示例代码 — 构建 TwoLayerNet 类，实现一个2层 Neural Netword 的学习

```
import matplotlib.pylab as plt
import numpy as np
import os
from mnist import load_mnist  # Load MNIST
from PIL import Image
```

函数

```
# Type 1. Basic Functions
```

基本函数

```
# Function - Numerical Differentiation
def num_diff(f, x):
    h = 1e-4
    return (f(x+h) - f(x-h)) / (2*h)
```

1. 数值微分

```
# Function - Numerical Gradient
def num_gradient(f, x):
    h = 1e-4 # 0.0001
    grad = np.zeros_like(x)

    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:
        idx = it.multi_index
        tmp_val = x[idx]
        x[idx] = float(tmp_val) + h
        fxh1 = f(x) # f(x+h)

        x[idx] = tmp_val - h
        fxh2 = f(x) # f(x-h)
        grad[idx] = (fxh1 - fxh2) / (2*h)

        x[idx] = tmp_val # 还原值
        it.iternext()

    return grad
```

2. 数值梯度

```
# Function - Gradient Descent
def gradient_descent(f, init_x, lr=0.01, step_num=100):
    x = init_x
    for i in range(step_num):
        grad = num_gradient(f, x)
        x -= lr * grad
    return x
```

3. 梯度下降

```
# Function - Sigmoid
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

4. Sigmoid函数

$\xrightarrow{x}$ Sigmoid $\xrightarrow{\frac{1}{1+e^{-x}}}$

```
# Function - ReLU
def relu(x):
    return np.maximum(0, x)
```

5. ReLU 函数 $f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases} = \max\{0, x\}$

```
# Function - Step Function
def step_function(x):
    return np.array(x > 0, dtype=np.int)
# Function - Sigmoid Gradient
def sigmoid_grad(x):
    return (1.0 - sigmoid(x)) * sigmoid(x)
```

6. Sigmoid Gradient

$f(x) = [1 - \mathrm{sig}(x)]\,\mathrm{sig}(x)$

```
# Type 2. Layers
```

```
# Function - Softmax Layer
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) # 溢出对策
    return np.exp(x) / np.sum(np.exp(x))
```

"层"函数

1. softmax层: $\xrightarrow{x_k}$ softmax $\xrightarrow{\frac{\exp(x_k)}{\Sigma \exp(x_i)}}$

```
# Function - Cross Entropy Error Layer
def cross_entropy_error(y, t):
    if y.ndim == 1:
        t = t.reshape(1, t.size)
        y = y.reshape(1, y.size)

    # 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)

    batch_size = y.shape[0]
    return -np.sum(np.log(y[np.arange(batch_size), t] + 1e-7)) / batch_size
```

2. Cross Entropy Error层:

$\xrightarrow[t_1, t_2, \cdots, t_n]{y_1, y_2, \cdots, y_n}$ Cross Entropy Error $\xrightarrow{-\Sigma t_i \log y_i}$

类

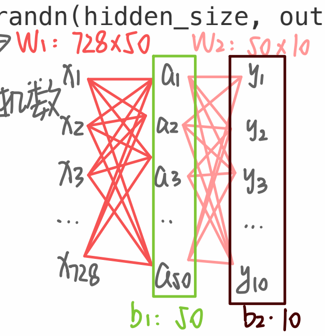
```
# Class - Two Layer Neural Network
class TwoLayerNet:

    def __init__(self, input_size, hidden_size, output_size, weight_init_std=0.01):
        # 初始化权重
        self.params = {}
        self.params['W1'] = weight_init_std * np.random.randn(input_size, hidden_size)
        self.params['b1'] = np.zeros(hidden_size)
        self.params['W2'] = weight_init_std * np.random.randn(hidden_size, output_size)
        self.params['b2'] = np.zeros(output_size)

    def predict(self, x):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']

        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)

        return y

    # x:输入数据, t:监督数据
    def loss(self, x, t):
        y = self.predict(x)

        return cross_entropy_error(y, t)

    def accuracy(self, x, t):
        y = self.predict(x)
        y = np.argmax(y, axis=1)
        t = np.argmax(t, axis=1)
```

一个2层的Neural Network类

构造函数

输入层规模 中间层规模 输出层规模 初始化权重参数

W1. W2为weight_int_std为参数的随机数



b1 b2初始化的0

预测函数

第一层(中间层): sig(a1)

→ x → a_1 → z_1 →

$a_1 = x W_1 + b_1$

第二层(输出层) softmax(a2)

→ z_1 → a_2 → y →

$a_2 = z_1 \cdot W_2 + b_2$

预测应该不用Softmax层?

cross_entropy_error

仅用于学习的cross_entropy_error层

统计准确率

正确个数　　　　数据规模

```
        accuracy = np.sum(y == t) / float(x.shape[0])
        return accuracy

    # x:输入数据, t:监督数据
    def numerical_gradient(self, x, t):
        loss_W = lambda W: self.loss(x, t)

        grads = {}
        grads['W1'] = num_gradient(loss_W, self.params['W1'])
        grads['b1'] = num_gradient(loss_W, self.params['b1'])
        grads['W2'] = num_gradient(loss_W, self.params['W2'])
        grads['b2'] = num_gradient(loss_W, self.params['b2'])

        return grads
```

数值梯度函数（耗时长）

```
    def gradient(self, x, t):
        W1, W2 = self.params['W1'], self.params['W2']
        b1, b2 = self.params['b1'], self.params['b2']
        grads = {}

        batch_num = x.shape[0]

        # forward
        a1 = np.dot(x, W1) + b1
        z1 = sigmoid(a1)
        a2 = np.dot(z1, W2) + b2
        y = softmax(a2)
```

误差反向传播法（快速）

正向: $x \to a_1 \to z_1 \to a_2 \to y$

```
        # backward
        dy = (y - t) / batch_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
```

反向: $\frac{\partial L}{\partial y} \to \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_2} \to \frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial b_1}$

返回综合梯度

Main

```
# Main
if __name__ == "__main__":

    # Get Data
    (x_train, t_train), (x_test, t_test) = load_mnist(normalize=True,
one_hot_label=True)

    # Init Network
    network = TwoLayerNet(input_size=784, hidden_size=50, output_size=10)

    # Set Variables
    iters_num = 10000  # 适当设定循环的次数
    train_size = x_train.shape[0]
    batch_size = 100
```

学习数据　　测试数据　　数据正规化: (0,255)→(0,1)

?

输入层: 784　　中间层: 50　　输出层: 10

实例化

← 循环10000次

训练样本60000个

每个batch 挑选100个样本

```
learning_rate = 0.1 ← 学习率: 0.1

train_loss_list = []
train_acc_list = []
test_acc_list = []                    Iterations per Epoch 每轮迭代次数
                                60000        100
每轮迭代次数: 600次
iter_per_epoch = max(train_size / batch_size, 1)

for i in range(iters_num):  10000次迭代
    batch_mask = np.random.choice(train_size, batch_size) 随机选 60000个中的100个
    x_batch = x_train[batch_mask]
    t_batch = t_train[batch_mask]

    # 计算梯度
    # grad = network.num_gradient(x_batch, t_batch)
    grad = network.gradient(x_batch, t_batch)  梯度

    # 更新参数
    for key in ('W1', 'b1', 'W2', 'b2'):
        network.params[key] -= learning_rate * grad[key]
                                  学习率0.1     梯度
    loss = network.loss(x_batch, t_batch)
    train_loss_list.append(loss)

    if i % iter_per_epoch == 0:  每进行 iter_per_epoch = 600次 (一个 epoch)
学习准确度   train_acc = network.accuracy(x_train, t_train)   共有 10000/600 ≈ 16次
测试准确度   test_acc = network.accuracy(x_test, t_test)
        train_acc_list.append(train_acc)
        test_acc_list.append(test_acc)
        print("train acc, test acc | " + str(train_acc) + ", " + str(test_acc))

# 绘制图形
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.savefig("output.png")
```