Layers,py

计算前向传播

def affine\_forward(x, w, b):

    """

    Computes the forward pass for an affine (fully-connected) layer.

    The input x has shape (N, d\_1, ..., d\_k) and contains a minibatch of N

    examples, where each example x[i] has shape (d\_1, ..., d\_k). We will

    reshape each input into a vector of dimension D = d\_1 \* ... \* d\_k, and

    then transform it to an output vector of dimension M.

    Inputs:

    - x: A numpy array containing input data, of shape (N, d\_1, ..., d\_k)

    - w: A numpy array of weights, of shape (D, M)

    - b: A numpy array of biases, of shape (M,)

    Returns a tuple of:

    - out: output, of shape (N, M)

    - cache: (x, w, b)

    """

本题中.

x: (2,4,5,6)

w: (120,3)

b: (3,)

    X=x.reshape(x.shape[0],-1)  #X:(2,120)

    out=X.dot(w)+b

计算反向传播

def affine\_backward(dout, cache):

    """

    Computes the backward pass for an affine layer.

    Inputs:

    - dout: Upstream derivative, of shape (N, M)

    - cache: Tuple of:

      - x: Input data, of shape (N, d\_1, ... d\_k)

      - w: Weights, of shape (D, M)

      - b: Biases, of shape (M,)

    Returns a tuple of:

    - dx: Gradient with respect to x, of shape (N, d1, ..., d\_k)

    - dw: Gradient with respect to w, of shape (D, M)

    - db: Gradient with respect to b, of shape (M,)

    """

    db=np.sum(dout,axis=0)  #db:(M,)

    X=x.reshape(x.shape[0],-1) #X:(N,D)  X.T(D,N)

    dw=X.T.dot(dout)

    dx1=dout.dot(w.T)

    dx=np.reshape(dx1,x.shape)

计算relu（前向）

def relu\_forward(x):

    """

    Computes the forward pass for a layer of rectified linear units (ReLUs).

    Input:

    - x: Inputs, of any shape

    Returns a tuple of:

    - out: Output, of the same shape as x

    - cache: x

    """

    out=np.maximum(0,x)

计算relu（反向）

def relu\_backward(dout, cache):

    """

    Computes the backward pass for a layer of rectified linear units (ReLUs).

    Input:

    - dout: Upstream derivatives, of any shape

    - cache: Input x, of same shape as dout

    Returns:

    - dx: Gradient with respect to x

    """

    dx=(x>0)\*dout

fc\_net.py

初始化权重矩阵w1,w2和偏差b1,b2

    def \_\_init\_\_(

        self,

        input\_dim=3 \* 32 \* 32,

        hidden\_dim=100,

        num\_classes=10,

        weight\_scale=1e-3,

        reg=0.0,

    ):

        """

        Initialize a new network.

        Inputs:

        - input\_dim: An integer giving the size of the input

        - hidden\_dim: An integer giving the size of the hidden layer

        - num\_classes: An integer giving the number of classes to classify

        - weight\_scale: Scalar giving the standard deviation for random

          initialization of the weights.

        - reg: Scalar giving L2 regularization strength.

        """

        # randn函数是基于零均值和标准差的一个高斯分布

        W1 = weight\_scale\*np.random.randn(input\_dim,hidden\_dim)#(3072，100)

        W2 = weight\_scale\*np.random.randn(hidden\_dim,num\_classes)#(100,10)

        b1 = np.zeros((hidden\_dim,))

        b2 = np.zeros((num\_classes,))

        self.params['W1'] = W1

        self.params['W2'] = W2

        self.params['b1'] = b1

        self.params['b2'] = b2

计算loss和grad

    def loss(self, X, y=None):

        """

        Compute loss and gradient for a minibatch of data.

        Inputs:

        - X: Array of input data of shape (N, d\_1, ..., d\_k)

        - y: Array of labels, of shape (N,). y[i] gives the label for X[i].

        Returns:

        If y is None, then run a test-time forward pass of the model and return:

        - scores: Array of shape (N, C) giving classification scores, where

          scores[i, c] is the classification score for X[i] and class c.

        If y is not None, then run a training-time forward and backward pass and

        return a tuple of:

        - loss: Scalar value giving the loss

        - grads: Dictionary with the same keys as self.params, mapping parameter

          names to gradients of the loss with respect to those parameters.

        """

计算scores

        # TODO: Implement the forward pass for the two-layer net, computing the    #

        # class scores for X and storing them in the scores variable.              #

        ############################################################################

        # \*\*\*\*\*START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)\*\*\*\*\*

第一层计算使用affine\_relu\_forward（包含relu），第二次计算使用affine\_forward（没有relu）

        W1=self.params['W1']

        W2=self.params['W2']

        b1=self.params['b1']

        b2=self.params['b2']

        out1,cache1=affine\_relu\_forward(X,W1,b1)

        scores,cache2=affine\_forward(out1,W2,b2)

计算loss和grad

############################################################################

        # TODO: Implement the backward pass for the two-layer net. Store the loss  #

        # in the loss variable and gradients in the grads dictionary. Compute data #

        # loss using softmax, and make sure that grads[k] holds the gradients for  #

        # self.params[k]. Don't forget to add L2 regularization!                   #

        #                                                                          #

        # NOTE: To ensure that your implementation matches ours and you pass the   #

        # automated tests, make sure that your L2 regularization includes a factor #

        # of 0.5 to simplify the expression for the gradient.

        #后向传播，计算loss和梯度

        loss,dscores=softmax\_loss(scores,y)#softmax\_loss在layers.py中

        loss+=0.5\*self.reg\*(np.sum(self.params['W1']\*\*2)+np.sum(self.params['W2']\*\*2))

        dout1,dw2,db2=affine\_backward(dscores,cache2)#第二层反向传播，affine\_backward函数在layers.py中

        dX,dw1,db1=affine\_relu\_backward(dout1,cache1)#第一层反向传播，affine\_relu\_backward函数在layer\_utils.py中

        dw1+=self.reg\*self.params['W1']

        dw2+=self.reg\*self.params['W2']

        grads['W1']=dw1

        grads['b1']=db1

        grads['W2']=dw2

        grads['b2']=db2

Solver solver.py

##############################################################################

# TODO: Use a Solver instance to train a TwoLayerNet that achieves at least  #

# 50% accuracy on the validation set.                                        #

##############################################################################

# \*\*\*\*\*START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)\*\*\*\*\*