NKTLab マニュアル

Convolutional Neural Network(CNN)の基礎

ver. 1.0.1

最終更新日: 2023/10/30

最終更新者: 藤澤 大世(3期)

YNU Initiative for Global Arts & Sciences
YOKOHAMA National University

パッチノート

2023/10/16 [ver. 0.0.0]　リリース(藤澤・3期)

2023/10/24 [ver. 0.0.1]　56, 58, 71, 76ページを修正(藤澤・3期)

2023/10/28 [ver. 1.0.0]　5, 6章を追加(藤澤・3期)

2023/10/30 [ver. 1.0.1]　目次にハイパーリンクを追加(藤澤・3期)

目次

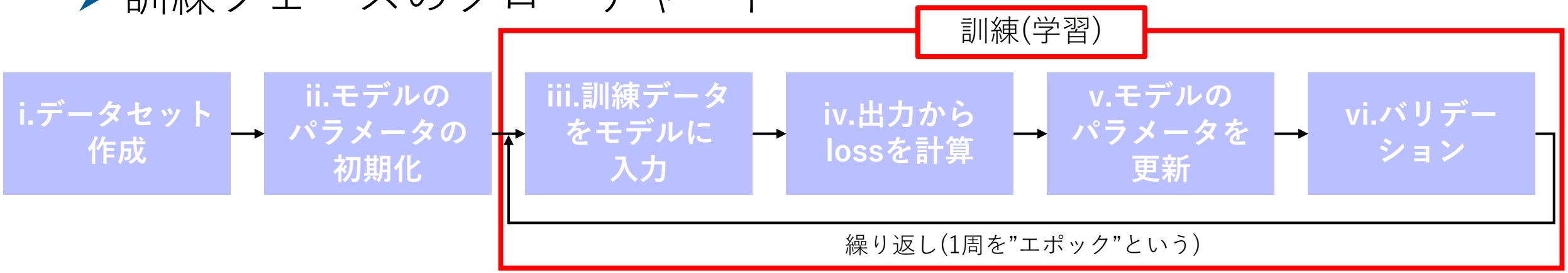
1. 理論編

概要

- **機械学習のフェーズ**
  1. 訓練(学習)フェーズ
     - 大量のデータと教師ラベルを用いてモデルを訓練し、既知のデータの特徴を学習させる
  2. テスト(推論)フェーズ
     - 訓練が終わったモデルを、別に取っておいた未知のデータで評価する

概要

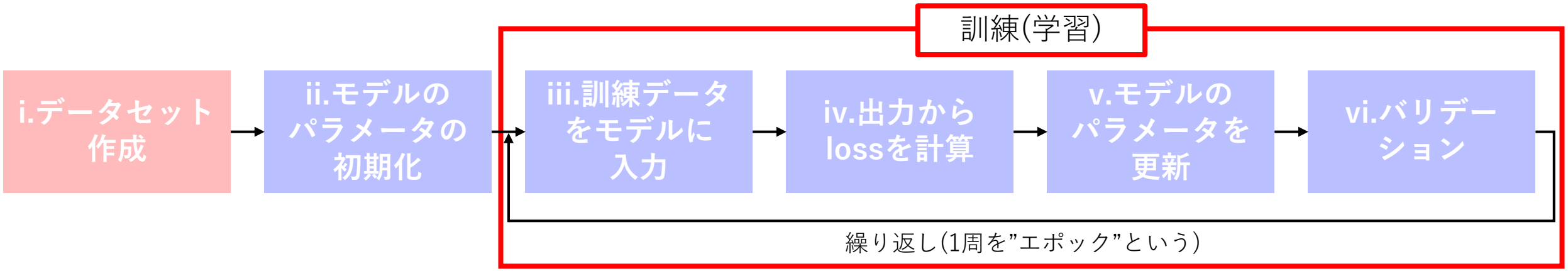
- **Deep Neural Network: DNN**
  - 多層のニューラルネットワークからなる機械学習モデルを構築し、回帰または分類タスクを解く手法。教師あり学習の一種。
    - モデル…「≒関数」と思ってよい。ある入力xに対して、ある出力yを返す。
    - 回帰の例…入力x: 顔写真(3次元配列), 出力y: 推定年齢(実数)
    - 分類の例…入力x: 名前(文字列), 出力y: 男性/女性である確信度(1次元配列)
  - 訓練フェーズのフローチャート



i. データセット作成

- **データの種類**
  - 訓練(train)データ: モデルのパラメータを更新するためのデータ。問題集。
  - 検証(validation)データ: テストに用いるモデルを選んだり、ハイパーパラメータを更新したりするためのデータ。模試。
  - テスト(test)データ: モデルを評価するためのデータ。受験本番。

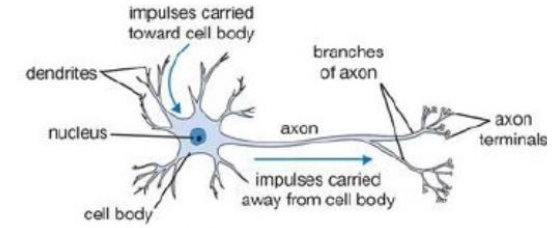
  → 各データに重複があってはならない(カンニングの禁止)。



ii. さまざまなモデル

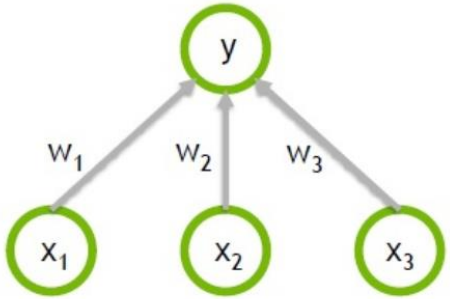
- **ニューラルネットワーク(Neural Network: NN)**
  - 人間の脳内の神経細胞(ニューロン)とそれらの結合を計算グラフで模倣
    - ノードはニューロンへの入力を表現
    - エッジはニューロン間の接続を表現
      - エッジが持つパラメータを訓練により更新していく

神経回路網



スタンフォード大学cs231講義ノートより

人工ニューロン



$y=F(w_1x_1+w_2x_2+w_3x_3)$

$F(x)=\max(0,x)$

訓練(学習)

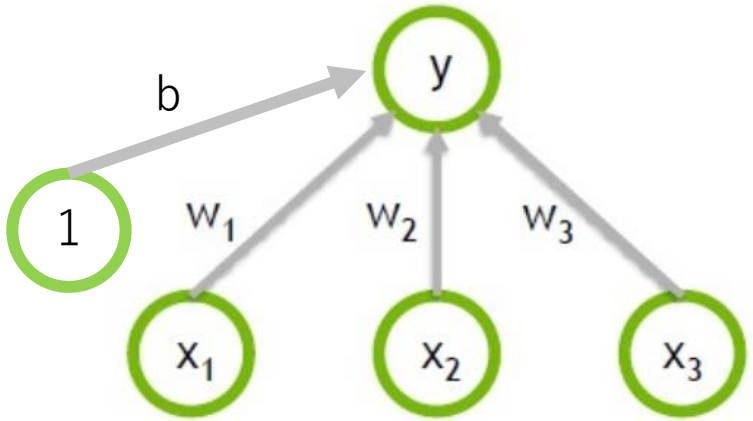
i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

繰り返し(1周を”エポック”という)

ii. さまざまなモデル

- **ニューラルネットワーク(Neural Network: NN)**

「単純パーセプトロン」



バイアス　　ウェイト

- $y = F(b + w_1x_1 + w_2x_2 + w_3x_3)$ ここまでは線形関数
  - エッジが持つパラメータにはバイアスとウェイトの2つがある　後で訓練により更新する
  - F: 活性化関数(activation function)
    - 複雑な関数を表現するためReLUなどの非線形関数が用いられる

ReLU: f(x) = max(0, x)

訓練(学習)

i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

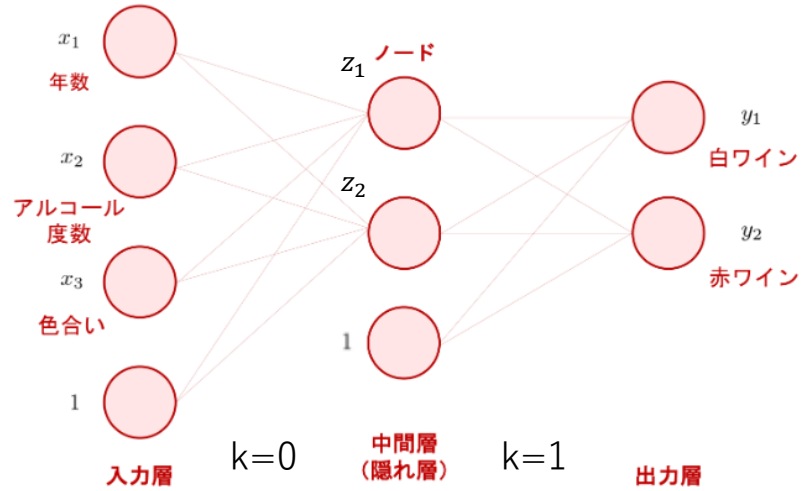
繰り返し(1周を”エポック”という)

ii. さまざまなモデル

すべてのノードがつながっているから各層を「全結合層」という

- **多層パーセプトロン(Multi Layer Perceptron: MLP)**
  - k番目の層間におけるj番目のノードからi番目のノードへのウェイト: w_{ij}^{k}
  - k番目の層間におけるi番目のノードへのバイアス: b_{i}^{k}

$$\rightarrow z_i = F\left(\sum_j \left(w_{ij}^{0} x_j + b_i^{0}\right)\right),\ y_i = F\left(\sum_j \left(w_{ij}^{1} z_j + b_i^{1}\right)\right)$$



訓練(学習)

i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

繰り返し(1周を”エポック”という)

ii. さまざまなモデル

すべてのノードがつながっているから各層を「全結合層」という

● 多層パーセプトロン(Multi Layer Perceptron: MLP)

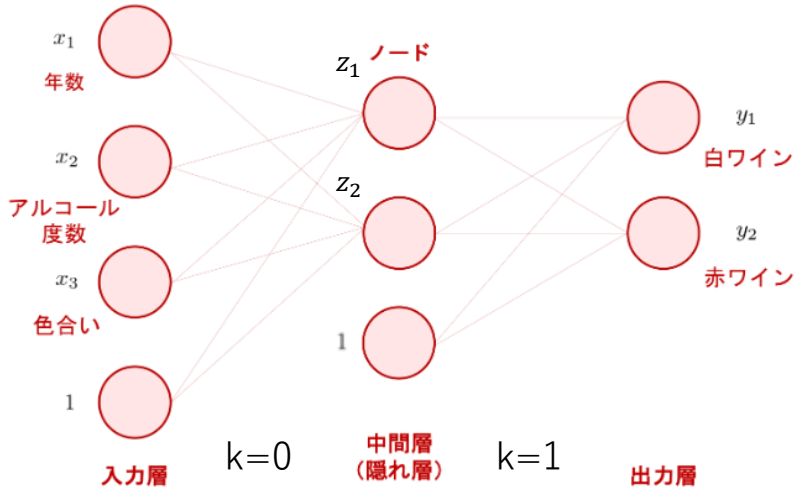
➢ $z_i = F(\sum_j(w_{ij}^1 x_j + b_i^1)),\ y_i = F(\sum_j(w_{ij}^2 z_j + b_i^2))$ を行列で表現

■ $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \boldsymbol{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}, \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, W_0 = \begin{bmatrix} w_{11}^0 & w_{12}^0 & w_{13}^0 \\ w_{21}^0 & w_{22}^0 & w_{23}^0 \end{bmatrix}, \boldsymbol{b}_0 = \begin{bmatrix} b_1^0 \\ b_2^0 \end{bmatrix}, W_1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 \\ w_{21}^1 & w_{22}^1 \end{bmatrix}, \boldsymbol{b}_1 = \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}$

$\rightarrow \boldsymbol{z} = F(W_0\boldsymbol{x} + \boldsymbol{b}_0),\ \boldsymbol{y} = F(W_1\boldsymbol{z} + \boldsymbol{b}_1)$

$\therefore \boldsymbol{y} = F(W_1 F(W_0\boldsymbol{x} + \boldsymbol{b}_0) + \boldsymbol{b}_1)$

↑各層が表す関数の合成関数



訓練(学習)

i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

繰り返し(1周を”エポック”という)

ii. さまざまなモデル

すべてのノードがつながっているから各層を「全結合層」という

● 多層パーセプトロン(Multi Layer Perceptron: MLP)

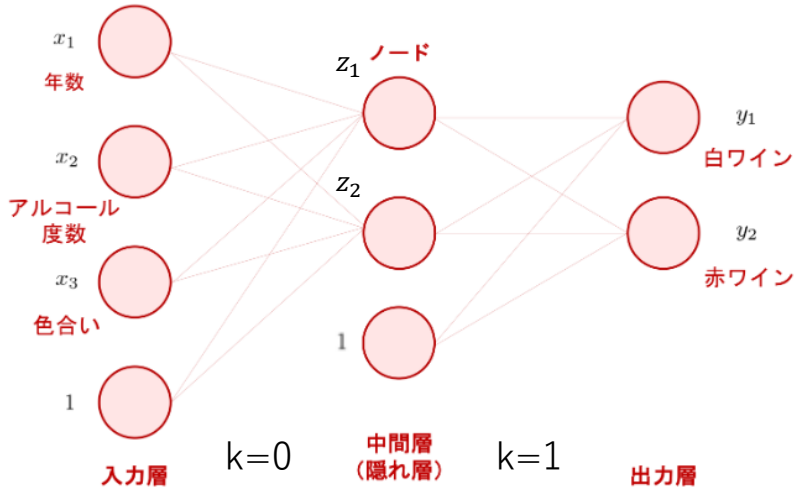
➢ バイアスをウェイトの行列に取り込むと

■ $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{bmatrix}, \boldsymbol{z} = \begin{bmatrix} z_1 \\ z_2 \\ 1 \end{bmatrix}, \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}, W_0 = \begin{bmatrix} w_{11}^0 & w_{12}^0 & w_{13}^0 & b_1^0 \\ w_{21}^0 & w_{22}^0 & w_{23}^0 & b_2^0 \end{bmatrix}, W_1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 & b_1^1 \\ w_{21}^1 & w_{22}^1 & b_2^1 \end{bmatrix}$

$\rightarrow \boldsymbol{z} = F(W_0\boldsymbol{x}), \boldsymbol{y} = F(W_1\boldsymbol{z})$

$\therefore \boldsymbol{y} = F\big(W_1F(W_0\boldsymbol{x})\big) = (f_1 \circ f_0)(\boldsymbol{x})$

↑各層が表す関数$f_k(\boldsymbol{x}) = F(W_k\boldsymbol{x})$の合成関数



訓練(学習)

i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

繰り返し(1周を”エポック”という)

ii. さまざまなモデル

すべてのノードがつながっているから各層を「全結合層」という

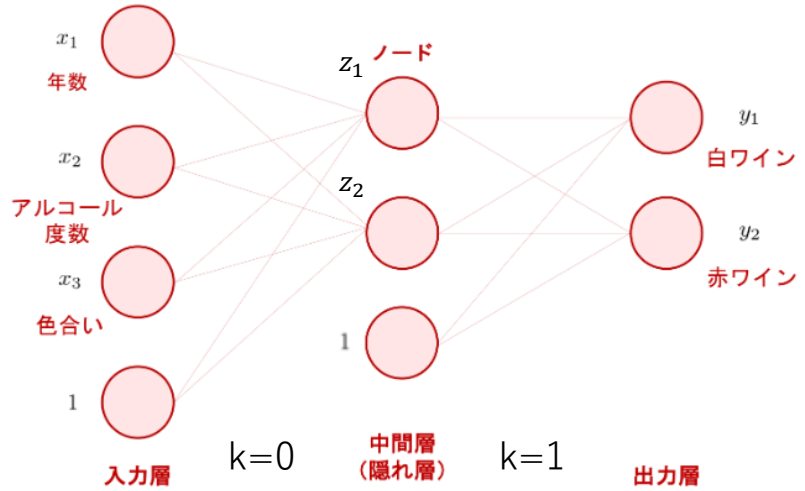
● 多層パーセプトロン(Multi Layer Perceptron: MLP)

- 隠れ層がk層ある場合、同様にして

$$\boldsymbol{y} = F\left(W_k F\left(W_{k-1} F\left(W_{k-2} \cdots F(W_0 \boldsymbol{x}) \cdots\right)\right)\right)$$
$$= (f_k \circ f_{k-1} \circ f_{k-2} \circ \cdots \circ f_0)(\boldsymbol{x})$$

- 活性化関数が非線形関数である必要性
  - 上式でFがないと、$\boldsymbol{y}$は$\boldsymbol{x}$の1次関数となる

  → 線形分離不可能なタスクに対応不可



訓練(学習)

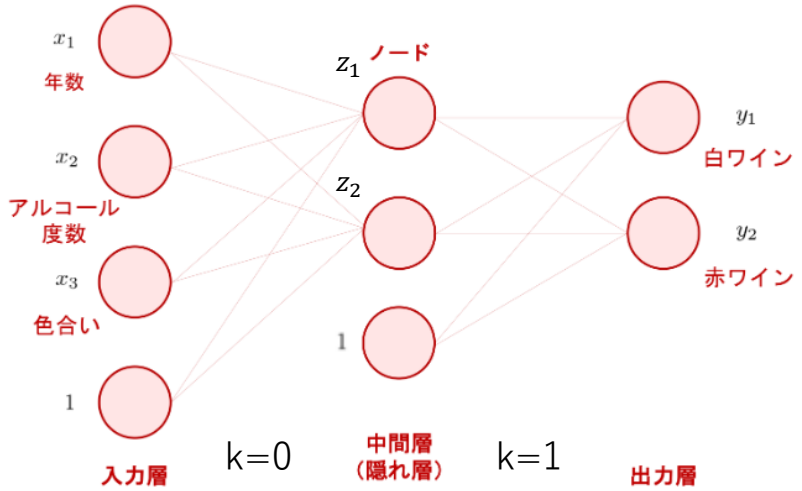
i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

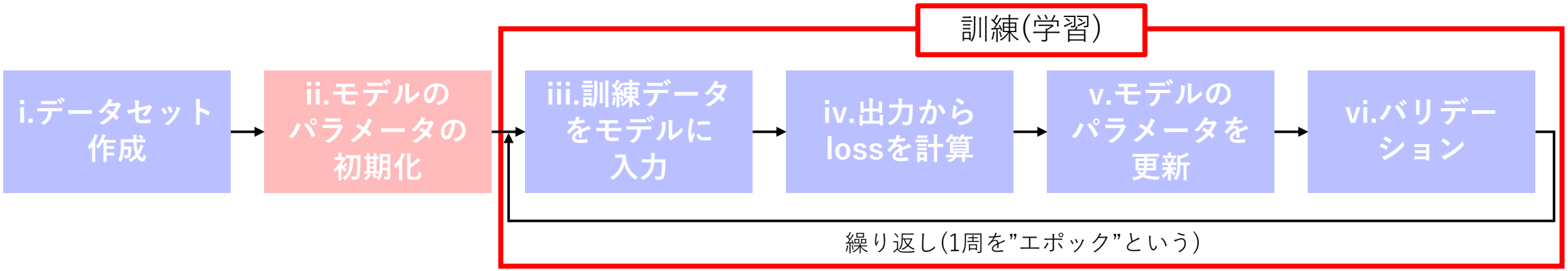
繰り返し(1周を”エポック”という)

ii. さまざまなモデル

すべてのノードがつながっているから各層を「全結合層」という

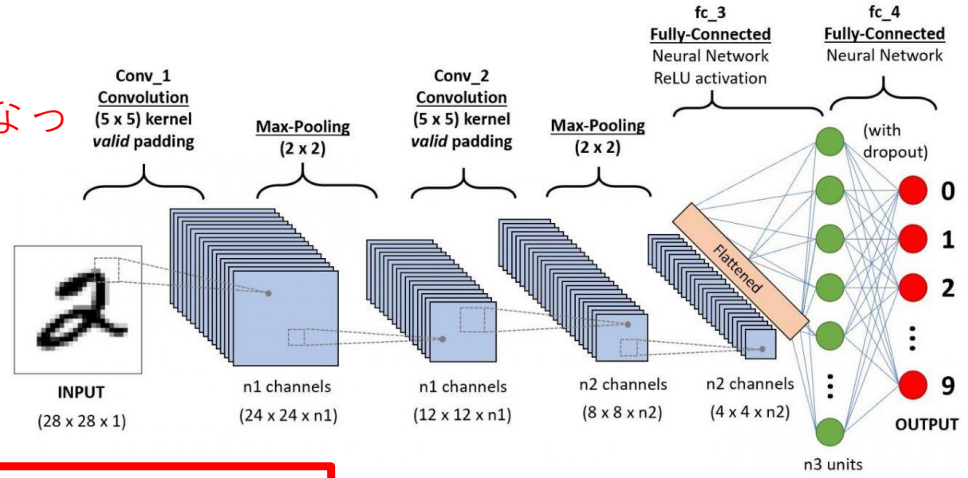
- **多層パーセプトロン(Multi Layer Perceptron: MLP)**
  - 訓練で更新する
    - ウェイト・バイアス(エッジのパラメータ)
  - 訓練で更新しない(ハイパーパラメータ)
    - 各層のノード数(NNの構造)
    - 活性化関数　など





ii. さまざまなモデル

- **Convolutional Neural Network: CNN**
  - 畳み込み層とプーリング層を有するニューラルネットワーク
    - 結合を隣接ピクセル間のみに限定
      →画像分類問題に強み ↓ノードが特徴マップになっているNNと考えてもよい
    - ノードは2~3次元に並べられる
      - 2次元のものを「特徴マップ」という
      - 3次元目(奥行)は「チャネル」という



訓練(学習)

i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

繰り返し(1周を”エポック”という)

ii. さまざまなモデル

- **Convolutional Neural Network: CNN**
  - 畳み込み(conv.)層
    - 各チャネルに1つ「カーネル」が存在
      - カーネルはエッジに対応
        (各値がウェイトとバイアスを持つ)

    → 入力へのカーネルの適用箇所をずらしながら畳み込み演算を実施し、特徴マップを得る

後で訓練により更新する

入力データ (5x5)

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

＊

カーネル (3x3)

1	2	3
4	5	6
7	8	9

→

?		

$$1\times1+2\times2+3\times3+4\times4+5\times5+6\times6+7\times7+8\times8+9\times9=411$$

訓練(学習)

i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

繰り返し(1周を”エポック”という)

ii. さまざまなモデル

- **Convolutional Neural Network: CNN**
  - 畳み込み(conv.)層
    - 主なハイパーパラメータ
      - チャネル数
      - カーネルサイズ
      - パディング
      - ストライド

※詳細は「ゼロから作るDeep Learning―Pythonで学ぶディープラーニングの理論と実装」を参照。

後で訓練により更新する

入力データ (5x5)

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

＊

カーネル (3x3)

1	2	3
4	5	6
7	8	9

?		

$1\times1+2\times2+3\times3+4\times4+5\times5+6\times6+7\times7+8\times8+9\times9=411$

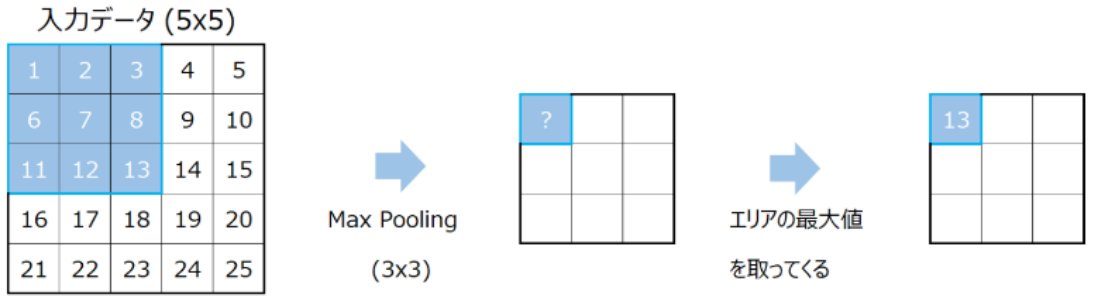
訓練(学習)

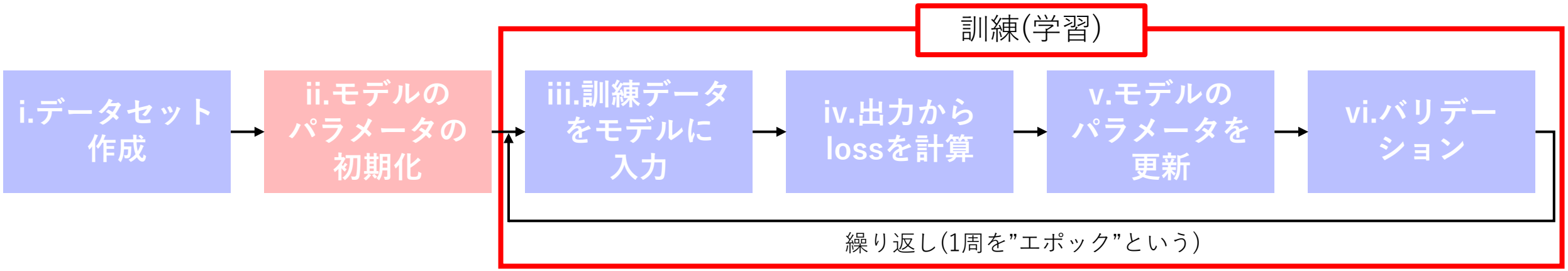
i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

繰り返し(1周を”エポック”という)

ii. さまざまなモデル

- **Convolutional Neural Network: CNN**
  - プーリング(pool.)層
    - 各入力に対して、ある範囲の代表値を抜き出すことを、その範囲をずらしながら繰り返し、特徴マップを得る
      - 訓練するパラメータはない
      - プーリング層の前後でチャネル数は変化しない

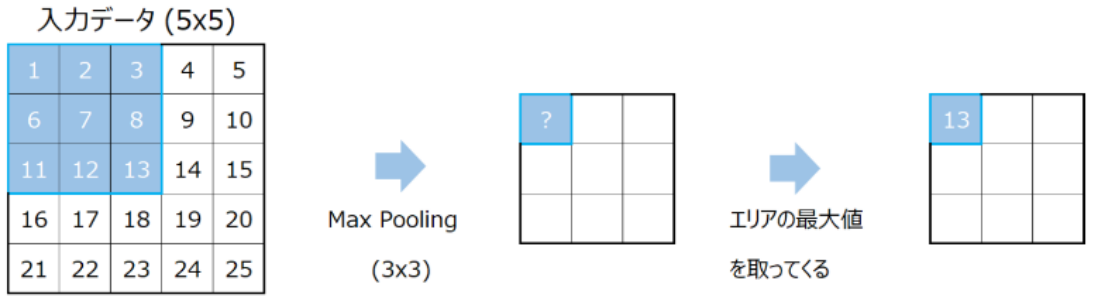


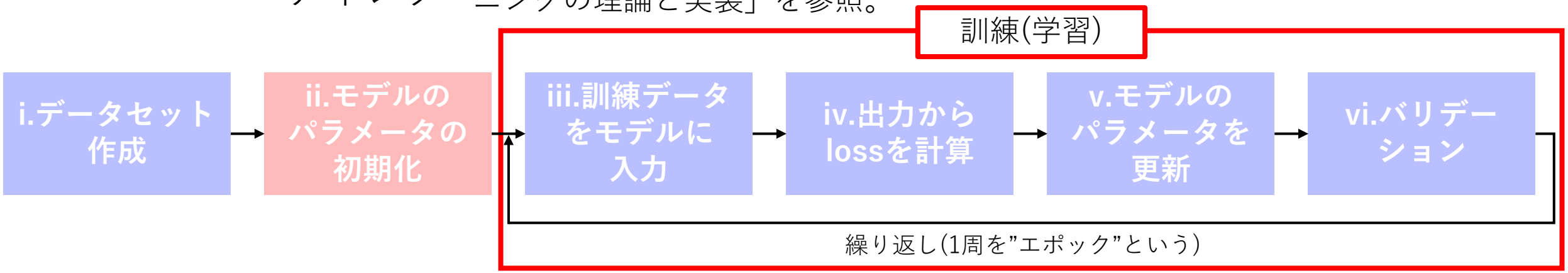


ii. さまざまなモデル

- **Convolutional Neural Network: CNN**
  - プーリング(pool.)層
    - 主なハイパーパラメータ
      - プーリングに使う代表値
      - カーネルサイズ
      - ストライド
      - パディング

※詳細は「ゼロから作るDeep Learning―Pythonで学ぶディープラーニングの理論と実装」を参照。

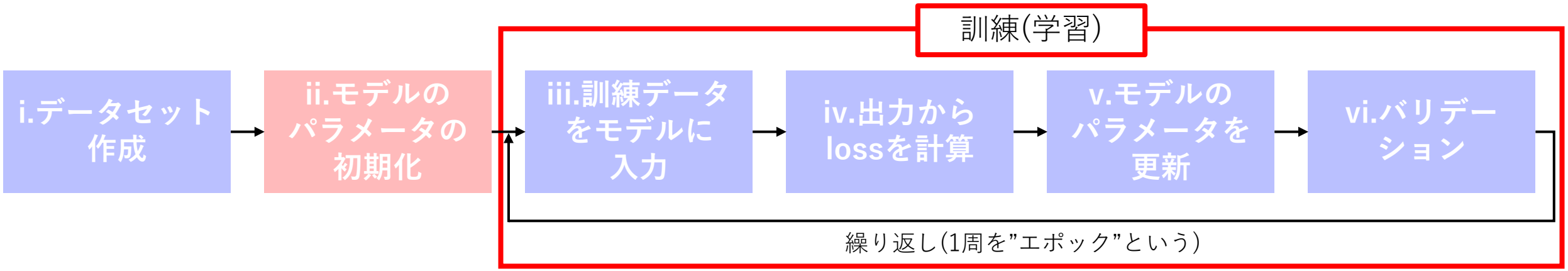




ii. さまざまなモデル

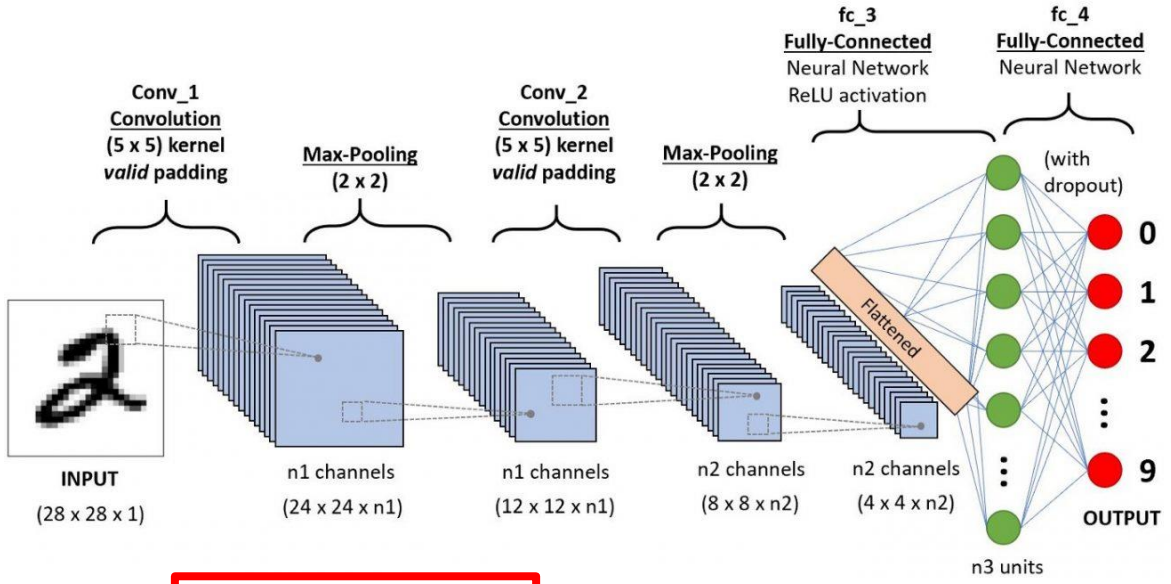
- **Convolutional Neural Network: CNN**
  - conv.層とpool.層の演算の定式化
    - チョー面倒くさい上難解すぎるためとばす。
      - そもそも全結合のとき入力$\boldsymbol{x}$や途中のノードが一次元配列だったが、CNNでは入力$\boldsymbol{X}$や途中のノードが(height, width, channel)の三次元配列になる。死ぬ。
      - でも行列で同じように表せることは一緒。

※詳細は「ゼロから作るDeep Learning―Pythonで学ぶディープラーニングの理論と実装」を参照。



ii. さまざまなモデル

- **Convolutional Neural Network: CNN**
  - モデルの構造
    - 始めにconv.層とpool.層で特徴抽出
    - 1次元ベクトル(平滑)化して全結合層へ
    - あとは通常のNNと同じ



訓練(学習)

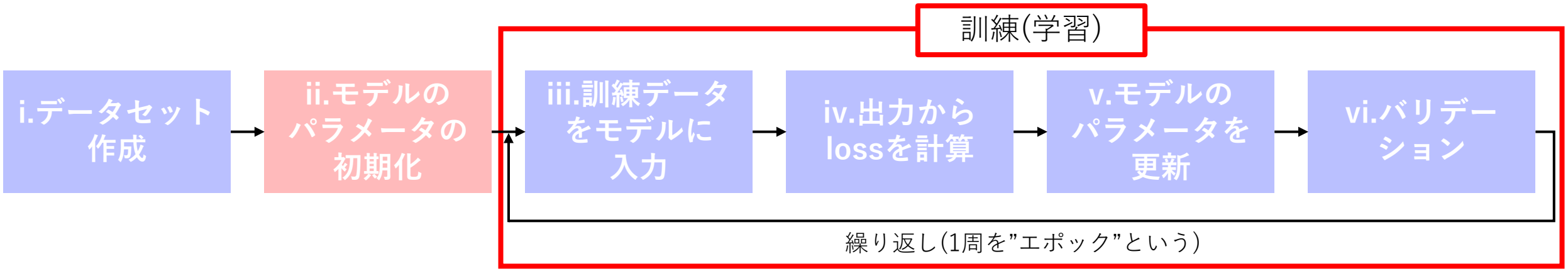
i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

繰り返し(1周を”エポック”という)

ii. さまざまなモデル

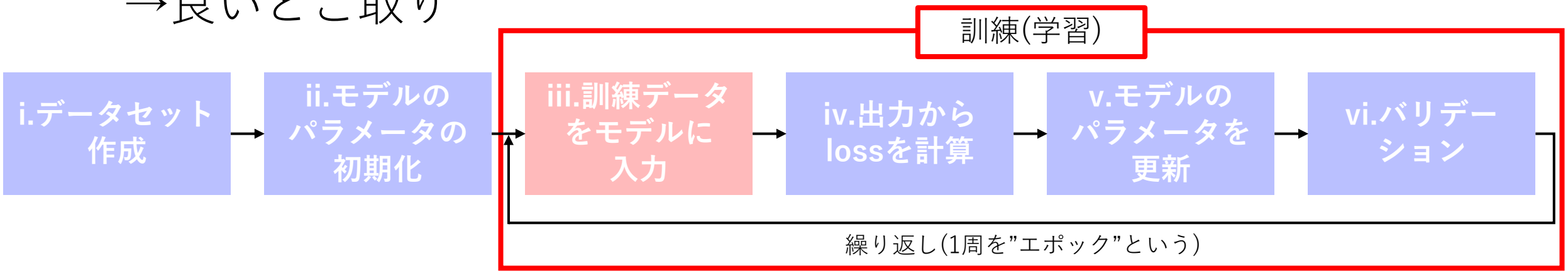
- **パラメータの初期化**
  - NNの各ノードに割り当てられたパラメータを、「ランダムかついい感じに」初期化する
    - そのときの目標や生じる問題点、それを回避するためのアルゴリズムなど、様々な議論があるが、とばす。

※詳細は「ゼロから作るDeep Learning―Pythonで学ぶディープラーニングの理論と実装」を参照。



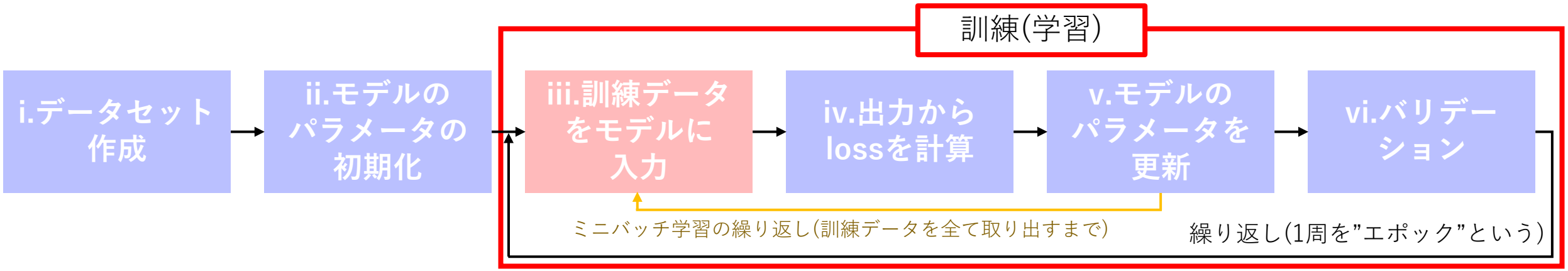
iii. データの入力

- **データの入力方法と訓練の進め方**
  - モデルに訓練データを全部まとめて入力: **バッチ学習**
    →最も理想的だがメモリに限りあり 大きいデータセットに不向き
  - モデルに訓練データを一件ずつ入力: **オンライン学習**
    →メモリの問題はないが過学習しやすいなどの問題あり
  - モデルに訓練データを数件ずつまとめて入力: **ミニバッチ学習**
    →良いとこ取り



iii. データの入力

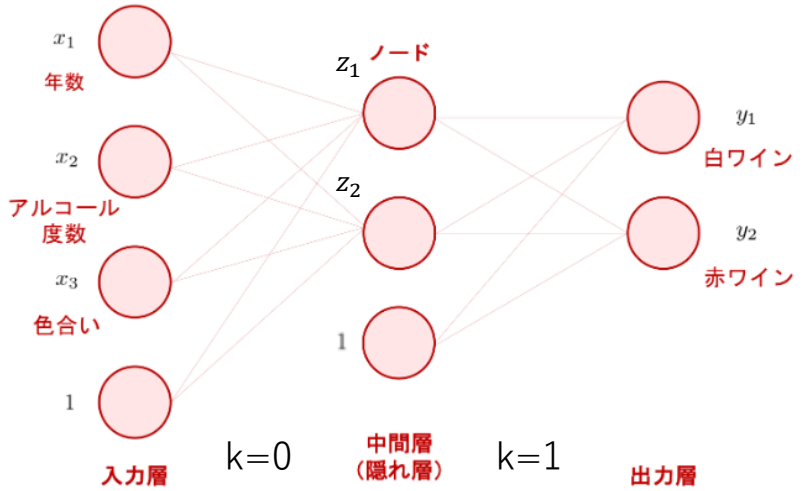
- **ミニバッチ学習: 訓練のループが入れ子になるので注意！**
  - 事前に設定した「バッチサイズ」件数分をまとめてモデルに入力して「v.パラメータ更新」まで行い、これを訓練データを全て取り出すまで繰り返す
  - 訓練データを全て取り出したら、次のエポックへ進む

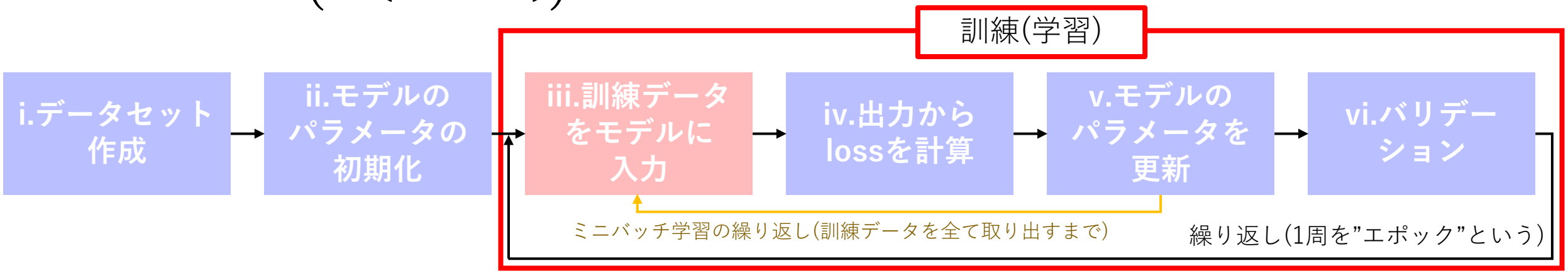


iii. データの入力

- **ミニバッチ学習: 訓練のループが入れ子になるので注意！**
  - バッチサイズbに対して、i件目のデータに対する各ベクトルを $\boldsymbol{x_i} = \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \\ 1 \end{bmatrix}, \boldsymbol{z}_i = \begin{bmatrix} z_{i1} \\ z_{i2} \\ 1 \end{bmatrix}, \boldsymbol{y}_i = \begin{bmatrix} y_{i1} \\ y_{i2} \end{bmatrix}$ とすれば、（ただし $W_0 = \begin{bmatrix} w_{11}^0 & w_{12}^0 & w_{13}^0 & b_1^0 \\ w_{21}^0 & w_{22}^0 & w_{23}^0 & b_2^0 \end{bmatrix}, W_1 = \begin{bmatrix} w_{11}^1 & w_{12}^1 & b_1^1 \\ w_{21}^1 & w_{22}^1 & b_2^1 \end{bmatrix}$）
    - $X = [\boldsymbol{x_1}, \boldsymbol{x_2}, \cdots, \boldsymbol{x_b}], Z = [\boldsymbol{z_1}, \boldsymbol{z_2}, \cdots, \boldsymbol{z_b}], Y = [\boldsymbol{y_1}, \boldsymbol{y_2}, \cdots, \boldsymbol{y_b}]$ に対して、

$$Y = F\left(W_1\left(F(W_0X)\right)\right) = (f_1 \circ f_0)(X)$$ ←各層が表す関数 $f_k(X) = F(W_kX)$ の合成関数



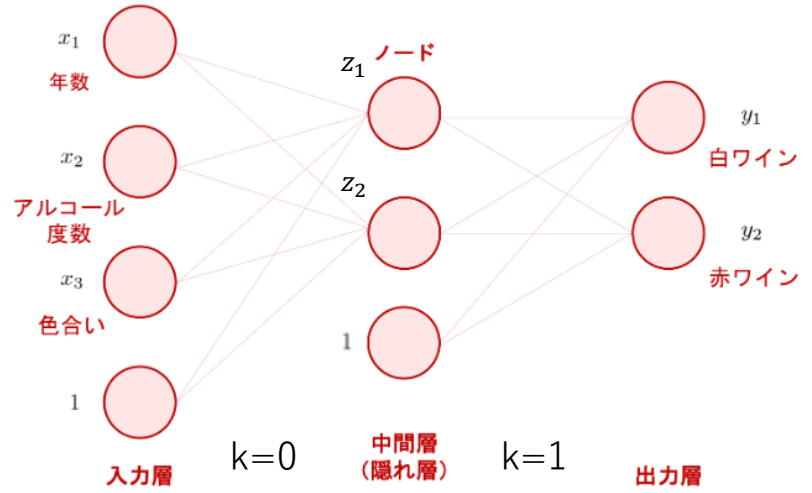


iii. データの入力

- **ミニバッチ学習: 訓練のループが入れ子になるので注意！**
  - 隠れ層がk層ある場合、同様にして

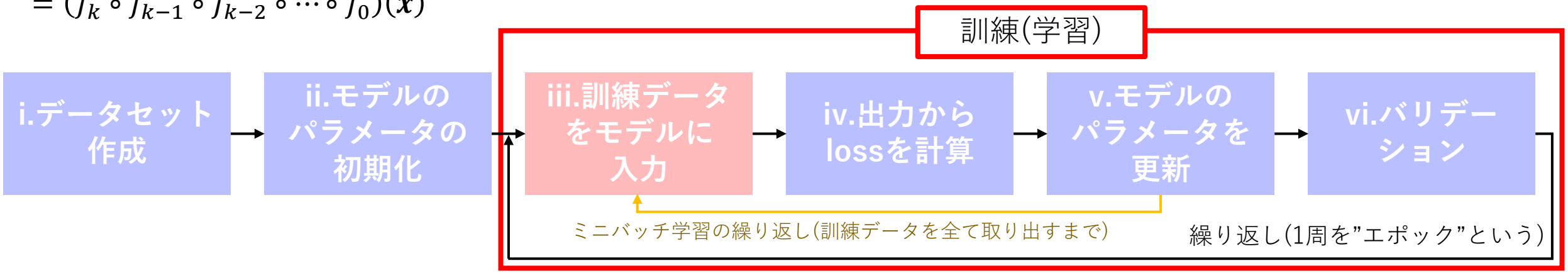
$$Y = F\left(W_k F\left(W_{k-1} F\left(W_{k-2} \cdots F(W_0 X) \cdots\right)\right)\right)$$
$$= (f_k \circ f_{k-1} \circ f_{k-2} \circ \cdots \circ f_0)(X)$$

←入力X は(batchsize, vec_size)の二次元配列
CNNだと(batchsize, height, width, channel)の四次元配列



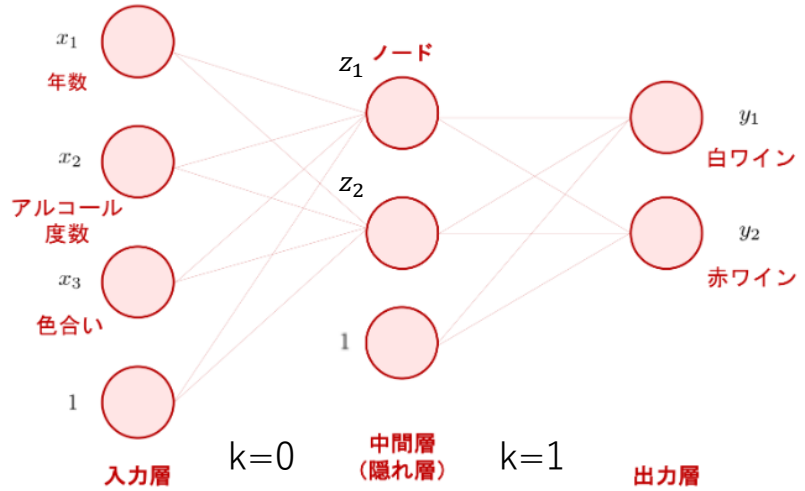
(cf.) オンライン学習(入力一件ずつ)と同じ！

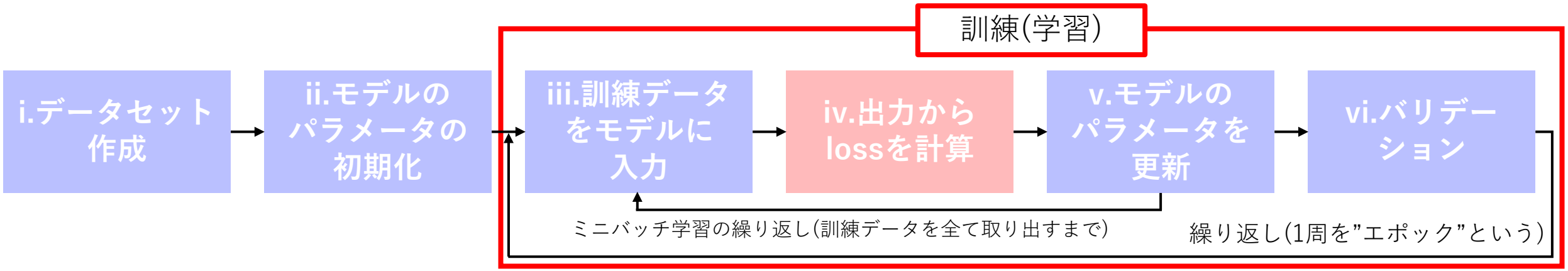
$$\boldsymbol{y} = F\left(W_k F\left(W_{k-1} F\left(W_{k-2} \cdots F(W_0 \boldsymbol{x}) \cdots\right)\right)\right)$$
$$= (f_k \circ f_{k-1} \circ f_{k-2} \circ \cdots \circ f_0)(\boldsymbol{x})$$



iv.出力からlossを計算

- **最終層ニューロンの活性化関数**
  - 分類問題でLossを計算するための準備
    - Softmax関数をよく使う。
      - 入力を分類の確信度合い([0,1]の実数値)に変換
      - n個(クラス数)の入力に対して合計が1になるように変換　する単調増加関数



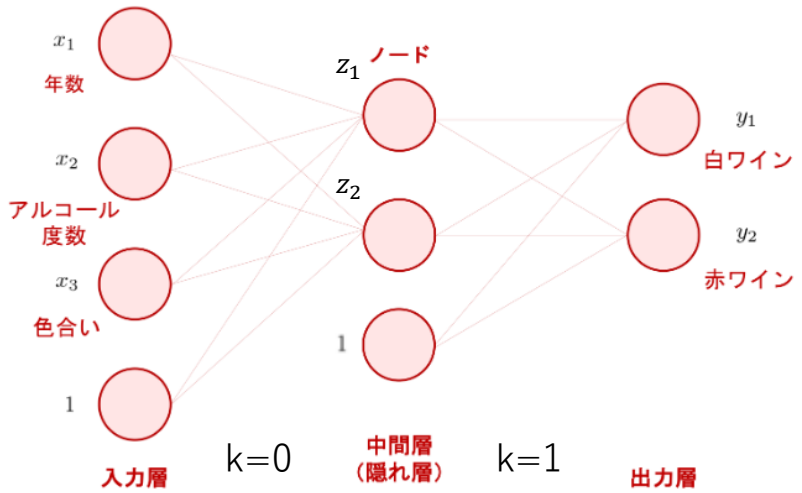


iv.出力からlossを計算

- **最終層ニューロンの活性化関数**
  - Softmax関数　入出力ともにクラス数ぶんの要素を持つベクトル　各出力値は[0,1]の実数、出力値の合計は1
    - i番目の入力に対するi番目の出力は

$$f_i(\boldsymbol{x}) = \frac{e^{x_i}}{\sum_{k=1}^{n} e^{x_k}}$$

    - 一般的に、この値が最も大きいクラスをNNの推論結果とする　$\begin{bmatrix}3\\4\\8\end{bmatrix}$を$\begin{bmatrix}0.1\\0.2\\0.7\end{bmatrix}$とかにする(計算は適当)　これならクラス3が推論結果



訓練(学習)

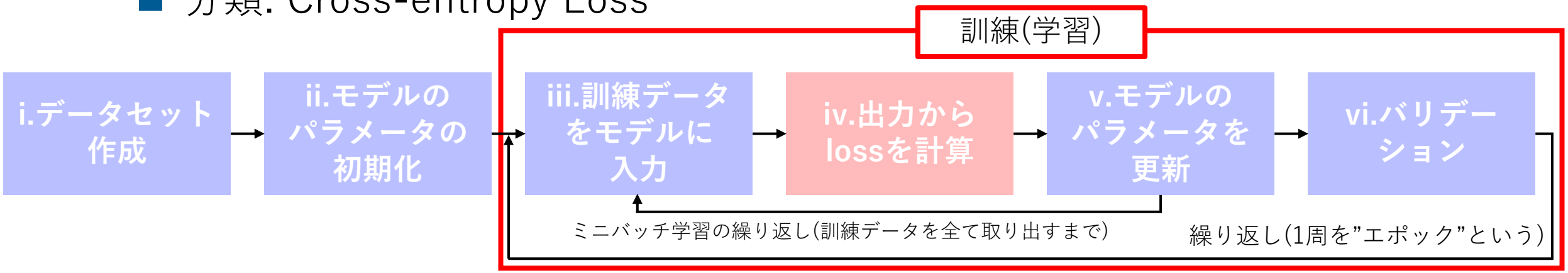
i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

ミニバッチ学習の繰り返し(訓練データを全て取り出すまで)

繰り返し(1周を”エポック”という)

iv.出力からlossを計算

- **損失関数(loss function)**
  - 現状のNNの予測結果と、実際の教師ラベルがどれだけ乖離しているかを表す関数。→小さいモデルほど優秀
    - 入力: NNの出力と教師データ
  - 回帰か分類かによって使うべきものが異なる。以下はよく使う。
    - 回帰: MSE Loss
    - 分類: Cross-entropy Loss

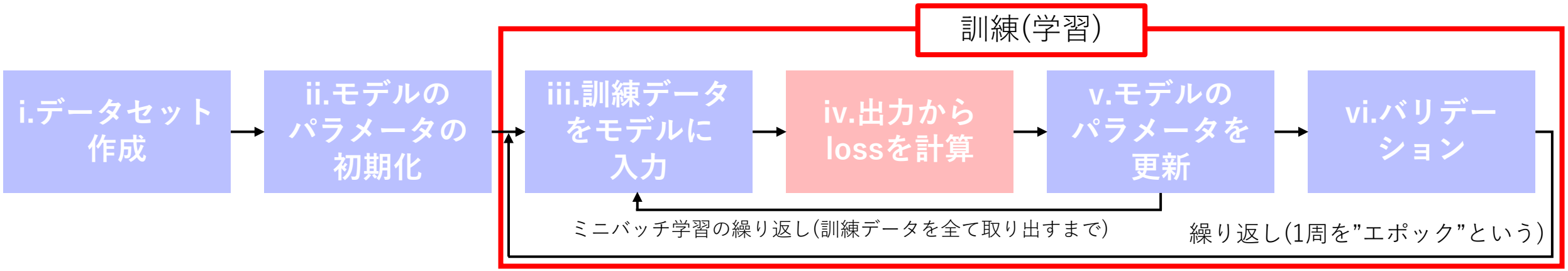


iv.出力からlossを計算

- **損失関数(loss function)**
  - Mean Squared Error (MSE) Loss…平均二乗誤差
    - NNの出力$\hat{y}$, 教師データyに対して、

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$$

    - ミニバッチ学習では、各出力$\hat{y}$, 教師データyに対する$\mathcal{L}$の平均をとる



iv.出力からlossを計算

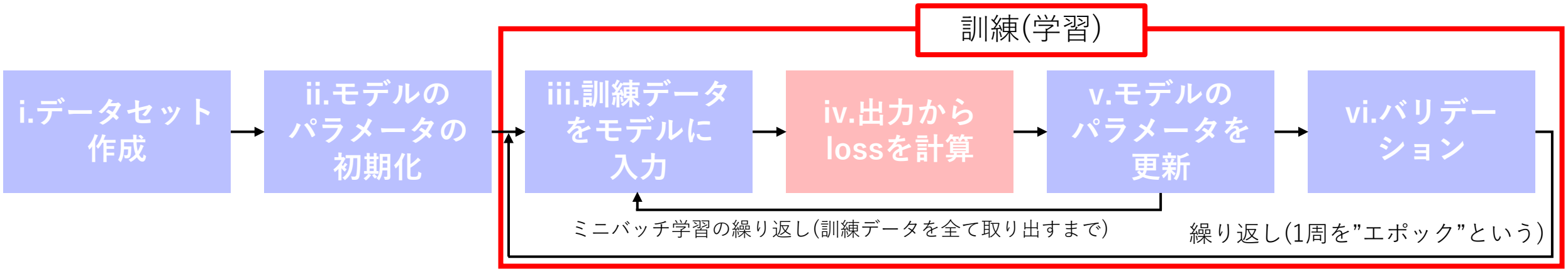
- **損失関数(loss function)**
  - Cross-entropy Loss…交差エントロピー誤差

↓今$\hat{\boldsymbol{y}}$はSoftmax関数を通って各値が[0,1]の実数で合計が1であるn次元のベクトル

- NNの出力$\hat{\boldsymbol{y}}$、教師データ$\boldsymbol{y}$、クラス数nの場合、$\boldsymbol{y}$が**one-hotコーディング**されている前提で、

$$\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{n} \boldsymbol{y}_i \ln(\hat{\boldsymbol{y}}_i)$$

- ミニバッチ学習では、各出力$\hat{\boldsymbol{y}}$, 教師データ$\boldsymbol{y}$に対する$\mathcal{L}$の平均をとる



iv.出力からlossを計算

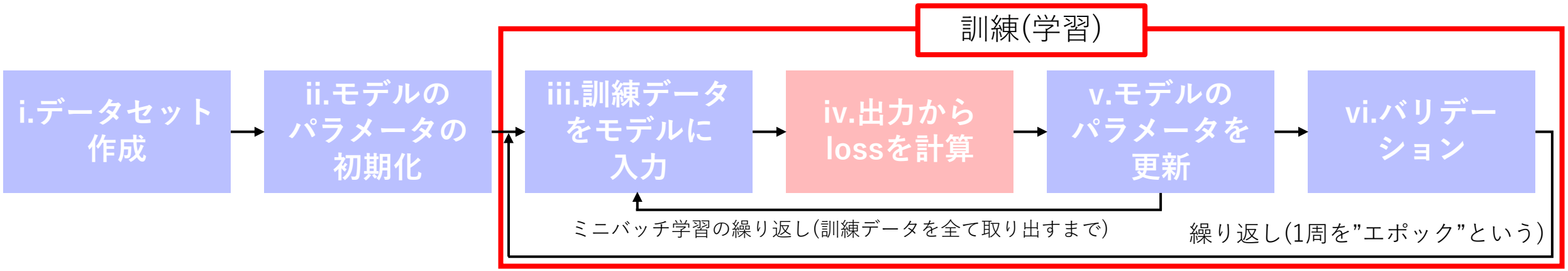
- **損失関数(loss function)**

（↓ $\hat{\boldsymbol{y}}$が教師、$\boldsymbol{y}$がNNの出力）

$$\mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}) = -\sum_{i=1}^{n} \boldsymbol{y}_i \ln(\hat{\boldsymbol{y}}_i)$$

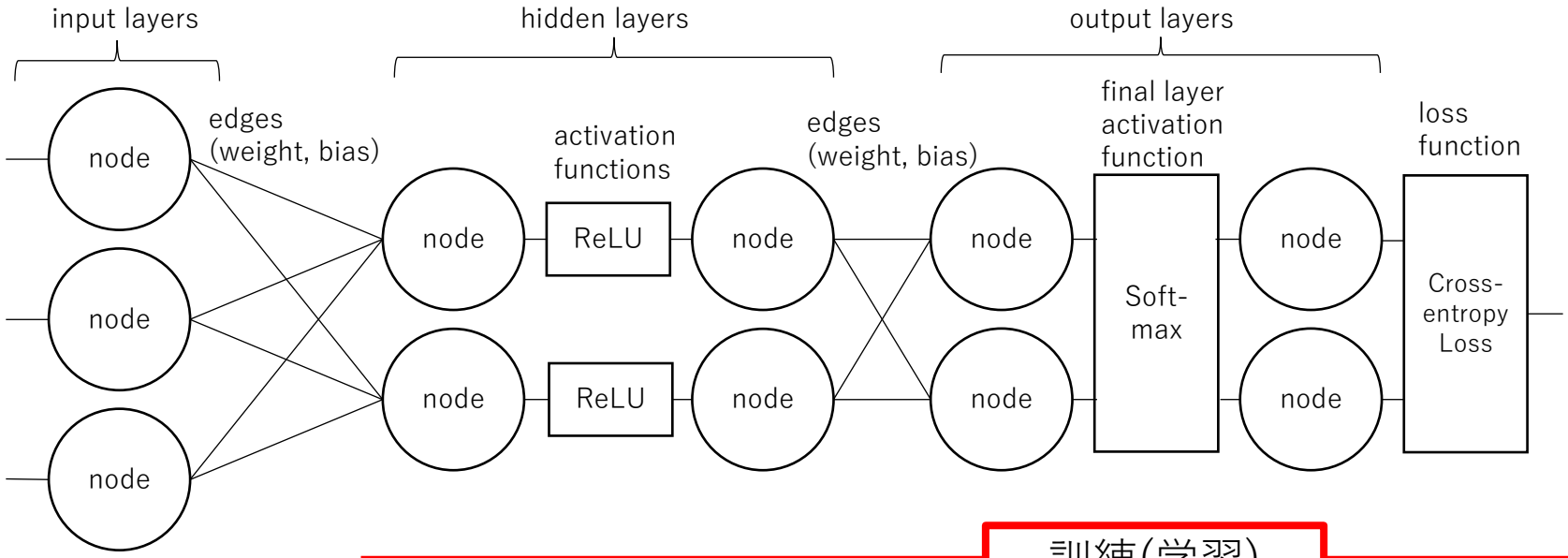
  - Cross-entropy Loss…交差エントロピー誤差
    - one-hotコーディング
      - 4クラスあって、クラス3が正解のとき、正解ラベル「3」を「$\begin{bmatrix}0\\0\\1\\0\end{bmatrix}$」のように「正解の位置だけ1、ほかは0」のベクトルにすること

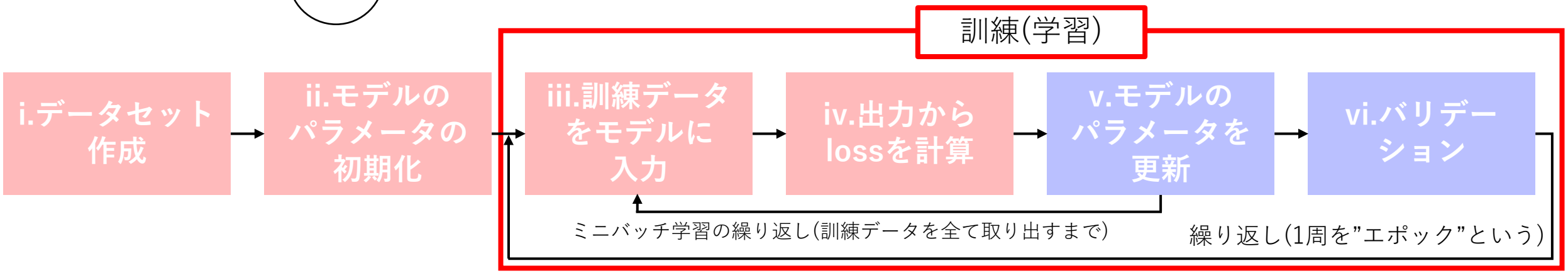
→こうしておくと正解ラベルがcのとき$\ln(\hat{\boldsymbol{y}}_c)$だけを足し合わせることになり、$\hat{\boldsymbol{y}}_c = 1$のとき(すなわち正解を100%の確信度で選んだとき)$\mathcal{L} = 0$(すなわち最高のモデル)となる



ここまでの復習

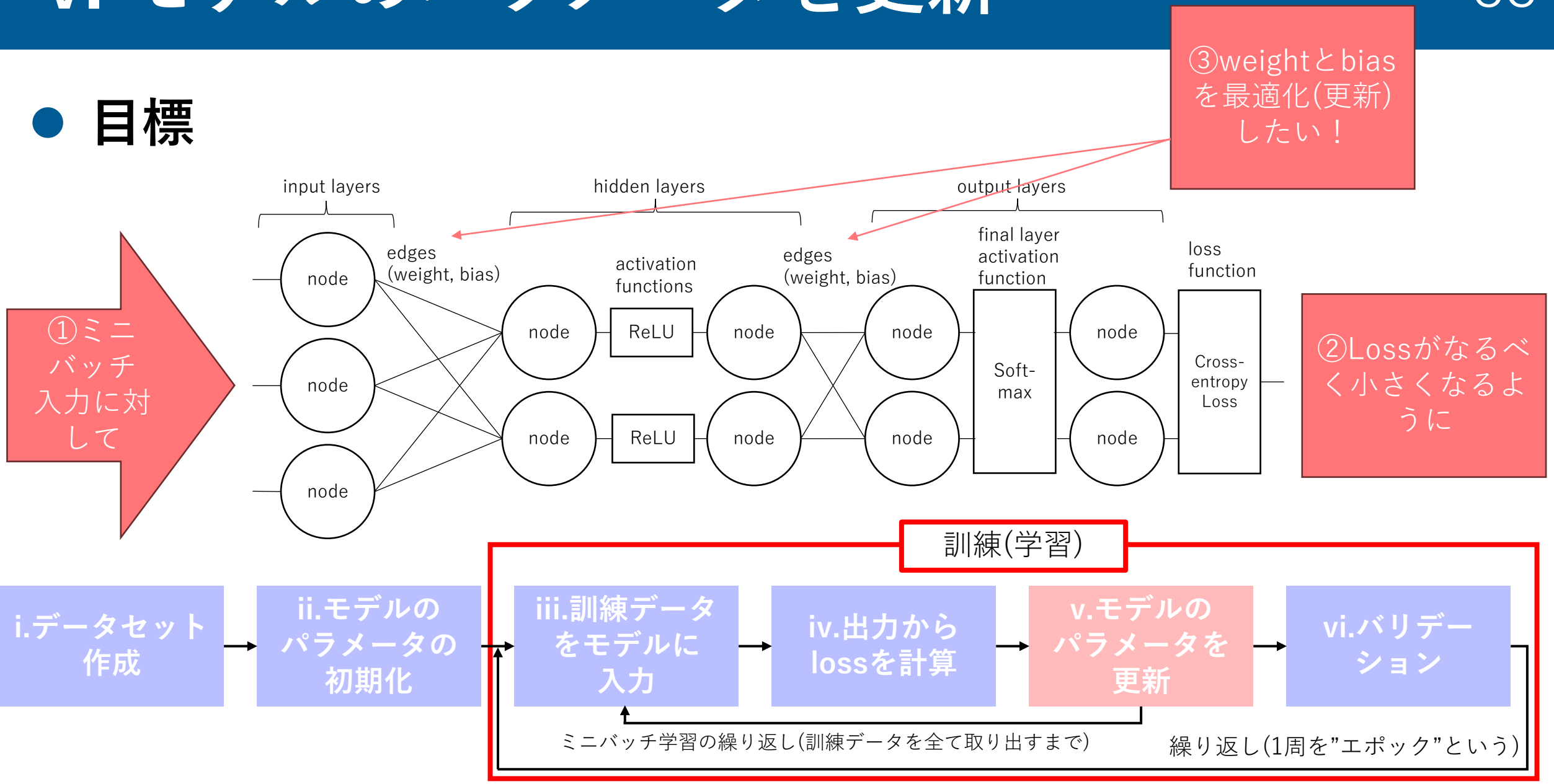
- **計算グラフが完成** (以下分類問題用MLPの例)





v. モデルのパラメータを更新

- 目標



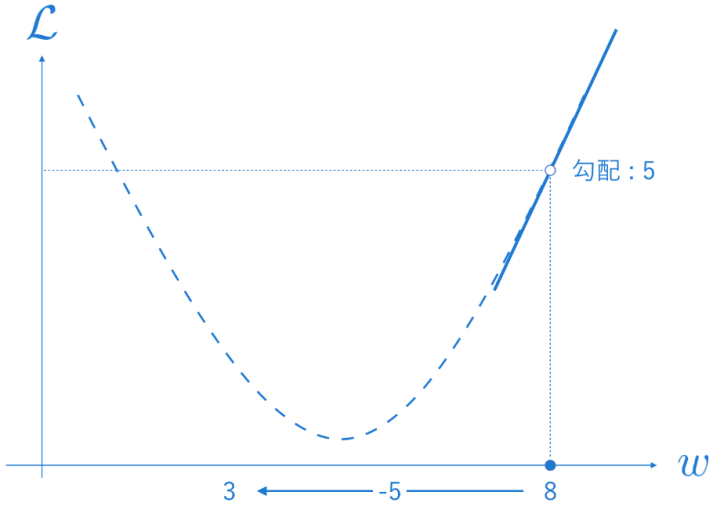
v. モデルのパラメータを更新

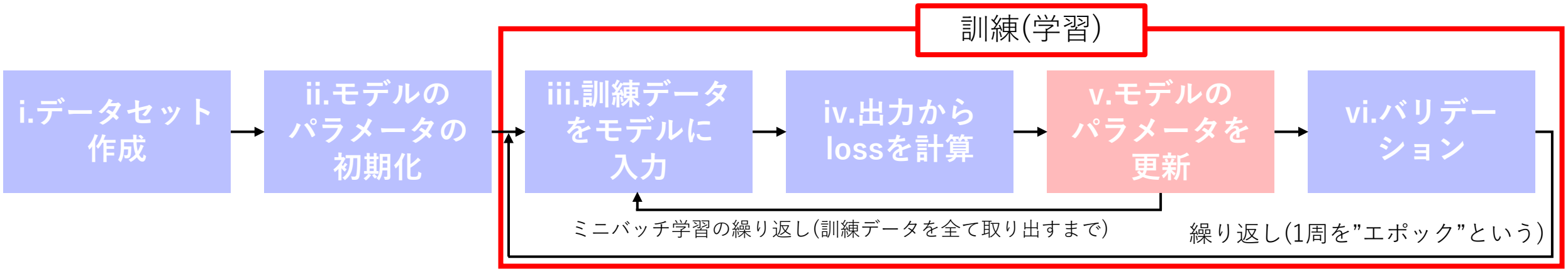
- **確率的勾配降下法 (Stochastic Gradient Descent: SGD)**
  - loss(= $\mathcal{L}$)に対する各パラメータ(= $\boldsymbol{w}$)の勾配(= $\frac{\partial \mathcal{L}}{\partial w}$)を用いて、

    ↓ γはハイパーパラメータ
    大きいと$\mathcal{L}$が収束しない
    小さいとローカルミニマムにはまる

    $\boldsymbol{w} \leftarrow \boldsymbol{w} - \boldsymbol{\gamma} \frac{\partial \mathcal{L}}{\partial w}$ と更新する($\boldsymbol{\gamma}$は学習率, 0-1)

    ↑これを繰り返してlossの最小値を探索

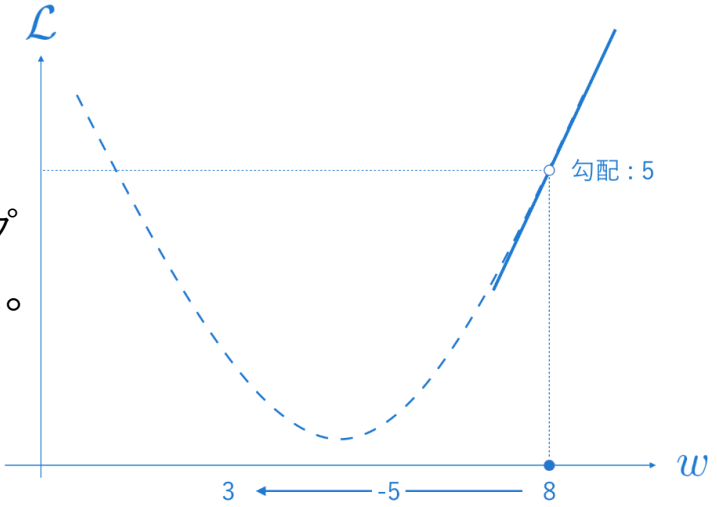


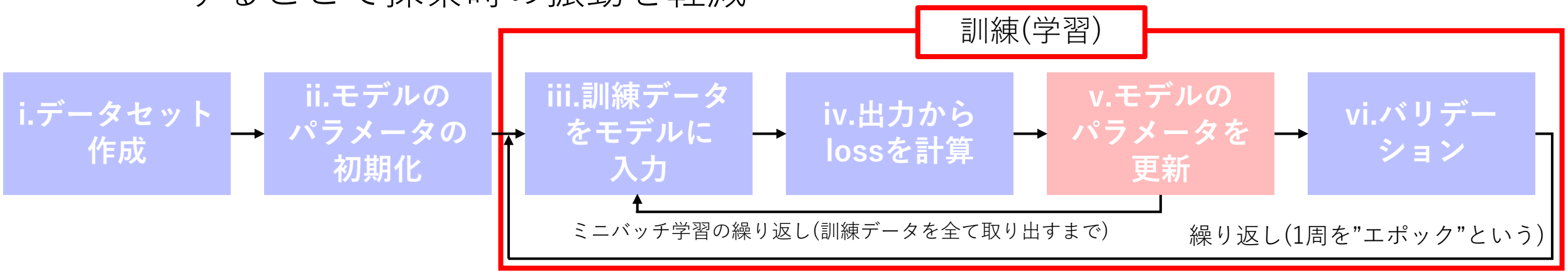


v. モデルのパラメータを更新

- **さまざまな最適化アルゴリズム(optimizer)**
  - SGD
    - 最も基本的なoptimizer
  - Adam
    - とりあえずこれ使っとけ
    - 移動平均を導入+勾配に応じて学習率を調整することで探索時の振動を軽減

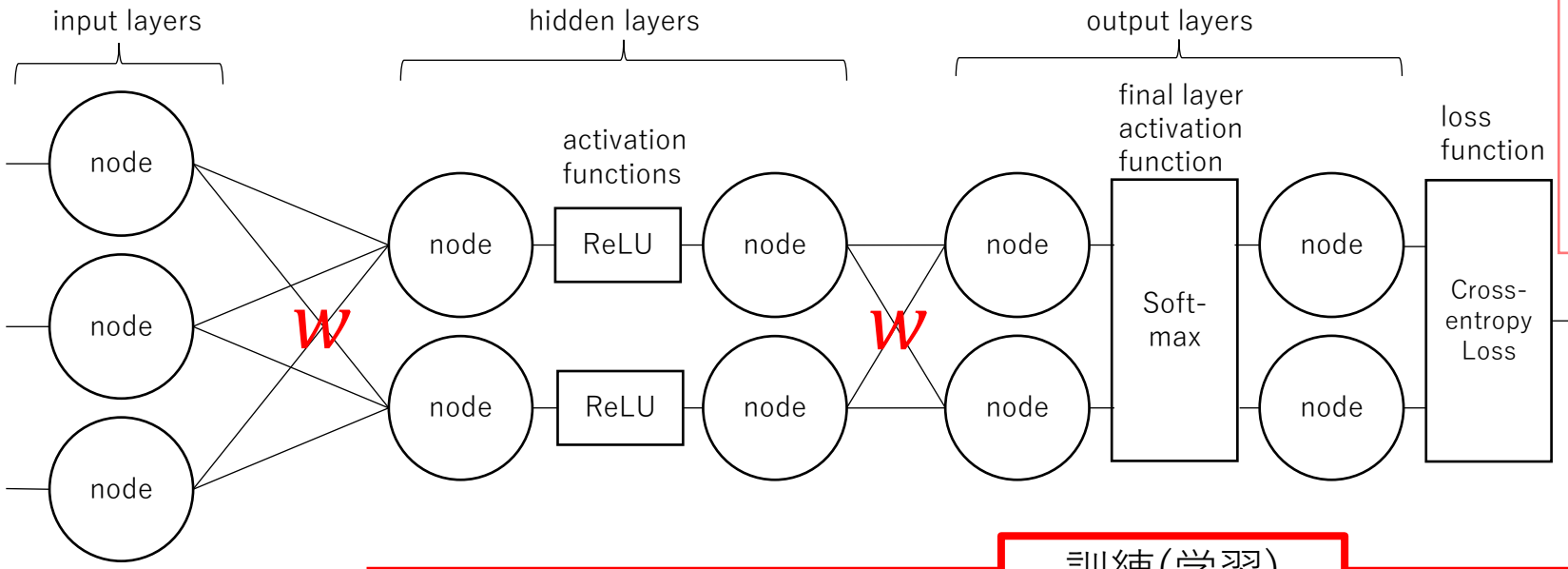
※詳細は「ゼロから作るDeep Learning―Pythonで学ぶディープラーニングの理論と実装」を参照。





v. モデルのパラメータを更新

- 勾配の計算: NNの全パラメータwに対して$\frac{\partial \mathcal{L}}{\partial w}$を計算…



NN全体の合成関数の導出
→変数が多すぎて無理

数値微分
→誤差が出る+2回流す必要あり

訓練(学習)

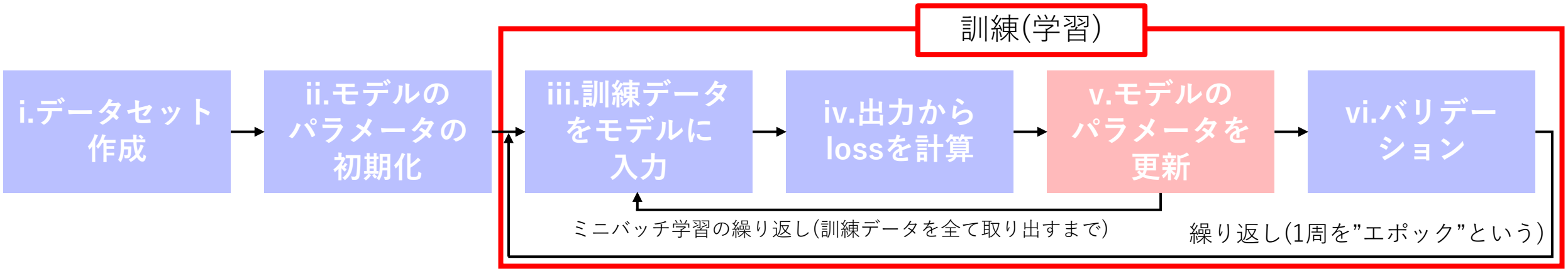
i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

ミニバッチ学習の繰り返し(訓練データを全て取り出すまで)

繰り返し(1周を”エポック”という)

v. モデルのパラメータを更新

- **誤差逆伝搬法**
  - ① 計算グラフ構築時、各エッジが表す関数を解析し微分しておく
  - ② 入力に対して損失関数$\mathcal{L}$の値を計算する
  - ③ 出力側から順に勾配を求め、入力側に向けて勾配を伝搬させる
    - 各ノードに記録した勾配値と微分の連鎖律を用いて計算効率UP
    - ノードによる勾配とパラメータによる勾配は異なるので混同に注意

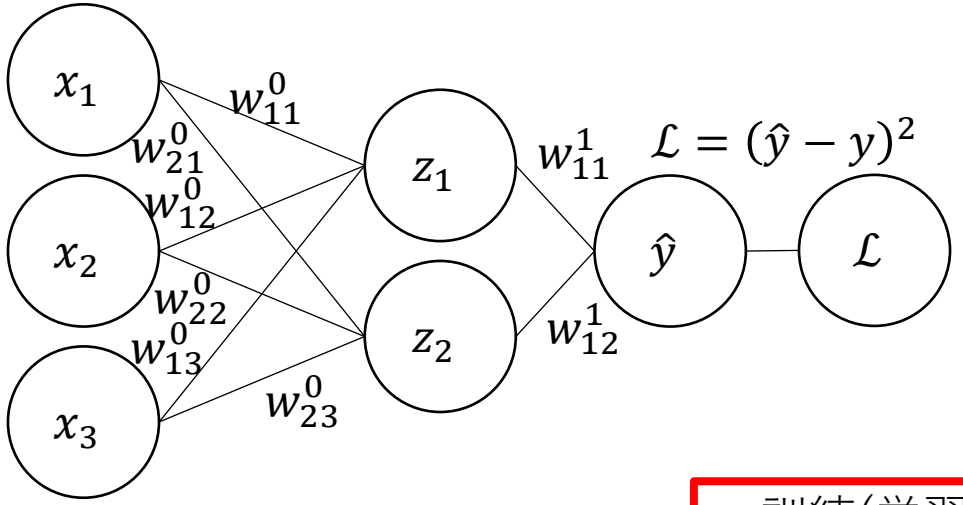


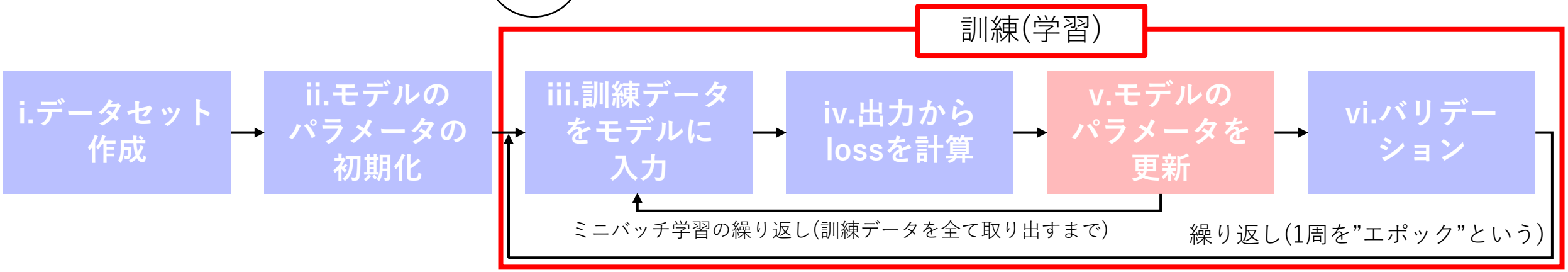
v. モデルのパラメータを更新

- **誤差逆伝搬法**
  - 例（バイアスと活性化関数は無視、正解ラベルy）

$w_{11}^0 = 3$
$w_{21}^0 = -2$
$w_{12}^0 = 1$
$w_{22}^0 = -3$
$w_{13}^0 = 2$
$w_{23}^0 = -1$

$w_{11}^1 = 3$
$w_{12}^1 = 2$
$y = 10$





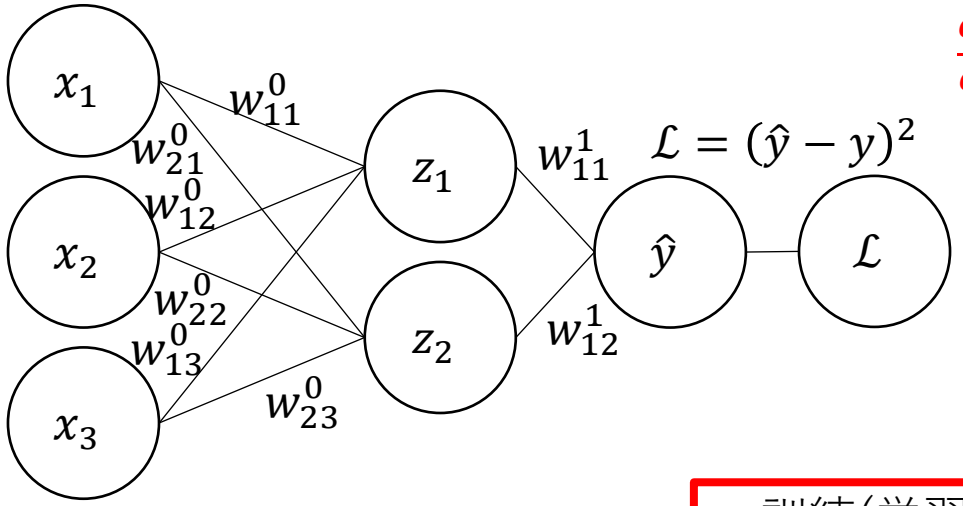
v. モデルのパラメータを更新

- **誤差逆伝搬法**

① 計算グラフ構築時、各エッジが表す関数を解析し微分しておく

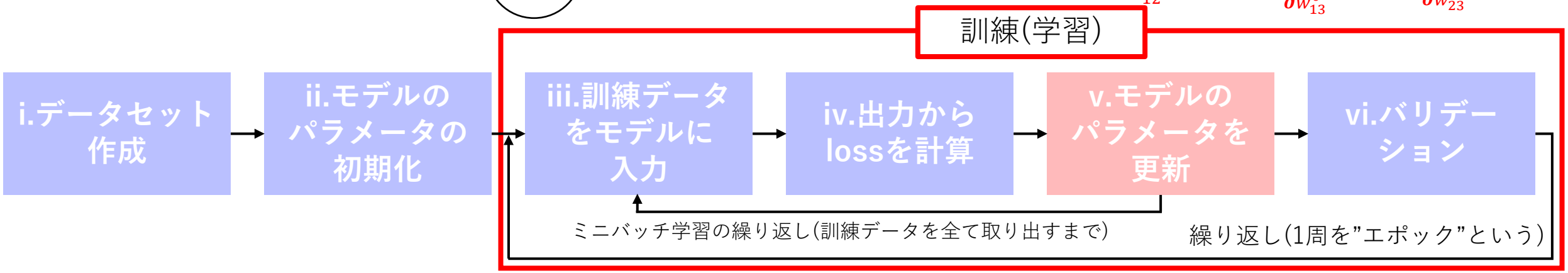
$$w_{11}^0 = 3$$
$$w_{21}^0 = -2$$
$$w_{12}^0 = 1$$
$$w_{22}^0 = -3$$
$$w_{13}^0 = 2$$
$$w_{23}^0 = -1$$

$$w_{11}^1 = 3$$
$$w_{12}^1 = 2$$
$$y = 10$$



$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - 10)$$
$$\frac{\partial \hat{y}}{\partial z_1} = w_{11}^1 = 3$$
$$\frac{\partial \hat{y}}{\partial w_{11}^1} = z_1$$
$$\frac{\partial \hat{y}}{\partial z_2} = w_{12}^1 = 2$$
$$\frac{\partial \hat{y}}{\partial w_{12}^1} = z_2$$

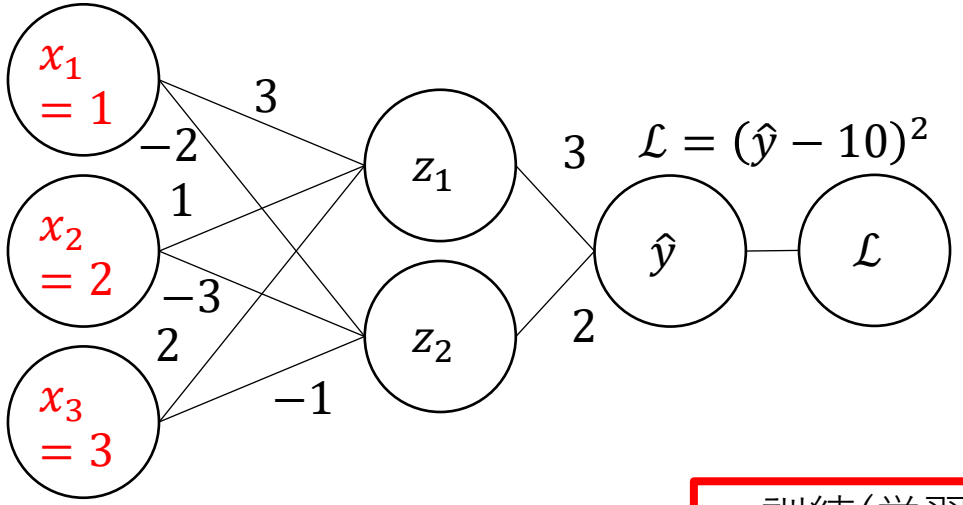
$$\frac{\partial z_1}{\partial x_1} = w_{11}^0 = 3 \quad \frac{\partial z_2}{\partial x_1} = w_{21}^0 = -2$$
$$\frac{\partial z_1}{\partial w_{11}^0} = x_1 \quad \frac{\partial z_2}{\partial w_{21}^0} = x_1$$
$$\frac{\partial z_1}{\partial x_2} = w_{12}^0 = 1 \quad \frac{\partial z_2}{\partial x_2} = w_{22}^0 = -3$$
$$\frac{\partial z_1}{\partial w_{12}^0} = x_2 \quad \frac{\partial z_2}{\partial w_{22}^0} = x_2$$
$$\frac{\partial z_1}{\partial x_3} = w_{13}^0 = 2 \quad \frac{\partial z_2}{\partial x_3} = w_{23}^0 = -1$$
$$\frac{\partial z_1}{\partial w_{13}^0} = x_3 \quad \frac{\partial z_2}{\partial w_{23}^0} = x_3$$

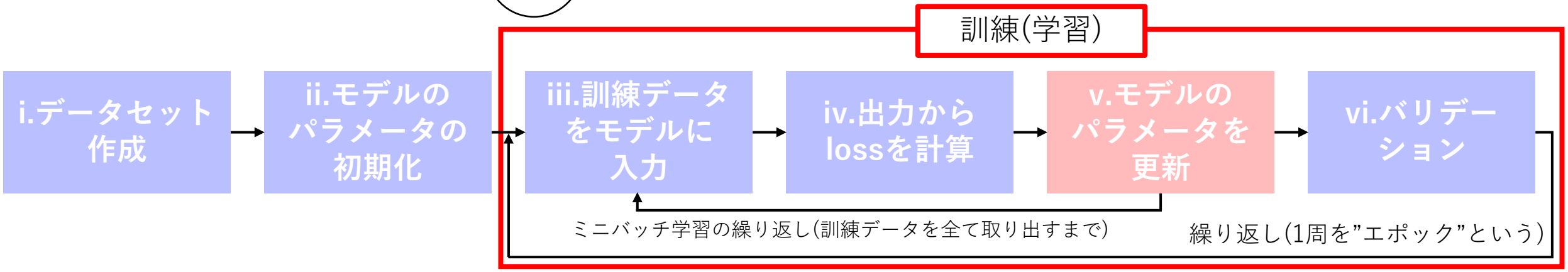


v. モデルのパラメータを更新

- **誤差逆伝搬法**

② 入力に対して損失関数$\mathcal{L}$の値を計算する（入力例）$(x_1, x_2, x_3) = (1,2,3)$

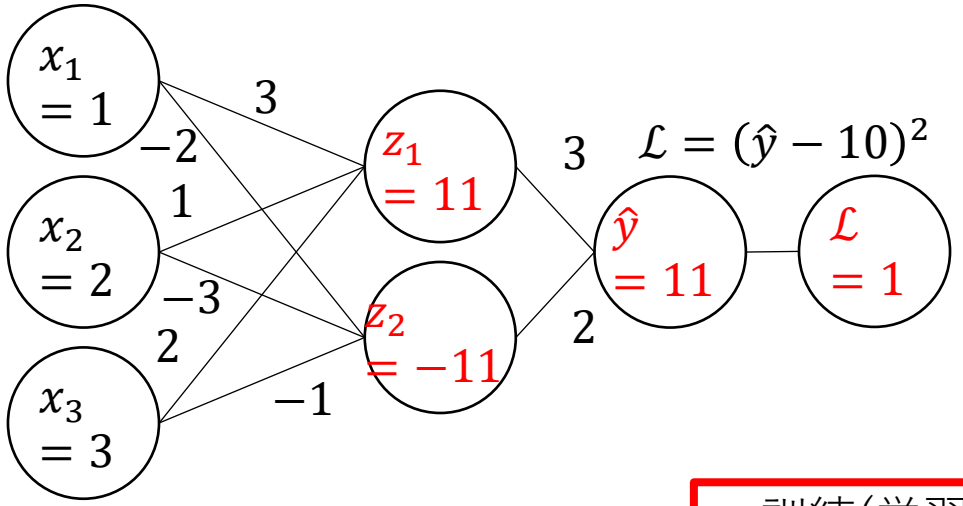


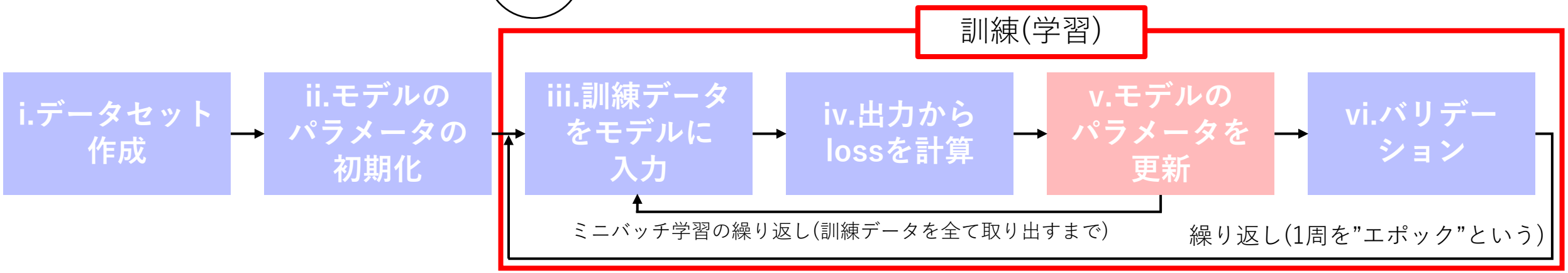


v. モデルのパラメータを更新

- **誤差逆伝搬法**

② 入力に対して損失関数$\mathcal{L}$の値を計算する（入力例）$(x_1, x_2, x_3) = (1,2,3)$



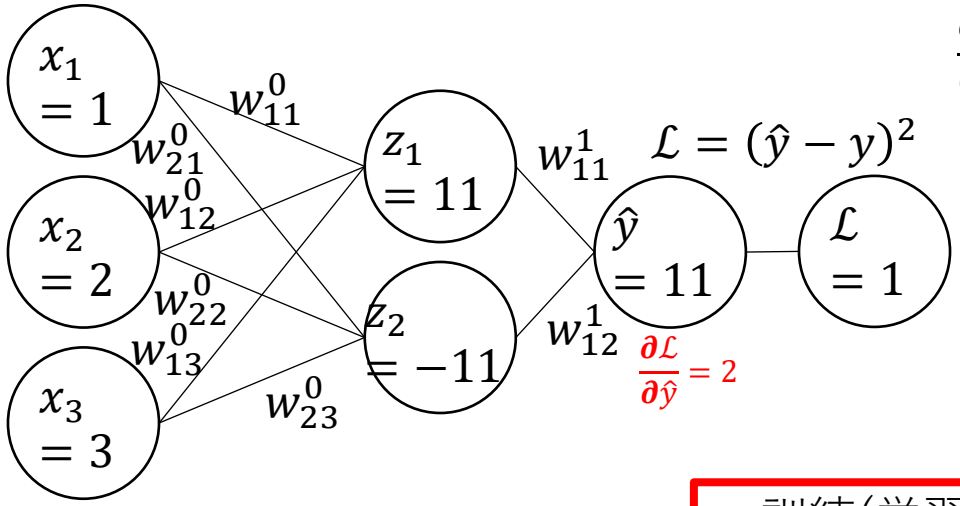


v. モデルのパラメータを更新

- **誤差逆伝搬法**

③ 出力側から順に勾配を求め、入力側に向けて勾配を伝搬させる

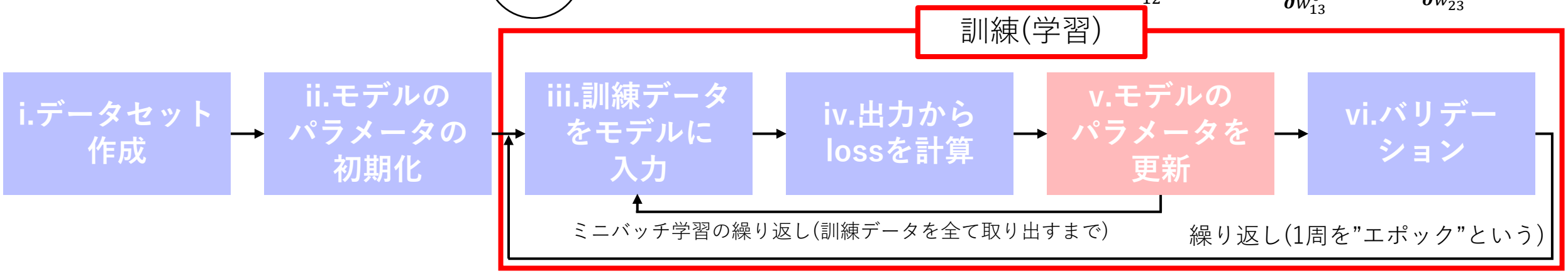
$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - 10) = 2$$



$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - 10) \quad \frac{\partial \hat{y}}{\partial z_1} = w_{11}^1 = 3 \quad \frac{\partial \hat{y}}{\partial w_{11}^1} = z_1 \quad \frac{\partial \hat{y}}{\partial z_2} = w_{12}^1 = 2 \quad \frac{\partial \hat{y}}{\partial w_{12}^1} = z_2$$

$$\frac{\partial z_1}{\partial x_1} = w_{11}^0 = 3 \quad \frac{\partial z_1}{\partial w_{11}^0} = x_1 \quad \frac{\partial z_1}{\partial x_2} = w_{12}^0 = 1 \quad \frac{\partial z_1}{\partial w_{12}^0} = x_2 \quad \frac{\partial z_1}{\partial x_3} = w_{13}^0 = 2 \quad \frac{\partial z_1}{\partial w_{13}^0} = x_3$$

$$\frac{\partial z_2}{\partial x_1} = w_{21}^0 = -2 \quad \frac{\partial z_2}{\partial w_{21}^0} = x_1 \quad \frac{\partial z_2}{\partial x_2} = w_{22}^0 = -3 \quad \frac{\partial z_2}{\partial w_{22}^0} = x_2 \quad \frac{\partial z_2}{\partial x_3} = w_{23}^0 = -1 \quad \frac{\partial z_2}{\partial w_{23}^0} = x_3$$



v. モデルのパラメータを更新

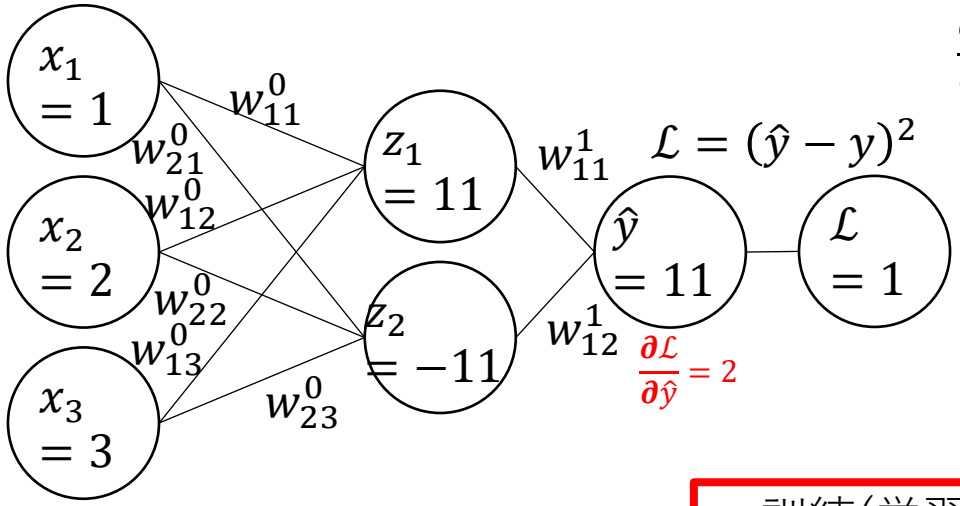
- **誤差逆伝搬法**

③ 出力側から順に勾配を求め、入力側に向けて勾配を伝搬させる

$$\frac{\partial \hat{y}}{\partial w_{11}^1} = z_1 = 11$$

微分の連鎖律より

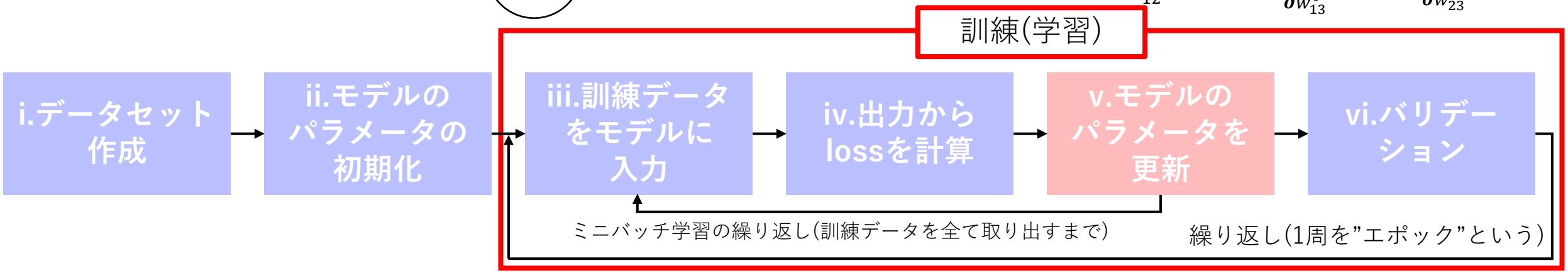
$$\frac{\partial \mathcal{L}}{\partial w_{11}^1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{11}^1} = 22$$



$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - 10) \quad \frac{\partial \hat{y}}{\partial z_1} = w_{11}^1 = 3 \quad \frac{\partial \hat{y}}{\partial w_{11}^1} = z_1 \quad \frac{\partial \hat{y}}{\partial z_2} = w_{12}^1 = 2 \quad \frac{\partial \hat{y}}{\partial w_{12}^1} = z_2$$

$$\frac{\partial z_1}{\partial x_1} = w_{11}^0 = 3 \quad \frac{\partial z_1}{\partial w_{11}^0} = x_1 \quad \frac{\partial z_1}{\partial x_2} = w_{12}^0 = 1 \quad \frac{\partial z_1}{\partial w_{12}^0} = x_2 \quad \frac{\partial z_1}{\partial x_3} = w_{13}^0 = 2 \quad \frac{\partial z_1}{\partial w_{13}^0} = x_3$$

$$\frac{\partial z_2}{\partial x_1} = w_{21}^0 = -2 \quad \frac{\partial z_2}{\partial w_{21}^0} = x_1 \quad \frac{\partial z_2}{\partial x_2} = w_{22}^0 = -3 \quad \frac{\partial z_2}{\partial w_{22}^0} = x_2 \quad \frac{\partial z_2}{\partial x_3} = w_{23}^0 = -1 \quad \frac{\partial z_2}{\partial w_{23}^0} = x_3$$



v. モデルのパラメータを更新

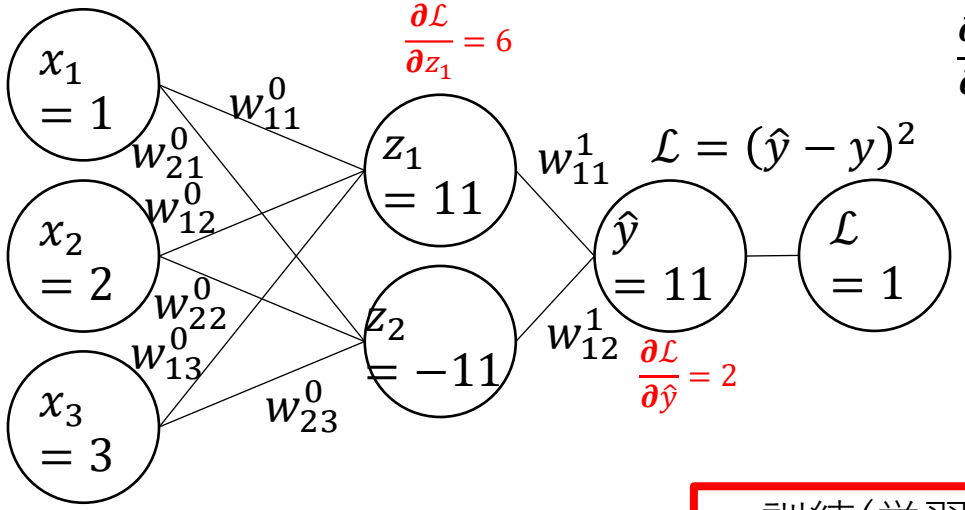
- **誤差逆伝搬法**

③ 出力側から順に勾配を求め、入力側に向けて勾配を伝搬させる

$$\frac{\partial \hat{y}}{\partial z_1} = 3$$

微分の連鎖律より

$$\frac{\partial \mathcal{L}}{\partial z_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1} = 6$$



$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - 10)$$

$$\frac{\partial \hat{y}}{\partial z_1} = w_{11}^1 = 3 \qquad \frac{\partial \hat{y}}{\partial w_{11}^1} = z_1$$

$$\frac{\partial \hat{y}}{\partial z_2} = w_{12}^1 = 2 \qquad \frac{\partial \hat{y}}{\partial w_{12}^1} = z_2$$

$$\frac{\partial z_1}{\partial x_1} = w_{11}^0 = 3 \quad \frac{\partial z_2}{\partial x_1} = w_{21}^0 = -2$$

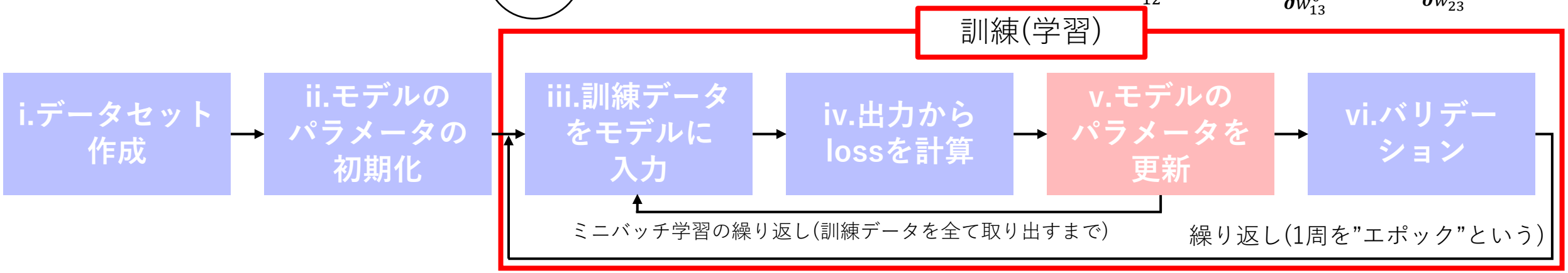
$$\frac{\partial z_1}{\partial w_{11}^0} = x_1 \quad \frac{\partial z_2}{\partial w_{21}^0} = x_1$$

$$\frac{\partial z_1}{\partial x_2} = w_{12}^0 = 1 \quad \frac{\partial z_2}{\partial x_2} = w_{22}^0 = -3$$

$$\frac{\partial z_1}{\partial w_{12}^0} = x_2 \quad \frac{\partial z_2}{\partial w_{22}^0} = x_2$$

$$\frac{\partial z_1}{\partial x_3} = w_{13}^0 = 2 \quad \frac{\partial z_2}{\partial x_3} = w_{23}^0 = -1$$

$$\frac{\partial z_1}{\partial w_{13}^0} = x_3 \quad \frac{\partial z_2}{\partial w_{23}^0} = x_3$$



v. モデルのパラメータを更新

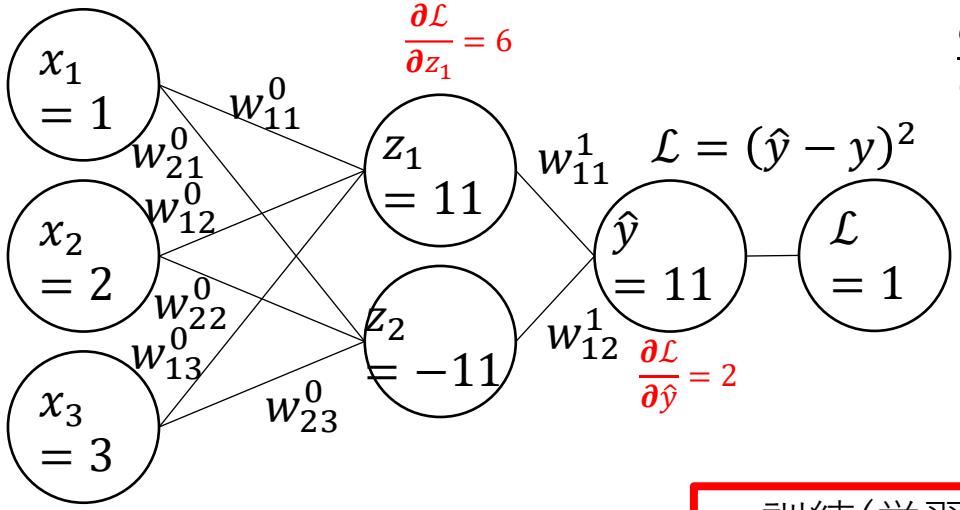
- **誤差逆伝搬法**

③ 出力側から順に勾配を求め、入力側に向けて勾配を伝搬させる

$$\frac{\partial \hat{y}}{\partial w_{12}^{1}} = z_2 = -11$$

微分の連鎖律より

$$\frac{\partial \mathcal{L}}{\partial w_{12}^{1}} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_{12}^{1}} = -22$$

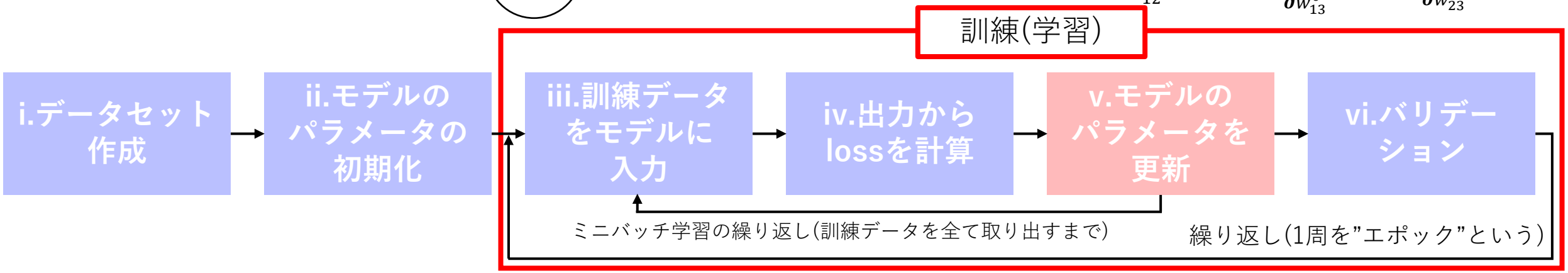


$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - 10) \quad \frac{\partial \hat{y}}{\partial z_1} = w_{11}^1 = 3 \quad \frac{\partial \hat{y}}{\partial w_{11}^1} = z_1 \quad \frac{\partial \hat{y}}{\partial z_2} = w_{12}^1 = 2 \quad \frac{\partial \hat{y}}{\partial w_{12}^1} = z_2$$

$$\frac{\partial z_1}{\partial x_1} = w_{11}^0 = 3 \quad \frac{\partial z_2}{\partial x_1} = w_{21}^0 = -2 \quad \frac{\partial z_1}{\partial w_{11}^0} = x_1 \quad \frac{\partial z_2}{\partial w_{21}^0} = x_1$$

$$\frac{\partial z_1}{\partial x_2} = w_{12}^0 = 1 \quad \frac{\partial z_2}{\partial x_2} = w_{22}^0 = -3 \quad \frac{\partial z_1}{\partial w_{12}^0} = x_2 \quad \frac{\partial z_2}{\partial w_{22}^0} = x_2$$

$$\frac{\partial z_1}{\partial x_3} = w_{13}^0 = 2 \quad \frac{\partial z_2}{\partial x_3} = w_{23}^0 = -1 \quad \frac{\partial z_1}{\partial w_{13}^0} = x_3 \quad \frac{\partial z_2}{\partial w_{23}^0} = x_3$$



v. モデルのパラメータを更新

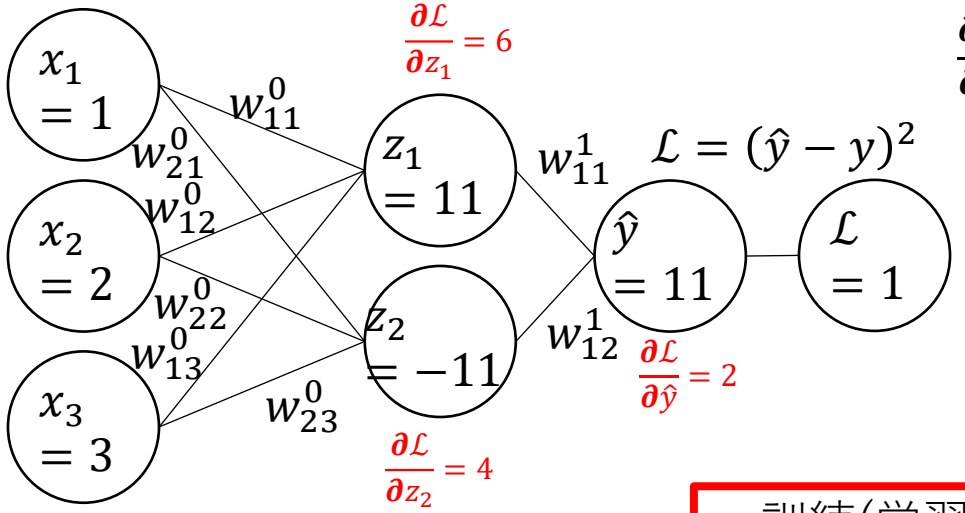
- **誤差逆伝搬法**

③ 出力側から順に勾配を求め、入力側に向けて勾配を伝搬させる

$$\frac{\partial \hat{y}}{\partial z_2} = 2$$

微分の連鎖律より

$$\frac{\partial \mathcal{L}}{\partial z_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_2} = 4$$



$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - 10)$$

$$\frac{\partial \hat{y}}{\partial z_1} = w_{11}^1 = 3 \qquad \frac{\partial \hat{y}}{\partial w_{11}^1} = z_1$$

$$\frac{\partial \hat{y}}{\partial z_2} = w_{12}^1 = 2 \qquad \frac{\partial \hat{y}}{\partial w_{12}^1} = z_2$$

$$\frac{\partial z_1}{\partial x_1} = w_{11}^0 = 3 \qquad \frac{\partial z_2}{\partial x_1} = w_{21}^0 = -2$$

$$\frac{\partial z_1}{\partial w_{11}^0} = x_1 \qquad \frac{\partial z_2}{\partial w_{21}^0} = x_1$$

$$\frac{\partial z_1}{\partial x_2} = w_{12}^0 = 1 \qquad \frac{\partial z_2}{\partial x_2} = w_{22}^0 = -3$$

$$\frac{\partial z_1}{\partial w_{12}^0} = x_2 \qquad \frac{\partial z_2}{\partial w_{22}^0} = x_2$$

$$\frac{\partial z_1}{\partial x_3} = w_{13}^0 = 2 \qquad \frac{\partial z_2}{\partial x_3} = w_{23}^0 = -1$$

$$\frac{\partial z_1}{\partial w_{13}^0} = x_3 \qquad \frac{\partial z_2}{\partial w_{23}^0} = x_3$$

訓練(学習)

i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

ミニバッチ学習の繰り返し(訓練データを全て取り出すまで)

繰り返し(1周を”エポック”という)

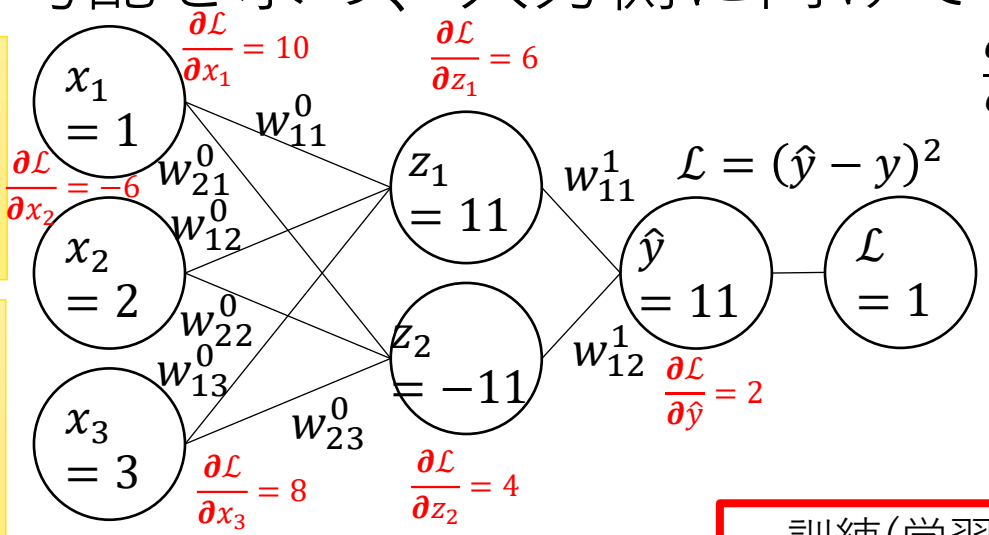
v. モデルのパラメータを更新

- **誤差逆伝搬法**

③ 出力側から順に勾配を求め、入力側に向けて勾配を伝搬させる

$$\frac{\partial \mathcal{L}}{\partial w_{11}^0} = \frac{\partial \mathcal{L}}{\partial z_1}\frac{\partial z_1}{\partial w_{11}^0} = 6,\ \frac{\partial \mathcal{L}}{\partial w_{21}^0} = \frac{\partial \mathcal{L}}{\partial z_2}\frac{\partial z_2}{\partial w_{21}^0} = 4$$
$$\frac{\partial \mathcal{L}}{\partial w_{12}^0} = \frac{\partial \mathcal{L}}{\partial z_1}\frac{\partial z_1}{\partial w_{12}^0} = 12,\ \frac{\partial \mathcal{L}}{\partial w_{22}^0} = \frac{\partial \mathcal{L}}{\partial z_2}\frac{\partial z_2}{\partial w_{22}^0} = 8$$
$$\frac{\partial \mathcal{L}}{\partial w_{13}^0} = \frac{\partial \mathcal{L}}{\partial z_1}\frac{\partial z_1}{\partial w_{13}^0} = 18,\ \frac{\partial \mathcal{L}}{\partial w_{23}^0} = \frac{\partial \mathcal{L}}{\partial z_2}\frac{\partial z_2}{\partial w_{23}^0} = 12$$

$$\frac{\partial \mathcal{L}}{\partial x_1} = \frac{\partial \mathcal{L}}{\partial z_1}\frac{\partial z_1}{\partial x_1} + \frac{\partial \mathcal{L}}{\partial z_2}\frac{\partial z_2}{\partial x_1} = 18 + (-8) = 10$$
$$\frac{\partial \mathcal{L}}{\partial x_2} = \frac{\partial \mathcal{L}}{\partial z_1}\frac{\partial z_1}{\partial x_2} + \frac{\partial \mathcal{L}}{\partial z_2}\frac{\partial z_2}{\partial x_2} = 6 + (-12) = -6$$
$$\frac{\partial \mathcal{L}}{\partial x_3} = \frac{\partial \mathcal{L}}{\partial z_1}\frac{\partial z_1}{\partial x_3} + \frac{\partial \mathcal{L}}{\partial z_2}\frac{\partial z_2}{\partial x_3} = 12 + (-4) = 8$$



$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = 2(\hat{y} - 10)$$
$$\frac{\partial \hat{y}}{\partial z_1} = w_{11}^1 = 3 \qquad \frac{\partial \hat{y}}{\partial w_{11}^1} = z_1$$
$$\frac{\partial \hat{y}}{\partial z_2} = w_{12}^1 = 2 \qquad \frac{\partial \hat{y}}{\partial w_{12}^1} = z_2$$

$$\frac{\partial z_1}{\partial x_1} = w_{11}^0 = 3 \quad \frac{\partial z_2}{\partial x_1} = w_{21}^0 = -2$$
$$\frac{\partial z_1}{\partial w_{11}^0} = x_1 \quad \frac{\partial z_2}{\partial w_{21}^0} = x_1$$
$$\frac{\partial z_1}{\partial x_2} = w_{12}^0 = 1 \quad \frac{\partial z_2}{\partial x_2} = w_{22}^0 = -3$$
$$\frac{\partial z_1}{\partial w_{12}^0} = x_2 \quad \frac{\partial z_2}{\partial w_{22}^0} = x_2$$
$$\frac{\partial z_1}{\partial x_3} = w_{13}^0 = 2 \quad \frac{\partial z_2}{\partial x_3} = w_{23}^0 = -1$$
$$\frac{\partial z_1}{\partial w_{13}^0} = x_3 \quad \frac{\partial z_2}{\partial w_{23}^0} = x_3$$

訓練(学習)

i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

ミニバッチ学習の繰り返し(訓練データを全て取り出すまで)

繰り返し(1周を”エポック”という)

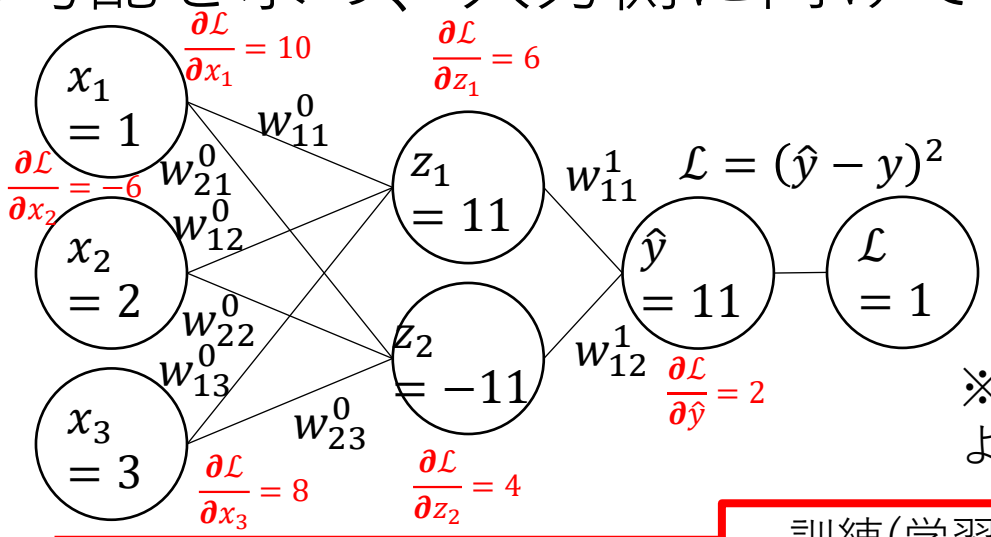
v. モデルのパラメータを更新

- **誤差逆伝搬法**

③ 出力側から順に勾配を求め、入力側に向けて勾配を伝搬させる

結局、

$$\frac{\partial \mathcal{L}}{\partial w_{11}^0} = 6, \frac{\partial \mathcal{L}}{\partial w_{21}^0} = 4$$
$$\frac{\partial \mathcal{L}}{\partial w_{12}^0} = 12, \frac{\partial \mathcal{L}}{\partial w_{22}^0} = 8$$
$$\frac{\partial \mathcal{L}}{\partial w_{13}^0} = 18, \frac{\partial \mathcal{L}}{\partial w_{23}^0} = 12$$
$$\frac{\partial \mathcal{L}}{\partial w_{11}^1} = 22, \frac{\partial \mathcal{L}}{\partial w_{12}^1} = -22$$



※ノードによる勾配とパラメータによる勾配は異なるので混同に注意

訓練(学習)

i.データセット作成 → ii.モデルのパラメータの初期化 → iii.訓練データをモデルに入力 → iv.出力からlossを計算 → v.モデルのパラメータを更新 → vi.バリデーション

ミニバッチ学習の繰り返し(訓練データを全て取り出すまで)

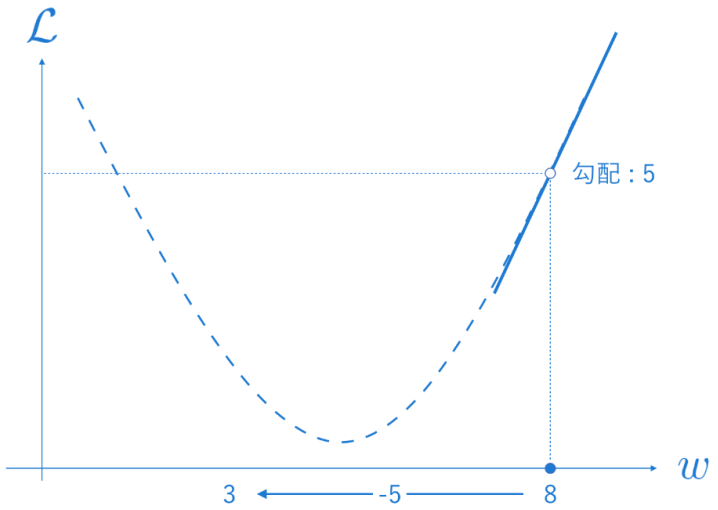
繰り返し(1周を”エポック”という)

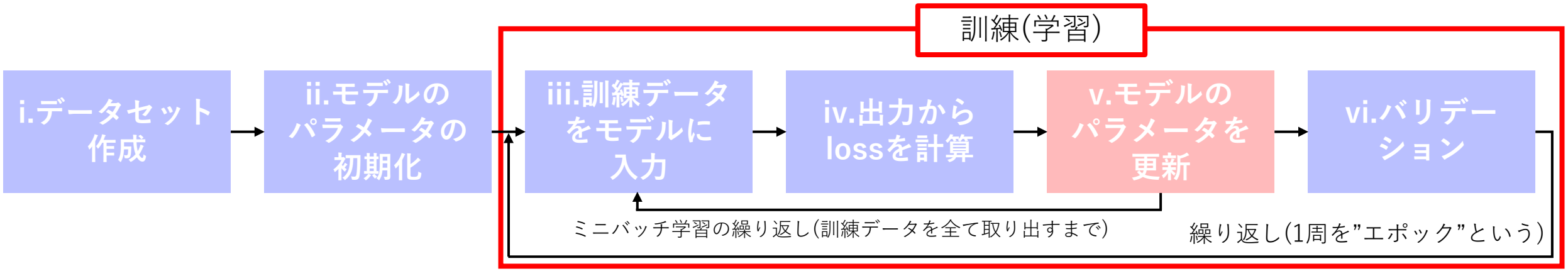
v. モデルのパラメータを更新

- **確率的勾配降下法（再掲）**
  - 誤差逆伝播法で全パラメータwにおける$\frac{\partial \mathcal{L}}{\partial w}$が求まったら、それぞれ

$$w \leftarrow w - \gamma \frac{\partial \mathcal{L}}{\partial w}$$

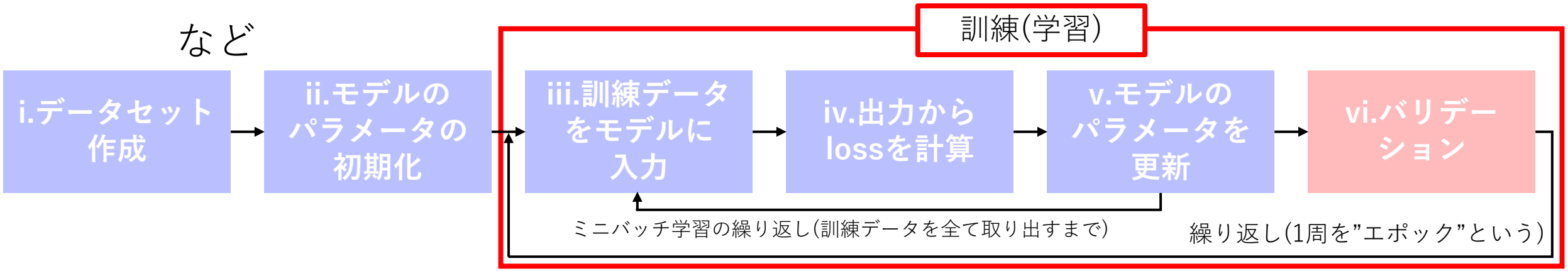
と更新する（γは学習率で0-1の実数）。





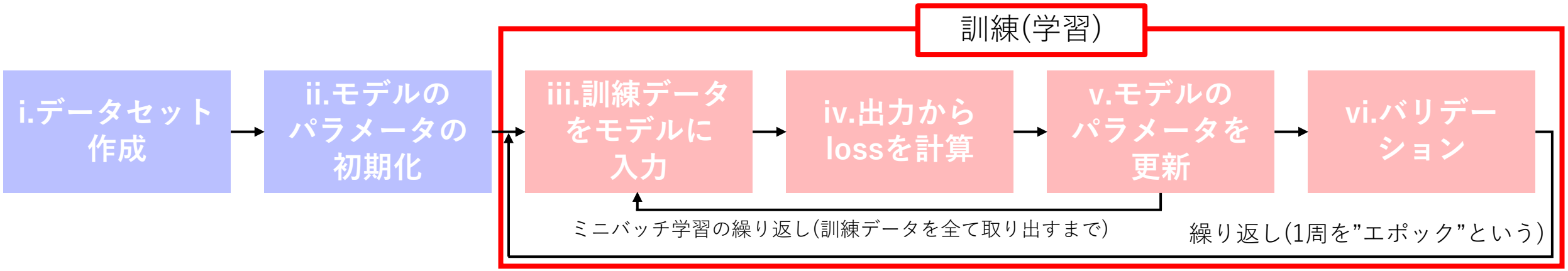
vi. バリデーション

- 一般的に各エポックの最後に行う(訓練データを全て使ったあと)
- 検証(validation)データ (再掲)
  - テストに用いるモデルを選んだり、ハイパーパラメータを更新したりするためのデータ。模試。
- 主に行うこと
  - 検証データによるモデルの評価・記録(Loss、正答率など)
  - 学習率の調整 (例) 5エポックLossが下がらなかったら学習率を半分にするなど



機械学習のフェーズ

- **訓練(学習)フェーズ**
  - iii~viを決められたエポック数だけ繰り返すことが最も一般的
    - バリデーション結果により途中で打ち切る(earlystop)こともある
- **テスト(推論)フェーズ**
  - 訓練(学習)が終わったら、テストデータを使ってモデルを評価する
    - バリデーション結果によりどのエポック時点のモデルを使うか選ぶ



諸々の問題

- **過学習(overfitting)**
  - モデルが訓練データに過剰にフィッティングし、未知のテストデータに対して性能が悪い(=汎化性能がない)状態
    - 主な原因
      - データの分散が大きすぎる
      - NNのパラメータが多すぎる
    - 主な解決策
      - データ数を増やす(データ拡張など)
      - drop-outやbatch normalization(正則化)を使用する
      - 層数を少なくする

※詳細は「ゼロから作るDeep Learning―Pythonで学ぶディープラーニングの理論と実装」を参照。

諸々の問題

- **勾配消失(gradient vanishment)**
  - あるノードで勾配が0に近くなると、その前の全てのノードの勾配も0に近くなり、学習がうまく進まなくなってしまう状態
    - 主な原因
      - NNの層数が多すぎる
      - 勾配消失が起こりやすい活性化関数を使っている
    - 主な解決策
      - 層数を少なくする
      - 勾配消失が起こりにくい活性化関数を使う

SGDの更新式: $\boldsymbol{w \leftarrow w - \gamma \frac{\partial \mathcal{L}}{\partial w}}$

※詳細は「ゼロから作るDeep Learning―Pythonで学ぶディープラーニングの理論と実装」を参照。

理論編 おわりに

- **DNNの学習の一連の流れ、やってきました**
  - 一回で理解するのはなかなか難しいので、さっさと実装編に入って、問題に当たったときに理論の未定着が原因なら、そこでもう一回勉強しましょう
- **理論が複雑だから、当然その実装もかなり複雑…**
  - ゆっくり理解していきましょう

Let’s go to the next chapter!!

2. 実装編

ディープラーニングフレームワーク

- **ニューラルネットワークの設計・学習・評価などに必要な一連の実装を容易にするためのPython上のフレームワーク**
  - PyTorch　←まずこれを習得しよう！
    - 主に研究分野で使われている。論文に載っているソースコードはPyTorchで書かれていることが多い。
  - TensorFlow
    - 主に企業が使っている。PyTorchより多くインターネット上に記事が載っている。Kerasと混ざって紛らわしい。
  - Keras
    - TensorFlowで使える便利な機能を提供する。便利すぎて、もはや単独のライブラリになってしまった。

PyTorch

- **「Tensor」という特殊なclassを操作する**
  - よく使うライブラリやメソッドの仕様を理解をしておくことが重要
  - numpy.ndarrayと似ている
    - 決定的な違い
      - GPUで計算するための機能が提供されている
        - ✓ GPU(Graphic Processing Unit): 行列計算をCPUより高速に行える
      - 計算グラフが自動で構築される
      - gradという属性に勾配情報を保存できる

準備

1. Python3.10とVSCodeをインストール
2. 好きなディレクトリでVSCodeを起動
3. 仮想環境を作る「py -3.10 –m venv {venv_name}」
4. 仮想環境を起動「./{venv_name}/Scripts/activate」
5. NKTLABのGithubからサンプルコードをダウンロードしてカレントディレクトリに配置 https://github.com/YNU-NakataLab/cnn_manual.git
6. 必要なライブラリをインポート
   「pip install -r ./torch/requirements.txt 」

※手順4 でエラーが出る場合は、Powershellを管理者として実行して「PowerShell Set-ExecutionPolicy RemoteSigned」を実行し、VSCodeを再起動して再実施

class torch.Tensor

- **行列形式のデータを扱うクラス**
  - インスタンス化方法
    1. import torch
    2. x = torch.tensor([[1, 2], [3, 4]])　←torch.tensor({数値, リスト, np.ndarrayなど})
  - 属性(クラス変数)の例
    - shape: tensorのサイズ
    - ndim: tensorの次元
    - dtype: tensorの要素の型
  - メソッドの例
    - numel: tensorの要素数
    - numpy: numpyへの変換
    - expand: リスト形式の引数をとって次元を拡張

Let’s run “./torch/practice_tensor_1.py”!!

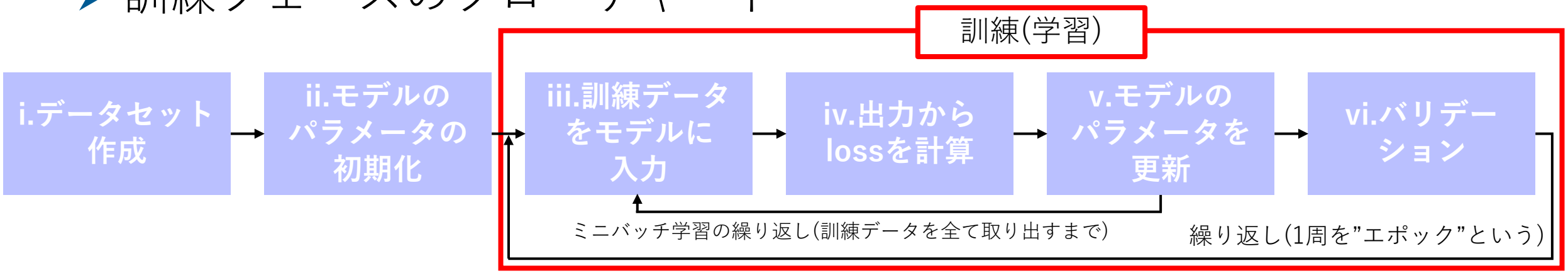
class torch.Tensor

- **勾配の計算**
  - インスタンス化時の引数に”require_grad=True”を追加
    - 勾配計算時はdtypeがfloat型でなければならないので注意
  - tensorに対して加減乗除などさまざまな演算を施し新たなtensorを作成すると、自動で計算グラフが構築され、新しいtensorのgrad_fn属性にその情報が保持される
    - 最初に自分で作ったtensorだけはgrad_fn属性がNoneのままになる
  - backwardメソッド
    - そのtensorの計算過程で使われた全てのtensorによる勾配値が、誤差逆伝搬法によって計算され、それぞれのtensorのgrad属性に記録される
      - 計算グラフの途中の勾配を取得するには事前にretain_gradメソッドを実行していることが必要
      - backwardメソッドを実行するtensorは要素数1(スカラー)のtensorでなければならない

Let’s run “./torch/practice_tensor_2.py”!!

NNを実装してみよう

- **全結合NN**
  - カリフォルニアの住宅価格を回帰する問題
- **CNN**
  - CIFAR-10データセットの分類問題
- **（復習）訓練フェーズと推論フェーズ**
  - 訓練フェーズのフローチャート



実験の準備

- **乱数シードの固定**　**Let’s go over “./torch/nn/set_seed.py”!!**
  - プログラム中でランダムに決まる要素(NNのパラメータの初期値など)が異なると、実験の再現性がなくなる

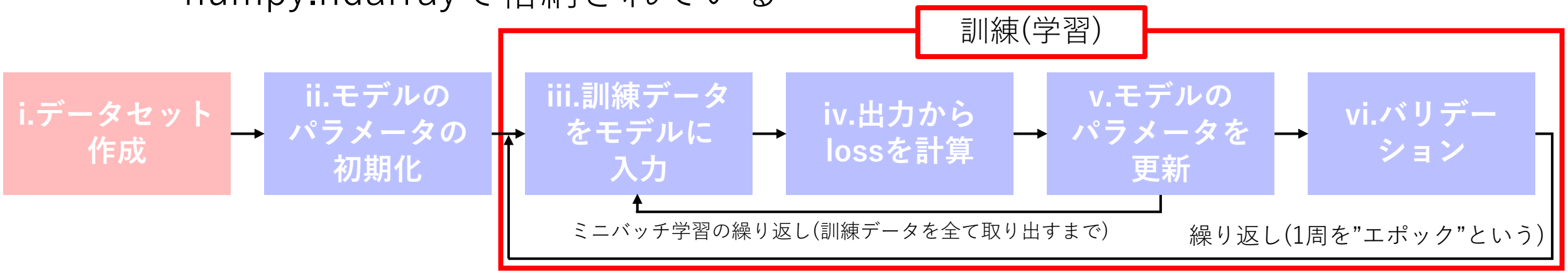
  → os, numpy, torch, random(Python標準ライブラリ)のシードを固定

- **ログ出力先の設定**　**Let’s go over “./torch/nn/logging_controller.py”!!**
  - 一回の実験に時間がかかることが多く、何らかの要因でターミナルが落ちてデータが失われたりすると死ぬ

    → ログをファイルに出力して、後で見返せるようにする

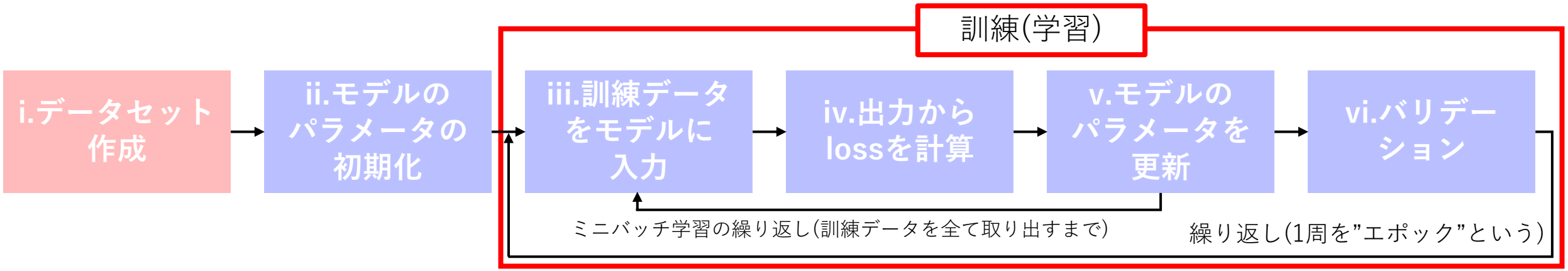
i. データセット作成

- **California Housing** Let's run "./torch/check_california_housing.py"!!
  - 住宅に関する8種類の説明変数(築年数、地域の人口など)からその住宅の価格を回帰するタスク
    → scikit-learnライブラリに用意されている
  - pandas.DataFlameとして読み込みどんなデータか見てみよう
    - 元々はdata属性に説明変数(8種)、target属性に目的変数(住宅価格)がnumpy.ndarrayで格納されている



i. データセット作成

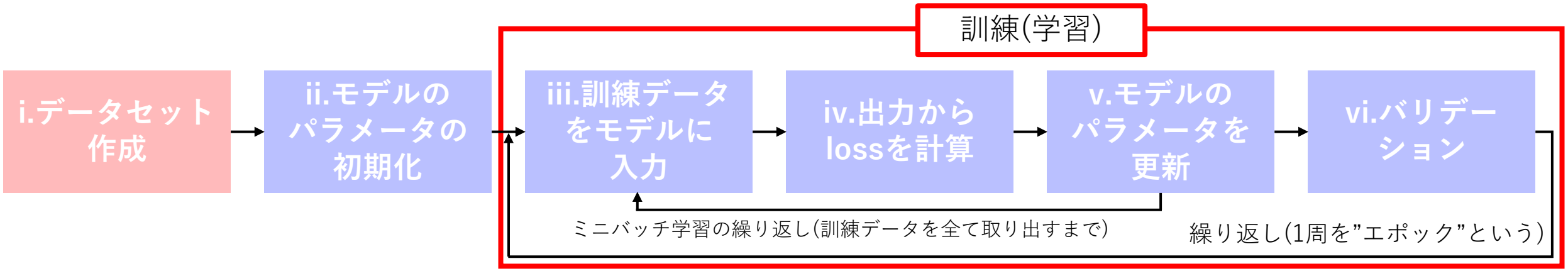
- **Dataset クラスを自作**
  - PyTorchのデータセットはtorch.utils.data.Datasetクラスを継承するのがルール
    - __getitem__メソッドと__len__メソッドはオーバーライド必須
      - __getitem__: 引数{id}番目のデータとターゲットを返すように書く
      - __len__:データセットのデータ数を返すように書く



i. データセット作成

- **3つのデータに分割**　　**Let’s run “./torch/nn/dataset.py”!!**
  - torch.utils.data.random_split関数が便利
    - 引数
      - dataset: torch.utils.data.Datasetを継承したデータセット
      - length: 割合を表すリスト (例) [0.8, 0.1, 0.1]
- **データの型はnumpy.float32にしておく**

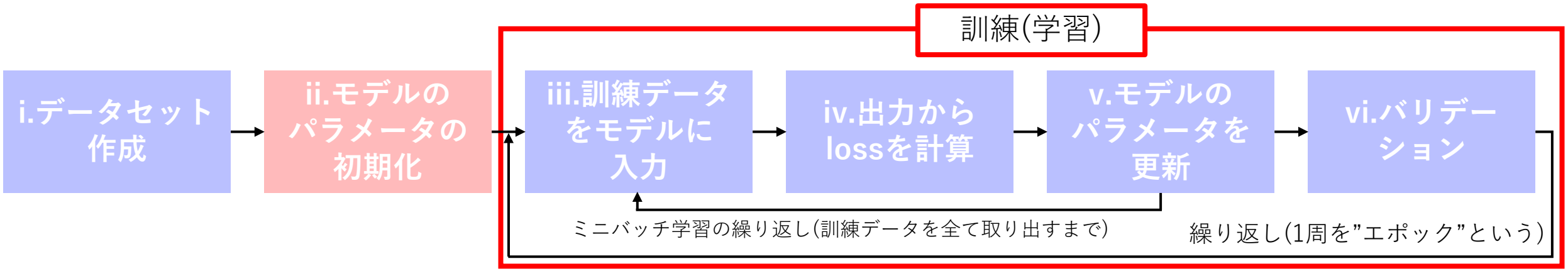
modelのパラメータがデフォルトでtorch.float32なので、dataがfloat64だったりすると後で怒られる。回避するために明示する。



ii. モデルの実装

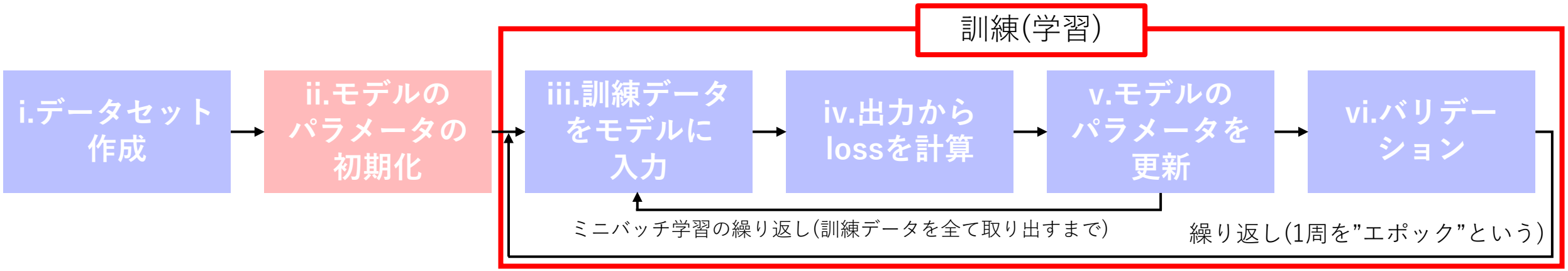
- **モデルの構成要素**
  - 層単位で全部用意されている(インスタンス化時に初期化される)
    - 全結合層: torch.nn.Linearクラス
    - conv.層: torch.nn.Conv2dクラス
    - pool.層: torch.nn.MaxPool2dクラス or torch.nn.functional.maxpool2d関数
    - relu関数: torch.nn.ReLUクラス or torch.nn.functional.relu関数
    - softmax関数: torch.nn.Softmaxクラス or torch.nn.functional.softmax関数

訓練するパラメータがある層は、classを使う
ない層は、関数でもよい



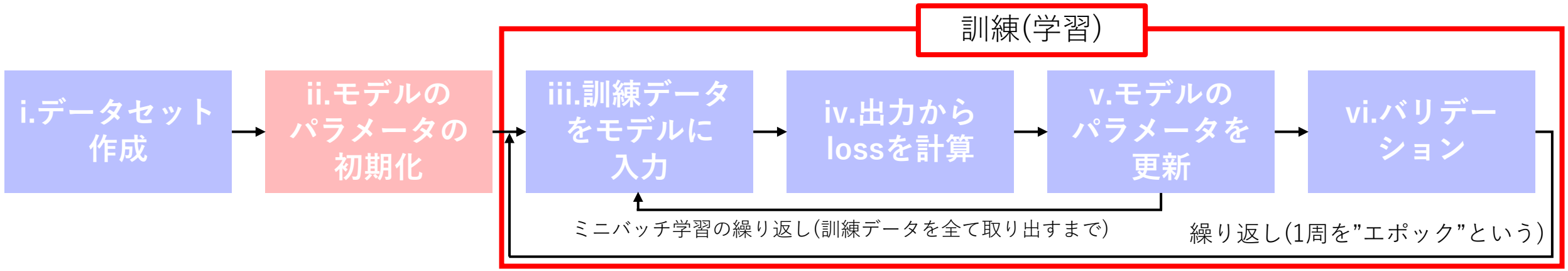
ii. モデルの実装

- **モデルの構成要素の属性の例**
  - weight: ウェイトを表すtensor(requires_gradは自動でTrue)
  - bias: バイアスを表すtensor(requires_gradは自動でTrue)



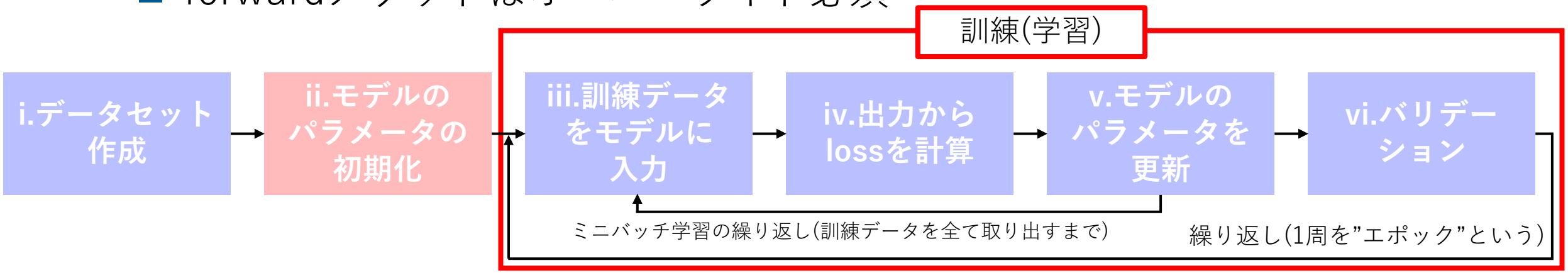
ii. モデルの実装

- **torch.nn.Linearクラス**
  - インスタンス化時の必須引数
    - in_features: 入力tensorの要素数
    - out_features: 出力tensorの要素数
  - forwardメソッドがあるのでインスタンスに直接入力tensorを引数に与えれば計算できる
  - ウェイトのサイズは(out_features, in_features)
  - バイアスのサイズは(out_features)



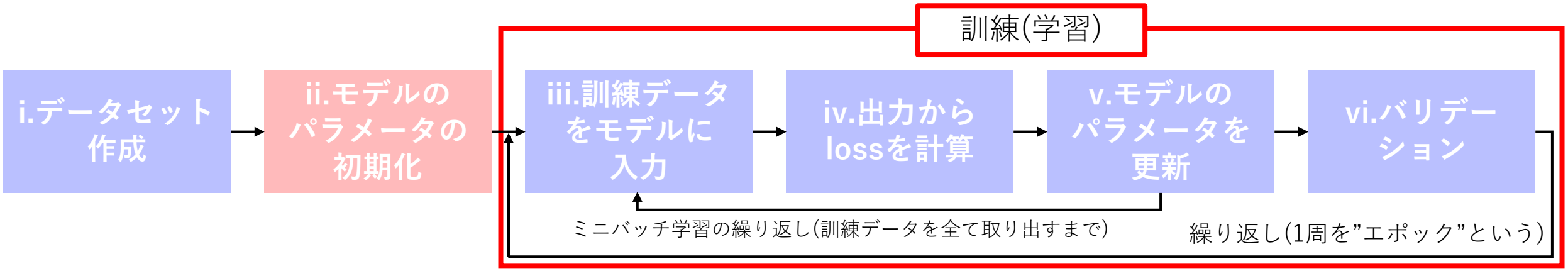
ii. モデルの実装

- **モデルの宣言**
  - inport文は「import torch.nn as nn」「imprort torch.nn.functional as F」と書くのが通例
  - PyTorchのモデルはnn.Moduleクラスを継承するのがルール
    - __init__メソッドでは、super().__init__()の後にメンバ変数としてネットワークに用いる層のクラス(or関数)をインスタンス化して持っておく
    - forwardメソッドはオーバーライド必須



ii. モデルの実装

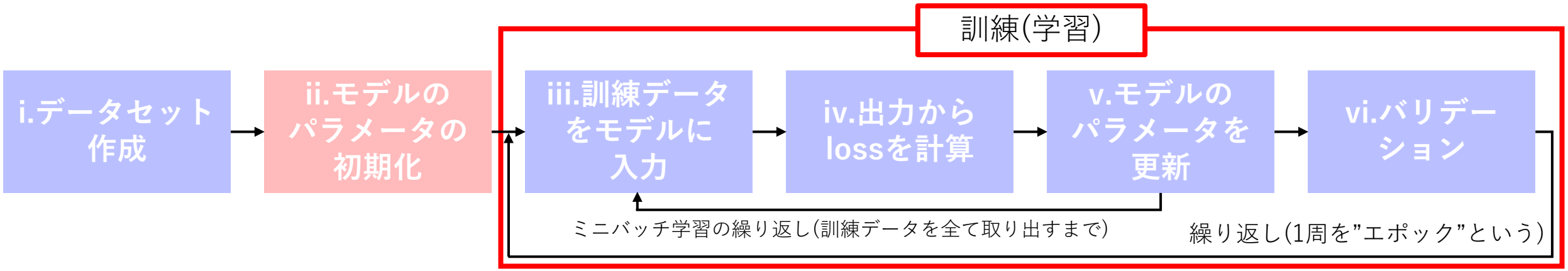
- **計算グラフの構築**
  - forwardメソッド (引数: モデルへの入力tensor)
    - ここに記述した数式に基づいて、自動で計算グラフが構築される
    - モデルの出力結果(Loss関数の直前)をreturn
    - オーバーライド必須
    - インスタンスに対して、直接tensorの引数を与えることで呼び出せる (__call__メソッドと似ている)



ii. モデルの実装

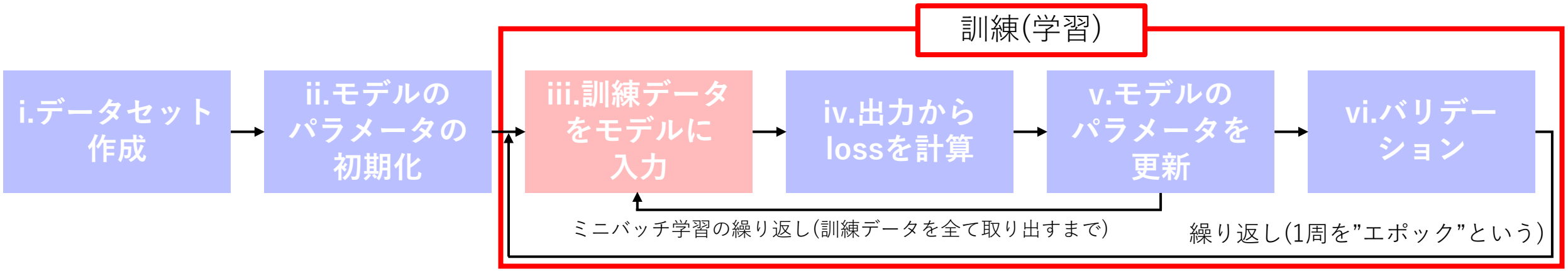
- **全体のパラメータの閲覧**
  - parametersメソッドは全体のパラメータを入力層側から、ウェイト、バイアスの順で生成するイテレータを返す
    - これを後でoptimizerに引数として渡す

Let's run "./torch/nn/model.py"!!



iii. データの入力

- **ミニバッチの作成**
  - torch.utils.data.DataLoaderクラスが便利
    - torch.utils.data.Datasetを継承したDatasetクラスのインスタンスを用いて、指定したバッチサイズのミニバッチを返すジェネレータ
    - 主な引数
      - dataset: データセット
      - batch_size: バッチサイズ
      - shuffle: シャッフルするか



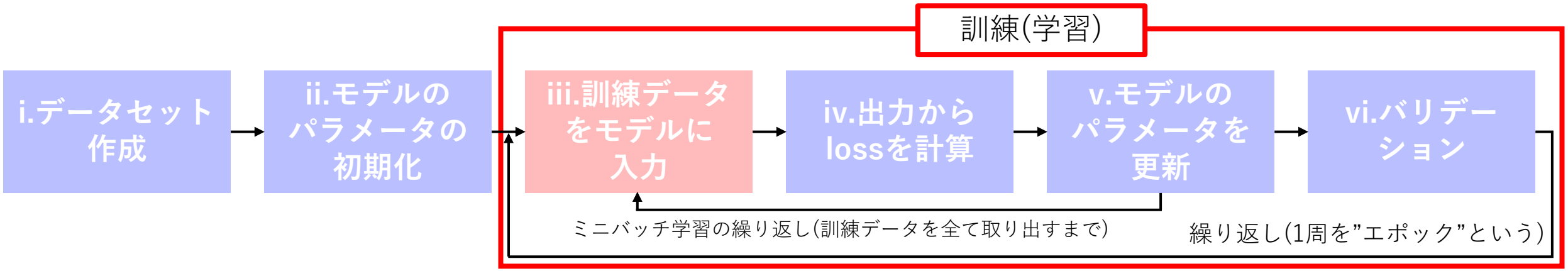
iii. データの入力

- **ミニバッチの作成**

Let’s run “./torch/practice_dataloader.py”!!

- torch.utils.data.DataLoaderクラスが便利
  - 戻り値
    - index0がデータ、index1が教師ラベルのリスト(自動でtensorになる)
      - ✓ サイズは、データが(batchsize, 説明変数数)、教師ラベルが(batchsize)
  - ジェネレータなので、for文で順番に取り出すか、iter({DataLoader})でイテレータにしてからnext({イテレータ})で1ミニバッチずつ取り出す

※入力がRGB画像の場合データのサイズは、(batchsize, 3, height, width)
※教師ラベルがone-hot codingされている場合そのサイズは、(batchsize, クラス数)

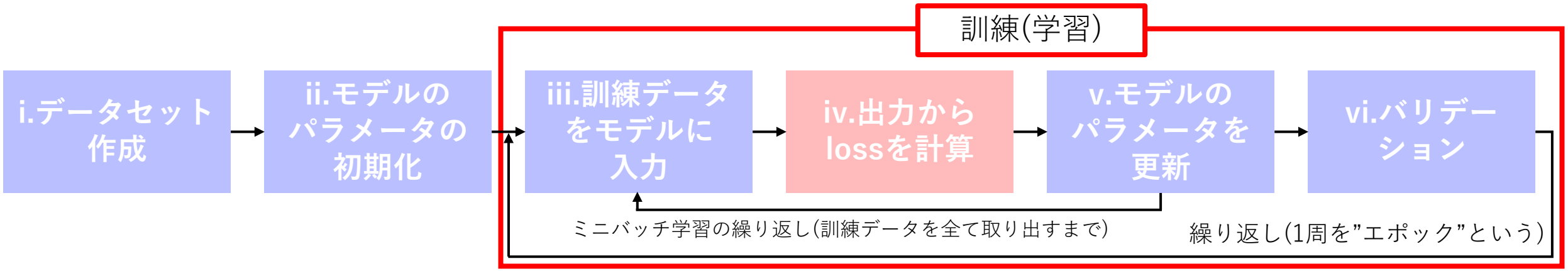


iv.出力からlossを計算

- **分類問題におけるCross-entropy Loss**

※回帰問題は、特に問題ない。CNNの演習で、分類問題を扱うので、そのとき再掲する。

  - nn.CrossEntropyLossクラス or F.cross_entropy関数を使用
    - one-hot encodingされている必要はない
    - いずれもはsoftmax関数による最終層活性化を含む
      - モデルのほうにsoftmaxを書かない

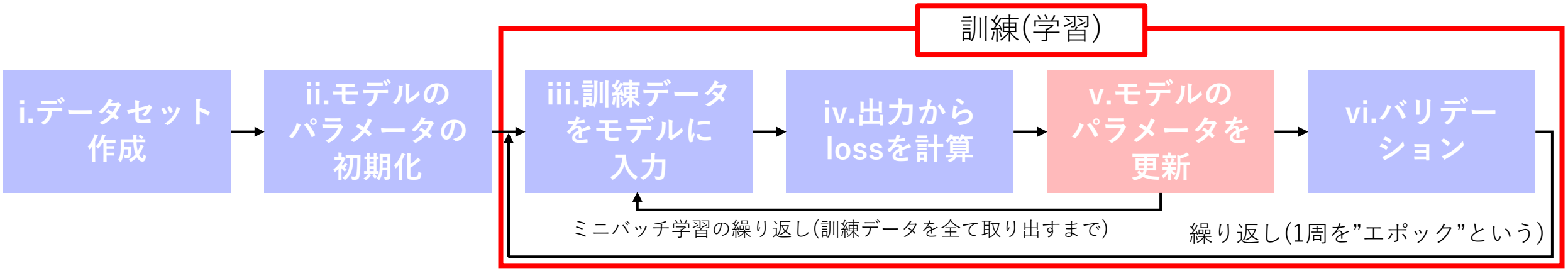


v. モデルのパラメータを更新

- **最適化アルゴリズム(optimizer)**

Let's run "./torch/practice_only_train.py" and confirm the loss decreases as the epoch increases!!

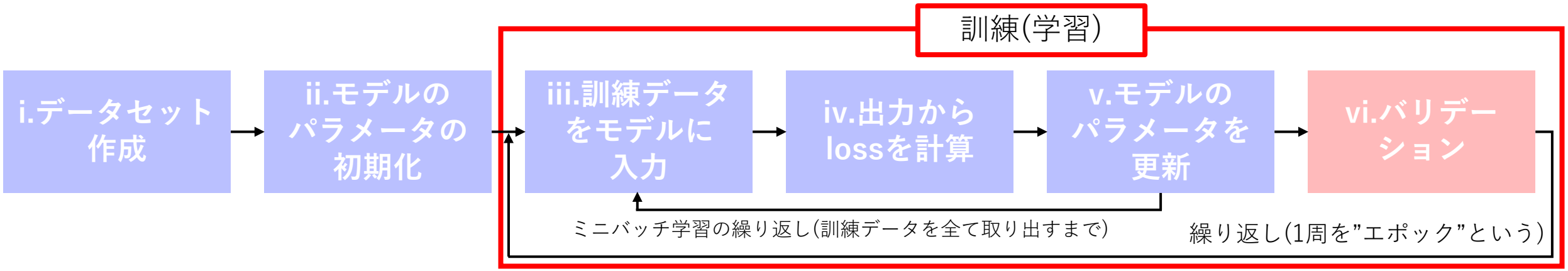
  - torch.optimにSGD、Adamなどが用意されている
    - 引数の例
      - params: modelのパラメータ model.parameters()で渡す
      - lr: 初期学習率 1e-3ぐらいが良いと思う
    - メソッドの例
      - step: その時点のgradを使って全訓練パラメータを一度に更新
      - zero_grad: gradを全消去 ←PyTorchはgradが加算されていく仕様なので、次のミニバッチに行くときに必ずzero_gradで勾配をリセットする



vi.バリデーション

- **パラメータの更新をしないようにだけ注意**
  - model.eval()を実行してdropoutなどの、モードによって挙動が異なる層のモードを推論モードに変更
    - 戻すにはmodel.train()を実行する
  - 「with torch.no_grad():」のコンテキスト内に書いた計算式は一切計算グラフが構築されない

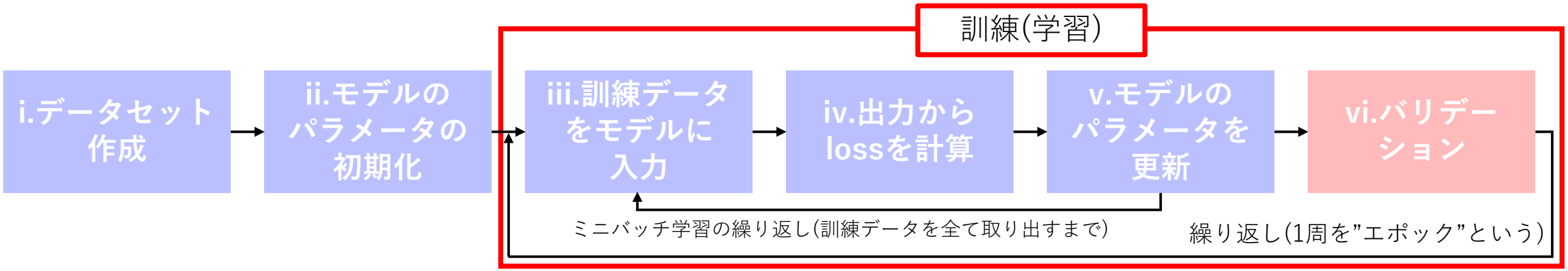
→バリデーションやテスト時はmodel.eval()を実行し、 torch.no_grad()コンテキスト内で計算を実行することを強く推奨



vi.バリデーション

- **モデルの保存**
  - 「torch.save(model.state_dict(), {保存するファイルのパス})」
    - バリデーション結果が最良のエポック時点のモデルをテストに使うとよい

- **学習率のスケジューリング、訓練の打ち切りなどをすることもある**



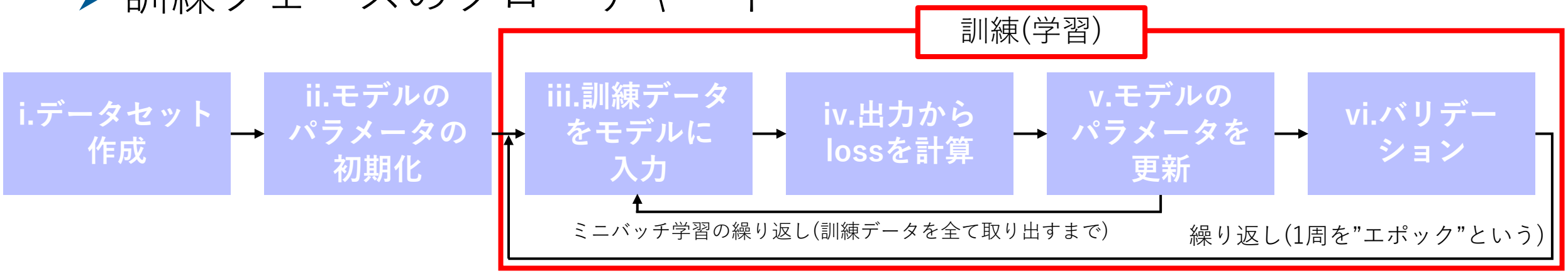
推論フェーズ

- **パラメータの更新をしないように注意**
  - model.eval()
  - with torch.no_grad(): コンテキスト
- **保存したモデルの読み込み**
  1. 「model = MyModel()」などでまずはモデルをインスタンス化
  2. 「model.load_state_dict(torch.load({保存ファイルのパス}))」でパラメータを上書きする形でロード

Let’s run “./torch/run_nn.py”!!

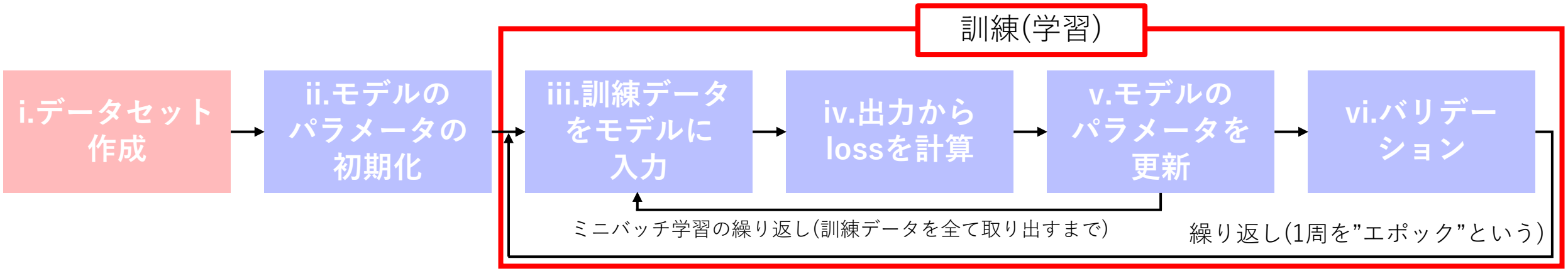
NNを実装してみよう

- 全結合NN
  - カリフォルニアの住宅価格を回帰する問題
- **CNN**
  - CIFAR-10データセットの分類問題
- **(復習) 訓練フェーズと推論フェーズ**
  - 訓練フェーズのフローチャート



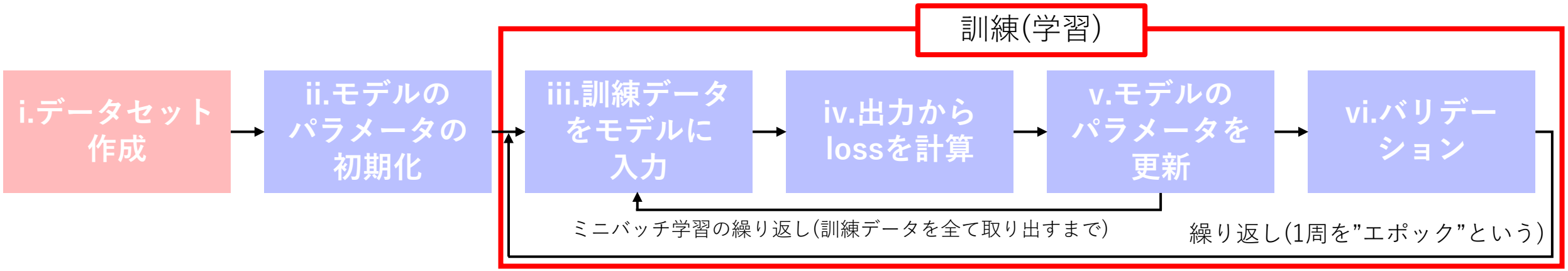
i. データセット作成

- **CIFAR-10**
  - RGBで(32, 32)ピクセルの画像に写っているものが、{飛行機,自動車,鳥,猫,鹿,犬,蛙,馬,船,トラック}のどれかに分類する問題
    → torchvisionライブラリに用意されている
  - 今回はtorchvisionが予め作ってくれているDatasetを使う
    - __getitem__を呼ぶとPIL型で返ってくる



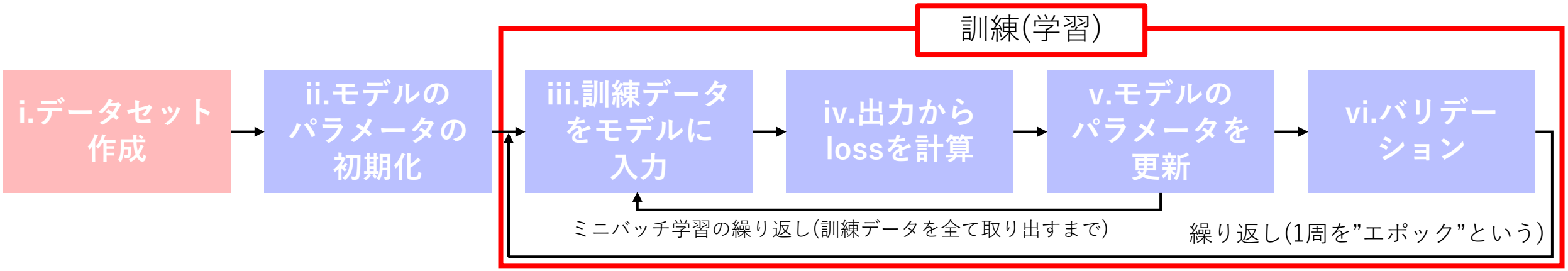
i. データセット作成

- **torchvision.datasets.CIFAR10**
  - trainデータ: インスタンス化時に引数にtrain=Trueを設定
    - 主な属性
      - data: (32,32,3)の画像50000枚で、サイズ(50000,32,32,3)のnp.ndarray
      - target: 要素数50000のlist
  - testデータ: インスタンス化時に引数にtrain=Falseを設定
    - 主な属性
      - data: (32,32,3)の画像10000枚で、サイズ(10000,32,32,3)のnp.ndarray
      - target: 要素数10000のlist



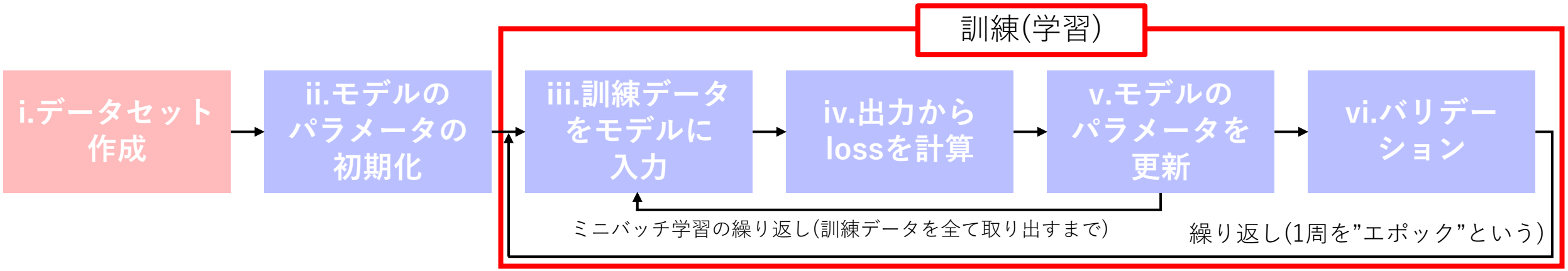
i. データセット作成

- **torchvision.transformsライブラリ**
  - 様々なデータの前処理を提供するクラスを集めたライブラリ
    - ToTensorクラス: PILやnp.ndarrayをtorch.Tensorに変換
    - Normalizeクラス: 指定した平均値・標準偏差で標準化
    - Lambdaクラス: 引数に取ったlambda関数を実行する(カスタマイズ可能)
    - Composeクラス: 上記のような変換インスタンスを要素に持つリスト形式の引数をとり、複数の前処理を連続して行う



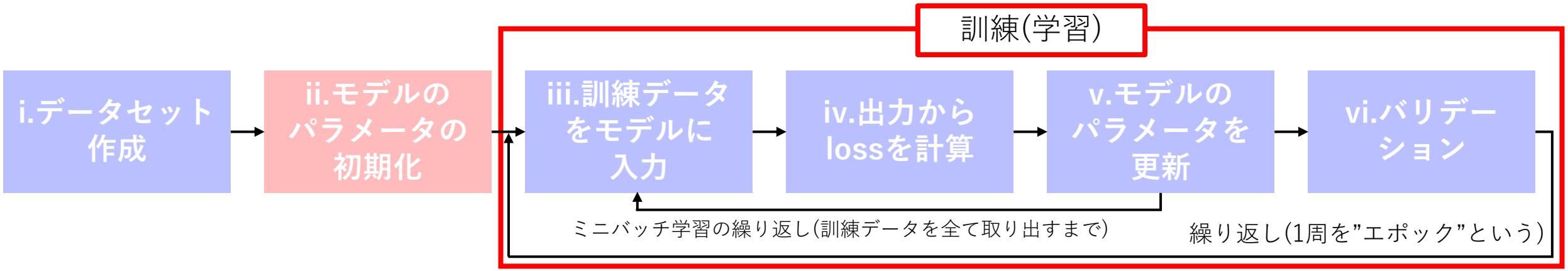
i. データセット作成

- **torchvision.transformsライブラリ** Let's run "./torch/practice_transform.py"!!
  - CIFAR10データセットのインスタンス化時に引数transform(data用), target_transform(targets用)に前処理のインスタンスを設定することで、__getitem__が呼ばれたとき、自動で前処理が実行される
  - 画像データは[0,1]に正規化、または平均0.5標準偏差0.5で標準化することが多い
  - PyTorchでは、画像は(channel, height, width)の順で持つことになっている。PIL画像にToTensorを使うと、勝手に入れ替えてくれる



ii. モデルの実装

- **CNN**
  - よくあるネットワークの構造は、「{conv. + relu + pool.}の繰り返し + flatten(平滑化) + {fc(全結合) + relu}の繰り返し」
  - 入力tensorは(batchsize, channel, height, width)のサイズでなければならない
    - batchsize=1でも4次元にすること



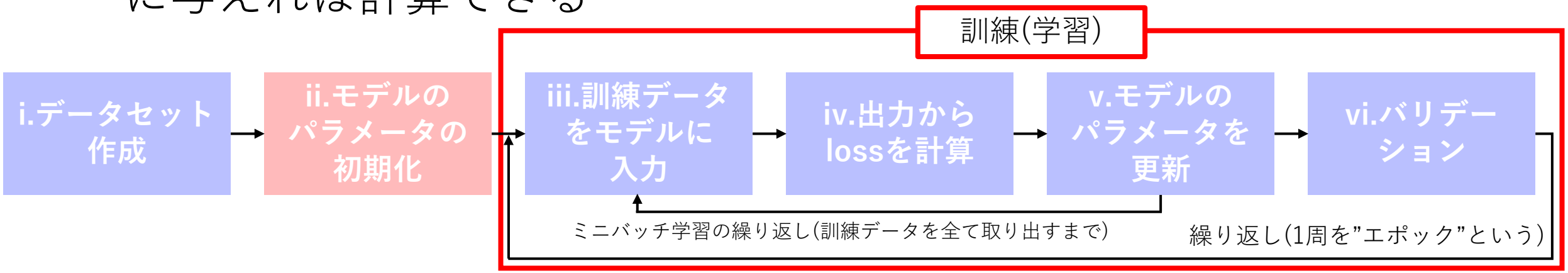
ii. モデルの実装

- **torch.nn.Conv2dクラス**
  - インスタンス化時の必須引数
    - in_channels: 入力チャネル数
    - out_features: 出力チャネル数
    - kernel_size: カーネルサイズ

※パディング、ストライドなども
ここで設定できる
デフォルトはそれぞれ0, 1

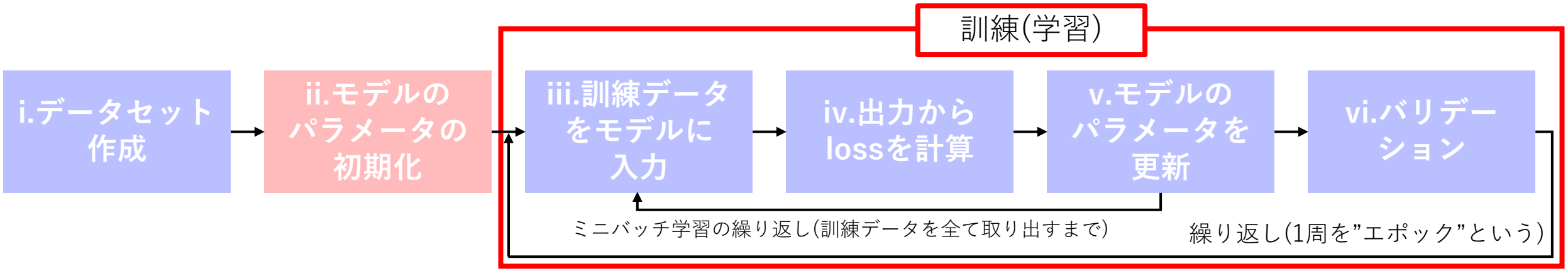
※入出力tensorのサイズは自動で
計算されるので書かない

  - forwardメソッドがあるのでインスタンスに直接入力tensorを引数に与えれば計算できる



ii. モデルの実装

- **torch.nn.Conv2dクラス**
  - ウェイトは(out_features,in_features,kernel_height,kernel_width)、バイアスは(out_features)のサイズになっている
    - ウェイトは各入力チャネル(in_features個)に対して、out_features個のカーネルがあるので、全体のカーネル数はその積で与えられる
    - バイアスは各入力チャネルに対して等しく与えられるので、出力チャネル数だけのパラメータがある

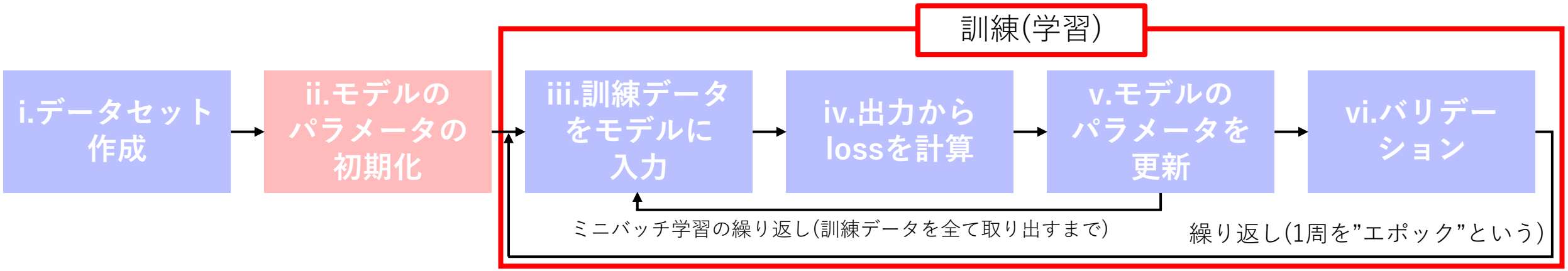


ii. モデルの実装

- **torch.nn.functional.max_pool2d関数**
  - 必須引数
    - input: 入力tensor
    - kernel_size: カーネルサイズ

※パディング、ストライドなどもここで設定できる
デフォルトはそれぞれ0, kernel_size

- **torch.nn.MaxPool2dクラス**
  - インスタンス化時の必須引数
    - kernel_size: カーネルサイズ



ii. モデルの実装

- **入出力tensorのサイズ**

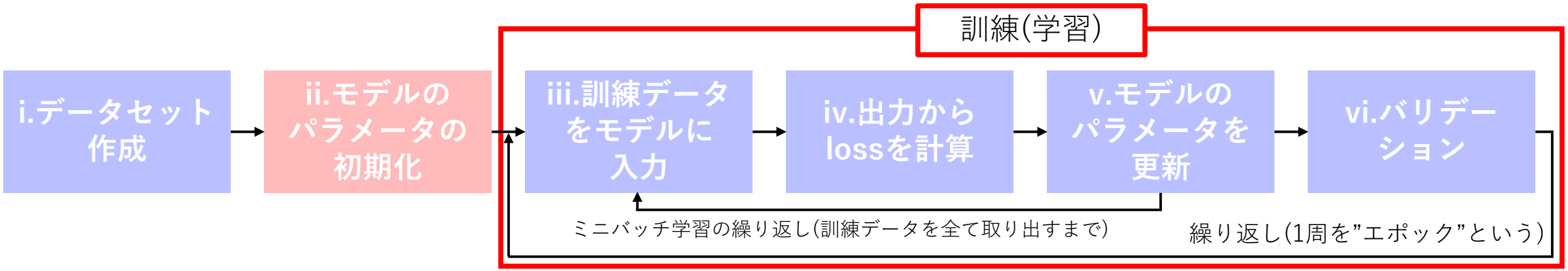
I: 入力画像サイズ, O: 出力画像サイズ,
K, カーネルサイズ, P: パディング, S: ストライド

$$O = \mathrm{floor}\left(\frac{I - K + 2P}{S}\right) + 1$$

※詳細は「ゼロから作るDeep Learning―Pythonで学ぶディープラーニングの理論と実装」を参照。

- K=3, P=1, S=1だとサイズが変わらない(conv.)
- P=0, S=K=2だとサイズが半分になる(pool.)
  ことを覚えておくと便利(よく使う)

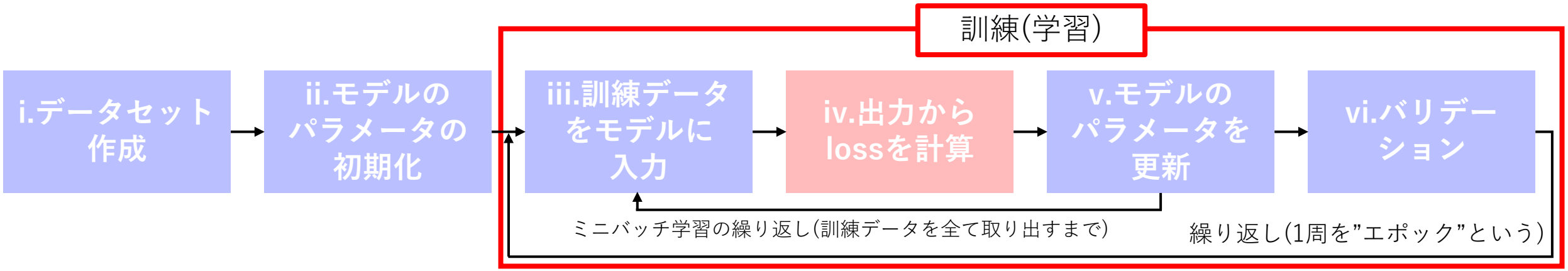
Let's run "./torch/cnn/model.py"!!



iv.出力からlossを計算（再掲）

- **分類問題におけるCross-entropy Loss**
  - nn.CrossEntropyLossクラス or F.cross_entropy関数を使用
    - one-hot encodingされている必要はない
    - いずれもsoftmax関数による最終層活性化を含む
      - モデルのほうにsoftmaxを書かない

Let’s run “./torch/run_cnn.py”!!



実装編 おわりに

- **NNとCNN、回帰と分類、やってきました**
  - 最初はなかなか入ってこないと思うけど、やってれば慣れます
    - このマニュアルをのちのち見返すとそこそこ学びがあると思います
  - PyTorch関連のライブラリは星の数ほどあります
    - 紹介しきるのは不可能なので頑張ってそれぞれのマニュアル読んでください (だいたい英語だけど…)
  - PyTorchが分かればTensorFlowも分かります
    - Pythonが分かればJavaが分かるのと一緒です
- **次はGPUのセットアップをしましょう**

Let’s go to the next chapter!!

3. 環境構築編

- この情報は、そのうち使い物にならなくなる可能性が高いです。その場合は、すみませんけど頑張ってください笑

CNNの実行時間

- **長い…こんなの待ってられるか！**
  - TensorをGPUに載せて高速に計算しましょう
  - NKTLABには2023月10月現在GPUが使えるマシンが8台あります
    - RTX-01 ~ RTX05
    - XEON03 ~ XEON05
  - 新しいGPUが来たら、セットアップをしましょう
    - やること
      1. 普通のクラスターと同じセットアップをする(Pythonインストール等)
      2. CUDAをセットアップ
      3. CUDAのバージョンに合ったGPU用のPyTorchをpip install

セットアップ手順 1

- **NKTLab Manualに従って、普通のクラスターと同じセットアップをしましょう**
  - NakataLab Manualは2023年10月現在、先生もいるTeamsの一般チャネルからアクセスできる共有フォルダにあります。OneNoteのやつです。
  - 特に、Python3.10.0以上をインストールしてください。

セットアップ手順 2

- **CUDAのセットアップをしましょう**
  - CUDAとは: NVIDIA製のGPUを動かすドライバ
  - いろいろ細かいことについてはネットや仕様書などを確認しましょう
- **とりあえずセットアップできればいい人は**
  1. 「¥¥192.168.11.3¥general¥02_開発環境¥33_GPU開発環境¥11.8」を対象のマシンにコピー
  2. 11.8にある「cuda_11.8.0_522.06_windows.exe」を管理者として実行
  3. インストーラの指示に従って普通にインストールする
  4. 11.8にある「cudnn-windows-x86_64-8.9.1.23_cuda11-archive」ディレクトリの中身を、まるまる「C:¥Program Files¥NVIDIA GPU Computing Toolkit¥CUDA¥v11.8」にコピーする

→これでCUDA ver.11.8 のセットアップ完了です

セットアップ手順 3

- **GPU用のPyTorchをインストールしましょう**
  - 「pip install torch」だとCPU用がインストールされます
  - GPU用はPythonのバージョン・CUDAのバージョンによってインストール文が変わります
- **とりあえずセットアップできればいい人は**
  - 「py -3.10 –m pip install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu118」

→これでPyTorch(,torchvision,torchaudio)のインストール完了です

セットアップ成功の確認

- **GPU用のPyTorchがインストールされたPythonを対話モードで実行し、**
  1. import torch
  2. torch.cuda.is_available()

を実行してTrueが返ってきたらセットアップ成功

環境構築編 おわりに

- **理論編・実装編と比べたら軽かったですね**
  - こんなの新しいGPUが来たときしか使わないので、そのときに見ながらセットアップしてください
- **さまざまな要因で、このセットアップ方法が使えなくなるのが先の話であることを祈っています…**
  - 必ずそのときが来るので、もしそうなったら頑張ってください
- **次が最後の章です セットアップしたGPUを使って時短しましょう いっぱい研究できてうれしいね**

Let’s go to the next chapter!!

4. GPU利用編

準備

1. 好きなディレクトリでVSCodeを起動
2. NKTLABのGithubからサンプルコードをダウンロードしてカレントディレクトリに配置 https://github.com/YNU-NakataLab/cnn_manual.git
3. 必要なライブラリをインポート
   「pip install scikit-learn 」

torch.Tensorクラス

- **GPU関連のメソッド・属性**
  - toメソッド → 非破壊処理、「x = x.to(“cuda”)」としないとtensorがメモリに乗るだけで変数にアクセスできないので注意
    - .to(“cuda”)でそのtensorをGPUに配置
    - .to(“cpu”)でそのtensorをCPUに配置
  - cpuメソッド
    - .cpu()と.to(“cpu”)は同じ
  - device属性
    - そのtensorがcpu上にあるかgpu上にあるかわかる

torch.Moduleクラス

- **GPU関連のメソッド・属性**
  - toメソッド →破壊処理 「model.to(device)」だけでOK
    - .to(“cuda”)でそのmodelをGPUに配置
    - .to(“cpu”)でそのmodelをCPUに配置
  - cpuメソッド
    - .cpu()と.to(“cpu”)は同じ

GPU上での計算

- **計算にかかわるすべてのtensorをGPUに配置して、今まで通りにコーディングすればよい**
  - 基本的にはinput tensorとmodelをGPUに配置しておけば問題ない
  - GPU上のtensorによる計算から生まれたtensorもまた、GPU上に配置される
  - torch.cuda.is_available()でgpuが利用可能かわかるので、初めに「device = “cuda” if torch.cuda.is_available() else “cpu”」と文字列型変数deviceを用意しておき、「model.to(device)」などと書けばgpu使用時とcpu使用時でコードを分ける必要がない

GPU上での計算

- **注意点**
  - GPU上のtensorとCPU上のtensorが混在した状態で計算することはできない
  - numpyメソッドやtorch.save関数、torch.load関数など、GPU上のtensorに対して実行できないものがある
    - .cpu()もしくは.to(“cpu”)でCPU上に移してから実行する

Let’s run “./practice_gpu.py”, “./run_nn_gpu.py”, and “./run_cnn_gpu.py”!!

5. TensorFlow v2 編

ディープラーニングフレームワーク

- **ニューラルネットワークの設計・学習・評価などに必要な一連の実装を容易にするためのPython上のフレームワーク**
  - PyTorch
    - 主に研究分野で使われている。論文に載っているソースコードはPyTorchで書かれていることが多い。
  - TensorFlow　←これを習得しよう！
    - 主に企業が使っている。PyTorchより多くインターネット上に記事が載っている。Kerasと混ざって紛らわしい。
  - Keras　←これを習得しよう！
    - TensorFlowで使える便利な機能を提供する。便利すぎて、もはや単独のライブラリになってしまった。

TensorFlow

- **PyTorchと同じくTensorを操作する**
  - ver. 1 と ver. 2で大きく仕様が異なる
    - どちらも習得しよう
    - まずver. 2から始めよう

- **TensorFlowとKerasの関係**
  - 本来はTensorFlowとKerasは別のライブラリだった
  - 今はTensorFlowの中にKerasが組み込まれていて、Kerasの便利機能を使いながら、TensorFlowでコーディングする
    - PyTorchで「import pytorch.nn.Linear」を使うように、TensorFlowでは「import tensorflow.keras.layers.Dense」を使う(同じ全結合層)

準備

1. Python3.10とVSCodeをインストール
2. 好きなディレクトリでVSCodeを起動
3. 3. 環境構築編 のCUDAのセットアップまでを行う
4. 仮想環境を作る「py -3.10 –m venv {venv_name}」
5. 仮想環境を起動「./{venv_name}/Scripts/activate」
6. 必要なライブラリをインポート
   「pip install -r ./tensorflow/requirements.txt 」

※手順4 でエラーが出る場合は、Powershellを管理者として実行して「PowerShell Set-ExecutionPolicy RemoteSigned」を実行し、VSCodeを再起動して再実施

class tf.Variable (tf.Tensor)

- **行列形式のデータを扱うクラス**
  - インスタンス化方法
    1. import tensorflow as tf
    2. x = tf.Variable([[1, 2], [3, 4]])　←tf.Variable({数値, リスト, np.ndarrayなど})
  - 属性(クラス変数)の例
    - shape: tensorのサイズ
    - dtype: tensorの要素の型
  - メソッドの例
    - numpy: numpyへの変換

Let's run "./tensorflow/practice_tensor_1.py"!!

- **tf.expand_dimsで次元を拡張(axisで追加する次元を指定)**

class tf.GradientTape

- **計算を記録し、後で勾配を取得するためのクラス**
  - torchとは異なり、tfではtensor自身に勾配は記録されない
  - 「with tf.GradientTape() as tape:」コンテキスト内に勾配を計算する変数の計算グラフを記述する
  - withを抜けた後に、gradientメソッドで勾配を取得できる(withを抜けているが一度だけ呼び出せる)
    - 引数
      - target: 微分される変数
      - source: 微分する変数のリスト
    - 戻り値: sourceに対応する勾配のリスト

Let’s run “./tensorflow/practice_tensor_2.py”!!

Dataset, Dataloader

- **tf.data.Datasetクラス**
  - tfのデータセットクラスで、データローダーを兼ねる
  - 「tf.data.Dataset.from_tensor_slices([data, labels])」でデータセット(ジェネレータ=データローダー)を作成
    → for文で取り出せる
  - メソッドの例
    - shuffle({データ数}) でデータセット内全体にわたってシャッフル
    - batch({バッチサイズ}) でミニバッチ化
    - map({関数})で全データに対して関数を実行(前処理やデータ拡張に使える)
  - 例
    - train_set = tf.data.Dataset.from_tensor_slices((X_train, y_train)).shuffle(len(y_train)).batch(100).map(lambda x: x / 255)

モデルの実装

- **tf.keras.layersにさまざまなレイヤーが用意されている**
  - 全結合層: Dense(units, activation, ...)
    - units(必須): 出力ノード数
    - activation: 活性化関数(“relu”などと指定)
    - torchとは異なり、入力ノード数は指定しない
  - conv.層: Conv2D(filters, kernel_size, padding, ...)
    - filters(必須): 出力チャネル数
    - kernel_size(必須): カーネルサイズ
    - padding: パディング数
    - torchとは異なり、入力チャネル数は指定しない

tfではウェイトshapeは
(height, width, in_channel, out_channel)の順！
特徴マップshapeは
(batchsize, height, width, channel)の順！

  - pool.層: MaxPool2D(...)
  - 活性化関数層: ReLU(...), SoftMax(...)など
  - 平滑化層: Flatten(...),ドロップアウト層: Dropout(...),合流層: Concatenate(...)
  - データ拡張層: RandomFlip(...),Reshape(...)など

モデルの実装

- **3つのモデルの定義方法**
  1. Sequential API
     - 最も簡単、分岐合流などを実装できない
  2. Functional API
     - 分岐合流などを実装できる
  3. Subclassing API
     - tf.keras.Modelを継承したクラスを定義する、PyTorchと似た書き方

実はPyTorchにもSequential APIのようなモデルの定義方法(torch.nn.Sequential)があるがあまり見ないので紹介しなかったTensorFlowのほうはどれもよく見るのですべて習得しておこう

- **注意**
  - tfでは最終層活性化関数(softmaxなど)の定義が必要

モデルの実装

- **Sequential API**
  - 「model = tf.keras.Model()」で空のmodelをインスタンス化して、「model.add(tf.keras.layers.Dense(units=100))」などでレイヤーを追加する方法
  - もしくは、tf.keras.Modelのインスタンス化時に、引数にレイヤーのリスト([tf.keras.layers.Dense(units=100), tf.keras.layers.Dense(units=100)]など)を与えてもよい
  - 入出力tensorが1つずつで、分岐や合流のないモデルしか定義出来ないので、初心者向けの書き方

モデルの実装

- **Functional API**
  - 「input = tf.keras.layers.Input()」で入力レイヤーを定義し、
    「x=tf.keras.layers.Conv2D(filters=32, kernel_size=3)(x)」
    「x=tf.keras.layers.ReLU()(x)」
    「x=tf.keras.layers.Flatten()(x)」
    「x=tf.keras.layers.Dense(units=10)(x)」
    「outputs= tf.keras.layers.ReLU()(x)」のように入出力をつなぎ、最後に「model = tf.keras.Model(inputs=inputs, outputs=outputs)」で入出力をモデルにセットする方法
  - 分岐や、Concatenate層を使って合流も定義できる
  - 複数の入出力を持つモデルも定義できる

モデルの実装

- **Subclassing API**
  - PyTorchのモデル定義と同じように、tf.keras.Modelを継承して、__init__関数内でsuper().__init__()の呼び出しと「self.fc1=tf.keras.layers.Dense(unit=100)」のようにレイヤーのメンバ変数定義を行い、modelへの入力xを引数に持つ**__call__**関数内にレイヤーの接続を記述する方法
    - PyTorchと異なりforward関数ではなく__call__関数なので注意
  - 最も柔軟にモデルを実装できる
  - Dropout, BatchNormalization層などは、__call__のtraining引数で、モードを変えるため、modelの__call__関数にもtraining引数をつけておくことが多い

loss func, optimizer, metric

- **loss func**
  - 「loss_func = tf.keras.losses.SparseCategoricalCrossentropy()」などで定義
    - 「Sparse」は、one-hotでなくてもいいことを示す
- **optimizer**
  - 「optimizer = tf.keras.optimizers.Adam()」などで定義
    - torchでは引数にmodel.parameters()を渡したがtfではそうはしない
- **metric(任意)**
  - 「metric = tf.keras.metrics.SparseCategoricalAccuracy()」などで定義

mini-batchループの実装

- **書き方が大きく2つに分かれる**
  1. 組み込みメソッドの利用
     - tf.keras.Modelのインスタンスにはcompile, fit, evaluate, predictメソッドが用意されていて、これらを利用すればfor loopを全く書かなくてよい
     - dataset, dataloaderの作成も必要ない
       - compile: optimizer, loss function, 評価値(metrics)などをmodelにセット → 全てをmodelに集約
       - fit(訓練): 入力x, ラベルy, バッチサイズ, エポック数, val.データの分割割合などを指定して訓練全体を実行
         - ✓ loggingも適切に行ってくれる
         - ✓ 戻り値はHistoryクラス history属性に辞書形式で学習履歴が残る
       - evaluate(評価): 入力x, ラベルyを指定し、lossと評価値を計算して返す
       - predict(推論): 入力xを指定し、modelの出力(推論結果)を計算して返す

要は、model, loss func, optimizer, metricsなどのパーツだけ手動でインスタンス化して、訓練/推論に関すること(パーツの運用)は全部tf.kerasに任せる方法

mini-batchループの実装

要は、訓練/推論に関すること(インスタンス化したパーツの運用)も全部自分で決めて書く方法

- **書き方が大きく2つに分かれる**
  2. custom training loopの実装
     - PyTorchと同じように、for loopを自分で書く
       → modelとoptimizerやloss functionは独立
     - 手動で書くもの
       - forward伝播: for loop内の「with tf.GradientTape() as tape」コンテキスト内でmodelの出力やlossを定義
         - ✓ tfではloss funcなどの引数の順番が({labels},{outputs})でtorchと逆なので注意
         - ✓ ミニバッチ1つ分の計算を関数化し、@tf.functionでデコレートすると高速化される
           - ただしデバッグがしにくくなるのでデバッグ時は外すことを推奨
       - backward伝播: 「gradients=tape.gradient(loss, model.trainable_weights)」で全パラメータに対するlossの勾配を取得し、「optimizer.apply_gradients(gradients, model.trainable_weights)」でパラメータを更新
       - logの出力: エポックやミニバッチごとのlossや評価値のlogging

コードを動かしてみよう

- **./tensorflow ディレクトリ内**
  - NN × 回帰

    Let’s run !!
    - run_nn.py … custom training loop × Subclassing API
    - run_nn_keras.py … fit × Sequential API (run_nn.pyのコメントアウトあり)
    - run_nn_keras_wo_old_code.py … fit × Sequential API (run_nn.pyのコメントアウトなし)
      - (run_nn_keras.pyとrun_nn_keras_wo_old_code.pyは同じコードです)
  - CNN × 分類
    - run_cnn.py … custom training loop × Subclassing API
    - run_cnn_keras.py … fit × Functional API

GPUの利用

- **TensorFlowでは、GPUが使える環境であれば勝手に使ってくれる**
  - 使える環境かどうかを確かめるには、Pythonのインタプリタで「 tf.config.list_physical_devices(‘GPU‘) 」を実行する。空でないリストが返ってくればOK！
- ~~**PyTorchはtoメソッドで切り替えが簡単にできるので個人的にはPyTorchの方が好きです**~~

TensorFlow v2編 おわりに

- **PyTorchが分かればTensorFlowも分かったでしょ**
  - 違うところだけ押さえておきましょう
- **最後にv1の話をして終わりです！**

Let’s go to the next chapter!!

6. TensorFlow v1 編

TensorFlow v1

- **v1 と v2 の違い**
  1. Define and Run (v1) か Define by Run (v2) か
  2. tf.Sessionがある(v1) か ない (v2) か
  3. tf.placeholderがある(v1) か ない (v2) か
- **v1習得の必要性**
  - v2に移行されたのは2019年のため、少し前の研究はv1で書かれていて、未だにv1で書き続けている研究者も一定数いる
    - v2の実行環境で、v1のコードを動かす
    - v1のコードを読んで理解する　　ことが必要になる場合がある
  - v2との違いだけ理解しておこう

TensorFlow v1

1. Define and Run (v1) か Define by Run (v2) か

- [v1] Define and Run: 先に計算グラフを構築してから値を流す
  - 新しいtensorの定義は、すなわち計算グラフの定義であり、その値は確定しない
- [v2] Define by Run: 値を流しながら計算グラフを構築する
  - 新しいtensorを定義すると、ただちにその値が確定する

2. tf.Sessionがある(v1) か ない (v2) か

- [v1] 計算グラフを構築した後、「with tf.Session() as sess:」コンテキスト内でsess.run({定義した計算グラフ})を実行することで、tensorの値が確定し、runメソッドの戻り値に返ってくる

3. tf.placeholderがある(v1) か ない (v2) か

- 変数の種類 **Let's run "./tensorflow/practice_session.py"!!**
  - [v1/v2] tf.Variable: 計算グラフ上の変数 NNで例えると、ウェイトやバイアス
    - [v1のみ]初期化が必要
  - [v1/v2] tf.constant: 計算グラフ上の定数
    - 値の変更不可
  - **[v1のみ] tf.placeholder**: 計算グラフ上の入力引数 NNで例えると、入力ノード
    - 型とshapeを指定して、メモリだけ確保する (shapeは省略可)
    - sessionコンテキスト内のsess.runのfeed_dict引数に実際の入力値を与えることで値が確定

- **v2環境でv1コードを動かす場合は「tf.compat.v1.disable_v2_behavior()」を実行**
  - 戻すには「tf.compat.v1.enable_v2_behavior()」を実行

おわりに

- お疲れ様でした！ここまで身につけたあなたは立派な深層学習エンジニアです！あとはたくさんコードを書いて慣れましょう！！良い研究ができることを祈っています。
- このマニュアルは約1週間で全て一人で書き上げたので、あちこちにミスがあると思います。見つけたら都度修正とバージョンアップをよろしくお願いします。

2023/10/16

ver. 0.0.0 制作

3期 藤澤 大世