DATA130011 PJ1

Yuquan Zhou

April 30, 2025

1 Introduction

This project implements neural networks using no ready-made deep learning tools, e.g. PyTorch. MLP and CNN networks are trained on the MNIST dataset for handwritten digit recognition. A package named **MyDL** is built on Numpy, which includes basic building blocks and functions for building and training a neural network.

All code has been released to github in branch 2025spring.

Dataset and best model weights can be found in OneDrive.

2 Project Structure

The code structure of the program is shown below:

```
Lab1_NNFromScratch ..... root directory
├── dataset ..... MNIST dataset
├── train_model.py ..... Training script
├── test_model.py ..... Test the model's accuracy
├── visualization.ipynb ..... Visualization of weights of trained models
├── playground.ipynb ..... Example of usage of MyDL
├── scripts ..... Shell scripts for training and testing
│   ├── train_mlp.sh
│   ├── train_resnet.sh
│   ├── test_mlp.sh
│   └── test_resnet.sh
├── MNIST_result ..... Trained model weights and results of MNIST
│   ├── figs ..... Visualization figures
│   ├── model_params ..... Model weights
│   └── resultsTraining log files(.npz) and plots
└── MyDL ..... Package written for deep-learning tasks
    ├── __init__.py
    ├── runner.py ..... Defines the runner class
    ├── data.py ..... dataset and dataloader class
    ├── optimizer.py ..... Optimizer and Scheduler class
    ├── tensor.py ..... Tensor class
    ├── nn ..... Neural network components
    │   ├── __init__.py
    │   ├── layers.py ..... Layers of neural networks
    │   ├── loss_func.py ..... loss functions
    │   └── network.py ..... NeuralNetwork class
    └── sample_networks ..... Networks used in this PJ
        ├── __init__.py
        ├── resnet.py
        └── mlp.py
```

3 Environment and Dataset

3.1 Environment

- Python Version: 3.10.16
- Cupy Version: cuda-11x 13.4.1

To boost running speed, the project adopts Cupy instead of Numpy. However, most operations are compatible between these two packages, so it is easy to switch to Numpy. Just change all "import cupy" to "import numpy" in MyDL is OK. The only difference is in loading npz files, which is already handled in the code.

Cupy relies on the CUDAToolkit. Please install CUDAToolkit first, and install Cupy according to your CUDAToolkit version.

3.2 Dataset

Uses the MNIST dataset of handwritten digits. It contains 60,000 training images and 10,000 testing ones. In this experiment, we divide training images into 50,000 in training set and 10,000 in validation set. **Experiment results are based on the scores obtained on validation set.**

4 Introduction to MyDL

MyDL is the core of the project. It is a mimic of Pytorch, thus the usage of it is alomost identical to PyTorch. Networks built using PyTorch can be migrated to MyDL easily with hardly any modification. MyDL contains most important components of a deep learning tool, and allows for convenient building and training of neural networks. Currently it only supports MLP and CNN networks.

4.1 MyTensor

The basis of MyDL is the tensor class named **MyTensor** defined in tensor.py. Similar to tensor in PyTorch, MyTensor supports **dynamically building computation graph** while doing calculations. The most commonly used binary operations including summation, subtraction, element-wise multiplication, matrix multiplication, and unary operations like exponential, indexing, expanding dimensions, etc., are supported.

While doing such operations on MyTensor objects, a father node(output) is generated(if requires gradient) and links are built between input and output nodes. Applying **.backward()** on a MyTensor object assigns that object with gradient 1 on every element, and then calculates gradient of all nodes involved in the computational graph(and requires grad). The gradient is propagated in a backward manner. To be specific, the backward() method first searches the graph using DFS, recursively building the topological order of nodes. Then the method calculates local gradient on each node in topological order and delivers it to the children of each node.

The MyTensor class benefits follow-up works. When writing layers that do not need special performance optimization, we can simply write the forward process without the need of worrying about to calculating backward gradient. However, we still need to specify the backward manner for layers like Conv2D, as for-loop is too slow.

4.2 nn

In nn all the layers used in this project are defined. Also a NeuralNetwork class which is analogous to nn.Module in PyTorch is defined. I also defined a Sequential class that plays the same role with Sequential in PyTorch. To build a model, you have to add layers to the model's attribution, and define the forward process(backward is not required). Layer parameters will be automatically added to the model. It is overall identical to PyTorch. Below is an example of building a ResNet:

```
from MyDL import nn

class ResiduleBlock(nn.NeuralNetwork):
    """
    Define the residual block
    """
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2D(in_channels, out_channels, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2D(out_channels, out_channels, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.cross_block = False
        if stride != 1 or in_channels != out_channels:
            self.cross_block = True
            self.conv_shortcut = nn.Conv2D(in_channels, out_channels, kernel_size=1,
                                           padding=0, stride=stride, bias=False)
            self.bn_shortcut = nn.BatchNorm2d(out_channels)

    def forward(self, x):
        out = nn.ReLU.forward(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        if self.cross_block:
            x = self.conv_shortcut(x)
            x = self.bn_shortcut(x)
        out = out + x  # residual connection
        out = nn.ReLU.forward(out)
        return out

class ResNetMNIST(nn.NeuralNetwork):
    """
    input: (batch, 1, 28, 28)
    ↓
    Conv 3x3, 16 channels + BN + ReLU
    ↓
    ResiduleBlockx2 (channel = 16, stride=1)
    ↓
    ResiduleBlockx2 (channel = 32, stride=2)
    ↓
    ResiduleBlockx2 (channel = 64, stride=2)
    ↓
    Average Polling (len 64 vector)
    ↓
    FC layer (64 -> 10)
    ↓
    out: (10)
    """
    def __init__(self, block=ResiduleBlock, num_classes=10):
        super(ResNetMNIST, self).__init__()
        self.in_channels = 16

        self.conv = nn.Conv2D(1, 16, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn = nn.BatchNorm2d(16)
```

```
        self.layer1 = self._make_layer(block, 16, 2, stride=1)
        self.layer2 = self._make_layer(block, 32, 2, stride=2)
        self.layer3 = self._make_layer(block, 64, 2, stride=2)
        self.avg_pool = nn.FullAveragePool2d()
        self.fc1 = nn.Linear(64, 10)

    def _make_layer(self, block, out_channels, blocks, stride):
        strides = [stride] + [1] * (blocks - 1)
        layers = []
        for s in strides:
            layers.append(block(self.in_channels, out_channels, s))
            self.in_channels = out_channels
        return nn.Sequential(*layers)

    def forward(self, x):
        out = nn.ReLU.forward(self.bn(self.conv(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.avg_pool(out)  # (batch, c)
        out = self.fc1(out)
        out = nn.Softmax.forward(out)
        return out
```

4.3 Other Components

Other components include optimizer, scheduler, dataset, dataloader, etc.

5 Important Implementation Details

5.1 Layers

5.1.1 Conv2D

Convolution operation requires for-loop, which will lower the running speed dramatically. To solve this problem, I used the as_strided() function in Numpy(Cupy) to change the striding behavior of an array without changing the memory address of each element, and uses einsum() to conveniently and efficiently do multiplication and summation operation between two tensors(the input and the convolutional kernel). Backward gradient calculation is also implemented in this way.

5.1.2 Batch Normalization

Batch normalization is computed as:

$$\boldsymbol{\mu}_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|}\boldsymbol{z}_i$$

$$\boldsymbol{\sigma}^2_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\sum_{i=1}^{|\mathcal{B}|}(\boldsymbol{z}_i - \boldsymbol{\mu}_{\mathcal{B}}) \odot (\boldsymbol{z}_i - \boldsymbol{\mu}_{\mathcal{B}})$$

$$\hat{\boldsymbol{z}}_i = \frac{\boldsymbol{z}_i - \boldsymbol{\mu}_{\mathcal{B}}}{\sqrt{\boldsymbol{\sigma}^2_{\mathcal{B}} + \epsilon}}$$

Where z is the input and ϵ is a small number(1e-8). This layer acts differently in training and evaluation. In training it is calculated as above, and gradient $\nabla_{\boldsymbol{z}}\hat{\boldsymbol{z}}$ needs to be calculated in a complete manner(gradients should not be aborted on $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. And in evaluation, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ should be the mean and variance of the whole dataset.

5.1.3 Softmax

Softmax is numerically unstable if calculated directly. It is instead calculated as:

$$\frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}} = \frac{e^{x_i - \max_i x_i}}{\sum_{i=1}^{n} e^{x_i - \max_i x_i}}.$$

This avoids overflow.

5.1.4 Tanh

Tanh is calculated as:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}.$$

It's derivative is:

$$\frac{\partial}{\partial x}\tanh(x) = 1 - \tanh^2(x).$$

I used this to faster the calculation, instead of relying on the auto grad feature of MyTensor.

5.2 Optimizer

SGD, momentum and Adam are implemented. Uses Adam for training as it performs the best. Adam updates model weights following algorithm 1.

Algorithm 1 *Adam*

Require: α : Initial learning rate
Require: $\beta_1, \beta_2 \in [0, 1)$: dacay rate of momentum and average gradient norm.
Require: $\boldsymbol{\theta}_0$: Initial parameter
  $m_0 \leftarrow 0$
  $v_0 \leftarrow 0$
  $t \leftarrow 0$
  repeat
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_{\boldsymbol{\theta}} f_t(\boldsymbol{\theta}_{t-1})$
    $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$
    $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
    $\hat{m}_t \leftarrow \frac{m_t}{1-\beta_1^t}$
    $\hat{v}_t \leftarrow \frac{v_t}{1-\beta_2^t}$
    $\boldsymbol{\theta}_t \leftarrow \boldsymbol{\theta}_{t-1} - \alpha \frac{\hat{m}_t}{t^p \sqrt{\hat{v}_t + \epsilon}}$
  until end

5.3 Loss Function

5.3.1 Cross Entropy

Cross entropy is calculated as:

$$\text{loss} = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{n} y_{ij} \log(\hat{y}_{ij}).$$

Where m is the sample number, and n is the class number. y_{ij} is the label, and $\hat{y_{ij}}$ is the prediction. When calculating, set $\hat{y_{ij}} = 1$ where $y_{ij} = 0$, and $\hat{y_{ij}} = \epsilon$ where $y_{ij} = 1$ to avoid $\log 0$.

5.4 Initialization

- Xavier initialization is used in FC layer. Initial value follows a uniform distribution in range $[-r, r]$ where $r = \sqrt{\frac{6}{M_{l-1} + M_l}}$, where M_l is the number of neurons of the l^{th} layer.

- Kaiming initialization is used in Conv2D layer. Here $r = \sqrt{\frac{6}{fan_{in}}}$ where $fan_{in} = c_{in} * h_k * h_w$, c_{in} is the input channel, h_k and w_k is the height and width of the kernel.

6 Experiments

6.1 Experiment Environment

All experiments are carried out on a single RTX4090.

6.2 Experimental settings

Experimental settings and corresponding results are shown in table 1 and table 2. Best models are tested on test set.

	Settings								Experimental Result				
	Hidden Size 1	Hidden Size 2	Dropout Rate	L2-Norm	Learning Rate	Batch Size	Optimizer	Epochs	Train. Loss	Val. Loss	Train. Acc.	Val. Acc.	Test Acc.
1	100	\	\	0	0.05	256	Adam	30	0.311	0.146	0.905	0.960	
2	600	\	\	0	0.05	256	Adam	30	0.214	0.112	0.940	0.972	
3	100	100	\	0	0.05	256	Adam	30	0.053	0.163	0.984	0.963	
4	600	100	\	0	0.05	256	Adam	30	**0.048***	0.135	**0.985***	0.967	
5	600	100	0.3	0	0.05	256	Adam	30	0.114	0.107	0.965	0.969	
6	600	100	0.3	0.5	0.05	256	Adam	30	0.132	**0.091***	0.959	**0.973***	0.979
7	600	100	0.5	0	0.05	256	Adam	30	0.196	0.119	0.939	0.967	
8	600	100	0.3	0	0.1	256	Adam	30	0.128	0.132	0.960	0.969	
9	600	100	0.3	0	0.01	256	Adam	30	0.104	0.105	0.967	0.971	
10	600	100	0.5	0	0.05	256	SGD	30	0.152	0.127	0.953	0.965	

Table 1: MLP Settings and Results

	Settings						Experimental Result				
	Learning Rate	Scheduler	Augment Ratio	Batch Size	Optimizer	Epoch	Train. Loss	Val. Loss	Train. Acc	Val. Acc.	Test Acc.
1	0.01	\	0	256	Adam	30	0.013	0.025	0.996	0.993	
2	0.01	\	0.5	256	Adam	30	0.015	0.021	0.996	0.994	
3	0.01	MultiStepLR	0.5	256	Adam	30	**0.010***	**0.017***	**0.998***	**0.994***	0.995

Table 2: ResNet Settings and Results

6.3 Findings

From the results above, we have the following findings:

1. **Hidden size**: Increasing the hidden size effectively lowers the training loss(comparing the 1st & 2nd, 3rd & 4th row in table 1).

2. **Number of hidden layers**: Increasing number of hidden layers from 1 to 2 makes training loss lower, which implies the enhancement in model expressiveness(comparing the 1st & 3rd, 2nd & 4th row in table 1). However, validation loss becomes even higher, which is a result of overfitting. Interestingly, ResNet achieved lower value of both training and validation loss. This may be because of better model structure of CNNs, which is more suitable for image information extraction than MLP networks. We can further conclude that whether overfitting depends on not only the model complexity, but model structure.

3. **Dropout**: Dropout increases training loss, but lowers validation loss at the same time.(Comparing the 4th & 5th/7th row in table 1) This implies mitigation of overfitting problem. This is because dropout can be regarded as ensemble learning. The expressiveness of active neurons in a forward process is lowered(also not in perfect accordance with training target), but in evaluation the result takes the advice of all models. However, it is also noticeable that best performance is achieved at a dropout rate of 0.3, not 0.5, which means dropout rate shouldn't be too high.

4. **Learning rate**: Learning has imposes influence on model training as well.(Comparing the 7th & 8th & 9th rows of table 1) Besides the convergence rate, it also affects model's final performance.

We can see that 0.01 learning rate is best for (600, 100) MLP training, with 0.1 coming second, and 0.05 performing the worst. However, the difference is relatively small.

5. **Optimizer**: Comparing 7th & 10th rows of table 1, Adam and SGD leads to similar final results. However, it is observed that SGD converges much slower at the start of training than Adam.

6. **Data Augmentation**: In training of ResNet, I set an experiment in which augmentation is made to the input images with probability 0.5(See the 2nd row in table 2). The augmentation consists of randomly rotating the image in [-45, 45] degrees and changing the brightness and contrast. It is observed that the training loss raises, but validation loss drops. And the model achieves better final result in respect of validation accuracy. This also indicates less overfitting, and more knowledge the augmented data brought to the model.

7. **Scheduler**: A scheduler dynamically adjusts the learning rate of the optimizer(See row 3 in table 2). MultiStepLR scheduler is adopted in one experiment. At 2000 and 4000 iteration, the learning rate shrinks by a ration of 0.2. It is found that the model converged better with this setting. Reason could be that model weights needs finer update in later stages of training.

8. **L2 normalization**: L2 normalization is adopted in row 6 of table 1. Comparing the 6th & 4th row of table 1, it can be verified that L2 norm serves as a strong method of mitigating overfitting. The model achieves best final results among all.

6.4 Training History

Training history of models' training & validation loss, training & validation accuracy is shown in fig. 5

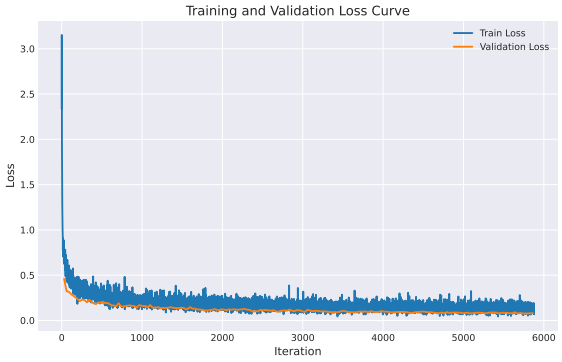

Figure 1: Loss-MLP

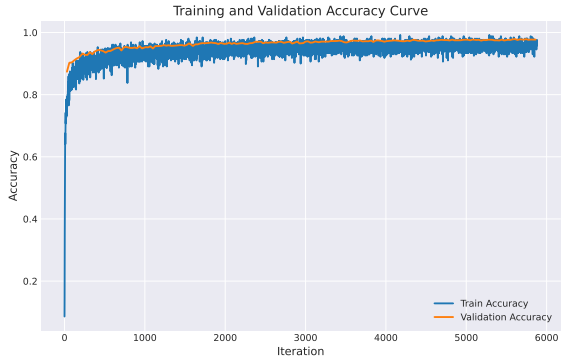

Figure 2: Accuracy-MLP

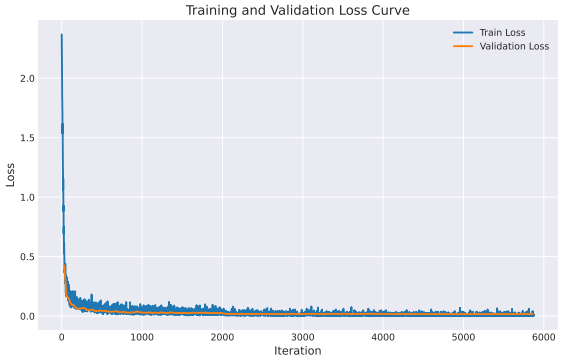

Figure 3: Loss-ResNet

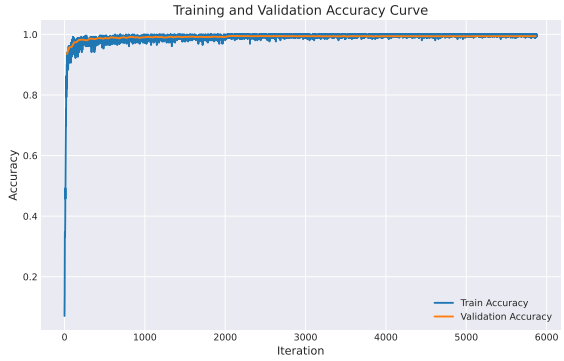

Figure 4: Accuracy-ResNet

Figure 5: Training history

6.5 Weight Visualization

6.5.1 MLP

fig. 6 is the visualization of the weight of the first FC layer of the MLP network(following the setting of row 6 in table 1). We can see vague digits in the visualization.

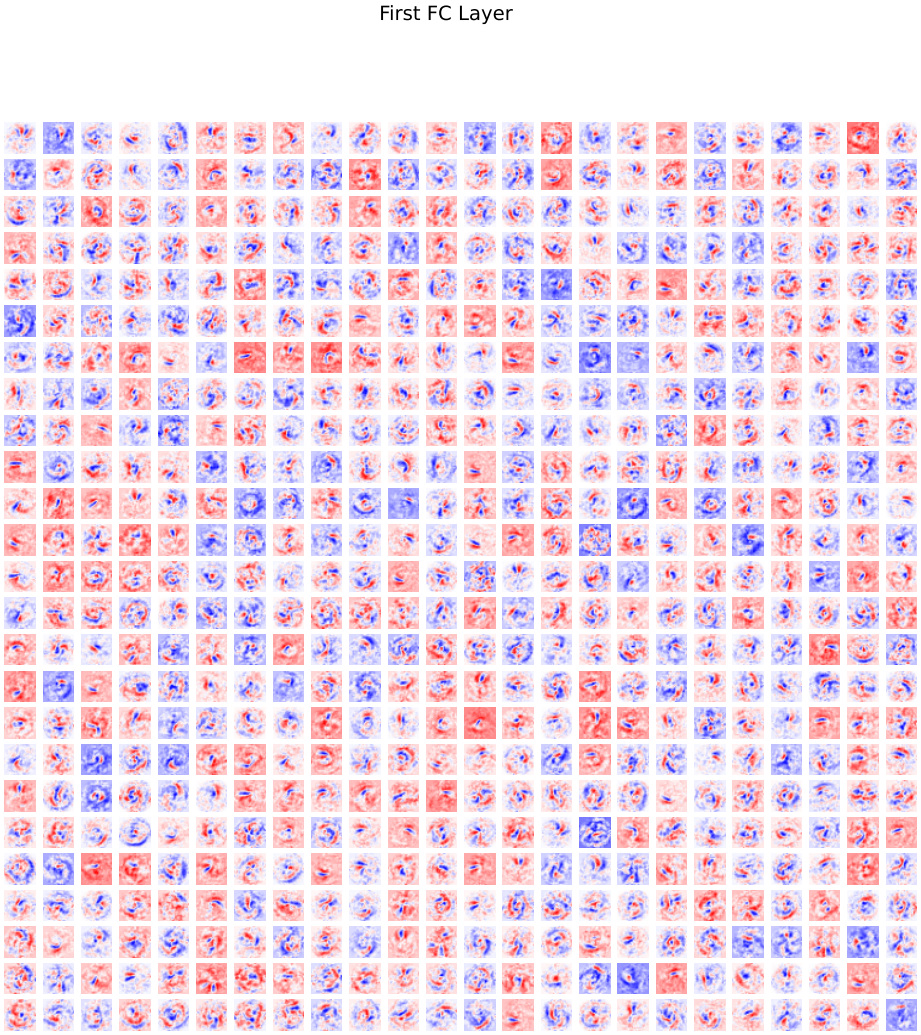

Figure 6: First FC layer of MLP network.

6.5.2 ResNet

For ResNet, I visualize the kernels of the first convolution layer in fig. 8. Also, I visualize one input and its output of the first convolution layer in fig. 7 and fig. 9. We can easily see how different kernels serve as different operators in extracting information on edges and brightness.

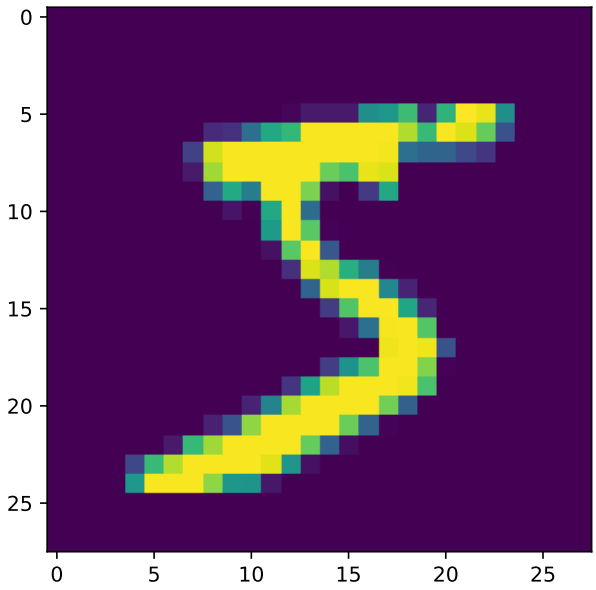

Figure 7: Sample Input

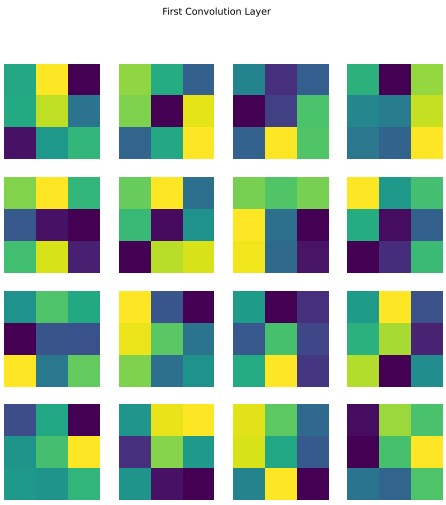

Figure 8: First convolution layer

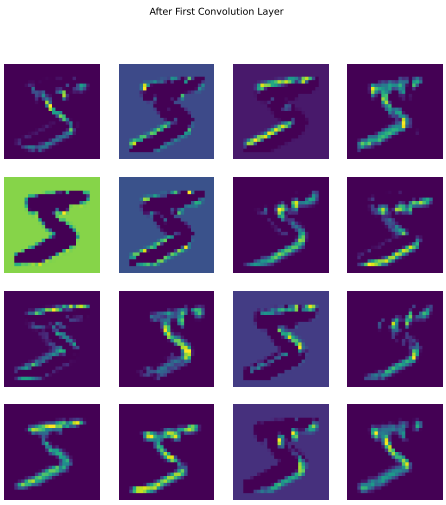

Figure 9: Output

Figure 10: Convolution kernel and output of first convolution layer.