Using Deep Learning techniques for 3D Object Recognition

Abhineet Kumar Singh
Student ID: 1395723

1 Background

1.1 Problem

This project deals with the problem of automatic object recognition in images including both handwritten characters in documents and real world 3D objects. The former is represented by the MNIST dataset[1] and the latter by the NORB dataset[2]. I have considered only the supervised learning case where the system is first trained using a labeled set of images ('training set') and the trained system is then tested on a new and hopefully unrelated set of images ('testing set'). The scope of this project is limited to investigating the performance of deep convolutional neural networks (CNN) for this task using several network architectures to observe how the training time and test error change by adding additional layers to the network or by changing the number and receptive field sizes of units in existing layers. My objective in this project is thus to implement and test a robust CNN system that makes it convenient for the user to design a wide range of network architectures and quickly evaluate their performance on different datasets.

1.2 Literature survey

Object recognition in images is a well researched area and a large body of literature exists on this subject. Both conventional and deep learning techniques have been used extensively for this task. Some of the popular methods in the former category include support vector machines (SVM)[3][1], k-nearest neighbors(K-NN)[4][5] and boosting[6]. This category also includes shallow neural networks like fully connected networks with one or two hidden layers[1] and small CNNs[7][1].

Deep learning methods include those that use deep belief networks (DBN) [9][10], restricted Boltzmann machines (RBM) [13, 12], recursive (or recurrent) neural networks (RNN) [15][14], deep CNNs[1, 2, 11, 17, 18] as well as hybrid architectures [8]. Several of these methods have been compared in [16] and [1]. Both [9] and [10] have demonstrated that the performance DBNs on object recognition tasks, while much better than shallow methods like SVMs, still fall short of CNNs. The work in [18] has also shown that CNNs remain effective on extremely large datasets with many classes like the *ImageNet* used in this work though they do take a speed and performance hit.

The CNNs used in [1] and [2] are fairly similar except that the former also has a fully connected layer before the output layer that the latter is missing. Also, [1] has utilized the MNIST dataset for testing while [2] has used the NORB dataset. Finally, while [1] has compared the performance of the CNN with a vast array of methods including linear classifiers, nearest neighbour classifiers, SVM and non linear classifiers like PCA along with several types of neural networks like RBF network,one and two layered fully connected networks and shallow CNNs, [2] has only used SVM and K-NN in addition to linear classifiers. Unlike most other classifiers in literature that operate on features extracted from the images, the systems in both these works have used the raw pixel values instead. The CNN presented in [7] is an early version of these systems that has too few layers to be considered deep.

2 Methods

2.1 Approach

As already stated in 1.1 I have used deep CNNs for classifying images in two datasets- MNIST and NORB. The basic architecture I have used is similar to *LeNet-5* that was presented in [1].

2.1.1 Theory

The defining feature of a CNN is the weight sharing that exists between several units in the same layer with their receptive fields located at different, possibly overlapping, locations in the image. This enables them to extract specific features like edges and corners at different locations in the image. A typical CNN consists of an alternating arrangement convolutional and sub sampling layers with one or more fully connected layers at the end. Weight sharing, however, is limited to the convolutional and sub sampling layers each of which is divided into several groups (or 'planes') of units with identical weights. These groups are called feature maps and each such map is applied to many overlapping areas in the input thereby extracting the same feature at multiple locations. Also, since there are several feature maps in each layer, multiple features are extracted at each location too. In fact, CNNs owe their nomenclature to the similarity that exists between this process of applying a fixed set of weights to each location in the image and the process of convolving an image with a fixed kernel.

The sub sampling layers that usually follow each convolutional layer are typically made up of units that perform local averaging (though they may also be max-pooling units) that reduces the resolution of each feature map in the preceding layer. This causes the exact location of each feature to become less important thus providing additional robustness to shifts and distortions that are common in natural images.

This architecture was first introduced in[19] and is inspired by the hierarchical processing that has been found in the mammalian visual cortex[20] where increasingly abstract representations of the scene are created by subsequent layers of cells. Similarly, the features extracted by a convolutional layer are combined by subsequent layers to extract more high-level features. Also, the weight sharing feature of convolutional units is meant to emulate the overlapping receptive fields of neighboring retinal cells. This, it is hoped, provides a degree of invariance to local object transformations like translations and pose variations.

2.1.2 Architecture

The architecture described here is the one presented in [1] which is the one that I have used as the base configuration for my tests though I had to make several simplifications that are presented in 1.1.

The CNN presented in [1] has 6 layers excluding the input and output layers. The first one, called C1, is a convolutional layer with 6 feature maps of size 28x28 each since each unit is connected to a 5x5 neighborhood of the 32x32 input image. Each unit calculates a weighted sum of its 25 inputs, adds a bias to the sum and passes it through a non linear activation function (hyperbolic tangent). With 25 weights and 1 bias, each unit (and thus each feature map) has 26 trainable parameters and with 6 feature maps, C1 has a total of $26 * 6 = 156$ parameters.

The second layer, S2, is a sub sampling layer with 6 feature maps where each unit in each map receives input from a 2x2 non overlapping region in the corresponding feature C1 map thus leading to a map size of 14x14. Each unit sums its inputs, multiplies the sum by a weight, adds a bias and finally passes it through the activation function. Since each feature map has 2 parameters, S2 has a total of 12 parameters.

The third layer is called C3 and is identical to C1 except that it has 16 feature maps and each map is connected to a subset of the 6 output maps of S2 (refer [1] for details). Each C3 unit calculates a weighted sum of 5x5 regions in each S2 map that it is connected to, adds them all up, adds a bias to the sum and passes it through the activation function. Each feature map thus has a size of 10x10 units. The fourth layer, s4, is likewise identical to S2 except it has 16 feature maps of size 5x5 with each unit again receiving input from a 2x2 non overlapping region of the corresponding C3 feature map.

The fifth layer is called C5 and is similar to C3 but with 120 feature maps each connected to *all* of the 16 S4 maps. Since each unit has a receptive field of 5x5 which is equal to the map size of S4, each C5 map has only one unit. The last hidden layer, called F6, has 84 units, each connected to all the 120 units (one per map) of C5.

The output of all 84 F6 units go to each of the n (if we are considering n classes) output units that calculate the sum of squared differences (or Euclidean distances) between their inputs and their weights. These 'weights' are actually arranged as a 7x12 binary image (hence the size of F6) each featuring a prototype of the character which that particular output unit represents. The use of this distributed output representation instead of the usual 1-of-N representation is stated in [1] as a way to avoid ambiguity between similar looking characters and also to perform better with large number of classes.

2.1.3 Implementation

I have used the CNN component of the Matlab deep learning toolbox [21] as a starting point for my implementation of a user customizable CNN based classifier. This toolbox, however, implements a very generic version of CNN and the code had to be modified heavily to make it suitable for my objectives. The code for the CNN component borrows heavily from an online tutorial [22] and I too had to refer to it to understand and modify the code. Following are the main additions I have made:

1. Support for fully connected layers since the toolbox offers only convolutional and sub sampling layer types.
2. Support for specifying different activation function (identity, sigmoid or tanh) for each layer since the toolbox uses the same fixed (sigmoid) function for each layer.
3. A simple input interface to let the user conveniently specify the network structure which is hard coded in the toolbox.
4. Support for partial connections between convolutional and sub sampling layers as has been used between C3 and S2 in [1].
5. The ability to read raw data from both MNIST and NORB (small) datasets and apply various pre-processing operations including scaling, padding and normalization.

As compared to LeNet-5 of [1], however, there are still a few shortcomings in the final system that could not be addressed due to time constraints:

1. The system still supports only the 1-of-N output representation and not the distributed version used in LeNet-5.
2. The system supports only the simple Euclidean distance loss function rather than the maximum a posteriori criterion stated in [1].
3. The system uses the same learning rate for all parameters in all the layers which also remains fixed during the training process. It does not support the stochastic diagonal Levenberg-Marquardt algorithm used in [1] for dynamically adjusting the rate.

More details about the code are present in the appendix.

2.1.4 Datasets

I have used two datasets for running the experiments:

1. **MNIST:** This is a dataset of handwritten characters[1] that contains images of digits (0-9) each 28x28 in size. There are 60000 training images (6000 per digit) and 10000 test images.

2. **NORB:** This is a dataset of stereo image pairs of 50 toys belonging to 5 generic classes including airplanes, trucks, cars, four legged animals and human figures [2]. The full database has almost 200,000 images divided into 10 parts but using even a single part with around 29000 images caused my system to hangup so I am using the small NORB dataset instead [23]. This contains 24300 pairs of training and an equal number of testing images, each 96x96 in dimensions. I have only considered the first image in each pair and have further divided these images into 3 groups of 8100 each. Out of these I have used the first and second parts for the experiments thus giving 8100 and 16200 training images . Since my system could not handle the entire test set, I have used a train to test size ratio of 5:1 (1620 and 3240 test images respectively). For most tests, I also scaled these images by a factor of 3 to get input images of size 32x32.

For both of these datasets I have used raw pixel values normalized to a range of 0-1 as inputs for the CNN.

2.2 Rationale

The main reason I am interested in exploring CNNs is because they attempt to emulate the functioning of the mammalian (and by extension human) visual cortex. I believe this is the best approach to vision problems since the vastly superior human visual system is the benchmark against which most of these systems are measured and whose performance they attempt to approximate.

As enumerated in 2.1.3, I had to make several simplifications to the *LeNet-5* architecture to make the project feasible. Most of these were made due to constraints of time and computational power. Also, it should be noted that the architecture stated in 2.1.2 is *not* the only one I have tested the system with.

3 Plan

3.1 Hypothesis

Since this is a purely experimental project, I don't really have any specific theory that I am trying to test. My sole objective is to manipulate different parameters of the CNN and observe the resultant performance to evaluate the effect of each of those parameters.

3.2 Experimental Design

I have experimented with the following parameters:

1. **Network structure:** Before stating the different architectures I have considered, let us define a compact representation for stating a particular architecture. Here, an architecture is denoted by a sequence of 2 or 3 character descriptions of different layers separated by dashes. A convolutional layer is represented by 3 characters as <no. of output maps>c<kernel size> while sub sampling and fully connected layers are represented by 2 characters as <scale>s and <no. of units>f respectively. Thus a string '6c5-2s-4f' represents a CNN with 3 hidden layers: a convolutional layer with 6 output maps having kernel size 5x5, a sub sampling layer with kernel size 2x2 and a fully connected layer with 4 units. Following are the two main architectures I have considered:

   (a) 6c5-2s-16c5-2s-120c-10f : This is a simplified version of LeNet-5 architecture where the main difference is that the fully connected layer has 10 units instead of 84 since I am using a 1-of-N output representation rather than the distributed one mentioned in [1]

   (b) 6c5-2s-12c5-2s : This is the basic architecture presented in [7] and was also the default architecture that comes hard coded with the deep learning toolbox.

   I have used several other architectures derived from these two either by changing the kernel size, the scale or the output map size or by adding, removing or replacing one or more rows in these.The details of these architectures can be found in the results section.

Note that these descriptions do not include the output layer which in all the experiments was a simple n unit layer that treats the output of the last layer as a vector and passes the weighted and biased sum through the activation function. Also note that most of these architectures were only tested with the NORB set since the small size of the MNIST images does not permit much variation in the architecture due to the various input-output constraints that must be satisfied by consecutive layers.

2. **Activation Functions:** As stated earlier, I have designed the system to be capable of using a different activation function for each layer type in cluding the output layer. In the list that follows, 0, 1 and 2 stand for identity, sigmoid and tanh functions respectively and are listed for convolutional, sub sampling, fully connected and output layer in order from left to right.
   (a) 1-0-1-1 : This is the default configuration used in the toolbox.
   (b) 2-0-2-2 : This is similar to the last one except that the sigmoid function has been replaced by the tanh function following [1]
   (c) 1-1-1-1 : This is similar to the first one with sub sampling layer getting a sigmoid function instead of simple summation.
   (d) 2-2-2-2 : This is similar to the second one with sub sampling layer getting a tanh function instead of simple summation.
3. **Learning rate:** I have tested several learning rates varying from 0.01 to 2.
4. **Training batch size**: This is the number of training samples that were used for each weight update. I have tested values from 2 to 50.
5. **Number of epochs**: This is the number of times the entire training set is passed through the system. I have varied this from 1 to 25.
6. **Input size**: This has only been tested for the NORB dataset for which I have used both the original 96x96 images as well as scaled down 32x32 images.
7. **Training set size**: This too has only been tested with the NORB set with training sizes of 16200 and 8100.
8. **Updating output weights**: It is mentioned in [1] that updating the output weights and bias during training can cause all the parameters to converge to constant values thus leading to a fixed out invariant to further training. To check this effect with different activation functions I have conducted experiments with both updates enabled and disabled.

4 Results and Analysis

4.1 MNIST Dataset

In all of the following experiments, unless otherwise stated, the parameter values used are as stated in Table 1.

Table 1: Default parameters for MNIST

Architecture	**Activation**	**Train size**	**Test size**	**Image Size**	**Epochs**	**Batch Size**	**Learning rate**
6c5-2s-12c5-2s	1-0-1-1	60000	10000	28x28	1	50	1.00

Architecture In addition to the default architecture, I have tested 3 others: one to change the number of output maps (6c5-2s-8c5-2s), one to change the kernel size (6c3-2s-12c4-2s) and one to change the scale (6c5-3s-12c3-3s). The other architecture mentioned in Sec. 3.2 could not be used here due to size constraints.

The results are presented in Table 2. As we can see, best results are achieved with decreased kernel size, though the training time there is much higher than the default case and the error improvement is not really much. Also, changing the scale has a drastic effect and causes all the network outputs to converge to a fixed value.

Table 2: Network architectures with MNIST

Architecture	**Test Error**	**Training time**
6c5-2s-12c5-2s	5.37%	112.000916
6c5-2s-8c5-2s	7.30%	98.709396
6c3-2s-12c4-2s	5.18%	184.150255
6c5-3s-12c3-3s	89.90%	91.103302

Epochs I experimented with 1, 5, 10 and 25 epochs and their results are in Table 3. We can see that the error consistently falls as more epochs are run. The per-epoch training time shows some significant, but mostly random variations.

Table 3: Epochs with MNIST

Epochs	**Test Error**	**Epoch Training time (average)**
1	5.37%	112.000916
5	2.07%	52.188843
10	1.67%	220.628684
25	1.30%	126.638912

Activation Functions I experimented with each of the 4 combinations mentioned earlier and all but the default converged to a fixed output, probably due to over fitting, as can be seen in Table 4. These tests were repeated with different epochs and learning rates but the convergence behavior remained consistent thus proving the great dependence of successful training on the activation functions. Not only did these configurations perform badly, they also took significantly longer to train.

Table 4: Activation Functions with MNIST

Activation	**Test Error**	**Training time**
1-0-1-1	5.37%	112.000916
1-1-1-1	89.90%	141.1223
2-0-2-2	90.20%	140.693909
2-2-2-2	90.18%	145.183733

Learning Rate I experimented with learning rates of 0.01, 0.05, 0.1, 0.5 1 and 2 and the results are listed in Table 5. As we can see, the familiar convergence occurs for all but two of the learning rates. The training times do not show any significant change, however. An important difference in the behavior of learning rates as compared to the earlier parameters is that they are very sensitive to both the batch size as well as the number of epochs and adjusting either can dramatically improve the results, as can be seen in Table 6.

Table 5: Learning Rates with MNIST

Learning rate	**Test Error**	**Training time (sec)**
0.01	88.65%	122.286215
0.05	88.65%	175.225081
0.1	88.65%	170.413997
0.5	7.96%	166.919743
1	5.37%	112.000916
2	89.90%	172.974762

Table 6: Learning Rates with MNIST (varying Epochs and Batch Sizes)

Learning rate	**Epochs**	**Batch Size**	**Test Error**	**Epoch Training Time (average)(sec)**
0.01	5	5	6.28%	442.976488
0.05	5	50	12.64%	153.543017
0.1	5	50	2.07%	52.188843
0.1	1	2	8.66%	1362.281223
0.5	5	50	2.38%	226.49197

Batch Size I tested using batch sizes of 2, 5, 10 and 50 and the results are presented in Table 7. The convergence behavior was observed with batch sizes 2 and 5 probably because these batch sizes are too small given the large number of training samples, thus leading to too frequent weight updates and thus over fitting.

Table 7: Batch Sizes with MNIST

Batch Size	**Test Error**	**Training time**
2	90.26%	1291.589919
5	89.72%	637.35549
10	6.95%	320.489555
50	5.37%	112.000916

Output parameters update The results of testing with and without output parameter updates are presented in Table 8 for 1 and 5 epochs. We can see that, contrary to what was mentioned in [1], not only does the performance not not improve on disabling weight updates, but it actually goes down significantly. This is probably because a different distributed output representation was used there.

4.1.1 NORB Dataset

In all of the following experiments, unless otherwise stated, the parameter values used are as stated in Table 9.

Architecture In addition, to both the architectures mentioned in Sec. 3.2, I have used the following architectures:

1. 6c7-3s-12c7-3s : This is a slightly modified version of the default architecture that is more suitable for a larger input image (for instance the 96x96 NORB images)

2. 6c9-4s-12c7-4s : This is a further extension of the last case meant to test the effect of larger averaging kernels.

Table 8: Output Updates on MNIST

Output Update	**Epochs**	**Test Error**	**Epoch Training time (average) (sec)**
Yes	1	5.37%	112.000916
No	1	11.51%	178.259364
Yes	5	2.07%	52.188843
No	5	5.42%	139.077932

Table 9: Default parameters for MNIST

Architecture	**Activation**	**Train size**	**Test size**	**Image Size**	**Epochs**	**Batch Size**	**Learning rate**
6c5-2s-12c5-2s	1-0-1-1	8100	1620	32x32	1	2	0.1

3. 6c5-2s-16c5-2s-120c5 : This is similar to the first one in Sec. 3.2 except without the fully connected layer.

4. 6c7-3s-12c7-3s-24c5-2s : This is a hybrid of the two main architectures with the fully connected layer of (a) replaced by a sub sampling layer.

The results using these architectures are presented in Table 10. The first thing we can notice is that the error rates are *much* higher here than for MNIST. This is to be expected since NORB is a much more complex dataset with far greater intra class variations. Best error rates were observed using higher kernel sizes and scales than the default architecture. It should be noted that these were tested on the original unscaled images since they were not compatible with the scaled down images. Apart from this, we can see that the convergence behavior occurs in all the architectures derived from that of *LeNet-5*. This is a consistent fact that was observed in all my experiments with NORB set. With no combination of parameters could I get anything but constant output.

Table 10: Architectures for NORB

Architecture	**Image Size**	**Test Error**	**Training time**
6c5-2s-12c5-2s	32x32	60.99000%	148.558113
6c-2s-16c-2s -120c-10f	96x96	80.98765%	4912.564771
6c7-3s-12c7-3s	96x96	42.65432%	419.595917
6c9-4s-12c7-4s	96x96	43.70370%	461.913822
6c-2s-16c-2s -120c	96x96	80.00000%	6172.62
6c7-3s-12c7-3s-24c5-2s	96x96	74.13580%	876.778727

Activation Functions I experimented with each of the 4 combinations mentioned earlier and, just as in MNIST, all but the default converged to a fixed output, as can be seen in Table 11. These tests were repeated with different epochs and learning rates but the convergence behavior remained consistent here too. In fact the worst performance was shown by the configurations involving the tanh function that was highly recommended in [1].

Epochs I experimented with 1, 5, 10 and 25 epochs and their results are in Table 12. A dramatic improvement in performance, with less than a third of the error rate, can be observed on going from 1 to 5 epochs. Further increases in epochs, however, fail to produce significant improvements, thus indicating the presence of some other performance bottleneck.

Table 11: Activation Functions for NORB

Activation	**Test Error**	**Training time**
1-0-1-1	60.99000%	148.558113
1-1-1-1	79.63%	346.823651
2-0-2-2	80.98765%	145.734406
2-2-2-2	80.98765%	143.88872

Table 12: Epochs for NORB

Epochs	**Test Error**	**Epoch Training time (average)(sec)**
1	60.99000%	148.558113
5	20.24691%	167.035381
10	17.53086%	170.621528
25	19.44444%	334.171942

Learning Rate I experimented with learning rates of 0.05, 0.1, 0.5 and 1 whose results are listed in Table 13. As we can see, the familiar convergence occurs for all but the default learning rate. The training times also do not show any significant change, just like in MNIST.

Table 13: Learning Rates for NORB

Learning rate	**Test Error**	**Training time (each epoch)**
0.05	82.59259%	198.533885
0.1	60.99000%	108.364934
0.5	80.74074%	213.61472
1	79.25926%	151.407015

Batch Size I tested using batch sizes of 2, 5, 10 and 50 and the results are presented in Table 14. A strong convergence behavior was observed with batch all but size 2 which is completely opposed to the behavior observed for MNIST. One possible explanation is that the training size here is much smaller (almost an eight). Also, it is likely that individual samples here contain much more information than in MNIST, so skipping them for individual updates (so to speak) has greater penalty here. Though the errors for all convergent cases are identical, the training time does show the expected decrease with increase in batch size due to the fewer batches that need to be run.

Training Set Size In addition to 8100 training samples I also ran some experiments with 16200 training samples (and correspondingly 3240 test samples). The results of these experiments are shown in Table 15. If we compare the first 3 entries here with those in Table 12, we find that the same pattern of improvement can be observed here too although the improvement on going from 1 to 5 epochs is less here due to the much improved performance with 1 epoch.

The last entry here is included to highlight the dependency of the learning rate on the training set size. Even though convergence was observed with the standard training set for all but the default learning rate (Table 13), which painted a grim picture of the effect that varying this parameter has on performance, yet here we see a significant improvement by simply bringing in more training samples.

Table 14: Batch Sizes for NORB

Batch Size	**Test Error**	**Training time (each epoch)**
2	60.99%	148.558113
5	80.98765%	108.364934
10	80.98765%	57.679653
50	80.98765%	40.066205

Table 15: Training Size with NORB

Training Samples	**Test Samples**	**Epochs**	**Learning Rate**	**Error**	**Training Time**
16200	3240	1	0.1	45.67901%	309.255207
16200	3240	5	0.1	20.77161%	396.468742
16200	3240	10	0.1	19.10494%	373.511934
16200	3240	1	0.5	59.22840%	397.755664

5 Conclusion

I have successfully implemented and tested a CNN on two different datasets and with a wide range of configurations obtaining many interesting results. One of the most noteworthy features of these results is the convergence to constant output that occurs so often. It may be caused by over fitting or under fitting or some other reasons but the fundamental shortcoming of such architectures that it points out is the over-specificity of each network configuration and the ease with which a great performing architecture can be rendered completely worthless by tiny modifications of even a single parameter.

This in turn indicates the need, for deep CNN in particular and deep architectures in general, of some mechanism to ensure the network's robustness to variations of its parameters. There should at least some way to guarantee that its performance, when it degrades due to faulty parameter adjustment, will do so gradually rather than going from 5% to 80% error in a single step, as I observed so frequently during my experiments.

Appendix

Code Description

The main function in the code is called `runCNN`. It calls the function `getOptionalParameters` to allow the user to enter some optional parameters through a convenient GUI. In addition to these, the user can also provide some optional command line arguments whose details are in `runCNN`. Following are the other functions that are called along with their purpose:

1. `loadMNISTData`/`loadSmallNORBData` :Loads MNIST/NORB data with the parameters specified either through the GUI or the command line.

2. `preProcessData` : Performs pre processing on the loaded data including scaling, normalizing and padding by calling `scaleData`, `normalizeData` and `padData` respectively.

3. `saveData` : Saves pre processed data as a `mat` file that can be loaded later thus avoiding the need to pre process the data again.

4. `readCNNStructure` : Reads the architecture of the CNN from a text file which is called 'layers.txt' by default. However, the user is allowed to select another file at run time if it does not exist. Formatting details of this file can be found in this function. Also, a couple of sample configuration files (`layers.txt` and `layers2.txt`) are included with the code for reference.

5. `writeCNNStructure` : Writes the CNN structure to a a text file called `cnn_structure.txt` allowing the user to examine it and confirm that the correct structure has been read.

6. `cnnTrain` : This is the main training function that calls the following functions in the course of training the CNN:

   (a) `getActivationFunction` : Returns fast inline implementations for the different activation functions and their gradients including sigmoid, tanh and identity functions.

   (b) `cnnFeedForward` : Pass the given batch of input data through the network and compute the outputs of all the units.

   (c) `cnnBackPropagate` : Use the computed outputs in a gradient descent based back propagation algorithm to compute the change in weights required for each of the units.

   (d) `cnnCheckGradients` : An optional function that can be used to check whether the gradients have been implemented correctly. Uses brute force numerical techniques to calculate the exact gradient and compares them with those computed by `cnnBackPropagate`.

   (e) `cnnApplyGradients` : Uses the weight updates calculated by `cnnBackPropagate` to update all the weights in the CNN.

7. `cnnTest/cnnTestInBatch` : Tests the trained network on the test data either all at once or in batch mode. The latter is a bit slower but can be handy when the test size is too large to be loaded/manipulated all at once.

In addition to these essential functions, the following utility functions are included too:

1. `readMNISTData/readSmallNORBData`: These can read the raw data files as available online and convert them into a more usable form, also dividing them into several parts if needed.

2. `writeCNNParams/writeCNNValues`: These can be used to write the parameters (weights and biases) and outputs of a trained network respectively which can be useful for debugging.

3. `showImages` These can be used to show samples images from the given dataset, divided by class to ensure that the datasets have been loaded correctly. These can be enabled through one of the optional arguments to `runCNN`.

Please note that, in order for the code to run successfully, the c**orrect data files must be placed in the NORB or MNIST sub folders**. These are too large to be included with the code so must be generated using the `readMNISTData/readSmallNORBData` functions mentioned above.

References

[1] Y. Lecun , L. Bottou , Y. Bengio, P. Haffner, "Gradient-based learning applied to document recognition", *in Proc. of the IEEE*, 1998.

[2] Y. Lecun, F.J. Huang, L. Bottou, "Learning Methods for Generic Object Recognition with Invariance to Pose and Lighting", *in Proc. CVPR*, 2004.

[3] D. DeCoste and B. Scholkopf, "Training Invariant Support Vector Machines", *Machine Learning*, 2002.

[4] Daniel Keysers, Thomas Deselaers, Christian Gollan and Hermann Ney , "Deformation Models for Image Recognition", *IEEE Transactions On Pattern Analysis And Machine Intelligence*, 2007.

[5] S. Belongie, J. Malik and J. Puzicha, "Shape matching and object recognition using shape contexts", *International Journal of Computer Vision* , 2002.

[6] B. Kegl and R. Busa-Fekete "Boosting products of base classifiers", *in Proc. 26th Annual International Conference on Machine Learning*, 2009.

[7] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel, "Handwritten Digit Recognition with a Back-Propagation Network", *Advances in Neural Information Processing Systems*, 1990.

[8] Richard Socher, Brody Huval, Bharath Bhat, Christopher D. Manning, Andrew Y. Ng, "Convolutional-Recursive Deep Learning for 3D Object Classification", *Advances in Neural Information Processing Systems*, 2012.

[9] Vinod Nair and Georey E. Hinton, "3D Object Recognition with Deep Belief Nets", *Advances in Neural Information Processing Systems* , 2009.

[10] Y. Tang and A. Mohamed "Multiresolution Deep Belief Networks", *Journal of Machine Learning Research*, 2012.

[11] L. Quoc, J. Ngiam, Z. Chen, D. Chia, P. W. Koh, and A. Ng, "Tiled convolutional neural networks", *NIPS 23*, 2010.

[12] R. Salakhutdinov and G. Hinton, "Deep Boltzmann machines", *AISTATS*, 2009.

[13] G. Desjardins and Y. Bengio, "Empirical Evaluation of Convolutional RBMs for Vision", *Technical Report*, 2008.

[14] M. Bianchini, M. Maggini, L. Sarti, F. Scarselli "Recursive neural networks for object detection", *in Proc. IEEE International Joint Conference on Neural Networks*, 2004.

[15] A. Graves, M. Liwicki, S. Fernandez, R. Bertolami, H. Bunke, J. Schmidhuber, "A Novel Connectionist System for Improved Unconstrained Handwriting Recognition", *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2009.

[16] L. Bottou, C. Cortes, J. Denker,H. Drucker,I. Guyon, L. Jackel,Y. LeCun, U. Muller,E. Sackinger,P. Simard and V. Vapnik, "Comparison of classifier methods: a case study in handwritten digit recognition", *in Proc. 12th International. Conference on Pattern Recognition*, 1994.

[17] D. Ciresan, U. Meier, J. Masci, L. Gambardella and J. Schmidhuber, "Flexible, High Performance Convolutional Neural Networks for Image Classication", *in Proc. 22nd international Joint Conference on Artificial Intelligence*, 2011.

[18] A. Krizhevsky, I. Sutskever and G. Hinton, "ImageNet Classication with Deep Convolutional Neural Networks", *Advances in Neural Information Processing Systems*, 2012.

[19] K. Fukushima, "Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position", *Biological Cybernetics*, 1980.

[20] D. Hubel and T.Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex", *Journal of Physiology*, 1962.

[21] R. Palm, "DeepLearnToolbox", *Available online at https://github.com/rasmusbergpalm/DeepLearnToolbox*, 2014.

[22] J. Bouvrie, "Notes on Convolutional Neural Networks", *Available online at http://cogprints.org/5869/*, 2006.

[23] Fu Jie Huang and Yann LeCun "THE small NORB DATASET", *Available online at http://www.cs.nyu.edu/ ylclab/data/norb-v1.0-small/*, 2005.