HW3: CNN Implementation

王昊文

2022 年 1 月 11 日

1 Overview

In this homework, I will be implementing a simple CNN from scartch by using only Numpy, the model will be evaluated on a simple 3-class fruit image dataset.

The whole homework strictly follows the object-orientated philosophy, which treats *datatset*, *data loader*, *trainer*, *logger* as different objects, so that it can be very convinient to write your own custom objects. You can combine all your settings and put them under `config.json` inside the root folder.

Also, you will find that the structure is highly similar to pytorch template, which allows me to have a basic building block to build this project.

2 Details

2.1 Data

In the fruit dataset, we have a total of 1471 of training data, and 498 testing data. Each image in the dataset has a height and width of 32, and each image contains four channels, the first three represents RGB, and the last channel is the alpha value, which represents the transparency of each color. However, the alpha value will not be used in the CNN, so will be dropped will doing preprocessing.

There are 3 types of the fruits, they are: Carambola, Lychee, Pear.

2.2 Model

2.2.1 Intuition and loss function selection

This is a simple task, since there are only three classes, so I will be implementing a light, shallow CNN network for this task. The model acheives great results as we will discuss it later in the report, despite its shallowness.

For the loss function, it is obvious that Softmax with cross-entropy loss is the best choice for this task.

2.2.2 Architecture

Note that is defined under `model.py`. The total trainable parameters is 1,050,227. More details are listed in table 1. Note that all convolution

Layer	Output Shape	Param Num.
Conv2d-1	[16, 8, 32, 32]	224
ReLU-2	[16, 8, 32, 32]	0
Conv2d-3	[16, 16, 32, 32]	1,168
ReLU-4	[16, 16, 32, 32]	0
Linear-5	[16, 64]	1,048,640
ReLU-6	[16, 64]	0
Linear-7	[16, 3]	195

Table 1: A list of the layers of the current model

have a kernel size of 3, stride 1 and 0 padding.

2.3 Training and validation

As shown in figure 1, our loss curve converged really quickly in just a few epochs. As for the accuracy shown in figure 2, The model started out at around 60% and quickly reached 100% near 10 epochs. When training, I set the batch size to 32. During training, I used SGD as the optimizer, with learning rate of 10^{-3}. Also, the learning rate is scheduled to decay at a rate of 10^{-6}.
The above training details shows great results.

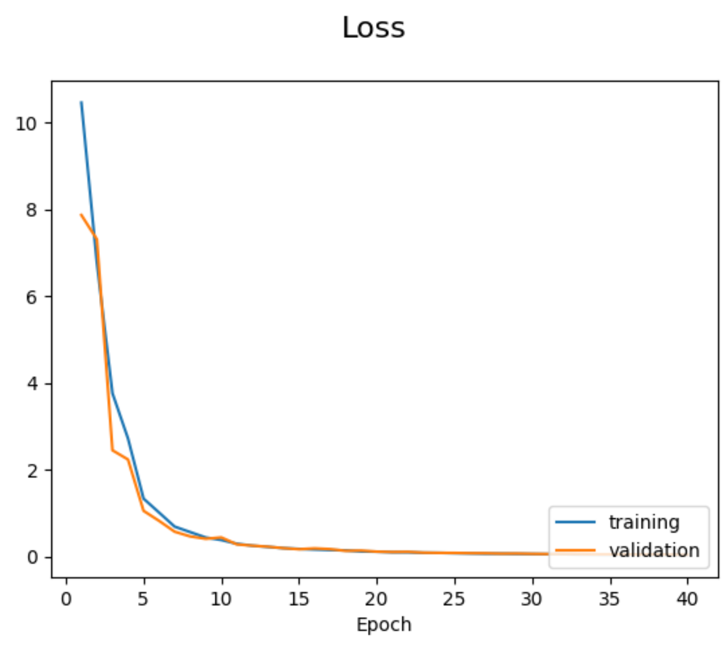


Figure 1: The cross-entropy loss curve of the model

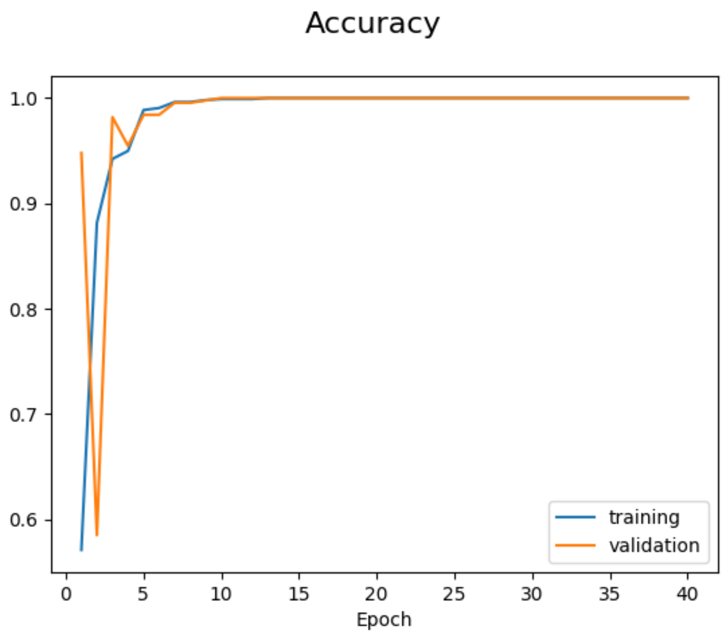
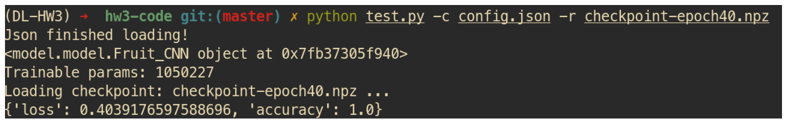


Figure 2: The accuracy curve of our model

2.4 Performace

I have provided a best model `checkpoint-epoch40.pth` and the config `config.json` under the root folder. You can view the testing accuracy by using the `test.py` given under the root folder. You can also use `checkpoint-epoch5.pth`, this is a mid-product while training, which has only 89% accuracy to show that the accuracy results are correct.



```
(DL-HW3) ➜  hw3-code git:(master) ✗ python test.py -c config.json -r checkpoint-epoch40.npz
Json finished loading!
<model.model.Fruit_CNN object at 0x7fb37305f940>
Trainable params: 1050227
Loading checkpoint: checkpoint-epoch40.npz ...
{'loss': 0.4039176597588696, 'accuracy': 1.0}
```

Figure 3: Results on our testing set

3 Result Visualization

In this part, we will show something other than the loss and accuracy. We will show the feature map of the 2 convolutions to visualize what the network is doing.

In the first row, it consists the input image and the three channels it has. The second and the third row are 4 of the 8 channels from our CNN layers. We can see that the first Convolution is extracting the overall feel from the input image, and the second Convolution can sort of getting the shape of the image.

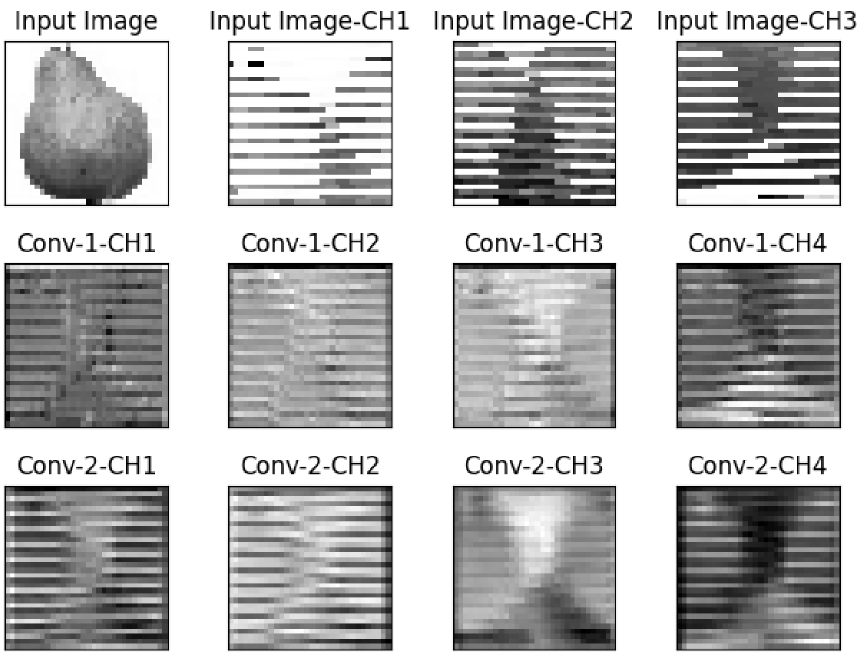


Figure 4: Pear 1

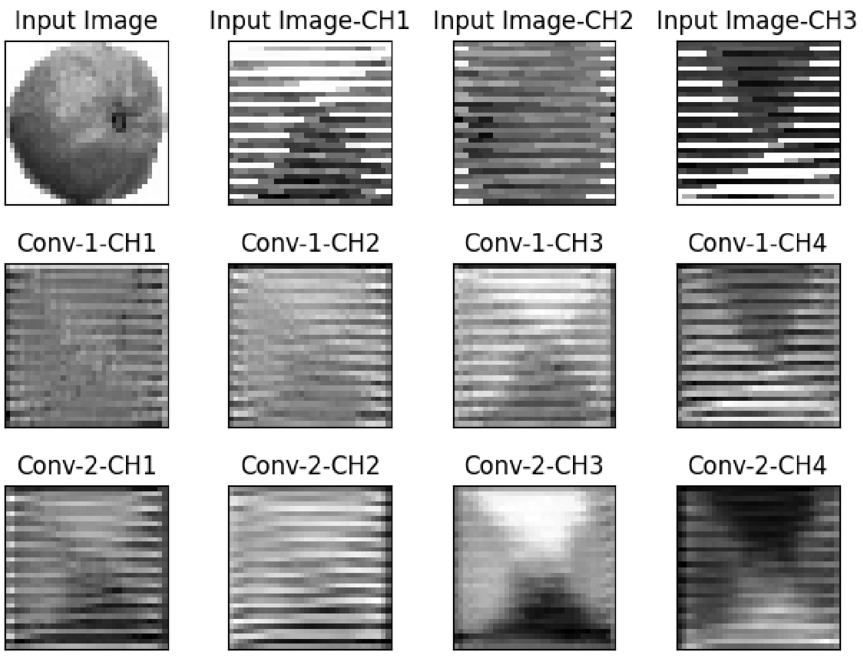


Figure 5: Pear 2

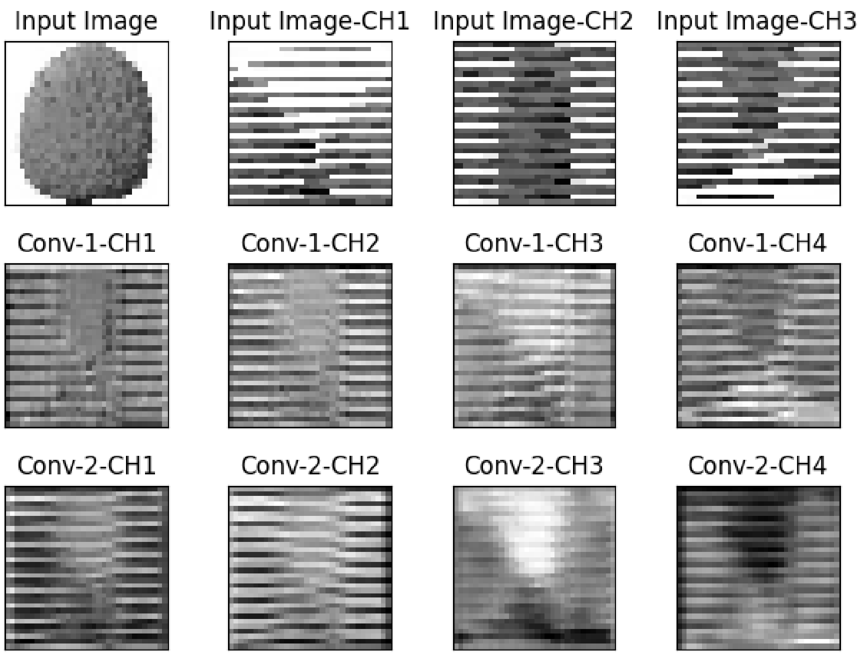


Figure 6: Lychee 1

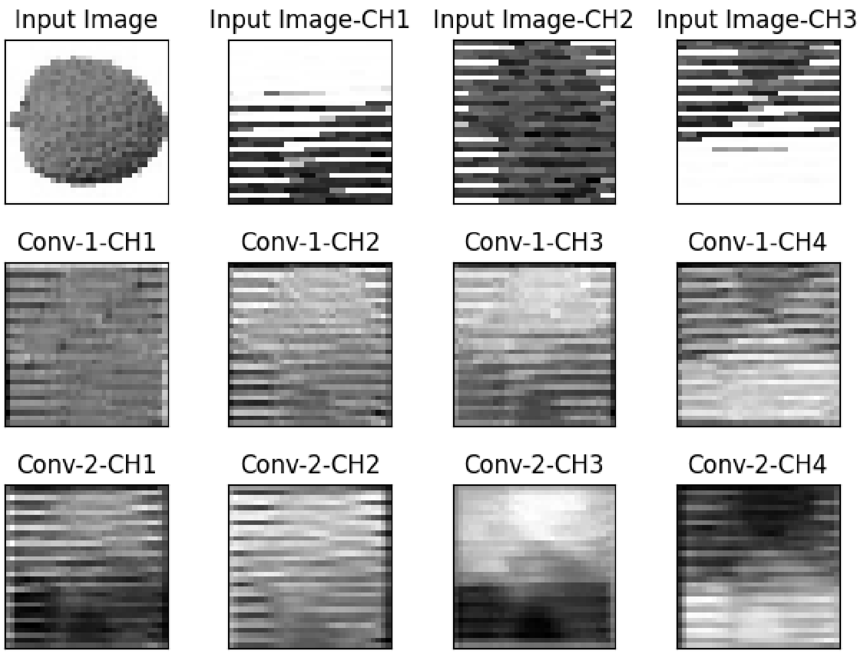


Figure 7: Lychee 2

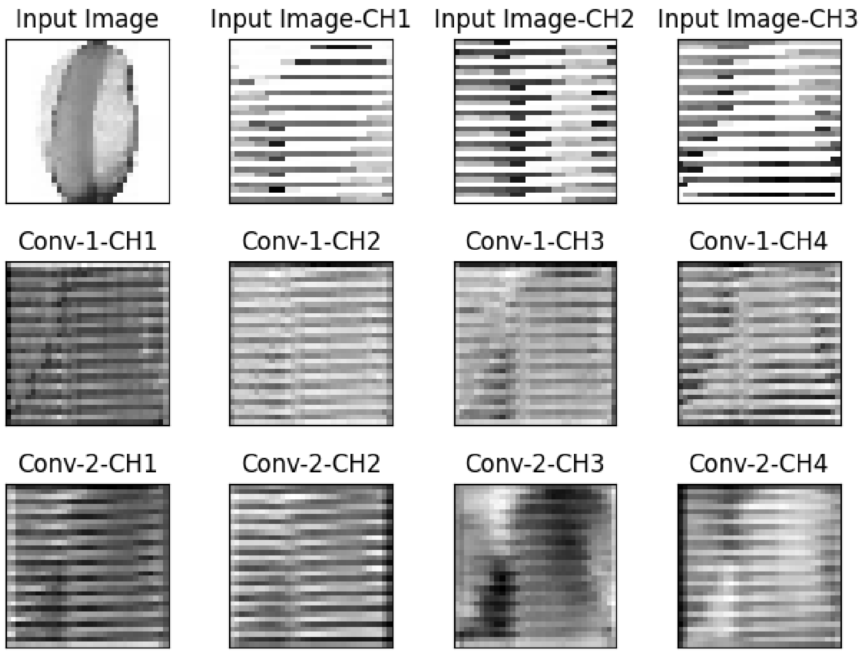


Figure 8: Carambola 1

4 Challenges

In the process of writing this homework, The hardest part is definitely implementing the convolution operation. At first, I wrote a triple for loop implementation(width, heght, number of kernels). However, that took forever to train with only CPU. So after closer incpection on the vectorized numpy code and some really tricky techniques, to stack up all the input and layers, I was able to nearly 10 times the speed on doing convolution.

The most interesting part is looking at the feature maps. This really gives insight of what our Convolution layers are looking at. However it took me quite a while to correctly show the feature maps, with only the matplotlib library.

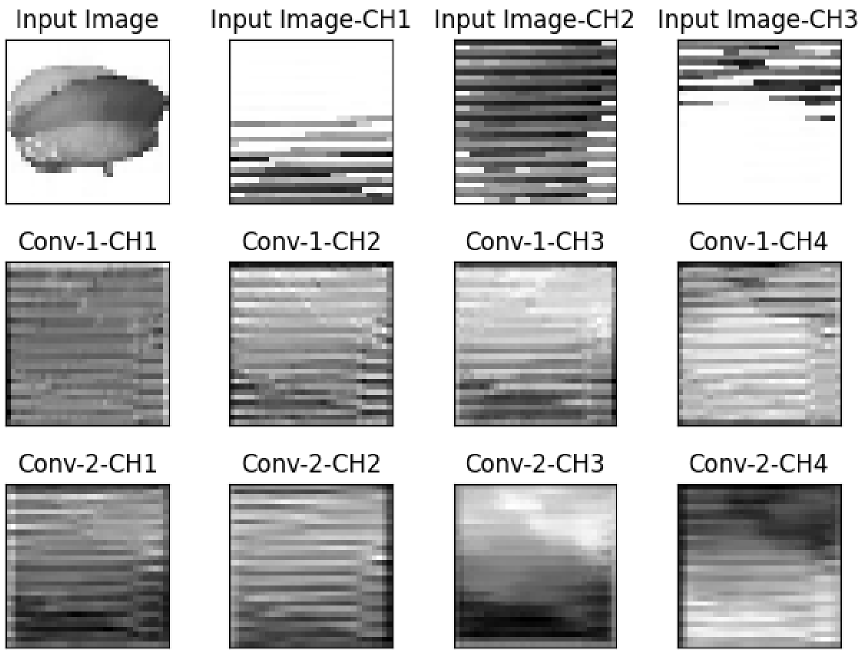


Figure 9: Carambola 2