Computer Vision – 67542

MATLAB Assignment 4 – Convolutional Neural Networks

Preliminary Instructions

General

- Submission deadline: 20.6.2016.
- Late submission is allowed, but every day past the deadline will reduce 5 points from the assignment grade.
- The assignment is individual (i.e. you are not allowed to work in pairs or groups).
- Submit your solution through the course website. All of your files should be zipped in a folder named *matlab4_[ID].zip* ([ID] standing for your 9-digit ID).
- Your submission should include all .m files used during the assignment. Provide a brief description of each code file in a text file named *readme.txt*.
- Your code should be supplemented by a PDF file named *report.pdf*, which will include documentation of your results.
- The assignment consists of several mandatory parts and a bonus part. The number of points a part is worth is listed beside it.
- The assignment uses a rather small dataset (see below) and we will often train with relatively few iterations. As a consequence, resulting train and test errors may vary considerably depending on random factors in training (such as initialization or SGD batches). Therefore, when specifying train/test errors, you are required to **run the same setting 5 times and report the average** across these runs.
- **Good luck!**

Dataset

In this assignment we will work on a subset of MNIST database. MNIST consists of 28 by 28 grayscale images representing handwritten digits (0 through 9). It has 60K images for training and 10K images for testing, both distributed evenly between the 10 classes. MNIST is considered to be a trivial benchmark (for example, applying SVM with Gaussian kernel on the image pixels gives an accuracy of almost 99%). To make the learning task a bit more challenging (and reduce training run-times), we will work with a subset of 2.5K training images selected arbitrarily from the full training set. The test set will be used in full.

In the folder *data* you will find four binary files holding the images and labels of the train and test sets. For your convenience, a script file named *data_disp_script.m* is provided in the root folder of the assignment. This script loads and displays a few images along with their labels, thereby specifying the file formats.

EasyConvNet Package

The assignment is based on a convolutional neural network (ConvNet) package named EasyConvNet, written by Shai Shalev-Shwartz. This package provides a very simple and modular MATLAB implementation of ConvNets, which is reasonably fast, and even supports GPU execution. It includes the following files, located in the root folder of the assignment:

File Name	Description
dataClass.m	Class for loading data from file (network input)
lossClass.m	Class implementing different loss functions (network output)
ConvNet.m	Main file of the package. Class that implements a sequence of layers, which together form a ConvNet.
visualizeNetwork.m	Function that receives a network and produces LaTeX code visualizing its connections
demoMNIST.m	Simple file demonstrating the usage of the package for MNIST classification
LICENSE	License information
README.md	Very basic description of the package

Table 1 - EasyConvNet files

You are welcome to inspect the package files, and run *demoMNIST.m* (which in our case will use only 2.5K training images).

Template Script

The file *template_script.m*, located in the root folder of the assignment, contains a template for constructing, training and evaluating a ConvNet. Your solutions to the different parts of the assignment may be based on this template.

The ConvNet architecture and training scheme implemented by *template_script.m* are very simplistic, and will evolve as you progress throughout the assignment. In the outset, as you may see in the script, the network consists of a single convolutional layer with 5 kernels of size 5x5 (stride 1), followed by ReLU activation and 2x2 max-pooling (no overlap), which in turn is followed by a dense affine layer with 10 (number of classes) outputs. For training, SGD with Nesterov momentum is employed, but the momentum variable μ is set to 0, so in effect the optimization is no more than vanilla SGD. The batch size is 1 and the number of iterations is

10K, giving 4 epochs (rounds over training set) in total. The L2 regularization parameter λ is set to 0, and the learning rate η is fixed at 0.05.

You may start by running *template_script.m*, and observing the obtained train and test accuracies. Besides computing and printing these, the script also displays a sample of misclassified test images, along with their true and predicted labels. Notice how some of these images are confusing even for a human, whereas others seem more definite.

Part 1 (10 pts)

In this part we will investigate the effect of SGD batch size on optimization convergence rate, i.e. on the **train** error achieved per given number of examples passed through the network in training (with repetitions). Specifically, we will create a script named *batch_size_script.m* (based on *template_script.m*), running with batch sizes of 1, 2, 4, 8 and 20, in each case setting the number of iterations to cover exactly 4 epochs (e.g. for batch size 20 we will have 500 iterations). Since larger batch sizes provide more accurate gradient estimations, they enjoy the possibility of a larger learning rate η. We will thus try, for each batch size independently, learning rates of 0.5, 0.1 and 0.05, and report the best found train error. Produce a plot of these train errors as a function of the batch size and attach it to your report. What is the optimal batch size? Explain the tradeoff that comes into play here. How do you expect the complexity of the dataset (image size, number of classes, intra-class appearance variability etc.) to affect the optimal batch size?

Part 2 (10 pts)

In this part we investigate the effect of momentum on training convergence. Starting again from *template_script.m*, prepare a script named *momentum_script.m*, in which values of 0.9, 0.95 and 0.99 will be tried for the momentum variable μ. For each case independently, choose a learning rate η out of {5e-2, 1e-2, 5e-3, 1e-3, 5e-4, 1e-4} that gives the lowest train error. Add to your report the train error and best learning rate for each momentum value. Can you explain why higher momentum values require lower learning rates?

Part 3 (10 pts)

With *template_script.m* as starting point, prepare a script named *lambda_script.m* in which λ - the L2 regularization parameter, takes on the values 0, 1e-4, 5e-4, 1e-3, 5e-3, 1e-2 and 5e-2. Plot the resulting train and test errors as a function of λ. How does this plot compare to those you are used to seeing in classical machine learning?

Part 4 (5 pts)

Try replacing 'Xavier' initialization of affine weights in *template_script.m* with 'Zeros'. What happened? Does this phenomenon occur in classical machine learning as well?

Part 5 (5 pts)

Modify the learning rate η in *template_script.m* to decrease by a factor of 10 after 3 epochs (7.5K iterations). How does this affect the training error? Can you provide an intuitive explanation for this?

Part 6 (20 pts)

The objective in this part is to reach the best possible **test** accuracy, with the network architecture defined in *template_script.m*. Specifically, prepare a script named *best_small_script.m*, that is based on *template_script.m*, and in which the training hyper-parameters below are configured to give optimal test accuracy:

- SGD batch size
- Number of iterations T
- Momentum variable μ
- Learning rate η (may change during training)
- L2 regularization parameter λ

Add to your report the selected configuration for these hyper-parameters, as well as the resulting test error and its corresponding train error.

Part 7 (10 pts)

Prepare a script named *small_large_script.m*, that applies the following training hyper-parameters:

- SGD batch size: 10
- Number of iterations T : 10K
- Momentum variable μ : 0.9
- Learning rate η : 0.005
- L2 regularization parameter λ : 0.0005

to the ConvNet we have worked with thus far, and to a much larger ConvNet consisting of the following layers:

- Convolutional layer with 20 kernels of size 5x5 (stride 1)
- ReLU activation
- Max-pooling with 2x2 windows (no overlap)

- Convolutional layer with 50 kernels of size 5x5 (stride 1)
- ReLU activation
- Max-pooling with 2x2 windows (no overlap)
- Dense affine layer with 500 output nodes
- ReLU activation
- Dense affine layer with 10 (number of classes) output nodes

What happened to the train and test errors? How does this relate to the well-known bias-variance tradeoff of classical machine learning?

Part 8 (30 pts)

Your mission in this part is to reach the best possible test accuracy, while tuning both training hyper-parameters and network architecture. Prepare a script named *best_script.m*, in which you construct the best ConvNet you have found, train it using the optimal training scheme, and measure its train and test errors. Add to your report the details of the selected network architecture and training scheme, as well as the resulting train and test errors.

There are many possible directions to explore in search of the optimal network architecture. To name just a few, you are recommended to try the following:

- Use of average-pooling instead or in addition to max-pooling
- Increasing the number of convolution-ReLU-pooling and/or dense affine layers
- Incorporating overlap into pooling operations
- Increasing the number of kernels in convolutional layers
- Increasing the number of outputs in dense affine layers
- Increasing/decreasing kernel size in convolutional layers
- Migrating to Network in Network type architecture (1x1 convolutions, global average pooling instead of dense affine layers)

Bonus Part (25 pts)

As you probably have noticed in Part 8 (30 pts), the main challenge one has to cope with for improving test accuracy is overfit. This situation is very common in deep learning, and is addressed with various forms of regularization. One form, which has proven to be very effective for neural networks, is called *dropout*. In dropout, neurons are randomly zeroed out in each forward-backward pass taking place during training. At test time, all neurons are active, but are scaled to account for the excessive activation compared to training. For example, if the probability of a neuron being zeroed out in training is p (a new draw is made for every example passed through the network), its output is multiplied by $1-p$ at test time.

The purpose of this bonus part is to further improve test accuracy using dropout. For this sake, review the paper "Dropout: A simple way to prevent neural networks from overfitting", integrate dropout into EasyConvNet package as a new type of layer (it is currently not supported), and repeat Part 8 (30 pts) while using this powerful regularizer. Include in your submission the modified EasyConvNet code files, as well as a script named *dropout_script.m*, which constructs, trains and evaluates the optimal classifier found. Add to your report the architectural and training details of this classifier, the train and test errors it produces, and any interesting conclusions you have reached while tackling this task.