Image Recognition Problem with Convolutional Neural Network - Representation Learning Approach

ECS 271 Machine Learning
Final Project
Representation Learning Module

Yongkang Zhao 997032437
Shuxin Li 912525987

Image Recognition Problem with Convolutional Neural Network a Representation Learning Approach

By Yongkang Zhao, Shuxin Li

Contents

1. Background

1.1.Problem

Image recognition problem refers to the process of classifying raw image using Machine Learning techniques.
In the project, we want to analyze MNIST handwritten digit recognition problem. Handwritten digit recognition is an important problem in optical character recognition, and it has been used as a test case for theories of pattern recognition and machine learning algorithms for many years. Recent advancement in representation learning approach, deep learning methods yields the state-of-the-art results, namely the convolutional neural network (CNN). However, we only know the result that CNN has the best performance in the digit recognition problem, but we are not sure why properties of CNN produces the best performance. Therefore, we want to conduct some experiments to reveal and understand some properties of CNN. Properties that very helpful to digit recognition problem such as invariance of small translation and distortions to input data. Specifically, we focus on testing CNN's performance on classifying digital image when distortions exist.

1.2.Literature Survey

Convolutional neural network is inspired by the discovery of locally sensitive, orientation-selective neurons in the cat's visual system by Hubel and Wiesel [1]. The first implementation of a CNN was the so-called Neocognitron proposed by Fukushima [2,3,4,5], which has been originally applied to the problem of handwritten digit recognition. An important breakthrough of CNNs came with the widespread use of the Back-propagation learning algorithm. LeCun et al. [6] presented the first CNN that was trained by Back-propagation and applied it to the problem of handwritten digit recognition, obtaining state-of-the-art performance [7, 8]. Very recently, the convolutional structure has been imported into RBMs [9] and DBNs [10]. An important innovation [10] is the design of a generative version of the pooling / subsampling units, which worked beautifully in the experiments reported, yielding state-of-the- art results not only on MNIST digits but also on the Caltech-101 object classification benchmark.

2. Method

2.1.Approach

CNN Architecture

We conduct CNN algorithm to analyze MNIST handwritten digit recognition problem. CNN is a multi-layer perceptron that is the special design for identification of two-dimensional image information. CNN usually contains five kinds of layers: input layer, convolution layer, subsampling layer, fully connected hidden layer and output layer. The convolution layer, subsampling layer and fully connected hidden layer can be multiple based on different problems. The layers of a CNN have neurons arranged in 3 dimensions: width, height and depth. The neurons inside a layer are only connected to a small region of the layer before it. Distinct types of layers, both locally and completely connected, are stacked to form CNN architecture. The input layer contains an image map as a 3-dimensional neuron. The convolutional layer contains several feature maps (3-dimension) depending on the number and size of filters. The subsampling layer contains several sub-sampling feature maps (3-dimension) depending on previous convolutional layer and sub-region size. The fully connected hidden layer contains several 1-dimensional neurons, which is fully connected with previous layer. The output layer contains some neutrons (1-dimension) based on different goals.

CNN Properties

When the input dimension is high, as in images, for ANN, the number of connections or the number of free parameters is also high, because each hidden unit would be fully connected to the input layer. If the number of training examples is relatively small compared to complexity, then over-fitting will happen. CNN improve it because weight sharing and sparse connectivity structure, it can highly reduce the number of estimated parameters. Another property of CNN is its ability of invariance to small translations and local distortions. Finally, CNN takes into account correlations of neighboring input data exploited by local connectivity, and it can preferable extract local features and combine them subsequently.

CNN Implementation

Feed Forward:

When layer l is an input neuron layer, and layer $l+1$ is a convolutional layer
Then

$$V_j^{(l+1)} = \sum_{(u,v)\in K} w_j^{(l+1)}(u,v) y^{(l)}(x+u, y+v) + b_j^{(l+1)}$$

$$y_j^{(l+1)}(x,y) = \phi^{(l+1)}\left(V_j^{(l+1)}(x,y)\right)$$

where $\phi^{(l+1)}$ *is a linear activation function*

$$K = \{(u,v) \in N^2 \mid 0 \le u < C_u \ \ and \ \ 0 \le v < C_v\}$$

When layer l is convolutional layer, and layer $l+1$ is a subsampling layer

$$V_j^{(l+1)} = w_j^{(l+1)} \times \sum_{(u,v)\in K} y_i^{(l)}\left(S_u x + u, S_v y + v\right) + b_j^{(l+1)}$$

$$y_j^{(l+1)}(x,y) = \phi^{(l+1)}\left(V_j^{(l+1)}(x,y)\right)$$

where $\phi^{(l+1)}$ *is hyperbolic tangent function*

$$K = \{(u,v) \in N^2 \mid S_u - 1 \le u \le 0 \ and \ S_v - 1 \le v < 0\}$$

When layer l is a subsampling layer, and layer $l+1$ is a convolutional layer

$$V_j^{(l+1)} = \sum_{i \in I} \sum_{(u,v) \in K} w_{ji}^{(l+1)}(u,v) y_i^{(l)}(x+u, y+v) + b_j^{(l+1)}$$

$$y_j^{(l+1)}(x,y) = \phi^{(l+1)}\left(V_j^{(l+1)}(x,y)\right)$$

where $\phi^{(l+1)}$ *is a linear activation function*

$$K = \{(u,v) \in N^2 \mid 0 \le u < C_u \ \text{ and } \ 0 \le v < C_v\}$$

$$I = \{i \in N \mid \text{node } i \text{ is connected to node } j\}$$

When layer l is a subsampling layer, and layer $l+1$ is a neutron layer

$$V_j^{(l+1)} = \sum_i w_{ji}^{(l+1)} \cdot y_i^{(l)} + b_j^{(l+1)}$$

$$y_j^{(l+1)} = \phi^{(l+1)}\left(V_j^{(l+1)}\right)$$

where $\phi^{(l+1)}$ *is hyperbolic tangent function*

When layer l is a neutron layer, and layer $l+1$ is also a neutron layer

$$V_j^{(l+1)} = \sum_i w_{ji}^{(l+1)} \cdot y_i^{(l)} + b_j^{(l+1)}$$

$$y_j^{(l+1)} = \phi^{(l+1)}\left(V_j^{(l+1)}\right)$$

where $\phi^{(l+1)}$ *is hyperbolic tangent function*

Backward Propagation:

The loss function is

$$RSS = \frac{1}{2}\sum_j \left(y_j - t_j\right)^2$$

Where y_j is the prediction, and t_j is the true value.

When layer l is an output neutron layer and layer $l-1$ is a hidden neutron layer

$$\delta_j^{(l)} = \left(y_j^{(l)} - t_j^{(l)}\right)\phi'^{(l)}(V_j^{(l)})$$

$$\Delta w_{ji}^{(l)} = \frac{\partial RSS}{\partial w_{ji}} = \delta_j^{(l)} y_i^{(l-1)}$$

$$\Delta b_j^{(l)} = \delta_j^{(l)}$$

Then update $w_{ji}^{(l)}$ and $b_j^{(l)}$

$$w_{ji}^{(l)} = w_{ji}^{(l)} - \eta \Delta w_{ji}^{(l)}$$

$$b_j^{(l)} = b_j^{(l)} - \eta \Delta b_j^{(l)}$$

When layer l is a hidden neutron layer, and layer $l-1$ is a subsampling layer, also layer $l+1$ is an output neutron layer

$$\delta_j^{(l)} = \left(\sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)}\right)\phi'^{(l)}(V_j^{(l)})$$

$$\Delta w_{ji}^{(l)} = \delta_j^{(l)} y_i^{(l-1)}$$
$$\Delta b_j^{(l)} = \delta_j^{(l)}$$

where $\delta_k^{(l+1)}$ is the local gradient in output neutron layer $l+1$, and $w_{ji}^{(l+1)}$ is the weight from the node k to node j.

Then update $w_{ji}^{(l)}$ and $b_j^{(l)}$

$$w_{ji}^{(l)} = w_{ji}^{(l)} - \eta \Delta w_{ji}^{(l)}$$
$$b_j^{(l)} = b_j^{(l)} - \eta \Delta b_j^{(l)}$$

When layer l is a subsampling layer, and layer $l-1$ is a convolutional layer, also layer $l+1$ is a hidden layer

$$\delta_j^{(l)}(x,y) = \left(\sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)}(x,y) \right) \phi'^{(l)}(V_j^{(l)}(x,y))$$
$$\Delta w_{ji}^{(l)} = \sum_{(x,y)} \left(\delta_j^{(l)}(x,y) \sum_{(u,v)\in K} y_i^{l-1}(S_u x + u, S_v y + v) \right)$$
$$\Delta b_j^{(l)} = \sum_{(x,y)} \delta_j^{(l)}(x,y)$$
$$K = \{(u,v) \in \square^2 | S_u - 1 \le u \le 0 \; and \; S_v - 1 \le v < 0\}$$

where $\delta_k^{(l+1)}$ is the local gradient in hidden neutron layer $l+1$, and $\delta_k^{(l+1)}$ is the weight matrix from the node k in layer $l+1$ to subsampling map j.

When layer l is a convolutional layer, and $l-1$ is a subsampling layer or input layer, and layer $l+1$ is a subsampling layer.

$$\delta_j^{(l)}(x,y) = \delta_k^{(l+1)}\left(\lfloor x/S_u \rfloor, \lfloor y/S_v \rfloor \right) \cdot w_{kj}^{(l+1)} \cdot \phi'^{(l)}\left(V_j^{(l)}(x,y) \right)$$
$$\Delta w_{ji}^{(l)}(u,v) == \sum_{(x,y)} \left(\delta_j^{(l)}(x,y) \cdot y_i^{(l-1)}(x+u, y+v) \right)$$
$$\Delta b_j^{(l)} = \sum_{(x,y)} \delta_j^{(l)}(x,y)$$

where $\delta_k^{(l+1)}$ is the local gradient in subsampling layer $l+1$, and $w_{kj}^{(l+1)}$ is the weight from subsampling map k to feature map j.

Then update $w_{ji}^{(l)}$ and $b_j^{(l)}$

$$w_{ji}^{(l)} = w_{ji}^{(l)} - \eta \Delta w_{ji}^{(l)}$$
$$b_j^{(l)} = b_j^{(l)} - \eta \Delta b_j^{(l)}$$

When layer l is a subsampling layer, and layer $l-1$ is a convolutional layer, also layer $l+1$ is a convolutional layer.

$$\delta_j^{(l)}(x,y)=\left(\sum_{k\in K_j}\sum_{u,v}\delta_k^{(l+1)}(x,y)\cdot w_{kj}^{(l+1)}(u,v)\right)\phi'^{(l)}\left(V_j^{(l)}(x,y)\right)$$

$$\Delta w_{ji}^{(l)}(u,v)==\sum_{(x,y)}\left(\delta_j^{(l)}(x,y)\cdot\sum_{(u,v)\in K}y_i^{(l-1)}(S_u x+u,S_v y+v)\right)$$

$$\Delta b_j^{(l)}=\sum_{(x,y)}\delta_j^{(l)}(x,y)$$

$$K=\{(u,v)\in N^2\mid S_u-1\le u\le 0\ and\ S_v-1\le v<0\}$$

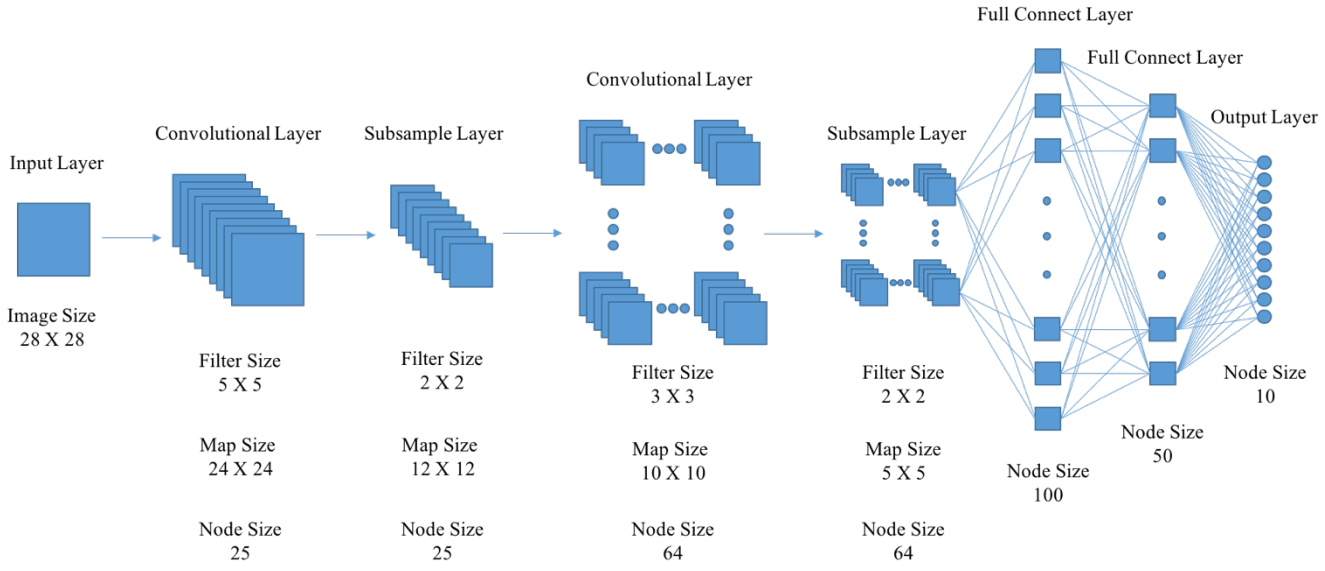
where $\delta_k^{(l+1)}$ is the local gradient in convolutional layer $l+1$, and $w_{kj}^{(l+1)}$ is the weight from feature map k to subsampling layer j.

Then update $w_{ji}^{(l)}$ and $b_j^{(l)}$

$$w_{ji}^{(l)}=w_{ji}^{(l)}-\eta\Delta w_{ji}^{(l)}$$

$$b_j^{(l)}=b_j^{(l)}-\eta\Delta b_j^{(l)}$$

Final System of Convolutional Neural Network

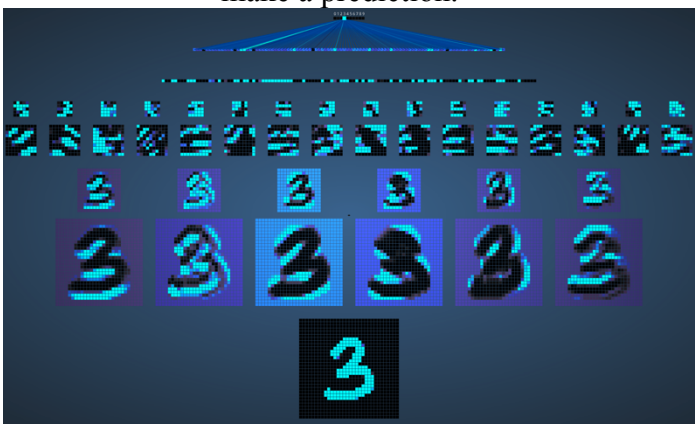


2.2.Rationale

One can think of Convolutional Neural Network as learning representation part and ANN part. Meaning, without directly feeding raw image data into the ANN for classification, instead, preprocess the image so that only important part of the image is left for the ANN to compute. One key advantage of Convolutional Neural Network is its ability to mitigate many types of distortion to the image. For example, an image representing number 3 from MNIST data set would look like this.

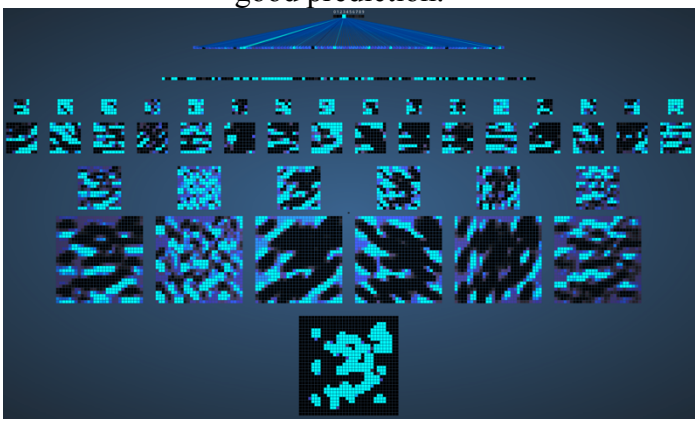
Without distortions

A convolutional neural network would have no problem extracting features from this image and make a prediction.



With distortions

Even with the noise, the same convolutional neural network will be able to filter out noise and make a good prediction.



We can compare both image for CNN's last two fully connected layers, they look very similar, that is because of the convolutional process has successfully filtered out noise, and features then recombined to reconstruct the image. In short, convolutional neural network tries to find filters that can extract important features from image then combines these extracted features to form a new image for a simple ANN to classify. This example represents the most important properties of CNN, which is weight sharing, local connectivity and 3-D neutrons. That is, they automatically learn local feature extractors, they are invariant to small translation and distortions in the input pattern, and they implement the principle of weight sharing which drastically reduces the number of free parameters and thus increases their generalization capacity compared to ANN architectures without this property. We implement CNN with back-propagation. Notice that convolutional neural networks have been discovered some time before LeCun's implementation, but what is so special about LeCun's implementation was that LeCun's method incorporated with back-propagation which enables an iterative method to have the algorithm discovered sets of filters suited for the classification problem.

3. Plan

3.1 Hypothesis

Our hypothesis is that CNN is invariant to small distortion of image, and can have better performance than other machine learning methods when distortion happen.

3.2 Experimental Design

The test on the hypothesis of CNN's ability to outperform other methods in terms of prediction of distorted images will be conducted using simulated data based on data from MNIST dataset.

Description of Data

Training data is going to be generated by randomly picking 100 samples for each class from the MNIST dataset. Each sample in the MNIST dataset is an image of a handwritten digit in size of 28 by 28, and is converted in to a matrix representation with value 0 representing whitest area of an image, and 255 representing the darkest area of an image. Testing data will be simulated based on 100 non training data from the MNIST dataset 10 samples for each class. Five new data sets will be created with each distortion method applied to every testing data, for testing and evaluating model's performance.

Type of simulated distortions

Increase in noise
Replace some pixels of images with random patches of dots or lines

Change in scale
Zoom in or out with some small ratio

Rotation of object
Clock wise and counter clock wise rotation from 1 to 90 degrees

Change in object location
Randomly shifting image up down left right so the image is no longer centered

Change in level of contrast
Fully or partially whiten or darken image

List of Models used for comparting performance:

Support Vector Machine
- *SVM-Linear*
- *SVM-RBF*
- *SVM-Polynomial*

K-Nearest Neighbor
- *KNN-Euclidean-50*
- *KNN-Euclidean-10*
- *KNN-Euclidean-1*

Artificial Neural Network:
- *ANN-1 Hidden layer 10 Nodes*
- *ANN-1 Hidden layer 50 Nodes*
- *ANN-1 Hidden layer 100 Nodes*

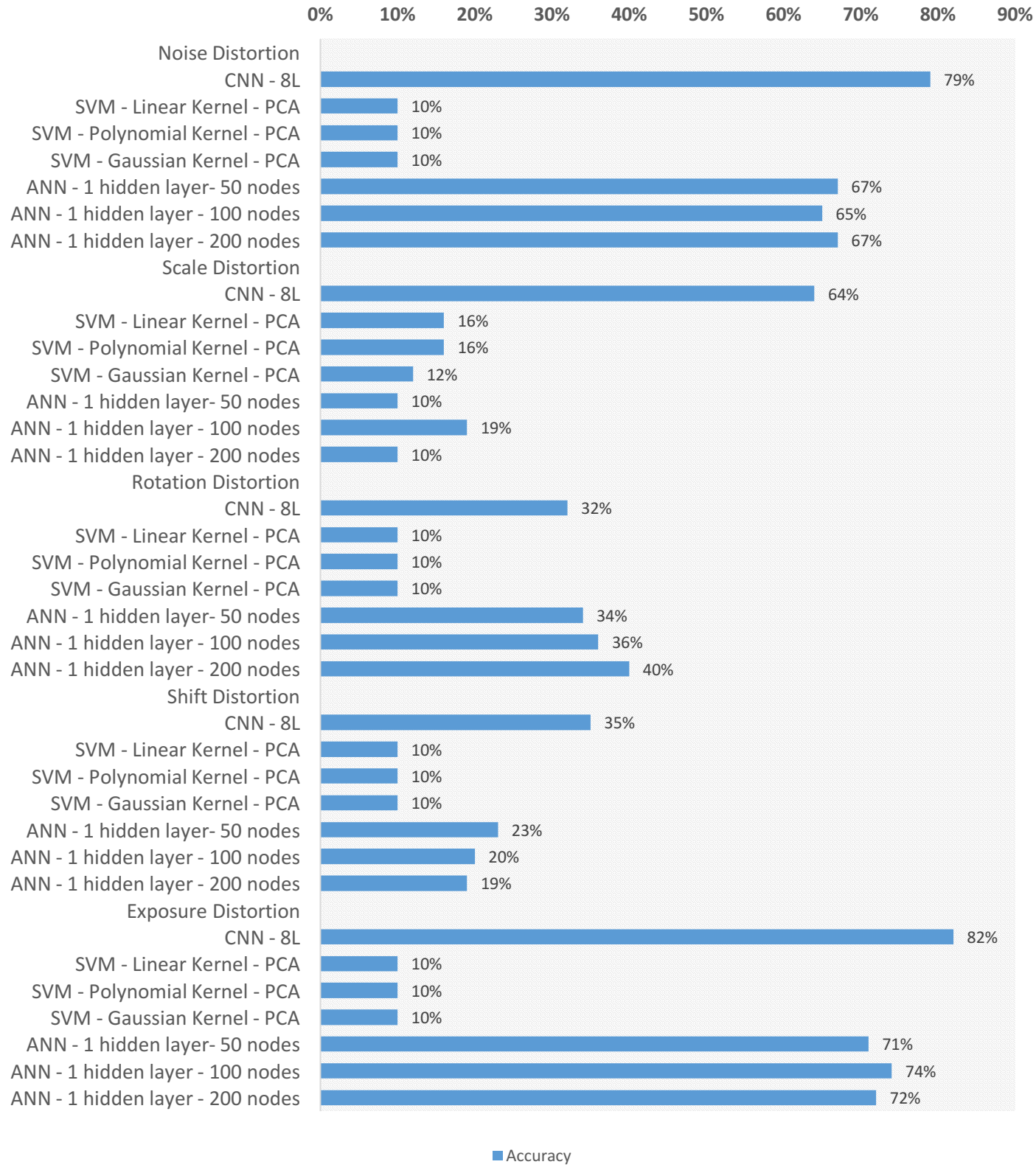
Description of Evaluation Process

For every model, the same training data is used to train. And for evaluation, every model is tested with every testing dataset. Evaluation score is the percentage of image being correctly classified.

4. Experiment

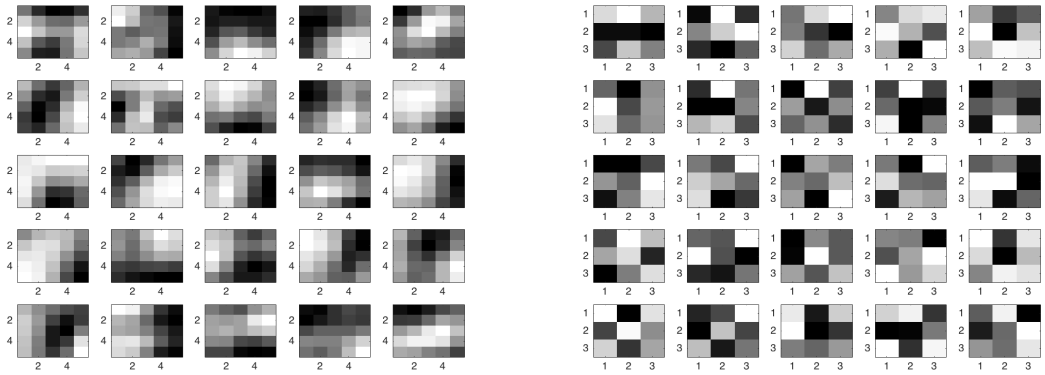
4.1. Results

As we can see from the table below, when dealing with distortion, CNN has significant more advantage against other models in most cases except distortion of rotation.



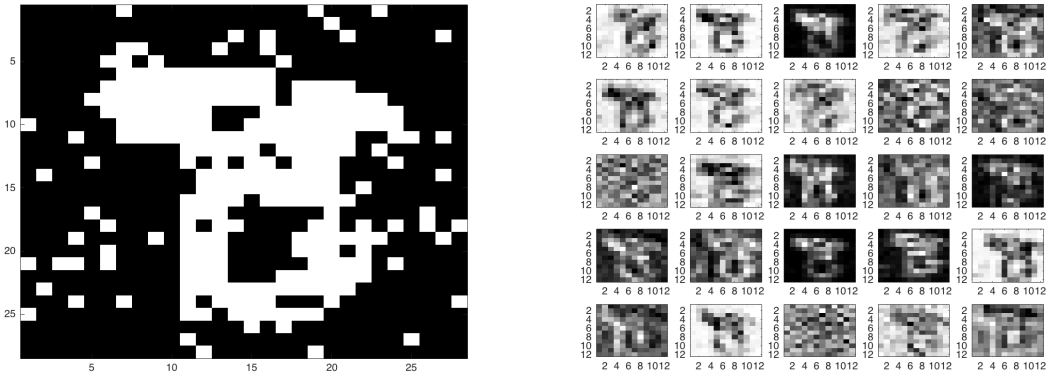
4.2. Critical Evaluation

The experimental result shows that CNN model is a more robust model. Although rotation distortion performed worst other ANN, we believe the main reason is that the random sparse connection in the CNN architecture may not be optimally configured, because we didn't perform cross validation to select a better architecture. However, from the results CNN outperform on other distortion types, which shows its strong generalization capacity. In this section we present how CNN takes advantage of filtering methods to out preform other models. Below are the filters our final system has learnt and use to help classifying image.



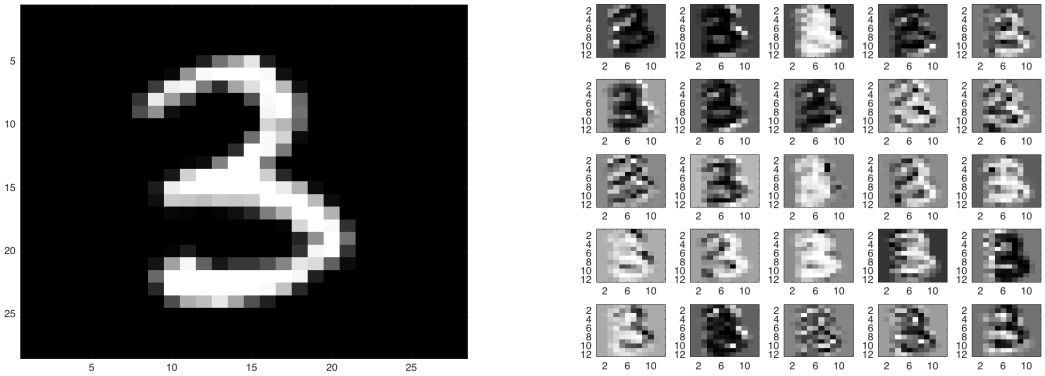
The first set of filters seems to be capturing edges and lines, while the second set of filter seems too abstracted to understand. We can see how well these filters can perform against unseen distorted images with images below.

Image with noise



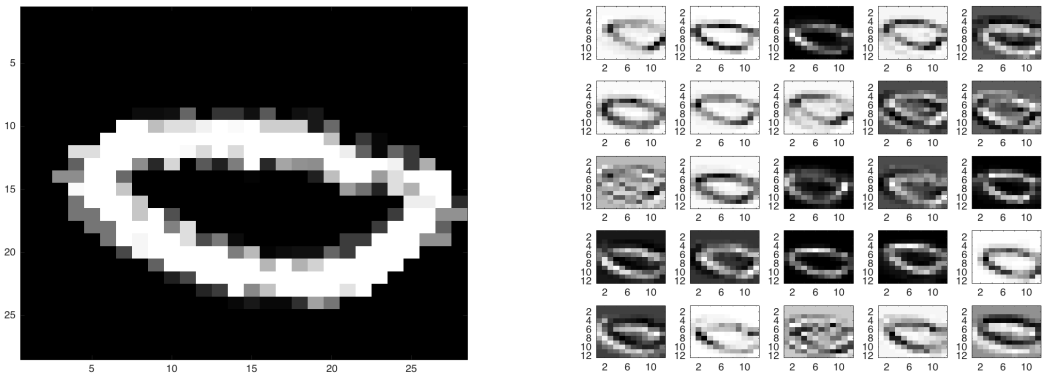
We can see the filtered feature map on the right has significantly reduced the noise on the plot.

Image with change in scale



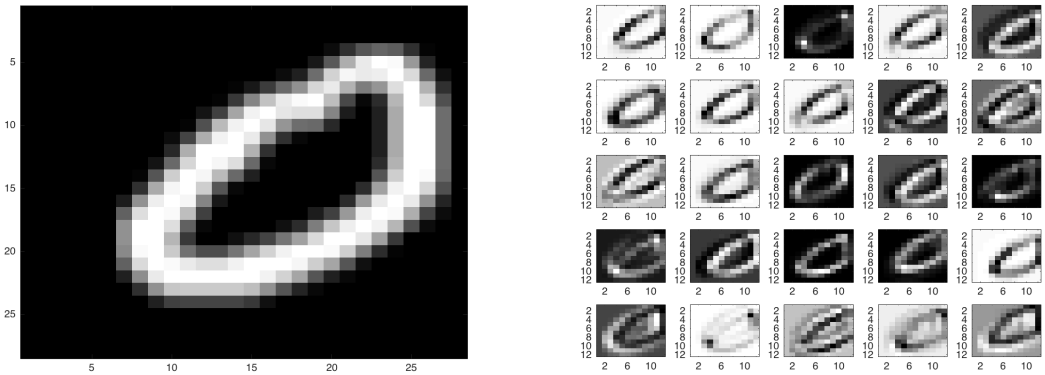
We can see the image is down scaled a bit, but filtering can still center the important features in the feature map.

Image with Rotation



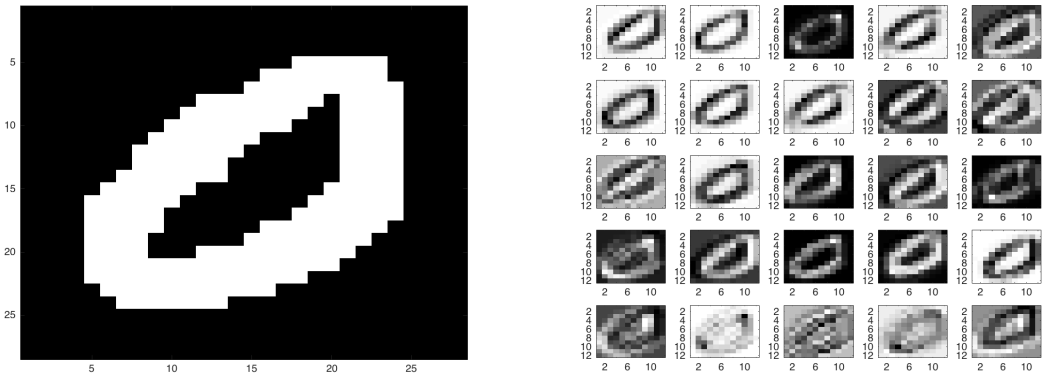
This set of images shows how CNN isn't able to rotate the rotated image to a form that is used in the training dataset. Thus resulting poor performance.

Image with shift in location



We can see the image is shifted to the right side by a small amount, and filters will have no problem identifying features.

Image with change in contrast



Seems severe change in contrast has very little effect on filtering.

5. Conclusion

5.1. Lessons Learned

In this project, we focus on testing CNN's performance on classifying digital image when distortion exists. We conducted an experiment to compare the ability to filter out nonessential information among several machine learning methods. The experiment result shows that CNN is a more robust model for classifying distorted image dataset. From the project, we learnt CNNs' nice properties, and we learn how to implement CNN with back-propagation, also we learn that the most important ideas in CNN are weight sharing, local connectivity, sparse connectivity, and 3-D neutrons.

6. Reference

[1] D. Hubel and T. Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. Journal of Physiology, 160:106–154, 1962.
[2] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," Biological Cybernetics, vol. 36, pp. 193–202, 1980.
[3] K. Fukushima. A neural-network model for selective attention in visual pattern recognition. Biological Cybernetics, 55:5–15, 1986.
[4] K. Fukushima. Analysis of the process of visual pattern recognition by the neocognitron. Neural Networks, 2:413–421, 1989.
[5] K. Fukushima and T. Imagawa. Recognition and segmentation of con- nected characters with selective attention. Neural Networks, 6:33–41, 1993.
[6] Y. LeCun, B. Boser, J.S. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Handwritten digit recognition with a back-propagation net- work. In David Touretzky, editor, Advances in Neural Information Processing Systems 2, pages 396–404. Morgan Kaufman, Denver, CO, 1990.
[7] Y. Le Cun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, 1998.
[8] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hub- bard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," Neural Computation, vol. 1, no. 4, pp. 541–551, 1989.
[9] G.DesjardinsandY.Bengio,"EmpiricalevaluationofconvolutionalRBMsfor vision," Technical Report 1327, D´epartement d'Informatique et de Recherche Op´erationnelle, Universit´e de Montr´eal, 2008.
[10] H. Lee, R. Grosse, R. Ranganath, and A. Y. Ng, "Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations," in Proceedings of the Twenty-sixth International Conference on Machine Learn- ing (ICML'09), (L. Bottou and M. Littman, eds.), Montreal (Qc), Canada: ACM, 2009.

7. Code Appendix

```
%% Reset environment
clear; close all; clc;
addpath('forward', 'backward');
addpath(genpath('DeepLearningToolbox'));

%addpath('AnimalFaceClassification/dataset/AnimalFace');
%% Load Data
load('mnist_uint8');

% training data
labels = unique(train_y, 'rows');
index = randperm(size(train_x,1));
train_x = train_x(index,:);
train_y = train_y(index,:);


training = cell(1);
trainingLabels = cell(1);

for i = 1:size(labels,1)
    counter = 0;
    for n = 1:size(train_y)
        if counter < 10
            if(sum(labels(i,:) == train_y(n,:)) == 10)
                training{i,1}(n,:) = train_x(n,:);
                trainingLabels{i,1}(n,:) = train_y(n,:);
                counter = counter +1;
            end
        end
    end
    training{i,1} = training{i,1}(sum(training{i,1},2) ~= 0,:);
    trainingLabels{i,1} = trainingLabels{i,1}(sum(trainingLabels{i,1},2) ~= 0,:);
end

training = double(cell2mat(training));
trainingLabels = double(cell2mat(trainingLabels));



%testing data
labels = unique(test_y, 'rows');
index = randperm(size(test_x,1));
test_x = test_x(index,:);
test_y = test_y(index,:);


testing = cell(1);
testingLabels = cell(1);

for i = 1:size(labels,1)
    counter = 0;
    for n = 1:size(test_y)
        if counter < 10
            if(sum(labels(i,:) == test_y(n,:)) == 10)
                testing{i,1}(n,:) = test_x(n,:);
                testingLabels{i,1}(n,:) = test_y(n,:);
                counter = counter +1;
            end
        end
```

```
    end
    testing{i,1} = testing{i,1}(sum(testing{i,1},2) ~= 0,:);
    testingLabels{i,1} = testingLabels{i,1}(sum(testingLabels{i,1},2) ~= 0,:);
end

testing = double(cell2mat(testing));
testingLabels = double(cell2mat(testingLabels));

%% distortion 1 - noise
noiseTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imnoise(reshape(testing(i,:),28,28), 'salt & pepper',0.2);
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    noiseTest(i,:) = X;
end
%% distortion 1 - scale
scaleTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imresize(reshape(testing(i,:),28,28), 1+rand*0.1);
    X = imcrop(X,[round(size(X)/2)-28/2 27 27]);
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    scaleTest(i,:) = X;
end
%% distortion 1 - rotation
rotationTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imrotate(reshape(testing(i,:),28,28),(rand*2-1)*90);
    X = imcrop(X,[round(size(X)/2)-28/2 27 27]);
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    rotationTest(i,:) = X;
end
%% distortion 1 - location
locationTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imtranslate(reshape(testing(i,:),28,28),[4*(rand*2-1), 4*(rand*2-
1)],'FillValues',0);
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    locationTest(i,:) = X;
end
%% distortion 1 - light exposure
exposureTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imadjust(reshape(testing(i,:),28,28));
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    exposureTest(i,:) = X;
end

%% Load Data
% meaning of y := [straight, left, right, up, neutral, happy, sad, angry, open,
sunglasses]
%[X, Y] = PrepData('cmufaces/FULL');
%im2double(rgb2gray(imread()));
%cifar 10

animaltype = ls('AnimalFaceClassification/dataset/AnimalFace/');
animaltype = textscan( animaltype, '%s', 'delimiter', '\t\n' );
animaltype = animaltype{1}(~cellfun('isempty',(animaltype{1})));
% read image
```

```
image = cell(1);
label = cell(1);
for t = 1:length(animaltype)
    imagefilenames = ls(sprintf('%s%s','AnimalFaceClassification/dataset/AnimalFace/',
animaltype{t}));
    imagefilenames = textscan( imagefilenames, '%s', 'delimiter', '\t\n' );
    imagefilenames = imagefilenames{1}(~cellfun('isempty',(imagefilenames{1})));
    image{t,1} = zeros(length(imagefilenames), 22500);
    label{t,1} = zeros(length(imagefilenames), 1);
    for f = 1:length(imagefilenames)
       X = imread(sprintf('%s%s/%s','AnimalFaceClassification/dataset/AnimalFace/',
animaltype{t}, imagefilenames{f}));
       if size(X,1) == 150 && size(X,2) == 150 && size(X,3) == 3
        image{t}(f,:) = reshape(rgb2gray(X),1,22500);
        label{t,1}(f,1) = t;
       end
       image{t} = image{t}(sum(image{t},2) ~= 0,:);
       label{t,1} = label{t,1}(label{t,1} ~= 0);
    end
end

% remove category with sample count < 98;
for i = 1:length(image)
    tf(i) = size(image{i},1) >= 98;
end
image = image(tf,:);
label = label(tf);

% combine data
X = cell2mat(image);
Y = cell2mat(label);
% define Y
label_animal = animaltype(unique(Y));
labels = unique(Y);
for i = 1:length(labels)
   Y(Y == labels(i)) = i;
end
% convert Y into matrix
Y_matrix = zeros(size(Y,1), length(labels));
for i = 1:size(Y_matrix,1)
    Y_matrix(i,Y(i)) = 1;
end
Y = Y_matrix;
% Normalize  X   [-1, 1]
X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
% rescale Y
Y = Y * 2 - 1;

%% setup training and testing data
% normalize training
for i = 1:size(training,1)
    X = training(i,:);
    training(i,:) = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
end
% rescale label
trainingLabels = trainingLabels * 2 - 1;
%% Define Convolutional Neural Network Layer Setups
% define subsampling to convolutional layer connection relation
sc_connection = cell(1);

sc_connection{4} = zeros(25,16);
while(sum(sum(sc_connection{4}) == 0) > 0)
    for i = 1:25
```

```
        sc_connection{4}(i,randperm(16,2)) = 1;
    end
end


sc_connection{6} = zeros(16,25);
while(sum(sum(sc_connection{6}) == 0) > 0)
    for i = 1:16
        sc_connection{6}(i,randperm(25,4)) = 1;
    end
end

sc_connection{8} = zeros(16,25);
while(sum(sum(sc_connection{8}) == 0) > 0)
    for i = 1:16
        sc_connection{8}(i,randperm(25,2)) = 1;
    end
end


cnn.layer_def = {
    struct('type', 'input layer'      , 'layersize',   1, 'nodesize'    , [28 28]) %input
layer
    struct('type', 'convolution layer', 'layersize',   25, 'filtersize'  , [5 5])
%convolution layer
    struct('type', 'subsample layer'  , 'layersize',   25, 'scale'       , 2)       %sub
sampling layer
    struct('type', 'convolution layer', 'layersize',   16, 'filtersize'  , [3 3])
%convolution layer
    struct('type', 'subsample layer'  , 'layersize',   16, 'scale'       , 2)
%subsampling layer
    %struct('type', 'convolution layer', 'layersize',  25, 'filtersize'  , [4 4])
%convolution layer
    %struct('type', 'subsample layer'  , 'layersize',  25, 'scale'       , 2)
%subsampling layer
    %struct('type', 'convolution layer', 'layersize',  25, 'filtersize'  , [3 3])
%convolution layer
    %struct('type', 'subsample layer'  , 'layersize',  25, 'scale'       , 2)
%subsampling layer
    struct('type', 'output layer'     , 'layersize',  10, 'nodesize'    , 1)
%subsampling layer
};

%% initialize Layers

[cnn.layer, cnn.weight, cnn.bias]= initializer(cnn.layer_def);


%%
iter = 100;
X = training;
Y = trainingLabels;
index = randperm(size(X,1),2);

%%
alpha = 0.009;
for i = 1:iter
    disp(sprintf('Iteration %3d', i));
    cnn = cnnTrain(X(index,:), Y(index,:), cnn, sc_connection, alpha);
end
```

```
%% predict

pred = forwardProp(reshape(scaleTest(61,:), cnn.layer{1,3}(1), cnn.layer{1,3}(2))', cnn,
sc_connection);
figure(1)
colormap gray;
imagesc(pred.layer{1,2}{1})

figure(2)
colormap gray;
for p = 1:25
    subplot(5,5,p)
    imagesc(pred.layer{3,2}{p})
end
%% ANN
Y(Y == -1) = 0;
%%
% hidden layer(s)
hiddenlayer = [200];
nn = nnsetup([size(X,2) hiddenlayer size(Y,2)]);
nn.activation_function = 'sigm';
%%
nn.learningRate = 0.1;
opts.numepochs = 300;
opts.batchsize = 1;
%% Train NN
nn = nntrain(nn, X, Y , opts);
%% Predict training

Predictions = predict(nn.W, X);

for i = 1:size(Predictions,1)
Predictions(i,:)  = max(Predictions(i,:))  == Predictions(i,:);
end

Accuracy = sum(sum(Predictions == testingLabels,2) == 10)/100

%% Noise
Predictions = predict(nn.W, noiseTest);

for i = 1:size(Predictions,1)
Predictions(i,:)  = max(Predictions(i,:))  == Predictions(i,:);
end

Accuracy = sum(sum(Predictions == testingLabels,2) == 10)/100
%% Scale
Predictions = predict(nn.W, scaleTest);

for i = 1:size(Predictions,1)
Predictions(i,:)  = max(Predictions(i,:))  == Predictions(i,:);
end

Accuracy = sum(sum(Predictions == testingLabels,2) == 10)/100
%% Rotation
Predictions = predict(nn.W, rotationTest);

for i = 1:size(Predictions,1)
Predictions(i,:)  = max(Predictions(i,:))  == Predictions(i,:);
end

Accuracy = sum(sum(Predictions == testingLabels,2) == 10)/100
```

```
%% Shift
Predictions = predict(nn.W, locationTest);

for i = 1:size(Predictions,1)
Predictions(i,:)  = max(Predictions(i,:))  == Predictions(i,:);
end

Accuracy = sum(sum(Predictions == testingLabels,2) == 10)/100
%% Exposure
Predictions = predict(nn.W, exposureTest);

for i = 1:size(Predictions,1)
Predictions(i,:)  = max(Predictions(i,:))  == Predictions(i,:);
end

Accuracy = sum(sum(Predictions == testingLabels,2) == 10)/100
```

```
function [tempcnn] = cnnTrain(X, Y, cnn, sc_connection, alpha)
        tempcnn = cnn;
    for i = 1:size(X,1)
        tic
        %cnn = forwardProp(reshape(X(i,:), cnn.layer{1,3}(1), cnn.layer{1,3}(2)), cnn,
sc_connection); % for face
        tempcnn = forwardProp(reshape(X(i,:), tempcnn.layer{1,3}(1),
tempcnn.layer{1,3}(2))', tempcnn, sc_connection);

%         disp(tempcnn.layer{6,2});
%         disp(tempcnn.weight{2,1}{1});
        %%%%
        pred = tanh(cell2mat(tempcnn.layer{6,2})/2);
%         disp(cnn.weight{10,1}{1}(1,1:7) )
%         disp(cnn.weight{9,1})
%         disp(cnn.weight{8,1}{1}(1,1:3) )
%         disp(cnn.weight{7,1})
%         disp(tempcnn.weight{6,1}{1}(1,1:3) )
%         disp(tempcnn.weight{5,1})
%         disp(tempcnn.weight{4,1}{1}(1,1:5) )
%         disp(tempcnn.weight{3,1})
%         disp(tempcnn.weight{2,1}{1}(1,1:7) )
%
%




        %pred(pred > 0) = 1;
        %pred(pred <= 0) = -1;

        y = Y(i,:)';
        error = mean((pred - y).^2);
        disp(sprintf('Image %3.0d SSE %3.5f ',i,error));

         disp([pred y]);
%
%
%         toc
        %disp(y);
        figure(1)
        colormap gray;
        imagesc(tempcnn.layer{1,2}{1})

        figure(2)
        colormap gray;
        for p = 1:25
            subplot(5,5,p)
            imagesc(tempcnn.layer{3,2}{p})
        end

        figure(3)
        colormap gray;
        for p = 1:16
            subplot(4,4,p)
            imagesc(tempcnn.layer{5,2}{p})
        end
%
%         figure(4)
%         colormap gray;
%         for p = 1:25
%             subplot(5,5,p)
%             imagesc(cnn.layer{7,2}{p})
```

```
%         end
        figure(6)
        colormap gray;
        for p = 1:25
            subplot(5,5,p)
            imagesc(tempcnn.weight{2,p}{:})
        end

        figure(7)
        colormap gray;
        for p = 1:25
            subplot(5,5,p)
            imagesc(tempcnn.weight{4,1}{p})
        end
%
%         figure(5)
%         colormap gray;
%         counter = 0;
%         for p = 1:9
%             for pp = 1:length(tempcnn.weight{4,p})
%                 counter = counter +1;
%                 subplot(25,9,counter);
%                 imagesc(tempcnn.weight{4,p}{pp});
%             end
%         end
        %


%
%         figure(5)
%         colormap gray;
%         for p = 1:25
%             subplot(5,5,p)
%             imagesc(cnn.layer{9,2}{p})
%         end

         drawnow;

        %disp(cnn.layer{6,2}{1});
        tempcnn = backwardProp(Y(i,:), tempcnn, sc_connection, alpha);

    end
end
```

```
function [h, display_array] = displayData(X, example_width)
%DISPLAYDATA Display 2D data in a nice grid
%   [h, display_array] = DISPLAYDATA(X, example_width) displays 2D data
%   stored in X in a nice grid. It returns the figure handle h and the
%   displayed array if requested.

% Set example_width automatically if not passed in
if ~exist('example_width', 'var') || isempty(example_width)
    example_width = round(sqrt(size(X, 2)));
end

% Gray Image
colormap(gray);

% Compute rows, cols
[m n] = size(X);
example_height = (n / example_width);

% Compute number of items to display
display_rows = floor(sqrt(m));
display_cols = ceil(m / display_rows);

% Between images padding
pad = 1;

% Setup blank display
display_array = - ones(pad + display_rows * (example_height + pad), ...
                       pad + display_cols * (example_width + pad));

% Copy each example into a patch on the display array
curr_ex = 1;
for j = 1:display_rows
    for i = 1:display_cols
        if curr_ex > m,
            break;
        end
        % Copy the patch

        % Get the max value of the patch
        max_val = max(abs(X(curr_ex, :)));
        display_array(pad + (j - 1) * (example_height + pad) + (1:example_height), ...
                      pad + (i - 1) * (example_width + pad) + (1:example_width)) = ...
                        reshape(X(curr_ex, :), example_height, example_width) / max_val;
        curr_ex = curr_ex + 1;
    end
    if curr_ex > m,
        break;
    end
end

% Display Image
h = imagesc(display_array, [-1 1]);

% Do not show axis
axis image off

drawnow;

end
```

```
function [y] = dtan(x)

%     if(exp(x)==Inf)
%         y = 0;
%     else
%         y = 2*exp(x)/(1+exp(x))^2;
%     end
    y = 1./(1+cosh(x));

end


function [ layer, weights, bias] = initializer( layer_def )

    layer = cell(1);
    weights = cell(1);
    bias = cell(1);

    for i = 1:length(layer_def)
        if strcmp(layer_def{i}.type, 'input layer')
            % layer Size
            layer{i,1} = layer_def{i}.layersize;
            % allocate memory for input layer
            layer{i,2} = zeros(layer_def{i}.nodesize(1:2));
            % node size
            layer{i,3} = layer_def{i}.nodesize(1:2);

        elseif strcmp(layer_def{i}.type, 'convolution layer') && strcmp(layer_def{i-
1}.type, 'input layer')
            % layer Size
            layer{i,1} = layer_def{i}.layersize;
            % node size
            layer{i,3} = (layer{i-1,3} - layer_def{i}.filtersize) + 1;
            % allocate memory for convolution layer's V
            layer{i,2} = cell(layer_def{i}.layersize,1);
            for n = 1:length(layer{i,2})
                layer{i,2}{n} = zeros(layer{i,3});
                weights{i,n} = cell(layer{i-1,1},1);
                for k = 1:layer{i-1,1}
                    weights{i,n}{k} = rand(layer_def{i}.filtersize)*2-1;
                end

                bias{i,n} = rand(1)*2-1;
            end
        elseif strcmp(layer_def{i}.type, 'convolution layer') && strcmp(layer_def{i-
1}.type, 'subsample layer')
            % layer Size
            layer{i,1} = layer_def{i}.layersize;
            % node size
            layer{i,3} = (layer{i-1,3} - layer_def{i}.filtersize) + 1;
            % allocate memory for convolution layer's V
            layer{i,2} = cell(layer_def{i}.layersize,1);
            for n = 1:length(layer{i,2})
                layer{i,2}{n} = zeros(layer{i,3});
                weights{i,n} = cell(layer{i-1,1},1);
                for k = 1:layer{i-1,1}
                    weights{i,n}{k} = rand(layer_def{i}.filtersize)*2-1;
                end
                bias{i,n} = rand(1)*2-1;
            end
        elseif strcmp(layer_def{i}.type, 'subsample layer')
            % layer Size
```

```
            layer{i,1} = layer_def{i}.layersize;
            % allocate memory for subsample layer's V
            layer{i,2} = cell(layer_def{i-1}.layersize,1);
            for n = 1:length(layer{i,2})
                layer{i,2}{n} = zeros(layer{i-1,3}/layer_def{i}.scale);
                weights{i,n} = rand(1)*2-1;
                bias{i,n} = rand(1)*2-1;
            end
            % node size
            layer{i,3} = layer{i-1,3}/layer_def{i}.scale;

        elseif strcmp(layer_def{i}.type, 'output layer')
            % layer Size
            layer{i,1} = layer_def{i}.layersize;
            % allocate memory for output layer's V
            layer{i,2} = cell(layer_def{i}.layersize,1);
            % node Size
            layer{i,3} = [layer_def{i}.nodesize, 1];

            for k = 1:layer{i,1}
                weights{i,k} = cell(layer{i-1,1},1);
                for kn = 1:layer{i-1,1}
                    weights{i,k}{kn} = rand(layer{i-1,3})*2-1;
                end
            end
        end
    end

    layer{1,2} = cell(layer(1,2));
end


function [ A ] = predict(Ws, A )
    nsample = length(A);
    for i = 1:length(Ws)
        A = sigm([ones(nsample,1) A] * Ws{i}');
    end
end
```

```
function [ data, labels ] = PrepData( folderpath )
%% import image

%get image names
addpath(folderpath);
imagefilenames = ls(folderpath);
imagefilenames = textscan( imagefilenames, '%s', 'delimiter', '\t\n' );
imagefilenames = imagefilenames{1}(~cellfun('isempty',(imagefilenames{1})));

%%

%get image

data =
zeros(length(imagefilenames),size(imread(imagefilenames{1}),1)*size(imread(imagefilenames
{1}),2));
for i = 1:length(imagefilenames)
    %read in
    img = imread(imagefilenames{i});
    %vectorize
    img = reshape(img, [1,size(img,1)*size(img,2)]);
    data(i,:) = img;
end

clear i img;

%%
% prep label
% label = [straight, left, right, up, neutral, happy, sad, angry, open, sunglasses]
labels = zeros(length(imagefilenames), 10);
for i = 1:length(imagefilenames)
    text = textscan( imagefilenames{i}, '%s', 'delimiter', '_.' );
    text = text{1}(2:4);
    if strcmp(text{1}, 'straight')
            labels(i,1) = 1;
    elseif strcmp(text{1}, 'left')
            labels(i,2) = 1;
    elseif strcmp(text{1}, 'right')
            labels(i,3) = 1;
    elseif strcmp(text{1}, 'up')
            labels(i,4) = 1;
    else
            disp(i)
            disp(text{1})
    end

    if strcmp(text{2}, 'neutral')
            labels(i,5) = 1;
    elseif strcmp(text{2}, 'happy')
            labels(i,6) = 1;
    elseif strcmp(text{2}, 'sad')
            labels(i,7) = 1;
    elseif strcmp(text{2}, 'angry')
            labels(i,8) = 1;
    else
            disp(i)
            disp(text{2})
    end

    if strcmp(text{3}, 'open')
            labels(i,9) = 1;
    elseif strcmp(text{3}, 'sunglasses')
```

```
            labels(i,10) = 1;
    else
            disp(i)
            disp(text{3})
    end

end

end


function [ tempcnn ] = forwardProp(X, cnn , sc_connection)
    tempcnn = cnn;
    tempcnn.layer{1,2}{1} = X;
%     disp('Forward Propogating');
    for l = 1:(length(tempcnn.layer_def)-1)

        %disp(cnn.layer_def{l}.type);
        if strcmp(tempcnn.layer_def{l}.type, 'input layer') &&
strcmp(tempcnn.layer_def{l+1}.type, 'convolution layer')
            tempcnn = input_convolution(tempcnn, l);

        elseif strcmp(tempcnn.layer_def{l}.type, 'convolution layer') &&
strcmp(tempcnn.layer_def{l+1}.type, 'subsample layer')
            tempcnn = convolution_subsample(tempcnn, l);

        elseif strcmp(tempcnn.layer_def{l}.type, 'subsample layer') &&
strcmp(tempcnn.layer_def{l+1}.type, 'convolution layer')
            tempcnn = subsample_convolution(tempcnn, l, sc_connection{l+1});

        elseif strcmp(tempcnn.layer_def{l}.type, 'subsample layer') &&
strcmp(tempcnn.layer_def{l+1}.type, 'output layer')
            tempcnn = subsample_output(tempcnn, l);

        end
    end
end



function [ tempcnn ] = input_convolution(cnn, l)
tempcnn = cnn;
    filterNum = tempcnn.layer{l+1,1};

    V = cell(filterNum,1);

    for k = 1:filterNum
        V{k} = -1*conv2(tempcnn.layer{l,2}{1},tempcnn.weight{l+1,k}{1},'valid');
    end

    tempcnn.layer{l+1,2} = V;

end


function [ tempcnn ] = subsample_convolution( cnn, l, sc_connection)
    tempcnn = cnn;
    for i = 1:length(tempcnn.layer{l,2})
        Ym{i,1} = tanh(tempcnn.layer{l,2}{i}/2);
    end
```

```
    NumM = tempcnn.layer{l,1};
    NumI = tempcnn.layer{l+1,1};

    % all Yi
    MapC2 = cell(NumI, NumM);
    for i = 1:NumI
        for m = 1:NumM
            % sparse weights
            if(sc_connection(m,i) == 1)

                filter = tempcnn.weight{l+1,i}{m};
                y = zeros(size(Ym{m}) - (size(filter)-1));
                for i2 = 1:size(y,1)
                    for j2 = 1:size(y,2)
                        y(i2, j2) = sum(sum(filter.*Ym{m}(i2:i2+size(filter,1)-
1,j2:j2+size(filter,2)-1)));
                    end
                end
                MapC2{i,m} = y;
            end
        end
    end

    % some feature map combined
    Yi = cell(NumI,1);
    for i = 1:NumI
        Yi{i} = 0;
        for m =1:NumM
            if(length(MapC2{i,m})>0)
                Yi{i} = MapC2{i,m} + Yi{i};
            end
        end
    end


    for i = 1:NumI
        Yi{i} = Yi{i} + tempcnn.bias{l+1,i};
    end

    tempcnn.layer{l+1,2} = Yi;

end


function [ tempcnn ] = subsample_output( cnn , l)
tempcnn = cnn;
% Output is a matrix
Yj = tempcnn.layer{l,2};


NumJ = length(Yj);
NumOutput = tempcnn.layer{l+1,1};

% initial output
V = zeros(NumOutput,1);

Vk = cell(NumOutput,1);
for k=1:NumOutput
```

```
    for j=1:NumJ
        V(k) = V(k) + (sum(sum(tempcnn.weight{l+1,k}{j}.*tanh(Yj{j}/2))));
    end
    Vk{k} = V(k);
end

tempcnn.layer{l+1,2} = Vk;

end


function [ tempcnn ] = convolution_subsample(cnn, l)
%CONVOLUTION_SUBSAMPLE Summary of this function goes here
%   Detailed explanation goes here

    tempcnn = cnn;

    NumS1 = tempcnn.layer{l+1,1};

    V = cell(NumS1,1);

    for k = 1:NumS1
        reducedmap = tempcnn.layer{l+1,2}{k};
        featuremap = tanh((tempcnn.layer{l,2}{k})/2);
        for i = 1:size(featuremap,1)/2
            for j = 1:size(featuremap,2)/2
               reducedmap(i,j) = sum(sum(featuremap(((tempcnn.layer_def{l+1}.scale*i-
1):tempcnn.layer_def{l+1}.scale*i), (tempcnn.layer_def{l+1}.scale*j-
1:tempcnn.layer_def{l+1}.scale*j))));
            end
        end
        V{k} = tempcnn.weight{l+1,k} * (reducedmap/(tempcnn.layer_def{l+1}.scale^2)) +
tempcnn.bias{l+1,k};
    end

    tempcnn.layer{l+1,2} = V;
end


function [tempcnn] = Back_convolution_subsample(cnn, layer, alpha)
    %% from l-2 to l-3
    tempcnn  = cnn;
    deltaJ = tempcnn.layer{layer+1, 4};
    sub = tempcnn.layer_def{layer+1}.scale;

    for i = 1:tempcnn.layer{layer,1}
        for x = 1:tempcnn.layer{layer,3}(1)
            for y = 1:tempcnn.layer{layer,3}(2)
                deltaI{i}(x,y) = deltaJ{i}(round(x/sub),
round(y/sub))*tempcnn.weight{layer+1,i}*dtan(tempcnn.layer{layer,2}{i}(x,y));
            end
        end
    end

    for i=1:tempcnn.layer{layer,1}
        for m=1:tempcnn.layer{layer-1,1}
            for u = 0:tempcnn.layer_def{layer}.filtersize(1)-1
                for v = 0:tempcnn.layer_def{layer}.filtersize(2)-1
                    temp = 0;
                    for x = 1:tempcnn.layer{layer,3}(1)
                        for y = 1:tempcnn.layer{layer,3}(2)
```

```
                            temp = temp +  deltaI{i}(x,y)*tanh(tempcnn.layer{layer-
1,2}{m}(x+u, y+v)/2);
                        end
                    end
                    deltaWim{i,m}(u+1, v+1) = temp;
                end
            end
        end
        deltabi{i} = sum(sum(deltaI{i}));
    end


    for i=1:tempcnn.layer{layer,1}
        for m=1:tempcnn.layer{layer-1,1}
            if(length(tempcnn.weight{layer,i}{m})>0)
                'before'
                tempcnn.weight{layer,i}{m}
                tempcnn.weight{layer,i}{m} = tempcnn.weight{layer,i}{m} -
alpha*deltaWim{i,m};
                'after'
                tempcnn.weight{layer,i}{m}
                'delta'
                deltaWim{i,m}
            end
        end
        tempcnn.bias{layer,i} = tempcnn.bias{layer,i} - alpha*deltabi{i};
    end
    tempcnn.layer{layer, 4} = deltaI;
end

function [tempcnn] = Back_Propogation(cnn, y_true, layer, alpha)
% y_true is a matrix
% layer is current layer in backward propogation

    tempcnn  = cnn;
    for k=1:tempcnn.layer{layer,1}
        deltaK{k} = (tanh(tempcnn.layer{layer,2}{k}/2) -
y_true(k))*dtan(tempcnn.layer{layer,2}{k});

    end

    for k=1:tempcnn.layer{layer,1}
        for j= 1:tempcnn.layer{layer-1,1}
            %dell = alpha*deltaK{k}*tanh(tempcnn.layer{layer-1,2}{j}/2);
            %disp(dell(1:4,1:4));
            %dt = tanh(tempcnn.layer{layer-1,2}{j}/2);
            %disp(dt(1:4,1:4));

            tempcnn.weight{layer,k}{j} = tempcnn.weight{layer,k}{j} -
alpha*deltaK{k}*tanh(tempcnn.layer{layer-1,2}{j}/2);
            %disp(sum(sum(tempcnn.weight{layer,k}{j} == cnn.weight{layer,k}{j})));
        end
    end
    tempcnn.layer{layer,4} = deltaK;



end

function [tempcnn] = Back_subsample_convolution(cnn, layer, alpha, sc_connection)
    %% from layer l-3 to l-4
```

```
    tempcnn = cnn;
    deltaI = tempcnn.layer{layer+1, 4};
    sc_connection = sc_connection';

    for x= 1:tempcnn.layer{layer,3}(1)
        for y =1:tempcnn.layer{layer,3}(2)
            for m=1:tempcnn.layer{layer,1}
                temp = 0;
                for i=find(sc_connection(:,m)==1)'
                    for u=0:tempcnn.layer_def{layer+1}.filtersize(1)-1
                        for v=0:tempcnn.layer_def{layer+1}.filtersize(2)-1
                            temp = temp +
deltaI{i}(min(x,tempcnn.layer{layer+1,3}(1)),min(y,tempcnn.layer{layer+1,3}(2)))*tempcnn.
weight{layer+1,i}{m}(u+1, v+1);
                        end
                    end
                end
                deltaM{m}(x,y) = temp*dtan(tempcnn.layer{layer,2}{m}(x,y));
            end
        end
    end

    sub = tempcnn.layer_def{layer}.scale;
    for s=1:tempcnn.layer{layer,1}
        temp = 0;
        for x = 1:tempcnn.layer{layer,3}(1)
            for y = 1:tempcnn.layer{layer,3}(2)
                temp = temp + deltaM{s}(x,y) * sum(sum(tanh(tempcnn.layer{layer-
1,2}{s}((sub*(x-1)+1):(sub*x),(sub*(y-1)+1):(sub*y))/2)));
            end
        end
        tempcnn.weight{layer,s} = tempcnn.weight{layer,s} - alpha*temp;
        tempcnn.bias{layer,s} = tempcnn.bias{layer,s} - alpha*sum(sum(deltaM{s}));
    end

    tempcnn.layer{layer, 4} = deltaM;
end

function [tempcnn] = Back_subsample_output(cnn, layer, alpha)
    %% from l-1 to l-2
%     tempcnn = cnn;
%     deltaK = tempcnn.layer{layer+1, 4};
%
%     for j=1:tempcnn.layer{layer,1}
%         temp = 0;
%         for k=1:tempcnn.layer{layer+1,1}
%             temp = temp + deltaK{k}*sum(sum(tempcnn.weight{layer+1,k}{j}));
%         end
%         deltaJ{j} = temp*dtan(tempcnn.layer{layer,2}{j});
%     end
%
%     sub = tempcnn.layer_def{layer}.scale;
%     for j=1:tempcnn.layer{layer,1}
%         temp = 0;
%         for x = 1:tempcnn.layer{layer,3}(1)   % Node Size
%             for y = 1:tempcnn.layer{layer,3}(2)
%                 temp = temp + deltaJ{j}(x,y)* sum(sum(tanh(tempcnn.layer{layer-
1,2}{j}(sub*(x-1)+1:sub*x,sub*(y-1)+1:sub*y)/2)));
%             end
%         end
%         tempcnn.weight{layer,j} = tempcnn.weight{layer,j} - alpha*temp;
%         tempcnn.bias{layer,j} = tempcnn.bias{layer,j} - alpha*sum(sum(deltaJ{j}));
%     end
```

```
%
%     tempcnn.layer{layer, 4} = deltaJ;
% end


    %% from l-1 to l-2
    tempcnn = cnn;
    deltaK = tempcnn.layer{layer+1, 4};

    for j=1:tempcnn.layer{layer,1}
        for y = 1:tempcnn.layer{layer,3}(2)
            for x = 1:tempcnn.layer{layer,3}(1)
                temp = 0;
                for k=1:tempcnn.layer{layer+1,1}
                    temp = temp + deltaK{k}*tempcnn.weight{layer+1,k}{j}(x,y);
                end
                deltaJ{j} = temp*dtan(tempcnn.layer{layer,2}{j});
            end
        end
    end

    sub = tempcnn.layer_def{layer}.scale;
    for j=1:tempcnn.layer{layer,1}
        temp = 0;
        for x = 1:tempcnn.layer{layer,3}(1)   % Node Size
            for y = 1:tempcnn.layer{layer,3}(2)
                temp = temp + deltaJ{j}(x,y)* sum(sum(tanh(tempcnn.layer{layer-
1,2}{j}(sub*(x-1)+1:sub*x,sub*(y-1)+1:sub*y)/2)));
            end
        end
        tempcnn.weight{layer,j} = tempcnn.weight{layer,j} - alpha*temp;
        tempcnn.bias{layer,j} = tempcnn.bias{layer,j} - alpha*sum(sum(deltaJ{j}));
    end

    tempcnn.layer{layer, 4} = deltaJ;
end
function [ tempcnn ] = backwardProp( Y, cnn , sc_connection, alpha)
%     disp('Backward Propogating');
    tempcnn = cnn;
    for l = sort(2:length(tempcnn.layer_def),2, 'descend')

        %disp(cnn.layer_def{l}.type);
        if  strcmp(tempcnn.layer_def{l}.type, 'output layer') &&
strcmp(tempcnn.layer_def{l-1}.type, 'subsample layer')
            tempcnn = Back_Propogation(tempcnn, Y, l, alpha);

        elseif strcmp(tempcnn.layer_def{l+1}.type, 'output layer') &&
strcmp(tempcnn.layer_def{l}.type, 'subsample layer')
            tempcnn = Back_subsample_output(tempcnn, l, alpha);

        elseif strcmp(tempcnn.layer_def{l+1}.type, 'subsample layer') &&
strcmp(tempcnn.layer_def{l}.type, 'convolution layer')
            tempcnn = Back_convolution_subsample(tempcnn, l, alpha );

        elseif strcmp(tempcnn.layer_def{l+1}.type, 'convolution layer') &&
strcmp(tempcnn.layer_def{l}.type, 'subsample layer')
            tempcnn = Back_subsample_convolution(tempcnn, l, alpha, sc_connection{l+1});

        end
    end
end
```

```
%% Load Data
load('mnist_uint8');

% training data
labels = unique(train_y, 'rows');
index = randperm(size(train_x,1));
train_x = train_x(index,:);
train_y = train_y(index,:);


training = cell(1);
trainingLabels = cell(1);

for i = 1:size(labels,1)
    counter = 0;
    for n = 1:size(train_y)
        if counter < 10
            if(sum(labels(i,:) == train_y(n,:)) == 10)
                training{i,1}(n,:) = train_x(n,:);
                trainingLabels{i,1}(n,:) = train_y(n,:);
                counter = counter +1;
            end
        end
    end
    training{i,1} = training{i,1}(sum(training{i,1},2) ~= 0,:);
    trainingLabels{i,1} = trainingLabels{i,1}(sum(trainingLabels{i,1},2) ~= 0,:);
end

training = double(cell2mat(training));
trainingLabels = double(cell2mat(trainingLabels));



%testing data
labels = unique(test_y, 'rows');
index = randperm(size(test_x,1));
test_x = test_x(index,:);
test_y = test_y(index,:);


testing = cell(1);
testingLabels = cell(1);

for i = 1:size(labels,1)
    counter = 0;
    for n = 1:size(test_y)
        if counter < 10
            if(sum(labels(i,:) == test_y(n,:)) == 10)
                testing{i,1}(n,:) = test_x(n,:);
                testingLabels{i,1}(n,:) = test_y(n,:);
                counter = counter +1;
            end
        end
    end
    testing{i,1} = testing{i,1}(sum(testing{i,1},2) ~= 0,:);
    testingLabels{i,1} = testingLabels{i,1}(sum(testingLabels{i,1},2) ~= 0,:);
end

testing = double(cell2mat(testing));
testingLabels = double(cell2mat(testingLabels));
```

```
%% distortion 1 - noise
noiseTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imnoise(reshape(testing(i,:),28,28), 'salt & pepper',0.2);
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    noiseTest(i,:) = X;
end
%% distortion 1 - scale
scaleTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imresize(reshape(testing(i,:),28,28), 1+rand*0.1);
    X = imcrop(X,[round(size(X)/2)-28/2 27 27]);
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    scaleTest(i,:) = X;
end
%% distortion 1 - rotation
rotationTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imrotate(reshape(testing(i,:),28,28),(rand*2-1)*90);
    X = imcrop(X,[round(size(X)/2)-28/2 27 27]);
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    rotationTest(i,:) = X;
end
%% distortion 1 - location
locationTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imtranslate(reshape(testing(i,:),28,28),[4*(rand*2-1), 4*(rand*2-
1)],'FillValues',0);
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    locationTest(i,:) = X;
end
%% distortion 1 - light exposure
exposureTest = zeros(100,784);
for i = 1:size(testing,1)
    X = imadjust(reshape(testing(i,:),28,28));
    X = (X - min(min(X))) / max(max(X)) - min(min(X)) * 2 - 1;
    X = reshape(X, 1, 28*28);
    exposureTest(i,:) = X;
end
%%  rbf-0.1, linear-0.1, polynomial-0.1
TestData = noiseTest;
%%  rbf-0.1, linear-0.1, polynomial-0.1
TestData = rotationTest;
%%  rbf-0.12; linear-0.16; polynomial-0.16
TestData = scaleTest;
%%   linear -0.1, rbf - 0.1; polynomial-0.1
TestData = exposureTest;
%%   linear-0.1 rbf-0.1 polynomial-0.1
TestData = locationTest;
%%
% transform train_y
Y_tr = zeros(size(train_y,1),1);
for i=1:size(train_y,1)
    Y_tr(i) = find(train_y(i,:)==1)-1;
end

% X_whole
X_whole = [train_x; TestData];
idxtr = 1:size(train_x,1);
```

```
idxte = (size(train_x,1)+1):size(X_whole,1);

% pca
[X_whole_pca, score, latent] = pca(double(X_whole)');
i = 1;
while((sum(latent(1:i))/sum(latent)) < 0.8)
    PCA_dim = i;
    i = i+1;
end
X_tr = X_whole_pca(idxtr,1:PCA_dim);
X_te = X_whole_pca(idxte,1:PCA_dim);

true_label = zeros(100,1);
for i=1:100
    true_label(i) = find(trainingLabels(i,:)==1)-1;
end
Y_te = true_label;
%%
rd = randperm(60000);
ridx = rd(1:1000);
%%
[pred_score_matrix, pred_matrix, FinalPredMatrix]=svm_predict_composite(X_tr(ridx,:),
Y_tr(ridx), X_te, 1, 'rbf', 1, 1, 9);
length(find(FinalPredMatrix==Y_te))/length(Y_te)
%%

function [funMargin, predict] = svm_predict(X_train, Y_train, X_test, cost, ChosenKernel,
sigma, offset, degree)
% X_train is N*M matrix
% Y_train is N*1 matrix
% X_test is N2*M2 matrix
% cost is the constraint
% ChosenKernel is the kernel choosed (linear, rbf, polynomial)
% sigma is kernel rbf's parameter
% offset and degree is polynomial's parameter

% Output funMargin is f(x) = W'X + b
% Output predict is 1 if >0 otherwise is -1

[alphas, spt_idx, intercept] = svm_train(X_train, Y_train, cost, ChosenKernel, sigma,
offset, degree);
N = size(X_test, 1);
funMargin = zeros(N,1);
predict = zeros(N,1);
for i=1:N
    funMargin(i) = decision_Fun(X_train, X_test(i,:), alphas, Y_train, ChosenKernel,
sigma, offset, degree) + intercept;
end
predict(find(funMargin>0))=1;
predict(find(funMargin<0))=-1;
end



function [pred_score_matrix, pred_matrix, FinalPredMatrix]=svm_predict_composite(X_train,
Y_train, X_test, cost, ChosenKernel, sigma, offset, degree)
unique_class = unique(Y_train);
class_num = length(unique_class);
pred_score_matrix = zeros(size(X_test,1), class_num);
pred_matrix = zeros(size(X_test,1), class_num);
FinalPredMatrix = zeros(size(X_test,1), 1);
```

```
for i = 1:class_num
    disp(i)
    Class_groups = ismember(Y_train, unique_class(i))*2 - 1;
    % test using test instances
    [funMargin, predict] = svm_predict(X_train, Class_groups, X_test, cost, ChosenKernel,
sigma, offset, degree);
    % make the 1's score be positive and 0's score be negative
    pred_score_matrix(:, i) = funMargin;
    pred_matrix(:, i) = predict;
end

for i = 1:size(X_test,1)
    predMaxIdx = find(pred_score_matrix(i,:)== max(pred_score_matrix(i,:)))-1;
    FinalPredMatrix(i) = predMaxIdx ;
end

end


function [alphas, spt_idx, intercept] = svm_train(X_train, Y_train, cost, ChosenKernel,
sigma, offset, degree)
% X_train is N*M matrix
% Y_train is N*1 matrix
% cost is the constraint
% ChosenKernel is the kernel choosed (linear, rbf, polynomial)
% sigma is kernel rbf's parameter
% offset and degree is polynomial's parameter

% alphas is the dual problem solution
% spt_idx is the index of support vectors
% intercept is the w0

% calculate the observation number
N = size(X_train, 1);

% initial kernel matrix
if strcmp(ChosenKernel, 'rbf');
    Kmatrix = zeros(N, N);
    for i = 1:N
        for j = 1:N
            Kmatrix(i,j) = kernelfun(X_train(i,:), X_train(j,:), ChosenKernel, sigma,
offset, degree);
        end
    end
elseif strcmp(ChosenKernel, 'polynomial');
    Kmatrix = (X_train*X_train' + offset).^degree;
else
    Kmatrix = X_train*X_train';
end
% quadratic programing to solve dual problem of SVM
options = optimset('LargeScale','on','MaxIter',2000);
eps = 1e-4;
Y = Y_train*Y_train';
H = Kmatrix.*Y + eps.*eye(N);
f = -1.*ones(N, 1);
Aeq = Y_train';
beq = 0;
LB = zeros(N, 1);
UB = cost.*ones(N, 1);
alphas = quadprog(H, f, [], [], Aeq, beq, LB, UB,[],options);
spt_idx = find(alphas > 1e-4);
```

```
% compute the intercept w0 (Andrew Ng svm mentioned that)
idx_neg = find(Y_train==-1);
idx_poz = find(Y_train==1);
neg_d = zeros(size(idx_neg, 1), 1);
poz_d = zeros(size(idx_poz, 1), 1);
for i = 1:size(idx_neg, 1)
    neg_d = decision_Fun(X_train, X_train(idx_neg(i),:), alphas, Y_train, ChosenKernel,
sigma, offset, degree);
end
for i = 1:size(idx_poz, 1)
    poz_d = decision_Fun(X_train, X_train(idx_poz(i),:), alphas, Y_train, ChosenKernel,
sigma, offset, degree);
end
intercept = -(max(neg_d) + min(poz_d))/2;

end
```

```
function K = kernelfun(x, y, ChosenKernel, sigma, offset, degree)
% x and y is N*1 matrix
% ChosenKernel is the kernel choosed (linear, rbf, polynomial)
% sigma is kernel rbf's parameter
% offset and degree is polynomial's parameter
% output K is to return a value

if strcmp(ChosenKernel, 'polynomial');
    K = (sum(x.*y) + offset)^degree;
elseif strcmp(ChosenKernel, 'rbf');
    K = exp(-sum((x-y).^2)/(2*sigma^2));
else strcmp(ChosenKernel, 'linear');
    K = sum(x.*y);
end

end
```