Chapter 13.　Parallelizing Neural Network　Training with Theano (Theanoによるニューラルネットワークの訓練の並列化)

- Theanoを用いて最適化した機械学習コードを作成
- 人工ニューラルネットワークのための活性化関数を選択
- 簡単にかつ素早い実験のために、Keras 深層学習ライブラリを使用

13.1 Theanoを用いた式の構築、コンパイル、実行

- Theanoの設計方針：Pythonを用いて機械学習モデルを効果的にトレーニングする
- Bengio Y.　2008　LISA(Laboratoire d'Informatique des Systemes Adaptatifs)研究室
- 負荷の高い計算の実行における課題

  Pythonの実行はデフォルトで1つのコアに制限：GIL(Global Interpreter Lock)

  複数のコアの使用には multiprocessing ライブラリを利用－とは言っても、高級なマシンでも8や16コア以上のものは少ない

  ピクセル密度の高い画像処理ではパラメタ数が爆発的に増加

  ⇒ GPUの利用が鍵　　GPUはマシンの中の小さなコンピュータ・クラスタ、しかも比較的安価

  ただしGPUを対象としたコードを書くのは簡単ではない：CUDAやOepnCLなどがあるものの…

  そこでTheano が役にたつ

13.1.1　Theanoとは何か

- 多次元配列（テンソル）に対し、数式の実装、コンパイル、評価を効率良く行うことを目的としたフレームワーク---GPUの活用に効果、CPUではNumPyの1.8倍、GPUでは11倍の速さ
- TheanoとNumPyとの関係：NumPyをベースとした構築、構文も似て、NumPyの使用者にとって使いやすい。また SymPyとも類似

  NumPyでは変数とその結合方法を定義すればコードが1行ずつ実行される

  　Theanoでは問題とその分析方法の説明を最初に書く→Theanoがコードの最適化とコンパイルを自動的に行う。最適化したコード生成には、問題のスコープをTheanoに教える
- 本章ではTheanoの基本概念と、機械学習への応用方法を学ぶ

  Theanoについて詳しくは http://deeplearning.net/software/theano/

13.1.2 はじめてのTheano

- PyPIからTheanoをインストール:　　pip install Theano
- 記号を用いた数式を評価するため、テンソルを中心に構築

  注：テンソルはスカラー(階数0のテンソル)、ベクトル(階数1のテンソル)、行列（(階数2のテンソル)などの一般化

例：重みw_1, バイアスw_0, サンプル点xの総入力zの計算：　$z = x_1 \times w_1 + w_0$

そのコード: 3つのステップ=　記号(Variableオブジェクト)を定義、コードをコンパイル、実行

```
>>> import theano
>>> from theano import tensor as T
```

```
# initialize  初期化
>>> x1 = T.scalar()
>>> w1 = T.scalar()
>>> w0 = T.scalar()
>>> z1 = w1 * x1 + w0
# compile  コンパイル
>>> net_input = theano.function(inputs=[w1, x1, w0],  outputs=z1)
# execute  実行
>>> print('Net input: %.2f' % net_input(2.0, 1.0, 0.5))
Net input: 2.50
```

- Theanoでは変数の型(dtype)の宣言が必要: 64bit整数、32bit整数、浮動小点数を選択

13.1.3 Theanoの設定

- Theanoは32ビットのメモリアドレスだけをサポート　--- OSではほとんど64ビット
  理由:数式の評価をGPUで高速処理するため
- 機械学習のアルゴリズムの実装は主として浮動小数点数を扱う: float64を使用
  ただしプロトタイプをCPUで作成し実際のコードをGPUで実行するには、float64とfloat32の切り替えが必要:
  CPUではどちらも使用可能、GPUでは float32を使用
- Theanoの浮動小数点数の変数のデフォルト設定の表示

  ```
  >>> print(theano.config.floatX)
  float64
  ```
- 浮動小数点数の変数のデフォルト設定をfloat32に切り替える

  ```
  >>> theano.config.floatX = 'float32'
  ```

  環境変数を用いてグローバルに設定を変更する(Linuxのシェルにおいて)

  ```
  export THEANO_FLAGS=floatX=float32
  ```

  特定のPython実行のみに適用する場合は(Linuxのシェルにおいて)

  ```
  THEANO_FLAGS=floatX=float32 python  自分のスクリプト.py
  ```
- CPUとGPUの切り替え方法について
  CPUとGPUのどちらを使用しているかの確認

  ```
  >>> print(theano.config.device)
  cpu
  ```

  デフォルトをCPUとするのが推奨: プロトタイプ作成とデバッグが簡単になる
  CPUで実行する場合　 THEANO_FLAGS=device=cpu,floatX=float64 python 自分のスクリプト.py
  GPUで実行する場合　THEANO_FLAGS=device=gpu,floatX=float32 python　自分のスクリプト.py
  環境設定ファイルの利用: (Linuxの場合) ~/.theanorc ファイルを作成し設定を書き込む:
  例:　echo -e "¥n[global]¥nfloatX=float32¥ndevice=gpu¥n" >> ~/.theanorc
  (実際に書き込まれる内容は、次のWindowsの場合と同じ)

MacOS XやLinux以外の場合: 要するにWindowsの場合は「c:¥Users¥アカウント」 の下に .theanorcファイルを作り、以下の内容を書き込む:

```
[global]
floatX=float32
 device=gpu
```

13. 1.4 配列構造の操作

- Theanoのtensorモジュールを用いて、配列構造を操作する
- 2×3行列を作成、列の和を求める

```
>>> import numpy as np
>>>import theano.tensor as T        # 教科書にはこれが欠けている
# 初期化:変数の定義
>>> x = T.fmatrix(name='x')
>>> x_sum = T.sum(x, axis=0)
# コンパイル
>>> calc_sum = theano.function(inputs=[x], outputs=x_sum)
# 実行(Python list)
>>> ary = [[1, 2, 3], [1, 2, 3]]
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2. 4. 6.]
# 実行(NumPy array)
>>> ary = np.array([[1, 2, 3], [1, 2, 3]], dtype=theano.config.floatX)
>>> print('Column sum:', calc_sum(ary))
Column sum: [ 2. 4. 6.]
```

三つの基本ステップの確認:変数の定義、コンパイル、実行

上の例は、Pythonのリスト型でも、numpyのndarray型でもTheanoが操作できることを示したもの

- 注意: 今の例ではTheanoのtensor.fmatrix関数に引数name='x'を与えてTheanoの変数を定義し、それをPythonの変数xにセットした。このとき、引数nameを与えなければ、

```
>>> print(x)
<TensorType(float32, matrix)>
```

と表示される。それに対し、今の例のようにnameを与えると、

```
the print function:
>>> print(x)
```

となることに注意しよう。なお、TensorTypeはtypeメソッドで表示できる:

```
>>> print(x.type())
<TensorType(float32, matrix)>
```

- Theanoはメモリ空間を CPUとGPUに割り振って管理する。「共有変数」とすることにより、複数の関数からアクセルできるだけではなく、コンパイル後でもオブジェクトの値を更新することが可能になる

```
#初期化（変数の定義）
x = T.fmatrix(name='x')
w = theano.shared(np.asarray([[0.0, 0.0, 0.0]], dtype=theano.config.floatX))
z = x.dot(w.T)
update = [[w, w + 1.0]]
# コンパイル
net_input = theano.function(inputs=[x],  updates=update,  outputs=z)
# 実行
data = np.array([[1, 2, 3]], dtype=theano.config.floatX)
for i in range(5):
    print('z%d:' % i, net_input(data))
z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

この例では w を共有変数とし (sharedメソッド)、numpyの配列(3次元配列、要素はいずれも0.0)を値にセットしている。$z=x.w^T$ とし、値の更新は w=w+1.0としている。dataにはnumpyの配列[1,2,3]を与え、変数xを通してwの（したがってzの）値更新を5回行い、それぞれのzの値を表示させている。

この方法だと、[1,2,3]の値は変更されないのに何回もCPUからGPUに値が転送される→無駄

givens変数を用いると、コンパイルの前に値を挿入でき、したがって値の転送回数が減らせる

```
# 初期化（変数の定義）
data = np.array([[1, 2, 3]], dtype=theano.config.floatX)
x = T.fmatrix(name='x')
w = theano.shared(np.asarray([[0.0, 0.0, 0.0]], dtype=theano.config.floatX))
z = x.dot(w.T)
update = [[w, w + 1.0]]
# コンパイル
net_input = theano.function(inputs=[],    updates=update,
                    givens={x: data}, outputs=z)
# 実行
for i in range(5):
    print('z%d:' % i, net_input())
z0: [[ 0.]]
z1: [[ 6.]]
z2: [[ 12.]]
z3: [[ 18.]]
z4: [[ 24.]]
```

givens属性はPythonの「辞書」で、変数とPythonオブジェクトを対応付けている

13. 1.5 以上のまとめ:最小二乗法線形回帰の実装

- 10個の訓練サンプルからなる1次元データの用意

```
import numpy as np
>>> X_train = np.asarray([[0.0], [1.0], [2.0], [3.0], [4.0], [5.0], [6.0], [7.0], [8.0], [9.0]],
                         dtype=theano.config.floatX)
>>> y_train = np.asarray([1.0, 1.3, 3.1, 2.0, 5.0, 6.3, 6.6, 7.4, 8.0, 9.0],
                         dtype=theano.config.floatX)
```

注意: dtypeを指定しているのは、GPUとCPUを切り替えられるようにするため

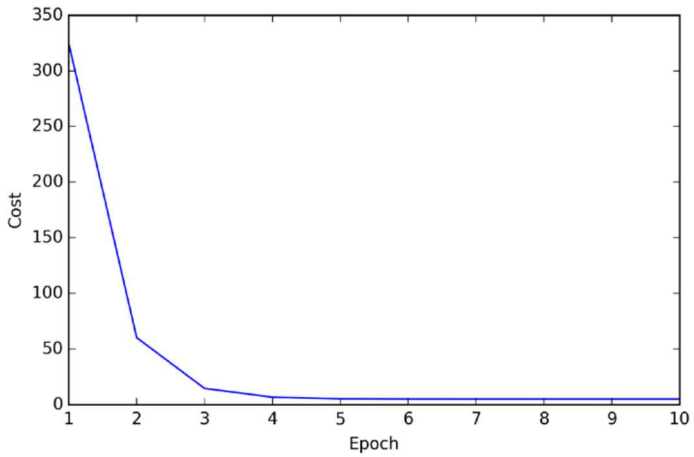
- 重みを学習する訓練関数を実装。誤差平方和(SSE)コスト関数を使用、w_0はバイアス

コードは省略. ただし以下に注目

```
# コストの計算
net_input = T.dot(X, w[1:]) + w[0]          # net_input = x.w^T
errors = y - net_input                      # errors = y – net_input
cost = T.sum(T.pow(errors, 2))              # cost = ∑errors² = ∑(y − w^T x)
# 微分係数と重みの更新
gradient = T.grad(cost, wrt=w)              # grad = ∂cost/∂w
update = [(w, w - eta0 * gradient)]         # w ← w − η grad = w − η ∂cost/∂w
```

grad関数はwrtパラメタ(この場合はw)の変数についての部分を計算:微分対象のcostはerrorの関数、errorsはnet_inputの関数、net_inputはwの関数

- 線形回帰モデルの訓練とSSEコスト関数値のチェック(収束したかどうかの検査)

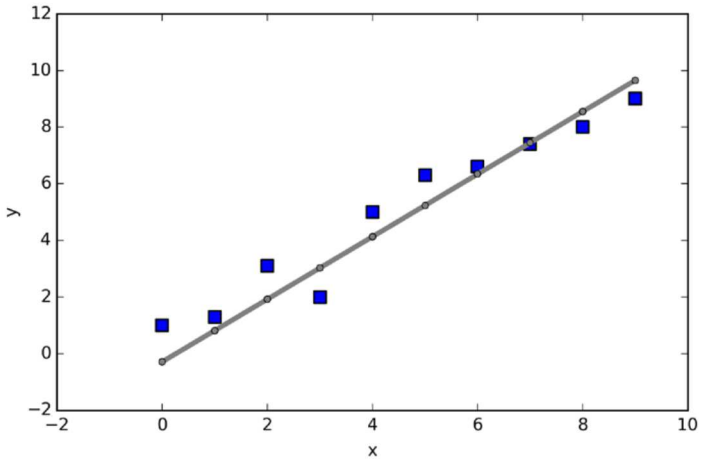


- 入力特徴量に基づく予測のための関数

```
def predict_linreg(X, w):
    Xt = T.matrix(name='X')                 # 変数の定義(下の行も)
    net_input = T.dot(Xt, w[1:]) + w[0]     # コンパイル(次の行)
```

```
predict = theano.function(inputs=[Xt], givens={w: w}, outputs=net_input)
return predict(X)                    # 実行
```

これに、訓練データを与えて結果をプロットする(コードは省略)



13.2 フィードフォワードニューラルネットワークの活性化関数

- 多層ニューラルネットワークの実装に役立つシグモイド関数の選択肢
  微分可能であればどのような関数でも活性化関数として使える
  ただし、実際には「線形活性化関数」は役に立たない　－　非線形性が必要
  非線形性関数の代表:ロジスティック関数
  　その問題点:シグモイド関数が0を出力する場合、学習に時間がかかる、訓練において極小値に陥る可能性がある　→　双曲線正接(tanh)はよい性質がある

13.2.1 ロジスティック関数ふりかえり

- ロジスティック関数はシグモイド関数の一種、二値分類タスクでサンプルの分類確率のモデル化が可能
  $\varphi_{logistic}(z) = \frac{1}{1+e^{-z}}$　　ここで z = $w^T x = \sum w_i x_i$　(w_0はバイアスユニット)

例: X = [1, 1.4, 1.5]、w = [0, 0.2, 0.4]、$p(X) = \frac{1}{1+e^{-w^T X}}$　とすると、p(X) = 0.707 のように入力データがクラス1に分類される確率(解釈できるような値)が計算できる

- 複数のロジスティック活性化関数を用いた場合は、確率と解釈可能な値を出力するものではない

その例:　(コードの一部と結果のみ表示)

```
Z = W.dot(A)
y_probas = logistic(Z)
print('Probabilities:¥n', y_probas)
Probabilities:
 [[ 0.87653295]
 [ 0.57688526]
 [ 0.90114393]]
```

この三つの数値を足すと1を超えええる

確率ではなく、各クラスへの所属可能性を示すものとみなせばよい⇒ソフトマックス関数という一般化

13.2.2 ソフトマックス関数を用いた多クラス分類の確率推定

- ソフトマックス(softmax)関数： ロジスティック回帰の一般化、多クラス分類におけるクラス所属確率の計算に利用される　$P(\mathrm{y}=1 \mid \mathrm{z}) = \varphi_{softmax}(z) = \frac{e_i^z}{\sum_{m=1}^{M} e_m^z}$

  (この式は、ベイズの定理において、分母を周辺確率、分子はその一部の確率、としたのを思い起こさせる)

- Python による実装: (softmax_activiation 関数はこの時点では不要)

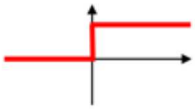
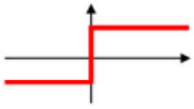
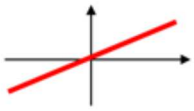
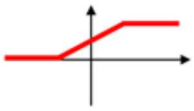
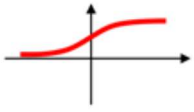
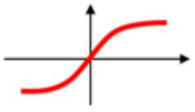
```
def softmax(z):       # ソフトマックス関数
    return np.exp(z) / np.sum(np.exp(z))
y_class = np.argmax(Z, axis=0)          # クラス分類
```

13. 2. 3　双曲線正接関数を用いて出力範囲を拡大する

- 双曲線正接関数(tanh: hyperbolic tangent)　$\varphi_{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^-} = 2\varphi_{logistic}(2z) - 1$

  開区間(-1,1)の値域をとる⇒バックプロパゲーションアルゴリズムの収束を改善

- いろいろな活性化関数のまとめ:

Activation function	Equation	Example	1D Graph
Unit step 単位ステップ（ヘビサイド）	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	パーセプトロン	
Sign (Signum) 符号	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	パーセプトロン	
Linear 線形	$\phi(z) = z$	ADALINE 線形回帰	
Piece-wise linear 区分線形	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	SVM(サポートベクターマシン)	
Logistic (sigmoid) ロジスティック	$\phi(z) = \frac{1}{1 + e^{-z}}$	ロジスティック回帰 多層ニューラルネット	
Hyperbolic tangent 双曲線正接	$\phi(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	多層ニューラルネット	

13.3　Keras を用いたニューラルネットワークの学習

- Keras: Theano をベース（今では Tensorflow も）としたライブラリ、2015 年から開発。GPU を使ったニューラルネットワークの学習の高速化が可能

ニューラルネットワークの実現を容易にする直観的な API

pip install Keras　によってインストール

- backend を切り替える方法

(1) ~/.keras/keras.json　を編集する

デフォルトでは Theano が backend になっているはずなので

```
{
    "image_dim_ordering": "tf",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "theano"
}
```

という内容になっている(注: Json 形式だが、Python の dict と思って読めばよい、項目の順番は無視してよい)。この backend の値を"theano"から"tensorflow"に変えれば、backend が変更できる

(2)環境変数を定義する。次のように KERAS_BACKEND という変数の設定をすれば変更可能

```
KERAS_BACKEND=tensorflow python -c "from keras import backend"
```

- MNIST データセットの手書き数字分類の多層パーセプトロンの実装

load_mnist は 12 章と同じもの。次も 12 章と同様

```
X_train, y_train = load_mnist('mnist', kind='train')
X_test, y_test = load_mnist('mnist', kind='t10k')
```

学習データの準備: MNIST の画像配列を 32 ビット形式に変換

```
import theano
theano.config.floatX = 'float32'
X_train = X_train.astype(theano.config.floatX)
X_test = X_test.astype(theano.config.floatX)
```

クラスラベルを one-hot 形式に変換（Keras のツールを使用）

```
from keras.utils import np_utils
print('First 3 labels: ', y_train[:3])
y_train_ohe = np_utils.to_categorical(y_train)
print('¥nFirst 3 labels (one-hot):¥n', y_train_ohe[:3])
```

その結果:

```
First 3 labels:  [5 0 4]
First 3 labels (one-hot):
[[0.  0.  0.  0.  0.  1.  0.  0.  0.  0.]# '5'
 [1.  0.  0.  0.  0.  0.  0.  0.  0.  0.]      # '0'
 [0.  0.  0.  0.  1.  0.  0.  0.  0.  0.]]     # '4'
```

ニューラルネットワークの実装:12 章とほぼ同じアーキテクチャを採用。

ただし隠れ層を1つ追加し、活性化関数をロジスティック関数から双曲線正接関数に、出力層の活性化関数はソフトマックス関数に変更

```
from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import SGD
np.random.seed(1)        # 乱数の初期化

model = Sequential()                              # モデルの初期化
model.add(Dense(input_dim=X_train.shape[1],   # 隠れ層1:  入力ユニット数(=入力データの次元数)
                output_dim=50,                    # 出力ユニット数
                init='uniform',                   # 重みを一様乱数で初期化
                activation='tanh'))               # 双曲線正接関数を活性化関数に
model.add(Dense(input_dim=50,                  # 隠れ層2:  入力ユニット数=出力ユニット数=50
                output_dim=50,
                init='uniform',
                activation='tanh'))
model.add(Dense(input_dim=50,                  # 出力層:  入力ユニット数=50
                output_dim=y_train_ohe.shape[1],      # 出力ユニット数=カテゴリ数
                init='uniform',
                activation='softmax'))                # ソフトマックス関数を活性化関数に

sgd = SGD(lr=0.001, decay=1e-7, momentum=.9)      # 確率的勾配降下法(学習率、減衰定数等設定)
model.compile(loss='categorical_crossentropy', optimizer=sgd,
                        metrics=['accuracy'])     # モデルのコンパイル(metrics オプション必要)
```

- keras.models では Sequential しか定義されていないようである:これがフィードフォワードニューラルネットワークのモデル。モデルの初期化に続いて層を追加する。最初は入力を受けるので、「入力層になる」
  注意:本文中の別なところでは「最初の隠れ層」とある。実際には「入力ユニットだけの層」を入力層とよび、入力層から重み係数をかけた入力を受けて次の層に渡すのが「隠れ層」。Keras は「(本来の)層と層をつなぐネットワークを「層」と呼んでいるので、誤解しないように。
- 上の Keras のコードではバイアスユニットがないことにも注意(値が 0 となっている)
  出力層のユニットの個数は one-hot 形式のクラスラベル配列の列の個数
  モデルのコンパイルにはオプティマイザの定義が必要:ここでは確率的勾配降下法
  cross-entropy の指定:ソフトマックス関数で他クラス分類において交差エントロピーを一般化

```
model.fit(X_train, y_train_ohe,                  # 学習
          nb_epoch=50,                           # エポック数
          batch_size=300,                        # バッチサイズ
          verbose=1,                             # 実行時に表示
          validation_split=0.1)                  # 検証用データの割合
```

- モデルの学習に fit を用いる－バッチあたり 300 学習、50 エポックにわたって学習

validation_split 引数の仕様により、エポック後の検証が可能

```
Train on 54000 samples, validate on 6000 samples
Epoch 1/50
54000/54000 [==============================] - 0s - loss: 2.2290 - acc: 0.3595 - val_loss: 2.1092 - val_acc: 0.5357
Epoch 2/50
54000/54000 [==============================] - 0s - loss: 1.8840 - acc: 0.5285 - val_loss: 1.6066 - val_acc: 0.5635
（中略）
Epoch 49/50
54000/54000 [==============================] - 0s - loss: 0.2015 - acc: 0.9411 - val_loss: 0.1887 - val_acc: 0.9440
Epoch 50/50
54000/54000 [==============================] - 0s - loss: 0.1996 - acc: 0.9416 - val_loss: 0.1876 - val_acc: 0.9473

train_acc = np.sum(y_train == y_train_pred, axis=0) / X_train.shape[0]
print('Training accuracy: %.2f%%' % (train_acc * 100))
Training accuracy: 94.02%
y_test_pred = model.predict_classes(X_test, verbose=0)
test_acc = np.sum(y_test == y_test_pred, axis=0) / X_test.shape[0]
print('Test accuracy: %.2f%%' % (test_acc * 100))
Test accuracy: 93.30%
```

turingでやってみたところ教科書とは異なる結果になった。また12章の結果(Training accuracy: 97.59%、Test accuracy: 95.62%)より若干悪い結果となった（学習回数が違う）。

参考: 学習回数を 500 エポックにして試した結果(turing の python3+ keras+Theano 使用)

Training accuracy: 98.05%

Test accuracy: 95.94%

過学習気味であることと、テスト材料に対する正解率はほぼ同等となった

チューニングパラメタを最適化していないため正解率が低い ⇒ 学習率、モーメンタム、荷重減衰、隠れユニットの個数を調整する必要がある

注: Keras 以外に、Pylearn2, Lasagne などのライブラリがある

著者による注意:「深層学習」はこの本のテーマではない

Hinton, Ng, LeCun, Schmidhuber, Bengio など第一人者の研究に注目する必要がある

Scikit-learn, Theano, Keras などのメーリングリストへの参加も必要

白井が調べた注意点: keras の backend は tensorflow と theano が選べるが、それぞれの画像の入力チャンネルが違うので、どちらかを対象としたプログラムを backend を変えると次元のエラーが出ることがある

追記: https://github.com/transcranial/keras-js Keras をウェブブラウザ上で実行する JavaScript ライブラリ

Chainer による MNIST との比較

Keras

```
from keras.models import Sequential
from keras.layers.core import Dense
from keras.optimizers import SGD



model = Sequential()
model.add(Dense(input_dim=X_train.shape[1],
                output_dim=50,
                init='uniform',
                activation='tanh'))

model.add(Dense(input_dim=50,
                output_dim=50,
                init='uniform',
                activation='tanh'))

model.add(Dense(input_dim=50,
                output_dim=y_train_ohe.shape[1],
                init='uniform',
                activation='softmax'))

sgd = SGD(lr=0.001, decay=1e-7, momentum=.9)
model.compile(loss='categorical_crossentropy',
      optimizer=sgd, metrics=[’accuracy’])

model.fit(X_train, y_train_ohe,
          nb_epoch=50,
          batch_size=300,
          verbose=1,
          validation_split=0.1)

y_test_pred = model.predict_classes(X_test, verbose=0)
test_acc = np.sum(y_test == y_test_pred, axis=0) /
X_test.shape[0]
```

Chainer (tutorial の example から)

```
import chainer
import chainer.functions as F
import chainer.links as L
from chainer import training
from chainer.training import extensions


# Network definition
class MLP(chainer.Chain):
    def __init__(self, n_in, n_units, n_out):
        super(MLP, self).__init__(
            l1=L.Linear(n_in, n_units),  # first layer
            l2=L.Linear(n_units, n_units),  # second layer
            l3=L.Linear(n_units, n_out),  # output layer
        )
    def __call__(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)

model = L.Classifier(MLP(784, args.unit, 10))
# Setup an optimizer
optimizer = chainer.optimizers.Adam()
optimizer.setup(model)
# Set up a trainer
updater = training.StandardUpdater(train_iter, optimizer,
  device=args.gpu)
trainer = training.Trainer(updater, (args.epoch, 'epoch'),
  out=args.out)
# Evaluate the model with the test dataset for each epoch
trainer.extend(extensions.Evaluator(test_iter, model,
    device=args.gpu))

# Run the training
trainer.run()
```

id:aidiary「深層学習ライブラリ Keras」(2016-03-28) http://aidiary.hatenablog.com/entry/20160328/1459174455

- ニューラルネットの構造は model にさまざまなレイヤを add()することで構築する。
- 活性化関数や Dropout も層とみなす。
- model のコンパイル時に最適化アルゴリズムと誤差関数(例では交差エントロピー)を指定する。誤差関数として平均二乗誤差も使える。
- 最適化アルゴリズムは基本的な SGD からより効率的な Adam、RMSprop までそろっている。実装も簡単
- 学習は scikit-learn と同じく fit()というメソッドでできる。Keras は基本的に教師あり学習しか考えておらず RBM のような教師なし学習を実装するには自作が必要。
- 収束判定の Early-stopping はコールバックとして実装されており、収束したら自動的にループが止まるようになっている。

```
early_stopping = EarlyStopping(monitor='val_loss', patience=2)
# training
hist = model.fit(X_train, y_train,
    batch_size=batch_size, verbose=1,
    nb_epoch=nb_epoch, validation_split=0.1,
    callbacks=[early_stopping])
```

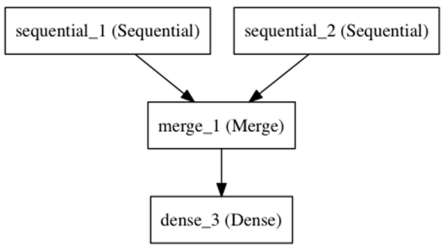
- fit()の戻り値の history オブジェクトに各エポックのコストや精度が保存されるため、あとで結果をプロットするのに使える。
- verbose=1 と指定することで学習の進捗をリアルタムに棒グラフで表示してくれる。各エポックにかかった時間、訓練データの loss/accuracy、バリデーションデータの loss/accuracy も一緒に表示される。
- 畳み込みニューラルネットやリカレントニューラルネットを書く機能もある。最先端の論文に出てくるようなアルゴリズムも Keras による実装が公開されている: Deep dream, DCGAN, VGG-16, Deep Q-learning, Musig Generation, AlphaGo , etc.

Keras: Python の深層学習ライブラリー https://keras.io/ja/

「Sequential モデルで Keras に触れてみよう」 https://keras.io/ja/getting-started/sequential-model-guide/

- Sequential (系列) モデルは層を積み重ねたもの。単純に .add() メソッドを用いて層を追加できる。
- 複数の Sequential のインスタンスをマージ層を使って 1 つの出力にマージできる。出力は新しい Sequential モデルの 1 層目として使える。 以下は 2 つの入力がマージされている:

```
from keras.layers import Merge
left_branch = Sequential()
left_branch.add(Dense(32, input_dim=784))
right_branch = Sequential()
right_branch.add(Dense(32, input_dim=784))
merged = Merge([left_branch, right_branch], mode='concat')
final_model = Sequential()
final_model.add(merged)
final_model.add(Dense(10, activation='softmax'))
```



- 2 つに分岐したモデルの学習

```
final_model.compile(optimizer='rmsprop', loss='categorical_crossentropy')
final_model.fit([input_data_1, input_data_2], targets)  # we pass one data array per model input
```

- これでほとんどのモデルを Keras で実装できる。Sequential や Merge では表現できないより複雑なモデルは Functional API を使って定義できる
- コンパイル
  モデルの学習を始める前に compile メソッドを使って学習経過の設定を行う必要がある。compile は 3 つの引数を取る: 最適化手法、損失関数、評価指標のリスト
- 学習
  Keras のモデルは Numpy の配列のデータとラベルを用いて学習する。典型的には fit 関数を使う

例 1: VGG 風の CNN

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Convolution2D, MaxPooling2D
from keras.optimizers import SGD

model = Sequential()
# input: 100x100 images with 3 channels -> (3, 100, 100) tensors.
# this applies 32 convolution filters of size 3x3 each.
model.add(Convolution2D(32, 3, 3, border_mode='valid', input_shape=(3, 100, 100)))
model.add(Activation('relu'))
model.add(Convolution2D(32, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Convolution2D(64, 3, 3, border_mode='valid'))
model.add(Activation('relu'))
model.add(Convolution2D(64, 3, 3))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
# Note: Keras does automatic shape inference.
model.add(Dense(256))          # 256 次元のベクトルに
model.add(Activation('relu'))
```

```
model.add(Dropout(0.5))

model.add(Dense(10))        # カテゴリ数 10
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

model.fit(X_train, Y_train, batch_size=32, nb_epoch=1)
```

例 2: LSTM を用いた系列データの分類

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

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

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

model.fit(X_train, Y_train, batch_size=16, nb_epoch=10)
score = model.evaluate(X_test, Y_test, batch_size=16)
```

例 3: CNN と Gated Recurrent Unit を用いた画像の説明文生成モデル(単語単位の embedding を使い、説明文は最大で 16 文字) ただし、良い結果を得るためには事前学習で得られた重みで初期化されたより大きい CNN が必要

例 4: 多層 LSTM による系列分類

例 5: 2 つの系列データをそれぞれ別の LSTM encoder に渡し、その結果をマージして分類

keras にモデルの可視化をしてもらおう！ http://www.mathgram.xyz/entry/keras/graph

keras の keras.utils.visualize_util の中にある plot モジュール

例 3: CNN と Gated Recurrent Unit を用いた画像の説明文生成モデル

```
max_caption_len = 16                    #  説明文は最大で 16 文字
vocab_size = 10000                      #  語彙サイズ

# first, let's define an image model that will encode pictures into 128-dimensional vectors.
# it should be initialized with pre-trained weights. (前半は  VGG 風の CNN と同じ)
image_model = Sequential()
image_model.add(Convolution2D(32, 3, 3, border_mode='valid', input_shape=(3, 100, 100)))
image_model.add(Activation('relu'))
image_model.add(Convolution2D(32, 3, 3))
image_model.add(Activation('relu'))
image_model.add(MaxPooling2D(pool_size=(2, 2)))

image_model.add(Convolution2D(64, 3, 3, border_mode='valid'))
image_model.add(Activation('relu'))
image_model.add(Convolution2D(64, 3, 3))
image_model.add(Activation('relu'))
image_model.add(MaxPooling2D(pool_size=(2, 2)))

image_model.add(Flatten())
image_model.add(Dense(128))    #      画像が 128 次元のベクトルに

# let's load the weights from a save file. (あらかじめ学習した重みの取り込み)
image_model.load_weights('weight_file.h5')

# next, let's define a RNN model that encodes sequences of words into sequences of 128-dimensional word vectors
#.単語列を 128 次元の単語ベクトルに変換する RNN モデルの定義
language_model = Sequential()
language_model.add(Embedding(vocab_size, 256, input_length=max_caption_len)) # vocab_size=10000
language_model.add(GRU(output_dim=128, return_sequences=True))
language_model.add(TimeDistributed(Dense(128)))

# let's repeat the image vector to turn it into a sequence.
image_model.add(RepeatVector(max_caption_len))

# the output of both models will be tensors of shape (samples, max_caption_len, 128).
# let's concatenate these 2 vector sequences.
model = Sequential()
```

```
model.add(Merge([image_model, language_model], mode='concat', concat_axis=-1))
# let's encode this vector sequence into a single vector
model.add(GRU(256, return_sequences=False))
# which will be used to compute a probability distribution over what the next word in the caption should be!
model.add(Dense(vocab_size))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy', optimizer='rmsprop')

# images(画像)は numpy の浮動小数点数の配列(サンプル数、チャネル数=3, 幅、高さ)
# captions(説明文)は numpy の整数配列(サンプル数、説明文の最大長さ)
#             部分的な説明文(partial_captions)を表す単語列からなる
#「next_words(次の単語)」は numpy の浮動小数点数配列(サンプル数、語彙サイズ)
#          対応する部分的説明文における次単語のカテゴリコード (0 と 1)からなる
model.fit([images, partial_captions], next_words, batch_size=16, nb_epoch=100)
```

例 4: 多層 LSTM による系列

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

data_dim = 16
timesteps = 8
nb_classes = 10

# expected input data shape: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
               input_shape=(timesteps, data_dim)))  #
returns a sequence of vectors of dimension 32
model.add(LSTM(32, return_sequences=True))  # returns a
sequence of vectors of dimension 32
model.add(LSTM(32)) # return a single vector of dimension 32
model.add(Dense(10, activation='softmax'))

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

# generate dummy training data
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, nb_classes))

# generate dummy validation data
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, nb_classes))

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