機械学習講習会

第二回：線形回帰と勾配降下法

2025/06/26

@Kobakos32

#event/workshop/

前回の復習

- **機械学習の種類**: 教師あり学習、教師なし学習、強化学習
- **教師あり学習**: 入力 x から出力 y を予測する関数 f を学習
  - $y \approx f(x)$
- **学習**: 予測が正解に近づくように関数のパラメータを調整
- **損失関数**: 予測と実際の値の「近さ」を数値化する関数
- **線形回帰**: $y = wx + b$ の形で関係をモデル化

今回は実際に線形回帰モデルを**学習**させる方法を学びます。

もくじ

1. 線形回帰モデルの復習
2. 損失関数の導入
3. 勾配降下法
4. 分類問題への応用
5. 線形回帰の限界

線形回帰モデルの復習

線形回帰モデル

問題設定: 入力 x から出力 y を予測したい

モデル: $y = wx + b$

- w: 重み（係数）- 直線の傾き
- b: バイアス（切片）- y軸との交点

学習の目標: 訓練データに最もよく合う w と b を見つける

変数の変更

前回は、損失関数は y (正解値) と $wx + b$ (予測値)の関数として定義しました。

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

今回は、w と b の関数として定義します。

$$L(w, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - (wx_i + b))^2$$

（学習段階で動かすものはパラメーターなので）

（復習）平均二乗誤差 (MSE: Mean Squared Error)

最もよく使われる損失関数の一つ：

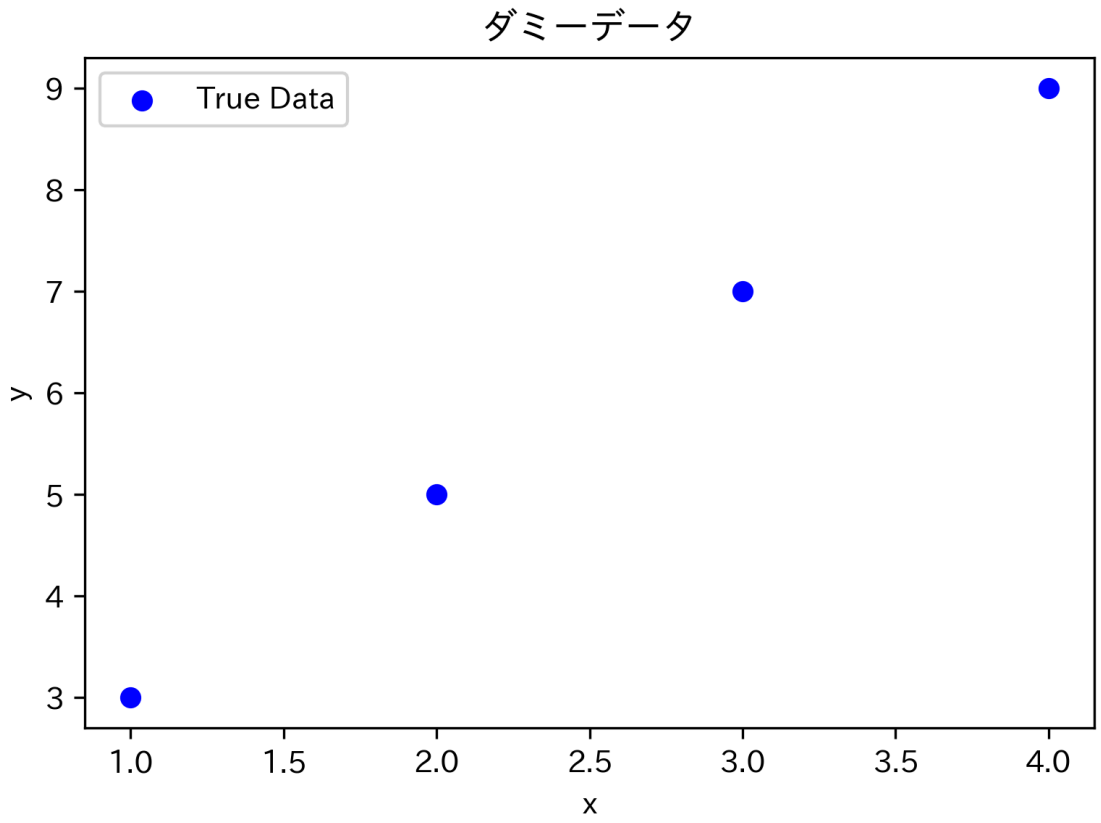
$$L(w, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - (wx_i + b))^2$$

- N: データの数
- (x_i, y_i): i 番目のデータの組
- $wx_i + b$: i 番目の予測値
- $(y_i - (wx_i + b))^2$: 誤差の二乗

ダミーデータ

まずは、とても簡単なダミーデータを考えてみましょう。

x	y
1	3
2	5
3	7
4	9



損失関数の例

先ほどのデータで $w = 1, b = 3$ の場合：

x	実際のy	予測 $\hat{y} = wx + b$	誤差 $y - \hat{y}$	誤差の二乗
1	3	4	-1	1
2	5	5	0	0
3	7	6	1	1
4	9	7	2	4

$$L(1, 3) = \frac{1}{4}(1 + 0 + 1 + 4) = \frac{6}{4} = 1.5$$

勾配降下法

問題の定義

目標: 損失関数 $L(w, b)$ を最小化したい

$$\min_{w,b} L(w, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - (wx_i + b))^2$$

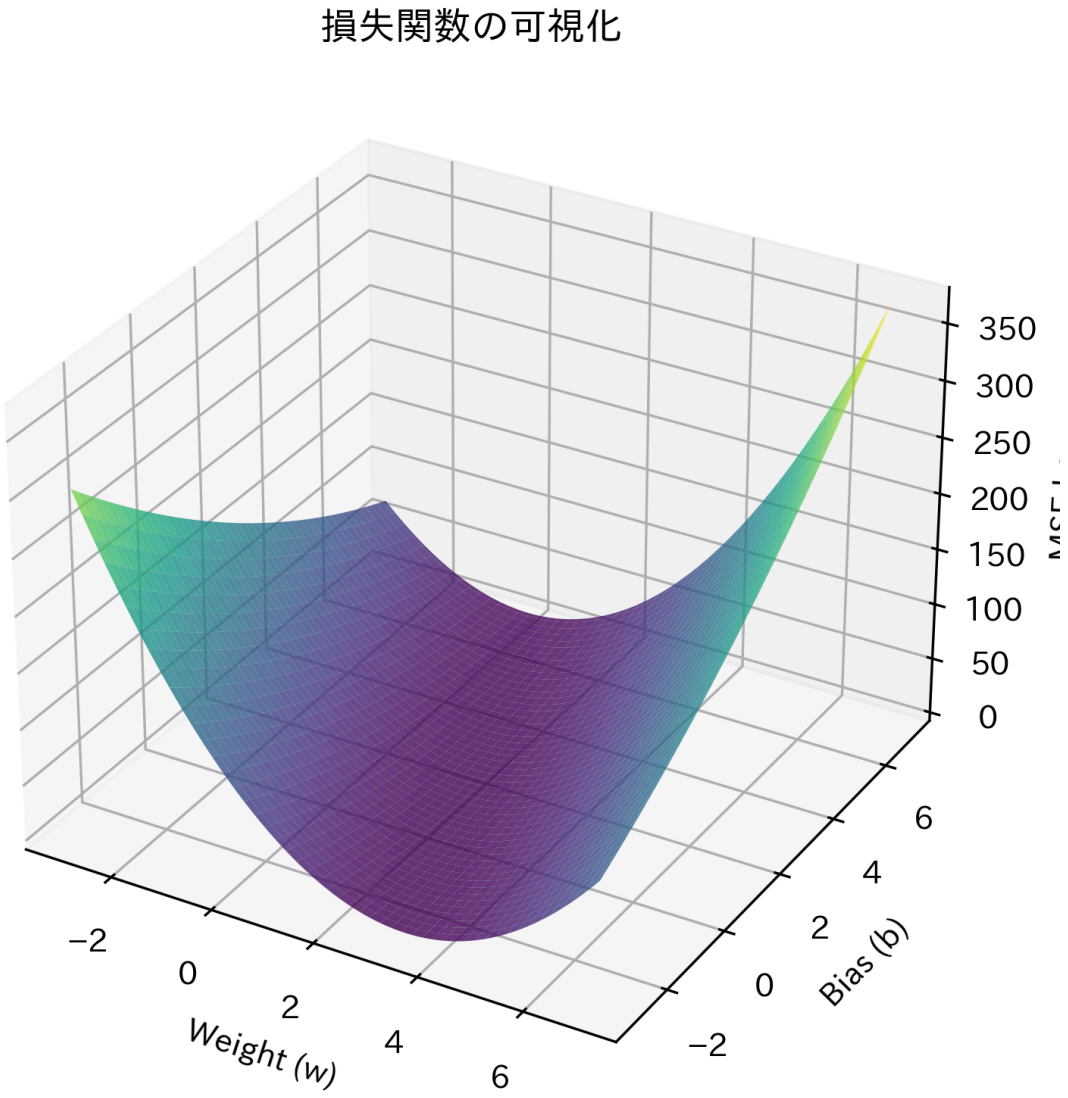
アプローチ:

1. 解析的に解く（上の式は w , b についての二次関数！）
2. 数値的に解く（勾配降下法など） ← こちらを学ぶ

なぜ勾配降下法なのか

- 今回はモデルとして一次関数 $y = wx + b$ を使っているが、実際はそうとは限らない。
- モデルの選択に（なるべく）よらずに使える手法だとありがたい

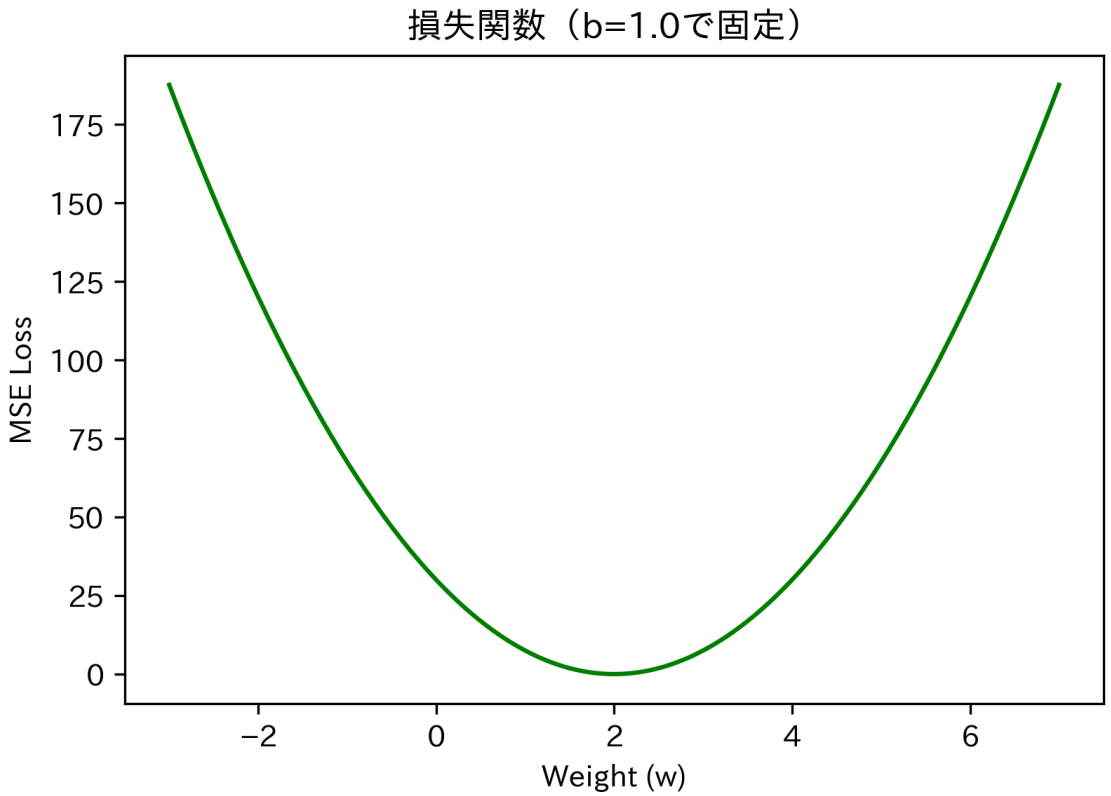
損失関数の可視化



w と b を動かしたときに損失関数がどう変化するかを可視化したもの。

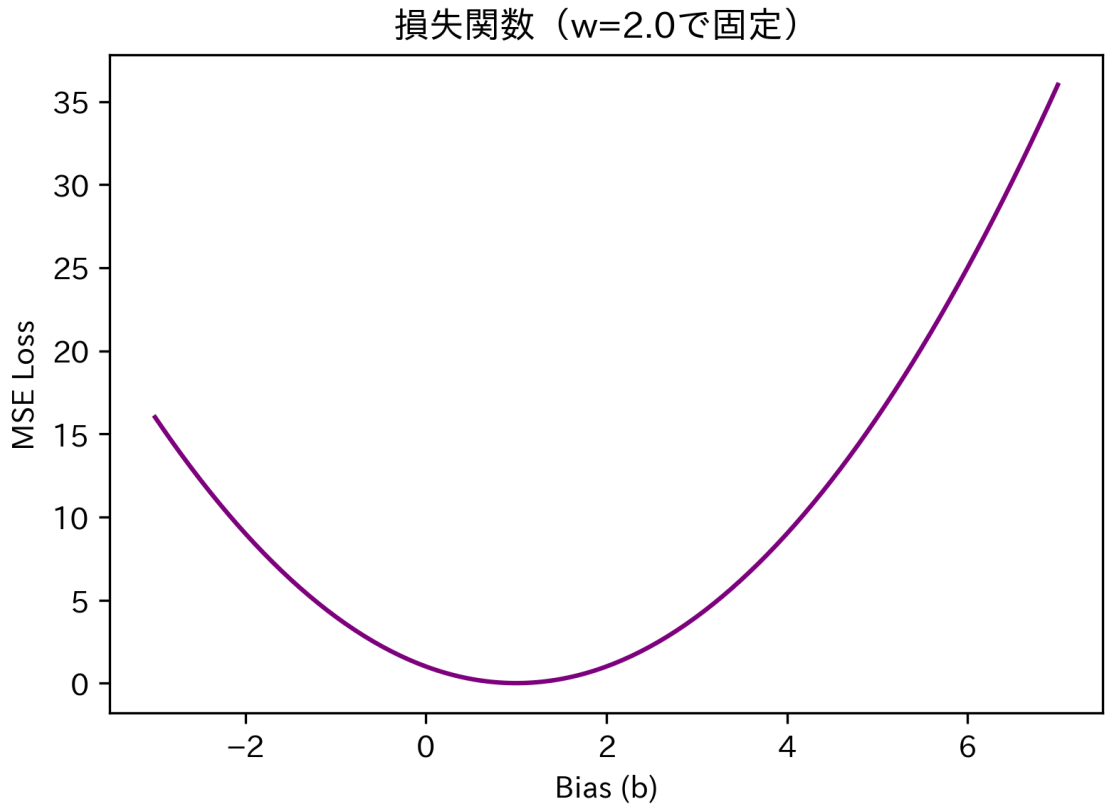
wに関する損失関数の変化

(bを固定)



bに関する損失関数の変化

(wを固定)



勾配降下法の直感

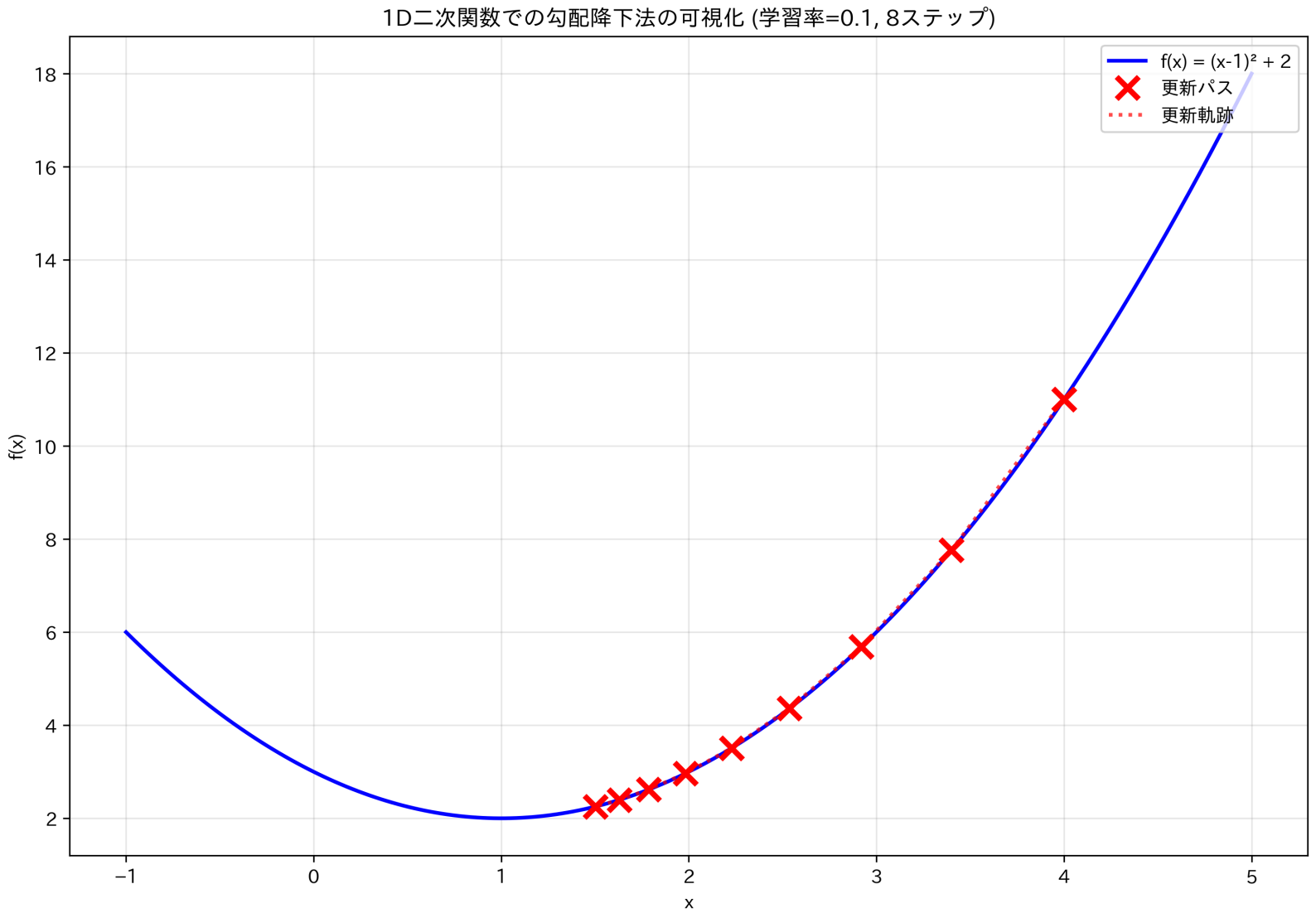
まず1次元（w のみ）で考えてみましょう。

イメージ:

- 現在位置から一番急な下り坂の方向に進む
- 少しずつ移動して、最終的に底（最小値）に到達

数学的には: 関数の微分（勾配）を使って最小値を探す

イメージ


1D二次関数での勾配降下法の可視化 (学習率=0.1, 8ステップ)
f(x) = (x-1)² + 2
更新パス
更新軌跡
f(x)
x
18
16
14
12
10
8
6
4
2
−1
0
1
2
3
4
5

勾配とは？

1次元の場合:

- 勾配 = 微分 = 関数の傾き
- $\frac{dL}{dw}$: w を少し変化させたときの L の変化率

勾配の符号:

- 正 → 右に進むと関数が増加 → 左に移動すべき
- 負 → 右に進むと関数が減少 → 右に移動すべき

移動方向: 勾配の**逆方向**に移動する

学習率 (Learning Rate) α

学習率 α: パラメータ更新の「**歩幅**」を決める重要なハイパーパラメータ
（**ハイパーパラメータ**：学習を始める前に設定するパラメータ）

α が大きすぎる場合:

- 歩幅が大きすぎて最小値を飛び越える
- 振動したり発散したりする

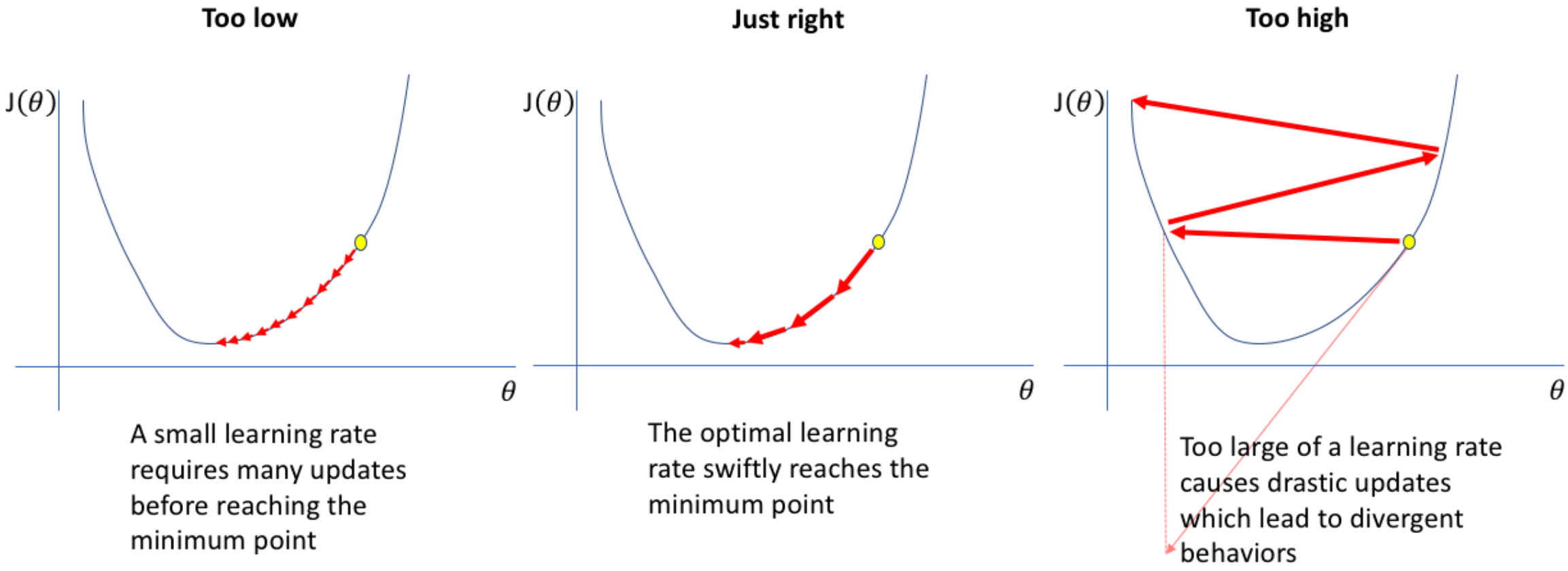
α が小さすぎる場合:

- 収束が非常に遅い
- 計算時間がかかりすぎる

適切な α:

- 安定して最小値に収束
- 適度な速度で学習が進む

典型的な値: 0.01, 0.001, 0.0001



https://www.jeremyjordan.me/nn-learning-rate/

1次元の勾配降下法アルゴリズム

ステップ1: パラメータを初期化

- w_0 を適当な値に設定

ステップ2: 勾配を計算

- $\frac{dL}{dw}$ を計算

ステップ3: パラメータを更新

- $w \leftarrow w - \alpha \frac{dL}{dw}$

ステップ4: 収束するまでステップ2-3を繰り返す

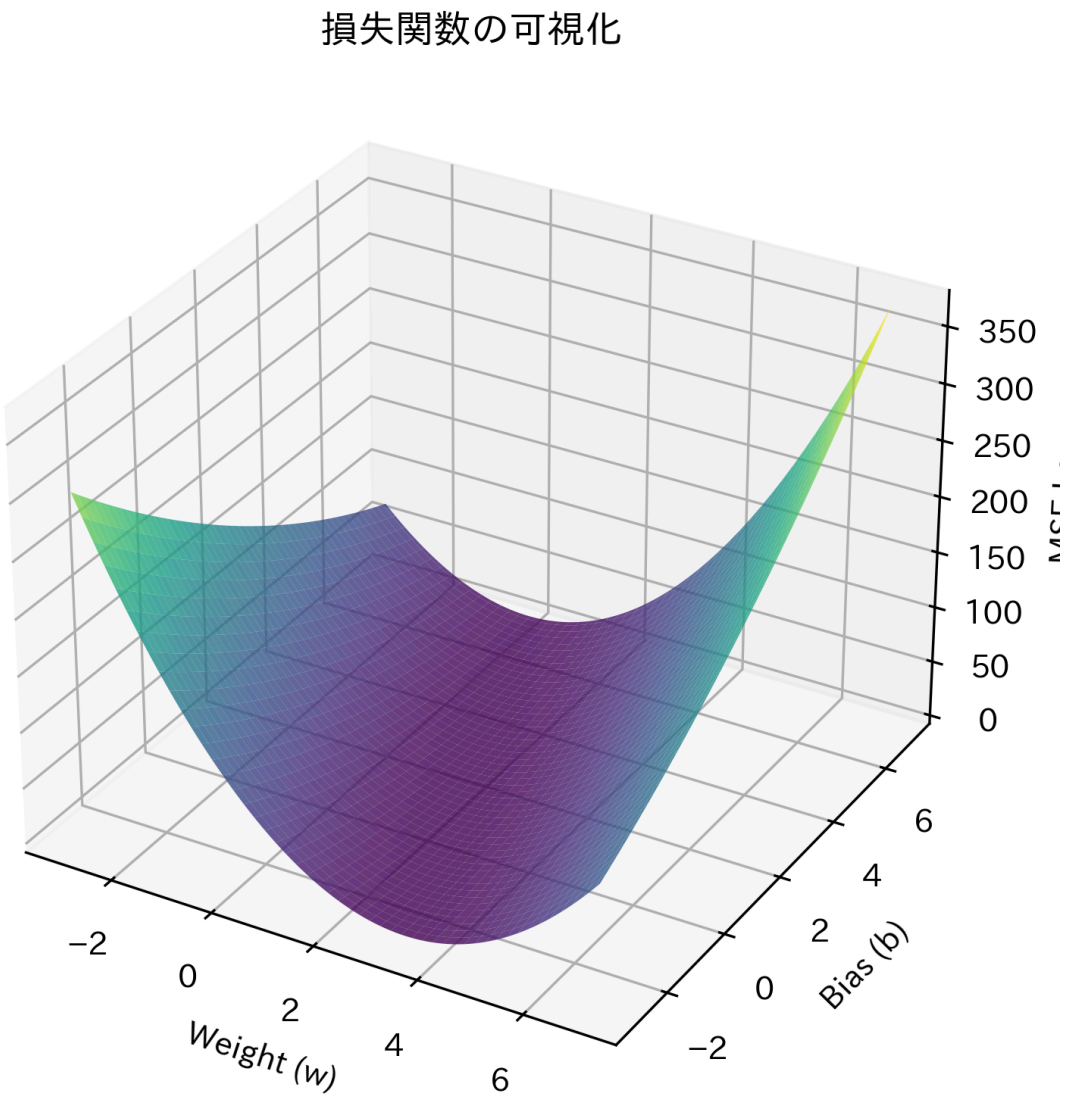
多次元への拡張：偏微分

損失関数 $L(w, b)$ のように変数が複数ある場合、各変数について個別に傾きを考えます。これが**偏微分**です。

- $\frac{\partial L}{\partial w}$: b を固定して w を少し動かしたときの L の変化率
- $\frac{\partial L}{\partial b}$: w を固定して b を少し動かしたときの L の変化率

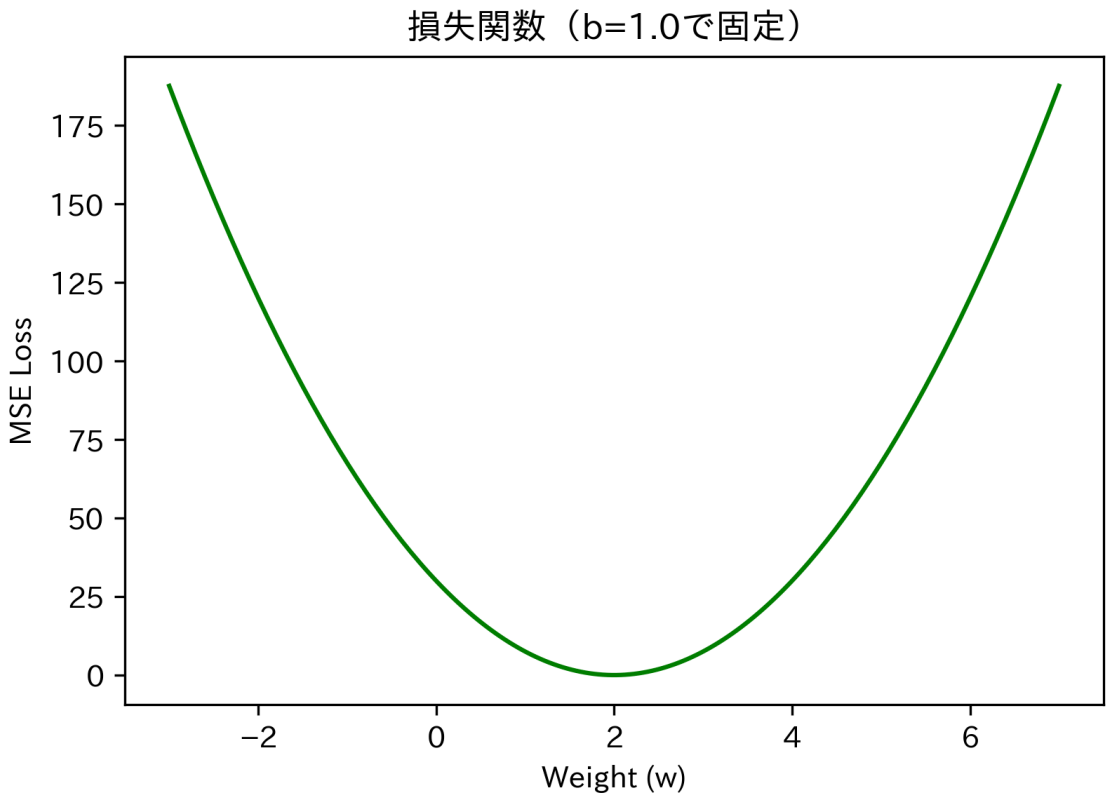
直感的なイメージ:
3Dのグラフを、ある軸に平行な平面で「スライス」して、その断面の曲線の傾きを求める操作です。

(再掲)三次元の損失平面

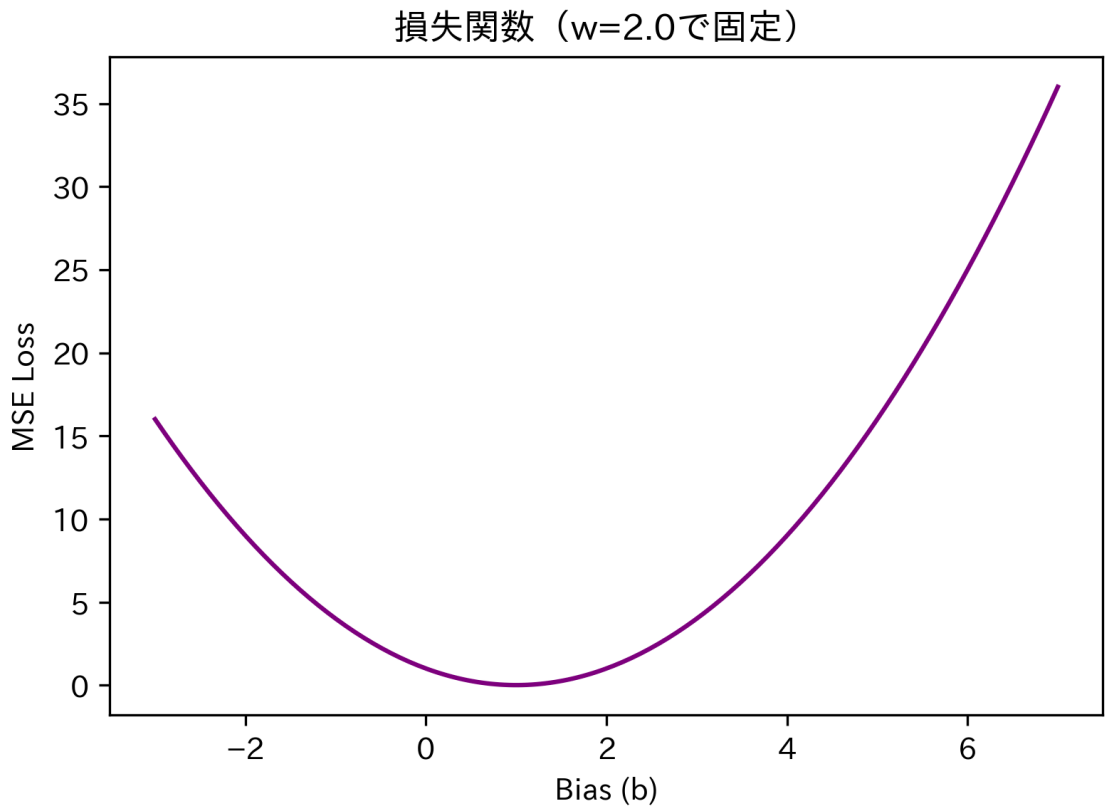


w と b を色々動かしたとき、損失関数 $L(w, b)$ の値はこのように変化するのでした。

(bを固定)


損失関数（b=1.0で固定）
MSE Loss
175
150
125
100
75
50
25
0
−2
0
2
4
6
Weight (w)

(wを固定)


損失関数（w=2.0で固定）
MSE Loss
35
30
25
20
15
10
5
0
−2
0
2
4
6
Bias (b)

偏微分の計算

偏微分には、連鎖率という合成関数の微分の多変数バージョンがあります。

$$L(w, b) = L(z(w, b))$$

のように、w と b　が　z という関数を通じて影響しているとき、

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial w}$$
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial b}$$

と計算できます。

勾配降下法のアルゴリズム（二次元）

ステップ1: パラメータを初期化（スタート地点の設定）

- w_0, b_0 を適当な値に設定

ステップ2: 勾配を計算

- $\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b}$ を計算

ステップ3: パラメータを更新

- $w \leftarrow w - \alpha \frac{\partial L}{\partial w}$
- $b \leftarrow b - \alpha \frac{\partial L}{\partial b}$

ステップ4: 収束するまでステップ2-3を繰り返す

平均二乗誤差の勾配計算

$$L(w, b) = \frac{1}{N} \sum_{i=1}^{N} (y_i - (wx_i + b))^2$$

w に関する偏微分:

$$\frac{\partial L}{\partial w} = \frac{1}{N} \sum_{i=1}^{N} 2(y_i - (wx_i + b)) \cdot (-x_i) = -\frac{2}{N} \sum_{i=1}^{N} x_i (y_i - (wx_i + b))$$

b に関する偏微分:

$$\frac{\partial L}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} 2(y_i - (wx_i + b)) \cdot (-1) = -\frac{2}{N} \sum_{i=1}^{N} (y_i - (wx_i + b))$$

勾配降下法の更新式

パラメータ更新:

$$w_{new} = w_{old} - \alpha \left(-\frac{2}{N} \sum_{i=1}^{N} x_i (y_i - (w_{old} x_i + b_{old})) \right)$$

$$b_{new} = b_{old} - \alpha \left(-\frac{2}{N} \sum_{i=1}^{N} (y_i - (w_{old} x_i + b_{old})) \right)$$

これを繰り返すことで、最適な w と b に近づいていく！

Pythonによる実装

実装の前に：Numpyライブラリ

- **配列 (Array)**: 同じ型のデータをまとめて扱うことができる
- **ベクトル化 (Vectorization)**: forループを使わずに配列全体の計算を一度に実行できるため、コードが簡潔になり、実行速度も速い

```
import numpy as np

# Numpy配列の作成
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# 配列の要素ごとの計算 (ベクトル化)
c = a + b  # -> array([5, 7, 9])
d = a * 2  # -> array([2, 4, 6])
```

Numpyを使うことで、数式に近い形でコードを書くことができます。

Numpy配列の操作例

Numpyの便利な関数を見てみましょう。

```
import numpy as np

X = np.array([1, 2, 3, 4])
y = np.array([3, 5, 7, 9])
y_pred = np.array([2, 4, 6, 8]) # 仮の予測値

# 誤差 (y - y_pred)
error = y - y_pred # -> array([1, 1, 1, 1])

# 要素ごとの積 (X * error)
weighted_error = X * error # -> array([1, 2, 3, 4])
```

Numpy配列の操作例（続き）

```
# 平均値 (mean)
# (1 + 2 + 3 + 4) / 4 = 2.5
mean_error = np.mean(weighted_error) # -> 2.5

# 合計値 (sum)
# 1 + 2 + 3 + 4 = 10
sum_error = np.sum(weighted_error) # -> 10
```

`np.mean()` は `np.sum()` を要素数で割ったものと同じです。勾配の式 $\frac{1}{N}\sum(...)$ の計算に便利です。

実装とアルゴリズムの対応

先ほど学んだ勾配降下法のアルゴリズムを、コードに落とし込んでいきましょう。

アルゴリズムのステップ	Pythonコードでの対応
1. パラメータを初期化	`w = 0.0`, `b = 0.0`
2. 勾配を計算	`grad_w = ...`, `grad_b = ...`
3. パラメータを更新	`w = w - alpha * grad_w`
4. 繰り返す	`for i in range(epochs):`

ステップ1 & 2: データ準備とパラメータ初期化

まずは、学習に使うデータと、モデルのパラメータを用意します。

```
import numpy as np

# 1. データの準備 (ダミーデータ、自由に変更してみてください)
X = np.array([1, 2, 3, 4])
y = np.array([3, 5, 7, 9])

# 2. パラメータの初期化
w = 0.0
b = 0.0

# ハイパーパラメータの設定
learning_rate = 0.01 # 学習率α
epochs = 1000        # 繰り返し回数
```

ステップ3: 学習ループ

ここが勾配降下法の中核です。計算した勾配を使ってパラメータを繰り返し更新します。

```
# 3. 学習ループ
for i in range(epochs):
    # a. 予測: y = wx + b
    y_pred = w * X + b

    # b. 勾配の計算 (前のスライドの数式をコード化)
    # dL/dw = -2/N * sum(X * (y - y_pred))
    # dL/db = -2/N * sum(y - y_pred)
    grad_w = -2 * np.mean(X * (y - y_pred))
    grad_b = -2 * np.mean(y - y_pred)
```

ステップ3: 学習ループ（続き）

```
    # c. パラメータの更新（前のスライドの更新式をコード化）
    w = w - learning_rate * grad_w
    b = b - learning_rate * grad_b

    # 学習経過の表示
    if i % 100 == 0:
        loss = np.mean((y - y_pred)**2)
        print(f"Epoch {i}: Loss = {loss:.4f}, w = {w:.4f}, b = {b:.4f}")
```

学習結果の確認

学習ループが終わると、`w` と `b` は最適値に近づいているはずです。

```
# ... (学習ループの後) ...

print(f"\n学習後のパラメータ:")
print(f"w = {w:.4f}") # -> 2.0 に近い値
print(f"b = {b:.4f}") # -> 1.0 に近い値
```

このダミーデータは $y = 2x + 1$ という関係なので、`w` が2、`b` が1に近づけば学習成功です。

線形回帰の発展形

パラメーターを増やす

今までは $f(x) = wx + b$ の形の線形モデルを使ってきましたが、学習に使ってきた手法はそれに限られたものではありません。
例えば、

複雑な関数を使う

- $f(x) = w_1 x + w_2 x^2 + b$
- $f(x) = w_1 x + w_2 \sin(x) + b$
- $f(x) = w_1 x + w_2 \log(x) + b$
- $f(x) = w_1 x + w_2 \exp(x) + b$

入力変数を増やす

- $f(x_1, x_2) = w_1 x_1 + w_2 x_2 + b$
- $f(x_1, x_2) = w_1 x_1 + w_2 x_2 + w_3 x_1 x_2 +$

などでも、損失関数のパラメーターによる微分さえ求められれば、勾配降下法を使って学習できます。

線形分類

回帰 vs 分類:

- **回帰**: 連続値を予測（例：温度、価格）
- **分類**: カテゴリを予測（例：スパム/非スパム、手書きの数字が0~9のうちどれか）

分類する対象が二つの時を**二値分類**､そうでない場合を**多クラス分類**と呼ぶ。

確率を出力する

分類問題では、モデルに入力がどのクラスに属するかの**確率**を出力させたい。

- 例えば、入力が「スパム」である確率を0.8、そうでない確率を0.2と予測する

つまり二値分類（分類するクラスが二つ）の時、

$f : \mathbb{R} \to \mathbb{R}$ だったのが $f : \mathbb{R} \to [0, 1]$ になります。

線形分類モデル

線形分類モデル（ロジスティック回帰）:

$$z = wx + b$$

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

途中までは線形回帰と同じだが、最後にシグモイド関数(次で解説)を付けくわえて、0と1の間に収める。シグモイド関数に通す前の値 (z) を**ロジット**と呼ぶ。

全て合わせると、

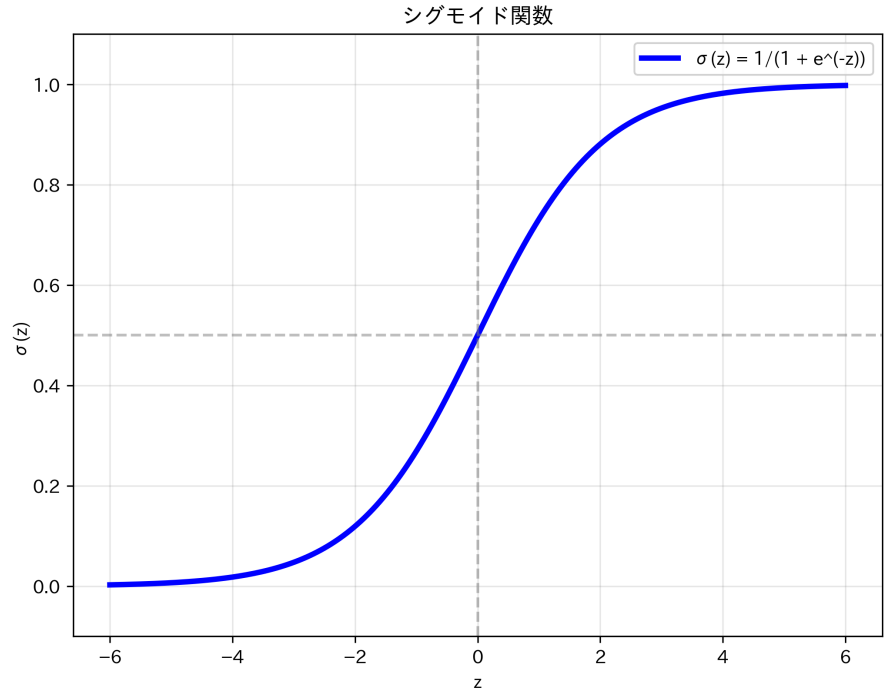
$$y = f(x) = \sigma(wx + b) = \frac{1}{1 + e^{-(wx+b)}}$$

シグモイド関数

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

特徴:

- 出力が常に0と1の間
- $z = 0$ のとき $\sigma(z) = 0.5$
- z が大きいほど1に近づく
- z が小さいほど0に近づく



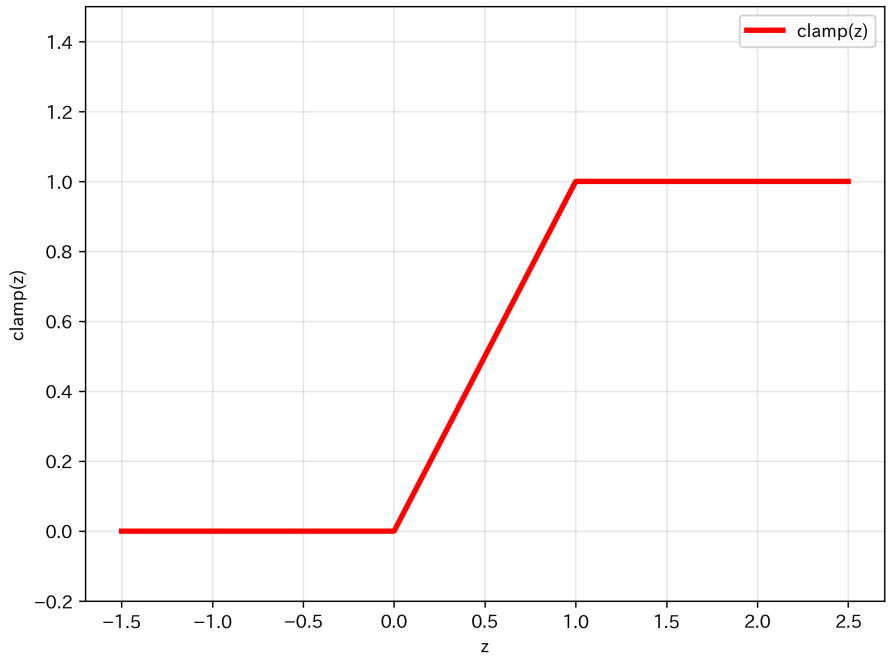
判定: $p \geq 0.5$ ならクラス1、$p < 0.5$ ならクラス0

シグモイド関数を使う嬉しさ

例えば、

$$clamp(z) = \begin{cases} 0 & (z < 0) \\ z & (0 \leq z < 1) \\ 1 & (z \geq 1) \end{cases}$$

という関数を使っても出力は0と1の間に収まりますね



ところが、この関数は $z \leq 0$ のとき 0 に、$z \geq 1$ のとき微分係数が0 になってしまう（w, b を少し変化させても、最終的な出力は一切変わらない。）

(連鎖率から)、モデルの予測が0または1の時はパラメーターの更新が一切されず、学習も進まない

分類の損失関数：交差エントロピー

二値分類の交差エントロピー損失（Binary Cross Entropy）：

$$L = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(p_i) + (1-y_i)\log(1-p_i)]$$

- $y_i \in \{0, 1\}$: 実際のラベル
- $p_i \in [0, 1]$: クラス1の予測確率

交差エントロピー損失の直感的理解

まずは、これが損失関数としてうまく機能することを確認しましょう。

- **正解がクラス1 ($y = 1$) の場合**: $L = -\log(p)$
  - $p = 1$ (完璧な予測) → $L = 0$ (損失なし)
  - $p = 0.5$ (曖昧な予測) → $L = 0.69$
  - $p = 0$ (完全に間違い) → $L = \infty$ (巨大な損失)
- **正解がクラス0 ($y = 0$) の場合**: $L = -\log(1 - p)$
  - $p = 0$ (完璧な予測) → $L = 0$
  - $p = 1$ (完全に間違い) → $L = \infty$

特徴: 確信を持って間違えると非常に大きなペナルティ！

交差エントロピーの導出（少し難しめ）

最尤推定(さいゆうすいてい)

- 手元にあるデータが得られる確率（尤度）が、最も大きくなるようにモデルのパラメータを決める

正解がクラス1のサンプルがあったとして、あるパラメーター (w_0, b_0) によるモデルが確率0.2を出力したとします。
この時、そのサンプルは確率0.2でクラス1になるはずだった、と考えて、もっとも（尤も）らしさが0.2であるとします。

厳密にいえば上の例の0.2という数字は確率ではなく尤度という概念です

尤度によるパラメーターの比較

同じサンプルに対して、別のパラメーター (w_1, b_1) によるモデルが確率0.8を出力したとします。この時、
(w_0, b_0) ：このパラメーターからサンプルが生成される確率は0.2
よりも、
(w_1, b_1) ：このパラメーターからサンプルが生成される確率は0.8
のほうが”ありえる”ので、より妥当なパラメーターであると言えます。

データセット全体では同時確率を考えることになるので、

$$P(y,p) = \prod_{i=1}^{N} \begin{cases} p_i & (y_i = 1) \\ 1 - p_i & (y_i = 0) \end{cases}$$

となります。
積の形は扱いにくいので対数をかけて、機械学習では損失の最小化を目的とするのでマイナスをかけると、

$$L = -\log(P(y,p)) = -\sum_{i=1}^{N} \begin{cases} \log(p_i) & (y_i = 1) \\ \log(1 - p_i) & (y_i = 0) \end{cases}$$

これが二値分類の交差エントロピー損失です。場合分けを実現するために正解ラベルをかけると全く同じになりますね。

交差エントロピー損失の勾配

嬉しい性質: 交差エントロピー損失の**ロジットによる**偏微分は非常にシンプルです。

$$\frac{\partial L}{\partial z} = y_i - p_i$$

- 実際のラベルと予測確率の差がそのまま勾配になる
- 予測が正解に近いほど勾配は小さくなる
- 予測が大きく外れているほど勾配は大きくなる

この結果と合成関数の微分を使うと、**勾配降下法**も同様に適用できます！

多クラスの場合

多クラス分類の場合、それぞれのクラスについて確率を出力したいので、

$f : \mathbb{R} \to [0, 1]^K$ という形になります。

正解ラベルもこの形に合わせて、正解のクラスが1、それ以外が0のベクトルとします。

多クラス分類: K 個のクラスがある場合、ソフトマックス関数を使います。

$$p_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}$$

一見複雑ですが、すべてのロジットを指数関数に通した後（正の値になる）に、合計が1になるように正規化しているだけです。

多クラスの交差エントロピー損失

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} y_{ik}\log(p_{ik})$$

クラスについても和をとっているためシグマが二重になっていますが、基本的な考え方は二値分類と同じです。

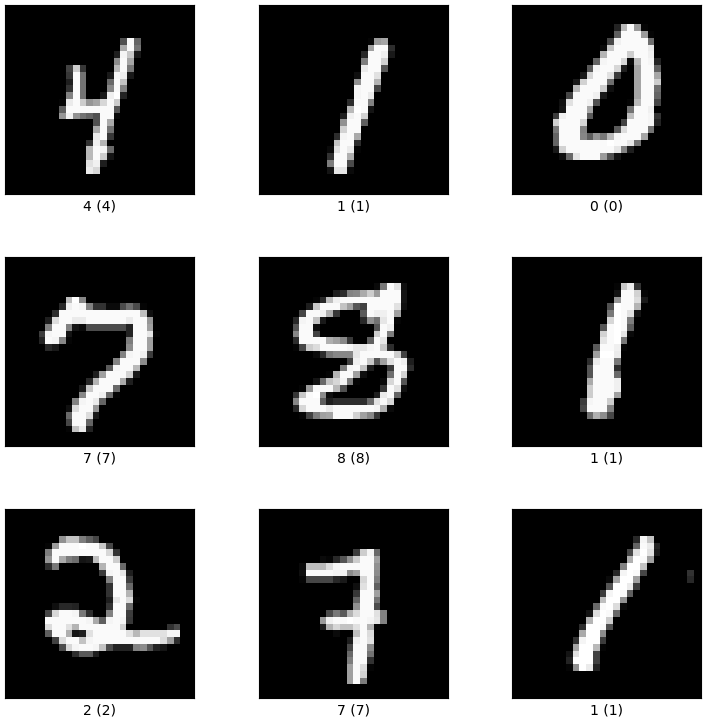
y_{ik} がかけられているので、正解のクラスについてのみ損失が計算されるようになっています。
つまり、

$$L = \begin{cases} -\log(p_{ik}) & (y_{ik} = 1) \\ 0 & (y_{ik} = 0) \end{cases}$$

線形モデルの限界

実際のデータでの性能比較: MNIST手書き数字認識

MNIST データセット: 28×28ピクセルの手書き数字(0-9)を分類する問題



線形モデル vs ニューラルネットワーク

線形モデル (ロジスティック回帰)

- 入力は784個のベクトル (28×28を1次元につぶす)
- **精度: 約92-94%**

ニューラルネットワーク

- ピクセル間の関係性を学習
- 隠れ層で複雑なパターンを発見
- **精度: 約97-98%**

同じデータ、同じ学習手法（勾配降下法）を使っても、**4-6%の性能差**が生まれます。

なぜこれほど性能差が生まれるのか？

線形モデルの課題:

- 学習できる関数の形が単純
- 複雑にしたければ人がうまく設計しないといけない
- 高次元データでは特徴量の組み合わせが膨大になる

ニューラルネットワーク（MLP）の利点:

- 複数の非線形変換を組み合わせて複雑なパターンを学習
- 隠れ層で有用な特徴表現を自動で発見
- 人間の特徴量設計が（そこまで）いらない

次回: ニューラルネットワークがどのようにこれを実現するかを学びます！

まとめ

1. **損失関数**: 予測と実際の値の「近さ」を数値化
2. **勾配降下法**: 損失を最小化する汎用的な最適化手法
3. **学習率**: 収束の速度と安定性を決める重要なパラメータ
4. **分類への応用**: シグモイド関数と交差エントロピー損失
5. **線形モデルの限界**: 複雑なパターンは表現できない

次回: 線形回帰よりも格段に表現力が高いニューラルネットワークについて学びます！