# 深層学習

## Feed Forward Network - 順伝播型ネットワーク

# 話の流れ

- 線形性、非線形性
- ニューロン
  - ニューラルネットワーク
- 活性化関数
- コスト関数

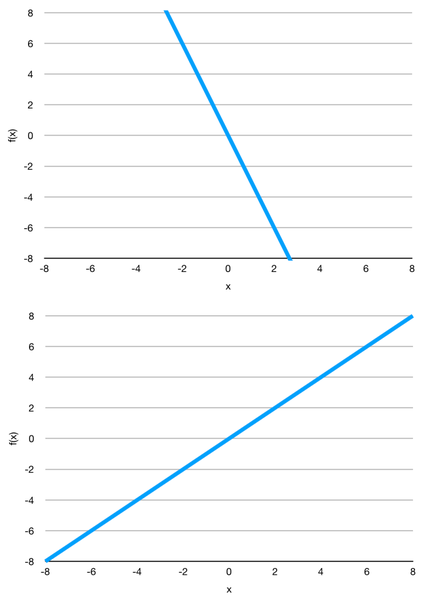
# 線形性
# linearity

# 線形性

- 大雑把にいうと、直線で表現できるような性質
- もう少し具体的にいうと、以下の2つが成り立つこと(x,yはベクトル)
  - f(x+y) = f(x) + f(y)
  - f(cx) = cf(x) : cは定数

# 非線形性

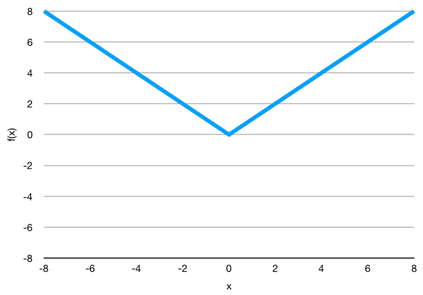
- 線形ではないものすべて

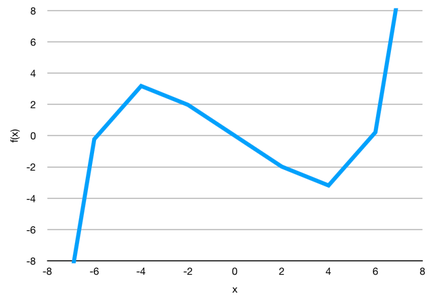






線形




非線形

# 機械学習における線形モデル

- 機械学習の文脈では次のようなモデルを線形モデルとよぶ
- 書き方はいろいろあるが、意味しているものは同じ

$$y = \theta_0 + \sum_{i=1}^{k} \theta_i x_i + \epsilon$$

$$y = \mathrm{w}^\top \mathbf{x} + b$$

$$y = \mathrm{w}^\top \mathbf{x}$$

# 機械学習における線形モデル

- なにかの要因x1, x2, x3, …, xn があったときに、それぞれの要因に適当な係数w1〜wnを掛けて、出力yを求めるような問題を考える

- たとえばx1が年齢、x2が体重、x3がBMI、…としたときにyとしてある病気にかかるリスクを計算したい

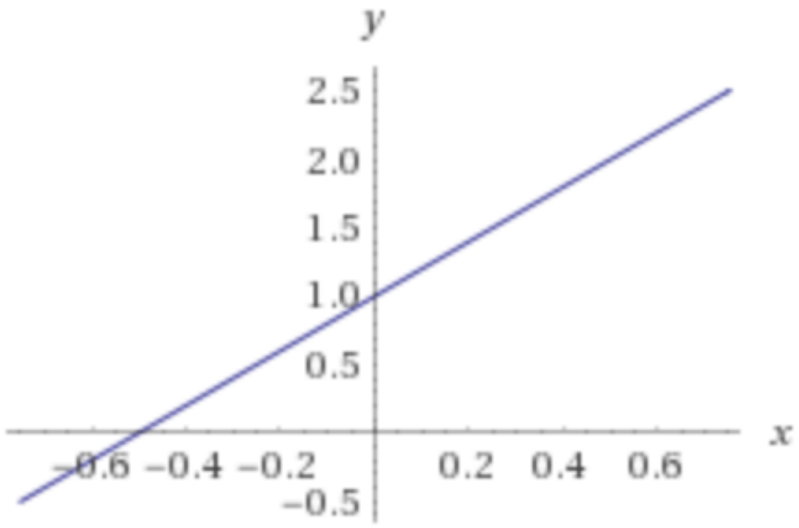
- つまり

$$\mathrm{y}=\mathrm{w}_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

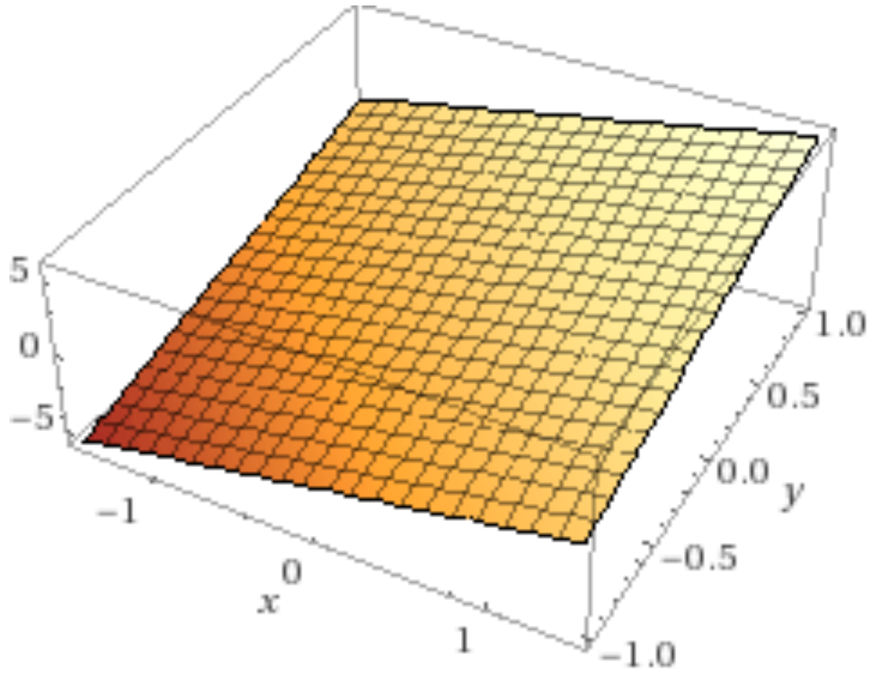
$$\mathrm{y}=\sum \mathbf{W}^\top \mathbf{X}$$

# 線形モデル

Open




y=2x+1




z=2x+3y

このようなモデルはどのようなwを求めようが、結局のところ直線(超平面)で近似するようなモデルになる

# 線形モデル=直線か？

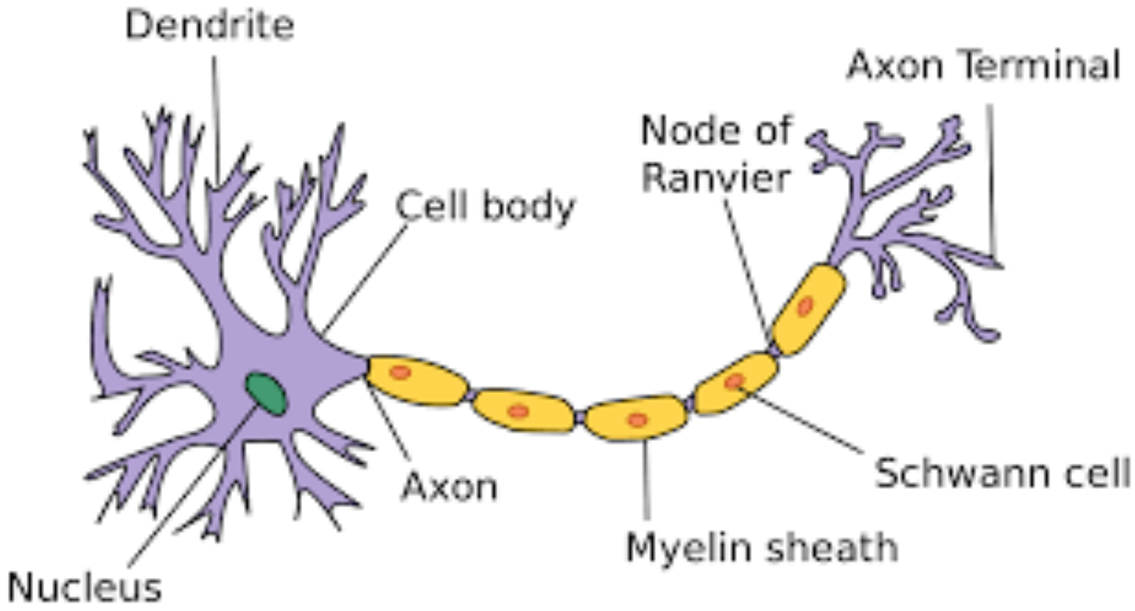
- y=ax+bのようなグラフが直線なのはそうだが、線形モデルを使うとまっすぐな分離面しか引けないかというとそういうわけでもない
- 一般的には「線形モデルではまっすぐな分離面をひく」というような説明がなされるしその認識でもだいたいあっているので、この講座でもそう説明しています
- 気になる人は線形モデルを使って曲線を引く方法を考えてみてください

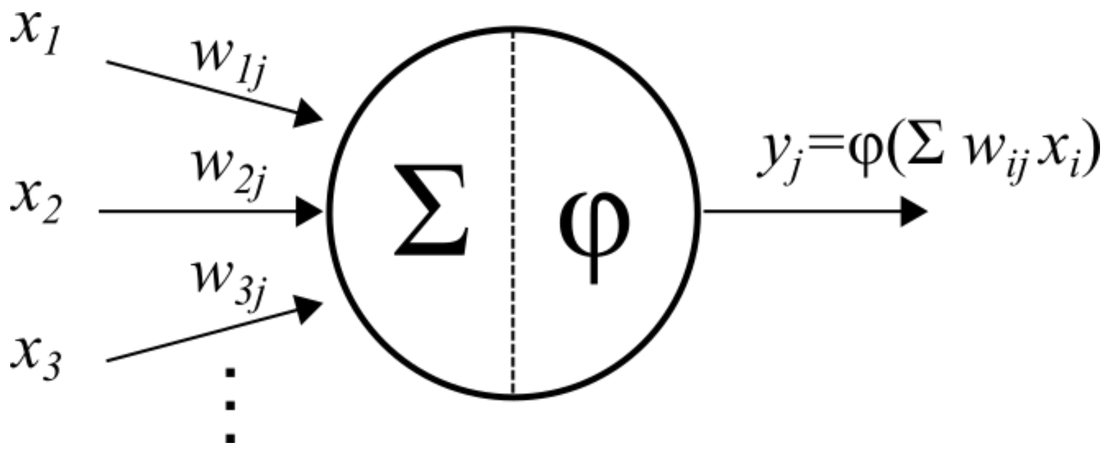
# 線形・非線形な関数の描画

```python
import numpy as np
import matplotlib.pyplot as plt

n_cols = 2
n_rows = 2

plt.figure(figsize=(20, 8))

x = np.arange(-5, 5, 0.1)

y = x
plt.subplot(n_rows,n_cols,1)
plt.title('x')
plt.ylim(-10,10); plt.plot(x, y)

y = -2 * x - 3 * x + x
plt.subplot(n_rows,n_cols,2)
plt.title('2x-3x+x')
plt.ylim(-10,10); plt.plot(x, y)

y = np.sin(x)
plt.subplot(n_rows,n_cols,3)
plt.title('sin(x)')
plt.ylim(-10,10); plt.plot(x, y)

y = np.exp(x)
plt.subplot(n_rows,n_cols,4)
plt.title('exp(x)')
plt.ylim(-10,10); plt.plot(x, y)
```

# ニューロン
# Neuron

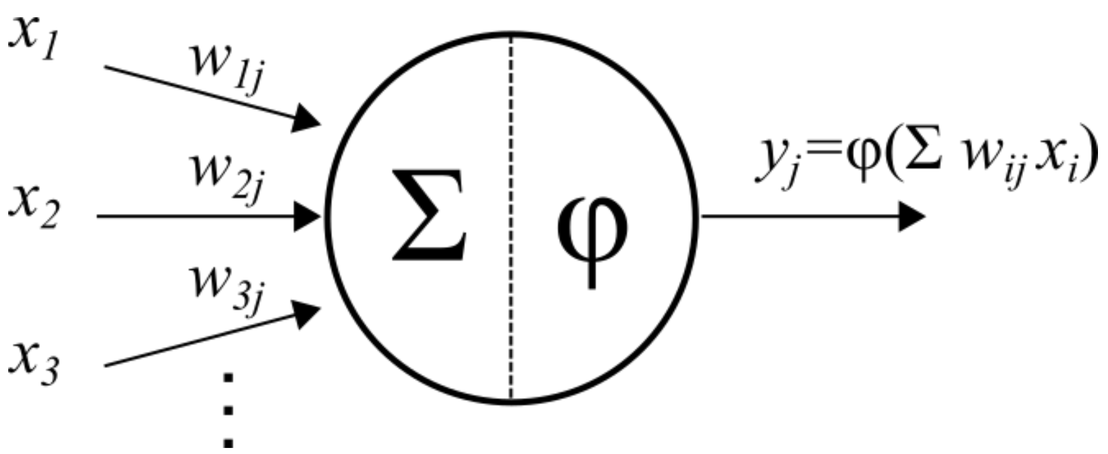
# ニューロンと人工ニューロン



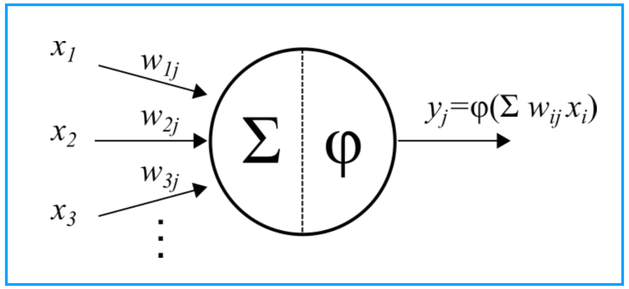



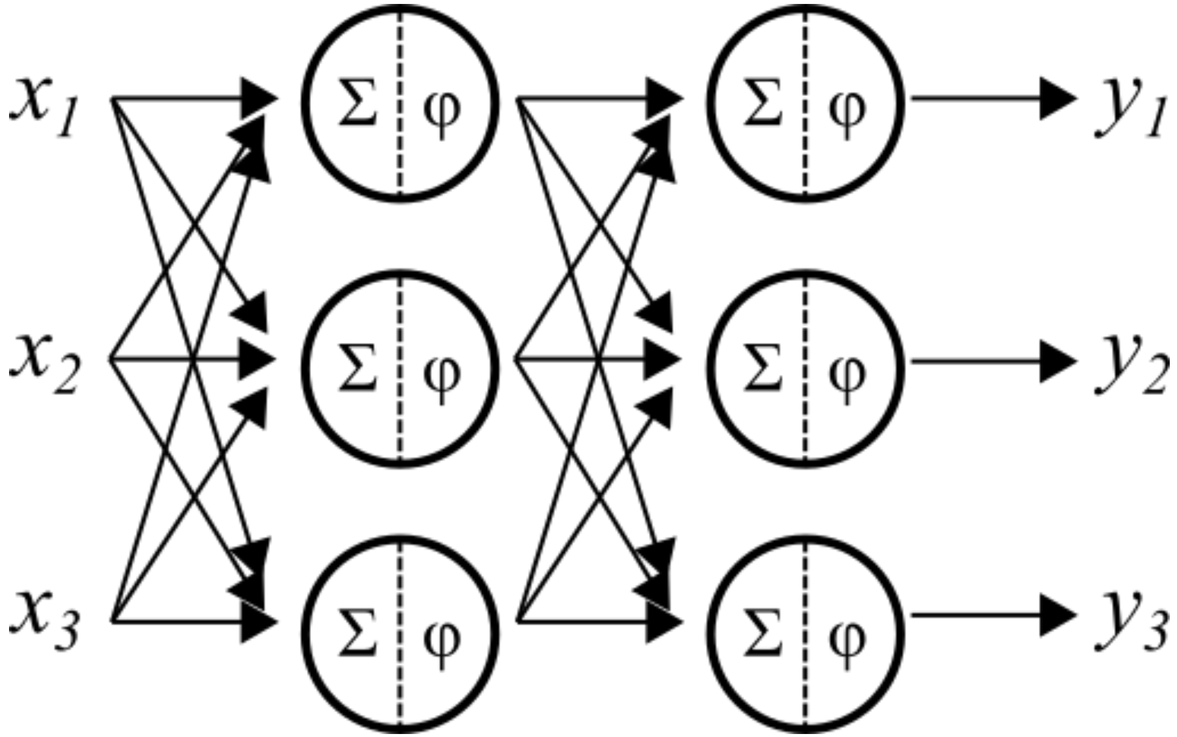
# ニューロンと人工ニューロン

- つまり、いくつかの入力xがあったときにそれぞれに重みwをかけ、結果をすべて足してから活性化関数を通して出力する
- 言い換えると入力ベクトルXに重みベクトルWをかけて足す、内積をとる操作になる



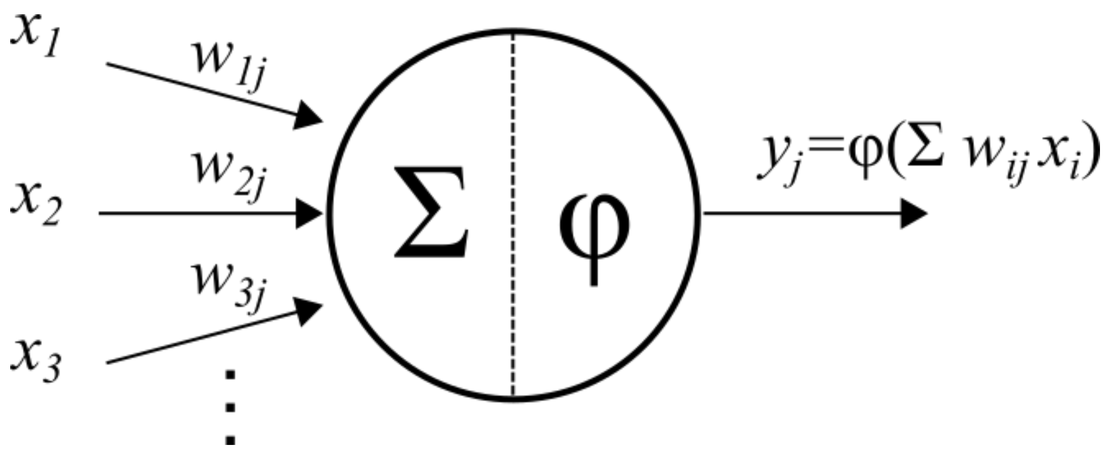
# ニューラルネットワーク(feedforward)



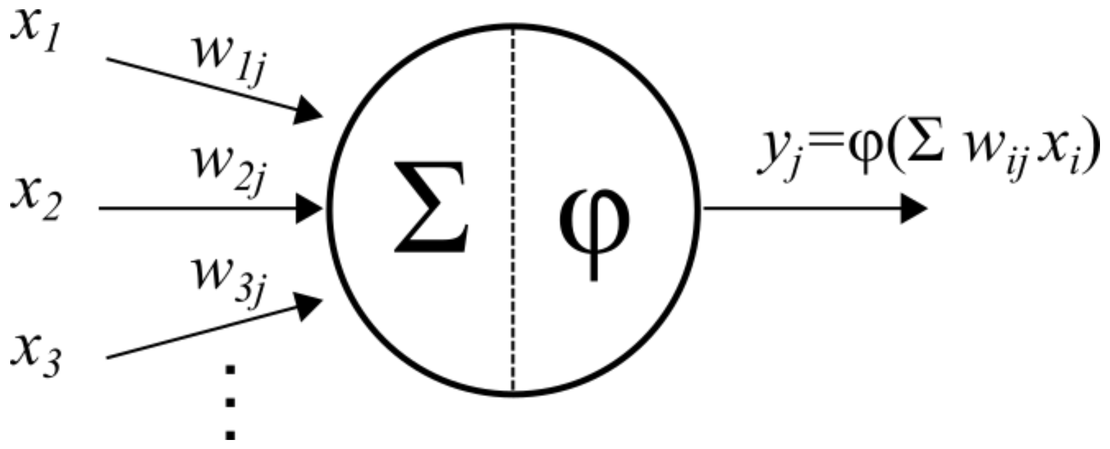
# なぜ活性化関数？

- 活性化関数がないとただの線形モデルにしかならない
- 線形モデルを組み合わせても線形であることには変わらないので、多層にする意味が薄れる

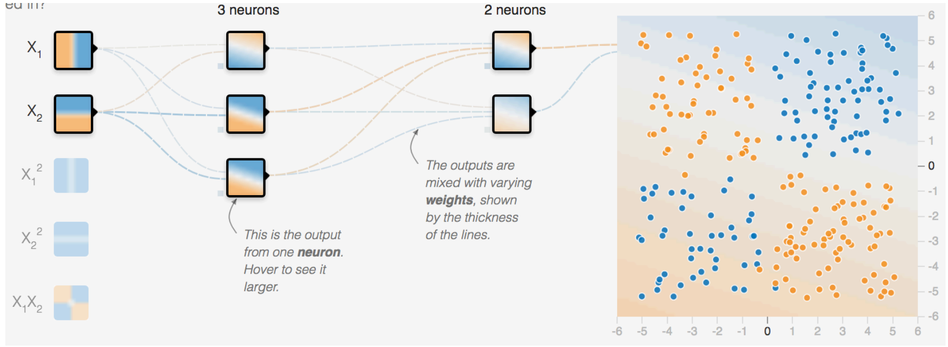


# なぜ活性化関数？

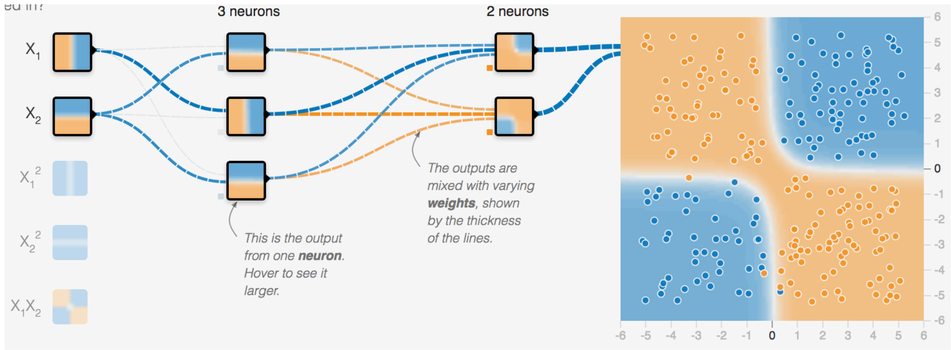
- 活性化関数を通してモデルに非線形性を与えることができる
- より複雑で自由度の高いモデルを作ることができる
- しかし学習は非常に難しくなる…



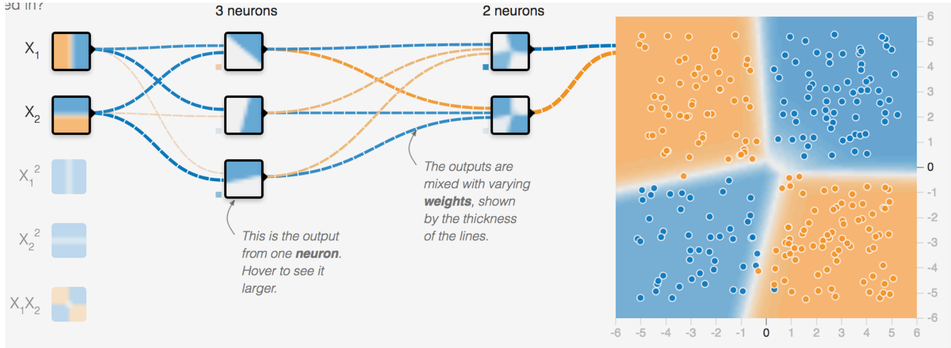
# 活性化関数による振る舞いの違い




Linear




Tanh

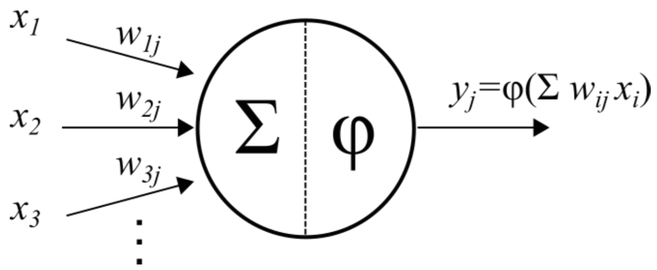


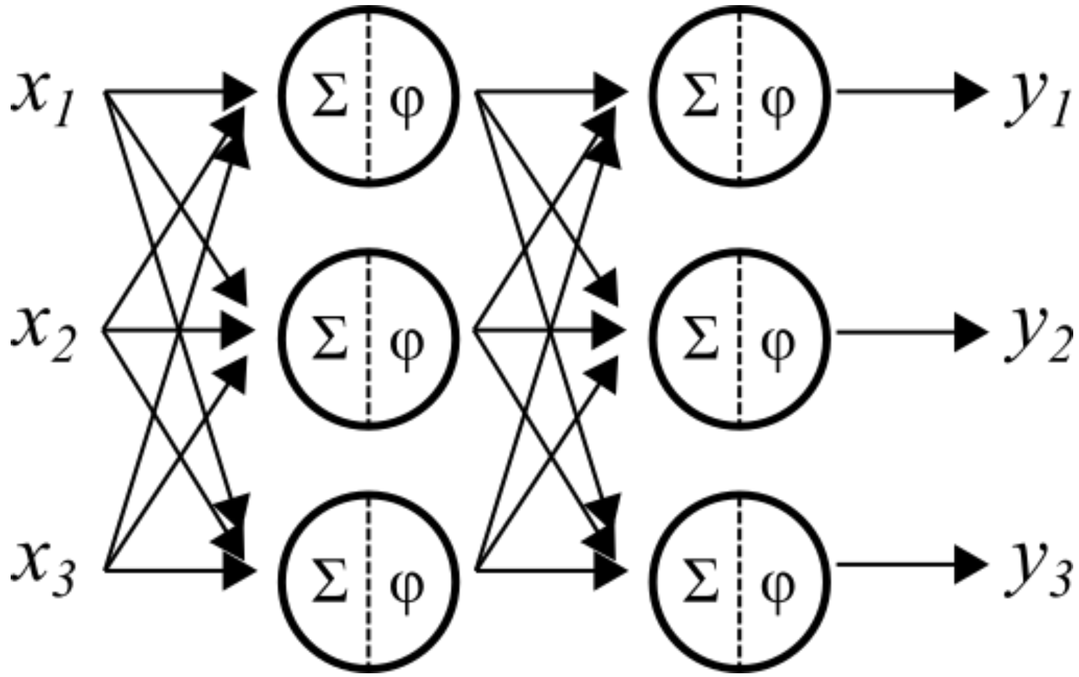

ReLU

# ニューラルネットワークの学習

- 下のようなNNがあったとき、各重みw(とφ)をうまく決めれば、y=f(x)が学習できるはず（いろいろな条件はあるが）

- ただ、困ったことにこのwを求めるのはかなり難しいことが知られている



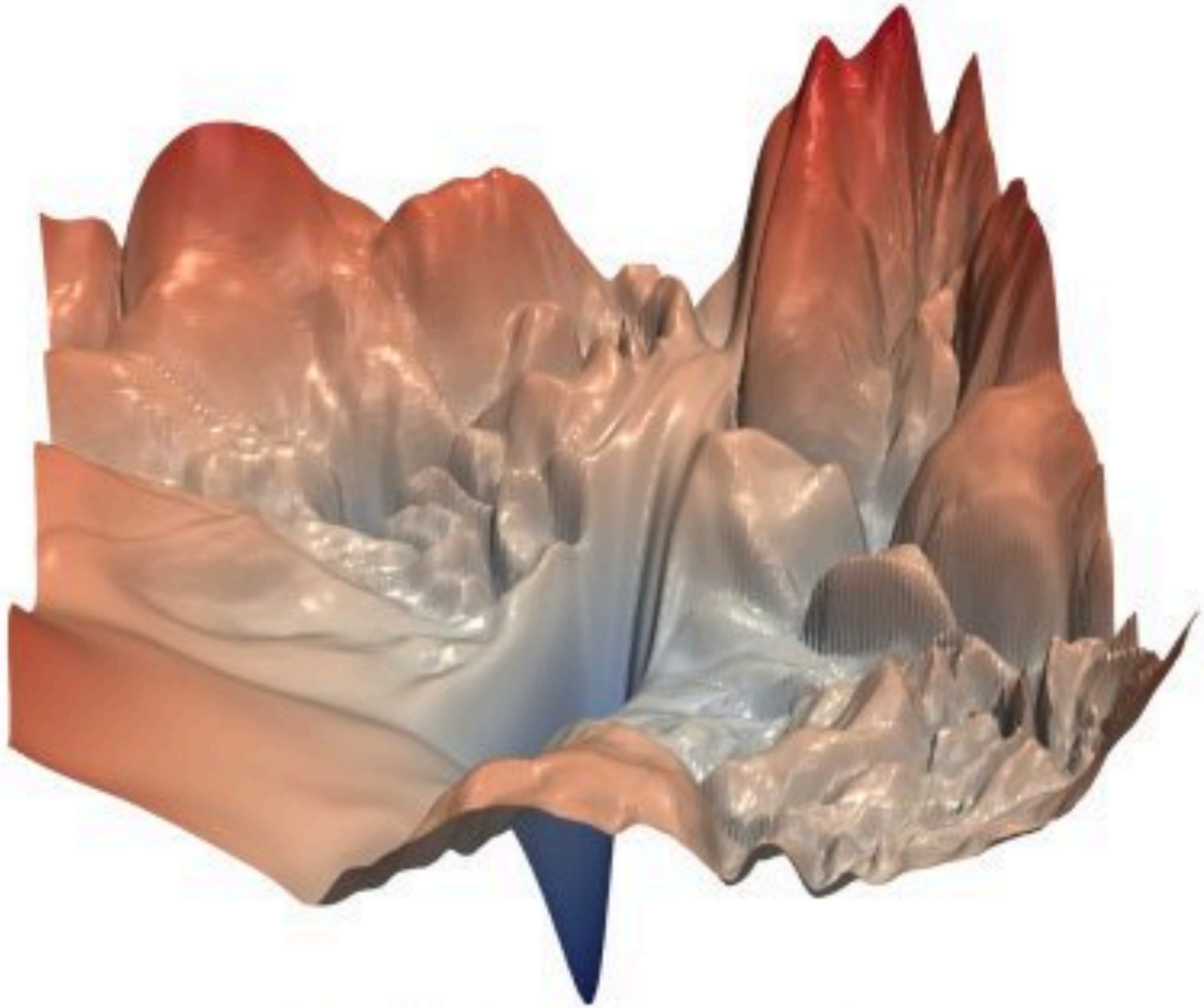
# パラメータを求める

- たくさんのパラメータがあるときに全体を最適化したい、というのは情報工学ではよく出てくる
- いろいろなやりかたがあるが、それぞれ難易度が違う

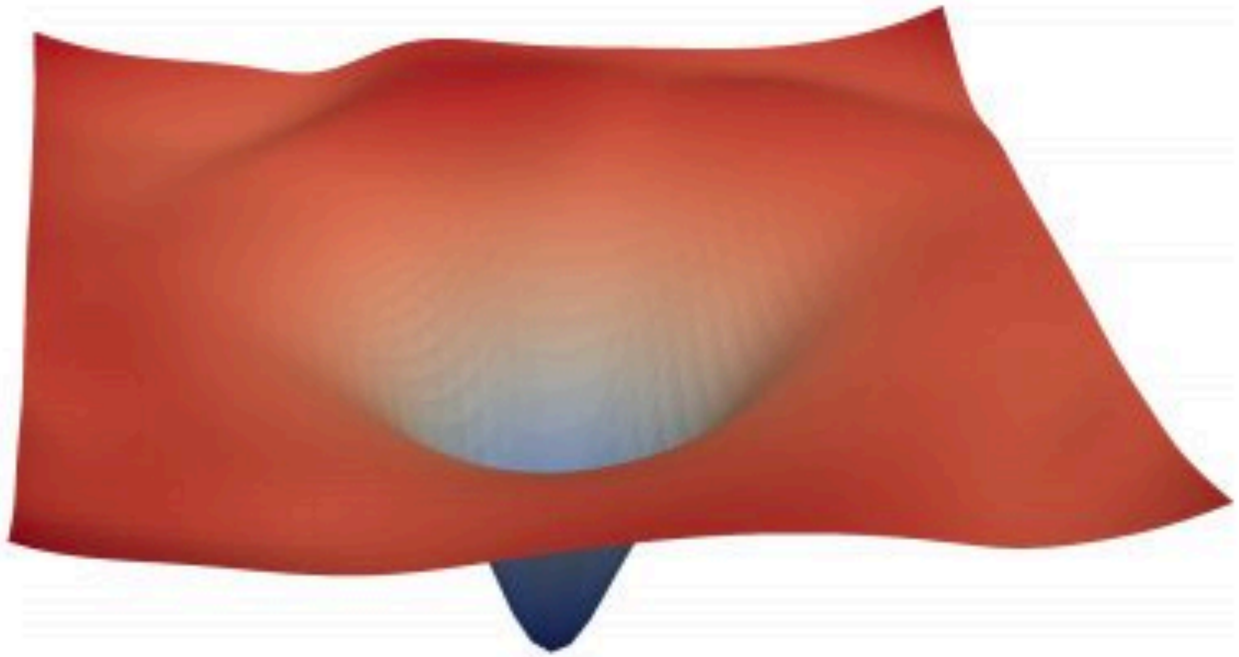
# パラメータを求める

1. 凸最適化
    - 線形分類器など：この種の最適化の中ではいちばん簡単
2. **非凸最適化で勾配ベースの方法**
    - 多層のニューラルネットワーク（ディープラーニングも）はこれ
3. 非凸最適化で勾配フリーな方法
    - とても難しい(時間がかかる)のでできれば避けたい

# コスト関数(損失関数)



(a) without skip connections



(b) with skip connections

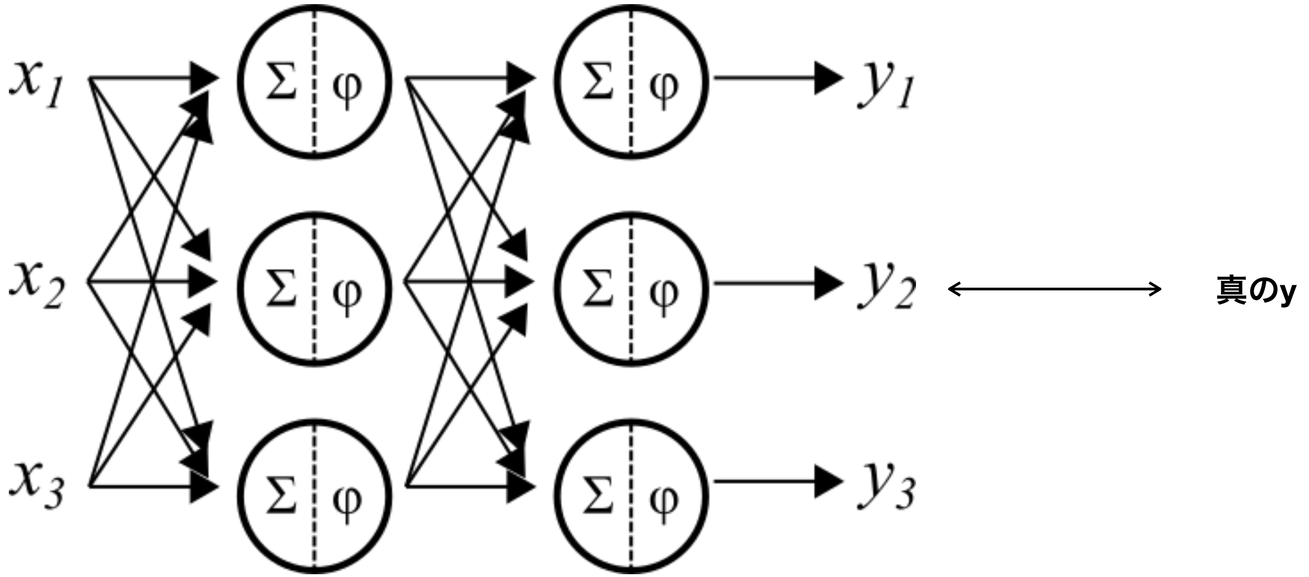
# コスト関数（損失関数）

- 何かの基準で「悪さ」を決めて、悪さが少なくなるようなパラメータを学習すると考え、この基準をコスト関数とよぶ
- しかし一般にニューラルネットワークは非線形モデルとなる
  - このようなモデルはコスト関数が非凸になる
  - 凸関数であれば最適化は容易だが、NNは最適化が難しい
  - 後に述べる確率的勾配降下法などでの反復計算が必要

# NNの最適化

そこで、次のようなアプローチで最適化することを試みる

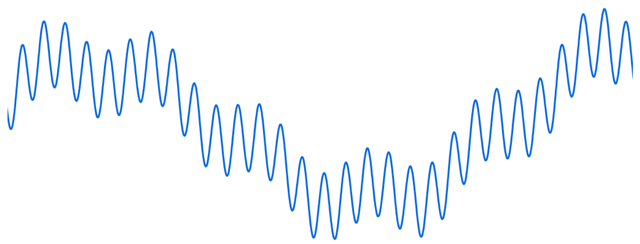
- 正解の値とNNが出した値を比べる
- どれくらい間違っているのか(損失)を求める
- 間違いが少なくなるようにすこしだけWを増減させる

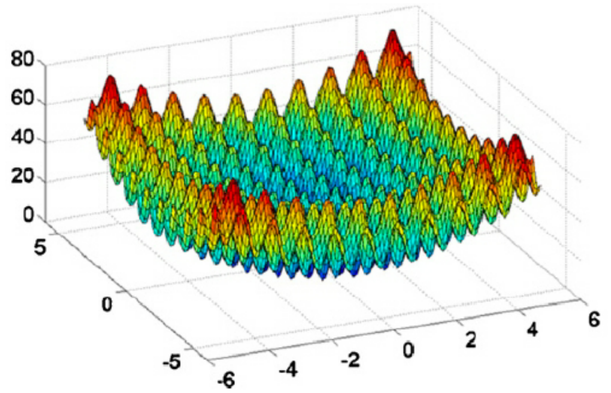


# NNの最適化

NNの最適化（良いパラメータを探すこと）はかなり面倒くさい

- NNのコスト関数は複雑過ぎてどこが大域的最適解なのか不明
- 非線形・非凸だが、連続で微分可能であることは既知
- 実際にDeep Learningのコスト関数がどのような形(性質)を持っているのかはまだよくわかってないことが多い

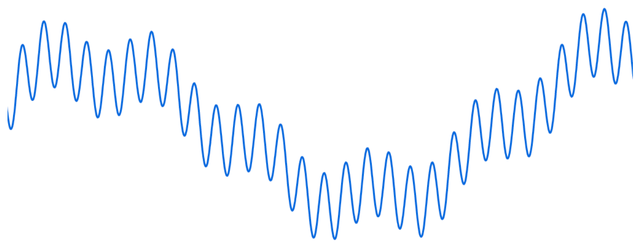


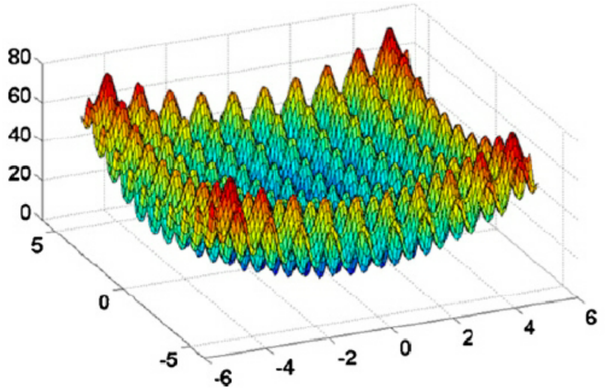



**連続で微分可能だが非線形非凸な関数のイメージ図**

# コスト関数

- NNにおいて最適な重みを求めるにはどうすればよいか？
- 適当なコスト関数（損失関数）を定義する
  - この値が大きければ良くない、小さければ良いということになる（このような関数は無数にある）
- 今の状態からなるべく損失が小さくなるように各パラメータを少しだけ動かす、というのを繰り返す

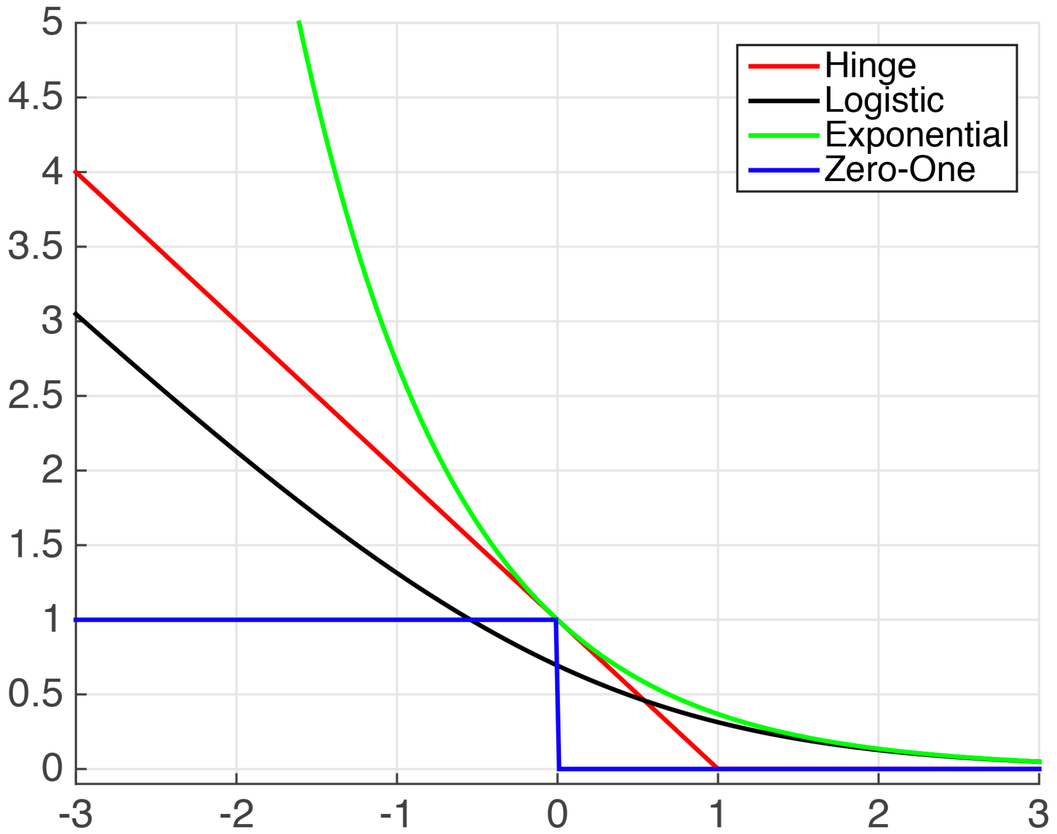





連続で微分可能だが非線形非凸な関数のイメージ図

# コスト関数

- 単純な例を考えてみる
  - 正解だったら0, 間違えてたら1を返す関数を考える(0-1損失)
  - これもコスト関数の要件は満たすが、学習はできない
    - 今の状態がどれくらい良いのか？が不明
    - 微分しても0
    - 非連続
- Deep Learningではよいコスト関数がすでに考案されている
  - 計算が簡単＆収束が早い＆理論が明快



# コスト関数

- それなら、たとえば二乗誤差はどうか？
  - つまり (y-t)^2を損失とする
    - これなら普通に学習はできる
    - ただしNNに適した高速に収束するコスト関数がすでに提案されているので、普通はそちらを使う
    - 代表例は cross entropy
      - Cross entropy はよく出てくるので、導出してみよう

# コスト関数

- それなら、たとえば二乗誤差はどうか？
  - つまり (y-t)^2を損失とする
    - これなら普通に学習はできる
    - ただしNNに適した高速に収束するコスト関数がすでに提案されているので、普通はそちらを使う
    - 代表例は cross entropy
      - Cross entropy はよく出てくるので、導出してみよう
      - 「ニューラルネットワークの尤度が最大化するようにパラメータを決める」という考えから出発すると導出できる

# 最尤推定

- 入力xに対する正解 $y_t$ が当たりの事後確率を $p(y_t \mid x,W)$とする
- 逆に、 $p(x,W|y_t)$ を考えて、尤度とよぶ
  - 正解yを観測できたときにどのようなxとWがありえそうか？
- x は学習データとして与えられているので、尤度を最大化するような W を決めれば“良い”ネットワークになるはず

# 最尤推定

- σをシグモイド関数とし、ニューロンが発火する確率を

  p(C = 1| x) = σ (W・x + b)

- 逆に発火しない確率を以下とする

  p(C = 0| x) = 1 - σ (W・x + b)

- y=σ (W・x + b) だから、下のように簡単にかける

$$\mathrm{p}(\mathrm{C} = \mathrm{t} \mid x) = y^t(1-y)^{1-t}$$

# 最尤推定

**3層のFFネットワークに入力xを与えた時の出力zのもとで、**
**教師信号t=1である確率の推定値を考える**

- 入力x
- 中間層y
- 出力z
- ネットワークの尤度L(W) は

$$\boldsymbol{x} = (x_1, ..., x_I)^T$$

$$\zeta_j = \sum_{i=1}^{I} a_{ji} x_i$$

$$y_j = \sigma(\zeta_j)$$

$$\eta_k = \sum_{j=1}^{J} b_{kj} y_j$$

$$z_k = \sigma(\eta_k)$$

$$\mathrm{L(W)} = \prod_{p=1}^{P} \prod_{k=1}^{K} z_{pk}^{t_{pk}} (1 - z_{pk})^{(1-t_{pk})}$$

# 最尤推定

- ネットワークの尤度を対数尤度に変換すると

$$L = \prod_{p=1}^{P} \prod_{k=1}^{K} z_{pk}^{t_{pk}} (1 - z_{pk})^{(1-t_{pk})}$$

$$l = \sum_{p=1}^{P} \sum_{k=1}^{K} (t_{pk} \log z_{pk} + (1 - t_{pk}) \log(1 - z_{pk}))$$

- これは符号を変えればクロスエントロピーと同じである
- クロスエントロピーを使った損失は(たとえば二乗誤差より)収束が早い
- KLダイバージェンスを最小化するのと等価

# 活性化関数
# Activation Function

# 活性化関数

- ニューラルネットワーク全体の良し悪しを決める指標が損失関数
- 各ニューロンについてる$\phi$、活性化関数についても色々なものがある
- Goodfellow本では、出力層か否かで活性化関数を分類しているので、ここでもそのように紹介する
  - 隠れ層は単にうまく高速に学習さえできればなんでもよい
  - 出力層は「人間がどのような出力が欲しいか？」で決まる

# 活性化関数に欲しい性質

- 活性化関数は無数に考えられるが、次のような性質を持っていることが望ましい（あるいは必須）
    - 学習が早い
        - 勾配が消えない、誤差が伝わりやすい
    - 高速に良い解に向かう
    - 出力層の場合：値域が扱いやすい（たとえば0〜1とか）
    - 微分可能
    - 計算が簡単、微分しても簡単（計算コストが低い=早い）

# 活性化関数
# （出力層）

# 出力ユニット

- ここで紹介するのは以下の3つ
  - 線形ユニット（線形関数）
  - シグモイドユニット（シグモイド関数）
  - ソフトマックスユニット（ソフトマックス関数）
- Goodfellow本ではある活性化関数が適用されたニューロンを指して「○○ユニット」と呼んでいるが、あまり一般的ではないと思う

# ガウス出力分布のための線形ユニット

- NNが計算した結果をそのまま欲しい場合もある
- たとえば明日の株価を予測したい、気温を予測したいなど
- この場合は活性化関数を使わなければ良い
    - 言い換えると恒等写像を使えば良い

# ガウス出力分布のための線形ユニット

- NNが計算した結果をそのまま欲しい場合もある
- たとえば明日の株価を予測したい、気温を予測したいなど
- この場合は活性化関数を使わなければ良い
    - 言い換えると恒等写像を使えば良い

# ベルヌーイ出力分布のための シグモイドユニット

- シグモイド(Sigmoid)関数を出力層に使うことは結構多い
- どんな入力が来てもかならず出力は0〜1に収まる
  - 確率が欲しいときに便利
- 式が簡単、微分してもシンプルな式のまま
- 連続で有界なので扱いやすい
- が、単に経験則から選んでいるわけではなく、これもちゃんと導出できる

# ベルヌーイ出力分布のための シグモイドユニット

- 出力がベルヌーイ分布(0か1をとる)に従っているとする
- 今入力xに対して出力y=1となるような条件付き確率を

$$p = p(y_i = 1|\boldsymbol{x}_i)$$

- とし、事後確率の比の対数を線形和で表す事を考える

$$\log \frac{p(y_i = 1|\boldsymbol{x}_i)}{1 - p(y_i = 1|\boldsymbol{x}_i)} = \boldsymbol{w}^T \boldsymbol{x}$$

これは結局、入力xの線形関数としてモデル化できる。

$$\log \frac{p(y_i = 1|\boldsymbol{x}_i)}{1 - p(y_i = 1|\boldsymbol{x}_i)} = \boldsymbol{w}^T \boldsymbol{x}$$

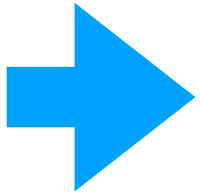
$$\frac{p(y_i = 1|\boldsymbol{x}_i)}{1 - p(y_i = 1|\boldsymbol{x}_i)} = e^{\boldsymbol{w}^T \boldsymbol{x}}$$

$$p(y_i = 1|\boldsymbol{x}_i) = \frac{e^{\boldsymbol{w}^T \boldsymbol{x}}}{1 + e^{\boldsymbol{w}^T \boldsymbol{x}}}$$

$$= \frac{1}{1 + e^{-\boldsymbol{w}^T \boldsymbol{x}}}$$

$$= \sigma(\boldsymbol{w}^T \boldsymbol{x})$$

**sigmoid関数とよぶ**



# マルチヌーイ出力分布のための ソフトマックスユニット

- マルチヌーイ分布についても考えてみる
- カテゴリカル分布とも呼ばれる
- N個の目をもつサイコロを振ったときの出目の分布
- 例) 画像がどのクラスに該当するかどうか分類する

# ソフトマックスユニット

- Sigmoidなどの出力そのままだと扱いづらい
- そこで、k番目の出力ユニットの値を次のように求める

$$y_k = \frac{e^{\boldsymbol{x}_k}}{\sum_{i=1}^{K} e^{\boldsymbol{x}_i}} = \frac{\exp(\boldsymbol{x}_k)}{\sum_{i=1}^{K} \exp(\boldsymbol{x}_i)}$$

- $0 < y_k < 1$ かつ $y_1 + \ldots + y_K = 1$ になる(確率として扱える)

# ソフトマックスユニット

- 他の性質として、ある出力ユニットyの値が相対的に大きかった場合、yの値が1により近くなり、他は小さい値となる。

- 極端な場合は、特定のyだけがほぼ1に近い値で、他は0に近い値をとる(Winner-Take-All, WTA)

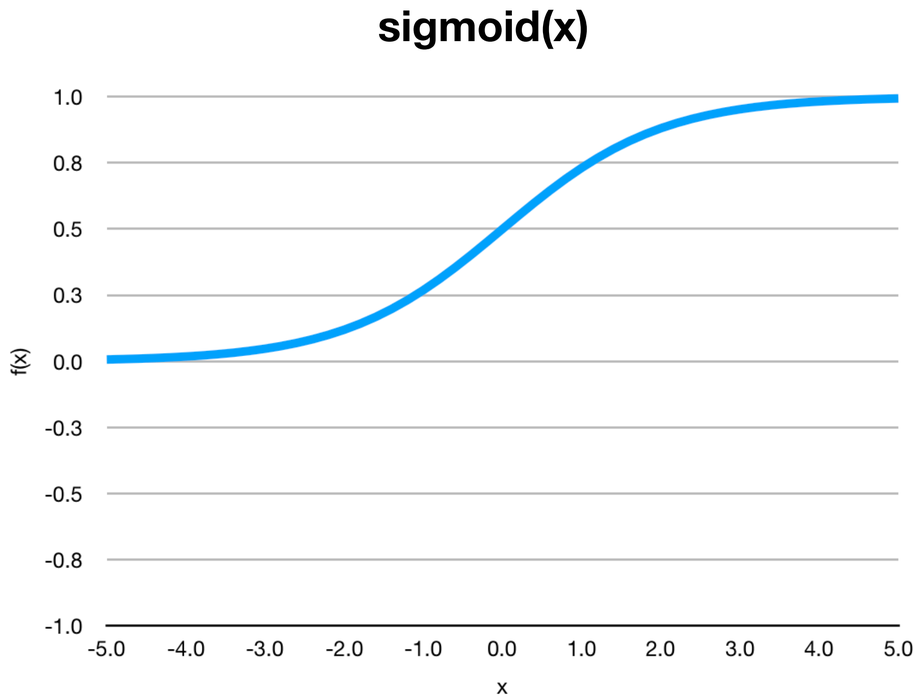
- softmaxの名前もここからきている

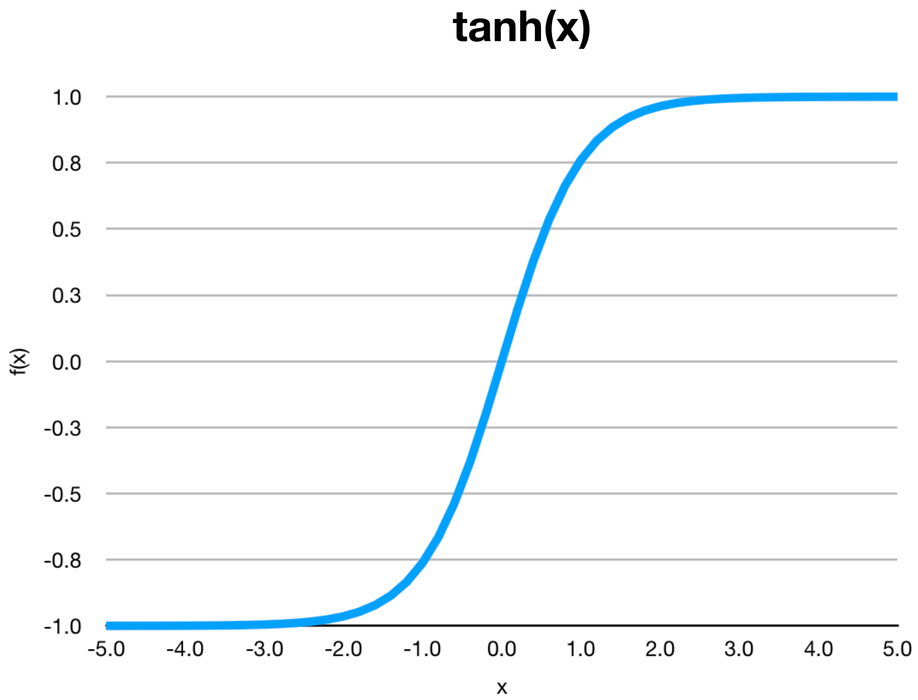
# 活性化関数
# （中間層、隠れ層）

# 隠れユニット

- ここで紹介するのは以下の3つ
  - シグモイドユニット（シグモイド関数）
  - ハイパボリックタンジェント(tanh)
  - ReLU
- 隠れ層での活性化関数は人間には直接関係ないので、とにかく高速に良い解にいけるようなものを選びたい

# Sigmoid, Tanh




$$y = \frac{1}{1+e^{-x}}$$

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

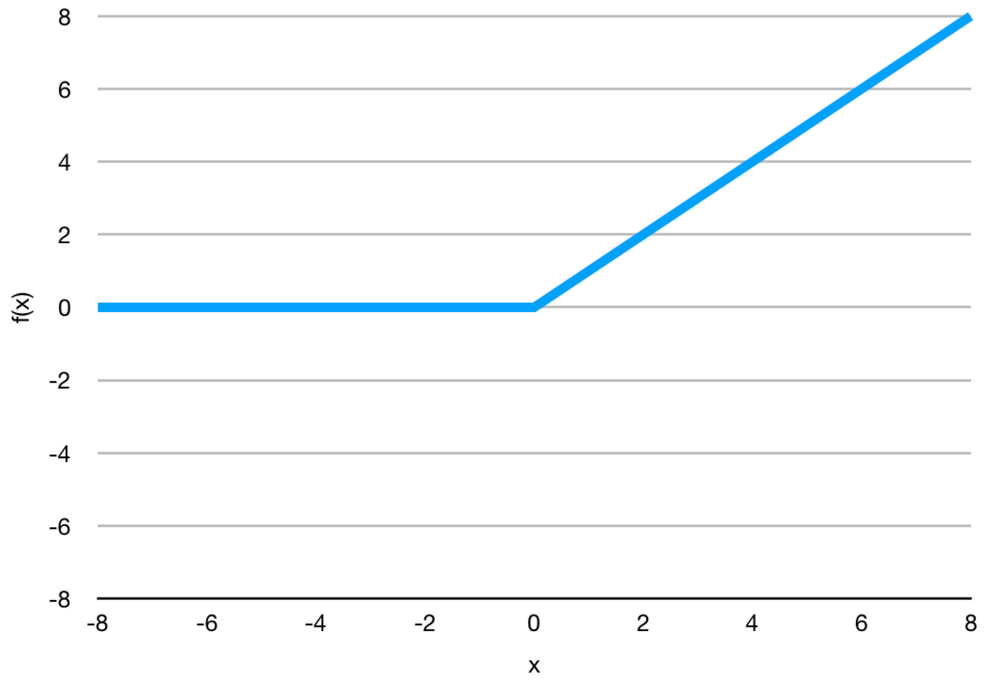
- どちらも非常に似ているが、実際 $\tanh(z) = 2\sigma(z)-1$
- tanhは-1〜1, sigmoidは0〜1の値をとる
- どちらも勾配が消えてしまうため、隠れユニットとしては好ましくない
- 0付近でも入力に敏感なためやはり最適化が難しい
- |x|が大きいと勾配が消えるし、原点付近は線形に近い

# 正規化線形関数

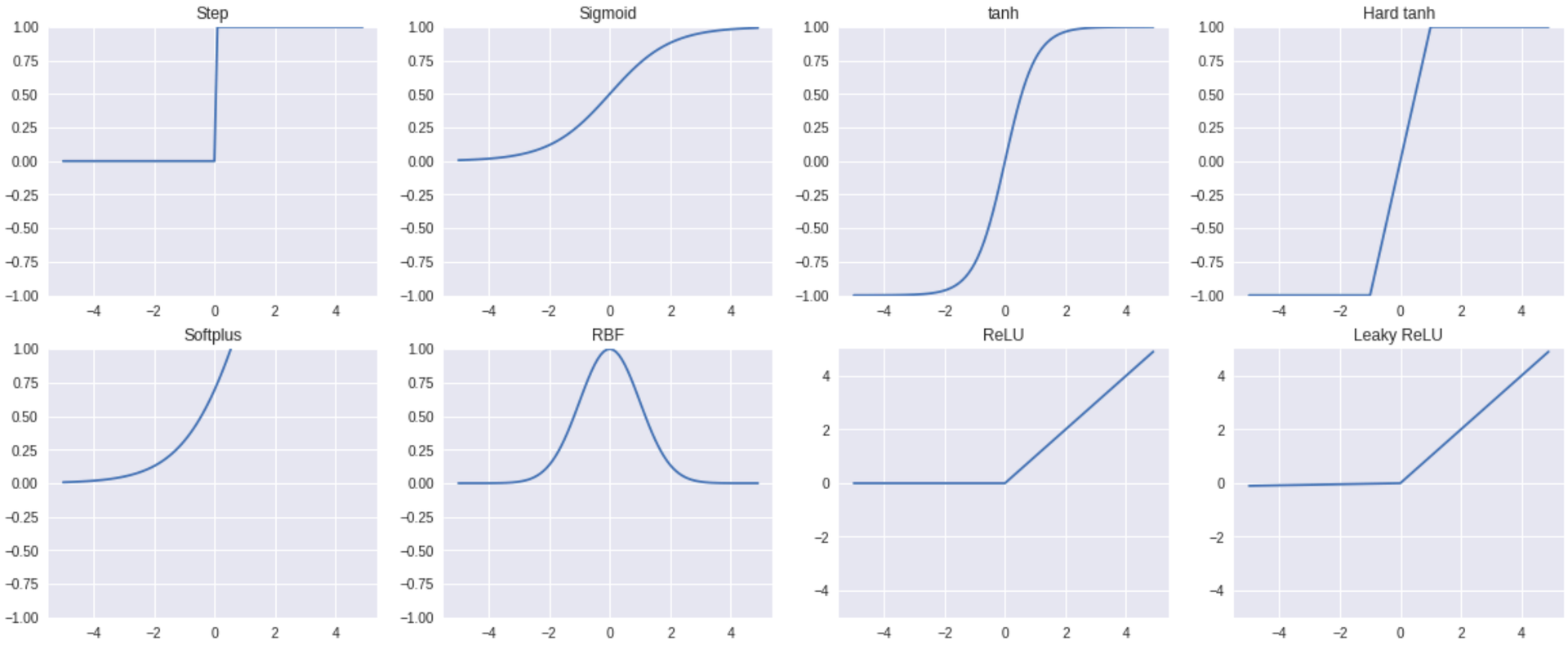
# (Rectified Linear Unit, Rectifier, ReLU)

- ランプ関数などとも呼ばれる

- g = max(0, z) → 0以下なら0,　それ以外はそのまま

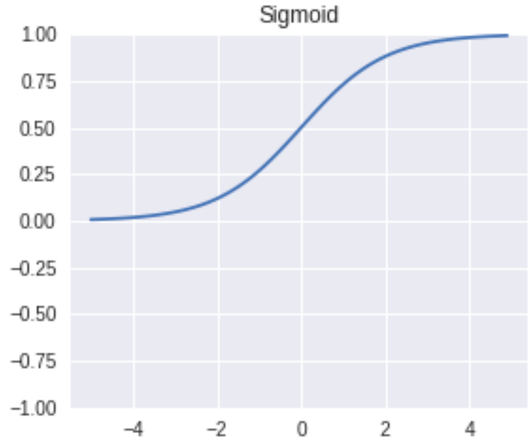


# 演習
# 出力ユニット・隠れユニットを実装



# 実装(numpy)

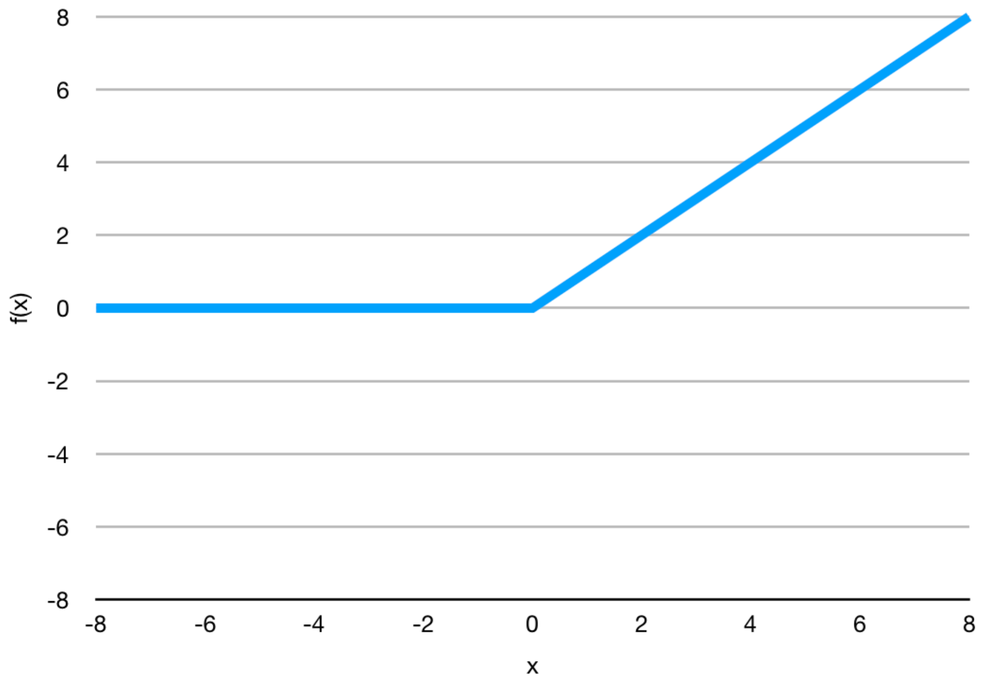
```python
import numpy as np
import matplotlib.pyplot as plt

n_cols = 4
n_rows = 2

plt.figure(figsize=(20, 8))

x = np.arange(-5, 5, 0.1)

#
# Sigmoid
#
y = 1./(1. + np.exp(-x))
plt.subplot(n_rows,n_cols,1)
plt.title('Sigmoid')
plt.ylim(-1,1)
plt.plot(x, y)
```




**フレームワークを使う場合は、例えばTensorflowなら単に tf.sigmoid(x)**

# ReLU

- 勾配が消えない
- 計算コストが低い
- 性能が良い



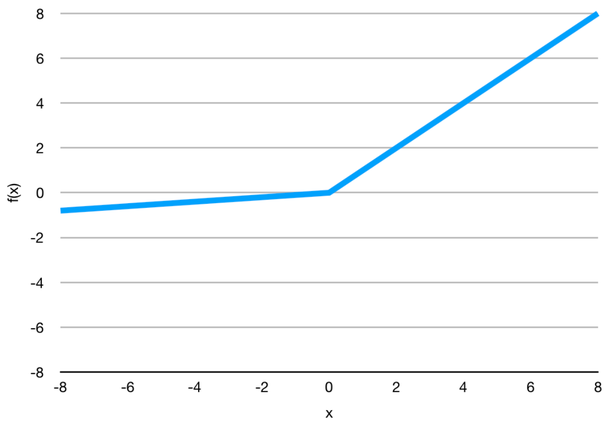
# ReLUの微分

- ReLUは非連続な部分があり微分可能ではない
- ReLUの場合だと x = 0 のとき微分できない
- …ので、劣微分(subdifferential)を考える
- 結論としては単に次のような定義でよい

$$\frac{\partial f}{\partial x} = \begin{cases} 1 & (x \geq 0) \\ 0 & (x < 0) \end{cases}$$

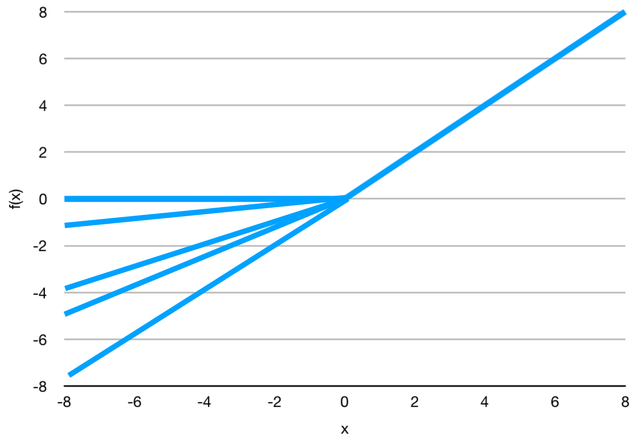
# ReLU一族

## Leaky ReLU



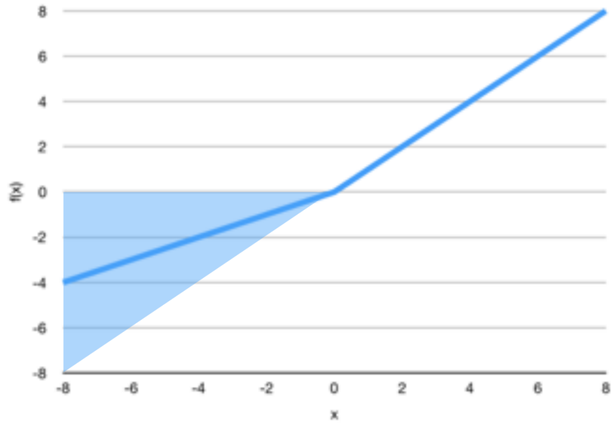

負側にも傾斜をつける

## ReLU




ランダムに傾斜させる
(学習時はランダム、
認識時は平均)

## Parametric ReLU (PReLU)




傾斜具合も学習する

# その他の隠れユニット

# (RBF, Softplus, Hard Tanh)

- RBF(Radial Basis Function)

$$\exp(-1/\sigma^2 \| W - x \|^2)$$

- Softplus

$$\log(1+e^x)$$

- Hard Tanh

$$\max(-1, \min(1,x))$$

などなど

- いずれの手法も最適化が難しく、今は使われていない

# アーキテクチャの設計

# 万能近似定理
# (Universal Approximation Theorem)

- 定数でない、連続な有界単調増加の関数を隠れ層の出力に使った3層以上のニューラルネットワークは、任意の連続関数を任意の精度で表現可能

- 簡単に言えば、非線形な隠れユニットを使っていれば、(Deepでなくても)3層のネットワークでどんな問題でも解けるということ

- ただし、隠れ層のユニット数が指数的に増加する問題が存在することが知られている

- n入力の3層のニューラルネットワークを考えた時、最悪$2^n$の隠れユニットが必要だが、これは現実的ではない
  - 例: MNIST(784次元)→1.0e+236個
  - 計算量的に不可能 or 過学習する
- また、ネットワークが問題を表現可能であるとはいっても、それを学習可能かどうかは保証されない
- とはいえ、多層にすることでこれらの問題は緩和可能で、隠れユニットを減らしたり汎化性能をあげたりすることができる