Implementation Explanation

For assignment 1, a CNN classifier was built and tested on MNIST dataset. This report describes the codes required to be implemented. The theoretical explanation of convolution layer forward & backward, FC layer forward & backward, Pooling layer forward and backward, dropout layer forward & backward, SoftmaxCrossentropy forward & backward and Optimizer Adam. All symbols and annotations will follow the format in the lecture notes.

Convolution Forward

Input: $x \in R^{C \cdot l_h \cdot l_w}$

Kernel: $W \in R^{C \cdot k_h \cdot k_w}$

Output: $y \in R^{o_h \cdot o_w}$, where

$y_{ij} = \sum_{c=0}^{C} \sum_{a=0}^{k_h-1} \sum_{b=0}^{k_w-1} x_{i+a,j+b,x} \cdot W_{a,b,c}$,

$o_h = \left\lfloor \frac{l_h+p-k_h}{s} \right\rfloor + 1$, and $o_w = \left\lfloor \frac{l_w+p-k_w}{s} \right\rfloor + 1$

In above equations, *C* is number of channels, l_h is input height, l_w is input width, k_h is filter height, k_w is filter width, *p* is padding size, *s* is stride step. To calculate the output, simplest way is to use a for loop. However, it takes too long to run for loop. In the actual implementation, the image is transferred to column matrix for dot multiplication.

Convolution Backward

The backward gradient w.r.t *W* is defined as below

$\frac{\mathrm{d}E}{\mathrm{d}W} = \frac{\mathrm{d}E}{\mathrm{d}out} \cdot \frac{\mathrm{d}out}{\mathrm{d}W} = X^T \cdot \frac{\mathrm{d}E}{\mathrm{d}out}$

Where *E* is the mean squared error, *W* is the input weight, and *out* is the output

The backward gradient w.r.t *b* can be calculated as the sum of d_{out} for all the outputs affected by that bias.

Pooling Forward

Pooling reduces the dimensionality of each feature map but retains the most important information. Pooling can be of different types: Max, Average etc. In case of Max Pooling, we define a spatial neighbourhood and take the largest element from the rectified feature map within that window. Instead of taking the largest element we could also take the average (Average Pooling) in that window. To implement pooling layer, the feature map inputs are stacked. The matrix is transformed in a way that each column is a pooling window. The maximum (average) value is taken across the second dimension. The output is reshaped to a 4-dimension matrix with size $N \cdot f \cdot o_h \cdot o_w$, where N is batch, f is number of feature maps, o_h and o_w is output height and width respectively.

Pooling Backward

Max pooling

The backward max pooling layer back-propagates the input gradient $G \in R^{l_1 \cdot l_2 \cdots l_p}$ computed on the preceding layer. The backward layer propagates to the next layer only the elements of the gradient that correspond to the maximum values pooled from subtensors in the forward computation step.

To compute the value tensor $Z = (z_{i_1 \ldots i_p}) \in R^{n_1 \ldots n_p}$ such that:

$z_{i_1 \ldots i_p} = \frac{\mathrm{d}E}{\mathrm{d}x_{i_1 \ldots i_p}} = \sum_{j_{k_1}, a \in V_1(i_k), j_{k_2}, b \in V_2(i_k)} \delta_{a \cdot m_2 + b, t} g_{i_1 \ldots j_{k_1} \ldots j_{k_2} \ldots i_p}$

Average pooling

the backward average pooling layer back-propagates the input gradient $G = (g^{(1)}g^{(2)}...g^{(p)})$ of size $m_1 m_2 ... m_p$ computed on the preceding layer. The backward layer propagates the elements of the gradient multiplied by the coefficient $1/(f_{k_1} f_{k_2})$ to the corresponding pooled subtensors of the tensor *Z:*

$z_{i_1...i_{k_1}...i_{k_2}...i_p} = \frac{1}{f_{k_1} f_{k_2}} g_{i_1...j_{k_1}...j_{k_2}...i_p}, j_{k_l} = \left\lfloor \frac{i_{k_l}}{f_{k_l}} \right\rfloor, l \in 1, 2,$

Dropout Forward

For training

Given, Input: $x \in R^{H \cdot W}$, dropout probability: *p*,

We have, scale= $1 / (1 - p)$.

Mask- numpy. Random.binomial(1,(1-p),size=size of x)*($1 / (1 - p)$) x=x**Mask*

For testing

The input x is returned

Dropout Backward

For training the input x is multiplied by the *mask.*

For testing the input x is returned.

FC layer forward

The forward fully-connected layer computes values

$$y_i(x_1, \ldots, x_n) = \sum_{k=1}^{n} s_{ki} w_{ki} x_k + b_i$$

FC layer backward

The backward fully-connected layer computes the following values

$$z_k = \sum_{j=1}^{m} g_j \cdot s_{kj} \cdot w_{kj},$$

$$\frac{\partial \mathrm{E}}{\partial w_{kj}} = g_j \cdot s_{kj} \cdot x_k,$$

$$\frac{\partial \mathrm{E}}{\partial b_j} = g_j,$$

Where $k = 1, 2, ..., n, j = 1, 2, ..., m$. *E* is the objective function used at the training stage, and g_j is the input gradient computed on the preceding layer.

Model Tuning and Findings

Regularization

The original model does not contain drop out layer. Therefore, it is heavily overfitted though the training accuracy is high. To make the model generalize, one dropout layer is added after the first pooling layer. Therefore, the overall architecture is input-conv1-relu1-pooling1-dropout1-conv2-relu2-pooling2-flatten-fclayer1-relu3-fclayer2. Different dropout probabilities are tested, and the result is as below

Dropout Probability	0.2	0.3	0.4	0.5
Training Accuracy	0.9333	0.9667	0.93333	0.9667
Validation Accuracy	0.96117	0.9597	0.95425	0.9509
Testing Accuracy	**0.9624**	0.9614	0.9560	0.954

As shown in the result, dropout ratio 0.2 is the best among all experiments.

Optimization

Next, with dropout ratio set to 0.2, four optimization algorithms were tested. They are Adam, RMSprop, SGD and Adagrad.

Adam and RMSprop converged faster within fewer number of epoch compared SGD. The reason is these tow optimization algorithms update parameters individually with an adaptive learning rate. However, SGD + momentum and Adam are faster than RMSprop. The reason could be this model being a smaller model effect of individual parameter updates and adaptive learning rate is surpassed by the effect of momentum. Adam being fastest, its best training accuracy is 93.33% and test accuracy is 96.24%. Therefore, it was decided to remain with Adam.

Therefore, the best classifiers among all the experiments conducted was below

- Optimization mechanism- Adam
- Dropout ratio-0.2