

# ラピッドチャレンジ「課題レポート」

佐藤晴一

2021 年 5 月 27 日

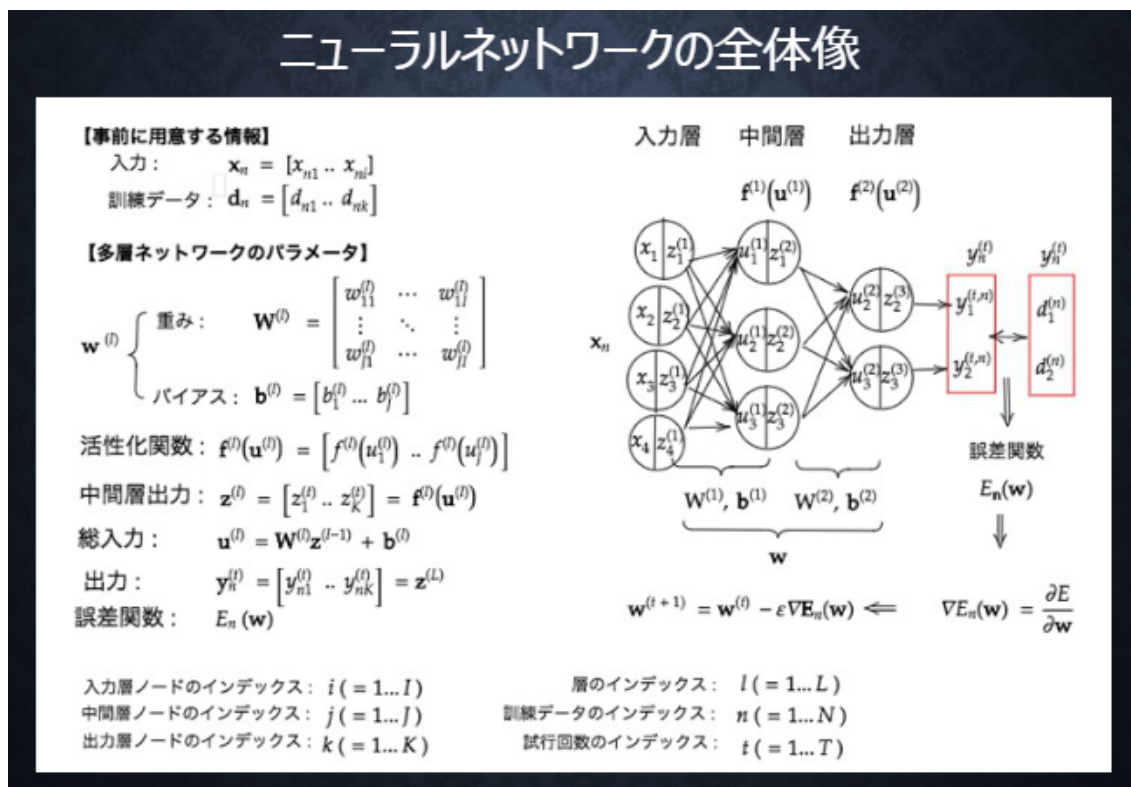
# 第1章

## 深層学習 Day1

ディープラーニングは、結局何をやろうとしているか。それは「明示的なプログラムの代わりに多数の中間層を持つニューラルネットワークを用いて、入力値から目的とする出力値に変換するを数学モデルを構築すること。」にある。また、ニューラルネットワークにおける最終目的は、重み  $w$  とバイアス  $b$  の最適化にある。

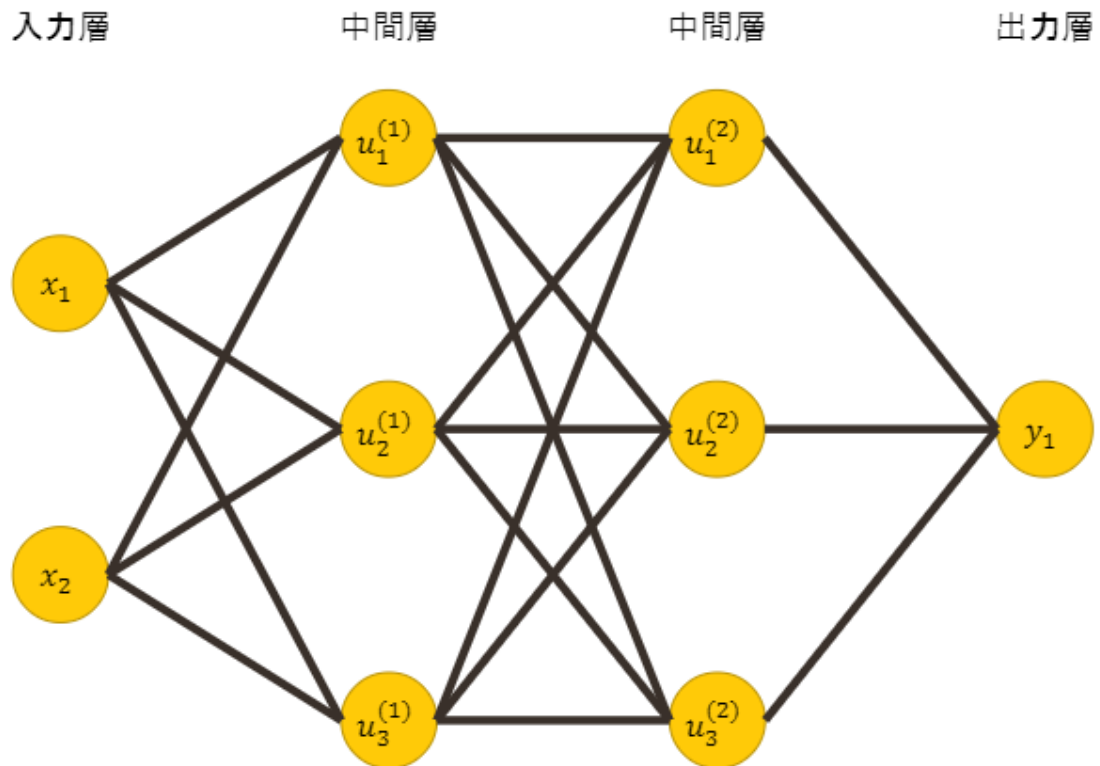
**ニューラルネットワークの全体像** 人間の視覚野に関する神経伝達プロセスをヒントにパーセプトロンの概念が生まれ、ニューラルネットワークはそれを応用し、諸問題を「入力層」、「中間層」、「出力層」という各層とそれらをつなぐノード（weight 含む）、バイアス等により定式化したもの。

与えられたデータに最も適合した形で識別や分類をするための学習の段階で、ノード間の weight を誤差関数の最小化によって学習器を実現する。この際、誤差逆伝播法の手法を用いて、誤差関数の最小化に導く。iteration 毎の計算は、機械学習で学んだ勾配降下法と同じである。



### 確認テスト

次のネットワークをかけ。入力層（2 ノード 1 層）、中間層（3 ノード 2 層）、出力層（1 ノード 1 層）



### ニューラルネットワークでできること

大きな枠組みでいうと、識別と生成モデルに区分される。以下は、識別モデルに特化してその一例を列挙する。

1. 「回帰」 結果予想（売上予想、株価予想）、ランキング（競馬順位予想、人気順位予想）
2. 「分類」 猫写真の判別、手書き文字認識、花の種類分類

### ニューラルネットワーク

#### 1. 回 帰

連続する実数値を取る関数の近似。具体的な回帰分析手法の一例は次の通り。

Ex. 線形回帰、回帰木、ランダムフォレスト、ニューラルネットワークがあげられる。

#### 2. 分 類

性別（男あるいは女）や動物の種類など離散的な結果を予想するための分析。具体的な分類手法の一例は次の通り。

Ex. ベイズ分類、ロジスティック回帰、決定木、ランダムフォレスト、ニューラルネットワーク）

### 深層学習の実用例

自動売買（トレード）、チャットボット、翻訳、音声解釈、囲碁、将棋 AI

## 1.1 入力層～中間層

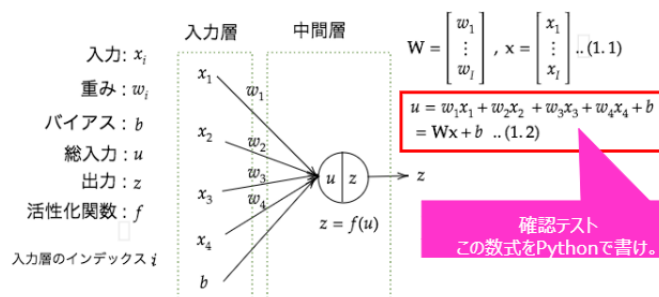
### ■要点（まとめ）

各特徴量のデータセット（数値）を受け取る場所を「ノード」と呼ぶ。特に、一番初めのデータ入力を行う層を「入力層」と呼ぶ。またデータの要素毎の重要度の軽重をつけて、それを「重み  $w$ 」と呼ぶとともに、「バイアス」と呼ばれるいわゆるペデスタル量を加えて、線形結合したものを「中間層」と呼ばれる層に入力する。

最終的にニューラルネットワーク（NN）の最適化を図る際は、重み  $w$  とバイアス  $b$  のパラメータを調整（更新）することに相当する\*1。

$$u = w \cdot x + b$$

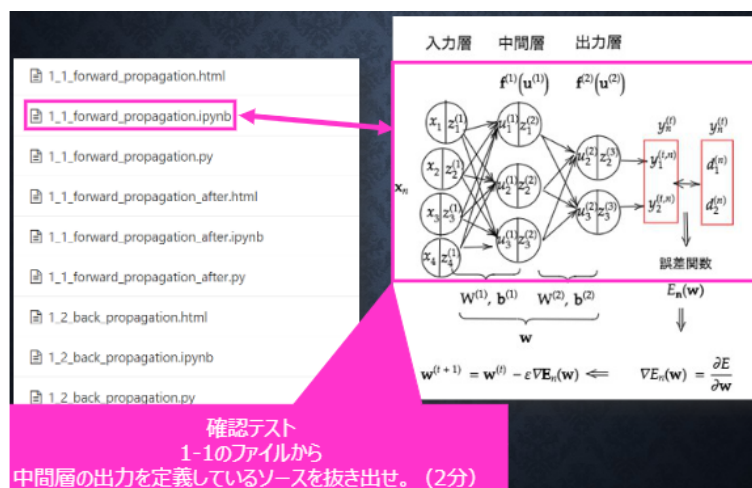
### 確認テスト



Ans.

$$u1 = \text{np.dot}(x, W1) + b1$$

### 確認テスト



Ans.

$$\begin{cases} z = \text{functions.relu}(u) & \text{@順伝播 (単層・単ユニット)} \\ z = \text{functions.sigmoid}(u) & \text{@順伝播 (単層・複数ユニット)} \end{cases}$$

\*1  $x - yx$  平面の直線の式で表現すれば、傾きが重み、切片がバイアスに相当し、NN による学習により、データに適合した傾き、切片を計算することになる。

## ■実装演習成果（キャプチャ、サマリー、考察）

---

```
1 # インポートと関数定義
2 import numpy as np
3 from common import functions
4
5 # 順伝播（単層・単ユニット）
6
7 # 重み
8 W = np.array([[0.1], [0.2]])
9
10 # バイアス
11 b = 0.5
12
13 # 入力値
14 x = np.array([2, 3])
15
16 # 総入力
17 u = np.dot(x, W) + b
18
19 # 中間層出力
20 z = functions.relu(u)
```

---

```
1 # 順伝播（3層・複数ユニット）
2
3 # ウェイトとバイアスを設定
4 # ネットワークを作成
5 def init_network():
6     print("##### ネットワークの初期化#####")
7     network = {}
8
9     #試してみよう_
10    #各パラメータのを表示_shape
11    #ネットワークの初期値ランダム生成_
12
13    network['W1'] = np.array([
14        [0.1, 0.3, 0.5],
15        [0.2, 0.4, 0.6]
16    ])
17    network['W2'] = np.array([
18        [0.1, 0.4],
19        [0.2, 0.5],
20        [0.3, 0.6]
21    ])
22    network['W3'] = np.array([
23        [0.1, 0.3],
```

```

24         [0.2, 0.4]
25     ])
26     network['b1'] = np.array([0.1, 0.2, 0.3])
27     network['b2'] = np.array([0.1, 0.2])
28     network['b3'] = np.array([1, 2])
29
30     print_vec重み("1", network['W1'])
31     print_vec重み("2", network['W2'])
32     print_vec重み("3", network['W3'])
33     print_vecバイアス("1", network['b1'])
34     print_vecバイアス("2", network['b2'])
35     print_vecバイアス("3", network['b3'])
36
37     return network
38
39 # プロセスを作成
40 # : 入力値 x
41 def forward(network, x):
42
43     print("##### 順伝播開始#####")
44
45     W1, W2, W3 = network['W1'], network['W2'], network['W3']
46     b1, b2, b3 = network['b1'], network['b2'], network['b3']
47
48     # 1層の総入力
49     u1 = np.dot(x, W1) + b1
50
51     # 1層の総出力
52     z1 = functions.relu(u1)
53
54     # 2層の総入力
55     u2 = np.dot(z1, W2) + b2
56
57     # 2層の総出力
58     z2 = functions.relu(u2)
59
60     # 出力層の総入力
61     u3 = np.dot(z2, W3) + b3
62
63     # 出力層の総出力
64     y = u3
65
66     print_vec総入力("1", u1)
67     print_vec中間層出力("1", z1)
68     print_vec総入力("2", u2)
69     print_vec出力("1", z1)

```

```

70     print出力合計(": " + str(np.sum(z1)))
71
72     return y, z1, z2
73
74 # 入力値
75 x = np.array([1., 2.])
76 print_vec入力("", x)
77
78 # ネットワークの初期化
79 network = init_network()
80
81 y, z1, z2 = forward(network, x)

```

---

## サマリ

コード実装はシンプルな内容である。NN の要図を段階的に入力層から中間層へ導く計算過程をプログラミングすればよい。今後、Python プログラミングで共通的な内容となるが、冒頭で numpy というライブラリや Scikit-learn などの有力な計算フレームワークを import することが重要である。

独特な行列入力形式 (array) や行列の掛け算形式 (dot) などの表現に慣れていくことが必要である。また、今回は、別途 function (各種活性化関数) を定義したファイルを冒頭で from で読み込んでいることにも注意が必要である。

## 1.2 活性化関数

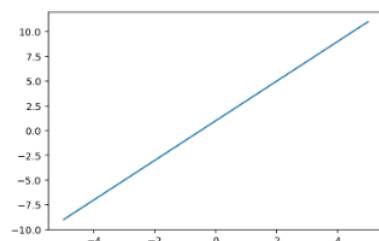
### ■要点 (まとめ)

ニューラルネットワークにおいて、次の層への出力の大きさを決める非線形の関数。入力値の値によって、次の層への信号の ON/OFF や強弱を定める働きをもつ。先ほどの入力層から中間層への計算では、線形な計算結果 ( $\mathbf{W}$  と  $\mathbf{x}$  の内積) であったが、この活性化関数 (非線形関数) を用いることでバラエティに富んだ出力 (識別) を獲得することができる。

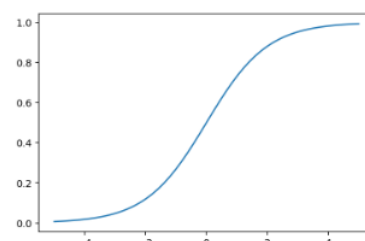
### 確認テスト

線形と非線形の違いを図にかいて簡易に説明せよ。

Ans.



線形な関数



非線形な関数

ちなみに、非線形関数は、線形関数が満たす以下の加法性や斉次性の数学的性質を満たさない。

$$\text{加法性: } f(x + y) = f(x) + f(y), \quad \text{斉次性: } f(kx) = kf(x)$$

## 中間層用の活性化関数

### 1. ステップ関数

閾値を超えたら発火する関数であり、出力は常に1か0。パーセプトロン（ニューラルネットワークの前身）で利用された関数。課題は、0 - 1 の間を表現できず、線形分離可能なものしか学習できなかった。現在ではほぼ使われていない（歴史的遺物）。

### 2. シグモイド（ロジスティック）関数

0 - 1 の間を緩やかに変化する関数で、ステップ関数では ON/OFF しかない状態に対し、信号の強弱を伝えられるようになり、予想ニューラルネットワーク普及のきっかけとなった。課題は、大きな値では出力の変化が微小なため、勾配消失問題を引き起こす事があった。

### 3. ReLU 関数

今最も使われている活性化関数勾配消失問題の回避とスパース化に貢献することで良い成果をもたらしている。

## ■実装演習成果（キャプチャ、サマリー、考察）

```
1 import numpy as np
2
3 # 中間層の活性化関数
4 # シグモイド関数（ロジスティック関数）
5 def sigmoid(x):
6     return 1/(1 + np.exp(-x))
7
8 # 関数 ReLU
9 def relu(x):
10     return np.maximum(0, x)
11
12 # ステップ関数（閾値）0
13 def step_function(x):
14     return np.where( x > 0, 1, 0)
```

## 確認テスト

配布されたソースコードより該当する箇所を抜き出せ。 ※ 中間層の出力  $z = f(u)$

Ans.

$z = \text{functions.sigmoid}(u)$

## サマリ

Python プログラミングに記法に乗り取り、各活性化関数を実装したもの。ReLU 及びステップ関数はその定義からして、特異な表式になっているので、注意が必要。実装時は、自ら定義する関数の名前、引数、return（戻り値）で出力する計算式を正確に書き下す必要がある。



## 1.3 出力層

### ■要点（まとめ）

中間層の出力は、次の中間層の入力となっており、最後に出力層の役割としては我々が知りたい識別の情報を得るための判断基準を計算結果としてアウトプットする部分。

実際に NN に学習させるときには、データの特徴量を入力層からインプットして中間層を経て「出力層まで計算した結果」と「訓練データ（人間が作成した正解ラベルデータ）」を比較すればよい。

### 1.3.1 誤差関数

一般的に、分類の時は、「クロスエントロピー誤差」を用いて、回帰の時は、「平均二乗誤差」を用いる。

**2乗誤差** 実装では、`mean_squared_error` で計算している。

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^n (y_i - d_i)^2$$

**交差（クロス）エントロピー**

$$E(\mathbf{w}) = - \sum_{i=1}^n d_i \log y_i$$

ここで、 $d_i$  は訓練データの正解ラベル、 $y_i$  は NN で出力した結果。

#### 確認テスト

なぜ、引き算でなく二乗するか述べよ。また、 $1/2$  はどういう意味を持つか述べよ

Ans.

まず 2 乗する理由引き算を行うだけでは、各ラベルでの誤差で正負両方の値が発生し、全体の誤差を正しくあらわすのに都合が悪い。2 乗してそれぞれのラベルでの誤差を正の値になるようにする。

また、 $1/2$  にする理由は、実際にネットワークを学習するときに行う誤差逆伝搬の計算で、誤差関数の微分を用い際の計算式を簡単にするためであり、本質的な意味はない。

---

```
1 # 誤差関数
2 # 平均二乗誤差
3 def mean_squared_error(d, y):
4     return np.mean(np.square(d - y)) / 2
5
6 # 回帰
7 # ネットワーク 2-3-2
8
9 # ウェイトとバイアスを設定
10 # ネットワークを作成
11 def init_network():
12     print("##### ネットワークの初期化#####")
13
14     network = {}
15     network['W1'] = np.array([
```

```

16         [0.1, 0.3, 0.5],
17         [0.2, 0.4, 0.6]
18     ])
19     network['W2'] = np.array([
20         [0.1, 0.4],
21         [0.2, 0.5],
22         [0.3, 0.6]
23     ])
24     network['b1'] = np.array([0.1, 0.2, 0.3])
25     network['b2'] = np.array([0.1, 0.2])
26
27     print_vec重み("1", network['W1'])
28     print_vec重み("2", network['W2'])
29     print_vecバイアス("1", network['b1'])
30     print_vecバイアス("2", network['b2'])
31
32     return network
33
34 # プロセスを作成
35 def forward(network, x):
36     print("##### 順伝播開始#####")
37
38     W1, W2 = network['W1'], network['W2']
39     b1, b2 = network['b1'], network['b2']
40     # 隠れ層の総入力
41     u1 = np.dot(x, W1) + b1
42     # 隠れ層の総出力
43     z1 = functions.relu(u1)
44     # 出力層の総入力
45     u2 = np.dot(z1, W2) + b2
46     # 出力層の総出力
47     y = u2
48
49     print_vec総入力("1", u1)
50     print_vec中間層出力("1", z1)
51     print_vec総入力("2", u2)
52     print_vec出力("1", y)
53     print出力合計(": " + str(np.sum(z1)))
54
55     return y, z1
56
57 # 入力値
58 x = np.array([1., 2.])
59 network = init_network()
60 y, z1 = forward(network, x)
61 # 目標出力

```

```

62 d = np.array([2., 4.])
63 # 誤差
64 loss = functions.mean_squared_error(d, y)
65 ## 表示
66 print("\n#### 結果表示####")
67 print_vec中間層出力("", z1)
68 print_vec出力("", y)
69 print_vec訓練データ("", d)
70 print_vec誤差("", loss)

```

---

### 1.3.2 出力層の活性化関数

出力層の中間層との違いは次の通り。

#### 値の強弱

1. 中間層：閾値の前後で信号の強弱を調整し、必要な情報を抽出するもの。
2. 出力層：抽出した情報を我々が理解しやすい形に変換したもの。信号の大きさ（比率）はそのままに変換する。

#### 確率出力

1. 分類問題の場合、出力層の出力は 0 – 1 の範囲に限定し、総和を 1 とする必要がある。

以上のように、求める性質の相違から出力層と中間層で利用される活性化関数が異なる。

#### 出力層用の活性化関数

1. ソフトマックス関数  $k$  クラス分類とした場合、 $k$  クラス分類のそれぞれの確率の総和が 1 となるような定義。
2. 恒等写像
3. シグモイド（ロジスティック）関数

全結合NN –出力層の種類			
	回帰	二値分類	多クラス分類
活性化関数	恒等写像 $f(u) = u$	シグモイド関数 $f(u) = \frac{1}{1 + e^{-u}}$	ソフトマックス関数 $f(i, u) = \frac{e^{u_i}}{\sum_{k=1}^K e^{u_k}}$
誤差関数	二乗誤差	交差エントロピー	
<b>【訓練データサンプルあたりの誤差】</b> $E_n(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^I (\mathbf{y}_n - \mathbf{d}_n)^2 \quad \dots \text{二乗誤差}$ $E_n(\mathbf{w}) = - \sum_{i=1}^I d_i \log y_i \quad \dots \text{交差エントロピー}$		<b>【学習サイクルあたりの誤差】</b> $E(\mathbf{w}) = \sum_{n=1}^N E_n$	

### 確認テスト

ソフトマックス関数の数式に該当するソースコードを示し、一行ずつ処理の説明をせよ。

Ans.

```
def softmax(x):
    if x.ndim == 2:
        x = x.T
        x = x - np.max(x, axis=0)
        y = np.exp(x) / np.sum(np.exp(x), axis=0)
        return y.T
    x = x - np.max(x) # オーバーフロー対策
    return np.exp(x) / np.sum(np.exp(x))
```

基本的な numpy のライブラリを用いた計算であり、序盤の if 文は、データセットの種類に対して幅広く対応するための前処理のコード。次にオーバーフロー対策のコードを含み、最後に、numpy の関数 (exp) を用いてソフトマックス関数の定義に基づき、演算させている。

### 確認テスト

交差エントロピー関数の数式に該当するソースコードを示し、一行ずつ処理の説明をせよ。

Ans.

```
def cross_entropy_error(d, y):
    if y.ndim == 1:
        d = d.reshape(1, d.size)
        y = y.reshape(1, y.size)
        # 教師データがone-hot-vectorの場合、正解ラベルのインデックスに変換
        if d.size == y.size:
            d = d.argmax(axis=1)
        batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) / batch_size
```

基本的な numpy のライブラリを用いた計算であり、序盤の if 文は、データセットの種類に対して幅広く対応するための前処理のコード。最後に、numpy の関数 (log) を用いて交差エントロピー関数の定義に基づき、演算させている。特にバッチ数に応じた除算をしている点に注意が必要。また、クロスエントロピーの計算において、実装時に  $1e-7$  とあるのは、log 関数が 0 の近傍で  $-\infty$  に飛ぶことを避けるために設定している。

### ■実装演習成果（キャプチャ、サマリー、考察）

---

```
1 # 出力層の活性化関数
2 # ソフトマックス関数
3 def softmax(x):
4     if x.ndim == 2:
5         x = x.T
6         x = x - np.max(x, axis=0)
7         y = np.exp(x) / np.sum(np.exp(x), axis=0)
8         return y.T
9
10    x = x - np.max(x) # オーバーフロー対策
11    return np.exp(x) / np.sum(np.exp(x))
```

```

12
13 # ソフトマックスとクロスエントロピーの複合関数
14 def softmax_with_loss(d, x):
15     y = softmax(x)
16     return cross_entropy_error(d, y)
17
18 # 誤差関数
19 # 平均二乗誤差
20 def mean_squared_error(d, y):
21     return np.mean(np.square(d - y)) / 2
22
23 # クロスエントロピー
24 def cross_entropy_error(d, y):
25     if y.ndim == 1:
26         d = d.reshape(1, d.size)
27         y = y.reshape(1, y.size)
28
29     # 教師データが one-hot-の場合、正解ラベルのインデックスに変換 vector
30     if d.size == y.size:
31         d = d.argmax(axis=1)
32
33     batch_size = y.shape[0]
34     return -np.sum(np.log(y[np.arange(batch_size), d] + 1e-7)) / batch_size

```

---

## 1.4 勾配降下法 (Gradient descent)

### ■要点 (まとめ)

深層学習のそもそもの目的は、学習を通して誤差を最小にするネットワークを生成することであった。言い換えれば、誤差関数  $E(\boldsymbol{w})$  を最小化するパラメータ  $\boldsymbol{w}$  を発見することである。具体的には、勾配降下法を利用してパラメータを最適化を図る。この際、一度にすべての学習データ使って、パラメータ更新を行っている (バッチ学習)。

### 1.4.1 勾配降下法

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \epsilon \nabla E, \quad \text{where} \quad \nabla E = \frac{\partial E}{\partial \boldsymbol{w}}$$

ここで  $\epsilon$  は学習率 (learning rate) と呼ばれるハイパーパラメータでモデルのパラメータの収束しやすさを調整するものである<sup>\*2</sup>

学習率が大きすぎた場合、最小値にいつまでもたどり着かず発散してしまう。一方、学習率が小さい場合発散することはないが、小さすぎると収束するまでに時間がかかってしまうので適用にあたっては経験が必要。

---

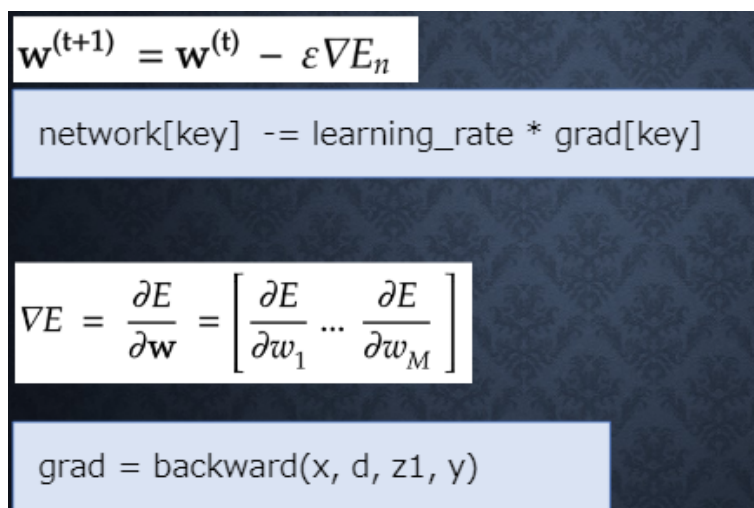
<sup>\*2</sup> <https://qiita.com/masatomix/items/d4e5fb3b52fa4c92366f>。

初心者の内は、誤差関数の収斂状況を出力させることでチェックが行える\*3。

#### 確認テスト

該当するソースコードを探してみよう。

Ans.


$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E_n$$

```
network[key] -= learning_rate * grad[key]
```

$$\nabla E = \frac{\partial E}{\partial \mathbf{w}} = \left[ \frac{\partial E}{\partial w_1} \dots \frac{\partial E}{\partial w_M} \right]$$

```
grad = backward(x, d, z1, y)
```

#### 各種アルゴリズム

勾配降下法の学習率の決定、収束性向上のためのアルゴリズムについて複数の論文が公開され、よく利用されている。以下、一例を示す。

1. Momentum
2. AdaGrad：更新量が自動的に調整され、学習が進むと学習率が小さくなる。
3. RMSprop：AdaGrad アルゴリズムにおける学習が進むにつれて、急速に学習率が低下するという問題を解決したアルゴリズム
4. Adadelta
5. Adam：勾配との指数移動平均を用いるアルゴリズム

#### 1.4.2 確率的勾配降下法 (SGD)

勾配降下法（全サンプルデータの平均誤差を計算）とは異なり、次式で与えられるようにランダムに抽出した  $n$  番目のサンプルデータの誤差を計算し、次ステップ（エポック）に進む。この際、誤差関数の値を小さくする方向に重み  $\mathbf{w}$  及びバイアス  $\mathbf{b}$  を更新していくことになる。

これは、パソコンのメモリ負荷を軽減するためであり、一般的に機械学習で使用する PC のメモリは 48G や 64G, 128G 程度。大量のデータを扱う深層学習では、メモリ不足が生起するため、オンライン学習によって少しずつデータを使用してパラメータ更新をして行く方が効率が良くなる。

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \nabla E_n, \quad \text{where} \quad \nabla E = \frac{\partial E}{\partial \mathbf{w}}$$

\*3 Prof. Andrew Ng も courcera 「Machine Learning」 内でこの点について言及していた。

確率的勾配降下法には、以下のメリットがある。

1. データが冗長な場合の計算コストの軽減
2. 望まない局所極小解に収束するリスクの軽減
3. オンライン（逐次）学習<sup>\*4</sup>ができる

## 確認テスト

オンライン学習とは何か 2 行でまとめよ。

Ans.

学習データが入ってくるたびに都度パラメータを更新し、学習を進めていく方法。一方、バッチ学習では一度にすべての学習データを使ってパラメータ更新を行う。

---

```
1 # 確率勾配降下法
2 import sys, os
3 sys.path.append(os.pardir) # 親ディレクトリのファイルをインポートするための設定
4 import numpy as np
5 from common import functions
6 import matplotlib.pyplot as plt
7
8 def print_vec(text, vec):
9     print("*** " + text + " ***")
10    print(vec)
11    #print("shape: " + str(x.shape))
12    print("")
13
14
15 # In[ ]:
16
17
18 # サンプルとする関数
19 #の値を予想する yAI
20
21 def f(x):
22     y = 3 * x[0] + 2 * x[1]
23     return y
24
25 # 初期設定
26 def init_network():
27     # print("##### ネットワークの初期化#####")
28     network = {}
29     nodesNum = 10
30     network['W1'] = np.random.randn(2, nodesNum)
31     network['W2'] = np.random.randn(nodesNum)
32     network['b1'] = np.random.randn(nodesNum)
```

---

<sup>\*4</sup> 毎回データを少し（1 つずつ）使ってパラメータを更新する手法。

```

33     network['b2'] = np.random.randn()
34
35     # print_vec重み ("1", network['W1'])
36     # print_vec重み ("2", network['W2'])
37     # print_vecバイアス ("1", network['b1'])
38     # print_vecバイアス ("2", network['b2'])
39
40     return network
41
42 # 順伝播
43 def forward(network, x):
44     # print("##### 順伝播開始#####")
45
46     W1, W2 = network['W1'], network['W2']
47     b1, b2 = network['b1'], network['b2']
48     u1 = np.dot(x, W1) + b1
49     z1 = functions.relu(u1)
50
51     ## 試してみよう
52     #z1 = functions.sigmoid(u1)
53
54     u2 = np.dot(z1, W2) + b2
55     y = u2
56
57     # print_vec総入力 ("1", u1)
58     # print_vec中間層出力 ("1", z1)
59     # print_vec総入力 ("2", u2)
60     # print_vec出力 ("1", y)
61     # print出力合計 ("": " + str(np.sum(y)))
62
63     return z1, y

```

---

### 1.4.3 ミニバッチ勾配降下法

バッチ（一括）学習とオンライン学習の間とったものであり、毎回全データの中から一部（ミニバッチ）を取り出してパラメータ更新を行う手法である。確率的勾配降下法が「ランダムに抽出したサンプルの誤差」を計算していたのに対して、ミニバッチ勾配降下法では、「ランダムに分割したデータの集合（ミニバッチ） $D_t$  に属するサンプルの平均誤差」を用いる点にある。このとき、 $N_t$  をバッチ数という。例えば、10 万枚の画像データを 500 枚毎のバッチに分けた場合、2,000 個のミニバッチにデータを分割して計算していくことになる。

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \epsilon \nabla E_t, \quad \text{where} \quad E_t = \frac{1}{N_t} \sum_{n \in D_t} E_n, \quad N_t = |D_t|$$

ミニバッチ勾配降下法のメリットとしては、確率的勾配降下法のメリットを損なわず、計算機の計算資源を有効利用できることにある。特に CPU を利用したスレッド並列化や GPU を利用した SIMD(Single

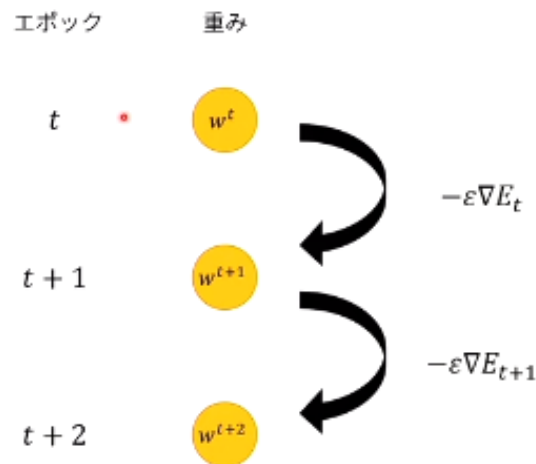


Instruction Multi Data) 並列化など\*5。

### 確認テスト

先ほど示したミニバッチ勾配降下法の数式の意味を図に書いて説明せよ。

Ans.



## 1.5 誤差逆伝播法

### ■要点 (まとめ)

誤差勾配の計算に、数値微分を用いると以下ようになる。

$$\frac{\partial E}{\partial w_m} \sim \frac{E(w_m + h) - E(w_m - h)}{2h}$$

原理的には、パラメータ更新は数値微分を用いても計算は可能であるが、1つ1つの  $w$  に対して繰り返し同じ計算を大量にしないといけないデメリットが生起する。

そこで、考えられたのが「誤差逆伝播法」である。これは、算出された誤差を、出力層側から順に微分し、前の層前の層へと伝播させるもの。最小限の計算で各パラメータでの微分値を解析的に計算する手法である\*6。

### 微分の連鎖率

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} \frac{\partial y}{\partial u} \frac{\partial u}{\partial w}$$

### 確認テスト

誤差逆伝播法では不要な再帰的处理を避ける事が出来る。既に行った計算結果を保持しているソースコードを抽出せよ。

Ans.

以下の実装コード (10-27 行) を参照。

\*5 1 エポックの間では、パラメータ更新を行っていないので、並列計算が可能になる。

\*6 計算結果 (=誤差) から微分を逆算することで、不要な再帰的計算を避けて微分を算出できる。

## ■実装演習成果（キャプチャ、サマリー、考察）

```
1 # 誤差逆伝播
2 def backward(x, d, z1, y):
3     # print("\n##### 誤差逆伝播開始#####")
4
5     grad = {}
6
7     W1, W2 = network['W1'], network['W2']
8     b1, b2 = network['b1'], network['b2']
9
10    # 出力層でのデルタ
11    delta2 = functions.d_mean_squared_error(d, y)
12    # の勾配 b2
13    grad['b2'] = np.sum(delta2, axis=0)
14    # の勾配 W2
15    grad['W2'] = np.dot(z1.T, delta2)
16    # 中間層でのデルタ
17    #delta1 = np.dot(delta2, W2.T) * functions.d_relu(z1)
18
19    ## 試してみよう
20    delta1 = np.dot(delta2, W2.T) * functions.d_sigmoid(z1)
21
22    delta1 = delta1[np.newaxis, :]
23    # の勾配 b1
24    grad['b1'] = np.sum(delta1, axis=0)
25    x = x[np.newaxis, :]
26    # の勾配 W1
27    grad['W1'] = np.dot(x.T, delta1)
28
29    # print_vec偏微分 ("重み_1", grad["W1"])
30    # print_vec偏微分 ("重み_2", grad["W2"])
31    # print_vec偏微分 ("バイアス_1", grad["b1"])
32    # print_vec偏微分 ("バイアス_2", grad["b2"])
33
34    return grad
35
36 # サンプルデータを作成
37 data_sets_size = 100000
38 data_sets = [0 for i in range(data_sets_size)]
39
40 for i in range(data_sets_size):
41     data_sets[i] = {}
42     # ランダムな値を設定
43     data_sets[i]['x'] = np.random.rand(2)
```

```

44
45     ## 試してみよう入力値の設定_
46     # data_sets[i]['x'] = np.random.rand(2) * 10 -5 # ~のランダム数値-55
47
48     # 目標出力を設定
49     data_sets[i]['d'] = f(data_sets[i]['x'])
50
51 losses = []
52 # 学習率
53 learning_rate = 0.07
54
55 # 抽出数
56 epoch = 1000
57
58 # パラメータの初期化
59 network = init_network()
60 # データのランダム抽出
61 random_datasets = np.random.choice(data_sets, epoch)
62
63 # 勾配降下の繰り返し
64 for dataset in random_datasets:
65     x, d = dataset['x'], dataset['d']
66     z1, y = forward(network, x)
67     grad = backward(x, d, z1, y)
68     # パラメータに勾配適用
69     for key in ('W1', 'W2', 'b1', 'b2'):
70         network[key] -= learning_rate * grad[key]
71
72     # 誤差
73     loss = functions.mean_squared_error(d, y)
74     losses.append(loss)
75
76 print("##### 結果表示#####")
77 lists = range(epoch)
78
79
80 plt.plot(lists, losses, '.')
81 # グラフの表示
82 plt.show()

```

---

## 1.6 Appendix

### ■ディープラーニングの開発環境

1. ローカル (PC が家にある環境) : CPU、GPU (並列処置)
2. クラウド (データセンターにある PC を間借り) : AWS、GCP

PC に入っている部品が大切。大量のデータを扱い、数値計算を高速にやるために開発環境は以下のように発展し、計算速度が進展してきた。

CPU(どんな PC) < GPU(ゲーム PC) < FPGA(自分でプログラムできる計算機) < ASIC(自分でプログラムできない)

ここで ASIC は、業者に発注して専門的な計算だけに特化したもの。自らプログラムの改変はできない。各部品の価格は、GPA は 2 万円、FPGA は 10 万円程度。ASIC になると数億円程度の費用がかかる\*7。

現在は、まずは、GPU で深層学習を行うモデリングを実験し、次いでクラウド環境で TPU をクラウド環境で使うことがスタンダードな開発環境となっている。

### ■入力層の設計

入力としてとり得るデータは、数字の集まりであればなんでも良い。例えば、

1. 連続する実数
2. 確率
3. フラグ値

ここでフラグ値  $[0, 0, 1]$  のようなデータを「one-hot-label」と呼ぶ。

一方、入力層として取るべきでないデータは次の通り。

1. 欠損値が多いデータ
2. 誤差の大きいデータ
3. 出力そのもの、出力を加工した情報
4. 連続性の無いデータ (背番号とか)
5. 無意味な数が割り当てられているデータ

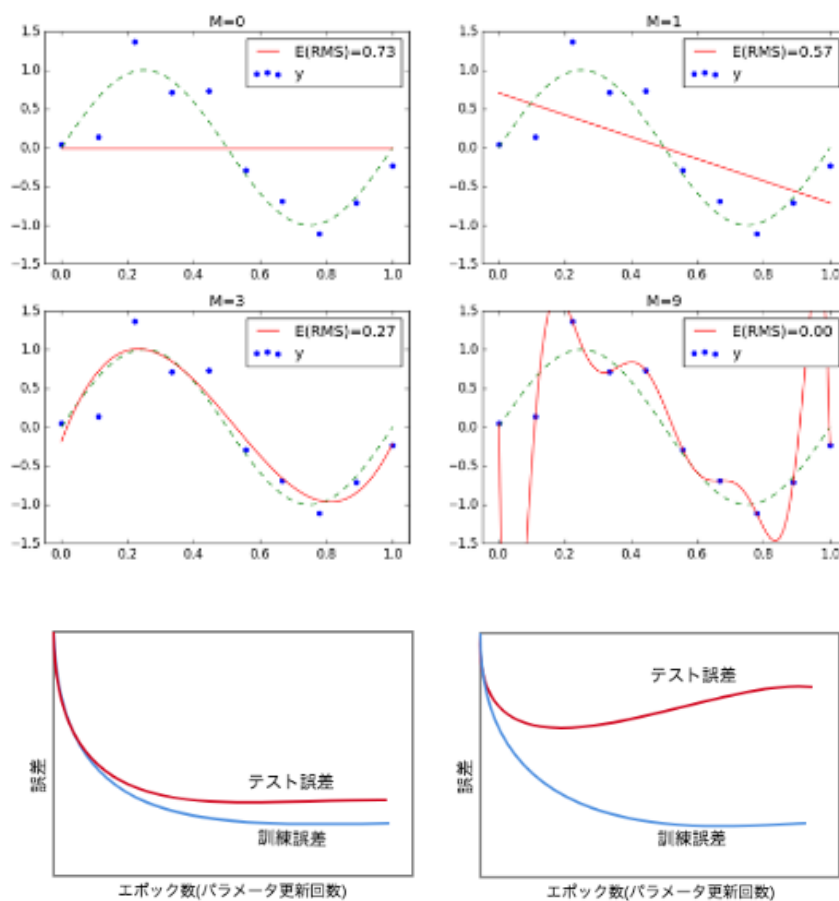
実際は、欠損値等を含むデータを取り扱うことも多い。次のような前処理が考えられている。

1. 欠損値の扱い
  - (a) ゼロで詰める (ただし...)
  - (b) 欠損値を含む集合を除外 (欠損値を含むデータを 1 行まると除外)
  - (c) 入力として採用しない (欠損値を含む特徴量を含む 1 列を除外)
2. データの結合
3. 数値の正規化・正則化

---

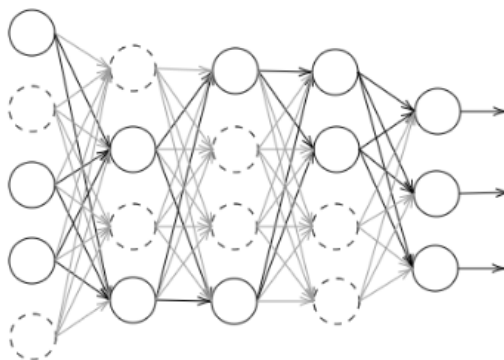
\*7 Google が開発した TPU が有名

## ■過学習



## ■ドロップアウト

Dropout とは、ニューラルネットワークの学習時に、一定割合のノードを不活性化させながら学習を行うことで過学習を防ぎ（緩和し）、精度をあげるために手法。ニューラルネットワークは訓練データに対するトレース能力に優れており、わりと簡単に過学習を起こしてしまうため、正則化や Dropout のような手法を用いることは重要である。具体的な Dropout のイメージは以下の図のようになる。



## ■データ集合の拡張 (Dataset Augmentation)

データセットが少ないという状況は、機械学習の際によく生起する。この時に、用いる手法がデータ拡張 (Data Augmentation) である。単純に、元のデータセットからデータを水増しする内容である。特に、分類タスク（画像認識）に効果が高いことが知られている。一方で、それ以外のタスク、例えば、密度推定のためのデータは水増しができない。水増しするためには、密度の情報が必要。（例えば、身長分布の分布が知りたければ、正しく身長のデータセットを集めなければならない。）

学習データが不足するときに人工的にデータを作り水増しする手法



**データ拡張の注意点** データ拡張の際の注意点としては、データ拡張の結果、データセット内で混同するデータが発生しないようにする必要がある。元データで割り当てられていた正解ラベルに対して、データ拡張の結果、ご認識を与える入力データを生成する可能性があるため。

### データ拡張の種類



**データ拡張の適用** 最後に、データ拡張の効果とモデル性能かの見極めが重要である。

1. データ拡張の効果と性能評価
  - (a) データ拡張を行うとしばしば劇的に汎化性能が向上する。
  - (b) ランダムなデータ拡張を行うときは学習データが毎度異なるため再現性に注意。
2. データ拡張とモデルの捉え方
  - (a) 一般的に適用可能なデータ拡張（ノイズ付加など）はモデルの一部として捉える（ドロップアウトなど）。
  - (b) 特定の作業に対してのみ適用可能なデータ拡張（クロップなど）は入力データの事前加工として捉える。例: 不良品検知の画像識別モデルに製品の一部だけが拡大された画像は入力されない。

## 参考文献

- [1] 奥村晴彦, 黒木裕介 『L<sup>A</sup>T<sub>E</sub>X 2<sub>ε</sub> 美文書作成入門 第 7 版』(技術評論社, 2017)
- [2] 加藤公一, 『機械学習のエッセンス』(SB クリエイティブ, 2018)
- [3] 大重美幸, 『Python 3 入門ノート』(ソーテック, 2018)
- [4] 大関真之, 『機械学習入門』(オーム, 2018)
- [5] Andreas C. Müller et al., 『Python ではじめる機械学習』(オライリージャパン, 2018)
- [6] 加藤公一 (監修), 『機械学習図鑑』(翔泳社, 2019)
- [7] 岡谷貴之, 『深層学習』(講談社, 2015)
- [8] 竹内一郎, 『サポートベクトルマシン』、講談社, 2015)
- [9] 斎藤康毅, 『ゼロから作る Deep Learning』(オライリージャパン, 2019)
- [10] Guido van Rossum, 『Python チュートリアル (第 4 版)』(オライリージャパン, 2021)