@IT
atmarkIT

作って試そう！ ディープラーニング工作室

かわさきしんじ, Deep Insider編集部[著]

機械学習やディープラーニングってどんなもの？

実際にコードを書きながら「人工知能／機械学習／ディープラーニング」を学んでいこう。まずはその概要とそのために便利に使える Google Colab を紹介。

（2020 年 03 月 24 日）

人工知能？　機械学習？　ディープラーニング？

このところ、ニュースなどで「人工知能（AI）」「機械学習」「ディープラーニング」という言葉を聞かない日はありません。これらの用語には「人が作った知能」「人の脳神経を模したヤツね」「ニューラルネットワークみたいな？」などといったイメージがあると思いますが、具体的にはこうした技術を使ってどんなことができるのでしょうか。パッと思い付くのは次のようなことです。

- 画像認識／画像合成
- 音声認識／音声合成
- 自然言語処理（翻訳、会話など）
- ゲーム（囲碁、将棋、TV ゲームなど）
- 推測（株価、レコメンドなど）
- 制御（自動運転、ロボットなど）

これらは「何かを基に何らかの判断を行う」こととまとめられます。例えば、「桜の花を見て、それを桜と認識する」「誰かから話しかけられて、その意味を理解する」「左カーブに合わせて、ステアリングを操作する」といったことです。

人が行うのであれば、これらは「知的な活動」とでも呼べるでしょう。そうした知的な活動を人工的な存在であるソフトウェア（と、そこに何らかの形で接続されたハードウェア、そこから取得できるデータなど）を使って模倣するものが人工知能だといえます。

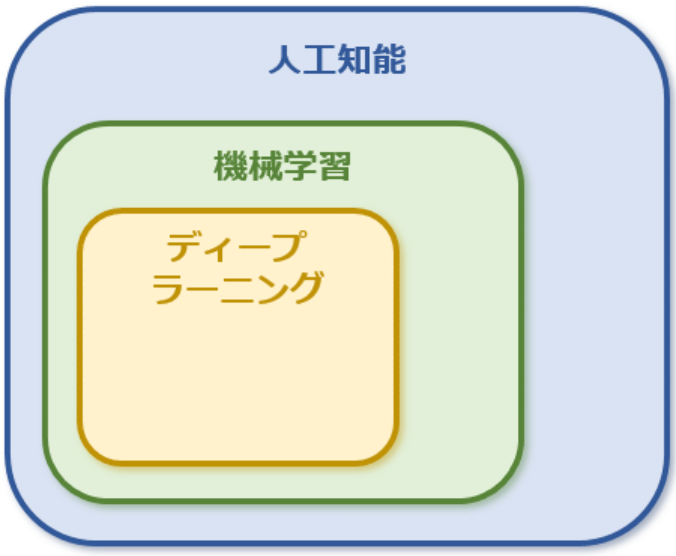
人工知能は 1950 年代にその概念が登場し、その後、紆余（うよ）曲折を経る中で、春の時代や冬の時代を何度かくぐり抜けて現在に至っています。そして、現在ではコンピューターの性能向上やソフトウェア／理論の発展に後押しされ、ある程度は現実的なものとして社会に受け入れられるようになりました（現在、広く使われている人工知能は、特定の問題を解決可能な「弱い AI」と呼ばれるものです）。

「機械学習」（Machine Learning、ML）とは、今述べた人工知能を実現するための手段の一つです。その名前から分かる通り、「機械」（コンピューター）に「学習」をさせることで、知性の獲得を目指します。このときには、関連性を持つ大量のデータを機械に入力し、それらから今解決しようとしている問題を解くために必要な何らかのパターンやルールのようなものを見つけ出します（知性の獲得）。人が何かを身に付けようとしたら、何度も同じことを繰り返して、身体の動かし方を体得したり、「ネコの特徴は○○に現れているのだな」と理解したり、「この部品はそろそろ壊れるかもしれないな」と推測したりと、自分が身に付けたい何かに関することを（知らず知らずのうちに、あるいは意識的に）学んでいくことに似ています。

機械に学習をさせるとして、その方法にもさまざまなものがあります。その一つが、人間の脳を模倣した「ニューラルネットワーク」です。「ディープラーニング」とは、このニューラルネットワークを進化させたものです。これについては後で少し詳しく見てみます。

ここまでの話を簡単にまとめましょう。より詳しい説明は本フォーラムの連載記事「機械学習&ディープラーニング入門（概要編）」などを参考にしてください。

- 人工知能：人間が日々行っている知的な活動を人工的に実現すること
- 機械学習：人工知能の種類の一つで、大量のデータを基にコンピューターに「学習」をさせることで、人の知的活動を実現すること
- ディープラーニング：機械学習を行う方法の一つで「ニューラルネットワーク」と呼ばれる手法を強力にしたもの

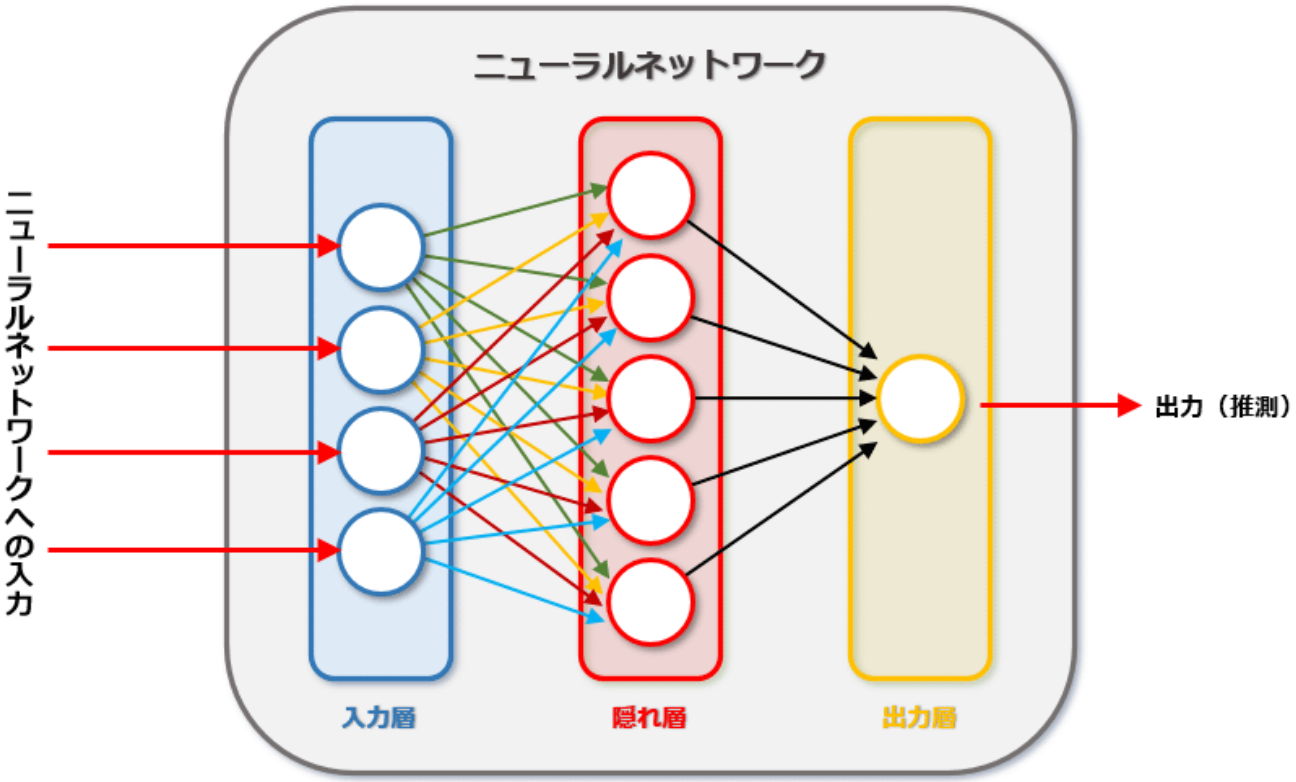


人工知能／機械学習／ディープラーニングの関係

機械学習で使われる他の手法については、機会があれば、連載の中で取り上げる予定です。

ニューラルネットワーク／ディープラーニングとは

本連載で主に取り上げるのは「ニューラルネットワーク」や「ディープラーニング」と呼ばれる手法です。一般に、ニューラルネットワークは入力層、隠れ層、出力層の 3 つの層で構成されますが、ディープラーニングで使われるディープニューラルネットワークでは、隠れ層を複数にする（隠れ層を深く＝ディープにする）ことで入力に対する出力の精度をより向上させたものと考えられます（なお、以下ではこれらのネットワークのことを「ニューラルネットワーク」あるいは単に「ネットワーク」と表記します）。



ニューラルネットワークの例

上に示した図にある一つ一つの○（これを「ノード」と呼びます）とそこから他の○へと伸びている線（これを「エッジ」と呼びます）が、脳の神経細胞（ニューロン）を模したものです。それらが見ての通り、ネットワークを形成していることから、「ニューロンのネットワーク」＝「ニューラルネットワーク」と呼ばれます（余談ですが、脳神経の世界では、ニューロンからニューロンへは軸索――上の図におけるエッジ――を介して電気信号として情報が伝えられます。このとき、ニューロンの軸索を流れる信号は、伝達先のニューロンの「樹状突起」と呼ばれる部分（ノードにある、エッジと接する箇所）へと伝わります。そして、軸索と樹状突起が接触している部分のことを「シナプス」と呼びます。図の矢印が該当するといえます）。

上の図では一番左側が入力層、一番右側が出力層となり、その間に隠れ層が 1 つあります。そして、一番左の入力層に何かしらのデータを入れてやることで、このネットワークは計算を始め、一番右の出力層から何らかの回答を弾き出します。この過程のことを「推測」「推論」などと、その回答のことを「推測結果」「推論結果」などと呼ぶこともあります。

→目次に戻る

重要なのはそれぞれのニューラルネットワークが処理できるデータは、「どういう学習を行ったか」によって決定される点です。音声認識用に学習を行ったニューラルネットワークに対して、画像データを送り込んでも、恐らくそれは適切な答えを計算できません。

現状ではニューラルネットワークで解決したい課題があったとしたら、そのために必要なデータを用意して、それをコンピューターが学習しやすい形に整形し、それを使ってニューラルネットワークに学習をさせて、問題解決に役立つようにしなければなりません。

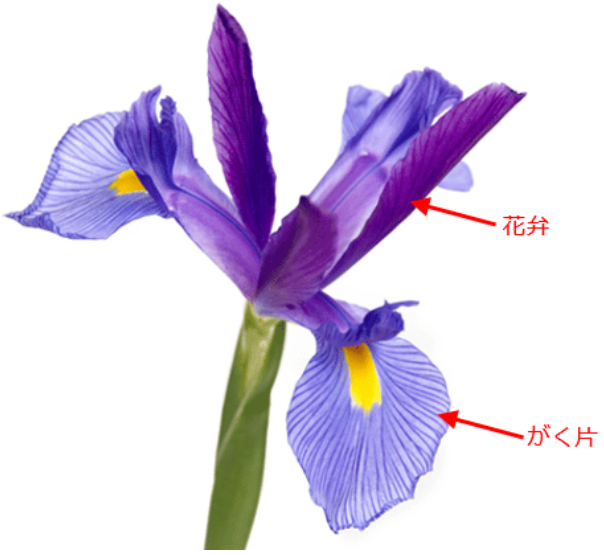
その一方で、多くの企業（Microsoft や Google など）が学習済みの高精度なニューラルネットワークを使った人工知能サービスも提供しています。それらを活用すれば、例えば画像認識のような一般的な処理であれば、自力でそのためのニューラルネットワークを作る必要がないこともあるでしょう（もちろん、そうはいかないこともあります）。

本連載では、そうした人工知能サービスの足下にも及ばないものでありますが、簡素なニューラルネットワークを自分で作りながら、人工知能／機械学習／ディープラーニングに関連する概念、基礎知識、用語などを理解することを目的としています。

ニューラルネットワークにおける学習とは

ところで、先ほどは「ニューラルネットワークは入力層、1 つ以上の隠れ層、出力層で構成される」という話をしました。しかし、入力層に何かのデータを入れたら、そこから何らかの結果が適切に得られる理由は何でしょう。

例えば、あやめの花弁の長さと幅、がく片の長さと幅という 4 つのデータを基に、あやめの種類を分類したデータがあったとします（そうしたサンプルデータは実際にあり、次回で紹介します）。

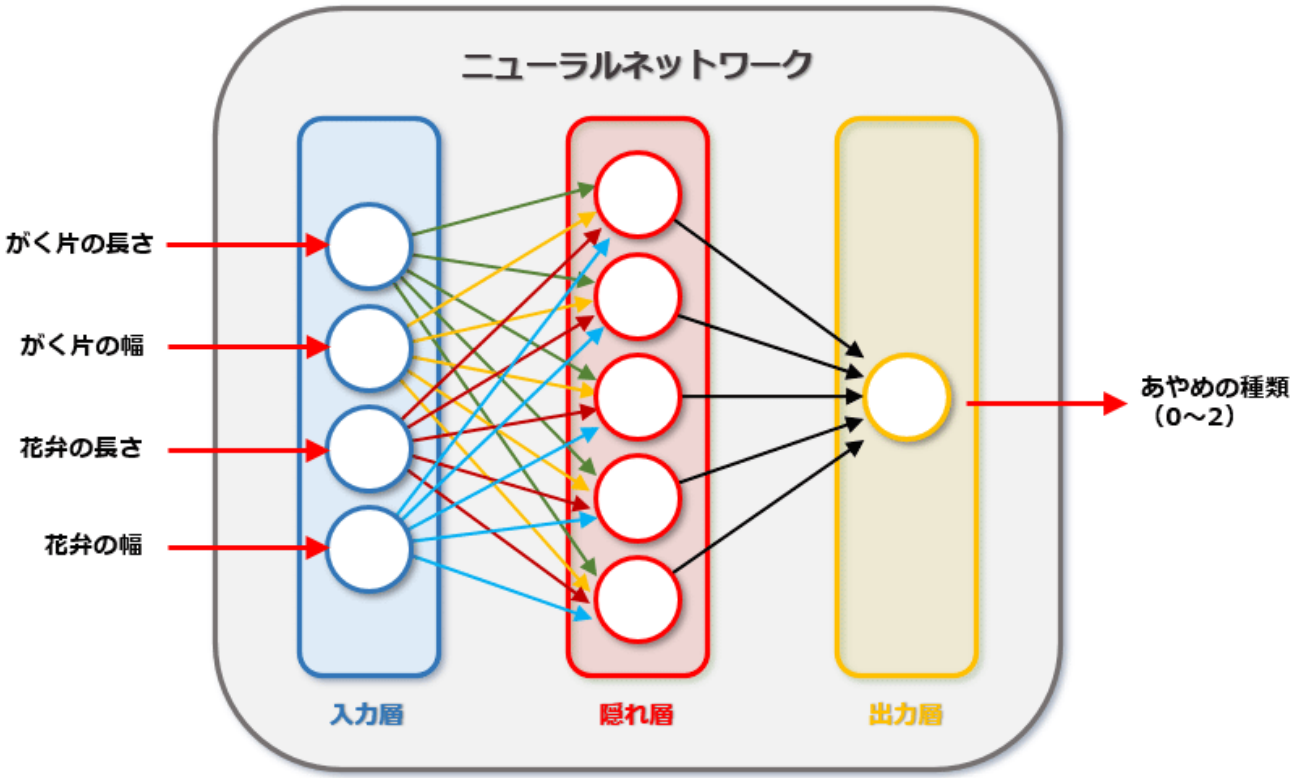


あやめの花の構造

→目次に戻る

そして、それらの値を基にあやめの種類を「推測」するようなニューラルネットワークが欲しくなったとしましょう。入力層に「がく片の長さ , がく片の幅 , 花弁の長さ , 花弁の幅」という 4 つのデータを与えると、あやめの種類（setosa、versicolor、virginica）のどれであるかを計算してくれるということです。ここであやめの種類には 0（setosa）、1（versicolor）、2（virginica）という番号を振っておきます。

すると、ここで求めたいニューラルネットワークは次のようなものになります（先ほどの図と同じ構造なのは、もともと、そうなることを想定して書いたからです）。



あやめの種類を推測するニューラルネットワーク

そして、このニューラルネットワークに十分に「学習」をさせれば、「がく片の長さ、がく片の幅、花弁の長さ、花弁の幅」として「5.7, 2.8, 4.1, 1.3」というデータを入力すると「1」（versicolor）であると、「4.7, 3.2, 1.3, 0.2」を入力すれば「0」（setosa）であると答えてくれるわけです。では、なぜこうした計算（推測）が行えるのでしょう。

脳神経の世界では、ニューロン間の結合（シナプスの結合）にはそれぞれに強度があります。これはあるニューロンから別のニューロンへ信号が伝達される際の影響の大きさを表すものです。加えて、ニューロンからニューロンへと信号を伝達するかどうかも重要です。あるニューロンが受け取った刺激の大きさが特定の値（これをしきい値などと呼びます）を超えると、そのニューロンは「活性化」します。そして、活性化したニューロンは、次のニューロンへと自分が受け取った信号を伝えます。すなわち、あるニューロンが次のニューロンに信号を伝達するかは、ニューロンが活性化されたか否かで決定されます（この辺の機構は、ソフトウェアとしてのニューラルネットワークとは少々異なるものですが、これについては後日取り上げることにします）。

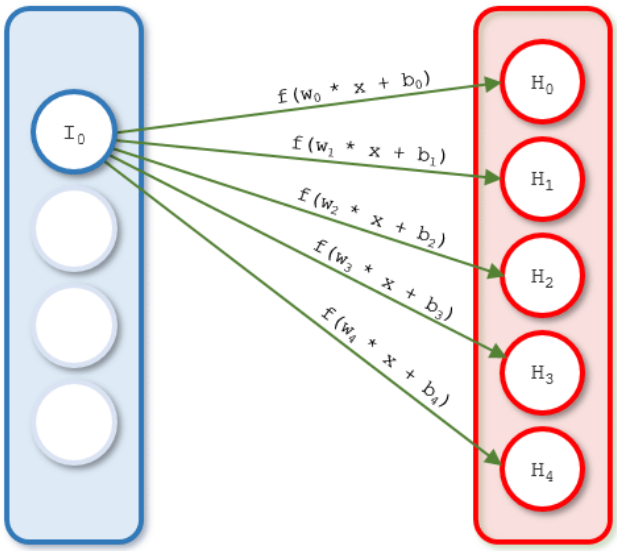
→目次に戻る

1 つのニューロンは多数のニューロンからの信号を受け取り、それをまた 1 つ以上のニューロンへと伝えたり伝えなかったりします。何らかの信号（入力）を基に判断が行われる際には、それに関連した多数のニューロンで構成されるネットワーク内を信号が伝わったり伝わらなかったりすることで、「これは○○」「これは ××」という結果が得られるということです（こうした処理がほんの一瞬で行われています）。

これをコンピューターで再現するために、ディープラーニング用のフレームワークでは「重み」「バイアス」「活性化関数」などと呼ばれるものが使われています（これらは層と層、あるいはノードとノードのつながりごとに定められる値、または関数です）。「重み」（weight、重み付け、ウエィト）の値は受け取った入力に乗算され、その結果に対して「バイアス」（bias）の値が加算されます。

次のノードへの出力を「y」とすると、「y」の値は「入力 × 重み+バイアス」として計算できます。このことから、重みは次のノードへの影響の大きさ、バイアスは「入力 × 重み」にはかせるゲタのようなもの＝「ニューロンの活性化のしやすさ」と捉えることができるでしょう。そして、重みとバイアスを用いて計算した結果を「活性化関数」（activation function）に渡すことで、次のノードに伝達する値が決定されます（人の脳であれば、受け取った刺激がしきい値より大きいかどうか、つまりニューロンが活性化するか否かで、信号が伝達されるか否かが決定されますが、ソフトウェアの方では活性化関数により算出された何らかの値が常に次のノードに渡されます）。

これを図にしたものを以下に示します。ただし、上の画像のままではノードとエッジの数が多いので、ここでは入力層の一番上のノード（「I_0」とします）と、隠れ層の 5 つのノード（「H_0」～「H_4」とします）とのつながりだけを示します。また、活性化関数をここでは「f(x)」としましょう。あるノードから次のノードに渡す値を算出するときに使用する重みは、ノードとノードのつながりごとに異なる値（「w_0」～「w_4」）とし、バイアスについては隠れ層のノード（「H_0」～「H_4」）ごとに「b_0」～「b_4」が使われるものとします。



x：このノード（I_0）が受け取った値
w_0～w_4：このノード（I_0）から隠れ層のノード（H_0～H_4）に伝える値を計算する際に使用する重み
b_0～b_4 ：このノードから隠れ層のノード（H_0～H_4）に伝える値を計算する際に使用するバイアス
f()：活性化関数

ソフトウェアで模されたニューラルネットワークにおける情報の伝達

→目次に戻る

入力層の4つのノードと隠れ層の5つのノードを全てエッジでつなぐと、20本になることはすぐ分かります。ノードからノードに流れる数値を計算するには、この数に応じた重みやバイアスも必要です（隠れ層から出力層についても同様です）。つまり、ニューラルネットワークを作るには、多数の数値をなるべく簡単に扱えるような仕組みが必要です（これについては後で話をします）。

さて、ノードごとに定められた重みとバイアスを基に計算した結果（この例では「w0 * x + b0」の計算結果など）は、活性化関数（この例では「f()」）に渡されて、その値が次の層へ伝えられます。そして、次の層でもやはり重みとバイアスを基に計算を行い、その結果が次の層に伝えられ、……、といったように何らかの値がニューラルネットワークを流れていき、出力層から最終的な計算結果が得られます。

今考えている例であれば、4つのデータを基にニューラルネットワーク内でさまざまな計算が行われて、最終的に0～2の数値が得られます（あやめの種類に合わせた0、1、2のように整数値が得られればよいのですが、次回に紹介するサンプルコードでは実際には実数値＝浮動小数点数値が得られるので、それを整数値にしてやる必要があります。ただし、出力層でどのような値を得るかはニューラルネットワークのモデルの設計にもよります）。

このニューラルネットワークにがく片と花弁の長さ／幅を入力した際には、各ノードが持つ重みとバイアスの値が適切なものでなければ、ちゃんとした答えは返ってきません。そこで、それらの値を適切に定めるために、たくさんのデータをネットワークに流して、流すたびに重みとバイアスが適切なものかどうかを確認し、「これはまだ違うなぁ」というのであれば、それらの値を「少しずつ」変化させていきながら、最終的に「おおよそ正しい」であろう重みとバイアスの値を見つけ出す行為がディープラーニングにおける「学習」（learning）です。あるいは、この過程を「訓練」とか「トレーニング」などと呼ぶこともあります。

学習を行う目的でニューラルネットワークに入力するデータのことを「訓練データ」などと呼びます。また、学習の結果、できあがったニューラルネットワークがどの程度の精度で正しい結果を推測できるかを確認する必要があります。このときに使用するデータのことを「テストデータ」（もしくは「精度検証データ」）などと呼びます（これ以外にも学習を進める過程で必要な設定項目もあります。例えば、ニューラルネットワークを構成するのか層の数、各層のノード数、使用する活性化関数などがそうです。これらを「ハイパーパラメーター」と呼びます）。

学習の仕方にはいろいろとあります。例えば、次回で紹介するのは典型的な「教師あり学習」と呼ばれる手法です。教師あり学習では、学習時にニューラルネットワークへ入力するデータ（入力データ）と「正解ラベル」や「教師データ」と呼ばれる「この入力に対する正解の出力はこれ」という値を使用します。そして、ニューラルネットワークからの出力と正解ラベルを比較して、それを基に重みやバイアスを更新していきます。

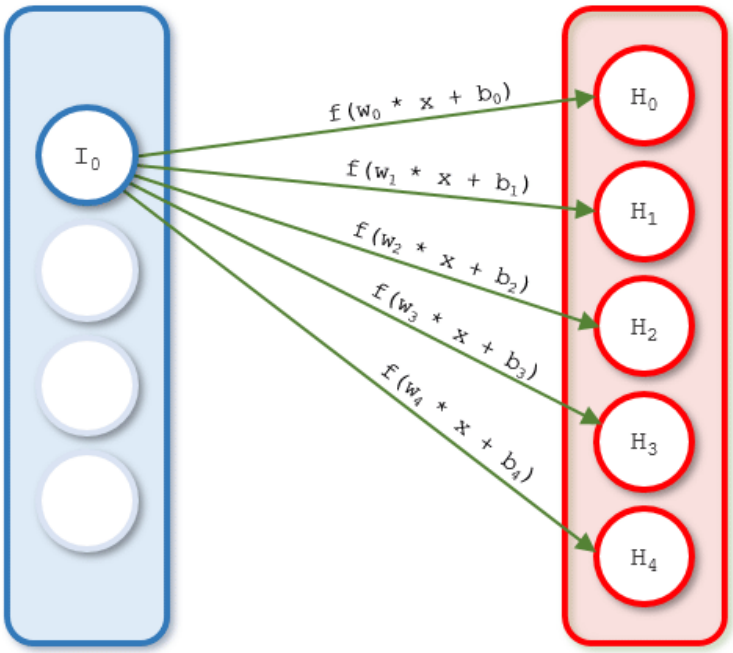
教師あり学習以外にも「教師なし学習」「強化学習」といった手法もありますが、これらについては、今回は触れず、今後、折を見て紹介することにしましょう。

用語が少したくさん出てきたので、ここで簡単にまとめておきます。

- 人工知能：人間が日々行っている知的な活動を人工的に実現すること
- 機械学習：人工知能の種類の一つで、大量のデータを基にコンピューターに「学習」をさせることで、人の知的活動を実現すること
- ディープラーニング：機械学習を行う方法の一つでニューラルネットワークを強力にしたもの
- ニューロン：脳の神経細胞
- ニューラルネットワーク：複数のニューロンで構成されるネットワーク、およびそれをソフトウェアで模したもの。何らかの入力から、何らかの出力を行う
- ノード：ニューロンをソフトウェアで模したものの構成部品。重みやバイアスを持ち、次のノードとはエッジで接続される
- エッジ：ニューロンをソフトウェアで模したものの構成部品。ノードと次のノードの接続を表す
- 重み／バイアス：ノードが持つ属性で、次のノードへ渡す信号の値を計算する際に使われる
- 活性化関数：各ノードで計算された値を基に、次のノードへ伝えられる値を決定する関数
- 学習：ノードが持つ重みやバイアスを微調整して、ニューラルネットワークが適切な出力を得られるようにする過程
- 教師あり学習：あるデータ（入力）に対応する正解となるデータを用いることで、入力から正解を推測できるように重みとバイアスを決定していく過程

次に機械学習と Python の関係について考えてみましょう。

ここで考えることが一つあります。それは機械学習やニューラルネットワークを行う際には、非常に多くの変数を取り扱うことになりがちなことです。例えば、先ほどの例では、入力層の 1 つのノードから隠れ層のノードへの出力を決める際に 5 つ（次の層のノード数）の重みと 1 つのバイアス（と活性化関数）を使用していました（以下に図を再掲します）。



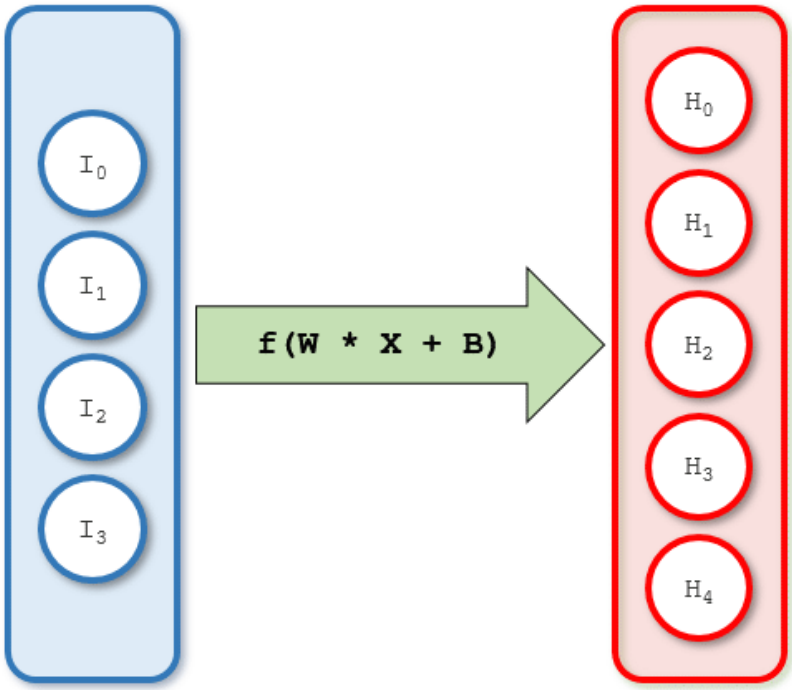
x：このノード（I_0）が受け取った値
w_0～w_4：このノード（I_0）から隠れ層のノード（H_0～H_4）に伝える値を計算する際に使用する重み
b_0～b_4 ：このノードから隠れ層のノード（H_0～H_4）に伝える値を計算する際に使用するバイアス
f()：活性化関数

入力層の 1 つのノードから隠れ層に渡す値を計算するには 5 つの重みと 1 つのバイアスが必要

入力層には 4 つのノードがあるので、入力層から隠れ層へ伝えられる値を計算するだけでも、重みが 20 個とバイアスが 5 個必要だということです。ちなみに、隠れ層から出力層への出力には重みが 5 個、バイアスが 1 個必要になります。つまり、このシンプルなニューラルネットワークですら、合計で 30 個を超えるパラメーターを取り扱う必要があります。

つまり、ニューラルネットワークを作成する際には、入力されるデータ、ノードが持つ重みとバイアス、それらを基にした計算処理、そうしたことを簡便にプログラムコードとして表現できる必要があります。少しだけ詳しく説明すれば、複数のデータを一括して扱える多次元配列とそれを扱う演算があると便利です。入力や重み、バイアスなどを配列にまとめて格納し、それらを（実際には多数の変数が含まれているとしても）単一の変数のように扱えるのが理想的です。

今述べたようなことが可能ならば、上の図は次のように簡便に記述できるかもしれません（これは仮想的な書き方で、実際にこう書けるとは限らないことに注意してください）。



入力層のノードが持つ重み：

$$W = \begin{bmatrix} [w_0, & w_1, & w_2, & w_3], \\ [w_4, & w_5, & w_6, & w_7], \\ [w_8, & w_9, & w_{10}, & w_{11}], \\ [w_{12}, & w_{13}, & w_{14}, & w_{15}], \\ [w_{16}, & w_{17}, & w_{18}, & w_{19}] \end{bmatrix}$$

H_0が受け取る値の計算に使われる重み
H_1が受け取る値の計算に使われる重み
H_2が受け取る値の計算に使われる重み
H_3が受け取る値の計算に使われる重み
H_4が受け取る値の計算に使われる重み

入力層のノードが持つバイアス： $B = [b_0, b_1, b_2, b_3, b_4]$
入力層への入力： $X = [x_0, x_1, x_2, x_3]$
活性化関数： $f()$

多次元配列に入力、重み、バイアスをまとめることで簡便な記述が可能

そして、まさにそうしたライブラリがあったことから、Python はこの分野においてよく使われるようになりました。特に有名なのは、NumPy です。これは多次元配列とその演算を高速に処理できるライブラリであり、NumPy をベースとしたライブラリやフレームワークが現在では多数存在しています。また、コードとそれに付随するドキュメントを効果的に記述できるソフトウェアが登場して、反復的なコード記述と実行を効率的に行えるようになったことも、こうした分野における Python の爆発的な広まりを後押ししたといえます。

このようなライブラリやアプリとしては、例えば次のようなものが挙げられます。

- NumPy：科学計算パッケージ。多次元配列を直観的かつ高速に操作可能
- SciPy：NumPy をベースとした広範な科学計算を行うための Python モジュール
- pandas：データ解析／操作を行うためのライブラリ
- matplotlib：印刷品質のグラフや図を描画するためのライブラリ
- scikit-learn：Python 用の機械学習ライブラリ

- Tensorflow：Google が開発した機械学習プラットフォーム
- PyTorch：Facebook が開発した機械学習フレームワーク
- Jupyter Notebook ／ JupyterLab：実行可能なコードを含んだドキュメントを作成可能な Web アプリケーション

これら以外にも、機械学習やディープラーニングに利用できるさまざまなライブラリやフレームワークが Python にはあります。機械学習に関わる多くの人が Python に集まったことから、そのために使えるさまざまなソフトウェアが作り出され、さらに人を呼び寄せることになり、機械学習のエコシステム（生態系）ができあがったといえます。もちろん、他のプログラミング言語を使っても機械学習を学んだり、それを実地に活用したりは可能ですが、「これからやってみよう」というスタート地点としては、多くの人が活発に活動をしている Python はよい選択肢といえます。

本連載では、Python をプログラミング言語として、機械学習フレームワークとしては主に PyTorch を使います。また、各回で作成したプログラムは Google がホストしている Jupyter Notebook 環境である Google Colaboratory（以下、Google Colab）に掲載する予定です。

なお、本連載では Google Colab の使い方を詳細に説明することはしません。基本的な使い方については「機械学習＆ディープラーニング入門（作業環境準備編）」の「Lesson 2　Google Colaboratory（Jupyter Notebook）の準備と、ノートブックの作成」を参考にしてください。Google Colab を使うには Google アカウントが必須なことには注意が必要です。

また、Python コードの構文についても詳しくは説明しません。ざっくりとした説明はしますが、具体的な構文で分からないところがあるときには、「Python 入門」の該当する回を参考にしてください。

以下では、Google Colab でノートブックの新規作成とコードの入力／実行までを簡単に見ておきましょう。

Google Colab の使い方

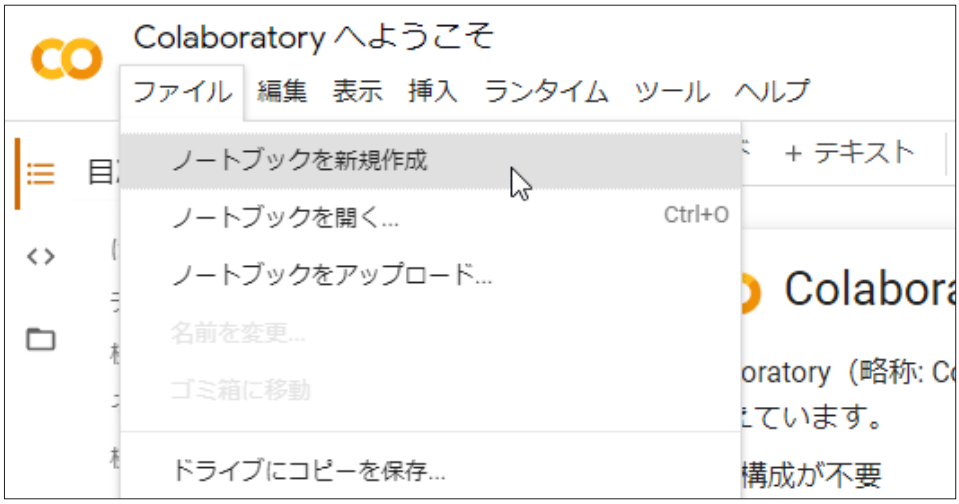
まずは Google Colab のページを開き、Google アカウントでログインしていなければログインしてください。

Google Colab へのログイン

初めてログインをすると、上の画面でログイン済みになっているページが表示されます。

Google Colab にログインをした後の画面

ここで［ファイル］メニューにある［ノートブックを新規作成］を選択すると、新しい「ノートブック」が作成されます。

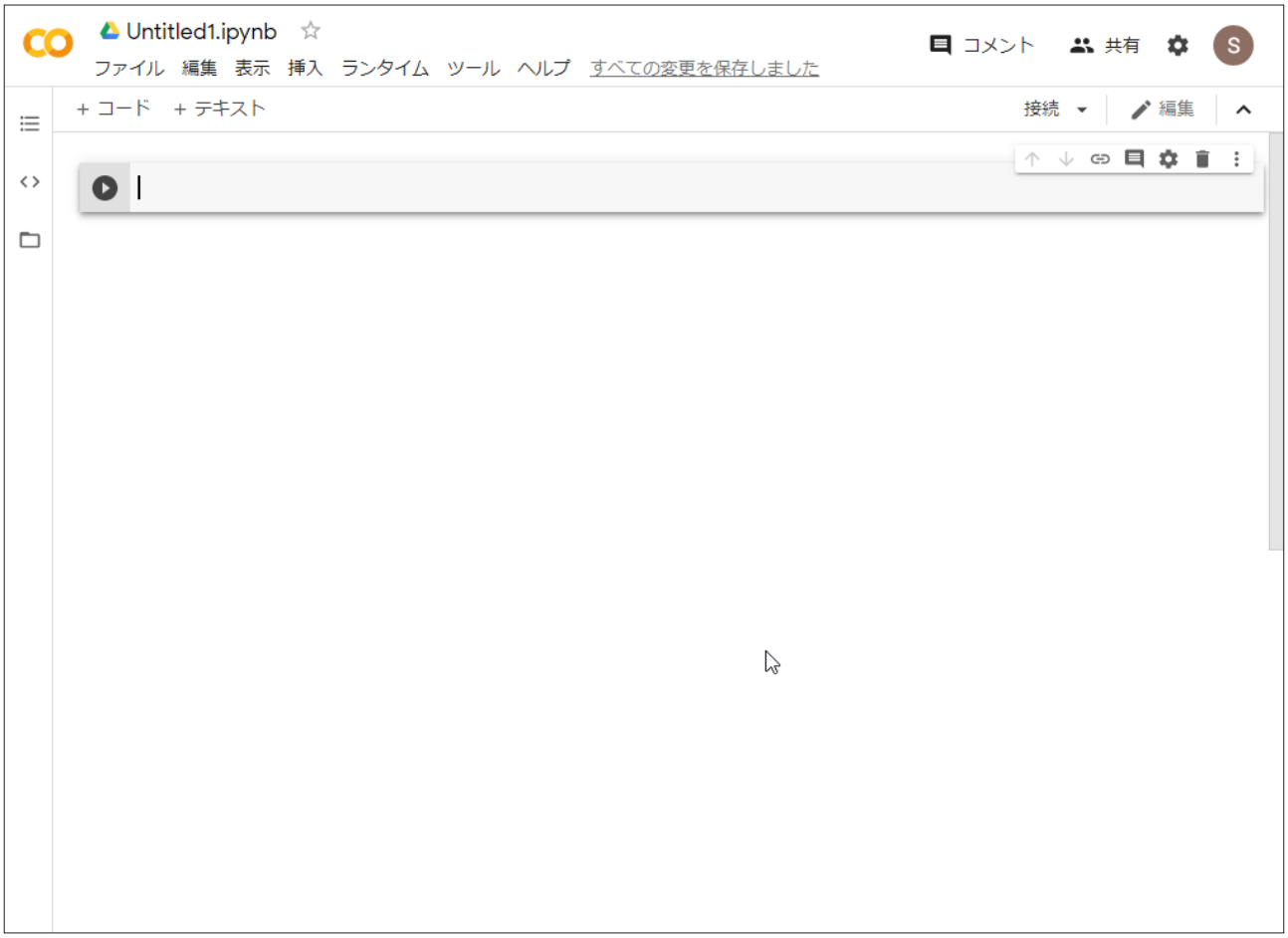


ノートブックの新規作成

あるいは、Google Colab で既にノートブックを作ったことがあるのであれば、「Google Colab」などを検索語として検索して、表示されたリンクをクリックすると、次のようにどのノートブックを使用するかを問い合わせるダイアログが表示されるかもしれません。このときには、使用するノートブックをクリックするか、下にある［ノートブックを新規作成］をクリックしてください。

ノートブックの選択画面

ノートブックには、Python のプログラムコードと、それに関連するドキュメントやメモを書き入れられるようになっています（コードを書く「セル」と、ドキュメントを書く「セル」があります）。ノートブックを新規作成すると、最初にコードを入力するセルが 1 つだけ表示されます。



作成されたノートブック

書いたコードを実行するには、セルの左端にある［セルを実行］ボタンをクリックします。試しにここでは以下のコードを入力してみましょう。

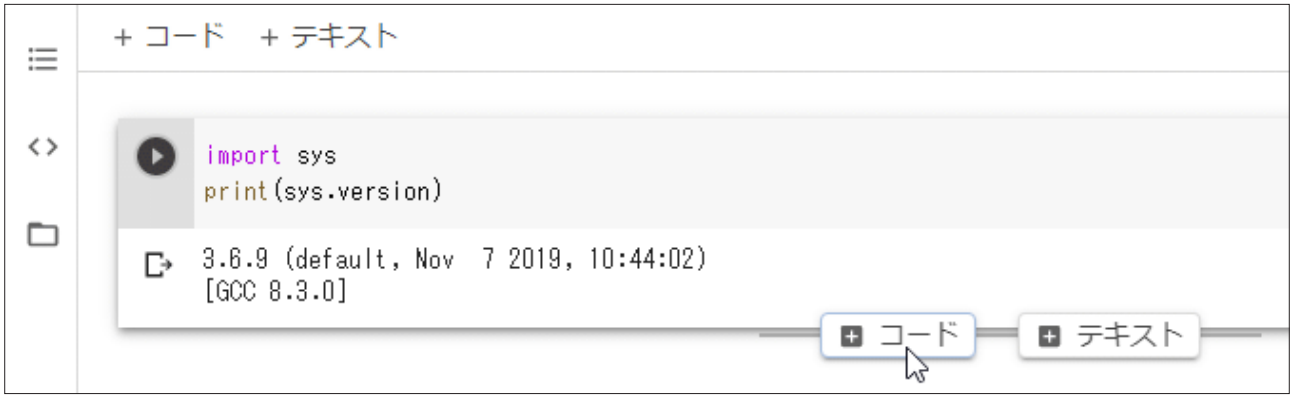
```
import sys
print(sys.version)
```

Google Colab で実行されている Python のバージョンを調べる

セルに上記のコードを入力して、［セルを実行］ボタンをクリックすると、次のようにその実行結果が表示されます。［Ctrl］＋［Enter］キーを押すと、それまで入力をしていたセルのコードが実行されるので、キーボード派の方はそちらの方法を選択するのもありでしょう。

実行結果

このことから、Google Colab で実行されている Python のバージョンは 3.6.9 であることが分かりました。コードを入力するセルを追加するには、上にある［+ コード］ボタンをクリックするか、セルの下端の真ん中当たりでマウスをホバーさせるとポップアップ表示される［+ コード］ボタンをクリックするかします（キーボードショートカットもあるので、キーボード派の方はそちらを使うのが簡単です）。



セルの新規作成

→目次に戻る

コードを入力しては実行し、次のコードを入力しては実行し、……、と対話的にプログラムを実行しながら（つまり、個々のステップの動作を確認しながら）、一つのドキュメントとしてプログラムを構築していけるのが、Google Colab や Jupyter Notebook ／ JupyterLab のよいところです。エラーが出たら、そのコードをその場で修正して、再実行といったことも簡単に行えます。

Google Colab で新規にノートブックを作成して、セルにコードを入力し、それを実行して、次のセルを追加するところまでが分かれば、取りあえずは十分です。Google Colab の便利な使い方については、必要に応じて、今後の回で取り上げる予定です。

少々長くなったので、今回はここまでとしましょう。次回は、あやめの特徴を記したデータを入力すると、その種類を推測するニューラルネットワークを実際に作成しながら、ニューラルネットワークやディープラーニングによく出てくる用語や概念について見ていきます。

Hello Deep Learning：
ニューラルネットワークの作成手順

あやめの品種を推測するニューラルネットワークを作りながら、データセットの準備、ニューラルネットワークの定義、学習とテストまでの手順を駆け足で見てみましょう。

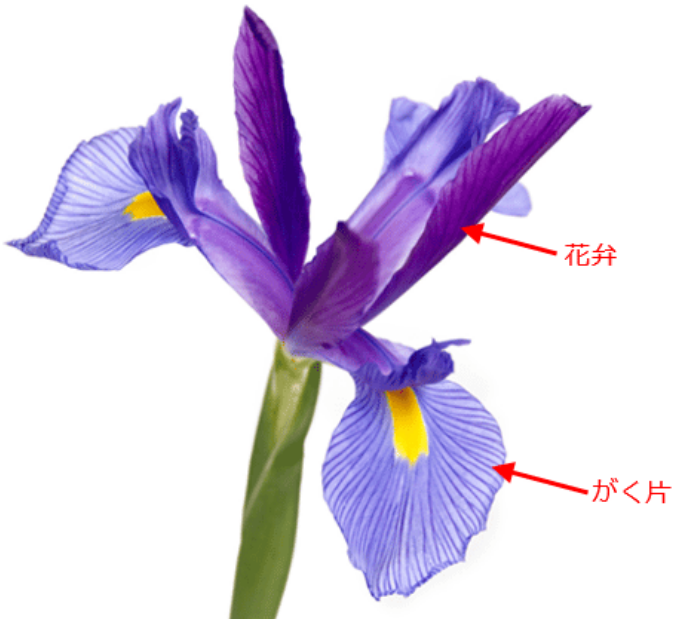
（2020 年 04 月 03 日）

前回は人工知能、機械学習、ディープラーニングなどの基本的な考え方、Google Colaboratory（以下、Google Colab）の基本的な使い方を見ました。今回から数回に分けて、あやめの品種を分類するニューラルネットワークを作りながら、その作成の大まかな流れと、コードの詳細について見ていきましょう。

今回は、ニューラルネットワークを作成していく大まかな手順を紹介します。その途中では、少し難解な用語が出てくるかもしれません。しかし、それらを全て今の段階で理解しなくてもかまいません。コードの詳細な説明は次回以降に行うこととして、まずは「こんな感じで作るんだ」というのを感じてくれれば十分です。

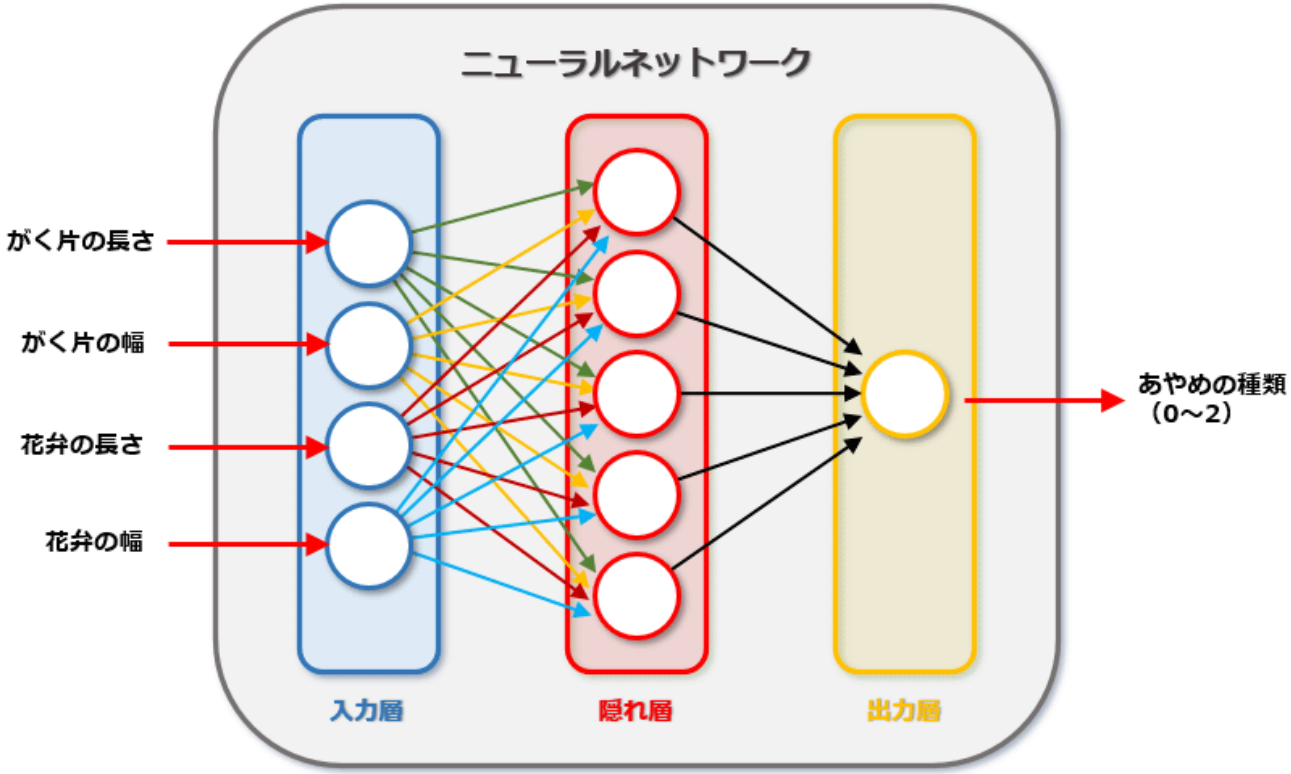
ニューラルネットワーク作成の手順

今回は前回に言及した「がく片の長さと幅、花弁の長さと幅からあやめの品種を推測する」ニューラルネットワークを作成してみます。ここで使用するのは、いわゆる「教師あり学習」と呼ばれる方法です。



あやめの花の構造（前回の再掲）

その全体像は前回にも述べた通り、以下のようなものでした。



今回作成するニューラルネットワーク

そして、このニューラルネットワークの作成は、次のような手順で進めます。

- データセットの準備と整形
- ニューラルネットワークの定義
- 学習（訓練）と精度の検証

以下ではまずこの手順について簡単にまとめておきましょう。

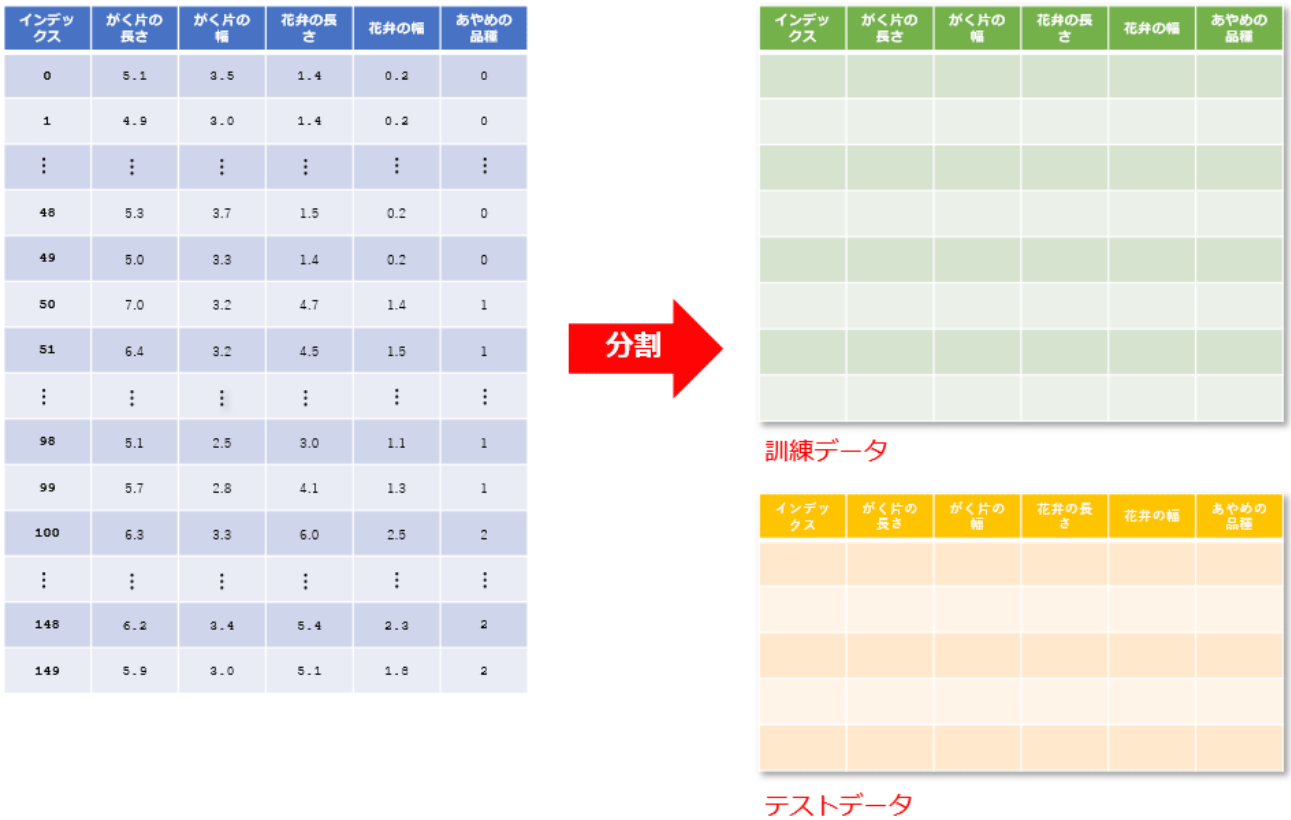
データセットの準備と整形

「データセット」とは、大量のデータが一定のフォーマットで並べられたものと理解すればよいでしょう。今回の例なら、あやめの特徴（がく片の長さと幅、花弁の長さと幅の 4 つの数値）と、その品種が並んだものです（この後で取り扱うデータセットでは、以下の図とは少々異なる形式になっていますが、実質的にはこれと同様です）。

インデックス	がく片の長さ	がく片の幅	花弁の長さ	花弁の幅	あやめの品種
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
⋮	⋮	⋮	⋮	⋮	⋮
49	5.0	3.3	1.4	0.2	0
50	7.0	3.2	4.7	1.4	1
⋮	⋮	⋮	⋮	⋮	⋮
149	5.9	3.0	5.1	1.8	2

データセット

データセットに含まれる大量のデータは一般に、幾つかに分割して使われます。1 つはまさに学習を行うための入力データとして。また、学習が終わったニューラルネットワークの性能を試すためのデータとして使われるものもあります。今回は触れませんが、学習のやり方を調整するために使うデータへと分割する場合もあります（その場合は、データセットは 3 つに分割されることになります）。



データセットは分割して使われる

→目次に戻る

また、データセットに含まれるデータを、これから使用する Python のフレームワーク（ライブラリ）である PyTorch で扱えるように、少し加工する必要もあることにも注意してください。

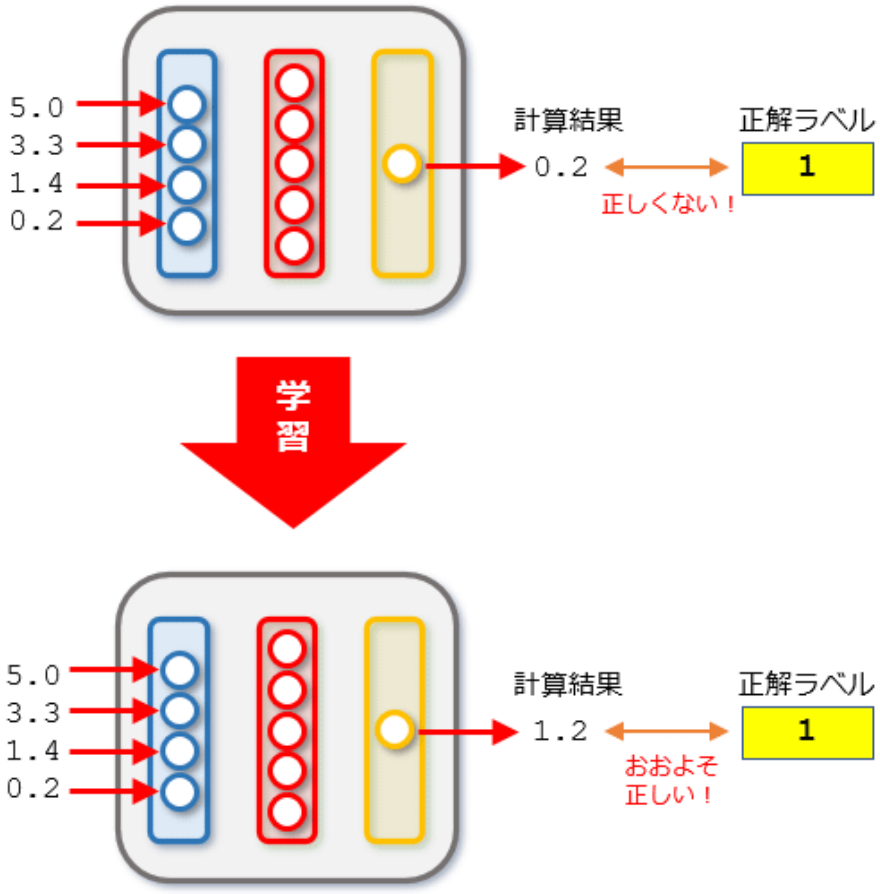
ニューラルネットワークの定義

その次にニューラルネットワークを表すクラスを定義します。今回は既に名前が挙がっていますが、PyTorch と呼ばれるディープラーニング用フレームワークを使用します（Google Colab 環境には既にインストール済みです）。

このときには、プログラミング言語として Python を用いて、上で示した構成のニューラルネットワークがどんなものかをコードとして記述していきます。といっても、PyTorch の力を借りることで、前回に述べた「重み」や「バイアス」といった事柄については、プログラマーがあまり気にしなくても済むようになっています（それでも、そうした点まである程度理解できるようになることを本連載では目的としているので、ゆくゆくは実際にそうした部分まで踏み込んでみていくことになるでしょう）。

学習と評価

次にくるのが学習です。この段階では、ここまでの 2 つの手順で用意したデータセットとニューラルネットワークを使って、実際にあやめの特徴を記したデータを入力すると、その品種を推測できるようにしていきます（以下の図では計算結果が浮動小数点数値になっています。そのため、これを整数値に変換する作業が必要になることが予測できます）。



学習によってニューラルネットワークはおおよそ正しい結果を計算できるようになる

→目次に戻る

この後では、学習前にニューラルネットワークが出力した値（推測結果）と、学習後のニューラルネットワークが出力した結果についても見てみることにします。そうすることで、「確かにニューラルネットワークが学習して、おおよそ正しい結果を出せるようになった」と実感できるはずです。

これでおおよその手順についての説明はおしまいです、以下では実際にコードを見ながら、ニューラルネットワークを作成していくことにしましょう。

それでは、前回に紹介した方法で Google Colab のページを開いて、ノートブックを新規に作成しましょう（本稿のコードはここで公開しているので、必要に応じて参照してください）。

データセットの準備と整形

今回の例となるあやめの品種を推測するニューラルネットワークを作成するのに必要なデータセットは、前回に名前を挙げた scikit-learn という機械学習フレームワークに含まれています。そして、このフレームワークは Google Colab の実行環境に事前にインストールされています。そこで、ここではこれを利用して、データセットを読み込むことにしましょう。

ここでは次の作業を行います。

1. あやめのデータセットの読み込み（sklearn.datasets.load_iris 関数を使用）
2. データセットの分割（sklearn.model_selection.train_test_split 関数を使用）
3. データの整形（PyTorch のユーティリティー関数を使用）

本稿の冒頭でも述べたように、読み込んだデータセットはニューラルネットワークの学習用とその評価（テスト）用に分割する必要があることも忘れないでください。そして、最後にそれらを今回使用するフレームワークである PyTorch で使えるように少々の整形（データ型の変換）を行います。では、上の手順に従ってコードを書いていくことにしましょう。

あやめのデータセットの読み込み

データセットを読み込むには scikit-learn が提供する sklearn.datasets モジュールから load_iris 関数をインポートして、実行するだけです。実際のコードを以下に示します。

```
from sklearn.datasets import load_iris
iris = load_iris()
```

あやめのデータセットを読み込むコード

このコードを書いて、セルを実行してください。その結果を以下に示します（何も出力はありません）。

```
from sklearn.datasets import load_iris
iris = load_iris()
```

実行結果

これだけであやめのデータセットの読み込みは完了です。手順に従うと次はデータセットの分割です。が、その前に少しだけ、このデータセットについて説明をしておきます。詳しい説明は次回に行うので、「このデータセットはこんな構造になっているんだ」ということだけを把握しておきましょう。これを知らないことには、以下でいったい何をしているのか分からなくなるかもしれません。

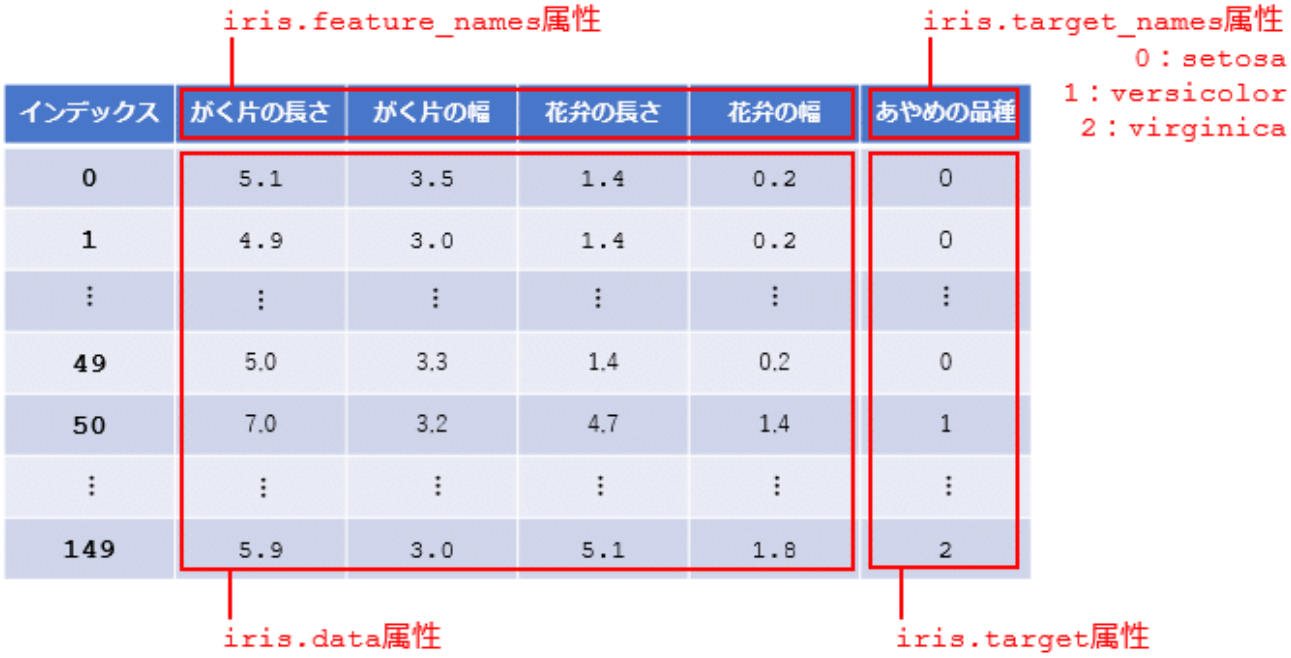
このデータセットには次のような属性（インスタンス変数）があります。

- data 属性：個々のあやめのデータを 3 品種 ×50 個含んだ NumPy 配列。1 つのデータは「がく片の長さ」「がく片の幅」「花弁の長さ」「花弁の幅」の 4 つの浮動小数点数値で構成され、それが合計で 150 個並んだ「配列の配列」。
- feature_names 属性：上で述べたデータの説明。1 つのデータが上で述べた順で並んでいることを説明した文字列を含むリスト
- target 属性：data 属性の配列の同じインデックス位置にある NumPy 配列（がく片／花弁の長さ／幅を表す 4 つの数値を要素とする配列）が何の品種であるかを示す「0」「1」「2」の整数値のいずれか
- target_names 属性：target 属性の各整数値があやめのどの品種であるかを説明する文字列を含む NumPy 配列（「0」は「setosa」という品種に、「1」は「virsicolor」という品種に、「2」は「verginica」という品種に対応しています）

なお、ここでいう「NumPy 配列」とは、NumPy が提供するデータ型で、Python のリストと似た使い勝手を持ちながら、多次元配列を効率的に扱えるようにしたものです。scikit-learn は NumPy をベースに作られた機械学習フレームワークであり、load_iris 関数により読み込まれたデータセットは、NumPy にネイティブなデータ型にまとめられているということです。そのため、後からこれを PyTorch で扱えるように変換します。

その一方で、機械学習やディープラーニングの世界では、配列（一定の形式で複数のデータを並べたもの）のことを「テンソル」と呼ぶことがよくあります。例えば、data 属性は「2 次元（階数が 2）のテンソル」などと呼びます。この後は配列や多次元配列と同じ意味で「テンソル」という言葉も出てくるので覚えておいてください。

先ほどの図に上記の属性を書き加えたものを以下に示します。



インデックス	がく片の長さ	がく片の幅	花弁の長さ	花弁の幅	あやめの品種
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
⋮	⋮	⋮	⋮	⋮	⋮
49	5.0	3.3	1.4	0.2	0
50	7.0	3.2	4.7	1.4	1
⋮	⋮	⋮	⋮	⋮	⋮
149	5.9	3.0	5.1	1.8	2

データセットと属性の関係

学習と評価で実際に使うデータは、data 属性と target 属性に格納されているということです（feature_names 属性と target_names 属性は 2 つの属性を説明するためのもので、データセットを人が調べるときや、画面に何らかのデータを表示するときに使用します）。iris.data 属性に格納されているデータ（とそこから分割されるデータ）はニューラルネットワークに実際に入力されるデータ（入力データ）であり、iris.target 属性はニューラルネットワークが iris.data 属性に格納されている値を基に計算（推測）した結果がどんな値になるべきかを示すデータ（正解ラベル、教師データ）です。

ここで注意しておきたいのは、上の図から分かる通り、data 属性には品種ごとにデータがまとめて格納されている点です（後でデータセットを分割するときには、これらがバラバラになるようにシャッフルします）。

せっかくなので、データを幾つか表示してみましょう。ここでは iris.data 属性と iris.target 属性の値を zip 関数でひとまとめにして、enumerate 関数でインデックスとその内容が得られるようなコードを書いています（インデックスは先頭の 5 個のデータだけを表示するために使用しています）。

```
for idx, item in enumerate(zip(iris.data, iris.target)):
    if idx == 5:
        break
    print('data:', item[0], ', target:', item[1])
```

先頭 5 つの要素を表示

実行結果を以下に示します。

```
for idx, item in enumerate(zip(iris.data, iris.target)):
    if idx == 5:
        break
    print('data:', item[0], ', target:', item[1])
```

```
data: [5.1 3.5 1.4 0.2] , target: 0
data: [4.9 3.  1.4 0.2] , target: 0
data: [4.7 3.2 1.3 0.2] , target: 0
data: [4.6 3.1 1.5 0.2] , target: 0
data: [5.  3.6 1.4 0.2] , target: 0
```

実行結果

すると、「data:」に続けて角かっこ「[]」に囲まれた 4 つの数値（入力データ）と、「target:」に続けて 4 つの数値に対応する整数値（正解ラベル）が表示されました。先ほども述べたように、iris.data 属性には品種ごとに 50 個のデータが順番に並べられているので、「target」の値は全て「setosa」という品種を表す「0」だけになっています。

データセットのおおまかな内容が分かったところで、このデータセットを分割しましょう。

データセットの分割

データセットの分割と聞くと難しく感じるかもしれませんが、今回に限っては scikit-learn が提供する関数（sklearn.model_selection モジュールの train_test_split 関数）を使うだけです。実際のコードを以下に示しましょう。ここでは分割前のデータ数と、分割後のデータ数も表示するようにしてあります。

```
from sklearn.model_selection import train_test_split
print('length of iris.data:', len(iris.data))  # iris.data のデータ数
print('length of iris.target:', len(iris.target))  # iris.target のデータ数

# iris.data と iris.target に含まれるデータをシャッフルして分割
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target)
print('length of X_train:', len(X_train))
print('length of y_train:', len(y_train))
print('length of X_test:', len(X_test))
print('length of y_test:', len(y_test))
```

データセットの分割

このコードでは、sklearn.model_selection モジュールから train_test_split 関数をインポートした後で、読み込んだデータセット（iris.data 属性と iris.target 属性）の要素数を表示し、次にインポートした train_test_split 関数でそれらを分割して、分割後のデータセットの要素数を表示しています。

気を付けなければいけないのは、学習時にニューラルネットワークに入力されるデータ（iris.data 属性）と、その計算結果との比較に使われるデータ（iris.target 属性）が別々になっている点です。そのため、上の「train_test_split(iris.data, iris.target)」という呼び出しでは、両者を渡しています（こうすることで、以下で述べるシャッフルが行われた際に、分割後の配列において入力データと正解ラベルの対応関係が維持されます）。

もう一つ、コメント中に「シャッフル」とありますが、これは元の 2 つのデータセットの要素をランダムに並べ替えることを意味しています。既に述べたように、このデータセットには 3 種類のあやめの品種ごとに 50 個のデータ（合計で 150 個のデータ）が並べられています。このまま頭から順番に分割してしまうと、分割後のデータに偏りが発生してしまうので、ここではデータセットをシャッフルして、分割後のデータセットに 3 品種のデータがだいたい同じくらいの割合で含まれるようにしているのです。

この関数呼び出しを行うと、変数 X_train と X_test には iris.data 属性の 150 個のデータを分割したものが、変数 y_train と y_test には iris.target 属性の 150 個のデータを分割したものが代入されます。

上記コードを実行した結果を以下に示します。

```
from sklearn.model_selection import train_test_split
print('length of iris.data:', len(iris.data))  # iris.dataのデータ数
print('length of iris.target:', len(iris.target))  # iris.targetのデータ数

# iris.dataとiris.targetに含まれるデータをシャッフルして分割
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target)
print('length of X_train:', len(X_train))
print('length of y_train:', len(y_train))
print('length of X_test:', len(X_test))
print('length of y_test:', len(y_test))
```

```
length of iris.data: 150
length of iris.target: 150
length of X_train: 112
length of y_train: 112
length of X_test: 38
length of y_test: 38
```

実行結果

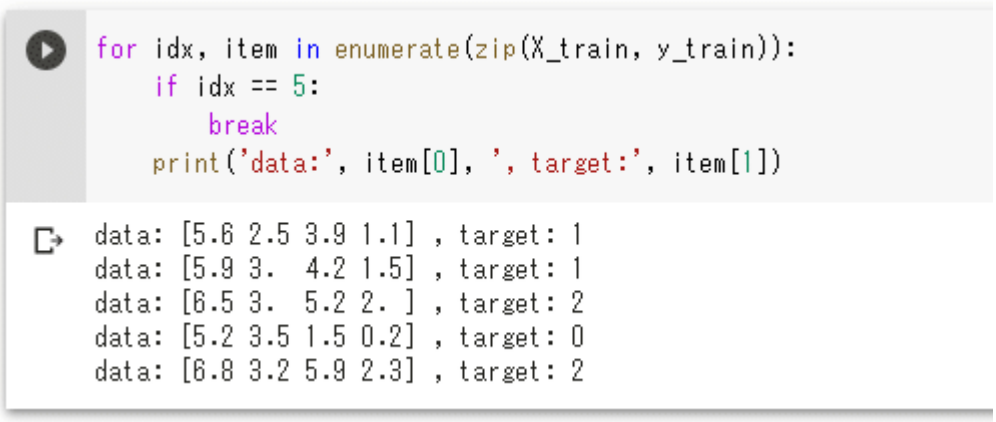
この結果から分かる通り、150 個のデータが学習用に 112 個のデータ（元のデータセットの 75%）、テスト用に 38 個のデータ（元のデータセットの 25%）へと分割されました。

分割後のデータを幾つか見てみましょう。

```
for idx, item in enumerate(zip(X_train, y_train)):
    if idx == 5:
        break
    print('data:', item[0], ', target:', item[1])
```

シャッフルして分割された後のデータセットの先頭 5 要素を表示

これを実行した結果を以下に示します（分割時にデータはシャッフルされるので、読者が実際にこのコードを試すと、これとは異なる出力が得られるでしょう）。



実行結果

今度はがく片と花弁の長さや幅もバラバラで、それらに対応するあやめの種類も異なるものが表示されました。

データの整形

データセットの準備の最後に、分割後のデータセットを今回使用するフレームワークである PyTorch で使えるように少しデータ型の変換を行います。読み込んだデータは scikit-learn が提供するものでした。scikit-learn は NumPy と呼ばれる科学計算用を高速かつ簡便に行うためのパッケージをベースに作られていて、先ほども述べたように、あやめの特徴を記したデータは「NumPy 配列」と呼ばれるデータ構造に格納されています。そこで、これらを PyTorch で扱えるように変換する必要があるのです。

実際のコードを以下に示します。

```
import torch

X_train = torch.from_numpy(X_train).float()
y_train = torch.tensor([[float(x)] for x in y_train])
X_test = torch.from_numpy(X_test).float()
y_test = torch.tensor([[float(x)] for x in y_test])
```

データ型の変換

このコードではまず、import 文で torch パッケージをインポートしています。このパッケージは、PyTorch が提供する多次元配列「テンソル」や、それを扱うための演算子、各種のユーティリティー関数などを定義したものです。これをインポートすることで、NumPy 配列から PyTorch のテンソルを作成する from_numpy 関数や、リストや数値などの Python オブジェクトから PyTorch のテンソルを作成するのに使える tensor 関数などを利用できるようになります。

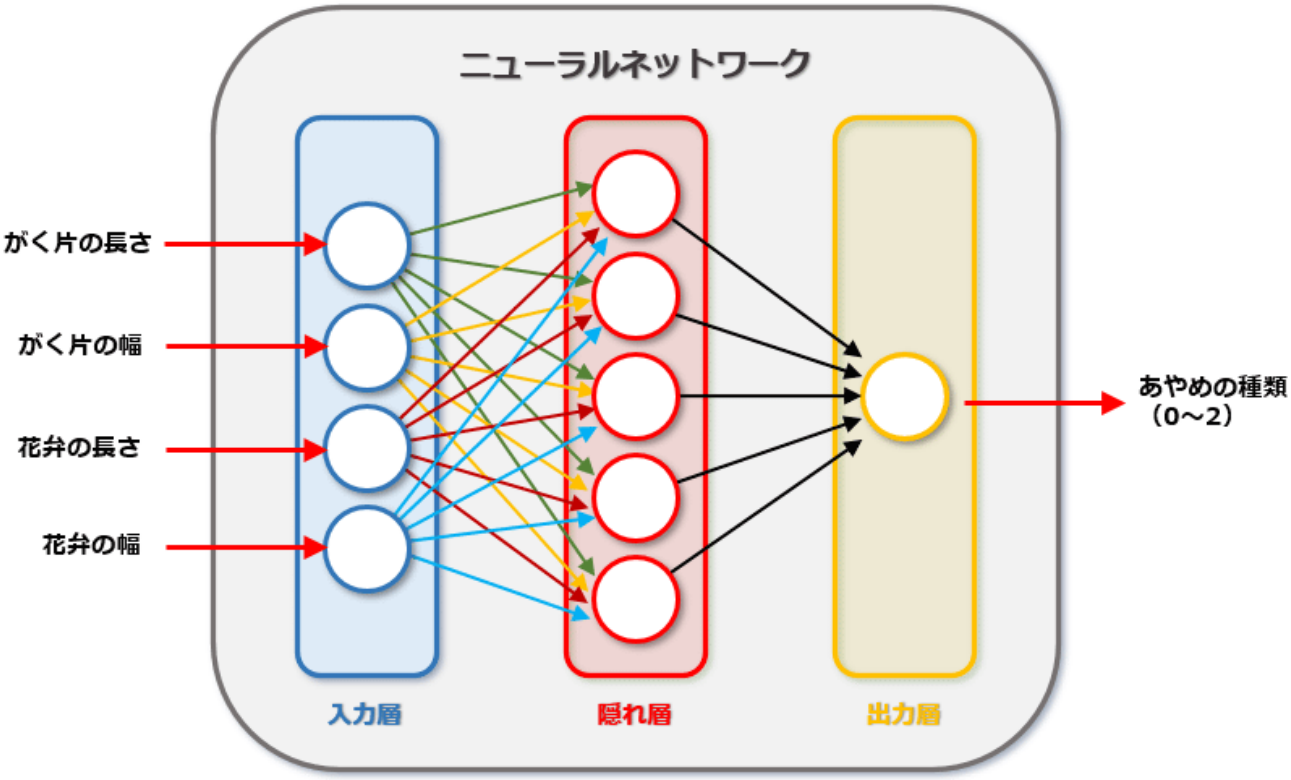
実際、その下のコードではこれらを使って、torch.from_numpy 関数を使って NumPy 配列から PyTorch のテンソルを作成したり、torch.tensor 関数を使って PyTorch のテンソルを作成したりしています。これらのコードの詳しい説明も次回以降に行うことにしましょう。ここでは、「NumPy 配列を PyTorch で使えるように変換している」ことだけを覚えておけば十分です。

なお、実際にどのようなフォーマットのデータに整形するかは、自分が作成するニューラルネットワークの仕様や、学習やその精度評価で使用する損失関数、最適化アルゴリズム などとの兼ね合いになるので、常に上のような変換を行うとは限りません。

このコードは特に画面に何かを表示したりはしませんが、忘れずに実行しておいてください。データセットの準備はこれで完了です。次にニューラルネットワークの定義に移りましょう。

ニューラルネットワークの定義

ここでは PyTorch が提供するフレームワークの機能を利用して、ニューラルネットワークを表すクラスを定義していきます。その前に、ここで作成するニューラルネットワークについて思い出しておきましょう。



今回作成するニューラルネットワーク（再掲）

ここでは全部で 3 つの層で構成されるニューラルネットワークを作成するのでした。

- 入力層：ノードは 4 つ（がく片／花弁の長さ／幅を受け取ります）、出力は 5 つ
- 隠れ層：ノードは 5 つ（入力層からの出力を各ノードが受け取ります）、出力は 1 つ
- 出力層：ノードは 1 つ（隠れ層からの出力を受け取り、それをそのまま出力します）

ある層のノードが次の層の全てのノードと接続されるようなニューラルネットワークをここでは作成します。このようなニューラルネットワークは PyTorch では Linear クラス（torch.nn.Linear クラス）を使うことで簡潔に記述できます。

実際のコードを以下に示します。コードを入力したら、実行しておきましょう。

```
from torch import nn

INPUT_FEATURES = 4
HIDDEN = 5
OUTPUT_FEATURES = 1

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(INPUT_FEATURES, HIDDEN)
        self.fc2 = nn.Linear(HIDDEN, OUTPUT_FEATURES)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        return x
```

ニューラルネットワークを表すクラス

まず、PyTorch が提供する torch.nn モジュールをインポートしています（「nn」は「neural network」の頭字語でしょう）。PyTorch では torch.nn モジュールでニューラルネットワークの基底クラスとなる Module クラスを定義しています（下のクラス定義を見ると、torch.nn.Module クラスを基底クラスとしていることが分かりますね）。

その下の 3 行では、先ほど述べた入力層のノード数（INPUT_FEATURES）、隠れ層のノード数（HIDDEN）、出力の数（OUTPUT_FEATURES）を変数に代入しています。最後の Net クラスの定義が、ここで使用するニューラルネットワークを記述したものです。

```
class Net(nn.Module):
    def __init__(self):
        # ……省略……

    def forward(self, x):
        # ……省略……
```

Net クラスは torch.nn.Module クラスを継承する

Net クラスは nn.Module クラス（torch.nn.Module クラス）を基底クラスとすることで、PyTorch が提供するニューラルネットワーク（PyTorch では「ニューラルネットワークモジュール」と呼んでいます）の全ての機能を継承するようになっています（つまり、これだけでニューラルネットワークの基本機能を備えるということです）。そして、__init__ と forward の 2 つのメソッドを定義しています。

__init__ メソッドでは、最初に基底クラスの __init__ メソッドを呼び出して、その初期化を行っています（詳細な説明は省きますが、Python 2 だと「super(Net, self).__init__()」と書く必要がありますが、ここでは Python 3 を対象としているので、このようなシンプルな書き方をしています）。その後に、2 つのインスタンス変数 self.fc1 と self.fc2 に、先ほど述べた Linear クラスのインスタンスを代入しています。

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(INPUT_FEATURES, HIDDEN)  # 入力層（入力:4、出力:5）
        self.fc2 = nn.Linear(HIDDEN, OUTPUT_FEATURES)  # 隠れ層（入力:5、出力:1）

    def forward(self, x):
        # ……省略……
```

__init__ メソッド

Linear クラスのインスタンスを生成する際には、そのインスタンスが表す層（ここでは入力層と隠れ層に相当するインスタンスを生成しています。出力層については後述）への入力の数と、その層から次の層への出力の数を指定します。インスタンス変数 self.fc1 へ代入する Linear クラスのインスタンスの生成では「self.fc1 = nn.Linear(INPUT_FEATURES, HIDDEN)」のようにして、入力は INPUT_FEATURES（4）、出力は HIDDEN（5）を渡しているので、これが入力層を表すインスタンスということです。インスタンス変数 self.fc2 についても同様です。

ここで「self.fc3 というインスタンス変数に出力層を表す Linear クラスのオブジェクトを代入する必要はないの？」と思う方もいらっしゃるかもしれません。が、ここでは隠れ層からの出力をそのまま外部への出力としてしまうことにしているので、出力層に対応するインスタンスは作成していません。

注意してほしいのは、__init__ メソッドで行っているのは「ニューラルネットワークを構成する層の定義」だけであることです。

__init__ メソッドで定義した層が実際にどのようにつながっているか（ニューラルネットワークがどのように計算を連ねていくか）は forward メソッドで定めています。forward メソッドは入力 x（あやめの特徴を示す 4 つのデータ）を受け取り、それを self.fc1 で処理して、その結果をに torch.sigmoid 関数に通した結果を今度は self.fc2 メソッドで処理し、その結果を戻り値（ニューラルネットワークの計算結果）としています（ここで「self.fc1 と self.fc2 は Linear クラスのインスタンスなのに、メソッドのように呼び出している」ことに気付くかもしれません。が、PyTorch ではこのような書き方ができるようになっています。詳しくは後続の回で説明します）。

```
class Net(nn.Module):
    def __init__(self):
        # ……省略……

    def forward(self, x):
        x = self.fc1(x)  # 入力データ→入力層
        x = torch.sigmoid(x)  # 入力層→活性化関数
        x = self.fc2(x)  # 活性化関数→隠れ層→出力
        return x
```

forward メソッド

ここで出てきた「torch.sigmoid 関数」が、前回にも出てきた「活性化関数」と呼ばれるものです。ここでは、ソフトウェアにおけるニューラルネットワークの世界では、活性化関数は「ある層で処理した結果を、別の層へ渡すときにさらに変換を加える」ものだと思っておいてください。

いろいろと分からないことを積み残したままかもしれませんが、今回はあくまでもニューラルネットワークを作って学習とテストをするまでをザッと見ることが目的なので、そこはあまり気にしないようにしましょう。ともかく、__init__ メソッドと forward メソッドで上のようなコードを書いたことで、「入力データ→入力層→活性化関数→隠れ層→（出力層→）出力」という流れが書けたことになります。

ここで実際の学習に向かう前に、ちょっとこのクラスを使ってみましょう。

```
net = Net()  # ニューラルネットワークのインスタンスを生成

outputs = net(X_train[0:3])  # 訓練データの先頭から 3 個の要素を入力
print(outputs)
for idx in range(3):
    print('output:', outputs[idx], ', label:', y_train[idx])
```

ニューラルネットワークを使ってみる

このコードでは最初に Net クラスのインスタンスを生成し、次に先ほど分割したデータのうち、X_train の先頭から 3 つをニューラルネットワークに入力しています。その結果を変数 outputs に受け取ったら、それをまずは表示して、次にそれらの値と、対応する正解ラベル（変数 y_train）とを比較表示しています。

実際に実行した結果を以下に示します（読者がこのコードを試した場合、以下とは具体的な数値は異なるものが得られるでしょう）。

```
net = Net()  # ニューラルネットワークのインスタンスを生成

outputs = net(X_train[0:3])  # 訓練データの先頭から3個の要素を入力
print(outputs)
for idx in range(3):
    print('output:', outputs[idx], ', label:', y_train[idx])
```

```
tensor([[0.5395],
        [0.5385],
        [0.5209]], grad_fn=<AddmmBackward>)
output: tensor([0.5395], grad_fn=<SelectBackward>) , label: tensor([1.])
output: tensor([0.5385], grad_fn=<SelectBackward>) , label: tensor([1.])
output: tensor([0.5209], grad_fn=<SelectBackward>) , label: tensor([2.])
```

実行結果

最初の出力からは、このニューラルネットワークは、「計算結果を唯一の要素とする配列」を要素とする配列（テンソル）になっていることが分かります（3 つのデータを入力したので、出力結果である配列の要素数も 3 です）。そして、その次の出力を見ると、学習をする前のニューラルネットワークが計算した値が正解ラベルの値とはかけ離れたものになっていることも分かります。今から行う「学習」とは、この計算結果を右側の正解ラベルの値へと近づけていく過程に他なりません。

学習（訓練）と精度の検証

前回も述べたように、今から行う「学習」では、上で見たような形でニューラルネットワークにデータを入力し、それらから得られる計算結果（推測結果）と正解ラベルとを比較しながら、ニューラルネットワークが内部で持っている重みやバイアスを更新していきます。

このときには「損失関数」と呼ばれる関数を用いて、計算結果と正解ラベルとの誤差を比べたり、それらを基に「最適化」と呼ばれる処理を行いながら、重みやバイアスを調整したりしていきます。これらの要素については後続の回で詳しく説明するものとします。ここでは PyTorch を使って学習を行う典型的な（ただし、一般的なものよりもシンプルな）コードを紹介します。

学習の手順はだいたい次のようなものです。

1. ニューラルネットワークに X_train に格納したデータを入力する（112 個）
2. 損失関数を用いて、計算結果と正解ラベルとの誤差を計算する（計算結果は損失 と呼ばれる）
3. 誤差逆伝播（バックプロパゲーション）や最適化と呼ばれる処理によって重みやバイアスを更新する
4. 上記の処理を事前に定めた回数だけ繰り返す

この手順を実際のコードとして表現したものが以下です。

```
net = Net()  # ニューラルネットワークのインスタンスを生成

criterion = nn.MSELoss()  # 損失関数
optimizer = torch.optim.SGD(net.parameters(), lr=0.003)  # 最適化アルゴリズム

EPOCHS = 2000  # 上と同じことを 2000 回繰り返す
for epoch in range(EPOCHS):
    optimizer.zero_grad()  # 重みとバイアスの更新で内部的に使用するデータをリセット
    outputs = net(X_train)  # 手順 1：ニューラルネットワークにデータを入力
    loss = criterion(outputs, y_train)  # 手順 2：正解ラベルとの比較
    loss.backward()  # 手順 3-1：誤差逆伝播
    optimizer.step()  # 手順 3-2：重みとバイアスの更新

    if epoch % 100 == 99:  # 100 回繰り返すたびに損失を表示
        print(f'epoch: {epoch+1:4}, loss: {loss.data}')

print('training finished')
```

学習を行うコード

コードには上記手順に対応する箇所にその旨のコメントを記しています。手順 1 の前に「重みとバイアスの更新で内部的に使用するデータをリセット」していますが、これはそういう作法だと考えておいてください。また、繰り返しを 100 回行うたびに、その時点で計算結果と正解ラベルとの誤差がどのくらいになったか（損失：loss.data）を表示するようにもしてあります。簡単には、「誤差」がゼロに近いほど、計算結果と正解ラベルの差が少ないと考えられます。つまり、ここでの学習とは誤差がゼロに近いものになるように、重みやバイアスを更新していくということでもあります。

実際に試してみた結果を以下に示します。

```
net = Net()  # ニューラルネットワークのインスタンスを生成

criterion = nn.MSELoss()  # 損失関数
optimizer = torch.optim.SGD(net.parameters(), lr=0.003)  # 最適化手法

EPOCHS = 2000  # 上と同じことを2000回繰り返す
for epoch in range(EPOCHS):
    optimizer.zero_grad()  # 重みとバイアスの更新で内部的に使用するデータをリセット
    outputs = net(X_train)  # 手順1：ニューラルネットワークにデータを入力
    loss = criterion(outputs, y_train)  # 手順2：正解ラベルとの比較
    loss.backward()  # 手順3-1：誤差逆伝播
    optimizer.step()  # 手順3-2：重みとバイアスの更新

    if epoch % 100 == 99:  # 100回繰り返すたびに損失を表示
        print(f'epoch: {epoch+1:4}, loss: {loss.data}')

print('training finished')
```

```
epoch:  100, loss: 0.6393837928771973
epoch:  200, loss: 0.5842536687850952
epoch:  300, loss: 0.5612550973892212
epoch:  400, loss: 0.5348739624023438
epoch:  500, loss: 0.5019711852073669
epoch:  600, loss: 0.4604303240776062
epoch:  700, loss: 0.40949442982673645
epoch:  800, loss: 0.35218316316604614
epoch:  900, loss: 0.29507753252983093
epoch: 1000, loss: 0.24343958497047424
epoch: 1100, loss: 0.19943009316921234
epoch: 1200, loss: 0.16361494362354279
epoch: 1300, loss: 0.13570287823677063
epoch: 1400, loss: 0.1147782951593399
epoch: 1500, loss: 0.09958261251449585
epoch: 1600, loss: 0.08879823237657547
epoch: 1700, loss: 0.08124153316020966
epoch: 1800, loss: 0.0759531706571579
epoch: 1900, loss: 0.07221029698848724
epoch: 2000, loss: 0.0694972425699234
training finished
```

実行結果

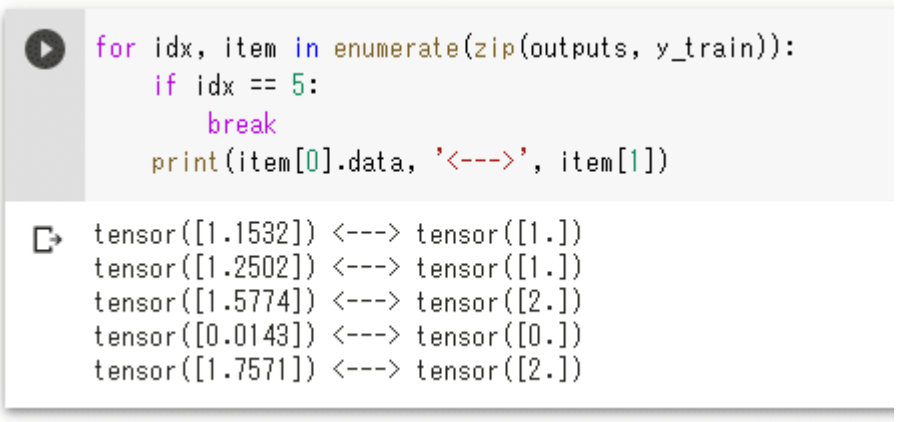
損失が徐々に減少していることが分かるでしょうか。最終的には損失が 0.07 程度になりました。100 回の学習を行った時点の損失が 0.6 を越えていたことを考えるとずいぶんと推測結果と正解ラベルの値が近づいたことが分かります（この数値が意味するところについては、次回以降に説明をします）。

実際に、（最後に学習を行ったときの）計算結果を最初から 5 つ、対応する正解ラベルと共に表示してみましょう。

```
for idx, item in enumerate(zip(outputs, y_train)):
    if idx == 5:
        break
    print(item[0].data, '<--->', item[1])
```

学習後の計算結果と正解ラベルの比較

このコードを実行した結果は次の通りです（読者が上のコードを実際に試したときには、これとは異なる結果になるでしょう）。



実行結果

どうでしょう。学習前にニューラルネットワークが算出した結果と比べると、かなり近しい値が得られているようです。ただし、ここで欲しいのは 0、1、2 のいずれかの整数値です。上の結果を見るに、小数点以下 1 桁目で四捨五入をするといい感じに整数値が得られそうです。というわけで、ここでは計算結果を含んだ配列の各要素に 0.5 を加算したものを整数型に変換するコードを書いてみましょう。

```
predict = (outputs + 0.5).int()
for idx, item in enumerate(zip(predict, y_train)):
    if idx == 5:
        break
    print('output:', item[0], ', label:', item[1])
```

浮動小数点数値の計算結果を整数値に変換するコード

「outputs + 0.5」という部分を見て「へ?」と思った方もいるかもしれません。これは、「ouputs の全要素に 0.5 を加算する」という処理を行います。NumPy が提供する多次元配列では「配列の各要素に加算／減算する」ような処理をこのように分かりやすく記述できるようになっています（この機能を NumPy では「ブロードキャスト」と呼んでいます）。PyTorch のテンソルは NumPy の配列とは異なるものですが、こうした使い方は PyTorch でも可能です。後は、変換後のデータ（predict）と正解ラベルの先頭から 5 つの要素を表示するコードになっています。

実行結果を以下に示します。

```
predict = (outputs + 0.5).int()
for idx, item in enumerate(zip(predict, y_train)):
    if idx == 5:
        break
    print('output:', item[0], ', label:', item[1])
```

```
output: tensor([1], dtype=torch.int32) , label: tensor([1.])
output: tensor([1], dtype=torch.int32) , label: tensor([1.])
output: tensor([2], dtype=torch.int32) , label: tensor([2.])
output: tensor([0], dtype=torch.int32) , label: tensor([0.])
output: tensor([2], dtype=torch.int32) , label: tensor([2.])
```

実行結果

数値部分にのみ着目すると、どうもいい感じです。最後に、112 個の入力データから計算した結果がどれだけ正解ラベルと一致しているかを確認してみましょう。

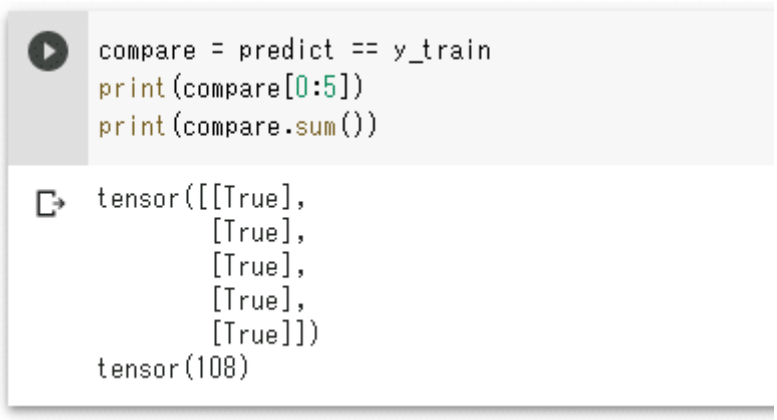
```
compare = predict == y_train
print(compare[0:5])
print(compare.sum())
```

整数値に変換した計算結果と正解ラベルを比較して、正しい結果が何個あったかを確認

ここでもコードを見てギョッとする人がいるかもしれません。「predict == y_train」というのは、「predict（変換後の整数値を格納した配列）と y_train（正解ラベル）の各要素を比較して、それらが等しければ対応するインデックス位置の値を True に、等しくなければ False とする配列を返す」という処理をします。つまり、[True, True, True, False, True, ……] のような配列が compare に代入されます。上のコードではその先頭から 5 つの要素を表示するようにしています。後から示す実行結果で今述べたような配列が得られていることを確認してください。

「compare.sum()」というのは、その配列の要素の総和を求める処理です。ただし、Python では True は 1 と、False は 0 と見なされることを思い出してください。よって、「総和を求める」＝「True の要素の数を数える」ということです。

では、実行結果を示します。



```
compare = predict == y_train
print(compare[0:5])
print(compare.sum())
```

```
tensor([[True],
        [True],
        [True],
        [True],
        [True]])
tensor(108)
```

実行結果

どうでしょう。「108」個が正しいことが分かりましたね。112 個のうちの 108 個が正しいので、正解率 は 96.4%といえます。なかなかの精度といってもよいでしょう。取りあえず、ここではこれでニューラルネットワークは完成したものとしましょう。

次に、完成したニューラルネットワークの精度を確認します。上で確認したのは、あくまでも「訓練に使用したデータに対する正解率」です。既に述べた通り、訓練に使用していない未知のデータに対しても、このニューラルネットワークが適切な値を返せなければ、このニューラルネットワークには意味がありません。そのため、本稿の冒頭ではデータを訓練データとテストデータに分割したのでした。

そこで、上と同様な手順で、ニューラルネットワークにテストデータを入力して、その値をテストデータの正解ラベルと比較してみましょう。

実際のコードは以下のようになります。

```
outputs = net(X_test)

predict = (outputs + 0.5).int()
compare = predict == y_test

print(f'correct: {compare.sum()} / {len(predict)}')
for value, label in zip(predict, y_test):
    print('predicted:', iris.target_names[value.item()], '<--->',
        'label:', iris.target_names[int(label.item())])
```

テストデータを使って、あやめの品種を推測するコード

ここでは学習を行うわけではないので、for 文によるループで誤差を計算して、重みやバイアスを更新するといった処理は必要ありません。そのため、Net クラスのインスタンスにテストデータを入力し、得られた計算結果を小数点 1 桁目で四捨五入して整数値化したら、それをテストデータ用の正解ラベルと比較するだけです。最後には全てのデータを出力するコードも書いてあります（最後の行で使っている item メソッドは、配列の値を取り出すものです）。

実行結果を以下に示します（品種の推測結果と正解ラベルを比較した出力は途中で省略しています）。もちろん、読者が上のコードを実際に試したときには、これとは異なる結果になるでしょう。

```
outputs = net(X_test)

predict = (outputs + 0.5).int()
compare = predict == y_test

print(f'correct: {compare.sum()} / {len(predict)}')
for value, label in zip(predict, y_test):
    print('predicted:', iris.target_names[value.item()], '<--->',
        'label:', iris.target_names[int(label.item())])
```

```
correct: 38 / 38
predicted: versicolor <---> label: versicolor
predicted: setosa <---> label: setosa
predicted: setosa <---> label: setosa
predicted: virginica <---> label: virginica
predicted: virginica <---> label: virginica
predicted: virginica <---> label: virginica
predicted: virginica <---> label: virginica
predicted: versicolor <---> label: versicolor
```

実行結果

38 個（150 個の 3/4）のテストデータのうち、38 個が正しいということで、100%の正解率となりました（ちょっとビックリしました。が、学習に使用したデータでは完璧な推測ができたわけではないことも思い出してください。つまり、このニューラルネットワークはどんなデータに対しても常に正しく判断できるとは限らないことには注意しましょう）。どうやらこのニューラルネットワークは、訓練データ以外のデータが入力されても、それがどんな品種のあやめなのかをある程度は正しく判断できることが確認できました。

今回は駆け足で、ニューラルネットワークを作成する手順を眺めてきました。ニューラルネットワークを作成し、それを学習させ、その精度を評価するまでには、データセットの準備、ニューラルネットワークの定義、学習、評価というステップを踏みます。今回はその大枠を眺めました。次回は今回の書いたコードを基に、各段階について少し詳しく見ていくことにしましょう。

データセット、多次元配列、多クラス分類

前回のコードを基に、データセットと多次元配列、データセットを分割する意味、出力層に 3 つのノードを持たせた場合のあやめの分類などについて取り上げます。

（2020 年 04 月 10 日）

前回はあやめの品種を推測するニューラルネットワークを作りながら、その手順を駆け足で眺めました。今回からは数回にわたって、そのコードを少し詳しく見たり、コードに手を入れたりしながら、ニューラルネットワークなどについての理解を深めていきましょう。まずデータセットに関連する事柄を取り上げます。

データセットと多次元配列

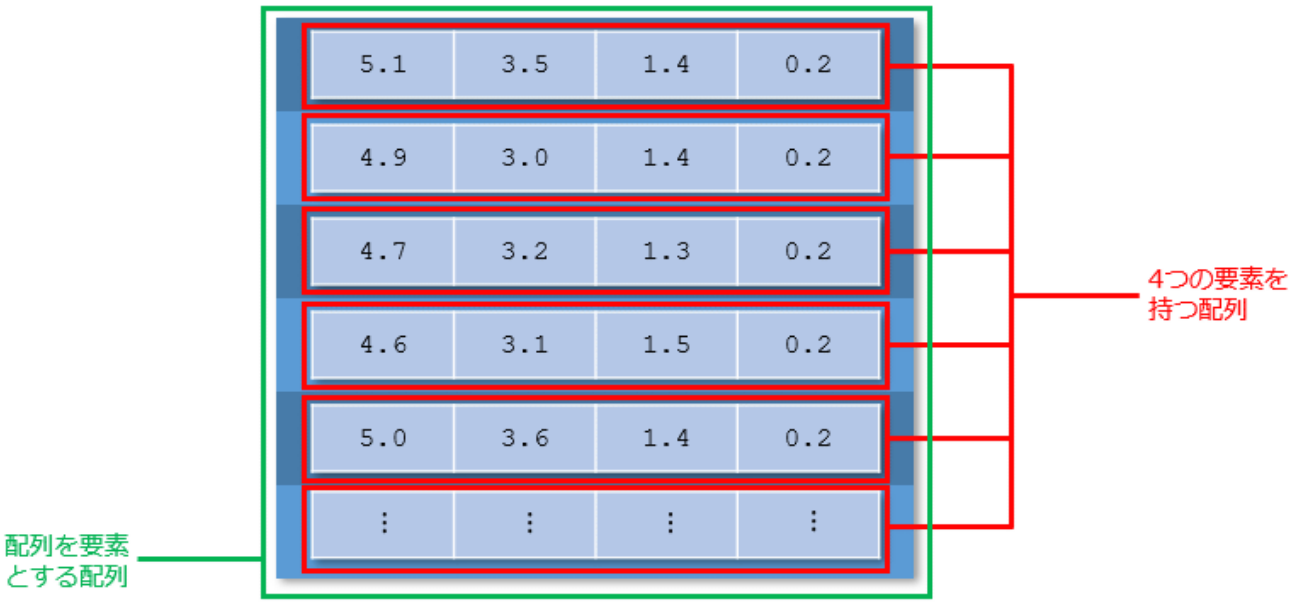
前回に見たあやめのデータセットは、scikit-learn が提供するものでした。以下では、このデータセットを少し詳しく見ながら、データセットとその扱いについて取り上げます。

あやめのデータセットは次のコードで読み込みました。

```
from sklearn.datasets import load_iris
iris = load_iris()
```

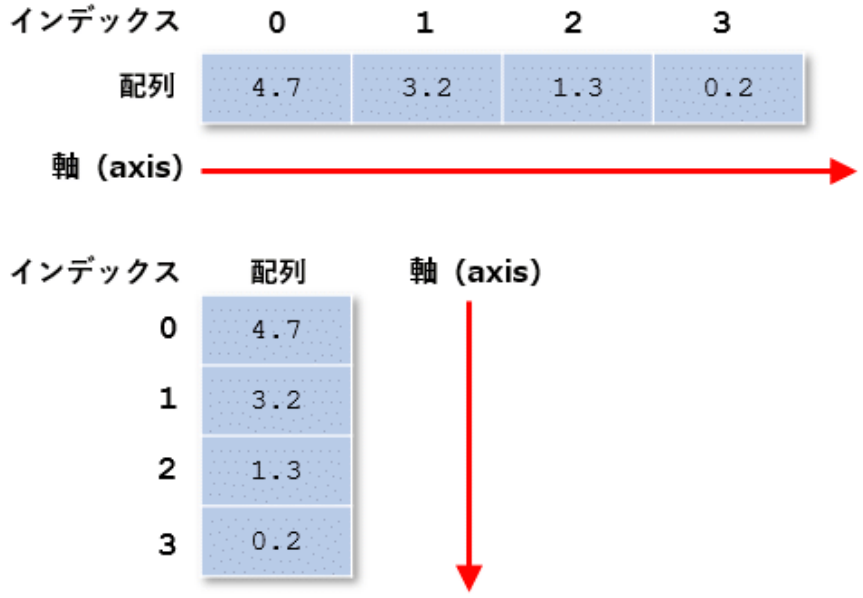
あやめのデータセットを読み込むコード

前回も述べましたが、データセットには一定の形式で複数のデータが格納されています。このうちの、data 属性は「4 つの浮動小数点数値を要素とする配列」を要素とする配列（配列の配列）でした。



data 属性は「配列の配列」

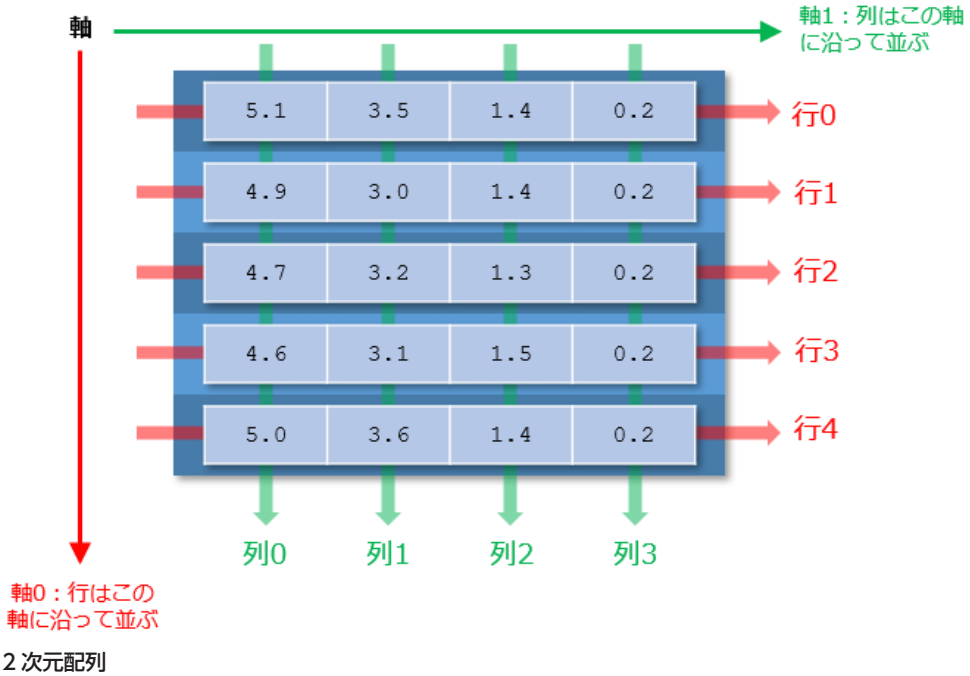
最もシンプルな配列は、その要素が一方向に並べられたものです。



最もシンプルな 1 次元の配列

その要素にはインデックス（配列内での特定の要素がどこにあるかを示す値）を 1 つ指定することでアクセスできます。このとき、配列が並んでいる方向のことを「軸」（axis）と呼ぶことがあります。また、軸の数を「次元数」「階数」などと呼びます。最もシンプルな配列は、軸が 1 つで、次元数も 1 です。

これに対して、配列の配列は、1 次元の配列がもう 1 つの軸に沿って並べられているものと考えられます。すなわち、配列の配列は軸が 2 つある（次元数が 2 の）データ構造です。このことから、今述べたような「4 つの浮動小数点数値を要素とする配列」を要素とする配列は、よく「2 次元配列」と呼ばれます。数学用語を使って「行列」とも呼ばれます。



2 次元配列

→目次に戻る

上の図に示したように、2 次元配列では縦軸（軸 0）に沿って行の要素が並べられ、横軸（軸 1）に沿って列の要素が並べられます。2 次元の配列で特定の要素にアクセスするには、インデックスを 1 つ、または 2 つ指定します。

1 つ指定したときには、配列を要素としている（軸 0 に沿って並んでいる外側）配列（行）のインデックスが指定されたものとして扱われます。つまり、これにより外側の要素の配列となっている配列（今回の例なら、個々のあやめのデータにアクセスすることになります）。

インデックスを2つ指定すると、それは配列の要素となっている配列の特定の要素にアクセスすることになります（今回の例であれば、個々のあやめのデータを構成するがく片の長さ／幅、花弁の長さ／幅のいずれかの要素にアクセスすることになります）。

あやめのデータセットでは、そのがく片の長さと幅、花弁の長さと幅の 4 つの数値をひとまとめにした配列を要素とする配列を扱うだけなので、2 次元の配列で十分でしたが、画像のようにもともとが 2 次元の構造を持つデータを大量に含むデータセットでは 3 次元の配列が出てくることもあるでしょう（ただし、こうしたデータは画像という 2 次元のデータを 1 次元のデータに変換して使用することもよくあります）。

機械学習やニューラルネットワークの世界では、重みやバイアス、入力されるデータ、計算結果の出力など、さまざまな場面で変数を大量に使用します。また、実際の計算処理では、行列の操作や演算も頻繁に登場します。そのため、多数の変数を一括して、高速に処理できる科学計算ライブラリである NumPy や、それを基に発展したさまざまなフレームワークが使われるのは前回にもお話しした通りです。

また、PyTorch などのフレームワークでは、配列などのデータ構造のことを「テンソル」と総称することも前回に紹介した通りです。また、こうしたデータ構造の中でも 1 次元配列は「ベクトル」とか「ベクター」と、2 次元配列のことは「行列」などと呼ぶこともよくあります。3 次元配列など、次元（階数）が 3 を超えるものについては特別な名称はなく、「テンソル」と呼ぶのが一般的です。これらの用語にも徐々に慣れていく必要があるでしょう。

用語に慣れるのと同時に、テンソルの扱いにも慣れていく必要もあります。そこで、PyTorch のテンソルの扱いを少しだけ見ておきましょう。詳しくは「PyTorch のテンソル&データ型のチートシート」を参照してください。

```
import torch

X = torch.tensor(iris.data)  # 入力データを PyTorch のテンソルに変換

# X のサイズを調べる
print(X.size())
print(X.shape)  # NumPy と同様な書き方
print(len(X))  # 外側の配列の要素数（行数）
print(X.dim())  # テンソルの次元数
print(X.ndim)  # NumPy と同様な書き方

# テンソルの要素のデータ型を調べる
print(X.dtype)

# テンソルの要素へのアクセス
print(X[0])  # 先頭行
print(X[0][0])  # 先頭行の先頭要素
print(X[1:3])  # 第 1 行と第 2 行
print(X[1:3, 1:3])  # 第 1 行と第 2 行の 1 列目と 2 列目の要素
print(X[1:3][1:3])  # この書き方は意味が異なる点に注意
```

テンソルの扱い

上のコードを実行した結果は次の通りです。

```python
import torch

X = torch.tensor(iris.data)  # 入力データをPyTorchのテンソルに変換

# Xのサイズを調べる
print(X.size())
print(X.shape)  # NumPyと同様な書き方
print(len(X))  # 外側の配列の要素数（行数）
print(X.dim())  # テンソルの次元数
print(X.ndim)  # NumPyと同様な書き方

# テンソルの要素のデータ型を調べる
print(X.dtype)

# テンソルの要素へのアクセス
print(X[0])  # 先頭行
print(X[0][0])  # 先頭行の先頭要素
print(X[1:3])  # 第1行と第2行
print(X[1:3, 1:3])  # 第1行と第2行の1列目と2列目の要素
print(X[1:3][1:3])  # この書き方は意味が異なる点に注意
```

```
torch.Size([150, 4])
torch.Size([150, 4])
150
2
2
torch.float64
tensor([5.1000, 3.5000, 1.4000, 0.2000], dtype=torch.float64)
tensor(5.1000, dtype=torch.float64)
tensor([[4.9000, 3.0000, 1.4000, 0.2000],
        [4.7000, 3.2000, 1.3000, 0.2000]], dtype=torch.float64)
tensor([[3.0000, 1.4000],
        [3.2000, 1.3000]], dtype=torch.float64)
tensor([[4.7000, 3.2000, 1.3000, 0.2000]], dtype=torch.float64)
```

実行結果

気を付けてほしいのは、最後の 2 行の出力です。どちらも行列のスライスを取得するコードですが、取り出され方が異なります。「X_train[1:3, 1:3]」は行列の第 1 行と第 2 行から第 1 列と第 2 列の要素を取り出すものです。一方、「X_train[1:3][1:3]」は行列の第 1 行と第 2 行からなる配列を取り出した上で、その第 1 行と第 2 行を取得するものです（インデックスはゼロ始まりなので、実際には第 1 行だけが得られます）。その上にある出力と比較すると、どのようにデータが取り出されているかが分かるでしょう。

今回は、こうした操作が出てくることはあまりありませんが、後続の回で、必要に応じて紹介していくことにします。

データセットの分割について

前回は、読み込んだデータセットを学習と評価（テスト）に使用する目的で 2 つに分割しました。

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target)
```

データセットを分割するコード

上のコードではデータセットを 2 つに分割しています。これは前回も見たように、ニューラルネットワークでモデルの学習を行い、その評価（検証／テスト）を行うためです。しかし、実際に自分でニューラルネットワークを作成していると分かりますが、学習を行う際には試行錯誤がつきものです。例えば、ニューラルネットワークのノードの数を変更したり、層を増やしたり減らしたり、活性化関数を変更したりということが発生します（こうした要素を「ハイパーパラメーター」と呼びます）。

このような調整を行う前には、ある時点で作成した（完成前の）ニューラルネットワークモデルがどの程度の精度を持っているかを調べる必要があります。精度を調べて、ハイパーパラメーターを調整して、また学習をやり直して、その精度を評価して……という過程を経ながら、最適なハイパーパラメーターを決定する場合もありますし、あるいは異なるハイパーパラメーターを使って学習させたニューラルネットワークモデルを幾つか用意して選ぶといった場合もあるでしょう。このように、試行錯誤の段階では、訓練データと、暫定的に作成したモデルの精度を検証するデータの 2 つのデータが必要になります（前回は単純に scikit-learn が提供する train_test_split 関数を用いて分割していましたが、実際には訓練データと評価データを分割するにはさまざまな手法があります）。

学習と評価を繰り返して完成したモデルは学習データと評価データに対してはよい精度で結果を算出できるようになっています。一方、それが本当に汎用的に使えるかどうか（すなわち、未知のデータについてもよい精度で結果を算出できるかどうか）をテストするためには、今述べた 2 つのデータとは独立したデータ（テストデータ）が必要です。

こうしたことから、データセットを次の 3 つに分割するというのが最近では一般的になっています。

- 訓練データ：ニューラルネットワークでの学習（モデル内の重みやバイアスの更新）に使われる
- 評価データ：学習後のモデルの性能を検証するために使われる
- テストデータ：最終的に完成したモデルが汎用的に使えるかどうかをテストするのに使われる

前回は学習時のさまざまなパラメーターは決め打ちとして、試行錯誤することなく、ニューラルネットワークのモデルに学習をさせ、一度の学習でモデルを完成させたので、2 つに分割すれば十分でした。

2 つ（あるいは 3 つ）に分割したデータセットは、学習やその後の評価、テストで使われます。データセットのデータをそのまま素直に扱えれば問題はないでしょうが、前回も見たようにフレームワークに合わせたデータ型の変換が必要になることもあるでしょうし、作成するモデルにうまく適合するようにデータを変換しなければならないこともあります。そこで、次ページでは、前回に見たニューラルネットワークモデルを少し変更して、データセットに含まれるデータやモデルの計算結果の扱いや解釈がどう変わるかを見てみます。

変数名について

上のコード例では、データセットを分割して、「X_train」「y_test」などの変数に代入していました。ここでこれらの変数名について少しだけ話をしておきます。

数学の世界では、関数とその値をよく「y=f(x)」と表します。これは「f」という関数（function）に「x」という変数が持つ値を与えると、その結果が「y」になることを意味します。ここで、これから作成するモデルを関数として考えると、「入力→関数（モデル）→出力」のようにも書き表せます。このとき入力するデータ（あやめの特徴を示す 4 つの値）は前述の「x」に、出力（モデルにより算出されたあやめの種類）は「y」に相当することはお分かりでしょう。

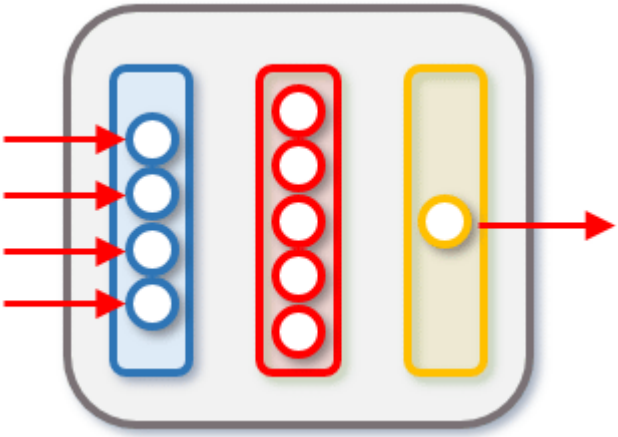
このことに合わせて、ここでは学習やテストのときにモデルに入力する値は「X」で、その出力と比較する正解ラベルは「y」で始まるような名前としています。「X」が小文字ではなく大文字になっているのは、入力するものがデータセット、つまり『「1 個以上の値で構成される配列」を要素とする配列』＝「行列」であり、数学では、行列は大文字で表現することからこのような命名方法になっています。

その後に続く「train」「test」というのは、読んで字のごとく、それが学習（訓練、train）で使われるか、できたモデルの評価（テスト、test）で使われるかを意味します。これら 2 つの要素をアンダースコア「_」でつなげたものをここでは変数名として使っています。

もちろん、これ以外の命名法もあるでしょう。例えば、「X_train」ではなく「train_data」としたり、「y_train」ではなく「train_label」としたりすることが考えられます。こちらの方法でも、変数の値が訓練データであること、訓練に使用する正解ラベルであることが分かります。いずれにせよ、その変数が何のためのデータを参照するものかが分かる名前にすることが大事です。

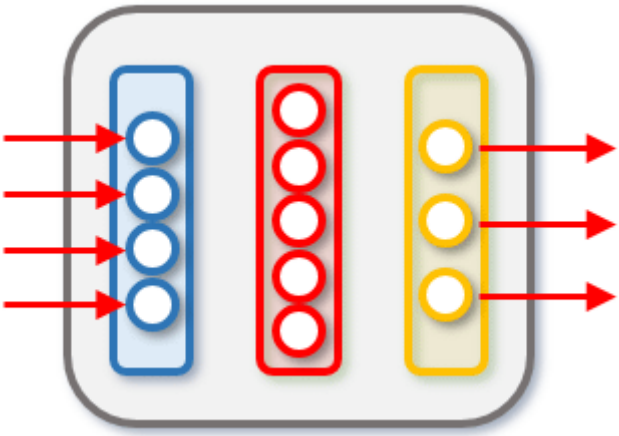
出力層に 3 つのノードを持たせて、あやめの分類を行う

既に述べた通り、データセットに格納されるデータは一定の形式を持っています。ニューラルネットワークにデータを入力したときに算出されるデータにも一定の形式があります。前回作成したネットワークでは、入力層が 4 つのノード、隠れ層が 5 つのノード、出力層が 1 つのノードを持っていて、ニューラルネットワークモデルに 4 つのデータを入力すると、値が 1 つ算出されるようになっていました（実際には、2 次元配列となっているデータセットをニューラルネットワークモデルに入力すると、「計算結果を唯一の要素とする配列」を要素とする配列が得られていたことも思い出してください）。



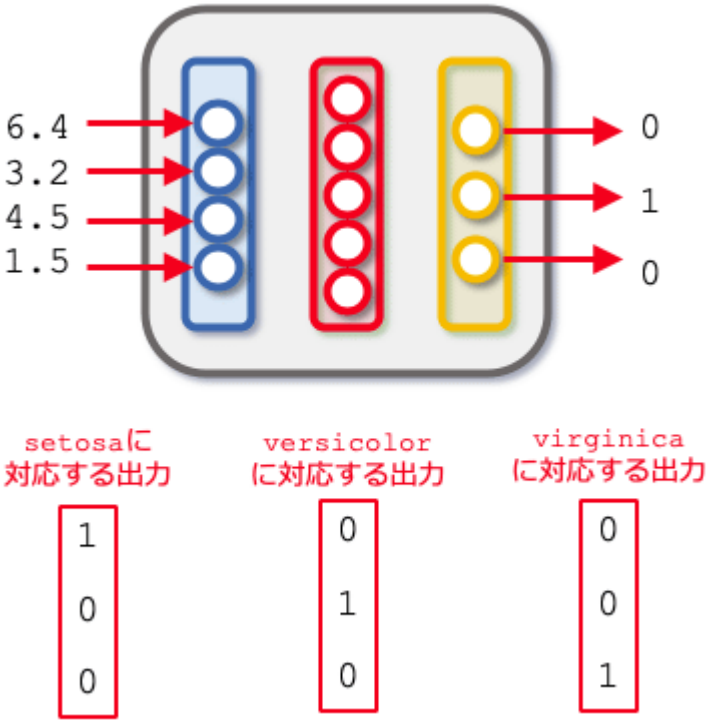
前回作成したニューラルネットワーク（ノードをつなぐエッジは省略）

ここで、ネットワークを少しだけ変更して、出力層にノードが 3 つあるものを考えてみましょう。



今回作成するニューラルネットワーク（ノードをつなぐエッジは省略）

なぜ出力が 3 つかというと、推測するあやめの品種が 3 種類だからです。出力層から各要素の値が True ／ False（1 ／ 0）である配列が出力されるとして、例えば setosa であれば [1, 0, 0] と、versicolor であれば [0, 1, 0] と、virginica であれば [0, 0, 1] と出力するようにすることで、3 種類のあやめを分類できます。



このニューラルネットワークモデルへの入力と出力の例

あやめの品種の推測のように、幾つかの種類（クラス）に分類できるデータが多数あるときに、ニューラルネットワークモデルにそれらの種類を推測させることを「多クラス分類」と呼びます（クラスが 3 つ以上ある場合。あやめの品種は 3 つなので、多クラス分類です。2 つの種類のどちらであるかを分類することは「2 クラス分類」といい、多クラス分類とは区別されています）。このときには、上の図に示したように出力層にはクラスの数に合致するノードを持たせることがよくあります。

このモデルでは、あやめの品種（setosa、versicolor、virginica）に対応するそれぞれのノードから 1（に近い値）が、それ以外のノードからは 0（に近い値）が出力されるように学習を行うことで、あやめの分類を可能にします。

ところで、このデータセットの target 属性には整数値が格納されていて、それらは次のような意味を持っていました。

- 0：対応するデータは setosa である
- 1：対応するデータは versicolor である
- 2：対応するデータは virginica である

これらの値（正解ラベル、教師データ）と、ニューラルネットワークモデルの出力（推測結果）の関係を整理すると次のようになります。

- setosa のデータが入力された場合：[1, 0, 0] に近い値が出力されればよい（正解ラベルの値は 0）
- versicolor のデータが入力された場合：[0, 1, 0] に近い値が出力されればよい（正解ラベルの値は 1）
- virginica のデータが入力された場合：[0, 0, 1] に近い値が出力されればよい（正解ラベルの値は 2）

このとき、正解ラベルは整数値ですが、これは「出力された配列のインデックス」として解釈できます。すると、理想的な出力が得られたときには、正解ラベルが示すインデックス位置にある要素の値だけが 1（に近い値）となります。逆に、あまりよろしくない値が得られたときには、正解ラベルが示すインデックス位置にある要素の値は 1 からは外れた値になります（実際には、Softmax 関数を使うことで、配列の要素が 0 と 1 の間に収まるようにしていますが、これについては後述の回で取り上げます）。

品種	モデルからの理想的な出力	対応する正解ラベル
setosa	[1, 0, 0]	0
versicolor	[0, 1, 0]	1
virginica	[0, 0, 1]	2

ニューラルネットワークモデルの計算結果と正解ラベルの対応

このことを利用して、例えば versicolor に対応するデータが入力されたら [0, 1, 0] になるべく近い値（[0.04, 0.9, 0.06] など）が得られるように（正解ラベルの値は 1）、重みやバイアスを更新することが可能です。つまり、3 要素の配列と正解ラベルを比べて、正解ラベルをインデックスとして見たときに、対応するインデックス位置の値をなるべく大きく、それ以外はなるべく小さくするようなニューラルネットワークモデルにするわけです。

コードの詳細な説明は割愛しますが、実際にこれを行うコードを以下に示します。

```
from sklearn.datasets import load_iris
iris = load_iris()

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target)

import torch

X_train = torch.from_numpy(X_train).float()
X_test = torch.from_numpy(X_test).float()
y_train = torch.from_numpy(y_train).long()
y_test = torch.from_numpy(y_test).long()
```

```
from torch import nn

INPUT_FEATURES = 4
HIDDEN = 5
OUTPUT_FEATURES = 3

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(INPUT_FEATURES, HIDDEN)
        self.fc2 = nn.Linear(HIDDEN, OUTPUT_FEATURES)
        self.softmax = torch.nn.Softmax(dim=1)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        x = self.softmax(x)  # softmax 関数で配列の要素の総和が 1 になるように変換
        return x

net = Net()

criterion = nn.CrossEntropyLoss()  # 交差エントロピー損失を求める関数を利用
optimizer = torch.optim.SGD(net.parameters(), lr=0.3)

EPOCHS = 2000
for epoch in range(EPOCHS):
    optimizer.zero_grad()
    outputs = net(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    if epoch % 100 == 99:
        print(f'epoch: {epoch+1:4}, loss: {loss.data}')
```

```
print('training finished')

for idx, item in enumerate(zip(outputs, y_train)):
    if idx == 5:
        break
    print(item[0], item[1])

values, predicted_idx = torch.max(outputs, 1)
print((predicted_idx == y_train).sum())
```

出力層に 3 つのノードを持たせて、あやめの分類を行うニューラルネットワークモデル（正解ラベルを出力層の 3 要素の配列に対するインデックスとして使用する方法）

基本的な構造は前回に紹介したものと同様ですが、出力層のノード数が 3 になっていることと（強調書体とした「OUTPUT_FEATURES = 3」行）、これにより出力が 3 要素の配列（の配列）となるため、クラスのコードや学習の仕方を少々変更しているのが大きな変更点です。具体的には、交差エントロピーと呼ばれる手法で、モデルの計算結果と正解ラベルの誤差を求めているところ、出力層で torch.nn.Softmax クラスを使っているところ、最後に計算結果からニューラルネットワークがあやめの品種を推測した結果を取り出す部分などがそうですが、これらについては今回は詳しい説明は省略します。

実行結果を以下に示します。

```
    if idx == 5:
        break
    print(item[0], item[1])

values, predicted_idx = torch.max(outputs, 1)
print((predicted_idx == y_train).sum())
```

```
epoch:  100, loss: 1.0075467824935913
epoch:  200, loss: 0.8594541549682617
epoch:  300, loss: 0.8081828355789185
epoch:  400, loss: 0.7696171402931213
epoch:  500, loss: 0.7300953269004822
epoch:  600, loss: 0.6916478872299194
epoch:  700, loss: 0.662449836730957
epoch:  800, loss: 0.6428940892219543
epoch:  900, loss: 0.6298216581344604
epoch: 1000, loss: 0.6207440495491028
epoch: 1100, loss: 0.6141586303710938
epoch: 1200, loss: 0.6091873049736023
epoch: 1300, loss: 0.6053061485290527
epoch: 1400, loss: 0.6021895408630371
epoch: 1500, loss: 0.5996271967887878
epoch: 1600, loss: 0.5974783897399902
epoch: 1700, loss: 0.5956464409828186
epoch: 1800, loss: 0.5940622687339783
epoch: 1900, loss: 0.592675507068634
epoch: 2000, loss: 0.5914487242698669
training finished
tensor([0.0084, 0.9898, 0.0018], grad_fn=<SelectBackward>) tensor(1)
tensor([1.3333e-05, 6.8758e-03, 9.9311e-01], grad_fn=<SelectBackward>) tensor(2)
tensor([7.7223e-06, 4.3720e-03, 9.9562e-01], grad_fn=<SelectBackward>) tensor(2)
tensor([1.1942e-04, 4.8935e-02, 9.5095e-01], grad_fn=<SelectBackward>) tensor(1)
tensor([0.0155, 0.9832, 0.0012], grad_fn=<SelectBackward>) tensor(1)
tensor(109)
```

実行結果

「training finished」の後にある for 文による出力は、計算結果と対応する正解ラベルの組です。少し読みにくいのですが、3 つの浮動小数点数値と、最後にある「tensor(……)」にだけ注目してください。前者がもちろんニューラルネットワークによる計算結果（あやめの品種の推測結果）です。このうち、各行の出力の中に、1 にかなり近い値が 1 個だけあり、他の値はほぼ 0 になっていることに注目してください。

例えば、最初の行ではインデックス 1 の要素だけが、1 に近い値（0.9898）になっています。モデルが versicolor に対応する [0, 1, 0] という値におおよそ近い値を推測したということです。これに対応する右側の正解ラベルは「tensor(1)」つまり versicolor です。このことから、このデータについてはうまく推測できていることが分かります。他の行の出力についても同様です、といいたいところですが、for 文の出力の 4 行目では、正解ラベルが「tensor(1)」になっているにも関わらず、出力を見るとインデックス 2 の値が最大になっています。これは推測に失敗していることを表しています。

一番下の行の出力は、112 個の訓練データのうち、正しく推測されたものの数の表示です。ここでは 109 個が一致しているので、上のように間違いはあるものの、よい感じに学習できたと思われます（ここではモデルの評価は省略します）。

最後に、正しい結果が得られたかを計算するときに行っている「values, predicted_idx = torch.max(outputs, 1)」という処理についてだけ、簡単に説明をしておきましょう。これは「outputs の各行（3 要素の配列）から最大値とそのインデックスを調べて、それらをタプルにまとめる」という処理を行っています。代入先の変数の values には最大値を要素とする配列が、predicted_idx にはそのインデックス（0 ～ 2）を要素とする配列が返されます。

ここで重要なのは、値そのものではなく、それが 3 要素の配列の何番目であったかです。例えば、[0.0084, 0.9898, 0.0018] ならインデックス 1 の要素の値が最大なので、対応する y_train の要素が 1 であれば、推測した結果と正解ラベルが等しくなったと考えられます（上の for 文の最初の出力）。そこで、ここではインデックスを格納している predicted_idx だけを用いて、その後の処理を行っています。

次の「(predicted_idx == y_train).sum()」という処理では、インデックスの値を含む配列と y_train の各要素とを比較して、True が幾つあったかを数えるだけです（前回にも似たコードを見ました）。こうすることで、モデルが推測した値と、正解ラベルの比較をしています。

今回は前回に紹介したコードや、それを基に少し変更をしたコードを見ながら、データセットや多次元配列の基本的な考え方、データセットの分割では実際には 3 つに分割することがあること、出力層に 3 つのノードを持たせた場合のあやめの分類について見てきました。最後に紹介したニューラルネットワーククラス（および前回に紹介したクラス）については、次回に詳しく見ていくことにしましょう。

ニューラルネットワークの内部では何が行われている？

あやめの品種を推測するニューラルネットワーククラスの動作を確認しながら、その内部でどんな処理が行われているのかを見ていきましょう。

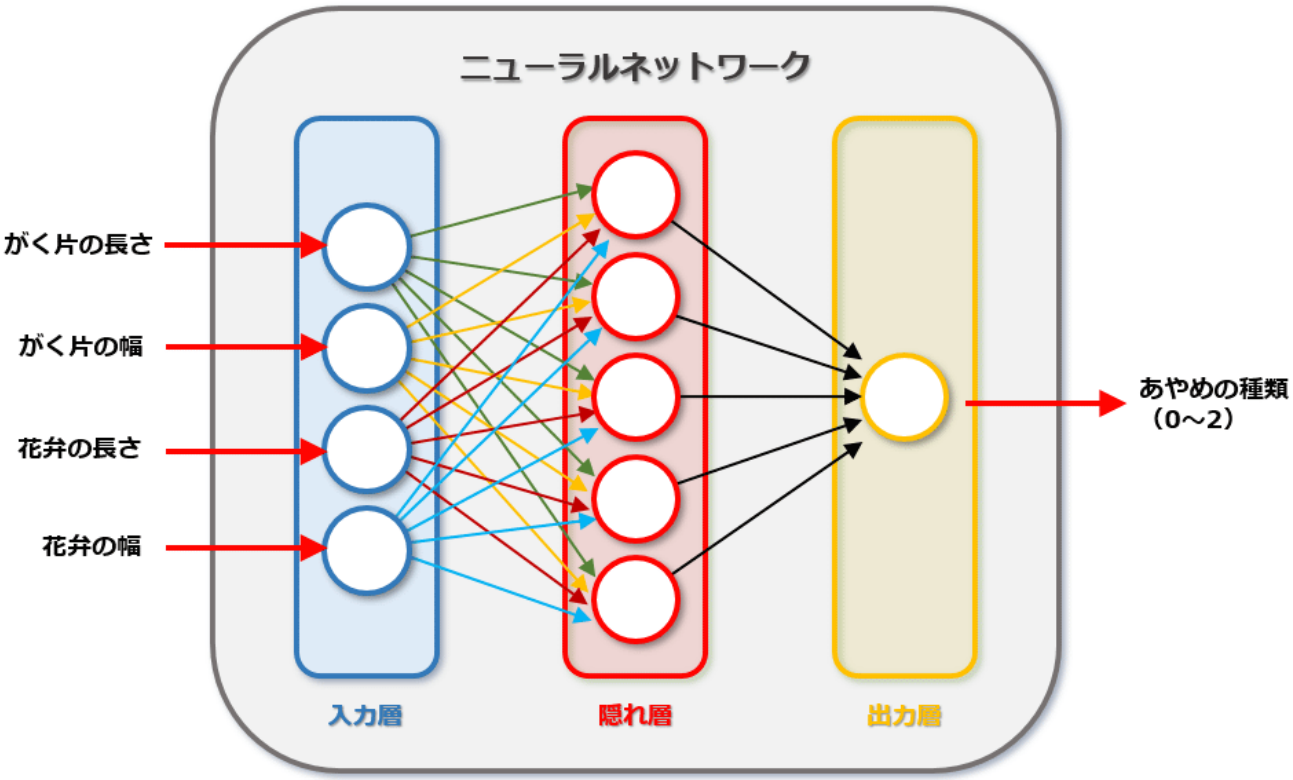
（2020 年 04 月 21 日）

前回はあやめのデータセットをスタート地点として、データセットにまつわるさまざまな話をしました。今回は、ニューラルネットワーククラス（のインスタンス）が実際にはどんな処理をしているのかを見ながら、全結合層、重みとバイアスを使った計算の実際などについて見ていきます。

重みとバイアスはどこにある？

以下では、第 2 回に作成したニューラルネットワーククラスを例に、これが一体どんなことをしているのかを簡単に見ていきましょう。なお、今回のコードはこのリンク先で公開しています。記事を読みながら、実際に実行してみるのもよいでしょう。

念のため、このニューラルネットワークがどんな構造であるかを以下に図示しておきます。



第 2 回で作成したニューラルネットワーク

このときに記述した、データセットを読み込んで、分割するコードと、ニューラルネットワークを表すクラスのコードを以下に示します（ただし、import 文は先頭にまとめて、分割後の要素数を表示するコードなどは削除しました）。

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import torch
from torch import nn

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target)

X_train = torch.from_numpy(X_train).float()
y_train = torch.tensor([[float(x)] for x in y_train])
X_test = torch.from_numpy(X_test).float()
y_test = torch.tensor([[float(x)] for x in y_test])

INPUT_FEATURES = 4
HIDDEN = 5
OUTPUT_FEATURES = 1

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(INPUT_FEATURES, HIDDEN)
        self.fc2 = nn.Linear(HIDDEN, OUTPUT_FEATURES)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        return x
```

データセットを分割するコードと、ニューラルネットワーククラスを定義するコード

ここでは、このコードを基に、少し手を動かしてコードを試しながら、Net クラスがどんなクラスであるのかを試してみましょう。

第 1 回や第 2 回では「学習とは、ニューラルネットワークモデルへの入力に対して、適切な推測値を算出できるように、重みやバイアスを更新していく過程である」といったことを述べましたが、実際のコードにはそんなものは少しも登場していなかったことを疑問に思っていた方もいるでしょう。そこで、まずは実際にそんなものがあるかを確認してみましょう。

```
net = Net()

print('weight')
print(net.fc1.weight)
print('bias')
print(net.fc1.bias)
```

Net クラスのインスタンスのインスタンス変数 fc1 が持つ重みとバイアスを表示するコード

これは Net クラスのインスタンスを生成して、そのインスタンス変数 fc1 が持っている weight 属性と bias 属性を表示するコードです。インスタンス変数の名前から分かるように、weight 属性は fc1（入力層）の重みを、bias 属性はバイアスを表します。

実際に実行してみると、次のような結果になります。ただし、それぞれの値は fc1 に代入される Linear クラスのインスタンスの初期化時にランダムに決定されるので、読者が同じコードを試しても、その結果は以下とは異なるものになるでしょう。

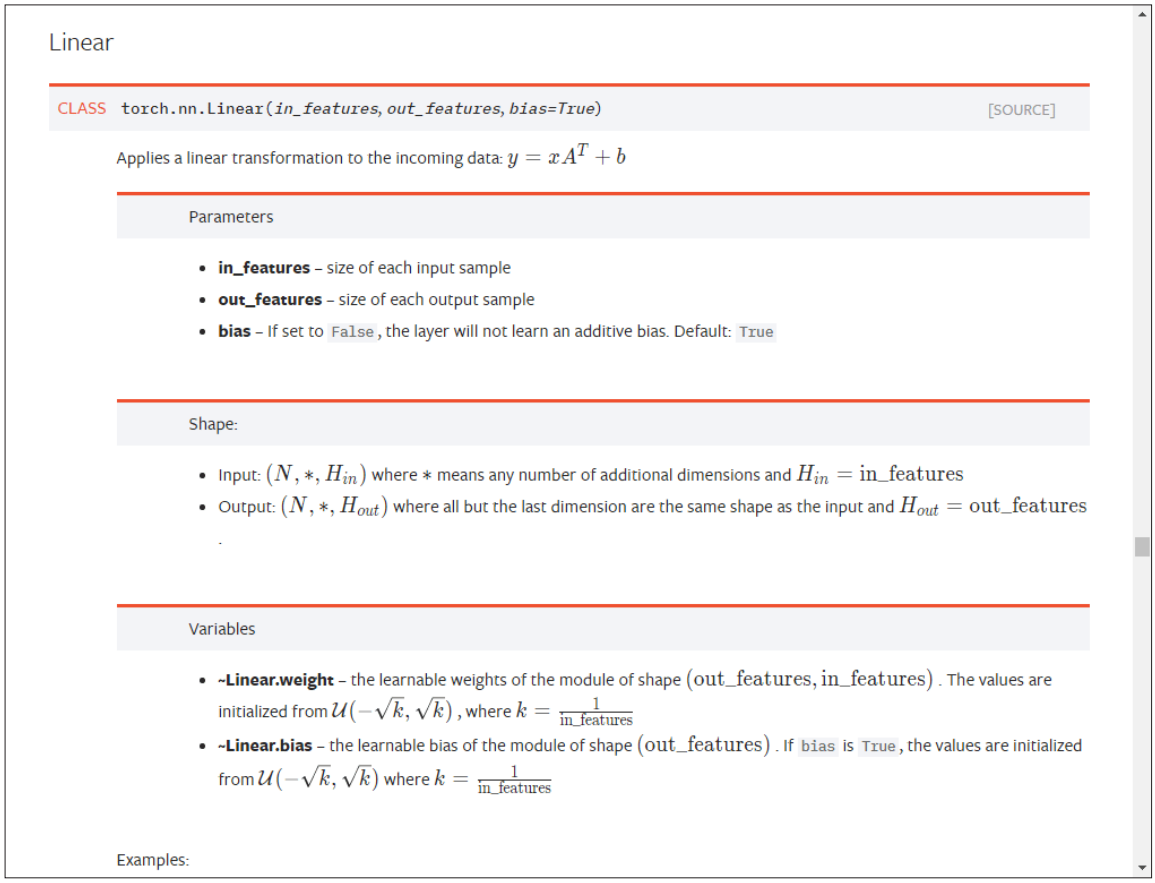
```
net = Net()

print('weight')
print(net.fc1.weight)
print('bias')
print(net.fc1.bias)
```

```
weight
Parameter containing:
tensor([[-0.4764, -0.1864, -0.3762, -0.0953],
        [ 0.2042,  0.2025, -0.2075, -0.1600],
        [-0.4648,  0.0523, -0.1995,  0.1795],
        [-0.3157,  0.3784,  0.2732,  0.4909],
        [ 0.3402, -0.0375,  0.2654, -0.2488]], requires_grad=True)
bias
Parameter containing:
tensor([ 0.2545,  0.0735,  0.3961, -0.0111,  0.1450], requires_grad=True)
```

実行結果

今述べたことからも想像できますが、上のコードをもう一度実行すれば、重みとバイアスの値はまた別のものになります（実行結果は省略）。

重みとバイアスの初期値がどう決まるかは、Linear クラスのドキュメントに記述があります。



Linear

CLASS torch.nn.Linear(*in_features*, *out_features*, *bias=True*) [SOURCE]

Applies a linear transformation to the incoming data: $y = xA^T + b$

Parameters

- **in_features** – size of each input sample
- **out_features** – size of each output sample
- **bias** – If set to `False`, the layer will not learn an additive bias. Default: `True`

Shape:

- Input: $(N, *, H_{in})$ where $*$ means any number of additional dimensions and $H_{in} = \text{in\_features}$
- Output: $(N, *, H_{out})$ where all but the last dimension are the same shape as the input and $H_{out} = \text{out\_features}$.

Variables

- **~Linear.weight** – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **~Linear.bias** – the learnable bias of the module of shape (out_features) . If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

Examples:

Linear クラスのドキュメント

一番の下の方にある weight 属性の説明を見ると、何やら難しそうなことが書いてあります。少し説明しましょう。ここで、インスタンス変数 fc1 に代入される Linear クラスのインスタンスは「nn.Linear(INPUT_FEATURES, HIDDEN)」のようにして生成していました。この INPUT_FEATURES が上図の「in_features」に相当します。

上のコードでは、INPUT_FEATURES の値は「4」なので、k の値は 1/4、その平方根は 1/2 です。そして、上の画像に見られる「U」は「uniform distribution」（一様分布 *1）を意味します。これは、特定の範囲（ここでは -1/2 ～ 1/2）に一様に分布している値からランダムに選ばれたものを使って、重みが初期化されるということです（これはバイアスも同様です）。そう思って、上の実行結果を見ると、今述べたように -1/2 ～ 1/2 の範囲の値が並んでいることが分かると思います。

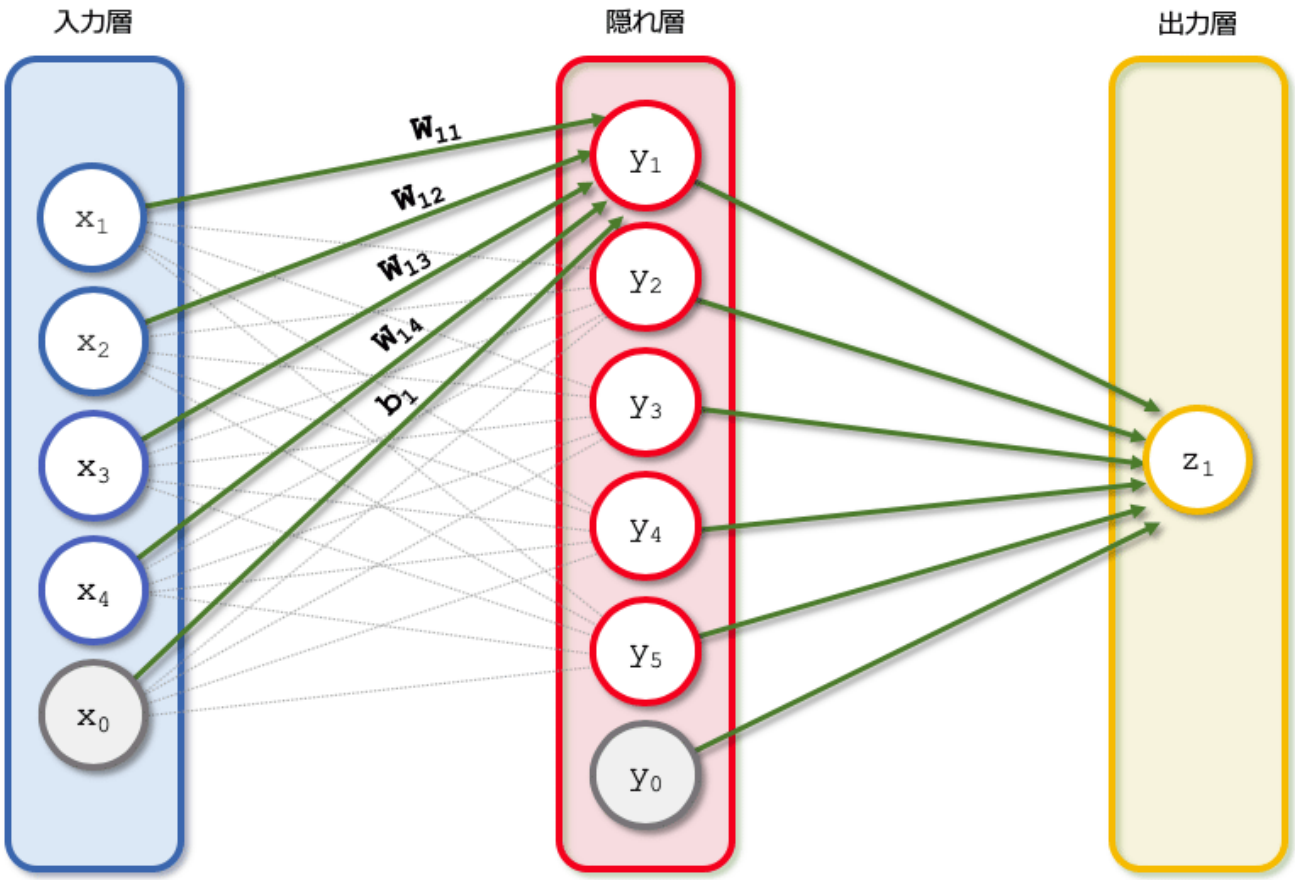
*1 その範囲に含まれる全ての値が、他の値と同じ間隔で並んでいるといった意味です。言い換えると、その範囲にある数値が全て同じ確率で選択されると考えてもよいでしょう。例えば、サイコロを振ったときに出る目は全て 1/6 の確率で選ばれます。よって、サイコロの出目を考えると、「1 ～ 6」は一様分布していると考えられます（実際にサイコロを 6 回振ったときに全ての値が 1 回ずつ出るわけではないことは既にご存じの通りです）。これと同様に、上の例では -1/2 ～ 1/2 の範囲の数値が同じ確率でランダムに選択されると考えられます。

なお、重みをどのような値に初期化するかの方法にはいろいろとあります。ただ、0 や 1 のような一定の値ではなく、何らかのルールに従ったランダムな値に初期化するのが一般的です。また、使用する活性化関数との関係から、相性がよい初期化方法というのもありますが、ここでは詳細な説明は省略します。

何はともあれ、ニューラルネットワークの層を表すインスタンス変数に重みとバイアスが格納されていることはこれで分かりました。これらを使って、ニューラルネットワークのモデルではどんな計算が行われるかを実際に見てみましょう。

重みとバイアスを使った計算の実際

その前に、このニューラルネットワークの構造について振り返りましょう。以下は入力層から出力層までに、このニューラルネットワークで計算される値の流れを示したものです。特に入力層のノードから隠れ層の最初のノード（y1）へと渡される（伝播される）値に着目して、その計算に必要な「重み」と「バイアス」を書き込んであります。



入力層から隠れ層の先頭のノードへ出力される値を決定する処理

ここで注目してほしいことは幾つかあります。まず、隣り合う層の全てのノードがエッジ（ノードをつなぐ線）で接続（結合）されていることです。このような結合のことを「全結合」などと呼びます。

また、入力層には 5 つのノードがありますが、その x1 ～ x4 は入力変数です。つまりこのニューラルネットワークが受け取るがく片の長さ／幅、花弁の長さ／幅はこれらの変数に渡されます。x0 は、次の層に伝播される値を計算するときに使用されるバイアスに対応したものです。この値は通常「1」と考えます（バイアスがある場合）。この値が「1」ならば、バイアスを w11 などの重みと同様に扱おうとしたときに、バイアスの値が常に出力値の計算に使われるようになります（各層からの出力値を計算する際に重みとバイアスを 1 つの行列にまとめられるという効果もありますが、これについてはここでは説明しません）。隠れ層と出力層も同様です。前の層から伝播される値を受け取るノード（と隠れ層ではバイアスに対応するノード）があります。

上の図における wij は、「重み」を意味する「w」に、値を受け取る側のノードの番号（y1 なら「1」）と値を出力する側ノードの番号（x2 なら「2」）を表す 2 つの数値が付加されたものです。つまり、x2 から y1 へ出力される値を計算するときに使用する値は w12 のように表されます。

このように表現すると、隠れ層の最初のノード（y1）が（活性化関数を介して）受け取る値は次のように計算できます（先に述べましたが、多くの場合、x0 は通常 1 であり、「b1×x0」ではなく、単に「b1」のように記述することもよくあります）。なお、実際にはこの計算結果を活性化関数でさらに変換したものが次の層への入力となることには注意してください。

$$w_{11} \times x_1 + w_{12} \times x_2 + w_{13} \times x_3 + w_{14} \times x_4 + b_1 \times x_0$$

この計算は、隠れ層の 1 つのノードについてのみ行ったものであることにも注意が必要です。実際には、残る 4 つのノードについても同様な計算を行うことで、入力層から出力される値が決まります。隠れ層からの出力についても同様です。

今までに見てきた Net クラスのインスタンス変数 fc1 と fc2 は、ある層と次の層がどのように結合されるかを表しています。以下に __init__ メソッドのコードを示します。

```
def __init__(self):
    super().__init__()
    self.fc1 = nn.Linear(INPUT_FEATURES, HIDDEN)
    self.fc2 = nn.Linear(HIDDEN, OUTPUT_FEATURES)
```

__init__ メソッド

例えば、fc1 は「self.fc1 = nn.Linear(INPUT_FEATURES, HIDDEN)」のように初期化されています。これは「前の層（入力層）のノード数が INPUT_FEATURES（4）で、次の層（隠れ層）のノード数が HIDDEN（5）であり、これらが全結合される」ことを意味しています。fc2 であれば「前の層（隠れ層）のノード数は HIDDEN（5）で、次の層（出力層）のノード数が OUTPUT_FEATURES（1）であり、これらが全結合される」となります。ある層と次の層が「全結合」であることは、Linear クラス（torch.nn.Linear クラス）のインスタンスを生成しているところから分かります（というか、PyTorch では一般に 2 つの層が全結合する場合にはこのクラスを使用するということです）。

なお、2 つのインスタンス変数には何気なく「fc1」「fc2」という名前を付けていましたが、これらの「fc」は「全結合」（full connected）を意味しています。

第 2 回でも述べましたが、実際の計算がどのように行われるかは、forward メソッドに記述されています。

```
def forward(self, x):
    x = self.fc1(x)
    x = torch.sigmoid(x)
    x = self.fc2(x)
    return x
```

forward メソッドではニューラルネットワーク内部で行う計算が定められる

ここでは、__init__ メソッドで定義したインスタンス変数 fc1 と fc2 を、「self.fc1(x)」「self.fc2(x)」のように、それがあたかもメソッドであるかのように呼び出しています。これは Linear クラスのインスタンスが「呼び出し可能オブジェクト」だからです（詳細については本稿の末尾に掲載するコラムを参照のこと）。ここでは、Linear クラス（に限らず、PyTorch のニューラルネットワークモジュールクラス）のインスタンスは関数やメソッドのように呼び出せることだけを覚えておいてください。

forward メソッドの最初の行だけに注目してみましょう。実は、PyTorch では「self.fc1(x)」のような呼び出しを行うと、多くの場合は回り回って、そのクラスで定義されている forward メソッドが呼び出されるようになっています。Linear クラスの forward メソッドでは、何らかの計算（上で述べたのと同様な計算）をして、その結果を返します。そして、その値が活性化関数を経由して、次の層へと渡されます。このように入力層から出力層の方向へとニューラルネットワークによる計算結果が渡されていくことを「順伝播」（forward propagation）と呼びます。forward メソッドはこうした処理を実行することから、このような名前が付いたのでしょう。

それはともかく、ここでいいたいのは、1 行目と「net.fc1.forward(x)」では（この場合は）同じ処理が行われるということです。そこで、先ほど分割したデータセットの訓練データから先頭の要素だけを取り出して、それを Linear クラスの forward メソッドに渡してみましょう（もちろん、「net.fc1(x)」でも同様です。が、ここではあえて明示的に forward メソッドを呼び出してみます）。

```
x = X_train[0]
print('x:', x)

result = net.fc1.forward(x)
print(result)
```

Linear クラスで定義されている forward メソッドを呼び出してみるコード

実行結果を以下に示します。

```
x = X_train[0]
print('x:', x)

result = net.fc1.forward(x)
print(result)
```

```
x: tensor([4.9000, 3.1000, 1.5000, 0.2000])
tensor([-3.2414,  1.3587, -1.9826,  0.1233,  2.0443], grad_fn=<AddBackward0>)
```

実行結果

net.fc1.forward メソッドに 4 つの数値からなる訓練データを 1 個渡すことで、5 つの要素からなる配列（テンソル）が得られました。これらを活性化関数に通したものが、ノードを 5 つ（とバイアスに対応する 1 つのノードを）持つ隠れ層への入力となります。

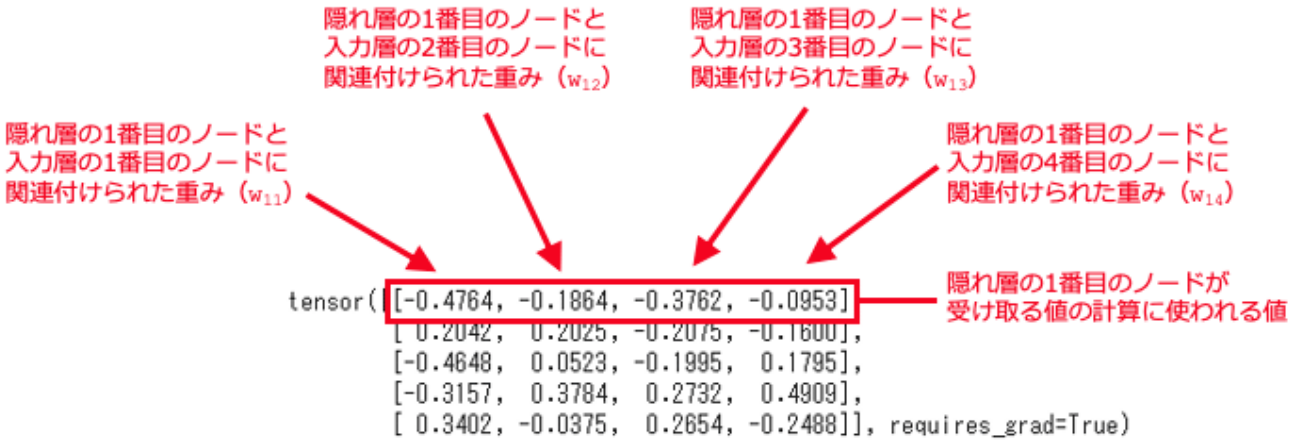
では、これと同じことを自分の手でやってみましょう。その前に、もう一度、以下のコードを実行して、入力層の重みとバイアスを確認しておきます。

```
print('weight')
print(net.fc1.weight)
print('bias')
print(net.fc1.bias)
```

入力層の重みとバイアスを確認

以下の実行結果で注目してほしいのは、重み（weight）が 5 行 4 列の行列（2 次元配列）になっていることと、バイアスが 5 つの要素からなるベクトル（1 次元配列）となっているところです。重みをまとめた行列では、隠れ層の個々のノードが受け取る値の計算に使われる重みが、各行に（4 つの数値を要素とする配列として）まとめられています。

隠れ層の最初のノード（先ほどの図なら y1）に関する計算では、この行列の最初の要素である配列が使われて、その配列の先頭要素が入力層の最初のノード（先ほどの図なら x1）から出力される値を計算するのに使われるといった具合です（つまり、w11 に相当します）。一方、バイアスを含む配列では、隠れ層の各ノードが受け取る値を計算する際に、対応する要素が使われます（最初のノードに関する計算では、バイアスを格納している配列の最初の要素が使われるといった具合です）。



重みの使われ方

そこで、ここでは weight 属性の値を変数 w に代入しておきましょう。バイアスの値は変数 b に代入しておきます。入力層に与えられた値は、既に見た通り変数 x に代入されています。

```
w = net.fc1.weight
b = net.fc1.bias
# x = X_train[0]
```

入力層の重みを変数 w に、入力層のバイアスを変数 b に代入しておく

すると、隠れ層の最初のノード（y1）が受け取る値は次のように計算できます。

w[0][0] * x[0] + w[0][1] * x[1] + w[0][2] * x[2] + w[0][3] * x[3] + b[0]

この式と先ほど示した、隠れ層の最初のノードが受け取る値の式を比べてみてください（x_0 は通常「1」であり、b_1 と書いても構わないことに注意）。

$w_{11} \times x_1 + w_{12} \times x_2 + w_{13} \times x_3 + w_{14} \times x_4 + b_1 \times x_0$

これまでに示してきた図では重みや入力変数などを 1 始まりの添字で表現していました。しかし、Python ではインデックスは 0 で始まります。そのため、インデックスと添字がズレてしまっていますが、それを除けば、同じ形式になっていることが分かります。他のノードへの出力を計算するときも同様です。では、実際に先ほど forward メソッドを呼び出したときに得られた結果と上記の式を使った計算結果が等しくなるかを見てみましょう。

```
o0 = w[0][0] * x[0] + w[0][1] * x[1] + w[0][2] * x[2] + w[0][3] * x[3] + b[0]
o1 = w[1][0] * x[0] + w[1][1] * x[1] + w[1][2] * x[2] + w[1][3] * x[3] + b[1]
o2 = w[2][0] * x[0] + w[2][1] * x[1] + w[2][2] * x[2] + w[2][3] * x[3] + b[2]
o3 = w[3][0] * x[0] + w[3][1] * x[1] + w[3][2] * x[2] + w[3][3] * x[3] + b[3]
o4 = w[4][0] * x[0] + w[4][1] * x[1] + w[4][2] * x[2] + w[4][3] * x[3] + b[4]
print(o0.data, o1.data, o2.data, o3.data, o4.data)
```

「net.fc1.forward」メソッドの呼び出しと同じことを手作業で行うコード

以下に実行結果を示します。

```
o0 = w[0][0] * x[0] + w[0][1] * x[1] + w[0][2] * x[2] + w[0][3] * x[3] + b[0]
o1 = w[1][0] * x[0] + w[1][1] * x[1] + w[1][2] * x[2] + w[1][3] * x[3] + b[1]
o2 = w[2][0] * x[0] + w[2][1] * x[1] + w[2][2] * x[2] + w[2][3] * x[3] + b[2]
o3 = w[3][0] * x[0] + w[3][1] * x[1] + w[3][2] * x[2] + w[3][3] * x[3] + b[3]
o4 = w[4][0] * x[0] + w[4][1] * x[1] + w[4][2] * x[2] + w[4][3] * x[3] + b[4]
print(o0.data, o1.data, o2.data, o3.data, o4.data)
```

```
tensor(-3.2414) tensor(1.3587) tensor(-1.9826) tensor(0.1233) tensor(2.0443)
```

実行結果

重みとバイアスの更新

重みやバイアスは学習の過程で更新されていきます。そこで、今回は学習を一度だけ行って、重みが更新されている様子を確認するだけとしましょう。実際のコードを以下に示します。

```
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.03)

print('before learning')
print('weight')
print(net.fc1.weight)

print('learn once')
outputs = net(X_train)
loss = criterion(outputs, y_train)
loss.backward()
optimizer.step()

print('after learning')
print('weight')
print(net.fc1.weight)
```

学習を一度だけ行い、入力層の重みとバイアスがどう変化したかを確認するコード

学習を行うコード自体は前回までに見てきたものをさらにシンプルにしただけのものです。Net クラスのインスタンスに訓練データを渡し、その出力から損失関数で損失を計算して、それを基に重みやバイアスを更新していると思ってください。

実行結果を以下に示します。

```
criterion = nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.03)

print('before learning')
print('weight')
print(net.fc1.weight)

print('learn once')
outputs = net(X_train)
loss = criterion(outputs, y_train)
loss.backward()
optimizer.step()

print('after learning')
print('weight')
print(net.fc1.weight)
```

```
before learning
weight
Parameter containing:
tensor([[-0.4764, -0.1864, -0.3762, -0.0953],
        [ 0.2042,  0.2025, -0.2075, -0.1600],
        [-0.4648,  0.0523, -0.1995,  0.1795],
        [-0.3157,  0.3784,  0.2732,  0.4909],
        [ 0.3402, -0.0375,  0.2654, -0.2488]], requires_grad=True)
learn once
after learning
weight
Parameter containing:
tensor([[-0.4762, -0.1862, -0.3764, -0.0954],
        [ 0.2124,  0.2060, -0.2002, -0.1573],
        [-0.4649,  0.0523, -0.1996,  0.1794],
        [-0.3093,  0.3807,  0.2799,  0.4934],
        [ 0.3409, -0.0373,  0.2664, -0.2484]], requires_grad=True)
```

実行結果

ほんの少しですが、学習前と学習後で重みとバイアスが更新されているのが分かるはずです。実際の学習では、こうした処理を何度も何度も繰り返して、訓練データを基にした推測結果が正解ラベルに近い値となるまで、重みとバイアスを更新していくことになります。これについては次回に詳しく見ていくことにします。

呼び出し可能オブジェクト

本稿では「呼び出し可能オブジェクト」という言葉が出てきたので、これについて最後に少し説明しておきましょう。よく知らないし、知りたくもないという方は、読み飛ばしても構いません。ただし、PyTorchでは「ニューラルネットワーククラスのインスタンスは、関数やメソッドのように呼び出せる」ことだけは覚えておきましょう。

Pythonには「呼び出し可能オブジェクト」というオブジェクトがあります。関数やメソッドはもちろん呼び出し可能オブジェクトの代表的な存在です。

ですが、上で見たように Linear クラスのインスタンス（あるいは、上で定義している Net クラスのインスタンス）もまた呼び出し可能オブジェクトです。Python では、あるオブジェクトが __call__ 特殊メソッドを持っている場合、それは関数のように呼び出すことができるようになっていて、Linear クラスのインスタンスはまさにこのメソッドを持っているのです。

実際にあることを確認してみましょう。

```
linear = nn.Linear(4, 5)
print(linear.__call__)
```

Linear クラスのインスタンスに __call__ メソッドがあるか

これを実行すると次のようになります。

```
linear = nn.Linear(4, 5)
print(linear.__call__)
```

```
<bound method Module.__call__ of Linear(in_features=4, out_features=5, bias=True)>
```

実行結果

この結果を見ると、Linear クラスのインスタンスの __call_ メソッドは、Module.__call__ メソッドに束縛されていることが分かります。詳細なことはともかくとして、これで Linear クラスのインスタンスには __call__ メソッドがあり、関数のように呼び出せることが分かりました。

__call__ メソッドを持つオブジェクトは、インスタンス名にかっこ「()」を付加して、そこに 0 個以上の引数を渡すことで呼び出せます。呼び出し可能オブジェクトを使って呼び出しを行うと、Python の処理系により、対応する __call__ メソッドが呼び出されて、そのパラメーターに呼び出し可能オブジェクトに渡した引数が渡されるようになっています（そのため、__call__ メソッドのパラメーターリストと、オブジェクトに渡す引数リストが一致している必要があります）。

先ほど「Linear クラスのインスタンスの __call_ メソッドは、Module.__call__ メソッドに束縛されている」と述べたことから分かる通り、PyTorch ではニューラルネットワークの基底クラスである Module クラスで __call_ メソッドが定義されています。そのため、ニューラルネットワーククラスのインスタンスは全て関数のように呼び出せます。

ちなみに、Module.__call__ メソッドの内部ではいろいろと処理を行っていますが、上述したように、ニューラルネットワーククラスの forward メソッドは __call__ メソッドから内部的に呼び出されるような構造になっています。そのため、上で定義した Net クラスのインスタンスを net とすると、「net(入力データ)」のようにするだけで、その forward メソッドが呼び出されます。この構造を利用して、Net クラスでは、forward メソッドでこのニューラルネットワークで行う計算処理を記述するようになっているのです。

ニューラルネットワークの学習でしていること

シンプルな関数をニューラルネットワークに見立てて、その係数を学習させながら、その過程でどんなことが行われているかを見ていきます。

（2020 年 04 月 27 日）

前回は、ニューラルネットワークが何らかの値を推測するのに必要な重みやバイアスがどこにあるのか、それらを使ってどんなふうに計算が行われるのかを見ました。今回は、重みを例に、学習の過程でこれがどのようにして新しい値に更新されていくかを見てみます。

学習とは？

第 2 回では、次のようなコードで学習を行っていました。

```
net = Net()  # ニューラルネットワークのインスタンスを生成

criterion = nn.MSELoss()  # 損失関数
optimizer = torch.optim.SGD(net.parameters(), lr=0.003)  # 最適化アルゴリズム

EPOCHS = 2000  # 2000 回繰り返す
for epoch in range(EPOCHS):
    optimizer.zero_grad()  # 手順 0:重みとバイアスの更新で内部的に使用するデータをリセット
    outputs = net(X_train)  # 手順 1：ニューラルネットワークにデータを入力
    loss = criterion(outputs, y_train)  # 手順 2：正解ラベルとの比較
    loss.backward()  # 手順 3：誤差逆伝播
    optimizer.step()  # 手順 4：重みとバイアスの更新

    if epoch % 100 == 99:  # 100 回繰り返すたびに損失を表示
        print(f'epoch: {epoch+1:4}, loss: {loss.data}')

print('training finished')
```

学習を行うコードの例

まず自分が記述したニューラルネットワークを表すクラスのインスタンスを生成して、損失関数と呼ばれる関数、最適化アルゴリズムという重みとバイアスを更新するのに使用するアルゴリズムを選択しています。

ここでは、損失関数として PyTorch が提供する MSELoss クラス（torch.nn.MSELoss クラス）のインスタンスを代入しています。これは、ある値と別の値の距離（誤差）の 2 乗を損失として表すものです。そして、最適化アルゴリズムには、同じく PyTorch が提供する SGD クラス（torch.optim.GSD クラス）のインスタンスを代入しています。SGD とは「stochastic gradient descent」（確率的勾配降下法）の略で、PyTorch では損失関数や誤差逆伝播法などと組み合わせて、重みを最適な値に更新していくために使用します（本稿ではこれについては深くは触れません）。

その後は 2000 回のループで実際の学習に入ります。ループ回数を指定している変数 EPOCHS の「EPOCH（S）」とは「時代」のような意味の語ですが、ここでは学習を行うひとまとまりの単位（この場合は、訓練データを使った一度の学習）のようなものだと考えるとよいでしょう。

ループの内部では次のようなことをしています。

- 手順 0：重みとバイアスの更新で内部的に使用するデータのリセット
- 手順 1：ニューラルネットワークへの訓練データの入力
- 手順 2：ニューラルネットワークが計算した値と正解ラベルを比較して、損失を計算
- 手順 3：誤差逆伝播法により、重みとバイアスの更新に内部で使用するデータを計算
- 手順 4：選択した最適化アルゴリズムを使用して、重みとバイアスを更新

このうちの手順 1、つまりニューラルネットワークへの訓練データの入力については、これまでにも何度も見ています。今回はその後（とそれに付随する手順 0）について、そこでどんなことが行われているのかを、手を動かしながら、簡単に見ていくことにします。

今回学習させるもの

今回はニューラルネットワーククラスを定義したりはしません。シンプルに重みは 1 つだけで、バイアスはなし、入力も 1 つだけとします。つまり、これは「y = w × x」という式で表せます。この重みがどのようにして更新されていくかをみることで、学習時にはどんなことが行われているかが想像できるようになるはずです。

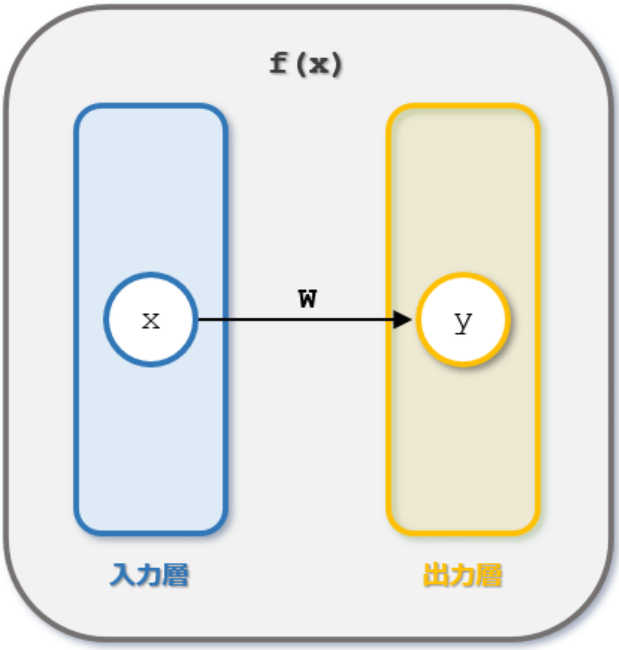
実際には、重み（w）の初期値は 1.95、それに対応する正解ラベルの値は 2.0 であるとします。つまり、「f(x) = 1.95 × x」という式が「f(x) = 2.0 × x」へと更新されていく過程を（最初の部分だけ）見ていきます。ここでは、x の値を 1 とします。つまり、f(x) の出力は 1.95 となりますが、実際には 2.0 になってほしいということです。

今述べたことをコードで表現すると次のようになります（変数 w に代入している重みを表すテンソルのインスタンス生成で指定している「requires_grad=True」については後述します。今はそういうものだと思っておいてください）。なお、今回のコードはこのリンク先で公開しています。

```
import torch

w = torch.tensor([[1.95]], requires_grad=True)  # 重みの初期値は 1.95 とする
t = torch.tensor([[2.0]])  # 重みの正解は 2.0
x = torch.tensor([1.0])  # 関数への入力は 1.0 とする

def f(x):  # 関数 f(x) = w * x の定義
    return w * x

print('w:', w)
print('t:', t)
```

重みと正解ラベル、重みを使って計算を行う関数

関数 f(x) をニューラルネットワークと見なせば、これは入力層と出力層のみで構成され、上の計算を行うニューラルネットワークのようにも考えられることに気が付いたでしょうか（ただし、今回はバイアス b を省略したと考えてください）。それっぽくするために、上に示した変数 w は「1 行 1 列の行列」（1 行 1 列の 2 次元配列）として定義しています。変数 x も「1 要素のベクトル」（1 要素の 1 次元配列）となっています（正解ラベルに相当する変数 t は、以下で損失を計算する際に都合がよいように変数 w と同じ形式としています）。



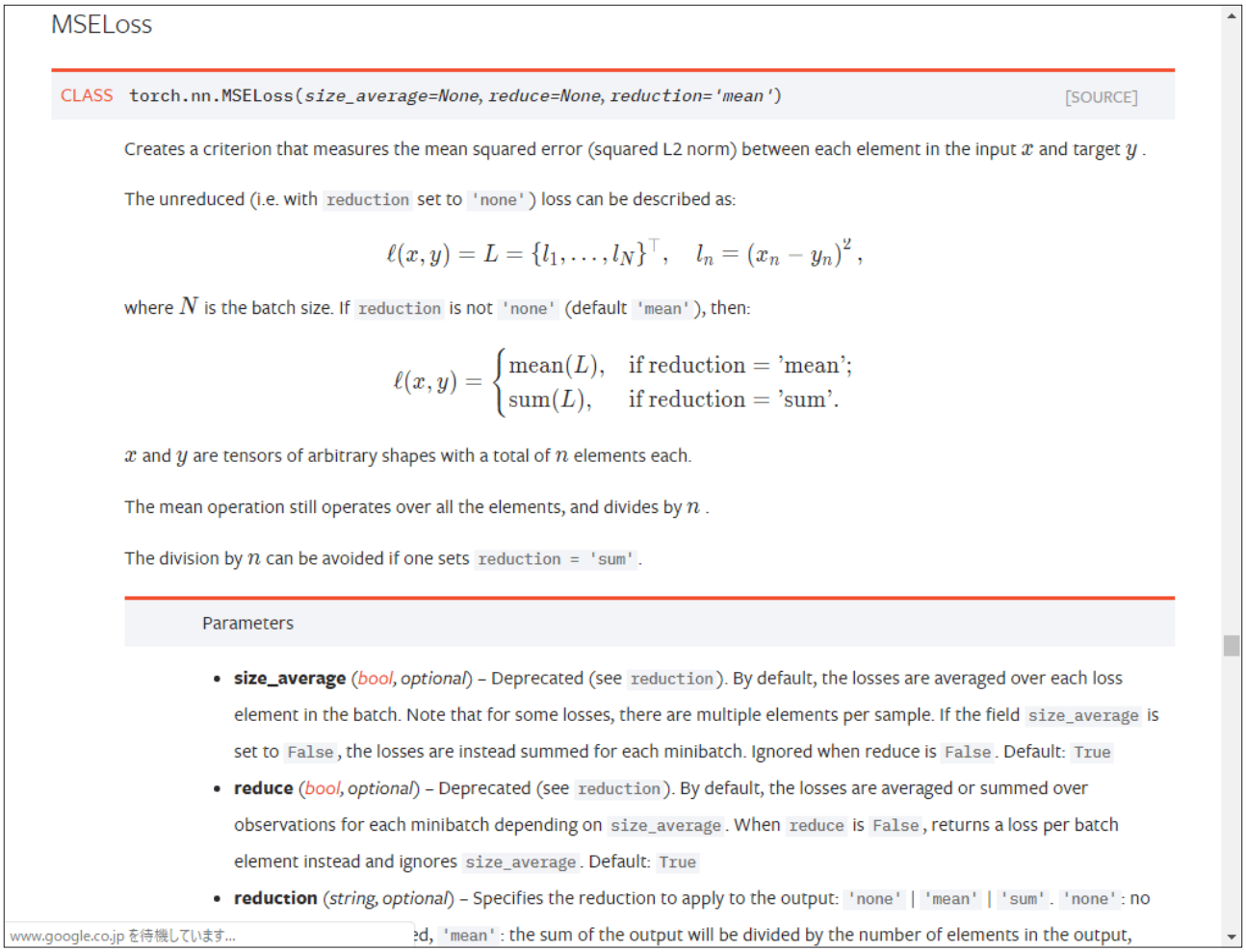
入力層のノード数が 1、出力層のノード数が 1、隠れ層がないニューラルネットワーク（のようなもの）

この後は、重みがどのように更新されていくかを観察していくことにしましょう。

損失の計算

これまでの連載の中でも述べていますが、ここで見ている（教師あり学習における）ニューラルネットワークの学習とは「損失を最小とするように、重みとバイアスを更新していく」過程のことです。損失とは、ニューラルネットワークが算出した出力値（推測値）と正解ラベルの値との誤差（距離）のことです。これが最小になるのは、出力値と正解ラベルが等しくなったときです。まずはこのことを頭に入れておきましょう。以下では損失の求め方について見ていきます（上記の手順 2 に相当）。

損失（誤差、距離）を計算するには幾つかの方法があります。ここでは、PyTorch が提供する MSELoss クラス（torch.nn.MSELoss クラス）を使用することにします。



MSELoss クラス

MSE とは「Mean Squared Error」の略で、日本語にすると「平均二乗誤差」です。MSE では、同じ要素数のベクトル（一次元配列）が 2 つあったときに、同じインデックス位置にある 2 つの要素で減算を行った結果を二乗したものを、それらの要素の誤差（二乗誤差）として考え、ベクトルの全要素についてそれらを計算し、それらの総和を要素数で割った値（平均値）を求めます。

今回の例では、比べるのは 1 つの出力値と 1 つの正解ラベルだけなので、平均の取りようがないのですが、これを使って話を進めることにしましょう。

ここで比較する対象（誤差を計算する 2 つの要素）は、関数 f(x) の値と、正解ラベルです。変数 x の値をここでは 1 とすることは既に述べました。よって、重みの初期値とした 1.95 と、変数 x の値である 1 を乗じた結果である 1.95 と、正解ラベルの値である 2 の差である -0.05（または 0.05）を二乗した 0.0025 が今回求める誤差となります。

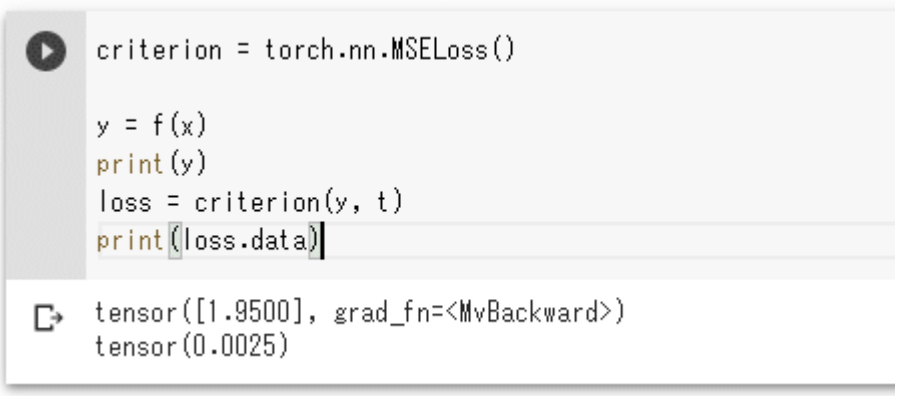
実際にそうなるかを調べてみましょう。

```
criterion = torch.nn.MSELoss()

y = f(x)
loss = criterion(y, t)
print(loss.data)
```

損失を計算する

ここでは、MSELoss クラスのインスタンスを生成して、それを変数 criterion に代入することで、損失関数を作成しています（「criterion」とは「何かをするときの基準や尺度」といった意味です。この場合、この後に行う重みの更新で実際にどんな処理をするかの判断基準になるといった意味でしょう）。後は、関数 f(x) を呼び出して、その値と正解ラベルの値を criterion に渡すことで損失を計算しています。

実行結果を以下に示します。



実行結果

このような計算を実際に行う関数を自分で定義するとしたら、次のようになるでしょう（mycriterion 関数はベクトルを受け取ることを前提としています。「1」のような配列以外の値を与えると例外が発生します）。

```
def mycriterion(x, y):
    result = (x - y) ** 2
    return result.sum() / len(result)
```

自作の損失関数

実際に、この関数を使ってみると、上のコードは次のようにも書けます。

```
myloss = mycriterion(y, t)
print(myloss)
```

自作の損失関数を使った損失の計算

「損失」（または誤差や距離）をどのように定義するかによって、損失関数はさまざまな種類に分類されます。ここでは平均二乗誤差を例としましたが、いずれにせよ、損失関数ではニューラルネットワークから得た値と、正解ラベルとの誤差を何らかの方法で計算することと覚えておきましょう。

次は、この損失を基に重みを更新していく作業を見ていきます。

重みの更新の過程

「重みを更新」するといっても、どうすればよいでしょう。ここではゴールは見えています。1.95 という重みを 2.0 に近づけていくことです。

ここで、先ほどの損失関数について少し考えてみましょう。MSELoss クラスを使って作成した損失関数は、出力値を含むベクトルと正解ラベルを含むベクトルを受け取り、対応する 2 つの値を減算して二乗した値の平均を損失としていました。ただし、ここでは 1 つの計算値と対応する正解ラベルの誤差を求めているだけであることには注意してください。よって、出力値を output、正解ラベルを label としたときに損失関数で行っている処理は「(output - label) ** 2」と同じと考えられます（「平均」の概念が抜け落ちていますが、これは話をシンプルにするためです）。

この値が最小（理想的には 0）となるような重みを見つけることがここで行っている学習の目的です。一方の比較対象（正解ラベル）はシンプルに 2 です。そこで、「(output - label) ** 2」（label = 2）という式（関数）を最小化する output の値（さらには重み w の値）が何かを考えます（もちろん、output = 2.0 ですし、w = 2.0 です）。

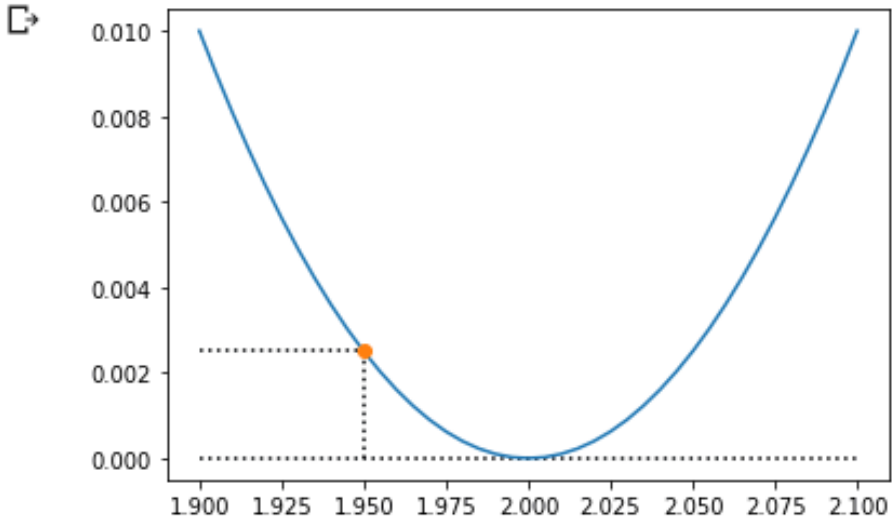
以下のコードでは、今述べた形で損失を返す関数「calc_loss」を定義して、変数 output には「重みが 1.9 ～ 2.1 の範囲で変化したときの、f(x) = w × x（x = 1）の計算結果（をエミュレートしたもの）」を、定数 LABEL には正解ラベルの値である 2 を代入しています。そして、calc_loss 関数が計算した損失（myloss）をグラフにプロットしています。

```
import matplotlib.pyplot as plt

def calc_loss(output, label):
    return (output - label) ** 2

# 1.9 ～ 2.1 の範囲の重み w に対し、関数 f(x) に x = 1 を与えたときの計算結果
output = torch.arange(1.9, 2.1, 0.005)

LABEL = 2

myloss = calc_loss(output, LABEL)
plt.plot(output, myloss)
plt.plot(1.95, calc_loss(1.95, LABEL), marker='o')
plt.hlines(0, 1.9, 2.1, linestyle=':')
plt.hlines(0.0025, 1.9, 1.95, linestyle=':')
plt.vlines(1.95, 0, 0.0025, linestyle=':')
plt.show()
```

損失を計算する関数

現在のところ、重みは 1.95、x の値は 1 であることから、関数 f(x) の計算値である 1.95 とそのときに得られる誤差も図にプロットするようにしました（横軸の値は関数 f(x) に対して x = 1 としたときの計算結果に対応します）。これを実行した結果が以下です（グラフ描画のコードについての説明は省略します）。

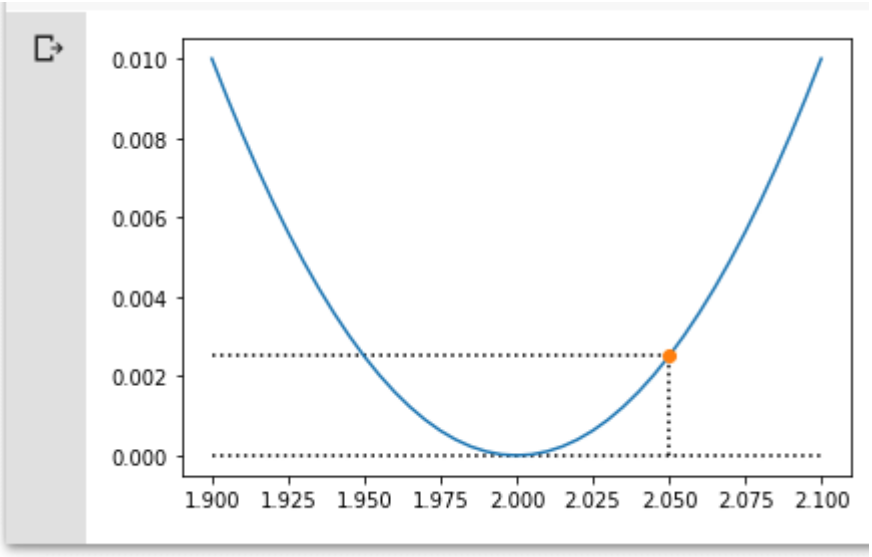
```
import matplotlib.pyplot as plt

def calc_loss(output, label):
    return (output - label) ** 2

# 重みwが変化して、関数f(x)の計算結果が変わったことをエミュレート
output = torch.arange(1.9, 2.1, 0.005)

LABEL = 2

myloss = calc_loss(output, LABEL)
plt.plot(output, myloss)
plt.plot(1.95, calc_loss(1.95, LABEL), marker='o')
plt.hlines(0, 1.9, 2.1, linestyle=':')
plt.hlines(0.0025, 1.9, 1.95, linestyle=':')
plt.vlines(1.95, 0, 0.0025, linestyle=':')
plt.show()
```



実行結果

これを見れば分かる通り、損失が最小になるのは、関数 f(x) に x = 1 を与えたときの計算結果が 2.0 となったときで、現在の重みの値「1.95」をこれに近づけるように更新していけばよいということになります。

人がグラフを見れば、「現在の横軸（output = f(x) の値）が 2.0 に近づくように、w を増加させていけばよいだろう」ということは一目瞭然です。しかし、コンピューターにはなかなかそうはいきません。そこで重要なのが、重みを増加させるのか、減少させるのかの判断です。例えば、重み w が 2.05 まで増加したらどうなるでしょうか。



重みを 2.05 まで増やすと、その計算結果（2.05 × 1）は 2.05 となる

もちろん、今度は関数 f(x) の計算結果は 2.05 となり（x = 1）、損失を最小とする値を超えてしまいました。となると、重みを減らす方向に戻る必要があります。

ここで重みを増加させるのか、減少させるのかの判断に役立つ指標があります。それは「勾配」と呼ばれるものです。あるいは、「グラフの（接線の）傾き」といってもよいでしょう。今の例では、横軸の値となる output = 関数 f(x) の値が 2.0 になるまではグラフは右肩下がりでその勾配はマイナスです。一方、2.0 を越えればグラフは右肩上がりでその勾配はプラスになります。損失を最小にする箇所（関数 f(x) の計算結果が 2.0）では勾配はゼロとなります。

つまり、勾配がマイナスなら重みを増加させて、プラスなら減少させていけばよいということです。そして、勾配がゼロとなる地点が見つかれば、そこで最適な重みが見つかったことになります（実際には、これほど簡単な話ではありません。グラフの頂点が複数あるような場合には、実は最適な値ではないところで勾配がゼロとなってしまう可能性もあります。こうした問題を解決する方法も考えられています。が、ここでは損失を求めるグラフが単純な 2 次曲線であるため、傾きがゼロとなる重みを求めるだけでよしとします）。

なお、勾配を表す英単語は「gradient」であることから、それを省略した「grad」などが勾配に関連する属性や変数、関数などの名前ではよく使われることも覚えておきましょう（以下のコードでは、勾配を求める関数に「calc_grad」と名前を付けてあります）。

その勾配（傾き）を求めるには幾つかの方法があります。一つは損失関数の微分です。数学の難しい話を抜きにして、ざっくりとした話をするので、難しいという方は流して読んでもらってもかまいません。

上のグラフを描画するコードでは損失は「(output - label) ** 2」として求めていました。この式において、output = f(x) であることを思い出してください。さらに、f(x) は重み w と入力値 x の乗算でもありました。

つまり、この関数では「(w * x -label) ** 2」と同じ計算をしているということです。これを展開したものを、やはり Python のコードで表現すると「w ** 2 * x ** 2 - 2 * w * x * label + label ** 2」となります。ここでは w が変化することで、損失関数の勾配がどう変化するかに着目しているので、上記の式を「w」で微分（偏微分）します。その結果は「2 * x ** 2 * w - 2 * x * label」です。そして、ここでは w = 1.95、x = 1、label = 2 でしたから、これらの値をこの式に代入することで、勾配が得られます（式からは output が消えて、w、x、label に関するものになっていることに注意してください）。

なお、数学的な深い話は「AI・機械学習の数学入門 — 中学・高校数学のキホンから学べる」をご覧ください。

これを実際に行うコードを以下に示します。

```
def calc_grad(w, x, label):
    return 2 * x ** 2 * w - 2 * x * label

grad = calc_grad(w, x, LABEL)
print('grad:', grad)
```

重みが 1.95 の関数 f(x) に 1 を与えた場合の損失関数の傾き

実行結果を以下に示します。

```
def calc_grad(w, x, label):
    return 2 * x ** 2 * w - 2 * x * label

grad = calc_grad(w, x, LABEL)
print('grad:', grad)
```

```
grad: tensor([[-0.1000]], grad_fn=<SubBackward0>)
```

実行結果

これにより、重みを 1.95、x = 1、label = 2 としたときの損失関数の傾きは「-0.1000」になったことが分かりました。

関数の最小値（あるいは最大値）を求めることを「最適化問題」と呼ぶことがよくあります。そして、今見たような勾配を手がかりとして、その値を探すことを一般に「勾配法」と呼びます。機械学習やニューラルネットワークの世界では、この手法を用いて、損失関数の最小値を求める値を探すことがよくありますが、これを「降下勾配法」と呼びます。降下勾配法のアルゴリズムにも幾つかの種類があり、冒頭で紹介したコードに含まれていた「torch.optim.SGD クラス」はそうした最適化アルゴリズムの 1 つを表すクラスです（この後も使用しますが、この例ではオーバースペックなものでもあります）。

ここでは、損失関数の微分によって、重みを更新するための手がかりである勾配を求めましたが、現在ではより効率的に勾配を求める方法として、「誤差逆伝播法」（backpropagation、バックプロパゲーション）が使われています。この詳細については後続の回に譲りますが、これを実際に行っているのが、以下のコード（本稿冒頭で示したコード）で、強調書体とした行です。

```
outputs = net(X_train)  # 手順 1：ニューラルネットワークにデータを入力
loss = criterion(outputs, y_train)  # 手順 2：正解ラベルとの比較
loss.backward() # 手順 3：誤差逆伝播
optimizer.step()  # 手順 4：重みとバイアスの更新
```

誤差逆伝播を行っているコードの例

実は PyTorch では、テンソルに対して、どのような操作が行われたかを記録できるようになっています。このことを利用して、あるノードから別のノードへ渡される値を計算するときに、「ここでは何と何を乗算した」「ここではアレとコレを加算した」など、何が行われたかの情報がそのテンソルに記録されます。そして、勾配を計算する際には、計算した結果からその情報を遡りながら、各ノードにおける勾配を効率よく計算できるようになっているのです。この情報を保存するためには、テンソルを作成する際に、「requires_grad」キーワード引数に True を指定する必要があります。

そのため、先ほどの重みを格納する 1 行 1 列の行列の定義では、次のようにして、テンソルを生成していたのです。

```
w = torch.tensor([[1.95]], requires_grad=True)
```

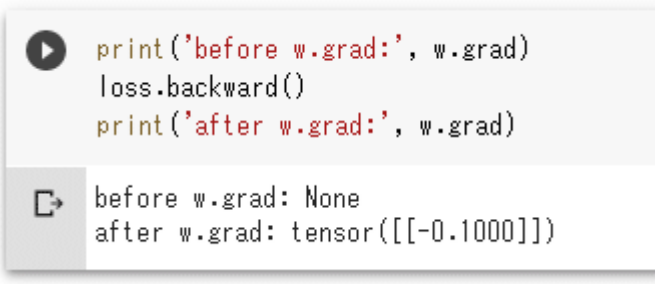
テンソルに対する操作を記録するには「requires_grad=True」を指定する

というわけで、ここで backward メソッドを呼び出して、実際に誤差逆伝播を実行してみましょう。ここでは、criterion 呼び出しによって返された値（変数 loss）に対して、これを行ってみます。なお、勾配はテンソルの grad 属性に保存されるので、backward メソッドの呼び出しの前後で、これが本当に変化するかも確認します。

```
print('before w.grad:', w.grad)
loss.backward()
print('after w.grad:', w.grad)
```

誤差逆伝播を行い、勾配が計算されたかを確認

実行結果を以下に示します。



実行結果

どうでしょう。呼び出し前の w.grad 属性の値は「None」でしたが、呼び出し後は「-0.1000」になっています。先ほど手作業で実行した勾配も「-0.1000」だったので、どちらの方法でも同じ勾配が得られたと考えてよいでしょう。ここまでの話が冒頭に示したコードにおける手順 3 に相当します。

最後に、得られた勾配を利用して、重みを更新します（手順 4 に相当）。ここで登場するのが先ほども少し出てきた最適化アルゴリズムです。ここではこれまでと同様に、torch.optim.SGD クラスのインスタンスを生成することにします。

```
optimizer = torch.optim.SGD([w], lr=0.3)
```

最適化アルゴリズムには SGD を選択

SGD クラスのインスタンスを生成する際には、更新を行う対象と学習率（learning rate）と呼ばれる値を指定する必要があります。ここでは更新の対象には 1 行 1 列の行列である w を含んだリストを、学習率を表すキーワード引数 lr には 0.3 を指定しています（学習率については後述します）。

ちなみに、これまでに見てきたあやめの品種の分類では次のようにしてインスタンスを生成していました。

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.003)  # 最適化アルゴリズム
```

あやめの品種を分類するニューラルネットワークでの最適化アルゴリズムの選択

あやめの品種を分類するために定義した Net クラスでは（コードは省略）、そのインスタンス変数 fc1 と fc2 に入力層から隠し層、出力層へと渡る値を計算するために必要な重みやバイアスが保存されているのは前回も見た通りです。そこで、Net クラスのインスタンスに対して parameters メソッドを呼び出して、それらを反復するジェネレータを手に入れて、それをインスタンスの生成に使用していると考えてください。

では、実際に重みを更新しましょう。これには step メソッドを呼び出すだけです（手作業で同じことを行うのは省略します）。ただし、今度は重み w が本当に更新されるかを確認するために、変数 w の値を step メソッド呼び出しの前後で調べます。

```
print(w)
optimizer.step()
print(w)
```

重みの更新

実行結果を以下に示します。

実行結果

重みが 1.9500 から 1.9800 に増加したことが確認できました。ところで、増加した「0.0300」はどこから出てきた値なのでしょうか。実はこれは「勾配」と「学習率」を乗じたものになっています。先ほど計算した勾配（w.grad）は「-0.1000」で、学習率には「lr=0.3」を指定していました。これらを乗じた結果である「-0.0300」を重みから減算することで、実際には 0.0300 を加算しています。

先ほどのグラフからは、この損失関数では、勾配は最小値に近づくほどに緩くなり、最小値から遠ざかるほどにきつくなることが分かります。そして、勾配と学習率を乗じた値を加算／減算することから、勾配がゼロになる地点よりも離れたところでは、重みの変化は大きくなり、勾配がゼロに近づくに従って重みの変化が少なくなることにも注意してください。

これに対して、学習率はいわば学習（パラメーターの更新）をどのくらいのスピードで進めるかを決定するファクタです。小さすぎる値を指定すると、勾配が大きな場合でも、いつまで経っても学習が進まず、逆に大きすぎる値を指定すると、勾配が小さな時でも、重みの変化量が大きくなりすぎて、適切な値が見つからないといった状況を招くこともあります。そのため、適当な値を指定する必要がありますが、これにはその場その場での試行錯誤が必要になるかもしれません。

最後に、同じ工程をもう一度だけ実行してみます。

```
print(w.grad)
optimizer.zero_grad()
print(w.grad)

y = f(x)
print(y)
loss = criterion(y, t)
loss.backward()
print('updated w.grad:', w.grad)
optimizer.step()
print('updated w:', w)
```

もう一度、重みを更新する

最初に行っているのは、勾配（w.grad 属性）をリセットする処理です。grad 属性に記録された勾配は、リセットをしない限り、累積されていくようになっていますが、これをゼロにして、学習ごとの勾配だけを使うようにしています。

その後は、これまでに見たコードそのままです。途中と最後には勾配の値と更新後の重みを表示するコードも含めてあります。

実行結果は次の通りです。

```
print(w.grad)
optimizer.zero_grad()
print(w.grad)

y = f(x)
print(y)
loss = criterion(y, t)
loss.backward()
print('updated w.grad:', w.grad)
optimizer.step()
print('updated w:', w)
```

```
tensor([[-0.1000]])
tensor([[0.]])
tensor([1.9800], grad_fn=<MvBackward>)
updated w.grad: tensor([[-0.0400]])
updated w: tensor([[1.9920]], requires_grad=True)
```

実行結果

重みの値が理想値である 2.0 にまた少し近づいたことが分かります。また、勾配の値は 0.04 です。これと学習率の 0.3 を乗じると「0.012」が得られます。これと前回の重み「1.98」を加算した結果「1.992」が新しい重みの値となっている点にも注意してください。

後はこの作業を繰り返し実行していくことで、重みが 2.0 に近づいていくはずです。これについてはコードを示しませんが、冒頭に示したようにループ処理を書くことで必要なだけ学習を行えるはずです。

今回は学習の過程で、重みがどのようにして更新されていくかを見ました。ここまでであやめの品種を分類するニューラルネットワークのコードがどうしてあのようになっていたかはおおよそ説明し終わりました。最後に残ったのは、全結合を行うクラスです。次回は、これを行うクラスを簡易的に実装してみる予定です。

自分だけの Linear クラスを作ってみよう

PyTorch が提供する Linear クラスの簡易版を作りながら、全結合型のニューラルネットワークで何が行われるのかを見ていきます。

（2020 年 05 月 12 日）

前回は、学習によってどのように重みが更新されていくかといった話をしました。今回は、実際にニューラルネットワークの各層にあるノードを全結合し、ある層から別の層へと渡される値を計算するクラスを作ってみます。

なお、今回のコードはこのリンク先で公開しているので、必要に応じて参照してください。

自分だけの Linear クラスを作ってみる

PyTorch を使えば、torch.nn.Linear クラスを使うことで、全結合を行う 3 層構造のニューラルネットワークを例えば次のように書けることは、これまでに何度も見てきました。

```
from torch import nn

INPUT_FEATURES = 4
HIDDEN = 5
OUTPUT_FEATURES = 1

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(INPUT_FEATURES, HIDDEN)
        self.fc2 = nn.Linear(HIDDEN, OUTPUT_FEATURES)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        return x
```

全結合を行い、あやめの品種を推測するクラスの例

ここでは、Linear クラスの代わりに使える MyLinear クラスを作ることで、「self.fc1 = nn.Linear(INPUT_FEATURES, HIDDEN)」のようにしていたところを「self.fc1 = MyLinear(INPUT_FEATURES, HIDDEN)」のように書けるようにすることが目標です。

といっても難しいことはないので、安心してください。必要なのは、重みとバイアスを格納するインスタンス変数を用意することと、forward メソッドで入力値と重み、バイアスを使って次のノードに送る値を計算するようにすることの 2 つだけです。PyTorch が提供する torch.nn.Module クラスと、torch.nn.functional.linear 関数を使うことで、とても簡単にこの処理を行うクラスを実装できます。

それでは、実際にコードを書いていくことにしましょう。

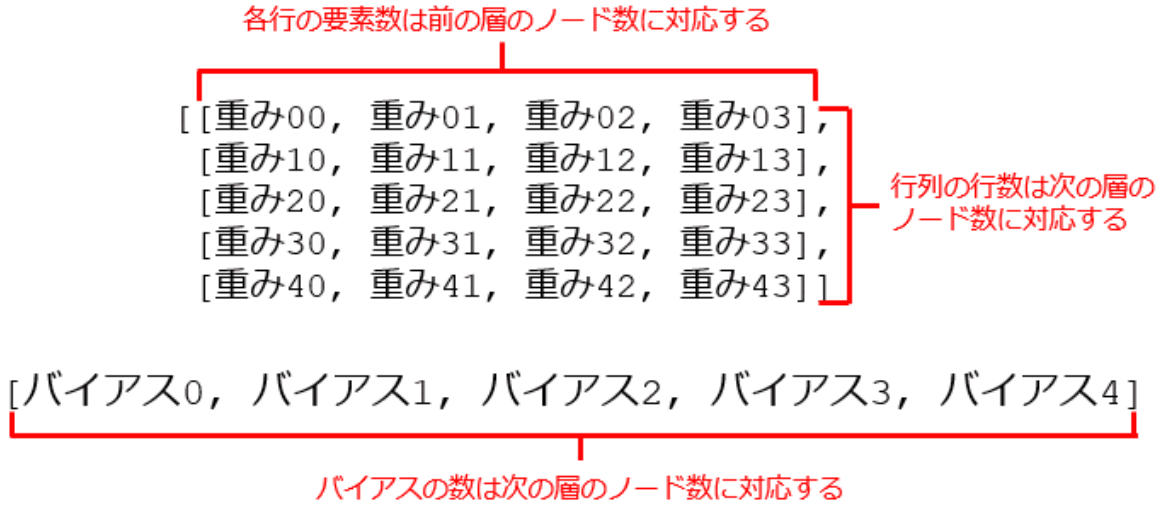
Module クラスの派生クラスを定義して重みとバイアスを持たせる

上でも述べた通り、ここでは PyTorch が提供する Module クラス（torch.nn.Module クラス）を利用することで、ニューラルネットワークモジュールが持つ最低限の機能を継承するクラスを定義します。つまり、MyLinear クラスはおおよそ次のような形を取ることになります。

```
from torch import nn

class MyLinear(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()  # 基底クラスの初期化
        # 重みとバイアスを格納するインスタンス変数を定義
    def forward(self, x):
        # ある層からの入力 x を受け取り、次の層への出力を計算する
        pass
```

MyLinear クラスの基本形

詳しい話は第 4 回でしていますが、重みは行列として、バイアスはベクトル（一次元配列）として管理することにします（重みとバイアスを 1 つの行列の要素として扱う方法についても最後に簡単に触れます）。



重みとバイアスを格納するインスタンス変数

上の図から分かりますが、ここで重要になるのは、重みを格納する行列の行数が出力先のノードの数（出力値の数）と同じになり、行列の列数が入力側のノードの数（入力値の数）と同じになることです。上のコードでは、in_features が入力側のノード数を、out_features が出力側のノード数を表しているので、重みを格納する行列は「out_features 行、in_features 列」の行列となります。バイアスは、出力側のノードの数と同じ要素を持つベクトルとします。

これをコードに落とし込むと次のようになります。

```
import torch
from torch import nn
from math import sqrt

class MyLinear(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.in_features = in_features  # 入力値の数を保存
        self.out_features = out_features  # 出力値の数を保存

        # 重みを格納する行列の定義
        k = 1 / in_features
        weight = torch.empty(out_features, in_features).uniform_(-sqrt(k),
sqrt(k))
        self.weight = nn.Parameter(weight)
```

```
        # バイアスを格納するベクトルの定義
        bias = torch.empty(out_features).uniform_(-k, k)
        self.bias = nn.Parameter(bias)

    def forward(self, x):
        # ある層からの入力 x を受け取り、次の層への出力を計算する
        pass
```

MyLinear クラスに重みとバイアスを持たせる

このコードでは、torch.empty メソッドを使って、行列とベクトルを作成しています。このメソッドは、「初期化されていないデータを含んだテンソル（行列、ベクトルなど）」を作成するものです。引数には、作成する行列やベクトルのサイズを指定します。重みを格納する行列については「行数 , 列数」の順番で指定することには注意してください。そのため、ここでは「torch.empty(out_features, in_features)」としています（既に述べたように、行列は out_features 行、in_features 列となるため）。

その後に付いている「.uniform_(……)」というのは、初期化されていないデータを含んだ行列の初期化を行うための uniform_ メソッド呼び出しです。最後にアンダースコア「_」があることにも注意が必要です。これは、メソッドの呼び出しで使用したテンソルの内容をインプレースで書き換えることを意味しています（PyTorch では処理対象の内容をインプレースで書き換えるものには、その名前の最後にアンダースコアを付けることで、そのことを示すようになっています）。

ここでは、PyTorch の Linear クラスと同様な初期化を行うようにしています（k = 1/in_features として、-sqrt(k) ～ sqrt(k) の範囲から適当な値を使って初期化）。これはバイアスの初期値についても同様です。

ここまでは、__init__ メソッドのローカル変数 weight と bias を使って行列とベクトルを作成していましたが、最後にそれらを使って torch.nn.Parameter クラスのインスタンスを作成して、それらをインスタンス変数 self.weight と self.bias に代入しています（これは主に 1 行が長くなるのを避けるため、複数行に分けているだけです）。Parameter クラスのオブジェクトをニューラルネットワークモジュールの属性（インスタンス変数）とすると、それらを parameters メソッドの呼び出しで列挙できるようになります。最適化アルゴリズムの選択の際に、次のようなコードを書いていたことを覚えているでしょうか。

```
optimizer = torch.optim.SGD(net.parameters(), lr=0.003)
```

最適化アルゴリズムの選択時には、最適化の対象となるパラメーターを指定する

このコードでは、Net クラスのインスタンスである net オブジェクトに対して parameters メソッドを呼び出して

いますが、これによりそのニューラルネットワークモデルが持つ重みやバイアスを列挙するジェネレータを取得していました。Parameter クラスを使うと、この列挙されるパラメーターの一覧に、上で作成した重みやバイアスが自動的に追加されるようになっているのです。ここでは、この仕組みを利用することにしています。これを行わなければ、例えば次のような形で重みやバイアスをリスト（やタプル）に格納して、最適化アルゴリズムの選択時に渡すことになるでしょう。

```
optimizer = torch.optim.SGD([net.fc1.weight, net.fc1.bias, ……], lr=0.003)
```

最適化アルゴリズムの選択時に、その対象とする重みやバイアスを直接指定する

以上が __init__ メソッドで行っていることです。forward メソッドのコードは先の話として、MyLinear クラスと Linear クラスのインスタンスを生成して、重みやバイアスが似たようなものになっているかを見てみましょう。

```
INPUT_FEATURES = 4
HIDDEN = 5
OUTPUT_FEATURES = 1

linear1 = torch.nn.Linear(INPUT_FEATURES, HIDDEN)
linear2 = MyLinear(INPUT_FEATURES, HIDDEN)

print('Linear class')
for param in linear1.parameters():
    print(param)

print('¥nMyLinear class')
for param in linear2.parameters():
    print(param)
```

MyLinear クラスと Linear クラスの重みとバイアスを比較

実行結果を以下に示します。

```
INPUT_FEATURES = 4
HIDDEN = 5
OUTPUT_FEATURES = 1

linear1 = torch.nn.Linear(INPUT_FEATURES, HIDDEN)
linear2 = MyLinear(INPUT_FEATURES, HIDDEN)

print('Linear class')
for param in linear1.parameters():
    print(param)

print('¥nMyLinear class')
for param in linear2.parameters():
    print(param)
```

```
Linear class
Parameter containing:
tensor([[ 0.4406, -0.3006,  0.3959,  0.3785],
        [-0.3144,  0.1140,  0.3145, -0.3046],
        [ 0.0600, -0.0655,  0.4480, -0.4433],
        [ 0.2649,  0.1160,  0.4214, -0.2123],
        [ 0.3275,  0.0173, -0.3873,  0.2271]], requires_grad=True)
Parameter containing:
tensor([-0.1773,  0.4084, -0.2671,  0.0835, -0.3718], requires_grad=True)

MyLinear class
Parameter containing:
tensor([[-0.3449, -0.4724, -0.1639, -0.4831],
        [ 0.1979,  0.4244,  0.4354,  0.3254],
        [-0.2674, -0.1539,  0.3189, -0.0801],
        [-0.2241, -0.4901, -0.3768,  0.1123],
        [-0.1376, -0.0225, -0.3512,  0.1264]], requires_grad=True)
Parameter containing:
tensor([-0.0544, -0.0524, -0.2466, -0.1337,  0.2258], requires_grad=True)
```

実行結果

1 つ 1 つの数値は異なりますが、それらがおよそ -1/2 ～ 1/2 の範囲に収まっていること（ここでは INPUT_FEATURES ＝ 4 なので、k ＝ 1/in_features の値が 1/4 であり、その平方根が 1/2 となることに注意）と、重みを格納する行列がどちらも 5 行 4 列になっていること（out_features 列、in_features 列の行列なので、ここでは 5 行 4 列となります）、バイアスを格納するベクトルの要素がどちらも 5 つであることに注目してください。重ねて述べておきますが、これは PyTorch のやり方に従ったものであり、これとは異なる実装方法もあります。

forward メソッドを実装する

次に、入力値 x を受け取って、重みとバイアスを利用して、次の層に向けて出力する値を計算する処理を、forward メソッドに書いてみましょう。といっても、実は PyTorch が提供する torch.nn.functional.linear 関数（以下、linear 関数）を使えば、ほんの 1 行で書けてしまいます。

```
import torch
from torch import nn
from math import sqrt

class MyLinear(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.in_features = in_features  # 入力値の数を保存
        self.out_features = out_features  # 出力値の数を保存

        # 重みを格納する行列の定義
        k = 1 / in_features
        weight = torch.empty(out_features, in_features).uniform_(-sqrt(k),
sqrt(k))
        self.weight = nn.Parameter(weight)

        # バイアスを格納するベクトルの定義
        bias = torch.empty(out_features).uniform_(-k, k)
        self.bias = nn.Parameter(bias)

    def forward(self, x):
        return torch.nn.functional.linear(x, self.weight, self.bias)
```

forward メソッドの実装

linear 関数は、入力値 x、行列 self.weight、ベクトル self.bias を受け取り、それに対して、第 4 回で見たような「出力値＝入力値 × 重み＋バイアス」に相当する行列演算を行ってくれます（この詳細についてはまた後ほど見てみます）。

取りあえずは MyLinear クラスが完成したものとして、実際にこれが PyTorch の Linear クラスと同様に使えるかを試してみましょう。

MyLinear クラスを使ってみる

ここではあやめの品種を分類するものとして、以下のコードでデータセットを用意します。

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import torch

iris = load_iris()
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target)

X_train = torch.from_numpy(X_train).float()
y_train = torch.tensor([[float(x)] for x in y_train])
X_test = torch.from_numpy(X_test).float()
y_test = torch.tensor([[float(x)] for x in y_test])
```

データセットの準備

MyLinear クラスを使った、ニューラルネットワーククラスのコードは次のようになります（これまでの Net クラスと違うのは、torch.nn.Linear クラスではなく、MyLinear クラスを使うところだけです）。

```
INPUT_FEATURES = 4
HIDDEN = 5
OUTPUT_FEATURES = 1

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = MyLinear(INPUT_FEATURES, HIDDEN)
        self.fc2 = MyLinear(HIDDEN, OUTPUT_FEATURES)

    def forward(self, x):
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        return x
```

MyLinear クラスを使って、全結合を行う Net クラスの定義

学習をするコードもこれまでに見てきた通りです。

```
net = Net()

criterion = nn.MSELoss()
optimizer = torch.optim.SGD(net.parameters(), lr=0.003)

EPOCHS = 2000
for epoch in range(EPOCHS):
    optimizer.zero_grad()
    outputs = net(X_train)
    loss = criterion(outputs, y_train)
    loss.backward()
    optimizer.step()

    if epoch % 100 == 99:
        print(f'epoch: {epoch+1:4}, loss: {loss.data}')

print('training finished')
```

MyLinear クラスを使って学習を行うコード

以上のコードをセルに記述、実行した結果を以下に示します。

```
print('training finished')
```

```
epoch:  100, loss: 0.5252503156661987
epoch:  200, loss: 0.49113455414772034
epoch:  300, loss: 0.4539352357387543
epoch:  400, loss: 0.41156408190727234
epoch:  500, loss: 0.36538785696029663
epoch:  600, loss: 0.31782034039497375
epoch:  700, loss: 0.27164319157600403
epoch:  800, loss: 0.2292831987142563
epoch:  900, loss: 0.19235317409038544
epoch: 1000, loss: 0.16154353320598602
epoch: 1100, loss: 0.13676872849464417
epoch: 1200, loss: 0.11742796748876572
epoch: 1300, loss: 0.10266588628292084
epoch: 1400, loss: 0.09157230705022812
epoch: 1500, loss: 0.08330662548542023
epoch: 1600, loss: 0.07715743035078049
epoch: 1700, loss: 0.07255769520998001
epoch: 1800, loss: 0.06907449662685394
epoch: 1900, loss: 0.06638820469379425
epoch: 2000, loss: 0.06426838785409927
training finished
```

実行結果

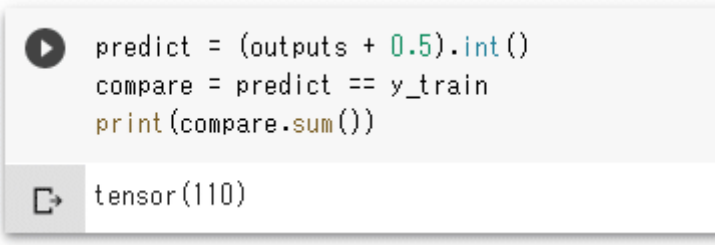
→目次に戻る

実行結果を見ると、損失も徐々に 0 に近づいていることから、MyLinear クラスがうまく動いていることが予想できます。変数 outputs には最後の学習で推測した結果が格納されているので、これと教師データとの比較を行うことで、ここではこのニューラルネットワークモデルがどのくらいの精度を持っているかを確認してみましょう（本来は検証まで行うべきですが、ここでは省略します）。といっても、このコードもこれまでの回で紹介してきたものと変わりません（推測結果を四捨五入して、教師データである y_train の各要素と比較して、何個が True となったかを数えているだけです）。

```
predict = (outputs + 0.5).int()
compare = predict == y_train
print(compare.sum())
```

訓練データを基にした推測結果と教師データとの比較

実行結果を以下に示します。



実行結果

推測結果と教師データとでは 112 個中の 110 個が一致しました。なかなかよい結果となりました。

というわけで、PyTorch が提供する Linear クラスを簡素化した MyLinear クラスの実装はこれで終わりです。最後に forward メソッドで行っている処理について、もう少し詳しく見ていくので、興味のある方は読み進めてみてください。

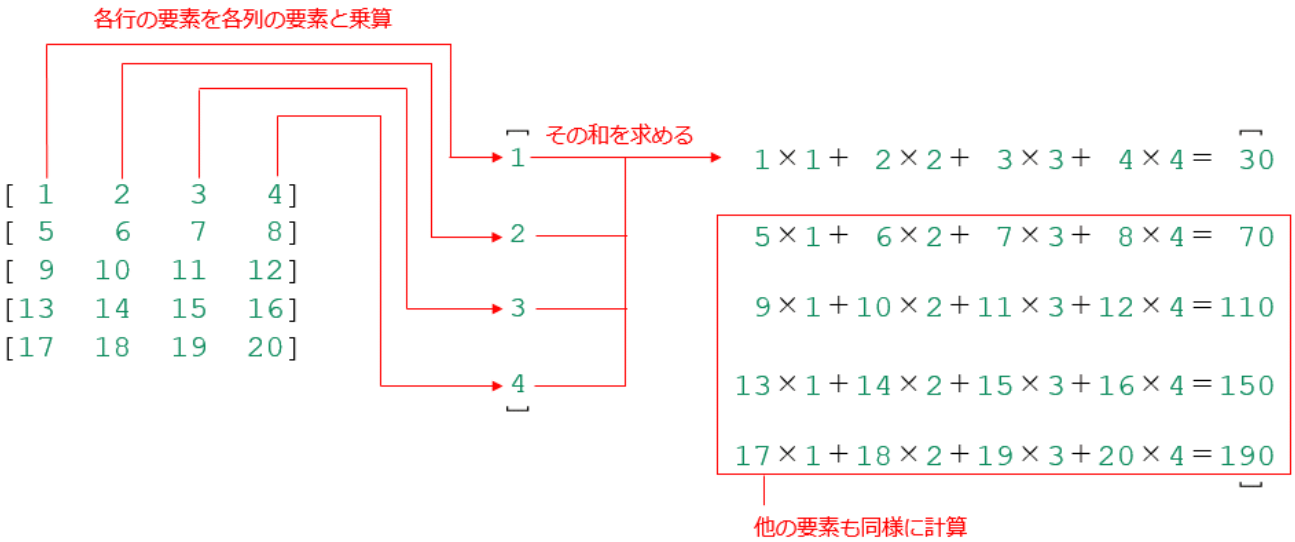
出力値を計算するコードを自分で書く

ここでは、MyLinear クラスから少し離れて、重みを格納する行列と同じ 5 行 4 列の行列を変数 w に、バイアスを格納するベクトルと同じ 5 要素のベクトルを変数 b に、入力値となる 4 要素からなるベクトルを変数 x に代入して、それらを基に出力値を計算してみることにしましょう。それぞれの値には、分かりやすくなるように、浮動小数点数ではなく整数値を用います。

```
w = torch.tensor([[1, 2, 3, 4],
                  [5, 6, 7, 8],
                  [9, 10, 11, 12],
                  [13, 14, 15, 16],
                  [17, 18, 19, 20]])
b = torch.tensor([1, 2, 3, 4, 5])
x = torch.tensor([1, 2, 3, 4])
```

変数 w、b、x は重み、バイアス、入力値と同じ形式でデータが格納されている

ここで注意したいのは、入力値 x と重み w の乗算がどのように行われるかです。簡単にまとめると、重みを格納する行列の各行の要素（4 つ）と、入力値 x の 4 つの要素をそれぞれ乗算したものの和にバイアスの値を加算することで、「出力値＝重み × 入力値＋バイアス」という計算が行われます（バイアスも含めて一度の行列演算で全てを計算する方法についても後で見ます）。行列とベクトルの乗算の形で図にすると次のようになります。



行列（重み）とベクトル（入力値）の乗算

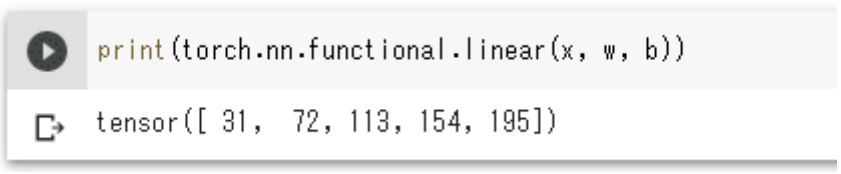
上の図では、ベクトルが 1 行 1 列の行列のように表されている点に注意してください。このとき、1 つ目の行列の各行の各要素を、2 つ目のベクトルの各要素と掛け合わせたものの和を要素とするベクトルが計算結果となります。この結果、5 要素から成るベクトルが得られるというわけです。

そして、実際にこのような計算を行ってくれるのが、PyTorch が提供する linear 関数なのでした。実際に計算してみましょう。

```
print(torch.nn.functional.linear(x, w, b))
```

linear 関数を使って出力値を求める

実行結果を以下に示します。



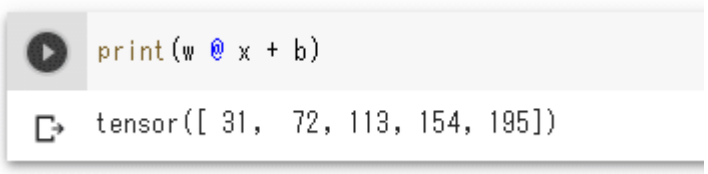
実行結果

これと同じ結果を、自分の手で求めるにはどうすればよいでしょう。簡単なのは Python の @ 演算子を使うことです。

```
print(w @ x + b)
```

Python の @ 演算子を使う

実行結果を以下に示します。

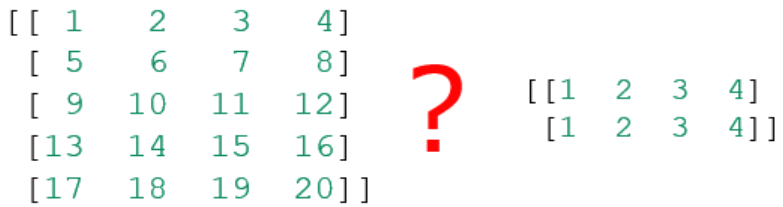


実行結果

同じ結果になりましたね。ところで、今は入力値がベクトル 1 つだけでした。先ほどの図からも分かるように、このときにはベクトルが 1 行 1 列の行列のように扱われ、その要素は縦軸方向に並べられて、重みを格納する行列の行要素とうまいこと乗算されています（この縦軸方向に要素が並べられるのがポイントです。後述）。しかし、ニューラルネットワークには通常、訓練データなどとして多くの入力値（ベクトルを行列にまとめたもの）が一度に渡されます。この場合の計算について少し考えてみましょう。

行列と行列から出力値を計算する方法

以下では、入力値が行列である場合を単純化した例として、入力値 x には [1, 2, 3, 4] というベクトルを 2 つ格納する行列（[[1, 2, 3, 4], [1, 2, 3, 4]]）が代入されているものとします。イメージとしては、次のような 2 つの行列から出力値を計算することになります。



行列（重み）と行列（入力値）の乗算

このような2つの行列を乗算するにはどうすればよいのでしょうか。実はこれには少し工夫が必要になります。その方法を見る前に、入力値を格納する行列から行単位でデータを取り出す方法を見てみましょう。なぜかといえば、そうすることで、先ほど見た、行列とベクトル（この場合は、入力値を格納する行列の行要素）の乗算に問題を還元できるからです。実際にこれを行うコードを以下に示します。

```
x = torch.tensor([[1, 2, 3, 4], [1, 2, 3, 4]])  # 入力値 x は 2 行 4 列の行列

result = torch.zeros(len(x), len(w))  # 2 行 5 列で 0 を要素とする行列
idx = 0
for item in x:
    result[idx] = w @ item + b
    idx += 1

print(result)
```

行列の要素（ベクトル）ごとに出力値を計算する方法

このコードでは入力値 x から 1 行ずつベクトルを取り出し、それを先ほど見た @ 演算子を使って「w @ item + b」という計算をすることで、出力値を得て、それを「入力値の行数と同じ行数で（この場合は 2）、出力値の数と同じ数だけの列数（この場合は 5）」の行列に順次代入していくコードです（torch.zeros メソッドは、指定した行数、列数で要素を 0 とするテンソルを作成します）。

実行結果を以下に示します。

```
x = torch.tensor([[1, 2, 3, 4], [1, 2, 3, 4]])  # 入力値xは2行4列の行列

result = torch.zeros(len(x), len(w))  # 2行5列で0を要素とする行列
idx = 0
for item in x:
    result[idx] = w @ item + b
    idx += 1

print(result)
```

```
tensor([[ 31.,  72., 113., 154., 195.],
        [ 31.,  72., 113., 154., 195.]])
```

実行結果

[1, 2, 3, 4] というベクトルと重みの行列の乗算結果は [31, 72, 113, 154, 195] という 5 要素のベクトルでした。上のコードではこれと同じベクトルを 2 つ要素とする行列だったので、これは予想通りの結果といえます。

そして、ここで見た方法よりもスマートに行列と行列を乗算する方法はもちろんあります（NumPy や PyTorch などのフレームワークは、まさにこのような行列操作＝テンソル操作を得意としたフレームワークなのですから）。

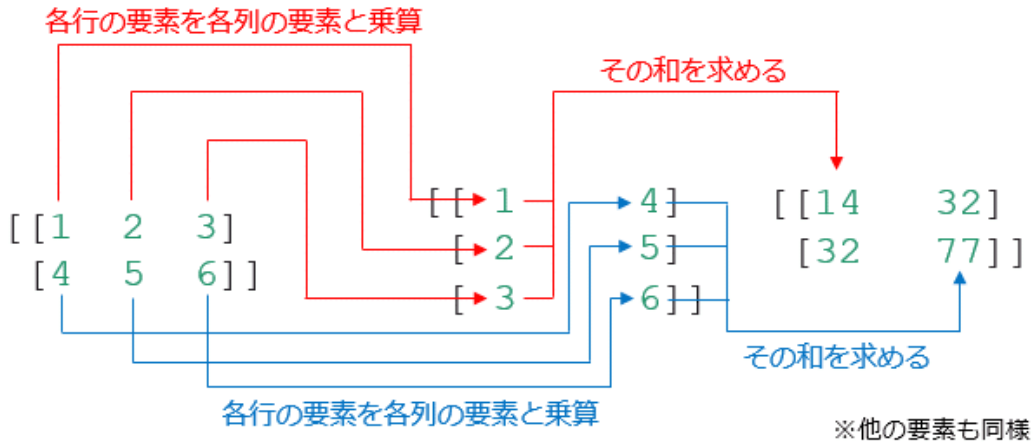
行列と行列の乗算の基本

実際にその方法を見ていく前に行列の乗算について簡単に説明をしておきましょう。実は、行列同士の乗算では、「1 つ目の行列に格納されている各行の要素と、2 つ目の行列に格納されている各列の要素」の乗算が行われます。そして、それらの和を要素とする行列が作られます。言葉では分かりにくいので、簡単な例を示します。

```
foo = torch.tensor([[1, 2, 3], [4, 5, 6]])  # 2 行 3 列の行列
bar = torch.tensor([[1, 4], [2, 5], [3, 6]])  # 3 行 2 列の行列

print(foo)
print(bar)
```

2 行 3 列の行列と、3 行 2 列の行列

ここでは、2 行 3 列の行列 foo と 3 行 2 列の行列 bar を作成しています。これらの乗算がどう行われるかを以下に示します。



行列 foo と行列 bar の乗算

行列 foo と行列 bar の乗算は、行列 foo の各行の要素と行列 bar の各列の要素を乗じた値の和を要素とした 2 行 2 列の行列を生じるということです（上の図では明示していませんが、行列 foo の 1 行目と行列 bar の 2 列目でも同様に計算が行われ、それが結果として得られる行列の 2 行 2 列目の値となります。行列 foo の 2 行目と行列 bar の 1 列目も同様で、これが結果として得られる行列の 2 行 1 列目の値となります）。実際にコードで試してみましょう（ここでは Python の @ 演算子を使用します）。

```
print(foo @ bar)
```

行列 foo と行列 bar の乗算

実行結果を以下に示します。

```
[12] foo = torch.tensor([[1, 2, 3], [4, 5, 6]])  # 2行3列の行列
     bar = torch.tensor([[1, 4], [2, 5], [3, 6]])  # 3行2列の行列

     print(foo)
     print(bar)

     tensor([[1, 2, 3],
             [4, 5, 6]])
     tensor([[1, 4],
             [2, 5],
             [3, 6]])
```

行列fooと行列barの乗算

```
print(foo @ bar)

tensor([[14, 32],
        [32, 77]])
```

実行結果

重要なのは、行列 foo の列数と、行列 bar の行数が一致していることです。そうでないと行列同士の乗算ができません。これが行列同士を乗算する際の基本です。

重みを格納している行列と入力値を格納している行列の乗算

基本を見たところで、大本の問題に立ち返りましょう。ここでは、次のような行列同士の乗算を行うことが目的でした。

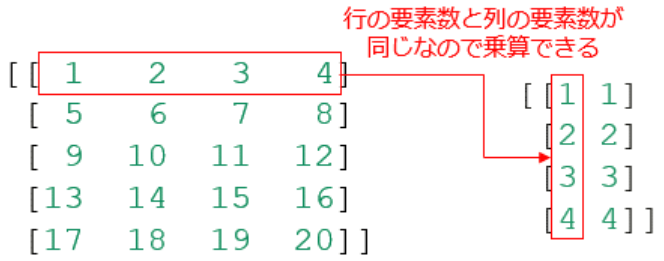
```
[[ 1   2   3   4]
 [ 5   6   7   8]
 [ 9  10  11  12]
 [13  14  15  16]
 [17  18  19  20]]   ?   [[1  2  3  4]
                          [1  2  3  4]]
```

重みを格納する 5 行 4 列の行列と、入力値である 2 行 4 列の行列では、前者の列数と後者の行数が異なるので乗算ができない（再掲）

しかし、この 2 つの行列では重み w の列数が 4 であるのに対して、入力値 x の行数は 2 となっています。これでは乗算できないのは既に述べた通りです。加えて、行列と行列の乗算では 1 つ目の行列に格納されている各行の各要素が、2 つ目の行列に格納されている各列の各要素と掛け合わされるのでした。ところが、入力値を格納する行列では、重み行列の行要素と掛け合わせたい要素が縦軸方向ではなく、横軸方向に並んでいます。

つまり、ここで行いたいのは、実は重みを格納している行列の行要素と、入力値を格納している行要素の乗算です（行列とベクトルの乗算ではこれが自然と行えていたということです。縦軸方向にベクトルの要素が並んでいたことを思い出しましょう）。

逆にいえば、入力値が行列のときには、行と列を入れ替えることで、うまく計算ができそうです。行と列を入れ替えるとは、この場合、2 行 4 列だった行列を、4 行 2 列の行列にして、縦軸／横軸を入れ替えて、要素を並べ替えるということです。以下に例を示します。



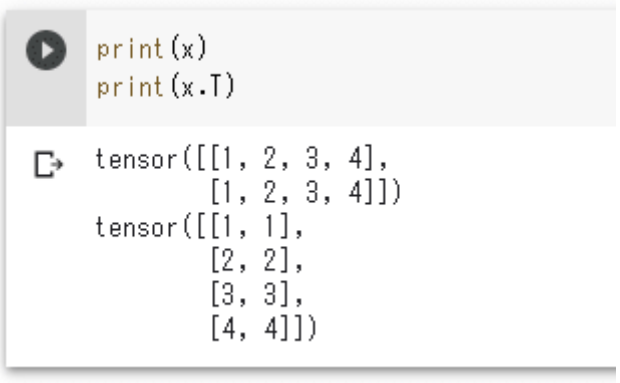
5 行 4 列の行列と 2 行 4 列の行列の乗算

このような、元の行列から行と列を入れ替えた行列のことを「転置行列」と呼びます。そして、転置行列を得るには、PyTorch のテンソルが持つ属性 T を利用できます（この「T」は転置行列を意味する「Transposed Matrix」からきています）。実際に試してみましょう。

```
print(x)
print(x.T)
```

転置行列

実行結果を以下に示します。



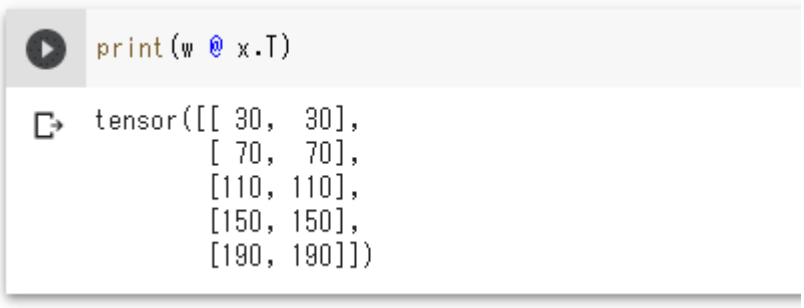
実行結果

2 行 4 列の行列だったのが、4 行 2 列の行列に転置できたのが分かります。これと 5 行 4 列の行列である重み w であれば、次のように計算できるでしょう。

```
print(w @ x.T)
```

重み w と x の軸を入れ替えたものを乗算

しかし、実行結果を見てください。



実行結果

この結果は望んでいたものではありません（欲しいのは 2 行 5 列の行列です）、バイアス b との加算も無理です。もちろん、この結果に対して、さらに行と列を入れ替えてもよいのですが、重み w の方を転置してみましょう。

```
print(w)
print(w.T)

print(x @ w.T)
print(x @ w.T + b)
```

重みを格納する行列を転置行列とした場合

ここでは入力値 x が @ 演算子の前に、重み w の軸を入れ替えたものが後に置かれていることにも注意してください。これにより、次のような行列同士の乗算が行われることになります。

```
[[1  2  3  4]      @      [[ 1   5   9  13  17]
 [1  2  3  4]]             [ 2   6  10  14  18]
                           [ 3   7  11  15  19]
                           [ 4   8  12  16  20]]
```

2 行 4 列の行列と 4 行 5 列の行列の乗算

これなら、1 つ目の行列の列数と、2 つ目の行列の行数が同じなので、うまく乗算できるはずですし、最初に見た行列とベクトルの乗算で行っていたものと同様な計算が行われる（＝出力値がきちんと計算できる）ことも分かります。計算結果は 5 要素のベクトルを要素とする行列となるので、バイアスの加算も可能です。

実行結果を以下に示します。

```
print(w)
print(w.T)

print(x @ w.T)
print(x @ w.T + b)
```

```
tensor([[ 1,  2,  3,  4],
        [ 5,  6,  7,  8],
        [ 9, 10, 11, 12],
        [13, 14, 15, 16],
        [17, 18, 19, 20]])
tensor([[ 1,  5,  9, 13, 17],
        [ 2,  6, 10, 14, 18],
        [ 3,  7, 11, 15, 19],
        [ 4,  8, 12, 16, 20]])
tensor([[ 30,  70, 110, 150, 190],
        [ 30,  70, 110, 150, 190]])
tensor([[ 31,  72, 113, 154, 195],
        [ 31,  72, 113, 154, 195]])
```

実行結果

うまくいっていることが分かります。

実際に、forward メソッドにこの結果を組み込むと次のように書けます。

```
import torch
from torch import nn
from math import sqrt

class MyLinear(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features

        k = 1 / in_features
        weight = torch.empty(out_features, in_features).uniform_(-sqrt(k),
sqrt(k))
        self.weight = nn.Parameter(weight)

        bias = torch.empty(out_features).uniform_(-k, k)
        self.bias = nn.Parameter(bias)

    def forward(self, x):
        #result = torch.zeros(len(x), self.out_features)
        #idx = 0
        #for item in x:
        #    result[idx] = self.weight @ item + self.bias
        #    idx += 1
        #return result
        return x @ self.weight.T + self.bias
        #return torch.nn.functional.linear(x, self.weight, self.bias)
```

転置行列を利用した行列演算を用いて forward メソッドを実装する

コメントアウトした形で、先ほどの行ごとにデータを抜き出して計算するコードも含めてあります。興味のある方は、こちらのコードも試してみてください。

バイアスを重みに含めて行列の乗算を行ってみる

最後に、バイアスを行列演算に含める方法についても見ておきましょう。ざっくりとした話になりますが、重みの 1 つの要素としてバイアスを捉え、それに常に入力値 1 を乗じることで、重みと入力値の乗算結果にバイアスの値を加算するのではなく、行列の乗算にバイアスを含めることが可能です。

実際のコードを以下に示します。

```
w2 = torch.cat([b.unsqueeze(0), w.T])
ones = torch.full((len(x), 1), fill_value=1, dtype=torch.long)
print('w2:', w2)
print('ones:', ones)
x2 = torch.cat([ones, x], dim=1)
print('x2:', x2)
```

重みを格納する行列の要素とバイアスを捉える

詳しい説明は省略しますが、変数 w2 の内容は重み w の転置行列の先頭にバイアス b を付加したものです。変数 ones には、入力値 x の行数と同じ行数で、列数が 1 の行列（その値は全て 1）が代入されます。変数 x2 には、変数 x の第 1 列の前に変数 ones の行列を付加したものが代入されます。これにより、以下に示す実行結果のような行列が得られます。

```
w2 = torch.cat([b.unsqueeze(0), w.T])
ones = torch.full((len(x), 1), fill_value=1, dtype=torch.long)
print('w2:', w2)
print('ones:', ones)
x2 = torch.cat([ones, x], dim=1)
print('x2:', x2)
```

```
w2: tensor([[ 1,  2,  3,  4,  5],
        [ 1,  5,  9, 13, 17],
        [ 2,  6, 10, 14, 18],
        [ 3,  7, 11, 15, 19],
        [ 4,  8, 12, 16, 20]])
ones: tensor([[1],
        [1]])
x2: tensor([[1, 1, 2, 3, 4],
        [1, 1, 2, 3, 4]])
```

実行結果

バイアスの要素が新しい重み w2 の先頭行にあること、新しい入力 x2 の先頭に 1 が追加されていることを確認してください（添字を使ってバイアスを表すときにはよく w0 のように、またこれと乗じられる値 1（入力値）はよく x0 のように表現されることがあります。ここでは、これに合わせて、インデックス 0 の位置にこれらを挿入しています。もちろん、それぞれの末尾にこれらを追加するやり方もあるでしょう）。先ほど見た「出力値＝入力値 × 重み+バイアス」という計算は、実はこのようにすることで、一度の行列演算「入力値 × 重み」という計算にまとめることができます。例えば、入力値 x2 の最初の行と重み w2 の最初の列との乗算であれば、その計算は次のようになります。

- 1×1 + 1×1 + 2×2 + 3×3 + 4×4

これを計算すると「1 + 1 + 4 + 9 + 16」で「31」となります。これまでに計算してきた「出力値＝重み × 入力値+バイアス」と同じことが行われていることに注目してください（ただし、バイアスを含んだ計算を行う項が最初にあります）。

実際にそうなるかどうかを実際に試すコードが以下です。今度は重み w2 は既に転置行列となっているので、属性 T で転置行列を得ていないことには注意してください。

```
print(x2 @ w2)
```

重みを格納する行列にバイアスを含めて出力値を計算する

実行結果を以下に示します。

```
print(x2 @ w2)

tensor([[ 31,  72, 113, 154, 195],
        [ 31,  72, 113, 154, 195]])
```

実行結果

これまでと同様な結果が得られたことから、重みと入力値を表す行列の乗算の中にバイアスが含まれたことが分かりました。

今回は、全結合を行うクラスを自分で作りながら、そこで実際にどんな処理が行われているのかを見てきました。あやめのデータセットを扱う例は今回で終わりとして、次回は MNIST を用いた画像認識の手順を見ていくことにしましょう。

MNIST の手書き数字を
全結合型ニューラルネットワークで処理してみよう

より高度なニューラルネットワークの作成に移る前に、これまでの知識を使って、MNIST の手書き数字を認識するプログラムを作ってみます。

（2020 年 05 月 21 日）

前回までは、あやめの品種の推測を題材にニューラルネットワークの基本となる要素について見てきました。今回からは手書き数字の認識を題材にもう少し高度な話題を見ていきましょう。

MNIST

今回からは 0 ～ 9 までの手書き数字を集めた MNIST データベースを使用して、それらの数字を認識するニューラルネットワークモデルを作成します。

MNIST データベースに含まれている手書き文字（抜粋）

MNIST データベースには、上に示したような手書きの数字（と対応する正解ラベル）が訓練データとして 6 万個、テストデータとして 1 万個格納されています。この膨大な数のデータを使用して、手書きの数字を認識してみようというのが目標です。

今回は、これまでに見てきた全結合型のニューラルネットワークを作成して、これを実際に試してみましょう。今回紹介するコードはここで公開しているので、必要に応じて参照してください。

データセットの準備と探索

本連載で使用している機械学習フレームワークである PyTorch には今述べた MNIST を手軽に扱えるようにするための torchvision パッケージが用意されています（「vision」が付いているのは、このパッケージがコンピューターによる視覚の実現＝コンピュータービジョンに由来するのでしょう）。このパッケージを使って実際に MNIST データベースからデータセットを読み込んでみましょう。

```
import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])

BATCH_SIZE = 20

trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      transform=transform, download=True)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE,
shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                     transform=transform, download=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE,
shuffle=False)
```

MNIST データベースからデータセットを読み込むコード

最初の 3 行では torch パッケージ、torchvision パッケージ、torchvision パッケージが提供する transforms パッケージをインポートしています。最後の transforms パッケージには、画像（を構成する数値データ）を変換するためのクラスが含まれていて、これを使って、MNIST データベースに格納されているデータを PyTorch で扱えるように変換作業が行われます。実際、その直後で、tranforms パッケージに含まれる ToTensor クラス、Normalize クラスを組み合わせた変換処理を行うオブジェクトを変数 trnasform に代入しています（これについては後で簡単に見ます）。

定数 BATCH_SIZE の値「20」は訓練データ（とテストデータ）から一度に何個のデータを読み込むかを指定する値（バッチサイズ）です。前回まで使用していたあやめのデータセットは 150 個という極めて少ない量のデータセットでしたが、今回のデータセットには学習用とテスト用に合わせて 7 万個のデータがあるので、学習／テストを行う際にはそれらを分割して読み込むことにします（その下の変数 trainloader と testloader と合わせて、これらについても後述します）。

変数 trainset と testset には、訓練データとテストデータが正解ラベル込みで代入されます。これらから実際にデータを取り出すときに使用するのが変数 trainloader と testloader に代入されている DataLoader クラスのインスタンスです。

変数 trainset と testset には、torchvision.datasets モジュールが提供する MNIST クラスのインスタンスが代入されています。このインスタンスの生成時には、次のような引数を指定します。

- root：データセットファイルを置くディレクトリを指定
- train：訓練データを生成するか、テストデータを生成するかを指定
- transform：trainset ／ testset からデータを取り出す際に、MNIST の生データに対して行う変換処理を指定。ここでは変数 transform を指定
- download：必要に応じてインターネットから MNIST データセットをダウンロードするかどうかを指定

また、変数 trainloader と testloader に代入される、DataLoader クラスのインスタンス生成時には、それぞれに対応するデータセットに加えて、次のような引数を指定しています。

- batch_size：一度に読み込むデータ数（バッチサイズ）を指定
- shuffle：読み込むデータをシャッフルするかどうかを指定

変数 trainloader に代入する DataLoader インスタンスの生成では引数 shuffle に True を指定しています。これは変数 trainset を使って学習を行う際に、最初にデータをランダムに並べ替えることを意味しています。その学習で 6 万個のデータを使い切って学習が一区切り付いた後（この区切りのことを「エポック」といいます。つまり、この場合は 6 万個のデータが 1 つのエポックとして扱われます）、同じデータセットを使って次のエポックを開始する際には、またデータセット内のデータがランダムに並べ替えられます。これは、同じ並びでデータを取り出すのではなく、6 万個のデータから任意の順序でデータをピックアップすることで、学習結果に偏りを生じさせないようにするためです。テストデータについては shuffle を False にしていますが、これは最終的な確認を行うという観点から、シャッフルの必要がないためです。

次に、読み込んだデータセットから先頭のデータを少し見てみましょう。データベースから読み込んだデータセットは変数 trainset の data 属性にアクセスすることでアクセスできるので、インデックス 0 を指定すればその先頭要素が得られます。

```
print(f'image: {len(trainset.data[0])} x {len(trainset.data[0][0])}')
for item1 in trainset.data[0]:
    for item2 in item1:
        print(f'{item2.data:4}', end='')
    print()
```

先頭にある手書き文字の値を表示するコード

このコードは上で述べた通り、変数 trainset の data 属性にアクセスして、その先頭にある手書き文字を構成する数値を二重ループで表示するものです（画像は 2 次元のデータとなっているので、このように二重ループで処理をしています）。

このコードを実行すると次のような実行結果になります（環境によっては表示が崩れるかもしれません）。

```
print(f'image: {len(trainset.data[0])} x {len(trainset.data[0][0])}')
for item1 in trainset.data[0]:
    for item2 in item1:
        print(f'{item2.data:4}', end='')
    print()
```

```
image: 28 x 28
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   3  18  18  18 126 136 175  26 166 255 247 127   0   0   0   0
   0   0   0   0   0   0   0   0  30  36  94 154 170 253 253 253 253 253 225 172 253 242 195  64   0   0   0   0
   0   0   0   0   0   0   0  49 238 253 253 253 253 253 253 253 253 251  93  82  82  56  39   0   0   0   0   0
   0   0   0   0   0   0   0  18 219 253 253 253 253 253 198 182 247 241   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0  80 156 107 253 253 205  11   0  43 154   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0  14   1 154 253  90   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0 139 253 190   2   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0  11 190 253  70   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0  35 241 225 160 108   1   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0  81 240 253 253 119  25   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0  45 186 253 253 150  27   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  16  93 252 253 187   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0 249 253 249  64   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  46 130 183 253 253 207   2   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0  39 148 229 253 253 253 250 182   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0  24 114 221 253 253 253 253 201  78   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0  23  66 213 253 253 253 253 198  81   2   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0  18 171 219 253 253 253 253 195  80   9   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0  55 172 226 253 253 253 253 244 133  11   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0 136 253 253 253 212 135 132  16   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
```

実行結果

最初にある「image: 28 x 28」というのは MNIST データベースに含まれている手書き文字が 28×28 ピクセルのサイズになっていることを示すものです。その後には 0 ～ 255 の範囲の整数値が表示されています。つまり、1 つの手書き数字は 28×28 のサイズで、各ピクセルの値は 0 ～ 255 の範囲にあるということです。さらに、全体をぼんやり眺めてみると、数字らしいものが浮かび上がっていることも分かります。このデータは何の数字を表しているのでしょうか。何となく想像は付きますが、実際の画像も見てみましょう。

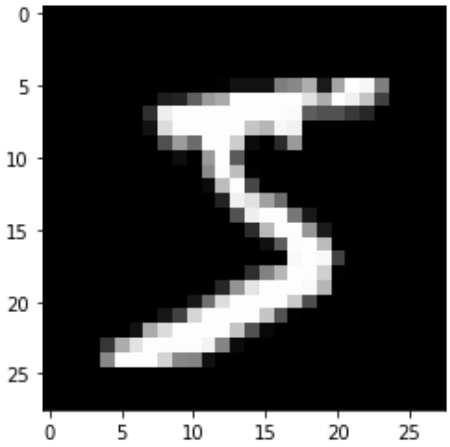
```
import matplotlib.pyplot as plt


plt.imshow(trainset.data[0], cmap='gray')
```

Matplotlib を使って、先頭の手書き数字を表示する

実行結果は次の通りです。

```
import matplotlib.pyplot as plt

plt.imshow(trainset.data[0], cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7fe0cf9f03c8>
```



実行結果

これは数字の「5」のように見えますが、本当にそうか、実際に対応する正解ラベルを表示してみましょう。正解ラベルは trainset の targets 属性に格納されているので、「print(trainset.targets[0])」としてもよいのですが、ここではちょっと違う方法でデータを取り出してみます。

```
image, label = trainset[0]
print(label)
#print(trainset.targets[0])
```

先頭の手書き数字に対応する正解ラベルを表示する

先ほどとの違いは、変数 trainset に対して直接インデックスを指定しているところと、その戻り値が 2 つある（2 つの要素で構成されるタプル）ということです。torchvision.datasets.MNIST クラスは torch.utils.data.Dataset クラスを（間接的に）継承したクラスで、インデックス指定を行うことで、そのインデックスに対応するデータと正解ラベルを取得できるようになっています。ここでは先頭の手書き文字の正解ラベルを取得するのに、これを使ってみました。実行結果を以下に示します。

```
image, label = trainset[0]
print(label)
#print(trainset.targets[0])
```

```
5
```

実行結果

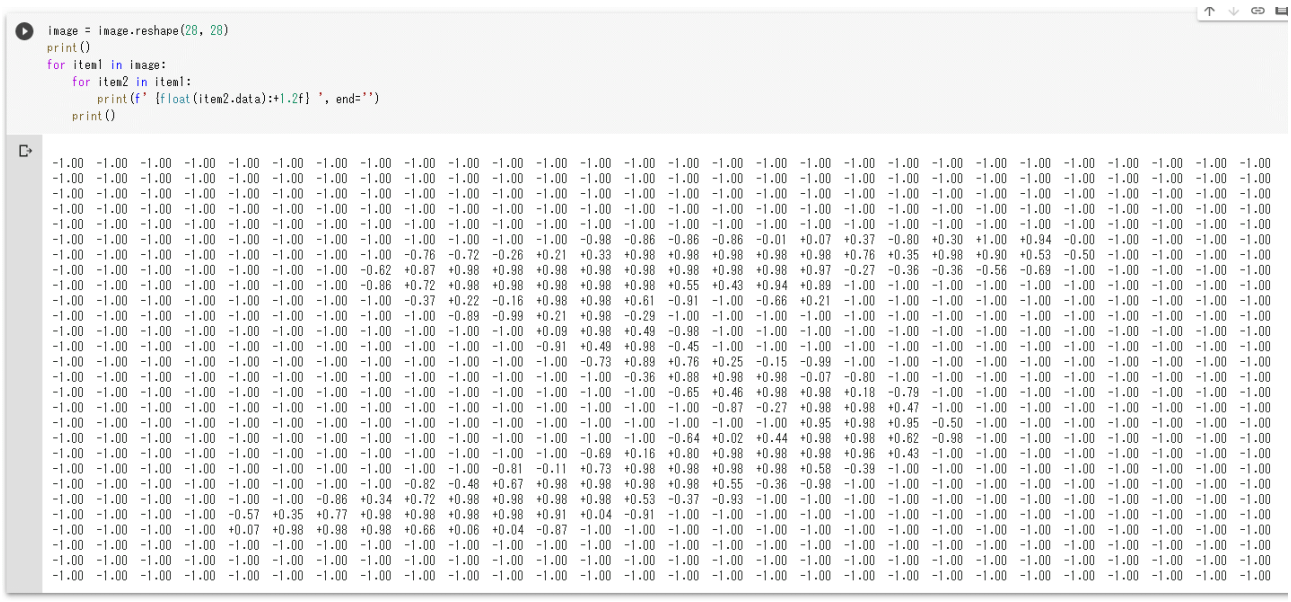
予想通りに「5」と表示されました。先ほどの数字は 5 を描いたものだったということです。ここで、上の手順で変数 image に取り出したデータについても見てみましょう。

```
image = image.reshape(28, 28)
print()
for item1 in image:
    for item2 in item1:
        print(f' {float(item2.data):+1.2f} ', end='')
    print()
```

MNIST クラスのインスタンスに対してインデックス指定を行って得た画像データを表示

最初の行で行っているのは、変数 image に取り出したデータが「28×28 の画像データをただ一つの要素とする配列」となっているので、これを「28×28 の画像データ」に変換する処理です。その後は、trainset.data[0] の値を表示するのと同様な処理です（ただし、浮動小数点数を表示するようになっている点には注意してください）。

実行結果を以下に示します（これも環境によっては表示が崩れるかもしれません）。



実行結果

注目してほしいのは、先ほどの MNIST データベースに格納されていた 0 ～ 255 の値が、今度は -1.0 ～ 1.0 の値に変換されている点です。MNIST クラスのインスタンス生成では次のようなコードを書いていました。

```
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])

# …… 省略 ……

trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                      transform=transform, download=True)
# …… 省略 ……
```

MNIST クラスのインスタンスを生成するコード

先ほどのコードでは、trainset に対してインデックス指定などの手段で値を得るときには、0 ～ 255 の値が -1.0 ～ 1.0 の値に変換されていました。そして、上記のコードを見ると、transform では、ToTensor クラスと Normalize クラスのインスタンスが指定されています。実は前者は 0 ～ 255 の範囲の数値を 0 ～ 1.0 の範囲の浮動小数点数値に変換するためのものです。そして、後者はインスタンス生成時に第 1 引数に指定した値を m、第 2 引数に指定した値を s としたときに、おおざっぱにいうと「出力＝（入力－ m）／ s」という計算を行うものです（m と s がタプルになっているのは、RGB 値など複数のチャネルで画像が構成されている場合に、チャネルごとにそれらを指定できるようにするため）。

ここではどちらも 0.5 なので、0.0 ～ 1.0 の範囲の浮動小数点数値が -1.0 ～ 1.0 の範囲の数値へと変換されます。例えば、入力（ToTensor クラスで変換された値）が 0 であれば、Normalize による変換の結果は「(0.0 － 0.5）／ 0.5 ＝ -1.0」となります。入力が 0.5 なら「(0.5 － 0.5）／ 0.5 ＝ 0.0」に、入力が 1 なら「(1.0 － 0.5）／ 0.5 ＝ 1.0」となります。

このような変換を自動的に行うのが、transform 引数の役割です。実際に学習を行う段階では、ループ処理の中で繰り返し trainset からデータを取り出して、それをニューラルネットワークに入力していきますが、このときに今述べたような変換処理が自動的に行われます。なお、このようなある範囲の値を別の一定範囲の値へと変換することを正規化（normalize）と呼びます。

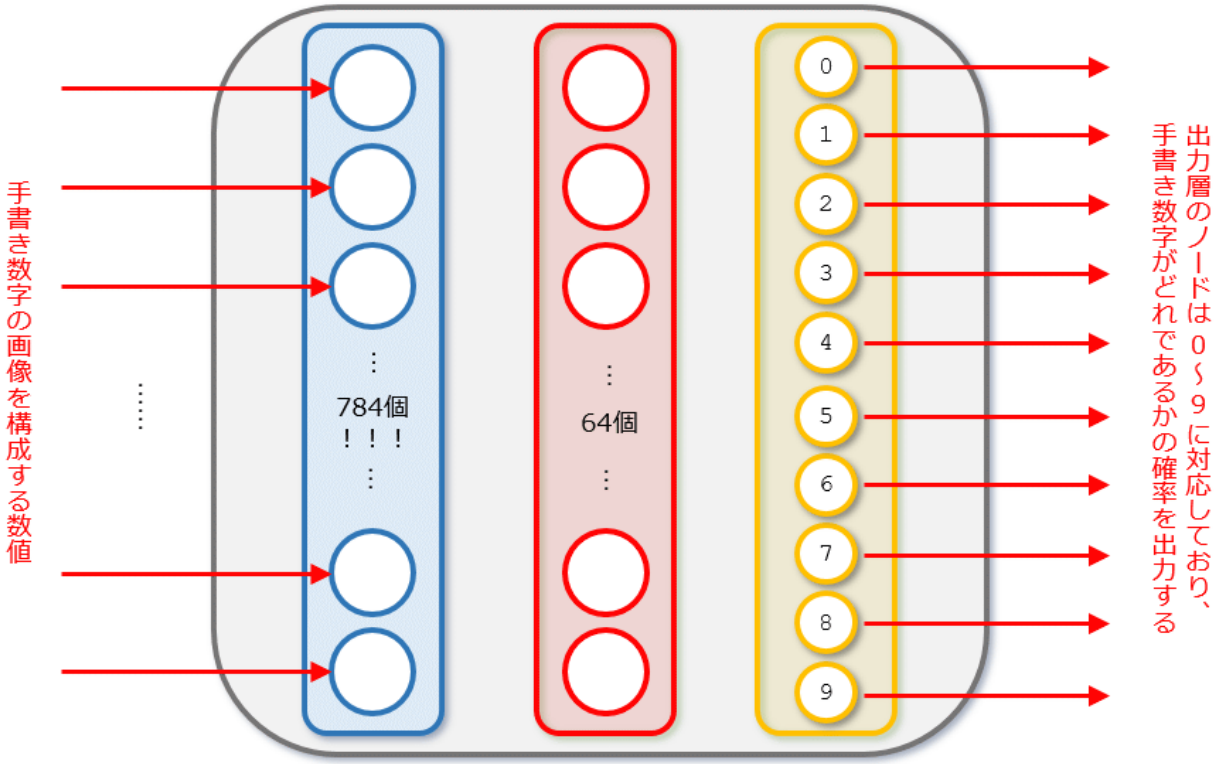
ここまでの話をまとめると次のようになります。

- 手書き数字のサイズは 28×28 ピクセルで、各ピクセルが持つ値は 0 ～ 255
- 訓練データとしては上記の手書き数字が 6 万個、テストデータとしては 1 万個用意されている
- 学習やテストでデータを取り出すときに自動的に 0 ～ 255 の範囲の値が -1.0 ～ 1.0 の範囲の値に変換される

では、1 枚の手書き数字を構成する 28×28（＝ 784）個の値を（複数）受け取り、その数字が何であるかを推測するニューラルネットワーククラスを定義してみましょう。

ニューラルネットワーククラスの定義

「ニューラルネットワーククラスを定義してみましょう」とはいってみましたが、ここでは全結合を行うシンプルなニューラルネットワークを定義します。ここでは入力層は 784 個（28×28 個）のノードを持ち、隠れ層のノード数は 64 個、出力層のノード数は 10 個とします。



ここで定義するニューラルネットワーククラス

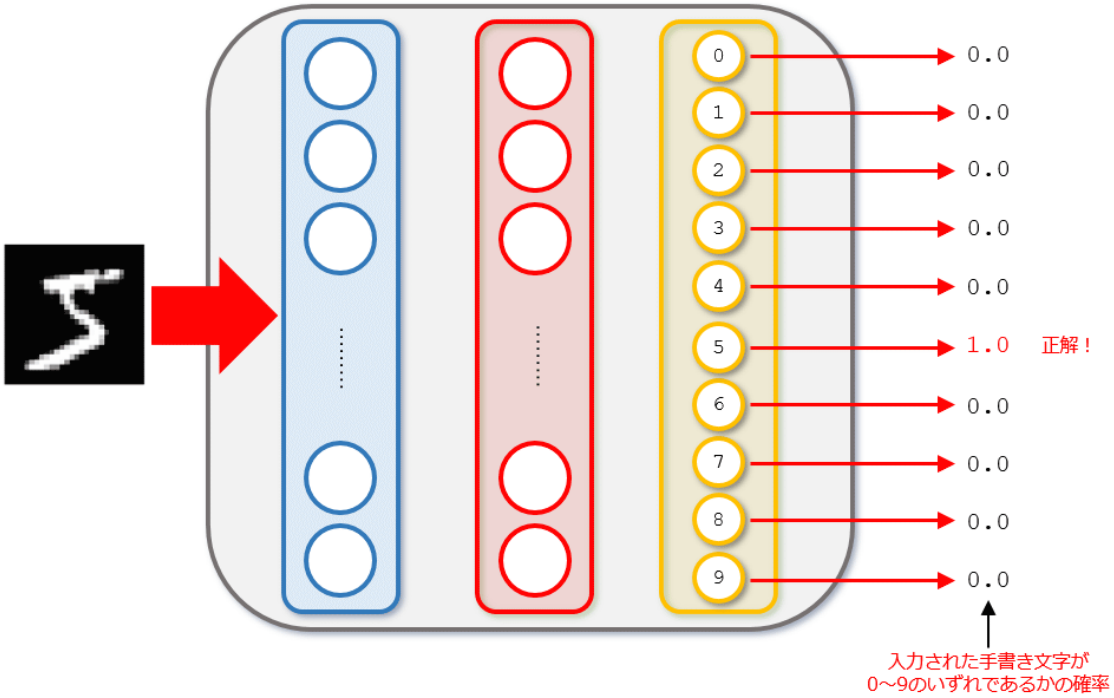
入力の数こそ多いのですが、これはあやめの品種の分類と同様に多クラス分類と呼ばれる処理です。出力が 10 個なのは、推測結果（0 ～ 9）に応じて、対応するノードが一番大きな数値を出力するようにするためです。

推測結果	モデルからの理想的な出力	対応する正解ラベル
0	1, 0, 0, 0, 0, 0, 0, 0, 0, 0	0
1	0, 1, 0, 0, 0, 0, 0, 0, 0, 0	1
2	0, 0, 1, 0, 0, 0, 0, 0, 0, 0	2
3	0, 0, 0, 1, 0, 0, 0, 0, 0, 0	3
4	0, 0, 0, 0, 1, 0, 0, 0, 0, 0	4
5	0, 0, 0, 0, 0, 1, 0, 0, 0, 0	5
6	0, 0, 0, 0, 0, 0, 1, 0, 0, 0	6
7	0, 0, 0, 0, 0, 0, 0, 1, 0, 0	7
8	0, 0, 0, 0, 0, 0, 0, 0, 1, 0	8
9	0, 0, 0, 0, 0, 0, 0, 0, 0, 1	9

推測結果とニューラルネットワークモデルの計算結果と正解ラベル

→目次に戻る

つまり、以下のようなニューラルネットワークモデルを作ることが目標です。ただし、以下で見ていくコードではソフトマックス関数と呼ばれる関数を使っていません。この関数は各ノードからの出力値の合計値が 1.0 となる、つまり、各ノードからの出力値を確率として捉えられるような値を返す活性化関数ですが、これを使っていないので、実際には最大の値を出力したノードが、ニューラルネットワークが推測した数字であると判断をしています（各ノードの出力値の中で最大値のものがやはりソフトマックス関数を通したときに、最大の確率のとなるので、ここではそれでよしとしましょう）。



こんな出力になるのが理想

基本的な形は、前回までに定義したものと同じです。そのため、コードも見慣れたものになっています。

```
class Net(torch.nn.Module):
    def __init__(self, INPUT_FEATURES, HIDDEN, OUTPUT_FEATURES):
        super().__init__()
        self.fc1 = torch.nn.Linear(INPUT_FEATURES, HIDDEN)
        self.fc2 = torch.nn.Linear(HIDDEN, OUTPUT_FEATURES)
        #self.softmax = torch.nn.Softmax(dim=1)
    def forward(self, x):
        x = self.fc1(x)
        x = torch.nn.functional.relu(x)
        x = self.fc2(x)
        #x = self.softmax(x)
        return x
```

全結合型で手書き数字を認識するニューラルネットワーククラス

これまでに見てきたコードと異なるのは、インスタンス生成時にパラメーターに、入力層のノード数、隠れ層のノード数、出力層のノード数を受け取るようにしたことと、活性化関数に ReLU を使用しているところくらいです（__init__ メソッドと forward メソッドの最後では先ほど述べたように torch.nn.Softmax 関数に関連するコードをコメントとして記述していますが、後述する損失関数の内部で同様な処理が行われるので、ここでは省略しています。興味のある方は試してみてください）。

それ以上の説明は不要でしょう。というわけで、次に実際に学習をするコードを見てみます。

学習とテスト

実際に学習を行うコードの基本形もこれまでに見てきたものとあまり変わりません。

まずは上で定義した Net クラスのインスタンスを生成します。

```
INPUT_FEATURES = 28 * 28
HIDDEN = 64
OUTPUT_FEATURES = 10

net = Net(INPUT_FEATURES, HIDDEN, OUTPUT_FEATURES)
```

Net クラスのインスタンスを生成

次に損失関数と最適化アルゴリズムを選択します。今回は、損失関数に多クラス分類でよく使われる torch.nn.CrossEntropyLoss クラスを使っています。PyTorch では CrossEntropyLoss クラスのインスタンスを使って、損失を計算するときには、出力値（10 個の浮動小数点数値を要素とするテンソル）と、その中で値が最大の要素を指すインデックスを渡すことになっています。そして、先ほど見たように、正解ラベルはまさに数字を表すインデックス値になっていたことを思い出してください。実際に、以下のコードではそのようにしているのが分かります（この関数の内部では、実際にはソフトマックス関数の一種が使われるため、Net クラスの定義にはこれを含めていません）。

```
import torch.optim as optim

criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

損失関数と最適化アルゴリズムの選択

PyTorch の CrossEntropyLoss 関数による損失の計算方法などについては、後続の回で取り上げるタイミングがあれば、そこで紹介します。

それでは、最後に学習を行うコードを示します。

```
EPOCHS = 2

for epoch in range(1, EPOCHS + 1):
    running_loss = 0.0
    for count, item in enumerate(trainloader, 1):
        inputs, labels = item
        inputs = inputs.reshape(-1, 28 * 28)

        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if count % 500 == 0:
            print(f'#{epoch}, data: {count * 20}, running_loss: {running_loss
/ 500:1.3f}')
            running_loss = 0.0

print('Finished')
```

学習を行うコード

注意点としては、あやめの品種の分類では全てのデータをニューラルネットワークに入力していましたが、今回はデータの数がそれよりもはるかに多くなっています。そのため、上述したようにエポックごとに訓練データをシャッフルして、そこから 20 個（バッチサイズ）のデータをニューラルネットワークに入力するようになっていることが挙げられます。そのため、for ループが二重になっていることに注意してください。内側の for ループでは、enumerate 関数を使って、trainloader 経由でデータを 20 個取り出すと同時にカウンターの値を増やして、その後の処理で使用しています。このように小さな塊（ミニバッチ）ごとに処理を進めていく方法をミニバッチ学習と呼びます。

また、「inputs.reshape(-1, 28 * 28)」というのは、変数 inputs に取り出した画像データ（その形状は [20, 1, 28, 28] です。つまり、チャネル（1）／画像の高さ（28）／画像の幅（28）を要素とするテンソルを 20 個格納するテンソルです）を、入力層の 784 ノードに合わせて、20×784 というサイズのデータに変換する操作です（「20」ではなく「-1」と指定しているのは、「28×28 ＝ 784 は重要だけど、それ以外はよろしくやってください」と reshape メソッドに丸投げすることを意味しています）。20 はバッチサイズ（画像データの個数）なので、これはつまり画像の高さと幅で構成されていた 2 次元のデータの 1 次元のデータに展開したものを 20 個含むテンソルにする操作ということになります（チャネルは、torchvisions.transforms.ToTensor クラスを使って読み込んだデータを PyTorch のテンソルに変換する段階で自動的に付加されますが、ここでは不要な次元です）。

変数 running_loss は、20 個のデータをニューラルネットワークに入力した結果を基に算出した損失を内部のループを実行するたびに加算していき、ループ変数 count の値が 500 になった時点（つまり、500 回のループ × 20 個＝ 1 万個のデータを処理した時点）でその平均値を出力するために使っています。

その他の処理はこれまでと同様です。

実行結果を以下に示します。

```
EPOCHS = 2

for epoch in range(1, EPOCHS + 1):
    running_loss = 0.0
    for count, item in enumerate(trainloader, 1):
        inputs, labels = item
        inputs = inputs.reshape(-1, 28 * 28)

        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        if count % 500 == 0:
            print(f'#{epoch}, data: {count * 20}, running_loss: {running_loss / 500:1.3f}')
            running_loss = 0.0

print('Finished')
```

```
#1, data: 10000, running_loss: 1.068
#1, data: 20000, running_loss: 0.463
#1, data: 30000, running_loss: 0.394
#1, data: 40000, running_loss: 0.369
#1, data: 50000, running_loss: 0.332
#1, data: 60000, running_loss: 0.334
#2, data: 10000, running_loss: 0.320
#2, data: 20000, running_loss: 0.309
#2, data: 30000, running_loss: 0.308
#2, data: 40000, running_loss: 0.280
#2, data: 50000, running_loss: 0.275
#2, data: 60000, running_loss: 0.271
Finished
```

実行結果

上のコードでは、定数 EPOCHS の値が 2 なので、2 つのエポックを実行して、1 つのエポックのたびに 6 万個のデータが使われて学習が実行されていることが分かります。損失も徐々に小さくなっていますね。

学習が終わったところで、最後に使用した訓練データを基にした推測結果とその正解ラベルとを比べてみましょう。

```
_, predicted = torch.max(outputs, 1)
print(predicted)
print(labels)
```

最後の訓練データを基にした推測結果と正解ラベルを比較

ここで行っている「torch.max(outputs, 1)」というのは、「各行（入力データから得られた推測値である 10 個の数値）の中で最大のものを選び、(最大値 , そのインデックス) というタプルを返す」処理です。この呼び出しでは今述べたように、(最大値 , そのインデックス) というタプルを返しますが、ここで必要なのはインデックスだけです。そのため、代入文の左辺ではアンダースコア「_」を使って、「_, predicted = ……」のようにインデックスだけを受け取っています（実際には「print(_)」のようにすれば、最大値も表示できますが、一般にアンダースコア 1 つだけの変数はプログラマーによる「このデータは不要です」という表明として扱われます）。

これを実行すると次のようになります。

```
_, predicted = torch.max(outputs, 1)
print(predicted)
print(labels)
```

```
tensor([5, 4, 1, 3, 0, 9, 1, 8, 3, 0, 3, 1, 6, 2, 3, 1, 7, 5, 8, 7])
tensor([5, 4, 1, 3, 0, 9, 1, 8, 3, 3, 3, 1, 6, 2, 3, 1, 2, 5, 8, 7])
```

実行結果

おおむねよさそうな結果になっているのが確認できました。

最後にテストデータを使って、このニューラルネットワークモデルのテストを行ってみましょう（機械学習では、学習と評価を繰り返して、学習に関連するハイパーパラメーターの調整などを行いながら、ニューラルネットワークモデルの精度を高めて、最後にテストを行うのが常道ですが、ここでは省略します）。

```
correct = 0
total = 0

with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        inputs = inputs.reshape(-1, 28 * 28)
        outputs = net(inputs)
        _, predicted = torch.max(outputs, 1)
        total += len(outputs)
        correct += (predicted == labels).sum().item()

print(f'correct: {correct}, accuracy: {correct} / {total} = {correct / total}')
```

ニューラルネットワークモデルをテストする

2 つの変数 correct と total は、このニューラルネットワークモデルのテストで正解となった推測結果の数と処理をしたデータの総数を保存しておくために使います。その後にある、「with torch.no_grad():」という with 文は、そのブロックで行う処理では勾配の計算を無効にするものです（ここでは学習を行わずに、テストをしているので、このようにすることで無駄な処理を行わずに済むようになります）。

with 文のブロック内では for ループを使って、testloader から 20 個ずつ取り出したデータをニューラルネットワークモデルに入力しています。そして、その推測結果で正しいものの数を上述した変数 correct に、処理したデータの数（常に 20 ではありますが）を変数 total に加算していき、最後に正解の数と精度を表示するようにしました。

実行結果を以下に示します。

```
correct = 0
total = 0

with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        inputs = inputs.reshape(-1, 28 * 28)
        outputs = net(inputs)
        _, predicted = torch.max(outputs, 1)
        total += len(outputs)
        correct += (predicted == labels).sum().item()

print(f'correct: {correct}, accuracy: {correct} / {total} = {correct / total}')
```

```
correct: 9278, accuracy: 9278 / 10000 = 0.9278
```

実行結果

ここでは全結合型のニューラルネットワークを作成しましたが、それでも 92%程度の精度が出ていることが分かりました。とはいえ、実は画像認識を行うニューラルネットワークではもっとよい方法があることが知られています。次回以降では、その方法について取り上げていきましょう。

CNN なんて怖くない！ その基本を見てみよう

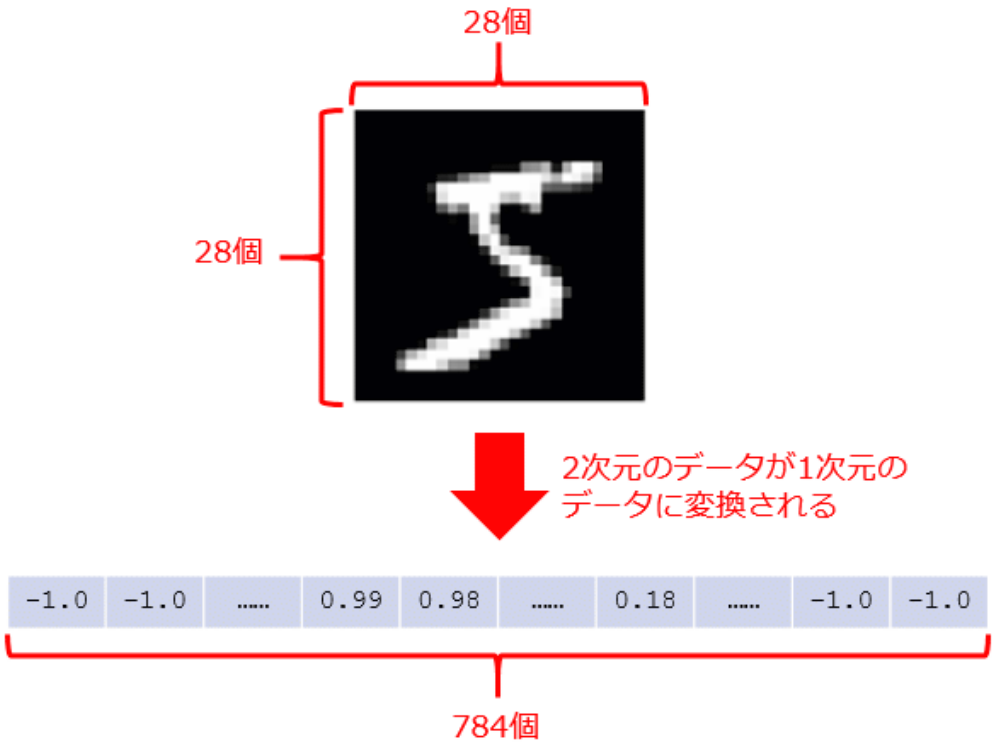
画像認識などでよく使われる CNN（畳み込みニューラルネットワーク）ではどんなことが行われているのでしょう。図を見ながら、CNN の基本を理解しましょう。

（2020 年 05 月 29 日）

前回は、全結合型のニューラルネットワークを使って MNIST の手書き数字を認識してみました。今回は、CNN の基本について見た後で、PyTorch を使って CNN で手書き数字を認識するコードを紹介します。

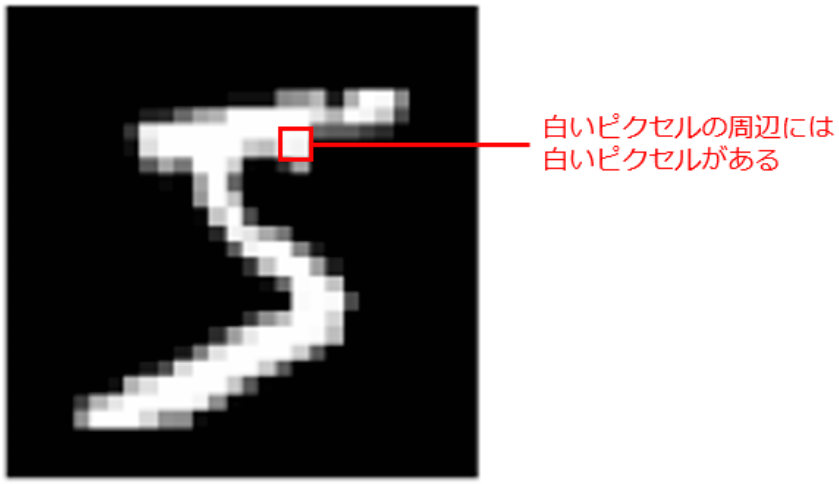
CNN（Convolutional Neural Network）の概要

前回は 2 次元の画像（手書き数字）を全結合型のニューラルネットワークを使って認識してみました。しかし、これには問題があります。入力層のノード数が 28×28 ＝ 784 個だったことを思い出してください。これが意味するのは、2 次元の構造を持っていたデータが 1 次元のデータに展開されてしまったということです。



全結合型のニューラルネットワークでは 2 次元データを 1 次元のデータに展開して入力としていた

2 次元の画像データでは、あるピクセルとその周辺のピクセルとの関係は重要な情報です。MNIST はグレースケール（白黒）の画像を集めたデータセットで、背景が黒、数字が白として表現されていました（ここでは単純化して白と黒としていますが、実際には 0 ～ 255 の値で、白が強くなるほど数値は大きくなります）。となると、どこかのピクセルが白であれば（数字を表すピクセルのいずれかであれば）、その周辺も数字の一部である＝隣接するピクセルが白である可能性が高くなります（1 点のピクセルだけが白というのは、数字においてはあまりないでしょう）。もちろん、数字と背景の境界辺りでは、周りに黒いピクセルもあるでしょうし、実際にはそのような白いピクセルと黒いピクセルとの関係が数字の「特徴」を表すことになります。



白いピクセルの周辺には白いピクセルがある可能性が高い

全結合型のニューラルネットワークでは、こうした情報を無視して、全てを 1 次元のデータとして、学習したり、テストしたりしていました。

そうではなく、これらの情報を加味して処理をするニューラルネットワークを作成するにはどうしたらよいでしょう。これを実現するためによく使われているのが、今回紹介する CNN（Convolutional Neural Network、畳み込みニューラルネットワーク）と呼ばれるものです。

CNN では 2 次元のデータを小さな区分に分割して、それらと何らかの特徴を表すデータとを比較しながら、元のデータがどんな特徴を含んだものであるかを調べていきます。この「何らかの特徴を表すデータ」のことを「カーネル」「ウィンドウ」「フィルター」などと呼ぶことがあります。ただし、本稿では PyTorch のドキュメントで一般に使われている「カーネル」という語を使うことにします。

話を少し単純にするために、ここでは 5×5 ピクセルの × と○を画像データとして考えましょう。ここで -1.0 は黒を、1.0 は白を表すものとします。

1.0	-1.0	-1.0	-1.0	1.0
-1.0	1.0	-1.0	1.0	-1.0
-1.0	-1.0	1.0	-1.0	-1.0
-1.0	1.0	-1.0	1.0	-1.0
1.0	-1.0	-1.0	-1.0	1.0

-1.0	1.0	1.0	1.0	-1.0
1.0	-1.0	-1.0	-1.0	1.0
1.0	-1.0	-1.0	-1.0	1.0
1.0	-1.0	-1.0	-1.0	1.0
-1.0	1.0	1.0	1.0	-1.0

5×5 ピクセルに描かれた×と○

ここで × のデータにはどんな特徴があるでしょう。

- 右下がりの直線（＼）がある
- 左下がりの直線（／）がある
- それらが交差する（×）

○のデータは、それを八角形のようなものだと近似すれば、次のような特徴を持つといえるかもしれません。

- 右下がりの直線（＼）がある
- 左下がりの直線（／）がある
- 横線（ー）がある
- 縦線（｜）がある

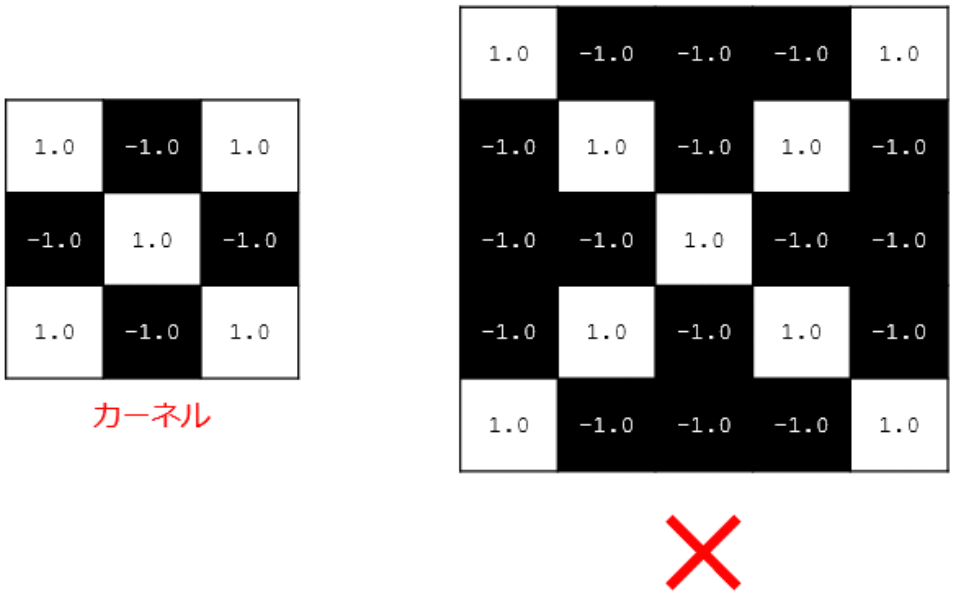
カーネルはこのような特徴を表すもので、画像データ（を分割した小さな部分）にどんな特徴が含まれているかを調べるために使われます。

ここで、（実際には）カーネルが保持するデータ（特徴）は人間が指定する必要がない点には注意してください。CNN では「ある画像がどんな特徴を持つかをニューラルネットワークが学習してくれる」というわけです。実際には、CNN からの出力は（後述するプーリングや活性化関数などによる処理を経て）全結合を行うネットワークに接続され、そこで画像が何であるかの推測が行われるのですが、それについてはまた後で見ることにしましょう。その前に、CNN の「C」つまり「Convolution」「畳み込み」がどんな処理なのかを見てみます。

畳み込み

CNN は「畳み込みニューラルネットワーク」のことですが、ここでは実際にどんな処理（畳み込み）が行われるのでしょうか。先ほどは、画像データを小さな区分に分割して、それとカーネルを比較すると述べました。これをもう少し詳しく見てみます。

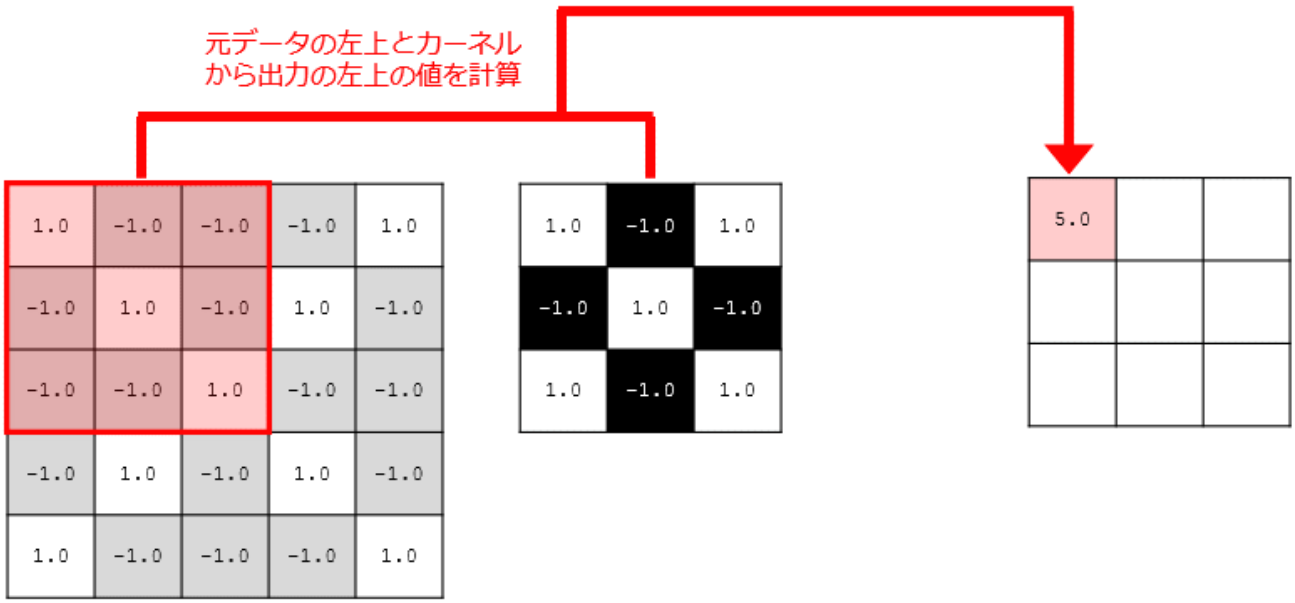
カーネルは 3×3、5×5 などの小さな 2 次元データと考えてください。ここでは 3×3 というサイズの 2 次元配列（テンソル）とし、これと先ほどの × のデータを例として、どんなことが行われるかを紹介しましょう（カーネルの値は学習により得られたものだと考えてください）。



カーネルと画像データ（×のデータ）

カーネルは見ての通り、画像データに交差している箇所があるかどうか調べるために、× のデータの中央部と同じようなデータになっています。

まず画像データにカーネルを適用するときには、その左上から右下に向かって、カーネルのサイズと同じサイズのデータを取り出して、それとカーネルを使って行列の演算を行っていきます。そして、その結果を CNN からの出力に左上から並べていきます。



畳み込み

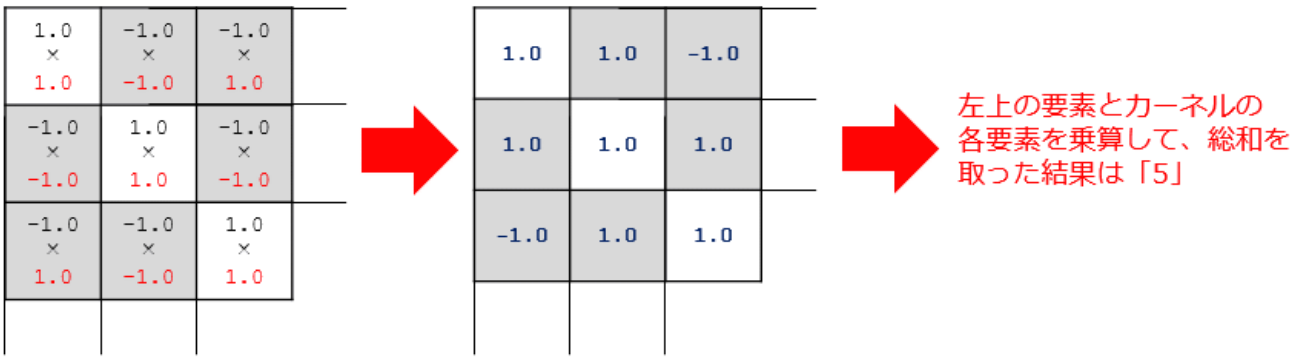
このような画像データの区画とカーネルとの演算を行った結果を出力としていく処理が「畳み込み」です（1 つの新しいデータの中に、複数のデータが持つ情報を含めていくこと）。この出力を「特徴マップ」と呼ぶこともあります。これは畳み込まれたデータが、元の画像データの特徴を表しているからです（後述）。

また、上の図を見ると分かりますが、畳み込みにより、出力データのサイズが元の画像データよりも小さくなっていることに注意してください。5×5 の画像データと 3×3 のカーネルを使って計算をすると、計算結果（特徴マップ）のサイズは 3×3 となります。このようなデータサイズの減少を避けたり、画像データの端にあるデータを使った畳み込み処理の回数を増やしたりするなどの目的で画像データの上下左右に「パディング」と呼ばれる要素（値は一般に「0」のことが多い）を付加することもあります。

0.0	0.0	0.0	0.0	0.0	0.0	0.0
0.0	1.0	-1.0	-1.0	-1.0	1.0	0.0
0.0	-1.0	1.0	-1.0	1.0	-1.0	0.0
0.0	-1.0	-1.0	1.0	-1.0	-1.0	0.0
0.0	-1.0	1.0	-1.0	1.0	-1.0	0.0
0.0	1.0	-1.0	-1.0	-1.0	1.0	0.0
0.0	0.0	0.0	0.0	0.0	0.0	0.0

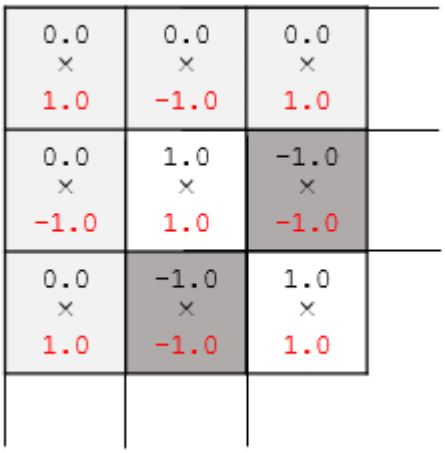
元データにパディングを付加したもの

畳み込みの際には、カーネルの各要素と、画像データからピックアップした区画において、同じ位置にある要素同士を乗算して、それらの和を出力値とします。例えば、画像データの左上の区画とカーネルとの処理であれば、次のようになります（ここでは、カーネルの要素の値は赤字で表示しています。また見やすくなるよう、少し背景色を変更しています）。



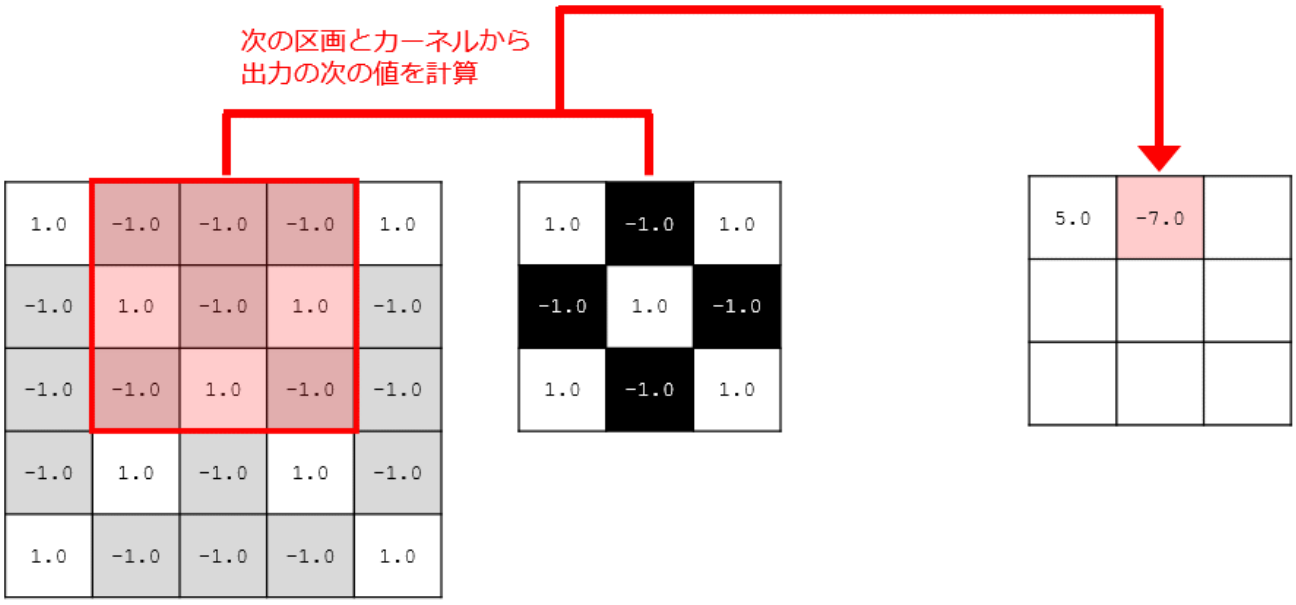
画像データの左上の区画とカーネルの各要素を乗算して、その総和が出力の左上の値となる

パディングをしていれば、次のようになります（総和を取ると「4」になります）。



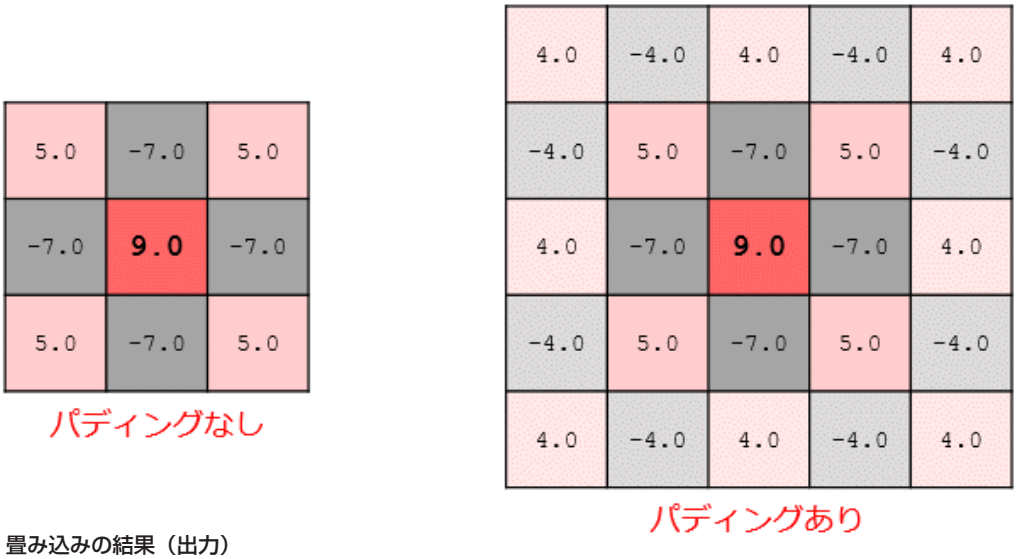
画像データにパディングをしている場合の左上の要素とカーネルの要素の計算

左上の要素とカーネルとの処理が終わったら、カーネルを右に 1 つズラして、同じ処理を行います。この適用結果が CNN の左上から 2 番目のデータとなります（カーネルをどれだけズラすかを「ストライド」と呼びます。ここでは 1 つズラす＝ストライドは 1、としていますが実際にはその幅は指定できるのが一般的です）。



次の区画とカーネルの計算

このようにして左上から右下に向かって、画像データにカーネルを適用すると最終的には次のような結果（特徴マップ）が得られます（パディングを入れた場合と入れない場合を計算しました）。以下では負の値は背景色をグレーとして、正の値は背景色を赤として表示しました（値が大きいほど濃い色）



畳み込みの結果（出力）

パディングがない場合の出力が、パディングがある場合の出力の中央にあることに注意してください。

ところで、先ほど「畳み込まれたデータが、元の画像データの特徴を表している」と述べましたが、これはどういう意味でしょうか。ここではカーネルは × という記号の交差点を探すためのデータでした。そして、画像データでは、交差する箇所は中央にありました。そこで特徴マップを見ると、中央のデータが「9」と最大の値になっています。

これは「交差する」という特徴が、画像データでは中央にあることを示していると考えられます。出力の値がマイナスになっている箇所（パディングなしの計算結果の上部中央など）は、そうした特徴が見られないということです。それ以外の場所（同じくパディングなしの計算結果の左上や右下など）はある程度はそうした特徴も見えるが、完全にそうともいいきれないような場所です。実際、画像データの四隅には斜めの直線はあるけれど、交差はしていないので、交差という特徴の一部は示していると考えられます。このように値の大きさで、カーネルが表す特徴が画像のどこにあるかを示すのが特徴マップです。

このように、カーネルという小サイズの特徴を検索する道具を用いることで、元となった画像データから何らかの特徴を探し出すことができます。このときに注目したいのは、「現在探している特徴が元データのどこにあっても、それを見つけ出せる」点です。

ここで使っているサンプルデータでは、5×5 のデータで対角線の値を 1.0 とすることにより × を表していました。しかし、これが上下あるいは左右にズレても、カーネルを使ってデータの全体をくまなく見ていくことで、交差する箇所を見つけ出せます。これは MNIST のような手書き数字にもいえます。すなわち、カーネルを使って 28×28 のサイズの全ピクセルをくまなく走査していくことで、数字を含んだ画像データから特徴を拾い出せるということです。

ここではカーネルは交差を探すためのものが 1 つだけでしたが、右下がりの直線、左下がりの直線、横線、縦線などを表すカーネルがあれば、それらを組み合わせることで、×と○を含んだ画像を CNN に入力すると、それらの特徴を含んだ特徴マップが得られることは想像できるでしょう。そして、CNN は多数の入力データを基にカーネルが特徴をうまく示すものとなるように、その重みやバイアスを学習していくというわけです。なお、これらのカーネルの数のことを「チャネル」と呼ぶことがあります。

PyTorch では今述べたような処理を行うためのクラスとして Conv2d クラスが提供されています。

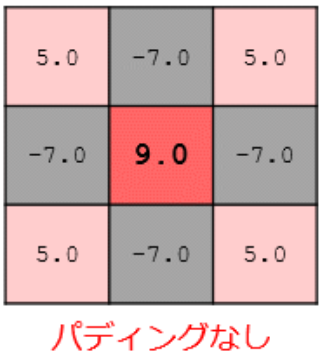
そして、CNN においてはもう 1 つ「プーリング」と呼ばれる重要な処理があります。

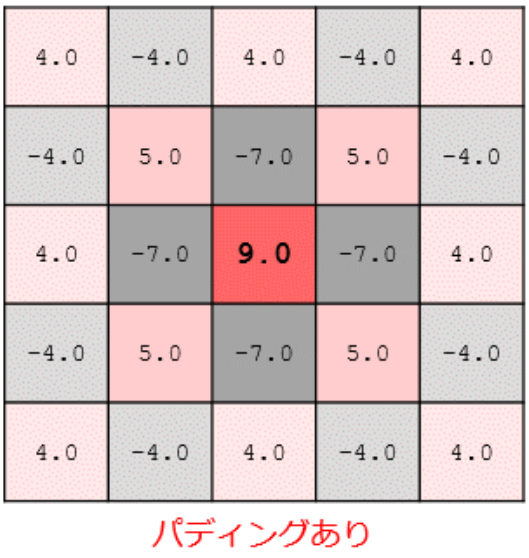
プーリング

「プーリング」とは、畳み込みによって得た特徴（特徴マップ）から重要な要素は残しながら、データ量を削減する処理です。通常は入力（特徴マップ）を小さなサイズの区画（2×2、3×3 など。これもやはりウィンドウとかカーネルと呼びます）に分けて、その区画内で特徴的な値（最大値、平均値など）を取り出して、それをプーリングの出力とします。

ここでは、上で得た特徴マップに対して、2×2 のサイズでプーリングを行ってみましょう。多くの場合は最大値を取り出すので、ここでもそうしてみます。

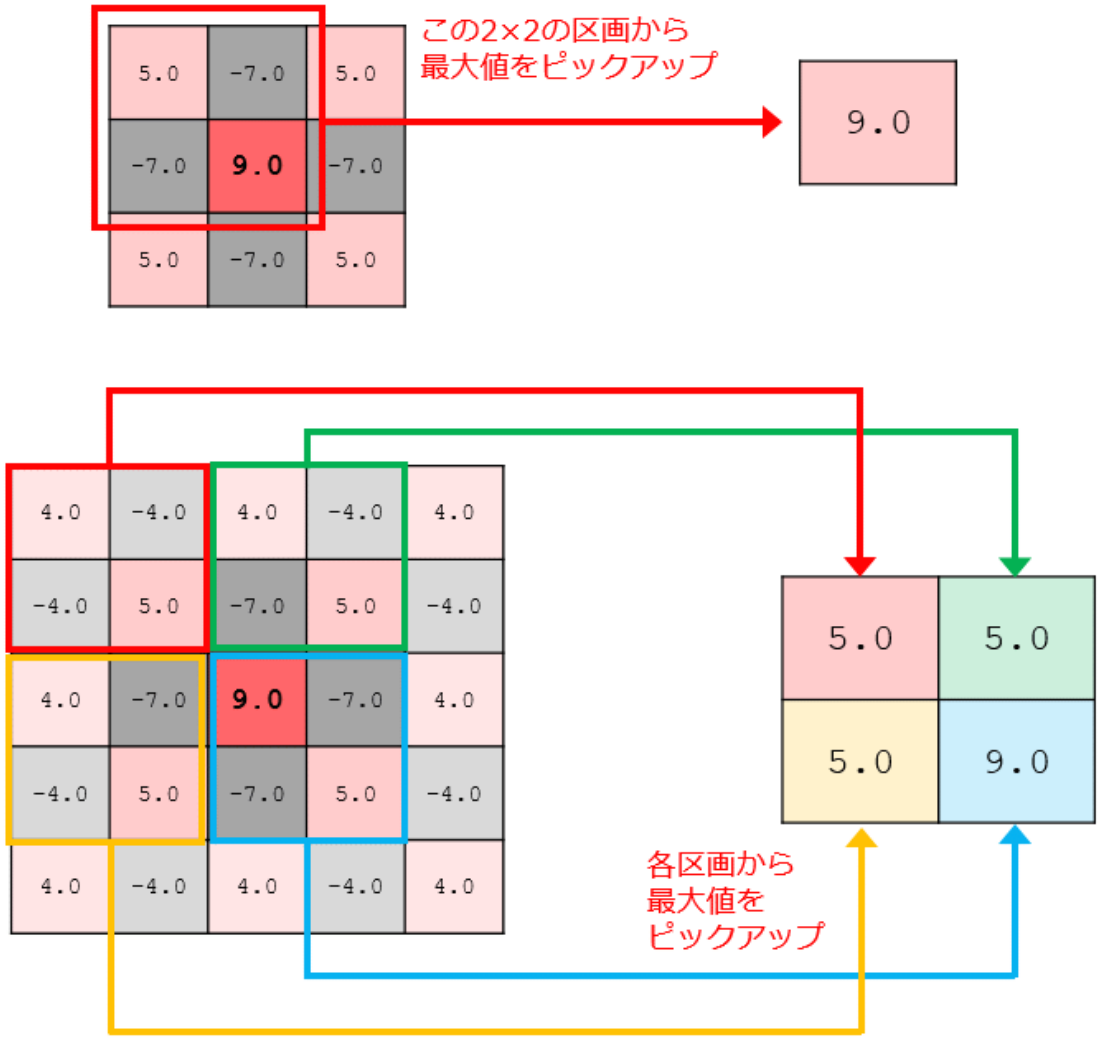
特徴マップは次のようなものでした。





特徴マップ（再掲）

これに対して、左上から右下に向かって、先ほどと同様な順序で値を取り出していくと次のようになります。



プーリング処理の結果

元の画像データにパディングをしていない方では、特徴マップは 3×3 というサイズだったので、左上の 2×2 の要素の中で最大値である「9」という要素だけが取り出されました。パディングをしている方では 2×2 のサイズで最大値が取り出されました。いずれにしても交差していることが強く強調されるデータ（9）がうまく取り出されると同時に、データ量が削減されました。

プーリングにも、畳み込みと同様な特徴があります。それは入力（この場合は特徴マップ）の中で多少のズレがあっても、もともとの特徴を示すデータをうまく拾い上げられる点です。画像データはピクセル単位での処理をするので、元のデータや重み、バイアスなどによって、特徴マップのどこに特徴といえる値が出てくるかはそのときどきで変わるかもしれません。そんなときでも、プーリングを行うことで必要なデータをうまく取り出せるのがプーリングのメリットといえます。

PyTorch ではこの処理を行うクラスとして MaxPool2d クラスなどが提供されています。

畳み込みは元データが持つ微細な特徴を見つけ出す処理、プーリングは畳み込みによって見つかった特徴マップの全体像を大まかな形で表現する処理（大きな特徴だけをより際立たせる処理）と考えることもできるでしょう。

畳み込みとプーリング（とその間に挟み込む活性化関数）という組み合わせを何層かに重ねることで、入力層に近いところでは今述べた微細な特徴を、入力層から遠い層では全体的な（抽象的な）特徴を表現できるようになります。そうして得られたものを全結合により推測を行う層（全結合層）へと渡して、最終的に分類を行うというのが CNN による画像認識の手順となります。

畳み込み層とプーリング層と全結合層で構成されるニューラルネットワーク

PyTorch で CNN を実装する

ここまでは CNN でどんな処理が行われるかを見てきましたが、実際にこれを使って MNIST の手書き数字を推測するコードを最後に見ておきましょう。といっても、コードはこれまでのものとほとんど変わりません。

まず MNIST データベースからデータセットを読み込むコードです。これについて前回と同じコードです。torchvision.datasets.MNIST クラスと torch.utils.data.DataLoader クラスを使って、データセットを読み込んで、それを反復するデータローダーを用意しています。

```
import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,), (0.5,))])

BATCH_SIZE = 20

trainset = torchvision.datasets.MNIST(root='./data', train=True,
                                       transform=transform, download=True)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=BATCH_SIZE,
shuffle=True)

testset = torchvision.datasets.MNIST(root='./data', train=False,
                                      transform=transform, download=True)
testloader = torch.utils.data.DataLoader(testset, batch_size=BATCH_SIZE,
shuffle=False)
```

MNIST データベースからデータセットを読み込むコード

これまでに見てきたような畳み込みとプーリングを行うクラスの定義を以下に示します。

```
class Net(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(1, 6, 5)
        self.pool = torch.nn.MaxPool2d(2, 2)
        self.conv2 = torch.nn.Conv2d(6, 16, 5)
        self.fc1 = torch.nn.Linear(16 * 16, 64)
        self.fc2 = torch.nn.Linear(64, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = torch.nn.functional.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.pool(x)
        x = x.reshape(-1, 16 * 16)
        x = self.fc1(x)
        x = torch.nn.functional.relu(x)
        x = self.fc2(x)
        return x
```

CNN を利用して MNIST の手書き数字を認識する Net クラス

既に述べたように、PyTorch には 2 次元データの畳み込みを行うためのクラスとして Conv2d クラスが、最大値のプーリングを行うクラスとして MaxPool2d クラスが用意されています。ここでは、それらのインスタンスを作成しています。Conv2d クラスのインスタンスは 2 つ作成して、インスタンス変数 conv1 と conv2 に代入しています。forward メソッドを見ると分かりますが、畳み込みは 2 回行うということです。

インスタンス変数 conv1 のインスタンス生成では、「Conv2d(1, 6, 5)」のように引数を指定しています。第 1 引数の「1」は「入力チャネルの数」です。MNIST は RGB 値のようなカラー画像ではなく、各ピクセルが 0 ～ 255（の値を -1 ～ 1 の範囲の浮動小数点数に変換したもの）だけのデータなので、ここでは 1 を指定しています。第 2 引数は「出力チャネルの数」ですが、これが実質的にはカーネルの数を表します。ここでは 6 個のカーネルを作成するということです。第 3 引数は「カーネルのサイズ」です。ここでは「5×5」のサイズのカーネルを作成するということになります。

インスタンス変数 conv2 の生成では、「Conv2d(6, 16, 5)」のように引数を指定しています。第 2 引数と第 3 引数の指定は上と同様なので説明は不要でしょう。しかし、第 1 引数の「6」については少し説明が必要です。インスタンス変数 conv1 では出力チャネルの数を「6」としていました。これが活性化関数とプーリングによる処理を経て、インスタンス変数 conv2 へと渡されます。そのため、データの入力元となるインスタンス変数 conv1 の出力チャネルの数と、データを受け取るインスタンス変数 conv2 の入力チャネルの数を一致させておく必要があります。そのため、ここでは「6」を指定しています（なお、これらの引数の値は筆者が適当に定めたもので、特に理由はありません。もっとよい値があるかもしれません）。

MaxPool2d クラスのインスタンスは 1 つだけ作成して、それをインスタンス変数 pool に代入しています。2 回の畳み込みの（結果を活性化関数で処理した）結果は、このインスタンスで処理してプーリングを行っています。引数は「MaxPool2d(2, 2)」となっているので、2×2 のサイズでプーリングを行うことを意味しています。

最後に、それらを、全結合を行うインスタンス変数 fc1 と fc2 で処理するだけです（インスタンス変数 fc1 のノード数は、16×16 = 256 個となっているのは、畳み込みとプーリングによって得た、全結合層への入力の数を筆者が確認して、その数を指定しました）。

最後に学習とテストを行うコードを以下に示します。

```
import torch.optim as optim

net = Net()

criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

EPOCHS = 2

for epoch in range(1, EPOCHS + 1):
    running_loss = 0.0
    for count, item in enumerate(trainloader, 1):
        inputs, labels = item

        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
```

```
            loss.backward()
            optimizer.step()

            running_loss += loss.item()
            if count % 500 == 0:
                print(f'#{epoch}, data: {count * 20}, running_loss: {running_loss
/ 500:1.3f}')
                running_loss = 0.0

print('Finished')

correct = 0
total = 0

with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        outputs = net(inputs)
        _, predicted = torch.max(outputs, 1)
        total += len(outputs)
        correct += (predicted == labels).sum().item()

print(f'correct: {correct}, accuracy: {correct} / {total} = {correct /
total}')
```

学習を行うコード

これも基本的には、前回と同じコードなので説明は省略します。

これらのコードを実行すると、結果は次のようになります。

```
            running_loss = 0.0

print('Finished')

correct = 0
total = 0

with torch.no_grad():
    for data in testloader:
        inputs, labels = data
        outputs = net(inputs)
        _, predicted = torch.max(outputs, 1)
        total += len(outputs)
        correct += (predicted == labels).sum().item()

print(f'correct: {correct}, accuracy: {correct} / {total} = {correct / total}')
```

```
#1, data: 10000, running_loss: 1.898
#1, data: 20000, running_loss: 0.428
#1, data: 30000, running_loss: 0.241
#1, data: 40000, running_loss: 0.173
#1, data: 50000, running_loss: 0.135
#1, data: 60000, running_loss: 0.131
#2, data: 10000, running_loss: 0.110
#2, data: 20000, running_loss: 0.104
#2, data: 30000, running_loss: 0.099
#2, data: 40000, running_loss: 0.092
#2, data: 50000, running_loss: 0.089
#2, data: 60000, running_loss: 0.081
Finished
correct: 9780, accuracy: 9780 / 10000 = 0.978
```

実行結果

前回の全結合型のニューラルネットワークでは 92%程度の精度でしたが、今回はそれよりも高い精度で認識できていることが分かります。

今回は CNN による画像認識の基礎知識とそれを実際に行うコードを見ました。コードについては少し駆け足になってしまいましたが、次回は手を動かしながら、実際のコードでどんなことが行われているかを見ていくことにします。

CNN なんて怖くない！ コードでその動作を確認しよう

CNN による画像認識ではどんなふうに処理が進むのかを、実際に手を動かしながら確認していきましょう。

（2020 年 06 月 05 日）

前回は CNN で画像を認識する際の基本的な仕組みを理論的な面から説明しました。今回は、PyTorch を使って、実際にコードを動かしながら、その動作を確認していきましょう。

横線と縦線のどちらであるかを推測する CNN

前回は × と○を例として、カーネル（フィルター、ウィンドウ）を使い、画像の特徴がどこに現れているかを特徴マップに畳み込み、それをプーリングによって強調するという話をしました。

今回はさらにシンプルに横棒と縦棒を表すデータを例として、畳み込み層（+プーリング層）と全結合層で行われる処理（とは、PyTorch のニューラルネットワーククラスの forward メソッドで行われる処理です）について実際に見ていきます。

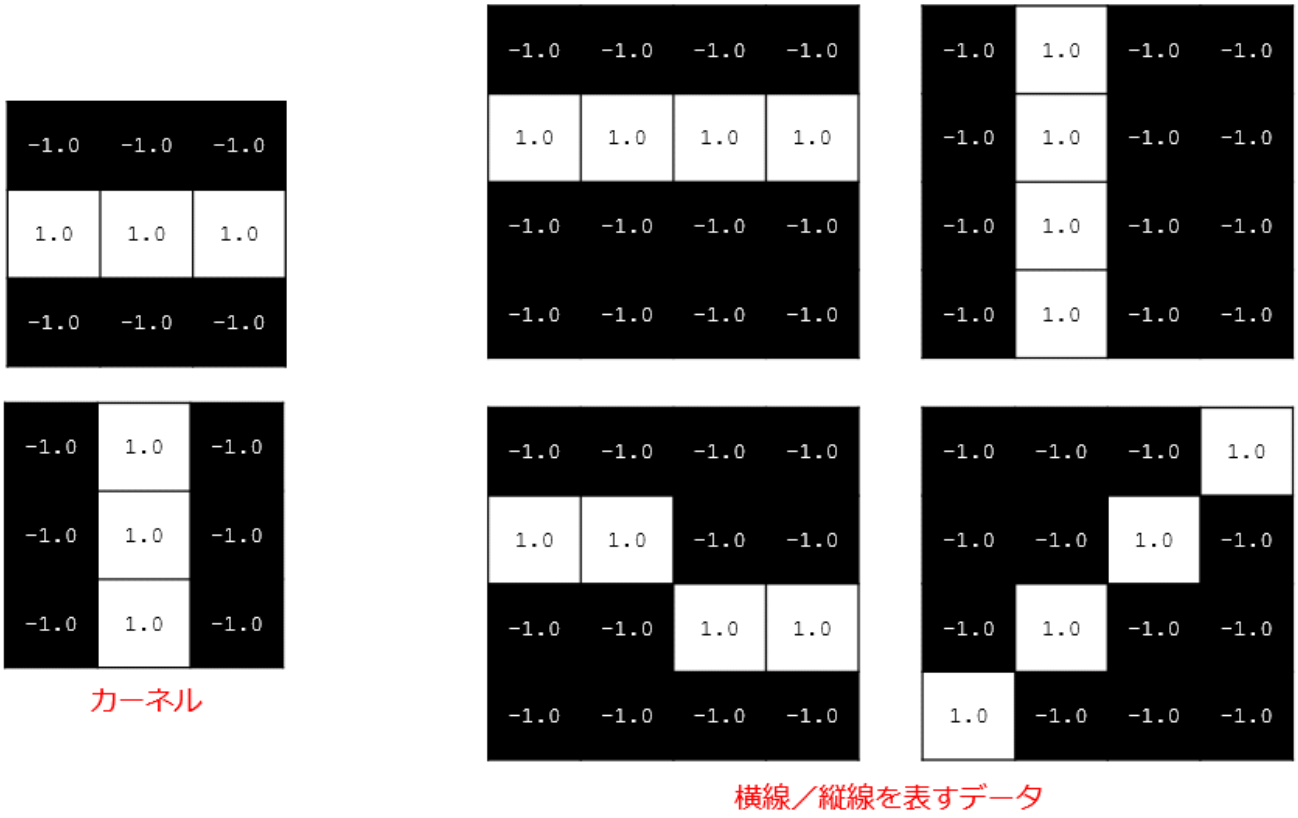
前回に紹介した MNIST の手書き数字を認識するニューラルネットワークは次のようなものでした。

```
class Net(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(1, 6, 5)
        self.pool = torch.nn.MaxPool2d(2, 2)
        self.conv2 = torch.nn.Conv2d(6, 16, 5)
        self.fc1 = torch.nn.Linear(16 * 16, 64)
        self.fc2 = torch.nn.Linear(64, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = torch.nn.functional.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.pool(x)
        x = x.reshape(-1, 16 * 16)
        x = self.fc1(x)
        x = torch.nn.functional.relu(x)
        x = self.fc2(x)
        return x
```

MNIST の手書き数字を認識する Net クラス

今回のコードは、細かなところに関しては上記のコードと異なりますが、基本的には上と同じコードを手で動かしながら、その動作を確認します。「畳み込み→活性化関数→プーリング→畳み込み→活性化関数→プーリング→全結合」という流れを取りあえずは把握しておきましょう。

ここでは以下のようなデータを用意しました。



今回使用するデータ

特徴を抜き出す対象の画像（に相当するもの）は 4×4 のサイズのものが 4 種類、カーネルは 3×3 のサイズのものが 2 種類とします（この後、2 つ目の畳み込み層も登場するので、それ用のカーネルもありますが、それについては後述します）。

横棒を表すデータが 2 つ、縦棒を表すデータが 1 つ、どちらともいえないデータが 1 つあります。カーネルは横線を表すものと、縦線を表すものになっているのはすぐに分かるでしょう。2 つのカーネルを使って 4 つのデータを調査すると、実際にどのような特徴がピックアップされるのかを試してみることにしましょう。なお、今回のコードはこのリンクで公開しています。

データと各層の準備

まず畳み込みとプーリングを行うための Conv2d クラスと MaxPool2d クラスのインスタンスを用意します。

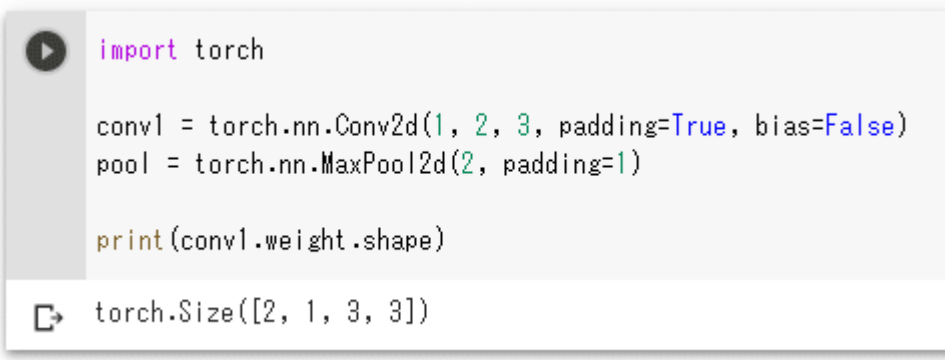
```
import torch

conv1 = torch.nn.Conv2d(1, 2, 3, padding=True, bias=False)
pool = torch.nn.MaxPool2d(2, padding=1)

print(conv1.weight.shape)
```

畳み込み（1 回目）とプーリングを行うオブジェクトの用意

1 回目の畳み込みを行うオブジェクトである conv1 の生成では、入力チャネル数に 1、出力チャネル数（カーネル数）に 2、カーネルのサイズに 3 を指定しています。入力チャネルの数が 1 なのは、前回と同様、これが RGB 値のように複数のチャネルを持つものではないからです。また、データが小さいのでここではパディングを付加するようにしています。話を簡単にするためにバイアスもここでは使わないことにしました。

プーリングを行うオブジェクトでは、カーネルのサイズは 2×2 で、畳み込みと同様にパディングをすることにしました。

なお、カーネルは実際には畳み込みを行うオブジェクトの重みとなるので、上のコードではその形状（各次元の要素数）を確認しています。これを実行した結果が以下です。



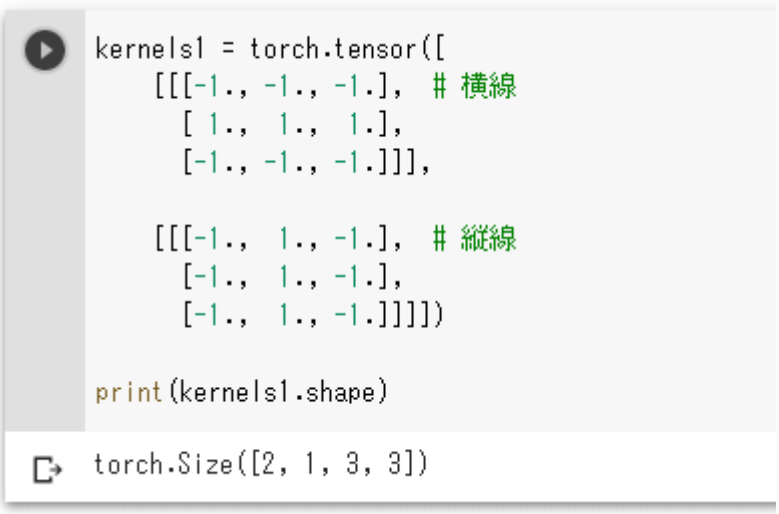
実行結果

この結果が意味するのは、「チャネル数が 1 で、サイズが 3×3 のカーネル（重み）が 2 つある」ということです。カーネルを表すデータはこれに合わせて作成します。

```
kernels1 = torch.tensor([
    [[[-1., -1., -1.],  # 横線
      [ 1.,  1.,  1.],
      [-1., -1., -1.]]],

    [[[-1.,  1., -1.],  # 縦線
      [-1.,  1., -1.],
      [-1.,  1., -1.]]]])

print(kernels1.shape)
```

カーネルを表すデータ

1 つ目のカーネルは第 1 行（中央の行）の要素が全て「1」です。これで横線という特徴を表しています。対して、2 つ目のカーネルでは第 1 列（中央の列）がそうなっています（縦線）。最後にその形状を確認しています。実際の実行結果は次の通りです。



実行結果

conv1 の重みと同じ形状になっているのを確認したところで、今回はこのカーネルを conv1 の重みとしてしまいましょう（学習させるのが目的ではなく、カーネルを使って 4 つのデータの畳み込みとプーリングを試すのが目的なので）。これには次のように、conv1 オブジェクトの weight.data 属性にカーネルを代入するのが簡単です。

```
conv1.weight.data = kernels1
```

カーネルを conv1 の重みとする

これでカーネルの準備ができました。次に画像に相当する 4 つのデータを用意しましょう。以下のコードでは、-1 を背景として、そこに 1 で横線、縦線、どちらでもない線を表したものを、変数 sample_data に代入しています。

```
sample_data = torch.tensor(
    [[[-1., -1., -1., -1.],  # 横線
      [ 1.,  1.,  1.,  1.],
      [-1., -1., -1., -1.],
      [-1., -1., -1., -1.]],
     [[-1.,  1., -1., -1.],  # 縦線
      [-1.,  1., -1., -1.],
      [-1.,  1., -1., -1.],
      [-1.,  1., -1., -1.]],
     [[-1., -1., -1., -1.],  # 横線
      [ 1.,  1., -1., -1.],
      [-1., -1.,  1.,  1.],
      [-1., -1., -1., -1.]],
     [[-1., -1., -1.,  1.],  # 左下がりの直線
      [-1., -1.,  1., -1.],
      [-1.,  1., -1., -1.],
      [ 1., -1., -1., -1.]]])

print(sample_data.shape)
```

横線と縦線とどちらでもない画像に相当するデータ

4 つ目のデータは左下がりの直線を表すもので、横線とも縦線ともいえるようないえないようなものになっています。これらをカーネルを使って調査するとどうなるかも後で見てみましょう。

最後にこのデータの形状も確認しています。実行結果は次の通りです。

```
sample_data = torch.tensor(
    [[[-1., -1., -1., -1.],  # 横線
      [ 1.,  1.,  1.,  1.],
      [-1., -1., -1., -1.],
      [-1., -1., -1., -1.]],
     [[-1.,  1., -1., -1.],  # 縦線
      [-1.,  1., -1., -1.],
      [-1.,  1., -1., -1.],
      [-1.,  1., -1., -1.]],
     [[-1., -1., -1., -1.],  # 横線
      [ 1.,  1., -1., -1.],
      [-1., -1.,  1.,  1.],
      [-1., -1., -1., -1.]],
     [[-1., -1., -1.,  1.],  # 左下がりの直線
      [-1., -1.,  1., -1.],
      [-1.,  1., -1., -1.],
      [ 1., -1., -1., -1.]]])

print(sample_data.shape)
```

```
torch.Size([4, 4, 4])
```

実行結果

この通り、4×4 のデータ（テンソル）を 4 つ格納するテンソルとなっていますが、チャネル数の指定がないので、以下のようにして、「チャネル数が 1、サイズが 4×4 のデータ」を 4 つ格納するテンソルに変換しておきます。

```
sample_data = sample_data.reshape(4, 1, 4, 4)
```

サンプルデータの形状を変更

興味のある方は、このデータを出力して、かっこ「[]」がどこに増えているかを確認しておきましょう。

以上で元のデータとカーネルの準備ができました。それでは実際に畳み込みとプーリングで、これらがどう処理されるかを確認していきましょう。

畳み込み

畳み込みを行うには、上で作成した conv1 オブジェクトに、変数 sample_data を渡すだけです。その結果は変数 f_map1 に代入しておきましょう（前回も述べましたが、畳み込みの結果は特徴マップと呼ばれることがあります。そこでここでは変数名を「feature map」を表す「f_map1」としています。「1」はこれが 1 回目の畳み込みであることを示します）。

```
f_map1 = conv1(sample_data)
print(f_map1)
```

畳み込みを行い、その結果を表示

実行結果を以下に示します。

```
tensor([[[[-4., -6., -6., -4.],
          [ 6.,  9.,  9.,  6.],
          [-2., -3., -3., -2.],
          [ 0.,  0.,  0.,  0.]],

         [[ 0.,  0.,  0.,  0.],
          [ 0.,  1.,  1.,  0.],
          [ 0.,  1.,  1.,  0.],
          [ 0.,  2.,  2.,  0.]]],


        [[[ 0.,  0.,  0.,  0.],
          [ 0.,  1.,  1.,  2.],
          [ 0.,  1.,  1.,  2.],
          [ 0.,  0.,  0.,  0.]],

         [[-4.,  6., -2.,  0.],
          [-6.,  9., -3.,  0.],
          [-6.,  9., -3.,  0.],
          [-4.,  6., -2.,  0.]]],


        [[[-4., -4., -2.,  0.],
          [ 6.,  5.,  1., -2.],
          [-2.,  1.,  5.,  6.],
          [ 0., -2., -4., -4.]],

         [[ 0.,  2.,  0.,  0.],
          [ 0.,  1.,  1.,  0.],
          [ 0.,  1.,  1.,  0.],
          [ 0.,  0.,  2.,  0.]]],


        [[[ 0., -2.,  0.,  0.],
          [ 0.,  3.,  1.,  2.],
          [ 2.,  1.,  3.,  0.],
          [ 0.,  0., -2.,  0.]],

         [[ 0.,  0.,  2.,  0.],
          [-2.,  3.,  1.,  0.],
          [ 0.,  1.,  3., -2.],
          [ 0.,  2.,  0.,  0.]]]], grad_fn=<MkldnnConvolutionBackward>)
```

実行結果

注目してほしいのは、4 つのデータを入力したのに対して、出力された特徴マップは 8 つのデータを含んでいるように見えている点です。これは 1 つのデータを、2 つのカーネルで調べたそれぞれの結果、つまり、4×2 = 8 個の特徴マップが出力されたということです。最初の 2 つの塊は 1 つ目のデータ（横線）を 2 つのカーネルで調査した結果です。1 つ目は横方向に「6., 9., 9., 6.」という（他の数値と比べて）大きな数値が出ています。これは横線を、横線を示すカーネルで調査した結果であり、「なるほど」という感じがします。一方、その次の出力では 0、1、2 という値が存在するだけです。これは縦線の特徴を表すカーネルで調査した結果、そうしたところはあまり見られないことが数値的にも分かる感じがします。この傾向は 3 つ目のデータの調査結果（特徴マップ）でも同様です。

その下の 2 つは縦線を表すデータを調査したもので、これらについては上と逆の結果になっていることが分かります。

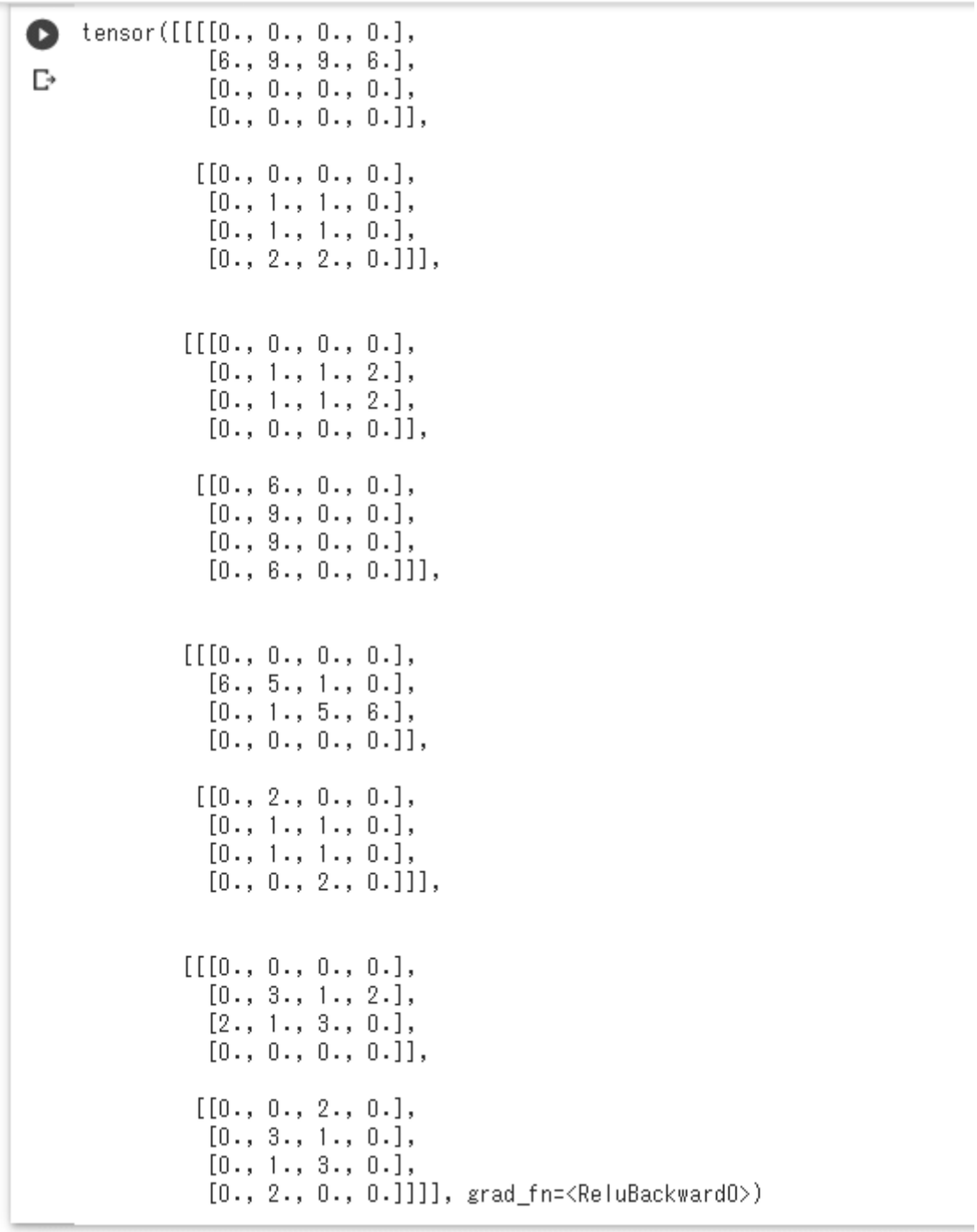
4 つ目のデータ（左下がりの直線）から得られた特徴マップからは、これが横線であることを示しているとも、縦線であることを示しているともいえないことが分かります。

畳み込みを行い特徴マップを得ることで、それぞれの元データの特徴がうまく得られました。これをプーリングするのですが、前回のコードではその前に活性化関数 ReLU を適用していました。この関数は 0 以下の値については 0 を、正の値については、元の値を返すという関数です。手順に従って、特徴マップにこれを適用してみましょう。

```
f_map1 = torch.nn.functional.relu(f_map1)
print(f_map1)
```

活性化関数 ReLU を特徴マップに適用

実行結果を以下に示します。



```
tensor([[[[0., 0., 0., 0.],
          [6., 9., 9., 6.],
          [0., 0., 0., 0.],
          [0., 0., 0., 0.]],

         [[0., 0., 0., 0.],
          [0., 1., 1., 0.],
          [0., 1., 1., 0.],
          [0., 2., 2., 0.]]],


        [[[0., 0., 0., 0.],
          [0., 1., 1., 2.],
          [0., 1., 1., 2.],
          [0., 0., 0., 0.]],

         [[0., 6., 0., 0.],
          [0., 9., 0., 0.],
          [0., 9., 0., 0.],
          [0., 6., 0., 0.]]],


        [[[0., 0., 0., 0.],
          [6., 5., 1., 0.],
          [0., 1., 5., 6.],
          [0., 0., 0., 0.]],

         [[0., 2., 0., 0.],
          [0., 1., 1., 0.],
          [0., 1., 1., 0.],
          [0., 0., 2., 0.]]],


        [[[0., 0., 0., 0.],
          [0., 3., 1., 2.],
          [2., 1., 3., 0.],
          [0., 0., 0., 0.]],

         [[0., 0., 2., 0.],
          [0., 3., 1., 0.],
          [0., 1., 3., 0.],
          [0., 2., 0., 0.]]]], grad_fn=<ReluBackward0>)
```

実行結果

負の値が 0 に変換され、正の値だけが残りました。実際には、この結果を使ってプーリングを行うことになります。

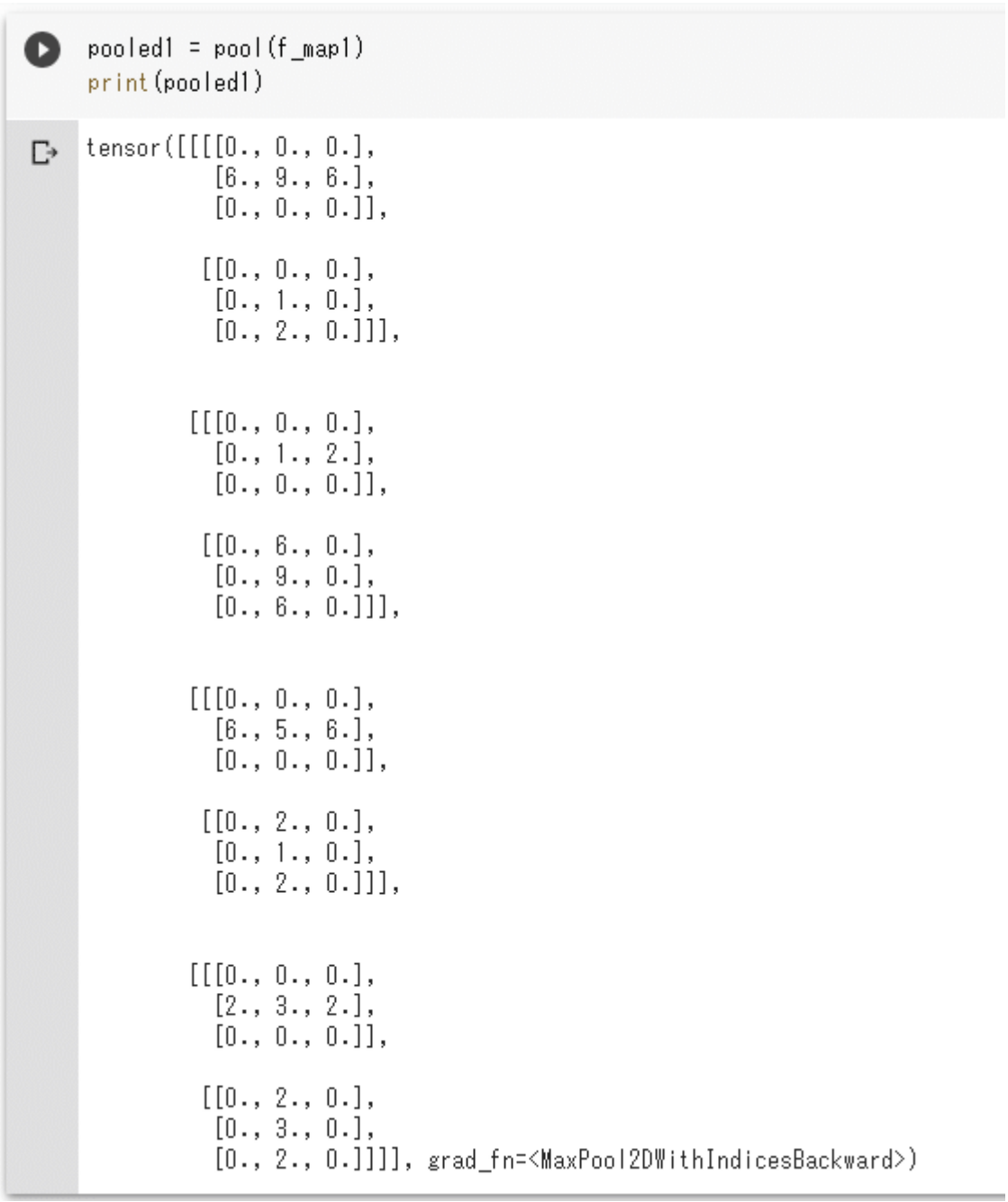
プーリング

では、実際にプーリングを行ってみましょう。といっても、これも先ほど作成した pool オブジェクトに活性化関数を通した結果を渡すだけです。

```
pooled1 = pool(f_map1)
print(pooled1)
```

活性化関数を適用した結果に対してプーリングを行う

pool オブジェクトの作成時には、パディングを指定していたので、ここでは特徴マップ（f_map1）の外側にパディングが付加されて（値は 0）、6×6 のサイズのデータを 2×2 のサイズの分割した各区画から最大値がピックアップされて、3×3 のサイズのデータが得られます。実行結果は次の通りです。



```
pooled1 = pool(f_map1)
print(pooled1)
```

```
tensor([[[[0., 0., 0.],
          [6., 9., 6.],
          [0., 0., 0.]],

         [[0., 0., 0.],
          [0., 1., 0.],
          [0., 2., 0.]]],


        [[[0., 0., 0.],
          [0., 1., 2.],
          [0., 0., 0.]],

         [[0., 6., 0.],
          [0., 9., 0.],
          [0., 6., 0.]]],


        [[[0., 0., 0.],
          [6., 5., 6.],
          [0., 0., 0.]],

         [[0., 2., 0.],
          [0., 1., 0.],
          [0., 2., 0.]]],


        [[[0., 0., 0.],
          [2., 3., 2.],
          [0., 0., 0.]],

         [[0., 2., 0.],
          [0., 3., 0.],
          [0., 2., 0.]]]], grad_fn=<MaxPool2DWithIndicesBackward>)
```

実行結果

1 つ目のデータから 3 つ目のデータまでは、横線、縦線という特徴がしっかりと出ていることが分かります（4 つ目のデータではきちんとどちらともいえない特徴が得られています）。これらが 2 回目の畳み込みの入力データとなります。

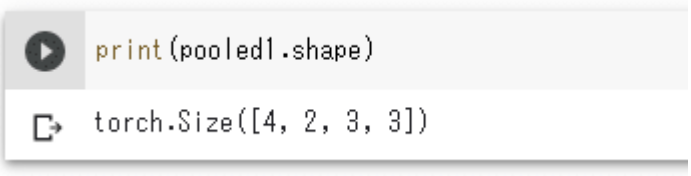
畳み込みとプーリング（2 回目）

ここで注意することがあります。それは、先ほどの畳み込みとプーリングで得られた結果の形状です。1 回目の畳み込みではカーネル（チャネル）が 2 つあったので、特徴マップは 1 つのデータについて 2 つ得られていました。これは、畳み込みへの入力チャネルが今度は 2 つあるということです。実際に確認してみましょう。

```
print(pooled1.shape)
```

プーリングまで行った結果の形状を確認

実行結果は次のようになります。



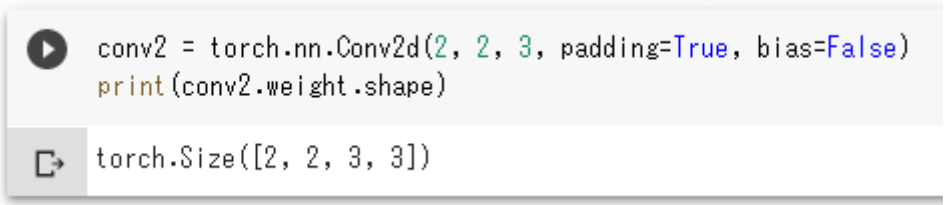
実行結果

これは「3×3 のサイズ、チャネル数が 2 のテンソル」を 4 つ格納するテンソルということです。そこで、2 回目の畳み込みを行う Conv2d クラスのインスタンス（「conv2」としましょう）は次のようにして生成することになります。ここでは、conv1 オブジェクトと同様に、カーネルは 2 つ、パディングを付加、バイアスはなしとしています。

```
conv2 = torch.nn.Conv2d(2, 2, 3, padding=True, bias=False)
print(conv2.weight.shape)
```

2 回目の畳み込みで使用する conv2 オブジェクトの作成

このときには、カーネルの形状についても注意が必要です。実行結果を見てください。



実行結果

最初のカーネルの形状は「torch.Size([2, 1, 3, 3])」、つまり「3×3 のサイズでチャネル数が 1 のテンソルが 2 つ」となっていましたが。この実行結果が意味するのは、2 回目の畳み込みで使うカーネルは「torch.Size([2, 2, 3, 3])」、つまり「3×3 のサイズでチャネル数が 2 のテンソルが 2 つ」です。入力するチャネルの数に合わせて、カーネルのチャネル数も 2 つとなるわけです。2 回目の畳み込みでどのようなカーネルを作ればよいかの判断は難しいのですが（だからこそ、CNN ではそうしたものを自動的に作成してくれるともいえるでしょう）、ここでは上で見た特徴に合わせて、横線と縦線を表すものを重複して指定しておきましょう（実際に学習をさせると全く別のカーネルが得られるのではないかと、筆者は予想しています）。

```
kernels2 = torch.tensor([
    [[[-1., -1., -1.],
      [ 1.,  1.,  1.],
      [-1., -1., -1.]],
     [[-1., -1., -1.],
      [ 1.,  1.,  1.],
      [-1., -1., -1.]]],

    [[[-1.,  1., -1.],
      [-1.,  1., -1.],
      [-1.,  1., -1.]],
     [[-1.,  1., -1.],
      [-1.,  1., -1.],
      [-1.,  1., -1.]]]])

print(conv2.weight.shape)
conv2.weight.data = kernels2
```

2 回目の畳み込みで使用するカーネル

1 つ目のカーネルは横線を表すテンソルを 2 つ重複させています。2 つ目は同様に縦線を表すテンソルを重複させています（もしかしたら、カーネルは 1 つで、それぞれのチャネルに横線と縦線を表すものを指定するとよかったのかもしれませんが、ここではこのまま進めましょう）。

実行結果を以下に示します。カーネルの形状が上で見た重みの形状と一致していることを確認してください。

```
kernels2 = torch.tensor([
    [[[-1., -1., -1.],
      [ 1.,  1.,  1.],
      [-1., -1., -1.]],
     [[-1., -1., -1.],
      [ 1.,  1.,  1.],
      [-1., -1., -1.]]],

    [[[-1.,  1., -1.],
      [-1.,  1., -1.],
      [-1.,  1., -1.]],
     [[-1.,  1., -1.],
      [-1.,  1., -1.],
      [-1.,  1., -1.]]]])

print(conv2.weight.shape)
conv2.weight.data = kernels2
```

```
torch.Size([2, 2, 3, 3])
```

実行結果

この点以外は、これまでと同様なので、以下では一気にコードを実行してしまいましょう。途中経過が気になる方はコメントアウトしている行を適宜実行して、特徴マップやそれらを活性化関数に通した結果を確認してください。

```
f_map2 = conv2(pooled1)  # 畳み込み
#print(f_map2)
f_map2 = torch.nn.functional.relu(f_map2)  # 活性化関数
#print(f_map2)
pooled2 = pool(f_map2)  # プーリング
print(pooled2)
```

畳み込みとプーリングの実行（2回目）

実行結果を以下に示します。

```
f_map2 = conv2(pooled1)  # 畳み込み
#print(f_map2)
f_map2 = torch.nn.functional.relu(f_map2)  # 活性化関数
#print(f_map2)
pooled2 = pool(f_map2)  # プーリング
print(pooled2)
```

```
tensor([[[[ 0.,  0.],
          [14., 20.]],

         [[ 0.,  0.],
          [ 0.,  0.]]],


        [[[ 0.,  0.],
          [ 0.,  0.]],

         [[ 0., 14.],
          [ 0., 20.]]],


        [[[ 0.,  0.],
          [ 8., 14.]],

         [[ 0.,  0.],
          [ 0.,  0.]]],


        [[[ 0.,  0.],
          [ 4.,  6.]],

         [[ 0.,  4.],
          [ 0.,  6.]]]], grad_fn=<MaxPool2DWithIndicesBackward>)
```

実行結果

今度は 2×2 のテンソル 2 つが組になった結果が得られました。実際には、これらを全結合層へ入力することで、それらが横線なのか、縦線なのかの推測が行われるということです。そこで、全結合層へ入力しやすいように、その形状を変更しておきます。

```
print(pooled2.shape)
pooled2 = pooled2.reshape(-1, 2 * 2 * 2)
print(pooled2)
```

プーリング結果の形状を確認して、その形状を変更する

実行結果を以下に示します。

```
print(pooled2.shape)
pooled2 = pooled2.reshape(-1, 2 * 2 * 2)
print(pooled2)
```

```
torch.Size([4, 2, 2, 2])
tensor([[ 0.,  0., 14., 20.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0., 14.,  0., 20.],
        [ 0.,  0.,  8., 14.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  4.,  6.,  0.,  4.,  0.,  6.]], grad_fn=<ViewBackward>)
```

実行結果

実行結果を見ると分かりますが、元は「2×2 のサイズ、チャネル数が 2 のテンソルが 4 つ」だったものを上のコードではこれを 2 次元のデータに展開しました（reshape メソッドに指定している引数は「任意の要素数 , チャネル数 × 行数 × 列数」を意味します）。つまり、4×8 のサイズのテンソルになっています。この 8 個の数値が 1 つのデータがどんな特徴を持つかを示すデータ「特徴量」と呼ばれるものです。

最初の行と3行目のデータは前半4つのデータに大きな数値が含まれています。これらは横線を示すデータだったので、1 つ目のカーネルでの調査結果である前半に強い数値が出ているということです。2 行目は縦線なので、その逆になっています。これらのことから、横線と縦線では多くの場合、このような特徴がよく見られることが示唆されます（ただし、畳み込みとプーリングの結果によっては、上の傾向とは異なる特徴量が得られることもあります。そうした要素も含めて適切に推測できるように、横線／縦線の推測を行う全結合層では重みとバイアスの学習が行われます）。

ここであやめの品種の分類のことを思い出してください。あやめの品種の分類では、あやめの品種を表す特徴量は既にデータセットとして用意されていました。しかし、前回の CNN による手書き数字の認識や今見たような横線、縦線の畳み込み／プーリングでは、特徴量を CNN によって取り出していると考えられます。さらにいえば、特徴量を取り出すために必要な重み（やバイアス）も学習を通して自動的に得られます。ニューラルネットワークの大きなメリットがここにあるといえます（ここでは重みは筆者による決め打ちでしたが）。

大量のデータから人が、特徴量が見つけ出したり、それを見つけるための重みやバイアスを決定したりするのは非常に難しいことですが、ニューラルネットワークはそれらを自動的に行ってくれるのです。

後はこれを（適切に重みとバイアスを学習済みの）全結合型ニューラルネットワークに入力することで、4 つのデータが横線なのか縦線なのかを判断できるということになります。

全結合型のニューラルネットワークへの入力

というわけで、先ほど得た 4 つのデータ（特徴量）を全結合型のニューラルネットワークに入力するのはよいのですが、ここまでの話では元のデータが 4 つしかありません（というか、それらが何を表すものかを推測するニューラルネットワークを作ろうというのですから、本来、それら 4 つのデータを使うわけにはいきません）。

そこで今見た 4 つのデータとは別に 40 個のデータを作って、そこから 40 個の特徴量を得た上で、それらを基に学習を行う全結合型のニューラルネットワークをやっつけで作成しました。ここからは本筋とはあまり関係ないので、サクサクと飛ばしていきます。完全なコードと実行結果は公開しているノートブックを参照してください。

```
inputs = torch.tensor(
    [[[-1., -1., -1., -1.],
      [ 1.,  1.,  1.,  1.],
      [-1., -1., -1., -1.],
      [-1., -1., -1., -1.]],
     [[-1.,  1., -1., -1.],
      [-1.,  1., -1., -1.],
      [-1.,  1., -1., -1.],
      [-1.,  1., -1., -1.]],

              # 省略

     [[-1., -1.,  1., -1.],
      [-1., -1.,  1., -1.],
      [-1., -1., -1.,  1.],
      [-1., -1., -1.,  1.]],
     [[ 1., -1., -1., -1.],
      [ 1., -1., -1., -1.],
      [-1.,  1., -1., -1.],
      [-1., -1.,  1., -1.]]])

labels = torch.tensor([
  0, 1, 0, 0, 1, 0, 0, 1, 0, 1,
  1, 0, 1, 0, 1, 1, 0, 1, 0, 0,
  1, 1, 0, 1, 0, 0, 1, 1, 0, 1,
```

```
  0, 0, 1, 0, 1, 0, 1, 0, 1, 1
], dtype=torch.float)
```

横線／縦線を表す 40 個のデータとその正解ラベル

先ほどと同じ手順で、これらのデータから 40 個の特徴量を抽出します。

```
inputs = inputs.reshape(-1, 1, 4, 4)
c1 = conv1(inputs)
c1 = torch.nn.functional.relu(c1)
p1 = pool(c1)
c2 = conv2(p1)
c2 = torch.nn.functional.relu(c2)
p2 = pool(c2)
inputs2fc = p2.reshape(-1, 2 * 2 * 2)
inputs2fc = inputs2fc.detach()  # 勾配計算はしない

#print(inputs2fc)
#for idx, item in enumerate(zip(inputs2fc, labels)):
#    print(f'idx: {idx}, item: {item[0].data}, label: {item[1]}')
```

40 個のデータから特徴量を抽出

これらのデータを使って学習を行う全結合型のニューラルネットワークは次のようにします。

```
class Net(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = torch.nn.Linear(8, 10)
        self.fc2 = torch.nn.Linear(10, 1)
    def forward(self, x):
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        x = torch.sigmoid(x)
        return x
```

横線か縦線かを推測する全結合型のニューラルネットワーククラス

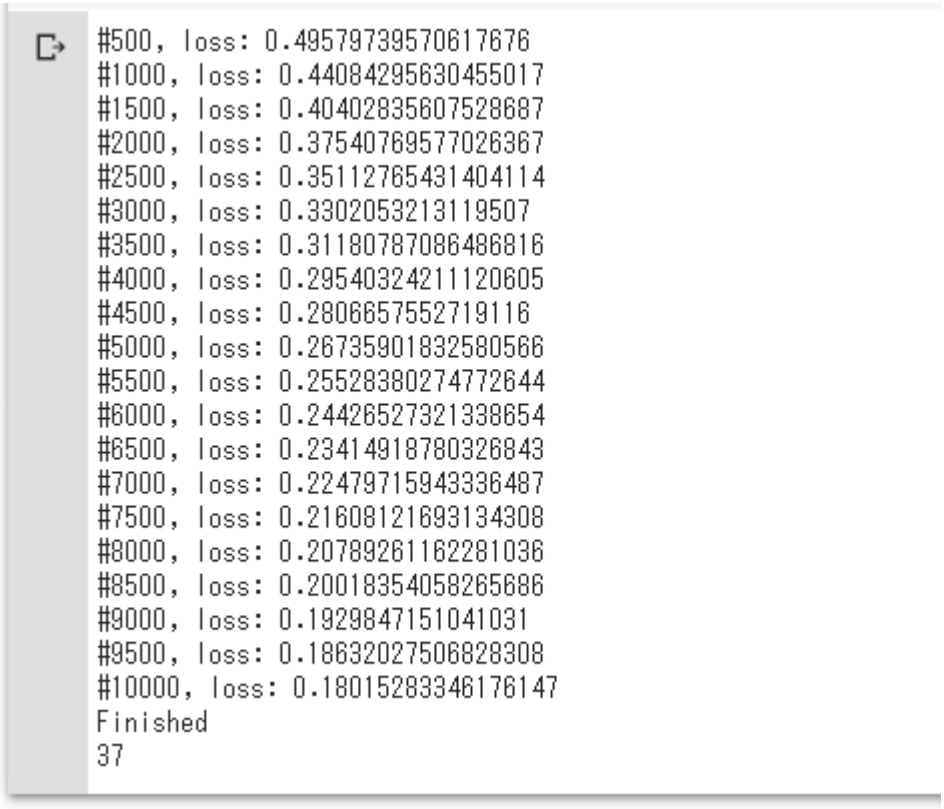
このニューラルネットワークモデルへの入力は、既に見た通り 8 個だったので、入力層のノード数は 8 としています（隠れ層のノード数「10」は筆者が適当に決めたものです）。ここでは横線か縦線かを判断するので（2 クラス分類）、出力層のノード数は 1 つになっています。この値が 0.5 より小さければ、0（横線）、0.5 以上なら縦線（1）という判断をするように正解ラベルを設定してあります（インスタンス変数 fc2 の出力を活性化関数である torch.sigmoid 関数に渡すことで、出力が 0 ～ 1 の範囲に収まるようにしている点に注意）。

学習を行うコードは次の通りです。

```
import torch.optim as optim

net = Net()

criterion = torch.nn.BCELoss()
optimizer = optim.SGD(net.parameters(), lr=0.03)

EPOCHS = 10000

for epoch in range(1, EPOCHS + 1):
    optimizer.zero_grad()
    outputs = net(inputs2fc)
    loss = criterion(outputs.squeeze(), labels)
    loss.backward()
    optimizer.step()

    if epoch % 500 == 0:
        print(f'#{epoch}, loss: {loss.data}')

print('Finished')

predicted = (outputs.reshape(1, -1) + 0.5).int()
(predicted == labels).sum().item()
```

学習を行い、最後の学習結果（outputs）を使って、簡易的にその精度を確認

実行結果を以下に示します。



```
#500, loss: 0.49579739570617676
#1000, loss: 0.44084295630455017
#1500, loss: 0.40402835607528687
#2000, loss: 0.37540769577026367
#2500, loss: 0.35112765431404114
#3000, loss: 0.3302053213119507
#3500, loss: 0.31180787086486816
#4000, loss: 0.29540324211120605
#4500, loss: 0.2806657552719116
#5000, loss: 0.26735901832580566
#5500, loss: 0.2552838027477264 4
#6000, loss: 0.24426527321338654
#6500, loss: 0.23414918780326843
#7000, loss: 0.22479715943336487
#7500, loss: 0.21608121693134308
#8000, loss: 0.20789261162281036
#8500, loss: 0.20018354058265686
#9000, loss: 0.1929847151041031
#9500, loss: 0.18632027506828308
#10000, loss: 0.18015283346176147
Finished
37
```

実行結果

それなりに学習できたようなので、このニューラルネットワークモデルに、先ほど作成した 4 つの特徴量を入力してみましょう。

```
print(pooled2)
result = net(pooled2)
print(result)
```

学習後のニューラルネットワークモデルに、4 つの特徴量を入力

4 つのデータは順に横線／縦線／よれた横線／左下がりの直線を表すものでした。これらがうまく判断できたでしょうか。

```
print(pooled2)
result = net(pooled2)
print(result)
```

```
tensor([[ 0.,  0., 14., 20.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  0.,  0.,  0., 14.,  0., 20.],
        [ 0.,  0.,  8., 14.,  0.,  0.,  0.,  0.],
        [ 0.,  0.,  4.,  6.,  0.,  4.,  0.,  6.]], grad_fn=<ViewBackward>)
tensor([[0.0092],
        [0.9907],
        [0.0471],
        [0.5186]], grad_fn=<SigmoidBackward>)
```

実行結果

1 つ目と 3 つ目の推測値は横線を表す 0 に極めて近いものになっていますし、2 つ目の推測値は縦線を表す 1 に極めて近いものになっているので、どうやらこのニューラルネットワークモデルはうまいこと推測を行えているようです。最後のデータについては、0 にも 1 にも近くない中間の値となっているので、左下がりの直線をどう判断すればよいか難しいことを表しているといえます。

最後に、今までに見てきた畳み込み、プーリング、全結合を行うニューラルネットワーククラスをひとまとめにしたクラスも示しておきましょう。興味のある方は、上で示した学習を行うコードを少し改変して、学習を行い、畳み込み層の重みがどのようになるかを確認してみてください（公開しているノートブックにはそれらのコードも含めてあります）。

```
class Net2(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(1, 2, 3, padding=True, bias=False)
        self.pool = torch.nn.MaxPool2d(2, padding=1)
        self.conv2 = torch.nn.Conv2d(2, 2, 3, padding=True, bias=False)
        self.fc1 = torch.nn.Linear(8, 10)
        self.fc2 = torch.nn.Linear(10, 1)
    def forward(self, x):
        x = self.conv1(x)
        x = torch.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = torch.relu(x)
        x = self.pool(x)
        x = x.reshape(-1, 2 * 2 * 2)
        x = self.fc1(x)
        x = torch.sigmoid(x)
        x = self.fc2(x)
        x = torch.sigmoid(x)
        return x
```

畳み込みから全結合までをまとめたニューラルネットワーククラス

今回は前回に説明した内容を、実際にコードを動かしながら確認してみました。次回は学習に関連する事項について少し見ていくことにします。

プログラムを関数にまとめて
実行結果をグラフにプロットしよう

ニューラルネットワークを使って学習や評価を行うコードを関数にまとめてみます。また、データセットを学習に使うものと精度評価に使うものに分割する方法、学習結果のグラフ化、過学習の抑制などについても簡単に見てみましょう。

（2020 年 06 月 12 日）

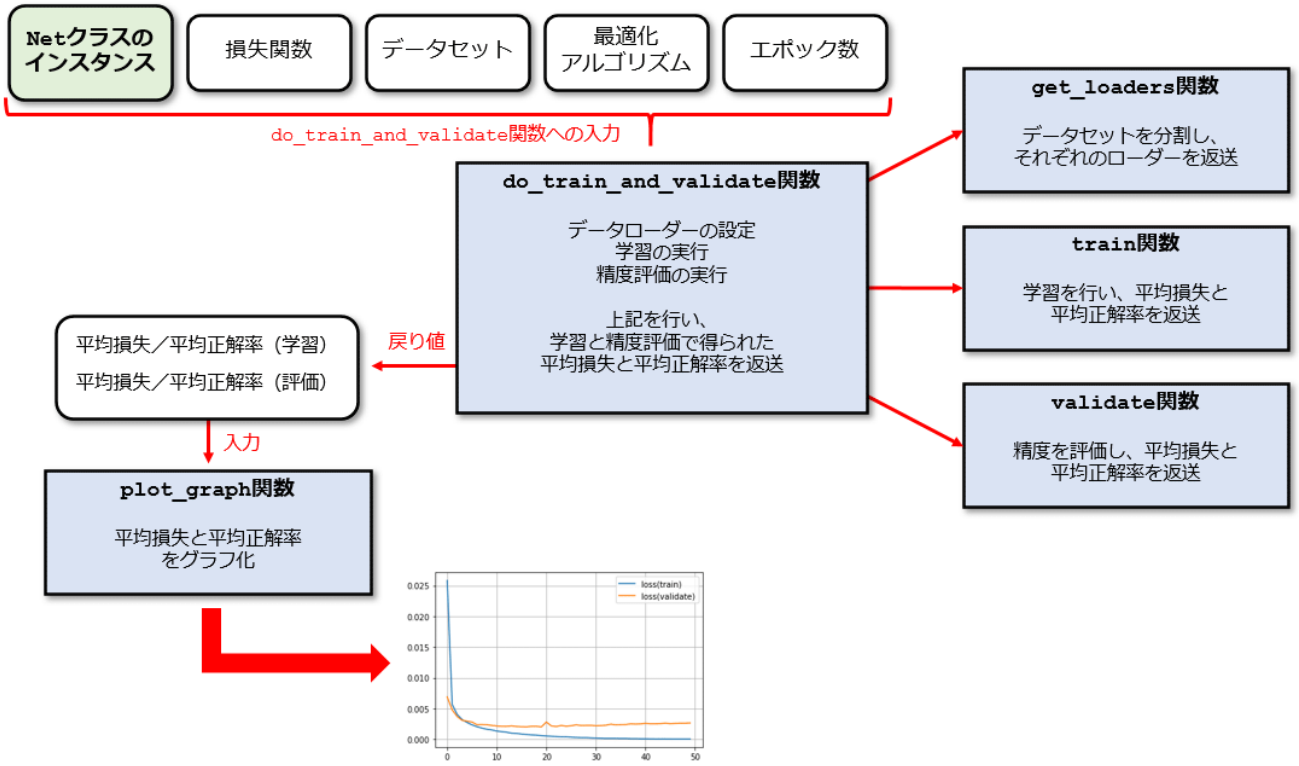
前回までは Google Colab のノートブックを使って、プログラムをベタに書いてきました。今回は CNN を使って MNIST の手書き数字を認識するプログラムを幾つかのクラスと関数にまとめ、それらを実行し、実行結果をグラフにプロットしてみましょう。

今回作成するプログラム

ここでは次のようなクラス／関数を作成します。

- Net クラス：CNN を使って手書き数字を認識するクラス
- train 関数：Net クラス（と損失関数、最適化アルゴリズム）を用いて、学習を行う関数
- validate 関数：学習後の Net クラスのインスタンスの精度を評価する関数
- do_train_and_validate 関数：MNIST の手書き数字を受け取り、train 関数と validate 関数を呼び出す関数
- get_loaders 関数：訓練データを、訓練用／評価用のデータセットに分割して、それらを読み出すために使用するデータローダーを返す関数
- plot_graph 関数：1 エポックごとの損失、精度をグラフにプロットするための関数

これらの関係をざっくりと図にしたものを以下に示します。



今回作成するプログラムの構成

Net クラスは「CNN なんて怖くない！　その基本を見てみよう」の最後で紹介したコードを（ほぼ）そのまま使用しています。

学習や精度の評価を行うコードも基本的にはこれまでと同様で、それらを関数にまとめただけなので、コードを見て、「難しい……」と思うことはあまりないと思います。ただし、今回は合計 6 万個の学習用の手書き数字を訓練に使うものと、精度の評価に使うものに分割してみましょう。テスト用の手書き数字は、精度評価が終わったニューラルネットワークモデルに入力して、それが未知のデータに対しても汎用的に使えるかどうかの確認に使います。

訓練データを分割して、それぞれに対応するデータローダーを作成するために、今回は上に示した get_loaders 関数を定義することにしました。

また、今回は 1 エポック（6 万個のデータ）の学習と精度評価を複数回繰り返して、学習を進めるたびにニューラルネットワークモデルがどのくらい賢くなっていくかを調べることにしました。繰り返しの回数は 50 回です（この回数は筆者が適当に決めました）。そのため、全てが終わるまでには少々の時間がかかることは頭の中に入れておきましょう。

それではこれらのクラスと関数、それらに付随するコードを見ていきましょう。

インポート文とデータセットの準備

クラスや関数を見る前に、必要となるモジュールなどのインポートと、データセットを準備しておきます。これを行うコードが以下です。

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.optim as optim
import matplotlib.pyplot as plt
from torch import nn
from torchvision.datasets import MNIST
from torch.utils.data import DataLoader
from torch.utils.data import random_split

transform = transforms.Compose([transforms.ToTensor(), transforms.
Normalize((0.5,), (0.5,))])

BATCH_SIZE = 20

trainset = MNIST(root='./data', train=True, transform=transform,
download=True)
#trainloader = DataLoader(trainset, batch_size=BATCH_SIZE, shuffle=True)

testset = MNIST(root='./data', train=False, transform=transform,
download=True)
testloader = DataLoader(testset, batch_size=BATCH_SIZE, shuffle=False)
```

impot 文とデータセットの準備

import 文が多くなっていますが、これは今回使用するものを（コードを書いた後に筆者が整理して）全てここにまとめたからです。また、画面の都合でコードが横に長くなりすぎないように、明示的にインポートしているクラスもあります（MNIST クラスや DataLoader クラスなど）。

その後にはこれまでに何度も見てきたデータセットを取得するコードと（テストデータについては）データセットからデータローダーを作成するコードが続きます。先ほども述べた通り、今回は訓練データを訓練につかうものと精度の評価に使うものに分割するので、ここでは訓練データ用のデータローダーは作成しません（コードをあえてコメントアウトしてあります）。

Net クラス

次に Net クラスを見てみましょう。そのコードを以下に示します。

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 16, 64)
        self.fc2 = nn.Linear(64, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = nn.functional.relu(x)
        x = self.pool(x)
        x = self.conv2(x)
        x = self.pool(x)
        x = x.reshape(-1, 16 * 16)
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.fc2(x)
        return x
```

Net クラス

既に見飽きた感もあるでしょうが、Net クラスでは CNN の畳み込み層（Conv2d クラス）とプーリング層（MaxPool2d クラス）を使って、2 次元データ（手書き数字）から特徴量を抽出し、それらを全結合層（Linear クラス）へ渡すようになっています。これらが実際にどのような振る舞いをするのかについては前回の記事を参照してください。

train 関数と validate 関数

上記の Net クラスのインスタンスを使って、学習を実行するのが train 関数です。そのコードは次のようになっています。

```
def train(net, dataloader, criterion, optimizer):
    net.train()

    total_loss = 0.0
    total_correct = 0
    for inputs, labels in dataloader:
        optimizer.zero_grad()
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        total_correct += (predicted == labels).sum().item()

    avg_loss = total_loss / len(dataloader.dataset)
    accuracy = total_correct / len(dataloader.dataset)

    return avg_loss, accuracy
```

train 関数

この関数はニューラルネットワークモデル（net。この場合は Net クラスのインスタンス）、データローダー（dataloader。下で説明する get_loaders 関数で得た学習で使用する分の訓練データを読み込むためのデータローダー）、損失関数（criterion）、最適化アルゴリズム（optimizer）をパラメーターに受け取ります。

関数内のコードでは、まず「net.train()」呼び出しを行っています。これはニューラルネットワーククラスのインスタンスを「訓練モード」に設定するものです。PyTorch が提供するニューラルネットワークモジュールでは訓練モードと評価モードで動作が異なるものがあるため、今回は明示的にこれを設定しています。

その後にある 2 つの変数 total_loss と total_correct は、関数が受け取ったデータローダーに格納されている全ての要素を使って学習を行う際に算出される損失の総計（total_loss）と、ニューラルネットワークモデルからの出力と正解ラベルとを比較して正解だった数の総計（total_correct）を蓄積していくのに使っています。

学習を行うコード自体はこれまでと同様です。

最後に、損失の総計をデータローダーのデータ数で除算することで、今回の学習における損失の平均値を、また、正解数の総計を同じくデータローダーのデータ数で除算することで今回の学習における精度の平均を計算して、それらを戻り値としています。

これらのデータは、後でエポックごとの損失と精度をグラフにプロットするために使用します。

一方、上のコードで学習を行ったニューラルネットワークモデルを使用して、その精度を確認するために使用するのが以下に示す validate 関数です。

```
def validate(net, dataloader, criterion):
    net.eval()

    with torch.no_grad():
        total_loss = 0.0
        total_correct = 0
        for inputs, labels in dataloader:
            outputs = net(inputs)
            loss = criterion(outputs, labels)

            total_loss += loss.item()
            _, predicted = torch.max(outputs, 1)
            total_correct += (predicted == labels).sum().item()

    avg_loss = total_loss / len(dataloader.dataset)
    avg_accuracy = total_correct / len(dataloader.dataset)

    return avg_loss, avg_accuracy
```

validate 関数

train 関数とは異なり、validate 関数は既存のニューラルネットワークモデルの精度を評価するのが目的で、学習を行うことはありません。そのため、この関数ではニューラルネットワークモデル（net）、データローダー（dataloader）、損失関数（criterion）をパラメーターに受け取るようにしています。

内部のコードでは、最初に「net.eval()」呼び出しで、ニューラルネットワーククラスのインスタンスを「評価モード」に設定しています。これは train 関数で行っていた「net.train()」呼び出しと対になるものです。

その後は、「with torch.no_grad()」という with 文のブロックで「ここでは学習に必要な勾配計算は行わない」ことを明示した上で（こうすることで PyTorch 内部での関連する処理を行わずに済むようになります）、データローダーに受け取った精度評価用のデータをニューラルネットワークモデルへと入力し、そこから損失と正解数を計算して、train 関数と同様に 2 つの変数 total_loss と total_correct に蓄積していきます。

最後に損失の総計と正解数の総計からそれらの平均値を求めて、戻り値としています。これらを実際に呼び出しているのが次に示す do_train_and_validate 関数です。

do_train_and_validate 関数

この関数は、実質的に学習と精度評価を行うキモとなっています。ノートブックでは、Net クラス／損失関数／最適化アルゴリズムのインスタンスを作成して、最初に生成した訓練データ（データセット）と繰り返しの回数（エポック数）と共にこの関数を呼び出すことで、学習と精度評価を行うようになっています。

```
def do_train_and_validate(net, trainset, criterion, optimizer, epochs):

    trainloader, validloader = get_loaders(trainset)

    history = {}
    history['train_loss_values'] = []
    history['train_accuracy_values'] = []
    history['valid_loss_values'] = []
    history['valid_accuracy_values'] = []

    for epoch in range(1, epochs + 1):
        print(f'epoch: {epoch:2}')

        t_loss, t_accu = train(net, trainloader, criterion, optimizer)
```

```
        v_loss, v_accu = validate(net, validloader, criterion)

        print(f'train_loss: {t_loss:.6f}, train_accuracy: {t_accu:3.4%},',
              f'valid_loss: {v_loss:.6f}, valid_accuracy: {v_accu:3.4%}')

        history['train_loss_values'].append(t_loss)
        history['train_accuracy_values'].append(t_accu)
        history['valid_loss_values'].append(v_loss)
        history['valid_accuracy_values'].append(v_accu)

    return history
```

do_train_and_validate 関数

今も述べましたが、この関数はニューラルネットワークモデル（net）、訓練データ（trainset）、損失関数（criterion）、最適化アルゴリズム（optimizer）、繰り返し回数（epochs）をパラメーターに受け取ります。

受け取った訓練データ（trainset）は、次に紹介する get_loaders 関数に渡すことで、実際に学習に使用するデータセットを読み込むためのデータローダーと、その精度評価に使用するデータセットを読み込むためのデータローダーとが得られます。ここでは前者を変数 trainloader に、後者を変数 validloader に保存しています。

その下にある辞書 history には、以下の for 文で実行される学習と訓練で得られた損失／正解率の平均値をリストの要素として蓄積していきます。

- history['train_loss_values']：1 エポックの学習で得られた平均損失を要素とするリスト
- history['train_accuracy_values']：1 エポックの学習で得られた平均正解率を要素とするリスト
- history['valid_loss_values']：1 エポックの精度評価で得られた平均損失を要素とするリスト
- history['valid_accuracy_values']：1 エポックの精度評価で得られた平均正解率を要素とするリスト

その後は、パラメーター epochs に指定された回数だけ for 文のループで、train 関数および validate 関数を呼び出して、上記の辞書の要素（リスト）にそれぞれの値を追加していくようにしました。これにより、50 回の繰り返しを指定すれば、学習によって得られた平均損失／平均正解率と、精度評価によって得られた平均損失／平均正解率が 50 回分たまります。今回はこれをグラフにプロットすることで、学習がどのように進められていくかを確認することにしましょう。

そして、今回は訓練データを、学習に使用するものと精度評価に使用するものの 2 つに分割するのでした。これを行うのが、次に紹介する get_loaders 関数です。

get_loaders 関数

既に述べたように、今回は MNIST の手書き数字の訓練データを学習に使うものと、精度評価に使うものの 2 つに分割します。データセットを分割するには、幾つかの方法がありますが、今回はシンプルに PyTorch の random_split 関数を使用します。

この関数はデータセットと、分割後のデータセットの要素数を格納するリストをパラメーターに受け取り、データセットを分割してくれます。分割後のデータセットの要素はランダムに並べ替えられます。MNIST の手書き数字（訓練データ）は 6 万個あるので、ここではそのうちの 8 割を学習に、残りの 2 割を精度評価に使うことにしましょう。後は分割後の 2 つのデータセットを基に DataLoader クラスに渡して、データローダーを作成するだけです。

実際の get_loaders 関数のコードは次のようになります。

```
def get_loaders(dataset):
    train_size = int(len(dataset) * 0.8)
    valid_size = len(dataset) - train_size
    train_set, valid_set = random_split(dataset, (train_size, valid_size))
    trainloader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True)
    validloader = DataLoader(valid_set, batch_size=BATCH_SIZE, shuffle=False)
    return trainloader, validloader
```

get_loaders 関数

get_loaders 関数では、最初に述べたようパラメーター dataset に受け取ったデータセットの長さに 0.8 を乗じた値を学習に使用するデータセットの要素数として（48,000 個）、残りを精度評価に使うデータセットの要素数としています（12,000 個）。

この 2 つの値と、データセットを random_split 関数に渡した後は、得られたデータセットを使って 2 つのデータローダーを作成して、それを戻り値としています。

plot_graph 関数

最後に、do_train_and_validate 関数の戻り値である平均損失／平均正解率をグラフにプロットする plot_graph 関数を紹介します。

```
def plot_graph(values1, values2, rng, label1, label2):
    plt.plot(range(rng), values1, label=label1)
    plt.plot(range(rng), values2, label=label2)
    plt.legend()
    plt.grid()
    plt.show()
```

plot_graph 関数

ここでは 2 つの平均損失、2 つの平均正解率を別個のグラフにプロットするものとして、2 つの値と X 軸の値となる繰り返し回数、それぞれの値のラベルをパラメーターに受け取るようにしています。

これを Matplotlib.pyplot を使用して、グラフに描画しているだけです。

以上で準備ができたので、次に実際にこれらを使って学習と精度評価を行ってみましょう。

学習と精度評価の実行

既に述べたように、学習と精度評価を実行するには do_train_and_validate 関数を呼び出すだけです。ただし、その前に Net クラスのインスタンスや損失関数、最適化アルゴリズムの選択などを行う必要があります。また、ここでは繰り返し回数（エポック数）は 50 とします。

```
net = Net()
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
EPOCHS = 50

history = do_train_and_validate(net, trainset, criterion, optimizer, EPOCHS)

print('Finished')
```

学習と精度評価を実行する

このコードを実行すると、次のようにエポックごとに学習で得られた平均損失／平均正解率、精度評価で得られた平均損失／平均正解率が画面に表示されていきます。

```
net = Net()
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
EPOCHS = 50

history = do_train_and_validate(net, trainset, criterion, optimizer, EPOCHS)

print('Finished')
```

```
epoch:  1
train_loss: 0.025837, train_accuracy: 84.8063%, valid_loss: 0.006892, valid_accuracy: 95.7917%
epoch:  2
train_loss: 0.005678, train_accuracy: 96.5396%, valid_loss: 0.004780, valid_accuracy: 97.0583%
epoch:  3
train_loss: 0.004045, train_accuracy: 97.5563%, valid_loss: 0.003705, valid_accuracy: 97.7167%
epoch:  4
train_loss: 0.003206, train_accuracy: 98.0542%, valid_loss: 0.003080, valid_accuracy: 98.0167%
epoch:  5
train_loss: 0.002726, train_accuracy: 98.3646%, valid_loss: 0.002945, valid_accuracy: 98.2250%
epoch:  6
train_loss: 0.002340, train_accuracy: 98.5125%, valid_loss: 0.002801, valid_accuracy: 98.3833%
epoch:  7
train_loss: 0.002045, train_accuracy: 98.7229%, valid_loss: 0.002370, valid_accuracy: 98.5417%
epoch:  8
train_loss: 0.001811, train_accuracy: 98.8937%, valid_loss: 0.002398, valid_accuracy: 98.6167%
epoch:  9
train_loss: 0.001631, train_accuracy: 98.9729%, valid_loss: 0.002360, valid_accuracy: 98.5333%
epoch: 10
train_loss: 0.001514, train_accuracy: 99.0354%, valid_loss: 0.002249, valid_accuracy: 98.7833%
epoch: 11
train_loss: 0.001329, train_accuracy: 99.1583%, valid_loss: 0.002161, valid_accuracy: 98.7500%
epoch: 12
train_loss: 0.001238, train_accuracy: 99.2104%, valid_loss: 0.002113, valid_accuracy: 98.8583%
epoch: 13
train_loss: 0.001138, train_accuracy: 99.2667%, valid_loss: 0.002108, valid_accuracy: 98.8333%
epoch: 14
```

実行結果

実行完了までにはそれなりの時間がかかる点には注意してください。

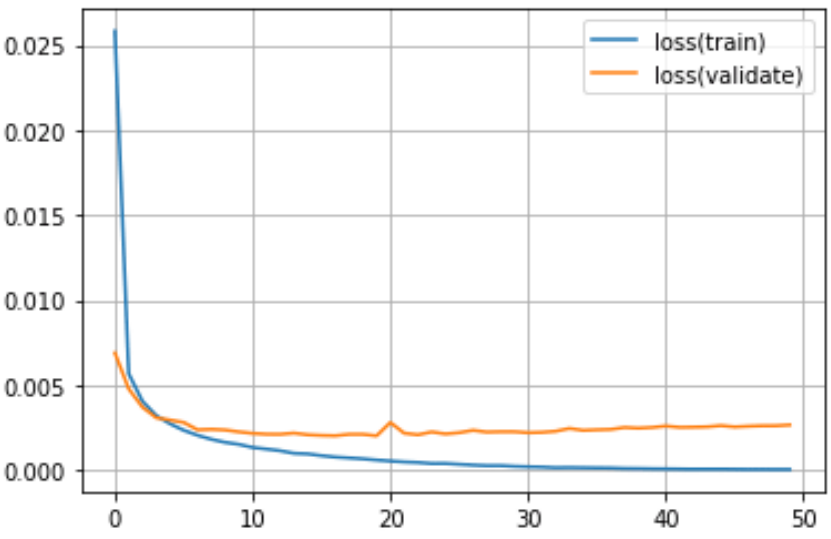
実行が終わったら、得られた損失や成果率をグラフにプロットしてみましょう。まずは、平均損失からです。以下のコードでは辞書に格納されていた各値を個別の変数に展開した後に、それらを使ってグラフをプロットしています。

```
t_losses = history['train_loss_values']
t_accus = history['train_accuracy_values']
v_losses = history['valid_loss_values']
v_accus = history['valid_accuracy_values']

plot_graph(t_losses, v_losses, EPOCHS, 'loss(train)', 'loss(validate)')
```

学習と精度評価での平均損失を比較するグラフをプロット

実行結果を以下に示します。

```
t_losses = history['train_loss_values']
t_accus = history['train_accuracy_values']
v_losses = history['valid_loss_values']
v_accus = history['valid_accuracy_values']

plot_graph(t_losses, v_losses, EPOCHS, 'loss(train)', 'loss(validate)')
```



実行結果

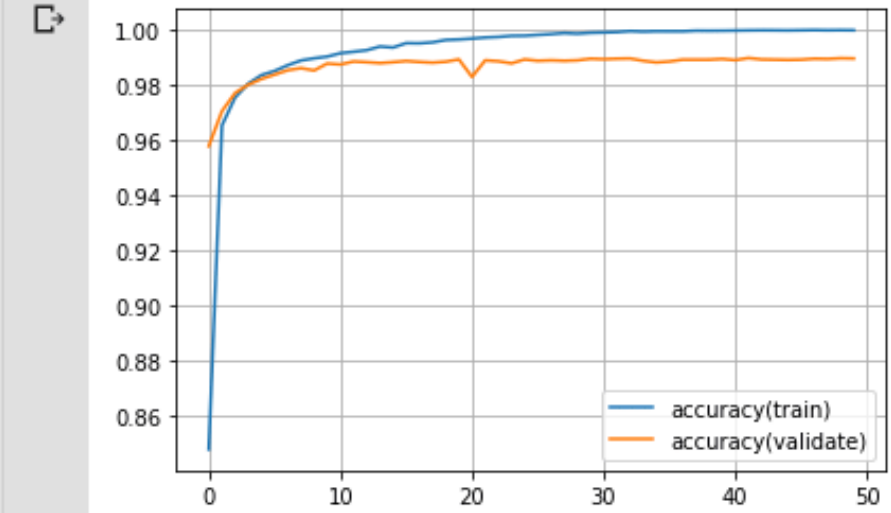
平均正解率についても同様のコードでグラフをプロットしてみます。

```
plot_graph(t_accus, v_accus, EPOCHS, 'accuracy(train)', 'accuracy(validate)')
```

学習と精度評価での平均正解率を比較するグラフをプロット

実行結果を以下に示します。

```
plot_graph(t_accus, v_accus, EPOCHS, 'accuracy(train)', 'accuracy(validate)')
```



実行結果

これら 2 つのグラフを見て、学習により得られた平均損失と平均正解率と、精度評価により得られたそれらとの間に乖離（かいり）があるに気が付いたでしょうか。平均損失に注目すると、学習で得られたものは 0.0000 に向かって進み、最後にはほぼそうなっているのに対して、精度評価で得られた方は途中から 0.0000 には近づかずに一定の値（0.0025）付近でウロウロして、その後、少し増えていくようにも見えます。

この傾向は do_train_and_validate 関数呼び出しによる画面出力からも確認できます。興味のある方は実行結果を見て、学習と精度評価によって得られた平均損失と平均正解率がどんな数値になっているのか、その変化を確認してみてください。例えば、平均損失について見てみると、学習によって得られた方は 0.0000 へ近づこうとして、精度評価で得られた方は 0.0025 の辺りをうろうろしながら、しだいに大きくなっていくのが分かるはずです。

一方、平均正解率でも同様な傾向が見えます。つまり、学習で得られた方は 100%へと着実に向かっているようです。対して、精度評価で得られた方では途中から 99%近辺で頭打ちになっているようです。

これが意味するのは、ニューラルネットワークモデルが「過学習」の状態にあるということです（といっても、99%の精度が出ているので、それほど悪くはありません。また、未知のデータである testset、そのローダーである testloader を使って、テストを行ってみたい方は validate 関数に testloader を渡して、その結果を確認してください。その実行結果もノートブックには含めてあります）。

過学習とは

ここでいう「過学習」とは、「ニューラルネットワークが、学習に使用した訓練データに過剰に適合してしまっている状態」のことです。訓練データとは人を例にすれば、試験前に参考書で何度も解いてみる「過去問」のようなものです。そればかりを解いていると、テストで同じ問題が出たときには、「これは前にやったヤツだ」となります。それと同じように、エポックを何度も繰り返していく中で、ニューラルネットワークモデルに特定のパターンのデータが何度も入力されることで、「このときにはコレ」という解答を出しやすいように、重みやバイアスが調整されてしまっていると考えられます。

このように訓練データに過剰に適合してしまうことで、未知のデータ（過去問にはなかった問題）にはうまく対応できないかもしれません（もちろん、既に述べたようにここでは精度評価でも99%近い値が出ているので、この場合はあまり気にする必要はないかもしれません）。

過学習について詳しく見るのは別の機会として、ここでは「ドロップアウト」と呼ばれる手法を使って、これを抑制してみましょう。ドロップアウトとは、「学習の際に、ニューラルネットワークを構成するノードをランダムに無効化する」というテクニックで、これにより過学習を抑制しようというものです。

PyTorch では、これを行うためのクラスとして、Dropout クラスなどが提供されているので、今回はこれを全結合層に組み込んでみましょう。

実際のクラス定義を以下に示します。

```
class Net2(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 16, 64)
        self.dropout = nn.Dropout()
        self.fc2 = nn.Linear(64, 10)
    def forward(self, x):
        x = self.conv1(x)
        x = nn.functional.relu(x)
        x = self.pool(x)
```

```
        x = self.conv2(x)
        x = self.pool(x)
        x = x.reshape(-1, 16 * 16)
        x = self.fc1(x)
        x = nn.functional.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        return x
```

ドロップアウトを組み込んだニューラルネットワーククラス Net2

強調書体で表したところが変更点です。Dropout クラスのインスタンスの生成時には、無効化するノードの割合を指定できますが、ここでは指定していません。これによりデフォルトの 0.5 が指定されたものと見なされ、半分のノードが毎回ランダムに無効化されるようになります。forward メソッドでは、入力層から隠れ層へと信号が伝播するときにドロップアウトを実行するようにしてあります。

このクラス Net2 を使って、上記と同様に do_train_and_validate 関数を呼び出してみましょう。

```
net = Net2()
criterion = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
EPOCHS = 50

history = do_train_and_validate(net, trainset, criterion, optimizer, EPOCHS)

print('Finished')
```

ドロップアウトを実装した Net2 クラスを使用して、学習と精度評価を行う

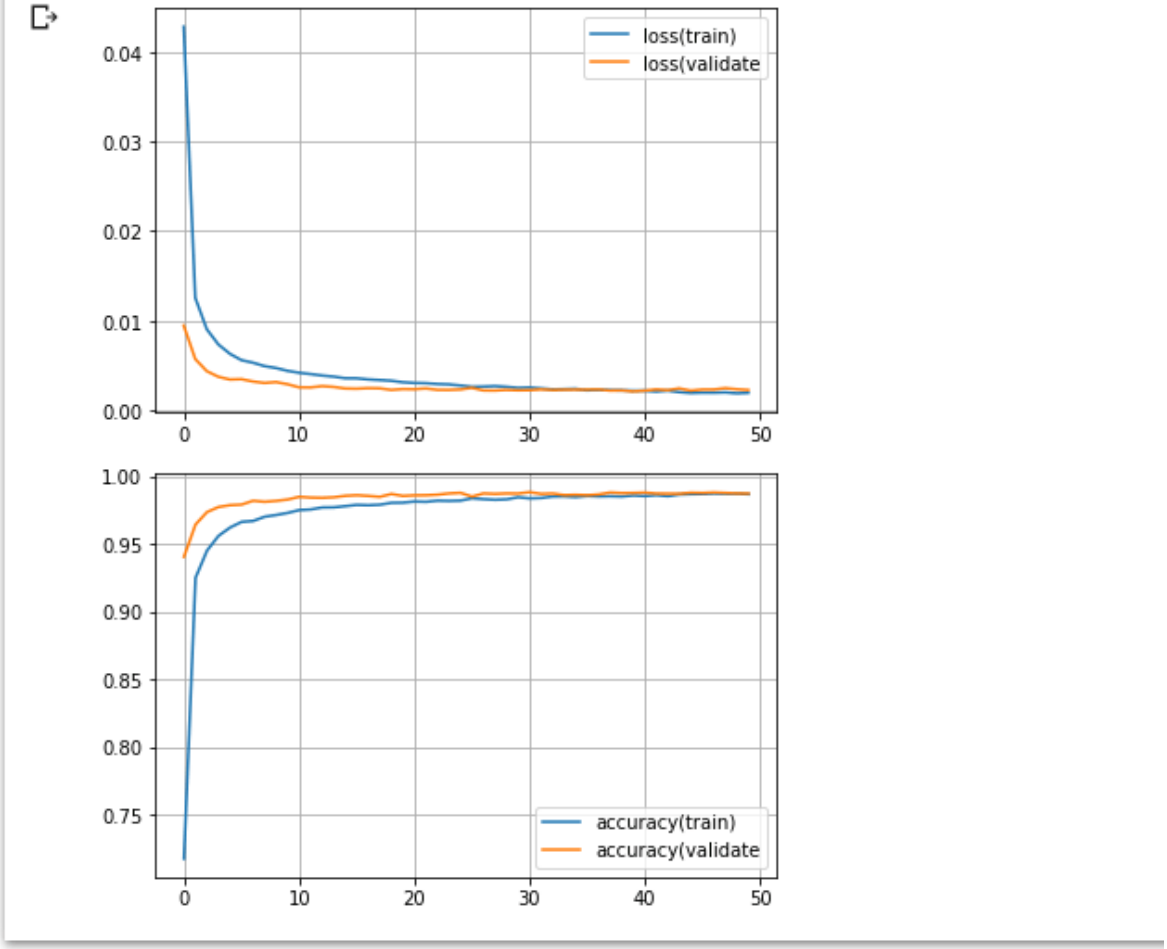
実行結果は省略して、得られた平均損失と平均正解率からグラフをプロットしてみます。

```
t_losses = history['train_loss_values']
t_accus = history['train_accuracy_values']
v_losses = history['valid_loss_values']
v_accus = history['valid_accuracy_values']

plot_graph(t_losses, v_losses, EPOCHS, 'loss(train)', 'loss(validate)')
plot_graph(t_accus, v_accus, EPOCHS, 'accuracy(train)', 'accuracy(validate)')
```

平均損失／平均正解率を比較するグラフをプロット

実行結果を以下に示します。

```
t_losses = history['train_loss_values']
t_accus = history['train_accuracy_values']
v_losses = history['valid_loss_values']
v_accus = history['valid_accuracy_values']

plot_graph(t_losses, v_losses, EPOCHS, 'loss(train)', 'loss(validate')
plot_graph(t_accus, v_accus, EPOCHS, 'accuracy(train)', 'accuracy(validate')
```



実行結果

先ほどよりも、学習と精度評価における乖離が見られなくなっている点に注目してください。学習時に得られたデータと精度評価で得られたデータとで乖離が見られないというのは、両者に対して同じ程度の損失、正解率になっているということです。つまり、訓練データに対して過剰に適合していない＝過学習の状態にはなっていないと見なせます。これがドロップアウトの効用です。過学習を抑制する方法としては、この他にも早期終了と呼ばれる手法などがありますが、ここでは説明は省略します。

今回は CNN による手書き数字の認識を行うニューラルネットワークを例に、ニューラルネットワーククラスの定義だけではなく、学習や精度評価を行うコードなどを関数として構造化し、その実行結果をグラフにプロットする方法や、訓練データの分割、過学習と呼ばれる状態を抑制する方法などについて見ました。

編集：@IT 編集部
発行：アイティメディア株式会社