中高生でもわかるニューラルネットワーク

芝浦工業大学 数理科学研究会

2018 年 11 月 3 日

※何か不明な点や計算ミス等がありましたら加筆修正しますので指摘をお願いします

制作: AL18018 市毛 竣

目次

1　研究背景

最近何かと話題になっている人工知能. 発端となったのは 2016 年の Google が開発したディープラーニング搭載済みの AI と韓国の囲碁のチャンピョンが囲碁の勝負をし,AI 側が勝利をしたことだろう. では世間で言われているディープラーニングとその根本にあるニューラルネットワークの理論はどのようになっているのか, またその理論をアウトプットすることで自身の理解を深めようと思い発表に至る.

1.1　ニューラルネットワークの小歴史

ニューラルネットワーク自体は 1900 年代中盤にすでに発掘されたものである. ニューラルネットワークの概念はある程度確立されていたものの 1900 年代後半には冷や飯を食っている状態だった.

2　ニューラルネットワークとパーセプトロンとは

2.1　紹介

ディープラーニングに入る前にまず, ニューラルネットワークとパーセプトロンの話をする. ニューラルネットワークとは人の脳神経 (neuron) を参考にしたモデル（学習法) である. そのための例としてパーセプトロンを紹介しよう.

2.2　パーセプトロンの基本構造

パーセプトロンは複数の信号 (イメージとしては川や電流など名が荒れていくもの) を入力として受け取り, 一つの信号を出力する. 出力される値は多くの場合 1 か 0 のみである.

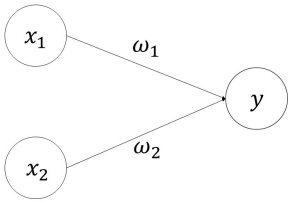

図 1　パーセプトロンの例

具体的にこれを表す関数を作ってみようと思う. 入力される値を x_1, x_2. 重みを ω_1,ω_2, 閾値 (しきいち）を θ とする. 重みとは川の斜面と思ってもらっていい. 斜面が大きいほど水の流れる強さ (値) が大きくなる. 閾値とは出力された値が 1 か 0 か決めるための値である. 例を出すと次のような式である.

$$y = \begin{cases} 0 & (\omega_1 x_1 + \omega_2 x_2 \leqq \theta) \\ 1 & (\omega_1 x_1 + \omega_2 x_2 > \theta) \end{cases} \tag{1}$$

2.3　論理回路とパーセプトロン

ここでは論理回路を用いてパーセプトロンやニューラルネットワークの強みを紹介する. 論理回路には様々なものがあるが, まず初めに AND ゲート,NAND ゲート,OR ゲートを紹介する.

これらはパーセプトロンを使いプログラミングして表すことができる.

XOR ゲート, またの名を' 排他的論理和' とも呼ばれる論理回路だが, これは単一のパーセプトロンを用いてプログラミングで表すのは非常に難しい. ではどうすればいいのかというと, ここでパーセプトロンの強みが出てくる.

表 1　AND ゲート (どちらも 1 の時,1 を出力しそれ以外は 0 を出力する)

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1

表 2　NAND ゲート (AND ゲートの逆.(NO AND))

x_1	x_2	y
0	0	1
0	1	1
1	0	1
1	1	0

表 3　OR ゲート (どちらかに 1 が含まれていれば,1 を出力する)

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	1

表 4　XOR ゲート

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

パーセプトロンの強みは層を重ねることができるというところにある. 今,NAND と OR ゲートを使い得られた結果をそれぞれ s_1, s_2とする.

x_1	x_2	s_1
0	0	1
0	1	1
1	0	1
1	1	0

x_1	x_2	s_2
0	0	0
0	1	1
1	0	1
1	1	1

この時に s_1, s_2 に対して AND ゲートを利用すると

s_1	s_2	y
1	0	0
1	1	1
1	1	1
0	1	0

を得る. つなげてあらわすと

表 5　XOR ゲートの真理値表

x_1	x_2	s_1	s_2	y
0	0	1	0	0
1	0	1	1	1
0	1	1	1	1
1	1	0	1	0

とこのように間接的なやり方で XOR ゲートを表すことができる. このパーセプトロンの状態は下の図のように表すことができる. 層を増やして (入力された数字を変換して) 単層では出来なかった XOR ゲートを表現できた. つまり層を増やして表現できる範囲を広げたことになる. 簡単にディープラーニングを説明すると層を増やして表現できる幅を広げたのがディープラーニングである.

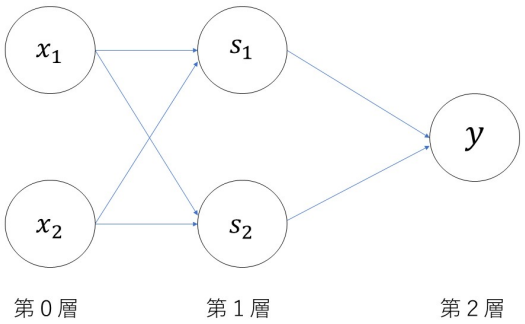

図 2　パーセプトロンの例

また, なぜ一つの式で表せないかというと, 上のパーセプトロンの式は次のように変形できる.

$$\begin{cases} \omega_1 x_1 + \omega_2 x_2 \leqq \theta \\ \omega_1 x_1 + \omega_2 x_2 > \theta \end{cases} \tag{2}$$

$$\begin{cases} x_2 \leqq -\dfrac{\omega_1}{\omega 2} x_1 + \dfrac{\theta}{\omega_2} \\ x_2 > -\dfrac{\omega_1}{\omega 2} x_1 + \dfrac{\theta}{\omega_2} \end{cases} \tag{3}$$

このことを踏まえて先のゲートを確認し下の図を見てみると, AND ゲートのように XOR ゲートは直線でなく曲線でないと分けることはできない. この差は上のパーセプトロンの式 (1) の
で分けられる領域が直線の領域であるためである.

2.4　ニューラルネットワークの基本構造

ニューラルネットワークを図に表すと下の図のようになる. 左の列を入力層, 一番右の列を出力層, 中間の列を中間層と呼ぶ (または隠れ層). 図を見る限りパーセプトロンと似たような構図をしている.

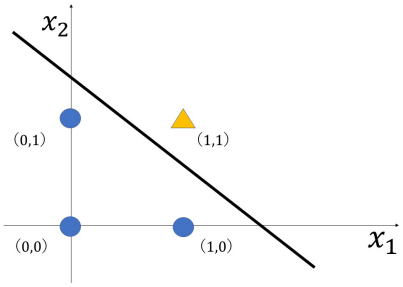

図 3　AND ゲート

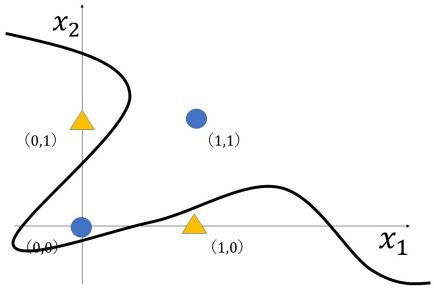

図 4　XOR ゲート

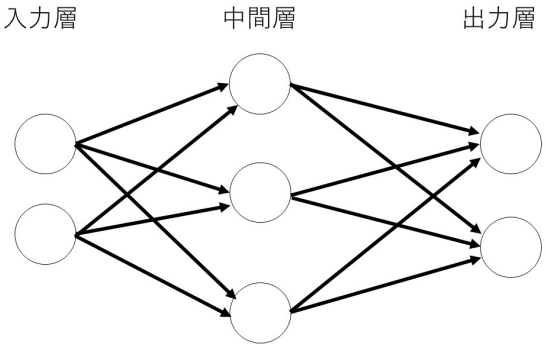

図 5　ニューラルネットの一例

3　活性化関数

3.1　活性化関数の前に

活性化関数の話をする前に (1) の閾値 θ を $-b$ として表す．この b はニューラルネットワーク上ではバイアスと呼ばれ発火 (信号を出力) しやすさを表している. ニューラルネットワークは下のような図になる

$$y = \begin{cases} 0 & \omega_1 x_1 + \omega_2 x_2 + b \leqq 0 \\ 1 & \omega_1 x_1 + \omega_2 x_2 + b > 0 \end{cases} \tag{4}$$

ここで (4) の式をよりシンプルな形にしてみようと思う．場合分けの操作-0 を超えたら 1 を出力し, そうでなければ 0 を出力する操作-を一つの関数で表す．ここでは新しい関数 $h(x)$ という新しい関数を導入する．

$$y = h(\omega_1 x_1 + \omega_2 x_2 + b) \tag{5}$$

$$h(x) = \begin{cases} 0 & x \leqq 0 \\ 1 & x > 0 \end{cases} \tag{6}$$

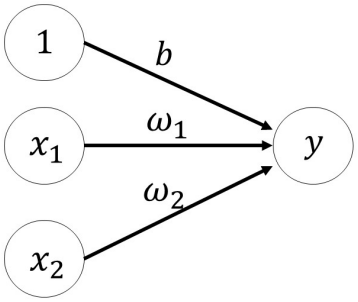

図 6　バイアスありパーセプトロン

ここで出た $h(x)$ という関数だが, このような関数-入力信号の総和を出力信号に変換する関数-は, 一般に活性化関数と呼ばれる. 活性化関数は入力信号の総和がどのように活性化するか (発火するか) を決定する役割がある.
ここで (4) の式を書き換えていく. 式 (4) は重み付きの入力信号の総和を計算し, そしてその和が活性化関数によって変換される, という 2 段階の処理をしている.

$$a = \omega_1 x_1 + \omega_2 x_2 + b \tag{7}$$

$$y = h(a) \tag{8}$$

これら二つの式のニューラルネットは下の図のような形になる. なお, ○の部分はノードと呼ばれる.

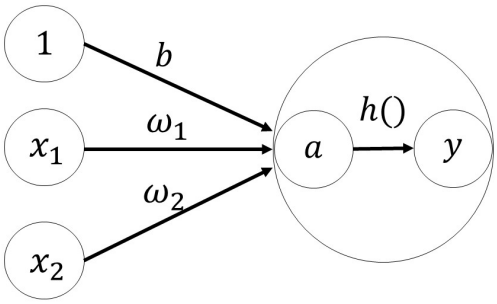

図 7　活性化関数によるプロセス

3.2　ステップ関数とシグモイド関数

式 (6) で表される活性化関数はステップ関数 (階段関数) と呼ばれるものである. つまりパーセプトロンは数ある活性化関数のなかでステップ関数が利用されているということになる. では, ニューラルネットワークでは何が利用されているのか. それはシグモイド関数である.

シグモイド関数

e は自然対数 2.7182...である.

$$h(x) = \frac{1}{1 + \exp(-x)} = \frac{1}{1 + \mathrm{e}^{-x}}$$

漸近線は $h(x) = 1$ と $h(x) = 0$

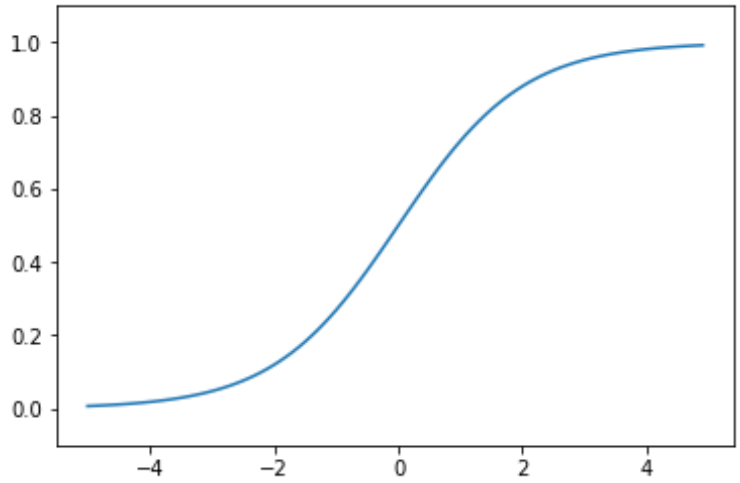

図 8　シグモイド関数のグラフ

上の図がシグモイド関数のグラフとなっている．これを踏まえて (5) の式のステップ関数のグラフを下に記す．両方とも似たようなグラフを描いていることがわかる．違いは 出力される値と滑らかさである．ステップ関数では出力される値は 0 や 1 であるのに対し, シグモイド関数では 0.731...,0.8456... など 0 から 1 の範囲のなかで出力される．この連続的な実数がニューラルネットワークでは重要な役割を持つ．

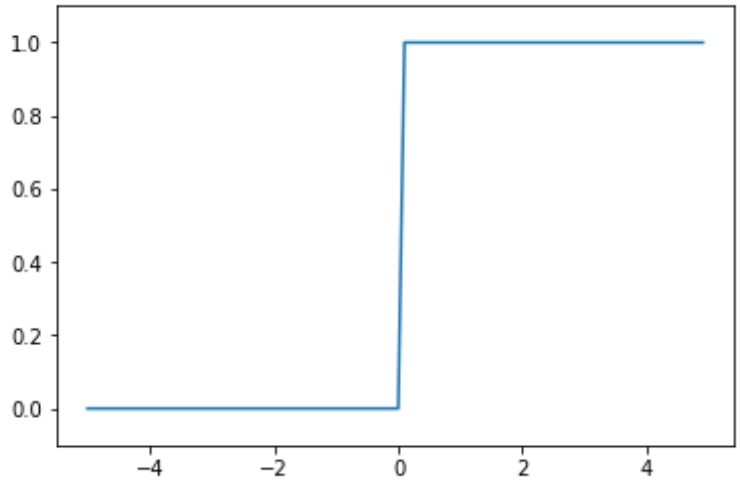

図 9　ステップ関数のグラフ

最近では RELU 関数と呼ばれる活性化関数がニューラルネットワークにおいて主に使われている．理由は単純な関数かつ, 上述の二つの関数よりも早く学習することができ, より良い結果を得られることが多いためである．

RELU 関数

$$h(x) = \begin{cases} x & (x > 0) \\ 0 & (x \leqq 0) \end{cases} \tag{9}$$

3.3　活性化関数と非線形性

次に活性化関数と非線形性について述べる．線形性の定義は次のようになる．

定義 3.1　関数などの演算 f が次の x, y, p を満たすとき, f のそのような性質を線形性という

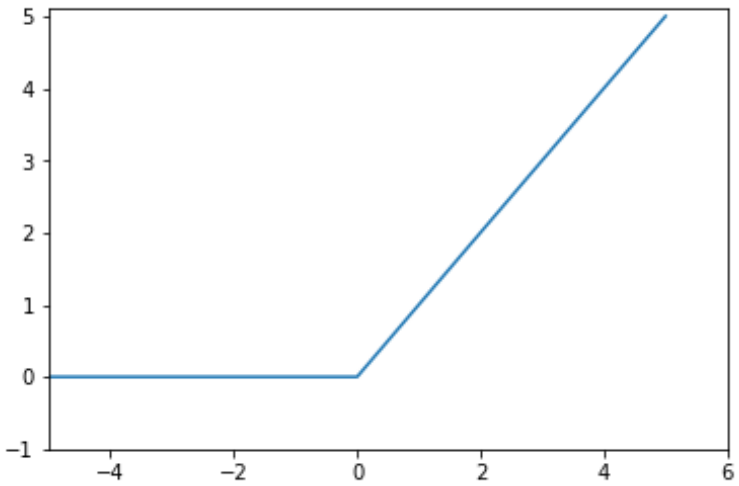

図 10　RELU 関数のグラフ

1 $f(x+y) = f(x) + f(y)$
2 $f(px) = pf(x)$

さて, この線形性が活性化関数とどのような関係があるのか. 結論から言うと活性化関数は非線形の関数である必要がある. それを示すために次の証明を行う.

証明　線形性である演算 f, g があるとき, 合成関数 $f \circ g$(もしくは $g \circ f$) は線形性である.
線形性である条件を f, g に適応する

1 $f(x+y) = f(x) + f(y)$

2 $f(px) = pf(x)$

3 $g(x+y) = gf(x) + g(y)$

4 $g(px) = pg(x)$

これらを使い
$f \circ g(x+y) = f \circ (g(x) + g(y)) = f \circ g(x) + f \circ g(y)$
$f \circ g(px) = f \circ pg(x) = pf \circ g(x)$
が求められる.$g \circ f$ も同様に求めることができる. 以上より題意は満たされた.

例 3.1　活性化関数を $f(x) = cx$ とする.(線形性) 0 層目の入力を x_1, x_2, x_3 1 層目出力を y_1, y_2, 2 層目の出力を z_1, z_2. $\omega_{ji}^{(k)}$ は k 層目の i 番目の入力ノードから j 番目の出力ノードに向けての重みとすると

1 $y_1 = c(\omega_{11}^{(1)} x_1 + \omega_{12}^{(1)} x_2 + \omega_{13}^{(1)} x_3)$

2 $y_2 = c(\omega_{21}^{(1)} x_1 + \omega_{22}^{(1)} x_2 + \omega_{23}^{(1)} x_3)$

3 $z_1 = c(\omega_{11}^{(2)} y_1 + \omega_{12}^{(2)} y_2) = c^2(\omega_{11}^{(2)}(\omega_{11}^{(1)} x_1 + \omega_{12}^{(1)} x_2 + \omega_{13}^{(1)} x_3) + \omega_{12}^{(2)}(\omega_{21}^{(1)} x_1 + \omega_{22}^{(1)} x_2 + \omega_{23}^{(1)} x_3)) = c^2((\omega_{11}^{(2)} \omega 11^{(1)} + \omega_{12}^{(2)} \omega 21^{(1)}) x_1 + (\omega_{11}^{(2)} \omega_{12}^{(1)} + \omega_{12}^{(2)} \omega_{22}^{(1)}) x_2 + (\omega_{11}^{(2)} \omega_{13}^{(1)} + \omega_{12}^{(2)} \omega_{23}^{(1)}) x_3)$

$$4\ z_1 = c(\omega_{21}^{(2)} y_1 + \omega_{22}^{(2)} y_2) = c^2(\omega_{21}^{(2)}(\omega_{11}^{(1)} x_1 + \omega_{12}^{(1)} x_2 + \omega_{13}^{(1)} x_3) + \omega_{22}^{(2)}(\omega_{21}^{(1)} x_1 + \omega_{22}^{(1)} x_2 + \omega_{23}^{(1)} x_3)) = c^2((\omega_{21}^{(2)}\omega 11^{(1)} + \omega_{22}^{(2)}\omega 21^{(1)})x_1 + (\omega_{21}^{(2)}\omega_{12}^{(1)} + \omega_{22}^{(2)}\omega_{22}^{(1)})x_2 + (\omega_{21}^{(2)}\omega_{13}^{(1)} + \omega_{22}^{(2)}\omega_{23}^{(1)})x_3)$$

[3][4] の x_i の左の括弧内の重みをまとめて $\acute{\omega}_i$ とまとめると結局は $f(x) = c^2 x$ で表すことができる.

つまりは活性化関数が線形であると層を深くしても単層のみでできるため, ニューラルネットワークの強みが出ず, 層を深くしても表現力があがらなくなってしまう.

3.4 出力層の設計

機械学習 (ニューラルネットワークを使う深層学習はこのひとつ) の問題は, '分類問題' と' 回帰問題' に分けることができる.

分類問題

データがどのクラスに属するか. という問題

例) 人の写った画像から, そのような人が男性か女性のどちらであるかを分類する

回帰問題

ある入力データから（連続的な）数値の予測をする. という問題

例) 人の写った画像から, その人の体重を予測する (57.4kg など)

ニューラルネットワークは, 以上の分類問題と回帰問題の両方に用いることができる. ただし, どちらの問題を用いるかによって出力層の活性化関数を変更する必要がある. 一般的には分類問題にはソフトマックス関数. 回帰問題には恒等関数を使う.

3.5 恒等関数とソフトマックス関数

回帰問題に使われる恒等関数は, そのまま出力する. 入ってきたものに対して何も手を加えない関数, それが恒等関数である. そのため, 出力層で恒等関数を用いるときは, 入力信号をそのまま出力することになる. プロセスは下の図のようになる. (σ は活性化関数... 今の状態は恒等関数を表している)

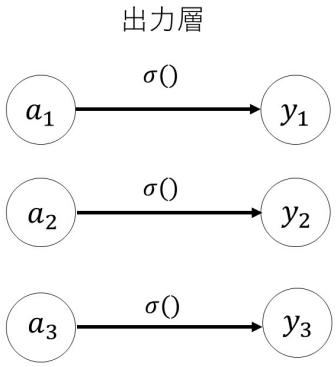

図 11 恒等関数

分類問題で使われるソフトマックス関数は, 次の式で表す.

$$y_k = \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} \tag{10}$$

ここでは出力層のノードが全部で n 個あるとして, k 番目の出力 y_k をもともめる計算式を表している. プロセスは下の図のようになっている. 図の通りソフトマックス関数はすべての入力信号から矢印による結びつきがある. また

式から分かる通り, この式を確率としてみることができる. という重要な性質を持っている.

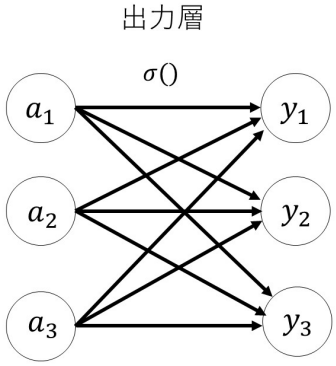

図 12　ソフトマックス関数

3.6　ソフトマックス関数の改善

知っての通り, ニューラルネットワークはコンピュータなどを使い演算される. そこでコンピュータを使う上で問題とされるものがオーバーフローである. ソフトマックス関数では, 指数関数の計算を行うことになるが, その際, 指数関数の値が容易に大きくなってしまう. 例えば e^{10} は 20000 を超え, e^{100} では正 (せい) 超し (10 の 40 乗), e^{1000} ではコンピュータ上で inf... 無限が返ってくる. このような大きな値で計算してしまうとどうしても値が不安定になってしまう. そのため, 次のような式に変形する.

$$\begin{aligned} y_k &= \frac{\exp(a_k)}{\sum_{i=1}^{n} \exp(a_i)} \\ &= \frac{C \exp(a_k)}{C \sum_{i=1}^{n} \exp(a_i)} \\ &= \frac{\exp(a_k + \log C)}{\sum_{i=1}^{n} \exp(a_i + \log C)} \\ &= \frac{\exp(a_k + A)}{\sum_{i=1}^{n} \exp(a_i + A)} \end{aligned} \tag{11}$$

例えば a_i の中の最大の数が 1000 だとすると, A の値を 1005 などにするとオーバーフローすることなく計算することができる.

3.7　ニューラルネットワークの動きまとめ

ニューラルネットワーク上で行われている計算を上の図のまとめておく. $h(), \sigma()$ は活性化関数. 各値の右上の () は層の番号, 右下の二桁目の数は次層のノードの番目, 一桁目は前層のノードの番目を表している.

4　ニューラルネットワークの学習

4.1　学習

よく言われる機械学習とはその名の通り人の介入を極力避け, 集められたデータから答え (パターン) を見つけようとする, 試みである. その一つであるニューラルネットの学習とは先に説明した重みやバイアスを決定することである. ニューラルネット, いわゆる深層学習は従来の機械学習よりも人の介入を避けることができるという重要な性質を持つ. データは先に説明した入力層に送られる. つまり機械学習においてデータはなくてはならないものである.

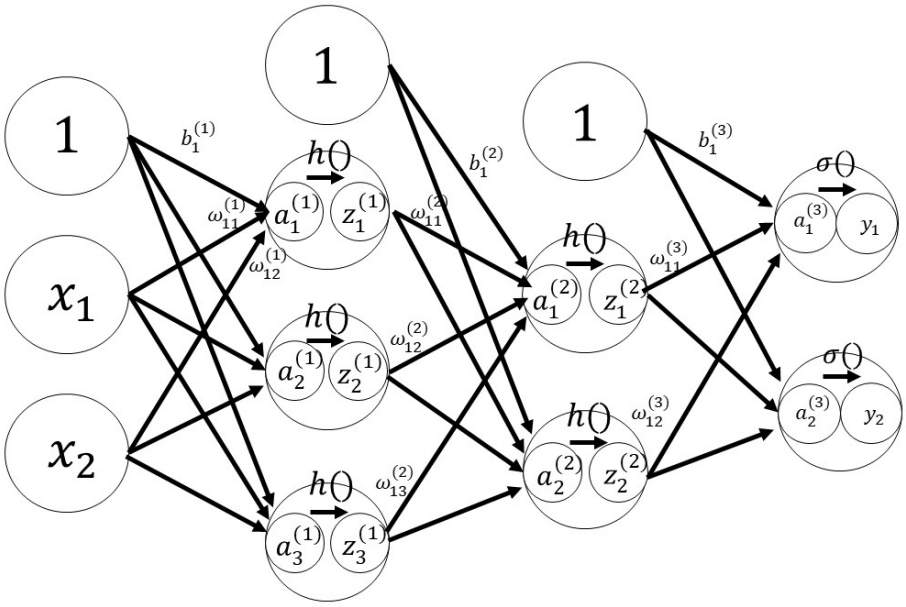

図 13　入力データ 2 個　出力データ 2 個のニューラルネットの動き

4.2　訓練データとテストデータ

機械学習の問題では, 訓練データとテストデータにわけて学習や実験を行うのが一般的である. 訓練データで学習 (重みやバイアスを決定) し, そしてテストデータでそのモデルを評価する. なぜテストデータが必要かというと, 求められているのは汎用的な能力（汎化能力）, つまりまだ見ぬデータに対しての能力であるためである. 手書き文字をある人だけのデータを使い学習したとしても, その他の人にも使えなかったら実用性がないことを思い浮かべてみてほしい. また, 訓練データの学習をしすぎて訓練データにしか適用できない状態を過学習という. この過学習を避けることは機械学習においての重要な課題である.

4.3　損失関数

ニューラルネットワークではある一つの指標によって現在の状態を表す. どれだけ精度が良いかわかり, その指標を基準にして最適な重みパラメータを探索する. それが損失関数である. 重みを探索する際その損失関数が小さくなるようにする. 一般的には 2 乗和誤差や交差エントロピー誤差などが使われる.

4.4　2 乗和誤差と交差エントロピー誤差

2 乗和誤差は損失関数を E, 出力された値を y_k, 教師データ (訓練データ) の値を t_k とすると次のようにあらわされる.

$$E = \frac{1}{2}\sum_k (y_k - t_k)^2 \tag{12}$$

例えば出力された値 (ソフトマックス関数で確率化) と教師データをそれぞれ

$$[y_0, y_1, y_2, ...y_9] = [0.1, 0.05, 0.6, 0.0, 0.05, 0.1, 0.0, 0.1, 0.0, 0.0]$$

$$[t_0, t_1, t_2, ...t_9] = [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]$$

とすると. 値は $E = 0..009750.....$ となる. (ちなみに今回教師データの場合は 1 と書いているのを正解の部分としている. この表記を one-hot 表現という) 別の場合だと

$$[y_0, y_1, y_2, ...y_9] = [0.1, 0.05, 0.1, 0.0, 0.05, 0.1, 0.0, 0.6, 0.0, 0.0]$$

(正解部分の確率が低い) のとき, $E = 0.5975000....$ となり確かに損失関数の値はは高くなっている (適合していない). つぎに交差エントロピー誤差ににについて述べる. 交差エントロピー誤差は次の式で表される.

$$E = -\sum_k t_k \log y_k \tag{13}$$

上の出力データと訓練データを使うとそれぞれ 0.51308... と 2.3025... となりこれまでの議論と一致する.

4.5 微分

突然だがランナーが 2km を 10 分で走るとすると, 速さは 0.2[km/s] となる. これは 10 分間の平均の速さを計算している. では 10 分を 1 分, 1 秒, 0.0001 秒... と小さくしていくとある瞬間の速さを求めることができる. このある瞬間の変化の量を微分という. 数式で次のように定義される.

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{14}$$

左辺の $\frac{df(x)}{dx}$ は, $f(x)$ の x についての微分–x に対する $f(x)$ の変化の度合い–を表す記号である. x の小さな変化に対して $f(x)$ がどれだけ変化するかということを意味する.

4.6 損失関数の微分

ニューラルネットワークの学習では, 最適なパラメータ（重みとバイアス) を探索する際に損失関数の値をできるだけ小さくなるように探索する. そのために損失関数の微分 (正確には勾配) を計算し, それを手掛かりにパラメータを更新する. 重み損失関数に対しての微分は「その重みパラメータを少しだけ変化させたとき, 損失関数がどのように変化するのか」を表す. 例えば微分の値が負の値であったとき. 重みの値を正の方向に変化させることで損失関数を減少させることができる. 逆に, 微分の値が正の時重みを負の方向に変化させることで損失関数を減らすことができる. これを表す式を下に記す.

$$\omega = \omega - \eta \frac{\partial L}{\partial \omega} \tag{15}$$

ηは学習率 (パラメータの更新量) と呼ばれる.一回の学習でどれだけ学習すべきか, パラメータを更新するかを決めることを表す. $\frac{\partial L}{\partial \omega}$は重み$\omega$の微小な変化に対しての損失関数の L の変化の量を表している.[*1] このような重みパラメータの更新 10 回, 100 回... と繰り返し (繰り返し続けしすぎると過学習となってしまう), 損失関数が小さくなるように重みパラメータを決めていくことを勾配法（特に勾配降下法) と呼ぶ.

4.7 学習アルゴリズムのまとめ

では一度どのように学習しているか整理してみよう. 基本的なステップは次のようになる.

ステップ 1

訓練データの中からランダムに一部のデータを選び出す. (同じデータだけを使わないため)
重みの初期値をランダムに決める[a]

[a] ここでは重みの初期値に関してはいろいろな方法があるがここでは議論しない. 気になる人は Xavier の初期値や He の初期値などを調べてほしい.

ステップ 2

出力されたデータと訓練データからなる損失関数を減らす重みの勾配を求める (微分して変化の量を求める).

[*1] 表現が違うが, やっていることは微分と似たようなことをやっている. ここでは省略するので詳しくは偏微分や勾配で調べてほしい

ステップ 3

求めた変化の量を使いパラメータを更新する.

$$\omega = \omega - \eta \frac{\partial L}{\partial \omega} \tag{16}$$

ステップ 4

ステップ 1 から 3 を繰り返して最適な重みパラメータを探していく.

ステップ 5

重みが決まったらそれを使いテストデータで汎化能力 (まだ見ぬデータに対しての能力) を確認する.

学習の過程がどのようになっているのかの例を下の図に乗せる. iteration が学習した回数, loss が損失関数となっている. 図の通り学習を続けると損失関数の値はだんだんと減っていっている.

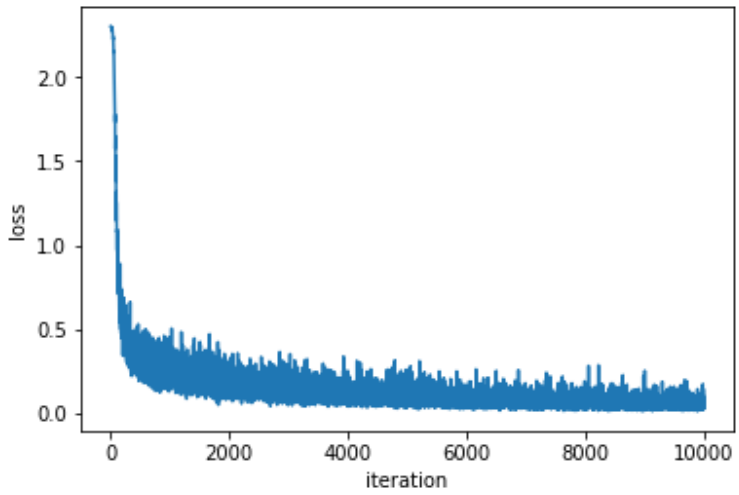

図 14　損失関数の推移

そして学習によって得られた重みパラメータを使いどのくらい適合しているのか (汎化能力があるのか) を表す図を下に乗せる. accuracy は精度, epoch は学習を途中の期間を表す. 図を見るとこのモデルは非常に高い精度を持っていることになる.

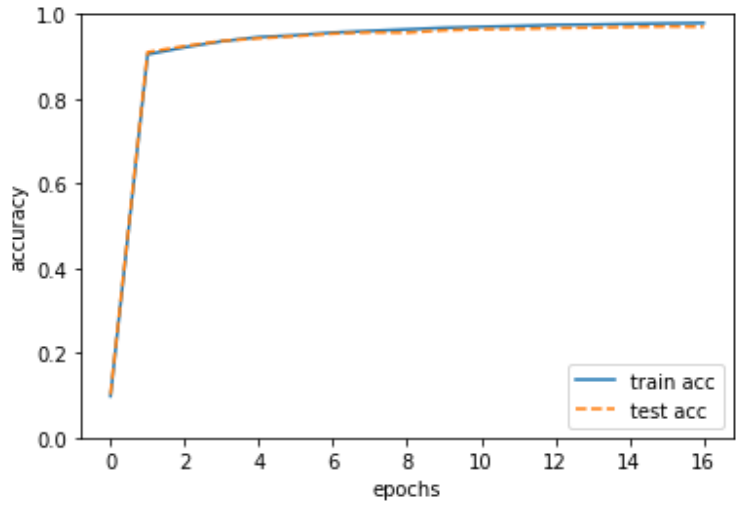

図 15　訓練データとテストデータに対する認識精度の推移

5　総括

今回紹介したニューラルネットワークの理論はまだまだスタート地点に過ぎない．というのもこの理論は 1900 年代後半にすでに確立されたものなので決して最新の理論ではない．重みの初期値や学習率の最適な値, 勾配消失問題など様々な課題が解決されつつあるのが今の状態であり, それによってニューラルネットワーク, 深層学習が日の目を浴びているのである．まだまだ分からないことが多いニューラルネットワーク．少しでも興味を持っていただけたら幸いだ．

6　今後について

同じ機械学習の中に強化学習というものがある．この強化学習では教師データなし (代わりに報酬というものを使う) で学習を行う．大体の感覚としては人間が試行錯誤して学習する–自転車に何回も乗って乗れるようになる–というものである．今度はこの強化学習について学び, ゆくゆくは深層強化学習と呼ばれる理論につなげていこうと思う．

参考文献

[1] ゼロから作る Deep Learning -Python で学ぶディープラーニングの理論と実装-・斎藤 康毅
[2] 深層学習 (機械学習プロフェッショナルシリーズ)・岡谷 貴之