Python による機械学習演習

2018-8-1

担当: 鈴木大慈
東京大学大学院情報理工学系研究科数理情報学専攻
e-mail: taiji@mist.i.u-tokyo.ac.jp

1 演習の概要

1.1 演習の目的

- Python の使い方を習得する
- 機械学習の各種手法を実際に動かす

Python は汎用プログラミング言語であり，多数の標準ライブラリに支えられ様々なアプリケーションで用いられている言語である．特に機械学習においては一つの標準的なプログラミング言語である．インタプリタ上で実行されることを前提としており，R や Matlab が使えればそれらと近い感覚で利用できる．

想定環境：

- Python 3.6 (Anaconda): Anaconda は Python の数値計算環境を構築するための様々なパッケージをまとめた無料ディストリビューションである．

1.2 演習の流れ

演習の流れは以下のようになる．

- **第一段階** Python のインストールと基本的な文法を覚える．
- **第二段階** scikit-learn を用いて機械学習手法を用いた簡単なデータ解析を行う．
- **第三段階** より進んだ内容の学習問題を実行し，試行錯誤してみる（選択課題：手書き文字認識 or word2vec）．
- **第四段階** 演習を通して理解した内容を発表．

演習は基本的に本資料に書いてあるコードを順番に実行しゆけばよい．理解を深めるために練習問題の他にも時間が許す限り自分でプログラムを組んでいろいろ試して欲しい．また，このノートのみで python や scikit-learn の全容を説明することは不可能であるため，不明点は適宜自分で調べたり質問しながら進めて欲しい．

1.3 提出物

演習を終えたら以下のものを提出すること．

- Exercise を iPython notebook で解き，適宜説明コメントを加えたものの html ファイルを提出．
- 第四段階の発表資料を提出．

iPython notebook はブラウザ上で python を実行できる環境である．コマンドラインで

```
ipython notebook
```

とすれば起動できる．特に図をブラウザ上に出力するためには notebook の冒頭で

```
%matplotlib inline
```

と入力すれば良い．html ファイルは “File”>“Download as”>“HTML (.html)” で保存できる．なお，本レポートは Google colab といった類似した機能を実現するサービスで行っても構わない．

2 Python の基本的な文法

2.1 行列の扱い

NumPy は行列計算などの数値計算を行うためのパッケージである．NumPy は Anaconda に標準で入っている．Anaconda を使っていない場合は適宜インストールする必要がある．

ここからは NumPy の配列の使い方を示す．

まずは NumPy の import をする:

```
>>> import numpy as np
```

配列・行列の定義 (numpy.ndarray)

```
>>> x = np.array([1, 2, 3])
>>> x
array([1, 2, 3])
>>> x = [1,2,3]
>>> x
[1, 2, 3]
>>>
>>> x = np.array([[1, 2], [0, 3.0]])
>>> x
array([[ 1.,  2.],
       [ 0.,  3.]])
>>> x.T #行列の転置
array([[ 1.,  0.],
       [ 2.,  3.]])
>>>
>>> x = np.array([1.0, 2.0, 3.0])
>>> y = np.array([x, (10, 20, 30)])
>>> y
array([[  1.,   2.,   3.],
       [ 10.,  20.,  30.]])
>>>
>>> x = np.zeros((2,3)) #0 行列
>>> x
array([[ 0.,  0.,  0.],
```

```
       [ 0.,  0.,  0.]])
>>> x = np.ones((3,4)) #1 行列
>>> x
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> x = np.eye(3) #単位行列
>>> x
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>>
>>> x = np.arange(9)
>>> x
array([0, 1, 2, 3, 4, 5, 6, 7, 8])
>>> x = np.arange(2,9)
>>> x
array([2, 3, 4, 5, 6, 7, 8])
>>> x = np.arange(2,9,0.1)
>>> x
array([ 2. ,  2.1,  2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,
        3.1,  3.2,  3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,
        4.2,  4.3,  4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,
        5.3,  5.4,  5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3,
        6.4,  6.5,  6.6,  6.7,  6.8,  6.9,  7. ,  7.1,  7.2,  7.3,  7.4,
        7.5,  7.6,  7.7,  7.8,  7.9,  8. ,  8.1,  8.2,  8.3,  8.4,  8.5,
        8.6,  8.7,  8.8,  8.9])
>>>
>>> x = np.arange(16)
>>> x
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
>>> x = x.reshape((4,4)) #3x3 行列に変更
>>> x.shape
(4, 4)
>>> x
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

行列のインデックス操作

```
>>> x = np.arange(16).reshape((4,4))
>>> x
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> x[1,2]
6
>>> x[0:2]  #1 から 2 行目までを取り出し
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
>>> x[0:2,1:3] #部分行列の取り出し
array([[1, 2],
       [5, 6]])
>>> x[2:]   #2 行以降全部
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
>>> x[:-1]  #最後から 1 行目まで
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> x[:,3]  #4 列めの取り出し
array([ 3,  7, 11, 15])
```

変数型 Python は変数の型を勝手に解釈するが，こちらが明示することにより無用なバグを避けられる．特に丸めには注意が必要.

```
>>> x = x.astype(np.int)
>>> x.dtype
dtype('int32')
>>> x = x.astype(np.float)
>>> x.dtype
dtype('float64')
```

行列の演算

```
>>> a = np.array([[1, 0], [0, 3.0]])
>>> b = np.array([[0, 2], [3, 4.0]])
>>> a+b #行列の足し算
array([[ 1.,  2.],
       [ 3.,  7.]])
>>> np.dot(a,b) #行列の掛け算
array([[  0.,   2.],
       [  9.,  12.]])
>>> a*b #要素ごとの掛け算
array([[  0.,   0.],
       [  0.,  12.]])
```

```
>>>
>>> c = np.array([1,2])
>>> np.linalg.solve(a,c) #ax = c を解く
array([ 1.        ,  0.66666667])
>>> np.linalg.inv(a) #a の逆行列
array([[ 1.        ,  0.        ],
       [ 0.        ,  0.33333333]])
```

乱数の生成

```
>>> #一様分布
... x = np.random.rand(3,4)
>>> x
array([[ 0.13666122,  0.68166613,  0.68950671,  0.67868824],
       [ 0.551999  ,  0.19124743,  0.4551671 ,  0.7007503 ],
       [ 0.84431638,  0.43218764,  0.76020289,  0.61226638]])
>>> #正規分布
... x = np.random.randn(3,4) #標準正規分布 3x4
>>> x
array([[-2.13290801, -0.98600094,  0.21842737,  1.10289088],
       [ 2.28912061, -1.4002147 , -1.14987935, -1.48441992],
       [-1.85269866,  0.39634935, -0.45133648,  1.35437221]])
>>> #二項分布
... x = np.random.binomial(n=100,p=0.4) #表がでる確率 0.4 で 100 回
>>> x
42
>>> #ポアソン
... x = np.random.poisson(lam=3) #λ=10 のポアソン分布
>>> x
3
>>> #どれも size=(10,10) と引数を設定すれば任意の大きさのランダム行列を生成できる
```

2.2 条件分岐と For 分

If 文

```
domain = "us"

if domain == "jp":
    print(u"日本です")
elif domain == "us":
    print(u"アメリカです")
else:
    print(u"その他の国です")
```

For 文

```
>>> x = np.zeros(10)
>>> for i in range(10): #0 から 9 までループ
...     x[i] = i
...
>>> x
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])

# enumerate でループする際にインデックスつきで要素を得ることができる.
>>> x = np.zeros(6)
>>> for i,j in enumerate(range(4,10)):
...     x[i] = j
...
>>> x
array([ 4.,  5.,  6.,  7.,  8.,  9.])

>>> x = [i**2 for i in range(10)] #このような書き方もある.
>>> x
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

#zip で二つのリストを同時にループできる.
>>> list1 = [1, 2, 3]
>>> list2 = [4, 5, 6]
>>> for (a, b) in zip(list1, list2):   #list1,list2 を同時にループ
...     print(a,b)
...
1 4
2 5
3 6
>>> list3 = [7, 8]
>>> for (a, b) in zip(list1, list3): #要素数が少ないリストに合わせてループ
...     print(a,b)
...
1 7
2 8
```

図の描画

図は matplotlib を使えば描画できる.

```
>>> import matplotlib.pyplot as plt
>>> x = np.random.randn(10)
>>> b = 2
>>> y = b*x + np.random.randn(10)*0.3
```

```
>>> xx = np.linspace(min(x),max(x),100)
>>> plt.plot(x,y,"*")
>>> line1 = plt.plot(xx,b*xx,label='true line') #label は凡例のラベル
>>> plt.legend() #凡例を表示
>>> plt.show()
```

2.3 演習：線形回帰

Exercise 1 ある n, p に対して，$X_{i,j} \sim N(0,1)$ (i.i.d.) $(i = 1, \ldots, n, j = 1, \ldots, p)$ と $\beta_j^* \sim N(0,1)$ (i.i.d.) $(j = 1, \ldots, p)$ を生成し，

$$y_i = \sum_{j=1}^{p} X_{i,j}\beta_j^* + \epsilon_i$$

なるモデル（ただし $\epsilon_i \sim 0.5 \times N(0,1)$）に従って y_i も生成せよ．観測データ $\{y_i, (X_{i,j})_j\}_{i=1}^n$ を用いて最小二乗法の解を計算するプログラムを書け．ただし，最小二乗法の解は線形代数の関数を使って求めること．後述のライブラリは使ってはいけない．

$n = 100, p = 1$ として，最小二乗解と真の解と推定誤差を算出せよ．また，生成したデータと真の関数および推定した関数のグラフを書け．

3 scikit-learn を用いた機械学習

scikit-learn は Python の機械学習ライブラリである．豊富な関数が用意されており，scikit-learn のドキュメントページを見るだけで機械学習の勉強ができる．本演習では実データで回帰と判別をしてみる．

とりあえず，以後必要になるパッケージを以下のように読み込んでおく．

```
import numpy as np
from sklearn import datasets
from sklearn import linear_model
from sklearn.svm import SVC
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error
import sklearn.metrics as skmet
```

3.1 回帰

scikit-learn にはデフォルトでいくつかのデータが収録されている．それらは `sklearn.datasets` から呼び出せる．

ここでは，boston housing データを用いて，リッジ回帰をしてみる．

```
#scikit-learn に収録されているデータを使う
boston = datasets.load_boston()
clf = linear_model.Ridge (alpha = .5)
clf.fit(boston.data, boston.target)

predicted = clf.predict(boston.data[-10:-1])
```

```
testy = boston.target[-10:-1]

print(predicted)
print(testy)

#結果の確認とプロット
predicted = clf.predict(boston.data)
testy = boston.target

fig, ax = plt.subplots()
ax.scatter(testy, predicted)
ax.plot([testy.min(), testy.max()], [testy.min(), testy.max()], 'k--', lw=4)
ax.set_xlabel('Measured')
ax.set_ylabel('Predicted')
plt.show()
```

デフォルトのリッジ回帰推定量は

$$\hat{\beta}, \hat{b} = \underset{\beta \in \mathbb{R}^p, b \in \mathbb{R}}{\arg\min} \|Y - X\beta - \mathbf{1}b\|^2 + \alpha\|\beta\|^2$$

で与えられる．$\hat{\beta}, \hat{b}$ はそれぞれ，

```
clf.coef_
clf.intercept_
```

で取り出せる．

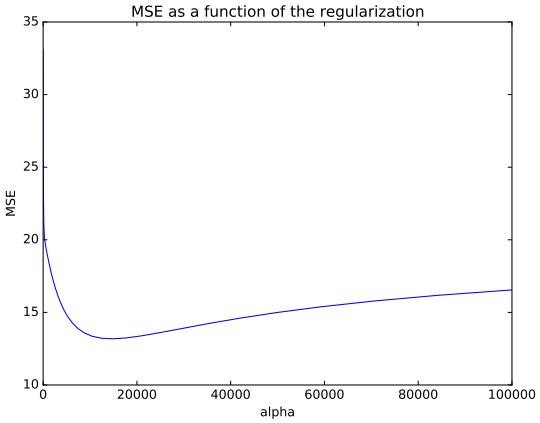
Exercise 2 リッジ回帰推定量を求めるプログラムを自力で作成し，boston housing データを用いて scikit-learn の結果と比較せよ．

次に，正則化パラメータ alpha と予測誤差との関係をプロットしてみる．

```
(n,p) = boston.data.shape
X, X_te = np.split(boston.data, [n*0.8]) #トレーニングデータとテストデータに分割
y, y_te = np.split(boston.target, [n*0.8])  #トレーニングデータとテストデータに分割

n_alphas = 200
alphas = np.logspace(-10, 5, n_alphas)
pred_acc = []
for a in alphas:
    clf.set_params(alpha=a)
    clf.fit(X, y)
    y_pred = clf.predict(X_te)
    pred_acc.append(mean_squared_error(y_te, y_pred))
```

```
plt.plot(alphas, pred_acc)
plt.xlabel('alpha')
plt.ylabel('MSE')
plt.title('MSE as a function of the regularization')
plt.show()
```

Exercise 3 上のコードを実行し，図を出力せよ．以下のような図が出ていれば成功である．



3.2 判別

判別も回帰と同様にして行える．iris データを用いて実行してみよう．

```
iris = datasets.load_iris()
clf = SVC() #Support vector machine
# カーネル関数のデフォルトは rbf カーネル（ガウスカーネル）
clf.fit(iris.data, iris.target)
print(list(clf.predict(iris.data[:3]))) #結果の確認
print(list(iris.target[:3])) #結果の確認
```

回帰と同様に正則化パラメータと判別誤差の関係を見てみる．

```
(n,p) = iris.data.shape
tmp = np.random.permutation(n)
X, X_te = np.split(iris.data[tmp], [n*0.8])
y, y_te = np.split(iris.target[tmp], [n*0.8])

n_Cs = 10
Cs = np.logspace(-2, 2, n_Cs)
# clf.set_params(kernel='linear')
pred_acc = []
for C in Cs:
    clf.set_params(C=C)
    clf.fit(X,y)
```

```
    y_pred = clf.predict(X_te)
    pred_acc.append(skmet.accuracy_score(y_te, y_pred))

print(pred_acc)
```

confusion matrix を出力してみる．

```
C = 1
clf.set_params(C=C)
clf.fit(X,y)
y_pred = clf.predict(X_te)
res1 = skmet.confusion_matrix(y_te, y_pred)
print(res1)
```

Exercise 4 Confusion matrix の意味を調べ，述べよ．また，kernel を線形カーネル (linear kernel) にして判別をし，適当な正則化パラメータにおける confusion matrix を出力せよ．

3.3 大規模データで判別

手書き文字データとして，MNIST データセットを用いた判別を行う．MNIST データは以下のコマンドでダウンロードできる．

```
from sklearn.datasets import fetch_mldata
mnist = fetch_mldata('MNIST original', data_home=".")
```

データをランダムに抽出し，描画すると図 1 のようになる．

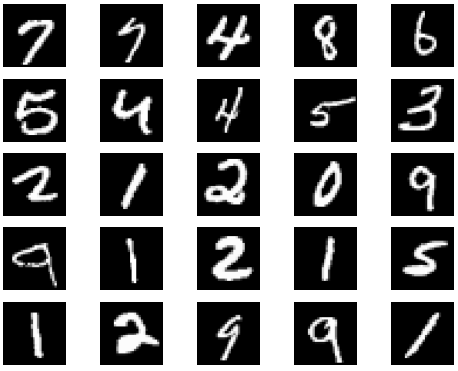


図 1　MNIST データの一部

なお，描画は以下のようにして行った．

```
p = np.random.random_integers(0, len(mnist.data), 25)
fig = plt.figure(0)
for index, (data, label) in enumerate(zip(mnist.data[p], mnist.target[p])):
    ax = plt.subplot(5,5,index + 1)
```

```
    ax.imshow(data.reshape(28, 28), cmap = 'gray', interpolation='nearest')
    ax.set_axis_off()

plt.show()
```

10,000 個のデータを用いて線形カーネルで学習してみる．

```
(n,p) = mnist.data.shape
tmp = np.random.permutation(n)
index = tmp[:10000]
index_test = tmp[10000:-1]

X, X_te = np.split(mnist.data[tmp], [10000])
y, y_te = np.split(mnist.target[tmp], [10000])

clf = SVC()

clf.set_params(kernel='linear',C=100)
clf.fit(X,y)
y_pred = clf.predict(X_te)
print(skmet.accuracy_score(y_te, y_pred))
```

正解率はだいたい 0.9096 で，学習にも時間がかかる．

Exercise 5 k-近傍法 (KNeighborsClassifier) など，他の学習方法を試してみよ．判別誤差を改善させることはできるか？

4 深層学習による手書き文字認識（選択課題１）

ここでは，深層学習（Deep learning）を用いて MNIST データの判別を行う．深層学習は人間の脳と似た構造を持った学習モデルであり，いくつもの層を重ねた形になっている．深層学習の詳細は例えば [3, 1] を参考のこと．

深層学習の Python ライブラリは PyTorch，Pylearn，TensorFlow など多数あるが，本演習では手軽さを重視して，Chainer と呼ばれるライブラリを用いる．Chainer のインストールは簡単で，Windows で Anaconda を用いている場合は，コマンドプロンプトで

```
pip install chainer
```

とすれば良い．あとは python のコード中で

```
import chainer
```

のようにすれば良い．

```
#!/usr/bin/env python
import argparse
```

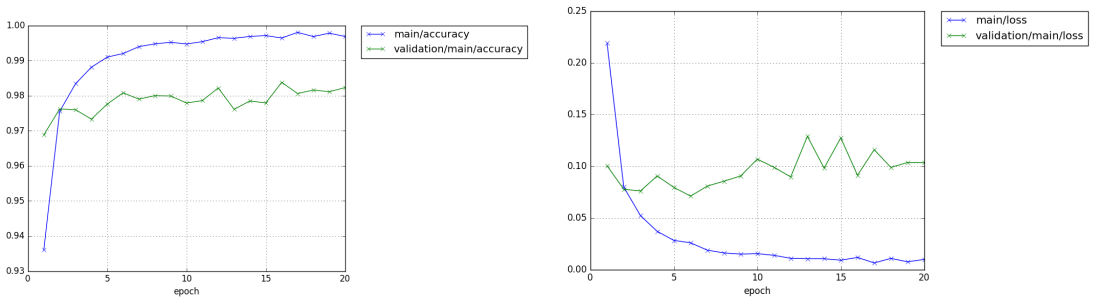
```
import chainer
import chainer.functions as F
import chainer.links as L
from chainer import training
from chainer.training import extensions


# Network definition
class MLP(chainer.Chain):

    def __init__(self, n_units, n_out):
        super(MLP, self).__init__()
        with self.init_scope():
            # the size of the inputs to each layer will be inferred
            self.l1 = L.Linear(None, n_units)  # n_in -> n_units
            self.l2 = L.Linear(None, n_units)  # n_units -> n_units
            self.l3 = L.Linear(None, n_out)  # n_units -> n_out

    def forward(self, x):
        h1 = F.relu(self.l1(x))
        h2 = F.relu(self.l2(h1))
        return self.l3(h2)

# 余裕があれば CNN も試してもらいたい (GPU 利用推奨)
class MyCNN(chainer.Chain):
    def __init__(self, n_units, n_out, stride=1):
        w = math.sqrt(2)
        super(MyCNN, self).__init__(
            conv1 = L.Convolution2D(None, n_units, 3, stride, 1, w, nobias=False),
            conv2 = L.Convolution2D(n_units, n_units, 3, stride, 1, w, nobias=False),
            conv3 = L.Convolution2D(n_units, n_units, 3, stride, 1, w, nobias=False),
            l1 = L.Linear(None, n_out)
        )
        self.train = True

    def __call__(self, x, t):
        h1 = F.relu(self.conv1(x))
        h2 = F.relu(self.conv2(h1))
        h3 = F.relu(self.conv3(h2))
        return self.l3(h3)

if __name__ == '__main__':
```

```
parser = argparse.ArgumentParser(description='Chainer example: MNIST')
parser.add_argument('--batchsize', '-b', type=int, default=100,
                    help='Number of images in each mini-batch')
parser.add_argument('--epoch', '-e', type=int, default=20,
                    help='Number of sweeps over the dataset to train')
parser.add_argument('--frequency', '-f', type=int, default=-1,
                    help='Frequency of taking a snapshot')
parser.add_argument('--gpu', '-g', type=int, default=-1,
                    help='GPU ID (negative value indicates CPU)')
parser.add_argument('--out', '-o', default='result',
                    help='Directory to output the result')
parser.add_argument('--resume', '-r', default='',
                    help='Resume the training from snapshot')
parser.add_argument('--unit', '-u', type=int, default=100,
                    help='Number of units')
parser.add_argument('--noplot', dest='plot', action='store_false',
                    default=True,
                    help='Disable PlotReport extension')

args = parser.parse_args()

print('GPU: {}'.format(args.gpu))
print('# unit: {}'.format(args.unit))
print('# Minibatch-size: {}'.format(args.batchsize))
print('# epoch: {}'.format(args.epoch))
print('')

# Set up a neural network to train
# Classifier reports softmax cross entropy loss and accuracy at every
# iteration, which will be used by the PrintReport extension below.
model = L.Classifier(MLP(args.unit, 10))
if args.gpu >= 0:
    # Make a specified GPU current
    chainer.backends.cuda.get_device_from_id(args.gpu).use()
    model.to_gpu()  # Copy the model to the GPU

# Setup an optimizer
optimizer = chainer.optimizers.Adam()
optimizer.setup(model)

# Load the MNIST dataset
# 初回はダウンロードに時間がかかるが二回目以降は早くなる.
```

```
train, test = chainer.datasets.get_mnist()

# 学習の反復を行う iterator の準備
train_iter = chainer.iterators.SerialIterator(train, args.batchsize)
test_iter = chainer.iterators.SerialIterator(test, args.batchsize,
                                             repeat=False, shuffle=False)

# Set up a trainer (学習の準備はこれで終わり)
updater = training.updaters.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))

# Dump a computational graph from 'loss' variable at the first iteration
# The "main" refers to the target link of the "main" optimizer.
trainer.extend(extensions.dump_graph('main/loss'))

# Take a snapshot for each specified epoch
frequency = args.epoch if args.frequency == -1 else max(1, args.frequency)
trainer.extend(extensions.snapshot(), trigger=(frequency, 'epoch'))

# Write a log of evaluation statistics for each epoch
trainer.extend(extensions.LogReport())

# Save two plot images to the result dir
 # 学習の経過は./result/loss.png や./result/accuracy.png に出力される.
if args.plot and extensions.PlotReport.available():
    trainer.extend(
        extensions.PlotReport(['main/loss', 'validation/main/loss'],
                              'epoch', file_name='loss.png'))
    trainer.extend(
        extensions.PlotReport(
            ['main/accuracy', 'validation/main/accuracy'],
            'epoch', file_name='accuracy.png'))

# Print selected entries of the log to stdout
# Here "main" refers to the target link of the "main" optimizer again, and
# "validation" refers to the default name of the Evaluator extension.
# Entries other than 'epoch' are reported by the Classifier link, called by
# either the updater or the evaluator.
```

```
# ./result/log に途中経過のログが出力される.
trainer.extend(extensions.PrintReport(
    ['epoch', 'main/loss', 'validation/main/loss',
     'main/accuracy', 'validation/main/accuracy', 'elapsed_time']))

# Print a progress bar to stdout
trainer.extend(extensions.ProgressBar())

if args.resume:
    # Resume from a snapshot
    chainer.serializers.load_npz(args.resume, trainer)

# Run the training
trainer.run()



# Save the model and the optimizer
print('save the model')
chainer.serializers.save_hdf5('mlp.model', model)
print('save the optimizer')
chainer.serializers.save_hdf5('mlp.state', optimizer)
```

上のコードを実行した結果，学習の更新回数と判別精度の関係が図 2 のように./result/accuracy.png および./result/loss.png に出力される．



(a) 判別精度　　(b) ロス関数

図 2　深層学習の更新回数と判別精度およびロス関数

Exercise 6 上のコードを理解して，実行せよ．どのような例で誤判別しているだろうか．また，隠れ層の数とユニット数などを変えて，判別精度の変化を観察し，グラフなどにまとめて報告せよ．別のデータセットを用いて実験しても良い．

5　Word2vec を動かす（選択課題２）

Word2vec とは単語のベクトル表現を学習する手法である．これより，

$$\text{‘King’} + \text{‘Woman’} - \text{‘Man’} = \text{‘Queen’}$$

のように単語の足し引きが可能になる．ここでは，自然言語用の gensim ライブラリを用いて word2vec を動かしてみる．

Word2vec を理解するためには CBOW (Continuous Bag-of-Words) と skip-gram を知る必要がある．CBOW は周辺の単語からある単語が出てくる確率を記述するモデルで，一方 skip-gram はその逆のある単語が与えられた時にその周辺にどのような単語が表れやすいかを推定するモデルである．その単語の出現頻度を表すために，各単語 w に θ_w なるベクトル表現を割り当てる．このベクトル表現を用いて，skip-gram ならば，

$$P(w_O|w_I) = \frac{\exp(\langle\theta_{w_O}, \theta_{w_I}\rangle)}{\sum_{w'} \exp(\langle\theta_{w'}, \theta_{w_I}\rangle)}$$

として単語 w_O が w_I の周辺に現れる確率を表現する．あとは，データから θ_w を学習すれば良い（実際は最尤推定）．その際，分母の $\sum_{w'}$ の部分の計算が時間かかかるので，元論文 [2] では negative sampling という手法を提案している．詳しくは [4] を参照．この演習では以下のスライドの内容を理解していれば十分である:

`http://www.slideshare.net/unnonouno/20140206-statistical-semantics`

次のコードでは text8 データセットを用いて word2vec を学習している (コードは電子的に配布予定)．text8 データセットは

`http://mattmahoney.net/dc/text8.zip`

からダウンロードする（下のコードでは`./mldata` なるフォルダに保存してあることを想定)．

```
import sys
import os

from gensim.models import word2vec

train_file = "./mldata/text8"
result_dir = "./result/text8_gensim"

import logging
logging.basicConfig(
    format='%(asctime)s : %(levelname)s : %(message)s',
    level=logging.INFO)



data = word2vec.Text8Corpus(train_file)
model = word2vec.Word2Vec(size=100, window=5, min_count=5, workers=7)

model.build_vocab(data)
```

```
print("*** Training model", file=sys.stderr)
model.train(data)

print("*** Saving model", file=sys.stderr)
model.save_word2vec_format(
    os.path.join(result_dir, "vectors.txt"), binary=False,
    fvocab=os.path.join(result_dir, "vocab.txt"))
```

結果を図示してみる．ある単語のグループに対応した単語ベクトルの集合を主成分分析で二次元に落とし，可視化してみる．

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt
import pandas as pd
from gensim.models import word2vec

result_dir = "./result/text8_gensim"

model = word2vec.Word2Vec.load_word2vec_format(
                        os.path.join(result_dir, "vectors.txt"), binary=False)
wordlist = {'france','paris','germany','berlin','poland','warsaw','moscow','russia'}

df_city = model[wordlist]
pca = PCA(n_components=2)
pca.fit(df_city)

tmp = pca.transform(df_city)
pca_df = pd.DataFrame(tmp)
pca_df.index = wordlist
pca_df.columns = ['PC1','PC2']

ax = pca_df.plot(kind='scatter', x='PC2', y='PC1', figsize=(16,8))

for i, word in enumerate(wordlist):
    ax.annotate(word, (pca_df.iloc[i].PC2, pca_df.iloc[i].PC1))

plt.show()
```

データセット text8 ほどのサイズではそこまで精度は出ないが，上のようなコードでそれらしい結果は得られる．また，gensim は C でコンパイルしていないと python コードが使われ，速度が遅くなる．C でコンパイルすることで大幅な速度改善ができる．環境によっては Cython が入っていれば，高速版が使われる．

Exercise 7 上のコードを理解し，実行して結果を解釈せよ．別の単語の組み合わせで主成分分析した結果を図示せよ．また学習結果をいろいろと操作して，どのような解釈ができるか報告せよ．たとえば

```
model.most_similar('dog')
```

としてみよ．別のデータセットを用いて実験しても良い．

参考文献

[1] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[2] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[3] 岡谷貴之. **深層学習**. 機械学習プロフェッショナルシリーズ. 講談社, 2015.

[4] 西尾泰和. *word2vec* **による自然言語処理**. O'Reilly Japan, 2014.