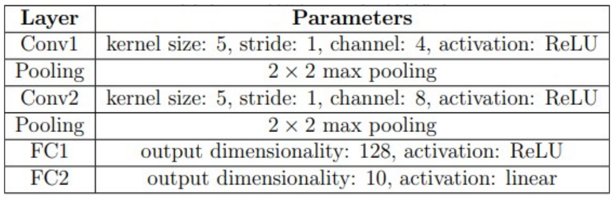
CMPE 597 HW1

İzzet Emre Küçükkaya
Boğaziçi University
Istanbul, Turkey
izzet.kucukkaya@boun.edu.tr

I. Introduction

In this homework, it is asked to implement convolutional neural network and test it using MNIST data. Then for sanity check pytorch library is used. The network is asked to be implemented is Fig. 1



Layer	Parameters
Conv1	kernel size: 5, stride: 1, channel: 4, activation: ReLU
Pooling	2×2 max pooling
Conv2	kernel size: 5, stride: 1, channel: 8, activation: ReLU
Pooling	2×2 max pooling
FC1	output dimensionality: 128, activation: ReLU
FC2	output dimensionality: 10, activation: linear

Fig. 1. CNN Layers [1]

Furthermore, it is asked to implement a simple neural network to make binary classification using 2 dimensional future space. The data can be seen in Fig. 2 where * plots are the test data and circles are train data.

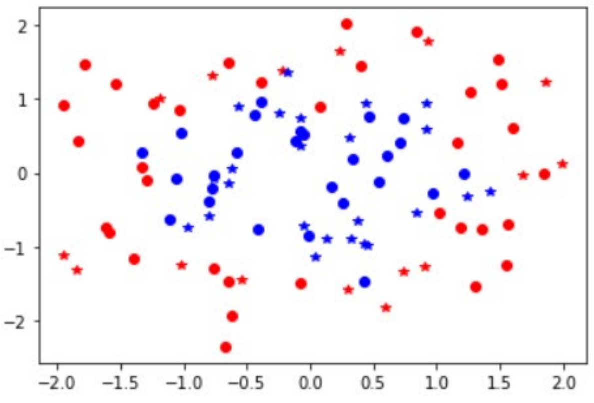


Fig. 2. Question 2 Data Plot

II. Implementation

A. Convolution Layer

First things first, convolution process is defined. In order to define that layer, first, forward method is defined

1) Forward: Let input is 3D matrix with [input_channel, input_size, input_size] dimensions
Let a filter is 2D matrix with [kernel_size, kernel_size] x Let $output\_size = \frac{input\_size - kernel\_size}{stride} + 1$
Let output is 3D matrix with [output_channel, output_size, output_size] dimensions
out_i is defined as the ith channel of output
Convolution is defined:

$$out_i[k, m] = \sum\sum_j filter_i * input\_part_j$$

where $input\_part_j$ is 1 x kernel x kernel sized part of jth channel of input and part from k*stride to k*stride + kernel and from m*stride to m*stride + kernel

2) Backward: In forward process input and output is stored. In this section, gradient must be calculated and weights must be calculated using this gradient and pass the resultant gradient to the remaining layers. Input_part is defined in the same way. The grad matrix taken as input from forward layers and out_grad matrix is passed to backward layers.
$grad\_filter_i = \sum_j grad[i, k, m] * input\_part_j$
$grad\_out\_part_j = \sum_i filter_i * grad[i, k, m]$
where $grad\_out\_part_j$ is jth channel of output grad's part from k*stride to k*stride + kernel and from m*stride to m*stride + kernel and $input\_part_j$ is 1 x kernel x kernel sized part of jth channel of input and part from k*stride to k*stride + kernel and from m*stride to m*stride + kernel
Then filters are recalculated using grad_filter
filters = filters - grad_filter * learning_rate

B. Max Pooling

1) Forward: output[i,j,k] = max($input\_part_i$)
where $input\_part_i$ is 1 x kernel x kernel sized part of ith channel of input and part from j*stride to j*stride + kernel and from k*stride to k*stride + kernel
2) Backward: (x,y) = argmax($input\_part_i$)
$out[i, j * stride + x, k * stride + y] = grad[i, j, k]$
where $input\_part_i$ is 1 x kernel x kernel sized part of ith channel of input and part from j*stride to j*stride + kernel and from k*stride to k*stride + kernel

C. Fully Connected

1) Forward: First the input data is flatten. Weight is defined as (input_size * output_size) 2D matrix and bias is defined as (output_size) 1D vector then,
$out = input * weight + bias$
2) Backward: In order to recalculate bias and weight,
grad_weight = input * grad
grad_bias = 1 * grad

Then,
$grad\_out = grad * weight$
$weight = weight - grad\_weight * learning\_rate$
$bias = bias - grad\_bias * learning\_rate$

D. Softmax and Cross Entropy Loss Function

1) Forward: $softmax(x) = \frac{e^x}{\sum e^{x_i}} and$
$CrossEntropy(y, y\_pred) = \sum -y_i * log(y\_pred_i)$
in our case $y_i = 1$ where i is the label and 0 otherwise.

2) Backward: First we need to calculate dCrossEntropy then, dSoftmax
$dCrossEntropy(y\_pred):$
$y\_pred[label] = y\_pred[label] - 1$
$dSoftmax(x) = Softmax(x) * (1 - Softmax(x))1001[2]$
and
$grad : y\_pred[label] = y\_pred[label] - 1$
$grad\_out = grad * dSoftmax(input)$

E. ReLU

1) Forward: relu(x) = x if x $>$ 0 else 0

2) Backward: drelu(x) = 1 if x $>$ 0 else 0
and $grad\_out = drelu(input) * grad$

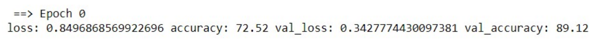
F. Overall Implementation

In order to calculate loss and the output probabilities, all feedforward processes of all layers are done one by one starting from the beginning. After loss and output probabilities are calculated, backward processes of layers are done starting from the last layer and going backwards. Stochastic gradient descent is used. Output dimensions of layers are following:
Conv1: 4x24x24
MaxPool1: 4x12x12
Conv2: 8x8x8
MaxPool2: 8x4x4
FC1: 128
FC2: 10

III. Results

A. MNIST Data

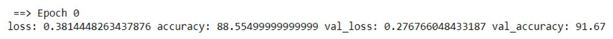
1) From Scratch: My own implementation lasts too long even tough i used numba library to parellalize fors. Therefore, number of epoch is chosen 1.
Also, input data is normalized to [-0.5, 0.5] range.



```
 ==> Epoch 0
loss: 0.8496868569922696 accuracy: 72.52 val_loss: 0.3427774430097381 val_accuracy: 89.12
```

Fig. 3. Own Implementation Result After 1 Epoch

As can be seen in the Fig. 3, after only 1 epoch, approximately 90% accuracy has been reached.
Then I wanted to test what happens if learning rate increases. After I increase learning rate the loss is started to decrease faster but as the learning process continues, exp function in softmax is overflowed. Therefore, I tried a learning rate

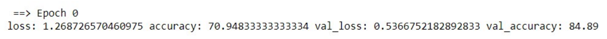


```
 ==> Epoch 0
loss: 0.3814448263437876 accuracy: 88.55499999999999 val_loss: 0.276766048433187 val_accuracy: 91.67
```

Fig. 4. With Learning Rate Scheduler

scheduler to start with a high valued learning rate and decrease it gradually. Then the result can be seen in Fig. 4

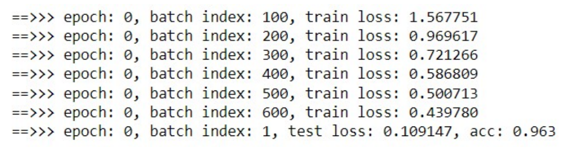
Also I tested the dataset using only 2 fully connected layers model. The results can be seen in Fig. 5



```
 ==> Epoch 0
loss: 1.268726570460975 accuracy: 70.94833333333334 val_loss: 0.5366752182892833 val_accuracy: 84.89
```

Fig. 5. Own Implementation Using Only 2 FC Layer

2) Using Pytorch: In order to be able to compare, i trained pytorch network for 1 epoch either with learning rate equals to 0.01. The results can be seen in Fig. 6.



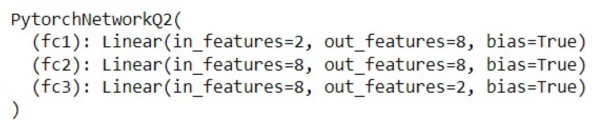
```
==>>> epoch: 0, batch index: 100, train loss: 1.567751
==>>> epoch: 0, batch index: 200, train loss: 0.969617
==>>> epoch: 0, batch index: 300, train loss: 0.721266
==>>> epoch: 0, batch index: 400, train loss: 0.586809
==>>> epoch: 0, batch index: 500, train loss: 0.500713
==>>> epoch: 0, batch index: 600, train loss: 0.439780
==>>> epoch: 0, batch index: 1, test loss: 0.109147, acc: 0.963
```

Fig. 6. Pytorch Implementation of CNN

As can be seen in figures, the result of the model using pytorch are greater but own implementation is not bad at all.

B. Q2 Dataset

The Dataset plot can be seen in Fig. 2. I defined a pytorch model which can be seen in Fig. 7.



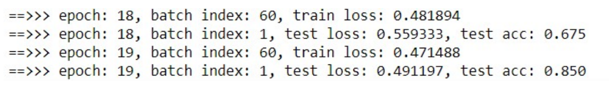
```
PytorchNetworkQ2(
  (fc1): Linear(in_features=2, out_features=8, bias=True)
  (fc2): Linear(in_features=8, out_features=8, bias=True)
  (fc3): Linear(in_features=8, out_features=2, bias=True)
)
```

Fig. 7. Pytorch Implementation of Simple Network

Furthermore the training results can be seen in Fig. 8.

1) Plots: In Fig. 9 greens represent the data which are correctly classified and the reds represent which are not.
In Fig. 10, the decision boundaries can be seen. As we can see it is a nonlinear decision boundary thanks to ReLU.

2) Can nonlinear decision boundaries be obtained using ReLU?: Because ReLU is a nonlinear function thanks to its breaking point at 0, we can obtain non-linear decision boundaries using ReLU.
In order for a process to be a linear,
$a * f(x) + b * f(y) = f(ax + by)$ but
$-1 * ReLU(1) = -1$ and $ReLU(-1 * 1) = 0$.
Because they are not equal, ReLU is a nonlinear process
Then, let $g(x) = k(ReLU(f(x)))$ is the decision boundary where f and k are $k(x) = ax + b$
$f(x) = cx + d$ and
$g(x) \geq 0 : class_0$
$g(x) < 0 : class_1$



```
==>>> epoch: 18, batch index: 60, train loss: 0.481894
==>>> epoch: 18, batch index: 1, test loss: 0.559333, test acc: 0.675
==>>> epoch: 19, batch index: 60, train loss: 0.471488
==>>> epoch: 19, batch index: 1, test loss: 0.491197, test acc: 0.850
```

Fig. 8. Results of Simple Network

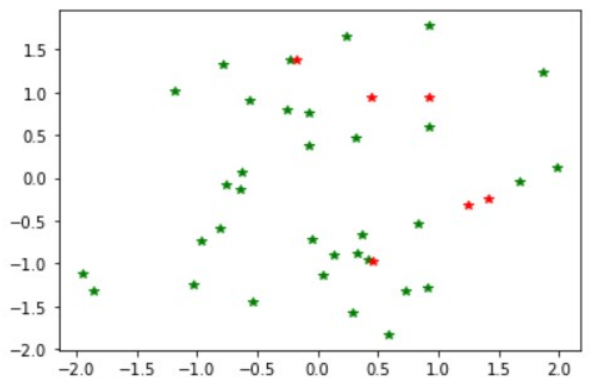


Fig. 9. True - False Detections

Then, $g(x) = a(ReLU(cx + d)) + b$
Where $cx + d > 0$
$g(x) = acx + da + b$
Where $cx + d \leq 0$
$g(x) = b$
Therefore, we get nonlinear decision boundaries as in Fig. 10

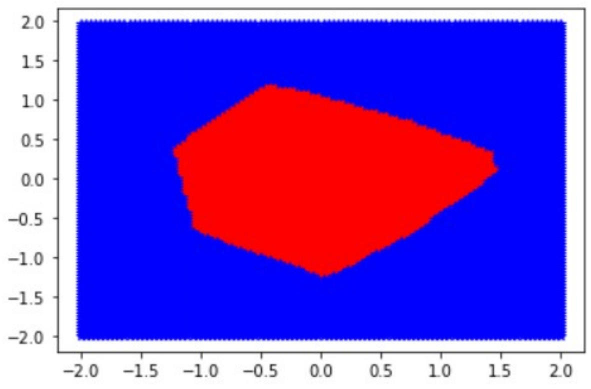


Fig. 10. Decision boundaries

References

[1] I. M. Baytaş, "CMPE 597 HW1 Description," 2022.
[2] Q. Viper, "Convolutional Neural Networks From Scratch on Python."