认知科学与类脑计算

实验报告五

日期：2019/5/30

班级：16 人工智能

姓名：贾乘兴

学号：201600301304

一.需求分析：

Poggio 以及 Serre 等人于 2007 年提出了一个仿脑、基于特征组合的对象特征提取模型-HMAX 模型，该模型的理论基础是生物学中对视皮层神经细胞进行对象识别机制的研究。HMAX 模型通过 Gabor 滤波、求最大值操作以及交替进行模板匹配模拟了人眼视皮层中神经细胞进行对象识别处理过程。HMAX 模型在模式识别领域主要作用于识别对象的特征提取，所提取的特征称为 HMAX 特征。HMAX 模型是一个层次式的结构，分为 5 层：S1 层、C1 层、S2 层、C2 层、VTU 层。S1 层、C1 层、S2 层、C2 层分别对应视皮层中的简单细胞和复杂细胞，而 VTU 层对应于识别细胞。本次实验的目的是加深对 HMAX 模型的理解，能够使用 HMAX 模型解决简单问题。

二.概要设计：

利用下载好的 MNIST 数据集以及已有的 hmax 模型的参数，构建 HMAX 模型。使用 MNIST 数据集中的训练集训练 svm 和神经网络模型，使用测试集测试训练好的网络。

三.详细设计：

HMAX 模型是一个层次式的结构，分为 5 层：S1 层、C1 层、S2 层、C2 层、VTU 层。

S1 层。该层事实上是用 4 个方向及 16 个尺度的 Gabor 滤波器组对输入图像进行滤波，得到 64 个响应图。在尺度上，HMAX 算法分的相当细致，它将 16 个尺度分成 8 个子带，每个子带包含两个相邻尺度，以便在之后的 C1 层进行整合。Gabor 函数滤波如下所示：

$$F(x,y) = exp(-\frac{(x_0^2+\gamma^2 y_0^2)}{2\sigma^2})\times cos(\frac{2\pi}{\lambda}x_0)$$
$$x_0 = xcos\theta + ysin\theta$$
$$y_0 = -xsin\theta + ycos\theta$$

其中，σ是高斯函数的有效宽度，θ是方向，λ是波长。根据 V1 简单细胞来对参数进行调整和确定。S1 响应形成 8 个波段，每个波段有 4 个方向和 2 个级别。每个波段有两个滤波器，一共有 64 个不同的 S1 响应，有四个方向（$\theta = 0^{\circ}, 45^{\circ}, 90^{\circ}, 135^{\circ}$）。

C1 层。如前所述，64 个响应图依子带编号分成 8 组，每组包括 S1 层得到的 2 个相邻尺度及所属每个尺度上所有的 4 个方向响应。C1 层的操作是依组依方向进行的。具体做法是先将任一个滤波器响应图划分成 8*8 的格子，在每个格子中求响应的最大值。这样对每个响应图都能得到一张采样过的最大值图；然后，对组内的两个相邻尺度的最大值图的对应像素再求一次最大值，以最终得到具有不变性质的响应。值得注意的是，以上是 8x8 区域没有重叠的情况，在这种情况下，减采样倍数为 8；实际上经常采用 8x8 区域相互重叠一半的情况以进一步增加不变性，在这种情况下减采样倍数为 4，数据经过 C1 层之后，我们在每组中得到的是 4 个方向的不变响应。注意最大值不能跨方向选取。对

于物体识别，可以在 C1 层生成的不变响应中抽取每类特征，并加以保存作为训练结果。其具体做法为：随机找到一个子带，在 4 方向不变响应图中抽取一块 m*m*4 的立方体，可以称之为 patch。

S2 层。在该层中，再次使用滤波器对 C1 层的输出进行滤波，产生新一轮的响应图。只是这次滤波器变成了之前随机抽取的那些，而非第一层硬性规定的 Gabor 滤波器。滤波的方式也有所不同，不再是卷积运算，而是直接以 L2 距离作比较，再用高斯核映射成相似度。注意到随机抽取的 patch 是 n*n*4 的，因此做完滤波后，对每一个 patch 我们能得到 8 个组各一张图的响应。假设 patch 数目是 K，则共有 8*K 张响应作为 S2 的输出。

C2 层。对每一个 patch 遍历 8 个组的响应，找到最大的那个值。如此，对每张输入图片，最后得到一个 K 维向量。

VTU 层。该层可以使用 SVM 及其他方法，将 C2 层输出的 K 维向量放入该层进行训练。

整个模型可以看作进行初步的特征提取，得到的 k 维向量的基础上实现多分类

VTU 层分类方法：

1. SVM 多分类

   SVM 是二分类模型，可训练多个 SVM 作为多分类模型

2. 多层感知机

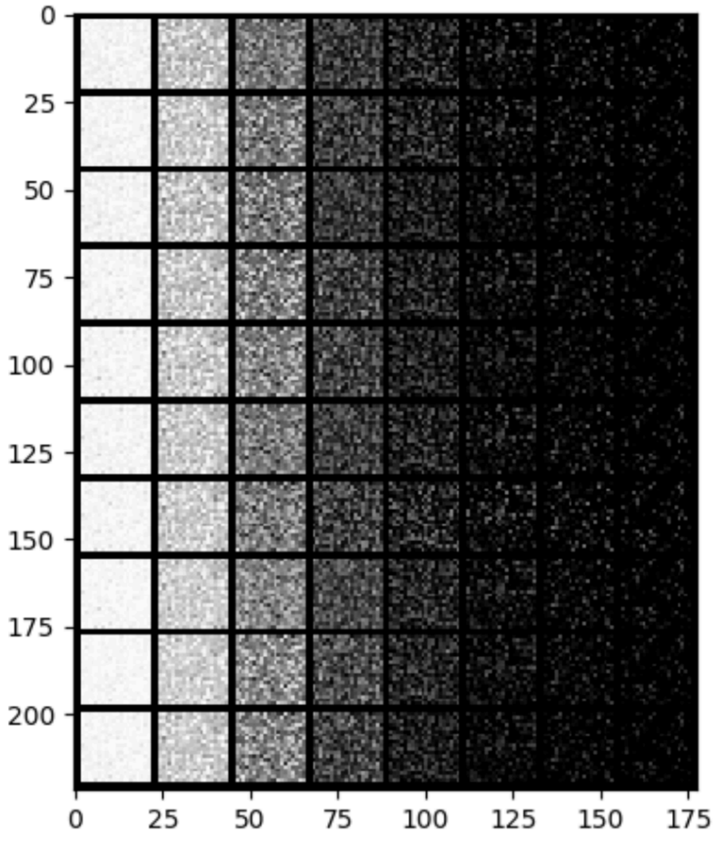
   训练多层全联接层

四.调试分析：

神经网络模型中，设计了一层隐藏层，结构较为简单，但对于 mnist 问题较为适合

svm 多分类模型中，由于数据过大，最终采用了随机采样的方法，读取了 5000 个样本进行训练，结果受样本采集影响较大

该分类中，我们将特征可视化，特征图肉眼不可分



从上到下分别为[3, 6, 4, 7, 9, 4, 6, 9, 7, 8]

五.测试结果：

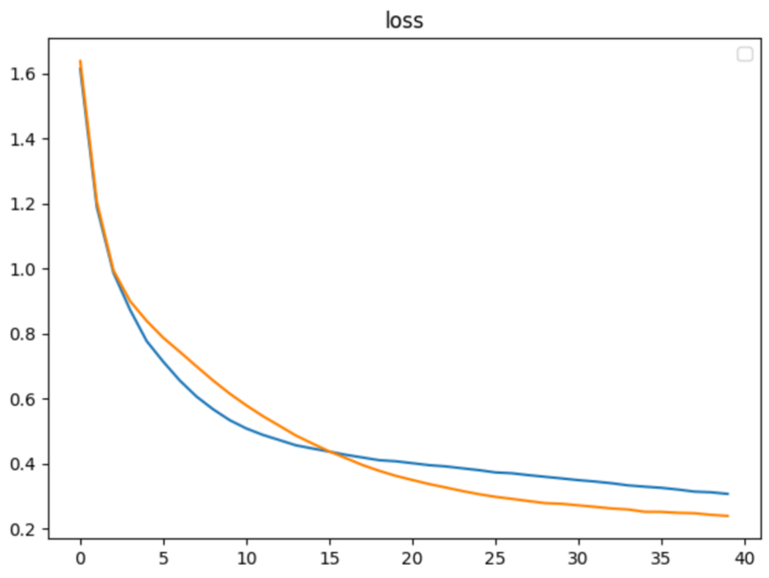
模型最后的层采用了两种方法，结果如下

svm 多分类模型：

最终正确率 67%

神经网络模型：

训练集与测试集 loss 变化



测试集准确率（epoch）：

0

accuracy on the whole test set 0.4787

10

accuracy on the whole test set 0.8094

20

accuracy on the whole test set 0.8835

30

accuracy on the whole test set 0.9072

39

accuracy on the whole test set 0.9171

最终可达到 91%

六.附录：

附录代码如下 文件为 hmax.py 以及 example.py、hmax_test.py、net.py

hmax.py

```
import numpy as np
from scipy.io import loadmat
import torch
from torch import nn


def gabor_filter(size, wavelength, orientation):
    """Create a single gabor filter.
    Parameters
    ----------
    size : int
        The size of the filter, measured in pixels. The filter is square,
hence
        only a single number (either width or height) needs to be specified.
    wavelength : float
        The wavelength of the grating in the filter, relative to the half the
        size of the filter. For example, a wavelength of 2 will generate a
        Gabor filter with a grating that contains exactly one wave. This
        determines the "tightness" of the filter.
    orientation : float
        The orientation of the grating in the filter, in degrees.
    Returns
    -------
    filt : ndarray, shape (size, size)
        The filter weights.
    """
    lambda_ = size * 2. / wavelength
    sigma = lambda_ * 0.8
    gamma = 0.3  # spatial aspect ratio: 0.23 < gamma < 0.92
    theta = np.deg2rad(orientation + 90)

    # Generate Gabor filter
    x, y = np.mgrid[:size, :size] - (size // 2)
    rotx = x * np.cos(theta) + y * np.sin(theta)
    roty = -x * np.sin(theta) + y * np.cos(theta)
    filt = np.exp(-(rotx**2 + gamma**2 * roty**2) / (2 * sigma ** 2))
    filt *= np.cos(2 * np.pi * rotx / lambda_)
    filt[np.sqrt(x**2 + y**2) > (size / 2)] = 0
    # Normalize the filter
    filt = filt - np.mean(filt)
    filt = filt / np.sqrt(np.sum(filt ** 2))

    return filt


class S1(nn.Module):
    """A layer of S1 units with different orientations but the same scale.
    The S1 units are at the bottom of the network. They are exposed to the
raw
    pixel data of the image. Each S1 unit is a Gabor filter, which detects
    edges in a certain orientation. They are implemented as PyTorch Conv2d
    modules, where each channel is loaded with a Gabor filter in a specific
    orientation.
    Parameters
    ----------
    size : int
        The size of the filters, measured in pixels. The filters are square,
        hence only a single number (either width or height) needs to be
        specified.
    wavelength : float
        The wavelength of the grating in the filter, relative to the half the
        size of the filter. For example, a wavelength of 2 will generate a
        Gabor filter with a grating that contains exactly one wave. This
        determines the "tightness" of the filter.
    orientations : list of float
        The orientations of the Gabor filters, in degrees.
    """
    def __init__(self, size, wavelength, orientations=[90, -45, 0, 45]):
        super().__init__()
        self.num_orientations = len(orientations)
        self.size = size

        # Use PyTorch's Conv2d as a base object. Each "channel" will be an
        # orientation.
        self.gabor = nn.Conv2d(1, self.num_orientations, size,
                               padding=size // 2, bias=False)

        # Fill the Conv2d filter weights with Gabor kernels: one for each
        # orientation
        for channel, orientation in enumerate(orientations):
            self.gabor.weight.data[channel, 0] = torch.Tensor(
                gabor_filter(size, wavelength, orientation))
```

```python
        # A convolution layer filled with ones. This is used to normalize the
        # result in the forward method.
        self.uniform = nn.Conv2d(1, 4, size, padding=size // 2, bias=False)
        nn.init.constant_(self.uniform.weight, 1)

        # Since everything is pre-computed, no gradient is required
        for p in self.parameters():
            p.requires_grad = False

    def forward(self, img):
        """Apply Gabor filters, take absolute value, and normalize."""
        s1_output = torch.abs(self.gabor(img))
        norm = torch.sqrt(self.uniform(img ** 2))
        norm.data[norm == 0] = 1  # To avoid divide by zero
        s1_output /= norm
        return s1_output


class C1(nn.Module):
    """A layer of C1 units with different orientations but the same scale.
    Each C1 unit pools over the S1 units that are assigned to it.
    Parameters
    ----------
    size : int
        Size of the MaxPool2d operation being performed by this C1 layer.
    """
    def __init__(self, size):
        super().__init__()
        self.size = size
        self.local_pool = nn.MaxPool2d(size, stride=size // 2,
                                       padding=size // 2)

    def forward(self, s1_outputs):
        """Max over scales, followed by a MaxPool2d operation."""
        s1_outputs = torch.cat([out.unsqueeze(0) for out in s1_outputs], 0)

        # Pool over all scales
        s1_output, _ = torch.max(s1_outputs, dim=0)

        # Pool over local (c1_space x c1_space) neighbourhood
        return self.local_pool(s1_output)
```

```python
class S2(nn.Module):
    """A layer of S2 units with different orientations but the same scale.
    The activation of these units is computed by taking the distance between
    the output of the C layer below and a set of predefined patches. This
    distance is computed as:
      d = sqrt( (w - p)^2 )
        = sqrt( w^2 - 2pw + p^2 )
    Parameters
    ----------
    patches : ndarray, shape (n_patches, n_orientations, size, size)
        The precomputed patches to lead into the weights of this layer.
    activation : 'gaussian' | 'euclidean'
        Which activation function to use for the units. In the PNAS paper, a
        gaussian curve is used ('guassian', the default), whereas the MATLAB
        implementation of The Laboratory for Computational Cognitive
        Neuroscience uses the euclidean distance ('euclidean').
    sigma : float
        The sharpness of the tuning (sigma in eqn 1 of [1]_). Defaults to 1.
    References:
    -----------
    .. [1] Serre, Thomas, Aude Oliva, and Tomaso Poggio. “A Feedforward
           Architecture Accounts for Rapid Categorization.” Proceedings of the
           National Academy of Sciences 104, no. 15 (April 10, 2007): 6424–29.
           https://doi.org/10.1073/pnas.0700622104.
    """
    def __init__(self, patches, activation='gaussian', sigma=1):
        super().__init__()
        self.activation = activation
        self.sigma = sigma

        num_patches, num_orientations, size, _ = patches.shape

        # Main convolution layer
        self.conv = nn.Conv2d(in_channels=num_orientations,
                              out_channels=num_orientations * num_patches,
                              kernel_size=size,
                              padding=size // 2,
                              groups=num_orientations,
                              bias=False)
        self.conv.weight.data = torch.Tensor(
            patches.transpose(1, 0, 2, 3).reshape(1600, 1, size, size))

        # A convolution layer filled with ones. This is used for the distance
        # computation
        self.uniform = nn.Conv2d(1, 1, size, padding=size // 2, bias=False)
        nn.init.constant_(self.uniform.weight, 1)

        # This is also used for the distance computation
        self.patches_sum_sq = nn.Parameter(
            torch.Tensor((patches ** 2).sum(axis=(1, 2, 3))))

        self.num_patches = num_patches
        self.num_orientations = num_orientations
        self.size = size

        # No gradient required for this layer
        for p in self.parameters():
            p.requires_grad = False

    def forward(self, c1_outputs):
        s2_outputs = []
        for c1_output in c1_outputs:
            conv_output = self.conv(c1_output)

            # Unstack the orientations
            conv_output_size = conv_output.shape[3]
            conv_output = conv_output.view(
                -1, self.num_orientations, self.num_patches, conv_output_size,
                conv_output_size)

            # Pool over orientations
            conv_output = conv_output.sum(dim=1)

            # Compute distance
            c1_sq = self.uniform(
                torch.sum(c1_output ** 2, dim=1, keepdim=True))
            dist = c1_sq - 2 * conv_output
            dist += self.patches_sum_sq[None, :, None, None]

            # Apply activation function
            if self.activation == 'gaussian':
                dist = torch.exp(- 1 / (2 * self.sigma ** 2) * dist)
            elif self.activation == 'euclidean':
                dist[dist < 0] = 0  # Negative values should never occur
                torch.sqrt_(dist)
                dist = -dist
            else:
                raise ValueError("activation parameter should be either "
                                 "'gaussian' or 'euclidean'.")

                s2_outputs.append(dist)
            return s2_outputs


class C2(nn.Module):
    """A layer of C2 units operating on a layer of S2 units."""
    def forward(self, s2_outputs):
        """Take the maximum value of the underlying S2 units."""
        maxs = [s2.max(dim=3)[0] for s2 in s2_outputs]
        maxs = [m.max(dim=2)[0] for m in maxs]
        maxs = torch.cat([m[:, None, :] for m in maxs], 1)
        return maxs.max(dim=1)[0]


class HMAX(nn.Module):
    """The full HMAX model.
    Use the `get_all_layers` method to obtain the activations for all layers.
    If you are only interested in the final output (=C2 layer), use the model
    as any other PyTorch module:
        model = HMAX(universal_patch_set)
        output = model(img)
    Parameters
    ----------
    universal_patch_set : str
        Filename of the .mat file containing the universal patch set.
    s2_act : 'gaussian' | 'euclidean'
        The activation function for the S2 units. Defaults to 'gaussian'.
    Returns
    -------
    c2_output : list of Tensors, shape (batch_size, num_patches)
        For each scale, the output of the C2 units.
    """
    def __init__(self, universal_patch_set, s2_act='gaussian'):
        super().__init__()

        # S1 layers, consisting of units with increasing size
        self.s1_units = [
            S1(size=7, wavelength=4),
            S1(size=9, wavelength=3.95),
            S1(size=11, wavelength=3.9),
            S1(size=13, wavelength=3.85),
            S1(size=15, wavelength=3.8),
```

```python
    S1(size=17, wavelength=3.75),
    S1(size=19, wavelength=3.7),
    S1(size=21, wavelength=3.65),
    S1(size=23, wavelength=3.6),
    S1(size=25, wavelength=3.55),
    S1(size=27, wavelength=3.5),
    S1(size=29, wavelength=3.45),
    S1(size=31, wavelength=3.4),
    S1(size=33, wavelength=3.35),
    S1(size=35, wavelength=3.3),
    S1(size=37, wavelength=3.25),
]

# Explicitly add the S1 units as submodules of the model
for s1 in self.s1_units:
    self.add_module('s1_%02d' % s1.size, s1)

# Each C1 layer pools across two S1 layers
self.c1_units = [
    C1(size=8),
    C1(size=10),
    C1(size=12),
    C1(size=14),
    C1(size=16),
    C1(size=18),
    C1(size=20),
    C1(size=22),
]

# Explicitly add the C1 units as submodules of the model
for c1 in self.c1_units:
    self.add_module('c1_%02d' % c1.size, c1)

# Read the universal patch set for the S2 layer
m = loadmat(universal_patch_set)
patches = [patch.reshape(shape[[2, 1, 0, 3]]).transpose(3, 0, 2, 1)
         for patch, shape in zip(m['patches'][0], m['patchSizes'].T)]

# One S2 layer for each patch scale, operating on all C1 layers
self.s2_units = [S2(patches=scale_patches, activation=s2_act)
              for scale_patches in patches]

# Explicitly add the S2 units as submodules of the model
for i, s2 in enumerate(self.s2_units):
```

```
            self.add_module('s2_%d' % i, s2)

        # One C2 layer operating on each scale
        self.c2_units = [C2() for s2 in self.s2_units]

        # Explicitly add the C2 units as submodules of the model
        for i, c2 in enumerate(self.c2_units):
            self.add_module('c2_%d' % i, c2)

    def run_all_layers(self, img):
        """Compute the activation for each layer.
        Parameters
        ----------
        img : Tensor, shape (batch_size, 1, height, width)
            A batch of images to run through the model
        Returns
        -------
        s1_outputs : List of Tensors, shape (batch_size, num_orientations,
height, width)
            For each scale, the output of the layer of S1 units.
        c1_outputs : List of Tensors, shape (batch_size, num_orientations,
height, width)
            For each scale, the output of the layer of C1 units.
        s2_outputs : List of lists of Tensors, shape (batch_size,
num_patches, height, width)
            For each C1 scale and each patch scale, the output of the layer of
            S2 units.
        c2_outputs : List of Tensors, shape (batch_size, num_patches)
            For each patch scale, the output of the layer of C2 units.
        """
        s1_outputs = [s1(img) for s1 in self.s1_units]

        # Each C1 layer pools across two S1 layers
        c1_outputs = []
        for c1, i in zip(self.c1_units, range(0, len(self.s1_units), 2)):
            c1_outputs.append(c1(s1_outputs[i:i+2]))

        s2_outputs = [s2(c1_outputs) for s2 in self.s2_units]
        c2_outputs = [c2(s2) for c2, s2 in zip(self.c2_units, s2_outputs)]

        return s1_outputs, c1_outputs, s2_outputs, c2_outputs

    def forward(self, img):
        """Run through everything and concatenate the output of the C2s."""
        c2_outputs = self.run_all_layers(img)[-1]
        c2_outputs = torch.cat(
            [c2_out[:, None, :] for c2_out in c2_outputs], 1)
        return c2_outputs

    def get_all_layers(self, img):
        """Get the activation for all layers as NumPy arrays.
        Parameters
        ----------
        img : Tensor, shape (batch_size, 1, height, width)
            A batch of images to run through the model
        Returns
        -------
        s1_outputs : List of arrays, shape (batch_size, num_orientations,
height, width)
            For each scale, the output of the layer of S1 units.
        c1_outputs : List of arrays, shape (batch_size, num_orientations,
height, width)
            For each scale, the output of the layer of C1 units.
        s2_outputs : List of lists of arrays, shape (batch_size, num_patches,
height, width)
            For each C1 scale and each patch scale, the output of the layer of
            S2 units.
        c2_outputs : List of arrays, shape (batch_size, num_patches)
            For each patch scale, the output of the layer of C2 units.
        """
        s1_out, c1_out, s2_out, c2_out = self.run_all_layers(img)
        return (
            [s1.cpu().detach().numpy() for s1 in s1_out],
            [c1.cpu().detach().numpy() for c1 in c1_out],
            [[s2_.cpu().detach().numpy() for s2_ in s2] for s2 in s2_out],
            [c2.cpu().detach().numpy() for c2 in c2_out],
        )
```

example.py

```
import torch
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import pickle

import hmax
from tqdm import tqdm

# load
```

```python
batch_size = 16
print('Data load')
train_dataset = datasets.MNIST(root='./data/', train=True,
transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root='./data/', train=False,
transform=transforms.ToTensor())

train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,
shuffle=False)

# Initialize the model with the universal patch set
print('Constructing model')
model = hmax.HMAX('./universal_patch_set.mat')


# Determine whether there is a compatible GPU available
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')

# Run the model on the example images
print('Running model on', device)
model = model.to(device)
c2x = None
c2y = None
for (X, y) in tqdm(train_loader):
    if c2x is None:
        c2x = model(X.to(device))
        c2y = y
    else:
        c2x = torch.cat([c2x, model(X.to(device))])
        c2y = torch.cat([c2y, y])


c2x_t = None
c2y_t = None
for (X, y) in tqdm(train_loader):
    if c2x_t is None:
        c2x_t = model(X.to(device))
        c2y_t = y
    else:
        c2x_t = torch.cat([c2x_t, model(X.to(device))])
        c2y_t = torch.cat([c2y_t, y])
```

```python
print('Saving output c2 to: c2.pt')
result = {'x':c2x, 'y':c2y, 'xt':c2x_t, 'yt':c2y_t}
torch.save(result, 'c2.pt')
print('done')
```

net.py

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.autograd import Variable

train_data = torch.load('mnist_feature_train')
test_data = torch.load('mnist_feature_test')

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3200, 1024)
        self.fc2 = nn.Linear(1024, 10)

    def forward(self, x):
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        return F.log_softmax(x)

model = Net()
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)

def train(epoch):
    print(epoch)
    for batch_idx, (data, target) in enumerate(train_data):
        output = model(data)
        loss = F.nll_loss(output, target)

        optimizer.zero_grad()   # 所有参数的梯度清零
        loss.backward()        #即反向传播求梯度
        optimizer.step()       #调用 optimizer 进行梯度下降更新参数

def test():
    test_loss = 0
    correct = 0
    for data, target in test_data:
        data, target = Variable(data, volatile=True), Variable(target)
        output = model(data)
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).cpu().sum()

    test_loss /= len(test_data)
    print('accuracy on test'.format(correct.numpy() / 10000))


for epoch in range(20):
    train(epoch)
    test()
```