

Google Colaboratory の基本的な使い方

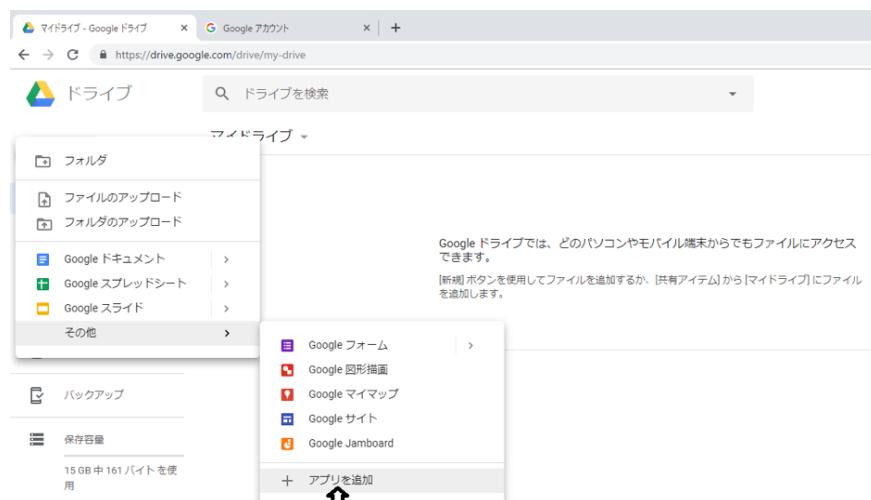
本節は、セミナーインフォ岡田様よりご提供いただきました。Google Colaboratory は以下では **colab** と表記します。

Colab の起動

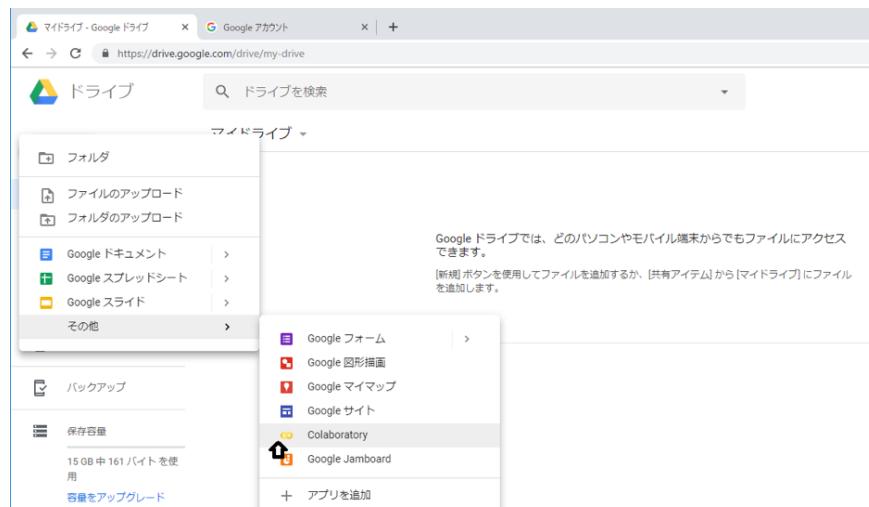
1. [Google Drive](#) の新しいアカウントを作成して下さい。(浅川注: [Gmail](#)などのGoogleアカウントをお持ちであれば無料で利用できます)
2. [Google Drive](#) を起動して、「+新規」（合）をクリックします。



3. Colab が入っていない方は、「+アプリを追加」（合）をクリックで **colab** を追加してください。



4. 以下のように **colab** がでてきますので、「Colaboratory」（合）をクリックして立ち上げて下さい。



5. 立ち上げると以下のような画面が現れます。



デモファイルの実行

1. デモファイルを実行しましょう。以下のファイル名をクリックすると **colab** 上で実行可能です。[2019si_image_recognition_demo.ipynb](#)

デモファイルの内容は以下のとおりです。

```
import numpy as np
from keras.applications.resnet50 import ResNet50
from keras.preprocessing import image
from keras.applications.resnet50 import preprocess_input, decode_predictions

def recognize(image_path='cat.jpg'):
    model = ResNet50(weights='imagenet')
    img = image.load_img(image_path, target_size=(224, 224))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)

    preds = model.predict(x)
    p = decode_predictions(preds)
    for pp in p:
        for pp2 in pp:
            print('{0}:{1:.05f}'.format(pp2[1], pp2[2]))
```

わずか 18 行のデモファイルですが、これだけで画像認識が可能です。最初のこのセルを実行してください。実行方法は jupyter notebook と同様です。すなわち シフトキーを押しながらエンターキーを押すか、セルの左上にある三角形をクリックします。セルの直下に

Using TensorFlow backend.

と表示されれば成功です

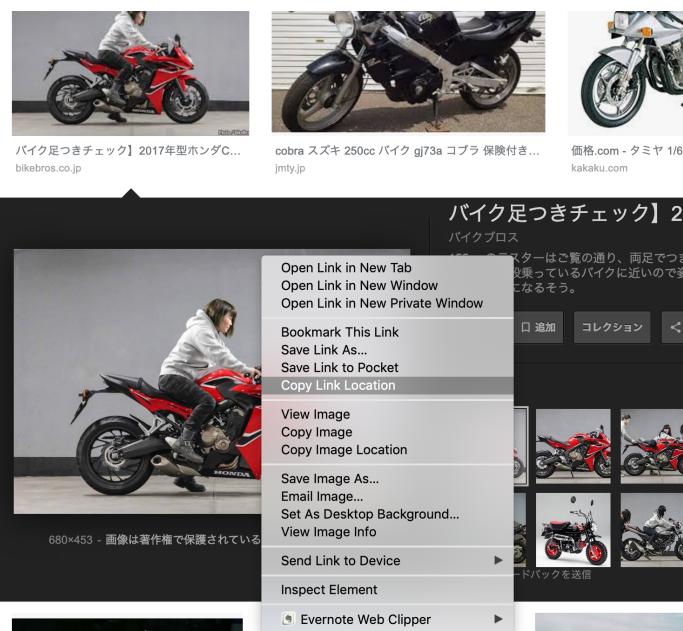
1. [グーグル画像検索](#) でご自身で画像を選んでください。



バイク



ここではバイクの画像を検索してみました。下図のように画像にマウスを合わせてマウスの右ボタンをクリックして、画像のアドレスを取得してください。お使いのパソコンやOSのバージョンにより操作は異なります。図では `Copy Image Location` ですが、「画像の場所を保存する」などと表示される可能性もあります。



選んだ画像の場所の情報から `colab` 上で画像入手します。

Using TensorFlow backend.

```
1 !wget http://img.bikebros.co.jp/vb_img/photo/topics/180213/img/07.jpg
```

上図で赤丸で囲った部分が画像の URL をペーストした部分です。画像の URL の前に `!wget` を入れてからペースとしてください。`!wget` はウェブ上からファイル入手するコマンドです。

この画像を認識させるには、入手した画像のファイル名を指定して `recognize` 関数を呼び出します。上で入手した画像のファイル名は最後のスラッシュ (/) 以降の文字列が画像のファイル名です。上の場合 `07.jpg` が画像のファイル名になります。

認識結果は以下のように表示されました。

```
[5]    1 | recognize('07.jpg').
```

```
↳ crash_helmet:0.50574
moped:0.28435
motor_scooter:0.15286
mountain_bike:0.01785
disk_brake:0.00632
```

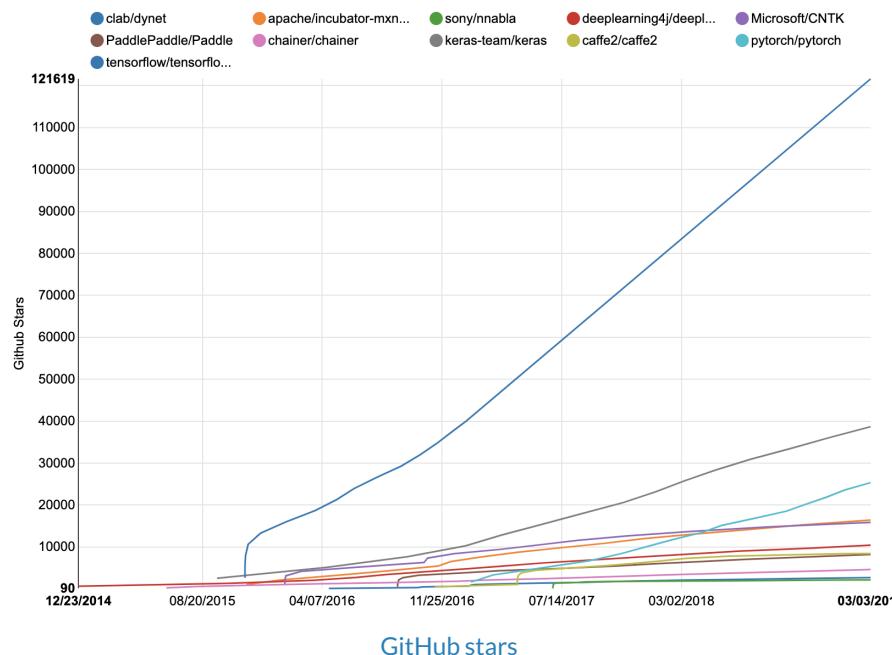
すなわちこの画像は、クラッシュヘルメットである確率が0.50574、次いでモペッド、すなわちエンジン付き自転車で0.15286、第3位がモータースクーターで0.15286でした。

画像の認識精度のみならず、最近のディープラーニングによる画像認識をわずかなコードでデモしてみました。

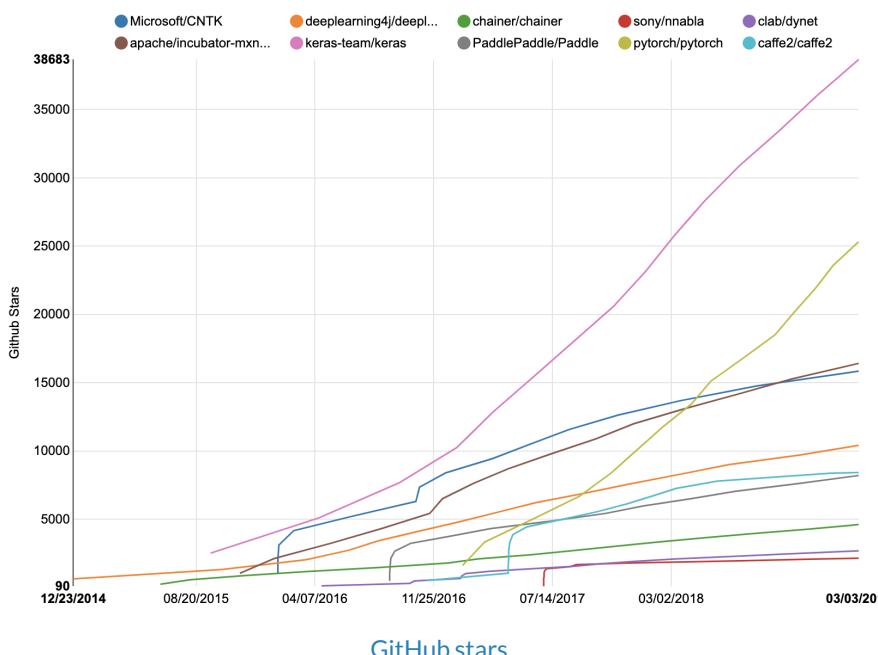
深層学習フレームワーク

Python 上で動作する無料の深層学習フレームワークは数多く提案されています。

下図は主要な深層学習フレームワークについて GitHub の星の数をカウントした結果を示しています。2019年03月03日時点で第2位にトリプルスコアで圧勝しているのがTensorflowです。GitHub 上での星の数の計測ですから、MATLAB や SPSS といった有料のフレームワークは含まれていません。



上図を見ればオープンソースの深層学習フレームワークでは勝負がついていると言っても良いでしょう。TensorFlowだけ特別なので、その他のフレームワークの動向を見るために、TensorFlowを除いて再描画した図が下図になります。上図と下図とでは縦軸の縮尺が異なることに注意してください。



この図で第一位と第二位の線の傾き急であることが読み取れます。この図での第1位が [Keras](#) です。Keras は TensorFlow (と正確にはもう一つ) のフレームワークを背後で利用する仕組みです。第二位は [PyTorch](#) です。PyTorch は柔軟な記述が可能であり研究者のコミュニティでは人気があります。

このページに示した2つの図から判断して、以下の3つの深層学習フレームワークがトレンドでもあると言えるでしょう。

- [TensorFlow](#)
- [Keras](#)
- [PyTorch](#)

上記以外にも [R](#) や JavaScript で機械学習やニューラルネットワークを実行する枠組みも用意されています。[TensorFlow.js](#) は TensorFlow の JavaScript 版です。JavaScript の強みを生かしてブラウザ上から [Node.js](#) で実行可能です。

- [姿勢推定デモ](#)

上記の流れは今回は扱いません。

このページは <https://keras.io/ja/> からの抜粋です。

Kerasとは

Kerasは、Pythonで書かれた、TensorFlowまたはCNTK、Theano上で実行可能な高水準のニューラルネットワークライブラリです。Kerasは、迅速な実験を可能にすることに重点を置いて開発されました。アイデアから結果に到達するまでのリードタイムをできるだけ小さくすることが、良い研究をするための鍵になります。

次のような場合で深層学習ライブラリが必要なら、Kerasを使用してください：

- 容易に素早くプロトタイプの作成が可能（ユーザーフレンドリー、モジュール性、および拡張性による）
- CNNとRNNの両方、およびこれらの2つの組み合わせをサポート
- CPUとGPU上でシームレスな動作

[Keras.io](#)のドキュメントを読んでください。

KerasはPython 2.7-3.6に対応しています。

ガイドライン

- **ユーザーフレンドリー**: Kerasは機械向けでなく、人間向けに設計されたライブラリです。ユーザーインテラクションを前面と中心においています。Kerasは、認知負荷を軽減するためのベストプラクティスをフォローします。一貫したシンプルなAPI群を提供し、一般的な使用事例で要求されるユーザーアクションを最小限に抑え、ユーザーエラー時に明確で実用的なフィードバックを提供します。
- **モジュール性**: モデルとは、できるだけ制約の少ない接続が可能で、独立した、完全に設定可能なモジュールの、シーケンスまたはグラフとして理解されています。特に、ニューラルネットワークの層、損失関数、最適化、初期化、活性化関数、正規化はすべて、新しいモデルを作成するための組み合わせ可能な、独立したモジュールです。
- **拡張性**: 新しいモジュールが（新しいクラスや関数として）簡単に追加できます。また、既存のモジュールには多くの実装例があります。新しいモジュールを容易に作成できるため、あらゆる表現が可能になっています。このことからKerasは先進的な研究に適しています。
- **Pythonで実装**: 声明形式の設定ファイルを持ったモデルはありません。モデルはPythonコードで記述されています。このPythonコードは、コンパクトで、デバッグが容易で、簡単に拡張できます。

30秒でKerasに入門しましょう。

Kerasの中心的なデータ構造は`_model_`で、レイヤーを構成する方法です。主なモデルは `Sequential` モデルで、レイヤーの線形スタックです。更に複雑なアーキテクチャの場合は、`Keras functional API`を使用する必要があります。これでレイヤーのなす任意のグラフが構築可能になります。

`Sequential` モデルの一例を見てみましょう。

```
from keras.models import Sequential  
model = Sequential()
```

`.add()` で簡単にレイヤーを積み重ねることができます：

```
from keras.layers import Dense  
  
model.add(Dense(units=64, activation='relu', input_dim=100))  
model.add(Dense(units=10, activation='softmax'))
```

実装したモデルがよさそうなら `.compile()` で訓練プロセスを設定しましょう。

```
model.compile(loss='categorical_crossentropy',  
              optimizer='sgd',  
              metrics=['accuracy'])
```

必要に応じて、最適化アルゴリズムも設定できます。Kerasの中心的な設計思想は、ユーザーが必要なときに完全にコントロール（ソースコードの容易な拡張性を実現する究極のコントロール）できる一方で、適度に単純にすることです。

```
model.compile(loss=keras.losses.categorical_crossentropy,  
              optimizer=keras.optimizers.SGD(lr=0.01, momentum=0.9, nesterov=True))
```

訓練データをミニバッチで繰り返し処理できます。

```
# x_train and y_train are Numpy arrays --just like in the Scikit-Learn API.  
model.fit(x_train, y_train, epochs=5, batch_size=32)
```

代わりに、バッチサイズを別に規定できます。

```
model.train_on_batch(x_batch, y_batch)
```

1行でモデルの評価ができます。

```
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

また、新しいデータに対して予測もできます:

```
classes = model.predict(x_test, batch_size=128)
```

質問応答システムや画像分類、ニューラルチューリングマシン、word2vecやその他多くのモデルは高速かつシンプルに実装可能です。深層学習の根底にあるアイデアはとてもシンプルです。実装もシンプルであるべきではないでしょうか？

Kerasについてのより詳細なチュートリアルについては、以下を参照してください。

- [Getting started with the Sequential model](#)
- [Getting started with the functional API](#)

リポジトリの[examples folder](#)にはさらに高度なモデルがあります。メモリネットワークを用いた質問応答システムや積層LSTMを用いた文章生成などです。

```
<!--
```

インストール

Kerasをインストールする前にKerasのバックエンドをインストールしてください：TensorFlowやTheano、CNTKがあります。TensorFlowを推奨しています。

- [TensorFlow installation instructions.](#)
- [Theano installation instructions.](#)
- [CNTK installation instructions.](#)

次のようなオプショナルな依存のインストールも考慮してもいいかもしれません。

- cuDNN (KerasをGPUで動かす場合は推奨) .

- HDF5とh5py (Kerasのモデルをディスクに保存する場合は必須) .
- graphvizとpydot (可視化でモデルのグラフ描画に利用) .

これでKerasをインストールできるようになりました。Kerasをインストールするには2つの方法があります：

- **PyPIからKerasをインストール（推奨）：**

```
sudo pip install keras
```

virtualenvを利用している場合、sudoを回避できます：

```
pip install keras
```

- **代替方法：GithubソースからKerasをインストール：**

まず、[git](#)でKerasをクローンします：

```
git clone https://github.com/keras-team/keras.git
```

そして、Kerasのフォルダに[cd](#)してインストールコマンドを実行します。

```
cd keras  
sudo python setup.py install
```

TensorFlowからCNTKやTheanoへの変更

デフォルトでは、KerasはTensorFlowをテンソル計算ライブラリとしています。Kerasのバックエンドを設定するには、[この手順](#)に従ってください。

-->

サポート

質問をしたり、開発に関するディスカッションに参加できます：

- Keras Google group上で
- Keras Slack channel上で。チャンネルへのリクエストするには[このリンク](#)を使ってください。

Githubのissuesにバグレポートや機能リクエストを投稿できます。まず[ガイドライン](#)を必ず読んでください。

どうしてこのライブラリにKerasという名前を付けたのですか？

Keras (*κέρας*) はギリシア語で角を意味します。古代ギリシア文学およびラテン文学における文学上の想像がこの名前の由来です。最初にこの想像が見つかったのは_Odyssey_で、夢の神 (*Oneiroi*, 単数形 *Oneiros*) は、象牙の門を通って地上に訪れて偽りのビジョンで人々を騙す神と、角の門を通って地上に訪れて起こるはずの未来を知らせる神とに分かれているそうです。これは *κέρας* (角) / *κραίνω* (遂行) と *ἐλέφας* (象牙) / *ἐλεφαίρομαι* (欺瞞) の似た響きを楽しむ言葉遊びです。

Kerasは当初プロジェクトONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System) の研究の一環として開発されました。

"Oneiroi are beyond our unravelling --who can be sure what tale they tell? Not all that men look for comes to pass. Two gates there are that give passage to fleeting Oneiroi; one is made of horn, one of ivory. The Oneiroi that pass through sawn ivory are deceitful, bearing a message that will not be fulfilled; those that come out through polished horn have truth behind them, to be accomplished for men who see them." Homer, Odyssey 19. 562 ff (Shewring translation).

なぜKerasを使うか?

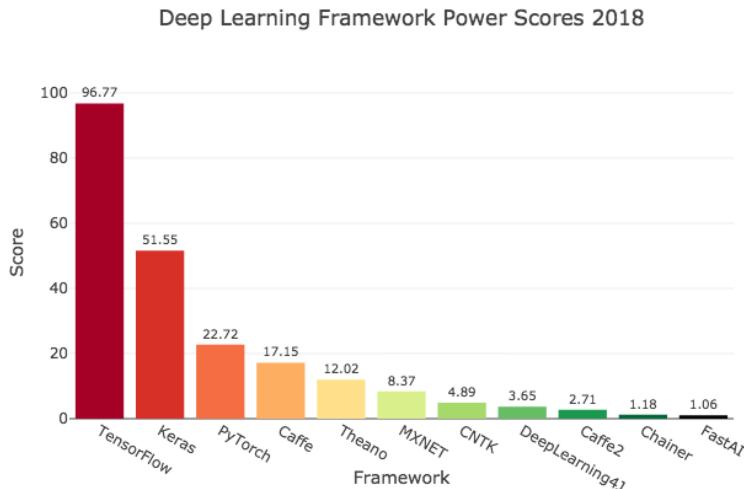
このページは <https://keras.io/ja/why-use-keras/> からの抜粋です

今日、数え切れない数の深層学習フレームワークが存在します。なぜ他のライブラリではなくて、Kerasを使うのでしょうか？ここでは、Kerasが既存の選択肢に引けを取らない理由のいくつかを紹介します。

Kerasはシンプルかつ柔軟に使用できます

- Kerasは、機械ではなく、人間のために設計されたAPIです。 [Kerasは認知的負荷を軽減するためのベストプラクティスに従っています](#): 一貫性のあるシンプルなAPIを提供し、一般的なユースケースで必要なユーザーの操作を最小限に抑え、エラー時には明確で実用的なフィードバックを提供します。
- これにより、Kerasは簡単に学ぶことが出来て、簡単に使う事が出来ます。Kerasユーザーは、生産性が高く、競争相手よりも、より多くのアイデアを試す事が出来ます。-- これにより、[機械学習のコンテストで勝つのに役立ちます](#)。
- 手軽さがあっても、柔軟性がなければいけません: Kerasは低レベルな深層学習言語（特にTensorFlow）と統合しているので、基本の深層学習言語で構築されたものを実装する事が出来ます。特に、[`tf.keras`](#)として、Keras APIはTensorFlowワークフローとシームレスに統合されています。

Kerasは事業と研究コミュニティの両方で幅広く使用されています



Deep learning frameworks ranking computed by Jeff Hale, based on 11 data sources across 7 categories

2018年中旬の時点では、Kerasは25万以上の個人ユーザーがあり、TensorFlow自体を除いて、他の深層学習フレームワークよりも事業と研究コミュニティの両方で多く採用されています（そしてKeras APIは[`tf.keras`](#)を経由することでTensorFlowの公式なフロントエンドとなっています）。

あなたはすでにKerasで構築された機能を日常で使用しています -- Netflix, Uber, Yelp, Instacart, Zocdoc, Squareなど多くの企業がKerasを使用しています。特に、自社製品の核となる部分で深層学習を用いているようなスタートアップ企業で人気があります。

また, Kerasは, 深層学習研究者の間でも人気があり, プレプリント・サーバarXiv.orgにアップロードされた, 科学技術論文で言及されているフレームワークの中で二番目に使用されています。また, Kerasは, 大規模な科学機関, 例えば, CERNやNASAの研究者によって採用されています。

Kerasは簡単にモデルを製品化できます

Kerasのモデルは, 他の深層学習フレームワークよりも多くのプラットフォームで, 簡単にデプロイできます。

- iOS ([Apple's CoreML](#)経由, Kerasのサポートは正式にAppleから提供されています) . こちらが[チュートリアル](#)です.
 - Android (TensorFlow Androidランタイム経由) 例: [Not Hotdog app](#).
 - ブラウザ ([Keras.js](#)や, [WebDNN](#)などのGPU利用が可能なJavaScriptランタイム経由)
 - Google Cloud ([TensorFlow-Serving](#)経由)
 - ([Flask](#)アプリのような) Pythonのウェブアプリのバックエンド.
 - JVM ([SkyMind](#)によって提供されたDL4J モデル経由)
 - ラズベリーパイ
-

Kerasは複数のバックエンドをサポートし, 一つのエコシステムに縛られません

Kerasは複数のバックエンドエンジンをサポートしています。重要な事に, 組み込みレイヤーのみで構成されるKerasモデルは, 全てのバックエンド間で移植可能です。:一つのバックエンドを使用して学習したモデルを用いて, 別のバックエンドを使用してモデルをロードする事が出来ます。 (例えば, デプロイなどで用いる事が出来ます) 使用可能なバックエンドは以下のとおりです。

- TensorFlow バックエンド (from Google)
- CNTK バックエンド (from Microsoft)
- Theano バックエンド

Amazonも現在, KerasのMXNetバックエンドの開発にも取り組んでいます。

また, KerasモデルはCPU以外の様々なハードウェアプラットフォームで学習する事が出来ます。

- [NVIDIA GPUs](#)
 - [Google TPUs](#) (TensorFlowバックエンドかつ, Google Cloud経由)
 - OpenGLが使用出来るAMDのようなGPU ([the PlaidML Keras](#)バックエンド経由)
-

Kerasは複数のGPU, 分散学習のサポートが強力です

- Kerasは[複数GPU並列処理](#)のための組み込みサポートもあります。
 - Uberの[Horovod](#)は, Kerasモデルを最もサポートしています。
 - Kerasモデルは[TensorFlow Estimators](#)に変換する事が出来ます。また, [Google CloudのGPUクラスター](#)を用いて学習が出来ます。
 - Kerasは[Dist-Keras](#) (from CERN) と [Elephas](#)経由でSpark上で走らせる事が出来ます。
-

Kerasの開発は深層学習の主要企業によってサポートされています

Kerasの開発は主にGoogleによってサポートされ, Keras APIはTensorFlowに `tf.keras` としてパッケージ化されています。加えて, MicrosoftはCNTK Kerasバックエンドを管理しています。Amazon AWSはMXNetサポートを開発中です。その他, NVIDIA, Uber, Apple (CoreML) によって, サポートされています。



SequentialモデルでKerasを始めてみよう

`Sequential` (系列) モデルは層を積み重ねたものです。

`Sequential` モデルはコンストラクタにレイヤーのインスタンスのリストを与えることで作れます:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

`.add()` メソッドで簡単にレイヤーを追加できます。

```
model = Sequential()
model.add(Dense(32, input_dim=784))
model.add(Activation('relu'))
```

入力のshapeを指定する

モデルはどのような入力のshapeを想定しているのかを知る必要があります。このため、

`Sequential` モデルの最初のレイヤーに入力のshapeについての情報を与える必要があります（最初のレイヤー以外は入力のshapeを推定できるため、指定する必要はありません）。入力のshapeを指定する方法はいくつかあります：

- 最初のレイヤーの `input_shape` 引数を指定する。この引数にはshapeを示すタプルを与えます（このタプルの要素は整数か `None` を取ります。`None` は任意の正の整数を期待することを意味します）。
- `Dense` のような2次元の層では `input_dim` 引数を指定することで入力のshapeを指定できます。同様に、3次元のレイヤーでは `input_dim` 引数と `input_length` 引数を指定することで入力のshapeを指定できます。
- (stateful recurrent networkなどで) バッチサイズを指定したい場合、`batch_size` 引数を指定することができます。もし、`batch_size=32` と `input_shape=(6, 8)` を同時に指定した場合、想定されるバッチごとの入力のshapeは `(32, 6, 8)` となります。

このため、次のコードは等価となります。

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
```

```
model = Sequential()
model.add(Dense(32, input_dim=784))
```

コンパイル

モデルの学習を始める前に、`compile` メソッドを用いどのような学習処理を行なうかを設定する必要があります。`compile` メソッドは3つの引数を取ります：

- 最適化アルゴリズム: 引数として、定義されている最適化手法の識別子を文字列として与える (`rmsprop` や `adagrad` など)，もしくは `Optimizer` クラスのインスタンスを与えることができます

ます。参考: [最適化](#)

- 損失関数: モデルが最小化しようとする目的関数です。引数として、定義されている損失関数の識別子を文字列として与える (`categorical_crossentropy` や `mse` など)、もしくは目的関数を関数として与えることができます。参考: [損失関数](#)
- 評価関数のリスト: 分類問題では精度として `metrics=['accuracy']` を指定したくなるでしょう。引数として、定義されている評価関数の識別子を文字列として与える、もしくは自分で定義した関数を関数として与えることができます。

```
# マルチクラス分類問題の場合
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 2値分類問題の場合
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# 平均二乗誤差を最小化する回帰問題の場合
model.compile(optimizer='rmsprop',
              loss='mse')

# 独自定義の評価関数を定義
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

訓練

KerasのモデルはNumpy配列として入力データとラベルデータから訓練します。モデルを訓練するときは、一般に `fit` 関数を使います。ドキュメントは[こちら](#)。

```
# 1つの入力から2クラス分類をするモデルにおいては

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# ダミーデータの作成
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

# 各イテレーションのバッチサイズを32で学習を行なう
model.fit(data, labels, epochs=10, batch_size=32)
```

```
# 1つの入力から10クラスの分類を行なう場合について（カテゴリ分類）

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# ダミーデータ作成
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# ラベルデータをカテゴリの1-hotベクトルにエンコードする
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# 各イテレーションのバッチサイズを32で学習を行なう
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

例

いますぐKerasを始められるようにいくつか例を用意しました！

examples folder フォルダにはリアルデータセットを利用したモデルがあります.

- CIFAR10 小規模な画像分類：リアルタイムなdata augmentationを用いたConvolutional Neural Network (CNN)
- IMDB 映画レビューのセンチメント分類：単語単位のLSTM
- Reuters 記事のトピック分類：多層パーセプトロン (MLP)
- MNIST 手書き文字認識：MLPとCNN
- LSTMを用いた文字単位の文章生成

...など

多層パーセプトロン (MLP) を用いた多値分類:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# ダミーデータ生成
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)

model = Sequential()
# Dense(64) は、64個のhidden unitを持つ全結合層です。
# 最初のlayerでは、想定する入力データshapeを指定する必要があります、ここでは20次元としてます。
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

MLPを用いた二値分類:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout

# 疑似データの生成
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))

model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

VGG-likeなconvnet

```

import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD

# 疑似データ生成
x_train = np.random.random((100, 100, 100, 3))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)
x_test = np.random.random((20, 100, 100, 3))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(20, 1)), num_classes=10)

model = Sequential()
# 入力: サイズが100x100で3チャンネルをもつ画像 -> (100, 100, 3) のテンソル
# それぞれのlayerで3x3の畳み込み処理を適用している
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(100, 100, 3)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

model.fit(x_train, y_train, batch_size=32, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=32)

```

LSTMを用いた系列データ分類:

```

from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, output_dim=256))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)

```

1D Convolutionを用いた系列データ分類:

```

from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D

model = Sequential()
model.add(Conv1D(64, 3, activation='relu', input_shape=(seq_length, 100)))
model.add(Conv1D(64, 3, activation='relu'))
model.add(MaxPooling1D(3))
model.add(Conv1D(128, 3, activation='relu'))
model.add(Conv1D(128, 3, activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

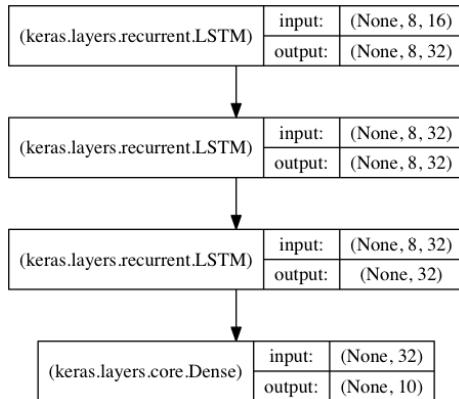
model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)

```

Stacked LSTMを用いた系列データ分類

このモデルは、3つのLSTM layerを繋げ、高等表現を獲得できるような設計となっています。

最初の2つのLSTMは出力系列をすべて出力しています。しかし、最後のLSTMは最後のステップの状態のみを出力しており、データの次元が落ちています(入力系列を一つのベクトルにしているようなものです)。



```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10

# 想定する入力データshape: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
              input_shape=(timesteps, data_dim))) # 32次元のベクトルのsequenceを出力する
model.add(LSTM(32, return_sequences=True)) # 32次元のベクトルのsequenceを出力する
model.add(LSTM(32)) # 32次元のベクトルを一つ出力する
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# 疑似訓練データを生成する
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, num_classes))

# 疑似検証データを生成する
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, num_classes))

model.fit(x_train, y_train,
          batch_size=64, epochs=5,
          validation_data=(x_val, y_val))
```

同じようなStacked LSTMを"stateful"にする

Stateful recurrent modelは、バッチを処理し得られた内部状態を次のバッチの内部状態の初期値として再利用するモデルの一つです。このため、計算複雑度を調整できるようにしたまま、長い系列を処理することができるようになりました。

[FAQにもstateful RNNsについての情報があります](#)

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10
batch_size = 32

# 想定している入力バッチshape: (batch_size, timesteps, data_dim)
# 注意: ネットワークがstatefulであるため, batch_input_shapeをすべてうめて与えなければなりません
# バッチkのi番目のサンプルは、バッチk-1のi番目のサンプルの次の時系列となります。
model = Sequential()
model.add(LSTM(32, return_sequences=True, stateful=True,
              batch_input_shape=(batch_size, timesteps, data_dim)))
model.add(LSTM(32, return_sequences=True, stateful=True))
model.add(LSTM(32, stateful=True))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# 疑似訓練データを生成
x_train = np.random.random((batch_size * 10, timesteps, data_dim))
y_train = np.random.random((batch_size * 10, num_classes))

# 疑似検証データを生成
x_val = np.random.random((batch_size * 3, timesteps, data_dim))
y_val = np.random.random((batch_size * 3, num_classes))

model.fit(x_train, y_train,
          batch_size=batch_size, epochs=5, shuffle=False,
          validation_data=(x_val, y_val))
```

functional APIでKerasを始めてみよう

functional APIは、複数の出力があるモデルや有向非巡回グラフ、共有レイヤーを持ったモデルなどの複雑なモデルを定義するためのインターフェースです。

ここでは `Sequential` モデルについて既に知識があることを前提として説明します。

シンプルな例から見てきましょう。

例1: 全結合ネットワーク

下記のネットワークは `Sequential` モデルによっても定義可能ですが、functional APIを使ったシンプルな例を見てきましょう。

- レイヤーのインスタンスは関数呼び出し可能で、戻り値としてテンソルを返します
- `Model` を定義することで入力と出力のテンソルは接続されます
- 上記で定義したモデルは `Sequential` と同様に利用可能です

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels) # starts training
```

全てのモデルはレイヤーと同じように関数呼び出し可能です

functional APIを利用することで、訓練済みモデルの再利用が簡単になります：全てのモデルを、テンソルを引数としたlayerのように扱うことができます。これにより、モデル構造だけでなく、モデルの重みも再利用できます。

```
x = Input(shape=(784,))
# This works, and returns the 10-way softmax we defined above.
y = model(x)
```

一連のシーケンスを処理するモデルを簡単に設計できます。例えば画像識別モデルをたった1行で動画識別モデルに応用できます。

```
from keras.layers import TimeDistributed

# Input tensor for sequences of 20 timesteps,
# each containing a 784-dimensional vector
input_sequences = Input(shape=(20, 784))

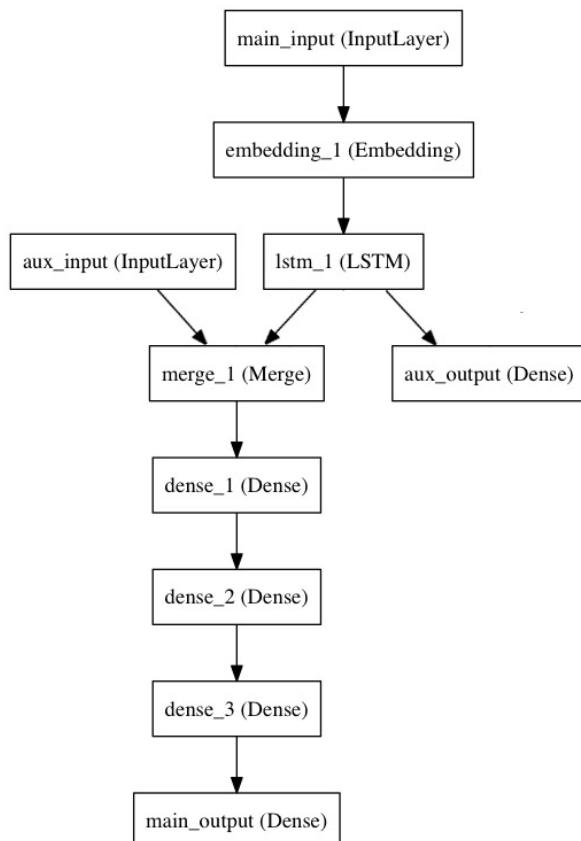
# This applies our previous model to every timestep in the input sequences.
# the output of the previous model was a 10-way softmax,
# so the output of the layer below will be a sequence of 20 vectors of size 10.
processed_sequences = TimeDistributed(model)(input_sequences)
```

多入力多出力モデル

functional APIは複数の入出力を持ったモデルに最適です。複数の複雑なデータストリームを簡単に扱うことができます。

Twitterの新しいニュースヘッドラインを受信した際、そのツイートのリツイートやライクの回数を予測する例を考えます。主な入力はヘッドラインの単語のシーケンスですが、スペースとして、ヘッドラインの投稿時間などのデータを入力として追加します。このモデルは2つの損失関数によって訓練されます。モデルにおける初期の主損失関数を使うことは、深い層を持つモデルにとっては良い正則化の構造です。

以下がモデルの図になります。



functional APIを利用してこのネットワークを実装してみましょう。

main inputはヘッドラインを整数のシーケンス（それぞれの整数は単語をエンコードしたしたもの）として受け取ります。整数の範囲は1から10000となり（単語数は10000語），各シーケンスは長さ100単語で構成されます。

```
from keras.layers import Input, Embedding, LSTM, Dense
from keras.models import Model

# Headline input: meant to receive sequences of 100 integers, between 1 and 10000.
# Note that we can name any layer by passing it a "name" argument.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# This embedding layer will encode the input sequence
# into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

# A LSTM will transform the vector sequence into a single vector,
# containing information about the entire sequence
lstm_out = LSTM(32)(x)
```

ここでは補助損失を追加し、LSTMとEmbeddingレイヤーをスムーズに訓練できるようにしますが、モデルでは主損失がはるかに高くなります。

```
auxiliary_output = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)
```

この時点では、auxiliary_inputをLSTM出力と連結してモデルに入力します。

```

auxiliary_input = Input(shape=(5,), name='aux_input')
x = keras.layers.concatenate([lstm_out, auxiliary_input])

# We stack a deep densely-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# And finally we add the main logistic regression layer
main_output = Dense(1, activation='sigmoid', name='main_output')(x)

```

2つの入力と2つの出力を持ったモデルを定義します。

```
model = Model(inputs=[main_input, auxiliary_input], outputs=[main_output, auxiliary_output])
```

モデルをコンパイルし、補助損失に0.2の重み付けを行います。様々な `loss_weights` や `loss` を対応付けるためにリストもしくは辞書を利用します。`loss` に1つの損失関数を与えた場合、全ての出力に対して同一の損失関数が適用されます。

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
              loss_weights=[1., 0.2])
```

モデルに入力と教師データをリストで渡すことで訓練できます。

```
model.fit([headline_data, additional_data], [labels, labels],
          epochs=50, batch_size=32)
```

入力と出力に名前付けを行っていれば（"name"引数を利用），下記のような方法でモデルをコンパイルできます。

```

model.compile(optimizer='rmsprop',
              loss={'main_output': 'binary_crossentropy', 'aux_output': 'binary_crossentropy'},
              loss_weights={'main_output': 1., 'aux_output': 0.2})

# And trained it via:
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': labels, 'aux_output': labels},
          epochs=50, batch_size=32)

```

共有レイヤー

他のfunctional APIの利用例として、共有レイヤーがあります。共有レイヤーについて考えてみましょう。

ツイートのデータセットの例を考えてみましょう。2つのツイートが同じ人物からつぶやかれたかどうかを判定するモデルを作りたいとします。（例えばこれによりユーザーの類似度を比較できます）

これを実現する一つの方法として、2つのツイートを2つのベクトルにエンコードし、それらをマージした後、ロジスティクス回帰を行うことで、その2つのツイートが同じ人物から投稿されたかどうかの確率を出力できます。このモデルはポジティブなツイートのペアとネガティブなツイートのペアを用いて訓練できます。

問題はシンメトリックであるため、1つめのツイートのエンコードメカニズムは2つめのツイートのエンコード時に再利用出来ます。ここではLSTMの共有レイヤーによりツイートをエンコードします。

functional APIでこのモデルを作成してみましょう。入力として `(280, 256)` のバイナリーフィールドをとります。サイズが256の280個のシーケンスで、256次元のベクトルの各次元は文字（アルファベット以外も含めた256文字の出現頻度の高いもの）の有無を表します。

```

import keras
from keras.layers import Input, LSTM, Dense
from keras.models import Model

tweet_a = Input(shape=(280, 256))
tweet_b = Input(shape=(280, 256))

```

それぞれの入力間でレイヤーを共有するために、1つのレイヤーを生成し、そのレイヤーを用いて複数の入力を処理します。

```

# This layer can take as input a matrix
# and will return a vector of size 64
shared_lstm = LSTM(64)

# When we reuse the same layer instance
# multiple times, the weights of the layer
# are also being reused
# (it is effectively *the same* layer)
encoded_a = shared_lstm(tweet_a)
encoded_b = shared_lstm(tweet_b)

# We can then concatenate the two vectors:
merged_vector = keras.layers.concatenate([encoded_a, encoded_b], axis=-1)

# And add a logistic regression on top
predictions = Dense(1, activation='sigmoid')(merged_vector)

# We define a trainable model linking the
# tweet inputs to the predictions
model = Model(inputs=[tweet_a, tweet_b], outputs=predictions)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.fit([data_a, data_b], labels, epochs=10)

```

共有レイヤーの出力や出力のshapeを見てみましょう。

"ノード"の概念

ある入力を用いてレイヤーを関数呼び出しするときは常に新しいテンソル（レイヤーの出力）を生成しており、レイヤーにノードを追加すると入力のテンソルと出力のテンソルはリンクされます。同じレイヤーを複数回呼び出す際、そのレイヤーは0, 1, 2...とインデックスされた複数のノードを所有することになります。

以前のバージョンのKerasでは、`layer.get_output()`によって出力のテンソルを取得でき、`layer.output_shape`によって形を取得できました。もちろん現在のバージョンでもこれらは利用可能です（`get_output()`は`output`というプロパティーに変更されました）。しかし複数の入力が接続されているレイヤーはどうしたらよいでしょうか？

1つのレイヤーに1つの入力しかない場合は問題はなく`.output`がレイヤーが单一の出力を返すでしょう。

```

a = Input(shape=(280, 256))

lstm = LSTM(32)
encoded_a = lstm(a)

assert lstm.output == encoded_a

```

複数の入力がある場合はそうはなりません。

```

a = Input(shape=(280, 256))
b = Input(shape=(280, 256))

lstm = LSTM(32)
encoded_a = lstm(a)
encoded_b = lstm(b)

lstm.output

>> AttributeError: Layer lstm_1 has multiple inbound nodes,
hence the notion of "layer output" is ill-defined.
Use `get_output_at(node_index)` instead.

```

下記は正常に動作します。

```
assert lstm.get_output_at(0) == encoded_a
assert lstm.get_output_at(1) == encoded_b
```

シンプルですね。

`input_shape` と `output_shape` についても同じことが言えます。レイヤーが1つのノードしか持っていない、もしくは全てのノードが同じ入出力のshapeであれば、レイヤーの入出力のshapeが一意に定まり、`layer.output_shape` / `layer.input_shape` によって1つのshapeを返します。しかしながら、1つの `Conv2D` レイヤーに `(32, 32, 3)` の入力と `(64, 64, 32)` の入力を行った場合、そのレイヤーは複数のinput/output shapeを持つことになるため、それぞれのshapeはノードのインデックスを指定することで取得できます。

```
a = Input(shape=(32, 32, 3))
b = Input(shape=(64, 64, 3))

conv = Conv2D(16, (3, 3), padding='same')
conved_a = conv(a)

# Only one input so far, the following will work:
assert conv.input_shape == (None, 32, 32, 3)

conved_b = conv(b)
# now the `input_shape` property wouldn't work, but this does:
assert conv.get_input_shape_at(0) == (None, 32, 32, 3)
assert conv.get_input_shape_at(1) == (None, 64, 64, 3)
```

その他の例

コード例から学び始めることは最良の手法です。その他の例も見てみましょう。

Inception module

Inceptionモデルについての詳細は[Going Deeper with Convolutions](#)を参照。

```
from keras.layers import Conv2D, MaxPooling2D, Input

input_img = Input(shape=(256, 256, 3))

tower_1 = Conv2D(64, (1, 1), padding='same', activation='relu')(input_img)
tower_1 = Conv2D(64, (3, 3), padding='same', activation='relu')(tower_1)

tower_2 = Conv2D(64, (1, 1), padding='same', activation='relu')(input_img)
tower_2 = Conv2D(64, (5, 5), padding='same', activation='relu')(tower_2)

tower_3 = MaxPooling2D((3, 3), strides=(1, 1), padding='same')(input_img)
tower_3 = Conv2D(64, (1, 1), padding='same', activation='relu')(tower_3)

output = keras.layers.concatenate([tower_1, tower_2, tower_3], axis=1)
```

Residual connection on a convolution layer

Residual networksモデルについての詳細は[Deep Residual Learning for Image Recognition](#)を参照してください。

```
from keras.layers import Conv2D, Input

# input tensor for a 3-channel 256x256 image
x = Input(shape=(256, 256, 3))
# 3x3 conv with 3 output channels (same as input channels)
y = Conv2D(3, (3, 3), padding='same')(x)
# this returns x + y.
z = keras.layers.add([x, y])
```

Shared vision model

このモデルでは、2つのMNISTの数字が同じものかどうかを識別するために、同じ画像処理のモジュールを2つの入力で再利用しています。

```

from keras.layers import Conv2D, MaxPooling2D, Input, Dense, Flatten
from keras.models import Model

# First, define the vision modules
from keras.layers import Conv2D, MaxPooling2D, Input, Dense, Flatten
from keras.models import Model

# First, define the vision modules
digit_input = Input(shape=(27, 27, 1))
x = Conv2D(64, (3, 3))(digit_input)
x = Conv2D(64, (3, 3))(x)
x = MaxPooling2D((2, 2))(x)
out = Flatten()(x)

vision_model = Model(digit_input, out)

# Then define the tell-digits-apart model
digit_a = Input(shape=(27, 27, 1))
digit_b = Input(shape=(27, 27, 1))

# The vision model will be shared, weights and all
out_a = vision_model(digit_a)
out_b = vision_model(digit_b)

concatenated = keras.layers.concatenate([out_a, out_b])
out = Dense(1, activation='sigmoid')(concatenated)

classification_model = Model([digit_a, digit_b], out)

```

Visual question answering model

このモデルは写真に対する自然言語の質問に対して1単語の解答を選択できます。

質問と画像をそれぞれベクトルにエンコードし、それらを1つに結合して、解答となる語彙を正解データとしたロジスティック回帰を訓練させることで実現できます。

```

from keras.layers import Conv2D, MaxPooling2D, Flatten
from keras.layers import Input, LSTM, Embedding, Dense
from keras.models import Model, Sequential

# First, let's define a vision model using a Sequential model.
# This model will encode an image into a vector.
vision_model = Sequential()
vision_model.add(Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=(224, 224, 3)))
vision_model.add(Conv2D(64, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(128, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Flatten())

# Now let's get a tensor with the output of our vision model:
image_input = Input(shape=(224, 224, 3))
encoded_image = vision_model(image_input)

# Next, let's define a language model to encode the question into a vector.
# Each question will be at most 100 word long,
# and we will index words as integers from 1 to 9999.
question_input = Input(shape=(100,), dtype='int32')
embedded_question = Embedding(input_dim=10000, output_dim=256, input_length=100)(question_input)
encoded_question = LSTM(256)(embedded_question)

# Let's concatenate the question vector and the image vector:
merged = keras.layers.concatenate([encoded_question, encoded_image])

# And let's train a logistic regression over 1000 words on top:
output = Dense(1000, activation='softmax')(merged)

# This is our final model:
vqa_model = Model(inputs=[image_input, question_input], outputs=output)

# The next stage would be training this model on actual data.

```

Video question answering model

画像のQAモデルを訓練したので、そのモデルを応用して動画のQA modelを作成してみましょう。適切な訓練を行うことで、短い動画や（例えば、100フレームの人物行動）や動画を用いた自然言語のQAへ応用することができます（例えば、「その少年は何のスポーツをしていますか？」 「サッカーです」）。

```
from keras.layers import TimeDistributed

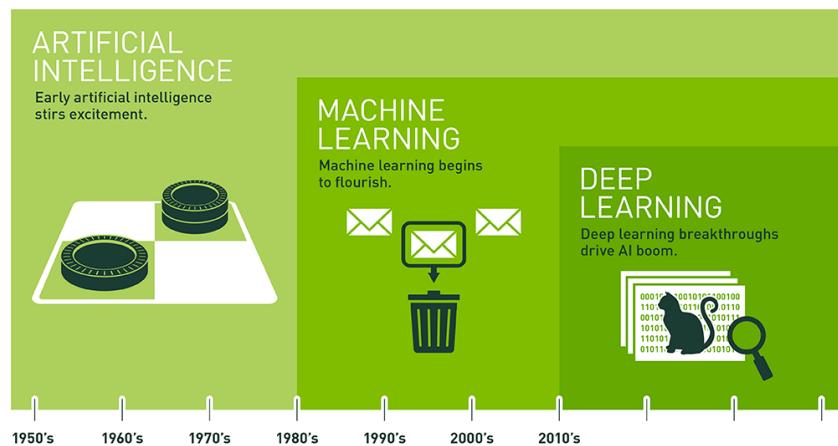
video_input = Input(shape=(100, 224, 224, 3))
# This is our video encoded via the previously trained vision_model (weights are reused)
encoded_frame_sequence = TimeDistributed(vision_model)(video_input) # the output will be a sequence
encoded_video = LSTM(256)(encoded_frame_sequence) # the output will be a vector

# This is a model-level representation of the question encoder, reusing the same weights as before
question_encoder = Model(inputs=question_input, outputs=encoded_question)

# Let's use it to encode the question:
video_question_input = Input(shape=(100,), dtype='int32')
encoded_video_question = question_encoder(video_question_input)

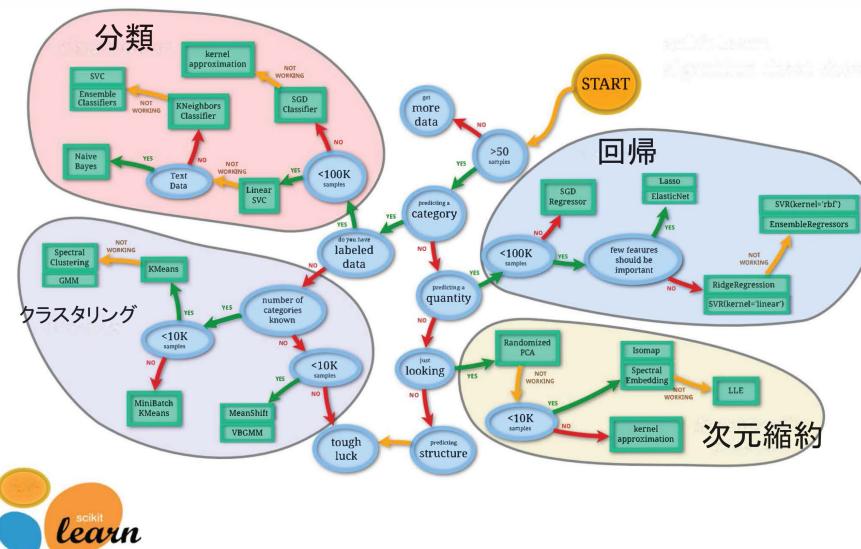
# And this is our video question answering model:
merged = keras.layers.concatenate([encoded_video, encoded_video_question])
output = Dense(1000, activation='softmax')(merged)
video_qa_model = Model(inputs=[video_input, video_question_input], outputs=output)
```

人工知能, 機械学習, ニューラルネットワーク

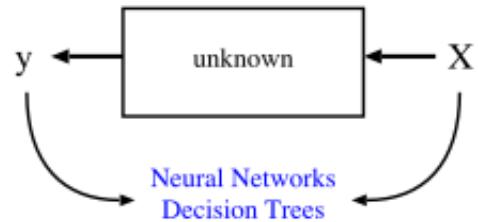
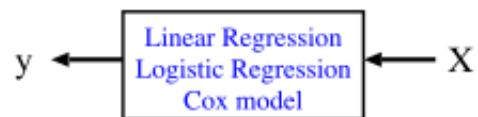
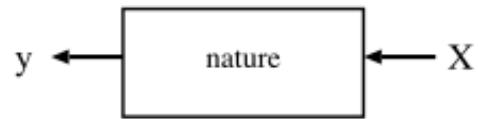


<https://developer.nvidia.com/deep-learning>

分類, 回帰, 次元圧縮, 視覚化



統計学と機械学習



Breiman(2001)[^1]

[^1]: Breiman, Leo, Statistical Modeling: The Two Cultures, **Statistical Science**, 2001, Vol. 16, No. 3, 199–231.

統計的仮説検定と機械学習

- **機械学習**: ルールベースのプログラミングではなくデータから学習するアルゴリズム
- **統計的モデリング**: 数学的方程式を用いてデータ内の変数間の関係を定式化

出典 <https://www.kdnuggets.com/2016/11/machine-learning-vs-statistics.html>

| 統計学 | 機械学習 |
|------|--------|
| 推定 | 学習 |
| 分類器 | 仮説 |
| データ点 | 例, 事例 |
| 回帰 | 教師あり学習 |
| 分類 | 教師あり学習 |
| 共変量 | 特徴 |
| 反応 | ラベル |

Table: 統計学と機械学習.

- 出典 <https://normaldeviate.wordpress.com/2012/06/12/statistics-versus-machine-learning-5-2/>

獲得すべき重要な概念

- 線形回帰, 重回帰, 多項回帰
- 過学習, 交差妥当性, 正則化
- 誤差関数, 損失関数, エネルギー関数
- 教師あり/教師なし学習
- ロジスティック回帰
- k近傍法

- サポートベクターマシン
- ナイーブベイズ

学習 = 表象 + 評価 + 最適化

| Representation | Evaluation | Optimization |
|---------------------------|-----------------------|----------------------------|
| Instances | Accuracy/Error rate | Combinatorial optimization |
| K-nearest neighbor | Precision and recall | Greedy search |
| Support vector machines | Squared error | Beam search |
| Hyperplanes | Likelihood | Branch-and-bound |
| Naive Bayes | Posterior probability | Continuous optimization |
| Logistic regression | Information gain | Unconstrained |
| Decision trees | K-L divergence | Gradient descent |
| Sets of rules | Cost/Utility | Conjugate gradient |
| Propositional rules | Margin | Quasi-Newton methods |
| Logic programs | | Constrained |
| Neural networks | | Linear programming |
| Graphical models | | Quadratic programming |
| Bayesian networks | | |
| Conditional random fields | | |

Domingos(2012) Tab.1 より

特徴学習 Feature Engineering

- 特徴エンジニアリング: 機械学習モデルがクラスを容易に区別できるよう、データから有用なパターンを抽出する技術
- たとえば陸生動物か水棲動物が画像内に居るかどうかについて、青緑の画素数と青の画素数を数えるなど
- 優れた分類のために考慮する必要があるクラスの数を制限するため、機械学習モデルに役立つ
- 特徴エンジニアリングは、ほとんどの予測課題で良好な結果を得るために最も重要な手法。
- しかし、異なるデータセットと異なる種類のデータは異なる特徴エンジニアリング手法を必要とするため、学習して習得することは困難。
- 特徴エンジニアリングを科学よりも芸術にする原則が存在
- 1つのデータセットで使用できる機能は、他のデータセットでは使用できないことがよくある（たとえば、次の画像データセットには陸上動物のみが含まれています）。
- 特徴エンジニアリングの難しさと関連する作業は、特徴を学習できるアルゴリズムを追求する主な理由。つまり、機能を自動的にエンジニアリングするアルゴリズムです。

特徴学習（物体認識、音声認識など）によって多くの課題を自動化することができる。特徴エンジニアリングは難しい課題（Kaggleマシン学習競技の大部分の課題のような）でうまくいくための最も効果的な手法の1つである。

特徴学習 Feature Learning

特徴学習アルゴリズムは、クラスを区別し、それらを自動的に抽出して分類または回帰プロセスで使用するのに重要な共通パターンを見つける。特徴学習は、特徴エンジニアリングがアルゴリズムによって自動的に実行されると考えることができます。深層学習では、畳み込み層は、複雑さが増す非線形の特徴（例えば、プロブ、エッジ->鼻、目、頬->顔）の階層を形成するために、次の層への画像において良好な特徴を見つける上で非常に優れている。最終層は、これらの生成されたすべての特徴を分類または回帰に使用する（畳み込みネットの最後の層は、本質的に多項ロジスティック回帰）。

深層学習 Deep Learning

階層的特徴学習では非線形特徴の複数層を抽出し、全特徴を組み合わせて予測を行う分類子にそれらを渡す。いくつかの層から複雑な特徴を学習することはできないため、非線形特徴の非常に深い階層を積み重ねることに興味があります。単一の非線形変換から抽出できる最も多くの情報が含まれているため、画像の場合、単一の層のための最善の特徴はエッジとプロブであることを

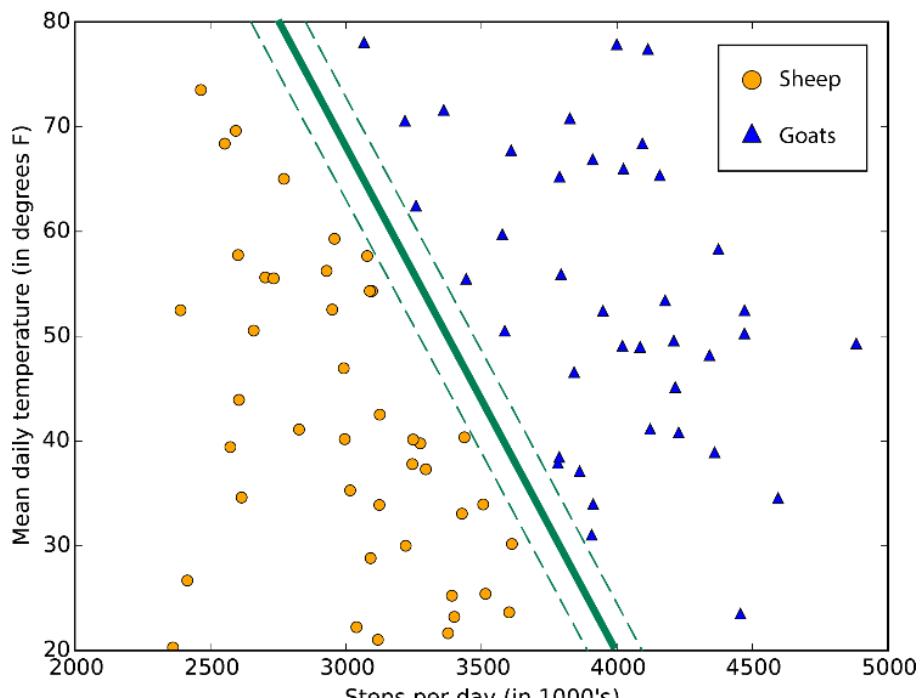
数学的に示すことができます。より多くの情報を含む特徴を生成するには、入力に対して直接操作することはできませんが、最初の特徴（辺とプロブ）を再度変換して、クラスを区別する詳細情報を含むより複雑な特徴を作成する必要があります。

視覚野の情報を受け取るニューロンの第1の階層は、特定のエッジおよびプロブに敏感であり、視覚的なパイプラインをさらに下回る脳領域は、顔などのより複雑な構造に敏感であることが示されている。

深層学習の領域が存在する以前に、階層的特徴学習が使用されていたが、これらのアーキテクチャでは、勾配が非常に小さくなりすぎて非常に深い層に学習信号を提供できない。これらのアーキテクチャが浅い学習アルゴリズム（サポートベクトルマシンなど）。

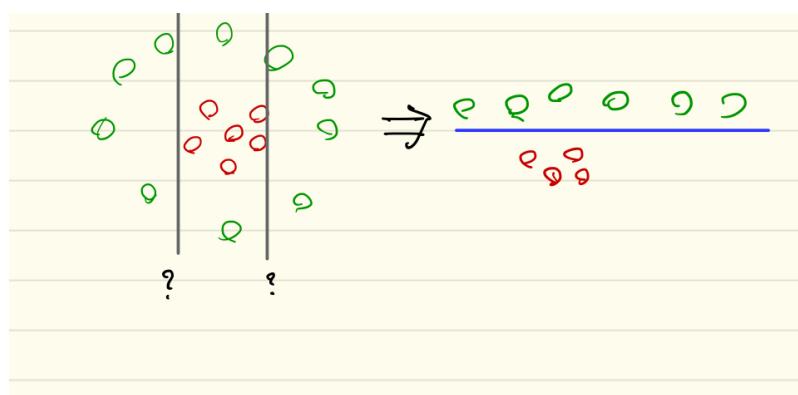
サポートベクターマシン SVM

サポートベクターマシン(以下SVM)の基本的なアイデアは、分類すべきクラスの間に最大の道を引くことがあります。これは **マージンの最大化** などと呼ばれます。



<https://docs.microsoft.com/en-us/azure/machine-learning/studio/algorithm-choice>

カーネルトリック kernel trick



正則化

決定境界上で**マージン**が小さい場合: 正しく分類できるようにマージンを小さくする

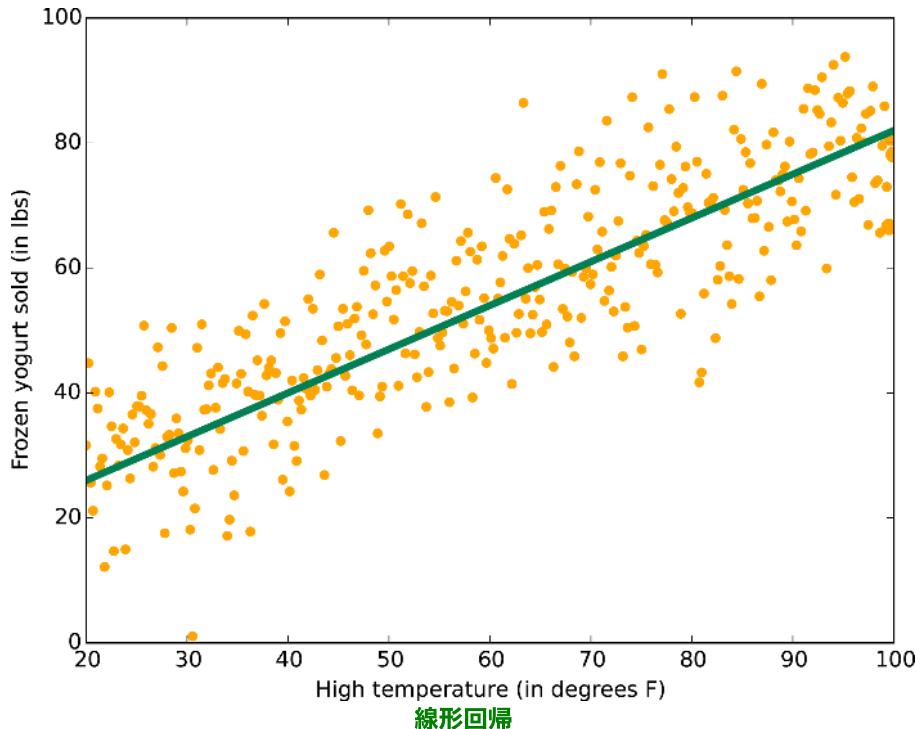
多クラスへの拡張

2群識別問題に分解して組み合わせ

1. 一対残り one-vs-all
2. 多対多

線形回帰 Linear regression

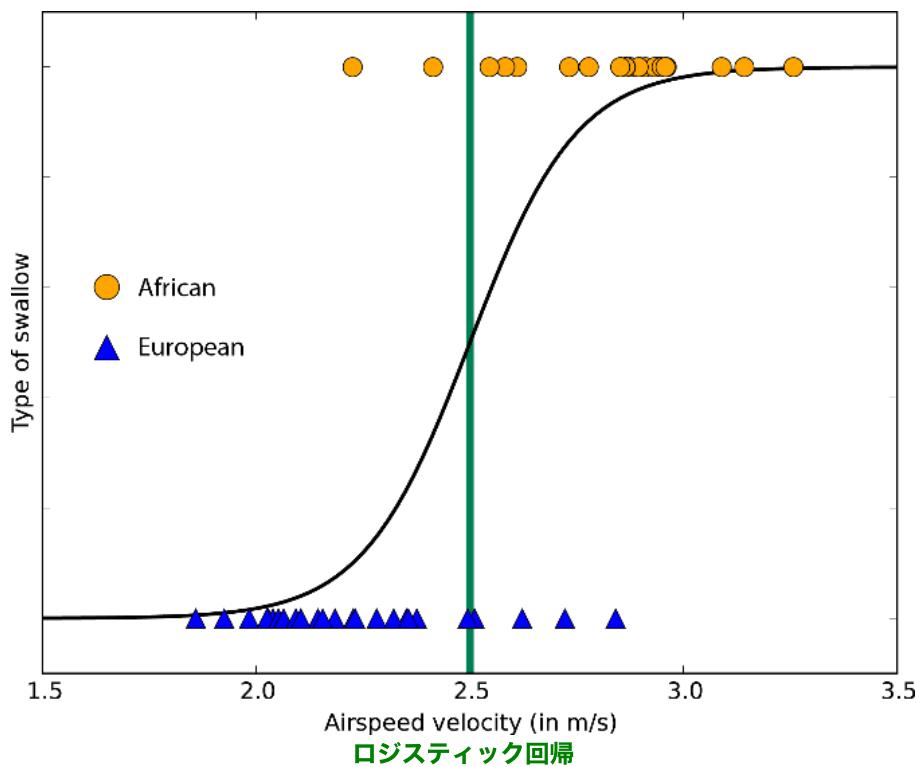
線形回帰は基本的な手法であり、観測されたデータの直線的関係を予測する手法です。



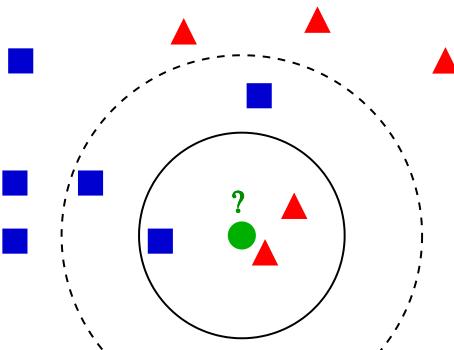
線形回帰

ロジスティック回帰 Logistic regression

ロジスティック回帰は回帰という言葉が使われているので紛らわしいですが、回帰ではなく分類手法に属します。データを2群に判別することを考えます。このとき、一方の群を0とし、他方を1とします。0と1との関係はどちらでも構いません。それぞれの群に属する確率を考えれば、0属する確率 P_0 は1に属する確率 P_1 から1を引いた値として求められるからです。 $P_0 + P_1 = 1$ が必ず成り立つので、この確率を予測することは、確率に対して回帰させるとみなしても良いので回帰という言葉が使われます。



ロジスティック回帰



https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm

計算方法

1. $k (\geq 1)$ を定める
2. テストデータに最も近い k 個のデータを既存のラベル付け済みデータから検索する
3. 上に基づいて分類する

k-NN の特徴:

1. 全訓練データを保存
2. 学習をしない
3. 各事例間の距離に従って判別

用語の整理

1. k-近傍法 k-nearest neighbor methods: k-NN
2. k-平均法 k-means (clustering) methods: k-means
3. NN は Neural Networks の場合と Nearest Neighbor の場合がある

用語の整理

- k-NN は strong{lazy} な学習であると言われます。この場合 lazy の反対語は \strong{eager} です。
- lazy を「のろまな」「怠惰な」と意味ではありません
- lazy: 意思決定時に全データを使用
- eager: 意思決定時に用いる判別関数は定まっている。**学習(訓練)時に用いたデータを用いない**

コンピュータサイエンスでは

- lazy evaluation: 遅延評価 (https://en.wikipedia.org/wiki/Lazy_evaluation)
- eager evaluation: 先行評価 (https://en.wikipedia.org/wiki/Eager_evaluation)

Pros and Cons

<https://medium.com/@adi.bronshtein/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7>

Pros:

- 外れ値に対して頑健
- データに関して特定な仮定が不要
- アルゴリズムが簡単: 説明可能性, 解釈可能性
- 高精度

Cons:

- 計算コスト高: lazy learning なため
 - メモリ容量高: 訓練データをすべて保存
 - 予測が遅い (大きなNの場合)
 - 無関連な特徴に対して敏感
-

ランダムフォレスト random forest

機械学習のメニューを眺めると、木、森、ジャングルなどの言葉が出てきます。これらは

ブースティング Boosting

<https://codesachin.wordpress.com/tag/adaboost/>

弱 weak learners 学習器を組み合わせて、**強**学習器 strong learner を構成するメタアルゴリズム

1. 弱学習器 weak learner

- 単純な学習器、当て推量 random guess より少し精度が良い程度。

2. アンサンブル学習 ensemble learning

- 全弱学習器の荷重和が SOTA になることもある

3. 逐次構築 Iteratively build

- **ランダムフォレスト**など**バギング**によるアンサンブル学習手法では、各学習器は**並列学習**
- 個々の弱学習器は、他の弱学習器とは無関係に学習可能
- **ブースティング**では学習が逐次的。各学習段階ステップでは、前段階までの学習結果の残差を評価し、学習
- 弱学習器が逐次的に総モデルに追加される

バギング\strong{Bagging} とはブートストラップによる学習器の束ね方\strong{B}ootstrap\strong{agg}regat\strong{ing} の略。

アンサンブル学習 ensemble learning

- 弱学習器 weak learner (原論文では **WeakLearn**) を組み合わせて**強**学習器を構成
- CNN のアンサンブルではここで紹介する手法とは別の方法も使われる (@2015Hinton_distill) がここでは触れない
- **メタ学習** meta learning アルゴリズム

バギングとブースティング

- バギング bagging : ランダムフォレスト
- ブースティング boosting : gradBoost, adaBoost

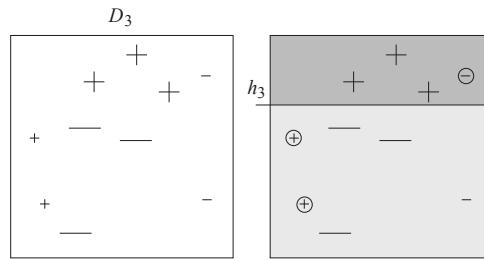
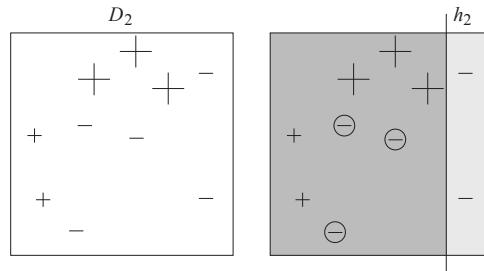
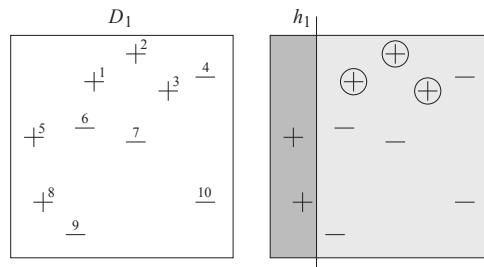


Domingos(2012) Fig.1を改変

ブースティングの流れ

1. 最初の弱学習器を訓練し F_0 とする
2. 時刻 τ 以前までの全学習器 $F_{i,t-1}$ の学習器の残差 r_i を評価
3. 残差 r_i に基いて弱学習器 r_i を訓練

例



$$H = \text{sign} \left(0.42 \begin{array}{|c|c|} \hline \text{+} & \\ \hline \end{array} + 0.65 \begin{array}{|c|c|} \hline & \text{-} \\ \hline \end{array} + 0.92 \begin{array}{|c|c|} \hline \text{+} & \text{-} \\ \hline \end{array} \right)$$

$$= \begin{array}{|c|c|c|} \hline \text{+} & \text{+} & \text{-} \\ \hline \text{+} & \text{-} & \text{-} \\ \hline \text{+} & \text{-} & \text{-} \\ \hline \end{array}$$

Pros and Cons of AdaBoost

Pros

- 実装が容易
- 広い適用可能性
- 特徴選択器として作用
- 高い般化性能
- オーバーフィット

Cons

- 局所解に陥る場合もある

See also <http://www.boosting.org>

エコシステム

インターネットの普及により知識共有が加速しました。オープンソースソフトウェアやソーシャルメディアのみならず、以下のエントリを含むエコシステムが昨今の進歩の源泉であろうと考えます。

- AMT
- ArXiv
- GitHub
- Google
- Kaggle
- Linux
- Medium
- Python
- Reddit, stackoverflow, stackexchange, 各種 SNS

特に最新の論文が [arXiv](#)に投稿され、そのソースコードが [GitHub](#)で公開されることはこの分野の進歩の大きな要因の一つと言えます。おなじく、[Kaggle](#)のようなデータ共有により知識は一層拡大し加速しています。

Python 上の標準的なライブラリの利用

1980 年代後半に教育用およびスクリプト用の言語として考案された Python は、それ以来、学界や産業界の多くのプログラマー、エンジニア、研究者、そしてデータ科学者にとって不可欠なツールとなっています。

競合するツールとしては

- [Microsoft Excel](#)
- [R](#)
- [MATLAB](#)
- [SPSS](#)

などがあるのはご存知でしょう。上記のライバルたちに比して Python は大規模で成熟したエコシステムに支えられていることが特徴と考えられます。逆に言えば Python を使っていないと有効な情報が得られないことが多いようです。この辺の事情は、ビジネスにおける Microsoft Excel, Web 開発における javascript と同じ状況だと考えられます。

- [NumPy](#) 多次元配列の操作、計算など
- [SciPy](#) 数学、科学、技術、の数値計算など
- [Pandas](#) データ処理、操作など
- [Matplotlib](#) 図など
- [Scikit-Learn](#) 機械学習のアルゴリズムを適用
- [seaborn](#) matplotlib に基づくデータ視覚化インターフェイス
- [Jupyter notebook](#) Python を Web ブラウザ上でインタラクティブに実行
- [Colaboratory](#) jupyter notebook のクラウド版だと考えてください。別項で説明します

- ブラウザを立ち上げて <https://colab.research.google.com/notebooks/welcome.ipynb?hl=ja> にアクセスしてください。
- [Google Colaboratory のチュートリアルビデオ](#) も参照してください
- Google Colaboratory は Jupyter Notebook をご存知であれば、ほぼ使い方は同じです。
 - notebook はセルと呼ばれる単位から成り立っています。
 - セルはコード、テキスト、画像で構成されます。
 - クラウドベースの実行系ですので、インストールの手間は不要です。
 - Google Doc に保存可能で Google Drive 経由でシェアできます
 - テキストはマークダウン形式で書きます
 - 少し時間がかかりますが Keras によるニューラル画像変換を実行してみることをお勧めします
 - <https://research.google.com/seedbank/seeds>
 - [How to Train Your Models in the Cloud](#)
 - https://colab.research.google.com/notebooks/basic_features_overview.ipynb
 - <http://colab.research.google.com/>

オリジナル URL: <https://research.google.com/colaboratory/faq.html>

- Colaboratory とは何ですか (What is Colaboratory?)

Google Colaboratory は機械学習教育研究用ツールです。環境構築に必要な事前のセットアップが不要な Jupyter Notebook 環境です。

- サポートしているブラウザは何ですか (What browsers are supported?)

Google Colaboratory は大抵のブラウザで動作します。動作テスト済ブラウザは、[Chrome](#) と [Firefox](#) です。

- 無料で使えますか (Is it free to use?)

無料です。

- ジュピター(Jupyter)と Colaboratory の違いは何ですか (What is the difference between Jupyter and Colaboratory?)

Colaboratory はオープンソースの [ジュピター\(Jupyter\)](#) を元にしています。ジュピターノートブック (Jupyter notebook) を他のユーザと共有することができます。ローカルな環境に対する、ダウンロード、インストール、実行、などは必要ありません。

- colaboratory.jupyter.org との関係ありますか (How is this related to colaboratory.jupyter.org?)

2014 年に Jupyter 開発チームと我々は本ツールの初期バージョンを共同で開発していました。以来 Colaboratory は発展し続け、グーグル内部用途となっています。

- Notebook はどこに保存されますか、また、シェアできますか (Where are my notebooks stored, and can I share them?)

Colaboratory ノートブックは [Google Drive](#) に保存されます。Google Docs や Sheets と同様にシェア可能です。右上のシェアボタンを押してください。Google Drive の [ファイル共有の手引き](#) に従ってください。

- Notebook をシェアする場合、何がシェアされますか (If I share my notebook, what will be shared?)

Notebook のシェアを選択すると、テキスト、コード、出力という notebook の全内容が共有されます。コードセルの出力を抑制させることは可能です。出力をシェアするには、保存時に [編集 > ノートブック設定 > コードセル出力の抑制](#) を選択してください。仮想環境の使用時は、使用時の設定ファイルやライブラリはシェアされません。ですので、ライブラリのインストールやカスタマイズをセル内に含めておくと良いでしょう。[libraries](#) や [files](#) を参照してください。

- 二人のユーザが同時に同じ notebook を編集したらどうなりますか (What happens if two users edit the same notebook at the same time?)

変更の反映は即時に全員へなれます。グーグル Docs の編集結果が編集中全ユーザに視認可能なのと同様です。

- 以前作成したジュピター(IPython)ファイルをインポートすることはできますか (Can I import an existing Jupyter/IPython notebook into Colaboratory?)

できます。ファイルメニューからノートブックのアップロードを選んでください

- Python3 (R, Scale)って何ですか (What about Python3? (or R, Scala, ...))

Colaboratory は、Python バージョン 2.7 と Python バージョン 3.6 をサポートしています。これら以外の R や Scala といったジュピターカーネルの使用を希望するユーザがいることは承知しています。将来的にはサポートするつもりですが、時間の制約が実現していません。

- Colaboratory ノートブックを検索するには? (How can I search Colaboratory notebooks?)

検索ボックスの **ドライブ**を選んでください。左上にある colaboratory のロゴをクリックすればグーグルドライブ上の全てのファイルを閲覧できます。**ファイル->最新のファイル**を開けば、最近閲覧したファイルを開くことができます。

- コードはどこで実行されるのですか。ブラウザのウィンドウを閉じてしまったら実行中のコードはどうなりますか (Where is my code executed? What happens to my execution state if I close the browser window?)

コードは仮想マシンで実行されます。仮想マシンはしばらく放置するとリフレッシュされます。仮想マシンがリフレッシュされるまでの最長寿命はシステムに寄ります^{^2}。

- 自分のデータを出力できますか (How can I get my data out?)

colaboratory 上で作成したノートブックをグーグルドライブからダウンロードできます。[instructions](#) や colaboratory のファイルメニューの中を御覧ください。colaboratory ノートブックは、オープンソースのジュピターノートブック形式(拡張子.ipynb)で保存されています。

- GPU は利用できますか。どうして GPU が使えない場合があるのですか (How may I use GPUs and why are they sometimes unavailable?)

colaboratory はインタラクティブな利用を想定しています。バックグラウンドで GPU を長時間実行すると停止させる場合があります。colaboratory を仮想通貨のマイニングに使わないでください。仮想通貨マイニングはサポート対象外です。長時間の継続利用に際は[ローカルランタイム](#)を推奨しています。

- 仮想マシンの実行をリセットできますか。どうしてリセットできない時があるのですか (How can I reset the virtual machine(s) my code runs on, and why is this sometimes unavailable?)

"ランタイム" メニュー内の "全てのランタイムをリセット" は割り当てられた管理下の全仮想マシンに対して実施されます。これは仮想マシンに不都合があった場合、たとえばシステムファイルを誤って上書きしてしまった場合など、に有効です。colaboratory は計算資源を消費するこのような事態に制約を課しています。このような事態が発生したら、しばらく待ってから再試行してください。

- バグを見つけました/質問があります。問い合わせ先を教えてください (I found a bug or have a question, who do I contact?)

colaboratory を開いて、「ヘルプ」メニューから「フィードバックを送る」を選んでください

[^1]: 日本語訳:浅川伸一 asakawa@ieee.org

- [Python](#)
 - [基本データ型](#)
 - [コンテナ](#)
 - [リスト](#)
 - [辞書](#)
 - [集合](#)
 - [タプル](#)
 - [関数](#)
 - [クラス](#)
- [Numpy](#)
 - [Arrays](#)
 - [Array indexing](#)
 - [Datatypes](#)
 - [Array math](#)
 - [Broadcasting](#)
- [SciPy](#)
 - [Image operations](#)
 - [MATLAB files](#)
 - [Distance between points](#)
- [Matplotlib](#)
 - [プロット](#)
 - [サブプロット](#)
 - [画像](#)

Python

Python は高級言語であり動的マルチパラダイムのプログラミング言語である。Python のコードは強力なアイデアを数行で表現できる擬似言語に喩えられる。例えば古典的なクイックソートアルゴリズムは Python では以下のようにになる:

```
def quicksort(arr):  
    if len(arr) <= 1:  
        return arr  
    pivot = arr[int(len(arr) / 2)]  
    left = [x for x in arr if x < pivot]  
    middle = [x for x in arr if x == pivot]  
    right = [x for x in arr if x > pivot]  
    return quicksort(left) + middle + quicksort(right)  
  
print(quicksort([3,6,8,10,1,2,1])) # 印字( "[1, 1, 2, 3, 6, 8, 10]" )
```

Python のバージョン

現在 Python は 2 種類のバージョンがサポートされている。2.7 系と 3 系である。Python 3.0 以来、前方互換性が廃棄されて 2.7 系と 3 系では互いに交換可能でない。本稿全コード 2.7 で動作する。

Python のバージョンを調べるにはコマンドラインから `python -version` とするとチェックできる。

基本データ型

他の多くの言語と同じく Python には [整数型](#)、[浮動小数点型](#)、[ブール\(真偽\)型](#)、[文字列型](#) という基本データ型がある。これらのデータ型は他のプログラミング言語と類似している。

数値

整数型と浮動小数点型は他言語と同様に動作する。

```
x = 3
print(type(x)) # 印字 "<type 'int'>"
print(x)       # 印字 "3"
print(x + 1)   # 加算; 印字 "4"
print(x - 1)   # 減算; 印字 "2"
print(x * 2)   # 乗算; 印字 "6"
print(x ** 2)  # 指数; 印字 "9"
x += 1
print(x)       # 印字 "4"
x *= (2)
print(x)       # 印字 "8"
y = 2.5
print(type(y)) # 印字 "<type 'float'>"
print(y, y + 1, y * 2, y ** 2) # 印字 "2.5 3.5 5.0 6.25"
```

他言語と異なりPythonには増分 `x++`, 減分 `x-` を行う単項演算子はない。

Pythonには倍精度整数, 複素数型も標準実装されている。詳細は

<https://docs.python.org/2/library/stdtypes.html#numeric-types-int-float-long-complex>に掲載されている。

ブール型(真偽型): Pythonは2値論理演算用の記号(`&&`[`and`], `||`[`or`], など)が用意されている。

```
t = True
f = False
print(type(t)) # 印字 "<type 'bool'>"
print(t and f) # 論理積 AND; 印字 "False"
print(t or f)  # 論理和 OR; 印字 "True"
print(not t)   # 論理否定 NOT; 印字 "False"
print(t != f)  # 排他的論理和 XOR; 印字 "True"
```

文字列

Pythonは文字列の扱いに優れる

```
hello = 'hello'          # 文字列リテラルはシングルクオート
world = "world"          # あるいはダブルクオート。両方可能
print(hello)              # 印字 "hello"
print(len(hello))        # 文字列長の印字; 印字 "5"
hw = hello + ' ' + world # 文字列連結
print(hw)                # 印字 "hello world"
hw12 = '%s %s %d' % (hello, world, 12) # sprintf 形式のフォーマット
print(hw12)               # 印字 "hello world 12"
```

文字列オブジェクトは有益なメソッド群である。

```
s = "hello"
print(s.capitalize())      # 文字列の語頭の大文字化; 印字 "Hello"
print(s.upper())           # 文字列の大文字化; 印字 "HELLO"
print(s.rjust(7))          # 文字列の右寄せ, 空白で埋まる; 印字 "    hello"
print(s.center(7))         # 文字列のセンタリング, 空白で埋める; 印字 " hello "
print(s.replace('l', '(ell)')) # 部分文字列の置換; 印字 "he(ell)(ell)o"
print(' world '.strip())   # 空白の除去; 印字 "world"
```

全文字列メソッドは <https://docs.python.org/2/library/stdtypes.html#string-methods> に記載されている。

コンテナ

Pythonにはリスト, 辞書, 集合, タプルというコンテナが標準実装されている。

リスト

リストは配列のPython実装であるが, サイズ可変, かつ, 異なる要素を持つことができる。

```

xs = [3, 1, 2]      # リストの作成
print(xs, xs[2])   # 印字 "[3, 1, 2] 2"
print(xs[-1])       # 負の指定子はリスト末端からの計数; 印字 "2"
xs[2] = 'foo'        # リストには異なる要素を含むことができる
print(xs)            # 印字 "[3, 1, 'foo']"
xs.append('bar')     # リスト末に新要素を追加
print(xs)            # 印字
x = xs.pop()         # リストの最終要素取り除きその値を返す
print(x, xs)         # 印字 "bar [3, 1, 'foo']"

```

上と同様に血みどろの全リストの詳細が

<https://docs.python.org/2/tutorial/datastructures.html#more-on-lists> に記載されている。

スライス:

Python には一旦リストにアクセスすれば、その下位リストへ簡単にアクセスする記法がある。これをスライスと呼ぶ。

```

nums = list(range(5)) # range は組み込み関数で整数からなるリストを生成する
print(nums)           # 印字 "[0, 1, 2, 3, 4]"
print(nums[2:4])     # 2番目の要素から4番目未満のスライス; 印字 "[2, 3]"
print(nums[2:])       # 2番目から最後までのスライス; 印字 "[2, 3, 4]"
print(nums[:2])       # 先頭要素から指定子2までの要素をスライス; 印字 "[0, 1]"
print(nums[:])         # リスト全体のスライスを得る; 印字 "[0, 1, 2, 3, 4]"
print(nums[:-1])     # スライス指定子には負値もとることができる; 印字 "[0, 1, 2, 3]"
nums[2:4] = [8, 9]    # スライスして新しい値を割り当てる
print(nums)           # 印字 "[0, 1, 8, 8, 4]"

```

NumPy の配列の項においてスライシングについて再び触れることとする。

ループ:

以下のようにリストの要素を繰り返すことができる。

```

animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
    # 各行に印字 "cat", "dog", "monkey"

```

あるループ内で各要素のインデックスを参照したければ、繰り返し内部で組み込み関数 `[enumerate]` を使えば良い。

```

animals = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
    # 印字行毎に "#1: cat", "#2: dog", "#3: monkey"

```

リスト内包表記:

プログラミングにおいてはある型のデータを別の型へ変換したい場合が多い。自乗を計算するコードを以下に示す:

```

nums = [0, 1, 2, 3, 4]
squares = []
for x in nums:
    squares.append(x ** 2)
print(squares)  # 印字 [0, 1, 4, 9, 16]

```

リスト内包表記を使えばより簡単なコードが書ける:

```

nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)  # 印字 [0, 1, 4, 9, 16]

```

辞書:

Java の Map や Javascript のオブジェクトと同様、辞書とは、キーとペア値を集めたものである。

```
d = {'cat': 'cute', 'dog': 'furry'} # データから辞書を作成
print(d['cat'])                  # 辞書の項目から内容を表示; 印字 "cute"
print('cat' in d)                # 辞書内を検索しキーの存在を返す; 印字 "True"
d['fish'] = 'wet'                 # 新項目を追加
print(d['fish'])                  # 印字 "wet"
# print(d['monkey'])           # 辞書 d からキー'monkey'を検索。存在しないとキーエラー
print(d.get('monkey', 'N/A'))    # 辞書の検索デフォルト値付き; 印字 "N/A"
print(d.get('fish', 'N/A'))      # 辞書の検索デフォルト値付き; 印字 "wet"
del(d['fish'])                   # 辞書内の項目を削除
print(d.get('fish', 'N/A'))      # "fish" は辞書にないので "N/A"
```

辞書についてもウェブ上に文書が掲載されている。

ループ:

辞書のキーを繰り返す簡便法がループである:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal in d:
    legs = d[animal]
    print('A %s has %d legs' % (animal, legs))
# 印字 "A person has 2 legs", "A spider has 8 legs", "A cat has 4 legs"
```

辞書の各項目に対応する値を取り出すには `iteritems` メソッドを用いる:

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A %s has %d legs' % (animal, legs))
# 印字 "A person has 2 legs", "A spider has 8 legs", "A cat has 4 legs"
```

辞書内包表記:

リスト内包表記と同様、辞書内包表記にも簡便な方法が存在する。以下の例のとおり:

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x ** 2 for x in nums if x % 2 == 0}
print(even_num_to_square) # 印字 "{0: 0, 2: 4, 4: 16}"
```

集合:

集合とは異なる要素を持つ順番を持たないデータの集まりである。以下の例を参照:

```
animals = {'cat', 'dog'}
print('cat' in animals) # 集合内の要素の存在の有無のチェック; 印字 "True"
print('fish' in animals) # 印字 "False"
animals.add('fish') # 集合に要素を追加
print('fish' in animals) # 印字 "True"
print(len(animals)) # 集合の要素数を返す; 印字 "3"
animals.add('cat') # 集合に既存の要素を加えても何も起きない
print(len(animals)) # 印字 "3"
animals.remove('cat') # 集合から要素を削除
print(len(animals)) # 印字 "2"
```

今までと同じく、集合の全てについて記述した文書が上梓されている。

繰り返し:

集合における繰り返しもリストに対する繰り返しと同じ記法である。しかし集合には順序がないので、集合内の各要素について順番を仮定してはならない:

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#%d: %s' % (idx + 1, animal))
    # 印字 "#1: fish", "#2: dog", "#3: cat"
```

集合内包表記:

リストや辞書と同様、集合内包にも簡易記法が存在する:

```
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums) # 印字 "set([0, 1, 2, 3, 4, 5])"
```

タプル:

タプルとは値の順序の変更が不能な順序付きリストである。タプルはリストに類似しているが、辞書のキーとして集合を要素として持つことが可能である。リストにはこの特徴は無い。簡単な例を示す:

```
d = {(x, x + 1): x for x in range(10)} # タプルキーを持つ辞書の生成
t = (5, 6) # タプルの生成
print(type(t)) # 印字 "<type 'tuple'>"
print(d[t]) # 印字 "5"
print(d[(1, 2)]) # 印字 "1"
```

<https://docs.python.org/2/tutorial/datastructures.html#tuples-and-sequences> には更に詳しい情報が記載されている。

関数

Python の関数はキーワード `def` を用いて定義される。以下の例:

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
# 印字 "negative", "zero", "positive"
```

関数は以下のようにオプションキーワード付きで定義可能である:

```
def hello(name, loud=False):
    if loud:
        print('HELLO, %s' % name.upper())
    else:
        print('Hello, %s!' % name)

hello('Bob') # 印字 "Hello, Bob"
hello('Fred', loud=True) # 印字 "HELLO, FRED!"
```

Python のクラス情報は <https://docs.python.org/2/tutorial/controlflow.html#define-functions> を参照のこと

クラス

Pythonにおけるクラス定義の記法は直截的である。

```

class Greeter:

    # Constructor
    def __init__(self, name):
        self.name = name # インスタンスの生成

    # Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, %s!' % self.name.upper())
        else:
            print('Hello, %s' % self.name)

g = Greeter('Fred') # Greeter クラスのインスタンスを生成
g.greet()           # インスタンスマетодの呼び出し; 印字 "Hello, Fred"
g.greet(loud=True) # インスタンスマethodの呼び出し; 印字 "HELLO, FRED!"
```

クラスについての詳細は以下である <https://docs.python.org/2/tutorial/classes.html>

NumPy

NumPyはPythonにおける科学技術計算の核となるライブラリである。高性能多次元配列オブジェクトと配列処理用ツールから構成されている。MATLABが既知ならば、本チュートリアルからNumPyを始めるのが良いだろう。

配列

NumPyの配列は値の格子状配置である。全要素の値は同じデータ型であり、非負整数のタプルで指定可能である。

Pythonにおける配列は入れ子になったリストであり、カギカッコを用いる。

```

import numpy as np

a = np.array([1, 2, 3])          # 次元数 1 の配列の作成
print(type(a))                  # 印字 "<type 'numpy.ndarray'>"
print(a.shape)                  # 印字 "(3,)"
print(a[0], a[1], a[2])         # 印字 "1 2 3"
a[0] = 5                         # 配列の要素を一つ変更
print(a)                         # 印字 "[5, 2, 3]"

b = np.array([[1,2,3],[4,5,6]])  # 次元数が 2 の配列の作成
print(b.shape)                  # 印字 "(2, 3)"
print(b[0, 0], b[0, 1], b[1, 0]) # 印字 "1 2 4"
```

NumPyには配列を生成する関数が用意されている:

```

import numpy as np

a = np.zeros((2,2))  # 全要素がゼロの配列の作成
print(a)             # 印字 "[[ 0.  0.]
                      #       [ 0.  0.]]"

b = np.ones((1,2))  # 全要素がゼロの配列の作成
print(b)             # 印字 "[[ 1.  1.]]"

c = np.full((2,2), 7) # 定数行列を作成
print(c)             # 印字 "[[ 7.  7.]
                      #       [ 7.  7.]]"

d = np.eye(2)        # 2x2 の単位行列の作成
print(d)             # 印字 "[[ 1.  0.]
                      #       [ 0.  1.]]"

e = np.random.random((2,2)) # 乱数で初期化した配列の作成
print(e)             # 印字例 "[[ 0.91940167  0.08143941]
                      #       [ 0.68744134  0.87236687]]"
```

配列を作成する他の方法は<http://docs.scipy.org/doc/numpy/user/basics.creation.html#arrays-creation>で読むことができる。

配列指定

NumPy には指定子を配列化する方法が複数存在する。

スライス:

Pythonにおけるリストと同様、NumPyの配列にはスライスが適用可能である。配列の各次元毎にスライスを指定可能である。

```
import numpy as np

# 3行4列で階数が2の行列の作成
# [[ 1  2  3  4]
# [ 5  6  7  8]
# [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# 下位行列のスライスし、上2行の1列目2列目を用いて2行2列の行列を作成
# [[2 3]
# [6 7]]
b = a[:2, 1:3]

# スライスは元行列への参照なので、スライスへの変更は元データに影響を及ぼす
print(a[0,1])    # 印字 "2"
b[0, 0] = 77    # b[0, 0] は a[0, 1] と等価
print(a[0,1])    # 印字 "77"
```

整数指定子とスライス指定子を混在させることができる。しかし、そうすると元行列よりも次元の低い行列を作成することになる。MATLABにおける行列スライスとは異なることに注意せよ。

```
import numpy as np

# 3行4列の2次の行列を作成
# [[ 1  2  3  5]
# [ 4  6  7  8]
# [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# 行列の中央行へのアクセス2種類
# スライスと整数の混在は階数の低下をもたらす
# 一方、スライスのみを使用した場合には原行列の次元低減は発生しない
row_r1 = a[1, :]    # 行列 a の 2 行目の全列表示
row_r2 = a[1:2, :]  # 行列 a の 2 行目の全列表示
print(row_r1, row_r1.shape)  # 印字 "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # 印字 "[[5 6 7 8]] (1, 4)"

# 行列に対する列への操作も同様
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # 印字 "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # 印字 "[[ 2
                             #          [ 6]
                             #          [10]] (3, 1)"
```

整数型配列指定子:

NumPyの配列に対してスライスを実行すると、結果は元行列の下位配列となる。これに対して、整数型変数からなる配列に対してスライスを実行すれば、新たな配列が構成される。以下のとおり：

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

# 整数型配列の例 戻り値の配列の shape は (3,)
print(a[[0, 1, 2], [0, 1, 0]])  # 印字 "[1 4 5]"

# 上例は整数型配列を指定子とした場合と等しい：
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))  # 印字 "[1 4 5]"

# 整数型配列を指定子として用いると元配列と同じ要素を再利用できる
print(a[[0, 0], [1, 1]])  # 印字 "[2 2]"

# 先の整数型の配列指定子に等しい
print(np.array([a[0, 1], a[0, 1]])) # 印字 "[2 2]"
print(np.array([a[0, 1], a[0, 1]])) # 印字 "[2 2]"
```

ブーリアン配列指定子:

ール(2値)型配列指定子は配列の任意の要素を取り出すことが可能である。この指定子により配列から任意の条件を満たす要素からなる配列を作成することが可能である。

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])
bool_idx = (a > 2) # 2 より大きい配列の要素を探す
# これにより numpy の配列と等しい shape でブール型の
# 配列を返す。bool_idx は a の各要素が 2 より大きい
# か否かを示す
print(bool_idx)    # 印字 "[[False False]
#                   [ True  True]
#                   [ True  True]]"

# ブール型配列指定子を用いて階数1の行列を作ることができる
# bool_idx の True に対応する要素で構成される行列である
print(a[bool_idx]) # 印字 "[3 4 5 6]"

# 上例を簡潔に表記可能である
print(a[a > 2])   # 印字 "[3 4 5 6]"
```

本稿の簡潔性ゆえNumPyの配列指定子の詳細については多くを記載していない。詳細は文献を参照のこと。

データ型

NumPy の配列は同じ型の要素からなるグリッドである。配列作成には一連のデータ型を利用可能である。NumPyは配列のデータ型が何であるかを類推しようとする。以下のようにオプション引数により明示的に配列のデータ型を指定することも可能である:

```
import numpy as np

x = np.array([1,2])      # numpy にデータ型を選択させる
print(x.dtype)           # 印字 "int64"

x = np.array([1.0, 2.0])  # numpy にデータ型を選択させる
print(x.dtype)           # 印字 "float64"

x = np.array([1, 2], dtype=np.int64) # データ型の強制指定
print(x.dtype)           # 印字 "int64"
```

NumPy の全データ型については<http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html>を参照のこと。

数学的配列

基本的数学関数の操作は配列の要素毎の演算である。NumPy モジュールの関数として演算子を上書き可能である。

```

import numpy as np

x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

# 対応する要素の和、両例とも同じ結果を得る
# [[ 6.0  8.0]
# [10.0 12.0]]
print(x + y)
print(np.add(x, y))

# 対応する要素の差、両例とも同じ結果を得る
# [[-4.0 -4.0]
# [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))

# 対応する要素の積、両例とも同じ結果を得る
# [[ 5.0 12.0]
# [21.0 32.0]]
print(x * y)
print(np.multiply(x, y))

# 対応する要素の商、両例とも同じ結果を得る
# [[ 0.2          0.33333333]
# [ 0.42857143   0.5         ]]
print(x / y)
print(np.divide(x, y))

# 各要素の平方根
# [[ 1.           1.41421356]
# [ 1.73205081  2.          ]]
print(np.sqrt(x))

```

MATLAB と異なり、`[*]` は要素毎の積であり、行列の積ではない。ベクトルの内積、ベクトルと行列の積、行列の積には`[dot]`を用いる。`[dot]` は NumPy モジュールと配列オブジェクトのメソッドインスタンスに適用可能である。

```

import numpy as np

x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

# ベクトルの内積両者とも同じく 219 を得る
print(v.dot(w))
print(np.dot(v, w))

# 行列とベクトルの積。両者とも同じく [29 67] を得る
print(x.dot(v))
print(np.dot(x, v))

# 行列と行列の積。両者とも同じく以下の行列を得る
# [[19 22]
# [43 50]]
print(x.dot(y))
print(np.dot(x, y))

```

NumPy は行列操作遂行に有役な関数を数多く提供している。`[sum]` など

```

import numpy as np

x = np.array([[1,2],[3,4]])

print(np.sum(x))          # 総和を計算 10と印字
print(np.sum(x, axis=0))  # 各列の総和を計算: "[4 6]" と印字
print(np.sum(x, axis=1))  # 各行の総和を計算: "[3 7]" と印字

```

NumPy の全数学関数リストは <http://docs.scipy.org/doc/numpy/reference/routines.math.html> から入手可能である。

配列を用いた数学関数の計算だけでなく、配列の `reshape` や他のデータ操作が多数必要となる。単純な例は行列の転置である。行列の転置には配列オブジェクトの`[T]`を使えばよい。

```

import numpy as np

x = np.array([[1,2], [3,4]])
print(x)      # 印字 "[[1 2]
               #           [3 4]]"
print(x.T)   # 印字 "[[1 3]
               #           [2 4]]"

# 階数1の行列を転置してもなにも変わらないことに注意
v = np.array([1,2,3])
print(v)      # 印字 "[1 2 3]"
print(v.T)   # 印字 "[1 2 3]"

```

NumPy は配列操作関数が多い。<http://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html>を参照されたい。

プロードキャスト

プロードキャストは算術的 operation を行う際、NumPy が異なる shape の配列に強力な処理機構を行うことを可能にする。

行列の各行に対して定数ベクトルを加える場合を以下に示す:

```

import numpy as np

# 行列 x の各行に対してベクトル v を加え結果を y に格納
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)    # x と同じ shape を持つ空行列を生成

# 明示的繰り返しを用いて行列 x の各行にベクトル v の要素を加算
for i in range(4):
    y[i, :] = x[i, :] + v

# y は以下のようになる
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
# [11 11 13]]
print(y)

```

行列 `x` が巨大な場合 Python の明示的なループは遅い。行列 `x` の各行にベクトル `v` を加えることはスタッツ領域に積まれた `v` のコピーと行列 `vv` の和を構成することに等しい。以下の例のようになる。

```

import numpy as np

# 行列 x の各要素にベクトル v を加える
# 行列 y に結果を格納
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))  # v から4つのスタッツを作成
print(vv)                # 印字 "[[1 0 1]
                           #           [1 0 1]
                           #           [1 0 1]
                           #           [1 0 1]]"

y = x + vv # x と vv の各要素を加える
print(y)    # 印字 "[[ 2  2  4
               #           [ 5  5  7]
               #           [ 8  8 10]
               # [11 11 13]]"

```

NumPy のプロードキャストは `y` の多重コピーを作成することなく計算を実行する。プロードキャストを用いたバージョンを以下に示す:

```

import numpy as np

# 行列 x の各行にベクトル v を加える
# 結果を行列 y に格納する
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v # プロードキャストを使って x の各行に v を加える
print(y)    # 印字 "[[ 2  2  4]
               #           [ 5  5  7]
               #           [ 8  8 10]
               # [11 11 13]]"

```

`y = x + v` の行は、`x` は `(4, 3)` の次数を持つ、`v` は `(3,)` の次数を持つことがブロードキャストにより `y = x + v` が計算される。もし `v` が次数 `(4, 3)` であるように作用する。各行は `v` のコピーであり各要素の和が計算される。

2つの行列をブロードキャストする場合以下の規則に従う:

1. もしごとの行列が同じ次数でなければ低次数の行列を拡張して等しくする。
2. 二つの行列が、特定の次元のみ一致している場合、片方の配列の次元が1であっても、他の次元も同じサイズの次元にする。
3. 二つの配列全ての次元に互換性があれば両者ともブロードキャストされる。
4. ブロードキャスト後、各配列は両入力配列の次数が等しいとして振る舞う。
5. 一方の配列のサイズが1であり他方の配列サイズが1より大きい場合、最初の配列は次元を繰り返し複写して動作する

上記の意味が不明なら、<http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html> の <http://wiki.scipy.org/EricsBroadcastingDoc> 説明を読まれたい。

ブロードキャストをサポートする関数はユニバーサル関数として知られる。ユニバーサル関数のリストは <http://docs.scipy.org/doc/numpy/reference/ufuncs.html#available-ufuncs> を参照のこと。

ブロードキャストの例を示す:

```
import numpy as np

# ベクトルの外積を計算
v = np.array([1, 2, 3]) # v の次数 (3,)
w = np.array([4, 5]) # w の次数 (2,)
# 外積を計算するため v の次数を列へと変形する
# 次数 (3, 1) のベクトル; w に対してブロードキャストする
# 外積の計算結果である次数 (3, 2) の行列を得る
# [[ 4  5]
#  [ 8 10]
#  [12 15]]
print(np.reshape(v, (3, 1)) * w)

# 行列の各行にベクトルを加算する
x = np.array([[1, 2, 3], [4, 5, 6]])
# x は次数 (2, 3) であり v は (3,) である。v はブロードキャストされて (2, 3) となり
# x has shape (2, 3) and v has shape (3,) so they broadcast to (2, 3),
# 以下の行列を得る:
# [[2 4 6]
#  [5 7 9]]
print(x + v)

# 行列の各列にベクトルを加える
# x の次数は (2, 3) であり、w の次数は (2,) である。x を転置すれば
# 次数は (3, 2) となり w はブロードキャストされ次数は (3, 2) となる。
# 結果は転置され次数 (2, 3) の行列となり以下の行列を得る:
# [[ 5  6  7]
#  [ 9 10 11]]
print((x.T + w).T)
# 別解としては次数 (2, 1) の行ベクトルに対して次数を変換し、
# その結果をブロードキャストして直接外積を計算する
print(x + np.reshape(w, (2, 1)))

# 行列の定数倍:
# 行列 x の次数は (2, 3)。Numpy はスカラを次数 () の配列として扱う:
# スカラはブロードキャストされ次数 (2, 3) の行列となり以下の結果を得る
# [[ 2  4  6]
#  [ 8 10 12]]
print(x * 2)
```

ブロードキャストによりコードは簡潔になり、高速化する。可能な限りブロードキャストを使うよう努めるべきである。

NumPy の文書

ここでは NumPy に関して概説したが完全な記述には程遠い。NumPy のリファレンス <http://docs.scipy.org/doc/numpy/reference/> には詳細が記されている。

NumPy は高性能な多次元配列を提供し、配列操作の基本道具を提供している。SciPy は (<http://docs.scipy.org/doc/scipy/reference/>) NumPy で定義された配列と科学技術に有益な関数数多く用意している。

SciPy を理解するには<http://docs.scipy.org/doc/scipy/reference/index.html> を観ることである。ここではSciPy の有益点いくつかに焦点をあてる。

画像操作

SciPy は画像操作の基本関数を提供する。例えば、ディスクから画像を読み込んで NumPy の配列に代入し、NumPy の配列を画像としてディスクに書き込み、画像のサイズを変更する関数などである。ここでは、これらの関数例を供覧する

```
from scipy.misc import imread, imsave, imresize

# JPEG 画像を NumPy 配列に読み込む
img = imread('assets/cat.jpg')
print(img.dtype, img.shape) # 印字 "uint8 (400, 248, 3)"

# 各色チャンネルを異なるスカラ定数により色合いを変える。
# 画像は (400, 248, 3) の次数である。
# 次数 (3,) の配列 [1, 0.95, 0.9] を掛ける;
# numpy のプロードキャストは赤チャンネルは変更せず
# 緑と青とをそれぞれ 0.95 倍, 0.9 倍する
img_tinted = img * [1, 0.95, 0.9]

# 色合いを変更した画像を縦横 300 画素の画像に変換する
img_tinted = imresize(img_tinted, (300, 300))

# 変更した画像をディスクに書き出す

imsave('assets/cat_tinted.jpg', img_tinted)
```

Distance between points

SciPy は点の集合間の距離を計算するための有益な関数が定義されている。

[[scipy.spatial.distance.pdist](#)] 関数は所与の集合に属する全点間の距離を計算する:

```
import numpy as np
from scipy.spatial.distance import pdist, squareform

# 2次の行からなる以下の配列を定義する:
# [[0 1]
#  [1 0]
#  [2 0]]
x = np.array([[0, 1], [1, 0], [2, 0]])
print(x)

# x の全ての行のユークリッド距離の計算
# d[i, j] は x[i, :] と x[j, :] との距離であり以下の行列となる
# [[ 0.          1.41421356  2.23606798]
#  [ 1.41421356  0.          1.          ]
#  [ 2.23606798  1.          0.          ]]
d = squareform(pdist(x, 'euclidean'))
print(d)
```

詳細は<http://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html>に記されている。

類似の関数[[scipy.spatial.distance.cdist](#)] は 2つの集合の 2 点間の全対の距離を計算する。
<http://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.cdist.html>を参照のこと。

Matplotlib

Matplotlib は描画ライブラリである。本節では Mycmdmatplotlib.pyplot モジュールを概説する。MATLAB と同等のものである。

プロット

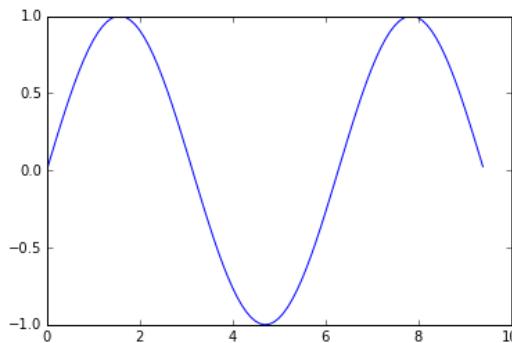
Matplotlib の最重要関数は[`plot`] である。2次元の描画が可能である。以下に例を示す:

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

# x, y 座標に正弦波を描く
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

# matplotlib の plot 関数で描画
plt.plot(x, y)
plt.show() # 画面に表示するために plt.show() を使う
```

上記のコードを実行することで以下の画像を得る。

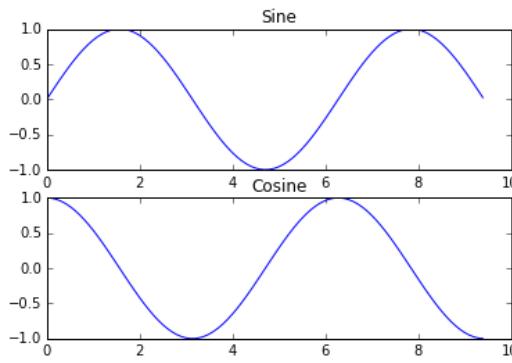


さらに手を加えると複数の線、タイトル、凡例、軸ラベルを描画可能である:

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

# x, y 座標上に正弦波と余弦波の曲線をプロットする
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

# matplotlib の plot を用いる
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
plt.show()
```



[`plot`]関数の詳細はhttp://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plotから読むことができる。

サブプロット

[`subplot`]関数を用れば同一図に別のプロットを入れることが可能である。以下に例を示す:

```

import numpy as np
import matplotlib.pyplot as plt

# x, y 座標上で正弦波と余弦波をプロット
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

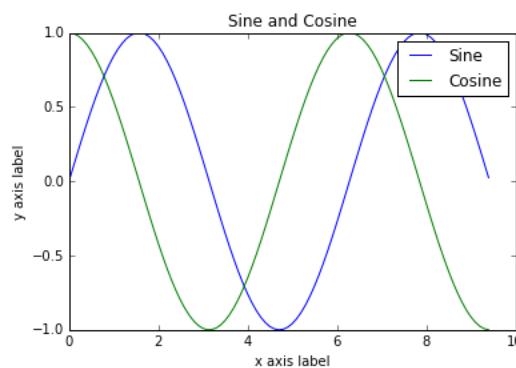
# 高さ2、幅1のサブプロット用格子を設定
# 最初のサブプロットをアクティブにする
plt.subplot(2, 1, 1)

# 最初のプロットを作成
plt.plot(x, y_sin)
plt.title('Sine')

# 2番目のサブプロットをアクティブにし、プロットを作成
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

# 図の画面表示
plt.show()

```



[`subplot`]関数の詳細はhttp://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.subplotから読むことができる。

画像

[`imshow`]関数を使って画像を表示することが可能である:

```

import numpy as np
from scipy.misc import imread, imresize
import matplotlib.pyplot as plt

img = imread('assets/cat.jpg')
img_tinted = img * [1, 0.95, 0.9]

# 原画像の表示
plt.subplot(1, 2, 1)
plt.imshow(img)

# 色合いを変えた画像の表示
plt.subplot(1, 2, 2)
plt.imshow(img_tinted)
plt.show()

```



2.7 系と 3.4 系の相違についてはセバスチャン・ラシュカ (Sebastian Raschka) のブログ記事が参考になった(<http://sebastianraschka.com/Articles/2014_pandas_tutorial.html#plotting-with-matplotlib>)

[^2]: 訳注: 現行では[`python3`]コマンドで3系のPythonが動作するパッケージが散見される
(Anaconda<, Ubuntu, Homebrew, MacPorts, など)

[^3]: 訳注: ドキュメントはURL直下の数字が2か3かの違いにより、バージョンごとに文書が整備されているので3系のURLを逐次紹介することは避ける

だからと言って日本語文字ファイル名問題が解決した訳ではない。もちろんそれはPythonの責任ではない

[^6]: Python, NumPy用語で `rank` とは配列の次元数を表す(1ならベクトル, 2なら行列), `shape` とは配列の各次元の要素数を指すタプル