UNIVERSITY OF MISSOURI – COLUMBIA

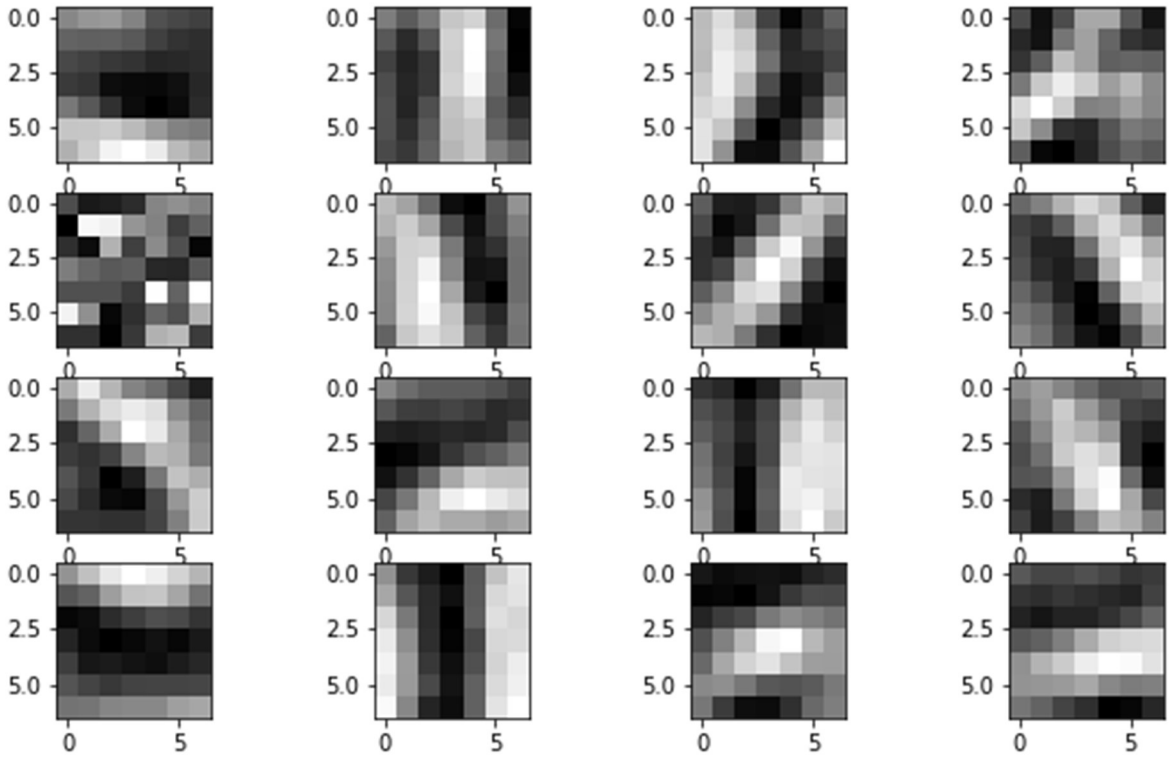
DEPARTMENT OF ENGINEERING

"A PROUD TRADITION, ENGINEERING THE FUTURE!"

ECE 8890: Neural Network – fall 2019

Project 1: Convolutional Neural Network (CNN)

Profs. James Keller and Derek Anderson



Waylon (Wenlong) Wu

March 19, 2019

Contents

I. Introduction

About ten years ago, the approaches to object recognition make essential use of traditional machine learning methods. ImageNet dataset [1] was released in 2010 as a benchmark dataset to test the performance of classifying algorithms. The ImageNet dataset is a huge dataset which consists of over 15 million labeled high-resolution images in over 22,000 categories. The remarkable breakthrough happened in 2012 that AlexNet [2] made much progress on the ImageNet. The AlexNet using convolutional neural network is developed on LeNet [3] and takes advantage of the power of graphics processing units (GPUs). Then the wave of the neural network and deep learning begins. Plenty of convolutional neural network architectures such as VGGNet, GoogLeNet, ResNet was proposed to achieve better and better performance on image classification. At the same time, artificial intelligence gains attention of people because of the success of the application of convolutional neural network such as face recognition in the industrial world. (I guess this is also the reason for the new opening of this neural network class.)

For the first project in this class, we focus on the Convolution Neural Network (CNN). We are given part of the MNIST dataset [2] which is a large database of handwritten digits. It contains 60,000 training images and 10,000 testing images. We are going to implement a backpropagation scheme [4] to learn the weights and bias in the neural network though the modern deep learning frameworks such as TensorFlow and PyTorch are easily accessible to us. We are going to open the box of CNN and see how it works.

II. Technical Description

II.A Fully Connect Layer (Multi-Layer Perceptron / MLP)

A typical neural network consists of many layers with many neurons in the same layer. The simplest neural network only has one input layer and one output layer, but it can be complicated by adding more hidden layers. Fig 1 is an example of feedforwards neural network with one hidden layer.

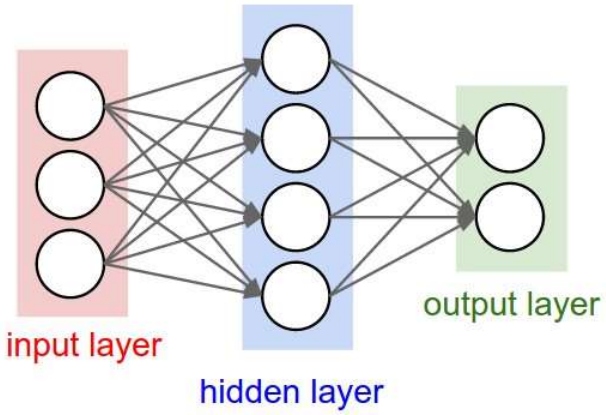


Fig 1. A 2-layer Neural Network (one hidden layer of 4 neurons and one output layer with 2 neurons)

(a.) Activation Function

A feedforward neural network performs a nonlinear input-output mapping. The mapping is called activation. There are 3 mainly used activation function: sigmoid, tanh and relu. Every activation function (or non-linearity) takes a single number and performs a certain fixed mathematical operation on it. The visualization of each activation function is shown in Fig 2.

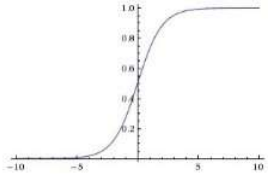
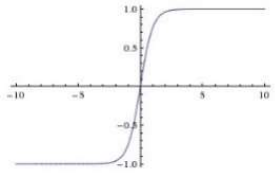
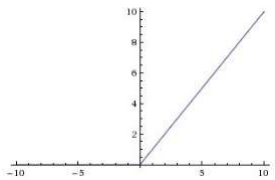
Sigmoid	Tanh	Relu
$\phi(v) = \frac{1}{1+e^{-v}}$	$\phi(v) = \frac{e^{v}-e^{-v}}{e^{v}+e^{-v}}$	$\phi(v) = f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$
		

Fig 2. Sigmoid, Tanh and Relu activation functions

- **Sigmoid**

The sigmoid non-linearity takes a real-valued number and "squashes" it into a range between 0 and 1. In particular, large negative numbers become 0 and large positive numbers become 1. The sigmoid function has been frequent used historically since it has a nice interpretation as the firing rate of a neuron. In practice, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used. It has two major drawbacks [5]:

1. Sigmoid saturate and kill gradients.
2. Sigmoid outputs are not zero-centered.

- **Tanh**

The tanh non-linearity squashes a real-valued number to the range [-1, 1]. Like the sigmoid, its activation saturates, but unlike the sigmoid neuron, its output is zero-centered. Therefore, in practice, the tanh non-linearity is always preferred to the sigmoid nonlinearity.

- **Relu**

The Rectified Linear Unit (ReLU) has become very popular in the last few years. It was found to greatly accelerate the convergence of stochastic gradient descent compared to the sigmoid/tanh functions [2]. Besides, the Relu can be implemented by simply thresholding a matrix of activations at zeros, thus can reduce the expensive computations. However, the relu units can be fragile during training can be "die". In some cases, a large gradient flowing through a relu activation function could cause the weights to update in such as way that the neuron will never activate on any data point again. There are some improved versions of relu function such as Leaky relu, which is designed to fix the "dying relu" problem.

(b.) Feedforward

A single neuron is shown in Fig 3. The input of neuron i is $[y_0, y_1(k), y_i(k), \dots, y_n(k)]^T$, where y_0 is the fixed input related to weight w_{j0}, which equals the bias b_j of the neuron j. the weight related to neuron j is $[w_{j0}, w_{j1}(\mathrm{k}), w_{ji}(\mathrm{k}), \dots, w_{jn}(\mathrm{k})]$.

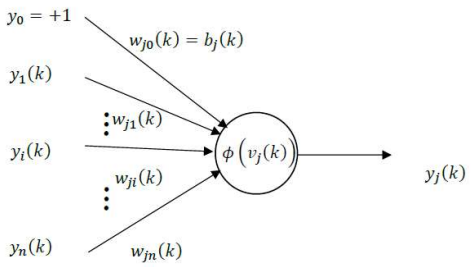


Fig 3. Neuron j in a typical neural network

The output of neuron is calculated as

$$y_j(j) = \phi_j(v_j(k)) \tag{1}$$

and

$$v_j(k) = \sum_{i=0}^{n} w_{ji}(k) y_i(k) \tag{2}$$

where $v_j(k)$ is the activation function of neuron j. It could be sigmoid, tanh or relu.

(c.) Backpropagation

Each neuron in the neural network has its own associated weights. The network has to be trained so that it will have good performance. Backpropagation is the main training algorithm nowadays. With backpropagation, weights are updated by gradient descent, which means weights of neurons are adjusted by calculating the gradient of the cost function. The cost function can also be considered as a loss function. In backpropagation, a common cost function is to calculate the difference between the expected label and the real network output.

$$e_j(k) = d_j(k) - y_j(k) \tag{3}$$

where $d_j(k)$ is the expected label and $y_j(k)$ is the actual output of neuron j. Given the training sample $\{x(k), d(k)\}$, the total error energy of the whole network is

$$E(k) = \sum_j E_j(k) = \frac{1}{2} e_j^2(k) \tag{4}$$

To minimize the total error energy, the backpropagation algorithm applies a $\Delta w_{ji}(k)$ to $w_{ji}(k)$. $\Delta w_{ji}(k)$ can be obtained by calculating the partial derivative $\frac{\partial E(k)}{\partial w_{ji}(k)}$. According to the chain rule,

$$\frac{\partial E(k)}{\partial w_{ji}(k)} = \frac{\partial E(k)}{\partial e_j(k)} \frac{\partial e_j(k)}{\partial y_j(k)} \frac{\partial y_j(k)}{\partial v_j(k)} \frac{\partial v_j(k)}{\partial w_{ji}(k)} = -e_j(k) \phi_j'(v_j(k)) y_j(k) \tag{5}$$

and

$$\Delta w_{ji}(k) = -\alpha \frac{\partial E(k)}{\partial w_{ji}(k)} = \alpha \delta_j(k) y_i(k) \tag{6}$$

where

$$\delta_j(k) = f(x) = \begin{cases} e_j(k) \phi_j'(v_j(k)), & \text{if neuron } j \text{ is an output neuron} \\ \phi_j'(v_j(k)) \sum_l \delta_l(k) w_{lj}(k), & \text{if neuron } j \text{ is a hidden neuron} \end{cases} \tag{7}$$

Neuron l is an output neuron connected to the hidden neuron j. And equation (6) updates weights based on the delta rule. The delta rule is a gradient descent learning rule with a backpropagation foundation, used to update the weights of artificial neurons in the neural network.

II.B Convolutional Layer

The purpose of the convolutional layer is to extract features from input data. For the CNN, inputs are usually images. Convolution maintains the relationship between pixels by learning features with small squares of input images. A general convolution neural network architecture is shown in Fig 4 [6].

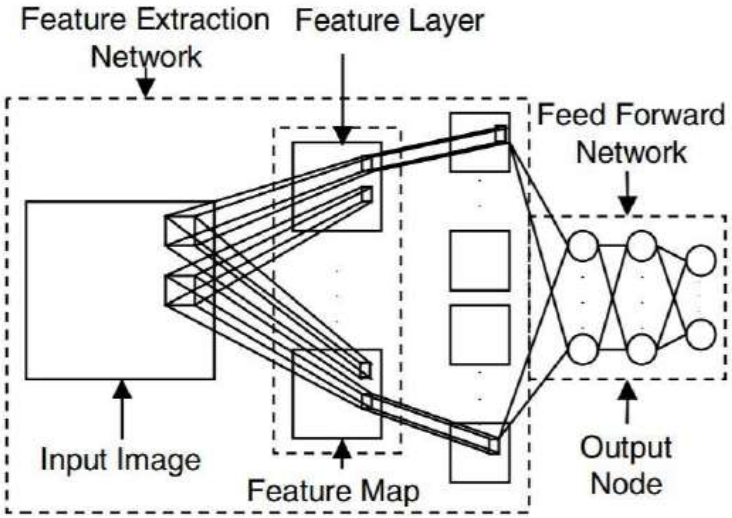


Fig 4. A general convolution neural network architecture

A feature map is obtained by convoluting the input image with a linear filter, adding a bias, and then applying a nonlinear function. Convolution is a linear operator. But in the real world, most data for CNNs to learn are nonlinear; thus, a nonlinear function (relu is usually used) is used to introduce nonlinearity to the network.

The Fig 5 from [7] shows an example of convolution example.

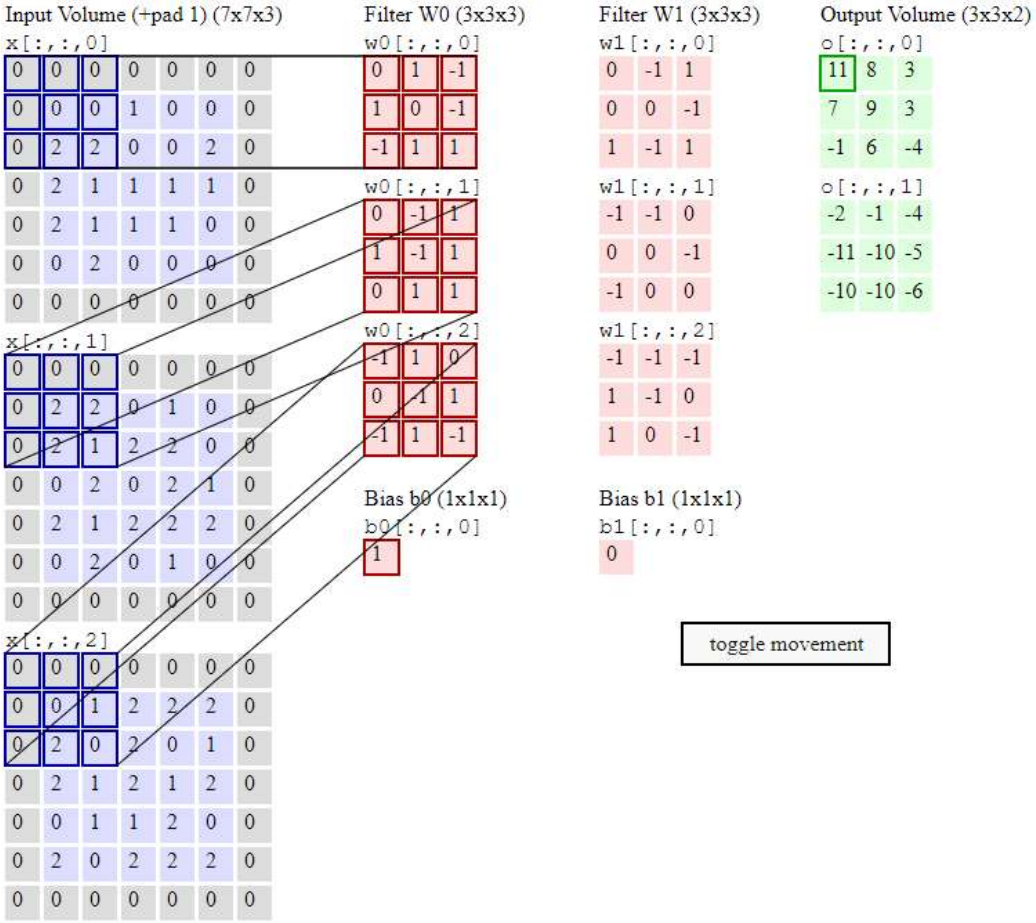


Fig 5. Convolution operation example

The most important and complicated thing in a convolution neural network is how to actually do the convolution. One way to do this is to leave the image a square and to implement the sliding window directly. However, this approach takes much computation and runs very slow. The dataset in part 3 has 6,000 images. I don't want to spend hours waiting for the algorithm converges. Another idea is to unroll the image into a vector according to the kernel size so that we can do matrix multiplication.

An example on how to unroll image is shown in Fig 6. The Kernel size is 2x2. For a gray image 5 x 5, each value in the kernel produces a features map. The final convolution result is the aggregation of all feature maps (it's a sum in convolution case). This idea comes from [9], which is what Dr. Popescu mentioned in class.

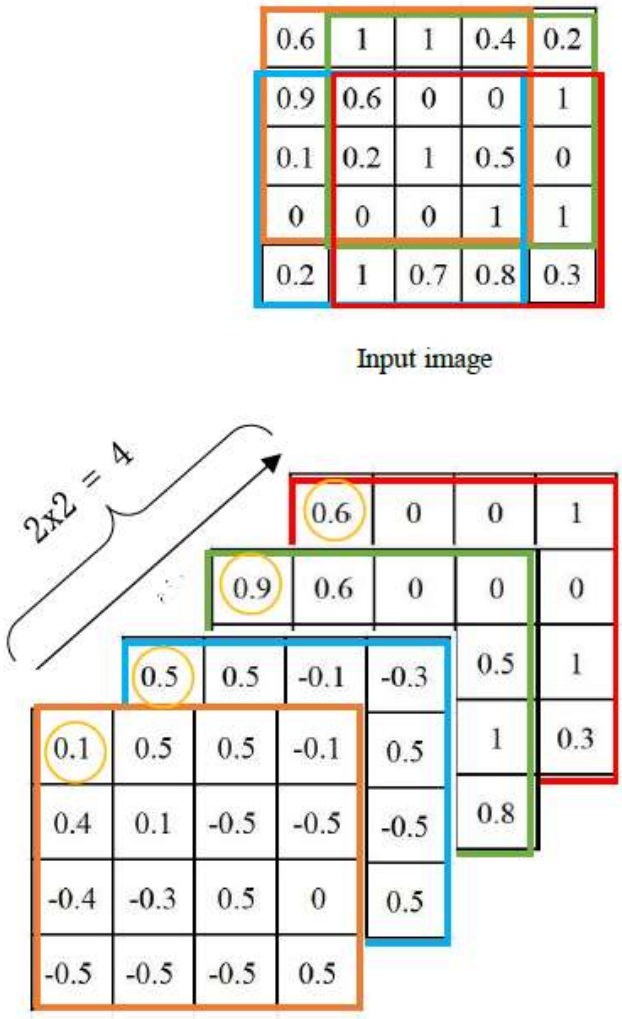


Fig 6. An example of how to unroll image

III. Code Description

1. Import libraries

```
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
from scipy.io import loadmat
```

2. Define activation functions

```
## define activation function
def sigmoid(x, derive=False):
    if derive:
        return x * (1 - x)
    return 1 / (1 + np.exp(-x))

def tanh(x, derive=False):
    if derive:
        return (1 - x) * (1 + x)
    return (np.exp(x) - np.exp(-x)) / (np.exp(x) + np.exp(-x))

def relu(x, derive=False):
    if derive:
        return 1. * (x > 0)
    return x * (x > 0)
```

3. Import data and formulating data

```
x = loadmat('Part2.mat')
digit_1_train = x['Matrix1_Train']
digit_3_train = x['Matrix3_Train']
digit_1_test = x['Matrix1_Test']
digit_3_test = x['Matrix3_Test']

# adding a bias term in the data
digit_1_train = np.concatenate((digit_1_train,np.ones([1, digit_1_train.shape[1]])), axis=0)
digit_3_train = np.concatenate((digit_3_train,np.ones([1, digit_3_train.shape[1]])), axis=0)
digit_1_test = np.concatenate((digit_1_test,np.ones([1, digit_1_test.shape[1]])), axis=0)
digit_3_test = np.concatenate((digit_3_test,np.ones([1, digit_3_test.shape[1]])), axis=0)

# making the label as one hot vector ([0, 1] or [1, 0])
digit 1 train label=np.concatenate((np.ones([1,digit 1 train.shape[1]]),np.zeros([1,digit 1 train.shape[1]])),axis=0)
digit 3 train label=np.concatenate((np.zeros([1,digit 3 train.shape[1]]),np.ones([1,digit 3 train.shape[1]])),axis=0)
digit_1_test_label=np.concatenate((np.ones([1,digit_1_test.shape[1]]),np.zeros([1,digit_1_test.shape[1]])),axis=0)
digit_3_test_label=np.concatenate((np.zeros([1,digit_3_test.shape[1]]),np.ones([1,digit_3_test.shape[1]])),axis=0)

# concatenating training set and label
x_train_1 = np.concatenate((digit_1_train,digit_1_train_label),axis=0)
x train 3 = np.concatenate((digit 3 train,digit 3 train label),axis=0)
x_train = np.concatenate((x_train_1, x_train_3),axis=1)

x_test_1 = np.concatenate((digit_1_test,digit_1_test_label),axis=0)
x_test_3 = np.concatenate((digit_3_test,digit_3_test_label),axis=0)
x_test = np.concatenate((x_test_1, x_test_3),axis=1)
```

```
Digit number 1 & 3 dataset x_train: train shape is (787, 34)
Digit number 1 & 3 dataset x_test: test shape is (787, 100)
```

34 is the number of training samples, 100 is the number of testing samples;

787 = 28 * 28 (image size) + 1(bias) + 2(one hot vector size / label size)

4. Define hyper-parameters and initialize the weights

```
# learning rate
eta = 0.05

# initialize weights with random numbers
np.random.seed(2019)
k1 w = np.random.normal(0,1,(28 * 28 + 1, 1))
k2_w = np.random.normal(0,1,(28 * 28 + 1, 1))

epoch = 100
```

```
err = np.zeros((epoch, 1))
inds = np.arange(0,x_train.shape[1],1)
```

5. Training the algorithm through backpropagation

```
for k in range(epoch):

    # init error
    err[k] = 0

    # random shuffle of data each epoch
    inds = np.random.permutation(inds)
    for i in range(x_train.shape[1]):

        # random index
        inx = inds[i]

        # forward pass
        v = np.ones((2, 1))
        v[0] = np.dot(x_train[:28*28+1,inx],k1_w)
        v[1] = np.dot(x train[:28*28+1,inx],k2 w)
        o = np.reshape(sigmoid(v),[1,2])

        # error
        err[k] = err[k] + np.sum(0.5 * (o - x_train[28*28+1:,inx]) ** 2)

        # backpropagation

        # output layer
        delta_1 = (-1.0) * (x_train[28*28+1:,inx] - o)
        delta_2 = sigmoid(o, derive = True)

        # hidden layer
        delta_hw = np.reshape(x_train[:28*28+1,inx],[-1,1]) * delta_1 * delta_2

        # update rule
        k1 w = k1 w + (-1.0) * eta * np.reshape(delta hw[:,0],[28*28+1,1])
        k2_w = k2_w + (-1.0) * eta * np.reshape(delta_hw[:,1],[28*28+1,1])
```

$$y_j(j) = \phi_j(v_j(k))$$
$$v_j(k) = \sum_{i=0}^{n} w_{ji}(k) y_i(k)$$

$$\mathrm{E(k)} = \sum_j E_j(k) = \frac{1}{2} e_j^2(k)$$

$$e_j(k) = d_j(k) - y_j(k)$$

$$\frac{\partial E(k)}{\partial w_{ji}(k)} = \frac{\partial E(k)}{\partial e_j(k)} \frac{\partial e_j(k)}{\partial y_j(k)} \frac{\partial y_j(k)}{\partial v_j(k)} \frac{\partial v_j(k)}{\partial w_{ji}(k)}$$
$$= -e_j(k) \phi_j'(v_j(k)) y_j(k)$$

6. Evaluating the training result

```
# Look at the training data: see how many training data is labeled correct
train_acc = 0
for i in range(x_train.shape[1]):

    # forward pass
    v = np.ones((2, 1))
    v[0] = np.dot(x train[:28*28+1,i],k1 w)
    v[1] = np.dot(x train[:28*28+1,i],k2 w)
    o = np.reshape(sigmoid(v),[1,2])
    oo = np.reshape(np.zeros([1,2]),[2,])
    oo[np.argmax(o)] = 1

    #print(str(i) + ": produced: " + str(o) + " wanted " + str(x train[28*28+1:,i]))
    if(np.array equal(x train[28*28+1:,i],oo)):
        train_acc = train_acc + 1

print(str(train_acc) + ' out of ' + str(x_train.shape[1]) + ' in the training set was labeled correctly')

# Look at the testing data: see how many testing data is labeled correct
test_acc = 0
for i in range(x test.shape[1]):

    # forward pass
    v = np.ones((2, 1))
    v[0] = np.dot(x_test[:28*28+1,i],k1_w)
    v[1] = np.dot(x test[:28*28+1,i],k2 w)
    o = np.reshape(sigmoid(v),[1,2])

    oo = np.reshape(np.zeros([1,2]),[2,])
    oo[np.argmax(o)] = 1
    #print(str(i) + ": produced: " + str(oo) + " wanted " + str(x_test[28*28+1:,i]))
    if(np.array_equal(x_test[28*28+1:,i],oo)):
        test_acc = test_acc + 1

print(str(test acc) + ' out of ' + str(x test.shape[1]) + ' in the testing set was labeled correctly')
```

The part 3 code is partially similar to the part 2 code. **Only the below part differs:**

3. Formulating data

```
# produce one hot vector for ten classes
label = np.eye(10).reshape((10,10,1)) # ten digits, each one_hot vector
train_label = np.repeat(label, repeats= TrainData.shape[2], axis=2) # 10 x 10 x 20
test_label = np.repeat(label, repeats= TestData.shape[2], axis=2)
...
```

```
Train data with bias, adding label, after reshaping shape is: (795, 5000)
Test data with bias, adding label, after reshaping shape is: (795, 5000)
```

5,000 is the number of training/testing samples;

795 = 28 * 28 (image size) + 1(bias) + 10(one hot vector size / label size)

5. Initialize the weights

```
kernel_size = 7
kernel_num = 16
k_w = np.random.normal(0,1,(kernel_size * kernel_size + 1, kernel_num)) * 0.01
h1_w = np.random.normal(0,1,((28 - kernel_size + 1)*(28 - kernel_size + 1)*(kernel_num) + 1, 10)) * 0.01
```

```
16 kernel weight shape is: (50, 16)
First fully connected layer weight shape is: (7745, 10)
```

6. Training the algorithm through backpropagation

```
for k in range(epoch):

    # init error
    err[k] = 0

    # random shuffle of data each epoch
    inds = np.random.permutation(inds)
    for i in range(TrainData.shape[1]):

        # random index
        inx = inds[i]

        # one image
        img = np.reshape(TrainData[:28*28, inx], [28, 28])

        # forwards pass
        # transform matrix C
        C = np.ones((28 - kernel size + 1, 28 - kernel size + 1, kernel size * kernel size))

        for m in range(kernel_size):
            for n in range(kernel_size):
                C[:,:,m * kernel_size + n] = img[m:m+(28 - kernel_size + 1),n:n+(28 - kernel_size + 1)]
        C=np.concatenate((C, np.ones([28 - kernel_size + 1, 28 - kernel_size + 1,1])),axis=2)

        h = relu(np.dot(C, k w)) # 22 x 22 x 16

        h = np.reshape(h, [(28 - kernel_size + 1)*(28 - kernel_size + 1)*(kernel_num), 1])
        h = np.concatenate((h,np.asarray([[1]])),axis=0)
        o = sigmoid(np.dot(h.T, h1_w))

        # MSE error
        #err[k] = err[k] + 0.5 * np.sum((o - TrainData[785:,inx]) ** 2)

        # cross entropy
        err[k] = err[k] + (-1.0) * np.sum(TrainData[785:,inx] * np.log(o + 10 ** -8))
        # backpropagation

        # output layer
        delta_1 = (-1.0) * (TrainData[785:,inx] - o) # (1, 10)
        #delta_2 = sigmoid(o, derive=True) # (1, 10), use if MSE
        delta_2 = np.ones_like(delta_1) # cross entropy

        # update the fully connected hidden layer
        delta hw = np.dot(h, delta 1 * delta 2)  # (22 * 22 * 16 + 1, 10)

        # update the convolutional kernel
        delta_3 = relu(h, derive=True) # (22 * 22 * 16 + 1, 1)
        delta_kw_tmp  =  np.sum(np.dot(delta_3,  delta_1  *  delta_2)  *  h1_w,axis=1)[:-1].reshape([(28  -
kernel_size + 1),(28 - kernel_size + 1),kernel_num])
```

$$y_j(j) = \phi_j(v_j(k))$$
$$v_j(k) = \sum_{i=0}^{n} w_{ji}(k) y_i(k)$$

$$\mathrm{E(k)} = \sum_j E_j(k) = \frac{1}{2} e_j^2(k)$$

$$e_j(k) = d_j(k) - y_j(k)$$

```
        delta_kw  =  np.dot(np.transpose(C,axes=[2,0,1]).reshape([k_w.shape[0],-1]),  delta_kw_tmp.reshape([-
1,kernel_num]))

        # mini-batch tmp value
        h1_w_tmp = h1_w_tmp + (-1.0) * eta * delta_hw # (22 * 22 * 16 + 1, 10)
        k w tmp = k w tmp + (-1.0) * eta * delta kw # (7 * 7 + 1, 16)

        # update rule
        if((k * TrainData.shape[1] + i) % mini_batch == 0):
            h1_w = h1_w + h1_w_tmp / mini_batch
            k w = k w + k w tmp / mini batch
            h1 w tmp = np.zeros like(h1 w tmp)
            k w tmp = np.zeros like(k w tmp)
    print('The ' + str(k+1) + ' epoch error is: ' + str(err[k]))
```

IV. Experiments and Results

IV.A Part 2

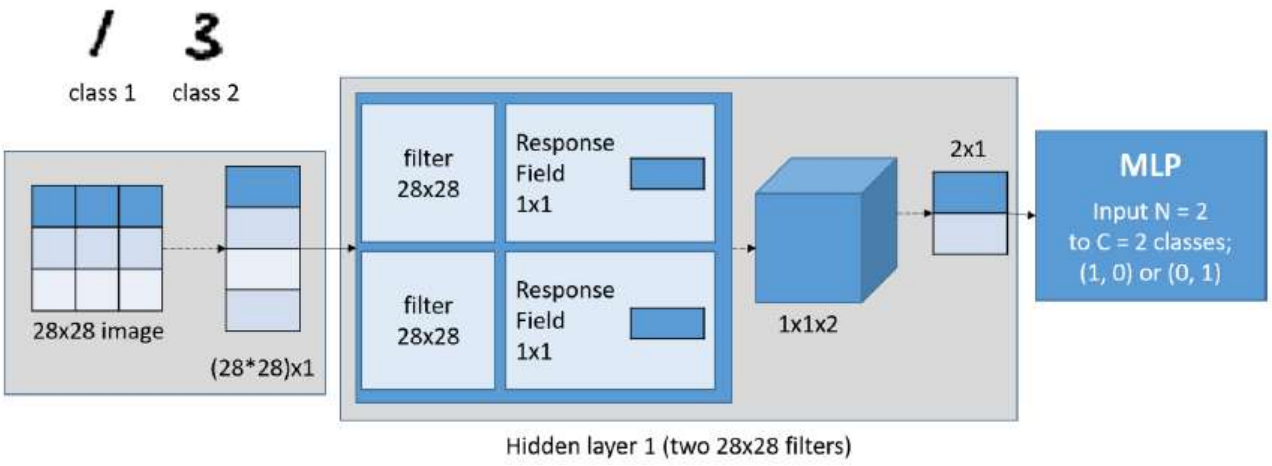


Fig 7. Two classification problem in part 2

This architecture shown in Fig 7 is actually a fully connected layer instead of convolutional neural network.

Options in this part:

- Data presentation (serial, batch, mini-batch)
- Order (fixed or shuffle data at each epoch)
- Network selection (activation functions)
- Initialization
- Learning rate(s)

Before we testing these different combinations, let's make a baseline prototype so that we can compare.

The hyperparameters of **the baseline prototype** in the part to are:

- Data presentation: serial
- Order: shuffle
- Network selection: sigmoid
- Initialization: random number between 0 and 1
- Learning rate: 0.05
- Epoch: 100
- Update method: Stochastic Gradient Descent (SGD)
- Loss function: Mean Squared Error(MSE)

The error plot function of the baseline prototype is shown in Fig 8.

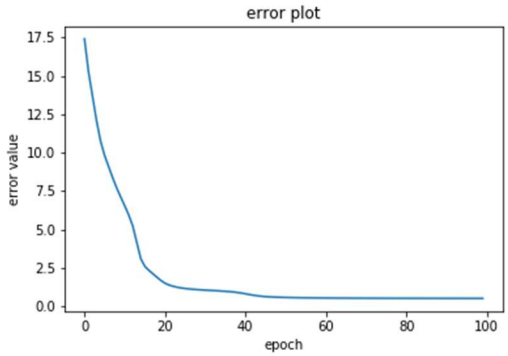
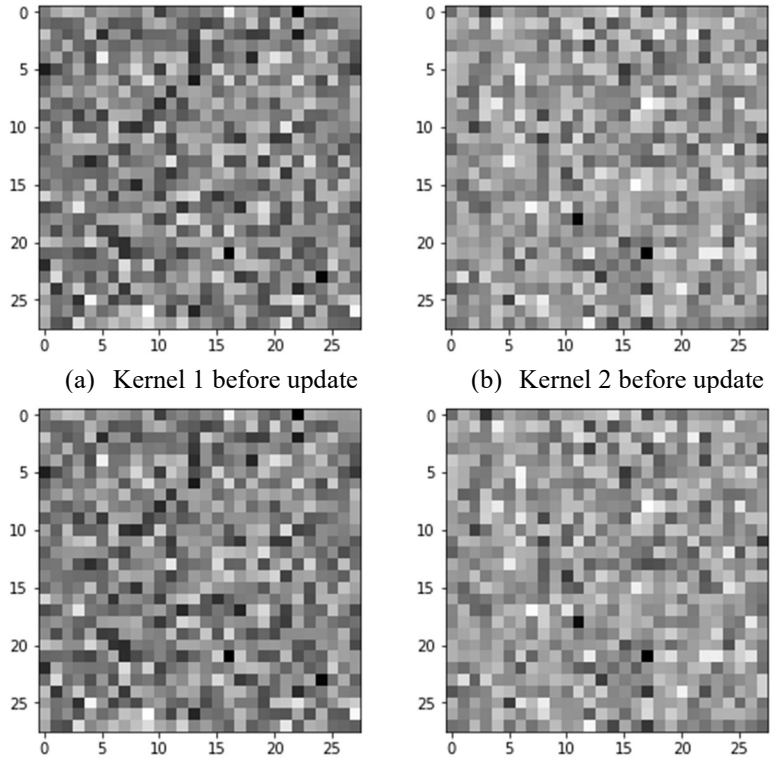


Fig 8. The error plot of the baseline prototype

The kernel visualization is shown in Fig 9.



(a) Kernel 1 before update (b) Kernel 2 before update

(c) Kernel 1 after update (d) Kernel 2 after update

Fig 9. Kernel visualization before and after backpropagation update

As we can see in the Fig 9, the kernels seem like unchanged and they don't make much sense to me. Perhaps this kernel is too big to capture the information.

Then we tested the final weights on both training set and testing set:

```
33 out of 34 in the training set was labeled correctly (33/34)
77 out of 100 in the testing set was labeled correctly (77/100)
```

Above result is set to be the result of the baseline in part 2. Then we try different hyper-parameters to see their performance.

(1) Data representation (serial, batch, mini-batch)

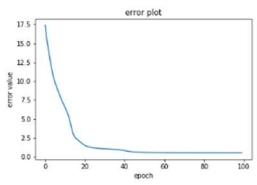
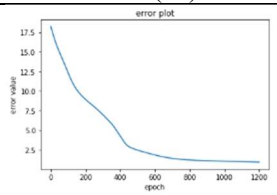
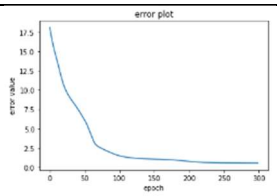
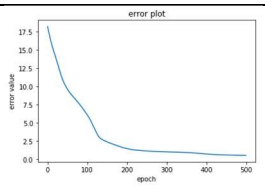
Serial (baseline)	Batch (34)	Mini-batch size 5	Mini-batch size 10
 Epoch: 100	 Epoch: 1200	 Epoch: 300	 Epoch: 500
Training: 33/34 Testing: 77/100	Training: 33/34 Testing: 75/100	Training: 33/34 Testing: 77/100	Training: 33/34 Testing: 76/100

Fig 10. Different data representation performance

As we can see in the Fig 10, the training and testing performance does not improve as we increase the batch size. Besides, we have to increase the number of epoch to make sure the algorithm converges. So for this small dataset in part 2, I think "serial" data representation is enough for us to learn.

(2) Order (fixed or shuffle data at each epoch)

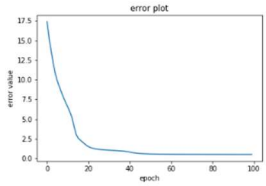
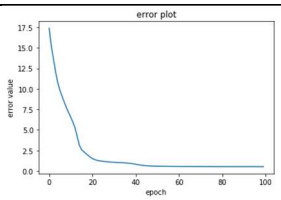
Random shuffle (baseline)	Fixed
 Epoch: 100	 Epoch: 100
Training: 33/34 Testing: 77/100	Training: 33/34 Testing: 77/100

Fig 11. The influence of order (shuffle and fixed)

The order doesn't seem to influence the performance. I guess it's because we have a small dataset so the influence doesn't show up. Let's wait to see the influence in part 3 when we have a larger dataset.

(3) Network selection (activation functions)

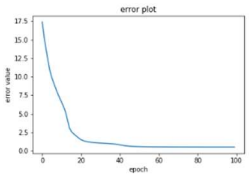
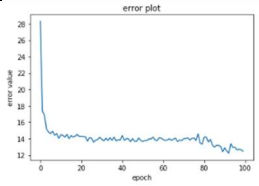
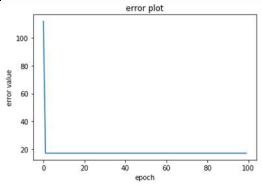
Sigmoid (baseline)	Tanh	Relu
 Epoch: 100	 Epoch: 100	 Epoch: 100
Training: 33/34 Testing: 77/100	Training: 23/34 Testing: 58/100	Training: 21/34 Testing: 55/100

Fig 12. The influence of activation functions

As seen in Fig 12, the sigmoid activation performs the best. However, tanh and relu activation functions converge faster than sigmoid though they do not achieve good performance as sigmoid did. I guess the reason causing this is that we use one hot vector ([0, 1] or [1, 0]) to represent output value which is close to the output range of sigmoid function (between 0 and 1). The output range of tanh is between -1 and 1. The negative value in the tanh function seems not to make sense in this case.

(4) Initialization

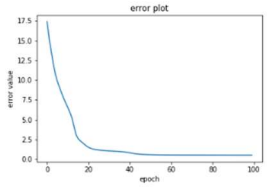
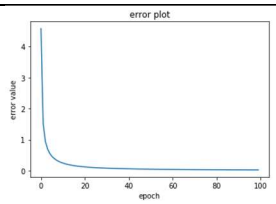
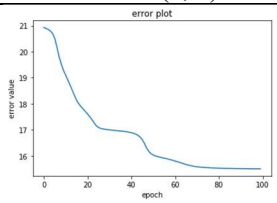
Random (0, 1) (baseline)	All 0	Random (0, 5)
		
Epoch: 100	Epoch: 100	Epoch: 100
Training: 33/34 Testing: 77/100	Training: 34/34 Testing: 81/100	Training: 15/34 Testing: 45/100

Fig 13. The influence of initialization

All 0 initialization performs better than random initialization between 0 and 1, which surprises me a little. If the weights are initialized as 0, then each weights vector will do the same thing. In the case of part 2, we only have two kernels. When I look at the kernel values, I found *kernel_1_weights* = - *kernel_2_weights*, which means they were doing the opposite thing. However, when the number of kernels increases, I guess all zero initialization is not going to work. So for now I'll stick to the random initialization between 0 and 1 and wait to see what all zero initialization performs in part 3.

(5) Learning rate(s)

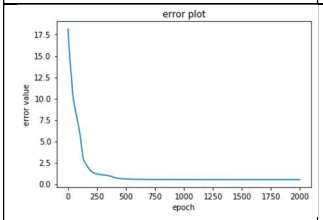
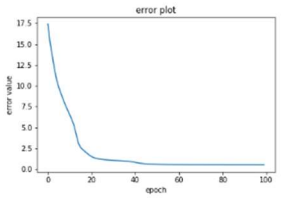
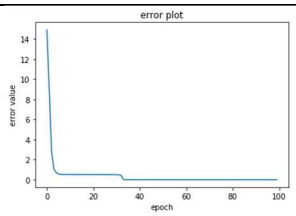
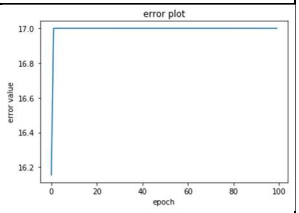
0.005	0.05 (baseline)	0.5	20
			
Epoch: 2,000	Epoch: 100	Epoch: 100	Epoch: 100
Training: 33/34 Testing: 79/100	Training: 33/34 Testing: 77/100	Training: 34/34 Testing: 82/100	Training: 17/34 Testing: 50/100

Fig 14. The influence of learning rate

Setting a suitable learning rate is usually hard. Setting it too small will take many epochs to converge. Setting it too big will lead the algorithm to be unable to learn anything. When the learning rate is 0.05, the error curve converges at the epoch of 50. However, it will have another drop-off after many epochs, just as shown in the case of learning rate = 0.5 (epoch 35). So sometimes we need some practice to select the best learning rate.

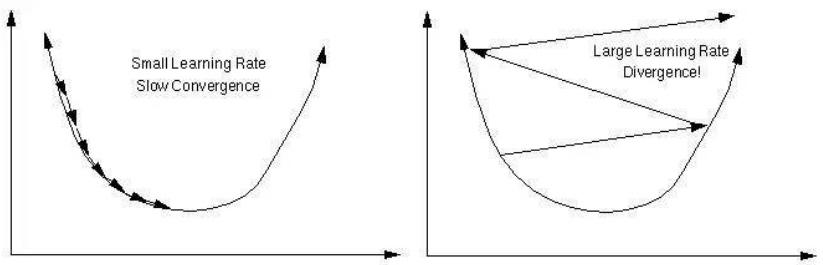


Fig 15. Small and large learning rate influence

IV.B Part 3 – structure 1 (conv-10)

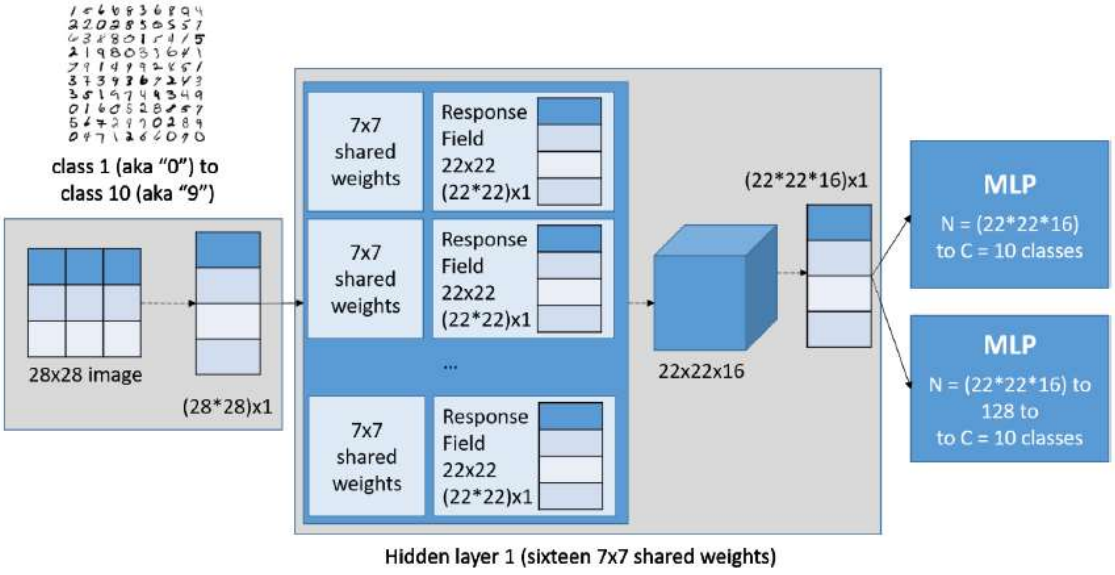


Fig 16. The convolutional architecture in part 3

Options in this part:

- Loss function & output layer function
- Data presentation (serial, batch, mini-batch)
- Order (fixed or shuffle data at each epoch)
- Network selection (activation functions)
- Initialization
- Learning rate(s)

Before we testing these different combinations, let's make a baseline prototype so that we can compare.

The hyperparameters of the baseline prototype in the part are:

- Data presentation: serial
- Order: shuffle
- Network selection: relu (hidden layer), softmax (output)
- Initialization: random number between 0 and 0.01
- Learning rate: 0.01
- Update method: Stochastic Gradient Descent (SGD)
- Loss function: Cross entropy

The error plot function of the baseline prototype is shown in Fig 17.

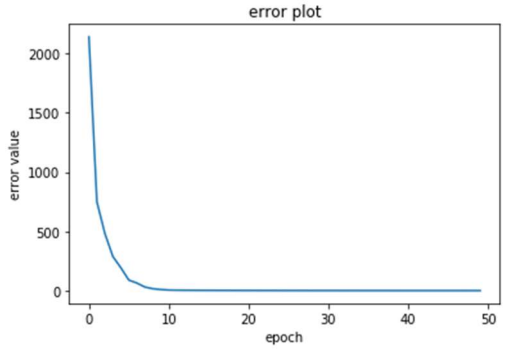
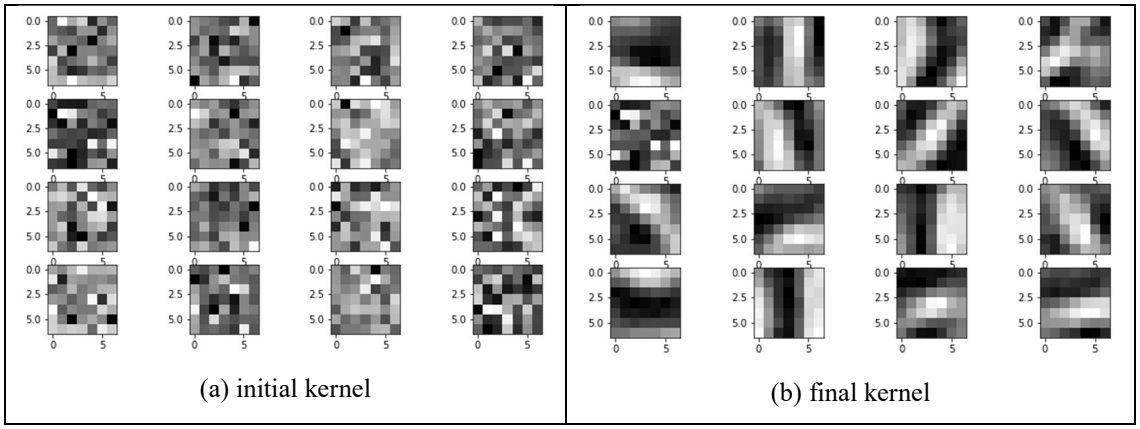


Fig 17. The error plot of the baseline prototype

We tested the final weights on both training set and testing set:

```
5000 out of 5000 in the training set was labeled correctly (5000/5000)
4821 out of 5000 in the testing set was labeled correctly (4821/5000)
```

The kernel visualization is shown in Fig 18.



(a) initial kernel (b) final kernel

Fig 18. Initial and final kernel visualization

Above result is set to be the result of the baseline in part 3 structure 1. Then we try different hyper-parameters to see their performance.

(1) Loss function & output function

In part two, we use the sigmoid activation function as an output layer function because it is a two-class classification problem. In this part, the problem is a multi-class classification problem. Usually, cross entropy loss and softmax function are used together in this case. The main problem of MSE and sigmoid is that it suffers from the gradient vanishing problem. The gradient of the sigmoid function gets much smaller as the algorithm learns. One way to deal with this problem is to set a threshold to the gradient, which is called "**gradient clipping**". When the gradient is smaller than a threshold value, set the gradient as that threshold value.

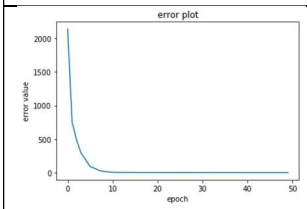
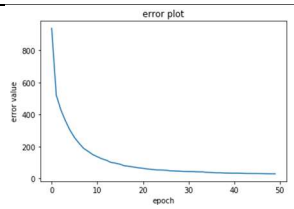
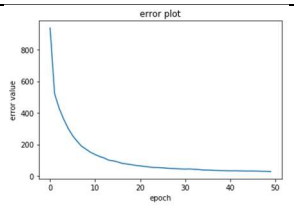
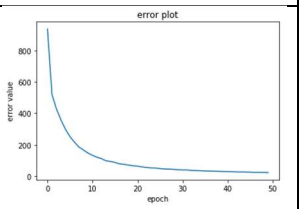
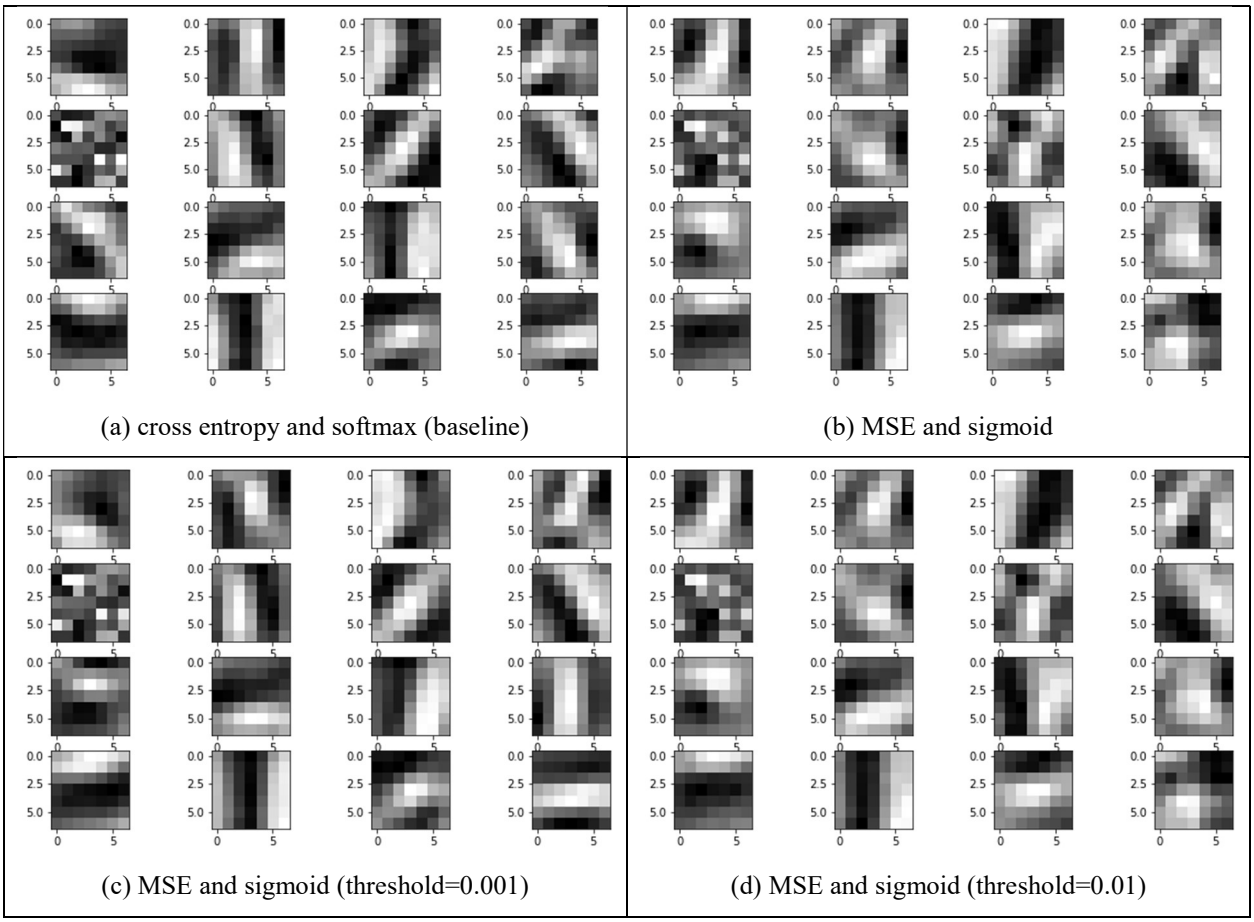
Cross entropy + softmax (baseline)	MSE + sigmoid	MSE + sigmoid (threshold=0.001)	MSE + sigmoid (threshold=0.01)
			
Epoch: 50	Epoch: 50	Epoch: 50	Epoch: 50
Training: 5000/5000 Testing: 4821/5000	Training: 4954/5000 Testing: 4802/5000	Training: 4956/5000 Testing: 4805/5000	Training: 4970/5000 Testing: 4831/5000

Fig 19. The influence of loss function and output function

As we can see in Fig 19, the combination of cross entropy and softmax performs better than the combination of MSE and sigmoid. What's more, the combination of cross entropy and softmax converges (at epoch 10) faster than the combination of MSE and sigmoid (at epoch 40). The combination of MSE and sigmoid with gradient clipping performs better than that without gradient clipping. But the clipping value also needs to be tuned, which adds the

complexity of the algorithm. So I think the combination of cross entropy and softmax is more suitable for multi-object classification problem. The kernel visualization in each case is shown in Fig 20.



(a) cross entropy and softmax (baseline)

(b) MSE and sigmoid

(c) MSE and sigmoid (threshold=0.001)

(d) MSE and sigmoid (threshold=0.01)

Fig 20. The kernel visualization for different loss function and output layer activation function

In Fig 20, they all look to capture some structure information in the image.

(2) Data representation (serial, batch, mini-batch)

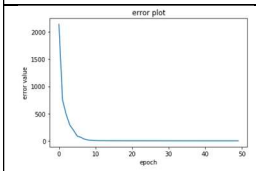
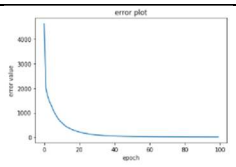
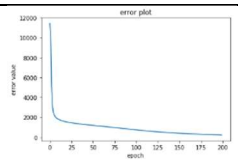
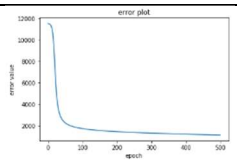
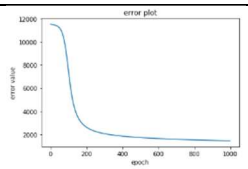
Serial (baseline)	Mini-batch size 10	Mini-batch size 100	Mini-batch size 1000	Batch (5000)
 Epoch: 50	 Epoch: 100	 Epoch: 300	 Epoch: 500	 Epoch: 1000
Training: 5000/5000 Testing: 4821/5000	Training: 5000/5000 Testing: 4794/5000	Training: 4956/5000 Testing: 4737/5000	Training: 4684/5000 Testing: 4537/5000	Training: 4593/5000 Testing: 4507/5000

Fig 21. The influence of data presentation

Using the mini-batch method can help the training converge faster. However, the performance of the serial method is still the best. The serial method, which is actually the “online learning”, is to update weights on every single sample. If the dataset has some noisy points, then it will update the weights in the wrong direction. As we can see in Fig 21, the curve of the serial method is not smooth as the curve of the mini-batch method. The reason why the serial method

achieves the best score in this case I guess is because the dataset in this part is very separable that does not have many noises. The mini-batch method is often used with parallel computing to speed up the learning of the algorithm. The minibatch size needs also to be tuned to achieve good results. The kernel visualization in each case is shown in Fig 22.

(a) serial (baseline)

(b) mini batch size 10

(c) mini batch size 100

(d) mini batch size 1000

(e) batch (5000)

Fig 22. The kernel visualization for different data representation

In Fig. 22, as the mini batch size grows, the performance of kernels seems to be less efficient.

(3) Order (fixed or shuffle data at each epoch)

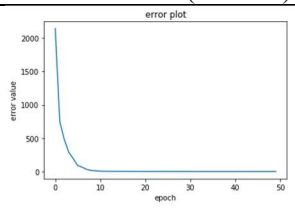
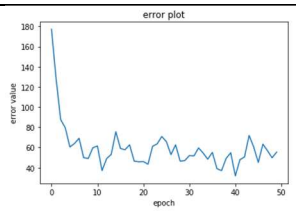
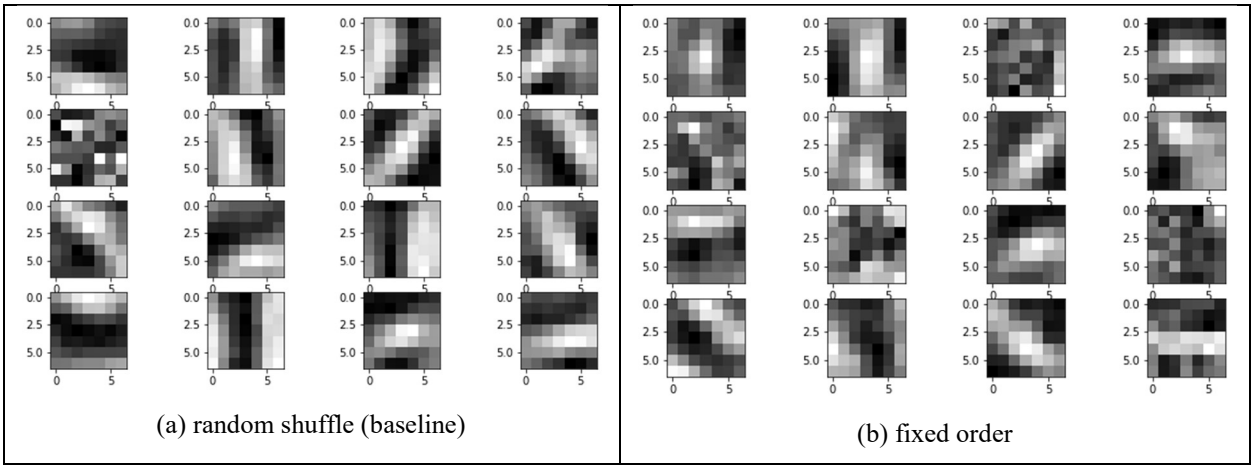
Random shuffle (baseline)	Fixed
	
Epoch: 50	Epoch: 50
Training: 5000/5000 Testing: 4821/5000	Training: 3571/5000 Testing: 3411/5000

Fig 23. The influence of order

The random shuffle method wins with no doubt in this experiment. The fixed order method converges slower and achieves lower score than the random shuffle method. So we should random shuffle the data on each epoch during training. The kernel visualization in each case is shown in Fig 24.



(a) random shuffle (baseline)

(b) fixed order

Fig 24. The kernel visualization for different order

(4) Network selection (activation functions)

Since the output value should be one hot vector, the softmax function performs the best in the output layer, as discussed above. So we will change different activation functions in the hidden layer and see their performances.

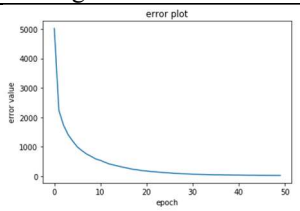
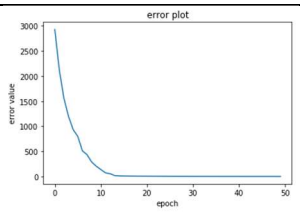
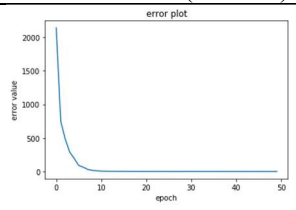
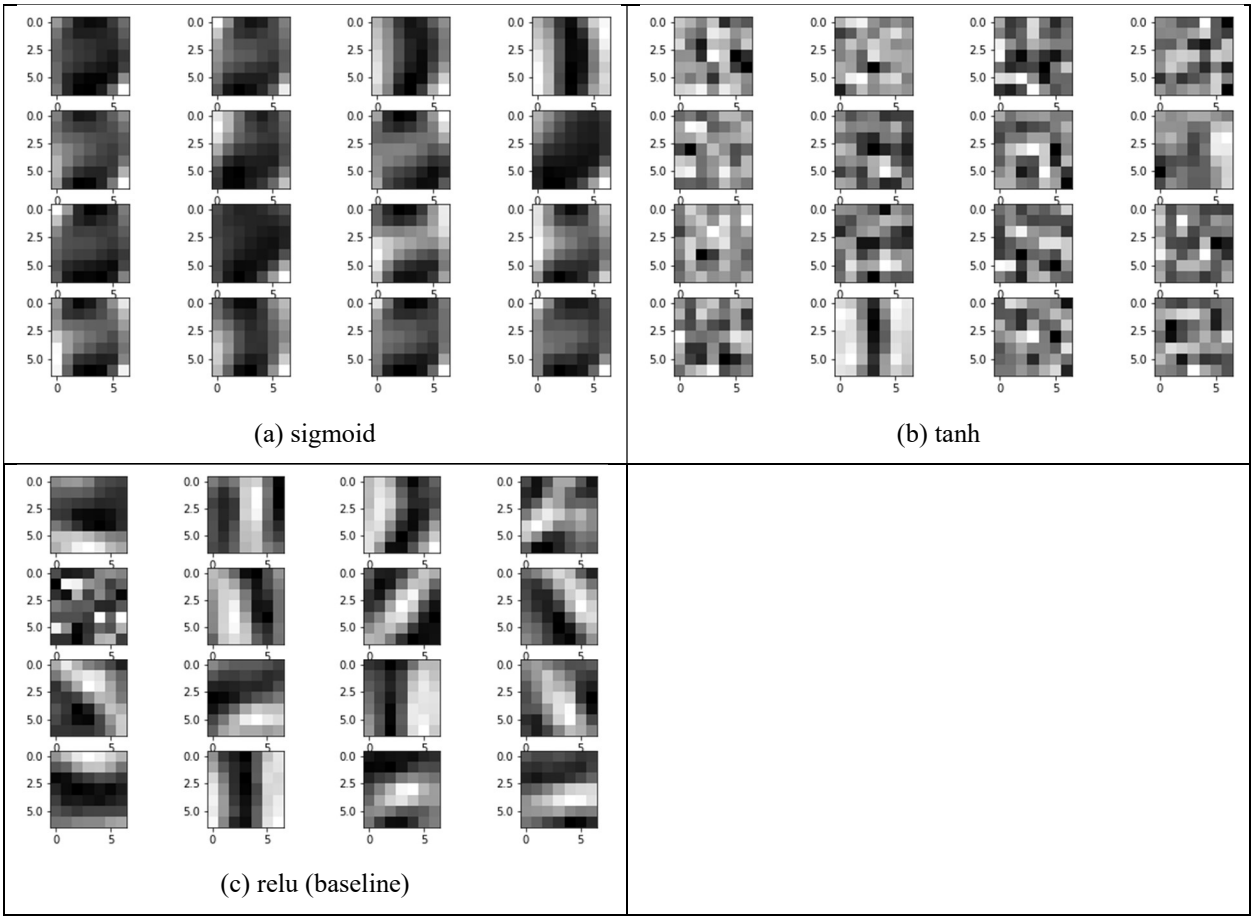
Sigmoid + softmax	Tanh + softmax	Relu + softmax (baseline)
		
Epoch: 50	Epoch: 50	Epoch: 50
Training: 4999/5000 Testing: 4770/5000	Training: 5000/5000 Testing: 4694/5000	Training: 5000/5000 Testing: 4821/5000

Fig 25. The influence of activation function selection

The relu activation function outperforms than the sigmoid and tanh activation function obviously. The sigmoid function converges slower than the other two. The relu function is the best activation function in hidden layer in this

case, simple but effective. The kernel visualization in each case is shown in Fig 26.



(a) sigmoid

(b) tanh

(c) relu (baseline)

Fig 26. The kernel visualization for the different activation function

In Fig 26, the kernel visualization of relu activation function is still the best. Sigmoid activation function has fewer firings. Tanh activation function does not even capture the structure information in the image.

(5) Initialization

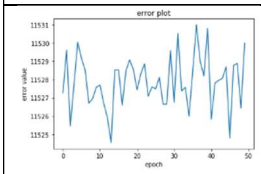
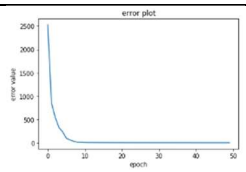
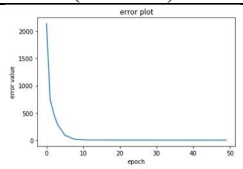
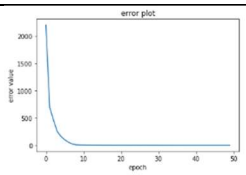
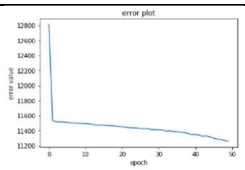
All 0	Random (0, 0.001)	Random (0, 0.01) (baseline)	Random (0, 0.1)	Random (0, 1)
 Epoch: 50	 Epoch: 50	 Epoch: 50	 Epoch: 50	 Epoch: 50
Training: 500/5000 Testing: 500/5000	Training: 5000/5000 Testing: 4820/5000	Training: 5000/5000 Testing: 4821/5000	Training: 5000/5000 Testing: 4799/5000	Training: 790/5000 Testing: 713/5000

Fig 27. The influence of initialization

Although all zero method achieves the best score in part 2, it fails completely in this part. As I said in part 2, all the 16 kernels in this part do the same thing so they have limited ability to separate different classes. The random method (0, 01) is better than the others. So initializing the weights as small values is still important. The kernel visualization

in each case is shown in Fig 28.

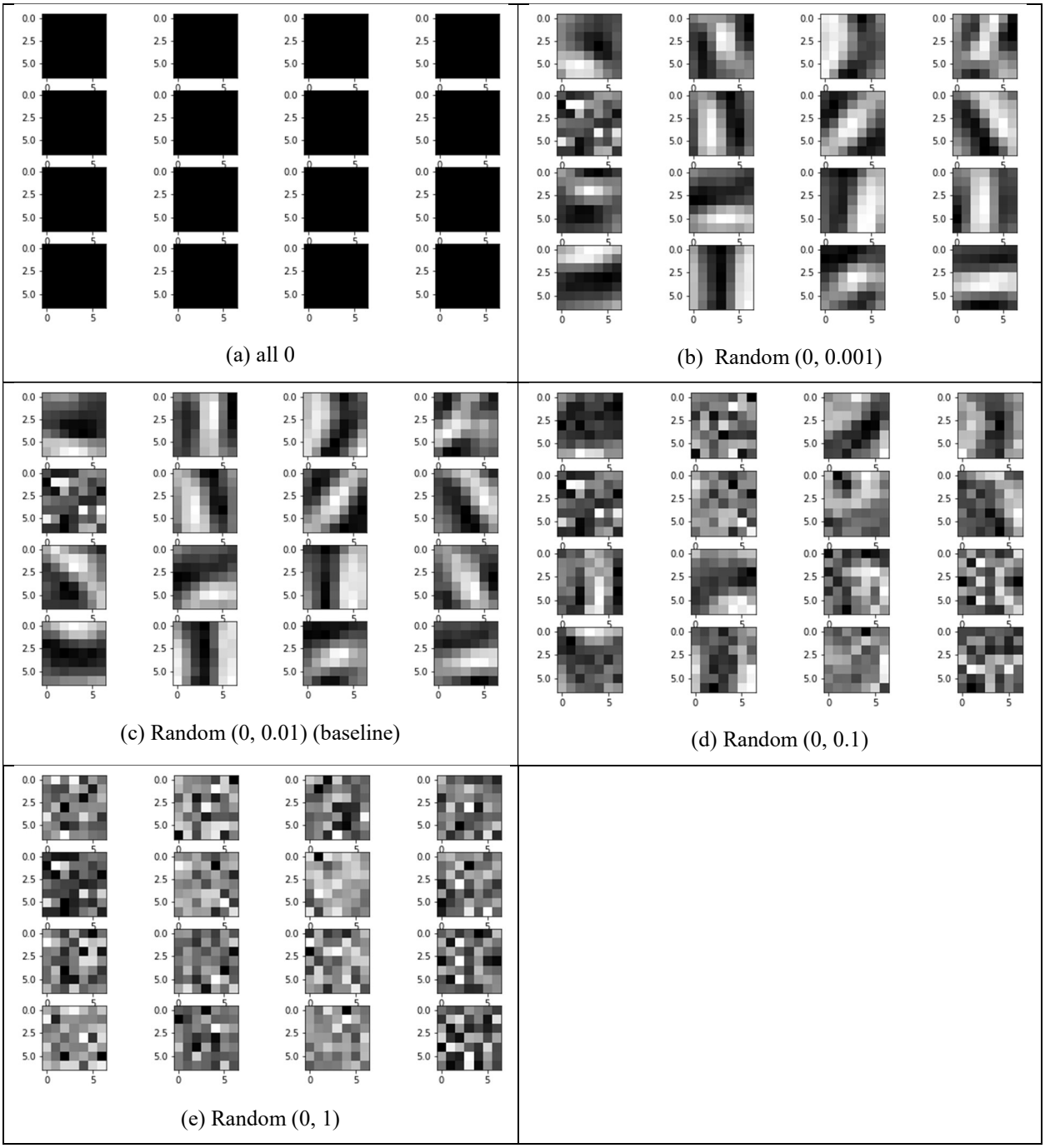


Fig 28. The kernel visualization for different initialization

In Fig 28, Random (0, 0.1) and random (0, 1) fail to capture the structure information in the image. So initializing the weights as small values can help the kernel easily learn the structure information.

(6) Learning rate(s)

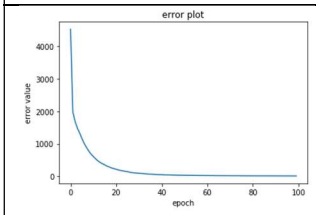
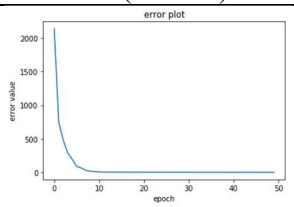
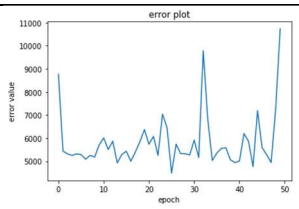
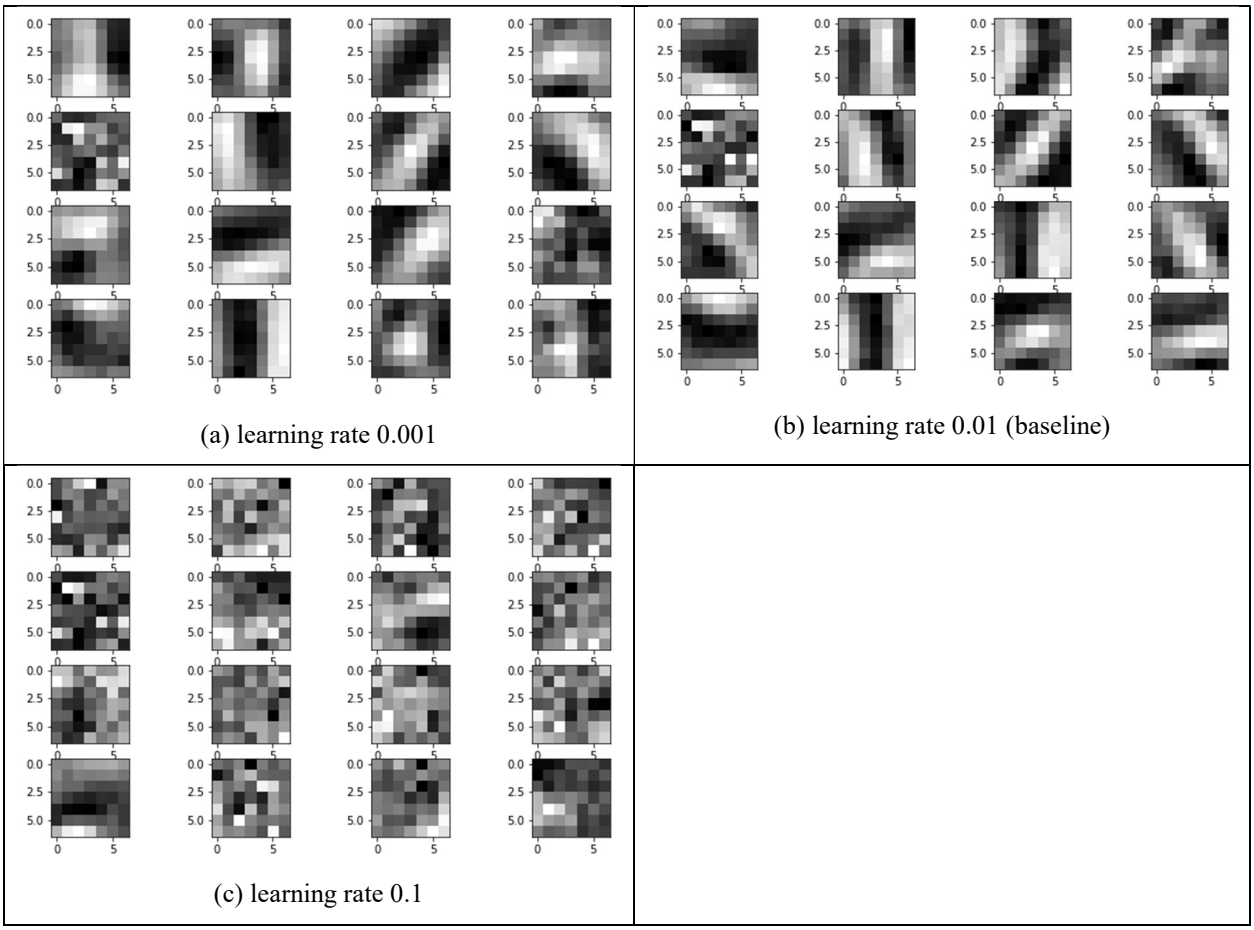
0.001	0.01 (baseline)	0.1
		
Epoch: 100	Epoch: 50	Epoch: 50
Training: 5000/5000 Testing: 4793/5000	Training: 5000/5000 Testing: 4821/5000	Training: 3489/5000 Testing: 2936/5000

Fig 29. The influence of learning rate

The learning rate is the most important influential parameter when tuning the algorithm. When the learning rate is too small, it will take many epochs to converge. On the other hand, when the learning rate is too big, it will not converge at all. In this experiment, the learning rate 0.01 has the best performance. The kernel visualization in each case is shown in Fig 30.



(a) learning rate 0.001

(b) learning rate 0.01 (baseline)

(c) learning rate 0.1

Fig 30. The kernel visualization for different learning rate

In Fig 30, learning rate 0.1 seems to be too large to capture the structure information in the image.

IV.C Part 3 – structure 2 (conv-128-10)

Options in this part:

- Data presentation (serial, batch, mini-batch)
- ~~Order (fixed or shuffle data at each epoch)~~ => shuffle is proven to be better
- ~~Network selection (activation functions)~~ => relu activation is proven to be better
- ~~Initialization~~ => random (0, 0.01) is proven to be better
- Learning rate(s)
- ~~Loss function & output layer function~~ => cross entropy & softmax is proven to be better

Before we testing these different combinations, let's make a baseline prototype so that we can compare.

The hyperparameters of the baseline prototype in the part to are:

- Data presentation: serial
- Order: shuffle
- Network selection: relu (hidden layer), softmax (output)
- Initialization: random number between 0 and 0.01
- Learning rate: 0.01
- Update method: Stochastic Gradient Descent (SGD)
- Loss function: cross entropy

The error plot function of the baseline prototype is shown in Fig 31.

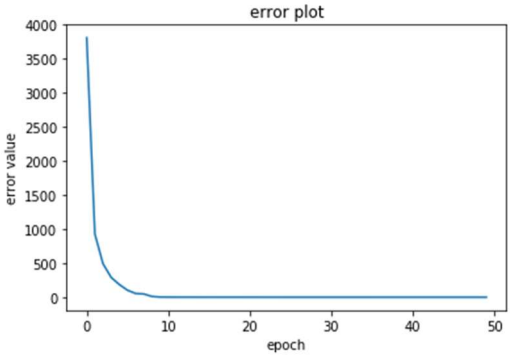


Fig 31. The error plot of the baseline prototype in structure 2

We tested the final weights on both training set and testing set:

```
5000 out of 5000 in the training set was labeled correctly (5000/5000)
4807 out of 5000 in the testing set was labeled correctly (4807/5000)
```

The kernel visualization is shown in Fig. 32.

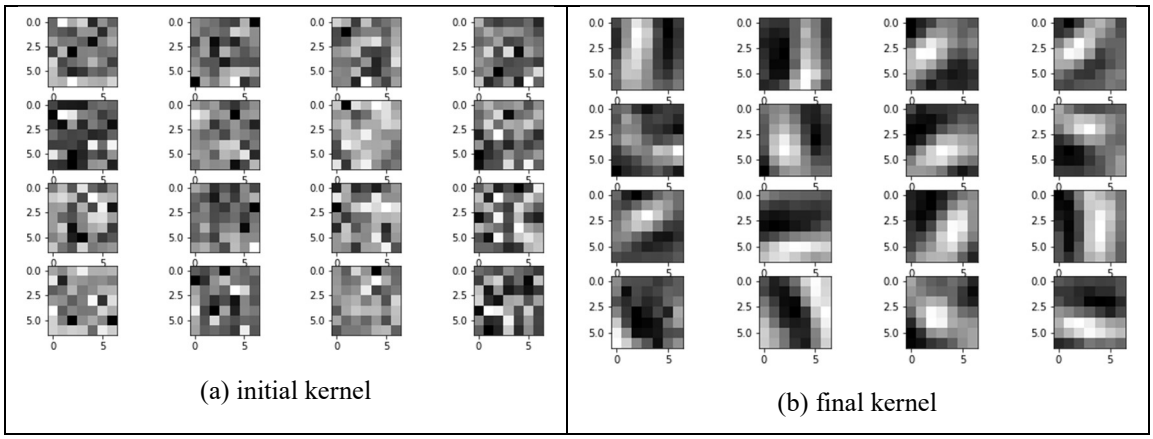


Fig 32. Initial and final kernel visualization

(1) Data presentation (serial, batch, mini-batch)

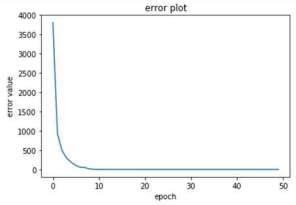
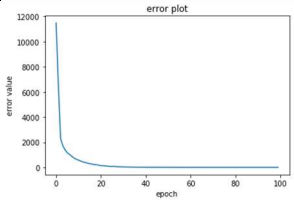
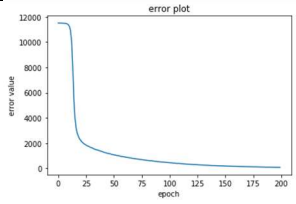
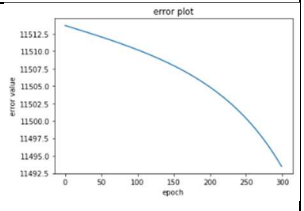
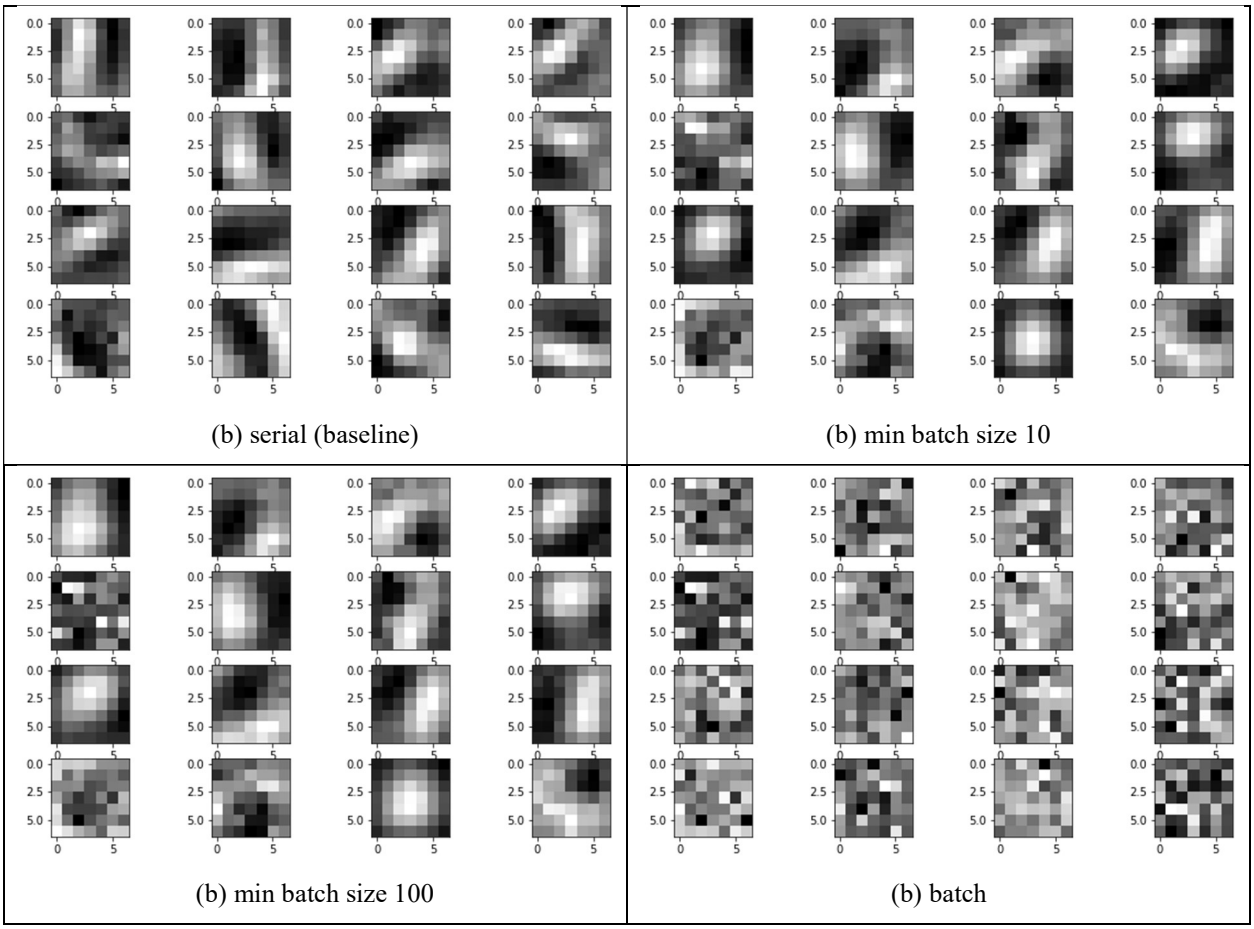
Serial (baseline)	Mini-batch size 10	Mini-batch size 100	Batch
			
Epoch: 50	Epoch: 100	Epoch: 200	Epoch: 300
Training: 5000/5000 Testing: 4807/5000	Training: 5000/5000 Testing: 4762/5000	Training: 4988/5000 Testing: 4719/5000	Training: 500/5000 Testing: 500/5000

Fig 33. The influence of the data presentation

Serial method, which is also called “online learning”, still performs the best. As minibatch size increases, the algorithm takes much more epochs to converge. The kernel visualization in each case is shown in Fig 34.



(b) serial (baseline) (b) min batch size 10

(b) min batch size 100 (b) batch

Fig 34. The kernel visualization for different data representation

In Fig 34, the batch method does not converge at all though it has already trained for 300 epochs. So the batch method is not suitable for a dataset that has a large number of samples.

(2) Learning rate(s)

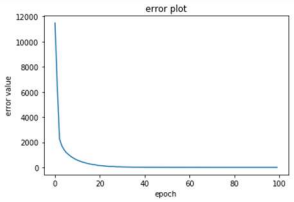
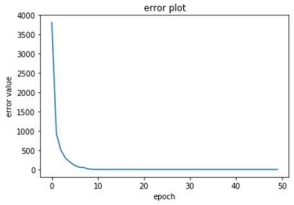
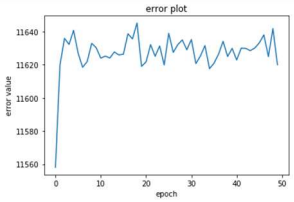
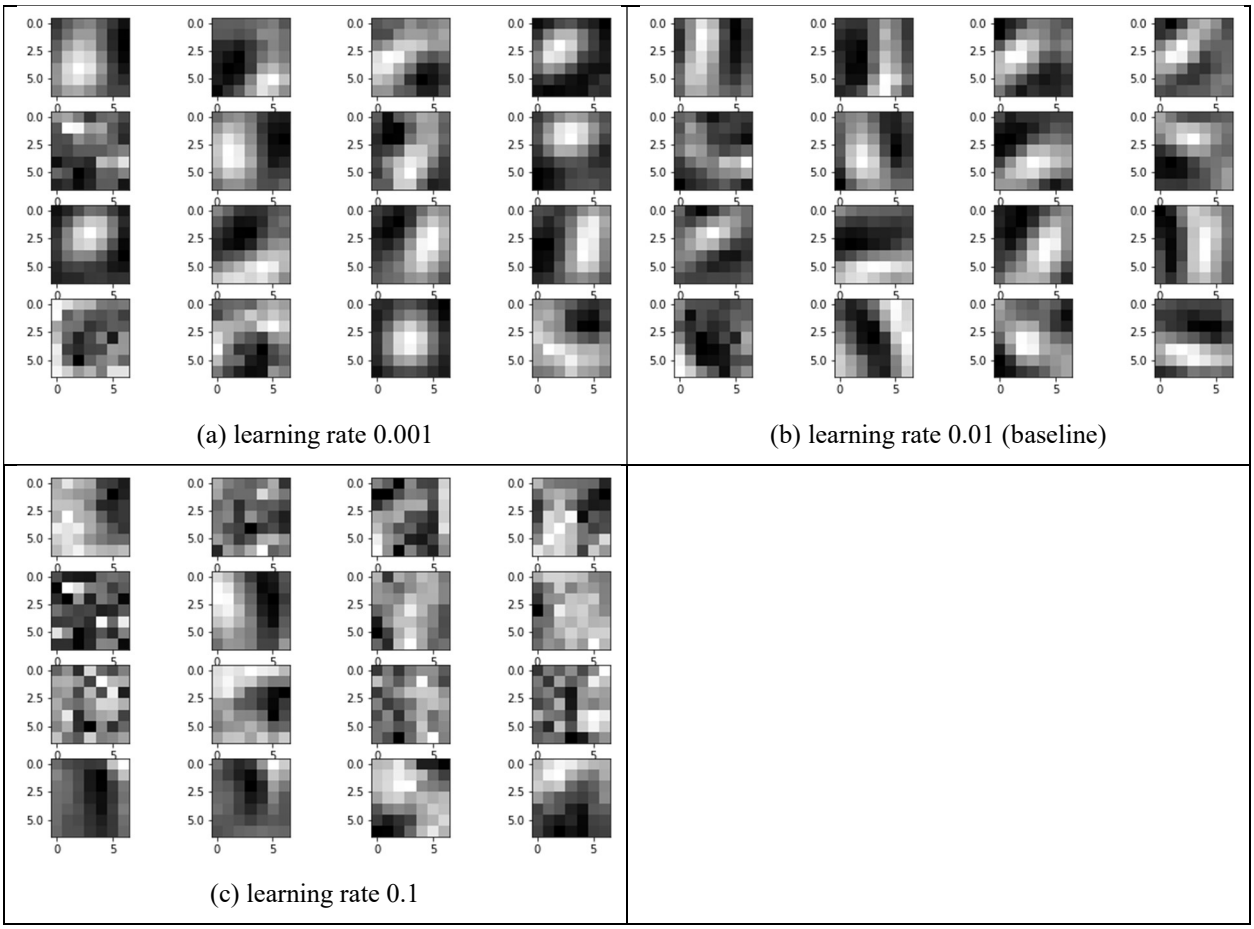
0.001	0.01 (baseline)	0.1
 Epoch: 100	 Epoch: 50	 Epoch: 50
Training: 5000/5000 Testing: 4758/5000	Training: 5000/5000 Testing: 4807/5000	Training: 500/5000 Testing: 501/5000

Fig 35. The influence of the learning rate

Small learning rate 0.001 takes more epochs (30 epochs vs. 10 epochs) to converge than the learning rate 0.01. However, a large learning rate 0.1 does not converge at all. The kernel visualization in each case is shown in Fig 36.



(a) learning rate 0.001

(b) learning rate 0.01 (baseline)

(c) learning rate 0.1

Fig 36. The kernel visualization for different learning rate

Thus, selecting a suitable learning rate can not only achieve high accuracy but also converge faster.

(3) Check overfitting

In the above experiment, we achieve almost 100% (5000/5000) training accuracy in the training set. That is a good result, but it may be due to the overfitting of the training set. In this part, we will look at the testing accuracy at each epoch at the same time as the algorithm learns from the training set.

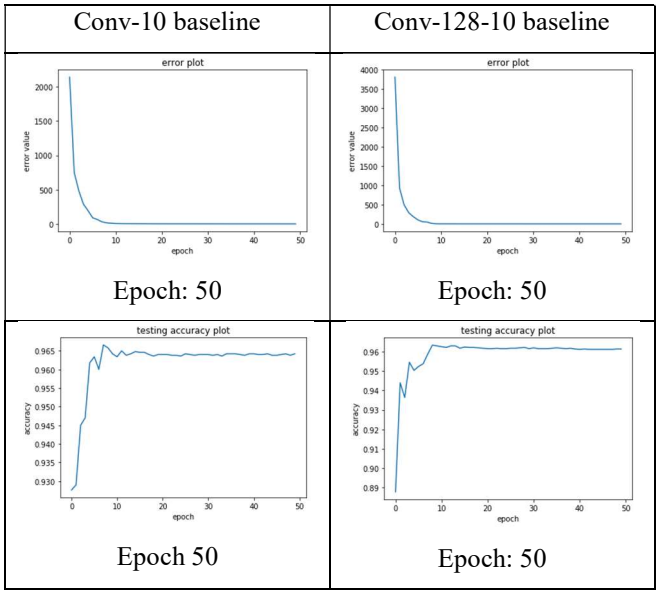


Fig 37. testing accuracy plot in each baseline structure

As shown in Fig 37, the algorithm converges at epoch 10. The testing accuracy after epoch 10 is pretty stable (~96.5%). So I think the algorithm learns pretty well and does not overfit the training set.

IV.D MNIST dataset

I used the structure 1 (conv-10) to test its performance on the MNIST dataset, which has 60,000 training samples and 10,000 testing samples.

The hyperparameters of the prototype in this part are:

- Data presentation: serial
- Order: shuffle
- Network selection: relu (hidden layer), softmax (output)
- Initialization: random number between 0 and 0.01
- Learning rate: 0.01
- Update method: Stochastic Gradient Descent (SGD)
- Loss function: cross entropy

The error plot function of the baseline prototype is shown in Fig 38.

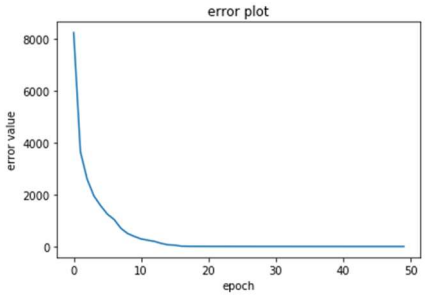
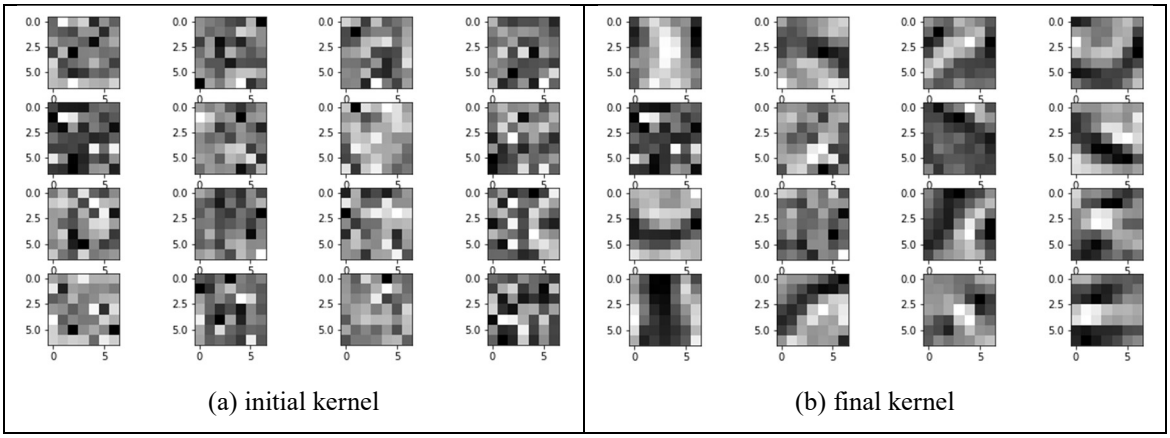


Fig 38. The error plot of the prototype on MNIST dataset

We tested the final weights on both training set and testing set:

```
60000 out of 60000 in the training set was labeled correctly (60000/60000)
9872 out of 10000 in the testing set was labeled correctly (9872/10000)
```

The kernel visualization is shown in Fig. 39.



(a) initial kernel (b) final kernel

Fig 39. Initial and final kernel visualization on MNIST dataset

IV.E Part 4 - deconvolution

Take the first example image from each digit and push it through your trained network. For each image, generate all sixteen response fields. For each field, apply "deconvolution" (convolution transpose). Show your results and discuss them.

Note: the dataset image given in this project is transposed so I will use the image from the MNIST dataset.

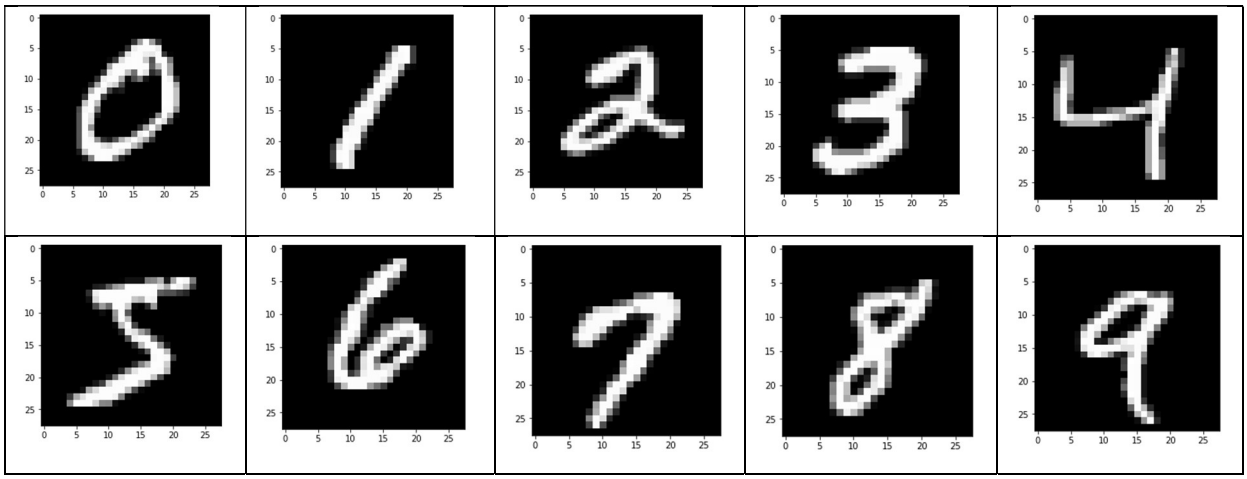


Fig 40. First example image from each digit from MNIST dataset

The deconvolution of sixteen local fields for each digit is shown in Fig 41. As we can see, each kernel captures one texture information. For example, look at the 13th kernel (bottom left), it detects the vertical information in the image. For each digit's 13th deconvolution in Fig 41, the color is bright if there is a vertical line. Digit "4" and "9" have more firings than digit "3" and "5" because digit "4" and "9" have a vertical line. And all the other kernels do the similar thing: the 3rd and 11th kernel detect "/" shape, the 12th kernel detects "\" shape and etc. The 5th kernel seems not working.

Fig 41. Deconvolution of all sixteen response fields for each digit

V. Lessons Learned

In this project, I learned how fully and convolutional neural network works. Understanding the concept is easy, but programming and debugging is hard. I have practiced a lot using Python and Numpy in this project. The most difficult part in this project I think is the backpropagation part. Although modern neural network research uses deep learning framework, it's still worthy to write the backpropagation code, especially on the convolution part. By tuning the hyper-parameters, I learned how these hyper-parameters actually influence the training:

- Serial method (online learning) is still effective in a small dataset; as the increase of the mini-batch size, the algorithm needs more epochs to converge;
- Random shuffle order converges faster than the fixed order;
- Relu activation function, a simple but effective activation, converges faster than sigmoid and tanh activation;
- Initialization is of vital importance to the training; initializing with a small value (0.001 to 0.01) is usually a good choice;
- Setting suitable learning rate is the priority of the tuning algorithm. A small learning rate takes many epochs to converge, and a large learning rate fails to make the algorithm converge. Usually, we need some practice or intuition or select the best learning rate.

The most difficulty I had in this project is how to debug the algorithm. In the beginning, I have no idea what or how to do next when the algorithm does not converge or does not even learn. Jupyter Notebook is a good interface to visualize the output but is not good at debugging to see the intermediate value. So I turn to Spyder IDE to set up breakpoint and see the intermediate values. Spyder IDE is like MATLAB that can run and stop.

The biggest problem that I had for days is the gradient vanishing problem. I use sigmoid as output layer activation function and MSE as loss function. When doing the backpropagation, the gradient is x * (1 - x). As the computed output gets closer and closer to 0 or 1, the value of gradient x * (1 - x) gets smaller and smaller. After a few epochs, the weights do not change at all. One way to solve it is to do the "gradient clipping", which is to set a threshold to the activation gradient if the gradient value is smaller than a threshold. The sigmoid activation function with gradient clipping code is shown below:

```
clipping_val = 0.05
def sigmoid(x, derive=False): # the updated sigmoid function
    if derive:
        tmp = x * (1 - x)
        x_der = (tmp > clipping_val) * tmp + (tmp < clipping_val) * clipping_val
        return x_der
    return 1 / (1 + np.exp(-x))
```

Using this kind of sigmoid function help speed up the converging of the algorithm. However, the clipping value need also to be tuned. Another approach to solving it is to use cross-entropy loss and softmax function as output layer activation function to update the weights in a different way. This combination eliminates the term x * (1 - x) so that the algorithm will converge fast.

There are many deep learning optimization approaches that I have not tried to improve the training algorithm such as Adam update algorithm, batch normalization algorithm and etc. The backpropagation code will be much complicated using these advanced methods. I guess that's the reason why people all use deep learning framework such as TensorFlow and PyTorch. What we need is only a few lines of code:

```
def baseline_model():
    # create model
    model = Sequential()
    model.add(Conv2D(16, (7, 7), input_shape=(1, 28, 28), activation='relu'))
    model.add(Flatten())
    #model.add(Dense(128, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    # Compile model
    model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])
return model

# build the model
model = baseline_model()
# Fit the model
model.fit(X_train, y_train, validation_data=(X_test, y_test), epochs=50, batch_size=1, verbose=2)
```

The above code will do the exact same thing as structure 1 (conv-10) in part 3, which will save our plenty of time and work if we just want to use it rather than understand it.

This project helps me to rethink the inner structure of the convolutional neural network. The most disappointing part is that I don't have powerful computational resources (mine is i7-4770, 16G). Although I wrote the code to be as efficient as I can, the waiting time is still so long. In this project, I did not implement vectorization for batch processing, which I think can help speed up the training of the algorithm in the future.

VI. References

[1] Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A large-scale hierarchical image database." (2009): 248-255.

[2] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." In *Advances in neural information processing systems*, pp. 1097-1105. 2012.

[3] LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86, no. 11 (1998): 2278-2324.

[4] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." *Cognitive modeling* 5, no. 3 (1988): 1.

[5] http://cs231n.github.io/neural-networks-1/

[6] Won, Yonggwan. Nonlinear correlation filter and morphology neural networks for image pattern and automatic target recognition. University of Missouri-Columbia, 1995.

[7] http://cs231n.github.io/convolutional-networks/

[8] https://stats.stackexchange.com/questions/277203/differentiation-of-cross-entropy (cross entropy)

[9] Shen, S. Multi-scale target detection based on morphological shared-weight neural network (Thesis, University of Missouri - Columbia) 2017.