CS280 - Computer Vision - Spring 2015
Homework 2

1 Digit Recognition using Hand Designed Features

1.1 Warming up

In this assignment we will do digit recognition, using the original MNIST dataset [1]. In the first part, we will try to reproduce many of the results presented in [2]. The state-of-the-art error rate on this dataset is roughly 0.5%; the methods in [2] will get you to 1.1% (modulo implementation details). In the second part, we will use Caffe and explore how deep learning performs.

Before starting, read Appendix A. To avoid computational expense, we will only train on a subset of the data. (Once you have finished this assignment, if you have enough compute power at your disposal, you can try your code on the full dataset.) The file `data/train_small.mat` contains 7 different subsets, containing 100, 200, 500, 1000, 2000, 5000 and 10000 training points respectively. The full training set containing 60000 images is in `data/train.mat`. The test set is in `data/test.mat`.

Q1 As a warm-up, use the raw pixels as features to train a linear svm. Plot the error rate vs the number of training examples.

1.2 Spatial pyramids

The performance can be improved if we consider more sophisticated features rather than raw pixel values. [2] consider a feature where the image is overlaid with a grid in which each cell is of size $c \times c$ and the values inside each cell are added up to produce one feature per cell. [2] also suggest using multiple overlapping grids that are offset by $\frac{c}{2}$ instead of a single grid. We will construct two sets of grids, one with a cell size of 4 pixels and another with a cell size of 7 pixels. The final feature vector will be the raw pixel values, together with the features from all the cells concatenated together.

Q2 (a) Explain briefly why adding these features should help over raw pixel values.

(b) Implement these features and use them to train a linear svm. Plot the error rate vs the number of training examples. Do you get a significant boost over Q1?

1.3 Orientation histograms

To get to state-of-the-art performance however, we will need to go beyond pixel intensity. We will use a variant of PHOG(Pyramidal Histogram of Oriented Gradients, see section 4.1 in [3]). To construct the PHOG feature vector, we first compute the gradient magnitude and orientation at each pixel in the image. Then we construct "pyramid features" similar to Q2. However, instead of summing the pixel intensities in each cell we will construct a histogram of orientations in each cell. Orientation histograms typically have between 8 to 10 bins.

The final feature vector is the concatenations of all the histograms from all the cells. Typically the histograms are normalized (so that the bin counts sum to 1) before being concatenated.

Q3 (a) Explain briefly why gradient orientations should help over pixel intensities : what is the gradient orientation trying to capture?

(b) Implement these features and use them to train a linear SVM. For computing the gradient, use the tap filter:$[-1\ 0\ 1]$. Use 9 orientation bins. Use the same cell sizes (4 and 7) as in Q2. Plot the error rate vs the number of training examples. Do you get a significant boost over Q2?

(c) Does the performance drop if you don't normalize the histograms before concatenating them? Why or why not? Plot error rate vs the number of training examples.

(d) How does performance change if you replace the tap filter with a Gaussian derivative filter?

(e) Explain how you choose the hyper-parameter in linear SVM and why.

1.4 Visualizing errors

Q4 Visualize the images on which you go wrong. Do the errors your classifier makes seem reasonable to you? How many of these images do *you* as a human have a hard time recognizing?

1.5 (Optional) Using kernel SVM

Q5 (a) Use kernel SVM (for instance, RBF kernel) with the features obtained in Q3. What is the performance?

(b) Explain how you choose the kernel and kernel parameters.

1.6 Conclusion

The system you have built now should be almost on par with the state of the art as far as features are concerned. You can try training on the full training set to get an idea of how good you do. For example, an intersection kernel SVM [3] could give you better performance.

2 Digit Recognition using Learned Features

Neural Networks form the core of leading techniques of object recognition. In this homework, we will train our own neural networks using Caffe. It is recommended that you read the Caffe tutorial prior to caffenating your neural networks.

All neural networks consist of input layer, hidden layers and output layer. There are 10 types of digits in MNIST and thus our output layer will consist of 10 units. We will use the Soft-Max loss. We will experiment with the hidden layers, choice of non-linearity and some other aspects of neural networks.

2.1 Fully Connected Network

The simplest way to use a neural network for object recognition is by treating the full image as a vector. For example, the digit images of size 28×28 can be represented as a 784-D vector. We will use two hidden layer consisting of 20, 500 units each. Each unit a layer layer is connected to all inputs in the previous layer. Such a layer is called Fully Connected or Inner Product layer. Between each pair of hidden layers - you should use the ReLU (Rectified Linear Unit) non-linearity.

2.1.1 To Do:

1. Draw a schematic of the neural network architecture described above. Clearly label all the layers of the network (including the non-linearity).
2. Train this network for digit classification. Report the accuracy on the test set. How many parameters does this network have?
3. Use three hidden layers of 20, 50 and 500 units respectively. Report the accuracy on the test set. How many parameters does this network have?
4. Compare the performance of both these networks. Why is the performance greater/lower or the same?
5. Change the non-linearity to Sigmoid and re-train the network. Do you observe any differences? If the depth of the network is increased - what effect would the Sigmoid non-linearity have on the magnitude of the gradients? Is this desirable? Is the behavior of ReLu units different?

2.2 Convolutional Neural Network

There are two properties of the visual world that we can exploit:

1. Irrespective of location of the objects in the image, the visual appearance of objects remains the same.
2. Objects can be thought of to be composed of parts. Therefore instead of modeling the entire image, we can model smaller parts of the object.

These two ideas and findings from neuroscience inspired Convolutional Neural Networks (ConvNets). The ConvNets were proposed by Yann LeCun in 1989 for classifying digits. This specific architecture proposed in the 1989 paper is referred as the LeNet. We will train the LeNet architecture for digit classification.

2.2.1 To Do:

1. Report the performance of the LeNet architecture. How many parameters does this network have? Train the LeNet using only 10K examples and compare the performance with the SVM classifier you trained in the first part of the homework.
2. Both LeNet and FC network have 20, 50, 500 units in the three hidden layers. Considering this, compare the performance of LeNet with the Fully connected(FC) network with three hidden layers.
3. Now, reduce the number of units in the third hidden layer of the LeNet to 32. Lets call this LeNet2. Compare the performance with the fully connected network. Why is the performance greater/lower or the same?
4. Experiment with the kernel sizes in LeNet2. Try kernel sizes of 3 and 7. What do you find?
5. Visualize the first layer filters of the original LeNet network. What do you find? Based on this visualization - can you comment on what computation first layer is performing?
6. Based on your results, list the pros and cons of using a FC network in comparison to the ConvNet. Is the ConvNet more scalable to larger images? If yes, why and if no, why not?

2.3 Extra Credit

Visualize the digits that were incorrectly predicted by the LeNet. Compare these with digits incorrectly predicted by the SVM. Do you notice any differences in the errors made by these two algorithms?

2.4 Additional Questions(Optional)

These questions are not required and will not be graded. If you want to play around with Neural Networks - you can consider these:

1. Is Deeper always better? - Is a narrower and deeper model better than a broader and a more shallower model? Change the number of layers while keeping the number of parameters fixed. What do you find? What happens when you have less training data?
2. Problem of sluggish training: Plot the error as a function of number of training iterations. You will observe that the error reduces drastically initially and then the decrease becomes more sluggish. Can you think of any way to overcome this problem?
3. Data Augmentation: Augment the training set, by translating and rotating the digits. Does this improve performance? Add gaussian random noise to training examples. How does this effect performance?
4. Dropout: Dropout [4] has been proposed a method of preventing overfitting. The way dropout work is - that in every iteration of the training phase, the activation of a randomly chosen subset of units in a given layer is set to zero. The fraction of units that are chosen for dropout is set to a constant. Lets call this the dropout rate. This scheme of regularization can be interpreted as an instance of model averaging [4]. Perform the following experiment: consider only 5000 training examples. Implement dropout after the ReLu layer following the 500 unit fully connected layer in the LeNet. Does adding dropout improve performance?
5. Understanding Dropouts: Experiment with different dropout rates. Study the relationship between the sparsity of features and the amount of dropouts. What do you observe?

6. Finding the location of the digits: Can you devise a method to find where the digit is? The digit can be localized by predicting the (x,y,w,h) coordinates - where x,y denote a point in the image and w, h are the width and height of the rectangle which contain the digit. In MNIST most digits are centered - which is not very interesting. Generate a few testing examples - by randomly shifting the location of digits. Visually evaluate your method.

7. Visualizing the embedding of digits: The third hidden layer of ConvNets represents each digit in 500-D space. Visualize this 500-D space in 2-D by doing Principal Component Analysis (PCA). Do you find any interesting pattern.

3 Thinking about projects (Required)

Is there a cool application/problem you have in mind that requires object recognition? If not, think of one. Write a short description (no more than one paragraph) of this application/problem. Even if you have a crazy idea write it down. We can later reduce it to something that is doable. Your final project does not needs to be what you are thinking of right now!

4 Instructions

1. This HW can be done in groups of 2. We only need 1 submission per group.

2. Please submit through bcourse (under HW2). In the text box, please put in your name, email address and student ID; and also your partner's if you worked with a partner. Please upload only the following:

   (a) `A single PDF file`, containing your answers to all questions (and all supporting images and plots). This is the only thing that we will look at for grading purposes.

3. The HW is due on March 9, 2015, at 11:55 PM.

A Guidelines for running SVM

Before you begin, you need to make sure everything compiles well. You will need MATLAB, and either a Mac or Linux. If you are in Windows, you will need a version of `make`. Most files are just MATLAB files, so no compilation is required. However, you need to compile the linear SVM package that you will download from `http://www.csie.ntu.edu.tw/~cjlin/liblinear/`. Place the directory you download from `LIBLINEAR` into the code subdirectory and cd into it, e.g cd to`code/liblinear-x`, then type `make`. Then cd into `code/liblinear-x/matlab`. Edit the Makefile to point to your local copy of MATLAB, and then type `make`. For further information, please read the `README` file of the package.

For kernel SVM, you need to download from `http://www.csie.ntu.edu.tw/~cjlin/libsvm/` and the installation of `LIBSVM` is very similar to that of `LIBLINEAR`.

We provide two functions for you. `code/benchmark.m` will take in the predicted labels and true labels and return the error rate, and also the indices of the labels that are wrong. `code/montage_images.m` will produce a montage of a set of images: use this, for instance, to visualize the images on which you make errors.

B Guidelines for using Neural Networks

There are three steps involved in using Caffe for training neural networks:

1. Preparing the data in a suitable format.

2. Defining the architecture of the network.

3. Defining the solver file.

Some parts have been detailed below, and other parts you can find at the Caffe website.

B.1 Preparing your Data

The data has been split into two sets - train and test. We will use the train set for training and test set for testing. In practice, the data is split into 3 sets - train, validation and test. The network is trained on the train set and the performance is evaluated on the validation set. This process is repeated for different values of parameters that can influence the performance of the learning algorithm (for eg, the C variable in SVM or the choice of non-linearity, number of layers etc in a NNet). The optimal parameters are chosen by evaluating the performance on the validation set. Using this set of parameters, the learning algorithm is trained (on train + validation set) and finally evaluated on the test set. This is a fair way of measuring performance. For the sake of simplicity, in this homework we will skip constructing the validation set and only work with the train and test set.

You are free to work with any form of inputting data into Caffe (Caffe Data Formats). The MNIST dataset can be downloaded from [1]. We have processed the data into HDF5 format which can downloaded from bCourses. Depending on how much computational power you have, feel free to use 10K, 20K or all 60K examples for training your network. The syntax of the HDF5 layer reads like:

```
layer {
  name: "mnist"
  type: "HDF5Data"
  top: "data"
  top: "label"
  include {
    phase: TRAIN
  }
  hdf5_data_param {
    source: "/path/to/h5source.txt"
    batch_size: 64
  }
}
```

The name of the this layer is "mnist", it produces two outputs named "data" and "label". This layer is used in the phase TRAIN and reads data from "h5source.txt". The file "h5source.txt" contains a list of HDF5 files, which are read sequentially by the HDF5Data layer. Each HDF5 file, listed in "h5source.txt" contains data for one batch (i.e. 64 images and 64 labels in this example). Each such HDF5 file has two dataspaces - "data" and "label", which are read during the forward pass of the HDF5Data layer. For your reference, a sample "h5source.txt" file has been provided with the started code. If you directly use "h5source.txt" provided with the started code - you will end up using the entire MNIST training dataset. If due to computational constraints, you want to use less number of training images - you can create a new file, which contains a subset of lines contained in the file "h5source.txt"

B.2 LeNet Architecture

The LeNet architecture is provided in the file lenet_train_test.prototxt.

B.3 Testing your Neural Network

You can test your neural network in multiple ways. The most straightforward way is to use the command:

```
caffe test --model=ArchitectureFileName --weights=ModelFileName
```

The ArchitectureFileName is the prototxt file which defines the architecture of the network and ModelFileName is the name of the file which stores the learned weights (It will generally have the prefix .caffemodel).

You may also want to consider using the python or matlab wrappers to compute performance. PyCaffe (Python wrapper of caffe) is documented quite well and you may want to check out examples on the Caffe webpage.

B.4 Visualizing the Weights

Refer to: `http://nbviewer.ipython.org/github/BVLC/caffe/blob/master/examples/filter_visualization.ipynb`

References

[1] `http://yann.lecun.com/exdb/mnist/`

[2] S. Maji and J. Malik. Fast and Accurate Digit Classification. Technical Report, EECS Berkeley. `http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-159.pdf`.

[3] S. Maji, A. Berg and J.Malik. Classification using Intersection Kernel Support Vector Machines is Efficient. In *CVPR*, 2008.

[4] Geoffrey E. Hinton and Nitish Srivastava and Alex Krizhevsky and Ilya Sutskever and Ruslan Salakhutdinov, Improving neural networks by preventing co-adaptation of feature detectors, 2012, http://arxiv.org/abs/1207.0580,