BP 神经网络

1.MNIST 数据集的读取

代码：

```
from mnist import load_mnist
(x_train,t_train), (x_test,t_test)=load_mnist(normalize=True, flatten=True, one_hot_label=True)
print(x_train.shape, t_train.shape, x_test.shape, t_test.shape)
```

2.神经网络

2.1mini-batch 批量选取数据

代码：

```
from mnist import load_mnist
# 读取数据：
(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, flatten=True, one_hot_label=True)
epoch = 20000   # 对一批数据的迭代次数
for i in range(epoch):
    batch_mask = np.random.choice(train_size, batch_size)   # 从 0 到 60000 随机选 100 个数
    x_batch = x_train[batch_mask]   # 索引 x_train 中随机选出的行数，构成一批数据
    y_batch = net.predict(x_batch)   # 计算这批数据的预测值
    t_batch = t_train[batch_mask]   # 同 x_batch
```

2.2 前向传播

2.2.1 前向传播时，我们可以构造一个函数，输入数据，输出预测值

代码：

```
def predict(x,t):
        a1 = np.dot(x, w1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, w2) + b2
        y = softmax(a2)
```

2.2.2 需要用到激活函数得出各节点的输出值，因此涉及到 sigmoid，sigmoid 的导数和 softmax 函数

代码：

```
import numpy as np
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
def sigmoid_grad(x):
    return (1.0 - sigmoid(x)) * sigmoid(x)
```

```
def softmax(x):
    if x.ndim == 2:
        x = x.T
        x = x - np.max(x, axis=0)
        y = np.exp(x) / np.sum(np.exp(x), axis=0)
        return y.T
    x = x - np.max(x)
    return np.exp(x) / np.sum(np.exp(x))
```

2.2.3 选用交叉熵误差作为损失函数来衡算神经网络的精度

代码：

```
def loss(y, t):
    # 监督数据是 one-hot-vector 的情况下，转换为正确解标签的索引
    if t.size == y.size:
        t = t.argmax(axis=1)  # 找出一行中最大数值的索引号
    batch_size = y.shape[0]
    s = y[np.arange(batch_size), t]   # 找出 y 中对应于标签 t 中正确解位置的预测值
    return -np.sum(np.log(s + 1e-7)) / batch_size   # s+1e-7 防止取到无穷大
```

2.2.4 识别精度

代码：

```
    def accuracy(x,t):
        y = predict(x)   # y 为 100*10 的矩阵，因为前面选取了一批数据（包含 100 个数据）
        p = np.argmax(y, axis=1)  # 找出 y 中最大值的索引号，构成 1*100 的矩阵
        q = np.argmax(t, axis=1)  # 找出 t 中最大值的索引号，构成 1*100 的矩阵
        acc = np.sum(p == q) / len(y)  # 按布尔类型求和，在除以数据个数
        return acc
```

2.3 反向传播

2.3.1 构建神经网络

```
import numpy as np
from functions import sigmoid, sigmoid_grad, softmax, loss
class TwoLayerNet:
    def __init__(self, input_size, hidden_size, output_size, weight_init_std):
        # 初始化权重
        self.dict = {}   # 创建一个字典用于存储 w1, b1, w2, b2
        self.dict['w1'] = weight_init_std * np.random.randn(input_size, hidden_size)
```

```
        self.dict['b1'] = np.zeros(hidden_size)
        self.dict['w2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.dict['b2'] = np.zeros(output_size)
    def predict(self, x):
        w1, w2 = self.dict['w1'], self.dict['w2']
        b1, b2 = self.dict['b1'], self.dict['b2']
        a1 = np.dot(x, w1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, w2) + b2
        y = softmax(a2)
        return y
    def loss(y, t):
        if t.size == y.size:
            t = t.argmax(axis=1)
        batch_size = y.shape[0]
        return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
    def gradient(self, x, t):
        w1, w2 = self.dict['w1'], self.dict['w2']
        b1, b2 = self.dict['b1'], self.dict['b2']
        grads = {}
        a1 = np.dot(x, w1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, w2) + b2
        y = softmax(a2)
        num = x.shape[0]
        dy = (y - t) / num
        grads['w2'] = np.dot(z1.T, dy)
        grads['b2'] = np.sum(dy, axis=0)
        da1 = np.dot(dy, w2.T)
        dz1 = sigmoid_grad(a1) * da1
        grads['w1'] = np.dot(x.T, dz1)
        grads['b1'] = np.sum(dz1, axis=0)
        return grads
    def accuracy(self,x,t):
```

```
        y = self.predict(x)
        p = np.argmax(y, axis=1)
        q = np.argmax(t, axis=1)
        acc = np.sum(p == q) / len(y)
        return acc
```

3.训练神经网络

代码:

```
import numpy as np
import matplotlib.pyplot as plt
from TwoLayerNet import TwoLayerNet
from mnist import load_mnist

(x_train, t_train), (x_test, t_test) = load_mnist(normalize=True, one_hot_label=True)
net = TwoLayerNet(input_size=784, hidden_size=50, output_size=10, weight_init_std=0.01)

epoch = 20000
batch_size = 100
lr = 0.1

train_size = x_train.shape[0]   # 60000
iter_per_epoch = max(train_size / batch_size, 1)   # 600

train_loss_list = []
train_acc_list = []
test_acc_list = []

for i in range(epoch):
    batch_mask = np.random.choice(train_size, batch_size)   # 从 0 到 60000 随机选 100 个数
    x_batch = x_train[batch_mask]
    y_batch = net.predict(x_batch)
    t_batch = t_train[batch_mask]
    grad = net.gradient(x_batch, t_batch)

    for key in ('w1', 'b1', 'w2', 'b2'):
```

```
        net.dict[key] -= lr * grad[key]
    loss = net.loss(y_batch, t_batch)
    train_loss_list.append(loss)

    # 对每批数据记录一次精度和当前的损失值
    if i % iter_per_epoch == 0:
        train_acc = net.accuracy(x_train, t_train)
        test_acc = net.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print('第' + str(i + 1) + '次迭代"train_acc, test_acc, loss :| ' + str(train_acc) + ", " + str(test_acc)
+ ',' + str(loss))

# 绘制 精度 =f（迭代批数）的图像
markers = {'train': 'o', 'test': 's'}
x = np.arange(len(train_acc_list))
plt.plot(x, train_acc_list, label='train acc')
plt.plot(x, test_acc_list, label='test acc', linestyle='--')
plt.xlabel("epochs")
plt.ylabel("accuracy")
plt.ylim(0, 1.0)
plt.legend(loc='lower right')
plt.show()
```

CNN 卷积神经网络

1.数据集的读取，以及数据预定义

代码：

```
from tensorflow.examples.tutorials.mnist import input_data
# 读取 MNIST 数据集
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
# 预定义输入值 X、输出真实值 Y、placeholder 为占位符
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10]) # x、y_现在都是用占位符表示，当程序运行到一定指令，向 x、y_传入具体的值后，就可以代入进行计算了
# shape=[None, 784]是数据维度大小——因为 MNIST 数据集中每一张图片大小都是 28 * 28 28*2828* 28 的，计算时候是将 28 * 28 28*2828* 28 的二维数据转换成一个一维的、长度为 784 的新向量。None 表示其值大小不定，意即选中的 x、y_的数量暂时不定
keep_prob = tf.placeholder(tf.float32) # keep_prob 是改变参与计算的神经元个数的值
x_image = tf.reshape(x, [-1,28,28,1])
```

2.权重、偏置值函数

代码：

```
def weight_variable(shape):
    # 产生随机变量
    initial = tf.truncated_normal(shape, stddev=0.1) # truncated_normal()函数：选取位于正态分布均值=0.1 附近的随机值
    return tf.Variable(initial)


def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

3.卷积函数、池化函数定义

代码：

```
def conv2d(x, W):
    # stride = [1,水平移动步长,竖直移动步长,1]
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')


def max_pool_2x2(x):
    # stride = [1,水平移动步长,竖直移动步长,1]
```

return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1], padding='SAME') # 池化函数用简单传统的 2x2 大小的模板做 max pooling，池化步长为 2，选过的区域下次不再选取

4.第一次卷积+池化

由于 MNIST 数据集图片大小都是 28*28，且是黑白单色，所以准确的图片尺寸大小是 28*28*1(1 表示图片只有一个色层，彩色图片都 RGB3 个色层)，所以经过第一次卷积后，输出的通道数由 1 变成 32，图片尺寸变为:28*28*32(相当于拉伸了高)。再经过第一次池化，池化步长是 2*2，相当于每四个小格子池化成一个数值，所以经过池化后图片尺寸为 14*14*32

代码:

```
x_image = tf.reshape(x, [-1,28,28,1])
# 卷积层 1 网络结构定义
# 卷积核 1：patch=5×5;in size 1;out size 32;激活函数 reLU 非线性处理
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])
# output size 28*28*32
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
# output size 14*14*32
h_pool1 = max_pool_2x2(h_conv1)
```

5.第二次卷积+池化

第一次卷积+池化输出的图片大小是 14*14*32，经过第二次卷积后图片尺寸变为：14*14*64。再经过第二次池化(池化步长也是 2*2)，最后输出的图片尺寸为 7*7*64

代码:

```
#卷积层 2 网络结构定义
#卷积核 2：patch=5×5;in size 32;out size 64;激活函数 reLU 非线性处理
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
# output size 14*14*64
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
# output size 7 *7 *64
h_pool2 = max_pool_2x2(h_conv2)
```

6.全连接层 1、全连接层 2

全连接层的输入就是第二次池化后的输出，尺寸是 7*7*64，全连接层 1 设置有 1024 个神经元。

tf.reshape(a,newshape)函数，当 newshape = -1 时，函数会根据已有的维度计算出数组的

另外 shape 属性值。

keep_prob 是为了减小过拟合现象。每次只让部分神经元参与工作使权重得到调整。只有当 keep_prob = 1 时，才是所有的神经元都参与工作。全连接层 2 设置有 10 个神经元，相当于生成的分类器。

经过全连接层 1、2，得到的预测值存入 prediction 中

代码:

```
# 全连接层 1
W_fc1 = weight_variable([7*7*64,1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1,7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# 全连接层 2
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])
prediction = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

7.梯度下降法优化、求准确率

由于数据集太庞大，这里采用的优化器是 AdamOptimizer，学习率是 1e-4

tf.argmax(prediction,1)返回的是对于任一输入 x 预测到的标签值，tf.argmax(y_,1)代表正确的标签值

correct_prediction 这里是返回一个布尔数组。为了计算我们分类的准确率，我们将布尔值转换为浮点数来代表对与错，然后取平均值。例如：[True, False, True, True]变为[1,0,1,1]，计算出准确率就为 0.75

代码:

```
#二次代价函数:预测值与真实值的误差
loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=prediction))
#梯度下降法:数据太庞大,选用 AdamOptimizer 优化器
train_step = tf.train.AdamOptimizer(1e-4).minimize(loss)
#结果存放在一个布尔型列表中
correct_prediction = tf.equal(tf.argmax(prediction,1), tf.argmax(y_,1))
#求准确率
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

8.其他说明，保存参数

代码：

for i in range(1000):

batch = mnist.train.next_batch(50) # batch 是来源于 MNIST 数据集，一个批次包含 50 条数据

if i%100 == 0:

train_accuracy = accuracy.eval(feed_dict={x:batch[0], y_: batch[1], keep_prob: 1.0}) # feed_dict=({x: batch[0], y_: batch[1], keep_prob: 0.5}语句：是将 batch[0]，batch[1]代表的值传入 x，y_

print("step",i, "training accuracy",train_accuracy)

train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5}) # keep_prob = 0.5 只有一半的神经元参与工作

'''

#保存模型参数

saver.save(sess, './model.ckpt')

print("test accuracy %g"%accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))

'''

9.结果展示

训练 700 次时候，成功率已经到达 98%，越往后学习，准确率越高

```
Extracting MNIST_data\train-images-idx3-ubyte.gz
Extracting MNIST_data\train-labels-idx1-ubyte.gz
Extracting MNIST_data\t10k-images-idx3-ubyte.gz
Extracting MNIST_data\t10k-labels-idx1-ubyte.gz
step 0 training accuracy 0.06
step 100 training accuracy 0.82
step 200 training accuracy 0.92
step 300 training accuracy 0.98
step 400 training accuracy 0.94
step 500 training accuracy 0.96
step 600 training accuracy 0.94
step 700 training accuracy 0.98
```

全连接神经网络

构造了一个简单的全连接神经网络，输入为 28*28 的一维向量，隐藏层节点数为 256，经过一个 PReLU 激活函数，最后输出图片属于某一类别的可能性

1.定义含一个隐藏层的全连接网络

```
#全连接网络
from torch import nn,optim
class linear_model(nn.Module):
    def _init_(self):
        super(linear_model, self)._init_()
        self.dense=nn.Sequential(nn.Linear(28*28,256), nn.PReLU(),nn.Linear(256, 10)) # PReLU 效
果比 ReLU 好
    def forward(self, x):
        return self.dense(x)
```

2.模型的训练与测试（学习率为 0.1，优化器为 SGD）

```
#learning_rate=0.1,optim=SGD
from tqdm.notebook import tqdm
from torch.autograd import Variable

model = linear_model().cuda()
loss = nn.CrossEntropyLoss()
opt = optim.SGD(model.parameters(), lr=learning_rate)

for epoch in range(n_epochs):
    running_loss = []
    #data_train 是一个划分了 batch 的数据加载器，每一个 batch 看作一个训练集进行训练
    for x, y in tqdm(data_train):
        x = Variable(x).cuda()
        y = Variable(y).cuda()
        model.train() #启用 BatchNormalization 和 Dropout 进行训练模式
        y_prediction = model(x)
        l = loss(y_prediction, y)
        opt.zero_grad() #将梯度设为 0，因为每次训练把一个 batch 当作一个训练集进行训练
        l.backward() #反向传播计算梯度
        opt.step() #通过梯度下降进行一次参数更新，比如 w=w-a*(grad(w))
```

```
        running_loss.append(float(l))
    running_acc = []
    for x, y in data_test:
        x = Variable(x).cuda()
        y = Variable(y).cuda()
        model.eval() #不启用 BatchNormalization 和 Dropout 进行测试模式
        y_prediction = model(x)
        y_prediction = torch.argmax(y_prediction, dim=1) #dim=1 表示取每一行最大值的索引
        acc = float(torch.sum(y_prediction == y)) / batch_size_test
        running_acc.append(acc)

print("#summary:%4d%.4f%.2f%%"%(epoch,np.mean(running_loss),np.mean(running_acc)*100))
```

结果如下：

```
#summary:    0 0.4360 92.62%
100%  875/875 [00:01<00:00, 672.93it/s]
#summary:    1 0.2085 94.57%
100%  875/875 [00:01<00:00, 648.84it/s]
#summary:    2 0.1360 95.64%
100%  875/875 [00:01<00:00, 671.87it/s]
#summary:    3 0.0991 96.21%
100%  875/875 [00:01<00:00, 645.82it/s]
#summary:    4 0.0743 97.30%
100%  875/875 [00:01<00:00, 489.25it/s]
#summary:    5 0.0621 97.41%
100%  875/875 [00:01<00:00, 583.18it/s]
#summary:    6 0.0507 97.31%
100%  875/875 [00:01<00:00, 597.71it/s]
#summary:    7 0.0436 97.37%
100%  875/875 [00:01<00:00, 589.78it/s]
#summary:    8 0.0357 97.39%
100%  875/875 [00:01<00:00, 667.56it/s]
```

3.模型的训练与测试(学习率为 0.01，优化器为 SGD)

```
#learning_rate=0.01,optim=SGD
learning_rate = 0.01
model = linear_model().cuda()
loss = nn.CrossEntropyLoss()
opt = optim.SGD(model.parameters(), lr=learning_rate)

for epoch in range(n_epochs):
    running_loss = []
    #data_train 是一个划分了 batch 的数据加载器，每一个 batch 看作一个训练集进行训练
    for x, y in tqdm(data_train):
        x = Variable(x).cuda()
        y = Variable(y).cuda()
        model.train() #启用 BatchNormalization 和 Dropout 进行训练模式
        y_prediction = model(x)
        l = loss(y_prediction, y)
        opt.zero_grad() #将梯度设为 0，因为每次训练把一个 batch 当作一个训练集进行训练
        l.backward() #反向传播计算梯度
        opt.step() #通过梯度下降进行一次参数更新，比如 w=w-a*(grad(w))
        running_loss.append(float(l))
    running_acc = []
    for x, y in data_test:
        x = Variable(x).cuda()
        y = Variable(y).cuda()
        model.eval() #不启用 BatchNormalization 和 Dropout 进行测试模式
        y_prediction = model(x)
        y_prediction = torch.argmax(y_prediction, dim=1) #dim=1 表示取每一行最大值的索引
        acc = float(torch.sum(y_prediction == y)) / batch_size_test
        running_acc.append(acc)

print("#summary:%4d%.4f%.2f%%"%(epoch,np.mean(running_loss),np.mean(running_acc)*100))
```

结果如下：

```
#summary:   0 1.2101 85.68%

100%  875/875 [00:01<00:00, 679.46it/s]

#summary:   1 0.4887 88.49%

100%  875/875 [00:01<00:00, 603.05it/s]

#summary:   2 0.3911 89.74%

100%  875/875 [00:01<00:00, 659.60it/s]

#summary:   3 0.3510 90.55%

100%  875/875 [00:01<00:00, 657.23it/s]

#summary:   4 0.3259 91.11%

100%  875/875 [00:01<00:00, 608.06it/s]

#summary:   5 0.3066 91.48%

100%  875/875 [00:01<00:00, 543.37it/s]

#summary:   6 0.2897 92.01%

100%  875/875 [00:01<00:00, 529.12it/s]

#summary:   7 0.2743 92.32%

100%  875/875 [00:01<00:00, 550.37it/s]

#summary:   8 0.2583 92.61%

100%  875/875 [00:01<00:00, 622.86it/s]
```