

CS 4391 Introduction to Computer Vision

Homework 5 *

Professor Yu Xiang

April 17, 2025

Download the [homework5_programming.zip](#) file from eLearning, Assignments, Homework 5. Finish the following programming problems and submit your scripts to eLearning. You can zip all files for submission. Our TA will run your scripts to verify them.

Install the Python packages needed by

- `pip install -r requirement.txt`

Here are some useful resources:

- Python basics <https://pythonbasics.org/>
- Numpy <https://numpy.org/doc/stable/user/basics.html>
- OpenCV https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html

For this homework, you cannot use any deep learning libraries such as PyTorch or TensorFlow.

When using back-propagation to train neural networks, we compute local gradients of each layer and combine them to learn the weights in the neural networks. Fig. 1 illustrates this process. In this example, we have a layer that takes a matrix (tensor) x with dimension $D_x \times M_x$ and a matrix (tensor) y with dimension $D_y \times M_y$ as input. The output of this layer is z with dimension $D_z \times M_z$.

In the **forward function**, this layer computes the output z given x and y . It can also output a *cache* object that contains all the values needed during back-propagation.

In the **backward function**, this layer receives the *upstream gradients* and the *cache* object, and compute the *downstream gradients*. In this example, the upstream gradients is $\frac{\partial L}{\partial z}$, where L denotes the final loss function of the network. Note that the loss function outputs a scalar. Therefore, $\frac{\partial L}{\partial z}$ is with dimension $D_z \times M_z$ that is the same as z . We use chain rule to compute the downstream gradients

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}, \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y},$$

*This homework is adapted from Dr. Justin Johnson at the University of Michigan

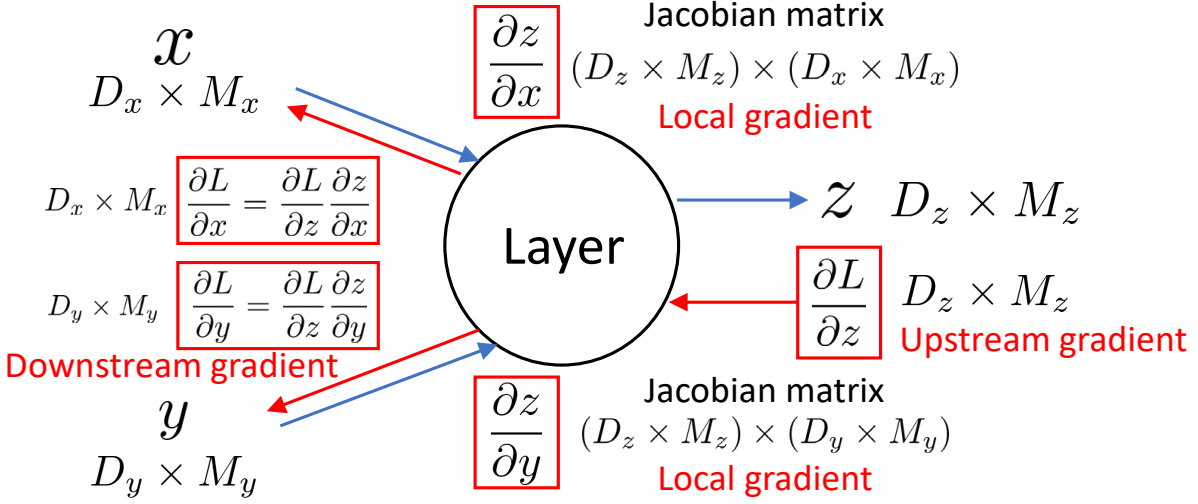


Figure 1: Back-propagation of gradients.

where $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ are the local gradients in this layer. They are the Jacobian matrices:

$$\left(\frac{\partial z}{\partial x}\right)_{ij} = \frac{\partial z_i}{\partial x_j}, \left(\frac{\partial z}{\partial y}\right)_{ij} = \frac{\partial z_i}{\partial y_j}.$$

We can consider $\frac{\partial z}{\partial x}$ as a matrix with dimension $(D_z \times M_z) \times (D_x \times M_x)$. Then we can do a matrix-vector multiplication to compute $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x}$, which have the same dimension as x . Similarly, we can compute $\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y}$.

In the following problems, you need to implement the fully-connected layer, the ReLU layer, the softmax loss function and the L2 regularization loss function in `neuralnet/layers.py`. After your implementation, you need to use the script `neuralnet/gradcheck_layers.py` to perform numeric gradient checking on your implementations. Given a function $f : \mathcal{R} \rightarrow \mathcal{R}$, we can approximate the gradient of f at a point $x_0 \in \mathcal{R}$ using central difference:

$$\frac{\partial f}{\partial x}(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h}. \quad (0.1)$$

The difference between all numeric and analytic gradients should be less than 10^{-9} . Keep in mind that numeric gradient checking does not check whether you have correctly implemented the forward pass. It only checks whether the backward pass you have implemented actually computes the gradient of the forward pass that you implemented.

Problem 1

(3 points) Back-propagation of fully connected layer.

Implement the `fc_forward()` function and the `fc_backward()` function in `neuralnet/layers.py` after reading the following derivation.

After your implementation, you need to use the script `neuralnet/gradcheck_layers.py` to perform numeric gradient checking on your implementations.

The input to a FC layer is a tensor x with shape (N, D_x) , where N is the batch size and D_x is the dimension of the feature. The weight matrix W in the FC layer is with shape (D_x, D_w) and the bias b of the FC layer is a vector with dimension D_w . The output of the FC layer is a tensor y with shape (N, D_w) . The i th row of the y matrix is computed by

$$y_i = x_i W + b, \quad (1.1)$$

where y_i and x_i are the i th row of y and x , respectively, i.e., the i th data point in the batch. With all the data points, we have

$$\begin{bmatrix} y_{1,1} & y_{1,2} & \cdots & y_{1,D_w} \\ y_{2,1} & y_{2,2} & \cdots & y_{2,D_w} \\ \vdots & \vdots & \vdots & \vdots \\ y_{N,1} & y_{N,2} & \cdots & y_{N,D_w} \end{bmatrix} = \begin{bmatrix} x_{1,1} & x_{1,2} & \cdots & x_{1,D_x} \\ x_{2,1} & x_{2,2} & \cdots & x_{2,D_x} \\ \vdots & \vdots & \vdots & \vdots \\ x_{N,1} & x_{N,2} & \cdots & x_{N,D_x} \end{bmatrix} \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,D_w} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,D_w} \\ \vdots & \vdots & \vdots & \vdots \\ w_{D_x,1} & w_{D_x,2} & \cdots & w_{D_x,D_w} \end{bmatrix} + \begin{bmatrix} b_1 & b_2 & \cdots & b_{D_w} \\ b_1 & b_2 & \cdots & b_{D_w} \\ \vdots & \vdots & \vdots & \vdots \\ b_1 & b_2 & \cdots & b_{D_w} \end{bmatrix}. \quad (1.2)$$

In the `backward function` of the FC layer, we receive upstream gradients $\frac{\partial L}{\partial y}$ with shape (N, D_w) .

We need to compute the downstream gradients $\frac{\partial L}{\partial x}$, $\frac{\partial L}{\partial w}$ and $\frac{\partial L}{\partial b}$. To do so, let's first consider

$$\frac{\partial L}{\partial x} = \sum_{i=1}^N \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} \cdot \frac{\partial y_{i,j}}{\partial x} \quad (1.3)$$

$$= \sum_{i=1}^N \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} \begin{bmatrix} 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ w_{1,j} & w_{2,j} & \cdots & w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & 0 \end{bmatrix} \text{ (the } i\text{th row)} \quad (1.4)$$

$$= \sum_{j=1}^{D_w} \begin{bmatrix} \frac{\partial L}{\partial y_{1,j}} w_{1,j} & \frac{\partial L}{\partial y_{1,j}} w_{2,j} & \cdots & \frac{\partial L}{\partial y_{1,j}} w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial L}{\partial y_{i,j}} w_{1,j} & \frac{\partial L}{\partial y_{i,j}} w_{2,j} & \cdots & \frac{\partial L}{\partial y_{i,j}} w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial L}{\partial y_{N,j}} w_{1,j} & \frac{\partial L}{\partial y_{N,j}} w_{2,j} & \cdots & \frac{\partial L}{\partial y_{N,j}} w_{D_x,j} \end{bmatrix} \quad (1.5)$$

$$= \begin{bmatrix} \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{1,j}} w_{1,j} & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{1,j}} w_{2,j} & \cdots & \frac{\partial L}{\partial y_{1,j}} w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} w_{1,j} & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} w_{2,j} & \cdots & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} w_{D_x,j} \\ \vdots & \vdots & \vdots & \vdots \\ \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{N,j}} w_{1,j} & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{N,j}} w_{2,j} & \cdots & \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{N,j}} w_{D_x,j} \end{bmatrix} \quad (1.6)$$

$$= \frac{\partial L}{\partial y} W^T. \quad (1.7)$$

We have a compact formula to compute the downstream gradients of x as

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} W^T. \quad (1.8)$$

Similarly, we have

$$\frac{\partial L}{\partial W} = \sum_{i=1}^N \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} \cdot \frac{\partial y_{i,j}}{\partial W} \quad (1.9)$$

$$= \sum_{i=1}^N \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} \underbrace{\begin{bmatrix} 0 & \cdots & x_{i,1} & \cdots & 0 \\ 0 & \cdots & x_{i,2} & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \cdots & x_{i,D_x} & \cdots & 0 \end{bmatrix}}_{\text{the } j\text{th column}} \quad (1.10)$$

$$= \sum_{i=1}^N \begin{bmatrix} \frac{\partial L}{\partial y_{i,1}} x_{i,1} & \cdots & \frac{\partial L}{\partial y_{i,j}} x_{i,1} & \cdots & \frac{\partial L}{\partial y_{i,D_w}} x_{i,1} \\ \frac{\partial L}{\partial y_{i,1}} x_{i,2} & \cdots & \frac{\partial L}{\partial y_{i,j}} x_{i,2} & \cdots & \frac{\partial L}{\partial y_{i,D_w}} x_{i,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \frac{\partial L}{\partial y_{i,1}} x_{i,D_x} & \cdots & \frac{\partial L}{\partial y_{i,j}} x_{i,D_x} & \cdots & \frac{\partial L}{\partial y_{i,D_w}} x_{i,D_x} \end{bmatrix} \quad (1.11)$$

$$= \begin{bmatrix} \sum_{i=1}^N \frac{\partial L}{\partial y_{i,1}} x_{i,1} & \cdots & \sum_{i=1}^N \frac{\partial L}{\partial y_{i,j}} x_{i,1} & \cdots & \sum_{i=1}^N \frac{\partial L}{\partial y_{i,D_w}} x_{i,1} \\ \sum_{i=1}^N \frac{\partial L}{\partial y_{i,1}} x_{i,2} & \cdots & \sum_{i=1}^N \frac{\partial L}{\partial y_{i,j}} x_{i,2} & \cdots & \sum_{i=1}^N \frac{\partial L}{\partial y_{i,D_w}} x_{i,2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \sum_{i=1}^N \frac{\partial L}{\partial y_{i,1}} x_{i,D_x} & \cdots & \sum_{i=1}^N \frac{\partial L}{\partial y_{i,j}} x_{i,D_x} & \cdots & \sum_{i=1}^N \frac{\partial L}{\partial y_{i,D_w}} x_{i,D_x} \end{bmatrix} \quad (1.12)$$

$$= \mathbf{x}^T \frac{\partial L}{\partial \mathbf{y}}. \quad (1.13)$$

Therefore,

$$\frac{\partial L}{\partial W} = \mathbf{x}^T \frac{\partial L}{\partial \mathbf{y}}. \quad (1.14)$$

Lastly, we compute

$$\frac{\partial L}{\partial b} = \sum_{i=1}^N \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} \cdot \frac{\partial y_{i,j}}{\partial b} \quad (1.15)$$

$$= \sum_{i=1}^N \sum_{j=1}^{D_w} \frac{\partial L}{\partial y_{i,j}} \underbrace{\begin{bmatrix} 0 & \cdots & 1 & \cdots & 0 \end{bmatrix}}_{\text{the } j\text{th column}} \quad (1.16)$$

$$= \sum_{i=1}^N \begin{bmatrix} \frac{\partial L}{\partial y_{i,1}} & \cdots & \frac{\partial L}{\partial y_{i,j}} & \cdots & \frac{\partial L}{\partial y_{i,D_w}} \end{bmatrix} \quad (1.17)$$

$$= \begin{bmatrix} \sum_{i=1}^N \frac{\partial L}{\partial y_{i,1}} & \cdots & \sum_{i=1}^N \frac{\partial L}{\partial y_{i,j}} & \cdots & \sum_{i=1}^N \frac{\partial L}{\partial y_{i,D_w}} \end{bmatrix} \quad (1.18)$$

$$= \mathbf{1}^T \frac{\partial L}{\partial \mathbf{y}}. \quad (1.19)$$

That is

$$\frac{\partial L}{\partial b} = \mathbf{1}^T \frac{\partial L}{\partial \mathbf{y}}, \quad (1.20)$$

where $\mathbf{1}$ denotes a column vector with all 1s.

Problem 2

(2 points) Back-propagation of ReLU layer.

Implement the `relu_forward()` function and the `relu_backward()` function in `neuralnet/layers.py`. The ReLU activation function is defined as

$$\text{ReLU}(x) = \max(0, x) \quad (2.1)$$

$$= \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise,} \end{cases} \quad (2.2)$$

for each element in a tensor.

After your implementation, you need to use the script `neuralnet/gradcheck_layers.py` to perform numeric gradient checking on your implementations.

Problem 3

(3 points) Back-propagation of Softmax Loss Function.

Implement the `softmax_loss()` function in `neuralnet/layers.py` after reading the following material.

After your implementation, you need to use the script `neuralnet/gradcheck_layers.py` to perform numeric gradient checking on your implementations.

The input to a softmax loss function layer is a tensor x with shape (N, C) , where N is the batch size and C is the number of categories to be classified. The softmax loss function first converts the scores x into a set of N probability distributions over the categories, defined as:

$$p_{i,c} = \frac{\exp(x_{i,c})}{\sum_{j=1}^C \exp(x_{i,j})}, i = 1, 2, \dots, N, c = 1, 2, \dots, C. \quad (3.1)$$

Then the softmax loss function is defined as

$$L = -\frac{1}{N} \sum_{i=1}^N \log(p_{i,y_i}), \quad (3.2)$$

where $y_i \in \{1, 2, \dots, C\}$ is the ground truth label for the i th data point.

A naive implementation of the softmax loss function can result in numeric instability when the value of some $x_{i,c}$ in Eq. (3.1) is large. Then it can cause overflow with $\exp(x_{i,c})$. To avoid this, we can compute the probabilities by

$$p_{i,c} = \frac{\exp(z_{i,c})}{\sum_{j=1}^C \exp(z_{i,j})} = \frac{\exp(x_{i,c} - M_i)}{\sum_{j=1}^C \exp(x_{i,j} - M_i)}, i = 1, 2, \dots, N, c = 1, 2, \dots, C, \quad (3.3)$$

where $M_i = \max_c x_{i,c}$, i.e., the maximum score for data point i among the categories, and $z_{i,c} = x_{i,c} - M_i$. By doing so, we can avoid overflow with the exponential. It is not hard to see that

$$p_{i,c} = \frac{\exp(x_{i,c} - M_i)}{\sum_{j=1}^C \exp(x_{i,j} - M_i)} = \frac{\exp(x_{i,c}) \exp(-M_i)}{\sum_{j=1}^C \exp(x_{i,j}) \exp(-M_i)} = \frac{\exp(x_{i,c})}{\sum_{j=1}^C \exp(x_{i,j})}. \quad (3.4)$$

Your softmax implementation should use this max-subtraction trick for numeric stability. You can run the script `neuralnet/check_softmax_stability.py` to check the numeric stability of your softmax loss implementation.

In the **backward function** of the softmax loss function, we need to compute the downstream gradients $\frac{\partial L}{\partial x}$ with shape (N, C) . First, we compute the gradients of L with respect to p_{i,y_i} in Eq. (3.2) as

$$\frac{\partial L}{\partial p_{i,y_i}} = -\frac{1}{N p_{i,y_i}}. \quad (3.5)$$

Note that

$$\frac{\partial L}{\partial p_{i,c}} = 0, \forall c \neq y_i. \quad (3.6)$$

Next, we compute

$$\begin{aligned}
\frac{\partial L}{\partial z_{i,c}} &= \sum_{i'=1}^N \sum_{c'=1}^C \frac{\partial L}{\partial p_{i',c'}} \cdot \frac{\partial p_{i',c'}}{\partial z_{i,c}} \\
&= \sum_{c'=1}^C \frac{\partial L}{\partial p_{i,c'}} \cdot \frac{\partial p_{i,c'}}{\partial z_{i,c}} \\
&= \frac{\partial L}{\partial p_{i,y_i}} \cdot \frac{\partial p_{i,y_i}}{\partial z_{i,c}}.
\end{aligned} \tag{3.7}$$

From the lecture, we know that

$$\frac{\partial p_{i,y_i}}{\partial z_{i,c}} = p_{i,y_i}(\delta_{y_i,c} - p_{i,c}), \tag{3.8}$$

where

$$\delta_{y_i,c} = \begin{cases} 1, & \text{if } c = y_i \\ 0, & \text{otherwise.} \end{cases} \tag{3.9}$$

By substituting Eq. (3.5) and Eq. (3.8) into Eq. (3.7), we have

$$\frac{\partial L}{\partial z_{i,c}} = -\frac{1}{N p_{i,y_i}} \cdot p_{i,y_i}(\delta_{y_i,c} - p_{i,c}) \tag{3.10}$$

$$= \frac{p_{i,c} - \delta_{y_i,c}}{N} \tag{3.11}$$

Lastly, we have $z_{i,c} = x_{i,c} - M_i$. It can be shown that this max-subtraction does not change the downstream gradients. Therefore, we have

$$\frac{\partial L}{\partial x_{i,c}} = \frac{\partial L}{\partial z_{i,c}} = \frac{p_{i,c} - \delta_{y_i,c}}{N}. \tag{3.12}$$

Problem 4

(2 points) Back-propagation of L2 regularization.

Implement the `l2_regularization()` function in `neuralnet/layers.py` after reading the following material.

After your implementation, you need to use the script `neuralnet/gradcheck_layers.py` to perform numeric gradient checking on your implementations.

L2 regularization implements the L2 regularization loss on the parameters in the network:

$$L(W) = \frac{\lambda}{2} \|W\|^2 = \frac{\lambda}{2} \sum_i W_i^2, \quad (4.1)$$

where the sum ranges over all scalar elements of the weight matrix W and λ is a hyperparameter controlling the regularization strength. The downstream gradients of this loss function is

$$\frac{\partial L}{\partial W_i} = \lambda W_i, \quad (4.2)$$

for each element in W .