A Major Project On

Deep Learning Neural Networks for Vision and Classification in Torch7

Submitted in partial fulfillment of the requirements of the degree of

Bachelor of Technology in

Computer Engineering

Submitted by

Aviral Takkar (2K11/CO/031)

Aishwarya Deep (2K11/CO/010)

Amanjeet Singh Bhatia (2K11/CO/020)

Under the esteemed guidance of

Mr. Rajesh Kumar Yadav

Assistant Professor

Computer Engineering Department, DTU, Delhi

2011 – 2015

Delhi Technological University

Delhi – 110042

Certificate

This is to certify that the major project entitled **"Deep Learning Neural Networks for Vision and Classification in Torch7"** is a bona fide record of work done at **Delhi Technological University** by **Aviral Takkar (Roll number 2K11/CO/031), Aishwarya Deep (Roll number 2K11/CO/010) and Amanjeet Singh Bhatia (Roll number 2K11/CO/020)** for partial fulfilment of the requirements of the degree of Bachelor of Technology in Computer Engineering. This project was carried out under my supervision and has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma to the best of my knowledge and belief.

(Mr. Rajesh Kumar Yadav)
Assistant Professor & Project Guide
Department of Computer Engineering
Delhi Technological University

Date: _____

Acknowledgement

It gives us a great sense of pleasure to present the report of our Bachelor of Technology project entitled **"Deep Learning Neural Networks for Vision and Classification in Torch7"** in our final year. We would like to express our deep gratitude to Mr. Rajesh K Yadav, Assistant Professor, Department of Computer Engineering, Delhi Technological University, for his patient guidance, enthusiastic encouragement and useful critiques of this research work. His willingness to give his time so generously has been appreciated.

We humbly extend our words of gratitude to other faculty members and our peers for their invaluable help and support.

Aviral Takkar
2K11/CO/031
Bachelor of Technology in Computer Engineering,
Delhi Technological University
aviraltakkar@gmail.com

Aishwarya Deep
2K11/CO/010
Bachelor of Technology in Computer Engineering,
Delhi Technological University
aishwaryadeep16@gmail.com

Amanjeet Singh Bhatia
2K11/CO/010
Bachelor of Technology in Computer Engineering,
Delhi Technological University
amanjeet.3987@gmail.com

Abstract

We develop deep learning neural networks in Torch7 for two applications. First, to implement the Kaggle Forest Cover Type Prediction Challenge [4], which is essentially a classification problem. Based on 55 cartographic parameters of 15120 training examples, we perform supervised training of a multi-layer neural network to predict the forest cover type. We find that for learning such a large number of parameters, a multi-layer perceptron is not adequate as it is very prone to overfitting, and is not able to generalize well to data, as the number of parameters to learn increases, in the presence of less training data. Thus, the need for Convolutional Neural Nets, to capture hidden relationships and dependencies among the input vector, is clearly seen. Second, based on the paper "*Convolutional Neural Networks Applied to House Numbers Digit Classification*" [8] by LeCun et al, we implement the Convolutional Neural Network described there in Torch7 on the SVHN [5] dataset. We achieve high accuracy in the test results, and make a comparative analysis of changing 'p' values in Lp-pooling versus accuracy and of using Multi Stage Feature Learning versus Single Stage Feature Learning.

Keywords: Convolutional Neural Networks, SVHN, Classification, Vision, Deep learning.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

1.1 Deep Learning

Deep Learning is an upcoming area of Machine Learning Research, with many remarkable recent successes, such as 97.5% accuracy on face recognition, nearly perfect German traffic sign recognition, or even Dogs vs Cats image recognition with 98.9% accuracy (all results on the ImageNet [1] dataset of images). Many winning entries in recent Kaggle Data Science competitions have used Deep Learning.

The term "deep learning" refers to the method of training multi-layered neural networks, and became popular after papers by Geoffrey Hinton and his co-workers from the University of Toronto, Canada, showed a fast way to train such networks using backpropagation. These methods aim at learning feature hierarchies with features from higher levels of the hierarchy formed by the composition of lower level features, as shown in Figure 1.

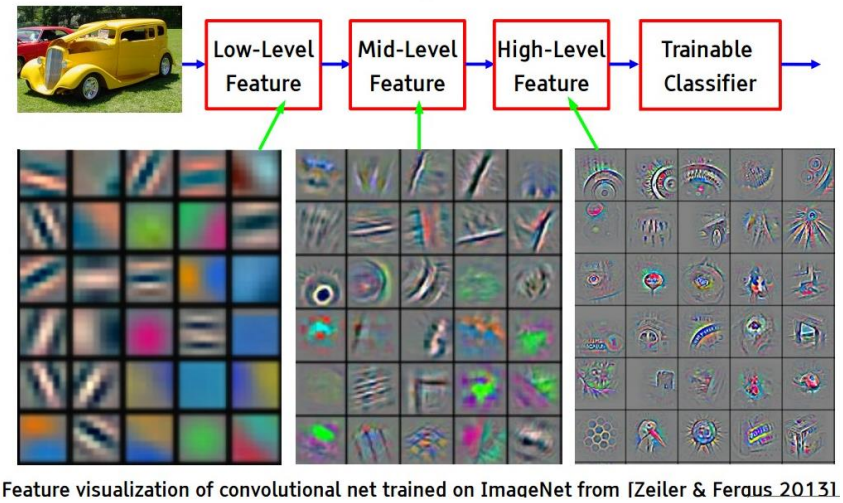


Figure 1. Learning in a deep network.

This automatic learning of features at multiple levels allows a system to learn complex functions mapping the input to the output directly from data, without depending completely on human-crafted features. Given that humans may not know how to specify explicitly certain features in terms of raw data, this is especially important for higher-level abstractions. The ability to automatically learn powerful features will become increasingly important as the amount of data and range of applications to machine learning

methods continues to grow. By depth of architecture, we refer to the number of levels of composition of non-linear operations in the function learned [10]. Whereas most current learning algorithms correspond to 'shallow' architectures (1, 2 or 3 levels), the mammal brain is organized in a deep architecture [11] with a given input percept represented at multiple levels of abstraction, each level corresponding to a different area of cortex. Humans often describe such concepts in hierarchical ways, with multiple levels of abstraction. The brain also appears to process information through multiple stages of transformation and representation. This is particularly clear in the primate visual system [11], with its sequence of processing stages: detection of edges, primitive shapes, and moving up to gradually more complex visual shapes. Inspired by the architectural depth of the brain, neural network researchers had wanted for decades to train deep multi-layer neural networks, but no successful attempts were reported before 2006: researchers reported positive experimental results with typically two or three levels (i.e., one or two hidden layers), but training deeper networks consistently yielded poorer results. Hinton et al. at University of Toronto introduced Deep Belief Networks (DBNs), with a learning algorithm that greedily trains one layer at a time, exploiting an unsupervised learning algorithm for each layer, a Restricted Boltzmann Machine (RBM).

Since 2006, deep networks have been applied with success not only in classification tasks [11, 12, 13, 14, 15, 16], but also in regression [17], dimensionality reduction [18, 19], modeling textures, modeling motion [20, 21, 22], object segmentation, information retrieval, robotics, [23, 24, 25] natural language processing [26], and collaborative filtering.

In May 2014, Baidu, the Chinese search giant, has hired Andrew Ng, a leading Machine Learning and Deep Learning expert (and co-founder of Coursera) to head their new AI Lab in Silicon Valley, setting up an AI & Deep Learning race with Google (which hired Geoff Hinton) and Facebook (which hired Yann LeCun to head Facebook AI Lab). These developments clearly indicate that deep learning algorithms are gaining popularity at a massive scale due to their large learning potential.

Current approaches to object recognition make essential use of machine learning methods. To improve their performance, we can collect larger datasets, learn more powerful models, and use better techniques for preventing overfitting. Simple recognition tasks can be solved quite well with datasets of small size, especially if they are augmented with label-preserving transformations. For example, the current best error rate on the MNIST digit-recognition task (<0.3%) approaches human performance. [27]

But objects in realistic settings exhibit considerable variability, so to learn to recognize them it is necessary to use much larger training sets.

The new larger datasets include LabelMe [2], which consists of hundreds of thousands of fully-segmented images, and ImageNet [1], which consists of over 15 million labeled high-resolution images in over 22,000 categories. To learn about thousands of objects from millions of images, we need a model with a large learning capacity. However, the immense complexity of the object recognition task means that this problem cannot be specified even by a dataset as large as ImageNet, so our model should also have lots of prior knowledge to compensate for all the data we don't have. Convolutional neural networks (CNNs), one such class of models [3], have a capacity which can be controlled by varying their depth and breadth, and they also make strong assumptions about the nature of images(namely, stationarity of statistics and locality of pixel dependencies). Thus, compared to standard feedforward neural networks with similarly-sized layers, CNNs have much fewer connections and parameters and so they are easier to train, while their theoretically-best performance is likely to be only slightly worse.

In our project, we target to train our model to a very restricted set of image recognition task, namely that of character recognition in a real world setting. The SVHN dataset is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images. [5] We use format two of this dataset, namely the character level ground truth format. All digits in this have a fixed resolution of 32-by-32 pixels. The original character bounding boxes are extended in the appropriate dimension to become square windows, so that resizing them to 32-by-32 pixels does not introduce aspect ratio distortions. We obtain the data in the format required by Torch7 (i.e. t7 format) from [28].

Deep Learning Networks attempt to learn multiple features of the image at stage in the network, with the complexity of the features increasing as we propagate through the network, as demonstrated in Figure 1. Despite their attractive capabilities, CNNs are prohibitively expensive to train due to the nature and quantities of input they deal with while performing object recognition. The network's size is only limited by the size of the memory available on our computer as well as the amount of training time we are willing to provide for. In our experiments with CNNs, we train our model for up to 8 hours, and use up to 1 GB of memory.

Neural networks and machine learning algorithms in general require a flexible environment where new algorithmic prototypes and experiments can be set up as quickly as possible with best possible computational performance. To that end, we use a new framework called Torch7 [6], that is especially suited to achieve both of these competing goals.

1.2 Torch7

Torch7 is a versatile numeric computing framework and machine learning library, which extends a lightweight and powerful programming language, Lua. It provides a flexible environment to design, train and deploy learning machines. High performance is obtained via efficient OpenMP/SSE and CUDA implementations of low-level numeric routines. Torch7 can also easily be interfaced to third-party software thanks to Lua's light C interface.

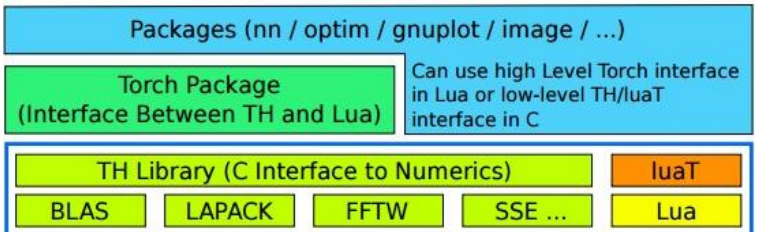


Figure 2. Torch7 architecture stack.

Modular Structure of Torch7. Low level numerical libraries are interfaced with TH to provide a unified tensor library. luaT provides essential data structures for object/class manipulation in Lua. The core Torch package uses TH and luaT to provide a numerical computing environment purely in Lua. All other packages can use either Torch interface from inside Lua scripting environment or can interface low-level C interfaces for increased performance optimizations.

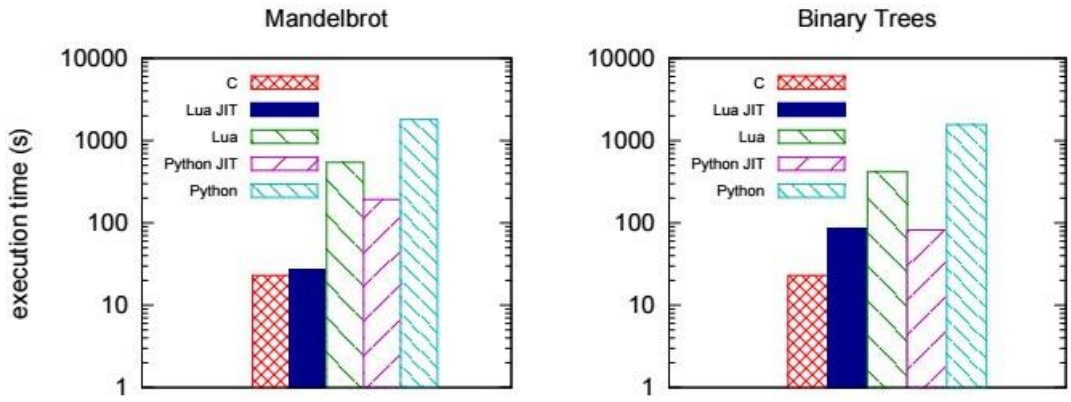


Figure 3. Performance evaluation of various machine learning platforms.

This figure compares runtime efficiency of various available machine learning platforms. Clearly, Torch7 is a winner. Torch7 not only provides for neural network support, but provides a versatile environment for the development of new numeric algorithms in general. As seen in Figure 3, LuaJIT is the fastest interpreted language. Lua offers at least two important advantages over Python. First and foremost, the simplicity of integrating existing C/C++ libraries is very important. Many efficient numerical algorithms are implemented in specialized packages in BLAS, LAPACK, FFTW and similar libraries. A lightweight interface to existing code is crucial for achieving a high performance environment. Second, since Lua is embeddable in C/C++, any prototyped application can be turned into a final system/product with very little extra effort. Since Lua is written in pure C and does not have dependency to any external library, it can be easily used in embedded applications like, Android, iOS 10, FPGAs 11 and DSPs [6].

In Torch7, neural networks are viewed as a set of modules to be connected in a graph. The 'nn' package we use in our project provides a standard set of neural network modules. We use the sequential set of models available here in. The input format is the torch tensor, provided for in the Torch TH library. Tensors are also used to represent output states and internal states. Tensors are used as data containers to interact seamlessly with the rest of the library

In Torch7, the neural network modules are able to compute the partial derivative with respect to its parameters and its inputs, given the partial derivatives with respect to its outputs. Thus, any complicated network structure can be trained using gradient-based optimization methods. We experiment with the supported Batch, mini-batch and stochastic gradient descent algorithms. More advanced algorithms, such as second-order gradient descent algorithms like conjugate gradient or LBFGS are also possible, thanks to a numerical package called "optim". While this optimization package is designed to be used stand-alone, it also provides second-order optimization capabilities for neural networks when used with the "nn" package. Other packages which we use are 'optim', providing steepest descent and other advanced optimization algorithms, and 'image', which provides for all image processing related functions such as loading, saving, color preprocessing etc.

1.3 Problem Description

In this project, we first demonstrate the lack of learning capacity of conventional feed forward neural networks in the presence of fewer training data as compared to a much larger number of parameters to be trained in the model. For this purpose, we implement a feed forward neural network for the problem of Forest Cover Type Prediction [4], and analyze its results by observing the confusion matrices generated

and plotting the training and test accuracy plots. Secondly, we implement a Convolutional Neural Network trained on the SVHN [5] dataset for vision, and analyze the results. We show using these results that Convolutional Neural Networks have a much larger learning capacity due to their ability to extract features of interest from the input data, create complex internal representations for them, and then be able to make generally correct predictions about the data. They overcome overfitting by reducing the total number of parameters to train. Finally, we make conclusions about the nature of deep learning neural networks from the results obtained. All implementations are done on the Torch7 platform, a powerful numeric computing framework built on Lua language.

1.4 Outline of this report

This report is organized as follows. First we consider the requirements to implement this project in terms of hardware, software and datasets. Next we consider the design of our neural network models. We use the feed forward neural network for forest cover type prediction. We consider the various hyper-parameters such as the number of hidden units in each hidden layer, the number of hidden layers, the optimization method to employ (Stochastic Gradient Descent, Batch Gradient Descent, etc.), validation methods to use (k-fold cross validation etc.), regularization constant, etc. For the SVHN prediction problem, we design the Convolutional Neural Network model to implement. We discuss the various options for deciding on the number of filters to use and pooling methods to be employed (such as Max pooling, min pooling, Lp-pooling etc.) in the sampling and subsampling stages. The fully connected layer to follow these filters is then designed vis-à-vis the feed forward neural network model used for the former problem.

We next consider the implementation in Torch7 and representation of the model in Torch7, training times for both models, and the various experiments which led us to choose specific hyper parameters for optimal results.

Finally we examine the results obtained, and discuss the inferences we make from them, and present our findings in the concluding section of the report.

Chapter 2: Requirements

2.1 Software Requirements

To implement the modules of this project, we have the following **software requirements**.

1. Platform for coding the networks – Torch7
2. Programming Language – LuaJIT
3. Platform for plotting graphs and analyzing results – Matlab
4. Operating system compatible with platform – Ubuntu 14.04
5. Documentation software – MS Office
6. **Torch7 libraries required:**
    a. Neural Network Package **'nn'**: This package provides an easy and modular way to build and train simple or complex neural networks using Torch.
    b. Optimization Package **'optim**': This package contains several optimization routines for Torch. Each optimization algorithm is based on the same interface, defined as follows.

        x*, {f}, ... = optim.method(func, x, state).

    c. 'image': An image processing package. It provides all the standard image processing functions such as loading and saving images, rescaling, rotating, converting color spaces, filtering operations etc.

2.2 Hardware Requirements

Hardware requirements of the implementation of the project are as follows.

1. Memory requirements for reading SVHN Dataset – approximately 1 GB.
2. Processor requirements – Intel i3 processor; greater processing power would lead to results faster.

2.3 Dataset Requirements

Dataset requirements – to train the two models, we obtained the following data from their respective sources.

1. Forest Cover Type Prediction Challenge - The study area includes four wilderness areas located in the Roosevelt National Forest of northern Colorado. Each observation is a 30m x 30m patch. We predict an integer classification for the forest cover type. We obtained the data from [4].

   The seven types are:

   1. *Spruce/Fir*
   2. *Lodgepole Pine*
   3. *Ponderosa Pine*
   4. *Cottonwood/Willow*
   5. *Aspen*
   6. *Douglas-fir*
   7. *Krummholz*

   The training set (15120 observations) contains both features and the Cover Type [1].

   Data Fields

   Elevation - Elevation in meters.
   Aspect - Aspect in degrees azimuth.
   Slope - Slope in degrees.
   Horizontal_Distance_To_Hydrology - Horizontal distance to nearest surface water features.
   Vertical_Distance_To_Hydrology - Vertical distance to nearest surface water features.
   Horizontal_Distance_To_Roadways - Horizontal distance to nearest roadway.
   Hillshade_9am (0 to 255 index) – Hill shade index at 9am, summer solstice.
   Hillshade_Noon (0 to 255 index) – Hill shade index at noon, summer solstice.
   Hillshade_3pm (0 to 255 index) – Hill shade index at 3pm, summer solstice.
   Horizontal_Distance_To_Fire_Points - Horizontal distance to nearest wildfire ignition points.
   Wilderness_Area (4 binary columns, 0 = absence or 1 = presence) - Wilderness area designation.
   Soil_Type (40 binary columns) - Soil Type designation.
   Cover_Type (7 types, integers 1 to 7) - Forest Cover Type designation.

   The wilderness areas are:

1 - Rawah Wilderness Area

2 - Neota Wilderness Area

3 - Comanche Peak Wilderness Area

4 - Cache la Poudre Wilderness Area

The soil types are:

1 Cathedral family - Rock outcrop complex, extremely stony.

2 Vanet - Ratake families complex, very stony.

3 Haploborolis - Rock outcrop complex, rubbly.

4 Ratake family - Rock outcrop complex, rubbly.

5 Vanet family - Rock outcrop complex complex, rubbly.

6 Vanet - Wetmore families - Rock outcrop complex, stony.

7 Gothic family.

8 Supervisor - Limber families complex.

9 Troutville family, very stony.

10 Bullwark - Catamount families - Rock outcrop complex, rubbly.

11 Bullwark - Catamount families - Rock land complex, rubbly.

12 Legault family - Rock land complex, stony.

13 Catamount family - Rock land - Bullwark family complex, rubbly.

14 Pachic Argiborolis - Aquolis complex.

15 unspecified in the USFS Soil and ELU Survey.

16 Cryaquolis - Cryoborolis complex.

17 Gateview family - Cryaquolis complex.

18 Rogert family, very stony.

19 Typic Cryaquolis - Borohemists complex.

20 Typic Cryaquepts - Typic Cryaquolls complex.

21 Typic Cryaquolls - Leighcan family, till substratum complex.

22 Leighcan family, till substratum, extremely bouldery.

23 Leighcan family, till substratum - Typic Cryaquolls complex.

24 Leighcan family, extremely stony.

25 Leighcan family, warm, extremely stony.

26 Granile - Catamount families complex, very stony.

27 Leighcan family, warm - Rock outcrop complex, extremely stony.

28 Leighcan family - Rock outcrop complex, extremely stony.

29 Como - Legault families complex, extremely stony.

30 Como family - Rock land - Legault family complex, extremely stony.

31 Leighcan - Catamount families complex, extremely stony.

32 Catamount family - Rock outcrop - Leighcan family complex, extremely stony.

33 Leighcan - Catamount families - Rock outcrop complex, extremely stony.

34 Cryorthents - Rock land complex, extremely stony.

35 Cryumbrepts - Rock outcrop - Cryaquepts complex.

36 Bross family - Rock land - Cryumbrepts complex, extremely stony.

37 Rock outcrop - Cryumbrepts - Cryorthents complex, extremely stony.

38 Leighcan - Moran families - Cryaquolls complex, extremely stony.

39 Moran family - Cryorthents - Leighcan family complex, extremely stony.

40 Moran family - Cryorthents - Rock land complex, extremely stony.

2. SVHN Vision - SVHN is a real-world image dataset for developing machine learning and object recognition algorithms with minimal requirement on data preprocessing and formatting. It can be seen as similar in flavor to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images). SVHN is obtained from house numbers in Google Street View images. We obtained the dataset from [5].

   This problem is that of a restricted scene: recognizing the house number of real world house numbers.

   The entire end-to-end system (described later) includes two main stages: (i) a detection stage that locates individual house numbers in a large image, and (ii) a recognition stage that performs a search over possible character locations in the detected house number, classifying each candidate frame as one of ten digits (0 through 9) [5]. Therefore, when the classifier is presented with a house view number, it first extracts useful features from it, and then tries to predict its value using a fully connected logistic network.

Figure 4. Cropped Samples of the SVHN dataset.

Character level ground truth in an MNIST-like format. All digits have been resized to a fixed resolution of 32-by-32 pixels. The original character bounding boxes are extended in the appropriate dimension to become square windows, so that resizing them to 32-by-32 pixels does not introduce aspect ratio distortions. Nevertheless this preprocessing introduces some distracting digits to the sides of the digit of interest.

Figure 5. An example of a real world house number in the SVHN set.

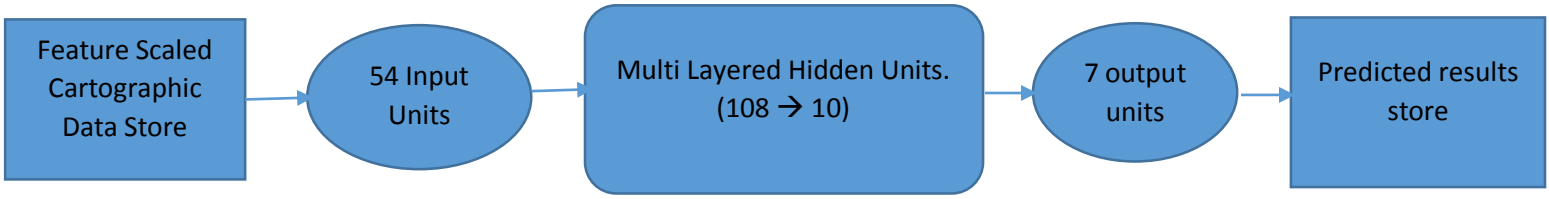
Chapter 3: Design and Analysis

3.1 Data Flow Diagrams

1. Forest Cover Type Prediction – Data Flow through the feed forward neural network.

   We have 15120 training instances, each containing 54 input parameters. These input parameters are fed into the network through the 54 input units. They then pass to the first hidden layer, where weighted linear sums of the input are fed into each hidden unit. Some hidden units may be activated at this stage. The second hidden layer mimics the functioning of the first layer except its input parameters are dependent on the previous layer.

   This is the fully connected 2 layer feed forward neural network. In the last stage, we have a logistic classifier to classify the forest cover type into one of the 7 types. The input, internal and output states are all stored in Torch tensors.



Total number of input parameters/training example = 54

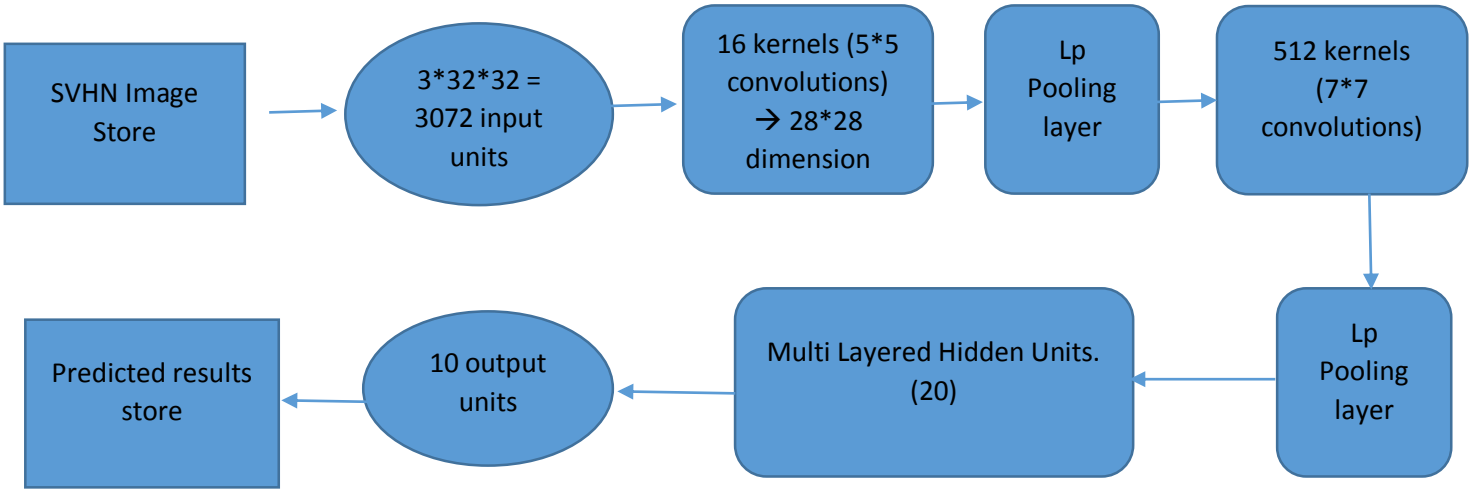
Total number of parameters to learn = 54*108 + 108*10 + 10*7 = 6982

Total number of training examples = 15120

2. SVHN Prediction using Convolutional Neural Networks

   We learn the input feature representations using banks of filters, generally called the sampling layer, followed by a pooling layer, also called the sub-sampling layer. Once we have built up the feature

representation, we add a final softmax classification layer that is trained using maximum likelihood. The full stacked network is finally fine-tuned using back-propagation.



Total number of input parameters per image = 3*32*32 = 3072 (3 channels per 32 by 32 image)

Total number of parameters to learn = (16*5*5) + (512*7*7) + (512*20) = 35728

Therefore, a preliminary comparison yields that #parameters/input sample in MLP by #parameters/input sample in CNN is ~~ 11.

This indicates that using a convolutional neural network decreases the number of parameters to be learned by more than 10 times as compared to a feed forward network, giving much faster training time and greater accuracy. The aim of this project is to demonstrate this by implementing the two networks and analyzing the results obtained.

A major drawback of many feature learning systems is their complexity. These algorithms usually require a careful selection of multiple hyper-parameters such as learning rates, momentum, sparsity penalties, weight decay, and so on that must be chosen through cross-validation. For the purpose of choosing these parameters, we experimented repeatedly with different network settings until we got satisfactory results. However, this method lacks any formal framework and therefore, results remain conditional.

3.2 Kaggle Challenge – Forest Cover Type Prediction

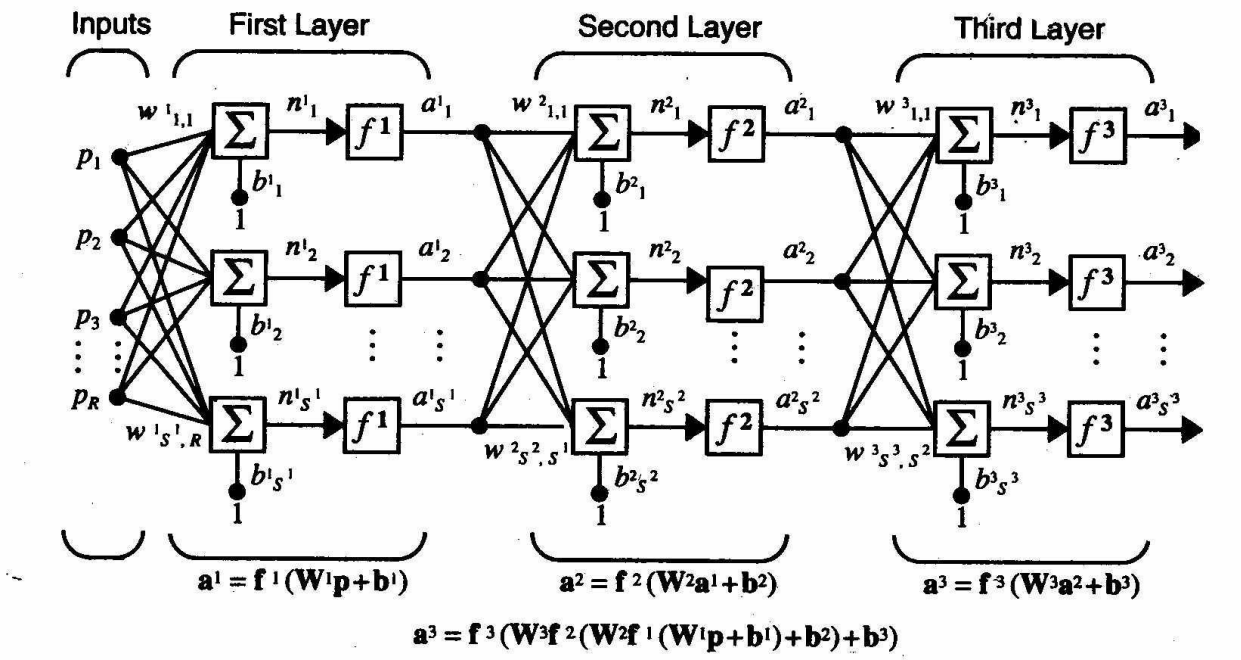


Figure 6. Feed Forward Neural Network Model

This figure illustrates the multi-layer perceptron model implemented to analyze the performance of conventional feed forward neural networks in the presence of a large number of parameters and fewer training examples.

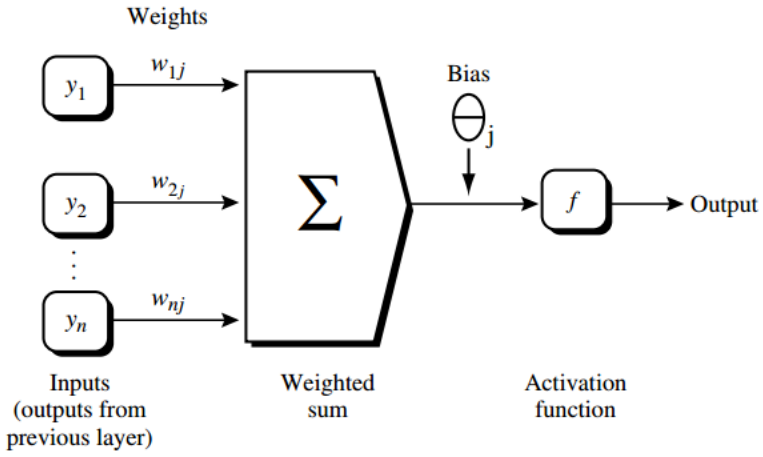


Figure 7. A hidden unit in the MLP network.

The first hidden layer consists of approximately double the number of input parameters. This number of hidden units is generally used in a multi-layer perceptron as a large number of hidden units lead to overfitting.

- Since there are 54 input parameters, the number of hidden units in the first hidden layer are 100.
- In the second hidden layer, we reduced this number to 10.
- The non-linearity function used in both layers was '**Tanh()**' due to its conveniently calculable derivative, required for backpropagation.

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}},$$

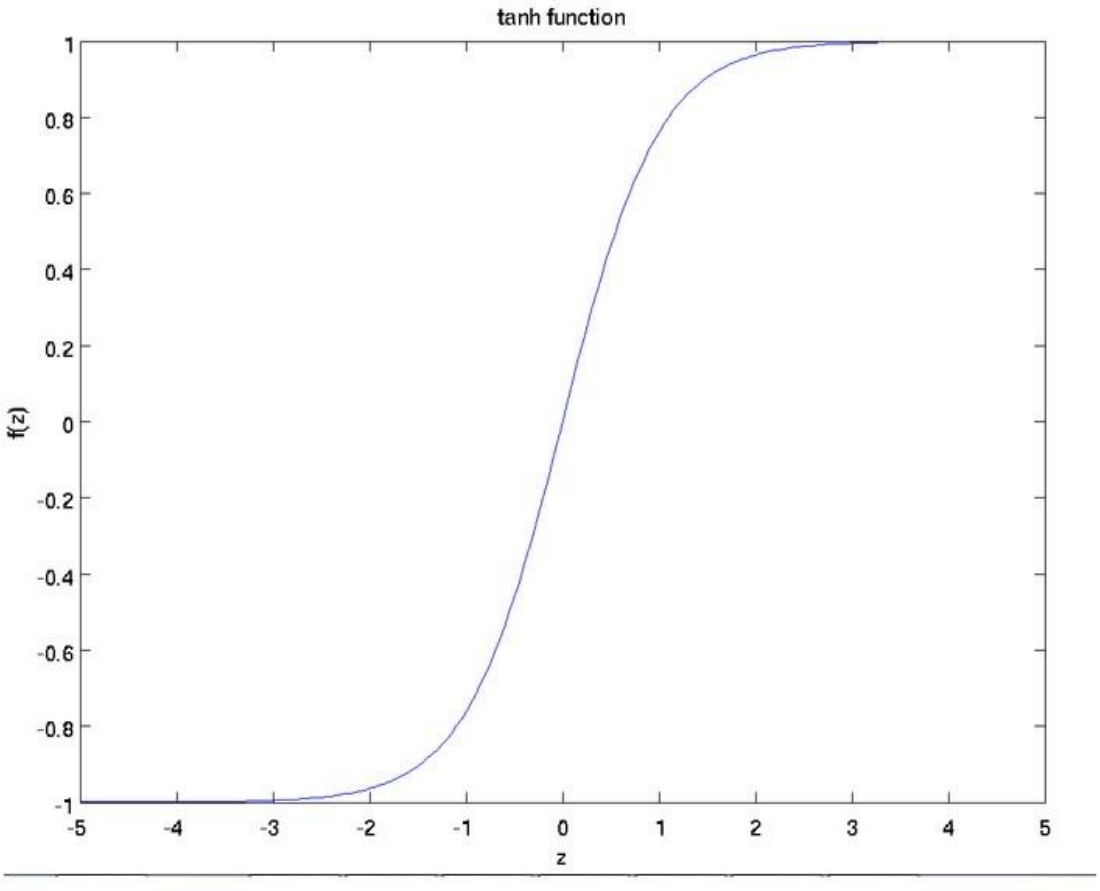


Figure 8. Tanh function.

- Lastly, we used a **LogSoftMax** regression model to select the final activation unit [8]. This model generalizes logistic regression to classification problems where the class label *y* can take on more than two possible values (in this case, 7). Thus, in our training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$, we now have that $y^{(i)} \in \{1, 2, \ldots, k\}$.

  The hypothesis now estimates the following function.

$$\begin{aligned} p(y^{(i)} = j | x^{(i)}; \theta) &= \frac{e^{(\theta_j - \psi)^T x^{(i)}}}{\sum_{l=1}^{k} e^{(\theta_l - \psi)^T x^{(i)}}} \\ &= \frac{e^{\theta_j^T x^{(i)}} e^{-\psi^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}} e^{-\psi^T x^{(i)}}} \\ &= \frac{e^{\theta_j^T x^{(i)}}}{\sum_{l=1}^{k} e^{\theta_l^T x^{(i)}}}. \end{aligned}$$

- While preprocessing the data, we normalized all columns to values between 0 and 1 by the following method.

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- Penalizing the size of the coefficients is a common strategy for robust modeling in regression/classification with high-dimensional data. For regularizing the cost function (a method used to counter overfitting), we augment the cost function with an additional regularization constant.
- The regularization parameter ƛ is a control on fitting parameters. As the magnitudes of the fitting parameters increase, there will be an increasing penalty on the cost function. This penalty is dependent on the squares of the parameters as well as the magnitude of ƛ.

3.3 SVHN Prediction

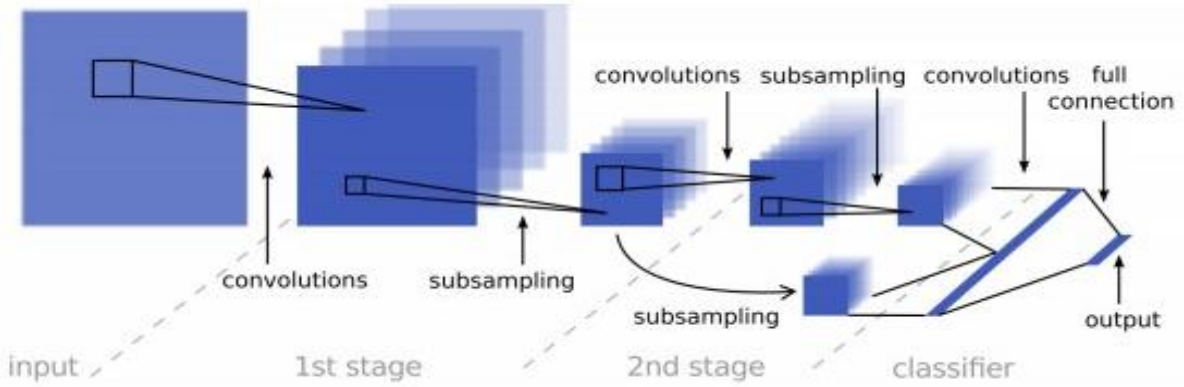


Figure 9. A 2 stage ConvNet architecture where Multi Stage features are fed to a 2 layer classifier. The 1st stage features are branched out, subsampled again, and then concatenated to 2nd stage features.

The model we implement for this problem is a class of models called Convolutional Neural Networks. Convolutional Neural Nets are hierarchical feature learning neural networks whose structure is biologically inspired. ConvNets learn features all the way from pixels to the classifier.

Convolutional neural networks are organized in layers: convolutional layers and subsampling layers. Each layer has a topographic structure, i.e., each neuron is associated with a fixed two dimensional position that corresponds to a location in the input image, along with a receptive field (the region of the input image that influences the response of the neuron). At each location of each layer, there are a number of different neurons, each with its set of input weights, associated with neurons in a rectangular patch in the previous layer. The same set of weights, but a different input rectangular patch, are associated with neurons at different locations. One untested hypothesis is that the small fan-in of these neurons (few inputs per neuron) helps gradients to propagate through so many layers without diffusing so much as to become useless. Note that this alone would not suffice to explain the success of convolutional networks, since random sparse connectivity is not enough to yield good results in deep neural networks. However, an effect of the fan-in would be consistent with the idea that gradients propagated through many paths gradually become too diffuse, i.e., the credit or blame for the output error is distributed too widely and thinly. Another hypothesis (which does not necessarily exclude the first) is that the hierarchical local connectivity structure is a very strong prior that is particularly appropriate for vision tasks, and sets the parameters of the whole network in a favorable region (with all non-connections corresponding to zero weight) from which gradient-based optimization works well. [10]

Natural images have the property of being “stationary”, meaning that the statistics of one part of the image are the same as any other part. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and we can use the same features at all locations. More precisely, having learned features over small (say 8x8) patches sampled randomly from the larger image, we can then apply this learned 8x8 feature detector anywhere in the image. Specifically, we can take the learned 8x8 features and ”convolve” them with the larger image, thus obtaining a different feature activation value at each location in the image.

3.3.1 Convolutions

On the relatively small images that we were working with (e.g., 8x8 patches for the sparse auto-encoder assignment, 28x28 images for the MNIST dataset), it is computationally feasible to learn features on the entire image, with fully connected feed forward neural networks. However, with larger images (e.g., 32x32; 96x96 images) learning features that span the entire image (fully connected networks) is very computationally expensive—there would be about 104 input units, and assuming we want to learn 100 features, there would be on the order of 106 parameters to learn. The feedforward and backpropagation computations would also be about 102 times slower, compared to 28x28 images.

One simple solution to this problem is to restrict the connections between the hidden units and the input units, allowing each hidden unit to connect to only a small subset of the input units. Specifically, each hidden unit will connect to only a small contiguous region of pixels in the input. (For input modalities different than images, there is often also a natural way to select "contiguous groups" of input units to connect to a single hidden unit as well; for example, for audio, a hidden unit might be connected to only the input units corresponding to a certain time span of the input audio clip.)

This idea of having locally connected networks also draws inspiration from how the early visual system is wired up in biology. Specifically, neurons in the visual cortex have localized receptive fields (i.e., they respond only to stimuli in a certain location).

Natural images have the property of being stationary, meaning that the statistics of one part of the image are the same as any other part. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and we can use the same features at all locations.

More precisely, having learned features over small (say 8x8) patches sampled randomly from the larger image, we can then apply this learned 8x8 feature detector anywhere in the image. Specifically, we can

take the learned 8x8 features and convolve them with the larger image, thus obtaining a different feature activation value at each location in the image.

Given some large $r \times c$ images x_{large}, we first train a sparse auto-encoder on small $a \times b$ patches x_{small} sampled from these images, learning k features $f = \sigma(W(1)x_{\text{small}} + b(1))$ (where σ is the sigmoid function), given by the weights $W(1)$ and biases $b(1)$ from the visible units to the hidden units. For every $a \times b$ patch x_s in the large image, we compute $f_s = \sigma(W(1)x_s + b(1))$, giving us $f_{\text{convolved}}$, a $k \times (r - a + 1) \times (c - b + 1)$ array of convolved features.

3.3.2 Pooling

After obtaining features using convolution, we would next like to use them for classification. In theory, one could use all the extracted features with a classifier such as a softmax classifier, but this can be computationally challenging. Consider for instance images of size 96x96 pixels, and suppose we have learned 400 features over 8x8 inputs. Each convolution results in an output of size (96 − 8 + 1) * (96 − 8 + 1) = 7921, and since we have 400 features, this results in a vector of 892 * 400 = 3,168,400 features per example. Learning a classifier with inputs having 3+ million features can be unwieldy, and can also be prone to over-fitting.

To address this, we use the "stationarity" property of images, which implies that features that are useful in one region are also likely to be useful for other regions. Thus, to describe a large image, one natural approach is to aggregate statistics of these features at various locations. After obtaining our convolved features as described earlier, we decide the size of the region, say $m \times n$ to pool our convolved features over. Then, we divide our convolved features into disjoint $m \times n$ regions. We may pool over this region such as max pooling or mean pooling. A specific kind of pooling, called L2-pooling, can be described by the following figure. This can be further generalized to Lp-pooling, where we use powers of 'p' instead of power of 2.

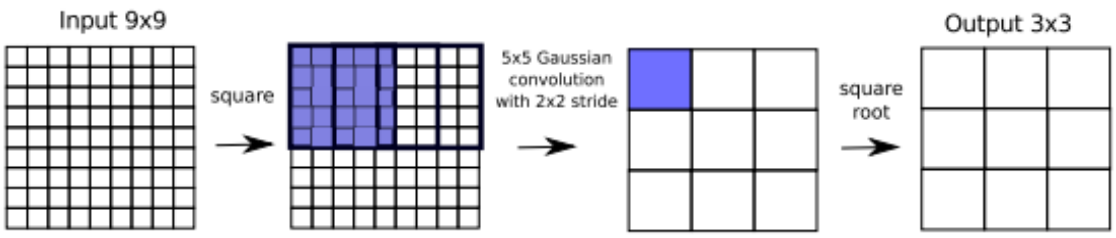


Figure 10. L2 Pooling

A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers as in a standard multilayer neural network. The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units. A CNN consists of a number of convolutional and subsampling layers optionally followed by fully connected layers. The input to a convolutional layer is a $m\ x\ m\ x\ r$ image where m is the height and width of the image and r is the number of channels, e.g. an RGB image has $r = 3$. The convolutional layer will have k filters (or kernels) of size $n\ x\ n\ x\ q$ where n is smaller than the dimension of the image and q can either be the same as the number of channels r or smaller and may vary for each kernel. The size of the filters gives rise to the locally connected structure which are each convolved with the image to produce k feature maps of size $m - n + 1$. Each map is then subsampled typically with mean or max pooling over $p\ x\ p$ contiguous regions where p ranges between 2 for small images (e.g. MNIST) and is usually not more than 5 for larger inputs. We have used Lp-pooling as described in [8]. Either before or after the subsampling layer an additive bias and sigmoidal nonlinearity is applied to each feature map.

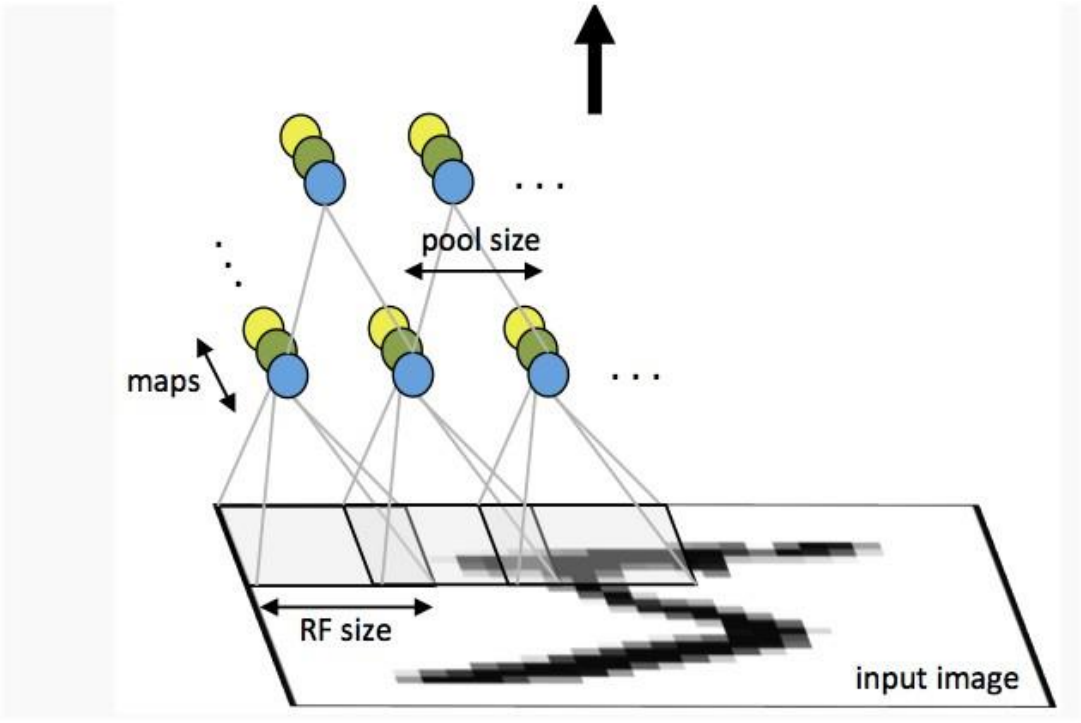


Figure 11. First layer of a convolutional neural network with pooling. Units of the same color have tied weights and units of different color represent different filter maps. [9]

3.3.3 *Data Pre-processing of the SVHN dataset*

The SVHN classification dataset [3] contains 32x32 images with 3 color channels. The dataset is divided into three subsets: train set, extra set and test set. The extra set is a large set of easy samples and train set is a smaller set of more difficult samples. This distribution allows to measure success on easy samples but puts more emphasis on difficult ones.

YUV is a color space typically used as part of a color image pipeline. It encodes a color image or video taking human perception into account, allowing reduced bandwidth for chrominance components,

thereby typically enabling transmission errors or compression artifacts to be more efficiently masked by the human perception than using a "direct" RGB-representation. [29]

Samples are pre-processed with a local contrast normalization (with a $7x7$ kernel) on the Y channel of the YUV space followed by a global contrast normalization over each channel.

Figure 12. SVHN Cropped Numbers Sample.

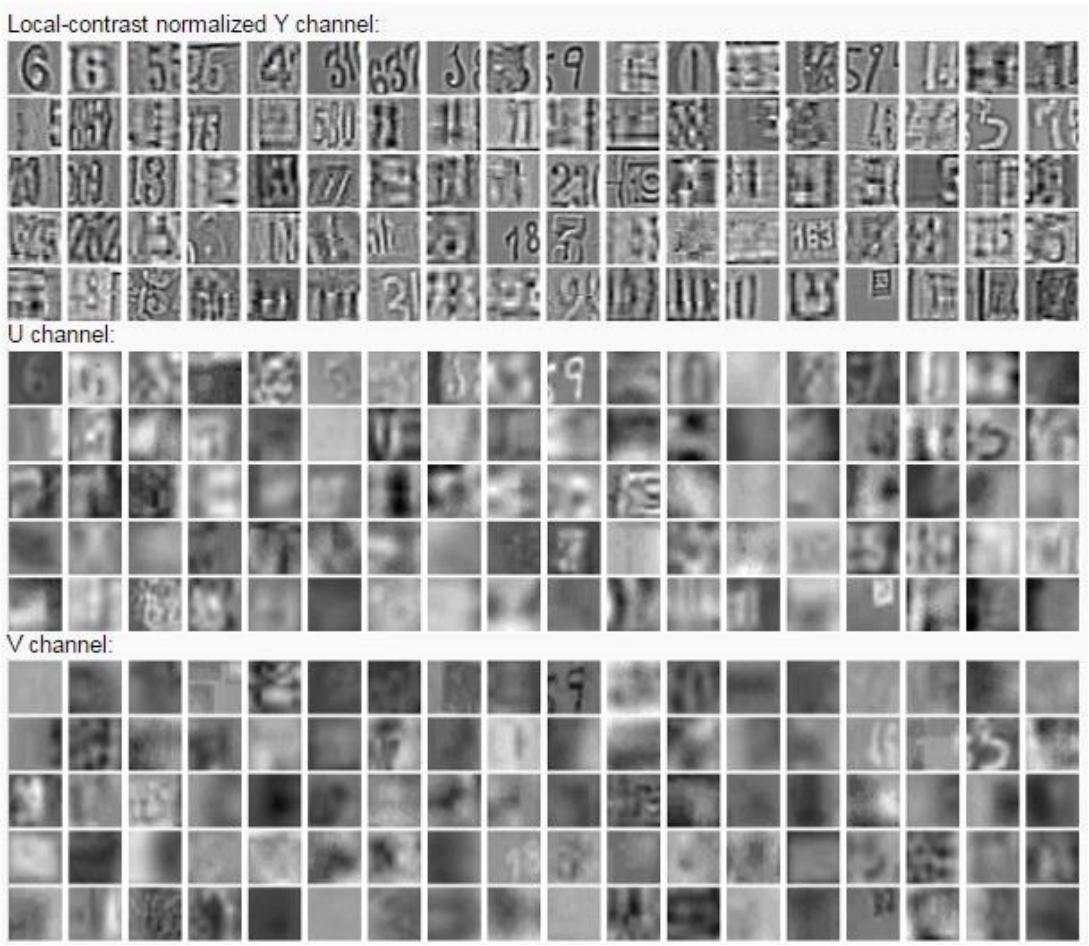


Figure 13. SVHN Preprocessed Sample

3.3.4 Architecture of the CNN

A Convolutional Neural Network (CNN) is comprised of one or more convolutional layers (often with a subsampling step) and then followed by one or more fully connected layers as in a standard multilayer neural network. The architecture of a CNN is designed to take advantage of the 2D structure of an input image (or other 2D input such as a speech signal). This is achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features. Another benefit of CNNs is that they are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units. In this article we will discuss the architecture of a CNN and the back

propagation algorithm to compute the gradient with respect to the parameters of the model in order to use gradient based optimization. [5]

The input to a convolutional layer is a m x m x r image where m is the height and width of the image and r is the number of channels. In our case, r = 3 (three components of the YUV color space), and m = 32.

The convolutional layer will have k filters (or kernels) of size n x n x q where n is smaller than the dimension of the image and q can either be the same as the number of channels r or smaller and may vary for each kernel. The size of the filters gives rise to the locally connected structure which are each convolved with the image to produce k feature maps of size m−n+1. Each map is then subsampled typically with mean or max pooling over y x y contiguous regions where y ranges between 2 for small images.

The CNN has 2 stages of feature extraction and a two-layer non-linear classifier. The first convolution layer produces 16 features with $5x5$ convolution filters while the second convolution layer outputs 512 features with $7x7$ filters. The output to the classifier also includes inputs from the first layer, which provides local features/motifs to reinforce the global features. The classifier is a 2-layer non-linear classifier with 20 hidden units. Hyper-parameters such as learning rate, regularization constant and learning rate decay were tuned on the validation set. We use stochastic gradient descent as our optimization method and shuffle our dataset after each training iteration. For the pooling layers, we compare Lp-pooling for the value p = 1, 2, 4, 8, 12.

3.3.5 Optimization of parameters using Stochastic Gradient Descent

The standard gradient descent algorithm updates the parameters θ of the objective J(θ) as,

$$\theta = \theta - \alpha \nabla \theta E[J(\theta)]$$

where the expectation in the above equation is approximated by evaluating the cost and gradient over the full training set. Stochastic Gradient Descent (SGD) simply does away with the expectation in the update and computes the gradient of the parameters using only a single or a few training examples. The new update is given by,

$$\theta = \theta - \alpha \nabla \theta J(\theta; x(i), y(i))$$

with a pair $(x(i), y(i))$ from the training set.

Generally each parameter update in SGD is computed w.r.t a few training examples or a minibatch as opposed to a single example. The reason for this is twofold: first this reduces the variance in the parameter update and can lead to more stable convergence, second this allows the computation to take advantage of highly optimized matrix operations that should be used in a well vectorized computation of the cost and gradient. A typical minibatch size is 256, although the optimal size of the minibatch can vary for different applications and architectures.

In SGD the learning rate α is typically much smaller than a corresponding learning rate in batch gradient descent because there is much more variance in the update. Choosing the proper learning rate and schedule (i.e. changing the value of the learning rate as learning progresses) can be fairly difficult. One standard method that works well in practice is to use a small enough constant learning rate that gives stable convergence in the initial epoch (full pass through the training set) or two of training and then halve the value of the learning rate as convergence slows down. An even better approach is to evaluate a held out set after each epoch and anneal the learning rate when the change in objective between epochs is below a small threshold. This tends to give good convergence to a local optima. Another commonly used schedule is to anneal the learning rate at each iteration t as $ab + t$ where a and b dictate the initial learning rate and when the annealing begins respectively. More sophisticated methods include using a backtracking line search to find the optimal update.

One final but important point regarding SGD is the order in which we present the data to the algorithm. If the data is given in some meaningful order, this can bias the gradient and lead to poor convergence. Generally a good method to avoid this is to randomly shuffle the data prior to each epoch of training.

Momentum - If the objective has the form of a long shallow ravine leading to the optimum and steep walls on the sides, standard SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum. The objectives of deep architectures have this form near local optima and thus standard SGD can lead to very slow convergence particularly after the initial steep gains. Momentum is one method for pushing the objective more quickly along the shallow ravine. The momentum update is given by,

$$v = \gamma v + \alpha \nabla \theta J\big(\theta; x(i), y(i)\big)$$

$$\theta = \theta - v$$

In the above equation v is the current velocity vector which is of the same dimension as the parameter vector θ. The learning rate α is as described above, although when using momentum α may need to be

smaller since the magnitude of the gradient will be larger. Finally $\gamma \in (0,1]$ determines for how many iterations the previous gradients are incorporated into the current update. Generally γ is set to 0.5 until the initial learning stabilizes and then is increased to 0.9 or higher.

3.3.6 Calculating derivatives using Backpropagation

Let $\delta^{(l+1)}$ be the error term for the $(l+1)$-st layer in the network with a cost function $J(W, b; x, y)$ where (W, b) are the parameters and (x, y) are the training data and label pairs. If the l-th layer is densely connected to the $(l+1)$-st layer, then the error for the l-th layer is computed as

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

and the gradients are

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

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

$$\delta_k^{(l)} = \text{upsample}\left((W_k^{(l)})^T \delta_k^{(l+1)}\right) \bullet f'(z_k^{(l)})$$

Where k indexes the filter number and $f'(z_k^{(l)})$ is the derivative of the activation function. The upsample operation has to propagate the error through the pooling layer by calculating the error w.r.t to each unit incoming to the pooling layer. For example, if we have mean pooling then $upsample$ simply uniformly distributes the error for a single pooling unit among the units which feed into it in the previous layer. In max pooling the unit which was chosen as the max receives all the error since very small changes in input

would perturb the result only through that unit. Finally, to calculate the gradient w.r.t to the filter maps, we rely on the border handling convolution operation again and flip the error matrix $\delta_k^{(l)}$ the same way we flip the filters in the convolutional layer.

$$\nabla_{W_k^{(l)}} J(W, b; x, y) = \sum_{i=1}^{m} (a_i^{(l)}) * \mathrm{rot90}(\delta_k^{(l+1)}, 2),$$
$$\nabla_{b_k^{(l)}} J(W, b; x, y) = \sum_{a,b} (\delta_k^{(l+1)})_{a,b}.$$

Where $a^{(l)}$ is the input to the l-th layer, and $a^{(1)}$ is the input image. The operation $(a_i^{(l)}) * \delta_k^{(l+1)}$ is the "valid" convolution between i-th input in the l-th layer and the error w.r.t. the k-th filter [9].

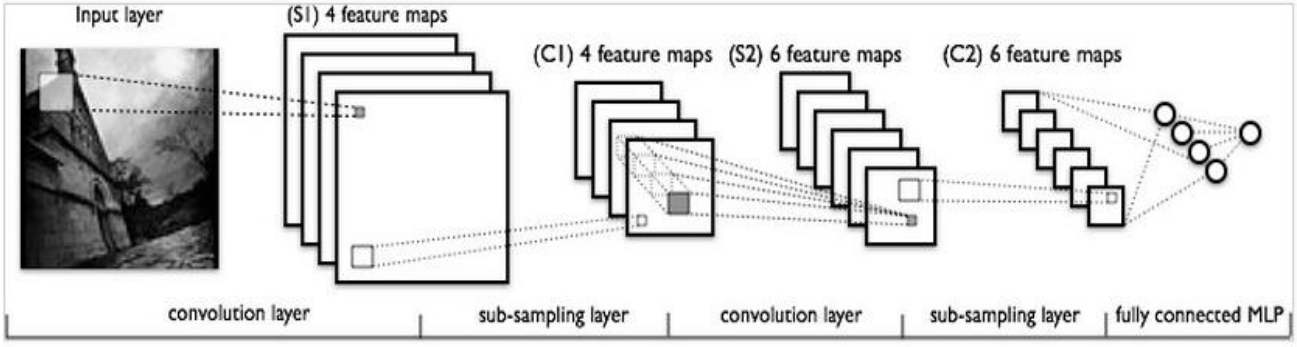


Figure 14. A bank of different filters allows each patch of an image to be represented in several ways! Also, convolutions ensure detector of the same feature are available 'everywhere'.

Chapter 4: Implementation

4.1 Forest Cover Type Prediction

Our experiments with the network described above for this problem showed did not give good results on the test data. Supervised training of the MLP was performed. We experimented by changing the number of hidden layers, the number of hidden units in each layer, weight decay constant, batch sizes (in Stochastic Gradient Descent) etc. Our implementation used the 'nn' neural network library in Torch7 for describing the neural network model. Next we used the 'optim' library for running various optimization algorithms such as Stochastic Gradient Descent, 'CG', 'LBFGS', and 'ASGD', and also for determining the confusion matrix at each epoch.

The following trace shows the implementation of various computational units in the model using Torch libraries.

```
<torch> set nb of threads to 2
<trainer> using model:
nn.Sequential {
  [input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> output]
  (1): nn.Linear(ninputs, nhidden)
  (2): nn.Tanh
  (3): nn.Linear(nhidden, nhidden_2)
  (4): nn.Tanh
  (5): nn.Linear(nhidden_2, #classes)
  (6): (nn.LogSoftMax())
}
```

This model was defined as follows in Torch7.

classes = {'Spruce/Fir','Lodgepole Pine','Ponderosa Pine','Cottonwood/Willow','Aspen','Douglas-fir','Krummholz'}

model = nn.Sequential()

model:add(nn.Reshape(ninputs))

model:add(nn.Linear(ninputs, nhidden))

model:add(nn.Tanh())

model:add(nn.Linear(nhidden, nhidden_2))

model:add(nn.Tanh())

model:add(nn.Linear(nhidden_2, #classes))

model:add(nn.LogSoftMax())

parameters,gradParameters = model:getParameters()

criterion = nn.ClassNLLCriterion()

4.2 SVHN

We used supervised training on the ConvNet. These deep learning neural networks are capable of learning multiple features of the input at each stage of learning. We used stochastic gradient descent as our optimization method and shuffled our dataset after each training iteration. In all there are about 73000 training images. We experimented with various values of 'p' in Lp-pooling. The architecture of the model has been described previously. There are 10 output classes. To implement the Convolutional Neural Network model and for running the optimization on the network, we use 'nn' and 'optim' libraries.

The following trace shows the implementation of various computational units in the model using Torch libraries.

<torch> set nb of threads to 2

<trainer> using model:

nn.Sequential {

[input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> (7) -> (8) -> (9) -> (10) -> (11) -> (12) -> (13) -> output]

(1): nn.SpatialConvolutionMap

(2): nn.Tanh

(3): nn.Sequential {

[input -> (1) -> (2) -> (3) -> output]

(1): nn.Power

```
   (2): nn.SpatialSubSampling
   (3): nn.Power
 }
 (4): nn.SpatialSubtractiveNormalization
 (5): nn.SpatialConvolutionMap
 (6): nn.Tanh
 (7): nn.Sequential {
   [input -> (1) -> (2) -> (3) -> output]
   (1): nn.Power
   (2): nn.SpatialSubSampling
   (3): nn.Power
 }
 (8): nn.SpatialSubtractiveNormalization
 (9): nn.Reshape
 (10): nn.Linear
 (11): nn.Tanh
 (12): nn.Linear
 (13): nn.LogSoftMax
}
```

This model was defined as follows in Torch7.

```
  classes = {'1','2','3','4','5','6','7','8','9','0'}
  model = nn.Sequential()

  model:add(nn.SpatialConvolutionMap(nn.tables.random(3,16,1), 5, 5))
  model:add(nn.Tanh())
  model:add(nn.SpatialLPPooling(16,24,2,2,2,2))

  model:add(nn.SpatialSubtractiveNormalization(16, image.gaussian1D(7)))
  model:add(nn.SpatialConvolutionMap(nn.tables.random(16, 256, 4), 5, 5))
  model:add(nn.Tanh())
  model:add(nn.SpatialLPPooling(256,12,2,2,2,2))
```

```
model:add(nn.SpatialSubtractiveNormalization(256, image.gaussian1D(7)))
model:add(nn.Reshape(256*5*5))
model:add(nn.Linear(256*5*5, 128))
model:add(nn.Tanh())
model:add(nn.Linear(128,#classes))
model:add(nn.LogSoftMax())
```

We passed the method of optimization, momentum, learning rate, amount of data to use for the purpose of testing and training as command line arguments to the script, to facilitate easy experimentation and choice of these parameters for superior results, due to the absence of any framework for quickly arriving at the best parameters, in our model.

The feature pooling layer treats each feature map separately. In Lp-pooling, the neighborhoods are stepped by a stride larger than 1 (but smaller than or equal the pooling neighborhood). This results in a reduced-resolution output feature map which is robust to small variations in the location of features in the previous layer.

The SpatialSubtractiveNormalization applies a spatial subtraction operation on a series of 2D inputs using kernel for computing the weighted average in a neighborhood. The neighborhood is defined for a local spatial region that is the size as kernel and across all features. For an input image, since there is only one feature, the region is only spatial.

The kernel is generally chosen as a Gaussian when it is believed that the correlation of two pixel locations decrease with increasing distance. On the feature dimension, a uniform average is used since the weighting across features is not known.

Chapter 5: Results and Discussion

5.1 Forest Cover Type Prediction

Our experiments with a 2 layer neural network on the Kaggle challenge yielded poor results. For instance, at epoch number 11 of the training, the confusion matrix was the following.

Confusion Matrix for Forest Cover Type Classification:

Table 1. Confusion Matrix for Forest Cover Type Prediction – Training Data.

Matrix	Row Accuracy	Class
[1303 383 2 0 65 10 183]	66.958%	Spruce/Fir
[368 1165 32 0 385 68 21]	57.136%	Lodge-pole Pine
[0 2 1324 200 71 422 0]	65.577%	Ponderosa Pine
[0 0 97 2014 0 49 0]	93.241%	Cottonwood/Willow
[1 169 50 0 1840 62 0]	86.711%	Aspen
[0 16 521 112 35 1361 0]	66.553%	Douglas-fir
[163 0 0 0 7 0 1699]	90.904%	Krummholz

Average row correct: 75.296986954553%
Average rowUcol correct (VOC measure): 61.299539889608%
Global correct: 75.394366197183%
Number of hidden units: Layer 1 ~ 100; Layer 2 ~ 10
Optimization method: Stochastic Gradient Descent
Number of train cases: 14200

Table 2. Confusion Matrix for Forest Cover Type Prediction - Test Data.

Matrix	Row Accuracy	Class
[47 38 1 0 48 35 45]	21.963%	Spruce/Fir
[22 23 5 4 11 33 23]	19.008%	Lodge-pole Pine
[1 0 33 2 0 105 0]	23.404%	Ponderosa Pine
[0 0 0 0 0 0 0]	nan%	Cottonwood/Willow
[3 5 1 0 8 17 4]	21.053%	Aspen
[0 2 10 2 1 100 0]	86.957%	Douglas-fir
[79 45 1 0 43 10 113]	38.832%	Krummholz

Average row correct: 35.202650477489%
Average rowUcol correct (VOC measure): 19.156333555778%
Global correct: 35.217391304348%
Number of hidden units: Layer 1 ~ 100; Layer 2 ~ 10
Optimization method: Stochastic Gradient Descent
Number of test cases: 921

As seen, the predictions seem to be random and inaccurate. The multi-layer perceptron model is clearly not useful for learning in this case. On the other hand, Convolutional Neural Networks performed considerably well and fairly accurate predictions were made by it on the SVHN dataset. In the following subsections, we make a side by side comparison of training and test accuracy.

Possible reasons for overfitting may be

- Model is too complex (too many predictors).
- Training data too noisy.
- Model being refined over time with ever increasing data inputs.
- Training set too small.
- A very rich hypothesis space.

We infer that we have a large number of parameters to learn given fewer input data. A rich hypothesis space may imply the presence of other models which may fit better to this data, but for our purposes, it demonstrates that feed forward neural networks are too complex to be trained for this particular problem.

5.2 SVHN prediction

The following confusion matrix represents the results obtained for test data, a collection of about 23000 images, at the 4th training epoch.

Confusion Matrix for the CNN model:

Table 3. Confusion Matrix for CNN Model - Training Data.

Matrix	Row Accuracy	Class
[13158 86 96 197 35 44 137 25 20 63]	94.928%	1
[103 9999 83 84 35 24 107 40 80 30]	94.464%	2
[121 125 7674 65 222 58 52 97 62 21]	90.314%	3
[232 92 63 6877 31 33 30 25 50 25]	92.210%	4
[54 36 231 62 6282 129 13 27 42 6]	91.282%	5
[59 26 86 55 150 5082 17 148 22 82]	88.738%	6
[182 116 57 33 20 14 5125 16 11 21]	91.600%	7
[66 68 147 49 44 152 15 4401 68 35]	87.235%	8
[61 103 62 49 64 13 25 47 4168 67]	89.461%	9
[90 31 27 23 6 59 24 31 68 4589]	92.745%	0

Average row correct: 91.297550201416%
Average rowUcol correct (VOC measure): 84.371314644814%
Global correct: 91.943432026974%

Table 4. Confusion Matrix for CNN Model - Test Data.

Matrix	Row Accuracy	Class
[4782 20 38 114 11 19 51 13 3 48]	93.783%	1
[41 3855 54 51 31 27 38 21 21 10]	92.914%	2
[60 67 2355 36 109 59 9 102 76 9]	81.714%	3
[92 23 40 2256 27 30 17 8 15 15]	89.417%	4
[14 23 55 14 2098 145 5 10 15 5]	88.003%	5
[21 8 27 16 37 1790 6 35 5 32]	90.541%	6
[125 27 19 10 14 14 1793 2 8 7]	88.806%	7
[17 17 37 23 28 132 4 1352 26 24]	81.446%	8
[27 74 14 13 17 20 12 17 1330 71]	83.386%	9
[27 18 12 4 8 86 9 5 9 1566]	89.794%	0

Average row correct: 87.980436086655%
Average rowUcol correct (VOC measure): 78.779903054237%
Global correct: 89.032728948986%

The following confusion matrices reveal the performance after 6 epochs and p = "24".

Table 5. Confusion matrix for the CNN model. Train data. p = 24.

Matrix	Accuracy	Class
[[4788 27 37 96 12 10 68 12 4 45]	93.901%	[class: 1]
[54 3818 74 51 15 18 42 23 40 14]	92.022%	[class: 2]
[65 52 2450 34 47 27 12 93 91 11]	85.010%	[class: 3]
[153 28 51 2189 16 16 21 7 22 20]	86.762%	[class: 4]
[17 22 157 21 2013 85 10 19 32 8]	84.438%	[class: 5]
[27 9 67 33 49 1643 10 65 14 60]	83.106%	[class: 6]
[126 43 25 10 4 6 1786 4 8 7]	88.460%	[class: 7]
[25 14 68 20 12 50 3 1393 56 19]	83.916%	[class: 8]
[23 57 26 15 10 11 6 10 1391 46]	87.210%	[class: 9]
[34 15 14 7 3 37 10 13 36 1575]]	90.310%	[class: 0]

+ average row correct: 87.513372898102%

+ average rowUcol correct (VOC measure): 78.117732405663%

+ global correct: 88.529502151199%

Table 6. Confusion Matrix for CNN model. Test Data. P = 24.

Matrix	**Accuracy**	**Class**
[[4613 71 69 111 15 18 133 7 8 54]	90.469%	[class: 1]
[27 3909 78 33 18 12 45 3 18 6]	94.215%	[class: 2]
[46 88 2471 22 62 26 23 49 85 10]	85.739%	[class: 3]
[125 63 68 2137 26 18 25 6 34 21]	84.701%	[class: 4]
[15 29 136 17 2061 83 13 7 18 5]	86.451%	[class: 5]
[22 15 79 25 69 1649 19 31 13 55]	83.409%	[class: 6]
[74 48 23 8 7 6 1841 0 7 5]	91.184%	[class: 7]
[15 42 112 25 19 93 8 1282 43 21]	77.229%	[class: 8]
[13 93 28 8 17 10 16 8 1362 40]	85.392%	[class: 9]
[25 21 18 8 7 37 17 6 37 1568]]	89.908%	[class: 0]

+ average row correct: 86.869733333588%

+ average rowUcol correct (VOC measure): 77.241097092628%

+ global correct: 87.94176398279%

These confusion matrices clearly demonstrate that CNN provide an excellent model, especially for the vision problem. We can see that training and testing accuracy go fairly hand in hand (as visualized in the graphs below), as opposed to the previous problem, where overfitting was evident from the first iteration itself.

Using purely supervised learning, we can achieve such high accuracy in results on a dataset of some complexity when it comes to learning via neural network models. Given enough computational power, we may be able to significantly improve on these results, with needing any more data. We note that the performance of the model decreases if the number of convolution-pooling layer is just one i.e. it is observed that the depth of the network plays in a crucial role in its learning capabilities.

5.3 Observations

Clearly, we can see the superiority of CNNs when dealing with a large number of parameters. The 'stationary' property of the images is exploited here, meaning that the statistics of one part of the image are the same as any other part. This suggests that the features that we learn at one part of the image can also be applied to other parts of the image, and we can use the same features at all locations i.e. 'convolve' them.

We believe these results can be improved upon greatly by employing greater hardware support for increased speed in training. It would also allow us to experiment with larger values of p and allow us to train for a much larger number of epochs. Results may also be improved by using unsupervised methods and we plan to do so in the future. From the results at hand, we can see that for p = 12, we achieve highest accuracy.

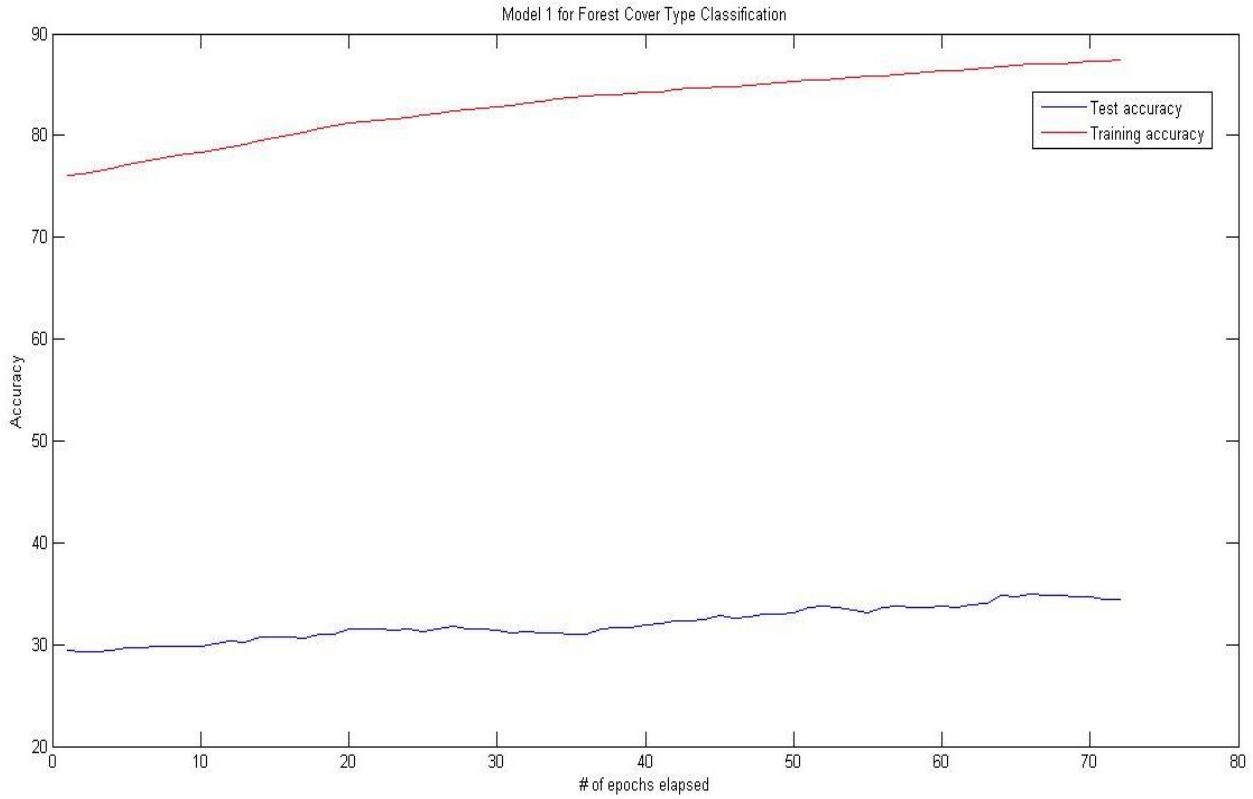


Figure 15. Accuracy versus number of epochs - FNN model 1

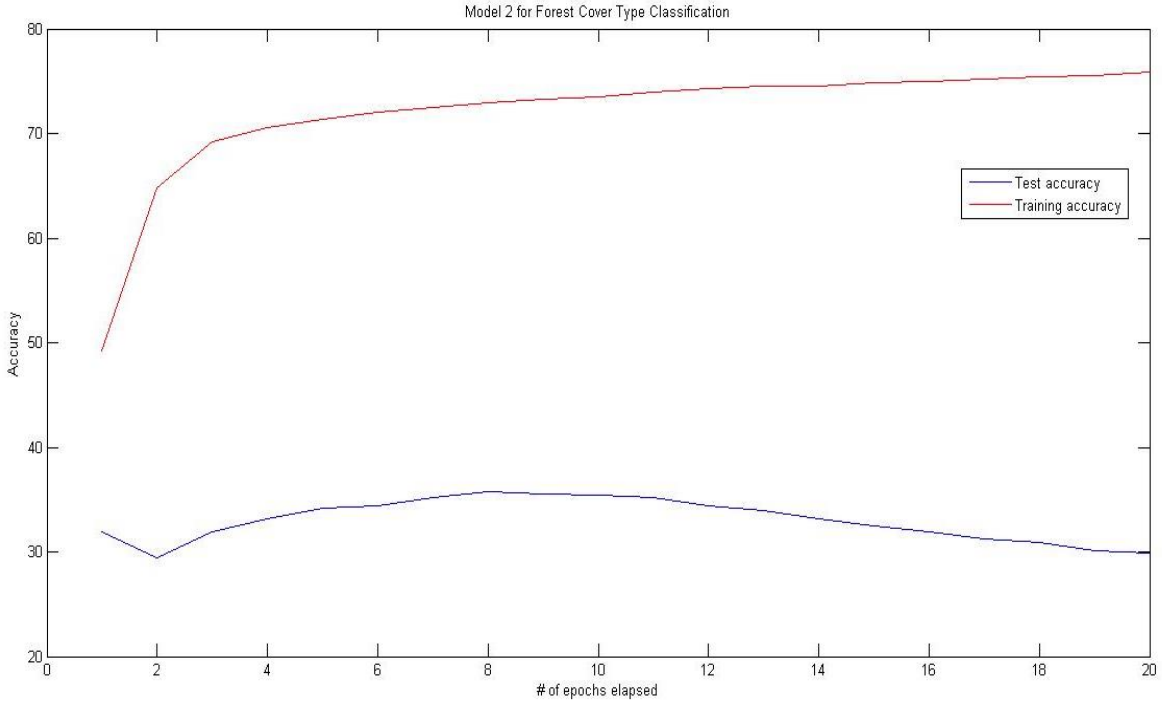


Figure 16. Accuracy versus number of epochs - FNN model 2

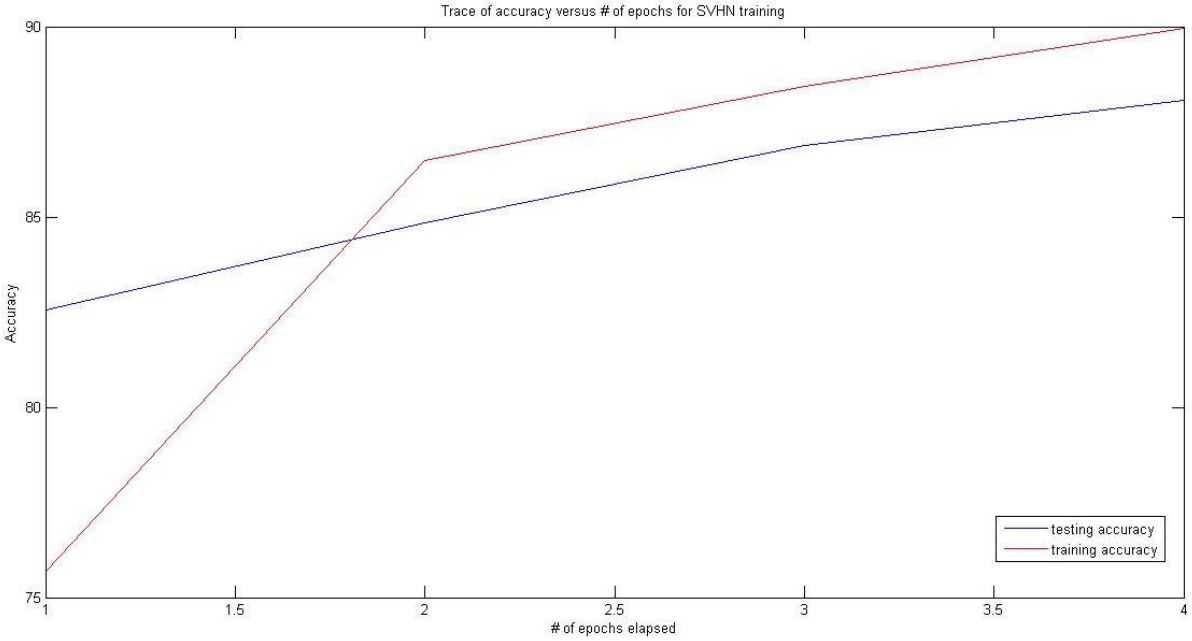


Figure 17. Accuracy versus number of epochs; p = 12; CNN model

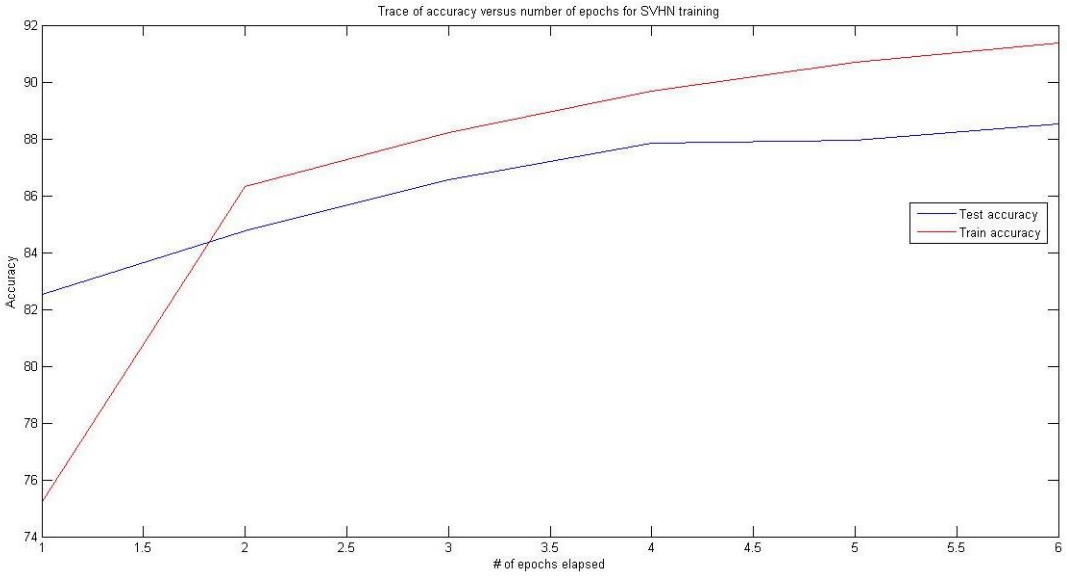


Figure 18. Accuracy versus number of epochs; p = 24; CNN model

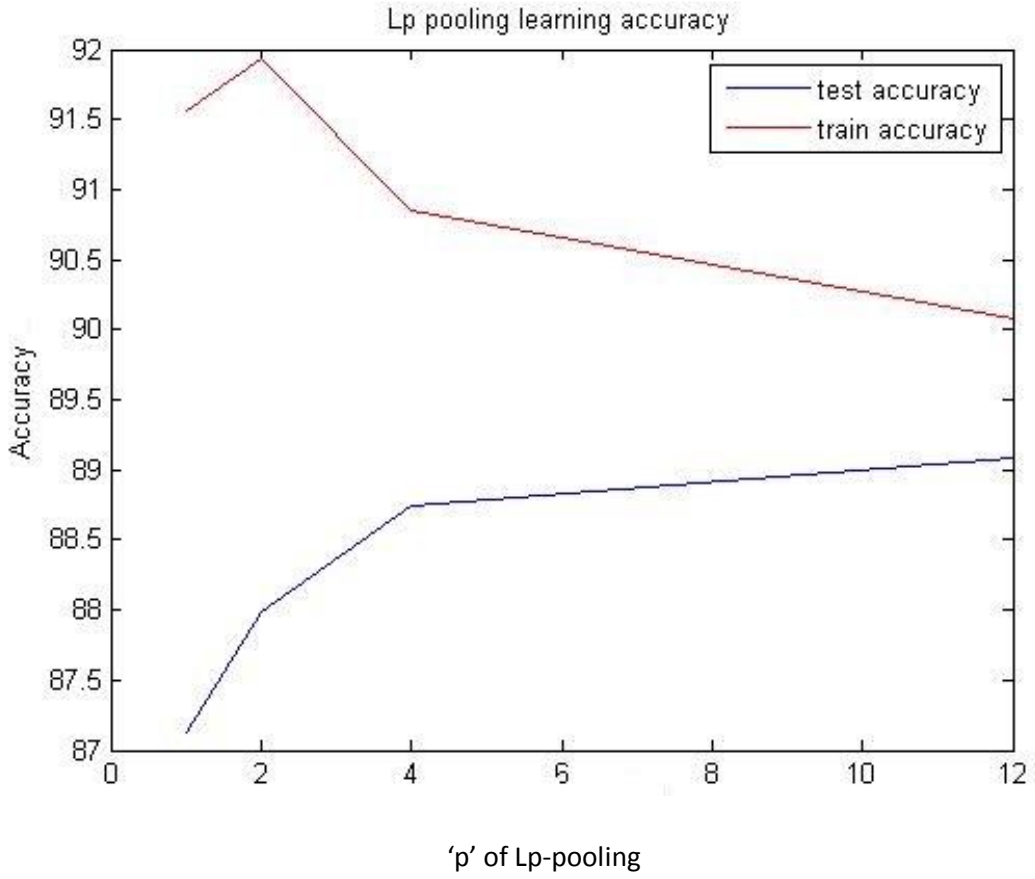


Figure 19. Training accuracy vs 'p' of Lp-pooling.

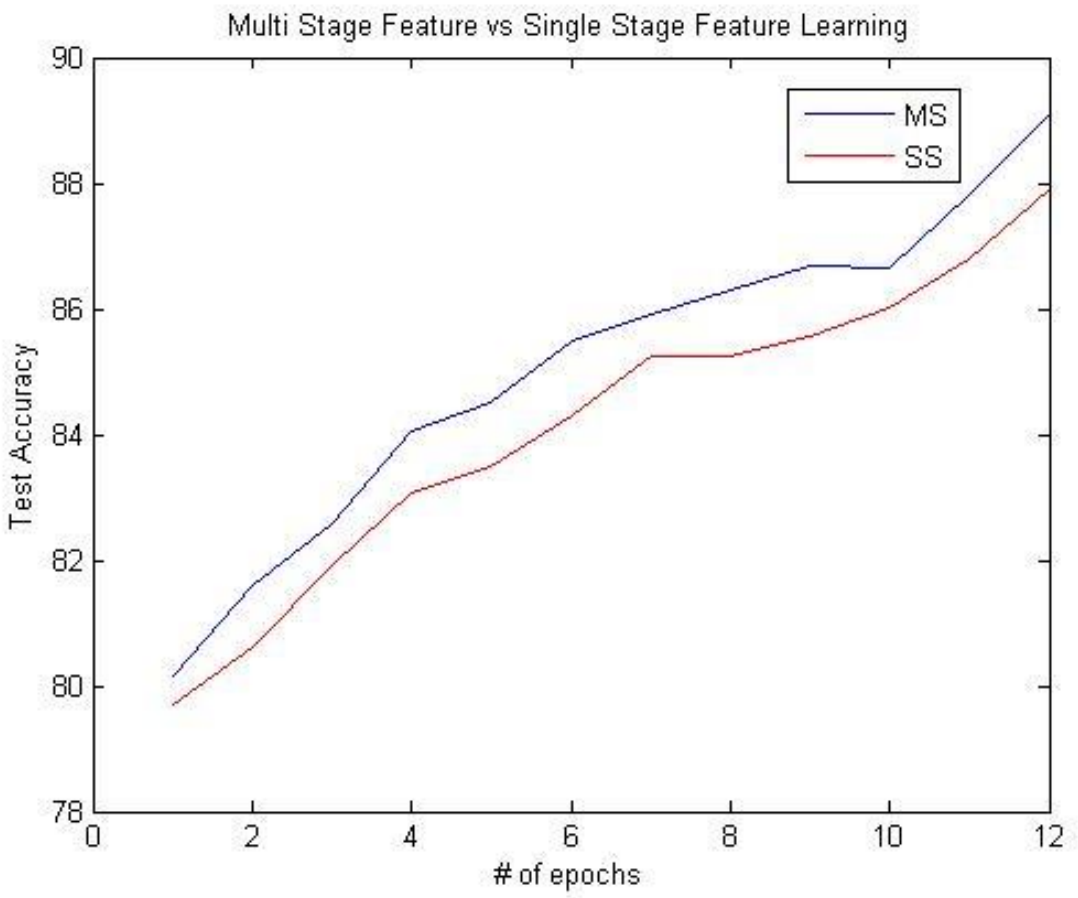


Figure 20. Improvement of Multi-Stage features (MS) over Single-Stage features (SS) in accuracy on testing set. Time taken to train over 12 epochs was 8 hours.

The figures shown above summarize the results of this project. Figures 14 and 16 depict the accuracy of the Feed Forward Neural Network model versus the number of epochs. The models differ in terms of the number of hidden units. These were the top two models. Clearly we can see that training accuracy was way more than the test accuracy throughout the training period. Figures 17 and 18 depict the accuracy of the Convolutional Neural Network. Owing to hardware limitations, we were unable to run the training for more than the indicated number of epochs. Here we observe that training and testing accuracy are almost similar, indicating that there is no overfitting and the model is able to generalize its internal representations well enough to make predictions on unseen data.

Figure 19 shows the classifier accuracy as a function of 'p'. It can be seen that for larger 'p' values, we get more accurate classifications, which is an expected result. A higher 'p' value indicates that regions of

interest in the picture are being intensified and those of low interest are being ignored. Given the limited amount of training time we could afford, and since higher 'p' values would only better results if run for larger number of epochs, results obtained for p = 24 and p = 32 have not been reported, even thought they were performed.

Figure 20 indicates that using Multi Stage Feature learning enhances the capability of the classifier. This is again expected as we use more detailed information about the image while classifying with MSF learning.

Multi Stage feature learning is an architecture of the CNN wherein certain features learned in the initial stages of the network are fed to the softmax classifier directly. The rationale for doing this is that even though the fan-in of neurons in the network is small, some features may diffuse through the network so much by the time they reach the softmax layer that they may be too faint to be detected. Therefore, we randomly select some features to be fed into the softmax directly. As seen in the graph above, this strategy clearly works. MS features have consistently improved performance in other work [30, 31, 32] and in this work as well.

5.4 Comparison with Human Performance

As opposed to the clean, isolated, quasi-binary MNIST characters, recognizing a single character of the SVHN dataset is a non-trivial task, even for trained humans. Andrew Ng et al evaluate how well such models perform in comparison to humans have sampled images from the set of images that their best models misclassified. They then evaluated the estimated accuracy of a few of the authors on the original resolution, cropped character, i.e., without the full house-number context. Below is presented their estimated human performance on the failures from the best model as a function of the image resolution. The main causes for human mistakes made in this digit recognition task were foreground/background confusion and wrong classifications due to very low resolution/blurred images. They also estimated the unconditioned probability of a human error on samples from SVHN-test and estimate the accuracy of human performance on this dataset as 98.0%.

Table 7. Human performance on misclassified instances.

HEIGHT IN PIXELS	$1-25$	$26-50$	$51-75$	$76-100$	$101-125$
HUMAN ACCURACY	$82.0 \pm 2.0\%$	$90.0 \pm 1.5\%$	$96.0 \pm 1.0\%$	100%	100%

5.5 Some Applications of training on SVHN

5.5.1 *Improving Map Services*

In addition to serving millions of users daily with panoramic imagery [33], Street View images taken worldwide are used to improve the accuracy of maps and address geocoding. House numbers detected in the Street View imagery contributes to that goal in various ways. First, detected house numbers provide us with a view angle in addition to the geocode of the address. This view angle is useful, for instance, when users search for an address in Street View, or when displaying their final destination in driving directions. Without the house number-based view angle the panorama presented will be a default one, which very likely does not point to the desired building. Thus, the user will need to turn the camera by hand in order to achieve the desired view. With the 6 house number-based view angle, the user will be led to the desired address immediately, without any further interaction needed. Second, house number recognition helps improve address geocoding. In most countries, very few addresses have associated geographic coordinates. The geocode of the majority of addresses is therefore usually computed by interpolating the geocodes of neighboring addresses. Such interpolation introduces errors which may be significant, and may result in poor user experience. With the vehicle's location, and the house number-based view angle, a better geocode of the building of interest can be computed in a straightforward way.

5.5.2 *Automatic detection and recognition of House Numbers in Street View images*

Manually detecting and transcribing house numbers in billions of available images is a daunting task. In order to automate this process, a digit recognition system [5] has been deployed within the Street View image processing pipeline. The aim is to detect and transcribe building numbers in Street View images automatically. The solution presented in [5] combines three stages: detection, recognition and verification. First the Street View panoramas are scanned by a dedicated building number detector. A sliding window classifier based on a set of elementary

features computed from image intensities and gradients is used (see [34] for details). The building number detections are then sent to the recognition module which uses the character classification models.

Input to the recognizer are the house-number patches obtained by cropping detection boxes from the full Street View images. The approach assumes that digits in these patches are aligned horizontally, with no vertical overlap between them. Although this assumption does not strictly hold for our data (see Figure 3), the approach is quite robust to small deviations and so can handle the slight in-plane rotations present in most house-number patches. Other measures are taken to handle rotations.

The recognition consists of two steps: character segmentation, where a set of candidate vertical boundaries between characters (breakpoints) is found, and character hypothesis search. Nevertheless, performance is still far from reaching human level, even though the input patches for the recognition are near optimal, as opposed to the loose detections obtained in practice from the detection phase.

Conclusion

We conclude from our experiments and the results obtained that convolutional neural networks are extremely capable of learning a large number of features and the ability to capture different features at various levels is what makes them so useful, especially in the case of images. Conventional multi-layer perceptrons are inadequate by themselves to learn a large number of parameters in the absence of large amount of data as seen, and are prone to overfitting in such cases. In the future, we would like to experiment with CNNs on more powerful hardware and train larger networks on larger training sets of images, such as ImageNet. We expect to observe high performance.

Through this project and the results obtained, it is clear that CNNs have a vastly greater learning potential than multi-layer perceptrons, due to their ability to extract features from the input and classify on the basis of these features. The learning capacity of the CNNs can be adjusted by controlling their height and width.

Lp-pooling is an effective way of pooling the features. It enhances stronger features in the image patch being learnt and diminishes weaker features, thus enabling better learning.

Learning in CNNs is much faster than MLPs, since only its last two layers are fully connected. This makes training faster and easier. It reduces over-fitting. Each layer of the CNN is learning a different feature, from simple abstract features to more complicated ones as we propagate through the network. Using Multi-Stage features is an effective way to increase performance and also to capture geo-spatial information in an image, which may be lost due to pooling.

Additionally, it is important to note that in this approach we have performed only supervised training of the CNN. The best previous methods are unsupervised learning methods (k-means, auto-encoders). In the future, we plan to perform unsupervised training to learn the features, and compare its results with this method.

References

1. © 2014 Stanford Vision Lab, Stanford University, Princeton University http://www.image-net.org/
2. B.C. Russell, A. Torralba, K.P. Murphy, and W.T. Freeman. Labelme: a database and web-based tool for image annotation. International journal of computer vision, 77(1):157–173, 2008.
3. S.C. Turaga, J.F. Murray, V. Jain, F. Roth, M. Helmstaedter, K. Briggman, W. Denk, and H.S. Seung. Convolutional networks can learn to generate affinity graphs for image segmentation. Neural Computation, 22(2):511–538, 2010.
4. Use cartographic variables to classify forest categories http://www.kaggle.com/c/forest-cover-type-prediction
5. Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, Andrew Y. Ng Reading Digits in Natural Images with Unsupervised Feature Learning *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*.
6. Implementing Neural Networks Efficiently Ronan Collobert1 , Koray Kavukcuoglu2 , and Cl´ement Farabet3,4 1 Idiap Research Institute Martigny, Switzerland 2 NEC Laboratories America Princeton, NJ, USA 3 Courant Institute of Mathematical Sciences New York University, New York, NY, USA 4 Universit´e Paris-Est Equipe A3SI - ESIEE Paris, France http://pub.clement.farabet.net/tricksofthetrade12.pdf
7. http://ufldl.stanford.edu/wiki/index.php?title=SoftmaxRegression&oldid=2288
8. Convolutional Neural Networks Applied to House Numbers Digit Classification. Pierre Sermanet, Soumith Chintala and Yann LeCun, The Courant Institute of Mathematical sciences. New York University.{sermanet,soumith,yann}@cs.nyu.edu. http://arxiv.org/abs/1204.3968v1
9. http://ufldl.stanford.edu/tutorial/supervised/ConvolutionalNeuralNetwork/
10. Learning Deep Architectures for AI By Yoshua Bengio; http://www.iro.umontreal.ca/~bengioy/papers/ftml_book.pdf
11. T. Serre, G. Kreiman, M. Kouh, C. Cadieu, U. Knoblich, and T. Poggio, “A quantitative theory of immediate visual recognition,” Progress in Brain Research, Computational Neuroscience: Theoretical Insights into Brain Function, vol. 165, pp. 33–56, 2007.
12. A. Ahmed, K. Yu, W. Xu, Y. Gong, and E. P. Xing, “Training hierarchical feed-forward visual recognition models using transfer learning from pseudo tasks,” in Proceedings of the 10th European Conference on Computer Vision (ECCV’08), pp. 69–82, 2008.

13. Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, “Greedy layer-wise training of deep networks,” in Advances in Neural Information Processing Systems 19 (NIPS’06), (B. Sch¨olkopf, J. Platt, and T. Hoffman, eds.), pp. 153– 160, MIT Press, 2007.
14. H. Larochelle, D. Erhan, A. Courville, J. Bergstra, and Y. Bengio, “An empirical evaluation of deep architectures on problems with many factors of variation,” in Proceedings of the Twenty-fourth International Conference on Machine Learning (ICML’07), (Z. Ghahramani, ed.), pp. 473–480, ACM, 2007.
15. H. Schwenk and M. Milgram, “Transformation invariant autoassociation with application to handwritten character recognition,” in Advances in Neural References Information Processing Systems 7 (NIPS’94), (G. Tesauro, D. Touretzky, and T. Leen, eds.), pp. 991–998, MIT Press, 1995.
16. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” Nature, vol. 323, pp. 533–536, 1986.
17. R. Salakhutdinov and G. E. Hinton, “Using deep belief nets to learn covariance kernels for Gaussian processes,” in Advances in Neural Information Processing Systems 20 (NIPS’07), (J. Platt, D. Koller, Y. Singer, and S. Roweis, eds.), pp. 1249–1256, Cambridge, MA: MIT Press, 2008.
18. R. Salakhutdinov and G. E. Hinton, “Learning a nonlinear embedding by preserving class neighbourhood structure,” in Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS’07), San Juan, Porto Rico: Omnipress, 2007.
19. G. E. Hinton and R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” Science, vol. 313, no. 5786, pp. 504–507, 2006
20. S. Osindero and G. E. Hinton, “Modeling image patches with a directed hierarchy of Markov random field,” in Advances in Neural Information Processing Systems 20 (NIPS’07), (J. Platt, D. Koller, Y. Singer, and S. Roweis, eds.), pp. 1121–1128, Cambridge, MA: MIT Press, 2008.
21. R. Sutton and A. Barto, Reinforcement Learning: An Introduction. MIT Press, 1998.
22. G. Taylor and G. Hinton, “Factored conditional restricted Boltzmann machines for modeling motion style,” in Proceedings of the 26th International Conference on Machine Learning (ICML’09), (L. Bottou and M. Littman, eds.), pp. 1025–1032, Montreal: Omnipress, June 2009
23. A. Torralba, R. Fergus, and Y. Weiss, “Small codes and large databases for recognition,” in Proceedings of the Computer Vision and Pattern Recognition Conference (CVPR’08), pp. 1–8, 2008
24. R. Salakhutdinov and G. E. Hinton, “Using deep belief nets to learn covariance kernels for Gaussian processes,” in Advances in Neural Information Processing Systems 20 (NIPS’07), (J. Platt, D. Koller, Y. Singer, and S. Roweis, eds.), pp. 1249–1256, Cambridge, MA: MIT Press, 2008.

25. I. Levner, Data Driven Object Segmentation. 2008. PhD thesis, Department of Computer Science, University of Alberta
26. A. Mnih and G. E. Hinton, “A scalable hierarchical distributed language model,” in Advances in Neural Information Processing Systems 21 (NIPS’08), (D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, eds.), pp. 1081–1088, 2009.
27. Handwritten Digit Classification; Subhransu Maji and Jitendra Malik; http://ttic.uchicago.edu/~smaji/projects/digits/
28. http://torch7.s3-website-us-east-1.amazonaws.com/data/housenumbers/
29. http://en.wikipedia.org/wiki/YUV
30. J. Fan, W. Xu, Y. Wu, and Y. Gong. Human tracking using convolutional neural networks. Neural Networks, IEEE Transactions on, 21(10):1610 –1623, 2010.
31. P. Sermanet, K. Kavukcuoglu, and Y. LeCun. Traffic signs and pedestrians vision with multi-scale convolutional networks. In Snowbird Machine Learning Workshop, 2011.
32. P. Sermanet and Y. LeCun. Traffic sign recognition with multi-scale convolutional networks. In Proceedings of International Joint Conference on Neural Networks, 2011.
33. D. Anguelov, C. Dulong, D. Filip, C. Frueh, S. Lafon, R. Lyon, A. Ogale, L. Vincent, and J. Weaver. Google street view: Capturing the world at street level. Computer, 43(6):32–38, June 2010.
34. A. Frome, G. Cheung, A. Abdulkader, M. Zennaro, B. Wu, A. Bissacco, H. Adam, H. Neven, and L. Vincent. Large-scale privacy protection in google street view. In ICCV, pages 2373– 2380. IEEE, 2009.