Input dimension: N

Hidden layer dimension: H

Classes: C

Network architecture:

input - fully connected layer - ReLU - fully connected layer - softmax

        W1: First layer weights; has shape (D, H)

        b1: First layer biases; has shape (H,)

        W2: Second layer weights; has shape (H, C)

        b2: Second layer biases; has shape (C,)

        Inputs:

        - input\_size: The dimension D of the input data.

        - hidden\_size: The number of neurons H in the hidden layer.

        - output\_size: The number of classes C.

Def loss

 def loss(self, X, y=None, reg=0.0):

        """

        Compute the loss and gradients for a two layer fully connected neural

        network.

        Inputs:

        - X: Input data of shape (N, D). Each X[i] is a training sample.

        - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is

          an integer in the range 0 <= y[i] < C. This parameter is optional; if it

          is not passed then we only return scores, and if it is passed then we

          instead return the loss and gradients.

        - reg: Regularization strength.

        Returns:

        If y is None, return a matrix scores of shape (N, C) where scores[i, c] is

        the score for class c on input X[i].

        If y is not None, instead return a tuple of:

        - loss: Loss (data loss and regularization loss) for this batch of training

          samples.

        - grads: Dictionary mapping parameter names to gradients of those parameters

          with respect to the loss function; has the same keys as self.params.

        """

在y不存在的时候计算前向传播的scores

        #两个部分，分别是线性部分：计算wx+b,然后非线性部分：ReLu

        z1=X.dot(W1)+b1

        h1=np.maximum(0,z1)

        scores=h1.dot(W2)+b2#第二层

计算损失函数

        scores=scores-np.max(scores,axis=1,keepdims=True) #NxC

        #计算概率

        exp\_scores=np.exp(scores) #NxC

        probilities=exp\_scores/np.sum(exp\_scores,axis=1,keepdims=True)  #NxC

        #计算每个样本的损失函数

        Loss\_i=-np.log(probilities[range(N),y])

        loss=np.sum(Loss\_i)/N

        loss=loss+reg\*np.sum(W1\*W1)+reg\*np.sum(W2\*W2)

计算梯度

        dscores=probilities #NxC

        dscores[range(N),y]-=1

        dscores/=N  #NxC

        # 求W2,b2的梯度

        db2=np.sum(dscores,axis=0)#（C,）

        dh1=dscores.dot(W2.T)   #NxH

        dW2=h1.T.dot(dscores)+2\*reg\*W2  #HxC

        # 激活函数relu的梯度

        dRelu=(h1>0)\*dh1

        # 求W1,b1的梯度

        dW1=X.T.dot(dRelu)+2\*reg\*W1#DxH

        db1=np.sum(dRelu,axis=0)#(H.)

        grads['b2'] = db2

        grads['W2'] = dW2

        grads['W1'] = dW1

        grads['b1'] = db1

    def train(self, X, y, X\_val, y\_val,

              learning\_rate=1e-3, learning\_rate\_decay=0.95,

              reg=5e-6, num\_iters=100,

              batch\_size=200, verbose=False):

        """

        Train this neural network using stochastic gradient descent.

        Inputs:

        - X: A numpy array of shape (N, D) giving training data.

        - y: A numpy array f shape (N,) giving training labels; y[i] = c means that

          X[i] has label c, where 0 <= c < C.

        - X\_val: A numpy array of shape (N\_val, D) giving validation data.

        - y\_val: A numpy array of shape (N\_val,) giving validation labels.

        - learning\_rate: Scalar giving learning rate for optimization.

        - learning\_rate\_decay: Scalar giving factor used to decay the learning rate

          after each epoch.

        - reg: Scalar giving regularization strength.

        - num\_iters: Number of steps to take when optimizing.

        - batch\_size: Number of training examples to use per step.

        - verbose: boolean; if true print progress during optimization.

        """

创建随机batch

            idx = np.random.choice(num\_train,batch\_size,replace=True)

            X\_batch = X[idx,:]

            y\_batch = y[idx]

梯度更新

            self.params['W2'] += -learning\_rate \* grads['W2']

            self.params['b2'] += -learning\_rate \* grads['b2']

            self.params['W1'] += -learning\_rate \* grads['W1']

            self.params['b1'] += -learning\_rate \* grads['b1']

    def predict(self, X):

        """

        Use the trained weights of this two-layer network to predict labels for

        data points. For each data point we predict scores for each of the C

        classes, and assign each data point to the class with the highest score.

        Inputs:

        - X: A numpy array of shape (N, D) giving N D-dimensional data points to

          classify.

        Returns:

        - y\_pred: A numpy array of shape (N,) giving predicted labels for each of

          the elements of X. For all i, y\_pred[i] = c means that X[i] is predicted

          to have class c, where 0 <= c < C.

        """

预测

        scores = self.loss(X)

        y\_pred = np.argmax(scores, axis=1)

调整超参数

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

# TODO: Tune hyperparameters using the validation set. Store your best trained  #

# model in best\_net.                                                            #

#                                                                               #

# To help debug your network, it may help to use visualizations similar to the  #

# ones we used above; these visualizations will have significant qualitative    #

# differences from the ones we saw above for the poorly tuned network.          #

#                                                                               #

# Tweaking hyperparameters by hand can be fun, but you might find it useful to  #

# write code to sweep through possible combinations of hyperparameters          #

# automatically like we did on the previous exercises.                          #

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

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

best\_val = -1

learning\_rates=[1e-3,1e-2]

regularization\_strengths=[0.25,0.5]

input\_size=32\*32\*3

hidden\_size=100

num\_classes=10

results={}

net = TwoLayerNet(input\_size, hidden\_size, num\_classes)

for learning\_rate in learning\_rates:

  for regularization\_strength in regularization\_strengths:

    net=TwoLayerNet(input\_size,hidden\_size,num\_classes)

    loss\_hist = net.train(X\_train, y\_train, X\_val, y\_val,

            num\_iters=1000, batch\_size=200,

            learning\_rate=learning\_rate, learning\_rate\_decay=0.95,

            reg=regularization\_strength, verbose=False)

    y\_train\_pred=net.predict(X\_train)

    training\_accuracy = np.mean(y\_train == y\_train\_pred)

    y\_val\_pred=net.predict(X\_val)

    validation\_accuracy=np.mean(y\_val\_pred==y\_val)

    results[(learning\_rate, regularization\_strength)] = (training\_accuracy, validation\_accuracy)

    if best\_val<validation\_accuracy:

      best\_val=validation\_accuracy

      best\_net=net