Convolutional Neural Networks

Akshat Dave, Muhammad Ahmed Riaz, Nicholas Kinkade
Computer Science and Engineering
University of California, San Diego
{akdave,mriaz,nkinkade}@ucsd.edu

(Programming Assignment 4, Submitted : 5-June-2015)

Abstract

Convolutional neural networks have been used successfully in recent literature for learning applications, and particularly for classification of images. In this report, we explore the application of convolutional neural networks to multiclass classification, specifically, in the classification of handwritten digits in the *MNIST* dataset (as directed in the fourth programing assignment of **CSE291: Neural Networks**). We present a simple to use reconfigurable architecture for developing a convolutional neural network model. We evaluate the performance of some of these architectures/models on the *MNIST* dataset. Interestingly, we discover that the max pooling marginally improves accuracy and that the network architecture has a significant impact on the iteration profile. We report an accuracy of greater than 91% on the test data classification of the *MNIST* images.

1 Introduction

This report discusses the problems posed in the fourth programming assignment[1] for **CSE291: Neural Networks**. We use convolutional neural networks for the purpose of classification, namely for digit recognition from the *MNIST* dataset. Our implementational contributions include:

1. Support for a reconfigurable user-defined architecture
2. Support for several activation functions including rectified linear units, logistic sigmoid and modified tanh
3. Support for both mean-pooling and max-pooling
4. Support for arbitrary convolutional kernel size, arbitrary pooling window and arbitrary feature map count at each convolutional layer
5. All parameter gradient checking utility

As we have designed a generalizable architecture, we claim and subsequently demonstrate in the experiments section that our system handles each of the specified requirements of PA4. We cite sources [1] and [2] as an inspiration for datastructures and optimizations for batch processing.

The remaining paper is organized as follows: Section 2 describes the method, Section 3 discusses the experiments, while Section 4 concludes the paper.

2 Method

Given an input space X, we wish to map this to an output space y. We refer to X as the features while y as the labels. In the case that we have the input space as the form of images, we model this mapping

[1] http://tinyurl.com/pbcyp8c

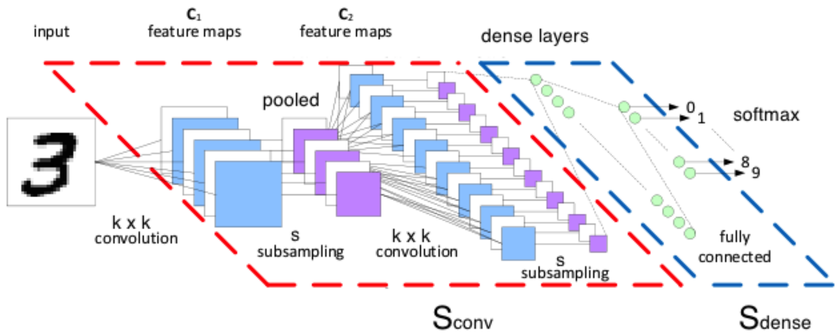

Figure 1: Schematic of a generic convolutional neural network. The dense section S_{dense} is reconfigurable to allow $n_d \geqslant 1$ layers where the last layer is a softmax layer. Similarly, the convolutional layer is reconfigurable to allow $n_c \geqslant 1$ layers where the first layer is an input layer. This image is an edited graphic from the TUE-course page

$h : X \mapsto y$ as a convolutional neural network parameterized by weights at each layer. The design of the neural network is similar to that presented in Fig. 1, where S_{conv} is the convolutional section and S_{dense} densely connected section. The convolutional section (S_{conv}) accomodates the convolution and the pooling layers, while the densely connected section (S_{dense}) accomodates a multilayered densely connected neural network. We constrain the design to allow S_{conv} only before the S_{dense} (as per requirements) however the order of layers within the network is free to configure. Our convolutional section supports both mean-pooling and max-pooling layers. The advantage of such a feed forward multilayered architecture is that it is able to learn internal representations. The mapping h is learnt by the backpropagation algorithm. Since in our application we use these networks for multi-class classification, the final layer is a softmax layer. We demonstrate that training this network is efficient in that the gradients can be analytically computed for training. We proceed to discuss backpropagation (as described in [2]):

Suppose we have a fixed training set $(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})$ of m training examples. We are able to train our neural network using batch gradient descent. Specifically, for a mini-batch of size m with training examples $(x^{(i)}, y^{(i)})$, we have the cross entropy loss function to be:

$$L(W, b, \theta) = -\sum_{i=1}^{m} \sum_{k=1}^{K} \mathbf{1}\{y^{(i)} = k\} \log \frac{\exp \theta^{(k)T} z^{(n_d,i)}}{\sum_{j=1}^{K} \exp \theta^{(k)T} z^{(n_d,i)}} \text{ where } z^{(n_d,i)} = h_{W,b}(x^{(i)})$$

Note above that W, b are the weights and biases of all the layers in the network except the final softmax layer which is parameteried by θ for explanatory purposes. More details can be found in the UFLDL lecture notes (notes-link). Our aim is to find the parameters that minimize the loss function, in other words:

$$\{W^*, b^*, \theta^*\} = \arg\min_{W,b,\theta} \Big(-\sum_{i=1}^{m} \sum_{k=1}^{K} \mathbf{1}\{y^{(i)} = k\} \log \frac{\exp \theta^{(k)T} h_{W,b}(x^{(i)})}{\sum_{j=1}^{K} \exp \theta^{(k)T} h_{W,b}(x^{(i)})} \Big)$$

We generally regularize this loss function by adding a penalty to the norm of the parameters. Thus, given a training set of m examples, we can define the regularized loss function to be:

$$L(W, b, \theta) \quad = \quad \frac{1}{m} \sum_{i=1}^{m} L(W, b, \theta; x^{(i)}, y^{(i)}) + \frac{\lambda}{2} \left\| \mathbf{vec}\left(\begin{bmatrix} W \\ b \\ \theta \end{bmatrix} \right) \right\|^2 \qquad (1)$$

[2] http://ufldl.stanford.edu/

In order to minimize the loss function, we proceed to take the derivative (since the loss function is differentiable) in order to design the update rule for gradient descent. We use the chain rule to minimize the loss function with respect to the weights at each layer. At this point we augment the weights W and θ such that $W = (W, \theta)$. It can be shown that the gradient of the loss function with respect to the parameters of the densely connected layers is given by:

$$\nabla_{W^{(l)}} L(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T, \text{ and, } \nabla_{b^{(l)}} L(W, b; x, y) = \delta^{(l+1)} \;\; \forall n_c + 1 \leqslant l \leqslant n_d$$

while for the convolutional layers, the gradient with respect to the $k^t h$ filter is given by:

$$\nabla_{W_k^{(l)}} L(W, b; x, y) = \sum_{i=1}^{m} (a_i^{(l)})^T * \Phi(\delta_k^{(l+1)}), \text{ and, } \nabla_{b_k^{(l)}} L(W, b; x, y) = \sum_{a,b} (\delta_k^{(l+1)})_{a,b} \;\; \forall 1 \leqslant l \leqslant n_c$$

Where $a^{(l)}$ is the activation vector of the l^{th} layer and Φ is a horizontal matrix flip/reflection operator and $(*)$ is the convolution operator. We recall that for a network with n_c convolution/pooling layers and n_d densely connected layers we use a backpropagation rule similar to multilayered neural networks. Here we use the shorthand δ to indicate the propagated error terms. For the densely connected layers we have the update rule:

$$\delta^{(l)} = \begin{cases} -(y - a^{(n_l)}) \cdot f'(z^{(n_l)}), & \text{if } l = n_d \\ (W^{(l)})^T \delta^{(l+1)}) \cdot f'(z^{(l)}) & \text{if } n_c + 1 \leqslant l \leqslant n_d - 1 \end{cases}$$

While for the convolutional/pooling layers we have:

$$\delta_k^{(l)} = \Psi_s((W_k^{(l)})^T \delta_k^{(l+1)})) \cdot f'(z_k^{(l)}), \text{if } 1 \leqslant l \leqslant n_c$$

Where $f'(z^{(l)})$ is the derivative of the activation function at the layer l with input z and Ψ_s is an upsampling function with sampling factor s. In the absence of a pooling layer, $s = 1$. We define $(\cdot)$ as the hadamard product.

We allow our network to work with various activation functions including the logistic sigmoid, the modified hyperbolic tangent and the rectified linear function. The derivative $f'(z^{(l)})$ for the logistic sigmoid function is simply given by $f'(z^{(l)}) = z^{(l)} \cdot (1 - z^{(l)})$. Here, $z^{(l)}$ is computed as (for all densely connected layers):

$$z^{(l)} = W^{(l)} f(z^{(l-1)}), \;\; \forall n_c + 1 \leqslant l \leqslant n_d$$

While for the convolutional layers it is given by:

$$z_k^{(l)} = \sum_{i=1}^{|l-1|} W_{ki}^{(l)} * f(z_i^{(l-1)}), \;\; \forall 1 \leqslant l \leqslant n_c$$

Where $|l-1|$ is the number of feature maps output from layer $l-1$. Finally, we have the update equations:

$$W^{(l)} = W^{(l)} - \alpha(\frac{\Delta W^{(l)}}{m} + \lambda W^{(l)}), \text{ and,} \tag{2a}$$

$$b^{(l)} = b^{(l)} - \alpha(\frac{\Delta b^{(l)}}{m}) \tag{2b}$$

Where the update for $\Delta W^{(l)}$ and $\Delta b^{(l)}$ are given by

$$\Delta W^{(l)} := \Delta W^{(l)} + \nabla_{W^{(l)}} L(W, b; x, y) \text{ and } \Delta b^{(l)} := \Delta b^{(l)} + \nabla_{b^{(l)}} L(W, b; x, y)$$

We optimize this function by minibatch gradient descent. The algorithm for gradient descent is generic, in-that if the batch size is $m = 1$, we have stochastic gradient descent, is $1 < m < |\mathbf{X}|$, we have minibatch gradient descent, and if $m = |\mathbf{X}|$ we have batch gradient descent. The algorithm is given in Alg. 2. For gradient descent we use momentum for faster convergence. Our stopping criteria is based upon maximum iterations.

For verifying correctness of our gradients, we perform a gradient check using the following test:

$$\text{verify if } (y^{(i)} - h_\theta(x^{(i)}))x_j^{(i)} \approx \lim_{\epsilon \to 0} \frac{L(\theta_j + \epsilon) - L(\theta_j - \epsilon)}{2\epsilon}$$

The algorithm is provided in Alg. 1.

Algorithm 1 Gradient check algorithm

procedure DOGRADIENTCHECK(θ, ϵ)
    $t \leftarrow \phi$ (equality test flag)
    for each j **do**
        $\hat{g}_j \leftarrow \big(L(\theta_j + \epsilon) - L(\theta_j - \epsilon)\big)/2\epsilon$
        $g_j \leftarrow \nabla_{\theta_j} L(\theta)$
        if $g_j \approx \hat{g}_j$ **then**
            $t_j \leftarrow 1$ (success)
        else
            $t_j \leftarrow 0$ (fail)
        end if
    end for
return t
end procedure

Algorithm 2 Gradient descent for error backpropagation. This algorithm is used for stochastic gradient descent when $m = 1$, for mini-batch gradient descent when $1 < m < |\mathbf{X}|$, and, for batch gradient descent when $m = |\mathbf{X}|$, where $|\mathbf{X}|$ is the size of the training dataset). Note that this algorithm was modified to include momentum (found in the appendix) to improve the results

procedure DOBACKPROP(α, m, N)
Inputs: (1) α : learning rate (2) m : batch size (3) N : maximum iterations
    $i \leftarrow 1$, randomly initialize weights
    while $i \leqslant N$ **do**
        Perform a feedforward pass, computing the activations for each layer
        For the output layer (layer n_l), $\delta^{(n_l)} \leftarrow -(y - a^{(n_l)}) \cdot f'(z^{(n_l)})$
        $\forall l \in \{n_l - 1, n_l - 2, n_l - 3, \ldots, n_c + 1\}$ set $\delta^{(l)} \leftarrow ((W^{(l)})^T \delta^{(l+1)}) \cdot f'(z^{(l)})$
        $\forall l \in \{n_c, n_l - 1, n_l - 2, \ldots, 1\}$ set $\delta_k^{(l)} \leftarrow \Psi_s(((W_k^{(l)})^T \delta_k^{(l+1)})) \cdot f'(z_k^{(l)})$
        Compute : $\nabla_{W^{(l)}} L(W, b; x, y) = \delta^{(l+1)}(a^{(l)})^T$ and $\nabla_{b^{(l)}} L(W, b; x, y) = \delta^{(l+1)}$
        $\forall l, \Delta W^{(l)} \leftarrow 0$, $\Delta b^{(l)} \leftarrow 0$
        for $1 \leqslant i \leqslant m$ **do**
            Use backprop from sec. 2 to compute $\nabla_{W^{(l)}} L(W, b; x, y)$ and $\nabla_{b^{(l)}} L(W, b; x, y)$
            $\Delta W^{(l)} \leftarrow \Delta W^{(l)} + \nabla_{W^{(l)}} L(W, b; x, y)$ and $\Delta b^{(l)} \leftarrow \Delta b^{(l)} + \nabla_{b^{(l)}} L(W, b; x, y)$
        end for
        Update parameters
        $W^{(l)} \leftarrow W^{(l)} - \alpha(\frac{\Delta W^{(l)}}{m} + \lambda W^{(l)})$ and $b^{(l)} \leftarrow b^{(l)} - \alpha(\frac{\Delta b^{(l)}}{m})$
        $i \leftarrow i + 1$
    end while
return W, b
end procedure

model architecture name	architecture description	layer details
MA1	1 input layer 2 convolutional layers 1 pooling layer 1 densely connected layer 1 softmax output layer	1 channel input 2 conv. feat maps from 9×9 kernels 4 conv. feat maps from 5×5 kernels 4×4 max pooling window 30 fully connected units layer 10 output units
MA2	1 input layer 2 convolutional layers 2 pooling layers 1 densely connected layer 1 softmax output layer	1 channel input 2 conv. feat maps from 5×5 kernels 2×2 pooling window 2 conv. feat maps from 5×5 kernels 4×4 pooling window 10 fully connected units layer 10 output units
MA3	1 input layer 1 convolutional layer 1 pooling layer 1 softmax output layer	1 channel input 20 conv. feat maps from 9×9 kernels 2×2 pooling window 10 output units

Table 1: Detailed specification of the models developed for experiments

3 Experiments

In this section we discuss the datasets used for each task, the experimental results and the parameters used by our models.

3.1 Dataset description

We use the *MNIST* dataset for our experiments. The dataset comprises $60,000$ labeled training examples and $10,000$ labeled test images. The task is to identify the digit class $1, 2, \ldots 10$ for the test data. We train a few convolutional neural network models and discuss these here.

3.2 Training and results

For training the parameters we use backpropagation with loss minimization using mini-batch gradient descent (MGD) as discussed in section 2. Gradient descent is a first order optimization method which uses a first order Taylor approximation to update the parameters in the direction of the negative gradient of the loss function.

Table 1 enlists the model architectures that we evaluated with gradient checking. We verify whether the analytical solution agrees with the numerical approximation as described in section 2 for each of the architectures (**MA1**, **MA2**, **MA3**) using each of the three activation functions viz. logistic sigmoid (σ), modified tanh (τ) and the rectified linear activation (ρ). The gradient check passed for each of the activation functions for each of the architectures and some of the results of the gradient check are shown in Table 4

Table 2 enlists the models developed for evaluation on the *MNSIT* dataset. The parameters of the model are provided in the same table along with the architecture used.

For each of the models (**CNN1** and **CNN2)** discussed, the results on *MNIST* are provided in Table 3. Finally, Fig. 2 provides all the model training iteration profiles.

Model	parameter	description	value
CNN1	α	learning rate	$1e-1$
	N_m	max iterations for mini-batch gradient descent	$2e2$
	m	mini-batch size	256
	γ	momentum	0.95
	ϵ	epochs	3
	-	pooling type	max
	$f()$	activation function	tanh
	-	architecture used	**MA2**
CNN2	α	learning rate	$1e-1$
	N_m	max iterations for mini-batch gradient descent	$1e2$
	m	mini-batch size	256
	γ	momentum	0.95
	ϵ	epochs	3
	-	pooling type	max
	$f()$	activation function	rectified linear
	-	architecture used	**MA3**
CNN3	α	learning rate	$1e-1$
	N_m	max iterations for mini-batch gradient descent	$1e2$
	m	mini-batch size	256
	γ	momentum	0.95
	ϵ	epochs	3
	-	pooling type	max
	$f()$	activation function	modified tanh
	-	architecture used	**MA3**
CNN4	α	learning rate	$1e-1$
	N_m	max iterations for mini-batch gradient descent	$1e2$
	m	mini-batch size	256
	γ	momentum	0.95
	ϵ	epochs	3
	-	pooling type	mean
	$f()$	activation function	modified tanh
	-	architecture used	**MA3**

Table 2: Shows the parameters for each of the model architectures used for *MNIST* classification

model	training accuracy	test accuracy	training time
CNN1	90.1%	90.7%	163s
CNN2	91.2%	92.1%	625s
CNN3	91.0%	91.68%	632s
CNN4	89.9%	90.50%	467s

Table 3: Results of experiments for each of the models

Model Architecture	True gradient $\nabla L(\theta)$	Numerical approximation $\frac{(L(\theta+\epsilon)-L(\theta+\epsilon))}{2\epsilon}$	Relative error	ϵ
MA1	$9.506783e-7$ $9.540871e-7$ $2.344425e-7$ $3.84988e-7$	$9.506751e-7$ $9.540813e-7$ $2.344502e-7$ $3.849920e-7$	$3.20e-12$ $5.81e-12$ $7.74e-12$ $3.79e-12$	$1e-4$ $1e-4$ $1e-4$ $1e-4$
MA2	$6.4193e-4$ $6.539991e-7$ $7.383305e-7$ $5.741183e-7$ $2.214921e-7$	$6.4193e-4$ $6.539902e-7$ $7.383316e-7$ $5.741252e-7$ $2.214905e-7$	$5.81e-13$ $8.93e-12$ $1.10e-12$ $7.96e-13$ $1.65e-12$	$1e-4$ $1e-4$ $1e-4$ $1e-4$ $1e-4$
MA3	$2.184758e-4$ $1.547129e-4$ $1.489933e-5$ $-1.406128e-4$	$2.184758e-4$ $1.547129e-4$ $1.489933e-5$ $-1.406128e-4$	$2.43e-12$ $9.33e-13$ $2.23e-12$ $8.41e-12$	$1e-4$ $1e-4$ $1e-4$ $1e-4$

Table 4: Shows numerical approximation and its comparison to the true gradient

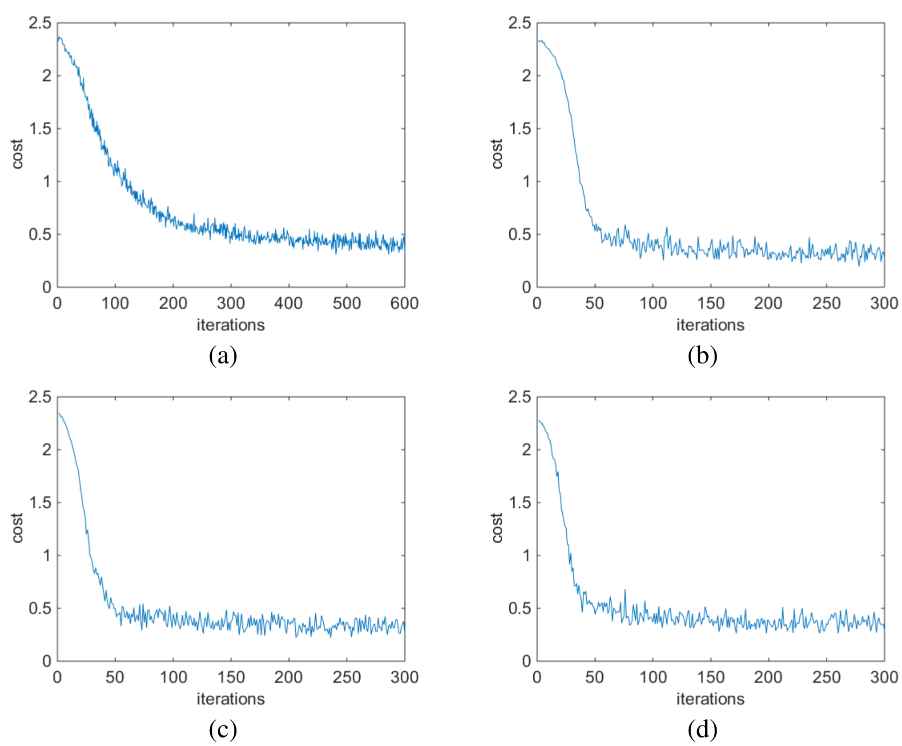

Figure 2: Figures show the iteration profiles for the models developed (a) **CNN1** (b) **CNN2** (c) **CNN3** (d) **CNN4**

3.3 Discussion

We find that classification between the *MNIST* digits yields over 90% accuracy. The four models we train perform relatively well with **CNN1** reporting an accuracy of 90.7% ,**CNN2** with an accuracy of 92.1%, **CNN3** with an accuracy 91.68, **CNN4** with an accuracy of 90.50%.

We observe max pooling increases the accuracy as compared to mean pooling as seen in the 1% increase in accuracy between **CNN4** and **CNN3**. Additionally, we observe that the architecture has an effect on the convergence trend. We see that architectures **CNN2**, **CNN3** and **CNN4** have similar convergence trends as compared to **CNN1** which has a different architecture. As **CNN1** is deeper, it drops slower in cost as compared to **CNN2**, **CNN3** and **CNN4**.

We observed that the logistic sigmoid (iteration profiles not included) gives slightly lower performance (85.5% training accuracy and 86.5% test accuracy) compared to using the modified tanh and the rectified linear units. We also observe that the training time increases with the size of the batch.

The gradient check algorithm ensures that the relative error in between the approximation and the true gradient is always less than $O(\epsilon)$.

4 Conclusion

We have demonstrated that our proposed system allows the design of reconfigurable convolutional neural network architectures with several convolutional/pooling layers and several densely connected layers. We have presented various exemplar model architectures, viz. **MA1**, **MA2** and **MA3**. Finally, we have demonstrated that the models created by our system, namely, **CNN1**, **CNN2**, **CNN3** and **CNN4** all perform classifaction on the *MNIST* dataset with over 90% accuracy.

References

[1] A. Vedaldi and K. Lenc, "MatConvNet – Convolutional Neural Networks for MATLAB", *CoRR* (2014).

[2] "Prediction as a candidate for learning deep hierarchical models of data", *Palm* (2012)

Appendix : Source Code

Listing 1: calculate_cost.m

```
function cost = calculate_cost(netOut,labels, model, lambda)
%regularization term
reg = 0;
for l=1:numel(model.conv.layers)
    if(strcmp(model.conv.layers{l}.type,'conv'))
        for i = 1:numel(model.conv.layers{l}.W)
            for j = 1:numel(model.conv.layers{l}.W{i})
                reg = reg + sum(sum((model.conv.layers{l}.W{i}{j}.^2)));
            end
        end
    end
end
for i=1:numel(model.dense.layers)
    reg = reg + sum(sum((model.dense.layers{i}.W.^2)));
end
cost = reg * lambda/2;

%cost function
m = size(netOut,2);
for i = 1:m
    cost = cost - log(netOut(labels(i),i))/m;
end
end
```

Listing 2: cnnConvolve.m

```
function convolvedFeatures = cnnConvolve(filterDim, numFilters, ...
                                         images, W, b, activationType)
%cnnConvolve Returns the convolution of the features given by W and b with
%the given images
%
% Parameters:
%  filterDim - filter (feature) dimension
%  numFilters - number of feature maps
%  images - large images to convolve with, matrix in the form
%           images(r, c, image number)
%  W, b - W, b for features from the sparse autoencoder
%         W is of shape (filterDim,filterDim,numFilters)
%         b is of shape (numFilters,1)
%  activationType - 1 : logistic sigmoid function
%                   2 : rectified linear unit function
%                   3 : funny tanh function
%
% Returns:
%  convolvedFeatures - matrix of convolved features in the form
%                      convolvedFeatures(imageRow, imageCol, featureNum, imageNum)

numImages = size(images, 3);
imageDim = size(images, 1);
convDim = imageDim - filterDim + 1;

convolvedFeatures = zeros(convDim, convDim, numFilters, numImages);

[activationFunc, ~] = select_act(activationType);

% Instructions:
%   Convolve every filter with every image here to produce the
%   (imageDim - filterDim + 1) x (imageDim - filterDim + 1) x numFeatures x numImages
%   matrix convolvedFeatures, such that
%   convolvedFeatures(imageRow, imageCol, featureNum, imageNum) is the
%   value of the convolved featureNum feature for the imageNum image over
%   the region (imageRow, imageCol) to (imageRow + filterDim - 1, imageCol + filterDim - 1)
%
% Expected running times:
%   Convolving with 100 images should take less than 30 seconds
%   Convolving with 5000 images should take around 2 minutes
%   (So to save time when testing, you should convolve with less images, as
%   described earlier)


for imageNum = 1:numImages
  for filterNum = 1:numFilters

    % convolution of image with feature matrix
```

```
    convolvedImage = zeros(convDim, convDim);

    % Obtain the feature (filterDim x filterDim) needed during the convolution
    %OUR CODE
    filter = W(:,:,filterNum);

    % Flip the feature matrix because of the definition of convolution, as explained later
    filter = rot90(squeeze(filter),2);

    % Obtain the image
    im = squeeze(images(:, :, imageNum));

    % Convolve "filter" with "im", adding the result to convolvedImage
    % be sure to do a 'valid' convolution
    %OUR CODE
    convolvedImage = convolvedImage + conv2(im, filter, 'valid');

    % Add the bias unit
    % Then, apply the activation function to get the hidden activation
    %OUR CODE
    bias = b(filterNum);
    convolvedImage = convolvedImage + bias;
    convolvedImage = activationFunc(convolvedImage);

    convolvedFeatures(:, :, filterNum, imageNum) = convolvedImage;
  end
end
end
```

Listing 3: cnnCost.m

```
function [cost, grad, predDist] = cnnCost(theta,images,labels,numClasses,...
                                model, lambda, ...
                                activationType, ...
                                pred_only)
%default to no prediction
if ~exist('pred_only','var')
    pred_only = false;
end;

%reshape parameters
[model] = cnnParamsToStack(theta, model);

%deciding activation function and gradient of activation function
[act_fun, grad_fun, ~] = select_act(activationType);

%forward propogate
[model] = conv_forward_pass(images, model, act_fun, grad_fun);
[convOut] = vectorize_conv_out(model);
[model, netOut] = hidden_forward_pass(convOut, model, act_fun, grad_fun);

%if we are only performing prediction
if pred_only
    predDist = netOut; % prediction distributions
```

```
    grad = [];
    cost = -1;
    return;
end

%compute cost
cost = calculate_cost(netOut, labels, model, lambda);

%backprop
[model, delta] = hidden_backprop(labels, model, netOut, numClasses
    , lambda);
[deltaOut] = vectorize_delta_out(delta);
[model] = conv_backprop(model, deltaOut, lambda);

%convert deltas
gradModel = makeGradModel(model);

%convert gradients to vector
grad = cnnStackToParams(gradModel);
end
```

Listing 4: cnnExercise.m

```
%% Convolution and Pooling Exercise

%  Instructions
%  ------------
%
%  This file contains code that helps you get started on the
%  convolution and pooling exercise. In this exercise, you will
    only
%  need to modify cnnConvolve.m and cnnPool.m. You will not need
    to modify
%  this file.

%
    %======================================================================

%% STEP 0: Initialization and Load Data
%  Here we initialize some parameters used for the exercise.

imageDim = 28;         % image dimension

filterDim = 8;          % filter dimension
numFilters = 100;         % number of feature maps

numImages = 60000;    % number of images

poolDim = 3;          % dimension of pooling region

% Here we load MNIST training images
addpath ../common/;
images = loadMNISTImages('../common/train-images-idx3-ubyte');
images = reshape(images,imageDim,imageDim,numImages);

W = randn(filterDim,filterDim,numFilters);
b = rand(numFilters, 1);

```

```
% %======================================================================
%% STEP 1: Implement and test convolution
%  In this step, you will implement the convolution and test it on
%  on a small part of the data set to ensure that you have implemented
%  this step correctly.

%% STEP 1a: Implement convolution
%  Implement convolution in the function cnnConvolve in cnnConvolve.m

%% Use only the first 8 images for testing
convImages = images(:, :, 1:8);

convolvedFeatures = cnnConvolve(filterDim, numFilters, convImages, W, b, 1);

%% STEP 1b: Checking your convolution
%  To ensure that you have convolved the features correctly, we have
%  provided some code to compare the results of your convolution with
%  activations from the sparse autoencoder

% For 1000 random points
for i = 1:1000
    filterNum = randi([1, numFilters]);
    imageNum = randi([1, 8]);
    imageRow = randi([1, imageDim - filterDim + 1]);
    imageCol = randi([1, imageDim - filterDim + 1]);

    patch = convImages(imageRow:imageRow + filterDim - 1, imageCol:imageCol + filterDim - 1, imageNum);

    feature = sum(sum(patch.*W(:,:,filterNum)))+b(filterNum);
    feature = 1./(1+exp(-feature));

    if abs(feature - convolvedFeatures(imageRow, imageCol, filterNum, imageNum)) > 1e-9
        fprintf('Convolved feature does not match test feature\n');
        fprintf('Filter Number    : %d\n', filterNum);
        fprintf('Image Number      : %d\n', imageNum);
        fprintf('Image Row         : %d\n', imageRow);
        fprintf('Image Column      : %d\n', imageCol);
        fprintf('Convolved feature : %0.5f\n', convolvedFeatures(imageRow, imageCol, filterNum, imageNum));
        fprintf('Test feature : %0.5f\n', feature);
        error('Convolved feature does not match test feature');
    end
end

disp('Congratulations! Your convolution code passed the test.');

% %======================================================================
```

```
%% STEP 2: Implement and test pooling
% Implement pooling in the function cnnPool in cnnPool.m

%% STEP 2a: Implement pooling
% NOTE: Implement cnnPool in cnnPool.m first!
pooledFeatures = cnnPool(poolDim, convolvedFeatures, 1);

%% STEP 2b: Checking your pooling
% To ensure that you have implemented pooling, we will use your pooling
% function to pool over a test matrix and check the results.

testMatrix = reshape(1:64, 8, 8);
expectedMatrix = [mean(mean(testMatrix(1:4, 1:4))) mean(mean(testMatrix(1:4, 5:8))); ...
                  mean(mean(testMatrix(5:8, 1:4))) mean(mean(testMatrix(5:8, 5:8))); ];

testMatrix = reshape(testMatrix, 8, 8, 1, 1);

pooledFeatures = squeeze(cnnPool(4, testMatrix, 1));

if ~isequal(pooledFeatures, expectedMatrix)
    disp('Pooling incorrect');
    disp('Expected');
    disp(expectedMatrix);
    disp('Got');
    disp(pooledFeatures);
else
    disp('Congratulations! Your pooling code passed the test.');
end
```

Listing 5: cnnInitParams.m

```
function [theta, model] = cnnInitParams(imageDim,model,init_zero)
%initialize convolutional layers
inCount = 1;
mapDim = [imageDim imageDim];
for l = 1 : numel(model.conv.layers)
        if strcmp(model.conv.layers{l}.type, 'conv')
        mapDim = mapDim - model.conv.layers{l}.kernelDim + 1;
        fanOut = model.conv.layers{l}.outCount * model.conv.layers{l}.kernelDim * model.conv.layers{l}.kernelDim ;
        for j = 1 : model.conv.layers{l}.outCount
            fanIn = inCount * model.conv.layers{l}.kernelDim * model.conv.layers{l}.kernelDim;
            for i = 1 : inCount
                s = sqrt(6 / (fanIn + fanOut));
                model.conv.layers{l}.W{i}{j} = rand(model.conv.layers{l}.kernelDim)*2*s - s;
                if(init_zero)
                    model.conv.layers{l}.W{i}{j}  = model.conv.layers{l}.W{i}{j}*0;
                end
            end
            model.conv.layers{l}.b{j} = 0;
        end
        inCount = model.conv.layers{l}.outCount;
        end

```

```
    if strcmp(model.conv.layers{l}.type, 'pool')
        mapDim = mapDim / model.conv.layers{l}.poolDim;
    end
end

model.outDim = prod(mapDim) * inCount;

%initialize dense layers
for l = 1 : numel(model.dense.layers)
    if l > 1
        fanIn = model.dense.layers{l-1}.outCount;
    else
        fanIn = model.outDim;
    end;
    fanOut = model.dense.layers{l}.outCount;
    s = sqrt(6) / sqrt(fanIn + fanOut);

    model.dense.layers{l}.W = rand(fanOut, fanIn)*2*s - s;
    if(init_zero)
        model.dense.layers{l}.W = model.dense.layers{l}.W*0;
    end
    model.dense.layers{l}.b = zeros(fanOut, 1);
end

%convert to vector form
theta = cnnStackToParams(model);
end
```

Listing 6: cnnParamsToStack.m

```
function [model] = cnnParamsToStack(theta, model)
cur_pos = 1;

%convolutional layers
inCount = 1;
for l = 1 : numel(model.conv.layers)
    if strcmp(model.conv.layers{l}.type, 'conv')
        for j = 1 : model.conv.layers{l}.outCount
            for i = 1 : inCount
                wlen = model.conv.layers{l}.kernelDim ^ 2;
                model.conv.layers{l}.W{i}{j} = reshape(theta(
                    cur_pos:cur_pos+wlen-1), model.conv.layers{l}.
                    kernelDim, model.conv.layers{l}.kernelDim);
                cur_pos = cur_pos + wlen;
            end
            model.conv.layers{l}.b{j} = theta(cur_pos);
            cur_pos = cur_pos + 1;
        end
        inCount = model.conv.layers{l}.outCount;
    end
end

%dense layers
hiddenDepth = numel(model.dense.layers);
inCount = model.outDim;
for d = 1:hiddenDepth
    outCount = model.dense.layers{d}.outCount;

    wlen = double(outCount * inCount);
```

```
    model.dense.layers{d}.W = reshape(theta(cur_pos:cur_pos+wlen
        -1), outCount, inCount);
    cur_pos = cur_pos+wlen;

    blen = outCount;
    model.dense.layers{d}.b = reshape(theta(cur_pos:cur_pos+blen
        -1), outCount, 1);
    cur_pos = cur_pos+blen;

    inCount = outCount;
end
end
```

Listing 7: cnnPool.m

```
function pooledFeatures = cnnPool(poolDim, convolvedFeatures,
    poolType)
%cnnPool Pools the given convolved features
%
% Parameters:
%  poolDim - dimension of pooling region
%  convolvedFeatures - convolved features to pool (as given by
    cnnConvolve)
%                      convolvedFeatures(imageRow, imageCol,
    featureNum, imageNum)
%  poolType - 1 : MEAN
%             2 : MAX
%
% Returns:
%  pooledFeatures - matrix of pooled features in the form
%                   pooledFeatures(poolRow, poolCol, featureNum,
    imageNum)
%

numImages = size(convolvedFeatures, 4);
numFilters = size(convolvedFeatures, 3);
convolvedDim = size(convolvedFeatures, 1);

pooledFeatures = zeros(convolvedDim / poolDim, ...
        convolvedDim / poolDim, numFilters, numImages);

% Instructions:
%   Now pool the convolved features in regions of poolDim x
    poolDim,
%   to obtain the
%   (convolvedDim/poolDim) x (convolvedDim/poolDim) x numFeatures
    x numImages
%   matrix pooledFeatures, such that
%   pooledFeatures(poolRow, poolCol, featureNum, imageNum) is the
%   value of the featureNum feature for the imageNum image pooled
    over the
%   corresponding (poolRow, poolCol) pooling region.
%
%   Use mean pooling here.
for imageNum = 1:numImages
  for filterNum = 1:numFilters
      convolvedFeature = convolvedFeatures(:, :, filterNum,
          imageNum);
      if(poolType == 1)
          %perform mean filtering
```

```
            feature = conv2(convolvedFeature, ones(poolDim) / (poolDim*poolDim) , 'same');
        else
            %perform max filtering
            feature = ordfilt2(convolvedFeature, poolDim*poolDim, true(poolDim));
        end
        %subsample
        pooledFeatures(:, :, filterNum, imageNum) = feature(ceil(poolDim / 2) : poolDim : end, ceil(poolDim / 2) : poolDim : end);
    end
end

end
```

Listing 8: cnnStackToParams.m

```
function theta = cnnStackToParams(model)
theta = [];
%convolutional layers
inCount = 1;
for l = 1 : numel(model.conv.layers)
    if strcmp(model.conv.layers{l}.type, 'conv')
        for j = 1 : model.conv.layers{l}.outCount
            for i = 1 : inCount
                theta = [theta; model.conv.layers{l}.W{i}{j}(:)];
            end
            theta = [theta; model.conv.layers{l}.b{j}];
        end
        inCount = model.conv.layers{l}.outCount;
    end
end

%dense layers
for d = 1:numel(model.dense.layers)
    theta = [theta ; model.dense.layers{d}.W(:) ; model.dense.layers{d}.b(:)];
end
end
```

Listing 9: cnnTrain.m

```
clear;
%convolutional neural network
addpath(genpath('../../'));

%network configuration
activationType = 3;
imageDim = 28;
numClasses = 10;
grad_check=true;
init_zero=false;
model.conv.layers = {
    struct('type', 'input')
%       struct('type', 'conv', 'outCount', 2, 'kernelDim', 5)
%       struct('type', 'pool', 'poolType', 'max', 'poolDim', 2)
%       struct('type', 'conv', 'outCount', 3, 'kernelDim', 5)
%       struct('type', 'pool', 'poolType', 'max', 'poolDim', 2)

```

```
    struct('type', 'conv', 'outCount', 20, 'kernelDim', 9)
    struct('type', 'pool', 'poolType', 'mean', 'poolDim', 2)

%     struct('type', 'conv', 'outCount', 2, 'kernelDim', 5)
%     struct('type', 'pool', 'poolType', 'max', 'poolDim', 2)
%     struct('type', 'conv', 'outCount', 3, 'kernelDim', 5)
%     struct('type', 'pool', 'poolType', 'max', 'poolDim', 4)
};
model.dense.layers = {
%     struct('outCount', 10)
%     struct('outCount', 10)
    struct('outCount', numClasses)
};

%load MNIST training images
addpath ../common/;
images = loadMNISTImages('../common/train-images-idx3-ubyte');
images = reshape(images,imageDim,imageDim,[]);
labels = loadMNISTLabels('../common/train-labels-idx1-ubyte');
labels(labels==0) = 10; % Remap 0 to 10

%initialize parameters
[theta, model] = cnnInitParams(imageDim,model,init_zero);

%gradient check
if(grad_check)
    ran = randperm(60000,100);
    gradCheckImages = images(:,:,ran);
    gradCheckLabels = labels(ran);
    gradient_check(numel(theta), 0.1, @(theta)cnnCost(theta, gradCheckImages, gradCheckLabels, numClasses, model, ...
        0.001, activationType, false), 3);
end

%training configuration
options.epochs = 3;
options.numIterations = 200;
options.minibatch = 256; %100
options.alpha = 0.1;
options.momentum = 0.95;

%train
tic;
opttheta = minFuncSGD(@(theta,images,labels)cnnCost(theta, images, labels, numClasses, model,...
                          0.001, activationType, false),theta,images,labels,options);
fprintf('Optimization took %f seconds.\n', toc);

%load MNIST testing images
testImages = loadMNISTImages('../common/t10k-images-idx3-ubyte');
testImages = reshape(testImages,imageDim,imageDim,[]);
testLabels = loadMNISTLabels('../common/t10k-labels-idx1-ubyte');
testLabels(testLabels==0) = 10; % Remap 0 to 10

%calculate training accuracy
[~,cost,preds] = cnnCost(opttheta, images, labels, numClasses, model,...
                          0.001, activationType, true);
```

```
[~, pred] = max(preds);
train_acc = mean(pred==reshape(labels, size(pred)));

%calculate test accuracy
[~, cost, preds] =  cnnCost(opttheta, testImages, testLabels, numClasses, model, ...
                        0.001, activationType, true);
[~, pred] = max(preds);
test_acc = mean(pred==reshape(testLabels, size(pred)));

%display accuracies
fprintf('Training Accuracy is %f\n', train_acc);
fprintf('Test Accuracy is %f\n', test_acc);
```

Listing 10: conv_backprop.m

```
function [model] = conv_backprop(model, deltaIn, lambda)
n = numel(model.conv.layers);

%reshape feature vector deltas into output map style
outActSize = size(model.conv.layers{n}.act{1});
outActDim = outActSize(1) * outActSize(2);
for j = 1 : numel(model.conv.layers{n}.act)
    if( length(outActSize)<3 )
        outActSize(3) = 1;
    end
    model.conv.layers{n}.d{j} = reshape(deltaIn(((j - 1) * outActDim + 1) : j * outActDim, :), outActSize(1), outActSize(2), outActSize(3));
end

%calculate deltas
for l = (n - 1) : -1 : 1
    %backprop previous layer
    backDeltas = cell(numel(model.conv.layers{l}.act), 1);
    if strcmp(model.conv.layers{l+1}.type, 'conv')
        for i = 1 : numel(model.conv.layers{l}.act)
            backDeltas{i} = zeros(size(model.conv.layers{l}.act{1}));
            for j = 1 : numel(model.conv.layers{l + 1}.act)
                backDeltas{i} = backDeltas{i} + convn(model.conv.layers{l + 1}.d{j}, flip(flip(model.conv.layers{l + 1}.W{i}{j}, 1), 2), 'full');
            end
        end
    elseif strcmp(model.conv.layers{l+1}.type, 'pool')
        for i = 1 : numel(model.conv.layers{l}.act)
            if(strcmp(model.conv.layers{l+1}.poolType, 'mean'))
                backDeltas{i} = expand(model.conv.layers{l + 1}.d{i}, [model.conv.layers{l + 1}.poolDim model.conv.layers{l + 1}.poolDim 1]) / model.conv.layers{l + 1}.poolDim ^ 2;
            end
            if(strcmp(model.conv.layers{l+1}.poolType, 'max'))
                backDeltas{i} = zeros(size(model.conv.layers{l + 1}.d{i},1)*model.conv.layers{l + 1}.poolDim, size(model.conv.layers{l + 1}.d{i},2)*model.conv.layers{l + 1}.poolDim, size(model.conv.layers{l + 1}.d{i},3));
                for j = 1 : size(model.conv.layers{l+1}.mask{i},3)
```

```
                    mask = model.conv.layers{l+1}.mask{i}(:,:,j);
                    deltas = model.conv.layers{l+1}.d{i}(:,:,j);
                    mask = mask(:);
                    deltas = deltas(:);
                    curDelta = backDeltas{i}(:,:,j);
                    curDelta(mask) = deltas;
                    backDeltas{i}(:,:,j) = curDelta;
                end
            end

        end
    end

    %multiply by gradient of activation function
    if ~strcmp(model.conv.layers{l}.type, 'input')
        for i = 1 : numel(model.conv.layers{l}.act)
            model.conv.layers{l}.d{i} = model.conv.layers{l}.actGrad{i} .* backDeltas{i};
        end
    end
end

%calculate gradients
for l = 2 : n
    if strcmp(model.conv.layers{l}.type, 'conv')
        for j = 1 : numel(model.conv.layers{l}.act)
            for i = 1 : numel(model.conv.layers{l - 1}.act)
                model.conv.layers{l}.dW{i}{j} = convn(flipall(model.conv.layers{l - 1}.act{i}), model.conv.layers{l}.d{j}, 'valid') / size(model.conv.layers{l}.d{j}, 3);
            end
            model.conv.layers{l}.db{j} = sum(model.conv.layers{l}.d{j}(:)) / size(model.conv.layers{l}.d{j}, 3);
        end
    end
end

%regularization
for l = 2 : n
    if(strcmp(model.conv.layers{l}.type,'conv'))
        for i = 1:numel(model.conv.layers{l}.W)
            for j = 1:numel(model.conv.layers{l}.W{i})
                model.conv.layers{l}.dW{i}{j} = model.conv.layers{l}.dW{i}{j} + model.conv.layers{l}.W{i}{j} * lambda;
            end
        end
    end
end
end
```

Listing 11: conv_forward_pass.m

```
function [ model ] = conv_forward_pass( images, model, act_fun, grad_fun )
n = numel(model.conv.layers);
model.conv.layers{1}.act{1} = images;
inCount = 1;
%for every layer
```

```
for l = 2 : n
    %convolution layer
    if strcmp(model.conv.layers{l}.type, 'conv')
        for j = 1 : model.conv.layers{l}.outCount
            sz = [model.conv.layers{l}.kernelDim - 1 model.conv.layers{l}.kernelDim - 1 0];
            z = zeros(size(model.conv.layers{l - 1}.act{1}) - sz);
            for i = 1 : inCount
                z = z + convn(model.conv.layers{l - 1}.act{i}, model.conv.layers{l}.W{i}{j}, 'valid');
            end
            z = z + model.conv.layers{l}.b{j};
            model.conv.layers{l}.act{j} = act_fun(z);
            model.conv.layers{l}.actGrad{j} = grad_fun(z);
        end
        inCount = model.conv.layers{l}.outCount;
    %pooling layer
    elseif strcmp(model.conv.layers{l}.type, 'pool')
        if strcmp(model.conv.layers{l}.poolType, 'mean')
            for j = 1 : inCount
                z = convn(model.conv.layers{l - 1}.act{j}, ones(model.conv.layers{l}.poolDim) / (model.conv.layers{l}.poolDim ^ 2), 'valid');   % !! replace with variable
                model.conv.layers{l}.act{j} = z(1 : model.conv.layers{l}.poolDim : end, 1 : model.conv.layers{l}.poolDim : end, :);
                model.conv.layers{l}.actGrad{j} = 1;
            end
        else
            for j = 1 : inCount
                z = zeros(size(model.conv.layers{l-1}.act{j}));
                mask = zeros(size(model.conv.layers{l-1}.act{j}));
                for i = 1 : size(model.conv.layers{l-1}.act{j}, 3)
                    [z(:,:,i), mask(:,:,i)] = slideMax(model.conv.layers{l-1}.act{j}(:,:,i), model.conv.layers{l}.poolDim);
                end
                model.conv.layers{l}.act{j} = z(1 : model.conv.layers{l}.poolDim : end, 1 : model.conv.layers{l}.poolDim : end, :);
                model.conv.layers{l}.actGrad{j} = 1;
                model.conv.layers{l}.mask{j} = mask(1 : model.conv.layers{l}.poolDim : end, 1 : model.conv.layers{l}.poolDim : end, :);
            end
        end
    end
end
end

function [xMax, mask] = slideMax(x, poolDim)
%sliding max pooling, also gives mask with indications where
%max pooling came from
%can certainly be more optimized
poolDim = poolDim-1;
[width, height] = size(x);
xMax = zeros(width,height);
mask = zeros(width,height);
```

```
for i = 1 : width
    for j = 1 : height
        maxVal = -Inf;
        maxK = -1;
        maxL = -1;
        for k = max(1,i) : min(i+poolDim,width)
            for l = max(1,j) : min(j+poolDim,height)
                if(x(k,l) > maxVal)
                    maxVal = x(k,l);
                    maxK = k;
                    maxL = l;
                end
            end
        end
        xMax(i,j) = maxVal;
        mask(i,j) = maxK + (maxL-1)*width;
    end
end
end
```

Listing 12: funnyTan.m

```
function f = funnyTan(X)
f = 1.7159 * tanh(2/3 * X);
end
```

Listing 13: funnyTanGradient.m

```
function g = funnyTanGradient(X)
g = 1.7159*2/3 * sech(2/3.*X).^2;
end
```

Listing 14: gradient_check.m

```
function gradient_check(thetaSize, checkFrac, costGradientFunc,
    checkCount)
%inputs:
% thetaSize - number of elements in theta
% checkFrac - fraction of elements in theta to check each time
% costGradientFunc - function to calculate cost (with input theta)
%                    and gradient (with input theta)
% checkCount - number of checks to do

%checks checkCount random theta value gradients
for i = 1 : checkCount
    %get random theta
    theta = rand(thetaSize,1)*0.001;

    [~, g] = costGradientFunc(theta);

    %check gradient for this theta
    single_gradient_check(theta, g, costGradientFunc, checkFrac);
end
end

function single_gradient_check(theta, g, costGradientFunc,
    checkFrac)
EPSILON = 0.0001;
n = size(theta, 1);
thetaIndices = 1:n;%randperm(n, uint16(checkFrac*n));
```

```
k = 1;
for i = thetaIndices
    theta_i_plus = theta;
    theta_i_minus = theta;
    theta_i_plus(i) = theta_i_plus(i) + EPSILON;
    theta_i_minus(i) = theta_i_minus(i) - EPSILON;
    [cost_plus, ~] = costGradientFunc(theta_i_plus);
    [cost_minus, ~] = costGradientFunc(theta_i_minus);
    g_i_approximation = (cost_plus - cost_minus) / (2 * EPSILON);
    delta = abs(g(i) - g_i_approximation);
    fprintf('%d\n',k);
    if(g(i) == 0)
        assert(delta < 1e-10);
    else
        delta_ratio = delta / g(i);
        %assert(delta_ratio < EPSILON * 10);
        if(delta_ratio >= EPSILON * 10)
            fprintf('%e %e %0.2e  %0.2e\n', g(i), g_i_approximation, EPSILON, delta);
        end
    end
    if k<10
        %print first 10 approximations
        fprintf('%e %e %0.2e  %0.2e\n', g(i), g_i_approximation, EPSILON, delta);
    end
    k = k + 1;
end
end
```

Listing 15: hidden_backprop.m

```
function [model, delta] = hidden_backprop(labels, model, netOut, numClasses, lambda)
numHidden = numel(model.dense.layers) - 1;
m = size(netOut,2);

%stack of gradients (initialize to zeros)
for i = 1:numel(model.dense.layers)
    model.dense.layers{i}.dW = zeros(size(model.dense.layers{i}.W));
    model.dense.layers{i}.db = zeros(size(model.dense.layers{i}.b));
end

delta = cell(m,1);
%backprop for every data point
for i_data = 1:m
    %label indicator vector
    labelIndicator = zeros(numClasses,1);
    labelIndicator(labels(i_data)) = 1;
    %compute delta for output layer
    delta{i_data}{numHidden+2} = -(labelIndicator - netOut(:, i_data)); %HERE
    %backprop delta
    for i=(numHidden+1):-1:1
        %activation
        a = model.dense.layers{i}.act{i_data};
        %gradient of activation
        g = model.dense.layers{i}.actGrad{i_data};
```

```
        %compute delta for hidden layer
        delta{i_data}{i} = (model.dense.layers{i}.W' *delta{i_data
            }{i+1}).*g;
        %update gradient stack
        model.dense.layers{i}.dW = model.dense.layers{i}.dW +
            delta{i_data}{i+1}*a'/m;
        model.dense.layers{i}.db = model.dense.layers{i}.db +
            delta{i_data}{i+1}/m;
    end
end

%regularization
for i = 1:numel(model.dense.layers)
    model.dense.layers{i}.dW = model.dense.layers{i}.dW + model.
        dense.layers{i}.W*lambda;
end
end
```

Listing 16: hidden_forward_pass.m

```
function [ model, netOut ] = hidden_forward_pass( data , model,
    act_fun, grad_fun )
%dense forward pass
numHidden = numel(model.dense.layers) - 1;
netOut = zeros(numel(model.dense.layers{end}.b),size(data,2));
for i_data = 1:size(data,2)
    %first layer
    model.dense.layers{1}.act{i_data} = data(:,i_data);
    model.dense.layers{1}.actGrad{i_data} = 1;
    %other layers
    for i=1:(numHidden+1)
        z = model.dense.layers{i}.W * model.dense.layers{i}.act{
            i_data} + model.dense.layers{i}.b;
        if(i < numHidden+1)
            model.dense.layers{i+1}.act{i_data} = act_fun(z);
            model.dense.layers{i+1}.actGrad{i_data} = grad_fun(z);
        end
    end
    netOut(:,i_data) = exp(z)/sum(exp(z));
end
end
```

Listing 17: invFunnyTanGradient.m

```
function g = invFunnyTanGradient(act)
     g = (1.7159*2/3) *(1-(act./(1.7159)).^2);
end
```

Listing 18: invLogisticSigmoidGradient.m

```
function g = invLogisticSigmoidGradient(act)
    g = act.*(1-act);
end
```

Listing 19: invReluGradient.m

```
function g = invReluGradient(act)
    g =  double(act>0);
end
```

Listing 20: logisticSigmoid.m

```
function f = logisticSigmoid(X)
f = 1 ./ (1 + exp(-X));
end
```

Listing 21: logisticSigmoidGradient.m

```
function g = logisticSigmoidGradient(X)
g = logisticSigmoid(X) .* (1 - logisticSigmoid(X));
end
```

Listing 22: makeGradModel.m

```
function gradModel = makegradModel(model)
%create a model to convert the cell structure to a param vector
for l = 1:numel(model.conv.layers)
    if(strcmp(model.conv.layers{l}.type,'conv'))
        gradModel.conv.layers{l}.outCount = model.conv.layers{l}.outCount;
        for i = 1:numel(model.conv.layers{l}.W)
            for j = 1:numel(model.conv.layers{l}.W{i})
                gradModel.conv.layers{l}.W{i}{j} = model.conv.layers{l}.dW{i}{j};
            end
        end
        for j = 1 : numel(model.conv.layers{l}.b)
            gradModel.conv.layers{l}.b{j} = model.conv.layers{l}.db{j};
        end
        gradModel.conv.layers{l}.type = 'conv';
    else
        gradModel.conv.layers{l}.type = 'unused';
    end
end

for l = 1 : numel(model.dense.layers)
    gradModel.dense.layers{l}.W = model.dense.layers{l}.dW;
    gradModel.dense.layers{l}.b = model.dense.layers{l}.db;
end
end
```

Listing 23: minFuncSGD.m

```
function [opttheta] = minFuncSGD(funObj,theta,data,labels,options)
epochs = options.epochs; %epochs
numIterations = options.numIterations; %number of iterations
alpha = options.alpha; %learning rate
minibatch = options.minibatch; %minibatch size
m = length(labels); % training set size
mom = options.momentum; %momentum
velocity = zeros(size(theta));

all_cost =[];
for e = 1 : epochs
    for i = 1 : numIterations
        ind = randperm(m, minibatch);
        mb_data = data(:,:,ind);
        mb_labels = labels(ind);
        [cost, grad] = funObj(theta,mb_data,mb_labels);
        velocity = (1-mom) * alpha * grad + mom * velocity;
```

```
        theta = theta - velocity;

        all_cost = [all_cost, cost]; plot(all_cost);
        drawnow update
        disp(cost)
    end
    velocity = velocity * 0;
    alpha = alpha / 2;
end

%save('costs_info', all_cost, theta);

opttheta = theta;
end
```

Listing 24: relu.m

```
function f = relu(X)
f = max(0, X);
end
```

Listing 25: reluGradient.m

```
function g = reluGradient(X)
g = double(relu(X) > 0);
end
```

Listing 26: vectorize_conv_out.m

```
function [convOut] = vectorize_conv_out(model)
    %concatenate all end layer feature maps into vector
    convOut = [];
    for j = 1 : numel(model.conv.layers{end}.act)
        actSize = size(model.conv.layers{end}.act{j});
        if( length(actSize)<3 )
            actSize(3) = 1;
        end
        convOut = [convOut; reshape(model.conv.layers{end}.act{j},
            actSize(1) * actSize(2), actSize(3))];
    end
end
```

Listing 27: vectorize_delta_out.m

```
function deltaOut = vectorize_delta_out(delta)
%get deltaOut from delta
deltaOut = [];
for i = 1 : numel(delta)
    deltaOut = [deltaOut delta{i}{1}];
end
end
```