MLP (Multi-Layer Perceptron)

ここでは、mlp.pyのコードが各所で何をやっているのかを解説していきます。

データセットのフォーミュレーション

まず、スクリプトが呼ばれて、真っ先に実行されるmain関数を見ていきましょう。

```
def main(description, gpu, output):

...

  logging.info('fetch MNIST dataset')
  mnist = fetch_mldata(description)
  mnist.data = mnist.data.astype(numpy.float32)
  mnist.data /= 255
  mnist.target = mnist.target.astype(numpy.int32)
  data_train, data_test, target_train, target_test = train_test_split(mnist.data, mnist.target)
  data = data_train, data_test
  target = target_train, target_test
```

main関数の冒頭では上記に引用したように、MNISTのデータを構築したMLPの処理に適用できるようにデータのフォーミュレーションを行います。それらの処理を箇条書きで書くと、

- mnistのデータを取得する：fetch_mldata
- mnistのデータの各要素をfloat32型でCastする。：mnist.data.astype(numpy.float32)
    - 元は0-255までの整数型ですが、これを浮動小数点型に変換します
- mnistのデータを255で割って、正規化する
- 元のデータを学習用のデータと、テスト用のデータに分割する：train_test_split
- 分割したデータをデータとラベルにまとめる：data, target

ここで最も大事な作業は、データの正規化です。活性化関数であるシグモイド関数は0付近で立ち上がるため、あまり絶対値が大きな数で計算すると、うまく動きません。そのため、データを正規化（全体のデータを0-1の値域になるように変換）することでこれらの活性化のシーケンスがうまく動くようにするのです。

また、scipy.normalizeのtrain_test_splitという関数はとても便利な関数で、ある大きなデータセットからランダムで、学習用のデータとテスト用のデータを抽出することができます。オプショナルな引数でそれらの比率を調整することもでき、これから学習をする上でずっと使っていく関数だと思いますので、覚えておきましょう。

MLPクラスの実装

では次にMLPクラスの実装について見ていきましょう。

必要な各要素の定義

MLPクラスのコンストラクタ（__init__関数)では、学習に必要な各クラス変数を定義しています。

```
    def __init__(
        self,
        data,
        target,
```

```
5         n_inputs=784,
6         n_hidden=784,
7         n_outputs=10,
8         gpu=-1
9     ):
10
11        self.model = FunctionSet(
12            l1=F.Linear(n_inputs, n_hidden),
13            l2=F.Linear(n_hidden, n_hidden),
14            l3=F.Linear(n_hidden, n_outputs)
15        )
16
17        if gpu >= 0:
18            self.model.to_gpu()
19
20        self.x_train, self.x_test = data
21        self.y_train, self.y_test = target
22
23        self.n_train = len(self.y_train)
24        self.n_test = len(self.y_test)
25
26        self.gpu = gpu
27        self.optimizer = optimizers.Adam()
28        self.optimizer.setup(self.model)
```

上記がコンストラクタのコードを引用したものです。それぞれ

- ニューラルネットワークの各層の変数の定義：FunctionSet
- GPU使用時のモデル変数のコンバート：self.model.to_gpu
- 学習データ、検証データを、データとラベルに小分け
- データ数の保存self.n_train, n_test
- gpu使用の有無：self.gpu
- 学習器の設定：self.optimizer

を行っています。

フィードフォワードの処理の定義

次に、フィードフォワードの処理について書いています。フィードフォワードの処理は以下の関数foward()で定義されています。

```
1 def forward(self, x_data, y_data, train=True):
2         x, t = Variable(x_data), Variable(y_data)
3         h1 = F.dropout(F.relu(self.model.l1(x)), train=train)
4         h2 = F.dropout(F.relu(self.model.l2(h1)), train=train)
5         y = self.model.l3(h2)
6         return F.softmax_cross_entropy(y, t), F.accuracy(y, t)
```

ここで、数式をおさらいしましょう。各層の式は以下のように表されます。

$$\mathbf{y_{L+1}} = f(\mathbf{W_L}\,\mathbf{x_L} + \mathbf{b_L})$$

このとき y_{L+1} は、$L+1$ 層の出力、$\mathbf{x_L}$ は L 層の入力、$\mathbf{W_L}, \mathbf{b_L}$ はそれぞれ重みを表す行列、バイアス、$f(x)$はシグモイド関数などの活性化関数となります。

このプログラムでは、入出力層が一層ずつ、隠れ層が2層、計4層のフィードフォワード型ネットワークが構成されているので、$\mathbf{x}$が入力されてから、$\mathbf{y}$が得られるまでの式の展開は以下のようになります。プログラムの3,4,5,6行目が以下の式と等価の処理を行っています。

$$\mathbf{h_1} = relu(\mathbf{W_1 x} + \mathbf{b_1})$$

$$\mathbf{h_2} = relu(\mathbf{W_2 h_1} + \mathbf{b_2})$$

$$\mathbf{y} = softmax(\mathbf{W_3 h_2} + \mathbf{b_3})$$

ここで$relu$はReLU(ランプ、Rectified Linear Unit)関数と呼ばれる活性化関数で、シグモイド関数より、学習を効率的に行えることがこれまでの研究で示されています。

また、コードでは見慣れないdropoutという関数がさらに適用されています。dropoutとは、学習の間に故意に表現力を落とす手法のことで、ニューラルネットワークの学習で良く発生する局地に陥る現象を防ぐ上で効果的であると言われています。このコードでは、そのdropoutの処理を入れています。

学習全般の定義

最後に、学習全般の定義をしている関数train_and_test()の中身を見ていきましょう。このコードを総括すると、各試行(epoch)毎に以下の2つの処理を繰り返していることがわかります。

1. パラメータの学習
2. パラメータのテスト

それぞれを順に追っていきます。

パラメータの学習

学習は、ミニバッチ法と呼ばれる手法で行われます。以下、サイトの抜粋です。

> 学習データ記録をほぼ等しいサイズのグループに分割し、1つのグループを渡した後でシナプスの重みを更新します。つまり、ミニバッチ学習では記録のグループの情報を使用します。この処理では、必要に応じてデータグループを再利用します。ミニバッチ学習はバッチ学習とオンライン学習の中間に位置し、「中規模サイズの」データセットの場合に最適です。この手続きでは、ミニバッチ学習の学習記録数を自動的に決定したり、1 より大きな整数またはメモリに格納するケースの最大数以下の整数を指定することができます。[オプション] タブで、メモリー内に格納するケースの最大数を設定できます。

解説の通り、ミニバッチ学習は、バッチ学習（学習データを全部使う）とオンライン学習（学習データを一個ずつ使う）の中間にある手法で、局所最適に陥りにくいという特性を持ちます。

```
    for i in xrange(0, self.n_train, batchsize):
            x_batch = self.xp.asarray(self.x_train[perm[i:i+batchsize]])
            y_batch = self.xp.asarray(self.y_train[perm[i:i+batchsize]])

            real_batchsize = len(x_batch)

            self.optimizer.zero_grads()
            loss, acc = self.forward(x_batch, y_batch)
            loss.backward()
            self.optimizer.update()

            sum_loss += float(cuda.to_cpu(loss.data)) * real_batchsize
            sum_accuracy += float(cuda.to_cpu(acc.data)) * real_batchsize

        self.train_accuracies.append(sum_accuracy / self.n_train)
        self.train_losses.append(sum_loss / self.n_train)
```

上記では、xrange関数でbatchsizeぶんごとに学習を行い、誤差情報と精度情報を加算していきます。最後に、総和をデータサイズで割ってその試行（epoch）での精度と誤差を求め、配列に格納します。なお、

optimizer.zero_grads(), optimizer.backward(), optimizer.update()が学習部分（それぞれ、勾配の初期化、学習、各フック処理の呼び出し）に当たります。詳しくは以下のchainerの解説を読むと良いでしょう。

- http://docs.chainer.org/en/stable/tutorial/basic.html

パラメータのテスト

学習されたパラメータを学習で使っていないデータで再検証するのがその次のループ処理となります。基本的に上記のパラメータの学習と同じ処理を行いますが、使っているデータが、x_trainから、x_testに、学習に当たる処理がないことが見て取れると思います。

```
        for i in xrange(0, self.n_test, batchsize):
            x_batch = self.xp.asarray(self.x_test[i:i+batchsize])
            y_batch = self.xp.asarray(self.y_test[i:i+batchsize])

            real_batchsize = len(x_batch)

            loss, acc = self.forward(x_batch, y_batch, train=False)

            sum_loss += float(cuda.to_cpu(loss.data)) * real_batchsize
            sum_accuracy += float(cuda.to_cpu(acc.data)) * real_batchsize


        self.test_accuracies.append(sum_accuracy / self.n_test)
        self.test_accuracies.append(sum_loss / self.n_test)
```

これらの学習とテストの結果が標準出力に出力されるため、出力で学習の経緯をトレースすることができます。

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2, 出力層1の計4層で構成されているネットワークを、入力層1, 隠れ層4, 出力層1の計6層のネットワークにしてみましょう。
  - ヒント：主にFunctionSetと、forward関数の修正が必要です。
- **課題1-2**: OptimizerでADAMを使っていると思いますが、これを確率的勾配降下法に置き換えましょう。
- **課題1-3**: [難] 現在、1-9の画像をそれぞれラベル1,2,3,4,5,6,7,8,9と識別するように学習していますが、これを9,8,7,6,5,4,3,2,1と識別するように学習してみましょう。

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
  - ヒント：forward関数って何をしているんでしたっけ？train_and_test関数の後半のパラメータのテストでは何をしているんでしたっけ？を考えると答えが出てくると思います。
- **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。
    - 入力は同じだよ。混同行列がわからない人はGoogleってみよう

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。