
Practical introduction for deep learning system
impress
top gear
PyTorchによる画像・自然言語処理
ディープラーニング実装入門
吉崎 亮介／祖父江 誠人＝著
PythonのAIライブラリを徹底活用！
深層学習システムの仕組みを基礎から解説
クラウドを利用した基盤構築も詳しく紹介
インプレス

■サンプルコードのサイト

本書のサンプル環境は、下記サイトで公開されています。

https://hub.docker.com/r/kikagaku/pytorch-topgear

■正誤表のWebページ

正誤表を掲載した場合、以下のURLのページに表示されます。

https://book.impress.co.jp/books/1119101036

はじめに

昨今のデータサイエンス界隈は活気にあふれています。日替わりで話題になる AI の新技術、世界中どこでも話題の絶えない研究技術。中でも「ディープラーニング」は、多くの開発者の関心事になっています。本書を手にとられた皆さんも、少なからずこの流行を感じているのではないでしょうか。

現在ディープラーニングの使い方を解説する記事や書籍は多く存在しますが、機械学習の導入方法まで網羅して教えてくれているコンテンツは多くありません。フレームワークの使い方だけを学んだとしても、実務への導入にはまだまだ大きなギャップが存在します。

最新のディープラーニングフレームワークの記述から環境構築の方法、クラウドとの連携、その先の運用までまとめることが、大きな価値になるのではないだろうかと考えたのが、本書執筆のきっかけです。日頃は教育企業として、3 万人を超える方々にデータサイエンスを教えてきた経験から、無駄を省き、必要な要素だけをまとめられたと自負しています。

本書を通じて、より多くの方がディープラーニングの「活用」のための実装力を身につける手助けとなれば幸いです。

2020 年秋

吉﨑亮介

本書の構成

本書は 9 章で構成されています。

1、2 章は、人工知能、機械学習の基礎知識として必要な周辺知識と最低限知っておきたい数学の知識を紹介しています。

3、4 章ではディープラーニングフレームワークである PyTorch の導入編として、基本的な使い方とサードパーティである PyTorch Lightning の記述方法まで紹介しています。6、8 章でも PyTorch Lightning を使用して、ディープラーニングのホットなジャンルである画像処理、自然言語処理の考え方を図を取り入れながら解説しています。

5 章では、環境構築の選択肢として代表的な Azure、Docker、Google Colaboratory の 3 つを紹介しています。環境構築がお済みでない方は、必ず読んでください。

7、9 章では AI の活用を考え、Microsoft 社が提供しているクラウドサービス Azure を用いて、ハイパーパラメータのチューニングと学習済みモデルのデプロイまで一連の流れを紹介しています。実践へと踏み出す一歩を手助けします。

本書の読み方

本書を手にとっていただき、誠にありがとうございます。本書はどこから読み始めていただいても構いません。目次を参照して、気になる章から読み始めてください。

Python および機械学習初学者の方は環境構築から必要となるので、冒頭から順番に知識を積み上げながら読むことをお勧めしています。特に知識を網羅的に吸収したいと考えている方にもお勧めします。

また、機械学習の習得に必要な線形代数、微分、統計学の最低限の数学基礎知識や Python の記述方法、よく使用する Python パッケージの説明は本書では扱いません。しかし、それらの前提となる知識や、その他本書でカバーしていないコンテンツについて、株式会社キカガクが提供している無料オンライン学習サイト「KIKAGAKU」に掲載しています。もしもわからない単語やパッケージがありましたら、下記 URL を参照してください。

`https://www.kikagaku.ai/`

Docker イメージ

本書と同じ環境で学びを進めるために Docker イメージを用意しています。パッケージのバージョン等、書籍と同様で進めたい方は、以下からプルを行い、進めてください。

`https://hub.docker.com/r/kikagaku/pytorch-topgear`

イメージを入手したあと、`docker run` コマンドで `docker run -it -p 8888:8888 kikagaku/pytorch-topgear` と実行すると、localhost の 8888 ポートで Jupyter Notebook にアクセスできます。パスワード「kikagaku」でログインすると、本書のサンプルを実行することが可能です。詳細は上記のサイトをご覧ください。

目　次

第 7 章　ハイパーパラメータの最適化

第 8 章　自然言語処理

第 9 章　デプロイ

第1章 人工知能・機械学習・ディープラーニング

本書ではディープラーニングをメインテーマとして扱いますが、この分野で頻出する用語として、人工知能、機械学習、ディープラーニングがあります。これらの違いを最初に把握しておきましょう。

1.1 人工知能（AI）

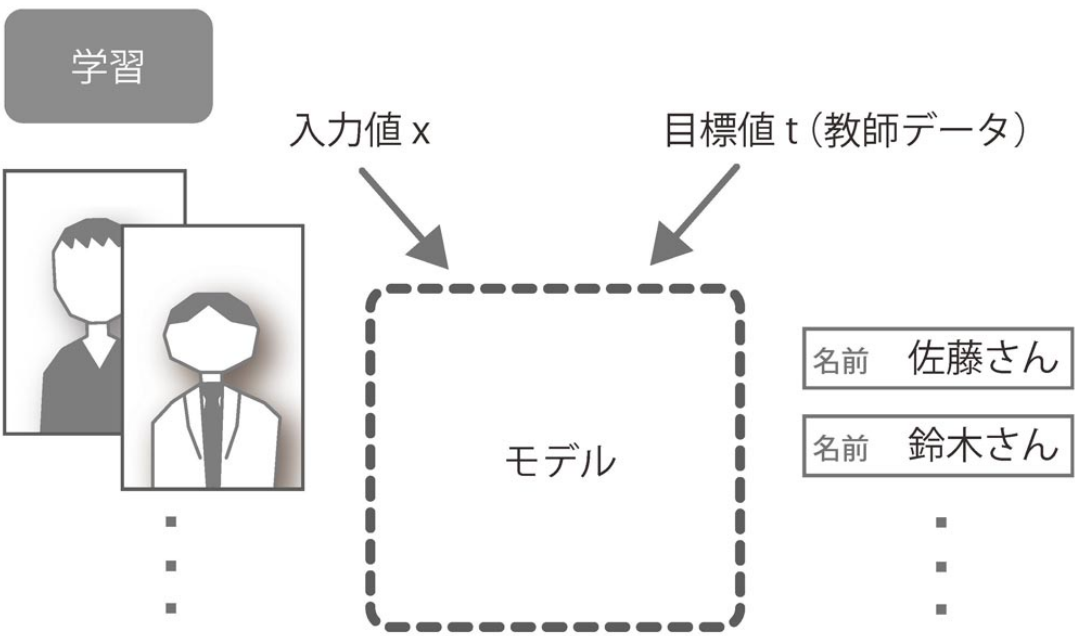
人工知能が3つの概念の中で最も大きな枠となり、人間のように**学習**や**推論**を行わせる概念を指しています。人工知能と言うと、最近でこそ、収集したデータに基づいて学習させるデータ駆動型のイメージが強いですが、本来の意味はそれだけではありません。たとえば、人間が学習・習得した知識・ノウハウをプログラミングすることにより形式化・再現可能にして、それをコン

ピュータで動かすことも、人工知能と言えます。もちろん、この後で紹介する機械学習のように、集めたデータに基づいて（データ駆動型で）コンピュータ自身が学習も行って、学習した結果に基づいて推論を行う場合も、人工知能と言えます。

人工知能を勉強すると必ず出てくるキーワードとして、先ほど述べた学習と推論があります。「どういったことを行うのか？」という具体的なイメージ（＝推論したいこと、問題設定）を持っておくことで、実際に現場へ導入する際に、「どういった情報を集めるべきか」（＝学習すべきデータ）が明確になるため、ここではこの学習と推論に焦点を当てましょう。

学習

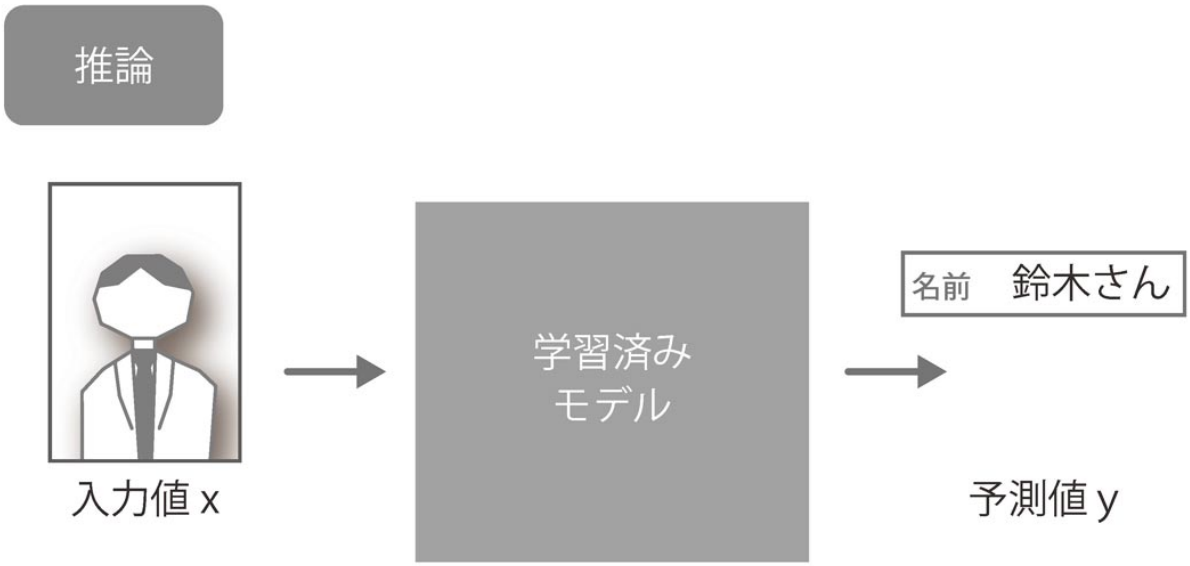
学習とは、規則性を見つけてモデル化することを指します。たとえば、顔写真の名前を推測したいといった問題設定の場合、肌の色などを用いた人間のノウハウを使って名前を推測できるようなプログラムを記述する方法が最初に考えられます。それに対して、後述する機械学習では、下図のように顔写真と名前を 1 つのセットにして与えることで、その名前を予測するロジック自体は特に考えなくても機械が自動的に見つけてくれるという方法です。ただし、自動的に見つけるためには、その関係性を機械が見つけられるだけの十分なデータ量が必要となります。



推論

学習を終えて得られた**学習済みモデル**を用いて、新しい画像に対してその予測値を求めることを、**推論**と言います。AI 搭載のプロダクトには、この学習済みモデルを用いた推論機能が組み込まれています。

AI を使用することが初めての場合には、先に学習が必要です。ただし最近では、人間の顔を検出するなどといった汎用的な問題に対する学習済みモデルが公開されていることも多く、そのような場合は学習のフェーズを飛ばして推論から考え始めることができます。しかし、企業で必要なモデルの大半は、企業内のデータを使って構築したい独自のモデルのはずです。



1.2 機械学習

収集したデータに基づいてモデルの学習を行い、そのデータの構造の特徴をつかむことを、**機械学習**と言います。その対象となるのが、人工知能でも紹介した、ノウハウをプログラミングすることです。機械学習では、データに基づいてその規則性を見つけます。

機械学習には、以下の 3 つのトピックがあります。

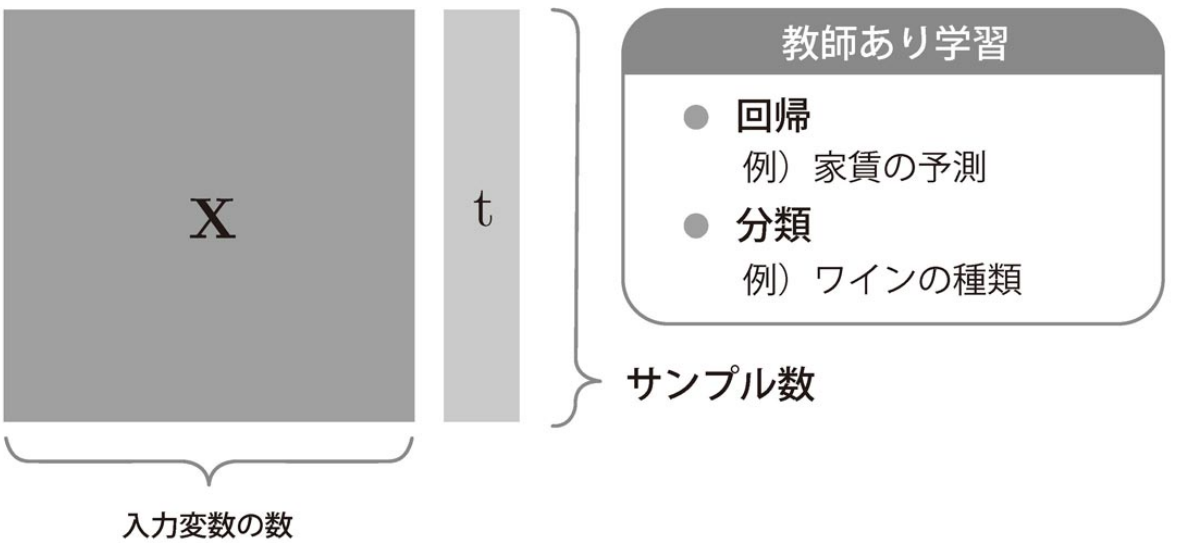
- 教師あり学習
- 教師なし学習
- 強化学習

教師あり学習

教師あり学習では、人工知能の学習の図（p.2）で示したように、入力値 x に対して目標値 t を与えて、その関係性を見つけることをタスクとします。教師あり学習は、大きく**回帰**と**分類**に分けられます。回帰は出力となる値が連続値で表現され、たとえば家賃 50,000 円といったよう

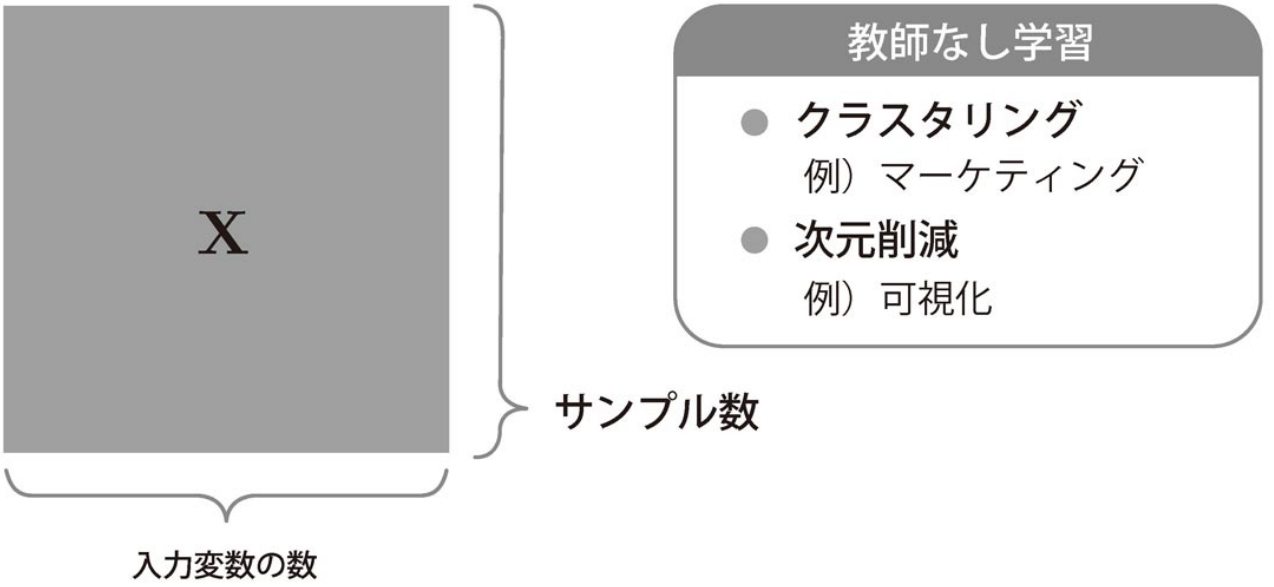
な値です。それに対して、分類は属するクラスを予測し、すでに例として紹介した「佐藤さん」「鈴木さん」といったようになります。

教師あり学習の最大の特徴は、**教師データが必要**であるということです。教師データとは「佐藤さん」「鈴木さん」のように学習のために必要な答えのデータです。この教師データは地道に溜め続けるか、人間が手作業でラベル付けを行う必要があります。



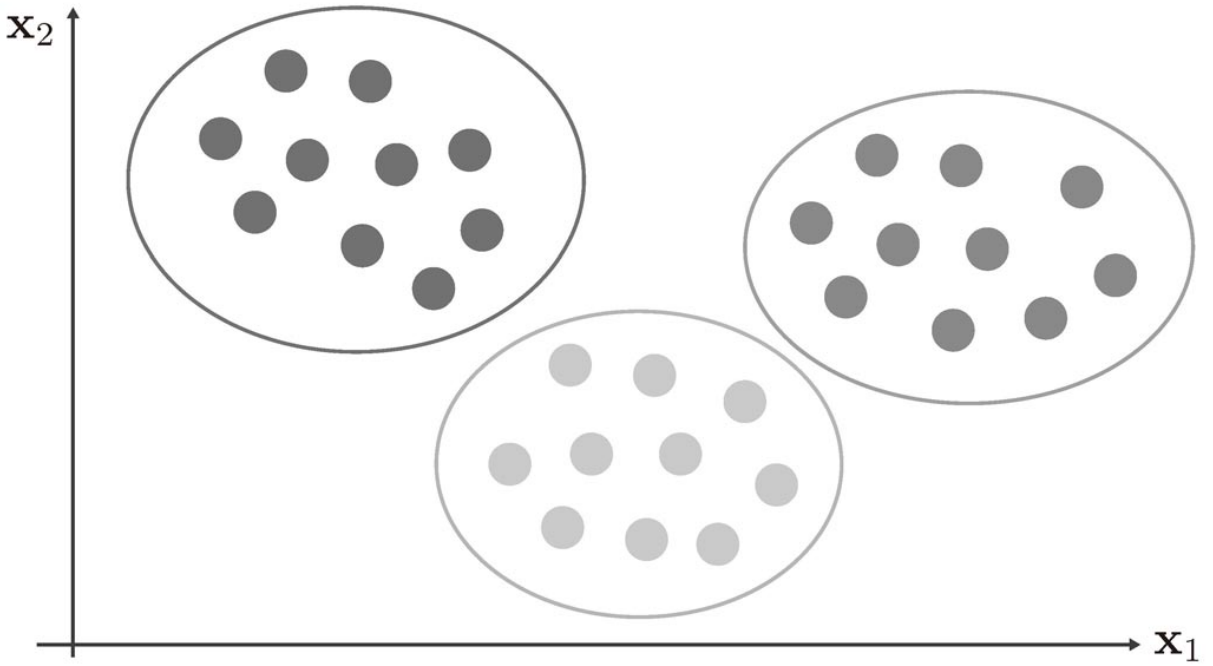
教師なし学習

教師なし学習は、教師データがないため、入出力間の関係性を見つけるのではなく、与えられたデータの特性をモデル化して把握することを目的としています。教師なし学習としては**クラスタリング**や**次元削減**が有名ですが、ここではクラスタリングについて紹介します。



クラスタリングでは、近くに存在するデータをまとめた**クラスター**を作ります。もちろん、答えがない中で各クラスターに分けないといけないのですが、分けるべきクラスターの数や、その分割の妥当性は測ることができません。そこで、その妥当性を別のアクションで計測します。

たとえば、これまでメルマガを全員に同じ内容で送っていたところを、3 つのクラスターが存在すると仮定して、クラスターごとに別々の内容を送るとします。このとき、メールの開封率がこれまで 30% だったところが 45% に上昇したら、そのクラスタリングが妥当だったと思われます。さらにメールの開封率を高めるようにクラスターの分割を工夫していきます。



教師なし学習では、答えこそありませんが、そのデータの構造を把握することで、次のアクションへつなげる根拠作りができます。

強化学習

AlphaGo（https://deepmind.com/research/case-studies/alphago-the-story-so-far）が囲碁で人間に勝利して以来、強化学習も注目を浴びるようになってきています。**強化学習**は、教師あり学習や教師なし学習とは異なり、少数あるいはまったくデータがない状況でも学習を行うことができ、考え方も少し異なります。

わかりやすい例としては、地図も与えられずに部屋に置かれたロボットが部屋の構造を把握するときに、強化学習を用います。部屋の構造がわからないときには、とりあえず障害物に当たるまで歩いてみます。そして障害物に当たると、方向を変えてまた歩いてみます。これを繰り返すことで部屋の情報を把握できます。このときに、強化学習ではランダムに歩くのではなく、規則性を導入することで効率よく探索ができるように工夫します。行動しながらデータを収集し、そのデータで学習を行い、次の行動を決めていくといったプロセスまでが強化学習には含まれているところが特殊と言えます。

このような説明を聞くと、経営の意思決定なども強化学習で行うことができれば万能ではと思うかもしれませんが、強化学習にも適用するための条件があります。それは、シミュレーションできる環境、もしくは実機で何度も再現できる環境が必要であるということです。この条件を満たすことができる代表的な領域が、ゲームとロボットです。

ゲームの領域では、ルールが決まっているため、何度でもそのゲームを再現することができ、多くの行動の試行錯誤ができます。これにより、より強い手を打つ行動を学習できます。

ロボットの領域では、物理法則というルールがあり、この法則に基づいてシミュレータを作ることができます。シミュレータ上ではゲーム同様に何度でも試行錯誤ができ、洗練された行動を行えるように学習できます。

これに対して、会社の経営の意思決定にはルールがあるでしょうか。このルールがシミュレートできるくらいにロジックが明らかであれば、そのときはあえて強化学習を用いなくても最適な行動を決められるはずです。ここまでで、強化学習を使えるかどうかを考える基準が把握できたでしょう。

1.3 ディープラーニング

機械学習はあくまでも概念であり、固有の具体的な手順というものはありません。そこで実際

にコンピュータが実行できるレベルまで具体的な手順を明確化したものを、**アルゴリズム**と呼びます。そして、この機械学習アルゴリズムの１つが**ディープラーニング**です。もちろん、ディープラーニング以外にも「決定木」や「ロジスティック回帰」といった他の機械学習アルゴリズムも存在します。

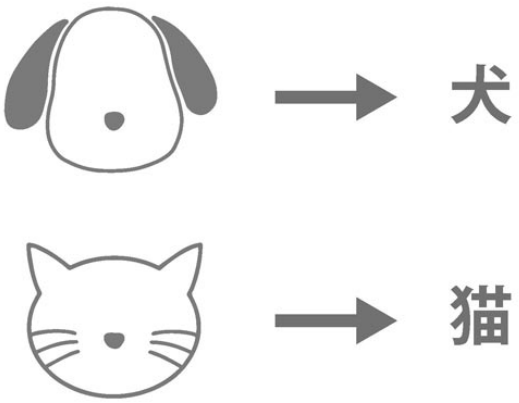
しかし、ディープラーニングも機械学習アルゴリズムの１つではあるのに、従来の機械学習アルゴリズムとは取り上げられ方が違うように感じます。この理由について考えてみましょう。

ディープラーニングは、主に教師あり学習で用いられます。教師なし学習や強化学習でも用いられますが、本書では教師あり学習の用途で使用することを主としているので、ひとまず教師あり学習で考えましょう。教師あり学習では、入力データと教師データを与え、その関係性をそれぞれのアルゴリズムで表現できるようにパラメータを調整していきます。この概念自体には従来の機械学習アルゴリズムとディープラーニングに違いはありません。それでは、何が違うのでしょうか。

その答えの１つは、取り扱えるデータの形式です。従来の機械学習アルゴリズムは以下の図に示すようにスプレッドシートのような形式に構造化されたテーブルデータのみを扱うことを前提に、アルゴリズムが考えられていました。たとえば、駅からの距離や最寄りのスーパーとの距離などを入力として、家賃を予測するといったケースがあります。このようなデータの関係性を見つけられれば、多くのビジネスで役に立つはずです。

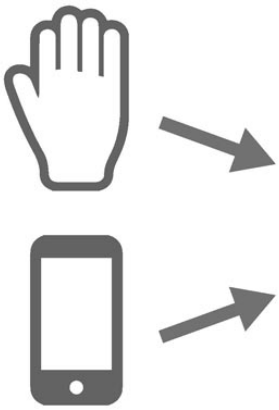
部屋の広さ	駅からの距離	家賃
60	600	10
30	800	4
⋮	⋮	⋮
40	300	8

しかし、ここで１つ疑問があります。与えられた画像が「犬」か「猫」かを予測したいときには、データがスプレッドシートのように構造化されたものではなく、画像というこれまでとは異なる形式のものを扱わなければなりません。従来の機械学習アルゴリズムでは、これがなかなかうまく扱えませんでした。



もちろん、まったく扱えなかったというわけではありません。**特徴抽出**と呼ばれる機構を前処理として用意しておき、その機構によって取り出した**特徴量**を入力データとして与えてやればよいはずです。

たとえば、手とスマートフォンを見分けるような問題設定で考えてみましょう。手は肌色、スマートフォンは黒色と特徴が出ているため、光の三原色である RGB の値を入力変数として抽出してみることができます。このように特徴抽出すれば、構造化されたデータとして扱えます。

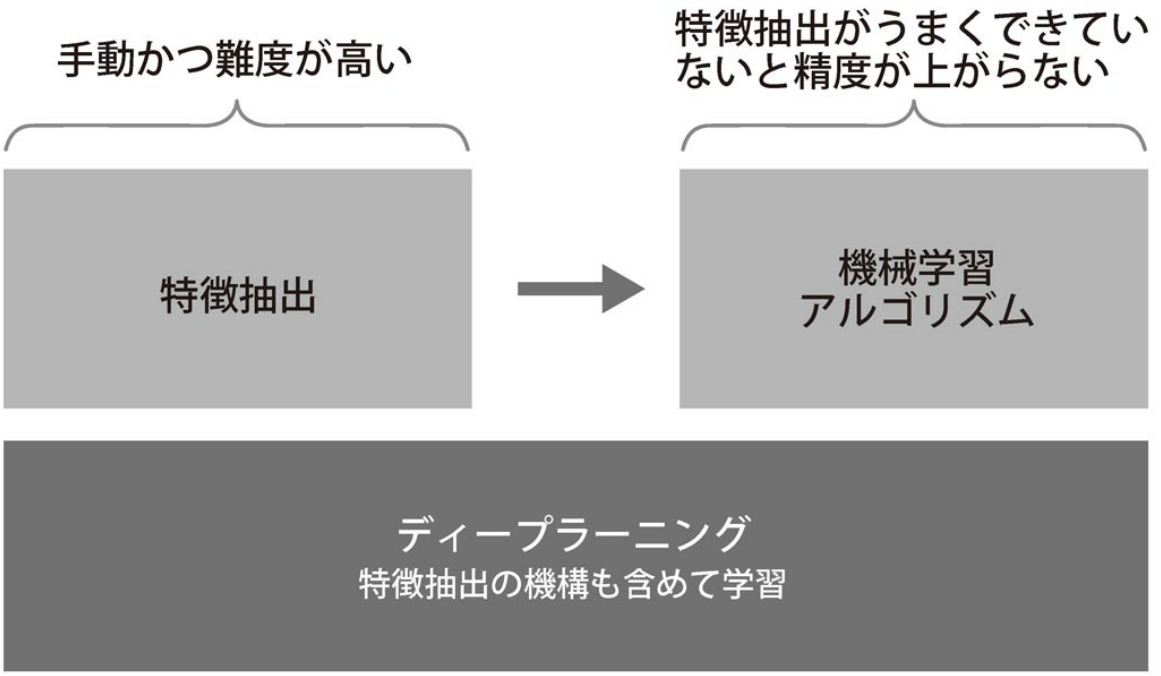


Red	Green	Blue	カテゴリ
1	0.8	0.7	手
0	0	0	スマートフォン

上記は代表する色などから分類できるような極端に簡単な例ですが、犬と猫を分類する場合にはどのように特徴抽出すればよいか再度考えてみましょう。そもそも、犬と猫の違いを言語化すること自体が難しく、これをコンピュータが判断可能なアルゴリズムに落とし込んでいくことも、相当難度が高いと言えます。そのため特徴抽出をうまく適用できず、従来の機械学習アルゴリズムでは上手に扱えませんでした。

ディープラーニングは、この問題にメスを入れました。従来の機械学習ではテーブルデータしか扱えなかったことに対して、画像を画像のまま扱いながら特徴抽出できる機構を構造に加えることで、犬と猫の違いを言語化できなかったとしても、その特徴抽出を学習によって同時に行え

るようにしたのです。



「ディープラーニングは他の機械学習アルゴリズムよりも精度が良い」と語られることも多いのですが、筆者は「ディープラーニングは他の機械学習アルゴリズムでは考慮されていなかった機構を加えることで、扱えるデータの形式の幅を広げた」と考えます。従来の機械学習アルゴリズムが扱えるようなテーブルデータに関しては、ディープラーニングはそこまで大きな変化はありません。むしろ、構造が他のアルゴリズムよりもさらにブラックボックス化されているため、好まれない場合もあるくらいです。

また、画像と同様に扱える幅が広がった領域として、自然言語があります。日英翻訳などの翻訳機能が数年前に大きく改善されたことは記憶に新しいところですが、その大きな改善の裏にはディープラーニングの登場がありました。自然言語はなぜディープラーニングと相性が良かったのでしょうか。

たとえば、「私は吉崎です。よろしくお願いします。」という「挨拶」の文章と、「私はあなたのことが好きです。」という「告白」の文章をそれぞれ分類したいとしましょう。従来の機械学習アルゴリズムでは、どうにかテーブルデータまで落とし込まないと入力データとして受け付けられないのですが、では一体どのようにこれらの文章を数値データに落とし込めばよいでしょうか。この見当がつかないことが、自然言語の難しさです。どのように数値に落とし込むのかという定量評価ができない問題と、文章の長さが文章ごとにまったく異なっていて統一のフォーマットとしてどのように扱えばよいかわからないという問題があります。

このように長さが異なるデータのことを**可変長データ**と言います。それに対して、長さが毎回

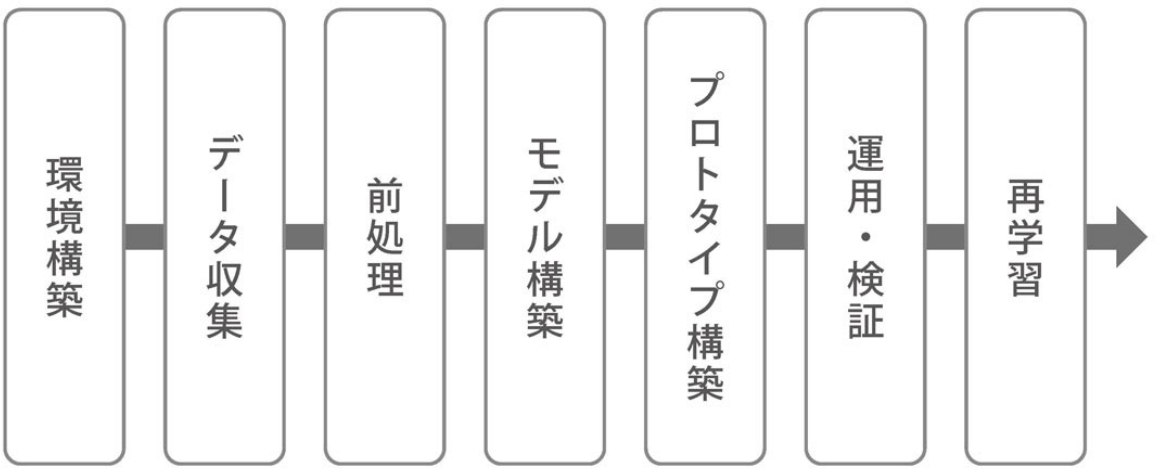
同じデータを**固定長データ**と言います。これらの問題に対処できる上手な機構を設けることで、ディープラーニングで解ける問題設定として扱えるようになり、結果として翻訳機能の性能が大きく改善しました。論より証拠がその有用性を証明したわけです。この機構については解説が複雑になるため、学びを深めた後にある、自然言語処理の章（第 6 章）までの楽しみとして、とっておいてください。

ディープラーニングが万能であるかのような世間の騒がれ方ですが、問題設定によってはディープラーニングでないほうが妥当なこともあるため、本書で学び進めながら技術選定の取捨選択ができるようになってください。ただし、従来の機械学習よりも幅広い問題設定に対応できるような柔軟性の高い機構を有しているディープラーニングの考え方を学ぶことは、これからシステム設計を行う上で必ず役に立つはずなので、ぜひ楽しみながら学んでいただければ幸いです。

1.4 ディープラーニング活用までの流れ

本書では研究として論文に書くことをゴールとするのではなく、実務で活用できるというところをゴールに置いています。

初学者が取りかかり始めてからどのような道筋があるのかを整理しておきましょう。理論や実装について把握できれば、次は具体的にプロジェクトを進めていくことになります。プロジェクトの進め方の一例を以下に示します。

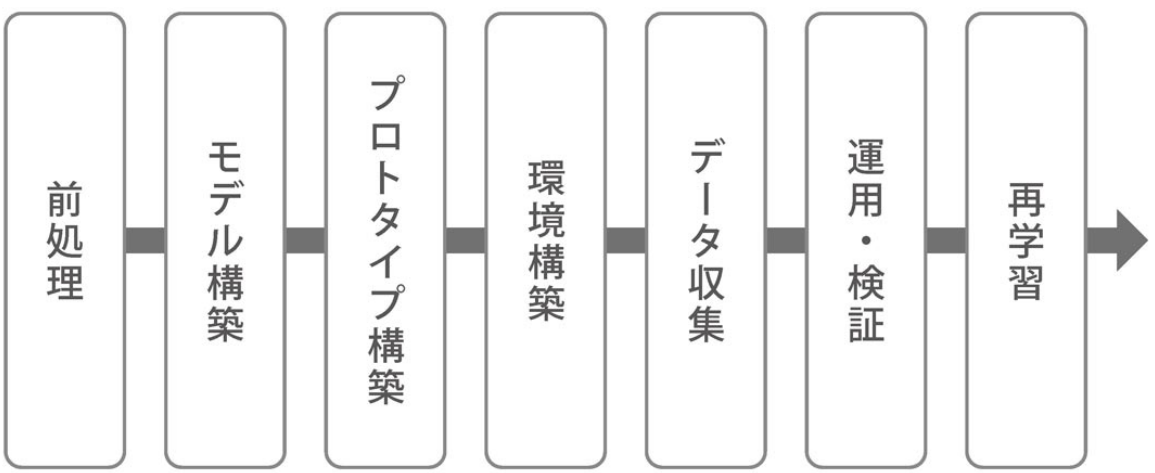


ディープラーニングを学ぶということは基本的にモデル構築を学ぶことですが、実際のプロ

ジェクトを考えてみると、それは一工程に過ぎません。この認識のギャップも、実際のプロジェクトを進める際の障害になっていると感じています。

書籍でディープラーニングの理論を身に付け、いざ実際のプロジェクトへ進もうとすると、実力不足でモデル構築まで全然たどり着けず、思うようにプロジェクトが進められないということが想定されます。この実力不足というのは、モデル構築を行うスキルが足りていないという意味ではなく、環境構築やデータ収集・前処理を行うときに必要なスキルが不足しているという意味です。この点まで網羅している書籍は少ないと筆者は感じています。

そこで本書では、これらの全行程を網羅した内容となるように構成を考えました。そして、この全行程を学んでいく中で注意した点として、各工程の頭から順番に学ばないということです。モデル構築といった本書のメインターゲットであるところはプログラミングに慣れていない方でもその実装を学ぶことを通して慣れていけますが、先頭にある環境構築はある程度コマンド操作に慣れていなければ進めるのが難しいことがあります。そのため、本書の構成は、これらすべての工程を学ぶことができるようにしつつ、学ぶことに対する難度が高いものほど後半に持ってきています。初学者の方は前から順番に学び進めることで、ディープラーニングの活用に必要なスキルセットをひととおり習得できます。



もちろん、本書を読み通せばそれで完璧というわけではありません。モデル構築に関しても、最新の論文ではさらに多くの議論がなされていますし、それぞれの項目に関しても深く研究できることがたくさんあります。ただし、本書をひととおり読むことで、何がどのような役割をしているか理解できるため、実際にプロジェクトでの利用において何に関して深く進めていくべきかも明確に見えてくるはずです。「何をどのようにどういう順番で学んでいくとよいのかがわからない」という初学者によくある問題への答えを本書が示せることを願っています。

1.5 ディープラーニング周辺の技術地図

ディープラーニング自体は機械学習アルゴリズムの 1 つですが、この技術を実務に活かそうとすると環境構築から始まり、いくつものプロセスがあると説明しました。そのプロセスを学んでいく際に、クラウドや GPU といった専門用語が頻出するため、まずは周辺にある技術の位置付けについて、整理しておきましょう。

計算機資源 (Computing Resource)

ディープラーニングでは学習や推論の際に、コンピュータによって計算を行う必要があります。通常は手元にあるコンピュータで計算を行うため、計算の処理にかかる時間がスペックに依存します。この手元にあるコンピュータのことを**ローカル**と呼びます。しかし、計算を行う環境は手元にあるコンピュータ以外でも可能で、高性能な計算性能を持つマシンを利用することも選択肢となります。

この選択肢の最初の分岐は、**オンプレミス**と**クラウド**という 2 つです。オンプレミスは、サーバーを購入して自社の中のサーバールームなどに置き、そのサーバーへアクセスして利用するという考え方です。それに対してクラウドは、自社でサーバーは保有せずに、クラウドサービスの提供事業者から外部サーバーを借りて運用するという考え方です。オンプレミスにしてもクラウ

ドにしても、自分の手元にあるマシンではない、別のマシンへ接続して計算を行います。このように手元にないマシンのことを、**リモート**と呼びます。

長期的に考えるとオンプレミスのほうが安いと言われていますが、オンプレミスでは一度購入すると新型への切り替えが難しく、また保守運用も自社内の人員で行う必要があるため、小規模な場合には人件費を含めたトータルコストで考えると高いこともあります。一方、クラウドでは時間単位で借りることができるため、使用していない時間は課金されず、常に最新の機器を使えます。どのくらい使用するかによって金額が変わるため、最終的にかかる総額を見積るのが面倒といったデメリットはありますが、小規模に使用していく場合にはクラウドのほうがメリットが大きいでしょう。

最近では**エッジ**という言葉も登場していますが、これはクラウドとオンプレミスに並ぶものではなく、オンプレミスでの選択肢の 1 つです。エッジとは、サーバーよりもユーザーに近い端末を指しており、たとえばスマートフォンが挙げられます。すべての処理をオンプレミスのサーバーで行うとユーザー数が増えた際に負荷がかかりすぎるため、エッジの端末で処理を行うことで負荷を分散させます。これからはエッジコンピューティングという言葉も浸透していくと予想されるため、オンプレミスとクラウドに加えて覚えておきましょう。

演算装置

サーバー内にある演算装置としては、CPU（Central Processing Unit）と GPU（Graphics Processing Unit）の 2 つが代表的です。CPU はローカルの PC のどれにも必ず搭載されており、なじみがあると思いますが、指示された処理を行うための装置です。GPU も同様に処理を行いますが、もともとはゲームなどのグラフィックの性能を高めるための演算装置として開発されました。CPU が複雑な直列の処理に向いているのに対し、GPU はグラフィック用に単純な処理を並列で行うことに特化して設計されています。

ディープラーニングでは簡単な足し算や掛け算といった単純な処理を多く行います。これに GPU での演算が向いているという背景から、（本来の用途とは異なりますが）ディープラーニングの処理を高速化するためによく使用されるようになりました。ローカルの PC にディープラーニングなどの科学計算向けに特化した GPU が搭載されていることはまずないため、クラウドやオンプレミスにあるリモート上の GPU が搭載されたマシンを使用します。本書でも GPU を使用した計算の高速化の例を紹介します。

それ以外にも最近では、Google 社が開発している TPU（Tensor Processing Unit）という、GPU よりもディープラーニングで使用される計算に特化した演算装置も登場しています。また、FPGA（Field-Programmable Gate Array）と呼ばれる、設計者が構成を設定できる集積回路にも注目が集まっています。FPGA では必要な計算に最適な回路の構成を設定できるため、ディープラーニングの演算向けに開発された TPU よりも、さらに効率よく高速に計算できる

可能性があります。トレンドとして、汎用的な処理を行う演算装置から、ディープラーニング向けに特化した演算装置へとどんどん移り変わっているわけです。この FPGA の実用化には、Microsoft 社をはじめとする企業や研究機関が関わっています。GPU での計算に慣れてきた頃には、次のステップとして TPU や FPGA があるため、覚えておきましょう。

オペレーティングシステム（OS）

Windows や macOS といった OS は記憶装置（ハードディスクなど）などのやり取りを標準化しており、その OS を前提としてソフトウェアが開発されています。ソフトウェアを開発する側でなければ、各 OS による特性の違いを意識することは少ないでしょう。ディープラーニングを含めた機械学習を実装するための環境構築の手順が異なることが、チームでの開発の難点になることがあります。それぞれの OS やバージョンが違うと、環境構築の再現性を確保できないことがままあるのです。

また、本格的にディープラーニングを使っていく上では、Linux も使うことになります。Windows は有償であり、macOS は Apple 製品のみでしか使用できないという問題に対して、Linux は無償で利用できる OS なので、サーバーでよく採用されています。OS 自体は Linux と総称されますが、具体的にはその OS に必要なアプリケーションを組み合わせたディストリビューションを使います。代表的なディストリビューションとして、Debian と RedHat があります。ただし、Debian や RedHat のディストリビューションを使用している人は少なく、Debian からさらに派生した Ubuntu や、RedHat から派生した CentOS を使用することが一般的です。特に、ディープラーニングの開発では Ubuntu をベースとしている例が多いため、チーム開発で他の人が使用しているといった制約がなければ、Ubuntu を使うことを本書では推奨します。基本的には CLI（Command Line Interface）と呼ばれるようにコマンドでマシンの操作を行い、マウスを使った GUI（Graphical User Interface）ではありません。Windows や macOS では GUI による操作が基本なので最初は慣れないかもしれませんが、混み入ったことをしなければ、ファイルの作成や削除、アプリケーションのインストール程度ですので、使いながら慣れていけるでしょう。

仮想化環境

マシンによって OS が異なることで開発環境を統一できず、それぞれの OS での環境構築の手順を毎回探して実施するとなると、一苦労です。また、サーバーも 1 台だけでなく 100 台以上を扱うことも実務では想定されるため、その環境構築を毎回手作業で行うことも現実的ではありません。

そこで、**仮想化環境**と呼ばれる技術を利用して、どの OS でも同じように環境構築ができ、構築した環境を簡単に使用できるように工夫します。仮想化環境を実現するためのソフトウェア

はいくつかありますが、その中でディープラーニング開発を行う上で現在最も人気のあるのが**Docker**です。Dockerの層を挟むことにより、Windows・macOS・Linuxを問わず、同一の環境で開発ができます。また、構築した環境を簡単に配布できるため、チームの誰かが環境を構築すれば、他の人は苦労して環境構築を行う必要がありません。本書でもやや難しい環境構築に関しては、構築済みの環境をDockerによって配布していますので、読者の皆さんは学習に集中できます。

```
https://hub.docker.com/r/kikagaku/pytorch-topgear
```

イメージを入手したあと、`docker run`コマンドで`docker run -it -p 8888:8888 kikagaku/pytorch-topgear`と実行すると、localhostの8888ポートでJupyter Notebookにアクセスできます。パスワード「kikagaku」でログインすると、本書のサンプルを実行することが可能です。詳細は上記のサイトをご覧ください。

異なるOSを吸収できるとはいうものの、実際には特殊なOS上で作業するのではなく、基本はLinuxです。特にUbuntuを含めたDebian系のOSをベースにしている場合が多いため、その点でもLinuxの操作に慣れておくことにはメリットがあります。Dockerについては本書でその簡単な使い方も紹介するため、ディープラーニングの一連の流れに慣れてきた頃に次のステップとして学んでください。

プログラミング言語

会話を行う際に日本語や英語といった言語（自然言語）の種類があるように、機械に対してプログラミングを行う際には、どのプログラミング言語を使用するかを決める必要があります。役割ごとに複数のプログラミング言語を組み合わせて使用する必要があります。読者の皆さんがよく使用するWebアプリケーションの見た目を作ることが得意な言語もあれば、その背景にあるデータベースとのやり取りなどを行うことが得意な言語もあります。もちろん、データ解析が得意な言語もあります。

データ解析の得意な言語としては、PythonやR、研究機関ではMATLABといった言語が代表的です。これらの同じ領域を対象としたプログラミング言語は、基本的には組み合わせず、どれか1つを選択して使います。

チームのメンバーが使用している言語を選択すれば引き継ぎが楽なので、その言語を選択すればよいでしょう。もし、そのような制約がない状況ではどの言語を選べばよいでしょうか。

まず、無料か有料かという分岐点があります。PythonやRは無料ですが、MATLABは有料です。無料を選ぶとして、次にPythonかRかでは、どちらを選ぶとよいでしょうか。言語の特性というよりも、その言語に紐づくコミュニティやフレームワークの観点で決めることを筆者は推奨します。個人的な印象として、ディープラーニング開発を行うときにはPythonを使用して

いることが多いと感じます。その理由として、ディープラーニングは 1 から組み上げるのではなく、企業や組織が作成した**フレームワーク**と呼ばれる枠組みを使って組むことが一般的ですが、そのディープラーニングフレームワークの大半が、Python 向けに作られているのです。そのため、本書でも Python を採用しています。

また、Python を採用するのには、ディープラーニングでよく使われているということ以外にも、もう 1 つ理由があります。R はデータ解析に特化した言語ですが、Python はデータ解析のほか Web アプリケーションの開発にも長けています。そのため、解析した結果を Web アプリケーションなどのシステムに組み込むことも容易です。解析から組み込みまで 1 つの言語で行うことができれば、覚えることが少なくて済むというメリットがあります。

パッケージ

Python では、他の人が作成した**パッケージ**と呼ばれるプログラムを利用して、開発の速度を上げることができます*1。データ解析で有名なパッケージとしては NumPy、Pandas、Matplotlib などがあります。Web アプリケーションでは Flask や Django といったパッケージが有名です。

ディープラーニングでは本書で扱う PyTorch や TensorFlow、Chainer などが有名です。これらはそれぞれの特色がありますが、研究の論文は PyTorch で書かれていることが多く、実務では Tensorflow やその派生の Keras が使われていることが多い印象です。もちろんその逆に、PyTorch を実務に使うこともよくありますし、研究の論文が TensorFlow で書かれていることもあります。

*1 PyTorch などのディープラーニング向けのものは、厳密にはパッケージではなくフレームワークと呼ばれていますが、その違いを説明することは難しく、利用目線ではパッケージとフレームワークは大きく変わらないため、現時点では気にしなくても問題ありません。もし厳密な話が気になる方は「制御の反転」がキーワードになります。

第2章 ニューラルネットワークの数学

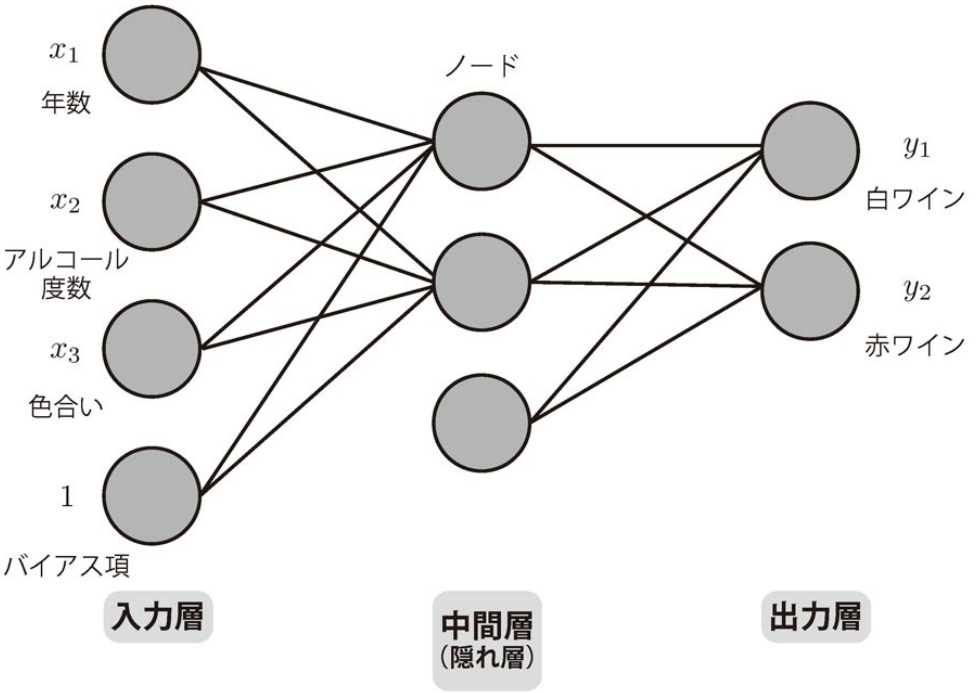
本章では、ディープラーニングの基礎となるニューラルネットワークのアルゴリズムについて紹介します。ディープラーニングも内部はニューラルネットワークですが、画像や自然言語処理といったタスクに向けた特定のネットワークの構造を持っていたり、これから紹介する層の数を深く複雑にした場合でもうまく学習できるような工夫がされていたりするため、基本的なニューラルネットワークとは切り分けて解説します。

2.1 ニューラルネットワークの概要

ニューラルネットワークは回帰や分類といった教師あり学習のアルゴリズムです。重回帰分析や決定木などと問題設定は変わりませんが、微分可能な構造をつなげていくことでより柔軟なネットワーク構造の設計を可能にしています。ディープラーニング以前の機械学習アルゴリズムでは扱いにくかった画像や自然言語といった領域でも成果が上がっているのは、この恩恵です。構造を工夫することで教師なし学習にも利用できたり、強化学習にも利用できたりするので、機械学習の全分野で登場します。

それでは、ニューラルネットワークの構造を見ていきましょう。次ページの図のように、今回は例題として、白ワインと赤ワインを分類するようなケースを考えます。

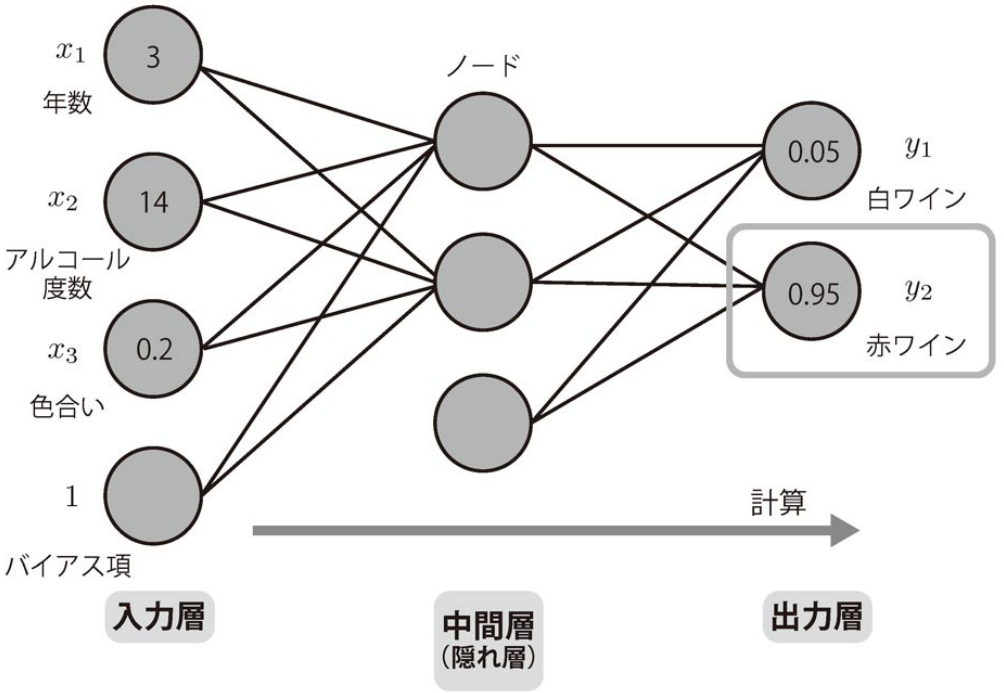
ニューラルネットワークは**ノード**をつなぎ合わせた**層**を持つ構造になっています。各ノード間をすべてつないでいるため、この構造を**全結合層**と呼びます。**入力層**では、入力変数の数だけノードを用意します。バイアス用の項も用意するのが一般的です。**出力層**は分類したカテゴリの数だけノードを用意します。次ページの図では2値の分類なので、出力変数の数も2個になっています。**中間層**は入力変数を特定の処理で変換し、より分類するために有利になるような値を得られるようにします。



ニューラルネットワークでは、数学やプログラミングを考える際に **2 層で 1 セット**として扱います。小さな単位に切り分けてその集合体として考えることで、複雑なネットワークにおいても議論が容易になります。

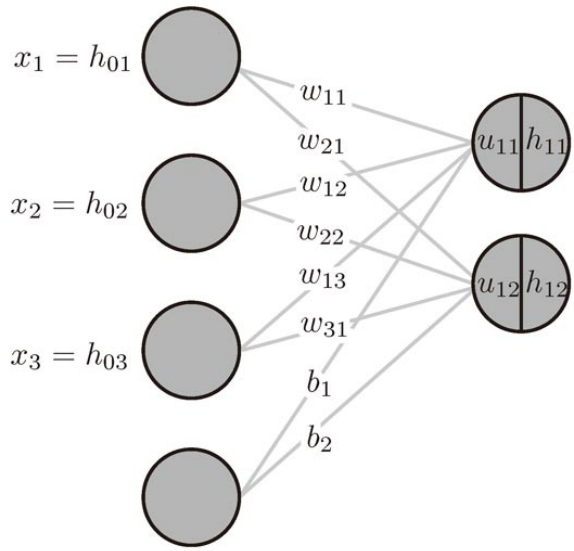
具体的な計算手順はこれから紹介していきますが、次ページの図のように $\mathbf{x} = [3,\ 14,\ 0.2]^{\mathrm{T}}$ の入力値を与えると計算が行われ、出力の予測値は $\mathbf{y} = [0.05,\ 0.95]^{\mathrm{T}}$ のように得られます。予測値の総和は 1 となります。つまり、白ワインである確率が 5%、赤ワインである確率が 95% といったような予測を行っています。この値をもとに、最も確率の高い赤ワインを選択します。

したがって、内部のアルゴリズムでは確率の値を求める回帰となっており、この計算によって得られた値に基づいて分類しています。



2.2 順伝播

概要で紹介したニューラルネットワークの具体的な計算の流れを紹介します。入力された値に基づいて予測値を計算する際に順方向に値を伝播させていくため、**順伝播**と呼びます。ニューラルネットワークの数学を考える場合には、次ページの図のように 2 層で 1 セットで考えると前述しました。w や b は、重みとバイアスです。u や h が新しく登場していますが、それぞれの計算を行う**線形変換**と**非線形変換**をこれから紹介します。



線形変換

線形変換では重みと入力変数の線形結合を行います。したがって、線形変換後の変数 u_{11} と u_{22} は以下のように計算されます。

$$u_{11} = w_{11}h_{01} + w_{12}h_{02} + w_{13}h_{03} + b_1$$

$$u_{12} = w_{21}h_{01} + w_{22}h_{02} + w_{23}h_{03} + b_2$$

この式は、入力ベクトル $\mathbf{h}_0$、重み行列 $\mathbf{W}_{10}$、バイアスベクトル $\mathbf{b}_1$、および出力 $\mathbf{u}_1$ を

$$\mathbf{h}_0 = \begin{bmatrix} h_{01} \\ h_{02} \\ h_{03} \end{bmatrix}$$

$$\mathbf{W}_{10} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$

$$\mathbf{b}_1 = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$\mathbf{u}_1 = \begin{bmatrix} u_{11} \\ u_{12} \end{bmatrix}$$

と定義すれば、

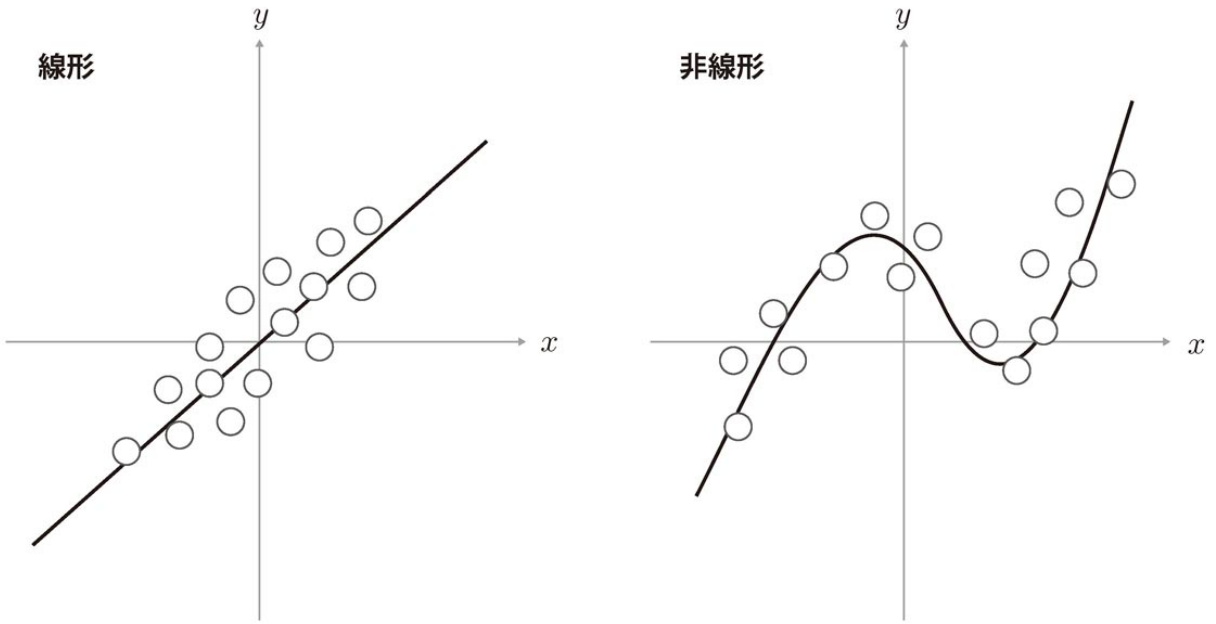
$$\begin{bmatrix} u_{11} \\ u_{12} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} h_{01} \\ h_{02} \\ h_{03} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$\mathbf{u}_1 = \mathbf{W}_{10}\mathbf{h}_0 + \mathbf{b}_1$$

と行列とベクトルを用いた形式ですっきりとまとめることができました。

非線形変換

下図の左に示すように、線形変換では入出力間が線形な関係性を持つ場合のみ、その関係性を表現できます。一方、下図の右に示すように、入出力間が非線形な関係を保つ場合には、その関係性をうまく捉えることができません。



そこでニューラルネットワークでは、各層において線形変換に続いて非線形変換を施し、層を積み重ねて作られるニューラルネットワーク全体としても非線形性を持つことができるようにしています。この非線形変換を行う関数のことを、ニューラルネットワークの文脈では**活性化関数**と呼びます。

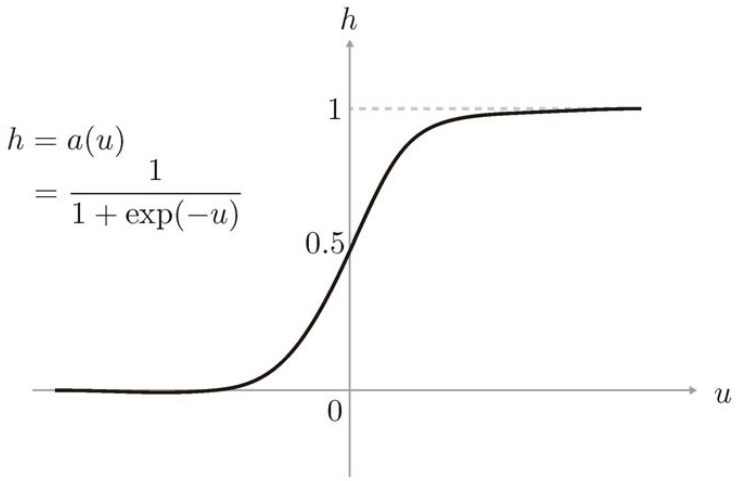
線形変換の節で用いた例に戻って、説明を進めます。線形変換を行った結果 u_{11}, u_{12} のそれぞれに活性化関数 a を用いて非線形変換を行い、その結果を h_{11}, h_{12} とおきます。つまり、

$$h_{11} = a(u_{11})$$
$$h_{12} = a(u_{12})$$

です。これらは**活性値**と呼ばれ、もう 1 つ層が続いているならば、次の層に渡される入力の値となります。活性化関数にはいくつもの種類がありますが、ここでは代表的な活性関数を紹介していきます。

ロジスティックシグモイド関数

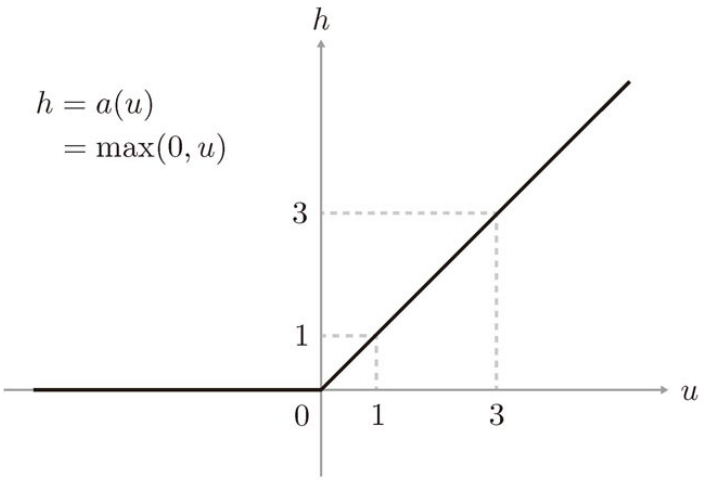
最もよく知られている活性関数はおそらく下図に示す**ロジスティックシグモイド関数**（以下、**シグモイド関数**）です。この活性化関数が従来よく用いられてきました。入力値 u に基づいて、0〜1 の間に値を変換します。関数の形からわかるとおり、直線で表現できないような構造であるため、入出力間の非線形性を一部では捉えることができます。



しかし近年、層の数が多いニューラルネットワークでは、シグモイド関数は活性化関数としてほとんど用いられません。その理由の 1 つは、シグモイド関数を活性化関数に採用すると、**勾配消失**という現象によって学習が進行しなくなる問題が発生しやすくなるためです。今は、ディープラーニングのように**多層化**を進めて複雑な構造を表現する場合にシグモイド関数ではうまく対応できないことが多いという程度の認識で構いません。この問題の詳細は本章の最後で紹介します。

ReLU 関数

この問題を回避するために、最近では次ページの図に示すような**正規化線形関数**（ReLU：rectified linear unit）がよく用いられています。以降、**ReLU 関数**と呼びます。一見すると線形な関数に感じるかもしれませんが、二次元の場合は直線でなければ非線形です。つまり、$y = wx + b$ で表現できなければ非線形であり、その点では、途中で傾きが変わる ReLU 関数は非線形な関数と言えます。



$\max(0, u)$ は、0 と u を比較して**大きなほうの値を返す**関数です。すなわち、**ReLU は入力が負の値の場合には出力は 0 で一定**であり、**正の値の場合は入力をそのまま出力する**という機能を持っていることがわかります。シグモイド関数では、入力が 0 から離れた値をとると、どんどん曲線の傾き（勾配）が小さくなっていき、平らになっていくことが前ページの図からも見てとれます。それに対し、ReLU 関数は入力の値が正であれば、いくら大きくなっていっても、傾き（勾配）は一定です。これがシグモイド関数で問題となっていた勾配消失に対して有効に働きます。この点も詳細は本章の最後に解説します。

Softmax 関数

Softmax 関数は主に分類の問題で使用される活性化関数であり、

$$a(u) = \frac{\exp(u)}{\sum_{m=1}^{M} \exp(u_m)}$$

です。ここで、M は変換後の層のノードの数であり、$\exp(\cdot)$ は自然対数 e を底とした指数関数を表しています。

Softmax 関数の数式自体は難しく見えるかもしれませんが、その機能は出力される値の総和を 1 とすることです。ニューラルネットワークの概要で紹介したとおり、分類の場合、ニューラルネットワークから出力される変数の総和が 1 となり、それぞれの変数は 0〜1 の間で値をとります。これはニューラルネットワーク自体の特性ではなく、計算の最後に Softmax 関数を適用しているためでした。

たとえば、$\mathbf{u} = [-1, 1, 2]^{\mathrm{T}}$ の場合に Softmax 関数を適用してみましょう。NumPy を用いると簡単にできます。

```
import numpy as np

u = np.array([-1, 1, 2])

# exp(u)のそれぞれの計算
exp_u = np.exp(u)
exp_u
```

```
array([0.36787944, 2.71828183, 7.3890561 ])
```

ここからわかるとおり、入力される値が負の場合にも、指数関数の特性上、正の値に変換できています。後はすべての値の総和で割ることで、総和を 1 に調整できます。

```
# 総和
exp_u.sum()
```

```
10.475217368561138
```

```
# Softmax関数適用後の値
a = exp_u / exp_u.sum()
a
```

```
array([0.03511903, 0.25949646, 0.70538451])
```

```
# Softmax関数適用後の値の総和
a.sum()
```

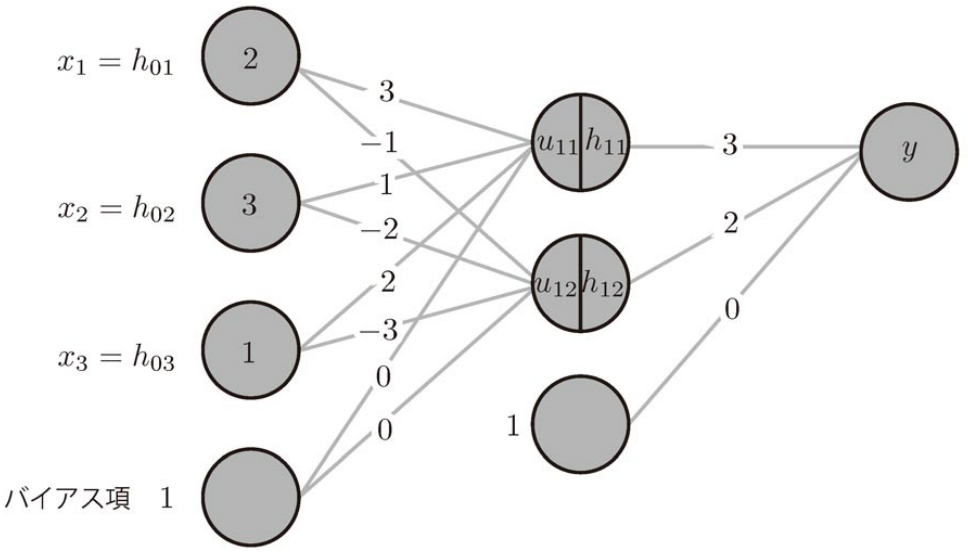
```
1.0
```

分類において Softmax 関数は頻出するので、覚えておきましょう。

数値例で流れを確認

ここまでは数式での説明が多く、現実的な問題と少しずつ乖離してきたため、一度具体的な数値例でどのような計算の流れになっているかを確認することにしましょう。次の図のようにネットワークの入力値と重みを決定しました。なお、活性化関数には ReLU 関数を使用します。こ

こでは出力変数が 1 つの回帰の問題設定としています。



詳細は次節で説明しますが、ここでニューラルネットワークと重回帰分析の間に大きな違いがあります。重回帰分析については、https://www.kikagaku.ai/ の「機械学習の基礎」を参照してください。

ニューラルネットワークも重回帰分析も目的関数を最適化するようにパラメータである w や b を決定することが目的でした。重回帰分析はこれを達成するために目的関数を微分して勾配を求め、その勾配が 0 となるようにパラメータを決定しています。

ニューラルネットワークも同じように進めればよいように感じますが、大きな違いがあります。その謎は活性化関数に隠されています。ReLU 関数は $\max(\cdot, \cdot)$ の関数を持っており、この関数はどのように微分すればよいでしょうか。入力される値によって結果が異なるため、値を入れない代数の段階ではこの微分を行うことができないのです。したがって、重回帰分析のように勾配を 0 にするようなパラメータを直接求めることができません。

そこで、ニューラルネットワークを含めたいくつかのアルゴリズムでは、**パラメータの初期値をランダムに決定**し、そこから徐々に調整していくという方法をとります。詳細は次節で紹介しますので、今はとりあえずパラメータの初期値がないと計算が始まらないという程度の認識で構いません。そのため、今回の数値例においても、パラメータの初期値を設定しています。バイアス b の初期値は 0 と設定されるのが普通なので、ここでも 0 としています。

入力として $\mathbf{x} = [2, 3, 1]^{\mathrm{T}}$ が与えられているため、u_{11} と u_{12} は、

$$u_{11} = 3 \times 2 + 1 \times 3 + 2 \times 1 + 0 = 11$$

$$u_{12} = -1 \times 2 - 2 \times 3 - 3 \times 1 + 0 = -11$$

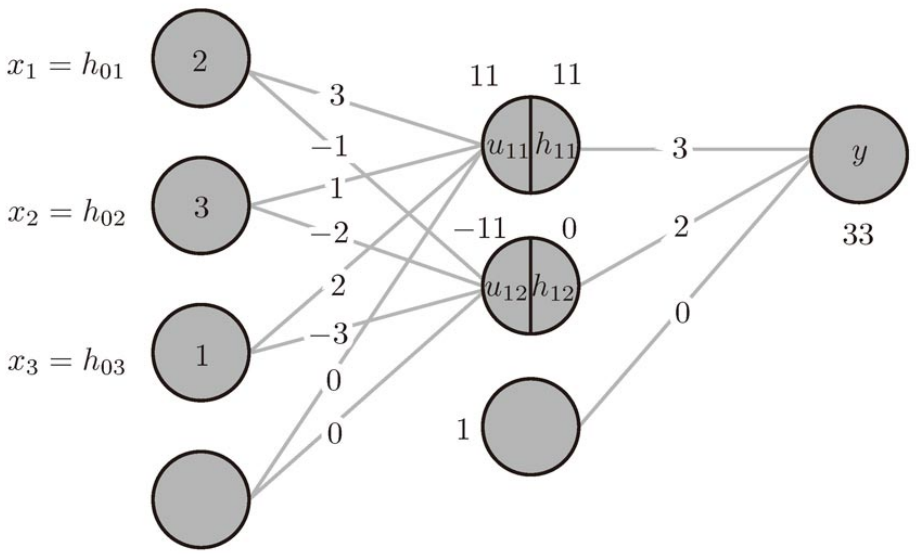
のように求まります。活性化関数は ReLU 関数であるため、h_{11} と h_{12} は

$$h_{11} = \max(0, u_{11}) = \max(0, 11) = 11$$
$$h_{12} = \max(0, u_{12}) = \max(0, -11) = 0$$

となります。これらの値をもとに予測値 y は

$$y = 3 \times 11 + 2 \times 0 + 0 = 33$$

と計算できました。ニューラルネットワークを定式化して説明しているときは難しく見えたかもしれませんが、実際の計算は足し算と掛け算程度で構成されているため、計算自体は非常に簡単であることがわかります。



目的関数

ニューラルネットワークの学習には、**微分可能**であれば、解きたいタスクに合わせてさまざまな目的関数を利用できます。ここでは

- 回帰問題でよく用いられる**平均二乗誤差**（mean squared error）
- 2 クラスの分類問題でよく用いられる**二値交差エントロピー**（binary cross entropy）
- 多クラスの分類問題でよく用いられる**交差エントロピー**（cross entropy）

という代表的な 3 つの目的関数を紹介します。

平均二乗誤差

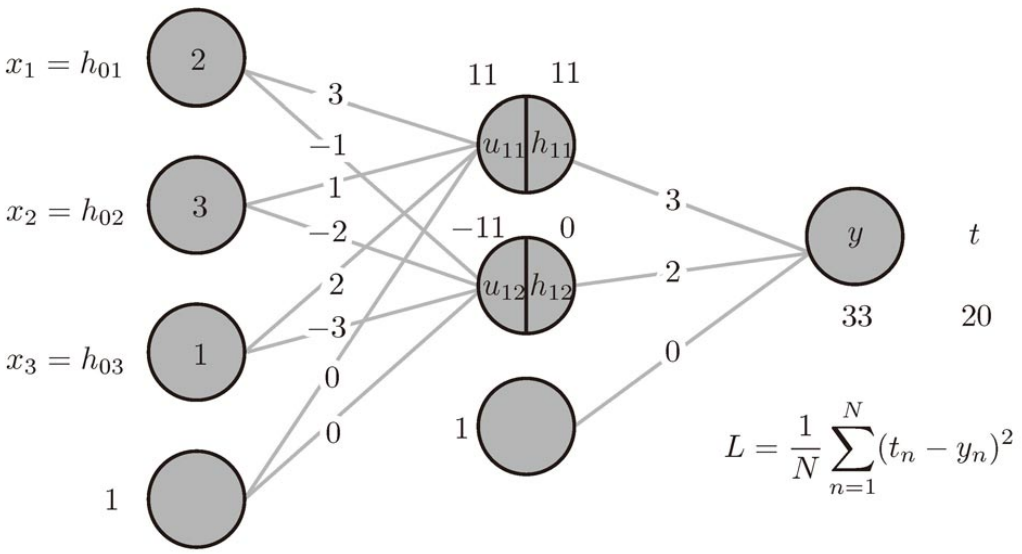
平均二乗誤差（mean squared error）は、回帰問題を解きたいときによく用いられる目的関数

です。二乗和誤差と似ていますが、各データ点における誤差の総和をとるだけでなく、それをデータ数で割って、誤差の平均値を計算している点が異なります。式で表すと、以下のようになります。

$$L = \frac{1}{N}\sum_{n=1}^{N}(t_n - y_n)^2$$

ここで N はサンプル数、y_n は n 個目のデータに対するニューラルネットワークの出力値、t_n は n 個目のデータに対する望ましい正解の値です。

ここで、下図のような例を考えてみましょう。



今、ある入力 $\mathbf{x}$ を与えたときのこのニューラルネットワークの出力 y は、33 でした。もしこの入力に対応する教師データが $t = 20$ であるとすると、今全体のサンプル数が 1 だとして、平均二乗誤差は

$$L = \frac{1}{1}(20 - 33)^2 = 169$$

と計算できます。

二値交差エントロピー

白ワインと赤ワインのように二値分類の場合には、**二値交差エントロピー**（binary cross entropy）を用います。二値交差エントロピーは、

$$L = -\sum_{n=1}^{N} t_n \log y_n + (1 - t_n)\log(1 - y_n)$$

のような形式です。目標値 t_n は 0 か 1 が与えられ、予測値 y_n は Softmax 関数が施されるため

0 と 1 の間で値をとります。

$t_n \log y_n + (1 - t_n) \log(1 - y_n)$ に注目してもう少し掘り下げてみましょう。二値交差エントロピーの最小化問題を考えますが、式全体に $-$ が付いているため、この項目は最大化を行いたいことがわかります。$t_n = 0$ のときには、

$$\log(1 - y_n)$$

となり、y_n が 0 に近いほど値が大きくなります。$t_n = 1$ のときには

$$\log y_n$$

となり、y_n が 1 に近いほど値が大きくなります。

この確率の考え方や log が登場する背景などは、確率や情報量を学ぶと、より理解が深まります。ただし、現状では 2 クラスの分類では目的関数として二値交差エントロピーを利用している、という程度で問題ありません。確率や情報量の数学はこれまでの数学よりも多少難度が高いため、余裕が出てきた頃に理解を深めてください。

交差エントロピー

交差エントロピー（cross entropy）は、多クラスの分類問題を解きたい際によく用いられる目的関数であり、二値交差エントロピーを拡張したものです。

例として K クラスの分類問題を考えてみましょう。ある入力 $\mathbf{x}$ が与えられたとき、ニューラルネットワークの出力層に K 個のノードがあり、それぞれがこの入力が k 番目のクラスに属する確率

$$y_k = p(y = k|\mathbf{x})$$

を表しているとします。これは、入力 $\mathbf{x}$ が与えられたという**条件のもとで**、予測クラスを意味する y が k であるような確率を表す**条件付き確率**です。

ここで、$\mathbf{x}$ が所属するクラスの正解が

$$\mathbf{t} = \begin{bmatrix} t_1 & t_2 & \dots & t_K \end{bmatrix}^{\mathrm{T}}$$

というベクトルで与えられているとします。ただし、このベクトルは $t_k\,(k = 1, 2, \dots, K)$ **のいずれか 1 つだけが 1 であり、それ以外は 0 であるようなベクトル**であるとします。これを**ワンホットベクトル**と呼びます。

そして、この 1 つだけ値が 1 となっている要素は、その要素のインデックスに対応したクラスが正解であることを意味します。たとえば $t_3 = 1$ であれば、3 つ目のクラス（3 というインデックスに対応するクラス）が正解であるというわけです。

以上を用いて、交差エントロピーは次のように定義されます。

$$-\sum_{k=1}^{K} t_k \log y_k$$

これは、t_k が $k = 1, \ldots, K$ のうち正解クラスである 1 つの k の値のみ 1 となるので、正解クラスであるような k での $\log y_k$ を取り出して -1 を掛けているのと同じです。

また、N 個すべてのサンプルを考慮すると、交差エントロピーは

$$L = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k} \log y_{n,k}$$

となります。

2.3 逆伝播

数値例で紹介したとおり、パラメータの値を決めると順伝播により予測値を計算できます。この予測値に対して目的関数を計算でき、どの程度の間違いが残されているかを把握することができるので、この情報に基づいて、より良い方向へパラメータを調整していきたいと思います。この節では、パラメータの最適化を考えていきましょう。

目的関数の勾配が求まらない問題

重回帰分析と同じように進めるのであれば、目的関数をパラメータで微分して勾配ベクトルを求め、その勾配ベクトルがゼロベクトルとなるように、各パラメータの値を求めればよいでしょう。ただし、前述したように、ここで 1 つ問題があります。それは、今回の目的関数を計算する中で ReLU 関数を採用する場合に、$\max(\cdot, \cdot)$ が含まれていることです。この関数は実際に値を入れていないと関数の値を決定できません。つまり、全体を見通して目的関数の微分を行うことができません。微分を行うことができなければ、パラメータの最適化もできません。

このように、これから紹介するアルゴリズムでは、モデルや目的関数の定式化こそできるものの、パラメータの最適化を考えた段階で、実際に数値を入れてみるまで判断がつかないような構造となっているものもあります。それでは、このような場合、どのように対処しながらパラメータの最適化を行えばよいのでしょうか。

勾配降下法

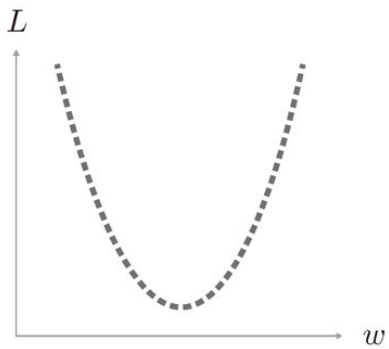
この数値を入れてみるまで判断がつかない構造となっている場合の解決策の 1 つとして、**勾配降下法**という最適化アルゴリズムがあります。このアルゴリズムの考え方はこれまでとは異なる

ため、最初に整理しておきます。

重回帰分析などでは、勾配ベクトルがゼロベクトルとなるようなパラメータを逆算して求めていったため、式変形を行うノート（解析的に解いていく作業）上で最適なパラメータを求める作業が完結するものでした。そのため、$\mathbf{w} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{t}$ に対して、データを当てはめてコンピュータで計算すれば、一度の計算で最適な値を求めることができました。

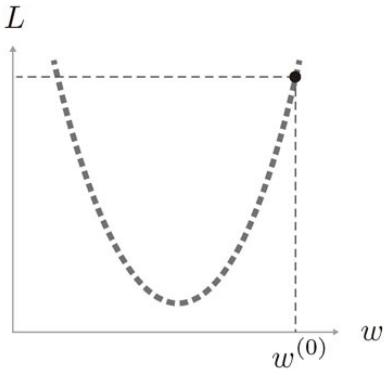
一方、勾配降下法はノート上で完結するものではありません。コンピュータに試行錯誤をさせる中で良いパラメータを求めていきます。司令塔である数学と、優秀な実行者であるコンピュータがタッグを組んで少し泥臭く進めていく方法です。

それでは、具体的なアルゴリズムを紹介していきます。まずは議論を進めるために、下図のような目的関数を考えます。

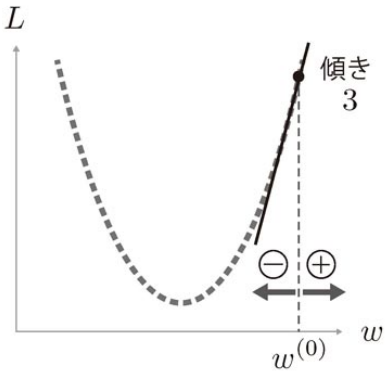


実際の目的関数は上図のようにきれいな二次関数のような形式となっていないこともありますが、まずは議論の本質に専念するために上図のような目的関数で解説を進めます。また、ここで計算を行うコンピュータの立場に立って考えるとわかりやすいのですが、パラメータ w に対してどのような目的関数の値になるかはまったくわかっていないため、点線で書いてある目的関数の形状は神のみぞ知るといったところです。このようなまったくヒントもない中で神のみぞ知る目的関数の値を最小化するようなパラメータ w を求めなければなりません。

勾配降下法では、ランダムに w の値を仮で決め、その目的関数の値を計算します。この仮の値を $w^{(0)}$ のように上付きの添字で更新された順番で管理しましょう。たとえ仮の値でも w が決定できると、分類を行うための直線もしくは（超）平面をもとに予測値を計算できます。この予測値を計算できれば、目的関数 L の値も当然ですが計算できます。



ランダムに決定したパラメータの値なので、目的関数の値が最小値をとることはありませんが、コンピュータにとってはまったく形の見えていなかった目的関数を形をつかむための手がかりが 1 つ増えるという進展がありました。さらに、論理が飛躍しますが、ニューラルネットワークでは全体での微分こそできないものの、具体的に値が入ると $\max(\cdot,\cdot)$ の計算ができるので、この点における勾配を求めることができます。この計算は後ほど紹介するため、ひとまずここでは計算できるものであると受け入れて進めましょう。



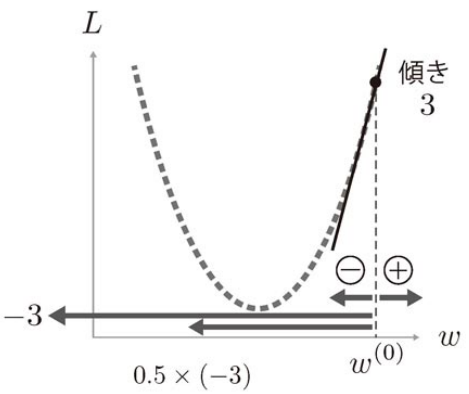
そして、上図のようにたとえば、接線の傾きが 3 のように求まるとします。この接線の傾きが求まりましたが、これまでは「傾きが 0」といった使い方しか紹介していませんでした。ここではさらに発展的な使い方を紹介します。それは、次の点を求めるための「方向」と「移動量」を勾配（接線の傾き）により決められることです。

まず、方向について考えます。ランダムに決めた $w^{(0)}$ から目的関数を最小化するためには、$+$ 方向と $-$ 方向のどちらを選択すればよいでしょうか。神のみぞ知る目的関数の形がわかっている場合であれば、$-$ 方向であることは明らかですが、コンピュータにとってはまったくわかりません。そこで、今回の**接線の傾きの符号**に着目します。傾きが正であるということは、$+$ 方向に進めると目的関数はさらに大きな値となってしまい、逆に $-$ 方向に進めると目的関数の値は

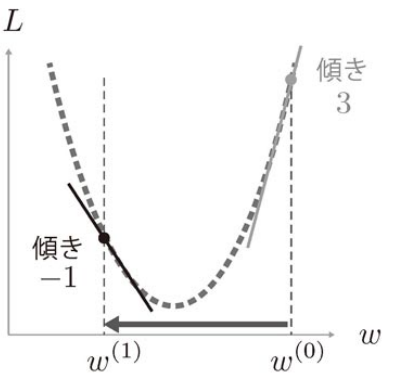
小さくできることが推測できるはずです。そこで、戦略として接線の傾きで得られた符号と**逆方向**に進む意思決定をします。

方向が決まったら、次はどのくらい移動を行うかの「移動量」を考えます。傾きが 3 の場合と傾きが 1 の場合に同じような量を移動せずに、その傾きの量を移動量に反映させることを考えます。傾きが大きいときは目的関数の最小値から遠いと考えて大きく移動し、傾きが小さいときは目的関数の最小値に近いと考えて小さく移動させるわけです。そのため、接線の傾きが 3 のときは、「負の方向に現状の位置から 3 移動する」といったように方向と移動量を決定します。

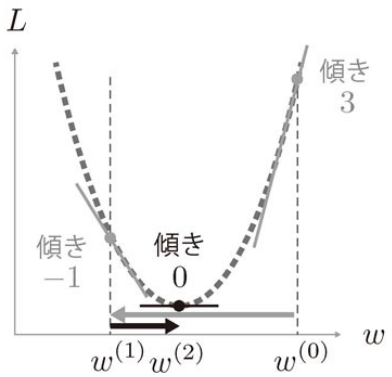
ただし、下図に示すように、実際の座標の値に対して接線の傾きが大きすぎるために移動しすぎてしまうこともあります。これでは目的関数が最小となるようなパラメータに収束していくことができません。この問題に対処するために、**学習係数** ρ と呼ばれる定数を移動量の重みとして導入します。たとえば学習係数を $\rho = 0.5$ とした場合、移動量が $0.5 \times 3 = 1.5$ のように小さく抑制されます。適切な学習係数の値は問題設定によって異なるため、それぞれに調整が必要ですが、この学習係数の導入によってパラメータの値が収束しない問題に対処できます。



接線の傾きを利用して移動後の新しいパラメータの値を $w^{(1)}$ とし、またこの点に関しても微分を行い、傾きの値を求めます。



これを何度も繰り返すことで、損失関数が最小となる点では最終的に傾きが 0 となるため、傾き 0 が得られたら計算を終了します。コンピュータで計算する際には厳密に 0 となることは基本的にないため、$\varepsilon = 10^{-3}$ のように十分小さな値として決めた ε を用いて、傾き$<\varepsilon$ となれば終了、というように現実的なラインを決めるのが一般的です。また、繰り返しの回数を 100 回のように決めておき、所定回数で更新を行った最終的なパラメータの値を用いることも、実用ではよくあります。



ここまでが勾配降下法のアルゴリズムの考え方です。このように、全体を見通しての微分を行うことはできなくても、パラメータを仮で決めたならば勾配を計算できる性質を利用したパラメータの最適化でした。この考え方は機械学習をはじめとした最適化ではよく登場するので、ぜひ覚えておきましょう。

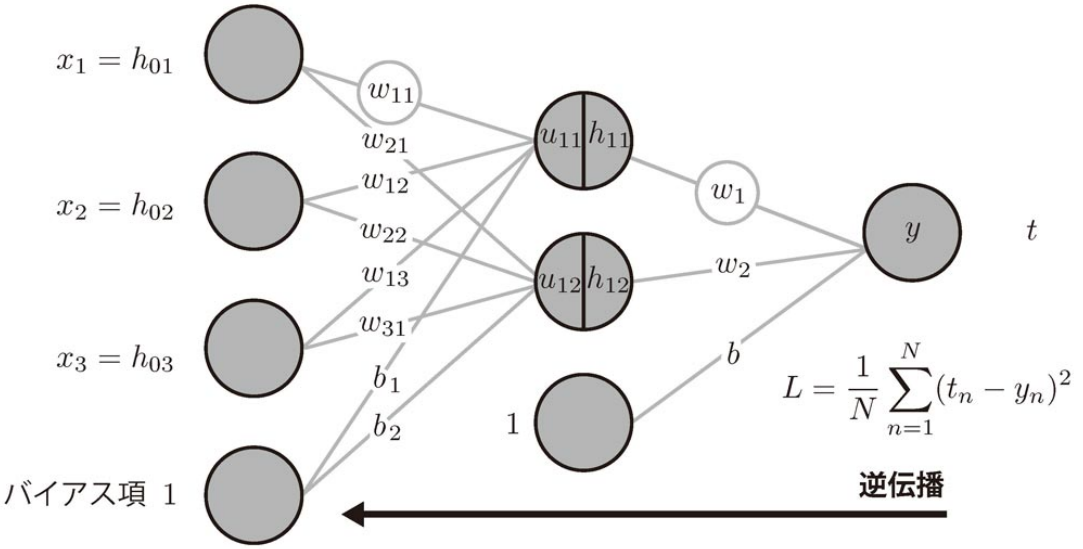
勾配降下法を数式として整理します。パラメータは実際には複数の値を持つため、$\mathbf{w}$ のようにベクトルで表現するとしましょう。k 番目のパラメータ $\mathbf{w}^{(k)}$ において、目的関数の値 $L(\mathbf{w}^{(k)})$ を求め、その勾配が $\dfrac{\partial}{\partial \mathbf{w}} L(\mathbf{w}^{(k)})$ のように表されます。逆方向は $-\dfrac{\partial}{\partial \mathbf{w}} L(\mathbf{w}^{(k)})$ のように $-$ を付けて表し、この値に学習係数 ρ を掛けて移動するため、k 番目から $k+1$ 番目にパラメータを更新するときは、

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} - \rho \frac{\partial}{\partial \mathbf{w}} L(\mathbf{w}^{(k)})$$

のようになります。なお、初期値 $\mathbf{w}^{(0)}$ がないと、この計算は始めることができないため、初期値はランダムに決めていました。初期値を $\mathbf{w}^{(0)} = \mathbf{0}$ のように決めることもあります。

目的関数の微分

勾配降下法では各パラメータの値が決められている場合であれば、その点における勾配を求めてパラメータを更新していくことで、パラメータの調整ができます。ニューラルネットワークにおける勾配の導出を見ていきましょう。まずは、次の図のように各パラメータに文字を割り振りました。



ここではこの w_1 と w_{11} の 2 つの点における勾配を計算します。同じ層のパラメータは同じ手順で勾配を求めることができます。

まずは、目的関数に対する w_{11} の勾配 $\dfrac{\partial L}{\partial w_{11}}$ を求めます。目的関数 L から微分を行う w_{11} までの間を式で整理しておきましょう。

$$L = \frac{1}{N}\sum_{n=1}^{N}(t_n - y_n)^2$$

$$y_n = w_1 h_{11} + w_2 h_{12} + b$$

これより、連鎖律を用いると、

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial w_1}$$

のようにそれぞれの式に対応する微分へと切り分けることができます。それぞれの微分を合成関数の微分も用いながら行うと、

$$\frac{\partial L}{\partial y_n} = -\frac{2}{N}\sum_{n=1}^{N}(t_n - y_n)$$

$$\frac{\partial y_n}{\partial w_1} = h_{11}$$

となります。数値例で求めた値を代入すると、

$$\frac{\partial L}{\partial y_n} = -\frac{2}{1}(20 - 33) = 26$$

$$\frac{\partial y_n}{\partial w_1} = h_{11}$$

となるため、最終的な勾配の値が

$$\frac{\partial L}{\partial w_1} = 26 \times 11 = 286$$

となります。学習係数 $\rho = 0.001$ の場合、この勾配の値を用いて、更新後の w_{11} は、

$$w_{11} = 3 - 0.01 \times 286 = 2.714$$

となります。

次に、w_{11} に関して微分を行います。こちらも連鎖律を用いて微分の計算を行うため、目的関数 L から w_{11} までの式を整理しておきます。

$$\begin{aligned}
L &= \frac{1}{N}\sum_{n=1}^{N}(t_n - y_n)^2 \\
y_n &= w_1 h_{11} + w_2 h_{12} + b \\
h_{11} &= a(u_{11}) \\
u_{11} &= w_{11} h_{01} + w_{12} h_{02} + w_{13} h_{03} + b_1
\end{aligned}$$

これより、連鎖律を用いると、

$$\begin{aligned}
\frac{\partial L}{\partial w_{11}} &= \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial w_{11}} \\
&= \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial h_{11}}\frac{\partial h_{11}}{\partial w_{11}} \\
&= \frac{\partial L}{\partial y_n}\frac{\partial y_n}{\partial h_{11}}\frac{\partial h_{11}}{\partial u_{11}}\frac{\partial u_{11}}{\partial w_{11}}
\end{aligned}$$

のように分解できます。$\dfrac{\partial L}{\partial y_n}$ はすでに求めているため、それ以外の項の微分は

$$\begin{aligned}
\frac{\partial y_n}{\partial h_{11}} &= w_1 \\
\frac{\partial h_{11}}{\partial u_{11}} &= a'(u_{11}) \\
\frac{\partial u_{11}}{\partial w_{11}} &= h_{01}
\end{aligned}$$

となります。ここで、ReLU 関数 $a(u)$ は

$$a(u) = \max(0, u) = \begin{cases} u & u > 0 \\ 0 & u \leq 0 \end{cases}$$

であるため、その導関数 $a'(u)$ は、

$$a'(u) = \begin{cases} 1 & u > 0 \\ 0 & u \leq 0 \end{cases}$$

となります。これより、各項目に値を代入すると、

$$\frac{\partial y_n}{\partial h_{11}} = 3$$
$$\frac{\partial h_{11}}{\partial u_{11}} = a'(11) = 1$$
$$\frac{\partial u_{11}}{\partial w_{11}} = 2$$

が得られます。ここで、w_1 は更新前のパラメータを使用していることに注意してください。すべてのパラメータの更新は同時に行われるため、w_{11} が更新されるまで w_1 の更新も行われません。これより、全体の微分係数は、

$$\frac{\partial L}{\partial w_{11}} = 26 \times 3 \times 1 \times 2 = 156$$

となります。この値を用いて、更新後のパラメータは

$$w_{11} = 3 - 0.001 \times 156 = 2.844$$

となります。

層がさらに深くなると、この計算がさらに長くなっていくイメージです。ただし、今回の計算でもわかるとおり、出力に近いところから計算していくと、前に戻っていくにつれて、すでに計算を行っていて、流用できる項目がありました。この項目を流用できれば、計算量も削減できます。このように、特定の共通する項目を逆方向から伝播していくと計算量を削減できます。この方法は**誤差逆伝播法**として知られています。予測値の計算は順方向に値を伝播させていきましたが、パラメータの更新のために必要な勾配の計算が逆方向に値を伝播させるため、**逆伝播**と呼ばれています。

2.4 学習の工夫

勾配が消失する問題

活性化関数について初めに触れた際、シグモイド関数には勾配消失という現象が起きやすくなるという問題があり、現在はあまり使われていないと述べました。その理由についてもう少し詳しく見ていきましょう。

まずはシグモイド関数の導関数を導出します。ここで必要な微分の公式として、

$$\{\exp(ax)\}' = a\exp(ax)$$
$$\left(\frac{1}{x}\right)' = -\frac{1}{x^2}$$

の 2 つを覚えておきましょう。これらを用いてシグモイド関数の微分は

$$
\begin{aligned}
\frac{\partial a(u)}{\partial u} &= \frac{\partial}{\partial u}\left(\frac{1}{1+\exp(-u)}\right) \\
&= \frac{-1\times\big(-\exp(-u)\big)}{\big(1+\exp(-u)\big)^2} \\
&= \frac{\exp(-u)}{\big(1+\exp(-u)\big)^2} \\
&= \frac{1}{1+\exp(-u)}\times\frac{\exp(-u)}{1+\exp(-u)} \\
&= \frac{1}{1+\exp(-u)}\times\frac{1+\exp(-u)-1}{1+\exp(-u)} \\
&= \frac{1}{1+\exp(-u)}\times\left(1-\frac{1}{1+\exp(-u)}\right) \\
&= a(u)\big(1-a(u)\big)
\end{aligned}
$$

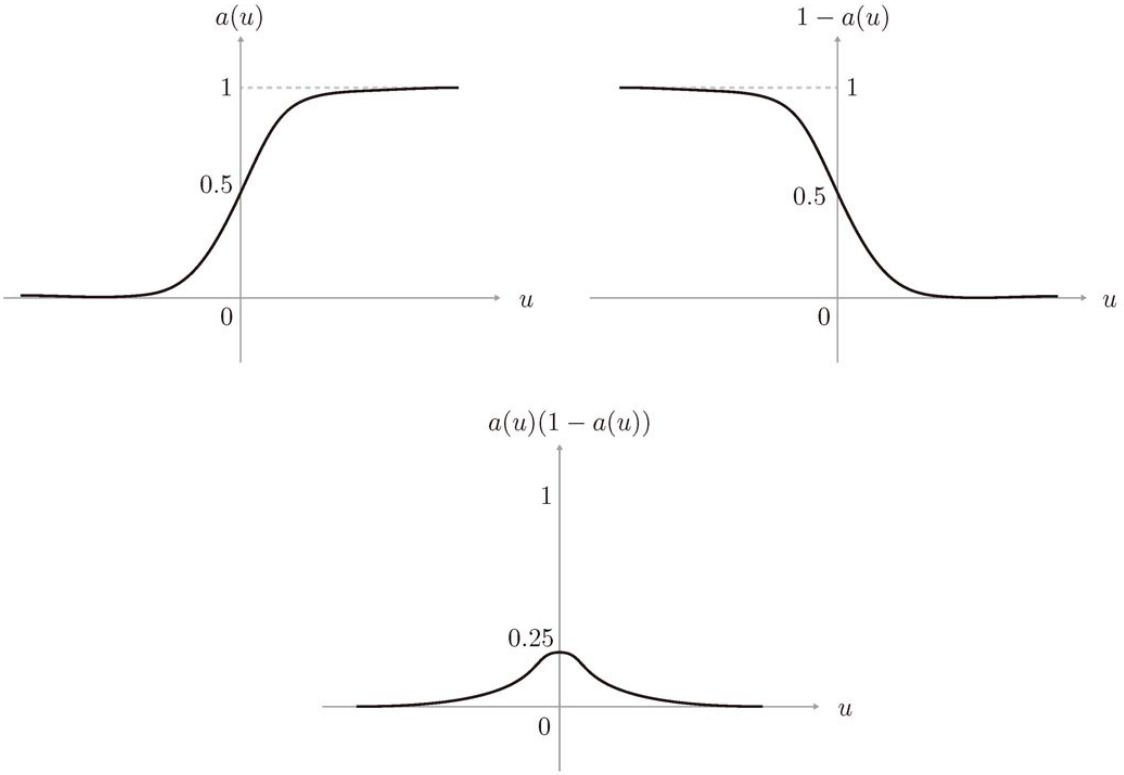
となります。後半はテクニックが必要となりましたが、要するに、シグモイド関数の微分は $a(u)$ と $\big(1-a(u)\big)$ の乗算により表現できるということです。

$$
\begin{aligned}
a(u) &= \frac{1}{1+\exp(-u)} \\
a'(u) &= a(u)(1-a(u))
\end{aligned}
$$

さて、この導関数の値を入力変数に関してプロットしてみると、次ページの図のようになります。

この図の上 2 つは、導関数を構成する 2 つの部分、$a(u)$ および $1-a(u)$ の値を別々にプロットしたもので、中央下の図は全体の導関数の値となります。中央下の導関数の形から明らかなように、入力が原点から遠くなるにつれて勾配の値がどんどん小さくなり、0 に漸近していくことがわかります。

各パラメータの更新量を求めるには、前節で説明したように、**そのパラメータよりも先のすべての関数の勾配を掛け合わせる**必要がありました。このとき、活性化関数にシグモイド関数を用いていると、勾配は必ず**最大でも 0.25** という値にしかなりません。つまり、シグモイド関数がニューラルネットワーク中に現れるたびに、目的関数の勾配は**多くとも** 0.25 倍されてしまいます。これは、層数が増えていけばいくほど、この最大でも 0.25 にしかならないような値が、繰り返し掛け合わされることになってしまい、入力に近い層に流れていく勾配はどんどん 0 に近づいていってしまいます。



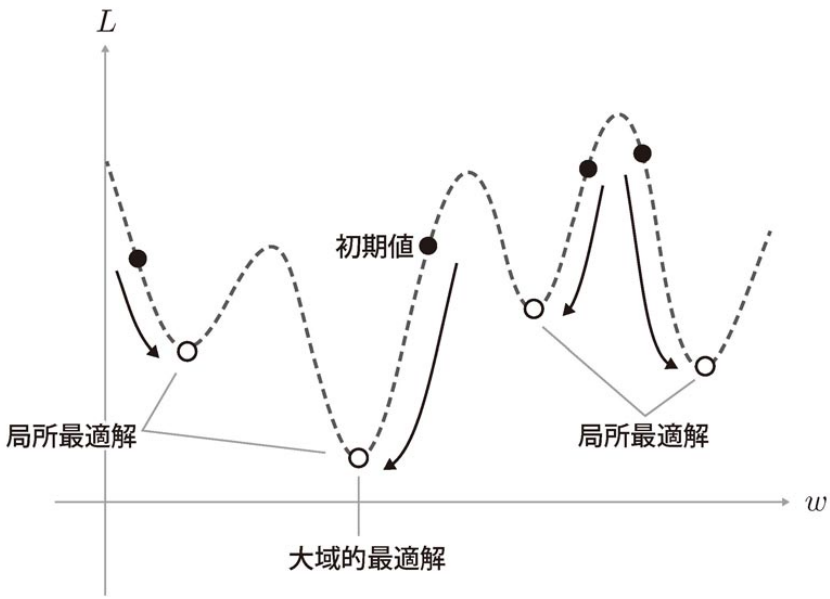
具体例を見てみましょう。これまでは 3 層のニューラルネットワークを用いて説明していましたが、4 層の場合を考えてみます。すると、一番入力に近い線形変換のパラメータの勾配は、多くとも目的関数の勾配を $0.25 \times 0.25 = 0.0625$ 倍したものということになります。層数が 1 つ増えるたびに、指数的に勾配が小さくなるということがよくわかります。

ディープラーニングでは、4 層よりもさらに多くの層を積み重ねたニューラルネットワークが用いられます。そうすると、活性化関数としてシグモイド関数を使用した場合、**目的関数の勾配が入力に近い関数が持つパラメータへほぼまったく伝わらなくなってしまいます**。あまりにも小さな勾配しか伝わってこなくなると、パラメータの更新量がほとんど 0 になるため、どんなに目的関数が大きな値になっていても、入力層に近い関数が持つパラメータは変化しなくなります。つまり初期化時からほとんど値が変わらなくなるということになり、学習が行われていないという状態になるわけです。これが**勾配消失**であり、長らく深い（十数層を超える）ニューラルネットワークの学習が困難であった要因の 1 つでした。

この解決策として、ReLU 関数が提案されました。ReLU 関数の勾配は**最大で 1** であり、ReLU 関数を通すことで勾配が消失しないことがわかります。多層のニューラルネットワークに対する学習も勾配消失を回避しながら行えるようになっています。

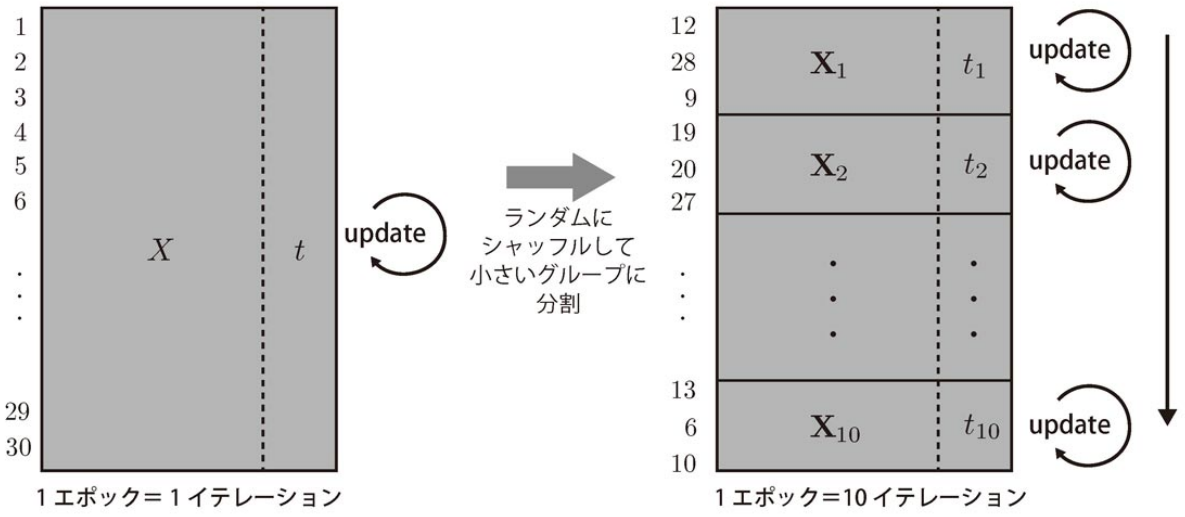
ミニバッチ学習

目的関数の最適化を考えた際には、二次関数のように最小となる点が 1 つの場合を前提にしていましたが、実際には下図のように局所的に最小な点をいくつも持つ関数が想定されます。



上図からわかるとおり、勾配降下法を用いると、初期値によって収束する解が異なります。それぞれの局所的に見たときの局所最適解に収束するのではなく、本来は全体を見渡したときの最適解である大域的最適解となるパラメータを求めたいことがわかります。ただし残念ながら、今の研究では一部の理想的な条件という前提を除き、すべての関数に対する大域的最適解を求める方法はありません。そのため、良くない局所最適解に陥るのを避ける方法を考えることにします。

すべてのデータを用いて学習を行うと、初期値に依存してある 1 つの局所最適解に陥りますが、どこかにランダムな要素を持たせることで、毎回同じ点に陥ることを避けられます。ニューラルネットワークでは、学習に使用するデータをすべてのサンプルを用いるのではなく、次ページの図のようにサンプルの一部をランダムに抽出して用いる**ミニバッチ学習**を採用するのが大半です。



ミニバッチ学習では、以下の手順で訓練を行います。

1. 訓練データセットから一様ランダムに N_b (> 0) 個のデータを抽出する
2. その N_b 個のデータをまとめてニューラルネットワークに入力し、それぞれのデータに対する目的関数の値を計算する
3. N_b 個の目的関数の値の平均をとる
4. この平均の値に対する各パラメータの勾配を求める
5. 求めた勾配を使ってパラメータを更新する

上図では、全体のサンプル数 $N = 30$ で、バッチサイズ $N_b = 3$ の例です。上図にもあるように、バッチサイズとは一度の更新に用いるデータのサンプル数です。

勾配降下法では、データを用いてパラメータを 1 度更新するまでのことを **1 イテレーション**（iteration）と呼びます。バッチ学習では訓練データセットのデータ全体を 1 度の更新に一気に用いるため、1 エポック＝ 1 イテレーションとなります。一方、たとえばデータセットを 10 個のグループに分割してグループごとに更新を行うミニバッチ学習の場合は、1 エポック＝ 10 イテレーションとなります。

このようなミニバッチ学習を用いた勾配降下法は特に、**確率的勾配降下法**（stocastic gradient descent：**SGD**）と呼ばれます。現在多くのニューラルネットワークの最適化手法は、この SGD をベースとした手法となっています。

第3章 PyTorch（基礎編）

本章では、ディープラーニングフレームワーク PyTorch の基本的な機能を使って、PyTorrch を使ったネットワークの訓練がどのような処理で構成されているかを簡潔に紹介します。

本書で使用する Docker イメージにはあらかじめ PyTorch がインストールされているため、環境構築の必要はありません。もし皆さんの PC に PyTorch をインストールしたい場合は、PyTorch 公式ページ（https://pytorch.org/）の Quick Start に書かれている手順を参考にインストールしてください。

3.1 PyTorchの概要

PyTorchとは

PyTorch はオープンソースのディープラーニングフレームワークです。GitHub リポジトリ pytorch（`https://github.com/pytorch/pytorch`）で活発に開発が行われています。**オープンソースソフトウェア**（OSS、Open Source Software）なので、誰でも PyTorch のすべてのコードを見ることができます。

ディープラーニングフレームワークとは、これまでの章で説明してきたようなニューラルネットワークの設計・訓練・評価などに必要な一連の実装を容易にするためのフレームワークです。特に層数の多いニューラルネットワークを GPU などを用いて高速に訓練できるようにする機能を備えたものを指すことが多く、PyTorch のほかにもさまざまなディープラーニングフレームワークが発表されています。

他のフレームワークと比較して、PyTorch は研究開発の分野で盛んに使用されているという特徴があります。新技術の実装やテストを容易にできるという点が、PyTorch の大きな強みです。

PyTorchの準備

PyTorch は本書で提供する Docker イメージにあらかじめインストールされているので、ノートブックを開くとすでに PyTorch のモジュールを読み込む準備が整っています。最初に

PyTorch を import して PyTorch を使う準備をします。Docker の使い方については、第 5 章を参照してください。

```
import torch
```

PyTorch のバージョンを確認します。

```
torch.__version__
```

```
'1.3.1'
```

警告の非表示

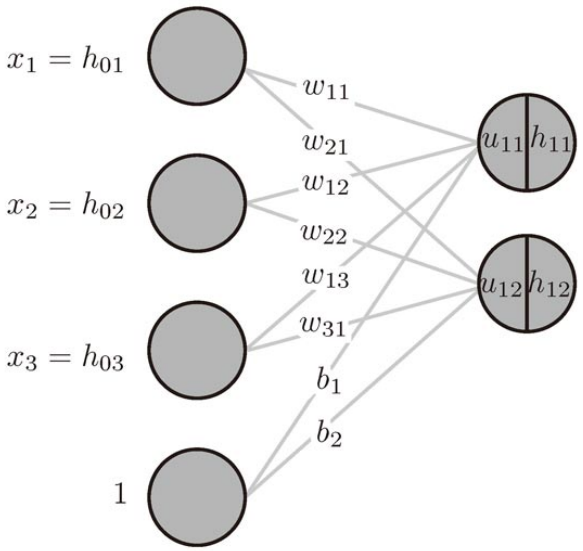
PyTorch や他のパッケージを使用する際、バージョンの変更などで実行には支障のない警告（warnings）が表示されることがあります。警告の内容を読めばエラーではないことがわかりますが、当面は気にする必要がないため、Jupyter Notebook 上での警告を非表示に設定しておきましょう。Jupyter Notebook の使い方については、第 5 章を参照してください。

```
import warnings

# 警告の非表示
warnings.filterwarnings('ignore')
```

3.2 ネットワークの定義

全結合層の定義

数学においてもプログラミングにおいても、全結合層（fully-connected layer）では、2 層で 1 セットとして扱います。まずは、3 ノードの入力層と 2 ノードの出力層の部分を表現していきましょう。




```
import torch.nn as nn
```

ノードの数が 3 から 2 のリンクを fc（fully-connected layer の略）として、以下のように宣言します。

```
fc = nn.Linear(3, 2)
```

これだけで完了です。nn.Linear は、重回帰分析の**線形結合**という意味です。

宣言したリンクの**重み**（W）と**バイアス**（b）はランダムに初期化されています。

```
fc.weight
```

```
Parameter containing:
tensor([[-0.2076,  0.0534,  0.3774],
        [ 0.4663, -0.1719, -0.5624]], requires_grad=True)
```

```
fc.bias
```

```
Parameter containing:
tensor([-0.2617, -0.1796], requires_grad=True)
```

これらのパラメータを最適化する上での初期値として使用します。

乱数のシードを固定して再現性を確保

ここで、**パラメータ（重みやバイアス）の値がランダムに初期化される**ため、実行するたびに結果が異なってしまうという問題が存在します。これではたとえば他の誰かの成果物を確認する際、再現性がとれず、結果が異なってしまうことになります。

そこで、**乱数のシードを固定する**という方法を使います。これで乱数の作られる過程が固定され、**再現性の確保**ができます。

実際に乱数のシードを固定してみましょう。

```
torch.manual_seed(1)
```

乱数のシードを固定した後で、重みを確認してみます。

```
fc = nn.Linear(3, 2)

fc.weight
```

```
Parameter containing:
tensor([[ 0.2975, -0.2548, -0.1119],
        [ 0.2710, -0.5435,  0.3462]], requires_grad=True)
```

```
fc.bias
```

```
Parameter containing:
tensor([-0.1188,  0.2937], requires_grad=True)
```

この値と同じになっていれば、乱数のシードをうまく固定できていることがわかります。

線形変換

手計算で練習した、u の値を計算してみましょう。u の値を求める際の線形変換は `torch.nn.Linear` の call メソッドとして定義されています。

計算の前に、PyTorch で使用するデータ型について初学者の人はつまずきやすいので、確認し

ておきます。PyTorch を使用する際は、以下のように torch.tensor により PyTorch の標準で使用するデータ型に変換します。

```
# リストからPyTorchのTensorへ変換
x = torch.tensor([[1, 2, 3]])
x
```

```
tensor([[1, 2, 3]])
```

```
# 型の確認
type(x)
```

```
torch.Tensor
```

上記のように torch.Tensor という形式が標準になるので、覚えておきましょう。しかし、まだ落とし穴が存在します。線形変換は nn.Linear の call メソッドとして定義されているため、以下のように計算を行います。

```
# 線形変換
u = fc(x)
```

```
RuntimeError                              Traceback (most recent call last)
<ipython-input-15-c771fe3d55f7> in <module>
      1 # 線形変換
----> 2 u = fc(x)
 …
RuntimeError: Expected object of scalar type Float but got scalar type Long for argument #2 'mat➪
1' in call to _th_addmm
```

しかし、ここでエラーが発生します。この原因は入力値 x のデータ型にあります。torch.Tensor の形式で正しいのですが、さらに詳細なデータ型の設定が必要になります。

```
# 詳細なデータ型の確認
x.dtype
```

```
torch.int64
```

この結果のとおり、入力値 x のデータ型が int64 になっています。これは整数値を 64 ビットで扱う形式です。入力値は実数値で 32 ビットとするのが基本なので、このデータ型を変換します。

```
# float32に変換
x = torch.tensor([[1, 2, 3]], dtype=torch.float32)
x
```

```
tensor([[1., 2., 3.]])
```

```
# データ型の確認
x.dtype
```

```
torch.float32
```

これで再度、線形変換の計算を行います。

```
u = fc(x)
u
```

```
tensor([[-0.6667,  0.5164]], grad_fn=<AddmmBackward>)
```

このように、線形変換後の値を求めることができました。

非線形変換

ここでは活性化関数として **ReLU 関数**を採用します。PyTorch で使用する関数はすべて torch.nn.functions に含まれています。これを F として読み込みましょう。

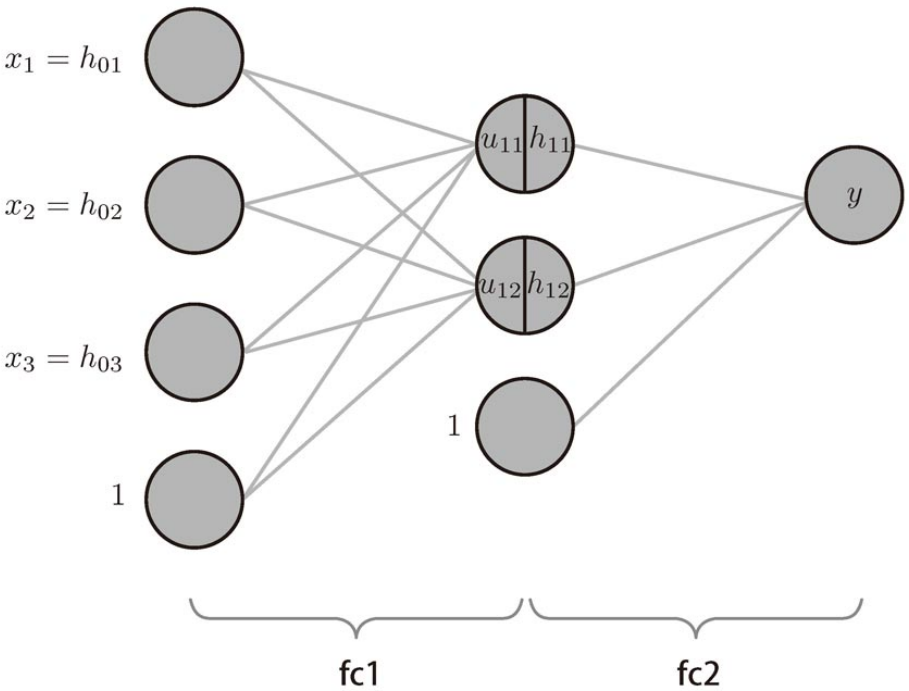
```
import torch.nn.functional as F

# ReLU関数
h = F.relu(u)
h
```

```
tensor([[0.0000, 0.5164]], grad_fn=<ReluBackward0>)
```

結果のとおり、正の値をそのまま、負の値を 0 と出力する ReLU 関数が正しく適用されていることがわかります。

数値例

下図に示すようなニューラルネットワークを定義して、$x = [1, 2, 3]$ に対する予測値 y を求めてみましょう（乱数のシードは 1 で固定します）。



コードは以下のとおりです。

```
# 乱数のシードを固定
torch.manual_seed(1)

# 入力値の定義
x = torch.tensor([[1, 2, 3]], dtype=torch.float32)
x
```

```
tensor([[1., 2., 3.]])
```

```
# 全結合層の定義
fc1 = nn.Linear(3, 2)
fc2 = nn.Linear(2, 1)

# 線形変換
u1 = fc1(x)
u1
```

```
tensor([[-0.6667,  0.5164]], grad_fn=<AddmmBackward>)
```

```
# 非線形変換
h1 = F.relu(u1)

# 線形変換
y = fc2(h1)
y
```

```
tensor([[0.1514]], grad_fn=<AddmmBackward>)
```

損失関数

出力の予測値 y が計算できたので、損失関数の値も求めます。損失関数の計算には目標値 t が必要なので、今回は $t = 1$ とします。今回の問題設定を回帰とした場合、損失関数は平均二乗誤差（MSE：mean squared error）を用います。また、目標値のデータ型も適切に設定しないとエラーが出るので、回帰と分類でそれぞれ以下のように覚えておきましょう。

- **回帰**：float32
- **分類**：int64

```
# 目標値
t = torch.tensor([[1]], dtype=torch.float32)
t
```

```
tensor([[1.]])
```

```
# 平均二乗誤差
loss = F.mse_loss(t, y)
loss
```

```
tensor(0.7201, grad_fn=<MeanBackward0>)
```

このように損失関数も PyTorch では torch.nn.functions(F) の中に定義されています。

これで一連の流れを理解できましたが、PyTorch ではネットワーク内での計算を行う際に、torch.nn（nn）を用いる場合と、torch.nn.functional（F）を用いる場合の 2 通りの選択肢がありました。この違いは以下のとおりです。

- nn：パラメータを持つ
- F：パラメータを持たない

nn.Linear で定義した全結合層は weight や bias といったデータから調整すべきパラメータを持っていました。それに対して、ReLU 関数や平均二乗誤差は関数の処理内でパラメータを持っていません。PyTorch ではこの基準で nn か F のどちらに格納されているか決まっているため、この選択の基準を理解しておくと、ほしい機能があるときに探しやすくなります。

3.3 学習

それでは、PyTorch での順伝播に関する一連の流れを理解できたところで、データセットが与えられたときにネットワークを学習させるまでの実践的な一連の流れを紹介します。次章で紹介する PyTorch Lightning を用いると、これから紹介する流れをより簡単に記述できますが、内部の流れをまずは理解しておかないと PyTorch Lightning での記述も理解できません。基礎をしっかり押さえていきましょう。流れは以下のとおりです。

- **Step 1**：データセットを準備
- **Step 2**： ネットワークを定義
- **Step 3**： 損失関数を選択
- **Step 4**：最適化手法を選択
- **Step 5**：ネットワークを学習

データセットを準備

この例では、scikit-learn に用意されている、「アヤメの品種」を分類する Iris という問題に取り組みます。分類する品種は virginica（0）、versicolor（1）、setosa（2）の 3 種類で、入力変数は以下の 4 つです。

ID	変数名	説明	データ型
0	sepa_length	がく片の長さ	float
1	sepa_width	がく片の幅	float
2	petal_length	花びらの長さ	float
3	petal_width	花びらの幅	float

scikit-learn では sklearn.datasets モジュールにサンプル用のデータセットが豊富に用意されています。ここではその 1 つである load_iris() を用います。

```
from sklearn.datasets import load_iris

# Irisデータセットの読み込み
x, t = load_iris(return_X_y=True)

# サイズの確認
x.shape, t.shape
```

```
((150, 4), (150,))
```

```
# 型の確認
type(x), type(t)
```

```
(numpy.ndarray, numpy.ndarray)
```

```
# データ型の確認
x.dtype, t.dtype
```

```
(dtype('float64'), dtype('int64'))
```

PyTorch では torch.Tensor 形式が標準であり、分類の問題ということも考慮して、データ型

もあわせて変換します。

```
x = torch.tensor(x, dtype=torch.float32)
t = torch.tensor(t, dtype=torch.int64)

# 型の確認
type(x), type(t)
```

```
(torch.Tensor, torch.Tensor)
```

```
# データ型の確認
x.dtype, t.dtype
```

```
(torch.float32, torch.int64)
```

PyTorch では、学習時に使用するデータ x と t を 1 つのオブジェクト dataset にまとめます。TensorDataset を使用して dataset に格納しましょう。

```
from torch.utils.data import TensorDataset

# 入力値と目標値をまとめて、1つのオブジェクトdatasetに変換
dataset = TensorDataset(x, t)
dataset
```

```
<torch.utils.data.dataset.TensorDataset at 0x129096160>
```

```
# 型の確認
type(dataset)
```

```
torch.utils.data.dataset.TensorDataset
```

```
#（入力変数，教師データ）のようにタプルで格納されている
dataset[0]
```

```
(tensor([5.1000, 3.5000, 1.4000, 0.2000]), tensor(0))
```

```
# 型の確認
type(dataset[0])
```

```
tuple
```

```
# 1サンプル目の入力値
dataset[0][0]
```

```
tensor([5.1000, 3.5000, 1.4000, 0.2000])
```

```
# 1サンプル目の目標値
dataset[0][1]
```

```
tensor(0)
```

```
# サンプル数はlenで取得可能
len(dataset)
```

```
150
```

次に、学習用とテスト用にデータセットを分割します。これはディープラーニングに限らず機械学習全般で使用する考え方です。

たとえば、受験勉強をするとしましょう。10 年分の過去問を書店で購入し、その過去問を使って学習と実力テストを行うにあたって、次のどちらを選択するでしょうか。

- 10 年分の過去問で学習し、同じ 10 年分の過去問で実力テスト
- 前半 5 年分の過去問で学習し、後半 5 年分の過去問で実力テスト

前者の場合、すでに学習済みの過去問では答えを知っているため、実力を測ることができませ

ん。そのため、後者が実力を測るという意味で正しい方法になり、機械学習でもモデルの予測性能に関する実力を測るため、**学習データ**（traininig data）と**テストデータ**（test data）に用いるデータセットを分けて学習とテストを行います。英語では train と test であり、train を学習もしくは訓練と日本語訳します。直訳では訓練のほうが適切ですが、多くの記事で**学習**と訳されているため、本書でも train の訳として学習を採用しています。

また、学習データとテストデータだけでなく、**検証データ**（validation data）も含めて 3 分割とするケースも多くあります。

この検証データの存在を理解するために、まず**ハイパーパラメータ**を紹介します。ニューラルネットワークでは重みやバイアスをパラメータとして学習しますが、その前に、**中間層のノード数**や**層の数**などのネットワークの構造に関する変数を決定しないと、パラメータの数も決定できません。このように、パラメータを学習する以前に、決めておかなければ始まらないパラメータのことを、**ハイパーパラメータ**と呼びます。ニューラルネットワークでは、前述したノード数や層の数、最適化手法の学習係数などがハイパーパラメータとして代表的です。層の順番なども一種のハイパーパラメータです。

ここからわかることとして、ニューラルネットワークでは学習したパラメータの値に対する評価だけでなく、ハイパーパラメータに対する評価まで行わなければ不十分であるということです。ニューラルネットワークでは、

- ハイパーパラメータの決定（検証データ）
- パラメータの決定（学習データ）

のように、決定したハイパーパラメータに紐づいたパラメータを、学習データに基づいて最適化します。そして、その結果に対する評価を検証データに基づいて行います。たとえば、あるハイパーパラメータの組み合わせの場合に学習させた結果、検証データに対する損失関数の値が 1.0 と 0.8 だったならば、損失関数の値が 0.8 のほうを、優れたハイパーパラメータの組み合わせとして採用します。つまり、検証データは学習データに基づいて学習した結果に対する評価を行っているように見えて、実はハイパーパラメータの学習に用いているのです。テストデータは一切の学習に用いてはならないため、ハイパーパラメータとパラメータの学習をそれぞれ行う必要のあるニューラルネットワークでは、合計 3 種類のデータに分割するのが一般的です。

それでは、`torch.utils.data.randomsplit` を使用してオリジナルのデータセットを学習データ、検証データ、テストデータに分割しましょう。最初にそれぞれのサンプル数を比率から決定していきます。

```
# 各データセットのサンプル数を決定
# train : val: test = 60% : 20% : 20%
n_train = int(len(dataset) * 0.6)
n_val = int(len(dataset) * 0.2)
n_test = len(dataset) - n_train - n_val

# それぞれのサンプル数を確認
n_train, n_val, n_test
```

```
(90, 30, 30)
```

```
from torch.utils.data import random_split

# ランダムに分割を行うため、シードを固定して再現性を確保
torch.manual_seed(0)

# データセットの分割
train, val, test = random_split(dataset, [n_train, n_val, n_test])

# サンプル数の確認
len(train), len(val), len(test)
```

```
(90, 30, 30)
```

ミニバッチ学習を行う場合には、各データセットからバッチサイズ分のサンプルを取得する必要があります。このサンプル抽出の際、学習時にはランダムにシャッフルをして抽出するなどの工夫があり、自前で実装することもできますが、PyTorch にはランダムシャッフルして抽出するなどの機能を実現するために torch.utils.data.DataLoader が用意されています。

```
# バッチサイズの決定
batch_size = 10

from torch.utils.data import DataLoader

# shuffleはデフォルトでFalseのため、学習データのみTrueに指定
train_loader = DataLoader(train, batch_size, shuffle=True)
val_loader = DataLoader(val, batch_size)
test_loader = DataLoader(test, batch_size)
```

ここまでで、PyTorch によりネットワークを学習させるために必要なデータの準備は完了

です。

ネットワークを学習

今回は入力変数が 4、分類するクラスの数が 3 なので、全結合層 fc1 と fc2 のノードの数を次のように決めます。

- **fc1**：input: 4 => output: 4
- **fc2**：input: 4 => output: 3

PyTorch ではネットワークの記述にクラスを利用します。使用する全結合層などパラメータを含んだ層を__init__() メソッド内に定義し、順伝播の計算を forward() メソッド内に記述します。順伝播の計算の流れは、線形変換（fc1）→非線形変換（ReLU）→線形変換（fc2）→非線形変換（Softmax）とします。分類の場合に必要な Softmax 関数は記述する必要がなく、理由は次項で説明します。前出の例とほぼ同じように記述しますが、PyTorch では x が入力されて、出力にも x を上書きするように記述することが多く、本書でもその記述を採用しています。

```
class Net(nn.Module):

    # 使用するオブジェクトを定義
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(4, 4)
        self.fc2 = nn.Linear(4, 3)

    # 順伝播
    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return x
```

クラス内に定義したネットワーク構造を持つインスタンス net を作成します。

```
# 乱数のシードを固定して再現性を確保
torch.manual_seed(0)

# インスタンス化
net = Net()

# ネットワークの確認
net
```

```
Net(
  (fc1): Linear(in_features=4, out_features=4, bias=True)
  (fc2): Linear(in_features=4, out_features=3, bias=True)
)
```

損失関数を選択

今回は 3 クラスの分類なので、損失関数として**クロスエントロピー**を採用します。ここでクロスエントロピーは、

$$L = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{n,k} \log y_{n,k}$$

のように予測値を対数変換する処理を持ちます。また、この予測値 $y_{n,k}$ の計算の前に活性化関数として Softmax 関数による処理があります。つまり、Softmax → Log の順番で計算を行います。

PyTorch では計算を高速化させるために、Softmax 関数と対数変換を同時に行う関数である `F.log_softmax` が用意されています。Softmax → Log の計算をそれぞれ行うと速度として遅いこと、そして計算後の値が安定しないといった理由で、それぞれを別々に計算するのではなく一度にまとめて計算するほうが優れているようです。そして、PyTorch に用意されている `F.cross_entropy` では、内部の計算に `F.log_softmax` が使われています。したがって、事前に Softmax 関数の計算を行う必要はありません。これがネットワークの定義の部分で Softmax 関数を設けていなかった理由です。

PyTorch では損失関数を `criterion` という名前で定義するのが慣習です。

```
criterion = nn.CrossEntropyLoss()
criterion
```

```
CrossEntropyLoss()
```

最適化手法を選択

ネットワークの学習にあたり使用する最適化手法を選択します。`optimizer` という変数名で定義するのが一般的です。ここでは確率的勾配降下法（SGD）を選択することにします。`optimizer` を定義する際に、引数としてネットワークのパラメータを渡す必要があり、パラメータの取得には `net.parameters()` を用います。もう 1 つの引数として、学習係数（lr：learning rate）も設定します。

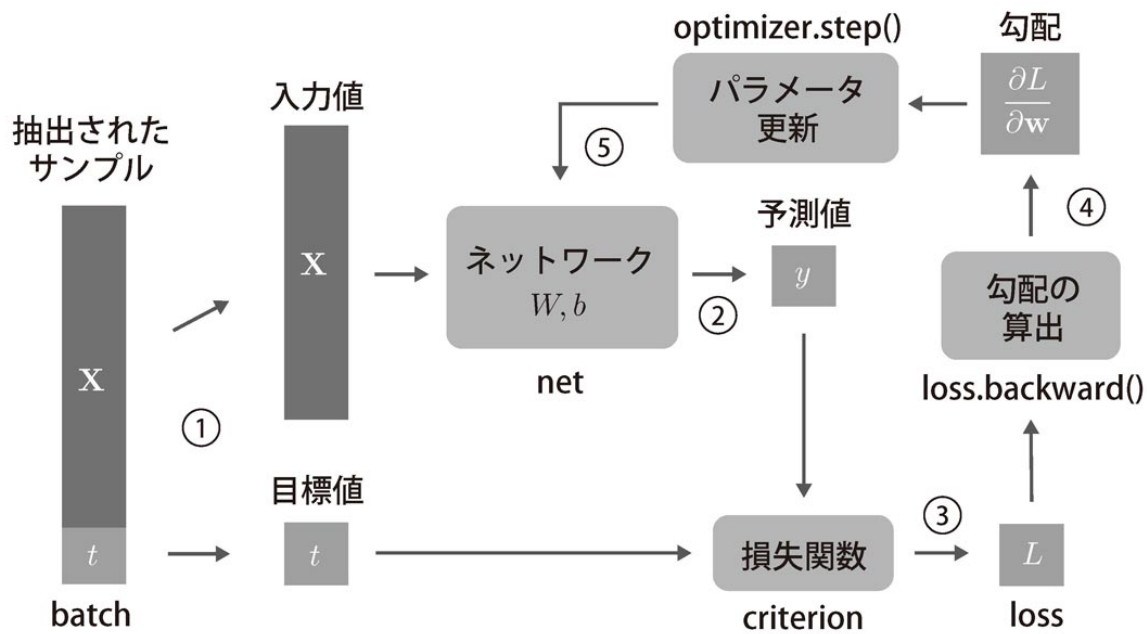
```
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
optimizer
```

```
SGD (
Parameter Group 0
    dampening: 0
    lr: 0.1
    momentum: 0
    nesterov: False
    weight_decay: 0
)
```

ネットワークを学習

ネットワークを学習する際には以下の手順を繰り返します。

1. ミニバッチ単位でサンプル $\mathbf{X}$、t を抽出
2. 現在のパラメータ W、b を利用して、順伝播で予測値 y を算出
3. 目標値 t と予測値 y から損失関数 L を算出
4. 誤差逆伝播法に基づいて各パラメータの勾配 $\frac{\partial L}{\partial \mathbf{w}}$ を算出
5. 勾配の値に基づいて選択した最適化手法によりパラメータ W、b を更新



ディープラーニングのフレームワークによって多少の設定の違いはありますが、基本的には前記の流れを押さえておけばよいでしょう。

それでは、前記の流れを具体的に実装していきます。まず、学習用のデータセット train_loader からバッチサイズ分のサンプルを抽出します。train_loader は DataLoader のオブジェクトであり、次のように for 文を使うことで、各イテレーションにおけるミニバッチを抽出できます。

```
for batch in train_loader:
    batchを利用した処理
```

ここでは挙動を確認するために、for 文において 1 イテレーション分取り出すことができるイテレータ iter を用います。イテレータは内部にリスト型や辞書型の変数を保持して、各要素を順に出力するオブジェクトのことです。このイテレータから順番に値を取り出すときには、next を使います。

```
# バッチサイズ分のサンプルの抽出
batch = next(iter(train_loader))
batch
```

```
[tensor([[6.4000, 3.2000, 5.3000, 2.3000],
         [6.4000, 2.7000, 5.3000, 1.9000],
         [5.9000, 3.0000, 4.2000, 1.5000],
         [6.9000, 3.1000, 5.4000, 2.1000],
         [5.5000, 2.4000, 3.8000, 1.1000],
         [5.6000, 2.7000, 4.2000, 1.3000],
         [5.8000, 2.7000, 5.1000, 1.9000],
         [6.9000, 3.1000, 4.9000, 1.5000],
         [6.3000, 2.3000, 4.4000, 1.3000],
         [7.2000, 3.0000, 5.8000, 1.6000]]),
 tensor([2, 2, 1, 2, 1, 1, 2, 1, 1, 2])]
```

```
# 入力値と目標値に分割
x, t = batch

# 入力値の確認
x
```

```
tensor([[6.4000, 3.2000, 5.3000, 2.3000],
        [6.4000, 2.7000, 5.3000, 1.9000],
        [5.9000, 3.0000, 4.2000, 1.5000],
        [6.9000, 3.1000, 5.4000, 2.1000],
        [5.5000, 2.4000, 3.8000, 1.1000],
        [5.6000, 2.7000, 4.2000, 1.3000],
        [5.8000, 2.7000, 5.1000, 1.9000],
        [6.9000, 3.1000, 4.9000, 1.5000],
        [6.3000, 2.3000, 4.4000, 1.3000],
        [7.2000, 3.0000, 5.8000, 1.6000]])
```

```
# 目標値の確認
t
```

```
tensor([2, 2, 1, 2, 1, 1, 2, 1, 1, 2])
```

このように、バッチサイズ分の入力値と目標値のサンプルを抽出できました。

次に、パラメータの値を用いて予測値を算出します。まず、現状のパラメータの値を確認します。

```
# 全結合層fc1の重み
net.fc1.weight
```

```
Parameter containing:
tensor([[-0.0037,  0.2682, -0.4115, -0.3680],
        [-0.1926,  0.1341, -0.0099,  0.3964],
        [-0.0444,  0.1323, -0.1511, -0.0983],
        [-0.4777, -0.3311, -0.2061,  0.0185]], requires_grad=True)
```

```
# 全結合層fc1のバイアス
net.fc1.bias
```

```
Parameter containing:
tensor([ 0.1977,  0.3000, -0.3390, -0.2177], requires_grad=True)
```

```
# 全結合層fc2の重み
net.fc2.weight
```

```
Parameter containing:
tensor([[ 0.1816,  0.4152, -0.1029,  0.3742],
        [-0.0806,  0.0529,  0.4527, -0.4638],
        [-0.3148, -0.1266, -0.1949,  0.4320]], requires_grad=True)
```

```
# 全結合層fc2のバイアス
net.fc2.bias
```

```
Parameter containing:
tensor([-0.3241, -0.2302, -0.3493], requires_grad=True)
```

これらのパラメータの値とバッチサイズ分抽出したサンプルを用いて予測値を算出します。順伝播の計算は Net クラスの forward() メソッドに記述してあります。

```
# 予測値の算出
y = net.forward(x)
y
```

```
tensor([[-0.1763, -0.2113, -0.3944],
        [-0.2700, -0.2233, -0.3658],
        [-0.2746, -0.2239, -0.3644],
        [-0.2552, -0.2214, -0.3703],
        [-0.3241, -0.2302, -0.3493],
        [-0.3003, -0.2271, -0.3566],
        [-0.2212, -0.2171, -0.3807],
        [-0.3241, -0.2302, -0.3493],
        [-0.3241, -0.2302, -0.3493],
        [-0.3241, -0.2302, -0.3493]], grad_fn=<AddmmBackward>)
```

Net クラスは nn.Module を継承しており、nn.Module では forward() メソッドを call メソッドで呼び出せるため、次の記述でも上記と同じ処理になります。

```
# callメソッドを用いたforwardの計算（推奨）
y = net(x)
y
```

```
tensor([[-0.1763, -0.2113, -0.3944],
        [-0.2700, -0.2233, -0.3658],
        [-0.2746, -0.2239, -0.3644],
        [-0.2552, -0.2214, -0.3703],
        [-0.3241, -0.2302, -0.3493],
        [-0.3003, -0.2271, -0.3566],
        [-0.2212, -0.2171, -0.3807],
        [-0.3241, -0.2302, -0.3493],
        [-0.3241, -0.2302, -0.3493],
        [-0.3241, -0.2302, -0.3493]], grad_fn=<AddmmBackward>)
```

この予測値を用いて、損失関数を算出します。

```
# 損失関数の計算
# criterionのcallメソッドを利用
loss = criterion(y, t)
loss
```

```
tensor(1.1118, grad_fn=<NllLossBackward>)
```

この loss に基づいて勾配の算出を行います。勾配を求める前に、勾配の情報が格納されている場所を確認します。勾配の情報は各パラメータに格納されていますが、初期状態では値は何も格納されていません。

```
# 全結合層fc1の重みに関する勾配
net.fc1.weight.grad

# 全結合層fc1のバイアスに関する勾配
net.fc1.bias.grad

# 全結合層fc2の重みに関する勾配
net.fc2.weight.grad

# 全結合層fc2のバイアスに関する勾配
net.fc2.bias.grad
```

損失関数の勾配を以下のように求めます。

```
# 勾配の算出
loss.backward()
```

この算出後に、勾配の情報を確認します。

```
# 全結合層fc1の重みに関する勾配
net.fc1.weight.grad
```

```
tensor([[0.0000, 0.0000, 0.0000, 0.0000],
        [0.7165, 0.3319, 0.5857, 0.2248],
        [0.0000, 0.0000, 0.0000, 0.0000],
        [0.0000, 0.0000, 0.0000, 0.0000]])
```

```
# 全結合層fc1のバイアスに関する勾配
net.fc1.bias.grad
```

```
tensor([0.0000, 0.1137, 0.0000, 0.0000])
```

```
# 全結合層fc2の重みに関する勾配
net.fc2.weight.grad
```

```
tensor([[ 0.0000,  0.0375,  0.0000,  0.0000],
        [ 0.0000,  0.0202,  0.0000,  0.0000],
        [ 0.0000, -0.0577,  0.0000,  0.0000]])
```

```
# 全結合層fc2のバイアスに関する勾配
net.fc2.bias.grad
```

```
tensor([ 0.3361, -0.1451, -0.1909])
```

上記のように、各パラメータに関する勾配が求められました。loss に記述されているメソッドを実行し、net の中のパラメータに変化があったことに戸惑うかもしれませんが、net → y → loss のように計算を行うとその関係性も保存されており、勾配の算出の際には逆向きに net ← y ← loss のように計算結果が格納されているため、net 内の属性に変化がありました。

これらの値を用いて、パラメータの更新を行います。パラメータの更新には optimizer を用います。

```
# 勾配の情報を用いたパラメータの更新
optimizer.step()
```

更新後のパラメータを更新前のパラメータと比較すると、値が変化していることがわかります。

```
# 全結合層fc1の重み
net.fc1.weight
```

```
Parameter containing:
tensor([[-0.0037,  0.2682, -0.4115, -0.3680],
        [-0.2642,  0.1009, -0.0685,  0.3740],
        [-0.0444,  0.1323, -0.1511, -0.0983],
        [-0.4777, -0.3311, -0.2061,  0.0185]], requires_grad=True)
```

```
# 全結合層fc1のバイアス
net.fc1.bias
```

```
Parameter containing:
tensor([ 0.1977,  0.2886, -0.3390, -0.2177], requires_grad=True)
```

```
# 全結合層fc2の重み
net.fc2.weight
```

```
Parameter containing:
tensor([[ 0.1816,  0.4114, -0.1029,  0.3742],
        [-0.0806,  0.0509,  0.4527, -0.4638],
        [-0.3148, -0.1208, -0.1949,  0.4320]], requires_grad=True)
```

```
# 全結合層fc2のバイアス
net.fc2.bias
```

```
Parameter containing:
tensor([-0.3577, -0.2157, -0.3302], requires_grad=True)
```

これで一連の流れを確認できました。数学の流れを把握しておけば、後は PyTorch 側で用意されているメソッドを用いるだけで大枠を実装できることがわかります。たとえば誤差逆伝播法を一から実装することは大変ですが、PyTorch ならば `loss.backward()` のたった 1 行で実装できてしまいます。

また、一連の流れでは紹介していませんでしたが、以下の 2 つも一連の処理の中で行います。

- データを使用するデバイスに転送
- パラメータの勾配を初期化

まず、データのデバイスへの転送ですが、ディープラーニングでは GPU を用いた演算の高速化を利用するのが一般的であり、その場合は学習時にデータとモデルの両方を GPU のメモリ上に転送する必要があります。学習に用いる計算機に GPU が搭載されている必要がありますが、たとえば Google Colaboratory のようなサービスでも無料で手軽に GPU を用いた演算を試せます。これについては第 6 章で説明しますが、今は CPU を用いた計算でも問題ありません。GPU が使用できる状況にあるかは、以下の関数で確認できます。

```
# 演算に使用できるGPUの有無を確認
torch.cuda.is_available()
```

```
False
```

ここでは `False` と表示されているため、GPU ではなく CPU での演算になります。自動的にデバイスを選択できるようにするには、次のように記述します。

```
# GPUの設定状況に基づいたデバイスの選択
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
device
```

```
device(type='cpu')
```

今は GPU が使用できない状況なので、cpu が選択されています。GPU が使用できる状況であれば、0 番目の GPU が cuda:0 のように指定されることになります*1。

モデルとデータを次のようにデバイスへ転送します。

```
# 指定したデバイスへのモデルの転送
net.to(device)
```

```
Net(
  (fc1): Linear(in_features=4, out_features=4, bias=True)
  (fc2): Linear(in_features=4, out_features=3, bias=True)
)
```

```
# 指定したデバイスへの入力値の転送
x = x.to(device)
x
```

```
tensor([[6.4000, 3.2000, 5.3000, 2.3000],
        [6.4000, 2.7000, 5.3000, 1.9000],
        [5.9000, 3.0000, 4.2000, 1.5000],
        [6.9000, 3.1000, 5.4000, 2.1000],
        [5.5000, 2.4000, 3.8000, 1.1000],
        [5.6000, 2.7000, 4.2000, 1.3000],
        [5.8000, 2.7000, 5.1000, 1.9000],
        [6.9000, 3.1000, 4.9000, 1.5000],
        [6.3000, 2.3000, 4.4000, 1.3000],
        [7.2000, 3.0000, 5.8000, 1.6000]])
```

*1 CUDA とは、NVIDIA 社が開発・提供している GPU 向けの統合開発環境です。

```
# 指定したデバイスへの目標値の転送
t = t.to(device)
t
```

```
tensor([2, 2, 1, 2, 1, 1, 2, 1, 1, 2])
```

次に、勾配の初期化について説明します。勾配の情報は loss.backward() で算出できますが、loss.backward() では求めた勾配情報が各パラメータの grad に代入されるのではなく、現状の勾配情報に加算されます。これは**累積勾配**（accumulated gradient）を用いたほうがよいアルゴリズムに備えたものですが、本書ではその実装までは扱いません。そのため、ここでは PyTorch の仕様であるという程度の認識で十分です。パラメータの勾配を求める前には勾配情報の初期化が必要であるということは覚えておいてください。

```
# 勾配情報の初期化
optimizer.zero_grad()
```

初期化後にパラメータの勾配を確認すると、すべての値が 0 で初期化されています。

```
# 初期化後の勾配情報
net.fc1.weight.grad
```

```
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
```

```
net.fc1.bias.grad
```

```
tensor([0., 0., 0., 0.])
```

学習ループ

それでは、ここまでに学んだ知識をまとめて、本格的な学習ループを記述します。

```
# エポックの数
max_epoch = 1

# ネットワークの初期化
torch.manual_seed(0)

# ネットワークのインスタンス化とデバイスへの転送
net = Net().to(device)

# 最適化手法の選択
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)

for epoch in range(max_epoch):

    for batch in train_loader:

        # バッチサイズ分のサンプルを抽出
        x, t = batch

        # 学習に使用するデバイスへデータの転送
        x = x.to(device)
        t = t.to(device)

        # パラメータの勾配を初期化
        optimizer.zero_grad()

        # 予測値の算出
        y = net(x)

        # 目標値と予測値から損失関数の値を算出
        loss = criterion(y, t)

        # 損失関数の値を表示して確認
        # .item(): tensor.Tensor => float
        print('loss:', loss.item())

        # 各パラメータの勾配を算出
        loss.backward()

        # 勾配の情報を用いたパラメータの更新
        optimizer.step()
```

```
loss: 1.1118199825286865
loss: 1.0638254880905151
loss: 1.0752708911895752
loss: 1.0017006397247314
```

```
loss: 1.0963518619537354
loss: 0.9922471046447754
loss: 0.9230359792709351
loss: 0.8453549146652222
loss: 0.8222911953926086
```

上記のように 1 エポック（9 イテレーション）分の学習を行い、損失関数の値が順調に小さくなっていることがわかります。エポックの数は、学習用データセットを何回繰り返し学習させるかを意味します。

ここで、分類の問題設定であるため、結果がひと目でわかりやすいように**正解率**（accuracy）を追加してみましょう。100 サンプル中 97 サンプルを正しいクラスに分類できていれば正解率は 97%、といったように直感的にわかりやすい指標です。

正解率の算出にあたり、まずは予測値において最も値が大きなクラスの番号を取得します。最大値を求めるには torch.max 関数、最大値に対する要素番号を求めるには torch.argmax 関数を用います。今回は属するクラスの番号だけでよいので、torch.argmax を使います。

```
# dim=0で行ごとの最大値に対する要素番号を取得（dim=0は列ごと）
y_label = torch.argmax(y, dim=1)

# 予測値から最大となるクラスの番号を取り出した結果
y_label
```

```
tensor([0, 0, 1, 0, 0, 1, 1, 0, 0, 0])
```

```
# 目標値
t
```

```
tensor([0, 1, 2, 0, 0, 2, 2, 1, 0, 0])
```

この結果から、比較演算子を使って、正解率を求めます。

```
# 値が一致しているか確認
y_label == t
```

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

```
# 値がTrueとなる個数の総和
(y_label == t).sum()
```

```
tensor(5)
```

この値を要素数で割ると正解率を求めることができますが、今は torch.Tensor の int 型となっているため、割り算して 1 を下回る場合には小数点以下が切り捨てられて 0 となってしまいます。この簡単な解決策として、1.0 を乗じて float 型に変換しておきましょう。

```
# int => float
(y_label == t).sum() * 1.0
```

```
tensor(5.)
```

要素数で割ることで正解率を求められます。

```
# 正解率
acc  = (y_id == t).sum() * 1.0 / len(t)
acc
```

```
tensor(0.5000)
```

上記のように、正解率の計算を記述できました。これを学習ループに追加して確認しましょう。

```
# ネットワークの初期化
torch.manual_seed(0)

# ネットワークのインスタンス化とデバイスへの転送
net = Net().to(device)

# 最適化手法の選択
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)

for epoch in range(max_epoch):

    for batch in train_loader:

        x, t = batch

        x = x.to(device)
```

```
        t = t.to(device)

        optimizer.zero_grad()

        y = net(x)

        loss = criterion(y, t)

        # New：正解率の算出
        y_label = torch.argmax(y, dim=1)
        acc  = (y_label == t).sum() * 1.0 / len(t)
        print('accuracy:', acc)

        loss.backward()

        optimizer.step()
```

```
accuracy: tensor(0.5000)
accuracy: tensor(0.5000)
accuracy: tensor(0.3000)
accuracy: tensor(0.9000)
accuracy: tensor(0.1000)
accuracy: tensor(0.6000)
accuracy: tensor(0.9000)
accuracy: tensor(0.5000)
accuracy: tensor(0.5000)
```

学習後の正解率を検証

これまでは学習データに対する処理を書いてきましたが、検証データやテストデータに対する結果も見ておく必要があります。学習データの場合とほとんど同じですが、1 点だけ異なるのは、検証データやテストデータに対しては学習を行わないため、勾配の情報が必要ないという点です。そのため、with torch.no_grad の中に検証用のコードを記述し、勾配に関する無駄な計算や計算機リソースの占有を避けましょう。

検証データでもテストデータでも対応できるように、data_loader を引数としてとり、使用する際には val_loader もしくは test_loader を指定します。

```
# 正解率の計算
def calc_acc(data_loader):

    with torch.no_grad():

        accs = []  # 各バッチの結果格納用
```

```
        for batch in data_loader:
            x, t = batch
            x = x.to(device)
            t = t.to(device)
            y = net(x)

            y_label = torch.argmax(y, dim=1)
            acc  = (y_label == t).sum() * 1.0 / len(t)
            accs.append(acc)

    # 全体の平均を算出
    avg_acc = torch.tensor(accs).mean()

    return avg_acc

# 検証データで確認
val_acc = calc_acc(val_loader)
val_acc
```

```
tensor(0.6000)
```

```
# テストデータで確認
test_acc = calc_acc(test_loader)
test_acc
```

```
tensor(0.4333)
```

この関数を学習ループに追加することで、学習中においても検証データに対する状況を確認できます。

本章では、PyTorch の仕様の確認から、学習ループの記述まで、幅広く紹介しました。次章では、今回の学習の流れをより簡単に記述できる PyTorch Lightning という PyTorch ラッパーを紹介します。この PyTorch Lightning を理解するには学習ループを把握しておく必要があり、本章での経験が次章に活きてきます。

第4章 PyTorch（応用編）

本章では、基礎編の内容を踏まえて、PyTorch Lightning による学習ループの簡略化から始め、精度向上のテクニック、ハイパーパラメータのチューニングまで実践的な内容を扱っていきます。データセットは前章までと同じ Iris を用いるので、GPU でなく CPU でも十分計算できます。

4.1 PyTorch Lightningによる学習ループの簡略化

PyTorch の標準的な学習ループの記述は、for 文が二重で登場するなど、初学者にとって易しい記述とは言えません。また、上級者にとっても、コードが複雑であるということはエラーのデバッグ処理が面倒になるといった悪影響があります。これを楽にするために、skorch のようなラッパーがありますが、簡単なネットワーク構成や設定でないと逆に設定が面倒になるといったデメリットもあります。PyTorch は研究開発に積極的に使われている背景があるため、skorch で完璧に対応できるような簡単なケースでは満足できないことが多々あると想定されます。

PyTorch ラッパーを使用した学習ループの簡略化

そこで、ある程度の学習ループの記述が必要になりつつも、複雑な設定ができる PyTorch ラッパーを用いることをお勧めします。PyTorch のラッパーとしては、以下の 3 つが有名です。

- Ignite（https://github.com/pytorch/ignite）
- PyTorch Lightning（https://github.com/PyTorchLightning/pytorch-lightning）
- Catalyst（https://github.com/catalyst-team/catalyst）

もともとは PyTorch チームが Ignite を開発しており、筆者も Ignite を使って開発をしていましたが、2019 年後半から PyTorch Lightning の人気が高まっています。PyTorch Lightning は Ignite よりもシンプルに記述でき、かつ分散処理などの本格的な開発にも耐えられる仕様で作ら

れているという大きなメリットがあります。

そこで本書では、PyTorch Lightning がこれからの PyTorch ラッパーの標準になると予想し、PyTorch Lightning を採用して説明することにします。Kaggle に挑戦する人たちの間では Catalyst も使われ始めているようで、こちらも今後の成長に注目です。これらのラッパーは現在も開発が進められており、バージョンが変わるごとに書き方が変わることも日常茶飯事です。ここで紹介するコードを動かすには、本書で使用するバージョンに合わせておくことをお勧めします。

環境構築

PyTorch Lightning のインストールは pip で行います。

```
# Windowsの場合
python -m pip install pytorch-lightning

# macOSの場合
pip3 install pytorch-lightning
```

インストール後に、正しく読み込みできるか確認しておきましょう。

```
import pytorch_lightning

# バージョンの確認
pytorch_lightning.__version__
```

```
'1.0.3'
```

本書では、執筆時点（2020 年 9 月）で最新版だった PyTorch Lightning 1.0.3 を用います。うまく動作しない場合には、pip でインストールを行う際にこのバージョンに合わせてみてください。

```
import torch

torch.__version__
```

```
'1.6.0'
```

PyTorch のバージョンは 1.6.0 です。

警告の非表示も設定しておきましょう。

```
import warnings

warnings.filterwarnings('ignore')
```

学習の準備

問題設定は前章の基礎編と同じです。データの読み込みから使用できるデータセットの形式への変換まで行っておきましょう。DataLoader の設定は PyTorch Lightning 側で用意されているため、不要です。

```
from sklearn.datasets import load_iris
from torch.utils.data import DataLoader, TensorDataset, random_split

# Irisデータセットの読み込み
x, t = load_iris(return_X_y=True)

# PyTorchで学習に使用できる形式へ変換
x = torch.tensor(x, dtype=torch.float32)
t = torch.tensor(t, dtype=torch.int64)

# 入力値と目標値をまとめて、1つのオブジェクトdatasetに変換
dataset = TensorDataset(x, t)

# 各データセットのサンプル数を決定
# train : val : test = 60% : 20% : 20%
n_train = int(len(dataset) * 0.6)
n_val = int(len(dataset) * 0.2)
n_test = len(dataset) - n_train - n_val

# ランダムに分割を行うため、シードを固定して再現性を確保
torch.manual_seed(0)

# データセットの分割
train, val, test = random_split(dataset, [n_train, n_val, n_test])

# バッチサイズの定義
batch_size = 32

# Data Loaderを用意
# shuffleはデフォルトでFalseのため、訓練データのみTrueに指定
train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True, drop_last=True)
val_loader = torch.utils.data.DataLoader(val, batch_size)
test_loader = torch.utils.data.DataLoader(test, batch_size)
```

PyTorch Lightning によるネットワークと学習手順の定義

PyTorch Lightning では、ネットワークを定義するクラスの中に、学習の手順も記述します。標準的な PyTorch の書き方を学んだ後であれば、学習ループの一部を抽出しているだけとわかるので、つまずきは少ないはずです。ネットワークを定義する際のクラスは nn.Module を継承していましたが、これを PyTorch Lightning の LightningModule を継承するように変更します。まずは検証データとテストデータを抜いた学習データのみに対する最小限のクラスを設計していきます。

```
import pytorch_lightning as pl
import torch.nn as nn
import torch.nn.functional as F

class Net(pl.LightningModule):

  def __init__(self):
    super().__init__()

    self.bn = nn.BatchNorm1d(4)
    self.fc1 = nn.Linear(4, 4)
    self.fc2 = nn.Linear(4, 3)

  def forward(self, x):
    h = self.bn(x)
    h = self.fc1(h)
    h = F.relu(h)
    h = self.fc2(h)
    return h

  def training_step(self, batch, batch_idx):
    x, t = batch
    y = self(x)
    loss = F.cross_entropy(y, t)
    return loss

  def configure_optimizers(self):
    optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
    return optimizer
```

新しく configure_optimizers()、training_step() というメソッドを追加しています。メソッドとしては新しいですが、概念としてはいずれもすでに紹介してきたものであり、configure_optimizers() は最適化手法の設定、training_step() は学習ループのミニバッチ抽出後の処理です。

最も基本的な PyTorch Lightning での記述は、これで設定完了です。ここから簡単にネット

ワークの学習を行うことができます。なお、前章までは入力値や目標値を学習に使用するデバイスに渡す処理を明示していましたが、この処理は PyTorch Lightning 側で行われるため、気にする必要がありません。

ネットワークの学習

ネットワークの学習には、PyTorch Lightning で用意されている Trainer を用います。まずは Trainer のデフォルトの設定で学習を行います。

```
# GPU を含めた乱数のシードを固定
pl.seed_everything(0)
net = Net()

# 学習を行う Trainer
trainer = pl.Trainer(max_epochs=5)

# 学習の実行
trainer.fit(net, train_loader, val_loader)
```

これで学習が完了しました。前章までは for 文の中に複雑に処理を書いていたことを思えば、はるかにすっきりとした印象です。また、複雑なネットワークの構成も__init__と forward 内に書くことで実現でき、複雑な学習のステップがあったとしても training_step 内に書くことで実現できます。このように、PyTorch Lightning は手軽さと柔軟さの両方をあわせ持ったラッパーであり、研究開発にとって有力な選択肢になるはずです。

ここで、Trainer の引数の中で代表的なものを紹介しておきます。

引数名	デフォルトの値	説明
max_epochs	1000	学習時の最大エポック数
min_epochs	1	学習時の最小エポック数
gpus	None	使用する GPU の数
distributed_backend	None	分散学習の方法

エポック数も一種のハイパーパラメータであり、適切な値は状況に依存するため、計算時間的に可能な範囲の max_epochs を指定しておく方法が現実的な最良の解と言えます。

TensorBoard による結果の可視化

PyTorch では、TensorBoard と呼ばれる、結果をきれいに可視化できるパッケージを用いることができます。PyTorch Lightning から TensorBoard に必要な形式のログファイルを標準で出力できるので、設定はほとんど必要ありません。

TensorBoard を用いるために、パッケージのインストールを pip から実行します。

```
# Windowsの場合
python -m pip install tensorboard

# macOSの場合
pip3 install tensorboard
```

```
import tensorboard

# バージョンの確認
tensorboard.__version__
```

```
'2.0.2'
```

TensorBoard へのログ出力ができるように、設定を 1 箇所だけ書き換えましょう。

```
class Net(pl.LightningModule):

  def __init__(self):
    super().__init__()

    self.bn = nn.BatchNorm1d(4)
    self.fc1 = nn.Linear(4, 4)
    self.fc2 = nn.Linear(4, 3)

    self.train_acc = pl.metrics.Accuracy()

  def forward(self, x):
    h = self.bn(x)
    h = self.fc1(h)
    h = F.relu(h)
    h = self.fc2(h)
    return h

  def training_step(self, batch, batch_idx):
    x, t = batch
    y = self(x)
    loss = F.cross_entropy(y, t)
    # ログを追加
    # on_stepをTrueにすると各イテレーションごとの値が記録される（default：True）
    # on_epochをTrueにすると各エポックごとの値が記録される（default：False）
    self.log('train_loss', loss, on_step=True, on_epoch=True)
    self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
```

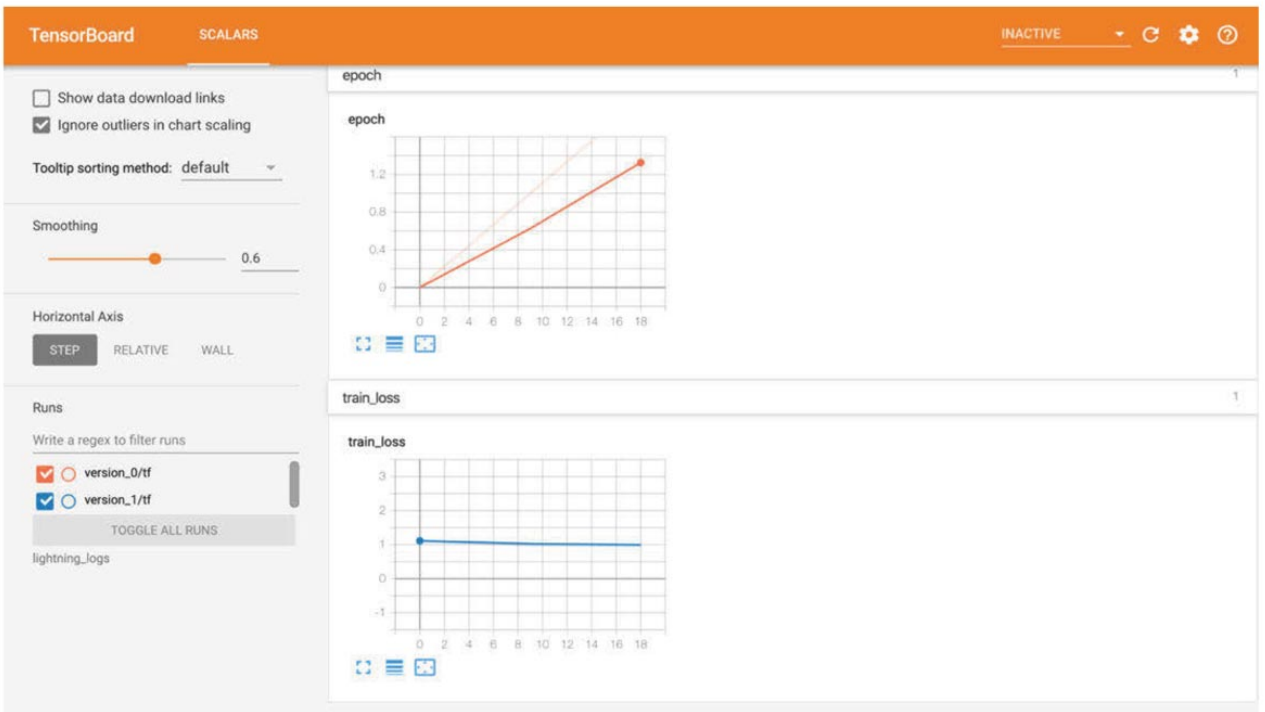
```
    return loss

  def configure_optimizers(self):
    optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
    return optimizer
```

これまでと同様にネットワークの学習を行います。

```
# 学習の実行
pl.seed_everything(0)
net = Net(0)
trainer = pl.Trainer(max_epochs=30)
trainer.fit(net, train_loader, val_loader)
```

実行後、このファイルと同じフォルダ内に lightning_logs というフォルダが自動的にできており、その中に version_0 という結果を格納したフォルダも作成されています。この結果を確認するには、以下のコマンドを Terminal 上で実行してください。

```
%load_ext tensorboard
%tensorboard --logdir lightning_logs/
```



ここで、TensorBoard を起動しようとした際に「ValueError: Duplicate plugins for name projector」というエラーが出た場合、複数バージョンの TensorBoard がインストールされている可能性があります。pip uninstall tensorboard では 1 バージョン分しかアンインストールできず、エラーの対処にはなりません。このようなときには、以下のようにインストールされているディレクトリ（site-packages）の PATH を確認して、そのディレクトリにアクセスし、複数存在する TensorBoard のディレクトリを手動で削除します。そして、再度 pip で tensorboard パッケージをインストールすればエラーを解決できます。

```
# パッケージがインストールされているディレクトリの確認
tensorboard.__file__
```

```
'/usr/local/lib/python3.7/site-packages/tensorboard/__init__.py'
```

検証データの追加

検証データに対する結果の計算などは任意で追加することができ、validation_step に記述します。validation_step は検証データに対する各イテレーションの結果を集計します。検証データやテストデータに対する計算の場合には、torch.no_grad() を用いて勾配情報を持たないようにしていましたが、そういった処理も PyTorch Lightning 側で設定されています。

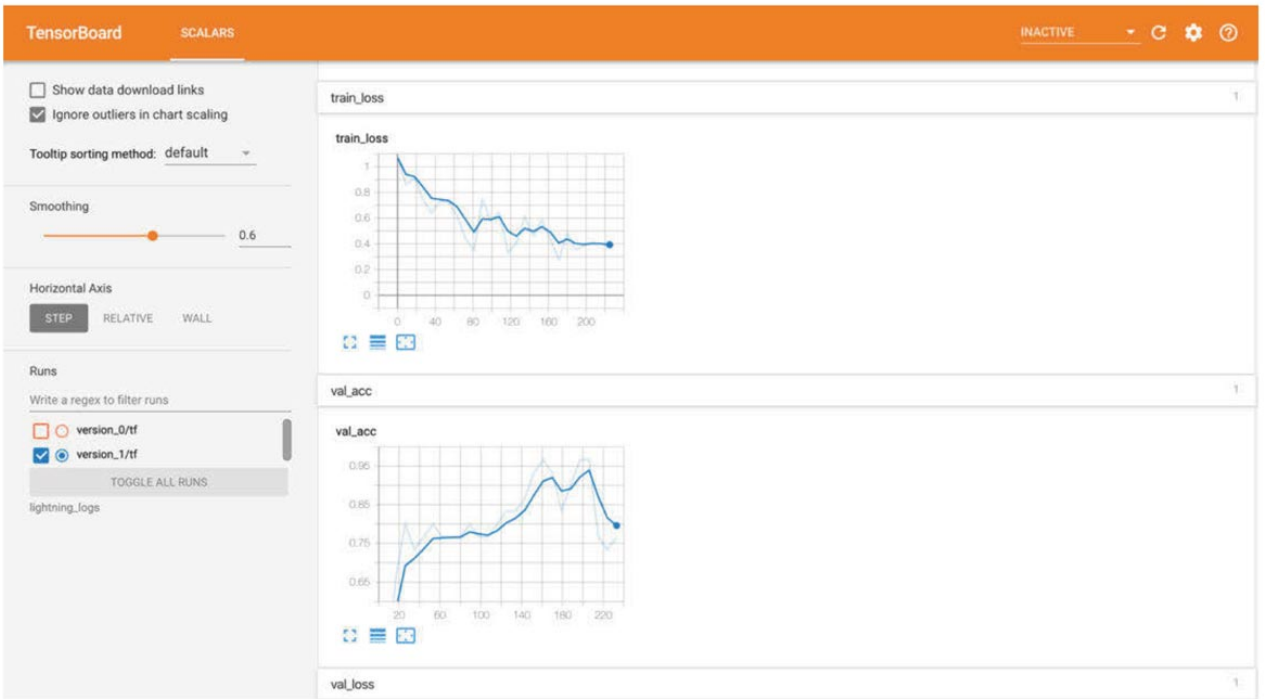
```
class Net(pl.LightningModule):

  def __init__(self):
    super().__init__()

    self.bn = nn.BatchNorm1d(4)
    self.fc1 = nn.Linear(4, 4)
    self.fc2 = nn.Linear(4, 3)

    self.train_acc = pl.metrics.Accuracy()
    self.val_acc = pl.metrics.Accuracy()
    self.test_acc = pl.metrics.Accuracy()

  def forward(self, x):
    h = self.bn(x)
    h = self.fc1(h)
    h = F.relu(h)
    h = self.fc2(h)
    return h

  def training_step(self, batch, batch_idx):
    x, t = batch
```

```
    y = self(x)
    loss = F.cross_entropy(y, t)
    self.log('train_loss', loss, on_step=True, on_epoch=True)
    self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
    return loss

  # 検証データに対する処理
  def validation_step(self, batch, batch_idx):
    x, t = batch
    y = self(x)
    loss = F.cross_entropy(y, t)
    self.log('val_loss', loss, on_step=False, on_epoch=True)
    self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
    return loss

  # テストデータに対する処理
  def test_step(self, batch, batch_idx):
    x, t = batch
    y = self(x)
    loss = F.cross_entropy(y, t)
    self.log('test_loss', loss, on_step=False, on_epoch=True)
    self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
    return loss

  def configure_optimizers(self):
    optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
    return optimizer
```

それでは、設計した内容に基づいて検証データでの結果の確認も含めた学習を行っていきます。

```
# 学習の実行
pl.seed_everything(0)
net = Net(0)
trainer = pl.Trainer(max_epochs=30)
trainer.fit(net, train_loader, val_loader)

# テストデータで検証
results = trainer.test(test_dataloader=test_loader)
```

TensorBoard を確認すると、検証データに対する val_acc や val_loss も追加されており、結果を確認できます。



これで PyTorch Lightning による学習データを用いたネットワークの学習、検証データとテストデータに対する結果の確認まで一連の流れを把握することができました。定義するクラスこそこれまでより大きくなりましたが、学習の部分や検証のコードが劇的に減っているため、ネットワーク構築の試行錯誤が容易になります。

4.2　Ax によるハイパーパラメータの調整

PyTorch でハイパーパラメータを調整するフレームワークとして、Ax（https://github.com/facebook/Ax）が 2019 年 5 月に登場しました。ベイズ最適化という機械学習と一緒によく用いられる最適化手法を扱うフレームワークである、BoTorch（https://github.com/pytorch/botorch）をベースに作られています。Ax、BoTorch のどちらも Facebook が開発したものです。

ほかにも、Optuna などベイズ最適化をベースとしたハイパーパラメータの調整を行うフレームワークも公開されていますが、ここでは PyTorch を正式にサポートしている Facebook 製であることを理由に Ax を採用しました。このハイパーパラメータ調整はパブリッククラウドである Microsoft Azure でも機能が提供されているため、本書の後半ではこちらを用いるケースも紹介します。

環境構築

これまでのパッケージと同様、Ax のインストールは以下のように行います。

```
# Windowsの場合
python -m pip install ax-platform

# macOSの場合
pip3 install ax-platform
```

```
import ax

# バージョンの確認
ax.__version__
```

```
'0.1.6'
```

例題

ハイパーパラメータの調整を行う前に、簡単な例題に対して Ax による最適化を実行していきます。2 つの変数を持つ目的関数

$$f(x) = x_1^2 + x_2^2$$

の最小化を行います。ここで、変数の値域は $x_i \in [-10,\ 10](i = 1, 2)$ とします。答えは $x_1 = x_2 = 0$ と明確であり、この答えが求まるかを検証します。パラメータと目的関数の 2 つを定義するだけでよく、残りの最適化を行うアルゴリズムは Ax 側でサポートされています。

最適化を行うパラメータの種類を type によって指定できます。

type	内容	指定	例
range	指定された範囲で値を選択	bounds	`[-10, 10]`
fixed	1 つの値を固定	value	`10`
choice	複数の値の中から選択	values	`[1, 10, 100]`

range を使用する場合、`[-1, 1]` とすると、$-1, 0, 1$ のように整数の範囲内で値が選択され、`[-1., 1.]` とすると $-1.0, 0.999, \ldots$ のように実数値の範囲内で選択されます。ここではパラメータを連続値の range を用いて定義します。

```
parameters = [
    {'name': 'x1', 'type': 'range', 'bounds': [-10., 10.]},
    {'name': 'x2', 'type': 'range', 'bounds': [-10., 10.]},
]
```

そして、このパラメータに基づいて、最適化したい関数を evaluation_function として定義します。以下の x1 と x2 は、先ほど定義したパラメータを受け取ります。そして、それぞれの値を 2 乗して加算し、f を出力します。

```
def evaluation_function(parameters):
    x1 = parameters.get('x1')
    x2 = parameters.get('x2')
    f = x1**2 + x2**2
    return f
```

これで準備完了です。最適化を実行し、答えを確認します。

```
# 最適化の実行
# minimize=Trueで最小化、Falseで最大化
# 乱数を使用するため、シードの固定も再現性確保には必須
results = ax.optimize(parameters, evaluation_function, minimize=True, random_seed=0)
```

```
[INFO 12-09 01:13:18] ax.service.utils.dispatch: Using Bayesian Optimization generation strate⇨
gy: GenerationStrategy(name='Sobol+GPEI', steps=[Sobol for 5 arms, GPEI for subsequent arms], ⇨
generated 0 arm(s) so far). Iterations after 5 will take longer to generate due to model-fitting.
[INFO 12-09 01:13:18] ax.service.managed_loop: Started full optimization with 20 steps.
[INFO 12-09 01:13:18] ax.service.managed_loop: Running optimization trial 1...
[INFO 12-09 01:13:18] ax.service.managed_loop: Running optimization trial 2...
 …
[INFO 12-09 01:13:28] ax.service.managed_loop: Running optimization trial 19...
[INFO 12-09 01:13:29] ax.service.managed_loop: Running optimization trial 20...
```

```
# 結果の確認
best_parameters, values, experiment, model = results

# 最適化後に得られたパラメータ
best_parameters
```

```
{'x1': -0.03563694574402376, 'x2': 0.011034888584450897}
```

```
# 最適化後に得られたパラメータでの関数の値
values
```

```
({'objective': 0.00050982121091891 49},
 {'objective': {'objective': 8.72053185034275e-05}})
```

この結果からわかるとおり、多少の誤差はありますが、真の解に近しい値が得られています。このように関数を定義できれば、パラメータを最適化できます。

ただし、ディープラーニングでは学習データに基づいてパラメータの最適化を行えるため、厳密には重みやバイアスといったパラメータの最適化を使用するわけではありません。ディープラーニングでは層やノードの数、最適化手法の学習係数などが決まった状況でパラメータの最適化を行いますが、これらのあらかじめ決まっていたハイパーパラメータが最適とは限りません。そのため、パラメータ以外にもハイパーパラメータも最適化を行う必要があり、ここに Ax を用います。

最適化の準備

それでは、前述の問題に対してハイパーパラメータの最適化を行います。今回調整するハイパーパラメータは、中間層のノード数 n_mid と学習係数 lr とします。ネットワークを定義する際に、引数に中間層のノードをとれるよう、以下のように変更を加えます。

```
class Net(pl.LightningModule):

  def __init__(self, n_mid=4, lr=0.01):
    super().__init__()
    self.fc1 = nn.Linear(4, n_mid)
    self.fc2 = nn.Linear(n_mid, 3)
    self.lr = lr

    self.train_acc = pl.metrics.Accuracy()
    self.val_acc = pl.metrics.Accuracy()
    self.test_acc = pl.metrics.Accuracy()

  def forward(self, x):
    h = self.fc1(x)
    h = F.relu(h)
    h = self.fc2(h)
    return h

  def training_step(self, batch, batch_idx):
    x, t = batch
```

```
    y = self(x)
    loss = F.cross_entropy(y, t)
    self.log('train_loss', loss, on_step=True, on_epoch=True)
    self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
    return loss

  # 検証データに対する処理
  def validation_step(self, batch, batch_idx):
    x, t = batch
    y = self(x)
    loss = F.cross_entropy(y, t)
    self.log('val_loss', loss, on_step=False, on_epoch=True)
    self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
    return loss

  # テストデータに対する処理
  def test_step(self, batch, batch_idx):
    x, t = batch
    y = self(x)
    loss = F.cross_entropy(y, t)
    self.log('test_loss', loss, on_step=False, on_epoch=True)
    self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
    return loss

  def configure_optimizers(self):
    optimizer = torch.optim.SGD(self.parameters(), lr=self.lr)
    return optimizer
```

まずは一度、最適化を行わない状況で学習を行います。

```
pl.seed_everything(0)
net = Net()
trainer = pl.Trainer(max_epochs=10)
trainer.fit(net, train_loader, val_loader)
```

ハイパーパラメータの最適化を行う関数には、検証データに対する最終的な loss か accuracy のどちらかを用います。今回は accuracy を採用し、この指標を最大化するように最適化を行います。

まずは、現状の accuracy を確認しましょう。trainer の属性に callback_metrics があり、ここに validation_end の返り値が格納されています。

```
# 最後のエポックに対する検証データのaccuracy
trainer.callback_metrics['val_acc']
```

```
0.9333333373069763
```

今回の目的関数は、上記で得られる accuracy を採用すればよいことがわかりました。それでは、パラメータと目的関数の定義を行い、最適化を実行しましょう。

```
parameters = [
    {'name': 'n_mid', 'type': 'range', 'bounds': [1, 100]},  # 1〜100の整数
    {'name': 'lr', 'type': 'range', 'bounds': [0.001, 0.1]}  # 0.001から0.1までの実数
]

def evaluation_function (parameters):

    # パラメータの取得
    n_mid = parameters.get('n_mid')
    lr = parameters.get('lr')

    # ネットワークの定義と学習
    torch.manual_seed(0)
    net = Net(batch_size=10, n_mid=n_mid, lr=lr)
    trainer = Trainer(show_progress_bar=False)
    trainer.fit(net)

    # 検証データに対するaccuracyを目的関数の値として返す
    val_acc = trainer.callback_metrics['val_acc']

    # 確認のためテストデータのaccuracyも算出
    trainer.test()
    test_acc = trainer.callback_metrics['test_acc']

    # 各試行の選択されたハイパーパラメータの値と結果を表示
    print('n_mid:', n_mid)
    print('lr:', lr)
    print('val_acc:', val_acc)
    print('test_acc:', test_acc)

    return val_acc

# 最適化の実行
# デフォルトがminimize=Falseであり、今回は最大化であるため、デフォルトでOK
results = ax.optimize(parameters, evaluation_function, random_seed=0)

# 結果を取得
best_parameters, values, experiment, best_model = results

# 得られた最適なパラメータ
best_parameters
```

```
{'n_mid': 61, 'lr': 0.0027084381673943975}
```

```
# 最適なパラメータにおける目的関数の値
values
```

```
({'objective': 0.9666667057036189},
 {'objective': {'objective': 7.13449581416142e-10}})
```

この結果より、検証データに対する accuracy が 0.97 程度のハイパーパラメータを見つけることができ、ハイパーパラメータチューニング前の検証データに対する正解率であった 0.76 から大幅に改善していることがわかります。確認用に見ていたテストデータに対しては 0.9 程度であるため、7% 程度正解率が落ちていることを考慮すると、本番で運用を始めるまでどの程度の正解率になるかは油断できないという印象です。

本章では、次章以降で多用していく PyTorch Lightning の使い方から、手軽にベイズ最適化によるハイパーパラメータ調整を行える Ax の使い方まで紹介しました。

第5章 環境構築

本章では、機械学習を行う上で重要な環境構築について扱っていきます。AI エンジニアと呼ばれる人材が求められる時代となり、研究に携わっていなくても職業として AI を選択する人が増えています。筆者の会社もそうですが、専門知識のない人に AI について教える教育業も増え始めています。そのような教育業者のもとで機械学習に関する数学とプログラミングを学んだ後、Kaggle など実践的なモデル構築の練習を行い、AI エンジニアの職に就けることを目指します。

このように機械学習のモデル構築だけをひたすら練習してきた人が実務に出た際、まずつまずくのが「環境構築」です。多くの教育業者は構築済みの環境を使って解析を行っているため、実習中にこの力が身に付いていません。手元の PC（ローカル）で解析を行う場合は、クリック 1 つでインストールできるソフトウェアが配布されている便利な時代なので問題ないのですが、最近ではディープラーニングを動かすために GPU を積んだ環境を整えることは必須となってきています。

5.1 環境構築の選択肢

本章では、環境構築の選択肢として次の 3 つを紹介します。

1. Google Colaboratory
2. Microsoft Azure
3. Docker

昨今クラウドサービスを利用した機械学習環境構築が話題となっており、PaaS としては Microsoft Azure、SaaS としては Google アカウントさえ作成すれば無料で GPU を利用できる Google Colaboratory。そして最近では仮想化技術を取り入れるケースも増えており、ローカルで構築した環境を OS の異なるリモートでも即時にまったく同じものを展開できる Docker があります。環境構築の選択肢はほかにもありますが、ここでの内容を押さえておけば勉強や仕事の現場で困ることは少なくなるので、ぜひ覚えていきましょう。

本書は基本的に Microsoft Azure の VM（Virtual Machines）および、GPU が使える仮想化

技術 Docker を利用した環境を使用しますが、その前に Google Colaboratory について説明しておきます。

5.2 Google Colaboratory

ここでは Google Colaboratory の利用方法を簡単に説明します。必要な事前準備は次の 2 点です。

- Google アカウント（お持ちでない場合は「Google アカウントの作成」（https://accounts.google.com/signup/v2/webcreateaccount?flowName=GlifWebSignIn&flowEntry=SignUp）から作成）
- Chrome もしくは Firefox のインストール

Google Colaboratory（以下 Colab）は、クラウド上で Jupyter Notebook 環境を提供する Google の Web サービスです。Jupyter Notebook は Web ブラウザ上で主に以下のようなことが可能なオープンソースの Web アプリケーションであり、データ分析の現場や研究、教育などで広く用いられています。

- プログラムの実行と、その結果の確認
- メモや解説などの記述の追加に Markdown を使用可能

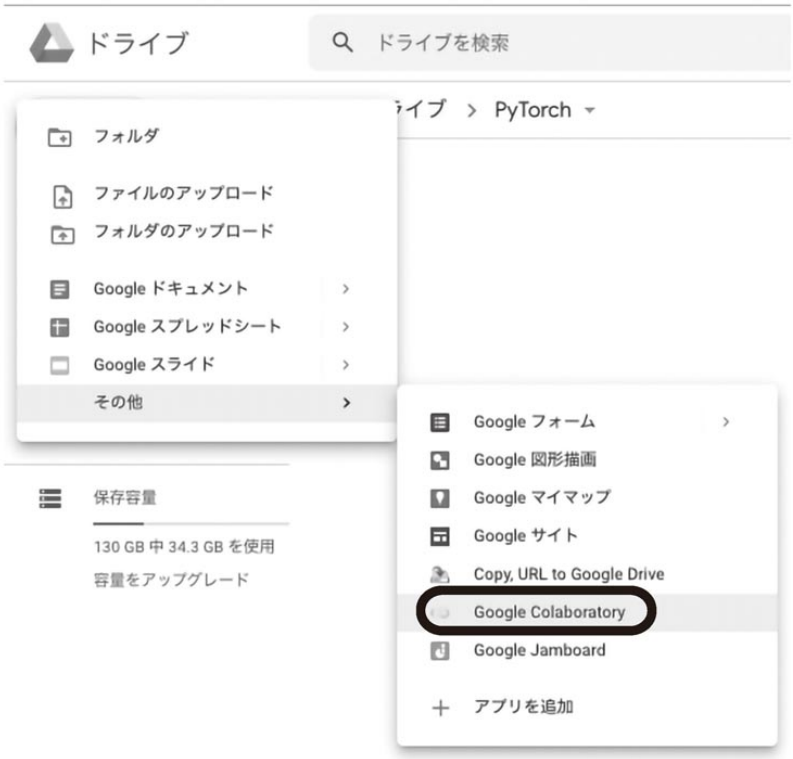
Colab では無料で GPU も使用できますが、そのランタイムは**最大 12 時間**で消えてしまうため、時間を長く要する処理などは別途環境を用意する必要があります。学び始めのうちは、数分から数時間程度で終わる処理がほとんどなので気にする必要はありませんが、本格的に使っていく場合は有料のクラウドサービスを利用するなどして、環境を整えるようにしましょう。

Colab を開く

まずは Web ブラウザで https://www.google.com/ にアクセスして、Google の自分のアカウントにログインしてください。

ログインが完了したら、次の画面のように Google Drive のページに移動します。Google Drive とは、ファイルなどを保存できるオンラインストレージサービスです。

Google Drive の画面の右上にある「新規」ボタンをクリックし、次のように新規のノートブックを作成します。



これで Google Drive から Colab を使えるようになりました。Google Drive 内に作業フォルダを作成して、資料をアップロードしましょう。フォルダの作成やファイルのアップロードを行うには、画面上で右クリックすると選択肢が表示されます。

必要なデータセットなどは Google Drive 上にアップロードしておいて、Colab 上に展開していくことになります。

ノートブックとセル

Colab 上の Jupyter Notebook を以降、単に**ノートブック**と呼びます。

ノートブックは、**セル**と呼ばれるブロックを複数持つことができます。新しいノートブックを作った直後は、何も書かれていないセルが 1 つだけ存在している状態になっています。セルの内側のどこかをクリックすると、そのセルを選択できます。

セルには、**コードセル**と**テキストセル**の 2 種類があります。コードセルは Python のコードを書き込み実行するためのセル、テキストセルは Markdown 形式で文章を書くためのセルです。

それぞれのセルタイプについて、もう少し詳しく説明します。

コードセル

コードセルは、Python のコードを書き込み、実行できるセルです。実行するには、コードセルを選択した状態で、Ctrl + Enter キーまたは Shift + Enter キーを押します。試しに、以下の内容をセルに記入し、Shift + Enter キーを押してみてください。

```
print('Hello world!')
```

```
Hello world!
```

すぐ下に「Hello world!」という文字列が表示されるはずです。最初のセルに書き込んだのは Python のコードで、与えられた文字列を表示する関数である `print()` に、`'Hello world!'` という文字列を渡しています。これを今実行したため、その結果が下に表示されています。

テキストセル

テキストセルでは、Markdown 形式で記述された文章を扱います。テキストセルをクリックして編集モードにすると、Markdown 形式で文章を装飾するための記号が見えるようになります。

その状態で Shift + Enter キーを押すと、元の文章の表示に戻ります。

Colab から Google Drive を使う

Google Drive を、Colab で開いたノートブックから利用できます。ノートブック中でコードを実行して作成したファイルなどを保存したり、逆に Google Drive 上に保存されているデータを読み込んだりすることができます。

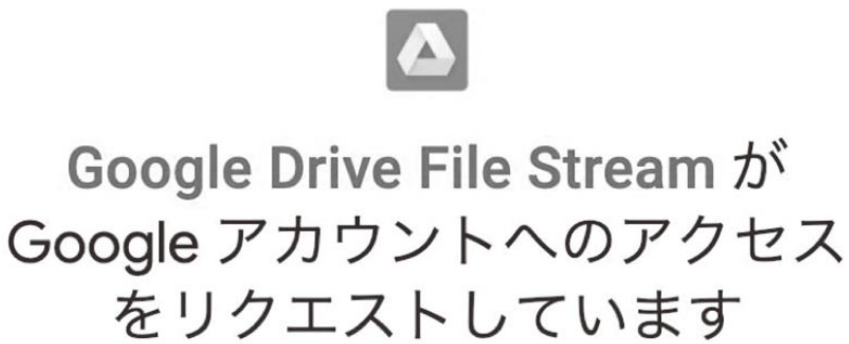
Colab 上のノートブックから Google Drive を使うには、Colab 専用のツールを使って、`/content/drive` という PATH に現在ログイン中の Google アカウントが持っている Google Drive のスペースをマウントします。

```
from google.colab import drive
drive.mount('/content/drive')
```

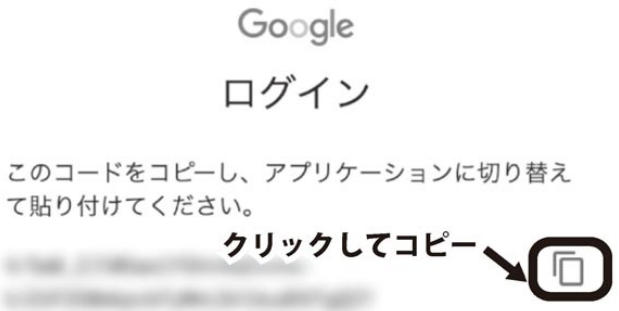
このノートブックを Colab で開いてから初めて上のコードセルを実行した場合は、次のようなメッセージが表示されます。

指示に従って表示されている URL へアクセスしてください。すると、「アカウントの選択」と書かれたページに飛び、すでにログイン済みの場合はログイン中の Google アカウントのアイコンやメールアドレスが表示されます。利用したいアカウントをクリックして、次に進んでください。

次に、「Google Drive File Stream が Google アカウントへのアクセスをリクエストしています」と書かれたページに飛びます。



右下に「許可」と書かれたボタンがあるので、これをクリックしてください。すると、次のように認証コードが記載されたページへ移動します（紙面では認証コード部分をぼかしています）。



このコードを選択してコピーするか、右側にあるアイコンをクリックしてコピーしてください。

元のノートブックへ戻り、「Enter your authorization code:」というメッセージの下にある空欄に、先ほどコピーした認証コードを貼り付けて、Enter キーを押してください。「Mounted at /content/drive」と表示されたら、準備は完了です。

次の内容を記入したセルを実行して、自分の Google Drive が Colab からアクセス可能になっていることを確認しましょう。

```
# 'My Drive'の表記が出ていればマウントがうまく行われています。
!ls 'drive/'
```

上記で実行しているのは Python のコードではありません。Jupyter Notebook では、コードセル中に「!」が先頭に付いている行は特別に解釈されます。「!ls」は、次に続くディレクトリの中にあるファイルまたはディレクトリの一覧を表示せよ、という意味です。

Colab の便利なショートカット

Colab の使用中にセルのタイプの変更やセルの複製・追加などの操作をするには、メニューから該当する項目を選ぶ方法以外に、キーボードショートカットを利用する方法もあります。

次によく使うショートカットキーをまとめておきます。多くのショートカットキーは二段階になっており、まず Ctrl + M キーを押してから、それぞれの機能によって異なるコマンドを入力する手順になっています。

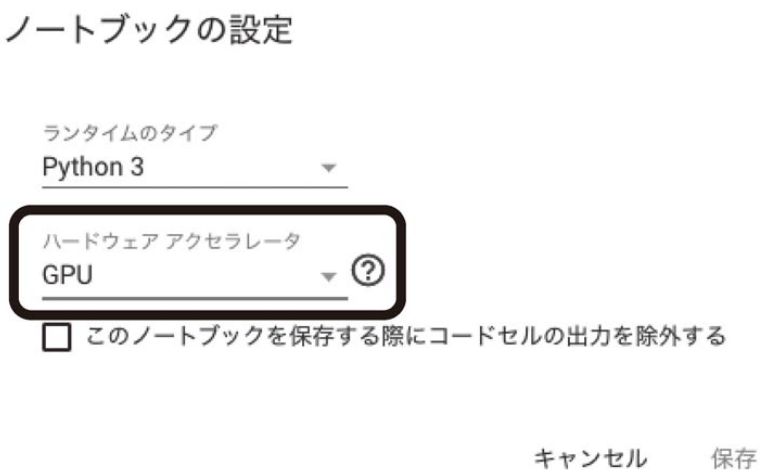
説明	コマンド
Markdown モードへ変更	Ctrl + M → M
Code モードへ変更	Ctrl + M → Y
セルの実行	Shift + Enter
セルを上に追加	Ctrl + M → A
セルを下に追加	Ctrl + M → B
セルのコピー	Ctrl + M → C
セルの貼り付け	Ctrl + M → V
セルの消去	Ctrl + M → D
コメントアウト	Ctrl +/

ショートカットを使用するだけで作業時間に差が出るので、使っていきましょう。

GPU を使用する

Colab では GPU を無料で使用できます。初期設定では GPU を使用しない設定になっているので、これを変更する必要があります。GPU を使用するには、画面上部のタブの中の「Runtime」（ランタイム）をクリックし、「Change runtime type」（ランタイムのタイプを変更）を選択します。

そして、次のように「Hardware accelerator」（ハードウェアアクセラレータ）を GPU に変更します。



これで Colab 上で GPU を使用できるようになりました。設定としてはこれで十分です。すぐに実験したい場合にはぴったりで、環境構築が必要ないという点も機械学習の敷居を低くしてくれるので、活用しましょう。

5.3 Microsoft Azure

ここからは本章から扱う環境の構築方法になります。Microsoft Azure は、開発者や IT プロフェッショナルがアプリケーションのビルド、デプロイ、管理に使用できる各種クラウドサービスを統合した、現在も拡大を続けている集合体であり、サービスは世界規模のデータセンターネットワークを介して配信されています。Azure では、お好みのツール、アプリケーション、フレームワークを使用し、好きな場所で自由にビルドとデプロイを行うことができます。

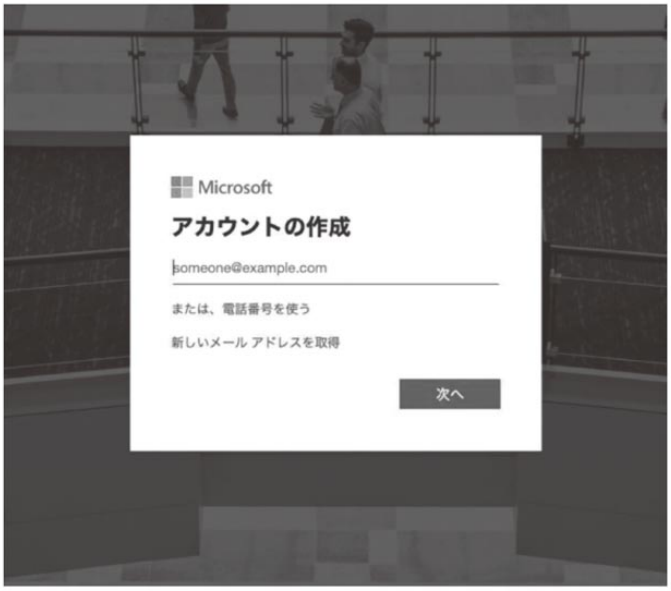
Azure アカウントの作成

最初に Azure アカウントの作成を行いましょう。

- Microsoft Azure：`https://azure.microsoft.com/`

上記の URL にアクセスすると、右上に「無料アカウント」のボタンがあるので、そこから手順に沿ってアカウントを作成します。

メールアドレスを登録すると確認メールが届くので、認証コードを入力してください。



クレジットカードの設定や個人情報を入力する画面が表示されます。

必要情報を記入し、以下の「Azure のご利用を開始できます」の画面が表示されれば、利用する準備は完了です。

アカウントの作成が終わりました。初めてアカウントを作成すると、無料で 22,500 円分のクレジットが付いてきます。初めの一歩としてディープラーニングで遊ぶには十分なクレジットなので、ありがたく使っていきましょう。

Azure ポータルにログイン

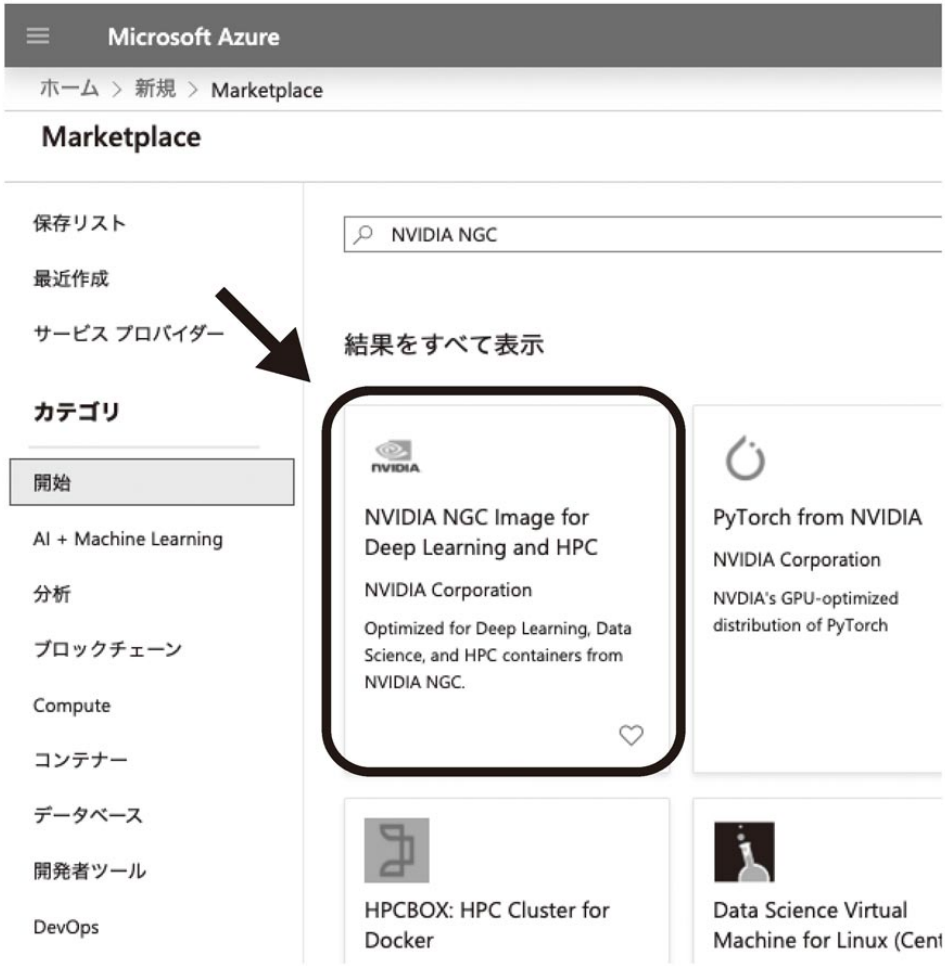
先ほどの画面から「ポータルに移動」をクリックしましょう。使用方法について説明するために「ツアーを開始」という案内が表示されますが「後で行う」をクリックしてホームに移動してください。

インスタンスを作成してみよう

それでは、実際に仮想マシンを立ち上げてみましょう。この仮想マシンのことを**インスタンス**と呼びます。

NVIDIA NGC で検索

「新規」タブから「NVIDIA NGC」で検索をすると、いくつか検索結果が出てきます。以降では環境構築で Docker を使用していきますので、用意されているイメージのリソースを選択しましょう。



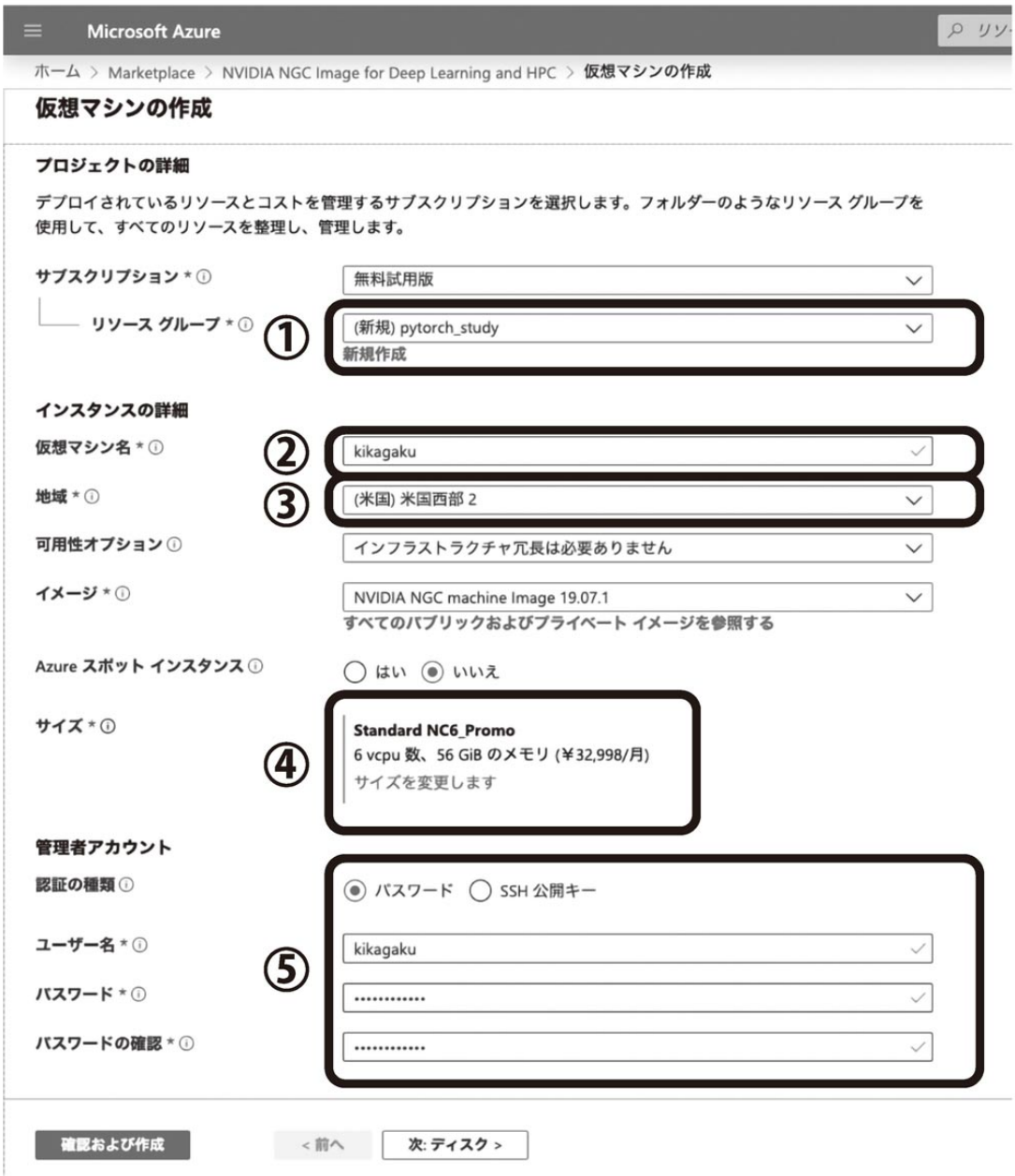
NVIDIA NGC Image for Deep Learning and HPC で作成

「NVIDIA NGC Image for Deep Learning and HPC」を選択します。NVIDIA Driver や Docker などで構成されています。

「作成」をクリックして詳細設定に進んでください。

仮想マシンの詳細設定

仮想マシンの作成にあたり、詳細を設定する必要があります。設定する箇所が 5 つありますので、順番に紹介していきます。



① **リソースグループ**：1 つのアプリケーションを構成するためのサービス、Web サービス、データベース、ストレージ、ネットワークなど、相互依存している複数のリソースを、サー

ビスごとではなく任意のグループでまとめて管理できます。

② **仮想マシン名**：仮想マシンには最大 15 文字までの名前を割り当てます。この名前はオペレーティングシステム（OS）内のディスク名の一部などとして利用されています。

③ **地域**：Microsoft Azure で作成されるすべてのリソースは、世界各地の複数の地理的リージョンに分散管理されます。VM（Virtual Machines）の場合、仮想ハードディスクの格納場所を指定します。

④ **サイズ**：使用する VM のサイズは、実行する作業の大きさによって決まります。さらに、選択したサイズによって、処理能力、メモリ、ストレージの容量などの要素が決まります。VM のサイズおよびオペレーティングシステムに基づいて時間単位の料金が請求されます。

⑤ **管理者アカウント**：ユーザー名とパスワードを設定します。VM に SSH 接続する際にも使用するので、忘れないように必ずどこかにメモしておいてください。

選択する地域によって扱えるサイズが変わってきますので、GPU が扱える地域を選択しましょう。今回はプロモーション価格になっている NC_Promo シリーズを選択することにします。さらに高インスタンスが必要な重い処理が必要となる方は、以下の URL を参考にサイズを決定してください。

- `https://azure.microsoft.com/ja-jp/pricing/details/virtual-machines/series/`

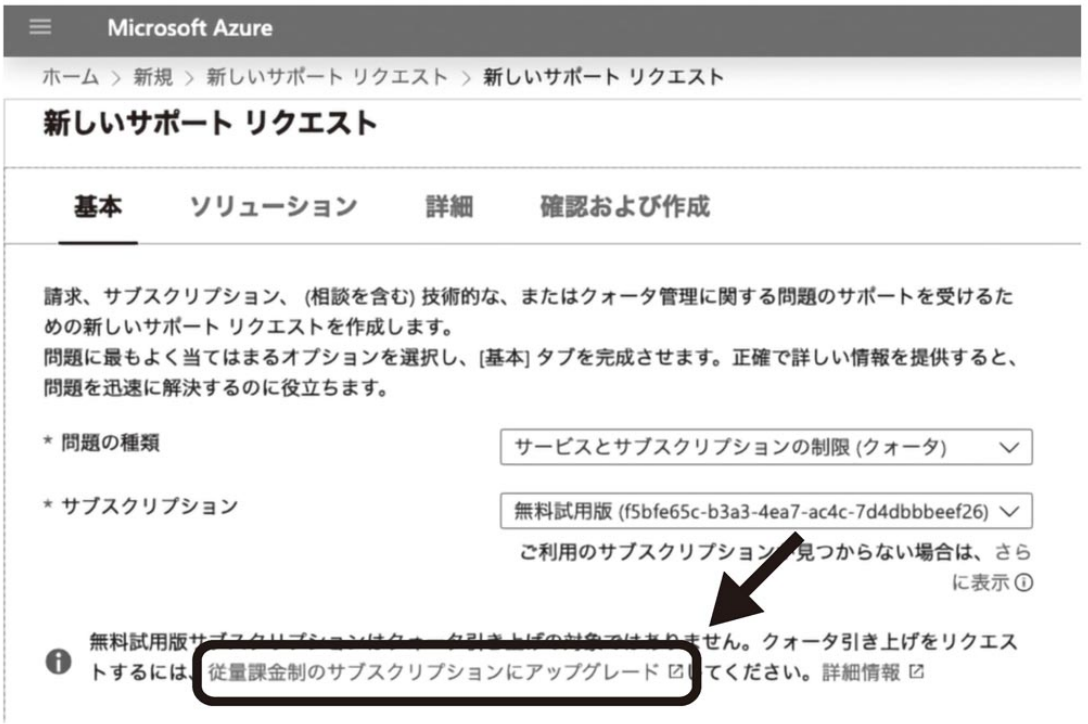
仮想マシンの作成

ほかにもディスクの設定や不本意な課金を防ぐための自動シャットダウンの設定などがありますが、ここでは作成まで一気に進めていきます。タブ右端の「確認および作成」に進み、「作成」をクリックします。

しかし、エラーが出てしまいます。

Azure リソースグループ内のリソースのクォータ（Azure のさまざまなリソースの制限）は、サービス管理クォータと同様に、サブスクリプションごとではなく、サブスクリプションによってアクセス可能なリージョンごとに決められています。ここでは米国西部を選択しましたが、デフォルトの許容最大数は 4 と定められており、「NC6_Promo」はクォータ数 6 が必要なので、コア数の追加要求が必要になりました。

Azure ポータルの「ホーム」→「新規」→「新しいサポートリクエスト」からクォータ引き上げリクエストを送ることができますが、無料試用版サブスクリプションは引き上げ対象ではありません。従量課金制サブスクリプションにアップグレードする必要があります。



「ホーム」→「アップグレード」からサブスクリプションのアップグレードを申請できます。従量課金制になりますが、無料試用版でもらっている 22,500 円分のクレジットがなくなるわけではなく、無料クレジットが終わると、従量課金に移行されます。

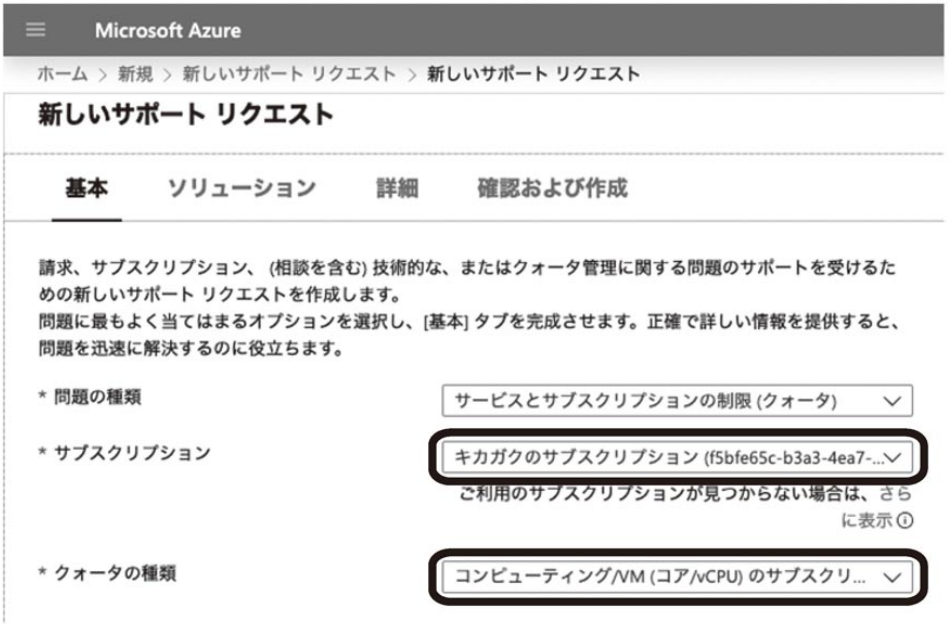
アップグレードにも種類があり、月額料金が高いほど手厚いサポートをリアルタイムに受けることができます。今回は無料の「テクニカル サポートなし」を選択しておきます。

まだクォータの引き上げリクエストを要請していませんが、従量課金制サブスクリプションにするとクォータの制限がリージョンあたり 20 まで引き上げられるので、もう一度仮想マシンの作成に戻ってみましょう。

作成できました。デプロイまでに数分かかるので、待っている間にクォータの引き上げリクエストを要請してみましょう。

クォータの引き上げリクエスト

従量課金制サブスクリプションにアップグレードしたら、リクエストを送れる状態になっています。「サブスクリプション」には命名したサブスクリプション名を、「クォータの種類」には「コンピューティング/VM（コア/vCPU）のサブスクリプション」を選択し、次へ進んでください。



続いて、リクエストの詳細設定をしていきます。サポート情報や連絡先情報を入力してください。ここで大事なポイントはクォータの詳細です。今回は地域は「米国西部 2」、サイズは「NC Promo Series」、「DSv3 Series」、「Dv3 Series」の 3 つで 100 コアをリクエストします。こちらの量は個人で動かす分には 100 あれば十分です。地域によって制限や上限があり、リクエストが通らないこともありますので、その都度サポートと交渉してください。

それでは詳細を確認し、リクエストを作成しましょう。

リクエスト完了です。登録したメールアドレスまたは電話番号に返信がきますので、結果をお待ちください。この手順を踏めば今後さらに高インスタンスを大量に使用したい場合などに困ることなくリクエストできるので、覚えておくとよいでしょう。

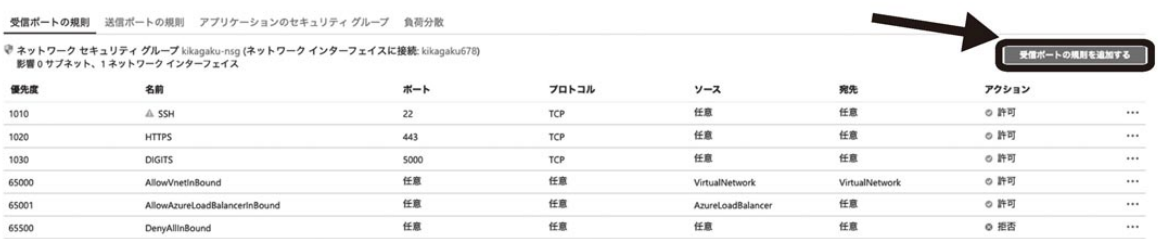
ポートの開放

デプロイが完了したら、SSH 経由で Azure の VM へアクセスします。

デプロイが完了すると、次のようにパブリックアドレスが表示されるため、このアドレスを控えておきましょう。

また、この後に使う Jupyter Notebook では IP アドレス以外に、ポート（入口のようなもの）番号 8888 を使用します。基本的にはセキュリティ対策としてイレギュラーなポート番号は閉じられているため、このポートを開けてやる必要があります。この処理を行わない場合、ポートが閉まっているため、Jupyter Notebook を Azure 上に立ち上げてもまったく反応がないということになります。

ポートの開放はサイドバーの「ネットワーク」から行います。Azure では GUI でポートの開放まで行えるので、とても簡単です。以下の「ポート規則を追加する」をクリックします。



「名前」に適当な名前を指定して「宛先ポート範囲」を 8888 に変更し、「適用」をクリックすれば、ポート開放のための処理が完了します。デプロイが行われ、数分後には指定したポート番号が使用可能になります。

SSH で VM へ接続

割り振られた IP アドレスをコピーしておき、macOS の方は Terminal、Windows の方は Git Bash を起動後、その IP アドレスを指定します。

```
> ssh kikagaku@<パブリックIPアドレス>
```

初めてのログインの際は次のような確認が表示されますが、気にせず「yes」として次に進んでください。

```
> ssh kikagaku@<パブリックIPアドレス>
The authenticity of host '<パブリックIPアドレス>' can't be established.
ECDSA key fingerprint is SHA256:Rm+ayBiVkM+nnRm05oh/hYtnFNyt40Ra70As+bHdNaA.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

VM の新規作成を行ったときのパスワードでログインしてください。

```
> ssh kikagaku@<パブリックIPアドレス>
The authenticity of host '<パブリックIPアドレス>' can't be established.
ECDSA key fingerprint is SHA256:Rm+ayBiVkM+nnRm05oh/hYtnFNyt40Ra70As+bHdNaA.
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '<パブリックIPアドレス>' (ECDSA) to the list of known hosts.
kikagaku@<パブリックIPアドレス>'s password:
```

ログインに成功すると、以下のようなターミナルの画面に入ることができます。

```
kikagaku@kikagaku:~$
```

これで Microsoft Azure の VM を利用した基本的な環境構築は完了です。ここからは、実際に解析環境を整えるために Docker を使って Python 環境や GPU 環境を構築します。

Azure VM 内で Docker コンテナを立ち上げる

続いて、実際に VM 内で GPU を使った解析環境を構築していきます。

第 1 章でも紹介しましたが、**Docker** は OS レベルでの仮想化環境を提供するソフトウェアです。この仮想化技術により、Windows、maxOS、Linux などのホスト OS に依存することがなく、開発を進めることができます。たとえば、ローカルは macOS で作業していたとしても、リモートのサーバーを借りたときにはそこには Linux を入れるのが一般的であり、macOS で環境構築した手順とは異なる方法が必要となります。そのため、macOS と Linux の 2 つの手順をそれぞれ把握しておく必要がありました。これに対し、macOS と Linux 上で動作する Docker の環境上で開発を行えば、それぞれの OS の差異はほとんどなく、開発を進められます。また、Python で必要なパッケージをインストール済みで渡すことができたりと、開発からデプロイまで大きな恩恵を受けることができます。

NVIDIA NGC Image for Deep Learning and HPC には、執筆時点の Docker の最新版である 19.03 が最初からインストールされています。

NVIDIA Container Toolkit とは

Docker では仮想的な環境を簡単に共有できますが、そのままでは GPU に接続することができません。そこで登場したのが **NVIDIA Container Toolkit**[*1]です。中身としては、NVIDIA が Docker をもとに作った、CUDA 環境のイメージを簡単に作れるようにするためのものです。CUDA は、NVIDIA が開発している、GPU による並列計算処理のための開発環境です。CUDA や cuDDN（CUDA Deep Neural Network library の略で、ニューラルネットワーク計算用のライブラリ）を手作業で導入するのは非常に面倒なのですが、そのような問題を一気に解決してくれます。

本書用 GPU 搭載 VM を作成

次章からの作業向けに、まずは VM 内で使用する Docker イメージを pull（ダウンロード）します。

```
sudo docker pull kikagaku/handson_pytorch
```

sudo は、ユーパーユーザーなどの別のユーザー権限でコマンドを実行するためのコマンドです。

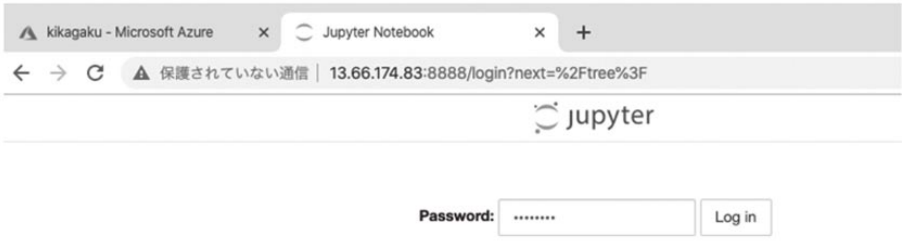
ダウンロードが完了したら、このイメージをベースにコンテナを立ち上げます。GPU を使用するために--gpus オプションが必要になっているので注意してください。

```
sudo docker run --gpus -all -it -p 8888:8888 kikagaku/handson_pytorch
```

これでコンテナが立ち上がり、自動的に Jupyter Notebook も立ち上がります。これを確認するには、先ほどポートを開放した<IVMのパブリックアドレス>:8888 に手元の Web ブラウザからアクセスしてください。

次のように Jupyter Notebook へアクセスできれば成功です。

＊1　かつては NVIDIA-docker という名前で呼ばれていました。



パスワードは「kikagaku」です。

これで環境が完成しました。Docker を使えば非常に簡単に環境を構築できます。

5.4 Docker

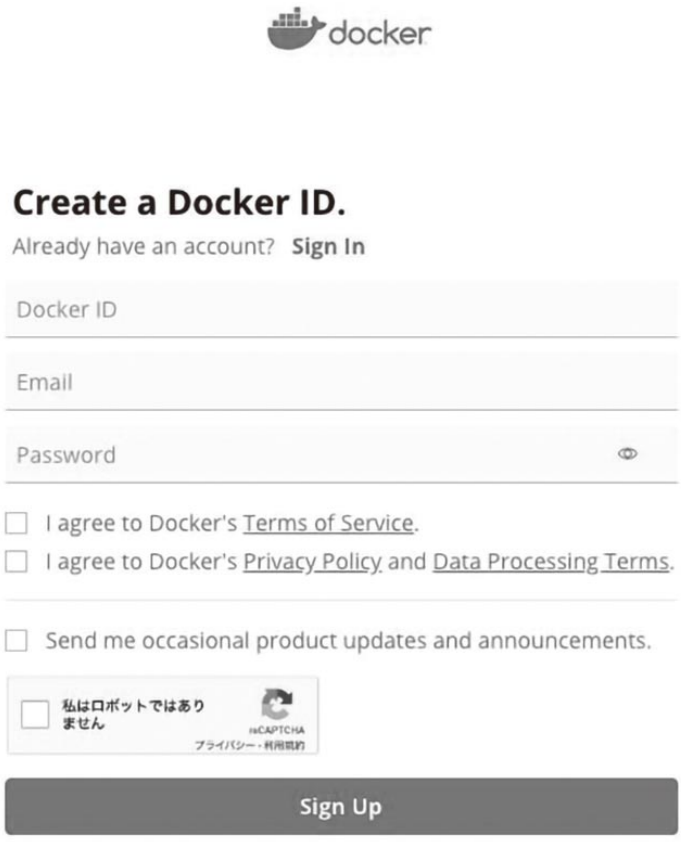
Docker については前述しましたが、ここでは Azure で紹介した Docker とは別の使用方法を紹介します。

Docker のインストール

Docker をインストール済みでない方は、下記に沿ってインストールを行ってください。なお、Windows 10 より前のバージョン、もしくは Windows10 Home Edition では Docker for Windows がうまくインストールできないため、Docker Toolbox を使うのですが、こちらは別途いろいろと知識が必要ですので、Ubuntu 16.04 LTS 版をインストールして、そちらに Docker をインストールして使うほうが楽かもしれません。難しい場合は以下のとおりにご自身の PC のバージョンなどを確認して、環境構築を行ってください。

アカウントの作成

インストーラーをダウンロードする前に Docker Hub のアカウントが必要となります。Docker Hub（https://hub.docker.com/）は、Docker のイメージをホスティングするリポジトリを提供しているサービスです。無料で使い始めることができます。



Windows（Windows 10 Professional またはWindows 10 Enterprise エディションの場合）

Docker Desktop for Windows（https://docs.docker.com/docker-for-windows/install/）を利用してインストールできます。インストーラーの手順どおり進めましょう。

Windows（Windows 10 Home などのエディションの場合）

Docker Toolbox（https://docs.docker.com/toolbox/overview/）をインストールします。インストールの手順は長くなるため、今回はその他の記述は省いて進めていきます。

https://qiita.com/maemori/items/52b1639fba4b1e68fccd の記事などをもとにインストールを進めてください。

macOS

https://docs.docker.com/docker-for-mac/install/ にアクセスし、Docker Desktop for Mac のインストーラーをダウンロードの上、インストールを進めてください。

Docker へログイン

Docker の動作確認も兼ねて、Terminal 上で Docker のログインを行いましょう。次のコマンドを入力後、Docker Hub で設定したアカウント情報を入力してログインしてください。

```
docker login
```

ログインに成功すると「Login Succeeded」と表示され、準備完了です。

Docker の基礎

Docker では**イメージ**と**コンテナ**の 2 つがよく登場します。イメージは構築された環境を再現できる設計図です。プログラミングで登場するクラスのような存在です。そして、そのイメージをベースに実体化させたものをコンテナと呼びます。インスタンスに対応するものです。このイメージを共有して、それぞれの計算機でコンテナを立てて、作業をするという流れです。このイメージを作るには 2 つの方法があります。

1 つ目はダウンロードしてきたベースとなるイメージからコンテナを立てて、コンテナの中でさらに必要なソフトウェアをインストールし、そのコンテナをもとに新しいイメージを作る方法です。この方法のメリットは、コンテナの中で作業を行うため、エラー対処がしやすい点が挙げられます。一方、すべての手順を覚えておかないとイメージ化した後ではどのようにインストールしたかわからず、再現性を確保することが難しい点がデメリットとして挙げられます。

2 つ目は、Dockerfile と呼ばれる Docker のイメージを作るための設定ファイルに環境構築の手順をすべて列挙し、イメージを作成する方法です。この方法では、どのような手順でインストールを行ったかすべて残せるため、次回以降の環境構築においても再現性を確保できます。また、GitHub などで Dockerfile を共有し、環境構築の手順を他の人に教えることもできます。一方、環境構築中にエラーが出たときの対処が初心者では難しい点がデメリットとして挙げられます。

イメージを作る方法はどちらも一長一短ですので、慣れないうちはコンテナを立てて作業するところから始めて、慣れてきたら Dockerfile を使ったイメージの作成にチャレンジするのがよいでしょう。

登場するコマンドの一覧は以下のとおりです。

- `push`：Docker Hub などのリポジトリへイメージをアップロード
- `pull`：Docker Hub などのリポジトリからイメージをダウンロード
- `run`：イメージから新規のコンテナを作成
- `commit`：コンテナから新しいイメージを作成
- `build`：Dockerfile から新しいイメージを作成

コンテナの扱い方

それでは、初心者向けの方法として紹介していた、コンテナを新しく立てて作業を行い、イメージを作る手順を紹介します。Python がインストールされているイメージをダウンロードし、そこに NumPy をインストールした新しいイメージを作成します。

イメージのダウンロード

該当するイメージを Docker Hub から探すのですが、Python のバージョンごとに分けられたイメージは Docker Hub 上に準備されています。

ログインが完了すると、Docker Hub からイメージをダウンロードできます。

```
# docker pull <イメージ名>
docker pull python
```

該当するイメージが Docker Hub からダウンロードされます。ダウンロード後のイメージは以下のコマンドで確認できます。

```
docker images
# REPOSITORY  TAG     IMAGE ID      CREATED       SIZE
# python      latest  4c0fd7901be8  42 hours ago  929MB
```

今回はイメージ名を python としましたが、タグ（tag）と呼ばれるバージョンを指定することもできます。タグを使用しない場合は latest と呼ばれる最新版のバージョンがダウンロードされます。Python 3.7 のバージョンを指定したイメージをダウンロードしてみます。

```
# docker pull <イメージ名>:<タグ>
docker pull python:3.7
```

コンテナの立ち上げ

コンテナを立ち上げる run を使って、ダウンロードしてきたイメージからコンテナを作成します。

```
docker run python:3.7
```

しかし、上記のコマンドを実行しても、何も反応がありません。実はコンテナ自体を立ち上げることに失敗しているわけではありません。docker ps -a のコマンドでこれまで立てたコンテナの一覧を見ることができます。

```
docker ps -a
# CONTAINER ID  IMAGE       COMMAND    CREATED             STATUS                          PORTS ⇨
 NAMES
# da66de9f5e30  python:3.7  "python3"  About a minute ago  Exited (0) About a minute ago       ⇨
 priceless_haslett
```

上記の結果のとおり、python:3.7 をベースイメージとした priceless_haslett という名前のコンテナが作られています。名前は、コンテナの作成時にオプションで指定しなければランダムに決まります。

コンテナの作成自体はできていたのですが、実行する処理内容がなく、すぐに Exited とコンテナを停止させてしまったのです。この問題を避けるには、run のオプションとして、-it を付けるとよいでしょう。これにより、コンテナと標準入出力を共有できます。

```
docker run -it python:3.7
Python 3.7.4 (default, Jul  9 2019, 00:06:43)
[GCC 6.3.0 20170516] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> exit()
```

Python の対話モードが起動しました。起動した Python のバージョンを見ると、3.7.4 となっているため、確かに Python 3.7 のようです。このように、Docker を使えば複数のバージョンに対するテストも容易に行えることがわかります。

NumPy をインストールしたいのですが、Python の対話モードでは pip を使うことができません。このようなときにはコンテナの起動時にエントリーポイントを指定する方法があります。エントリーポイントとは、起動するときに実行するコマンドのことです。エントリーポイントとして/bin/bash を指定すると、サーバーにログインしたような bash コマンドが使えるモードとなります。NumPy をインストールして、読み込めるか確認します。

```
docker run -it python:3.7 /bin/bash
root@8fb75f2f34b1:/# pip install numpy
root@8fb75f2f34b1:/# python
>>> import numpy
>>> exit()
root@8fb75f2f34b1:/# exit
```

これで NumPy のインストールが完了しました。一度抜けてしまったため、もう一度コンテナを立ち上げ、NumPy がインストールされているか確認しましょう。

```
docker run -it python:3.7 /bin/bash
root@361acbc898d4:/# python
>>> import numpy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'numpy'
```

この結果のとおり、先ほど NumPy をインストールしたはずでしたが、もう一度コンテナを立ち上げると NumPy が見つからないというエラーが起こります。この理由は、先ほど NumPy のインストールを行ったコンテナと、今回 NumPy がインストールされているか検証したコンテナはまったく別のものだからです。run コマンドは新しいコンテナの立ち上げを毎回行います。そのため、一度作成したコンテナに再度アクセスしたいと思っても、run ではアクセスすることができません。

作成済みのコンテナに再度アクセスするには、start もしくは exec を使います。停止中（Exited）のコンテナの場合は start、起動中（Up）のコンテナの場合は exec を使います。docker ps -a で 2 つ前のコンテナの名前を探し、start で再度起動させましょう。

```
docker start -i confident_bassi
root@8fb75f2f34b1:/# python
>>> import numpy
>>> exit()
```

上記の結果のとおり、NumPy がインストール済みであることを確認できました。新規のコンテナを立てるか、既存のコンテナに再度入るかを区別するように気を付けましょう。

コンテナをイメージ化

NumPy のインストールを行ったコンテナをイメージ化しましょう。新しく作るイメージには Docker Hub のユーザー名も付けておくことを推奨します。Docker Hub へ push するときに<ユーザー名>/<イメージ名>のように名前を付けていないと、push が拒否されてしまうためです。

```
# docker commit <コンテナ名> <ユーザー名>/<イメージ名>
docker commit confident_bassi kikagaku/docker-test
```

イメージが正常に作成されているか確認しましょう。

```
docker images
# REPOSITORY              TAG                  IMAGE ID          CREATED             SIZE
# kikagaku/docker-test    latest               99c3b31f6ce7      2 minutes ago       939MB
```

イメージをアップロード

作成したイメージを Docker Hub へアップロードします。

```
docker push kikagaku/docker-test
```

アップロードが完了したら、今回の場合 https://hub.docker.com/r/kikagaku/docker-test に情報が反映されます。これで誰でも kikagaku/docker-test のイメージを使えるようになりました。

本章では環境構築として、無料で使える Colab、高火力インスタンスを使うための Azure VM や、よく使われる仮想化技術 Docker の使用方法など幅広く紹介しました。環境構築は目を背けられがちですが、環境が揃わなければデータ解析は始まりません。皆さんの環境構築が少しでも改善されることを願っています。

第6章 画像処理

本章では、畳み込みニューラルネットワークの基礎とそれに必要な画像処理について学んでいきます。画像処理を学ぶにあたり、そもそも画像とは一体どんな形で扱われるのかを確認していきましょう。

6.1 画像処理入門

CNN（Convolutional Neural Network）を学ぶために、まず **OpenCV** や **Pillow** を使用した画像処理の基礎を習得しましょう。ライブラリを使用して、画像をどのように加工していくのか、フィルタという概念はどのようなものであるかを学びます。

Python で画像を扱うライブラリとしては、OpenCV と Pillow が有名です。画像をインライン表示するために、最初に設定を行っておきましょう。

```
# Jupyter Notebook内のでのインライン表示
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
```

警告も非表示に設定しておきます。

```
import warnings

warnings.filterwarnings('ignore')
```

OpenCV

OpenCV は cv2 という名前で登録されています。OpenCV のインストールは一癖あり、単純に pip ではインストールできないのですが、Google Colaboratory や Docker イメージで配布した環境ではすでにインストール済みです。画像 sample.png については、

https://drive.google.com/open?id=1n0fI1A-KMq5PCQTje1M088b_S3hh4dDb からダウンロードし、data フォルダに置いてください。

```
import cv2

cv2.__version__
```

```
'4.1.2'
```

```
# 画像の読み込み
img = cv2.imread('data/sample.png')
```

OpenCV に読み込まれた画像は NumPy の ndarray 形式で扱われるため、これまでの慣れた操作をそのまま使用できます。

```
# 変数のクラスを確認
type(img)
```

```
numpy.ndarray
```

```
# サイズの確認
img.shape
```

```
(512, 512, 3)
```

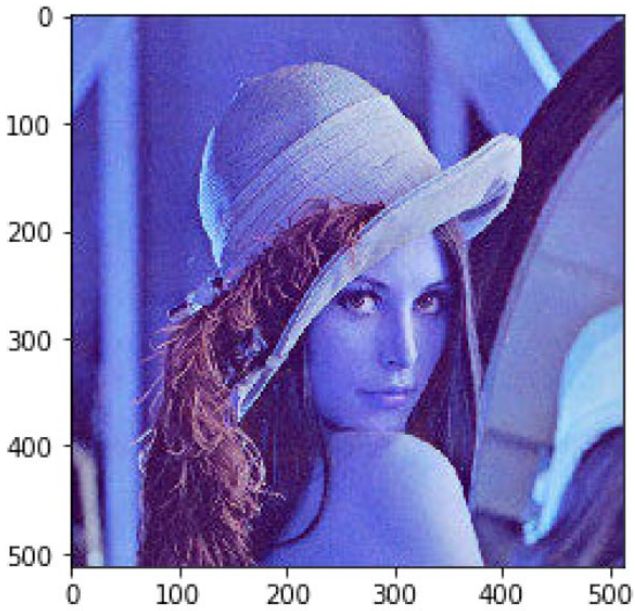
```
# 詳細なデータ型の確認
img.dtype
```

```
dtype('uint8')
```

uint8 は unsigned int の略で、0～255 までを表現可能な符号なし（正の値のみ）の 8 ビット整数です。

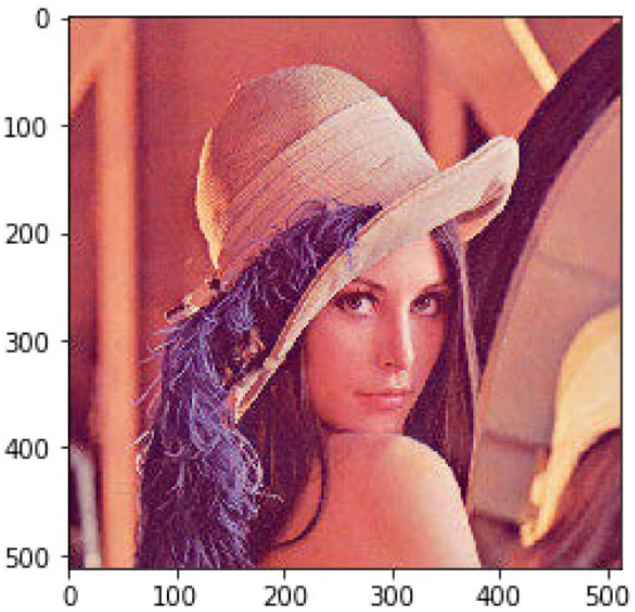
OpenCV では画像が **BGR** の順で格納されているため、Matplotlib を使用した画像の描画の際には、おかしな色合いで表示されてしまいます（紙面では白黒のため違いがわかりませんが)。

```
import matplotlib.pyplot as plt

plt.imshow(img)
```



Matplotlib が **RGB** の順で格納されていることを前提としているので、正しい色合いで表示するために BGR を RGB の順に格納されるように変換する必要があります。

```
# BGR -> RGB
img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

plt.imshow(img_rgb)
```



Pillow

Pillow は Python パッケージを管理するためのサービスで、誰でもパッケージを登録できる PyPI に PIL という名前で登録されています。

```
from PIL import Image

img = Image.open('data/sample.png')
```

Pillow も同様にクラスを確認してみましょう。

```
type(img)
```

```
PIL.PngImagePlugin.PngImageFile
```

残念ながら、Pillow で読み込んだ変数は慣れた NumPy の形式ではありません。
しかし、Pillow で読み込んだ画像は、そのまま変数名を打つだけで表示できます。

```
img
```

見た目にはわかりにくいのですが大きな違いとして、OpenCV は **BGR** の順に変数に格納されており、Pillow は **RGB** の順に格納されています。これを忘れたまま学習に OpenCV、推論に Pillow を使うと予測結果がおかしくなるので、注意してください。

Pillow を NumPy の形式に変換するには、NumPy の array に渡すだけです。

```
import numpy as np

img = np.array(img)

# クラスの確認
type(img)
```

```
numpy.ndarray
```

```
# サイズの確認
img.shape
```

```
(512, 512, 3)
```

```
# 詳細なデータ型の確認
img.dtype
```

```
dtype('uint8')
```

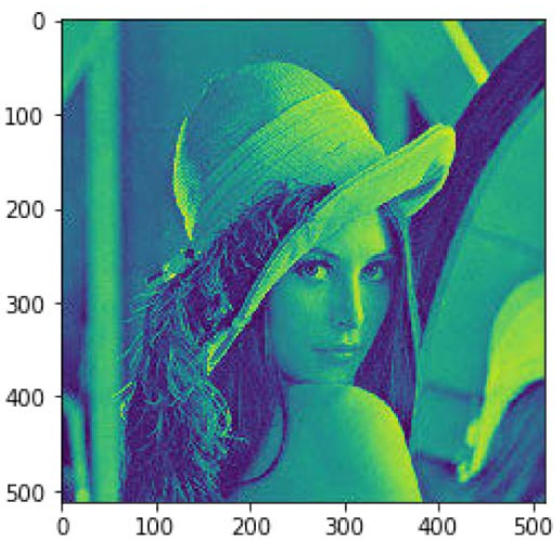
グレースケール変換

それでは、代表的な処理であるグレースケール変換を施してみましょう。

```
img_gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

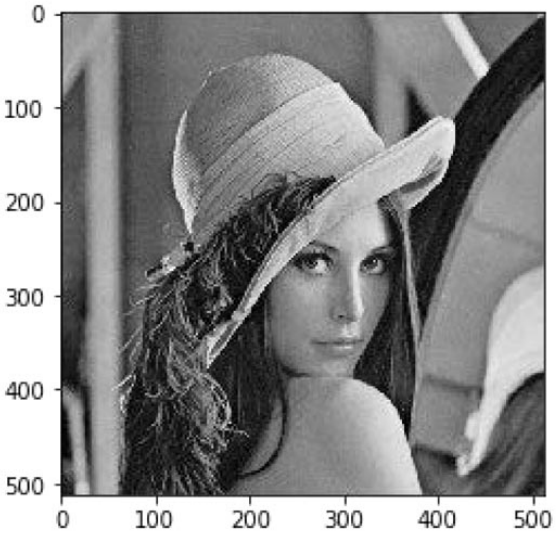
Matplotlib の imshow は RGB が渡される前提なので、グレースケールでは変な色合いになってしまいます（これも白黒紙面だと違いはあまりわかりませんが）。

```
plt.imshow(img_gray)
```



そのため、グレースケールであることを宣言してプロットを行いましょう。

```
plt.gray()
plt.imshow(img_gray)
```



正しく表示されました。

グレースケール変換を施したことによって、R・G・B の 3 つのチャネルの画像が 1 つのチャネルになりましたが、これはサイズを確認することでわかります。

```
img_gray.shape
```

```
(512, 512)
```

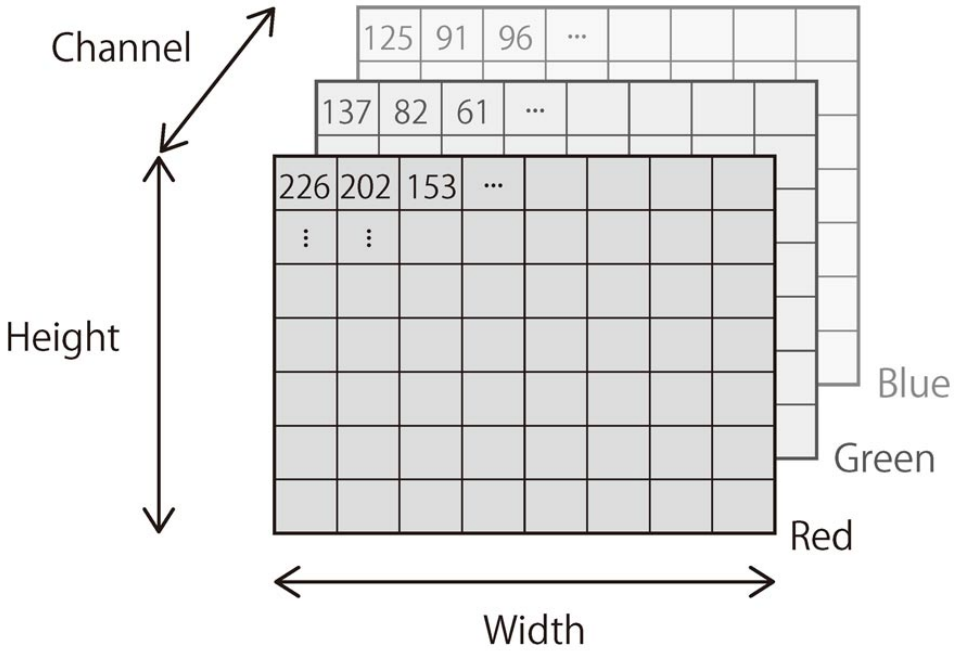
6.2 畳み込みニューラルネットワーク

ディープラーニングのブームは、画像解析において、ILSVRC（ImageNet Large Scale Visual Recognition Competition）のコンペティションで従来の解析手法よりもディープラーニングが精度を上回った頃から始まったと言われています。それから現在まで 8 年ほど経ちますが、画像処理の領域において目覚ましい進展を遂げています。

画像の特徴抽出

ここまでに全結合のニューラルネットワークを学びましたが、画像をどのように入力変数として全結合のニューラルネットワークに取り入れたらよいでしょうか。一般的にカラー画像は、光の三原色である Red、Green、Blue の 3 枚の画像を重ね合わせて表現しています。各ピクセ

ルにはそれぞれの輝度が格納されており、コンピュータで扱う場合には 0～255 の符号なし整数（unsigned integer）の 8 ビット値で表現することが一般的です。また、高さ（Height）、幅（Width）、チャネル（Channel）の 3 つが画像には存在します。数学的には行列をまとめたものなので、テンソルに相当します。



本題に戻ると、このように高さ、幅、チャネルの 3 次元で表される画像を、どうやってベクトルを入力変数とする全結合のニューラルネットワークで扱ったらよいでしょうか。まず、次のように横方向に切り取って連結していく方法が思いつきます。

$$\mathbf{x}^{\mathrm{T}} = [[\bigcirc\bigcirc\cdots\bigcirc]\cdots[\bigcirc\bigcirc\cdots\bigcirc]\cdots[\bigcirc\bigcirc\cdots\bigcirc]\cdots]$$

この方法であればベクトルとして扱うことができるので、全結合のニューラルネットワークで扱えます。実務ではこの例のように、どのように入力変数として扱うかなどを考える機会が多いものです。

では、この単純に切り出して連結する方法に何か欠点はあるのでしょうか（それともこれが完璧な方法でしょうか）。まず 1 点目として、すべてのチャネルにおいて画素単位で切り出して連結しているため、画像の持つ位置情報を失ってしまっていることが問題として挙げられます。2 点目としては、画像サイズが大きい場合、入力変数の数が膨大となる可能性があります。たとえば、一般的に使われているディスプレイの解像度は 1920×1080 ですので、画像全体の画素

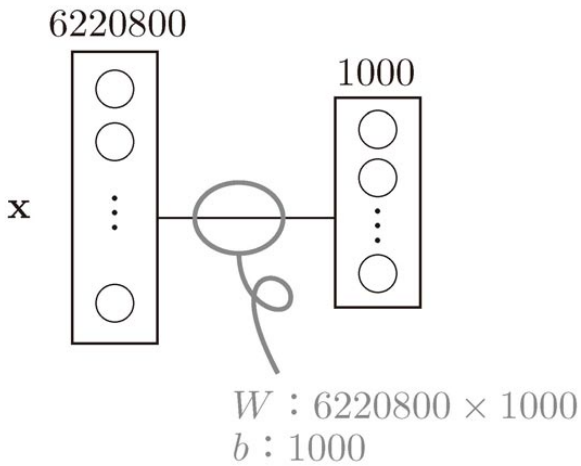
数は

$$1920 \times 1080 \times 3 = 6220800$$

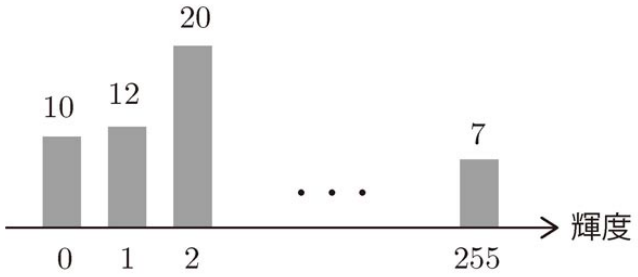
となります。これを入力変数として、次の層のノードの数を 1000 とした場合、パラメータの数は

$$6220800 \times 1000 + 1000 = 6220801000$$

となり、約 62 億個のパラメータを調整しなければなりません。高々数千～数万枚のデータセットによってこのパラメータの数を調整するのは現実的とは言えません。

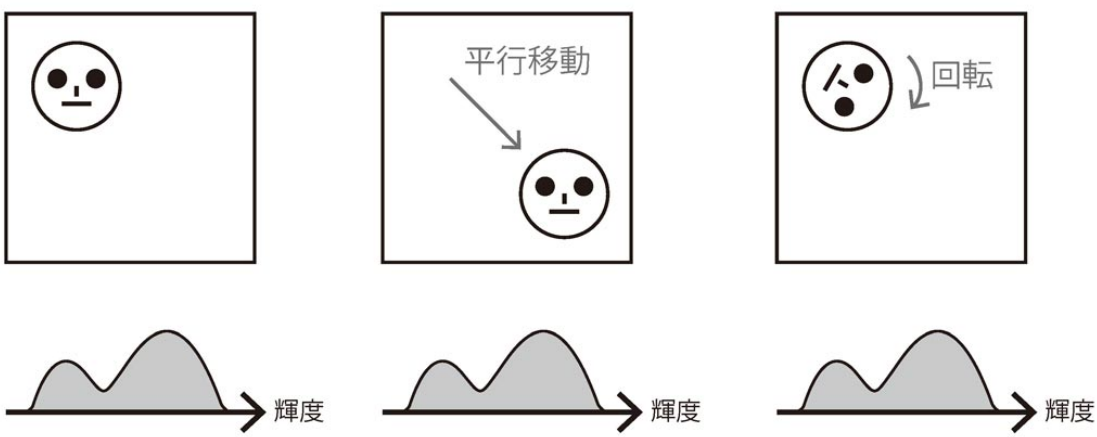


そこで次の手段として、**ヒストグラム**の採用を考えます。ヒストグラムは画素の輝度の値をカウントします。下図のように、ヒストグラムではどのような画像サイズに対しても、0～255 の 256 次元のベクトル（RGB の 3 チャネルを考慮した場合 $256 \times 3 = 768$）となるため、前述のパラメータの数を大幅に減らせます。第 1 章でも説明しましたが、このように、オリジナルの入力の値を別の値に変換して取り扱いやすくする処理を特徴抽出と呼び、取り出された値を特徴量と呼ぶのでした。



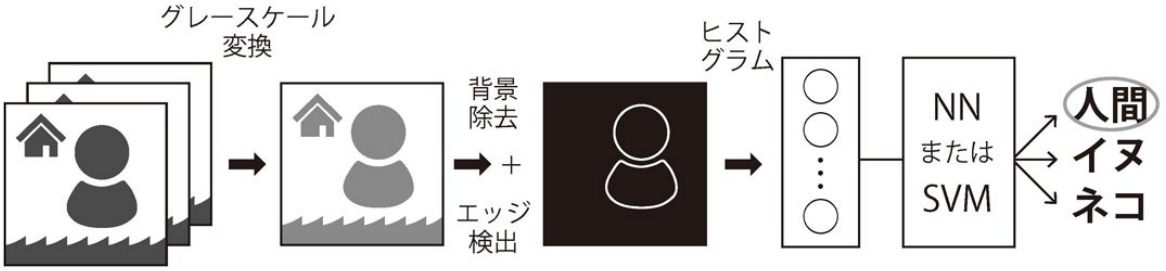
$$\mathbf{x} = \begin{bmatrix} 0 & 1 & 2 & \cdots & 255 \\ 10 & 12 & 20 & \cdots & 7 \end{bmatrix}$$

このヒストグラムの値を入力変数とすることの利点は、入力となるベクトルの次元数が減るだけではありません。色の輝度のカウントを用いるので、次の図に示すように**平行移動**や**回転**に対してもヒストグラムが変化せず、平行移動や回転に対して堅牢な手法です。これは実用上極めて大事な観点です。カメラで撮影した写真の顔が常に同じ位置に映っていることはまずなく、この対処が特徴量変換では望ましいのです。もしくは、データセット内にいろいろな位置や回転した顔の画像があれば、それをもとに学習するため、必然的に堅牢にすることもできますが、画像の枚数を豊富に準備できることが前提となり、そう簡単にはいかないこともままあります。特徴量を抽出する前処理の段階で、この情報を吸収できるのが望ましいでしょう。



実際に特徴抽出を利用した画像解析の流れを下図に示します。画像内の物体を人間か犬か猫に分類したいような場合、一般的に白黒のグレースケール画像に変換し、必要のない背景を除去して、ヒストグラムでベクトルに変換します。ベクトルに変換された値を入力変数として、ニューラルネットワークや後述するサポートベクターマシン（SVM）などを学習し、分類するモデルを作ります。これが一世代前までの画像処理の流れでした（もちろん、今でもよく使われてい

ます)。

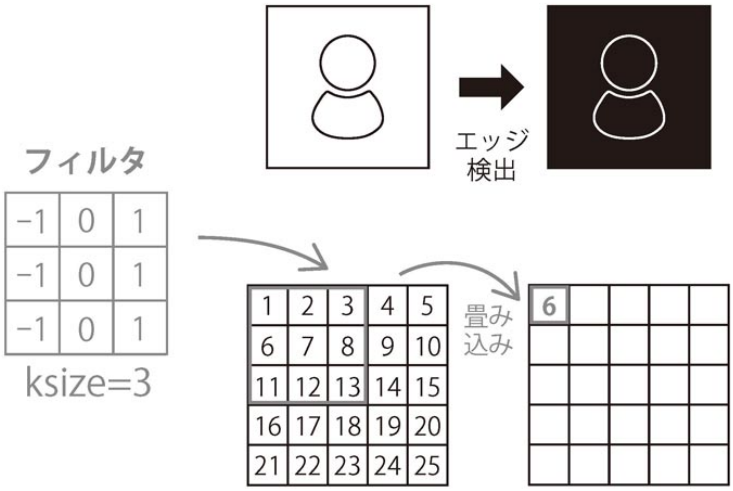


しかし、ここで問題があることに気付かれたでしょうか。一見それらしい流れですが、具体的にどのように背景除去を行えばよいのでしょうか。また、ヒストグラムに変換することが最適な特徴抽出と言えるでしょうか。人間であれば肌色といった色の情報をベースに進めればよいでしょうが、それが犬や猫にも同じように適用できるでしょうか。そもそも背景除去を行わなければならないのでしょうか。考え出すときりがないくらい、この流れには指摘すべき点が存在します。

仮にこの流れでよいと仮定しても、背景除去などを実装するためには画像処理に精通したエンジニアがうまく除去できるようなアルゴリズムを実装して、チューニングして、……と短くても1日、長ければ1か月以上時間がかかる作業であり、経験と勘に大きく左右されるという欠点もあります。機械学習では理論上筋が通っていても、業務上困難な場合があるため、この落とし穴も押さえておく必要があります。

この問題を解決する糸口が、この節で扱う畳み込みニューラルネットワーク（CNN：Convolutional Neural Network）ですので、楽しみにしておいてください。CNNの説明の前に、そもそも従来の画像処理ではどのように背景除去やエッジ検出を行っていたかを考えましょう。

フィルタ

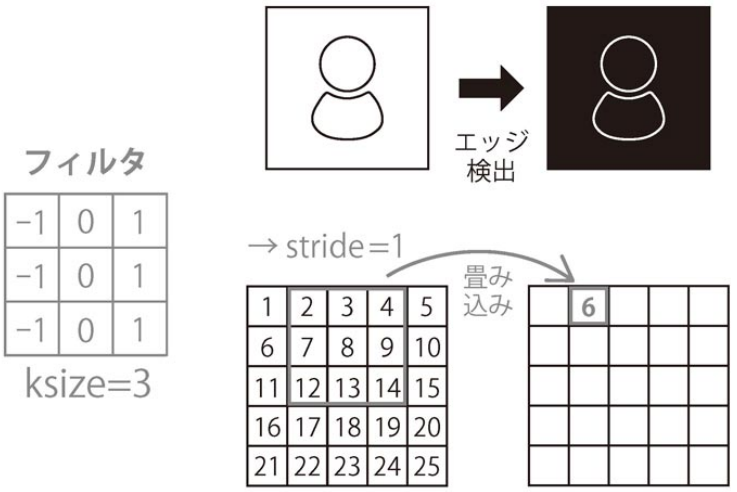


エッジ検出の処理を例に、画像処理の実装方法について考えます。画像処理では**フィルタ**と呼ばれるものがあり、これをかけることによって、画像が変換されます。フィルタは**カーネル**と呼ばれることもあります。

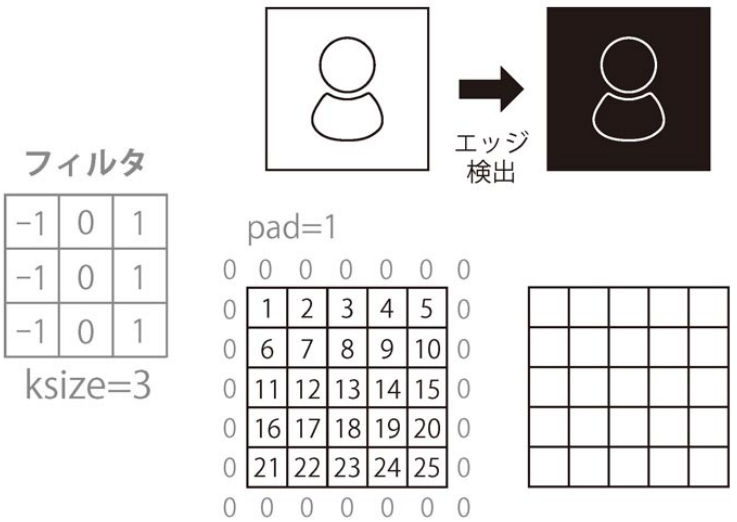
フィルタをかけるとは具体的にはどのような処理かというと、フィルタの値を重みとして、画像の輝度を入力として線形結合の計算を行計算を行うことです。フィルタをかけた後の値は

$$
\begin{aligned}
&-1\times1+0\times2+1\times3\\
&-1\times4+0\times5+1\times6\\
&-1\times7+0\times8+1\times9\\
&=6
\end{aligned}
$$

となります。このフィルタをかける処理のことを**畳み込み**（Convolution）と呼びます。ここではフィルタ（カーネル）のサイズを `ksize=3` としています（プログラミングの際にこのような変数名で登場することが多いです）。そして、フィルタをかける位置を 1 つずらして、また畳み込みの演算を行います。これを画像すべてに対して適用していきます。このフィルタをずらす量を**ストライド**と呼び、今回は 1 つずつずらすので `stride=1` となります。



もう 1 つ考慮すべきなのは、畳み込みを行った後の画像サイズです。ksize=3 で stride=1 の場合、畳み込み後の画像が 3×3 になることは想像がつくでしょうか。元の画像サイズが 100×100 の場合、畳み込み後のサイズは 98×98 となります。畳み込み後に画像サイズが変わると、プログラミングの裏側の処理が複雑になります。そこで、画像の周囲を 0 で埋める**パディング**と呼ばれる前処理を行います。これにより、たとえば 100×100 の画像では、全周囲に 1 つずつ 0 で埋めるパディング（pad=1）を行うと 102×102 となり、これに対してフィルタのサイズ 3（ksize=3）の畳み込みを適用すると、畳み込み後に画像が 100×100 となって、元の画像サイズが保たれます。

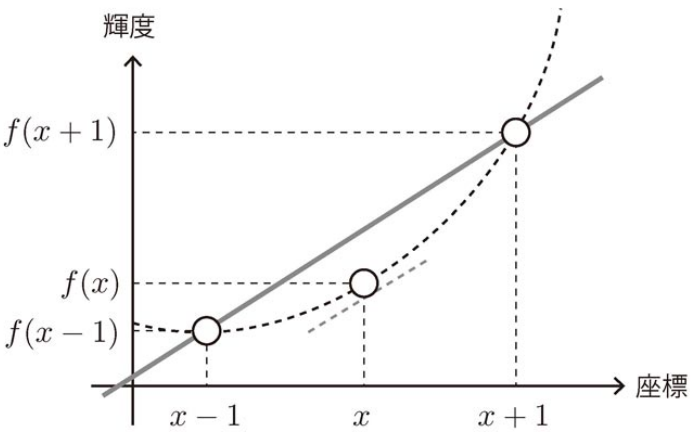


このように、画像処理ではフィルタを決めて畳み込み演算を行い、画像の変換処理を施します。ここで、なぜエッジ検出に

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

のような値を使ったのでしょうか。このようなフィルタをかけるとなぜエッジを検出できるのでしょうか。

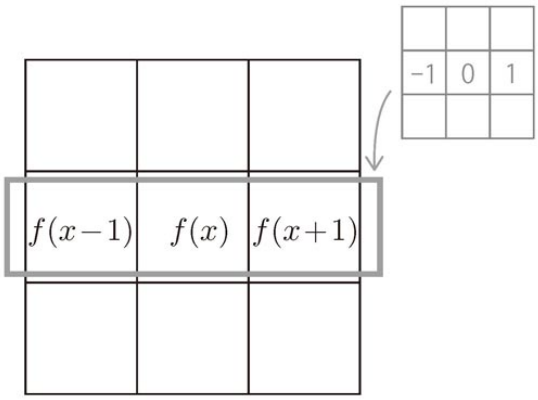
まず、物体のエッジ（輪郭）を捉える際、周囲の物体に対して人間がどのように判断しているかを考えてみましょう。おそらく、色合いが大きく異なる箇所で物体の輪郭を感じるでしょう。そのため、エッジを検出するためには、輝度の変化量を用いればよさそうです。変化量は微分を使って求められます。



前図のように、各座標 x に対する輝度を表す関数 $f(x)$ があり、この勾配 $f'(x)$ が変化量を表しています。ただし、画像内で座標情報から輝度情報を表現する関数 $f(x)$ を求めることは現実的に困難です。一方、前後の座標における輝度 $f(x-1)$ と $f(x+1)$ は求まっているので、2 点を通る直線の傾き

$$\begin{aligned} a &= \frac{f(x+1)-f(x-1)}{(x+1)-(x-1)} \\ &= \frac{f(x+1)-f(x-1)}{2} \end{aligned}$$

で近似できます。



上図のようにフィルタをかけると、

$$
\begin{aligned}
&-1 \times f(x-1) + 0 \times f(x) + 1 \times f(x+1) \\
&= f(x+1) - f(x-1)
\end{aligned}
$$

が得られます。定数項を無視すると、変化量として計算したい値と一致することがわかります。つまり、エッジ検出として紹介していたフィルタの値は適当に決めたものではなく、この変化量を取得したいという意図があって設定されたものでした。一見単なる簡単な数値に見えるものも、れっきとした意味を持っているのです。

例として紹介した横方向の微分フィルタを実装していきます。前述しましたがフィルタのことをカーネルとも呼び、サンプルのソースコードでは kernel という変数名がよく使われます。

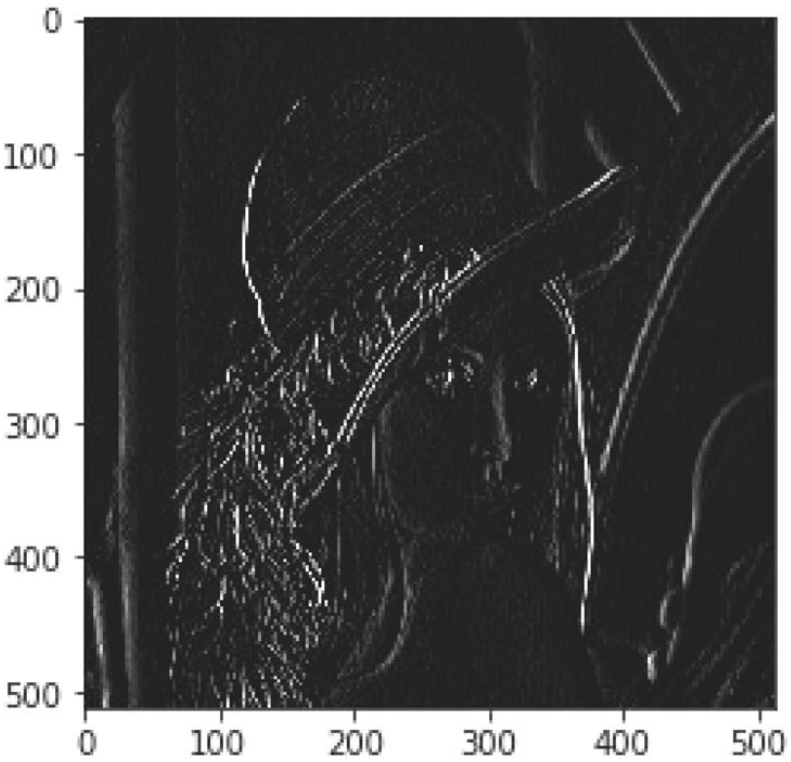
```
# 横方向の微分フィルタの定義
kernel = np.array([
    [-1, 0, 1],
    [-1, 0, 1],
    [-1, 0, 1]
])
```

畳み込み（Convolution）の演算を行いましょう。

```
# 畳み込み演算
img_conv = cv2.filter2D(img_gray, -1, kernel)
```

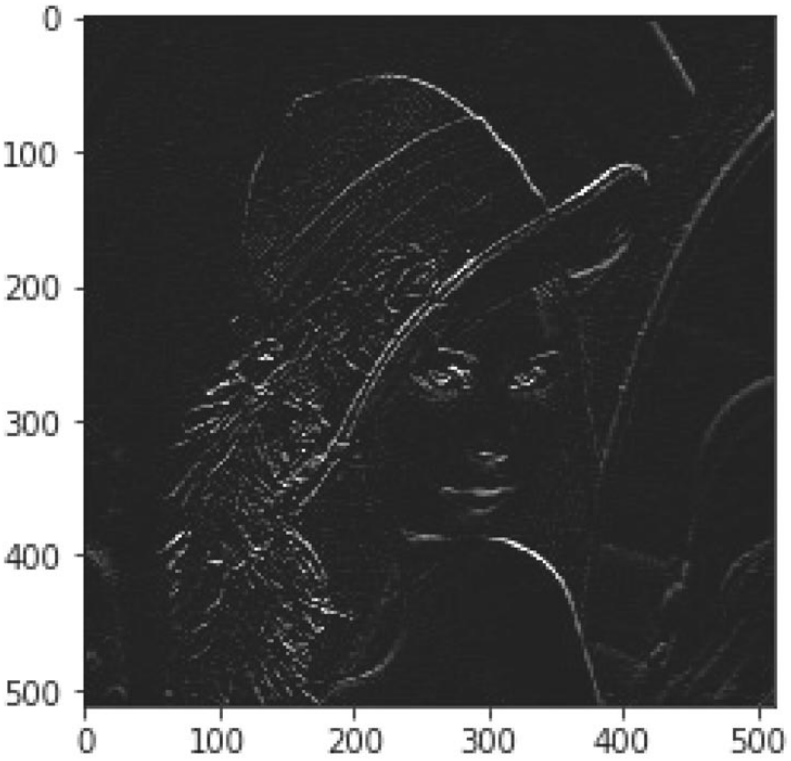
こちらをプロットしてみましょう。

```
plt.gray()
plt.imshow(img_conv)
```



横方向に対して確かにエッジが検出できていることが直感的にわかります。
縦方向も試してみましょう。

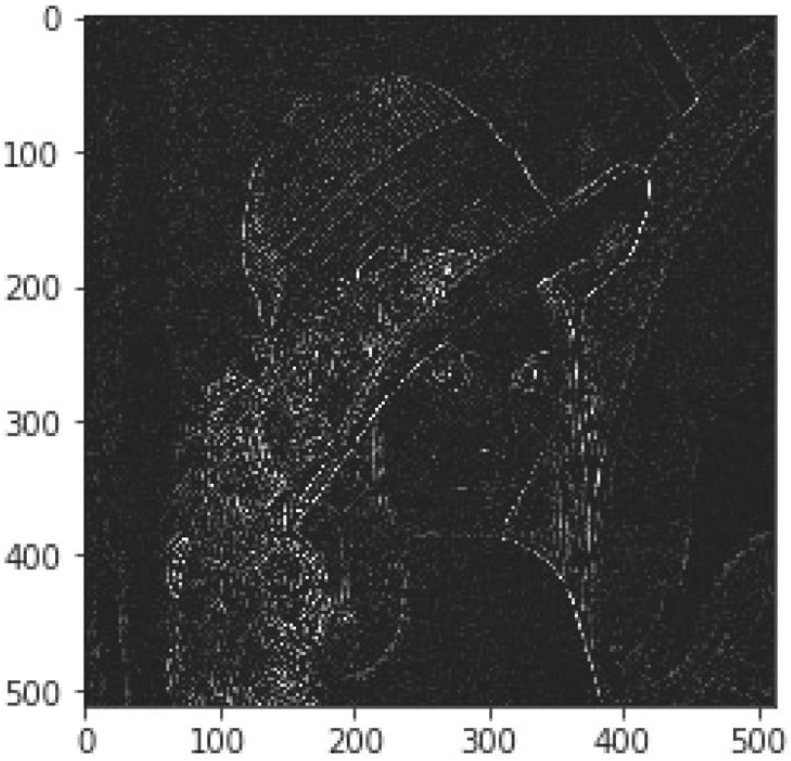
```
# 縦方向の微分フィルタの定義
kernel = np.array([
    [-1, -1, -1],
    [  0,  0,  0],
    [  1,  1,  1]
])

# 畳み込み演算
img_conv = cv2.filter2D(img_gray, -1, kernel)

plt.gray()
plt.imshow(img_conv)
```



別の代表的なフィルタとして、**ラプラシアンフィルタ**と**平滑化フィルタ**を紹介します。

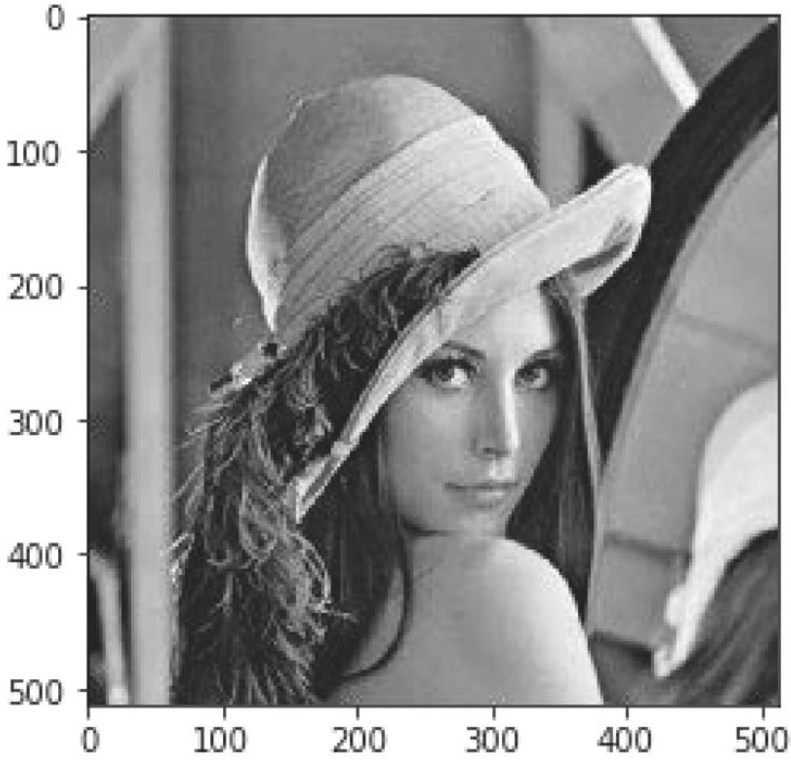
前掲の微分フィルタは一次微分の情報を利用していましたが、ラプラシアンフィルタでは二次微分を利用します。二次微分とは、一度微分した情報をさらに微分した情報です。ラプラシアンフィルタは縦横の両方向を考慮することができます。

```
# ラプラシアンフィルタの定義
kernel = np.array([
    [1, 1, 1],
    [1, -8, 1],
    [1, 1, 1]
])

img_conv = cv2.filter2D(img_gray, -1, kernel)

plt.gray()
plt.imshow(img_conv)
```



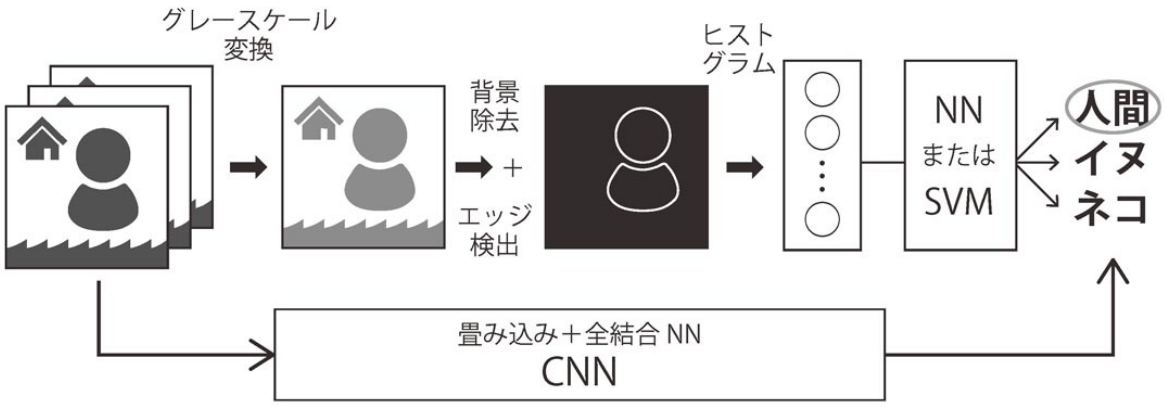
平滑化フィルタは、周りの画素の平均をとり、ノイズを除去する用途などで用いられるフィルタです。フィルタが 3×3 の 9 要素で構成される場合、それぞれのフィルタの重みが $1/9$ となります。

```
# 平滑化フィルタの定義
kernel = np.array([
    [1, 1, 1],
    [1, 1, 1],
    [1, 1, 1]
]) / 9

img_conv = cv2.filter2D(img_gray, -1, kernel)

plt.gray()
plt.imshow(img_conv)
```

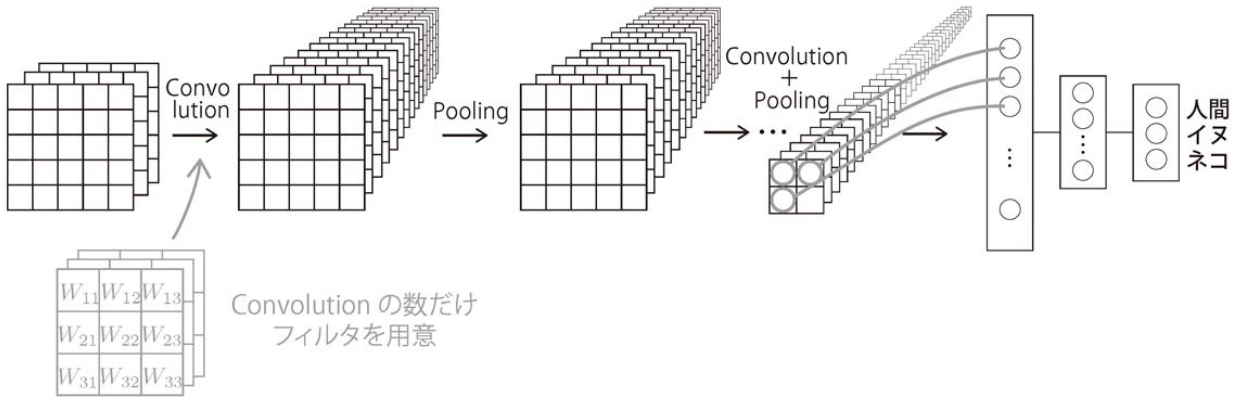


フィルタからCNNへ

エッジ検出のフィルタの値はわかりましたが、次の問題として、犬と猫を判別するためのフィルタの値はいくつでしょうか。また、背景を除去できるフィルタの値はいくつでしょうか。こう考えると、一見便利そうに見えるフィルタでしたが、そのフィルタの値を決定することができなければ使うことができず、このフィルタの値を決めるのが難しいタスクであることがわかります。



従来の画像解析が前ページの図の流れだとすると、これから紹介する CNN は、この畳み込みと全結合 NN の働きを一体化させ、このプロセスをすべてニューラルネットワークの中に包括したものです。CNN の出現により、以前の経験と勘によって行われていた前工程を自動化できるようになりました。



CNN は従来使われてきた画像処理のフィルタから着想を得ており、人間と犬と猫を判別できるようなフィルタを経験と勘で求めることが難しいのであれば、これも一種のパラメータとして学習させればよいと考えました。フィルタをかけて新しい画像を生成する処理を **Convolution**、画像を縮小させる処理を **Pooling** と呼びます。新たに変換した画像を生成し、これを 1/2 などに縮小していく処理を交互に繰り返していきます。一般に Convolution の処理では、もともと 3 チャネルであった画像を 64 チャネル、128 チャネル、256 チャネル、……とチャネルを増やしていきます。そして、画像を縮小していくと、徐々に全結合のニューラルネットワークに形が近づいていく様子が見てとれます。このように、大事な情報を抽出しながら徐々に小さくしていきます。こうすることで、最初からベクトルとして切り取ってしまうヒストグラムに比べて、上下左右の位置関係も比較的考慮できます。

6.3 CNN の計算

データセットの確認

今度のデータセットは犬と猫ではなく、よく使われる **MNIST** という 0〜9 までの手書き文字を扱います。画像に関するデータセットやよく使う処理は **Torch Vision** というライブラリにまとめられているため、その使い方も覚えていきましょう。

```
import torch
import torchvision
from torchvision import transforms

torch.__version__, torchvision.__version__
```

```
('1.3.1', '0.4.2')
```

transforms.Compose の中で、データを読み込んだ後に行う処理を定義します。PyTorch で学習するには torch.Tensor という型に変更する必要があるため、ToTensor() を使います。

```
transform = transforms.Compose([
    transforms.ToTensor()
])
```

torchvision.datasets から MNIST を読み込みます。

```
train = torchvision.datasets.MNIST(root='data', train=True, download=True, transform=transform)
```

データセットの中身を確認してみましょう。

```
train
```

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: data
    Split: Train
    StandardTransform
Transform: Compose(
               ToTensor()
           )
```

```
# サンプル数
len(train)
```

```
60000
```

```
# 入力値と目標値がタプルで格納
type(train[0])
```

```
tuple
```

タプルで入力値と目標値が格納されているため、要素番号を指定することでそれぞれ確認できます。

```
# 入力値
train[0][0]
```

```
tensor([[[0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
 ...
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000]]])
```

```
# 目標値
train[0][1]
```

```
5
```

```
# 入力値のサイズ -> (0:channel, 1:height, 2:width)
train[0][0].shape
```

```
torch.Size([1, 28, 28])
```

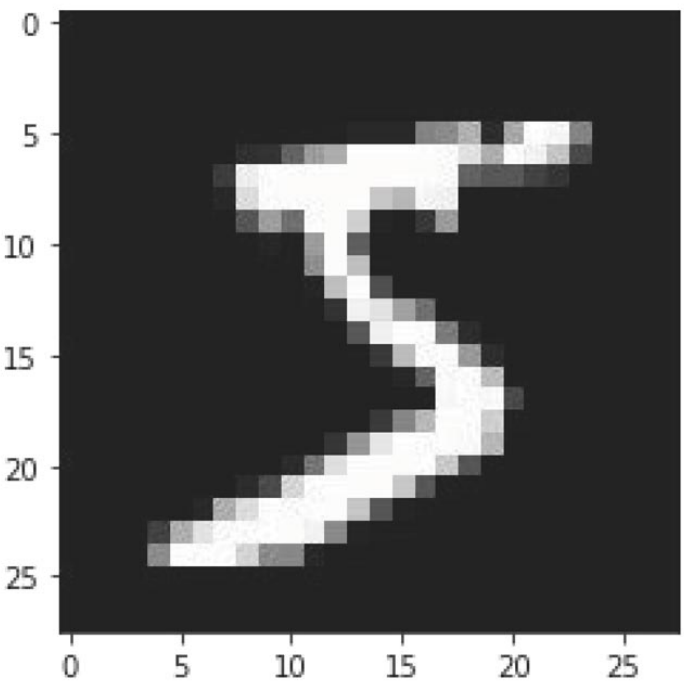
入力値のサイズに注目すると、画像が (height, width, channel) の順ではなく、(channel, height, width) の順に格納されていることがわかります。CNN で扱う際には、この順に並べておかなければなりません。ただし、Torch Vision を使う場合、こういった処理も自動的に内部で行ってくれますので、実装上は特に気にしなくても問題ありません。

それでは、試しにデータセットのサンプルの 1 つをプロットしてみましょう。

```
# (0:channel, 1:height, 2:width) -> (1:height, 2:width, 0:channel)
img = np.transpose(train[0][0], (1, 2, 0))

img.shape
```

```
torch.Size([28, 28, 1])
```

```
img = img.reshape(28, 28)

plt.gray()
plt.imshow(img)
```



特徴抽出

テスト用の画像を 1 つ用いて、convolution → pooling → fc の処理の流れを確認していきましょう。

```
import torch.nn as nn
import torch.nn.functional as F
```

MNIST の画像の 1 つ目を使用します。

```
x = train[0][0]
```

サイズを確認してみましょう。

```
x.shape
```

```
torch.Size([1, 28, 28])
```

こちらは、すでに PyTorch で扱うことのできる (channel, height, width) のサイズとなっているため、そのまま使用します。

PyTorch で実装された Convolution を使いましょう。torch.nn に Conv2d が用意されています。引数の内容は以下のとおりです。

- in_channels：入力のチャネルの数
- out_channels：出力のチャネルの数
- kernel_size：フィルタのサイズ
- stride：フィルタを動かす幅（デフォルト値は 1）
- padding：画像の外側を囲う数（デフォルト値は 0）

```
# 畳み込み層の定義
conv = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1, padding=1)
```

宣言した時点で、フィルタの重みがランダムに割り振られます。中身を確認してみましょう。

```
conv.weight
```

```
Parameter containing:
tensor([[[[-0.3171,  0.2042, -0.2792],
          [ 0.1189, -0.0790,  0.2038],
          [-0.3249,  0.2486,  0.1692]]],


        [[[ 0.1925,  0.1097, -0.0998],
          [ 0.1183, -0.0529, -0.0448],
          [-0.2012, -0.0937, -0.3142]]],


        [[[ 0.3256, -0.3006,  0.0958],
          [ 0.1870, -0.2193, -0.2407],
          [ 0.1866, -0.0917, -0.2225]]],


        [[[ 0.0203,  0.2160,  0.0926],
          [-0.1871, -0.2142, -0.1378],
          [-0.2802,  0.3110,  0.3094]]]], requires_grad=True)
```

```
conv.weight.shape
```

```
torch.Size([4, 1, 3, 3])
```

サイズは (出力画像のchannel，入力画像のchannel，height，width) となっています。つまり、この場合は (1，3，3) のフィルタが 4 セット生成されています。channel=3 のカラー画像に対してフィルタをかける場合は、(3，3，3) のフィルタが 4 セット生成されることになります。

それでは、定義した conv を入力値 x に対して適用しましょう。通常はミニバッチ単位で入力するため、サイズは (channel，height，width) ではなく、(batchsize，channel，height，width) となります。

```
# batchsize=1となるようにリサイズ => (batchsize, channels, height, width)
x = x.reshape(1, 1, 28, 28)

# 畳み込み演算
x = conv(x)
x
```

```
tensor([[[[ 0.1382,  0.1382,  0.1382,  ...,  0.1382,  0.1382,  0.1382],
          [ 0.1382,  0.1382,  0.1382,  ...,  0.1382,  0.1382,  0.1382],
          [ 0.1382,  0.1382,  0.1382,  ...,  0.1382,  0.1382,  0.1382],
          ...,
          [ 0.1382,  0.1382,  0.1382,  ...,  0.1382,  0.1382,  0.1382],
          [ 0.1382,  0.1382,  0.1382,  ...,  0.1382,  0.1382,  0.1382],
          [ 0.1382,  0.1382,  0.1382,  ...,  0.1382,  0.1382,  0.1382]],

         [[-0.2280, -0.2280, -0.2280,  ..., -0.2280, -0.2280, -0.2280],
          [-0.2280, -0.2280, -0.2280,  ..., -0.2280, -0.2280, -0.2280],
          [-0.2280, -0.2280, -0.2280,  ..., -0.2280, -0.2280, -0.2280],
          ...,
          [-0.2280, -0.2280, -0.2280,  ..., -0.2280, -0.2280, -0.2280],
          [-0.2280, -0.2280, -0.2280,  ..., -0.2280, -0.2280, -0.2280],
          [-0.2280, -0.2280, -0.2280,  ..., -0.2280, -0.2280, -0.2280]],

         [[-0.2743, -0.2743, -0.2743,  ..., -0.2743, -0.2743, -0.2743],
          [-0.2743, -0.2743, -0.2743,  ..., -0.2743, -0.2743, -0.2743],
          [-0.2743, -0.2743, -0.2743,  ..., -0.2743, -0.2743, -0.2743],
          ...,
          [-0.2743, -0.2743, -0.2743,  ..., -0.2743, -0.2743, -0.2743],
          [-0.2743, -0.2743, -0.2743,  ..., -0.2743, -0.2743, -0.2743],
          [-0.2743, -0.2743, -0.2743,  ..., -0.2743, -0.2743, -0.2743]],

         [[ 0.2169,  0.2169,  0.2169,  ...,  0.2169,  0.2169,  0.2169],
          [ 0.2169,  0.2169,  0.2169,  ...,  0.2169,  0.2169,  0.2169],
          [ 0.2169,  0.2169,  0.2169,  ...,  0.2169,  0.2169,  0.2169],
          ...,
          [ 0.2169,  0.2169,  0.2169,  ...,  0.2169,  0.2169,  0.2169],
          [ 0.2169,  0.2169,  0.2169,  ...,  0.2169,  0.2169,  0.2169],
          [ 0.2169,  0.2169,  0.2169,  ...,  0.2169,  0.2169,  0.2169]]],
       grad_fn=<MkldnnConvolutionBackward>)
```

```
x.shape
```

```
torch.Size([1, 4, 28, 28])
```

(batchsize, channels, height, width) なので、channels が 4 に増えたことがわかります。

次に、Pooling を実装してみましょう。Pooling は torch.nn.functional に max_pool2d として用意されています。カーネルサイズが (2, 2) の Pooling は、画像のサイズが半分になります。

```
# Poolingの演算
x = F.max_pool2d(x, kernel_size=2, stride=2)
```

サイズを確認してみましょう。

```
x.shape
```

```
torch.Size([1, 4, 14, 14])
```

(1，4，28，28) → (1，4，14，14) と、縮小されていることがわかります。

このようにして、画像の特徴を抽出できました。

全結合層と結合

畳み込み層で取得した値を全結合層に入力していきます。ここで問題となるのが、データのサイズです。上記で求めた値は、1 サンプルにつき (4，14，14) のテンソルで定義されていますが、全結合層に入力するときは**ベクトルでなければなりません**。そこで、「テンソル→ベクトル」に変換する **Flatten** と呼ばれる処理が必要になります。

サイズは (4，14，14) なので、ベクトル化するには、まず $4 \times 14 \times 14$ の値を求めます。

```
print('channel :', x.shape[1])
print('height :', x.shape[2])
print('width :', x.shape[3])
```

```
channel : 4
height : 14
width : 14
```

```
x_shape =  x.shape[1] * x.shape[2] * x.shape[3]
x_shape
```

```
784
```

この数値をもとに (サンプル数，ベクトルの要素数) のサイズに変更すれば、全結合層に入力できます。サイズを変更するには view 関数を使います。

```
# 今回はベクトルの要素数が決まっているため、サンプル数は自動で設定
# -1とするともう片方の要素に合わせて自動的に設定される
x = x.view(-1, x_shape)

x.shape
```

```
torch.Size([1, 784])
```

```
# 全結合層の定義
fc = nn.Linear(x_shape, 10)  # 784 => 10

# 線形変換
x = fc(x)

x.shape
```

```
torch.Size([1, 10])
```

このように、畳み込み演算から全結合層での線形変換までの流れを確認できました。

6.4 画像分類（MNIST）

それでは、実際に CNN のネットワークを学習しましょう。データセットには MNIST を使用し、0～9 までの手書き文字を分類する問題に取り組みます。

データセットの準備

torchvision.datasets から MNIST の画像データセットを読み込みます。trian、valid、学習用データセット（train）、検証用データセット（valid）、テスト用データセット（test）に切り分ける流れはこれまでどおりです。transform は前節で定義した torch.Tensor 形式に変換する処理を用います。

```
# データセットの取得（データがない場合はダウンロード）
train_val = torchvision.datasets.MNIST(root='data', train=True, download=True, transform=transf⇨
orm)
test = torchvision.datasets.MNIST(root='data', train=False, download=True, transform=transform)

# train : val = 0.8 : 0.2
n_train = int(len(train_val) * 0.8)
n_val = len(train_val) - n_train

# ランダムに分割を行うため、シードを固定して再現性を確保
torch.manual_seed(0)

# trainとvalを分割
train, val = torch.utils.data.random_split(train_val, [n_train, n_val])

# Data Loaderを用意
batch_size = 32

train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True, drop_last=True)
val_loader = torch.utils.data.DataLoader(val, batch_size)
test_loader = torch.utils.data.DataLoader(test, batch_size)
```

ネットワークの定義と学習

前節までと同じ流れです。フルカラー画像なので、in_channels が 3 となることに注意しましょう。

```
from torch.utils.data import DataLoader
import pytorch_lightning as pl
from pytorch_lightning import Trainer

class Net(pl.LightningModule):

    def __init__(self):
        super().__init__()

        # 畳み込み層を定義
        self.conv = nn.Conv2d(in_channels=1, out_channels=3, kernel_size=(3, 3))
        # 全結合層を定義
        self.fc = nn.Linear(507, 10)

        self.train_acc = pl.metrics.Accuracy()
        self.val_acc = pl.metrics.Accuracy()
        self.test_acc = pl.metrics.Accuracy()

    def forward(self, x):
```

```
        # 畳み込み
        h = self.conv(x)
        # 最大値プーリング
        h = F.max_pool2d(h, kernel_size=(2, 2), stride=2)
        # ReLU 関数
        h = F.relu(h)
        # ベクトル化
        h = h.view(-1, 507)
        # 線形変換
        h = self.fc(h)
        return h

    def training_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
        return optimizer
```

ネットワークの学習

今回の学習では GPU を使用します。Trainer のインスタンス化の際、gpus の引数に使用する GPU の数を指定するだけで、GPU での演算に切り替えることが可能です。

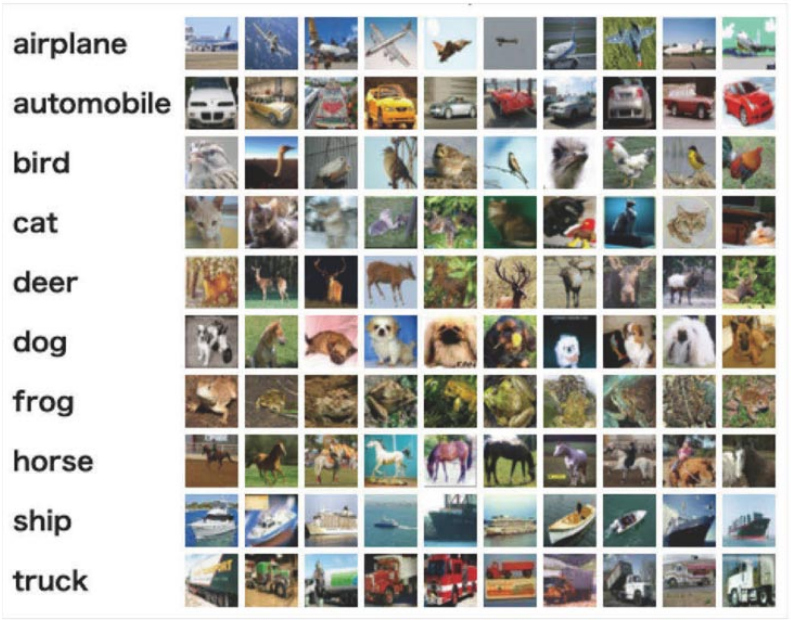
```
# 訓練の実行
pl.seed_everything(0)
net = Net()
trainer = pl.Trainer(max_epochs=10, gpus=1)
trainer.fit(net, train_loader, val_loader)

# テストデータで検証
results = trainer.test(test_dataloaders=test_loader)

# 最終的に得られた結果を確認
results
```

```
{'test_loss': 0.20699308812618256, 'test_acc': 0.939648449420929}
```

検証データとテストデータに対しても約 98% の正解率で分類できるモデルを構築できました。古典的な画像処理のフィルタを押さえておくことで、CNN は簡単に理解できます。

6.5　画像分類（CIFAR10）

MNIST の分類は簡単なネットワークの定義でもある程度の正解率が得られますが、もう少し難しい問題設定で試してみましょう。**CIFAR10** と呼ばれる、以下のような 10 クラスの分類を行います。CIFAR10 は MNIST のグレースケール画像とは異なり、フルカラー画像です。CIFAR10 も MNIST と同様に、Torch Vision にデータセットが用意されています。



データの準備

データのダウンロードからデータセットの分割まで、これまでと同じ流れで行いましょう。

```
# CIFAR10
train_val = torchvision.datasets.CIFAR10(root='data', train=True, download=True,
transform=transform)
test = torchvision.datasets.CIFAR10(root='data', train=False, download=True, transform=transform)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

```
# train : val = 0.8 : 0.2
n_train = int(len(train_val) * 0.8)
n_val = len(train_val) - n_train

# ランダムに分割を行うため、シードを固定して再現性を確保
torch.manual_seed(0)

# trainとvalを分割
train, val = torch.utils.data.random_split(train_val, [n_train, n_val])

# Data Loaderを用意
batch_size = 256
```

```
train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True, drop_last=True)
val_loader = torch.utils.data.DataLoader(val, batch_size)
test_loader = torch.utils.data.DataLoader(test, batch_size)
```

分割後のサンプル数を確認します。

```
len(train), len(val), len(test)
```

```
40000
```

ネットワークの定義と学習

今回はチャネルの数を 3 → 64 → 128 → 256 のように増やしていきながら、畳み込みのたびにハーフサイズの Pooling を行います。これは VGG16 という有名なネットワーク構造を参考に、さらに簡略化したネットワーク構造を採用しています。

```
class Net(pl.LightningModule):

    def __init__(self):
        super().__init__()

        # 畳み込み層を定義
        self.conv = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=(3, 3), padding=(1, 1))
        # 全結合層を定義
        self.fc = nn.Linear(1536, 10)

        self.train_acc = pl.metrics.Accuracy()
        self.val_acc = pl.metrics.Accuracy()
        self.test_acc = pl.metrics.Accuracy()

    def forward(self, x):
        # 畳み込み
        h = self.conv(x)
        # 最大値プーリング
        h = F.max_pool2d(h, kernel_size=(2, 2), stride=2)
        # ReLU 関数
        h = F.relu(h)
        # ベクトル化
        h = h.view(-1, 1536)
        # 線形変換
        h = self.fc(h)
        return h

    def training_step(self, batch, batch_idx):
        x, t = batch
```

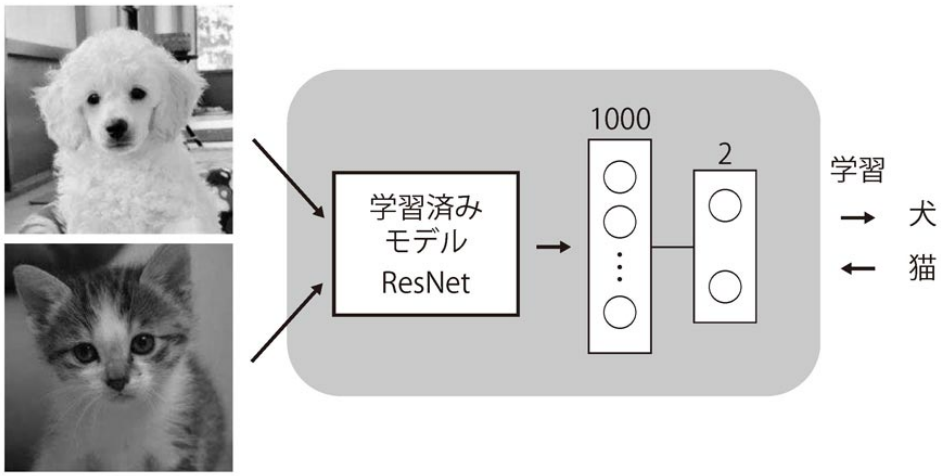
```
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
        return optimizer
[17]:
# 学習の実行
pl.seed_everything(0)
net = Net()
trainer = pl.Trainer(max_epochs=10, gpus=1)
trainer.fit(net, train_loader, val_loader)

# テストデータで検証
results = trainer.test(test_dataloaders=test_loader)
results
```

```
{'test_loss': 1.1822822093963623, 'test_acc': 0.607617199420929}
```

この結果から、10 クラスの分類であるため、ベースラインとなる正解率が 10% とすると、正解率を 60% までは上昇させることができています。ただ、それでも正解率は高いとは言えません。

6.6 ファインチューニング

ファインチューニングとは、異なるデータセットで学習済みのモデルに関して一部を再利用して、新しいモデルを構築する手法です。モデルの構造とパラメータをそのまま活用し、**特徴抽出器**としての機能を果たします。手持ちのデータセットのサンプル数が少ないために精度があまり出ない場合でも、ファインチューニングを使用すれば、性能が向上する可能性があります。



学習済みモデルの活用

学習済みモデルは世界中で公開されていますが、それぞれのタスクに合わせて、類似のデータセットで学習を行った学習済みモデルを使用する必要があります。

今回は、ILSVRC で使われている、1000 クラスの物体を分類するタスクの学習済みモデルを利用します。この ILSVRC の学習済みモデルはフレームワーク側で用意されていることが多く、簡単に使い始めることができます。まずは、この学習済みモデルで試してみるのがよいでしょう。また、ネットワークの構造は、ILSVRC にて 2015 年に優勝した **ResNet** というものを使用します。

```
from torchvision.models import resnet18

# ResNetを特徴抽出器として使用
model_conv = resnet18(pretrained=True)
```

resnet18 の引数である pretrained を False に設定すると、学習済みのパラメータは使用せずに、ネットワークの構造だけ利用できます。

また、学習済みモデルを利用するときには、学習時に使用していた画像サイズや正規化を合わせる必要があります。PyTorch の公式ページ（https://pytorch.org/docs/stable/torchvision/models.html）に指定があり、まず画像のサイズは 224×224 となります。画像の変換は transform に定義しておくと便利です。

```
from torchvision import transforms, datasets

# 訓練済みモデルに合わせた前処理を追加
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# データセットの取得
train_val = datasets.CIFAR10('./', train=True, download=True, transform=transform)
test = datasets.CIFAR10('./', train=False, download=True, transform=transform)

# trainとvalに分割
torch.manual_seed(0)
n_train, n_val = 40000, 10000
train, val = torch.utils.data.random_split(train_val, [n_train, n_val])

# バッチサイズの定義
batch_size = 256

# Data Loaderを定義
train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True, drop_last=True, ⇨
num_workers=8)
val_loader = torch.utils.data.DataLoader(val, batch_size, num_workers=8)
test_loader = torch.utils.data.DataLoader(test, batch_size)
```

学習用のデータを 1 サンプル使って、推論までの計算を確認していきましょう。

```
# 1サンプル目の入力値を抽出
x = train[0][0]

# サイズの確認
x.shape
```

```
torch.Size([3, 224, 224])
```

```
# 先頭に batch_sizeを追加（batch_size=1）
x = torch.reshape(x, (1, 3, 224, 224))

# サイズの確認
x.shape
```

```
torch.Size([1, 3, 224, 224])
```

```
# 推論
y = model_conv(x)

# サイズの確認
y.shape
```

```
torch.Size([1, 1000])
```

この結果からわかるように、学習したタスクが 1000 クラスの分類であったため、1000 個の値が出力値として得られています。

ネットワークの学習

それでは、一連の流れが確認できたので、学習済みモデルを用いたファインチューニングを行います。ファインチューニング時には活用する学習済みモデルのパラメータは学習させずに固定します。ネットワークの構成が大きくなったこと、そして画像のサイズが大きくなったことにより、学習にかかる時間は大幅に増えます。そのため、今回は最大 3 エポックに制限しておきましょう。

```
class Net(pl.LightningModule):

    def __init__(self):
        super().__init__()

        # 特徴抽出器（output：1000クラス）
        self.feature_extractor = resnet18(pretrained=True)
        # 訓練済みのパラメータを固定
        for param in self.feature_extractor.parameters():
            param.requires_grad = False

        # 全結合層
```

```
        self.fc = nn.Linear(1000, 10)

        self.train_acc = pl.metrics.Accuracy()
        self.val_acc = pl.metrics.Accuracy()
        self.test_acc = pl.metrics.Accuracy()

    def forward(self, x):
        h = self.feature_extractor(x)
        h = self.fc(h)
        return h

    def training_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True, prog_bar=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
        return optimizer

# 訓練の実行
pl.seed_everything(0)
net = Net()
trainer = pl.Trainer(max_epochs=10, gpus=1)
trainer.fit(net, train_loader, val_loader)

# テストデータで検証
results = trainer.test(test_dataloaders=test_loader)
results
```

```
{'test_loss': 0.6540393829345703, 'test_acc': 0.7758988738059998}
```

上記の結果のとおり、10 エポック分の学習で正解率を 77% まで高めることができました。さらなる学習やハイパーパラメータの調整で、正解率をより高められることが期待できます。

6.7 自家製データの利用

これまでは PyTorch 側で用意されていたデータセットを用いていましたが、実プロジェクトでは用意したデータに対して学習を実行するのが普通です。ここではその実装方法を紹介します。

今回は https://drive.google.com/open?id=1cvQc6BR6mZoidsvRRqaHI9AhPhWE8aLP で配布している cat_dog というフォルダに猫と犬の画像がそれぞれ 100 枚ずつ格納されており、この 2 クラスの分類を行います。まずは、画像データの一部を確認してみましょう。

```
img = Image.open('data/cat_dog/cat/1.png')
img
```

```
img = Image.open('data/cat_dog/dog/1.png')
img
```

それでは、PyTorch の学習に利用できる形に変換します。Torch Vision には datasets.ImageFolder というクラスが用意されており、ここにクラスごとにフォルダ分けした PATH を指定すれば、画像の変換からラベル付けまで自動で行ってくれます。ファインチューニングを利用するため、前回と同様に transform を用意します。今回も画像によってサイズが異なるため、リサイズは必須です。

```
# 画像の前処理
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# 前処理とラベル付け
dataset = torchvision.datasets.ImageFolder('data/cat_dog', transform)
dataset
```

```
Dataset ImageFolder
    Number of datapoints: 200
    Root location: data/cat_dog
    StandardTransform
Transform: Compose(
               Resize(size=(224, 224), interpolation=PIL.Image.BILINEAR)
               ToTensor()
               Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
           )
```

これで、いつも PyTorch で利用している形式に変更できました。データを確認しておきましょう。

```
# 1サンプル目の確認
# タプル形式で(x, t)が格納
dataset[0]
```

```
(tensor([[[ 1.1187,  0.9988,  0.9646,  ...,  1.5982,  1.6153,  1.7009],
          [ 1.0502,  0.9646,  0.9474,  ...,  1.3413,  1.3413,  1.4269],
          [ 1.0331,  0.9646,  0.9646,  ...,  1.4098,  1.4269,  1.5125],
          ...,
          [ 0.2282,  0.1939,  0.1939,  ...,  0.0912,  0.0912,  0.1083],
          [ 0.2624,  0.2282,  0.2111,  ...,  0.1426,  0.1426,  0.1939],
          [ 0.3138,  0.2796,  0.2624,  ...,  0.2453,  0.2282,  0.2967]],

         [[ 0.9580,  0.8880,  0.8880,  ...,  1.6583,  1.6758,  1.7633],
          [ 0.9055,  0.8529,  0.8880,  ...,  1.4132,  1.4307,  1.5182],
          [ 0.9230,  0.8880,  0.9230,  ...,  1.4832,  1.5007,  1.5882],
          ...,
          [-0.4426, -0.4776, -0.4776,  ..., -0.4076, -0.4076, -0.3901],
          [-0.4076, -0.4426, -0.4776,  ..., -0.3550, -0.3550, -0.3025],
          [-0.3550, -0.4076, -0.4251,  ..., -0.2500, -0.2675, -0.1975]],

         [[ 1.1411,  1.1062,  1.1237,  ...,  1.9428,  1.9603,  2.0474],
          [ 1.1062,  1.0714,  1.1237,  ...,  1.6465,  1.6640,  1.7511],
          [ 1.1062,  1.0888,  1.1585,  ...,  1.6291,  1.6465,  1.7337],
          ...,
          [-0.6715, -0.7238, -0.7064,  ..., -0.6367, -0.6367, -0.6193],
          [-0.6890, -0.7238, -0.7587,  ..., -0.5844, -0.5844, -0.5321],
          [-0.6367, -0.6890, -0.6890,  ..., -0.4973, -0.4973, -0.4101]]]), 0)
```

データセットを学習、検証、テスト用に分割します。

```
# 訓練 : 検証 : テスト = 60% : 20% : 20%
n_train = int(len(dataset) * 0.6)
n_val = int(len(dataset) * 0.2)
n_test = len(dataset) - n_train - n_valid

# データセットの分割
torch.manual_seed(0)
batch_size = 256

# Data Loaderを用意
train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True, drop_last=True)
val_loader = torch.utils.data.DataLoader(val, batch_size)
test_loader = torch.utils.data.DataLoader(test, batch_size)
```

それでは、ネットワークの定義を行い、学習させていきます。

```
class Net(pl.LightningModule):

    def __init__(self):
        super().__init__()

        # 特徴抽出器（output：1000クラス）
        self.feature_extractor = resnet18(pretrained=True)
        # 訓練済みのパラメータを固定
        for param in self.feature_extractor.parameters():
            param.requires_grad = False

        # 全結合層
        self.fc = nn.Linear(1000, 2)

        self.train_acc = pl.metrics.Accuracy()
        self.val_acc = pl.metrics.Accuracy()
        self.test_acc = pl.metrics.Accuracy()

    def forward(self, x):
        h = self.feature_extractor(x)
        h = self.fc(h)
        return h

    def training_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True, prog_bar=True)
        return loss
```

```
    def validation_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
        return optimizer

# 学習の実行
pl.seed_everything(0)
net = Net()
trainer = pl.Trainer(max_epochs=10, gpus=1)
trainer.fit(net, train_loader, val_loader)

# テストデータで検証
results = trainer.test(test_dataloaders=test_loader)
results
```

```
{'test_loss': 0.10261714458465576, 'test_acc': 0.949999988079071}
```

学習済みモデルで使用していたクラスの中に犬と猫も含まれていたため、簡単な学習で高い正解率を実現できていることがわかります。

本章では画像処理の基礎から CNN への道のり、ファインチューニングや自作データの読み込み方まで、幅広く紹介しました。

第7章 ハイパーパラメータの最適化

本章では、ハイパーパラメータの最適化方法について紹介していきます。Ax による最適化方法を第 4 章で紹介しましたが、本章ではクラウドを使ったさらに効率的な調整方法を紹介します。

7. Azure ML

Azure ML（Azure Machine Learning）は、ハイパーパラメータの調整など試行錯誤を楽に行うことができるツールの集まりです。特に、マルチノードへの分散処理の展開や不要なノードのシャットダウンなども自動的に行ってくれるため、必要なリソースを必要なだけ効率的に使えます。

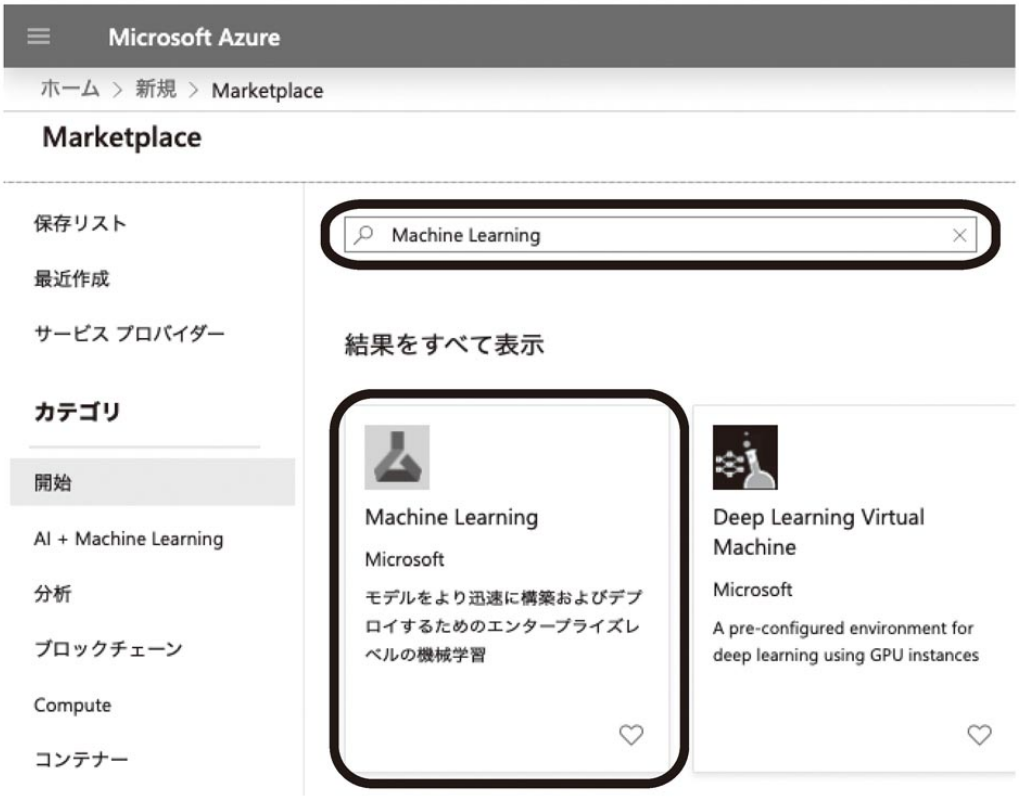
これまでは GPU を搭載した仮想マシン（VM：Virtual Machines）を利用していましたが、実験が大規模になってきたら、こちらを使うことをお勧めします。

7.2 Azure ML の立ち上げ

Azure ポータル上から Azure ML の設定を行います。1 つずつ設定していきましょう。

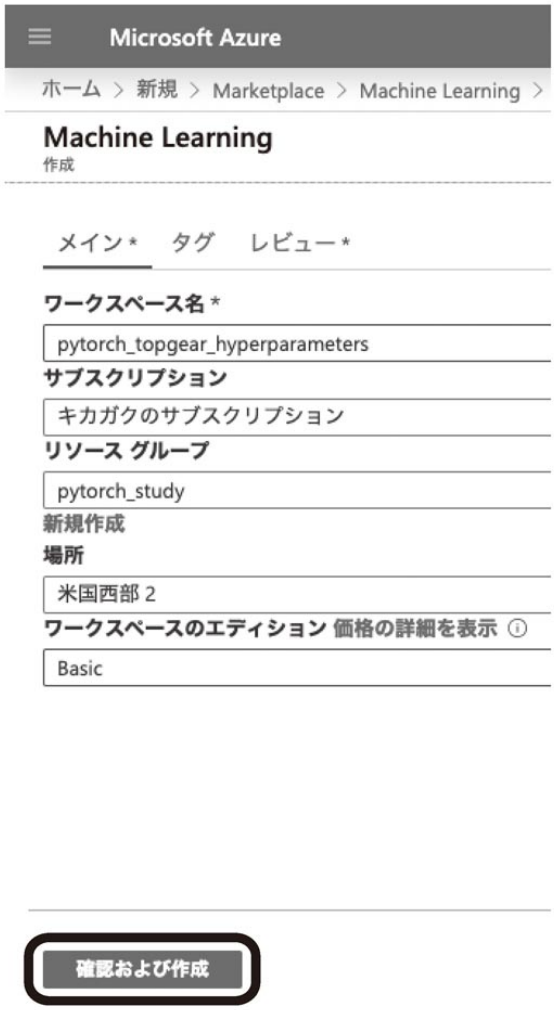
リソースの作成

「新規」から「Machine Learning」を検索してください。



「Machine Learning」を選択し、「作成」をクリックして進みます。名前やリソースグループなどの詳細を設定していきます。

「ワークスペース名」と「リソースグループ」に入力する名前は任意です。リソースグループは新規作成するか、VM と同じリソースグループにしておいても問題ありません。場所は「米国西部 2」と設定しておきましょう。

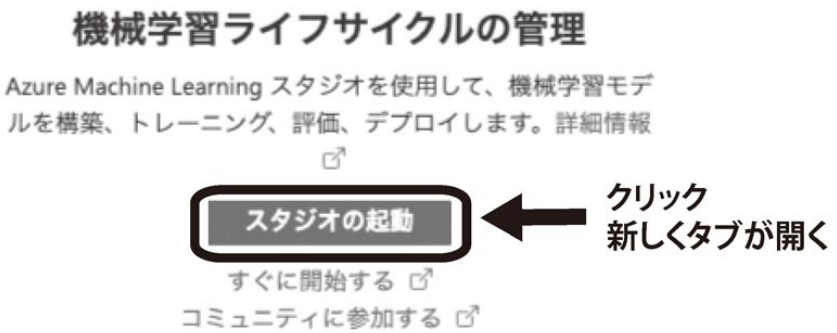


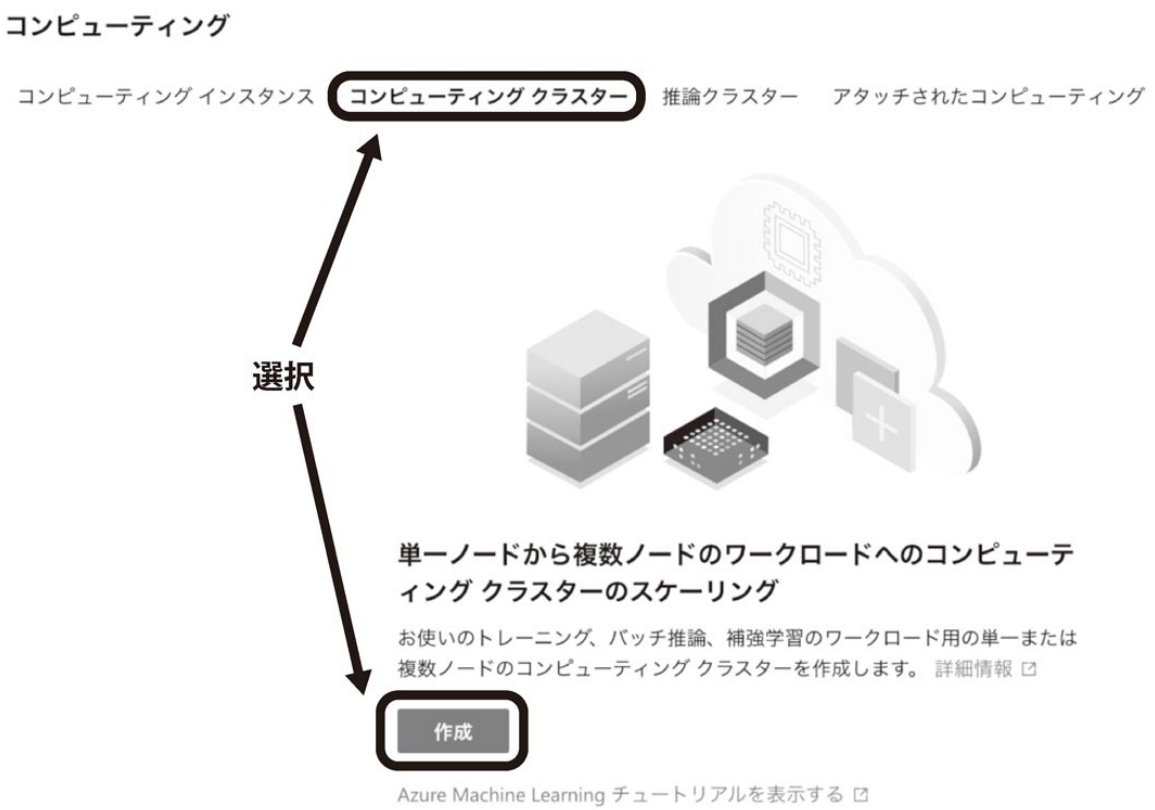
設定が完了すると、「確認および作成」から上記のようにデプロイまで進行していきます。デ

プロイには数分時間がかかるので待ちましょう。

計算機環境の構築

デプロイが完了したら、Azure ポータル上で計算を行うためのリソースを作成していきます。次の画面を参考に手順どおり進んでください。





名前（Computer name）および GPU リソースを設定します。このリソースを設定するときのポイントとして、仮想マシンのサイズ（Virtual Machine size）は、各マシンに GPU が 1 枚

でよければ NC6、GPU が 4 枚積まれているほうがよい場合は NC24 を選択します。また、VM の優先度（Virtual Machine priority）は低優先度（Low Priority）にしておくと、実際に提示されている金額の 8 割引きで利用できるので、こちらを推奨します。

ノードの最小値（Minumum number of nodes）は必ずデフォルトのとおり 0 にしておきましょう。ここに 1 や 2 などの数値を入れてしまうと、計算をしなくてもその数のぶんだけマシンが立ち上がった状態になり、常に課金されてしまいます。0 としておけば、計算がない状況では課金されないように設定されます。ノードの最大数（Maximum number of nodes）は使える金額に合わせて設定しましょう。スケールダウンのアイドル時間（Idle seconds before scale down）は短いほど無駄がないのですが、一度終了すると新しいマシンを立ち上げるまで数分かかって意外と面倒なので、デバッグ時は 600 秒（5 分）のように長めに設定しておき、実運用するときには 60 秒など短めに設定しておくと良いでしょう。

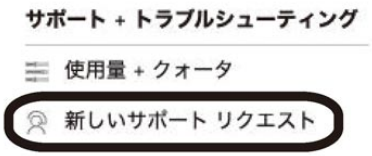
以下のようにリソースが起動していれば成功です。

これで基本設定は完了です。

7.3　低優先度クォータのリクエスト

低優先度クォータを利用する場合、複数の GPU を同時に使用するほうが実行時間が速く、課金額も低くなるため、クォータの引き上げをこのタイミングでリクエストしておきましょう。左

下の「新しいサポート リクエスト」に進んでください。



ここでは「Machine Learning Service」のクォータの制限を指定します。

「Low Priority vCPUs」を選択し、クォータ数を決定します。200 ほどあれば NC6 を 30 台程度並列で実行できるのでお勧めです。

入力事項を確認したら、リクエストを送信してください。

これで完了です。後はサポートからの返信を待ちましょう。

それではここから、Azure ポータル上ではなく、Jupyter Notebook 上でハイパーパラメータのチューニングを実行していきます。

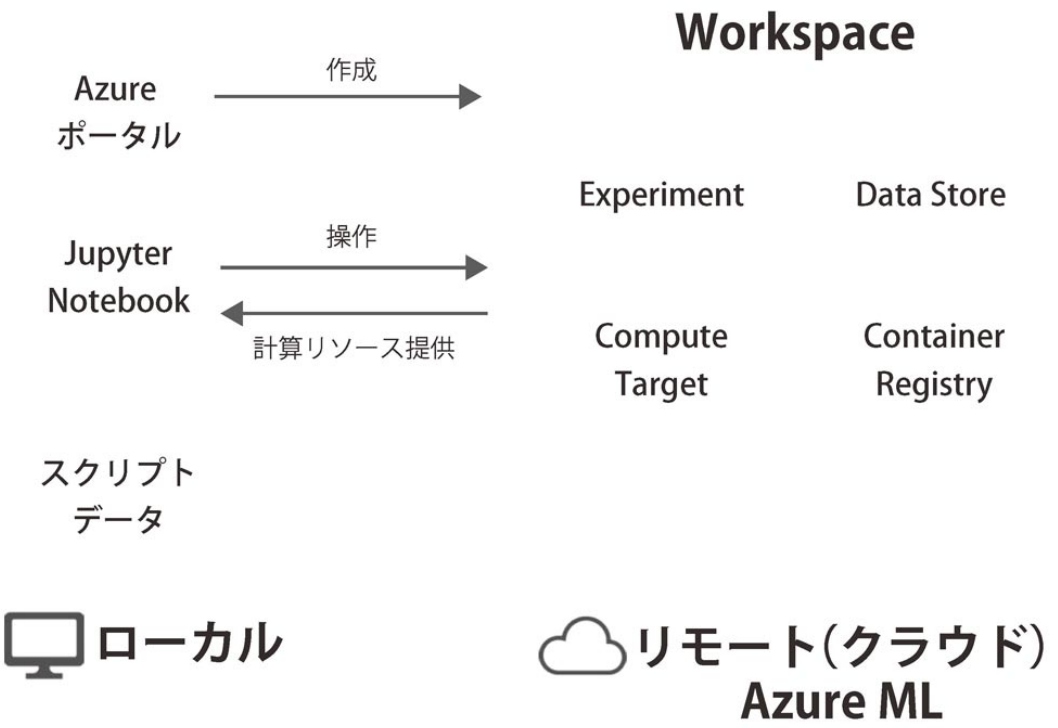
7.4 Azure ML SDK のインストール

Azure ML にアクセスするには Azure ML 用の SDK が必要となるため、次のスクリプトを実行してパッケージをインストールしましょう。SDK とは、ソフトウェア開発者にとって必要なものをまとめてくれている開発ツールキットです。

```
# 必要なSDKをインストール
!pip install azureml
!pip install azureml-core
!pip install azureml-sdk
!pip install azureml-widgets

# 必要なモジュールをインポート
import numpy as np
import os
import json
import re

import azureml
from azureml.core import Workspace, Experiment
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

# バージョンの確認
azureml.core.VERSION
```

```
'1.0.79'
```

関係図を示すと次のようになります。

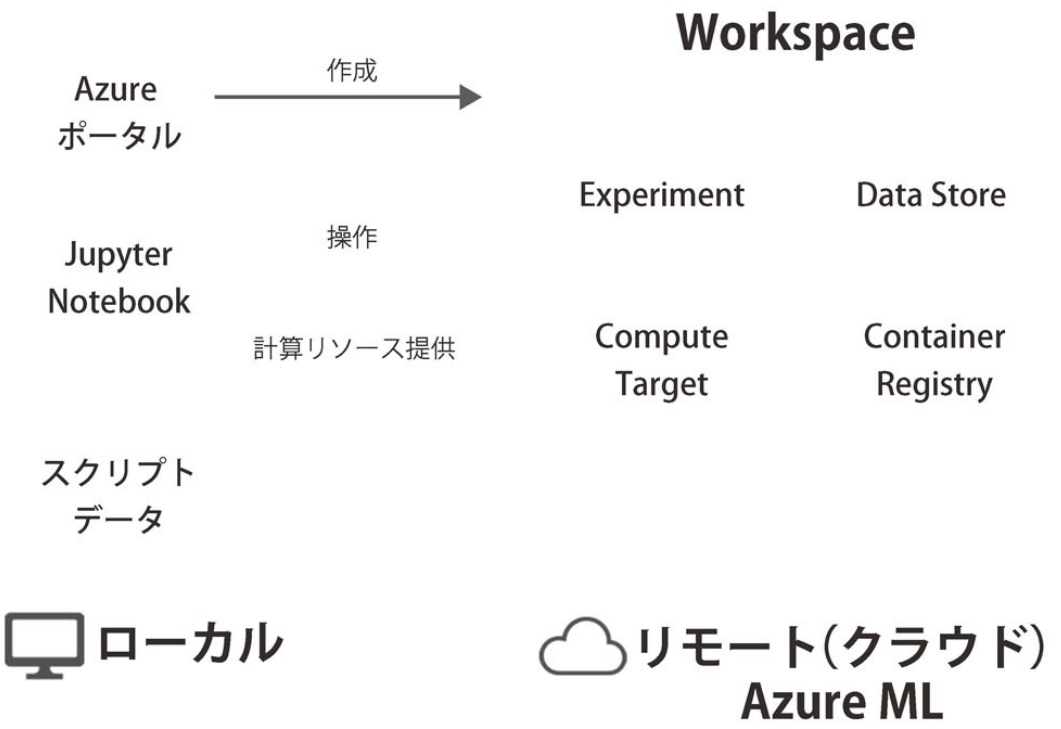


1. Azure ポータルから Azure ML を作成
2. Juputer Notebook から操作
3. Azure ML が Jupyter Notebook にリッチな計算リソースを提供

クラウド環境を使うことで、必要な分だけリッチな計算リソースの恩恵を受けられるので、とても便利です。しかも、今回使用した NC6 であれば、1 時間あたり$0.90 なので 100 円程度で利用できます。基本料金が安価となっているので、さらに GPU が多く積まれている環境も使ってみてください。

Workspace

Workspace は、Azure ML のリソースを統合したものを指します。すでにポータル上で作成済みなので、その Workspace を指定します。

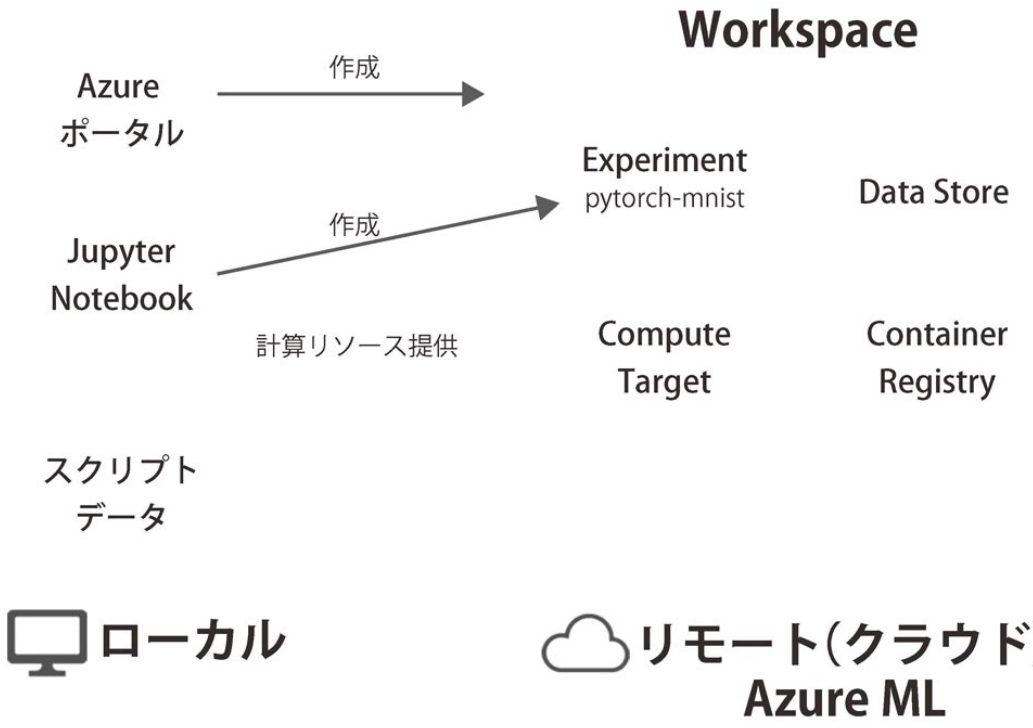



```
# 各種設定
name = '<your-compute-name>'
subscription_id = '<your-subscription-id>'
resource_group = '<your-resource-group>'

# Workspaceを取得
ws = Workspace.get(
    name = name,
    subscription_id = subscription_id,
    resource_group = resource_group
)

ws
```

```
Workspace.create(name='pytorch_topgear_parameter',
subscription_id='f5bfe65c-b3a3-4ea7-ac4c-7d4dbbbeef26',
resource_group='pytorch_topgear_parameter')
```

Experiment

実験する環境を作成します。ここから実行結果を追跡できます。



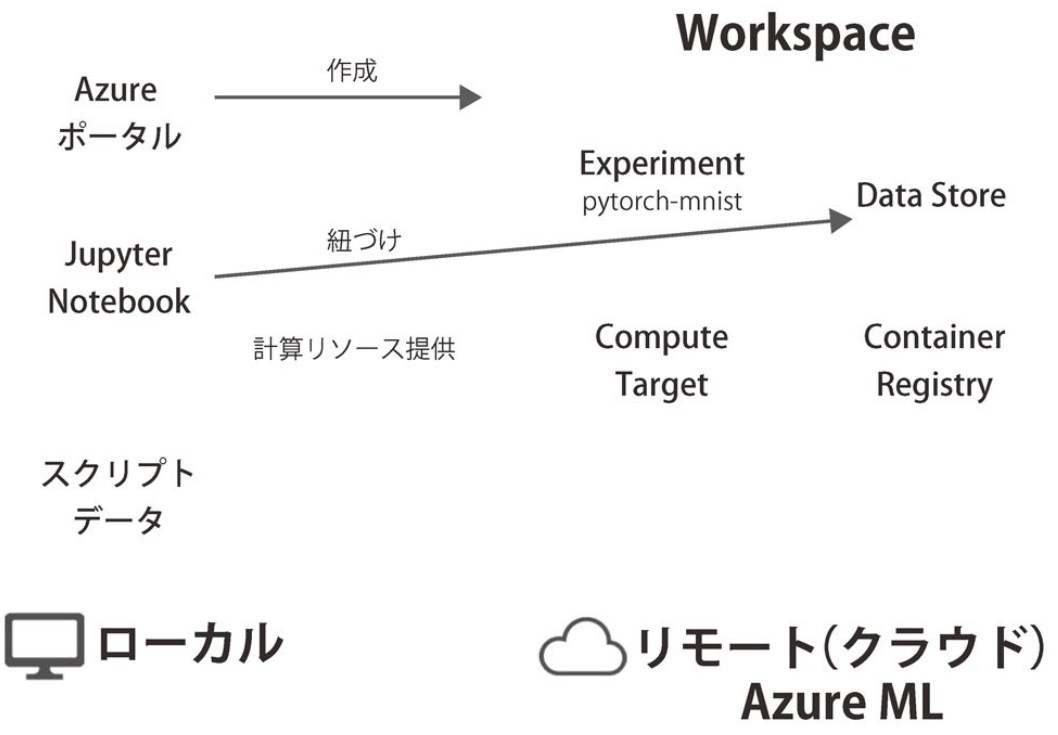

```
# 実験環境の作成
exp = Experiment(workspace=ws, name='pytorch-mnist-simple')
exp
```

Name	Workspace	Report Page	Docs Page
pytorch-mnist-simple	pytorch_topgear_parameter	Link to Azure Machine Learning studio	Link to Documentation

Data Store

実験結果やスクリプトなどのデータを格納しておく場所です。Workspace を作る時点で一緒

に作成されているため、Workspace に紐づいた Data Store を確認します。

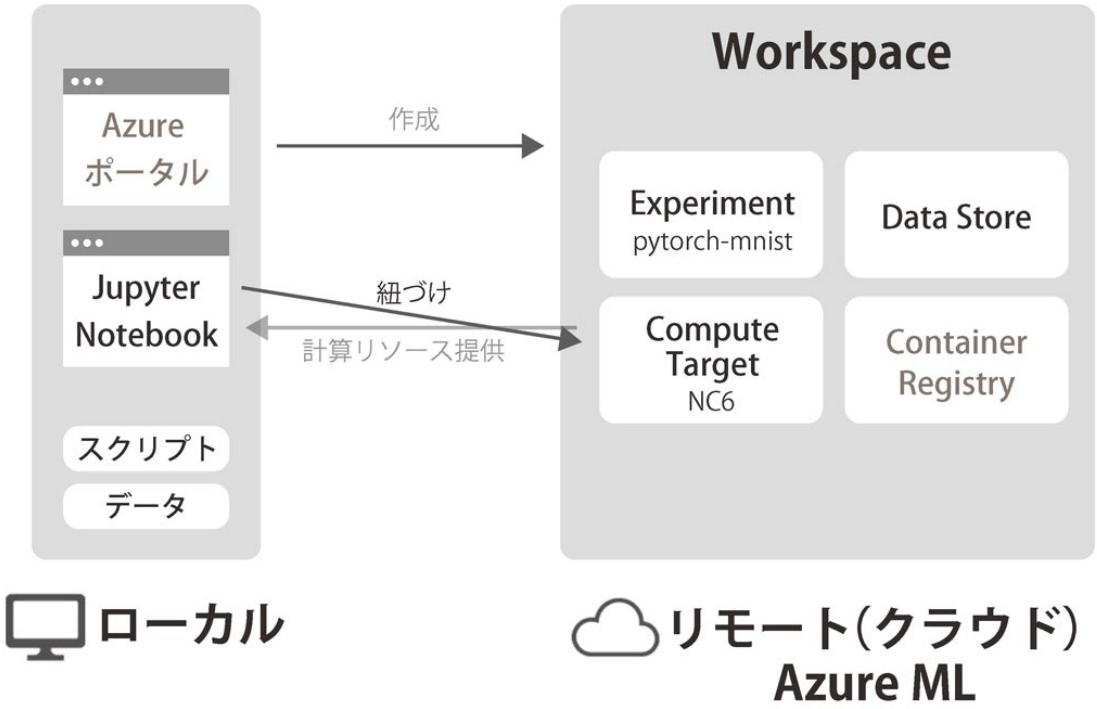



```
# 紐づけ場所の確認
ds = ws.get_default_datastore()
ds
```

```
<azureml.data.azure_storage_datastore.AzureBlobDatastore at 0x121aa6f90>
```

Compute Target

計算をマルチノードで行う計算機環境です。こちらもポータル上で作成済みなので、紐づけておきましょう。




```
# 計算機との紐づけ（NC6を使用）
compute_target = ComputeTarget(workspace=ws, name='gpu-compute')
compute_target
```

```
AmlCompute(workspace=Workspace.create(name='pytorch_topgear_parameter', subscription_id='f5bfe65c-b3a3-4ea7-ac4c-7d4dbbbeef26', resource_group='pytorch_topgear_parameter'), name=gpu-compute, id=/subscriptions/f5bfe65c-b3a3-4ea7-ac4c-7d4dbbbeef26/resourceGroups/pytorch_topgear_parameter/providers/Microsoft.MachineLearningServices/workspaces/pytorch_topgear_parameter/computes/gpu-compute, type=AmlCompute, provisioning_state=Succeeded, location=westus2, tags=None)
```

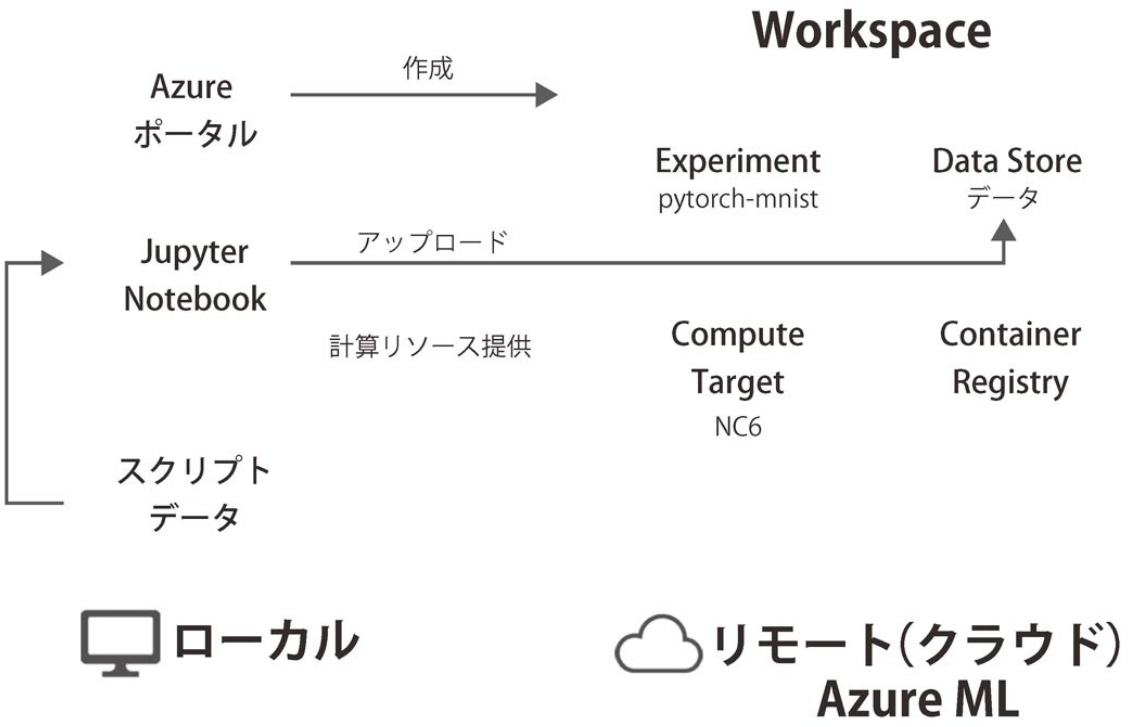
```
# 計算機リソース
comp_dict = compute_target.get_status().serialize()
vm_size = comp_dict['vmSize']
print('VM size:', vm_size)

def get_gpu_count(vm_size):
    pattern = r'\d{1,2}'
    s = re.search(pattern, vm_size)
    gpu = vm_size[s.start():s.end()]
    return int(gpu) // 6
```

```
gpu = get_gpu_count(vm_size)
print('# GPUs:', gpu)
```

```
VM size: STANDARD_NC6
# GPUs: 1
```

7.5 データセットの準備

データストア上に学習や検証に使用するデータセットをアップロードしておきましょう。



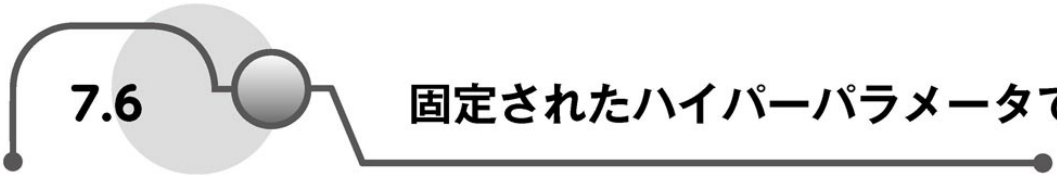

```
import torch
import torchvision
from torchvision import transforms, datasets

# Tensor型に変換
transform = transforms.Compose([
    transforms.ToTensor()
```

```
])

# MNISTを使用
train_val = torchvision.datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
test = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform=transform)

n_train = int(len(train_val) * 0.8)
n_val = len(train_val) - n_train

torch.manual_seed(0)
train, val = torch.utils.data.random_split(train_val, [n_train, n_val])
```

Data Store にデータをアップロードします。upload() メソッドにデータの PATH を指定します。

```
# Data Storeにデータをアップロード
ds.upload(src_dir='data/MNIST', target_path='MNIST', overwrite=False)
```

```
Uploading an estimated of 20 files
Uploading data/MNIST/MNIST/processed/test.pt
 …
Uploaded data/MNIST/raw/t10k-images-idx3-ubyte, 20 files out of an estimated total of 20
Uploaded 20 files

$AZUREML_DATAREFERENCE_fc4d3e2c1b6d43aabbc76b6ddfc071a4
```



7.6 固定されたハイパーパラメータでの検証

複数のハイパーパラメータをチューニングする前に、1 組のハイパーパラメータで正しく動作するか確認しましょう。

ArgumentParser

コードの流れは Ax と同じですが、ハイパーパラメータの指定方法が異なるため、この違いを確認します。Ax ではハイパーパラメータの場所を直接 parameters.get('パラメータ名', 数値) という形で辞書として指定していました。これに対し、Azure ML では Python の標準モジュールの argparse を使います。argparse は**プログラムの実行時に引数を指定する**ことがで

きるモジュールです。使い方としては、

1. パーサーを作る
2. 引数を追加する
3. 引数を解析する
4. 引数を指定してプログラムを実行する

という 4 ステップになっています。Azure ML ではこの引数の指定を script_params という変数が担っており、プログラム実行の際にはこの script_params からハイパーパラメータが渡されてモデルの学習が実行される仕組みです。

まずは、ホーム右上の「New」から「Text File」を開き、名前を argparse_sample.py として作成しましょう。中のコードは次のとおりです。

```
import argparse

# 1. パーサー（parser）を作成
parser = argparse.ArgumentParser()
# 2. parserが受け取ることの可能な引数を追加
parser.add_argument('--input', type=str)
# 3. 引数を解析
args = parser.parse_args()

# inputで受け取った情報を表示
print(args.input)
```

それでは、作成した Python のファイルを実行しましょう。Jupyter Notebook の Terminal を（ホーム右上の「New」から）開いて、以下のコードを入力しましょう。Python のファイルは「python3 ファイル名」として実行できます。argparse を使用する際は、オプションとして --input 'Hello, world!' のように情報を渡します。

```
python argparse_sample.py --input 'Hello, world!'
```

「Hello, world!」と表示されれば成功です。

最後に、次のようにプログラムを関数を使うよう書き換えました。皆さんがこれまで実行してきた import にも対応できる汎用的なプログラムファイルとなりました。Terminal から実行すると__name__は__main__として定義されるため、if 文が True となり、main 関数が実行されます。import されたときは__name__が import の名前となるため、main 関数は実行されません。

```
import argparse

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--input', type=str)
    args = parser.parse_args()
    print(args.input)

if __name__ == '__main__':
     main()
```

Azure ML で ArgumentParser を作成して実行

azureml/script/train_mnist.py を作成しましょう。今回の検証で使用する train_mnist.py は以下のとおりです。

```
import argparse
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
from torchvision import transforms, datasets
from torch.utils.data import DataLoader
import pytorch_lightning as pl
from pytorch_lightning import Trainer
from azureml.core.run import Run

# 学習結果の取得
run = Run.get_context()

print('PyTorch version: ', torch.__version__)

class Net(pl.LightningModule):

    def __init__(self, hparams, num_workers=8):
        super(Net, self).__init__()
        self.hparams = hparams
        self.num_workers = num_workers
        self.conv = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(28 * 28, self.hparams.n_hidden)
        self.fc2 = nn.Linear(self.hparams.n_hidden, 10)

    def _dataloader(self, train):
        transform = transforms.Compose([transforms.ToTensor()])
        dataset = torchvision.datasets.MNIST(root=self.hparams.data_dir, train=train, download=T⇨
```

```
rue, transform=transform)
        loader = DataLoader(dataset, self.hparams.batch_size, shuffle=True, num_workers=self.num⇨
_workers)
        return loader

    def lossfun(self, y, t):
        return F.cross_entropy(y, t)

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(), lr=self.hparams.lr, momentum=self.hparams.mome⇨
ntum)

    def forward(self, x):
        x = self.conv(x)
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 28 * 28)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    @pl.data_loader
    def train_dataloader(self):
        return self._dataloader(train=True)

    def training_step(self, batch, batch_nb):
        x, t = batch
        y = self.forward(x)
        loss = self.lossfun(y, t)
        results = {'loss': loss}
        return results

    @pl.data_loader
    def val_dataloader(self):
        return self._dataloader(train=False)

    def validation_step(self, batch, batch_nb):
        x, t = batch
        y = self.forward(x)
        loss = self.lossfun(y, t)
        y_label = torch.argmax(y, dim=1)
        acc = torch.sum(t == y_label) * 1.0 / len(t)
        results = {'val_loss': loss, 'val_acc': acc}
        return results

    def validation_end(self, outputs):
        avg_loss = torch.stack([x['val_loss'] for x in outputs]).mean()
        avg_acc = torch.stack([x['val_acc'] for x in outputs]).mean()
        results =  {'val_loss': avg_loss, 'val_acc': avg_acc}
```

```
        return results

    @pl.data_loader
    def test_dataloader(self):
        return self._dataloader(train=False)

    def test_step(self, batch, batch_nb):
        x, t = batch
        y = self.forward(x)
        loss = self.lossfun(y, t)
        y_label = torch.argmax(y, dim=1)
        acc = torch.sum(t == y_label) * 1.0 / len(t)
        results = {'test_loss': loss, 'test_acc': acc}
        return results

    def test_end(self, outputs):
        avg_loss = torch.stack([x['test_loss'] for x in outputs]).mean()
        avg_acc = torch.stack([x['test_acc'] for x in outputs]).mean()
        results = {'test_loss': avg_loss, 'test_acc': avg_acc}
        return results

def main():

    parser = argparse.ArgumentParser()
    parser.add_argument('--data-dir', type=str, dest='data_dir', default='./data')
    parser.add_argument('--batch-size', type=int, dest='batch_size', default=50)
    parser.add_argument('--epoch', type=int, dest='epoch', default=20)
    parser.add_argument('--lr', type=float, dest='lr', default=0.01)
    parser.add_argument('--n-hidden', type=int, dest='n_hidden', default=100)
    parser.add_argument('--momentum', type=float, dest='momentum', default=0.9)

    hparams = parser.parse_args()

    torch.manual_seed(0)
    net = Net(hparams)
    trainer = Trainer(max_nb_epochs=hparams.epoch)

    print('start training')
    trainer.fit(net)
    print('finish training')

    # 検証&評価
    print('Validation Score:', trainer.callback_metrics)
    trainer.test()
    print('Test Score:', trainer.callback_metrics)

    # 学習結果の追跡
    run.log('lr', hparams.lr)
```

```
    run.log('n_hidden', hparams.n_hidden)
    run.log('momentum', hparams.momentum)
    run.log('accuracy', trainer.callback_metrics)

if __name__ == '__main__':
    main()
```

スクリプト作成のポイント

まったくのゼロからスクリプトを書き始めると、エラーが発生したときの対応に追われてしまうことが多いため、あまりお勧めできません。スクリプトを作成する際には、以下の手順に従って作成するのがよいでしょう。

1. Jupyter Notebook 上で関数を用いずに訓練を実行（引数をまとめておくとよい）
2. 1. で作成したプログラムを関数化
3. 関数化したプログラムの引数を argparse を用いて指定する

このようにまず動くことを確認した後に、関数化し、引数をまとめるという作業に順に移っていきましょう。

訓練結果の追跡

Azure ML の環境で訓練を実行し、結果を追跡するには、いくつかのコードを追記します。学習経過の追跡は以下のコードによって実行されます。新たなハイパーパラメータを追跡する際には run.log() を追記して、追跡対象を指定する必要があります。

```
from azureml.core.run import Run
run = Run.get_context()

run.log('lr', lr)
run.log('n_hidden', n_hidden)
run.log('momentum', momentum)
run.log('accuracy', trainer.callback_metrics)
```

ハイパーパラメータの調整を行うときに、これらのログは重要な役割を果たすので、覚えておいてください。

```
from azureml.widgets import RunDetails
from azureml.train.dnn import PyTorch
```

train_mnist.py に渡す引数を定義します。「'parser.add_argument()で指定した名前': 数値」という形式で、作成した引数に値を渡せます。

```
# 引数の設定
script_params = {
    # Data Storeでのdataの場所
    '--data-dir': ds.path('data').as_mount(),
    '--n-hidden': 100,
    '--batch-size': 50,
    '--epoch': 20,
    '--lr': 0.001,
    '--momentum': 0.09
}
```

学習の設定をしていきます。保存済みの実行ファイルの場所などを指定しましょう。

- source_directory：学習の実行ファイルの場所
- script_params：定義したハイパーパラメータ
- compute_target：計算機環境
- entry_script：学習の実行ファイル
- use_gpu：GPU の使用有無
- pip_packages：学習に必要なパッケージ
- max_run_duration_seconds：実行の最大許容時間

```
# 使用できるPyTorchのバージョン
PyTorch.get_supported_versions()
```

```
['1.0', '1.1', '1.2', '1.3']
```

```
# 学習の設定
estimator = PyTorch(
    source_directory = 'azureml/script',
    script_params = script_params,
    compute_target = compute_target,
    entry_script = 'train_mnist.py',
    framework_version = '1.3',
    use_gpu = True,
    pip_packages = ['pytorch-lightning'],
    max_run_duration_seconds = 1800
)
```

設定できたら submit で estimator の内容を参照して実行します。準備の段階では、estima

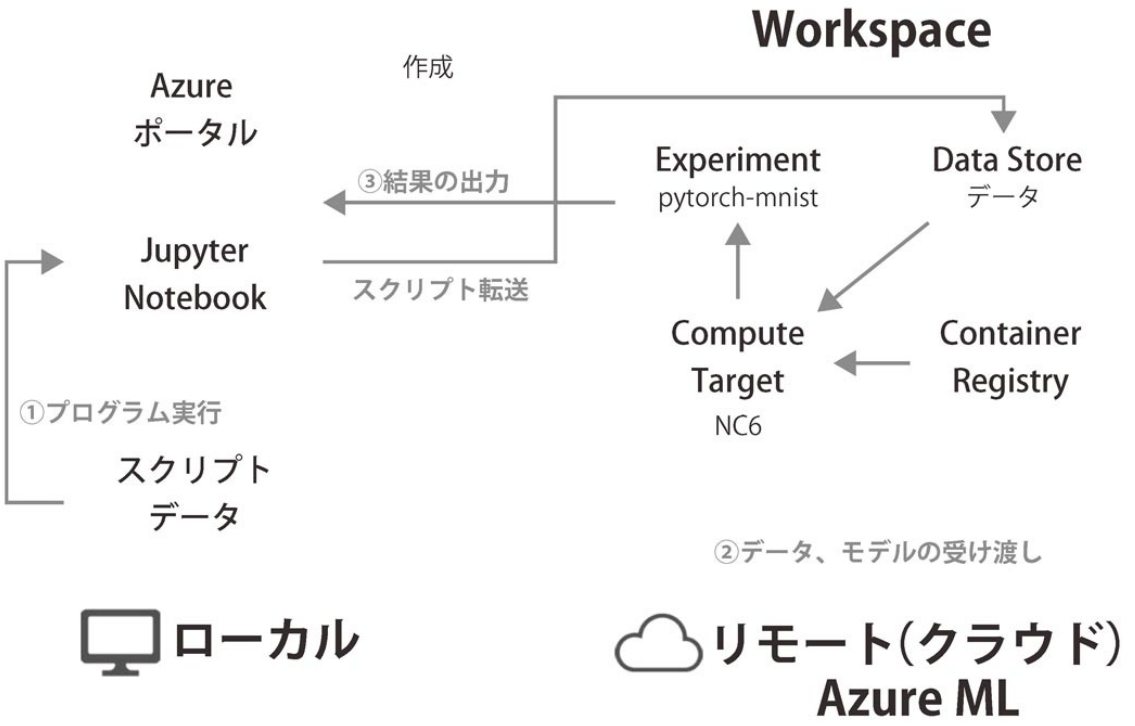
tor で指定された Python 環境に合わせて Docker イメージが作成され、それが Workspace の Azure Container Registry にアップロードされます。このステップは Python 環境ごとに一度だけ発生します。そして、script フォルダ内のすべてのファイルが Data Store に入り、実行されます。

```
# 実行を送信
run = exp.submit(estimator)
print(run)
```

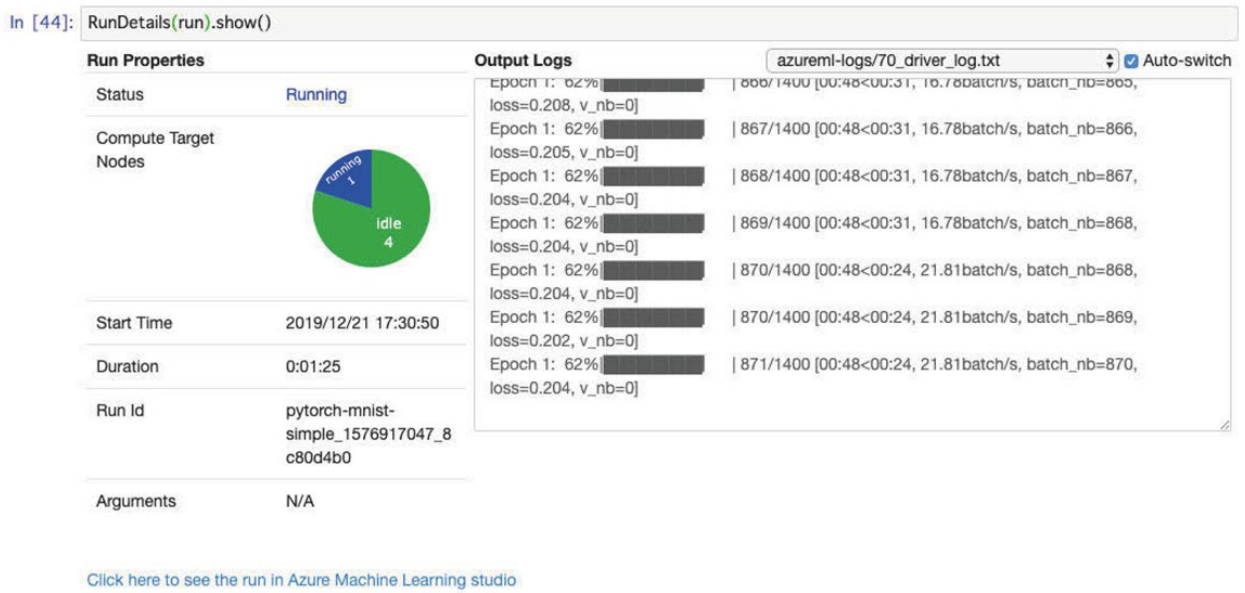
```
Run(Experiment: pytorch-mnist-simple,
Id: pytorch-mnist-simple_1576951674_034e44d1,
Type: azureml.scriptrun,
Status: Queued)
```

Jupyter Notebook 上で学習過程の進捗を表示するには、RunDetails().show() を実行します。一度 Jupyter Notebook を閉じてしまうと表示されませんが、以下のように進捗をリアルタイムに見ることができます。

プログラムを実行し、結果の詳細を表示してみましょう。



```
RunDetails(run).show()
```



今回はチューニングせず、決まった値で学習を進めましたが、Azure ML 上で学習できることは確認できました。それでは、ハイパーパラメータのチューニング方法として有名なランダムサーチとベイズ最適化を使って、ハイパーパラメータをチューニングしていきましょう。

7.7 Azure ML によるランダムサーチ

Azure ML で、ランダムサーチに基づいてハイパーパラメータを調整していきます。ランダムサーチ（Random Search）は、指定された探索範囲のハイパーパラメータをランダムに選び、効果を測定します。

- **良い点**：広い探索範囲において、グリッドサーチよりも早く最適解を見つけやすい
- **悪い点**：あくまでもランダムにハイパーパラメータを選択するため、その最適解が本当に最適なのかの保証がない

経験的にハイパーパラメータの探索範囲が広い、最適化したいハイパーパラメータの数が多いなど、計算リソースを軽減したい場合に用いられることが多いです。

ハイパーパラメータのチューニング

基本的な流れは先ほどまでと同じで、Workspace や Experiment との紐づけは終わっているので、早速ハイパーパラメータの調整に入ります。以下のコードでは早期終了も追加しています。

```
# 必要なモジュールの読み込み
from azureml.train.hyperdrive import RandomParameterSampling, BanditPolicy, HyperDriveConfig, Pr⇨
imaryMetricGoal
from azureml.train.hyperdrive import choice, uniform
```

ここでは、調べるハイパーパラメータの候補を RandomParameterSampling を使って定義します。「ハイパーパラメータ名: 数値」という形式で定義していきましょう。

各ハイパーパラメータに対して定義された探索空間を与えることで、構築したモデルのハイパーパラメータを Azure ML 側で自動的に調整します。ハイパーパラメータの種類は離散値でも連続値でもよく、いくつかの定義方法がありますが、よく使うものを紹介します。

- **離散値**：choice
- **連続値**：uniform

このほかにも探索範囲の指定方法はありますが、まずはこの 2 つを覚えておきましょう。ほかの方法も気になる方は、公式ドキュメント（https://docs.microsoft.com/ja-jp/azure/machine-learning/service/how-to-tune-hyperparameters）を参照してください。

```
# ランダムサーチで調べるハイパーパラメータの定義
parameters = RandomParameterSampling(
    {
        '--n-hidden': choice(50, 100, 200, 300, 500),
        '--lr': uniform(1e-6, 1e-1),
        '--momentum': uniform(1e-6, 1e-1)
    }
)
```

調整を行わない引数に関しては、script_params で準備します。

```
script_params = {
    '--data-dir': ds.path('data').as_mount(),
    '--epoch': 20,
    '--batch-size': 256
}

# 学習の設定
estimator = PyTorch(
    source_directory = 'azureml/script',
```

```
    script_params = script_params,
    compute_target = compute_target,
    framework_version='1.3',
    pip_packages = ['pytorch-lightning'],
    entry_script = 'train_mnist.py',
    use_gpu=True
)
```

BanditPolicy を設定すると、性能がある一定期間変わらない場合に学習を終了させることができます。これは早期終了と同じ効果です。性能の向上が見込めないのに学習を続けるような、不必要にリソースを使うことを理論的には防いでくれます。

設定に必要な引数の概要は以下のとおりです。

- slack_factor：早期終了を行う閾値（指定したメトリックの最高のものとの誤差）
- evaluation_interval：早期終了を確認する周期（エポック数）
- delay_evaluation：早期終了の確認を始めるタイミング（0 エポック目から何エポック遅らせるか）

```
policy = BanditPolicy(evaluation_interval=1, slack_factor=0.1)
```

本章では、Azure ML を使った効率的なハイパーパラメータのチューニング方法を紹介してきました。計算リソースを必要な分だけ借りるという経済的にも優しい選択肢なので、ぜひ活用してください。

第8章 自然言語処理

自然言語とは、私たちが日常的に使っている「言葉」のことです。文字として書かれた言葉をテキストデータ、声として発された言葉を音声データと呼びます。

そして自然言語処理とは、自然言語をコンピュータに理解させるための一連の処理を指し、機械翻訳・チャットボット・検索エンジンなど、言葉を対象とする技術全般に活用されています。

本章では自然言語の特徴量の 1 つである BoW の作成をゴールに、形態素解析と呼ばれる、自然言語を「意味を持つ最小の単位（単語）」に分割する手法を用いて、自然言語の前処理を理解していきます。

私	は	キカガク	です	。
名詞	助詞	名詞	助動詞	句点

BoW とは Bag of Words の略であり、文章中に出現する単語を数えて、その数を特徴量とする手法です。**one-hot 表現**と呼ばれる手法が用いられ、文章中における語順は考慮しないので、非常にシンプルな表現です。

BoW	1	0	1	1	0	1	0	1
私	1	0	0	0	0	0	0	0
は	0	0	0	1	0	0	0	0
キカガク	0	0	1	0	0	0	0	0
です	0	0	0	0	0	1	0	0
。	0	0	0	0	0	0	0	1

単語の one-hot 表現

8.1 自然言語を扱う難しさ

自然言語をコンピュータで扱う際の難しさとして、以下のような事項が挙げられます。

- 文章中の単語をどのように切り分けるのか、分かち書きの基準が難しい
- 入力値として、どのような特徴量が適切なのか判断が難しい
- 長さが文章によって異なる可変長のデータの扱いが難しい

分かち書きの基準が難しい

日本語は数ある言語の中でも自然言語処理が難しい言語の 1 つであると言われています。

というのも、英語の場合には「I am Kikagaku.」といったように単語別にスペースが存在しますが、日本語では「私はキカガクです。」と単語の切れ目が不明確です。この問題を解決するのが本章で扱う **MeCab** と呼ばれる形態素解析ライブラリです。

適切な特徴量の判断が難しい

自然言語はここまで扱ってきた構造化データの例（CSV）や画像データと異なり、何をもって数値化すればよいのか基準を定めることが困難です。構造化データであれば最初から売上などの数値として用意されており、画像データの場合は各ピクセルの明るさ（輝度）を数値化できます。

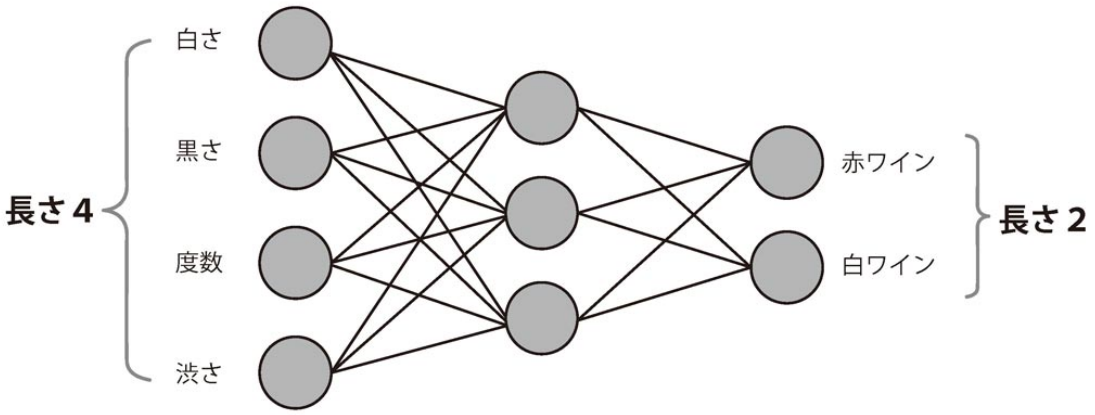
では、自然言語はいかにして数値化して特徴量を抽出すればよいのでしょうか？ この問題を解決する手法として **Embedding** というものがあり、「8.7 文章分類」で扱います。

可変長のデータの扱いが難しい

自然言語は、文章によってさまざまな長さ（単語数）を持ちます。

- 「はじめまして。」
- 「私はキカガクです。」
- 「AI に関する教育を行う会社です。」

ここまで扱ってきた機械学習の手法において、データの長さは一定でした。ワインの分類の例を次に示します。長さがサンプルによって変化しないデータを**固定長のデータ**と呼びます。



この問題を解決する手法として、**RNN** と呼ばれるネットワークが考案されました。これも「8.7 文章分類」で扱います。

8.2 自然言語処理を理解するロードマップ

ここまで整理した事項を理解して自然言語処理を習得するには、大きく分けて 3 つのスキルが必要です。

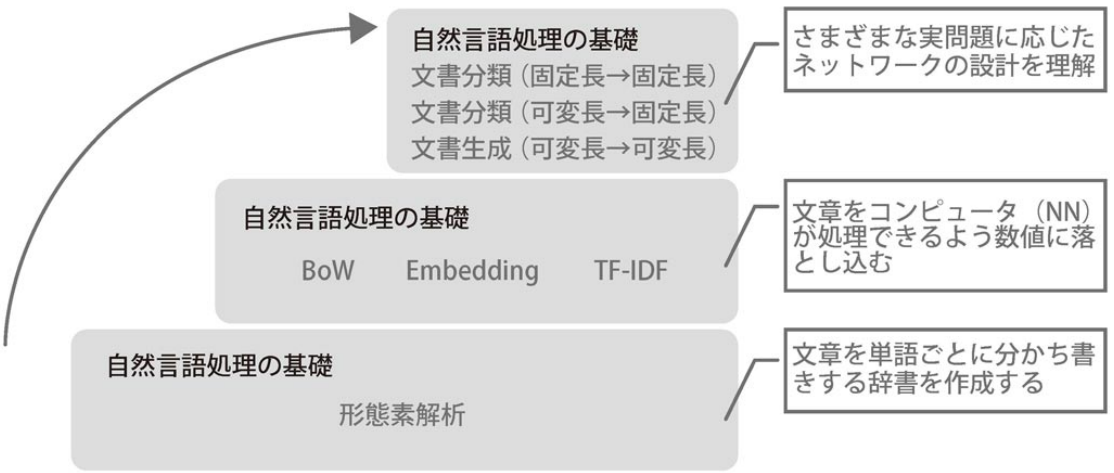
1. 文章を単語ごとに適切に分かち書きしたり、課題設定に応じて適切な語句を抽出する形態素解析のスキル
2. 自然言語をコンピュータが処理できる形式である数値に落とし込む特徴量抽出のスキル
3. さまざまな実問題に応じて適切なネットワークを設計するスキル

1. と 2. は前処理に関するスキルで、3. はネットワークの実装に関するスキルです。キーワードとともにロードマップを体系的に整理すると、以下のようになります。

1. 自然言語の扱い方
   - 形態素解析
2. 自然言語の特徴量
   - BoW
   - TF-IDF
   - Embedding
3. 実際問題への適用するネットワーク設計
   - 文書分類（入力：固定長、出力：固定長）

- 文章分類（入力：可変長、出力：固定長）
- 文章生成（入力：可変長、出力：可変長）

なお、本書では、句点で終わる 1 文を**文章**、文章が連なってできた 1 つの記事を**文書**と表すことにします。



8.3 形態素解析ライブラリ MeCab

自然言語を扱う際に「私はキカガクです。」といった文章のまま扱うのではなく、単語ごとに切り分けることで、特徴を捉えやすくなります。繰り返しになりますが、この処理を**形態素解析**と呼びます。

```
私 / は / キカガク / です / 。
```

形態素解析は研究によって積み重ねられたノウハウに依存する要素が多く、個人で 1 から実装することは非常に難しいですが、**MeCab** と呼ばれるライブラリを使用することで簡単に実装できます。

```
import MeCab
```

警告も表示に設定しておきましょう。

```
import warnings
warnings.filterwarnings('ignore')
```

MeCab を使用する際には出力形式をオプションとして指定します。主に使用するオプションは-Ochasen と-Owakati の 2 種類です。

違いは以下のとおりです。

- Owakati：文章を単語別に分かち書きするのみ
- Ochasen：分かち書きとあわせて、品詞などの形態素解析に必要な情報が得られる

-Ochasen は単語の品詞など詳細な情報が必要な際に使用します。シンプルに単語別に切り分ければよいだけなら、-Owakati を使用すればコード量を減らせるので、覚えておくとよいでしょう。

```
# -Owakatiを使用
mecab = MeCab.Tagger('-Owakati')
```

parse() メソッドを用いると、簡単に形態素解析後の結果を得られます。

```
res = mecab.parse('こんにちは、私の名前はキカガクです。')
res
```

```
'こんにちは 、 私 の 名前 は キカガク です 。 \n'
```

このように、-Owakati を使用すると単語ごとにスペースで区切られた状態が出力されます。

ここで、-Owakati を使用する際に留意する点が 2 つあります。それぞれ実装していきましょう。

1. 分かち書き結果はスペースで区切られているので、split() でリスト型に変換する
2. リストの要素番号を指定してエスケープシーケンス\n を取り除く

なお、split() は形態素解析で頻繁に使用するメソッドです。文字列を指定すると、その文字列を境目にして文章をリスト型に切り分けます。

```
# split()でリスト型に変換
res = res.split(' ')
res
```

```
['こんにちは', '、', '私', 'の', '名前', 'は', 'キカガク', 'です', '。', '\n']
```

```
# エスケープシーケンスを取り除く
res = res[:-1]
res
```

```
['こんにちは', '、', '私', 'の', '名前', 'は', 'キカガク', 'です', '。']
```

これで-Owakati の使用方法を理解できました。「8.7 文章分類」で使用するので、その際に改めて解説します。

続いて-Ochasen を使ってみましょう。ここでは-Ochasen を用いた形態素解析から、自然言語の特徴量である BoW の作成まで進めていきます。

```
# -Ochasenを使用
mecab = MeCab.Tagger('-Ochasen')

res = mecab.parse('こんにちは、私はキカガクです。')
print(res)
```

```
こんにちは      コンニチハ      こんにちは      感動詞
、      、      、      記号-読点
私      ワタシ  私      名詞-代名詞-一般
は      ハ      は      助詞-係助詞
キカガク        キカガク        キカガク        名詞-一般
です    デス    です    助動詞  特殊・デス      基本形
。      。      。      記号-句点
EOS
```

print() 関数を使用するときれいに表示されましたが、res をそのまま表示するとエスケープシーケンスを含んでおり、特徴量を定義する上で邪魔になってしまいます。

```
res
```

```
'こんにちは\tコンニチハ\tこんにちは\t感動詞\t\t\n、\t、\t、\t記号-読点\t\t\n私\tワタシ\t私\t名詞-代名詞-一般\t\t\nは\tハ\tは\t助詞-係助詞\t\t\nキカガク\tキカガク\tキカガク\t名詞-一般\t\t\nです\tデス\tです\t助動詞\t特殊・デス\t基本形\n。\t。\t。\t記号-句点\t\t\nEOS\n'
```

先ほどと同様に split() を使い、まずは改行のエスケープシーケンスである\n から分割しましょう。

```
# エスケープシーケンスを取り除く
res = res.split('\n')
res
```

```
['こんにちは\tコンニチハ\tこんにちは\t感動詞\t\t',
 '、\t、\t、\t記号-読点\t\t',
 '私\tワタシ\t私\t名詞-代名詞-一般\t\t',
 'は\tハ\tは\t助詞-係助詞\t\t',
 'キカガク\tキカガク\tキカガク\t名詞-一般\t\t',
 'です\tデス\tです\t助動詞\t特殊・デス\t基本形',
 '。\t。\t。\t記号-句点\t\t',
 'EOS',
 '']
```

切り分けられた結果は、リスト型に格納されています。

```
# 最初の要素を取り出す
res[0]
```

```
'こんにちは\tコンニチハ\tこんにちは\t感動詞\t\t'
```

続いて、リスト型に格納された各要素を\t で分割していきます。

```
# \tで分割
res[0].split('\t')
```

```
['こんにちは', 'コンニチハ', 'こんにちは', '感動詞', '', '']
```

最後に for 文でリスト型の全要素に対して\t による分割を実行します。

1 つ注意点として、res の後半 2 行には EOS (End of String) と空白 ('') が入っています。文章とは関係がないので先に取り除いておきましょう。

```
# 不要な後半2行を取り除く
res = res[:-2]
res
```

```
['こんにちは\tコンニチハ\tこんにちは\t感動詞\t\t',
 '、\t、\t、\t記号-読点\t\t',
 '私\tワタシ\t私\t名詞-代名詞-一般\t\t',
 'は\tハ\tは\t助詞-係助詞\t\t',
 'キカガク\tキカガク\tキカガク\t名詞-一般\t\t',
 'です\tデス\tです\t助動詞\t特殊・デス\t基本形',
 '。\t。\t。\t記号-句点\t\t']
```

```
# \tによる分割
res_split = []
for _res in res:
    res_split.append(_res.split('\t'))

res_split
```

```
[['こんにちは', 'コンニチハ', 'こんにちは', '感動詞', '', ''],
 ['、', '、', '、', '記号-読点', '', ''],
 ['私', 'ワタシ', '私', '名詞-代名詞-一般', '', ''],
 ['は', 'ハ', 'は', '助詞-係助詞', '', ''],
 ['キカガク', 'キカガク', 'キカガク', '名詞-一般', '', ''],
 ['です', 'デス', 'です', '助動詞', '特殊・デス', '基本形'],
 ['。', '。', '。', '記号-句点', '', '']]
```

このように、-Ochasen を使用することで、**文章の分かち書き＋品詞情報の取得**を簡単に実現できました。

8.4　名詞抽出

この後の文書分類では、名詞による影響が大きいため、名詞のみを抽出してリストに格納できるスキルが必要です。次の文章の名詞のみを抽出してリストに格納してみましょう。

- キカガクでは、ディープラーニングを含んだ機械学習や人工知能の教育を行っています。

まずはパースできるか試しましょう。

```
text = 'キカガクでは、ディープラーニングを含んだ機械学習や人工知能の教育を行っています。'

res = mecab.parse(text)
print(res)
```

```
キカガク        キカガク        キカガク        名詞-一般
で      デ      で      助詞-格助詞-一般
は      ハ      は      助詞-係助詞
、      、      、      記号-読点
ディープラーニング      ディープラーニング      ディープラーニング      名詞-一般
を      ヲ      を      助詞-格助詞-一般
含ん    フクン  含む    動詞-自立      五段・マ行      連用タ接続
だ      ダ      だ      助動詞  特殊・タ        基本形
機械    キカイ  機械    名詞-一般
学習    ガクシュウ      学習    名詞-サ変接続
や      ヤ      や      助詞-並立助詞
人工    ジンコウ        人工    名詞-一般
知能    チノウ  知能    名詞-一般
の      ノ      の      助詞-連体化
教育    キョウイク      教育    名詞-サ変接続
を      ヲ      を      助詞-格助詞-一般
行っ    オコナッ        行う    動詞-自立      五段・ワ行促音便        連用タ接続
て      テ      て      助詞-接続助詞
い      イ      いる    動詞-非自立    一段    連用形
ます    マス    ます    助動詞  特殊・マス      基本形
。      。      。      記号-句点
EOS
```

先ほど分かち書きした res を使い、それぞれを改行\n で分けましょう。特定の文字で分けるときは split() を使うのでした。

```
# \nで分割
res = res.split('\n')
res
```

```
['キカガク\tキカガク\tキカガク\t名詞-一般\t\t',
 'で\tデ\tで\t助詞-格助詞-一般\t\t',
 'は\tハ\tは\t助詞-係助詞\t\t',
 '、\t、\t、\t記号-読点\t\t',
 'ディープラーニング\tディープラーニング\tディープラーニング\t名詞-一般\t\t',
```

```
 'を\tヲ\tを\t助詞-格助詞-一般\t\t',
 '含ん\tフクン\t含む\t動詞-自立\t五段・マ行\t連用タ接続',
 'だ\tダ\tだ\t助動詞\t特殊・タ\t基本形',
 '機械\tキカイ\t機械\t名詞-一般\t\t',
 '学習\tガクシュウ\t学習\t名詞-サ変接続\t\t',
 'や\tヤ\tや\t助詞-並立助詞\t\t',
 '人工\tジンコウ\t人工\t名詞-一般\t\t',
 '知能\tチノウ\t知能\t名詞-一般\t\t',
 'の\tノ\tの\t助詞-連体化\t\t',
 '教育\tキョウイク\t教育\t名詞-サ変接続\t\t',
 'を\tヲ\tを\t助詞-格助詞-一般\t\t',
 '行っ\tオコナッ\t行う\t動詞-自立\t五段・ワ行促音便\t連用タ接続',
 'て\tテ\tて\t助詞-接続助詞\t\t',
 'い\tイ\tいる\t動詞-非自立\t一段\t連用形',
 'ます\tマス\tます\t助動詞\t特殊・マス\t基本形',
 '。\t。\t。\t記号-句点\t\t',
 'EOS',
 '']
```

EOS（End Of String）と空白（’’）が入っており、文章とは関係がないので先に取り除いておきましょう。

```
res = res[:-2]
res
```

```
['キカガク\tキカガク\tキカガク\t名詞-一般\t\t',
 …
 '。\t。\t。\t記号-句点\t\t']
```

次に、最初の要素に対して、タブ\t で分けます。

```
# \tで分割
res = res[0].split('\t')
res
```

```
['キカガク', 'キカガク', 'キカガク', '名詞-一般', '', '']
```

この結果から名詞であるかどうかを判定するには、左から 4 番目（要素番号 3）の要素にアクセスすればよいので、次のようになります。

```
# 左から4番目の要素
res[3]
```

```
'名詞-一般'
```

ここで、in を用いて文字列内に特定の単語が含まれているか否かを判定する方法を紹介します。

```
# 特定の単語が含まれているか否か判定
'キカガク' in '私はキカガクです'
```

```
True
```

この一連の手順を for 文と if 文を使って文章中の全単語に適用してみます。

```
# 形態素解析を実施
res = mecab.parse(text)

# 名詞（noun）のみを抽出
nouns = []
words = res.split('\n')[:-2]
for word in words:
    part = word.split('\t')
    if '名詞' in part[3]:
        nouns.append(part[0])

# 名詞抽出
nouns
```

```
['キカガク', 'ディープラーニング', '機械', '学習', '人工', '知能', '教育']
```

上記の例を参考に、以下の例文から名詞のみを抽出してみましょう。

- キカガクでは、ディープラーニングを含んだ機械学習や人工知能のキカガク流教育を行っています。
- キカガクの代表の吉崎は大学院では機械学習・ロボットのシステム制御、画像処理の研究に携わっていました。

- 機械学習は微分や線形代数を始めとした数学が不可欠で、数学が機械学習の知識を支える土台となります。

今後も名詞を抽出する際に便利なので、関数化しておくことをお勧めします。

```
# 関数化
def get_nouns(text):
    nouns = []
    res = mecab.parse(text)
    words = res.split('\n')[:-2]
    for word in words:
        part = word.split('\t')
        if '名詞' in part[3]:
            nouns.append(part[0])
    return nouns

# 例文
text1 = 'キカガクでは、ディープラーニングを含んだ機械学習や人工知能のキカガク流教育を行っています。'
text2 = 'キカガクの代表の吉崎は大学院では機械学習・ロボットのシステム制御、画像処理の研究に携わっていました。'
text3 = '機械学習は微分や線形代数を始めとした数学が不可欠で、数学が機械学習の知識を支える土台となります。'

nouns1 = get_nouns(text1)
nouns1
```

```
['キカガク', 'ディープラーニング', '機械', '学習', '人工', '知能', 'キカガク', '流', '教育']
```

```
nouns2 = get_nouns(text2)
nouns2
```

```
['キカガク', '代表', '吉崎', '大学院', '機械', '学習', 'ロボット', 'システム', '制御', '画像', '処理', '研究']
```

```
nouns3 = get_nouns(text3)
nouns3
```

```
['機械', '学習', '微分', '線形', '代数', '始め', '数学', '不可欠', '数学', '機械', '学習', '知識'⇨
, '土台']
```

8.5 特徴量への変換

自然言語における特徴量とは、何が当てはまるでしょうか。機械学習は入力値と目標値に分ける必要があり、そして入力値も目標値も長さはサンプルごとに一定でなければなりません。しかし、自然言語の文章というものは単語の数が異なり、これを単純に特徴量に採用することはできません。また、そもそも単語をどのように数値化するのでしょうか。

BoW

現時点でも、この自然言語処理の特徴量に関しては、議論がまだ活発に行われている最中ですが、最も簡単な特徴量として **BoW**（Bag of Words）があります。BoW は、単語の出現によって単語を数値に変換する方法です。

単純な例を挙げると、

- **単語の出現あり**：1
- **単語の出現なし**：0

に変換します。

例題

ここで例題からイメージをつかみましょう。

問題：以下の 3 つの文章を BoW に変換してください。

1. 私は電車が好きです。
2. 電車より車をよく使います。
3. 好きな果物はりんごです。

この 3 つの文に出現する単語をすべて羅列します。

```
[ 私　は　電車　が　好き　です　より　車　を　よく　使い　ます　な　果物　りんご ]
```

各文章に対し、羅列した単語を出現数に変換します。

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

BoW 用に辞書を作る

自然言語処理の BoW への変換は、gensim と呼ばれるライブラリで簡単に実装できます。また、PyTorch には、torchtext と呼ばれる自然言語処理に関する前処理を実装できるライブラリがあります。BoW を作成する程度であれば gensim で十分なのですが、それ以外は torchtext を使用するとよいでしょう。どちらも有名で頻繁に使用されるライブラリなので、両方を使えるようになっておくことをお勧めします。torchtext は次節以降で紹介していきます。

```
from gensim import corpora, matutils
```

今回抽出した名詞を 1 つのリストにまとめておきましょう。

```
# 名詞リスト
word_collect = [nouns1, nouns2, nouns3]
word_collect
```

```
[['キカガク', 'ディープラーニング', '機械', '学習', '人工', '知能', 'キカガク', '流', '教育'],
 ['キカガク',
  '代表',
  '吉崎',
  '大学院',
  '機械',
  '学習',
  'ロボット',
  'システム',
  '制御',
  '画像',
  '処理',
  '研究'],
 ['機械',
  '学習',
  '微分',
  '線形',
  '代数',
```

```
  '始め',
  '数学',
  '不可欠',
  '数学',
  '機械',
  '学習',
  '知識',
  '土台']]
```

まず BoW では、全体としてどのような名詞が使用されているか把握しなければならないため、辞書の作成を行います。この辞書内に含まれる名詞の数が BoW の特徴量の数になります。

```
# 辞書の作成
dictionary = corpora.Dictionary(word_collect)
```

作成後に dictionary を見ても、中身は把握できません。len() メソッドで辞書に登録された単語数はわかります。

```
dictionary
```

```
<gensim.corpora.dictionary.Dictionary at 0x121fcaf10>
```

```
# 辞書の単語数
len(dictionary)
```

```
25
```

辞書型は、items() メソッドを使って中身を確認できます。登録された単語を確認しておきましょう。

```
# .items()で単語を確認
for word in dictionary.items():
    print(word)
```

```
(0, 'キカガク')
(1, 'ディープラーニング')
(2, '人工')
(3, '学習')
```

```
(4, '教育')
(5, '機械')
(6, '流')
(7, '知能')
(8, 'システム')
(9, 'ロボット')
(10, '代表')
(11, '処理')
(12, '制御')
(13, '吉崎')
(14, '大学院')
(15, '画像')
(16, '研究')
(17, '不可欠')
(18, '代数')
(19, '土台')
(20, '始め')
(21, '微分')
(22, '数学')
(23, '知識')
(24, '線形')
```

重複がないように単語を選択すると、25 単語あることがわかりました。

BoW に変換する

それでは、こちらも gensim の機能を使って、文書ごとに単語が格納されている words を BoW に変換していきましょう。doc2bow() は名前のとおりドキュメントを BoW 形式に変換してくれるメソッドで、gensim で処理する際によく使われるメソッドです。

```
# doc2bowで変換
bow1_id = dictionary.doc2bow(nouns1)
bow1_id
```

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

```
bow2_id = dictionary.doc2bow(nouns2)
bow2_id
```

```
[(0, 1),
 (3, 1),
 (5, 1),
 (8, 1),
 (9, 1),
 (10, 1),
 (11, 1),
 (12, 1),
 (13, 1),
 (14, 1),
 (15, 1),
 (16, 1)]
```

```
bow3_id = dictionary.doc2bow(nouns3)
bow3_id
```

```
[(3, 2),
 (5, 2),
 (17, 1),
 (18, 1),
 (19, 1),
 (20, 1),
 (21, 1),
 (22, 2),
 (23, 1),
 (24, 1)]
```

このように、各インデックス番号に対して、辞書に対する単語の出現回数が 0 でない場合にカウントされた数が格納されていることがわかります。メモリを節約するために、出現した要素番号と数のみが返ってくるため、matutils.corpus2dense によって `[0, 0, 1, 0, ...]` のような長さが固定長になる固定長になることが望ましいです。BoW で変換しましょう。登録されている全単語数が引数に必要なので、最初に取得しておくことにします。

```
# 全単語数を取得
n_words = len(dictionary)

# 全単語数の長さに変換
bow = matutils.corpus2dense([bow1_id], n_words)
bow
```

```
array([[2.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [1.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.],
       [0.]], dtype=float32)
```

```
bow.shape
```

```
(25, 1)
```

計算できました。機械学習の入力値とする場合は入力ベクトルを横方向で格納することが多いので、転置を使いましょう。T を使うと転置できます。

```
# 横方向に変換
bow = matutils.corpus2dense([bow1_id], n_words).T
bow
```

```
array([[2., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

```
# 確認
bow.shape
```

```
(1, 25)
```

また、ベクトル化するため最初の要素のみ抽出します。

```
# ベクトル化
bow = matutils.corpus2dense([bow1_id], n_words).T[0]
bow
```

```
array([2., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

```
bow.shape
```

```
(25,)
```

期待どおりに動作することを確認できたので、この操作を全体に対しても適用しましょう。

```
# BoWによる特徴ベクトルの作成
x = []
for nouns in word_collect:
    bow_id = dictionary.doc2bow(nouns)
    bow = matutils.corpus2dense([bow_id], n_words).T[0]
    x.append(bow)

x[0]
```

```
array([2., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

```
x[1]
```

```
array([1., 0., 0., 1., 0., 1., 0., 0., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

このように、用意した辞書に基づいて文章中の単語数をカウントした BoW が作成されます。前出の図を再掲します。

BoW	1	0	1	1	0	1	0	1
私	1	0	0	0	0	0	0	0
は	0	0	0	1	0	0	0	0
キカガク	0	0	1	0	0	0	0	0
です	0	0	0	0	0	1	0	0
。	0	0	0	0	0	0	0	1

単語の one-hot 表現

TF-IDF とは

ここまで BoW を用いて文章から特徴ベクトルを生成する方法を扱ってきました。BoW はシンプルで扱いやすいですが、単語の出現回数をカウントするだけなので「です」「ます」など、大量に出現するわりに特に意味を持たない単語への重み付けが行われていません。

そこで考案された特徴抽出の手法が、**TF-IDF** です。TF-IDF は 2 つの項目の積として定義され、単語ごとに重み付けされた特徴ベクトルを生成します。

- **TF（Term Frequency）**：1 つの文章中で見た単語の出現頻度
- **IDF（Inverse Document Frequency）**：$\dfrac{1}{\text{すべての文章で見た単語の出現頻度}}$
- **TF-IDF**：$\dfrac{\text{1 つの文章中で見た単語の出現頻度}}{\text{すべての文章で見た単語の出現頻度}}$

言い換えると、以下のように整理できます。

- **TF**：1 つの文章中で何度も登場する単語の重みを増やす
- **IDF**：複数の文章にまたがって登場する単語の重みを減らす

このようにすることで、文書の特徴をより抽出できる単語に重み付けが行われます。

- 「野球」「サッカー」：スポーツ系の記事にだけ頻出するので重みが大きくなる
- 「です」「ます」：すべての記事で頻出するので重みが小さくなる

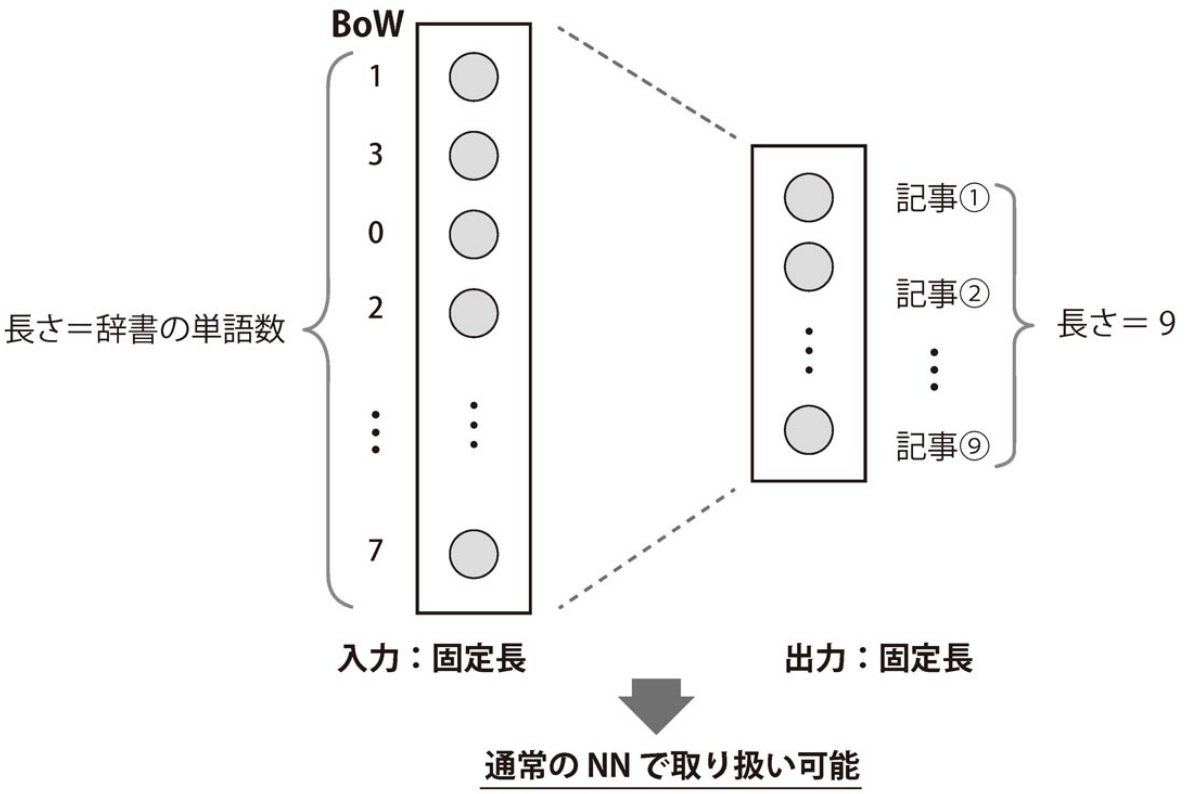
TF-IDF を用いた特徴ベクトルの生成はこの後実装するので、まずは概念を理解しておきましょう。

8.6 文書分類

自然言語処理に必要な前処理の手順がわかったので、実問題の 1 つである**文書分類**を扱います。

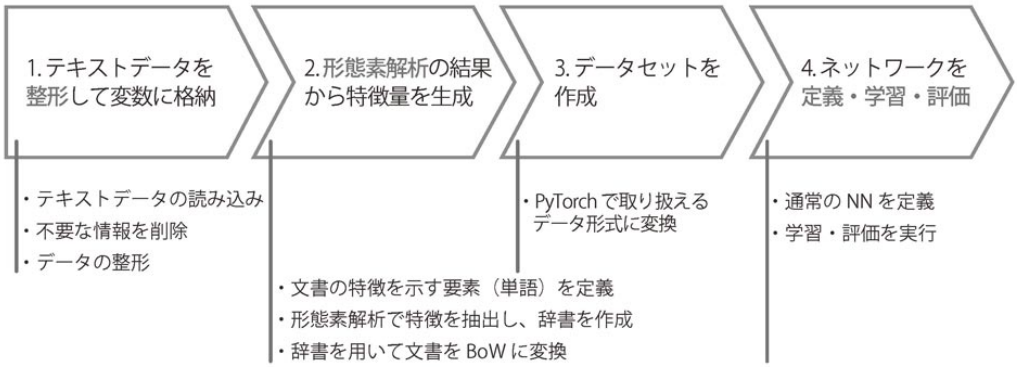
テーマとしては、入力した文書が事前に用意した 9 つのニュース記事のどれに該当するのか、カテゴリを予測する問題設定です。そのため出力は長さ 9 の固定長のベクトルです。また、文書を入力値として特徴抽出する際には BoW を使いますが、先ほど確認したとおり BoW は固定長ですから入力値も固定長となります。

つまり、自然言語処理のロードマップでも説明したように、文書分類で扱うデータは「入力：固定長、出力：固定長」なので、ここまで扱ってきた通常のニューラルネットワークで取り扱うことができます。



文書分類の流れ

テーマ設定が明確になったので、どういった流れになるかロードマップで確認しましょう。



テキストデータを整形して変数に格納

はじめに、フォルダ内のテキストデータを読み込む練習をします。今回は 9 つの記事カテゴリが文書分類の対象です。

テキストデータを読み込む際には glob というライブラリを使用します。テキストデータに限らず画像データや CSV など、あらゆるデータを読み込む際に glob が非常に有効なので、取り扱いを覚えておきましょう。glob を使うと、指定したフォルダ内に存在するファイルおよびフォルダの**相対 PATH** がリスト型に格納され、一覧で取得されます。

```
from glob import glob

# テキストデータの読み込み
dirs = glob('data/text/*')
dirs
```

```
['data/text/movie-enter',
 'data/text/it-life-hack',
 'data/text/kaden-channel',
 'data/text/topic-news',
 'data/text/livedoor-homme',
 'data/text/peachy',
 'data/text/sports-watch',
 'data/text/dokujo-tsushin',
 'data/text/smax']
```

取得された情報を見ると拡張子が付いていないので、フォルダの一覧が取得されたと考えられます。さらに glob で情報を取得してみます。

```
# 拡張子.txtが付くファイルの取得
filepaths = glob(dirs[0] + '/*.txt')
filepaths[:10]
```

```
['data/text/movie-enter/movie-enter-5978741.txt',
 'data/text/movie-enter/movie-enter-6322901.txt',
 'data/text/movie-enter/movie-enter-6176324.txt',
 'data/text/movie-enter/movie-enter-6573929.txt',
 'data/text/movie-enter/movie-enter-5914880.txt',
 'data/text/movie-enter/movie-enter-5878782.txt',
 'data/text/movie-enter/movie-enter-6354868.txt',
 'data/text/movie-enter/movie-enter-6035751.txt',
 'data/text/movie-enter/movie-enter-6689395.txt',
 'data/text/movie-enter/movie-enter-6375454.txt']
```

拡張子.txt が付いたテキストファイルの一覧を取得できました。これらのファイルの PATH を使用してテキストデータを読み込んでいくのですが、あわせてその文書がどのカテゴリに属する記事なのか示す目標値も用意する必要があります。

まずはテキストデータの読み込みを実施します。テキストデータの読み込みには open() と readlines() を組み合わせて次のように記述するので、覚えておきましょう。

```
# ファイルPATHを1つ取得
filepath = filepaths[1]
filepath
```

```
'data/text/movie-enter/movie-enter-6322901.txt'
```

```
# ファイルを取得
with open(filepath, encoding='utf-8') as f:
    text = f.readlines()
```

```
# 上から10行表示
text[:10]
```

```
['http://news.livedoor.com/article/detail/6322901/\n',
 '2012-02-29T11:45:00+0900\n',
 '藤原竜也、中学生とともにロケット打ち上げに成功\n',
 '\u3000「アンテナを張りながら生活をしていけばいい」\n',
 '\n',
 '\u30002月28日、映画『おかえり、はやぶさ』（3月10日より公開）と文部科学省とのタイアップとして、千代田区立神田一橋中学校に通う中学三年生と“宇宙”をテーマにした特別授業を行った。本作で主演を務める藤原竜也がサプライズで登場し、イベントを盛り上げた。\n',
 '\n',
 '\u3000イベントの挨拶で奥村展三文部科学副大臣は「みなさんは大きな夢を持っているということで、実現するために、文部科学省も環境を作り応援していますので、チャレンジ精神で頑張ってください。」と参加した生徒たちにエールを送った。\n',
 '\n',
 '\u3000今回の特別授業は、2部制で行われた。第1部では、ロケットの中はどうなっているのか、発射した時の音の大きさなど、ロケットの不思議を日本宇宙少年団とJAXA（宇宙航空研究開発機構）が説明。参加した生徒は興味深く授業を受けていた。講義の最後に藤原が登場すると、何も知らなかった生徒から大きな歓声が上がり、会場はさらに盛り上がった。\n']
```

不要な情報を取り除く

取得されたデータを見ると、1～2 行目に URL や日付などの本文とは関係のない情報が混ざっています。このような不要な情報は学習の妨げになるため、取り除いておきましょう。テキストデータがリスト型に格納されているため、3 行目以降のデータを抽出すれば解決するので、リスト型の要素番号を `[2:]` で指定するだけです。

```
# 3行目以降のデータを抽出
with open(filepath, encoding='utf-8') as f:
    text = f.readlines()[2:]

text[:10]
```

```
['藤原竜也、中学生とともにロケット打ち上げに成功\n',
 '\u3000「アンテナを張りながら生活をしていけばいい」\n',
 '\n',
 '\u30002月28日、映画『おかえり、はやぶさ』（3月10日より公開）と文部科学省とのタイアップとして、千代田区立神田一橋中学校に通う中学三年生と“宇宙”をテーマにした特別授業を行った。本作で主演を務める藤原竜也がサプライズで登場し、イベントを盛り上げた。\n',
 '\n',
 '\u3000イベントの挨拶で奥村展三文部科学副大臣は「みなさんは大きな夢を持っているということで、実現するために、文部科学省も環境を作り応援していますので、チャレンジ精神で頑張ってください。」と参加した生徒たちにエールを送った。\n',
 '\n',
 '\u3000今回の特別授業は、2部制で行われた。第1部では、ロケットの中はどうなっているのか、発射した時の音の大きさなど、ロケットの不思議を日本宇宙少年団とJAXA（宇宙航空研究開発機構）が説明。参加した生徒は興味深く授業を受けていた。講義の最後に藤原が登場すると、何も知らなかった生徒から大きな歓声が上がり、会場はさらに盛り上がった。\n',
 '\n',
 '\u3000第2部では、藤原も参加し、生徒とともにアルコールロケットを発射させた。ロケットと同じ原理で飛ばすことで、さらに理解を深めるというもの。実際の燃料は水素と酸素だが、今回は、水素が危険を伴うためアルコールで代用。発射させたアルコールロケットが見事にくす玉に命中すると、中から“祝！\u3000卒業「おかえり、はやぶさ」大ヒット祈願”という垂れ幕が登場した。参加した生徒132名との記念撮影も行われ、驚きと歓喜の中、イベントは終了した。\n']
```

データの整形

形態素解析を行う上で、文書全体を 1 つの文字列として扱う必要があるのですが、今は複数の文章に区切られてリスト型に格納されています。そのため、すべてのテキストを結合して 1 つの文字列にします。この処理には、リスト型の各要素を結合する常套手段である`''.join()` を使います。

```
# 3行目以降のデータを抽出
with open(filepath, encoding='utf-8') as f:
    text = f.readlines()[2:]
    text = ''.join(text)

text
```

'藤原竜也、中学生とともにロケット打ち上げに成功\n\u3000「アンテナを張りながら生活をしていけばいい」\n\n\u30002月28日、映画『おかえり、はやぶさ』（3月10日より公開）と文部科学省とのタイアップとして、千代田区立神田一橋中学校に通う中学三年生と“宇宙”をテーマにした特別授業を行った。本作で主演を務める藤原竜也がサプライズで登場し、イベントを盛り上げた。\n\n\u3000イベントの挨拶で奥村展三文部科学副大臣は「みなさんは大きな夢を持っているということで、実現するために、文部科学省も環境を作り応援していますので、チャレンジ精神で頑張ってください。」と参加した生徒たちにエールを送った。\n\n\u3000今回の特別授業は、2部制で行われた。第1部では、ロケットの中はどうなっているのか、発射した時の音の大きさなど、ロケットの不思議を日本宇宙少年団とJAXA（宇宙航空研究開発機構）が説明。参加した生徒は興味深く授業を受けていた。講義の最後に藤原が登場すると、何も知らなかった生徒から大きな歓声が上がり、会場はさらに盛り上がった。\n\n\u3000第2部では、藤原も参加し、生徒とともにアルコールロケットを発射させた。ロケットと同じ原理で飛ばすことで、さらに理解を深めるというもの。実際の燃料は水素と酸素だが、今回は、水素が危険を伴うためアルコールで代用。発射させたアルコールロケットが見事にくす玉に命中すると、中から“祝！\u3000卒業「おかえり、はやぶさ」大ヒット祈願”という垂れ幕が登場した。参加した生徒132名との記念撮影も行われ、驚きと歓喜の中、イベントは終了した。\n\n\u3000藤原から今回のイベントについてコメントも届いている。\n\n\n――『おかえり、はやぶさ』完成しましたね。\n藤原：素敵な映画ができました。今日集まってくれているのは中学三年生ということで、卒業を控えて大変なこともたくさんあると思います。でも諦めず前に進んでください。\n\n\n――久しぶりの学校だと思いますが、いかがですか？\n藤原：学校ってこんなに寒かったっけ？（笑）でも、そんな寒い中を友達と遊んだ事を思いだし、興奮しています。\n\n――藤原さんは宇宙に興味がありましたか？\n藤原：かなり興味があります。はやぶさもリアルタイムで見たり調べたりしていましたし、宇宙だけでなく、生物や生命体にも興味がありました。往復7年間、60億キロの宇宙の旅を、はやぶさは帰ってきたんですけど、今回、本作に出演しなければわからなかった、JAXAの人達のすごく大変な苦労を知ることができました。僕も含め普通の人だったらこんな絶望的な出来事は諦めてしまと思いますが、JAXAの人達は、数センチ、数ミリの希望を奇跡的にキャッチすることで帰還させたわけですから、すごいことですよね。\n\n\n――主演された役は夢と希望を持った役でしたが、中学三年生のみなさんと通じるところはありますか？\n藤原：すごくあると思います。映画の中で「成功とは意欲を失わずに、失敗に次ぐ失敗を繰りかえすことである。」というウィンストン・チャーチルの言葉があるんですが、まさに多くの経験をして失敗をしてもらって、意欲を失わないで次に繋げるということは、通じるところだと思います。\n\n\n――藤原さんの中学三年生の時の夢は何でしたか？\n藤原：僕はずっとサッカーしかしていませんでした。勉強もろくにやらなくて、学校へは給食を食べに行っているだけでした（笑）。サッカー選手になりたかったんですが、ある時に演劇の世界に入ることになり、気が付いたら演劇の世界に引っ張られていました。みなさんもどこでどんな道が待っているかわからないので、アンテナを張りながら生活をしていけばいいと思います。あとは、自分を変えたりするのは人との出会いだと思うので、たくさんの人と出会って、たくさん恋愛もして（笑）。楽に楽しく過ごすことが大事ですよね。\n\n\n――日本の宇宙開発プロジェクトに今後期待することは？\n藤原：日本の開発技術はアジアで1位でなくてはいけないという大変な状況の中で、多くの国民に希望を与えてくれたはやぶさですから、はやぶさ2号機に関しても、もっともっと国民一人一人が応援していければ、活気がでるんじゃないかなと思います。\n\n\n――最後に一言お願いします。\n藤原：JAXAの奇跡の物語を実感できますし、もう一つのテーマの、家族や仲間の絆というところもありますので、たくさんのみなさんに観てもらって、もっと宇宙へ興味を持ってもらいたいです。そして夢へ向かって少し光を射してくれるような映画になっていればいいなと思っています。\n\n\u3000本作は、JAXAの協力のもと、2011年6月13日に約60億キロの飛行の末、地球へと帰還した小惑星探査機“はやぶさ”にまつわるさまざまなエピソードを3D映画化したもの。出演には、藤原を始め、杏、三浦友和、まえだまえだの前田旺志郎など各年代のキャストが勢揃いし、はやぶさと人々の冒険の旅を描く。\n\n\u3000映画『おかえり、はやぶさ』は、3月10日(土)より3D&2D同時公開。\n\n・映画『おかえり、はやぶさ』 - 作品情報\n\n【関連記事】\n・藤原竜也、杏ら日本の俳優のスケールでは宇宙に行けない\n・「はやぶさ」のグルメ登場！\u3000うな重にカレーにケーキまで\n'

さらにここで、文中に挿入されているエスケープシーケンス\n および\u3000 を削除しておきましょう。該当する文字列を空文字'' に置換するだけです。

```
# エスケープシーケンスを置換
with open(filepath, encoding='utf-8') as f:
    text = f.readlines()[2:]
    text = ''.join(text)
    text = text.replace('\u3000', '')
    text = text.replace('\n', '')

text
```

'藤原竜也、中学生とともにロケット打ち上げに成功「アンテナを張りながら生活をしていけばいい」2月28日、映画『おかえり、はやぶさ』（3月10日より公開）と文部科学省とのタイアップとして、千代田区立神田一橋中学校に通う中学三年生と“宇宙”をテーマにした特別授業を行った。本作で主演を務める藤原竜也がサプライズで登場し、イベントを盛り上げた。イベントの挨拶で奥村展三文部科学副大臣は「みなさんは大きな夢を持っているということで、実現するために、文部科学省も環境を作り応援していますので、チャレンジ精神で頑張ってください。」と参加した生徒たちにエールを送った。今回の特別授業は、2部制で行われた。第1部では、ロケットの中はどうなっているのか、発射した時の音の大きさなど、ロケットの不思議を日本宇宙少年団とJAXA（宇宙航空研究開発機構）が説明。参加した生徒は興味深く授業を受けていた。講義の最後に藤原が登場すると、何も知らなかった生徒から大きな歓声が上がり、会場はさらに盛り上がった。第2部では、藤原も参加し、生徒とともにアルコールロケットを発射させた。ロケットと同じ原理で飛ばすことで、さらに理解を深めるというもの。実際の燃料は水素と酸素だが、今回は、水素が危険を伴うためアルコールで代用。発射させたアルコールロケットが見事にくす玉に命中すると、中から“祝！卒業「おかえり、はやぶさ」大ヒット祈願”という垂れ幕が登場した。参加した生徒132名との記念撮影も行われ、驚きと歓喜の中、イベントは終了した。藤原から今回のイベントについてコメントも届いている。――『おかえり、はやぶさ』完成しましたね。藤原：素敵な映画ができました。今日集まってくれているのは中学三年生ということで、卒業を控えて大変なこともたくさんあると思います。でも諦めず前に進んでください。――久しぶりの学校だと思いますが、いかがですか？藤原：学校ってこんなに寒かったっけ？（笑）でも、そんな寒い中を友達と遊んだ事を思いだし、興奮しています。――藤原さんは宇宙に興味がありましたか？藤原：かなり興味があります。はやぶさもリアルタイムで見たり調べたりしていましたし、宇宙だけでなく、生物や生命体にも興味がありました。往復7年間、60億キロの宇宙の旅を、はやぶさは帰ってきたんですけど、今回、本作に出演しなければわからなかった、JAXAの人達のすごく大変な苦労を知ることができました。僕も含め普通の人だったらこんな絶望的な出来事は諦めてしまと思いますが、JAXAの人達は、数センチ、数ミリの希望を奇跡的にキャッチすることで帰還させたわけですから、すごいことですよね。――主演された役は夢と希望を持った役でしたが、中学三年生のみなさんと通じるところはありますか？藤原：すごくあると思います。映画の中で「成功とは意欲を失わずに、失敗に次ぐ失敗を繰りかえすことである。」というウィンストン・チャーチルの言葉があるんですが、まさに多くの経験をして失敗をしてもらって、意欲を失わないで次に繋げるということは、通じるところだと思います。――藤原さんの中学三年生の時の夢は何でしたか？藤原：僕はずっとサッカーしかしていませんでした。勉強もろくにやらなくて、学校へは給食を食べに行っているだけでした（笑）。サッカー選手になりたかったんですが、ある時に演劇の世界に入ることになり、気が付いたら演劇の世界に引っ張られていました。みなさんもどこでどんな道が待っているかわからないので、アンテナを張りながら生活をしていけばいいと思います。あとは、自分を変えたりするのは人との出会いだと思うので、たくさんの人と出会って、たくさん恋愛もして（笑）。楽に楽しく過ごすことが大事ですよね。――日本の宇宙開発プロジェクトに今後期待することは？藤原：日本の開発技術はアジアで1位でなくてはいけないという大変な状況の中で、多くの国民に希望を与えてくれたはやぶさですから、はやぶさ2号機に関しても、もっともっと国民一人一人が応援していければ、活気がでるんじゃないかなと思います。――最後に一言お願いします。藤原：JAXAの奇跡の物語を実感できますし、もう一つのテーマの、家族や仲間の絆というところもありますので、たくさんのみなさんに観てもらって、もっと宇宙へ興味を持ってもらいたいです。そして夢へ向かって少し光を射してくれるような映画になっていればいいなと思っています。本作は、JAXAの協力のもと、2011年6

月13日に約60億キロの飛行の末、地球へと帰還した小惑星探査機“はやぶさ”にまつわるさまざまなエピソードを3D映画化したもの。出演には、藤原を始め、杏、三浦友和、まえだまえだの前田旺志郎など各年代のキャストが勢揃いし、はやぶさと人々の冒険の旅を描く。映画『おかえり、はやぶさ』は、3月10日(土)より3D&2D同時公開。・映画『おかえり、はやぶさ』 - 作品情報【関連記事】・藤原竜也、杏ら日本の俳優のスケールでは宇宙に行けない・「はやぶさ」のグルメ登場！うな重にカレーにケーキまで'

これでテキストデータを読み込む一連の手順を理解できたので、関数にまとめておきましょう。

```
# 前処理関数化
def preprocessing(filepath):
    with open(filepath, encoding='utf-8') as f:
        text = f.readlines()[2:]
        text = ''.join(text)
        text = text.replace('\u3000', '')
        text = text.replace('\n', '')
    return text

filepath = filepaths[1]
filepath
```

```
'data/text/movie-enter/movie-enter-6322901.txt'
```

```
preprocessing(filepath)
```

'藤原竜也、中学生とともにロケット打ち上げに成功「アンテナを張りながら生活をしていけばいい」2月28日、映画『おかえり、はやぶさ』（3月10日より公開）と文部科学省とのタイアップとして、千代田区立神田一橋中学校に通う中学三年生と“宇宙”をテーマにした特別授業を行った。本作で主演を務める藤原竜也がサプライズで登場し、イベントを盛り上げた。イベントの挨拶で奥村展三文部科学副大臣は「みなさんは大きな夢を持っているということで、実現するために、文部科学省も環境を作り応援していますので、チャレンジ精神で頑張ってください。」と参加した生徒たちにエールを送った。今回の特別授業は、2部制で行われた。第1部では、ロケットの中はどうなっているのか、発射した時の音の大きさなど、ロケットの不思議を日本宇宙少年団とJAXA（宇宙航空研究開発機構）が説明。参加した生徒は興味深く授業を受けていた。講義の最後に藤原が登場すると、何も知らなかった生徒から大きな歓声が上がり、会場はさらに盛り上がった。第2部では、藤原も参加し、生徒とともにアルコールロケットを発射させた。ロケットと同じ原理で飛ばすことで、さらに理解を深めるというもの。実際の燃料は水素と酸素だが、今回は、水素が危険を伴うためアルコールで代用。発射させたアルコールロケットが見事にくす玉に命中すると、中から“祝！卒業「おかえり、はやぶさ」大ヒット祈願”という垂れ幕が登場した。参加した生徒132名との記念撮影も行われ、驚きと歓喜の中、イベントは終了した。藤原から今回のイベントについてコメントも届いている。――『おかえり、はやぶさ』完成しましたね。藤原：素敵な映画ができました。今日集まってくれているのは中学三年生ということで、卒業を控えて大変なこともたくさんあると思います。でも諦めず前に進んでください。――久しぶりの学校だと思いますが、いかがですか？藤原：学校ってこんなに寒かったっけ？（笑）でも、そんな寒い中を友達と遊んだ事を思いだし、興奮しています。――藤原さんは宇宙に興味がありましたか？藤原：かなり興味があります。はやぶさもリアルタイムで見たり調べたりしていましたし、宇宙だけでなく、生物や生命体にも興味

```
がありました。往復7年間、60億キロの宇宙の旅を、はやぶさは帰ってきたんですけど、今回、本作に出演しなければわからなかった、JAXAの人達のすごく大変な苦労を知ることができました。僕も含め普通の人だったらこんな絶望的な出来事は諦めてしまと思いますが、JAXAの人達は、数センチ、数ミリの希望を奇跡的にキャッチすることで帰還させたわけですから、すごいことですよね。――主演された役は夢と希望を持った役でしたが、中学三年生のみなさんと通じるところはありますか？藤原：すごくあると思います。映画の中で「成功とは意欲を失わずに、失敗に次ぐ失敗を繰りかえすことである。」というウィンストン・チャーチルの言葉があるんですが、まさに多くの経験をして失敗をしてもらって、意欲を失わないで次に繋げるということは、通じるところだと思います。――藤原さんの中学三年生の時の夢は何でしたか？藤原：僕はずっとサッカーしかしていませんでした。勉強もろくにやらなくて、学校へは給食を食べに行っているだけでした（笑）。サッカー選手になりたかったんですが、ある時に演劇の世界に入ることになり、気が付いたら演劇の世界に引っ張られていました。みなさんもどこでどんな道が待っているかわからないので、アンテナを張りながら生活をしていけばいいと思います。あとは、自分を変えたりするのは人との出会いだと思うので、たくさんの人と出会って、たくさん恋愛もして（笑）。楽に楽しく過ごすことが大事ですよね。――日本の宇宙開発プロジェクトに今後期待することは？藤原：日本の開発技術はアジアで1位でなくてはいけないという大変な状況の中で、多くの国民に希望を与えてくれたはやぶさですから、はやぶさ2号機に関しても、もっともっと国民一人一人が応援していければ、活気がでるんじゃないかなと思います。――最後に一言お願いします。藤原：JAXAの奇跡の物語を実感できますし、もう一つのテーマの、家族や仲間の絆というところもありますので、たくさんのみなさんに観てもらって、もっと宇宙へ興味を持ってもらいたいです。そして夢へ向かって少し光を射してくれるような映画になっていればいいなと思っています。本作は、JAXAの協力のもと、2011年6月13日に約60億キロの飛行の末、地球へと帰還した小惑星探査機“はやぶさ”にまつわるさまざまなエピソードを3D映画化したもの。出演には、藤原を始め、杏、三浦友和、まえだまえだの前田旺志郎など各年代のキャストが勢揃いし、はやぶさと人々の冒険の旅を描く。映画『おかえり、はやぶさ』は、3月10日(土)より3D&2D同時公開。・映画『おかえり、はやぶさ』 - 作品情報【関連記事】・藤原竜也、杏ら日本の俳優のスケールでは宇宙に行けない・「はやぶさ」のグルメ登場！うな重にカレーにケーキまで'
```

ファイルからテキストデータを読み込む関数ができあがったので、for 文を用いて全体に適用しましょう。また、テキストデータが属するカテゴリ（目標値）も取得しておく必要があるので、for 文内で enumerate を使って要素番号を目標値とします。

```
# enumerateの使用方法
for (label, dir) in enumerate(dirs):
    print('要素番号：', label)
    print('フォルダ：', dir)
    print('――――')
```

```
要素番号： 0
フォルダ： data/text/movie-enter
――――
要素番号： 1
フォルダ： data/text/it-life-hack
――――
要素番号： 2
フォルダ： data/text/kaden-channel
――――
要素番号： 3
フォルダ： data/text/topic-news
```

```
――――
要素番号： 4
フォルダ： data/text/livedoor-homme
――――
要素番号： 5
フォルダ： data/text/peachy
――――
要素番号： 6
フォルダ： data/text/sports-watch
――――
要素番号： 7
フォルダ： data/text/dokujo-tsushin
――――
要素番号： 8
フォルダ： data/text/smax
――――
```

これらの手順を 1 つにまとめて、すべてのテキストデータを取得しましょう。

```
texts, labels = [], []

for (label, dir) in enumerate(dirs):
    # 各ディレクトリ内のファイルPATHを全取得
    filepaths = glob('{}/*'.format(dir))

    for filepath in filepaths:
        # テキストデータ取得
        text = preprocessing(filepath)
        texts.append(text)

        # 正解ラベル作成
        labels.append(label)

# 確認のためデータを表示
labels[1]
```

```
0
```

```
# 最初の100文字だけ表示
texts[1][:100]
```

```
'藤原竜也、中学生とともにロケット打ち上げに成功「アンテナを張りながら生活をしていけばいい」2月28日、映画『おかえり、はやぶさ』（3月10日より公開）と文部科学省とのタイアップとして、千代田区立神田一'
```

形態素解析の結果から特徴量を生成する

すべての入力値と、目標値を取得することができたので、入力値に形態素解析を行って特徴量を生成していきます。最初に文書の特徴を示す要素（単語）を定義しましょう。

今回は記事のカテゴリを予測するテーマですが、どのような単語が文書の特徴を示すのでしょうか？ たとえばスポーツに関する記事であれば「野球」「サッカー」、ファッションに関する記事であれば「靴」「洋服」といった単語が多く登場すると考えられます。つまり、記事のカテゴリを予測する上では**名詞**が重要な特徴を示す要素であると言えます。作成した名詞抽出用の関数 get_nouns を使って、文書全体で使用されている名詞をすべて word_collect というリストに格納していきましょう。

```
mecab = MeCab.Tagger('-Ochasen')

# 名詞抽出用関数
def get_nouns(text):
    nouns = []
    res = mecab.parse(text)
    words = res.split('\n')[:-2]
    for word in words:
        part = word.split('\t')
        if '名詞' in part[3]:
            nouns.append(part[0])
    return nouns

# 抽出した名詞を格納
word_collect = []
for text in texts:
    nouns = get_nouns(text)
    word_collect.append(nouns)

# 最初の20個だけ表示
word_collect[1][:20]
```

```
['藤原',
 '竜也',
 '中学生',
 'ロケット',
 '成功',
 'アンテナ',
```

```
 '生活',
 '2',
 '月',
 '28',
 '日',
 '映画',
 '3',
 '月',
 '10',
 '日',
 '公開',
 '文部',
 '科学',
 '省']
```

すべての名詞が格納されたリストを作成できたので、gensim を用いて BoW 用の辞書を作成します。

```
from gensim import corpora, matutils

# 辞書に含まれる単語数を確認
dictionary = corpora.Dictionary(word_collect)
len(dictionary)
```

```
62928
```

このまま辞書を使用してもよいのですが、出現回数が少ない単語をフィルタリングすることで特徴のある単語のみに絞れます。逆に、出現回数が多すぎる「私」「彼」などの単語は、文章中で重要な意味を持たないことも考えられるので、後述の「特徴量を BoW から TF-IDF に変更」で扱います。

```
# 出現回数でフィルタリング
dictionary.filter_extremes(no_below=20)

len(dictionary)
```

```
7295
```

このように全体で 20 回以上出現しない単語はフィルタリングすることで、単語数を抑えることができました。より重要な単語で特徴抽出できたこととあわせて、全体の計算量を削減する効果もあります。また、毎回辞書を作成しなくても済むよう、この段階で保存しておきましょう。

```
# 後から使えるように保存
dictionary.save_as_text('livedoordic.txt')
```

辞書を用いて文書を BoW に変換する

それでは作成した辞書を使って、文書を BoW に変換しましょう。

```
# 辞書内の単語数を取得
n_words = len(dictionary)

# BoWによる特徴ベクトルの作成
x = []
for nouns in word_collect:
    bow_id = dictionary.doc2bow(nouns)
    bow = matutils.corpus2dense([bow_id], n_words).T[0]
    x.append(bow)

import numpy as np

# BoW目標値をNumPy配列に変換
x = np.array(x)
t = np.array(labels)

x.shape, t.shape
```

```
((7376, 7295), (7376,))
```

```
# 作成したBoWを確認
x[0]
```

```
array([1., 1., 3., ..., 0., 0., 0.], dtype=float32)
```

```
x[0].shape
```

```
(7295,)
```

ここまでの手順で文書を BoW に変換できました。.shape で確認したとおり、入力 x は長さ 7295 のベクトルです。

学習の準備

機械学習で扱える形式でデータセットを準備できたので、PyTorch で取り扱えるデータセットの形式への変換までしておきましょう。

```
import torch
from torch.utils.data import DataLoader, TensorDataset, random_split

# PyTorchで学習に使用できる形式へ変換
x = torch.tensor(x, dtype=torch.float32)
t = torch.tensor(t, dtype=torch.int64)

# 入力値と目標値をまとめて、1つのオブジェクトdatasetに変換
dataset = TensorDataset(x, t)

# train : val : test = 60% : 20% : 20%
n_train = int(len(dataset) * 0.6)
n_val = int(len(dataset) * 0.2)
n_test = len(dataset) - n_train - n_val

# ランダムに分割を行うため、シードを固定して再現性を確保
torch.manual_seed(0)

# データセットの分割
train, val, test = random_split(dataset, [n_train, n_val, n_test])

# バッチサイズ
batch_size = 128

# Data Loaderを用意
train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True, drop_last=True)
val_loader = torch.utils.data.DataLoader(val, batch_size)
test_loader = torch.utils.data.DataLoader(test, batch_size)
```

ネットワークの定義と学習

文書分類は入力・出力ともに固定長なので、通常の分類問題と特に変わりはありません。シンプルに 3 層の NN を定義していきましょう。

```
import torch.nn as nn
import torch.nn.functional as F
import pytorch_lightning as pl

class Net(pl.LightningModule):

  def __init__(self):
    super().__init__()
```

```
    self.bn = nn.BatchNorm1d(7295)
    self.fc1 = nn.Linear(7295, 200)
    self.fc2 = nn.Linear(200, 9)

    self.train_acc = pl.metrics.Accuracy()
    self.val_acc = pl.metrics.Accuracy()
    self.test_acc = pl.metrics.Accuracy()

  def forward(self, x):
    h = self.bn(x)
    h = self.fc1(h)
    h = F.relu(h)
    h = self.fc2(h)
    return h

  def training_step(self, batch, batch_idx):
    x, t = batch
    y = self(x)
    loss = F.cross_entropy(y, t)
    self.log('train_loss', loss, on_step=True, on_epoch=True)
    self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
    return loss

  # 検証データに対する処理
  def validation_step(self, batch, batch_idx):
    x, t = batch
    y = self(x)
    loss = F.cross_entropy(y, t)
    self.log('val_loss', loss, on_step=False, on_epoch=True)
    self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
    return loss

  # テストデータに対する処理
  def test_step(self, batch, batch_idx):
    x, t = batch
    y = self(x)
    loss = F.cross_entropy(y, t)
    self.log('test_loss', loss, on_step=False, on_epoch=True)
    self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
    return loss

  def configure_optimizers(self):
    optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
    return optimizer
```

ネットワークの学習

それでは、ネットワークの学習に入ります。

```
# 学習の実行
pl.seed_everything(0)
net = Net(0)
trainer = pl.Trainer(max_epochs=30)
trainer.fit(net, train_loader, val_loader)

# テストデータで検証
results = trainer.test(test_dataloaders=test_loader)
results
```

```
{'test_loss': 0.3535411059856415, 'test_acc': 0.9300704598426819}
```

test_acc を見ると、93% の予測精度で記事のカテゴリを予測できると確認できました。

特徴量を BoW から TF-IDF に変更

復習になりますが、TF-IDF では文書内・文書間それぞれの出現頻度で単語を重み付けするので、「です」「ます」などの頻度が多いわりに重要ではない単語の重要度を低くすることができます。問題設定が簡単なのですでに高い予測精度が出ていますが、BoW と TF-IDF それぞれの結果検証を行ってみましょう。

BoW の実装が済んでいれば、TF-IDF の実装は非常に簡単です。scikit-learn に用意されている TfidfTransformer を使用すると、BoW をもとに生成された特徴ベクトルを TF-IDF に基づいて生成された特徴ベクトルに変換できます。

```
from sklearn.feature_extraction.text import TfidfTransformer

# BoWによる特徴ベクトルの作成
n_words = len(dictionary)
x = []
for nouns in word_collect:
    bow_id = dictionary.doc2bow(nouns)
    bow = matutils.corpus2dense([bow_id], n_words).T[0]
    x.append(bow)

# インスタンス化
tfidf = TfidfTransformer()

# precisionは小数点以下の桁数の指定
np.set_printoptions(precision=4)
```

```
# TF-IDFによる特徴ベクトルの生成
tf_idf = tfidf.fit_transform(x).toarray()
```

これだけで TF-IDF による特徴ベクトルの生成が完了しました。BoW と TF-IDF それぞれを比較してみると、単語の出現回数をカウントするだけの BoW よりも細かな特徴を捉えることに変化したことがわかります。

```
# 変換前：BoW
x[:2]
```

```
[array([1., 1., 3., ..., 0., 0., 0.], dtype=float32),
 array([1., 1., 1., ..., 0., 0., 0.], dtype=float32)]
```

```
# 変換後：TF-IDF
tf_idf[:2]
```

```
array([[0.0217, 0.0238, 0.0477, ..., 0.    , 0.    , 0.    ],
       [0.0155, 0.017 , 0.0114, ..., 0.    , 0.    , 0.    ]])
```

TF-IDF で学習を実行

```
# PyTorchで学習できる形式に変換
x = torch.tensor(tf_idf, dtype=torch.float32)
t = torch.tensor(labels, dtype=torch.int64)

type(x), x.dtype, type(t), t.dtype
```

```
(torch.Tensor, torch.float32, torch.Tensor, torch.int64)
```

```
dataset = torch.utils.data.TensorDataset(x, t)

torch.manual_seed(0)
train, val, test = torch.utils.data.random_split(dataset, [n_train, n_val, n_test])

train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True, drop_last=True)
val_loader = torch.utils.data.DataLoader(val, batch_size)
test_loader = torch.utils.data.DataLoader(test, batch_size)
```

```
# 学習の実行
pl.seed_everything(0)
net = Net(0)
trainer = pl.Trainer(max_epochs=30)
trainer.fit(net, train_loader, val_loader)
results = trainer.test(test_dataloaders=test_loader)
results
```

```
{'test_loss': 0.22957424819469452, 'test_acc': 0.9280407428741455}
```

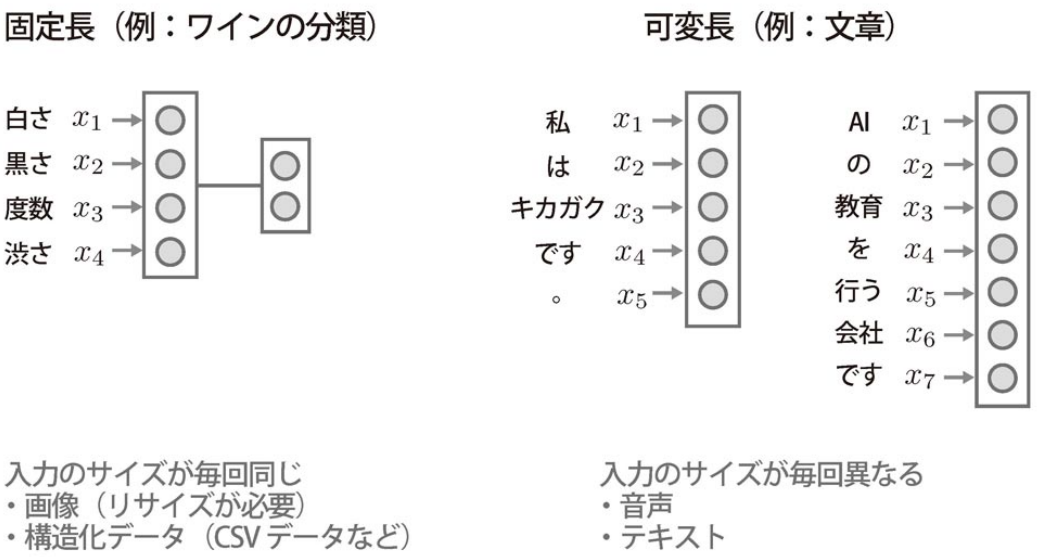
問題設定が簡単だったのか、あまり変化はありませんでした。ただ、文書内部の単語の出現頻度だけでなく、文書全体を通じた単語の出現頻度で重みを調整することで、予測精度の改善を期待できることがわかりました。

8.7 文章分類

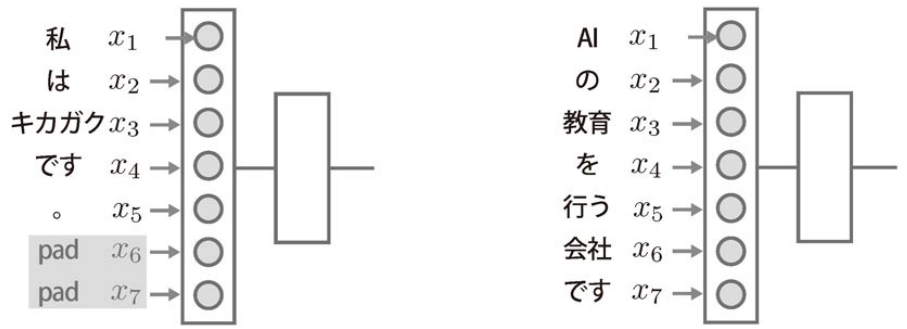
続いては、著名な作家の作品を判別できるようネットワークを訓練して、新たに入力された文章がどの作家の文章なのか予測する問題設定です。ここでは 3 名の作家の小説から文章を用意しました。

- 太宰 治
- 森 鴎外
- 坂口 安吾

これまでの問題設定と大きく異なる点としては、ネットワークへの入力が**可変長のデータ**であることです。可変長のデータをどのように工夫してネットワークに訓練させるのか、**パディング**と呼ばれる手法を用いて理解していきましょう。出力については、あらかじめ用意した数人の中から該当する作家を選ぶので、引き続き固定長です。



可変長のデータでも、パディングを使えば、入力のサイズを揃えて固定長のように扱うことができます。



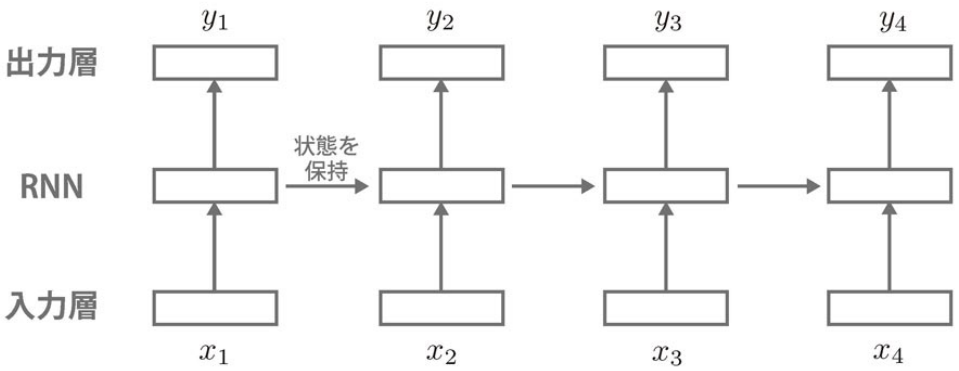
短い方の文章に「pad」を保管して、入力のサイズを揃える。

文章は**系列データ**と呼ばれ、「私」という名詞の次に「は」といった助詞が続くことが多いように、ある入力値が次以降（もしくはそれ以前）の入力値と関連し合って**順番に並んでいる**データを指します。

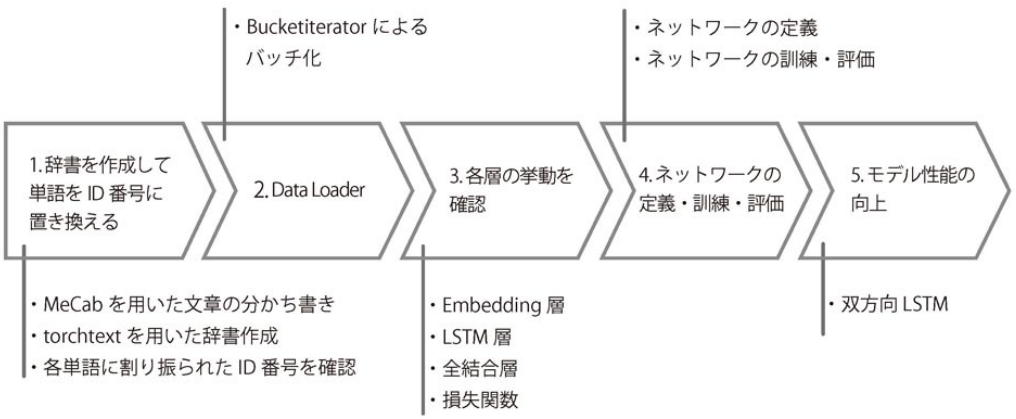
特に時間順で並んでいるデータを**時系列データ**と呼ぶので、あわせて覚えておきましょう。身

近な例として株価をイメージするとわかりやすいでしょう。こういった系列データでは、ある時点の入力情報を記憶し、次以降に伝えないと、うまく訓練が進みません。今回の問題設定でいうと、系列データである文章中に存在する「文脈」を捉えることに該当します。

そのために考案されたアルゴリズムが、**RNN**（Recurrent Neural Network）です。RNN は自然言語処理の分野で高い成果を挙げており、ディープラーニングで多く使用されているアルゴリズムの 1 つです。パディングとあわせて、可変長であることが多い系列データをどのように取り扱うのか一連のプロセスを理解しましょう。



問題設定が理解できたところで、全体像を確認しましょう。



辞書を作成して単語を ID 番号に置き換える

前出の文書分類では `MeCab.Tagger()` のオプションに`-Ochasen` を使用しましたが、今回は`-O wakati` を使用します。

```
# オプションに-Owakatiを使用
mecab = MeCab.Tagger('-Owakati')

res = mecab.parse('こんにちは。私の名前はキカガクです。')
res
```

```
'こんにちは 。 私 の 名前 は キカガク です 。 \n'
```

リスト型に変換し、エスケープシーケンス\n を取り除きます。

```
# split()でリスト型に変換
res = res.split(' ')
res
```

```
['こんにちは', '。', '私', 'の', '名前', 'は', 'キカガク', 'です', '。', '\n']
```

```
# エスケープシーケンスを取り除く
res = res[:-1]
res
```

```
['こんにちは', '。', '私', 'の', '名前', 'は', 'キカガク', 'です', '。']
```

次に、前処理で使用するための関数を作成しておきます。

```
def tokenize(text):
    return mecab.parse(text).split(' ')[:-1]

# 分かち書きを実行
tokenize('こんにちは。私の名前はキカガクです。')
```

```
['こんにちは', '。', '私', 'の', '名前', 'は', 'キカガク', 'です', '。']
```

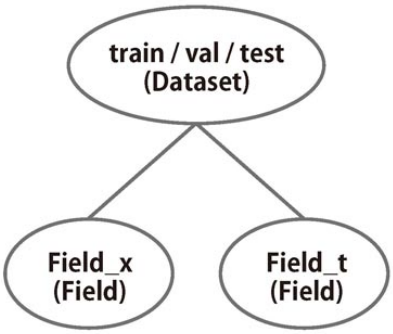
torchtext を用いた辞書作成

PyTorch には自然言語の前処理に特化した torchtext というライブラリがあります。gensim も非常に便利なライブラリでしたが、torchtext を使うと、データをオブジェクトとして包含していく形で、シンプルな表記で汎用性高く各変数を定義できます。torchtext で頻繁に使用する

モジュールは次の 3 つです。

- Field：入力・出力データごとに分かち書きの前処理と辞書を保持できる Field オブジェクトを生成します。前処理には自作の関数を定義しておき、オプションで引数として渡します。
- TabularDataset：CSV・TSV・JSON からデータを読み込んで、辞書作成のベースとなるテキストデータを保持する Dataset オブジェクトを生成します。前処理を定義した Field クラスを包含するので、辞書作成の際には Dataset オブジェクトを渡すだけで済み、以降のコードをシンプルに書けます。
- BucketIterator：DataLoader と同様の役割で、バッチごとに区切ってくれます。

包含関係を図示すると以下のとおりです。




```
from torchtext.data import Field, TabularDataset
```

では、辞書を作成していきましょう。まずは Field クラスを定義します。辞書作成に必要な前処理に応じて、さまざまなオプションを設定できます。() 内はデフォルトの設定値です。

- sequential（True）： テキストデータが可変長かどうかを示します。
- use_vocab（True）：すでに数値化された正解ラベルを使用する場合など、辞書を作成する必要がない場合に False とします。
- init_token（None）：「文章の最初」であることを示す文字列が必要な際に<sos>と設定します。
- eos_token（None）：「文章の最後」であることを示す文字列が必要な際に<eos>と設定します。
- tokenize（string.split）：自身で定義した分かち書き用の関数を引数から渡す際に使用します。

- is_target（False）：目標値かどうかを表す場合に使用します。

全部で 19 個のオプションがありますが、本書に関連するものをピックアップしました。すべてを確認したい方は torchtext 公式ドキュメント（https://torchtext.readthedocs.io/en/latest/data.html#fields）を参照してください。

```
# 入力値
field_x = Field(
    # 作成した関数を渡す
    tokenize = tokenize,
)

# 目標値
field_t = Field(
    # 目標値が数値のカテゴリ情報の場合に以下3つをセットで指定
    sequential = False,
    use_vocab = False,
    is_target = True
)
```

Field クラスが定義できたので、TabularDataset を用いて辞書作成のベースとなるテキストデータを読み込みましょう。

```
# CSVからデータを読み込む
train, val, test = TabularDataset.splits(
    path = 'data/novel/',
    train = 'train.csv',
    validation = 'test.csv',
    test = 'test.csv',
    format='csv',
    fields = [('x',field_x), ('t', field_t)]
)

len(train), len(val), len(test)
```

```
(12000, 3000, 3000)
```

指定した前処理が適切に行われているか確認しましょう。TabularDataset で作成したデータセットの内部を確認するには、vars() を使用します。なお、vars() は PyTorch 特有の関数ではなく、一般的にはクラスやインスタンスなどが持っている **dict** 属性（辞書型変数）を表示する際にも使用します。

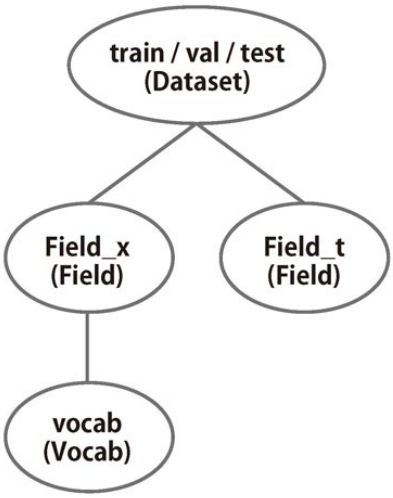
```
vars(train[0])['t']
```

```
'0'
```

```
vars(train[0])['x']
```

```
['…',
 '…',
 'ら',
 'す',
 'ほど',
 'その',
 'なか',
 'から',
 '赤',
 'や',
 '青',
 'や',
 '朽葉',
 'の',
 '色',
 'が',
 '湧い',
 'て',
 '来る']
```

データの読み込みが済んだので、Dataset オブジェクト内部で保持している Field クラスの build_vocab() メソッドを用いることで、簡単に辞書作成と ID 付与ができます。辞書を作成するだけであれば gensim でも可能ですが、以下の図に整理するように、Dataset、Field それぞれのオブジェクト内部に辞書が包含されるため、後述の BucketIterator に Dataset オブジェクトである train・val・test を引数として渡してあげるだけで簡単にその後の前処理を実施できるようになります。




```
# 辞書を作成
field_x.build_vocab(train, val, min_freq=3)
```

辞書に含まれる単語数を確認するには、.vocab と len() を組み合わせます。ネットワークの入力サイズ（one-hot 表現されたベクトルのサイズ）を指定する際に必要です。

```
# 辞書内の単語数を確認
len(field_x.vocab)
```

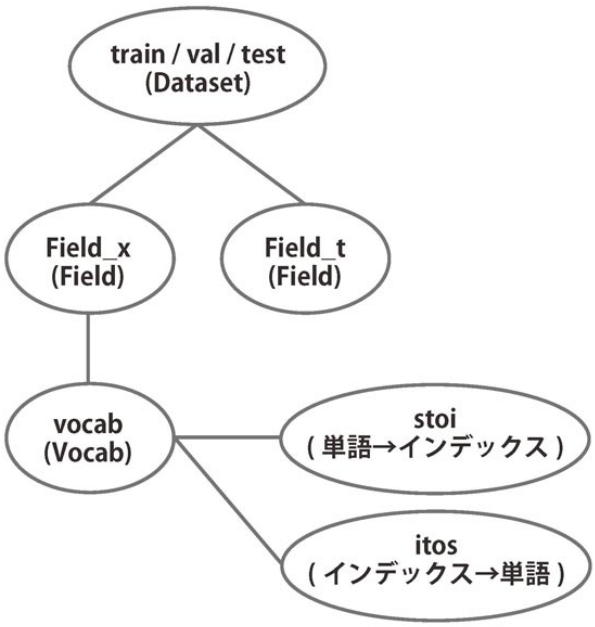
```
8402
```

各単語に割り振られた ID 番号を確認

PyTorch ではこのように、torchtext を利用して簡単に辞書を作成できます。各単語に割り振られた ID を確認するには、stoi を使います。

<unk>など、特殊な文字列が含まれているのは以下の理由です。また、<sos>と<eos>については Field クラスのオプションで指定したため登場します。

- **unk**：unknown の略。辞書に含まれない単語は<unk>で置換されます。
- **pad**：padding の略。最も長い文章を基準として、それより短い文章は<pad>を補完することで長さを調節します。
- **sos**：start of string の略。「文章の最初」であることを示す役割があります。
- **eos**：end of string の略。「文章の最後」であることを示す役割があります。




```
# 単語 -> インデックス
field_x.vocab.stoi
```

```
Out[205]: defaultdict(<bound method Vocab._default_unk_index of <torchtext.vocab.Vocab object at 0x1869f3410>>,
                      {'<unk>': 0,
                       '<pad>': 1,
                       'の': 2,
                       'た': 3,
                       '、': 4,
                       'に': 5,
                       'て': 6,
                       'は': 7,
                       'を': 8,
                       'が': 9,
                       'と': 10,
                       'で': 11,
                       'も': 12,
                       'し': 13,
                       'な': 14,
                       'ない': 15,
                       'から': 16,
                       'へ': 17,
                       'い': 18,
                       'ある': 19,
```

この結果はとても長くなるので、上から 20 単語の結果を示しました。

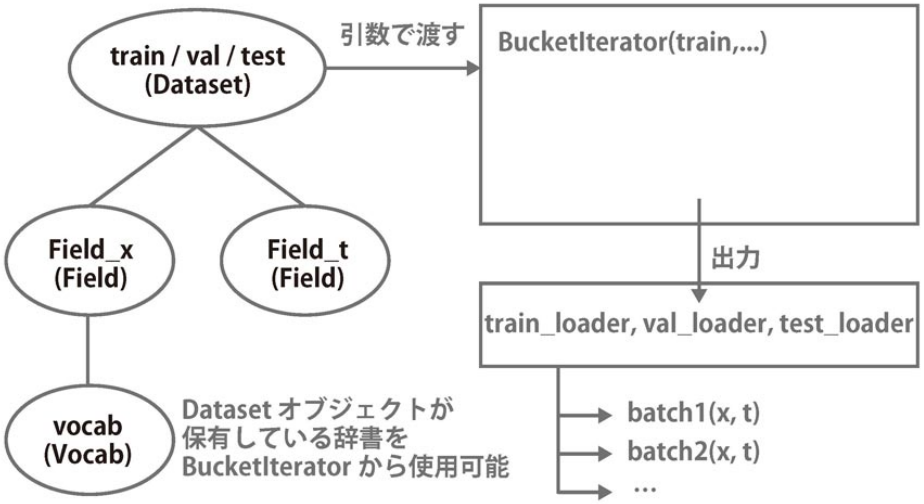
ミニバッチ学習の準備を行う

ここまでの処理でデータセットが用意できたので、ミニバッチ学習の準備を行いましょう。

torchtext に用意されているモジュール BucketIterator を用いると、Field クラスが保持している辞書を使って簡単に DataLoader を定義できます。行われる裏側の処理は以下のとおりです。

- イテレーションごとにミニバッチを渡すイテレータを定義
- Field クラスが保持している辞書に基づいて単語をインデックス化
- インデックス化の際に系列長（単語数）に応じてパディング。ここでは最も単語数が多い文章を基準に、各文章の末尾に 1（<pad>）を挿入して長さを調節している

なお、先ほど定義した dataset オブジェクト（train / val / test）や Field クラスとの関連は以下のとおりです。



BucketIterator による DataLoader の作成

これまでどおり PyTorch Lightning を使ってネットワークを学習させていきますが、今回は少し詳しく説明したい箇所がいくつかあるので、説明の際にはネイティブの PyTorch の書き方をしていきます。

```
from torchtext.data import BucketIterator

batch_size = 64

train_loader = BucketIterator(train, batch_size=batch_size, shuffle=True)
val_loader = BucketIterator(val, batch_size=batch_size)
test_loader = BucketIterator(test, batch_size=batch_size)
```

ここで、いくつか確認したいポイントがあります。

1. 単語が辞書内の ID に置き換わっている（インデックス化）
2. 各文章の末尾に 1 が挿入されてパディングできている
3. 各ミニバッチのサイズは [num_seq, batch] となる（文章に含む単語の数：num_seq、バッチサイズ：batch）

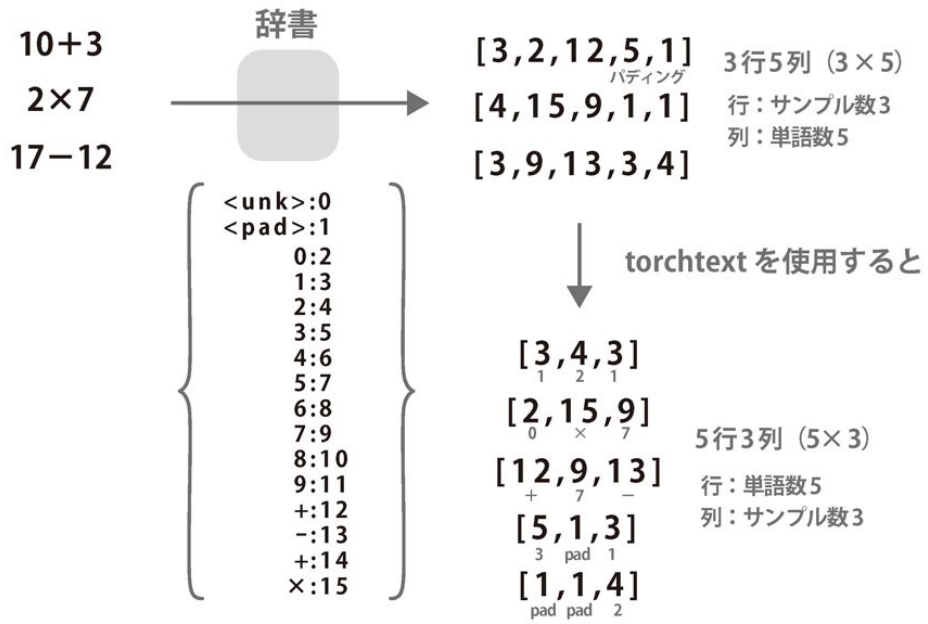
データを表示して確認しましょう。まずはポイント 1 と 2 です。DataLoader からデータを抽出するには、next() 関数と iter() 関数を使います。

```
batch = next(iter(train_loader))
batch.x
```

```
tensor([[   0,  147, 6404,  ...,    0, 2993,  345],
        [   5,    0,    8,  ...,   46,    7,    2],
        [7672,  328, 2586,  ...,    4,  101,   30],
        ...,
        [   1,    1,    1,  ...,    1,    1,    1],
        [   1,    1,    1,  ...,    1,    1,    1],
        [   1,    1,    1,  ...,    1,    1,    1]])
```

続いてポイント 3 です。これまで扱ってきたデータと同じように考えると、入力値のサイズは [batch, num_seq] と考えるのが自然です（文章に含む単語の数：num_seq、バッチサイズ：batch）。

しかし、torchtext を使用するとサイズが縦横逆になるので、注意が必要です。理由としては、たとえば batch.x[0] と指定すると各文章の 1 文字目を抽出でき、複数の文章を同時に扱う際の計算上の利便性が高いからです。




```
# 配列のサイズを確認
batch.x.shape
```

```
torch.Size([69, 64])
```

```
# 目標値
batch.t
```

```
tensor([0, 0, 0, 2, 1, 1, 1, 0, 0, 1, 1, 2, 1, 2, 0, 0, 1, 1, 1, 1, 0, 1, 0, 2,
        1, 1, 2, 2, 0, 2, 2, 2, 1, 0, 0, 0, 1, 1, 2, 2, 1, 2, 1, 0, 1, 0, 2, 1,
        2, 2, 2, 0, 0, 1, 0, 1, 2, 1, 2, 1, 1, 2, 1, 1])
```

そのため、バッチの要素番号 0 を取得すると、全文章の 1 文字目を取得できます。

```
# 1文字目を取得
batch.x[0]
```

```
tensor([   0,  147, 6404,  305,  589,   28,   21,  151, 1977, 2488,   89,  158,
        1156,  697,   31,  107,  615,  626, 4670,   20,    0, 1120,  103, 3681,
         223,  181,   21, 2469,  107,  417,  860,  119, 5727, 2553,  107,  993,
         181,  380, 1378, 1169,  296,   22,  334,   56,   21, 1539,  311, 5433,
```

```
         343, 1718,   21,  497,   37,    0,  782,  181,   21,  495,   51,  180,
        4593,    0, 2993,  345])
```

各層の挙動を確認

ここまでの作業でデータ作成、前処理、ミニバッチ学習の準備が完了しました。続いてネットワークを定義しますが、新しく登場する層があるので、各層の挙動を確認しましょう。

```
# 挙動の確認用に2サンプル取得
x = batch.x[:, :2]
t = batch.t[:2]
x.shape, t.shape
```

```
(torch.Size([69, 2]), torch.Size([2]))
```

```
t
```

```
tensor([0, 0])
```

```
# 縦方向が1の文章を表す
x[:10]
```

```
tensor([[   0,  147],
        [   5,    0],
        [7672,  328],
        [  95,    3],
        [   4,    1],
        [   0,    1],
        [   0,    1],
        [   5,    1],
        [   0,    1],
        [   6,    1]])
```

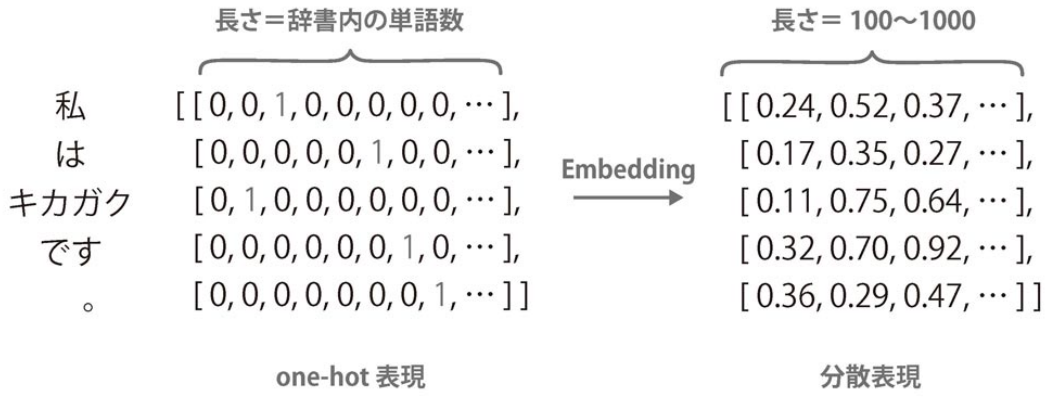
Embedding 層

Embedding 層には入力値の単語を分散表現して特徴抽出する目的があります。変換前の系列データ（入力値）から文脈などの情報を圧縮したベクトルを生成するので、それを特徴ベクトルとして次の層に渡します。まず、分散表現する前の **one-hot 表現**された入力値は長さ数万のベ

クトルのうち、1 個だけ「1」で、残りすべては「0」という状態になっています。

one-hot 表現はシンプルですが、各単語の特徴が表現されておらず、そのままネットワークに入力しても訓練がうまく行われません。

そのため、単語ごとの意味を何らかの形で特徴抽出する必要があります。そこで考案された手法が**分散表現**です。分散表現を行うと、入力値が以下の図のように変換され、one-hot 表現よりも単語の特徴が抽出されたベクトルを得られます。



もう 1 つ重要な事項として、Embedding 層では **sparse 性**を利用した無駄な計算の削減が行われています。sparse は日本語では「疎」と翻訳され、「すかすか」という意味があります。ネットワークの計算上で配列の要素に 0 が多い場合に使用される表現です。

one-hot 表現は一般的には長さ数万のベクトルであり、そのうちの 1 要素だけ「1」で、残りの要素はすべて「0」です。これを**疎なベクトル**と呼び、大量の 0 を計算から省くことで無駄な計算リソースを削減します。Embedding 層の計算時には複数サンプルを同時に入力するため、疎なベクトルが集まった疎な配列を計算します。

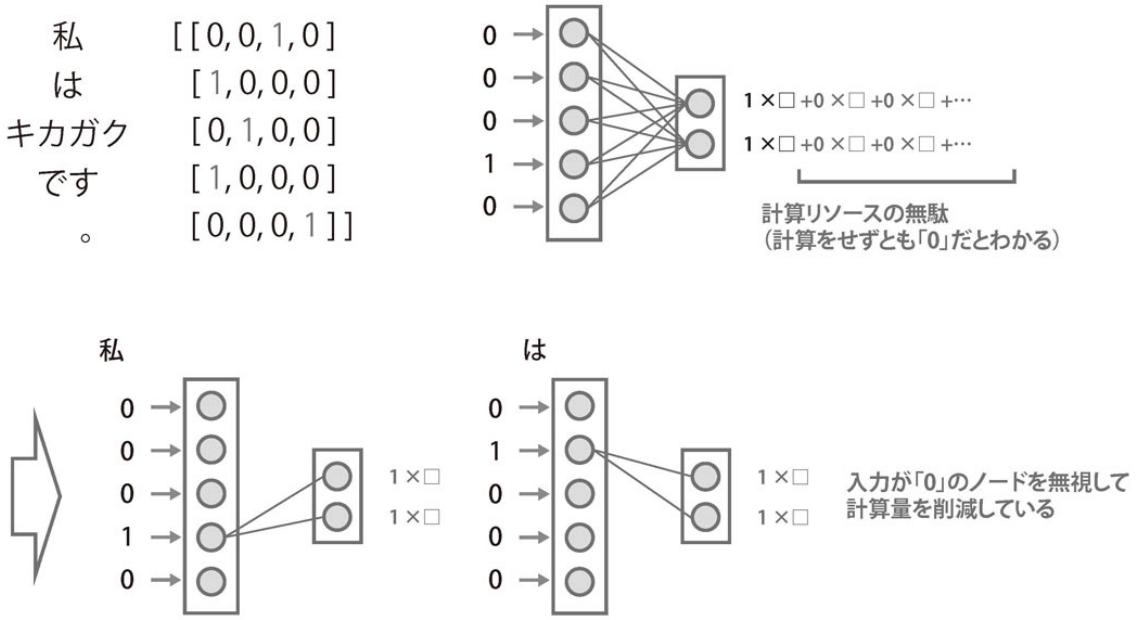


図 8.1　sparse 性を利用した計算量の削減

```
# Embedding層
n_input = len(field_x.vocab)
n_embed = 12

embed = nn.Embedding(n_input, n_embed, padding_idx=1)
x_embeded = embed(x)

# データのサイズを確認
x.shape
x_embeded.shape
```

```
torch.Size([69, 2, 12])
```

ここで padding_id=1 で<pad>に該当する ID を指定している箇所が重要です。これは**マスキング**と呼ばれる処理で、パディングした部分が学習の邪魔にならないようにしています。マスキングによって入力値の<pad>に該当する部分がすべて 0 となって出力されます。文章中では意味を持たない<pad>が学習に影響を与えないようになりました。

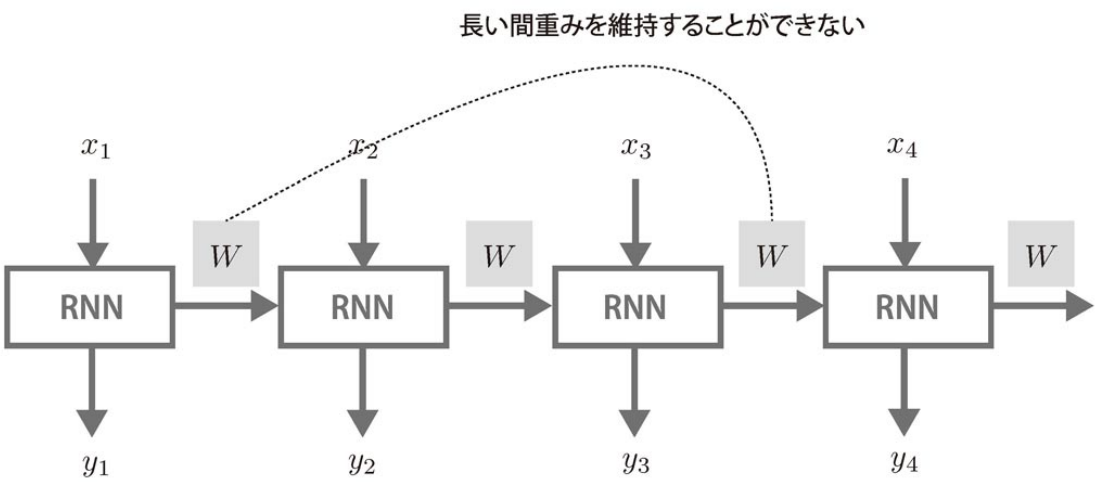
パディングは文章の最後に行うので、実際にデータで確認すると文章の後半に該当する配列の下部が 0 となっていることがわかります。

```
x_embeded
```

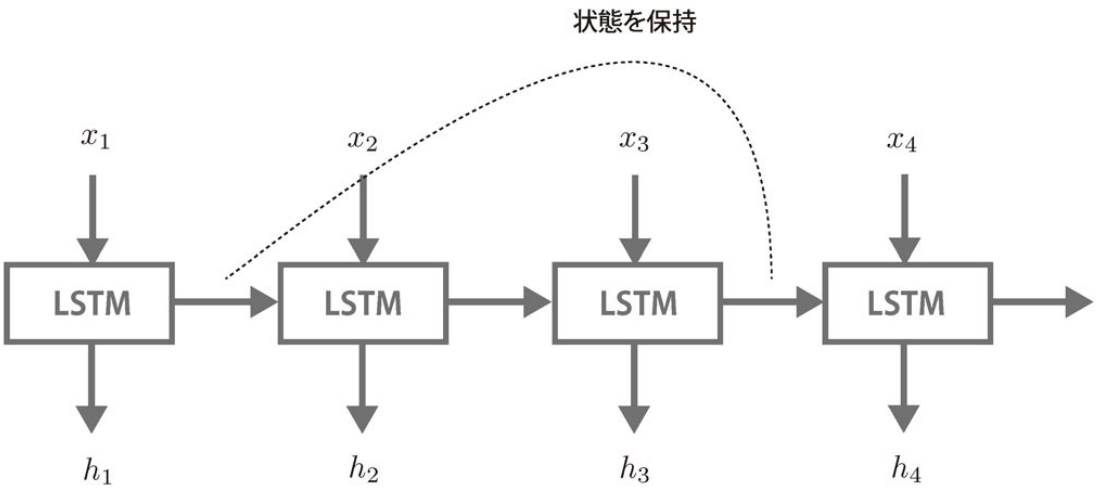
```
tensor([[[-1.2743,  0.5598, -1.1903,  ...,  0.1246, -1.1333, -0.2852],
         [ 0.8297,  2.3569,  0.9023,  ..., -0.5612,  1.3319,  0.7618]],

        [[-1.7170, -0.2622, -0.9636,  ...,  1.1120,  0.4999, -0.3123],
         [-1.2743,  0.5598, -1.1903,  ...,  0.1246, -1.1333, -0.2852]],

        [[ 0.6091, -1.7860,  0.0778,  ...,  0.1569, -2.0107, -0.8861],
         [ 0.6271, -2.6121,  0.7543,  ..., -1.0425,  1.4497, -2.5008]],

        ...,

        [[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]],

        [[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]],

        [[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]]],
       grad_fn=<EmbeddingBackward>)
```

LSTM 層（Long-Short Term Memory）

本節の冒頭でも述べたとおり、今回の問題設定では入力が文章（系列データ）なので、文章中の単語と単語のつながりである文脈を情報として保持して、「私」の後には「は」「を」が来るようなルールをネットワークに訓練させる必要があります。そこで、考案されたのが RNN アルゴリズムでした。



ただし、RNN の弱点として、数ステップ前の情報しか覚えることができず、文章が長くなってしまうと高いパフォーマンスを発揮できなくなることが挙げられます。その弱点を補ったアルゴリズムが、**LSTM**（Long-Short Term Memory）です。名前のとおり長期・短期の両面から情報保有が可能であり、RNN では訓練できなかった長期依存（離れたところに存在する単語からの文脈）も訓練可能になりました。




```
n_hidden = n_embed
n_layers = 1

# LSTM層
lstm = nn.LSTM(n_embed, n_hidden, n_layers)
x_lstm, (h, c) = lstm(x_embeded)

# サイズの確認
x_lstm.shape, h.shape, c.shape
```

```
(torch.Size([69, 2, 12]), torch.Size([1, 2, 12]), torch.Size([1, 2, 12]))
```

ネットワークの定義と学習

それでは新しく登場した層を理解できたところで、PyTorch Lightning を使ってネットワークの定義から学習まで実装していきましょう。データ読み込みの箇所が DataLoader から BucketIterator に変更したところ以外は同じです。

```
class Net(pl.LightningModule):

    def __init__(self, n_input, n_embed, n_hidden, n_layers, n_output):
        super(Net, self).__init__()
        self.embed = nn.Embedding(n_input, n_embed, padding_idx=1)
        self.lstm = nn.LSTM(n_embed, n_hidden, n_layers)
        self.fc = nn.Linear(n_hidden, n_output)

    def forward(self, x):
        x = self.embed(x)
        x, (h, c) = self.lstm(x)
        x = self.fc(h[-1])
        return x

    def training_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
        return loss


    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.01)

# 詳細設定
n_input  = len(field_x.vocab)
n_embed  = 100
n_hidden = 100
n_layers = 3
```

```
n_output = 9

# 学習の実行
pl.seed_everything(0)
net = Net(0)
trainer = pl.Trainer(max_epochs=30)
trainer.fit(net, train_loader, val_loader)
```

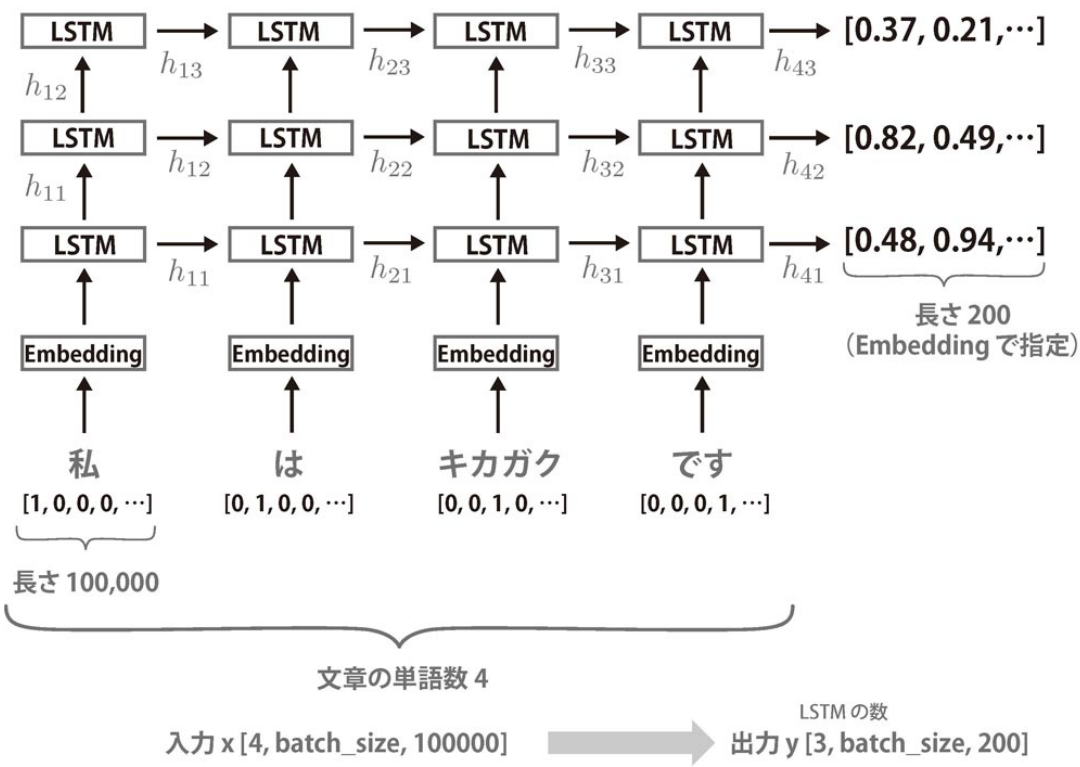
```
{'val_loss': 1.1031286716461182, 'val_acc': 0.3330167233943939}
```

```
# テストデータに対する検証
results = trainer.test(test_loaders=test_loader)
results
```

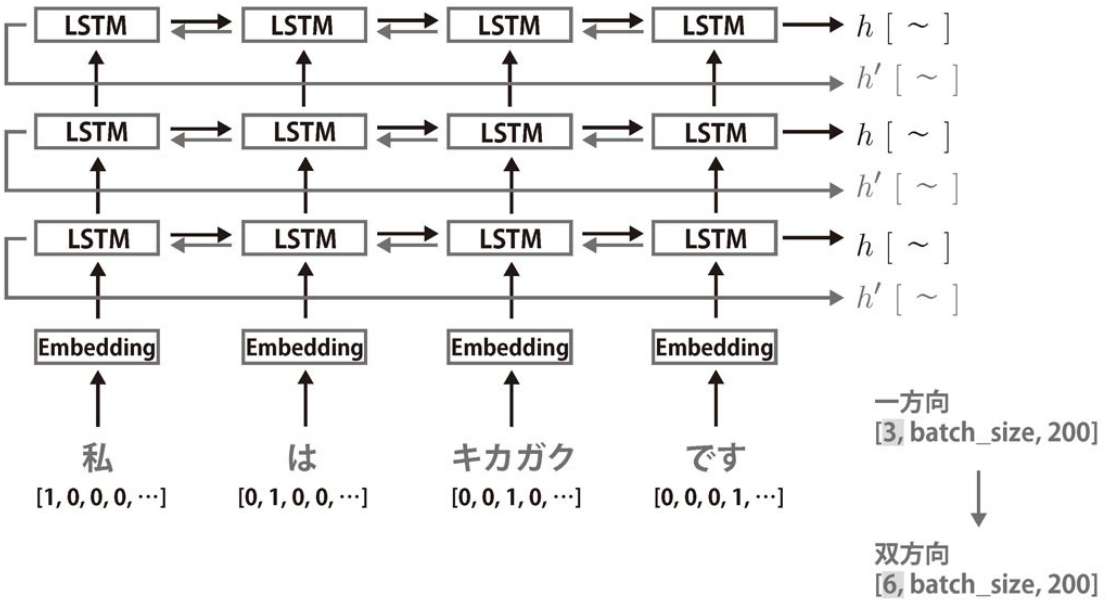
```
{'test_loss': 1.1030422449111938, 'test_acc': 0.3333016633987427}
```

モデル性能の向上

ここまで実装したネットワークの全体像を整理すると、次の図のようになります（一方向のLSTM）。文章を単語ごとに切り分け、語順どおりに入力して文脈などの情報を特徴抽出し、ネットワークを学習しています。



しかし、文章は前から後ろに一方向だけでなく、後ろから前に向けた逆方向の文脈もとることでさらにモデルの性能が向上することがわかっています。それが**双方向 LSTM** と呼ばれるアルゴリズムです。次の図のように文脈の順方向・逆方向の両方から見るので、さらに多くの情報を得られます。今後の実装では双方向 LSTM を使用するので、`bidirectional=True` を指定しましょう。




```
class Net(pl.LightningModule):

    def __init__(self, n_input, n_embed, n_hidden, n_layers, n_output):
        super(Net, self).__init__()
        self.embed = nn.Embedding(n_input, n_embed, padding_idx=1)
        self.lstm = nn.LSTM(n_embed, n_hidden, n_layers, bidirectional=True)
        self.fc = nn.Linear(n_hidden*2, n_output)

    def forward(self, x):
        x = self.embed(x)
        x, (h, c) = self.lstm(x)
        x = self.fc(h[-1])
        return x

    def training_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
```

```
        return loss

    def validation_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
        return loss


    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.01)

# 詳細設定
n_input  = len(field_x.vocab)
n_embed  = 100
n_hidden = 100
n_layers = 3
n_output = 9

# 学習の実行
pl.seed_everything(0)
net = Net(0)
trainer = pl.Trainer(max_epochs=30)
trainer.fit(net, train_loader, val_loader)
```

```
{'val_loss': 1.1818653345108032, 'val_acc': 0.578172504901886}
```

```
# テストデータに対する検証
results = trainer.test(test_loaders=test_loader)
results
```

```
{'test_loss': 1.1792887449264526, 'test_acc': 0.5792173147201538}
```

テストデータに対する予測精度が 33% から 58% に上がり、モデルの性能向上を確認できました。受験や筆記試験などでは前後の文脈を意識し、試験の問いに対して考えたりするので、その意味では人間の感覚と似ているところもあるのかもしれません。双方向からの情報を組み込むと精度が上がりました。こういったテクニックも覚えておきましょう。

8.8 文章生成

文書分類、文章分類の次は、文章生成を実装していきます。文章生成モデルは現在でもたくさんの研究がなされ、新しいモデルが出てきていますが、最も有名なモデルの 1 つが、**Seq2Seq**（Seqence to Sequence）です。ある「系列データ」を入力して、別の「系列データ」を出力する（置き換える）ルールを学習するモデルです。

系列データの代表格と言えば、**文章や音声（系列：語順、文脈）**です。この入力を日本語、出力を英語にした場合は**機械翻訳**、入力を質問、出力を回答とした場合は**チャットボット**と呼ばれるものになります。私たちの生活においても非常に馴染み深いものです。

文章生成のデータセット作成

データセットを作成するところから実装していきます。今回は足し算のデータセットを作成していきましょう。実際の文章を適用する際のデータセットの形の参考になるため、どのような形式であるかをしっかり見ておきましょう。手順としては以下のとおりです。

1. 1～9999 のランダムな整数を用意
2. 整数と記号「+」を文字列として結合
3. リストをデータフレーム形式に変換して、CSV データとして保存

ランダムな整数を用意

ランダムな数値の生成には、NumPy の numpy.random モジュールを使います。

```
import numpy as np

np.random.seed(0)

values = np.random.randint(1, 10000, (10, 2))
```

```
values
```

```
array([[2733, 9846],
       [3265, 4860],
       [9226, 7892],
       [4374, 5875],
       [6745, 3469],
       [ 706, 2600],
       [2223, 7769],
       [2898, 9894],
       [ 538, 6217],
       [6922, 6037]])
```

整数と記号「+」を文字列として結合

整数と記号「+」の記号を用意できたので、文字列として結合しましょう。

```
value_1, value_2 = values[0]
print(value_1, value_2)
```

```
2733 9846
```

```
# 足し算の文字列を作成
src = []
for (value_1, value_2) in values:
    symbol = '+'
    src_ = str(value_1) + symbol + str(value_2)
    src.append(src_)

src
```

```
['2733+9846',
 '3265+4860',
 '9226+7892',
 '4374+5875',
 '6745+3469',
 '706+2600',
 '2223+7769',
 '2898+9894',
 '538+6217',
 '6922+6037']
```

また、eval() 関数を使用することで文字列の計算を実行できます。

```
add_value = eval(src[0])
add_value
```

```
12579
```

```
# 足し算の結果を格納
add_value = []

for src_ in src:
    add_value_ = eval(src_)
    add_value.append(add_value_)

# 結果を表示
add_value
```

```
[12579, 8125, 17118, 10249, 10214, 3306, 9992, 12792, 6755, 12959]
```

CSV 形式でデータを保存

前処理として PyTorch ライブラリの torchtext を使用しますが、これは CSV からデータの読み込みを行うため、Pandas の DataFrame 形式に変換して CSV として保存しましょう。

```
import pandas as pd

# データフレームを定義
df = pd.DataFrame({'src': src, 'add_value': add_value})

df.head()
```

```
         src  add_value
0  2733+9846      12579
1  3265+4860       8125
2  9226+7892      17118
3  4374+5875      10249
4  6745+3469      10214
```

```
# データをCSV形式で保存
df.to_csv('sample.csv', header=None, index=None)
```

一連の処理を関数化する

一連の処理がわかったので、関数化して train / val / test データをまとめて作成します。

ここでとても重要な論点があります。単純に np.random.randint(1, 10000) で整数を生成してしまうと **4 桁同士**が全データの 80% を占める不均衡なデータとなってしまい、不適切です。そこで、各桁数のデータが均等になるように整数の生成を工夫しましょう。

```
# 再現性を確保
np.random.seed(0)

# データセット作成関数
def generate_dataset(filename, n_sample):
    srcs, add_values = [], []
    n_sample = int(n_sample / 5)

    symb = '+'

    # 4桁
    values = np.random.randint(1, 10000, (n_sample*2, 2))
    for (value_1, value_2) in values:
        src = str(value_1) + symb + str(value_2)
        add_value = '{:.0f}'.format(eval(src))
        srcs.append(src)
        add_values.append(add_value)

    # 3桁
    values = np.random.randint(1, 1000, (n_sample, 2))
    for (value_1, value_2) in values:
        src = str(value_1) + symb + str(value_2)
        add_value = '{:.0f}'.format(eval(src))
        srcs.append(src)
        add_values.append(add_value)

    # 2桁
    values = np.random.randint(1, 100, (n_sample, 2))
    for (value_1, value_2) in values:
        src = str(value_1) + symb + str(value_2)
        add_value = '{:.0f}'.format(eval(src))
        srcs.append(src)
        add_values.append(add_value)

    # 1桁
```

```
    values = np.random.randint(1, 10, (n_sample, 2))
    for (value_1, value_2) in values:
        src = str(value_1) + symb + str(value_2)
        add_value = '{:.0f}'.format(eval(src))
        srcs.append(src)
        add_values.append(add_value)

    df = pd.DataFrame({'src': srcs, 'add_value': add_values})
    df.to_csv(filename, header=None, index=None)
    return df

# 学習データ：10万サンプル、検証データ：3万サンプル、テストデータ：3万サンプル
df_train = generate_dataset('data/seq2seq/train.csv', 100000)
df_val = generate_dataset('data/seq2seq/val.csv', 30000)
df_test = generate_dataset('data/seq2seq/test.csv', 30000)

df_train.head()
```

```
         src add_value
0  2733+9846     12579
1  3265+4860      8125
2  9226+7892     17118
3  4374+5875     10249
4  6745+3469     10214
```

```
df_val.head()
```

```
         src add_value
0  2182+2121      4303
1   637+3614      4251
2  5218+2993      8211
3  5482+3946      9428
4  6587+6965     13552
```

```
df_test.head()
```

```
         src add_value
0   3884+372      4256
1  3180+8049     11229
2  3892+7262     11154
```

```
3   8145+8395      16540
4    8054+761       8815
```

足し算の式が入力値、足し算の結果が目標値となるようなデータセットを作成できました。それでは実際に Seq2Seq で足し算モデルを構築していきましょう。

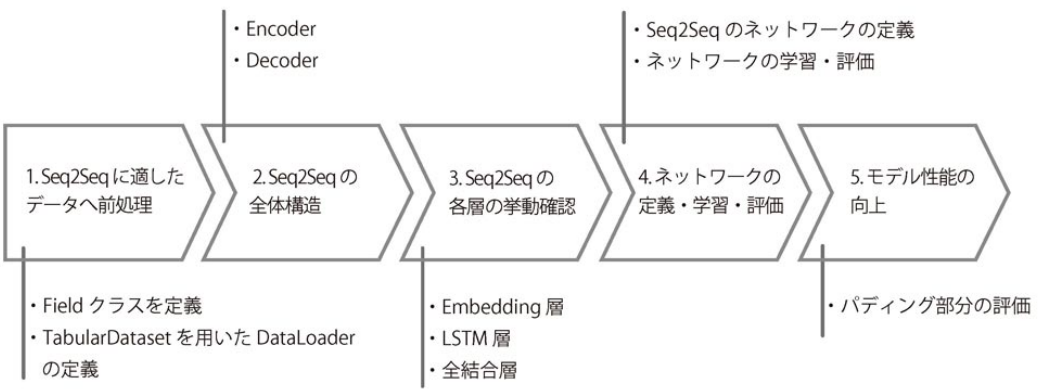
固定長と可変長

もう一度おさらいしておくと、Seq2Seq は「系列データ」を入力として、別の「系列データ」を出力するルールを学習するモデルです。今回であれば、足し算のルールを学習していきます。Seq2Seq がこれまでに紹介した機械学習の手法と大きく異なるのは、入力・出力ともに**可変長**である点です。

そのため、可変長の入力から可変長の出力を得るような機械学習において、どのようにその問題を解決しているのか意識して取り組んでいきましょう。

文章生成の流れ

基礎的な Seq2Seq の流れの理解を目的に、足し算を行うネットワークを学習していきます。



Seq2Seq に適したデータへ前処理

本節の文章生成では、12+3 という文字列を入力すると 15 という文字列が返ってくるような**足し算**をテーマとします。機械学習で足し算の規則性を再現できるのかという問題設定です。本節の冒頭でも述べたように、ここで用意するデータセットによって機械翻訳やチャットボットなどさまざまな用途に活用できます。

まずは作成したデータセットを確認しましょう。1 桁から 4 桁までの足し算をデータセットとして用意しました。data/seq2seq 内にある train.csvval.csvtest.csv を使います。

```
import pandas as pd

# データの読み込み
df_train = pd.read_csv('data/seq2seq/train.csv', header=None)
df_val = pd.read_csv('data/seq2seq/val.csv', header=None)
df_test = pd.read_csv('data/seq2seq/test.csv', header=None)

# データの長さ確認
len(df_train), len(df_val), len(df_test)
```

```
(100000, 30000, 30000)
```

それでは、作成したデータに必要な前処理を行っていきましょう。今回も文書分類で利用した torchtext を使います。繰り返しになりますが、torchtext で最初に覚えておきたいモジュールは 3 つです。

- Field
- TabularDataset
- BucketIterator

```
from torchtext.data import Field, TabularDataset, BucketIterator
```

1 文字ずつ切り分ける関数を定義

文章分類までは日本語を単語別に切り分ける必要があったので、MeCab を利用して形態素解析に取り組みました。今回の演算では数値・記号を 1 文字ずつに切り分ければよいので、for 文で単純に 1 文字ずつ抽出するだけで済みます。

```
# 切り分ける関数を定義
def tokenize(text):
    result = []
    for tok in text:
        result.append(tok)
    return result

tokenize('100+999')
```

```
['1', '0', '0', '+', '9', '9', '9']
```

また、一般に多く利用される記述方法として**リスト内包表記**というものがあるので、あわせて覚えておきましょう。コードを簡潔に記述できます。

```
# リスト内包表記
def tokenize(text):
    return [tok for tok in text]

tokenize('100+999')
```

```
['1', '0', '0', '+', '9', '9', '9']
```

Field クラスを定義

torchtext では、必要な前処理を Field クラスに定義します。その後、TabularDataset を使用すると、train / val / test それぞれのデータに対して一括で前処理を行うことができます。

- field_x：入力値（例：10 + 20）
- field_t：目標値（例：30）

オプションの設定については、「torchtext を用いた辞書作成」項を参照してください。

```
# 足し算の文字列
field_x = Field(
    tokenize = tokenize
)
```

日本語から英語に翻訳するように、Seq2Seq では文章から文章への変換を行います。その際、変換後の文章（目標値）について**文章の最初・最後**を明示する情報が必要となります。まずは Field クラスに以下のオプションを指定します。

- **文章の最初**：<sos>
- **文章の最後**：<eos>

```
# 足し算の正解データ
field_t = Field(
    tokenize = tokenize,
    init_token = '<sos>',
    eos_token = '<eos>'
)
```

CSV ファイルの読み込み＋ 1 文字ずつ切り分け

Field クラスを定義できたので、TabularDataset を用いて読み込んだ CSV データに前処理を行います。

```
# データの前処理
train, val, test = TabularDataset.splits(
    path = 'data/seq2seq/',
    train = 'train.csv',
    validation = 'val.csv',
    test = 'test.csv',
    format = 'csv',
    fields = [('x', field_x), ('t', field_t)]
)
```

データ作成で指定したサンプル数どおりになっているか len() メソッドで確認します。

```
# サンプル数の確認
len(train), len(val), len(test)
```

```
(100000, 30000, 30000)
```

指定した前処理が適切に行われているか確認しましょう。TabularDataset で作成したデータセットの内部を確認する際には vars() を使用します。

```
train[0]
```

```
<torchtext.data.example.Example at 0x11eff9cd0>
```

```
# 1文字ずつ切り分けがされていることを確認
vars(train[0])
```

```
{'x': ['2', '7', '3', '3', '+', '9', '8', '4', '6'],
 't': ['1', '2', '5', '7', '9']}
```

辞書を作成し、単語に番号を振る

「torchtext を用いた辞書作成」と同じように処理していきます。

```
# 辞書作成、ID取得
field_x.build_vocab(train)
field_t.build_vocab(train)

# stoi：単語 -> インデックス
field_x.vocab.stoi
```

```
defaultdict(<bound method Vocab._default_unk_index of <torchtext.vocab.Vocab object at 0x130ba8d⇨
10>>,
            {'<unk>': 0,
             '<pad>': 1,
             '+': 2,
             '9': 3,
             '2': 4,
             '6': 5,
             '5': 6,
             '3': 7,
             '7': 8,
             '4': 9,
             '8': 10,
             '1': 11,
             '0': 12})
```

```
field_t.vocab.stoi
```

```
defaultdict(<bound method Vocab._default_unk_index of <torchtext.vocab.Vocab object at 0x130ba8c⇨
d0>>,
            {'<unk>': 0,
             '<pad>': 1,
             '<sos>': 2,
             '<eos>': 3,
             '1': 4,
             '9': 5,
             '8': 6,
             '5': 7,
             '4': 8,
             '6': 9,
             '2': 10,
             '7': 11,
             '3': 12,
             '0': 13})
```

辞書に含まれる単語数を確認するには、.vocab と len() を組み合わせます。ネットワークの入力・出力サイズを指定する際に必要です。

```
# 単語数を確認
len(field_x.vocab), len(field_t.vocab)
```

```
(13, 14)
```

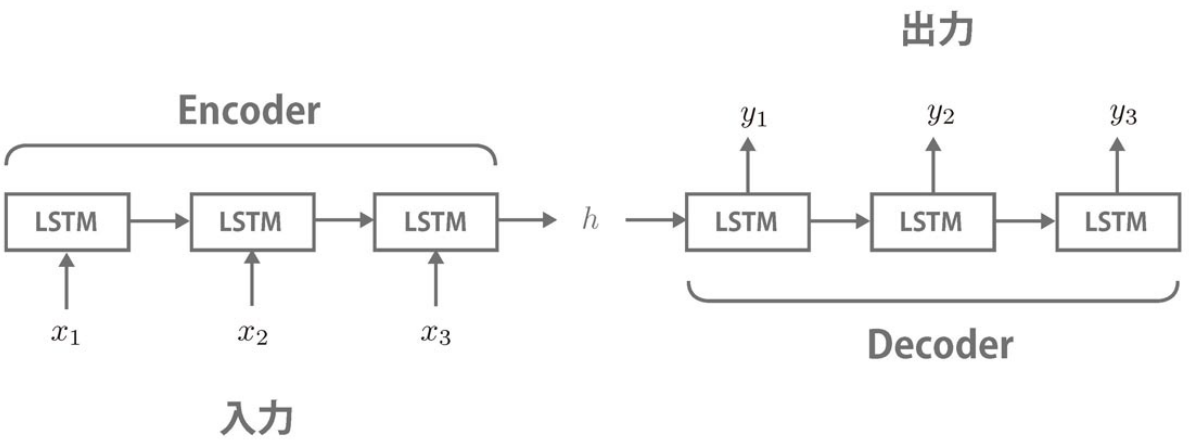
Seq2Seq の全体構造（Encoder と Decoder）

データの処理が終わったのでネットワークの定義に進んでいきますが、その前に今回使用する Seq2Seq のモデル構造について説明しておきます。Seq2Seq は次のように、別々の機能を持った 2 つのネットワークから成り立っています。

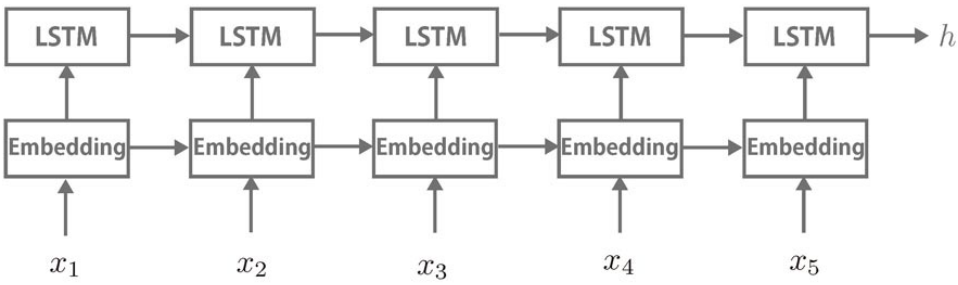
- **Encoder**：変換前の系列データを入力すると、文脈などの情報を圧縮したベクトルを出力する
- **Decoder**：Encoder から状態ベクトルを受け取り、変換後の系列データを出力する

機械翻訳で例えると、Encoder に日本語の文章を入力すると、Decoder から英語の文章が出力されることになります。今回の問題設定の演算では、以下のように考えます。

- **Encoder への入力**：10+25 などの計算前の文字列
- **Decoder からの出力**：計算結果である 25



Encoder に系列データを入力すると、文脈などの情報が圧縮された**隠れ状態ベクトル** h が出力されます。内部は前述の Embedding 層と LSTM 層の 2 層から成り立っており、次のような構造になっています。



Decoder は 3 つの層から形成されていて、Encoder と同じ点・異なる点は以下のとおりです。

層の構成

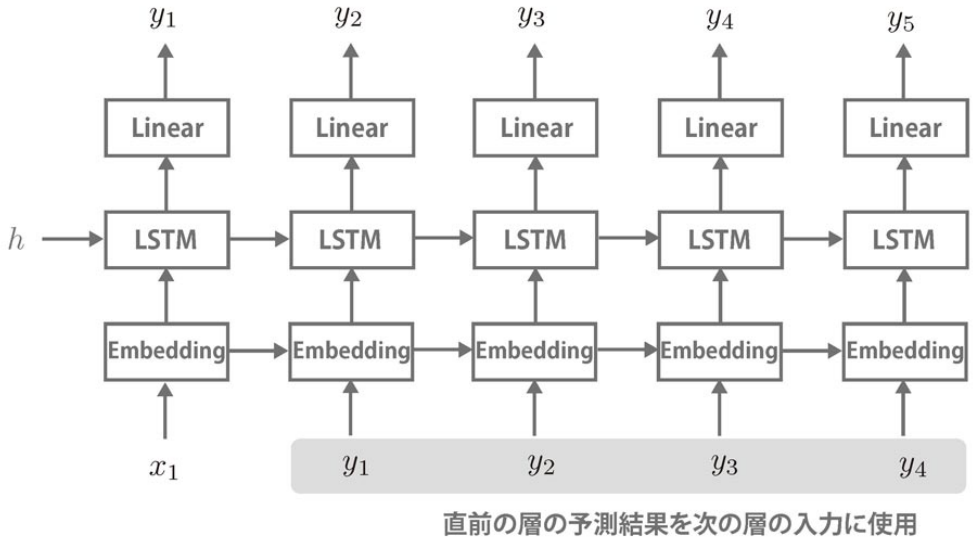
1. Embedding 層
2. LSTM 層
3. 全結合層

同じ点

- Embedding 層で単語を分散表現する
- LSTM 層で系列データの情報を圧縮して隠れ状態ベクトル h を得る

異なる点

- 隠れ状態ベクトル h を全結合層に入力し、予測結果を出力する
- Encoder では文章を一度にすべて入力していたが、Decoder では直前の予測結果を次の入力として逐次計算を行う



Seq2Seq の各層の挙動を確認

Encoder と Decoder を個々に実行して、計算によって配列のサイズがどのように変化するのか、またどういった情報が受け渡されているのかを把握しましょう。全体像を再整理すると、Seq2Seq は以下のような構造になっています。

Encoder

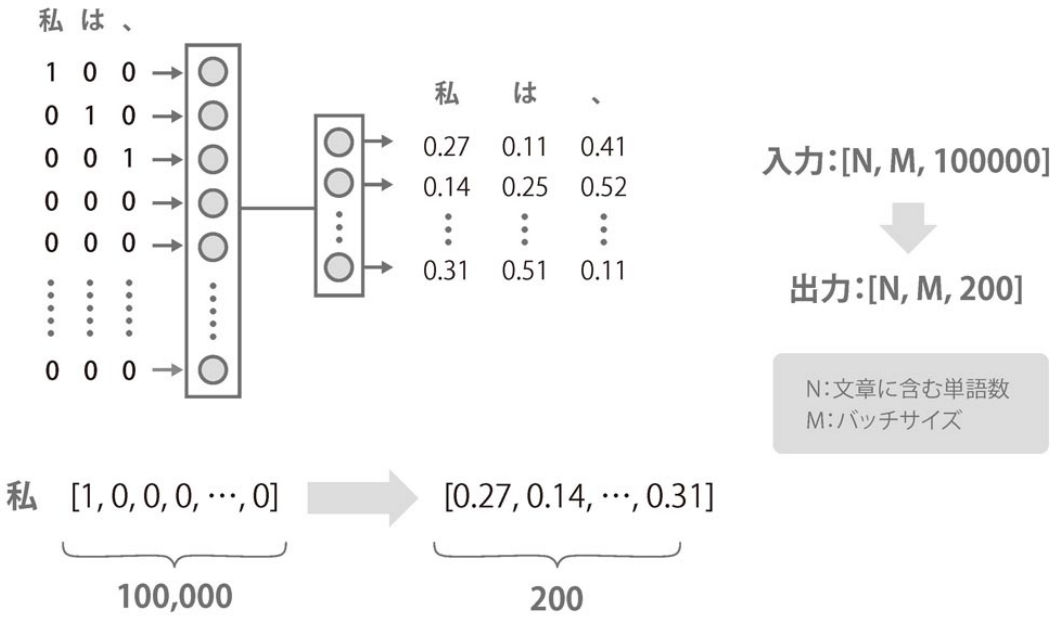
1. **Embedding 層**：単語の分散表現
2. **LSTM 層**：情報が圧縮された隠れ状態ベクトル h の出力

Decoder

3. Embedding 層
4. LSTM 層
5. 全結合層 ⇒ 出力結果を 3. の入力に利用

Encoder の Embedding 層

Embedding 層は前述したとおり単語を分散表現する役割があります。層への入力・出力の前後で、次の図のようなサイズの変化があります。



流れを確認するために BuckketIterator を定義します。

```
# DataLoaderの定義
batch_size = 512

train_loader = BucketIterator(train, batch_size=batch_size, shuffle=True)
val_loader = BucketIterator(val, batch_size=batch_size)
test_loader = BucketIterator(test, batch_size=batch_size)

# ミニバッチで読み込み
batch= next(iter(train_loader))

# 入力値
batch.x
```

```
tensor([[ 5, 10,  3,  ...,  6,  7,  8],
        [ 4,  2,  4,  ...,  8,  4,  2],
        [ 7, 10,  2,  ...,  7,  3,  9],
        ...,
        [ 8,  1,  1,  ...,  7,  4,  1],
        [ 1,  1,  1,  ...,  1, 10,  1],
        [ 1,  1,  1,  ...,  1,  4,  1]])
```

```
# 目標値
batch.t
```

```
tensor([[ 2,  2,  2,  ...,  2,  2,  2],
        [ 9,  4,  6,  ...,  5,  9,  7],
        [10,  9,  7,  ...,  9,  7,  4],
        ...,
        [13,  1,  3,  ...,  3, 10,  1],
        [ 3,  1,  1,  ...,  1,  3,  1],
        [ 1,  1,  1,  ...,  1,  1,  1]])
```

```
x = batch.x

# データのサイズ
x.shape
```

```
torch.Size([9, 512])
```

```
# 辞書内の単語数
len(field_x.vocab)
```

```
13
```

```
# データを確認
x
```

```
tensor([[ 5, 10,  3,  ...,  6,  7,  8],
        [ 4,  2,  4,  ...,  8,  4,  2],
        [ 7, 10,  2,  ...,  7,  3,  9],
        ...,
        [ 8,  1,  1,  ...,  7,  4,  1],
        [ 1,  1,  1,  ...,  1, 10,  1],
        [ 1,  1,  1,  ...,  1,  4,  1]])
```

```
n_input = len(field_x.vocab)
n_embed = 200
```

ここで Embedding 層を定義します。padding_idx=1 でマスキング処理をしています。

```
# Embedding層を定義
embed_enc = nn.Embedding(n_input, n_embed, padding_idx=1)
x_embeded = embed_enc(x)

x_embeded
```

```
tensor([[[-1.0274, -1.0866,  0.3783,  ..., -0.5288,  1.0809,  1.8299],
         [-1.6290, -0.7269,  0.2110,  ...,  1.1558, -2.0312, -1.0113],
         [-0.3184, -0.2595, -1.1316,  ..., -0.1448,  0.7104, -0.3751],
         ...,
         [ 1.3020, -1.7019,  0.1444,  ..., -0.8378,  0.7859,  1.1478],
         [ 0.6869, -0.3798, -0.5273,  ...,  0.1517,  0.4511,  0.1281],
         [-1.4781,  0.7007, -0.5237,  ..., -1.2467,  1.7170, -0.8431]],

        [[-0.1452, -0.4367, -1.7958,  ..., -0.1570,  0.8093, -0.4009],
         [ 0.5616, -0.7961,  1.9696,  ..., -0.1933,  0.7531, -0.6534],
         [-0.1452, -0.4367, -1.7958,  ..., -0.1570,  0.8093, -0.4009],
         ...,
         [-1.4781,  0.7007, -0.5237,  ..., -1.2467,  1.7170, -0.8431],
         [-0.1452, -0.4367, -1.7958,  ..., -0.1570,  0.8093, -0.4009],
         [ 0.5616, -0.7961,  1.9696,  ..., -0.1933,  0.7531, -0.6534]],

        [[ 0.6869, -0.3798, -0.5273,  ...,  0.1517,  0.4511,  0.1281],
         [-1.6290, -0.7269,  0.2110,  ...,  1.1558, -2.0312, -1.0113],
         [ 0.5616, -0.7961,  1.9696,  ..., -0.1933,  0.7531, -0.6534],
         ...,
         [ 0.6869, -0.3798, -0.5273,  ...,  0.1517,  0.4511,  0.1281],
         [-0.3184, -0.2595, -1.1316,  ..., -0.1448,  0.7104, -0.3751],
         [-1.6129,  0.7949,  0.2768,  ..., -1.0037,  0.5434, -1.1154]],

        ...,

        [[-1.4781,  0.7007, -0.5237,  ..., -1.2467,  1.7170, -0.8431],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         ...,
         [ 0.6869, -0.3798, -0.5273,  ...,  0.1517,  0.4511,  0.1281],
         [-0.1452, -0.4367, -1.7958,  ..., -0.1570,  0.8093, -0.4009],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]],
```

```
        [[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         ...,
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [-1.6290, -0.7269,  0.2110,  ...,  1.1558, -2.0312, -1.0113],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]],

        [[ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         ...,
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000],
         [-0.1452, -0.4367, -1.7958,  ..., -0.1570,  0.8093, -0.4009],
         [ 0.0000,  0.0000,  0.0000,  ...,  0.0000,  0.0000,  0.0000]]],
       grad_fn=<EmbeddingBackward>)
```

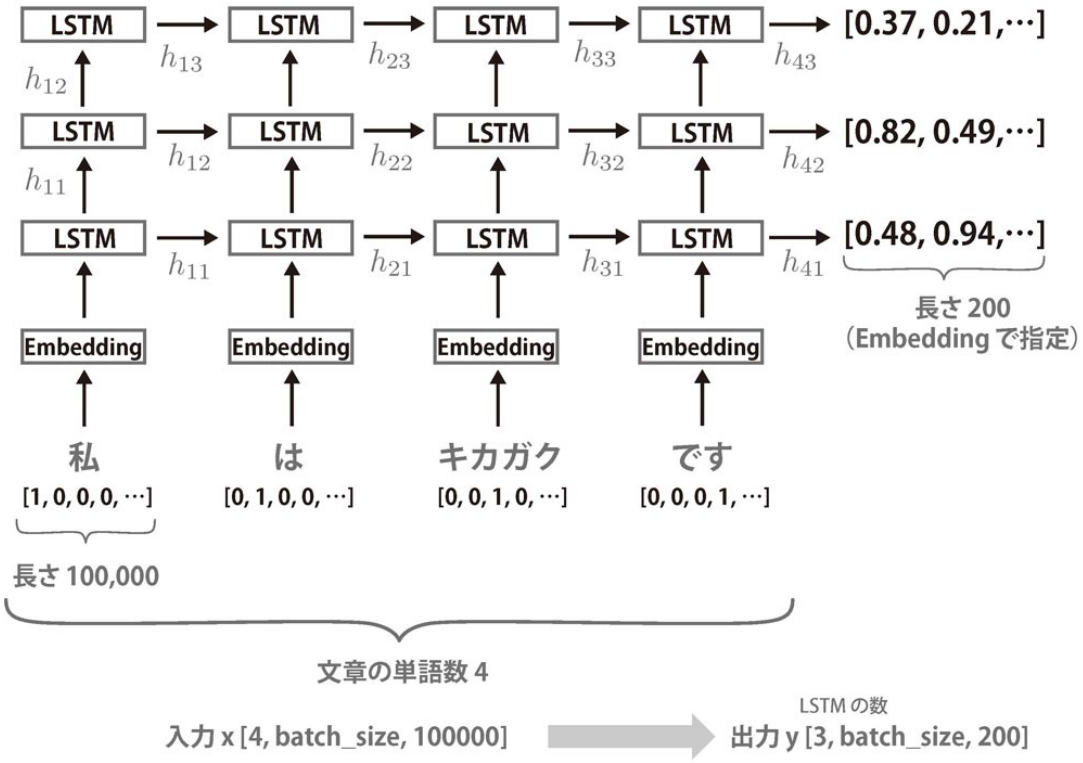
Embedding 層の処理が済んだので、処理前後のサイズ変化を確認します。

```
# サイズ変化を確認
print(x.shape)
print(x_embeded.shape)
```

```
torch.Size([9, 512])
torch.Size([9, 512, 200])
```

Encoder の LSTM 層

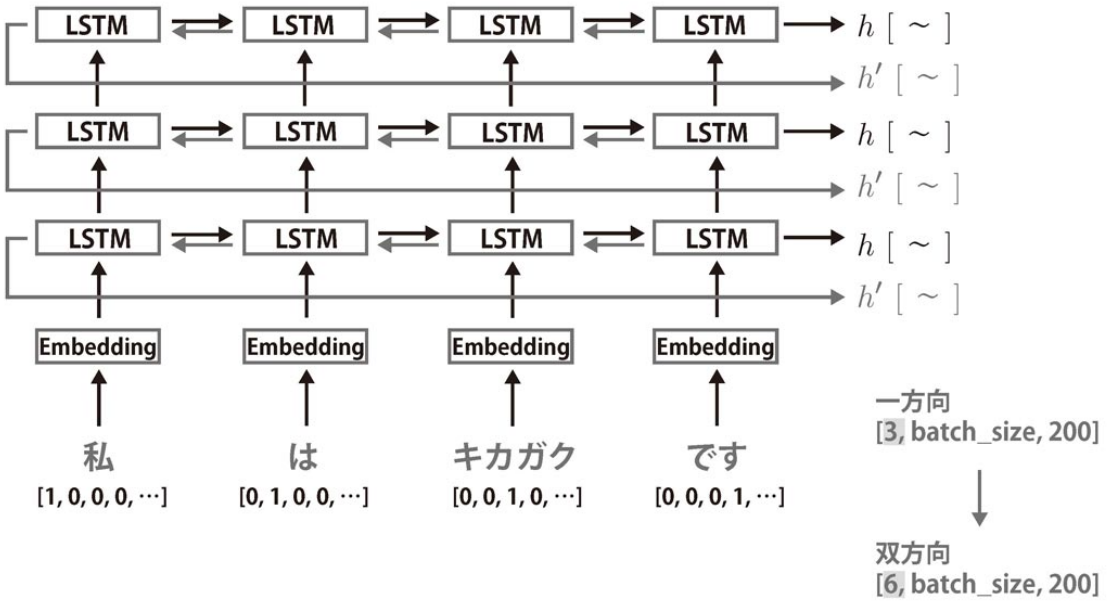
LSTM 層の役割は、変換前の系列データを入力すると文脈などの情報が圧縮された隠れ状態ベクトル h を出力することです。これにより、「私」の後には「は」「を」が来るようなルールをネットワークに学習させることができます。Embedding 層とあわせた全体像として、前掲の一方向の LSTM の処理が行われます。




```
# LSTM層の定義
n_hidden = 200
n_layers = 3

lstm_enc = nn.LSTM(n_embed, n_hidden, n_layers)
x_lstm, (h, c) = lstm_enc(x_embeded)

# サイズの変化を確認
print(x_embeded.shape)
print(x_lstm.shape, h.shape)
```

```
torch.Size([9, 512, 200])
torch.Size([9, 512, 200]) torch.Size([3, 512, 200])
```

また、LSTM には双方向 LSTM と呼ばれるものもあり、次の図のように文脈の順方向・逆方向の両方から見るのでさらに多くの情報を得ることができるのでした。




```
# bidirecrion: True
lstm_enc = nn.LSTM(n_embed, n_hidden, n_layers, bidirectional=True)
x_lstm, (h, c) = lstm_enc(x_embeded)

# サイズの変化を確認
print(x_embeded.shape)
print(x_lstm.shape, h.shape)
```

```
torch.Size([9, 512, 200])
torch.Size([9, 512, 400]) torch.Size([6, 512, 200])
```

`bidirectional = False`（一方向 LSTM）の場合は隠れ状態ベクトル h のサイズが `[3, 512, 200]` であったのに対して、`bidirectional = True`（双方向 LSTM）では `[6, 512, 200]` となったので、順方向・逆方向でサイズが 2 倍になったことがわかります。

以上で Encoder の各層の挙動を確認できました。

Encoder から Decoder への流れ

続いて、得られた隠れ状態ベクトルを Decoder に入力して、演算された後の出力データを得るまでの流れを確認しましょう。

```
# 目標値tを取得
t = batch.t
t
```

```
tensor([[ 2,  2,  2,  ...,  2,  2,  2],
        [ 9,  4,  6,  ...,  5,  9,  7],
        [10,  9,  7,  ...,  9,  7,  4],
        ...,
        [13,  1,  3,  ...,  3, 10,  1],
        [ 3,  1,  1,  ...,  1,  3,  1],
        [ 1,  1,  1,  ...,  1,  1,  1]])
```

x と同様に目標値 t のサイズも [num_seq, batch] となっています。

```
# サイズの確認
t.shape
```

```
torch.Size([7, 512])
```

Decoder に最初に入力するデータは直前の予測結果が存在しないので、手動で用意する必要があります。以下で目標値 t から 1 単語目を抽出しましょう。

```
# 1単語目を抽出
input = t[0, :]
input
```

```
tensor([2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
```

```
        2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
        2, 2, 2, 2, 2, 2, 2])
```

抽出されたデータを見ると、すべて 2 となっています。これは文章の最初であることを示す<sos>を表しています。

```
# itos：インデックス => 単語
field_t.vocab.itos[2]
```

```
'<sos>'
```

ここで注意が必要なのですが、抽出したデータのサイズを.shape で確認すると、1 次元のベクトルになっています。これは各サンプルから 1 単語目を取り出したために起きる現象です。

```
input.shape
```

```
torch.Size([512])
```

この状態で Embedding を行うと、出力される配列のサイズが [batch, n_embed] となります。しかし、LSTM の仕様として [num_seq, batch, n_embed] のように 1 文章に含まれる単語数が必要です。そこで、Embedding 層に入力する前に unsqueeze() を用いて単語数の情報 num_seq を追加し、エラーが起こらないように処理しておく必要があります。

```
# 次元を追加
input = input.unsqueeze(0)

# 1文章あたりの単語数が追加された
input.shape
```

```
torch.Size([1, 512])
```

Decoder の Embedding 層

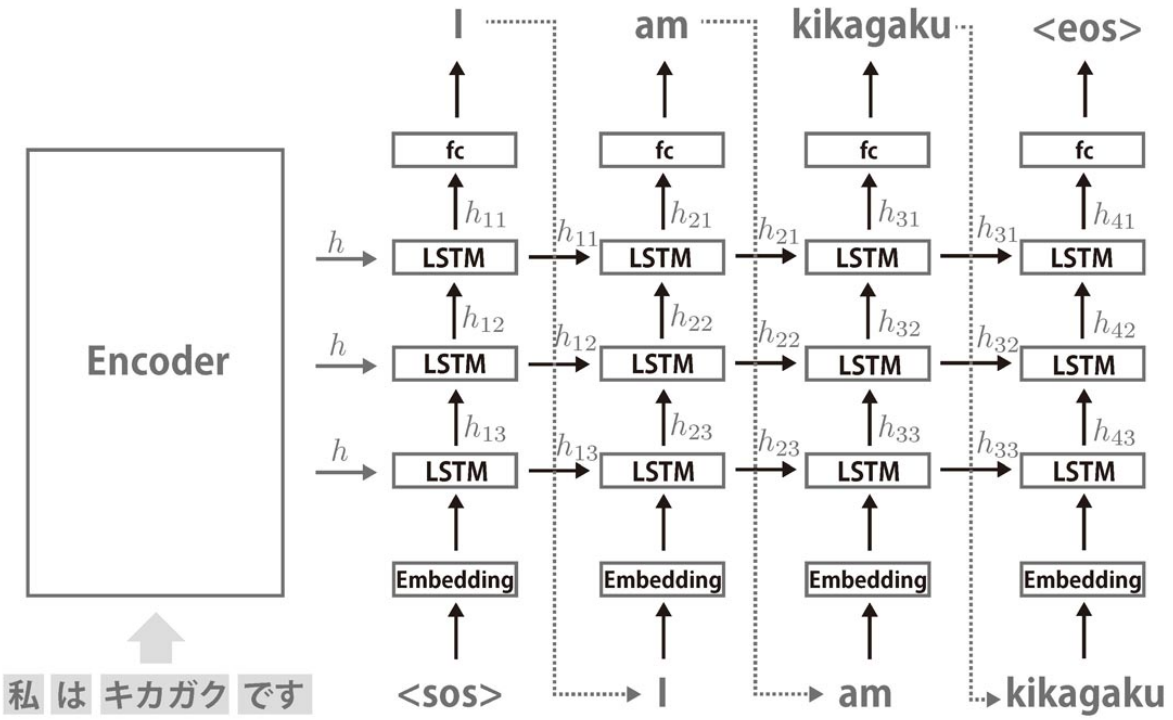
Embedding 層に関しては Encoder で実装したものと同じく、単語を分散表現する役割があります。

```
# DecoderのEmbedding層の定義
n_input = len(field_t.vocab)
n_embed = 200

embed_dec = nn.Embedding(n_input, n_embed, padding_idx=1)
t_embeded = embed_dec(input)

# サイズの変化を確認
print(input.shape)
print(t_embeded.shape)
```

```
torch.Size([1, 512])
torch.Size([1, 512, 200])
```

Decoder の LSTM 層

ここから説明する LSTM と全結合層は一体となって機能するため、改めて Decoder 全体の働きを 1 つの図にまとめて説明します。



1. 手動で用意した<sos>が Embedding 層～LSTM 層～全結合層を経て予測結果が出力される
2. 予測結果と、LSTM 層から出力される隠れ状態ベクトル h が次の予測を行う
3. 直前の結果が次の予測の入力になる伝播を繰り返して、文章全体が出力される

目標値 t から 1 単語ずつ取り出して入力するため、num_seq=1 となります。よって入力値のサイズは [1, batch, n_embed] です。

```
t_embeded.shape
```

```
torch.Size([1, 512, 200])
```

```
# DecoderのLSTM層を定義
n_hidden = 200
n_layers = 3

lstm_dec = nn.LSTM(n_embed, n_hidden, n_layers, bidirectional=True)
t_lstm, (h, c) = lstm_dec(t_embeded, (h, c))

print(t_lstm.shape)
print(h.shape)
```

```
torch.Size([1, 512, 400])
torch.Size([6, 512, 200])
```

Decoder の全結合層

最後に全結合層を確認しましょう。

```
n_output = len(field_t.vocab)
n_output
```

```
14
```

```
# 全結合層の定義
fc = nn.Linear(n_hidden, n_output)
y = fc(h[-1])

print(h[-1].shape)
print(y.shape)
print(y)
```

```
torch.Size([512, 200])
torch.Size([512, 14])
tensor([[-0.0496, -0.0695,  0.0664,  ...,  0.0561,  0.0072,  0.0392],
        [-0.0501, -0.0679,  0.0608,  ...,  0.0599,  0.0039,  0.0342],
        [-0.0476, -0.0687,  0.0652,  ...,  0.0611,  0.0084,  0.0355],
        ...,
        [-0.0446, -0.0699,  0.0646,  ...,  0.0594,  0.0115,  0.0364],
        [-0.0473, -0.0711,  0.0570,  ...,  0.0583,  0.0144,  0.0379],
        [-0.0551, -0.0646,  0.0621,  ...,  0.0649,  0.0082,  0.0411]],
       grad_fn=<AddmmBackward>)
```

```
# Softmax関数で0～1に収める
y_softmax = F.softmax(y, dim=1)
y_softmax
```

```
tensor([[0.0675, 0.0662, 0.0758,  ..., 0.0750, 0.0714, 0.0738],
        [0.0674, 0.0662, 0.0753,  ..., 0.0752, 0.0711, 0.0733],
        [0.0676, 0.0662, 0.0756,  ..., 0.0753, 0.0715, 0.0734],
        ...,
        [0.0677, 0.0660, 0.0755,  ..., 0.0752, 0.0716, 0.0734],
        [0.0676, 0.0660, 0.0750,  ..., 0.0751, 0.0719, 0.0736],
        [0.0671, 0.0664, 0.0754,  ..., 0.0756, 0.0715, 0.0738]],
       grad_fn=<SoftmaxBackward>)
```

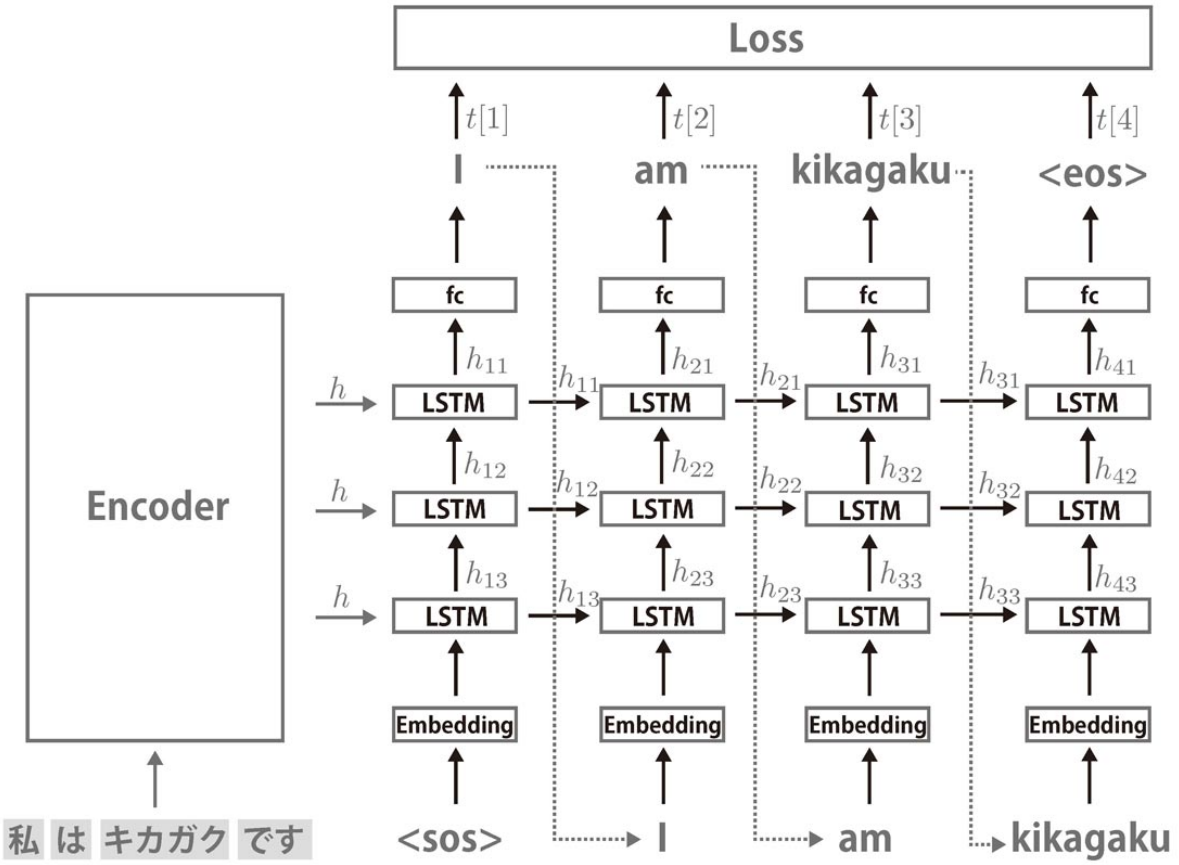
```
# 結果の確認：行方向で最も大きな値のインデックスをとる
result = y_softmax.max(1)[1]
result
```

```
tensor([ 2,  2,  2,  2, 11, 11, 11,  2, 11,  2,  2,  2,  2,  2,  2,  2, 11,  2,
         2, 11, 11,  2,  2, 11,  2,  2, 11,  2, 11, 11, 11,  2, 11,  2,  2, 11,
        11, 11,  2, 11, 11,  2,  2,  2,  2,  2, 11,  2, 11, 11,  2, 11,  2, 11,
         2, 11, 11,  2,  2,  2,  2, 11, 11, 11, 11, 11, 11, 11,  2, 11, 11,  2,
        11,  2, 11,  2,  2, 11,  2,  2, 11, 11, 11, 11,  2, 11,  2, 11,  2,  2,
         2, 11,  2, 11,  2, 11,  2,  2,  2, 11, 11, 11,  2,  2, 11,  2,  2, 11,
        11,  2, 11, 11,  2,  2, 11,  2,  2,  2, 11,  2, 11, 11, 11, 11,  2,  2,
         2,  2,  2,  2, 11,  2,  2,  2, 11, 11, 11,  2, 11,  2, 11,  2,  2, 11,
        11, 11,  2, 11, 11, 11, 11,  2,  2, 11,  2, 11,  2, 11,  2,  2, 11,  2,
        11, 11,  2,  2,  2, 11, 11,  2, 11, 11, 11,  2,  2, 11, 11, 11,  2,  2,
        11,  2, 11, 11,  2,  2, 11, 11,  2, 11, 11, 11, 11, 11, 11, 11,  2,  2,
         2, 11, 11,  2,  2,  2,  2,  2,  2,  2,  2,  2, 11, 11,  2,  2, 11,  2,
        11, 11,  2,  2,  2,  2, 11,  2, 11,  2,  2,  2,  2, 11,  2, 11, 11, 11,
         2, 11, 11,  2, 11,  2, 11, 11, 11, 11,  2,  2,  2, 11,  2,  2,  2,  2,
         2,  2,  2, 11,  2, 11,  2,  2, 11, 11,  2,  2,  2, 11,  2, 11, 11,  2,
        11, 11,  2, 11,  2, 11,  2, 11,  2, 11, 11, 11, 11, 11, 11, 11, 11,  2,
         2,  2,  2, 11, 11,  2, 11, 11, 11, 11, 11,  2,  2,  2, 11, 11,  2,  2,
        11, 11, 11,  2, 11,  2,  2,  2, 11,  2, 11,  2,  2,  2, 11,  2,  2, 11,
        11,  2,  2,  2,  2,  2, 11, 11,  2,  2, 11, 11,  2, 11,  2, 11, 11,  2,
         2,  2,  2,  2, 11, 11, 11, 11,  2, 11, 11, 11,  2, 11,  2, 11, 11, 11,
         2,  2,  2, 11,  2, 11,  2, 11, 11,  2,  2, 11,  2, 11, 11,  2,  2,  2,
         2, 11, 11, 11, 11,  2,  2,  2, 11, 11,  2,  2,  2, 11,  2,  2,  2, 11,
         2,  2, 11,  2, 11,  2,  2, 11, 11,  2, 11,  2,  2,  2, 11,  2,  2, 11,
        11,  2, 11,  2,  2, 11,  2, 11,  2, 11,  2, 11, 11, 11,  2,  2,  2,  2,
        11,  2, 11, 11,  2, 11, 11, 11,  2,  2,  2,  2,  2,  2,  2, 11, 11,  2,
         2,  2,  2,  2, 11, 11, 11,  2, 11, 11, 11, 11,  2, 11,  2, 11, 11, 11,
         2,  2,  2,  2, 11,  2, 11, 11, 11, 11,  2, 11, 11, 11, 11,  2, 11,  2,
         2,  2, 11,  2,  2, 11,  2, 11,  2, 11, 11,  2, 11, 11,  2, 11, 11,  2,
        11,  2,  2, 11,  2,  2, 11, 11])
```

これで Encoder と Decoder それぞれの動作を確認できました。続いて結果から損失関数を計算しましょう。

損失関数の計算

予測結果が出力された後は損失関数を計算します。次の図のように、Decoder から出力された結果 1 つ 1 つを目標値 t と比較して損失関数が計算されます。



損失関数を計算する際には、Seq2Seq の予測値 y と目標値 t を用意し、nn.CrossEntropyLoss() で算出します。ここで損失関数に入れる前に Softmax 関数を定義しないのは、Softmax 関数が PyTorch では損失関数の計算の中に包括されているためです。なお、1 文字目 t[0] を入力した結果として、y の予測結果は 2 文字目であるので、用意する目標値は 2 文字目の t[1] となります。

```
# 損失の計算
criterion = nn.CrossEntropyLoss()
loss = criterion(y, t[1])
loss
```

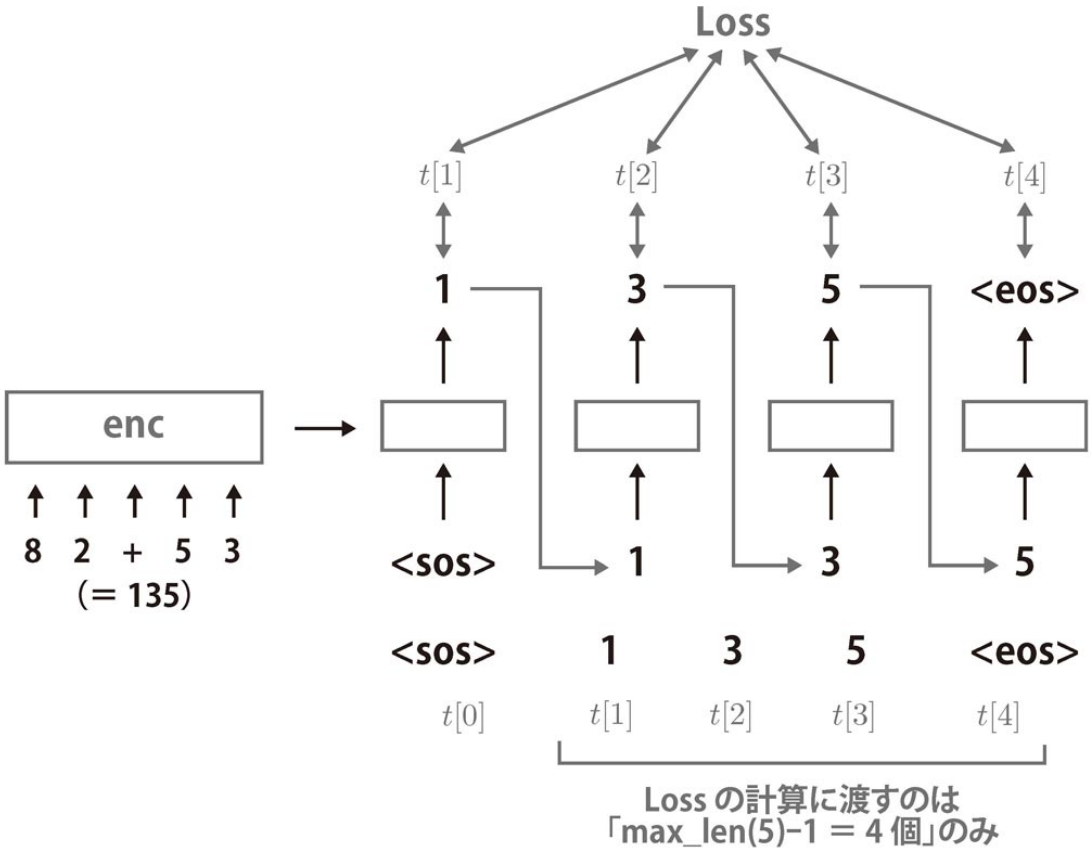
```
tensor(2.6457, grad_fn=<NllLossBackward>)
```

Seq2Seq のネットワークの定義と学習

Encoder、Decoder それぞれの挙動を確認してきました。それでは実際に 2 つのネットワークをつなぎ合わせて Seq2Seq を実装します。

```
# Encoderの定義
class Encoder(pl.LightningModule):

    def __init__(self, n_input, n_embed, n_hidden, n_layers):
        super(Encoder, self).__init__()
        self.n_layers = n_layers
        self.embed = nn.Embedding(n_input, n_embed, padding_idx=1)
        self.lstm = nn.LSTM(n_embed, n_hidden, n_layers, bidirectional=True)

    def forward(self, x):
        x = self.embed(x)
        x, (h, c) = self.lstm(x)
        return h, c

# Decoderの定義
class Decoder(pl.LightningModule):

    def __init__(self, n_output, n_embed, n_hidden, n_layers):
        super(Decoder, self).__init__()
        self.output_dim = n_output
        self.embed = nn.Embedding(n_output, n_embed, padding_idx=1)
        self.lstm = nn.LSTM(n_embed, n_hidden, n_layers, bidirectional=True)
        self.fc = nn.Linear(n_hidden, n_output)

    def forward(self, x, h, c):
        x = x.unsqueeze(0)
        x = self.embed(x)
        x, (h, c) = self.lstm(x, (h, c))
        y = self.fc(h[-1])
        return y, h, c
```

次のプログラムのように実装する際に注意してほしいのは、for t_ in range(max_len - 1)において、max_len（目標値 t のサイズ）から -1 している点です。以下の図のように<sos>を損失関数に渡さないようプログラムする必要があるのが理由です。




```
class Net(pl.LightningModule):

    def __init__(self, *args):
        super(Net, self).__init__()
        self.encoder = Encoder(n_input, n_embed_enc, n_hidden, n_layers)
        self.decoder = Decoder(n_output, n_embed_dec, n_hidden, n_layers)

    def forward(self, x, t):
        max_len, batch_size = t.shape
        t_vocab_size = self.decoder.output_dim
        y = torch.zeros(max_len - 1, batch_size, t_vocab_size)
        h, c = self.encoder(x)
        inputs = t[0, :]

        # <sos>  を損失関数に渡さない
        for i in range(max_len - 1):
            _y, h, c = self.decoder(inputs, h, c)
            y[i] = _y
            top1 = _y.max(1)[1]
            inputs = top1
```

```
        # conversion for evaluation
        y = y.view(-1, y.shape[-1])
        t = t[1:].view(-1)

        return y, t

    def training_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters(), lr=0.01)

n_input = len(field_x.vocab)
n_output = len(field_t.vocab)

# Embeding 後の次元数
n_embed_enc = 200
n_embed_dec = 200

# LSTM層の定義
n_hidden = 200
n_layers = 3

# 学習の実行
pl.seed_everything(0)
net = Net(n_input, n_embed_enc, n_embed_dec, n_hidden, n_layers)
trainer = pl.Trainer(max_epochs=30)
```

```
trainer.fit(net, train_loader, val_loader)
```

```
{'val_loss': 0.2951650023460388, 'val_acc': 0.8881276249885559}
```

```
# テストデータに対する検証
results = trainer.test(test_loaders=test_loader)
results
```

```
{'test_loss': 0.29532063007354736, 'test_acc': 0.8887067437171936}
```

結果を確認すると、問題単位の正解率が 89% となっており、高い精度のモデルを作成できました。

本章では、文書分類から文章分類、文章生成まで紹介しました。

第9章 デプロイ

本章では、Azure のサービスを活用してクラウド環境とローカル環境を連携しながら、機械学習の推論フェーズに重要な「デプロイ」を紹介します。多くの書籍や教育カリキュラムでは効率の良い学習方法や最新手法の紹介はありますが、実現場では推論環境を構築することが大切です。しかしその情報は少なく、独学でドキュメントを読み解いていくというのも初学者には敷居が高いのも現実です。ここでは IaaS と PaaS を使用した方法を学んでいきましょう。

9.1 Azure Blob Storage

本題に進む前に、学習済みモデルとデータセットが必要です。デプロイに使用するモデルの学習と **Azure Blob Storage** へのアップロードを行っていきます。Azure Blob Storage は、テキストデータやバイナリデータなどの大量の非構造化データを格納するために最適化されています。非構造化データとは、特定のデータモデルや定義に従っていないデータです。公式サイトでは、次の用途に適していると提案しています。

- 画像またはドキュメントをブラウザに直接配信する
- 分散アクセス用にファイルを格納する
- ビデオおよびオーディオをストリーミング配信する
- ログ・ファイルに書き込む
- バックアップと復元、リカバリー、アーカイブのためのデータを格納する
- オンプレミスサービスまたは Azure ホストサービスで分析するデータを格納する

ユーザーまたはアプリケーションは、世界のどこからでも Azure Blob Storage 内のオブジェクトにアクセスできます。どのようなファイル形式でも保存できる Azure Blob Storage に学習した `model` と `Network` を送ることで、複数の開発環境からモデルにアクセスできるようになります。

全体の流れは以下のとおりです。

1. Iris データセットでネットワークを学習
2. MNIST データセットでネットワークを学習
3. Azure Blob Storage に学習済みモデルを保存

それでは進んでいきましょう。

9.2 Irisデータでネットワークを学習

以前に登場した Iris データを、データの読み込み～使用できるデータセットの形式への変換まで行っておきましょう。

```
import torch
import torch.nn as nn
import torch.nn.functional as F

from sklearn.datasets import load_iris
from torch.utils.data import DataLoader, TensorDataset, random_split

# Irisデータセットの読み込み
x, t = load_iris(return_X_y=True)

# PyTorchで学習に使用できる形式へ変換
x = torch.tensor(x, dtype=torch.float32)
t = torch.tensor(t, dtype=torch.int64)

# 入力値と目標値をまとめて、1つのオブジェクトdatasetに変換
dataset = TensorDataset(x, t)

# 各データセットのサンプル数を決定
n_train = int(len(dataset) * 0.6)
n_val = int(len(dataset) * 0.2)
n_test = len(dataset) - n_train - n_val

# ランダムに分割を行うため、シードを固定して再現性を確保
torch.manual_seed(0)

# データセットの分割
train, val, test = random_split(dataset, [n_train, n_val, n_test])
batch_size = 32
```

```
# Data Loaderを用意
train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True, drop_last=True)
val_loader = torch.utils.data.DataLoader(val, batch_size)
test_loader = torch.utils.data.DataLoader(test, batch_size)
```

学習手順とネットワークの定義

ここまではネットワークの定義を Jupyter Notebook を前提に示してきましたが、ここからは推論でもネットワークを使用することを意識して、ネットワークを定義するクラスを外部ファイルに書き込んで保存することにします。

リスト 9.1　models/iris.py

```
# coding: utf-8
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
import pytorch_lightning as pl
from pytorch_lightning import Trainer

import numpy as np
from sklearn.datasets import load_iris

class Net(pl.LightningModule):

  def __init__(self):
    super().__init__()

    self.bn = nn.BatchNorm1d(4)
    self.fc1 = nn.Linear(4, 4)
    self.fc2 = nn.Linear(4, 3)

  def forward(self, x):
    h = self.bn(x)
    h = self.fc1(h)
    h = F.relu(h)
    h = self.fc2(h)
    return h

    def training_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
        return loss
```

```
    def validation_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
        return loss

  def configure_optimizers(self):
    optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
    return optimizer
```

```
import pytorch_lightning as pl
from pytorch_lightning import Trainer
```

このように、外部ファイルにネットワークを定義した場合には、普段使用しているライブラリと同じように import すれば定義した変数・関数を利用できます。今回は Net と呼ばれるネットワークで定義したので、以下のように読み込みます。

```
# 外部ファイルからの読み込み
from models.iris import Net

# 学習の実行
pl.seed_everything(0)
net = Net()
trainer = pl.Trainer(max_epochs=30)
trainer.fit(net, train_loader, val_loader)

# 評価する関数
def callback_metrics(trainer):
    print('Validation Score:', trainer.callback_metrics)
    trainer.test()
    print('Test Score:', trainer.callback_metrics)

callback_metrics(trainer)
```

```
Validation Score: {'val_loss': 0.15639089047908783, 'val_acc': 0.9666666984558105}
Test Score: {'test_loss': 0.1219145655632019, 'test_acc': 1.0}
```

学習済みモデルの保存

学習が完了したので、内部の重みをファイルに保存しましょう。先ほど保存したネットワークを定義する.py ファイルと、保存した重みを読み込むことで、どの環境でも今回学習したネットワークを再現できます。

```
# 重み保存
torch.save(net.state_dict(), 'models/iris.pt')
```

9.3 MNIST でネットワークの学習

続いては MNIST を用いてネットワークを学習していきます。前処理が少し変更になるだけで、全体の手順は同じです。第 6 章を思い返しながら実装していきましょう。

```
from torchvision import transforms, datasets

# 前処理
transform = transforms.Compose([
        transforms.ToTensor()
])

# データセットの取得（データがない場合はダウンロード）
train_val = datasets.MNIST(root='data', train=True, download=True, transform=transform)
test = datasets.MNIST(root='data', train=False, download=True, transform=transform)

# train : val = 0.8 : 0.2
n_train = int(len(train_val) * 0.8)
n_val = len(train_val) - n_train

# ランダムに分割を行うため、シードを固定して再現性を確保
torch.manual_seed(0)

# trainとvalを分割
train, val = torch.utils.data.random_split(train_val, [n_train, n_val])
batch_size = 32

# Data Loaderを用意
```

```
train_loader = torch.utils.data.DataLoader(train, batch_size, shuffle=True, drop_last=True)
val_loader = torch.utils.data.DataLoader(val, batch_size)
test_loader = torch.utils.data.DataLoader(test, batch_size)
```

ネットワークの定義

MNIST についても Iris と同様に推論に利用することを見越して、外部ファイルでネットワークを定義します。

リスト 9.2　models/mnist.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
from torchvision import transforms, datasets
from torch.utils.data import DataLoader
import pytorch_lightning as pl
from pytorch_lightning import Trainer

class Net(pl.LightningModule):

    def __init__(self):
        super().__init__()

        # 畳み込み層を定義
        self.conv = nn.Conv2d(in_channels=1, out_channels=3, kernel_size=(3, 3))
        # 全結合層を定義
        self.fc = nn.Linear(507, 10)

        self.train_acc = pl.metrics.Accuracy()
        self.val_acc = pl.metrics.Accuracy()
        self.test_acc = pl.metrics.Accuracy()

    def forward(self, x):
        # 畳み込み
        h = self.conv(x)
        # 最大値プーリング
        h = F.max_pool2d(h, kernel_size=(2, 2), stride=2)
        # ReLU 関数
        h = F.relu(h)
        # ベクトル化
        h = h.view(-1, 507)
        # 線形変換
        h = self.fc(h)
        return h

    def training_step(self, batch, batch_idx):
```

```
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('train_loss', loss, on_step=True, on_epoch=True)
        self.log('train_acc', self.train_acc(y, t), on_step=True, on_epoch=True)
        return loss

    def validation_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('val_loss', loss, on_step=False, on_epoch=True)
        self.log('val_acc', self.val_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def test_step(self, batch, batch_idx):
        x, t = batch
        y = self(x)
        loss = F.cross_entropy(y, t)
        self.log('test_loss', loss, on_step=False, on_epoch=True)
        self.log('test_acc', self.test_acc(y, t), on_step=False, on_epoch=True)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.SGD(self.parameters(), lr=0.01)
        return optimizer
```

```
# 外部ファイルからの読み込み
from models.mnist import Net

pl.seed_everything(0)
net = Net()
trainer = pl.Trainer(max_epochs=10, gpus=1)
trainer.fit(net, train_loader, val_loader)

# テストデータで検証
results = trainer.test(test_dataloaders=test_loader)

# 学習済みモデルの評価
callback_metrics(trainer)
```

```
Validation Score: {'val_loss': 0.17958417534828186, 'val_acc': 0.947265625}

Test Score: {'test_loss': 0.1784995198249817, 'test_acc': 0.947265625}
```

学習済みモデルの保存

以下のように学習済みモデルをファイルに保存しておきます。

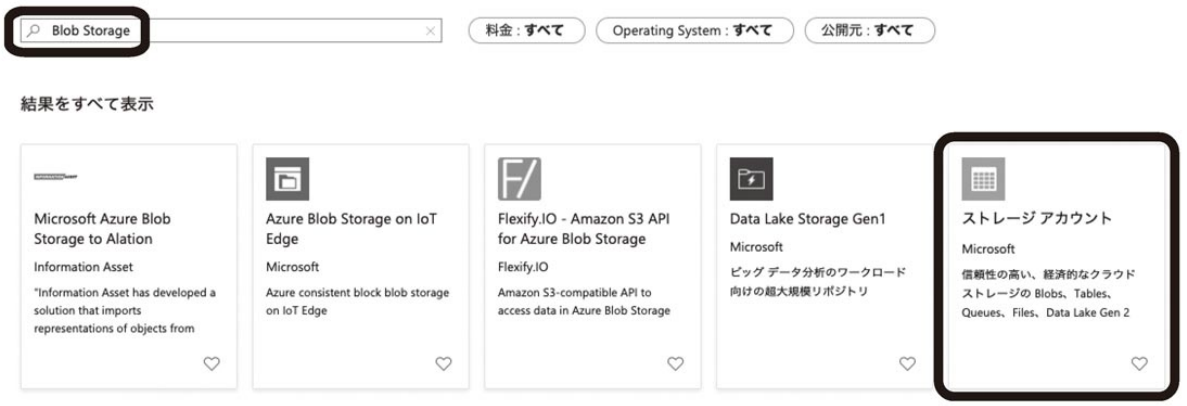
```
# 学習済みモデルの保存
torch.save(net.state_dict(), 'models/mnist.pt')
```

9.4　Azure Blob Storage モデルを保存

Iris と MNIST それぞれのデータセットについてネットワークの学習が完了し、ネットワークの定義と学習済みの重みを保存できました。このネットワークの定義・重みを他の環境からも使用できるように、リモート環境に保存しておきます。このリモート環境が Azure Blob Storage です。最適化されたモデル構造や重みファイルをリモート環境に保存しておくことで、チーム内で気軽に共有できます。複数の VM 間での情報共有など、幅広く使えるので、活用していきましょう。

ストレージアカウントの作成

それでは Azure ポータル上からストレージアカウントを作成しましょう。図の手順に沿って進行していきます。

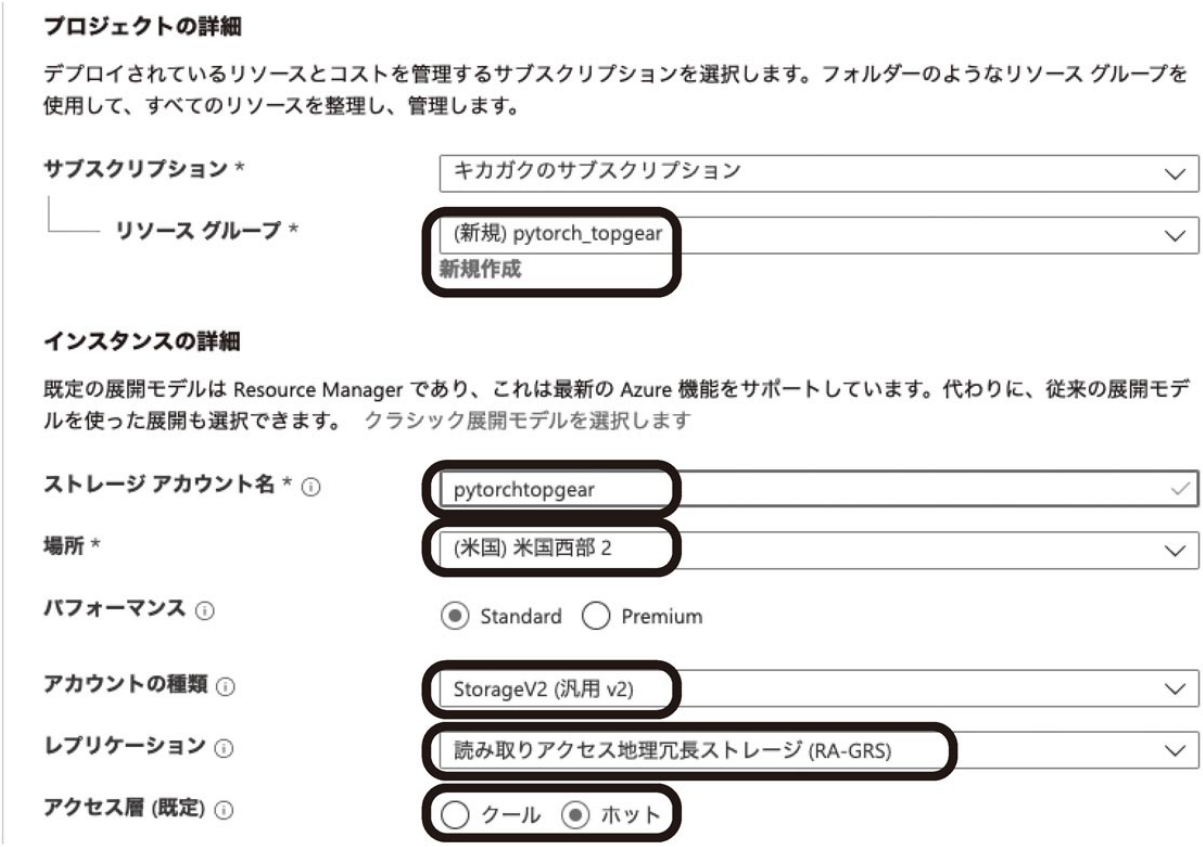


詳細設定に入ります。基本事項はこれまでの Azure 製品と似ていますが、ストレージアカウントの種類は Standard の場合、

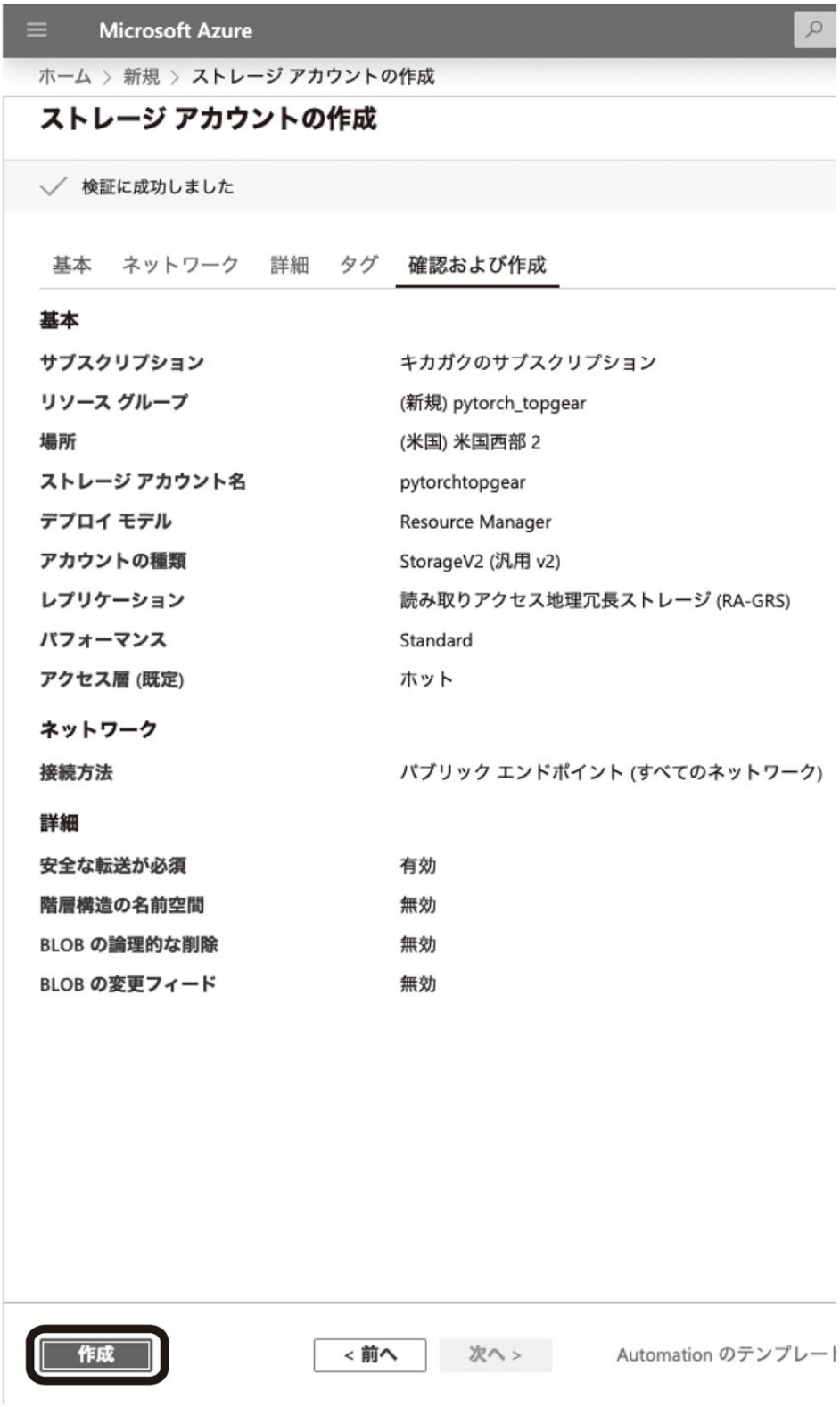
- 汎用 v1 アカウント
- 汎用 v2 アカウント
- BlobStorage

があり、Microsoft 社は汎用 v2 アカウントを推奨しています。最新の Azure Storage 機能をサポートしており、汎用 v1 と BlobStorage のすべての機能が組み込まれています。GB（ギガバイト）あたりの単価では、汎用 v1 よりやや安いこともありますので、経済的にも有利に働きます。最初に汎用 v1 などを選択された方も、ワンクリックで簡単にアップグレードできるので、試してみてください。

アクセス層は、頻繁にアクセスするか、アクセスする回数が少ないかで設定されるのですが、今回はデフォルトの「ホット」で進めておきます。



それでは「確認および作成」をクリックし、デプロイに進行してください。



無事に完了したら以下のような画面になるので、「リソースに移動」をクリックしましょう。

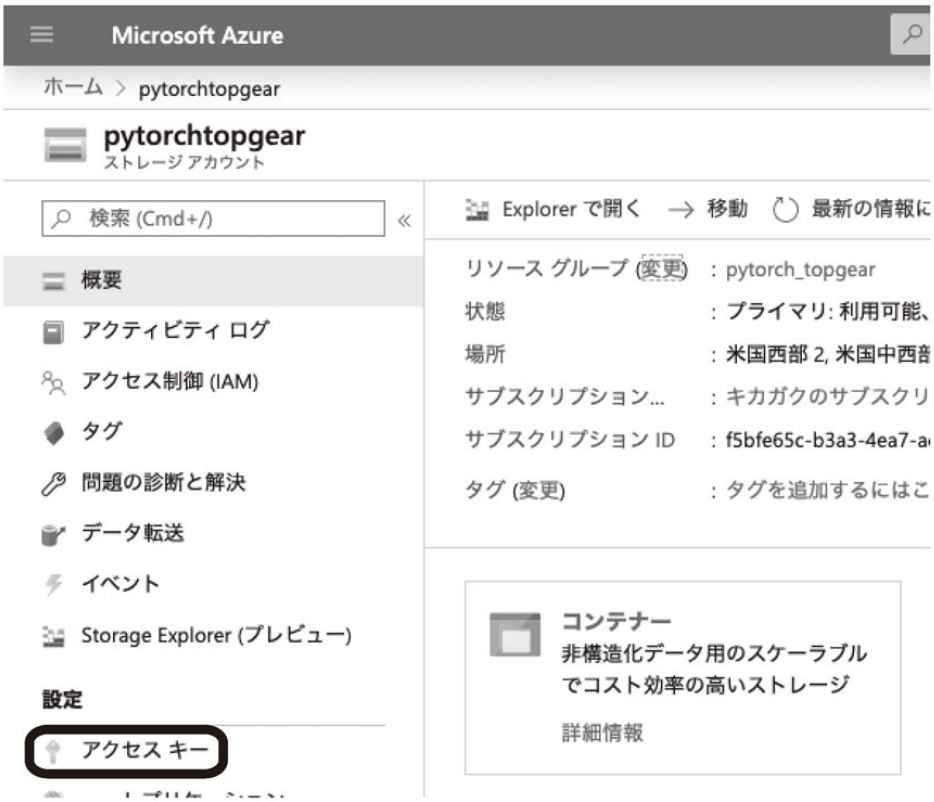
ライブラリをインストール

使用している Jupyter Notebook から Azure Blob Storage を使用できるよう、azure-storage-blob をインストールします。azure-storage-blob は、Python で Azure Blob Storage を扱うためのライブラリです。

```
!pip install azure-storage-blob

import azure.storage.blob as azureblob
```

Blob Storage との接続

まず、作成した Blob Storage との接続を行います。この接続には Blob Storage の account_name と account_key が必要となるので、Azure のポータルから確認しましょう。

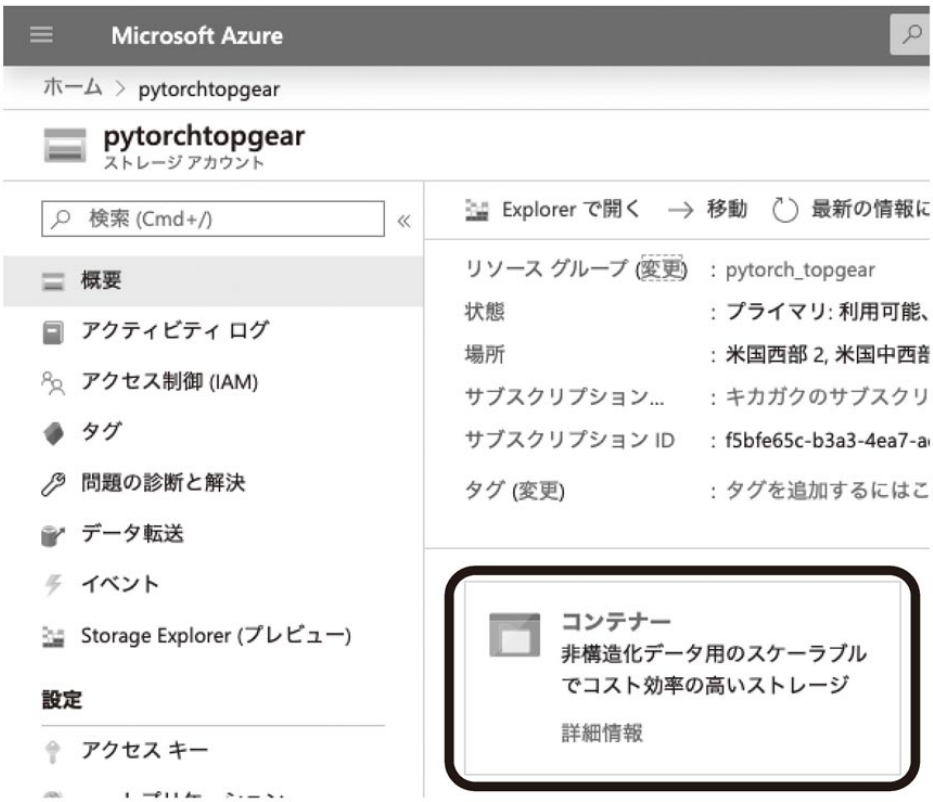


ストレージアカウント名が account_name、key1 のキーが account_key になります。取得しましょう。

いくつか接続方法はあるのですが、ここでは SAS（Shared Access Signature）トークンを使用した方法を選択します。SAS トークンは generate_account_sas() メソッドを使って、account_name、acount_key、許可の範囲、有効期限を自動生成できるように引数に指定します。

```
# 必要なモジュールをインポート
from datetime import datetime, timedelta
from azure.storage.blob import BlobServiceClient, generate_account_sas, ResourceTypes, AccountSa⇨
sPermissions

# Blobへ接続
sas_token = generate_account_sas(
    account_name = '<your-storage-account-name>',
    account_key = '<your-account-access-key>',
    resource_types = ResourceTypes(service=True),
    permission = AccountSasPermissions(read=True),
    expiry = datetime.utcnow() + timedelta(hours=1)
)

blob_service_client = BlobServiceClient(account_url='https://<your_account_name>.blob.core.windo⇨
ws.net', credential=sas_token)
```

これで Blob Storage との接続が完了しました。まだデータを保存していないので、コンテナ内部は空になっています。



models コンテナを作成し、先ほど作成した Iris、MNIST 用のプログラムと学習済み重みを保

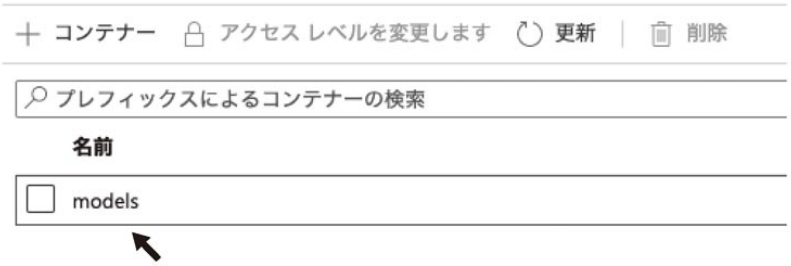
存しましょう。

ContainerClient の from_connection_string は、文字どおり**接続文字列**から認証し、コンテナを作成する方法です。

接続文字列をコピーして、conn_str に格納しましょう。

```
# コンテナを作成
from azure.storage.blob import ContainerClient

conn_str = '<your-connection-string>'
container_client = ContainerClient.from_connection_string(conn_str=conn_str,
container_name='models')

container_client.create_container()
```

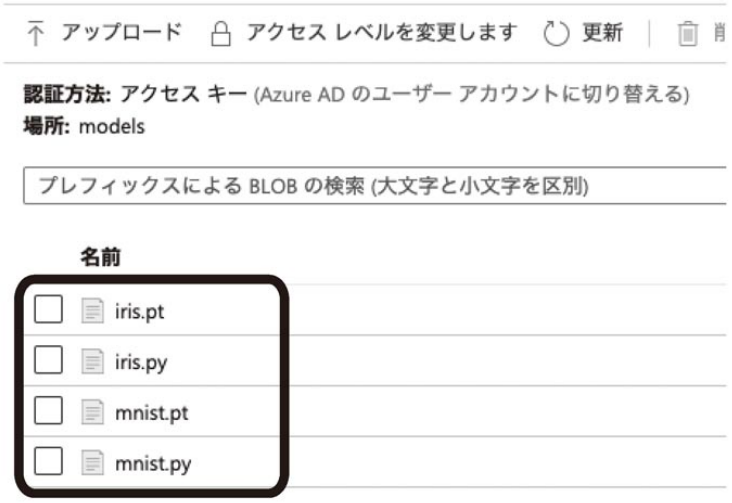
```
{'etag': '"0x8D7822290C840EF"',
 'last_modified': datetime.datetime(2019, 12, 16, 12, 22, 6, tzinfo=datetime.timezone.utc),
 'client_request_id': 'ac419d74-1ffe-11ea-aca3-acde48001122',
 'request_id': 'db10b5ab-301e-0036-550b-b4472c000000',
 'version': '2019-02-02',
 'date': datetime.datetime(2019, 12, 16, 12, 22, 6, tzinfo=datetime.timezone.utc),
 'error_code': None}
```



無事に models コンテナが完成しました。次にこの中にデータをアップロードしていきしょう。これも非常に簡単で、BlobClient.from_connection_string に upload_blob() メソッドがあります。読み込みたいファイルの PATH と保存したい blob_name を設定するだけでアップロードできます。

```
# 先ほど作成したファイルとPATHを指定する
data_dir = './models/'
filenames = [
    'iris.pt',
    'mnist.pt',
    'iris.py',
    'mnist.py'
]

from azure.storage.blob import BlobClient

# ファイルのアップロード
for filename in filenames:
    blob = BlobClient.from_connection_string(conn_str=conn_str, container_name='models', blob_na⇨
me=filename)
    with open(data_dir + filename, 'rb') as data:
        blob.upload_blob(data)
```

コマンド実行後に Azure ポータル画面からファイルを確認できれば成功です。



9.5 Azure VM を使った Web アプリケーションの基礎

実現場で学習を終えた後に推論環境を整備するとき、2 つの選択肢があります。

1. クラウドでサーバーを整える
2. オンプレミスで自社内でサーバーを整備する

クラウドを利用したサーバー環境を整備できれば、処理量に応じてスケールを変化させたり、システム異常が起きた際の対応をクラウド側で補ったりでき、保守が楽になることがあります。しかし、業界によっては、自社外にデータを展開するのがセキュリティ上難しいことも多いとよく耳にします。そのような場合には Web API サーバーを自ら簡単に作成できる技術が必須となります。今回はデータベースなどの紹介はしませんが、Flask と Docker、Azure Virtual Machines を使った簡易的な Web API のデプロイ方法を紹介します。

本節では Docker を使用して環境構築を行うため、Docker については第 5 章を参照してください。

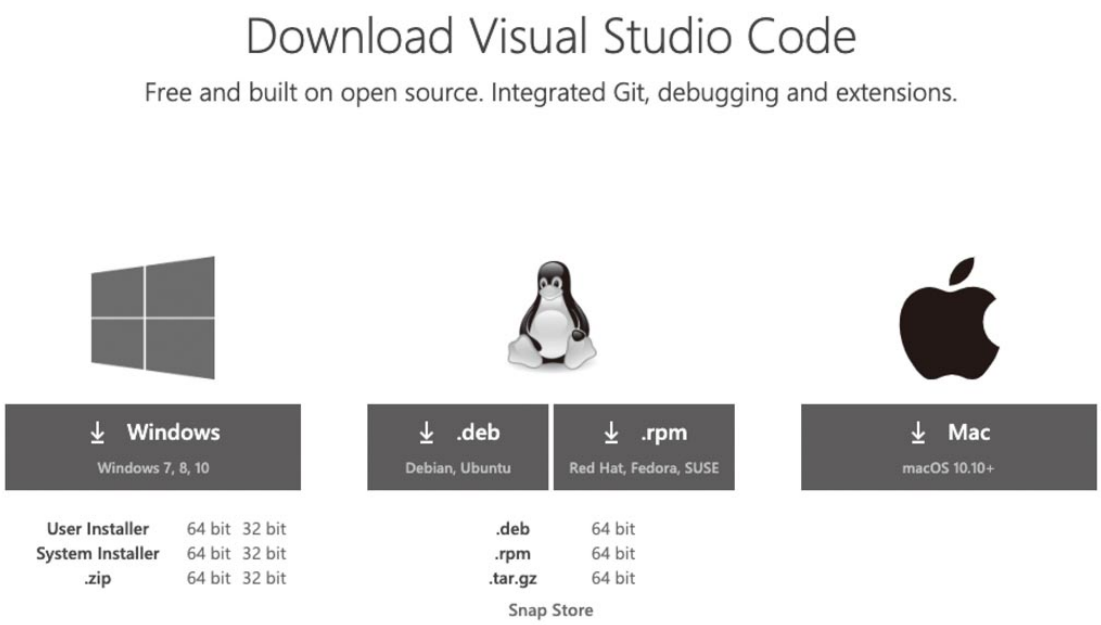
環境準備

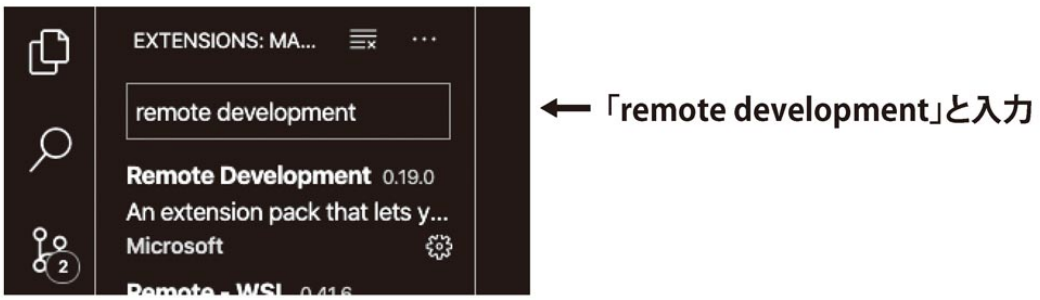
ポータル上から Azure VM を立ち上げます。ダウンロードに時間がかかるため、あらかじめ VM 上に pull しておいて他の環境準備に移りましょう。

```
sudo docker pull tiangolo/uwsgi-nginx-flask:python3.6
```

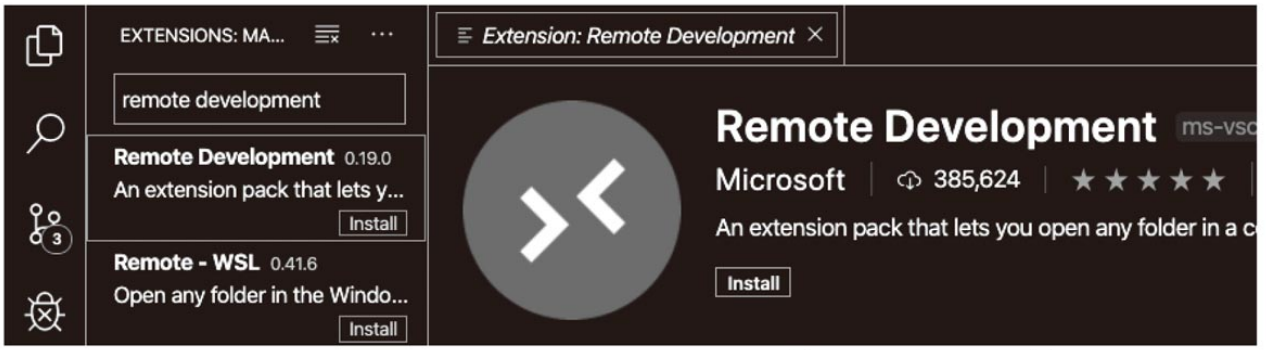
Visual Studio Code Insiders のインストール

Visual Studio Code は、Microsoft 社が開発しているソースコードエディタです。Windows、Linux、macOS 上で動作し、デバッグ、Git クライアントの統合、シンタックスハイライト、インテリセンス、スニペット、リファクタリングなどの機能を持ちます。カスタマイズも容易で、現在最も人気のあるエディタの 1 つになっています。Visual Studio Code の **Remote Development** と呼ばれる拡張機能を使用することで、リモート環境でもローカルと同じような感覚で操作し、開発できます。`https://code.visualstudio.com/download` からご利用の OS 向けのインストーラーをダウンロードしてください。

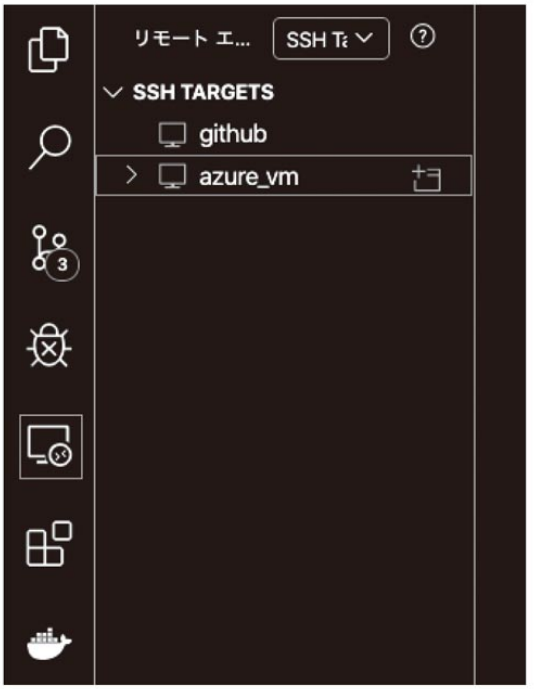




拡張機能「Remote Development」を使用します。名前のとおり、リモートから SSH 接続をして開発ができる優れものです。



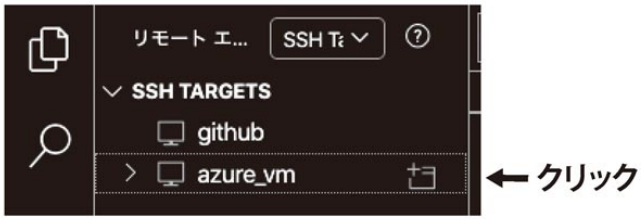
インストールを実行



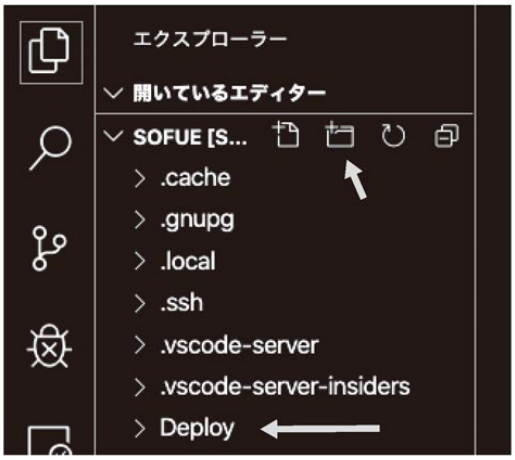
1. 歯車をクリック

2. .ssh/config を選択

config ファイルにどのマシンとリモートで接続するのかを記述します。Host はこちら側で管理するためのものなので、わかりやすいように自由に設定してください。HostName には VM の IP アドレス、User にはユーザー名を記述します。



Deploy フォルダを新たに作成したら、開発準備完了です。



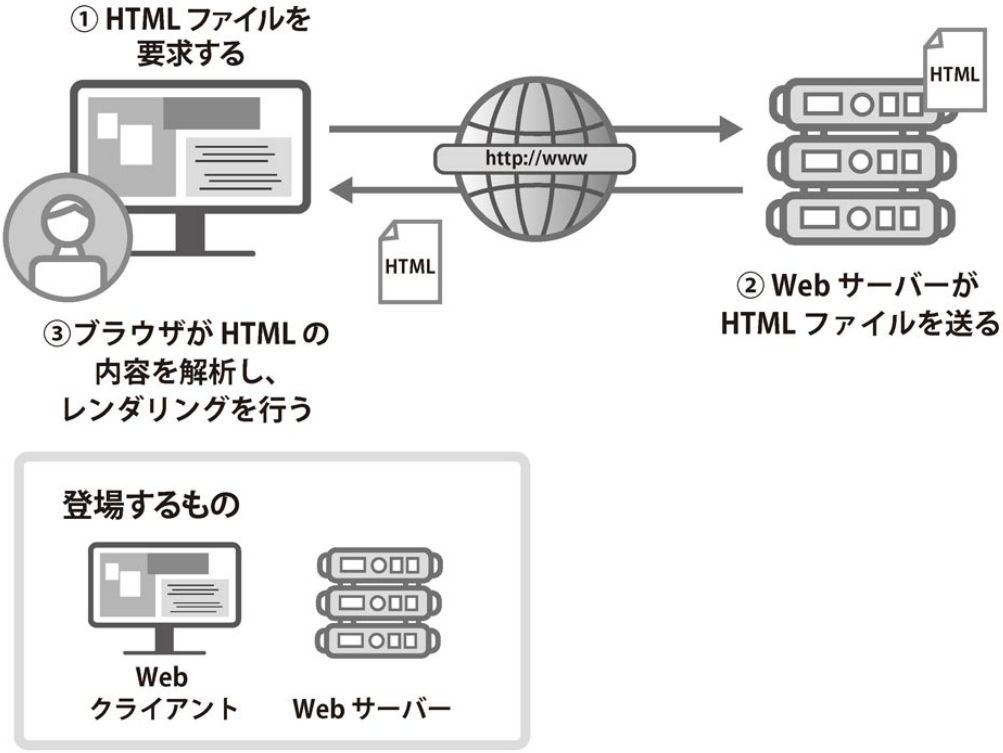
Deploy フォルダを作成

これからこの Deploy フォルダで作業していきます。その前に必要な知識をおさらいしておきましょう。

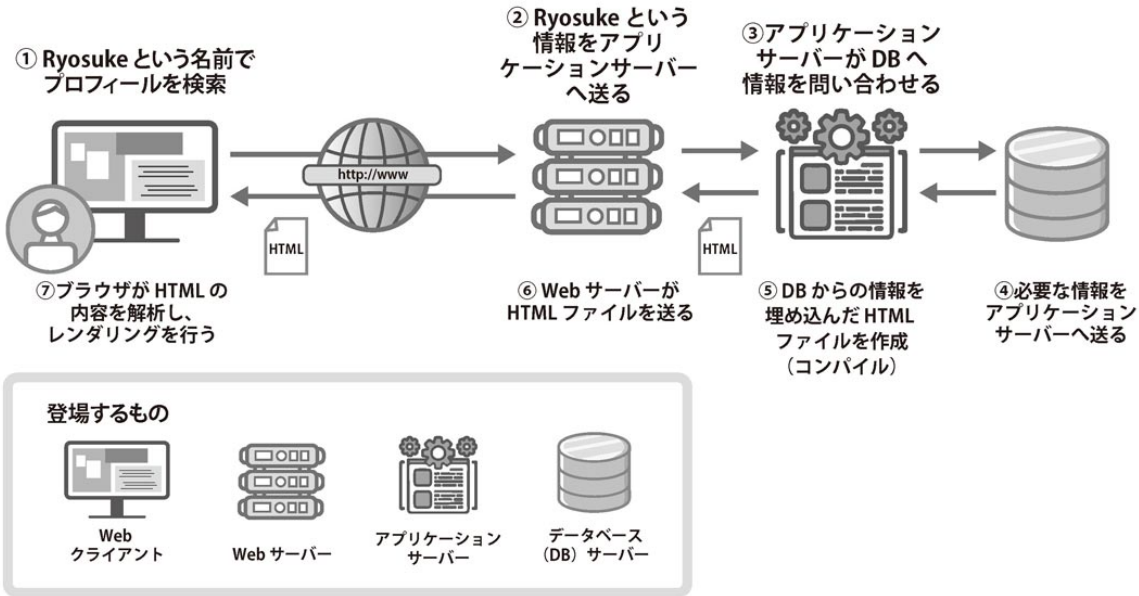
Web サイトと Web アプリケーション

一見、同じような言葉に聞こえますが、Web サイトと Web アプリケーションは、作る側からすると別の意味を持っています。

Web サイトは、次の図に示すように、Web サーバーへアクセスを行い、サーバー上に配置されている HTML ファイルを受け取り、ブラウザ側でレンダリングを行います。



これに対し、Web アプリケーションでは、単にファイルの授受だけではなく、Web クライアントから送られてきた情報に基づいて、データベース（DB）とのやり取りや HTML ファイルの作成などを行った後、その成果物としてアプリケーションサーバーから渡される HTML ファイルを Web クライアントに返す流れとなります。



Web フレームワーク

Web フレームワークとは、前述の Web アプリケーションの構築を簡単に実装できるツールです。1 から構築しようとすると非常に多くの工数を必要としますが、既存の Web フレームワークならば各フレームワークのチュートリアルを見ながらで学習コストも低く実装できます。

Python に対応した Web フレームワークにはいろいろありますが、ここでは必要最低限の機能を満たしていて軽量な **Flask** を採用することにします。

それでは Flask を使って推論サーバーを構築していきましょう。今回の推論は、これまでも扱ってきた Iris データの予測とします。始めにこれから使用するモジュールをインストールしておきます。

Flask の雛形

リスト 9.3　api.py

```
# coding: utf-8
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'Hello World'

# メイン関数
if __name__ == '__main__':
    app.run()
```

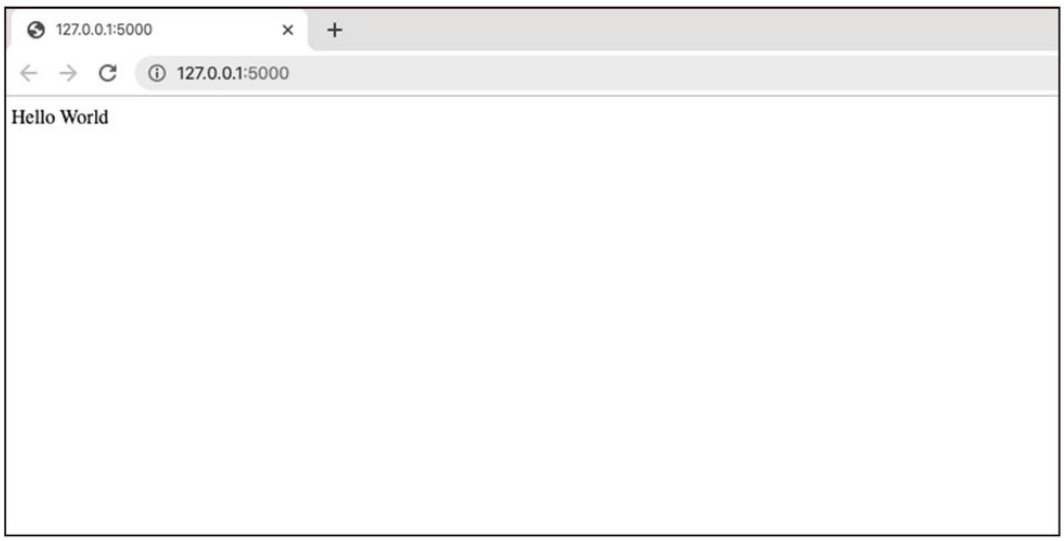
これを `api.py` としてファイルに保存し、Terminal で作業フォルダに移動後、以下のコマンドで実行します。

```
python api.py
```

実行後、

```
Running on http://<IPアドレス>:5000/ (Press CTRL+C to quit)
```

のように表示されれば正しく Web サーバーは起動しています。Web ブラウザから `http://<IP アドレス>:5000` にアクセスして次の画面になれば、初めての Flask による Web アプリケーションの作成は成功です。



JSON で出力の値を受け取る

早速推論処理を組み込んでいきたいところですが、その前に推論結果をどのような形式で受け取るかを確認しておきましょう。一般に Python では、辞書と同じく key と value を持った **JSON** と呼ばれる形式でデータのやり取りを行うのが一般的です。

Flask には jsonify という関数が準備されており、これを使って辞書型の変数を格納します。

リスト 9.4　iris_test.py

```
# coding: utf-8
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/')
def hello():
    ret = {'y': 3}
    return jsonify(ret)

# メイン関数
if __name__ == '__main__':
    app.run(debug=True)
```

では Web サーバーを立ち上げましょう。

```
python iris_test.py
```

Jupyter Notebook から Flask の Web API を呼び出していきます。リクエストには GET メソッドと POST メソッドがあります。

- **POST メソッド**：情報を URL に付加して送る
- **GET メソッド**：情報を Body という直接見えない内部に含めて送る

Python には requests というライブラリが準備されており、これで簡単にリクエスト（サーバーに情報の要求をする）を送ることができます。まずは GET メソッドのリスエストを送って、情報が返ってくるか確認しましょう。

```
import requests

# 起動しているURLを設定
url = 'http://<ipアドレス>:5000/'

# リクエストを送る
res = requests.get(url)
res
```

```
<Response [200]>
```

```
# 結果を確認
result = res.json()
result
```

```
{'y': 3}
```

```
result['y']
```

```
3
```

期待どおりの値が返ってきました。これが最も簡単ですが Web API です。次は POST メソッドで値付きリクエストを送ってみましょう。

POST メソッドで値付きのリクエストを送る

機械学習の現場で活用するには、入力値 x の情報を付けてリクエストを送る必要があります。この値付きの場合は一般に、POST メソッドを使用します。GET メソッドでも一応可能ではあるのですが、POST メソッドであれば SSL/TLS 通信経由で送信する値を暗号化することもで

き、セキュリティ上安全です。

リスト 9.5　iris_test.py

```
# coding: utf-8
from flask import Flask, jsonify, request

app = Flask(__name__)

# POSTメソッドに対応
@app.route('/', methods=['POST'])
def predict():
    # クエリの受け取り
    x = request.json['query']
    ret = {'x': x}
    return jsonify(ret)

# メイン関数
if __name__ == '__main__':
    app.run(debug=True)
```

```
from sklearn.datasets import load_iris

url = 'http://<ipアドレス>:5000/'

# Irisデータセットの読み込み
x, t = load_iris(return_X_y=True)
```

この入力値 x を使っていきます。NumPy 形式では扱えないものもあるため、Python のリスト形式にしておくことを推奨します。

```
# 最初のサンプルをクエリに設定（リスト形式にする）
query = list(x[0])
query, type(query)
```

```
([5.1, 3.5, 1.4, 0.2], list)
```

```
# 辞書型にしておく
params = {'query': query}
params
```

```
{'query': [5.1, 3.5, 1.4, 0.2]}
```

```
# POSTリクエスト
res = requests.post(url, json=params)
res
```

```
<Response [200]>
```

```
# 結果の確認
result = res.json()
result
```

```
{'x': [5.1, 3.5, 1.4, 0.2]}
```

送信した値がそのまま返ってきました。成功です。続いては、学習済みモデルを使って推論結果を返すようにしましょう。

学習済みモデルの読み込み

学習済みモデルを Blob Storage からダウンロードして、Flask に読み込みましょう。

リスト 9.6　iris_deploy.py

```
# coding: utf-8
from flask import Flask, jsonify, request
from datetime import datetime, timedelta
import azure.storage.blob as azureblob
import torch
import os

app = Flask(__name__)

# Blob Storageから定義ネットワークのクラスと学習済みモデルをダウンロード
model, pretrained = 'iris.py', 'iris.pt'
if not os.path.exists(model) or not os.path.exists(pretrained):
    print('downloading the model and the pretrained parameters')
    # Blobへ接続
    sas_token = azureblob.generate_account_sas(
        account_name = '<your-account-name>',
        account_key = '<your-account-key>',
        resource_types = azureblob.ResourceTypes(service=True),
```

```
        permission = azureblob.AccountSasPermissions(read=True),
        expiry = datetime.utcnow() + timedelta(hours=1)
    )
    blob_service_client = azureblob.BlobServiceClient(account_url='<your-accouunt-url>', credent⇨
ial=sas_token)

    # コンテナとの接続
    conn_str = '<your-connection-string>'
    container = 'models'
    for filename in [model, pretrained]:
        with open(filename, 'wb') as my_blob:
            blob = azureblob.BlobClient.from_connection_string(conn_str=conn_str, container_name⇨
=container, blob_name=filename)
            download_stream = blob.download_blob()
            my_blob.write(download_stream.readall())

# ネットワークの定義と学習済みモデルのロード
from iris import Net
net = Net()
net.load_state_dict(torch.load(pretrained))

# POSTメソッドに対応
@app.route('/', methods=['POST'])
def predict():
    # クエリの受け取り
    x = request.json['query']
    ret = {'x': x}
    return jsonify(ret)

# メイン関数
if __name__ == '__main__':
    app.run(debug=True)
```

さらに、推論処理も書き足していきましょう。上記のプログラムを書き換えて、送信されてきた値に対して学習済みモデルで推論を行い、予測値を返すようにします。

```
# coding: utf-8
from flask import Flask, jsonify, request
from datetime import datetime, timedelta
import azure.storage.blob as azureblob
import torch
import torch.nn.functional as F
import os

app = Flask(__name__)

# Blob Storageから定義ネットワークのクラスと学習済みモデルをダウンロード
```

```
model, pretrained = 'iris.py', 'iris.pt'
if not os.path.exists(model) or not os.path.exists(pretrained):
    print('downloading the model and the pretrained parameters')
    # Blobへ接続
    sas_token = azureblob.generate_account_sas(
        account_name = '<your-account-name>',
        account_key = '<your-account-key>',
        resource_types = azureblob.ResourceTypes(service=True),
        permission = azureblob.AccountSasPermissions(read=True),
        expiry = datetime.utcnow() + timedelta(hours=1)
    )
    blob_service_client = azureblob.BlobServiceClient(account_url='<your-accouunt-url>', credent⇨
ial=sas_token)

    # コンテナとの接続
    conn_str = '<your-connection-string>'
    container = 'models'
    for filename in [model, pretrained]:
        with open(filename, 'wb') as my_blob:
            blob = azureblob.BlobClient.from_connection_string(conn_str=conn_str, container_name⇨
=container, blob_name=filename)
            download_stream = blob.download_blob()
            my_blob.write(download_stream.readall())


# ネットワークの定義と学習済みモデルのロード
from iris import Net
net = Net()
net.load_state_dict(torch.load(pretrained))
net.eval()

# POSTメソッドに対応
@app.route('/', methods=['POST'])
def predict():
    # クエリの受け取り
    x = request.json['query']
    x = torch.tensor(x).unsqueeze(0)
    # 推論
    with torch.no_grad():
        y = F.softmax(net(x), 1)[0]
        _, index = torch.max(y, 0)

    result = {'label': int(index), 'probability': float(y[index])}
    return jsonify(result)

# メイン関数
if __name__ == '__main__':
    app.run(debug=True)
```

```
import requests

from sklearn.datasets import load_iris

# Irisデータセットの読み込み
x, t = load_iris(return_X_y=True)

# URLの指定
url = 'http://<IPアドレス>:5000/'

# 最初のサンプルをクエリに設定（リスト形式にする）
query = list(x[0])
query, type(query)

# 辞書型にしておく
params = {'query': query}
params
```

```
{'query': [5.1, 3.5, 1.4, 0.2]}
```

```
# POSTをリクエスト
res = requests.post(url, json=params)
res
```

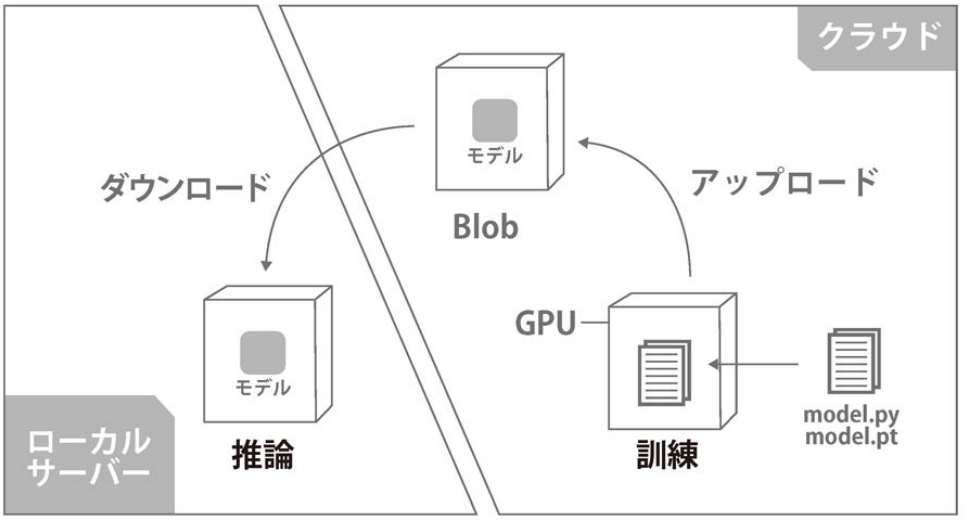
```
<Response [200]>
```

```
# 結果の確認
result = res.json()
result
```

```
{'label': 0, 'probability': 0.9751327633857727}
```

このように予測値が得られるような Web API を作ることができました。機械学習エンジニアは学習部分にだけ注目しがちですが、実際に使用するサービスでは推論部分を使用しており、モデル構築と同様に重要です。

Web API サーバーのデプロイ



ここでは Docker を使って簡単に Flask + Nginx の API サーバーを立てることを練習します。まず試験的に、Docker イメージをビルドして立ち上げるところまでを実際にやってみましょう。

デスクトップに flask_api ディレクトリを作成します。現在は空のディレクトリですが、後々の構成は次のような形にしていきます。

```
flask_api
├── Dockerfile
└── app
    └── main.py
```

実際の Web アプリケーションを本番環境で運用する場合は、Flask 内の app.run のような簡易の Web サーバーを使用するのではなく、Apache や Nginx のような Web サーバーを構築します。Flask の Web サーバーはローカル等の開発段階のテスト用です。これに対し、Nginx のような Web サーバーを構築するとなると、一気に敷居が上がります。

そこで登場するのが Docker です。Web サーバー Nginx の環境をすでに構築済みの Docker コンテナを立てて、初心者には非常に難しい環境構築を一瞬で終わらせてしまいましょう。

Docker + Nginx + Flask で本番でも使えるシステム構築

main.py という名前で、Flask のアプリケーションを作成します。

```
# coding: utf-8
from flask import Flask

app = Flask(__name__)
```

```
@app.route('/')
def hello():
    return 'Hello World'

# メイン関数
if __name__ == '__main__':
    app.run()
```

Dockerfile の作成

今回も Docker を使いますが、ここでは docker pull のように Docker Hub からダウンロードしてくるのはなく、Dockerfile と呼ばれるファイルを作成します。第 5 章でも説明しましたが、Dockerfile はあるイメージをもとに、さらに自分自身で必要な構成を加えた Docker イメージを構築できるファイルのことです。また、Dockerfile は単なるテキストファイルなので、データサイズは非常に小さく、これならば GitHub 上で管理することもできます。それでは、Dockerfile を作ってみましょう。

リスト 9.7　Dockerfile

```
FROM tiangolo/uwsgi-nginx-flask:python3.6

COPY ./app /app
```

これを Dockerfile という名前で flask_api ディレクトリ直下に保存しましょう。ここで注意してほしいこととして、Dockerfile には拡張子は付けません。

FROM は、どのイメージをもとにビルドを行うかの指定です。今回は Docker Hub で公開されている tiangolo/uwsgi-nginx-flask という簡単に環境構築できるイメージを使い、これをベースにさらに必要な変更を加えていきます。

COPY は、ビルドするイメージに対してファイルのコピーを行います。ここでは現在の作業ディレクトリ（カレントディレクトリ）下にある./app をイメージの/app にコピーしています。ちなみに、ベースにした tiangolo/uwsgi-nginx-flask のイメージでは、/app にあるアプリケーションを探すように config ファイルで設定されているため、その仕様に合わせるために flask_api/app ディレクトリを Docker コンテナ上の/app へコピーしています。もちろん config ファイルを編集する方法もありますが、まずは仕様に合わせるのが妥当です。

ほかにも Dockerfile にはいろいろな記述ができるので、作業しながら覚えていってください。

Docker イメージのビルド

それでは作成した Dockerfile をもとに新たな Docker イメージを作りましょう。この作業のことを**ビルド**（build）と呼びます。Docker 上でビルドを行うコマンドは build です。ここでは nginx-flask-hello-world というイメージ名にします。

```
sudo docker build -t nginx-flask-hello-world .
```

Successfully built と出ればビルド成功です。ビルドに使ったコマンドのオプションは以下のような構成になっています。

```
docker build -t <イメージ名>:<タグ> <参照したいDockerfileの場所>
```

このビルドが完了して新しい Docker イメージが作成されたら、sudo docker images を実行して現在存在するイメージを列挙してみましょう。「nginx-flask-hello-world」のイメージができていれば、ビルドは成功です。

コンテナの立ち上げ

それでは作成したイメージからコンテナを新しく立ち上げます。

```
sudo docker run -d --restart=always -p 80:80 -t nginx-flask-hello-world
```

今回は Web サーバーの状態を表示する必要は特にないので、コンテナ立ち上げ時のオプションとして-d を付け、コンテナを起動した後**ディタッチ**（起動したまま抜ける）をしました。

さらに、--restart=always というオプションも付けていますが、これはコンテナがクラッシュした際などに自動で再起動してくれるという便利な機能です。Web サーバー周りのコンテナを立ち上げるときにはこのオプションを付けておくとよいでしょう。

機械学習向けの Web API を Flask と Nginx（Docker）で構成

それでは上記の内容を踏まえ、学習済みモデルを使用して入力に対する出力を返す Web API を作りましょう。API 側のプログラムを以下のように記述します。

```
# coding: utf-8
from flask import Flask, jsonify, request
from datetime import datetime, timedelta
import azure.storage.blob as azureblob
import torch
import torch.nn.functional as F
import os

app = Flask(__name__)

# Blob Storageから定義ネットワークのクラスと学習済みモデルをダウンロード
model, pretrained = 'iris.py', 'iris.pt'
if not os.path.exists(model) or not os.path.exists(pretrained):
    print('downloading the model and the pretrained parameters')
```

```
    # Blobへ接続
    sas_token = azureblob.generate_account_sas(
        account_name = '<your-account-name>',
        account_key = '<your-account-key>',
        resource_types = azureblob.ResourceTypes(service=True),
        permission = azureblob.AccountSasPermissions(read=True),
        expiry = datetime.utcnow() + timedelta(hours=1)
    )
    blob_service_client = azureblob.BlobServiceClient(account_url='<your-accouunt-url>', credent⇨
ial=sas_token)

    conn_str = '<your-connection-string>'
    container = 'models'
    for filename in [model, pretrained]:
        with open(filename, 'wb') as my_blob:
            blob = azuleblob.BlobClient.from_connection_string(conn_str=conn_str, container_name⇨
=container, blob_name=filename)
            download_stream = blob.download_blob()
            my_blob.write(download_stream.readall())


# ネットワークの定義と学習済みモデルのロード
from iris import Net
net = Net()
net.load_state_dict(torch.load(pretrained))
net.eval()

# POSTメソッドに対応
@app.route('/', methods=['POST'])
def predict():
    # クエリの受け取り
    x = request.json['query']
    x = torch.tensor(x).unsqueeze(0)
    # 推論
    with torch.no_grad():
        y = F.softmax(net(x), 1)[0]
        _, index = torch.max(y, 0)

    result = {'label': int(index), 'probability': float(y[index])}
    return jsonify(result)

# メイン関数
if __name__ == '__main__':
    app.run(debug=True, port=80)
```

ほかの関数でも参照するため、model はグローバル変数として定義します。

インストールするライブラリの設定

実行に必要なライブラリをインストールするために、requirements.txt を作成します。今回必要なライブラリは以下のとおりです。

```
Flask
torch
azure-storage-blob
```

Dockerfile では、この requirements.txt を参照して pip コマンドでライブラリをインストールするよう設定します（RUN 行）。

```
FROM tiangolo/uwsgi-nginx-flask:python3.6

COPY requirements.txt /tmp
RUN pip install -U pip
RUN pip install -r /tmp/requirements.txt

COPY ./app /app
```

イメージとコンテナの作成

それではイメージのビルドとコンテナの立ち上げを行いましょう。作成した Dockerfile からイメージを作ります。

```
sudo docker build -t nginx-flask .
```

コンテナを立ち上げるには以下のようにします。

```
sudo docker run -d --restart=always --rm -it -p 80:80 --name api -t nginx-flask
```

デプロイした Web API を使用して推論

デプロイした Web API に対して POST リクエストを送り、結果が返ってくることを確認しましょう。流れは前にやったことと同じなので、以下のように値が返ってきたら成功です。

```
import requests

url = 'http://<IPアドレス>/'

from sklearn.datasets import load_iris

# Irisデータセットの読み込み
x, t = load_iris(return_X_y=True)
```

```
# 最初のサンプルをクエリに設定（リスト形式が必須）
query = list(x[0])
query, type(query)
```

```
([5.1, 3.5, 1.4, 0.2], list)
```

```
params = {'query': query}
params
```

```
{'query': [5.1, 3.5, 1.4, 0.2]}
```

```
res = requests.post(url, json=params)

result = res.json()
result
```

```
{'label': 0, 'probability': 0.9751327633857727}
```

これで IaaS を使用した簡易なデプロイは完了です。本番環境では、これらに加えてセキュリティや、大量のリクエストをさばくためにどのようにスケールするかなどを考慮する必要がありますが、Azure VM を使えばスケール等も容易です。また、Docker と Flask の知識があれば、機械学習に必要な基本的な環境を構築できることがわかりました。

9.6 Azure ML で学習

次に **PaaS**（Platform as a Service）を使ったデプロイを紹介します。推論サーバーを自作で構築するのもよいのですが、スケールアウトのタイミングやセキュリティ周りの知識が必要など、管理するハードルは決して低くありません。ここでは PaaS を使った例として、Azure ML を使ってデプロイまで実装します。

Azure ML の準備

これまでに紹介した流れと同様に、Azure ML の設定を行い、MNIST データセットを使ってハイパーパラメータチューニングを含めたモデルの学習から、学習済みの最も性能の高いモデルを推論サーバーへデプロイするまでの流れを紹介します。それでは始めましょう。

```
# 必要なモジュールの読み込み
import numpy as np
import os

import azureml
from azureml.core import Workspace, Experiment
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

# バージョンの確認
azureml.core.VERSION
```

```
'1.0.79'
```

Workspace

```
# 各種設定
name = '<your-compute-name>'
subscription_id = '<your-subscription-id>'
resource_group = '<your-resource-group>'

# Workspaceを取得
ws = Workspace.get(
    name = name,
    subscription_id = subscription_id,
    resource_group = resource_group
)

ws
```

```
Workspace.create(name='pytorch_topgear_parameter', subscription_id='f5bfe65c-b3a3-4ea7-ac4c-7d4d⇨
bbbeef26', resource_group='pytorch_topgear_parameter')
```

Experiment

```
# 実験環境
experiment = Experiment(
    workspace = ws,
    name = 'pytorch-mnist-mldeploy'
)

experiment
```

```
Experiment(Name: pytorch-mnist-mldeploy,
Workspace: pytorch_topgear_parameter)
```

Data Store

```
ds = ws.get_default_datastore()
ds
```

```
<azureml.data.azure_storage_datastore.AzureBlobDatastore at 0x122658f90>
```

Compute Target

```
# 計算リソース
compute_target = ComputeTarget(
    workspace=ws,
    name='gpu-compute'
)
compute_target
```

```
AmlCompute(workspace=Workspace.create(name='pytorch_topgear_parameter', subscription_id='f5bfe65c-b3a3-4ea7-ac4c-7d4dbbbeef26', resource_group='pytorch_topgear_parameter'), name=gpu-compute, id=/subscriptions/f5bfe65c-b3a3-4ea7-ac4c-7d4dbbbeef26/resourceGroups/pytorch_topgear_parameter/providers/Microsoft.MachineLearningServices/workspaces/pytorch_topgear_parameter/computes/gpu-compute, type=AmlCompute, provisioning_state=Succeeded, location=westus2, tags=None)
```

```
comp_info = compute_target.get_status().serialize()
print('VM size:', comp_info['vmSize'])
```

```
VM size: STANDARD_NC6
```

データの準備

MNIST を使います。いつもどおり準備していきましょう。

```
from torchvision import transforms, datasets

# Tensor形式に変換
transform = transforms.Compose([
    transforms.ToTensor()
])

train_val = datasets.MNIST(root='./data/MNIST', train=True, download=True, transform=transform)
test = datasets.MNIST(root='./data/MNIST', train=False, download=True, transform=transform)

# Data Storeにデータをアップロード
ds.upload(src_dir='./data/MNIST', target_path='data', overwrite=False)
```

```
Uploading an estimated of 20 files
 …
$AZUREML_DATAREFERENCE_678b3d2d009345f9995baceef2542443
```

学習用スクリプトの作成

aml_mnist/scripy/train_mnist.py を以下のとおり作成します。

```
# coding: utf-8
import argparse
import numpy as np
import os
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
from torchvision import transforms, datasets
from torch.utils.data import DataLoader
import pytorch_lightning as pl
from pytorch_lightning import Trainer
from azureml.core.run import Run

# 学習結果の取得
run = Run.get_context()

class Net(pl.LightningModule):
```

```
    def __init__(self, hparams, num_workers=8):
        super(Net, self).__init__()
        self.hparams = hparams
        self.num_workers = num_workers
        self.conv = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(28 * 28, self.hparams.n_hidden)
        self.fc2 = nn.Linear(self.hparams.n_hidden, 10)

    def _dataloader(self, train):
        transform = transforms.Compose([transforms.ToTensor()])
        dataset = torchvision.datasets.MNIST(root=self.hparams.data_dir, train=train, download=T⇨
rue, transform=transform)
        loader = DataLoader(dataset, self.hparams.batch_size, shuffle=True, num_workers=self.num⇨
_workers)
        return loader

    def lossfun(self, y, t):
        return F.cross_entropy(y, t)

    def configure_optimizers(self):
        return torch.optim.SGD(self.parameters(), lr=self.hparams.lr, momentum=self.hparams.mome⇨
ntum)

    def forward(self, x):
        x = self.conv(x)
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 28 * 28)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

    @pl.data_loader
    def train_dataloader(self):
        return self._dataloader(train=True)

    def training_step(self, batch, batch_nb):
        x, t = batch
        y = self.forward(x)
        loss = self.lossfun(y, t)
        results = {'loss': loss}
        return results

    @pl.data_loader
    def val_dataloader(self):
        return self._dataloader(train=False)

    def validation_step(self, batch, batch_nb):
        x, t = batch
```

```
        y = self.forward(x)
        loss = self.lossfun(y, t)
        y_label = torch.argmax(y, dim=1)
        acc = torch.sum(t == y_label) * 1.0 / len(t)
        results = {'val_loss': loss, 'val_acc': acc}
        return results

    def validation_end(self, outputs):
        avg_loss = torch.stack([x['val_loss'] for x in outputs]).mean()
        avg_acc = torch.stack([x['val_acc'] for x in outputs]).mean()
        results =  {'val_loss': avg_loss, 'val_acc': avg_acc}
        return results

    @pl.data_loader
    def test_dataloader(self):
        return self._dataloader(train=False)

    def test_step(self, batch, batch_nb):
        x, t = batch
        y = self.forward(x)
        loss = self.lossfun(y, t)
        y_label = torch.argmax(y, dim=1)
        acc = torch.sum(t == y_label) * 1.0 / len(t)
        results = {'test_loss': loss, 'test_acc': acc}
        return results

    def test_end(self, outputs):
        avg_loss = torch.stack([x['test_loss'] for x in outputs]).mean()
        avg_acc = torch.stack([x['test_acc'] for x in outputs]).mean()
        results = {'test_loss': avg_loss, 'test_acc': avg_acc}
        return results

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--data-dir', type=str, dest='data_dir', default='./data')
    parser.add_argument('--output_dir', type=str, dest='output_dir', default='./outputs')
    parser.add_argument('--batch-size', type=int, dest='batch_size', default=50)
    parser.add_argument('--epoch', type=int, dest='epoch', default=20)
    parser.add_argument('--n-hidden', type=int, dest='n_hidden', default=100)
    parser.add_argument('--lr', type=float, dest='lr', default=0.01)
    parser.add_argument('--momentum', type=float, dest='momentum', default=0.9)

    hparams = parser.parse_args()

    torch.manual_seed(0)
    net = Net(hparams)
    trainer = Trainer(max_nb_epochs=hparams.epoch)
```

```
    print('Start training')
    trainer.fit(net)
    print('Finish training')

    print('Validation Score:', trainer.callback_metrics)
    trainer.test()
    print('Test Score:', trainer.callback_metrics)

    # 学習結果の追跡
    run.log('lr', hparams.lr)
    run.log('n_hidden', hparams.n-hidden)
    run.log('momentum', hparams.momentum)
    run.log('best_acc', trainer.callback_metrics)

    # 学習済みモデルの保存
    os.makedirs(hparams.output_dir, exist_ok=True)
    torch.save(net.state_dict(), '{}/model.pt'.format(hparams.output_dir))

if __name__ == '__main__':
    main()
```

これまでと流れとしてはほぼ同じですが、1 点、学習済みモデルの保存の機能が最下部に追記されています。Azure ML では outputs フォルダ以下に置いたファイルは run の履歴と一緒に保存されるという特性があります。そのため、最も性能が良かった run と紐づけて後からモデルを取り出せるように、outputs フォルダ下にモデルを保存しています。

ハイパーパラメータチューニングの設定

ベイズ最適化を使って、ハイパーパラメータのチューニングを行っていきましょう。手順は第 7 章と同じ流れです。

```
# 必要なモジュールを読み込み
from azureml.widgets import RunDetails
from azureml.train.dnn import PyTorch
from azureml.train.hyperdrive import BayesianParameterSampling, HyperDriveConfig, PrimaryMetricG⇨
oal
from azureml.train.hyperdrive import choice, uniform

# ハイパーパラメータ探索範囲の設定
parameters = BayesianParameterSampling(
    {
        '--n-hidden': choice(range(10, 784)),
        '--lr': uniform(1e-6, 1e-1),
        '--momentum': uniform(1e-6, 1.0)
    }
```

```
)

script_params = {
    '--data-dir': ds.path('data').as_mount(),
    '--output_dir': './outputs',
    '--epoch': 20,
    '--batch-size': 128
}

estimator = PyTorch(
    source_directory = 'aml_mnist/script',
    script_params = script_params,
    compute_target = compute_target,
    pip_packages = ['pytorch-lightning'],
    framework_version = '1.3',
    entry_script = 'train_mnist.py',
    use_gpu=True
)

# 最大ノード数は4に設定
hyperdrive_config = HyperDriveConfig(
    estimator=estimator,
    hyperparameter_sampling=parameters,
    policy=None,
    primary_metric_name='test_accuracy',
    primary_metric_goal=PrimaryMetricGoal.MAXIMIZE,
    max_total_runs=4,
    max_concurrent_runs=4,
    max_duration_minutes=1800)

# 学習
hyperdrive_run = experiment.submit(config=hyperdrive_config)
print(hyperdrive_run)
```

```
Run(Experiment: pytorch-mnist-mldeploy,
Id: pytorch-mnist-mldeploy_1576941787172189,
Type: hyperdrive,
Status: Running)
```

```
# 学習過程を表示
RunDetails(hyperdrive_run).show()
```

```
_HyperDriveWidget(widget_settings={'childWidgetDisplay': 'popup', 'send_telemetry': False, 'log_level': 'NOTSE…
```

最も良いモデルの保存

HyperDriveRun クラスの get_best_run_by_primary_metric() メソッドで最も性能の良かった実施を抽出できます。

```
from azureml.train.hyperdrive import HyperDriveRun

# 実行結果の詳細を保存
hdr = HyperDriveRun(experiment,
                    run_id='pytorch-mnist-mldeploy_1576941787172189',
                    hyperdrive_config=hyperdrive_config)

print(hdr.get_metrics())
```

```
{'pytorch-mnist-mldeploy_1576941787172189_0': {'lr': 0.0378064918358727, 'n_hidden': 767, 'momentum': 0.329396445095349, 'best_acc': "{'test_loss': 0.05961843207478523, 'test_acc': 0.9821993708610535}"}, 'pytorch-mnist-mldeploy_1576941787172189_1': {'lr': 0.0757155691931625, 'n_hidden': 23, 'momentum': 0.0785633618606872, 'best_acc': "{'test_loss': 0.10974286496639252, 'test_acc': 0.9661787748336792}"}, 'pytorch-mnist-mldeploy_1576941787172189_2': {'lr': 0.0761472427162274, 'n_hidden': 160, 'momentum': 0.353614738793176, 'best_acc': "{'test_loss': 0.0605369471013546, 'test_acc': 0.9830893874168396}"}, 'pytorch-mnist-mldeploy_1576941787172189_3': {'lr': 0.0826280103042935, 'n_hidden': 300, 'momentum': 0.411366691962188, 'best_acc': "{'test_loss': 0.062250662595033646, 'test_acc': 0.9828916192054749}"}}
```

test_acc を見ると、pytorch-mnist-mldeploy_1576941787172189_2 が最も良い値になっています。

```
# 最も良かった値を表示
best_hdr = HyperDriveRun(experiment,
                    run_id='pytorch-mnist-mldeploy_1576941787172189_2',
                    hyperdrive_config=hyperdrive_config)

best_hdr
```

Experiment	Id	Type	Status	Details Page	Docs Page
pytorch-mnist-mldeploy	pytorch-mnist-mldeploy_1576941787172189_2	azureml.scriptrun	Completed	Link to Azure Machine Learning studio	Link to Documentation

```
#  最も良い値を表示
best_hdr.get_metrics()
```

この値を推論ネットワークの構築の際に使用するため、構造を覚えておきましょう。

9.7 Azure ML へのデプロイ

デプロイ自体は難しくありません。デプロイしたいモデルを決定するだけです。決めたモデルに基づいてイメージを作成し、そのイメージをデプロイしていきます。

```
# モデルの指定
model = best_hdr.register_model(
    model_name='pytorch_mnist_01',
    model_path='outputs/model.pt'
)

model.name
```

```
'pytorch_mnist_01'
```

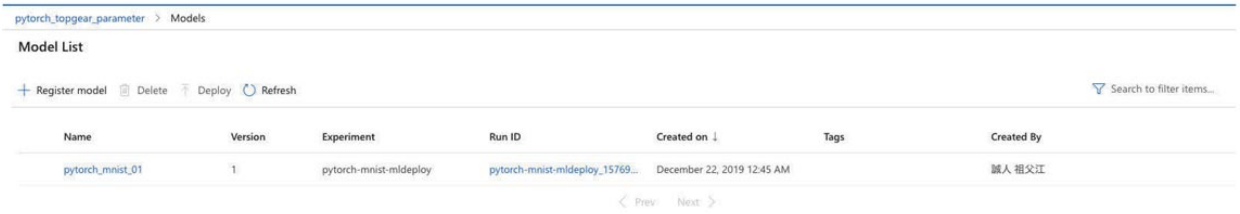
```
model.id
```

```
'pytorch_mnist_01:1'
```

```
model.version
```

```
1
```

Azure ポータルの左のタブからモデルを選択します。新しくデプロイしたモデルを追加できたことが確認できます。



イメージの作成

モデルを含めた推論まで行う Docker イメージを作成します。このイメージの作成には、次の 2 つのファイルが必要です。

- score.py：初期化 init() と推論処理 run()
- myenv.yml：必要なパッケージ

リスト 9.8　score.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import json
from azureml.core.model import Model

class Net(nn.Module):

    def __init__(self, n_hidden=160):
        super(Net, self).__init__()
        self.conv = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(28 * 28, n_hidden)
        self.fc2 = nn.Linear(n_hidden, 10)

    def forward(self, x):
        x = self.conv(x)
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 28 * 28)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

def init():
    global model

    model_path = Model.get_model_path('pytorch_mnist_01', _workspace=ws)
    model_state = torch.load(model_path, map_location=lambda storage, loc: storage)
    model = Net()
    model.load_state_dict(model_state)
```

```
    model.eval()

def run(data):
    # クエリの受け取り
    x = json.loads(data)['x']
    x = torch.tensor(x, dtype=torch.float32).unsqueeze(0)
    # 推論
    with torch.no_grad():
        y = F.softmax(model(x), 1)[0]
        _, index = torch.max(y, 0)

    result = {'label': int(index), 'probability': float(y[index])}
    return result
```

score.py の中身をデプロイする前に、エラーが起きないか確認しておきます。

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import json
from azureml.core.model import Model

class Net(nn.Module):

    def __init__(self, n_hidden=160):
        super(Net, self).__init__()
        self.conv = nn.Conv2d(in_channels=1, out_channels=4, kernel_size=3, stride=1, padding=1)
        self.fc1 = nn.Linear(28 * 28, n_hidden)
        self.fc2 = nn.Linear(n_hidden, 10)

    def forward(self, x):
        x = self.conv(x)
        x = F.max_pool2d(x, 2, 2)
        x = x.view(-1, 28 * 28)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# 学習済みパラメータ
model_path = Model.get_model_path('pytorch_mnist_01', version=1, _workspace=ws)
model_state = torch.load(model_path, map_location=lambda storage, loc: storage)

# モデルにパラメータをロード
model = Net()
model.load_state_dict(model_state)
model.eval()
```

```
Net(
  (conv): Conv2d(1, 4, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=784, out_features=160, bias=True)
  (fc2): Linear(in_features=160, out_features=10, bias=True)
)
```

それでは MNIST のデータ 1 枚を sample.png として推論してみましょう。

```
from PIL import Image
img = Image.open('data/sample.png')
img
```

```
# 前処理
from torchvision import transforms
transform = transforms.Compose([
        transforms.ToTensor()
])

x = transform(img)

type(x), x.dtype, x.shape
```

```
(torch.Tensor, torch.float32, torch.Size([1, 28, 28]))
```

```
# JSONで受け取れる形式に変換
data = json.dumps({'x': x.tolist()})

# クエリの受け取り
x = json.loads(data)['x']

# 画像用に次元を追加
x = torch.tensor(x, dtype=torch.float32).unsqueeze(0)

# 推論
with torch.no_grad():
    y = F.softmax(model(x), 1)[0]
```

```
    _, index = torch.max(y, 0)

{'label': int(index), 'probability': float(y[index])}
```

```
{'label': 5, 'probability': 0.9933018684387207}
```

うまく値が返ってきました。score.py はこれで良さそうです。

myenv.yml ファイルは以下のとおりです。pip の部分に記載しているパッケージ名は、必要に応じて変更してください。

リスト 9.9　myenv.yml

```
name: project_environment
dependencies:
- python=3.6.2

- pip:
  - azureml-defaults
  - torch
  - Pillow
channels:
- conda-forge
```

必要なファイルの準備が完了しました。イメージの展開を Azure ポータル上で行います。

イメージの作成には 5〜10 分ほどかかります。

イメージの展開

イメージの状態が「NotStarted」から「Succeeded」に変わったら、イメージを展開できます。**ACI**（Azure Container Instances）にモデルをデプロイしていく方法を追っていきます。ACI は、仮想マシンを管理する方法として有効なサービスで、複雑な処理ではなくコンテナ単位でタスク自動化やジョブ作成などの単純な処理を行うのに優れたソリューションです（`https://docs.microsoft.com/ja-jp/azure/container-instances/container-instances-overview`）。

モデルの展開後「Transactioning」という状態になり、うまくデプロイができると「Healthy」に変わります。「Unhealthy」や「Failed」となったらどこかでエラーが起きているので、ログを見ながら対処していく必要があります。

9.8　コマンドでの一連の流れの操作

ここまで Azure のポータル上でモデルからイメージを作成し、展開していく方法を紹介しましたが、Python のコマンドでも同じ一連の処理が可能です。どちらも基本的に同じですが、開発中にログを見たい場合には、コマンドで操作するほうがデバッグしやすいというメリットがあります。

```
# イメージの作成
from azureml.core.image import ContainerImage

image_config = ContainerImage.image_configuration(
    execution_script='score.py',
    runtime='python',
    conda_file='myenv.yml',
    enable_gpu=True
)

# イメージの展開
from azureml.core.webservice import AciWebservice

aciconfig = AciWebservice.deploy_configuration(
    cpu_cores=1,
    memory_gb=1
)

# デプロイの実行
from azureml.core.webservice import Webservice

service_name = 'mnistcli'
service = Webservice.deploy_from_model(
    workspace=workspace,
    name=service_name,
    models=[model],
    image_config=image_config,
    deployment_config=aciconfig
)

service.wait_for_deployment(show_output=False)
print(service.state)

# デバッグする際にログを確認
# print(service.get_logs())

# 使用した一連のサービスのリソースを開放
# service.delete()
```

9.9 推論サーバーへのリクエスト

デプロイ完了後、スコアリング URI がエンドポイントとなります。「展開」からスコアリング

URI をコピーしましょう。

属性	
説明	
状態	Healthy
コンピューティングの種類	ACI
サービス ID	mnist-pytorch
タグ	
作成日	2019/12/21 16:11:52 UTC
最終更新日時	2019/12/21 16:11:52 UTC
イメージ ID	mnist-pytorch:1
スコアリング URI	http://c8288063-830e-44f2-918d-9246faa95ef7.westus2.azurecontainer.io/score
CPU	0.1
メモリ	0.5 GB

それでは、MNIST で使われている画像から推論までの一連の流れを追ってみましょう。

```
import requests
from torchvision import transforms, datasets
from PIL import Image
import json

transform = transforms.Compose([
        transforms.ToTensor()
])

img = Image.open('data/sample.png')
img
```

```
x = transform(img)

type(x), x.dtype, x.shape
```

```
(torch.Tensor, torch.float32, torch.Size([1, 28, 28]))
```

```
# 展開したイメージのスコアリングURI
url = '<your-scoring-uri>'

# データをJSON形式に変更
data = json.dumps({'x': x.tolist()})

headers = {'content-type': 'application/json'}

# データを送信する
res = requests.post(url, data=data, headers=headers)
res
```

```
<Response [200]>
```

```
# 推論結果の確認
result = res.json()
result
```

```
{'label': 5, 'probability': 0.9933018684387207}
```

これで推論サーバーのデプロイが完了しました。Azure では、リソースが存在する限り微量ながら課金されていきますので、不必要なリソースはクリーンアップして終了しましょう。

本章ではハイパーパラメータのチューニングと、IaaS と PaaS を利用したデプロイの方法を紹介してきました。

繰り返しになりますが、機械学習エンジニアは学習部分にだけ注目しがちですが、実際に使用するサービスでは推論部分を使用しており、モデル構築と同様に重要です。クラウドを扱える技術も機械学習エンジニアとして必要になってきていますので、ぜひこの機会に学んでください。

おわりに

本書では、ディープラーニングのモデル構築から、環境構築、最終的にクラウドを用いたデプロイまで行うことができました。活用にフォーカスすると、陽の目が当たりがちなモデル構築ではなく、どのようにデプロイを行うのか、何をイベントのトリガーにして推論まで行うのかを設計する必要があります。イベントのトリガーの役割として最近注目されているのが、 iPaas (integration Platform as a Service) と呼ばれる異なるアプリケーション同士をつなげ、データを統合したりシステムを連携させたりすることのできるクラウドサービスで、有名なサービスとして Zapier や Anyflow があります。弊社でもコーディングすることに執着せず、あくまで問題解決の方法として iPaas を選択したり、コーディングを選択しています。ぜひ、iPaas などのツールとの融合もチャレンジしてみてください。

日進月歩で進化を遂げるディープラーニング界隈では、本書で紹介した内容がベストプラクティスではないかもしれません。しかし、基礎となる知識はいつの時代も無駄になることはありません。学んだ知識をもとにこれからの皆さんの学びの一助になれば幸いです。

最後に、本書を執筆するにあたり、多くの方々のご協力をいただきました。本書の編集に携わった関係諸氏からは、企画段階から常に適切なアドバイスで粘り強くサポートしていただきました。この場をお借りして深くお礼申し上げます。

吉崎亮介、祖父江誠人

索　引

●著者紹介

吉崎亮介（よしざき・りょうすけ）

舞鶴高専専攻科にてシステム制御及び最適化の研究に従事した後、京都大学大学院にて製造業向けの機械学習を用いた製造工程最適化の研究に従事。卒業後、株式会社 SHIFT にてソフトウェアテストの研究開発を経て、株式会社 Carat を共同創業。2017 年 1 月より株式会社キカガクを創業し、代表取締役社長に就任。ビジネスの現場で使える人工知能（AI）を目指し、企業向けの教育事業を手がける。
趣味は人気のサービスを利用してその理由を考察することで、最近は Netflix の韓国ドラマを楽しんで研究中。愛の不時着、梨泰院クラス、スタートアップなど資本主義を感じるテーマがお気に入り。

祖父江誠人（そふえ・まこと）

株式会社キカガク CTO（Chief Technical Officer）。高知大学理学部数学科で解析学を専攻。2018 年同社へ入社。
製造業特化コースの名古屋立ち上げ、無料学習サイト KIKAGAKU のコンテンツ開発、e-ラーニングサービス ikus.ai（イクサイ）を設計から開発まで従事。
機械学習から Web アプリケーション開発まで幅広く担当。趣味は麻雀で、学生時代にネット麻雀で 1 位になったこともある。麻雀仲間募集中。

●スタッフ
- カバーデザイン：岡田 章志
- レイアウトデザイン：田中 幸穂（画房 雪）
- DTP・編集協力：山本 宗宏（株式会社 Green Cherry）
- 編集：武藤 健志（株式会社トップスタジオ）、石川 耕嗣

ディープラーニング実装入門（じっそうにゅうもん）
PyTorch（パイトーチ）による画像·自然言語処理（ガゾウ シゼンゲンゴショリ）

2020年12月21日　初版第1刷発行

著　者　吉崎亮介（よしざきりょうすけ）／祖父江誠人（そ ふ え まこと）
発行人　小川 亨
編集人　高橋隆志
発行所　株式会社インプレス
〒101-0051　東京都千代田区神田神保町一丁目 105 番地
ホームページ　https://book.impress.co.jp/

印刷所　大日本印刷株式会社

978-4-295-01062-3　C3055