In [ ]:

```
from builtins import range
import numpy as np

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)
    """
    out = None
    ###########################################################################
    # TODO: Implement the affine forward pass. Store the result in out. You   #
    # will need to reshape the input into rows.                               #
    ###########################################################################
    row_n = x.shape[0] # N
    col_n = np.prod(x.shape[1:]) # D
    x_0 = x.reshape(row_n, col_n) # (N * D)
    out = x_0 @ w + b # (N * D) @ (D * M) + (M,) = (N * M)
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = (x, w, b)
    return out, cache


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,)
    """
    x, w, b = cache
    dx, dw, db = None, None, None
    ###########################################################################
    # TODO: Implement the affine backward pass.                               #
    ###########################################################################
    row = x.shape[0] # N
```

```
    col = np.prod(x.shape[1:]) # D
    x1 = x.reshape(row, col) # (N * D)

    dx = (dout @ w.T).reshape(x.shape) # (N * M) @ (M * D) = (N * D) => (N, d_1, ... d_
k)
    dw = x1.T @ dout # (D * N) @ (N * M) = (D * M)
    db = np.sum(dout, axis = 0) #dout.T @ np.ones(row) # (M * N) @ (N,) = (M,)
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx, dw, db


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 = None
    ###########################################################################
    # TODO: Implement the ReLU forward pass.                                  #
    ###########################################################################
    out = np.maximum(0, x)
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = x
    return out, cache


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 = None, cache
    ###########################################################################
    # TODO: Implement the ReLU backward pass.                                 #
    ###########################################################################

#     dx = dout.T @ (x > 0)
    dx = (x > 0) * dout
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx


def softmax_loss(x, y):
```

```
    """
    Computes the loss and gradient for softmax classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
      class for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """

    loss = 0.0
    dx = None
    ############################################################################
    # TODO: Implement the softmax loss                                         #
    ############################################################################
#     row = x.shape[0]
#     Sc_m = x - np.max(x,axis = 1,keepdims = True)
#     e_Sc_m = np.exp(Sc_m)
#     E = -Sc_m[np.arange(row), y] + np.log(np.sum(e_Sc_m[np.arange(row), y], axis=1, k
eepdims=True))
#     loss = np.sum(E)
#     dx = (e_Sc_m / np.sum(e_Sc_m, axis=1, keepdims=True))[np.arange(row), y]- 1


    e_Sc_m = np.exp(x - np.max(x, axis=1, keepdims=True))
    E = e_Sc_m / np.sum(e_Sc_m, axis=1, keepdims=True)
    row = x.shape[0]
    loss = -np.sum(np.log(E[np.arange(row), y]))

    dx = E.copy()
    dx[np.arange(row), y] = E[np.arange(row), y] - 1
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################
    return loss, dx
```

In [ ]:

```
from builtins import range
from builtins import object
import numpy as np

from layers import *


class FullyConnectedNet(object):
    """
    A fully-connected neural network with ReLU nonlinearity and
    softmax loss that uses a modular layer design. We assume an input dimension
    of D, a hidden dimension of [H, ...], and perform classification over C classes.

    The architecure should be like affine - relu - affine - softmax for a one
    hidden layer network, and affine - relu - affine - relu- affine - softmax for
    a two hidden layer network, etc.

    Note that this class does not implement gradient descent; instead, it
    will interact with a separate Solver object that is responsible for running
    optimization.

    The learnable parameters of the model are stored in the dictionary
    self.params that maps parameter names to numpy arrays.
    """

    def __init__(self, input_dim, hidden_dim=[10, 5], num_classes=10,
                 weight_scale=0.1):
        """
        Initialize a new network.

        Inputs:
        - input_dim: An integer giving the size of the input
        - hidden_dim: A list of integer giving the sizes of the hidden layers
        - num_classes: An integer giving the number of classes to classify
        - weight_scale: Scalar giving the standard deviation for random
          initialization of the weights.
        """
        self.params = {}
        self.hidden_dim = hidden_dim
        self.n_layers = 1 + len(hidden_dim)
        ############################################################################
        # TODO: Initialize the weights and biases of the net. Weights              #
        # should be initialized from a Gaussian centered at 0.0 with               #
        # standard deviation equal to weight_scale, and biases should be           #
        # initialized to zero. All weights and biases should be stored in the      #
        # dictionary self.params, with first layer weights                         #
        # and biases using the keys 'W1' and 'b1' and second layer                 #
        # weights and biases using the keys 'W2' and 'b2'.                         #
        ############################################################################
        ##### two layers
#         self.params['W1'] = np.random.normal(scale=weight_scale, size=(input_dim, hid
den_dim[0]))
#         self.params['W2'] = np.random.normal(scale=weight_scale, size=(hidden_dim[0],
num_classes))
#         self.params['b1'] = np.zeros(hidden_dim[0])
#         self.params['b2'] = np.zeros(num_classes)
        ######

        ##### multi-layers
```

```
        for i in range(self.n_layers):
            Wi = 'W' + str(i+1)
            bi = 'b' + str(i+1)

            # First hidden layer
            if i == 0:
                self.params[Wi] = np.random.normal(scale=weight_scale, size=(input_dim,
hidden_dim[i]))
                self.params[bi] = np.zeros(hidden_dim[i])
            # Output layer
            elif i == self.n_layers - 1:
                self.params[Wi] = np.random.normal(scale=weight_scale, size=(hidden_dim
[i-1], num_classes))
                self.params[bi] = np.zeros(num_classes)
            # Intermediate hidden layer
            else:
                self.params[Wi] = np.random.normal(scale=weight_scale, size=(hidden_dim
[i-1], hidden_dim[i]))
                self.params[bi] = np.zeros(hidden_dim[i])
         ######

        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################


    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.
        """
        scores = None
        caches = {}
        ############################################################################
        # TODO: Implement the forward pass for the net, computing the              #
        # class scores for X and storing them in the scores variable.              #
        ############################################################################

        ###### two layers
#         out1, cache1 = affine_forward(X, self.params['W1'], self.params['b1'])
#         out2, cache2 = relu_forward(out1)

#         out3, cache3 = affine_forward(out2, self.params['W2'], self.params['b2'])
#         scores = out3
        ######

        ###### multi-layers
```

```
        for i in range(self.n_layers-1):
            Wi = 'W' + str(i+1)
            bi = 'b' + str(i+1)

            # First hidden layer
            if i == 0:
                out = X

            # Intermediate layers
            out, caches['cache1 %d'%(i+1)] = affine_forward(out, self.params[Wi], self.
params[bi])
            out, caches['cache2 %d'%(i+1)] = relu_forward(out)


        # The last layer
        scores, caches['cache3'] = affine_forward(out, self.params['W'+str(self.n_layer
s)],self.params['b'+str(self.n_layers)])



        ######

        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

        # If y is None then we are in test mode so just return scores
        if y is None:
            return scores

        loss, grads = 0, {}
        ############################################################################
        # TODO: Implement the backward pass for the 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].                                                          #
        ############################################################################

        loss, dscores = softmax_loss(scores, y)

        ###### two layers
#         dx2, grads['W2'], grads['b2'] = affine_backward(dscores, cache3)

#         dx1 = relu_backward(dx2, cache2)
#         dx0, grads['W1'], grads['b1'] = affine_backward(dx1, cache1)

#         grads['W2'] -= 1e-3 * self.params['W2'] # 1e-3 is good
#         grads['W1'] -= 1e-3 * self.params['W1']
        ######

        ###### multi-layers

        for i in range(self.n_layers, 0, -1):
            if i == self.n_layers:
                # hidden layer
                dout, grads['W'+str(i)], grads['b'+str(i)] = affine_backward(dscores, c
aches['cache3'])
            else:
                dout = relu_backward(dout, caches['cache2 %d'%i])
                dout, grads['W'+str(i)], grads['b'+str(i)] = affine_backward(dout, cach
es['cache1 %d'%i])
```

```
            grads['W'+str(i)] -= 1e-3 * self.params['W'+str(i)]

        ######
        ###############################################################################
        #                               END OF YOUR CODE                              #
        ###############################################################################

        return loss, grads
```

tanh, sigmoid functions would have vanishing gradients problem.

Because when $x \to \infty$, or $x \to -\infty$, the gradient is small

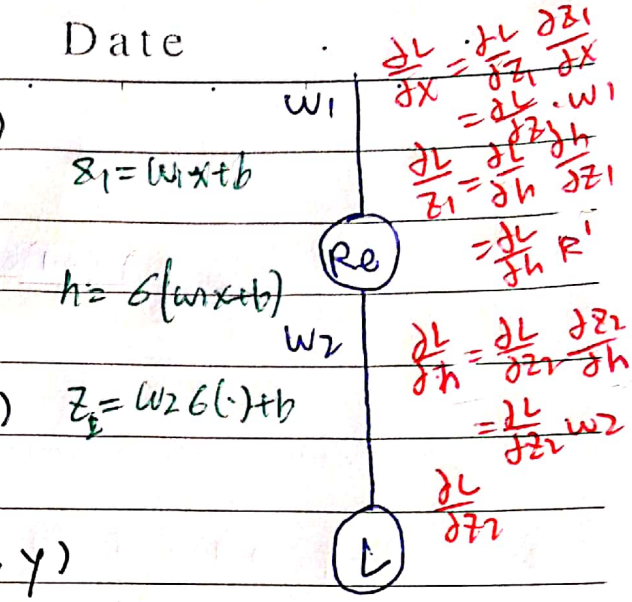
HW6

Di Zhen

August 2019

Kaggle Team Name: JoKer
'lr decay': 0.9
'num epochs': 10
'batch size': 32
'learning rate': 1e-5
hidden dim = [500,200]
Accuracy on Kaggle: 0.95140

Date



$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z_1}\frac{\partial z_1}{\partial x} = \frac{\partial L}{\partial z_1} \cdot w_1$

$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial h}\frac{\partial h}{\partial z_1} = \frac{\partial L}{\partial h} R'$

$\frac{\partial L}{\partial h} = \frac{\partial L}{\partial z_2}\frac{\partial z_2}{\partial h} = \frac{\partial L}{\partial z_2} w_2$

$\frac{\partial L}{\partial z_2}$

out, cache3 = affine_forward(x, w, b)

$z_2 = w_2 G(\cdot) + b$

loss, dscores = softmax_loss(scores, y)

$\frac{\partial L}{\partial z_2}$

dx, dw2, db = affine_backward(dout, cache3)

$\frac{\partial L}{\partial z_2} \times \frac{\partial z_2}{\partial h} = \frac{\partial L}{\partial z_2} \times w_2$

$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z_2} \times \frac{\partial z_2}{\partial b} = \frac{\partial L}{\partial z_2} \times 1$

dx = dout @ W.T, dw = x.T @ dout

$\frac{\partial L}{\partial h}$

out, cache2 = relu_forward(x)

$h = G(wx+b)$

$\frac{\partial L}{\partial z_1}$

dx = relu_backward(dout, cache2)

$\frac{\partial L}{\partial h} \times \frac{\partial h}{\partial z_1}$

dout * (x > 0)

out, cache1 = affine_forward(x, w, b)

$z_1 = w_1 x + b$

$\frac{\partial L}{\partial x}$

dx, dw1, db = affine_backward(dout, cache1)

$\frac{\partial L}{\partial z_1} \times \frac{\partial z_1}{\partial x} = \frac{\partial L}{\partial z_1} \times w_1$

dx = dout @ w.T

dw2 += dw2

dw1 += dw1

CS189 84.

HW 6

Date

2.a

when $k = t+1$

$(I * G)[t] = I(t+1)\,G[-1]$

when $k = t-1$

$(I * G)[t] = I(t-1)\,G[1]$

In order to make:

$$I * G[t] = \frac{(I[t+1] - I[t-1])}{2}$$

we have to design $G[1] = -\frac{1}{2}$, $G[-1] = \frac{1}{2}$ and for $t \neq 1$ or $t \neq -1$, $G[t] = 0$

(b)

```
for x in (0*s, 1*s, ..., W-w)
    for y in (0*s, 1*s, ..., H-h)
        total = 0
        for c in (r, g, b)
            fig = I[x:x+w, y:y+h]
            mask = fig * G
            total = total + sum(filter)
        output [x/s, y/s] = total
```

*: an operator for element-wise multiplication of two matrices

Sum( ): an operator for summing all elements in the matrix.

Date

(c).

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \\ \vdots & \vdots & \vdots \\ 1 & 0 & -1 \end{bmatrix}$$

vertical edge

(d) $(I * G) = R$

$$\frac{\partial L}{\partial G_c[x,y]} = \frac{\partial L}{\partial R} \frac{\partial R}{\partial G_c[x,y]}$$

$$= \frac{\partial L}{\partial R} \frac{\partial}{\partial G_c[x,y]} \sum_{a=1}^{w} \sum_{b=1}^{h} \sum_{c \in [r,g,b]} I_c[i+a, j+b] \cdot G_c[a,b]$$

$$= \sum_{i,j} \frac{\partial L}{\partial R[i,j]} I_c[i+x, j+y]$$

$$\frac{\partial L}{\partial I_c[x,y]} = \frac{\partial L}{\partial R} \frac{\partial R}{\partial I_c[x,y]}$$

$$= \frac{\partial L}{\partial R} \frac{\partial}{\partial I_c[x,y]} \sum_{a=1}^{w} \sum_{b=1}^{h} \sum_{c \in [r,g,b]} I_c[i+a, j+b] \cdot G_c[a,b]$$

$$= \sum_{i,j} \frac{\partial L}{\partial R[i,j]} G_c[x-i, y-j]$$

where $a = x - i$, $b = y - j$.

Date

(e)

max pooling $(I_c)[x, y]$

$$= \max_{a=[1..w]} \max_{b=[1..h]} I_c[x+a, y+b]$$

(f)

Use the derivative with respect to the output of maxpool layer to compute the derivative with respect to the input image of this layer.

Then use the derivative of input image of the layer as the derivative of output of previous layer

I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted.

Di Zhen