NN Class HW3

N26094207 王昶文

1. Neuron Network

1.1 Derivation of learning method

For convolution network, it's very similar to neural network, except the connection between the output and input is limited by convolution kernels, instead of fully connections. Illustration as shown in Fig. 1. Suppose we have a 4x4 input image, and we use a 3x3 kernel with stride=1 to do convolution. We can transfer the input image into a 1x16 array, with 1x9 weights.

In order to calculate the image output, each pixel in input times the kernel separately, and then summed up, enable us to get the result of convolution, named y_i. After passing an activation function, will get the output of the convolution layer.

For back propagation, we receive the 2x2 error matrix, after multiplying the derivative of activation function, we can get the error of y_i. To adjust the error passing to the previous layers, we calculate sum up the errors connected to the pixel, that is: for x_2, error passed equals to $w_1x_2+w_2x_1$.

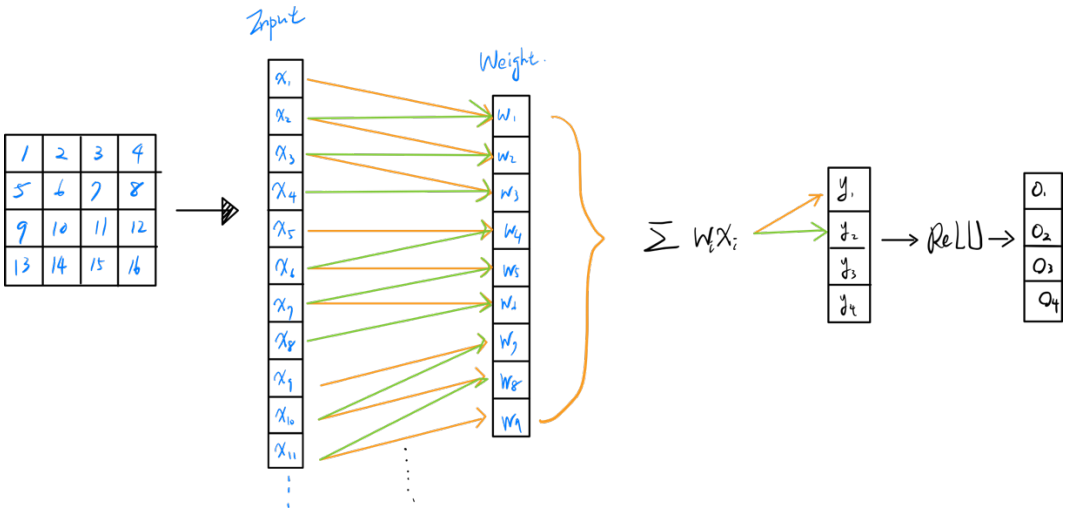

Fig. 1 Convolution structure

1.2 Code Architecture

For my codes, I used numpy as operation tool. There's a function called as strides, which allows a numpy matrix to change the shape of the matrix without

copying the matrix over and over again. For forward passing, I re-strides the matrix into the kernel size, shown in Fig.2, then, it can easily be operated with kernel, simply times 2 numpy matrix together.

For back propagation, I calculate the error passing by mapping the error back to the position it is contribute, as shown in Fig.3, then sum up to get the result to pass error. For weight adjustment, use errors multiplies the corresponding input we separated in Fig.2, and sum up as weight adjustment.

Model for my architecture is shown in Fig. 4, with 5x5 kernel x 3 feature maps in first convolution layer, and use average pooling with 4x4 kernel, strides = 2. Then fed into second convolution layer with 3x3 kernel x 5 feature maps, with average pooling of 3x3 kernel, strides=1. Then attached to a 3 layers Fully Connected Network, with 100, 64, 10 neurons respectfully. Last, attach it to a SoftMax layer as classifier.

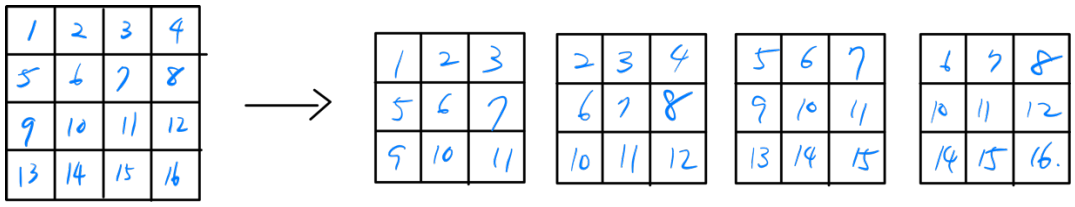

Fig. 2 Forward using as_strides operation

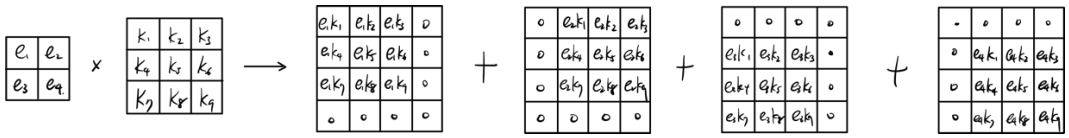

Fig. 3 Pass error using as_strides operation

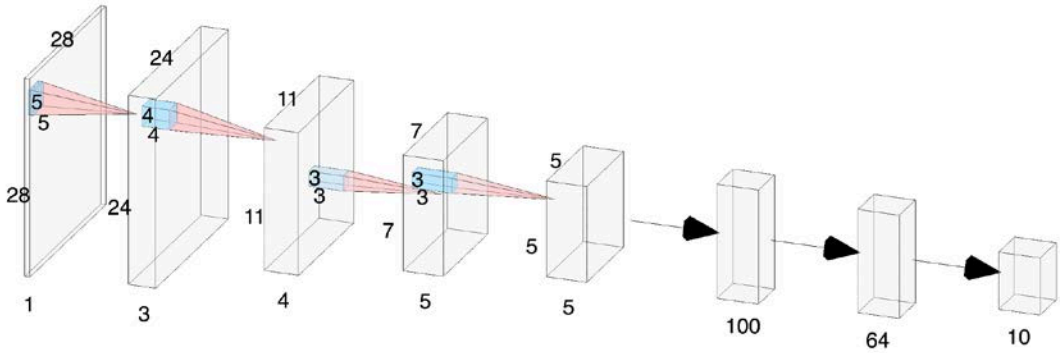

Fig. 4 Model Architecture

1.3 Results

Training time of my model is about 2 minutes an epochs, 60000 images in each epoch, and learning rate set to 0.001. Accuracy is about 96%, and loss is about 0.0996 at after 21 epochs. Results were shown in Fig.5-7, w represent

“with”, and w/o represents without.

Fig.8-10 are the results using batch size, it’s obvious that using batch size converges faster, and weight changes is smoother than not using batch.

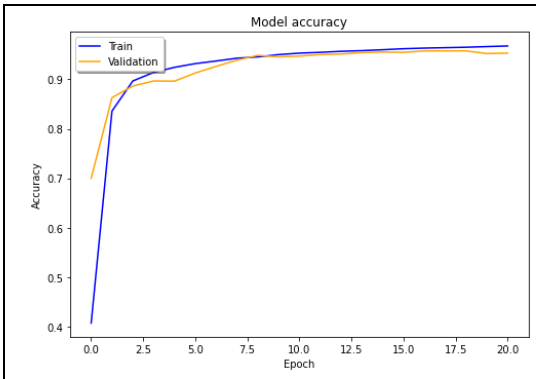

Fig. 5 Accuracy w/o batch

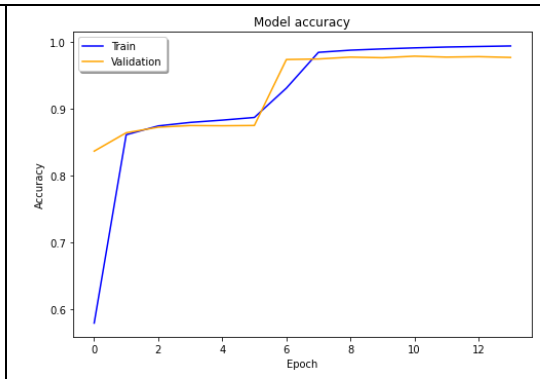

Fig. 8 Accuracy w batch

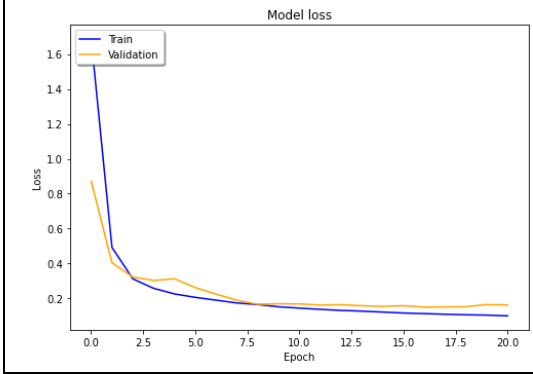

Fig. 6 Loss w/o batch

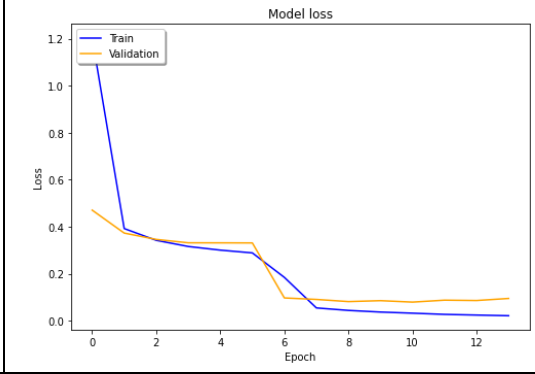

Fig. 9 Loss w batch

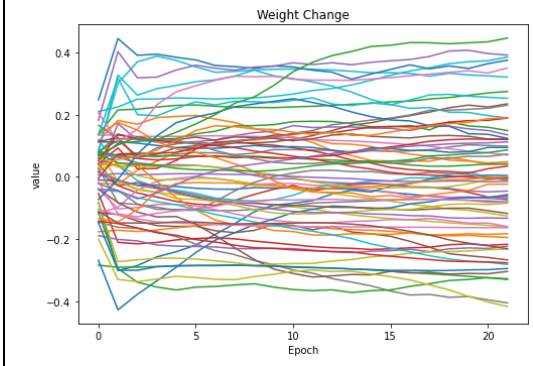

Fig. 7 Weight change w/o batch

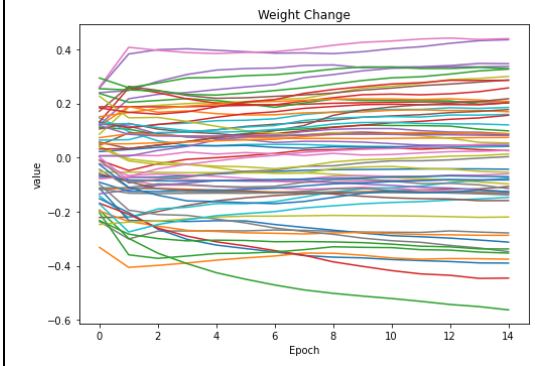

Fig. 10 Weight change w batch

1.4 Discussion

There are few problems I encountered. First is the weight initialization, another is the calculation time, batch size also effects, last is average pooling.

1.4.1 Weight initialization

It’s really tricky. In some worst case, the model can converge very poorly, train for 16 epochs resulting 10% on accuracy, which equals to random

guessing, in my model. In order to overcome this problem, I initial the weight using Xavier weight initialization, boundary of the random weight is shown as eq. (1). By using this method, my model is able to converge stabler, and has better result.

$$boundary = \sqrt{\frac{6}{N_{input} + N_{output}}} \qquad (1)$$

1.4.2 Calculation time

At first, my back propagation use “for loop” to calculate passing error, slow down the computation. Use about 1.5 hours to run 1 epoch. In order to speed up the computation, I turned to library of numpy. Using the method illustrated in Fig.3, I’m able to speed the computation time to 3 minutes per epoch.

1.4.3 Batch size

Batch size seems to be important to neural network. In my model, if the learning rate is too high, the training loss changes a lot, causing the model uneasy to converge. Therefore, adding batch size will be a better solution for the network to converge smoother. After adding batch size, my model is able to converge faster, reach 80% accuracy at 3rd epoch.

1.4.4 Average pooling problem

Using pooling, model is able to run faster, since pooling shrinks the image down, but it also blurs feature maps, therefore, it does reduce the accuracy. In my results, I got 98% accuracy at 8th epoch without average pooling, but it only has 96% accuracy when adding average pooling. However, average pooling did speed up training time. It takes 8 minutes per epoch when training without pooling, but only takes 2 minutes per epoch when using pooling.

Due to time constraints and lack of talent, I failed to finish the rest of the requirement.