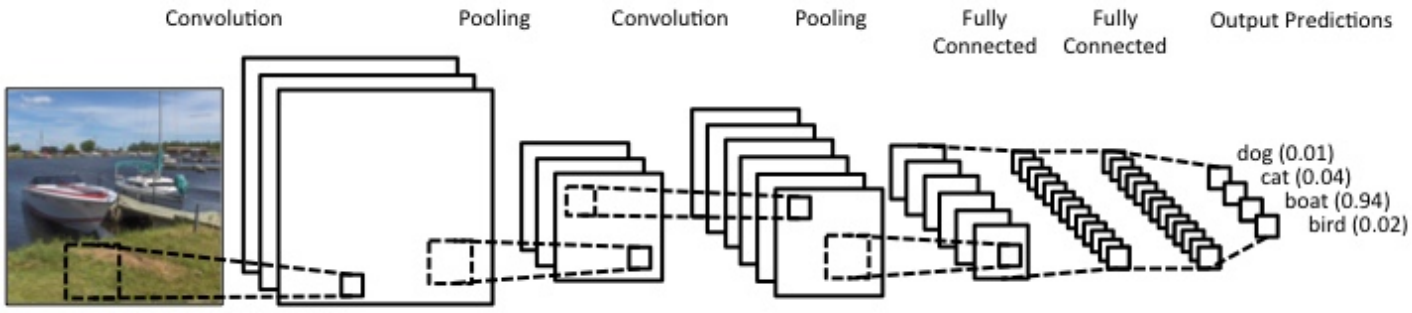
はじめに

CNN(Convolutional Neural Network)は、和名を畳み込みニューラルネットワークと言い、画像の特徴量抽出に特化したネットワークです。名前通り、画像を畳み込んで行く作業を繰り返して識別を行います。



たたみ込む作業は、ConvolutionとPoolingという作業に分かれます。前者が畳み込みを後者が画像サイズの圧縮をそれぞれ担当します。これらの作業を交互に繰り返していくことで、画像の階層構造を各カーネルに学習していくというのが、簡単な畳み込みニューラルネットワークの仕組みです。

畳み込み層の実装

では、その中核となる畳み込み層の実装から見ていきましょう。これまでのネットワークではFunctionSetクラスをそのまま使っていましたが、CNNの実装では、継承して独自の実装を加えています。

```
class CNNModel(FunctionSet):
    def __init__(self, in_channels=1, n_hidden=100, n_outputs=10):
        FunctionSet.__init__(
            self,
            conv1=F.Convolution2D(in_channels, 32, 5),
            conv2=F.Convolution2D(32, 32, 5),
            l3=F.Linear(288, n_hidden),
            l4=F.Linear(n_hidden, n_outputs)
        )

    def forward(self, x_data, y_data, train=True, gpu=-1):
        x, t = Variable(x_data), Variable(y_data)
        h = F.max_pooling_2d(F.relu(self.conv1(x)), ksize=2, stride=2)
        h = F.max_pooling_2d(F.relu(self.conv2(h)), ksize=3, stride=3)
        h = F.dropout(F.relu(self.l3(h)), train=train)
        y = self.l4(h)
        return F.softmax_cross_entropy(y, t), F.accuracy(y, t)

    def predict(self, x_data, gpu=-1):
        x = Variable(x_data)
        h = F.max_pooling_2d(F.relu(self.conv1(x)), ksize=2, stride=2)
        h = F.max_pooling_2d(F.relu(self.conv2(h)), ksize=3, stride=3)
        h = F.dropout(F.relu(self.l3(h)), train=train)
        y = self.l4(h)
        sftmx = F.softmax(y)
        out_data = cuda.to_cpu(sftmx.data)
```

```
27        return out_data
```

実はここにpredictの実装があるので、こちらを先に見ていると前の課題の一番の難関が乗り越えられたりします。

余談はともかく、この実装がMLPの実装と異なる点は以下のとおりです。

- 構成するネットワークの２段までが、畳み込み層である
- 畳み込み層では、LinearクラスではなくConvolutional2Dクラスを使う
- フォワードの計算にmax_poolingが入っている

つまり畳み込み層の部分だけ新しいクラスと関数を使って計算しているだけです。

畳み込み

では畳み込みの処理から見ていきましょう。FunctionSetに格納されている関数のうち、畳み込みの処理に該当するのは以下の行です。

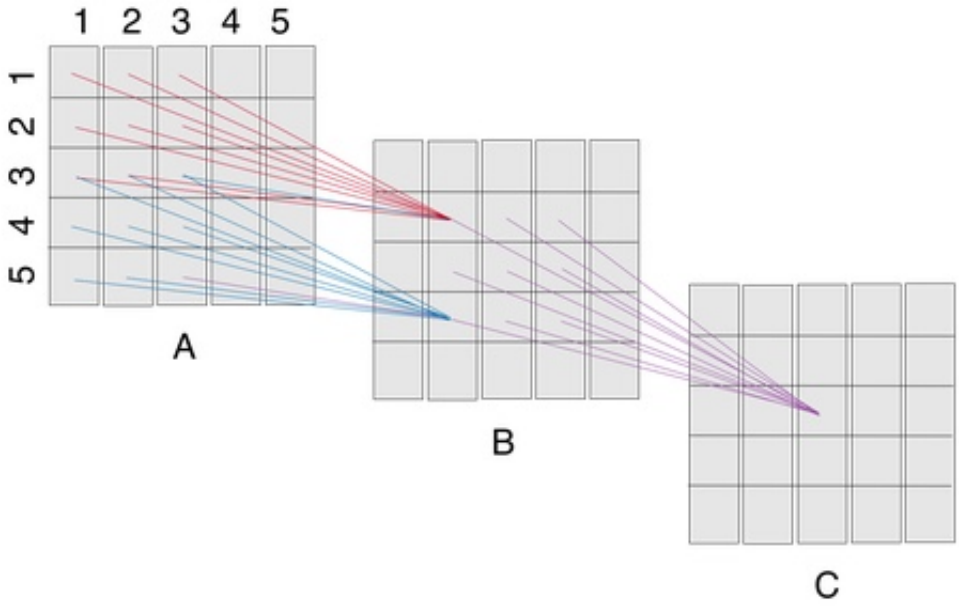
> conv1=F.Convolution2D(in_channels, 32, 5),

この関数の定義を見てみましょう。

> class chainer.links.Convolution2D(in_channels, out_channels, ksize, stride=1, pad=0, wscale=1, bias=0, nobias=False, use_cudnn=True, initialW=None, initial_bias=None)[source] Two-dimensional convolutional layer.
>
> This link wraps the convolution_2d() function and holds the filter weight and bias vector as parameters.
>
> Parameters:
>
> > in_channels (int) – Number of channels of input arrays.
> > out_channels (int) – Number of channels of output arrays.
> > ksize (int or pair of ints) – Size of filters (a.k.a. kernels). ksize=k and ksize=(k, k) are equivalent.
> > stride (int or pair of ints) – Stride of filter applications. stride=s and stride=(s, s) are equivalent.
> > pad (int or pair of ints) – Spatial padding width for input arrays. pad=p and pad=(p, p) are equivalent.
> > wscale (float) – Scaling factor of the initial weight.
> > bias (float) – Initial bias value.
> > nobias (bool) – If True, then this link does not use the bias term.
> > use_cudnn (bool) – If True, then this link uses cuDNN if available.
> > initialW (4-D array) – Initial weight value. If None, then this function uses to initialize wscale. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.
> > initial_bias (1-D array) – Initial bias value. If None, then this function uses to initialize bias. May also be a callable that takes numpy.ndarray or cupy.ndarray and edits its value.

上記にある通り、Convolution2Dの引数のうち、第一引数は入力されるチャネル数（RGB画像の場合は3チャネル、白黒画像の場合は1チャネル）、第二引数は出力されるチャネル数、つまりこの層で何種類の画像の部品パターンを想定するかを表します。第三引数のksizeはカーネルの大きさを表します。畳み込むとき読み込む画像の区画の大きさを表していると思ってください。今回の実装では、5が指定されているので、5x5の大きさの区画で畳み込まれることになります。この大きさは縦横で別のサイズを指定することも可能で、その場合はタプル型でその大きさを指定することになります。その他に、どのように区画をスライドさせるのかを決めるstrideなどいろいろなパラメータがあるので、CNNのパラメータの組み合わせは相当数

になると思ってください。



プーリング

さて、次にforward関数の実装を見ましょう。

```
h = F.max_pooling_2d(F.relu(self.conv1(x)), ksize=2, stride=2)
```

上記は畳み込み層の計算部分の一行を抜き出したものです。畳み込まれた画像の各成分に対して、relu関数が適用され、その出力をさらにpoolingしています。poolingは先に述べたように画像サイズを圧縮する作業です。poolingにはいくつかの種類があり、ウィンドウの中から最大値をとる最大値プーリング(max pooling)や平均値をとる平均値プーリング(average pooling)などがあります。今回はmax_poolingを使っていますね。このとき、ksizeは圧縮する際のウィンドウサイズを、strideはウィンドウの動かし方をそれぞれ表します。今回の実装の場合、ksizeが2なので、2x2のウィンドウサイズを1にするという作業を行うことになります。さらにstrideも2なので、重複部分なしで単純に画像を1/4にしていることがわかります。

畳み込み層からフィードフォワードへの展開

最後に畳み込み層からMLPへ展開する層の定義を見てみましょう。

```
l3=F.Linear(288, n_hidden),
```

畳み込み層は基本的に２次元画像がチャネル分ある構造になっており、これをすべて横に展開するという作業をしているのが、上記の3層目の処理になります。ここで、in_channelが288なのはなぜでしょうか？

簡単に計算してみましょう。

最初は28x28の画像があります。これが2つの畳み込み層でそれぞれ1/4にされて、3x3の画像となります。計算が合いませんね。追っていきましょう。

さて、畳み込みでのフィルタリングで画像は若干小さくなります。そのサイズは、

$$in\_channel - ksize + 1$$

となります。次にプーリング層で単純に1/ksizeで圧縮されます。これを2層繰り返すと、各チャネルの次元

数は以下のように遷移していきます。

> 1st conv: (28-5+1 , 28-5+1) = (24, 24)
> 1st pool: (24/2, 24/2) = (12, 12)
> 2nd conv: (12-5+1, 12-5+1) = (8, 8)
> 2nd pool: (8/3, 8/3) = (3, 3) # 端っこは2x2の部分が残りそれを最大値プーリングしている

上記で出力された3x3の画像が32チャネル分あるので、それらを全て横展開すると288次元となるのです。

$$3 \times 3 \times 32 = 288$$

この計算を自分のものにするには何度か実際にやってみるのが一番です。後述の課題でやってみましょう。

変わらぬtrain_and_test関数とデータの前処理

ここまで定義すればあとは他のDNNクラスと変わりません。特に、train_and_test関数はほとんどと言って良いほどその定義が変わらないのが見て取れると思います。

```
    def train_and_test(self, n_epoch=20, batchsize=100):
        epoch = 1
        while epoch <= n_epoch:
            logging.info('epoch {}'.format(epoch))

            perm = numpy.random.permutation(self.n_train)
            sum_train_accuracy = 0
            sum_train_loss = 0
            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.model.forward(x_batch, y_batch, train=True,
gpu=self.gpu)
                loss.backward()
                self.optimizer.update()

                sum_train_loss += float(loss.data) * real_batchsize
                sum_train_accuracy += float(acc.data) * real_batchsize

            logging.info(
                'train mean loss={}, accuracy={}'.format(
                    sum_train_loss / self.n_train,
                    sum_train_accuracy / self.n_train
                )
            )
            self.train_accuracies.append(sum_train_accuracy / self.n_train)
            self.train_losses.append(sum_train_loss / self.n_train)

            # evalation
            sum_test_accuracy = 0
            sum_test_loss = 0
            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)
```

```
40
41                 loss, acc = self.model.forward(x_batch, y_batch, train=False,
   gpu=self.gpu)
42
43                 sum_test_loss += float(loss.data) * real_batchsize
44                 sum_test_accuracy += float(acc.data) * real_batchsize
45
46             logging.info(
47                 'test mean loss={}, accuracy={}'.format(
48                     sum_test_loss / self.n_test,
49                     sum_test_accuracy / self.n_test
50                 )
51             )
52             self.test_accuracies.append(sum_test_accuracy / self.n_test)
53             self.test_losses.append(sum_test_loss / self.n_test)
54
55             epoch += 1
```

データの前処理で一つだけこれまでのDNNのメソッドと違う点は、入力ベクタを3次元のマトリックスに変換しているところです。main関数の該当部分を抜き出してみましょう。

```
1     mnist.data = mnist.data.astype(numpy.float32)
2     mnist.data /= 255
3     mnist.data = mnist.data.reshape(70000, 1, 28, 28)
4     mnist.target = mnist.target.astype(numpy.int32)
```

いままで、28x28=784の１次元配列だったmnist.dataが1チャネル x 28タテ x 28ヨコの3次元のマトリックスとなっています。これがCNNに入力を渡す時のお作法なので、覚えておきましょう。

課題

では、課題に取り組みましょう。

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**: CNNの2層目のksize, strideを3から2に変更して後述の処理を書いてみましょう。フィードフォワードに展開するところの計算を自分でやってみることが目的です。
- **課題1-2**: 畳み込み層のチャネル数を変えてみて、その識別率の変化を確認し、なぜその変化が訪れたのかを確認しましょう。

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。predictという関数はすでに実装されていますが、現状、まだこちらの要求する仕様にはなっていません。手書き文字ベクタを一つ入れると結果を一つ返すように改造してください。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。