**第11回　ディープラーニングとニューラルネットワークの歴史**

[株式会社電通国際情報サービス　⼩川雄太郎](https://book.mynavi.jp/manatee/list/?search_ext_col_04%5b%5d=%E6%A0%AA%E5%BC%8F%E4%BC%9A%E7%A4%BE%E9%9B%BB%E9%80%9A%E5%9B%BD%E9%9A%9B%E6%83%85%E5%A0%B1%E3%82%B5%E3%83%BC%E3%83%93%E3%82%B9%E3%80%80%E2%BC%A9%E5%B7%9D%E9%9B%84%E5%A4%AA%E9%83%8E&topics_ext_options_search=1)（著者）

SEやプログラマ、エンジニア、一般の方を対象に、少しずつ実際にプログラムを作りながら、強化学習および深層強化学習について解説していきます。

本連載をまとめ、さらに多くの記事を追加した書籍[**『つくりながら学ぶ！深層強化学習』**](https://book.mynavi.jp/manatee/books/detail/id=91985)を2018年7月に発売しました！

[](https://book.mynavi.jp/manatee/books/detail/id=91985)  
（上の書籍画像をクリックすると購入サイトに移動できます）

**はじめに**

[前回](https://book.mynavi.jp/manatee/detail/id=88997)は倒立振子課題であるCartPoleプログラムをQ学習を用いて制御する方法を解説しました。 今回は深層強化学習のカギとなる技術であるディープラーニング（深層学習）について解説します。

**ディープラーニングとは**

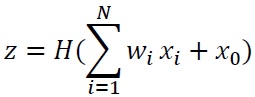
　ディープラーニングを一言で表すと、「層の深いニューラルネットワーク」です。 ですがこれだけの説明ではよく分かりませんので、人類がディープラーニングにたどり着くまでの歴史を追いながら、ひとつずつ解説していきます。

**マッカロック・ピッツモデル**

**ニューラルネットワーク**とは、人間のニューロン（神経）の活動を模倣した機械学習手法の一種です。例えば、手書き数字の画像を分類したいとします。その場合、入力データは画像であり、縦ピクセル数 × 横ピクセル数 × RGB値の入力データとなります。このデータをニューラルネットワークに入力として与えます。そして出力からその画像の数値ラベル（0-9）を読み取り、手書き数字の画像を分類します。

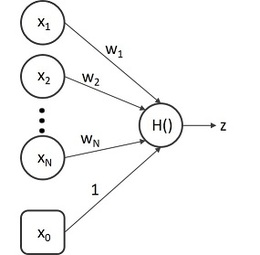
　ニューラルネットワークの原形となったモデルは、マッカロック・ピッツモデル（The McCulloch-Pitts Model）です[[1]](javascript:void(0))。**形式ニューロン**とも呼ばれ、ニューロンの活動を数理的に最も単純な形で模倣したモデルであり、1943年に発表されました。マッカロックは外科医で神経科学者であり、ピッツは数学者でした。神経科学者と数学者の共著論文でこのモデルが誕生しました。

マッカロック・ピッツモデルを数式で表すと以下の通りです。図で表すと、図11.1となります。



この式で、zは対象ニューロンの出力を表します。*xi*は対象としているニューロンとつながっているニューロン（i番目）の出力を表します。0か1の値を出力します。*w*iは対象ニューロンにつながるニューロン（i番目）との結合強度（結合の重み）です。*x*0は何らかの定数となります。 *H*()は**ヘビサイド関数**と呼ばれ、中身が正の場合は1を、負の場合は0を出力する関数です。

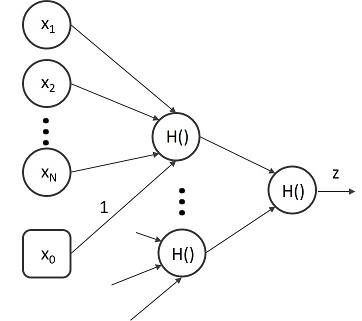
　例えば、(*w*1, *w*2) = (1, 1)のときに、*x*1=1, *x*2=1, *x*0=－1.5の場合、出力は *H*(0.5)=1となります。もし、*x*2=0だった場合には、出力は*H*(－0.5)=0となります。この例の結合パラメータの場合、マッカロック・ピッツモデルは論理演算のANDに相当します。

[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/11/McCulloch-Pitts.JPG)  
図11.1 マッカロック・ピッツモデル

　マッカロック・ピッツモデルは神経科学の知見である、**「全か無かの法則」**を数理的に表したモデルとなります。全か無かの法則は、1871年にボウディッチが発見した法則であり、ある一定上の強さの入力がないとニューロンはまったく反応せず、ある閾値を越えると突然次のニューロンへと出力を伝えるという法則です。ニューロンが次のニューロンへと出力を伝えられるのは「発火」と呼ばれる現象が起こるからです。つまりニューロンは、発火状態でない、もしくは発火状態の2状態で表されるデジタル素子として働いていることを示した生理学的知見です。

**パーセプトロン**

　マッカロック・ピッツモデルをたくさんつなげて多層にすれば、様々な入出力関係を再現でき、関数として使用できるのではないかと考えたのが、心理学者であったローゼンブラットです。彼はその考えを**パーセプトロン**と名づけ、1958年に発表しました[[2]](javascript:void(0))。図11.2となります。このようなニューロン素子をたくさんつなげたものをニューラルネットワークと呼びます。1943年のマッカロック・ピッツモデルの発表から15年も経っており、「やっとたくさんつなげるアイデアが出てきたのか、なんて進展が遅いんだ」と感じますが実際はそうではありません。

[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/11/Perceptron.JPG)  
図11.2 パーセプトロン

　いろんな入出力関係をパーセプトロンで再現するためには、ニューロン間の結合強度*w*iを都度課題に合わせて調整してあげる必要があります。ローゼンブラットは、課題ごとにこの結合強度をどう学習させれば良いのか「パーセプトロンの学習則」としてまとめました。パーセプトロンの学習則の詳細はここでは触れませんが、15年の時間がかかった理由のひとつに学習則を構築する必要性があったことが挙げられます。

　またこれまでニューロンとニューロンはシナプスという部分で機能的につながっているという点までは理解されていましたが、シナプスでニューロンが物理的に直接つながっているという「ゴルジの説」と、シナプスでは物理的にはつながっていないという「カハールの説」が共存していました。この大きな謎が電子顕微鏡の発達により、ローゼンブラットのパーセプトロンの発表直前の1950年代に明らかとなり、シナプスで間接的にニューロンがつなげていることが分かりました。また1952年には生理学者のホジキンとハクスレーによって、カハールの説に基づく神経の詳細な数理モデル（Hodgkin–Huxley model）が構築されました。

　このようなニューロンの生理学的知見の進展と、ニューロンの活動を模したパーセプトロンの出現は1960年代の世界中を熱狂させ、第1次ニューラルネットワークブームを引き起こしました。きっと当時の人たちは「人の脳と同じ仕組みで動くものが作れた。これを使えば人の脳と同じ知的なものが作れるはずだ！！」と思ったでしょう。もしかしたら、現代のディープラーニングブームと非常によく似ていたのかもしれません。ですが、その後パーセプトロンの限界を指摘する声が生まれました。

**パーセプトロンの限界**

　パーセプトロンの限界を提唱したのは、なんとローゼンブラットの高校の同級生で、数学、認知科学そしてニューラルネットワークの研究者であるミンスキーでした。

　ミンスキーは1層のパーセプトロンではXORのような入出力関係は表せないことを示します。人々はこれによりパーセプトロンへの期待を失い、第一次ニューラルネットワークブームが終了することへとつながりました。

　ですが、本質的な問題は実はこの点ではありません。パーセプトロンも多層にすればXORを表現することはでき、

*z* = *H*(*x*1+*x*2-2*H*(*x*1+*x*2-1.5)-0.5)

とすれば、*x*1, *x*2のXORはパーセプトロンで表現することができます。例えば、*x*1=1, *x*2=1の場合*z*= *H*(1+1-2-0.5)=0となります。

　本質的な問題は、ローゼンブラットの提唱したパーセプトロンの学習則では、この多層パーセプトロンの結合強度*w*iを学習できない点にありました。

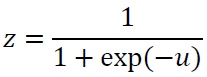
　それでは「学習則は脳の知見を元に学習すれば良いのでは？」と直感的に思いますが、この時点では脳内でニューロンの結合強度（シナプス間強度）が変化しているのかどうか明らかではなかったのです。仮説レベルでは**ヘッブ則**（Hebb's rule）と呼ばれる仮説が1949年に心理学者ヘッブに提唱されていましたが（マッカロックピッツモデルより後である点に注意）、実際に確認はされてはいませんでした（1973年BlissとLømoによって初めて確認）。加えて、ヘッブ則は多層パーセプトロンの学習則として適用できるものではありませんでした。

**バックプロパゲーションの出現**

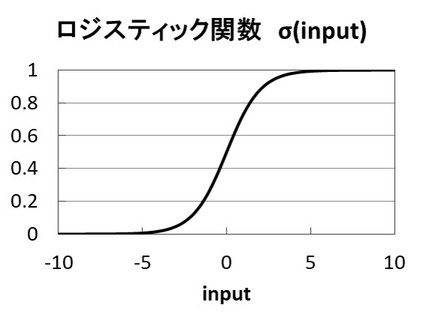
　第1次ニューラルネットワークブームが去り冬の時代を迎えていたニューラルネットワークをもう一度盛り上げたのが、1986年にラメルハートらが提唱した**バックプロパゲーション（誤差逆伝搬法）**です[[3]](javascript:void(0))。ラメルハートらと書きましたが、この「ら」が後に非常に重要となります。正確には、ラメルハート、ヒントン、ウィリアウムズの3人による共著論文となっています。

　パーセプトロンではニューロンの特性である「全か無かの法則」に従い、出力をヘビサイド関数で表していました。再掲しますと、ヘビサイド関数*H*()は、中身が正の場合は1を、負の場合は0を出力する関数です。この関数は入力0を境に出力が急激に0から1に変化するため、数学的に取り扱いが難しいものとなっていました。

　そこでラメルハートらはヘビサイド関数の使用はやめて、**ロジスティック関数**（シグモイド関数）を使うことにしました。ロジスティック関数は以下の式で表されます。



*u*は入力を示し、ロジスティック関数を図で表すと以下の図11.3となります。図のとおり、入力0付近で滑らかに出力が0から1へと変化し、変化の途中では0.5など、中途半端な値を出力します。

[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/11/logisticfunction.JPG)  
図11.3 ロジスティック関数（シグモイド関数）

　もはや0、1以外を出力する時点でニューロンの発火をベースとした生理学的知見である「全か無かの法則」からは離れてしまいます（ただし単一神経ではなく、神経細胞集団や、単一神経の確率的な挙動として考えた場合にはロジスティック関数でも生理学的に妥当とも考えられます）。

　ヘビサイド関数からロジスティック関数に変更したことで、数学的に取り扱いやすくなり、微分操作ができるようになりました。

　微分操作ができると何が嬉しいか解説します。適当な結合強度をニューラルネットワークに設定して、入力を与え出力を計算したときに、期待した出力と異なる場合に、その誤差とシグモイド関数の微分を用いれば、非線形問題を解く多層ニューラルネットワークの結合強度が学習可能となるのです。この学習法則をバックプロパゲーション（誤差逆伝搬法）と呼びます。ここではその数理的な詳細は触れません。

　なお、ラメルハートら以前に理化学研究所の甘利先生が1967年にバックプロパゲーションの概念を提唱していましたが、世界的にはラメルハートらの論文がスタンダードになってしまったという経緯などもあります。

　ロジスティック関数の導入とバックプロパゲーションによって、パーセプトロンの限界であった非線形問題の学習が実現され、「これを使えば人の脳と同じ知的なものが作れるはずだ！！」と第2次ニューラルネットワークブームが到来します。

　しかしながらこれまたブームが終焉します。複雑な入出力関係を学習させるには1層のニューロンの数を増やすか、層をつなげるしかありません。ニューラルネットワークを何層もつなげることを、層を深くする（deepにする）と呼びます。1層あたりのニューロンを増やす方法はあまり効率が良くなく、層を深くすることが重要でした。しかし、深いニューラルネットワークはラメルハートらの手法ではうまく結合強度を学習することができませんでした。その理由は勾配消失・爆発問題と呼ばれる現象が発生するためですが、ここでは詳細は触れません。

　結局、「複雑な入出力関係はニューラルネットワークでは実現できないのか・・・」となり、第2次ニューラルネットワークブームが去ってしまいます。

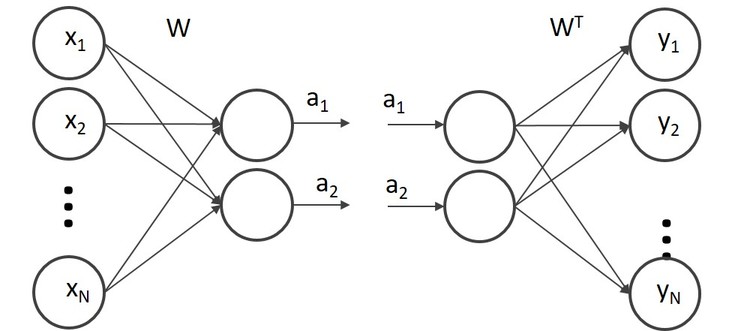
**ディープラーニングの誕生**

　深い層を持つニューラルネットワークでは結合強度がうまく学習できない問題に対して、挑戦しつづけたのがヒントンです。どこかで聞いた名前ですが、そう、バックプロパゲーションの提唱でラメルハートらの「ら」に含まれていた第二著者のヒントンです。彼はロジスティック関数の導入とバックプロパゲーションの提唱後も、深い層を持つニューラルネットワークの実現に挑戦し続けていました。

　ヒントンは2006年に**「ディープラーニング」**と呼ばれるアイデアでこの深い層を持つニューラルネットワークを実現させます[[4]](javascript:void(0))。彼のアイデアは、深い層を持つニューラルネットワークの結合強度をランダムな初期値から全て一気に学習させるのではなく、各層ごとに先にある程度の初期値を当たりをつけて与える作戦でした。各層ごとの結合強度の初期値を求めるため、彼はニューラルネットワークを1層ずつ切り出し、層への入力情報と層の出力情報がうまく情報圧縮できるように結合強度を与えれば良いと考えました。

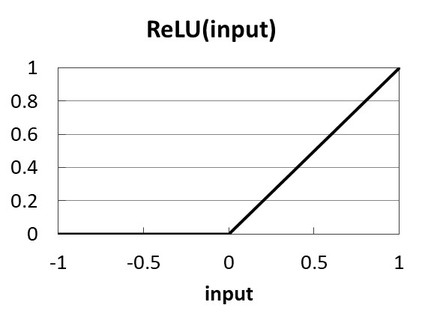
　例えば図11.4のように、N次元の入力*x*を結合強度行列Wのネットワークに与え、例えば2次元の出力*a*1と*a*2を得ます。その後、Wの転置（Transpose）を結合強度として持つネットワークに、得られた2次元の出力*a*1と*a*2を入力として与え、N次元の出力*y*を求めます。通常であれば、*x*と*y*はまったく異なる値になりますが、これができるだけ同じになるよう結合強度行列Wを学習させてあげます。この考え方は**オートエンコーダ**と呼ばれます。

　図11.4のオートエンコーダはN次元のデータを一度2次元に圧縮し、再度復元していることになります。つまり図11.4の左側の結合強度行列Wのニューラルネットワークは、N次元の入力を2次元にうまく情報圧縮するネットワークとなります。こうして各層が情報圧縮の機能を持つ状態の結合強度から学習を始めれば、多層ニューラルネットワークがうまく学習できることをヒントンらは提唱しました。なおオートエンコーダの結合行列の学習には制限ボルツマンマシンという手法が使われました。

[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/11/autoencoder.JPG)  
図11.4 オートエンコーダー

　2012年の画像認識コンテスト「ILSVRC（ImageNet Large Scale Visual Recognition Challenge）」で、ヒントンらはディープラーニングを用いた画像認識システムを提案し、歴代の優勝記録を大幅に更新する認識精度で優勝しました。この出来事がディープラーニングに注目が集まるきっかけとなります。

　現在のディープラーニングでは、初期のヒントンのアイデアであった、オートエンコーダや制限ボルツマンマシンは使用されておらず、主にロジスティック関数を図11.5に示す**ReLU**（relu rectified linear unit）などに置き換えて、バックプロパゲーションで実現されています。

[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/11/ReLU.JPG)  
図11.5 ReLU

　ヒントンらが層を重ねたニューラルネットワークであるディープラーニングを提案し、さらに改良が加えられてディープラーニングが取扱いやすくなり、2018年現在は第3次ニューラルネットワークブームや第3次人工知能ブームと呼ばれています。

　振り返ってみると結果的には、ニューロンへの入力の総和を出力へと変換する関数（活性化関数）がヘビサイド関数からシグモイド関数に代わり、そしてReLUなどへと変わっていき、それに付随してニューラルネットワークの可能性が広がっていったように感じます。

　このような歴史を経てニューラルネットワークは進歩し、ディープラーニングの出現により複雑な入出力関係も表現できるようになって、2018年現在のAIブームが到来しました。

**まとめ**

　今回は人類がディープラーニングにたどり着くまでの歴史を追いながら、ディープラーニングについて解説しました。次回はPyTorchと呼ばれるライブラリを利用してディープラーニングを実装する手法を解説します。

**引用**

[1] McCulloch, Warren S., and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." The bulletin of mathematical biophysics 5.4 (1943): 115-133.

[2] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.

[3] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." nature 323.6088 (1986): 533.

[4] Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." science 313.5786 (2006): 504-507.

**第12回　PyTorchによるディープラーニング実装入門（１）**

[株式会社電通国際情報サービス　⼩川雄太郎](https://book.mynavi.jp/manatee/list/?search_ext_col_04%5b%5d=%E6%A0%AA%E5%BC%8F%E4%BC%9A%E7%A4%BE%E9%9B%BB%E9%80%9A%E5%9B%BD%E9%9A%9B%E6%83%85%E5%A0%B1%E3%82%B5%E3%83%BC%E3%83%93%E3%82%B9%E3%80%80%E2%BC%A9%E5%B7%9D%E9%9B%84%E5%A4%AA%E9%83%8E&topics_ext_options_search=1)（著者）

SEやプログラマ、エンジニア、一般の方を対象に、少しずつ実際にプログラムを作りながら、強化学習および深層強化学習について解説していきます。

本連載をまとめ、さらに多くの記事を追加した書籍[**『つくりながら学ぶ！深層強化学習』**](https://book.mynavi.jp/manatee/books/detail/id=91985)を2018年7月に発売しました！

[](https://book.mynavi.jp/manatee/books/detail/id=91985)  
（上の書籍画像をクリックすると購入サイトに移動できます）

**はじめに**

[前回](https://book.mynavi.jp/manatee/detail/id=89172)は人類がディープラーニングに至るまでの技術の変遷を追いながら、ニューラルネットワークの歴史を外観しました。 今回は実際にディープラーニングの実装を解説します。ディープラーニングの実装にはPyTorchと呼ばれるライブラリを使用し、手書き数字の画像データ（MNIST）の分類を実行します。今回と次回の2回に分けて解説を行ないます。

**PyTorchとは**

　ディープラーニングを実装する際にはディープラーニング用のライブラリを使用するのが一般的です。ライブラリとしてはCaffe、TensorFlow、Keras、Chainerなどが有名です。KerasはTensorFlowやCaffeを使用しやすくするwrapperとなります。Chainerは日本のPreferred Networks社が開発したライブラリです。

　PyTorchはこれらのライブラリよりも後発で、最近生まれたディープラーニング用ライブラリです。元々はTorch7と呼ばれるLua言語で書かれたライブラリでした。このTorch7とPreferred Networks社のChainerをベースに2017年2月に作られたPython用ライブラリがPyTorchとなります。Chainerをforkして作られました。

　PyTorchおよびChainerの利点はDefine by Run（動的計算グラフ）と呼ばれる特徴です。Define by Runは入力データのサイズや次元数に合わせてニューラルネットワークの形や計算方法を変更することができます。例えばデータAは入力次元が4なのにデータBは入力次元が5といった場合に、入力次元に応じてニューラルネットワークの計算を変えるように実装することができます。このような入力データの次元がデータごとに異なる状況は自然言語処理ではとくに頻繁に発生します。

　一方でTensorFlowの特徴はDefine and Run（静的計算グラフ）と呼ばれます。Define and Runではニューラルネットワークの計算方法をはじめに決めてしまうため、入力データの次元がデータごとに異なる状況に対応しづらいという特徴があります。なおPyTochはDefine and Runでコードを書くこともできます。

　PyTorchは英語圏生まれの最新ライブラリであり、大変注目を集めています（TensorFlowにもEagerというDefine by Runバージョンが作られましたが）。最近の最新論文の内容をPyTorchで実装して発表する研究者が多く、PyTorchには代表的なディープラーニング手法の実装例がほとんどすぐに手に入るという利点があります。

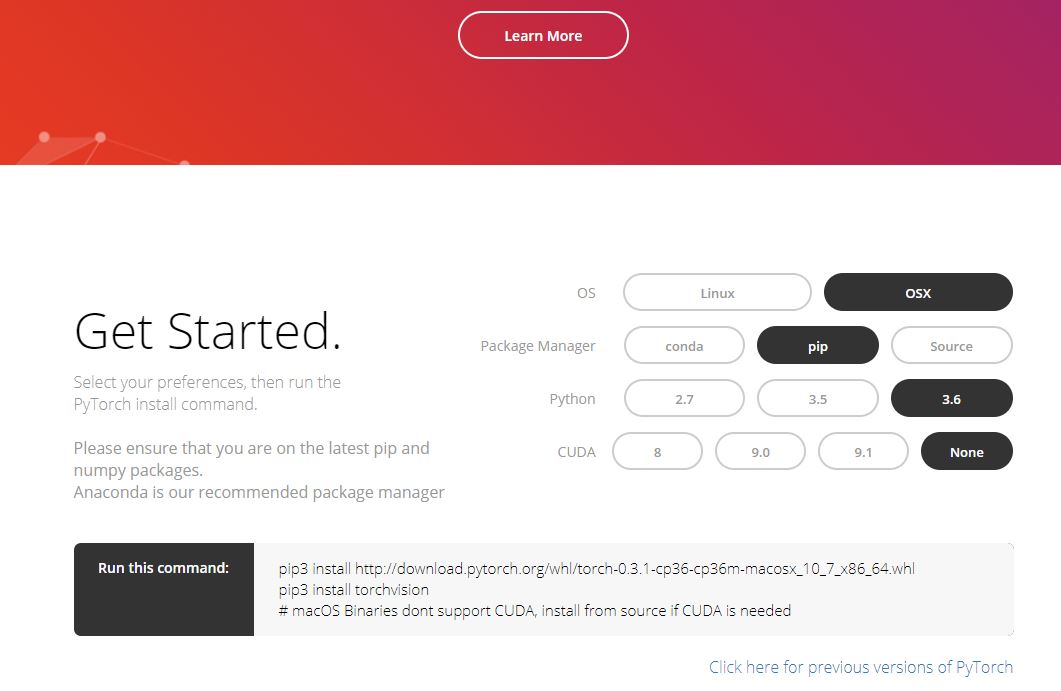
　以上説明したように、PyTorchは「Define by Runである」、「実装例が豊富にある」という利点を持ち、今後ますます注目が集まるライブラリです。そこで本連載ではPyTorchを使用して深層強化学習を実装します。

　今回はPyTorchを使用して、手書き数字の画像データ（MNIST）を分類するディープラーニングを実装します。

**PyTorchの実行環境整備**

　PyTorchは基本的にはmacOSとLinuxをサポートしています。公式サイト[[1]](javascript:void(0))にアクセスすると自分の実行環境を選択する画面があり、インストール用のコマンドが表示されます（図12.1）。ここでCUDAとはNVIDIAのGPUを使用するための環境です。

<http://pytorch.org/>

[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/12/pytorch_1.JPG)  
図12.1 PyTorchの公式サイト

　OSがWindowsの場合には以下のサイト[[2]](javascript:void(0))の解説を元に、Anaconda上でインストールすることができます。

<https://github.com/peterjc123/pytorch-scripts>

　この先はWindows 10のCUDA未使用という環境で解説を進めます。

　Anacondaを立ち上げ、前回まで使用していた仮想環境でターミナルを開きます。そして以下のコマンドを１つずつ実行して、PyTorchとPyTorchと一緒に使用するtorchvisionライブラリをインストールしてください。torchvisionは主に画像データをPyTorchで扱いやすくする関数が組み込まれたライブラリです。

|  |
| --- |
| conda install -c peterjc123 pytorch-cpu  pip install torchvision |

　以上でPyTorchを使用する環境の構築は終了です。ターミナルからJupyter Notebookを開き、Pythonの新規プログラムを作成して

|  |
| --- |
| import torch |

を実行して、エラーが出ないことを確認してください（importする名前はpytorchではなくtorchである点に注意してください）。

**PyTorchでMNIST**

　これからMNIST（Modified National Institute of Standards and Technology）と呼ばれる機械学習で最も一般的なチュートリアル課題をPyTorchで実装します。MNISTはアメリカ統計局職員および高校生が書いた手書き数字の画像データです。訓練データが6万枚、テストデータが1万枚提供されています。

　今回実現したいことは、テストデータの画像をニューラルネットワークに入力したときに、その画像が0-9のどの数字なのかを分類するディープラーニングを構築することです。

　構築の流れは主に、学習フェイズと推論フェイズに分かれます。学習フェイズでは訓練データを基にニューラルネットワークの素子間の結合強度を学習させます。推論フェイズではテストデータの手書き数字画像から、その数字を分類します。ここで訓練データとはニューラルネットワークの各ニューロン素子の間の結合強度（結合パラメータ）を学習させるためのデータを示し、テストデータとは学習したニューラルネットワークの精度を確かめるためのデータを示します。

**MNISTデータの取得**

　Anacondaを立ち上げ、前回まで使用していた仮想環境でターミナルを開きます。そして以下のコマンドを実行して、scikit-learnと呼ばれるライブラリをインストールします。

　scikit-learnは機械学習用のライブラリです。今回はMNISTの画像をダウンロードとダウンロードしたデータを訓練データとテストデータに分けるためにscikit-learnを使用します。

　なお、PyTorchとtorchvisionの関数でMNISTのデータをダウンロードすることもできます。ですがその方法ではPyTorch特有のデータの取り扱い方を理解しづらいです。そのため今回はscikit-learnを使用してMNISTをダウンロードし、PyTorch用にデータを変換するところから実装します。

　以下のコマンドをターミナルで実行し、scikit-learnをインストールしてください。

|  |
| --- |
| conda install scikit-learn |

　AnacondaからJupyter Notebookの新規Pythonファイルを作成し、以下のコマンドを実行して下さい。

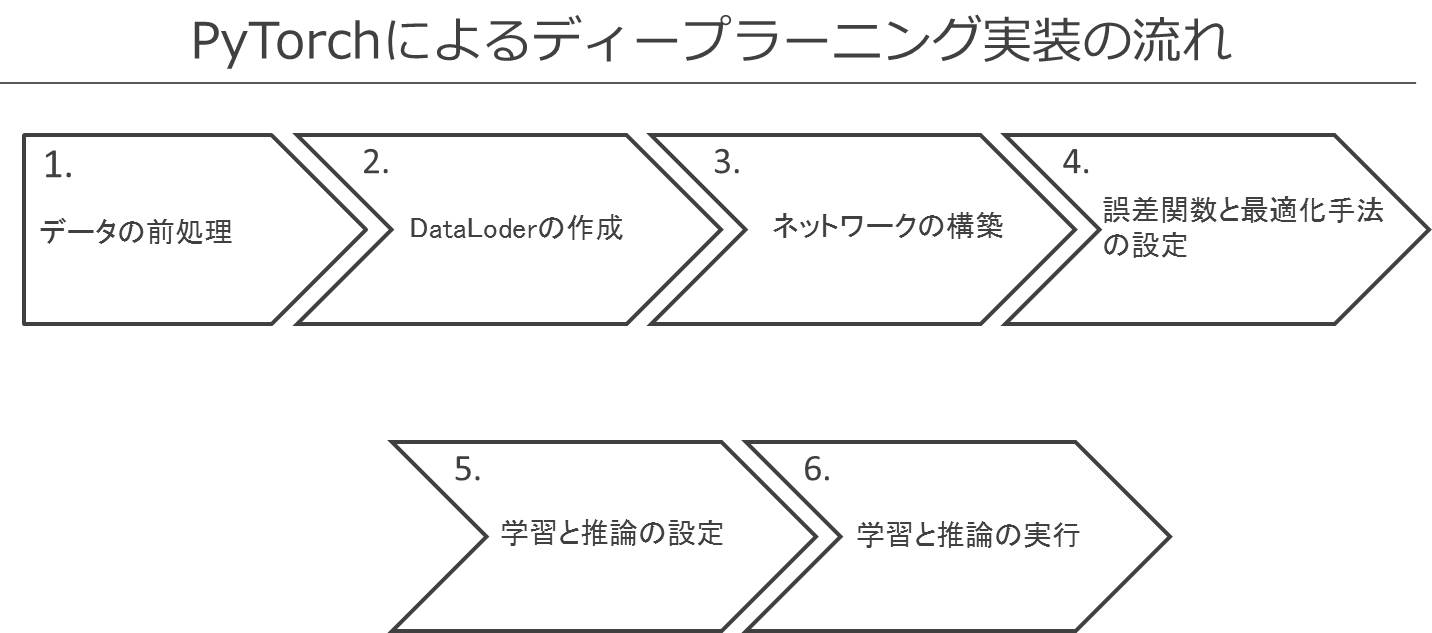
|  |
| --- |
| # 手書き数字の画像データMNISTをダウンロード    from sklearn.datasets import fetch\_mldata    mnist = fetch\_mldata('MNIST original', data\_home=".")  # data\_homeは保存先を指定します |

これで変数mnistにデータが格納されました。fetch\_mldata()は手書き数字の画像データとラベルデータをダウンロードするのですが、ときおりダウンロード先のサーバーの都合でうまく動かない場合があります。その場合何回か実行しているとうまくいくので、繰り返してみてください。

**PyTorchによるディープラーニングの実装**

　ここからPyTorchによるディープラーニングを実装します。今回は最低限の実装であるミニマムバージョンで解説します。PyTorchによるディープラーニングの実装は次の6ステップで行われます。

1. データの前処理
2. DataLoderの作成
3. ネットワークの構築
4. 誤差関数と最適化手法の設定
5. 学習と推論の設定
6. 学習と推論の実行

[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/12/pytorch_flow.JPG)  
図12.2 PyTorchの実装フロー

**1. データの前処理**

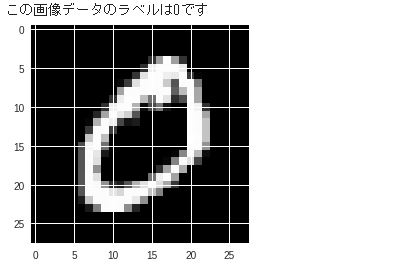
　データの前処理では、データをニューラルネットワークに投入できるように加工します。今回のMNISTデータの場合は、まずダウンロードしたデータを画像Xとラベルy（0-9）に分けて格納します。画像データはグレースケールの0-255の数値で表現されているので、255で割って0-1になるように正規化します。

|  |
| --- |
| # 1. データの前処理（画像データとラベルに分割し、正規化）    X = mnist.data / 255  # 0-255を0-1に正規化  y = mnist.target |

　ここで1つ目の手書き文字の画像とラベルを可視化してみましょう。

|  |
| --- |
| # MNISTのデータの1つ目を可視化する    import matplotlib.pyplot as plt  % matplotlib inline    plt.imshow(X[0].reshape(28, 28), cmap='gray')  print("この画像データのラベルは{:.0f}です".format(y[0])) |

すると以下の図12.3の結果が出力されます。変数Xには縦28×横28ピクセルの784の要素を持つNumpy形式のベクトルが7万画像分格納されています。

[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/12/mnist2.JPG)  
図12.3 MNISTの1つ目のデータを可視化

次回は2. DataLoderの作成から説明いたします。

**引用**

[1] PyTorch公式サイト　[http://pytorch.org/](http://pytorch.org/" \t "_blank)

[2] PyTorch for Windows <https://github.com/peterjc123/pytorch-scripts>

**第13回　PyTorchによるディープラーニング実装入門（２）**

[株式会社電通国際情報サービス　⼩川雄太郎](https://book.mynavi.jp/manatee/list/?search_ext_col_04%5b%5d=%E6%A0%AA%E5%BC%8F%E4%BC%9A%E7%A4%BE%E9%9B%BB%E9%80%9A%E5%9B%BD%E9%9A%9B%E6%83%85%E5%A0%B1%E3%82%B5%E3%83%BC%E3%83%93%E3%82%B9%E3%80%80%E2%BC%A9%E5%B7%9D%E9%9B%84%E5%A4%AA%E9%83%8E&topics_ext_options_search=1)（著者）

SEやプログラマ、エンジニア、一般の方を対象に、少しずつ実際にプログラムを作りながら、強化学習および深層強化学習について解説していきます。

本連載をまとめ、さらに多くの記事を追加した書籍[**『つくりながら学ぶ！深層強化学習』**](https://book.mynavi.jp/manatee/books/detail/id=91985)を2018年7月に発売しました！

[](https://book.mynavi.jp/manatee/books/detail/id=91985)  
（上の書籍画像をクリックすると購入サイトに移動できます）

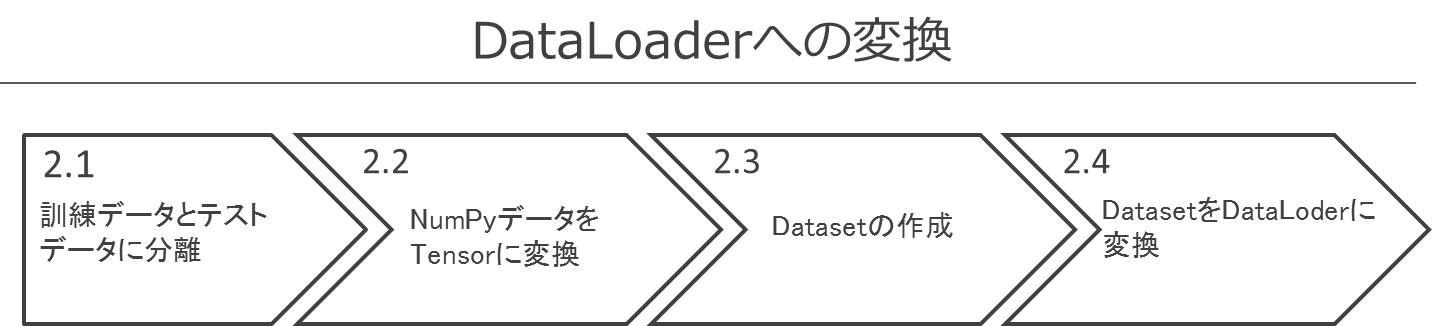
前回に引き続き、実際にディープラーニングの実装を解説します。PyTorchと呼ばれるライブラリを使⽤し、⼿書き数字の画像データ（MNIST）の分類を実⾏します。

**2. DataLoderの作成**

　続いて正規化したMNISTデータをPyTorchのニューラルネットワークで扱えるDataLoaderという変数へと変換します。

　DataLoaderへの変換は以下の4つの手続きからなります。

2.1. 訓練データとテストデータに分離  
2.2. NumPyデータをTensorに変換  
2.3. Datasetの作成  
2.4. DatasetをDataLoderに変換

[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/12/dataloader_flow.JPG)  
図12.4 DataLoderへの変換フロー

　この変換部分のコードは以下の通りとなります。

|  |
| --- |
| # 2. DataLoderの作成    import torch  from torch.utils.data import TensorDataset, DataLoader    # 2.1 データを訓練とテストに分割（6:1）  from sklearn.model\_selection import train\_test\_split  X\_train, X\_test, y\_train, y\_test = train\_test\_split(      X, y, test\_size=1/7, random\_state=0)    # 2.2 データをPyTorchのTensorに変換  X\_train = torch.Tensor(X\_train)  X\_test = torch.Tensor(X\_test)  y\_train = torch.LongTensor(y\_train)  y\_test = torch.LongTensor(y\_test)    # 2.3 データとラベルをセットにしたDatasetを作成  ds\_train = TensorDataset(X\_train, y\_train)  ds\_test = TensorDataset(X\_test, y\_test)    # 2.4 データセットのミニバッチサイズを指定した、Dataloaderを作成  # Chainerのiterators.SerialIteratorと似ている  loader\_train = DataLoader(ds\_train, batch\_size=64, shuffle=True)  loader\_test = DataLoader(ds\_test, batch\_size=, shuffle=False) |

　2.1の訓練データとテストデータへの分割はscikit-learnのtrain\_test\_split関数を用いて行います。今回はデータを訓練データ6万件とテストデータ1万件に分割しています。

　2.2ではNumpyデータをPytorchで扱える型の変数に変換します。torch.Tensorと呼ばれるPyTorch版のNumPyに変換します。ラベルのような整数データの場合はtorch.LongTensorを使用します。Tensor（テンソル）という言葉が聞きなれないですが、難しい概念ではありません。数値が1つだけの場合はスカラー、数値が1次元に並んでいるものはベクトル、2次元に並んでいるものは行列と呼ばれます。こうした多次元の数値表現のことをテンソルと呼びます（数学的には厳密な定義がありますが今回は気にする必要はありません）。

　2.3ではTensorに変換された画像データとラベルデータをTensorDatasetで一組にしています。データのセットなので画像とラベルを組にしたものとなります。訓練データとテストデータでそれぞれ作成します。

　2.4ではTensorDatasetを学習と推論がしやすいようにDataLoaderと呼ばれる形式に変換しています。DataLoaderではバッチサイズを指定します。バッチサイズとは、ニューラルネットワークの結合パラメータを学習する際に一度に使用するデータの数を決めます。データ全部を使用する方法をバッチ学習、一部を使用する方法をミニバッチ学習、データを1つずつ使用する方法をオンライン学習と呼びます。ミニバッチ学習を使用することが一般的です。テストデータはミニバッチにする必要はなさそうですが、データがたくさんあった場合に分散処理しやすいようにこちらもミニバッチにすることが多いです。さらにDataLoaderではデータをシャッフルするかどうか設定します。現在のDatasetは0-9まできれいに並んで格納されているのですが、訓練データに対してランダムな順番で学習を実行したいです。そこで訓練データのDataLoaderはデータをシャッフルさせます。テストデータは正答率を求めるだけなので、シャッフルの必要はありません。

　※Chainerでは、iterators.SerialIterator()でデータをまとめていましたが、PyTorchではDataLoaderを使用します。

**3. ネットワークの構築**

　続いて使用するニューラルネットワークを規定します。ここでは一番簡単な書き方を紹介します。これはTensorFlowとKerasライブラリを使用して書く方法と同じ書き方で、Define and Run形式の書き方となります。本記事の最後ではDefine by Run形式の書き方を紹介します。

|  |
| --- |
| # 3. ネットワークの構築  # Keras風の書き方 Define and Run    from torch import nn    model = nn.Sequential()  model.add\_module('fc1', nn.Linear(28\*28, 100))  model.add\_module('relu1', nn.ReLU())  model.add\_module('fc2', nn.Linear(100, 100))  model.add\_module('relu2', nn.ReLU())  model.add\_module('fc3', nn.Linear(100, 10))    print(model) |

　fc1は、28\*28=784の入力を持ち、それを100個のニューロンに出力する層です。入力の784素子と、出力の100素子はすべて結合しています（Fully Connected層と呼びます）。relu1は、fc1の100個のニューロンの出力をReLUによって変換します。ReLUは入力が負のときは0を、正のときはそのまま入力を出力するユニットです。同様に、100個の入出力から成るニューロン層fc2を加え、relu2でReLUで変換します。そして最後にfc3で0-9のラベルに対応する10個のニューロンに出力を与えます。各層のニューロンの数、例えば100であったり、層の数は試行錯誤しながら決めることになります。今回の構築したネットワークですと、入力層、中間層fc1、中間層fc2、中間層fc3、出力層というディープニューラルネットワークを構築しています。

**4. 誤差関数と最適化手法の設定**

　続いて、ネットワークの誤差関数と学習の手法を設定します。

　誤差関数とはニューラルネットワークの出力と実際の正解である望ましい出力との誤差をどのような関数で計算するのかを決めます。今回のような分類問題では誤差関数にはクロスエントロピー誤差を使用します。

　最適化手法とはニューラルネットワークの結合パラメータをどのような手法で更新学習するのかを決めます。今回は勾配法のなかでもAdamと呼ばれるアルゴリズムを使用するように設定します。以下のコード内のlrは学習率を示します。

|  |
| --- |
| # 4. 誤差関数と最適化手法の設定    from torch import optim    # 誤差関数の設定  loss\_fn = nn.CrossEntropyLoss()  # 変数名にはcriterionも使われる    # 重みを学習する際の最適化手法の選択  optimizer = optim.Adam(model.parameters(), lr=0.01) |

**5. 学習と推論の設定**

　続いて、学習と推論での動作を設定します。学習時と推論時の動作をそれぞれ設定します。

　学習では訓練データを入力して出力を求めます。その後、出力と正解との誤差を誤差関数に従って計算し、誤差をバックプロパゲーションして、最後に結合パラメータを更新学習させます。

　推論ではテストデータを入力して出力を求め、実際の正解と一致した割合を求めます。

　注意点がいくつかあります。まず学習時にはmodel.train()を実行し、ネットワークを学習モードにします。推論時にはmodel.eval()を実行して、推論モードに切り替えます。これは今回のネットワークでは関係ないのですが、ドロップアウトやBatch Normalization（バッチノーマライゼーション）といった手法を使用した場合に重要となります。さらに、ニューラルネットワークに入力する前にデータをVariable()で変換します。PyTorchではこのVariable関数を使用することで、変数を微分可能なものに変換します。また学習時にはoptimizer.zero\_grad()で毎回バックプロパゲーションの初期値をリセットしてあげます。

　関数trainとtestの引数であるepochとはデータを一通り使用する1試行のことを意味します。ミニバッチ学習なので、1epochの間に少しずつデータを使用して学習を進め、全データを一通り使用したら1epoch終了となります。ディープラーニングをきちんと行う場合には、1epochごとに学習と推論時の誤差関数の値を出力・保存し、学習の進捗状況を確認することになります。今回はミニマムバージョンなので、この部分は省略しています。また学習過程を可視化するのにTensorFlowのTensorBoardをwrapしたtensorboardXというライブラリもあります。

※Chainerでは学習と推論の設定に学習はtraining.Trainer()を、推論はtrainer.extend(extensions.Evaluator())を使用して設定できましたが、PyTorchでは自分で書きます。

|  |
| --- |
| # 5. 学習と推論の設定  # 5-1. 学習1回でやることを定義します  # Chainerのtraining.Trainer()に対応するものはない    from torch.autograd import Variable      def train(epoch):      model.train()  # ネットワークを学習モードに切り替える        # データローダーから1ミニバッチずつ取り出して計算する      for data, target in loader\_train:          data, target = Variable(data), Variable(target)  # 微分可能に変換          optimizer.zero\_grad()  # 一度計算された勾配結果を0にリセット            output = model(data)  # 入力dataをinputし、出力を求める          loss = loss\_fn(output, target)  # 出力と訓練データの正解との誤差を求める          loss.backward()  # 誤差のバックプロパゲーションを求める          optimizer.step()  # バックプロパゲーションの値で重みを更新する        print("epoch{}：終了\n".format(epoch)) |
| # 5. 学習と推論の設定  # 5-2. 推論1回でやることを定義します  # Chainerのtrainer.extend(extensions.Evaluator())に対応するものはない      def test():      model.eval()  # ネットワークを推論モードに切り替える      correct = 0        # データローダーから1ミニバッチずつ取り出して計算する      for data, target in loader\_test:          data, target = Variable(data), Variable(target)  # 微分可能に変換          output = model(data)  # 入力dataをinputし、出力を求める            # 推論する          pred = output.data.max(1, keepdim=True)[1]  # 出力ラベルを求める          correct += pred.eq(target.data.view\_as(pred)).sum()  # 正解と一緒だったらカウントアップ        # 正解率を出力      data\_num = len(loader\_test.dataset)  # データの総数      print('\nテストデータの正解率: {}/{} ({:.0f}%)\n'.format(correct,                                                     data\_num, 100. \* correct / data\_num)) |

**6. 学習と推論の実行**

　最後にネットワークの結合パラメータの学習と、学習後にテストデータで精度を求めます。ですがその前に、学習せずにテストデータを推論をしてみましょう。以下のコードを実行してください。

|  |
| --- |
| # 学習なしにテストデータで推論してみよう  test() |

上記セルを実行すると、

|  |
| --- |
| テストデータの正解率: 1051/10000 (11%) |

など、およそ10%程度の正解率が出力されます。まだネットワークが学習しておらず、数字が10種類あるのでランダムに選んだ場合とほぼ同じ結果です。

　続いて、ニューラルネットワークの結合パラメータを学習させ、再度テストデータで推論してみましょう。今回は6万件の訓練データに対して3epoch学習させます。

|  |
| --- |
| # 6. 学習と推論の実行  for epoch in range(3):      train(epoch)    test() |

すると、以下のような出力が得られます。

|  |
| --- |
| epoch0：終了  epoch1：終了  epoch2：終了  テストセットの正解率: 9616/10000 (96%) |

　学習後には正答率がおよそ95%程度となり、手書き数字をおおよそ正しく識別できるようになりました。

　特定の画像データを推論したい場合は以下のような実装します。なお推論用に作成した関数test()では、出力ラベルを求める際に、pred = output.data.max(1, keepdim=True)[1]としていましたが、今回はpred = output.data.max(0, keepdim=True)[1] と、引数の1が0に変わっています。これはtest()は複数の画像からなるミニバッチを推論していたのに対して、以下のセルでは1つの画像を推論しているからです。

|  |
| --- |
| # 例えば2018番目の画像データを推論してみる    index = 2018    model.eval()  # ネットワークを推論モードに切り替える  data = Variable(X\_test[index])  output = model(data)  # 入力dataをinputし、出力を求める    pred = output.data.max(0, keepdim=True)[1]  # 出力ラベルを求める    print("予測結果は{}".format(pred))      X\_test\_show = (X\_test[index]).numpy()  plt.imshow(X\_test\_show.reshape(28, 28), cmap='gray')  print("この画像データの正解ラベルは{:.0f}です".format(y\_test[index])) |

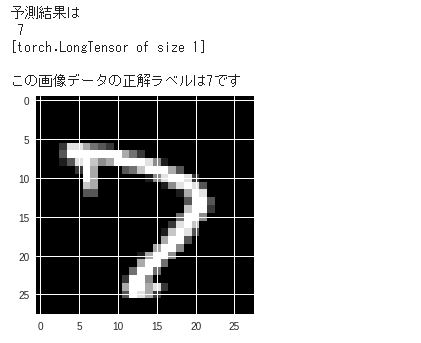
[](https://book.mynavi.jp/files/user/manatee/rensai/019_kyoukagakusyuu/12/mnist3.JPG)  
図12.5 学習後の出力結果

図12.3 学習後の出力結果

ニューラルネットワークの予想が7で、正解も7となっています。

以上が、PyTorchによるディープラーニングのミニマムバージョンの実装となります。

**PyTorchの使い方の補足**

上記の例ではネットワークの構築をDefine and Runで実装しましたが、ChainerのようにDefine by Runで実装するには以下のように変更します。

|  |
| --- |
| # 3. ネットワークの構築  # ニューラルネットワークの設定（Chainer風の書き方）  import torch.nn as nn  import torch.nn.functional as F      class Net(nn.Module):        def \_\_init\_\_(self, n\_in, n\_mid, n\_out):          super(Net, self).\_\_init\_\_()          self.fc1 = nn.Linear(n\_in, n\_mid)  # Chainerと異なり、Noneは受けつけない。          self.fc2 = nn.Linear(n\_mid, n\_mid)          self.fc3 = nn.Linear(n\_mid, n\_out)        def forward(self, x):          # 入力xに合わせてforwardの計算を変えられる＝Define by Run          h1 = F.relu(self.fc1(x))          h2 = F.relu(self.fc2(h1))          output = self.fc3(h2)          return output      model = Net(n\_in=28\*28, n\_mid=100, n\_out=10)  # ネットワークのオブジェクトを生成  print(model) |

入力データxに応じてforward計算式を変えられる点がDefine by Runの特徴です。

**まとめ**

　今回は手書き数字画像であるMNISTを分類するディープラーニングをPyTorchで実装しました。次回は深層強化学習のDQN（Deep Q-learning）を実装するために、DQNについて解説を行います。