

Deep Learning 輪読会 2017
第6章 深層順伝播型ネットワーク

2017.11.13, 2017.11.20
松尾研究室 特任研究員 村上遙
東京大学 中村・近藤研究室 石井 潤

Deep Learning 輪読会 2017
第6章 深層順伝播型ネットワーク (-6.2)

2017.11.13
松尾研究室
特任研究員 村上 遥

- 6.1 例：XORの学習
- 6.2 勾配に基づく学習
- 6.3 隠れユニット
- 6.4 アーキテクチャの設計
- 6.5 誤差逆伝播法およびその他の微分アルゴリズム
- 6.6 歴史ノート

はじめに

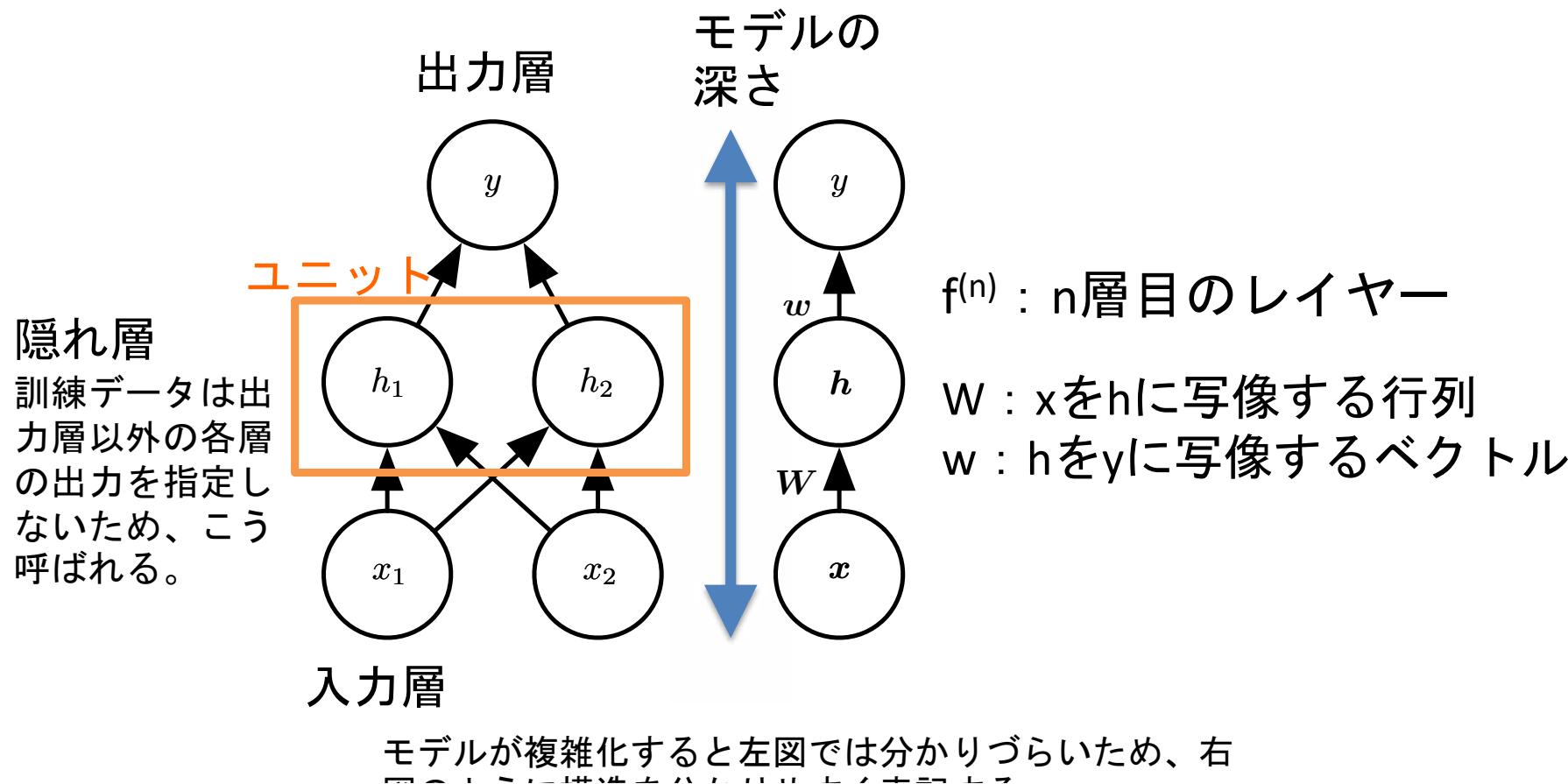
- 前章までの議論：機械学習の基礎、高次のデータにおける問題
 - 従来の機械学習アルゴリズムは適用分野は広いものの、次元の呪いという問題があった
- 本章(~6.2まで)の議論：深層順伝播型ネットワークのアイディアの説明
 - 例：XORの学習
- 深層順伝播型ネットワークにおけるモデル設計
 - コスト関数
 - 交差エントロピー他
 - 出力ユニット
 - 線形
 - シグモイド
 - ソフトマックス

深層順伝播型ネットワーク

- Deep feedforward networks
 - = feedforward neural networks
 - = MLP(multilayer perceptrons)
-
- 生物の脳神経の仕組みからヒントを得たアーキテクチャ
 - 脳の再現が目的ではなく、近年では工学的問題を解くために脳神経分野の知見を用いている
 - 脳で分かっていないことが多いため
 - 脳では実際には活性化関数が異なることが分かっているが、問題解決には有用

深層順伝播型ネットワーク

- 目的：ある関数 f^* を近似すること
- Cf. 分類器 $y=f^*(x)$ は $x \rightarrow y$ (写像カテゴリ)
- $y = f(x; \theta)$ 最も良い関数近似となる θ を学習

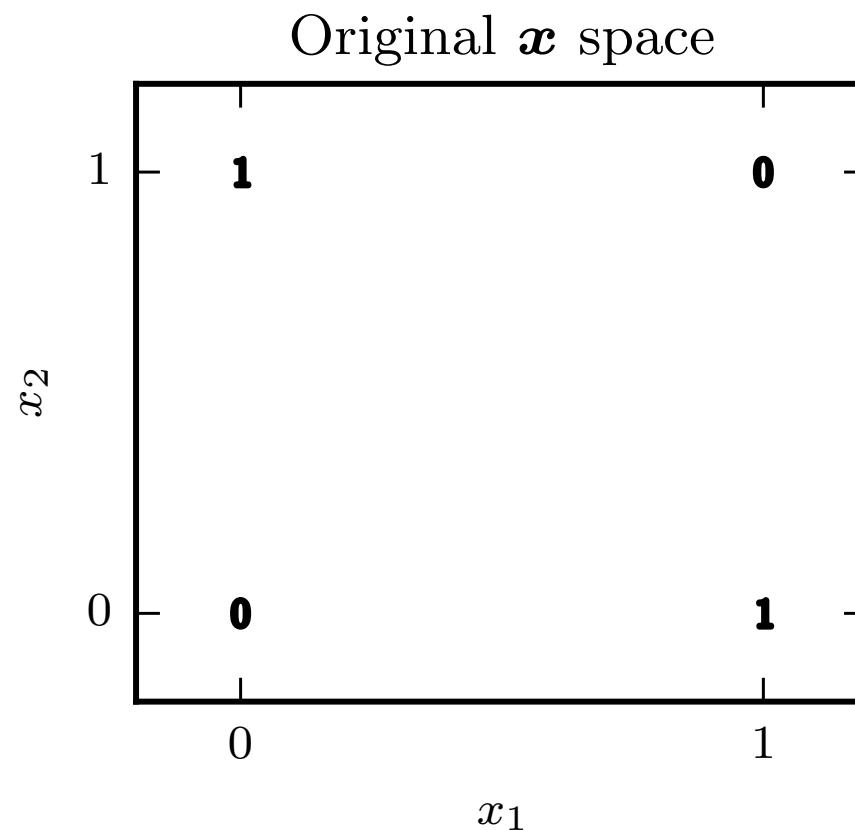


写像をどのように選択するか？

- 1) 非常に汎用的な ϕ を使う
 - (例) RBFカーネルに基づいたカーネルマシンで使われる無限次元 ϕ
 - 汎化性能が弱い
- 2) 自力設計
 - 個々のタスクごとに年月とかなりの労力を費やす
- 3) 表現自体を学習
 - 凸性を放棄するが、タスクごとに設計する必要がなくなる

6.1節 線形分離できない例) X-OR

- 1つの線を引いて分離できない



6.1 X-ORを解くには(1)

回帰問題として扱い、簡易的に平均二乗誤差を損失関数として使う

(本当はベルヌーイ出力分布としてシグモイドユニットを使ったほうが良い)

平均二乗誤差は

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{\mathbf{x} \in \mathbb{X}} (f^*(\mathbf{x}) - f(\mathbf{x}; \boldsymbol{\theta}))^2. \quad (6.1)$$

$\boldsymbol{\theta}$ が \mathbf{w} と b で表される線形モデルを選ぶと

$$f(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^\top \mathbf{w} + b. \quad (6.2)$$

正規方程式を使って、 \mathbf{w} と b に関して閉形式で $J(\boldsymbol{\theta})$ を最小化すると、 $\mathbf{w}=0$, $b=1/2$ が得られる

学習したパラメータで制御されるアフィン変換を使用し、固定された非線形関数（活性化関数）を適用することで特徴量を表現するのが定石なので、

\mathbf{W} を線形変換の重み、 c をバイアスとして

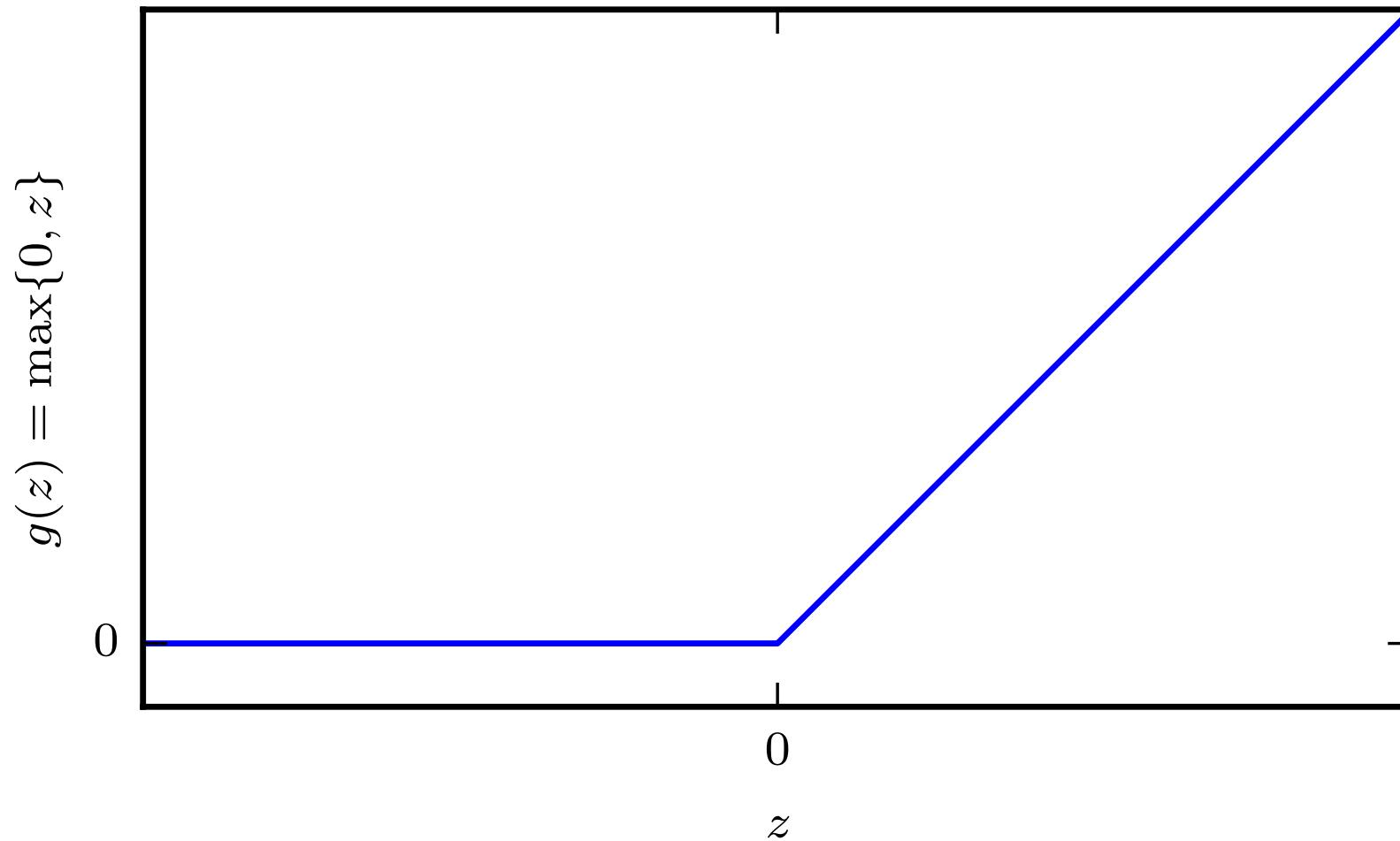
$$h = g(\mathbf{W}^\top \mathbf{x} + c)$$

と定義する。通常は要素ごとに適用されるものを選ぶ↓

$$h_i = g(\mathbf{x}^\top \mathbf{W}_{:,i} + c_i)$$

正規化線形関数(rectified linear unit, ReLU)

- $g(z) = \max\{0, z\}$ で定義される



- 標準的な活性化関数。勾配法で線形モデルを容易に最適化するための性質と線形モデルが良く汎化する為の性質を持っている

6.1 X-ORを解くには(2)

- 正規化線形関数(ReLU)を使うと完全なネットワークは

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = \mathbf{w}^\top \max\{0, \mathbf{W}^\top \mathbf{x} + \mathbf{c}\} + b. \quad (6.3)$$

- ただし、以下のように定義される

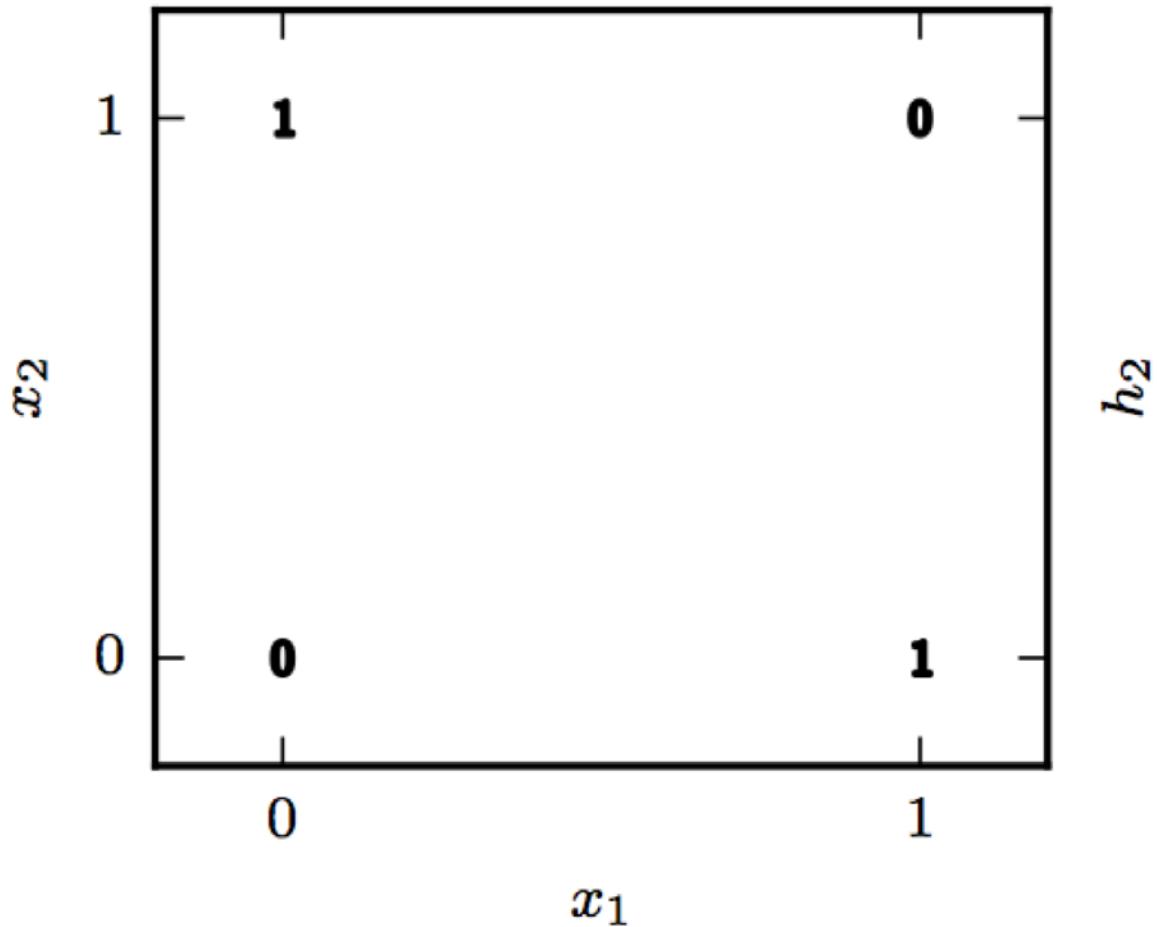
$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad (6.4)$$

$$\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad (6.5)$$

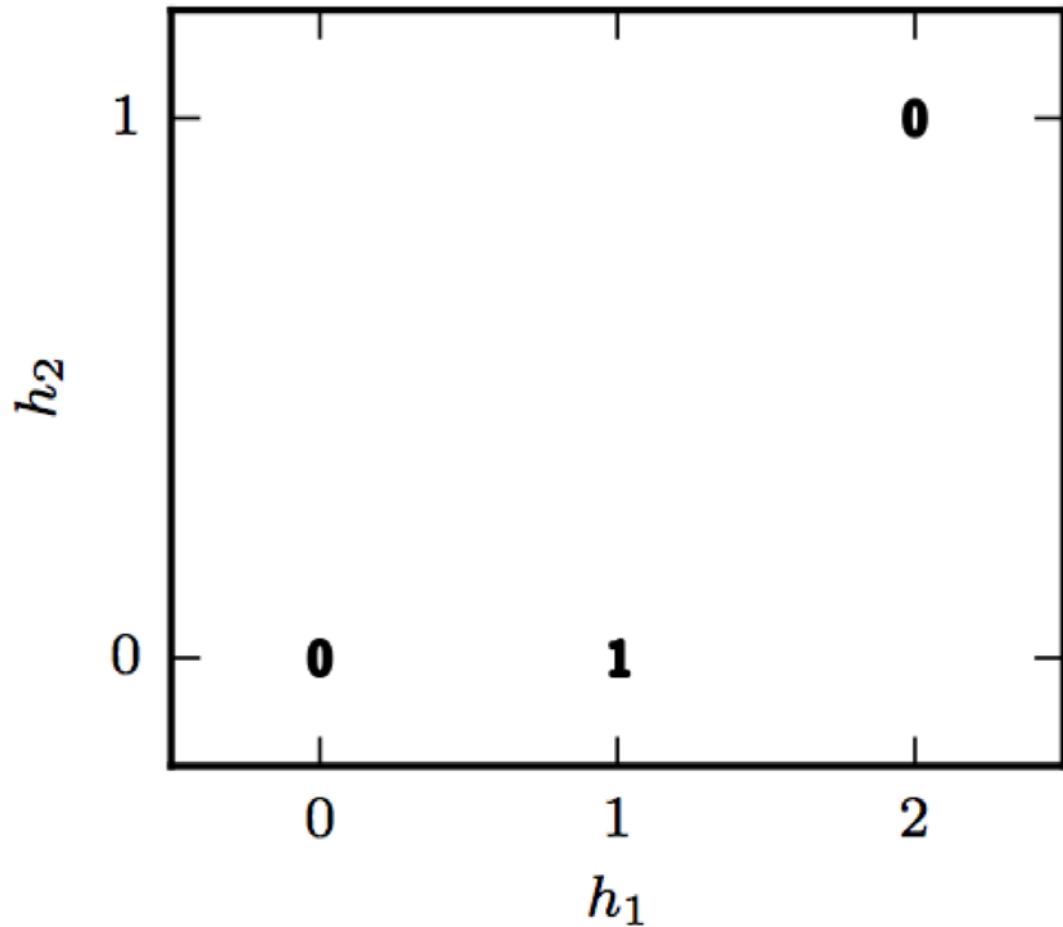
$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \quad (6.6)$$

写像により、分離が可能となった

Original \boldsymbol{x} space



Learned \boldsymbol{h} space



6.2節 勾配に基づく学習

- ニューラルネットワークの非線形性により損失関数のほとんどが非凸になる
→コスト関数の値を非常に小さい値に近づけていく勾配を用いた反復的な最適化が行われる
- 収束性の保証がないので、初期値が重要。順伝播型ニューラルネットワークでは全ての重みを小さいランダム値で初期化することが重要
- 深層ニューラルネットワーク設計の重要な点

- 1) コスト関数の選択
- 2) 出力ユニットの選択

6.2.1 深層ニューラルネットワークにおけるコスト関数

線形モデルなどの他のパラメトリックモデルと同様

だいたい、 $p(y|x;\theta)$

を定義し、最尤法の原則を利用

$$\text{交差エントロピー} : J(\theta) = -\mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \log p_{\text{model}}(\mathbf{y} | \mathbf{x}). \quad (6.12)$$

をコスト関数に使用

θ に関与しない項を使って再構成すると

$$J(\theta) = \frac{1}{2} \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x}; \theta)\|^2 + \text{const.} \quad (6.13)$$

最尤推定からコスト関数を導出すると、モデルごとにコスト関数を設計する負荷がなくなる

モデル $p(y | x)$ を決めれば自動的にコスト関数 $\log p(y | x)$ が決定される

コスト関数の勾配の注意点

- 学習アルゴリズムにとって指標となるように十分大きくかつ予測可能でなければならぬ
- 飽和する（平坦になる）と使えなくなるが、隠れユニット、出力ユニットの出力で活性化関数が飽和することは多い
- 負の対数尤度にすることで解決することが多い
 - 引数が大きな負の値になるコスト関数には飽和の原因の指数関数が含まれているから
 - 打ち消すことができる

6.2.1.2 条件付き統計量の学習

- x が与えられた時の y の条件付き統計量 1つだけを学習したいとき、
 - 例) y の平均を予測する予測器 $f(x; \theta)$
 - コスト関数は汎関数と見なせ、学習は関数を選択することだと考えられる
- 変分法(19章で説明) を使い、最適化問題を解くと以下が得られる

$$f^*(x) = \mathbb{E}_{y \sim p_{\text{data}}(y|x)}[y]. \quad (6.15)$$

- 平均絶対値誤差(mean absolute error)と呼ばれる結果も得られる

$$f^* = \arg \min_f \mathbb{E}_{x,y \sim p_{\text{data}}} \|y - f(x)\|_1 \quad (6.16)$$

- 残念ながら、これは勾配に基づく最適化に使われると低い性能を示すことが多い。その為、交差エントロピーのコスト関数のほうが良く使われる。

6.2.2 出力ユニット

- ・ コスト関数の選択と出力ユニットの選択は強く結びついている
- ・ 出力の表現方法の選択によって交差エントロピー関数の形が決まる
- ・ 「出力ユニット」と言っているが、以下の話は隠れユニットの出力に関しても同様に使える
- ・ 前提：順伝播型ネットワークが $h=f(x; \theta)$ で定義される隠れ特徴量の集合を提供する

6.2.2.1 ガウス出力分布のための線形ユニット

- 通称線形ユニット
- 非線形関数を伴わないアフィン変換に基づくユニット
- 特徴量 h が与えられた時 $\hat{y} = \mathbf{W}^\top h + b$ を出力

$$p(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; \hat{\mathbf{y}}, \mathbf{I}) \quad (6.17)$$

- 飽和することができないため勾配に基づく最適化手法と相性が良い、幅広い最適化アルゴリズムと共に利用可

6.2.2.2 ベルヌーイ出力分布のためのシグモイドユニット

- 二値変数 y の予測
- 例) 2 クラス分類問題
- 1 つの数だけで定義され、ニューラルネットワークは $P(y=1|x)$ のみを予測
([0,1])
- 線形ユニットを使い、値に閾値を設けると

$$P(y = 1 \mid \mathbf{x}) = \max \left\{ 0, \min \left\{ 1, \mathbf{w}^\top \mathbf{h} + b \right\} \right\}. \quad (6.18)$$

- シグモイド出力ユニットは以下で表される

$$\hat{y} = \sigma (\mathbf{w}^\top \mathbf{h} + b) \quad (6.19)$$

6.2.2.3 マルノスキー山のソフトマツヘント

(1)

- n 個の取り得る離散値に関する確率分布を表現したい場合、常にソフトマックス関数が利用できる
- 二値変数に関する確率分布を表現するために利用されたシグモイド関数を一般化したものと考えられる
- 分類器の出力として最もよく使われる。 n 個の異なるクラスに対する確率分布を表現する
- 二値変数から拡張するため、代わりに $z = \log \tilde{P}(y = 1 \mid \mathbf{x})$ を予測する
- 線形の層で正規化されていない対数確率を以下の条件の下で予測する

$$z = \mathbf{W}^\top \mathbf{h} + b, \quad (6.28)$$

$$z_i = \log \tilde{P}(y = i \mid \mathbf{x})$$

6.2.2.5 マルノスキー山ノリノイロのこみのノントマツフヘユート (2)

ソフトマックス関数は

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}. \quad (6.29)$$

で与えられ、ロジスティックシグモイドで最大対数尤度を利用した時と同様に

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j). \quad (6.30)$$

と定義できる

入力の各要素全てに同じスカラーを加えたとき、ソフトマックスの出力が不变である性質 $\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} + c)$. を用いると、 (6.32)

数値計算的に安定なソフトマックスは以下で表される

$$\text{softmax}(\mathbf{z}) = \text{softmax}(\mathbf{z} - \max_i z_i). \quad (6.33)$$

出力の和は常に 1 なので、あるユニットの値が大きいと勝者総取り状態になる。

コスト関数と出力ユニットの組み合わせ

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)
Continuous	Mixture of Gaussian	Mixture Density	Cross-entropy
Continuous	Arbitrary	See part III: GAN, VAE, FVBN	Various

Deep Learning 輪読会 2017
6.3 隠れユニット - 6.6 歴史ノート

2017.11.20
東京大学 情報理工学系研究科 システム情報学専攻
中村・近藤研究室
石井 潤

構成

6.3 隠れユニット

6.4 アーキテクチャの設計

6.5 誤差逆伝播法およびその他の微分アルゴリズム

6.6 歴史ノート

6.3 隠れユニット（活性化関数）

- ほとんどの隠れユニットにおいて、その違いは、活性化関数の違い

モデルの隠れ層の中の、隠れユニットの種類をどう選択するか

- (本節からは、順伝播型ネットワークに固有の問題に着目)

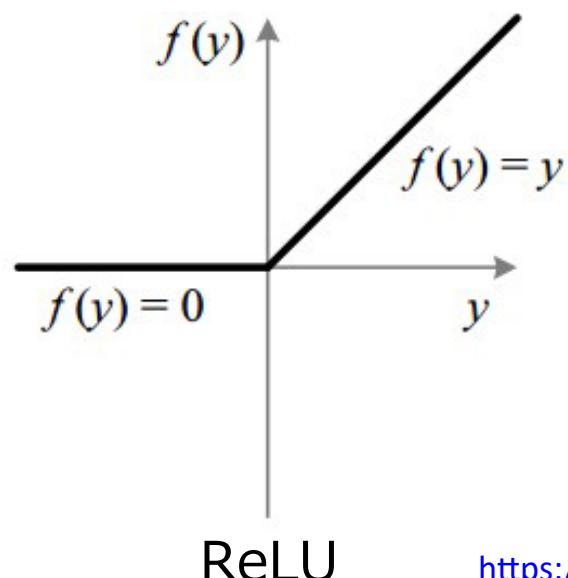
各種の隠れユニットについての基本的な洞察を説明

- 理論的な原則に基づく確実な指針はまだ少ない。研究が極めて活発
- 試行錯誤（実際にネットワークを訓練、性能評価）するしかない
- 基本的な洞察に基いて、試すユニットを決定
 - （標準的な選択として、ReLUが優れている）
- 活性化関数が微分不可能であることは、実際に無視しても問題がない

6.3.1 ReLUとその一般化 (1/2)

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

- ・ 線型関数と似ているので、最適化しやすい
- ・ ユニットが活性化している領域ではつねにその微分が大きいまま
- ・ 勾配は大きいだけでなく一定
- ・ バイアスの初期値は、0.1などの小さい正の値に設定するとよく機能する
 - ほとんどの入力が最初は活性化し、勾配を通過できる

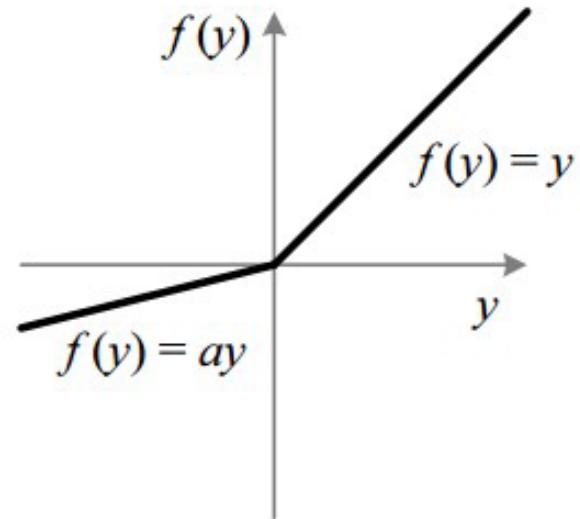


<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

6.3.1 ReLUとその一般化 (2/2)

ReLUの一般化

- 一般化式 $g(z) = \max(0, z) + \alpha \min(0, z)$
- Absolute value rectification : $g(z) = |z|$
- Leaky ReLU : $\alpha = 0.01$ 等の小さい値に
- パラメトリックReLU : α をパラメータとして学習



Leaky ReLU

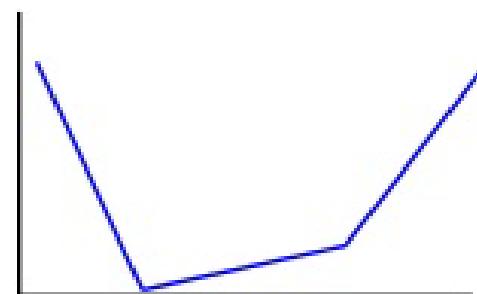
ReLUのさらなる一般化

- マックスアウトユニット(Maxout units)

$$g(z)_i = \max_{j \in \mathbb{G}^{(i)}} z_j \quad (6.37)$$

<https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

- 最大k個までの区分を組み合わせて区分線形凸関数を学習
- 活性化関数自体を学習する

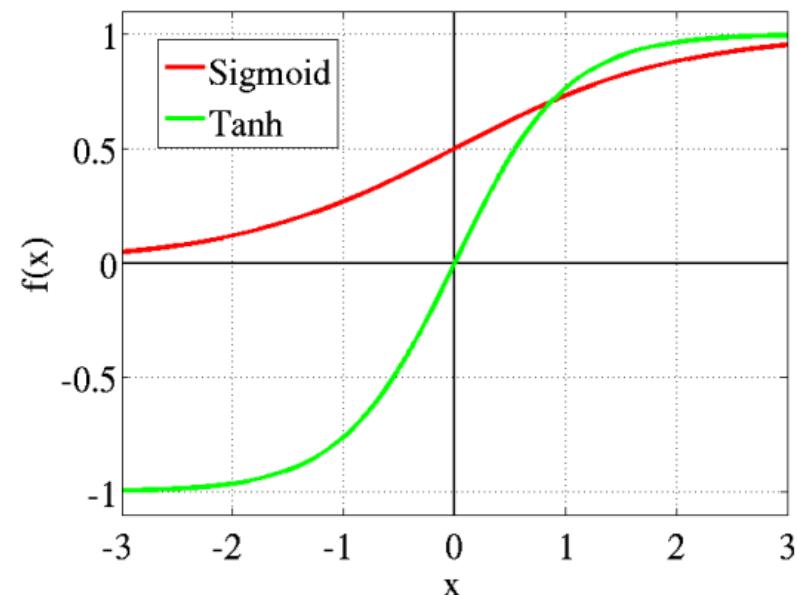


Maxout

6.3.2 ロジスティックシグモイドとハイパボリックタンジェント

シグモイド活性化関数 (sigmoidal activation function)

- ReLU登場以前によく使われていた活性化関数
 - ロジスティックシグモイド活性化関数 $g(z) = \sigma(z)$
 - ハイパボリックタンジェント活性化関数 $g(z) = \tanh(z)$
- 2関数は非常に近い関係 ($\tanh(z) = 2\sigma(2z) - 1$ より)
- 入力の絶対値が大きいときは出力が飽和し、0に近いところで敏感
 - 順伝播型ネットワークでは非推奨
- ハイパボリックタンジェントの方がよく機能する
 - 0付近では恒等関数に「似ている」から
- 順伝播型ネットワーク以外では一般的に使用される
 - 区分線形関数が使えない場合
 - 回帰結合型など



6.3.3 その他の隠れユニット

その他の隠れユニット

- あまり利用されない
- 微分可能であれば幅広い種類の関数が極めてよく機能する
- \cos を活性化関数として、MNISTデータセットを学習、エラー率1%
- 既存と同等のものはたくさんあるが、発表されない

一般的な隠れユニット

- 活性化関数なし = 恒等関数
- 線形変換
 - ランクを低くする
- ソフトマックスユニット
- 動径基底関数
- ソフトプラス
- Hard tanh

6.3.3 その他の隠れユニット (参考)

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1 + e^{-x}}$

6.4 アーキテクチャの設計

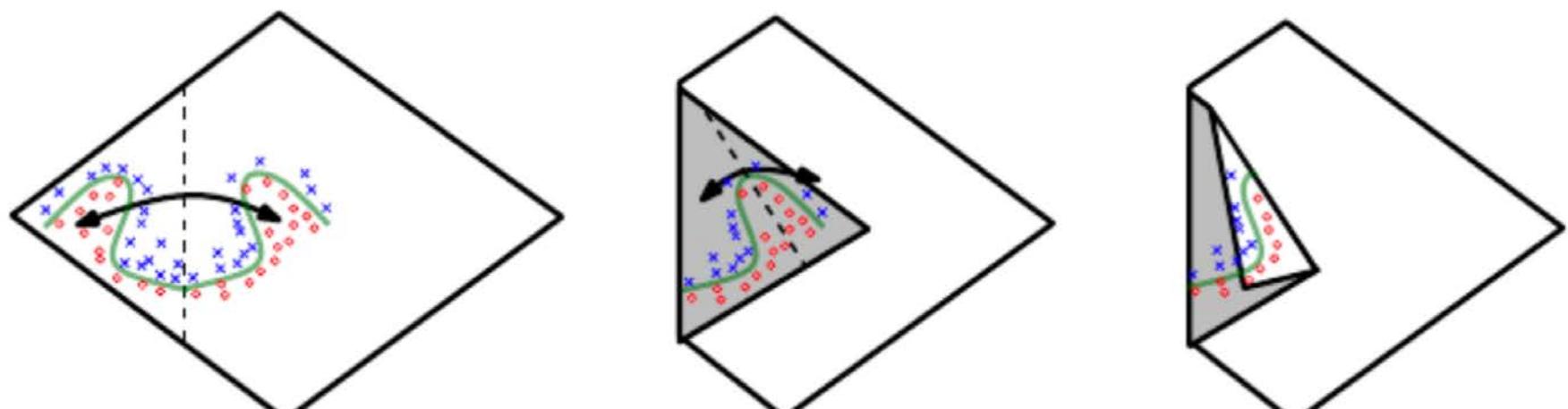
ニューラルネットワークのアーキテクチャ(architecture)

- ネットワーク全体の構造
 - ユニットの数
 - ユニット同士の結合
- 順伝播型ネットワークの設計において重要なこと
 - **ネットワークの深さ**
 - **各層の幅**
- タスクごとに理想的なネットワーク検証集合の誤差を観察することで実験的に発見

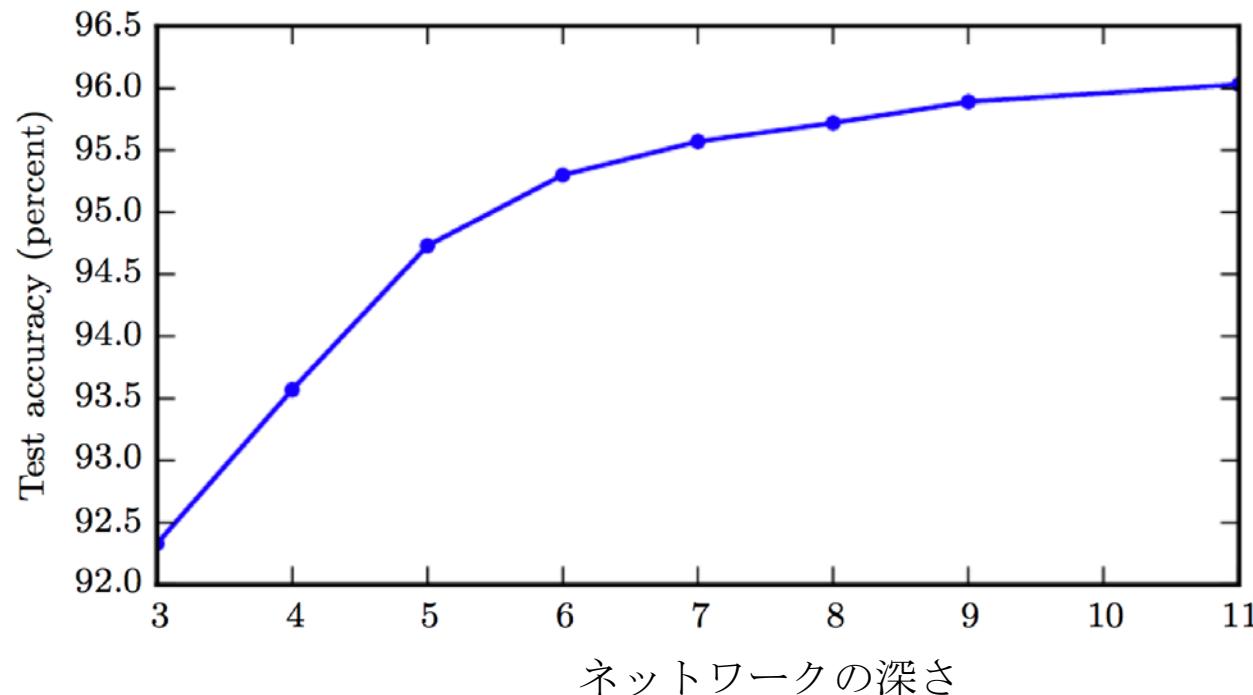
6.4.1 万能近似性と深さ (1/2)

万能近似定理

- ・ 「大きな」 MLPであればその関数を表現できる
- ・ 訓練アルゴリズムがその関数を学習できるかは保証されていない
 - 理由 1) 近似したい関数に対応するパラメータの値を発見できないかもしない
 - 理由 2) 過剰適合により間違った関数を選択するかもしれない
- ・ ネットワークが十分大きければ任意の精度で関数を近似できるが、どのくらいの大きさになるのかは述べられていない
- ・ ↗

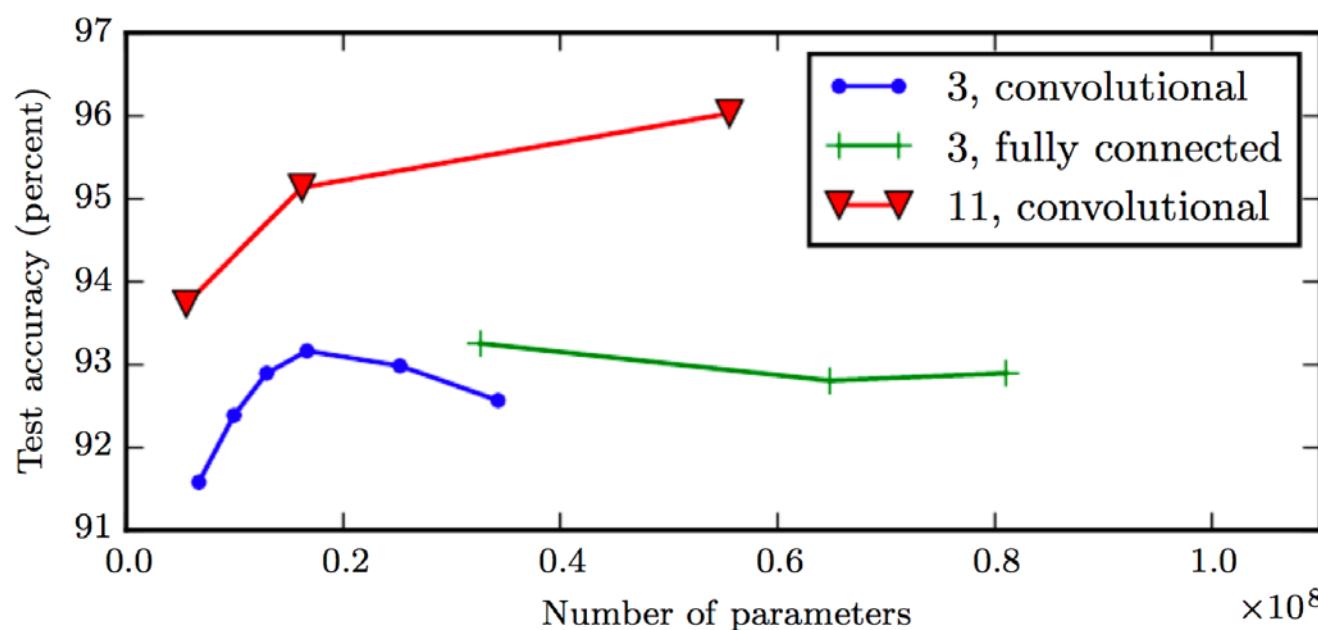


6.4.1 万能近似性と深さ (2/2)



深さの効果

- 層が深いほど、高い汎化性能



パラメータ数の効果

- 深いモデルの方がより良い性能
- 浅いモデルでは、パラメータ数を増やしても深いモデルと同等の性能向上が見込めない

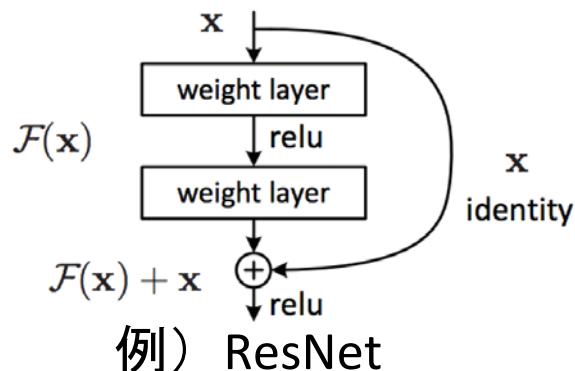
6.4.2 アーキテクチャ設計におけるその他の検討事項

タスク特化のネットワーク

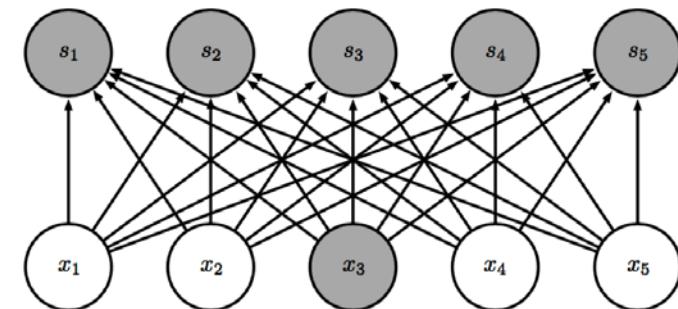
- 畳み込みニューラルネットワーク（9章），コンピュータビジョン向け
- 回帰結合型ニューラルネットワーク（10章），系列処理向け

層のスキップ

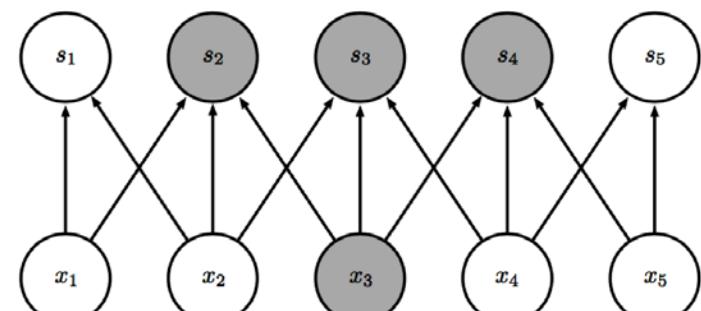
- 入力に近い層へ，勾配が伝わりやすくなる



例) ResNet



密な結合



疎な結合 (9章)

2つの層の接続

- 特化型ネットワーク（以降の章）の多くは，接続数が少ない
- 接続数を減らす方針は，問題に大きく依存する

6.5 誤差逆伝播法およびその他の微分アルゴリズム

順伝播 (forward propagation)

- 情報が、入力層から中間層を通って伝わり出力層まで伝播される流れ

誤差逆伝播法 (back-propagation, backprop)

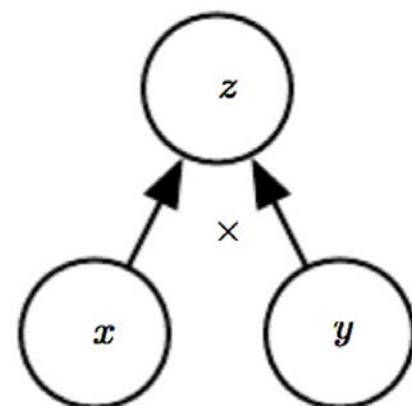
- 勾配の計算を行う、単純かつコストの低い手法
- 損失からの情報をネットワーク上で逆向きに伝播
- 具体的には、任意の f について、勾配 $\nabla_x f(x, y)$ を計算する手法
 - x はその微分を求めたい変数の集合
 - y は関数の入力として与えられるが微分の計算が必要のない変数の集合
 - ここでは f の出力が1つである最も一般的な場合に限定して説明

※誤差逆伝播法は学習アルゴリズム全体を意味せず、勾配計算に使用されるもの
計算した勾配を用いて学習を実行するのは、確率的勾配法などのアルゴリズム

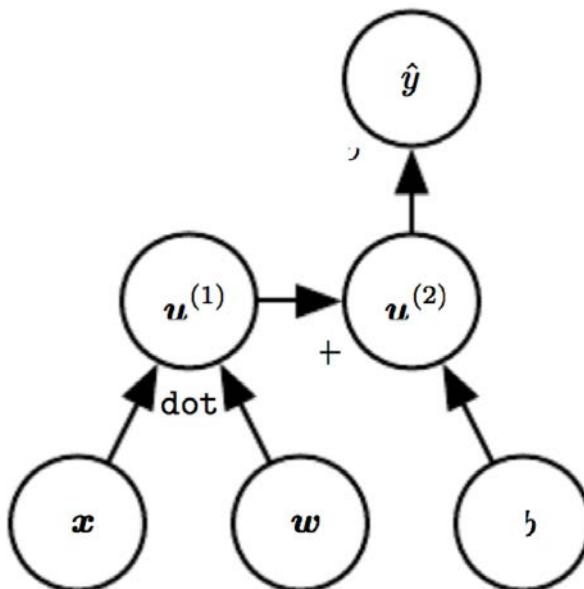
6.5.1 計算グラフ

計算グラフ

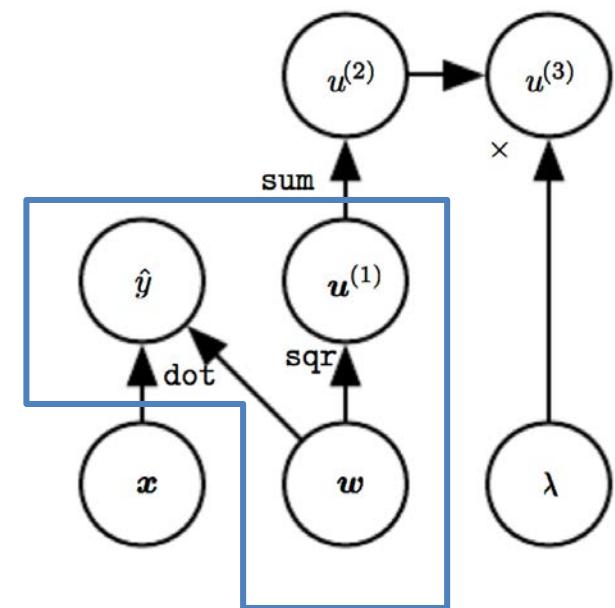
- 演算 (operation) : 1つ以上の変数の関数
- グラフの各ノードが変数を表す
 - 変数はスカラー, ベクトル, 行列, テンソル, あるいはそれ以外の形式の変数
- 変数 x にある演算を適用して変数 y が求められる場合, x から y への有向辺を引く



演算 \times で $z = xy$ を計算



$$\hat{y} = \sigma(x^T w + b)$$



1つの変数に複数の演算
を適用可能

6.5.2 微積分の連鎖律

微積分の連鎖律（※確率の連鎖律ではない）

- 「微分が既知の関数」から構成される関数 の微分を計算するのに利用される
- 誤差逆伝播法は、非常に効率のよい演算を特定の順番で適用することで連鎖律を計算
- スカラーの場合

$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}. \quad (6.44)$$

$$y = g(x), \quad z = f(g(x)) = f(y)$$

f と g はいずれも実数から実数への写像

- スカラー以外の場合にも一般化

$$\left| \frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}. \quad (6.45) \right.$$

$$x \in \mathbb{R}^m, y \in \mathbb{R}^n,$$

g は \mathbb{R}^m から \mathbb{R}^n への写像,
 f は \mathbb{R}^n から \mathbb{R} への写像

- (6.45)のベクトル表記

$$\nabla_x z = \left(\frac{\partial y}{\partial x} \right)^\top \nabla_y z, \quad (6.46)$$

$\left| \frac{\partial y}{\partial x}$ は g の $n \times m$ ヤコビ行列

- テンソルの場合

$$\nabla_{\mathbf{x}} z = \sum (\nabla_{\mathbf{x}} Y_j) \frac{\partial z}{\partial Y_j}. \quad (6.47)$$

6.5.3 誤差逆伝播のための連鎖律の再帰的な適用 (1/3)

連鎖率の例で説明

$$x = f(w), y = f(x), z = f(y).$$

$$\frac{\partial z}{\partial w} \quad (6.50)$$

$$= \frac{\partial z}{\partial y} \frac{\partial y}{\partial x} \frac{\partial x}{\partial w} \quad (6.51)$$

$$= f'(y) f'(x) f'(w) \quad (6.52)$$

$$= f'(f(f(w))) f'(f(w)) f'(w) \quad (6.53)$$

- **(6.52) 式**

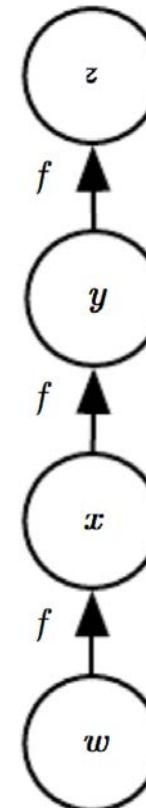
- 誤差逆伝播で使われる式

- 同じ計算を繰り返さないため、処理時間が短い

- **(6.53) 式**

- $f(w)$ が繰り返し計算されるため、(6.52)式にくらべて処理時間が長い

- メモリに制限ある場合に有効な手法



6.5.3 誤差逆伝播のための連鎖律の再帰的な適用 (2/3)

順伝播

Algorithm 6.1 A procedure that performs the computations mapping n_i inputs $u^{(1)}$ to $u^{(n_i)}$ to an output $u^{(n)}$. This defines a computational graph where each node computes numerical value $u^{(i)}$ by applying a function $f^{(i)}$ to the set of arguments $\mathbb{A}^{(i)}$ that comprises the values of previous nodes $u^{(j)}$, $j < i$, with $j \in Pa(u^{(i)})$. The input to the computational graph is the vector \mathbf{x} , and is set into the first n_i nodes $u^{(1)}$ to $u^{(n_i)}$. The output of the computational graph is read off the last (output) node $u^{(n)}$.

```
for  $i = 1, \dots, n_i$  do
     $u^{(i)} \leftarrow x_i$ 
end for
for  $i = n_i + 1, \dots, n$  do
     $\mathbb{A}^{(i)} \leftarrow \{u^{(j)} \mid j \in Pa(u^{(i)})\}$ 
     $u^{(i)} \leftarrow f^{(i)}(\mathbb{A}^{(i)})$ 
end for
return  $u^{(n)}$ 
```

6.5.3 誤差逆伝播のための連鎖律の再帰的な適用 (3/3)

誤差逆伝播

- 連鎖律

$$\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}} \quad (6.49)$$

Algorithm 6.2 Simplified version of the back-propagation algorithm for computing the derivatives of $u^{(n)}$ with respect to the variables in the graph. This example is intended to further understanding by showing a simplified case where all variables are scalars, and we wish to compute the derivatives with respect to $u^{(1)}, \dots, u^{(n_i)}$. This simplified version computes the derivatives of all nodes in the graph. The computational cost of this algorithm is proportional to the number of edges in the graph, assuming that the partial derivative associated with each edge requires a constant time. This is of the same order as the number of computations for the forward propagation. Each $\frac{\partial u^{(i)}}{\partial u^{(j)}}$ is a function of the parents $u^{(j)}$ of $u^{(i)}$, thus linking the nodes of the forward graph to those added for the back-propagation graph.

Run forward propagation (algorithm 6.1 for this example) to obtain the activations of the network.

Initialize `grad_table`, a data structure that will store the derivatives that have been computed. The entry `grad_table[u(i)]` will store the computed value of $\frac{\partial u^{(n)}}{\partial u^{(i)}}$.

```
grad_table[u(n)] ← 1
for j = n - 1 down to 1 do
```

The next line computes $\frac{\partial u^{(n)}}{\partial u^{(j)}} = \sum_{i:j \in Pa(u^{(i)})} \frac{\partial u^{(n)}}{\partial u^{(i)}} \frac{\partial u^{(i)}}{\partial u^{(j)}}$ using stored values:

```
grad_table[u(j)] ← ∑i:j ∈ Pa(u(i)) grad_table[u(i)] ∂u(i) / ∂u(j)
```

```
end for
```

```
return {grad_table[u(i)] | i = 1, ..., ni}
```

6.5.4 全結合MLPでの誤差逆伝播法 (1/2)

全結合MLPでの順伝播

Algorithm 6.3 Forward propagation through a typical deep neural network and the computation of the cost function. The loss $L(\hat{\mathbf{y}}, \mathbf{y})$ depends on the output $\hat{\mathbf{y}}$ and on the target \mathbf{y} (see section 6.2.1.1 for examples of loss functions). To obtain the total cost J , the loss may be added to a regularizer $\Omega(\theta)$, where θ contains all the parameters (weights and biases). Algorithm 6.4 shows how to compute gradients of J with respect to parameters \mathbf{W} and \mathbf{b} . For simplicity, this demonstration uses only a single input example \mathbf{x} . Practical applications should use a minibatch. See section 6.5.7 for a more realistic demonstration.

Require: Network depth, l
Require: $\mathbf{W}^{(i)}, i \in \{1, \dots, l\}$, the weight matrices of the model
Require: $\mathbf{b}^{(i)}, i \in \{1, \dots, l\}$, the bias parameters of the model
Require: \mathbf{x} , the input to process
Require: \mathbf{y} , the target output

$$\mathbf{h}^{(0)} = \mathbf{x}$$

for $k = 1, \dots, l$ **do**

$$\mathbf{a}^{(k)} = \mathbf{b}^{(k)} + \mathbf{W}^{(k)} \mathbf{h}^{(k-1)}$$
$$\mathbf{h}^{(k)} = f(\mathbf{a}^{(k)})$$

end for

$$\hat{\mathbf{y}} = \mathbf{h}^{(l)}$$
$$J = L(\hat{\mathbf{y}}, \mathbf{y}) + \lambda \Omega(\theta)$$

6.5.4 全結合MLPでの誤差逆伝播法 (2/2)

全結合MLPでの誤差逆伝播

Algorithm 6.4 Backward computation for the deep neural network of algorithm 6.3, which uses, in addition to the input \mathbf{x} , a target \mathbf{y} . This computation yields the gradients on the activations $\mathbf{a}^{(k)}$ for each layer k , starting from the output layer and going backwards to the first hidden layer. From these gradients, which can be interpreted as an indication of how each layer's output should change to reduce error, one can obtain the gradient on the parameters of each layer. The gradients on weights and biases can be immediately used as part of a stochastic gradient update (performing the update right after the gradients have been computed) or used with other gradient-based optimization methods.

After the forward computation, compute the gradient on the output layer:

$$\mathbf{g} \leftarrow \nabla_{\hat{\mathbf{y}}} J = \nabla_{\hat{\mathbf{y}}} L(\hat{\mathbf{y}}, \mathbf{y})$$

for $k = l, l-1, \dots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if f is element-wise):

$$\mathbf{g} \leftarrow \nabla_{\mathbf{a}^{(k)}} J = \mathbf{g} \odot f'(\mathbf{a}^{(k)})$$

Compute gradients on weights and biases (including the regularization term, where needed):

$$\nabla_{\mathbf{b}^{(k)}} J = \mathbf{g} + \lambda \nabla_{\mathbf{b}^{(k)}} \Omega(\theta)$$

$$\nabla_{\mathbf{W}^{(k)}} J = \mathbf{g} \mathbf{h}^{(k-1)\top} + \lambda \nabla_{\mathbf{W}^{(k)}} \Omega(\theta)$$

Propagate the gradients w.r.t. the next lower-level hidden layer's activations:

$$\mathbf{g} \leftarrow \nabla_{\mathbf{h}^{(k-1)}} J = \mathbf{W}^{(k)\top} \mathbf{g}$$

end for

6.5.5 シンボル間の微分

シンボリック (symbolic) 表現

- 代数式や計算グラフでは、特定の値を持たないシンボル(symbols)あるいは変数に対して演算を行う
- ニューラルネットワークを実際に使用・訓練する場合は具体的な数値 (numeric value) に置き換える

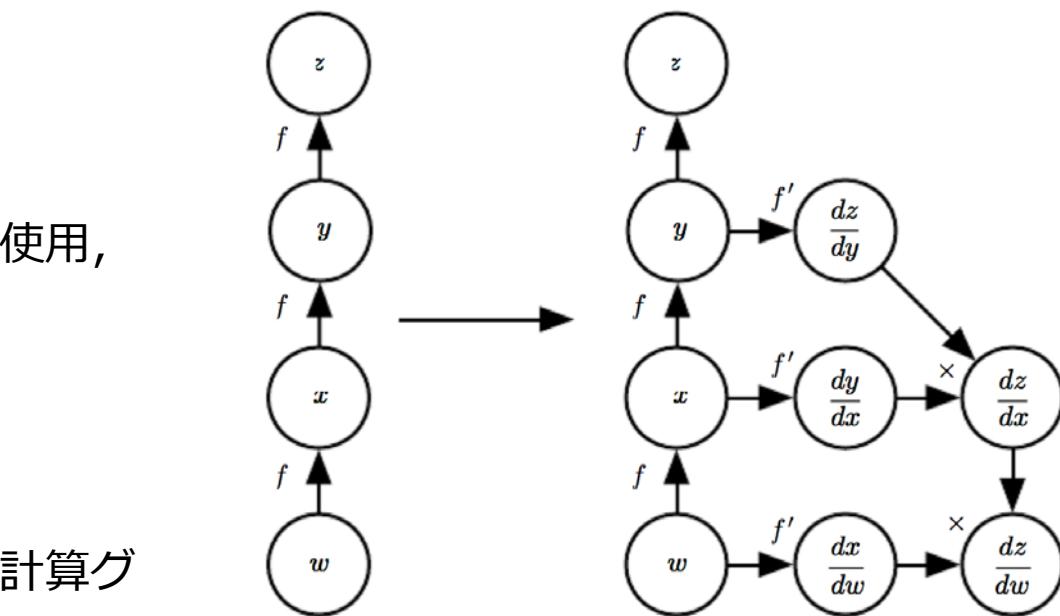
実装方法

「シンボルと数値間」の微分

- 計算グラフとそのグラフへの入力値の集合を使用、入力値についての勾配を記述する数値を返す
- TorchやCaffe

ノードを計算グラフに追加する

- 微分のシンボリックな記述を与えるノードを計算グラフに追加する（右図）
- TheanoやTensorflow



補足 ライブライリについて

Framework Comparison: Design Choices

Design Choice	Torch.nn	Theano-based	Caffe	autograd (NumPy, Torch)	Chainer	MXNet	Tensor-Flow
1.NN definition	Script (Lua)	Script* (Python)	Data (protobuf)	Script (Python, Lua)	Script (Python)	Script (many)	Script (Python)
2. Graph construction	Prebuild	Prebuild	Prebuild	Dynamic	Dynamic	Prebuild**	Prebuild
3. Backprop	Through graph	Extended graph	Through graph	Extended graph	Through graph	Through graph	Extended graph
4. Parameters	Hidden in operators	Separate nodes	Hidden in operators	Separate nodes	Separate nodes	Separate nodes	Separate nodes
5. Update formula	Outside of graphs	Part of graphs	Outside of graphs	Outside of graphs	Outside of graphs	Outside of graphs**	Part of graphs
6. Optimization	-	Advanced optimization	-	-	-	-	Simple optimization
57 Parallel computation	Multi GPU	Multi GPU (libgpuarray)	Multi GPU	Multi GPU (Torch)	Multi GPU	Multi node Multi GPU	Multi node Multi GPU

* Some of Theano-based frameworks use data (e.g. yaml)

** Dynamic dependency analysis and optimization is supported (no autodiff support)²²

6.5.6 一般的な誤差逆伝播法 (1/3)

動的計画法による実装

- 誤差逆伝播を素朴に実装すると、部分式の繰り返しにより実行時間が指数関数的に増加
- 動的計画法(dynamic programming) を用いることでこの問題を回避
 - 再計算を避けるために、中間的な結果を保存して活用
 - 表の穴埋めアルゴリズム
 - 各ノードは、そのノードの勾配を保存するエントリを表中に持つ

6.5.6 一般的な誤差逆伝播法 (2/3)

誤差逆伝播のアルゴリズム

- 勾配の表 grad_table を埋めていく

Algorithm 6.5 The outermost skeleton of the back-propagation algorithm. This portion does simple setup and cleanup work. Most of the important work happens in the `build_grad` subroutine of algorithm 6.6

Require: \mathbb{T} , the target set of variables whose gradients must be computed.

Require: \mathcal{G} , the computational graph

Require: z , the variable to be differentiated

Let \mathcal{G}' be \mathcal{G} pruned to contain only nodes that are ancestors of z and descendants of nodes in \mathbb{T} .

Initialize `grad_table`, a data structure associating tensors to their gradients

$\text{grad_table}[z] \leftarrow 1$

for \mathbf{V} in \mathbb{T} **do**

`build_grad($\mathbf{V}, \mathcal{G}, \mathcal{G}', \text{grad_table}$)`

end for

Return `grad_table` restricted to \mathbb{T}

6.5.6 一般的な誤差逆伝播法 (3/3)

Algorithm 6.6 The inner loop subroutine `build_grad(V, G, G', grad_table)` of the back-propagation algorithm, called by the back-propagation algorithm defined in algorithm 6.5.

Require: V , the variable whose gradient should be added to G and `grad_table`

Require: G , the graph to modify

Require: G' , the restriction of G to nodes that participate in the gradient

Require: `grad_table`, a data structure mapping nodes to their gradients

if V is in `grad_table` **then** # 既にメモがあつたらループを抜ける

 Return `grad_table[V]`

end if

$i \leftarrow 1$

for C in `get_consumers(V, G')` **do** # ノード V の子ノードのリストでイテレーション

$op \leftarrow \text{get_operation}(C)$ # 子ノード C の演算を取得

$D \leftarrow \text{build_grad}(C, G, G', \text{grad_table})$ # 子ノード C について、再帰的にサブルーチンを呼出

$G^{(i)} \leftarrow op.bprop(\text{get_inputs}(C, G'), V, D)$ # 子ノード C の誤差逆伝播

$i \leftarrow i + 1$

end for

$G \leftarrow \sum_i G^{(i)}$

`grad_table[V] = G`

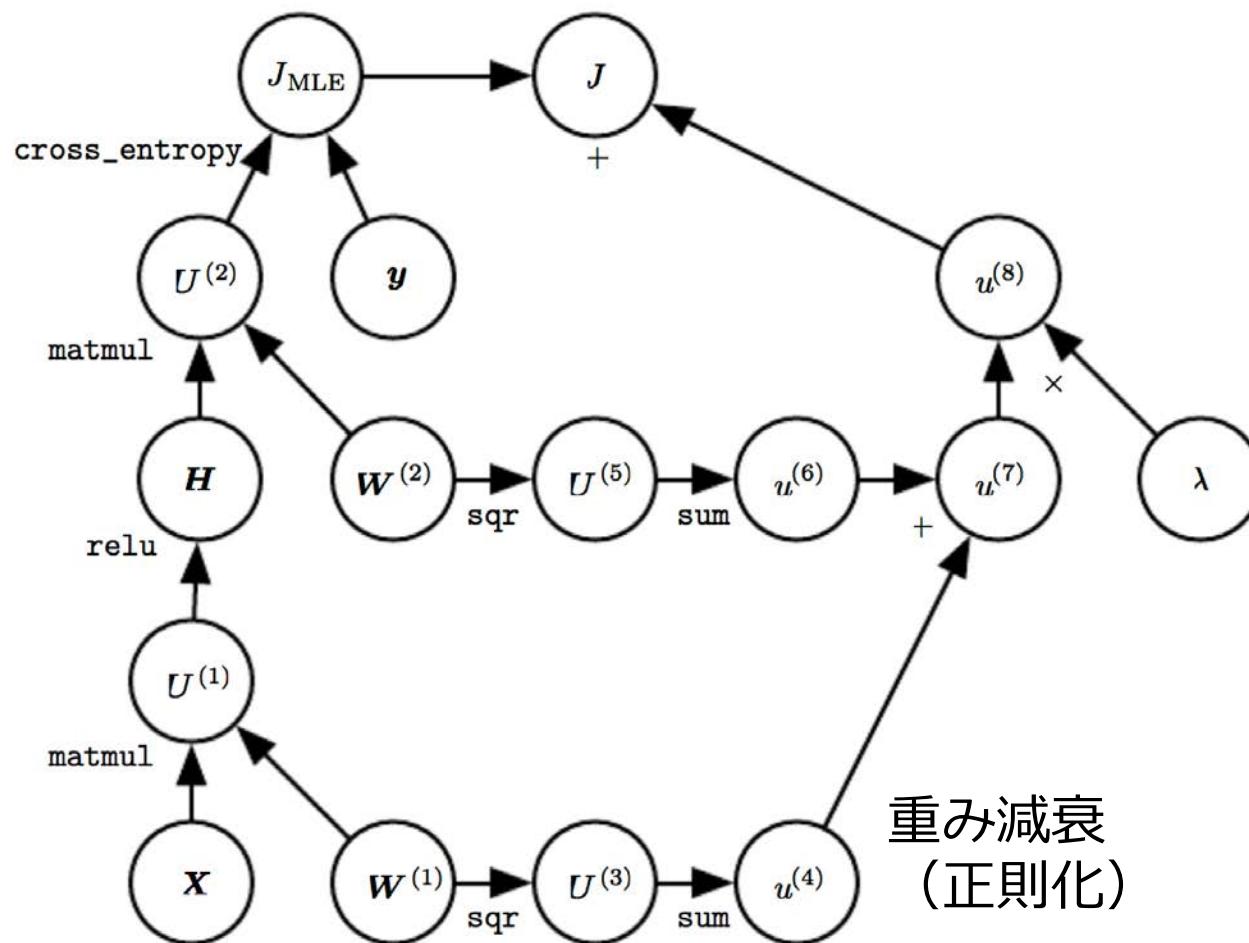
Insert G and the operations creating it into G

Return G

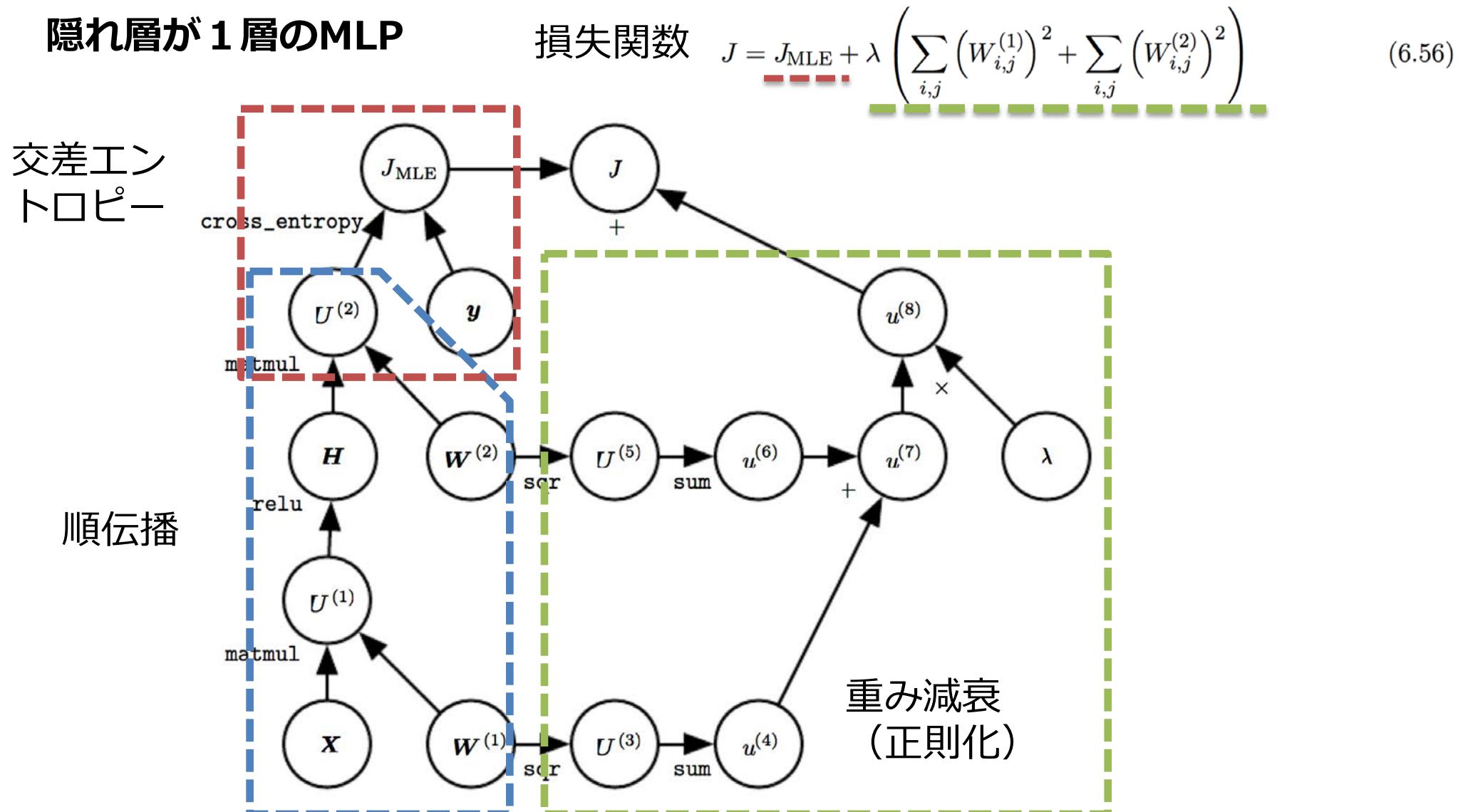
6.5.7 例：誤差逆伝播法による多層パーセプトロンの訓練 (1/5)

隠れ層が 1 層の MLP

損失関数
$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right) \quad (6.56)$$



6.5.7 例：誤差逆伝播法による多層パーセプトロンの訓練 (2/5)

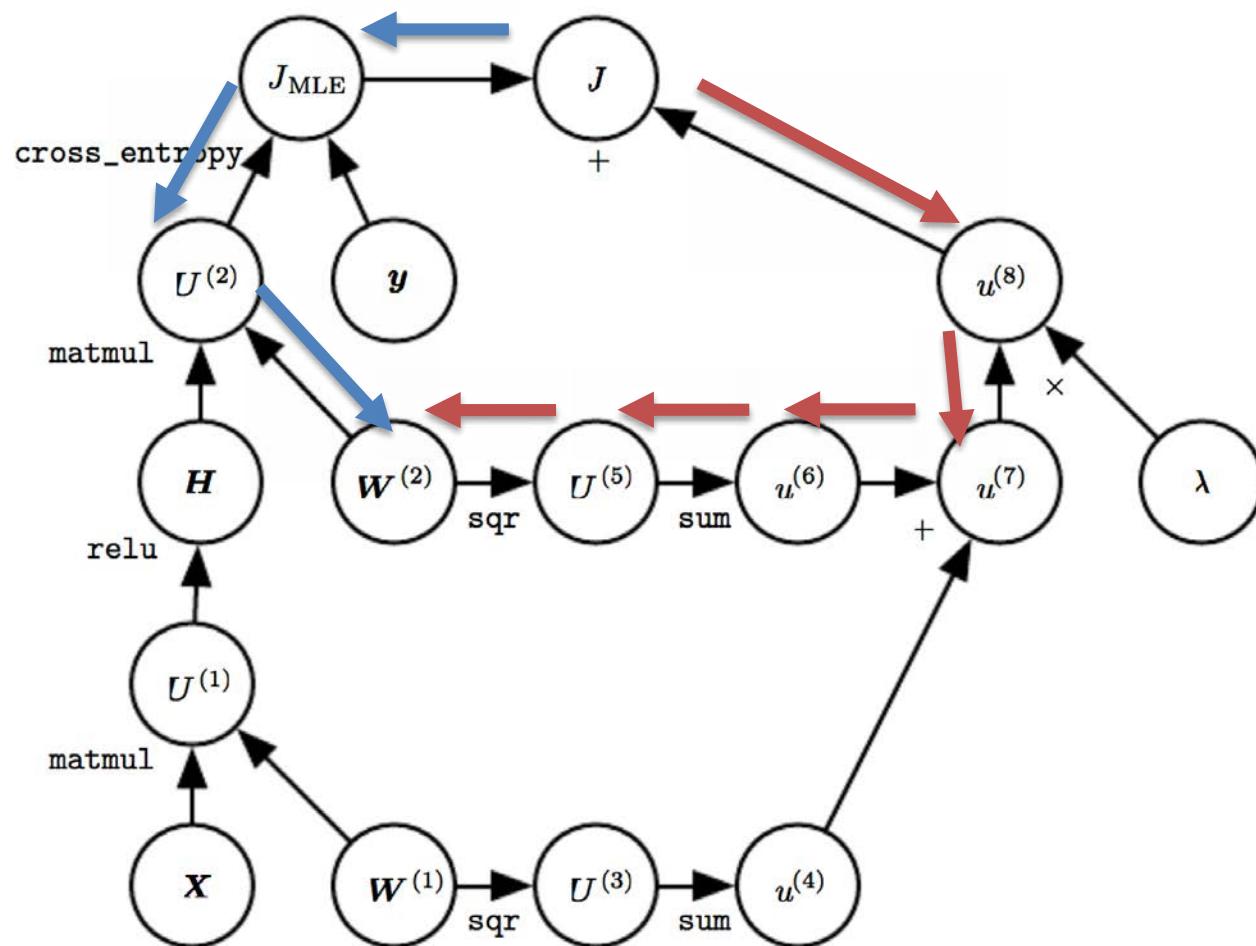


6.5.7 例：誤差逆伝播法による多層パーセプトロンの訓練 (3/5)

隠れ層が 1 層の MLP

損失関数

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} (W_{i,j}^{(1)})^2 + \sum_{i,j} (W_{i,j}^{(2)})^2 \right) \quad (6.56)$$



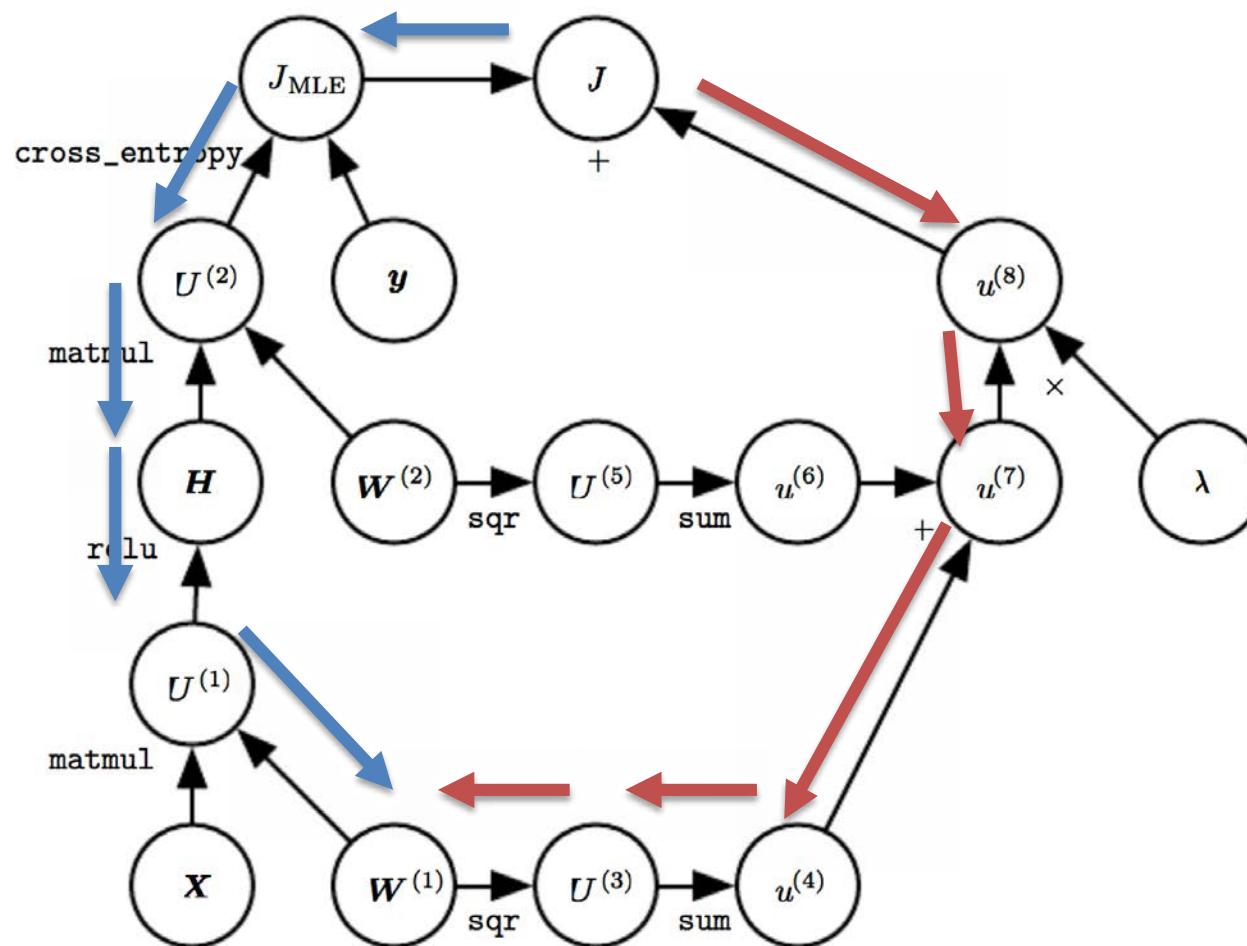
$\nabla_{W^{(2)}} J$ を求める
 G を $U^{(2)}$ の勾配とすると,
 $\nabla_{W^{(2)}} J = \underline{H^T G} + \underline{2\lambda W^{(2)}}$

6.5.7 例：誤差逆伝播法による多層パーセプトロンの訓練 (4/5)

隠れ層が 1 層の MLP

損失関数

$$J = J_{\text{MLE}} + \lambda \left(\sum_{i,j} \left(W_{i,j}^{(1)} \right)^2 + \sum_{i,j} \left(W_{i,j}^{(2)} \right)^2 \right) \quad (6.56)$$



$\nabla_{W^{(1)}} J$ を求める

まず、 H の勾配を求める

$$\nabla_H J = \mathbf{G} W^{(2)T}$$

これに $U^{(1)}$ の 0 未満の要素に対応する勾配の要素を 0 にしたものを \mathbf{G}' とすると、

$$\nabla_{W^{(1)}} J = \underline{\mathbf{X}^T \mathbf{G}'} + \underline{2\lambda W^{(1)}}$$

6.5.7 例：誤差逆伝播法による多層パーセプトロンの訓練 (5/5)

計算コスト

- MLPでは、行列積が主な計算コスト
- 順伝播の積和演算の回数 $O(w)$, w を重みの数とする
 - 重み行列の乗算
- 逆伝播の積和演算の回数 $O(w)$
 - 各重みの転置行列の乗算
- メモリのコスト $O(mn_h)$, m をミニバッチの事例数, n_h を隠れユニットの数とする
 - 隠れ層の非線形関数に対する入力（前の例だと H ）を格納

6.5.8 複雑化の要因 (1/2)

本章の誤差逆伝播の説明は単純化されていて、実装はより複雑

出力の数

- ほとんどの実装で、複数のテンソルを返す演算が必要

メモリ消費の制御方法の工夫

- 多数のテンソルの総和を計算する場合
 - (素朴) 1. テンソル毎に計算、メモリに格納 2. 足し合わせる
 - メモリ消費が大きい
 - (工夫) バッファを1つ用意して、計算した値をそのバッファに足し込む
 - メモリ消費のボトルネックを回避

6.5.8 複雑化の要因 (2/2)

データ型への配慮

- float32, float64, int など、それぞれのデータ型に応じた配慮が必要

勾配が定義されない演算への配慮

- ユーザから要求された勾配が未定義かどうかを監視・判断することが重要

6.5.9 深層学習コミュニティ以外での微分 (1/3)

深層学習コミュニティの微分手法は独自の文化

- 自動微分 (automatic differentiation) の分野
 - リバースモード蓄積 (reverse mode accumulation) という技術クラス
 - その中の特殊な手法としての、誤差逆伝播法

連鎖率の部分式の評価順序と計算コスト

- 例) ソフトマックス関数のクロスエントロピー損失 $J = -\sum_i p_i \log q_i$ の微分を考える
 - 代数式を単純化すると、 $\nabla_{z_i} J = q_i - p_i$
 - 誤差逆伝搬にはこのような単純化ができない
- Theano や TensorFlowのようなフレームワークは、既知のパターンを照合し繰り返しグラフを単純化している

6.5.9 深層学習コミュニティ以外での微分 (2/3)

入力サイズと出力サイズと計算コスト

- フォワードモード蓄積：入力数<出力数のとき，計算コストが小さい
- バックワードモード蓄積：入力数>出力数のとき，計算コストが小さい
- 例) ヤコビ行列の行列積 $ABCD$
 - A が多くの行をもち， D が列ベクトルの場合
 - 掛け算を後ろから逆方向に進める（バックワードモード）と，計算コストが小さい
 - 行列とベクトルの積を計算し続けることになるため
 - その逆の場合は，フォワードモードが適している

6.5.9 深層学習コミュニティ以外での微分 (3/3)

コミュニティによる違い

- 機械学習以外のコミュニティ
 - 関数を微分するプログラムを自動的に生成する関数を用いる
- 深層学習のコミュニティ
 - 明示的なデータ構造を利用して計算グラフを表現する
 - (欠点) 各演算に対してbpropメソッドを定義しなければならない
 - ユーザは定義された演算しかできない
 - (利点) 自動的な手法では再現できない方法で速度や安定性を向上可能
 - 各演算に対してカスタマイズした誤差逆伝播のルールを定義可なので

6.5.10 高次の微分 (1/2)

深層学習における高次の微分について

- TheanoとTensorFlowは、高次の微分に対応。
 - シンボリック微分を微分に適用
- 深層学習でスカラー関数の二階微分を求めるのは稀、ヘッセ行列の性質に興味

6.5.10 高次の微分 (2/2)

ヘッセ行列

- 深層学習でヘッセ行列を表現するのは不可能
 - 関数 $f : \mathbb{R}^n \rightarrow \mathbb{R}$ に対するヘッセ行列のサイズは $n \times n$
 - モデルのパラメータ n は数十億にも及ぶ
- 近似値を **クリロフ法 (Krylov methods)** で求める
 - 行列とベクトルの積のみを利用した反復的な手法
 - $$\mathbf{H}\mathbf{v} = \nabla_{\mathbf{x}} \left[(\nabla_{\mathbf{x}} f(\mathbf{x}))^\top \mathbf{v} \right]. \quad (6.59)$$
 - ソフトウェアライブラリで自動に計算可能

6.6 歴史ノート (1/4)

順伝播型ネットワークは非線形関数近似器とみなすことができる

- 数世紀にわたる汎用的な関数近似の進展が含まれる

ニューラルネットワークと学習手法

- 連鎖率 [Leibniz, 1676], 勾配降下法 [Cauchy, 1847]
- パーセプトロンのようなモデル(1940年代)
 - XOR関数を学習することができない等の欠点が指摘 → 分野の勢いが後退
- 多層パーセプトロン非線形関数の学習 [LuCun, 1985; Parker, 1985; Rumelhart et al., 1986]
 - 多層ニューラルネットワークの研究が活発に
 - 「コネクショニズム」学派
 - ニューロン間の接続が記憶と学習の肝として重要視
 - 分散表現の概念が含まれていた [Hinton et al., 1986]

6.6 歴史ノート (2/4)

ニューラルネットワークの性能向上(1986-2015)の理由

- 統計的な汎化の難易度が下がった
 - 背景：サイズの大きいデータ集合
- ニューラルネットワークのサイズが非常に大きくなった
 - 背景：計算機性能の向上とソフトウェア基盤の改善
- アルゴリズムの進歩
 - 損失関数を，平均二乗誤差(1980年代-1990年代)から交差エントロピーに
 - シグモイドとソフトマックスを出力として持つモデルの性能向上
 - 隠れユニットを，シグモイドから区分線形関数に

6.6 歴史ノート (3/4)

隠れユニットの歴史

- 初期は, rectifying nonlinearity (1970年代) が主流
 - シグモイド (1980年代) に置き換わる
 - ニューラルネットワークが非常に小さい間は, シグモイドの方が性能が良かったからだろう
 - 勾配消失問題の心配がないから ?
 - 微分不可能な点を含む活性化関数は割るべきという迷信(2000年初期)
 - rectifying nonlinearity の重要性 (2009, 2011)
 - 小さいデータ集合に対して [Jarrett et al., 2009]
 - rectifying nonlinearity を使うことは, 隠れ層の重みを学習することよりはるかに重要.
 - 正規化線形ネットワークの場合, 重みはランダムで十分
 - 大きいデータ集合に対して[Glorot et al., 2011]
 - 学習で有用な知識を十分に抽出, ランダムな重みより良い性能
 - ReLUは, 神経科学が深層学習アルゴリズムの発展に影響を与えていることの好例
- ※ $\text{ReLU} \in \text{rectifying nonlinearity}$?

6.6 歴史ノート (4/4)

評判

- 順伝播型ネットワークは他の確率モデル等と併用しなければ性能が出ないと考えられていた(2006頃-2012頃)
- ほかの機械学習タスクに適用できる強力な技術 (現在)
 - 教師なし学習を使って教師あり学習を補完 (2006)
 - 教師あり学習を使って教師なし学習を補完することが一般的になった (現在)

順伝播型ネットワークは潜在能力を発揮しきれていない

- より多くのタスクへの適用
- 最適化アルゴリズムとモデル設計の発展によるさらなる性能向上

参考文献

- Deep Learning
 - Ian Goodfellow, Yoshua Bengio, Aaron Courville
 - 日本語版
<https://www.amazon.co.jp/%E6%B7%B1%E5%B1%A4%E5%AD%A6%E7%BF%92-Ian-Goodfellow/dp/4048930621>