Exercise 5: Solution

Non-linearities

Sigmoid – Forward

```python
def forward(self, x):
    """
    :param x: Inputs, of any shape.

    :return out: Outputs, of the same shape as x.
    :return cache: Cache, stored for backward computation, of the same shape as x.
    """
    shape = x.shape
    out, cache = np.zeros(shape), np.zeros(shape)
    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass of Sigmoid activation function            #
    ########################################################################
    out = 1 / (1 + np.exp(-x))
    cache = out
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return out, cache
```

Note:
The output of sigmoid function is stored in the cache for the computation in backward pass.

Sigmoid – Backward

```
def backward(self, dout, cache):
    """
    :return: dx: the gradient w.r.t. input X, of the same shape as X
    """
    dx = None
    ########################################################################
    # TODO:                                                                #
    # Implement the backward pass of Sigmoid activation function           #
    ########################################################################
    dx = dout * cache * (1 - cache)
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return dx
```

Remark:
The derivative of sigmoid function is is $sigmoid * (1 - sigmoid)$

Relu – Forward

```python
def forward(self, x):
    """
    :param x: Inputs, of any shape.

    :return outputs: Outputs, of the same shape as x.
    :return cache: Cache, stored for backward computation, of the same shape as x.
    """
    out = None
    cache = None
    ########################################################################
    # TODO:                                                                #
    # Implement the forward pass of Relu activation function               #
    ########################################################################
    out = np.maximum(x, 0)
    cache = x
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return out, cache
```

Relu – Backward

```python
def backward(self, dout, cache):
    """
    :return: dx: the gradient w.r.t. input X, of the same shape as X
    """
    dx = None
    ########################################################################
    # TODO:                                                                #
    # Implement the backward pass of Relu activation function              #
    ########################################################################
    x = cache
    dx = dout
    dx[x < 0] = 0
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return dx
```

Affine Layers

Affine Layer– Forward

```python
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:
    :param x: A numpy array containing input data, of shape (N, d_1, ..., d_k)
    :param w: A numpy array of weights, of shape (D, M)
    :param b: A numpy array of biases, of shape (M,)
    :return out: output, of shape (N, M)
    :return cache: (x, w, b)
    """
    N, M = x.shape[0], b.shape[0]
    out = np.zeros((N,M))
    ########################################################################
    # TODO: Implement the affine forward pass. Store the result in out.    #
    # You will need to reshape the input into rows.                        #
    ########################################################################
    x_reshaped = np.reshape(x, (x.shape[0], -1))
    out = x_reshaped.dot(w) + b
    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    cache = (x, w, b)
    return out, cache
```

Remark: the input x, weights w and bias b are saved in cache, such that the backward pass can access them.

Affine Layer – Backward

```python
def affine_backward(dout, cache):
    """
    Computes the backward pass for an affine layer.
    Inputs:
    :param dout: Upstream derivative, of shape (N, M)
    :param cache: Tuple of:
      - x: Input data, of shape (N, d_1, ... d_k)
      - w: Weights, of shape (D, M)
      - b: A numpy array of biases, of shape (M,
    :return dx: Gradient with respect to x, of shape (N, d1, ..., d_k)
    :return dw: Gradient with respect to w, of shape (D, M)
    :return 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.                            #
    ########################################################################

    dw = (np.reshape(x, (x.shape[0], -1)).T).dot(dout)
    dw = np.reshape(dw, w.shape)

    db = np.sum(dout, axis=0, keepdims=False)

    dx = dout.dot(w.T)
    dx = np.reshape(dx, x.shape)
    ########################################################################
    #                          END OF YOUR CODE                            #
    ########################################################################
    return dx, dw, db
```

Remark:
Make sure *dw* and *dx* have same shape as *w* and *x*.

As shown in lecture 5, the average of gradient needs to be calculated. However, here the average operation is already calculated in backward pass of loss function. Therefore, we don't average *n* in the calculation of *dw* and *db*.

Questions?
Campuswire ☺