@IT
atmarkIT

普通のエンジニアが初めて動かすディープラーニング

一色政彦，デジタルアドバンテージ［著］

01.TensorFlow 2 + Keras（tf.keras）入門：
第 1 回　初めてのニューラルネットワーク実装、まずは準備をしよう
―― 仕組み理解 × 初実装（前編）

02.TensorFlow 2 + Keras（tf.keras）入門：
第 2 回　ニューラルネットワーク最速入門 ―― 仕組み理解 × 初実装（中編）

03.TensorFlow 2 + Keras（tf.keras）入門：
第 3 回　ディープラーニング最速入門 ―― 仕組み理解 × 初実装（後編）

TensorFlow 2 + Keras（tf.keras）入門：

第 1 回　初めてのニューラルネットワーク実装、まずは準備をしよう―― 仕組み理解 × 初実装（前編）

機械学習の勉強はここから始めてみよう。ディープラーニングの基盤技術であるニューラルネットワーク（NN）を、知識ゼロの状態から概略を押さえつつ実装してみよう。まずはワークフローを概観して、データ回りの処理から始める。

一色政彦，デジタルアドバンテージ（2019 年 09 月 19 日）

本連載では、「ニューラルネットワーク」を難しくないものと捉え、できるだけ視覚的かつシンプルに説明を行う。「機械学習を始めてみたいけど、数学や統計から勉強を始めるのは大変だなぁ」と思っている人に、「まずは始めてみよう」というきっかけを与えることを目的として、記事を執筆した。

本連載の読者対象は、完全な初心者を想定している。機械学習の知識はゼロで構わず、数学は中学数学レベル（二次関数など）を見て何となく理解できるレベルで問題ない。Python のコードは出てくるので、不安がある場合は、下記の連載などに目を通すとよいだろう。

- 『機械学習&ディープラーニング入門（概要編）』……「ディープラーニングって何？」という人
- 『同（コンピューター概論編）』……プログラミング未経験の人
- 『同（作業環境準備編）』……Jupyter Notebook ／ Google Colaboratory 未経験の人
- 『同（Python 編）』……Python 未経験の人
- 『同（データ構造編）』……NumPy 未経験の人
- 『Python 入門』……Python をマスターしたい人

ニューラルネットワークは難しくない

図 0-1　あい博士「ニューラルネットワークは難しくない!」、マナブくん「えー……」

→目次に戻る

「**ディープラーニング**」や、その基盤技術である「**ニューラルネットワーク**」について学び始めて、途中で挫折した人は少なくないだろう。実際に、ニューラルネットワークの解説を読むと、高校や大学以来、遠ざかっていた数式のオンパレードで、機械学習時の各ステップで用いられる各要素の技術／理論も多種多様である。まずこれに、面食らう人が少なくない。次に、そういった各要素を網羅的に勉強して、完全な知識としていくのは、それなりに時間のかかる大変な作業となる。このため、「本 1 冊最後まで読んでコードも書いてみたけど、全体的にはやっぱりよく分からない」という感想を持つ人も多いのではないだろうか。

確かに、実務で思い通りにディープラーニングを活用できるようになるまでには、やはり各要素技術／理論を十分に理解することが不可欠で、それでようやく技術や理論を適切に使い分けられるようになる。しかしながら、ニューラルネットワーク全体を概観してみると、やっていることはワンパターンで、コード内容も難しくないことに気付く。よって、とりあえずそのパターンを理解すれば、「ニューラルネットワークやディープラーニングがどのように機能するか」を理解するためのきっかけや枠組みとなるはずである。

そこで本連載では、最もシンプルなニューラルネットワークを実装しながら、そのパターンを説明する。それによって、「ニューラルネットワークがどのように機能するか」を理解することを目標としたい。

ニューラルネットワークの実装には、よりシンプルなコードとするため、**Python**（バージョン **3.6**）と、ディープラーニングのライブラリ「**TensorFlow**」の最新版 **2.0**（バージョン 2 系）を利用する。TensorFlow には書き方が何種類かあるが、本連載では高水準 API の **Keras**（以下、**tf.keras**）を使用する。開発環境に Google Colaboratory（以下、**Colab**）を用い、ニューラルネットワークの視覚化に「**ニューラルネットワーク Playground - Deep Insider**」（以下、**Playground**）を利用する。

本連載のテーマは『仕組みの理解 × 初めての実装』とし、全 3 回で、以下のタイトルを予定している。

- **前編：初めてのニューラルネットワーク実装、まずは準備をしよう**（今回）
- **中編：ニューラルネットワーク最速入門**
- **後編：ディープラーニング最速入門**

全 3 回の全てのサンプルコードは、下記のリンク先で実行もしくは参照できる。

Google Colabで実行する　　GitHubでソースコードを見る

さっそく前編の説明を始めよう！

ディープラーニングの大まかな流れ

機械学習／ディープラーニングの一般的な作業の流れは、『前掲の連載（概要編）』の「Lesson 3　機械学習&ディープラーニングの、基本的なワークフローを知ろう」という記事でも紹介しているが、本連載では、おおまかに下記の 8 つの工程に分けて、1 ステップずつ進めることにする。

- （1）データ準備
- （2）問題種別
- （3）前処理
- （4）“ 手法 ” の選択：モデルの定義
- （5）“ 学習方法 ” の設計：モデルの生成
- （6）学習：トレーニング
- （7）評価
- （8）テスト

この工程は、Playground（図 0-2）にある赤色の丸数字に対応している。

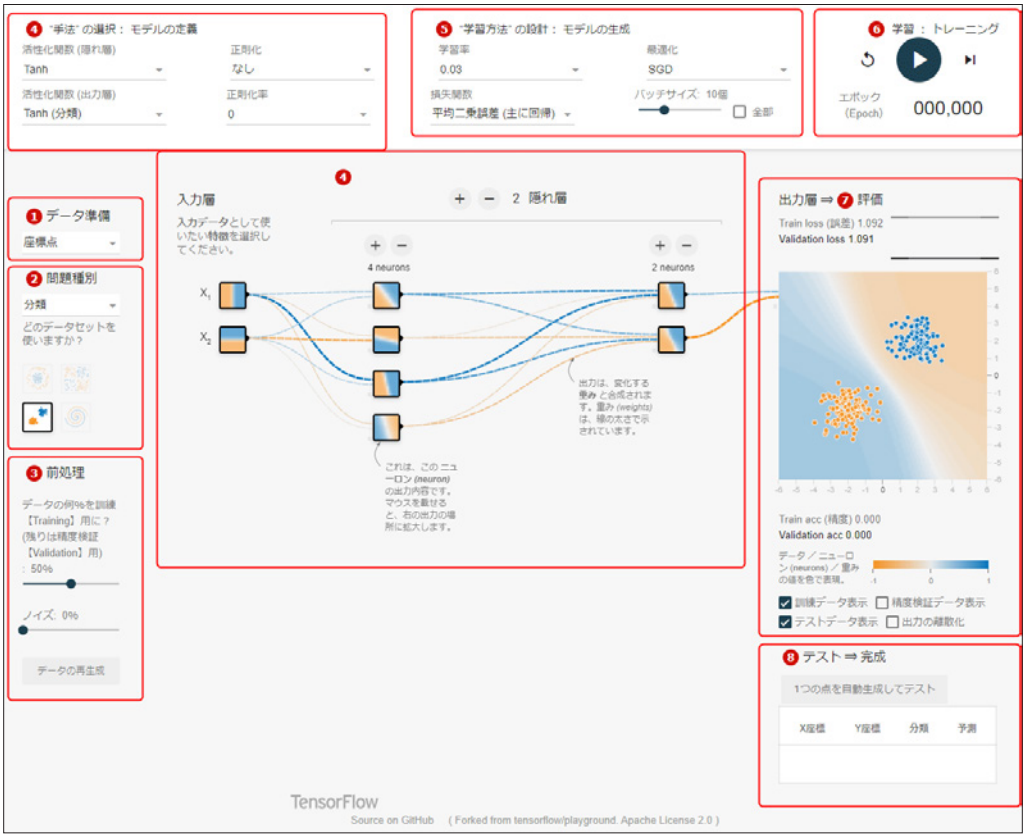

図 0-2　Playground 上に示された 8 つの工程

それでは（1）から順番に見ていこう。次回中編の（4）からニューラルネットワークの説明に入るが、その前に今回の前編ではその準備として（1）～（3）のデータに関する説明をさせてほしい。

→目次に戻る

（1）データ準備

まず、データの準備を行う。事前に、こちらの Playground（表示内容を絞り、初期値を設定済みのもの）をクリックして開いておいてほしい。

Playground による図解

これを Playground 上で実行するには、図 1-1 のように、左上の（1）で「座標点」を選択すればよい（※これしか選択できない）。

図 1-1　［データ準備］で「座標点」を選択

ここでいう**座標点**とは、横の X 軸が **-6.0 ～ 6.0**、縦の Y 軸も **-6.0 ～ 6.0** の数値を取る、2 次元の座標系にたくさんプロット（＝点描画）した全ての点のことである。今回は、このような点々データを使う。図 1-2 がそのプロット例である。

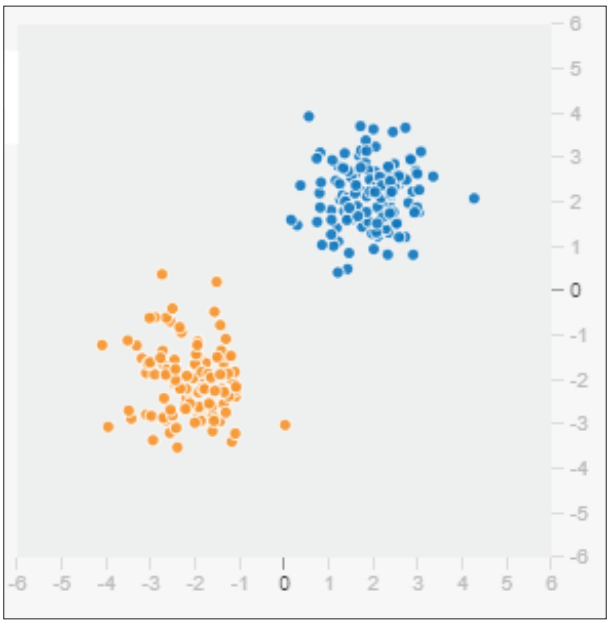

図 1-2　座標点がプロットされた 2 次元座標系

→目次に戻る

青色とオレンジ色の座標点がたくさんプロットされているのが分かる。点の色については後述する。

Python コードでの実装例

それでは、「座標点」データを準備するためのコードを Python で記述してみよう。

点々の生成処理は、ニューラルネットワークとは関係がないので、筆者が実装したライブラリ「playground-data」を使うことにする。このライブラリをインストールするには、Colab で新規ノートブックを作って、1 つのセルに以下を入力して実行してほしい（※ Colab ノートブックの作成方法はこちら、使い方はこちらを参照してほしい）。

```
!pip install playground-data
```

リスト 1　座標点データを生成するライブラリのインストール

実際のデータ生成は、ステップ（3）で行う。

（2）問題種別

次に、問題種別を選択する。

Playground による図解

これを Playground 上で実行するには、図 2-1 のように、左側の（2）で「**分類（Classify）**」を選択すればよい（※「**回帰（Regression）**」も選択できるが、今回は説明を割愛する）。

図 2-1　［問題種別］で「分類」を選択

→目次に戻る

分類というのは、「写真が犬か猫か」を判定する機械学習モデルを想像すると分かりやすいだろう。今回は、「犬か猫か」ではなく、「青色（＝ **1.0**）か、オレンジ色（＝ **-1.0**）か」を判定する。

なお、前掲の図 1-2 で示した 2 次元座標系では、青色の点群は右上の方、オレンジ色の点群は左下の方に偏って存在していた。よって、点群と点群の間に「線」を引けば、線の右上にある点は青色、線の左下にある点はオレンジ色、と予測／推論できる。実際にこれを実現するのがトレーニング（＝学習）後の機械学習モデル（通称：AI）なのである。

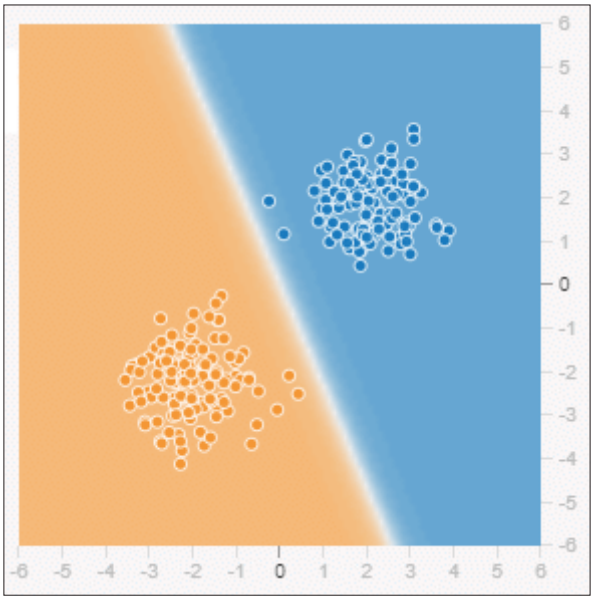

図 2-2　訓練後の 2 次元座標系（白い部分が境界線）

図 2-2 の背景色は、Playground でトレーニング後の機械学習モデルの予測結果を視覚化したものである（※プロットされている各点は、図 1-2 で示したデータと同じものである）。右上の青色の背景と、左下のオレンジ色の背景の間に白い線が出来ているのが分かるだろう。これが、前述の「線」に相当するもので、**決定境界（Decision boundary）**と呼ばれている。

ここまでで、「ニューラルネットワークで作成した機械学習モデルで、分類問題を解く」とは、どのようなことなのかがイメージできるようになっただろう。

なお Playground では、分類か回帰かで、選択できるデータの種類／種別が違う（※記事をコンパクトにするため、その違いの詳細は割愛する。選べる種別については後掲の図 2-4 を参照）。よってここで、データ種別も選択しておく。

今回は、図 1-2 や図 2-2 で示した 2 群に分かれて集まっている点データを使う。Playground 上でこれを行うには、図 2-3 のように［どのようなデータセットを使いますか？］欄で「ガウシアン（Gaussian）」（＝正規分布の「山」形状のように、ある基点に集中して分布するデータ）を選択すればよい。

図 2-3 ［どのようなデータセットを使いますか?］で「ガウシアン（Gaussian）」を選択

ここでは「ガウシアン」を選択したが、それ以外のデータセットの表示例を参考までに図 2-4 にまとめた。

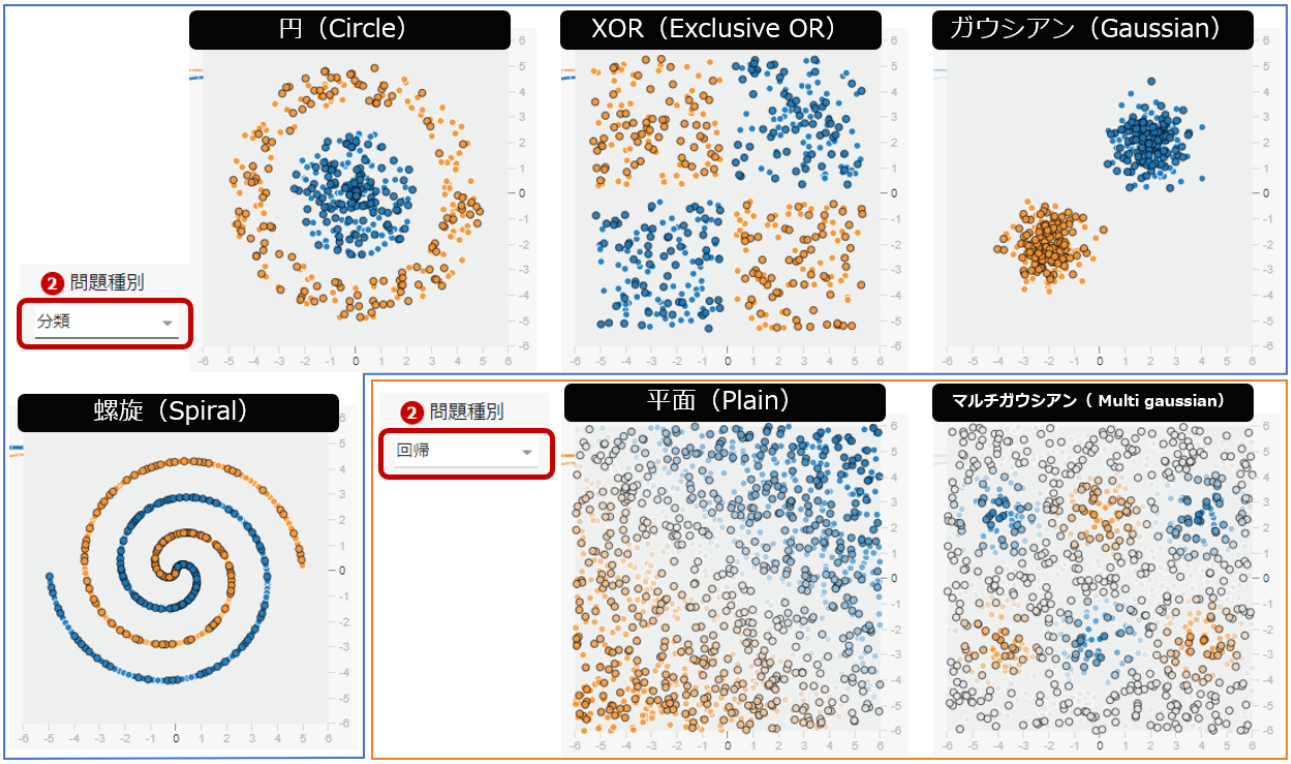

図 2-4 ［どのようなデータセットを使いますか?］で選択できるデータセットの表示例

分類（＝例えば犬猫のような離散的なデータの予測。この例では「青色か、オレンジ色か」）の場合は、円／ XOR ／ガウシアン／螺旋（らせん）が選択可能。

回帰（＝例えば気温のような連続的なデータの予測。この例では「青色～オレンジ色と連続する範囲のどのあたりの値か」）の場合は、平面／マルチガウシアンが選択可能。

→目次に戻る

Python コードでの実装例

それでは、「分類問題」と「データ種別」を設定するためのコードを Python で記述してみよう。

なお本連載では、Python のバージョン 3 系を必須とする（※ Colab にインストール済みの Python バージョンが分からない場合は、import sys; print('Python', sys.version) を実行すれば、バージョンを確認できる）。Python のバージョンは、Colab にデフォルトでインストール済みのものをそのまま使えばよいはずだ（※ 2019 年 9 月 17 日の執筆時は **Python 3.6.8** だった。バージョンの切り替え方法はこちらを参考にしてほしい）。

ここでは、ライブラリ「playground-data」のメインパッケージである plygdata を pg という別名でインポートし、その中で定義されている DatasetType クラスの ClassifyTwoGaussData クラス変数の値（＝分類で、2 つのガウシアンデータ）を、定数 PROBLEM_DATA_TYPE に代入することで（リスト 2）、これを表現する。この定数を pg.generate_data() 関数に指定して実際のデータ生成を行うが、このコードは後述のリスト 3 で説明する。※なお、データ生成方法は本連載独自のコードであるため、コード内容が理解できるよう意味を説明しているが、内容を覚える必要はない。

```
# playground-data ライブラリの plygdata パッケージを「pg」という別名でインポート
import plygdata as pg

# 問題種別で「分類（Classification）」を選択し、
# データ種別で「2 つのガウシアンデータ（TwoGaussData）」を選択する場合の、
# 設定値を定数として定義
PROBLEM_DATA_TYPE = pg.DatasetType.ClassifyTwoGaussData

# ※実際のデータ生成は、後述の「pg.generate_data() 関数の呼び出し」で行う
```

リスト 2　問題種別とデータ種別の選択

（3）前処理

それでは、実際のデータ生成を行おう。通常の機械学習では、収集しておいたデータから、使うデータを選別し、さらに機械学習モデルへの入力データとして使えるように欠損値を埋めたりなどして整え、最終的に、

- **学習用のデータ**（＝**トレーニングデータ**。以下、「訓練用」「訓練データ」と表記）
- **評価用のデータ**（＝**精度検証データ**。以下、「精度検証用」「精度検証データ」と表記）

に分割する。今回は独自にデータ生成を行うので、通常とは違い、これらの作業はひとまとめに行えるようになっている。

Playground による図解

実際に Playground 上で行っているのが、図 3-1 だ。

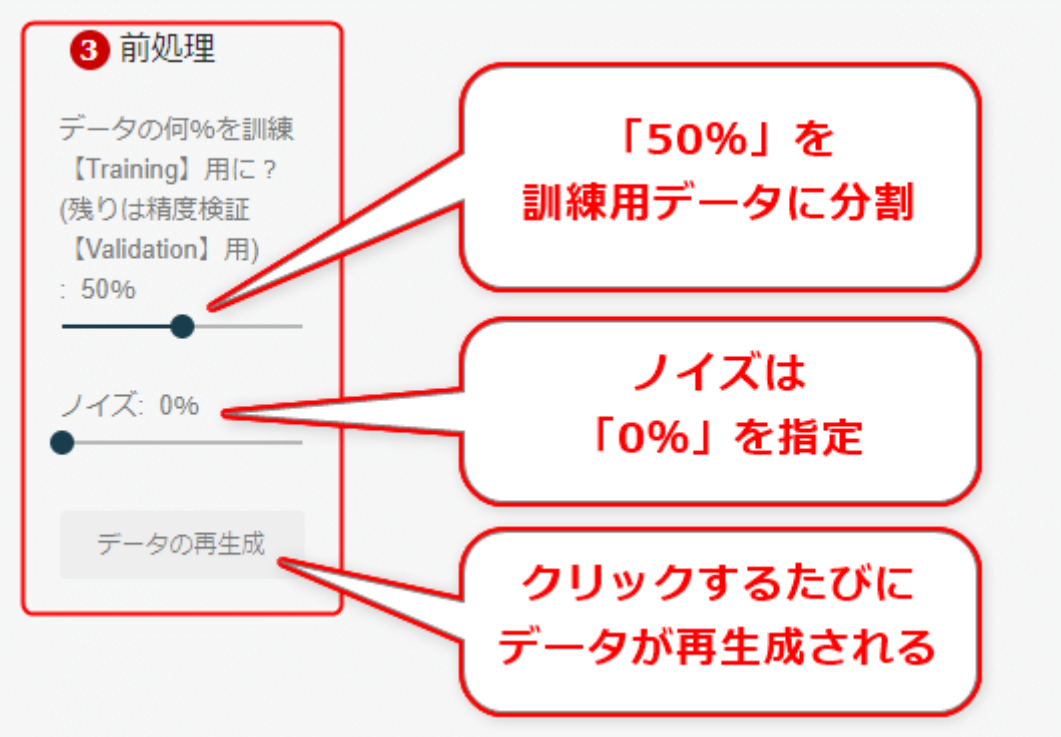

図 3-1　前処理として訓練用／精度検証用へのデータ分割やノイズを設定

図 3-1 のように、Playground の左側の（**3**）で［データの何%を訓練【Training】用に?］は「**50%**」を指定し、［ノイズ］は「**0%**」を指定する。

→目次に戻る

訓練用／精度検証用のデータ分割について

説明するまでもないと思うが、例えば訓練用を **80%**と指定すると、残りの **20%**が精度検証用になる。当然ながら、訓練データが多いほど、精度が高くなる。図 3-2 は、訓練データを最大の **90%**にした場合（左）と最小の **10%**にした場合（右）に、どちらも 100 回（学習単位は「**エポック**」と呼ばれる）学習した結果（背景）の比較である（※なお、この比較例では、差異が分かりやすくなるよう、データ種別として、より複雑な「円（サークル）」を選択した上で学習を実行した）。

図 3-2　訓練データが 90%の場合（左）と 10%の場合（右）の学習結果（背景）の比較

左の **90%**の背景が色濃くきれいな円になっており、高い精度のモデルが生成できたことが分かる。

だからといって訓練データを何が何でも多くすればよいわけでもない。精度検証データが少なすぎると、「十分な評価ができない」という問題も生じる。適切な割合は、「機械学習&ディープラーニング入門（概要編）Lesson 3」を参考にしながら、自分で考えて決める必要がある。今回は、データを半々に分けている。

ノイズについて

前掲の図 3-1 にあった［ノイズ］についても説明しておこう。**ノイズ（noise）**とは「外れ値などの余計なデータ」のことで、今回の座標点データセットでは、ノイズの数値を大きくすると、点群の散らばりが広くなる。当然ながら、ノイズが少ないほど、精度は高くなる。図 3-3 は、ノイズを最小の **0%**にした場合（左）と最大の **50%**にした場合（右）の、訓練データのプロット例（点描画）と、学習結果の比較（背景）である

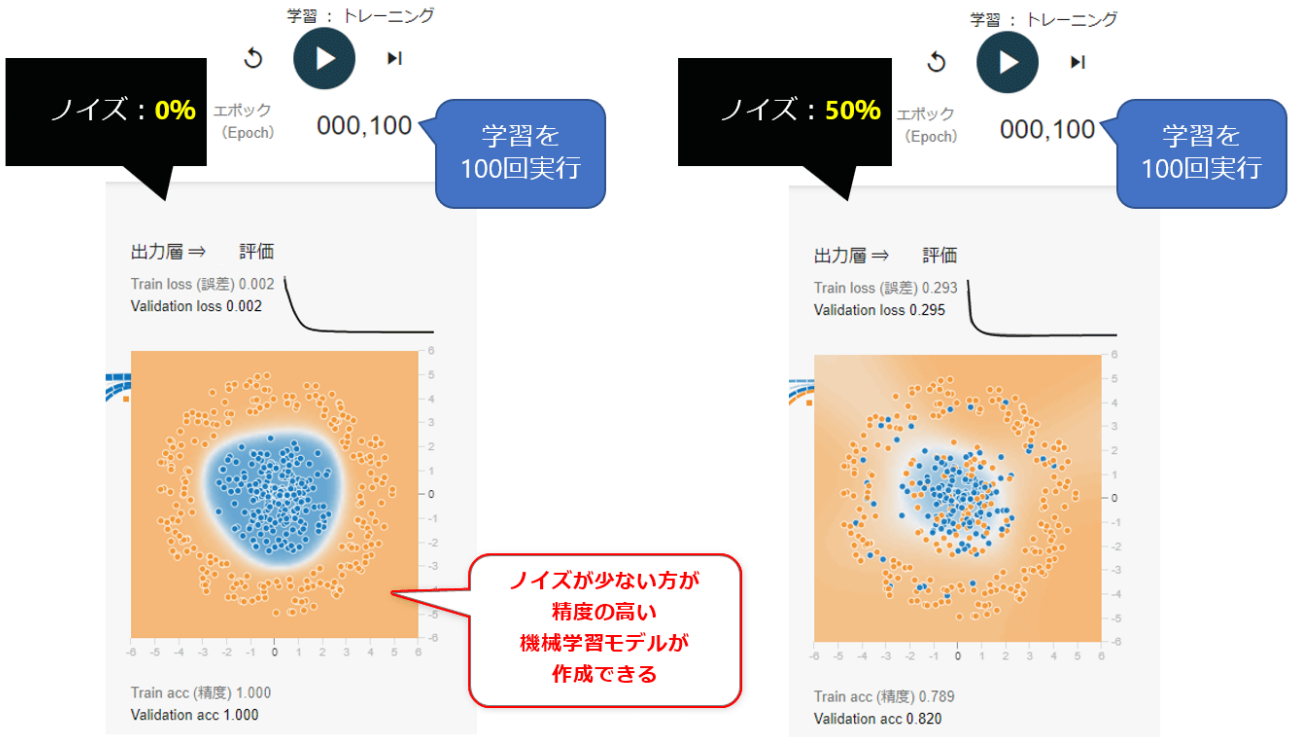

図 3-3　ノイズが 0%の場合（左）と 50%の場合（右）の、訓練データのプロット例（点描画）と、学習結果（背景）の比較

左の **0%**の方は背景が色濃くきれいな円になっており、高い精度のモデルが生成できたことが分かる。

このように基本的には、ノイズや外れ値は取り除いた方が、学習の収束が早く、モデルの**汎化性能**（＝未知のテストデータに対する学習済みモデルのパフォーマンス）も高まる。一方で、画像認識では画像にわざとノイズを乗せてより多く学習することで、より安定した学習モデルに仕上げるというテクニックがあるのも知っておいてほしい。要は、ノイズもケースバイケースで考えながら処理する必要があるというわけだ。今回は、ノイズを **0%**にしている。

Python コードでの実装例

それでは、「訓練用／精度検証用へのデータ分割」と「ノイズ」を指定してデータ生成するためのコードを Python で記述してみよう。これは次のようなコードになる。

```
# 各種設定を定数として定義
TRAINING_DATA_RATIO = 0.5  # データの何％を訓練【Training】用に？（残りは精度検証【Validation】用）：50%
DATA_NOISE = 0.0           # ノイズ：0%

# 定義済みの定数を引数に指定して、データを生成する
data_list = pg.generate_data(PROBLEM_DATA_TYPE, DATA_NOISE)

# データを「訓練用」と「精度検証用」を指定の比率で分割し、さらにそれぞれを「データ（X）」と「教師ラベル（y）」に分ける
X_train, y_train, X_valid, y_valid = pg.split_data(data_list, training_size=TRAINING_DATA_RATIO)
```

リスト 3　前処理としてのデータ分割

リスト 3 ではまず、データ分割の割合を指定する TRAINING_DATA_RATIO と、ノイズを指定する DATA_NOISE という 2 つの定数を定義している。

その次に、pg.generate_data() 関数を呼び出してデータを生成し、変数 data_list に代入している。関数の引数には、リスト 2 で定義した定数 PROBLEM_DATA_TYPE で「問題種別（**分類**）＋データ種別（**2 つのガウシアンデータ**）」を、さらに定数 DATA_NOISE で「ノイズ（**0%**）」を指定している。

生成された data_list（多次元リスト値）と、定数 TRAINING_DATA_RATIO で「ノイズ（訓練用を **50%**）」を引数に指定して、pg.split_data 関数を呼び出してデータを分割している（ちなみに、この split_data 関数は、データ分割でよく使われるライブラリ「scikit-learn」の train_test_split 関数を模して作ったものだ）。分割後のデータは以下の変数に代入されている。

- X_train：訓練データの座標点（X）。N 行 2 列のデータ
- y_train：訓練データの教師ラベル（y）。N 行 1 列のデータ
- X_valid：精度検証データの座標点（X）。N 行 2 列のデータ
- y_valid：精度検証データの教師ラベル（y）。N 行 1 列のデータ

いずれの変数も、ライブラリ「NumPy」の多次元配列値となっている（※ディープラーニングのライブラリは基本的に NumPy に対応しているのでそのまま使用できる）。具体的にどのようなデータが入っているかというと、図 3-4 がその一部の出力例である。

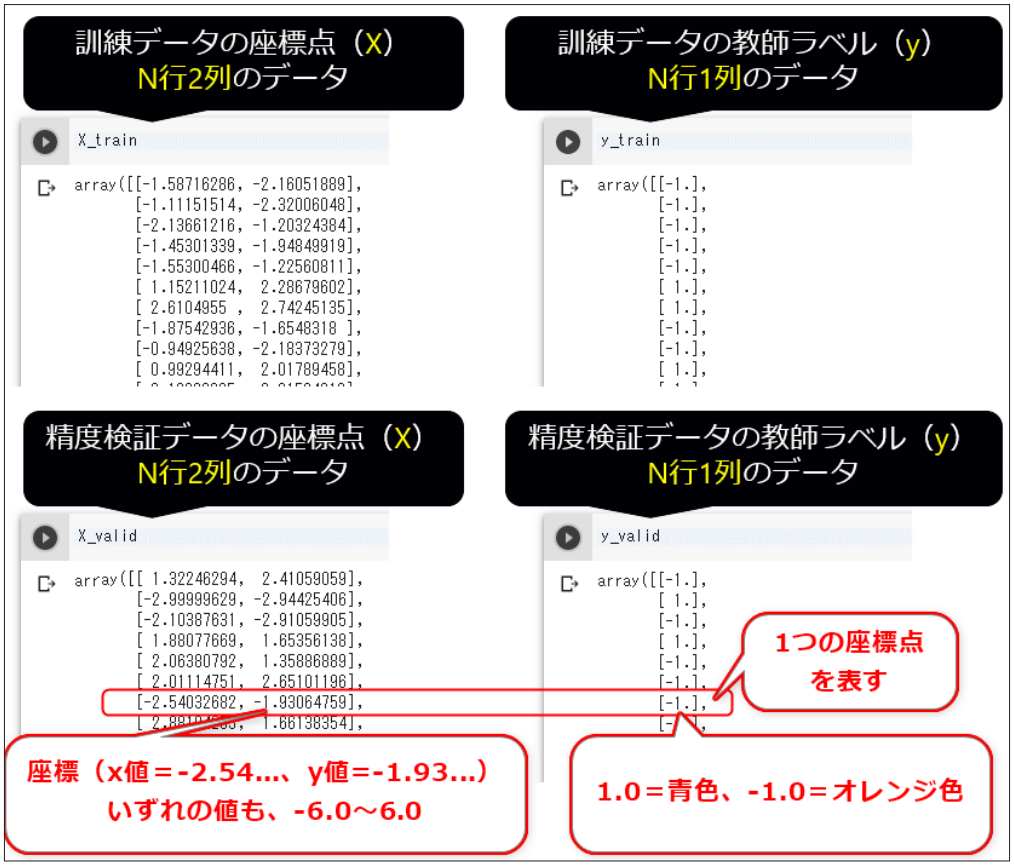

図 3-4　データ分割後の各変数の内容例

各行は、個々の座標点を表している。

座標点（X）にある 2 列の数値の意味は「座標（横の X 軸上の値、縦の Y 軸上の値）の点」であり、前述の通り、いずれも基本的に **-6.0** ～ **6.0** の数値を取る。

教師ラベル（y）にある 1 列の数値の意味は、前述の通り、**1.0** ＝青色、**-1.0** ＝オレンジ色である。

以上でデータの生成（通常はデータの取得～前処理）は完了だ。あとは上記の各データ変数を使って、ニューラルネットワークを説明していく。

今回前編ではワークフローの全 8 工程のうち、（**1**）～（**3**）のデータに関する処理内容を説明した。今回のデータを、次回中編の（**4**）から説明するニューラルネットワークで使っていくことになる。次回は、ニューラルネットワークの基本単位であるニューロンの説明から始める。前編・中編・後編の三部作は、まとめて読むことで価値があるので、ぜひ次回中編も継続して読み進めてほしい。Please follow on Twitter @DeepInsiderJP.

TensorFlow 2 + Keras（tf.keras）入門：

第 2 回　ニューラルネットワーク最速入門
—— 仕組み理解 × 初実装（中編）

ニューラルネットワーク（NN）の基礎の基礎。NN の基本単位であるニューロンはどのように機能し、Python +ライブラリでどのように実装すればよいのか。できるだけ簡潔に説明。また、活性化関数と正則化についても解説する。

一色政彦，デジタルアドバンテージ（2019 年 10 月 17 日）

前回は、ディープラーニングの大まかな流れを、下記の **8 つ**の工程で示した。

- （**1**）データ準備
- （**2**）問題種別
- （**3**）前処理
- （**4**）“ 手法 ” の選択：モデルの定義
- （**5**）“ 学習方法 ” の設計：モデルの生成
- （**6**）学習：トレーニング
- （**7**）評価
- （**8**）テスト

このうち、（**1**）～（**3**）のデータに関する処理内容は前回で説明済みである。そのデータを使って、今回は、（**4**）のニューラルネットワークのモデルの定義方法について説明する。それではさっそく説明に入ろう。※脚注や図、コードリストの番号は前回からの続き番号としている（前編・中編・後編は、切り離さず、ひとまとまりの記事として読んでほしいため連続性を持たせている）。

Google Colabで実行する　　GitHubでソースコードを見る

（4）“ 手法 ” の選択とモデルの定義：ニューロン

ニューラルネットワーク（＝神経ネットワーク）とは、その名前の通り、脳の神経ネットワーク構造を模した機械学習の手法である。そのニューラルネットワークを理解するために、その基本単位である**ニューロン**（**neuron**：神経細胞）から話を始めたい。なおニューロンは、細胞体もしくは単位を意味する**ユニット**（**unit**）や、ネットワーク構造における 1 つの結節点を意味する**ノード**（**node**）などの名称でも呼ばれるので知っておいてほしい。

ニューロンの基本機能は、関数と同じように、複数の入力を受け取り、それらを使って何らかの計算をし、その結果を **1 つ**の出力として生成することだ。例として、**2 つ**の入力を受け付けるニューロンを考えてみよう。

→目次に戻る

Playground による図解

そのようなニューロンを、実際に Playground 上で作成したのが、図 4-1 だ（※右端にある大きな表示は、実際にはニューラルネットワークにおける「出力層」にある 1 つのニューロンだが、ここの説明においてはプレビュー確認用のグラフ領域と見なして、「出力層」であることは無視してほしい。また左端のX_1／X_2も、ニューロンではなく単なる入力だと見なしてほしい）。

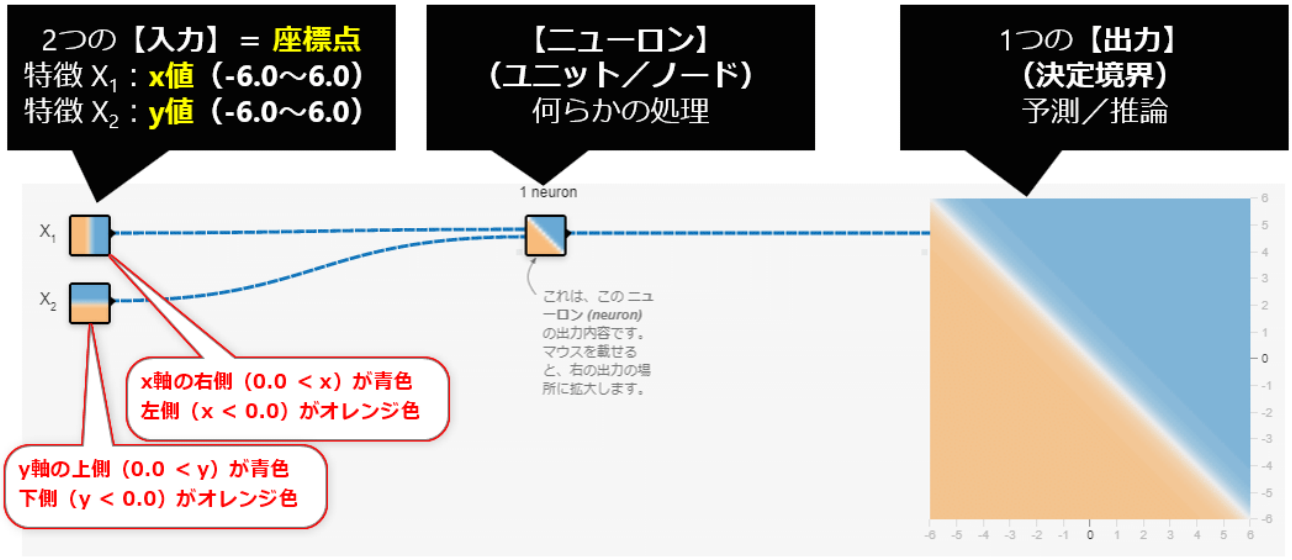

図 4-1　ニューロンの構造

なお座標が、(x, y) ではなく、(X_1, X_2) と表記されていることに注意してほしい。またX_1／X_2のような個々の入力値や個々のノードが表現するものは「**特徴**（**feature**）」（もしくは「**特徴量**」）と呼ばれ、例えばX_1の場合、図 4-1 の左上にあるX_1の四角を見ると、このデータは「右側（0.0 < x）が青色、左側（x < 0.0）がオレンジ色で、中央付近（x ＝ 0.0）は白色」、言い換えると「決定境界の線が中央にまっすぐに縦に引かれている」という特性を持っている。

ちなみに、図 4-1 と同じ図を作成するには、こちらのリンク先の Playground を開き、その中央の（**4**）で 3 つの線の上をそれぞれクリックし、それにより表示されるポップアップ上で［重み］（後述）に「**1.0**」を入力する。入力方法のヒントを図 4-2 にまとめた。

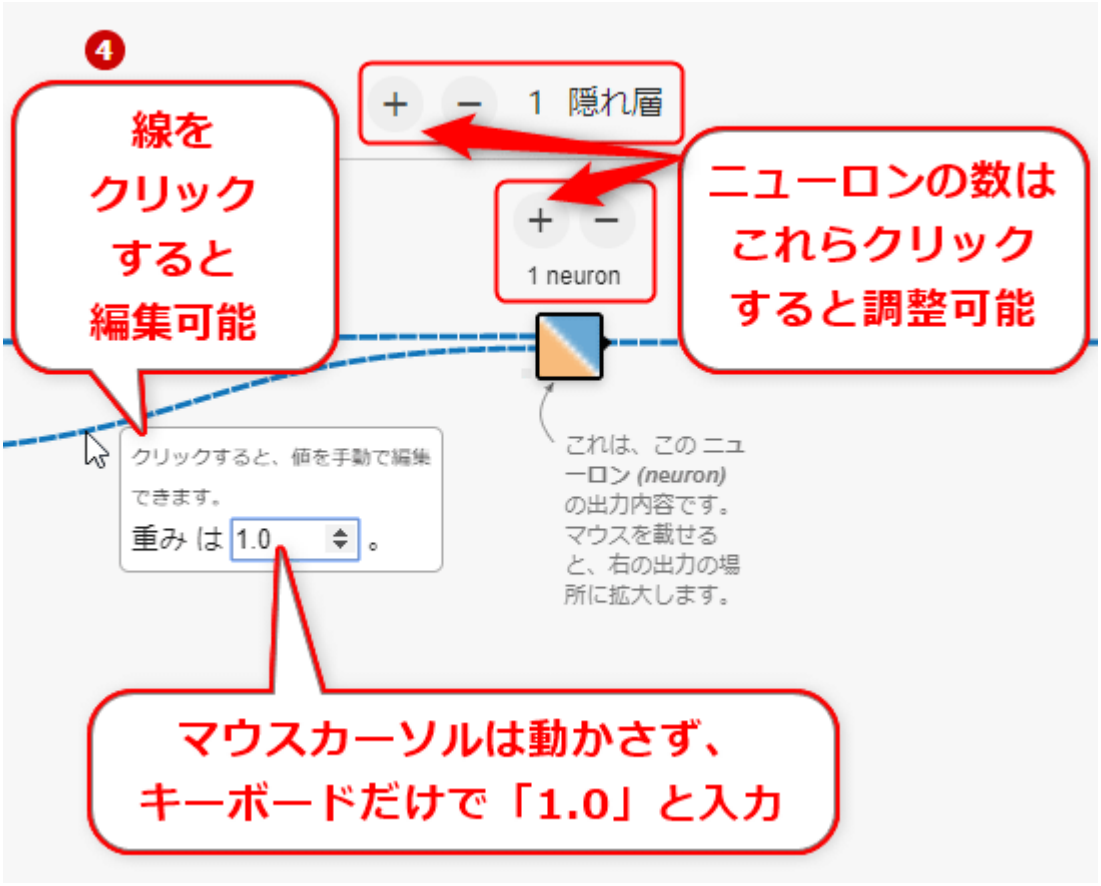

図 4-2　Playground のニューラルネットワークの編集方法（ノードと重み）

※操作性が若干悪いのだが、マウスカーソルを動かすと、ポップアップが消えて数値を入力できないので、注意してほしい。

入力

先ほど「重み」と出てきたが、**重み**（**weight**）とは入力を増幅～減衰させるためのものである。先ほどは、座標 (X_1, X_2) （変数）のそれぞれに対して、**1.0** と固定値を指定したので、何も増減しないことになるが、本来はこの重みが学習によって自動的に計算されて決まるのである。重みはそれぞれ w_1、w_2 というパラメーターで置くとしよう。よって、ニューロンへの入力は次の式で表現できる。

$$\text{ニューロンへの入力（未完成）} = (w_1 \times X_1) + (w_2 \times X_2)$$

（未完成）と書いたが、実はこれで終わりではない。中学校で一次関数を学んだとき「切片（せっぺん）」という概念があったのを覚えているだろうか？　例えば $y = ax + b$ という一次関数では、a が傾きで、b が切片である。b の値を **0**、**1**、**2**……と増やしていくと、一次関数のグラフに描かれた直線が、y 軸の上方向に **1** ずつズレ上がっていくのがイメージできるだろうか？　つまり切片があると、関数の位置をズラせるようになるのである。もし一次関数に切片がないと、常にグラフの原点 (0, 0) を通るのみになって融通が利かない。

これと同じことが、上記のニューロンの式にもいえる。ニューロンで切片に該当するのが**バイアス**（**bias**）である。バイアスの数値を増減させたときのイメージを、図 4-3 に示す。背景がズレていく様子が見て取れるだろう。

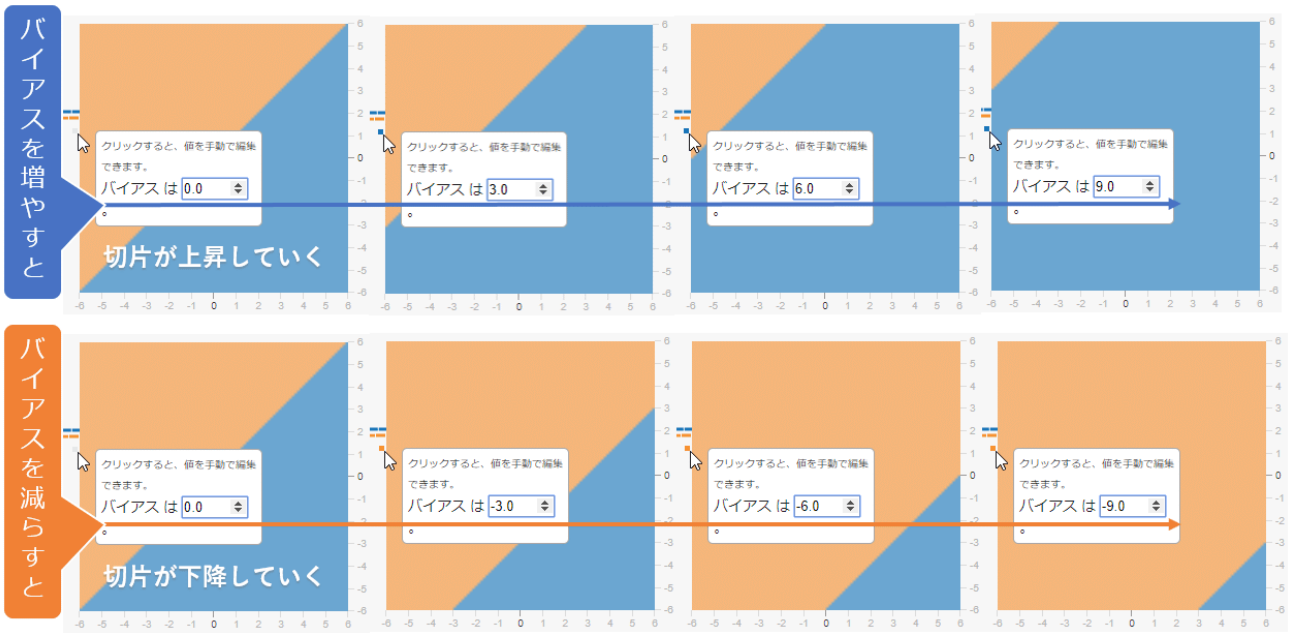

図 4-3　バイアスの数値を増減させたときのイメージ

ちなみに、Playground でバイアスを編集するには、ニューロンへの接続線の下にある小さな四角をクリックし、それにより表示されるポップアップ上で［バイアス］の数値を好きに入力するとよい。入力方法のヒントを図 4-4 にまとめた。数値が変わるたびに右側の大きなグラフがズレていくのが体験できるだろう。

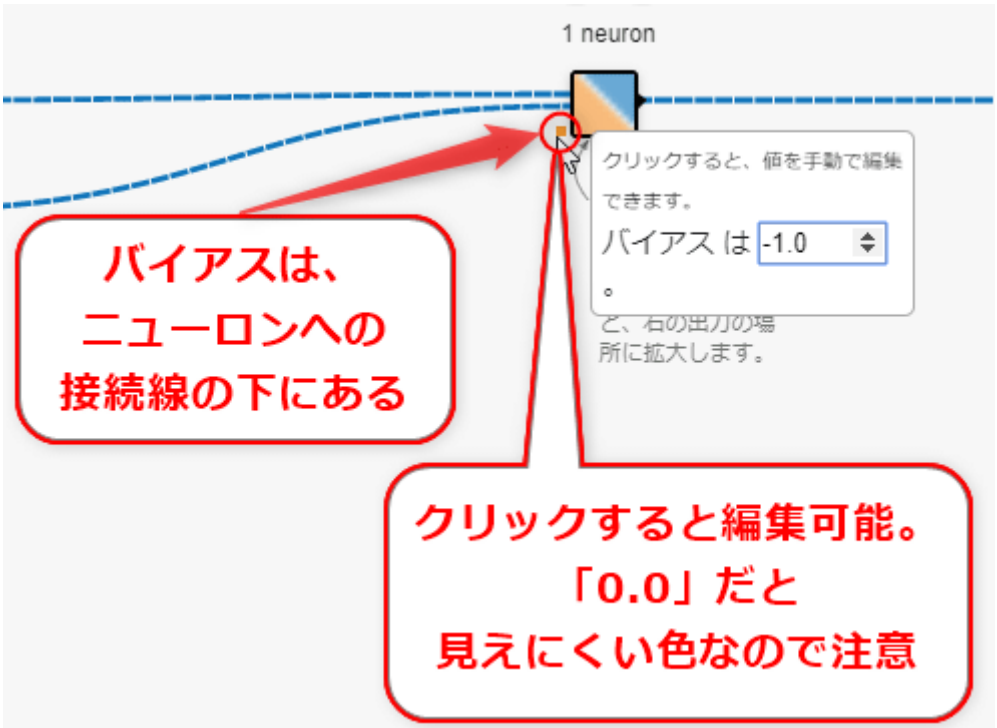

図 4-4　Playground のニューラルネットワークの編集方法（バイアス）

バイアスを b というパラメーターで表現して加えると、ニューロンへの入力式は次のようになる。

$$\text{ニューロンへの入力（完成）} = (w_1 \times X_1) + (w_2 \times X_2) + b$$

→目次に戻る

何らかの計算結果の出力

前述したように、複数の入力を受け取ったニューロンは、それらの入力を使って何らかの計算をし、その結果を**1 つ**の出力として生成する。この「何らかの計算」は、**活性化関数**（**Activation function**）と呼ばれている。活性化関数とは、無制限の入力を、予測可能な範囲に変換するためのものである。

一般的な活性化関数の一つに**シグモイド関数**（**Sigmoid function**）がある。この関数は、どんな入力も、つまり−∞〜+∞の数値を、0.0 ～ 1.0 の範囲の数値に変換する。具体的には、図 4-5 のように変換する。なお、グラフの見方は、x 軸が入力する数値で、y 軸が変換後に出力する数値である。これを見ると、大きすぎる負の数は **0** に収束し、大きすぎる正の数は **1** に収束することが分かる。

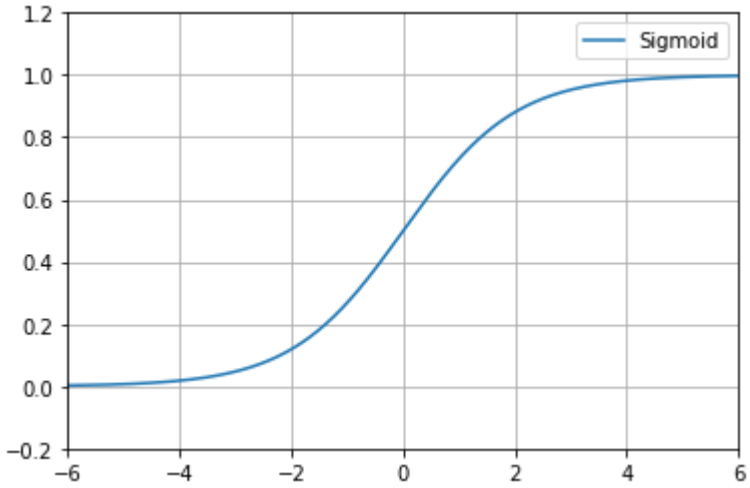

図 4-5　シグモイド関数のグラフ

ちなみに、Playground における最終的な出力は、**-1.0**（＝オレンジ色）～ **1.0**（＝青色）である。つまり、ニューラルネットワークの最後の最後にある活性化関数で、− 1.0 ～ 1.0 の範囲の数値に変換しているのだ。この範囲に変換するのにちょうどよい活性化関数が、**tanh 関数**（**Hyperbolic tangent function**）である。この関数は、図 4-6 のように変換する特性を持つ。

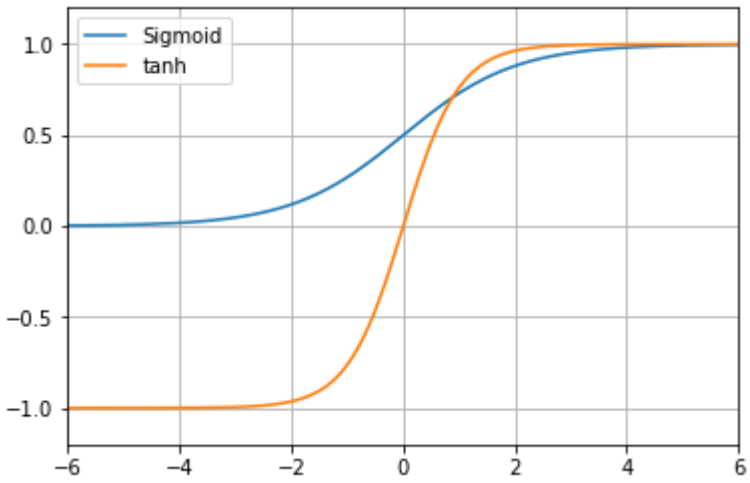

図 4-6　tanh 関数のグラフ（比較：シグモイド関数）

→目次に戻る

数値の範囲が最終的な出力の範囲と一致している方が、ニューロンの動きを把握しやすいと思うので、本稿ではシグモイド関数ではなく、tanh 関数を使うことにする。

他にも活性化関数はあるが、それらについては後述のニューラルネットワークの部分で簡単に説明する。

さて、それではニューロンからの出力式を書き出してみよう。活性化関数を a() と置くと、次のようになる。

$$\text{ニューロンからの出力} = a((w_1 \times X_1) + (w_2 \times X_2) + b)$$

「ニューロンからの出力＝活性化関数（ニューロンへの入力）」という式になっていることに気付けば難しくないだろう。参考までに計算してみよう。w1 ＝ 0.6、w2 ＝－ 0.2、b ＝ 0.8 だと仮定して、座標（**1.0**、**2.0**）をニューロンに入力すると、

$$\text{ニューロンからの出力} = a((0.6 \times 1.0) + (-0.2 \times 2.0) + 0.8)$$
$$= a(1.0) = \mathbf{0.761594...}$$

と計算され、出力値は**約 0.76** となる（※活性化関数自体の計算方法は割愛したので、上掲グラフにおける tanh 関数の曲線を見て、横軸 **1** に対応する縦軸の値を目視確認してみてほしい）。

Python コードでの実装例

それでは、「ニューロン」の入力と出力を行うコードを Python で記述してみよう。

TensorFlow 2.0 のインストール

本稿では、TensorFlow のバージョン 2 系を必須とする（※ Colab にインストール済みの TensorFlow バージョンが分からない場合は、import tensorflow as tf; print('TensorFlow', tf.__version__) を実行すれば、バージョンを確認できる）。

本稿が利用を前提とする Colab（前編参照）にデフォルトでインストール済みのバージョンは、（2019 年 10 月 15 日の執筆時点で）**TensorFlow 1.15.0-rc3** だった。このように、バージョンが 2.x ではない場合は、ライブラリ「TensorFlow」を最新版の **2.x** にアップグレードして使う必要がある。

そのためには、リスト 4-0 に示すいずれかのコードを実行して、アップグレード（もしくはインストール）する。実行後に、［RESTART RUNTIME］ボタンが表示されたら、クリックしてランタイムを再起動してほしい。ちなみに、そう遠くない将来、Colab ではこのインストール手順を実行しなくても、最新の TensorFlow 2.x が利用できるようになるだろう。

```
# 最新バージョンにアップグレードする場合
!pip install --upgrade tensorflow

# バージョンを明示してアップグレードする場合
#!pip install --upgrade tensorflow===2.0.0

# 最新バージョンをインストールする場合
#!pip install tensorflow

# バージョンを明示してインストールする場合
#!pip install tensorflow===2.0.0
```

リスト 4-0　［オプション］ライブラリ「TensorFlow」を最新バージョンにアップグレード

tensorflow パッケージのインポート

TensorFlow の利用環境が整ったとして話を進めよう。ここでは、ライブラリ「TensorFlow」のメインパッケージである tensorflow を tf という別名でインポートし、その中で定義されている tf.keras パッケージ（＝TensorFlow 同梱の Keras）を使って、モデル（＝その Model クラス *1 のオブジェクト）を作成する。

*1　tf.keras モジュール階層（＝名前空間）の Model クラスとしてアクセスできるが、実際のクラス実装は tensorflow.python.keras.engine.training モジュール階層にあり、別名となっている。

Keras におけるモデルの書き方

Keras によるモデルの作成方法／書き方は幾つかある。書き方は大きく分けて、

- **Sequential（積層型）モデル：**コンパクトで簡単な書き方
- **Functional（関数型）API：**複雑なモデルも定義できる柔軟な書き方
- **Subclassing（サブクラス型）モデル：**難易度は少し上がるが、フルカスタマイズが可能

の 3 種類がある。筆者のお勧めは Functional API か Subclassing モデル（**Subclassing API**）だが、今回は基本である Sequential モデル（**Sequential API**）を使って書いてみよう（※説明をできるだけシンプルにするために、Functional API や Subclassing モデルの書き方の説明は割愛する）。

さらに Sequential モデルにおいても、書き方は大きく分けて、

- **Sequential クラスのコンストラクター利用：**最も基本的でシンプルな書き方
- **Sequential オブジェクトの add メソッドで追加：**シンプルだが、より柔軟性のある書き方

の 2 種類がある。どちらを使うかは好みだが、取りあえずは基本である「コンストラクター（厳密には __init__ 関数）利用」の方を使って書いてみる（※「add メソッドで追加」の書き方についても割愛する）。

ここでは、**2 つ**の入力を受け付けて、それを使った計算結果を **1 つ**の出力として生成するニューロンを、モデル化（＝モデル設計）する。

モデルの定義

モデル設計のコードを先に示してから、コード内容を説明する。おおよその内容は、コード内のコメントから理解できるだろう。

```
# ライブラリ「TensorFlow」の tensorflow パッケージを「tf」という別名でインポート
import tensorflow as tf
# ライブラリ「NumPy」の numpy パッケージを「np」という別名でインポート
import numpy as np

# 定数（モデル定義時に必要となる数値）
INPUT_FEATURES = 2  # 入力（特徴）の数：2
LAYER1_NEURONS = 1  # ニューロンの数：1

# パラメーター（ニューロンへの入力で必要となるもの）
weight_array = np.array([[ 0.6 ],
                         [-0.2 ]])      # 重み
bias_array   = np.array([  0.8 ])       # バイアス

# 積層型のモデルの定義
model = tf.keras.models.Sequential([
  tf.keras.layers.Dense(
    input_shape=(INPUT_FEATURES,),      # 入力の形状
    units=LAYER1_NEURONS,               # ユニットの数
    weights=[weight_array, bias_array], # 重みとバイアスの初期値
    activation='tanh')                  # 活性化関数
])

# このモデルに、データを入力して、出力を得る（＝予測：predict する）
X_data = np.array([[1.0, 2.0]])         # 入力する座標データ（1.0、2.0）
```

```
print(model.predict(X_data))          # 出力を得る
# [[0.7615942]] ……などと表示される
```

リスト 4-1　ニューロンのモデル設計

上から順に説明すると、ディープラーニング用のライブラリ「TensorFlow」と、多次元配列&数値演算用のライブラリ「Numpy」のインポート後に、

- **入力の数**：INPUT_FEATURES = 2
- **ニューロンの数**：LAYER1_NEURONS = 1

を定数として定義している（※「出力の数」は **1** だが、ここでは使わないので定義していない）。

それらの定数を使ってモデル（リスト 4-1 では model オブジェクト）を作成するわけだが、前述の通り（tf.keras.models モジュール階層の）Sequential クラスのコンストラクターを利用している。このコンストラクターは、

```
__init__(
    layers=None
)
```

と定義されており（※不要な引数は説明を割愛）、引数の意味は以下のとおりだ。

- 引数の layers：レイヤー（層）をリスト値で指定する（※レイヤーは複数のニューロンで構成する層のこと、**1 層**、**2 層**、**3 層**と増やせる。後述のニューラルネットワークで詳しく説明する）

今回はニューラルネットワークではなく、**単一**のニューロンを実装したいので、レイヤーも**1 つ**のみとなる。Keras においてはさまざまな種類のレイヤーが用意されているが、今回は最も基本的な、

- **Dense レイヤー**：ニューラルネットワークで通常の全結合レイヤー

を使用する（※「全結合」（もしくは密結合）は文字通り「全て結合」という意味だが、今回はそれが「最も標準的なレイヤー間の接続方法」ということを押さえておけばよい）。Dense レイヤーのインスタンスをリスト化して、先ほどの引数 layers に指定する。前掲のリスト 4-1 を見ると、（tf.keras.layers モジュール階層の）Dense クラスのコンストラクターを利用して、インスタンスを生成していることが分かる。このコンストラクターは、

```
__init__(
    units,
```

```
    activation=None,
    **kwargs
)
```

と定義されており（※不要な引数は説明を割愛）、各引数の意味は以下のとおりである。

- 第 1 引数の units：ユニット（＝ニューロン）の数を指定する。今回は、先ほどの定数 LAYER1_NEURONS を指定すればよい
- 第 2 引数の activation：活性化関数を指定する。今回は tanh 関数を使用するので、文字列で「**tanh**」と指定する。Keras では代表的な活性化関数は事前に定義されており（詳細後述）、文字列で指定するだけで済む。用意されていない場合は、独自の計算式をここに指定することもできる
- 第 3 引数の kwargs：複数のキーワードとその値を指定できる。使えるキーワードは幾つかあるが（※不要なキーワードは説明を割愛）、その一つが input_shape で、もう一つが weights である
  - キーワード「input_shape」：入力の形状をタプル値で指定する。今回は、先ほどの定数 INPUT_FEATURES を指定すればよい。ただし、タプル型にするため (INPUT_FEATURES,) とタプル要素の最後に「,」を入れることで、タプル値と明示する必要が文法上ある
  - キーワード「weights」：「重み」の 2 次元配列値と「バイアス」の 1 次元配列値を、リスト値にまとめて指定する。今回は、[weight_array, bias_array] というリスト値を指定している。通常のニューラルネットワークでは、重みとバイアスは自動的に決まるもので、今回のように固定的な初期値を指定する必要はない（リスト 4-1 のような指定は説明のためのもので特殊な書き方である）

以上で、ニューロンのモデル定義は完了だ。まずはここまでをしっかりと押さえてほしい。後述のニューラルネットワークのモデル定義はこれを拡張していくだけなので、今回の説明は上記のコードが理解できることが鍵だ。

モデルへの入力と出力（フォワードプロパゲーション）

さて、それではこのモデルに入力を与えてみよう。今回は、先ほどの手計算の例と同じように座標（**1.0**、**2.0**）の **1 つ**だけをデータとする。前掲のリスト 4-1 では、1 行 2 列（＝行がデータの数、列が入力の数となっている）の 2 次元配列値を作り、X_data という変数に代入している。その X_data を引数に指定して、model オブジェクトの predict() メソッドを呼び出している。

predict() メソッドは、上記で行った「ニューロンからの出力」の計算を自動的に実行し、戻り値としてモデルからの出力を返す。リスト 4-1 では、戻り値は print() メソッドで出力しているので、Colab 上のコードセルの下に [[0.7615942]] と出力されるはずだ（※なお、[[……]] と数値が多重カッコで囲まれているのは、この出力が NumPy の多次元配列値だからだ）。この数値は、前述の手計算で示した出力値の**約 0.76** と全く同じであることを確認してほしい。

上記のコード例のように、モデルにデータを入力し、内部で何らかの変換をしてから、新たな出力値を得る流れ（プロセス）は、**フォワードプロパゲーション（Forward Propagation）**と呼ばれる。

以上でニューロンの解説は完了だ。続いて、ニューラルネットワークを見てみよう。

（4）“ 手法 ” の選択とモデルの定義：ニューラルネットワーク

単純なニューラルネットワークは、先ほどのニューロン（基本単位）をたくさんつなげてネットワーク化しただけのものである。先ほどのニューロンをキッチリと押さえていれば難しくない。

では、どのようなネットワーク構造にすればよいのか？　これについて見ていこう。

Playground による図解

ニューラルネットワークの基本形

ニューラルネットワークの基本形は、**入力層／隠れ層／出力層（Input layer ／ Hidden layer ／ Output layer）**という 3 層構造を形成することである（※隠れ層は、**中間層：Intermediate layer** とも呼ばれる）。図 4-7 の表示に示すように、Playground 上での表示もその基本形となっていることを確認してほしい。

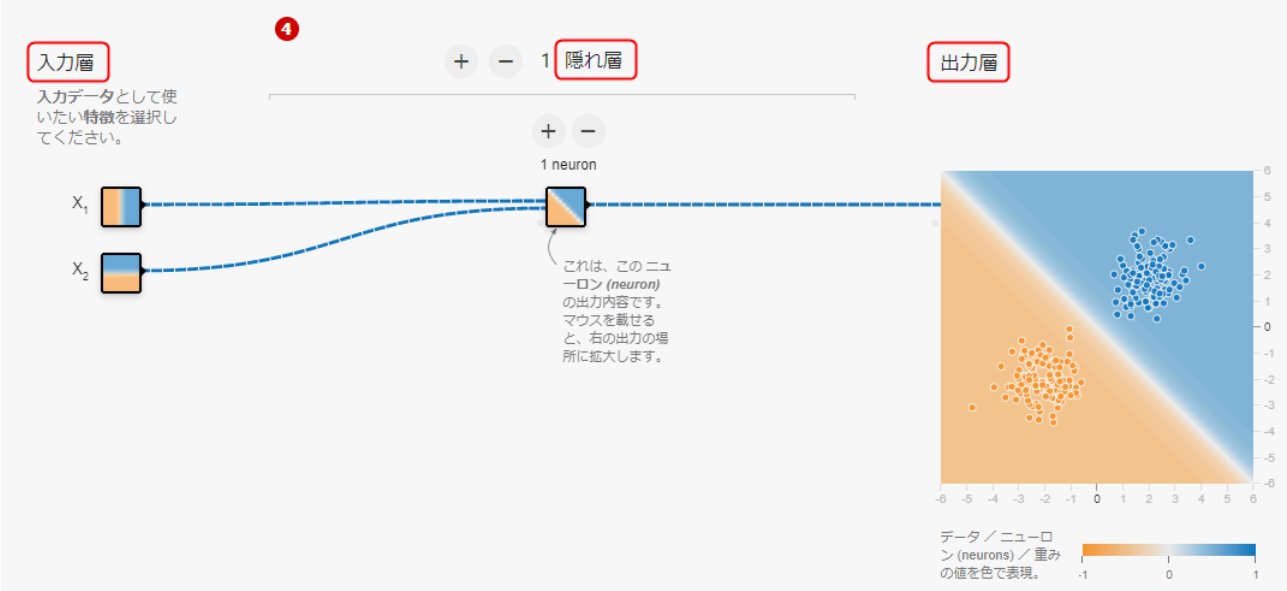

図 4-7　ニューラルネットワークの基本形「入力層／隠れ層／出力層」

※この画像は、未説明で不必要な表示を消すように加工している。

図 4-7 は、次のようなニューロンで構成されている

- **入力層：**X_1 と X_2 という特徴がある
- **隠れ層：**1 つのニューロンがある
- **出力層：**1 つのニューロンがある

→目次に戻る

ニューラルネットワークの場合、入力層にある特徴は、データの入力箇所であり、厳密にはニューロンではない。つまり、入力層のX_1やX_2は次の隠れ層にある各ニューロンへの入力としてそのまま使われるだけなのだ（＝活性化関数による変換は入力層では行われない）。

よって、先ほど「3 層構造」と書いたが、ネットワークの見た目は 3 層でも、実質的には 2 層構造である。このように層の数え方は見方や考え方によって変わってしまうので注意してほしい。

ちなみに、前掲のリスト 4-1 で input_shape=(INPUT_FEATURES,) という引数が書かれていたが、これがまさに「入力層（＝入力の形状）」を意味する。

説明がしつこくなるが、図 4-8 では、隠れ層にあるニューロンと、出力層にあるニューロンの入力と出力の流れを矢印で表現してみた。なお、出力層の出力は、次のニューロンへの入力ではなく、結果表示に使われるだけとなり、右側に大きく決定境界の図として描画されている。

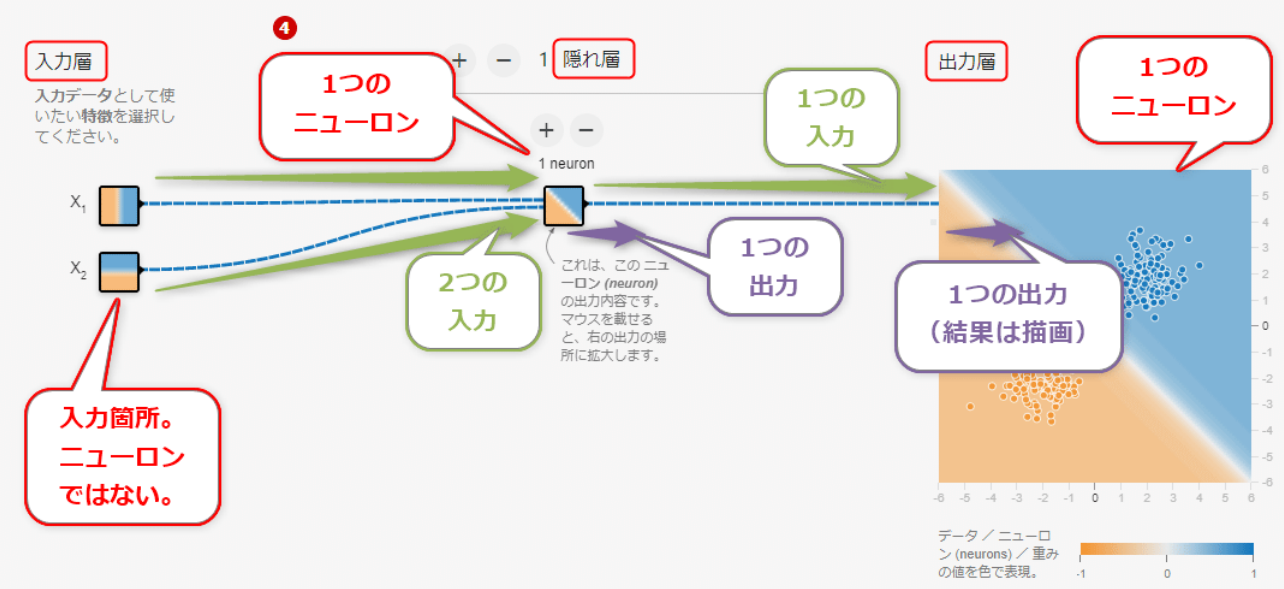

図 4-8　隠れ層と出力層における入力と出力の流れ

ニューロンの数を増やす

さて、先ほどは隠れ層も出力層も **1 つ**のニューロンしかなかった。ニューラルネットワークでは、それぞれニューロンの数を好きなだけ増やせる。Playground の提供機能では、出力層は **1 つ**のニューロン固定だが、隠れ層では **8 個まで**ニューロンを増やせる（図 4-9）。ちなみに Playground 上で実際に増やしてみる場合は、こちらを開いて［<数値> neurons］という表記の上にある［+］／［−］ボタンをクリックしてほしい。

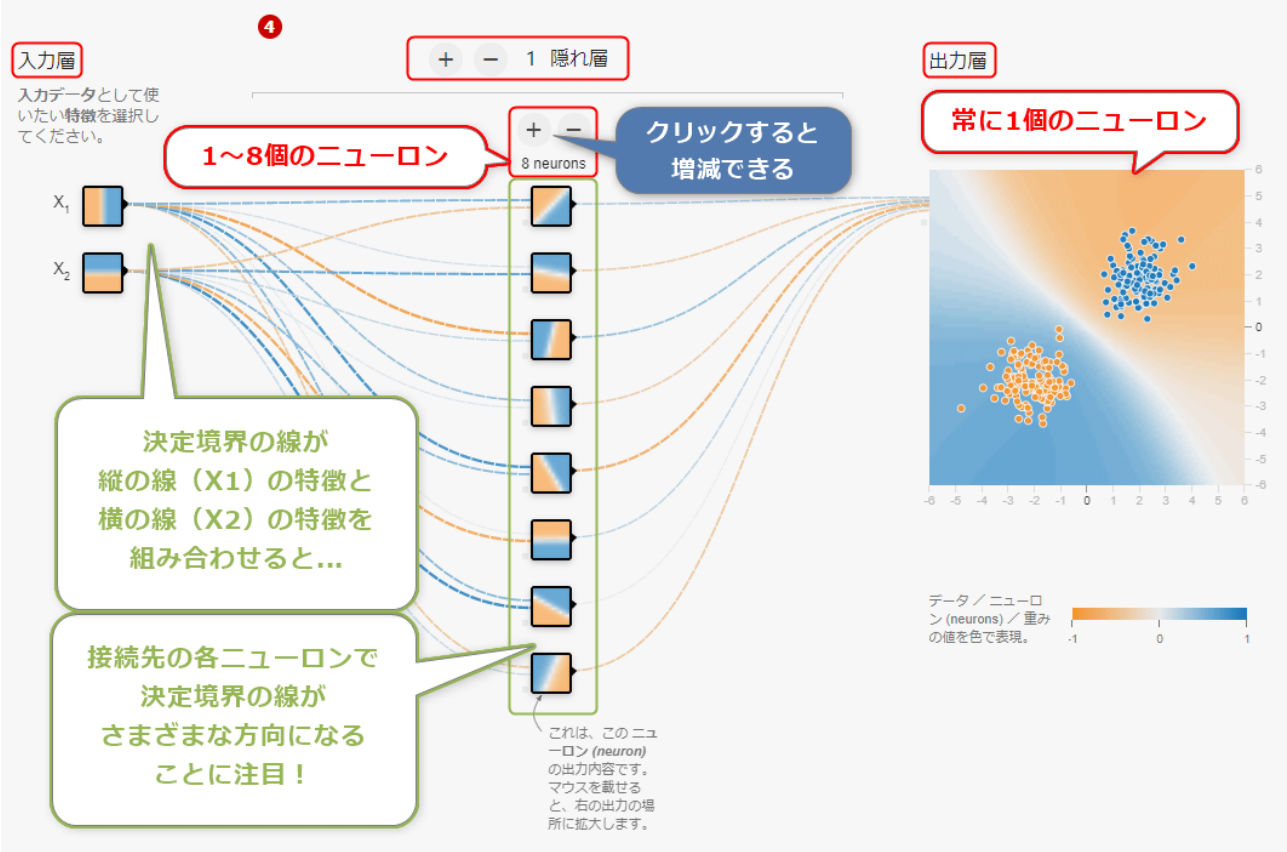

図 4-9　隠れ層のニューロンの数を 8 個に増やしたところ

図 4-9 の入力層にある **2 つ**のニューロンと、隠れ層にある **8 つ**のニューロンの、四角内の描画内容に注目してほしい。

入力層のX_1には縦線が、X_2には横線が描かれている。それに続く隠れ層の各ニューロンには、さまざまな方向の線が描かれている。これはつまり、縦線と横線をそれぞれ重み（前述の説明と同様に、w_1 や w_2 とする）を掛けて足し合わせると（＝ $(w_1 \times X_1) + (w_2 \times X_2)$）、斜めなどのさまざまな方向の線が生まれるということだ。

これによって、単一のニューロンよりも少し複雑な問題が解決できるようになる。前述の通り、隠れ層が **1 つ**のニューロンだけでも斜めの線が生成できることが確認できた。よって、前回選択したデータセット「ガウシアン（Gaussian）」の分類問題は、これで解決できる。では、より複雑なデータセット「円（Circle）」の場合はどうだろうか？（図 4-10）

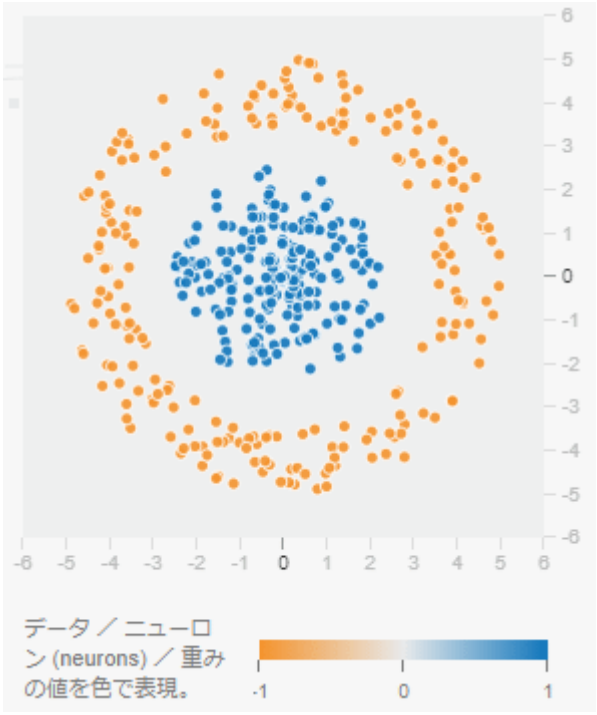

図 4-10　データセット「円（Circle）」

隠れ層にある **8 つ**のニューロンの先にあるのは、出力層にある **1 つ**のニューロンである。この出力層のニューロンもまた、隠れ層にある **8 つ**のニューロンが持つさまざまな方向の線を入力として、それぞれに重みを掛けたうえで足し合わせた後の結果が出力／描画されることになる。方向の線が多ければ多いほど、多種多様な図形が描画できるようになる。例えば隠れ層のニューロンが、

- **1 個**だと「線」になる
- **2 個**だと「角のある領域」になる
- **3 個**だと「三角」が作れるようになる
- **4 個**だと「四角」が作れるようになる
- **5 ～ 7 個**だと「五角形」～「七角形」
- **8 個**が一番円らしい円を描画できる

はずだ。となれば、隠れ層に **8 つ**のニューロンを用意すれば、もっとも確実に問題を解決できそうである。

しかし、ニューロン数を増やせば増やすほど計算に時間がかかるようになる。つまり処理スピードが遅くなる。よって、ニューロン数はできるだけ少なくした上で問題解決できるほど、通常はより良い策だろう。実際に、「円」の分類は、**3 つ**のニューロンで三角形が描画できれば解決できる。図 4-11 は、隠れ層のニューロン数が **1 個／2 個／3 個／8 個**の場合に、「円」の分類問題がどのように解決されるかを示したものである。

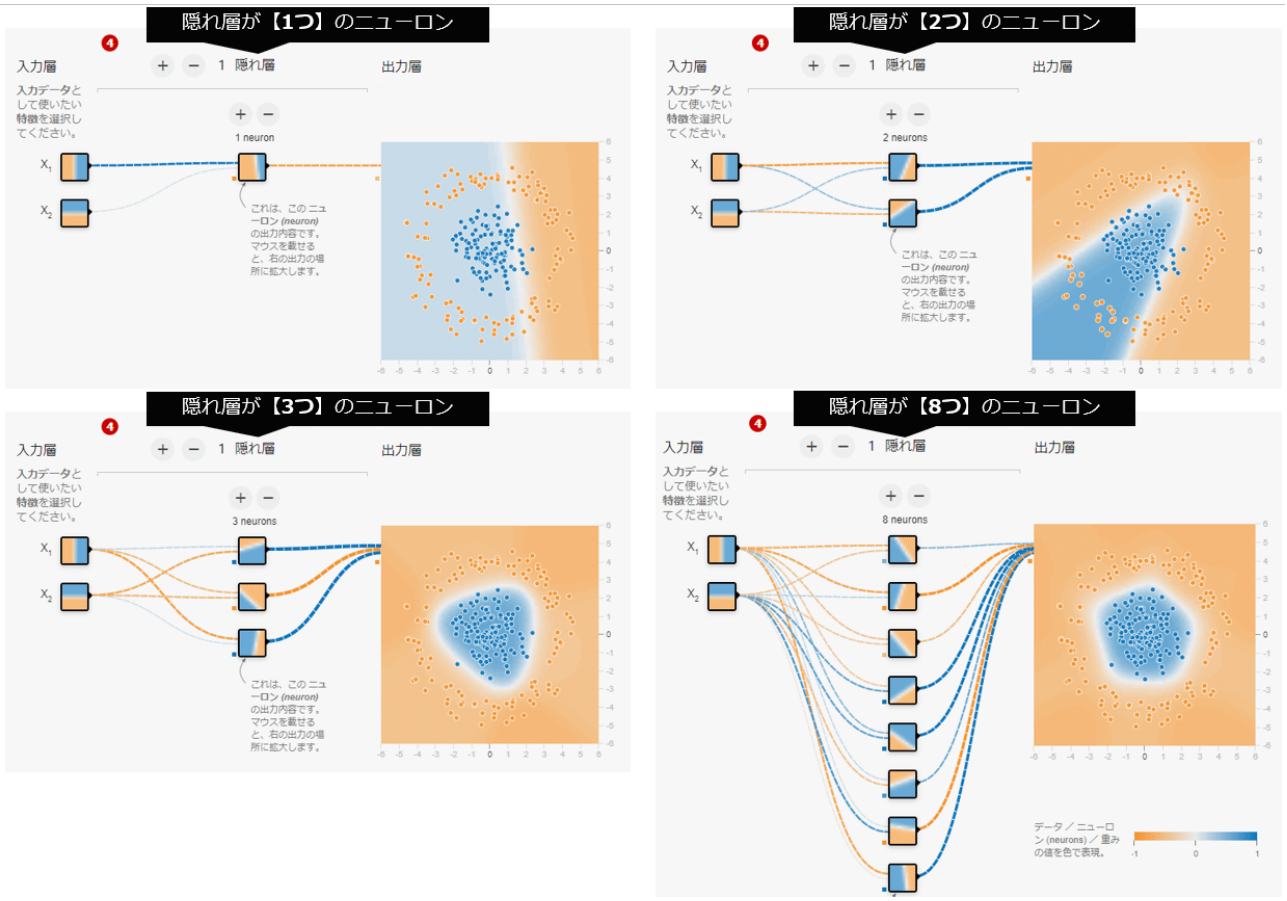

図 4-11　「円（Circle）」の分類問題解決例（隠れ層のニューロン数が 1 個／ 2 個／ 3 個／ 8 個の場合）
※いずれも 100 回（学習単位は「エポック」と呼ばれる）学習した結果（背景）の比較である。

当然ながら、左上の「**1 つ**（＝線）」や、右上の「**2 つ**（＝角のある領域）」では解決できない。右下の「**8 つ**（＝円に近い領域）」であれば解決できるが、これは計算処理に時間がかかる。左下の「**3 つ**（＝三角形）」であれば、問題もほぼ解決できるし、計算処理も比較的速い。よって、この問題例では、隠れ層にニューロンを **3 個**用意すれば、最も効率が良さそうだと分かる。

このようにして、ニューロンの数を決定できる。ただし現実的には、この例のように隠れ層の各ニューロンの特徴をあらかじめイメージできる単純な問題ケースは少ないので、ニューロンの数を増やしたり減らしたりしながら試行錯誤する必要性が発生する。

以上がニューラルネットワークの基本形である。

レイヤー（層）の数を増やす

さて、ニューラルネットワークの基本形は 3 層だったが、隠れ層は **1 層、2 層、3 層**と好きなだけ増やせる、と紹介済みだ。Playground の提供機能では、隠れ層のレイヤーは **6 個まで**増やせる（図 4-12）。ちなみに Playground 上で実際に増やしてみる場合は、こちらを開いて［<数値> 隠れ層］という表記の左にある［+］／［−］ボタンをクリックしてほしい。なおニューロンの数は、先ほどと同様に、各レイヤーで **8 個まで**増やせる。

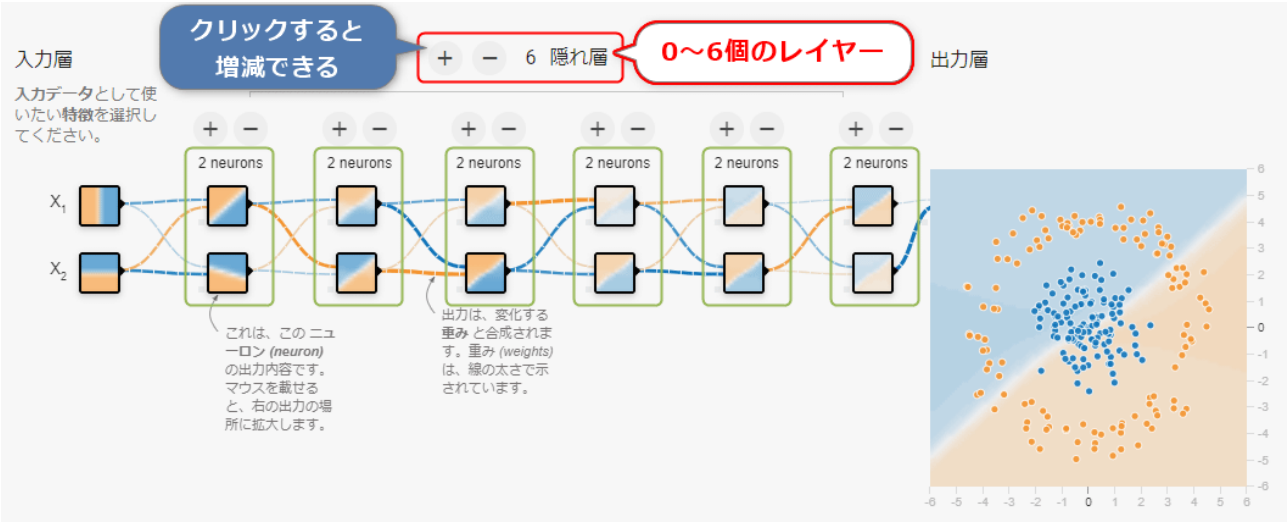

図 4-12　隠れ層のレイヤーの数を最大の 6 個に増やしたところ

隠れ層を **2 層以上**に増やしたニューラルネットワークは、基本形と差別化して、**ディープニューラルネットワーク（DNN：Deep Neural Network**、もしくは**深層ニューラルネットワーク**）と呼ばれる。

では、隠れ層にあるレイヤーの数を増やすと、どんな効果があるのだろうか？　これも先ほど説明したニューロン数を増やす理屈とほぼ同じである。例えば「三角」と「三角」を組み合わせると、非常に複雑な図形が表現できるようになるはずだ。要するに、レイヤーの数を増やせば増やすほど、高度で複雑な問題を解決できるようになるわけである。それこそが、DNN やディープラーニングの真価なのである。

ただし、ニューロン数を増やす際と同様の問題を抱えていて、レイヤー数を増やせば増やすほど計算に時間がかかるようになり、処理スピードが遅くなる。よって、レイヤー数もできるだけ少なくして問題解決できるほど、通常はより良い策となるだろう。Playground で用意している分類問題のデータセットの中で一番複雑なのは「螺旋（らせん、Spiral）」であるが、この問題を解決するには、隠れ層のレイヤーの数が **3 つ以上**になるとより解決しやすくなる。図 4-13 は、隠れ層のレイヤー数が **1 個／ 2 個／ 3 個／ 6 個の**場合に、「螺旋」の分類問題がどのように解決されるかを示したものである。

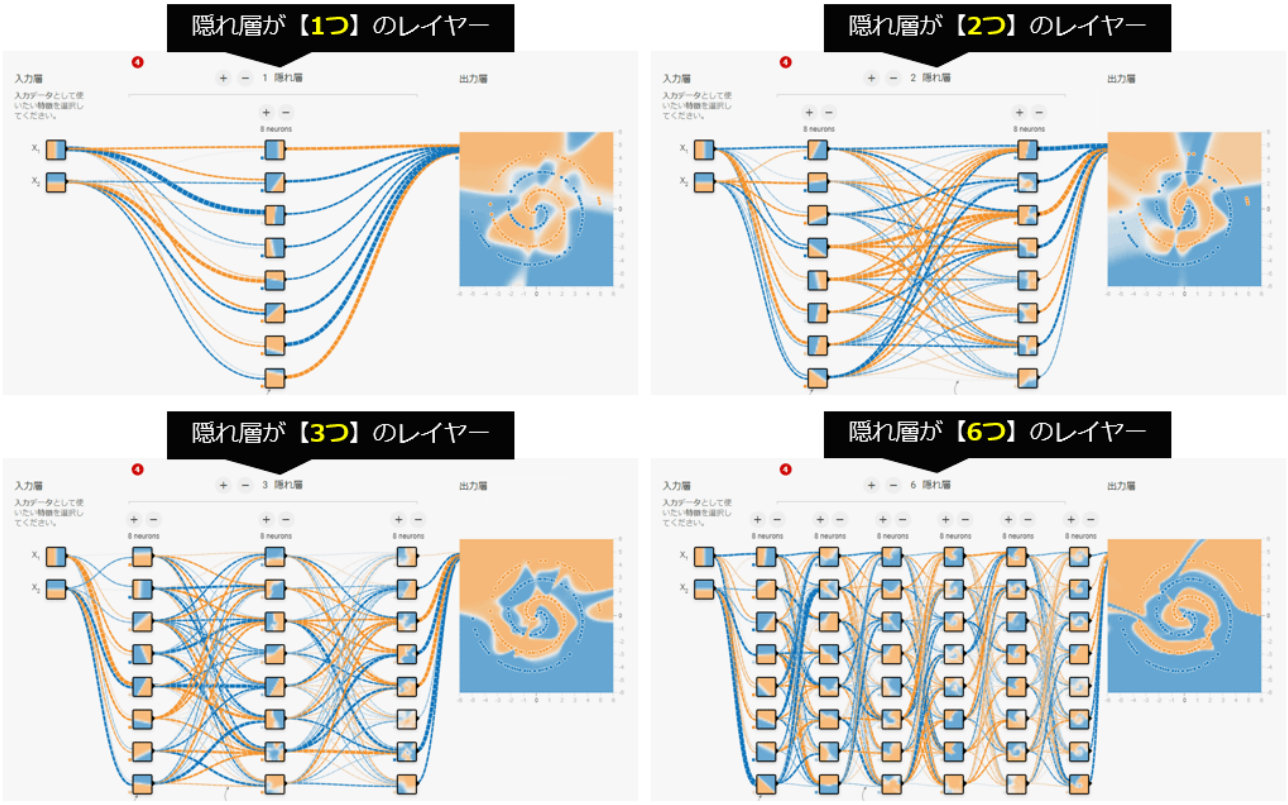

図 4-13 「螺旋（らせん、Spiral）」の分類問題解決例（隠れ層のレイヤー数が 1 個／ 2 個／ 3 個／ 6 個の場合）

※いずれも 1000 回（＝ 1000 エポック）学習した結果（背景）の比較である。

図 4-13 で各出力の背景色に注目すると、左上の「**1 つ**」、右上の「**2 つ**」、左下の「**3 つ**」、右下の「**6 つ**」の順でより正確に問題解決できるようになっていることが分かる。

次の図 4-14 は、前掲の図 4-13 の右上にある「**2 つ**」のレイヤーの例から、入力層と隠れ層の部分を抜き出したものである。各層内にある各ニューロンの、四角内の描画内容に注目してほしい。

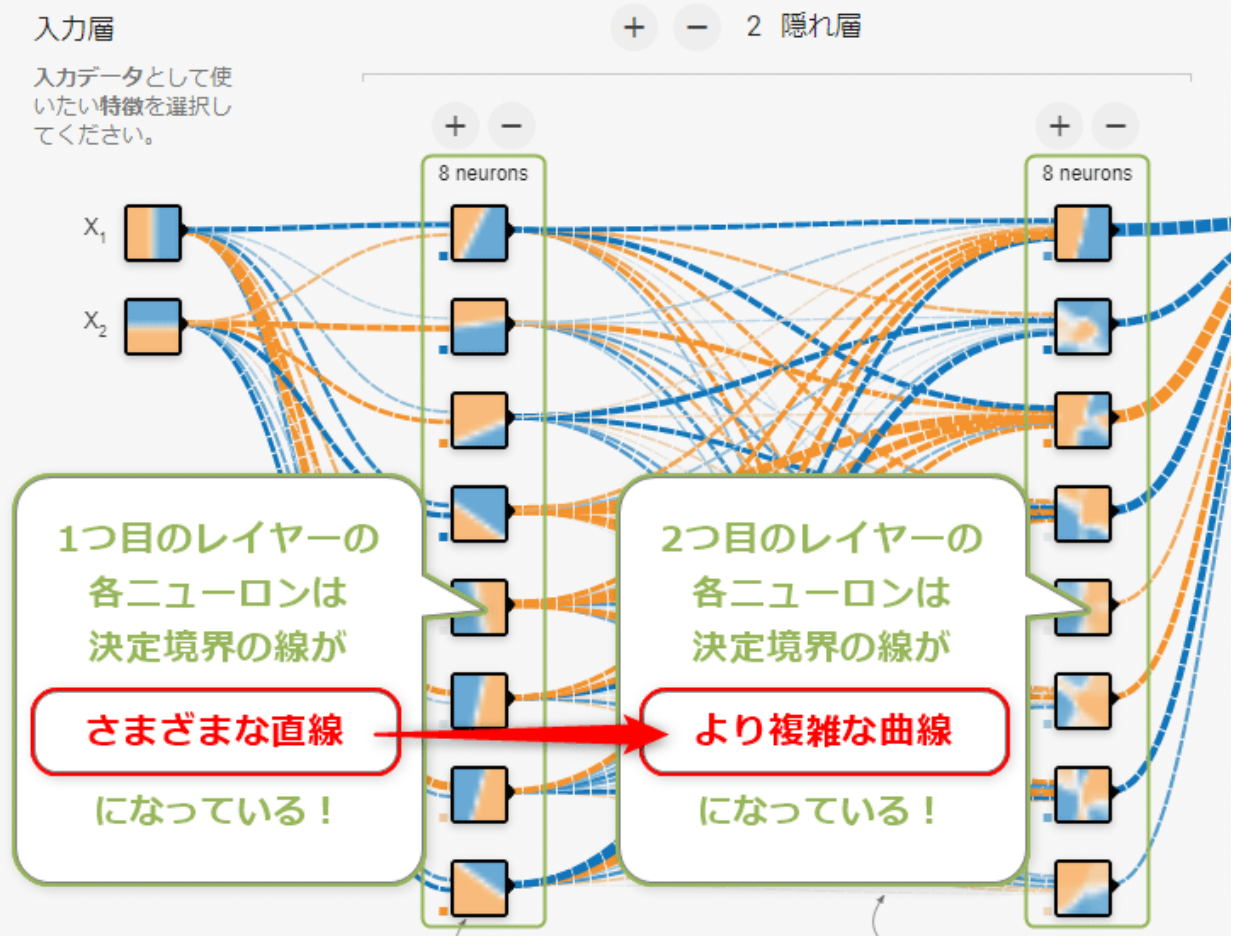

図 4-14 「円（Circle）」の分類問題解決例（隠れ層のレイヤー数が 1 個／ 2 個／ 3 個／ 8 個の場合）

1 つ目のレイヤーにある各ニューロンの描画内容はさまざまな方向の直線だが、**2 つ目**のレイヤーにある各ニューロンの描画内容はより複雑な曲線となっている。「レイヤーの数を増やせば増やすほど、より複雑な曲線や図形が表現できるようになり、高度な問題が解ける」と既に述べたが、これがその証左である。

以上がディープニューラルネットワークの基本である。

Python コードでの実装例

それでは、「ニューラルネットワーク」のモデルを Python で定義してみよう。

モデルの定義

といっても、基本は先ほどのニューロンの定義方法と同じなので、そこからの差異を後述のリスト 4-2 では太字で示した。

ここではモデルのみを定義し、重みやバイアスの初期値は指定しない（通常は明示的に個別の値を指定しない。先ほどは特殊な書き方だった）。また、先ほどはデータを入力してフォワードプロパゲーションを体験したが、（同じような体験となるので）今度は行わないこととする。

ここでは、隠れ層のレイヤーを **2 つ**用意し、各レイヤーのニューロンの数は **3 つずつ**にしてみよう。これは、次のようなコードになる。

```
# ライブラリ「TensorFlow」の tensorflow パッケージを「tf」という別名でインポート
import tensorflow as tf
# ライブラリ「NumPy」の numpy パッケージを「np」という別名でインポート
import numpy as np

# 定数（モデル定義時に必要となる数値）
INPUT_FEATURES = 2  # 入力（特徴）の数：2
LAYER1_NEURONS = 3  # ニューロンの数：3
LAYER2_NEURONS = 3  # ニューロンの数：3
OUTPUT_RESULTS = 1  # 出力結果の数：1

# 今度は重みとバイアスのパラメーターは指定しない（通常は指定しない）

# 積層型のモデルの定義
model = tf.keras.models.Sequential([
  # 隠れ層：1 つ目のレイヤー
  tf.keras.layers.Dense(
    input_shape=(INPUT_FEATURES,),      # 入力の形状（=入力層）
    units=LAYER1_NEURONS,               # ユニットの数
    activation='tanh'),                 # 活性化関数
  # 隠れ層：2 つ目のレイヤー
  tf.keras.layers.Dense(
    units=LAYER2_NEURONS,               # ユニットの数
    activation='tanh'),                 # 活性化関数
  # 出力層
  tf.keras.layers.Dense(
    units=OUTPUT_RESULTS,               # ユニットの数
    activation='tanh'),                 # 活性化関数
])
```

リスト 4-2　ニューラルネットワークのモデル設計

前掲のリスト 4-1 で示した「ニューロンのモデル設計」では、（tf.keras.layers モジュール階層の）Dense クラスのコンストラクターが **1 つだけ**だった。上掲のリスト 4-2 で示した「ニューラルネットワークのモデル設計」では、**3 つ**に増えている。隠れ層に **2 つ目**のレイヤーと出力層が追加されたからだ。

なお、隠れ層の **1 つ目**のレイヤーには、「入力層のデータ入力箇所」を意味するキーワード「input_shape」が指定されているが、それ以降のレイヤーでは、前のレイヤーからデータが流されるため、このキーワード指定が不要になっていることに注意してほしい。

他は特に難しいところはないだろう。このコード例は、**4 層**で構成される「ディープニューラルネットワーク」となっている。**3 層**の「ニューラルネットワークの基本形」にするには「隠れ層：**2 つ目**のレイヤー」の部分をカットすればいいし、逆に **5 層以上**にするには「隠れ層：**2 つ目**のレイヤー」の部分を **3 つ目**、**4 つ目**と増やしていけばよいだけである。

ちなみに Keras では、確認用にモデルの概要を出力することもできる。これには、model.summary() というコードを実行するだけだ。

```
model.summary()

Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 3)                 9
_________________________________________________________________
dense_2 (Dense)              (None, 3)                 12
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 4
=================================================================
Total params: 25
Trainable params: 25
Non-trainable params: 0
_________________________________________________________________
```

図 4-15　モデルの概要出力（Keras 機能：model.summary()）

念のため、図 4-15 の内容も順に説明しておこう。

Keras 内では「sequential_1」という名前のモデルが生成されており、「dense_1」という名前のレイヤー（＝隠れ層：**1 つ目**のレイヤー）の他、「dense_2」（＝隠れ層：**2 つ目**のレイヤー）、「dense_3」（＝出力層）の、合計 3 つのレイヤーがあることが分かる。

出力形状（Output Shape）は、順に **3 個**、**3 個**、**1 個**となる。

入力層から **2 つ目**のレイヤーへの接続線（＝重み）の数は 2×3 ＝ **6 個**で、バイアスの数は **3 個**となるため、合計 **9 個**が「dense_1」のパラメーター数（Param #： Parameter number）となる。同様に計算すると、「dense_2」のパラメーターは合計 **12 個**、「dense_3」のパラメーターは合計 **4 個**である。

パラメーター数をまとめると、9 ＋ 12 ＋ 4 ＝ **25 個**がパラメーターの総数（Total params）となり、学習／訓練可能なパラメーター数も **25 個**である（※訓練不可能は **0 個**。通常は訓練可能だが、必要があるときは訓練不可能にしてパラメーター値を " フリーズ " する機能が Keras にはある）。

ニューラルネットワークのモデル定義方法の基礎は以上で完了である。

Playground の上部にある「**(4)** 手法 " の選択： モデルの定義」を見ると、活性化関数と正則化を指定できる。その隣にある「**(5)** " 学習方法 " の設計： モデルの生成」には学習率や損失関数、最適化、バッチサイズなどがある。これらは学習方法をカスタマイズ／調整するためのもので、**ハイパーパラメーター**と呼ばれている。なお、ここまでに説明してきたニューロンやレイヤーの数もハイパーパラメーターの一種である。

本稿では最後に、活性化関数と正則化について簡単に紹介して終わりとする。

(4) " 手法 " の選択とモデルの定義： 活性化関数

まずは活性化関数から説明しよう。

Playground による図解

活性化関数

まずは Playground を開いて、図 4-16 に示すように［活性化関数 (隠れ層)］欄のプルダウンリストを開いて見よう。するとたくさんの活性化関数が表示される。

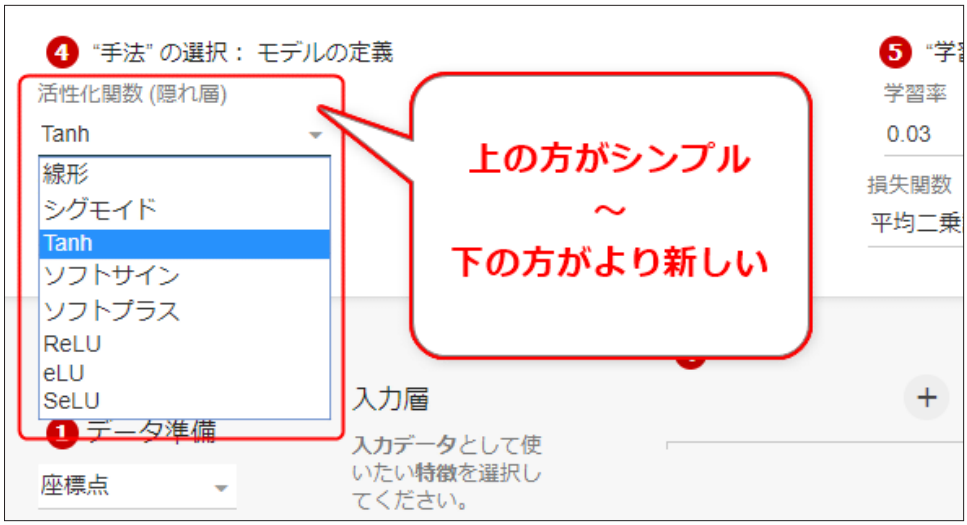

図 4-16　モデルの概要出力（Keras 機能 : model.summary()）

→目次に戻る

活性化関数については、既にシグモイド関数と tanh 関数を紹介した。これら以外にも、研究によってより良いものが次々と考案され続けている。ここでは、代表的なものとしてシグモイド関数と tanh 関数の他に、線形関数と ReLU 関数も取り上げて比較してみよう（図 4-17）。

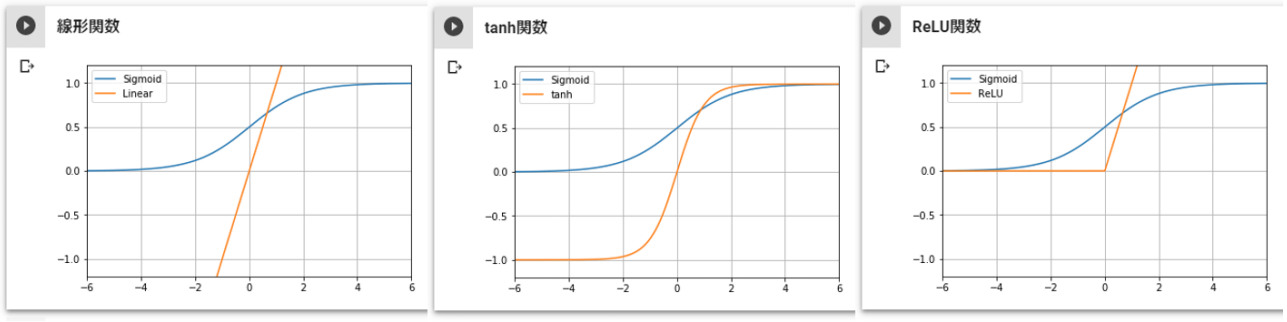

図 4-17　活性化関数の比較：線形関数／ tanh 関数／ ReLU 関数（基準：シグモイド関数）

違いが分かりやすいように、シグモイド関数を比較対象として、線形関数／ tanh 関数／ ReLU 関数の 3 つを表示した。なお、活性化関数は中身の数式よりも、その数式がどのような形の線を描くのかを意識することが重要である。

シグモイド関数は、−∞〜+∞の数値を 0.0 ～ 1.0 の範囲の数値に変換する、と説明済みだ。
一方、線形関数は値を変換せずにそのまま出力する。
tanh 関数も説明済みで、− 1.0 ～ 1.0 の範囲の数値に変換する。
ReLU 関数は、0.0 ～+∞の範囲の数値に変換する。

どの線形関数を使うかは、実際に使ってみて試行錯誤するしかないが、基本的に、

線形関数　＜　シグモイド関数　＜　tanh 関数　＜　ReLU 関数

の順に効率的に学習できる可能性が高い。ちなみに、例えば ReLU 関数のように最小値が **0.0** になる場合は、中間の隠れ層の各ニューロンは **-1.0** などのマイナス値が出力できないため、オレンジ色（**= -1.0**）が表現できず、白色（**= 0.0**）～青色（**= 1.0 以上**）で表現されることになるため、注意してほしい（図 4-18）。

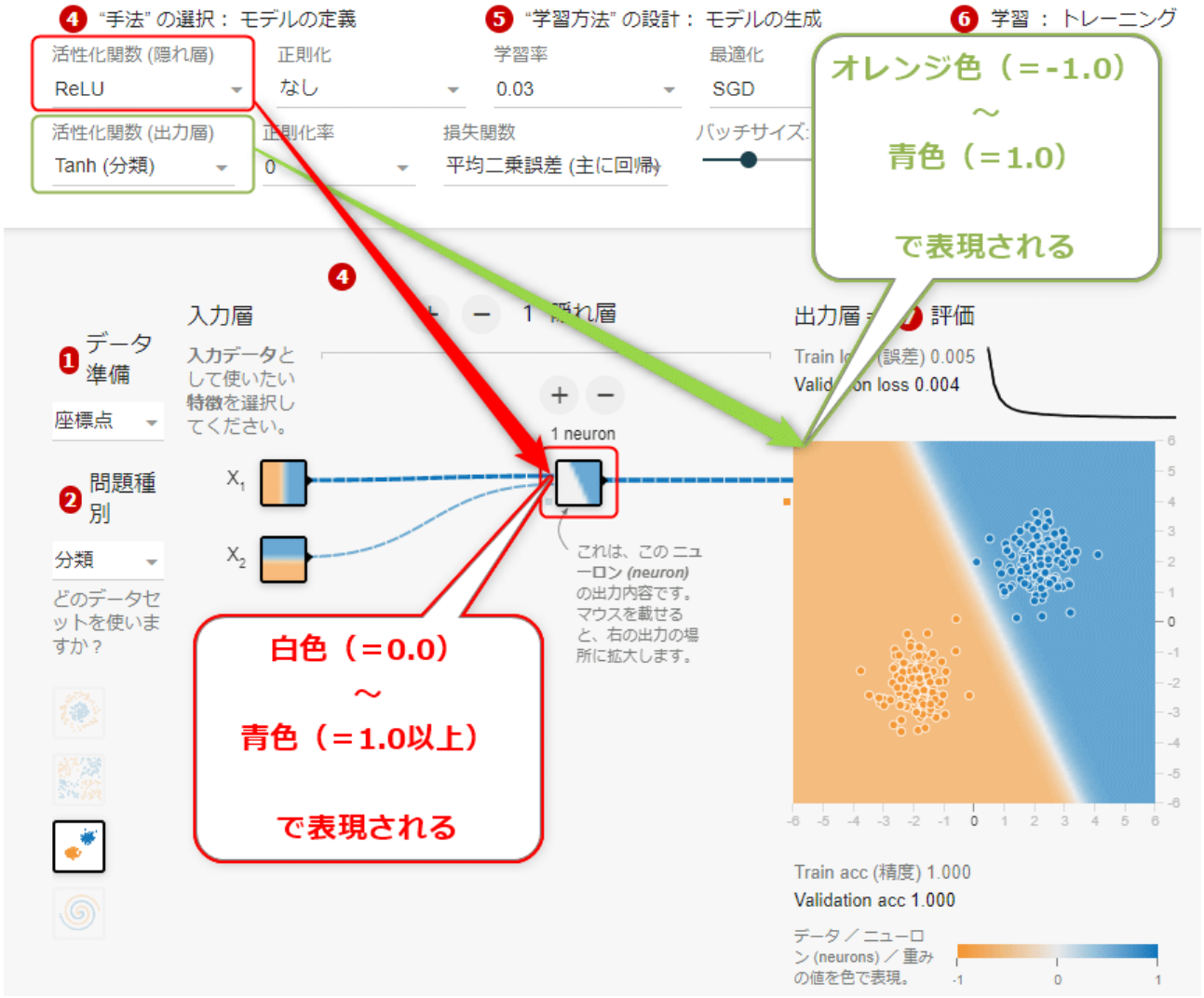

図 4-18　ReLU 関数／tanh 関数で描画されるニューロンの決定境界

なお、図 4-18 にも示した［活性化関数 (隠れ層)］欄のプルダウンリストでは、分類問題の場合、「Tanh 関数」しか選択できないようにしてある。これは、最終的な出力値をオレンジ色（**＝ -1.0**）～青色（**＝ 1.0**）の範囲で表現したいためである。

Python コードでの実装例

前述の通り、活性化関数を指定する箇所は、（tf.keras.layers モジュール階層の）Dense クラスのコンストラクターの引数「activation」である。リスト 4-1 やリスト 4-2 では「**tanh**」と記述した。この文字列を指定したい活性化関数の名前に書き換えるだけである。

リスト 4-3 では、実際に「**sigmoid**」という記載に変更している。ただし、全ての隠れ層に対する記載を変更する必要があるため、ACTIVATION という定数に切り出してから記載を変更した。なお、前後のコードは全く同じになるので省略している。

```
# ……省略……

# 定数（モデル定義時に必要となる数値）
# ……省略……
ACTIVATION = 'sigmoid'  # 活性化関数（ここを書き換える）：シグモイド関数

model = tf.keras.models.Sequential([
  # 隠れ層：1 つ目のレイヤー
  tf.keras.layers.Dense(
    input_shape=(INPUT_FEATURES,), # 入力の形状（＝入力層）
    units=LAYER1_NEURONS,          # ユニットの数
    activation=ACTIVATION)         # 活性化関数
    # ……省略……
])
```

リスト 4-3　活性化関数の指定例

Playground で選択可能な活性化関数は、Keras でも対応している。具体的には、以下のものを文字列で記述できる（※一部の関数は本稿の内容をできるだけシンプルにするために説明していない。この他、「softmax」なども記述可能である）。

- **linear**（線形）
- **sigmoid**（シグモイド）
- **tanh**（Tanh）
- **softsign**（ソフトサイン）
- **softplus**（ソフトプラス）
- **relu**（ReLU）
- **elu**（eLU）
- **selu**（SeLU）

（4）“ 手法 ” の選択とモデルの定義：正則化

次に正則化について説明しよう。

Playground による図解

正則化

正則化（Regularization、Regularizer）は、主に過学習（後述）を防ぐためのテクニックの一つである。過学習を防ぐテクニックとして他にはドロップアウト（Dropout）が有名であるが、Playground にはドロップアウト機能がないため、今回は説明を割愛する。

そもそも**過学習（overfitting、過剰適合）**とは、訓練データに特化して学習し過ぎると、それがかえって未知のデータ（テストデータなど）には適合しなくなることである。例えるなら、演歌ばかり歌っていると、どんな曲を歌うときでも、こぶしを利かせて歌ってしまうイメージだ。

正則化には、主に下記の 2 種類がある。

- **L1**：不要なニューロンを抑制／排除して、モデルの中身をシンプル（疎）にする
- **L2**：重みパラメーターにペナルティを加えて、学習し過ぎないようにする

特に「L2」による**重みの正則化**は、ニューラルネットワークでは**重み減衰（Weight Decay）**と呼ばれ、よく利用される。どちらを使うかはケースバイケースだが、基本的には L2 を使うことが多いだろう。

過学習が発生しやすいのは、データに（外れ値などの）ノイズが多く、しかも訓練データの数が少ない場合である。この状況を作るには、Playground を開いて、前回説明した［（3）前処理］の箇所で、

- データ分割：訓練用を **10%**
- ノイズ：**50%**

などと設定すればよい。実際にこれを行っているのが、図 4-19 である。

図 4-19　ノイズ（外れ値など）を多く、しかも訓練データを少なくする設定例

正則化は、上部で指定でき（図 4-19）、「**なし**」「**L1**」「**L2**」から選択できる。図 4-20-1 は「なし」、図 4-20-2 は「L1」、図 4-20-3 は「L2」を選択した場合の、ニューラルネットワークの例である。正則化率は、全て「**0.03**」を指定した（詳細後述）。

図 4-20-1　正則化「なし」の場合

図 4-20-2　正則化「L1」の場合

→目次に戻る

図 4-20-3　正則化「L2」の場合

※図 4-20-1 ／ 2 ／ 3 いずれも 1000 回（＝ 1000 エポック）学習した結果（背景）の比較である。

説明した通り、正則化「なし」は過学習を起こしており、未知のデータを入れると不正解を多く出す可能性のあるモデルとなってしまっている。

L1 正則化を行うと、隠れ層のニューロンが白色になっているものがあり、ニューロンが抑制／排除されているのが分かる。その結果、モデルがシンプルになり、適切に問題が解決できるようになっている。

L2 正則化を行うと、「なし」と比べて接続線（＝重み）が細く薄くなっており、学習し過ぎないようになったことが分かる。その結果、適切に問題が解決できるようになっている。

以上が正則化の効果だ。

正則化率

後は正則化率であるが、**0.1 ～ 0.001** など任意の数値を指定できる。先ほどの例（前掲の図 4-20-3）では **0.03** を指定することで、うまく過学習を軽減できた。それ以外の正則化率、具体的には **0.003 ／ 0.03 ／ 0.3 ／ 3** を指定した場合の学習結果の違いを、図 4-21 にまとめた。

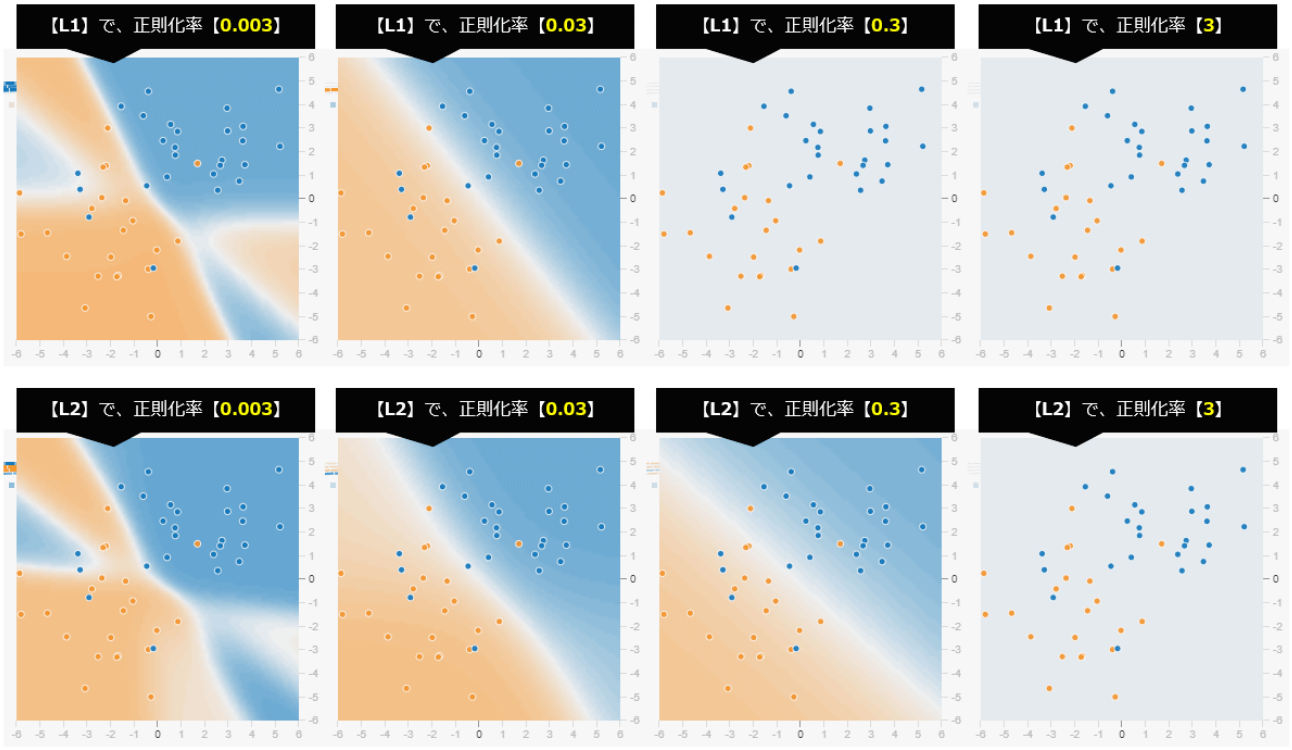

図 4-21　正則化「L1」（上段）と「L2」（下段）で正則化率の違いの比較（0.003 ／ 0.03 ／ 0.3 ／ 3 の場合）

※いずれも 1000 回（学習単位は「エポック」と呼ばれる）学習した結果（背景）の比較である。

まず、左端の **0.003** では、L1 も L2 も数値が小さすぎて、ほとんど学習結果に影響していないことが分かる。

次の **0.03** は、L1 も L2 もほどよく学習できているようである。

その次の **0.3** は、L1 は背景が白色になっており、全く学習できなくなってしまっている。L2 はそこそこ学習できているが、前の **0.03** よりも色が薄くなってしまった。

右端の **3** は、L1 も L2 も数値が大きすぎて、全く学習できていない。

このように、正則化率も程よい数値を試行錯誤で模索する必要がある。だいたいは **0.01** 付近を中心に探すとよい。

Python コードでの実装例

正則化についてもコードを示しておこう。重みの正則化を指定する箇所は、（tf.keras.layers モジュール階層の）Dense クラスのコンストラクターの引数「kernel_regularizer」である。

リスト 4-4 では、実際に「**L2** 正則化で、正則化率は **0.03**」を指定している。ただし、全てのレイヤー（隠れ層と出力層）に対して追記する必要があるため、REGULARIZATION という定数に切り出して指定した。なお、前後のコードはこれまでと全く同じになるので省略した。

```
# ……省略……

# 定数（モデル定義時に必要となる数値）
# ……省略……
REGULARIZATION = tf.keras.regularizers.l2(0.03)  # 正則化：L2、正則化率：0.03

model = tf.keras.models.Sequential([
  # 隠れ層：1 つ目のレイヤー
  tf.keras.layers.Dense(
    input_shape=(INPUT_FEATURES,),      # 入力の形状（＝入力層）
    units=LAYER1_NEURONS,               # ユニットの数
    activation=ACTIVATION,              # 活性化関数
    kernel_regularizer= REGULARIZATION)  # 正則化
    # ……省略……
])
```

リスト 4-4　正則化の指定例

このように、引数「kernel_regularizer」には、（tf.keras.regularizers モジュール階層の）l2() 関数を指定すればよい。**L1** 正則化の場合は、l1() 関数である。いずれの関数も引数に、正則化率を指定する。

なお、l1() ／ l2() 関数には、**0.01** がデフォルト引数で指定されている。そのため、正則化率を **0.01** としたい場合は、引数なしで呼び出せる。

今回は、少し長くなったが、（**4**）のニューラルネットワークのモデルの定義方法と、活性化関数、正則化について説明した。次回後編では、（**5**）～（**8**）の学習、評価、テストについて説明する。

TensorFlow 2 + Keras（tf.keras）入門：

第 3 回　ディープラーニング最速入門 —— 仕組み理解 × 初実装（後編）

いよいよ、ディープラーニングの学習部分を解説。ニューラルネットワーク（NN）はどうやって学習するのか、Python とライブラリではどのように実装すればよいのか、をできるだけ簡潔に説明する。

一色政彦，デジタルアドバンテージ（2019 年 12 月 16 日）

前々回、ディープラーニングの大まかな流れを、下記の **8 つ**の工程で示した。

- （**1**）データ準備
- （**2**）問題種別
- （**3**）前処理
- （**4**）“ 手法 ” の選択：モデルの定義
- （**5**）“ 学習方法 ” の設計：モデルの生成
- （**6**）学習：トレーニング
- （**7**）評価
- （**8**）テスト

このうち、（**1**）～（**4**）を前回までに説明した。引き続き今回は、（**5**）～（**8**）を解説する。いよいよ今回で完結である。それではさっそく説明に入ろう。※脚注や図、コードリストの番号は前回からの続き番号としている（前編・中編・後編は、切り離さず、ひとまとまりの記事として読んでほしいため連続性を持たせている）。

Google Colabで実行する　　GitHubでソースコードを見る

（5）“ 学習方法 ” の設計とモデルの生成：損失関数

前回の（**4**）でニューラルネットワークのモデルが定義できた（※前回示したように本連載では Keras を使用）。次に（**6**）学習を行うわけだが、その前に「どのように学習するか」の決定、つまり「（**5**）“ 学習方法 ” の設計」をしておく必要がある。具体的には、第 1 回から使っている「ニューラルネットワーク Playground - Deep Insider」（以下、Playground）の上部にある図 5-1 の 3 カ所、

- **損失関数**
- **最適化**
- **学習率**

がそれに該当する。なお、**バッチサイズ**は「最適化」の節で軽く触れ、後述の「学習」の章で詳しく説明する。

図 5-1　学習方法の設計

この 3 つを設定すればよいわけだ。1 つずつ説明していく。まずは損失関数を説明しよう。今回も、まずは Playground を使って図解で説明し、その後で具体的なコード実装方法を紹介する。

Playground による図解

Playground を開いて、上部にある（**5**）の［損失関数］欄で「平均二乗誤差」を選択する（図 5-2）。

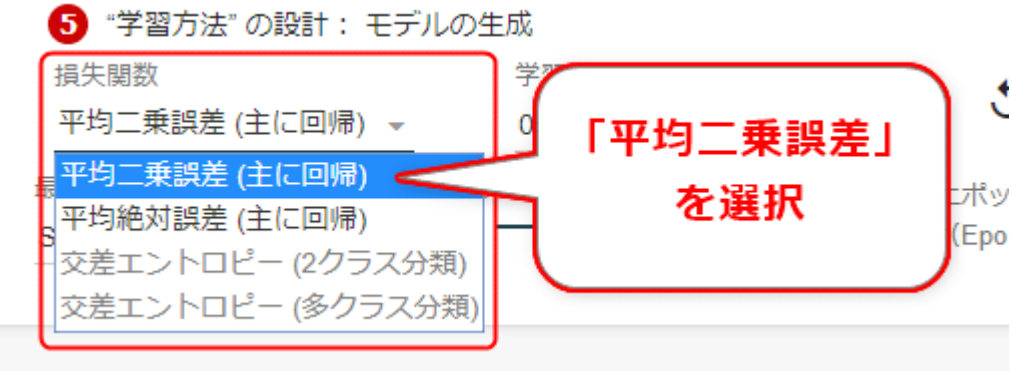

図 5-2　［損失関数］で「平均二乗誤差」を選択

※注意：「交差エントロピー」は、Playground に未実装のため、選択できない。

回帰問題／分類問題でよく使われる損失関数

「損失関数」や「平均二乗誤差」の意味は後述するが、平均二乗誤差が最も代表的な**損失関数**（意味は後述）である。分類問題か回帰問題かによって、よく使われる損失関数に違いがある。選択肢に「(主に回帰)」「(2 クラス分類)」「(多クラス分類)」というカッコ書きがあるが、その違いを示したものである。まとめておくと、

- **平均二乗誤差（MSE：Mean Squared Error）**＝回帰問題でよく使う損失関数
- **交差エントロピー誤差（Cross Entropy Error）**＝分類問題でよく使う損失関数

ということになる。今回は分類問題を扱うが、最も有名な「平均二乗誤差」を使う。交差エントロピー誤差については説明を割愛する（※「平均絶対誤差」については、後で「平均二乗誤差」との違いを示すので、それを参考にしてほしい）。

誤差とは

いずれの損失関数も、「誤差」を調べるための関数（＝入力を受けて、何らかの計算をして、出力するもの）であると、字面から分かるだろう。**誤差（エラー：error）**とは、言葉通りの意味で、計算で得られた値と、正解の値にある、「ズレ」のことである。

機械学習の場合、ニューラルネットワークのモデルに入力した値が、フォワードプロパゲーション（順伝播）によって処理され、出力される（※前回解説した）。その出力結果の値と、正解を示す教師ラベル（※第 1 回で解説した）の値がどれくらいズレているか（＝誤差）を調べるわけである（図 5-3）。

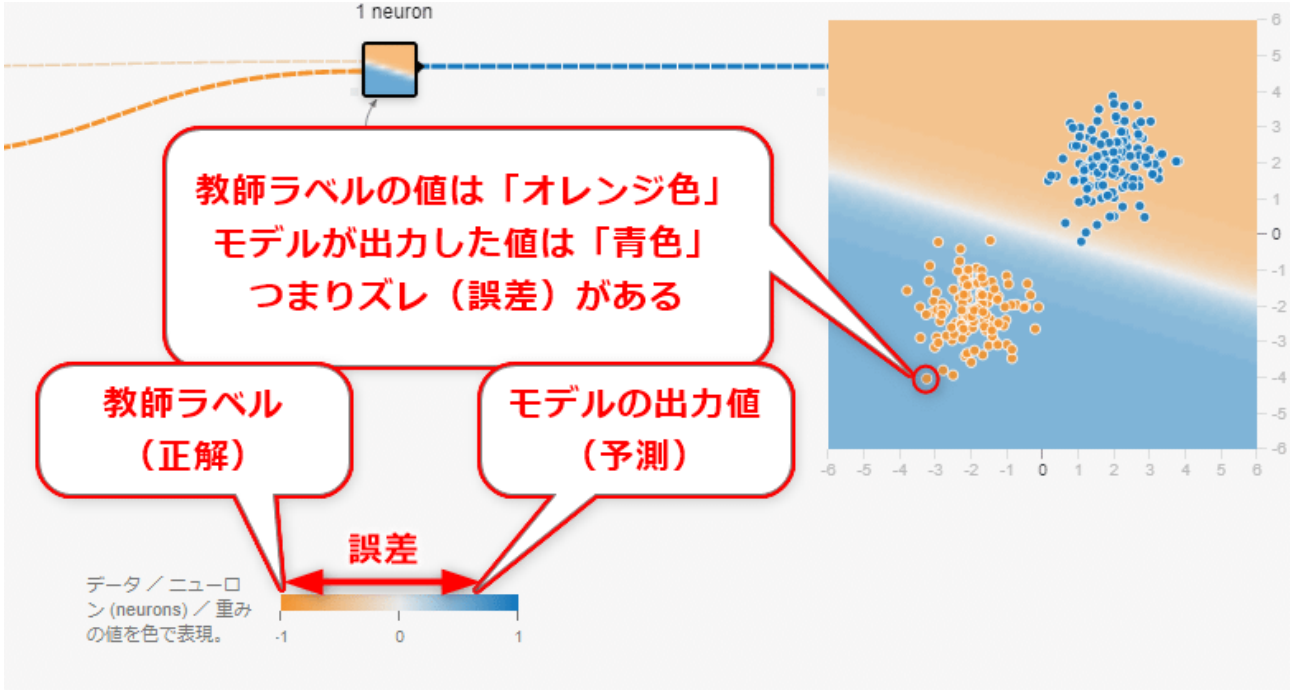

図 5-3　誤差

あとは、全てのデータに対して誤差を調べていき、それを平均すれば、モデルの平均的な誤差が算出できると考えられるだろう。しかし実は、その計算方法ではうまくいかない。

なぜ「二乗」するか

例えば 3 つのデータがあり、それぞれの誤差が「**+1.2**」「**-1.0**」「**-0.2**」だったとしよう。「平均的に正解からどれくらいズレているか」を見たい場合に、これらを単純に全部合計してデータ数の **3** で割ってしまうと「+1.2-1.0-0.2=**0.0**」となってしまう。つまりズレが全くないことになってしまう。これは違う。

ズレを計測するためには、距離が重要であるため、マイナスはプラスとして計算、つまり絶対値化して計算する必要がある。これを行うのが、前掲の「**平均絶対誤差**」（**MAE：Mean Absolute Error**）である。

しかし数学や統計学の世界では、実は絶対値は扱いにくい。マイナスをプラスにする計算方法としては、絶対値の他にも、二乗計算がある（※数字の右肩に小さく「2」と書く、中学で学ぶ計算式で、例えば **-2** の二乗は **+4** となる）。そして、二乗計算では、特に微分計算（高校で習う計算方法）がしやすくなって便利という利点もある。そのため数学や統計学では、マイナスをプラスにする計算には二乗がよく用いられるのだ。

「誤差を二乗してから平均を取る計算」である**平均二乗誤差** ***2** は、このような理由からニューラルネットワークで代表的な「損失関数」となっている。なお、平均二乗誤差などの計算結果の数値は、**損失（ロス：loss）**と呼ばれる。この損失を求めるための関数であるため、**損失関数（Loss Function）**と呼ばれるのである。

***2** 平均二乗誤差の「平均」とは、誤差の二乗和を「その**データ数**で割る」ことを意味する。しかし書籍などの説明によっては「**2**」で割る、いわば「**“1/2” 二乗和誤差**」（**SSE：Sum of Squared Error**）が用いられている。これは、「二乗した変数を微分した際に係数として **2** が出てくるので、それに **1/2** を掛けたら **1** になって計算が楽になるよね」という、いわばマジックナンバーなのである。例えばニューラルネットワークの学習をフルスクラッチで作ってみる場合、この “1/2” 二乗和誤差を使うと実際に楽である。ちなみに、Playground の損失関数も、実は平均二乗誤差ではなく “1/2” 二乗和誤差を用いているが、本稿のコード説明との一貫性を出すため、表記は「平均二乗誤差」とした。

Python コードでの実装例

では、損失関数として「平均二乗誤差」（Mean Squared Error）をコードで実装してみよう。tf.keras（＝TensorFlow 同梱の Keras）で、損失関数の計算式を独自に定義するのは難しくないが、Keras には代表的な損失関数があらかじめ用意されており、しかも文字列で指定するだけである（※前回説明した活性化関数と同じ）。今回は定数として切り出し、リスト 5-1 のように記載することにした。

```
# 定数（学習方法設計時に必要となるもの）
LOSS = 'mean_squared_error'  # 損失関数：平均二乗誤差
```

リスト 5-1　損失関数の定義

Playground で選択肢に挙げている損失関数は、Keras でも対応している。具体的には、以下のものを文字列で記述できる（※一部の関数は本稿の内容をできるだけシンプルにするために説明していない）。

- **mean_squared_error**（平均二乗誤差）
- **mean_absolute_error**（平均絶対誤差）
- **binary_crossentropy**（交差エントロピー誤差：2 クラス分類）
- **categorical_crossentropy**（交差エントロピー誤差：多クラス分類）

（5）“ 学習方法 ” の設計とモデルの生成：最適化

次に最適化を説明しよう。

Playground による図解

Playground 上では、（5）の［最適化］欄で「**SGD**」を選択する（図 5-4）。バッチサイズは後述するが、ここでは何も考えずに「**1**」を選択してほしい。

図 5-4 ［最適化］で「SGD」を選択

※注意：「SGD」以外は、Playground に未実装のため、選択できない。

「最適化？」「SGD ？」……疑問がいっぱいあるだろう。まずは最適化とは何かから説明する。

最適化とは

最適化（Optimization）とは、言葉通り、最も適している状態に変えることである。では、「ニューラルネットワークにおいて、最も適している状態とは、どのようなことか？」というと、先ほどの損失（loss）を極限まで最小化できていることを指す。

つまり、先ほどの損失関数（平均二乗誤差）の結果の数値をできるだけ **0** に近づければよいわけである。その最適化手法として、さまざまな方法（**最適化アルゴリズム**と呼ぶ）が考えられている。その基本が、「勾配法」なのである。

勾配法とは

勾配法（Gradient method）とは、勾配（gradient）、つまり坂道を下っていく（descent）ように計算することで最適化を行う手法のことである。

先ほど選択した「SGD」は、この勾配法の一種で、**確率的勾配降下法（SGD：Stochastic Gradient Descent）**を意味する。SGD は、最適化の計算を確率論を用いて（＝ stochastic に）実行することで、全部一括（＝**バッチ**と呼ぶ）の訓練データではなく、1 つ、もしくは複数セット（＝**ミニバッチ**と呼ぶ）の訓練データで学習して、その単位ごとに重みやバイアスを更新していく方法である（※詳しくは「バッチサイズ」の章で解説する）。先ほど［バッチサイズ］で「**1**」を選択したのは、このためである。

最適化を理解するキモは、「勾配法」を理解することにある。上記の説明では「何を言っているか」が分からないと思うので、まずは図 5-5 を見てイメージをしてもらおう。

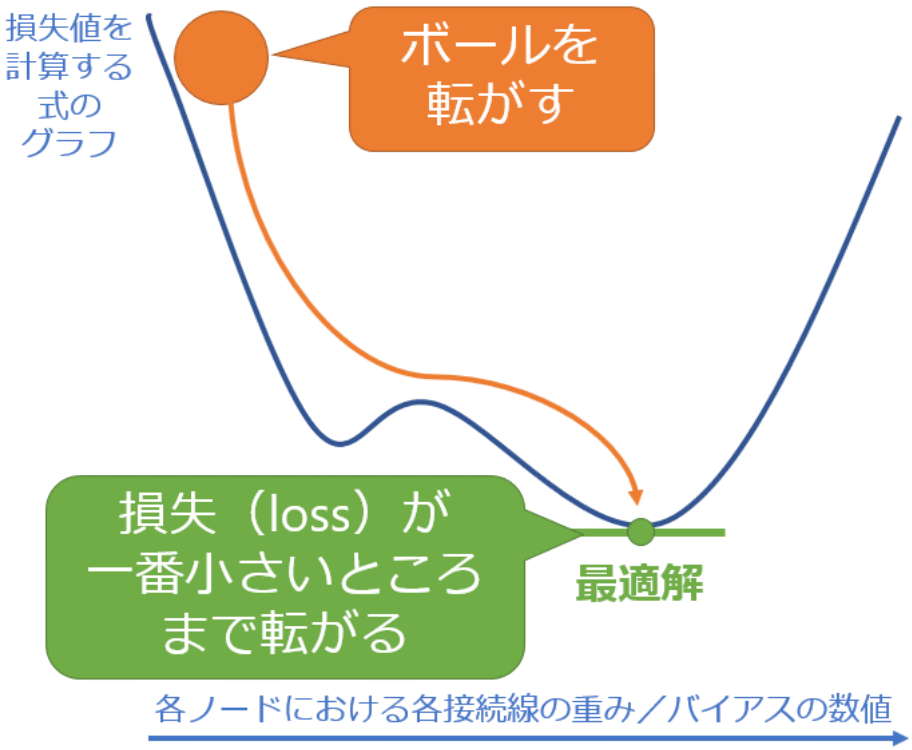

図 5-5　勾配法のイメージ

図 5-5 に示す青色の曲線が勾配（＝坂道）である。その坂道の線の上を、オレンジ色のボールが進むように投げた、とイメージしてほしい。投げた（＝学習した）回数だけ、ボールは進み、最終的に一番低い位置（＝損失が一番小さい場所、**最適解**）まで転がっていく。ただし、この世界は無重力であり、重力によって自動的に一番低い位置に転がるわけではく、1 回の学習によって少しずつ位置を変えていく、と考えてほしい。

ボールを投げる際には、青色の曲線上にある現在位置から見て、「坂道の左側が下りか、それとも右側が下りか」を判断、数学的に言うと「傾き」を判断して、左右どちらにボールを動かすかを決める。数学では、曲線における 1 地点（＝ボールがある位置）の傾きは、微分によって計算できる。また、パラメーターが複数ある曲線の傾きは、偏微分によって計算できる。

【コラム】偏微分と線形代数と統計学

ディープラーニングの習得で、「偏微分が必修」となるのは、上記のように最適化の計算のためである。

また「線形代数も必修」とされるのは、大量のデータをまとめて計算する際に、線形代数の行列演算が必要になるためである。

偏微分や線形代数の計算は、TensorFlow ／ Keras のようなライブラリが内部で行ってくれるので、単に実装するだけであれば、あまり気にする必要はない。しかし理論面を考える場合や、最適化アルゴリズムの違いを厳密に理解したい場合には、こういった数式の理解が必要となる。

さらに、統計学も最低、大学基礎教養レベル（統計検定 2 級レベル）の知識があった方が、ニューラルネットワークや機械学習の理論が理解しやすくなる。例えば最適化＆勾配法は回帰分析や重回帰分析と同じ考え方がベースになっているなど、統計学をベースとした考えが多いからだ。

本稿の内容をマスターしたら、ぜひ長期戦で、数学や統計学の知識も増やしていってほしい。

Python コードでの実装例

では、最適化として「SGD」（確率的勾配降下法）をコードで実装してみよう。これも、Keras には代表的な最適化アルゴリズム（**Optimizer：オプティマイザ**）があらかじめ用意されており、tf.keras.optimizers モジュール階層（＝名前空間）のクラスを使うだけなので簡単である。

今回はクラス名を定数として切り出し、リスト 5-2 のように記載することにした。なお、クラスのコンストラクター（厳密には __init__ 関数）に指定する引数は後述する。

```
# （必要に応じて）TensorFlow v2 の最新バージョンにアップグレードする必要がある
#!pip install --upgrade tensorflow
# ライブラリ「TensorFlow」の tensorflow パッケージを「tf」という別名でインポート
import tensorflow as tf

# 定数（学習方法設計時に必要となるもの）
OPTIMIZER = tf.keras.optimizers.SGD  # 最適化：確率的勾配降下法
```

リスト 5-2　最適化の定義

※本稿は、TensorFlow v2 を前提とする。前回、TensorFlow v2 の最新バージョンへのアップグレード方法を説明している。

Playground で選択肢に挙げている最適化アルゴリズム（オプティマイザ）は、Keras でも対応している。具体的には、以下のものが使える（※代表的なもののみ掲載）。

- **SGD**（確率的勾配降下法）
- **Adagrad**
- **RMSprop**
- **Adadelta**
- **Adam**
- **Adamax**
- **Nadam**

※中身のアルゴリズムの説明は簡単ではないので、本稿では説明を割愛する。どれを使ってもよいが、まずは基本となる SGD を使ってみるとよい。

（5）“ 学習方法 ” の設計とモデルの生成：学習率

続いて、非常に重要な「学習率」を説明しよう。

Playground による図解

Playground 上では、（5）の［学習率］欄で「0.03」を選択する（図 5-6）。

図 5-6 ［学習率］で「0.03」を選択

「学習率」は、最適化と関連する設定事項であり、「1 回の学習でニューラルネットワーク内の重みやバイアスを更新する量の調整値」を表す（＝ハイパーパラメーターの一つ）。といってもよく分からないと思うので、これもボールを転がす例でイメージをしてもらうことにする。図 5-7 を見てほしい。

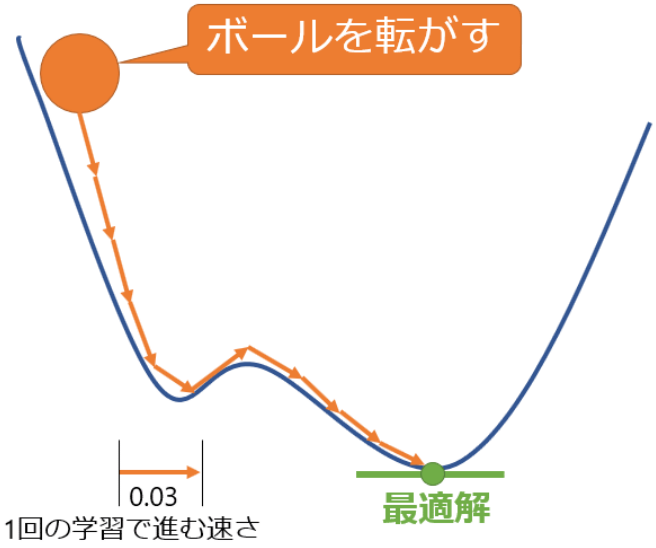

図 5-7　学習率のイメージ

図 5-7 では、ボールを 1 回投げると、そのたびに、最適解に向かって少しずつ転がっていく。ボールのスピードが速いと大量に転がり、遅いと少ししか転がらない。そのボールを投げるスピードに該当する数値が、**学習率（learning rate）**なのである。

学習率には、**0.1 ～ 0.001** など任意の数値を指定できる。大きい学習率を設定した方が、より速く学習できる。よって「大きい値の方がよいのでは」と思うかもしれないが、それはそれでなかなか学習が収束しない問題が発生したりするので、一概には言えない。例えば図 5-7 が学習率に **0.03** を指定した場合のイメージだとすると、図 5-8 は **0.3** を指定した場合のイメージである。

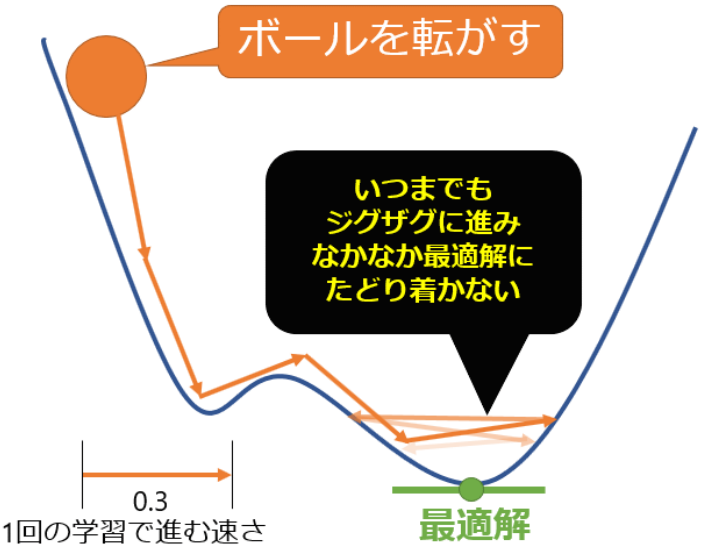

図 5-8　大きすぎる学習率の問題：収束しない

図 5-8 は矢印が長すぎるため、最適解を通り越して行きすぎてしまい、なかなか最適解にたどり着かない。その結果、最適解の近辺をいつまでもジグザグと行ったり来たりしてしまう。つまり、学習が最適解にきれいに収束しない問題が起こっているのである。

なら、「小さい値の方がよい」と思うかもしれないが、それだとなかなか学習が終わらない問題が発生したりするので、これも一概には言えない。それだけでなく、本当の最適解（**global optimal solution**、大局的な最適解）ではなく、局所的な最適解（**local optimal solution**、**局所解**、いわば「谷」）に収束して、その谷から抜け出せなくなる場合があるのだ（図 5-9）。

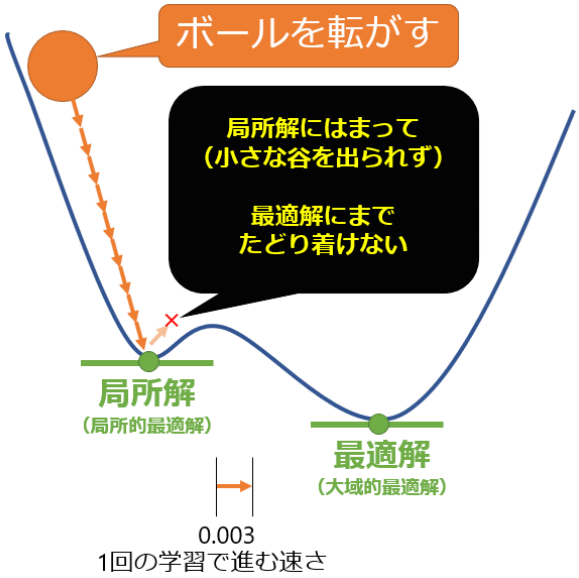

図 5-9　小さすぎる学習率の問題：学習が遅い、局所解にとらわれる

全ケース一律に当てはまる数値があるわけではないので、ケースバイケースで適切な学習率を試行錯誤しながら探す必要がある。だいたい **0.1** 前後から数値を小さくしていく方向で探せばよい。できるだけ高速に処理できて、しかも精度がそこそこよい学習率が、一般的には「より良い学習率」と言えるだろう。参考までに、**0.003 ／ 0.03 ／ 0.3 ／ 3** を指定した場合の学習結果の違いを、図 5-10 にまとめた。

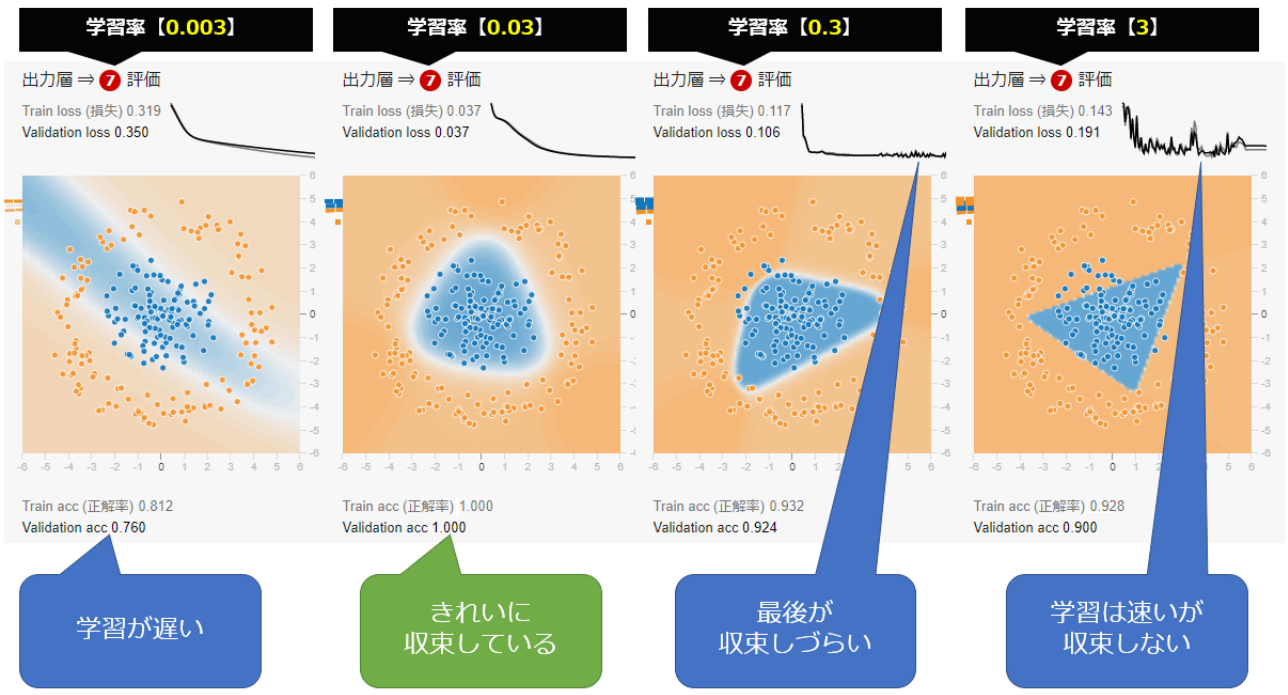

図 5-10　学習率の違いの比較（0.003 ／ 0.03 ／ 0.3 ／ 3 の場合）

※いずれも 100 回（学習単位は「エポック」と呼ばれる）学習した結果の比較である。

→目次に戻る

それぞれの結果の上部にある［Train loss］というのは「訓練データにおける損失」で、［Validation loss］は「精度検証データにおける損失」のことである。また、下部に表示されている［Train acc］（acc = accuracy）というのは「訓練データにおける正解率」で、［Validation acc］は「精度検証データにおける正解率」のことである。正解率（精度）とは、言葉通り、モデルの出力結果が「何%正解を出すか」の数値である。

まず、左端の **0.003** では、学習が遅く、100 回学習しただけでは［Train loss］が **0.319** などと **0.0** に近づききれていない。これはまだ学習途中を意味する。

次の **0.03** は、ほどよく学習できている。

その次の **0.3** は、［Train loss］のグラフの右端がギザギザとなっており、損失が大きくなったり小さくなったりと揺れている。これは、前述の「収束しない」に相当する現象である。

右端の **3** は、［Train loss］のグラフが激しくジグザグになっており、学習がうまく進んでいないことを表している。

Python コードでの実装例

では、学習率として「0.03」をコードで実装してみよう。この数値も定数として定義しておく（リスト 5-3）。

```
# 定数（学習方法設計時に必要となるもの）
LEARNING_RATE = 0.03    # 学習率：0.03
```

リスト 5-3　学習率の定義

以上で、「損失関数」「最適化」「学習率」の定義が終わった。あとはこれらを使って、モデルを生成すればよい。その作業に入る前に、前回の復習として、モデル定義のコードを見てみよう。リスト 5-4 の内容は前回の説明を理解していれば分かるはずである。

```
# ライブラリ「TensorFlow」の tensorflow パッケージを「tf」という別名でインポート
import tensorflow as tf

# 定数（モデル定義時に必要となる数値）
INPUT_FEATURES = 2    # 入力（特徴）の数：2
LAYER1_NEURONS = 3    # ニューロンの数：3
LAYER2_NEURONS = 3    # ニューロンの数：3
OUTPUT_RESULTS = 1    # 出力結果の数：1
ACTIVATION = 'tanh'    # 活性化関数（ここを書き換える）：tanh 関数
```

```
# 積層型のモデルの定義
model = tf.keras.models.Sequential([

  # 隠れ層：1 つ目のレイヤー
  tf.keras.layers.Dense(
    input_shape=(INPUT_FEATURES,),      # 入力の形状（＝入力層）
    units=LAYER1_NEURONS,               # ユニットの数
    activation=ACTIVATION),             # 活性化関数

  # 隠れ層：2 つ目のレイヤー
  tf.keras.layers.Dense(
    units=LAYER2_NEURONS,               # ユニットの数
    activation=ACTIVATION),             # 活性化関数

  # 出力層
  tf.keras.layers.Dense(
    units=OUTPUT_RESULTS,               # ユニットの数
    activation='tanh'),                 # 活性化関数（※ tanh 固定）
])
```

リスト 5-4　【前回の復習】モデルの定義

　また、Keras では、学習結果を評価するための指標を、モデル生成時に指定しておく必要がある。回帰問題では**損失**を見ればよいだろうが、分類問題では「正解率は何%か」を示す**精度（accuracy：正解率、正確度）**も見たいだろう。Keras では、下記のような評価指標が**評価関数（Evaluation Function）**として用意されている（※代表的なもののみ掲載。それぞれの具体的な使い方の説明は割愛する）。

- **accuracy**（正解率）
- **binary_accuracy**（2 クラス分類の正解率）
- **categorical_accuracy**（多クラス分類の正解率）

　しかし今回は、モデルからの出力値が活性化関数の「tanh 関数」により **-1.0 ～ 1.0** で出力するようになっていることに注意してほしい。通常は、**0.0 ～ 1.0** であるため、既存の binary_accuracy では正常に正解率が取得できない。そこで今回は、独自の評価関数を作成した。具体的にはリスト 5-5 のようになる。このような Keras のカスタマイズ機能は、本稿の説明範囲を超えているので、詳細は説明しない。コード内のコメントを参考にしてほしい。

```
import tensorflow.keras.backend as K

def tanh_accuracy(y_true, y_pred):          # y_true は正解、y_pred は予測（出力）
  threshold = K.cast(0.0, y_pred.dtype)             # -1 か 1 かを分ける閾値を作成
  y_pred = K.cast(y_pred >= threshold, y_pred.dtype) # 閾値未満で 0、以上で 1 に変換
  # 2 倍して -1.0 することで、0 ／ 1 を -1.0 ／ 1.0 にスケール変換して正解率を計算
  return K.mean(K.equal(y_true, y_pred * 2 - 1.0), axis=-1)
```

リスト 5-5　正解率（精度）のカスタム指標

それではモデルを生成する。これはリスト 5-6 のようなコードになる。

```
# 定数（学習方法設計時に必要となるもの）
LOSS = 'mean_squared_error'         # 損失関数：平均二乗誤差
OPTIMIZER = tf.keras.optimizers.SGD  # 最適化：確率的勾配降下法
LEARNING_RATE = 0.03                # 学習率：0.03

# モデルを生成する
model.compile(optimizer=OPTIMIZER(learning_rate=LEARNING_RATE),
              loss=LOSS,
              metrics=[tanh_accuracy])  # 精度（正解率）
```

リスト 5-6　モデルの生成

モデルの生成は、model オブジェクトの compile メソッドで行える。このメソッドは、

```
compile(
    optimizer,
    loss,
    metrics=None
)
```

と定義されており（※不要な引数は説明を割愛）、各引数の意味は以下の通りである。

- 第 1 引数の optimizer：最適化アルゴリズムを指定する。今回は、先ほどの定数 OPTIMIZER を指定すればよい
  - ・OPTIMIZER 定数の中身である SGD クラスは、コンストラクターを呼び出してインスタンス化する必要がある。そのコンストラクターの引数には、learning_rate というキーワードで学習率が指定できる。今回は、先ほどの定数 LEARNING_RATE を指定すればよい。指定しない場合は、既定値の **0.01** が使われる
- 第 2 引数の loss：損失関数を指定する。これも、先ほどの定数 LOSS を指定すればよい
- 第 3 引数の metrics：評価関数を指定できる（※評価については後述する）。分類問題では、典型的には 'accuracy' と文字列で評価関数を指定すればよい。ただし今回は自作の tanh_accuracy 関数を指定している。なお、評価関数は複数指定できるので、リスト値にして指定する必要がある

以上でモデルの生成は完了だ。

（6）学習：トレーニング

それでは、いよいよ学習を行っていこう。

Playground による図解

Playground を開いて、上部にある（6）の［実行／停止］ボタンをクリックするだけで、学習が始まる（図 6-1）。

図 6-1 ［実行／停止］ボタンをクリック

バックプロパゲーション（誤差逆伝播法）

学習（＝最適化）は、出力層から隠れ層へと、ニューラルネットワークの各層を逆順に進みながら、徐々に各ニューロンにおける各接続線の重みやバイアスを更新していく。前回説明したフォワードプロパゲーション（順伝播）とは逆の流れである。逆に伝播していくことから、**バックプロパゲーション（Back-propagation：Backprop、Backward propagation、逆伝播、誤差逆伝播法）**と呼ばれる。

バックプロパゲーションは、ニューラルネットワークの核となる機能なので、本来であれば、数式を理解し、実際にフルスクラッチでコードを書いて経験してみるのが一番である。しかしこれには、前述の偏微分と線形代数の理解が欠かせない。バックプロパゲーションのフルスクラッチ実装については、今後の連載企画に委ねるとして、今回は上記の機能概要紹介にとどめる。

エポック単位でのステップ学習

ニューラルネットワークでの学習単位は**エポック**と呼ばれる。**1 エポック**で全ての訓練データを **1 回**学習したことになる。

Playground で実行すると、永遠に学習してしまうため、エポックのメーターが回り続けることになる。しかし、1 エポックの学習ごとにどのような変化があるかを見たいときもあるだろう。そのようなときは、学習を停止させてから［実行／停止］ボタンの右にある［ステップ］ボタンをクリックすればよい（図 6-2）。

図 6-2 ［ステップ］ボタンをクリック

Python コードでの実装例

それでは、学習をコードで実装してみよう。そのコードを書く前に、前々回の復習として、データを取得して、訓練用／精度検証用にデータ分割するコードを再掲しておこう。

```
# （必要に応じて）座標点データを生成するライブラリをインストールする必要がある
#!pip install playground-data

# playground-data ライブラリの plygdata パッケージを「pg」という別名でインポート
import plygdata as pg

# 問題種別で「分類（Classification）」を選択し、
# データ種別で「2 つのガウシアンデータ（TwoGaussData）」を選択する場合の、
# 設定値を定数として定義
PROBLEM_DATA_TYPE = pg.DatasetType.ClassifyTwoGaussData

# 各種設定を定数として定義
```

```
TRAINING_DATA_RATIO = 0.5  # データの何％を訓練【Training】用に？(残りは精度検証【Validation】用)：50%
DATA_NOISE = 0.0           # ノイズ：0%

# 定義済みの定数を引数に指定して、データを生成する
data_list = pg.generate_data(PROBLEM_DATA_TYPE, DATA_NOISE)

# データを「訓練用」と「精度検証用」を指定の比率で分割し、さらにそれぞれを「データ（X）」と「教師ラベル（y）」に分ける
X_train, y_train, X_valid, y_valid = pg.split_data(data_list, training_size=TRAINING_DATA_RATIO)
```

リスト 6-1　【前々回の復習】データの取得と分割

学習を行うコードは、リスト 6-2 のようになる。

```
# 定数（学習方法設計時に必要となるもの）
BATCH_SIZE = 1   # バッチサイズ：1（選択肢は「1」～「30」）
EPOCHS = 100     # エポック数：100

# 学習する（※まだ実行しないこと）
hist = model.fit(x=X_train,                    # 訓練用データ
                 y=y_train,                    # 訓練用ラベル
                 validation_data=(X_valid, y_valid), # 精度検証用
                 batch_size=BATCH_SIZE,        # バッチサイズ
                 epochs=EPOCHS,                # エポック数
                 verbose=1)                    # 実行状況表示
```

リスト 6-2　学習

最初に、バッチサイズとエポック数が定数として定義されている。バッチサイズはこの後で説明するが、Playground と同様に取りあえず **1** を指定する。エポック数は **100** が指定されているので、100 エポックの学習を行うことになる。

実際の学習は、model オブジェクトの fit メソッドで行える。このメソッドは、

```
fit(
    x=None,
    y=None,
    validation_data=None,
    batch_size=None,
    epochs=1,
    verbose=1
)
```

と定義されており（※不要な引数は説明を割愛）、各引数の意味は以下の通りである。

- 第 1 引数の x：訓練用データを指定する。今回は、先ほど作成した X_train（＝訓練データの座標点）を指定
- 第 2 引数の y：訓練用ラベルを指定する。今回は、先ほど作成した y_train（＝訓練データの教師ラベル）を指定
- 第 3 引数の validation_data：精度検証データとそのラベルをタプルで指定する。今回は、先ほど作成した X_valid と y_valid をタプルで指定
- 第 4 引数の batch_size：バッチサイズを指定する。今回は、定数として定義した BATCH_SIZE を指定
- 第 5 引数の epochs：エポック数を指定する。今回は、定数として定義した EPOCHS を指定
- 第 6 引数の verbose：進行状況や結果のログを標準出力する。**1** を指定するとプログレスバー付きで表示され、**2** を指定するとエポックごとにより詳しく表示される。**0** を指定すると出力されない。今回は **1** を指定する

また、fit メソッドは History オブジェクト（この例では hist）を戻り値として返す。History オブジェクトの history 属性には、トレーニング（学習）時の各エポックでの損失値や評価関数値が記録されており、さらに上記のように精度検証データも指定された場合はその損失値や評価関数値も記録されている。この記録の見方は、後述の「評価」で説明する。

以上で学習のコードも完了だ。ここで実行すると、学習が実際に行われるのだが、まだ実行しないでほしい（※実行すると、その結果が保存されてしまう。この状態で、次の「リスト 6-3　ミニバッチ学習」を実行すると、学習済みモデルに追加で学習することになってしまうため）。次に、損失や正解率を見ていくが、その前に、説明をスキップしていたバッチサイズについて言及しておこう。

（6）学習：バッチサイズ

Playground による図解

あらためて示すと、バッチサイズは図 6-3 のように設定できる。

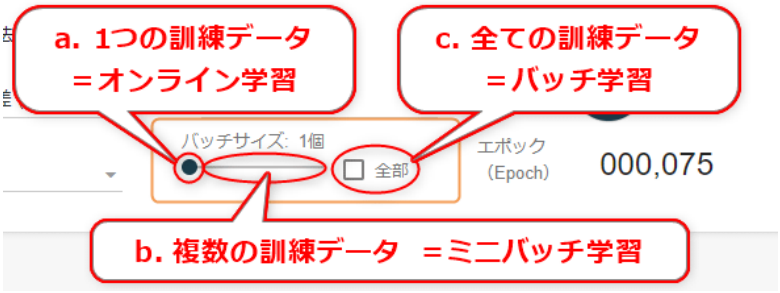

図 6-3　バッチサイズを設定

前述の「勾配法」の節でも簡単に示したが、**バッチサイズ（Batch size）**とは、「どのくらいの数量（＝サイズ）の訓練データをまとめて学習するか」を示す数値で、

a. 1 つの訓練データ：**オンライン学習（Online training ）**
b. 複数の訓練データ：**ミニバッチ学習（Mini-batch training ）**
c. 全ての訓練データ：**バッチ学習（Batch training ）**

という 3 パターンに大別できる。オンラインとは、文字通り、連続的につながっているデータを 1 つずつ処理して重みとバイアスを更新していくことを意味する。バッチとは、文字通り、一群の全データをまとめて処理して一気に重みとバイアスを更新していくこと。そしてミニバッチとは、その中間で、ある程度の小さいまとまりごとに処理して、その単位で重みとバイアスを更新していくことを意味する。

それぞれの代表的な勾配法にも以下のような違いがある。

a. オンライン学習＝**確率的勾配降下法（SGD：Stochastic Gradient Descent）**
b. ミニバッチ学習＝**ミニバッチ勾配降下法（Mini-batch Gradient Descent**、上記と同じく **SGD** とも呼ぶ）
c. バッチ学習＝**最急降下法（Steepest Descent、バッチ勾配降下法：Batch Gradient Descent）**

勾配法の名前を見ると、a と b は SGD（Stochastic Gradient Descent）で同じものと見ることができる。一方、c は「**勾配降下法**」（**GD：Gradient Descent**）とも呼ばれるが、「Stochastic」（確率的）が付いていない。「確率的」とは、データをランダムサンプリング（ランダムに抽出）して、勾配を計算して重みとバイアスを更新することを指す。本来であれば、全データを使って厳密に学習してから、重みとバイアスを更新するのが妥当だと考えられるだろう。しかし、確率論に基づけば、そのような部分的なデータを使っても、重みとバイアスを更新していけるというわけだ。

参考までに、［バッチサイズ］に **1** 個／ **15** 個／ **30** 個／**全部**（＝ 250 個）を指定した場合の学習結果の違いを、図 6-4 にまとめた。

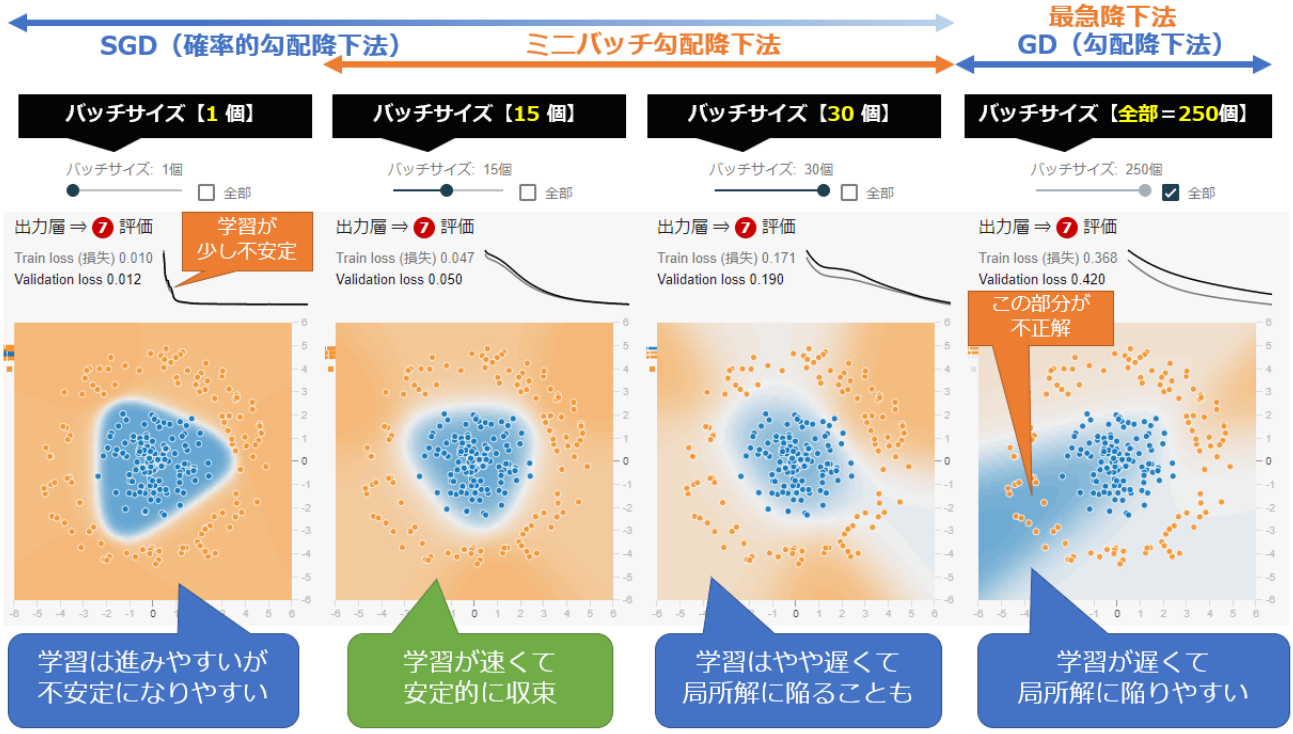

図 6-4　バッチサイズの違いの比較（1 個／ 15 個／ 30 個／全部= 250 個の場合）

※いずれも 100 エポック、学習した結果の比較である。

学習率に応じて最適なバッチサイズも変わるため一概には言えないが、学習率 **0.03** という状況下における、図 6-4 に示す各バッチサイズによる影響／効果を以下にまとめた。バッチサイズを決める上で参考にしてほしい。

まず、左端の **1** 個では、全て個別に（この例ではデータ 1 個 ×250 イテレーション）学習していくので、1 エポックが完了するスピードはやや遅いが、学習は進みやすい。実際、この例では 100 エポックで損失（loss）は **0.01** となっており、ほぼ学習が終わっている。その半面、損失の下降線を描いているグラフを見ると一部がジグザグと震えているのが分かるが、このように、1 つのデータに学習が揺さぶられて不安定になりやすい欠点もある。

次の **15** 個は、1 エポック（この例ではデータ 15 個 ×16 イテレーション）の学習スピードが速く、しかも一直線に安定的に学習できている。この中では最良である。

1 つ飛ばして右端の**全部**（＝ 250 個）は、全データを使うので今度は計算量が多くなって、1 エポック（この例ではデータ 250 個 ×1 イテレーション）が完了するスピードは遅くなり、しかも学習もなかなか進まない。バッチ学習（最急降下法）は、局所解に陥りやすいという欠点もあるので、多くのケースで SGD の方が好ましい。実際に、図 6-4 をこのまま学習を続けても、左下にあるオレンジ色の丸い点の背景色は青色（＝不正解）のままになり、損失が減らなくなった。これは局所解に陥っている状態である。

それを踏まえて、1 つ前の **30** 個を見ると、局所解を何とか避けられており、このまま学習を続けると損失は **0** に近づく。何度か実行を試すと局所解に陥ることもあったので、バッチサイズが大きすぎる場合には局所解への注意が必要だ。1 エポック（この例ではデータ 30 個 ×8 イテレーション）が完了するスピードは、データが小分けになって計算量が減るのでバッチ学習よりも速いが、バッチサイズ **15** 個よりは計算量が比較的多いのでやや遅い。

Python コードでの実装例

バッチサイズの設定方法は前述のリスト 6-2 で説明済みである。具体的にはリスト 6-3 のように、定数 BATCH_SIZE に「**15** 個」を指定するだけだ。もし全データ（最急降下法）を使いたい場合は、「**None**」を指定すればよい。

```
# 定数（学習方法設計時に必要となるもの）
BATCH_SIZE = 15  # バッチサイズ：15（選択肢は「1」～「30」）
EPOCHS = 100     # エポック数：100

# 学習する
hist = model.fit(x=X_train,                    # 訓練用データ
                 y=y_train,                    # 訓練用ラベル
                 validation_data=(X_valid, y_valid), # 精度検証用
                 batch_size=BATCH_SIZE,        # バッチサイズ
                 epochs=EPOCHS,                # エポック数
                 verbose=1)                    # 実行状況表示
```

リスト 6-3　ミニバッチ学習

以上で学習の説明は終わりである。ここで実際にリスト 6-3 のコードを実行してみよう。次に、その学習の結果について見ていこう。

（7）評価：損失のグラフ

それでは、学習結果を評価してみよう。

Playground による図解

既に何度か説明で触れているが、あらためて説明する。Playground を開いて、右側にある（7）評価の［Train loss］（＝訓練データにおける損失）／［Validation loss］（＝精度検証データにおける損失）と、その右にあるグラフを参照してほしい（図 7-1）。グラフの下には最終的な精度（正解率：acc = accuracy）も表示されている。

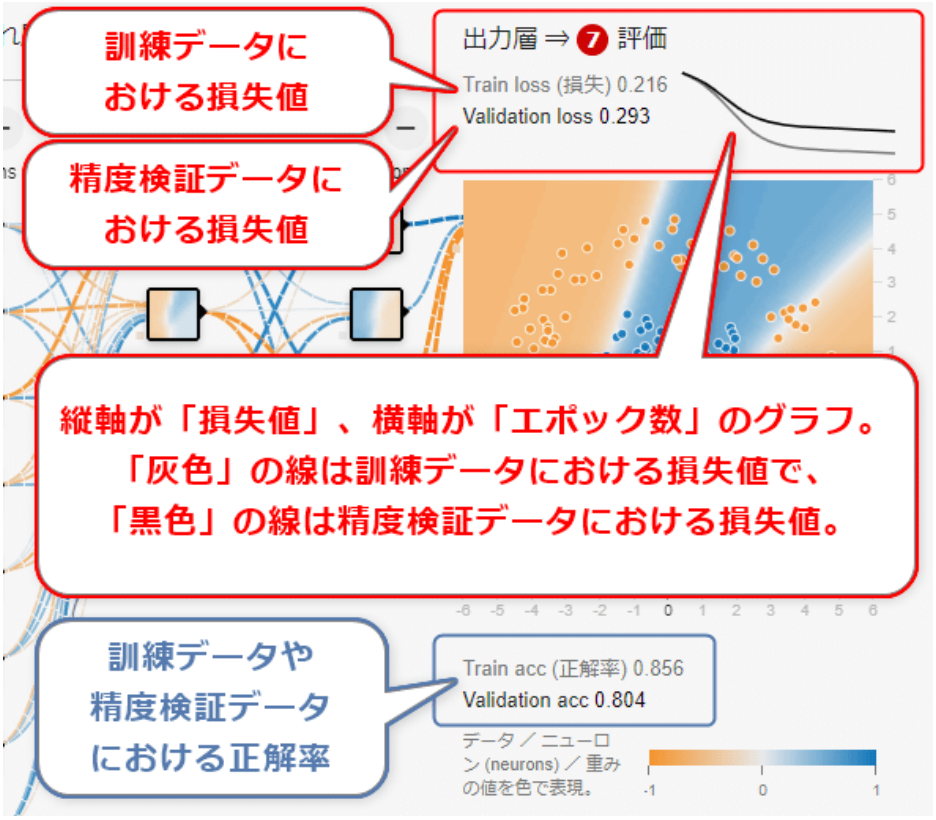

図 7-1　評価の損失値とグラフ

説明しなくても理解できると思うが、念のため、説明する。訓練データにより学習を行った結果の損失値が、灰色の文字で（この例では **0.216** と）表示され、その右に灰色の線がグラフ描画されている。エポック数が増えるに従って、損失値が低くなり、**0** に向かって収束していっているのが分かる。

一方、精度検証データを学習済みモデルに入力して計測（※学習はしておらず精度検証用の計測のみ）した損失値が、黒色の文字で（この例では **0.293** と）表示され、その右に黒色の線がグラフ描画されている。こちらも、訓練データほどではないが、**0** に向かって収束していっているのが分かる。

どちらも **0.001** などと損失ができるだけ小さくなるまで学習を続けた方がいいが、途中で損失値のグラフが横ばいになり、それ以上、損失値が減らなくなってくる。こうなれば学習を終えてよい。

Playground では目視でグラフを確認して止められるが、コードの場合は指定したエポック数まで学習が継続されてしまう。「損失値がほとんど変わらない状態になったら、早めに学習を打ち切りたい」というニーズはあるだろう。このようなニーズに応える、**早期終了（Early Stopping、早期停止）**と呼ばれる機能が、Keras には搭載されている。早期終了は、Playground には実装していないので、Playground では使えない。よって後述のコード実装で、早期終了の使い方を説明する。

Python コードでの実装例

それでは、損失値の表示や、その推移グラフの描画を、コード実装により実現してみよう。

損失の値については、実は先ほどのリスト 6-2 やリスト 6-3 のコードを実行すると、fit メソッドの第 6 引数 verbose=1 の指定により、自動的に表示される。図 7-2 がその表示例である。

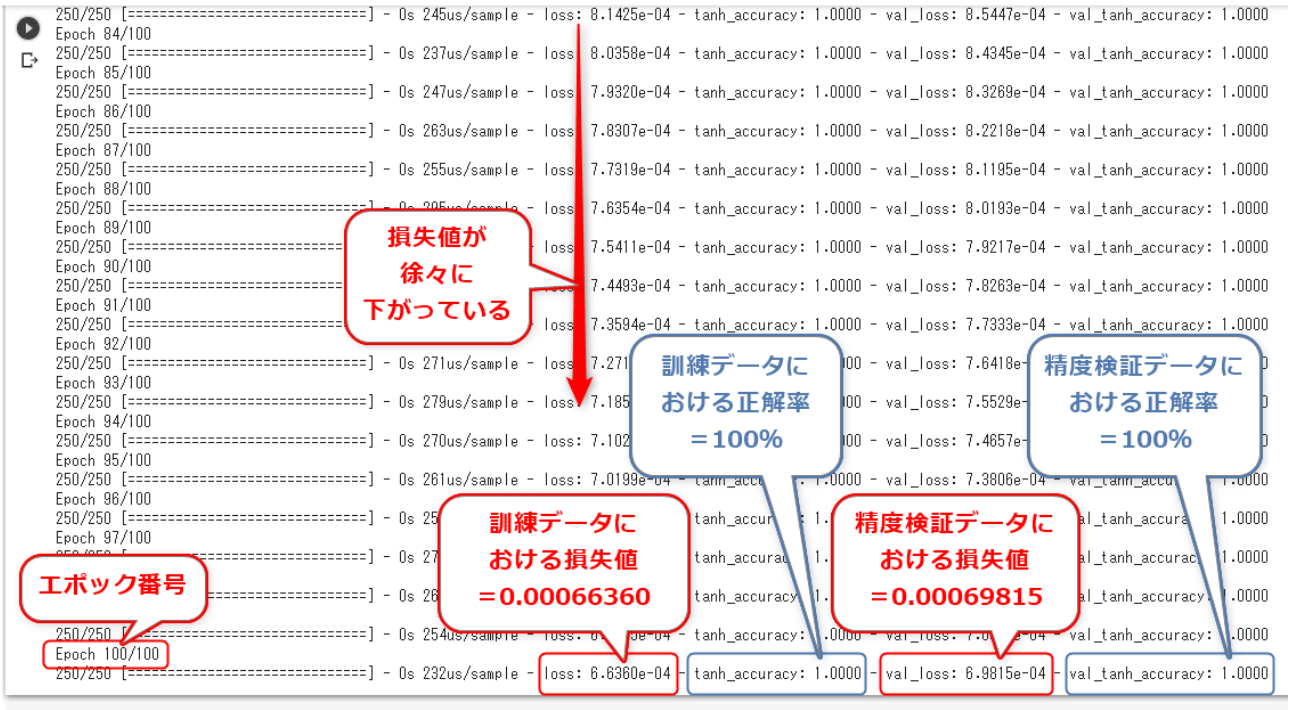

図 7-2　損失値の表示例

エポックが 1 回完了するごとに 1 行ずつログが標準出力される。エポック数 **100** の最後である **100** 番目の行を見ると、[loss] は **0.00066360**（＝ **6.6360e-04**、「e-04」は指数表記で 1/10 の 4 乗を意味する）となっており、訓練データにおける損失値はかなり小さい。

ついでに正解率も見てみよう。先ほど tanh_accuracy という精度（正解率）を測る評価指標を指定したので、[tanh_accuracy] には正解率が表示されている。この例では、**1.0000** と **100%**正解している。

この例では、訓練データにおける学習は成功している。念のため、精度検証データにおける損失と正解率も見ておこう。[val_loss] が精度検証データにおける損失値で、[val_tanh_accuracy] が正解率である。こちらも問題ない値となっていることが確認できる。

数字ではなく、グラフで視覚的に学習結果を把握したい場合は、fit メソッドの戻り値で返された History オブジェクト（本稿の例では hist）のログデータを使って、グラフ描画ライブラリ「Matplotlib」でグラフを描画するとよい。コードの書き方の説明は割愛するが、リスト 7-1 のようなコードを書けばよい。

```
import matplotlib.pyplot as plt

# 学習結果（損失）のグラフを描画
train_loss = hist.history['loss']
valid_loss = hist.history['val_loss']
epochs = len(train_loss)
plt.plot(range(epochs), train_loss, marker='.', label='loss (Training data)')
plt.plot(range(epochs), valid_loss, marker='.', label='loss (validation data)')
plt.legend(loc='best')
plt.grid()
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

リスト 7-1　損失値の推移グラフ描画

このコードを実行すると、図 7-3 のように描画される。

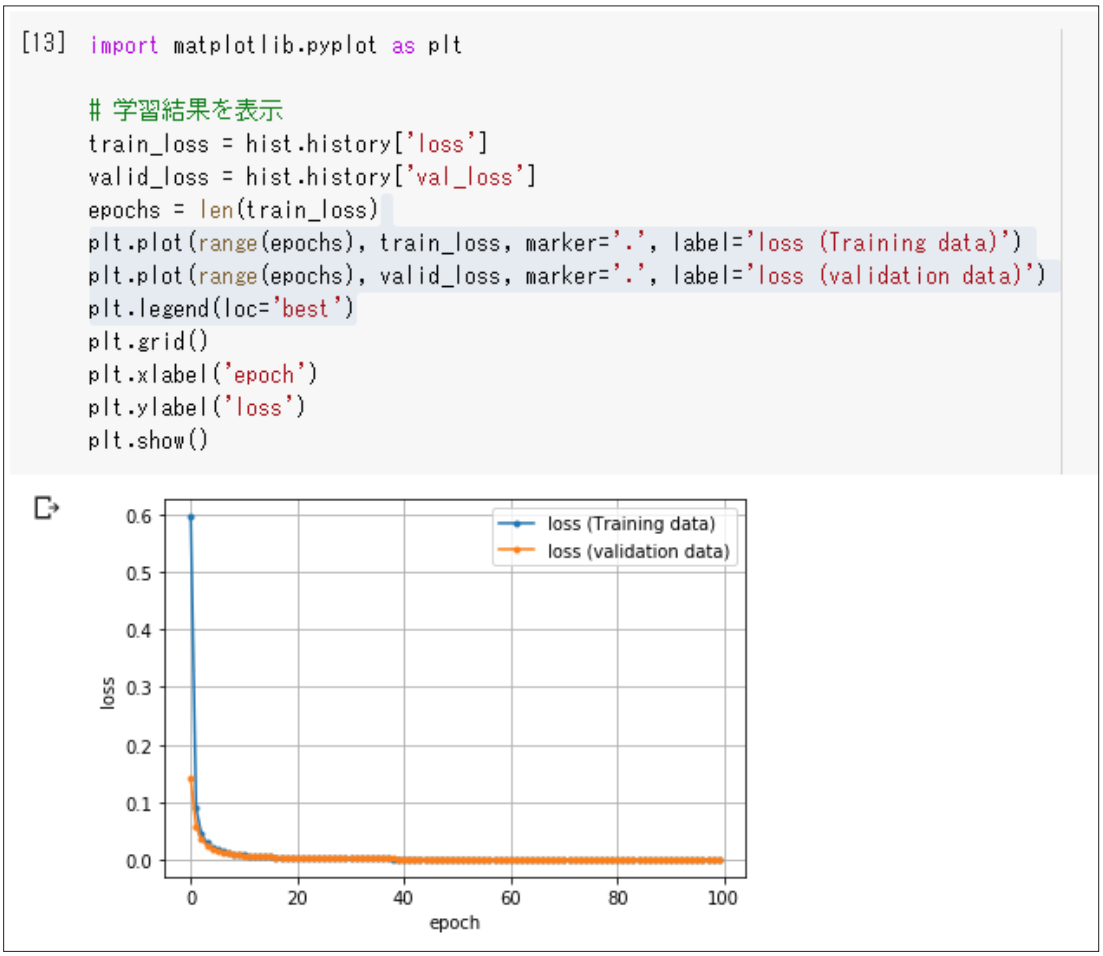

図 7-3　損失値推移グラフの描画例

→目次に戻る

正解率のグラフも同様に描画できるが、説明を割愛する。

早期終了と CSV ログ出力

上記のグラフでは、最後まで損失の減少が続いているので必要ないが、損失の減少が停滞してきたらそれ以上、学習する必要はない。このようなムダな学習を省くために、前述の通り、早期終了機能が活用できる。

Keras では、早期終了は「コールバック」という機能で実現できる。**コールバック（Callback）**とは、学習中（＝トレーニング中）のモデル内部から、何らかの機能を呼び出してもらう機構である。早期終了の場合は、（tf.keras.callbacks モジュール階層の）EarlyStopping クラスのインスタンスを、fit メソッドの引数 callbacks に、リスト値として指定すればよい。

また、コールバックには、他にも機能があり、例えば実行ログを CSV ファイルとして保存したりすることもできる。その CSV ロガーは、（tf.keras.callbacks モジュール階層の）CSVLogger クラスのインスタンスを、同様に引数 callbacks に指定すればよい。

この 2 つのコールバックを指定して学習するコードはリスト 7-2 のようになる。

```
# 早期終了
es = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=2)

# CSV ロガー
csv_logger = tf.keras.callbacks.CSVLogger('training.log')

# 学習する
hist = model.fit(x=X_train,                    # 訓練用データ
                 y=y_train,                    # 訓練用ラベル
                 validation_data=(X_valid, y_valid), # 精度検証用
                 batch_size=BATCH_SIZE,        # バッチサイズ
                 epochs=EPOCHS,                # エポック数
                 verbose=1,                    # 実行状況表示
                 callbacks=[es, csv_logger])   # コールバック
```

リスト 7-2　コールバックの指定（早期終了と CSV ログ出力）

EarlyStopping クラスのコンストラクターでは、引数 monitor に監視対象の損失（この例では val_loss ＝精度検証データの損失）を指定し、引数 patience には何エポック連続で数値に減少が見られないと学習を打ち切るかの数値（この例では **2** 回）を指定する。ちなみに、今回のサンプルで実行してみたが、学習は早期停止しなかった。

CSVLogger クラスのコンストラクターでは、ファイル名（この例では「training.log」）を引数に指定する。実行後に生成されたファイル内容を確認してみたところ、図 7-4 の CSV テキストデータが出力されていた。

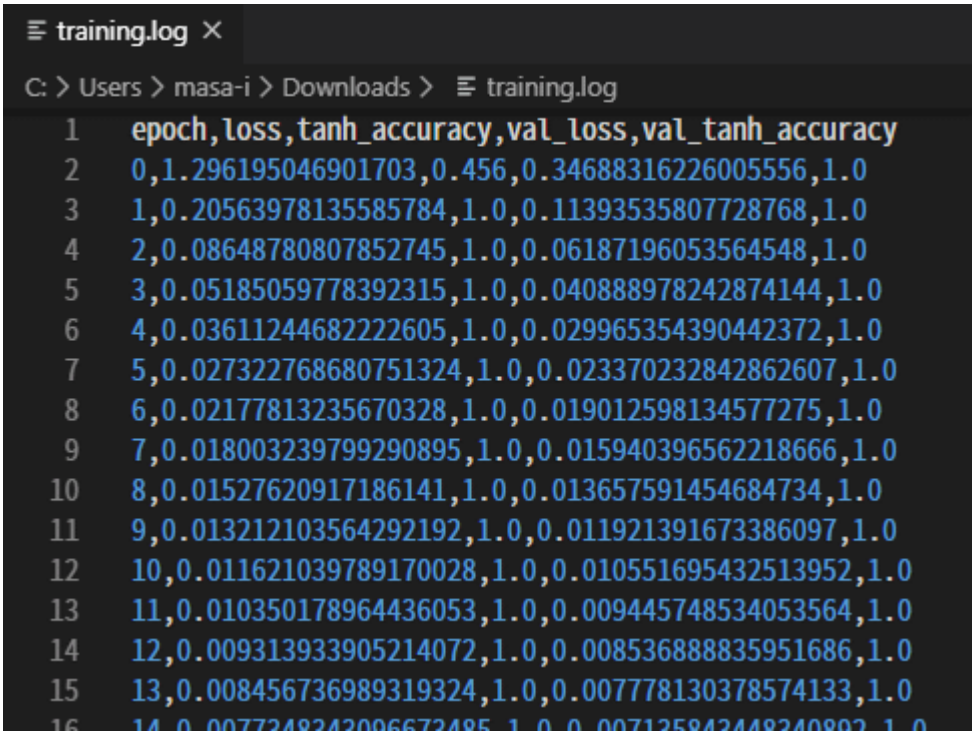

図 7-4　CSV ログ出力の例

→目次に戻る

（8）テスト：未知データで推論と評価

さて、ここまでで無事にモデルが学習できて、かなり小さい損失値で、分類の場合は十分な正解率が出せるようになったとしよう。「それなら、実運用に進んでよいか？」というと、「もう一段、テストをした方がよい」とされている。学習時に使った訓練データや、精度検証に用いた精度検証データには、何らかのバイアスがかかっている可能性が否定できないからだ。

Playground による図解

Playground では、右側にある出力層のグラフ上の任意の場所をクリックすることで、その地点のテストが行えるようになっている（図 8-1）。

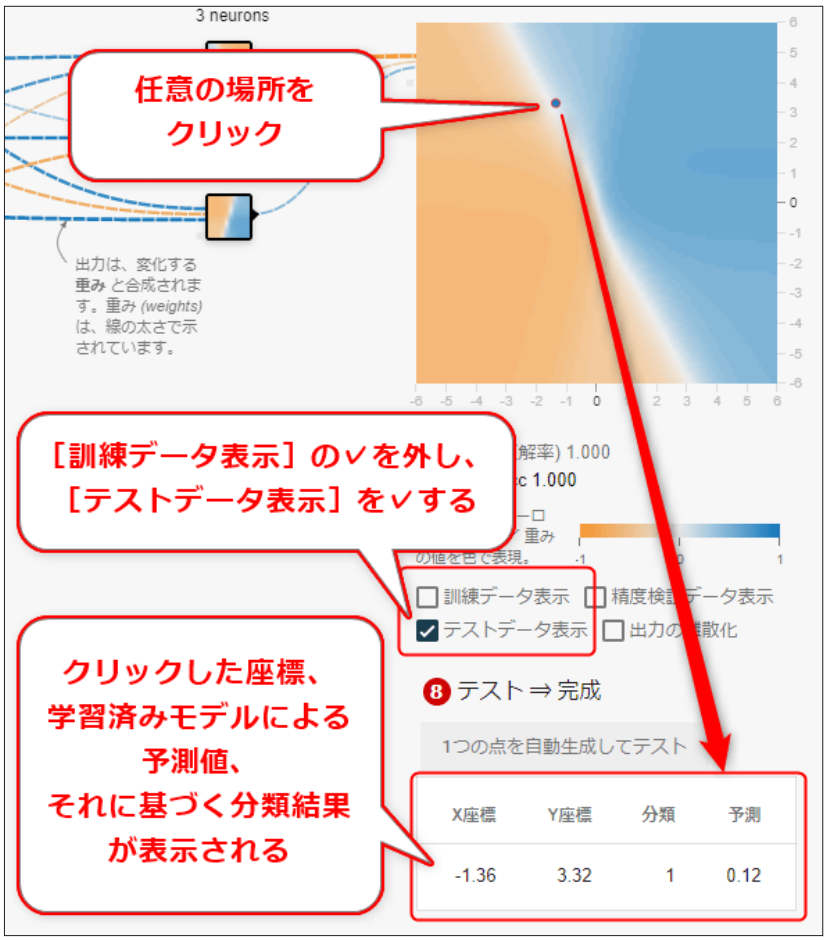

図 8-1　グラフ上の任意の場所をクリック

この例では座標（**-1.36、3.32**）をクリックしている。背景色が白い所をクリックしたので、予測値も **0.12** と **0.0** に近い数値となっている。分類問題では、**0.0 未満**がオレンジ色（**= -1**）で、**0.0 以上**が青色（**= 1**）にと、強引に振り分ける（＝**離散化：discretize** する）仕様なので、［分類］は **1**（＝青色）となっている。

ちなみに、Playground には［出力の離散化］というチェックボックスがあるが、ここにチェックを入れると、強引に **-1** か **1** に振り分けられるため、背景描画から白色の部分はなくなる。

Python コードでの実装例

ここまで、とても長く感じたかもしれないが、ついに最後のコードである。そのコードで、テストデータを新たに生成して入力として使い、学習済みモデルに結果出力（＝**予測：predict**）させたり、テストデータでの精度（＝「**汎化性能：Generalization performance**」と呼ぶ）に問題がないか**評価（evaluate）**したりしてみよう。

コードは、リスト 8-1 のようになる。なお、データの生成方法は、リスト 6-1 に掲載したものと同じなので、説明を割愛する。

```
import plygdata as pg
import numpy as np

# 未知のテストデータを生成
PROBLEM_DATA_TYPE = pg.DatasetType.ClassifyTwoGaussData
TEST_DATA_RATIO = 1.0  # データの何%を訓練【Training】用に？( 残りは精度検証【Validation】用 )：100%
DATA_NOISE = 0.0      # ノイズ：0%
data_list = pg.generate_data(PROBLEM_DATA_TYPE, DATA_NOISE)
X_test, y_test, _, _ = pg.split_data(data_list, training_size=TEST_DATA_RATIO)

# 学習済みモデルを使って推論
result_proba = model.predict(X_test)
result_class = np.frompyfunc(lambda x: 1 if x >= 0.0 else -1, 1, 1)(result_proba) # 離散化
# それぞれ 5 件ずつ出力
print('proba:'); print(result_proba[:5])  # 予測
print('class:'); print(result_class[:5])  # 分類

# 未知のテストデータで学習済みモデルの汎化性能を評価
score = model.evaluate(X_test, y_test)
print('test loss:', score[0])  # 損失
print('test acc:', score[1])   # 正解率
```

リスト 8-1　未知データによるテスト（推論と評価）

学習済みモデルによる推論／予測は、model オブジェクトの predict メソッドで行える。このメソッドの引数には、多次元配列値のテストデータ（この例では）を渡せばよい。戻り値として、出力結果（つまり予測値）が多次元配列値で返される。

また、最終的な損失は、model オブジェクトの evaluate メソッドで取得できる。このメソッドの引数には、多次元配列のテストデータと教師ラベルを渡せばよい。戻り値として、損失値が返される。本稿のように評価指標も指定した場合は、その評価（今回は正解率）も合わせて、リスト値で返される。

実際にリスト 8-1 を実行した結果、図 8-2 のようになった。

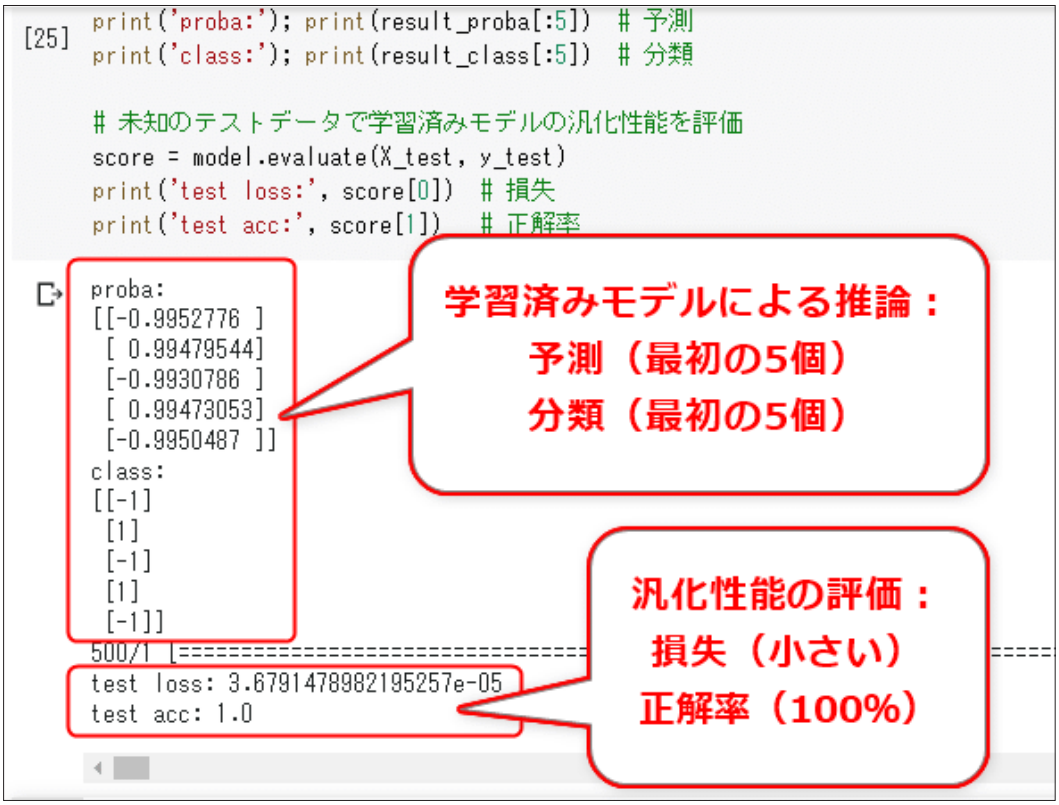

図 8-2　推論と評価の結果

図中の説明を読めば分かるように、推論／予測が正常に実行できており、汎化性能も十分あるという結果が出た。

まとめ

以上でニューラルネットワークの基本を一通り学んだことになる。今回の知識を武器に、ディープラーニングへの理解を深めていって、ゆくゆくは機械学習全般に強くなっていってほしい。

最後におまけとして、Colaboratory 上のサンプル（tf2-keras-neuralnetwork.ipynb）には、ニューラルネットワーク内の重みやバイアスを調べるためのサンプルコードを追記しておいた。気になる人はぜひ、リンク先を開いてコード内容を確かめてみてほしい。

編集：@IT 編集部
発行：アイティメディア株式会社