機械学習講習会

第三回：ニューラルネットワークの基礎

2025/07/01

@Kobakos32

前回の復習

1. **勾配降下法**: 損失関数の勾配（傾き）を頼りに、パラメータを少しずつ更新して損失を最小化する汎用的な方法を学んだ
2. **線形モデルの発展形**: 線形モデルのパラメタを増やしたり、損失関数としてクロスエントロピー誤差を使うことで分類問題にも対応できることを学んだ
3. **線形モデルの限界**: MNISTの手書き数字認識の例で、線形モデルでは複雑なパターンを捉えきれず、性能に限界があることを見た

課題: どうすれば、より複雑な関係性を学習できるのか？

本日のゴール

1. ニューラルネットワークの基本構造を理解する。
2. 構成要素である**線形層**と**活性化関数**の役割を説明できるようになる。
3. 簡単なニューラルネットワークの **計算（順伝播）** を手で追えるようになる。
4. ニューラルネットワークが持つ **表現力（万能近似定理）** の雰囲気を掴む。

もくじ

1. ニューラルネットワークの直感的イメージ
2. 構成要素(1)：線形層
3. 構成要素(2)：活性化関数
4. その他の構成要素
5. ニューラルネットワークの表現力
6. まとめ

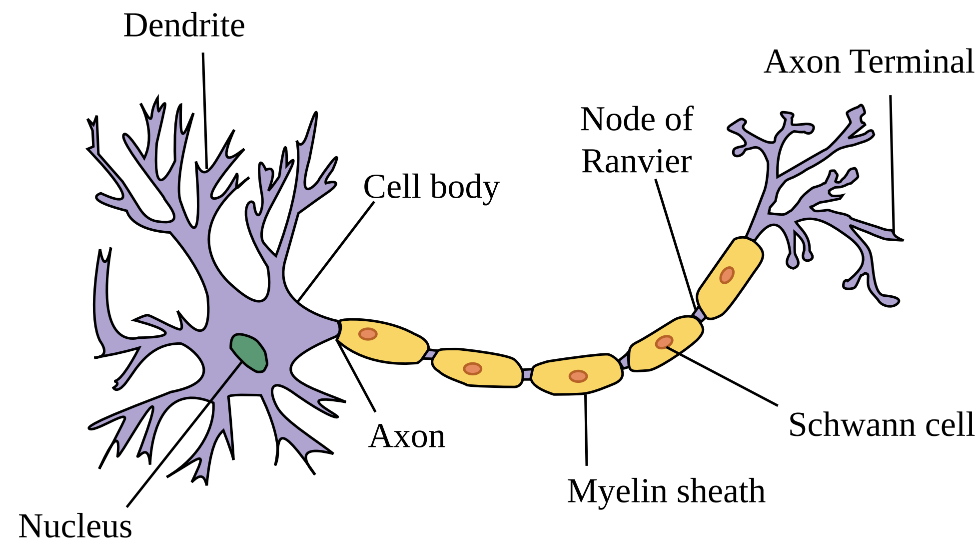
1. ニューラルネットワークの直感的イメージ

生体模倣としてのニューラルネットワーク

生物のニューロン

- **樹状突起**: 他のニューロンから信号を受け取る
- **細胞体**: 信号を統合・処理する
- **軸索**: 処理結果を他のニューロンへ伝達する

信号の合計がある閾値を超えると「発火」し、次のニューロンへ信号が伝わります。

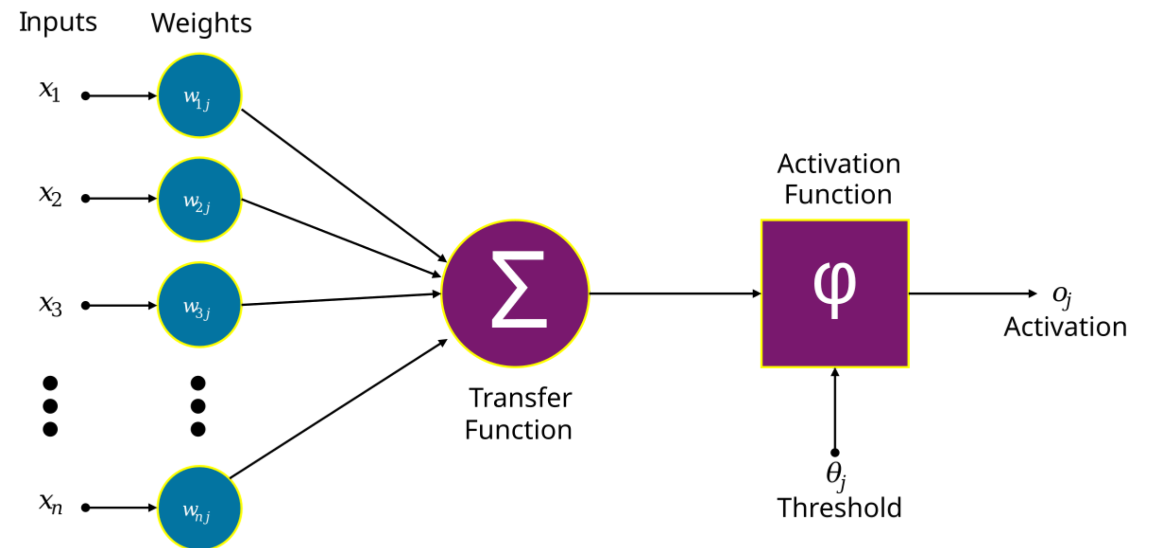


Wikimedia Commons

人工ニューロン

この生物の仕組みを数理モデルで模倣します。

- **入力** $(x_1, x_2, ...)$: 他のニューロンからの信号
- **重み** $(w_1, w_2, ...)$: 信号の重要度（シナプスの結合強度）
- **バイアス** (b): 発火のしやすさ（閾値）
- **活性化関数** (ϕ): 信号の合計を処理し、発火するかどうかを決める



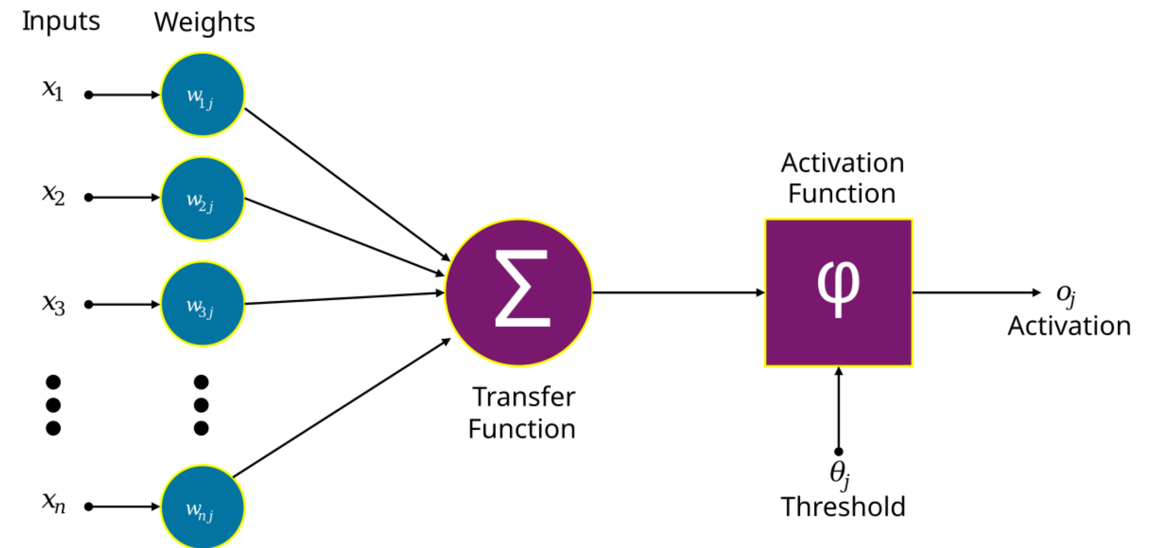
人工ニューロンのことを**パーセプトロン**と呼ぶこともあります。

Wikimedia Commons

人工ニューロン

人工ニューロンで起こる計算は以下の通りです

1. 入力に対して対応する重みをかける
2. 重みが付いた入力の合計を求める
3. バイアスを加える
4. 活性化関数を適用して出力を決定する

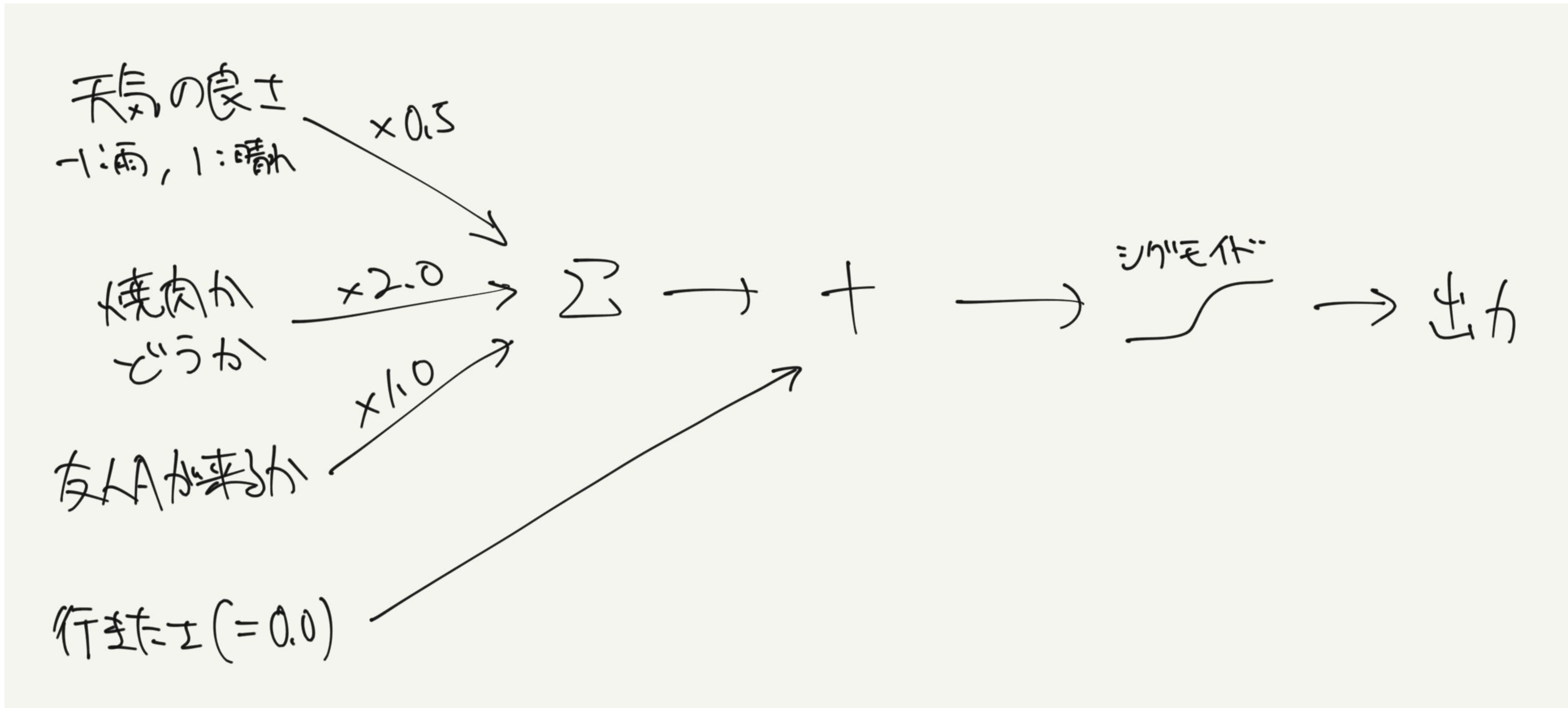


Wikimedia Commons

もう少し例を

人工ニューロンを使って、飲み会に参加するかどうかを決める例を考えます。

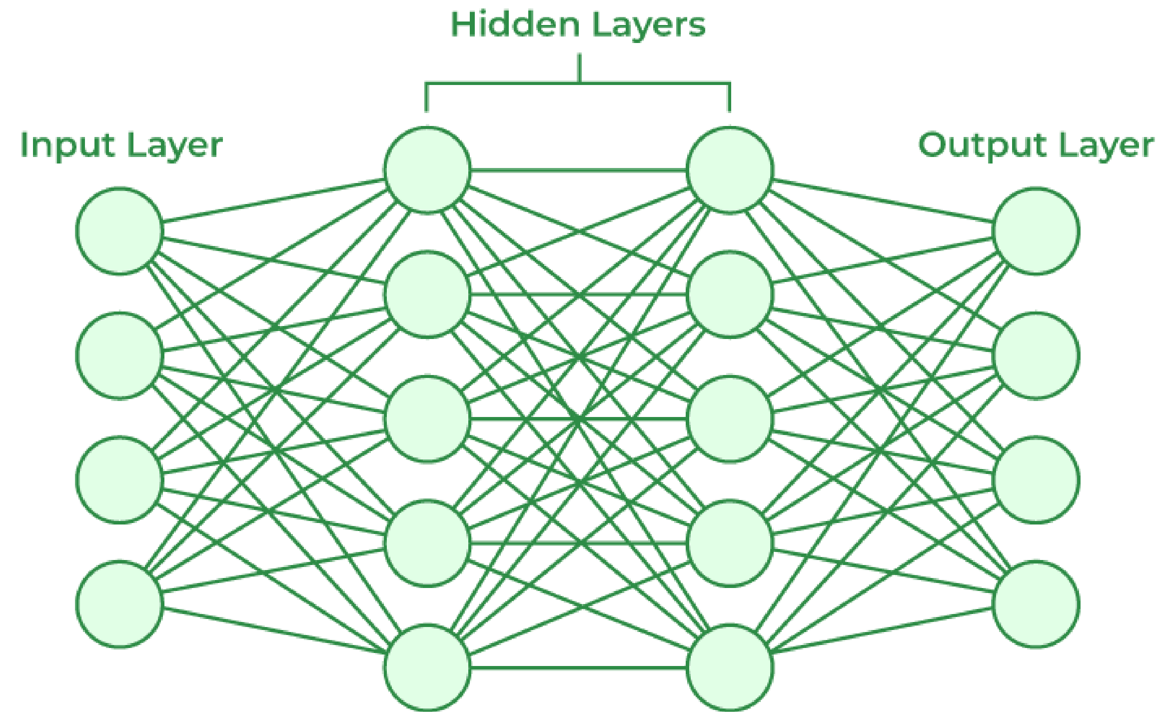
- **入力**: 天気、店の種類、参加する人
- **出力**: 参加するかどうかの確率
- **重み**: 各入力の重要度と関係（例: 天気が晴れなら参加しやすい）
- **バイアス**: そもそも参加する気があるかどうか（例: いつも参加したいと大きい）
- **活性化関数**: 確率として出力するための関数


天気の良さ
-1:雨, 1:晴れ
×0.5
焼肉か
どうか
×2.0
友人Aが来るか
×1.0
行きたさ(=0.0)
Σ
+
シグモイド
出力

層を重ねて特徴を捉える

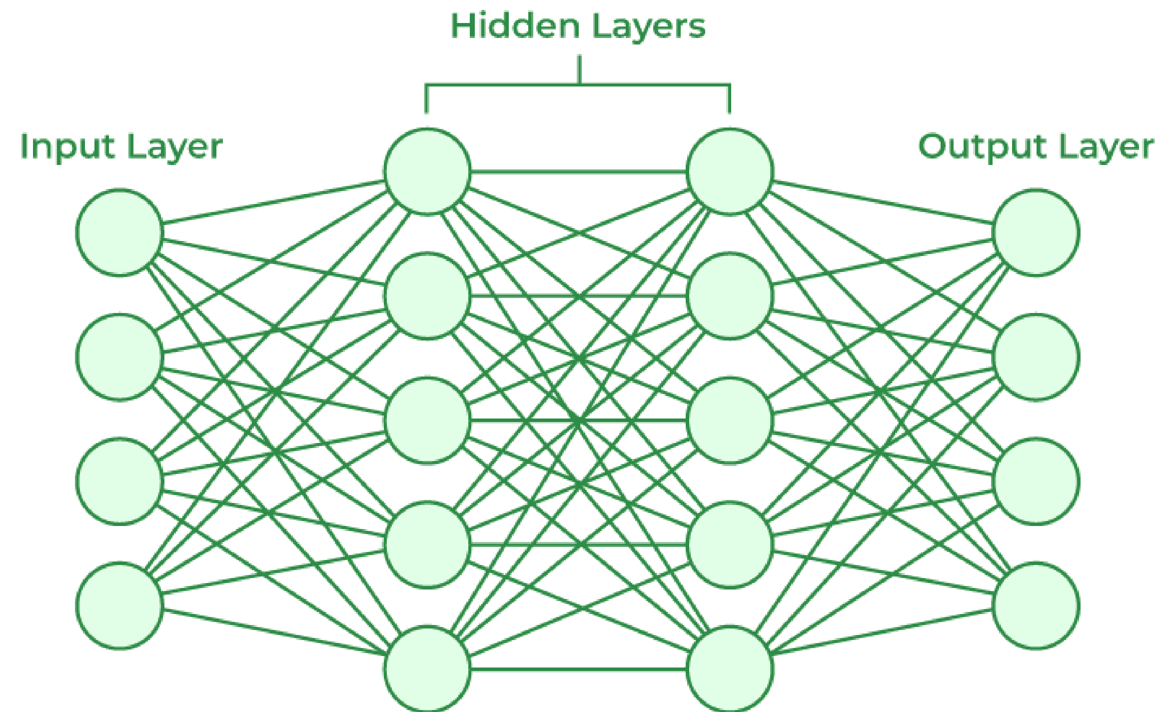
ニューラルネットワークは、この人工ニューロンを層状に多数重ねたものです。

アイデア: 入力から、より複雑で抽象的な特徴を段階的に学習する。



層を重ねて特徴を捉える

一番左の列を**入力層**、間の列を**隠れ層**、一番右の列を**出力層**と呼びます。
入力層は何もしない



2. 構成要素：線形層

複数の人工ニューロンを並べる

一つの人工ニューロンの場合は、

$$o = \phi(w_1x_1 + w_2x_2 + w_3x_3 + b)$$

となっていましたが、それが複数登場することになります

$$\begin{aligned} o_1 &= \phi(w_{11}x_1 + w_{12}x_2 + w_{13}x_3 + b_1) \\ o_2 &= \phi(w_{21}x_1 + w_{22}x_2 + w_{23}x_3 + b_2) \\ o_3 &= \phi(w_{31}x_1 + w_{32}x_2 + w_{33}x_3 + b_3) \end{aligned}$$

o_1, o_2, o_3 はそれぞれ異なるニューロンの出力、w_{ij}はニューロン i の入力 j に対する重み、b_i はニューロン i のバイアスです。

複数の人工ニューロンを並べる

実は、前スライドの積和演算の部分を取り出すと**行列積**の形で書けるんです

$$W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix} \quad x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} \quad o = \begin{pmatrix} o_1 \\ o_2 \\ o_3 \end{pmatrix}$$

と定義すると、

$$o = \phi(Wx + b)$$

(ϕ は要素ごとに活性化関数を適用することを意味します)

この積和演算の部分を取り出して、**線形層 (Linear Layer)** として扱います。

線形層 (Linear Layer)

- 別名: 全結合層 (Fully-Connected Layer), Dense層
- **役割**: 入力ベクトルを受け取り、重み行列 W とバイアスベクトル b を用いて線形変換を行う。

$$z = Wx + b$$

- x: 入力ベクトル (サイズ d_{in})
- W: 重み行列 (サイズ $d_{out} \times d_{in}$)
- b: バイアスベクトル (サイズ d_{out})
- z: 出力ベクトル (サイズ d_{out})

（今回は x は d_{in} 次元のデータ一つとしています。まとめて予測を計算するときも本質は同じです。）

厳密には線形変換ではなくAffine変換

線形層の計算：具体例

入力が3次元 ($d_{in} = 3$)、出力が2次元 ($d_{out} = 2$) の線形層を考えます。

- 入力: $x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 5 \\ 1 \end{pmatrix}$
- 重み: $W = \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{pmatrix} = \begin{pmatrix} 0.1 & 0.4 & 0.5 \\ 0.2 & 0.3 & 0.6 \end{pmatrix}$
- バイアス: $b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} 0.5 \\ -0.1 \end{pmatrix}$

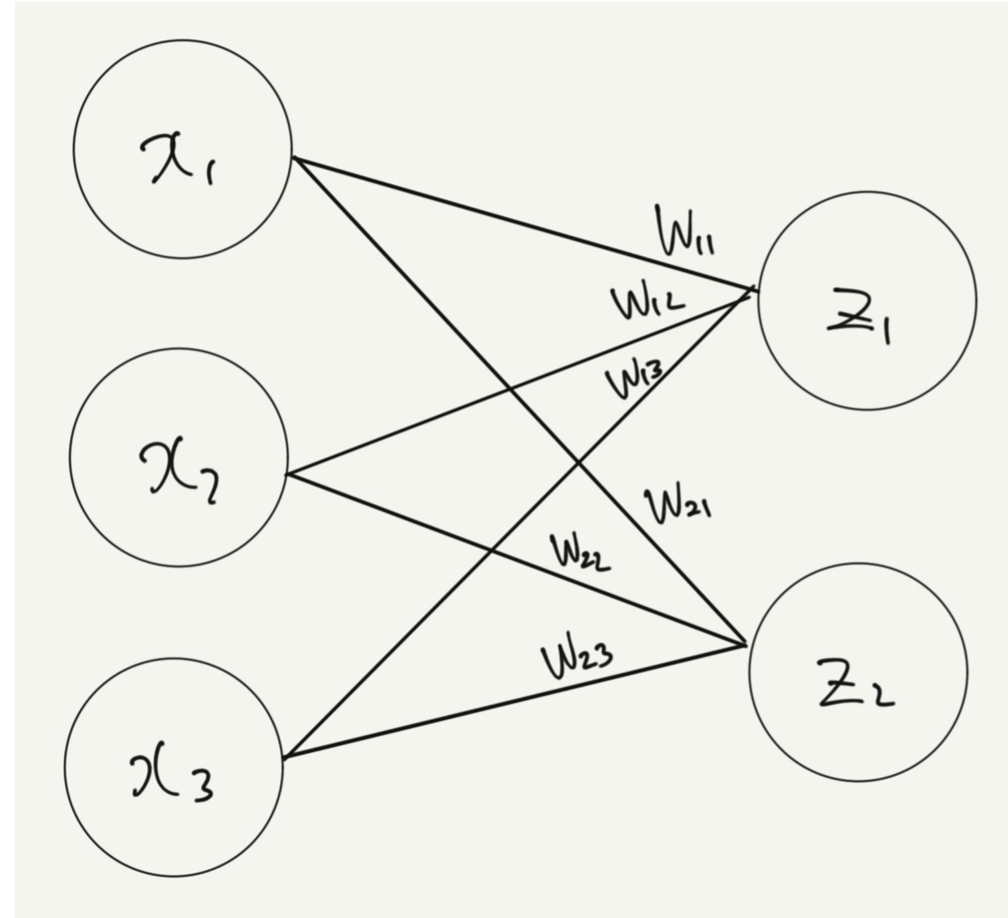
線形層の計算：具体例 (続き)

$$z = Wx + b$$

$$z_1 = (w_{11}x_1 + w_{12}x_2 + w_{13}x_3) + b_1$$
$$z_1 = (0.1 \cdot 2 + 0.4 \cdot 5 + 0.5 \cdot 1) + 0.5$$
$$z_1 = (0.2 + 2.0 + 0.5) + 0.5 = 3.2$$

$$z_2 = (w_{21}x_1 + w_{22}x_2 + w_{23}x_3) + b_2$$
$$z_2 = (0.2 \cdot 2 + 0.3 \cdot 5 + 0.6 \cdot 1) - 0.1$$
$$z_2 = (0.4 + 1.5 + 0.6) - 0.1 = 2.4$$

出力: $z = \begin{pmatrix} 3.2 \\ 2.4 \end{pmatrix}$



重みとバイアスの意味

- **重み (W)**: 入力特徴の**出力への寄与**を表す。
  - w_{ij} は、入力の j 番目の要素が、出力の i 番目の要素にどれだけ強く影響するかを示す。
  - 絶対値が大きいほど、強い関係性（正または負）がある。
- **バイアス (b)**: データの原点をずらす
  - 重み付き入力がなくても、ニューロンがどれだけ活性化しやすいか
  - 例えば、バイアスが大きいと、入力が0でも出力が大きくなる

3. 構成要素(2)：活性化関数

なぜ活性化関数が必要か？

もし線形層だけを重ねるとどうなるか？

- 1層目: $z_1 = W_1 x + b_1$
- 2層目: $z_2 = W_2 z_1 + b_2 = W_2(W_1 x + b_1) + b_2$

これを整理すると...
$z_2 = (W_2 W_1)x + (W_2 b_1 + b_2)$

$W' = W_2 W_1, b' = W_2 b_1 + b_2$ と置けば、$z_2 = W'x + b'$ となり、結局はただの1つの線形層と同じ。
線形変換を何度重ねても、表現力は線形のまま。

活性化関数 (Activation Function)

- **役割**: 線形層の出力に「非線形性」を導入する。これにより、ネットワークは線形モデルよりもはるかに複雑な関数を学習できるようになる。
- **アナロジー**: 生物のニューロンは、入力の合計がある閾値を超えたときに「発火」する。活性化関数はこのON/OFFのようなスイッチングの役割をモデル化している。

活性化関数を含めた一つの層の計算は

$$\text{Layer Output} = \text{Activation}(\text{Linear}(x)) = \phi(Wx + b)$$

とかけ、これを複数積み重ねた

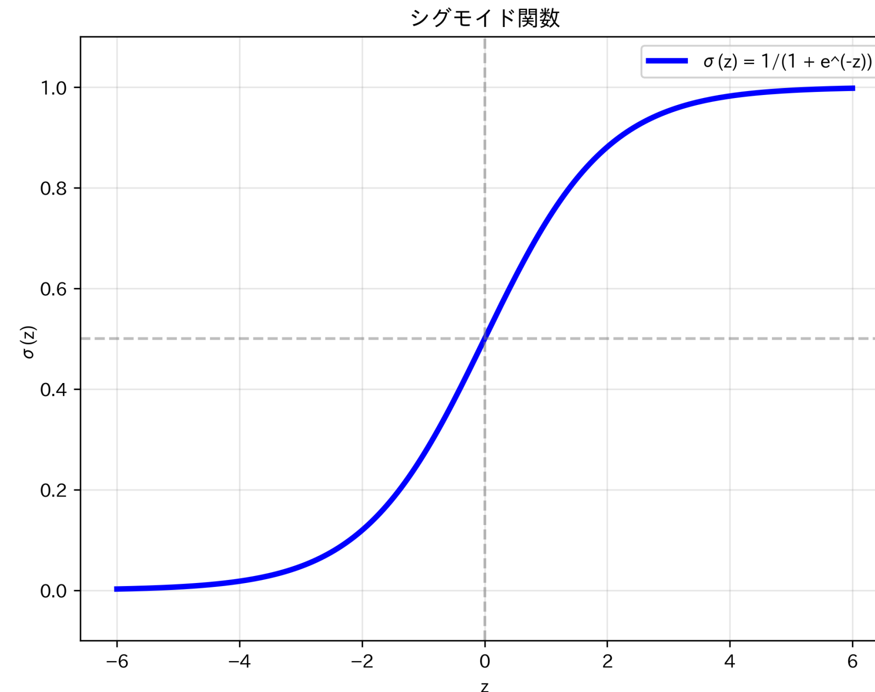
$$\text{Output} = \phi(W_n\phi(W_{n-1}\phi(\dots\phi(W_1x + b_1) + b_2)\dots) + b_n)$$

がニューラルネットワークの数式表現になる！！

代表的な活性化関数 (1): シグモイド関数

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

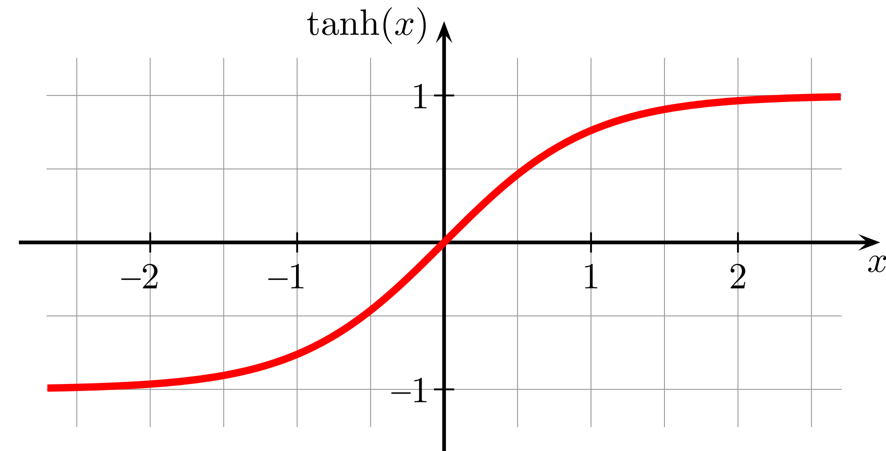
- **特徴:**
  - 出力を (0, 1) の範囲に押し込める。確率の出力などに便利。
- **問題点:**
  - 勾配消失問題：入力が0から離れると勾配がほぼ0になり、学習が進まなくなる。
  - 出力が常に正なので、値がドリフトしやすい



代表的な活性化関数 (2): Tanh (ハイパボリックタンジェント)

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

- **特徴**:
  - 出力を (-1, 1) の範囲に押し込める。
  - 出力が**ゼロ中心** (zero-centered) なので、シグモイドよりも学習が効率的に進むことが多い。
- **問題点**:
  - シグモイドと同様に、勾配消失問題は依然として存在する。

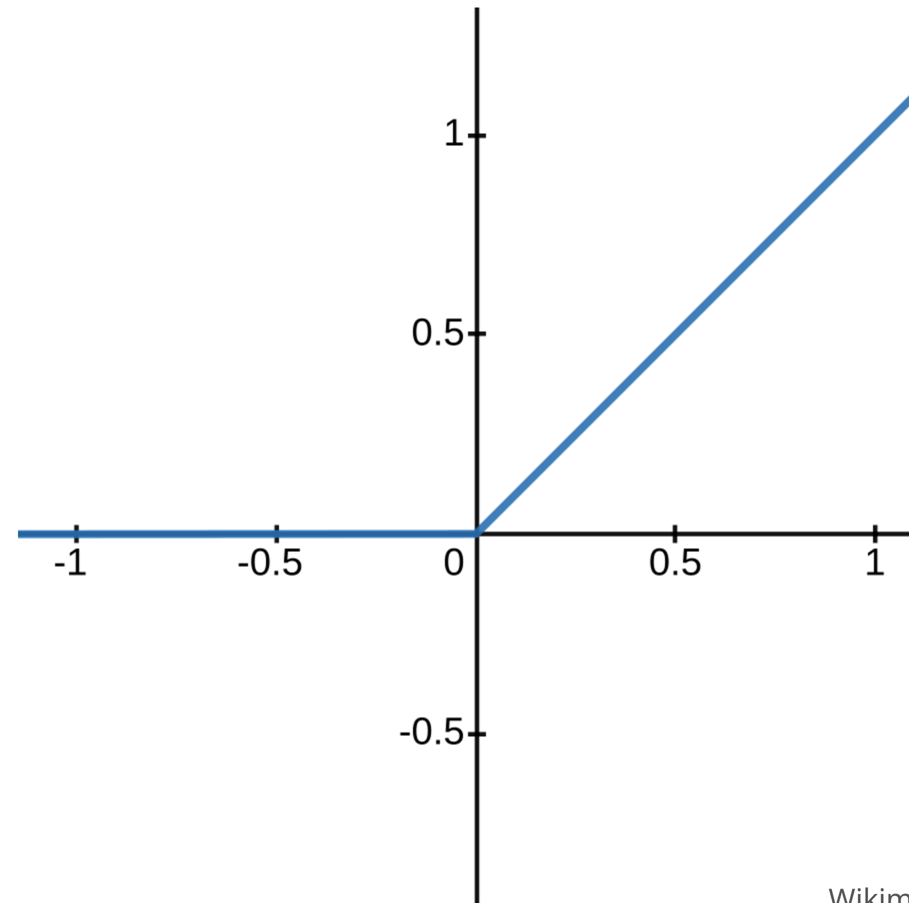


代表的な活性化関数 (3): ReLU

ReLU (Rectified Linear Unit): 広く使われている活性化関数。

$$\mathrm{ReLU}(z) = \max(0, z)$$

- **特徴**:
  - 計算が非常に高速（比較と最大値を取るだけ）。
  - $z > 0$ の領域では勾配が常に $1 (> 0)$
  - 入力が負の場合はパラメタは更新されない



Wikimedia Commons

ほかにもいろいろ（本質ではないので軽く）

- **Leaky ReLU**: $z < 0$ のときに小さな傾きを持たせることで、勾配が0になるのを防ぐ。

$$\text{Leaky ReLU}(z) = \max(0.01z, z)$$

- **Swish**: Googleが提案した活性化関数で、ReLUの"改良版"

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

- **GELU**: Transformerモデルでよく使われる（式は複雑）

活性化関数の選び方

- **隠れ層のデフォルト**: まずは **ReLU** を試すのが定石。
  基本的にはそれで十分だが個人的には GELU が好き
- **出力層**:
  - **二値分類: シグモイド関数**（確率を出力）
  - **多クラス分類: ソフトマックス関数**（合計1の確率分布を出力）
  - **回帰: 活性化関数なし**（恒等関数）

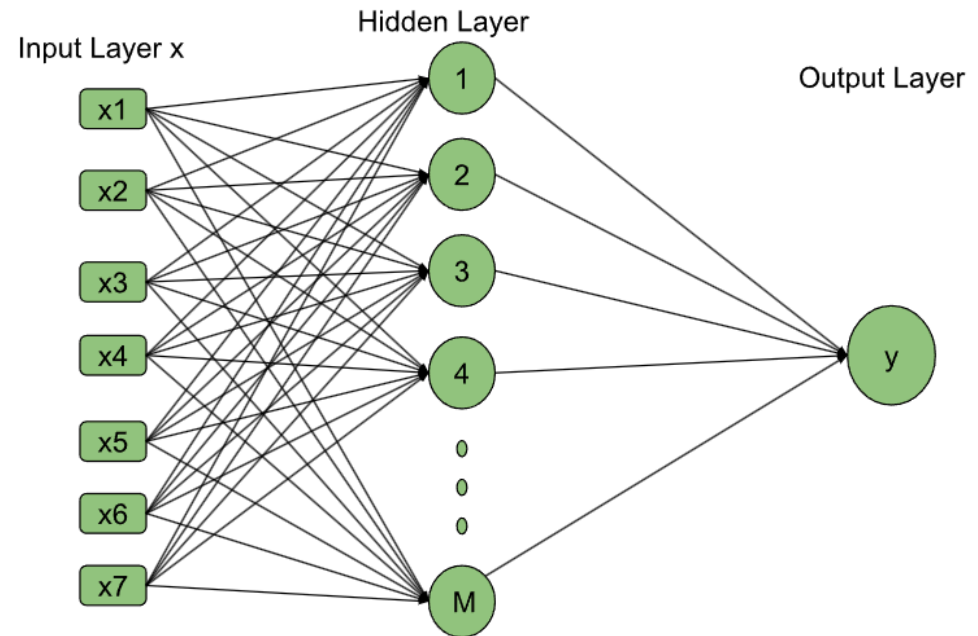
tanh や シグモイド関数を使うことは今はほぼないです。

5. ニューラルネットワークの表現力

万能近似定理 (Universal Approximation Theorem)

定理の主張（ざっくりと）：

「隠れ層が1層だけのニューラルネットワークでも、その隠れ層に十分な数のニューロンがあれば、どんな連続関数でも好きな精度で近似できる。」



https://medium.com/@jaiganesan/the-universal-approximation-theorem-the-backbone-of-mlp-ann-and-ffnn-2640fdbd3757

万能近似定理の意味

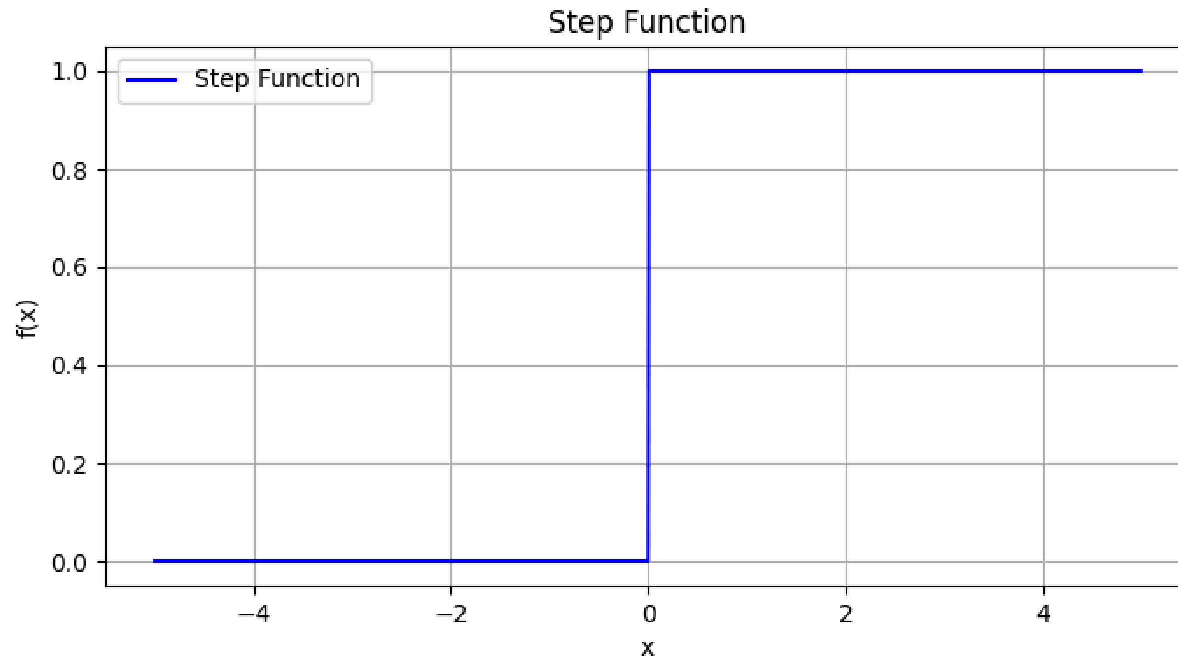
- 理論上、ニューラルネットワークは**極めて高い表現力**を持つ。
- 私たちが解きたい問題（関数）がどんなに複雑でも、十分に大きなニューラルネットワークならそれを表現できる可能性がある。

（※ただし、学習が成功するか、どれくらいのニューロンが必要か、未知データについてもうまくいくかは教えてくれない）

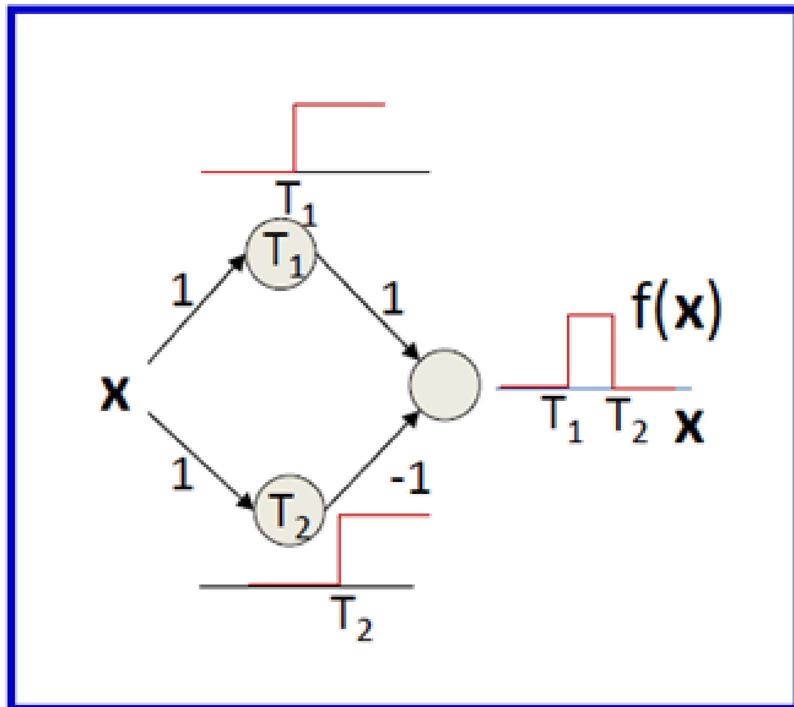
万能近似定理の直感的イメージ (ステップ関数の場合)

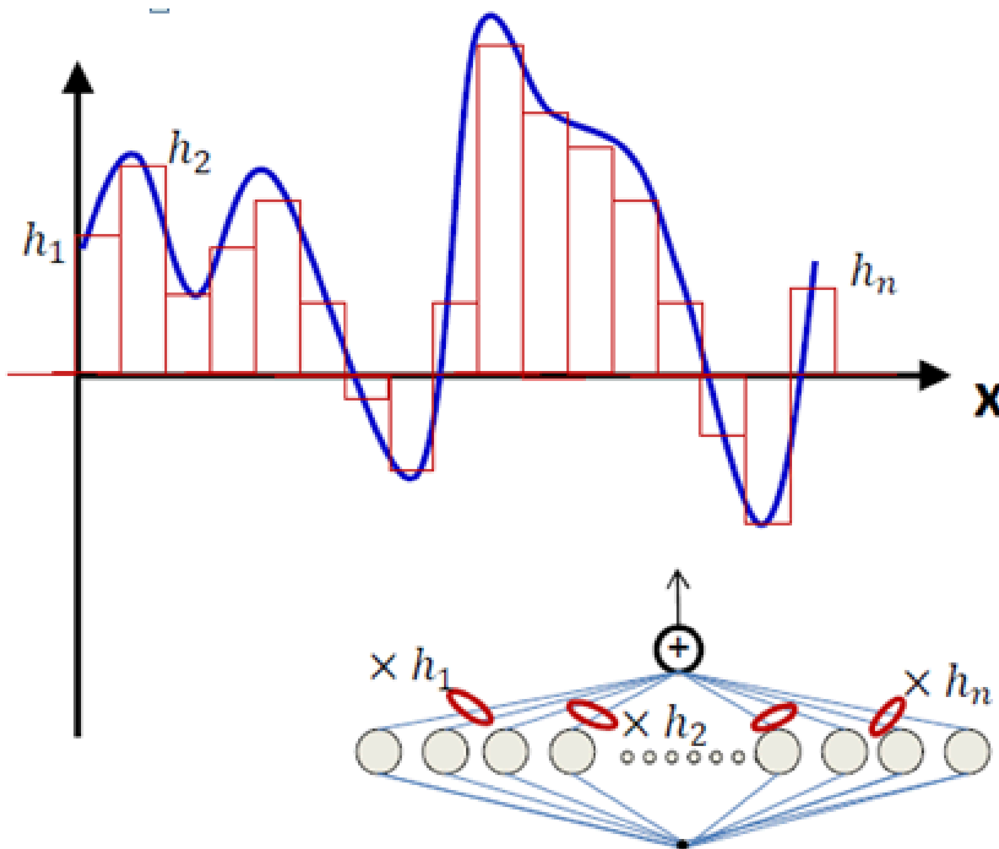
入出力が一次元で、活性化関数がステップ関数のニューラルネットワークを考えて、どんな関数でも近似できそうだということを示しましょう。

（ステップ関数を使うと学習できないが、あくまで例として）



万能近似定理の直感的イメージ (ステップ関数の場合)





https://medium.com/analytics-vidhya/neural-networks-and-the-universal-approximation-theorem-e5c387982eed

深さ vs 幅 (Deep vs Wide)

万能近似定理は「幅が広ければ」1層で十分と言っていますが、なぜ現代のモデルは「深い」(deep) のでしょうか？

- **効率性**: 深いネットワークは、同じ表現力をより少ないパラメータで達成できることが多い。
- **階層的特徴学習**: 深い構造は、単純な特徴から複雑な特徴へと、より自然な形で特徴の階層構造を学習するのに適している。

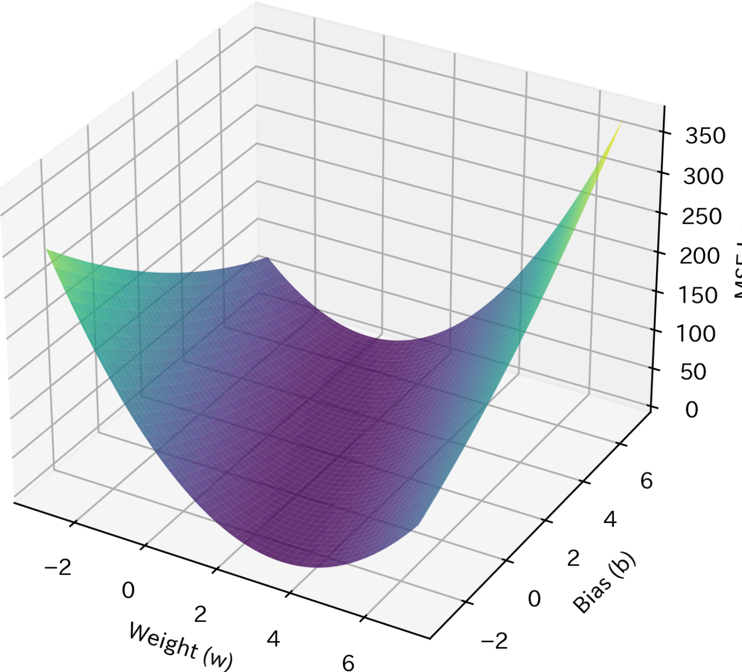
あくまで感覚ですが、
幅の広い浅いネットワーク: すべての特徴を一度に学習しようとする（丸暗記に近い）。
深いネットワーク: 段階的に、再利用可能な知識を学習しようとする

6. 次元の祝福 (Blessing of Dimensionality)

損失関数の見た目 (Loss Landscape)

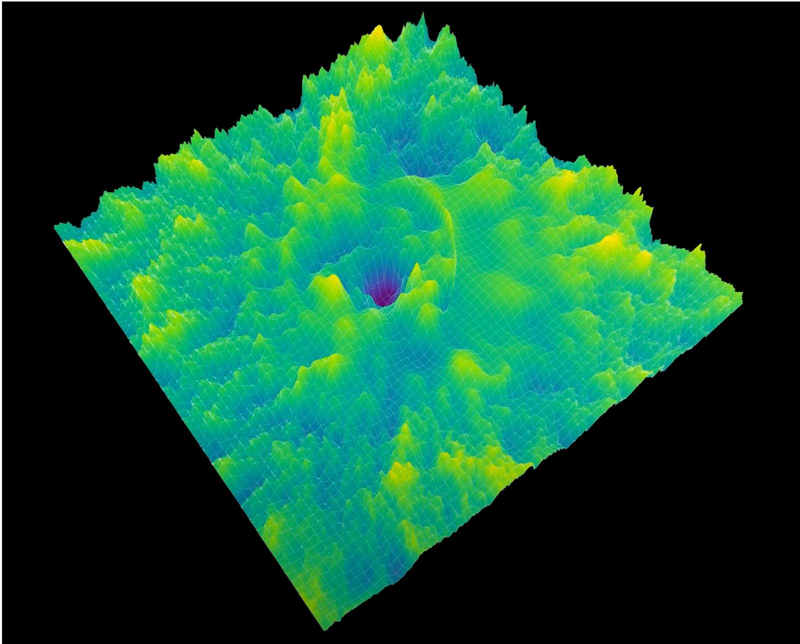
線形回帰の損失関数

- お椀型（凸関数）



ニューラルネットワーク（Llama3.2）の損失関数

- 非常に複雑で凸凹（非凸関数）

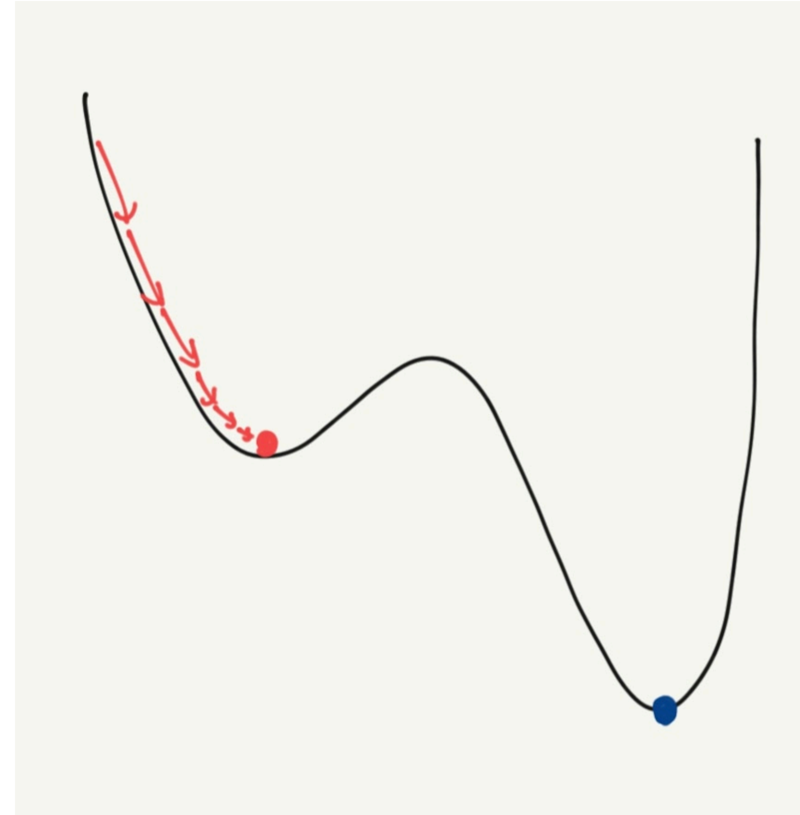


右図は https://www.youtube.com/watch?v=NrO20Jb-hy0

局所最適解

低次元の勾配降下法では、局所最適解（local minimum）は最適化の大きな障害

一度ハマってしまうと、そこから抜け出すのは困難です。
ニューラルネットワークの損失関数も凸凹なので、この問題は深刻に思えますが…？



赤い点が極所解、青い点が真の最適解

高次元では、ほとんどが「鞍点」

ニューラルネットワークのパラメータ数は数百万〜数千億にもなります。このような超高次元空間では、状況が大きく変わります。

- **局所最適解 (Local Minimum)**: 全ての方向に対して損失が増加する点。
- **鞍点 (Saddle Point)**: ある方向には損失が増加し、別の方向には損失が減少する点。

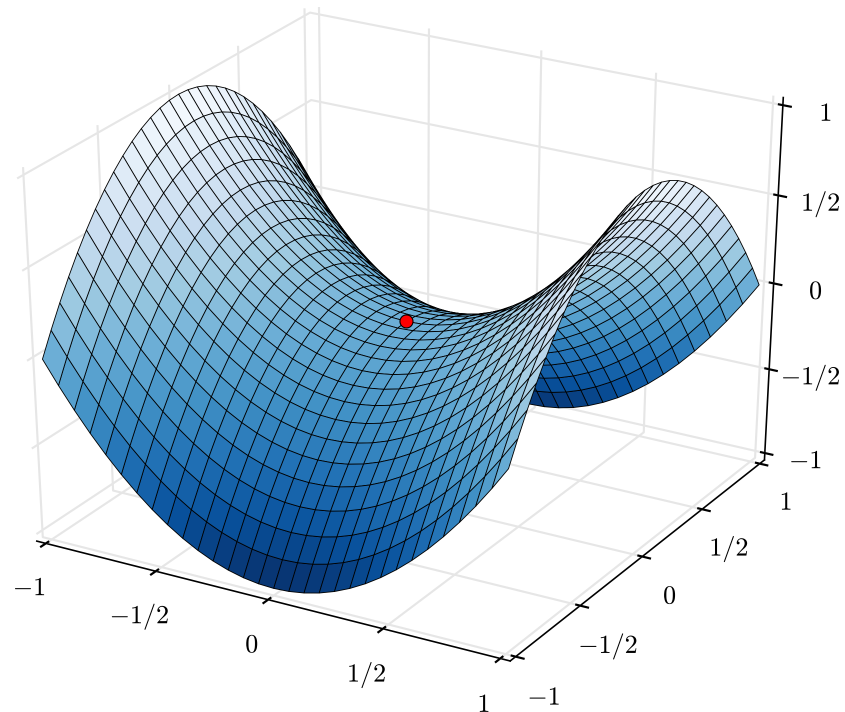
(どちらも微分係数は0)

意外なこと: 高次元空間では、勾配が0になる点の**ほとんどが鞍点**であり、真の局所最適解は非常に稀です。

鞍点のイメージ

馬の鞍（くら）のように、ある軸で切ると谷底に見えるが、別の方向から切ると山頂に見える点です。

勾配降下法ではピッタリと鞍点にたどり着かない限り学習は止まらない



Wikimedia Commons

なぜ局所最適解は稀なのか？

パラメータが N 個あるとします。ある微分係数が0の点が局所最適解であるためには、N **個全ての次元**に対して、その点が谷底である必要があります。

- ある1つの次元で谷底である確率を p とします (例: $p = 0.5$)
- N個全ての次元で同時に谷底である確率は p^N となります。

Nが巨大（例: $1,000,000$）な場合、p^N は天文学的に小さな値になり、ほぼ0に等しくなります。

結論: パラメータ空間のどこかには、必ずと言っていいほど「下り坂」が残っている！

高次元の「祝福」

- **次元の呪い (Curse of Dimensionality)**: 高次元空間ではデータが疎になるなど、多くの問題が難しくなる現象。
- **次元の祝福 (Blessing of Dimensionality)**: 最適化においては、次元が高いほど局所最適解にハマりにくくなり、探索空間が広がるという「祝福」の側面もある。

このため、非常に複雑な非凸関数であるにもかかわらず、単純な勾配降下法でニューラルネットワークの学習がうまくいくことが多いのです。

佐藤竜馬. (2024). 深層ニューラルネットワークの高速化. 技術評論社.

7. まとめ

本日のまとめ

- **ニューラルネットワーク**: 線形層と活性化関数を交互に重ねたモデル。層を重ねることで、単純な特徴から複雑な特徴への階層的な学習を可能にする。
- **線形層**: $z = Wx + b$ の計算を行い、入力を線形変換する。
- **活性化関数**: ネットワークに**非線形性**を導入し、表現力を飛躍的に高める。
- **順伝播**: 入力データが層を順番に通過し、最終的な予測値が出力されるまでの計算プロセス。
- **表現力**: ニューラルネットワークは、理論上どんな関数でも近似できる非常に高い表現力を持つ（**万能近似定理**）。
- **高次元最適化**: 損失関数は複雑だが、高次元では局所最適解が稀で鞍点がほとんど。そのため、勾配降下法でもうまく学習が進みやすい。

次回予告

- **次回**: 誤差逆伝播法 (Backpropagation)
- ニューラルネットワークの学習には、損失関数に対する各パラメータの勾配 $\frac{\partial L}{\partial W}$ や $\frac{\partial L}{\partial b}$ が必要。
- しかし、ネットワークが深くなると、この勾配計算は非常に複雑になる。
- **誤差逆伝播法**は、この膨大な勾配を効率的に計算するための賢いアルゴリズム。微分の連鎖律を応用したもの。