

Classification of Hand-Written Digits Using Scattering  
Convolutional Network  
- Final Report

Dongmian Zou

Email Address: zou@math.umd.edu

Department of Mathematics, University of Maryland, MD 20742

Advisor: Prof. Radu Balan

Email Address: rvbalan@math.umd.edu

Department of Mathematics, University of Maryland, MD 20742

Co-advisor: Dr. Maneesh Singh

Email Address: maneesh.singh@sri.com

SRI International, Princeton, NJ 08540

May 11, 2016

**Abstract**

In image classification it is advantageous to build classifiers based on feature extractors. It is demanded that the feature extractors unmask the identity of the image and remain stable with regard to small-scale deformations. A scattering convolutional network is such a feature extractor. In this project, we propose to use modern machine learning techniques to train the scattering convolutional network and test them on a standard database (MNIST) for the task of recognizing hand-written digits.

# Contents

<b>1</b>	<b>Background</b>	<b>3</b>
1.1	Convolutional Networks . . . . .	3
1.2	Scattering Convolutional Networks . . . . .	4
1.3	Machine Learning Techniques . . . . .	6
1.3.1	Gradient Descent Optimization . . . . .	6
1.3.2	Error Backpropagation . . . . .	7
1.3.3	Support vector machine . . . . .	7
<b>2</b>	<b>Approach</b>	<b>8</b>
2.1	Network Structure . . . . .	8
2.2	Optimization Problem . . . . .	9
2.3	Multi-Class Classification . . . . .	10
<b>3</b>	<b>Implementation</b>	<b>12</b>
3.1	Hardware and Software . . . . .	12
3.2	Database . . . . .	13
3.3	Implementation Details . . . . .	13
3.3.1	Forward Propagation . . . . .	14
3.3.2	LibSVM . . . . .	15
3.3.3	Back Propagation . . . . .	15
3.4	Cross-Validation . . . . .	17
<b>4</b>	<b>Validation</b>	<b>19</b>
<b>5</b>	<b>Testing</b>	<b>20</b>
5.1	Classification Results . . . . .	20
5.2	Stability Results . . . . .	21
5.2.1	Stability to Translation . . . . .	21
5.2.2	Stability to Deformation . . . . .	23
<b>6</b>	<b>Project Schedule and Milestones</b>	<b>25</b>
<b>7</b>	<b>Deliverables</b>	<b>26</b>

# 1 Background

Recently convolutional neural networks (ConvNets) have enjoyed conspicuous success in various pattern recognition tasks ([9], [10]). Although a clear understanding of why they perform so well is still absent, it is believed that it is the multi-layer structure that makes ConvNets outstanding ([3], [4]). As a feature extractor, a ConvNet is built to catch different features of an image in different layers. In [8], the authors apply wavelet theory to formulate a structure named Scattering Convolutional Network, which falls in the general category of ConvNets. The authors showed that the feature extractors they built are approximately invariant to translation and stable to small-scale deformation. The theory is extended to general semi-discrete frames in [12]. We are going to apply these theories to perform classification of hand-written digits.

## 1.1 Convolutional Networks

Convolutional networks (ConvNets) are artificial neural networks that “use convolution in place of general matrix multiplication in at least one place” ([1], Ch. 9). In one dimension, the convolution operation “\*” is defined by

$$(x * h)(t) = \int x(s)h(t - s)ds .$$

In the above, despite the symmetric role of  $x$  and  $h$ , we usually call  $x$  the input, and  $h$  the filter. A digital image is a discretized two-dimensional object, whence we consider the convolution operator in two dimension with the discrete measure:

$$(X * H)(t_1, t_2) = \sum_{s_1} \sum_{s_2} X(s_1, s_2)H(t_1 - s_1, t_2 - s_2) .$$

In ConvNets, generally  $X$  is the input to the current layer and  $H$  is the filter which is commonly of much smaller size than  $X$ . The reason is both for computational efficiency and for sparse interactions between pixels ([1], Ch. 9). Since the size of the filter is small, it makes more sense to use the equivalent definition for the convolution, that is,

$$(X * H)(t_1, t_2) = \sum_{s_1} \sum_{s_2} X(t_1 - s_1, t_2 - s_2)H(s_1, s_2) . \tag{1}$$

Here  $t_1, t_2, s_1, s_2$  are the pixel positions. Suppose  $X \in \mathbb{R}^{D \times D}$  and  $H \in \mathbb{R}^{d \times d}$  (we can replace  $\mathbb{R}$  by  $\mathbb{C}$  for some applications, but it does not affect our discussions here), in practice we

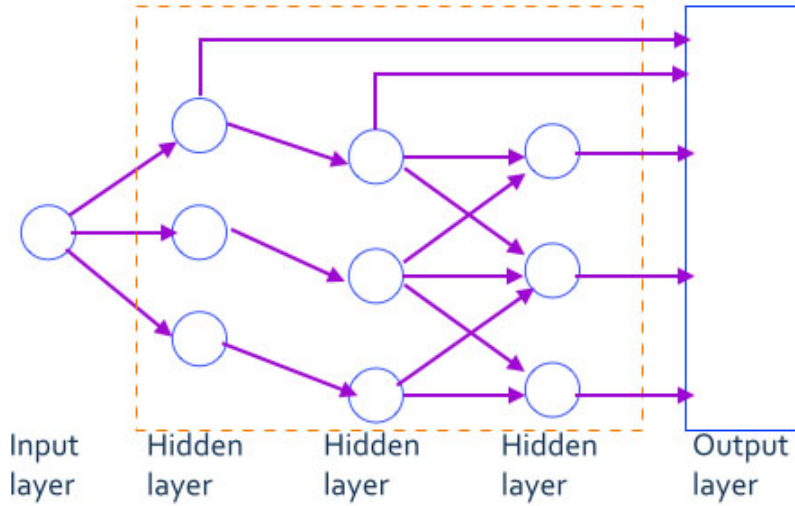


Figure 1: A typical ConvNet

would consider  $D > d$ . It is useful to think of  $X$  as a function  $X : \mathbb{Z}_D \times \mathbb{Z}_D \rightarrow \mathbb{R}$ . Then it makes sense when  $t_1 - s_1$  or  $t_2 - s_2$  is a negative number in (1).

Figure 1 illustrates a typical convolutional network. We see that between the input and the output it is natural to partition the neurons (operations) into several layers. In general, each layer of a ConvNet has three stages. The first stage is a convolution stage, where the input is convolved with filters; the second stage is a detector stage, where nonlinearity is applied to the output from the convolution stage; the last stage is a pooling stage, where local maximizing or averaging is applied to the output from the detector stage.

## 1.2 Scattering Convolutional Networks

Scattering Convolutional Networks are in the category of ConvNets with pre-defined filters ([8]). The filters are built by dilations of a wavelet. In the detector stage in each layer, the nonlinearity is chosen to be the absolute value function. In the pooling stage, a local averaging is done by convolution with a low-pass filter.

Let  $\psi$  be a wavelet in  $L^2(\mathbb{R}^d)$ . The dilation of  $\psi$  by  $\lambda$ ,  $\psi_\lambda$ , is defined by

$$\psi_\lambda(t) = \lambda^d \psi(\lambda t) .$$

Consider a path  $q = (\lambda_1, \lambda_2, \dots, \lambda_m)$ . The scattering propagator,  $U[q]$ , is defined by

$$U[q]x = || |x * \psi_{\lambda_1}| * \psi_{\lambda_2}| \cdots \psi_{\lambda_m} | ,$$

and the scattering transform,  $S[q]$ , is defined by

$$S[q]x = U[q]x * \phi_J ,$$

where  $J > 0$  is some pre-determined scale. For an input signal  $x$ , in the  $m$ -th layer, a path  $q = (\lambda_1, \lambda_2, \dots, \lambda_m)$  generates an output given by  $S[q]x$ . Figure 2 shows the process.

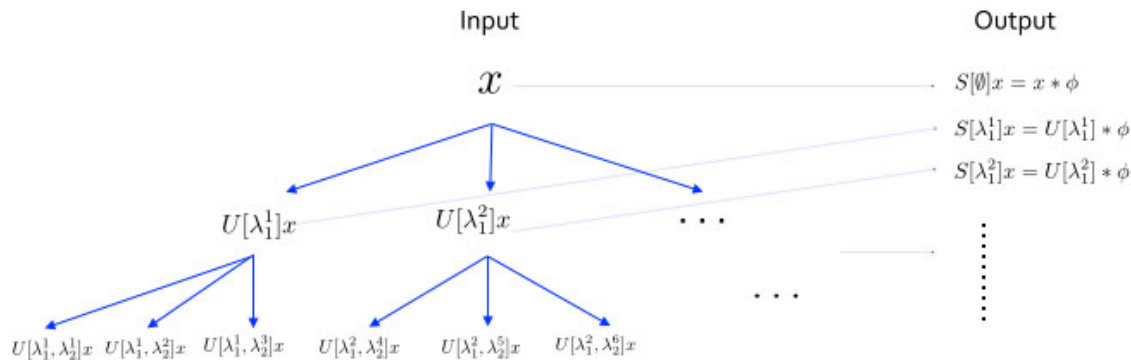


Figure 2: A typical scattering convolutional network

The Scattering Convolutional Networks are shown to be approximately translation invariant and stable to small-scale deformation. However, the stability is guaranteed only if we choose a specifically designed wavelet to start with. In [12], the authors extend the theory to general semi-discrete frames. They use the same network structure but give a less strict condition on the filters. This stability property is important for feature learning because we want the feature vector to be insensitive to different handwriting styles.

The stability properties still hold even if we use other nonlinearities in place of  $|\cdot|$ . For instance, we can use  $|\cdot|^2$  in place (in this case we have a polynomial network and there are other ways for training). If we have a bounded input signal then the scattering transform with  $|\cdot|^2$  is also approximately invariant to translation and stable to deformation for certain filters. In the following we mainly discuss the scattering network with  $|\cdot|$ , but we also test the results for the scattering network with  $|\cdot|^2$ . We use the same training techniques for both.

### 1.3 Machine Learning Techniques

Our project is motivated by the above scattering network and its extension. Instead of fixing everything in the network beforehand, we want the computer to “tell” us what parameters to use for this structure. For instance, if we want to relate the input  $x$  and the output  $y$  by some function  $y = f(x)$ , then the process for determining the precise form of  $f$  is called the *training phase* ([2], Ch. 1). From the perspective of human, we “train” the system; while from the perspective of the machine, it “learns” from the given data set. For our project, we are given a set of training data, where each image has a “label” that tells us the de facto output.

The machine learning techniques we will use are not limited to but include the gradient descent optimization, the error backpropagation and the support vector machine.

#### 1.3.1 Gradient Descent Optimization

In most machine learning tasks we would like to find the minimum of a *loss function* (also called a *cost function*). Suppose  $l(\boldsymbol{\lambda})$  is the loss function with respect to the parameter  $\boldsymbol{\lambda}$ . The gradient descent is an iterative method for locating the minimum of  $l$  (see [1], Ch. 8; [2], Ch. 5). The updating step is given by

$$\boldsymbol{\lambda}^{(\nu+1)} = \boldsymbol{\lambda}^{(\nu)} - \eta \nabla l(\boldsymbol{\lambda}^{(\nu)}) . \quad (2)$$

where  $(\nu)$  is the step number. At each step,  $\boldsymbol{\lambda}$  travels towards the direction of the steepest descent. The scalar  $\eta$  is called the *learning rate* by the machine learning community. It is the same as the *stepsize* in standard optimization literature.

There are some alternative optimization methods, for example, conjugate gradients and quasi-Newton methods. They are in general more efficient than the gradient descent method. However, for training convolutional networks, we usually have a large amount of training data. Therefore, the loss function will be the sum of a large number of terms if we use deterministic methods. Taking this into consideration, deterministic methods are rarely used for network training. In practice, most training algorithms use stochastic approximation for the gradient. The conjugate gradients and quasi-Newton methods are not amenable to stochastic gradients. Nevertheless, it has been empirically demonstrated that the stochastic gradient descent method performs well for training ConvNets ([9], [10]). We will use both deterministic and stochastic methods and compare the results.

### 1.3.2 Error Backpropagation

The idea of *error backpropagation* is that the partial derivative of the loss function  $l(\lambda)$  with respect to  $\lambda$  can be decomposed ([1], Ch. 6). It is based on the chain rule of taking derivatives.

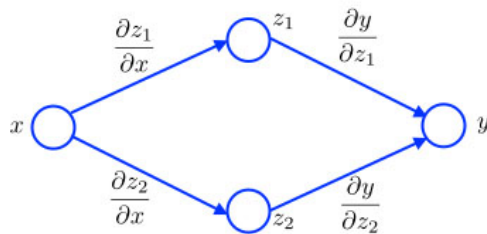


Figure 3: A simple illustration of backpropagation

Figure 3 epitomizes a simple neural network. The output  $y$  is connected to the input  $x$  through two intermediate outputs  $z_1$  and  $z_2$ . Hence we have

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z_1} \frac{\partial z_1}{\partial x} + \frac{\partial y}{\partial z_2} \frac{\partial z_2}{\partial x}.$$

It is as if we had a network in the opposite direction for the partial derivatives. In general, we compute the partial derivatives between each connected neurons, and then propagate backward along the network to get the derivative of the loss function with respect to the parameters that we want to train.

### 1.3.3 Support vector machine

The support vector machine (SVM) is a decision machine designed for two-class classification problems ([2], Ch. 7). A typical trained SVM has two parameters: the weights  $w$  and bias  $b$ . Suppose the two classes are given by  $\{\pm 1\}$ , then given an input  $y$ , the sign of  $\langle w, y \rangle + b$  determines whether  $y$  falls into the class  $\{-1\}$  or  $\{+1\}$ .

We would like to train the SVM parameters  $\{w, b\}$ . The idea of the SVM is to maximize the margin between the decision boundaries. It turns out that we need to maximize  $\|w\|^{-1}$ , that is, to minimize  $\|w\|$ . However, at the same time, we want to mildly penalize points that go to the wrong side. A standard way to do this is the following: suppose the inputs provided by the training data are  $\{y_n\}_{n=1}^N$  with labels  $\{a_n\}_{n=1}^N$  where each label  $a_n \in \{\pm 1\}$ ,

we want to solve

$$\min_{w,b} \frac{1}{2} \|w\|^2 + C \sum_{n=1}^N l(y_n, a_n; w, b) ,$$

where  $C$  is a pre-determined parameter and  $l$  is the hinge-loss function defined by

$$l(y, a; w, b) = \max(0, 1 - a(b + \langle w, y \rangle)) ,$$

where  $a \in \{\pm 1\}$  is the label corresponding to  $y$ . In the above, the parameter  $C$  seeks a balance between the two factors: maximizing the distance of the two classes (minimizing  $\|w\|^2/2$ ) and minimizing the loss caused by points that are wrongly classified ( $\sum_n l(y_n, a_n; w, b)$ ).

## 2 Approach

### 2.1 Network Structure

In this project, our task is the classification of hand-written digits. Our goal is to build a classifier that identifies the digit contained in our input image. Our classifier is composed of two parts. The first is a scattering convolutional network that extracts the features of the images and outputs a feature vector. The second is an SVM that does the classification job on the feature vector. In our project, the structure of the classifier is fixed, while some parameters in the classifier are to be either trained using machine learning techniques, or selected from a few possible choices.

Figure 4 shows the detailed structure. The scattering network is a four-layer neural network that consists of one input layer, two convolutional layers and one output layer. In each convolutional layer,  $h_k^j$  is the filter to be trained; “ $|\cdot|$ ” is the operation of taking absolute value pointwise;  $g$  is a fixed low-pass filter that does local averaging. A downsampling is done before we send the feature vector  $y$  to the SVM (this guarantees that the output of the scattering network is approximately of the same size as the input). We select  $g$  and  $L$  from several possible candidates after cross-validation.

As we have discussed in Section 1.3, once we have determined all the parameters of the network in Figure 4, we can use it for the classification task. Suppose we have an image  $x$  and want to determine the digit it represents. We use  $x$  as an input and let it propagate through the network. It first goes through the scattering network to generate the feature vector  $y$ , and then the SVM determines which class  $x$  belongs to by evaluating  $\langle w, y \rangle + b$ .



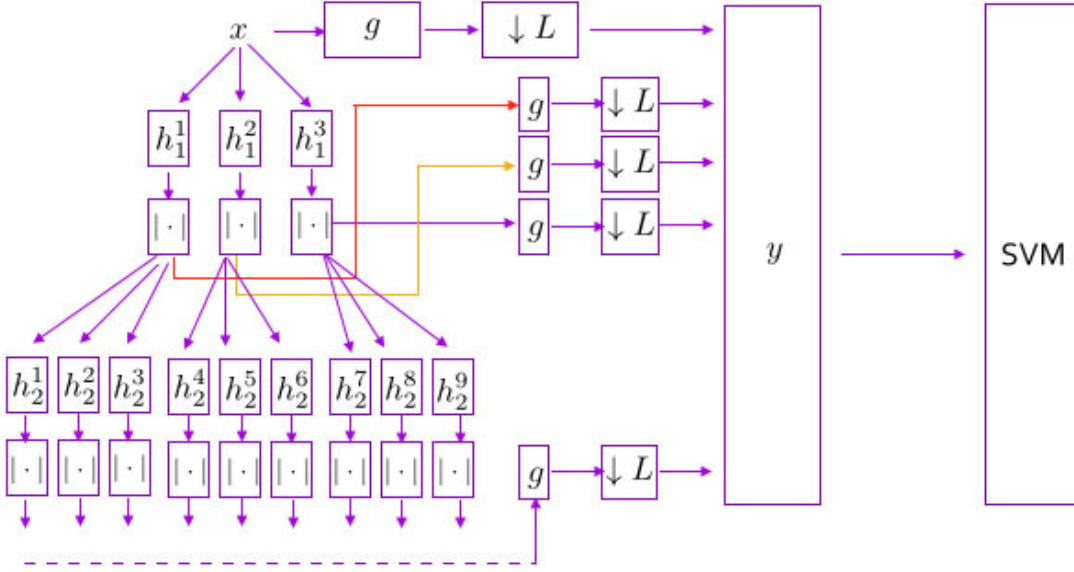


Figure 4: The structure of our scattering convolutional network

In our project, the two-dimensional filters,  $h_k^j$ , are parametrized as dilations of the tensor products of two one-dimensional wavelets. We use the same pre-defined wavelet  $\psi$  for both. That is,

$$h_k^j(t_1, t_2) = \psi_{\lambda_{k,1}^j} \otimes \psi_{\lambda_{k,2}^j}(t_1, t_2) = \lambda_{k,1}^j \lambda_{k,2}^j \psi(\lambda_{k,1}^j t_1) \psi(\lambda_{k,2}^j t_2) .$$

## 2.2 Optimization Problem

Let  $\lambda$  denote the vector composed of all the  $\lambda$ 's in the  $h$ 's. The training process deals with the unknown parameters  $\lambda; w, b$ . The trained  $\lambda; w, b$  are optimizers of the minimization problem of the SVM (see Section 1.3.3). Note that the feature vector  $y$  (which is the input of the SVM) now contains the unknown parameters  $\lambda$ . Therefore, according to the structure in Figure 4, the optimization problem for our project is as follows.

$$\min_{\lambda; w, b} \frac{1}{2} \|w\|^2 + C \sum_{n=1}^N l(y_n, a_n; w, b) , \quad (3)$$

where

$$l(y, a; w, b) = \max(0, 1 - a(b + \langle w, y \rangle)) ,$$

and  $y$  is the grouping of the following

(recall that “ $*$ ” is in the sense of Equation (1))

$$y_0 = x * g ;$$

$$y_1^j = \left| x * h_1^j \right| * g , 1 \leq j \leq 3 ;$$

$$y_2^j = \left| \left| x * h_1^{\lceil j/3 \rceil} \right| * h_2^j \right| * g , 1 \leq j \leq 9 ,$$

where the two-dimensional filters  $h_k^j$  is parametrized as

$$h_k^j(t_1, t_2) = \lambda_{k,1}^j \lambda_{k,2}^j \psi(\lambda_{k,1}^j t_1) \psi(\lambda_{k,2}^j t_2) .$$

The above training problem works for two-class classification, which is adapted to multi-class classification as we describe in the next section. In the optimization objective function (3),  $N$  can be either the total amount of training data (if we use a deterministic method), or the number of samples (if we use a stochastic method). The input of the network is  $N$  pairs of  $(x_n, a_n)$  where  $x_n$ 's are images of  $28 \times 28$  pixels (according to the database) and  $a_n$ 's are the corresponding labels indicating which class  $x$  is in (normalized so that  $a_n \in \{\pm 1\}$ ). The unknown parameters are the  $\lambda$ 's in the filters and the  $w$  and  $b$  in the SVM. The optimization problem with respect to  $\{\boldsymbol{\lambda}, w, b\}$  is a non-convex problem and difficult to solve. Nevertheless, if we fix  $\boldsymbol{\lambda}$ , then training the SVM is a convex optimization problem. Inspired by this fact, we will follow a two-step procedure. Specifically, we first fix  $\boldsymbol{\lambda}$  and train  $w$  and  $b$  (which is convex), and then fix  $w$  and  $b$  to train  $\boldsymbol{\lambda}$  (which is easier than the original problem and we use gradient descent for the minimization); and then iterate it until our stop criterion is met. We use a software called libSVM to train  $w$  and  $b$ , as described in Section 3.

Algorithm 1 summarizes our training process.

### 2.3 Multi-Class Classification

Now we discuss the multi-class variation of the above method. An SVM is inherently a binary classifier. If we want to build a multi-class classifier based on SVM methods, we

---

**Algorithm 1:** The algorithm for network training

---

Start with learning rate  $\eta$ , regularization parameter  $C$  ;  
randomly generate  $\lambda$ ;  
**while** *stop criterion not met* **do**  
    sample  $N$  examples  $\{x_1, x_2, \dots, x_N\}$  from the training set;  
    get the corresponding labels  $\{a_1, a_2, \dots, a_N\}$ ;  
    propagate forward to get  $\{y_1, y_2, \dots, y_N\}$ ;  
    call libSVM with input  $\{y_1, y_2, \dots, y_N\}$ ,  $\{a_1, a_2, \dots, a_N\}$  and  $C$ ;  
    update  $w, b \leftarrow$  output of libSVM;  
    set  $\mathbf{r} = 0$ ;  
    **for**  $n = 1$  *to*  $N$  **do**  
        compute  $\nabla_{\lambda} l(\lambda; x_n)$ ;  
         $\mathbf{r} \leftarrow \mathbf{r} + \nabla_{\lambda} l(\lambda; x_n)$ ;  
    update  $\lambda \leftarrow \lambda - \eta \mathbf{r}$  ;  
    adapt  $\eta$  accordingly.

---

have to train multiple SVM's. Broadly speaking, there are two approaches: "one-against-all" and "one-against-one" ([7]).

1. One-against-all: We have 10 classes  $\{0, 1, \dots, 9\}$ . We build 10 classifiers  $F_0, F_1, \dots, F_9$ . For  $p \in \{0, 1, \dots, 9\}$ , we use the label 1 if the input image is the number  $p$  and  $-1$  if the input image is not the number  $p$ . Note that each classifier has parameters  $\{\lambda_p, w_p, b_p\}$ . Our decision rule is

$$\text{output} = \arg \max_p \langle w_p, y_p \rangle + b_p ,$$

where  $y_p$  is the feature vector obtained from the convolutional network in  $F_p$ .

2. One-against-one: We build a classifier for each pair of classes (45 classifiers in total). In this case, our decision rule is given by a voting strategy: for each pair of classes, we use the associated classifier to choose one class, and then the vote of that class is added by one. Our decision rule is that we take the class that has the largest number of votes (if two classes have the same number of votes, then take an arbitrary one).

Note that in the above, we build multiple classifiers with different filters in the convolutional network. If we want to apply the above principle but only use one feature extractor,

then we need to formulate an optimization problem for the multi-class classification. The way to do this is not unique ([7]). One formulation based on (not exactly) “one-against-all” is:

$$\min_{\substack{\lambda; w_p, b_p \\ p=0, \dots, 9}} \frac{1}{2} \sum_{p=0}^9 \|w_p\|^2 + C \sum_{n=1}^N \sum_{p \neq a_n} l(y_n; w_p, b_p) ,$$

where  $a_n \in \{0, 1, \dots, 9\}$ ,

$$l(y_n; w_p, w_p) = \max(0, 2 + \langle w_p, y_n \rangle + b_p - \langle w_{a_n}, y_n \rangle - b_{a_n}) ,$$

and  $y_n$ 's are the same as given in Equation (3). The decision rule is the same as described in “one-against-all”.

Optimization problem for “one-against-one” is rarely seen in literature. Nevertheless, if we want to use only one feature extractor for “one-against-one”, we can consider the following problem:

$$\min_{\substack{\lambda; w_q, b_q \\ q=1, \dots, 45}} \frac{1}{2} \sum_{q=1}^{45} \|w_q\|^2 + C \sum_{n=1}^N \sum_{q: a_n \in A_q} l(y_n, a_n; w_q, b_q) ,$$

where  $a_n \in \{0, 1, \dots, 9\}$ ,  $A_q$  is the set composed of the two classes of the  $q$ -th pair, and

$$l(y_n, a_n; w_q, b_q) = \max(0, 1 - c_{n,q}(b_q + \langle w_q, y_n \rangle)) ,$$

where  $c_{n,q} \in \{\pm 1\}$  depends on the labeling of the  $q$ -th SVM. Since  $w$  and  $b$ 's are to be retrieved from the libSVM (which uses the “one-against-one” principle by default), we only need the partial derivatives of the objective function, which is simply the sum of the derivatives of the loss functions we considered in Section 2.2.

## 3 Implementation

### 3.1 Hardware and Software

We implement the algorithm on the personal laptop with

- CPU: 2GHz Intel Core i7
- Memory: 8 GB 1600 MHz DDR3
- OS: OS X El Capitan Version 10.11

The software we use is MATLAB R2015b. All the experiments are run in the terminal window without a GUI.

### 3.2 Database

The database we use is the standard MNIST database. MNIST is a publicly available database (<http://yann.lecun.com/exdb/mnist/>) of hand-written digits. The images are already preprocessed and formatted: they are of  $28 \times 28$  pixels; each pixel has a value ranging from 0 to 255 (we normalize the pixel values to be in  $[0, 1]$  when processing). The database provides 60,000 training images and 10,000 testing images. Examples of the images are given in Figure 5.

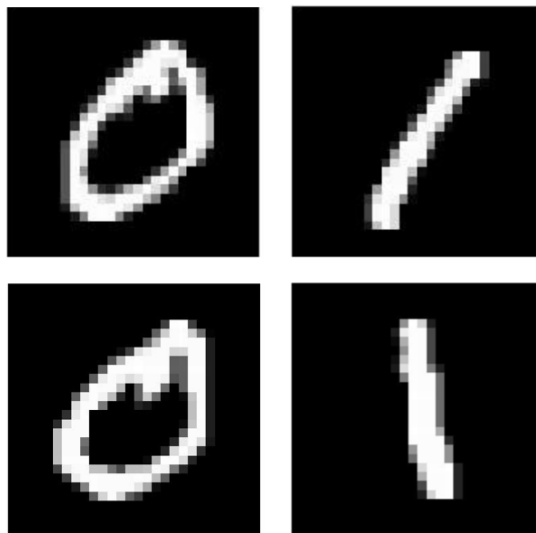


Figure 5: Examples of training images in the MNIST Database

In the training examples, we do not have an equal number of images for each digit. We take 5,400 images for each digit for training purpose. For cross-validation purposes, we further divide the 5,400 images into 3,600 images for training, and 1,800 for testing.

### 3.3 Implementation Details

Our implementation is based on Algorithm 1 described in Section 2. We run both a stochastic version and a deterministic version. In the stochastic method, the number of samples,  $N$ , is taken to be a small number (We take  $N = 10$  in this project); in the deterministic version,  $N$  is just the total number of images available for training (i.e. for two-class classification problem,  $N = 2 \times 3600 = 7200$ ). We repeat several steps until  $\lambda$  converges or the

number of iterations exceeds some value. Those steps are described in details as follows.

### 3.3.1 Forward Propagation

We randomly generate the starting value of  $\lambda$ . We make sure that (1) the starting value is positive; (2) the starting value for each branch is apart from each other. For each training image  $x_n$ , we need to get a corresponding feature vector  $y_n$ . This is done by forward propagation through the network. In our implementation, the operation of convolution is done following a variant version of Formula (1). The detail is shown in Figure 6. In this way, the output  $X * H$  is of the same size as  $X$ .

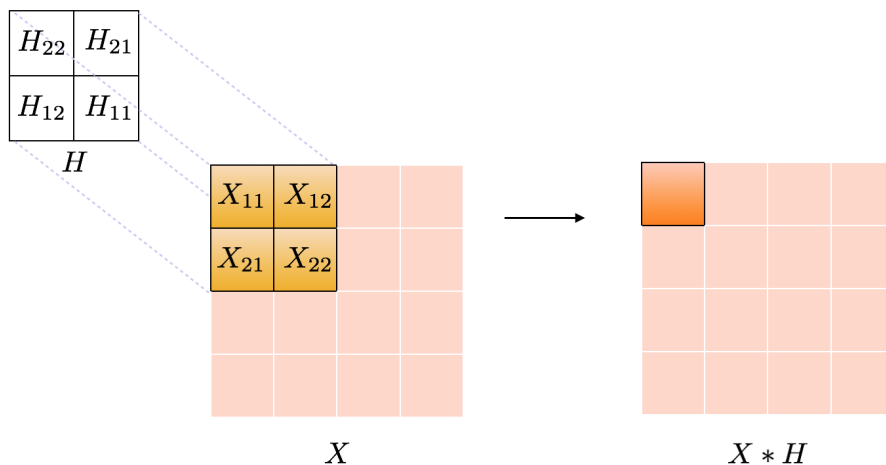


Figure 6: Convolution of  $X$  and  $H$ . Here the filter  $H$  is a small patch. We rotate it by  $180^\circ$  and align the upper left corner with the pixel in  $X$  that we want to compute the convolution value. For instance, as shown in this figure,  $(X * H)_{11} = X_{11}H_{22} + X_{12}H_{21} + X_{21}H_{12} + X_{22}H_{11}$ .

Note that there are 12 filters in our structure. Therefore, including the original signal  $x_n$  (which has size  $28 \times 28$ ), the feature vector  $y_n$  has size  $28 \times 28 \times 13$  before we downsample. Roughly speaking, the downsampled  $y_n$  has size  $28 \times 28 \times 13/L^2$ . We can consider it as a column vector as we send it to the SVM.

### 3.3.2 LibSVM

Once we have the feature vector  $y_n$ 's, we use libSVM to get  $w$  and  $b$ . LibSVM is a publicly available (<https://www.csie.ntu.edu.tw/~cjlin/libsvm/>) software for training SVM's.

The main program of libSVM is written in C. Therefore, before using it we need to generate “mex” files for MATLAB to run libSVM. For training purposes, we call the file “svmtrain.mexmaci64”. The input are two matrices: first, a matrix whose columns are the  $y_n$ 's; second, a row vector whose entries are the label  $a_n$ 's. The output is a structure from which we retrieve the optimizers  $w$  and  $b$ . Here  $w$  is a vector whose size is the same as  $y_n$  (the same for each  $n$ ),  $b$  is a scalar.

### 3.3.3 Back Propagation

After we update  $w$  and  $b$  from the SVM, we train the parameter  $\lambda$  from the filter  $h_k^j$ 's in the convolution layer. We have 12 filters and thus 24 parameters to train. With fixed  $w$  and  $b$ , the minimization problem (3) is turned into

$$\min_{\lambda} \sum_{n=1}^N l(\lambda; x_n),$$

where we use

$$l(\lambda; x_n) := l(y_n, a_n; w, b), \quad \forall n$$

to emphasize the fact that  $l$  depends on the unknown parameters  $\lambda$  and the training data  $x_n$ 's. The updating step follows Equation (2).

We now compute the gradient  $\nabla_{\lambda} l(\lambda; x)$  for a fixed  $x$ . Note that the loss function  $l(y, t; w, b)$  is not a smooth function with respect to  $y$ . We use a smooth function in place. Our choice is a  $C^1$  function  $L$  defined by

$$L(y, a; w, b) = \begin{cases} 0.5 - a(b + \langle w, y \rangle) & , \text{ if } a(b + \langle w, y \rangle) \leq 0; \\ 0.5(1 - a(b + \langle w, y \rangle))^2 & , \text{ if } 0 < a(b + \langle w, y \rangle) \leq 1; \\ 0 & , \text{ otherwise.} \end{cases}$$

Its partial derivative with respect to  $y$  is given by

$$\nabla_y L(y, a; w, b) = \begin{cases} -aw & , \text{ if } a(b + \langle w, y \rangle) \leq 1; \\ a(a(b + \langle w, y \rangle) - 1)w & , \text{ if } 0 < a(b + \langle w, y \rangle) \leq 1; \\ 0 & , \text{ otherwise.} \end{cases}$$

Also, we need to use some smooth function  $F$  in place of  $|\cdot|$ . Our choice is a  $C^\infty$  function  $F : \mathbb{R} \rightarrow \mathbb{R}$  defined by

$$F(t) = (|t|^2 + \epsilon^2)^{1/2}$$

for some small  $\epsilon$ . We use  $F$  (resp.  $F'$ ) for the operation of applying  $F$  (resp.  $F'$ ) pointwise as well (the same treatment as for  $|\cdot|$ ). We have

$$F'(t) = \frac{t}{(|t|^2 + \epsilon^2)^{1/2}}.$$

Moreover, we use  $\Psi(\lambda)$  defined by

$$\Psi(\lambda)(t) := \lambda\psi(\lambda t)$$

to emphasize the dependence on the variable  $\lambda$ . Note that

$$\frac{\partial L}{\partial \lambda_{k,i}^j} = \left\langle \nabla_{y_k^j} L, \frac{\partial y_k^j}{\partial \lambda_{k,i}^j} \right\rangle.$$

Now we give expressions for  $\partial y_k^j / \partial \lambda_{k,i}^j$ . We use  $\odot$  to denote pointwise multiplication of two vectors or matrices of the same size (which is an abuse of notation, but we do not plan to use  $\odot$  elsewhere). Since  $i$  takes value in  $\{1, 2\}$ , we use  $i'$  to denote the complement of  $i$  in  $\{1, 2\}$ . We have

$$\begin{aligned} \frac{\partial y_1^j}{\partial \lambda_{1,i}^j} &= \left[ F' \left( x * (\Psi(\lambda_{1,i}^j) \otimes \Psi(\lambda_{1,i'}^j)) \right) \odot \left( x * (\Psi'(\lambda_{1,i}^j) \otimes \Psi(\lambda_{1,i'}^j)) \right) \right] * g ; \\ \frac{\partial y_2^{3j-\iota}}{\partial \lambda_{1,i}^j} &= \left\{ F' \left( F \left( x * (\Psi(\lambda_{1,i}^j) \otimes \Psi(\lambda_{1,i'}^j)) \right) * \left( \Psi(\lambda_{2,i}^{3j-\iota}) \otimes \Psi(\lambda_{2,i'}^{3j-\iota}) \right) \right) \odot \right. \\ &\quad \left[ F' \left( x * (\Psi(\lambda_{1,i}^j) \otimes \Psi(\lambda_{1,i'}^j)) \right) \odot \left( x * (\Psi'(\lambda_{1,i}^j) \otimes \Psi(\lambda_{1,i'}^j)) \right) \right] \\ &\quad \left. * \left( \Psi(\lambda_{2,i}^{3j-\iota}) \otimes \Psi(\lambda_{2,i'}^{3j-\iota}) \right) \right\} * g, \quad \text{for } \iota = 1, 2, 3; \\ \frac{\partial y_2^j}{\partial \lambda_{2,i}^j} &= \left[ F' \left( F \left( x * (\Psi(\lambda_{1,i}^{[j/3]}) \otimes \Psi(\lambda_{1,i'}^{[j/3]})) \right) * \left( \Psi(\lambda_{2,i}^j) \otimes \Psi(\lambda_{2,i'}^j) \right) \right) \odot \right. \\ &\quad \left. \left( F \left( \left( x * (\Psi(\lambda_{1,i}^{[j/3]}) \otimes \Psi(\lambda_{1,i'}^{[j/3]})) \right) \right) * \left( \Psi'(\lambda_{2,i}^j) \otimes \Psi(\lambda_{2,i'}^j) \right) \right) \right] * g. \end{aligned}$$

The above expression is simply given by the chain rule. We can compute  $\partial L / \partial \lambda_{k,i}^j$  by backpropagation as described in Section 1.3. A sketched illustration is given in Figure 7.



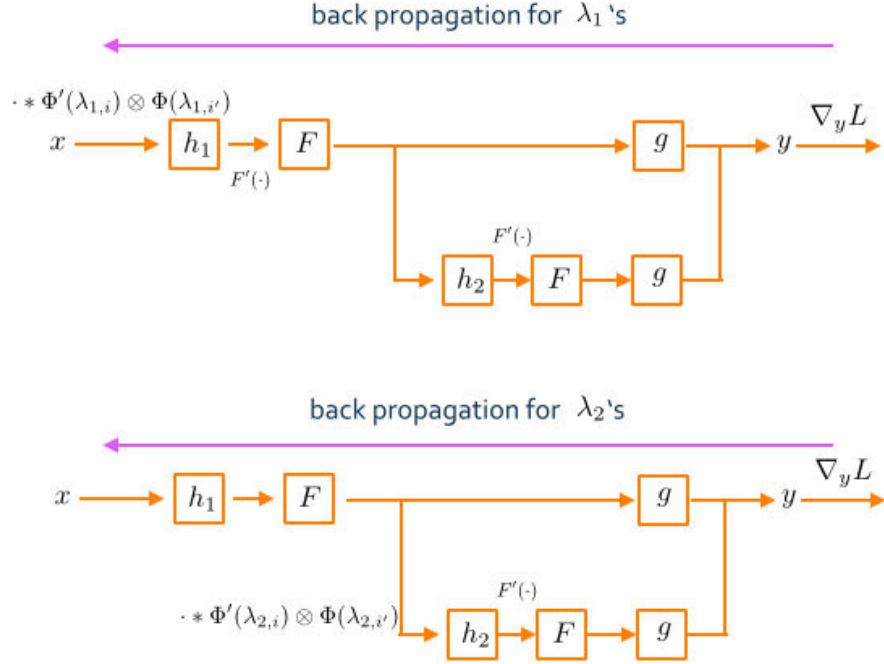


Figure 7: Illustration of the backpropagation process (“.” denotes the input from the last node). To compute the derivative, propagate backward, take the product of all marked values. Take the sum when two branches merge.

The updating step is  $\lambda^{(\nu+1)} = \lambda^{(\nu)} - \eta \sum_{n=1}^N \nabla_{\lambda} l(\lambda^{(\nu)}; x_n)$ . Note that this formula does not guarantee that we still get a positive  $\lambda$ . If any component of  $\lambda$  is non-positive, we replace  $\eta$  by a smaller one so that  $\lambda$  stays positive.

### 3.4 Cross-Validation

For parameter selection, recall that for each digit we split the 5,400 examples further into 3,600 for training and 1,800 for testing. We use cross-validation to determine the downsampling factor  $L$  and the low-pass filter  $g$ .

For the downsampling factor, since the size of the feature vector  $y_n$  is 13 times that of the input image  $x_n$ , we consider to select the downsampling factor from  $3 \times 3$  (3 for row, and 3 for column),  $4 \times 4$  and  $5 \times 5$ . For simplicity, we represent them as  $L = 3, 4, 5$ , respectively.

For the low-pass filter, we consider to select it between a standard sinc function (we denote this choice by LP) and a simple local averaging kernel (constant in each entry, which

is equivalent to taking average value of neighboring pixels) (we denote this choice by AV). In the implementation,  $g$  is taken to be of size 4-by-4.

We run both a stochastic method and a deterministic method. For the stochastic method ( $N = 10$ ), the result is demonstrated in Figure 8. We see that (LP) takes much longer time than (AV) and the error rate grows dramatically with the downsampling factor. On the other hand, the downsampling factor does not cause a big difference in the performance of (AV). We believe that “ $L = 3 / AV$ ” is a reasonable choice for the stochastic method.

	L = 3 / LP	L = 4 / LP	L = 5 / LP	L = 3 / AV	L = 4 / AV	L = 5 / AV
Running time (hour)	14.5	10.2	8.5	5.5	5.2	5.3
Error rate (%) for “0”	1.67	7.44	13.5	0.72	3.17	3.56
Error rate (%) for “1”	0.39	2.11	0	0.11	0	0.11
Sum of loss for “0”	399.3	555.1	976.8	253.6	456.6	494.3
Sum of loss for “1”	202.1	454.5	336.7	135.7	122.9	63.8

Figure 8: Testing results for the stochastic method. For instance, the column starting with “ $L = 3 / LP$ ” means that the training process takes 14.5 hours; the frequency of “reading 0 incorrectly” is 1.67 %; the frequency of “reading 1 incorrectly” is 0.39 %;  $\sum_{\{n:a_n=0\}} l(y_n, a_n; w, b) = 399.3$ ;  $\sum_{\{n:a_n=1\}} l(y_n, a_n; w, b) = 202.1$ .

On the other hand, for the deterministic method, we do not implement (LP) since it is inefficient. Figure 9 shows the results for (AV). For each  $L$ , the running time is much longer than the stochastic method. Nevertheless, the error rate is small and in this case we have much smaller sum of loss function. There is no significant difference in performance for each  $L$ , and “ $L = 3 / AV$ ” still appears to be a reasonable choice.

	L = 3 / AV	L = 4 / AV	L = 5 / AV
Running time (hour)	22.4	21.6	22.4
Error rate (%) for "0"	0.17	0.28	0.28
Error rate (%) for "1"	0.17	0	0.28
Sum of loss for "0"	8.6	10.6	15.7
Sum of loss for "1"	8.5	4.1	18.0

Figure 9: Testing results for the deterministic method

## 4 Validation

We use the MatConvNet Toolbox for validation. MatConvNet is a publicly available (<http://www.vlfeat.org/matconvnet/>) MATLAB toolbox for convolutional neural networks. It provides routines for implementing linear convolution, back propagation, gradient descent, etc.

The main elements of MatConvNet are *computational blocks* and *wrappers*. Computational blocks are used for computing fundamental building blocks of a network (e.g. the small blocks in Figure 4). They can do linear convolution, local averaging, etc. A wrapper is used for specifying the structure of the network (in our case, a chain of blocks).

Since in almost all convolutional networks it is the pixel values of the filters rather than the dilation factors that are trained, we cannot use the differentiation functions in MatConvNet. We still have to compute the derivatives according to the formulas in Section 3.3.3.

We use the computational blocks and the data structures from the software. In our main program, we treat each filter (block) individually when we do the convolution. Conversely, MatConvNet treat filters in one layer as a batch. In each convolution operation, the filters are considered as 4-dimensional, where the first two dimensions are for height and width, the third dimension is for the number of channels (one in our case), and the fourth dimension is for the number of filters as a batch (three in our case).

In our validation, we track how the  $\lambda$ 's change over iteration steps. We start with

the same randomly generated 24  $\lambda$ 's and use 100 training data and iterate 100 steps in Algorithm 1. The 100 groups of  $\lambda$ 's are saved in a 24-by-100 matrix for comparison and they are the same for both programs.

Figure 10 illustrates how the  $\lambda$ 's are updated in 100 steps.

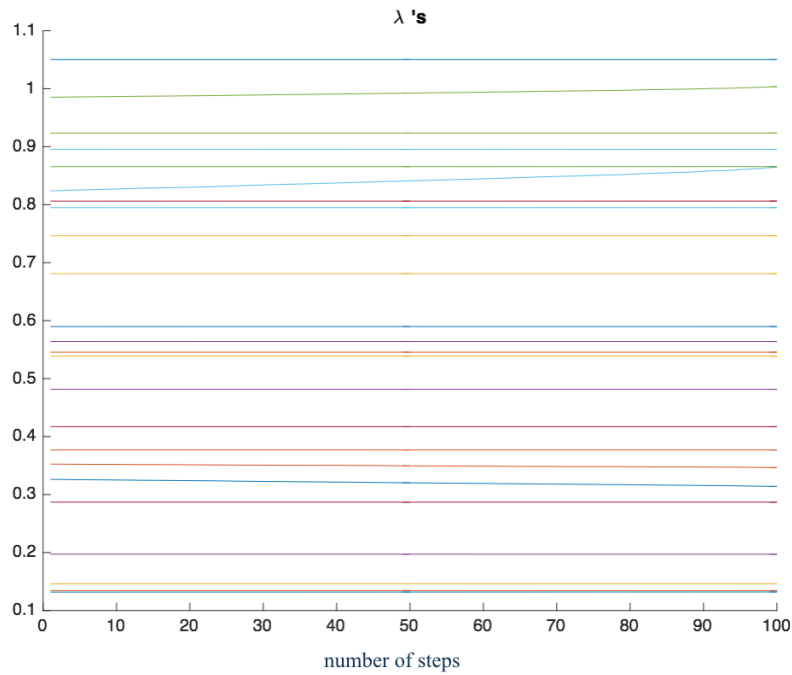


Figure 10: Change of  $\lambda$  values over 100 iteration steps

## 5 Testing

We have two parts for testing: the classification results and the stability results.

### 5.1 Classification Results

Figure 11 shows the binary classification results. It is surprising to see that running libSVM alone gives a perfect result. Nevertheless, the classification from the scattering network is satisfactory.

Figure 12 shows the multiclass classification results. We see that in this case a stochastic gradient descent method does not provide good result. The deterministic gradient descent,

error rate (%)	Class 0	Class 1
stochastic gradient descent	0.75	0
deterministic gradient descent	0.13	0.13
libSVM	0	0

Figure 11: Binary classification results

the libSVM and the deterministic gradient descent using the square function perform equally reasonably well.

error rate (%)	Class 0	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Class 8	Class 9
stochastic gradient descent	11.5	2.75	32.87	49.88	39.12	42.63	21.62	38.5	38.37	41
deterministic gradient descent	1.87	1.12	6	8.25	5.5	10.25	4.5	8	12	10.87
libSVM	3	1.62	6.25	7.5	4.87	9.37	5.25	7.75	10.87	10
Square (deterministic)	1.63	1.25	6.12	7.88	4.25	10.62	3.62	5.75	10.13	9.38

Figure 12: Multiclass classification results

## 5.2 Stability Results

### 5.2.1 Stability to Translation

In [8] the author proves that a scattering transform is translation invariant. However, the “invariance” is due to a limit behavior when the averaging is taken over the entire space domain. This is not a practical consideration. Nevertheless, it is still important to see how our scattering network response to a translation in the original image.

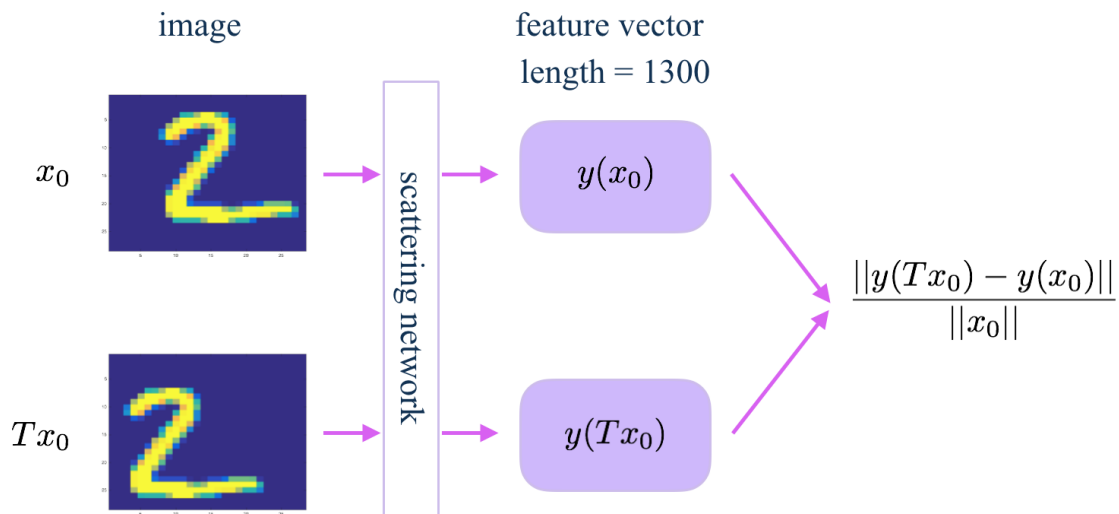


Figure 13: An illustration of the comparison process

Our experiment is illustrated as in Figure 13. For each image  $x_0$ , we apply a translation to get the translated image  $Tx_0$ . Letting them go through our scattering network we get the feature vectors  $y(x_0)$  and  $y(Tx_0)$ , respectively. We compare the norm of the difference of the feature vectors with the norm of the input image by taking the ratio

$$\frac{\|y(Tx_0) - y(x_0)\|}{\|x_0\|}.$$

Since our images are not continuous signals, we can only translate by integer-valued number of pixels. However, we would like to see the response to a small translation. Therefore, we do the fraction-valued translation in the following way. Suppose we would like to do translation of  $p/q$  pixels where  $p$  and  $q$  are coprime integers, we first upsample (linearly) the original 28-by-28 image to the size  $28q$ -by- $28q$ . We mark the positions of the original pixels, do the translation of  $p$  pixels, and then downsample to the original size by taking the pixels in the positions that we marked. An example of this process is illustrated in Figure 14.

We test the stability to translation for both using  $|\cdot|$  and using  $|\cdot|^2$  as the nonlinearity. We do the experiment for all the testing images and record the minimum, average and maximum of the ratios. The results are shown in Figure 15. We have small ratios for translation of a small number of pixels. We see that the ratio grows with the number of

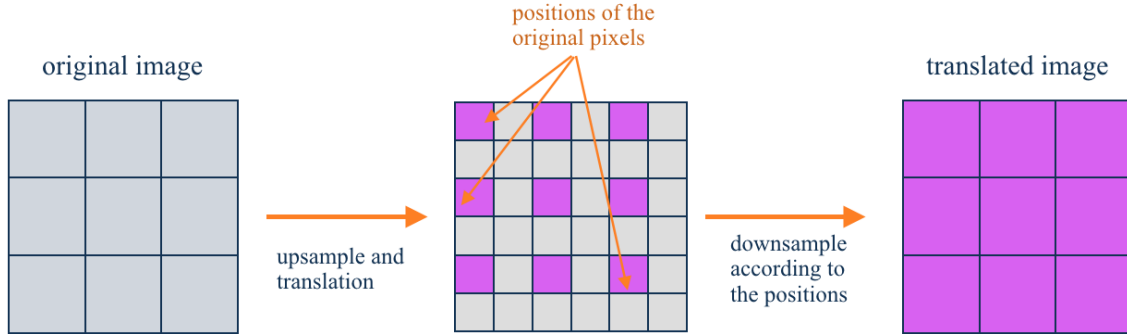


Figure 14: An illustration of translation of fractional pixels

pixels translated except that for  $|\cdot|$ , the translation by fractional pixels gives larger ratios than translation by one pixel. We believe that it is caused by the inaccuracy of our way to treat the fractional translation.

Translation (pixel)	min (%)	average (%)	max (%)
0.1	5.28	9.30	15.87
0.2	5.76	9.98	16.78
0.5	7.19	12.17	19.90
1	4.51	8.37	14.55
2	9.10	15.49	25.39
5	22.87	34.69	46.02

Translation (pixel)	min (%)	average (%)	max (%)
0.1	4.92	9.00	15.50
0.2	5.44	9.71	16.51
0.5	6.98	11.97	19.70
1	9.33	15.97	25.32
2	13.96	22.91	34.71
5	22.75	34.62	45.86

Figure 15: Test results for translation stability:  $\|y(Tx_0) - y(x_0)\| / \|x_0\|$

### 5.2.2 Stability to Deformation

For an image  $x_0 = x_0(t)$ , in [8] the deformed image is defined to be  $D(x_0) = D(x_0)(t) = x_0(t - \tau(t))$  for some smooth  $\tau$ . However this is not the most general definition fo deformation

in pattern theory. In our experiment we compute the ratio

$$\frac{\|y(D(x_0)) - y(x_0)\|}{\|x_0\|}$$

for two types of deformation  $D$ . The process is identical to what is done in Figure 13, with  $Tx_0$  be replaced with  $D(x_0)$ .

The first type of deformation is first upsample and then randomly downsample to the original size. Suppose the upsample factor is  $L$ , then we first do an upsampling (linearly) to get an image of size  $28L$ -by- $28L$ . Note that we still have 28-by-28 blocks of  $L$ -by- $L$  pixels. We then randomly pick one pixel from each block to compose a image of size 28-by-28. Figure 16 is illustration of this process.

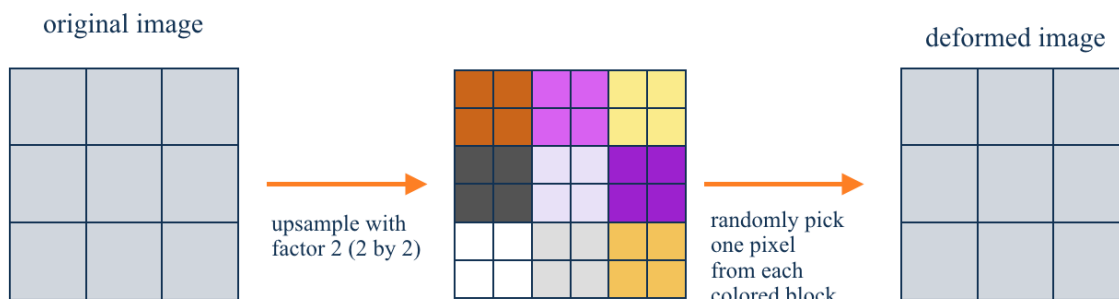


Figure 16: An illustration of deformation of Type 1

Figure 17 shows the results. We see that the ratio  $\|y(D(x_0)) - y(x_0)\| / \|x_0\|$  is small for both  $|\cdot|$  and  $|\cdot|^2$ . The ratio grows slightly with the increase of upsampling factor.

The second type of deformation is called *incomplete deformation* ([6]) in pattern theory. The deformed version of image  $D(x_0)$  is formed by setting certain pixel values to be zero in  $x_0$ . We do the experiment for (1). randomly zero out several pixels; (2) zero out a complete column; (3) zero out a complete row. The results are shown in Figure 18.



$ \cdot $				$ \cdot ^2$			
upsample factor	min (%)	average (%)	max (%)	upsample factor	min (%)	average (%)	max (%)
2	2.45	4.71	8.99	2	5.78	11.28	17.92
3	3.08	6.02	11.30	3	6.13	12.37	19.34
4	6.46	12.96	20.20	4	6.42	12.93	20.14
5	6.53	13.29	21.15	5	6.58	13.26	20.88

Figure 17: Test results for deformation stability:  $\|y(D(x_0)) - y(x_0)\| / \|x_0\|$  (Type 1)

$ \cdot $				$ \cdot ^2$			
test	min (%)	average (%)	max (%)	test	min (%)	average (%)	max (%)
randomly zero out	4.37	9.71	15.86	randomly zero out	4.36	9.21	14.73
take a column out	7.41	11.22	20.03	take a column out	7.27	10.95	19.88
take a row out	6.85	9.89	15.52	take a row out	6.42	9.55	15.15

Figure 18: Test results for deformation stability:  $\|y(D(x_0)) - y(x_0)\| / \|x_0\|$  (Type 2)

## 6 Project Schedule and Milestones

- September - October 2015: Define the project. Investigate in existing literature. Design the algorithm. (finished)
- November 2015: Write codes for training the convolutional filters. (finished)
- December 2015: Write codes for two-class classification. (finished)

- December 2015: Write codes for multi-class classification. (finished)
- February 2016: Complete validation. (finished)
- March 2016: Complete testing. (finished)
- April 2016: Wrap up the project. (finished)

## 7 Deliverables

1. Datasets: MNIST retrieved from <http://yann.lecun.com/exdb/mnist/>. The four files are the training data, the training labels, the testing data, the testing labels, respectively.
2. Toolboxes:
  - (1) LibSVM Ver 3.20, retrieved from <https://www.csie.ntu.edu.tw/~cjlin/libsvm/>. For the Mac system, MATLAB 2015b does not detect Xcode7, the solution is to install the patch *xcode7\_mexopts.zip*.
  - (2) MatConvNet Ver 1.0beta17, retrieved from <http://www.vlfeat.org/matconvnet/>. The installation process is contained in the MATLAB code.
3. MATLAB codes: The folder “MATLAB” contains the training and testing codes. The training and testing can be both done from *main.m*. The folder “Validation” contains the validation codes.
4. Trained network: We have a lot of saved data from the training results and the sizes are large. We provide the trained network for (1) the binary classification (2) the multiclass classification and (3) the multiclass classification using the square function as nonlinearity. The associated testing results are also included, respectively.
5. Documents including the proposal, the mid-year and final reports, the presentation slides, etc.

## References

- [1] Y. Bengio, I. J. Goodfellow and A. Courville, *Deep Learning*, Book in preparation for MIT press, 2015.
- [2] C. M. Bishop, *Pattern Recognition and Machine Learning*, New York: Springer, 2006.
- [3] J. Bruna et al., *A Theoretical Argument for Complex-Valued Convolutional Networks*, submitted, 2015.
- [4] J. Bruna and S. Mallat, *Invariant Scattering Convolution Networks*, Pattern Analysis and Machine Intelligence, IEEE Transactions 35(8) (2013), 1872–1886.
- [5] C-C. Chang and C-J. Lin, *LIBSVM : a library for support vector machines*. ACM Transactions on Intelligent Systems and Technology 27(2) (2011), 1–27. Software available at <http://www.csie.ntu.edu.tw/~cjlin/libsvm>.
- [6] U. Grenander, *Elements of Pattern Theory*. The Johns Hopkins University Press (1996).
- [7] C-W. Hsu and C-J. Lin, *A Comparison of Methods for Multiclass Support Vector Machines*, Neural Networks, IEEE Transactions 13(2) (2002), 415–425.
- [8] S. Mallat, *Group Invariant Scattering*, Comm. Pure Appl. Math., 65 (2012): 1331-1398.
- [9] A. Krizhevsky, I. Sutskever, G. E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, Advances in Neural Information Processing Systems 2 (2012): 1097-1105.
- [10] C. Szegedy et al, *Going Deeper with Convolutions*, Open Access Version available at [http://www.cv-foundation.org/openaccess/content\\_cvpr\\_2015/papers/Szegedy\\_Going\\_Deeper\\_With\\_2015\\_CVPR\\_paper.pdf](http://www.cv-foundation.org/openaccess/content_cvpr_2015/papers/Szegedy_Going_Deeper_With_2015_CVPR_paper.pdf), retrieved October 9, 2015.
- [11] A. Vedaldi and K. Lenc, *MatConvNet - Convolutional Neural Networks for MATLAB*, Proc. of the ACM Int. Conf. on Multimedia, 2015.
- [12] T. Wiatowski and H. Bölcskei, *Deep Convolutional Neural Networks Based on Semi-Discrete Frames*, Proc. of IEEE International Symposium on Information Theory (ISIT), Hong Kong, China, pp. 1212–1216, June 2015.