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)\*\*\*\*\*

solver = Solver(model, data,

                update\_rule='sgd',

                optim\_config={

                  'learning\_rate': 1e-3,

                },

                lr\_decay=0.95,

                num\_epochs=10, batch\_size=100,

                print\_every=100)

solver.train()

全连接网络 FullyConnectedNet fc\_net.py

    """

    A fully-connected neural network with an arbitrary number of hidden layers,

    ReLU nonlinearities, and a softmax loss function. This will also implement

    dropout and batch/layer normalization as options. For a network with L layers,

    the architecture will be

    {affine - [batch/layer norm] - relu - [dropout]} x (L - 1) - affine - softmax

    where batch/layer normalization and dropout are optional, and the {...} block is

    repeated L - 1 times.

    Similar to the TwoLayerNet above, learnable parameters are stored in the

    self.params dictionary and will be learned using the Solver class.

    """

初始化网络参数

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

        # TODO: Initialize the parameters of the network, storing all values in    #

        # the self.params dictionary. Store weights and biases for the first layer #

        # in W1 and b1; for the second layer use W2 and b2, etc. Weights should be #

        # initialized from a normal distribution centered at 0 with standard       #

        # deviation equal to weight\_scale. Biases should be initialized to zero.   #

        #                                                                          #

        # When using batch normalization, store scale and shift parameters for the #

        # first layer in gamma1 and beta1; for the second layer use gamma2 and     #

        # beta2, etc. Scale parameters should be initialized to ones and shift     #

        # parameters should be initialized to zeros.

        #初始化所有隐藏层的参数

        layer\_input\_dim=input\_dim #D

        for i,h\_dim in enumerate(hidden\_dims): #(1,H1)(2,H2)

          self.params['W%d'%(i+1)]=weight\_scale\*np.random.randn(layer\_input\_dim,h\_dim)#W1(D，H1) W2(H1,H2)

          self.params['b%d'%(i+1)]=np.zeros((h\_dim,))#b1(H1,) b2(H2,)

          if self.normalization=='batchnorm' or self.normalization=='layernorm':

            self.params['gamma%d'%(i+1)]=np.ones((h\_dim,))#初始化为1

            self.params['beta%d'%(i+1)]=np.zeros((h\_dim,))#初始化为0

          layer\_input\_dim=h\_dim#将该层的列数传给下一层的行数

        #初始化所有输出层的参数

        self.params['W%d'%self.num\_layers]=weight\_scale\*np.random.randn(layer\_input\_dim,num\_classes)

        self.params['b%d'%self.num\_layers]=np.zeros((num\_classes,))

前向传播

        # TODO: Implement the forward pass for the fully-connected net, computing  #

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

        #                                                                          #

        # When using dropout, you'll need to pass self.dropout\_param to each       #

        # dropout forward pass.                                                    #

        #                                                                          #

        # When using batch normalization, you'll need to pass self.bn\_params[0] to #

        # the forward pass for the first batch normalization layer, pass           #

        # self.bn\_params[1] to the forward pass for the second batch normalization #

        # layer, etc.                                                              #

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

        fc\_mix\_cache={} #初始化每层前向传播的缓冲字典

        if self.use\_dropout:  #如果开启了dropout，初始化其对应的缓冲字典

          dp\_cache={}

        #从第一个隐藏层开始循环，每一个隐藏层，传递数据out,保存每一层的缓冲cache

        out=X

        for i in range(self.num\_layers-1):

          w=self.params['W%d'%(i+1)]

          b=self.params['b%d'%(i+1)]

          if self.normalization=='batchnorm':

            gamma=self.params['gamma%d'%(i+1)]

            beta=self.params['beta%d'%(i+1)]

            out,fc\_mix\_cache[i]=affine\_bn\_relu\_forward(out,w,b,gamma,beta,self.bn\_params[i])

          else:

            out,fx\_mix\_cache[i]=affine\_relu\_forward(out,w,b)

          if self.use\_dropout:

            out,dp\_cache[i]=drop\_forward(out,self.drop\_param)

        #最后的输出层

        w=self.params['W%d'%(self.num\_layers)]

        b=self.params['b%d'%(self.num\_layers)]

        out,out\_cache=affine\_forward(out,w,b)

        scores=out

反向传播计算loss和梯度

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

        # TODO: Implement the backward pass for the fully-connected 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!               #

        #                                                                          #

        # When using batch/layer normalization, you don't need to regularize the scale   #

        # and shift parameters.                                                    #

        #                                                                          #

        # 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.

        loss,dout=softmax\_loss(scores,y)

        loss+=0.5\*self.reg\*np.sum(self.params['W%d'%(self.num\_layers)])

        #输出层梯度的反向传播,并把梯度保存在梯度字典grad[]中

        dout,dw,db=affine\_backward(dout,out\_cache)

        grads['W%d'%(self.num\_layers)]=dw+self.reg\*self.params['W%d'%(self.num\_layers)]

        grads['b%d'%(self.num\_layers)]=db

        #在每一个隐藏层处梯度的反向传播，不仅更新了梯度字典grad[]，还迭代计算出了loss

        for i in range(self.num\_layers-1):

          ri=self.num\_layers-i-2#倒数第ri+1隐藏层

          loss+=0.5\*self.reg\*np.sum(self.params['W%d'%(ri+1)]\*\*2) #迭代地补上每层的正则项给loss

          if self.use\_dropout:

            dout=drop\_backward(dout,dp\_cache[ri])

          if self.normalization=='batchnorm':

            dout,dw,db,dgamma,dbeta=affine\_bn\_relu\_forward(dout,fc\_mix\_cache[ri])

            grads['gamma%d'%(ri+1)]=dgamma

            grads['beta%d'%(ri+1)]=dbeta

          else:

            dout,dw,db=affine\_relu\_backward(dout,fc\_mix\_cache[ri])

          grads['W%d' %(ri+1,)] = dw + self.reg \* self.params['W%d' %(ri+1,)]

          grads['b%d' %(ri+1,)] = db

优化方法 optim.py

动量随机梯度下降 sgd\_momentum

*# Momentum update*

v **=** mu **\*** v **-** learning\_rate **\*** dx *# integrate velocity*

x **+=** v *# integrate position*

def sgd\_momentum(w, dw, config=None):

    """

    Performs stochastic gradient descent with momentum.

    config format:

    - learning\_rate: Scalar learning rate.

    - momentum: Scalar between 0 and 1 giving the momentum value.

      Setting momentum = 0 reduces to sgd.

    - velocity: A numpy array of the same shape as w and dw used to store a

      moving average of the gradients.

    v=config['momentum']\*v-config['learning\_rate']\*dw

    w+=v

    next\_w=w

RMSProp(Root Mean Square Prop)

cache **=** decay\_rate **\*** cache **+** (1 **-** decay\_rate) **\*** dx**\*\***2

x **+=** **-** learning\_rate **\*** dx **/** (np.sqrt(cache) **+** eps)

def rmsprop(w, dw, config=None):

    """

    Uses the RMSProp update rule, which uses a moving average of squared

    gradient values to set adaptive per-parameter learning rates.

    config format:

    - learning\_rate: Scalar learning rate.

    - decay\_rate: Scalar between 0 and 1 giving the decay rate for the squared

      gradient cache.

    - epsilon: Small scalar used for smoothing to avoid dividing by zero.

    - cache: Moving average of second moments of gradients.

    """

    config['cache']=config['cache']\*config['decay\_rate']+(1-config['decay\_rate'])\*dw\*dw

    w+= -config['learning\_rate']\*dw/(np.sqrt(config['cache'])+config['epsilon'])

    next\_w=w

Adam

m **=** beta1**\***m **+** (1**-**beta1)**\***dx

v **=** beta2**\***v **+** (1**-**beta2)**\***(dx**\*\***2)

x **+=** **-** learning\_rate **\*** m **/** (np.sqrt(v) **+** eps)

def adam(w, dw, config=None):

    """

    Uses the Adam update rule, which incorporates moving averages of both the

    gradient and its square and a bias correction term.

    config format:

    - learning\_rate: Scalar learning rate.

    - beta1: Decay rate for moving average of first moment of gradient.

    - beta2: Decay rate for moving average of second moment of gradient.

    - epsilon: Small scalar used for smoothing to avoid dividing by zero.

    - m: Moving average of gradient.

    - v: Moving average of squared gradient.

    - t: Iteration number.

    """

    config['m']=config['beta1']\*config['m']+(1-config['beta1'])\*dw

    config['v']=config['beta2']\*config['v']+(1-config['beta2'])\*dw\*dw

    w+= -config['learning\_rate']\*config['m']/( np.sqrt(config['v']) + config['epsilon']  )

    next\_w=w

Train a model

hidden\_dims=[1024,1024,1024]

weight\_scale=1e-2

reg=1e-1

model=FullyConnectedNet(hidden\_dims=hidden\_dims,dropout=1,normalization=None,weight\_scale=weight\_scale,reg=reg)

learning\_rate=1e-4

solver = Solver(model, data,

                update\_rule='adam',

                optim\_config={

                  'learning\_rate': learning\_rate,

                },

                lr\_decay=0.95,

                num\_epochs=5, batch\_size=128,

                print\_every=100)

solver.train()