

```
import numpy as np
import pdb
```

```
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)

```
    """
```

```
    # ===== #
    # YOUR CODE HERE:
    #   Calculate the output of the forward pass. Notice the dimensions
    #   of w are D x M, which is the transpose of what we did in earlier
    #   assignments.
    # ===== #
    out = np.dot(x.reshape(x.shape[0],-1),w) + b
    # ===== #
    # END YOUR CODE HERE
    # ===== #
```

```
    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)

```
    Returns a tuple of:
```

- dx: Gradient with respect to x, of shape (N, d_1, ..., 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
```

```
    # ===== #
    # YOUR CODE HERE:
    #   Calculate the gradients for the backward pass.
    # ===== #
```

```
    # dout is N x M
    # dx should be N x d_1 x ... x d_k; it relates to dout through multiplication with w, which is
    D x M
    # dw should be D x M; it relates to dout through multiplication with x, which is N x D after
```

```

reshaping
    # db should be M; it is just the sum over dout examples

flattened_x = x.reshape(x.shape[0],-1)
dx = np.dot(dout,w.T).reshape(x.shape)
dw = np.dot(flattened_x.T,dout)
db = np.sum(dout, axis = 0)
# ===== #
# END YOUR CODE HERE
# ===== #

return dx, dw, db

def relu_forward(x):
    """
    Computes the forward pass for a layer of rectified linear units (ReLU).

    Input:
    - x: Inputs, of any shape

    Returns a tuple of:
    - out: Output, of the same shape as x
    - cache: x
    """
    # ===== #
    # YOUR CODE HERE:
    # Implement the ReLU forward pass.
    # ===== #
    relu = lambda x: x * (x > 0)
    out = relu(x)
    # ===== #
    # END YOUR CODE HERE
    # ===== #

    cache = x
    return out, cache

def relu_backward(dout, cache):
    """
    Computes the backward pass for a layer of rectified linear units (ReLU).

    Input:
    - dout: Upstream derivatives, of any shape
    - cache: Input x, of same shape as dout

    Returns:
    - dx: Gradient with respect to x
    """
    x = cache

    # ===== #
    # YOUR CODE HERE:
    # Implement the ReLU backward pass
    # ===== #

    # ReLU directs linearly to those > 0
    dx = dout * (x.reshape(x.shape[0],-1) > 0)

    # ===== #
    # END YOUR CODE HERE
    # ===== #

    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 j th class for the i th input.
- *y*: Vector of labels, of shape (N,) where $y[i]$ is the label for $x[i]$ and $0 \leq y[i] < C$

```
Returns a tuple of:
```

- *loss*: Scalar giving the loss
 - *dx*: Gradient of the loss with respect to *x*
- ```
"""
```

```
probs = np.exp(x - np.max(x, axis=1, keepdims=True))
probs /= np.sum(probs, axis=1, keepdims=True)
N = x.shape[0]
loss = -np.sum(np.log(probs[np.arange(N), y])) / N
dx = probs.copy()
dx[np.arange(N), y] -= 1
dx /= N
return loss, dx
```