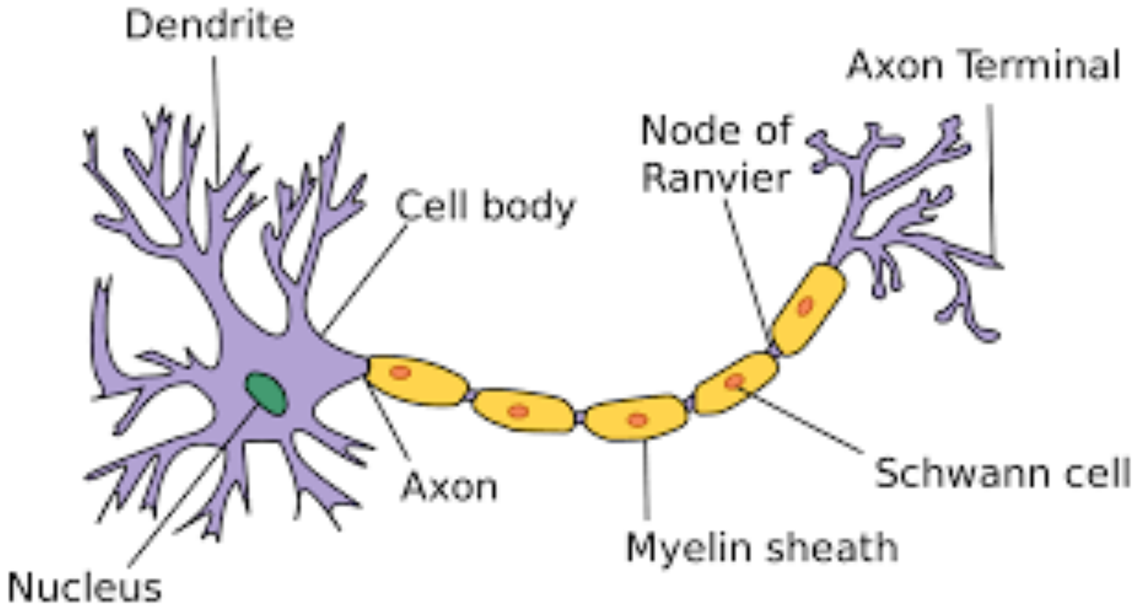
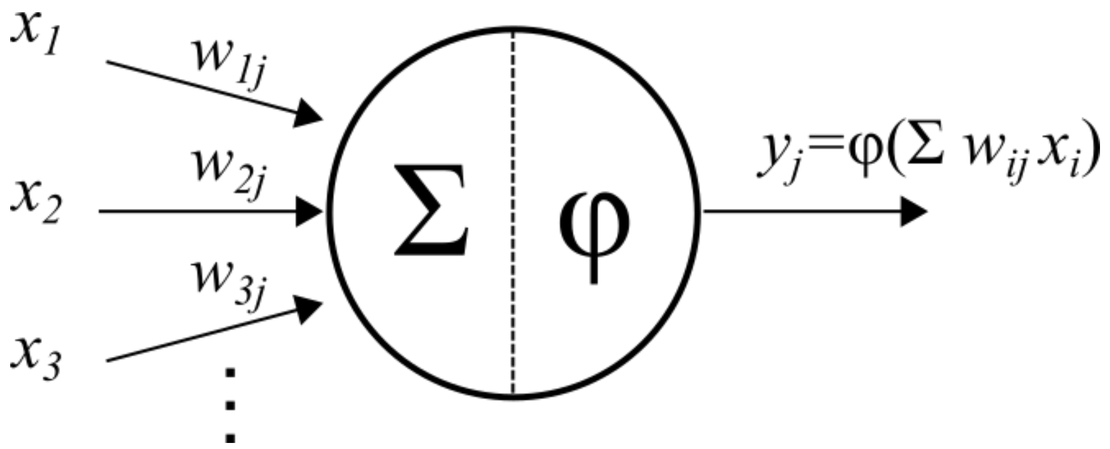
深層学習

Feed Forward Network - 順伝播型ネットワーク

ニューロン
Neuron

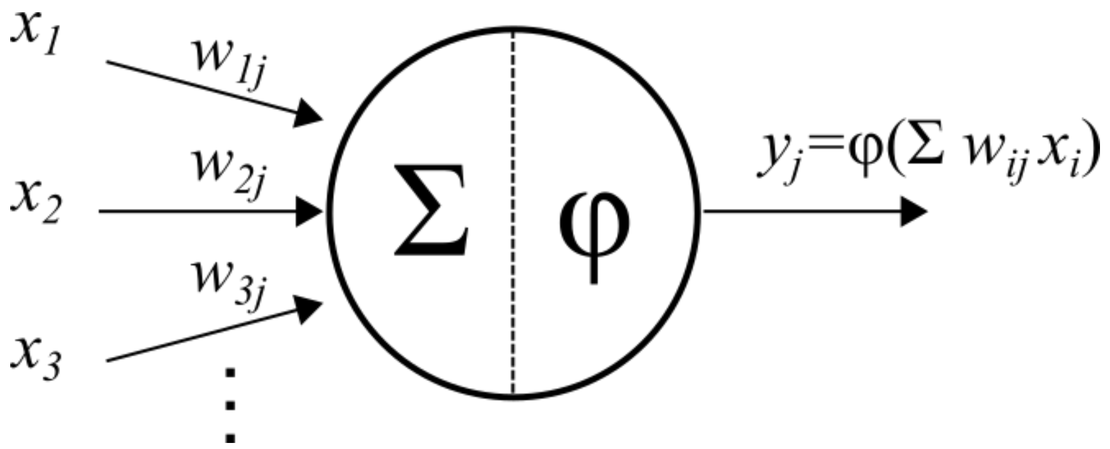
ニューロンと人工ニューロン



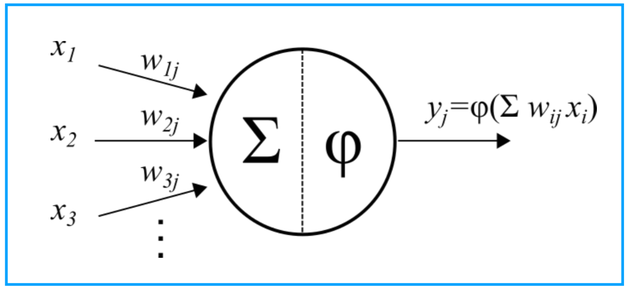


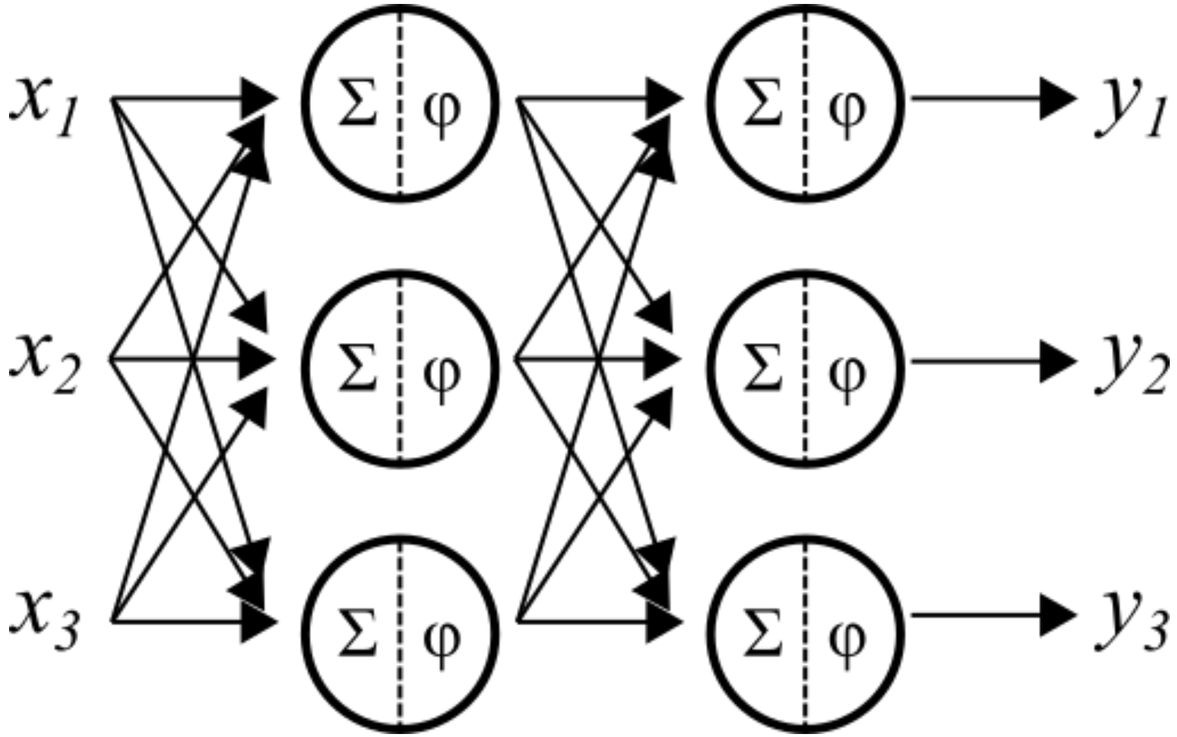
ニューロンと人工ニューロン

- 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
- ここでは省略しているがバイアス項bもある



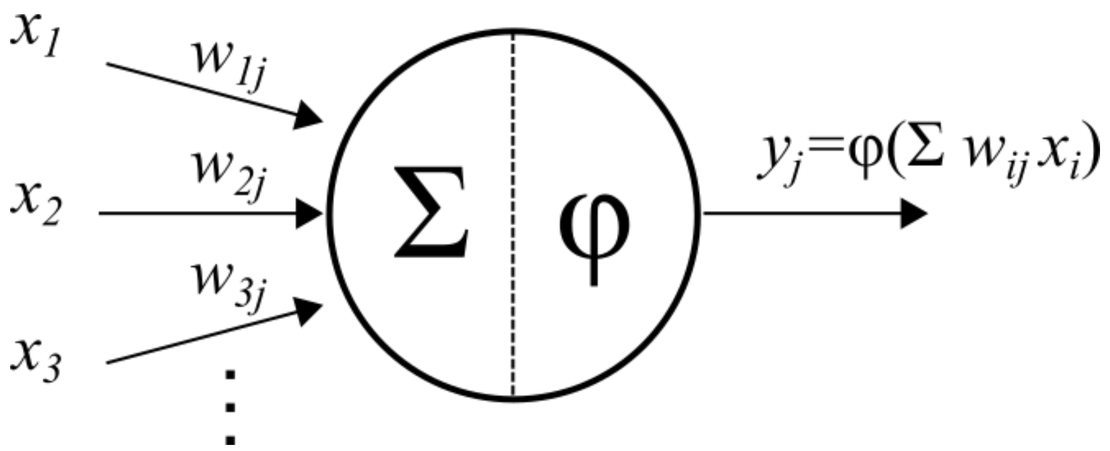
ニューラルネットワーク(feedforward)





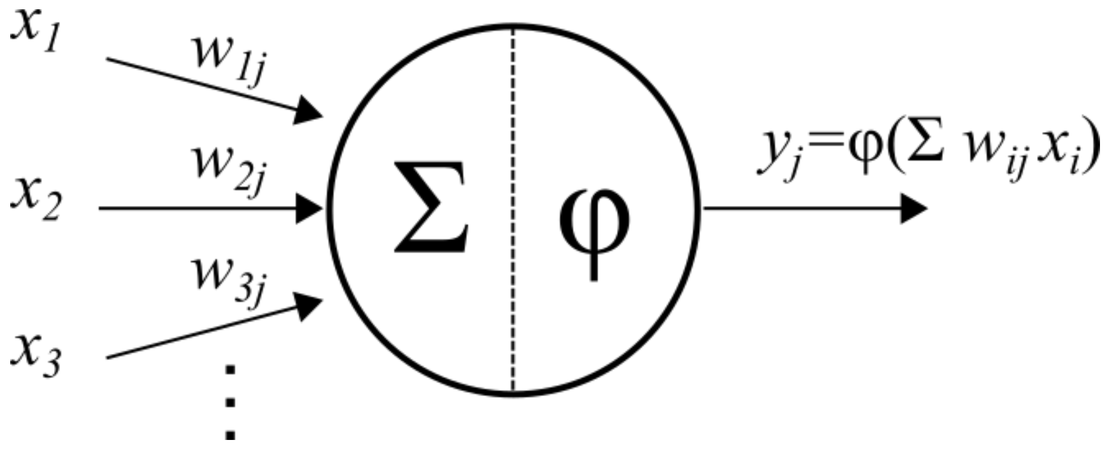
なぜ活性化関数？

- 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

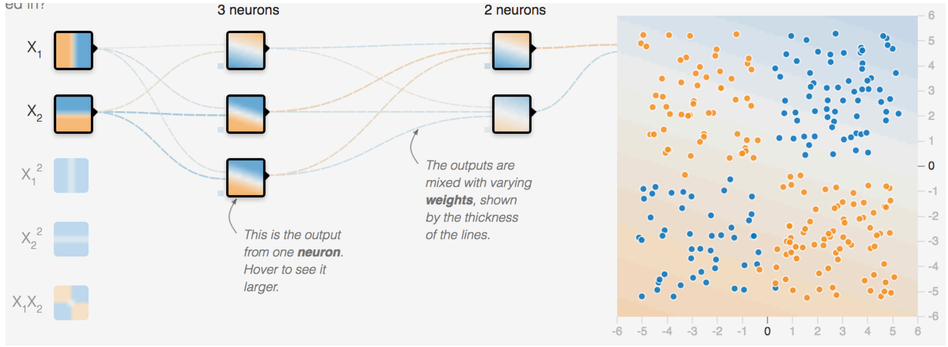


なぜ活性化関数？

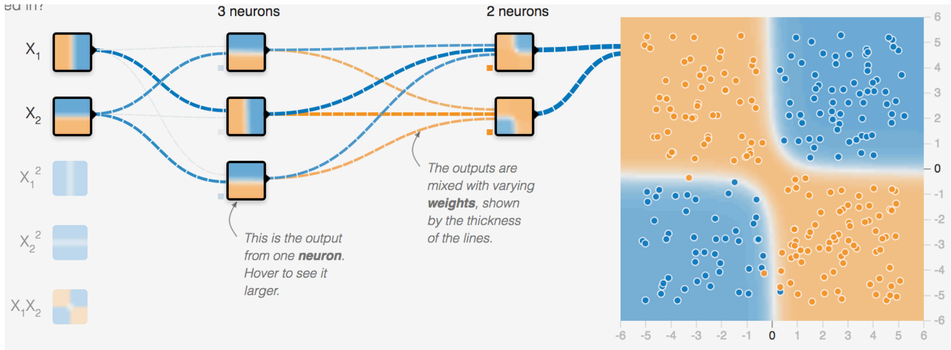
- 活性化関数を通してモデルに非線形性を与えることができる
- より複雑で自由度の高いモデルを作ることができる
- しかし学習は非常に難しくなる…



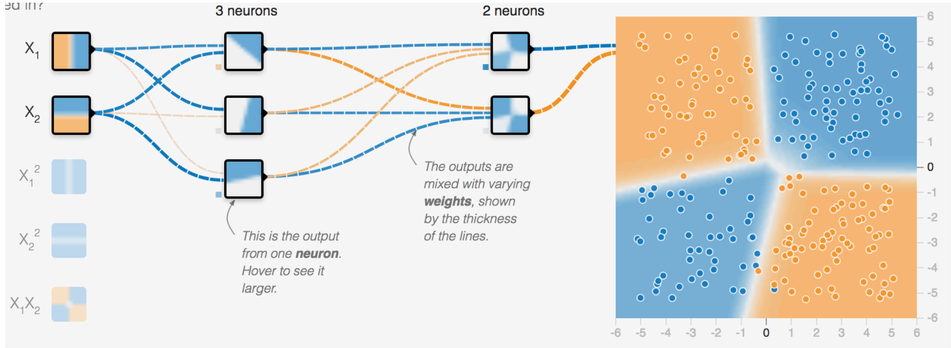
活性化関数による振る舞いの違い



Linear



Tanh

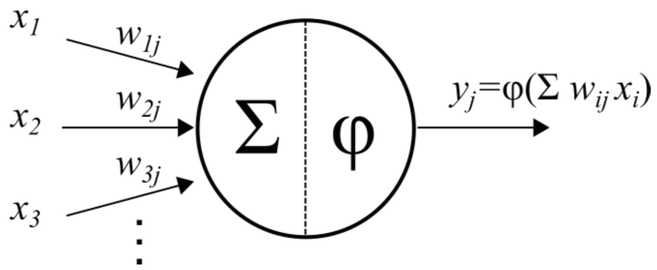


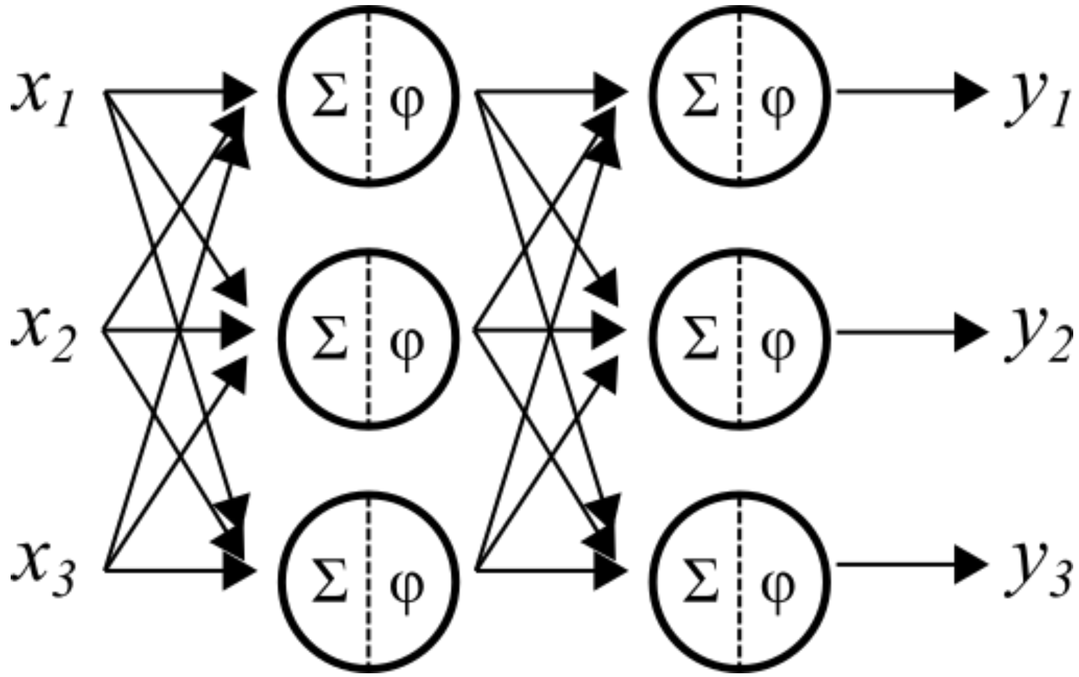
ReLU

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





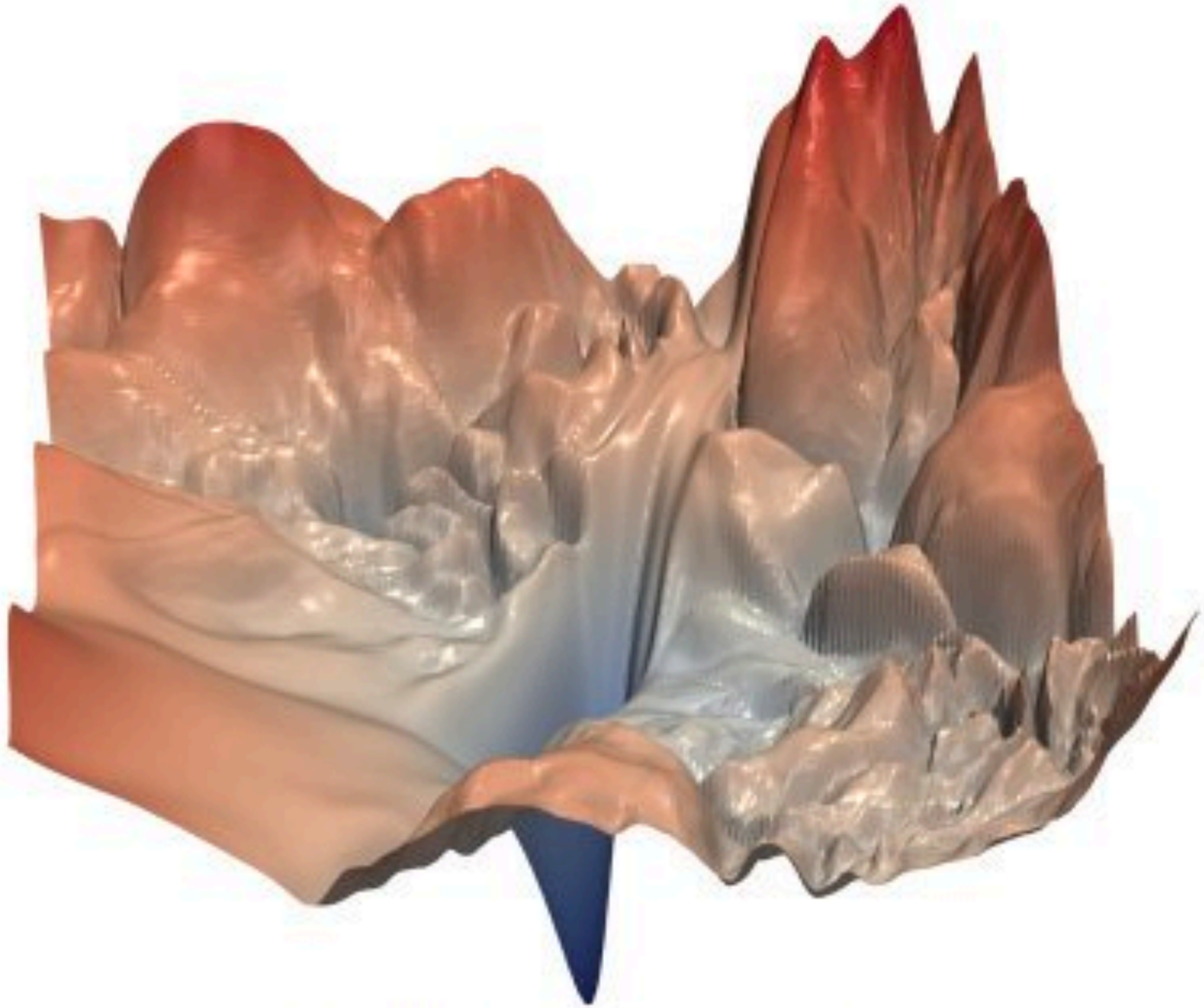
パラメータを求める

- 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

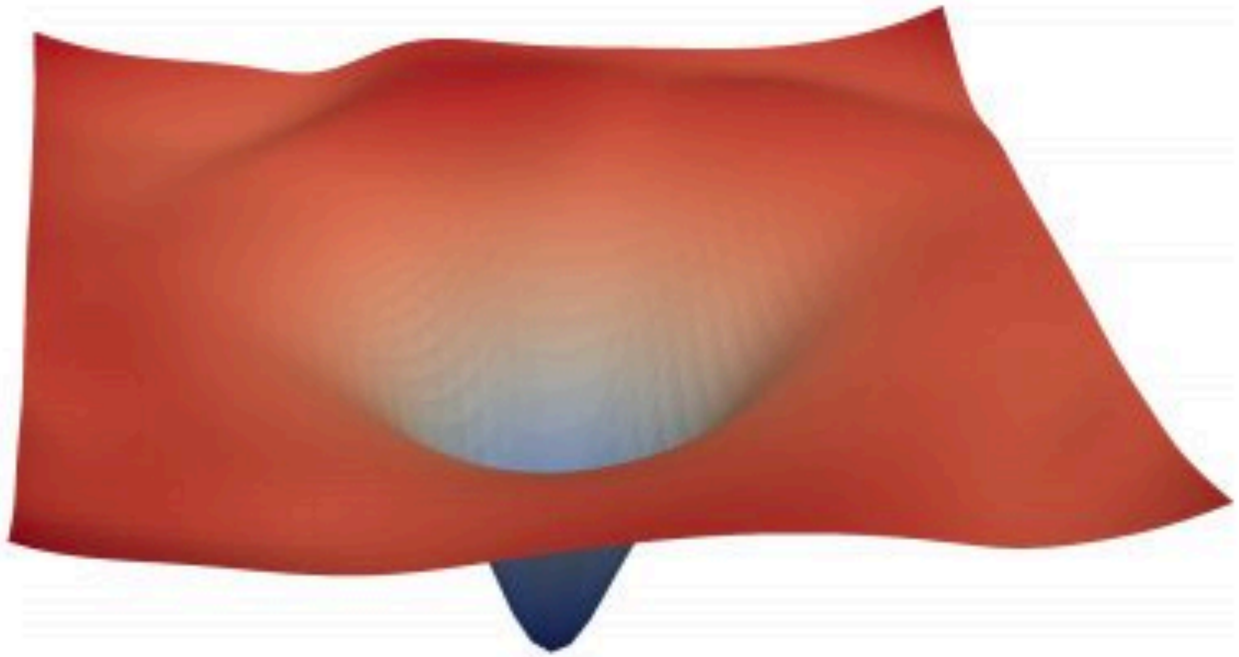
パラメータを求める

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

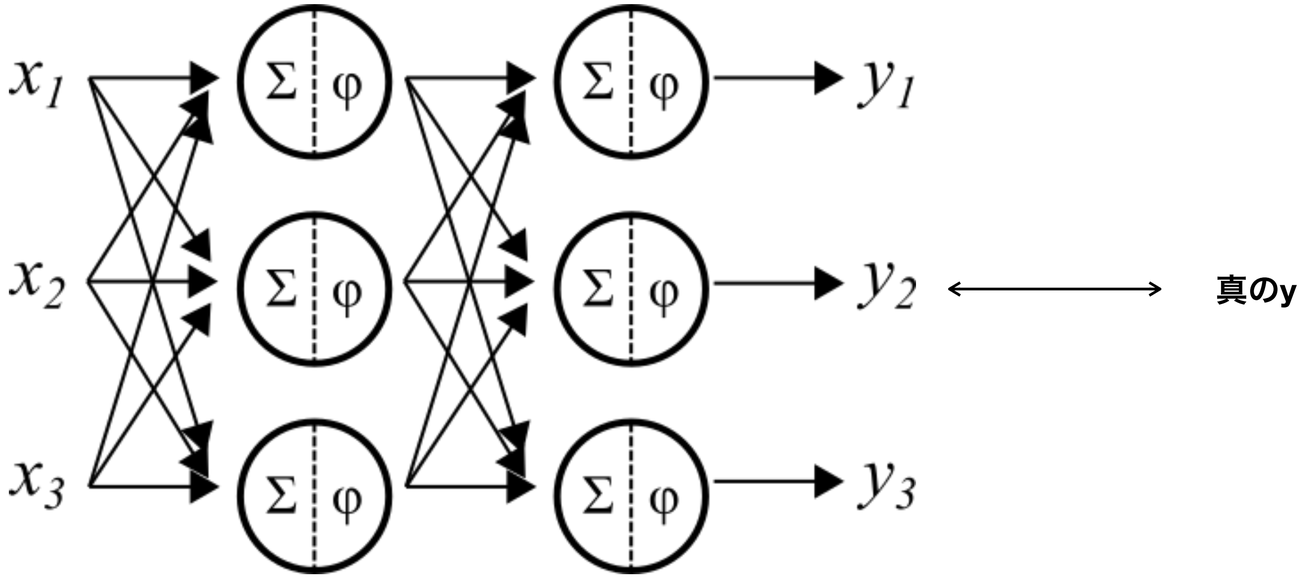
コスト関数（損失関数）

- 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

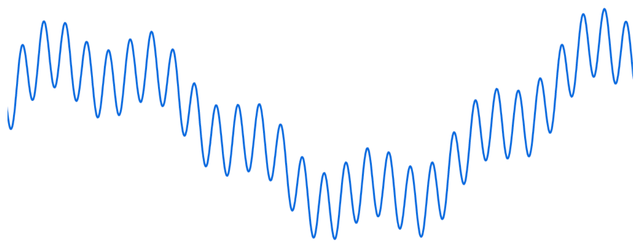
- 正解の値とNNが出した値を比べる
- どれくらい間違っているのか(損失)を求める
- 間違いが少なくなるようにすこしだけWを増減させる

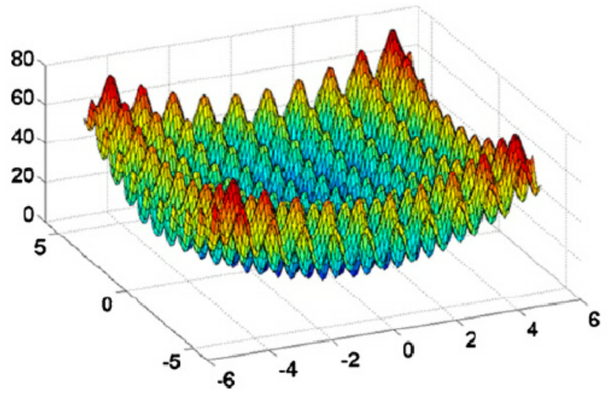


NNの最適化

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

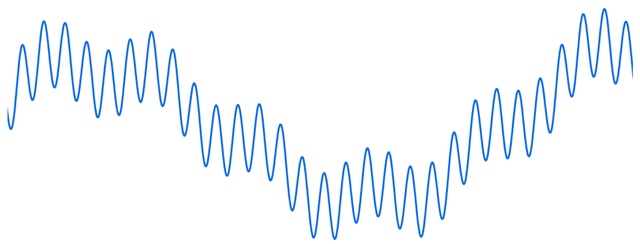


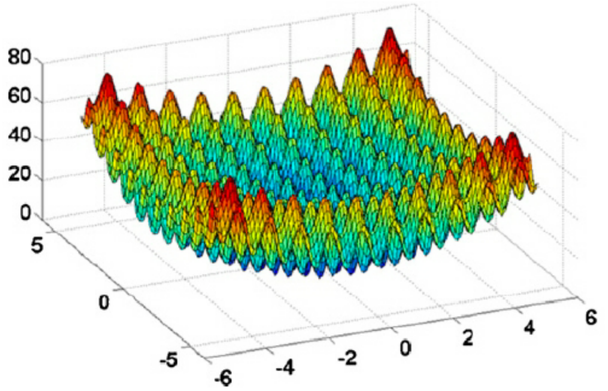


連続で微分可能だが非線形非凸な関数のイメージ図

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

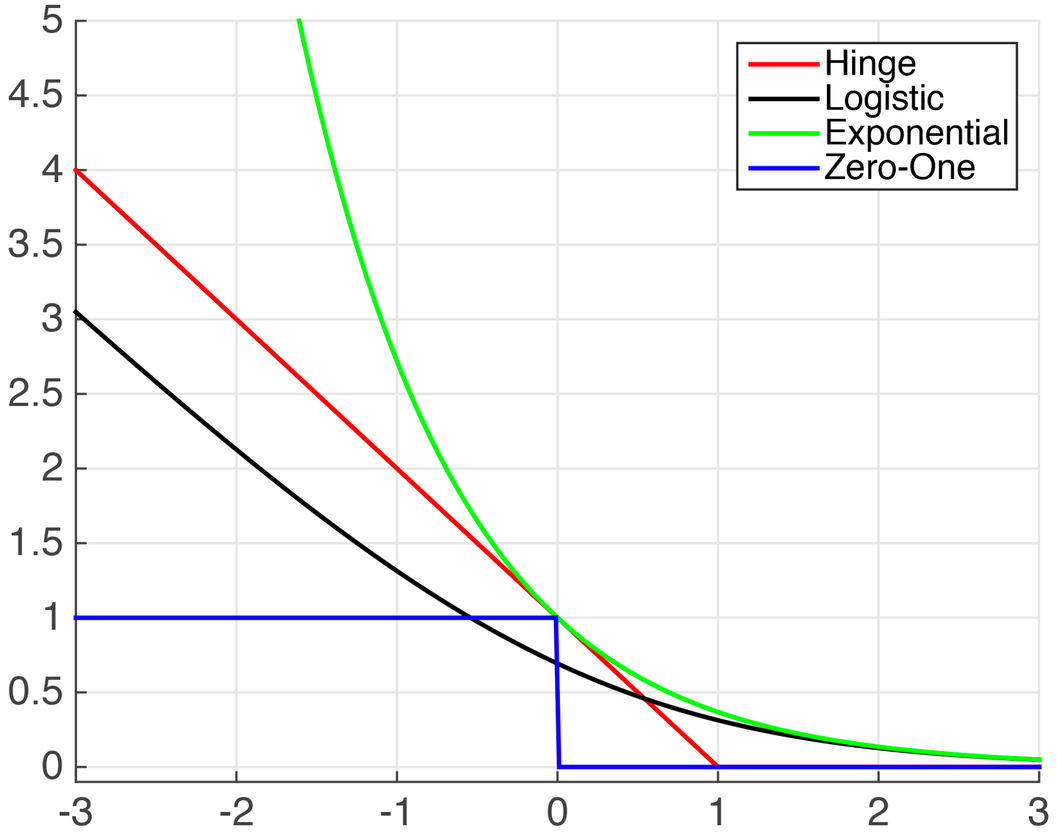




連続で微分可能だが非線形非凸な関数のイメージ図

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
  - 例) Cross entropy



Q. 0-1損失がダメな理由

- 今の状態がどれくらい良いかわからない
- 例
  - 金庫破りをしよう
  - ダイヤルが1つあって、これをクルクル回して当たりの数字を探す
  - この金庫はダイヤルを回すたびに「悪さ」が得られる
- 0-1損失は「合ってる」「間違ってる」としか返してくれない
- 「どれくらい悪いか」も教えてほしい…
- 「次は時計回りと反時計回り、どっちに回せばいいか」も知れたらもっと良い
  - →勾配が得られるということ

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ϕ、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）

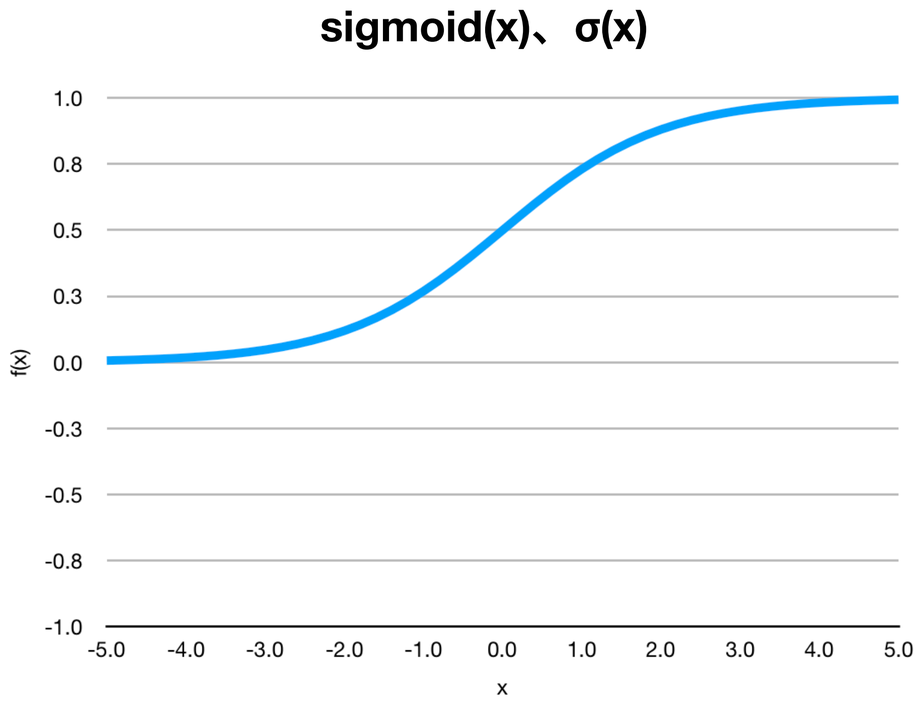
線形ユニット(linear)

- NNの出力をそのまま欲しいときに使う
- 基本的には回帰
  - 株価予測
  - 気温予測

シグモイドユニット(sigmoid)

- 0〜1に正規化して出力したいときに使う
- たとえば確率で欲しいとき
- このような性質の関数は他にもあるがSigmoidは以下から便利
  - 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
  - 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

Sigmoid



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

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)

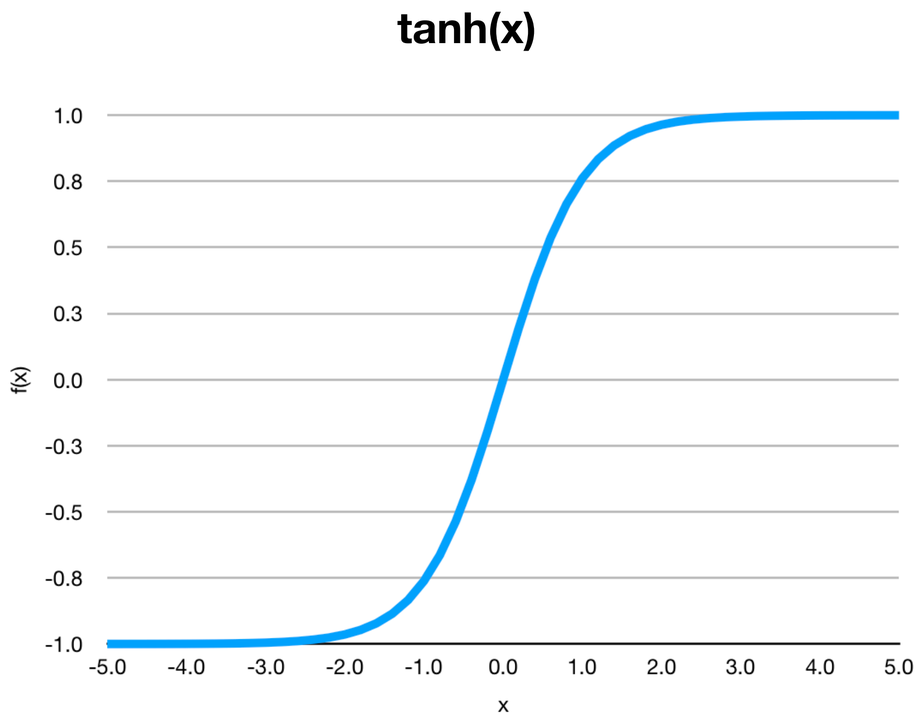
ソフトマックスユニット

- 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

Tanh



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

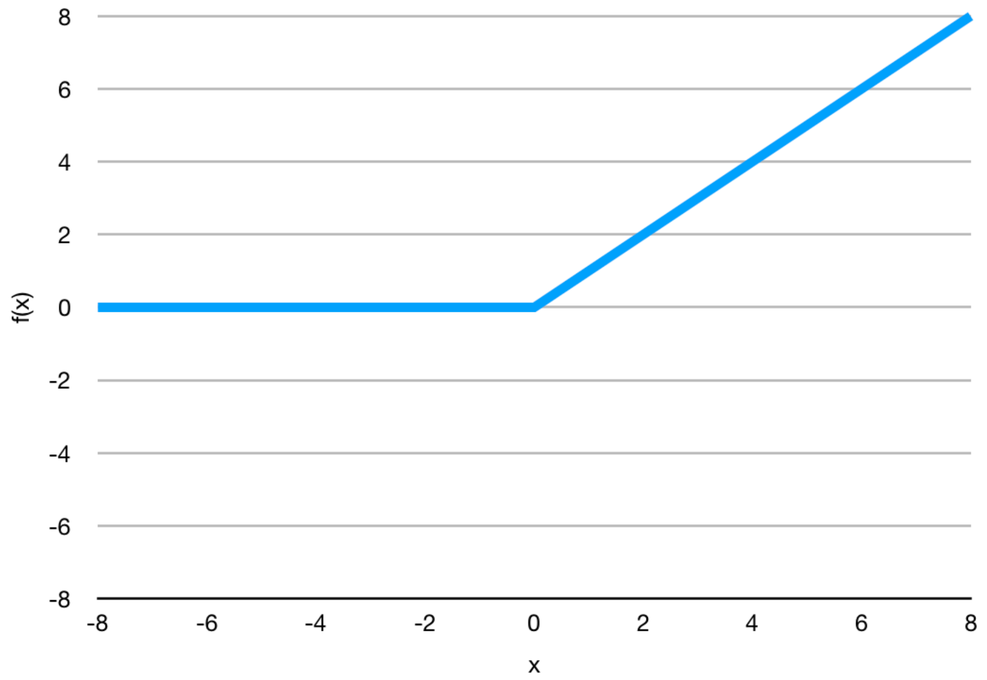
- どちらも非常に似ているが、実際 $\tanh(z) = 2\sigma(z)-1$
- tanhは-1〜1, sigmoidは0〜1の値をとる
- どちらも勾配が消えてしまうため、隠れユニットとしては好ましくない
- 0付近でも入力に敏感なためやはり最適化が難しい
- $|x|$が大きいと勾配が消えるし、原点付近は線形に近い
- Sigmoidは慣例的に$\sigma(x)$と表すことが多いので覚えておいてください

正規化線形関数

(Rectified Linear Unit, Rectifier, ReLU)

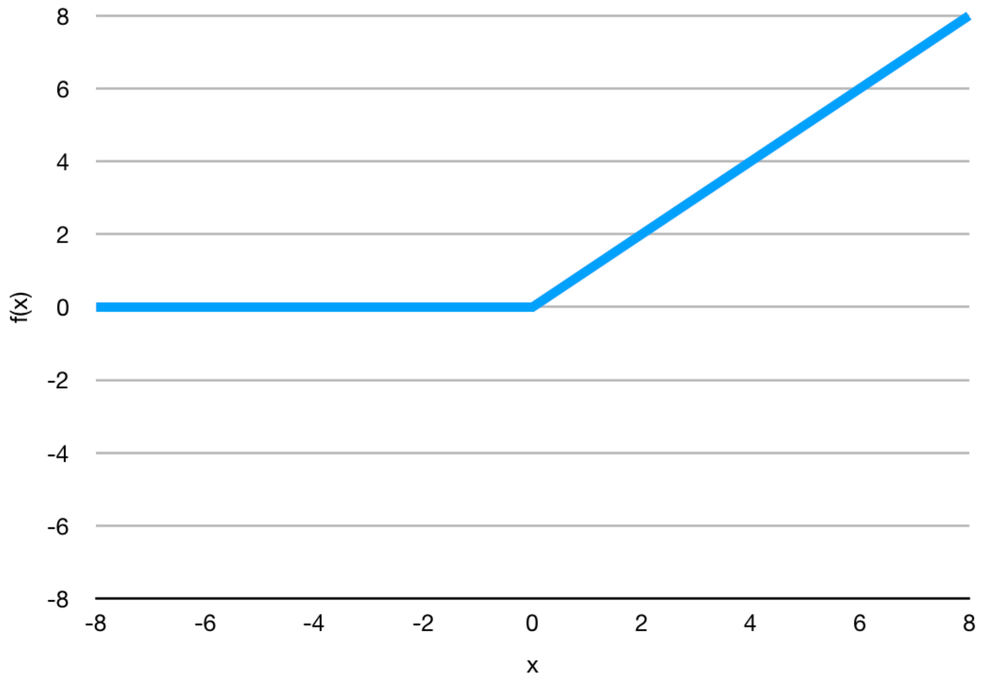
- ランプ関数などとも呼ばれる

- g = max(0, z) → 0以下なら0,　それ以外はそのまま



ReLU

- 勾配が消えない
- 計算コストが低い
- 性能が良い
- 非線形性が強い



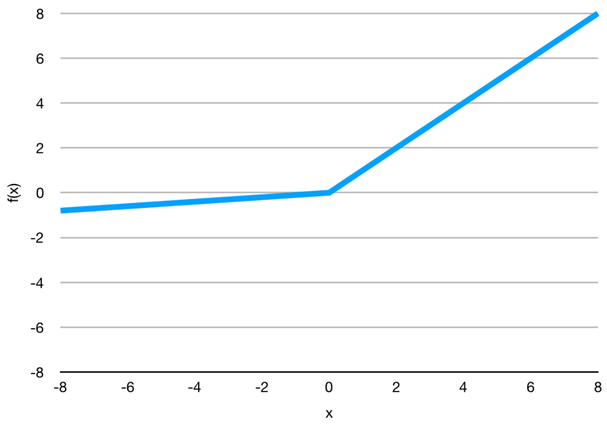
ReLUの微分

- 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

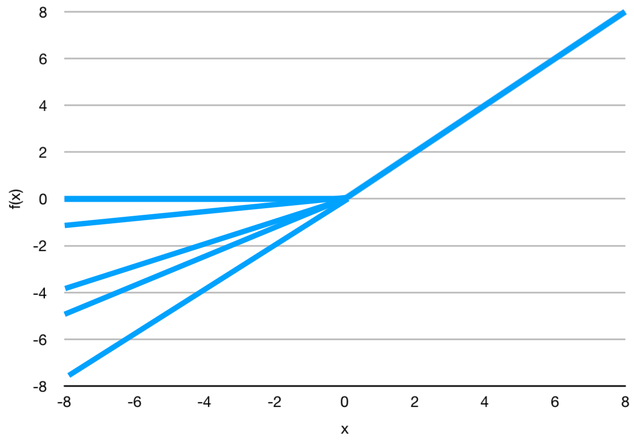
Leaky ReLU



負側にも傾斜をつける

※稀に使うことがある

RReLU



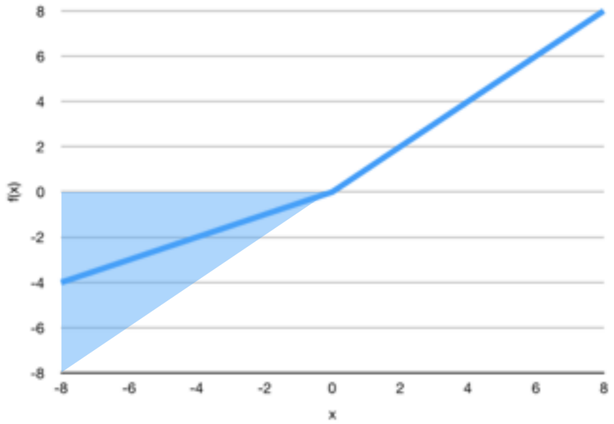
ランダムに傾斜させる

学習時はランダム、

認識時は平均

※今は使わない

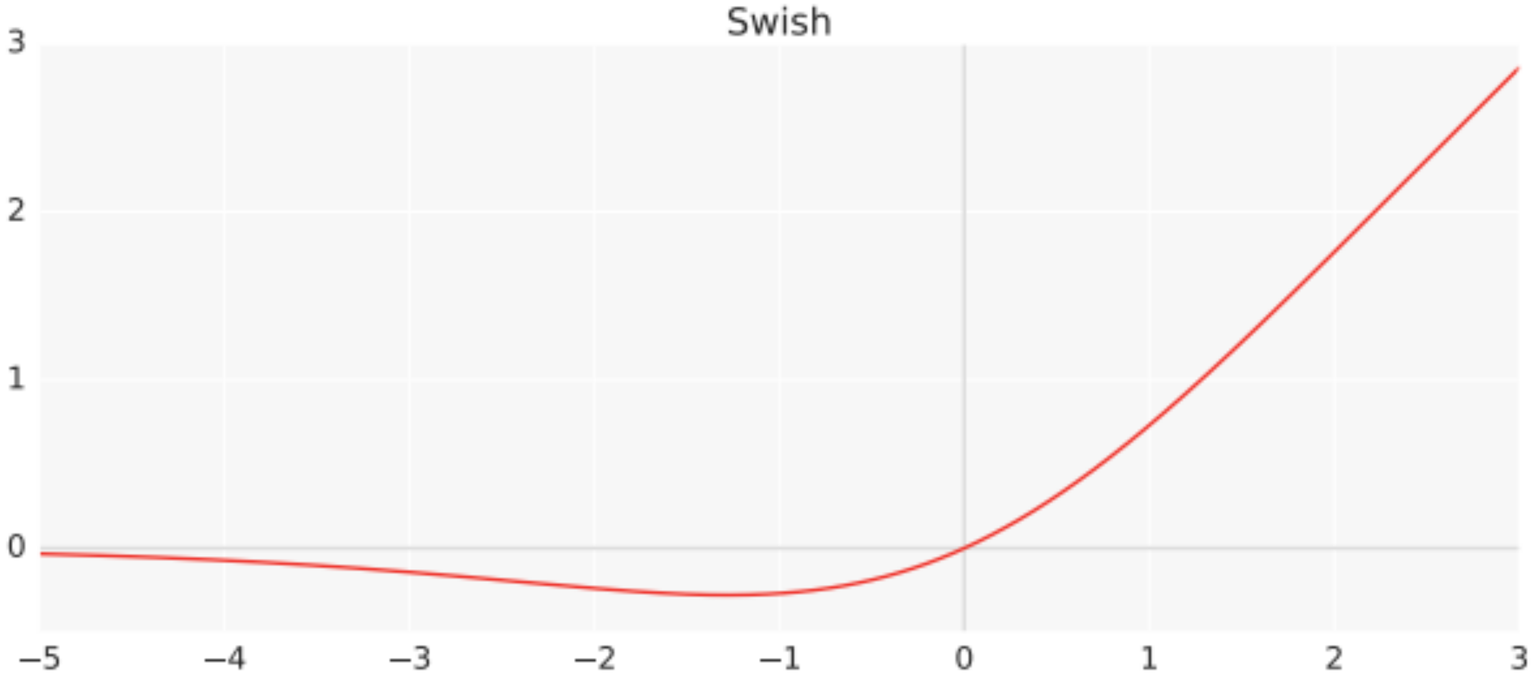
Parametric ReLU (PReLU)



傾斜具合も学習する

※今は使わない

Swish
(Self-gated Activation Function)

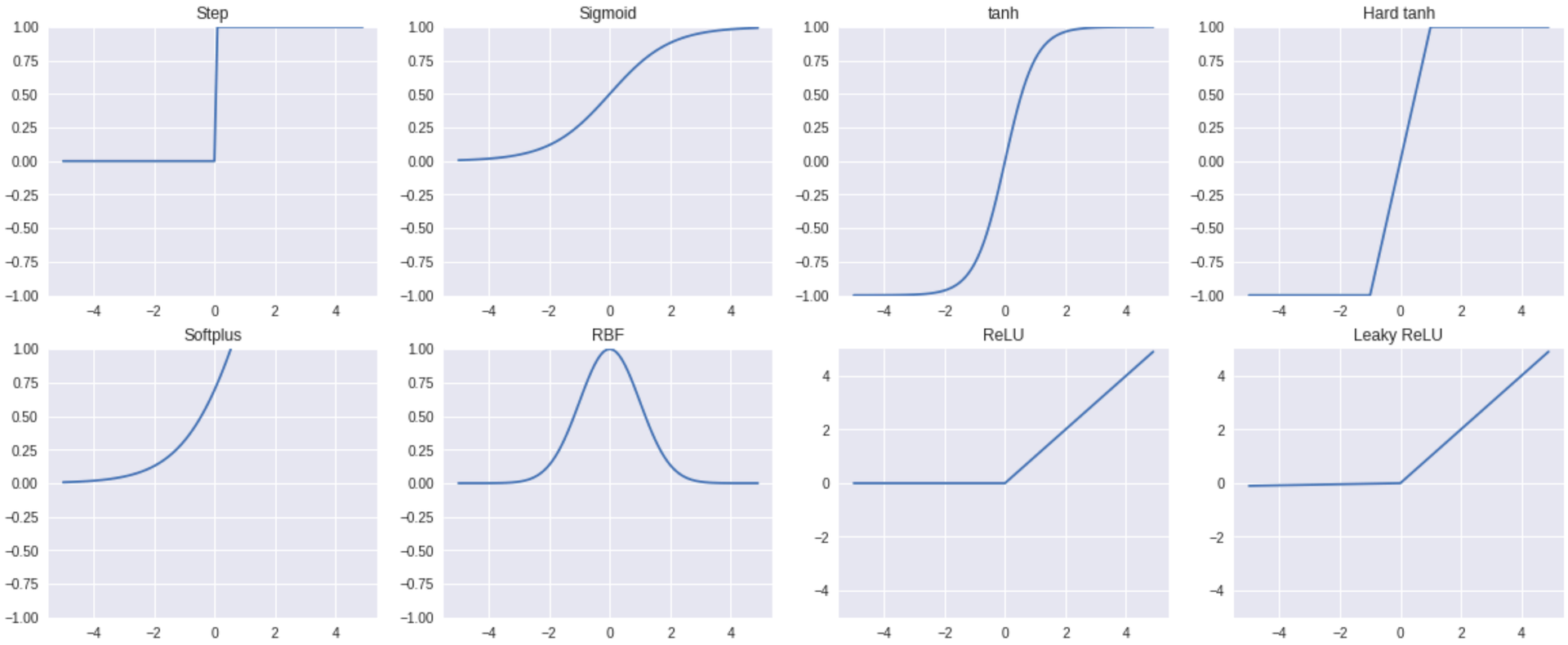


f(x) = x・σ(x)

最近少しずつ流行っている

経験的に性能が良いとされている

いろいろな出力ユニット



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

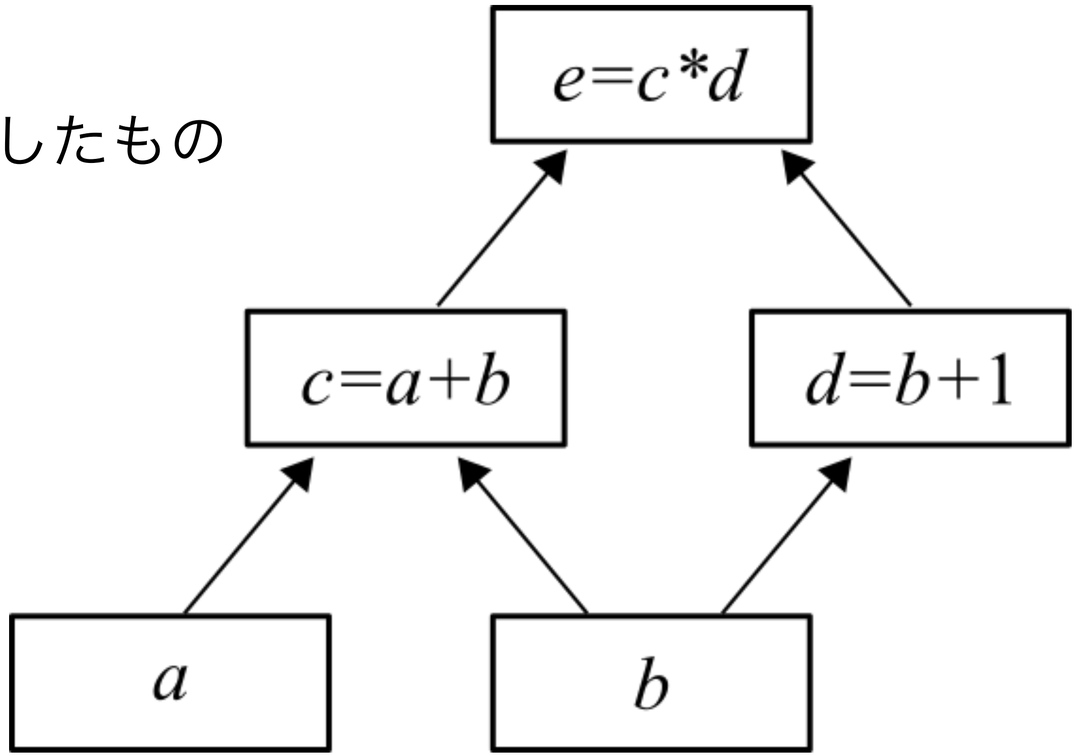
- n入力の3層のニューラルネットワークを考えた時、最悪2^nの隠れユニットが必要だが、これは現実的ではない
  - 例: MNIST(784次元)→1.0e+236個
  - 計算量的に不可能 or 過学習する
- また、ネットワークが問題を表現可能であるとはいっても、それを学習可能かどうかは保証されない
- とはいえ、多層にすることでこれらの問題は緩和可能で、隠れユニットを減らしたり汎化性能をあげたりすることができる

誤差逆伝播法

Backpropagation

計算グラフ

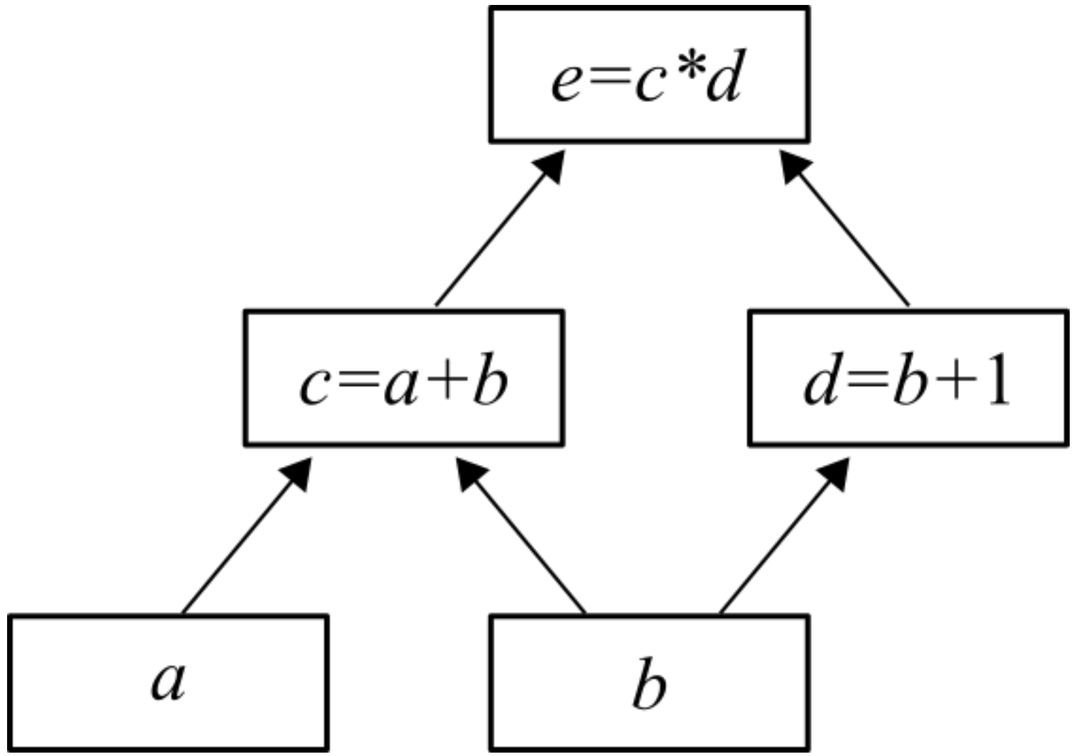
- これは下の数式を表現したもの
  - c=a+b
  - d=b+1
  - e=c*d



誤差逆伝播法を考える上で非常に重要

計算グラフ

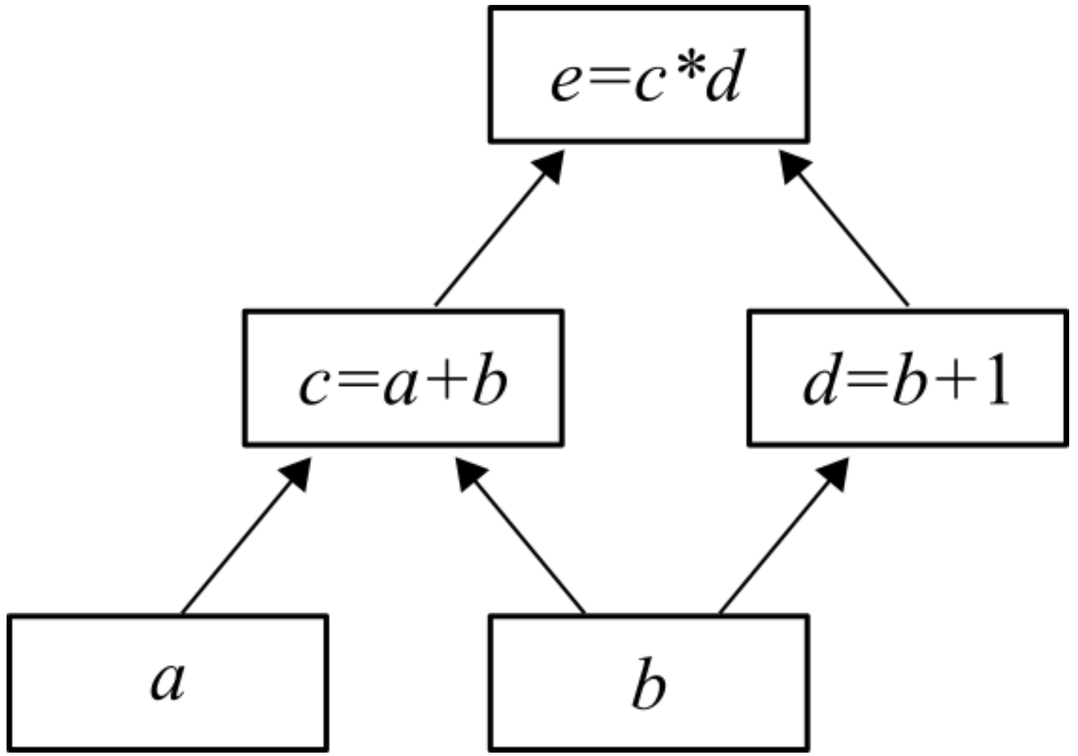
- a=2, b=1 とすると、
- c=3
- d=2
- e=6



誤差逆伝播法を考える上で非常に重要

Deep Learning における計算グラフ

- このような式をネットワークだと見立ててみる
- 重みの修正はaやbの値を少し変更することに相当する
- 今知りたいのは、aやbを少し変更したとき、eがどれくらい変化するか
- つまり微分したいが、ネットワークが大きいと計算が難しい



なんとか効率よく微分できないだろうか？

連鎖律

f(y), y(w) のとき

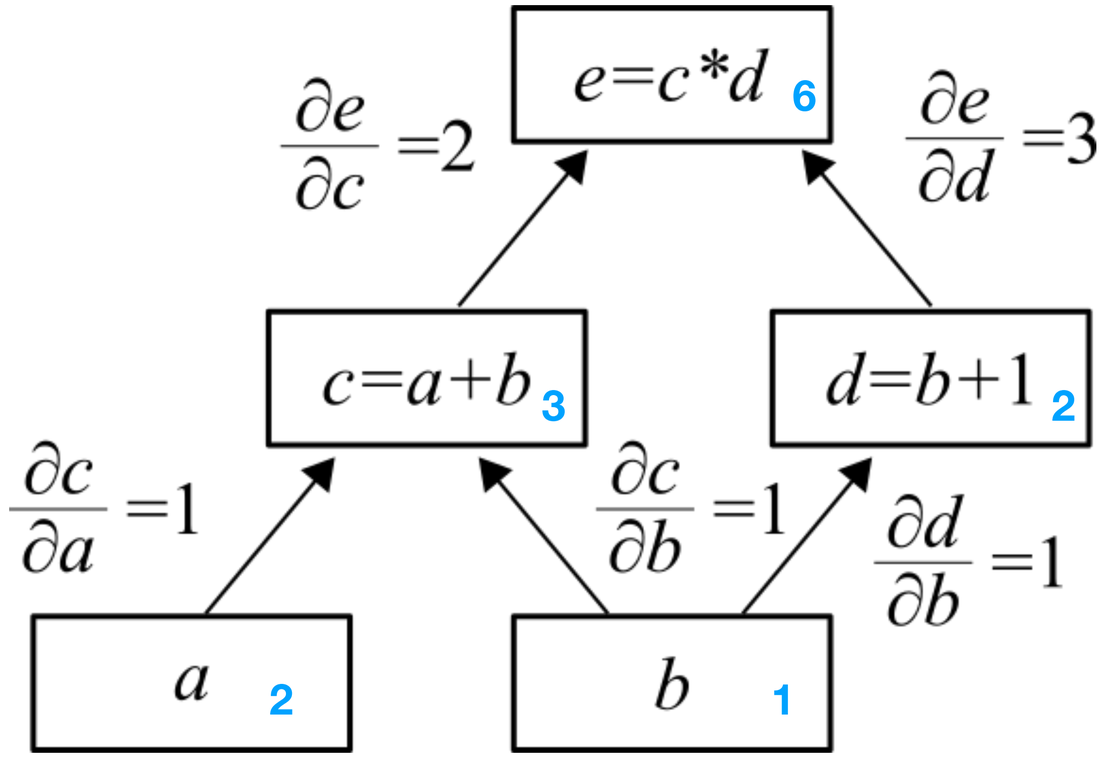
$$\frac{\partial f}{\partial w} = \frac{\partial f}{\partial y}\frac{\partial y}{\partial w}$$

f(y_1, y_2, …, y_k), y_k(*w*) のとき　(= 多変数関数の場合)

$$\frac{\partial f}{\partial w} = \sum_{k=1}^{K} \frac{\partial f}{\partial y_k}\frac{\partial y_k}{\partial w}$$

連鎖律

- 計算グラフの偏微分の値を求めるには？
  - たとえばbを少し変えたとき、eはどれくらい変わる？
- 直接つながっているノード間であれば簡単
- しかし例えば∂e/∂bは？
- 実は単に足していくだけで良い
  - ∂e/∂b=1*2+1*3=5

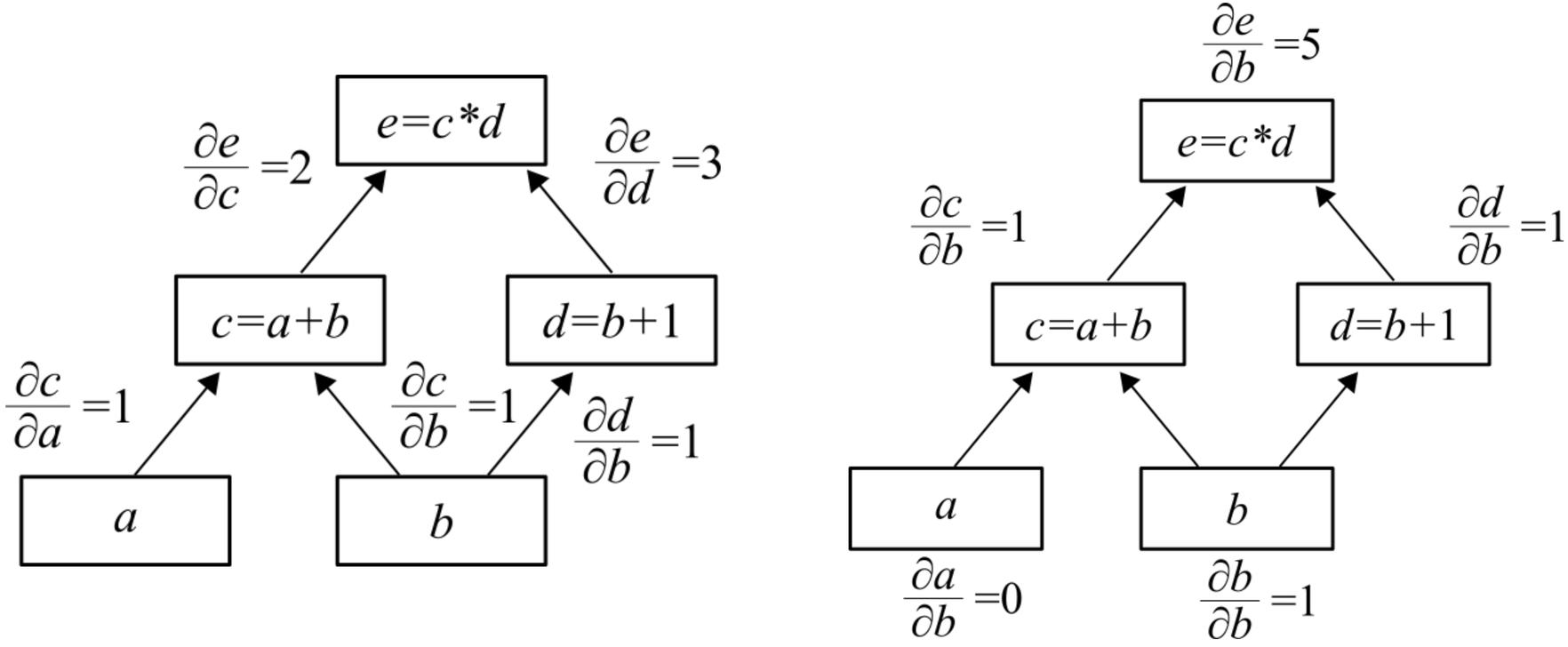


ところで
なぜ偏微分するのか

- 損失関数をfとして、すべてのパラメータ(重み)に関する偏微分を計算して並べたベクトルを**勾配**とよぶ。
- 勾配がわかれば、「損失が小さくなるようにパラメータを少し動かす」という計算ができる。
- もうちょっと砕けた言い方をすると、あるパラメータをどう増減させるとより損失が少なくなるか？をすべてのパラメータについて知りたい
- これを**効率的にやる方法が誤差逆伝播法**がやっていることである。
  - 計算グラフのようなものを考えず(誤差逆伝播法を使わず）に、もっとシンプルな方法で微分する方法もある（→数値微分）が、小さいネットワークならまだしも、計算コストが高すぎて非現実的

Forwardモード微分

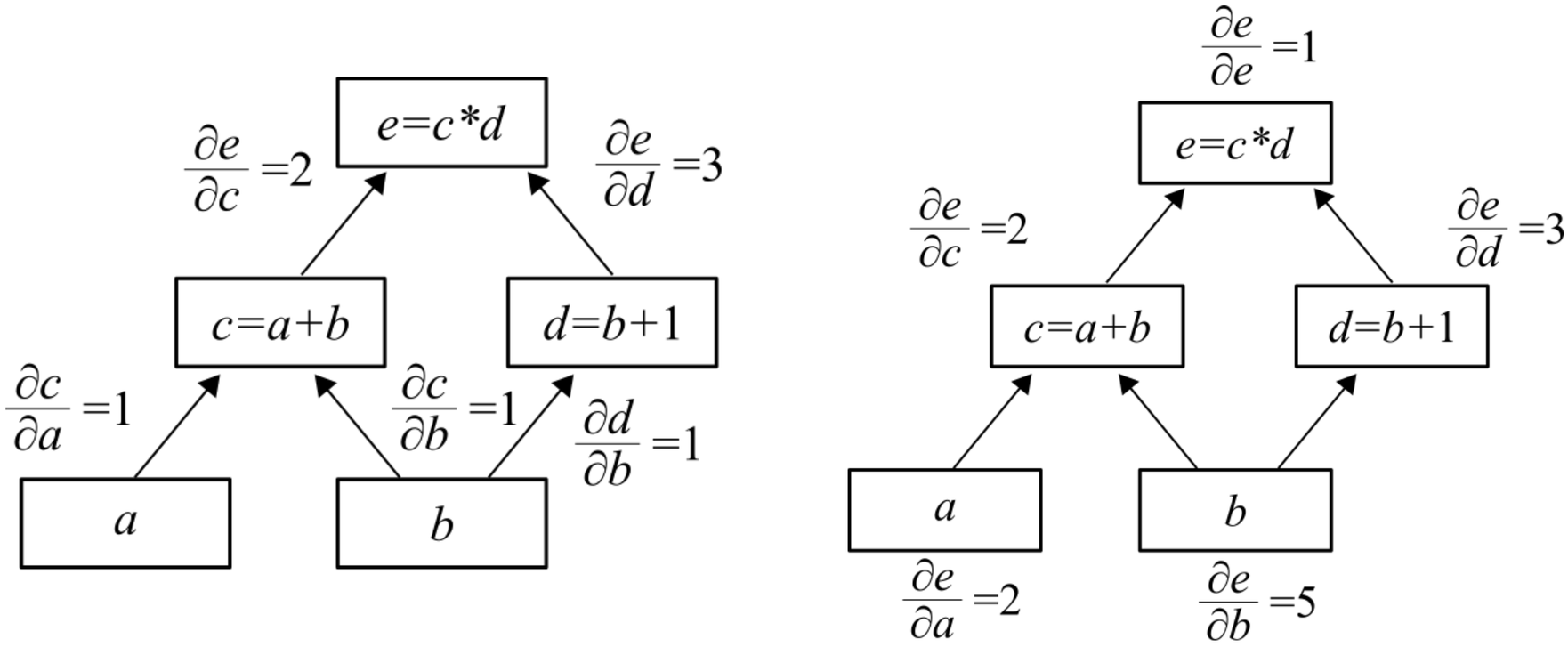
a=1, b=2 のとき



ある入力を変化させた時、どのように出力が変化していくかを見る方法

Reverseモード微分

a=1, b=2 のとき



ある出力を変化させた時、グラフの各点がどのように作用しているのかを見る方法

誤差逆伝播法

- Reverseモード微分を使って勾配を計算し、ネットワークの重みを少しずつ調整することで学習できる

  1. 入力層から出力層まで内積を求めて出力を得る

  2. 損失関数の各出力に関する微分δを求める

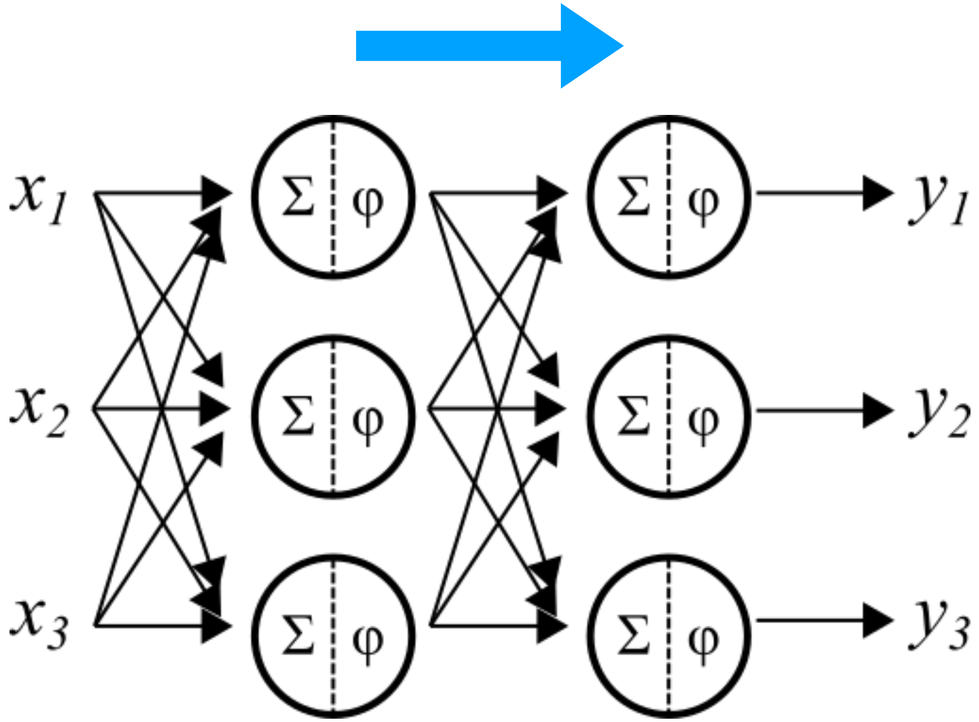
  3. δを逆伝播させていき、重みに関する全勾配を求める

  4. パラメータを更新

- 具体的な最適化方法やテクニックについては後日（→確率的勾配降下法）

誤差逆伝播法

順伝播でxとWからyを求める



逆伝播で既知のx,W,yからδを求めて、Wを更新する