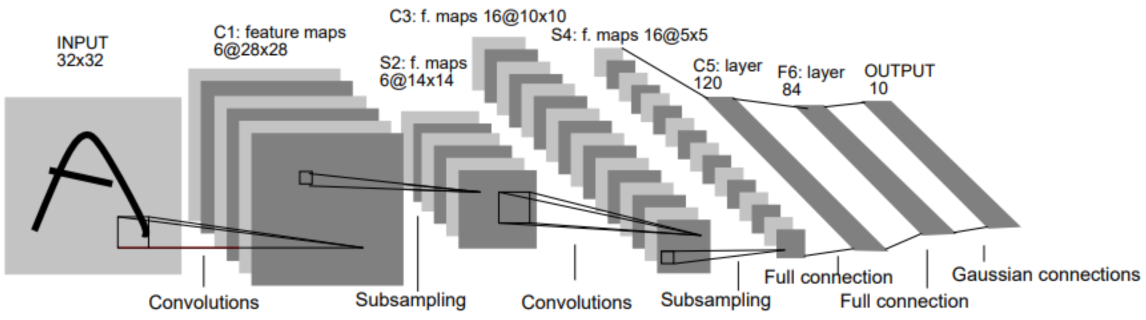
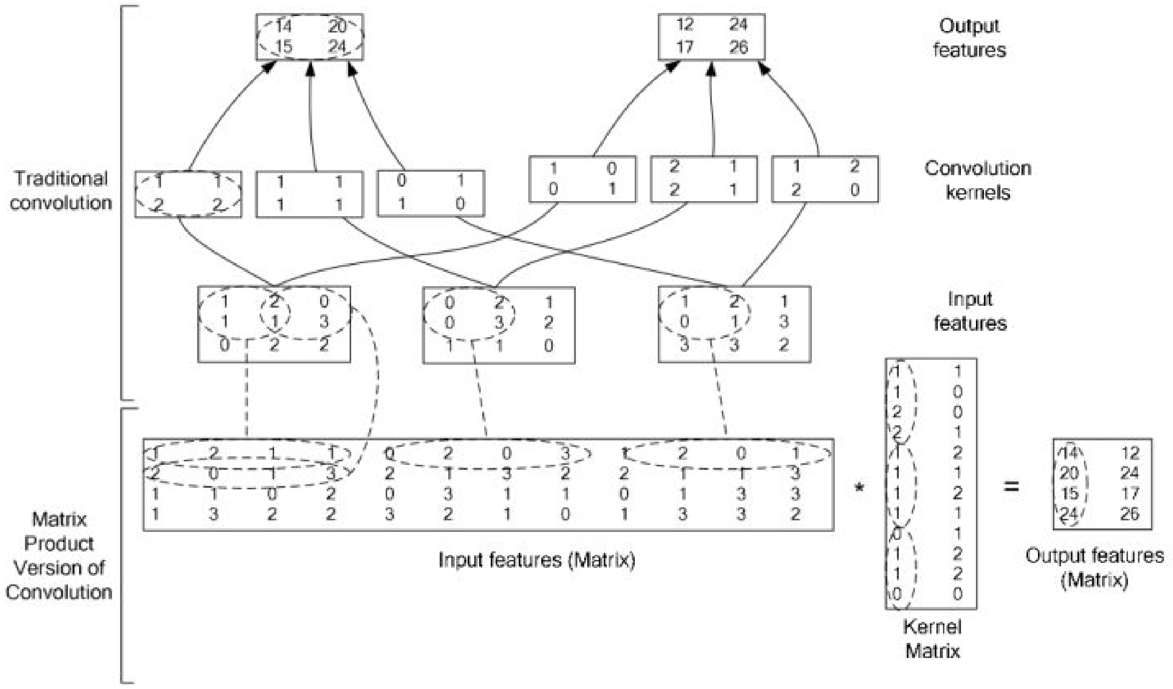
LeNet结构



Conv

原代码中实现的卷积采用的是暴力循环计算卷积结果的方式，运算速度较慢，我们可以通过im2col的方式加速卷积的运算，通过将图像和卷积核转换成行列向量的形式进行矩阵运算，提高缓存命中率，达到加速卷积运算的目的



```python
def im2col(image, kernel_size, stride):
    (N, C, H, W) = image.shape
    image_col = []
    for i in range(0, H - kernel_size + 1, stride):
        for j in range(0, W - kernel_size + 1, stride):
            col = image[:, :, i:i+kernel_size, j:j+kernel_size].reshape(-1)
            image_col.append(col)
    image_col = np.array(image_col)
    return image_col
```

以下速度测试代码在`conv_speed_test.ipynb`文件中

以`batch_size`为16，大小为28*28的图像作为输入数据

naive_Conv

```
class naive_Conv():
    """
    Conv layer
    """
    def __init__(self, Cin, Cout, F, stride=1, padding=0, bias=True):
        self.Cin = Cin
        self.Cout = Cout
        self.F = F
        self.S = stride
        self.W = {'val': np.random.normal(0.0,np.sqrt(2/Cin),(Cout,Cin,F,F)),
'grad': 0}
        self.b = {'val': np.random.randn(Cout), 'grad': 0}
        self.cache = None
        self.pad = padding

    def forward(self, X):
        X = np.pad(X, ((0,0),(0,0),(self.pad,self.pad),(self.pad,self.pad)),
'constant')
        (N, Cin, H, W) = X.shape
        H_ = H - self.F + 1
        W_ = W - self.F + 1
        Y = np.zeros((N, self.Cout, H_, W_))

        for n in range(N):
            for c in range(self.Cout):
                for h in range(H_):
                    for w in range(W_):
                        Y[n, c, h, w] = np.sum(X[n, :, h:h+self.F, w:w+self.F] *
self.W['val'][c, :, :, :]) + self.b['val'][c]

        self.cache = X
        return Y

    def backward(self, dout):
        X = self.cache
        (N, Cin, H, W) = X.shape
        H_ = H - self.F + 1
        W_ = W - self.F + 1
        W_rot = np.rot90(np.rot90(self.W['val']))

        dX = np.zeros(X.shape)
        dW = np.zeros(self.W['val'].shape)
        db = np.zeros(self.b['val'].shape)

        # dW
        for co in range(self.Cout):
            for ci in range(Cin):
                for h in range(self.F):
                    for w in range(self.F):
                        dW[co, ci, h, w] = np.sum(X[:,ci,h:h+H_,w:w+W_] *
dout[:,co,:,:])

        # db
        for co in range(self.Cout):
            db[co] = np.sum(dout[:,co,:,:])
        dout_pad = np.pad(dout, ((0,0),(0,0),(self.F,self.F),(self.F,self.F)), 
'constant')

        # dX
        for n in range(N):
            for ci in range(Cin):
                for h in range(H):
                    for w in range(W):
                        dX[n, ci, h, w] = np.sum(W_rot[:,ci,:,:] * dout_pad[n, 
:, h:h+self.F,w:w+self.F])

        return dX
```

```
naive_conv = naive_Conv(1,6,5)
%timeit naive_conv.forward(image)
484 ms ± 13.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
out = naive_conv.forward(image)
%timeit naive_conv.backward(out)
132 ms ± 2.58 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

im2col_Conv

```
class im2col_Conv():
    def __init__(self, in_channels, out_channels, kernel_size, stride=1):
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.weight_size = (self.out_channels, self.in_channels, 
self.kernel_size, self.kernel_size)
        self.stride = stride

        self.W = {'val': np.random.standard_normal(self.weight_size), 'grad': 0}
        self.b = {'val':  np.random.standard_normal((self.out_channels,1)), 
'grad': 0}

        self.cache = None

    def forward(self, x):
        (N,C,H,W) = x.shape
        self.input_shape = x.shape
        H_out = (H - self.kernel_size) // self.stride + 1
        W_out = (W - self.kernel_size) // self.stride + 1
        conv_out = np.zeros((N, self.out_channels, H_out, W_out))
        self.col_image = []

        weight_cols = self.W['val'].reshape(self.out_channels, -1)
        for i in range(N):
            img_i = x[i][np.newaxis, :]
            x_cols = im2col(img_i, self.kernel_size, self.stride)
            conv_out[i] = (np.dot(weight_cols, x_cols.T) + 
self.b['val']).reshape(self.out_channels, H_out, W_out)
            self.col_image.append(x_cols)
        self.col_image = np.array(self.col_image)

        return conv_out
    def backward(self, error):
        (N,C,_,_) = error.shape
        error_col = error.reshape(N,C,-1)
        for i in range(N):
            self.W['grad'] += np.dot(error_col[i],
self.col_image[i]).reshape(self.W['val'].shape)
        self.b['grad'] += np.sum(error_col, axis=
(0,2)).reshape(self.b['val'].shape)

        error_pad =np.pad(error, ((0,0), (0,0), (self.kernel_size - 1,
self.kernel_size - 1),
                             (self.kernel_size - 1, self.kernel_size - 1)),
'constant', constant_values=0)

        flip_weights = self.W['val'][:, :, ::-1, ::-1]
        flip_weights = flip_weights.swapaxes(0,1)
        col_flip_weights = flip_weights.reshape(self.in_channels, -1)

        col_pad_delta = np.array([im2col(error_pad[i][np.newaxis, :],
self.kernel_size, self.stride) for i in range(N)])
        next_delta = np.dot(col_pad_delta, col_flip_weights.T)
        next_delta = np.reshape(next_delta.transpose(0,2,1), self.input_shape)

        return next_delta

def im2col(image, kernel_size, stride):
    (N, C, H, W) = image.shape
    image_col = []
    for i in range(0, H - kernel_size + 1, stride):
        for j in range(0, W - kernel_size + 1, stride):
            col = image[:, :, i:i+kernel_size, j:j+kernel_size].reshape(-1)
            image_col.append(col)
    image_col = np.array(image_col)
    return image_col
```

```
im2col_conv = im2col_Conv(1,6,5)
%timeit im2col_conv.forward(image)
28.4 ms ± 544 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
%timeit im2col_conv.backward(out)
66.2 ms ± 1.04 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

可见利用im2col可让卷积层的速度提高6倍左右，而这只是im2col效率较低的一种实现，利用numpy的数组的`strides`属性配合`np.lib.stride_tricks.as_strided`可以更高效的实现im2col

```
def im2col_fast(image, kernel_size, stride):
    N, C, H, W = image.shape
    H_out = (H - kernel_size) // stride + 1
    W_out = (W - kernel_size) // stride + 1
    shape = (N, C, H_out, W_out, kernel_size, kernel_size)
    strides = (*image.strides[:-2], image.strides[-2]*stride,
image.strides[-1]*stride, *image.strides[-2:])
    A = np.lib.stride_tricks.as_strided(image, shape=shape, strides=strides)
    return A.transpose(0,2,3,1,4,5).reshape(N, H_out*W_out,
C*kernel_size*kernel_size)
```

fastim2col_Conv

```
class fastim2col_Conv():
    def __init__(self, in_channels, out_channels, kernel_size, stride=1):
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.weight_size = (self.out_channels, self.in_channels,
self.kernel_size, self.kernel_size)
        self.stride = stride

        self.W = {'val': np.random.standard_normal(self.weight_size), 'grad': 0}
        self.b = {'val':  np.random.standard_normal((self.out_channels,1)),
'grad': 0}

        self.cache = None

    def forward(self, x):
        (N,C,H,W) = x.shape
        self.input_shape = x.shape
        H_out = (H - self.kernel_size) // self.stride + 1
        W_out = (W - self.kernel_size) // self.stride + 1
        conv_out = np.zeros((N, self.out_channels, H_out, W_out))
        self.col_image = []

        weight_cols = self.W['val'].reshape(self.out_channels, -1)
        self.col_image = im2col_fast(x,self.kernel_size,self.stride)
        conv_out = (np.dot(self.col_image, weight_cols.T) +
self.b['val'].T).transpose(0,2,1).reshape(N, self.out_channels, H_out, W_out)

        return conv_out

    def backward(self, error):
        (N,C,_,_) = error.shape
        error_col = error.reshape(N,C,-1)
        for i in range(N):
            self.W['grad'] += np.dot(error_col[i],
self.col_image[i]).reshape(self.W['val'].shape)
        self.b['grad'] += np.sum(error_col, axis=
(0,2)).reshape(self.b['val'].shape)

        error_pad =np.pad(error, ((0,0), (0,0), (self.kernel_size - 1,
self.kernel_size - 1),
                          (self.kernel_size - 1, self.kernel_size - 1)),
'constant', constant_values=0)

        flip_weights = self.W['val'][:, :, ::-1, ::-1]
        flip_weights = flip_weights.swapaxes(0,1)
        col_flip_weights = flip_weights.reshape(self.in_channels, -1)

        col_pad_delta = im2col_fast(error_pad,self.kernel_size,self.stride)
        next_delta = np.dot(col_pad_delta, col_flip_weights.T)
        next_delta = np.reshape(next_delta.transpose(0,2,1), self.input_shape)

        return next_delta
```

```
fastim2col_conv = fastim2col_Conv(1,6,5)
%timeit fastim2col_conv.forward(image)
3.63 ms ± 108 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
%timeit fastim2col_conv.backward(out)
19.6 ms ± 767 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

这样的实现相比`naive_Conv`的速度提高了20倍左右，比原始的`im2col`快了4倍左右

Pooling

原代码使用的实现依旧是暴力4层循环，运行效率较低，同样是利用`im2col`进行加速

以下速度测试代码在`pool_speed_test.ipynb`文件中，输入`image`的`batch_size`为64，图像大小为28*28

naive_Maxpool

```
class naive_MaxPool():
    def __init__(self, F, stride):
        self.F = F
        self.S = stride
        self.cache = None

    def _forward(self, X):
        # X: (N, Cin, H, W): maxpool along 3rd, 4th dim
        (N,Cin,H,W) = X.shape
        F = self.F
        W_ = int(float(W)/F)
        H_ = int(float(H)/F)
        Y = np.zeros((N,Cin,W_,H_))
        M = np.zeros(X.shape) # mask
        for n in range(N):
            for cin in range(Cin):
                for w_ in range(W_):
                    for h_ in range(H_):
                        Y[n,cin,w_,h_] = np.max(X[n,cin,F*w_:F*(w_+1),F*h_:F*
(h_+1)])
                        i,j = np.unravel_index(X[n,cin,F*w_:F*(w_+1),F*h_:F*
(h_+1)].argmax(), (F,F))
                        M[n,cin,F*w_+i,F*h_+j] = 1
        self.cache = M
        return Y

    def _backward(self, dout):
        M = self.cache
        (N,Cin,H,W) = M.shape
        dout = np.array(dout)
        #print("dout.shape: %s, M.shape: %s" % (dout.shape, M.shape))
        dX = np.zeros(M.shape)
        for n in range(N):
            for c in range(Cin):
                #print("(n,c): (%s,%s)" % (n,c))
                dX[n,c,:,:] = dout[n,c,:,:].repeat(2, axis=0).repeat(2, axis=1)
        return dX*M
```

```
naive_maxpool = naive_MaxPool(2,2)
%timeit naive_maxpool._forward(image)
133 ms ± 620 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
out = naive_maxpool._forward(image)
%timeit naive_maxpool._backward(out)
556 µs ± 10.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

im2col_MaxPool

```
class im2col_MaxPool:
    def __init__(self, pool_h, pool_w, stride=1, pad=0):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.stride = stride
        self.pad = pad

        self.x = None
        self.arg_max = None

    def _forward(self, x):
        N, C, H, W = x.shape
        out_h = int(1 + (H - self.pool_h) / self.stride)
        out_w = int(1 + (W - self.pool_w) / self.stride)

        col = im2col(x, self.pool_h, self.pool_w, self.stride)
        col = col.reshape(-1, self.pool_h*self.pool_w)

        arg_max = np.argmax(col, axis=1)
        out = np.max(col, axis=1)

        out = out.reshape(N, out_h, out_w, C).transpose(0, 3, 1, 2)

        self.x = x
        self.arg_max = arg_max

        return out

    def _backward(self, dout):
        dout = dout.transpose(0, 2, 3, 1)

        pool_size = self.pool_h * self.pool_w
        dmax = np.zeros((dout.size, pool_size))
        dmax[np.arange(self.arg_max.size), self.arg_max.flatten()] =
dout.flatten()
        dmax = dmax.reshape(dout.shape + (pool_size,))

        dcol = dmax.reshape(dmax.shape[0] * dmax.shape[1] * dmax.shape[2], -1)
        dx = col2im(dcol, self.x.shape, self.pool_h, self.pool_w, self.stride,
self.pad)

        return dx

def im2col(input_data, filter_h, filter_w, stride=1):
    N, C, H, W = input_data.shape
    out_h = (H - filter_h)//stride + 1
    out_w = (W - filter_w)//stride + 1

    col = np.zeros((N, C, filter_h, filter_w, out_h, out_w))

    for y in range(filter_h):
        y_max = y + stride*out_h
        for x in range(filter_w):
            x_max = x + stride*out_w
            col[:, :, y, x, :, :] = input_data[:, :, y:y_max:stride,
x:x_max:stride]
    col = col.transpose(0, 4, 5, 1, 2, 3).reshape(N*out_h*out_w, -1)
    return col

def col2im(col, input_shape, filter_h, filter_w, stride=1, pad=0):
    N, C, H, W = input_shape
    out_h = (H + 2*pad - filter_h)//stride + 1
    out_w = (W + 2*pad - filter_w)//stride + 1
    col = col.reshape(N, out_h, out_w, C, filter_h, filter_w).transpose(0, 3, 4,
5, 1, 2)

    img = np.zeros((N, C, H + 2*pad + stride - 1, W + 2*pad + stride - 1))
    for y in range(filter_h):
        y_max = y + stride*out_h
        for x in range(filter_w):
            x_max = x + stride*out_w
            img[:, :, y:y_max:stride, x:x_max:stride] += col[:, :, y, x, :, :]

    return img[:, :, pad:H + pad, pad:W + pad]
```

```
im2col_maxpool = im2col_MaxPool(2,2,2)
%timeit im2col_maxpool._forward(image)
1.21 ms ± 16 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
%timeit im2col_maxpool._backward(out)
282 µs ± 5.81 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

使用`im2col`方法的速度相较原实现快了百倍以上

reshape_MaxPool

当池化区域为正方形时，还可以使用基于`reshape`和`broadcast`的池化方法

```
class reshape_MaxPool:
    def __init__(self, pool_h, pool_w, strides):
        self.pool_h = pool_h
        self.pool_w = pool_w
        self.strides = strides
        self.cache = None


    def _forward(self, x):
        N, C, H, W = x.shape
        assert self.pool_h == self.pool_w == self.strides, 'Invalid pool params'
        assert H % self.pool_h == 0
        assert W % self.pool_w == 0
        x_reshaped = x.reshape(N, C, H // self.pool_h, self.pool_h,
                                    W // self.pool_w, self.pool_w)
        out = x_reshaped.max(axis=3).max(axis=4)

        self.cache = (x, x_reshaped, out)
        return out

    def _backward(self, dout):
        dx_reshaped = np.zeros_like(self.cache[1])
        out_newaxis = self.cache[2][:, :, :, np.newaxis, :, np.newaxis]
        mask = (self.cache[1] == out_newaxis)
        dout_newaxis = dout[:, :, :, np.newaxis, :, np.newaxis]
        dout_broadcast, _ = np.broadcast_arrays(dout_newaxis, dx_reshaped)
        dx_reshaped = dout_broadcast * mask
        #The line blow is to ensure everyone get correct result
        #dx_reshaped /= np.sum(mask, axis=(3, 5), keepdims=True)
        dx = dx_reshaped.reshape(self.cache[0].shape)

        return dx
```

```
reshape_maxpool = reshape_MaxPool(2,2,2)
%timeit reshape_maxpool._forward(image)
108 µs ± 532 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
%timeit reshape_maxpool._backward(out)
753 µs ± 8.36 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

这种方法在有多个值和`max`一样大时梯度会被分给所有的元素，从而计算出错误的梯度，将注释的代码去掉可以解决这个问题，但会让效率下降40%，实际训练中对结果影响不大，所以不处理也行

Fully connected

没什么可以修改的地方

```
class FC():
    def __init__(self, D_in, D_out):
        self.cache = None
        self.W = {'val': np.random.normal(0.0, np.sqrt(2/D_in), (D_in,D_out)),
'grad': 0}
        self.b = {'val': np.random.randn(D_out), 'grad': 0}

    def forward(self, X):
        out = np.dot(X, self.W['val']) + self.b['val']
        self.cache = X
        return out

    def backward(self, dout):
        X = self.cache
        dX = np.dot(dout, self.W['val'].T).reshape(X.shape)
        self.W['grad'] = np.dot(X.reshape(X.shape[0], np.prod(X.shape[1:])).T,
dout)
        self.b['grad'] = np.sum(dout, axis=0)
        return dX
```

Loss function

原代码使用的是`CrossEntropyLoss`，也没有什么可以修改的

但代码里`softmax`没有做梯度稳定，虽然`softmax`层的代码有以下代码，通过减去最大值防止分子上溢出和分母下溢出，但是如果分子发生下溢出有可能导致softmax出来的结果是0，如果再去计算`NLLLoss`就相当与计算`log(0)`

```
maxes = np.amax(X, axis=1)
maxes = maxes.reshape(maxes.shape[0], 1)
Y = np.exp(X - maxes)
```

所以`NLLLoss`中额外加入了判断来处理`softmax`值为0的情况

```
for e in M:
    #print(e)
    if e == 0:
        loss += 500
    else:
        loss += -np.log(e)
```

一种解决方案如下

$$\log[f(x_i)] = \log\left(\frac{e^{x_i}}{e^{x_1}+e^{x_2}+\cdots e^{x_n}}\right) = \log\left(\frac{\frac{e^{x_i}}{e^M}}{\frac{e^{x_1}}{e^M}+\frac{e^{x_2}}{e^M}+\cdots\frac{e^{x_n}}{e^M}}\right) = \log\left(\frac{e^{(x_i-M)}}{\sum_j^n e^{(x_j-M)}}\right) = \log\left(e^{(x_i-M)}\right) - \log\left(\sum_j^n e^{(x_j-M)}\right) = (x_i-M) - \log\left(\sum_j^n e^{(x_j-M)}\right)$$

Dropout

训练时发现在`batch_size`为64，训练次数为25000的时候，模型有明显的过拟合现象（参数已保存为`过拟合.pkl`），此时训练集的准确度虽然高于90%，但测试集的准确率只有70%，因此为模型加入`Dropout`防止过拟合，定义如下

```
class Dropout():
    """
    Dropout layer
    """
    def __init__(self, p=1):
        self.cache = None
        self.p = p

    def forward(self, X):
        random_tensor = np.random.binomial(n=1, p=self.p, size=X.shape)
        self.cache = X, random_tensor
        return X*random_tensor

    def backward(self, dout):
        X, random_tensor = self.cache
        dX = dout*random_tensor/self.p
        return dX
```

Training

使用`SGD+Momentum`的方式训练30000次，`batch_size`为64，每10000步学习率下降10倍

```
model = LeNet5()
losses = []
```

```
optim = SGDMomentum(model.get_params(), lr=0.0001, momentum=0.80, reg=0.00003)
criterion = CrossEntropyLoss()
ITER = 30000
for i in range(ITER):
    # get batch, make onehot
    X_batch, Y_batch = get_batch(X_train, Y_train, batch_size)
    Y_batch = MakeOneHot(Y_batch, D_out)

    # forward, loss, backward, step
    X_batch = X_batch.reshape((batch_size, 1, 28, 28))
    Y_pred = model.forward(X_batch)
    loss, dout = criterion.get(Y_pred, Y_batch)
    model.backward(dout)
    optim.step()

    if i % 100 == 0:
        print("%s%% iter: %s, loss: %s" % (100*i/ITER,i, loss))
        losses.append(loss)

    if i == 10000:
        optim = SGDMomentum(model.get_params(), lr=0.00001, momentum=0.80,
reg=0.00003)

    if i == 20000:
        optim = SGDMomentum(model.get_params(), lr=0.000001, momentum=0.80,
reg=0.00003)
```

训练出来的模型在去掉`Dropout`后进行测试，得到的训练集准确度为98.1%，测试集为97.95%

```
TRAIN--> Correct: 58880 out of 60000, acc=0.9813333333333333
TEST--> Correct: 9795 out of 10000, acc=0.9795
```

参考

fast_layers.py

如何防止softmax函数上溢出和下溢出

Implement MATLAB's im2col 'sliding' in Python