Deep learning

Unsupervised Feature Learning and Deep Learning [1]

Ling Kangjie[2]

September 26, 2017

[1] http://ufldl.stanford.edu/tutorial/

[2] lingkangjie@Foxmail.com

Dedicated to somebody.

Contents

List of Figures

List of Tables

Preface

This tutorial will teach you the main ideas of Unsupervised Feature Learning and Deep Learning. By working through it, you will also get to implement several feature learning/deep learning algorithms, get to see them work for yourself, and learn how to apply/adapt these ideas to new problems. This tutorial assumes a basic knowledge of machine learning (specifically, familiarity with the ideas of supervised learning, logistic regression, gradient descent). If you are not familiar with these ideas, we suggest you go to this Machine Learning course[1] and complete sections II, III, IV (up to Logistic Regression) first.

Supervised Learning and Optimization

- Linear Regression
- Logistic Regression
- Debugging: Gradient Checking
- Softmax Regression
- Debugging methods

Convolutional Neural Network

- Feature Extraction Using Convolution
- Pooling And Downsample
- Networks Architecture
- Classical CNNs' Models

Unsupervised learning

- Autoencoders
- PCA whitening
- Sparse Coding

[1] `https://goo.gl/hSZRFq`

- Restricted Boltzmann Machine
- Deep Belief Networks

Reinforcement learning

In this section, it would cover the following problems:

- Introduction And Basic Equations
- Monte Carlo sampling
- Deep Q networks

Acknowledgements

Thanks my father, my mother and other person supported me in the past.

Lingkangjie
https://space.bilibili.com/8661791/#!/

1

Introduction

"What I cannot create, I do not understand."

– Richard Philip Feynman, *1918-1988*

1.1 Linear Regression

1.1.1 Problem Formulation and Function Minimization

As a refresher, we will start by learning how to implement linear regression.We will introduce two main concepts: the procedure of machine learning and function optimization.
Our goal in linear regression is to predict a target value $\boldsymbol{y}$ starting from a vector of input values $\boldsymbol{x} \in \mathbb{R}^n$ For example, we might want to make predictions about the price of a house so that $\boldsymbol{y}$ represents the price of the house in dollars and the elements $\boldsymbol{x}_j$ of $\boldsymbol{x}$ represent "features" that describe the house (such as its size and the number of bedrooms). Suppose that we are given many examples of houses where the features for the i'th house are denoted $\boldsymbol{x}^{(i)}$ and the price is $\boldsymbol{x}^{(i)}$. For short, we will denote our goal is to find a function $\boldsymbol{y} = \boldsymbol{h}(\boldsymbol{x})$ so that we have $\boldsymbol{y}^{(i)} \approx \boldsymbol{h}(\boldsymbol{x}^{(i)})$ for each training example. Here, $\boldsymbol{h}(\boldsymbol{x}^{(i)})$ is our hypothesis.
For linear regression, we have $\boldsymbol{h_\theta}(\boldsymbol{x}) = \sum_j \boldsymbol{\theta}_j \boldsymbol{x}_j = \boldsymbol{\theta}^T \boldsymbol{x}$. Here, $\boldsymbol{h_\theta}(\boldsymbol{x})$ represents a large family of functions parametrized by the choice of $\boldsymbol{\theta}$ parameter vector.(We call this space of functions a `hypothesis class`). In particular, we want to search for a choice of $\boldsymbol{\theta}$ that minimizes the cost function:

$$\boldsymbol{J}(\boldsymbol{\theta}) = \frac{1}{2}\sum_i (\boldsymbol{h}(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)})^2 = \frac{1}{2}\sum_i (\boldsymbol{\theta}^T \boldsymbol{x}^{(i)} - \boldsymbol{y}^{(i)})^2$$

We use gradient descent method to optimize the cost function. The gradient of $bmJ(\boldsymbol{\theta})$ is $\nabla_\theta \boldsymbol{J}(\boldsymbol{\theta})$, which is a vector:

$$\nabla_\theta \boldsymbol{J}(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\partial \boldsymbol{J}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_1} \\ \frac{\partial \boldsymbol{J}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_2} \\ \frac{\partial \boldsymbol{J}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_3} \\ \vdots \\ \frac{\partial \boldsymbol{J}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_n} \end{bmatrix}$$

Differentiating the cost function $\boldsymbol{J}(\boldsymbol{\theta})$ as given above with respect to a particular parameter θ_j gives us:

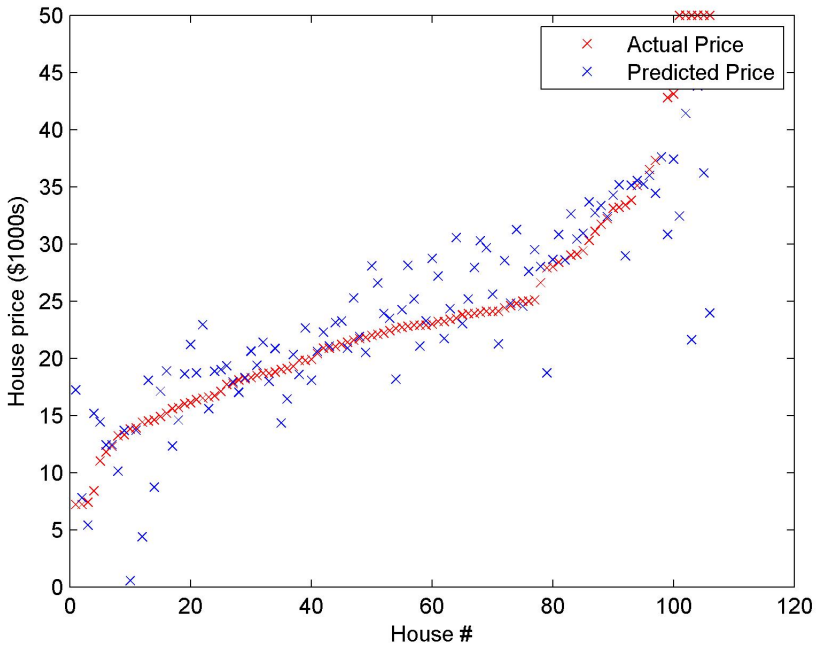
$$\frac{\partial \boldsymbol{J}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j} = \sum_i \boldsymbol{x}_j^{(i)}(\boldsymbol{h}(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)})$$

1.1.2 MATLAB Implementation

The file *ex1a_linreg.m* is to split and shuffle data, call *linear_regression.m* function and optimize it through minFunc toolbox, and plot the results, just as follow:

```
% the "housing.data" coming from http://archive.ics.uci.edu/ml.

clc;
clear;
close all;
addpath ../common
addpath ../common/minFunc_2012/minFunc
addpath ../common/minFunc_2012/minFunc/compiled

% Load housing data from file.
data = load('housing.data');
data=data'; % put examples in columns

% Include a row of 1s as an additional intercept feature.
data = [ ones(1,size(data,2)); data ];

% Shuffle examples.
data = data(:, randperm(size(data,2)));

% Split into train and test sets
% The last row of 'data' is the median home price.
train.X = data(1:end-1,1:400);
train.y = data(end,1:400);

test.X = data(1:end-1,401:end);
test.y = data(end,401:end);

[features_n,samples_n] = size(train.X);

% Initialize the coefficient vector theta to random values.
theta = rand(features_n,1);

% Run the minFunc optimizer with linear_regression.m as the objective.
tic;
options = struct('MaxIter', 200);
theta = minFunc(@linear_regression, theta, options, train.X, train.y);
fprintf('Optimization took %f seconds.\n', toc);

% Plot predicted prices and actual prices from training set.
```

```
40 actual_prices = train.y;
41 predicted_prices = theta'*train.X;
42
43 % Print out root-mean-squared (RMS) training error.
44 train_rms=sqrt(mean((predicted_prices - actual_prices).^2));
45 fprintf('RMS training error: %f\n', train_rms);
46
47 % Print out test RMS error
48 actual_prices = test.y;
49 predicted_prices = theta'*test.X;
50 test_rms=sqrt(mean((predicted_prices - actual_prices).^2));
51 fprintf('RMS testing error: %f\n', test_rms);
52
53 % Plot predictions on test data.
54 plot_prices=true;
55 if (plot_prices)
56   [actual_prices,I] = sort(actual_prices);
57   predicted_prices=predicted_prices(I);
58   plot(actual_prices, 'rx');
59   hold on;
60   plot(predicted_prices,'bx');
61   legend('Actual Price', 'Predicted Price');
62   xlabel('House #');
63   ylabel('House price (1000 dollars)');
64 end
```



The *linear_regression.m* file is:

```
1 function [f,g] = linear_regression(theta, X,y)
2   %
3   % Arguments:
```

```
4    %   theta - A vector containing the parameter values to optimize.
5    %   X - The examples stored in a matrix.
6    %       X(i,j) is the i'th coordinate of the j'th example.
7    %   y - The target value for each example.  y(j) is the target for exam
8    %
9
10   m=size(X,2);
11   n=size(X,1);
12
13   f=0;
14   g=zeros(size(theta));
15   % f: cost function sum , g:gradient sum
16  for row = 1:m
17       curr_x = X(:,row);
18       curr_y = y(:,row);
19       f = f + (1/2).*(theta'*curr_x-curr_y)^2;
20       g = g + curr_x.*(theta'*curr_x-curr_y);
21  end
22 % another version:vectorization
23 %   h = theta' * X - y;
24 %   f = (1/2) * h * h';
25 %   g = X*h';
```

1.2 Logistic Regression

In this section,We will use logistic regression to solve a classification problem. In linear regression, we tried to predict the value of $\boldsymbol{y}^{(i)}$ for the i'th example $\boldsymbol{x}^{(i)}$ using a linear function $\boldsymbol{y} = \boldsymbol{h_\theta}(\boldsymbol{x}) = \boldsymbol{\theta}^T\boldsymbol{x}$. This is a continuous-valued problem. For 2-class problems,we use logistic regression which outputs denote the probability for each class as the form:

$$\boldsymbol{P}(\boldsymbol{y}=1|\boldsymbol{x}) = \boldsymbol{h_\theta}(\boldsymbol{x}) = \frac{1}{1+\mathbf{exp}(-\boldsymbol{\theta}^T\boldsymbol{x})} = \sigma(\boldsymbol{\theta}^T\boldsymbol{x})$$

$$\boldsymbol{P}(\boldsymbol{y}=0|\boldsymbol{x}) = 1 - \boldsymbol{h_\theta}(\boldsymbol{x})$$

We use cross entropy to design the cost function:

$$\boldsymbol{J}(\boldsymbol{\theta}) = -\sum_i (\boldsymbol{y}^{(i)} log(\boldsymbol{h_\theta}(\boldsymbol{x}^{(i)})) + (1-\boldsymbol{y}^{(i)}) log(1-\boldsymbol{h_\theta}(\boldsymbol{x}^{(i)})))$$

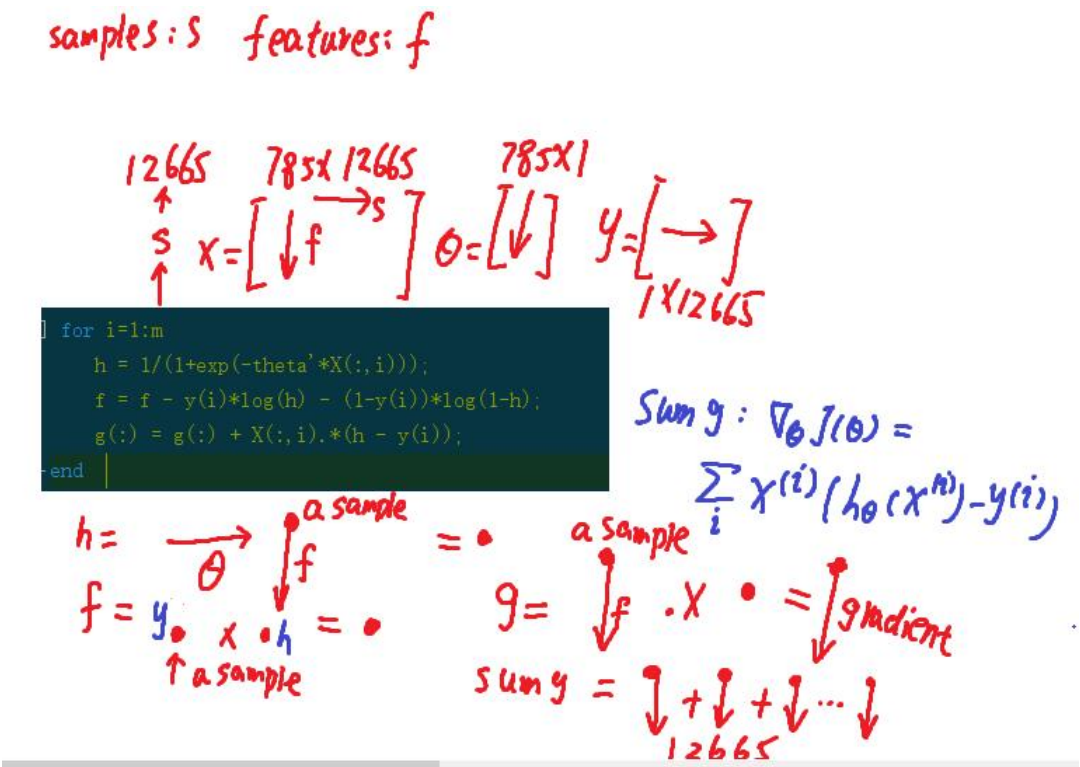
The gradient of $\boldsymbol{J}(\boldsymbol{\theta})$ and written in its vector form is:

$$\nabla_{\boldsymbol{\theta}}\boldsymbol{J}(\boldsymbol{\theta}) = \sum_i \boldsymbol{x}^{(i)}(\boldsymbol{h}(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)})$$

This is essentially the same as the gradient for linear regression except that the hypothesis function.

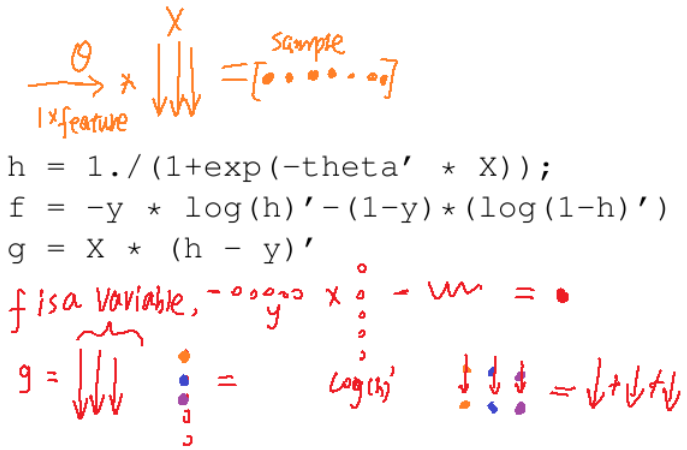
1.2.1 Matlab Implementation

There will involve 5 files:
1. *ex1b_logreg.m*: the main file, to optimize the cost function, train and test the model.
2. *loadMNISTImages.m ex1_load_minst.m*: *ex1_load_minst.m* call *loadMNISTImages.m* function to load MNIST data, return train and test data set. Data form description: train data, a structure, containing 2 constants, X with 784*12665 and y with 1*12665, in which 12665 is train sample numbers. Test data set have the same form with 2115 train samples.
3. *logistic_regression.m*: computing the cost function and its gradient. initializing the $\boldsymbol{\theta}$ is a 785*1 random vector, after transpose, multiplied with X to get results passed to *sigmoid* function. Finally we get the computed feedforward function in which each element denote a probability correspondingly for a sample belongs to a certain class. Compared to the true results, we want to compute the cost function in order to get the gradient.
NOTE: the shape of gradient matrix $\boldsymbol{\theta}$(in codes is $\boldsymbol{g}$) is just the same as $\boldsymbol{x}$.f is just a variable.



Vectorized version is:

```
1       h = 1./(1+exp(-theta' * X));
2       f = -y * log(h)'-(1-y)*(log(1-h)')
3       g = X * (h - y)'
```



4. *binary_classifiler_accuracy.m*: to calculate the model classification accuracy. Multiplied $\boldsymbol{X}$ input by the optimized $\boldsymbol{\theta}$ to get the output. If output is bigger than 0.5, that is to say classification correct.

1.3 Debugging: Gradient Checking

Sometimes, we need to check the gradient of objective function manually to insure that our optimization algorithm works correctly. Basically, according to derivative definition, for a extremely small ϵ, we have:

$$\boldsymbol{g}(\boldsymbol{\theta}) \approx \frac{\boldsymbol{J}(\boldsymbol{\theta}+\epsilon) - \boldsymbol{J}(\boldsymbol{\theta}-\epsilon)}{2 \times \epsilon}$$

Because $\boldsymbol{\theta}$ is a vector, if we want to check the i-th element of $\boldsymbol{g}$ vector, just:

$$\boldsymbol{g}_i(\boldsymbol{\theta}) \approx \frac{\boldsymbol{J}(\boldsymbol{\theta}^{i+}) - \boldsymbol{J}(\boldsymbol{\theta}^{i-})}{2 \times \epsilon}$$

Where

$$\boldsymbol{\theta}^{i\pm} = \boldsymbol{\theta} \pm \epsilon \times \vec{\boldsymbol{e}}_i$$

in which, $\vec{e}_i$ is the i-th basis vector of the same dimension as $\boldsymbol{\theta}$, with a `1` in the i-th position and `0`s everywhere else.

1.3.1 Matlab Implementation

We create a file called *grad_check.m*. In anywhere, We just pass the optimized function, initialized θ, number checks, and train data set to the *grad_check.m* function. The number checks mean that how many times we want to check the gradient of random selected dimension of θ. Using form likes:

```
error = grad_check(@logistic_regression_vec, ...
    theta, 10, train.X, train.y)
```

```
function average_error = grad_check(fun, theta0, num_checks, varargin)
  delta=1e-3;
```

```
  sum_error=0;
  fprintf(' Iter       i              err');
  fprintf('           g               g_estimated                f\n')
  for i=1:num_checks
    T = theta0;
    j = randsample(numel(T),1);
    T0=T; T0(j) = T0(j)-delta;
    T1=T; T1(j) = T1(j)+delta;

    [f,g] = fun(T, varargin{:});
    f0 = fun(T0, varargin{:});
    f1 = fun(T1, varargin{:});
    g_estimated = (f1-f0) / (2*delta);
    error = abs(g(j) - g_estimated);
    fprintf('% 5d  % 6d % 15g % 15f % 15f % 15f\n', ...
            i,j,error,g(j),g_estimated,f);
    sum_error = sum_error + error;
  end
  average_error = sum_error/num_checks;
```

1.4 Softmax Regression

We use logistic regression classifier to distinguish between two kinds of hand-written digits. For multiple classes, we use softmax regression classifier. For $\boldsymbol{y}^{(i)} \in \Re^K$ (K classes), normalized the hypothesis distribution:

$$\boldsymbol{h_\theta(x)} = \frac{1}{\sum_{j=1}^{K} \mathbf{exp}(\boldsymbol{\theta}^{(j)T}\boldsymbol{x})} \begin{bmatrix} \mathbf{exp}(\boldsymbol{\theta}^{(1)T}\boldsymbol{x}) \\ \mathbf{exp}(\boldsymbol{\theta}^{(2)T}\boldsymbol{x}) \\ \vdots \\ \mathbf{exp}(\boldsymbol{\theta}^{(K)T}\boldsymbol{x}) \end{bmatrix}$$

Note that the $\boldsymbol{\theta}$ is a n-by-K matrix, n denoted the input feature dimensions of $\boldsymbol{x}$. Where $\boldsymbol{\theta} = [\boldsymbol{\theta^{(1)}\theta^{(2)}...\theta^{(K)}}]$

1.4.1 Cost Function and Its properties

From my point of view, softmax regression is a two layers neural networks, and the $\boldsymbol{\theta}$ can be seen as the weigh of the networks. In the equation below, $\mathbf{1\{\bullet\}}$ is the indicator function $\mathbf{1\{a\ true\ statement\} = 1}$, and $\mathbf{1\{a\ false\ statement\} = 0}$. That is to say when $\bullet$ is true, the $\mathbf{1\{\bullet\}}$ outputs 1. Our cost function will be:

$$\boldsymbol{J(\theta)} = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{\boldsymbol{y}^{(i)=k}\} log \frac{\mathbf{exp}(\boldsymbol{\theta}^{(k)T}\boldsymbol{x}^{(i)})}{\sum_{j=1}^{K}\mathbf{exp}(\boldsymbol{\theta}^{(k)T}\boldsymbol{x}^{(i)})}\right]$$

When K = 2, the cost function of softmax regression becomes:

$$\boldsymbol{J(\theta)} = -\sum_{i}(\boldsymbol{y}^{(i)} log(\boldsymbol{h_\theta(x}^{(i)})) + (1-\boldsymbol{y}^{(i)}) log(1-\boldsymbol{h_\theta(x}^{(i)})))$$

Which is the logistic regression cost function. The gradient of softmax regression cost function is a Hessian matrix:

$$\nabla_{\boldsymbol{\theta}^{(k)}} \boldsymbol{J}(\boldsymbol{\theta}) = -\sum_{i=1}^{m} \left[\boldsymbol{x}^{(i)} \left(1\{\boldsymbol{y}^{(i)} = k\} - \frac{\mathbf{exp}(\boldsymbol{\theta}^{(k)T} \boldsymbol{x}^{(i)})}{\sum_{j=1}^{K} \mathbf{exp}(\boldsymbol{\theta}^{(j)T} \boldsymbol{x}^{(i)})} \right) \right]$$

in which, m is sample size, $\nabla_{\boldsymbol{\theta}^{(k)}} \boldsymbol{J}(\boldsymbol{\theta})$ is itself a vector. Softmax regression has an unusual property that is it has a `redundant` set of parameters. In other words, suppose we take each of our parameter vectors $\boldsymbol{\theta}^{(j)}$, and subtract some fixed vector $\boldsymbol{\psi}$ from it, so that every $\boldsymbol{\theta}^{(j)}$ is now replaced with $\boldsymbol{\theta}^{(j)} - \boldsymbol{\psi}$ (for every $j = 1, 2, ..., k$), which means $\boldsymbol{\theta}^{(j)}$ has redundant information. Let's see how it happens. For our hypothesis function $\boldsymbol{h_\theta(x)}$, each element in it has the form:

$$\frac{\mathbf{exp}(\boldsymbol{\theta}^{(k)T} \boldsymbol{x}^{(i)})}{\sum_{j=1}^{K} \mathbf{exp}(\boldsymbol{\theta}^{(j)T} \boldsymbol{x}^{(i)})}$$

If we subtract a $\boldsymbol{\psi}$ from $\boldsymbol{\theta}^{(k)}$ in numerator and denominator at the same time, it, saying $exp((\boldsymbol{\theta}^{(k)-\psi})^\mathsf{T})$. Because **exp** function properties, the calculated hypothesis result is just the same as not been subtracted. In a extreme situation, suppose $\boldsymbol{\theta}^{(K)} = \boldsymbol{\psi}$, one can reduce $K \times n$ parameters to $(K-1) \times n$ parameters needed to be optimized without harming the representational power of our hypothesis. When $K = 2$, the softmax regression becomes logistic regression.

$$\boldsymbol{h_\theta(x)} = \frac{1}{\mathbf{exp}(\boldsymbol{\theta}^{(1)\mathsf{T}} \boldsymbol{x}) + \mathbf{exp}(\boldsymbol{\theta}^{(2)\mathsf{T}} \boldsymbol{x})} \begin{bmatrix} \mathbf{exp}(\boldsymbol{\theta}^{(1)\mathsf{T}} \boldsymbol{x}) \\ \mathbf{exp}(\boldsymbol{\theta}^{(2)\mathsf{T}} \boldsymbol{x}) \end{bmatrix}$$

Tasking advantage of the properties that softmax regression hypothesis is overparameterized and setting $\boldsymbol{\psi = \theta^{(2)}}$, we can subtract $\boldsymbol{\theta^{(2)}}$ from each of the two parameters, giving us

$$\begin{aligned} \boldsymbol{h_\theta(x)} &= \frac{1}{\mathbf{exp}((\boldsymbol{\theta}^{(1)} - \boldsymbol{\theta}^{(2)})^\mathsf{T} \boldsymbol{x}) + 1} \begin{bmatrix} \mathbf{exp}((\boldsymbol{\theta}^{(1)} - \boldsymbol{\theta}^{(2)})^\mathsf{T} \boldsymbol{x}) \\ 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 - \frac{1}{\mathbf{exp}((\boldsymbol{\theta}^{(1)} - \boldsymbol{\theta}^{(2)})^\mathsf{T} \boldsymbol{x}) + 1} \\ \frac{1}{\mathbf{exp}((\boldsymbol{\theta}^{(1)} - \boldsymbol{\theta}^{(2)})^\mathsf{T} \boldsymbol{x}) + 1} \end{bmatrix} \\ &= \begin{bmatrix} 1 - \frac{1}{\mathbf{exp}(-\boldsymbol{\theta}'^\mathsf{T} \boldsymbol{x}) + 1} \\ \frac{1}{\mathbf{exp}(-\boldsymbol{\theta}'^\mathsf{T} \boldsymbol{x}) + 1} \end{bmatrix} \quad \text{(if we denote } \boldsymbol{\theta}^{(2)} - \boldsymbol{\theta}^{(1)} = -\boldsymbol{\theta}'\text{)} \end{aligned}$$

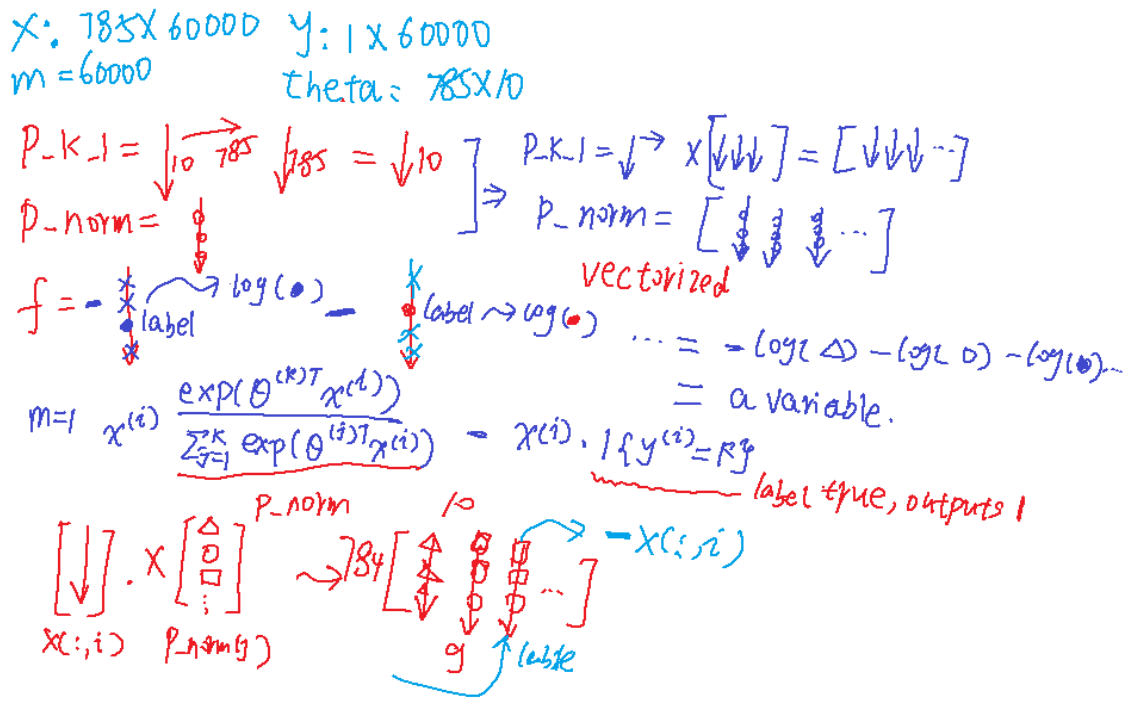
From the above derivation, we find that softmax regression predicts the probability of one of the classes as $\frac{1}{\mathbf{exp}(-\boldsymbol{\theta}'^\mathsf{T} \boldsymbol{x}) + 1}$, and that of the other class as $1 - \frac{1}{\mathbf{exp}(-\boldsymbol{\theta}'^\mathsf{T} \boldsymbol{x}) + 1}$, same as logistic regression.

1.4.2 Matlab Implementation

It will involve three files:
1. *ex1c_softmax.m*: the main file.
2. *multi_classifier_accuracy.m*: to estimate the model. 3. *softmax_regression_vec.m*: to compute the cost fucntion and its gradients. The unvectorized core codes are:

```
for i=1:m
      label = y(i);
```

```
3         p_k_1 = exp(theta'*X(:,i));
4         p_norm = p_k_1 ./ sum(p_k_1);
5         f = f - log(p_norm(label));
6       for j=1:num_classes
7           g(:,j) = g(:,j) + X(:,i).*(p_norm(j));
8       end
9         g(:,label) = g(:,label) - X(:,i);
10   end
11  f = f./m;
12  g(:,num_classes) = [];
13  g=g./m;
14  % make gradient a vector for minFunc
15  g=g(:);
```



Note: we would handle all 10 digit classes instead 2 classes as in the previous section. According to softmax regression properties, the last class corresponding to its gradient could be set 0, meaning the 10th class is left out of $\boldsymbol{\theta}$.

1.5 Neural Network and Backpropagation Algorithm

In this section, we will learn a model called Neural Network(NN), and see how does such model can help us to compute graph-based model. We will from another point of view to see softmax regression. First, I would introduce basic mathematical notations, and then briefly describe the network architecture.

1. $\boldsymbol{W}_{i,j}^{(l)}$: the parameter(or weigh) associated with the connection between unit j in layer l, and unit i in layer $l+1$. The network connection direction is from j to i.

2. $\boldsymbol{b}_i^{(l)}$: the bias associated with unit i in layer $l+1$.

3. $\boldsymbol{a}_i^l$: activation of unit i in layer l. Especially, $\boldsymbol{a}_i^{(1)} = \boldsymbol{x}_i$.

4. $\boldsymbol{W, b}$: combination of all the $\boldsymbol{W}_{i,j}^{(l)}$ and $\boldsymbol{b}_i^{(l)}$, are total parameters of network we basically want to learn.
5. m: training example size.
6. λ: weigh decay parameter.
7. $\boldsymbol{\delta}_{\boldsymbol{i}}^{(\boldsymbol{l})}$: error term, the core of our backpropagation algorithm, which measures how much that node(i-th node in layer l) responsible for any errors in our output.

The cost function(one-half squared-error cost function, with $\boldsymbol{L2}$ regularization) for the network to be:

$$\begin{aligned}\boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}) &= \left[\frac{1}{m}\sum_{i=1}^{m}\boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_{l-1}}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\boldsymbol{W}_{ji}^{(l)}\right)^2 \\ &= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\parallel \boldsymbol{h}_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x}^{(i)}) - \boldsymbol{y}^{(i)} \parallel^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_{l-1}}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}\left(\boldsymbol{W}_{ji}^{(l)}\right)^2\end{aligned}$$

As $\boldsymbol{\delta}_{\boldsymbol{i}}^{(\boldsymbol{l})}$ is a very important term in our backpropagation algorithm, we now formally describe how to compute it:

1. Perform a feedforward pass, computing The activations for layers $\boldsymbol{L_2}$, $\boldsymbol{L_3}$, and so on up to output layer $\boldsymbol{L_{n_l}}$.
2. For each output unit i in layer n_l (the output layer), set

   $$\boldsymbol{\delta}_i^{(n_l)} = \frac{\partial}{\partial \boldsymbol{z}_i^{(n_l)}}\frac{1}{2}\parallel \boldsymbol{y} - \boldsymbol{h}_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x}) \parallel^2 = -(\boldsymbol{y_i} - \boldsymbol{a}_i^{(n_l)}) \boldsymbol{\cdot} \boldsymbol{f'}(\boldsymbol{z}_i^{(n_l)})$$

   in which, $\boldsymbol{z}^{l+1} = \boldsymbol{W}^{(l)}a^{(l)} + b^{(l)}$, $\boldsymbol{a}^{(l+1)} = \boldsymbol{f}(\boldsymbol{z}^{(l+1)})$.
3. For $l = n_l - 1, n_l - 2, n_l - 3, ..., 2$ For each node i in layer l, set

   $$\boldsymbol{\delta}_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}}\boldsymbol{W}_{ji}^{(l)}\boldsymbol{\delta}_j^{(l+1)}\right)\boldsymbol{f'}(\boldsymbol{z}_i^{(l)})$$

4. Compute the desired partial derivatives, which are given as:

   $$\frac{\partial}{\partial \boldsymbol{W}_{ij}^{(l)}}\boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{a}_j^{(l)}\delta_i^{(l+1)}$$

   $$\frac{\partial}{\partial \boldsymbol{b}_i^{(l)}}\boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \delta_i^{(l+1)}$$

Finally, we can re-write the algorithm using matrix-vectorial notation. We will use `$\bullet$` to denote the element-wise product operator. We also do The same for $\boldsymbol{f'(\bullet)}$(so that $\boldsymbol{f'([z_1, z_2]) = [f'(z_1), f'(z_2)]}$). For step2, after vetorizing, we get (just eliminate i notation)

$$\boldsymbol{\delta}^{(n_l)} = -(\boldsymbol{y} - \boldsymbol{a}^{(n_l)}) \boldsymbol{\bullet} \boldsymbol{f'}(\boldsymbol{z}^{(n_l)})$$

For step3, the equation can be re-write as:

$$\boldsymbol{\delta}^{(l)} = \left((\boldsymbol{W}^{(l)})^T\boldsymbol{\delta}^{(l+1)}\right) \boldsymbol{\bullet} \boldsymbol{f'}(\boldsymbol{z}^{(l)})$$

For step4,

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)}$$

If we use sigmoid activation, $f'(z_i^{(l)}) = a_i^{(l)}(1 - a_i^{(l)})$.

1.5.1 BackPropogation Implementation Using Matlab

Based on the previous section, we will train a neural network classifier to classify the 10 digits in the MNIST dataset. First, let's define a neural network We will work for it later. We have a network with a layer of 784 input units, a layer of 10 output units and a single hidden layer of 256 hidden units.

- *run_train.m*
  The main script, to load MNIST data set, setup random initial weighs, make cost function optimized and compute accuracy on the test and train set.

- *initialize_weights.m*
  Initialling the weights between each layer, the results are stored in `stack` variable. As the network has 3 layers, neurons of each layer are 784-256-1. The `stack` variable has such a form: stack<1×2 cell>, stack(1, 1) has variables of W<256×784 double> and b<256×1 double>, stack(1, 2) has variables of W<10×256 double> and b<10×1 double>.

- *params2stack.m stack2params.m* Converts a flattened parameter vector into a nice `stack` structure or flattens a stack to a vector for us to work with. For example, if we use `stack = params2stack(params, ei)`, the `params` is our object we want to convert, and the `ei` is an auxiliary variable(maybe a cell structure in matlab) which tells us the configuration of the network we need to satisfy after converted. The total size of `params` here is 203430, according to `ai` structure, we have $203430 = 256 \times 784 + 256 + 10 \times 256 + 10$. Because `minFunc` only can deal with parameters vector, we need such mechanism to convert back and forth.

- *supervised_dnn_cost.m*
  Our core function, to slave cost and gradient computation. It contains three main components, forward prop, cost computation and gradients computed using back-propagation.

```
1  function [ cost, grad, pred_prob] = supervised_dnn_cost( theta, ei,..
2      data, labels, pred_only)
3  %SPNETCOSTSLAVE Slave cost function for simple phone net
4  %   Does all the work of cost / gradient computation
5  %   Returns cost broken into cross-entropy, weight norm, and prox reg
6  %        components (ceCost, wCost, pCost)
7
8  %% default values
9  po = false;
10 if exist('pred_only','var')
```

```
  po = pred_only;
end;

%% reshape into network
stack = params2stack(theta, ei);
numHidden = numel(ei.layer_sizes) - 1;
hAct = cell(numHidden+1, 1);%hAct is <2*1 cell>
gradStack = cell(numHidden+1, 1);%gradStack is <2*1 cell>
%% forward prop
%%% YOUR CODE HERE %%%
m = size(data,2);%m=60000
m_inv = 1/m;
for l = 1:(numHidden+1)
    if l==1
        hAct{l} = data; % just store input data<784*60000>
    else
% when run to here, l=2.
% stack{l-1}.W <256*784>,hAct{l-1}=hAct{1} <784*60000>
% after multiplying,<256*60000>meaning demension from
% 784 reduce to 256. stack{l-1}.b=stack{1} <256*1>
% repmat(stack{l-1}.b,[1,m]) is <256*60000>,meaning row direction
% repeats 1 times and column direction repeats 60000 times.
% because for a sample we have 256 dimensions in hidden layer,so
% we have 256 neurons,meaning has 256 bias for a sample.
% here hAct{l} is "L",l=2, not number 1.
        hAct{l} = stack{l-1}.W * hAct{l-1} + ...
            repmat(stack{l-1}.b,[1,m]);
        if strcmp(ei.activation_fun,'logistic')
            hAct{l} = sigmoid(hAct{l});
        elseif strcmp(ei.activation_fun,'tanh')
            hAct{l} = tanh(hAct{l});
        elseif strcmp(ei.activation_fun,'rlu')
            hAct{l}(hAct{l}<0) = 0;
        end
    end
end
% layer l=2,stack{l}.W <10,256>,hAct{l}< 256*60000>,
% stack{l}.b <10,1>
% repmat(stack{l}.b,[1,m])<10,60000>. From another perspective,
% the output layer mapping 256 demensions to 10 demensions.
% pred_prob <10*60000>
pred_prob = stack{l}.W * hAct{l} + repmat(stack{l}.b,[1,m]);
% perform element-by-element binary operation for two array.
% max(pred_prob,[],1) <1*60000>, here, "max" function
% produces the maximum values along the first
% dimension of "pred_prob".
% normalizes the output to range[0,1] to represent probability.
% bsxfun works like it,suppose
```

```
% A = [1 4 6  B=[2 2 2],after bsxfun(@minus,A,B) = [-1 2 4
%     5 7 3]                                         3 5 2]
% after sigmoid function, each neuron output
% a variable range[0,1],however
% for Softmax regression, we treat these outputs
% as weighed inputs,and put these outputs into
% exp power function and divided by its sum
% respectively for each sample
```

The cost The cost function is nearly identical to the softmax regression cost function. Note that instead of making predictions from the input data x the softmax function takes as input the final hidden layer of the network

$$\boldsymbol{h}_{W,b}(\boldsymbol{x}) \tag{1.1}$$

. The loss function is thus:

$$\boldsymbol{J}(\boldsymbol{\theta}) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{\boldsymbol{y}^{(i)=k}\} log \frac{\exp(\boldsymbol{\theta}^{(k)T}\boldsymbol{h}_{W,b}(\boldsymbol{x}^{(i)}))}{\sum_{j=1}^{K}\exp(\boldsymbol{\theta}^{(k)T}\boldsymbol{h}_{W,b}(\boldsymbol{x}^{(i)}))}\right]$$

```
pred_prob = bsxfun(@minus,pred_prob,max(pred_prob,[],1));
pred_prob = exp(pred_prob);
pred_prob = bsxfun(@rdivide, pred_prob, sum(pred_prob));

%% return here if only predictions desired.
if po
  cost = -1; ceCost = -1; wCost = -1; numCorrect = -1;
  grad = [];
  return;
end;

%% compute cost
%%% YOUR CODE HERE %%%
% groundTruth <10*60000>,meaning if groundTruth(j,i) not equal to 0,
% that j is the hand number label for input sample i
groundTruth = full(sparse(labels, 1:m, 1));
% groundTruth <10*60000>,pred_prob <10*60000>,after sum, <1*60000>
cost = -mean(sum(groundTruth .* log(pred_prob)));
% to add L2 regularization term.
ceCost = -(groundTruth - pred_prob);
wCost = 0;
% for all weighs in the network, just sum all weighs^2 up.Here wCost
% called weigh cost
for l=1:(numHidden+1)
    wCost = wCost + sum(sum(stack{l}.W .^ 2));
end
wCost = 0.5 * ei.lambda * wCost;
cost = cost + wCost;

%% compute gradients using backpropagation
```

```
%%% YOUR CODE HERE %%%
delta = cell(numHidden+1,1);%delta<2*1 cell>
for l=numHidden+1:-1:1 %from back to front, numHidden = 1
    % layer l is 2,2==1+1,output layer backpro
    if(l==numHidden+1)
        delta{l} = ceCost;
        %the residual between groundTruth and computed results
        % delta{2}<10*60000>,hAct{2}<256*60000>,actually is hidden
        % layer (layer 2) output (activation output).
        % I AM CONFUSED AT HERE.
        gradStack{l}.W = delta{l} * hAct{l}' .* m_inv;
        gradStack{l}.b = mean(delta{l},2);
    else
        % stack{2}.W'<60000*10>,delta{2}<10*60000>
        delta{l} = stack{l+1}.W' * delta{l+1};
        if strcmp(ei.activation_fun,'logistic')
            delta{l} = delta{l} .* hAct{l+1} .* (1-hAct{l+1});
        elseif strcmp(ei.activation_fun,'tanh')
            delta{l} = delta{l} .* (1-hAct{l+1}.^2);
        elseif strcmp(ei.activation_fun,'rlu')
            delta{l}(hAct{l+1}<0) = 0;
        end
        % I AM doubt here,why not consider the W2 matrix?
        % l = 1
        gradStack{l}.W = delta{l} * hAct{l}' .* m_inv + ...
            ei.lambda .* stack{l}.W;
        gradStack{l}.b = mean(delta{l},2);
    end
end

%% compute weight penalty cost and gradient for non-bias terms
%%% YOUR CODE HERE %%%

%% reshape gradients into vector
[grad] = stack2params(gradStack);
end

function sigm = sigmoid(x)
    sigm = 1./(1+exp(-x));
end
```

1.6 Convolutional Neural Networks and pooing

If you have already been familiar with convolutional neural network, here I will emphasize some important details of it only. Basically, suppose input volume is a *width*height*depth* image, which width size is always the same with height, meaning that is a square shape.

Filter is *width_filter*height_filter*depth_filter*, which always is the case that $width\_filter = height\_filter, depth = depth\_filter$. If we have K filters, then we will get K activation maps. For a activation map, suppose the padding size is for one border, it has output size $((width - width\_filter + 2 * paddingsize)/stride + 1) \times ((height - height\_filter + 2 * paddingsize)/stride + 1)$. In common settings, K is always the powers of 2, e.g. 32, 64, 128. For an element in activation, we get it by element-wise product of volume *width*height*depth* and *width_filter*height_filter*depth_filter*. For a next time, we move the filter volume a stride , do the element-wise product, and get another element of the activation map. If we have a 1×1 filter with depth of D, the corresponding activation map size will not change compared to input volume but its depth to D.
Pooling, also known as downshampling, captures a feature-invariance property. Say we have a activation map $m \times m$ needed to be pool. If we implement $n \times n$ pooling, Then, our output has $(m \div n) \times (m \div n)$ regions. For a region, we take $n \times n$ mean (or maximum) feature pooling over $m \times m$ activation map to obtain the pooled convolved features.

1.6.1 Implement convolution and Pooling

There are somes implementation tips. If we use conv2(image, W) MATLAB function, as MATLAB will first `flip` W before convolving W with image, we need to rotate it twice rot90 before put it into conv2 function, e.g.:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \underset{flip}{\overset{rotate-twice}{\Longleftrightarrow}} \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

- *cnnExercise.m* The main file to check our implementation whether it is right or not. To load MNISI data set, initial weigh and bias, call the *cnnConvolve.m* and *cnnPool.m* function.

- *cnnConvolve.m*
  $filterDim = 8$, meaning 8×8 filter. $numFilters = 100$, meaning has 100 filters. So W has shape of $< 8 \times 8 \times 100 >$. The b has shape of $< 100 \times 100 >$, but we only need $(numFilters, 1)$. The *images* shape $< 28 \times 28 \times 8 >$, meaning we have 8 image samples of size $< 28 \times 28 >$, each image has one depth because it is a gray image not RGB. The *convolvedFeatures* variable is a 4-D tensor, contains the activation maps, our convoluted results. For a activation map, it has dimension $28 - 8 + 1$, as our stride equals to one, and not padding. The *convolvedFeatures* variable should have $numFilters \times numImages$ activation maps.
  We apply 100 filters to a input sample (there total are 8 samples to run) shape $< 28 \times 28 >$ in turn and add bias(100 biases) corresponding to 100 filters, in other words, convolve every filter with every image to get the total 100*8 activation maps. After a filters and an image convoluted, we copy the intermediate variable *convolvedImage* into *convolvedFeatures*

```
convolvedImage = conv2(im,filter,'valid');
convolvedImage = sigmoid(convolvedImage + b(filterNum,1));
convolvedFeatures(:, :, filterNum, imageNum) = convolvedImage;
```

- *cnnPool.m*
  When we get the convolutional output, a 4-D tensor $convolvedFeatures$ variable, we can perform the pooling. The pooling output is also a 4-D tensor with its shape $< 21/3 \times 21/3 \times 100 \times 8 >$ (*convolvedDim / poolDim, convolvedDim / poolDim, numFilters, numImage*). There are some tricks to perform pooling: as we want a computing operations compatibility environment, we use *conv*2 operations again. We first convolute activation map with $< poolDim \times poolDim >$ matrix patch of 1s, and then downsample it with stride of $poolDim$, and each element in the result $< 7 \times 7 >$ patch will divided by $poolDim * poolDim$ to get final mean pooling.

```
for imageNum = 1:numImages
    for filterNum = 1:numFilters
    pooled = conv2(convolvedFeatures(:,:,filterNum,imageNum), ...
    ones(poolDim,poolDim), 'valid');
    pooledFeatures(:,:,filterNum,imageNum) =
    pooled(1:poolDim:end,1:poolDim:end) ./ (poolDim*poolDim);
    end
end
```

1.7 Build up Your Own Convolutional Neural Network

Implement the CNN cost and gradient computation in this step. Your network will have two layers. The first layer is a convolutional layer followed by mean pooling and the second layer is a densely connected layer into softmax regression. The cost of the network will be the standard cross entropy between the predicted probability distribution over 10 digit classes for each image and the ground truth distribution.

cnnTrain.m

Our main train file is to call other functions, train and test the CNN.

- *cnnInitParams.m*
  Before we perform forward propagation, we need to initial parameters for a single layer convolutional neural. In this function, the parameters space are: $filterDim = 9$, $numFilters = 20$, $imageDim = 28$, $poolDim = 2$, $numClasses = 10$. So after initialling, filter volume has shape of $< 9 \times 9 \times 20 >$ (the filter volume is W(matlab variable denoted by Wc) between input layer and hidden layer). After pooling, the acitvation map dimension $outDim$ are *(imageDim-filterDim+1)/poolDim*=10, so hidden neurons are $outDim^2 \times numFilters = 2000$. The output layer size are 10, so W(matlab variable denoted by Wd) between hidden layer and output layer has size $< 10 \times 2000 >$. For a filter, we touch it a bias, so we totally get $< 20 \times 1 >$ biases *bc*. For output layer, each neuron contact a bias, so there are $< 10 \times 1 >$ biases *bd*. At last, flatten and concatenated all these initialled parameters together, *cnnInitParams.m* returns a θ containing $< 21650 \times 1 >$(9∗9∗20+10∗2000+20+10) parameters.

- *cnnCost.m*
  Our core function, to perform forward progation, calculate cost, and compute back

propagation gradients, and make predictions if necessary. Our minibatch size is 256, so a *images* variable contains 256 pictures. The file *cnnParamsToStack.m* is to transform θ into four variables Wc, Wd, bc, bd. In other words, it unrolls θ. Our goal is to updata these four parameters gradients.

Table 1.1: The variable space

Variable Name	Value	variable Name	Value
$filterDim$	9	$images$	$< 28 \times 28 \times 256 >$
$labels$	$< 256 \times 1 >$	$numClasses$	10
$numFilters$	20	$poolDim$	2
θ	$< 21650 \times 1 >$	λ	0.0001
Wc	$< 9 \times 9 \times 20 >$	Wd	$< 10 \times 2000 >$
bc	$< 20 \times 1 >$	bd	$< 10 \times 1 >$

STEP 1a: Forward Propagation
In forward progation, we will conpute CNN layer, and Softmax layer.

Listing 1.1: Forward Computing for Convolutional Layer

```
% dimension of convolved output
convDim = imageDim-filterDim+1;
% dimension of subsampled output
outputDim = (convDim)/poolDim;
% is a 4-D tensor with shape <20*20*20*256>,
% for storing convolued results.
activations = zeros(convDim,convDim,numFilters,numImages);
% is a 4-D tensor with shape <10*10*20*256>,
% for storing pooling results.
activationsPooled = zeros(outputDim,outputDim, ...
numFilters,numImages);
activations = cnnConvolve(filterDim, numFilters, ...
images, Wc, bc);
activationsPooled = cnnPool(poolDim, activations);
% Reshape activations into 2-d matrix for Softmax layer,
% hiddenSize=2000, activationsPooled <2000*256>.
activationsPooled = reshape(activationsPooled,[],numImages);
```

Listing 1.2: Forward Computing for Softmax Layer

```
% for storing probability that each image belongs to each class.
   probs <10*256>
probs = zeros(numClasses,numImages);
% Wd<10*2000>,activationsPooled<2000*256>, hidden neurons 2000,
   number images 256
probs = Wd * activationsPooled + repmat(bd,[1,numImages]);
% normalizing ,probs <10*256>
probs = bsxfun(@minus, probs, max(probs, [], 1));
probs = exp(probs);
probs = bsxfun(@rdivide, probs, sum(probs));
```

STEP 1b: Calculate Cost

$$\boldsymbol{J}(\boldsymbol{\theta}) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{\boldsymbol{y}^{(i)=k}\} log \frac{\exp(\boldsymbol{\theta}^{(k)T}\boldsymbol{h}_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x}^{(i)}))}{\sum_{j=1}^{K}\exp(\boldsymbol{\theta}^{(k)T}\boldsymbol{h}_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x}^{(i)}))}\right]$$

Listing 1.3: Calculate Cost

```
cost = 0; % save results into cost
% groundTruth <10*256>
groundTruth = full(sparse(labels, 1:numImages, 1));
% numImages_inv = 1/number images = 1/256, cost is a variable, just a
    single number, neither vector nor matrix.
% probs <10*256>, groundTruth <10*256>, element wise.For a sample,
   compute 10 by 10 element wise to get one result, all the samples
   results added up.for details, see equation in file
   supervised_dnn_cost.m
cost = -numImages_inv*(groundTruth(:)'*log(probs(:))) ...
 + (lambda/2.)*(sum(Wd(:).^2)+sum(Wc(:).^2));
```

STEP 1c: Backpropagation

Before computing gradients for the 4 variables mentioned above, we need to calculate backpropagate errors θ. we will need to propagate this error through the subsampling and convolutional layer. To do this upsampling quickly, we use kron function to upsample the error and propagate through the pooling layer. If X is 2-by-3, Y is another matrix, then kron(X,Y) is [X(1,1)*Y X(1,2)*Y X(1,3)*Y X(2,1)*Y X(2,2)*Y X(2,3)*Y] e.g:

$$\begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \\ 3 & 3 & 4 & 4 \\ 3 & 3 & 4 & 4 \end{bmatrix} \leftarrow kron\left(\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}\right)$$

For a normal BP, we have:

$$\boldsymbol{\delta}^{(l)} = \left((\boldsymbol{W}^{(l)})^T\boldsymbol{\delta}^{(l+1)}\right) \bullet \boldsymbol{f}'(\boldsymbol{z}^{(l)})$$

If the l-th layer is a convolutional and subsampling layer then the error is propagated through as

$$\boldsymbol{\delta}^{(l)} = upsample\left((\boldsymbol{W}^{(l)})^T\boldsymbol{\delta}^{(l+1)}\right) \bullet \boldsymbol{f}'(\boldsymbol{z}^{(l)})$$

In our experiment, the l is second layer, hidden layer, $l = 2$. $\boldsymbol{W}^{(l)}$ is between layer 2 and layer 3 (the output layer). $\boldsymbol{\delta}^{(l+1)}$, $\boldsymbol{\delta}^{(3)}$, the output layer propagated error. $\boldsymbol{f}'(\boldsymbol{z}_i^{(l)})$ is the derivative of the activation function, and $\boldsymbol{f}'(\boldsymbol{z}_i^{(l)}) = \boldsymbol{a}_i^{(l)}(\boldsymbol{1} - \boldsymbol{a}_i^{(l)})$. k indexes the filter number.

Listing 1.4: Calculate Delta Error for CNN Layer

```
%delta <10*256>
delta = -(groundTruth - probs);
% 4-D tensor, Wd is between layer 2 and layer 3.
```

```
delta_pool = reshape(Wd'*delta, outputDim, outputDim, ...
numFilters, numImages);
% delta_conv <20*20*20*256>
delta_conv = zeros(convDim,convDim,numFilters,numImages);
% upsampling the delta_pool to delta_conv
for i=1:numImages
    for j=1:numFilters
    % strictly speaking, here delta_conv variable is denoted
    % the result of upsampleed pooling. After kron, get<20*20>
    delta_conv(:,:,j,i) = (1./poolDim^2) .* kron(...
    squeeze(delta_pool(:,:,j,i)), ones(poolDim));
    end
end
% To propagate pooling error through the convolutional layer,
% we simply need to multiply the incoming error by
% the derivative of the activation function as in the
% usual back propagation algorithm.
% activations, delta_conv: 4-D, element wise product
% activations valiable comes from CNN forward path, and have
% not been put into Softmax layer
delta_conv = activations .* (1-activations) .* delta_conv;
```

STEP 1d: Gradient Calculation There are 4 parameters gradient we need to calculate. Compute the gradient for the densely connected weights and bias, W_d and b_d, just as usual. In usual situation,

$$\nabla_{\boldsymbol{W}^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \delta^{(l+1)} (\boldsymbol{a}^{(l)})^T$$
$$\nabla_{\boldsymbol{b}^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \delta^{(l+1)}$$

To calculate the gradient w.r.t.(with respect to) to the filter maps, we rely on the border handling convolution operation again and flip the error matrix $\delta_k^{(l)}$ the same way we flip the filters in the convolutional layer. In layer 1 and layer 2 is to do convolutional and pooling oprations, $\boldsymbol{a}^1$ is image input layer data. k denoted k-th filter

$$\nabla_{\boldsymbol{W}_k^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \sum_{i-1}^{m} (\boldsymbol{a}_i^{(l)}) * rot90(\boldsymbol{\delta}_k^{(l+1)}, 2)$$
$$\nabla_{\boldsymbol{b}_k^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \sum_{a,b} (\boldsymbol{\delta}_k^{(l+1)})_{a,b}$$

Attention: For a sample, we calculate its gradients as this way

$$\frac{\partial}{\partial \boldsymbol{W}_{ij}^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{a}_j^{(l)} \delta_i^{(l+1)}$$
$$\frac{\partial}{\partial \boldsymbol{b}_i^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \delta_i^{(l+1)}$$

For a batch of sample, say we have m samples, the derivative of the overall cost function is:

$$\frac{\partial}{\partial \boldsymbol{W}_{ij}^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \boldsymbol{W}_{ij}^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)}) \right] + \lambda \boldsymbol{W}_{ij}^{(l)}$$

$$\frac{\partial}{\partial \boldsymbol{b}_{i}^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \boldsymbol{b}_{ij}^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$$

Listing 1.5: Gradient Calculation

```
% according the above two equations, for a batch of samples.
% delta <10*256> activationsPooled <2000*256>,
% Wd <10*2000> sum(delta, 2)<10,1>
Wd_grad = numImages_inv .* delta * ...
activationsPooled' + lambda .* Wd;
bd_grad = numImages_inv .* sum(delta, 2);
for i=1:numFilters
    for j=1:numImages
    % For a filter, we need to backpropagate error for
    % that filter with each image and aggregate over images.
    % Wc_grad contains 20 filters, meaning i=1:20.
    % squeeze(images(:,:,j)) <28*28>
    % squeeze(delta_conv(:,:,i,j)) <20*20>
        Wc_grad(:,:,i) = Wc_grad(:,:,i) + conv2(...
        squeeze(images(:,:,j)),...
        rot90(squeeze(delta_conv(:,:,i,j)),2), ...
        'valid');
    end
    % Wc_grad <9*9*20>, Wc <9*9*20>
    Wc_grad(:,:,i) = numImages_inv .* ...
    Wc_grad(:,:,i) + lambda .* Wc(:,:,i);

    % temp 4-D tensor, sum(temp(:)) is a variable
    % meaning, for a filter, we inspect all its
    % images into which the filter has ever filtered.
    % then we get this bais gradient represents all
    % all images the filter has ever seen.
    temp = delta_conv(:,:,i,:);
    bc_grad(i) = numImages_inv .* sum(temp(:));
end
% Unroll gradient into grad vector for minFuncd
grad = [Wc_grad(:) ; Wd_grad(:) ; bc_grad(:) ; bd_grad(:)];
```

- *computeNumericalGradient.m*
  Before we preform a large scale computation, it is a good habit to check gradient to insure our cost function calculated correctly. Once our code passes the gradient check we are ready to move onto training a real network on the full dataset. Make sure to switch the DEBUG boolean to false in order not to run the gradient check again.
  Suppose we have already defined all the computing envirionment for gradient checking such as the number of images, labels, filter dimension et al (for more details to

see the *cnnTrain.m* file). Then, we just change a tiny value of a certain dimension of θ parameters(a vector contains all the network weighs and biases) to see what happen in cost function J we have defined already. Compared the gradient J computed with our derivative definition computed, we can see how well our cost function works.

Listing 1.6: gradient checking

```
function numgrad = computeNumericalGradient(J, theta)
% Initialize numgrad with zeros
numgrad = zeros(size(theta));
epsilon = 1e-4;
for i =1:length(numgrad)
    oldT = theta(i);
    theta(i)=oldT+epsilon;
    pos = J(theta);
    theta(i)=oldT-epsilon;
    neg = J(theta);
    numgrad(i) = (pos-neg)/(2*epsilon);
    theta(i)=oldT;
    if mod(i,100)==0
        fprintf('Done with %d\n',i);
    end;
end;
end
```

- *computeNumericalGradient.m*
  Batch methods, such as limited memory BFGS, which use the full training set to compute the next update to parameters at each iteration tend to converge very well to local optima. They are also straight forward to get working provided a good off the shelf implementation (e.g. minFunc) because they have very few hyper-parameters to tune. However, often in practice computing the cost and gradient for the entire training set can be very slow and sometimes intractable on a single machine if the dataset is too big to fit in main memory. Another issue with batch optimization methods is that they don't give an easy way to incorporate new data in an 'online' setting. Stochastic Gradient Descent (SGD) addresses both of these issues by following the negative gradient of the objective after seeing only a single or a few training examples. The use of SGD In the neural network setting is motivated by the high cost of running back propagation over the full training set. SGD can overcome this cost and still lead to fast convergence. We will use SGD with momentum to accelerate computing and avoid local minimums to some extent.

Table 1.2: The variable space

Variable Name	Value	variable Name	Value
$funObj$	cost function handle	$data$	$< 28 \times 28 \times 60000 >$
$labels$	$< 60000 \times 1 >$	$peochs$	3
$alpha$	0.1000	$momentum$	0.9500
θ	$< 21650 \times 1 >$	$minibatch$	256

We obey the following equations to update our parameter θ

$$\boldsymbol{v} = \gamma \boldsymbol{v} + \alpha \nabla_{\boldsymbol{\theta}} \boldsymbol{J}(\boldsymbol{\theta}; \boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})$$

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \boldsymbol{v}$$

In above equation $\boldsymbol{v}$ is the current velocity vector which is of the same dimension as the parameter vector $\boldsymbol{\theta}$. Finally $\lambda \in (0, 1]$ determines for how many iterations the previous gradients are incorporated into the current update.

Listing 1.7: SGD with Momentum

```
% Setup for momentum
mom = 0.5;
momIncrease = 20;
velocity = zeros(size(theta));
it = 0;
for e = 1:epochs % 1:3
% randomly permute indices of data for quick minibatch sampling
    rp = randperm(m);
        for s=1:minibatch:(m-minibatch+1) % s = 1:256:(60000-256), total
           234 iterations
        it = it + 1;
        % increase momentum after momIncrease iterations
        % for a epoch, suppose after 19 iterations, when in 20-th
        % iterations, we updata mom to 0.9500, and for rest of itetations,
        % we use the 0.9500 momentum all the time
        if it == momIncrease % momIncrease = 20
            mom = options.momentum;
        end;
    % get next randomly selected minibatch
    mb_data = data(:,:,rp(s:s+minibatch-1)); %mb_data <28*28*256>
    mb_labels = labels(rp(s:s+minibatch-1)); % mb_labels <256*1>
    % evaluate the objective function on the next minibatch
    % cost: a variable, grad:<21650*1>
    [cost grad] = funObj(theta,mb_data,mb_labels);
    % Update velocity and theta
    % velocity, theta: <21650*1>
    velocity = mom.* velocity + alpha .* grad;
    theta = theta - velocity;
    fprintf('Epoch %d: Cost on iteration %d is %f\n',e,it,cost);
    end;
    % aneal learning rate by factor of two after each epoch
    % we are total 3 epochs, for a epoch, we observe all data set once
    alpha = alpha/2.0;
end;
opttheta = theta;
```

The last work is to test our model. As we access the optimized $\theta = opttheta$ from the file *computeNumericalGradient.m* output. We just put it to *cnnCost.m* file to compute forward output.

Listing 1.8: Test

```
[~,cost,preds]=cnnCost(opttheta,testImages,testLabels,numClasses,...
filterDim,numFilters,poolDim,true);
% testLabels <10000*1>,compare two data sets element by
% element to check our model accuracy.
acc = sum(preds==testLabels)/length(preds);
% Accuracy should be around 97.4% after 3 epochs
fprintf('Accuracy is %f\n',acc);
```

Listing 1.9: cnnCost.m

```
if pred
    % probs <10*10000>,10000 test samples.preds<1*10000>
    % suppose preds(1,1)=7,denoted the first test sample output 7.
    [~,preds] = max(probs,[],1);
    preds = preds';
    grad = 0;
    return;
end;
```

Congratulations, you've successfully implemented a Convolutional Neural Network!

2

Unsupervised Learning

2.1 Autoencoders

In last section of chapter 1, we pour great efforts to build up a CNN from scratch. In this section, we will introduce the first kind of unsupervised learning methods, autoencoders. To begin with, we will introduce some Sparse Autoencoder (SAE) notations. The SAE is consist of 3 layers: input layer, hidden layer and output layer. The number of neurons in input layer is the same as output layer. If the number of neurons in hidden layer are less than input layer, then meaning we constraint our network to learn some compression representations of inputs. It's very useful when the input features are correlated, because this algorithm will be able to discover some of those correlations. When there does not exit structure in the data, say inputs coming from i.i.d.(independent and identically distributed) Gaussian independent of the other features, it would very difficult for Autoencoder to learn such a compression task. If the number of neurons in hidden layer are more than input neurons, and if we impose a `sparsity` constraint on the hidden units, then the autoencoder will still discover interesting structure in the data, even if the number of hidden units is large.

- $\boldsymbol{x}$ Input features for a training example, $\boldsymbol{x} \in \Re^n$
- $\boldsymbol{y}$ Output or target values, a vector, $\boldsymbol{y} = \boldsymbol{x}$
- $(\boldsymbol{x^{(i)}, y^{(i)}})$ The $i-$th training example
- $\boldsymbol{h_{W,b}(x)}$ Output of our hypothesis on input $\boldsymbol{x}$, using parameters $\boldsymbol{W, b}$. It is a vector of the same dimension as the target value $\boldsymbol{y}$
- $\boldsymbol{W_{ij}^{(l)}}$ The parameter associated with connection between unit j in layer l, and unit i in layer $l+1$
- $\boldsymbol{b_i^{(l)}}$ The bias term associated with unit i in layer $l+1$. Can also be thought of as the parameter associated with the connection between the bias unit in layer l and unit i in layer $l+1$
- $\boldsymbol{\theta}$ Our parameter vector, can be seen as the results of taking the parameters $\boldsymbol{W, b}$ and unrolling them into a long column vector
- $\boldsymbol{a_i^{(i)}}$ Activation(output) of unit i in layer l of the networks. $\boldsymbol{a_i^{(1)} = x_i}$

- $\boldsymbol{f}(\cdot)$ The activation function. Here we use $\boldsymbol{f}(\boldsymbol{z}) = tanh(\boldsymbol{z})$
- $\boldsymbol{z}_i^{(i)}$ Total weighted sum of inputs to unit i in layer l, $\boldsymbol{a}_i^{(l)} = \boldsymbol{f}(\boldsymbol{z}_i^{(l)})$
- α Learning rate parameter
- s_l Number of units in layer l (not counting the bias unit).
- n_l Number layers in the network, just say how many layers in the network. Layer L_1 is usually the input layer, and layer L_{n_l} is the output layer
- λ Weight decay parameter
- $\hat{\boldsymbol{x}}$ the reconstruction outputs, same meaning as $\boldsymbol{h}_{\boldsymbol{W},\boldsymbol{b}}(\boldsymbol{x})$
- ρ Sparsity parameter, which specifies our desired level of sparsity
- $\hat{\rho}_i$ The average activation of hidden unit i (in the sparse autoencoder)
- β Weight of the sparsity penalty term (in the sparse autoencoder objective)

2.1.1 Sparse Autoencoder Implementation

STEP 1: Generate training set *sampleIMAGES.m*
Variable $IMAGES$ contains 10 images with the shape of 512×512, each image have been preprocessed using whitening menthod. Whitening is a preprocessing step which removes redundancy in the input, by causing adjacent pixels to become less correlated. We will randomly select an image, and sample randomly 8×8 image patch 1000 times for that image, so that we have 10000 patches for each patch has shape of 8×8, and reshaped it into a 64×10000 matrix. BUT in codes, we have a trick to implement it. The function *sampleIMAGES.m* will return 10000 normalized image patches with shape of 64×10000.

Table 2.1: The variable space

Variable Name	Value	variable Name	Value
$IMAGES$	<512*512*10>	$bumpatches$	10000
$patches$	$< 64 \times 10000 >$	$patchsize$	8
idx	from 0 to 10000		

Listing 2.1: backbone of sampleIMAGES

```
idx = 0;
img_rnd = randperm(10);
patch_x_rnd = randperm(512-7);
patch_y_rnd = randperm(512-7);
% consider the IMAGES shape particularity, we divide the sample process
    into
% two same parts but with different random characteristic.
for i=1:10
    for j=1:500
    idx = idx+1;
    patches(:,idx) = reshape(IMAGES(patch_x_rnd(j):(patch_x_rnd(j)+7), ...
```

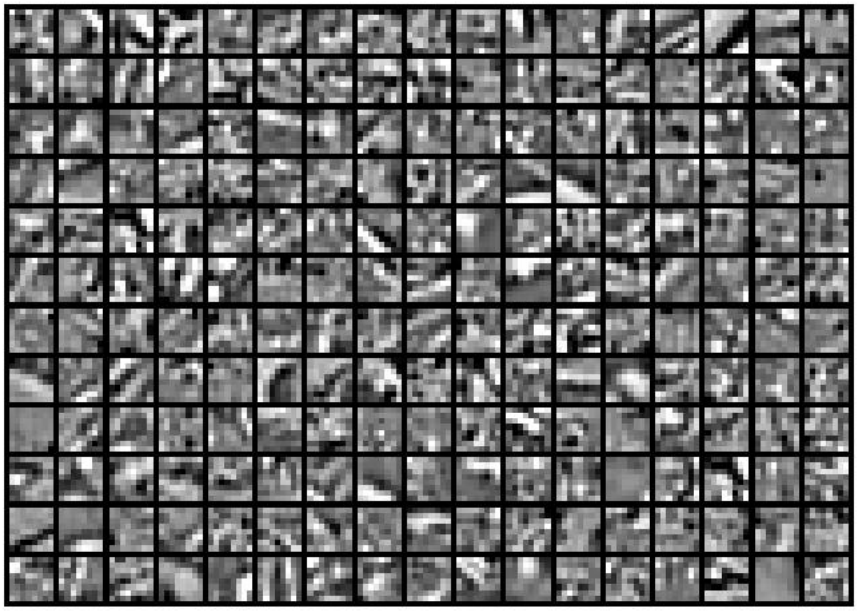
```
    patch_y_rnd(j):(patch_y_rnd(j)+7),img_rnd(i)), 64, 1);
    end
end
% we use the same code again to fill up the other half of 'patches'
   variable.
% note: inx here is from 5001 to 10000
img_rnd = randperm(10);
patch_x_rnd = randperm(512-7);
patch_y_rnd = randperm(512-7);

for i=1:10
    for j=1:500
    idx = idx+1;
     patches(:,idx) = reshape(IMAGES(patch_x_rnd(j):(patch_x_rnd(j)+7),
         ....
     patch_y_rnd(j):(patch_y_rnd(j)+7),img_rnd(i)), 64, 1);
    end
end

patches = normalizeData(patches);
function patches = normalizeData(patches)
% Squash data to [0.1, 0.9] since we use sigmoid as the activation
% function in the output layer

% Remove DC (mean of images).
patches = bsxfun(@minus, patches, mean(patches));

% Truncate to +/-3 standard deviations and scale to -1 to 1
% pstd: a variable
pstd = 3 * std(patches(:));
patches = max(min(patches, pstd), -pstd) / pstd;

% Rescale from [-1,1] to [0.1,0.9]
patches = (patches + 1) * 0.4 + 0.1;

end
```

In our main file *train.m*, we can visualize the image patches result using the file *display_network.m*. A visualization version maybe like following picture.

Listing 2.2: Visualizing Input image patches

```
patches = sampleIMAGES;
% size(patches,2) =10000 randi(10000,200,1) = <200*1>
% randi(imax,m,n) means generate a m-by-n array of integer values from the
    uniform
% distribution on the set 1:imax. display_network(A) function display a A
   matrix (n*m) which
% has m images with the shape of <squareroot of n, squareroot of n> (
   default)
```

```
6 display_network(patches(:,randi(size(patches,2),200,1)),8);
```



Then we use file *initializeParameters.m* to initialize parameters. As the number of input is just the same as the number of output, we only need two parameters to call this function. We put $hiddenSize = 25$ and $visibleSize = 64$, get the initialized parameters θ with shape $< 3289 \times 1 >$(64*25+64+25*62+25, each bias associates a neuron, input layer has bias and output layer has bias).
If you are already familiar with CNN and BP, then you know that our core function or our algorithm soul is to achieve cost function and compute its gradient. Dont't worry, you are unnecessarily afraid for it. In my point of view, it is much more easier than CNN's backpropagation, but still has some impressived equations we need to understand.

- *spareAutoencoderCost.m*

Table 2.2: The variable space

Variable Name	Value	variable Name	Value
$theta$	<3289*1>	$visibleSize$	64
$hiddenSize$	25	$lambda$	1.0000e-04
$sparsityParam$	0.0100	beta	3
$data$	<64*10000>	$W1$	$< 25 \times 64 >$
$W2$	$< 64 \times 25 >$	$b1$	$< 25 \times 1 >$
$b2$	$< 64 \times 1 >$		

We use sigmoid function for activation function, and matrices $W^{(1)} \in \Re^{s_1 \times s_2}$, $W^{(2)} \in \Re^{s_2 \times s_3}$, vectors $b^{(1)} \in \Re^{s_2}$, $b^{(2)} \in \Re^{s_3}$. The objective $J_{sparse}(W, b)$ contains 3 terms, corresponding to squared error term, the weight decay term, and the sparsity penalty.
We write $\boldsymbol{a}_j^{(2)}(\boldsymbol{x})$ to denote the activation of this hidden unit(unit j in layer 2) when the network is given a specific input $\boldsymbol{x}$. Then the average activation of hidden unit j (averaged over the training set) given as follow:

$$\hat{\rho}_j = \frac{1}{m} \sum_{i=1}^{m} \left[\boldsymbol{a}_j^{(x)}(\boldsymbol{x}^{(i)}) \right]$$

If we use KL divergence(KL-divergence is a standard function for measuring how different two different distributions are.) and sum up all $\hat{\rho}_j$ in hidden layer, (in codes, *mean_act1* is $\hat{\rho}_j$ for all j, ρ is *sparsityParam*=0.0100), then

$$\sum_{j=1}^{s_2} \rho log \frac{\rho}{\hat{\rho}_j} + (1-\rho) log \frac{1-\rho}{1-\hat{\rho}_j}$$

and also can be written as

$$\sum_{j=1}^{s_2} \boldsymbol{KL}(\rho || \hat{\rho}_j)$$

For $W2grad$

$$\nabla_{\boldsymbol{W}^{(l)}} \boldsymbol{J}(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, \boldsymbol{y}) = \delta^{(l+1)} (\boldsymbol{a}^{(l)})^T$$

$$\boldsymbol{W}^{(l)} = \boldsymbol{W}^{(l)} - \alpha \left[\left(\frac{1}{m} \Delta \boldsymbol{W}^{(l)} \right) + \lambda \boldsymbol{W}^{(l)} \right]$$

normally, we have

$$\boldsymbol{\delta}^{(l)} = \left((\boldsymbol{W}^{(l)})^T \boldsymbol{\delta}^{(l+1)} \right) \bullet \boldsymbol{f}'(\boldsymbol{z}^{(l)})$$

now instead compute

$$\boldsymbol{\delta}_i^{(2)} = \left(\left(\sum_{j=1}^{s_3} \boldsymbol{W}_{ji}^{(2)} \boldsymbol{\delta}_j^{(3)} \right) + \beta \left(\frac{\rho}{\hat{\rho}_i} + \frac{1-\rho}{1-\hat{\rho}_i} \right) \right) \boldsymbol{f}'(\boldsymbol{z}_i^{(2)})$$

In implementation, we will transmit *delta1* to *delta2*. Here *delta1* is contacted layer 2, these codes may not will be denoted. Because for $\delta^{(l)}$ should from n_l(contacted layer n_l) to 2(contacted layer 2). After all, we just know, we use the privious δ(suppose coming from layer l) and multiply the weigh $\boldsymbol{W}$ between layer l and $l-1$.

Listing 2.3: backbone of spareAutoencoderCost.m

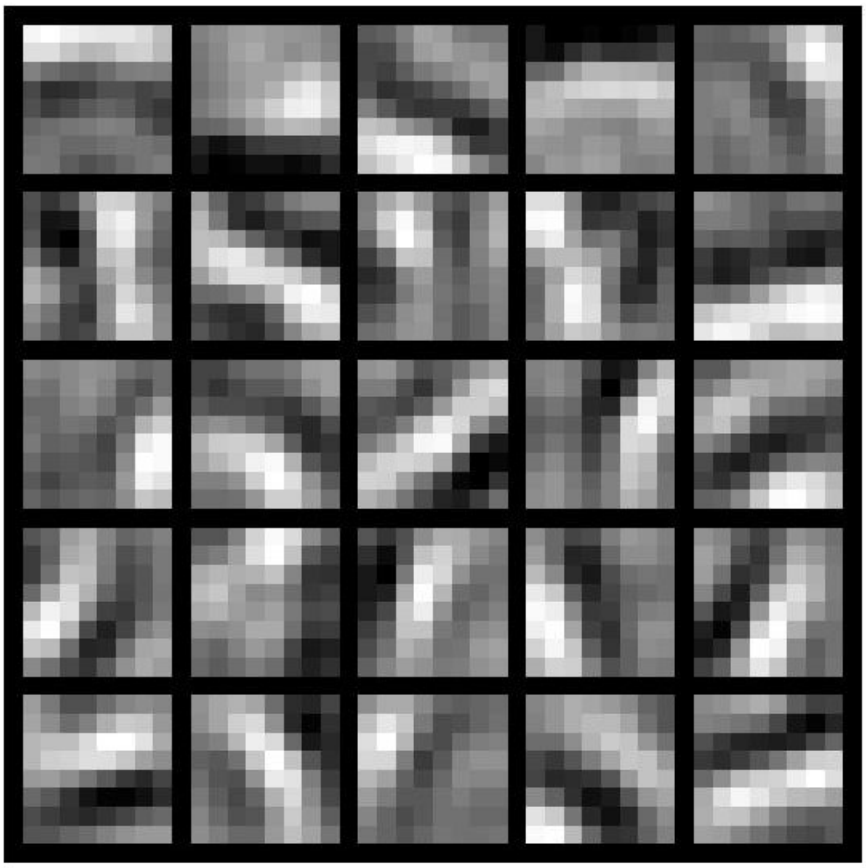
```
% numImages=10000
numImages = size(data,2);
numImages_inv = 1./numImages;
% for each image, we bind it a vector of bias vector b.
% note: we use all samples to compute, not a batch
% activations1 <25*10000> mean_act1 <25*1>
activations1 = sigmoid(W1*data+repmat(b1,[1,numImages]));
output = sigmoid(W2*activations1+repmat(b2,[1,numImages]));
mean_act1 = mean(activations1,2);
% squared_error <64*10000>
squared_error = 0.5.*(output - data).^2;
% squared error term, cost is a variable
cost = numImages_inv .* sum(squared_error(:));
% the weight decay term, cost is a variable
cost = cost + 0.5 .* lambda .* (sum(W1(:).^2) + sum(W2(:).^2));
% the sparsity penalty term, cost is a variable, after kl_div,
% will return <25*1>, and we just need to sum it up
cost = cost + beta .* sum(kl_div(sparsityParam, mean_act1));
% delta2 <64*10000>
delta2 = -(data-output) .* output .* (1-output);
% gradient of bias in layer 2 connected layer 3: b2grad <64*1>
b2grad = mean(delta2,2);
W2grad = numImages_inv .* delta2 * activations1' + lambda .* W2;
% delta_sparsity <25*10000>, we use repmat to generate 10000 samples
   delta sparsity,
% saying that for each sample, we really hope it has such a sparsity.
delta_sparsity = repmat(beta.*(-sparsityParam./mean_act1+(1-
   sparsityParam)./(1-mean_act1)), ...
[1,numImages]);
% delta2 <64*10000>, W2 <64*25>, delta1 <25*10000>
% delta1 has 25 rows, in hidden layer
delta1 = (W2' * delta2 + delta_sparsity) .* activations1 .* (1-
   activations1);
b1grad = mean(delta1,2);
% data in layer 1, is activation
W1grad = numImages_inv .* delta1 * data' + lambda .* W1;
grad = [W1grad(:) ; W2grad(:) ; b1grad(:) ; b2grad(:)];
```

At last, we use *computeNumericalGradient.m* to check our cost function whether its computing process is correct or not. Then we use *minFunc.m* to opitimize our cost function to get the *opttheta* variable. Finally, we need to visualize $W1$. Because θ is consist of 4 parameters, $W1$, $W2$, $b1$, $b2$ successively.

Listing 2.4: Visualization

```
% W1 <25*64>
W1 = reshape(opttheta(1:hiddenSize*visibleSize), hiddenSize,
   visibleSize);
```

```
display_network(W1', 12);
print -djpeg weights.jpg   % save the visualization to a file
```



2.2 PCA, PCA Whitening and ZCA Whitening on Images

For images, the input will be somewhat redundant, because the values of adjacent pixels in an image are highly correlated. We need a way to do dimensionality reduction and make features uncorrelated. The goal of whitening is to make the input less redundant; more formally, our desiderata are that our learning algorithms sees a training input where (i) the features are less correlated with each other, and (ii) the features all have the same variance. In detail, in order for PCA to work well, informally we require that (i) The features have approximately zero mean, and (ii) The different features have similar variances to each other. With natural images, (ii) is already satisfied even without variance normalization, and so we won't perform any variance normalization. In this section, we only need a file *pca_gen.m*

1. For each image $\boldsymbol{x}^{(i)}$, we might normalize its the intensity as follows:

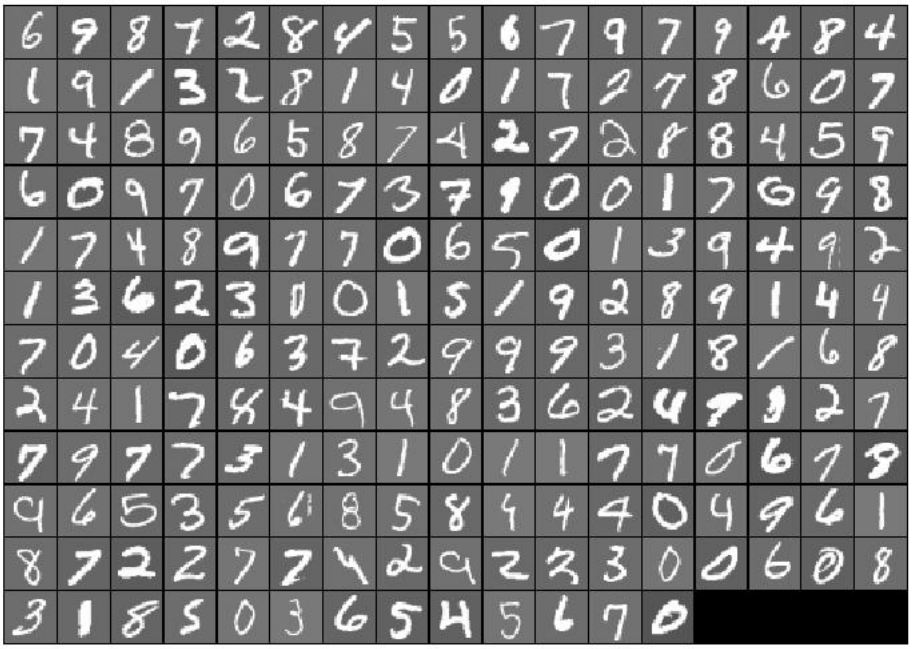
$$\mu^{(i)} := \frac{1}{n}\sum_{j=1}^{n} \boldsymbol{x}_j^{(i)}$$

$$\boldsymbol{x}_j^{(i)} := \boldsymbol{x}_j^{(i)} - \mu^{(i)}$$

for all j.
Here i is to denoted images index, and j is to denoted the i-th image j-th pixel intensity.

Listing 2.5: Zero-mean the data

```
% x is out input images set,<784*60000>, avg <1*60000>, a random
% selection of 200 samples for visualization as follows
avg = mean(x,1);
x = x - repmat(avg,size(x,1),1);
```



2. Next we need to compute the matrix Σ defined as follows:

$$\Sigma = \frac{1}{m}\sum_{i=1}^{m}(\boldsymbol{x}^{(i)})(\boldsymbol{x}^{(i)})^T$$

If $\boldsymbol{x}$ has zero mean, then Σ is exactly the covariance matrix of $\boldsymbol{x}$. In matlab, we can use *sigma = x * x' / size(x,2)*, which x have m samples with shape $< n*m >$. (what is the relationship between the equation and matlab code? I am confused about it.) When we use Matlab *svd* function, then the matrix U will contain the eigenvectors of Sigma (one eigenvector per column, sorted in order from top to bottom eigenvector), and the diagonal entries of the matrix S will contain the corresponding eigenvalues (also sorted in decreasing order). The matrix V will be equal to transpose of U, and can be safely ignored. The svd function actually computes the singular vectors and singular values of a matrix, which for the special case of a symmetric positive semi-definite matrix—which is all that we're concerned with here—is equal to its

eigenvectors and eigenvalues. $\boldsymbol{x}_{rot}$(The subscript `rot` comes from the observation that this corresponds to a rotation (and possibly reflection) of the original data.) is another version of original data $\boldsymbol{x}$. If we get $\boldsymbol{x}_{rot}$, then we can go from the rotated vectors $\boldsymbol{x}_{rot}$ back to the original data $\boldsymbol{x}$ as follows: $\boldsymbol{x} = \boldsymbol{U}\boldsymbol{x}_{rot}$

Listing 2.6: Zero-mean the data

```
xRot = zeros(size(x)); % We need to compute this
% sigma <784*784>
sigma = x * x' / size(x,2);
% u<784*784>, s<784*784>,v<784*784>
[u,s,v] = svd(sigma);
% xRot <784*10000>
xRot = u' * x; % rotated version of the data x
% If we want to a dimensioned version of original data, saying we
% want to keep only k dimensions, use the follows statement,
% where k is the number of eigenvectors to keep
% xTilde = U(:,1:k)' * x;
```

3. How can we insure that our xRot variable is right? Well, covariance matrix for the xRot should be a diagonal matrix with non-zero entries only along the main diagonal. So, we need to compute the covariance matrix of xRot and visualize it.

Listing 2.7: Check your implementation of PCA

```
covar = zeros(size(x, 1)); % You need to compute this
covar = xRot * xRot' / size(xRot,2);
% Visualise the covariance matrix. You should see a line across the
% diagonal against a blue background.
figure('name','Visualisation of covariance matrix');
% you will see a coloured diagonal line against a blue background,
% however, with so many samples, the diagonal line may not be
   apparent.
% you can check and see the concrete covar matrix in Matlab variables
    space
imagesc(covar);
```

4. Now we need to decide how many dimensions should be retain. We will pick k to be as small as possible, but so that at least 99% of the variance is retained.

Listing 2.8: Find k, the number of components to retain

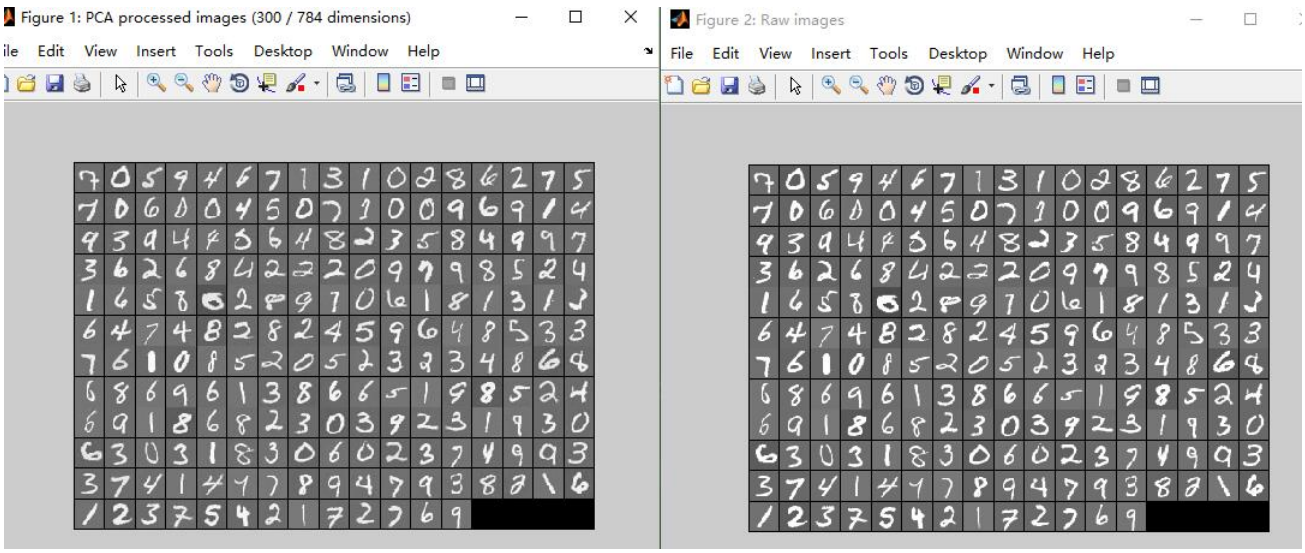
```
k = 0; % Set k accordingly
% s contains our all eigenvalues
total = sum(s(:));
% mat <784*784>
mat = zeros(size(s,1));
% size(s,1) = 784
for i=1:size(s,1)
    mat(i,i) = 1;
    % s is a diagonal matrix, only diagonal has data.
```

```
    % In face, we just sum k elements in s in turn by diagonal and
    % compare the result to total sum to see the percentage
    res = mat * s;
    if (sum(res(:)) / total) >= 0.99
        break;
    end
end
k = i;
```

5. We will recover our data with k dimensions as follows:

Listing 2.9: Implement PCA with dimension reduction

```
xHat = zeros(size(x));  % You need to compute this
u_k = u(:,1:k); % <784*300>
xHat = u_k * u_k' * x; % x <784*60000>
figure('name',['PCA processed images',sprintf('(%d / %d dimensions)',
    k, size(x, 1)),'']);
display_network(xHat(:,randsel));
figure('name','Raw images');
display_network(x(:,randsel))
```

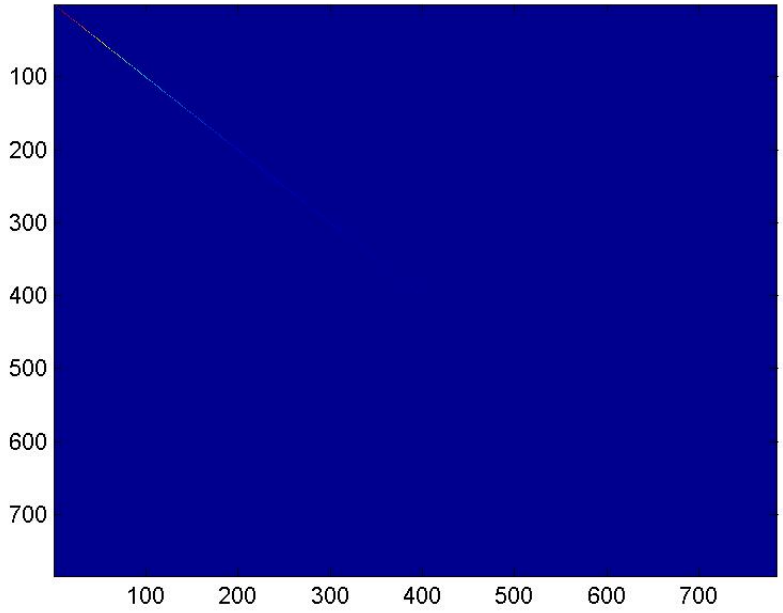


6. PCA whitening and regularisation.

Listing 2.10: Implement PCA with whitening and regularisation

```
% Without regularisation (set epsilon to 0 or close to 0),
epsilon = 0.1;
xPCAWhite = zeros(size(x));
% As we have compute, xRot = u' * x; xPCAWhite <784*60000>
xPCAWhite = diag(1./sqrt(diag(s)+epsilon)) * xRot;

% use the dimensioned x to do pca whitening
% xPCAWhite_rd = diag(1./sqrt(diag(s)+epsilon)) * u' * xHat;

```

```
10 %  Check our implementation of PCA whitening with and without
       regularisation.
11 %  PCA whitening without regularisation results a covariance matrix
12 %  that is equal to the identity matrix. PCA whitening with
       regularisation
13 %  results in a covariance matrix with diagonal entries starting
       close to
14 %  1 and gradually becoming smaller
15 covar = xPCAWhite * xPCAWhite' / size(xPCAWhite,2);
16 figure('name','Visualisation of covariance matrix');
17 imagesc(covar);
```



Listing 2.11: Implement ZCA whitening

7.

```
1 xZCAWhite = zeros(size(x));
2 % xZCAWhite <784*60000>
3 xZCAWhite = u * xPCAWhite;
4 % Visualise the data, and compare it to the raw data.
5 % You should observe that the whitened images have enhanced edges.
6 figure('name','ZCA whitened images');
7 display_network(xZCAWhite(:,randsel));
8 figure('name','Raw images');
9 display_network(x(:,randsel));
```

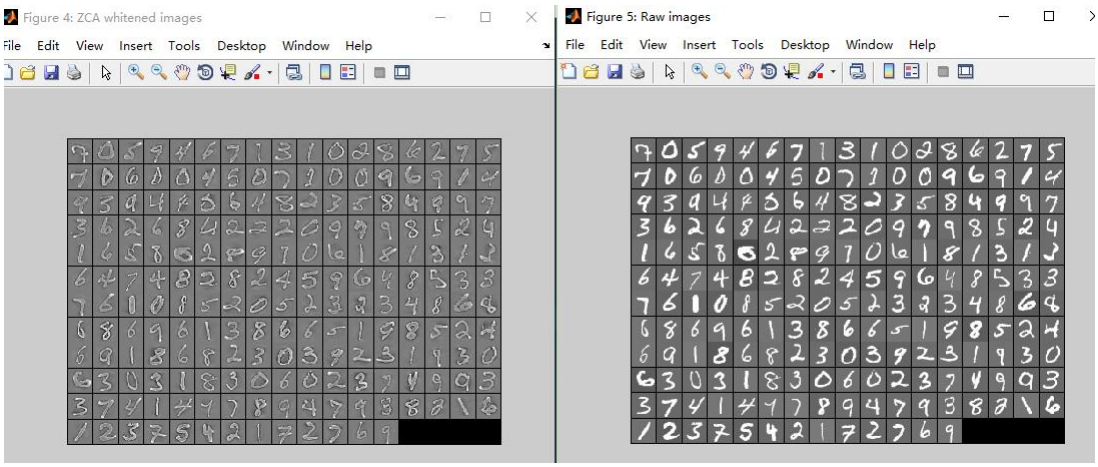


Figure 4: ZCA whitened images
File Edit View Insert Tools Desktop Window Help
Figure 5: Raw images
File Edit View Insert Tools Desktop Window Help

2.3 Reconstruction ICA

Independent Component Analysis (ICA) allows us to generate sparse representations of whitened data. But ICA, has difficulties arise when the number of features (rows of $\boldsymbol{W}$ matrix), exceed the dimensionality of input, $\boldsymbol{x}$.

2.4 Self-raught Learning

I have not completed it, but I know its principles.