mdCNN – Multi dimensional CNN

**Table of Contents**

[1. General 2](#_Toc532924283)

[2. dataset format – for classification 2](#_Toc532924284)

[3. Regression 2](#_Toc532924285)

[4. Network configuration 3](#_Toc532924286)

[A. Layer specification 3](#_Toc532924287)

[I. Input layer 4](#_Toc532924288)

[II. Fully connected layer 4](#_Toc532924289)

[III. Softmax layer 4](#_Toc532924290)

[IV. Convolutional layer 5](#_Toc532924291)

[V. batchnorm layer 5](#_Toc532924292)

[I. reshape layer 6](#_Toc532924293)

[II. output layer 6](#_Toc532924294)

[B. Hyper params 8](#_Toc532924295)

[C. Run params 8](#_Toc532924296)

[5. Training a network 9](#_Toc532924297)

[a) Using ‘Train’ function 9](#_Toc532924298)

[b) Train a network using backPropagate/feedForward 10](#_Toc532924299)

[6. Demo 12](#_Toc532924300)

# General

mdCNN network can handle input data with 1 2 or 3 dimensions. Every input can have several feature maps.

This document describes in short how to configure and train a network for classification. For regression see section #3

# dataset format – for classification

Input to ‘Train’ is a struct with 4 elements. I,labels,I\_test,in the following format:

* Training data is stored in array called ‘I’ , the labels are stored in a vector named ‘labels’. I and labels are at the same length.
* Testing data is stored in ‘I\_test’ array and ‘labels\_test’ vector.

Training process:

The network state is saved after each iteration loop. The net is saved to a file called ‘net.mat’. This file contains the full state of the network, including the Training state, so you can halt the training process anytime and then load the network from file using load(‘net.mat’)

It’s possible to load the network file, and then continue the training process by calling the ‘Train’ function

# Regression

In order to do regression (de-noising / auto encoders) ‘Train’ function cannot be used.

In this case training process is as follows:

1. Initialize a network
2. Build a batch of input data
3. Create expected output for the batch
4. Call 'backPropagate’ on the data and expected output
5. Repeat #2

Example can be seen under Demo folder ‘AutoEnc/demoAutoEnc.m’

More instruction in section 5, ‘Train a network using backPropagate/feedForward’

# Network configuration

To configure a network you must create a config file. Config file describes the network structure, training parameters and all other possible configuration.

Example config file :

%%%%%%%%%%%%%%%%%%%%% Layers specification %%%%%%%%%%%%%%%%%%

net.layers{end+1}.properties = struct('type','input', 'sizeFm' ,[28 28],'numFm',1 ); %inputLayer

net.layers{end+1}.properties = struct('type','conv','numFm',12 ,'kernel',5,'pad',2);

net.layers{end+1}.properties = struct('type','conv','numFm',24 ,'kernel',13);

net.layers{end+1}.properties = struct('type','batchNorm');

net.layers{end+1}.properties = struct('type','fc','numFm',128, 'dropOut' ,0.8);

net.layers{end+1}.properties = struct('type','fc','numFm',10);

net.layers{end+1}.properties = struct('type','softmax');

net.layers{end+1}.properties = struct('type','output','lossFunc',@CrossEnt,'costFunc',@CrossEnt\_Cost); %output Layer - classification

%%%%%%%%%%%%%%%%%%%%% Hyper params - training %%%%%%%%%%%%%%%%%%

net.hyperParam.trainLoopCount = 2048; % on how many images to train before evaluating the network

net.hyperParam.testImageNum = 1024; % after each loop, on how many images to evaluate network performance

net.hyperParam.ni\_initial = 0.1; % ni (step size) to start training process

net.hyperParam.ni\_final = 0.0005; % final ni to stop the training process

net.hyperParam.noImprovementTh= 8; % after how many iterations with no loss improvment to update ni

net.hyperParam.batchNum = 16;

net.hyperParam.normalizeNetworkInput = 1; %will normalize every samples with mean 0, var=1 before passing to net

net.hyperParam.randomizeTrainingSamples = 0; %shuffle train set

net.runInfoParam.verifyBP = 1; % gradient verification

Any configuration parameter that is not given a value in the config file is assigned with default value.

All possible configuration settings and default values can be found in ‘CreateNet.m’ in ‘initNetDefaults’ function and also listed below

**There are 3 groups of network configuration**

* Layer specification
* Hyper params
* Run params

## Layer specification

Describes the network structure and input/output size. Example:

net.layers{end+1}.properties = struct('type','input', 'sizeFm' ,[28 28 28],'numFm',1 ); %inputLayer

net.layers{end+1}.properties = struct('type','conv','numFm',7 , 'Activation',@Relu, 'dActivation',@dRelu,'kernel',5,'pad',2, 'stride', [2 2 4], 'pooling', [1 1 1]);

net.layers{end+1}.properties = struct('type','conv','numFm',17 , 'kernel',[5 5 3] ,'pad',[1 1 0], 'pooling', [1 1 1], 'dropOut' ,0.8);

net.layers{end+1}.properties = struct('type','batchNorm','initGamma',1,'initBeta',0,alpha',2^-5);

net.layers{end+1}.properties = struct('type','fc','numFm',128);

net.layers{end+1}.properties = struct('type','fc','numFm',10);

net.layers{end+1}.properties = struct('type','softmax');

net.layers{end+1}.properties = struct('type','output','lossFunc',@CrossEnt,'costFunc',@CrossEnt\_Cost); %classification Layer

Every network layer is a struct with the below fields (when not specified default is used) where ‘type’ specifies the layer type:

### Input layer

The input layer must be the first layer and specifies the size of a single sample from the dataset.

For example in case of RGB image:

net.layers{end+1}.properties = struct('type','input', 'sizeFm' ,[32 32],'numFm',3 );

For gray scale image, using Relu activation (Sigmoid is the default)

net.layers{end+1}.properties = struct('type','input', 'sizeFm' ,[32 32],'numFm',1, 'Activation',@Relu, 'dActivation',@dRelu,'kernel' );

For native 3D data

net.layers{end+1}.properties = struct('type','input', 'sizeFm' ,[28 28 28],'numFm',1 );

### Fully connected layer

‘numFM’ field indicates the number of outputs. Every neuron in the FC layer is connected to all outputs of the previous layer

Example – fully connected layer with 10 outputs:

net.layers{end+1}.properties = struct('type','fc','numFm',10);

### Softmax layer

Softmax layer usually appears after a fully connected layer (but not a must) , it performs the softmax function on the previous layer outputs.

Number of feature map (outputs) is derived from the previous layer

Example:

net.layers{end+1}.properties = struct('type','softmax');

### Convolutional layer

* **‘numFM’ –** number of feature map the layer has.
* **‘padding’/’stride’/’pooling’** – this property can have a scalar value or a vector value. When providing scalar this value will be used for all dimension i.e ‘stride’ ,3

When providing a vector, different value can be used per dimension. i.e ‘stride’ , [2 , 1, 5]

* **‘kernel’ –** size of the convolutional kernel for type 2 layers**.** This property can have a scalar value or a vector value**.**
* **‘dropOut’ –** specify the dropout ratio for the layer , number between 0 and 1 , where 1 means no dropout (default)

Default for padding is 0 for all dimensions (no padding)

Default for stride is 1 for all dimensions (no stride)

Default for pooling is 1 for all dimensions (no pooling)

For pooling/stride there is no requirement that the previous layer out is a multiple of the given value. In case this happens input is expended with zeroes.

Example:

net.layers{end+1}.properties = struct('type','conv','numFm',17 , 'kernel',[5 5 3] ,'pad',[1 1 0], 'pooling', [1 1 1], 'dropOut' ,0.8);

### batchnorm layer

The batch norm layer normalizes a batch of outputs with ‘initGamma’ and ‘initBeta’ as parameters

Example:

net.layers{end+1}.properties = struct('type','batchNorm','initGamma',1,'initBeta',0,alpha',2^-5);

Gamma and beta are adjusted variables. One per input.

alpha is a number smaller than one, used to calculate the running batch mean and variance (alpha filtering)

### reshape layer

When placing a conv layer after a fully connected layer sometimes a reshape is needed between the layers

Example

net.layers{end+1}.properties = struct('type','fc','numFm',7\*7);

net.layers{end+1}.properties = struct('type','reshape', 'sizeFm' ,[7 7],'numFm',1);

net.layers{end+1}.properties = struct('type','conv','numFm',12 ,'kernel',3,'pad',2);

Also before output layer it is sometimes needed to place a reshape layer (for regression)

Example:

net.layers{end+1}.properties = struct('type','reshape', 'sizeFm' ,[28 28],'numFm',1);

net.layers{end+1}.properties = struct('type','output','lossFunc',@MSE,'costFunc',@MSE\_Cost); %output layer - regression

### output layer

The output layer specifies the loss/cost function for doing back propagation.

The two main methods are MSE – mean square error and CrossEnt – Cross entropy

MSE is used mainly for regression, and crossEnt is used for classification.

Function pointers for loss function (derivative of the cost) and cost function need to be provided. It’s possible to use the built-in CrossEnt and MSE functions or specify a user defined functions.

Example for classification - cross entropy

net.layers{end+1}.properties = struct('type','output','lossFunc',@CrossEnt,'costFunc',@CrossEnt\_Cost); %classification Layer

Example for regression - MSE

net.layers{end+1}.properties = struct('type','output','lossFunc',@MSE,'costFunc',@MSE\_Cost); %regression Layer

Some loss/dloss functions exist in the ‘Training’ folder, but you can use any predefined functions.

**Activation field**

For layers an activation function handle can be given. Also the derivative of the activation needs to be given as a function handle

Example using Relu:

net.layers{end+1}.properties = struct('type','conv','numFm',7 , 'Activation',@Relu, 'dActivation',@dRelu,'kernel',5,'pad',2, 'stride', [2 2 4], 'pooling', [1 1 1]);

Example using Tanh:

net.layers{end+1}.properties = struct('type','fc','numFm',10,'Activation',@Tanh, 'dActivation',@dTanh);

When not specified, default activation is Sigmoid for fully connected/convolutional and Unit (no activation) for others

Some activation/dActivation functions exist in the ‘Training’ folder, but you can use any predefined functions.

## Hyper params

Hyper parameters affects mainly the training process

**Below are the valid ones:**

net.hyperParam.trainLoopCount=1000;%on how many samples to train before evaluating the network

net.hyperParam.testImageNum=2000;

net.hyperParam.batchNum = 1; % on how many samples to average weights update

net.hyperParam.ni\_initial = 0.05;% ni to start training process

net.hyperParam.ni\_final = 0.00001;% final ni to stop the training process

net.hyperParam.noImprovementTh=50; % if after noImprovementTh there is no improvement , reduce ni

net.hyperParam.momentum=0;

net.hyperParam.constInitWeight=nan; %Use nan to set initial weight to random. Any other value to fixed

net.hyperParam.lambda=0; %L2 regularization factor, set 0 for none. Above 0.01 not recommended

net.hyperParam.testOnData=0; % to perform testing after each epoc on the data inputs or test inputs

net.hyperParam.addBackround=0; % random background can be added to samples before passing to net in order to improve noise resistance.

net.hyperParam.testOnNull=0;% Training on non data images without any feature to detect

net.properties.skipLastLayerErrorCalc=1; % the input layer does not need errors hence calculation can be skipped

%%%%%%%%%%%%%% Augmentation %%%%%%%%%%%%%%

net.hyperParam.augmentImage=0; % set to 0 for no augmentation

net.hyperParam.augmentParams.noiseVar=0.02;

net.hyperParam.augmentParams.maxAngle=45/3;

net.hyperParam.augmentParams.maxScaleFactor=1.1;

net.hyperParam.augmentParams.minScaleFactor=1/1.5;

net.hyperParam.augmentParams.maxStride=4;

net.hyperParam.augmentParams.maxSigma=2;%for gauss filter smoothing

net.hyperParam.augmentParams.imageComplement=0;% will reverse black/white of the image

net.hyperParam.augmentParams.medianFilt=0; %between 0 and one - if this value is 0.75 it will zero all 75% lower points. 0 will mean no point is changed, 1 will keep the higest point only

%%%%%%%%%%%%%% Centralize image before passing to net? %%%%%%%%%%%%%%

net.hyperParam.centralizeImage=0;

net.hyperParam.cropImage=0;

net.hyperParam.flipImage=0; % fill randomly flip the input hor/vert before passing to the network. Improves learning in some instances

net.hyperParam.useRandomPatch=0;

net.hyperParam.testNumPatches=1; % on how many patches from a single image to perform testing. network is evaluated on several patches and result is averaged over all patches.

net.hyperParam.selevtivePatchVarTh=0; %in order to drop patches that their variance is less then th

net.hyperParam.testOnMiddlePatchOnly=0; %will test on the middle patch only

net.hyperParam.normalizeNetworkInput=0; %will normalize every input to net to be with var=1, mean 0

net.hyperParam.randomizeTrainingSamples=1; % randomize the samples selected from dataset during training

## Run params

%%%%%%%%%%%%%% Run info - parameters that change every iteration %%%%%%%%%%%%%%

net.runInfoParam.storeMinLossNet= 0; % this enables the trainer to store also the net with the lowest loss and max success rate found (in addition to the latest one)

net.runInfoParam.verifyBP = 1; % can perform pre-train back-propagation verification. Useful to detect faults in the application

# Training a network

There are 2 options for training a network:

- Using the ‘Train’ function

- Using the backPropagate/feedForward functions

‘Train’ function has some built in tools for data augmentation, step size updating, network saving after each iteration and performance plotting. ‘Train’ can be used only for classification. For regression skip to the next section

## Using ‘Train’ function

**For classification:**

1)create a network using a config file

net = CreateNet('../../Configs/mnist.conf');

2) create a dataset as explained in section #2 in this doc

3) call ‘Train’ function with the dataset containing the train/test samples:

net = Train(dataset,net, 15000);

This will train for 15000 images from the test set in a cyclic manner.

In order to train longer , you can specify ‘Inf’ as the last parameter, network will train until learning rate (ni) reach below the given threshold

During Train iterations the ‘net’ variable is saved to disk. Weights and info is saved in the ‘net’ variable.

You can call Train again to continue training from the last point.

In order to classify samples on the trained network you need to call ‘feedForward’

Please follow ‘checkNetwork’ for details on how to classify new samples or read below how use ‘feedForward’

## Train a network using backPropagate/feedForward

In order to train a network without using ‘Train’ function, one need to provide inputs to the net, and desired outputs.

**Input size:** The dimension of a single input is the dimensions of the first layer, plus the batch size.

For example, network with 20x20 RGB images, and batchNum=16 the size of the input will be

20x20x1x3x16 - where 3 is the number of feature maps, and 20x20x1 is the size of a single feature map

MNIST will be 28x28x1x1x16

network with using 3d CT images of size 32x32x10, batchNum=16, size of the input will be

32x32x10x1x16, where ‘1’ is the number of feature maps

**Output size:** desired output needs to be in the size of the output layer, plus the batch size

Example – classification using 10 labels, last layer before output is fully connected, batchNum =32.

Output size is 1x1x1x10x32

Classification – code outline:

**train on samples:**

% create a network

net = CreateNet('../../Configs/network.conf');

% create a batch of samples

Batch = zeros([net.layers{1}.properties.sizeOut net.hyperParam.batchNum]);

% assuming batch size is 2

Batch(:,:,:,:,1) = sample1;

Batch(:,:,:,:,2) = sample2;

% create a desired network output

expectedOut = zeros([net.layers{end}.properties.sizeOut net.hyperParam.batchNum]);

expectedOut (:,:,:,:,1) = [ 0 0 1 0 0 0 0 0 0 0]; % classified as label 3

expectedOut (:,:,:,:,2) = [ 0 0 0 0 0 0 0 0 0 1]; % classified as label 10

% compute gradients on the batch

net = backPropagate(net, Batch, expectedOut);

% update weights by gradients

net = updateWeights(net, net.runInfoParam.iterInfo(end).ni, net.hyperParam.momentum , net.hyperParam.lambda);

* repeat the weight update process with different samples.

**classify a single sample:**

% create a batch of a single sample

Batch = zeros([net.layers{1}.properties.sizeOut 1]);

Batch(:,:,:,:,1) = sample\_to\_test;

%classify the batch with the sample

net = feedForward(net, Batch , 1);

% select the highest probability from network activations in last layer

[~,netClassification] = max(squeeze(net.layers{end}.outs.activation));

fprintf('sample classified as %d\n', netClassification);

Regression – code outline:

**Regression - train on samples:**

% create a network

net = CreateNet('../../Configs/network.conf');

% create a batch of samples

Batch = zeros([net.layers{1}.properties.sizeOut net.hyperParam.batchNum]);

% assuming batch size is 2

Batch(:,:,:,:,1) = sample1; % for example an image with noise

Batch(:,:,:,:,2) = sample2;

% create a desired network output

expectedOut = zeros([net.layers{end}.properties.sizeOut net.hyperParam.batchNum]);

expectedOut (:,:,:,:,1) = expectedOut1; % for example the image without noise

expectedOut (:,:,:,:,2) = expectedOut2;

% compute gradients on the batch

net = backPropagate(net, Batch, expectedOut);

% update weights by gradients

net = updateWeights(net, net.runInfoParam.iterInfo(end).ni, net.hyperParam.momentum , net.hyperParam.lambda);

%repeat the weight update process with different samples.

**Regression - classify a single sample:**

% create a batch of a single sample

Batch = zeros([net.layers{1}.properties.sizeOut 1]);

Batch(:,:,:,:,1) = sample\_to\_test;

%classify the batch with the sample

net = feedForward(net, Batch , 1);

output = net.layers{end}.outs.activation;

# Demo

There are several examples for networks pre-configured to run :

* MNIST
* CIFAR10
* 3dMNIST - a special enhancement of MNIST dataset to 3D volumes.
* autoencoder demo for regression exist under the demo folder
* 1d Demo

For example, to run MNIST demo: Go into the folder 'Demo/MNIST’, Run 'demoMnist.m' file. The file will download MNIST dataset and start training the network.

After 15 iterations (several minutes) it will open a GUI where you can test the network performance. In addition layer 1 filters will be shown.

Other demos are run the same, set folder to the demo directory and start ‘demo\_\_\_.m’