## denoising Autoencoderの実装

ここでは、sdaの構成要素であるda.pyの動作を説明します。とは言っても、基本的な構成は先に説明したMLPと同じです。ここでは、denoising Autoencoder特有の特徴部分だけを重点的に説明します。

Autoencoderの特徴は非教師学習ということです。このネットワークは、砂時計型のネットワーク(MLP)を構成しますが、最終的な学習は入力と出力が一緒になるように学習することで、学習後にネットワークの隠れ層で得られるベクタを入力の低次元表現とみなすことができます。これを記述しているのが、forward関数です。

```
def forward(self, x_data, train=True):
        y_data = x_data
        # add noise (masking noise)
        x_data = self.get_corrupted_inputs(x_data, train=train)

        x, t = Variable(x_data), Variable(y_data)
        # encode
        h = self.encode(x)
        # decode
        y = self.decode(h)
        # compute loss
        loss = F.mean_squared_error(y, t)
        return loss
```

多層パーセプトロン（MLP）では、y_dataは与えられたラベル情報(1-9の数字）だったのに対し、今回の学習対象であるy_dataは、x_dataと同じものであることがまず宣言されています。これは回帰学習の一種で、この際の学習のコスト関数には最小二乗法の方が良いと言われています。ここでもmean_square_errorが学習に用いられています。

なお、MLPではdropout(relu(...))という関数が書いてありましたが、今回はencode, decodeという名前になっています。しかし、その定義を追っていくと、なんのことはなく、

```
def encode(self, x):
        return F.relu(self.model.encoder(x))

    def decode(self, h):
        return F.relu(self.model.decoder(h))
```

同じ関数が用いられていることがわかります（今回はdropoutは使われていません）。ちょっと冗長ですね・・

denoising Autoencoderのもう一つの特徴は、入力x_dataにノイズをあえて加えることで、ノイズ耐性をつけることです。その関数が、get_corrupted_inputです。

```
def get_corrupted_inputs(self, x_data, train=True):
        if train and self.corruption_level != 0.0:
            mask = self.rng.binomial(size=x_data.shape, n=1, p=1.0 -
self.corruption_level)
            mask = mask.astype(numpy.float32)
            mask = self.xp.asarray(mask)
            ret = mask * x_data
            # return self.xp.asarray(ret.astype(numpy.float32))
            return ret
```

```
        else:
            return x_data
```

numpyのランダム関数を使って、二項分布のノイズがかけられていることがわかります。

1. 砂時計型のネットワークを構成し、入力と出力が同じになるように学習すること
2. 入力にあえてノイズを載せることで、ノイズ耐性のあるネットワークを作ること

この2点がMLPの学習と異なるdenoising Autoencoderの特徴です。

# Stacked denoising Autoencoderの実装

Stacked denoising Autoencoderは、上記で説明したdenoising Autoencoderを多層化したものです。最初の演習で説明した通り、このネットワークはMLP学習の初期値決定に使われることがあります。sda.pyはこのSdAの学習で初期値決定(pre-training)を行った後、MLP学習を行い、効率的に識別学習をおこなうコードになっています。なので、Stacked denoising Autoencoder自体は非教師学習ですが、sda.py全体は教師学習をするコードになっています。

## pre-training: 初期値の決定

pre-trainingは、狭い定義でのStacked denoising Autoencoderの学習です。この学習では、各層のdenoising Autoencoderを順番に学習していきます。

```
def pre_train(self, n_epoch=20, batchsize=100):
        first_inputs = self.data

        # initialize first dAE
        self.dae1 = DA(self.rng,
                       data=first_inputs,
                       n_inputs=self.n_inputs,
                       n_hidden=self.n_hidden[0],
                       corruption_level=self.corruption_levels[0],
                       gpu=self.gpu)
        # train first dAE
        logging.info("--------First DA training has started!--------")
        self.dae1.train_and_test(n_epoch=n_epoch, batchsize=batchsize)
        self.dae1.to_cpu()
```

まず、一層目の学習を行います。次に、一層目の隠れ層の出力を用いて、二層目の学習を行います。

```
    # compute second iputs for second dAE
        tmp1 = self.dae1.compute_hidden(first_inputs[0])
        tmp2 = self.dae1.compute_hidden(first_inputs[1])
        if self.gpu >= 0:
            self.dae1.to_gpu()
        second_inputs = [tmp1, tmp2]

        # initialize second dAE
        self.dae2 = DA(
            self.rng,
            data=second_inputs,
            n_inputs=self.n_hidden[0],
            n_hidden=self.n_hidden[1],
            corruption_level=self.corruption_levels[1],
            gpu=self.gpu
```

first_input[0], first_input[1]は何か追えなくなってきている頃だと思いますが、これは各、x_train, x_testのことです。両方とも１層目のautoencoderで低次元化され、２層目のautoencoderの入力として用いられます。

同じ調子で３層目を学習します。

```
    tmp1 = self.dae2.compute_hidden(second_inputs[0])
        tmp2 = self.dae2.compute_hidden(second_inputs[1])
        if self.gpu >= 0:
            self.dae2.to_gpu()
        third_inputs = [tmp1, tmp2]

        # initialize third dAE
        self.dae3 = DA(
            self.rng,
            data=third_inputs,
            n_inputs=self.n_hidden[1],
            n_hidden=self.n_hidden[2],
            corruption_level=self.corruption_levels[2],
            gpu=self.gpu
        )
        # train third dAE
        logging.info("--------Third DA training has started!--------")
        self.dae3.train_and_test(n_epoch=n_epoch, batchsize=batchsize)
```

最後に、それぞれの層のencoder(入力から低次元化するネットワーク）で学習された変数の値を取り出し、あらかじめ作っておいたMLPのネットワークの初期値として代入します。

```
   # update model parameters
        self.model.l1 = self.dae1.encoder()
        self.model.l2 = self.dae2.encoder()
        self.model.l3 = self.dae3.encoder()

      self.setup_optimizer()
```

これが、pre_trainingの詳細です。

## fine-tuning: MLPの学習

fine_tuning関数は名前こそ変わっていますが、MLPのtrain_and_test関数とほぼ同じ構成をしています。なぜなら、仕組みが全く同じだからです（なので、説明は省きます）

MLPの学習の前に初期値を最適化しておくことで、より効率的に識別学習ができると言われています。

## 課題

では、課題に取り組みましょう。コードを読んでいくと、このプログラムには一点、絶対に必要な機能が足りていません。それは、識別学習をした後、学習されたパラメータを使って、新規のデータで識別をする機能です。今日から3回の演習では、それぞれのクラスに足りない。識別の関数を作っていってもらいます。

### 課題1: ウォーミングアップ

識別関数の作成の前に、ウォーミングアップとして、現在の機能を拡張してみましょう。

- **課題1-1**: 現在、入力層1, 隠れ層5, 出力層1の計7層で構成されているネットワークを、入力層1, 隠れ層7, 出力層1の計9層のネットワークにしてみましょう。
- **課題1-2**: ノイズを発生するget_corrupted_inputs関数を改良し、二項分布以外の分布でノイズをかけ

てみましょう。そして、その結果からなぜ二項分布が用いられているのかを考察しましょう。

**課題2: 識別関数を作ろう**

- **課題2-1**: 一つの手書き文字画像ベクタが入力されたら、その識別結果を返す関数predict()を実装しましょう。また、その関数を使った処理をmain関数に追加しましょう
- **課題2-2**: 上記で実装した関数predict()を元に、入力ベクタの配列とラベルデータの配列を入力とし、その正答率を返す関数accuracy()を実装しましょう
- **課題2-3**: [難] 上記の関数accuracy()を改造し、混同行列を出力する関数confusion_matrix()を実装しましょう。

なお、scipy, numpyの便利関数は好きなだけ使って構いません。なるべく短いコードで簡単に実装することを心がけましょう。[難]と書いてある課題は、難易度が格段に上がっています。簡単すぎてつまらない人用の課題なので、できなかったらできなかったらでよいです。解くためには、ニューラルネットワークへの理解と、python, numpy, scipへの習熟が必要です。