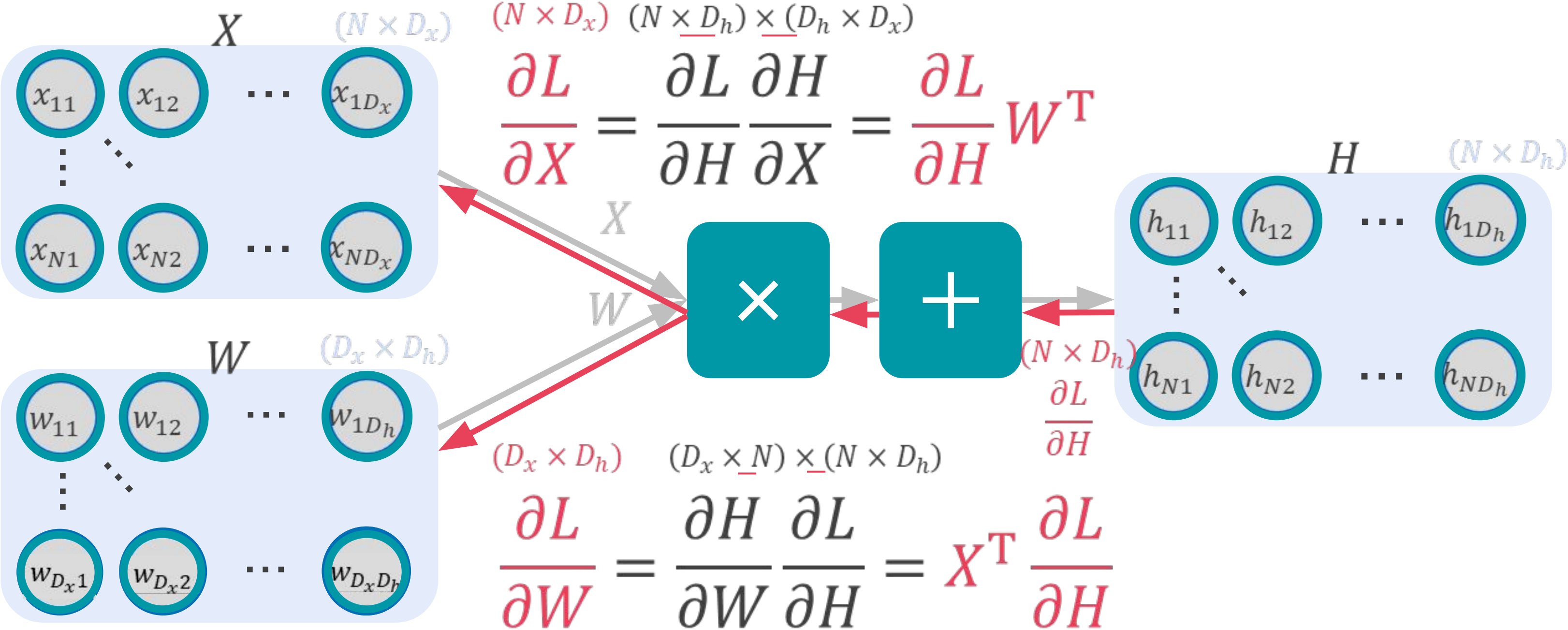


計算グラフ: バッチ版Affine変換

連鎖律を用いて、バッチ版Affine変換の逆伝播の詳細を記す

$$H = X \times W + B$$

$$\underset{(N \times D_x)}{\frac{\partial L}{\partial X}} = \underset{(N \times D_h) \times (D_h \times D_x)}{\frac{\partial L}{\partial H}\frac{\partial H}{\partial X}} = \frac{\partial L}{\partial H} W^{\mathrm{T}}$$



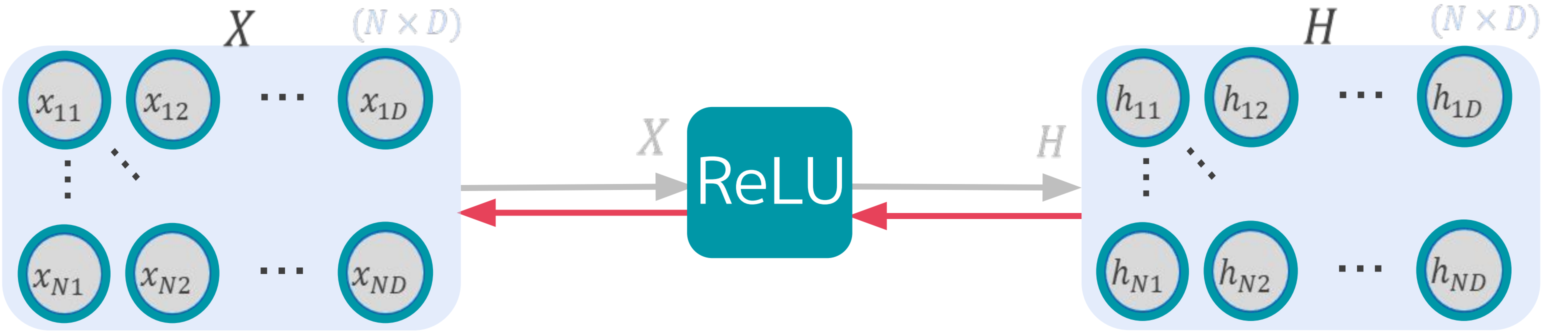
$$\underset{(D_x \times D_h)}{\frac{\partial L}{\partial W}} = \underset{(D_x \times N) \times (N \times D_h)}{\frac{\partial H}{\partial W}\frac{\partial L}{\partial H}} = X^{\mathrm{T}} \frac{\partial L}{\partial H}$$

計算グラフ: バッチ版ReLU

ReLU関数のバッチ版逆伝播を記す

$$h_{ij} = \mathrm{ReLU}(x_{ij}) = \begin{cases} x_{ij} \ (x_{ij} > 0) \\ 0 \ (x_{ij} \leq 0) \end{cases}$$

Point
順伝播では1要素ずつReLUに通したので、逆伝播でも「要素ごとの偏微分」を伝播させる



計算グラフ: バッチ版ReLU

バッチ版ReLU関数の微分

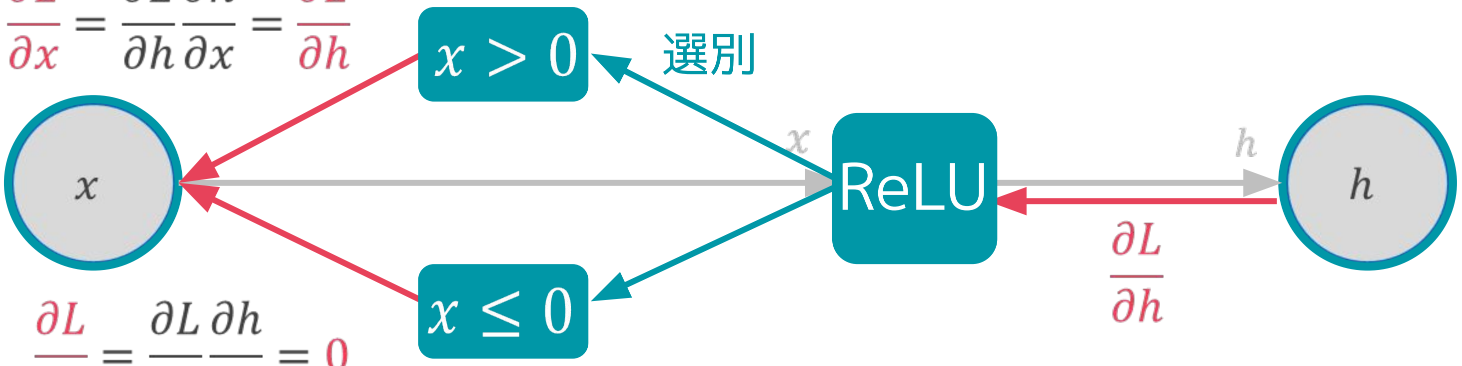
$$h_{ij} = \mathrm{ReLU}(x_{ij}) = \begin{cases} x_{ij} \ (x_{ij} > 0) \\ 0 \ (x_{ij} \leq 0) \end{cases}$$

要素ごとの偏微分 $\frac{\partial h_{ij}}{\partial x_{ij}}$ を求める

これはオンライン版と同じである

オンライン版の再掲

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial x} = \frac{\partial L}{\partial h}$$



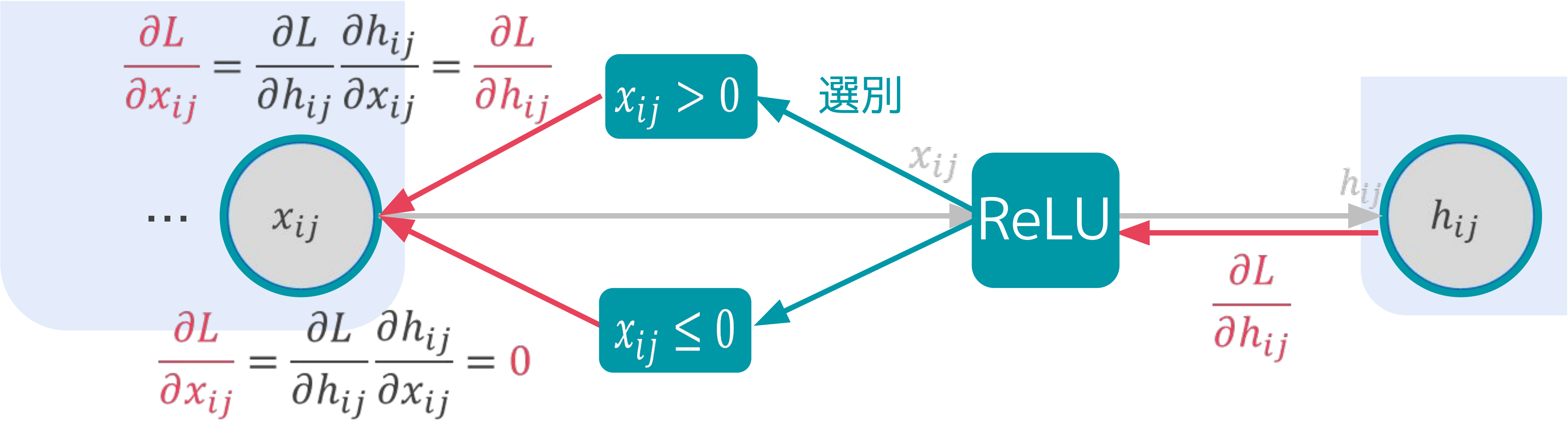
$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial x} = 0$$

計算グラフ: バッチ版ReLU

ReLU関数のバッチ版逆伝播の詳細を記す

$$h_{ij} = \mathrm{ReLU}(x_{ij}) = \begin{cases} x_{ij}\ (x_{ij} > 0) \\ 0\ (x_{ij} \leq 0) \end{cases}$$

Point
要素ごとの偏微分を逆伝播させる

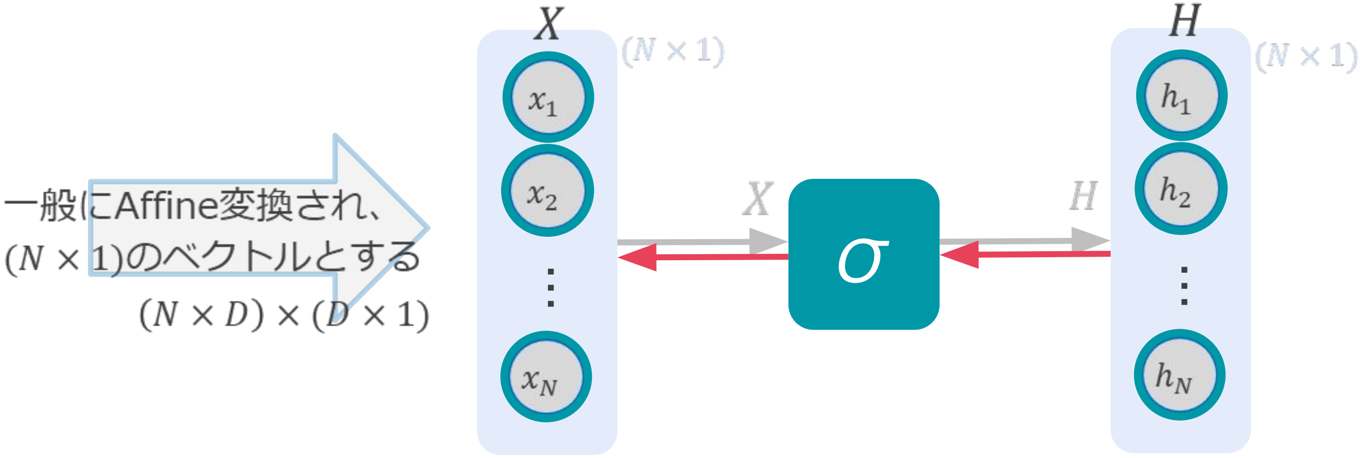


計算グラフ: バッチ版シグモイド

シグモイド関数のバッチ版逆伝播を記す

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

Point
ReLUと同様に、順伝播では1要素ずつσに通したので、逆伝播でも「要素ごとの偏微分」を伝播させる



計算グラフ: バッチ版シグモイド

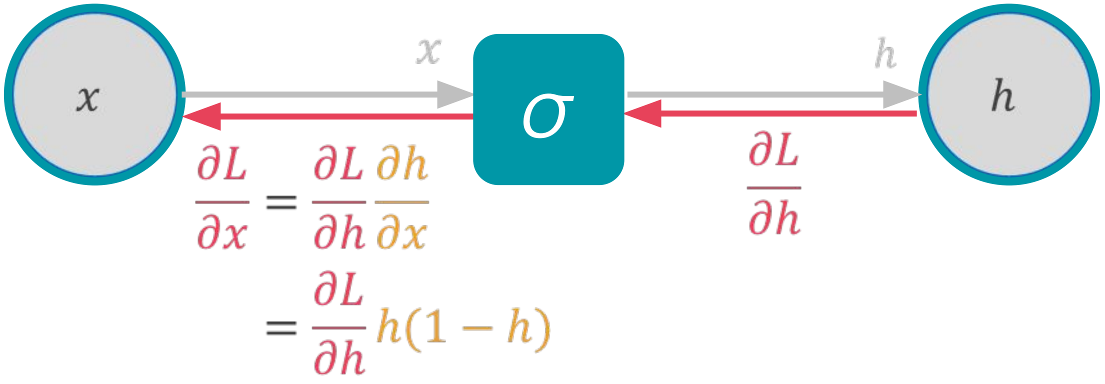
バッチ版シグモイド関数の微分

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

要素ごとの偏微分 $\frac{\partial h_{ij}}{\partial x_{ij}}$ を求める

これはオンライン版と同じである

オンライン版の再掲

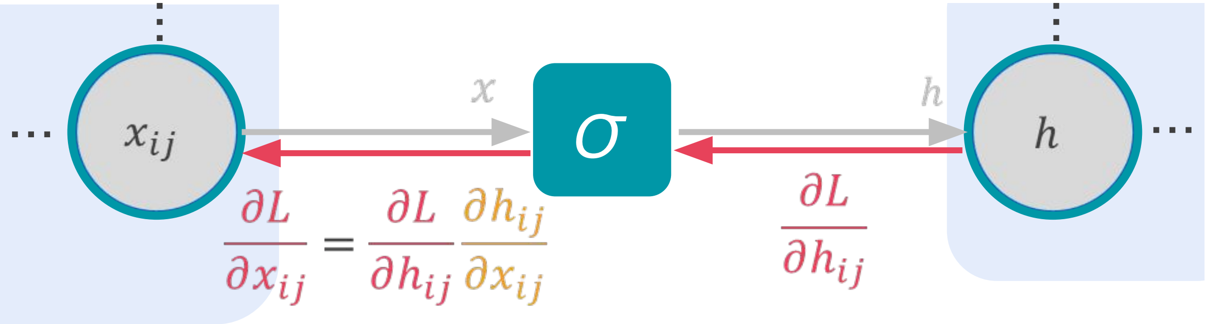


計算グラフ: バッチ版シグモイド

シグモイド関数のバッチ版逆伝播の詳細を記す

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

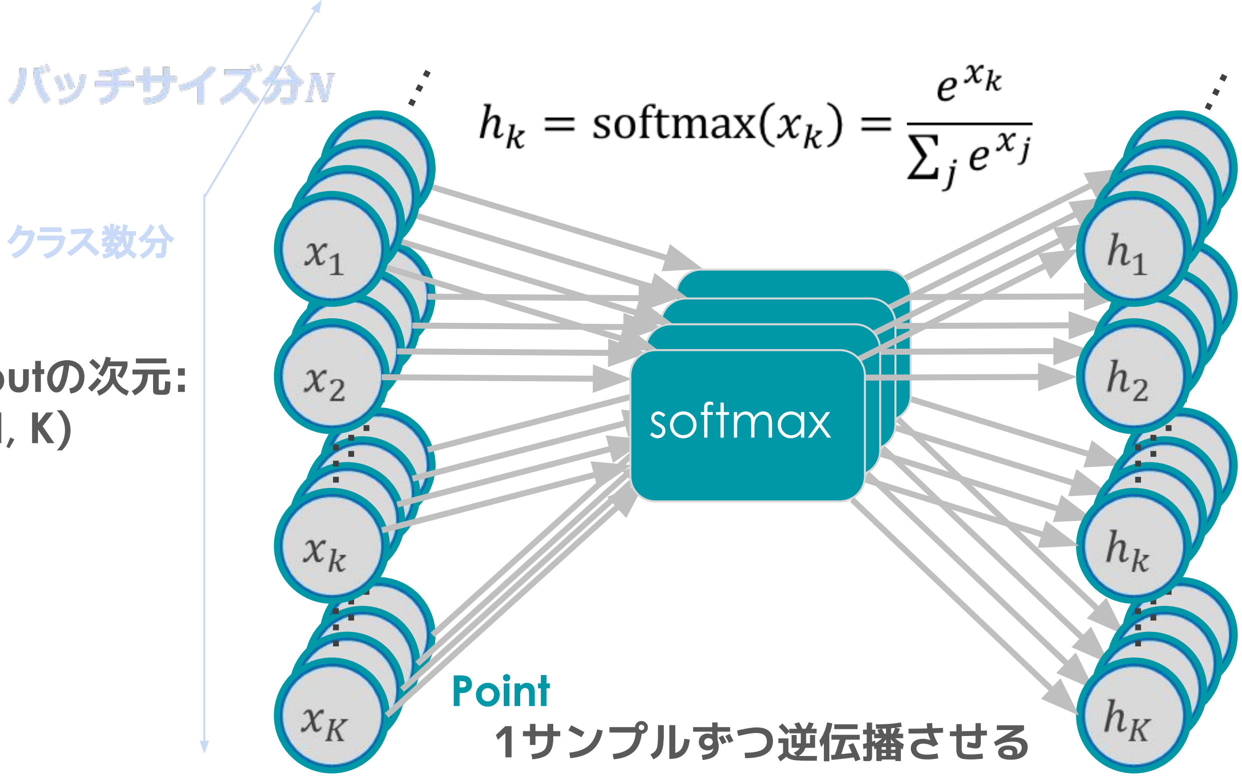
Point
要素ごとの偏微分を逆伝播させる



$$\frac{\partial L}{\partial x_{ij}} = \frac{\partial L}{\partial h_{ij}} \frac{\partial h_{ij}}{\partial x_{ij}}$$

$$= \frac{\partial L}{\partial h_{ij}} h_{ij}(1 - h_{ij})$$

計算グラフ: バッチ版Softmax



inputの次元:
(N, K)

outputの次元:
(N, K)

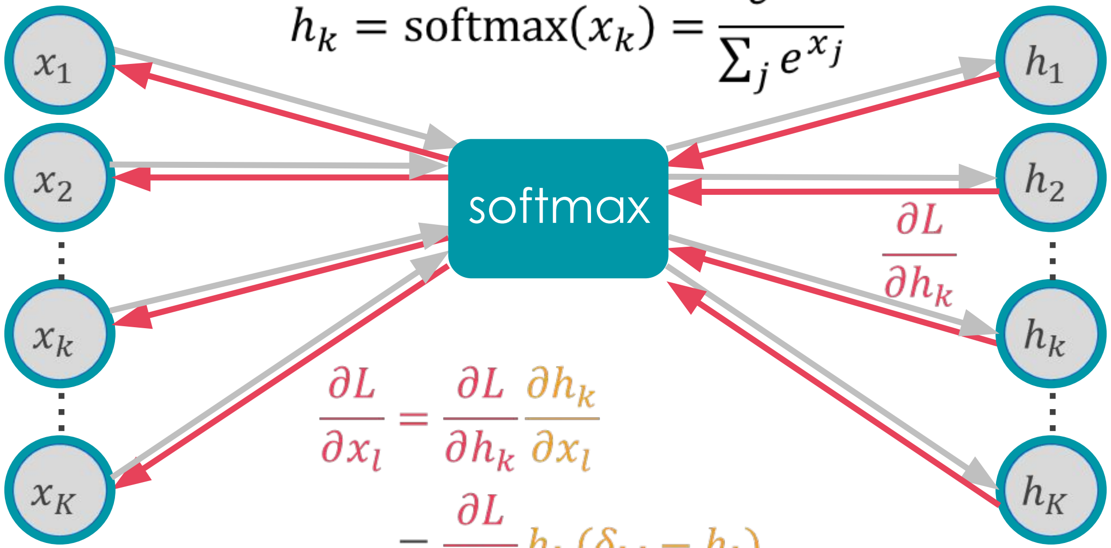
Point
1サンプルずつ逆伝播させる

計算グラフ: バッチ版Softmax

バッチ版softmax関数の微分

1サンプルごとに独立の計算なのでオンライン版×Nと考える

オンライン版の再掲



$$h_k = \mathrm{softmax}(x_k) = \frac{e^{x_k}}{\sum_j e^{x_j}}$$

$$\frac{\partial L}{\partial x_l} = \frac{\partial L}{\partial h_k}\frac{\partial h_k}{\partial x_l}$$

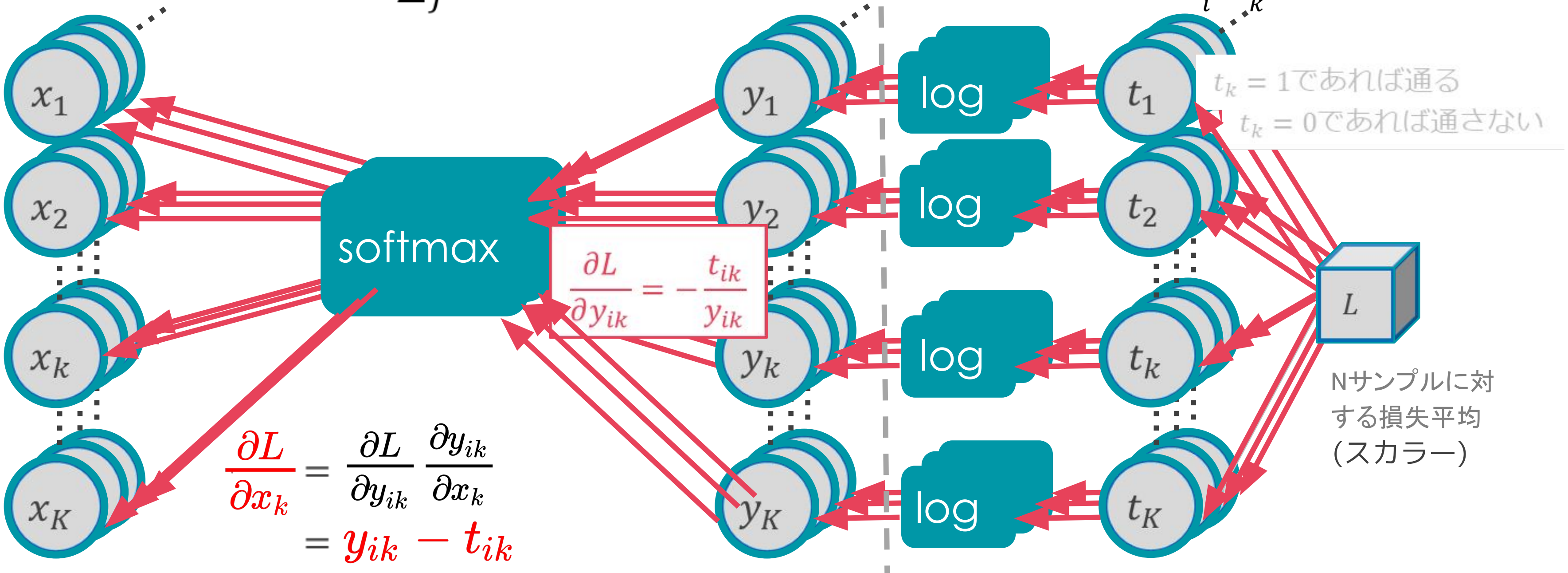
$$= \frac{\partial L}{\partial h_k} h_k(\delta_{kl} - h_l)$$

計算グラフ: バッチ版Softmax+クロスエントロピー

バッチ版Softmax With CrossEntropyの逆伝播の結果のみ記す

$$h_k = \mathrm{softmax}(x_k) = \frac{e^{x_k}}{\sum_j e^{x_j}}$$

$$L = CE(\boldsymbol{t}, \boldsymbol{y}) = -\frac{1}{N}\sum_i \sum_k t_{ik} \log y_{ik}$$



$$\frac{\partial L}{\partial x_k} = \frac{\partial L}{\partial y_{ik}} \frac{\partial y_{ik}}{\partial x_k} = y_{ik} - t_{ik}$$

分類問題に用いられる手法である。Kはクラス数を表す

タスクと出力ユニットと損失関数まとめ

AVILEN

タスクによって出力ユニットが異なり、適切な損失関数も異なる

タスク		出力ユニット	活性化関数	損失関数	損失関数の式
回帰	線形回帰	線形ユニット	恒等関数	二乗和誤差関数	$\sum_i (t_i - y_i)^2$
分類	二値分類	シグモイドユニット	sigmoid関数	バイナリークロスエントロピー関数	$-\sum_i (t_i \log y_i + (1 - t_i)\log(1 - y_i))$
分類	多値分類	ソフトマックスユニット	softmax関数	クロスエントロピー誤差関数	$-\sum_i \sum_k t_{ik} \log y_{ik}$

t_i: 目的変数
y_i: NN出力値
k: 所属クラスを表す添え字 (one-hot encoding後)
i: データサンプルの添え字 (何番目のサンプルかを表す)

バッチ処理ver.

【補足】線形ユニットに対する損失関数

バッチ処理でも、ガウス分布の対数尤度の最大化と二乗和誤差の最小化は等しい

前提

線形ユニットはガウス分布$\mathcal{N}(t;y,I)$を出力するためのユニットである

$$
\begin{aligned}
L(\theta) &= -\log N(t;y,I) \\
&= -\log e^{-\frac{\sum_i (t_i - y_i)^2}{2\sigma^2}} \\
&= \underbrace{\sum_i (t_i - y_i)^2}_{\text{二乗和誤差関数}} + \text{const.}
\end{aligned}
$$

t: 目的変数

y: NN出力値

$y_i = b + W^{\mathrm{T}} h$　(i番目のデータの予測値)

損失関数$L(\theta)$が最小となるようなWを求めることを「損失関数の最適化」という

【補足】シグモイドユニットに対する損失関数

前提

シグモイドユニットはベルヌーイ分布Bern(t;y)を出力するためのユニットである

$$L(\theta) = -\log \mathrm{Bern}(t; y)$$

$$= -\underbrace{\sum_i \left(t_i \log y_i + (1-t_i)\log(1-y_i)\right)}_{\text{バイナリークロスエントロピー誤差関数}}$$

t: 目的変数

y: NN出力値

$= \sigma(b + W^{\mathrm{T}} h)$

(i番目のデータの予測値)

【補足】one-hot encodingされている場合、
正解の予測確率の対数をとって和を求めるだけ

【補足】ソフトマックスユニットに対する損失関数

バッチ処理でも、
マルチヌーイ分布の対数尤度の最大化とクロスエントロピー誤差関数の最小化は等しい

前提

ソフトマックスユニットはマルチヌーイ分布Mul$n(t;y)$を出力するためのユニットである

$$L(\theta) = -\log \mathrm{Muln}(t; y)$$

$$= \underbrace{-\sum_i \sum_k t_{ik} \log y_{ik}}_{\text{クロスエントロピー誤差関数}}$$

【補足】one-hot encodingされている場合、
正解の予測確率の対数をとって和を求めるだけ

t: 目的変数

t_{ik}: i番目のデータのクラスk
のone-hot label

y: NN出力値

$y_{ik} = \mathrm{softmax}^{(k)}(b + W^{\mathrm{T}} h)$
(i番目のデータのクラスk
に対する予測値)

【Chapter03】MLP基礎（バッチ処理編）

最適化手法

ミニバッチ処理

バッチ処理では、すべての訓練データを一度にNNに流した

しかし【問題点】がある
データ数が膨大であれば計算量が膨大となりパラメータ更新が進みづらい

そこで**「ミニバッチ処理」**
訓練データをいくつかのグループに分けて
NNに投入する

ミニバッチ処理の【利点】
分割すると細かい更新が多く収束が速い

イメージ

バッチサイズ

バッチサイズとは、パラメータを一度更新するための損失計算に必要なデータ数

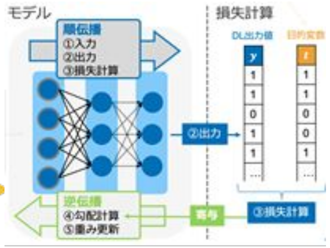
例)バッチサイズ＝100のとき

ID	col1	col2	col3	col4
1				
2				
…				
100				
101				
…				

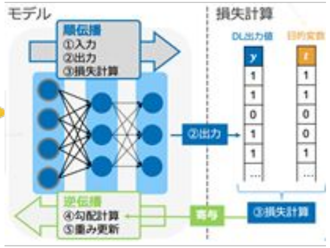
1回目の学習

2回目の学習

⋮



1.入力・出力
2.損失計算
3.勾配更新
4.重み更新



1.入力・出力
2.損失計算
3.勾配更新
4.重み更新

⋮

つまり・・・

オンライン学習

バッチサイズ＝1

バッチ学習

バッチサイズ
＝**全データ数**

バッチサイズ

バッチサイズとは、パラメータを一度更新するための損失計算に必要なデータ数

	オンライン学習	ミニバッチ処理	バッチ処理（バッチ学習）
バッチサイズN	1	中間の値 (例: 64/128/256)	全データ数
損失関数 （二乗和誤差関数）	$\sum_{n}^{1}(t_n - y_n)^2$	$\sum_{n}^{128}(t_n - y_n)^2$	$\sum_{n}^{N}(t_n - y_n)^2$

バッチサイズのトレードオフと有効なバッチサイズ

バッチサイズのトレードオフ

一般に、バッチサイズが大きければ勾配推定はより正確になるが、計算コストは高まる
一方で、バッチサイズが小さければ勾配推定の誤差を高める代わりに、更新頻度が早く収束が早くなると言われている

trade-offまとめ	勾配推定	収束まで
バッチサイズ大	より正確	遅い
バッチサイズ小	誤差を高める	早い

よく用いられる有効なバッチサイズ

計算リソースが限られている場合	16／32／64／128／256
計算リソースに余裕がある場合	512／1024　またはそれ以上

最適化とは、勾配の情報をヒントにパラメータをアップデートすること

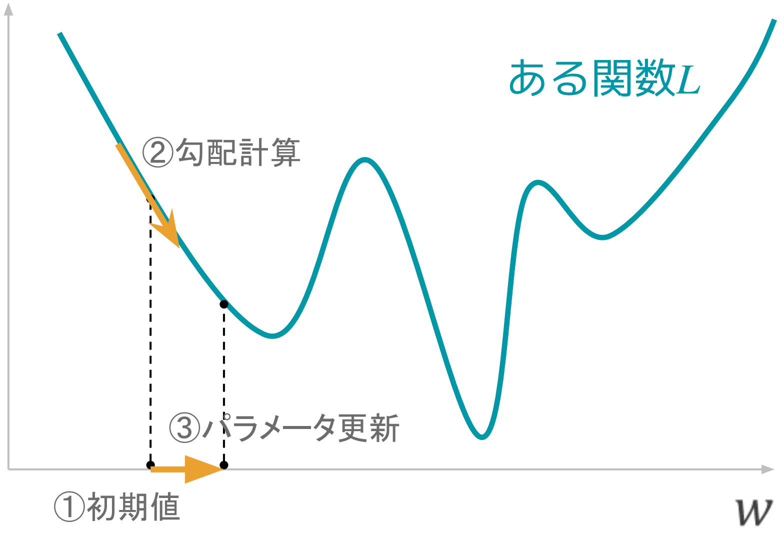
最適化の基本的流れ

①パラメータ w の初期値を決める

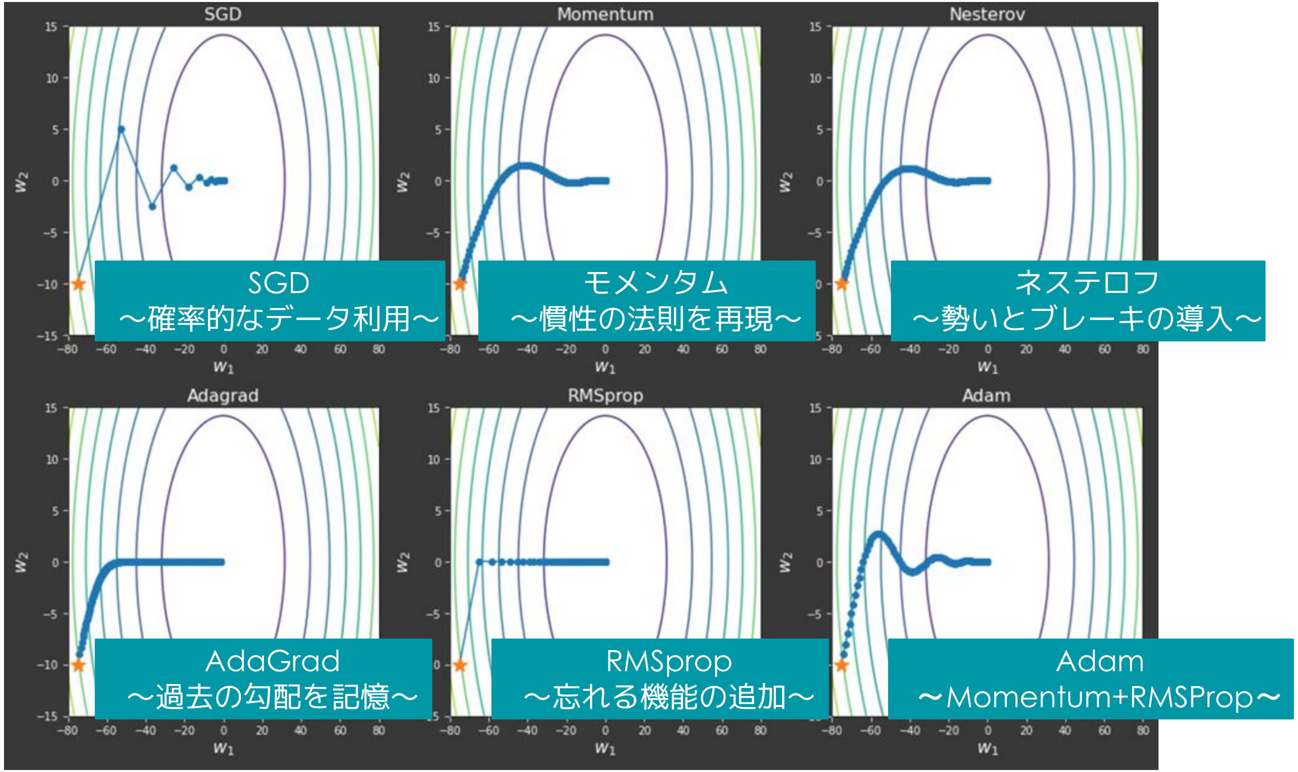
②その初期値における勾配 $\frac{\partial L}{\partial w}$ を計算

③勾配の大きさやその方向を頼りにパラメータ w を更新する

③の方法が各最適化手法によって異なる



最適化手法まとめ

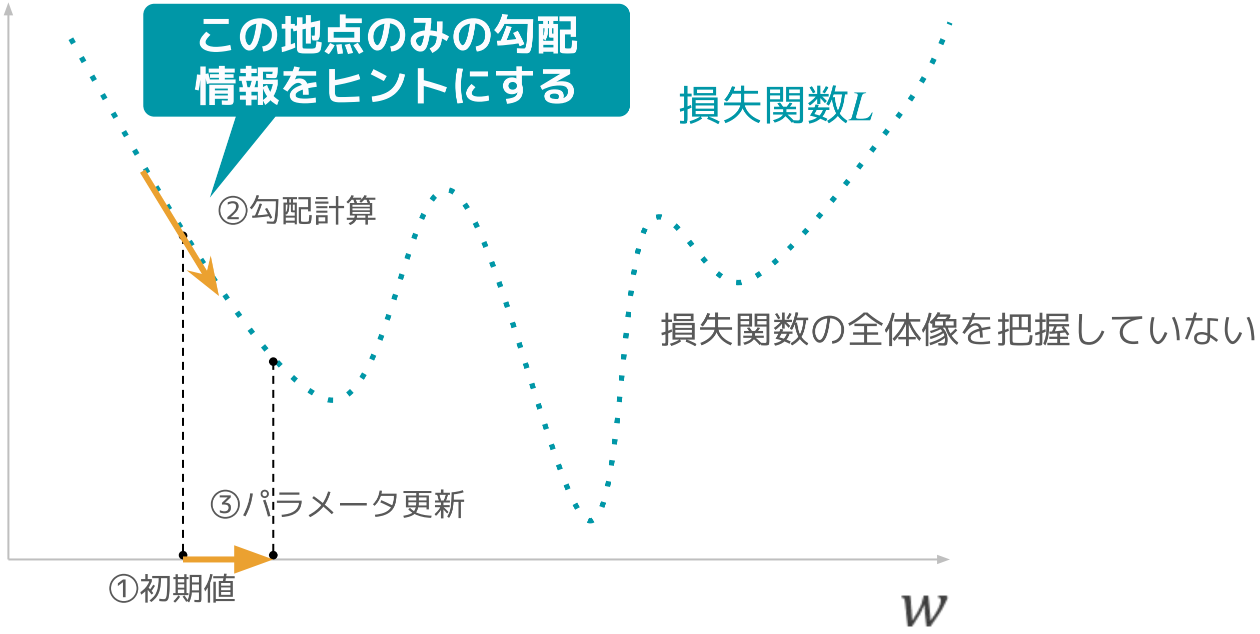


※対象の損失関数によって最適化手法の相性は異なる。図の損失関数の定義式は$L = w_1^2 + 5w_2^2$

最急降下法

最急降下法は、その地点での勾配方向に更新

イメージ図



最急降下法は、その地点での勾配方向に更新

数式

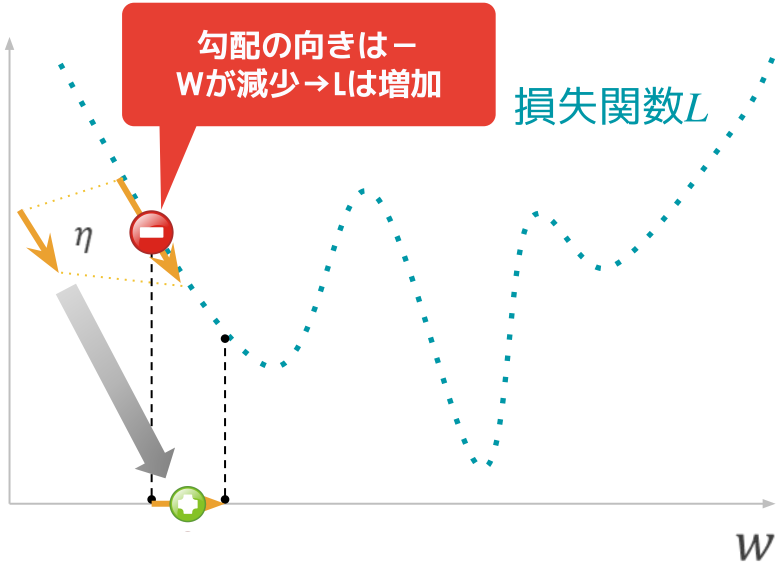
$$w_{t+1} = w_t - \eta \left(\frac{\partial L}{\partial w} \right)_{w=w_t}$$

ハイパーパラメータ(学習率)

η ：どれだけ更新させるか

特徴

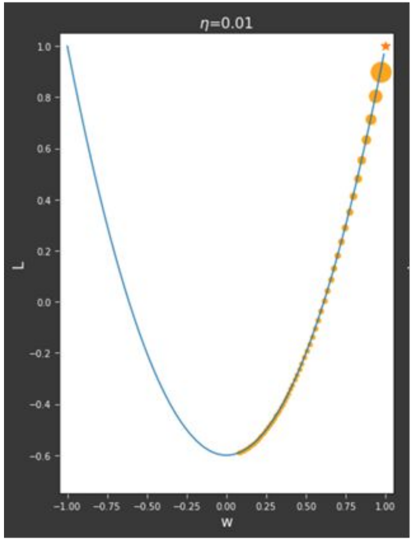
求めた勾配方向とは**逆向き**にその大きさだけパラメータを更新する



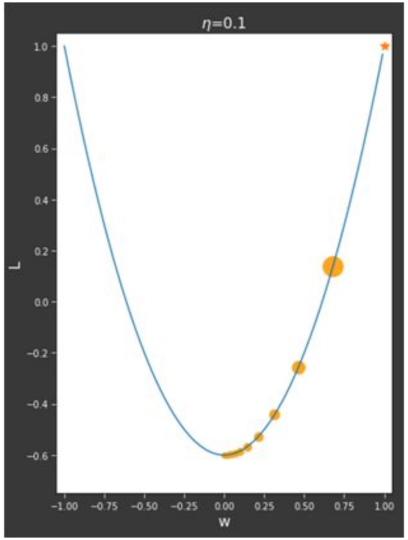
最急降下法は、全データを用いた損失関数を最適化するのでバッチ学習である

収束までの経路図

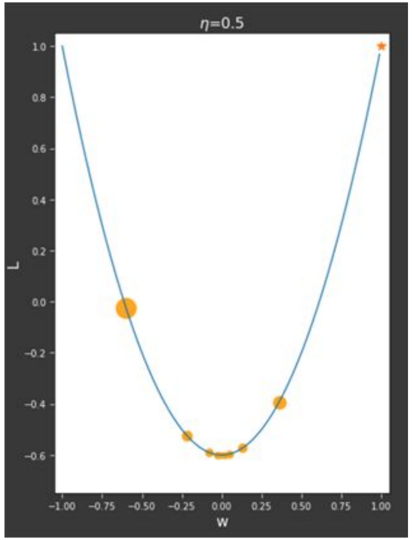
学習率ηが小さすぎ



学習率ηがちょうどいい



学習率ηが大きすぎ



動画で詳細に理解したい方：https://ai-trend.jp/basic-study/neural-network/optimizer/

最急降下法の欠点

最急降下法では「局所的最適解(local minima)」にはまってしまい、本当に低い谷へ落ち着くことが難しい
(⇔全体の最適解を大域的最適解(global minima)と呼んだりもする)

ここから抜け出せなくなる

こちらが望ましい

SGD(確率的勾配降下法)

SGDは、確率的な山の一地点での勾配方向に更新

数式

LはデータXに依存

$$w_{t+1} = w_t - \eta \left(\frac{\partial L(X)}{\partial w} \right)_{w=w_t}$$

ハイパーパラメータ(学習率)

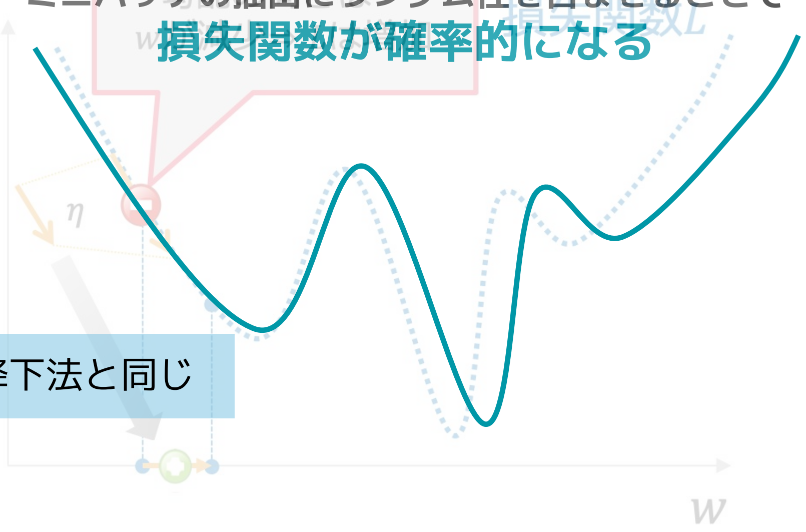
η: どれだけ更新させるか

特徴

求めた勾配方向とは逆向きにその
大きさだけパラメータを更新する

手法は最急降下法と同じ

ミニバッチの抽出にランダム性を含ませることで
損失関数が確率的になる

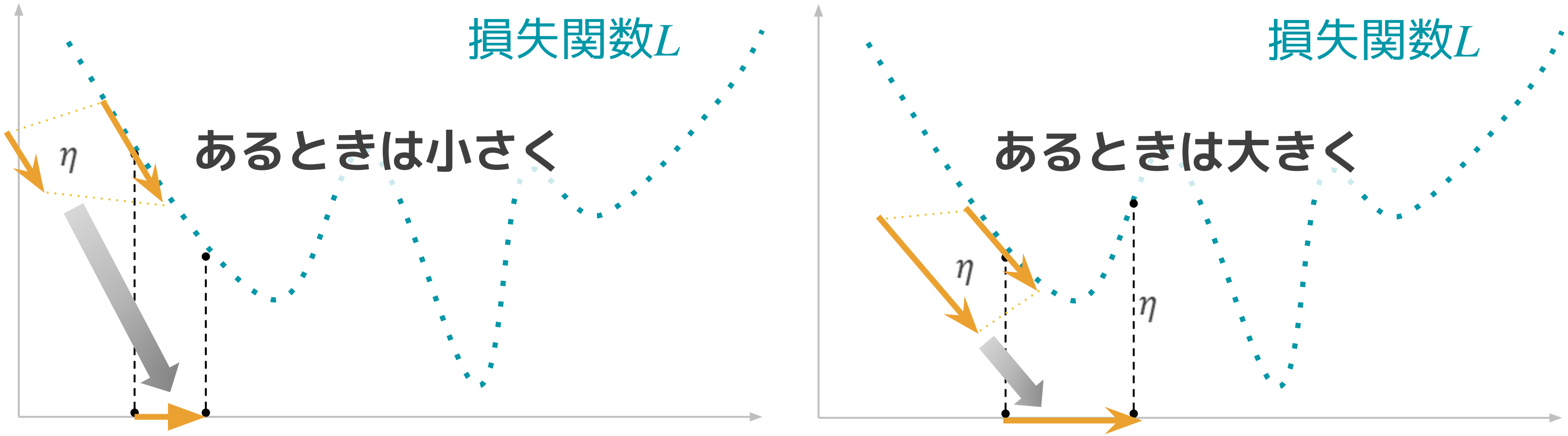


SGDは、グループ化されたサンプルを用いた損失関数を最適化するのでミニバッチ処理である

SGDは、データを変えて損失関数自体を確率的に与えた

【背景】SGDが登場するまでの解決策

例）学習率を確率的に変える

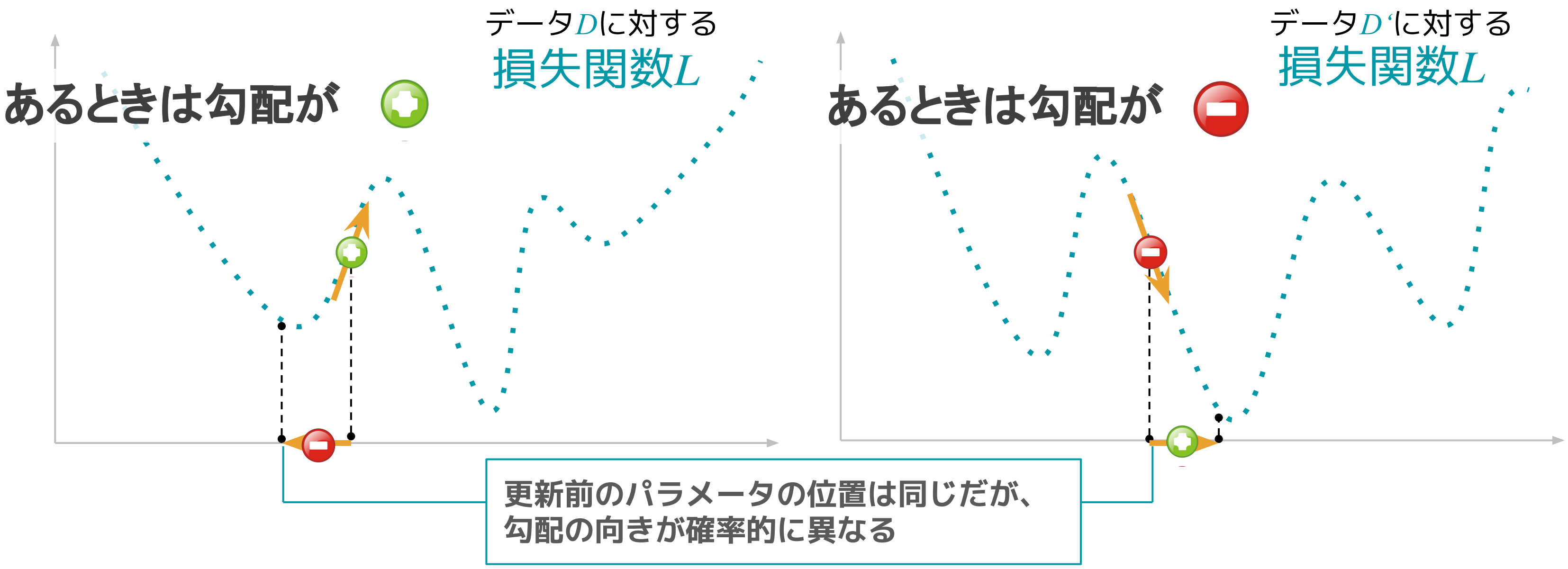


最適化手法を変えようとしていた
しかし、損失関数が複雑すぎてうまくいかない

SGDと最急降下法

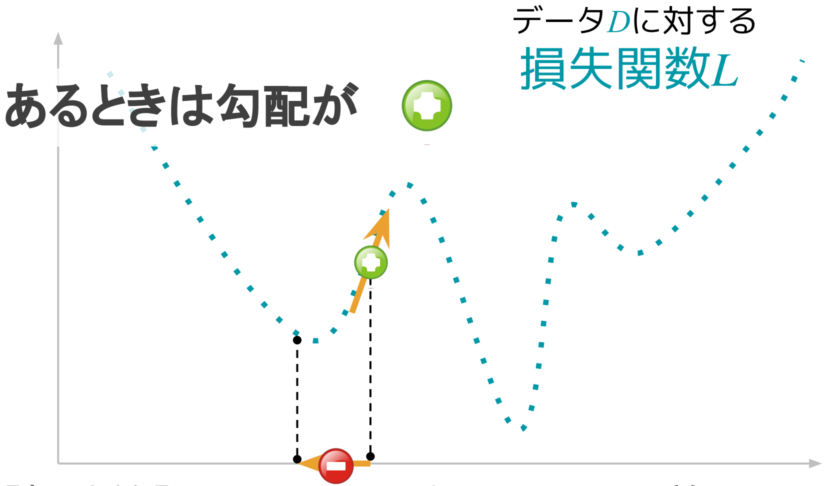
SGDは、データを変えて損失関数自体を確率的に与えた

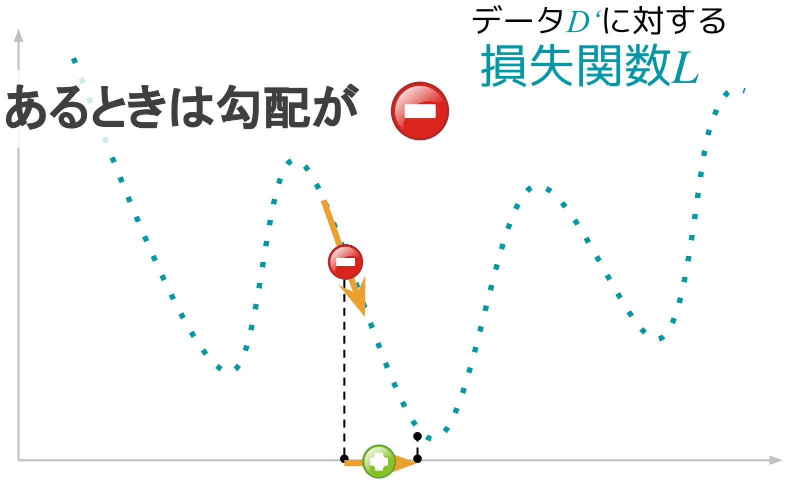
SGDは学習率ではなく、データを変えた



SGDは、最適化の途中で振動して収束が遅くなる

確率的に変化する損失関数を最適化することにも弱点がある。
それは、進む方向がランダムになり振動を助長し収束が遅くなること





【解決策】振動を抑制するように、前回までの勾配によって「勢い」を変化させる。
それがモメンタム

関数平面上で“ボール”が転がるようにパラメータ更新

数式

$$v_{t+1} = \alpha v_t - \eta \left(\frac{\partial L(X)}{\partial w} \right)_{w=w_t}$$

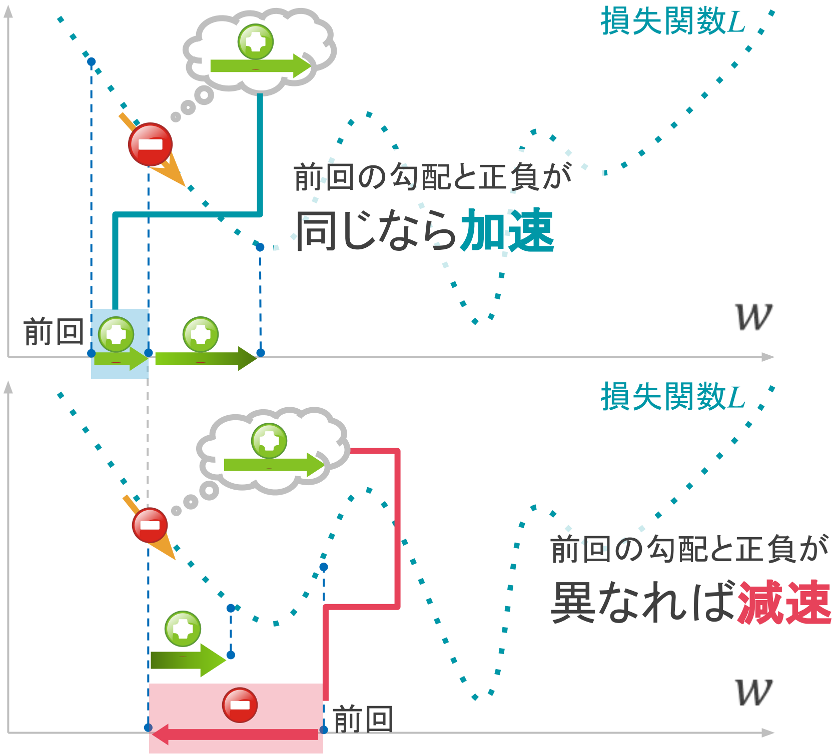
$$w_{t+1} = w_t + v_{t+1}$$

ハイパーパラメータ

η: 今回の損失の勾配の割合(学習率)
α: 前回までの勾配の割合

特徴

SGDに「慣性の法則」を導入
移動平均をとることで振動幅を抑制する



AVILEN

過去の勾配の二乗和を全て記憶し学習率を調整する

数式

$$h_t = h_{t-1} + \left(\frac{\partial L}{\partial w}\right)^2_{w=w_t}$$

$$\eta_t = \frac{\eta_0}{\sqrt{h_t}}$$

$$w_{t+1} = w_t - \eta_t \odot \left(\frac{\partial L}{\partial w}\right)_{w=w_t}$$

行列

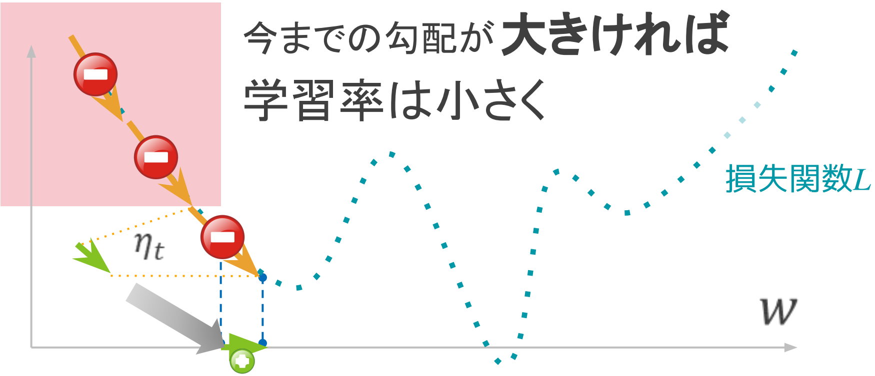
要素ごとの積であり、パラメータごとに異なる学習率を設定する

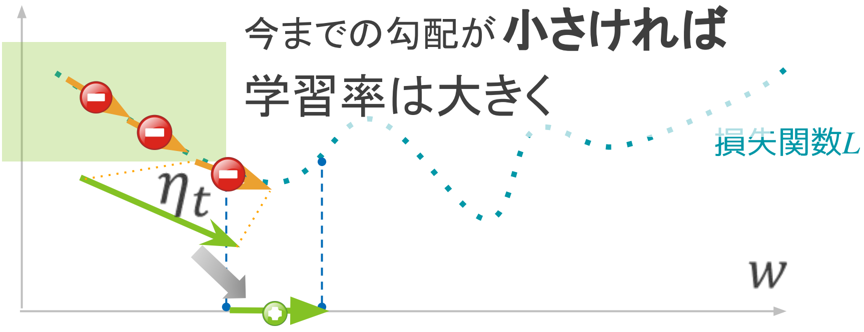
ハイパーパラメータ(学習率)

η_0: 学習率の初期値

特徴

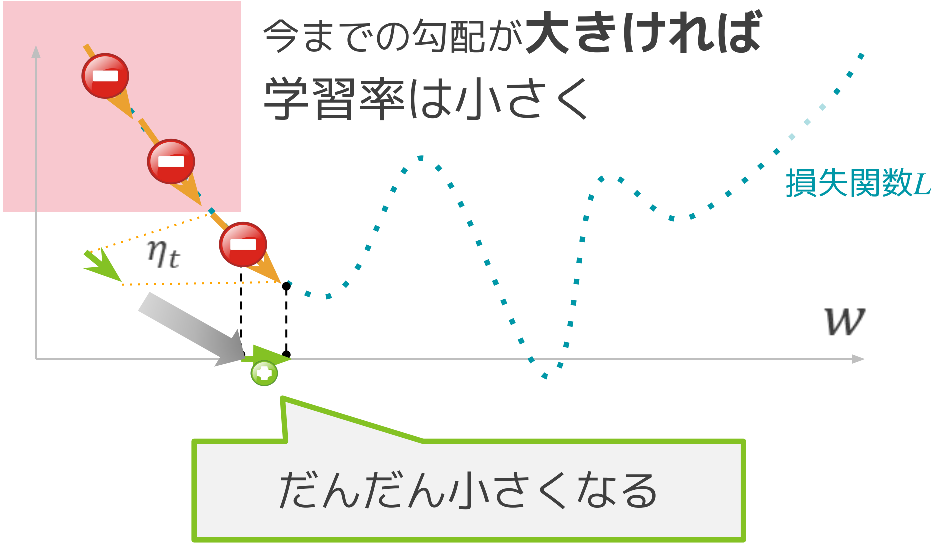
✓**パラメータ**ごとに学習率を更新
✓前回までに大きく更新されたパラメータの学習率は、より小さく調整される。その逆も然り。(右図参照)





AdaGradの問題点

**学習が進むにつれて、学習率が小さくなり、やがて0になる
ゆえに学習が進まなくなる可能性がある**



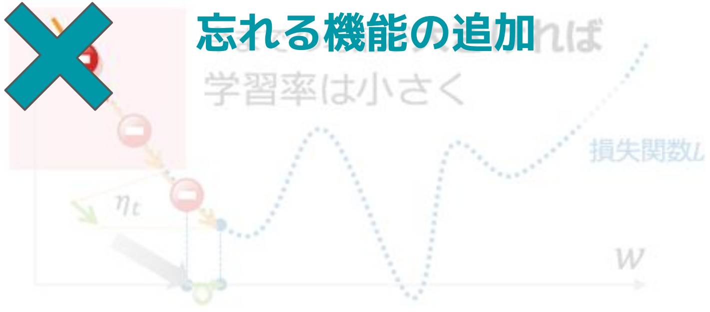
古い情報は忘れて、新しい勾配情報が反映されるように記憶し学習率を調整する

数式

$$h_t = \alpha h_{t-1} + (1-\alpha)\left(\frac{\partial L}{\partial w}\right)^2_{w=w_t}$$

$$\eta_t = \frac{\eta_0}{\sqrt{h_t}}$$

$$w_{t+1} = w_t - \eta_t \left(\frac{\partial L}{\partial w}\right)_{w=w_t}$$



ハイパーパラメータ(学習率)

η_0: 学習率の初期値
α: 新古の勾配情報の割合

特徴

過去の勾配情報と新しい勾配情報の割合を調整して記憶する。

AVILEN

勾配の２乗を記憶した「ボール」が関数平面上を転がる

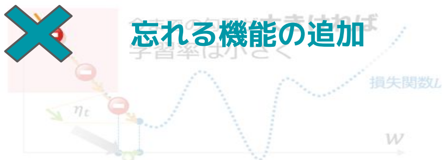
数式

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

初期段階の
不安定性の解消
(後述)

＋



忘れる機能の追加

忘れる機能の追加

特徴

Momentum + RMSProp
とりあえず「Adam」といった風潮がある

引用：https://arxiv.org/pdf/1412.6980.pdf

AVILEN

イテレーション数tでバイアス補正することで、初期段階の不安定性を解消

改善前の【課題】初期段階では $v_1 = 0$ のため、変動が大きくなり不安定になる

$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \quad \text{(イテレーション数}t\text{における勾配)}$$
$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \quad \text{(新旧の一次モーメントの加重平均)}$$
$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot {g_t}^2 \quad \text{(新旧の二次モーメントの加重平均)}$$
$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot m_t/(\sqrt{v_t} + \epsilon) \quad \text{(パラメータ更新)}$$

【改善施策】イテレーション数tでバイアス修正

$$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1}) \quad \text{(イテレーション数}t\text{における勾配)}$$
$$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t \quad \text{(新旧の一次モーメントの加重平均)}$$
$$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot {g_t}^2 \quad \text{(新旧の二次モーメントの加重平均)}$$
$$\widehat{m_t} \leftarrow m_t/(1-\beta_1^t) \quad \text{(イテレーション数}t\text{でバイアス補正された一次モーメント)}$$
$$\widehat{v_t} \leftarrow v_t/(1-\beta_2^t) \quad \text{(イテレーション数}t\text{でバイアス補正された二次モーメント)}$$
$$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m_t}/(\sqrt{\widehat{v_t}} + \varepsilon) \quad \text{(パラメータ更新)}$$

最適化手法実装

```python
# 最適化したい関数の定義
def func(x,):
    return x**2

# パラメータ初期値
x = torch.tensor(-2.0, requires_grad=True)
params = [x]
# 最適化手法を定義
# lr: 学習率
opt = optim.SGD(params, lr=0.01)  # SGD
opt = optim.SGD(params, lr=0.001, momentum=0.9)  # Momentum
opt = optim.SGD(params, lr=0.001, momentum=0.9, nesterov=True)  # Nesterov
opt = optim.Adagrad(params, lr=1.0)  # AdaGrad
opt = optim.RMSprop(params, lr=1.0)  # RMSProp
opt = optim.Adam(params, lr=1.0)  # Adam

#関数の出力に対して勾配を求め、optimizerで更新を繰り返す
for _ in range(80):
    opt.zero_grad()  # 勾配を０にセット(PyTorchのデフォルトは勾配が加算されていく)
    outputs = func(x)
    outputs.backward()  # 勾配計算
    opt.step()  # パラメータ更新
```

この章で学習したこと

1）バッチ処理（順伝播）

2）バッチ処理（逆伝播）

3）最適化手法

AVILEN